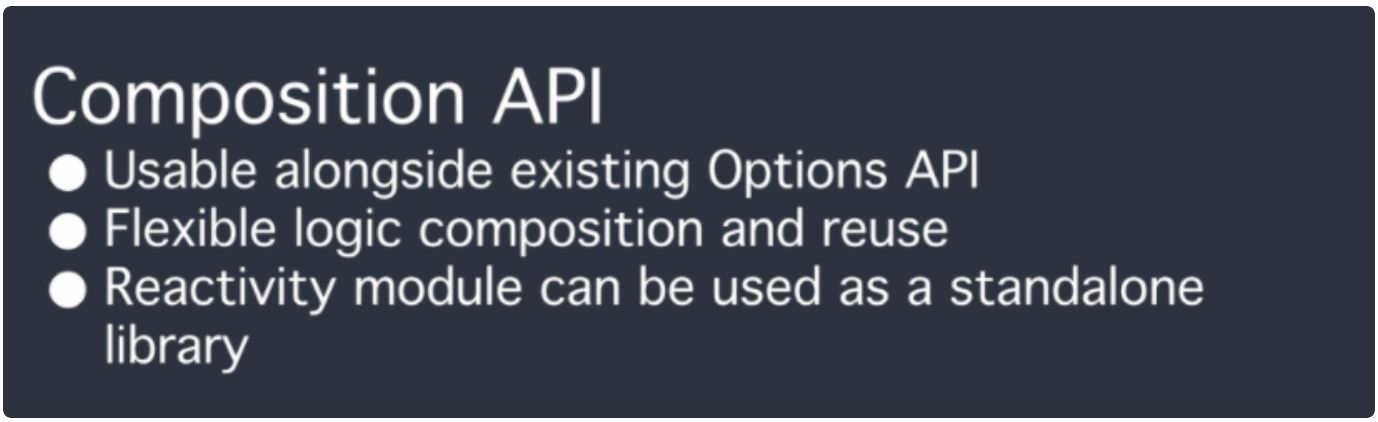
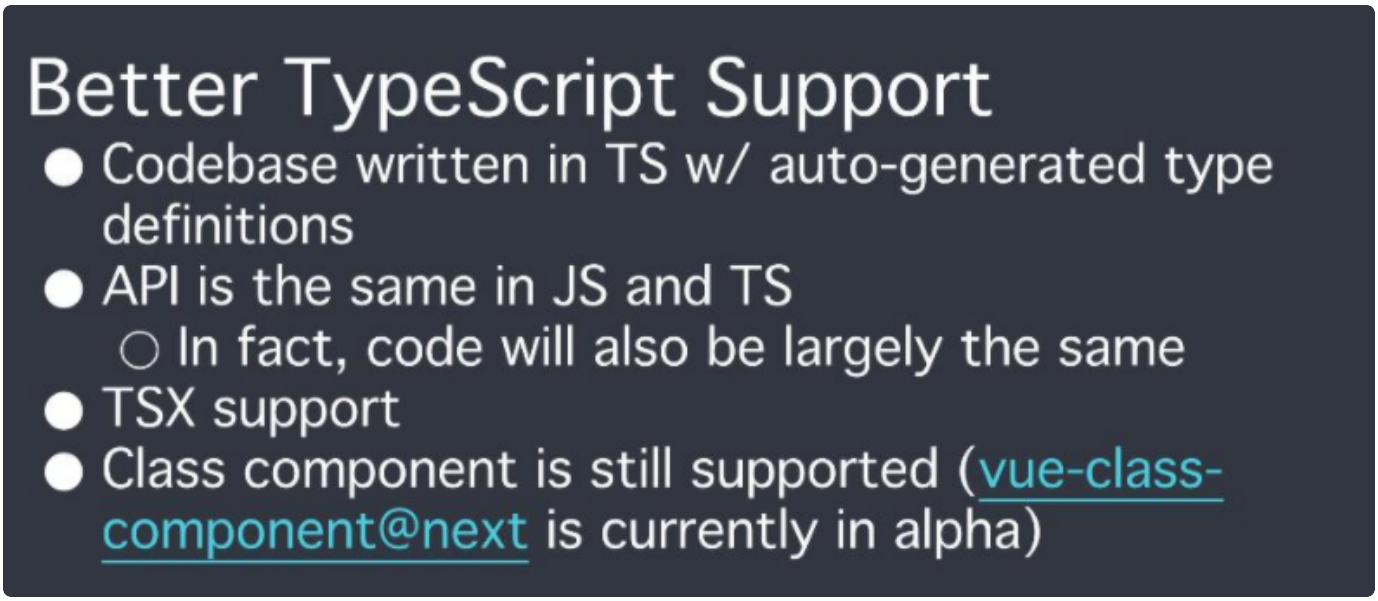
29.2.3.1. compositon Api

