1. v-if和v-for哪个优先级更高？如果两个同时出现，应该怎么优化得到更好的性能？

源码中找答案compiler/codegen/index.js

```
<p v-for="item in items" v-if="condition">
```

做个测试如下

```
<!DOCTYPE html>
<html>

<head>
    <title>Vue事件处理</title>
</head>

<body>
    <div id="demo">
        <h1>v-for和v-if谁的优先级高？应该如何正确使用避免性能问题？</h1>
        <!-- <p v-for="child in children" v-if="isFolder">{{child.title}}</p> --
>
        <template v-if="isFolder">
            <p v-for="child in children">{{child.title}}</p>
        </template>
    </div>
    <script src="../../dist/vue.js"></script>
    <script>
        // 创建实例
        const app = new Vue({
```

```
            el: '#demo',
            data() {
                return {
                    children: [
                        {title:'foo'},
                        {title:'bar'},
                    ]
                }
            },
            computed: {
                isFolder() {
                    return this.children && this.children.length > 0
                }
            },
        });
        console.log(app.$options.render);
    </script>
</body>
</html>
```

两者同级时，渲染函数如下：

```
(function anonymous(
) {
with(this){return _c('div',{attrs:{"id":"demo"}},[_c('h1',[_v("v-for和v-if谁的优先级高？应该如何正确使用避免性能问题？")]),_v(" "),
_l((children),function(child){return (isFolder)?_c('p',
[_v(_s(child.title))]):_e()})],2)}
})
```

> _l包含了isFolder的条件判断

两者不同级时，渲染函数如下

```
(function anonymous(
) {
with(this){return _c('div',{attrs:{"id":"demo"}},[_c('h1',[_v("v-for和v-if谁的优先级高？应该如何正确使用避免性能问题？")]),_v(" "),
(isFolder)?_l((children),function(child){return _c('p',
[_v(_s(child.title))])}):_e()],2)}
})
```

> 先判断了条件再看是否执行_l

结论：

1. 显然v-for优先于v-if被解析（把你是怎么知道的告诉面试官）
2. 如果同时出现，每次渲染都会先执行循环再判断条件，无论如何循环都不可避免，浪费了性能
3. 要避免出现这种情况，则在外层嵌套template，在这一层进行v-if判断，然后在内部进行v-for循环
4. 如果条件出现在循环内部，可通过计算属性提前过滤掉那些不需要显示的项

2. Vue组件data为什么必须是个函数而Vue的根实例则没有此限制？

源码中找答案：src\core\instance\state.js - initData()

> 函数每次执行都会返回全新data对象实例

测试代码如下

```
<!DOCTYPE html>
<html>

<head>
    <title>Vue事件处理</title>
</head>

<body>
    <div id="demo">
        <h1>vue组件data为什么必须是个函数? </h1>
        <comp></comp>
        <comp></comp>
    </div>
    <script src="../../dist/vue.js"></script>
    <script>
        Vue.component('comp', {
            template:'<div @click="counter++">{{counter}}</div>',
            data: {counter: 0}
        })
        // 创建实例
        const app = new Vue({
            el: '#demo',
        });
    </script>
</body>
</html>
```

```
▶ [Vue warn]: The "data" option should be a function that returns a per-instance value in component definitions.   debug.js:24

2 ▶ [Vue warn]: Property or method "counter" is not defined on the instance but referenced during render. Make sure that this property is reactive, either in the data option, or for class-based components, by initializing the property. See: https://vuejs.org/v2/guide/reactivity.html#Declaring-Reactive-Properties.   debug.js:24

found in

---> <Comp>
       <Root>
```

> 程序甚至无法通过vue检测

结论

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，那么状态变更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。

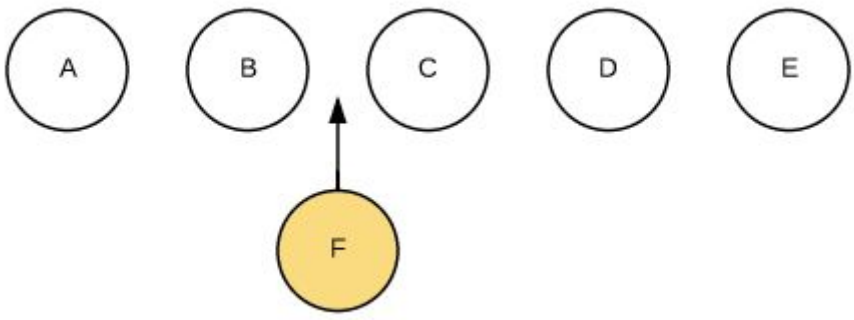
3. 你知道vue中key的作用和工作原理吗？说说你对它的理解。

源码中找答案：src\core\vdom\patch.js - updateChildren()

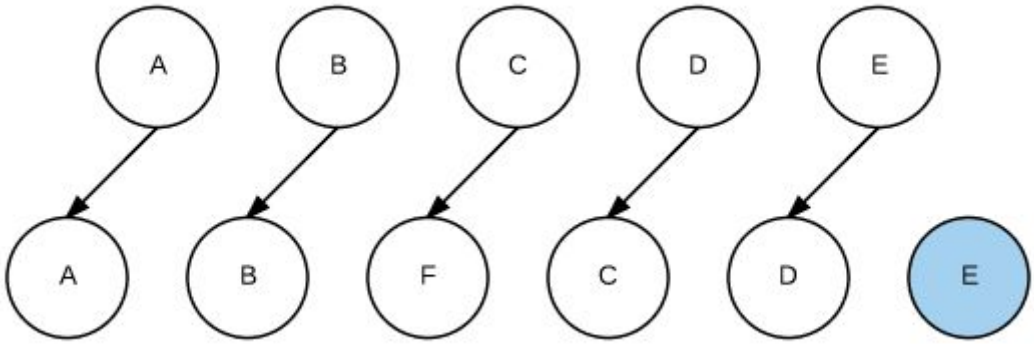
测试代码如下

```
<!DOCTYPE html>
<html>

<head>
    <title>03-key的作用及原理?</title>
</head>

<body>
    <div id="demo">
        <p v-for="item in items" :key="item">{{item}}</p>
    </div>
    <script src="../../dist/vue.js"></script>
    <script>
        // 创建实例
        const app = new Vue({
            el: '#demo',
            data: { items: ['a', 'b', 'c', 'd', 'e'] },
            mounted () {
                setTimeout(() => {
                    this.items.splice(2, 0, 'f')
                }, 2000);
            },
        });
    </script>
</body>

</html>
```

上面案例重现的是以下过程



不使用key



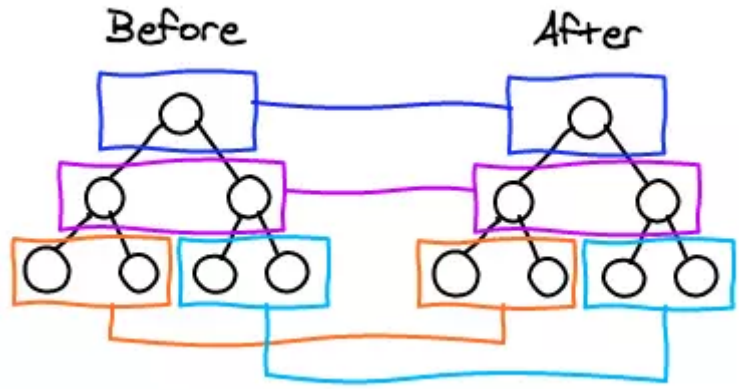
如果使用key

```
// 首次循环patch A
A B C D E
A B F C D E

// 第2次循环patch B
B C D E
B F C D E

// 第3次循环patch E
C D E
F C D E

// 第4次循环patch D
C D
F C D

// 第5次循环patch C
C
F C

// oldCh全部处理结束，newCh中剩下的F，创建F并插入到C前面
```

结论

1. key的作用主要是为了高效的更新虚拟DOM，其原理是vue在patch过程中通过key可以精准判断两个节点是否是同一个，从而避免频繁更新不同元素，使得整个patch过程更加高效，减少DOM操作量，提高性能。
2. 另外，若不设置key还可能在列表更新时引发一些隐蔽的bug
3. vue中在使用相同标签名元素的过渡切换时，也会使用到key属性，其目的也是为了让vue可以区分它们，否则vue只会替换其内部属性而不会触发过渡效果。

4. 你怎么理解vue中的diff算法?



源码分析1：必要性，lifecycle.js - mountComponent()

组件中可能存在很多个data中的key使用

源码分析2：执行方式，patch.js - patchVnode()

patchVnode是diff发生的地方，整体策略：深度优先，同层比较

源码分析3：高效性，patch.js - updateChildren()

测试代码：

```
<!DOCTYPE html>
<html>

<head>
    <title>Vue源码剖析</title>
    <script src="../../dist/vue.js"></script>
</head>

<body>
    <div id="demo">
        <h1>虚拟DOM</h1>
        <p>{{foo}}</p>
    </div>
    <script>
        // 创建实例
        const app = new Vue({
            el: '#demo',
            data: { foo: 'foo' },
            mounted() {
                setTimeout(() => {
                    this.foo = 'fooooo'
                }, 1000);
            }
        });
    </script>
</body>

</html>
```

总结

1.diff算法是虚拟DOM技术的必然产物：通过新旧虚拟DOM作对比（即diff），将变化的地方更新在真实DOM上；另外，也需要diff高效的执行对比过程，从而降低时间复杂度为O(n)。

2.vue 2.x中为了降低Watcher粒度，每个组件只有一个Watcher与之对应，只有引入diff才能精确找到发生变化的地方。

3.vue中diff执行的时刻是组件实例执行其更新函数时，它会比对上一次渲染结果oldVnode和新的渲染结果newVnode，此过程称为patch。

4.diff过程整体遵循深度优先、同层比较的策略；两个节点之间比较会根据它们是否拥有子节点或者文本节点做不同操作；比较两组子节点是算法的重点，首先假设头尾节点可能相同做4次比对尝试，如果没有找到相同节点才按照通用方式遍历查找，查找结束再按情况处理剩下的节点；借助key通常可以非常精确找到相同节点，因此整个patch过程非常高效。

5. 谈一谈对vue组件化的理解?

回答总体思路：

组件化定义、优点、使用场景和注意事项等方面展开陈述，同时要强调vue中组件化的一些特点。

源码分析1：组件定义

```
// 组件定义
Vue.component('comp', {
    template: '<div>this is a component</div>'
})
```

> 组件定义，src\core\global-api\assets.js

```
<template>
    <div>
        this is a component
    </div>
</template>
```

> vue-loader会编译template为render函数，最终导出的依然是组件配置对象。

源码分析2：组件化优点

lifecycle.js - mountComponent()

> 组件、Watcher、渲染函数和更新函数之间的关系

源码分析3：组件化实现

构造函数，src\core\global-api\extend.js

实例化及挂载，src\core\vdom\patch.js - createElm()

总结

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. 组件使用按分类有：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，框架后续会生成其构造函数，它们基于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，它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。

6. 谈一谈对vue设计原则的理解?

