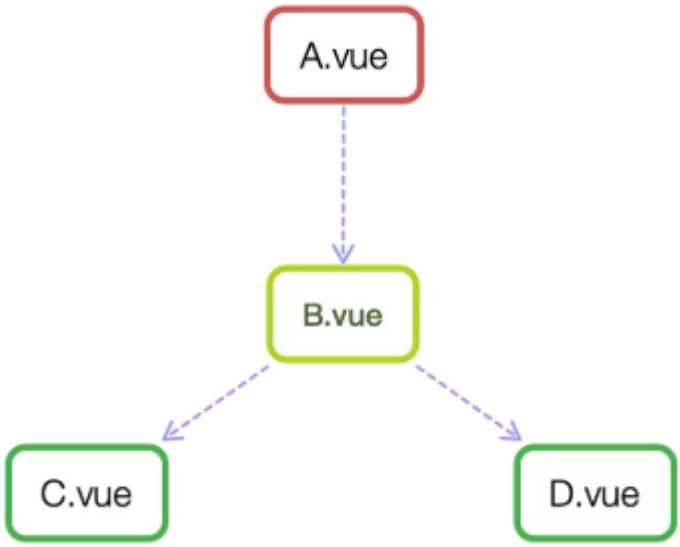
11. vue 中组件之间的通信方式

组件可以有以下几种关系：



A-B、B-C、B-D 都是父子关系

C-D 是兄弟关系

A-C、A-D 是隔代关系

不同使用场景，如何选择有效的通信方式？`vue` 组件中通信的几种方式？

1. `props` ★★
2. `$emit/$on` ★★ 事件总线
3. `vuex` ★★★
4. `$parent/$children`
5. `$attrs/$listeners`
6. `provide/inject` ★★★

vue 中组件之间通信？

> 常见使用场景可以分为三类：

- 父子组件通信
- 兄弟组件通信
- 跨层组件通信

1. `props`

> 父组件 A 通过 `props` 向子组件 B 传递值， B 组件传递 A 组件通过 `$emit` A 组件通过 `v-on/@` 触发

1-1 父组件=>子组件传值

```
// 父组件
<template>
    <div id="app">
        <Child v-bind:child="users"></Child> //前者自定义名称便于子组件调用，后者要传递 数据名
    </div>
</template>
<script>
    import Child from "./components/Child" //子组件
    export default {
        name: 'App',
        data(){
            return{
```

```
                users:["Eric","Andy","Sai"
]               }
        },
        components:{
            "Child":Child
        }
    }
</script>
// 子组件
<template>
    <div class="hello">
        <ul>
            <li v-for="item in child">{{ item }}</li> //遍历传递过来的值渲染页
面         </ul>
    </div>
</template>
<script>
    export default {
        name: 'Hello World',
        props:{
            child:{            //这个就是父组件中子标签自定义名字
              type:Array,      //对传递过来的值进行校验
              required:true    //必添
            }
          }
    }
</script>
```

总结：父组件通过 props 向下传递数据给子组件。

1-2 子组件=>父组件传值

```
// 子组件 Header.vue
<template>
  <div>
    <h1 @click="changeTitle">{{ title }}</h1> //绑定一个点击事件
  </div>
</template>
<script>
    export default {
      name: 'header',
      data() {
        return {
          title:"Vue.js Demo"
        }
      },
      methods:{
        changeTitle() {
          this.$emit("titleChanged","子向父组件传值"); //自定义事件   传递值“子向父组
件 传值”
        }
      }
    }
</script>
```

```
// 父组件
<template>
  <div id="app">
    <header v-
on:titleChanged="updateTitle"></header>     //与子组件
titleChanged 自定义事件保持一致
   // updateTitle($event)接受传递过来的文字
    <h2>{{ title }}</h2>
  </div>
</template>
<script>
import Header from "./components/Header"
    export default {
      name: 'App',
      data(){
        return{
          title:"传递的是一个值"
        }
      },
      methods:{
        updateTitle(e){   //声明这个函数
          this.title = e;
        }
      },
      components:{
       "app-header":Header,
      }
    }
</script>
```

