vue 生命周期

参考：https://cn.vuejs.org/v2/guide/instance.html#%E7%94%9F%E5%91%BD%E5%91%A8%E6%9C%9F%E5%9B%BE%E7%A4%BA

```
beforeCreate---->可以在这加个loading事件
created---->在这结束loading，还做一些初始化，实现函数自执行

beforeMount
mounted

beforeUpdate
updated

beforeDestroy---->钩子函数在实例销毁之前调用。在这一步，实例仍然完全可用。一般在这里销毁事件：页面销毁前注销该事件
destroyed---->
```

updated和 mouted

```
在update阶段使用this.$refs.xxx，就100%能找到该DOM节点。
updated与mounted不同的是，在每一次的DOM结构更新，vue都会调用一次updated(){}钩子函数！而mounted仅仅
只执行一次而已
```

vue组件优化1:按需加载:组件的异步加载（按需加载组件）

```
require：运行时调用，理论上可以运用在代码的任何地方，
import：编译时调用，必须放在文件开头

懒加载：component: resolve => require(['@/view/index.vue'], resolve)
用require这种方式引入的时候，会将你的component分别打包成不同的js，加载的时候也是按需加载，只用访问这个路由网址时才会加载这个js

非懒加载：component: index
如果用import引入的话，当项目打包时路由里的所有component都会打包在一个js中，造成进入首页时，需要加载的内容过多，时间相对比较长
vue的路由配置文件(routers.js)，一般使用import引入的写法，当项目打包时路由里的所有component都会打包在一个js中，在项目刚进入首页的时候，就会加载所有的组件，所以导致首页加载较慢，
而用require会将component分别打包成不同的js，按需加载，访问此路由时才会加载这个js，所以就避免进入首页时加载内容过多。
```

异步写法

```
<template>
  <div>
    <hell />
  </div>
</template>

<script>
export default {
```

```
  components: {
    hell(resolve) {
      require(["../components/hell2.vue"], resolve);
    }
  },
  data() {
    return {};
  }
};
</script>
```

如果我们想把一些组件和某一个子组件一起打包为代码块，通过添加注释的方式即可

```
const routes = [
    {
     name: "MyCompoent",
     path: 'my-compoent',
     component: () => import(/* webpackChunkName: 'group-btn' */ './my-
component')
    }
]
```

路由懒加载方法:通过异步组件和webpacm代码分割，实现路由懒加载，按需加载，提升路由页面加载速度。

路由懒加载方法
通过工厂函数返回一个Promise对象，异步加载组件

import() 返回一个promise对象

```
那么通过工厂函数返回
var myComponent = () => import('./my-component')

const routes = [
    {
     name: "MyCompoent",
     path: 'my-compoent',
     component: myComponent
    }
]
```

当页面很多，组件很多的时候，SPA页面在首次加载的时候，就会变的很慢。这是因为vue首次加载的时候把可能一开始看不见的组件也一次加载了，这个时候就需要对页面进行优化，就需要异步组件了
什么是异步组件?
异步组件就是定义的时候什么都不做，只在组件需要渲染（组件第一次显示）的时候进行加载渲染并缓存，缓存是以备下次访问。

为什么用异步组件?
在大型应用中，功能不停地累加后，核心页面已经不堪重负，访问速度愈来愈慢。为了解决这个问题我们需要将应用分割成小一些的代码块，并且只在需要的时候才从服务器加载一个模块，从而提高页面加载速度。
1.webpack和ES6推荐返回一个 Promise（推荐）

```
// 下面2行代码，没有指定webpackChunkName，每个组件打包成一个js文件。
const ImportFuncDemo1 = () => import('../components/ImportFuncDemo1')
const ImportFuncDemo2 = () => import('../components/ImportFuncDemo2')
```

```
// 下面2行代码，指定了相同的webpackChunkName，会合并打包成一个js文件。
// const ImportFuncDemo = () => import(/* webpackChunkName: 'ImportFuncDemo' */
'../components/ImportFuncDemo')
// const ImportFuncDemo2 = () => import(/* webpackChunkName: 'ImportFuncDemo' */
'../components/ImportFuncDemo2')
export default new Router({
    routes: [
        {
            path: '/importfuncdemo1',
            name: 'ImportFuncDemo1',
            component: ImportFuncDemo1
        },
        {
            path: '/importfuncdemo2',
            name: 'ImportFuncDemo2',
            component: ImportFuncDemo2
        }
    ]
})
```