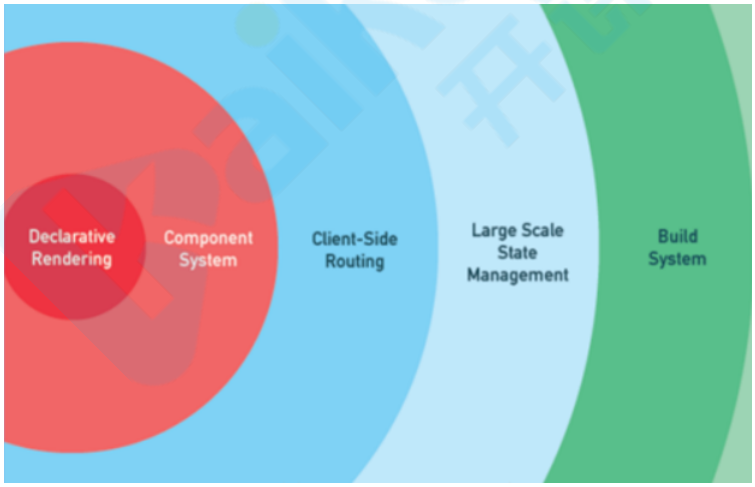
在vue的官网上写着大大的定义和特点：

- 渐进式JavaScript框架
- 易用、灵活和高效

所以阐述此题的整体思路按照这个展开即可。

渐进式JavaScript框架：

与其它大型框架不同的是，Vue 被设计为可以自底向上逐层应用。Vue 的核心库只关注视图层，不仅易于上手，还便于与第三方库或既有项目整合。另一方面，当与现代化的工具链以及各种支持类库结合使用时，Vue 也完全能够为复杂的单页应用提供驱动。



易用性

vue提供数据响应式、声明式模板语法和基于配置的组件系统等核心特性。这些使我们只需要关注应用的核心业务即可，只要会写js、html和css就能轻松编写vue应用。

灵活性

渐进式框架的最大优点就是灵活性，如果应用足够小，我们可能仅需要vue核心特性即可完成功能；随着应用规模不断扩大，我们才可能逐渐引入路由、状态管理、vue-cli等库和工具，不管是应用体积还是学习难度都是一个逐渐增加的平和曲线。

高效性

超快的虚拟 DOM 和 diff 算法使我们的应用拥有最佳的性能表现。

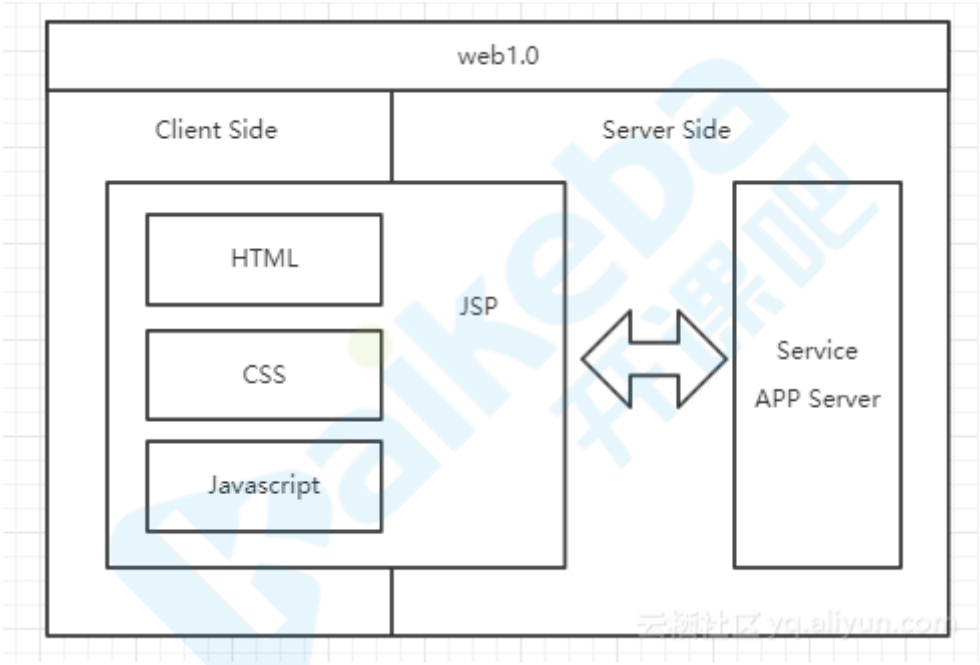
追求高效的过程还在继续，vue3中引入Proxy对数据响应式改进以及编译器中对于静态内容编译的改进都会让vue更加高效。

7. 谈谈你对MVC、MVP和MVVM的理解?

答题思路：此题涉及知识点很多，很难说清、说透，因为mvc、mvp这些我们前端程序员自己甚至都没用过。但是恰恰反映了前端这些年从无到有，从有到优的变迁过程，因此沿此思路回答将十分清楚。

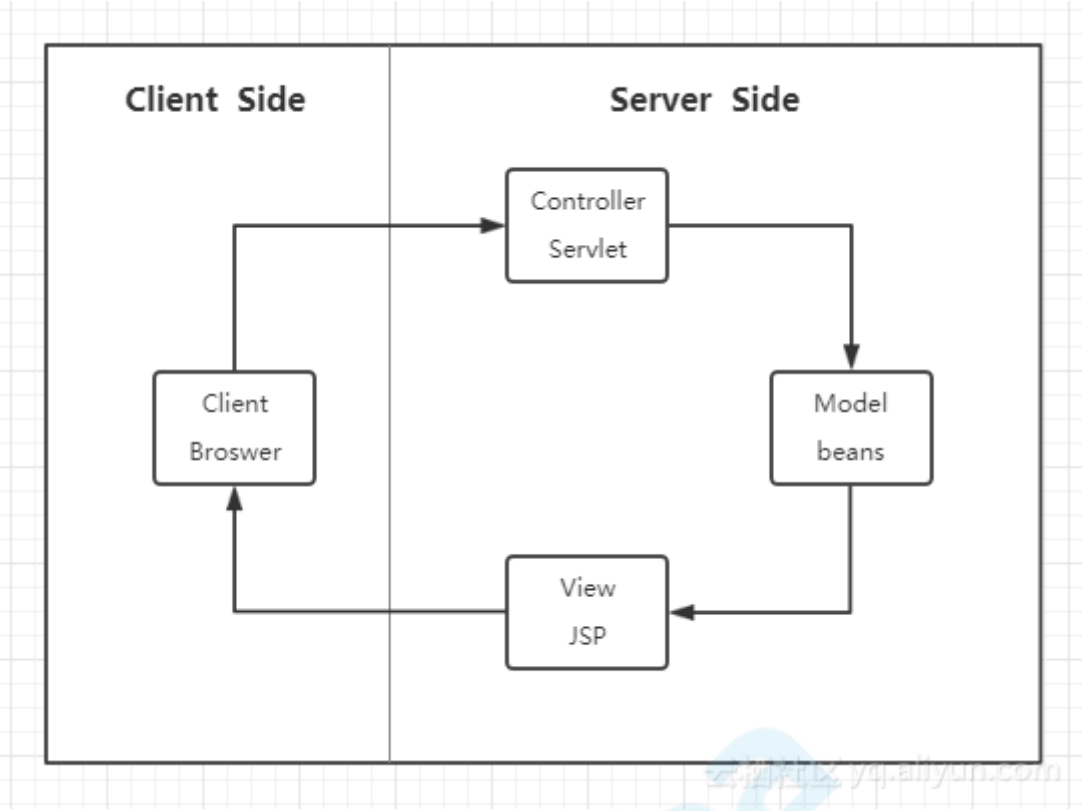
Web1.0时代

在web1.0时代，并没有前端的概念。开发一个web应用多数采用ASP.NET/Java/PHP编写，项目通常由多个aspx/jsp/php文件构成，每个文件中同时包含了HTML、CSS、JavaScript、C#/Java/PHP代码，系统整体架构可能是这样子的：



这种架构的好处是简单快捷，但是，缺点也非常明显：JSP代码难以维护

为了让开发更加便捷，代码更易维护，前后端职责更清晰。便衍生出MVC开发模式和框架，前端展示以模板的形式出现。典型的框架就是Spring、Structs、Hibernate。整体框架如图所示：



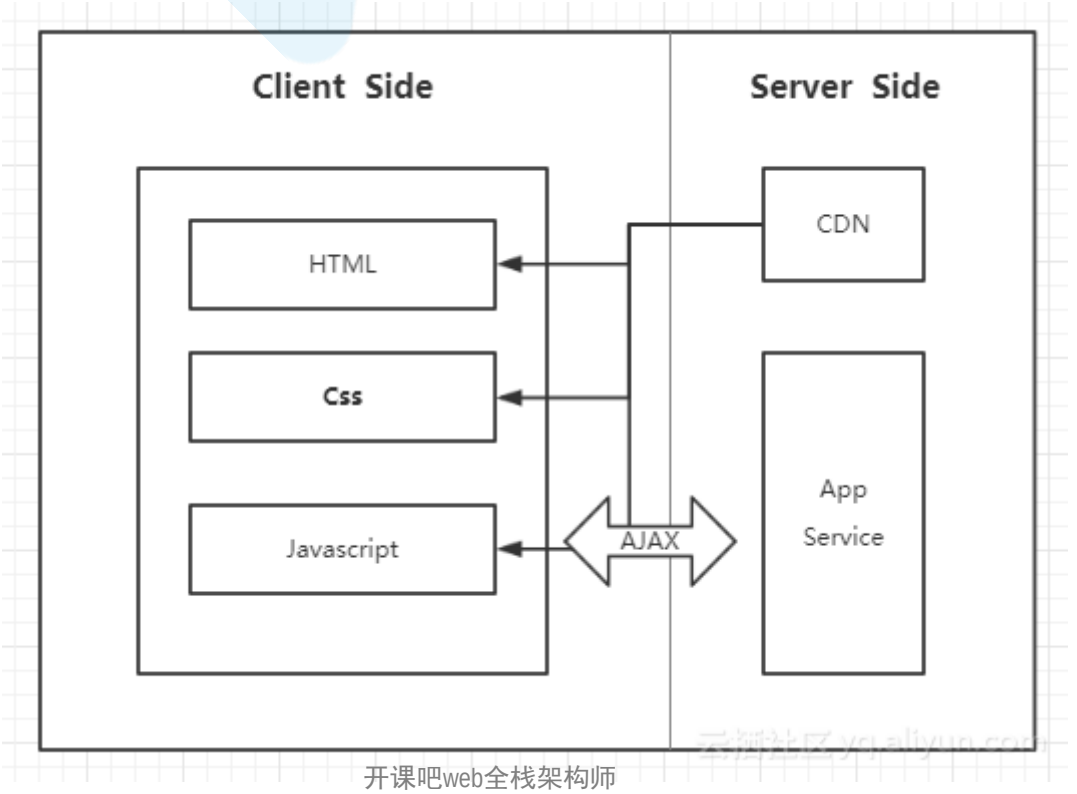
使用这种分层架构，职责清晰，代码易维护。但这里的MVC仅限于后端，前后端形成了一定的分离，前端只完成了后端开发中的view层。

但是，同样的这种模式存在着一些：

1. 前端页面开发效率不高
2. 前后端职责不清

web 2.0时代

自从Gmail的出现，ajax技术开始风靡全球。有了ajax之后，前后端的职责就更加清晰了。因为前端可以通过Ajax与后端进行数据交互，因此，整体的架构图也变化成了下面这幅图：



通过ajax与后台服务器进行数据交换，前端开发人员，只需要开发页面这部分内容，数据可由后台进行提供。而且ajax可以使得页面实现部分刷新，减少了服务端负载和流量消耗，用户体验也更佳。这时，才开始有专职的前端工程师。同时前端的类库也慢慢的开始发展，最著名的就是jQuery了。