总结：子组件通过 events 给父组件发送消息，实际上就是子组件把自己的数据发送到父组件。

2. $emit/$on => $bus

vue 实例 作为事件总线（事件中心）用来触发事件和监听事件，可以通过此种方式进行组件间通

信包括：父子组件、兄弟组件、跨级组件

例：

创建 bus 文件

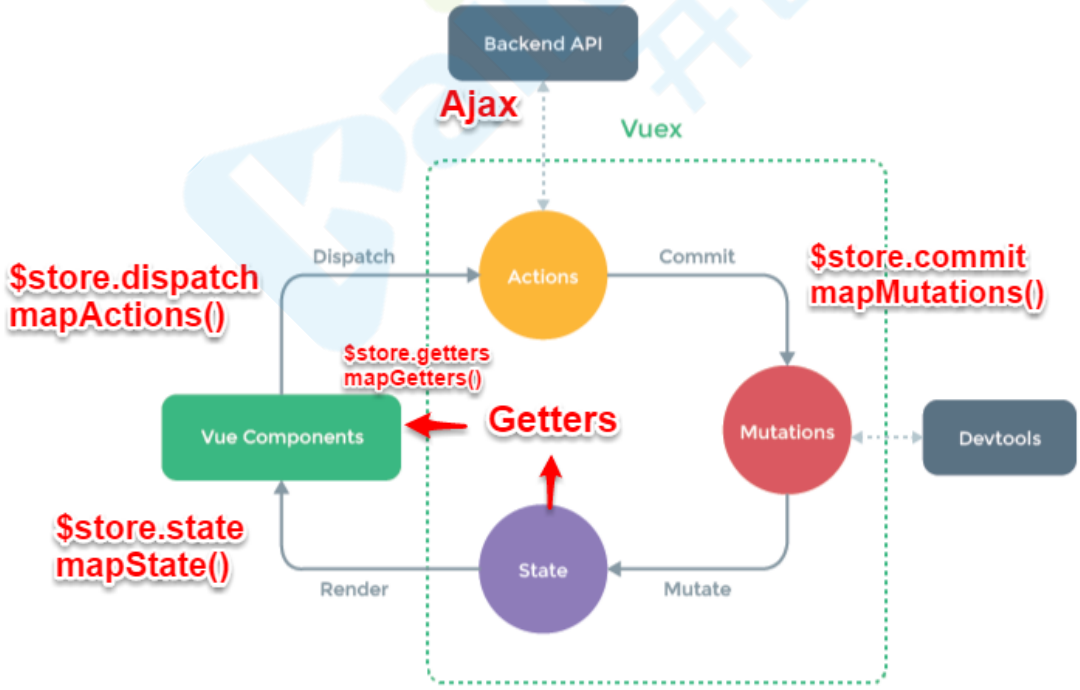
```
import Vue from 'vue'

export defult new Vue()
```

```
// gg 组件
<template id="a">
  <div>
    <h3>gg 组件</h3>
    <button @click="sendMsg">将数据发送给 dd 组件
</button>   </div>
</template>
<script>
import bus from './bus'
export default {
    methods: {
        sendMsg(){
            bus.$emit('sendTitle','传递的值')
```

```
        }
    }
}
</script>
```

```
// dd 组件
<template>
    <div>
        接收 gg 传递过来的值：{{msg}}
    </div>
</template>
<script>
import bus from './bus'
export default {
    data(){
        return {
            mag: ''
        }
    }
    mounted(){
        bus.$on('sendTitle',(val)=>{
            this.mag = val
        })
    }
}
</script>
```

3. vuex



1-1 vuex 介绍

`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(常见于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。

1-2 vuex 中核心概念

- `state`：`vuex` 的唯一数据源，如果获取多个 `state`,可以使用 `...mapState`。

```
export const store = new Vuex.Store({
// 注意 Store 的 S 大写
<!-- 状态储存 -->
   state: {
        productList: [
           {
           name: 'goods 1',
           price: 100

           }
       ]
   }
})
```

- `getter`：可以将 `getter` 理解为计算属性，`getter` 的返回值根据他的依赖缓存起来，依赖发生变 化才会被重新计算。

```
import Vue from 'vue'
import Vuex from 'vuex';
Vue.use(Vuex)

export const store = new Vuex.Store({
    state: {
        productList: [
            {
            name: 'goods 1',
            price: 100
            },
        ]
    },
    // 辅助对象 mapGetter
    getters: {
        getSaledPrice: (state) => {
            let saleProduct = state.productList.map((item) =>
{                 return {
                    name: '**' + item.name + '**',
                    price: item.price / 2
                }
            })
            return saleProduct;
        }
    }
})
```

```
// 获取 getter 计算后的值
export default {
    data () {
        return {
            productList :
this.$store.getters.getSaledPrice          }
    }
}
```

- `mutation`：更改 `vuex` 的 `state` 中唯一的方是提交 `mutation` 都有一个字符串和一个回调函数。 回调函数就是使劲进行状态修改的地方。并且会接收 `state` 作为第一个参数 `payload` 为第二个参 数， `payload` 为自定义函数， `mutation` 必须是同步函数。

```
// 辅助对象 mapMutations
mutations: {
    <!-- payload 为自定义函数名-->
    reducePrice: (state, payload) => {
        return state.productList.forEach((product) =>
{             product.price -= payload;
        })
    }
}
```

```
<!-- 页面使用 -->
methods: {
    reducePrice(){
        this.$store.commit('reducePrice',
4)     }
}
```

- `action`： `action` 类似 `mutation` 都是修改状态，不同之处，

> `action` 提交的 `mutation` 不是直接修改状态
> `action` 可以包含异步操作，而 `mutation` 不行
> `action` 中的回调函数第一个参数是 `context`，是一个与 `store` 实例具有相同属性的方法的
> 对象
> `action` 通过 `store.dispatch` 触发， `mutation` 通过 `store.commit` 提交

```
actions: {
// 提交的是 mutation，可以包含异步操作
    reducePriceAsync: (context, payload) => {
        setTimeout(()=> {
            context.commit('reducePrice', payload);  // reducePrice 为上一
步 mutation 中的属性
        },2000)
    }
}
```

```
<!-- 页面使用 -->
// 辅助对象 mapActions
methods: {
  reducePriceAsync(){
        this.$store.dispatch('reducePriceAsync',
2)    },
}
```

- `module`：由于是使用单一状态树，应用的所有状态集中到比较大的对象，当应用变得非常复杂是，`store` 对象就有可能变得相当臃肿。为了解决以上问题，vuex 允许我们将 `store` 分割成模
  块，每个模块拥有自己的 `state,mutation,action,getter`,甚至是嵌套子模块从上至下进行同样
  方式分割。

```
const moduleA = {
    state: {...},
    mutations: {...},
    actions: {...},
    getters: {...}
}
const moduleB = {
    state: {...},
    mutations: {...},
    actions: {...},
    getters: {...}
}
const store = new
Vuex.Store({    a: moduleA,
    b: moduleB
})
store.state.a
store.state.b
```

1-3 vuex 中数据存储 localStorage

`vuex` 是 `vue` 的状态管理器，存储的数据是响应式的。但是并不会保存起来，刷新之后就回到了初始 状态，具体做法应该在 `vuex` 里数据改变的时候把数据拷贝一份保存到 `localStorage` 里面，刷新之 后，如果 `localStorage` 里有保存的数据，取出来再替换 `store` 里的 `state`。

