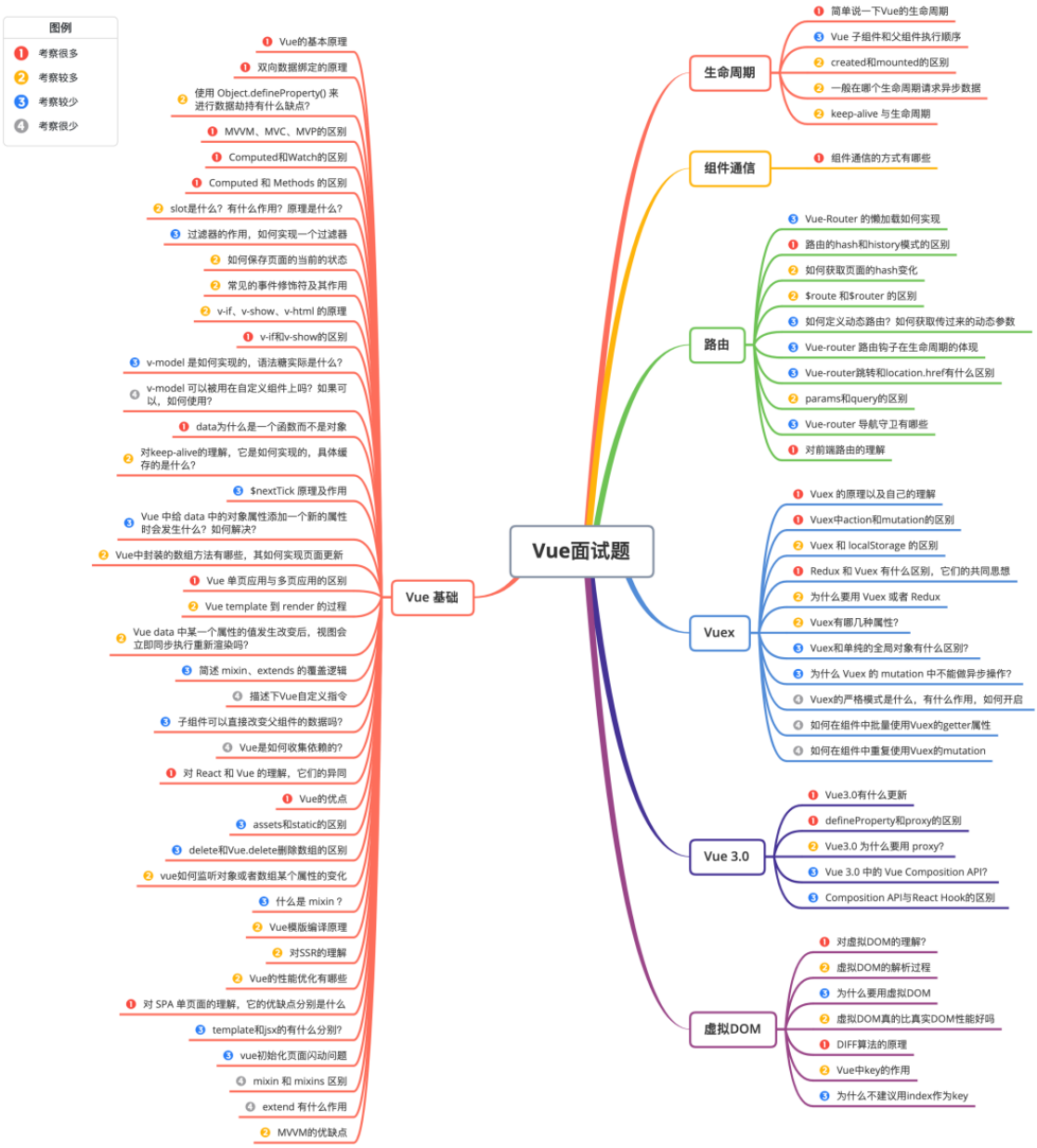
一、Vue基础
1. data为什么是一个函数而不是对象?
2. 为什么vue3.0使用proxy，抛弃了object.defineProperty?
3. 对SSR的理解
4. Vue如何监听对象或者数组某个属性的变化
5. 什么是mixin?
6. 对vue组件化的理解
7. Vue的性能优化有哪些?
8. v-model的实现原理
9、常见的事件修饰符及其作用
10、Vue单页面应用与多页面应用的区别
11、Vue是如何收集依赖的?
12、Vue的优点
13、assets 和 static的区别
14、delete 和 Vue.delete删除数组的区别
15、extend有什么作用
16、mixin 和 mixins 区别
17、MVVM的优缺点?
二、Vue原理篇
1. v-if、v-show、v-html的原理
2. vue的基本原理
3. 双向数据绑定原理
4. $nextTick原理及作用
    原理
5. Vue2.x里的 object.defineProperty()
    关于getter 和 setter机制
    关于数据劫持
    怎样使用Object.defineProperty操作数组?
6. Vue3.0 里的Proxy
    可以直接修改对象
    可以直接修改数组
    直接修改函数
7. 模板编译原理
三、Vue区别篇
1. MVVM、MVC、MVP的区别
    MVC
    MVVM（Vue使用）
    MVP
2. v-if 和 v-show的区别
3. Computed和Watch的区别
4. created 和 mounted 的区别
5. Vue-router跳转和location.href有什么区别
6. params 和 query的区别
7. Vuex 和localStorage 的区别
四、Vue生命周期
1. 说一下Vue的生命周期
2. Vue子组件和父组件执行顺序
3. 一般在哪个生命周期请求异步数据
4. keep-alive中的生命周期有哪些
五、组件通信
1. props / $emit
2. eventBus事件总线（$emit/$on）
3. 注入依赖（provide/inject）
4. ref / $refs
5. $parent / $children

6. $attrs / $listeners
总结
六、路由
1. Vue-Router的懒加载如何实现
2. 路由的hash 和 history模式的区别
3. 如何获取页面的hash变化
4. $route 和 $router 的区别
5. 如何定义动态路由？如何获取传过来的动态参数？
6. Vue-router 导航守卫
7. 对前端路由的理解
七、Vuex
1. vuex的原理
2. Vuex中action 和 mutation 的区别
3. 为什么Vuex 的 mutation中不能做异步操作？
八、Vue3.0
1. Vue3.0 有什么更新
九、虚拟DOM
前言
1. 什么是虚拟DOM？
2. 为什么要有虚拟DOM？
3. DOM-Diff算法
patch
4. Vue中key的作用
为什么不建议index作为key？



一、Vue基础

1. data为什么是一个函数而不是对象?

> 简单来说就是，因为组件是可以复用的，JS里对象是引用关系，如果组件data是一个对象，那么子组件中的data属性值会互相污染，产生副作用。
>
> 所以一个组件的data选项必须是一个函数，因此每个实例可以维护一份返回对象的独立拷贝。
> new Vue的实例时不会被复用的，因此不存在以上问题。

Vue组件可能存在多个实例，如果使用对象形式定义data，则会导致它们共用一个data对象，那么状态改变将会影响所有组件实例，这是不合理的；

采用函数形式定义，在initData时会将其作为工厂函数返回全新data对象，有效**规避多实例之间状态污染问题**。而在Vue根实例创建过程中则不存在该限制，也是因为根实例只能有一个，不需要担心这种情况。

2. 为什么vue3.0使用proxy，抛弃了object.defineProperty?

`Object.defineProperty`只能劫持对象的属性，因此我们需要对每个对象的每个属性进行遍历。vue2.X里，是通过 递归+遍历 data对象来实现对数据的监控的，如果属性值也是对象那么需要深度遍历，显然如果能劫持一个完整的对象才是更好的选择。

Proxy可以劫持整个对象，并返回一个新的对象。Proxy不仅可以代理对象，还可以代理数组。还可以代理动态增加的属性。

3. 对SSR的理解

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，服务端渲染只支持beforeCreate 个 created 两个钩子
- 当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

4. Vue如何监听对象或者数组某个属性的变化

当在项目中直接设置数组的某一项的值，或者直接设置对象的某个属性值，这个时候，你会发现页面并没有更新。这是因为 Object.defineProperty()限制，监听不到变化。

解决方式:

this.$set（要改变的数组/对象，要改变的位置/key，要改成什么value）

```
this.$set(this.arr,0,'hhh');  //改变数组
this.$set(this.obj,'c','fsdf');  //改变对象
```

调用一下几个数组的方法:

```
splice()、 push()、pop()、shift()、unshift()、sort()、reverse()
```

vue源码里缓存了array的原型链，然后重写了这几个方法，触发这几个方法的时候会observer数据，意思是使用这些方法不用再进行额外的操作，视图自动进行更新。推荐使用splice方法会比较好自定义，因为splice可以在数组的任何位置进行删除/添加操作。

`vm.$set`的实现原理是:

- 如果目标是数组，直接使用数组的splice方法触发响应式。
- 如果目标是对象，会先判断属性是否存在、对象是否是响应式，最终如果要对属性进行响应式处理，则是通过调用defineReactive 方法进行响应式处理（defineReactive方法就是Vue在初始化对象时，给对象属性采用 Object.defineProperty动态添加getter和setter的功能所调用的方法）

5. 什么是mixin?

- Mixin使我们能够为Vue组件编写可插拔和可重用的功能。

- 如果希望在多个组件之间重用一组组件选项，例如生命周期hook、方法等，则可以将其编写为mixin，并在组件中简单的引用它。
- 然后将mixin的内容合并到组件中。如果你要在mixin中定义生命周期hook，如果有冲突，那么组件首先会执行mixin里的hook，然后再执行自己的hook。
- 值为对象的选项，例如 `methods`、`components`、`directives`，将被合并为同一个对象。两个对象键值冲突时，取 **组件对象**的键值对。

```
// 定义一个 mixin 对象
const myMixin = {
  created() {
    this.hello()
  },
  methods: {
    hello() {
      console.log('hello from mixin!')
    }
  }
}

// 定义一个使用此 mixin 对象的应用
const app = Vue.createApp({
  mixins: [myMixin]
})

app.mount('#mixins-basic') // => "hello from mixin!"
```

6. 对vue组件化的理解

1. 组件是独立和可复用的代码组织单元。组件系统是Vue核心特性之一，它使开发者使用小型、独立和通常可复用的组件构建大型应用
2. 组件化开发能大幅提高应用开发效率、测试性、复用性等
3. 组件使用按分类由：页面组件、业务组件、通用组件
4. vue的组件是基于配置的，我们通常编写的组件是组件配置而非组件，框架后续会生成其他构造函数，他们基于VueComponent，扩展于Vue
5. Vue中常见组件化技术有：属性prop,自定义事件，插槽等，他们主要用于组件通信、扩展等
6. 合理的划分组件，有助于提升应用性能
7. 组件应该是高内聚、低耦合的
8. 遵循单向数据流的原则

7. Vue的性能优化有哪些?

（1）编码阶段

- 尽量减少data中的数据，data中的数据会增加getter和setter，会收集对应的watcher
- v-if和v-for不能连用
- 如果需要使用v-for给每项元素绑定事件时使用事件代理
- SPA页面采用keep-alive缓存组件
- 在更多情况下，使用v-if替代v-show
- key保证唯一
- 使用路由懒加载、异步组件
- 防抖、节流
- 第三方模块按需导入