当然，此架构也存在问题：缺乏可行的开发模式承载更复杂的业务需求，页面内容都杂糅在一起，一旦应用规模增大，就会导致难以维护了。因此，前端的MVC也随之而来。

前后端分离后的架构演变——MVC、MVP和MVVM

MVC

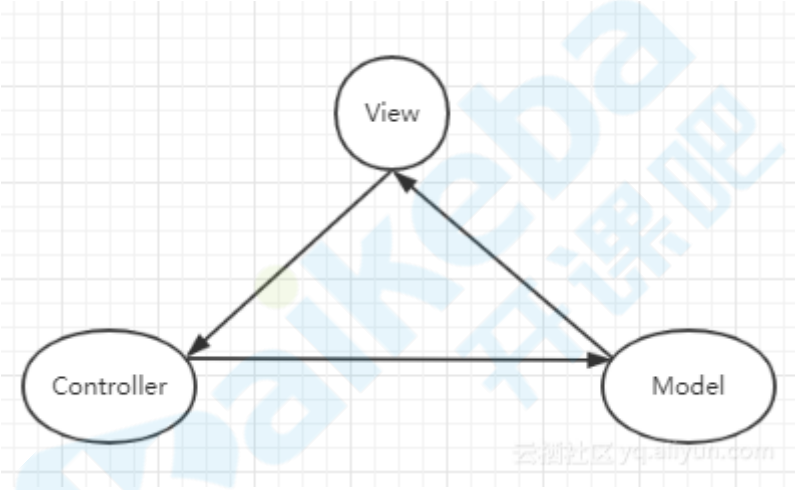
前端的MVC与后端类似，具备着View、Controller和Model。

Model：负责保存应用数据，与后端数据进行同步

Controller：负责业务逻辑，根据用户行为对Model数据进行修改

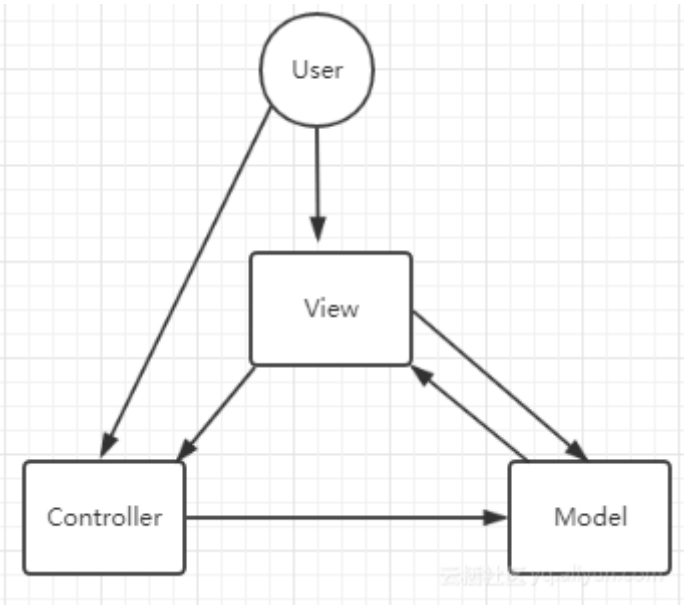
View：负责视图展示，将model中的数据可视化出来。

三者形成了一个如图所示的模型：



这样的模型，在理论上是可行的。但往往在实际开发中，并不会这样操作。因为开发过程并不灵活。例如，一个小小的事件操作，都必须经过这样的一个流程，那么开发就不再便捷了。

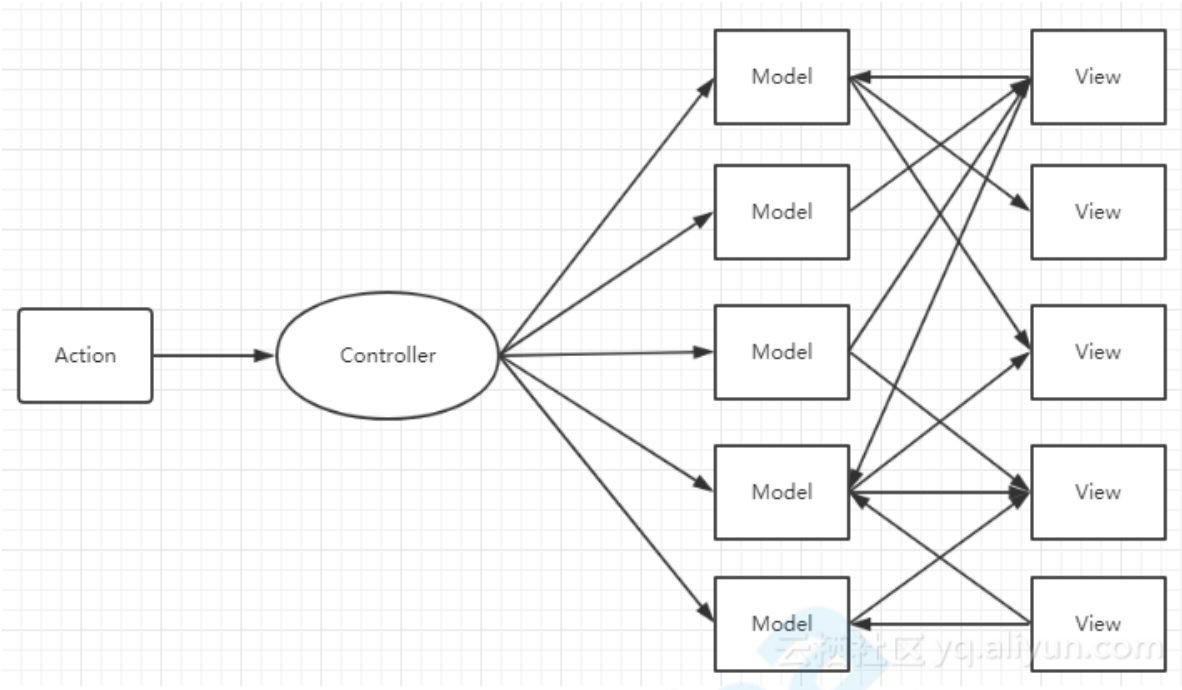
在实际场景中，我们往往会看到另一种模式，如图：



这种模式在开发中更加的灵活，backbone.js框架就是这种的模式。

但是，这种灵活可能导致严重的问题：

1. 数据流混乱。如下图：

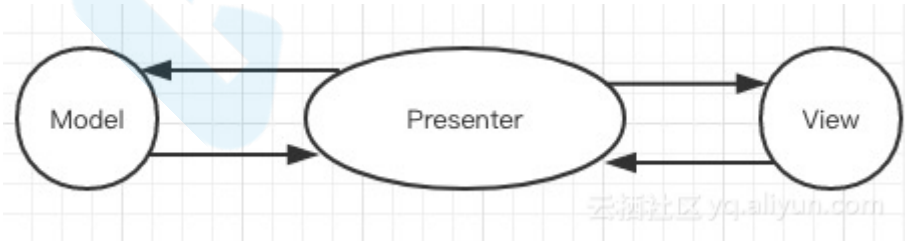


2. View比较庞大，而Controller比较单薄：由于很多的开发者都会在view中写一些逻辑代码，逐渐的就导致view中的内容越来越庞大，而controller变得越来越单薄。

既然有缺陷，就会有变革。前端的变化中，似乎少了MVP的这种模式，是因为AngularJS早早地将MVVM框架模式带入了前端。MVP模式虽然前端开发并不常见，但是在安卓等原生开发中，开发者还是会考虑到它。

MVP

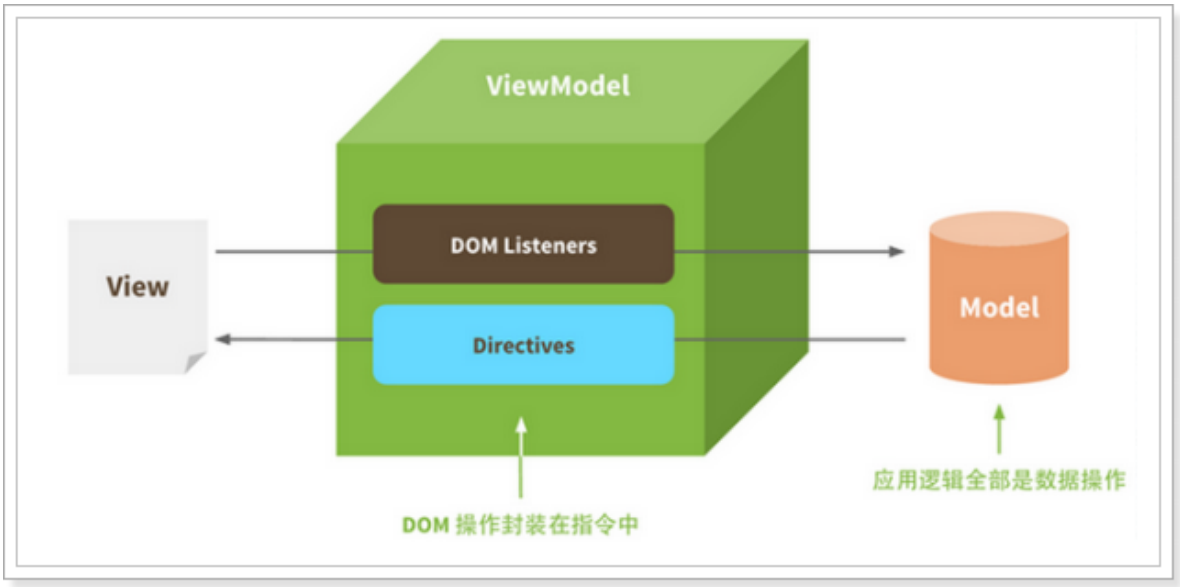
MVP与MVC很接近，P指的是Presenter，presenter可以理解为一个中间人，它负责着View和Model之间的数据流动，防止View和Model之间直接交流。我们可以看一下图示：



我们可以通过看到，presenter负责和Model进行双向交互，还和View进行双向交互。这种交互方式，相对于MVC来说少了一些灵活，VIew变成了被动视图，并且本身变得很小。虽然它分离了View和Model。但是应用逐渐变大之后，导致presenter的体积增大，难以维护。要解决这个问题，或许可以从MVVM的思想中找到答案。

MVVM

首先，何为MVVM呢？MVVM可以分解成(Model-View-VIewModel)。ViewModel可以理解为在presenter基础上的进阶版。如图所示：



ViewModel通过实现一套数据响应式机制自动响应Model中数据变化；

同时Viewmodel会实现一套更新策略自动将数据变化转换为视图更新；

通过事件监听响应View中用户交互修改Model中数据。

这样在ViewModel中就减少了大量DOM操作代码。

MVVM在保持View和Model松耦合的同时，还减少了维护它们关系的代码，使用户专注于业务逻辑，兼顾开发效率和可维护性。

总结

- 这三者都是框架模式，它们设计的目标都是为了解决Model和View的耦合问题。
- MVC模式出现较早主要应用在后端，如Spring MVC、ASP.NET MVC等，在前端领域的早期也有应用，如Backbone.js。它的优点是分层清晰，缺点是数据流混乱，灵活性带来的维护性问题。
- MVP模式在是MVC的进化形式，Presenter作为中间层负责MV通信，解决了两者耦合问题，但P层过于臃肿会导致维护问题。
- MVVM模式在前端领域有广泛应用，它不仅解决MV耦合问题，还同时解决了维护两者映射关系的大量繁杂代码和DOM操作代码，在提高开发效率、可读性同时还保持了优越的性能表现。

8. 你了解哪些Vue性能优化方法?