例：

```
let defaultCity = "上海"
try {
// 用户关闭了本地存储功能，此时在外层加个 try...catch
  if (!defaultCity){
  // f 复制一份
        defaultCity =
JSON.parse(window.localStorage.getItem('defaultCity'))        }
    }catch(e){
        console.log(e)
    }
export default new Vuex.Store({
  state: {
    city: defaultCity
  },
  mutations: {
    changeCity(state, city) {
```

```
      state.city = city
      try {
      window.localStorage.setItem('defaultCity',
JSON.stringify(state.city));       // 数据改变的时候把数据拷贝一份保存到
localStorage 里面
      } catch (e) {}
    }
  }
})
```

注意：vuex 里，保存的状态，都是数组，而 localStorage 只支持字符串，所以需要用 JSON 转换：

```
JSON.stringify(state.subscribeList)<font color="red">// array -> string</font>
JSON.parse(window.localStorage.getItem("subscribeList"))<font color="red">//
string -> array</font>
```

4. `$attr/$listeners`

1-1 简介

多级组件嵌套需要传递数据时，通常使用的方法是通过 vuex。但如果仅仅是传递数据，而不做中间处理，使用 vuex 处理，未免有点大材小用。为此 Vue2.4 版本提供了另一种方法----

`$attrs/$listeners`

- `$attrs`：包含了父作用域中不被 prop 所识别 (且获取) 的特性绑定 (class 和 style 除外)。当一个组件没有声明任何 prop 时，这里会包含所有父作用域的绑定 (class 和 style 除外)，并且可以通过 v-bind="$attrs" 传入内部组件。通常配合 interitAttrs 选项一起使用。
- `$listeners`：包含了父作用域中的 (不含 .native 修饰器的) v-on 事件监听器。它可以通过 v-on="$listeners" 传入内部组件

例：

```
// index.vue
<template>
  <div>
    <h2>王者峡谷</h2>
    <child-com1 :foo="foo" :boo="boo" :coo="coo" :doo="doo" title="前端工匠">
</child-com1>
  </div>
</template>
<script>
    const childCom1 = () => import("./childCom1.vue");
    export default {
      components: { childCom1 },
      data() {
        return {
          foo: "Javascript",
          boo: "Html",
          coo: "CSS",
          doo: "Vue"
        };
      }
    };
</script>
```

```
//childCom1.vue
<template
class="border">   <div>
    <p>foo: {{ foo }}</p>
```

```
    <p>childCom1 的$attrs: {{ $attrs }}</p>
    <child-com2 v-bind="$attrs"></child-com2>
  </div>
</template>
<script>
    const childCom2 = () => import("./childCom2.vue");
    export default {
      components: {
        childCom2
      },
      inheritAttrs: false, // 可以关闭自动挂载到组件根元素上的没有在 props 声明的属
性       props: {
        foo: String // foo 作为 props 属性绑定
      },
      created() {
        console.log(this.$attrs);
        // { "boo": "Html", "coo": "CSS", "doo": "Vue", "title": "前端工匠
" }       }
    };
</script>
```

```
// childCom2.vue
<template>
  <div class="border">
    <p>boo: {{ boo }}</p>
    <p>childCom2: {{ $attrs }}</p>
    <child-com3 v-bind="$attrs"></child-com3>
  </div>
</template>
<script>
const childCom3 = () => import("./childCom3.vue");
export default {
  components: {
    childCom3
  },
  inheritAttrs: false,
  props: {
    boo: String
  },
  created() {
    console.log(this.$attrs);
    // {"coo": "CSS", "doo": "Vue", "title": "前端工匠
" }   }
};
</script>
```

```
// childCom3.vue
<template>
  <div class="border">
    <p>childCom3: {{ $attrs }}</p>
  </div>
</template>
<script>
    export default {
      props: {
        coo: String,
```

```
        title:
String      }
    };
</script>
```

所示 `$attrs` 表示没有继承数据的对象，格式为{属性名：属性值}。Vue2.4 提供了 `$attrs` ，`$listeners` 来传递数据与事件，跨级组件之间的通讯变得更简单。

简单来说：`$attrs` 与`$listeners` 是两个对象，`$attrs` 里存放的是父组件中绑定的非 Props 属性，`$listeners` 里存放的是父组件中绑定的非原生事件。

5. `provide/inject`

1-1 简介

Vue2.2.0 新增 API,这对选项需要一起使用，以允许一个祖先组件向其所有子孙后代注入一个依赖，不论 组件层次有多深，并在起上下游关系成立的时间里始终生效。一言而蔽之：祖先组件中通过 provider 来 提供变量，然后在子孙组件中通过 inject 来注入变量。

provide / inject API 主要解决了跨级组件间的通信问题，不过它的使用场景，主要是子组件获取上级组 件的状态，跨级组件间建立了一种主动提供与依赖注入的关系。

例：

```
//a.vue
export default {
    provide: {
        name: '王者峡谷' //这种绑定是不可响应
的    }
}
```

```
// b.vue
export default {
    inject: ['name'],
    mounted () {
        console.log(this.name) //输出王者峡
谷    }
}
```

> A.vue，我们设置了一个 `provide:name` ，值为王者峡谷，将 name 这个变量提供给它的所有子组
> 件。

> B.vue ，通过 inject 注入了从 A 组件中提供的 name 变量，组件 B 中，直接通过 this.name 访问这个
> 变量了。

> 这就是 provide / inject API 最核心的用法。

> 需要注意的是：provide 和 inject 绑定并不是可响应的。这是刻意为之的。然而，如果你传入了一
> 个可监听的对象，那么其对象的属性还是可响应的----vue 官方文档,所以，上面 A.vue 的 name 如
> 果改变了，B.vue 的 this.name 是不会改变的。