- 可与现有的 `Options API` 一起使用
- 灵活的逻辑组合与复用
- `Vue3` 模块可以和其他框架搭配使用



29.2.3.2. 更好的Typescript支持

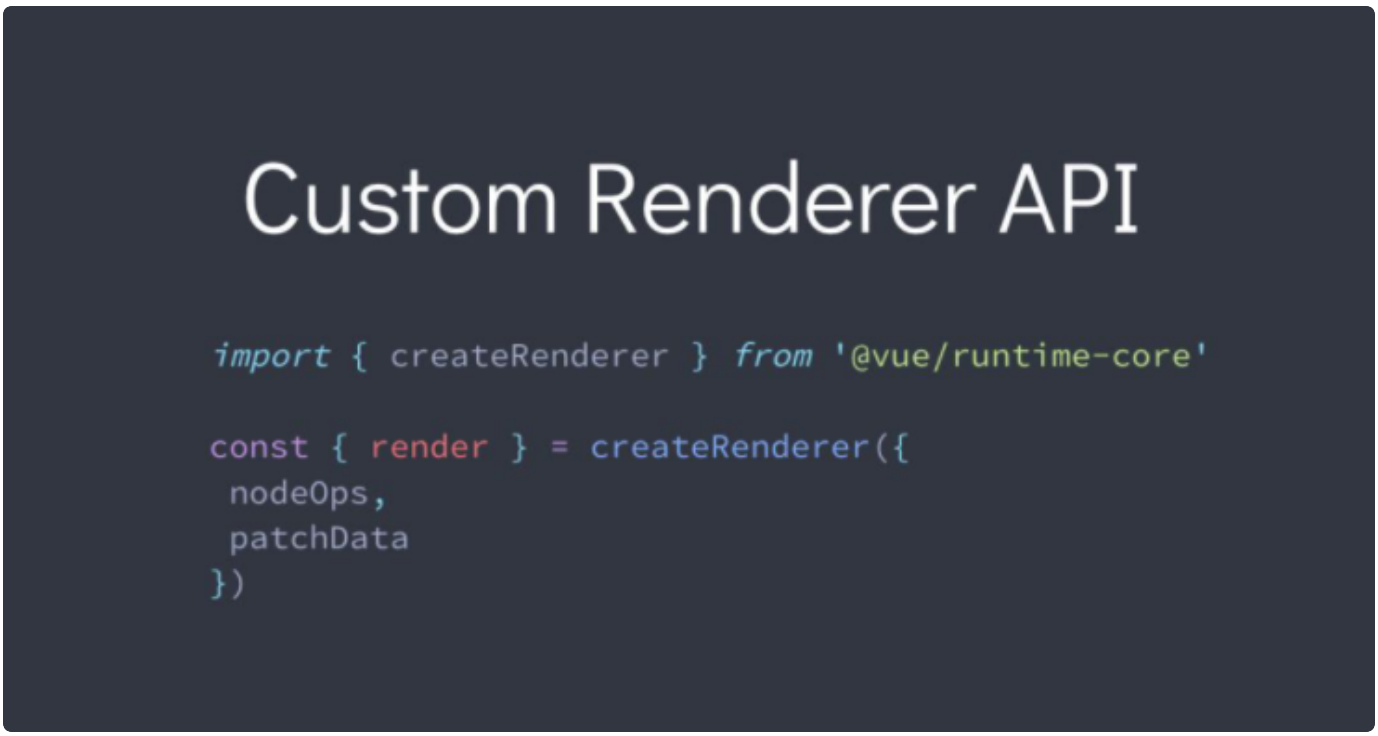
`VUE3` 是基于 `typescipt` 编写的，可以享受到自动的类型定义提示



29.2.3.3. 编译器重写

29.2.4. 更接近原生

可以自定义渲染 API



29.2.5. 更易使用

响应式 `Api` 暴露出来

Exposed reactivity API

```
import { observable, effect } from 'vue'

const state = observable({
 count: 0
})

effect(() => {
 console.log(`count is: ${state.count}`)
}) // count is: 0

state.count++ // count is: 1
```

轻松识别组件重新渲染原因

Easily identify why a component is re-rendering

```
const Comp = {
  render(props) {
    return h('div', props.count)
  },
  renderTriggered(event) {
    debugger
  }
}
```

29.3. Vue3新增特性

Vue 3 中需要关注的一些新功能包括：

- frаments
- Teleport
- composition Api
- createRenderer

29.3.1. frаments

在 `Vue3.x` 中，组件现在支持有多个根节点

JavaScript | 复制代码

```
<!-- Layout.vue -->
<template>
  <header>...</header>
  <main v-bind="$attrs">...</main>
  <footer>...</footer>
</template>
```

29.3.2. Teleport

`Teleport` 是一种能够将我们的模板移动到 `DOM` 中 `Vue app` 之外的其他位置的技术，就有点像哆啦A梦的“任意门”

