

Vue (<https://www.processon.com/view/link/5da6c108e4b002a6448895c3#map>)

vue2的数据绑定如何实现的？

- 1.vue会监视data中所有层次的属性
- 2.对象属性数据通过添加set方法来实现监视
- 3.数组元素也实现了监视：重写数组一系列更新元素的方法
 - 1) 调用原生对应的方法对元素进行处理
 - 2) 去更新界面

Vue 数据双向绑定主要是指：数据变化更新视图，视图变化更新数据。其中，View变化更新Data，可以通过事件监听的方式来实现，所以Vue数据双向绑定的工作主要是如何根据Data变化更新View。

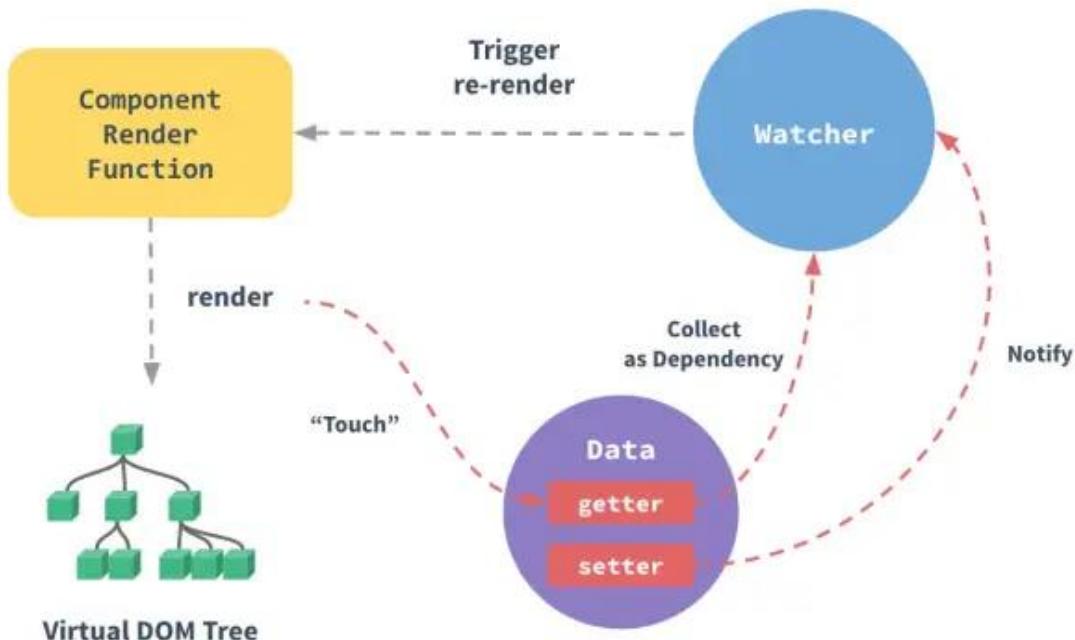
简述：

当你把一个普通的 JavaScript 对象传入 Vue 实例作为 data 选项，Vue 将遍历此对象所有的 property，并使用 Object.defineProperty 把这些 property 全部转为 getter/setter。

这些 getter/setter 对用户来说是不可见的，但是在内部它们让 Vue 能够追踪依赖，在 property 被访问和修改时通知变更。

每个组件实例都对应一个 watcher 实例，它会在组件渲染的过程中把“接触”过的数据 property 记录为依赖。

之后当依赖项的 setter 触发时，会通知 watcher，从而使它关联的组件重新渲染。



深入理解：

监听器 Observer：对数据对象进行遍历，包括子属性对象的属性，利用 Object.defineProperty() 对属性都加上 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。

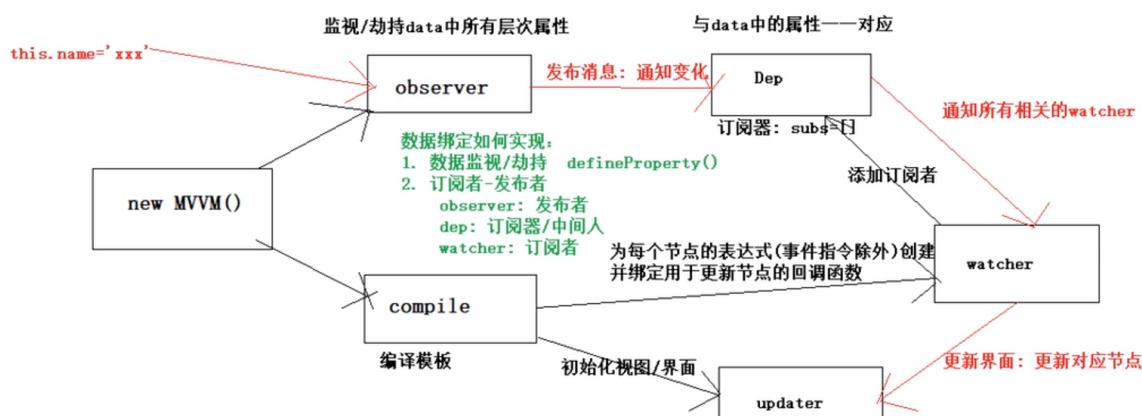
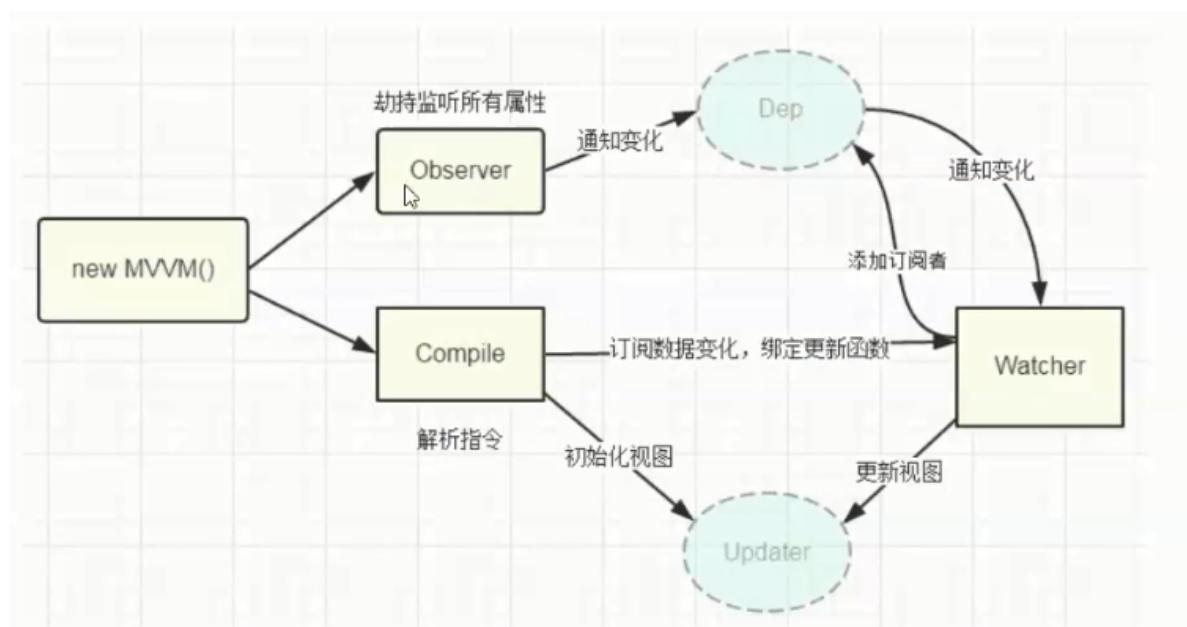
解析器 Compile：解析 Vue 模板指令，将模板中的变量都替换成数据，然后初始化渲染页面视图，并将每个指令对应的节点绑定更新函数，添加监听数据的订阅者，一旦数据有变动，收到通知，调用更新函数进行数据更新。

订阅者 Watcher：Watcher 订阅者是 Observer 和 Compile 之间通信的桥梁，主要的任务是订阅 Observer 中的属性值变化的消息，当收到属性值变化的消息时，触发解析器 Compile 中对应的更新函数。

每个组件实例都有相应的 watcher 实例对象，它会在组件渲染的过程中把属性记录为依赖，之后当依赖项的 setter 被调用时，会通知 watcher 重新计算，从而致使它关联的组件得以更新——这是一个典型的观察者模式

订阅器 Dep：订阅器采用发布-订阅设计模式，用来收集订阅者 Watcher，对监听器 Observer 和订阅者 Watcher 进行统一管理。

数据响应的原理



dep对象

什么时候创建？ 初始化=>在给data中的属性添加监视/劫持(setter)创建

创建几个？ 与data中属性一一对应，就是data中属性的个数

watcher对象

什么时候创建？ 初始化=>在对模板中某个节点(包含模板语法)实现内存初始化更新节点后创建
创建几个？与模板中表达式(插值/一般指令)一一对应，就是模板中表达式的个数

dep与watcher关系

dep=>watcher 一对多/1:n， 当前属性被模板中的多个表达式使用了

watcher=>dep 一对多/1:n， 当前表达式是一个多层的表达式

发布者-订阅者模式

发布者:observer(包含监视data数据变化的setter)

订阅者:watcher(包含了更新对应节点的回调函数)

订阅器:dep中间人(被对应属性的setter引用，与watcher建立了n:n的关系)

VUE3.X响应式数据原理

Vue3.x改用Proxy替代Object.defineProperty。

因为Proxy可以直接监听对象和数组的变化，并且有多达13种拦截方法。并且作为新标准将受到浏览器厂商重点持续的性能优化。

Proxy只会代理对象的第一层，Vue3是怎样处理这个问题的呢？

判断当前Reflect.get的返回值是否为Object，如果是则再通过reactive方法做代理，这样就实现了深度观测。

监测数组的时候可能触发多次get/set，那么如何防止触发多次呢？我们可以判断key是否为当前被代理对象target自身属性，也可以判断旧值与新值是否相等，只有满足以上两个条件之一时，才有可能执行trigger。

Proxy 与 Object.defineProperty 优劣对比

Proxy 的优势如下：

Proxy 可以直接监听对象而非属性；

Proxy 可以直接监听数组的变化；

Proxy 有多达 13 种拦截方法，不限于 apply、ownKeys、deleteProperty、has 等等是 Object.defineProperty 不具备的；

Proxy 返回的是一个新对象，我们可以只操作新的对象达到目的，而 Object.defineProperty 只能遍历对象属性直接修改；

Proxy 作为新标准将受到浏览器厂商重点持续的性能优化，也就是传说中的新标准的性能红利；

Object.defineProperty 的优势如下：

兼容性好，支持 IE9，而 Proxy 的存在浏览器兼容性问题，而且无法用 polyfill 磨平，因此 Vue 的作者才声明需要等到下个大版本(3.0)才能用 Proxy 重写。

vue的生命周期必须带着自定义组件怎么回事？

创建的过程是一颗树，深度优先，从上往下进行创建，然后从下往上挂载

parent>child

```
1 parent created
2 parent beforeMount
3   child created
4   child beforeMount
5   child mounted
6 parent mounted
```

Vue中组件生命周期调用顺序是什么样的？

组件的调用顺序都是先父后子，渲染完成的顺序是先子后父。

组件的销毁操作是先父后子，销毁完成的顺序是先子后父。

组件的data配置为什么是函数？

保证同一个组件的多个实例对象的data对象不是共用的，而是各自自己的data对象。

一个组件被复用多次的话，也就会创建多个实例。本质上，这些实例用的都是同一个构造函数。

如果data是对象的话，对象属于引用类型，会影响到所有的实例。

nextTick的实现原理是什么？

在下次 DOM 更新循环结束之后执行延迟回调，在修改数据之后立即使用 nextTick 来获取更新后的 DOM。

nextTick主要使用了宏任务和微任务。

根据执行环境分别尝试采用Promise、MutationObserver、setImmediate，如果以上都不行则采用 setTimeout 定义了一个异步方法，多次调用nextTick会将方法存入队列中，通过这个异步方法清空当前队列。

说一下虚拟Dom以及key属性的作用

由于在浏览器中操作DOM是很昂贵的。

频繁的操作DOM，会产生一定的性能问题。

Virtual DOM本质就是用一个原生的JS对象去描述一个DOM节点。是对真实DOM的一层抽象。(也就是源码中的VNode类，它定义在src/core/vdom/vnode.js中。)

虚拟 DOM 的实现原理主要包括以下 3 部分：

- 用 JavaScript 对象模拟真实 DOM 树，对真实 DOM 进行抽象；
- diff 算法 — 比较两棵虚拟 DOM 树的差异；
- patch 算法 — 将两个虚拟 DOM 对象的差异应用到真正的 DOM 树。

key 是为 Vue 中 vnode 的唯一标记，通过这个 key，我们的 diff 操作可以更准确、更快速

更准确：因为带 key 就不是就地复用了，在 sameNode 函数 `a.key === b.key` 对比中可以避免就地复用的情况。

所以会更加准确。

更快速：利用 key 的唯一性生成 map 对象来获取对应节点，比遍历方式更快

组件对象与vue什么关系

所有组件对象的原型对象都是一个vm对象

所有组件对象都能看到定义在vue原型对象上的属性或方法

mvvm模式

M: model(模型), vue中是data(为view提供数据)

V: view(视图), vue中是模板页面(显示data中的数据)

VM: ViewModel (视图模型) ,vue中是vue是实例对象(管理者:数据绑定/DOM监听)

路由模式 (不用脚手架)

hash模式:

路径中带#:<http://localhost:8080/#/home/news>

发请求的路径: <http://localhost:8080>项目根路径

响应: 返回来的总是index页面=>path部分(/home/news)被解析为前台路由路径

history模式:

路径中不带#:<http://localhost:8080/home/news>

发请求的路径: <http://localhost:8080/home/news>

响应: 404错误

希望: 如果没有对应的资源, 返回来的总是index页面=>path部分(/home/news)被解析为前台路由路径

解决: 添加配置

```
devServer: {  
  historyApiFallback:true//任意的404响应都被替换为index.html  
}  
  
output: publicPath:'',//引入打包的文件时路径以/开头
```

你的接口请求一般放在哪个生命周期中?

以在钩子函数 created、 beforeMount、 mounted 中进行调用, 因为在这三个钩子函数中, data 已经创建, 可以将服务端返回的数据进行赋值。

但是推荐在 created 钩子函数中调用异步请求, 因为在 created 钩子函数中调用异步请求有以下优点:

能更快获取到服务端数据, 减少页面loading 时间;

ssr不支持 beforeMount、 mounted 钩子函数, 所以放在 created 中有助于一致性;

Vue模版编译原理知道吗

简单说, Vue的编译过程就是将template转化为render函数的过程。会经历以下阶段 (生成AST树/优化/codegen) :

首先解析模版, 生成AST语法树(一种用JavaScript对象的形式来描述整个模板)。

使用大量的正则表达式对模板进行解析, 遇到标签、文本的时候都会执行对应的钩子进行相关处理。

Vue的数据是响应式的, 但其实模板中并不是所有的数据都是响应式的。

有一些数据首次渲染后就不会再变化，对应的DOM也不会变化。

那么优化过程就是深度遍历AST树，按照相关条件对树节点进行标记

这些被标记的节点(静态节点)我们就可以跳过对它们的比对，对运行时的模板起到很大的优化作用。

编译的最后一步是将优化后的AST树转换为可执行的代码。

说说你对SSR的了解

SSR也就是服务端渲染，也就是将Vue在客户端把标签渲染成HTML的工作放在服务端完成，然后再把html直接返回给客户端

SSR的优势

更好的SEO

首屏加载速度更快

SSR的缺点

开发条件会受到限制，服务器端渲染只支持beforeCreate和created两个钩子

当我们需要一些外部扩展库时需要特殊处理，服务端渲染应用程序也需要处于Node.js的运行环境

更多的服务端负载

对于将来的vue3.0特性你有什么了解？

监测机制的改变

3.0 将带来基于代理 Proxy 的 observer 实现，提供全语言覆盖的反应性跟踪。

消除了 Vue 2 当中基于 Object.defineProperty 的实现所存在的很多限制：

只能监测属性，不能监测对象

检测属性的添加和删除；

检测数组索引和长度的变更；

支持 Map、Set、WeakMap 和 WeakSet

模板

模板方面没有大的变更，只改了作用域插槽，2.x 的机制导致作用域插槽变了，父组件会重新渲染，而3.0 把作用域插槽改成了函数的方式，这样只会影响子组件的重新渲染，提升了渲染的性能。

同时，对于 render 函数的方面，vue3.0 也会进行一系列更改来方便习惯直接使用 api 来生成 vdom。

对象式的组件声明方式

vue2.x 中的组件是通过声明的方式传入一系列 option，和 TypeScript 的结合需要通过一些装饰器的方式来操作，虽然能实现功能，但是比较麻烦。

3.0 修改了组件的声明方式，改成了类式的写法，这样使得和 TypeScript 的结合变得很容易

其它方面的更改

支持自定义渲染器，从而使得 weex 可以通过自定义渲染器的方式来扩展，而不是直接 fork 源码来改的方式。

支持 Fragment (多个根节点) 和 Protal (在 dom 其他部分渲染组建内容) 组件，针对一些特殊的场景做了处理。

基于 tree shaking 优化，提供了更多的内置功能。

组件化常用技术

组件传值、通信

父组件=>子组件：

- 属性props

```
1 //child
2 props:{msg:String}
3
4 //parent
5 <HelloWorld msg="welcome to vue.js"/>
```

```
<HelloWorld msg="Welcome to Your Vue.js App"/>
```

```
1
2
3 <script>
4   export default {
5     name: 'HelloWorld',
6     props: {
7       msg: String
8     }
9   }
10  </script>
```

- 特性\$attrs

```
1 //child:并未在props中声明foo
2 <p>
3   {{$attrs.foo}}
4 </p>
5
6 //parent
7 <HelloWorld foo='foo'/>
```

- 引用refs

```
1 //parent
2 <HelloWorld ref='hw'/>
3
4 //child
5 this.$refs.hw.xx='xxxx'
```

- 子组件children

```
1 //parent
2 this.$children[0].xx='xxx'//取出来的子组件没有顺序
```

子组件=>父组件：自定义事件

```
1 //child  
2 this.$emit('add', good)  
3  
4 //parent  
5 <Cart @add='cartAdd($event)'></Cart>
```

兄弟组件：通过共同祖辈组件

通过共同的祖辈组件搭桥，parent或root

```
1 //borther1  
2 this.$parent.$on('foo', handle)  
3  
4 //borther2  
5 this.$parent.$emit('foo')
```

祖先和后代之间

由于嵌套层数过多，传递props不切实际，vue提供了provide/inject API完成该任务

- provide/inject：能够实现祖先给后代传值

```
1 //ancestor  
2 provide(){  
3     return {foo: 'foo'}  
4 }  
5  
6 //descendant  
7 inject:['foo']
```

注意：provide和inject主要为高阶插件/组件库提供用例，并不推荐直接用于应用程序代码中，我们更多会在开源组件库中见到

但是，反过来想要后代给祖先传值这种方案就不行了

任意两个组件之间：事件总线或vuex

- 事件总线：创建一个Bus类负责事件派发、监听和回调管理

```
1 //Bus: 事件派发、监听和回调管理  
2 class Bus{  
3     constructor(){  
4         this.callbacks={};  
5     }  
6     $on(name,fn){  
7         this.callbacks[name]=this.callbacks[name] || [];  
8         this.callbacks[name].push(fn)  
9     }  
10    $emit(name,args){  
11        if(this.callbacks[name]){  
12            this.callbacks[name].forEach(cb=>cb(args))  
13        }  
14    }  
15 }
```

```
16 }
17
18
19 //main.js
20 Vue.prototype.$bus=new Bus();
21
22 //child1
23 this.$bus.$on('foo',handle);
24 //child2
25 this.$bus.$emit('foo')
```

- vuex: 创建唯一的全局数据管理者store, 通过他管理数据并通知组件状态变更

插槽

Vue2.6.0之后采用全新v-slot语法取代之前的slot、 slot-scope

匿名插槽

```
1 //comp1
2 <div>
3   <slot></slot>
4 </div>
5
6 //parent
7 <comp>hello</comp>
```

具名插槽

```
1 //Comp2
2 <div>
3   <slot></slot>
4   <slot name="content"></slot>
5 <div>
6
7 //parent
8 <Comp2>
9   <!--默认插槽用default做参数-->
10  <template v-slot:default>具名插槽</template>
11  <!-- 具名插槽用插槽名做参数 -->
12  <template v-slot:content>内容...</template>
13 </Comp2>
```

作用域插槽

```
1 //comp3
2 <div>
3   <slot :foo='foo'></slot>
4 </div>
5
6 //parent
7 <Comp3>
8   <!--把v-slot的值指定为作用域上下文对象-->
9   <template v-slot:default="ctx">
10     来自子组件数据: {{ctx.foo}}
11   </template>
12 </Comp3>
```

表单组件实现

Form 管理数据模型-model、校验规则-rules、全局校验方法-validate

FormItem 显示标签-label、执行校验-prop和显示校验结果

Input 绑定数据模型-v-model、通知FormItem执行校验

需要思考的几个问题？

1. Input是自定义组件，它是怎么实现数据绑定的？
2. FormItem怎么知道何时执行校验，校验的数据和规则怎么得到？
3. Form怎么进行全局校验？它用什么办法把数据模型和校验规则传递给内部组件？

- input
 - 双向绑定：@input 和：value
 - 派发校验事件

```
1 <template>
2   <div>
3     <!-- $attrs是props之外的的部分 -->
4     <input :value="value" @input="onInput" v-bind="$attrs"/>
5   </div>
6 </template>
7
8 <script>
9 export default {
10   inheritAttrs:false, //避免顶层容器继承属性
11   props:{
12     value:{
13       type:String,
14       default:''
15     }
16   },
17 }
```

```

17     methods:{
18         onInput(e){
19             //通知父组件数值变化
20             this.$emit('input',e.target.value);
21
22             //事件派发,通知FormItem做校验,因为this.$emit(),<slot>
23             //</slot>没办法进行派发事件
24             this.$parent.$emit('validate');
25         }
26     }
27 };
28 </script>
29
30 <style>
31 </style>

```

- FormItem

- 给Input预留插槽-slot
- 能够展示 label和校验信息
- 能够进行校验

```

1 <template>
2     <div>
3         <label v-if="label">{{label}}</label>
4         <slot></slot>
5         <!--校验信息-->
6         <p v-if="errorMessage">{{errorMessage}}</p>
7     </div>
8 </template>
9
10 <script>
11 import Schema from "async-validator";
12 export default {
13     inject:['form'],
14     props:{
15         label:{
16             type:String,
17             default:''
18         },
19         prop:{
20             type:String
21         }
22     },
23     data(){
24         return{
25             errorMessage:''
26         }
27     },
28     mounted(){
29         //监听校验事件, 并执行监听
30         this.$on('validate',this.validate)
31     },
32     methods:{
33         validate(){
34             //执行组件校验

```

```

35 //1. 获取校验规则
36
37     const value= this.form.model[this.prop];
38     const rules= this.form.rules[this.prop];
39     const desc={[this.prop]:rules};
40     const schema=new Schema(desc);
41
42     //return 的是校验结果的promise
43     return schema.validate({[this.prop]:value},errors=>{
44         if(errors){
45             this.errorMessage=errors[0].message;
46         }else{
47             this.errorMessage='';
48         }
49     })
50 }
51 }
52 </script>
53 <style>
54
55 </style>
56
57 </style>

```

- Form

- 给FormItem留插槽
- 设置数据和校验规则
- 全局校验

```

1 <template>
2   <div>
3     <slot></slot>
4   </div>
5 </template>
6
7 <script>
8 export default {
9   provide() {
10     return {
11       form: this//传递form实例给后代, 比如FormItem用来校验
12     };
13   },
14   props:{
15     model:{
16       type:Object,
17       required:true
18     },
19     rules:{
20       type:Object
21     }
22   },
23   methods: {
24     validate(cb) {
25       //返回若干Promise数组
26       const tasks= this.$children
27         .filter(item=>item.prop)//过滤掉不需要校验的选项

```

```

28     .map(item=>item.validate());
29
30     //所有任务都通过校验通过
31     Promise.all(tasks)
32     .then(()=>{
33       cb(true)
34     }).catch(()=>{
35       cb(false)
36     })
37   },
38 },
39 };
40 </script>
41
42 <style>
43 </style>

```

.sync和v-model的异同

```

1  <!-- v-model是语法糖 -->
2  <input v-model="username"/>
3  <!-- 默认等效于下面这行 -->
4  <input :value="username" @input="username=$event"/>
5
6  <!-- 但是你可以通过设置model选项修改默认行为, checkbox.vue -->
7  {
8    model:{
9      prop: 'checked',
10     event: 'change'
11   }
12 }
13 <!-- 上面这样设置会导致上级使用v-model时行为变化, 相当于-->
14 <kCheckBox :checked="model.remember" @change="model.remember=$event">
</kCheckBox>
15 <!-- 场景: v-model通常用于表单控件, 它有默认行为, 同时属性名和事件名均可在子组件定义
-->
16
17 <!-- sync修饰符添加v2.4, 类似于v-model, 它能用于修改传递到子组件的属性, 如果像下面
这样写 -->
18 <input :value.sync="model.username" />
19 <!-- 等效于下面这行, 那么和v-model的区别只有事件名称的变化 -->
20 <input :value="username" @update:value="username=$event" />
21 <!-- 这里绑定属性名称更改, 相应的属性名也会发生变化 -->
22 <input :foo="username" @update:foo="username=$event" />
23 <!-- 场景: 父组件传递的属性子组件想修改 -->
24 <!-- 所以sync修饰符的控制能力都在父级, 事件名称也相对固定update:xx -->
25 <!-- 习惯上表单元素用v-model -->

```

弹窗组件

弹窗这类组件的特点是它们在当前vue实例之外独立存在，通常挂载于body；它们是通过js动态创建的，不需要在任何组件中声明。常见使用姿势：

```
1 this.$create(Notice,{  
2     title: "xxx",  
3     message: "登录",  
4     duration: 1000  
5 }).show();
```

create

创建指定组件实例的方式共有2种

创建create函数用于动态创建指定组件实例并挂载至body

```
1 //创建指定组件实例挂载于body上  
2 import Vue from "vue";  
3 export default function create(Component,props){  
4     //第一种解法  
5     //1.创建vue实例  
6     const vm=new Vue({  
7         render(h){  
8             //render方法提供给我们一个h函数，他可以渲染vnode  
9             return h(Component,{props})  
10        }  
11    }).$mount();//组件更新操作，此时更新目标没有做设置，意思是执行初始化操作，但是没有追加  
dom操作  
12    //2.上面vm帮我们创建组件实例  
13    //3.通过$children获取该组件实例  
14    // console.log(vm.$root);  
15    const comp=vm.$children[0];  
16    //4.追加至body  
17    document.body.appendChild(vm.$el);  
18  
19    //5.清理函数  
20    comp.remove=>{  
21        document.body.removeChild(vm.$el);  
22        vm.$destroy();  
23    }  
24    //6.返回组件实例  
25    return comp;  
26  
27    //第二种解法  
28    const Ctor=Vue.extend(Component);  
29    const comp=new Ctor({propsData:props});//通过propsData传递属性  
30    comp.$mount();  
31    document.body.appendChild(comp.$el);  
32    comp.remove=>{  
33        document.body.removeChild(comp.$el);  
34        comp.$destroy();  
35    }  
36    return comp;  
37}  
38
```

new Vue,vue.extend,vue.component有什么区别?

new vue是一个vue实例

vue.extend返回的是一个“扩展实例构造器”，也就是一个预设了部分选项的vue实例构造器

```
1 let myVue=Vue.extend({
2     //预设选项
3 })//返回一个“扩展实例构造”
4
5 //然后就可以这样来使用
6 let vm=new myVue({
7     //其他选项
8 })
```

vue.component是用来全局注册组件的方法，其作用是将通过vue.extend生成的扩展实例构造器注册(命名)为一个组件，可以简单理解为当在模版中遇到该组件名称作为该组件名称作为标签的自定义元素时，会自动调用类似，

props 的值需要通过 propsData 属性来传递。

new myVue这样的构造函数来生成组件实例，并挂载到自定义元素上。

render函数的作用是得到描述dom结构的虚拟dom

创建通知组件，Notice.vue

```
1 <template>
2     <div v-if="isShow">
3         <h3>{{title}}</h3>
4         <p>{{message}}</p>
5     </div>
6 </template>
7
8 <script>
9 export default {
10     props: {
11         title: {
12             type: String,
13             default: ""
14         },
15         message: {
16             type: String,
17             default: ""
18         },
19         duration: {
20             type: Number,
21         }
22     },
23     data() {
24         return {
25             isShow: false
26         };
27     },
28     methods: {
29         show() {
30             this.isShow = true;
31             setTimeout(()=>{
```

```

32         this.hide();
33     },this.duration)
34   },
35   hide() {
36     this.isShow = false;
37     this.remove();
38   }
39 }
40 };
41 </script>
42
43 <style scoped>
44 .box{
45   position: fixed;
46   width:100%;
47   top:16px;
48   left:0;
49   text-align:center;
50   pointer-events: none;
51 }
52 .box-content{
53   width:100px;
54   margin: 10px auto;
55   font-size:14px;
56   border: blue 3px solid;
57   padding:8px 16px;
58   background: #fff;
59   border-radius: 3px;
60   margin-bottom: 8px;
61 }
62 </style>

```

使用createapi

```

1 import Notice from "../../Notice/index";
2 import create from "@/utils/create";
3
4 export default {
5   components: {
6     KInput,
7     KFormItem,
8     KForm
9   },
10  data() {
11    return {
12      model: {
13        username: "tom",
14        password: ""
15      },
16      rules: {
17        username: [{ required: true, message: "用户必填" }],
18        password: [{ required: true, message: "密码必填" }]
19      }
20    };
21  },
22  methods: {
23    onLogin() {

```

```

24     //创建弹窗的实例
25     let notice = "";
26     this.$refs.loginForm.validate(isValid => {
27       if (isValid) {
28         // alert('denglu')
29         notice= create(Notice, {
30           title: "xxx",
31           message: "登录",
32           duration: 1000
33         });
34       } else {
35         // alert("校验失败");
36         notice= create(Notice, {
37           title: "xxx",
38           message: "失败",
39           duration: 1000
40         });
41       }
42       notice.show();
43     });
44   }
45 }
46 }
47 };

```

递归组件

递归组件是可以它们模板中调用自身的组件

```

1 <template>
2   <div>
3     <h3>{{data.title}}</h3>
4     <!-- 必须有结束条件 -->
5     <Node v-for="d in data.children" :key="d.id" :data="d"></Node>
6   </div>
7 </template>
8
9 <script>
10 export default {
11   name:'Node',//name对递归组件是必须的
12   props: {
13     data: {
14       type: Object,
15       require:true,
16     },
17   },
18 }
19 </script>
20
21
22
23
24 //使用
25 <Node :data="{id:'1',title:'递归组件',children:[{....}]}></Node>

```

实现Tree组件

Tree组件是典型的递归组件，其他诸如菜单组件都属于这一类，也是相当常见的

组件设计

Tree组件最适合的结构是无序列表ul,创建一个递归组件Item表示Tree选项，如果当前Item存在children，则递归渲染子树，以此类推；同时添加一个标识管理当前层级item的展开状态。

实现Item组件

```
1 <template>
2   <li>
3     <div @click="toggle">
4       {{model.title}}
5       <span v-if="isFolder">[{{open?'-':'+'}}]</span>
6     </div>
7     <ul v-show="open" v-if="isFolder">
8       <item class="item" v-for="model in model.children" :model="model"
9         :key="model.title"><Item>
10      </ul>
11    </li>
12  </template>
13
14 <script>
15   export default {
16     name:'Item',
17     props: {
18       model: {
19         type: Object,
20       },
21       data() {
22         return {
23           open: false
24         }
25       },
26       computed: {
27         isFolder() {
28           return this.model.children&&this.model.chidren.length;
29         }
30       },
31       methods: {
32         toggle() {
33           if(this.isFolder){
34             this.open=!this.open;
35           }
36         }
37       },
38     }
39   </script>
40
41 <style>
42
43 </style>
```

使用

```
1 <template>
```

```
2 <div id="app">
3   <ul>
4     <item class="item" :model="treeData"></item>
5   </ul>
6 </div>
7 </template>
8
9 <script>
10 import Item from "./Item";
11 export default {
12   name:'ItemIndex',
13   components:{Item},
14   data() {
15     return {
16       treeData: {
17         title:'web全栈架构师',
18         children:[
19           {
20             title:'Java架构师'
21           },
22           {
23             title:'JS高级',
24             children:[
25               {
26                 title:'ES6'
27               },
28               {
29                 title:'动效'
30               }
31             ],
32             title:'web全栈',
33             children:[
34               {
35                 title:'Vue训练营',
36                 expand:true,
37                 children:[
38                   {
39                     title:'组件化'
40                   },
41                   {
42                     title:'源码'
43                   },
44                   {
45                     title:'docker部署'
46                   }
47                 ],
48               },
49               {
50                 title:'React',
51                 children:[
52                   {
53                     title:'JSX'
54                   },
55                   {
56                     title:'虚拟DOM'
57                   }
58                 ],
59               }
60             ]
61           }
62         ]
63       }
64     }
65   }
66 }
```

```
60 </script>
61
62 <style>
63
64 </style>
```

vue-router

安装:

```
1 | vue add router
```

起步:

配置

```
1 import Vue from 'vue'
2 import VueRouter from 'vue-router'
3 import Home from '../views/Home.vue'
4
5 Vue.use(VueRouter)
6
7 const routes = [
8   {
9     path: '/',
10    name: 'home',
11    component: Home
12  },
13  {
14    path: '/about',
15    name: 'about',
16    meta:{auth:true},
17    // route level code-splitting
18    // this generates a separate chunk (about.[hash].js) for this route
19    // which is lazy-loaded when the route is visited.
20    component: () => import(/* webpackChunkName: "about" */'./views/About.vue')
21  }
22 ]
23
24 const router = new VueRouter({
25   mode: 'history',//模式:hash|history|abstract
26   base: process.env.BASE_URL,
27   routes
28 })
29
30 export default router
31
```

指定路由器

```
1 //main.js
2 new Vue({
3   router,
4   render: h => h(App)
5 }).$mount('#app')
```

路由视图

```
1 <router-view/>
```

导航链接

```
1 <router-link to="/">Home</router-link>
2 <router-link to="/about">About</router-link>
```

路由嵌套

应用界面通常由多层嵌套组件而成。同样地，URL中各段动态也按某种结构对应嵌套的各层组件

配置嵌套路由，router.js

```
1 path: '/',
2   name: 'home',
3   component: Home,
4   children:[{
5     path:'/list',name:'list',component>List
6   }]
```

父组件需要添加插座，Home.vue

```
1 <template>
2   <div class="home">
3     <h1>首页</h1>
4     <router-view></router-view>
5   </div>
6 </template>
7
```

动态路由

我们经常需要把某种模式匹配到的所有路由，全部映射到同一个组件

```
1 {
2   path: '/',
3   name: 'home',
4   component: Home,
5   children:[{
6     path: '/',name:'list',component>List
7   },{
8     path:'detail/:id',name:'detail',component:Detail
9   }]
10 }
```

跳转List.vue

```
1 <ul>
2   <li><router-link to="/detail/1">web全栈</router-link></li>
3 </ul>
```

获取参数，Detail.vue

```
1 <template>
2   <div>
3     <h2>商品详情</h2>
4     <p> {{$route.params.id}} </p>
5   </div>
6 </template>
```

传递路由组件参数：

```
1 {path:'detail/:id',component:Detail,props:true}
```

组件中以属性方式获取：

```
1 export default {props:['id']}
```

路由守卫

路由导航过程中有若干生命周期钩子，可以在这里实现逻辑控制

全局守卫，router.js

```
1 import Vue from 'vue'
2 import VueRouter from 'vue-router'
3 import Home from '../views/Home.vue'
4
5 Vue.use(VueRouter)
6
7 const routes = [
8   {
9     path: '/',
10    name: 'home',
11    component: Home,
12  },
13  {
14    path: '/about',
15    name: 'about',
16    meta:{auth:true},
17    // route level code-splitting
18    // this generates a separate chunk (about.[hash].js) for this route
19    // which is lazy-loaded when the route is visited.
20    component: () => import(/* webpackChunkName: "about" */'../views/About.vue')
21  }
22 ]
23
24 const router = new VueRouter({
25   mode: 'history',//模式:hash|history|abstract
26   base: process.env.BASE_URL,
27   routes
```

```
28 })  
29  
30 //全局守卫  
31 router.beforeEach((to, from, next)=>{  
32   if(to.meta.auth&&!window.isLogin){  
33     if(window.confirm("请登录")){  
34       window.isLogin=true;  
35       next();//登录成功, 继续  
36     }else{  
37       next("//");//放弃登录, 回首页  
38     }  
39   }else{  
40     next();  
41   }  
42 })  
43  
44 export default router  
45
```

路由独享守卫

```
1 beforeEnter(to, from, next){  
2   //路由内部知道自己需要认证  
3   if(!window.isLogin){  
4     //...  
5   }else{  
6     next();  
7   }  
8 }
```

组件内的守卫

```
1 export default{  
2   beforeRouteEnter(to, from, next){},  
3   beforeRouteUpdate(to, from, next){},  
4   beforeRouteLeave(to, from, next){}  
5 }
```

vue-router扩展

动态路由

利用\$router.addRoutes()可以实现动态路由添加，常用于用户权限控制

```
1 //router.js  
2 //返回数据可能是这样的  
3 //[{  
4 //  path: '/',  
5 //  name: 'home',  
6 //  component: 'Home',//Home  
7 //}]\n8  
9 //异步获取路由  
10 api.getRoutes().then(routes=>{  
11   const routeConfig=routes.map(route=>mapComponent(route));
```

```

12     router.addRoutes(routeConfig)
13 }
14
15 //映射关系
16 const compMap={
17     'Home':()=>import("./view/Home.vue")
18 }
19
20 //递归替换
21 function mapComponent(){
22     route.component=compMap[route.component];
23     if(route.children){
24         route.children=route.children.map(child=>mapComponent(child))
25     }
26     return route;
27 }

```

面包屑

利用\$route.matched可得到路由匹配数组，按顺序解析可得路由层次关系

```

1 //Breadcrumb.vue
2 watch: {
3     $route() {
4         // [{name:'home',path:'/'},{name:'list',path:'/list'}]
5         console.log(this.$route.matched);
6         //['home','list']
7         this.crumbData=this.$route.matched.map(m=>m.name||m.redirect);
8     },
9     immediate:true//这一行要加上，让它一开始执行一次
10 }

```

vue-router源码实现

通常用法

```

1 import Vue from 'vue'
2 import VueRouter from './vue-router'
3 import Home from '../views/Home.vue'
4 import About from '../views/About.vue'
5
6 Vue.use(VueRouter)
7
8 const routes = [
9     {
10         path: '/',
11         name: 'home',
12         component: Home,
13     },
14     {
15         path: '/about',
16         name: 'about',
17         component:About,
18     }
19 ]
20

```

```

21 const router = new VueRouter({
22   mode: 'history',//模式:hash|history|abstract
23   base: process.env.BASE_URL,
24   routes
25 })
26
27 //main.js
28 import router from "./krouter"

```

分析一下需要完成的任务：

- 1.要能解析routes配置，变成一个key为path， value为component的map
- 2.要能监控url变化事件，把最新的hash值保存到current路由
- 3.要定义两个全局组件:router-view用于显示匹配组件内容， router-link用于修改hash
- 4.current应该是响应式的，这样可以触发router-view的重新渲染

具体实现

```

1 //Krouter.js
2 let Vue;
3 class VueRouter{
4   constructor(options){
5     this.$options=options;
6
7     //创建一个路由path和route映射
8     this.routeMap={};
9
10    //将来当前路径current需要响应式
11    //利用Vue响应式原理可以做到这一点
12    this.app=new Vue({
13      data:{
14        current:'/'
15      }
16    })
17  }
18  init(){
19    //绑定浏览器事件
20    this.bindEvents();
21
22    //解析路由配置
23    this.createRouteMap(this.$options);
24
25    //创建router-link和router-view
26    this.initComponent();
27  }
28  bindEvents(){
29    window.addEventListener('load',this.onHashChange.bind(this))
30    window.addEventListener('hashchange',this.onHashChange.bind(this))
31  }
32  onHashChange(){
33    //http://localhost/#/home
34    this.app.current=window.location.hash.slice(1|| '/');
35  }
36  createRouteMap(options){
37    options.routes.forEach(item=>{
38      //['/home']: {path: '/home', component: Home}

```

```

39         this.routeMap[item.path]=item;
40     })
41   }
42   initComponent(){
43     //声明两个全局组件
44     vue.component('router-link',{
45       props:{
46         to:String
47       },
48       render(h){
49         //目标是:<a :href="to">xxx</a>
50         //this.$slots.default是xxx
51         // return (<a href={this.to}>{this.$slots.default}</a>)
52         return h('a',{attrs:
53           {href:'#'+this.to}},this.$slots.default);
54       }
55     });
56     vue.component('router-view',{
57       //箭头函数能保留this的指向, 这里指向VueRouter实例
58       render:(h)=>{
59         const Comp=this.routeMap[this.app.current].component;
60         return h(Comp)
61       }
62     });
63   }
64
65 //把VueRouter变为插件
66 VueRouter.install=function(_vue){
67   _Vue=_Vue;//这里保存, 上面使用
68   _Vue.mixin({//混入: 就是扩展vue
69     beforeCreate() {
70       //这里的代码将来会在外面初始化的时候被调用
71       //这样我们就实现了Vue扩展
72       //this是Vue组件实例
73       //但是这里只希望根组件执行一次
74       if(this.$options.router){
75
76         _Vue.prototype.$router=this.$option.router;
77         this.$options.init();
78       }
79     },
80   })
81 }
82
83 export default VueRouter;

```

测试注意:

不要出现router-view嵌套, 因为没有考虑, 把Home中的router-view先禁用导航链接修改为hash

考虑嵌套路由的情况

```

1 let _Vue;
2 class VueRouter {
3   static install(vue) {
4     _Vue = vue;
5     vue.mixins({

```

```
6     beforeCreate() {
7         if (this.$options.router) {
8             Vue.prototype.$router = this.$options.router;
9         }
10    });
11);
12 Vue.component("router-link", {
13     props: {
14         to: {
15             type: String,
16             required: true
17         }
18     },
19     render(h) {
20         return h(
21             "a",
22             {
23                 attrs: {
24                     href: "#" + this.to
25                 }
26             },
27             this.$slots.default
28         );
29     }
30});
31 Vue.component("router-view", {
32     render(h) {
33         //标记router-view深度
34         this.$vnode.data.routerView = true;
35         let depth = 0;
36         let parent = this.$parent;
37         while (parent) {
38             const vnodeData = parent.$vnode && parent.$vnode.data;
39             if (vnodeData) {
40                 if (vnodeData.routerView) {
41                     depth++;
42                 }
43             }
44             parent = parent.$parent;
45         }
46         //这里要找到相匹配的路径的组件
47         const route = this.$router.matched[depth];
48         if (!route) {
49             return;
50         }
51         const Component = route.component;
52         return h(Component);
53     }
54 });
55}
56 constructor(options) {
57     this.$options = options;
58     window.addEventListener("hashchange", this.onhashchange);
59     this.current = window.location.hash.slice(1) || "/";
60     // _Vue.util.defineReactive(this,'current',initial);
61     _Vue.util.defineReactive(this, "matched", []);
62     //match方法可以递归的遍历路由表
63     this.match();
```

```

64    }
65    match(routes) {
66      routes = routes || this.$options.routes;
67      for (const route of routes) {
68        if (route.path === "/" && this.current === "/") {
69          this.matched.push(route);
70          return;
71        }
72        if (route.path !== "/" && this.current.indexOf(route.path) != -1) {
73          this.matched.push(route);
74          if (route.children) {
75            this.match(route.children);
76          }
77          return;
78        }
79      }
80    }
81    onhashchange = () => {
82      this.current = window.location.hash.slice(1);
83      this.matched = [];
84      this.match();
85    };
86  }
87
88  export default VueRouter;
89

```

拓展

vue插件

```

1 //插件定义
2 MyPlugin.install=function(vue,options){
3   //1.添加全局方法和属性
4   vue.myGobalMethod=function(){
5     //逻辑...
6   }
7   //2.添加全局资源
8   vue.directive('my-directive',{
9     bind(el,binding,vnode,oldvnode){
10       //逻辑...
11     }
12   })
13
14   //3.注入组件选项(作用于所有组件)
15   vue.mixin({
16     created:function(){
17       //逻辑...
18     }
19   })
20
21   //4.添加实例方法
22   vue.prototype.$myMethod=function(methodOptions){
23     //逻辑...
24   }
25 }
26

```

```
27  
28  
29 //插件使用  
30 Vue.use(MyPlugin)
```

组件混入： mixin

混入(mixin)提供了一种分发分发Vue组件可复用功能的灵活方式

```
1 //定义一个混入对象  
2 let myMixin={  
3     created:function(){  
4         this.hello()  
5     },  
6     methods:{  
7         hello:function(){  
8             console.log('hello from mixin!');  
9         }  
10    }  
11 }  
12  
13 //定义一个使用混入对象的组件  
14 let component=Vue.extend({  
15     mixins:[myMixin]  
16 })
```

render函数详解

一些 场景中需要JavaScript的完全编程能力，这时可以用渲染函数，它比模板更接近编译器

```
render(h){
```

```
    return h(tag,...,[children])
```

```
}
```

createElement函数

```
1 {  
2     //与`v-bind:class`的API相同  
3     //接受一个字符串、对象或字符串和对象组成的数组  
4     'class':{  
5         foo:true,  
6         bar:false  
7     },  
8     //与`v-bind:style`的API相同  
9     //接受一个字符串、对象或字符串和对象组成的数组  
10    style:{  
11        color:'red',  
12        fontSize:'14px'  
13    },  
14    //普通的HTML特性  
15    attrs:{  
16        id:'foo'  
17    },  
18    //组件prop  
19    props:{  
20        myProp:'bar'
```

```
21  },
22  //DOM属性
23  domProps:{
24      innerHTML: 'baz'
25  },
26  //事件监听器在'on'属性内
27  //但不再支持加`v-on:keyon.enter`这样的修饰器
28  //需要在处理函数中手动检查keyCode
29  on:{
30      click:this.clickHandler
31  }
32 }
```

函数式组件

组件若没有管理任何状态，也没有监听任何传递给它的状态，也没有生命周期方法，只要一个接受一些prop的，可标记为函数组件，此时它没有上下文

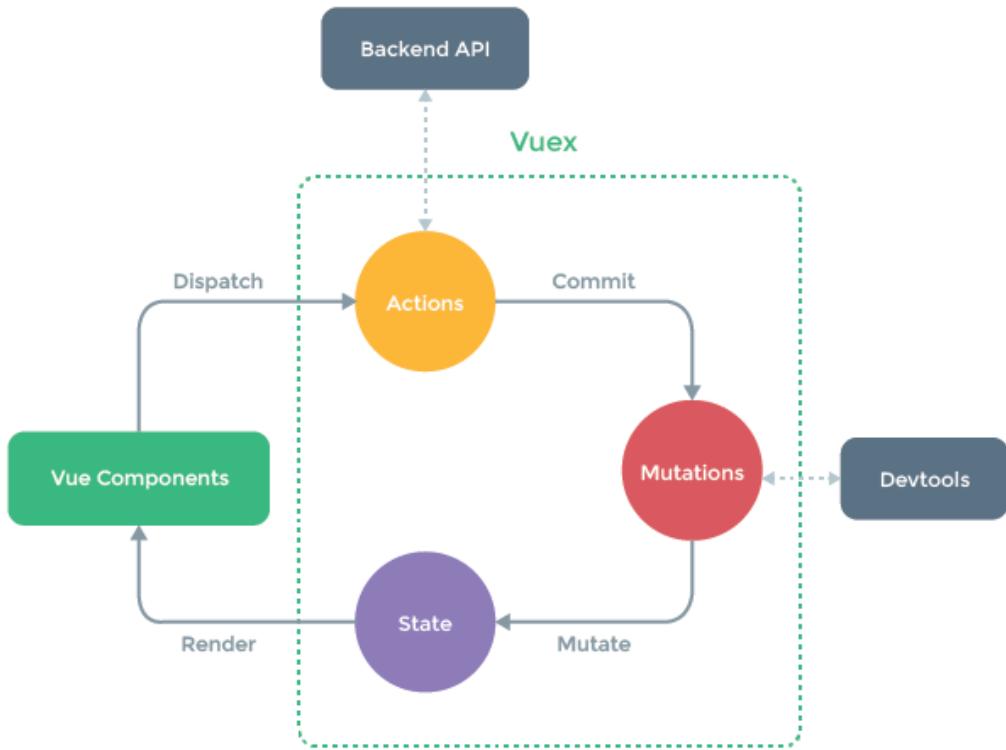
- props: 提供所有prop得对象
- children: VNode子节点的数组
- slots: 一个函数，反悔了包含所有插槽的对象
- scopedSlots: (2.6.0+) 一个暴露传入的作用域插槽的对象。也以函数形式暴露普通插槽
- data: 传递给组件的整个数据对象，作为createElement的第二个参数传入组件
- parent: 对父组件的引用
- listeners:(2.3.0+)一个包含了所有父组件为当前组件注册的事件监听器的对象。这是data.on的一个别名
- injections: (2.3.0+) 如果使用了inject选项，则该对象包含了应当被注入的属性

Vuex数据管理

Vuex是一个专为Vue.js应用开发的状态管理模式，集中式存储管理应用所有组件的状态

Vuex遵循“单向数据流”理念，易于问题追踪以及提高代码可维护性

Vue中多个视图依赖于统一状态时，视图间传参和状态同步比较困难，Vuex能够很好解决该问题



整合vuex

```
1 | vue add vuex
```

核心概念

- state状态、数据
- mutations更改状态的函数
- actions异步操作
- store包含以上概念

状态和状态变更

state保存数据状态，mutations用于修改状态，store.js

```

1 | export default new Vuex.Store({
2 |   state: {
3 |     counter:0
4 |   },
5 |   mutations: {//相当于react的reducer
6 |     add(state,num=1){
7 |       state.counter+=num;
8 |     }
9 |   }
10 | })

```

使用状态，vuex/index.vue

```

1 <template>
2   <div>
3     <div> 冲啊，手榴弹扔了{{store.state.counter}}个 </div>
4       <button @click="add">
5         扔一个
6       </button>
7     </div>
8   </template>
9   <script>
10    export default{
11      methods:{
12        add(){
13          this.$store.commit("increment");
14        }
15      }
16    }
17  </script>

```

派生状态-getters

从state派生出新状态，类似计算属性

```

1 export default new Vuex.Store({
2   getters:{
3     score(state){
4       return `共扔出: ${state.count}`
5     }
6   }
7 })

```

登录状态文字，App.vue

```
1 <span>{{store.getters.score}}</span>
```

动作-actions

复杂业务逻辑，类似于controller

```

1 export default new Vuex.Store({
2   actions: {
3     //复杂业务逻辑
4     //比如ajax请求
5     asyncAdd({commit,state}){
6       return new Promise(resolve=>{
7         setTimeout(()=>{
8           commit('add')
9           resolve({ok:1})
10        },1000)
11      })
12    }
13  },
14 })

```

使用actions

```

1 <template>
2   <div>
3     <div> 冲啊，手榴弹扔了{{$store.state.count}}个 </div>
4       <button @click="addAsync">
5         扔一个
6       </button>
7     </div>
8   </template>
9   <script>
10    export default{
11      methods:{
12        async asyncAdd(){
13          const result=await this.$store.dispatch('asyncAdd');
14          if(result.ok==1){
15            alert("操作成功|||")
16          }
17        }
18      }
19    }
20  </script>

```

模块化

按模块化的方式编写代码，store.js

```

1 const counter={
2   namespaced:true,
3   //...
4 }
5
6 export default new Vuex.Store({
7   modules:{
8     a:counter
9   }
10 })

```

使用变化，components/vuex/module.vue

```

1 <template>
2
3   <div>
4     {{$store.state.a.count}}
5     {{$store.getters['a/score']}}
6     <button @click="add">add</button>
7     <button @click="asyncAdd">add</button>
8   </div>
9
10 </template>
11
12 <script>
13 export default {
14   methods: {
15     add() {
16       this.$store.commit('a/add',2);
17     },
18     async asyncAdd(){

```

```

19         const result=await this.$store.dispatch('a/asyncAdd');
20         if(result.ok==1){
21             alert("操作成功|||")
22         }
23     },
24 }
25
26 </script>

```

vuex原理解析

初始化: Store声明, install实现, kvue.js

```

1 //1.维护状态state
2 //2.修改状态commit
3 //3.业务逻辑控制dispatch
4 //4.状态派发getter
5 //5.实现state响应式
6 //6.插件
7
8 let _Vue;
9 class Store{
10     constructor(options){
11         this._mutations=options.mutations;
12         this._actions=options.actions;
13         this._wrappedGetters=options.getters;
14         const computed={};
15         this.getters={};
16         const store=this;
17         Object.keys(this._wrappedGetters).forEach(key=>{
18             const fn=store._wrappedGetters[key];
19             computed[key]=function(){
20                 return fn(store.state)
21             }
22             //为getters定义只读属性
23             Object.defineProperty(store.getters,key,{
24                 get:>store._vm[key]
25             })
26         })
27
28         this._vm=new _Vue({
29             data(){
30                 return {
31                     //不希望被代理, 就加上$
32                     $$state:options.state
33                 },
34                 computed
35
36             }
37         })
38         this.commit=this.commit.bind(this);
39         this.dispatch=this.dispatch.bind(this);
40         this.dispatch=this.getters.bind(this);
41     }
42
43     get state(){
44         return this._vm._data.$$state

```

```

45 }
46
47 set state(v){
48     console.log('please use replacestate to reset state')
49 }
50
51 commit(type,payload){
52     const entry=this._mutations[type];
53     if(!entry){
54         return
55     }
56
57     entry(this.state,payload)
58 }
59
60 dispatch(type,payload){
61     const entry=this._mutations[type];
62     if(!entry){
63         return
64     }
65
66     return entry(this,payload)
67 }
68 getters(){
69 }
70 }
71 }
72
73 function install/vue){
74     _Vue=Vue;
75     _Vue.mixins({
76         beforeCreate() {
77             if(this.$options.store){
78                 Vue.prototype.$store=this.$options.store;
79             }
80         },
81     })
82 }
83 export default {store,install}

```

实现actions

```

1 class Store{
2     //options:{state:{count:0},mutations:{count(state){}}}
3     constructor(options={}){
4         this.actions=options.actions||[];
5     }
6
7     dispatch=(type,arg)=>{
8         const fn=this.actions[type];
9         return fn({commit:this.commit,state:this.state},arg)
10    }
11 }

```

实现getters

```

1 class Store{
2     //options:{state:{count:0},mutations:{count(state){}}}
3     constructor(options={}) {
4         options.getters&&this.handleGetters(options.getters)
5     }
6
7     //{getters: {score(state){return state.xxx}}}
8     handleGetters(getters) {
9         this.getters={};//store实例上的getters
10        //定义只读的属性
11        Object.keys(getters).forEach(key=>{
12            Object.defineProperty(this.getters,key,{ 
13                get:()=>{
14                    return getters[key](this.state);
15                }
16            })
17        })
18    }

```

为什么 Vuex 的 mutation 中不能做异步操作?

Vuex中所有的状态更新的唯一途径都是mutation，异步操作通过 Action 来提交 mutation实现，这样使得我们可以方便地跟踪每一个状态的变化，从而让我们能够实现一些工具帮助我们更好地了解我们的应用。

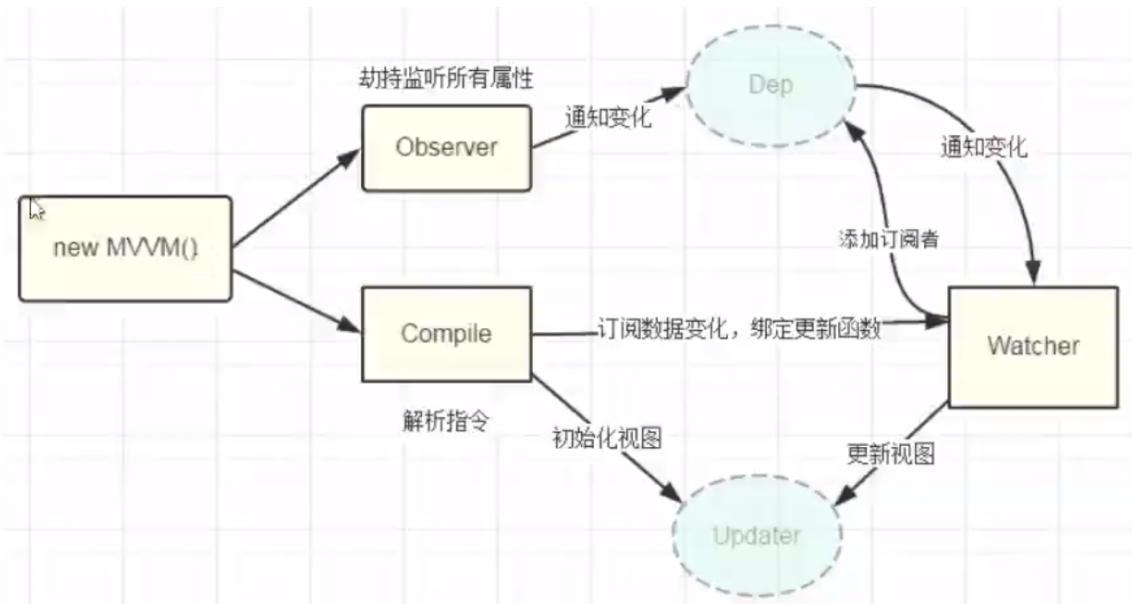
每个mutation执行完成后都会对应到一个新的状态变更，这样devtools就可以打个快照存下来，然后就可以实现 time-travel 了。

如果mutation支持异步操作，就没有办法知道状态是何时更新的，无法很好的进行状态的追踪，给调试带来困难。

新增：vuex的action有返回值吗？返回的是什么？

store.dispatch 可以处理被触发的 action 的处理函数返回的 Pro-mise，

Vue工作机制



初始化

在new vue()时会调用_init()进行初始化，会初始化各种实例方法、全局方法、执行一些生命周期、初始化props、data等状态。其中最重要的是data的响应化处理

初始化之后调用\$mount挂载组件，主要执行编译和首次更新

编译

编译模块分为三个阶段

1.parse：使用正则解析template中的vue的指令(v-xxx)变量等等形成抽象语法树AST

2.optimize：标记一些静态节点，用作后面的性能优化，在diff的时候直接略过

3.generate:把第一部生成的AST转化为渲染函数render function

虚拟dom

Virtual DOM是react首创，Vue2开始支持，就是用JavaScript对象来描述dom结构，数据修改的时候，我们先修改虚拟dom中的数据，然后数组做diff,最后再汇总所有的diff,力求做最少的dom操作，毕竟js里对比很快，而真实的dom操作太慢。

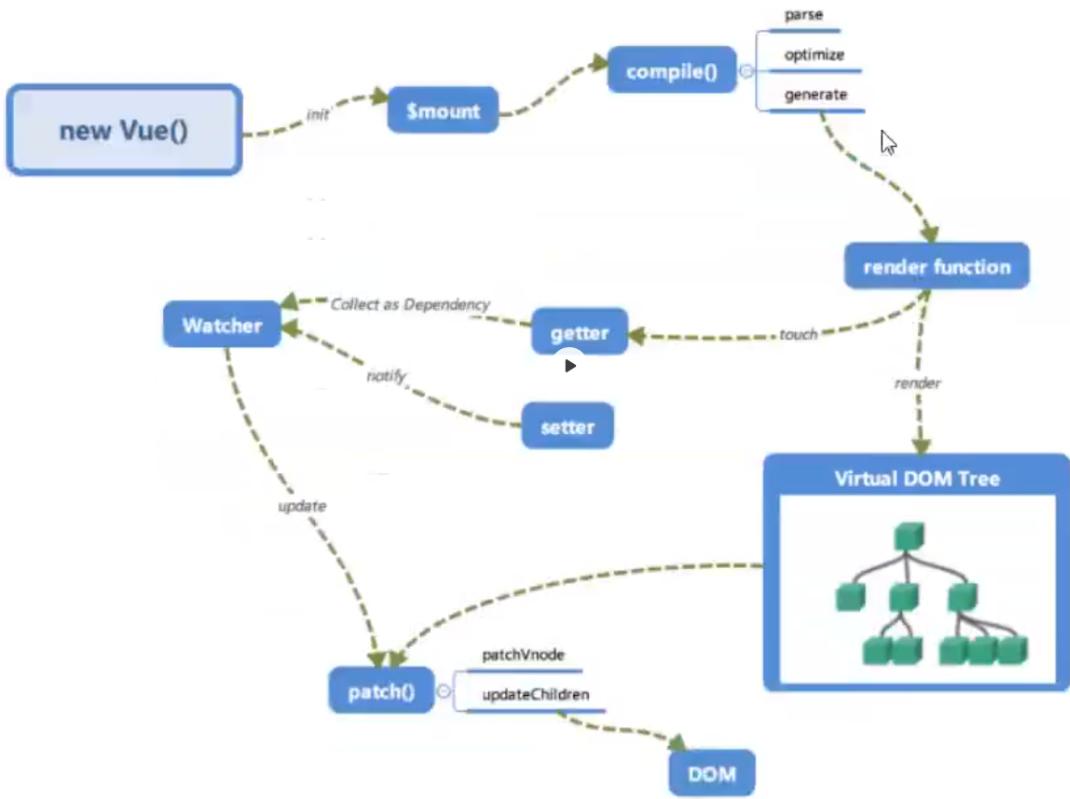
```
1 //vdom
2 {
3     tag:'div',
4     props:{
5         name:'zy1',
6         style:{color:red},
7         onClick:xx
8     },
9     children:[{
10         tag:'a',
11         text:'click me'
12     }]
13 }
```

```
1 <div name='zy1' style='color:red' @click='xx'>
2     <a>
3         click me
4     </a>
5 </div>
```

更新

数据修改触发setter,然后监听器会通知进行修改，通过对比新旧vdom树，得到最小修改，就是patch,然后只需要把这些差异修改即可

实现kvue



kvue源码

```

1 // Object.defineProperty()
2 // 拦截: 对某个对象的某个key做拦截
3 function defineReactive(obj, key, val) {
4     // 如果val是对象, 需要递归处理之
5     observe(val);
6
7     Object.defineProperty(obj, key, {
8         get() {
9             console.log("get", key);
10            return val;
11        },
12        set(newVal) {
13            if (val !== newVal) {
14                // 如果newVal是对象, 也要做响应式处理
15                observe(newVal);
16                val = newVal;
17                console.log("set", key, newVal);
18            }
19        }
20    });
21 }
22
23 // 遍历指定数据对象每个key, 拦截他们
24 function observe(obj) {
25     if (typeof obj !== "object" || obj === null) {
26         return obj;
27     }
28     new Observer(obj);
29     //创建一个Obersver实例
30 }
31

```

```
32 function proxy(vm, key) {
33     Object.key(vm[key]).forEach(k => {
34         Object.defineProperty(vm, k, {
35             get() {
36                 return vm[key][k];
37             },
38             set(v) {
39                 vm[key][k] = v;
40             }
41         });
42     });
43 }
44
45 class Observer {
46     constructor(value) {
47         this.value = value;
48         //遍历对象
49         this.walk(value);
50     }
51     walk(obj) {
52         Object.keys(obj).forEach(key => {
53             defineReactive(obj, key, obj[key]);
54         });
55     }
56 }
57
58 class KVue {
59     constructor(options) {
60         this.options = options;
61         this.$data = options.data;
62
63         //1.将data做响应式处理
64         observe(this.$data);
65         proxy(this, "$data");
66         new Compile("#app", this);
67     }
68 }
69
70 class Compile {
71     //el是宿主元素,vm-kvue实例
72     constructor(el, vm) {
73         this.$el = document.querySelector(el);
74         this.$vm = vm;
75
76         //解析模板
77         if (this.$el) {
78             //编译
79             this.compile(this.$el);
80         }
81     }
82
83     compile(el) {
84         //遍历它,判断当前遍历元素的类型
85         el.childNodes.forEach(node => {
86             if (node.nodeType === 1) {
87                 // console.log('编译元素', node.nodeName)
88                 this.compileElement(node);
89             } else if (this.isInter(node)) {