答题思路：根据题目描述，这里主要探讨Vue代码层面的优化

- 路由懒加载

```
const router = new VueRouter({
  routes: [
    { path: '/foo', component: () => import('./Foo.vue') }
  ]
})
```

- keep-alive缓存页面

```
<template>
  <div id="app">
    <keep-alive>
      <router-view/>
    </keep-alive>
  </div>
</template>
```

- 使用v-show复用DOM

```
<template>
  <div class="cell">
    <!--这种情况用v-show复用DOM，比v-if效果好-->
    <div v-show="value" class="on">
      <Heavy :n="10000"/>
    </div>
    <section v-show="!value" class="off">
      <Heavy :n="10000"/>
    </section>
  </div>
</template>
```

- v-for 遍历避免同时使用 v-if

```
<template>
    <ul>
      <li
        v-for="user in activeUsers"
        :key="user.id">
        {{ user.name }}
      </li>
    </ul>
</template>
<script>
    export default {
        computed: {
          activeUsers: function () {
            return this.users.filter(function (user) {
             return user.isActive
            })
          }
        }
    }
</script>
```

- 长列表性能优化
  - 如果列表是纯粹的数据展示，不会有任何改变，就不需要做响应化

```
export default {
  data: () => ({
    users: []
  }),
  async created() {
    const users = await axios.get("/api/users");
    this.users = Object.freeze(users);
  }
};
```

- 如果是大数据长列表，可采用虚拟滚动，只渲染少部分区域的内容

```
<recycle-scroller
  class="items"
  :items="items"
  :item-size="24"
>
  <template v-slot="{ item }">
    <FetchItemView
      :item="item"
      @vote="voteItem(item)"
    />
  </template>
</recycle-scroller>
```

> 参考vue-virtual-scroller、vue-virtual-scroll-list

- 事件的销毁

  Vue 组件销毁时，会自动解绑它的全部指令及事件监听器，但是仅限于组件本身的事件。

```
created() {
  this.timer = setInterval(this.refresh, 2000)
},
beforeDestroy() {
  clearInterval(this.timer)
}
```

- 图片懒加载

  对于图片过多的页面，为了加速页面加载速度，所以很多时候我们需要将页面内未出现在可视区域内的图片先不做加载， 等到滚动到可视区域后再去加载。

```
<img v-lazy="/static/img/1.png">
```

> 参考项目：vue-lazyload

- 第三方插件按需引入

  像element-ui这样的第三方组件库可以按需引入避免体积太大。

```
import Vue from 'vue';
import { Button, Select } from 'element-ui';

 Vue.use(Button)
 Vue.use(Select)
```

- 无状态的组件标记为函数式组件

```
<template functional>
  <div class="cell">
    <div v-if="props.value" class="on"></div>
    <section v-else class="off"></section>
  </div>
</template>

<script>
export default {
  props: ['value']
}
</script>
```

- 子组件分割

```
<template>
  <div>
    <ChildComp/>
  </div>
</template>

<script>
export default {
  components: {
    ChildComp: {
      methods: {
        heavy () { /* 耗时任务 */ }
      },
      render (h) {
        return h('div', this.heavy())
      }
    }
  }
}
</script>
```

- 变量本地化

```
<template>
  <div :style="{ opacity: start / 300 }">
    {{ result }}
  </div>
</template>
```

```
<script>
import { heavy } from '@/utils'

export default {
  props: ['start'],
  computed: {
    base () { return 42 },
    result () {
      const base = this.base // 不要频繁引用this.base
      let result = this.start
      for (let i = 0; i < 1000; i++) {
        result += heavy(base)
      }
      return result
    }
  }
}
</script>
```

- SSR

9. 你对Vue3.0的新特性有没有了解?

根据尤大的PPT总结，Vue3.0改进主要在以下几点：

- 更快
  - 虚拟DOM重写
  - 优化slots的生成
  - 静态树提升
  - 静态属性提升
  - 基于Proxy的响应式系统
- 更小：通过摇树优化核心库体积
- 更容易维护：TypeScript + 模块化
- 更加友好
  - 跨平台：编译器核心和运行时核心与平台无关，使得Vue更容易与任何平台（Web、Android、iOS）一起使用
- 更容易使用
  - 改进的TypeScript支持，编辑器能提供强有力的类型检查和错误及警告
  - 更好的调试支持
  - 独立的响应化模块
  - Composition API

虚拟 DOM 重写

期待更多的编译时提示来减少运行时开销，使用更有效的代码来创建虚拟节点。

组件快速路径+单个调用+子节点类型检测

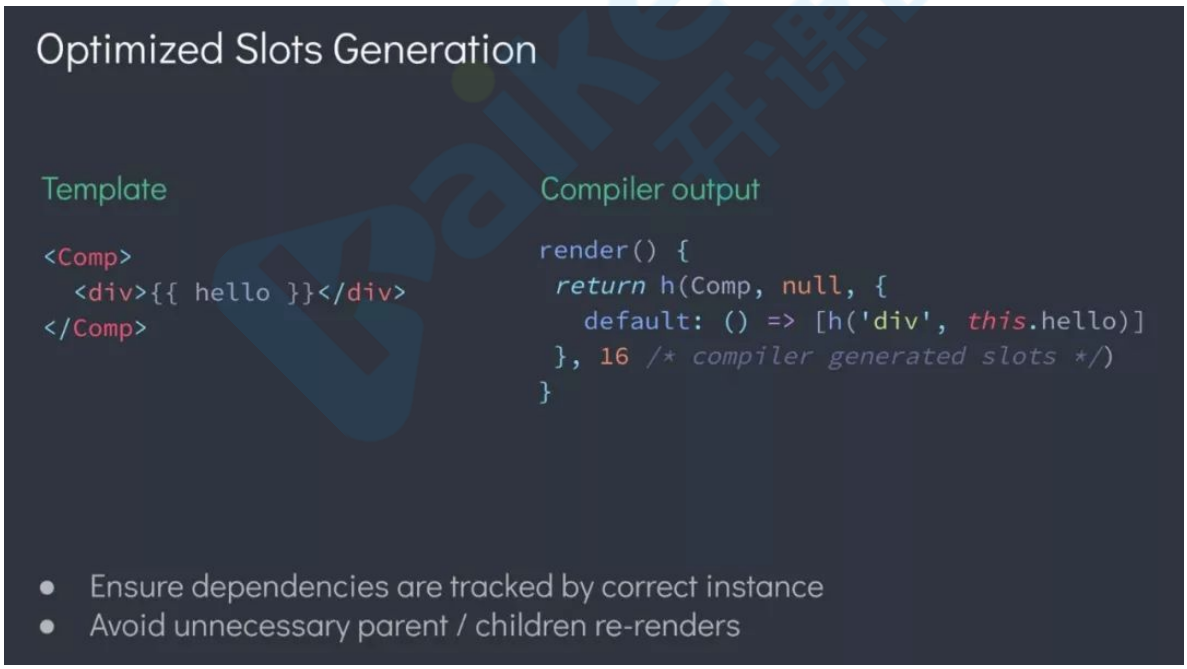
- 跳过不必要的条件分支

- JS引擎更容易优化

优化slots生成

vue3中可以单独重新渲染父级和子级

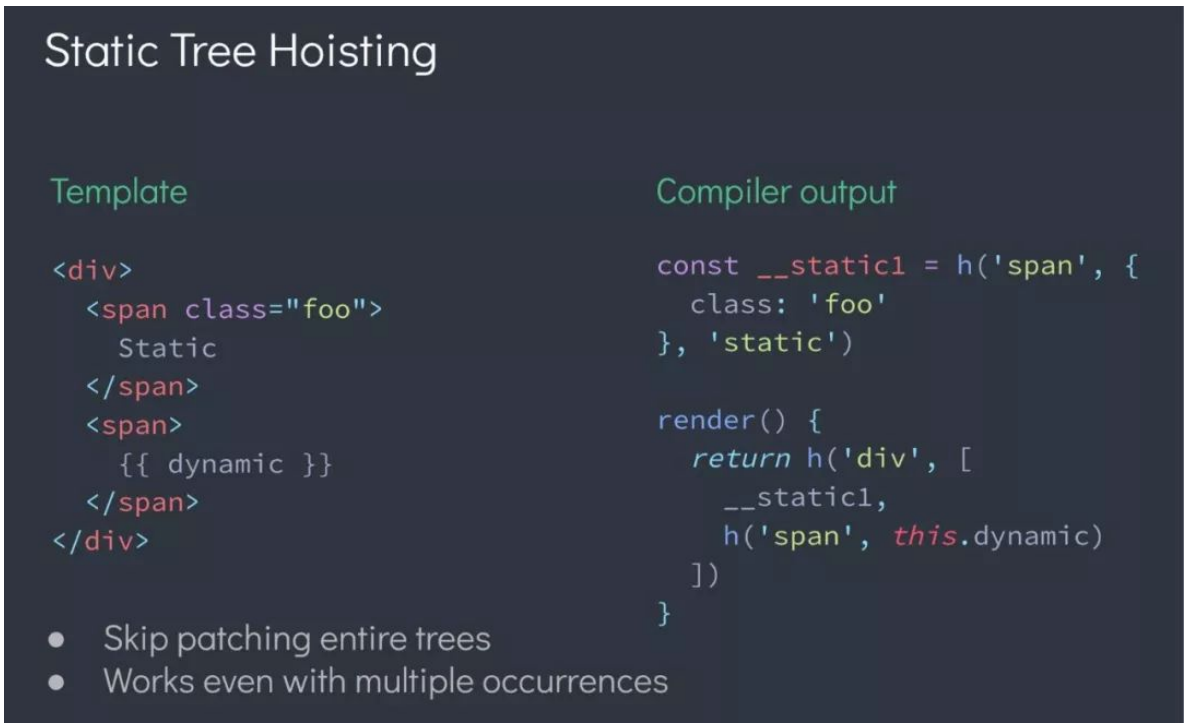
- 确保实例正确的跟踪依赖关系
- 避免不必要的父子组件重新渲染



静态树提升(Static Tree Hoisting)