在 `vue2` 中，像 `modals`，`toast` 等这样的元素，如果我们嵌套在 `Vue` 的某个组件内部，那么处理嵌套组件的定位、 `z-index` 和样式就会变得很困难

通过 `Teleport` ，我们可以在组件的逻辑位置写模板代码，然后在 `Vue` 应用范围之外渲染它

HTML | 复制代码

```
<button @click="showToast" class="btn">打开 toast</button>
<!-- to 属性就是目标位置 -->
<teleport to="#teleport-target">
    <div v-if="visible" class="toast-wrap">
        <div class="toast-msg">我是一个 Toast 文案</div>
    </div>
</teleport>
```

29.3.3. createRenderer

通过 `createRenderer`，我们能够构建自定义渲染器，我们能够将 `vue` 的开发模型扩展到其他平台

我们可以将其生成在 `canvas` 画布上

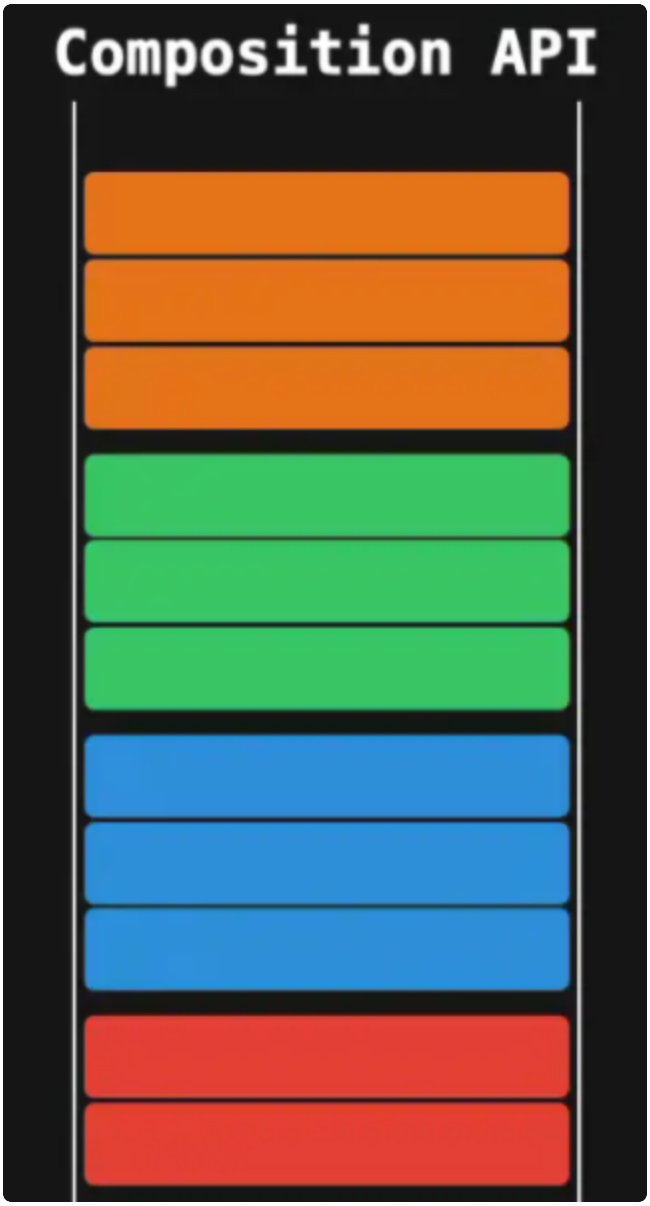
关于 `createRenderer`，我们了解下基本使用，就不展开讲述了

JavaScript | 复制代码

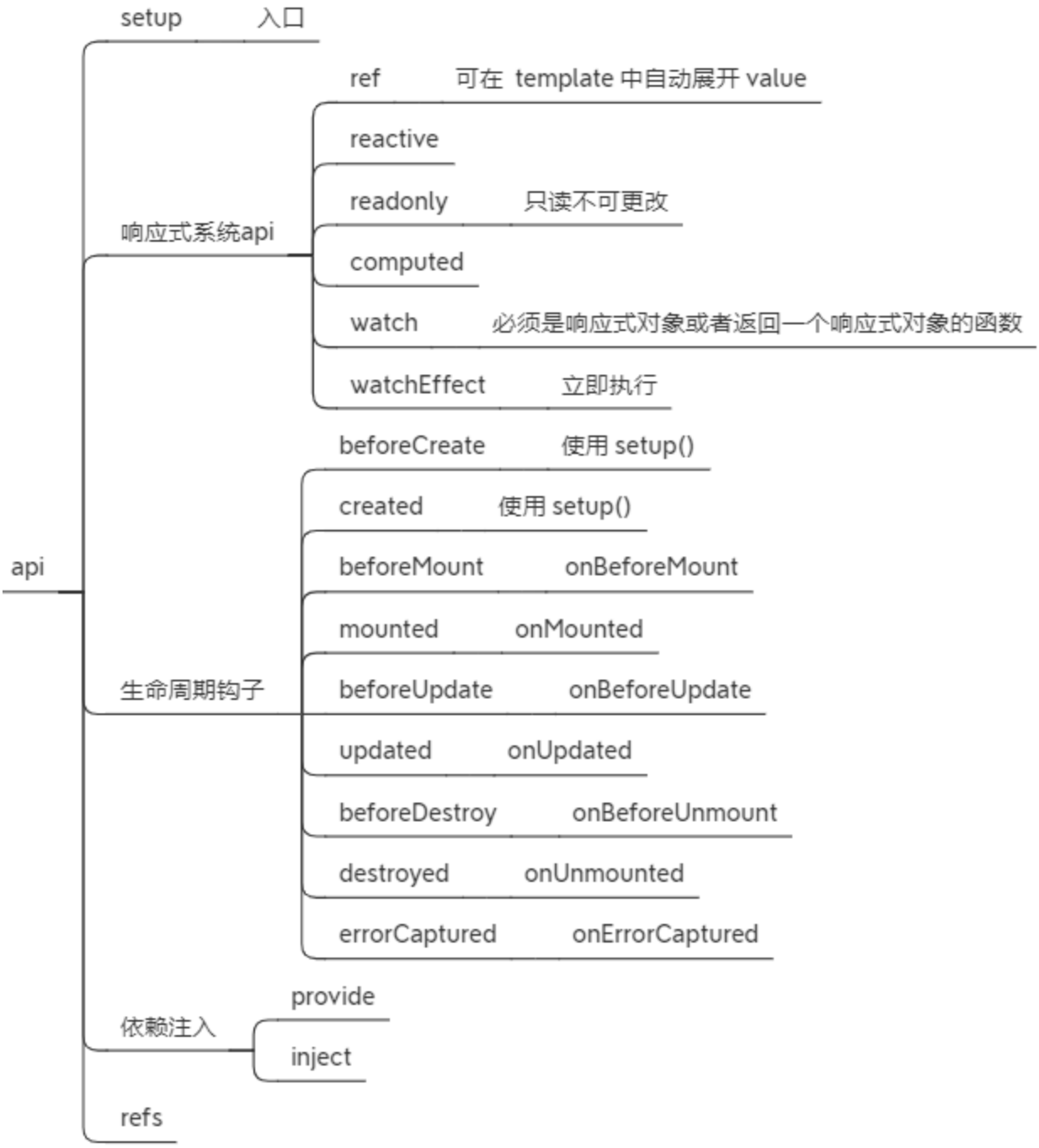
```javascript
import { createRenderer } from '@vue/runtime-core'

const { render, createApp } = createRenderer({
  patchProp,
  insert,
  remove,
  createElement,
  // ...
})

export { render, createApp }

export * from '@vue/runtime-core'
```

29.3.4. composition Api

composition Api，也就是组合式 `api`，通过这种形式，我们能够更加容易维护我们的代码，将相同功能的变量进行一个集中式的管理



关于 `compositon api` 的使用，这里以下图展开


api
setup 入口
响应式系统api
ref 可在 template 中自动展开 value
reactive