- 长列表滚动到可视区域动态加载
- 图片懒加载

(2) SEO优化

- 预渲染
- 服务端渲染SSR

(3) 打包优化

- 压缩代码
- Tree Shaking/Scope Hoisting
- 使用cdn加载第三方模块
- 多线程打包happypack
- sourceMap优化

(4) 用户体验

- 骨架屏
- PWA
- 还可以使用缓存（客户端缓存、服务端缓存）优化、服务单开启gzip压缩等

8. v-model的实现原理

vue中 v-model 可以实现数据的双向绑定，但是为什么这个指令就可以实现数据的双向绑定呢？其实 v-model是vue的一个语法糖。即利用v-model绑定数据后，又添加了一个input事件监听。

实现原理：

- v-bind 绑定响应数据
- 触发input事件并传递数据

实例：

```
<input v-model="text"></input>
//等价于
<input :value="text" @input="text = $event.target.value"></input>
//组件中使用：
<custom-input :value="text" @input="$event"></custom-input>
// 根据v-model原理模拟：
<input type="text" id="ipt1">
<input type="text" id="ipt2">
<script>
    var ipt1=document.getElementById('ipt1');
    var ipt2=document.getElementById('ipt2');
    ipt1.addEventListener("input",function(){
        ipt2.value=ipt1.value;
    })
</script>
```

9、常见的事件修饰符及其作用

- `.stop`：等同于JavaScript中的 `event.stopPropagation()`，防止事件冒泡；
- `.prevent`：等同于JavaScript中的 `event.preventDefault()`，防止执行预设的行为（如果事件可取消，则取消该事件，而不停止事件的进一步传播）；
- `.capture`：与事件冒泡的方向相反，事件捕获由外到内；
- `.self`：只会触发自己范围内的事件，不包含子元素；

- `.once`：只会触发一次。

10、Vue单页面应用与多页面应用的区别

概念：

- SPA单页面应用（SinglePage Web Application），指只有一个主页面的应用，一开始只需要加载一次js，css等相关资源。所有内容都包含在主页面，对每一个功能模块组件化。单页面应用跳转，就是切换相关组件，仅仅刷新局部资源。
- MAP多页面应用（MultiPage Application），指有多个独立页面的应用，每个页面必须重复加载js、css等相关资源。多页面应用跳转需要整个页资源刷新。

对比项 \ 模式	SPA	MPA
结构	一个主页面 + 许多模块的组件	许多完整的页面
体验	页面切换快，体验佳；当初次加载文件过多时，需要做相关的调优。	页面切换慢，网速慢的时候，体验尤其不好
资源文件	组件公用的资源只需要加载一次	每个页面都要自己加载公用的资源
适用场景	对体验度和流畅度有较高要求的应用，不利于 SEO（可借助 SSR 优化 SEO）	适用于对 SEO 要求较高的应用
过渡动画	Vue 提供了 transition 的封装组件，容易实现	很难实现
内容更新	相关组件的切换，即局部更新	整体 HTML 的切换，费钱（重复 HTTP 请求）
路由模式	可以使用 hash ，也可以使用 history	普通链接跳转
数据传递	因为单页面，使用全局变量就好（Vuex）	cookie 、localStorage 等缓存方案，URL 参数，调用接口保存等
相关成本	前期开发成本较高，后期维护较为容易	前期开发成本低，后期维护就比较麻烦，因为可能一个功能需要改很多地方

11、Vue是如何收集依赖的?

在初始化Vue的每个组件时，会对组件的data进行初始化，就会将普通对象变成响应式对象，在这个过程中便会进行依赖收集的相关逻辑，如下所示：

```
function defineReactive(obj,key,val){
    const dep = new Dep();
    ...
```

```
    Object.defineProperty(obj,key,{
        ...
        get:function reactiveGetter(){
            if(Dep.target){
                dep.depend();
                ...
            }
            return val
        }
        ...
    })
}
```

以上只保留了关键代码，主要就是 `const dep = new Dep()` 实例化一个Dep的实例，然后在get函数中通过 `dep.depend()` 进行依赖收集。

（1）Dep

Dep是整个依赖收集的核心。Dep是一个class，其中有一个关键的静态属性static，它指向了一个全局唯一 Watcher，保证了同一时间全局只有一个watcher被计算，另一个属性subs则是一个Watcher的数组，**所以Dep实际上就是对Watcher的管理。**

（2）Watcher

Watcher是一个class，它定义了一些方法，其中和依赖收集相关的主要有get、addDep等。

（3）过程

在实例化Vue时，依赖收集的相关过程如下：

初始化状态 initState，这中间便会通过 defineReactive将数据变成响应式对象，其中的getter部分便是用来收集依赖的。初始化最终会走mount过程，其中会实例化Watcher，进入Watcher中，便会执行this.get()方法。

```
updateComponent = () =>{
    vm._update(vm._render())
}
new Watcher(vm,updateComponent)
```

get 方法中的pushTarget实际上就是把 Dep.target赋值为当前的watcher。

this.getter.call(vm,vm)，这里的getter会执行vm._render()方法，在这个过程中便会触发数据对象的getter。那么每个对象值的getter都持有一个dep，在触发getter的时候会调用dep.depend()方法，也就会执行Dep.target.addDep(this)。刚才Dep.target已经被赋值为watcher，于是便会执行addDep方法，然后走到dep.addSub()方法，便将当前的watcher订阅到这个数据持有的dep的subs中，这个目的是为后续数据变化时候能通知到哪些subs做准备。所以在vm. _ render()过程中，会触发所有数据的getter，这样便已经完成了整个依赖收集的过程。

12、Vue的优点

- **轻量级框架**：只关注视图层，是一个构建数据的视图集合，大小只有几十KB；
- **简单易学**：中文文档
- **双向数据绑定**：保留了 angular的特点，在数据操作方面更为简单；
- **组件化**：保留了 react 的优点，实现了 HTML 的封装和重用，在构建单页面应用方面有着独特的优势。
- **视图、数据、结构分离**：使数据的更改更为简单，不需要进行逻辑代码的修改，只需要操作数据就能完成相关操作。

- **虚拟DOM**：dom操作是非常耗费性能的，不再使用原生的 DOM操作节点，极大解放DOM操作，但具体操作还是DOM，不过是换了另一种方式
- **运行速度更快**：相较于 react 而言，同样是操作虚拟DOM，就性能而言，Vue存在很大的优势。

13、assets 和 static的区别

相同点：`assets` 和 `static` 两个都是存放静态资源文件。项目中所需要的资源文件图片，字体图标，样式文件等都可以放在这两个文件下。

不同点：

- `assets` 中存放的静态资源文件在项目打包时，也就是运行 `npm run build` 时会将 `assets` 中放置的静态资源文件进行打包上传，所谓打包简单点可以理解为压缩体积，代码格式化。而压缩后的静态资源文件最终也会放置在 `static` 文件中跟着 `index.html` 一同上传至服务器。
- `static` 中放置的静态文件就不会要走打包压缩格式化等流程，而是直接进入打包好的目录，直接上传至服务器。
    - 因为避免了压缩直接进行上传，在打包时会提高一定的效率，但是 `static` 中的资源文件由于没有进行压缩等操作，所以文件的体积也就相对于 `assets` 中打包后的文件提交较大点。在服务器中就会占据更大的空间。

建议：将项目中 `template` 需要的样式文件js文件等都可以放置在 `assets` 中，走打包这一流程。减少体积。而项目中引入的第三方的资源文件如 `iconfont.css` 等文件可以放置在 `static` 中，因为这些引入的第三方文件已经经过处理，不再需要处理，直接上传。

14、delete 和 Vue.delete删除数组的区别

- `delete` 只是被删除的元素变成了 `empty/undefined`，其他的元素的键值还是不变。
- `Vue.delete` 直接删除了数组，改变了数组的键值。

15、extend有什么作用

这个API很少用到，作用是扩展组件生成一个构造器，通常会与 `$mount` 一起使用。

```
//创建组件构造器
let Component = Vue.extend({
    template: '<div>test</div>'
})
//挂载到#app上
new Component().$mount('#app')
//除了上面的方式，还可以用来扩展已有的组件
let SuperComponent = Vue.extend(Component)
new SuperComponent({
    created(){
        console.log(1)
    }
})
new SuperComponent().$mount('#app')
```

16、mixin 和 mixins 区别

`mixin` 用于全局混入，会影响到每个组件实例，通常插件都是这样做初始化的。

```
Vue.mixin({
    beforeCreate(){
        //...逻辑
        //这种方式会影响到每个组件的 beforeCreate钩子函数
    }
})
```

虽然文档不建议在应用中直接使用 `mixin`，但是如果不滥用的话也是很有帮助的，比如可以全局混入封装好的 `ajax` 或者一些工具函数等。

`mixins` 应该是最常用的扩展组件的方式了。如果多个组件中有相同的业务逻辑，就可以将这些逻辑剥离出来，通过 `mixins` 混入代码，比如上拉下拉加载数据这种逻辑等

另外要注意的是 `mixins` 混入的钩子函数会先于组件内的钩子函数执行，并且在遇到同名选项的时候也会选择性的进行合并。

17、MVVM的优缺点?

优点:

- 分离视图和模型，降低代码耦合，提高视图或者逻辑的重用性。
- 提高可测试性：ViewModel的存在可以帮助开发者更好地编写测试代码
- 自动更新DOM：利用双向绑定，数据更新后视图自动更新，让开发者从繁琐的手动DOM中解放。

缺点:

- Bug很难被调试: 因为使用双向绑定的模式，当你看到界面异常了，有可能是你View的代码有Bug，也可能是Model的代码有问题。数据绑定使得一个位置的Bug被快速传递到别的位置，要定位原始出问题的地方就变得不那么容易了。另外，数据绑定的声明是指令式地写在View的模版当中的，这些内容是没办法去打断点debug的
- 一个大的模块中model也会很大，虽然使用方便了也很容易保证了数据的一致性，当时长期持有，不释放内存就造成了花费更多的内存
- 对于大型的图形应用程序，视图状态较多，ViewModel的构建和维护的成本都会比较高。