1-2 provide 与 inject 怎么实现数据响应式

两种方法：

1-2-1

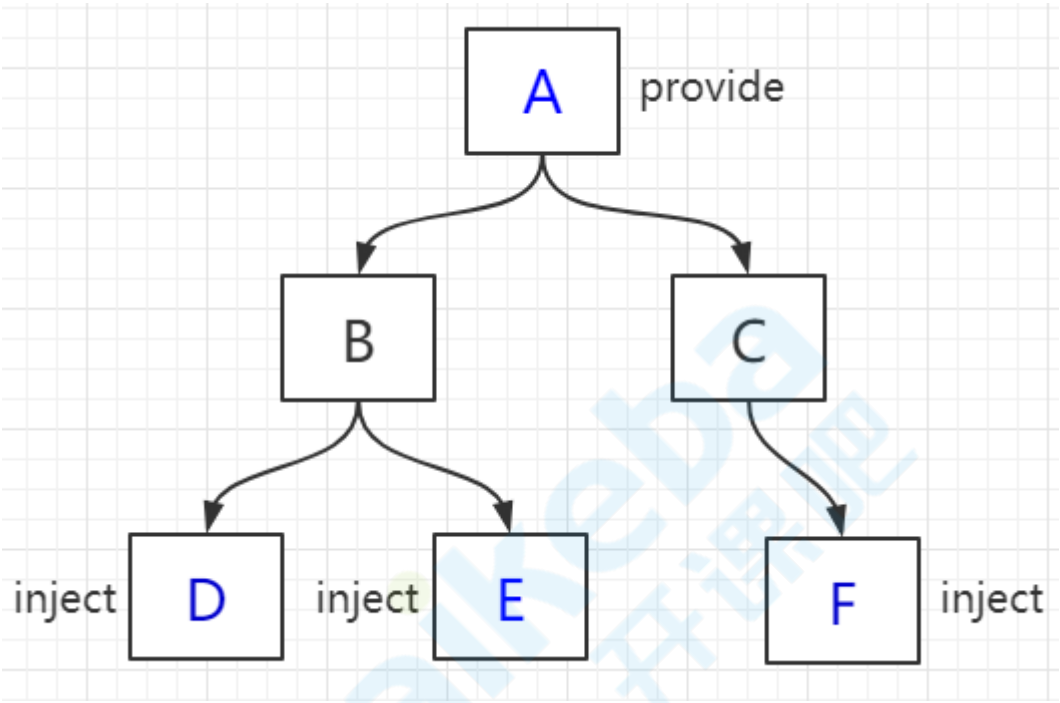
- provide 祖先组件的实例，然后在子孙组件中注入依赖，这样就可以在子孙组件中直接修改祖先组 件的实例的属性，不过这种方法有个缺点就是这个实例上挂载很多没有必要的东西比如 props， methods

1-2-2

- 使用 2.6 最新 API Vue.observable 优化响应式 provide(推荐)

例：

> 组件 D、E 和 F 获取 A 组件传递过来的 color 值，并能实现数据响应式变化，即 A 组件的 color 变化
> 后，组件 D、E、F 会跟着变（核心代码如下：）




```
// A 组件
<div>
      <h1>A 组件</h1>
      <button @click="() => changeColor()">改变
color</button>       <ChildrenB />
      <ChildrenC />
</div>
......
  data() {
    return {
      color: "blue"
    };
  },
  // provide() {
  //   return {
  //     theme: {
  //       color: this.color //这种方式绑定的数据并不是可响应的
  //     } // 即 A 组件的 color 变化后，组件 D、E、F 不会跟着变
  //   };
  // },
  provide() {
    return {
      theme: this//方法一：提供祖先组件的实例
    };
  },
```

```
  methods: {
    changeColor(color) {
      if (color) {
        this.color = color;
      } else {
        this.color = this.color === "blue" ? "red" : "blue";
      }
    }
  }
  // 方法二:使用 2.6 最新 API Vue.observable 优化响应式 provide
  // provide() {
  //   this.theme = Vue.observable({
  //     color: "blue"
  //   });
  //   return {
  //     theme: this.theme
  //   };
  // },
  // methods: {
  //   changeColor(color) {
  //     if (color) {
  //       this.theme.color = color;
  //     } else {
  //       this.theme.color = this.theme.color === "blue" ? "red" :
"blue";   //     }
  //   }
  // }


  // F 组件
<template functional>
  <div class="border2">
    <h3 :style="{ color: injections.theme.color }">F 组件
</h3>   </div>
</template>
<script>
export default {
  inject: {
    theme: {
      //函数式组件取值不一样
      default: () => ({})
    }
  }
};
</script>
```

> 注：provide 和 inject 主要为高阶插件/组件库提供用例，能在业务中熟练运用，可以达到事半功倍的效果！

6. `$parent / $children` 与 `ref`

- `ref`：如果在普通的 DOM 元素上使用，引用指向的就是 DOM 元素；如果用在子组件上，引用就
  指向组件实例
- `$parent / $children`：访问父 / 子实例

> 注意：这两种都是直接得到组件实例，使用后可以直接调用组件的方法或访问数据。我们先来看个用 ref 来访问组件的

例：

```
export default {
  data () {
    return {
      title: 'Vue.js'
    }
  },
  methods: {
    sayHello () {
      window.alert('Hello');
    }
  }
}
```

```
<template>
  <component-a ref="comA"></component-a>
</template>
<script>
  export default {
    mounted () {
      const comA = this.$refs.comA;
      console.log(comA.title);  //
Vue.js       comA.sayHello();  // 弹窗
    }
  }
</script>
```