readonly 只读不可更改
computed
watch 必须是响应式对象或者返回一个响应式对象的函数
watchEffect 立即执行
生命周期钩子
beforeCreate 使用 setup()
created 使用 setup()
beforeMount onBeforeMount
mounted onMounted
beforeUpdate onBeforeUpdate
updated onUpdated
beforeDestroy onBeforeUnmount
destroyed onUnmounted
errorCaptured onErrorCaptured
依赖注入
provide
inject
refs

简单使用:

JavaScript | 复制代码

```javascript
export default {
    setup() {
        const count = ref(0)
        const double = computed(() => count.value * 2)
        function increment() {
            count.value++
        }
        onMounted(() => console.log('component mounted!'))
        return {
            count,
            double,
            increment
        }
    }
}
```

29.3.5. 非兼容变更

29.3.6. Global API

- 全局 `Vue API` 已更改为使用应用程序实例
- 全局和内部 `API` 已经被重构为可 `tree-shakable`

29.3.7. 模板指令

- 组件上 `v-model` 用法已更改
- `<template v-for>` 和 非 `v-for` 节点上 `key` 用法已更改
- 在同一元素上使用的 `v-if` 和 `v-for` 优先级已更改
- `v-bind="object"` 现在排序敏感
- `v-for` 中的 `ref` 不再注册 `ref` 数组

29.3.8. 组件

- 只能使用普通函数创建功能组件
- `functional` 属性在单文件组件 `(SFC)`
- 异步组件现在需要 `defineAsyncComponent` 方法来创建