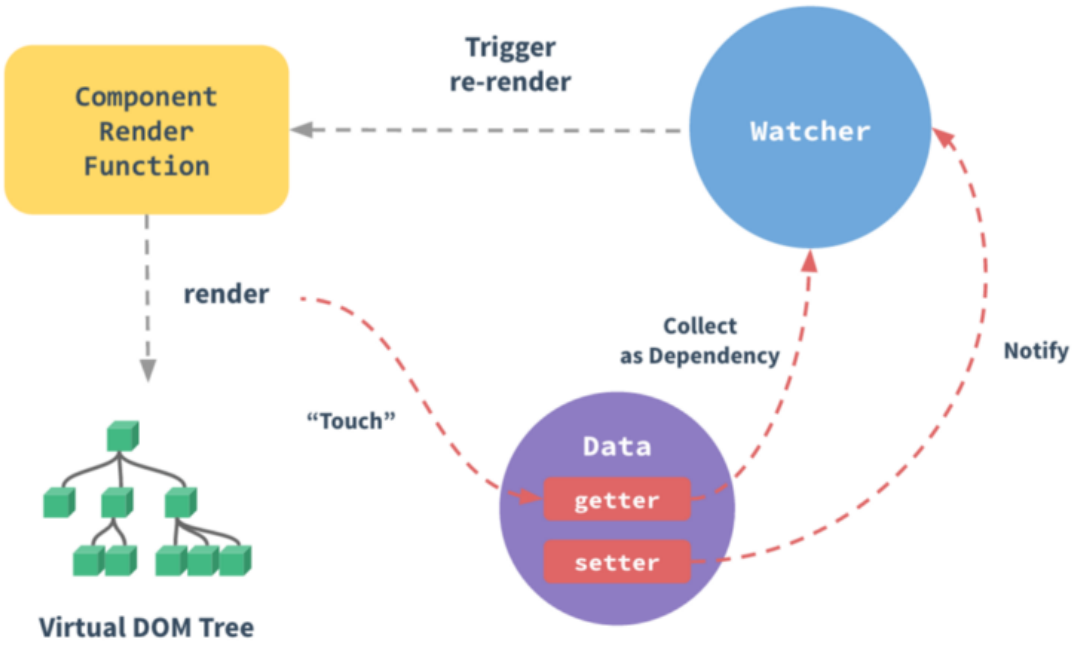
二、Vue原理篇

1. v-if、v-show、v-html的原理

- `v-if` 会调用addCondition方法，生成vnode的时候会忽略对应节点，render的时候就不会渲染；
- `v-show` 会生成VNode，render的时候也会渲染成真实节点，只是在render过程中会在节点的属性中修改show属性值，也就是常说的display。
- `v-html` 会先移除节点下的所有节点，调用html方法，通过addProp添加innerHTML属性，归根结底还是设置innerHTML为v-html的值。

2. vue的基本原理

当一个vue实例创建时，vue会遍历data中的属性，用 `Object.defineProperty` (vue3.0使用proxy)将他们转化为 getter/setter，并且在内部追踪相关依赖，在属性被访问和修改时通知变化。每个组件实例都有相应的watcher程序实例，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。



3. 双向数据绑定原理

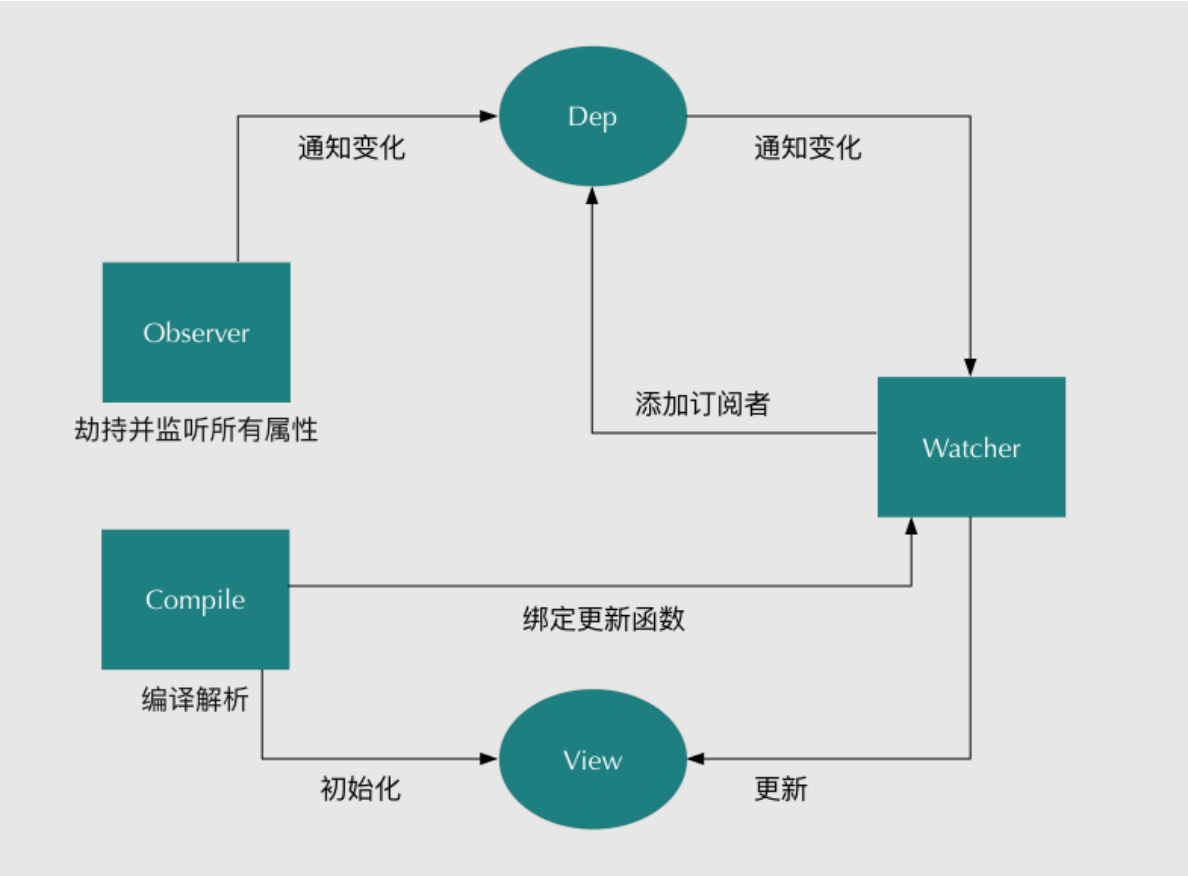
vue.js是采用 **数据劫持**结合 **发布者-订阅者模式**的方式，通过 `Object.defineProperty()` 来劫持各个属性的setter，getter，在数据变动时发布消息给订阅者，触发相应的监听回调。主要分为以下几个步骤：

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

   ① 在自身实例化时往属性订阅器（dep）里面添加自己

   ② 自身必须有一个update()方法

   ③ 待属性变动 dep.notice()通知时，能调用自身的update()方法，并触发Compile中绑定的回调，则功成身退
4. MVVM作为数据绑定的入口，整个Observer、Compile和 Watcher三者，通过 Observer来监听组件的model数据变化，通过Compile来解析编译模板指令，最终利用Watcher搭起Observer和Compile之间的通信桥梁，达到数据变化 -》视图更新；视图交互变化（input）-》数据model变更的双向绑定效果。



4. $nextTick原理及作用

> Vue的nextTick其本质是对JavaScript执行原理EventLoop（事件循环）的一种应用。

nextTick的核心是利用了如promise、MutationObserver、setImmediate、setTimeout的原生JavaScript方法来模拟应对的微/宏任务的实现，本质是为了利用JavaScript的这些异步回调任务队列来实现Vue框架中自己的异步回调队列。

nextTick不仅是vue内部的异步队列的调用方法，同时也允许开发者在实际项目中使用这个方法来满足实际应用中**对DOM更新数据时机的后续处理。**

原理

1. vue用异步队列的方式来控制DOM更新和nextTick回调后执行。
2. 微任务因为以高优先级特性，能确保队列中微任务在一次事件循环前被执行完毕。
3. 考虑兼容问题，vue做了微任务向宏任务的降级方案。

5. Vue2.x里的 object.defineProperty()

> 在正常操作下，这个方法不可修改、不可枚举、不可删除。除非设置了true。
>
> defineProperty ----> 定义属性，接收三个参数（要增加的对象，要增加的属性，描述对象），即 `Object.defineproperty(obj, prop , descriptor)`，默认返回的是obj。
>
> 劫持数据---》给对象进行扩展---》属性进行设置。

```
function defineProperty (){
    var _obj = {};
    Object.defineProperty(_obj, 'a' ,{
        //一下都是纯的数据描述，是否可以三改
        value:1,
        writable:true,  //可修改
        enumerable:true,  //可枚举
        configurable:true  //可删除
    })
}
```

关于getter 和 setter机制

每定义一个属性的时候 getter setter 机制

```
function defineProperty(){
    var _obj = {};
    var a = 1;
    //同时定义多个属性用es
    Object.defineProperties(_obj,{
        a:{
            get(){
                return '"这是" + a';
            },
            set(newVal){
                a = newVal;
                console.log('the value "a" 被定义为了'+ a);  //这里的a其实就是被
newVal赋值了
            }
        },
        b:{}
    })
}

var obj = defineProperty();
obj.a = 2; //调用set方法，将2的值传递给newVal，同时改变变量a的值，输出：the value "a" 被
定义为了2
console.log(obj.a);  //调用get方法，获取到a的值，输出：2。
```

> 在描述 descriptor里**不能**同时存在 **writable，enumerable，configurable 和set()、get()共存**

在上述代码中，打印 `obj.a` 的值输出了2，这是因为我们将 obj.a赋值为了2，调用了set方法，同时也会修改原有定义的a。故当下一次再调用get方法获取数据的时候，会去看看原有的数据有没有被修改，返回的是最新的数据。

关于数据劫持

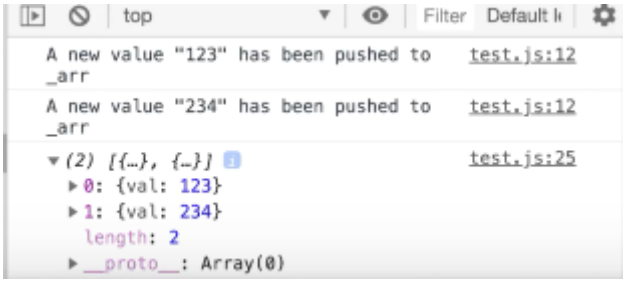
定义：数据劫持，指的是访问或者修改对象的某个属性时，通过一段代码拦截这个行为，**进行额外的操作**（比如加上console语句，或者输出一些长句子，向上面这样就是进行了额外的操作）或者修改返回结果。

怎样使用Object.defineProperty操作数组?