```

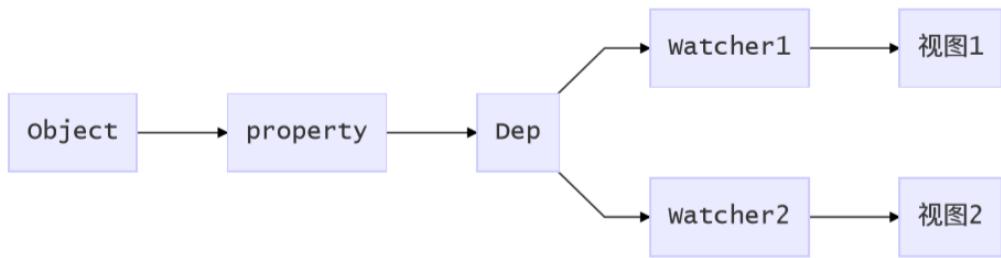
```
90         // console.log('编译文本',node.textContent)
91         this.compileText(node);
92     }
93
94     //递归
95     if (node.childNodes && node.childNodes.length > 0) {
96         this.compile(node);
97     }
98 });
99 }
100
101 // 判断插值表达式
102 isInter(node) {
103     return node.nodeType === 3 && /\{\{\.{*}\}\}/.test(node.textContent);
104 }
105
106 // 编译文本
107 compileText(node) {
108     this.update(node, RegExp.$1, "text");
109 }
110
111 // 编译元素: 分析指令、@事件
112 compileElement(node) {
113     // 获取属性并遍历之
114     const nodeAttrs = node.attributes;
115
116     Array.from(nodeAttrs).forEach(attr => {
117         // 指令: k-xxx="yyy"
118         const attrName = attr.name; // k-xxx
119         const exp = attr.value; // yyy
120         if (this.isDirective(attrName)) {
121             const dir = attrName.substring(2); // xxx
122             // 指令实际操作方法
123             this[dir] && this[dir](node, exp);
124         } else if (this.isEvent(attrName)) {
125             // 可能有@click
126             const dir = attrName.substring(1);
127             // window.addEventListener()
128             this.eventHandler(node, exp, dir);
129         }
130         // 处理事件
131     });
132 }
133 model(node, exp) {
134     //update方法只完成复制和更新
135     this.update(node, exp, "model");
136
137     //事件监听
138     node.addEventListener('input', e=>{
139         //新的值赋值给数据即可
140         this.$vm[exp]=e.target.value;
141     })
142 }
143 modelUpdater(node,value){
144     node.value=value;
145 }
146 isEvent(attr) {
147     return attr.slice(0, 1) === "@";
```

```

148 }
149 eventHandler(node, exp, dir) {
150   const fn = this.$vm.$options.methods && this.$vm.$options.methods[exp];
151   node.addEventListener(dir, fn.bind(this.$vm));
152 }
153 isDirective(attr) {
154   return attr.indexOf("k-") === 0;
155 }
156
157 // 执行text指令对应的更新函数
158 text(node, exp) {
159   this.update(node, exp, "text");
160 }
161
162 // k-text对应操作函数
163 textUpdater(node, val) {
164   console.log(val);
165   node.textContent = val;
166 }
167
168 html(node, exp) {
169   this.update(node, exp, "html");
170 }
171
172 htmlUpdater(node, val) {
173   node.innerHTML = val;
174 }
175
176 // 提取update, 初始化和更新函数创建
177 update(node, exp, dir) {
178   const fn = this[dir + "Updater"];
179   // 初始化
180   fn && fn(node, this.$vm[exp]);
181
182   // 更新
183   new Watcher(this.$vm, exp, function(val) {
184     fn && fn(node, val);
185   });
186 }
187
188
189 class Watcher {
190   constructor(vm, key, updaterFn) {
191     this.vm = vm;
192     this.key = key;
193     this.updaterFn = updaterFn;
194   }
195   update() {
196     this.updaterFn.call(this.vm, this.vm[this.key]);
197   }
198 }
199

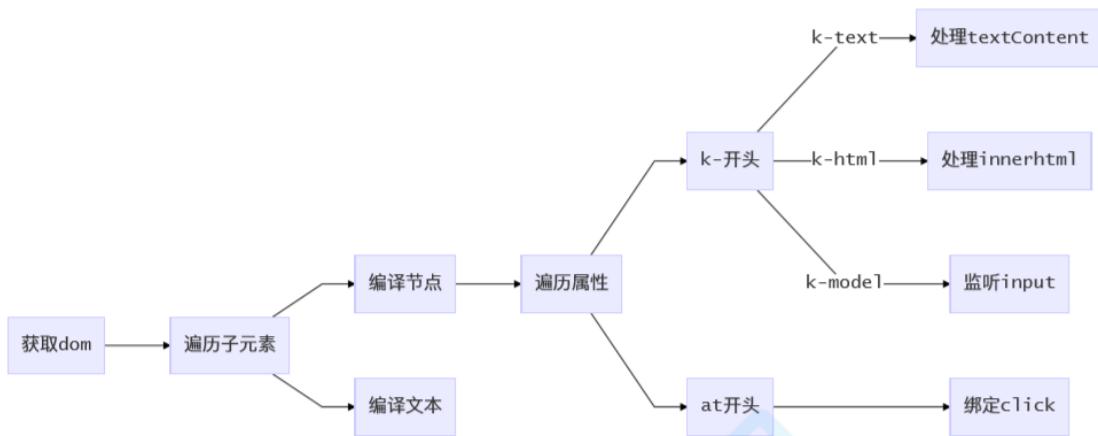
```

property/Dep/Watcher对应关系



compile源码

编译过程图



```

1 //遍历dom结构，解析指令和插值表达式
2 class Compile{
3   //el-带编译模板，vm-KVue实例
4   constructor(el,vm){
5     this.$vm=vm;
6     this.$el=document.querySelector(el);
7
8     //把模板中的内容移到片段操作
9     this.$fragment=this.node2Fragment(this.$el);
10    //执行编译
11    this.compile(this.$fragment);
12    //放回$el中
13    this.$el.appendChild(this.$fragment);
14  }
15
16  //
17  node2Fragment(el){
18    //创建片段
19    const fragment=document.createDocumentFragment();
20    //循环
21    let child;
22    while(child=el.firstChild){
23      fragment.appendChild(child);
24    }
25    return fragment;
26  }
27
28  compile(el){
  
```

```
29     const childNodes= e1.childNodes;
30     Array.from(childNodes).forEach(node=>{
31         if(node.nodeType==1){
32             //元素
33             console.log('编译元素'+node.nodeName);
34             this.compileElement(node)
35
36         }else if(this.isInter(node)){
37             //只关心{{xxxx}}
38             console.log('编译插值文本'+node.textContent)
39             this.compileText(node);
40         }
41
42         //递归子节点
43         if(node.children&&node.childNodes.length>0){
44             this.compile(node)
45         }
46     })
47 }
48 isInter(node){
49     return node.nodeType==3&&/^{\{.*\}}/.test(node.textContent)
50 }
51
52 compileElement(node){
53
54     // <div k-model='foo' k-text='test' @click="onClick"/></div>
55     //关心属性
56     const nodeAttrs=node.attributes;
57     Array.from(nodeAttrs).forEach(attr=>{
58         //规定: k-xxx='yyy'
59         const attrName=attr.name;//k-xxx
60         const exp=attr.value;//yyy
61         if(attrName.indexOf('k-')==0){
62
63             //指令
64             const dir=attrName.substring(2);//xxx
65             //执行
66             this[dir]&&this[dir](node,exp);
67         }
68
69         if(attrName.indexOf('@')==0){
70             const dir=attrName.substring(1);
71             this.eventHandler(node,this.$vm,exp,dir);
72         }
73     }
74
75 }
76
77 }
78 //文本替换
79 compileText(node){
80     console.log(RegExp.$1);
81
82
83     //表达式
84     const exp=RegExp.$1;
85     this.update(node,exp,"text")
86     // node.textContent=this.$vm[RegExp.$1];
```

```

87
88     }
89     html(node,vm,exp){
90         this.update(node,vm,exp,'html')
91     }
92     htmlUpdater(node,value){
93         node.innerHTML=value;
94     }
95
96     model(node,vm,exp){
97         this.update(node,vm,exp,'model');
98         node.addEventListener('input',(e)=>{
99             vm[exp]=e.target.value
100        });
101    }
102
103    modelUpdater(node,value){
104        node.value=value;
105    }
106    update(node,exp,dir){
107        const upator=this[dir+'Upator'];
108        upator&&upator(node,this.$vm[exp]);
109        //创建watcher实例，依赖收集完成了
110        new watcher(this.$vm,exp,function(value){
111            upator&&upator(node,value);
112        })
113    }
114
115    textUpdater(node,value){
116        node.textContent=value;
117    }
118    text(node,exp){
119        this.update(node,exp,'text');
120    }
121
122    //事件处理：给node添加事件监听， dir-事件名称
123    //通过vm.$options.methods[exp]获得回调函数
124    eventHandler(node,vm,exp,dir){
125        let fn=vm.$options.methods&&vm.$options.methods[exp];
126        if(dir&&fn){
127            node.addEventListener(dir,fn.bind(vm));
128        }
129    }
130 }

```

测试代码

```

1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4      <meta charset="UTF-8">
5      <meta name="viewport" content="width=device-width, initial-scale=1.0">
6      <meta http-equiv="X-UA-Compatible" content="ie=edge">
7      <title>Document</title>
8  </head>
9  <body>
10     <!-- <script src="kvue.js"></script>

```

```

11 <script>
12   const app= new KVue({
13     data:{foo:'foo',bar:{mua:'mua'}}
14   })
15   app.foo="aaa";
16   app.bar.mua="aaa";
17   console.log(app);
18 </script> -->
19
20 <div id="app">
21   <p>{{name}}</p>
22   <p k-text="name"></p>
23   <p>{{age}}</p>
24   <p>{{doubleAge}}</p>
25   <input type="text" k-model="name">
26   <button @click="changeName">哈哈</button>
27   <div k-html="html"></div>
28   <script src="kvue.js"></script>
29   <script>
30     const app=new KVue({
31       el:'#app',
32       data:{
33         name:'zyt',
34         age:1,
35         doubleAge:12,
36         html: '<button>这是一个按钮</button>'
37       },
38       created() {
39         console.log('开始');
40         setTimeout(()=>{
41           this.name='我是测试'
42         },1500)
43       },
44       methods:{
45         changeName(){
46           this.name='yw';
47           this.age=3;
48         }
49       }
50     })
51   </script>
52
53 </div>
54 </body>
55 </html>

```

Vue源码剖析

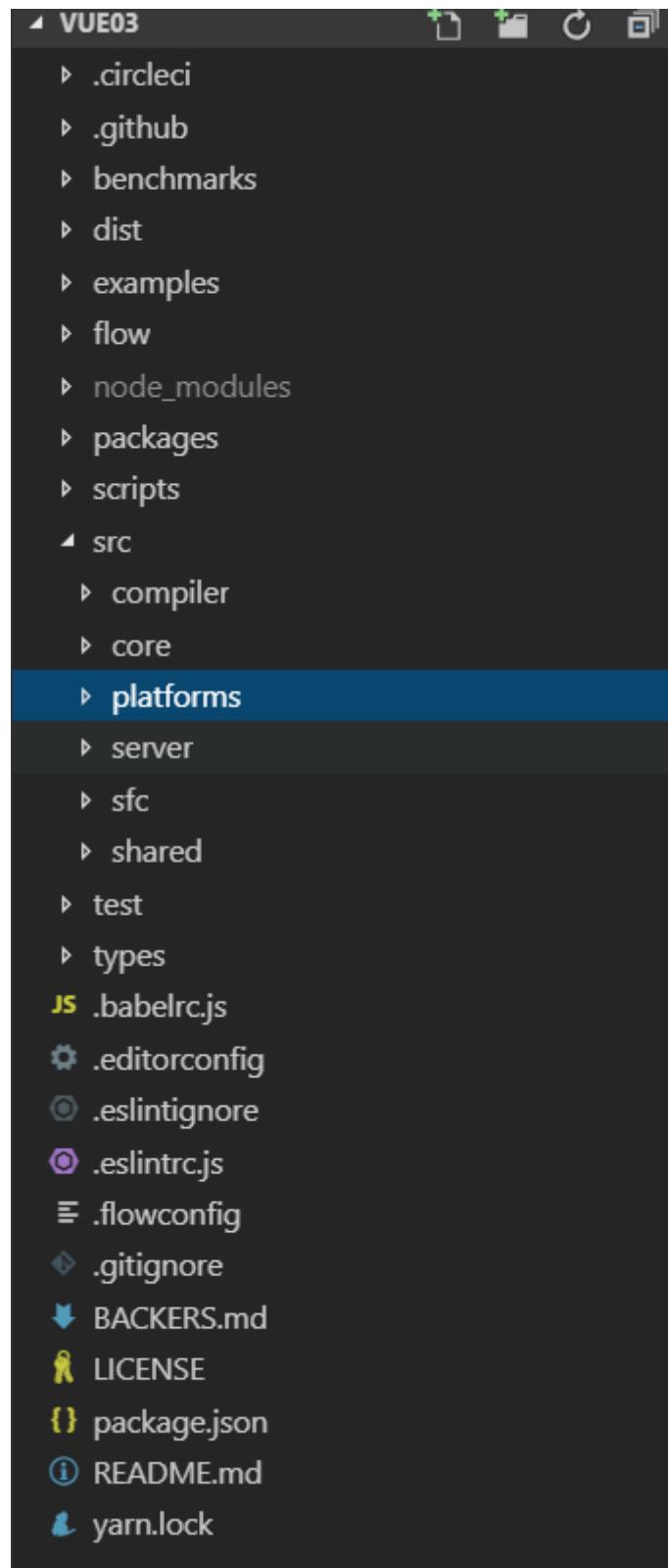
获取vue

项目地址: <https://github.com/vuejs/vue>

迁出地址: git clone <https://github.com/vuejs/vue.git>

当前版本号: 2.6.9

文件结构



调试vue项目的方式

- 安装依赖: npm i
- 安装打包工具: npm i rollup -g
- 修改package.json里面dev脚本:

```
1 | "dev": "rollup -w -c scripts/config.js --sourcemap --environment
TARGET:web-full-dev"
```

- 执行打包

```
1 | npm run dev
```

- 修改samples里面的文件引用新生成的vue.js

vue的初始化流程

vue响应机制逐行分析

整理启动顺序

```

1 new Vue() => _init() => $mount() => mountComponent() => updateComponent() /new
      watcher() => render() => update
2
3   $mount
4     -mountComponent
5       执行挂载，获取vdom并转换为dom
6     -new watcher()
7       创建组件渲染watcher
8     -updateComponent()
9       执行初始化或更新
10    -update()
11    初始化或更新，将传入vdom转换为dom，初始化执行的是dom创建操作
12    -render() src/core/instance/render.js
13    渲染组件，获取dom

```

platforms/web/entry-runtime-with-compiler.js

扩展默认\$mount方法：处理template或el选项

```

1 div#app
2
3 new Vue({
4   template: dom
5 }).$mount('#app')

```

src/platforms/web/runtime/index.js

安装web平台特有指令和组件

定义_patch__: 补丁函数，执行patching算法进行更新

定义\$mount:挂载vue实例到指定宿主元素(获取dom并替换宿主元素)

```

1 // install platform runtime directives & components
2 extend(Vue.options.directives, platformDirectives)
3 extend(Vue.options.components, platformComponents)
4 // install platform patch function
5 // 安装平台特有的patch函数，diff发生的地方
6 Vue.prototype._patch_ = inBrowser ? patch : noop
7 // public mount method
8 Vue.prototype.$mount = function (
9   el?: string | Element,
10   hydrating?: boolean
11 ): Component {

```

```
12 el = el && inBrowser ? query(el) : undefined
13 return mountComponent(this, el, hydrating)
14 }
```

- mountComponent core-instance/lifecycle.js

```
1 updateComponent = () => {
2     //首先执行vm._render(),返回vnode
3     //然后vnode,作为参数执行vm._update(),做dom更新
4     vm._update(vm._render(), hydrating)
5 }
6
7
8 //创建一个组件相关的watcher实例
9 //watcher/watcher选项会额外创建watcher
10 new watcher(vm, updateComponent, noop, {
11     before () {
12         if (vm._isMounted && !vm._isDestroyed) {
13             callHook(vm, 'beforeUpdate')
14         }
15     }
16 }, true /* isRenderwatcher */)
```

_render() src/core-instance/render.js

```
1 const { render, _parentVnode } = vm.$options
2 vnode = render.call(vm._renderProxy, vm.$createElement)
```

_update src/core-instance/lifecycle.js

```
1 if (!prevVnode) {
2     // initial render
3     //如果没有老vnode,说明在初始化
4     vm.$el = vm.__patch__(vm.$el, vnode, hydrating, false /* removeOnly */)
5 } else {
6     // updates
7     //更新周期直接diff,返回新的dom
8     vm.$el = vm.__patch__(prevVnode, vnode)
9 }
```

_patch src/platforms/web/runtime/patch.js

```
1 // the directive module should be applied last, after all
2 // built-in modules have been applied.
3 //扩展操作: 把通用模块和浏览器中特有模块合并
4 const modules = platformModules.concat(baseModules)
5
6 //工厂函数: 创建浏览器特有的patch函数, 这里主要解决跨平台问题
7 export const patch: Function = createPatchFunction({ nodeops, modules })
```

createPatchFunction src/core/vdom/patch.js

src/core/index.js

初始化全局api

具体如下：

```
1 | initGlobalAPI(Vue)
```

- initGlobalAPI
 - set
 - delete
 - nextTick
 - extend
 - ...

```
1 | Vue.set = set
2 | Vue.delete = del
3 | Vue.nextTick = nextTick
4 | initUse(Vue) // 实现 vue.use 函数
5 | initMixin(Vue) // 实现 vue.mixin 函数
6 | initExtend(Vue) // 实现 vue.extend 函数
7 | initAssetRegisters(Vue) // 注册实现 vue.component/directive/filter
```

src/core/instance/index.js

Vue构造函数定义

定义Vue实例API

```
1 | function Vue (options) {
2 |   if (process.env.NODE_ENV !== 'production' &&
3 |     !(this instanceof Vue))
4 |   ) {
5 |     warn('Vue is a constructor and should be called with the `new` keyword')
6 |   }
7 |   // 构造函数仅执行了 _init
8 |   this._init(options)
9 |
10
11  initMixin(Vue) // 实现上面的 _init 这个初始化方法
12  stateMixin(Vue) // 实现 $watch, $set, $delete
13  eventsMixin(Vue) // 实现 $emit, $on
14  lifecycleMixin(Vue) // 实现 _update, $forceUpdate, $destroy
15  renderMixin(Vue) // __render $nextTick
```

core/instance/init.js

创建组件实例，初始化其数据、属性、事件等

- initMixin

```
1 | initLifecycle(vm) // $parent
2 | initEvents(vm) // 事件监听器初始化
3 | initRender(vm) // vm.$createElement
4 | callHook(vm, 'beforeCreate')
5 | initInjections(vm) // resolve injections before data/props
6 | initState(vm) // 初始化 props, methods, data 等等
7 | initProvide(vm) // resolve provide after data/props
8 | callHook(vm, 'created')
```

- stateMixin

```

1  export function stateMixin (vue: Class<Component>) {
2    // flow somehow has problems with directly declared definition object
3    // when using Object.defineProperty, so we have to procedurally build up
4    // the object here.
5    const dataDef = {}
6    dataDef.get = function () { return this._data }
7    const propsDef = {}
8    propsDef.get = function () { return this._props }
9    if (process.env.NODE_ENV !== 'production') {
10      dataDef.set = function () {
11        warn(
12          'Avoid replacing instance root $data. ' +
13          'Use nested data properties instead.',
14          this
15        )
16      }
17      propsDef.set = function () {
18        warn(`$props is readonly.` , this)
19      }
20    }
21    Object.defineProperty(vue.prototype, '$data', dataDef)
22    Object.defineProperty(vue.prototype, '$props', propsDef)
23
24    vue.prototype.$set = set
25    vue.prototype.$delete = del
26
27    vue.prototype.$watch = function (
28      expOrFn: string | Function,
29      cb: any,
30      options?: Object
31    ): Function {
32      const vm: Component = this
33      if (isPlainObject(cb)) {
34        return createWatcher(vm, expOrFn, cb, options)
35      }
36      options = options || {}
37      options.user = true
38      const watcher = new Watcher(vm, expOrFn, cb, options)
39      if (options.immediate) {
40        try {
41          cb.call(vm, watcher.value)
42        } catch (error) {
43          handleError(error, vm, `callback for immediate watcher
44          "${watcher.expression}"`)
45        }
46      }
47      return function unwatchFn () {
48        watcher.teardown()
49      }
50    }

```

- eventsMixin

```
1  export function eventsMixin (vue: Class<Component>) {
```

```
2  const hookRE = /^hook:/  
3  Vue.prototype.$on = function (event: string | Array<string>, fn:  
Function): Component {  
    const vm: Component = this  
    if (Array.isArray(event)) {  
        for (let i = 0, l = event.length; i < l; i++) {  
            vm.$on(event[i], fn)  
        }  
    } else {  
        (vm._events[event] || (vm._events[event] = [])).push(fn)  
        // optimize hook:event cost by using a boolean flag marked at  
registration  
        // instead of a hash lookup  
        if (hookRE.test(event)) {  
            vm._hasHookEvent = true  
        }  
    }  
    return vm  
}  
19  
20  Vue.prototype.$once = function (event: string, fn: Function): Component {  
21      const vm: Component = this  
22      function on () {  
23          vm.$off(event, on)  
24          fn.apply(vm, arguments)  
25      }  
26      on.fn = fn  
27      vm.$on(event, on)  
28      return vm  
29 }  
30  
31  Vue.prototype.$off = function (event?: string | Array<string>, fn?:  
Function): Component {  
    const vm: Component = this  
    // all  
    if (!arguments.length) {  
        vm._events = Object.create(null)  
        return vm  
    }  
    // array of events  
    if (Array.isArray(event)) {  
        for (let i = 0, l = event.length; i < l; i++) {  
            vm.$off(event[i], fn)  
        }  
        return vm  
    }  
    // specific event  
    const cbs = vm._events[event]  
    if (!cbs) {  
        return vm  
    }  
    if (!fn) {  
        vm._events[event] = null  
        return vm  
    }  
    // specific handler  
    let cb  
    let i = cbs.length
```

```

57     while (i--) {
58       cb = cbs[i]
59       if (cb === fn || cb.fn === fn) {
60         cbs.splice(i, 1)
61         break
62       }
63     }
64   return vm
65 }
66
67 Vue.prototype.$emit = function (event: string): Component {
68   const vm: Component = this
69   if (process.env.NODE_ENV !== 'production') {
70     const lowerCaseEvent = event.toLowerCase()
71     if (lowerCaseEvent !== event && vm._events[lowerCaseEvent]) {
72       tip(
73         `Event "${lowerCaseEvent}" is emitted in component ` +
74         `${formatComponentName(vm)} but the handler is registered for
75         "${event}".
76         ` +
77         `Note that HTML attributes are case-insensitive and you cannot use
78         ` +
79         `v-on to listen to camelCase events when using in-DOM templates.
80         ` +
81         `You should probably use "${hyphenate(event)}" instead of
82         "${event}".
83       )
84     }
85   }
86   let cbs = vm._events[event]
87   if (cbs) {
88     cbs = cbs.length > 1 ? toArray(cbs) : cbs
89     const args = toArray(arguments, 1)
90     const info = `event handler for "${event}"`
91     for (let i = 0, l = cbs.length; i < l; i++) {
92       invokeWithErrorHandling(cbs[i], vm, args, vm, info)
93     }
94   }
95   return vm
96 }
97

```

- lifecycleMixin

```

1 export function lifecycleMixin (Vue: Class<Component>) {
2   Vue.prototype._update = function (vnode: VNode, hydrating?: boolean) {
3     const vm: Component = this
4     const prevEl = vm.$el
5     const prevVnode = vm._vnode
6     const restoreActiveInstance = setActiveInstance(vm)
7     vm._vnode = vnode
8     // Vue.prototype.__patch__ is injected in entry points
9     // based on the rendering backend used.
10    if (!prevVnode) {
11      // initial render
12      vm.$el = vm.__patch__(vm.$el, vnode, hydrating, false /* removeOnly
13    */

```

```
13     } else {
14         // updates
15         vm.$el = vm.__patch__(prevVnode, vnode)
16     }
17     restoreActiveInstance()
18     // update __vue__ reference
19     if (prevEl) {
20         prevEl.__vue__ = null
21     }
22     if (vm.$el) {
23         vm.$el.__vue__ = vm
24     }
25     // if parent is an HOC, update its $el as well
26     if (vm.$vnode && vm.$parent && vm.$vnode === vm.$parent._vnode) {
27         vm.$parent.$el = vm.$el
28     }
29     // updated hook is called by the scheduler to ensure that children are
30     // updated in a parent's updated hook.
31 }
32
33 vue.prototype.$forceUpdate = function () {
34     const vm: Component = this
35     if (vm._watcher) {
36         vm._watcher.update()
37     }
38 }
39
40 vue.prototype.$destroy = function () {
41     const vm: Component = this
42     if (vm._isBeingDestroyed) {
43         return
44     }
45     callHook(vm, 'beforeDestroy')
46     vm._isBeingDestroyed = true
47     // remove self from parent
48     const parent = vm.$parent
49     if (parent && !parent._isBeingDestroyed && !vm.$options.abstract) {
50         remove(parent.$children, vm)
51     }
52     // teardown watchers
53     if (vm._watcher) {
54         vm._watcher.teardown()
55     }
56     let i = vm._watchers.length
57     while (i--) {
58         vm._watchers[i].teardown()
59     }
60     // remove reference from data ob
61     // frozen object may not have observer.
62     if (vm._data.__ob__) {
63         vm._data.__ob__.vmCount--
64     }
65     // call the last hook...
66     vm._isDestroyed = true
67     // invoke destroy hooks on current rendered tree
68     vm.__patch__(vm._vnode, null)
69     // fire destroyed hook
70     callHook(vm, 'destroyed')
```

```

71  // turn off all instance listeners.
72  vm.$off()
73  // remove __vue__ reference
74  if (vm.$el) {
75      vm.$el.__vue__ = null
76  }
77  // release circular reference (#6759)
78  if (vm.$vnode) {
79      vm.$vnode.parent = null
80  }
81 }
82 }
```

- renderMixin

```

1  export function renderMixin (Vue: Class<Component>) {
2      // install runtime convenience helpers
3      installRenderHelpers(Vue.prototype)
4
5      Vue.prototype.$nextTick = function (fn: Function) {
6          return nextTick(fn, this)
7      }
8
9      Vue.prototype._render = function (): VNode {
10         const vm: Component = this
11         const { render, _parentVnode } = vm.$options
12
13         if (_parentVnode) {
14             vm.$scopedSlots = normalizeScopedSlots(
15                 _parentVnode.data.scopedSlots,
16                 vm.$slots,
17                 vm.$scopedSlots
18             )
19         }
20
21         // set parent vnode. this allows render functions to have access
22         // to the data on the placeholder node.
23         vm.$vnode = _parentVnode
24         // render self
25         let vnode
26         try {
27             // There's no need to maintain a stack because all render fns are
28             // called
29             // separately from one another. Nested component's render fns are
30             // called
31             // when parent component is patched.
32             currentRenderingInstance = vm
33             vnode = render.call(vm._renderProxy, vm.$createElement)
34         } catch (e) {
35             handleError(e, vm, `render`)
36             // return error render result,
37             // or previous vnode to prevent render error causing blank component
38             /* istanbul ignore else */
39             if (process.env.NODE_ENV !== 'production' && vm.$options.renderError)
40             {
41                 try {
42                     // ...
43                 } catch (e) {
44                     // ...
45                 }
46             }
47         }
48     }
49 }
```

```

39         vnode = vm.$options.renderError.call(vm._renderProxy,
40             vm.$createElement, e)
41         } catch (e) {
42             handleError(e, vm, `renderError`)
43             vnode = vm._vnode
44         }
45     } else {
46         vnode = vm._vnode
47     }
48 } finally {
49     currentRenderingInstance = null
50 }
51 // if the returned array contains only a single node, allow it
52 if (Array.isArray(vnode) && vnode.length === 1) {
53     vnode = vnode[0]
54 }
55 // return empty vnode in case the render function errored out
56 if (!(vnode instanceof VNode)) {
57     if (process.env.NODE_ENV !== 'production' && Array.isArray(vnode)) {
58         warn(
59             'Multiple root nodes returned from render function. Render
function ' +
60             'should return a single root node.',
61             vm
62         )
63     }
64     vnode = createEmptyVNode()
65 }
66 // set parent
67 vnode.parent = _parentVnode
68 return vnode
69 }
70 }
```

数据响应式

Vue一大特点是数据响应式，数据的变化会作用于UI而不用进行DOM操作。原理上来讲，是利用了JS语言特性Object.defineProperty()，通过定义对象属性setter方法拦截对象属性变更，从而将数值的变化转换为UI的变化。具体实现是在Vue初始化时，会调用initState，它会初始化data，props等，这里着重关注data初始化。

整体流程

initState (vm:Component) src/core/instance/state.js

初始化数据，包括props、methods、data、computed和watch

initData src/core/instance/state.js

```

1 //数据响应式
2 function initData (vm: Component) {
3     // observe data
4     observe(data, true /* asRootData */)
5 }
```

observe src/core/observer/index.js

```
1 | ob=new Observer(value)
2 | return ob
```

Observer

数据和对象响应化处理逻辑

```
1 /**
2  * walk through all properties and convert them into
3 * getter/setters. This method should only be called when
4 * value type is Object.
5 * 对象响应化
6 */
7 walk (obj: Object) {
8     const keys = Object.keys(obj)
9     for (let i = 0; i < keys.length; i++) {
10         defineReactive(obj, keys[i])
11     }
12 }
13
14 /**
15 * Observe a list of Array items.
16 * 数组元素响应化
17 */
18 observeArray (items: Array<any>) {
19     for (let i = 0, l = items.length; i < l; i++) {
20         observe(items[i])
21     }
22 }
23
24 }
```

defineReactive

数据拦截

```
1 /**
2  * Define a reactive property on an object.
3 */
4 export function defineReactive (
5     obj: Object,
6     key: string,
7     val: any,
8     customSetter?: ?Function,
9     shallow?: boolean
10 ) {
11     const dep = new Dep()
12
13     const property = Object.getOwnPropertyDescriptor(obj, key)
14     if (property && property.configurable === false) {
15         return
16     }
17
18     // cater for pre-defined getter/setters
19     const getter = property && property.get
20     const setter = property && property.set
```

```

21  if ((!getter || setter) && arguments.length === 2) {
22    val = obj[key]
23  }
24
25  //递归响应式处理
26  let childob = !shallow && observe(val)
27
28  //拦截
29  object.defineProperty(obj, key, {
30    enumerable: true,
31    configurable: true,
32    get: function reactiveGetter () {
33      const value = getter ? getter.call(obj) : val
34      if (Dep.target) {
35        //加入到dep管理watcher
36        dep.depend()
37        //若存在子对象
38        if (childob) {
39          childob.dep.depend()
40          if (Array.isArray(value)) {
41            //如果值是数组，要特殊处理
42            dependArray(value)
43          }
44        }
45      }
46      return value
47    },
48    set: function reactiveSetter (newVal) {
49      const value = getter ? getter.call(obj) : val
50      /* eslint-disable no-self-compare */
51      if (newVal === value || (newVal !== newVal && value !== value)) {
52        return
53      }
54      /* eslint-enable no-self-compare */
55      if (process.env.NODE_ENV !== 'production' && customSetter) {
56        customSetter()
57      }
58      // #7981: for accessor properties without setter
59      if (getter && !setter) return
60      if (setter) {
61        setter.call(obj, newVal)
62      } else {
63        val = newVal
64      }
65      //为什么需要递归? 'str' {foo: 'str'}
66      childob = !shallow && observe(newVal)
67      dep.notify()
68    }
69  })
70 }

```

Dep core/observer/dep.js

Dep负责管理一组watcher，包括watcher实例的增删及通知更新

```

1  export default class Dep {
2    static target: ?Watcher;

```

```

3  id: number;
4  subs: Array<watcher>;
5
6  constructor () {
7      this.id = uid++
8      this.subs = []
9  }
10
11 addSub (sub: watcher) {
12     this.subs.push(sub)
13 }
14
15 removeSub (sub: watcher) {
16     remove(this.subs, sub)
17 }
18
19 depend () {
20     if (Dep.target) {
21         Dep.target.addDep(this)
22     }
23 }
24
25 notify () {
26     // stabilize the subscriber list first
27     const subs = this.subs.slice()
28     if (process.env.NODE_ENV !== 'production' && !config.async) {
29         // subs aren't sorted in scheduler if not running async
30         // we need to sort them now to make sure they fire in correct
31         // order
32         subs.sort((a, b) => a.id - b.id)
33     }
34     for (let i = 0, l = subs.length; i < l; i++) {
35         subs[i].update()
36     }
37 }
38 }
39

```

Watcher src/core/observer/watcher.js

watcher和dep互相添加引用

```

1 /**
2  * Add a dependency to this directive.
3  */
4 addDep (dep: Dep) {
5     const id = dep.id
6     if (!this.newDepIds.has(id)) {
7         this.newDepIds.add(id)
8         this.newDeps.push(dep)
9         if (!this.depIds.has(id)) {
10             dep.addSub(this)
11         }
12     }

```

watcher更新逻辑：通常情况下执行queueWatcher, 执行异步更新

```

1  /**
2   * Subscriber interface.
3   * Will be called when a dependency changes.
4   */
5  update () {
6    /* istanbul ignore else */
7    if (this.lazy) {
8      this.dirty = true
9    } else if (this.sync) {
10      this.run()
11    } else {
12      queueWatcher(this)
13    }
14  }