使用静态树提升，这意味着 Vue 3 的编译器将能够检测到什么是静态的，然后将其提升，从而降低了渲染成本。

- 跳过修补整棵树，从而降低渲染成本
- 即使多次出现也能正常工作



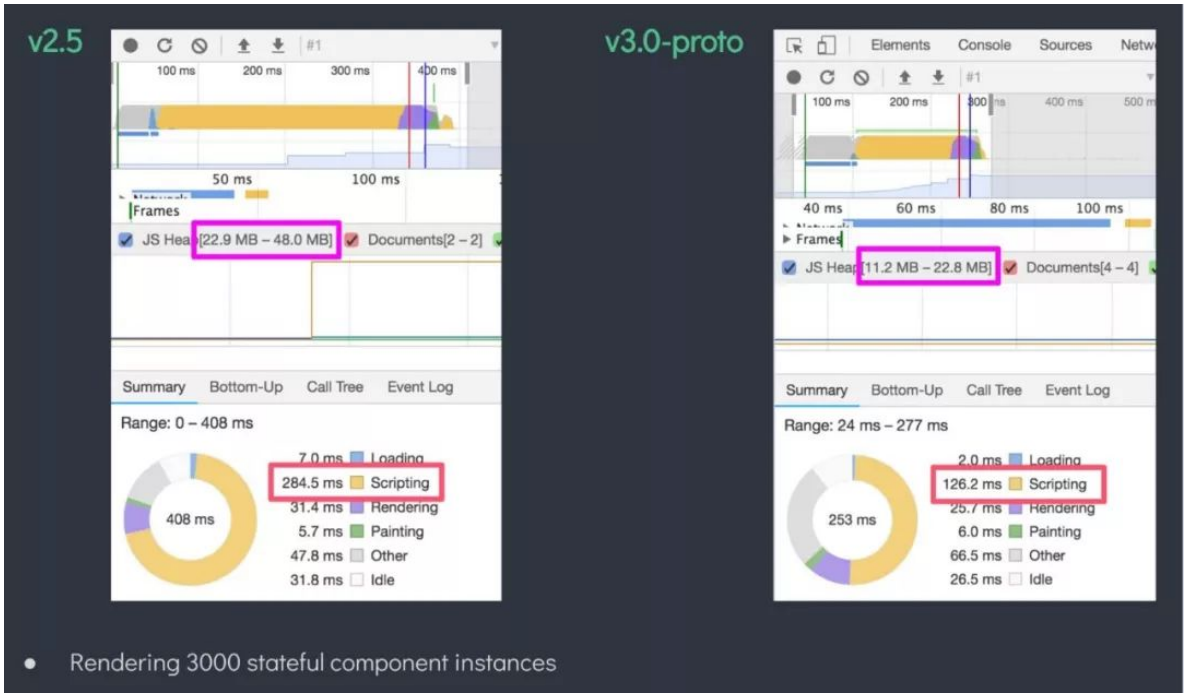
静态属性提升

使用静态属性提升，Vue 3打补丁时将跳过这些属性不会改变的节点。

基于 Proxy 的数据响应式

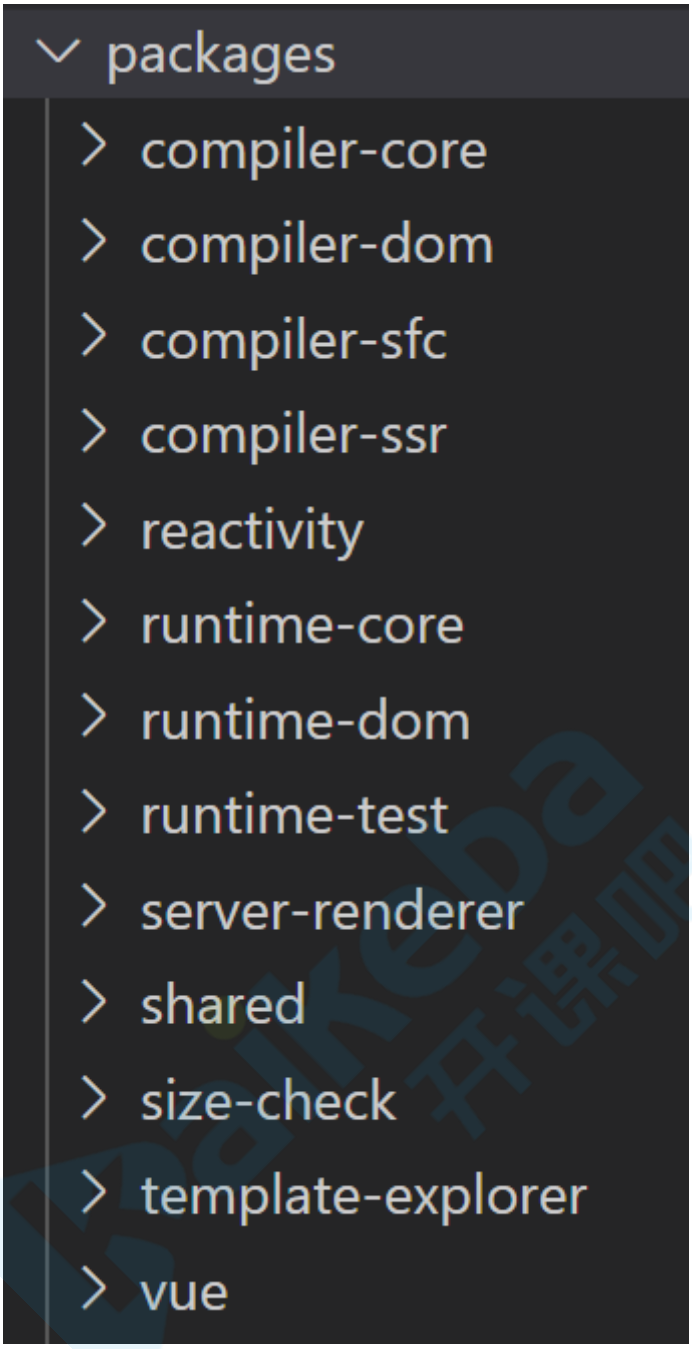
Vue 2的响应式系统使用 Object.defineProperty 的getter 和 setter。Vue 3 将使用 ES2015 Proxy 作为其观察机制，这将会带来如下变化：

- 组件实例初始化的速度提高100%
- 使用Proxy节省以前一半的内存开销，加快速度，但是存在低浏览器版本的不兼容
- 为了继续支持 IE11，Vue 3 将发布一个支持旧观察者机制和新 Proxy 版本的构建



高可维护性

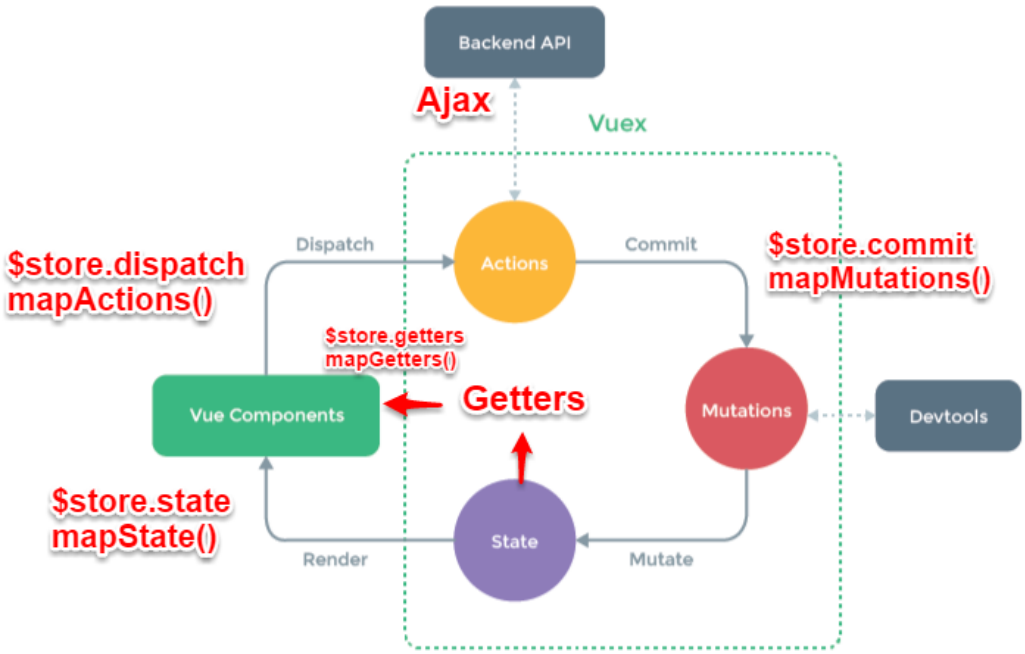
Vue 3 将带来更可维护的源代码。它不仅会使用 TypeScript，而且许多包被解耦，更加模块化。



10. 简单说一说vuex使用及其理解?

- vue中状态管理（登陆验证，购物车，播放器等）

`vuex`数据流程



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`state`中唯一的方法是提交`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) {}
    }
  }
})
```

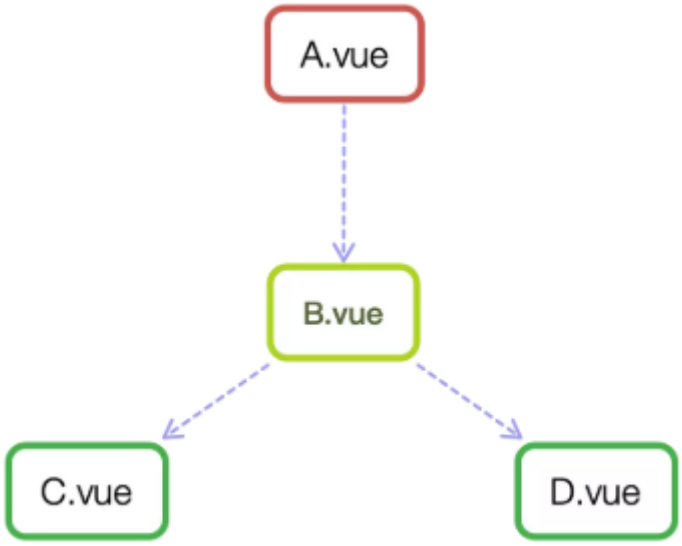
注意：`vuex`里保存的状态，都是数组，而`localStorage`只支持字符串。

总结：

- 首先说明Vuex 是一个专为 Vue.js 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状态以一种可预测的方式发生变化。
- vuex核心概念 重点同步异步实现　action mutation
- vuex中做数据存储 --------- local storage
- 如何选用vuex

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

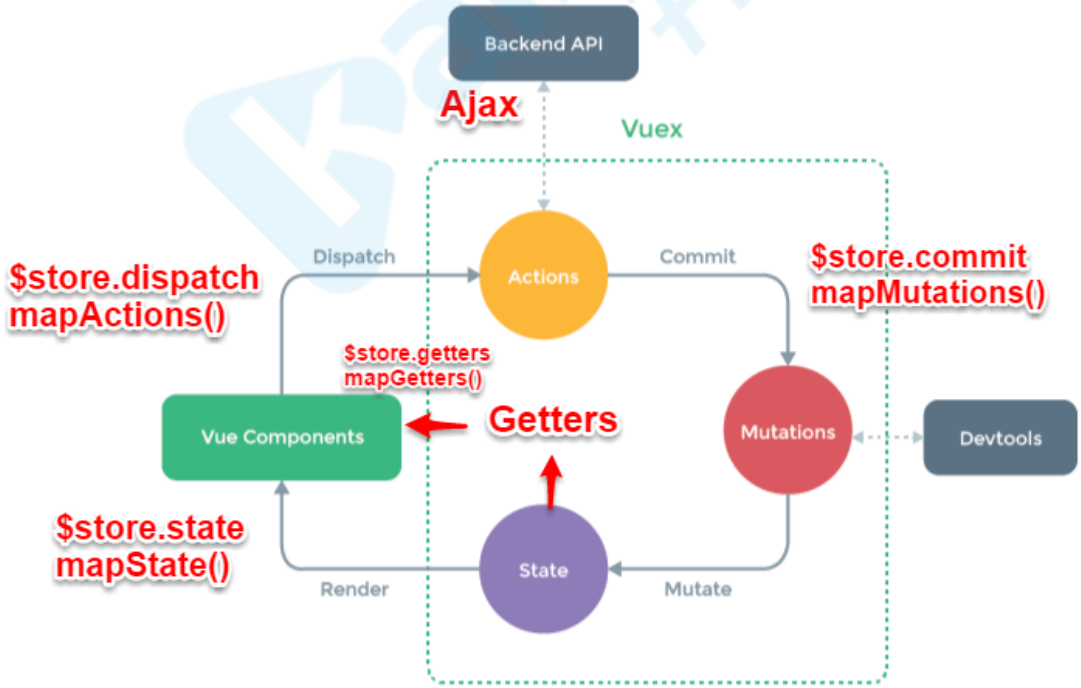
```
import Vue from 'vue'