因为Object.defineProperty不能监听数组的变化，数组的push、pop、shift、unshift、splice、sort、reverse方法不会被触发。

```
function DataArr(){
    var _val = null; //object
    var _arr = [];
    Object.defineProperty(this,'val',{
        get:function(){
            return _var;
        }
        set:function(newVal){
            _val = newVal;
            _arr.push({val:_val});
            console.log('A new value"' + _val + '" has been pushed to _arr')
        }
    });
    this.getArr = function(){
        return _arr;
    }
}

//使用new创建一个对象，这时this就指向创建的这个变量
var dataArr = new DataArr();
dataArr.val = 123;
dataArr.val = 234;
console.log(dataArr.getArr());
```



这里使用创建对象的形式来改变数组。

6. Vue3.0 里的Proxy

> Proxy是代理的意思，其实就是代理一个对象，替这个对象完成想要的功能。
>
> Proxy可以直接处理对象、数组、函数等引用类型的值。

形式：`Proxy(target,handler)`

- target：目标对象，你要进行处理的对象
- handler 容器　无数可以处理的对象方法，自定义对象属性的获取，复制，枚举，函数调用等功能。

可以直接修改对象

```
var target = {
    a:1,
    b:2
}
let proxy = new Proxy(target,{
    //get接收两个参数，第一个是要改变的对象，第二个是对象属性
    get(target,prop){
        return 'this is property value' + target[prop];
    }
    //set接收三个参数，第一个是要改变的对象，第二个是对象属性，第三个是要修改的值
    set(target,prop,value){
    target[prop] = value;
    console.log(target[prop]);
}
});

console.log(proxy.a);  //this is property value 1
console.log(target.a);  //1
proxy.b = 3;        //修改b的值为3
console.log(target);   //{a:1,b:3}
```

可以直接修改数组

```
let arr = [
    {name:'小明',age:19},
    {name:'小王',age:29},
    {name:'小化',age:13},
    {name:'小李',age:45},
];
let persons = new Proxy(arr,{
    get(arr,prop){
        return arr[prop];
    },
    set(arr,prop,value){
        arr[prop] = value;
    }
});
console.log(persons[2]);  //{name:'小化',age:13}
persons[1] = {name:'小转换',age:13};
console.log(persons,arr); //是修改后的结果，是一样的
```

直接修改函数

```
let fn = function(){
    console.log('i am a function');
}
fn.a = 123;
let newFn = new Proxy(fn,{
    get(fn,prop){
        return fn[prop] + 'this is a Proxy return';
    }
})
console.log(newFn.a)
```

7. 模板编译原理

Vue中的模板template无法被浏览器解析并渲染，因为这不属于浏览器的标准，不是正确的HTML语法，所以需要将template转化成一个JavaScript函数，这样浏览器就可以执行这一个函数并渲染出对应的HTML元素，就可以让视图跑起来了，这一个转化过程，就称为 **模板编译**。模板编译又分为三个阶段，解析parse，优化optimize，生成generate，最终生成可执行函数render。

- **解析阶段**：使用大量的正则表达式对template字符串进行解析，将标签、指令、属性等转化为抽象语法树AST。
- **优化阶段**：遍历AST，找到其中的一些静态节点并进行标记，方便在页面重渲染的时候进行diff比较时，直接跳过这一些静态节点，优化runtime的性能。
- **生成阶段**：将最终的AST转化为render函数字符串。

三、Vue区别篇

1. MVVM、MVC、MVP的区别

> 阮一峰对这三个模式的解读

这三种都是常见的软件架构设计模式，主要通过分离关注点的方式来组织代码结构、优化开发效率。

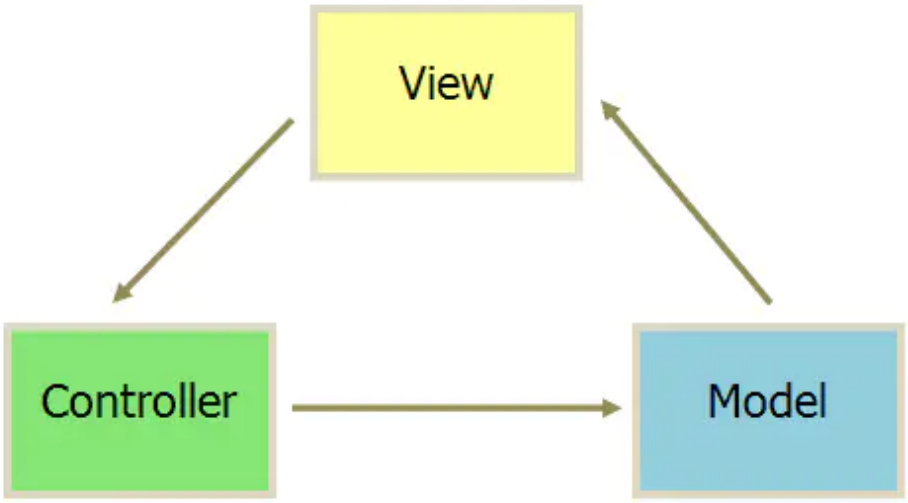
在开发单页面应用时，往往一个路由页面对应一个脚本文件，所有的页面逻辑都在一个脚本文件里。页面的渲染、数据的获取，对用户事件的响应所有的应用逻辑都混合在一起，这样在开发简单项目时，可能看不出什么问题，如果项目变得复杂，那么整个文件就会变得冗长、混乱，这样对项目开发和后期的项目维护是非常不利的。

MVC

`MVC` 通过分离 `Model`、`View`、`Controller` 的方式来组织代码结构。

其中 `View` 负责页面的显示，`Model` 负责存储页面的业务数据以及对应的数据操作。并且 `View` 和 `Model` 应用了观察者模式，当 `Model` 层发生改变的时候它会通知有关 `View` 层更新页面。

`Controller` 层是 `View` 和 `Model` 层的纽带，它**主要负责用户与应用的响应操作**，当用户与页面产生交互的时候，`Controller` 中的事件触发器就开始工作了，通过调用 `Model` 层，来完成对 `Model` 的修改，然后 `Model` 层再去通知 `View` 层更新。



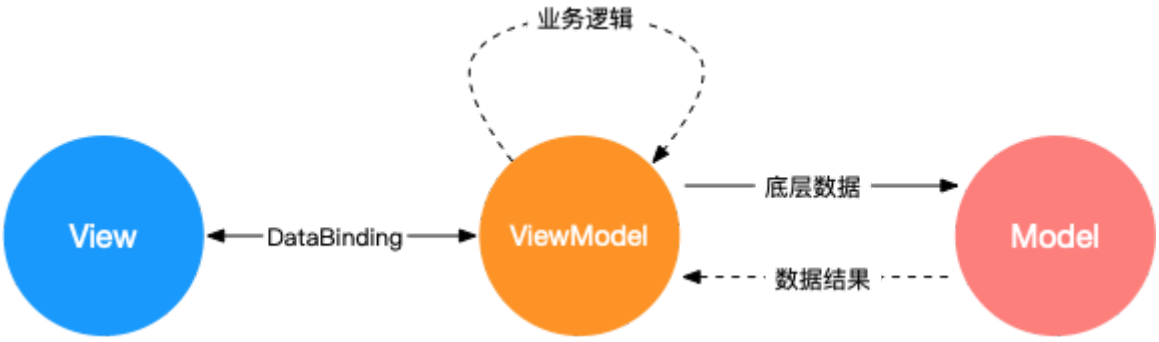
MVVM（Vue使用）

> MVVM分为 Model、View、ViewModel。

- `Model`代表数据模型，数据和业务逻辑都在`Model`层中定义
- `View`代表UI视图，负责数据的展示；
- `ViewModel`负责监听`Model`中数据的改变并控制视图的更新，处理用户交互操作（业务逻辑层）

Model和View并无直接关联，而是通过ViewModel来进行联系。**view和ViewModel之间有着双向数据绑定的联系。View的变动，自动反映在 ViewModel，反之亦然。**

这种模式实现了Model和View的数据的同步更新，因此开发者只需要专注于数据的维护操作即可，而不需要自己操作DOM。

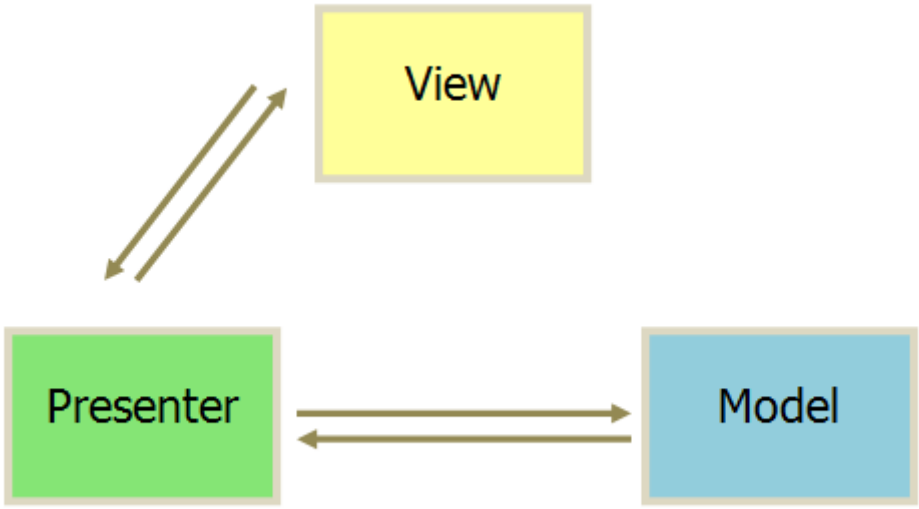


MVP

> 即 Model、View、Presenter。

MVP和MVC唯一不同的在于 Presenter和Controller。在MVC中，View层和Model层会耦合在一起，当项目逻辑变得复杂时，可能会造成代码的混乱，并且可能会对代码的复用性造成一些问题。

MVP通过使用Presenter来实现对View层和Model层的解耦。MVC中的Controller只知道Model的接口，因此它没有办法控制View层的更新，MVP中，VIew层的接口暴露给了Presenter，因此可以在Presenter中将Model的变化和View的变化绑定在一起以此来实现View和Model的同步更新。这样就实现了View和Model层的解耦，Presenter还包括了其他的响应逻辑。



2. v-if 和 v-show的区别

- **手段**：v-if是动态的向DOM树添加或者删除DOM元素；v-show是通过设置元素的display样式属性控制显隐；
- **编译过程**：v-if切换有一个局部编译/卸载的过程，切换过程中合适地销毁和重建内部的事件监听和子组件；v-show只是简单的基于css切换。
- **编译条件**：v-if是惰性的，如果初始条件为假，则什么也不做；只有在条件第一次变为真时才开始局部编译；v-show是在任何条件下，无论条件是否为真，都被编译，然后被缓存，而且DOM元素保留。
- **性能消耗**：v-if有更高的切换消耗；v-show有更高的初始渲染消耗
- **使用场景**：v-if适合运营条件不大可能改变；v-show适合频繁切换。

3. Computed和Watch的区别

区别：

- `computed`计算属性：依赖其他属性值，并且computed的值有缓存，只有它依赖的属性值发生改变，下一次获取computed的值才会重新计算computed的值。不支持异步。
- `watch监听器`：更多的是 **观察**的作用，**无缓存性**，类似于某些数据的监听回调，每当监听的数据发生变化时都会执行回调进行后续操作。

使用场景：

- 当需要进行数值计算，并且依赖与其他数据时，应该使用 `computed`，因为可以利用computed的缓存特性，避免每次获取值时都要重新计算。
- 当需要在数据变化时执行异步或开销较大的操作时，应该使用watch，使用watch选项允许执行异步操作（访问一个API），限制执行该操作的频率，并在得到最终结果前，设置中间状态。这都是计算属性无法做到的。

4. created 和 mounted 的区别

- **created**：在模板渲染成html前调用，即通常初始化某些属性值，然后再渲染成视图。
- **mounted**：在模板渲染成html后调用，通常是初始化页面完成后，再对html的DOM节点进行一些需要的操作。