```

queueWatcher src/core/observer/scheduler.js

推入队列，下个刷新周期执行批量任务，这是vue异步更新实现的关键

```

1 /**
2  * Push a watcher into the watcher queue.
3  * Jobs with duplicate IDs will be skipped unless it's
4  * pushed when the queue is being flushed.
5 */
6 export function queueWatcher (watcher: Watcher) {
7   const id = watcher.id
8   if (has[id] == null) {
9     has[id] = true
10    if (!flushing) {
11      queue.push(watcher)
12    } else {
13      // if already flushing, splice the watcher based on its id
14      // if already past its id, it will be run next immediately.
15      let i = queue.length - 1
16      while (i > index && queue[i].id > watcher.id) {
17        i--
18      }
19      queue.splice(i + 1, 0, watcher)
20    }
21    // queue the flush
22    if (!waiting) {
23      waiting = true
24
25      if (process.env.NODE_ENV !== 'production' && !config.async) {
26        flushSchedulerQueue()
27        return
28      }
29      nextTick(flushSchedulerQueue)
30    }
31  }
32}

```

nextTick将flushSchedulerQueue加入回调数组，启动timerFunc准备执行

```

1 export function nextTick (cb?: Function, ctx?: Object) {
2   let _resolve
3   callbacks.push(() => {

```

```

4   if (cb) {
5     try {
6       cb.call(ctx)
7     } catch (e) {
8       handleError(e, ctx, 'nextTick')
9     }
10    } else if (_resolve) {
11      _resolve(ctx)
12    }
13  })
14  if (!pending) {
15    pending = true
16    timerFunc()
17  }
18 // $flow-disable-line
19 if (!cb && typeof Promise !== 'undefined') {
20   return new Promise(resolve => {
21     _resolve = resolve
22   })
23 }
24 }

```

timerFunc指定了vue异步执行策略，根据执行环境，首选Promise,备选依次为：
MutationObserver,setImmediate,setTimeout

```

1  if (typeof Promise !== 'undefined' && isNative(Promise)) {
2    const p = Promise.resolve()
3    timerFunc = () => {
4      p.then(flushCallbacks)
5      // In problematic UIWebViews, Promise.then doesn't completely break, but
6      // it can get stuck in a weird state where callbacks are pushed into the
7      // microtask queue but the queue isn't being flushed, until the browser
8      // needs to do some other work, e.g. handle a timer. Therefore we can
9      // "force" the microtask queue to be flushed by adding an empty timer.
10     if (isIOS) setTimeout(noop)
11   }
12   isUsingMicroTask = true
13 } else if (!isIE && typeof MutationObserver !== 'undefined' && (
14   isNative(MutationObserver) ||
15   // PhantomJS and iOS 7.x
16   MutationObserver.toString() === '[object MutationObserverConstructor]'
17 )) {
18   // Use MutationObserver where native Promise is not available,
19   // e.g. PhantomJS, iOS7, Android 4.4
20   // (#6466 MutationObserver is unreliable in IE11)
21   let counter = 1
22   const observer = new MutationObserver(flushCallbacks)
23   const textNode = document.createTextNode(String(counter))
24   observer.observe(textNode, {
25     characterData: true
26   })
27   timerFunc = () => {
28     counter = (counter + 1) % 2
29     textNode.data = String(counter)
30   }
31   isUsingMicroTask = true
32 } else if (typeof setImmediate !== 'undefined' && isNative(setImmediate)) {

```

```
33 // Fallback to setImmediate.  
34 // Technically it leverages the (macro) task queue,  
35 // but it is still a better choice than setTimeout.  
36 timerFunc = () => {  
37   setImmediate(flushCallbacks)  
38 }  
39 } else {  
40   // Fallback to setTimeout.  
41   timerFunc = () => {  
42     setTimeout(flushCallbacks, 0)  
43   }  
44 }
```

数据响应式

数组比较特别，它的操作方法不会触发setter,需要特别处理

Observer

把修改过的数组拦截方法替换到当前数组对象上可以改变其行为

```
1 //判断当前value是不是数组或者object  
2 if (Array.isArray(value)) {  
3   if (hasProto) {  
4     protoAugment(value, arrayMethods)  
5   } else {  
6     copyAugment(value, arrayMethods, arrayKeys)  
7   }  
8   this.observeArray(value)  
9 } else {  
10  
11   //对象  
12   this.walk(value)  
13 }
```

arrayMethods src/core/observer/array.js

修改数组7个变更方法使其可以发送更新通知

```
1 const methodsToPatch = [  
2   'push',  
3   'pop',  
4   'shift',  
5   'unshift',  
6   'splice',  
7   'sort',  
8   'reverse'  
9 ]  
10  
11 /**
12  * Intercept mutating methods and emit events
13 */
14 methodsToPatch.forEach(function (method) {
15   // cache original method
16   const original = arrayProto[method]
17   //额外的事情是通知更新
18   def(arrayMethods, method, function mutator (...args) {
19     const result = original.apply(this, args)
```

```

20     const ob = this.__ob__
21     let inserted
22     switch (method) {
23       case 'push':
24       case 'unshift':
25         inserted = args
26         break
27       case 'splice':
28         inserted = args.slice(2)
29         break
30     }
31     if (inserted) ob.observeArray(inserted)
32     // notify change
33     ob.dep.notify()
34     return result
35   })
36 }
37

```

关于宏任务和微观任务

<https://jakearchibald.com/2015/tasks-microtasks-queues-and-schedules/>

属性更新是如何实现的?

patch.js src/core/vdom/patch.js

```

1 const hooks = ['create', 'activate', 'update', 'remove', 'destroy']
2
3
4 export function createPatchFunction (backend) {
5   let i, j
6   const cbs = {}
7   //传递进来的扩展模块和节点操作对象
8   const { modules, nodeOps } = backend
9
10  for (i = 0; i < hooks.length; ++i) {
11    //cbs['update']=[]
12    cbs[hooks[i]] = []
13    //modules:[attrs,klass,events,domProps,style,transition]
14    for (j = 0; j < modules.length; ++j) {
15      //modules[0]['update']是创建属性执行函数，其他hook以此类推
16      if (isDef(modules[j][hooks[i]])) {
17        cbs[hooks[i]].push(modules[j][hooks[i]])
18      }
19    }
20    /////cbs['update']:[fn,fn,fn,...]
21  }
22  function patchvnode(...){
23    if (isDef(data) && isPatchable(vnode)) {
24      //每次patch时先对属性做更新
25      for (i = 0; i < cbs.update.length; ++i) cbs.update[i](oldvnode, vnode)
26      if (isDef(i = data.hook) && isDef(i = i.update)) i(oldvnode, vnode)
27    }
28  }
29
30 }

```

模板编译

模板编译的主要目标是将模板(template)转换为渲染函数(render)

Vue2.0需要用到Vnode描述视图以及各种交互，手写显然不切实际，因此用户只需编写类似HTML代码的vue模板，通过编译器将模板转换为可返回VNode的render函数

体验模板编译

带编译的版本中，可以使用template或el的方式声明模板

```
1 <div>
2   <h1>
3     vue.js测试
4   </h1>
5   <p>
6     {{foo}}
7   </p>
8 </div>
9 <script>
10 //使用el方式
11   new Vue({
12     data:{foo:'foo'},
13     el:'#demo',
14   })
15 </script>! [体验模板编译] (G:\zy1\jest\vue图片\体验模板编译.png)
```

然后输出渲染函数

```
1 <script>
2   const app=new Vue({});
3   //输出render函数
4   console.log(app.$options.render)
5 </script>
```

输出结果大致如下：

```
function anonymous() {
  with (this) {
    return _c('div', { attrs: { "id": "demo" } }, [
      _c('h1', [_v("vue.js测试")]),
      _v(" "),
      _c('p', [_v(_s(foo))])
    ])
  }
}
```

元素节点使用createElement创建，别名_c

文本节点使用createTextVNode创建，别名_v

表达式先使用toString格式化，别名_s

模板编译过程

实现模板编译共有三个阶段：解析、优化和生成

解析-parse

解析器将模板解析为抽象语法树AST，只有将模板解析成AST后，才能基于它做优化或者生成代码字符串

调试查看得到的AST，/src/compiler/parser/index.js

```
▼ root: Object
  ► attrs: [...]
  ► attrsList: [...]
  ► attrsMap: {id: "demo"}
  ▼ children: Array(3)
    ► 0: {type: 1, tag: "h1", attrsList: Array(0), attrsMap: {...},
      ► 1: {type: 3, text: " ", start: 37, end: 42}
      ► 2: {type: 1, tag: "p", attrsList: Array(0), attrsMap: {...},
        length: 3
      ► __proto__: Array(0)
      end: 65
      parent: undefined
      plain: false
    ► rawAttrsMap: {id: {...}}
    start: 0
    tag: "div"
    type: 1
```

解析器内部分了HTML解析器、文本解析器和过滤解析器，最主要是HTML解析器，核心算法说明：

```
1 //src/compiler/parser/index.js
2 parseHTML(template, {
3   start(tag, attrs, unary){},//遇到开始标签的处理
4   end(){},//遇到结束标签的处理
5   chars(text){},//遇到文本标签的处理
6   comment(text){}/遇到注释标签的处理
7 })
```

优化-optimize

优化器的作用是在AST中找出静态子树并打上标记。静态子树是在AST中永远不变的节点，如纯文本节点。

标记静态子树的好处：

- 每次重新渲染，不需要为静态子树创建新节点
- 虚拟dom中的patch时，可以跳过静态子树

代码实现，src/compiler/optimizer.js-optimize

```

1 | export function optimize (root: ?ASTElement, options: CompilerOptions) {
2 |   if (!root) return
3 |   isStaticKey = genStaticKeysCached(options.staticKeys || '')
4 |   isPlatformReservedTag = options.isReservedTag || no
5 |   // first pass: mark all non-static nodes.
6 |   //找出静态节点并标记
7 |   markStatic(root)
8 |   // second pass: mark static roots.
9 |   //找出静态根节点并标记
10|   markStaticRoots(root, false)
11|

```

标记结束：

```

▼ ast: Object
  ► attrs: [{...}]
  ► attrsList: [{...}]
  ► attrsMap: {id: "demo"}
  ► children: (3) [{...}, {...}, {...}]
    end: 65
    parent: undefined
    plain: false
  ► rawAttrsMap: {id: {...}}
    start: 0
    static: false
    staticRoot: false
    tag: "div"
    type: 1

```

代码生成-generate

将AST转换成渲染函数中的内容，即代码字符串

generate方法生成渲染函数代码，src/compiler/codegen/index.js

```

1 | export function generate (
2 |   ast: ASTElement | void,
3 |   options: CompilerOptions
4 | ): CodegenResult {
5 |   const state = new CodegenState(options)
6 |   const code = ast ? genElement(ast, state) : '_c("div")'
7 |   return {
8 |     render: `with(this){return ${code}}`,
9 |     staticRenderFns: state.staticRenderFns
10|   }
11}
12
13 //生成的code长这样

```

```
14  `_c('div', {attrs:{'id':'demo'}}), [  
15    _c('h1', [_v("vue.js测试")]),  
16    _c('p', [_v(_s(foo))])  
17  ])`
```

v-if、v-for

着重观察几个结构性指令的解析过程

```
1 //解析v-if,src/compiler/parser/index.js  
2 function processIf (el) {  
3   const exp = getAndRemoveAttr(el, 'v-if')  
4   if (exp) {  
5     el.if = exp  
6     addIfCondition(el, {  
7       exp: exp,  
8       block: el  
9     })  
10  } else {  
11    if (getAndRemoveAttr(el, 'v-else') != null) {  
12      el.else = true  
13    }  
14    const elseif = getAndRemoveAttr(el, 'v-else-if')  
15    if (elseif) {  
16      el.elseif = elseif  
17    }  
18  }  
19}  
20  
21  
22 //代码生成, src/compiler/codegen/index.js  
23 function genIfConditions (  
24   conditions: ASTIfConditions,  
25   state: CodegenState,  
26   altGen?: Function,  
27   altEmpty?: string  
28 ): string {  
29   if (!conditions.length) {  
30     return altEmpty || '_e()'  
31   }  
32  
33   const condition = conditions.shift()  
34   if (condition.exp) {  
35     return `(${condition.exp})?${  
36       genTernaryExp(condition.block)  
37     }:${  
38       genIfConditions(conditions, state, altGen, altEmpty)  
39     }`  
40   } else {  
41     return `${genTernaryExp(condition.block)}'  
42   }  
43  
44 // v-if with v-once should generate code like (a)?_m(0):_m(1)  
45 function genTernaryExp (el) {  
46   return altGen  
47     ? altGen(el, state)  
48     : el.once
```

```
49     ? genOnce(el, state)
50     : genElement(el, state)
51   }
52 }
```

解析结果:

```
▶ attrsList: []
▶ attrsMap: {v-if: "foo"}
▶ children: [{...}]
  end: 46
  if: "foo"
  ▶ ifConditions: (2) [{...}, {...}]
  ▶ parent: {type: 1, tag: "div...
    plain: true
  ▶ rawAttrsMap: {v-if: {...}}
  start: 20
  tag: "h1"
  type: 1
```

生成结果:

```
1 "with(this){return _c('div',{attrs:{'id':'demo'}},[
2   (foo)?_c('h1',[_v(_s(foo))]):_c('h1',[_v("no title")]),
3   _v(" "),_c('abc')],1)"
```

插槽

组件编译的顺序是先编译父组件，再编译子组件

普通插槽是在父组件编译和渲染阶段生成vnodes,数组的作用域是父组件，子组件渲染的时候直接拿到这些渲染好的vnodes

作用域插槽，父组件在编译和渲染阶段并不会直接生成vnodes，而是在父节点保留一个scopedslots对象，存储着不同名称的插槽以及它们对应的渲染函数，只有在编译和渲染子组件阶段才会执行这个渲染函数生成vnodes，由于是在子组件环境执行的，所以对应的数据作用域是子组件实例

解析相关代码:

src/compiler/parser/index.js

```
1 // v-slot on <template>
2 // 处理<template v-slot:xxx="yyy">
3 const slotBinding = getAndRemoveAttrByRegex(el, slotRE)
4 if (slotBinding) {
5   const { name, dynamic } = getSlotName(slotBinding)//name是xxx
6   el.slotTarget = name//xxx赋值到slotTarget
7   el.slotTargetDynamic = dynamic
8   el.slotScope = slotBinding.value || emptySlotScopeToken // force it
  into a scoped slot for perf,yyy赋值到slotScope
```

```

9      }
10     //处理<slot>
11     if (el.tag === 'slot') {
12       el.slotName = getBindingAttr(el, 'name')//获取slot的name并赋值到slotName
13     }
14

```

生成相关

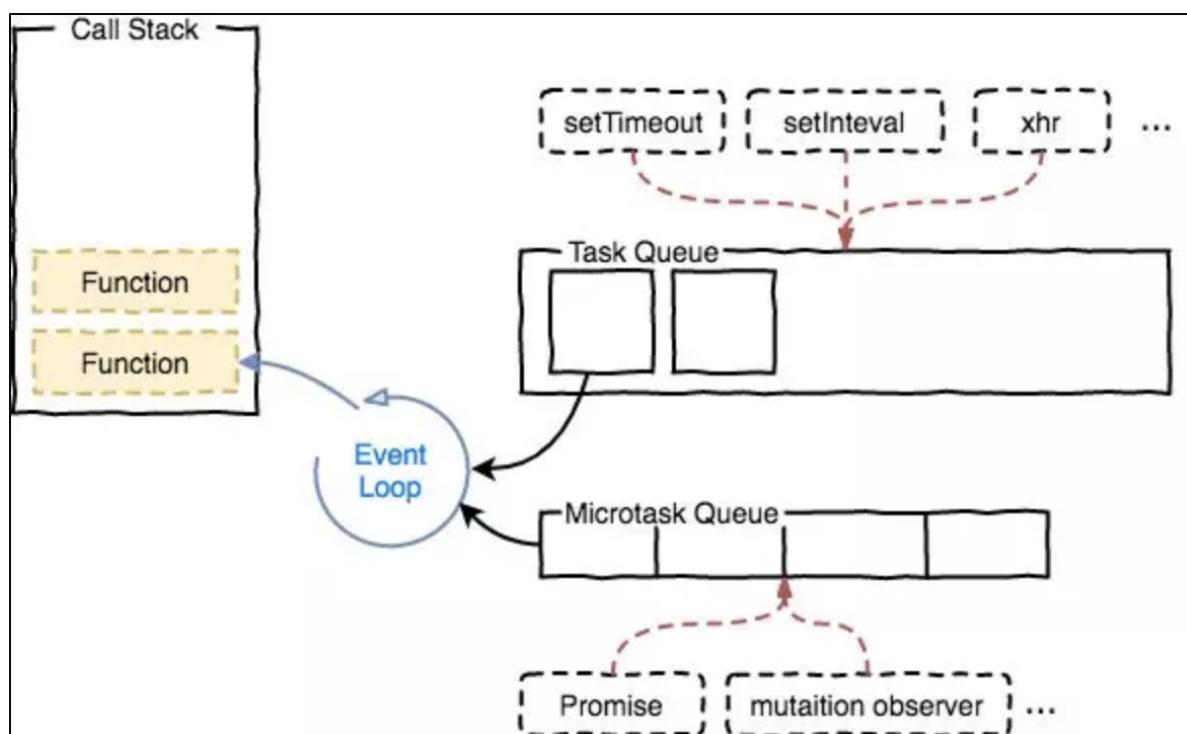
```

1 //genScopedSlot:这里把slotScope作为形参转化为工厂函数返回内容
2 const fn = `function(${slotScope}){` +
3   `return ${el.tag === 'template'
4     ? el.if && isLegacySyntax
5       ? `(${el.if})${genChildren(el, state) || 'undefined'}:undefined`
6         : genChildren(el, state) || 'undefined'
7       : genElement(el, state)
8     }` +
9
10 //reverse proxy v-slot without scope on this.$slots
11 const reverseProxy=slotScope?``:`,proxy:true`
12 return `key:${el.slotTarget||"default"},fn:${fn}${reverseProxy}```


```

异步更新队列(<https://www.processor.com/view/link/5e830387e4b0a2d87023890a#map>)

概念解析



- 事件循环Event Loop：浏览器为了协调事件处理、脚本执行、网络请求和渲染等任务而制定的工作机制
- 宏任务Task：代表一个个离散的、独立的工作单元。浏览器完成一个宏任务，在下一个宏任务执行开始前，会对面进行重新渲染。主要包括创建文档对象、解析HTML、执行主线JS代码以及各种事件如页面加载、输入、网络事件和定时器等。
- 微任务：微任务是更小的任务，是在当前宏任务执行结束后立即执行的任务。如果存在微任务，浏览器会清空微任务之后再重新渲染。微任务的例子有Promise回调函数、DOM变化等

体验一下 (https://jakearchibald.com/2015/tasks-microtasks-queues-and-schedules/?utm_source=html5weekly)

vue中的具体实现

Promise.then(flushQueue)



- 异步：只要侦听到数据变化，Vue 将开启一个队列，并缓冲在同一事件循环中发生的所有数据变更。
- 批量：如果同一个 watcher 被多次触发，只会被推入到队列中一次。去重对于避免不必要的计算和 DOM 操作是非常重要的。然后，在下一个的事件循环“tick”中，Vue 刷新队列执行实际工作。
- 异步策略：Vue 在内部对异步队列尝试使用原生的 Promise.then、MutationObserver 或 setImmediate，如果执行环境都不支持，则会采用 setTimeout 代替。

update() core\observer\watcher.js

dep.notify()之后watcher执行更新，执行入队操作

queueWatcher(watcher) core\observer\scheduler.js

执行watcher入队操作

nextTick(flushSchedulerQueue) core\util\next-tick.js

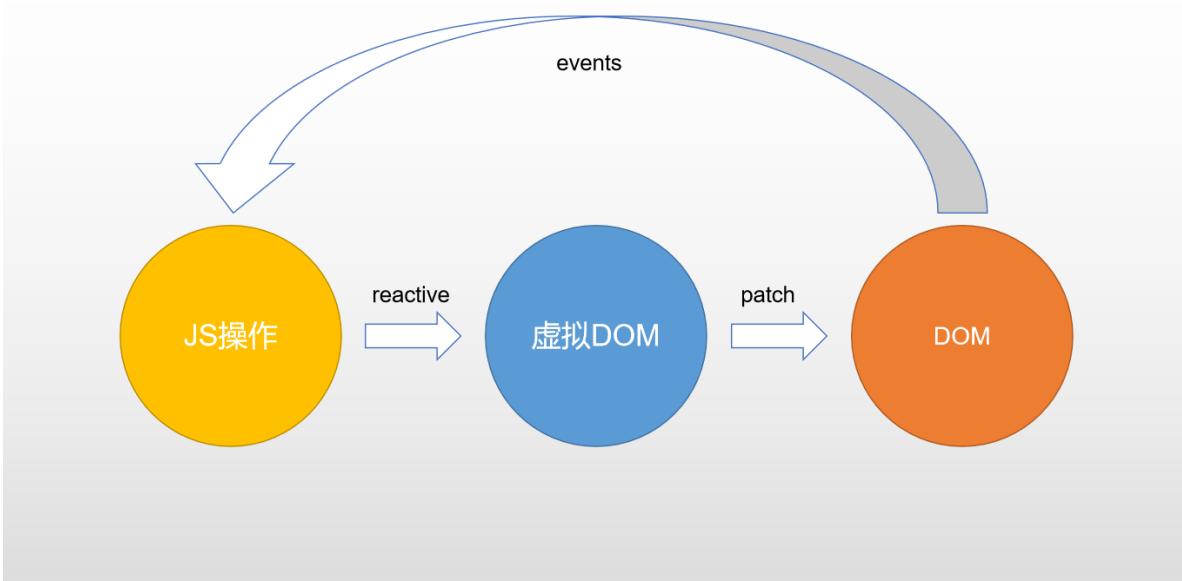
nextTick按照特定异步策略执行队列操作

虚拟DOM

概念

虚拟DOM (Virtual DOM) 是对DOM的JS抽象表示，它们是JS对象，能够描述DOM结构和关系。应用的各种状态变化会作用于虚拟DOM，最终映射到DOM上。

虚拟DOM的概念



体验虚拟DOM：

vue中虚拟dom基于snabbdom实现，安装snabbdom并体验

```
1  <!DOCTYPE html>
2  <html lang="en">
3
4  <head></head>
5
6  <body>
7      <div id="app"></div>
8      <!--安装并引入snabbdom-->
9      <script src="../../node_modules/snabbdom/dist/snabbdom.js"></script>
10     <script>
11         // 之前编写的响应式函数
12         function defineReactive(obj, key, val) {
13             Object.defineProperty(obj, key, {
14                 get() {
15                     return val
16                 },
17                 set(newVal) {
18                     val = newVal
19                     // 通知更新
20                     update()
21                 }
22             })
23         }
24         // 导入patch的工厂init, h是产生vnode的工厂
25         const { init, h } = snabbdom
26         // 获取patch函数
27         const patch = init([])
28         // 上次vnode, 由patch()返回
29         let vnode;
30         // 更新函数, 将数据操作转换为dom操作, 返回新vnode
31         function update() {
32             if (!vnode) {
33                 // 初始化, 没有上次vnode, 传入宿主元素和vnode
34                 vnode = patch(app, render())
```

```

35      }
36      else {
37          // 更新，传入新旧vnode对比并做更新
38          vnode = patch(vnode, render())
39      }
40  }
41  // 渲染函数，返回vnode描述dom结构
42  function render() {
43      return h('div', obj.foo)
44  }
45  // 数据
46  const obj = {}
47  // 定义响应式
48  defineReactive(obj, 'foo', '')
49  // 赋一个日期作为初始值
50  obj.foo = new Date().toLocaleTimeString()
51  // 定时改变数据，更新函数会重新执行
52  setInterval(() => {
53      obj.foo = new Date().toLocaleTimeString()
54  }, 1000);
55  </script>
56  </body>
57
58 </html>

```

优点

- 虚拟DOM轻量、快速：当它们发生变化时通过新旧虚拟DOM比对可以得到最小DOM操作量，配合异步更新策略减少刷新频率，从而提升性能

```
1 | patch(vnode, h('div', obj.foo))
```

- 跨平台：将虚拟dom更新转换为不同运行时特殊操作实现跨平台

```

1 <script src="../../node_modules/snabbdom/dist/snabbdom-style.js">
2 </script>
3 <script>
4     // 增加style模块
5     const patch = init([snabbdom_style.default])
6     function render() {
7         // 添加节点样式描述
8         return h('div', { style: { color: 'red' } }, obj.foo)
9     }
</script>

```

- 兼容性：还可以加入兼容性代码增强操作的兼容性

必要性

Vue 1.0中有细粒度的数据变化侦测，它是不需要虚拟DOM的，但是细粒度造成了大量开销，这对于大型项目来说是不可接受的。因此，Vue 2.0选择了中等粒度的解决方案，每一个组件一个watcher实例，这样状态变化时只能通知到组件，再通过引入虚拟DOM去进行比对和渲染。

整体流程

mountComponent() core/instance/lifecycle.js

渲染、更新组件

```
1 | updateComponent = () => {
2 |   // 实际调用是在lifecycleMixin中定义的_update和renderMixin中定义的_render
3 |   vm._update(vm._render(), hydrating)
4 | }
```

_render core/instance/render.js

生成虚拟dom

_update core\instance\lifecycle.js

update负责更新dom，转换vnode为dom

patch() platforms/web/runtime/index.js

patch是在平台特有代码中指定的

```
1 | Vue.prototype.__patch__ = inBrowser ? patch : noop
```

patch获取

patch是createPatchFunction的返回值，传递nodeOps和modules是web平台特别实现

```
1 | export const patch: Function = createPatchFunction({ nodeOps, modules })
```

platforms\web\runtime\node-ops.js

定义各种原生dom基础操作方法

platforms\web\runtime\modules\index.js

modules 定义了属性更新实现

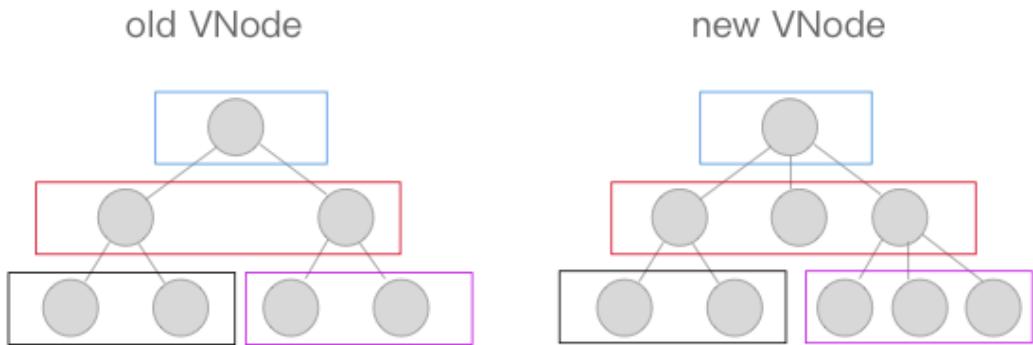
watcher.run() => componentUpdate() => render() => update() => patch()

patch实现

patch core\vdom\patch.js

首先进行树级别比较，可能有三种情况：增删改

- new Vnode不存在就删
- old Vnode不存在就增
- 都存在就执行diff执行更新



patchVnode

比较两个VNode，包括三种类型操作：属性更新、文本更新、子节点更新(比较的是虚拟dom，操作的是真实dom)

具体规则如下：

1. 新老节点均有children子节点，则对子节点进行diff操作，调用updateChildren
2. 如果新节点有子节点而老节点没有子节点，先清空老节点的文本内容，然后为其新增子节点。
3. 当新节点没有子节点而老节点有子节点的时候，则移除该节点的所有子节点。
4. 当新老节点都无子节点的时候，只是文本的替换。

测试，vdom.html

```

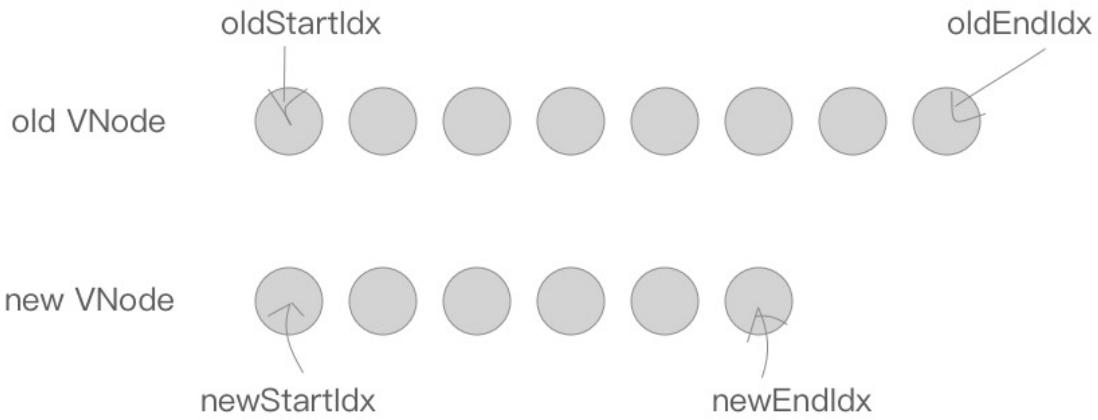
1 <div id="demo">
2   <h1>虚拟DOM</h1>
3   <p>{{foo}}</p>
4 </div>
```

```

1 // patchvnode过程分解
2 // 1.div#demo updateChildren
3 // 2.h1 updateChildren
4 // 3.text 文本相同跳过
5 // 4.p updateChildren
6 // 5.text setTextContent
```

updateChildren

updateChildren主要作用是用一种较高效的方式比对新旧两个VNode的children得出最小操作补丁。执行一个双循环是传统方式，vue中针对web场景特点做了特别的算法优化，我们看图说话：

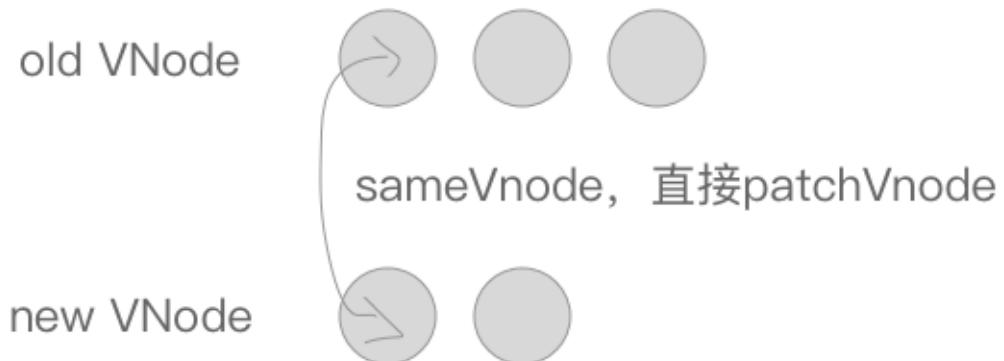


在新老两组VNode节点的左右头尾两侧都有一个变量标记，在遍历过程中这几个变量都会向中间靠拢。当 $oldStartIdx > oldEndIdx$ 或者 $newStartIdx > newEndIdx$ 时结束循环。

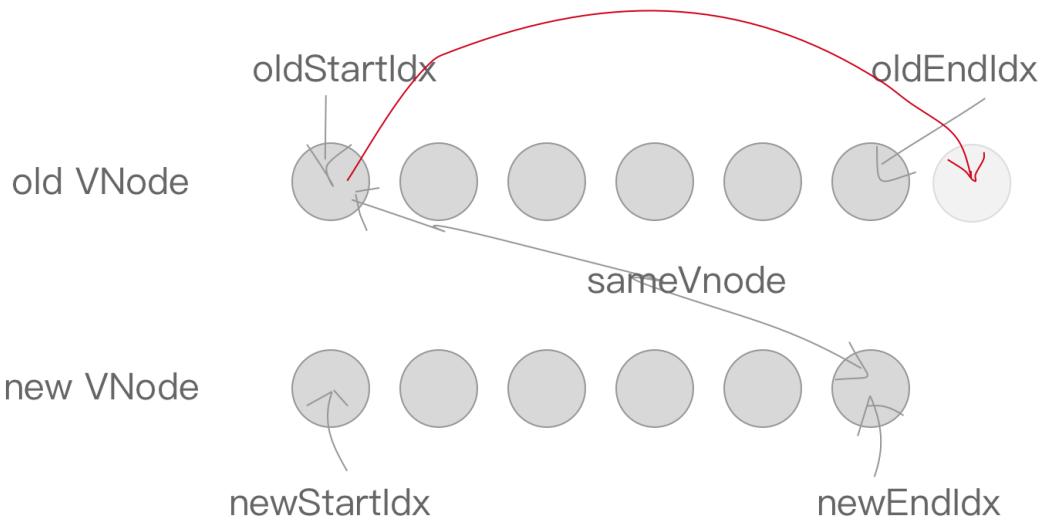
下面是遍历规则：

首先， $oldStartVnode$ 、 $oldEndVnode$ 与 $newStartVnode$ 、 $newEndVnode$ 两两交叉比较，共有4种比较方法。