注：这两种方法的弊端是，无法在跨级或兄弟间通信。

> 我们想在 component-a 中，访问到引用它的页面中（这里就是 parent.vue）的两个 component-b 组件，那这种情况下，就得配置额外的插件或工具了，比如 Vuex 和 Bus 的解决方案。

总结：

- vue 中常用的通信方式由 6 种，分别是：
  1. `props` ★★ （父传子）
  2. `$emit/$on` ★★ 事件总线 （跨层级通信）
  3. `vuex` ★★★（状态管理 常用 皆可） 优点：一次存储数据，所有页面都可访问
  4. `$parent/$children` （父 = 子 项目中不建议使用）缺点：不可跨层级
  4. `$attrs/$listeners` （皆可 如果搞不明白 不建议和面试官说这一种）
  5. `provide/inject` ★★★ （高阶用法 = 推荐使用 ） 优点：使用简单 缺点：不是响应式

12. vue-router 中的导航钩子由那些？

1.全局的钩子函数

- `beforeEach`(to，from，next) 路由改变前调用

- 常用验证用户权限
- `beforeEach` ()参数
  to ：即将要进入的目标路由对象
  from：当前正要离开的路由对象
  next：路由控制参数

  next()：如果一切正常，则调用这个方法进入下一个钩子

  next(false)：取消导航（即路由不发生改变）

  next('/login')：当前导航被中断，然后进行一个新的导航

  next(error)：如果一个 Error 实例，则导航会被终止且该错误会被传递给
- `router.onError` () `afterEach` (to，from) 路由改变后的钩子
  - 常用自动让页面返回最顶端
  - 用法相似，少了 next 参数

```
router.beforeEach((to, from, next) => {
    console.log(to.fullPath);
    if(to.fullPath!='/login'){//如果不是登录组件
        if(!localStorage.getItem("username")){//如果没有登录，就先进入 login
组件 进行登录
            next('/login');
        }else{//如果登录了，那就继续
            next();
        }
    }else{//如果是登录组件，那就继续。
        next();
    }
})
```

2.路由配置中的导航钩子

- `beforeEnter` (to，from，next)

```
const router = new VueRouter({
  routes: [
    {
      path: '/foo',
      component: Foo,
      beforeEnter: (to, from, next) =>
{        // ...
      },
      beforeEnter: (route) => {
        // ...
      }
    }
  ]
});
```

3.组件内的钩子函数

- 钩子函数介绍

1. `beforeRouteEnter`(to,from,next)
   - 该组件的对应路由被 `comfirm` 前调用。
   - 此时实例还没被创建，所以不能获取实例（this）
2. `beforeRouteUpdate`(to,from,next)
   - 当前路由改变，但改组件被复用时候调用
   - 该函数内可以访问组件实例(this)
3. `beforeRouteLeave`(to,from,next)
   - 当导航离开组件的对应路由时调用。
   - 该函数内可以访问获取组件实例（this）

```
const Foo = {
  template: `...`,
  beforeRouteEnter (to, from, next) {
    // 在渲染该组件的对应路由被 confirm 前调用
    // 不！能！获取组件实例 `this`
    // 因为当钩子执行前，组件实例还没被创建
  },
  beforeRouteUpdate (to, from, next) {
    // 在当前路由改变，但是该组件被复用时调用
    // 举例来说，对于一个带有动态参数的路径 /foo/:id，在 /foo/1 和 /foo/2 之间跳转
的 时候，
    // 由于会渲染同样的 Foo 组件，因此组件实例会被复用。而这个钩子就会在这个情况下被调
用。
    // 可以访问组件实例 `this`
  },
  beforeRouteLeave (to, from, next) {
    // 导航离开该组件的对应路由时调用
    // 可以访问组件实例 `this`
  }
}
```

4.路由监测变化

- 监听到路由对象发生变化，从而对路由变化做出响

  应

```
const user = {
    template:'<div></div>',
    watch: {
        '$route' (to,from){
            // 对路由做出响应
              //  to , from 分别表示从哪跳转到哪，都是一个对象
              // to.path   ( 表示的是要跳转到的路由的地址 eg:
/home );         }
    }
}
// 多了一个 watch，这会带来依赖追踪的内存开销，
// 修改
const user = {
    template:'<div></div>',
    watch: {
        '$route.query.id' {
            // 请求个人描述
        },
```

```
            '$route.query.page'
{               // 请求列表
            }
    }
}
```