5. Vue-router跳转和location.href有什么区别

- 使用 `location.href=/url` 来跳转，简单方便，但是刷新了页面；
- 使用 `history.pushState(/url)`，无页面刷新，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，因为vue-router就是用了 `history.pushState()`，尤其是在history模式下。

6. params 和 query的区别

> Vue Router 的params和query传参的使用和区别（详尽）

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.param.name`

url地址显示：query更加类似于ajax中get传参，params则类似于post，说的再简单一些，前者在浏览器地址栏中显示参数，后者则不显示。

注意：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。

> `$router`：是路由操作对象，只写对象
>
> `$route`：路由信息对象，只读对象

```
//query传参，使用path跳转
this.$router.push({
    path:'second',
    query: {
        queryId:'20180822',
        queryName: 'query'
    }
})

//query传参接收
this.queryName = this.$route.query.queryName;
this.queryId = this.$route.query.queryId;

//路由
{
path: '/second',
name: 'second',
component: () => import('@/view/second')
}
////////////////////////////////////////
//params传参 使用name
this.$router.push({
  name:'second',
  params: {
    id:'20180822',
     name: 'query'
  }
})

//params接收参数
this.id = this.$route.params.id ;
this.name = this.$route.params.name ;

//路由
{
path: '/second/:id/:name',
```

```
name: 'second',
component: () => import('@/view/second')
}
```

7. Vuex 和localStorage 的区别

（1）最重要的区别

- Vuex存储在内存中
- localStorage 则以文件的方式存储在本地，只能存储字符串类型的数据，存储对象需要JSON的stringify和parse方法进行处理。读取内存比读取硬盘速度更快。

（2）应用场景

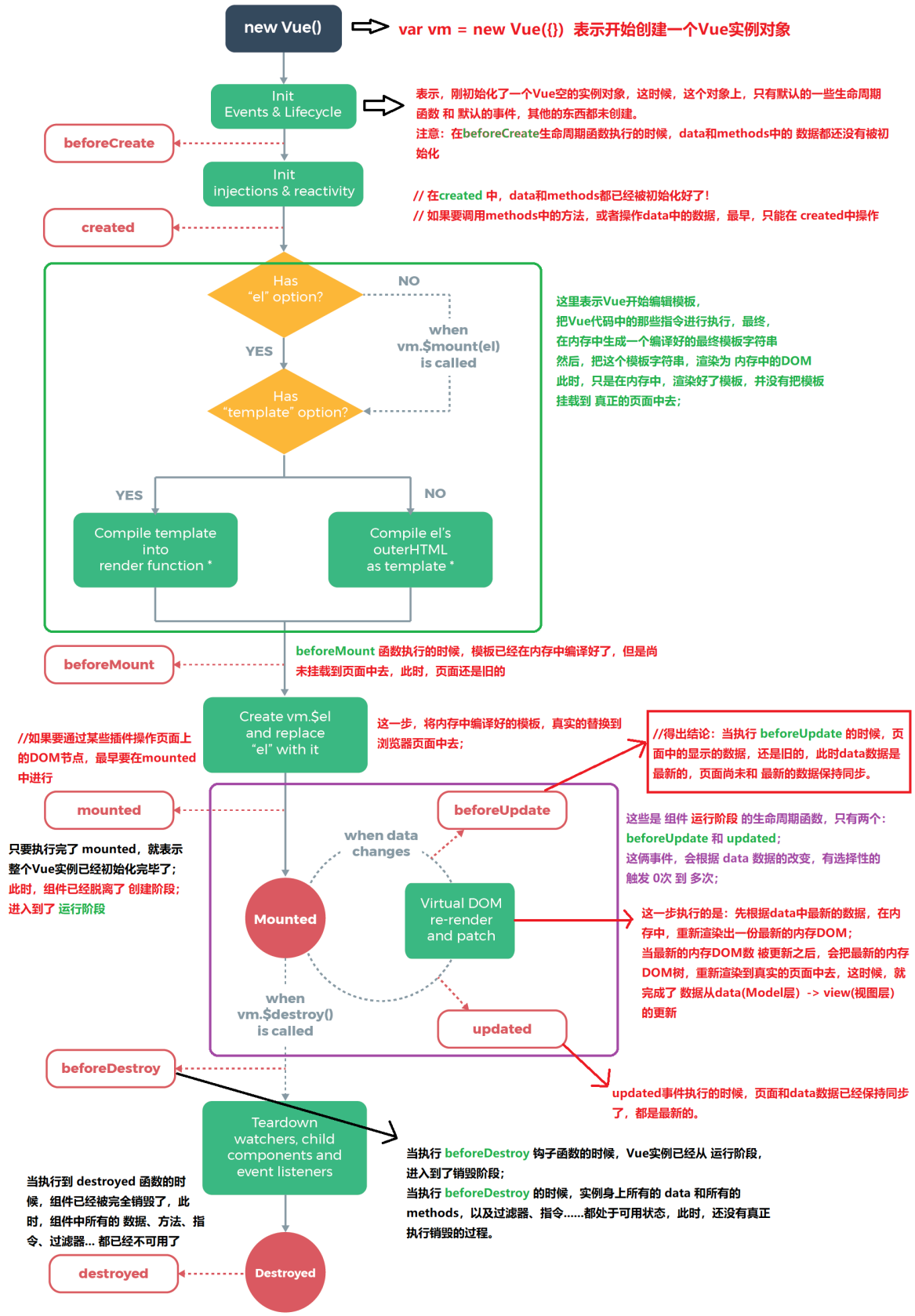
- Vuex是一个专为vuejs应用程序开发的状态管理模式。它采用集中式存储管理应用的所有组件的状态，并以相应的规则保证以一种可预测的方式发生变化。Vuex用于组件之间的传值。
- localStorage是本地存储，是将数据存储到浏览器的方法，一般是在跨页面传递数据时使用、
- Vuex能做到数据的响应式，localStorage不能

（3）永久性

刷新页面时Vuex存储的值会丢失，localStorage不会

注意：对于不变的数据确实可以用localStorage代替Vuex，但是当两个组件共有一个数据源（对象或数组）时，如果其中一个组件改变了该数据源，希望另一个组件响应该变化时，localStorage无法做到，原因就是区别1.

四、Vue生命周期



1. 说一下Vue的生命周期

Vue实例有一个完整的生命周期，也就是从开始创建、初始化数据、编译模板、挂载DOM -> 渲染、更新 -> 渲染、卸载等以系列过程，称这为vue的生命周期。

> 注：在这个过程中也会运行一些叫做**生命周期钩子**的函数，这给用户在不同阶段添加自己代码的机会。以下这些就是钩子函数：

1. **beforeCreate（创建前）**：数据观测和初始化事件还没开始，此时data的响应式追踪、event/watcher都还没有被设置，也就是说不能访问到data、computed、watch、methods上的方法和数据。
2. **created（创建后）**：实例创建完成，实例上配置的options包括 data、methods、computed、watch等都配置完成，但是此时渲染节点还未挂载到DOM，所以不能访问到`$el`属性。
3. **beforeMounted（挂载前）**：在挂载开始之前被调用，相关的render函数首次被调用。实例已完成以下的配置：编译模板、把data里面的数据和模板生成HTML。此时还没有挂载到页面上。
4. **mounted（挂载后）**：在el 被新创建的 vm.$el替换，并挂载到实例上去之后调用。实例已经完成以下的配置：用上面编译好的HTML内容替换el属性指向的DOM对象。完成模板中的html渲染到html页面中。此过程中进行ajax交互。
5. **beforeUpdate（更新前）**：响应式数据更新时调用，此时虽然响应式数据更新了，但是对应的真实DOM还没有被渲染。
6. **updated（更新后）**：由于数据更改导致的虚拟DOM重新渲染和打补丁之后调用。此时DOM已经根据响应式数据的变化更新了。调用时，组件DOM已经更新，所以可以执行依赖于DOM的操作。然而在大多数情况下，应该避免在此期间更改状态，因为这可能会导致更新无限循环。该钩子函数在服务器端渲染期间不被调用。
7. **beforeDestroy（销毁前）**：实例销毁之前调用。这一步，实例仍然完全可以使用，`this`仍能获取到实例。
8. **destroy（销毁后）**：实例销毁后调用，调用后，Vue实例指示的所有东西都会解绑定，所有的事件监听都会被移除，所有的子实例也会被销毁。该钩子函数在服务端渲染期间不被调用。

2. Vue子组件和父组件执行顺序

加载渲染过程：

1. 父组件 beforeCreate
2. 父组件 created
3. 父组件 beforeMount
4. 子组件 beforeCreate
5. 子组件 created
6. 子组件 beforeMount
7. 子组件 mounted
8. 父组件 mounted

更新过程：

1. 父组件 beforeUpdate
2. 子组件 beforeUpdate
3. 子组件 updated
4. 父组件 updated

销毁过程：

- 父组件 beforeDestroy
- 子组件 beforeDestroy
- 子组件 destroyed
- 父组件 destroyed

3. 一般在哪个生命周期请求异步数据

我们可以在钩子函数 created、beforeMounted、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。

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的数据，减少页面加载时间，用户体验更好。
- SSR不支持 beforeMounted、mounted钩子函数，放在created中有助于一致性。

4. keep-alive中的生命周期有哪些

> 如果需要在组件切换的时候，保存一些组件的状态防止多次渲染，就可以使用keep-alive组件包裹需要保存的组件。

keep-alive是Vue提供的一个内置组件，用来对组件进行缓存------在组件切换过程中将状态保留在内存中，防止重复污染DOM。

如果一个组件包裹了keep-alive，那么它会多出两个生命周期：deactivated、activated。同时，beforeDestroy 和 destroy就不会再被触发了，因为组件不会被真正的销毁。

当组件被换掉时，会被缓存到内存中，触发deactivated 生命周期；当组件被切换回来时，再去缓存里找这个组件、触发activated钩子函数。

五、组件通信

1. props / $emit

父组件通过 `props` 向子组件传递数据，子组件通过 `$emit` 和父组件通信。

（1）父组件向子组件传值

- `props` 只能是父组件向子组件传值，`props` 使得父子组件之间形成了一个单向下行绑定。子组件的数据会随着父组件不断更新。
- props可以接收多个数据，可以是任意类型的数据。

```
//父组件
<template>
    <div id="father">
        <son :msg="msgData" :fn="myFunction"></son>
    </div>
</template>
<script>
import son from "./son.vue";
export default {
    name: father,
    data() {
        msgData: "父组件数据";
    },
    methods: {
        myFunction() {
            console.log("vue");
        }
    },
    components: {
        son
    }
};
</script>
```

```
// 子组件
<template>
    <div id="son">
        <p>{{msg}}</p>
        <button @click="fn">按钮</button>
    </div>
</template>
<script>
export default {
    name: "son",
    props: ["msg", "fn"]
};
</script>
```