3.高级路由

```
const AsyncComponent = () => ({
  // 需要加载的组件 (应该是一个 `Promise` 对象)
  component: import('./MyComponent.vue'),
  // 异步组件加载时使用的组件
  loading: LoadingComponent,
  // 加载失败时使用的组件
  error: ErrorComponent,
  // 展示加载时组件的延时时间。默认值是 200 (毫秒)
  delay: 200,
  // 如果提供了超时时间且组件加载也超时了，
  // 则使用加载失败时使用的组件。默认值是：`Infinity`
  timeout: 3000
})
{
    path: '/AsyncComponent',
    name: 'AsyncComponent',
    component: AsyncComponent
},
```

Vue中组件间传值常用的几种方式 2021.07.12面试题

主要分为两类：
1.父子组件间的传值
2.非父子组件间的传值

1.父子组件间的传值

```
第一种方式：
props

第二种方式：ref 实现通信
1.如果ref用在子组件上，指向的是组件实例，可以理解为对子组件的索引，通过ref可能获取到在子组件里定义的属性和方法。
```

2.如果ref在普通的 DOM 元素上使用，引用指向的就是 DOM 元素，通过$ref可能获取到该DOM 的属性集合，轻松访问到DOM元素，作用与JQ选择器类似。

父：

```
<template>
  <div>
     <h1>我是父组件！</h1>
   <child ref="msg"></child>
  </div>
</template>
mounted() {
  console.log( this.$refs.msg);
  this.$refs.msg.getMessage('我是子组件一！')
}
```

子：

```
methods:{
  getMessage(m){
   this.message=m;
  }
}
```

拓展：ref是如何获取dom元素的

```
<div ref="testDom">11111</div>
getTest() {
  console.log(this.$refs.testDom)
}
```

如果想要真正地在DOM加载完成后拿到数据，就需要调用VUE的全局api

```
this.$nextTick(() => {})
mounted (){
    this.$nextTick(() => {
        this.$refs.index.style.paddingBottom = this.$refs.sus.clientHeight +
'px';
    });
}
```

拓展：vue实现响应式并不是数据发生变化后dom立即变化，而是按照一定的策略来进行dom更新。

```
$nextTick是在下一次dom更新循环结束之后执行延迟回调，在修改数据之后使用这个方法，立即更新dom.

this.tableData.specification.forEach((element, index) => {
 this.$nextTick(() => {
     this.tableData.specification[index].commission = ''
 })
})
//在监视数据改变的语句后，加上this.$nextTick(function(){ })。里面的函数在DOM渲染后执行
```

注意：

parent和children是获取组件和子组件的实例，只不过$children是一个数组集合，需要我们记住组件顺序才可以.

computed和watch

3.1 computed特性
1.是计算值，
2.应用：就是简化tempalte里面{{}}计算和处理props或$emit的传值
3.具有缓存性，页面重新渲染值不变化,计算属性会立即返回之前的计算结果，而不必再次执行函数
当一个属性受多个属性影响的时候就需要用到computed

3.2 watch特性
1.是观察的动作，
2.应用：监听props，$emit或本组件的值执行异步操作
3.无缓存性，页面重新渲染时值不变化也会执行
`watch` 还可以做一些特别的事情，例如监听页面路由，当页面跳转时，我们可以做相应的权限控制，拒绝没有权限的用户访问页面。

03.常用vue 指令

v-text：更新元素的 textContent。如果要更新部分的 textContent ，需要使用 {{ Mustache }} 插值。
v-html：更新元素的 innerHTML
v-show：------->切换元素的 display CSS 属性。block为显示，none为隐藏
v-if：------->控制dom节点的存在与否来控制元素的显隐
v-else：表示否则（与编程语言中的else是同样的意思）
v-else-if：（与编程语言中的else if是同样的意思）
v-for：可以循环数组，对象，字符串，数字，
v-on：绑定事件监听器。事件类型由参数指定。
v-bind：动态地绑定一个或多个属性（特性），或一个组件 prop 到表达式。
v-model：在表单控件或者组件上创建双向绑定
v-pre：跳过这个元素和它的子元素的编译过程。可以用来显示原始 Mustache 标签。跳过大量没有指令的节点会加快编译。
v-cloak：这个指令保持在元素上直到关联实例结束编译。和 CSS 规则如 [v-cloak] { display: none } 一起用时，这个指令可以隐藏未编译的 Mustache 标签直到实例准备完毕。
v-once:只渲染元素和组件一次。随后的重新渲染，元素/组件及其所有的子节点将被视为静态内容并跳过。这可以用于优化更新性能。