总结：

路由中的导航钩子有三种

- 全局
- 组件
- 路由配置

在做页面登陆权限时候可以使用到路由导航配置 （举例两三个即可）

监听路由变化怎么做

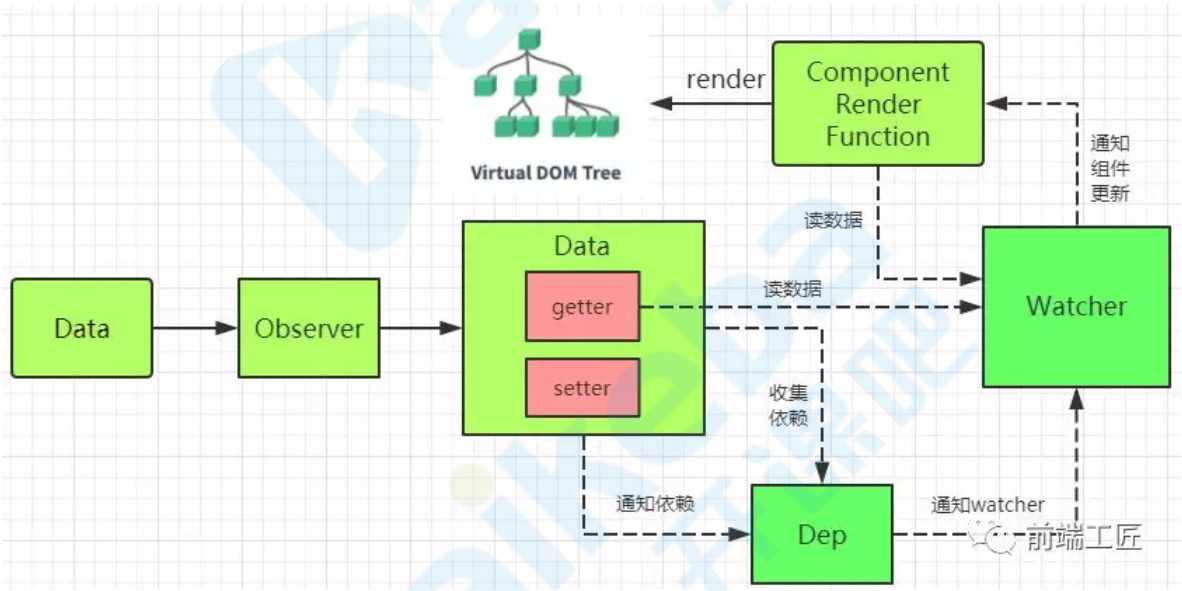
> 使用 watch 来对$route 监听

13.在最后

14. 说一说 vue 响应式理解？

响应式实现：

- `object.defineProperty`
- proxy(兼容性不太好)



observer 类

```
/* observer 类会附加到每一个被侦测的 object 上
* 一旦被附加上，observer 会被 object 的所有属性转换为 getter/setter 的
形式 * 当属性发生变化时候及时通知依赖
*/
// Observer 实例
export class Observer {
  constructor (value) {
    this.value = value
    if (!Array.isArray(value)) { // 判断是否是数组
      this.walk(value) // 劫持对象
    }
  }

  walk (obj) { // 将会每一个属性转换为 getter/setter 形式来侦测数据变
化     const keys = Object.keys(obj)
    for (let i = 0; i < keys.length; i++) {
      defineReactive(obj, keys[i], obj[key[i]]) // 数据劫持方法
    }
  }
function defineReactive(data,key,val){
    // 递归属性
    if(typeof val ==='object'){
        new Obeserve(val)
    }
```

```
    let dep = new Dep()
    Object.defineProperty(data,key,{
         enumerable:true,
        configurable:true,
        get:function(){
            dep.depend()
            return val
        },
        set:function(newVal){
            if(val===newVal){
                return
            }
            val = newVal
            dep.notify()
        }
    })
}
```

定义了 observer 类，用来将一个正常的 object 转换成被侦测的 object 然后判断数据类型

只有 object 类型才会调用 walk 将每一个属性转换成 getter/setter 的形式来侦测变化

最后在 defineReactive 中新增 new Observer（val）来递归子属性

当 data 中的属性变化时，与这个属性对应的依赖就会接收通知

dep 依赖收集

getter 中收集依赖，那么这些依赖收集到那？

```
export default class Dep {
  constructor () {
    this.subs = [] // 观察者集合
  }
 // 添加观察者
  addSub (sub) {
    this.subs.push(sub)
  }
 // 移除观察者
  removeSub (sub) {
    remove(this.subs, sub)
  }

  depend () { // 核心，如果存在 ，则进行依赖收集操
作     if (window.target) {
     this.addDep(window.target)
    }
  }

  notify () {
    const subs = this.subs.slice() // 避免污染原来的集合
    // 如果不是异步执行，先进行排序，保证观察者执行顺序
    if (process.env.NODE_ENV !== 'production' && !config.async)
{       subs.sort((a, b) => a.id - b.id)
    }
    for (let i = 0, l = subs.length; i < l; i++) {
      subs[i].update() // 发布执行
    }
```

```
  }
}
function remove(arr,item){
    if(arr.length){
        const index =
arr.indexOf(item)         if(index > -
1){
            return
arr.splice(index,1)          }
    }
}
```

> 收集的依赖时`window.target`,他到以是什么？
>
> 当属性变化时候我们通知谁？

watcher

> 是一个中介的角色，数据发生变化时通知它，然后它再去通知其他地
>
> 方

```
export default class Watcher {
    constructor (vm,expOrFn,cb) {
        // 组件实例对象
        // 要观察的表达式，函数，或者字符串，只要能触发取值操作
        // 被观察者发生变化后的回调
    this.vm = vm // Watcher 有一个 vm 属性，表明它是属于哪个组件
的       // 执行 this.getter()及时读取数据
    this.getter = parsePath(expOrFn)
      this.cb = cb
      this.value = this.get()
  }
    get(){
        window.target = this
        let value = this.getter.call(this.vm,this.vm)
        window.target = undefined
        return value
    }
    update(){
        const oldValue = this.value
        this.value = this.get()
        this.cb.call(this.vm,this.value,oldValue)
    }
}
```

总结

data 通过 Observer 转换成了 getter/setter 的形式来追踪变化

当外界通过 Watcher 读取数据的，会触发 getter 从而将 watcher 添加到依赖中

当数据变化时，会触发 setter 从而向 Dep 中的依赖（watcher）发送通知

watcher 接收通知后，会向外界发送通知，变化通知到外界后可能会触发视图更新，也有可能触发用户 的某个回调函数等

- 什么是响应式

我们先来看个例子：

```
<div id="app">
    <div>Price :￥{{ price }}</div>
    <div>Total:￥{{ price * num }}</div>
    <div>Taxes: ￥{{ totalPrice }}</div>
    <button @click="changePrice">改变价格</button>
</div>
var app = new Vue({
    el: '#app',
    data() {
        return {
            price: 5.0,
            num: 2
        };
    },
    computed: {
        totalPrice() {
            return this.price * this.num * 1.03;
        }
    },
    methods: {
        changePrice() {
            this.price = 10;
        }
    }
})
```