29.3.9. 渲染函数

- 渲染函数 `API` 改变
- `$scopedSlots` property 已删除，所有插槽都通过 `$slots` 作为函数暴露
- 自定义指令 API 已更改为与组件生命周期一致
- 一些转换 `class` 被重命名了：
  - `v-enter` -> `v-enter-from`
  - `v-leave` -> `v-leave-from`
- 组件 `watch` 选项和实例方法 `$watch` 不再支持点分隔字符串路径，请改用计算函数作为参数
- 在 `Vue 2.x` 中，应用根容器的 `outerHTML` 将替换为根组件模板 (如果根组件没有模板/渲染选项，则最终编译为模板)。`VUE3.x` 现在使用应用程序容器的 `innerHTML` 。

29.3.10. 其他改变

- `destroyed` 生命周期选项被重命名为 `unmounted`
- `beforeDestroy` 生命周期选项被重命名为 `beforeUnmount`
- `[prop default` 工厂函数不再有权访问 `this` 是上下文
- 自定义指令 API 已更改为与组件生命周期一致
- `data` 应始终声明为函数
- 来自 `mixin` 的 `data` 选项现在可简单地合并
- `attribute` 强制策略已更改
- 一些过渡 `class` 被重命名
- 组建 watch 选项和实例方法 `$watch` 不再支持以点分隔的字符串路径。请改用计算属性函数作为参数。
- `<template>` 没有特殊指令的标记 (`v-if/else-if/else` 、 `v-for` 或 `v-slot`) 现在被视为普通元素，并将生成原生的 `<template>` 元素，而不是渲染其内部内容。
- 在 `Vue 2.x` 中，应用根容器的 `outerHTML` 将替换为根组件模板 (如果根组件没有模板/渲染选项，则最终编译为模板)。`Vue 3.x` 现在使用应用容器的 `innerHTML` ，这意味着容器本身不再被视为模板的一部分。

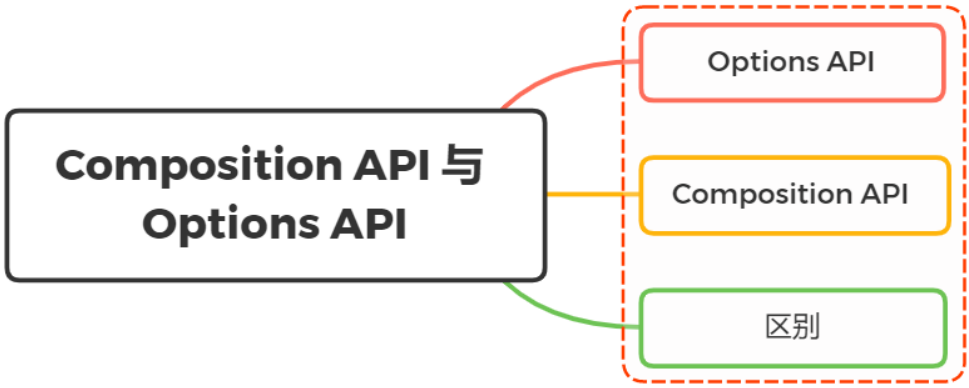
29.3.11. 移除 API

- `keyCode` 支持作为 `v-on` 的修饰符
- `$on` ， `$off` 和 `$once` 实例方法

- 过滤 `filter`
- 内联模板 `attribute`
- `$destroy` 实例方法。用户不应再手动管理单个 `Vue` 组件的生命周期。

Vue3面试真题（6题）

1. Vue3.0 所采用的 Composition Api 与 Vue2.x 使用的 Options Api 有什么不同?



1.1. 开始之前

`Composition API` 可以说是 `Vue3` 的最大特点，那么为什么要推出 `Composition Api`，解决了什么问题?