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

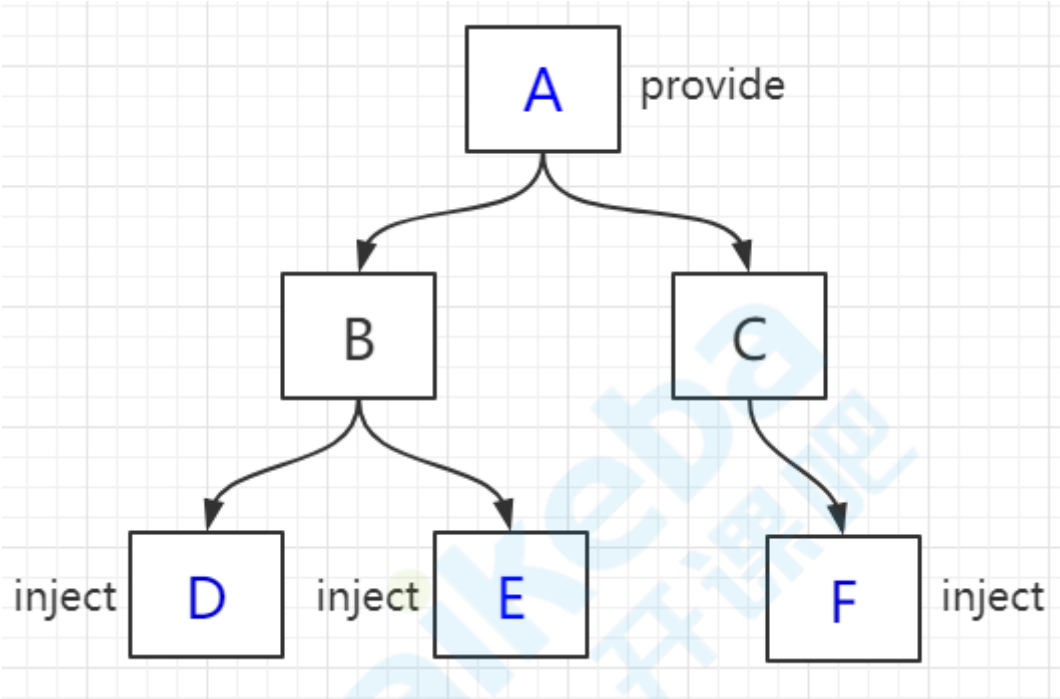
- provide祖先组件的实例，然后在子孙组件中注入依赖，这样就可以在子孙组件中直接修改祖先组件的实例的属性，不过这种方法有个缺点就是这个实例上挂载很多没有必要的东西比如props，methods

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. 什么是递归组件?

概念：组件是可以在它们自己的模板中调用自身的。

递归组件，一定要有一个结束的条件，否则就会使组件循环引用，最终出现的错误，我们可以使用v-if="false"作为递归组件的结束条件。当遇到v-if为false时，组件将不会再进行渲染。

既然要用递归组件，那么对我们的数据格式肯定是需要满足递归的条件的。就像下边这样，这是一个树状的递归数据。

```
// tree组件数据
list:[
     {
         "name": "web全栈工程师",
         cList: [
             {"name": "vue" },
             {
                 "name": "react",
                 cList: [
                     {
                         "name": 'javascrict',
                         cList: [{ "name": "css"}]
                     }
                 ]
              }
            ]
      },
      { "name": "web高级工程师" },
      {
        "name": "web初级工程师",
          cList: [
              { "name": "javascript" },
              { "name": "css" }
          ]
      },
 ]
```

案例：

```
<template>
  <div>
    <ul>
    <li v-for="(item,index) in list " :key="index">
      <p>{{item.name}}</p>
    <tree-muen :list='list'/>
    </li>
  </ul>
  </div>
</template>
<script>
export default {
  components: {
    treeMuen
  },
  data () {
    return {
      list: [
        {
         "name": "web全栈工程师",
         cList: [
             {"name": "vue" },
             {
                 "name": "react",
                 cList: [
                     {
                         "name": 'javascrict',
                         cList: [{ "name": "css“ }]
                     }
                 ]
              }
            ]
      },
      { "name": "web高级工程师" },
      {
        "name": "web初级工程师",
          cList: [
              { "name": "javascript" },
              { "name": "css" }
          ]
      },
  ],
   methods: {}

}
</script>
```

总结：

> 通过props从父组件拿到数据，递归组件每次进行递归的时候都会tree-menus组件传递下一级treeList数据，整个过程结束之后，递归也就完成了，对于折叠树状菜单来说，我们一般只会去渲染一级的数据，当点击一级菜单时，再去渲染一级菜单下的结构，如此往复。那么v-if就可以实现我们的这个需求，当v-if设置为false时，递归组件将不会再进行渲染，设置为true时，继续渲染。

总结：

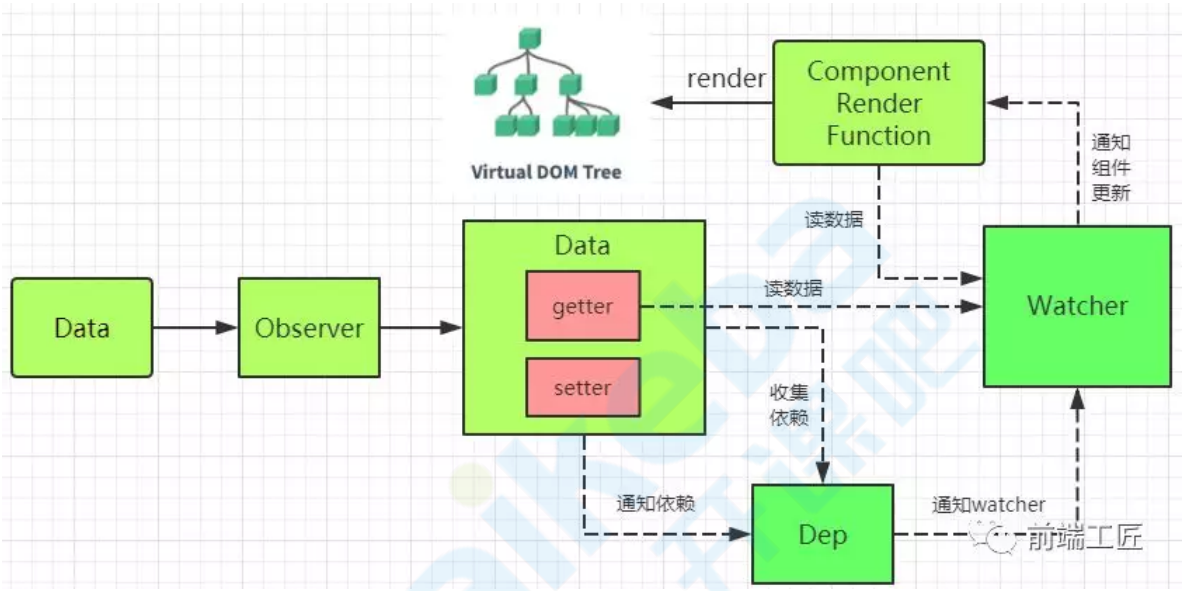
简述递归组件的实现和概念，可以封装成什么样的组件（收缩框）

使用它有什么优点

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

16. vue为什么要求组件模版只能有一个根元素?

从三方面考虑：

1. new Vue({el:'#app'})
2. 单文件组件中，template下的元素div。其实就是"树"状数据结构中的"根"。
3. diff算法要求的，源码中，patch.js里patchVnode()。

> 一

实例化Vue时：

```
<body>
    <div id='app'></div>
</body>
<script>
    var vm = new Vue({
        el:'#app'
    })
</script>
```

如果我在body下这样：

```
<body>
    <div id='app1'></div>
    <div id='app2'></div>
</body>
```

Vue其实并不知道哪一个才是我们的入口。如果同时设置了多个入口，那么vue就不知道哪一个才是这个'类'。

> 二

在webpack搭建的vue开发环境下，使用单文件组件时：

```
<template>
    <div>

  </div>
</template>
```

template这个标签，它有三个特性：

1. 隐藏性：该标签不会显示在页面的任何地方，即便里面有多少内容，它永远都是隐藏的状态，设置了display：none；
2. 任意性：该标签可以写在任何地方，甚至是head、body、sciprt标签内；
3. 无效性：该标签里的任何HTML内容都是无效的，不会起任何作用；只能innerHTML来获取到里面的内容。

一个vue单文件组件就是一个vue实例，如果template下有多个div那么如何指定vue实例的根入口呢，为了让组件可以正常生成一个vue实例，这个div会自然的处理成程序的入口，通过这个根节点，来递归遍历整个vue树下的所有节点，并处理为vdom，最后再渲染成真正的HTML，插入在正确的位置。

> 三

diff中patchvnode方法，用来比较新旧节点，我们一起来看下源码。

总结：

1. new Vue({el:'#app'})
2. 单文件组件中，template下的元素div。其实就是"树"状数据结构中的"根"。
3. diff算法要求的，源码中，patch.js里patchVnode()。

17. watch和computed的区别以及怎么选用?

区别

1. 定义/语义区别

> watch

```
<input type="text" v-model="foo" />
```

```
var vm = new Vue({
  el: '#demo',
  data: {
    foo: 1
  },
  watch: {
    foo: function (newVal,oldVal) {
      console.log(newVal+''+oldVal)
    }
  }
})
vm.foo = 2 // 2 1
```

> computed

```
var vm = new Vue({
  el: '#demo',
  data: {
    firstName: 'Foo',
    lastName: 'Bar'
  },
  computed: {
    fullName: function () {
      return this.firstName + ' ' + this.lastName
    }
  }
})
vm.fullName //Foo Bar  computed内部的函数调用的时候不需要加()
```

2. 功能区别

watch更通用，computed派生功能都能实现，计算属性底层来自于watch，但做了更多，例如缓存

3. 用法区别

computed更简单/更高效，优先使用

有些必须watch，比如值变化要和后端交互

使用场景

> watch

watch需要在数据变化时执行异步或开销较大的操作时使用，简单讲，当一条数据影响多条数据的时候，例如 搜索数据

> computed

对于任何复杂逻辑或一个数据属性在它所依赖的属性发生变化时，也要发生变化，简单讲。当一个属性受多个属性影响的时候，例如 购物车商品结算时

18. 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?

