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。

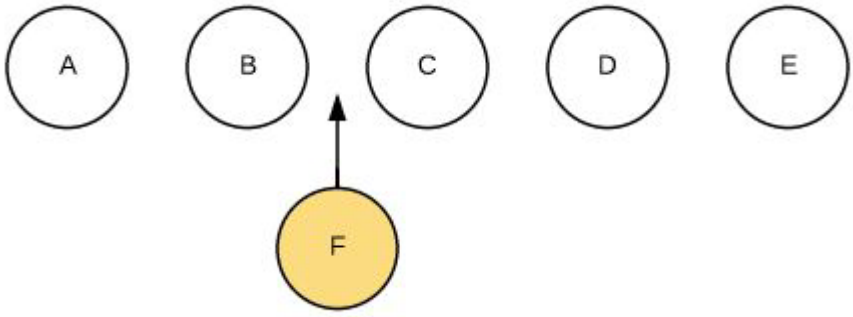
3. 你知道 vue 中 key 的作用和工作原理吗？说说你对它的理解。

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

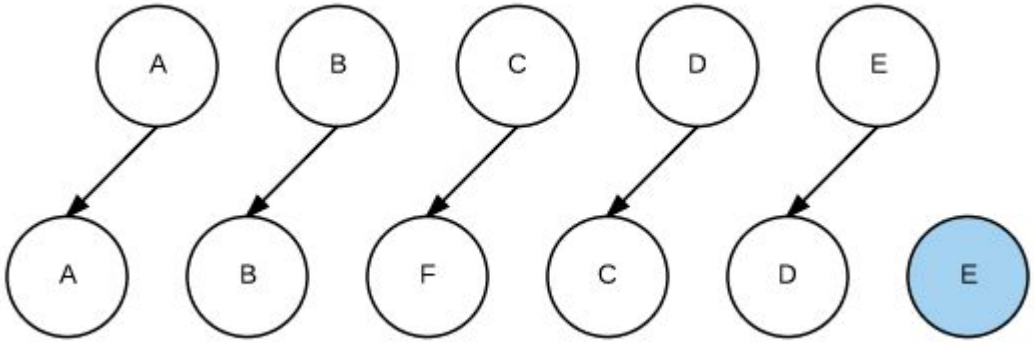
测试代码如下

```
<!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



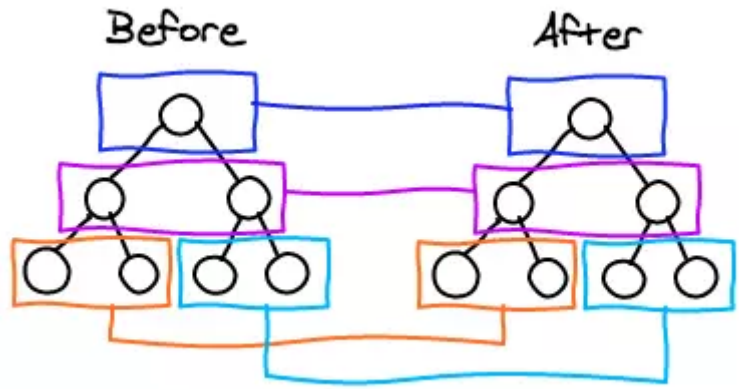
如果使用 key