（2）子组件向父组件传值

`$emit`绑定一个自定义事件，当这个事件被执行的时候就会将参数传递给父组件，而父组件通过`v-on`监听并接收参数。

```
// 父组件
<template>
  <div class="section">
    <com-article :articles="articleList" @onEmitIndex="onEmitIndex"></com-
article>
    <p>{{currentIndex}}</p>
  </div>
</template>

<script>
import comArticle from './test/article.vue'
export default {
  name: 'comArticle',
  components: { comArticle },
  data() {
    return {
      currentIndex: -1,
      articleList: ['红楼梦', '西游记', '三国演义']
    }
  },
  methods: {
    onEmitIndex(idx) {
      this.currentIndex = idx
    }
  }
}
</script>
```

```
//子组件
<template>
  <div>
    <div v-for="(item, index) in articles" :key="index"
@click="emitIndex(index)">{{item}}</div>
  </div>
</template>

<script>
```

```
export default {
  props: ['articles'],
  methods: {
    emitIndex(index) {
      this.$emit('onEmitIndex', index) // 触发父组件的方法，并传递参数index
    }
  }
}
</script>
```

2. eventBus事件总线（$emit/$on）

`eventBus`事件总线适用于 **父子组件、非父子组件**等之间的通信。使用步骤如下：

（1）创建 事件中心 管理 组件之间的通信：这是一个js文件

```
// event-bus.js
import Vue from 'vue'
export const EventBus = new Vue()
```

（2）发送事件

假设有两个兄弟组件 `firstCom` 和 `secondCom`

```
<template>
  <div>
    <first-com></first-com>
    <second-com></second-com>
  </div>
</template>

<script>
import firstCom from './firstCom.vue'
import secondCom from './secondCom.vue'
export default {
  components: { firstCom, secondCom }
}
</script>
```

在`firstCom`组件中发送事件：

```
<template>
  <div>
    <button @click="add">加法</button>
  </div>
</template>

<script>
import {EventBus} from './event-bus.js' // 引入事件中心

export default {
  data(){
    return{
      num:0
    }
  },
```

```
  methods:{
    add(){
      EventBus.$emit('addition', {
        num:this.num++
      })
    }
  }
}
</script>
```

（3）接收事件

在 `secondCom` 组件中接收事件：

```
<template>
  <div>求和: {{count}}</div>
</template>

<script>
import { EventBus } from './event-bus.js'
export default {
  data() {
    return {
      count: 0
    }
  },
  mounted() {
    EventBus.$on('addition', param => {
      this.count = this.count + param.num;
    })
  }
}
</script>
```

在上述代码中，这就相当于将 `num` 值存储在了事件总线中，在其他组件中可以直接访问。事件总线就相当于一个桥梁，不用组件通过它来通信。

虽然看起来比较简单，但是这种方法也有不便之处，如果项目过大，使用这种方式进行通信，后期维护起来会很困难。

3. 注入依赖（provide/inject）

这种方式就是Vue中的 **注入依赖**，该方法用于 **父子组件之间的通信。**这里父子也可以是祖孙组件，在层数很深的请况下，可以使用这种方法来进行传值。就不用一层一层的传递了。

`provide / inject` 是Vue提供的两个钩子，和 `data`、`methods` 是同级的。**并且 `provide` 的书写形式和 `data` 一样。**

- `provide` 钩子用来发送数据或方法
- `inject` 钩子用来接收数据或方法

在父组件中：

```
provide(){
    return{
        num:this.num
    };
}
```

在子组件中：

```
inject:['num']
```

还可以这样写，这样就可以访问父组件中的所有属性：

```
provide() {
 return {
    app: this
  };
}
data() {
 return {
    num: 1
  };
}

inject: ['app']
console.log(this.app.num)
```

> 注：依赖注入所提供的属性是 **非响应式**的。

4. ref / $refs

这也是实现 **父子组件** 之间的通信。

`ref`：这个属性用在子组件上，它的引用就指向了子组件的实例。可以**通过实例来访问组件的数据和方法。**

在子组件中：

```
export default {
    data(){
        return {
            name:'javascript'
        }
    },
    methods:{
        sayHello(){
            console.log('hello')
        }
    }
}
```

在父组件中：

```
<template>
  <child ref="child"></component-a>
</template>
<script>
import child from './child.vue'
export default {
    components:{child},
    mounted(){
        console.log(this.$ref.child.name);  //javascript
        this.$refs.child.sayHello();  //hello
    }
}
</script>
```

5. $parent / $children

- 使用 `$parent` 可以让组件访问父组件的实例（访问的是上一级父组件的属性和方法）
- 使用 `$children` 可以让组件访问子组件的实例。但是， `$children` 并不能保证顺序，并且访问的数据也**不是响应式的。**

在子组件中：

```
<template>
  <div>
    <span>{{message}}</span>
    <p>获取父组件的值为:  {{parentVal}}</p>
  </div>
</template>

<script>
export default {
  data() {
    return {
      message: 'Vue'
    }
  },
  computed:{
    parentVal(){
      return this.$parent.msg;
    }
  }
}
</script>
```

在父组件中：

```
// 父组件中
<template>
  <div class="hello_world">
    <div>{{msg}}</div>
    <child></child>
    <button @click="change">点击改变子组件值</button>
  </div>
</template>
```

```
<script>
import child from './child.vue'
export default {
  components: { child },
  data() {
    return {
      msg: 'Welcome'
    }
  },
  methods: {
    change() {
      // 获取到子组件
      this.$children[0].message = 'JavaScript'
    }
  }
}
</script>
```

在上面代码中，子组件获取到了父组件的 `msg` 值，父子间改变了子组件的 `message` 的值

需要注意:

- 通过 `$parent` 访问到的是上一级父组件的实例，可以使用 `$root` 来访问根组件的实例
- 在组件中使用 `$children` 拿到的是所有的子组件的实例，它是一个数组，并且是无序的
- 在根组件 `#app` 上拿 `$parent` 得到的是 `new Vue()` 的实例，在实例上再拿 `$parent` 得到的是 `undefined`，而在最底层的子组件那 `$children` 是个空数组
- $children 的值是数组，而 $parent是个对象。

6. $attrs / $listeners

用于爷孙之间这种隔代传递数据的场景。

- `$attrs`：继承所有的父组件属性（除了prop传递的属性、class和style），一般用在子组件的子元素上。
- `$listeners`：该属性是一个对象，里面包含了作用在这个组件上的所有监听器，可以配合 `v-on=$listeners` 将所有的事件监听器指向这个组件的某个特定的子元素（相当于子组件内继承父组件的事件）

假设有 A->B->C (父子孙)

A组件（APP.vue）

```
<template>
    <div id="app">
        //此处监听了两个事件，可以在B组件或者C组件中直接触发
        <child1 :p-child1="child1" :p-child2="child2" @test1="onTest1"
@test2="onTest2"></child1>
    </div>
</template>
<script>
import Child1 from './Child1.vue';
export default {
    components: { Child1 },
    methods: {
        onTest1() {
            console.log('test1 running');
        },
```

```
        onTest2() {
            console.log('test2 running');
        }
    }
};
</script>
```

B组件（Child1.vue）

```
<template>
    <div class="child-1">
        <p>props: {{pChild1}}</p>
        <p>$attrs: {{$attrs}}</p>
        <child2 v-bind="$attrs" v-on="$listeners"></child2>
    </div>
</template>
<script>
import Child2 from './Child2.vue';
export default {
    props: ['pChild1'],
    components: { Child2 },
    inheritAttrs: false,
    mounted() {
        this.$emit('test1'); // 触发APP.vue中的test1方法
    }
};
</script>
```

C组件（child2.vue）

```
<template>
    <div class="child-2">
        <p>props: {{pChild2}}</p>
        <p>$attrs: {{$attrs}}</p>
    </div>
</template>
<script>
export default {
    props: ['pChild2'],
    inheritAttrs: false,
    mounted() {
        this.$emit('test2');// 触发APP.vue中的test2方法
    }
};
</script>
```

在上述代码中：

- C组件中能直接出发test的原因在于B组件调用C组件时 使用v-on 绑定了 `$listeners` 属性。
- 在B组件中通过 v-bind 绑定 `$attrs` 属性，C组件可以直接获取到A组件中传递下来的prop（除了B组件中props声明的）

总结

（1）父子组件之间的通信

- 子组件通过props属性来接收父组件的数据，然后（在父组件中）父组件在子组件上注册监听事件，子组件通过$emit触发事件来向父组件发送数据。
- 通过ref属性给子组件设置一个名字。父组件通过`$refs.组件名`来获取子组件，子组件通过`$parent`获得父组件，这样也可以实现通信。
- 使用`provide/inject`，在父组件中通过provide提供变量，在子组件中通过 inject俩将变量注入到组件中，不论子组件有多深，只要调用了inject那么就可以注入provide中的数据。

（2）兄弟组件间通信

- 使用eventBus的方法，它的本质是通过创建一个空的Vue实例来作为消息传递的对象，通过的组件引入这个实例，通信的组件通过在这个实例上监听和触发事件，来实现消息的传递。
- 通过 $parent|$ref来获取到兄弟组件，也可以进行通信。

（3）任意组件之间

- 使用eventBus，其实就是创建一个事件中心，相当于中转站，可以用它来传递事件和接收事件。

> 如果业务逻辑复杂，很多组件之间需要同时处理一些公共的数据，这个时候采用上面这些方法可能不利于项目的维护。这个时候可以使用vuex，vuex的思想就是将一些公共数据抽离出来，将它作为全局的变量来管理，然后其他组件就可以对这个公共数据进行读写操作，这样就达到了解耦的目的。

六、路由

1. Vue-Router的懒加载如何实现

> 所谓的懒加载实际上就是按需加载，当我们需要的时候才给他加载出来。在页面上使用很多import属于非懒加载。

非懒加载：

```
import List from '@/components/list.vue'
const router = new VueRouter({
    routes:[
        {path:'/list',component:List}
    ]
})
```

（1）方案①（常用）：使用箭头函数 + import动态加载

```
const List = () => import('@/components/list.vue')
const router = new VueRouter({
    {path:'/list',component:List}
})
```

（2）方案②：使用箭头函数 + require动态加载

```
const router = new Vue-Router({
    routes:[
        {
            path:'/list',
            component:resolve => require(['@/components/list'],resolve)
        }
    ]
})
```

（3）方案③：使用webpack的require.ensure技术，也可以实现按需加载。在这种情况下，多个路由指定相同的chunkName，会合并打包成一个js文件。

```
//r就是resolve
const List = r => require.ensure([],()=>r(require('@/components/list')),'list');
//路由也是正常的写法 这种方法是官方推荐写的 按模块划分懒加载
const router = new Router({
    routes:[
    {
        path:'/list',
        component:List,
        name:'list'
    }
]
})
```

2. 路由的hash 和 history模式的区别

> Vue-router 有两种模式：hash模式 和 history模式。默认的路由模式是hash模式。

（1）hash模式

简介：hash模式是开发中的默认的模式，它的url带一个# ，如：http://www.abc.com/#/vue，它的hash值就是 `#/vue`

特点：hash值会出现在URL里面，但是不会出现在HTTP请求中，对后端完全没有影响。所以改变hash值，不会重新加载页面。这种模式的浏览器支持度很好，低版本的IE浏览器也支持这种模式。hash路由被称为是 **前端路由**，已经成为SPA（单页面应用）的标配。

原理：`onhashchange()`事件：

```
window.onhashchange = function(event){
    console.log(event.oldURL,event.newURL);
    let hash = location.hash.slice(1);
}
```

使用onhashchange()事件的好处就是，在页面的hash值发生变化时，无需向后端发起请求，window就可以监听事件的改变，并按照规则加载相应的代码。除此之外，hash值变化对应的url都会被浏览器记录下来，这样浏览器就能实现页面的前进和后退。虽然是没有请求后端服务器，但是页面和hash值和对应的url关联起来了。

（2）history模式

简介：history模式中的url没有 #，它使用的是传统的路由分发模式，即用户在输入一个URL时，服务器会接收这个请求，并解析这个URL，然后做出相应的逻辑处理。

特点：URL长这样：http://www.abc.com/vue/id。相比hash模式更好看。但是，history模式需要后台配置支持。如果后台没有正确配置，访问时会返回404.