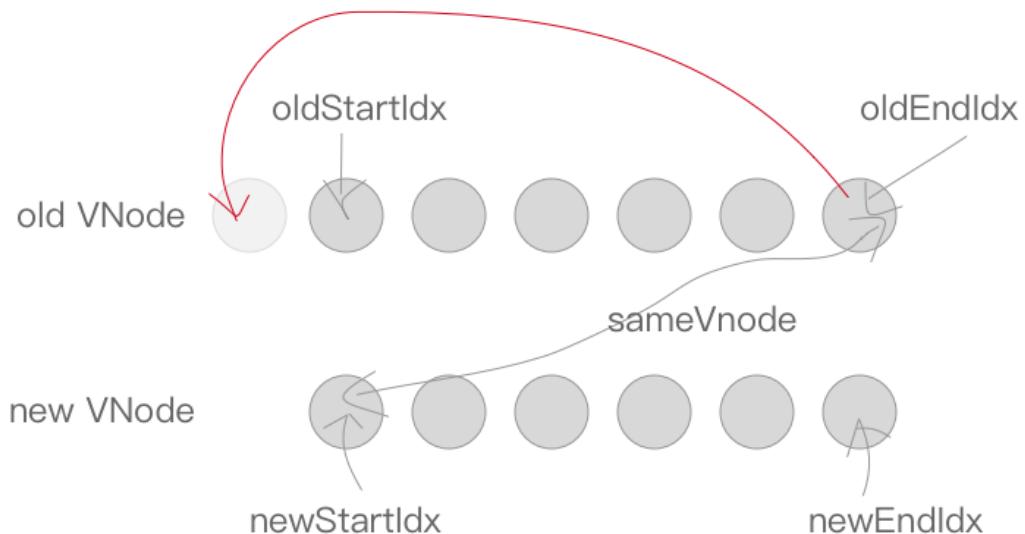
当 $oldStartVnode$ 和 $newStartVnode$ 或者 $oldEndVnode$ 和 $newEndVnode$ 满足 $sameVnode$ ，直接将该 VNode 节点进行patchVnode即可，不需再遍历就完成了一次循环。如下图，



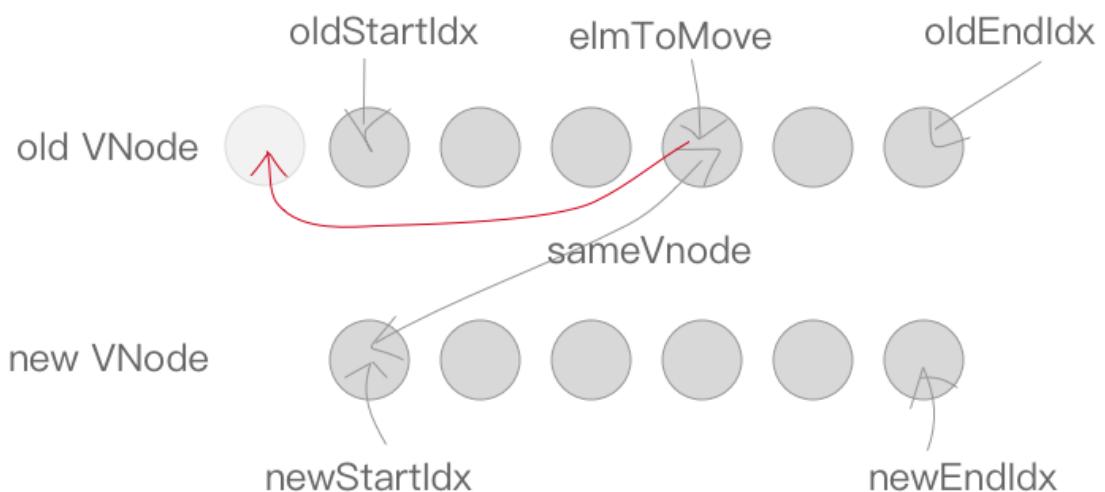
如果 $oldStartVnode$ 与 $newEndVnode$ 满足 $sameVnode$ 。说明 $oldStartVnode$ 已经跑到了 $oldEndVnode$ 后面去了，进行patchVnode的同时还需要将真实DOM节点移动到 $oldEndVnode$ 的后面



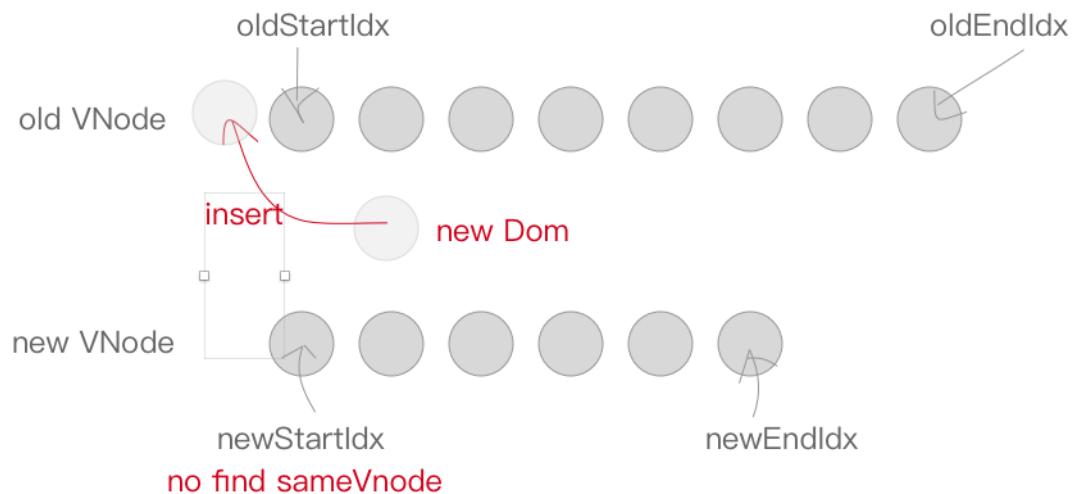
如果oldEndVnode与newStartVnode满足sameVnode，说明oldEndVnode跑到了oldStartVnode的前面，进行patchVnode的同时要将oldEndVnode对应DOM移动到oldStartVnode对应DOM的前面。



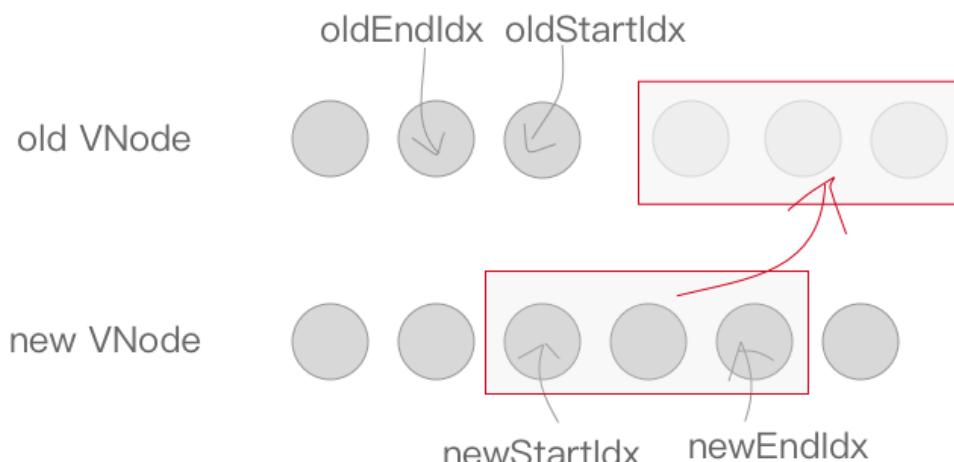
如果以上情况均不符合，则在old VNode中找与newStartVnode相同的节点，若存在执行patchVnode，同时将elmToMove移动到oldStartIdx对应的DOM的前面。



当然也有可能newStartVnode在old VNode节点中找不到一致的sameVnode，这个时候会调用createElm创建一个新的DOM节点。



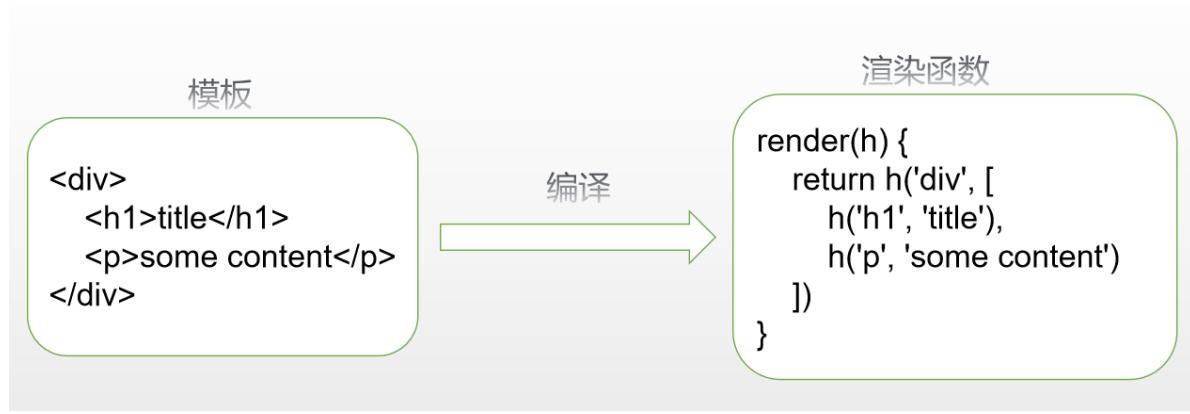
至此循环结束，但是我们还需要处理剩下的节点。当结束时 $oldStartIdx > oldEndIdx$ ，这个时候旧的VNode节点已经遍历完了，但是新的节点还没有。说明了新的VNode节点实际上比老的VNode节点多，需要将剩下的VNode对应的DOM插入到真实DOM中，此时调用addVnodes（批量调用createElm接口）。



但是，当结束时 $newStartIdx > newEndIdx$ 时，说明新的VNode节点已经遍历完了，但是老的节点还有剩余，需要从文档中删的节点删除。

模板编译

模板编译的主要目标是将模板(template)转换为渲染函数(render)



template=>render()

模板编译必要性

Vue 2.0需要用到VNode描述视图以及各种交互，手写显然不切实际，因此用户只需编写类似HTML代码的Vue模板，通过编译器将模板转换为可返回VNode的render函数。

体验模板编译

带编译器的版本中，可以使用template或el的方式声明模板

```
1 | (function anonymous(
2 | ) {
3 |   with(this){return _c('div',{attrs:{'id':'demo'}},[_c('h1',[_v("Vue模板编
4 | 译")]),_v(" "),_c('p',[_v(_s(foo))]),_v(" "),_c('comp')],1)}
5 | })
```

输出结果大致如下：

```
1 | (function anonymous() {
2 | with(this){return _c('div',{attrs:{'id':'demo'}},[
3 |   _c('h1',[_v("Vue模板编译")]),
4 |   _v(" "),
5 |   _c('p',[_v(_s(foo))]),
6 |   _v(" "),
7 |   _c('comp')],1)}
8 | })
```

- 1 元素节点使用createElement创建，别名_c
- 2 本文节点使用createTextVNode创建，别名_v
- 3 表达式先使用toString格式化，别名_s
- 4 其他渲染helpers: src\core\instance\render-helpers\index.js

整体流程

compileToFunctions

若指定template或el选项，则会执行编译，platforms\web\entry-runtime-with-compiler.js

编译过程

编译分为三步：解析、优化和生成，src\compiler\index.js

模板编译过程

实现模板编译共有三个阶段：解析、优化和生成

解析 - parse

解析器将模板解析为抽象语法树，基于AST可以做优化或者代码生成工作。

调试查看得到的AST，/src/compiler/parser/index.js，结构如下：

```
▼ root: Object
  ► attrs: [...]
  ► attrsList: [...]
  ► attrsMap: {id: "demo"}
  ▼ children: Array(3)
    ► 0: {type: 1, tag: "h1", attrsList: Array(0), attrsMap: {...},
    ► 1: {type: 3, text: " ", start: 37, end: 42}
    ► 2: {type: 1, tag: "p", attrsList: Array(0), attrsMap: {...},
      length: 3
    ► __proto__: Array(0)
    end: 65
    parent: undefined
    plain: false
  ► rawAttrsMap: {id: {...}}
    start: 0
    tag: "div"
    type: 1
```

解析器内部分了HTML解析器、文本解析器和过滤器解析器，最主要是HTML解析器

优化 - optimize

优化器的作用是在AST中找出静态子树并打上标记。静态子树是在AST中永远不变的节点，如纯文本节点。

标记静态子树的好处：

- 每次重新渲染，不需要为静态子树创建新节点
- 虚拟DOM中patch时，可以跳过静态子树

代码实现，src/compiler/optimizer.js - optimize

标记结束

```
▼ ast: Object
  ► attrs: [...]
  ► attrsList: [...]
  ► attrsMap: {id: "demo"}
  ► children: (3) [...], [...], [...]
    end: 65
    parent: undefined
    plain: false
  ► rawAttrsMap: {id: {...}}
    start: 0
    static: false
    staticRoot: false
    tag: "div"
    type: 1
```

代码生成-generate

将AST转换成渲染函数中的内容，即代码字符串。 generate方法生成渲染函数代码，
src/compiler/codegen/index.js

生成的code长这样

```
1 | `_c('div',{attrs:{'id':'demo'}},[
2 |   _c('h1',[_v("Vue.js测试")]),
3 |   _c('p',[_v(_s(foo))])
4 | )`
```

典型指令的实现：v-if、v-for

着重观察几个结构性指令的解析过程

解析v-if: parser/index.js

processIf用于处理v-if解析

解析结果：

```
▶ attrsList: []
▶ attrsMap: {v-if: "foo"}
▶ children: [...]
  end: 46
  if: "foo"
  ▶ ifConditions: (2) [...], [...]
  ▶ parent: {type: 1, tag: "div..."}
    plain: true
  ▶ rawAttrsMap: {v-if: {...}}
    start: 20
    tag: "h1"
    type: 1
  . . .
```

代码生成，codegen/index.js genIfConditions等用于生成条件语句相关代码

生成结果：

```
1 | "with(this){return _c('div',{attrs:{'id':'demo'}},[
2 |   (foo) ? _c('h1',[_v(_s(foo))]) : _c('h1',[_v("no title")]),
3 |   _v(" "),_c('abc'),1)}
```

解析v-for: parser/index.js

processFor用于处理v-for指令

解析结果：v-for="item in items" for:'items' alias:'item'

```

▼ el:
  type: 1
  tag: "b"
► attrsList: [{...}]
► attrsMap: {v-for: "s in arr", :key: "s"}
► rawAttrsMap: {v-for: {...}, :key: {...}}
► parent: {type: 1, tag: "div", attrsList: Arr...
► children: []
  start: 129
  end: 158
  for: "arr"
  alias: "s"
► __proto__: Object
exp: "s in arr"
► res: {for: "arr", alias: "s"}
this: undefined
Return value: undefined

```

代码生成，src\compiler\codegen\index.js：

genFor用于生成相应代码

生成结果

```

1 "with(this){return _c('div',{attrs:{'id':'demo'}},[_m(0),_v(" "),_e(_l((arr),function(s){return _c('b',{key:s},[_v(_s(s))])})),_v(" "),_c('comp')],2)}"
2
3
4

```

v-if, v-for这些指令只能在编译器阶段处理，如果我们要在render函数处理条件或循环只能使用if和for

```

1 Vue.component('comp', {
2   props: ['foo'],
3   render(h) { // 渲染内容跟foo的值挂钩，只能用if语句
4     if (this.foo=='foo') {
5       return h('div', 'foo')
6     }
7     return h('div', 'bar')
8   }
9 })

```

```
1 | (function anonymous(
2 | ) {
3 |   with(this){return _c('div',{attrs:{'id':'demo'}},[_m(0),_v(" "),_e(_s(foo)),_v(" "),_c('p'),_v(" ")])}
4 | }
5 | })
```

组件化机制

组件声明:Vue.component()

vue.js项目中一些最佳实践

项目安装

项目配置

```
npm i @vue/cli -g
```

```
vue create <项目名>
```

```
npm run eject(可以在vue项目中看到webpack的具体配置，具有不可逆)
```

在根目录新建vue.config.js

做一些基础配置：指定应用上下文、端口号、主页title

```
1 //vue.config.js
2 const port=7070;
3 const title='vue项目最佳实践';
4
5 module.exports={
6   publicPath: '/best-practice',
7   devServer:{
8     port
9   },
10  configureWebpack:{
11    //向index.html注入标题
12    name:title
13  }
14 }
15
16 //index.html
17 <title><%= webpackConfig.name %></title>
```

链式操作

比如：项目要使用icon,传统方案是图标字体（字体文件+样式文件），不便维护；svg方案采用svg-sprite-loader自动加载打包，方便维护

使用icon前先安装依赖:svg-sprite-loader

```
1 | npm i svg-sprite-loader -D
```

下载图标 (<https://www.iconfont.cn/>)，存入src/icons/svg中

修改规则和新增规则， vue.config.js

```
1 const port=7070;
2 const title='vue项目最佳实践';
3 const path=require('path')
4 //将传入的相对路径转换为绝对路径
5 function resolve(dir){
6     return path.join(__dirname,dir)
7 }
8
9 module.exports={
10     publicPath: '/best-practice',
11     devServer:{
12         port
13     },
14     configurewebpack:{
15         name:title
16     },
17     chainwebpack(config){
18         config.module.rule('svg')
19             .exclude.add(resolve('src/icons'));
20
21         //添加svg-sprite-loader
22         config.module.rule('icons')
23             .test(/\.\.svg$/)//设置test
24             .include.add(resolve('src/icons'))
25             .end()//add完上下文进入数组， 使用end回退
26             .use('svg-sprite-loader')//添加loader
27             .loader('svg-sprite-loader')//切换上下文到loader
28             .options({symbolId:'icon-[name]'});//指定选项
29             .end()
30     }
31 }
```

图标自动导入

```
1 //icons.index.js
2 import Icon from "@/components/Icon";
3 //图标自动导入
4 //利用webpack的require.context自动导入
5 //返回的req是只去加载svg目录中的模块的函数
6 const req=require.context('./svg',false,/\.svg$/)
7 req.keys().map(req)
8
9 //main.js
10 import './icons'
```

创建SvgIcon组件， ./components/SvgIcon.vue

```
1 <template>
2     <svg :class="svgClass" aria-hidden="true" v-on="$listeners">
3         <use:xlink:href="iconName"/>
4     </svg>
5 </template>
6 <script>
7 export default {
```

```

8     name:'SvgIcon',
9     props:{
10       iconClass:{  
11         type:String,  
12         required:true  
13       },  
14       className:{  
15         type:String,  
16         default:''  
17       }  
18     },  
19     computed: {  
20       iconName() {  
21         return `#icon-${this.iconClass}`  
22       },  
23       svgClass(){  
24         if(this.className){  
25           return `svg-icon ${this.className}`  
26         }else{  
27           return 'svg-icon'  
28         }  
29       }  
30     },  
31   }  
32 </script>  
33 <style scoped>  
34 .svg-icon{  
35   width:1em;  
36   height:1em;  
37   vertical-align: -0.15em;  
38   fill:currentColor;  
39   overflow: hidden;  
40 }  
41 </style>

```

注册，icons/index.js

```

1 import Vue from "vue";
2 import Icon from "@/components/Icon";
3 //图标自动导入
4 //利用webpack的require.context自动导入
5 //返回的req是只去加载svg目录中的模块的函数
6 const req=require.context('./svg',false,/\.svg$/)
7 req.keys().map(req)
8
9
10 //Icon组件全局注册
11 Vue.component('Icon',Icon)

```

使用，App.vue

```
1 | <svg-icon icon-class="qq"></svg-icon>
```

权限控制

路由分为两种，constantRoutes和asyncRoutes

定义路由,src/router/index.js

```
1 import Vue from 'vue'
2 import VueRouter from 'vue-router'
3 import Layout from "@/layout"; //布局页
4
5 Vue.use(VueRouter)
6
7 //通用页面
8 export const constRoutes=[
9   {
10     path: '/login',
11     component: ()=>import('@/views/Login'),
12     hidden:true //导航菜单忽略该项
13   },
14   {
15     path: '/',
16     component:Layout, //应用布局
17     redirect: '/home',
18     children:[{
19       path: 'home',
20       component: ()=>import(/*webpackChunkName:'home' */"@/views/home.vue"),
21       name: 'home',
22       meta: {
23         title: 'Home', //导航菜单项标题
24         icon: 'qq' //导航菜单项图标
25       }
26     }]
27   }
28 //权限页面
29 export const asyncRoutes=[
30   {
31     path: 'about',
32     component:Layout,
33     redirect: '/about/index',
34     children:[{
35       path: 'index',
36       component: ()=>import(/*webpackChunkName:'about' */"@/views/About.vue"),
37       name: 'about',
38       meta: {
39         title: 'About', //导航菜单项标题
40         icon: 'qq', //导航菜单项图标
41         roles:['admin','editor']
42       }
43     }]
44   }
45   export default new VueRouter({
46     mode: 'history',
47     base:process.env.BASE_URL,
48     routes:constRoutes
49   })
50 }
```

如果需求是每个页面的权限是后台管理系统配置的，而不是前端写死的，那该怎么办？

可以当用户登录后向后台请求可访问的路由表，从而动态生成可访问页面，操作和原来相同的，这里多了一步将后端返回路由表中组件名称和本地的组件映射步骤：

```
1 //前端组件名和组件映射表
2 const map={
3   login:require("login/index").default,//同步方式
4   login:()=>import("login/index")//异步方式
5 }
6
7 //服务端返回的map类似于
8 const serviceMap=[{
9   path:'login',component:'login',hidden:true
10 }]
11
12 //遍历serviceMap,将component替换为map[component],动态生成asyncRoutes
13 function mapComponent(route){
14   route.component=serviceMap[route.component];
15   if(route.children){
16     route.children=route.children.map(child=>mapComponent(child))
17   }
18   return route
19 }
```

路由守卫，创建./src/permission.js，并在main.js中引入

```
1 //做全局路由
2 import router from "./router";
3 import store from "./store/index2";
4 import { getToken } from "@/utils/auth";
5 const whiteList = ['login']
6
7 router.beforeEach(async(to, from, next) => {
8   const hasToken = getToken();
9   if (hasToken) {
10     if (to.path === '/login') {
11       //已登录
12       next({ path: '/' })
13     } else {
14       //已登录，获取用户角色
15       const hasRoles=store.getters.roles&&store.getters.roles.length>0;
16       if(hasRoles){
17         next();
18       }else{
19         try {
20           //先请求用户信息
21           const{ roles} =await store.dispatch('user/get')
22
23           //根据角色生成动态路由
24           const acRoutes=await
25           store.dispatch('permission/generateRoutes',roles)
26
27           //添加至router
28           router.addRoutes(acRoutes)
```

```

29         //重定向
30         next({...to,replace:true})
31     } catch (error) {
32         //出错需要重置令牌并重新登录(令牌过期, 网络错误等原因)
33         await store.dispatch('user/resetToken')
34         MessageChannel.error(error||'Has Error')
35         next(`/login?redirect=${to.path}`)
36     }
37 }
38 }
39 }
40 }
41 } else {
42     //用户无令牌
43     if (whiteList.indexOf(to.path) !== -1) {
44         next();
45     } else {
46
47         next(`/login?redirect=${to.path}`)
48     }
49 }
50 }
51 })

```

utils/auth.js

```

1 import Cookies from "js-cookie";
2
3 export const getToken=()=>{
4
5     return Cookies.get('token')
6 }
7 export const setToken=(token)=>{
8
9     return Cookies.set('token',token)
10 }
11 export const removeToken=()=>{
12
13     return Cookies.remove('token')
14 }

```

vuex相关模块实现，创建store/index.js

```

1 import Vue from "vue";
2 import Vuex from "vuex";
3 import user from "./user";
4 import permission from "./permission";
5
6 Vue.use(Vuex)
7
8 export default new Vuex.Store({
9     modules:{
10         user,
11         permission,
12     }
13 })

```

user模块：用户数据、用户登录等等,store/user.js

```
1 import { getToken, setToken, removeToken } from "@/utils/auth";
2 import { resolve } from "url";
3 import { reject } from "q";
4
5 const state = {
6     token: getToken(),
7     roles: [] // 用户角色
8 }
9
10 const mutations = {
11     SET_TOKEN: (state, token) => {
12         state.token = token;
13     },
14     SET_ROLES: (state, roles) => {
15         state.roles = roles;
16     }
17 }
18 const actions = {
19     // user login
20     login({ commit }, userInfo) {
21         const { username } = userInfo;
22         return new Promise((resolve, reject) => {
23             setTimeout(() => {
24                 if (username === 'admin' || username === 'zy1') {
25                     commit('SET_TOKEN', username);
26                     setToken(username);
27                     resolve();
28                 } else {
29                     reject('用户名、密码错误');
30                 }
31             }, 1000)
32         })
33     },
34     // get user info
35     getInfo({ commit, state }) {
36         return new Promise((resolve) => {
37             setTimeout(() => {
38                 const roles = state.token === 'admin' ? ['admin'] : ['editor'];
39                 commit('SET_ROLES', roles);
40                 resolve({ roles });
41             }, 1000)
42         })
43     },
44     // remove token
45     resetToken({ commit }) {
46         return new Promise((resolve) => {
47             commit("SET_TOKEN", '');
48             commit("SET_ROLES", []);
49             removeToken();
50             resolve();
51         })
52     }
53 }
54
55 export default {
```

```
56     namespaced:true,
57     state,
58     mutations,
59     actions
60 }
```

permission模板：路由配置信息、路由生成逻辑,store/permission.js

```
1 import {asyncRoutes,constRoutes} from "../router";
2 /**
3  * 根据路由meta.role确定是否当前用户拥有范文权限
4  * @role 用户拥有角色
5  * @route 待判定路由
6  */
7 function hasPermissission(roles,route){
8     //如果当前路由有roles字段则需判断用户访问权限
9     if(route.meta&&route.meta.roles){
10         //若用户拥有的角色中有包含在待判定路由角色表中的则拥有访问权
11         return roles.some(role=>route.meta.roles.includes(role))
12     }else{
13         //没有设置roles则无需判定即可访问
14         return true
15     }
16 }
17 /**
18  * 递归过滤AsyncRoutes路由表
19  * @routes 待过滤路由表,首次传入的就是AsyncRoutes
20  * @roles 用户拥有角色
21  *
22  */
23
24
25 export const filterAsyncRoutes=(routes,roles)=>{
26     const res=[];
27     routes.forEach(route=>{
28         //复制一份
29         const tmp={...route};
30         //如果用户有访问权则加入结果路由表
31         if(hasPermissission(roles,tmp)){
32             if(tmp.children){
33                 tmp.children=filterAsyncRoutes(tmp.children,roles)
34             }
35             res.push(tmp)
36         }
37     })
38     return res;
39 }
40
41
42 const state={
43     routes:[],//完整路由表
44     addRoutes:[]//用户可访问路由表
45 }
46
47 const mutations={
48     SET_ROUTES:(state,routes)=>{
49         state.addRoutes=routes;
```

```

50         state.routes=constRoutes.concat(routes);
51     }
52   }
53
54 const actions={
55   //路由生成: 在得到用户角色后第一时间调用
56   generateRoutes({commit},roles){
57     return new Promise(resolve=>{
58       let acsRoutes;
59       //用户是管理员则拥有完整访问权限
60       if(roles.includes("main")){
61         acsRoutes=asyncRoutes||[]
62       }else{
63         //否则需要根据角色做过滤处理
64         acsRoutes=filterAsyncRoutes(asyncRoutes,roles)
65       }
66       commit('SET_ROUTES',acsRoutes);
67       resolve(acsRoutes)
68     })
69   }
70 }
71
72 export default{
73   namespaced:true,
74   state,
75   mutations,
76   actions
77 }

```

getters编写,store/index2.js

```

1 <template>
2   <div class="app-wrapper">
3     <div class="main-container">
4       <router-view/>
5     </div>
6   </div>
7 </template>

```

用户登录页面, views/Login.vue

```

1 <template>
2   <div>
3     <h2>用户登录</h2>
4     <div>
5       <input type="text" v-model="username">
6       <button @click="login">登录</button>
7     </div>
8   </div>
9 </template>
10 <script>
11 export default {
12   data() {
13     return {
14       username: "admin"
15     }
16   }
17 }

```

```

16 },
17     methods: {
18         login() {
19             this.$store().dispatch("user/login", {username: this.username})
20             .then(()=>{
21                 this.$route.push({
22                     path: this.$route.query.redirect || "/"
23                 })
24             }).catch(err=>{
25                 alert(err)
26             })
27         }
28     },
29 }
30 </script>

```

按钮权限

有时权限控制粒度比较细，要精确到按钮、链接级别，此时可以封装一个指令，从而实现按钮界别权限控制

`inserted` 是自定义指令的一个钩子函数，该钩子函数在绑定指令的元素插入父节点时调用（仅保证父节点存在，但不一定已被插入文档中）。自定义指令还有其它钩子函数：

`binding` 是自定义指令钩子函数的一个参数（**只读**），该参数是一个对象，`value` 又是该对象的一个属性，表示指令的绑定值

```

1 //添加权限按钮, About.vue
2 <template>
3     <div class="about">
4         <h1>This is an about page</h1>
5         <button v-permission="['admin', 'editor']">editor button</button>
6         <button v-permission="['admin']">admin button</button>
7     </div>
8 </template>

```

创建v-permission指令，src/directive/permission.js

```

1 //完成一个指令，则指令通过传递进来的权限数组和当前用户角色数组过滤
2 //完成用户拥有要求的权限则可以看到，否则删除指令挂钩dom元素
3 import store from "@/store";
4 export default{
5     //el-挂载dom
6     //binding- v-perm="[]"
7     inserted(el,binding){
8         //获取值
9         const {value:permissionRoles}=binding;
10        //获取用户角色
11        const roles=store.getters.roles;
12        //合法性判断
13        if(permissionRoles&&permissionRoles instanceof Array
14        &&permissionRoles.length>0){
15            //判断用户角色中是否有要求
16            const hasPermission=roles.some(role=>{
17                return permissionRoles.includes(role)
18            });

```

```

19         //如果没有权限则删除当前dom
20         if(!hasPermisssion){
21             e1.parentNode&&e1.parentNode.removeChild(e1)
22         }
23     }else{
24         throw new Error('需指定数组类型权限,如v-permission')
25     }
26 }
27 }
28 }
```

注册指令,main.js

```

1 import vPermisssion from "./directive/permission";
2 Vue.directive("permission",vPermisssion)
```

该指令只能删除挂载指令的元素,对于那些额外生成的和指令无关的元素无能为力,比如:

```

1 <template>
2   <el-tabs>
3     <el-tab-plane label="用户管理" name="first" v-permission="['admin','editor']">
4       用户管理
5     </el-tab-plane>
6     <el-tab-plane label="配置管理" name="second" v-permission="['admin','editor']">
7       配置管理
8     </el-tab-plane>
9     <el-tab-plane label="角色管理" name="third" v-permission="['admin']">
10      角色管理
11    </el-tab-plane>
12    <el-tab-plane label="定时任务补偿" name="fourth" v-permission="['admin','editor']">
13      定时任务补偿
14    </el-tab-plane>
15  </el-tabs>
16 </template>
```

此时只能使用v-if来实现

```

1 <template>
2   <el-tabs>
3     <el-tab-pane v-if="checkPermission(['admin'])"/>
4   </el-tabs>
5 </template>
6 <script>
7 export default {
8   methods: {
9     checkPermission(permissionRoles) {
10       return roles.some(role=>{
11         return permissionRoles.includes(role)
12       })
13     }
14   },
15 }
16 </script>
```

导航菜单的生成

导航菜单是根据路由信息并结合权限判断而动态生成的。它需要支持路由的多级嵌套，所以这里要用到递归组件。

菜单结构是典型递归组件，利用之前实现的tree组件

数据准备，添加getter方法，store/index.js

```
1 | getters:{  
2 |     permission_routes:state=>state.permission.routes  
3 | }
```

修改sidmenu/index.vue

```
1 | <template>  
2 |     <div>  
3 |         <ul>  
4 |             <!-- 传递base-path是由于子路由是相对地址 -->  
5 |             <Item  
6 |                 :model="route"  
7 |                 v-for="route in permission_routes"  
8 |                 :key="route.path"  
9 |                 :base-path="route.path"  
10 |             >  
11 |  
12 |         </Item>  
13 |     </ul>  
14 |     </div>  
15 | </template>  
16 | <script>  
17 | import { mapGetters } from "vuex";  
18 | export default {  
19 |     components:{Item},  
20 |     components: {  
21 |         ...mapGetters(["permission_routes"]),  
22 |     },  
23 | }
```