```
// 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？

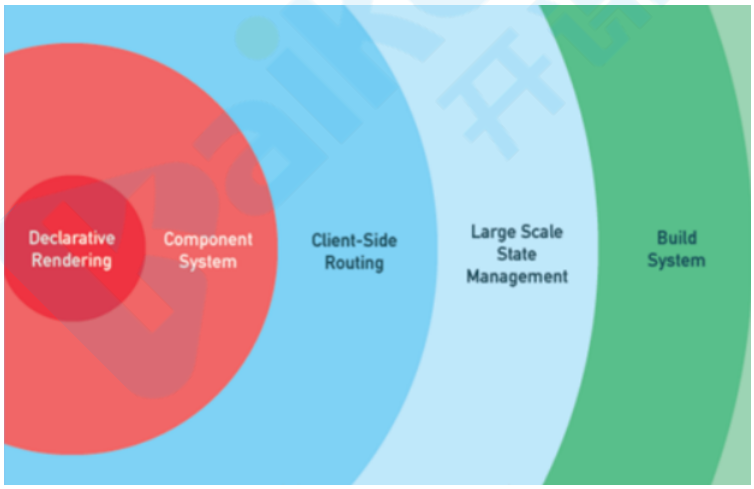
在 vue 的官网上写着大大的定义和特点：

- 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。

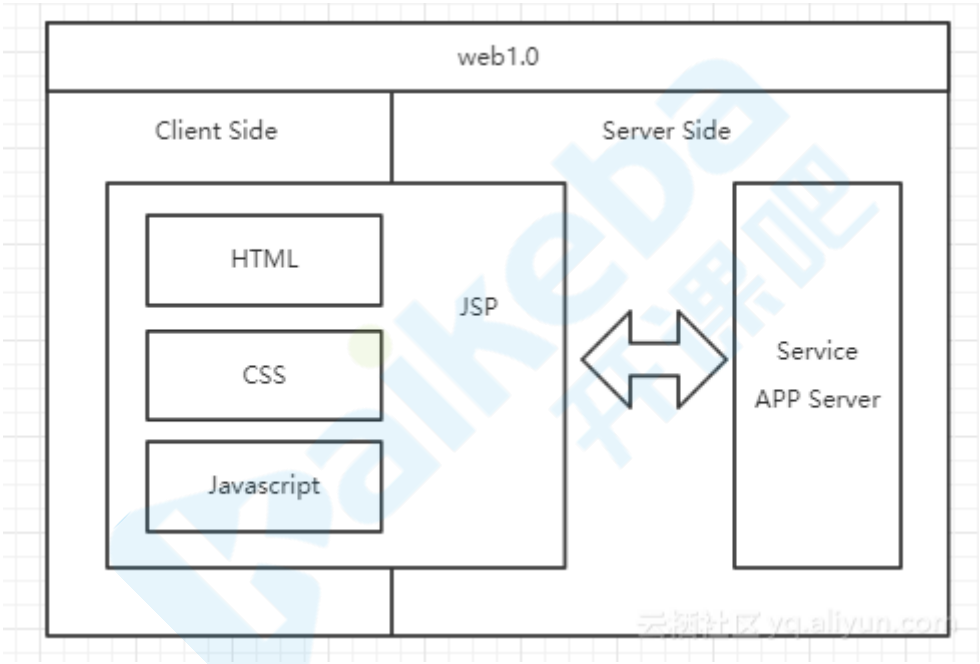
追求高效的过程还在继续，vue3 中引入 Proxy 对数据响应式改进以及编译器中对于静态内容编译的改进 都会让 vue 更加高效。