通常使用 `Vue2` 开发的项目，普遍会存在以下问题：

- 代码的可读性随着组件变大而变差
- 每一种代码复用的方式，都存在缺点
- TypeScript支持有限

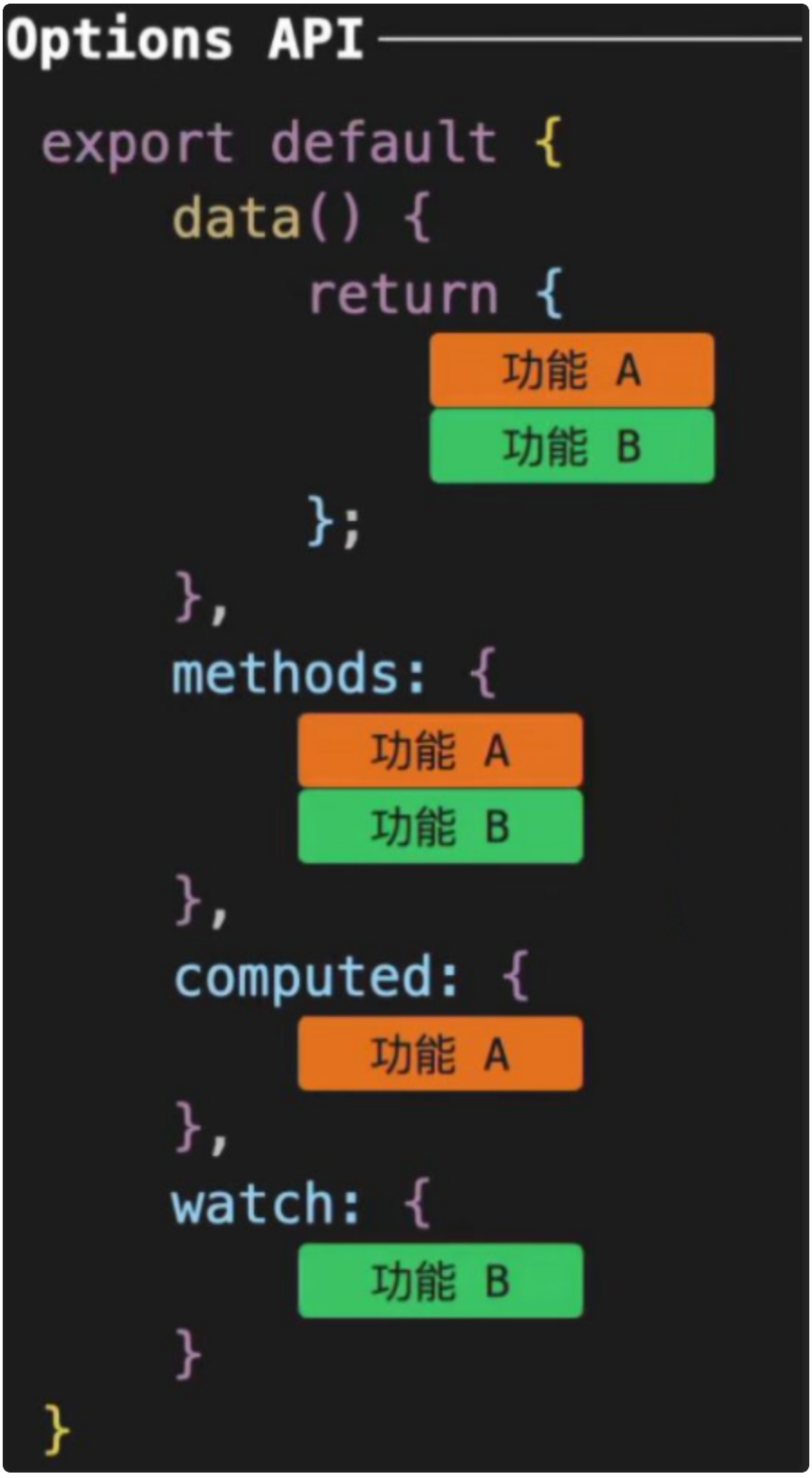
以上通过使用 `Composition Api` 都能迎刃而解

1.2. 正文

1.2.1. Options Api

`Options API`，即大家常说的选项API，即以 `vue` 为后缀的文件，通过定义 `methods`，`computed`，`watch`，`data` 等属性与方法，共同处理页面逻辑

如下图：



可以看到 `Options` 代码编写方式，如果是组件状态，则写在 `data` 属性上，如果是方法，则写在 `methods` 属性上...