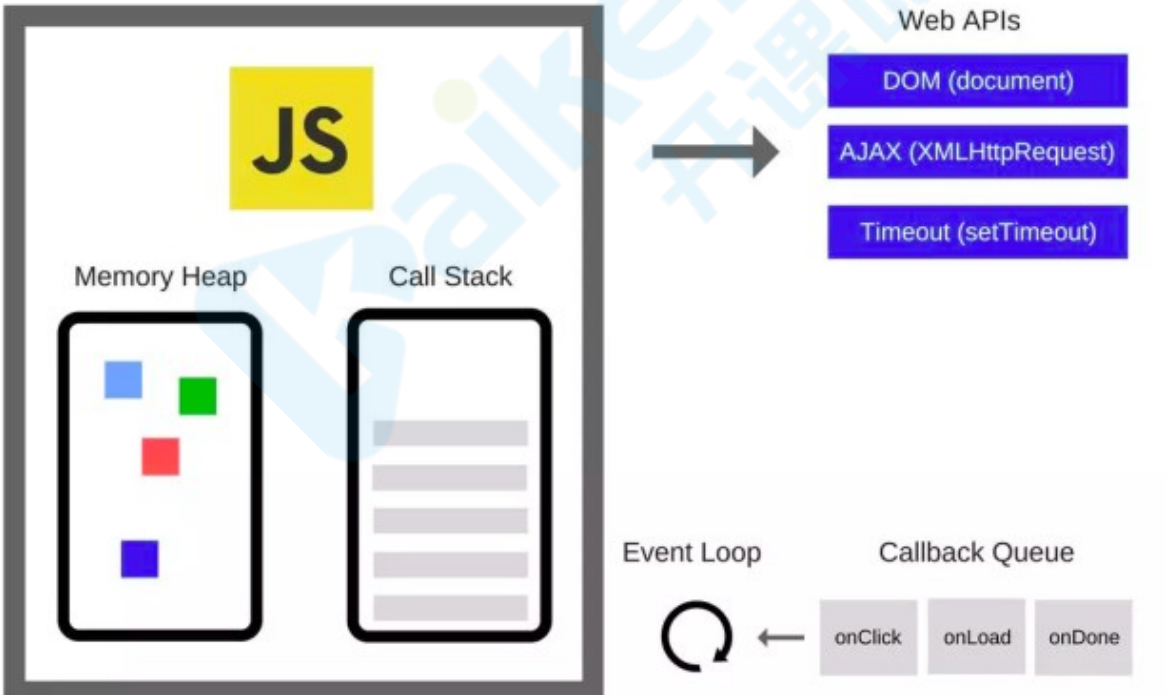
vue的源码中实现nextTick的地方：

```
//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

19. 你知道vue双向数据绑定的原理吗?

> 什么是双向数据绑定?

数据变化更新视图，视图变化更新数据

输入框内容变化时，data中的数据同步变化 view => model

data中的数据变化时，文本节点的内容同步变化 model => view

> 设计思想：观察者模式

Vue的双向数据绑定的设计思想为观察者模式。

Dep对象：Dependency依赖的简写，包含有三个主要属性id, subs, target和四个主要函数addSub, removeSub, depend, notify，是观察者的依赖集合，负责在数据发生改变时，使用notify()触发保存在subs下的订阅列表，依次更新数据和DOM。

Observer对象：即观察者，包含两个主要属性value, dep。做法是使用getter/setter方法覆盖默认的取值和赋值操作，将对象封装为响应式对象，每一次调用时更新依赖列表，更新值时触发订阅者。绑定在对象的_ *ob* _原型链属性上。

```
new Vue({
 el: '#app',
 data: {
  count: 100
 },
 ...
});
```

初始化函数initMixin：

```
Vue.prototype._init = function (options) {
 ...
 var vm = this;
 ...
 initLifecycle(vm);
 initEvents(vm);
 initRender(vm);
 callHook(vm, 'beforeCreate');
```

```
// 这里就是我们接下来要跟进的初始化Vue参数
initState(vm);
initInjections(vm);
callHook(vm, 'created');
...
};
```

初始化参数 initState：

```
function initState (vm) {
vm._watchers = [];
var opts = vm.$options;
if (opts.props) { initProps(vm, opts.props); }
if (opts.methods) { initMethods(vm, opts.methods); }
// 我们的count在这里初始化
if (opts.data) {
initData(vm);
} else {
observe(vm._data = {}, true /* asRootData */);
}
if (opts.computed) { initComputed(vm, opts.computed); }
if (opts.watch) { initWatch(vm, opts.watch); }
}
```

initData：

```
function initData (vm) {
var data = vm.$options.data;
data = vm._data = typeof data === 'function'
? data.call(vm)
: data || {};
if (!isPlainObject(data)) {
data = {};
}
...
// observe data
observe(data, true /* asRootData */);
```

将data参数设置为响应式：

```
/**
* Attempt to create an observer instance for a value,
* returns the new observer if successfully observed,
* or the existing observer if the value already has one.
*/
function observe (value, asRootData) {
if (!isObject(value)) {
return
}
var ob;
if (hasOwn(value, '__ob__') && value.__ob__ instanceof Observer) {
ob = value.__ob__;
} else if (
/* 为了防止value不是单纯的对象而是Regexp或者函数之类的，或者是vm实例再或者是不可扩展的 */
observerState.shouldConvert &&
!isServerRendering() &&
```

```
 (Array.isArray(value) || isPlainObject(value)) &&
 Object.isExtensible(value) &&
 !value._isVue
 ) {
 ob = new Observer(value);
 }
 if (asRootData && ob) {
 ob.vmCount++;
 }
 return ob
}
```

Observer类:

```
/**
 * Observer class that are attached to each observed
 * object. Once attached, the observer converts target
 * object's property keys into getter/setters that
 * collect dependencies and dispatches updates.
 */
var Observer = function Observer (value) {
 this.value = value;
 this.dep = new Dep();
 this.vmCount = 0;
 // def函数是defineProperty的简单封装
 def(value, '__ob__', this);
 if (Array.isArray(value)) {
 // 在es5及更低版本的js里，无法完美继承数组，这里检测并选取合适的函数
 // protoAugment函数使用原型链继承，copyAugment函数使用原型链定义（即对每个数组
defineProperty）
 var augment = hasProto
  ? protoAugment
  : copyAugment;
 augment(value, arrayMethods, arrayKeys);
 this.observeArray(value);
 } else {
 this.walk(value);
 }
};
```

observerArray:

```
/**
 * Observe a list of Array items.
 */
Observer.prototype.observeArray = function observeArray (items) {
 for (var i = 0, l = items.length; i < l; i++) {
 observe(items[i]);
 }
};
```

Dep类:

```
/**
 * A dep is an observable that can have multiple
 * directives subscribing to it.
 */
var Dep = function Dep () {
 this.id = uid$1++;
 this.subs = [];
};
```

walk函数：

```
/**
 * Walk through each property and convert them into
 * getter/setters. This method should only be called when
 * value type is Object.
 */
Observer.prototype.walk = function walk (obj) {
 var keys = Object.keys(obj);
 for (var i = 0; i < keys.length; i++) {
 defineReactive$$1(obj, keys[i], obj[keys[i]]);
 }
};
```

defineReactive：

```
/**
 * Define a reactive property on an Object.
 */
function defineReactive$$1 (obj, key, val, customSetter) {
 var dep = new Dep();

 var property = Object.getOwnPropertyDescriptor(obj, key);
 if (property && property.configurable === false) {
 return
 }

 // cater for pre-defined getter/setters
 var getter = property && property.get;
 var setter = property && property.set;

 var childOb = observe(val);
 Object.defineProperty(obj, key, {
 enumerable: true,
 configurable: true,
 get: function reactiveGetter () {
  var value = getter ? getter.call(obj) : val;
  if (Dep.target) {
  dep.depend();
  if (childOb) {
   childOb.dep.depend();
  }
  if (Array.isArray(value)) {
   dependArray(value);
  }
  }
  return value
```

```
 },
 set: function reactiveSetter (newVal) {
  var value = getter ? getter.call(obj) : val;
  // 脏检查，排除了NaN !== NaN的影响
  if (newVal === value || (newVal !== newVal && value !== value)) {
  return
  }
  if (setter) {
  setter.call(obj, newVal);
  } else {
  val = newVal;
  }
  childOb = observe(newVal);
  dep.notify();
 }
 });
}
```

Dep.target&depend():

```
// the current target watcher being evaluated.
// this is globally unique because there could be only one
// watcher being evaluated at any time.
Dep.target = null;

Dep.prototype.depend = function depend () {
 if (Dep.target) {
 Dep.target.addDep(this);
 }
};

Dep.prototype.notify = function notify () {
 // stablize the subscriber list first
 var subs = this.subs.slice();
 for (var i = 0, l = subs.length; i < l; i++) {
 subs[i].update();
 }
};
```

addDep():

```
/**
 * Add a dependency to this directive.
 */
Watcher.prototype.addDep = function addDep (dep) {
 var id = dep.id;
 if (!this.newDepIds.has(id)) {
 this.newDepIds.add(id);
 this.newDeps.push(dep);
 if (!this.depIds.has(id)) {
  // 使用push()方法添加一个订阅者
  dep.addSub(this);
 }
 }
};
```

dependArray():