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。

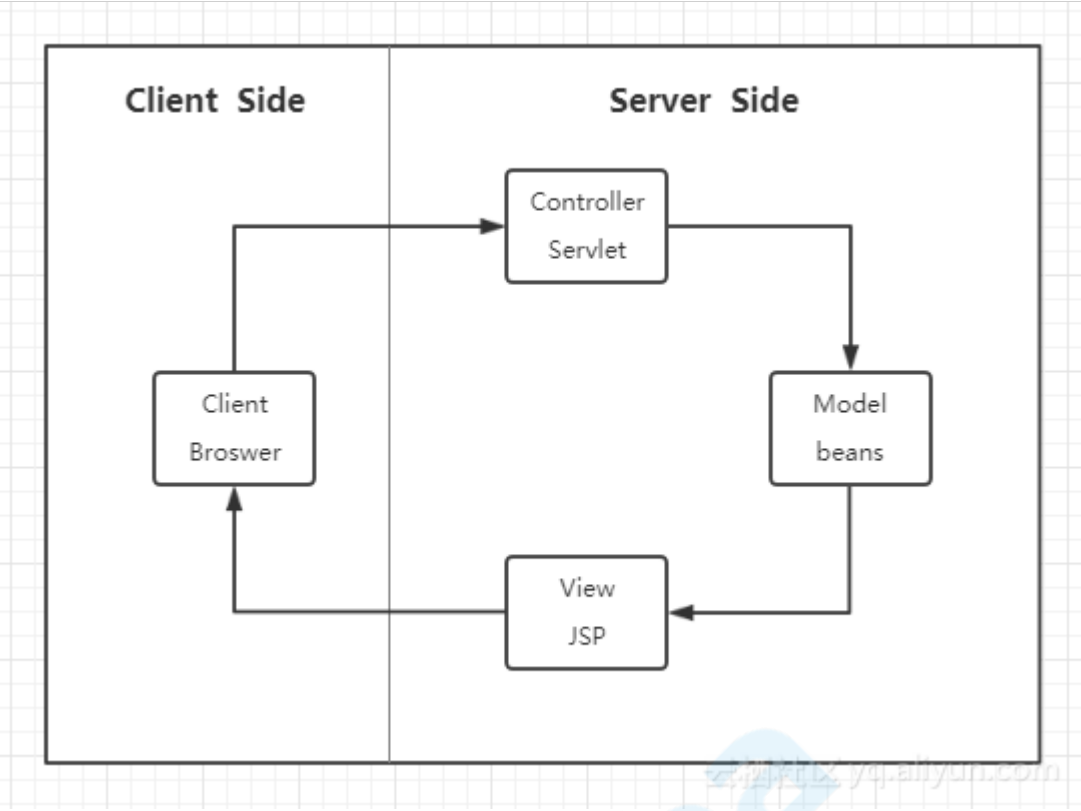
Web1.0 时代

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：



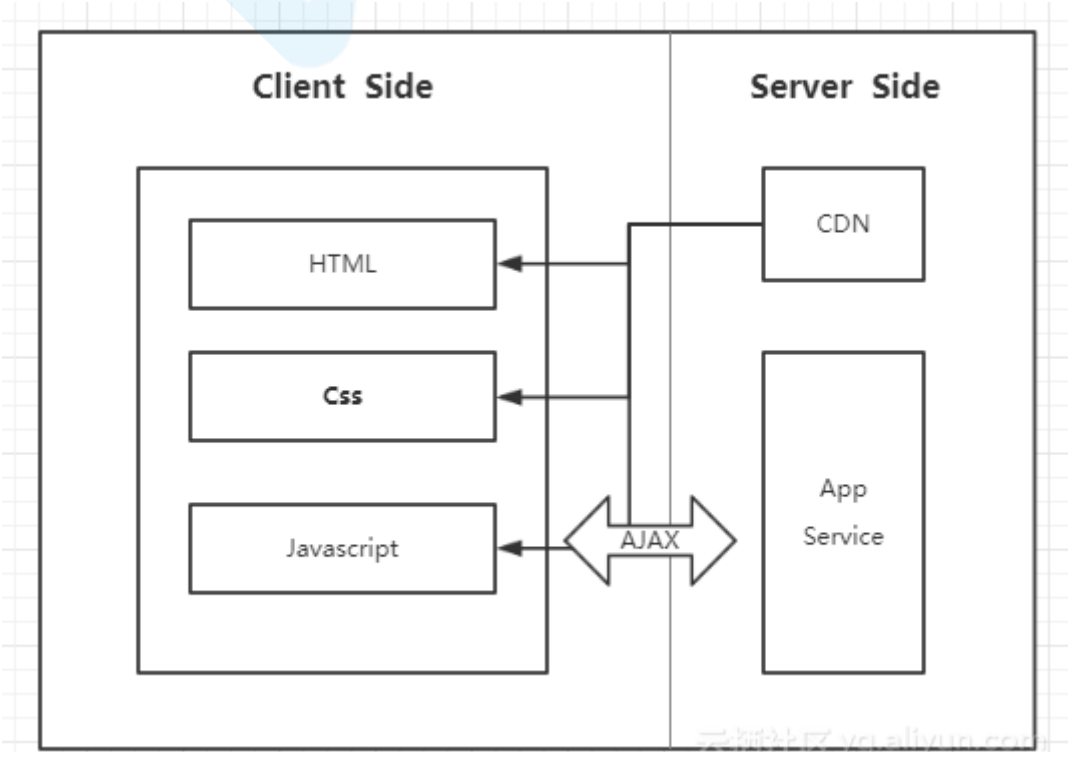
使用这种分层架构，职责清晰，代码易维护。但这里的 MVC 仅限于后端，前后端形成了一定的分离，前 端只完成了后端开发中的 view 层。

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

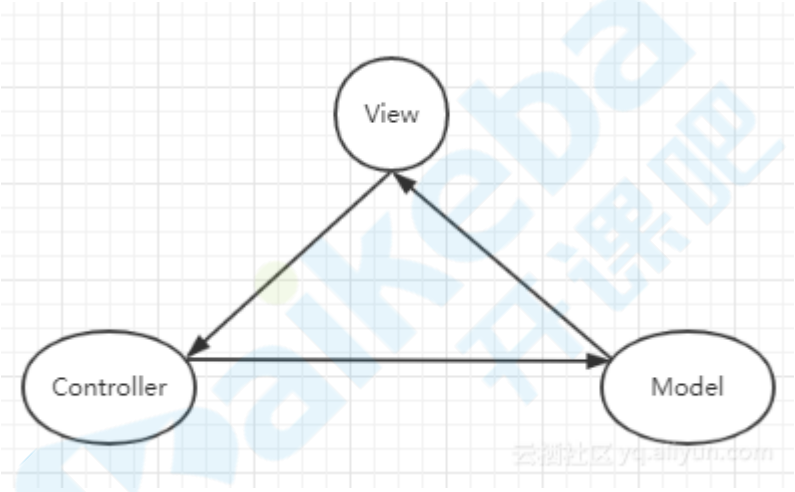
前端的 MVC 与后端类似，具备着 View、Controller 和 Model。