Item.vue改造

```
1 | <template>  
2 |     <!-- 1.hidden存在则不显示 -->  
3 |     <li v-if="!model.hidden">  
4 |         <div @click="toggle">  
5 |             <!-- 2.设置icon才显示图标 -->  
6 |             <Icon  
7 |                 v-if="model.meta&&model.meta.icon"  
8 |                 :icon-class="model.meta.icon"  
9 |             ></Icon>  
10 |             <!-- 3.设置title:如果是folder仅显示标题和展开状态 -->  
11 |             <span v-if="isFolder">  
12 |                 <span v-if="model.meta && model.meta.title">  
13 |                     {{model.meta.title}}</span>  
14 |                     [{{open?'-':'+'}}]
```

```

14         </span>
15         <!-- 4.如果是叶子节点，显示为链接 --&gt;
16         &lt;template v-else&gt;
17             &lt;router-link
18                 v-else-if="model.meta&amp;&amp;model.meta.title"
19                 :to="resolvePath(model.path)"
20                 &gt;{{model.meta.title}}
21             &lt;/router-link&gt;
22         &lt;/template&gt;
23     &lt;/div&gt;
24     <!-- 5.子树设置base-path --&gt;
25     &lt;ul v-show="open" v-if="isFolder"&gt;
26         &lt;Item
27             class="item"
28             v-for="route in model.children"
29             :model="route"
30             :key="route.path"
31             :base-path="reslovePath(model.path)"
32             &gt;
33             &lt;/Item&gt;
34         &lt;/ul&gt;
35     &lt;/li&gt;
36 &lt;/template&gt;
37 &lt;script&gt;
38 import path from "path";
39 export default {
40     name:'Item',
41     props: {
42         //新增basePath保存父路由path
43         basePath: {
44             type: String,
45             default: ''
46         },
47     },
48     methods: {
49         //拼接子路由完整path
50         resolvePath(routePath) {
51             return path.resolve(this.basePath,routePath)
52         }
53     },
54 }
55 &lt;/script&gt;
</pre>

```

利用element-ui做一个比较好的导航

创建侧边栏组件，component/sidebar/index.vue

```

1 <template>
2     <div>
3         <el-scrollbar wrap-class="scrollbar-wrapper">
4             <el-menu
5                 :default-active="activeMenu"
6                 :background-color="variables.menuBg"
7                 :text-color="variables.menuText"
8                 :unique-opened="false"
9                 :active-text-color="variables.menuActiveText"
10                :collapse-transition="false"

```

```

11         mode="vertical"
12     >
13     <sidebar-item
14       v-for="route in permission_routes"
15         :key="route.path"
16         :item="route"
17         :base-path="resolvePath(child.path)"
18     />
19     </el-menu>
20   </el-scrollbar>
21
22 </div>
23 </template>
24 <script>
25 import SidebarItem from "./SidebarItem";
26 import {mapGetters} from "store";
27 export default {
28   components:{SidebarItem},
29   computed: {
30     ...mapGetters(['permission_routes']),
31     activeMenu(){
32       const route=this.$route;
33       const {meta,path}=route;
34       //默认激活项
35       if(meta.activeMenu){
36         return meta.activeMenu
37       }
38       return path
39     },
40     variables(){
41       return {
42         menuText: '#bfcbd9',
43         menuActiveText: '#409EFF',
44         menuBg: '#304156'
45       }
46     }
47   },
48 }
49 </script>

```

创建侧边菜单栏项目组件,components/Sidebar/SidebarItem.vue

```

1 <template>
2   <div v-if="!item.hidden" class="menu-wrapper">
3     <template v-if="hasOneShowingChild(item.children,item)&&
4       (!onlyOneChild.children||onlyOneChild.noShowingChildren)&&!item.alwaysShow">
5       <router-link v-if="onlyOneChild.meta"
6         :to="resolvePath(onlyOneChild.path)">
7         <el-menu-item :index="resolvePath(onlyOneChild.path)"
8           :class="{'submenu-title-noDropdown':!isNest}">
9           <Item :icon="onlyOneChild.meta.icon||"
10             (item.meta&&item.meta.icon)" :title="onlyOneChild.meta.title">
11             </Item>
12           </el-menu-item>
13         </router-link>
14       </template>
15     </div>
16   </template>
17 </SidebarItem>

```

```
11      <el-submenu v-else ref="subMenu" :index="resolvePath(item.path)">
12          popper-append-to-body>
13              <template v-slot:title>
14                  <Item v-if="item.meta" :icon="item.meta&&item.meta.icon"
15                      :title="item.meta.title"/>
16              </template>
17              <sidebar-item
18                  v-for="child in item.children"
19                      :key="child.path"
20                      :is-nest="true"
21                      :item="child"
22                      :base-path="resolvePath(child.path)"
23                      class="nest-menu">
24                  />
25              </el-submenu>
26          </div>
27      </template>
28      <script>
29          import path from "path";
30          import Item from './Item'
31          export default {
32              name:'SidebarItem',
33              components:{Item},
34              props: {
35                  item: {
36                      type: Object,
37                      default: ''
38                  },
39                  isNest:{
40                      type:Boolean,
41                      default:false
42                  },
43                  basePath:{
44                      type:String,
45                      default:''
46                  }
47              },
48              data() {
49                  this.onlyOneChild=null;
50                  return {}
51              },
52              methods: {
53                  hasOneShowingChild(children=[],parent) {
54                      const showingChildren=children.filter(item=>{
55                          if(item.hidden){
56                              return false
57                          }else{
58                              //如果只有一个子菜单时设置
59                              this.onlyOneChild=item;
60                              return true
61                          }
62                      })
63                      //当只有一个子路由，该路由默认显示
64                      if(showingChildren.length==1){
65                          return true;
66                      }
67                  }
68              }
69          }
70      </script>
71      <style>
72          .nest-menu{
73              margin-left: 20px;
74          }
75      </style>
76  
```

```

67         if(showingChildren.length==0){
68             this.onlyOnechild={...parent,path: '',noShowingChildren:true}
69             return true
70         }
71         return false
72     },
73     resolvePath(routePath){
74         return path.resolve(this basePath,routePath)
75     }
76 },
77 }
78 </script>

```

创建菜单标签组件，components/sidebar/item.vue

```

1 <script>
2 export default {
3     name:'MenuItem',
4     functional:true,
5     props: {
6         icon: {
7             type: String,
8             default: ''
9         },
10        title:{ 
11            type:String,
12            default:''
13        }
14    },
15    render(h,context){
16        const {icon,title}=context.props;
17        const vnodes=[];
18        if(icon){
19            vnodes.push(<Icon icon-class={icon}/>)
20        }
21        if(title){
22            vnode.push(<span slot="title">{title}</span>)
23        }
24        return vnodes
25    }
26 }
27 </script>

```

数据交互

封装request

安装axios:npm i axios -S

创建@/utils/request.js

```

1 import axios from 'axios';
2 import { MessageBox, Meassage } from 'element-ui'
3 import store from '@/store'
4 import { getToken } from '@/utils/auth'
5

```

```

6 //创建axios实例
7 const service = axios.create({
8   baseURL: process.env.VUE_APP_BASE_API, //url基础地址，解决不同数据源url变化问题
9   //withCredentials:true, //跨域时若要发送cookies需设置该选项
10  timeout: 5000 //超时
11 })
12
13 //请求拦截
14 service.interceptors.request.use(
15   config => {
16     //do something
17     if (store.getters.token) {
18       //设置令牌请求头
19       config.headers['Authorization'] = 'Bearer ' + getToken();
20     }
21     return config;
22   },
23   error => {
24     //请求错误预处理
25     return Promise.reject(error);
26   }
27 )
28
29 //响应拦截
30 service.interceptors.response.use(
31   //通过自定义code判定响应状态，也可以通过HTTP状态码判定
32   response => {
33     //仅返回数据部分
34     const res = response.data;
35     //code不为1则判定为一个错误
36     if (res.code !== 1) {
37       Meassage({
38         message: res.message || 'Error',
39         type: 'error',
40         duration: 5 * 1000
41       })
42
43       //假设：10008-非法令牌；10012-其他客户端已登录；10014-令牌过期
44       if (res.code === 10008 || res.code === 10012 || res.code ===
45       10014) {
46         //重新登录
47         MessageBox.confirm(
48           "登录状态异常，请重新登录",
49           '确认登录信息',
50           {
51             confirmButtonText: '重新登录',
52             cancelButtonText: '取消',
53             type: 'warning'
54           }
55         ).then(() => {
56           store.dispatch('user/resetToken').then(() => {
57             location.reload();
58           })
59         })
60       }
61       return Promise.reject(new Error(res.message || 'Error'))
62     } else {

```

```
62         return res;
63     }
64 },
65 error => {
66     Meassage({
67         message: error.message || 'Error',
68         type: 'error',
69         duration: 5 * 1000
70     })
71     return Promise.reject(error)
72 }
73 )
74
75 export default service;
```

设置VUE_APP_BASE_API环境变量，创建.env.development文件

```
1 #base api
2 VUE_APP_BASE_API='/dev-api'
```

添加token的getter方法

```
1 token:state=>state.user.token
```

测试代码,创建@api/user.js

```
1 import request from "@/utils/request";
2 export function login(data){
3     return request({
4         url:'/user/login',
5         method:'post',
6         data
7     })
8 }
9
10 export function getInfo(){
11     return request({
12         url:'/user/info',
13         method:'get'
14     })
15 }
```

数据mock

本地mock修改vue.config.js,给devServer添加相关代码

```
1 const bodyParser=require('body-parser');
2 module.exports={
3     devServer:{
4         port,
5         before:app=>{
6             app.use(bodyParser.json());
7             app.use(
8                 bodyParser.urlencoded({
9                     extended:true
10                })
11            )
12        }
13    }
14 }
```

```

10         })
11     );
12     app.post('/dev-api/user/login',(req,res)=>{
13         const {username}=req.body;
14         if(username==='admim'||username==='jerry'){
15             res.json({
16                 code:1,
17                 data:username
18             })
19         }else{
20             res.json({
21                 code:10204,
22                 message:'用户名或密码错误'
23             })
24         }
25     })
26
27     app.get('/dev-api/user/info',(req,res)=>{
28         const auth=req.headers['authorization'];
29         const roles=auth.split(' ')[1]==='admin'?['admin']:
30             ['editor'];
31         res.json({
32             code:1,
33             data:roles
34         })
35     })
36 },
37

```

调用接口, @/store/modules/user.js

```

1 import { getToken, setToken, removeToken } from "@/utils/auth";
2 import { login, getInfo } from '@/api/user'
3
4 const state = {
5     token: getToken(),
6     roles: [] // 用户角色
7 }
8
9 const mutations = {
10     SET_TOKEN: (state, token) => {
11         state.token = token;
12     },
13     SET_ROLES: (state, roles) => {
14         state.roles = roles;
15     }
16 }
17 const actions = {
18     // user login
19     login({commit}, userInfo) {
20         const {username} = userInfo;
21         // 调用并处理结果, 错误处理以拦截无需处理
22         return login(userInfo).then((res) => {
23             commit('SET_TOKEN', res.data);
24             setToken(res.data);
25         })
26     }
27 }

```

```

26      // return new Promise((resolve,reject)=>{
27      //     setTimeout(()=>{
28      //         if(username==='admin'||username==='zy1'){
29      //             commit('SET_TOKEN',username);
30      //             setToken(username);
31      //             resolve();
32      //         }else{
33      //             reject('用户名、密码错误')
34      //         }
35      //     },1000)
36     // })
37 },
38 //get user info
39 getInfo({commit,state}){
40     return getInfo(state.token).then(({data:roles})=>{
41         commit('SET_ROLES',roles);
42         return roles;
43     })
44     // return new Promise((resolve)=>{
45     //     setTimeout(()=>{
46     //         const roles=state.token==='admin'?['admin']:['editor']
47     //         commit('SET_ROLES',roles)
48     //         resolve({roles})
49     //     },1000)
50     // })
51 },
52 //remove token
53 resetToken({commit}){
54     return new Promise((resolve)=>{
55         commit("SET_TOKEN",'');
56         commit("SET_ROLES",[]);
57         removeToken();
58         resolve();
59     })
60 }
61 }
62
63 export default{
64     namespaced:true,
65     state,
66     mutations,
67     actions
68 }

```

线上mock:easy-mock

使用步骤:

- 1.登录easy-mock网站
- 2.创建一个项目
- 3.创建需要的接口
- 4.调用: 修改base_url,.env.development

```
1 #base api,'xxx'即创建的baseUrl地址
2 VUE_APP_BASE_API='xxx'
3
```

解决跨域

如果请求的接口在另一台服务器上，开发时则需要设置代理避免跨域问题

添加代理设置, vue.config.js

```
1 module.exports={
2   proxy:{
3     //代理/dev-api/user/login到http://127.0.0.1:3000/user/login
4     [process.env.VUE_APP_BASE_API]:{
5       target:'http://127.0.0.1:3000/',
6       changeOrigin:true,
7       pathRewrite:{//api/user/login=>/user/login
8         ['^'+process.env.VUE_APP_BASE_API]:''
9       }
10      }
11    },
12  },
13}
```

创建一个独立接口服务其, src/test-server/index.js

```
1 const express = require('express');
2 const app = express();
3 const bodyParser = require('body-parser')
4
5 app.use(bodyParser.json());
6 app.use(
7   bodyParser.urlencoded({
8     extended: true
9   })
10 );
11 app.post('/dev-api/user/login', (req, res) => {
12   const { username } = req.body;
13   if (username === 'admin' || username === 'jerry') {
14     res.json({
15       code: 1,
16       data: username
17     })
18   } else {
19     res.json({
20       code: 10204,
21       message: '用户名或密码错误'
22     })
23   }
24 })
25
26 app.get('/dev-api/user/info', (req, res) => {
27   const auth = req.headers['authorization'];
28   const roles = auth.split(' ')[1] === 'admin' ? ['admin'] : ['editor'];
29   res.json({
30     code: 1,
```

```
31     data: roles
32   })
33 }
34
35 app.listen(3000)
```

测试

测试分类

常见的开发流程里，都有测试人员，它们不管内部实现机制，只看最外层的输入输出，这种被我们称为黑盒测试。比如根据测试用例在页面上测试业务逻辑的正确性，这种测试称之为E2E测试

更负责一些的我们称之为集成测试，就是集合多个测试过的单元一起测试

还有一种测试叫做白盒测试，我们针对一些内部核心实现逻辑编写测试代码，称之为单元测试

编写测试代码的好处

- 提供描述组件行为的文档
- 节省手动测试的时间
- 减少研发新特性时产生的bug
- 改进设计
- 促进重构

准备工作

在vue中，推荐使用jest

新建vue项目时

- 选择特性unit Testing和E2E testing
- 单元测试解决方案选择：jest
- 端到端测试解决方案选择：Cypress

在已存在项目中集成

运行：vue add @vue/unit-jest和vue add @vue/e2e-cypress

编写单元测试

新建test/unit/kaikeba.spec.js,*spec.js是命名规范

```
1 function add(num1,num2){
2   return num1+num2
3 }
4
5 //测试套件 test suite
6 describe('kaikeba', ()=>{
7   //测试用例 test case
8   it('测试add函数', ()=>{
9     //断言assert
10    expect(add(1,3)).toBe(3);
11    expect(add(1,3)).toBe(4);
12    expect(add(-2,3)).toBe(1);
13  })
14})
```

执行单元测试

执行: npm run test:unit

断言API简介

- describe: 定义一个测试套件
- it: 定义一个测试用例
- expect: 断言的判断条件

测试vue组件

创建一个vue组件components/Kaikeba.vue

```
1 <template>
2   <div>
3     <span>{{message}}</span>
4     <button @click="changeMsg">点击</button>
5   </div>
6 </template>
7
8 <script>
9 export default {
10 data() {
11   return {
12     message: 'vue-text'
13   }
14 },
15 created () {
16   this.message='开课吧';
17 },
18 methods: {
19   changeMsg() {
20     this.message='按钮点击'
21   }
22 },
23 }
24 </script>
25
26 <style>
27
28 </style>
```

测试该组件

```
1 //导入vue.js和组件，进行测试
2 import Vue from "vue";
3 import KaikebaComp from "@/components/Kaikeba";
4 describe('KaikebaComp', ()=>{
5   //检查原始组件
6   it('由created生命周期', ()=>{
7     expect(typeof KaikebaComp.created).toBe('function')
8   })
9
10 })
```

```
11 //评估原始组件选项中的函数的结果
12 it('初始data是vue-text', ()=>{
13     //检查data函数存在性
14     expect(typeof KaikebaComp.data).toBe('function')
15
16     //检查data返回的默认值
17     const defaultData=KaikebaComp.data();
18     expect(defaultData.message).toBe('hello!')
19 }
20 })
```

检查mounted之后

```
1 it('mount之后测data是开课吧', ()=>{
2     const vm=new Vue(KaikebaComp).$mount();
3     expect(vm.message).toBe('开课吧');
4 })
```

用户点击，用测试角度去屑代码，vue提供了专门针对测试的@vue/test-utils

```
1 import { mount } from "@vue/test-utils";
2 import KaikebaComp from "@/components/Kaikeba";
3 describe('KaikebaComp', ()=>{
4     const wrapper=mount(KaikebaComp);
5     wrapper.find('button').trigger("click");
6     //测试数据变化
7     expect(wrapper.vm.message).toBe('按钮点击');
8     //测试html渲染结果
9     expect(wrapper.find('span').html()).toBe('<span>点击按钮</span>');
10    //等效方式
11    expect(wrapper.find('span').text()).toBe('按钮点击');
12});
```

测试覆盖率

jest自带覆盖率，修改jest.config.js

```
1 module.exports = {
2     preset: '@vue/cli-plugin-unit-jest',
3     'collectCoverage':true,
4     'collectCoverageFrom':['src/**/*.{js,vue}']
5 }
```

E2E测试

借用浏览器的能力，站在用户测试人员的角度，输入框，点击按钮等，完全模拟用户，这个和具体的框架关系不大，完全模拟浏览器的行为

运行E2E测试

npm run test:e2e

修改e2e/spec/test.js

```

1 // https://docs.cypress.io/api/introduction/api.html
2
3 describe('My First Test', () => {
4   it('visits the app root url', () => {
5     cy.visit('/')
6     // cy.contains('h1', 'Welcome to Your Vue.js App')
7     cy.contains('span', '开课吧')
8   })
9 })

```

服务端渲染SSR

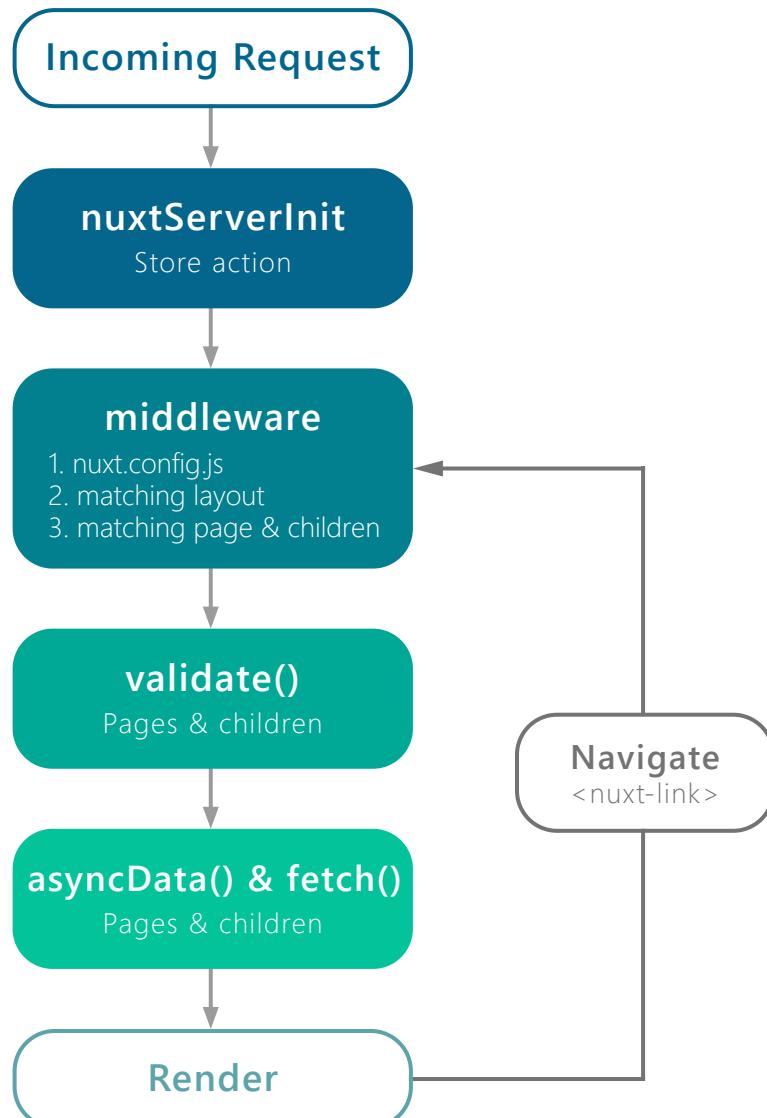
资源

1.vue ssr (<https://ssr.vuejs.org/zh/>)

2.nuxt.js (<https://nuxtjs.org/>)

知识点

nuxt渲染流程



nuxt安装

```
1 | npx create-nuxt-app <项目名>
```

目录结构

- assets: 资源目录 assets 用于组织未编译的静态资源如 LESS、SASS 或 JavaScript。
- components: 组件目录 components 用于组织应用的 Vue.js 组件。Nuxt.js 不会扩展增强该目录下 Vue.js 组件，即这些组件不会像页面组件那样有 asyncData 方法的特性。
- layouts: 布局目录 layouts 用于组织应用的布局组件。
- middleware: 中间件目录用于存放应用的中间件。
- pages: 页面目录 pages 用于组织应用的路由及视图。Nuxt.js 框架读取该目录下所有的 .vue 文件并自动生成对应的路由配置。
- plugins: 插件目录 plugins 用于组织那些需要在根 vue.js 应用 实例化之前需要运行的 Javascript 插件。
- static: 静态文件目录 static 用于存放应用的静态文件，此类文件不会被 Nuxt.js 调用 Webpack 进行构建编译处理。服务器启动的时候，该目录下的文件会映射至应用的根路径 / 下。
- store: 用于组织应用的 Vuex 状态树文件。Nuxt.js 框架集成了 Vuex 状态树 的相关功能配置，在 store 目录下创建一个 index.js 文件可激活这些配置。nuxt.config.js: 该文件用于个性化配置 Nuxt 应用。

路由生成

pages目录中所有*.vue文件自动生成应用的路由配置

查看.nuxt/router.js验证生成路由

导航

添加路由导航,layouts/default.vue

```
1 <template>
2   <div>
3     <nav>
4       <nuxt-link to="/">首页</nuxt-link>
5       <!-- 别名: n-link,NuxtLink,NLink -->
6       <n-link to="/admin">管理</n-link>
7       <NLINK to="/cart">购物车</NLINK>
8     </nav>
9     <!-- 页面内容占位符 -->
10    <nuxt />
11  </div>
12 </template>
```

禁用预加载:page not pre-fetched

商品列表, index.vue

```
1 <template>
2   <div>
3     <h2>商品列表</h2>
4     <ul>
5       <li v-for="good in goods" :key="good.id">
6         <nuxt-link :to=`/detail/${good.id}`>
7           <span>{{good.text}}</span>
8           <span>¥{{good.price}}</span>
```

```
9      </nuxt-link>
10     </li>
11   </ul>
12 </div>
13 </template>
14 <script>
15 export default {
16   data() {
17     return {
18       goods: [
19         { id: 1, text: "Web全栈架构师", price: 8999 },
20         { id: 2, text: "Python全栈架构师", price: 8999 }
21       ]
22     };
23   }
24 };
25 </script>
```

动态路由

以下划线作为前缀的.vue文件或目录会被定义为动态路由，如下面文件结构

```
1 pages/
2 --| detail/
3 ----| _id.vue
```

会生成如下路由配置：

```
1 {
2   path: "/detail/:id?",
3   component: _9c9d895e,
4   name: "detail-id"
5 }
```

如果detail/里面不存在index.vue，:id将被作为可选参数

嵌套路由

创建内嵌子路由，你需要添加一个.vue文件，同时添加一个与该文件同名的目录用来存放子视图组件

```
1 pages/
2 --| index/
3 ----| _id.vue
4 --| index.vue
```

会生成如下路由配置：

```
1 {
2   path: '/detail',
3   component: 'pages/detail.vue',
4   children: [
5     {path: ':id?', name: "detail-id"}
6   ]
7 }
```

测试代码，detail.vue

```
1 <template>
2 <div>
3 <h2>detail</h2>
4 <nuxt-child></nuxt-child>
5 </div>
6 </template>
```

nuxt-child等效于router-view

配置路由

要扩展 Nuxt.js 创建的路由，可以通过 router.extendRoutes 选项配置。例如添加自定义路由：

```
1 // nuxt.config.js
2 export default {
3   router: {
4     extendRoutes (routes, resolve) {
5       routes.push({
6         name: "foo",
7         path: "/foo",
8         component: resolve(__dirname, "pages/custom.vue")
9       });
10    }
11  }
12}
```

默认布局

查看layouts/default.vue

```
1 <template>
2 <nuxt/>
3 </template>
```

自定义布局

创建空白布局页面layouts/blank.vue,用于login.vue

```
1 <template>
2   <div>
3     <nuxt />
4   </div>
5 </template>
```

页面pages/login.vue使用自定义布局

```
1 export default{
2   layout:'blank'
3 }
```

自定义错误页面

创建layouts/error.vue

```
1 <template>
2   <div class="container">
3     <h1 v-if="error.statusCode === 404">页面不存在</h1>
4     <h1 v-else>应用发生错误异常</h1>
5     <nuxt-link to="/">首页</nuxt-link>
6   </div>
7 </template>
8
9 <script>
10 export default {
11   props: ['error'],
12   layout: 'blank'
13 }
14 </script>
15
16 <style>
```

页面

页面组件就是vue组件，只不过Nuxt.js为这些组件添加了一些特殊的配置项给首页添加标题和meta,index.vue

```
1 export default {
2   head() {
3     return {
4       title: "课程列表",
5       // vue-meta利用hid确定要更新meta
6       meta: [{ name: "description", hid: "description", content: "set page
7       meta" }],
8       link: [{ rel: "favicon", href: "favicon.ico" }],
9     };
10   },
11 }
```

更多页面配置项：

属性名 描述
asyncData :最重要选项, 支持 异步数据处理 , 该方法的第一个参数为当前页面组件的上下文对象 。 fetch与 asyncData 方法类似，用于在渲染页面之前获取数据填充应用的状态树（store）。不同的是 fetch 方法不会设置组件的数据。详情请参考 [关于fetch方法的文档](#) 。 head 配置当前页面的 Meta 标签, 详情参考 [页面头部配置API](#) 。 layout 指定当前页面使用的布局（layouts 根目录下的布局文件）。详情请参考 [关于布局 的文档](#) 。 loading如果设置为 false，则阻止页面自动调用 this.nuxt.loading.finish() 和this.nuxt.loading.start() ,您可以手动控制它,请看例子 ,仅适用于在nuxt.config.js中设置 loading 的情况下。请参考API配置 loading 文档 。 transition 指定页面切换的过渡动效, 详情请参考 [页面过渡动效](#) 。 scrollToTop 布尔值, 默认: false 。 用于判定渲染页面前是否需要将当前页面滚动至顶部。这个配置用于 嵌套路由 的应用场景。 validate 校验方法用于校验 动态路由 的参数。 middleware 指定页面的中间件, 中间件会在页面渲染之前被调用, 请参考 [路由中间件](#) 。

异步数据

asyncData方法使得我们可以在设置组件数据之前异步获取或处理数据

范例:获取商品数据

接口准备

- 安装依赖:npm i koa-router koa-bodyparser -S
- 创建接口文件,server/api.js

```
1 const Koa=require('koa');
2 const app=new Koa();
3 const bodyParser=require('koa-bodyparser');
4 const router=require('koa-router')({prefix:'/api'})
5
6 //设置cookie加密秘钥
7 app.keys=['some secret','another secret'];
8 const goods=[{
9     id:1,text:'web全栈架构师',price:1000
10 },{
11     id:2,text:'Python架构师',price:1000
12 }]
13
14 router.get('/goods',ctx=>{
15     ctx.body={
16         ok:1,
17         goods
18     }
19 })
20 router.post('/login',ctx=>{
21     ctx.body={
22         ok:1,
23         data:goods.find(good=>good.id==ctx.query.id)
24     }
25 })
26
27 router.post('/login',ctx=>{
28     const user=ctx.request.body;
29     if(user.username==='zy1'&&user.password==='123'){
30         //将token存入cookie
31         const token='a mock token';
32         ctx.cookies.set('token',token);
33         ctx.body={ok:1,token}
34     }else{
35         ctx.body={ok:0}
36     }
37 })
38
39
40 //解析post数据并注册路由
41 app.use(bodyParser());
42 app.use(router.routes());
43 app.listen(8080,()=>console.log('api服务已启动'))
```

整合axios

安装@nuxt/axios模块:npm i @nuxtjs/axios -S

win10有时需管理员权限启动vscode

配置:nuxt.config.js

```
1 modules:["@nuxtjs/axios"],  
2   axios:{  
3     proxy:true  
4   },  
5   proxy:{  
6     '/api':'http://localhost:8080'  
7   },
```

测试代码: 获取商品列表,index.vue

```
1 <script>  
2 export default{  
3   asyncData({$axios,error}) {  
4     //前后端都会执行, 时间点在beforeCreate之前  
5     //传递一个上下文对象  
6     //会和data合并  
7     //不能是用this访问组件实例  
8     try {  
9       const {ok,goods}=await $axios.$get('/api/goods');  
10      if(ok){  
11        return {goods}  
12      }  
13      //错误处理  
14      error({statusCode:400,message:'数据查询失败'})  
15    } catch (error) {  
16      error(error)  
17    }  
18  }  
19 </script>
```

测试代码: 获取商品详情, /index/_id.vue

```
1 <template>  
2   <div>  
3     <pre v-if="goodInfo">{{goodInfo}}</pre>  
4   </div>  
5 </template>  
6 <script>  
7 export default {  
8   async asyncData({ $axios, params, error }) {  
9     if (params.id) {  
10       // asyncData中不能使用this获取组件实例  
11       // 但是可以通过上下文获取相关数据  
12       const { data: goodInfo } = await $axios.$get("/api/detail", { params  
});  
13       if (goodInfo) {  
14         return { goodInfo };  
15       }  
16       error({ statusCode: 400, message: "商品详情查询失败" });  
17     }  
18   }  
19 </script>
```

```
17     } else {
18         return { goodInfo: null };
19     }
20 }
21 };
22 </script>
```

中间件

中间件会在一个页面或一组页面渲染之前运行我们定义函数，常用于权限控制、校验等任务

范例代码:管理员页面保护，创建middleware/auth.js

```
1 export default function({route, redirect, store}){
2     //如果没有token，则重定向到login
3     if(!store.state.user.token){
4         redirect('/login?redirect=' + route.path)
5     }
6 }
```

注册中间件， admin.vue

```
1 <script>
2     export default{
3         middleware: ['auth']
4     }
5 </script>
```

全局注册：将会对所有页面起作用， nuxt.config.js

```
1 router: {
2     middleware: ['auth']
3 },
```

状态管理vuex

应用根目录下如果存在 store 目录， Nuxt.js将启用vuex状态树。定义各状态树时具名导出 state,mutations, getters, actions即可。

范例:用户登录及登录状态保存， 创建store/user.js

```
1 export const state = () => ({
2     token: ""
3 });
4 export const mutations = {
5     init(state, token) {
6         state.token = token;
7     }
8 };
9 export const getters = {
10     isLoggedIn(state) {
11         return !!state.token;
12     }
13 };
14 export const actions = {
```

```

15  login({ commit, getters }, u) {
16    return this.$axios.$post("/api/login", u).then(({ token }) => {
17      if (token) {
18        commit("init", token);
19      }
20      return getters.isLoggedIn;
21    });
22  };
23};
24

```

登录页面逻辑, login.vue

```

1 <template>
2   <div>
3     <h2>用户登录</h2>
4     <el-input v-model="user.username"></el-input>
5     <el-input type="password" v-model="user.password"></el-input>
6     <el-button @click="onLogin">登录</el-button>
7   </div>
8 </template>
9 <script>
10 export default {
11   data() {
12     return {
13       user: {
14         username: "",
15         password: ""
16       }
17     };
18   },
19   methods: {
20     onLogin() {
21       this.$store.dispatch("user/login", this.user).then(ok => {
22         if (ok) {
23           const redirect = this.$route.query.redirect || "/";
24           this.$router.push(redirect);
25         }
26       });
27     }
28   }
29 };
30 </script>

```

插件

Nuxt.js会在运行应用之前执行插件函数，需要引入或设置Vue插件、自定义模块和第三方模块时特别有用

范例:添加请求拦截器附加token,创建plugins/interceptor.js

```
1 | export default function({$axios,store}){
2 |     $axios.onRequest(config=>{
3 |         if(store.state.user.token){
4 |             config.headers.Authorization='Bearer '+store.state.user.token;
5 |         }
6 |         return config;
7 |     })
8 | }
```

注册插件,nuxt.config.js

```
1 | plugins:["@/plugins/interceptor"]
```

nuxtServerInit

通过在store的根模块中定义 nuxtServerInit 方法，Nuxt.js 调用它的时候会将页面的上下文对象作为第2个参数传给它。当我们想将服务端的一些数据传到客户端时，这个方法非常好用。