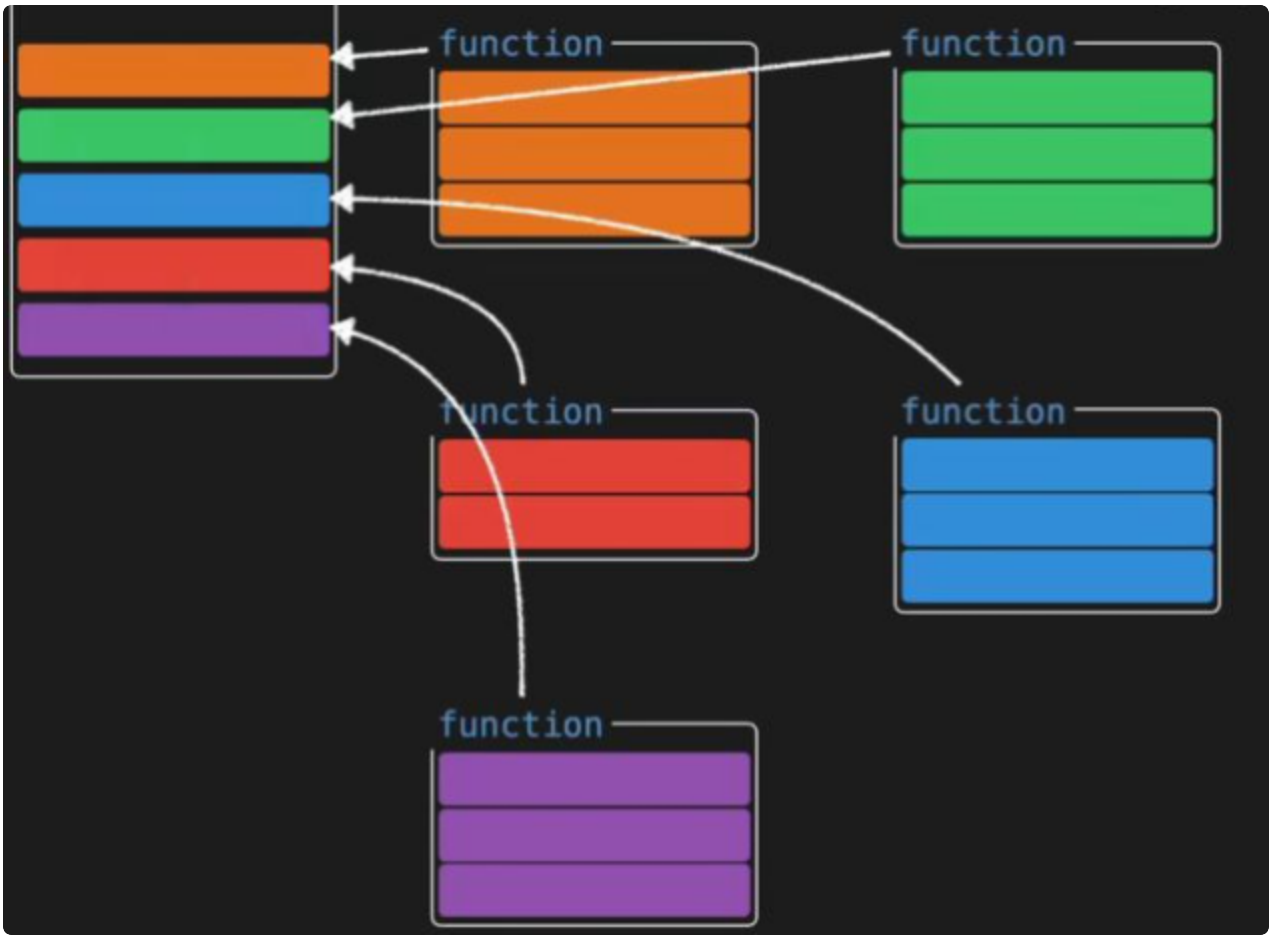
用组件的选项（`data`、`computed`、`methods`、`watch`）组织逻辑在大多数情况下都有效

然而，当组件变得复杂，导致对应属性的列表也会增长，这可能会导致组件难以阅读和理解

1.2.2. Composition Api

在 Vue3 Composition API 中，组件根据逻辑功能来组织的，一个功能所定义的所有 API 会放在一起（更加的高内聚，低耦合）

即使项目很大，功能很多，我们都能快速的定位到这个功能所用到的所有 API



1.2.3. 对比

下面对 `Composition Api` 与 `Options Api` 进行两大方面的比较

- 逻辑组织
- 逻辑复用

1.2.3.1. 逻辑组织

1.2.3.1.1. Options API

假设一个组件是一个大型组件，其内部有很多处理逻辑关注点（对应下图不用颜色）

可以看到，这种碎片化使得理解和维护复杂组件变得困难

选项的分离掩盖了潜在的逻辑问题。此外，在处理单个逻辑关注点时，我们必须不断地“跳转”相关代码的选项块

1.2.3.1.2. Compostion API

而 `Compositon API` 正是解决上述问题，将某个逻辑关注点相关的代码全都放在一个函数里，这样当需要修改一个功能时，就不再需要在文件中跳来跳去

下面举个简单例子，将处理 `count` 属性相关的代码放在同一个函数了

JavaScript | 复制代码

```javascript
function useCount() {
    let count = ref(10);
    let double = computed(() => {
        return count.value * 2;
    });

    const handleConut = () => {
        count.value = count.value * 2;
    };

    console.log(count);

    return {
        count,
        double,
        handleConut,
    };
}
```

组件上中使用 `count`

JavaScript | 复制代码

```javascript
export default defineComponent({
    setup() {
        const { count, double, handleConut } = useCount();
        return {
            count,
            double,
            handleConut
        }
    },
});
```

再来一张图进行对比，可以很直观地感受到 `Composition API` 在逻辑组织方面的优势，以后修改一个属性功能的时候，只需要跳到控制该属性的方法中即可

1.2.3.2. 逻辑复用

在 `Vue2` 中，我们是用过 `mixin` 去复用相同的逻辑

下面举个例子，我们会另起一个 `mixin.js` 文件

JavaScript | 复制代码

```javascript
export const MoveMixin = {
  data() {
    return {
      x: 0,
      y: 0,
    };
  },

  methods: {
    handleKeyup(e) {
      console.log(e.code);
      // 上下左右 x y
      switch (e.code) {
        case "ArrowUp":
          this.y--;
          break;
        case "ArrowDown":
          this.y++;
          break;
        case "ArrowLeft":
          this.x--;
          break;
        case "ArrowRight":
          this.x++;
          break;
      }
    },
  },

  mounted() {
    window.addEventListener("keyup", this.handleKeyup);
  },

  unmounted() {
    window.removeEventListener("keyup", this.handleKeyup);
  },
};
```