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

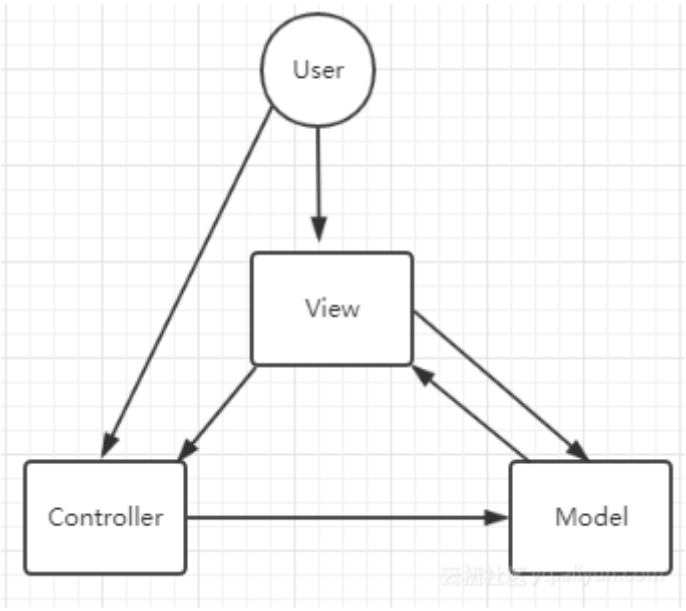
改 View：负责视图展示，将 model 中的数据可视化出来。

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。

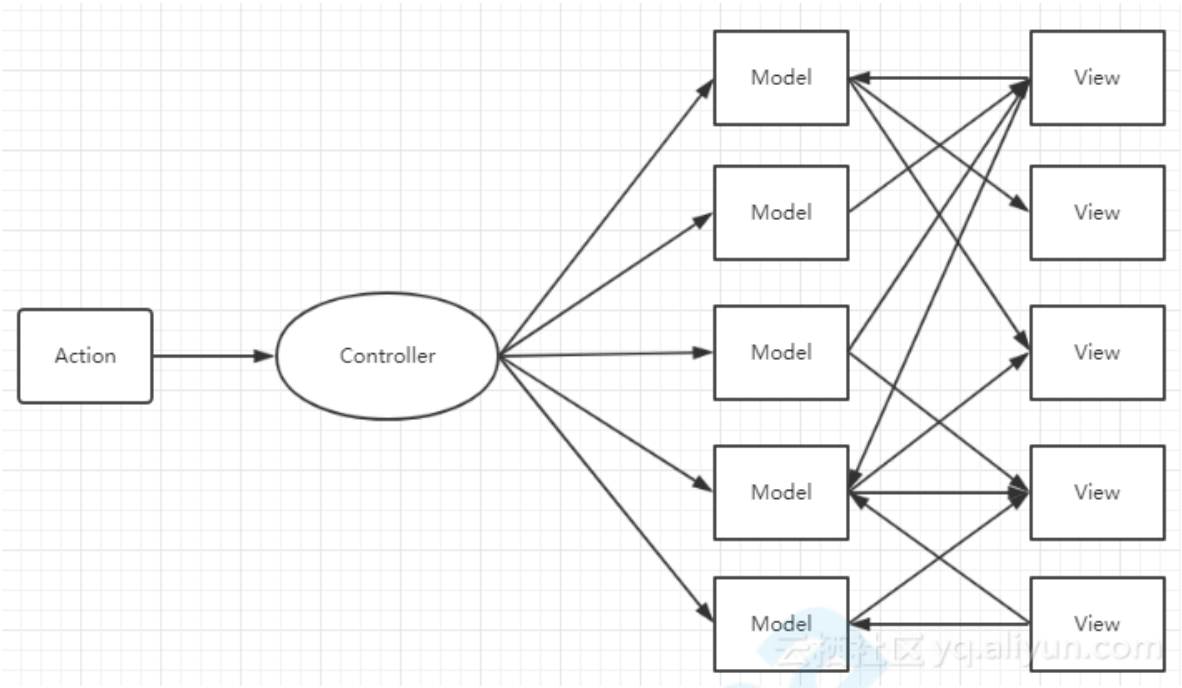
在实际场景中，我们往往会看到另一种模式，如图：



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：

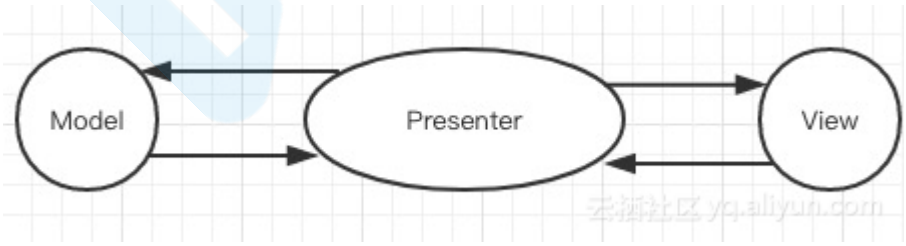


2. View 比较庞大，而 Controller 比较单薄：由于很多的开发者都会在 view 中写一些逻辑代码，逐渐的