上例中当 price 发生变化的时候，`Vue` 就知道自己需要做三件事情：

- 更新页面上 price 的值
- 计算表达式 `price * num` 的值，更新页面
- 调用 `totalPrice` 函数，更新页面

数据发生变化后，会重新对页面渲染，这就是 `Vue` 响应式！

想完成这个过程，我们需要：

1. 侦测数据的变化
2. 收集视图依赖了哪些数据
3. 数据变化时，自动“通知”需要更新的视图部分，并进行更新

对应专业俗语分别是：

- 数据劫持 / 数据代理
- 依赖收集
- 发布订阅模式

15. vue 如果想要扩展某个组件现有组件时怎么做？

1. 使用 Vue.mixin 全局混入

混入 **(mixins)** 是一种分发 **Vue** 组件中可复用功能的非常灵活的方式。混入对象可以包含任意组件选项。当组件使用混入对象时，所有混入对象的选项将被混入该组件本身的选项。mixins 选项接受一个 混合对象的数组。

```
<template>
  <div id="app">
    <p>num:{{ num }}</p>
    <P>
         <button @click="add">增加数量
</button>     </P>
  </div>
</template>

<script>
var addLog = { //额外临时加入时，用于显示日志
  updated : function() {
    console.log("数据发生变化,变化成" + this.num +
".");   }
}

export default {
  name: 'app',
  data() {
    return {
      num:1
    }
  },
  methods: {
    add(){
      this.num++
    }
  },
  updated(){
    console.log("我是原生的
update")   },
  mixins: [addLog] //混入
}
</script>

<style>

</style>
```

```
Vue.mixin({// 全局注册一个混入，影响注册之后所有创建的每个 Vue 实
例   updated: function () {
    console.log("我是全局的混入")
  }
})
```

mixins 的调用顺序:

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

2. 加 slot 扩展

- 默认插槽和匿名插槽

  slot 用来获取组件中的原内容。

```
<template id="hello">
  <div>
    <h3>kaikeba</h3>
    <slot>如果没有原内容，则显示该内容</slot>// 默认插
槽   </div>
</template>
<script>
  var vm=new Vue({
      el:'#app',
      components:{
      'my-hello':{
      template:'#hello'
    }
     }
 });
</script>
```

- 具名插槽

```
<div id="itany">
    <my-hello>
      <ul slot="s1">
    <li>aaa</li>
    <li>bbb</li>
    <li>ccc</li>
      </ul>
      <ol slot="s2">
    <li>111</li>
    <li>222</li>
    <li>333</li>
      </ol>
    </my-hello>
</div>
<template id="hello">
    <div>
      <slot name="s2"></slot>
      <h3>welcome to
kaikeba</h3>       <slot
name="s1"></slot>
    </div>
</template>
<script>
  var vm=new Vue({
    el:'#itany',
    components:{
      'my-hello':{
        template:'#hello'
       }
    }
 });
</script>
```

总结：

1. 使用 mixin 全局混入
2. 使用 slot 扩展

13. 你知道 nextTick 的原理吗?

nextTick 官方文档的解释，它可以在 DOM 更新完毕之后执行一个回调

```
// 修改数据
vm.msg = 'Hello'
// DOM 还没有更新
Vue.nextTick(function ()
{ // DOM 更新了
})
```

尽管 MVVM 框架并不推荐访问 DOM，但有时候确实会有这样的需求，尤其是和第三方插件进行配合的 时候，免不了要进行 DOM 操作。而 nextTick 就提供了一个桥梁，确保我们操作的是更新后的 DOM。

- vue 如何检测到 DOM 更新完毕呢？

能监听到 DOM 改动的 API：MutationObserver

- 理解 MutationObserver

MutationObserver 是 HTML5 新增的属性，用于监听 DOM 修改事件，能够监听到节点的属性、文本内
容、子节点等的改动，是一个功能强大的利器。

```
//MutationObserver 基本用法
var observer = new
MutationObserver(function(){   //这里是回调函数
  console.log('DOM 被修改了！');
});
var article = document.querySelector('article');
observer.observer(article);
```

vue 是不是用 MutationObserver 来监听 DOM 更新完毕的呢？

vue 的源码中实现 nextTick 的地方：

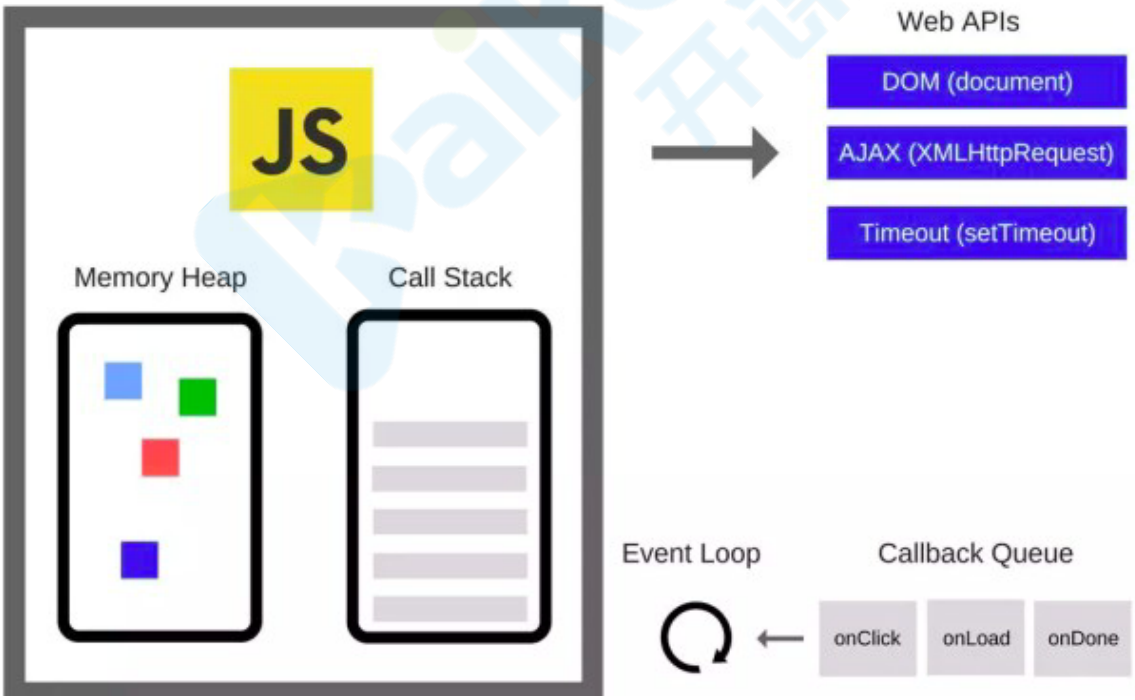
```
//vue@2.2.5 /src/core/util/env.js
if (typeof MutationObserver !== 'undefined' && (isNative(MutationObserver)
||  MutationObserver.toString() === '[object MutationObserverConstructor]'))
{
  var counter = 1
  var observer = new MutationObserver(nextTickHandler)
  var textNode = document.createTextNode(String(counter))
  observer.observe(textNode, {
      characterData: true
  })
  timerFunc = () => {
    counter = (counter + 1) % 2
    textNode.data = String(counter)
  }
}
```