安装依赖模块：npm i cookie-universal-nuxt -S

注册modules:['cookie-universal-nuxt'],

范例:登录状态初始化,store/index.js

```
1 | export const actions={
2 |     //该action只能出现在index
3 |     //且只能在服务端执行一次
4 |     //参数2是nuxt上下文
5 |     nuxtServerInit({commit},{app}){
6 |         const token=app.$cookies.get("token");
7 |         if(token){
8 |             console.log("nuxtServerInit:token:"+token);
9 |             commit('user/SET_TOKEN',token)
10 |         }
11 |     }
12 | }
```

服务端渲染应用部署

```
1 | npm run build
2 | npm start
```

生成内容在.nuxt/dist中

静态应用部署

Nuxt.js可依据路由配置将应用静态化，使得我们可以将应用部署至任何一个静态站点主机服务商

```
1 | npm run generate
```

注意渲染和接口服务器都需要处于启动状态

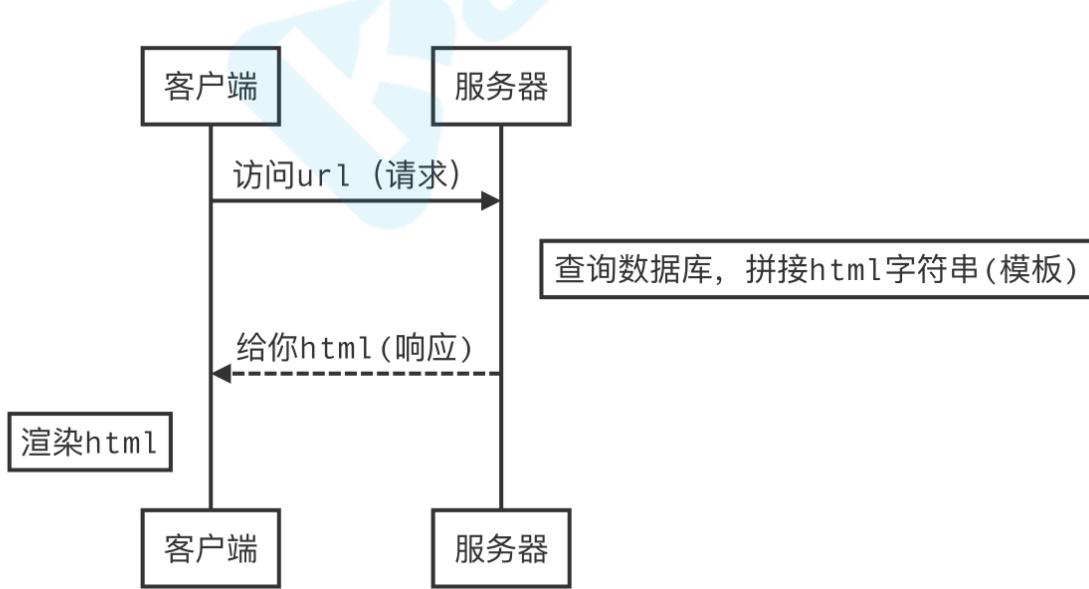
生成内容再dist中

Vue SSR实战

理解ssr

传统web开发

传统web开发，网页内容在服务端渲染完成，一次性传输到浏览器。

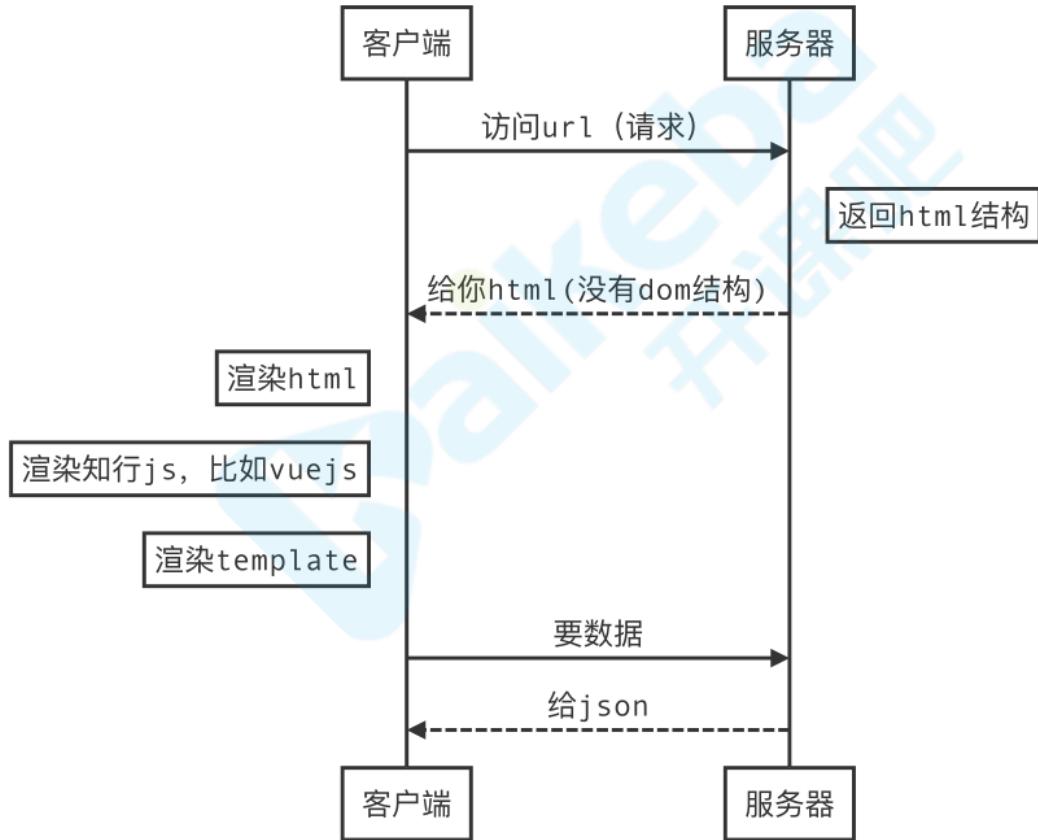


打开页面查看源码，浏览器拿到的是全部的dom结构

```
1 <div id="app">
2   <h1>开课吧</h1>
3   <p>开课吧真不错</p>
4 </div>
5
6
```

单页应用 Single Page App

单页应用优秀的用户体验，使其逐渐成为主流，页面内容由JS渲染出来，这种方式称为客户端渲染。



打开页面查看源码，浏览器拿到的仅有宿主元素#app，并没有内容。

```

1 <div id="app">
2   <h1>{{title}}</h1>
3   <p>{{content}}</p>
4 </div>
5 <script src="vue.js"></script>
6 <script>
7   new Vue({
8     el: '#app',
9     data: {
10       title: '开课吧',
11       content: '开课吧真不错'
12     }
13   })
14 </script>
15
16

```

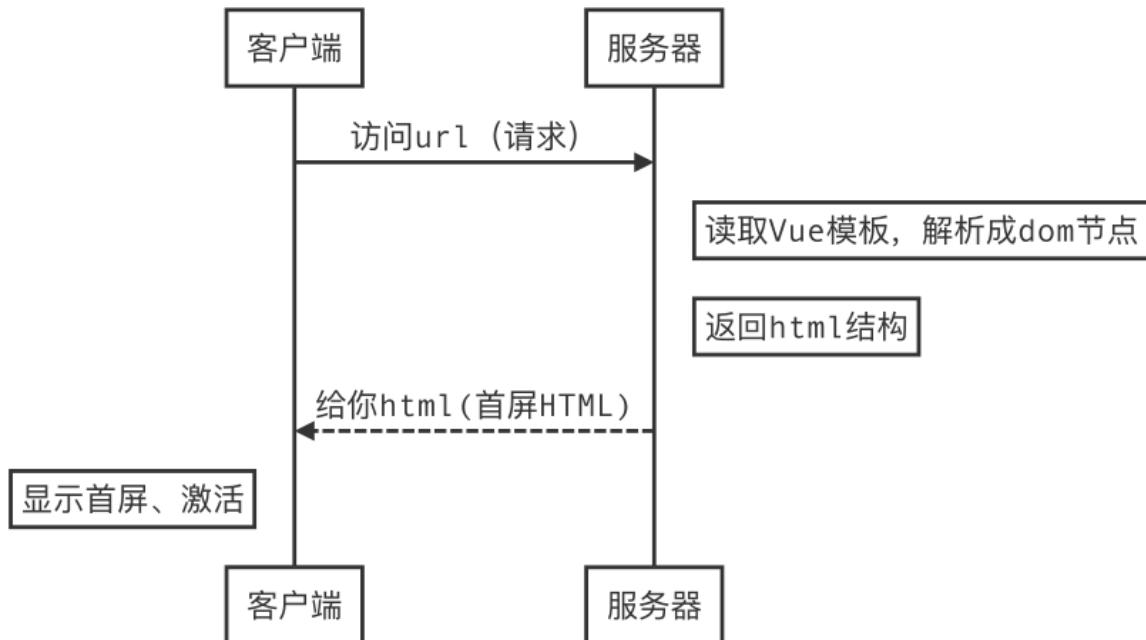
这里并没有实际内容

单页应用缺点：

1. seo
2. 首屏加载速度 优点：
3. 渲染计算放到客户端
4. 省流量

服务端渲染 Server Side Render

SSR解决方案，后端渲染出完整的首屏的dom结构返回，前端拿到的内容包括首屏及完整spa结构，应用激活后依然按照spa方式运行，这种页面渲染方式被称为服务端渲染 (server side render)



新建工程

vue-cli创建工程即可

```
1 | vue create ssr
```

演示项目使用vue-cli 4.x创建

安装依赖

```
1 | npm i vue-server-renderer express -S
```

要确保vue、vue-server-renderer版本一致

启动脚本

创建一个express服务器，将vue ssr集成进来

```
1 | const express=require('express');
2 | const app=express();
3 |
4 | const vue=require('vue');
5 | const {createBundleRenderer}=require('vue-server-renderer')
6 | const renderer =createBundleRenderer()
7 |
8 | app.get('/',async(req,res)=>{
9 |   const vm=new Vue({
10 |     data:{name:'村长真棒'},
11 |     template:'<div>{{name}}</div>'}
```

```

12  })
13 try {
14   const html=await renderer.renderToString(vm)
15   res.send(html)
16 } catch (error) {
17   res.status(500).send('服务器错误')
18 }
19 })
20
21 app.listen(3000)

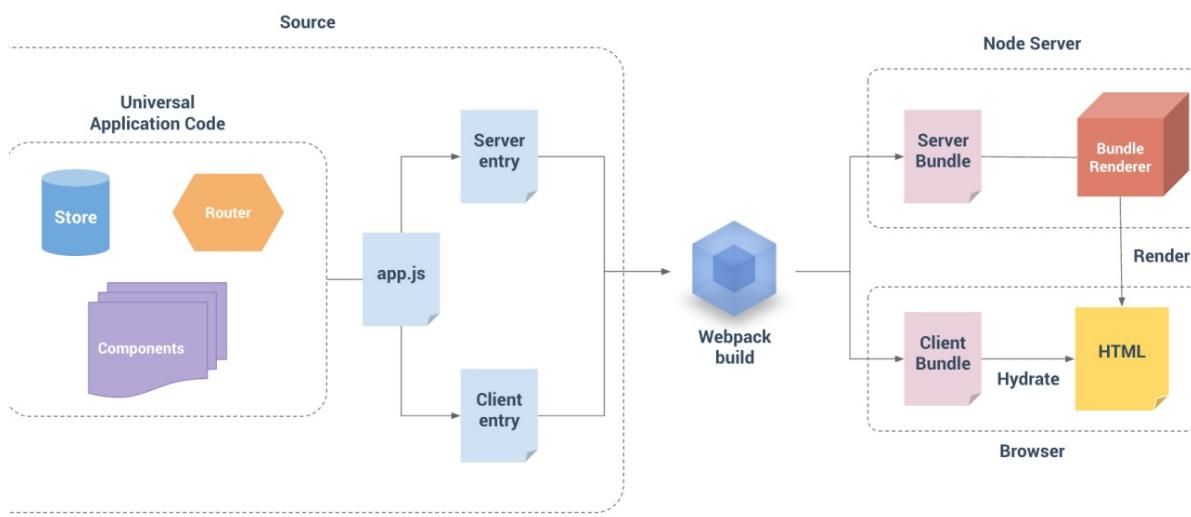
```

同构开发SSR应用

对于同构开发，我们依然使用webpack打包，我们要解决两个问题：服务端首屏渲染和客户端激活

构建流程

目标是生成一个「服务器 bundle」用于服务端首屏渲染，和一个「客户端bundle」用于客户端激活。



代码结构

除了两个不同入口之外，其他结构和之前vue应用完全相同

```

1 src
2   router
3     index.js # 路由声明
4   store
5     index.js # 全局状态
6   main.js # 用于创建vue实例
7   entry-client.js # 客户端入口，用于静态内容“激活”
8   entry-server.js # 服务端入口，用于首屏内容渲染

```

路由Vue-router

单页应用的页面路由，都是前端控制，后端值负责提供数据

一个简单的单页应用，使用vue-router,为了方便前后端公用路由数据，我们新建router.js对外暴露
createRouter

```
1 | npm i vue-router -S
```

```
1 //router.js
2 import Vue from "vue";
3 import Router from "vue-router";
4 import Index from "./components/index";
5 import Kkb from "./components/Kkb";
6
7 Vue.use(Router)
8
9 export function createRouter(){
10     return new Router({
11         routes:[
12             {path: '/', component:Index},
13             {path: '/kkb', component:Kkb}
14         ]
15     })
16 }
```

```
1 //src/components/index.vue
2 <template>
3     <div>
4         <h1>hi{{name}}</h1>
5     </div>
6 </template>
7
8 <script>
9 export default {
10     data() {
11         return {
12             name: '首页'
13         }
14     },
15 }
16 </script>
17
18 <style>
19
20 </style>
```

```
1 //src/components/Kkb.vue
2 <template>
3     <div>
4         <h1>hi{{name}}</h1>
5     </div>
6 </template>
7
8 <script>
```

```

9  export default {
10    data() {
11      return {
12        name: '开课吧'
13      }
14    },
15  }
16 </script>
17
18 <style>
19
20 </style>

```

```

1 //src/App.vue
2 <template>
3   <div id="app">
4     <div id="nav">
5       <router-link to="/">Home</router-link> |
6       <!-- <router-link to="/about">About</router-link> -->
7       <router-link to='/kkb'>开课吧</router-link>
8     </div>
9     <router-view/>
10    </div>
11  </template>
12
13 <style>
14 #app {
15   font-family: 'Avenir', Helvetica, Arial, sans-serif;
16   -webkit-font-smoothing: antialiased;
17   -moz-osx-font-smoothing: grayscale;
18   text-align: center;
19   color: #2c3e50;
20 }
21
22 #nav {
23   padding: 30px;
24 }
25
26 #nav a {
27   font-weight: bold;
28   color: #2c3e50;
29 }
30
31 #nav a.router-link-exact-active {
32   color: #42b983;
33 }
34 </style>
35

```

csr和ssr统一入口

跟之前不同，主文件是负责创建vue实例的工厂，每次请求均会有独立的vue实例创建。创建main.js：

```

1 //src/main.js
2 import Vue from "vue";
3 import App from "./App.vue";

```

```
4 import { createRouter } from "./router/router";
5
6 // 导出Vue实例工厂函数，为每次请求创建独立实例
7 // 上下文用于给Vue实例传递参数
8 export function createApp(context){
9     const router=createRouter();
10    const app=new Vue({
11        router,
12        context,
13        render:h=>h(App)
14    })
15    return {app,router}
16 }
```

服务端入口

上面的bundle就是webpack打包的服务端bundle，我们需要编写服务端入口文件src/entry-server.js。它的任务是创建Vue实例并根据传入url指定首屏

```
1 //src/entry-server.js
2 import { createApp } from "./createapp";
3 import { resolve } from "dns";
4 export default context=>{
5     //我们返回一个promise
6     //确保路由或组件准备就绪
7     return new Promise((resolve,reject)=>{
8         const {app,router}=createApp(context);
9         //跳转到首屏的地址
10        router.push(context.url);
11        router.onReady(()=>{
12            resolve(app)
13        },reject)
14    })
15 }
```

客户端入口

客户端入口只需创建vue实例并执行挂载，这一步称为激活。创建entry-client.js：

```
1 //客户端入口，用于客户端激活
2 //以下代码在浏览器端执行
3 import { createApp } from "./main";
4
5 const { app, router, store } = createApp();
6 router.onReady(() => {
7   app.$mount("#app");
8 });
9
```

后端加入webpack

配置

```
1 | npm install webpack-node-externals lodash.merge -D
```

具体配置

```
1 //vue.config.js
2 // 两个插件分别负责打包客户端和服务端
3 const VueSSRServerPlugin = require("vue-server-renderer/server-plugin");
4 const VueSSRClientPlugin = require("vue-server-renderer/client-plugin");
5 const nodeExternals = require("webpack-node-externals");
6 const merge = require("lodash.merge");
7 // 根据传入环境变量决定入口文件和相应配置项
8 const TARGET_NODE = process.env.WEBPACK_TARGET === "node";
9 const target = TARGET_NODE ? "server" : "client";
10 module.exports = {
11   css: {
12     extract: false
13   },
14   outputDir: "./dist/" + target,
15   configureWebpack: () => ({
16     // 将 entry 指向应用程序的 server / client 文件
17     entry: `./src/entry-${target}.js`,
18     // 对 bundle renderer 提供 source map 支持
19     devtool: "source-map",
20     // target设置为node使webpack以Node适用的方式处理动态导入,
21     // 并且还会在编译vue组件时告知`vue-loader`输出面向服务器代码。
22     target: TARGET_NODE ? "node" : "web",
23     // 是否模拟node全局变量
24     node: TARGET_NODE ? undefined : false,
25     output: {
26       // 此处使用Node风格导出模块
27       libraryTarget: TARGET_NODE ? "commonjs2" : undefined
28     },
29     // https://webpack.js.org/configuration/externals/#function
30     // https://github.com/liady/webpack-node-externals
31     // 外置化应用程序依赖模块。可以使服务器构建速度更快，并生成较小的打包文件。
32     externals: TARGET_NODE
33     ? nodeExternals({
34       // 不要外置化webpack需要处理的依赖模块。
35       // 可以在这里添加更多的文件类型。例如，未处理 *.vue 原始文件，
36       // 还应该将修改`global`（例如polyfill）的依赖模块列入白名单
37       allowlist: [/^.css$/]
38     })
39     : undefined,
40     optimization: {
41       splitChunks: undefined
42     },
43     // 这是将服务器的整个输出构建为单个 JSON 文件的插件。
44     // 服务端默认文件名为 `vue-ssr-server-bundle.json`
45     // 客户端默认文件名为 `vue-ssr-client-manifest.json`。
46     plugins: [TARGET_NODE ? new VueSSRServerPlugin() : new
VueSSRClientPlugin()]
47   },
48   chainWebpack: config => {
49     // cli4项目添加
50     if (TARGET_NODE) {
51       config.optimization.delete("splitChunks");
52     }
53     config.module
54       .rule("vue")
```

```
55     .use("vue-loader")
56     .tap(options => {
57       merge(options, {
58         optimizeSSR: false
59       });
60     });
61   }
62 };
63 
```

脚本配置

安装依赖

```
1 | npm i cross-env -D
```

定义创建脚本， package.json

```
1 | "scripts": {
2 |   "build:client": "vue-cli-service build",
3 |   "build:server": "cross-env WEBPACK_TARGET=node vue-cli-service build",
4 |   "build": "npm run build:server && npm run build:client"
5 | }
```

执行打包： npm run build

宿主文件

最后需要定义宿主文件，修改./public/index.html

```
1 | <!DOCTYPE html>
2 | <html lang="en">
3 |
4 | <head>
5 |   <meta charset="utf-8">
6 |   <meta http-equiv="X-UA-Compatible" content="IE=edge">
7 |   <meta name="viewport" content="width=device-width,initial-scale=1.0">
8 |   <title>Document</title>
9 | </head>
10 |
11 | <body>
12 |   <!--vue-ssr-outlet-->
13 | </body>
14 |
15 | </html>
```

服务器启动文件

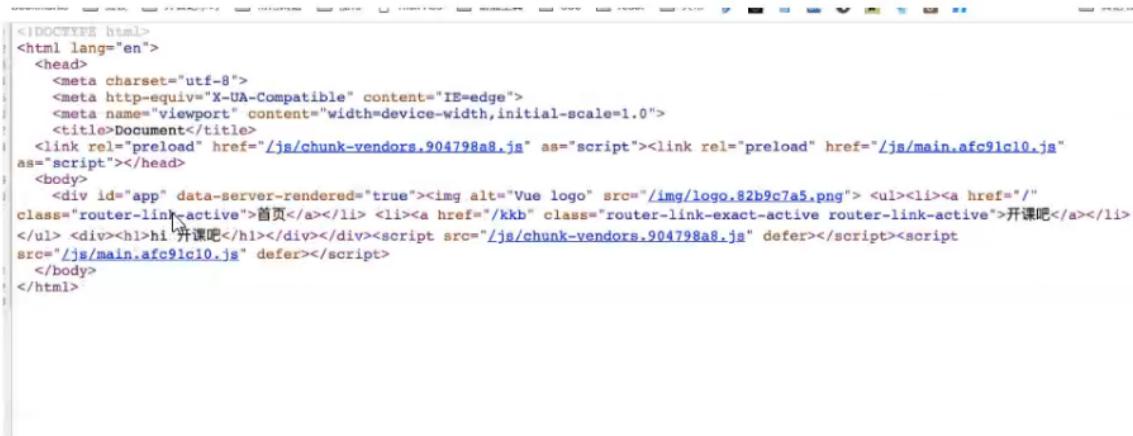
修改服务器启动文件，现在需要处理所有路由

```
1 | const express=require('express');
2 | const fs=require('fs');
3 | const app=express();
4 | const renderer = require('vue-server-renderer').createRenderer();
5 | 
```

```

6 //1.开放dist目录
7 app.use(express.static('./dist'))
8
9 //2.获得一个createBundleRenderer
10 const {createBundleRenderer}=require('vue-server-renderer');
11 const bundle=require('./dist/vue-server-bundle.json');
12 const clientManifest=require('./dist/vue-ssr-client-manifest.json');
13 // 获取文件路径
14 const resolve = dir => require('path').resolve(__dirname, dir)
15 // 第 1 步：开放dist/client目录，关闭默认下载index页的选项，不然到不了后面路由
16 app.use(express.static(resolve('../dist/client'), {index: false}))
17 // 第 2 步：获得一个createBundleRenderer
18 const { createBundleRenderer } = require("vue-server-renderer");
19 // 第 3 步：服务端打包文件地址
20 const bundle = resolve("../dist/server/vue-ssr-server-bundle.json");
21 // 第 4 步：创建渲染器
22 const renderer = createBundleRenderer(bundle, {
23   runInNewContext: false, // https://ssr.vuejs.org/zh/api/#runinnewcontext
24   template: require('fs').readFileSync(resolve("../public/index.html"), "utf-8"), // 宿主文件
25   clientManifest: require(resolve("../dist/client/vue-ssr-client-manifest.json")) // 客户端清单
26 });
27 app.get('*', async (req,res)=>{
28   // 设置url和title两个重要参数
29   const context = {
30     title:'ssr test',
31     url:req.url
32   }
33   const html = await renderer.renderToString(context);
34   res.send(html)
35 })
36
37 })

```



整合vuex

安装vuex

```
1 | npm i vuex -s
```

store/index.js

```
1 import vue from "vue";
```

```
2 import Vuex from "vuex";
3
4 Vue.use(Vuex);
5
6 export function createStore(){
7     return new Vuex.Store({
8         state: {
9             count:108
10        },
11        mutations: {
12            add(state){
13                state.count += 1;
14            }
15        },
16        actions: {
17        },
18        modules: {
19        }
20    })
21 }
```

挂载store, main.js

```
1 import Vue from "vue";
2 import App from "./App.vue";
3 import { createRouter } from "./router/router";
4 import { createStore } from "./store";
5
6 export function createApp(context){
7     const router=createRouter();
8     const store=createStore();
9     const app=new Vue({
10         router,
11         store,
12         context,
13         render:h=>h(App)
14     })
15     return {app,router,store}
16 }
```

使用

```
1 //src/components/App.vue
2 <h2 @click="$store.commit('add')">&nbsp;{{ $store.state.count }}</h2>
```

注意事项：注意打包和重启服务

数据预取

服务器端渲染的是应用程序的“快照”，如果应用依赖于一些异步数据，那么在开始渲染之前，需要先预取和解析好这些数据。

异步数据获取, store/index.js

```
1 import Vue from "vue";
2 import Vuex from "vuex";
```

```

3  Vue.use(vuex);
4  export function createStore() {
5    return new Vuex.Store({
6      state: {
7        count: 108
8      },
9      mutations: {
10        add(state) {
11          state.count += 1;
12        },
13        init(state, count) {
14          state.count = count;
15        }
16      },
17      actions: {
18        // 加一个异步请求count的action
19        getCount({ commit }) {
20          return new Promise(resolve => {
21            setTimeout(() => {
22              commit("init", Math.random() * 100);
23              resolve();
24            }, 1000);
25          });
26        }
27      }
28    );
29  }
30

```

组件中的数据预取逻辑，Index.vue

```

1  export default {
2    asyncData({ store, route }) {
3      // 约定预取逻辑编写在预取钩子asyncData中
4      // 触发 action 后，返回 Promise 以便确定请求结果
5      console.log(route)
6      return store.dispatch("getCount");
7    }
8  }

```

服务端数据预取，entry-server.js

```

1 //主要用于首屏渲染
2 import { createApp } from "./main";
3
4 //renderer传入一个url地址，即首屏地址
5 export default context => {
6   //返回一个Promise，确保路由中可能有异步操作
7   return new Promise((resolve, reject) => {
8     const { app, router, store } = createApp();
9     //获取首屏地址，并且跳过去
10    router.push(context.url);
11    router.onReady(() => {
12      //检查一下当前匹配的组件是否需要请求异步数据
13      // 获取匹配的路由组件数组
14      const matchedComponents = router.getMatchedComponents();

```

```

15     // 若无匹配则抛出异常
16     if (!matchedComponents.length) {
17         return reject({ code: 404 });
18     }
19
20     // 对所有匹配的路由组件调用可能存在的`asyncData()`
21     Promise.all(
22         matchedComponents.map(Component => {
23             if (Component.asyncData) {
24                 return Component.asyncData({
25                     store,
26                     route: router.currentRoute
27                 });
28             }
29         })
30     )
31     .then(() => {
32         // 所有预取钩子 resolve 后,
33         // store 已经填充入渲染应用所需状态
34         // 将状态附加到上下文, 且 `template` 选项用于 renderer 时,
35         // 状态将自动序列化为 `window.__INITIAL_STATE__`, 并注入 HTML。
36         context.state = store.state;
37         resolve(app);
38     })
39     .catch(reject);
40 }, reject);
41 );
42 };
43

```

客户端在挂载到应用程序之前, store 就应该获取到状态, entry-client.js

```

1 // 导出store
2 const { app, router, store } = createApp();
3 // 当使用 template 时, context.state 将作为 window.__INITIAL_STATE__ 状态自动嵌入
4 到
5 最终的 HTML // 在客户端挂载到应用程序之前, store 就应该获取到状态:
6 if (window.__INITIAL_STATE__) {
7     store.replaceState(window.__INITIAL_STATE__);
8 }

```

客户端数据预取处理, main.js

```

1 Vue.mixin({
2     beforeMount() {
3         const { asyncData } = this.$options;
4         if (asyncData) {
5             // 将获取数据操作分配给 promise
6             // 以便在组件中, 我们可以在数据准备就绪后
7             // 通过运行 `this.dataPromise.then(...)` 来执行其他任务
8             this.dataPromise = asyncData({
9                 store: this.$store,
10                route: this.$route
11            });
12        }
13    }
}

```

```
14 |});
```

TypeScript

准备工作

新建一个基于ts的vue项目

```
1 | vue create vue-ts
```

选项选择:

- 自定义选项:Manually select features
- 添加ts支持-TypeScript
- 基于类的组件-y
- tslint

已存在项目

```
vue add @vue/typescript
```

类型注解和类型检查

```
1 //类型注解
2 let foo='xxx'//类型推论
3 let bar:string;//类型注解
4
5 let abc:{foo:string,bar:number}
6 abc={foo:'foo',bar:1}
7
8 let efg:string|number;
9 efg='rfg';
10 efg=1;
11
12 //bar=1//wrong
13 bar='bar'
14
15 //数组类型约束
16 let names:string[];
17 names=['zyl','lily']
18
19 //任意类型
20 let baz:any;
21 baz=1;
22 baz='abc';
23
24 //any应用于数组
25 let list:any[]={1,true,'free'};
26 list[0]='abc'
27
28 //函数中使用类型
29 function greeting(person:string):string {
30     return 'hello'+person;
31 }
32 greeting('zyl')
33
```

```
34 function warnUser():void {
35     alert('lala')
36 }
37
38 //内置常见类型:string,number,boolean,void,any
39
```

vue中的应用，Hello.vue

```
1 <template>
2   <div>
3     <div v-for=" feature in features" :key="feature">{{feature}}</div>
4   </div>
5 </template>
6
7 <script lang='ts'>
8 import { Component, Prop, Vue, Emit, Watch } from "vue-property-decorator";
9 @Component
10 export default class Hello extends Vue {
11   features:Feature[];
12   constructor() {
13     super();
14     this.features = ["类型注解","编译型语言"];
15   }
16 }
17 </script>
18
19 <style>
20 </style>
```

类型别名

使用类型别名自定义类型

```
1 // 可以用下面这样方式定义对象类型
2 const objType: { foo: string, bar: string }
3 // 使用type定义类型别名，使用更便捷，还能复用
4 type Foobar = { foo: string, bar: string }
5 const aliasType: Foobar
```

范例：使用类型别名定义Feature，types/index.ts

```
1 export type Feature = {
2   id: number,
3   name: string
4 }
```

使用自定义类型，Hello.vue

```
1 <template>
2   <div>
3     <!--修改模板-->
4     <li v-for="feature in features" :key="feature.id">{{feature.name}}</li>
5   </div>
6 </template>
```

```
7 <script lang='ts'>
8 // 导入接口
9 import { Feature } from "@/types";
10
11 @Component
12 export default class Hello extends Vue {
13     // 修改数据结构
14     features: Feature[] = [{ id: 1, name: "类型注解" }];
15 }
16 </script>
```

联合类型

希望某个变量或参数的类型是多种类型其中之一

```
1 let union: string | number;
2 union = '1'; // ok
3 union = 1; // ok
```

交叉类型

想要定义某种由多种类型合并而成的类型使用交叉类型

```
1 type First = {first: number};
2 type Second = {second: number};
3 // FirstAndSecond将同时拥有属性first和second
4 type FirstAndSecond = First & Second;
```

范例：利用交叉类型给Feature添加一个selected属性

```
1 / types/index.ts
2 type Select = {
3     selected: boolean
4 }
5 export type FeatureSelect = Feature & Select
```

使用这个FeatureSelect, Hello.vue

```
1 features: FeatureSelect[] = [
2     { id: 1, name: "类型注解", selected: false },
3     { id: 2, name: "编译型语言", selected: true }
4 ];
```

```
1 <li :class="{selected: feature.selected}">{{feature.name}}</li>
```

```
1 .selected {
2     background-color: rgb(168, 212, 247);
3 }
```

函数

必填参：参数一旦声明，就要求传递，且类型需符合

```
1 //函数
2 //必填参数
3 //加上?表示可选，可选参数在必填参数后面
4 function sayHello(name:string,age?:number){
5     console.log(name,age)
6 }
7 function sayHello2(name:string,age:number=18){
8     console.log(name,age)
9 }
10
11 sayHello('tom',18)
12 sayHello2('tom')
13
14
15 /*函数重载：以参数的数量或类型区分多个同名函数
16 //先声明，再实现
17 function info(a:object):string;
18 function info(a:string):object;
19 //实现
20 function info(a:any):any{
21     if(typeof a==='object'){
22         return a.name;
23     }else{
24         return {name:a}
25     }
26 }
27
28 console.log(info({name:'tom'}))
29 console.log(info('tom'))
30
31 // 重载1
32 function watch(cb1: () => void): void;
33 // 重载2
34 function watch(cb1: () => void, cb2: (v1: any, v2: any) => void): void;
35 // 实现
36 function watch(cb1: () => void, cb2?: (v1: any, v2: any) => void) {
37     if (cb1 && cb2) {
38         console.log('执行watch重载2');
39     } else {
40         console.log('执行watch重载1');
41     }
42 }
```

vue应用，Hello.vue

```
1 <div>
2     <input type="text" @keydown.enter="addFeatures" />
3 </div>
```

```
1 //声明周期
2 created() {
3     console.log("created");
```

```

4     this.features = [{ id: 1, name: "类型注解" }];
5 }
6 addFeature(e: KeyboardEvent) {
7     // e.target是EventTarget类型, 需要断言为HTMLInputElement
8     const inp = e.target as HTMLInputElement;
9     const feature: FeatureSelect = {
10         id: this.features.length + 1,
11         name: inp.value,
12         selected: false
13     }
14     this.features.push(feature);
15     inp.value = "";
16 }