   就导致 view 中的内容越来越庞大，而 controller 变得越来越单薄。

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

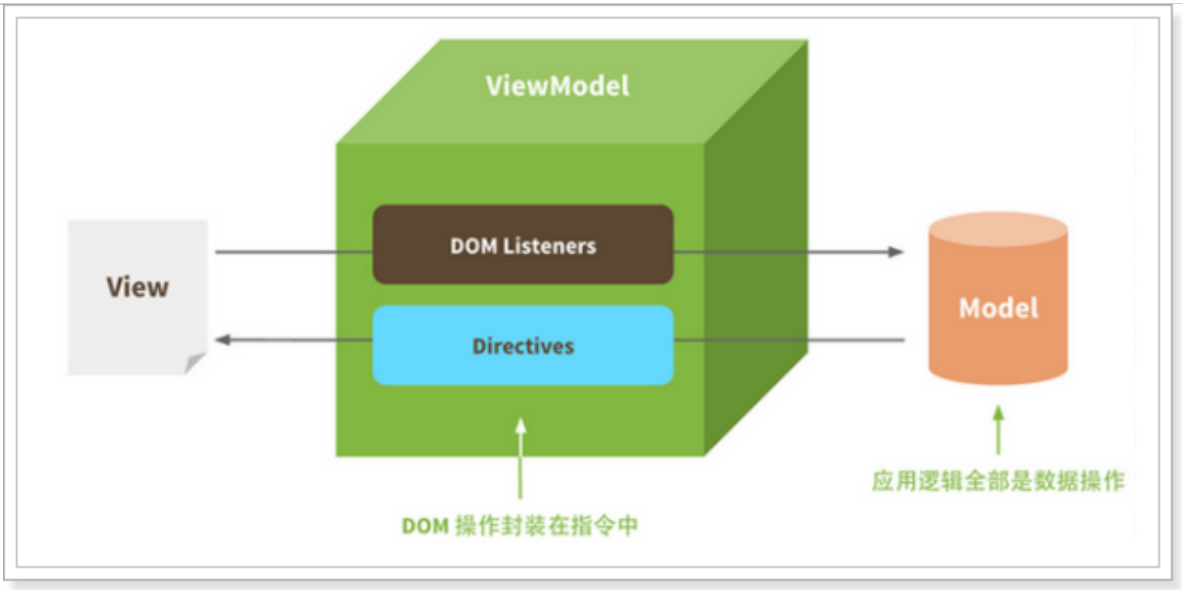
MVP 与 MVC 很接近，P 指的是 Presenter，presenter 可以理解为一个中间人，它负责着 View 和 Model 之 间的数据流动，防止 View 和 Model 之间直接交流。我们可以看一下图示：



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 }">     <FetchIte
mView
      :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