2、子组件向父组件传值

方式1：使用$emit传递事件给父组件，父组件监听该事件

```
父:
<m-child v-bind:msg="p2C" @showMsg='getChild' ref='child'></m-child>
子:
methods: {
    pushMsg() {
      this.$emit("showMsg", "这是子组件===>父组件的值");
    }
},
```

方式2：

```
第二种方式:
使用$parent.获取父组件对象，然后再获取数据对象,子组件代码:
mounted() {
    this.msg22 = this.$parent.msg2;
}
```

Vue-router

Vue-router跳转和location.href有什么区别

```
答：使用location.href='/url'来跳转，简单方便，但是刷新了页面；
     使用history.pushState('/url')，无刷新页面，静态跳转；
引进router，然后使用router.push('/url')来跳转，使用了diff算法，实现了按需加载，减少了dom的消耗。
其实使用router跳转和使用history.pushState()没什么差别的，因为vue-router就是用了history.pushState()，尤其是在history模式下。
```

params和query的区别

用法：query要用path来引入，params要用name来引入，接收参数都是类似的，分别是this.$route.query.name和this.$route.params.name。
url地址显示：query更加类似于我们ajax中get传参，params则类似于post，说的再简单一点，前者在浏览器地址栏中显示参数，后者则不显示
注意点：query刷新不会丢失query里面的数据
params刷新 会 丢失 params里面的数据。

为什么使用key

```
需要使用key来给每个节点做一个唯一标识，Diff算法就可以正确的识别此节点。
作用主要是为了高效的更新虚拟DOM。
```

vuex start

vuex start

vuex

vuex有哪几种属性
state => 基本数据(数据源存放地)
getters => 从基本数据派生出来的数据
mutations => 提交更改数据的方法，同步!
actions => 像一个装饰器，包裹mutations，使之可以异步。
modules => 模块化Vuex
辅助函数:
Vuex提供了mapState、MapGetters、MapActions、mapMutations等辅助函数给开发在vm中处理store。

使用:

```
import Vuex from 'vuex';
Vue.use(Vuex); // 1. vue的插件机制，安装vuex
let store = new Vuex.Store({ // 2.实例化store，调用install方法
    state,
    getters,
```

```
    modules,
    mutations,
    actions,
    plugins
});
new Vue({ // 3.注入store，挂载vue实例
    store,
    render: h=>h(app)
}).$mount('#app');
```

1.vuex的store是如何挂载注入到组件中呢?

```
import Vuex from 'vuex';
Vue.use(vuex);// vue的插件机制
```

2.利用vue的插件机制，使用Vue.use(vuex)时，会调用vuex的install方法，装载vuex，install方法的代码如下:

```
export function install (_Vue) {
  if (Vue && _Vue === Vue) {
    if (process.env.NODE_ENV !== 'production') {
      console.error(
        '[vuex] already installed. Vue.use(Vuex) should be called only once.'
      )
    }
    return
  }
  Vue = _Vue
  applyMixin(Vue)
}
```

applyMixin方法使用vue混入机制，vue的生命周期beforeCreate钩子函数前混入vuexInit方法，核心代码如下:

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

function vuexInit () {
    const options = this.$options
    // store injection
    if (options.store) {
      this.$store = typeof options.store === 'function'
        ? options.store()
        : options.store
    } else if (options.parent && options.parent.$store) {
      this.$store = options.parent.$store
    }
}
```

总结：分析源码，我们知道了vuex是利用vue的mixin混入机制，在beforeCreate钩子前混入vuexInit方法，vuexInit方法实现了store注入vue组件实例，并注册了vuex store的引用属性$store。store注入过程如下图所示:

2.疑问2：vuex的state和getters是如何映射到各个组件实例中响应式更新状态呢?

store实现的源码在src/store.js
从上面源码，我们可以看出Vuex的state状态是响应式，是借助vue的data是响应式，将state存入vue实例组件的data中；Vuex的getters则是借助vue的计算属性computed实现数据实时监听。
computed计算属性监听data数据变更主要经历以下几个过程：
1、我们在源码中找到resetStoreVM核心方法：

```
function resetStoreVM (store, state, hot) {
  const oldVm = store._vm

  // 设置 getters 属性
  store.getters = {}
  const wrappedGetters = store._wrappedGetters
  const computed = {}
  // 遍历 wrappedGetters 属性
  forEachValue(wrappedGetters, (fn, key) => {
    // 给 computed 对象添加属性
    computed[key] = partial(fn, store)
    // 重写 get 方法
    // store.getters.xx 其实是访问了store._vm[xx]，其中添加 computed 属性
    Object.defineProperty(store.getters, key, {
      get: () => store._vm[key],
      enumerable: true // for local getters
    })
  })

  const silent = Vue.config.silent
  Vue.config.silent = true
  // 创建Vue实例来保存state，同时让state变成响应式
  // store._vm._data.$$state = store.state
  store._vm = new Vue({
    data: {
      $$state: state
    },
    computed
  })
  Vue.config.silent = silent