```

类

class的特性

ts中的类和es6中大体相同, 这里重点关注ts带来的访问控制等特性

```

1 //class
2 class MyComp{
3     private _foo:string;//私有属性, 不能在类的外部访问
4     protected bar:string;//报复属性, 还可以在派生类中访问
5     readonly mua='mua';//只读属性必须在声明时或构造函数里初始化
6     static dong='dong';//MyComp.dong访问
7
8     //构造函数, 初始化成员变量
9     //参数加上修饰符, 能够定义并初始化一个成员属性
10    constructor(private tua='tua'){
11        this._foo='foo';
12        this.bar='bar';
13    }
14
15    //方法也有修饰符
16    private someMethod(){}
17
18    //存取器, 存取数据时刻添加额外逻辑, 在vue里面可以用作计算属性
19    get foo(){
20        return this._foo;
21    }
22
23    set foo(val){
24        this._foo=val;
25    }
26 }

```

vue应用: 声明自定义类型约束数据结构,Hello.vue

```

1 class Feature{
2     constructor(public id:number,public name:string){
3
4     }
5
6
7     @Component

```

```

8  export default class Hello extends Vue {
9    features:Feature[];
10   constructor() {
11     super();
12     this.features = [{id:1,name:"类型注解"},{id:2,name:"编译型语言"}];
13   }
14 }
```

vue应用：利用getter设置计算属性

```

1 <template>
2 <li>特性数量: {{count}}</li>
3 </template>
4 <script lang="ts">
5   export default class HelloWorld extends Vue {
6     // 定义getter作为计算属性
7     get count() {
8       return this.features.length;
9     }
10   }
11 </script>
```

class是语法糖，它指向的就是构造函数

```

1 class Person{//类指向构造函数
2   constructor(name,age){//constructor是默认方法, new实例时自动调用
3     this.name=name;//属性声明在实例上, 因为this指向实例
4     this.age=age;
5   }
6   say(){//方法会声明在原型上
7     return '我的名字叫'+this.name+'今年'+this.age+'岁了';
8   }
9 }
10
11 console.log(typeof Person);//function
12 console.log(Person==Person.prototype.constructor);//true
13
14 //等效于
15 function Person(name,age){
16   this.name=name;
17   this.age=age;
18 }
19 Person.prototype.say=function(){//方法会声明在原型上
20   return '我的名字叫'+this.name+'今年'+this.age+'岁了';
21 }
```

接口

interface，仅定义结构，不需要实现

```

1 //接口只声明结构不需要实现
2 interface Person{
3   firstName:string;
4   lastName:string;
5   sayHello():string;//要求实现方法
```

```

6 }
7
8 //实现接口
9 class Greeter implements Person{
10     constructor(public firstName='',public lastName=''){
11         sayHello(){
12             return 'Hello,'+this.firstName+' '+this.lastName
13         }
14     }
15 //面向接口编程
16 function greeting2(person:Person){
17     return person.sayHello();
18 }
19
20 // const user={firstName:'zheng',lastName:'yali'};
21 const user=new Greeter('Jane','User');//创建对象实例
22 console.log(user)
23 console.log(greeting2(user))

```

范例：修改Feature为接口形式

```

1 <script lang='ts'>
2     export interface Feature{
3         id:number,
4         name:string
5     }
6 </script>

```

泛型Generics

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

```

1 interface Result<T>{
2     ok:0|1;
3     data:T[];
4 }
5
6 function getData<T>():Result<T>{
7     const data:any[]=[
8         {id:1,name:"类型注解"},{id:2,name:"ts类型注解"}
9     ]
10    return {ok:1,data}
11 }
12
13 //使用泛型
14 this.features= getData<Feature>().data;
15
16 // 用尖括号方式指定T为string
17 getResult<string>('hello')
18 // 用类型推断指定T为number
19 getResult(1)

```

返回Promise

```
1 function getData<T>():Promise<Result<T>>{
2     const data :any= [
3         { id: 1, name: "类型注解", selected: true },
4         { id: 2, name: "nihao", selected: false }
5     ];
6     return new Promise((reslove)=>{
7         reslove({
8             ok: 1,
9             data
10        })
11    })
12 }
```

使用

```
1 //声明周期
2 private async created() {
3     console.log("created");
4     this.features=(await getData<Feature>()).data;
5 }
```

泛型优点:

- 函数和类可以支持多种类型，更加通用
- 不必编写多条重载，冗长联合类型，可读性好
- 灵活控制类型约束

范例：用axios获取数据 安装axios： npm i axios -S 配置一个模拟接口， vue.config.js

```
1 module.exports = {
2     devServer: {
3         before(app) {
4             app.get('/api/list', (req, res) => {
5                 res.json([
6                     { id: 1, name: "类型注解", selected: false },
7                     { id: 2, name: "编译型语言", selected: true }
8                 ])
9             })
10        }
11    }
12 }
```

使用接口， HelloWorld.vue

```
1 async mounted() {
2     console.log("HelloWorld");
3     const resp = await axios.get<FeatureSelect[]>('/api/list')
4     this.features = resp.data
5 }
```

声明文件

使用ts开发时如果要使用第三方js库的同时还想利用ts诸如类型检查等特性就需要声明文件，类似xx.d.ts
同时，vue项目中还可以在shims-vue.d.ts中对已存在模块进行补充
npm i @types/xxx 范例：利用模块补充\$axios属性到Vue实例，从而在组件里面直接用

```
1 // main.ts
2 import axios from 'axios'
3 Vue.prototype.$axios = axios;
4 // shims-vue.d.ts
5 import Vue from "vue";
6 import { AxiosInstance } from "axios";
7 declare module "vue/types/vue" {
8   interface Vue {
9     $axios: AxiosInstance;
10 }
11 }
```

范例：给krouter/index.js编写声明文件，index.d.ts

```
1 import VueRouter from "vue-router";
2 declare const router: VueRouter
3 export default router
```

装饰器

装饰器用于扩展类或者它的属性和方法。@xxx就是装饰器的写法

属性声明：@Prop 除了在@Component中声明，还可以采用@Prop的方式声明组件属性

```
1 export default class HelloWorld extends Vue {
2   // Props()参数是为vue提供属性选项
3   // !称为明确赋值断言，它是提供给ts的
4   @Prop({type: String, required: true})
5   private msg!: string;
6 }
```

事件处理：@Emit 新增特性时派发事件通知，Hello.vue

```
1 // 通知父类新增事件，若未指定事件名则函数名作为事件名（羊肉串形式）
2 @Emit()
3 private addFeature(event: any) { // 若没有返回值形参将作为事件参数
4   const feature = { name: event.target.value, id: this.features.length + 1 };
5   this.features.push(feature);
6   event.target.value = "";
7   return feature; // 若有返回值则返回值作为事件参数
8 }
```

变更监测：@Watch

```
1 | @watch('msg')
2 | onMsgChange(val:string, oldVal:any){
3 |   console.log(val, oldVal);
4 | }
```

状态管理推荐使用：vuex-module-decorators (<https://github.com/championswimmer/vuex-module-decorators>)

vuex-module-decorators 通过装饰器提供模块化声明vuex模块的方法，可以有效利用ts的类型系统。

安装

```
1 | npm i vuex-module-decorators -D
```

根模块清空，修改store/index.ts

```
1 | export default new Vuex.Store({})
```

定义counter模块，创建store/counter.ts

```
1 | import { Module, VuexModule, Mutation, Action, getModule } from 'vuex-module-decorators'
2 | import store from './index'
3 | // 动态注册模块
4 | @Module({ dynamic: true, store: store, name: 'counter', namespaced: true })
5 | class CounterModule extends VuexModule {
6 |   count = 1
7 |   @Mutation
8 |   add() {
9 |     // 通过this直接访问count
10 |     this.count++
11 |   }
12 |   // 定义getters
13 |   get doubleCount() {
14 |     return this.count * 2;
15 |   }
16 |   @Action
17 |   asyncAdd() {
18 |     setTimeout(() => {
19 |       // 通过this直接访问add
20 |       this.add()
21 |     }, 1000);
22 |   }
23 | }
24 |
25 | // 导出模块应该是getModule的结果
26 | export default getModule(CounterModule)
```

使用，App.vue

```
1 | <p @click="add">{{$store.state.counter.count}}</p>
2 | <p @click="asyncAdd">{{count}}</p>
```

```

1 import CounterModule from '@/store/counter'
2 @Component
3 export default class App extends Vue {
4     get count() {
5         return CounterModule.count
6     }
7     add() {
8         CounterModule.add()
9     }
10    asyncAdd() {
11        CounterModule.asyncAdd()
12    }
13}

```

装饰器实际上是工厂函数，传入一个对象，输出处理后的新对象

```

1 @Component
2 export default class Hello extends Vue {
3     @Prop({
4         type:String,
5         default:''
6     })
7     private msg!:string;//写给ts
8     @Watch('features',{deep:true})
9     private msgChange newVal:any,oldVal:any){
10         console.log(newVal,oldVal)
11     }
12
13 //不给emit传参，表示时间名称是方法名
14     @Emit()
15     private addFeatures(event: KeyboardEvent) {
16         //as: 类型断言，使类型更加具体，不是类型转换
17         const inp = event.target as HTMLInputElement;
18         const feature={id:this.features.length+1,name:inp.value}
19         this.features.push(feature);
20         inp.value = "";
21         //如果没有返回值，形参就是事件参数，否则返回值是
22         return feature;
23     }
24 }

```

vuex使用:vuex-class

安装

```
npm i vuex-class -S
```

定义状态，store.js

```

1 import Vue from 'vue'
2 import Vuex from 'vuex'
3
4 Vue.use(Vuex)
5
6 export default new Vuex.Store({
7     state: {

```

```

8   features:['类型检测','预编译']
9 },
10 mutations: {
11   addFeatureMutation(state:any,featureName:string){
12     state.features.push({id:state.features.length+1,name:featureName})
13   }
14 },
15 actions: {
16   addFeatureAction({commit},featureName:string){
17     commit('addFeatureMutation',featureName)
18   }
19 },
20 modules: {
21 }
22 })

```

使用, Hello.vue

```

1 import { State,Mutation,Action } from "vuex-class";
2 @Component
3 export default class Hello extends Vue {
4   //状态、动作、变更映射
5   @State features!:string[];
6   @Action addFeatureAction;
7   @Mutation addFeatureMutation;
8
9   private addFeatures(event: any) {
10     this.addFeatureAction(event.target.value);
11     event.target.value = "";
12     //如果没有返回值, 形参就是事件参数, 否则返回值是
13     return feature;
14   }
15 }

```

装饰器原理

装饰器实际上是一个函数，通过定义劫持，能够对类及其方法、属性提供额外的扩展功能

```

1 //自定义装饰器
2 function log(target:Function){
3   //target是构造函数
4   console.log(target==Foo); //true
5   target.prototype.log=function(){
6     console.log(this.bar);
7   }
8   //如果类装饰器返回一个值，它会使用提供的构造函数来替换类的声明
9 }
10 //方法装饰器
11 function dong(target:any,name:string,descriptor:any){
12   //target是原型或构造函数, name是方法名, descriptor是属性描述符
13   //方法的定义方式:Object.defineProperty(target,name,descriptor)
14   console.log(target[name]==descriptor.value);
15   //这里通过修改descriptor.value扩展了bar方法
16   const baz=descriptor.value;//之前的方法
17   descriptor.value=function(val:string){
18     console.log('dong');

```

```

19         baz.call(this, val);
20     }
21     return descriptor;
22 }
23
24 //属性装饰器
25 function mua(target, name){
26     //target是原型或构造函数, name是属性名
27     console.log(target === Foo.prototype);
28     target[name] = 'mua~~~'
29 }
30
31 @log
32 class Foo{
33     bar='bar';
34     @mua ns!:string;
35     @dong
36     baz(val:string){
37         this.bar=val;
38     }
39 }
40
41 const foo2=new Foo();
42 foo2.log();
43 console.log(foo2.ns);
44 foo2.baz('lala')

```

实战一下component,新建Decor.vue

```

1 <template>
2   <div>
3     {{msg}}
4   </div>
5 </template>
6
7 <script lang='ts'>
8 import {Prop, Vue} from "vue-property-decorator";
9 function Component(options:any){
10   return function(target:Function){
11     options.data=function(){
12       return {
13         features:[]
14       }
15     }
16   }
17   return Vue.extend(options)
18 }
19
20 @Component({
21   props:{
22     msg:{
23       type:String,
24       default:''
25     }
26   }
27 })
28 export default class Deco extends Vue{

```

```
29     features:string[]=[];
30 }
31
32 </script>
33
34 <style>
35
36 </style>
```

显然options中的选项都可以从Decor定义中找到

vue3初探 + 响应式原理

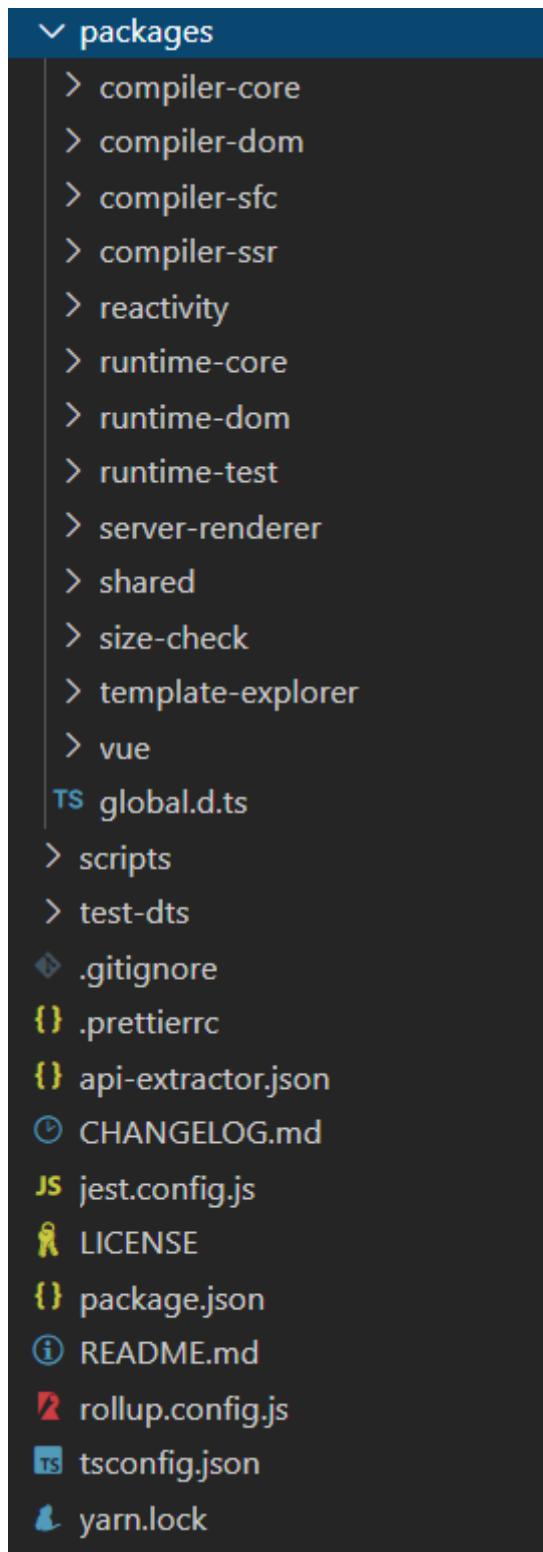
调试环境搭建

- 迁出Vue3源码： git clone <https://github.com/vuejs/vue-next.git>
- 安装依赖： yarn --ignore-scripts
- 生成sourcemap文件， package.json

```
1 | "dev": "node scripts/dev.js --sourcemap"
```

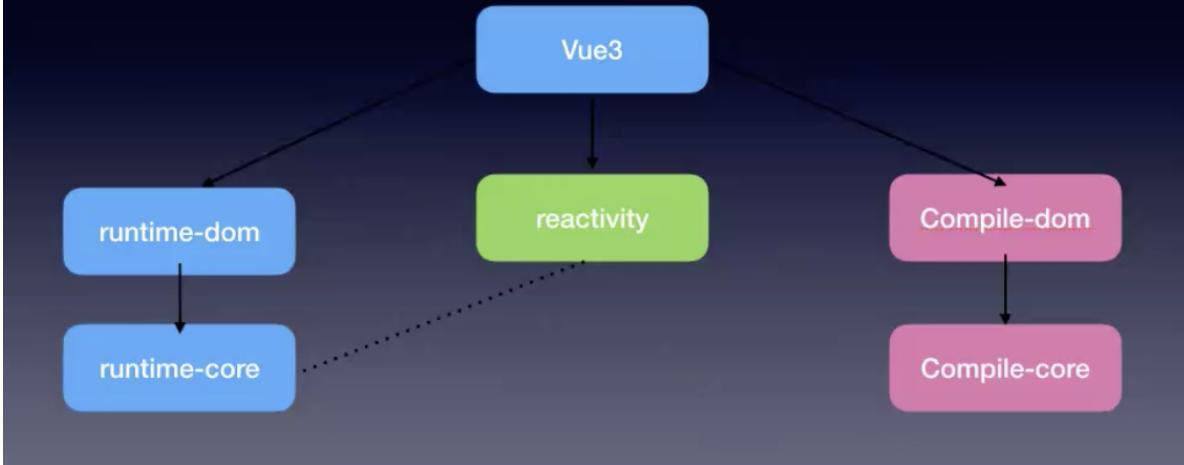
- 编译： yarn dev
 - 生成结果：
 - packages\vue\dist\vue.global.js
 - packages\vue\dist\vue.global.js.map
- 调试范例代码： yarn serve

源码结构



源码位置是在package文件件内，实际上源码主要分为两部分，编译器和运行时环境。

Vue3模块



- 编译器
 - compiler-core 核心编译逻辑
 - compiler-dom 针对浏览器平台编译逻辑
 - compiler-sfc 针对单文件组件编译逻辑
 - compiler-ssr 针对服务端渲染编译逻辑
- 运行时环境
 - runtime-core 运行时核心
 - runtime-dom 运行时针对浏览器的逻辑
 - runtime-test 浏览器外完成测试环境仿真
- reactivity 响应式逻辑
- template-explorer 模板浏览器
- vue 代码入口，整合编译器和运行时
- server-renderer 服务器端渲染
- share 公用方法

Vue 3初探

测试代码, ~/packages/examples/01-hello-vue3.html

```
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4  <meta charset="UTF-8">
5  <meta name="viewport" content="width=device-width, initial-scale=1.0">
6  <meta http-equiv="X-UA-Compatible" content="ie=edge">
7  <title>hello vue3</title>
8  <script src="../dist/vue.global.js"></script>
9  </head>
10 <body>
11 <div id="app">
12 <h1 @click="onclick">{{message}}</h1>
13 <comp></comp>
14 </div>
15 <script>
```

```

16 const { createApp } = vue
17 const app = createApp({
18   components: {
19     comp: {
20       template: '<div>this is a component</div>'
21     }
22   },
23   data: { message: 'Hello Vue3!' },
24   methods: {
25     onclick() {
26       console.log('click me');
27     }
28   },
29 }) .mount('#app')
30 </script>
31 </body>
32 </html>

```

Composition API

Composition API (<https://vue-composition-api-rfc.netlify.com/api.html%23setup>) 字面意思是组合API，它是为了实现基于函数的逻辑复用机制而产生的

基本使用

数据响应式，创建02-composition-api.html

```

1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4    <meta charset="UTF-8">
5    <meta name="viewport" content="width=device-width, initial-scale=1.0">
6    <meta http-equiv="X-UA-Compatible" content="ie=edge">
7    <title>Document</title>
8    <script src="../dist/vue.global.js"></script>
9  </head>
10 <body>
11 <div id="app">
12   <h1>Composition API</h1>
13   <div>count: {{ state.count }}</div>
14 </div>
15 <script>
16   const {
17     createApp,
18     reactive
19   } = vue;
20   // 声明组件
21   const App = {
22     // setup是一个新的组件选项，它是组件内使用Composition API的入口
23     // 调用时刻是初始化属性确定后，beforeCreate之前
24     setup() {
25       // 响应化：接收一个对象，返回一个响应式的代理对象
26       const state = reactive({ count: 0 })
27       // 返回对象将和渲染函数上下文合并
28       return { state }
29     }
30   }

```

```
31 | createApp(App).mount('#app')
32 | </script>
33 | </body>
34 | </html>
```

计算属性

```
1 | <div>doubleCount: {{doubleCount}}</div>
```

```
1 | const { computed } = Vue;
2 | const App = {
3 |   setup() {
4 |     const state = reactive({
5 |       count: 0,
6 |       // computed()返回一个不可变的响应式引用对象
7 |       // 它封装了getter的返回值
8 |       doubleCount: computed(() => state.count * 2)
9 |     })
10 |
11 | }
```

事件处理

```
1 | <div @click="add">count: {{ state.count }}</div>
```

```
1 | const App = {
2 |   setup() {
3 |     // setup中声明一个add函数
4 |     function add() {
5 |       state.count++
6 |     }
7 |     // 传入渲染函数上下文
8 |     return { state, add }
9 |   }
10 | }
```

侦听器: watch()

```
1 | const { watch } = Vue;
2 | const App = {
3 |   setup() {
4 |     // state.count变化cb会执行
5 |     watch(() => state.count, (val, oldval) => {
6 |       console.log('count变了:' + val);
7 |     })
8 |   }
9 | }
```

引用对象: 单值响应化

```
1 | <div>counter: {{ counter }}</div>
```

```
1 const { ref } = vue;
2 const App = {
3   setup() {
4     // 返回响应式的Ref对象
5     const counter = ref(1)
6     setTimeout(() => {
7       // 要修改对象的value
8       counter.value++
9     }, 1000);
10    // 添加counter
11    return { state, add, counter }
12  }
13}
```

体验逻辑组合

03-logic-composition.html

```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4 <meta charset="UTF-8">
5 <meta name="viewport" content="width=device-width, initial-scale=1.0">
6 <title>composition api</title>
7 <script src="../dist/vue.global.js"></script>
8 </head>
9 <body>
10 <div>
11 <h1>逻辑组合</h1>
12 <div id="app"></div>
13 </div>
14 <script>
15 const { createApp, reactive, onMounted, onUnmounted, toRefs } = Vue;
16 // 鼠标位置侦听
17 function useMouse() {
18   // 数据响应化
19   const state = reactive({ x: 0, y: 0 })
20   const update = e => {
21     state.x = e.pageX
22     state.y = e.pageY
23   }
24   onMounted(() => {
25     window.addEventListener('mousemove', update)
26   })
27   onUnmounted(() => {
28     window.removeEventListener('mousemove', update)
29   })
30   // 转换所有key为响应式数据
31   return toRefs(state)
32 }
33 // 事件监测
34 function useTime() {
35   const state = reactive({ time: new Date() })
36   onMounted(() => {
37     setInterval(() => {
38       state.time = new Date()
39     }, 1000)
```

```

40  })
41  return toRefs(state)
42 }
43 // 逻辑组合
44 const MyComp = {
45   template: `
46   <div>x: {{ x }} y: {{ y }}</div>
47   <p>time: {{time}}</p>
48 `,
49   setup() {
50     // 使用鼠标逻辑
51     const { x, y } = useMouse()
52     // 使用时间逻辑
53     const { time } = useTime()
54     // 返回使用
55     return { x, y, time }
56   }
57 }
58 createApp(MyComp).mount('#app')
59 </script>
60 </body>
61 </html>

```

对比mixins，好处显而易见：

- x,y,time来源清晰
- 不会与data、props等命名冲突

可维护性提高了

Vue3响应式原理

Vue2响应式原理回顾

```

1  // 1. 对象响应化：遍历每个key，定义getter、setter
2  // 2. 数组响应化：覆盖数组原型方法，额外增加通知逻辑
3  const originalProto = Array.prototype
4  const arrayProto = Object.create(originalProto)
5  ;['push', 'pop', 'shift', 'unshift', 'splice', 'reverse', 'sort'].forEach(
6    method => {
7      arrayProto[method] = function() {
8        originalProto[method].apply(this, arguments)
9        notifyUpdate()
10     }
11   }
12 )
13 function observe(obj) {
14   if (typeof obj !== 'object' || obj == null) {
15     return
16   }
17   // 增加数组类型判断，若是数组则覆盖其原型
18   if (Array.isArray(obj)) {
19     Object.setPrototypeOf(obj, arrayProto)
20   } else {
21     const keys = Object.keys(obj)
22     for (let i = 0; i < keys.length; i++) {
23       const key = keys[i]

```

```

24 defineReactive(obj, key, obj[key])
25 }
26 }
27 }
28 function defineReactive(obj, key, val) {
29 observe(val) // 解决嵌套对象问题
30 Object.defineProperty(obj, key, {
31 get() {
32 return val
33 },
34 set(newVal) {
35 if (newVal !== val) {
36 observe(newVal) // 新值是对象的情况
37 val = newVal
38 notifyUpdate()
39 }
40 }
41 })
42 }
43 function notifyUpdate() {
44 console.log('页面更新!')
45 }

```

vue2响应式弊端：

- 响应化过程需要递归遍历，消耗较大
- 新加或删除属性无法监听
- 数组响应化需要额外实现
- Map、Set、Class等无法响应式
- 修改语法有限制

Vue3响应式原理剖析

vue3使用ES6的Proxy特性来解决这些问题。创建04-reactivity.js

```

1 function reactive(obj) {
2 if (typeof obj !== 'object' && obj != null) {
3 return obj
4 }
5 // Proxy相当于在对象外层加拦截
6 // http://es6.ruanyifeng.com/#docs/proxy
7 const observed = new Proxy(obj, {
8 get(target, key, receiver) {
9 // Reflect用于执行对象默认操作，更规范、更友好
10 // Proxy和Object的方法Reflect都有对应
11 // http://es6.ruanyifeng.com/#docs/reflect
12 const res = Reflect.get(target, key, receiver)
13 console.log(`获取${key}:${res}`)
14 return res
15 },
16 set(target, key, value, receiver) {
17 const res = Reflect.set(target, key, value, receiver)
18 console.log(`设置${key}:${value}`)
19 return res
20 },
21 deleteProperty(target, key) {
22 const res = Reflect.deleteProperty(target, key)

```

```
23 console.log(`删除${key}:${res}`)
24 return res
25 }
26 })
27 return observed
28 }
```

测试代码

```
1 const state = reactive({
2   foo: 'foo'
3 })
4 // 1. 获取
5 state.foo // ok
6 // 2. 设置已存在属性
7 state.foo = 'fooooooo' // ok
8 // 3. 设置不存在属性
9 state.dong = 'dong' // ok
10 // 4. 删除属性
11 delete state.dong // ok
```

嵌套对象响应式

测试：嵌套对象不能响应

```
1 const state = reactive({
2   bar: { a: 1 }
3 })
4 // 设置嵌套对象属性
5 state.bar.a = 10 // no ok
```

添加对象类型递归

```
1 // 提取帮助方法
2 const isObject = val => val !== null && typeof val === 'object'
3 function reactive(obj) {
4   // 判断是否对象
5   if (!isObject(obj)) {
6     return obj
7   }
8   const observed = new Proxy(obj, {
9     get(target, key, receiver) {
10       // ...
11       // 如果是对象需要递归
12       return isObject(res) ? reactive(res) : res
13     },
14     // ...
15   })
}
```

避免重复代理

重复代理，比如

```
1 | reactive(data) // 已代理过的纯对象
2 | reactive(react) // 代理对象
```

解决方式：将之前代理结果缓存，get时直接使用

```
1 | const toProxy = new WeakMap() // 形如obj:observed
2 | const toRaw = new WeakMap() // 形如observed:obj
3 | function reactive(obj) {
4 |   //...
5 |   // 查找缓存，避免重复代理
6 |   if (toProxy.has(obj)) {
7 |     return toProxy.get(obj)
8 |   }
9 |   if (toRaw.has(obj)) {
10 |     return obj
11 |   }
12 |   const observed = new Proxy(..., ...
13 |   // 缓存代理结果
14 |   toProxy.set(obj, observed)
15 |   toRaw.set(observed, obj)
16 |   return observed
17 | }
18 | // 测试效果
19 | console.log(reactive(data) === state)
20 | console.log(reactive(state) === state)
```

依赖收集

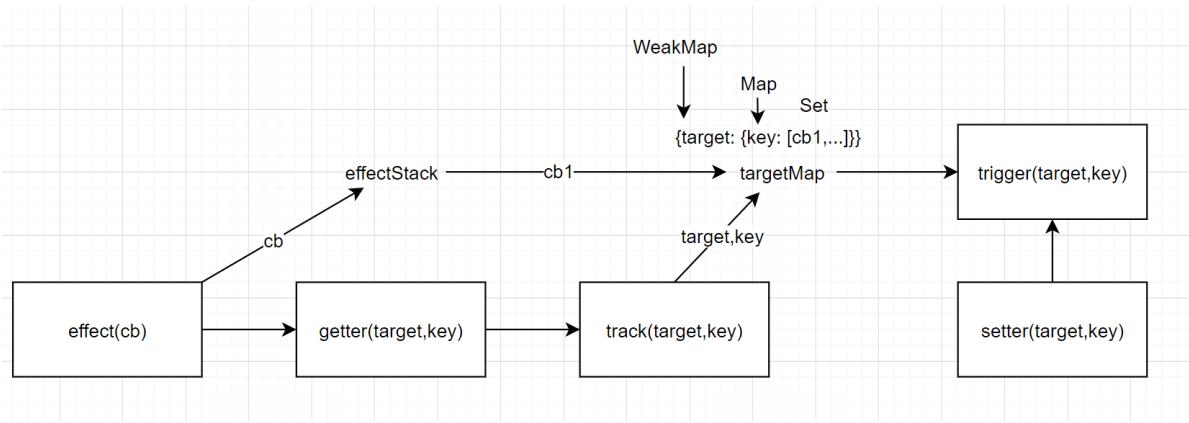
建立响应数据key和更新函数之间的对应关系。

用法

```
1 | // 设置响应函数
2 | effect(() => console.log(state.foo))
3 | // 用户修改关联数据会触发响应函数
4 | state.foo = 'xxx'
```

设计

实现三个函数：effect：将回调函数保存起来备用，立即执行一次回调函数触发它里面一些响应数据的getter track：getter中调用track，把前面存储的回调函数和当前target,key之间建立映射关系 trigger：setter中调用trigger，把target,key对应的响应函数都执行一遍



target, key 和响应函数映射关系

```

1 // 大概结构如下所示
2 // WeakMap Map Set
3 // {target: {key: [effect1,...]}}

```

实现

设置响应函数，创建effect函数

```

1 // 保存当前活动响应函数作为getter和effect之间桥梁
2 const effectStack = []
3 // effect任务：执行fn并将其入栈
4 function effect(fn) {
5   const rxEffect = function() {
6     // 1.捕获可能的异常
7     try {
8       // 2.入栈，用于后续依赖收集
9       effectStack.push(rxEffect)
10      // 3.运行fn，触发依赖收集
11      return fn()
12    } finally {
13      // 4.执行结束，出栈
14      effectStack.pop()
15    }
16  }
17  // 默认执行一次响应函数
18  rxEffect()
19  // 返回响应函数
20  return rxEffect
21}

```

依赖收集和触发

```

1 function reactive(obj) {
2   ...
3   const observed = new Proxy(obj, {
4     get(target, key, receiver) {
5       ...
6       // 依赖收集
7       track(target, key)
8       return isObject(res) ? reactive(res) : res
9     },
10    set(target, key, value, receiver) {

```

```
11 // ...
12 // 触发响应函数
13 trigger(target, key)
14 return res
15 }
16 })
17 }
18 // 映射关系表，结构大致如下：
19 // {target: {key: [fn1, fn2]}}
20 let targetMap = new WeakMap()
21 function track(target, key) {
22 // 从栈中取出响应函数
23 const effect = effectStack[effectStack.length - 1]
24 if (effect) {
25 // 获取target对应依赖表
26 let depsMap = targetMap.get(target)
27 if (!depsMap) {
28 depsMap = new Map()
29 targetMap.set(target, depsMap)
30 }
31 // 获取key对应的响应函数集
32 let deps = depsMap.get(key)
33 if (!deps) {
34 deps = new Set()
35 depsMap.set(key, deps)
36 }
37 // 将响应函数加入到对应集合
38 if (!deps.has(effect)) {
39 deps.add(effect)
40 }
41 }
42 }
43 // 触发target.key对应响应函数
44 function trigger(target, key) {
45 // 获取依赖表
46 const depsMap = targetMap.get(target)
47 if (depsMap) {
48 // 获取响应函数集合
49 const deps = depsMap.get(key)
50 if (deps) {
51 // 执行所有响应函数
52 deps.forEach(effect => {
53 effect()
54 })
55 }
56 }
57 }
```