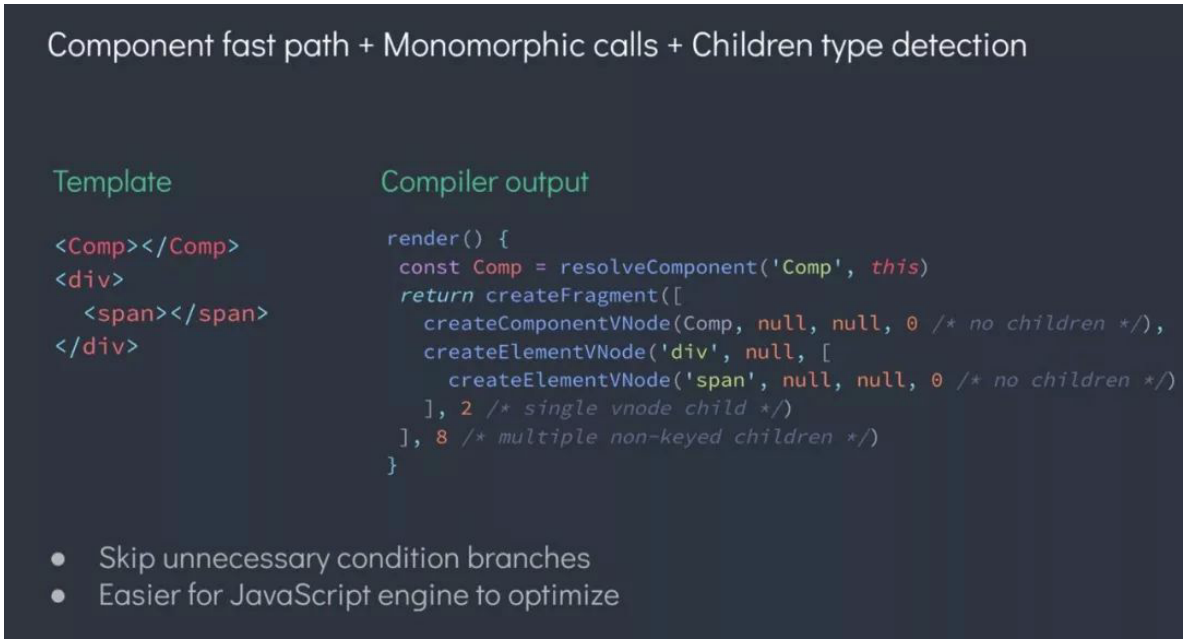
虚拟 **DOM** 重写

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

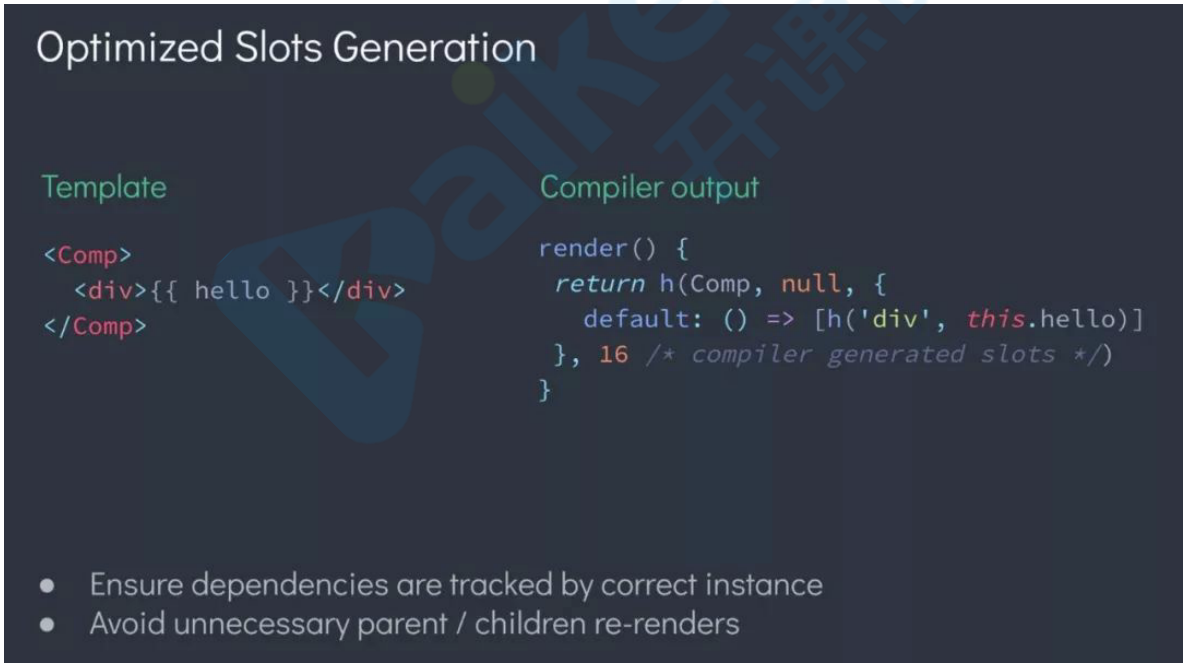
- JS 引擎更容易优化



优化 **slots** 生成

vue3 中可以单独重新渲染父级和子级

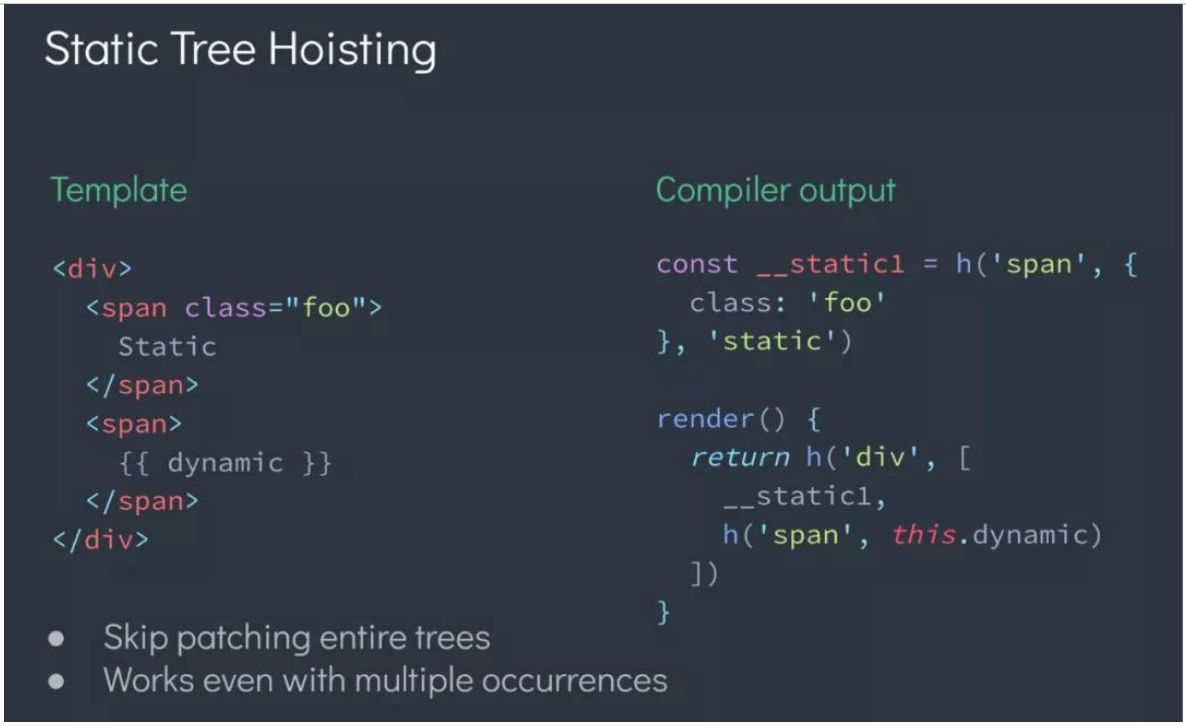
- 确保实例正确的跟踪依赖关系