  // 只能通过commit方式更改状态
  if (store.strict) {
    enableStrictMode(store)
  }
}
```

vuex end

vuex end

Object.defineProperty()用来生成或修改一个对象属性,用法:

Object.defineProperty(obj, prop, descriptor),--->返回值：传入函数的对象。即第一个参数obj
obj 要在其上定义属性的对象。
prop 要定义或修改的属性的名称。
descriptor 将被定义或修改的属性描述符。

obj： 必需。目标对象
prop： 必需。需定义或修改的属性的名字
descriptor： 必需。目标属性所拥有的特性
作用：当修改或定义对象的某个属性的时候，给这个属性添加一些特性:
1）设置的特性总结:

```
value:     设置属性的值
writable:     属性的值是否可以被重写。设置为true可以被重写；设置为false，不能被重写。默认为false。
enumerable:     设置为true可以被枚举；设置为false，不能被枚举。默认为false。
configurable:     目标属性是否可以被删除或是否可以再次修改特性 true | false,默认为false。
```

属性：writable

```
var obj = {}
//1、writable设置为false，不能重写。
Object.defineProperty(obj,"newKey",{
    value:"hello",
    writable:false
});
//2、更改newKey的值,发现值并没有修改
obj.newKey = "change value";
console.log( obj.newKey );  //hello
```

属性：enumerable

属性：configurable

1. 目标属性是否可以使用delete删除，目标属性是否可以再次设置特性
2. 设置为true可以被删除或可以重新设置特性；设置为false，不能被可以被删除或不可以重新设置特性。默认为false。

```
var obj = {}

//1、configurable设置为false，不能被删除。
Object.defineProperty(obj,"newKey",{
    value:"hello",
    writable:false,
    enumerable:false,
    configurable:false
});

//2、删除属性：实际上没有被删除
delete obj.newKey;
console.log( obj.newKey ); //hello
```

getter/setter

1. getter 是一种获得属性值的方法，setter是一种设置属性值的方法。
2. 当使用了getter或setter方法，不允许使用writable和value这两个属性
3. get或set不是必须成对出现，任写其一就可以,如果不设置方法，则get和set的默认值为undefined

```
var obj = {};
var initValue = 'hello';
Object.defineProperty(obj,"newKey",{
    get:function (){
        return initValue;             //当获取值的时候触发的函数
    },
    set:function (value){
        initValue = value;         //当设置值的时候触发的函数,设置的新值通过参数value拿到
    }
});
//获取值
console.log( obj.newKey );  //hello
//设置值
obj.newKey = 'change value';
console.log( obj.newKey ); //change value
```

```
//1、任意创建一个对象obj
var obj = {
    test:"hello"
}
// 原始值 obj = {test: "hello"}


//2、修改obj "test"值为"test的新值"
Object.defineProperty(obj,"test",{
    configurable:true | false,
    enumerable:true | false,
    value:'test的新值',
    writable:true | false
});
// 修改后 obj = {test: "test的新值"}


//3、对象新添加的属性的特性描述
Object.defineProperty(obj,"newKey",{
    configurable:true | false,
    enumerable:true | false,
    value:"newValue",
    writable:true | false
});
// 修改后 obj = {test: "test的新值", newKey: "newValue"}
```

```
//-------->设置该属性为数据描述符
var person = {}
Object.defineProperty(person,'a',{
     configurable:true, //可修改默认属性
     enumerable:true, //可枚举
     writable:true, //可修改这个属性的值
     value:1 //定义一个初始的值为1
})
```

```
console.log(person) //{a: 1}
person.a=2
console.log(person) //{a: 2}

for(var k in person){
     console.log(k) //a
}
```

```
//------->设置该属性为读取描述符
var person = {
    a:1
}
Object.defineProperty(person,'a',{
     get(){
         return 3 //当访问这个属性的时候返回3
     },
     set(val){
         console.log(val)//当设置这个属性的时候执行,val是设置的值
     }
})
person.a// 3,我们明明写的是a:1,怎么返回的3呢?这就是get()的威力了
person.a = 5// 5,相应的设置的时候执行了set()函数
```