```
/**
 * Collect dependencies on array elements when the array is touched, since
 * we cannot intercept array element access like property getters.
 */
function dependArray (value) {
 for (var e = (void 0), i = 0, l = value.length; i < l; i++) {
 e = value[i];
 e && e.__ob__ && e.__ob__.dep.depend();
 if (Array.isArray(e)) {
  dependArray(e);
 }
 }
}
```

数组的更新检测：

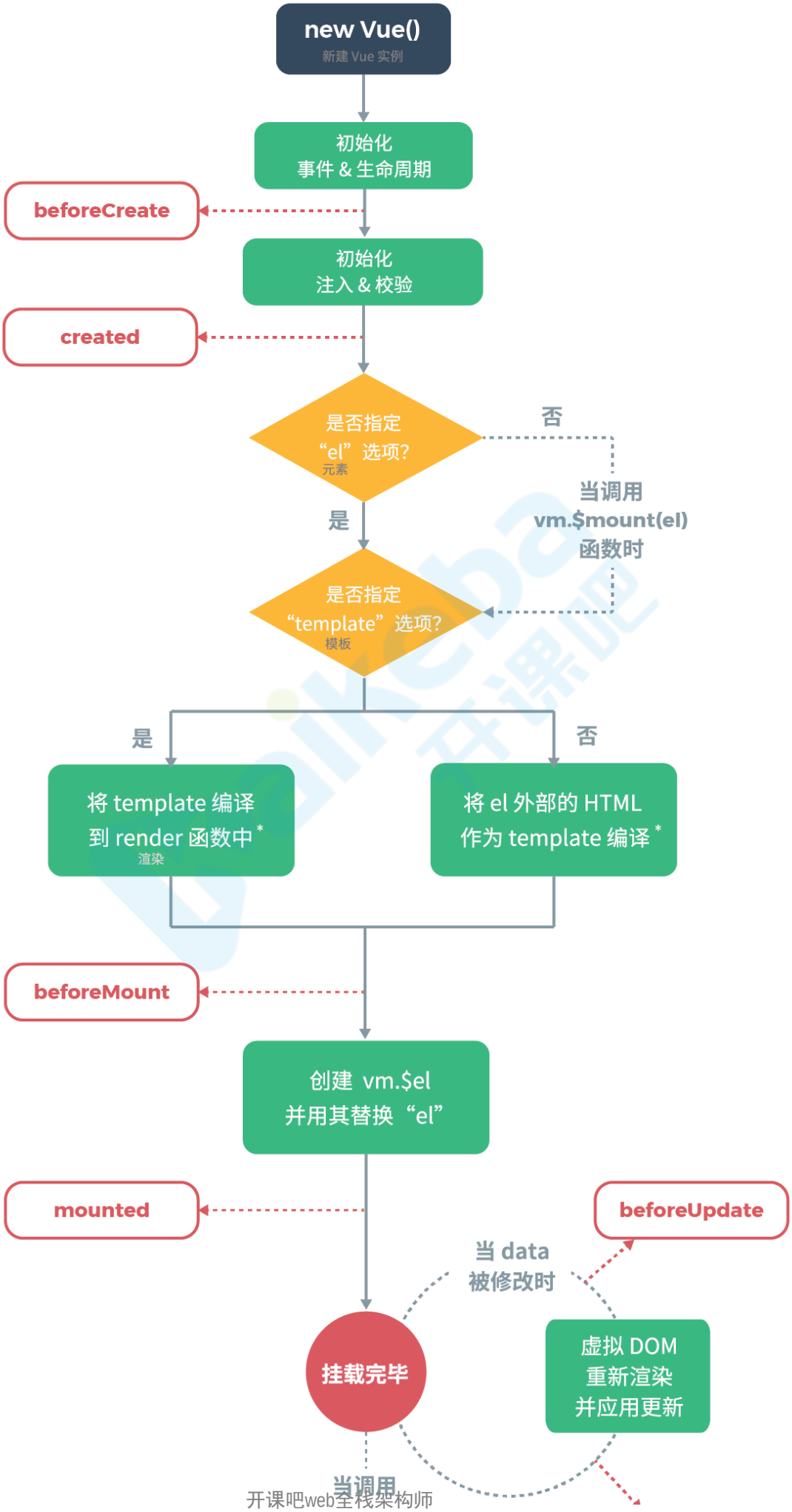
```
/*
 * not type checking this file because flow doesn't play well with
 * dynamically accessing methods on Array prototype
 */
var arrayProto = Array.prototype;
var arrayMethods = Object.create(arrayProto);
['push', 'pop', 'shift', 'unshift', 'splice', 'sort',
'reverse'].forEach(function (method) {
 // cache original method
 var original = arrayProto[method];
 def(arrayMethods, method, function mutator () {
 var arguments$1 = arguments;
 // avoid leaking arguments:
 // http://jsperf.com/closure-with-arguments
 var i = arguments.length;
 var args = new Array(i);
 while (i--) {
  args[i] = arguments$1[i];
 }
 var result = original.apply(this, args);
 var ob = this.__ob__;
 var inserted;
 switch (method) {
  case 'push':
  inserted = args;
  break
  case 'unshift':
  inserted = args;
  break
  case 'splice':
  inserted = args.slice(2);
  break
 }
 if (inserted) { ob.observeArray(inserted); }
 // notify change
 ob.dep.notify();
 return result
 });
});
```

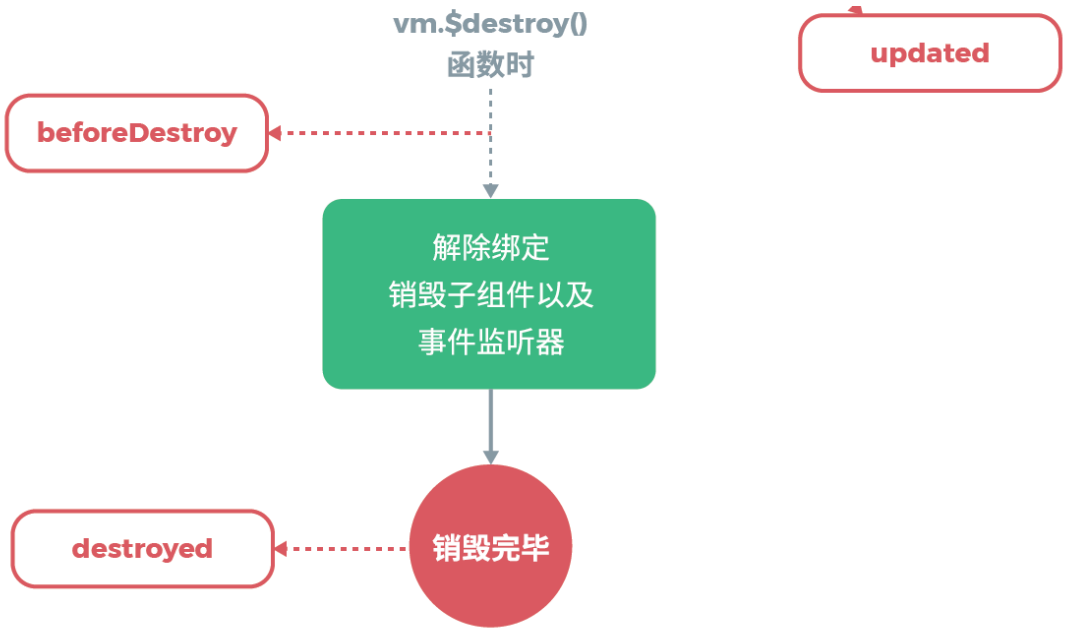
总结：

从上面的代码中我们可以一步步由深到浅的看到Vue是如何设计出双向数据绑定的，最主要的两点：

- 使用getter/setter代理值的读取和赋值，使得我们可以控制数据的流向。
- 使用观察者模式设计，实现了指令和数据的依赖关系以及触发更新。

20. 简单说一说vue生命周期的理解?


new Vue()
新建 Vue 实例
初始化
事件 & 生命周期
beforeCreate
初始化
注入 & 校验
created
是否指定
“el”选项?
元素
否
当调用
vm.$mount(el)
函数时
是
是否指定
“template”选项?
模板
是
否
将 template 编译
到 render 函数中*
渲染
将 el 外部的 HTML
作为 template 编译*
beforeMount
创建 vm.$el
并用其替换“el”
mounted
beforeUpdate
当 data
被修改时
挂载完毕
虚拟 DOM
重新渲染
并应用更新
当调用



* 如果使用构造生成文件（例如构造单文件组件），
模板编译将提前执行

下面我们从源码方面详细解释一下这张图

1. 实例化

> 显而易见，这个就是实例化。实例化之后，会执行以下操作。根据 Vue的源码，我们可以看到Vue的本质就是一个 `function`，`new Vue` 的过程就是初始化**参数**、**生命周期**、**事件**等一系列过程

源码路径: src/core/instance/index.js

2. 初始化事件 生命周期函数

> 首先就是初始化事件和生命周期函数。这时候，这个对象身上只有默认一些生命周期函数和默认的事件，其他的东西都未创建。

3. beforeCreate创建前

> 接着就是`beforeCreate（创建前）`执行。但是这个时候拿不到`data`里边的数据。data和methods中的数据都还没初始化。

4. 注射响应

> `injections(注射器) reactivity(响应)`给数据添加观察者。

5. created创建后

> `created创建后`执行。因为上边给数据添加了观察者，所以现在就可以访问到`data`里的数据了。这个钩子也是常用的，可以请求数据了。如果要调用methods中的方法或者操作data中数据，要在created里操作。要因为请求数据是异步的，所以发送请求宜早不宜迟，通常在这个时候发送请求。

源码路径: src/core/instance/init.js initMixin

6. 是否存在 el

> `el`指明挂载目标。这个步骤就是判断一下是否有写`el`，如果没有就判断有没有调用实例上的`$mount('')`方法调用。也就是下一张图。

- 这两个是等价的。

源码路径: src/core/instance/init.js

7. 判断是否有 template

> 判断是否有 `template`。

- 如果有 `template` 则渲染 `template` 里的内容。

- 如果没有 则渲染 `el` 指明的挂载对象里的内容。

源码路径: src/platforms/web/entry-runtime-with-compiler.js $mount

8. beforeMount挂载前

> `beforeMount挂载前` 执行。

9. 替换 el

> 这个时候会在实例上创建一个 `el` ，替换掉原来的 `el` 。也是真正的挂载。

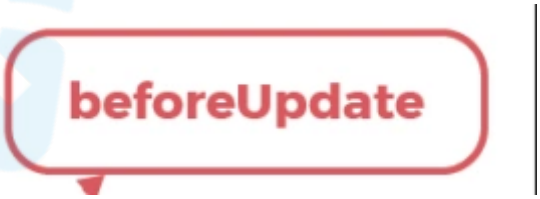
10. mounted挂载后

> `mounted挂载后` 执行。这个时候 `DOM` 已经加载完成了，可以操作 `DOM` 了。只要执行完了mounted，就表示整个vue实例已经初始化完毕了。这个也是常用的钩子。一般操作 `DOM` 都是在这里。

源码路径: src/platforms/web/runtime/index.js $mount -> src/core/instance/lifecycle.js mountComponent

11. dataChange

> 当数据有变动时，会触发下面这两个钩子。



源码路径: src/core/instance/lifecycle.js

- 在 `beforeUpdate更新前` 和 `updated更新后` 之间会进行 `DOM` 的重新渲染和补全。

- 接着是 `updated` 更新后

源码路径: src/core/observer/scheduler.js callUpdatedHooks

11. callDestroy

- `beforeDestroy销毁前` 和 `destroy销毁后` 这两个钩子是需要我们手动调用实例上的 `$destroy` 方法才会触发的。
- 当 `$destroy` 方法调用后。
- `beforeDestroy` 销毁前触发

- 移除数据劫持、事件监听、子组件属性 所有的东西还保留 只是不能修改

- `destroy` 销毁后触发

源码路径: src/core/instance/lifecycle.js lifecycleMixin $destroy

新增钩子

- activated：keep-alive 组件激活时调用。
  类似 created 没有真正创建，只是激活
- deactivated：keep-alive 组件停用时调用。
  类似 destroyed 没有真正移除，只是禁用
- 在 2.2.0 及其更高版本中，activated 和 deactivated 将会在 树内的所有嵌套组件中触发。

顺序验证代码如下

```
<body>
    <div id="app">
        {{msg}}
    </div>
    <script>
        let vm = new Vue({
            el:"#app", // 指明 VUE实例 的挂载目标 （只在 new 创建的实例中遵守）
            data:{msg:"kaikeba"},
            beforeCreate() {
               console.log('创建前')
            },
            created() {
               console.log('创建后')
            },
            beforeMount() {
               console.log('挂载前')
            },
            mounted() {
               console.log('挂载后')
            },
            beforeUpdate() {
                alert('更新前')
            },
            updated() {
                alert('更新后')
            },
            beforeDestroy() {
                alert('销毁前')
            },
            destroyed() {
                alert('销毁后')
            },
        })
        option
        // vm.$mount('#app') // 等价于 el:"#app"
        vm.$destroy()

        // init events, init lifecycle 初始事件，初始化生命周期钩子函数
        // init injections （注射器） reactivity （响应） 给数据添加观察者
        // Compile el's outerHTML as template 编译el的outerHTML作为模板
        // 在beforeMount mounted 之间 create vm.$el and replace “el” with it 会创建一个 el 代替自己的el对象
        // virtual DOM re-render and patch 虚拟DOM重新渲染和修补
        // when vm.$destroy() is called 当销毁函数vm.$destroy()调用时 才会调用销毁前后的生命周期
        // teardown watchs child components and event listeners 移除数据劫持、事件监听、子组件属性 所有的东西还保留 只是不能修改
    </script>
</body>
```

总结：

1.beforeCreate：在实例初始化之后，数据观测（data observe）和event/watcher事件配置之前被调用，这时无法访问data及props等数据；

2.created：在实例创建完成后被立即调用，此时实例已完成数据观测（data observer），属性和方法的运算，watch/event事件回调，挂载阶段还没开始，`$el`尚不可用。

3.beforemount:在挂载开始之前被调用，相关的render函数首次被调用。

4.mounted：实例被挂载后调用，这时el被新创建的vm.`$el`替换，若根实例挂载到了文档上的元素上，当mounted被调用时vm.$el也在文档内。注意mounted不会保证所有子组件一起挂载。

5.beforeupdata：数据更新时调用，发生在虚拟dom打补丁前，这时适合在更新前访问现有dom，如手动移除已添加的事件监听器。

6.updated：在数据变更导致的虚拟dom重新渲染和打补丁后，调用该钩子。当这个钩子被调用时，组件dom已更新，可执行依赖于dom的操作。多数情况下应在此期间更改状态。 如需改变，最好使用watcher或计算属性取代。注意updated不会保证所有的子组件都能一起被重绘。

7.beforedestory：在实例销毁之前调用。在这时，实例仍可用。

8.destroyed：实例销毁后调用，这时vue实例的所有指令都被解绑，所有事件监听器被移除，所有子实例也被销毁。