- 避免不必要的父子组件重新渲染



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

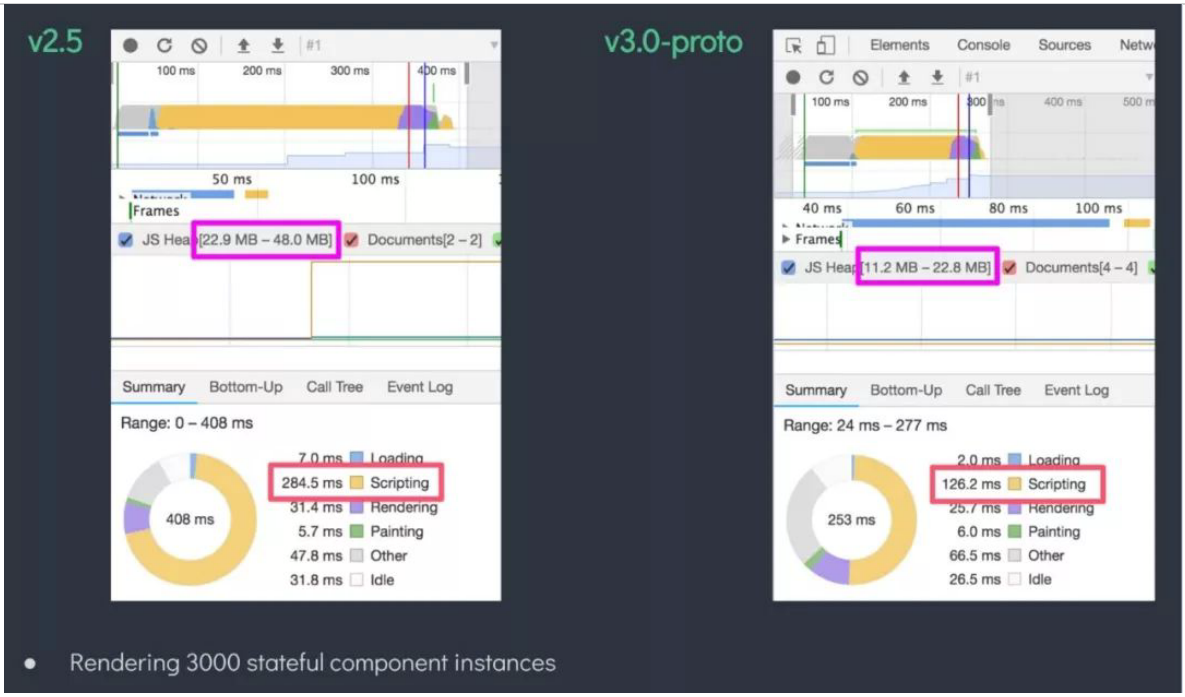


静态属性提升
使用静态属性提升，Vue 3 打补丁时将跳过这些属性不会改变的节点。

基于 **Proxy** 的数据响应式

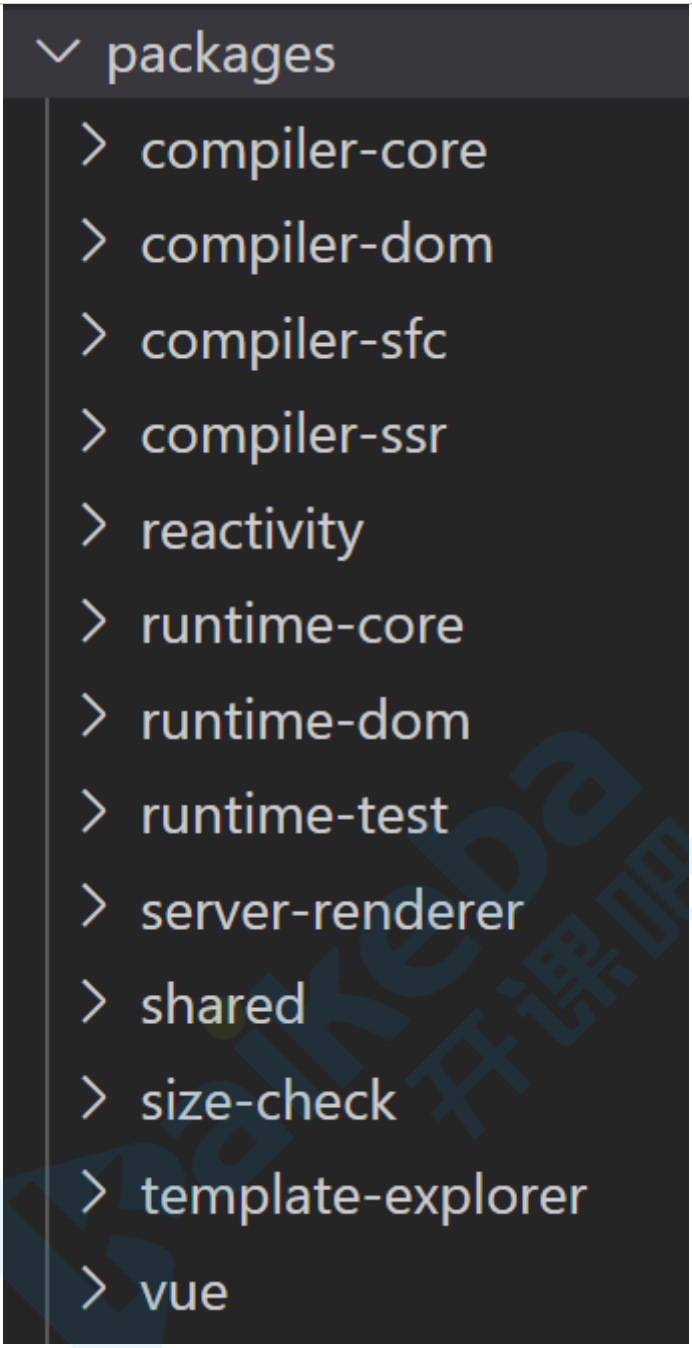
Vue 2 的响应式系统使用 Object.defineProperty 的 getter 和 setter。