API：history api可以分为两大部分，切换历史状态和修改历史状态。

- **修改历史状态**：包括了html5 history interface 中新增的 `pushState()` 和 `replaceState()` 方法，这两个方法应用于浏览器的历史记录栈，提供了对历史记录进行修改的功能。只是当他们进行修改时，虽然修改了URL，但浏览器不会立即向后端发送请求。如果要做到改变url但又不刷新页面的效果，就需要前端用上这两个API。
- **切换历史状态**：包括 `forward()`、`back()`、`go()`，对浏览器进行前进、后退、跳转操作。

如果要切换到history模式，就要进行以下配置（后端也要进行配置）：

```
const router = new VueRouter({
    mode:'history',
    routes:[...]
})
```

（3）两种模式的对比

调用 `history.pushState()` 相比于直接修改 hash，存在以下优势：

- pushState() 设置的新URL可以是与当前url**同源的**任意URL；而hash只可修改#后面的部分，因此只能设置与当前URL同文档的URL；
- pushState()设置的新url可以与当前url**一模一样**，这样也会把记录添加到栈中；而hash设置的新值必须和原来的不一样才会触发动作将记录添加到栈中。
- pushState()通过stateObject参数可以**添加任意类型的数据**到记录中；而hash只可添加短字符串；
- pushstate()可额外设置title属性供后续使用。
- hash模式下，仅hash符号之前的url会被包含在请求中，后端如果没有做到对路由的全覆盖，也不会返回404错误；history模式下，前端的url必须和实际向后端发起请求的url一致，如果没有对应的路由处理，将返回404错误。

> 无论使用哪种模式，本质都是使用的 `history.pushState`，每次pushState后，会在浏览器的浏览记录中添加一个新的记录，但是并 **不会触发页面刷新，也不会请求新的数据。**

3. 如何获取页面的hash变化

（1）监听$route的变化

```
//监听，当路由发生变化的时候执行
watch:{
    $route:{
        handler:function(val,oldVal){
            console.log(val);
        },
        //深度观察监听
        deep:true
    }
},
```

（2）window.location.hash读取#值

window.location.hash 的值可读可写，读取来判断状态是否改变，写入时可以在不重载网页的前提下，添加一条历史记录。

4. $route 和 $router 的区别

- `$route`是“路由信息对象”，包括 path 、params、hash、query、fullPath、matched、name等路由信息参数。
- `$router`是“路由实例”对象，包括了路由的跳转方法，钩子函数等。

5. 如何定义动态路由？如何获取传过来的动态参数?

（1）param方式

- 配置路由格式：`/router/:id`
- 传递的方式：在path后面跟上对应的值
- 传递后形成的路径：`/router/123`

1. 路由定义

```
 //在APP.vue中 方式一
<router-link :to="'/user/'+userId" replace>用户</router-link>

//在index.js中  方式二
{
    path:'/user/:userid',
    component:User
},
```

2. 路由跳转

```
// 方法1：
<router-link :to="{ name: 'users', params: { uname: wade }}">按钮</router-link

// 方法2：
this.$router.push({name:'users',params:{uname:wade}})

// 方法3：
this.$router.push('/user/' + wade)
```

3. 参数获取

通过`$route.params.userid`获取传递的值。

（2）query方式

- 配置路由格式：`/router`，也就是普通配置
- 传递的方式：对象中使用 query 的 key 作为传递方式
- 传递后形成的路径：`/route?id=123`

1）路由定义

```
//方式1：直接在router-link 标签上以对象的形式
<router-link :to="{path:'/profile',query:{name:'why',age:28,height:188}}">档案
</router-link>

// 方式2：写成按钮以点击事件形式
<button @click='profileClick'>我的</button>

profileClick(){
  this.$router.push({
    path: "/profile",
    query: {
```

```
        name: "kobi",
        age: "28",
        height: 198
    }
  });
}
```

2）跳转方法

```
// 方法1：
<router-link :to="{ name: 'users', query: { uname: james }}">按钮</router-link>

// 方法2：
this.$router.push({ name: 'users', query:{ uname:james }})

// 方法3：
<router-link :to="{ path: '/user', query: { uname:james }}">按钮</router-link>

// 方法4：
this.$router.push({ path: '/user', query:{ uname:james }})

// 方法5：
this.$router.push('/user?uname=' + jsmes)
```

3）获取参数

通过 $route.query 获取传递的值

6. Vue-router 导航守卫

详解

- 全局前置/钩子：beforeEach、beforeResolve、afterEach
- 路由独享的守卫：beforeEnter
- 组件内的守卫：beforeRouteEnter、beforeRouteUpdate、beforeRouteLeave

```
const router = new vueRouter({...})
router.beforeEach((to,from,next)=>{
    //...
})
```

- to：即将要进入的目标
- from：当前导航正要离开的路由

7. 对前端路由的理解

> 理解链接：
>
> - 知乎 | 简书

在前端技术早期，一个url对应一个页面，如果要从A页面切换到B页面，那么必然伴随着页面的刷新。这个体验不好（也就是说只有在用户刷新页面的情况下，才可以重新去请求数据）

后来ajax出现了，它允许人们在不刷新页面的情况下发起请求；与之共生的，还有“不刷新页面即可更新页面内容”这种需求。在这样的背景下，出现了**SPA（单页面应用）**。

SPA极大地提升了用户体验，它允许页面在不刷新的情况下更新页面内容，使页面内容切换更加流畅。但是在SPA诞生之初，人们并没有考虑到“定位”这个问题-------在内容切换前后，页面的URL都是一样的，这就带来了两个问题：

- SPA其实不知道当前页面“进展到了哪一步”。可能在下一个站点下经过了反复的“前进”才终于唤出了某一块内容，但是此时只要刷新一下页面，一切就会被清零，必须重复之前的操作、才可以重新对内容进行定位-------SPA并不会“记住”你的操作
- 由于有且仅有一个URL给页面做映射，这对SEO也不太友好，搜索引擎无法收集全面的信息

为了解决这个问题，前端路由出现了。

前端路由可以帮助我们在仅有一个页面的情况下，“记住”用户当前走到了哪一步------为SPA中的各个视图匹配一个唯一标识。这意味着用户前进、后退触发的新内容，都会映射到不同的URL上去。此时即便他刷新页面，因为当前的URL可以标识出它所在的位置，因此内容也不会丢失。

前端路由可以提供这样的解决思路：

- 拦截用户的刷新操作，避免服务端盲目响应、返回不符合预期的资源内容。把刷新这个动作完全放到前端逻辑里消化掉。
- 感知URL的变化。这里不是说要改造URL、凭空制造出N个URL来，而是说URL还是那个URL，只不过我们可以给它做一些微小的处理------这些处理并不会影响URL本身的性质，不会影响服务器对它的识别，只有我们前端感知得到。一旦我们感知到了，就会根据这些变化、用JS去给他生成不同的内容。

前端路由的实现：使用hash模式和history模式

七、Vuex

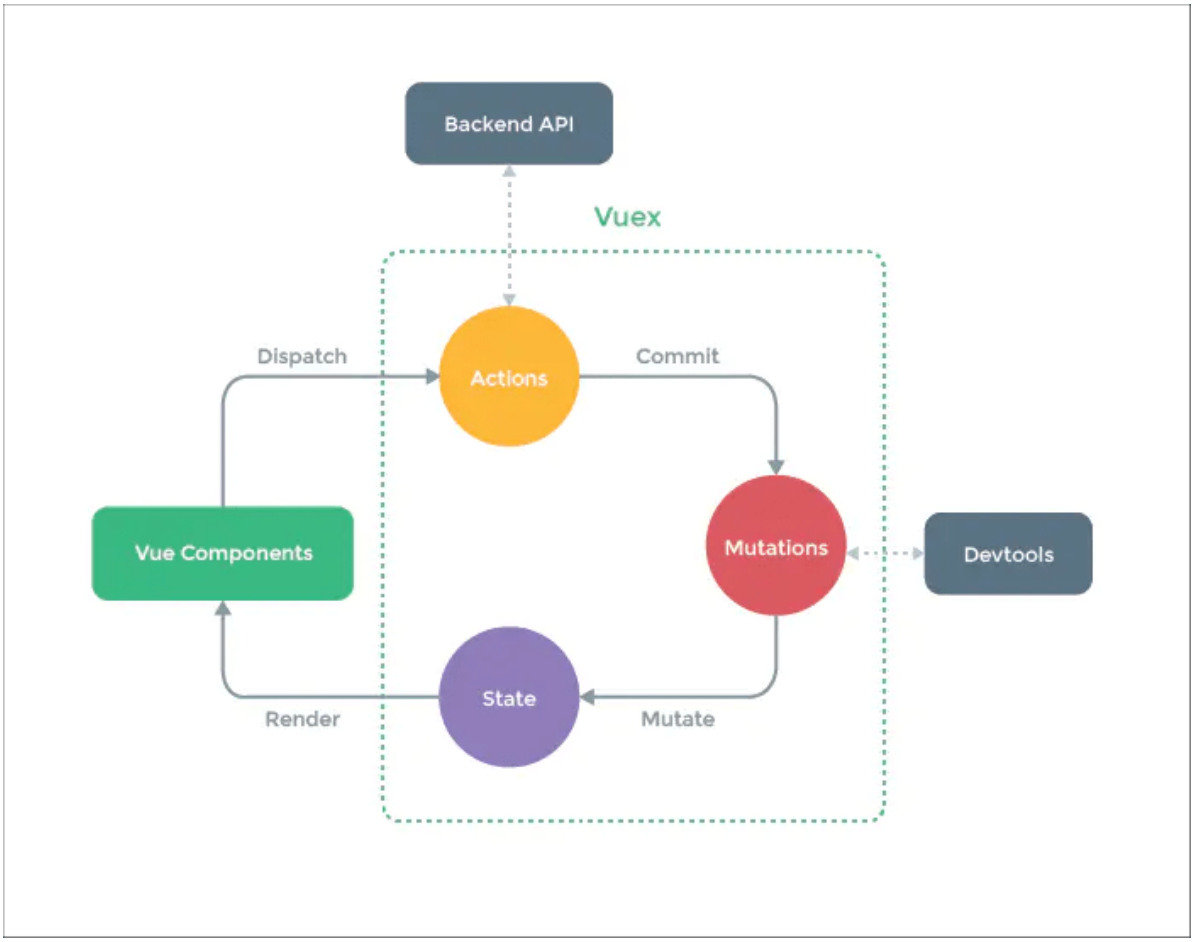
1. vuex的原理

我的博客解释

官网解释

vuex是一个专为vue.js应用程序开发的状态管理模式。，每一个VueX应用的核心就是store（仓库）。“store”基本上就是一个容器，它包含着你的应用中大部分的状态（state）