- 事件循环（Event Loop）

在 js 的运行环境中，通常伴随着很多事件的发生，比如用户点击、页面渲染、脚本执行、网络请求，等

等。为了协调这些事件的处理，浏览器使用事件循环机制。

简要来说，事件循环会维护一个或多个任务队列（task queues），以上提到的事件作为任务源往队列中加入任务。有一个持续执行的线程来处理这些任务，每执行完一个就从队列中移除它，这就是一次事 件循环。




```
for(let i=0; i<100; i++){
    dom.style.left = i + 'px';
}
```

事实上，这 100 次 for 循环同属一个 task，浏览器只在该 task 执行完后进行一次 DOM 更新。 只要让 nextTick 里的代码放在 UI render 步骤后面执行，岂不就能访问到更新后的 DOM 了？

> vue 就是这样的思路，并不是用 MO 进行 DOM 变动监听，而是用队列控制的方式达到目的。那么 vue 又是如何做到队列控制的呢？我们可以很自然的想到 setTimeout，把 nextTick 要执行的代码当作下一个 task 放入队列末尾。

> vue 的数据响应过程包含：数据更改->通知 Watcher->更新 DOM。而数据的更改不由我们控制，可能在任何时候发生。如果恰巧发生在重绘之前，就会发生多次渲染。这意味着性能浪费，是 vue 不愿意看到的。

> 所以，vue 的队列控制是经过了深思熟虑的。在这之前，我们还需了解 event loop 的另一个重要概念，microtask。

- microtask

从名字看，我们可以把它称为微任务。

每一次事件循环都包含一个 microtask 队列，在循环结束后会依次执行队列中的 microtask 并移除，然后 再开始下一次事件循环。
在执行 microtask 的过程中后加入 microtask 队列的微任务，也会在下一次事件循环之前被执行。也就是 说，macrotask 总要等到 microtask 都执行完后才能执行，microtask 有着更高的优先级。

microtask 的这一特性，是做队列控制的最佳选择。vue 进行 DOM 更新内部也是调用 nextTick 来做异步 队列控制。而当我们自己调用 nextTick 的时候，它就在更新 DOM 的那个 microtask 后追加了我们自己的 回调函数，从而确保我们的代码在 DOM 更新后执行，同时也避免了 setTimeout 可能存在的多次执行问 题。

常见的 microtask 有：Promise、MutationObserver、Object.observe(废弃)，以及 nodejs 中的 process.nextTick.

看到了 MutationObserver，vue 用 MutationObserver 是想利用它的 microtask 特性，而不是想做 DOM 监听。核心是 microtask，用不用 MutationObserver 都行的。事实上，vue 在 2.5 版本中已经删去了
MutationObserver 相关的代码，因为它是 HTML5 新增的特性，在 iOS 上尚有 bug。

那么最优的 microtask 策略就是 Promise 了，而令人尴尬的是，Promise 是 ES6 新增的东西，也存在兼容 问题呀。所以 vue 就面临一个降级策略。

- vue 的降级策略

上面我们讲到了，队列控制的最佳选择是 microtask，而 microtask 的最佳选择是 Promise.但如果当前

环

境不支持 Promise，vue 就不得不降级为 macrotask 来做队列控制了。

macrotask 有哪些可选的方案呢？前面提到了 setTimeout 是一种，但它不是理想的方案。因为 setTimeout 执行的最小时间间隔是约 4ms 的样子，略微有点延迟。

在 vue2.5 的源码中，macrotask 降级的方案依次是：setImmediate、MessageChannel、setTimeout.

setImmediate 是最理想的方案了，可惜的是只有 IE 和 nodejs 支持。

MessageChannel 的 onmessage 回调也是 microtask，但也是个新 API，面临兼容性的尴尬。

所以最后的兜底方案就是 setTimeout 了，尽管它有执行延迟，可能造成多次渲染，算是没有办法的办法 了。

总结：

以上就是 vue 的 nextTick 方法的实现原理了，总结一下就是：

1. vue 用异步队列的方式来控制 DOM 更新和 nextTick 回调先后执行

2. microtask 因为其高优先级特性，能确保队列中的微任务在一次事件循环前被执行完毕

3. 因为兼容性问题，vue 不得不做了 microtask 向 macrotask 的降级方案