Vue 3 将使用 ES2015 Proxy 作为其观察机制，这将会带来如下变化：

- 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

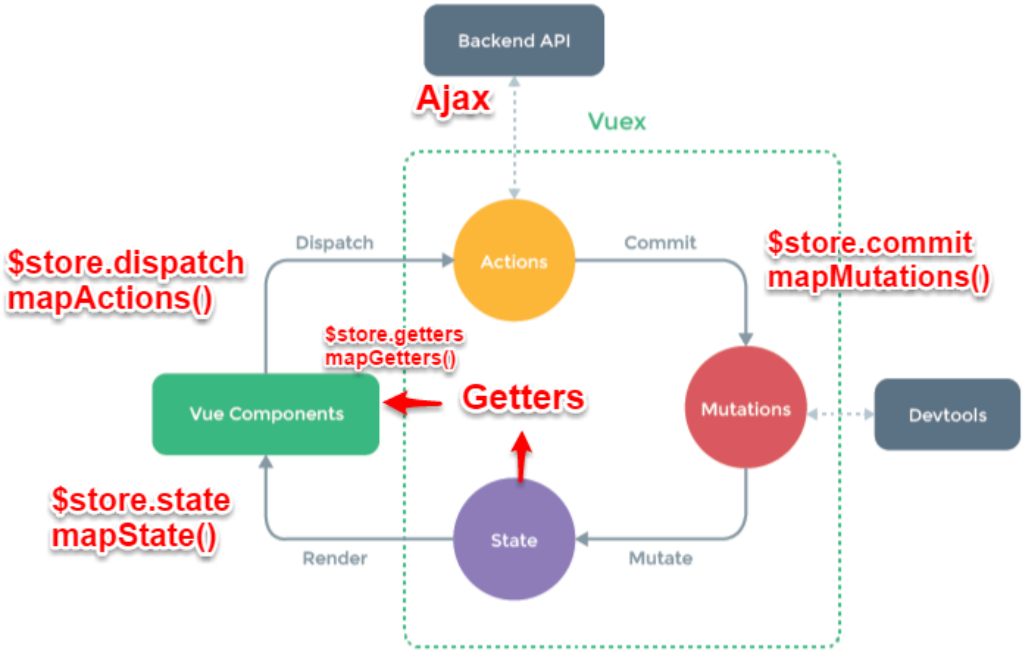
Vue 3 将带来更可维护的源代码。它不仅会使用 TypeScript，而且许多包被解耦，更加模块化。



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