- vuex的状态存储是响应式的。当vue组件从store中读取状态的时候，若store中的状态发生变化，那么相应的组件也会相应地得到高效更新。
- 改变store中的状态的唯一途径就是**显示地提交（commit）mutation**。这样就可以方便地跟踪每一个状态的变化。



Vuex为Vue Components建立起了一个完整的生态圈，包括开发中的API调用一环。

（1）核心流程中的主要功能：

- Vue Components 是Vue组件，组件会触发（dispatch）一些事件或动作，也就是图中的Actions。
- 在组件中发出的动作，肯定是想获取或者改变数据的，但是在vuex中，数据是集中管理的，不能直接去更改数据，所以会把这个动作提交（Commit）到Mutations中；
- 然后Mutations就去改变（Mutate）State 中的数据
- 当State 中的数据被改变之后，就会重新渲染（render)到Vue Components中去，组件展示更新后的数据，完成一个流程。

（2）各模块在核心流程中的主要功能：

- `Vue Components`：Vue组件。HTML页面上，负责接收用户操作等交互行为，执行dispatch方法触发对应action进行回应。
- `dispatch`：操作行为触发方法，是唯一能执行action的方法。（调度、派遣、分配、触发）
- `actions`：操作行为处理模块。负责处理Vue Components接收到的所有交互行为。包括 **同步/异步**操作，支持多个同名方法，按照注册的顺序依次触发。向后台API请求的操作就在这个模块中进行，包括触发其他action 以及 mutation 的操作。该模块提供了Promise的封装，以支持action 的链式触发。
- `commit`：改变状态提交操作方法。对mutation进行提交，是唯一能执行mutation的方法。
- `mutations`：状态改变操作方法。是Vuex修改state的唯一推荐方法，其他修改方式在严格模式下将会报错。该方法**只能进行同步操作**，且方法名只能全局唯一。操作之中会有一些hook暴露出来，以进行state的监控等。（突变）
- `state`：页面状态管理容器对象。集中存储Vuecomponents中data对象的零散数据，全局唯一，以进行统一的状态管理。页面显示所需的数据从该对象中进行读取，利用Vue的细粒度数据响应机制来进行高效的状态更新。
- `getters`：state对象读取方法。图中没有单独列出该模块，应该被包含在了render中，VueComponents 通过该方法读取全局state对象。（相当于计算属性）

```
import Vue from 'vue'
import Vuex from 'vuex'
//1.安装插件
Vue.use(Vuex)
//2. 创建对象，里面会放一下固定的对象
const store = new Vuex.Store({
    state:{   //基本数据
        counter:1000
    },
    //提交更改数据的方法，同步
    mutations:{

    },
    actions:{}, //像一个装饰器，包裹mutation，使之可以异步
    getters:{}, //从基本数据派生出来的数据（类似于计算属性）
    modules:{}  // 模块化Vuex
})
//3.导出store对象
export default store
```

例子:

```
const store = new Vuex.Store({
    state: {
        count: 0
    },

    getters: {
        countPlus: state => {
            return state.count + 1
        }
    },

    mutations: {
        increment: (state, payload) => {
            state.count += payload
        }
    }
})
new Vue({
    el: '.app',
    store,
    computed: {
        count: function() {
            return this.$store.state.count
        }
    },
    methods: {
        increment: function() {
            this.$store.commit('increment', 10)
        }
    },
    template: `
        <div>
            {{ count }}
            <button @click='increment'>点我</button>
```

```
        </div>
    `
})
```

总结:

Vuex实现了一个单向数据流，在全局拥有一个State存放数据，当组件要更改State中的数据时，必须通过Mutation 提交修改信息，Mutation同时提供了订阅者模式供外部插件调用获取 State数据的更新。而当所有异步操作（常见与调用后端接口异步获取更新数据）或批量的同步操作需要走Action，但Action也是无法直接修改State的，还是需要通过Mutation 来修改State的数据，最后，根据State的变化，渲染到视图上。

2. Vuex中action 和 mutation 的区别

`mutation`中的操作是一系列的**同步**函数，用于修改state中的变量的状态。当使用Vuex时需要通过commit来提交需要操作的内容。mutation非常类似于事件：每个mutation都有一个字符串的 事件类型（type）和一个 回调函数（handler）。这个回调函数就是实际进行状态更改的地方，并且它会接收state作为第一个参数：

```
const store = new Vuex.Store({
    state:{
        count:1
    },
    mutations:{
        increment(state){
            state.count++  //变更状态
        }
    }
})
```

当触发一个类型为increment的mutation时，需要调用此函数：

```
store.commit('increment')
```

而Action类似于mutation，不同点在于：

- Action 可以包含任意异步操作
- Action提交的是mutation，而不是直接变更状态

```
const store = new Vuex.Store({
    state:{
        conut:0
    },
    mutations:{
        increment(state){
            state.count++
        }
    },
    actions:{
        increment(context){
            context.commit('increment')
        }
    }
})
```

Action 函数接受一个与store实例具有相同方法和属性的context对象，因此可以调用 context.commit 提交一个 mutation，或者通过 context.state 和 context.getters来获取 state 和getters。

所以，两者的不同点如下：

- Mutation 专注于修改state，理论上是修改State的唯一途径；Action业务代码、异步请求。
- Mutation：必须同步执行；Action：可以异步，但不能直接操作State。
- 在视图更新时，先触发Actions，Actions再触发mutation
- mutation的参数是state，它包含store中的数据；Action的参数是context，它是state的父级，包含state、getters

3. 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，这样可以**方便地跟踪每一个状态的变化**，从而能够实现一些工具帮助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。**（调试困难）

八、Vue3.0

1. Vue3.0 有什么更新

（1）监测机制的变化

- 3.0将带来基于proxy的observer实现，提供全语言覆盖的反应性跟踪
- 消除了Vue2 中基于object.defineProperty的实现所存在的很多限制。

（2）只能监测属性，不能检测对象

- 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、WeakSet 和 WeakMap

（3）模板

- 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

（4）对象式的组件声明方式

- vue2 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做，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

（5）其他方面的更改

- 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其他部分渲染组件内容）组件，针对一些特殊的场景做了处理。
- 基于tree shaking优化，提供了更对的内置功能。

九、虚拟DOM

前言

虚拟DOM，Virtual DOM，这个概念对我们而言其实是不陌生的，在目前主流的三大框架中，都有涉及到使用虚拟DOM，接下来看看在Vue中的虚拟DOM的存在和实现。

1. 什么是虚拟DOM?

首先，虚拟DOM的本质还是JS对象。通常把组合成一个DOM节点的必要东西通过Js对象表示出来，那么这个JS对象就可以用来描述这个DOM节点，我们把这个Js对象就称为是这个真是DOM节点的虚拟DOM节点。

2. 为什么要有虚拟DOM?

Vue是数据驱动视图，也就是说，数据发生变化，视图也就要随之发生更新，在更新的时候难免还是要操作DOM的，但是如果操作真实DOM还是非常复杂的，在每一个真实的DOM节点下的属性是非常多的，直接操作真实DOM是非常消耗性能的一件事，所以虚拟DOM应运而生。

来看一下他们重排重绘的性能消耗：

- 真实DOM：生成HTML字符串+重建所有的DOM元素
- 虚拟DOM：生成VNode + DOMDiff + 必要的dom更新

3. DOM-Diff算法

Vnode最大的用途就是在数据变化前后生成真实DOM对应的虚拟DOM节点，然后就可以对比新旧两份VNode，找出差异所在，然后更新有差异的DOM节点，最终达到以最少操作真实DOM更新视图的目的。

对比新旧两份VNode并找出差异的过程就是所谓的DOM-Diff过程。

patch

在 Vue 中，把 `DOM-Diff` 过程叫做 `Patch` 过程。patch意为 “补丁”，即指对旧的 `VNode` 修补，打补丁从而得到新的 `VNode`。

核心思想：

- 所谓旧的 `VNode`（即 `oldNode`）就是数据变化之前视图所对应的虚拟DOM节点，而新的 `VNode` 是数据变化之后将要渲染的新的视图所对应的虚拟 `DOM` 节点。
- 所以我们要以生成的新的 `VNode` 为基准，对比旧的 `oldNode`。
- 如果新的 `VNode` 上有的节点而旧的 `oldNode` 上没有，那么就在旧的 `oldNode` 上加上去；（**添加**）
- 如果新的 `VNode` 上没有的节点而旧的 `oldNode` 上有，那么就在旧的 `oldNode` 上去掉（**删除**）
- 如果某些节点在新旧上都有，那么就以新的 `VNode` 为准，更新旧的 `oldNode`，从而让新旧节点相同（**覆盖**）

以新的VNode为基准，改造旧的oldNode使之成为跟新的VNode一样，这就是patch过程要干的事。

4. Vue中key的作用

vue中key值的作用可以分为两种情况来考虑：

- 第一种情况是 v-if 中使用key。由于Vue会尽可能高效的渲染元素，通常会复用已有元素而不是从头开始渲染。因此当使用 `v-if` 来实现元素切换时，如果切换前后含有相同类型的元素，那么这个元

素就会被复用。如果是相同的input元素，那么切换前后用户的输入不会被清除，这样是不符合需求的。因此可以通过使用key来唯一的标识一个元素，这个情况下，使用key的元素不会被复用。**这个时候key的作用是用来标识一个独立的元素。**

- 第二种情况是 `v-for` 中使用key。用 v-for 更新已渲染过的元素列表时，它默认使用“就地复用”的策略。如果数据项的顺序发生了改变，Vue不会移动DOM元素来匹配数据项的顺序，而是简单复用此处的每个元素。因此通过为每个列表提高一个key值，来以便Vue跟踪元素的身份，从而高效的实现复用。**这个时候key的作用是为了高效的更新渲染虚拟DOM。**

key是VNode的唯一id，依靠key，我们的diff操作可以更加准确，更加快速。diff算法的过程中，先会进行新旧节点的首尾交叉对比，当无法匹配的时候会用新节点的key和旧节点的key进行对比，从而找到相应的旧节点。

为什么不建议index作为key?

使用index作为key和没写基本上没有去区别，因为不管数组的顺序怎么颠倒，index都是0,1,2...这样排列，导致Vue会复用错误的旧子节点，做很多额外的工作。