然后在组件中使用

```javascript
<template>
  <div>
    Mouse position: x {{ x }} / y {{ y }}
  </div>
</template>
<script>
import mousePositionMixin from './mouse'
export default {
  mixins: [mousePositionMixin]
}
</script>
```

使用单个 `mixin` 似乎问题不大，但是当我们一个组件混入大量不同的 `mixins` 的时候

```javascript
mixins: [mousePositionMixin, fooMixin, barMixin, otherMixin]
```

会存在两个非常明显的问题：

- 命名冲突
- 数据来源不清晰

现在通过 `Compositon API` 这种方式改写上面的代码

JavaScript | 复制代码

```javascript
import { onMounted, onUnmounted, reactive } from "vue";
export function useMove() {
  const position = reactive({
    x: 0,
    y: 0,
  });

  const handleKeyup = (e) => {
    console.log(e.code);
    // 上下左右 x y
    switch (e.code) {
      case "ArrowUp":
        // y.value--;
        position.y--;
        break;
      case "ArrowDown":
        // y.value++;
        position.y++;
        break;
      case "ArrowLeft":
        // x.value--;
        position.x--;
        break;
      case "ArrowRight":
        // x.value++;
        position.x++;
        break;
    }
  };

  onMounted(() => {
    window.addEventListener("keyup", handleKeyup);
  });

  onUnmounted(() => {
    window.removeEventListener("keyup", handleKeyup);
  });

  return { position };
}
```

在组件中使用

JavaScript | 复制代码

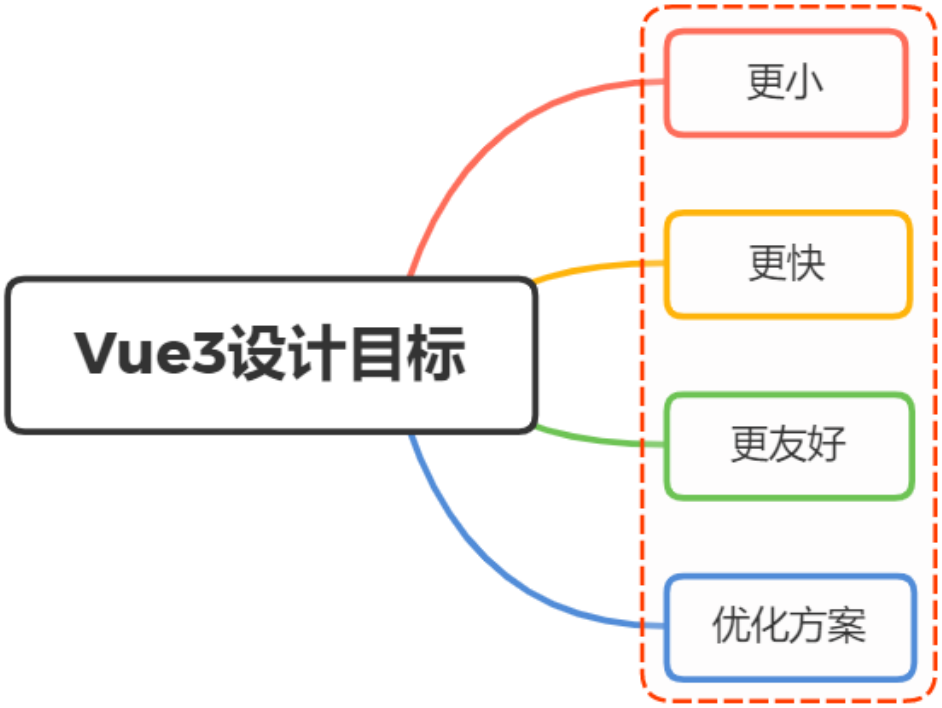
```
<template>
  <div>
    Mouse position: x {{ x }} / y {{ y }}
  </div>
</template>

<script>
import { useMove } from "./useMove";
import { toRefs } from "vue";
export default {
  setup() {
    const { position } = useMove();
    const { x, y } = toRefs(position);
    return {
      x,
      y,
    };

  },
};
</script>
```

可以看到，整个数据来源清晰了，即使去编写更多的 hook 函数，也不会出现命名冲突的问题

1.3. 小结

- 在逻辑组织和逻辑复用方面，`Composition API` 是优于 `Options API`
- 因为 `Composition API` 几乎是函数，会有更好的类型推断。
- `Composition API` 对 `tree-shaking` 友好，代码也更容易压缩
- `Composition API` 中见不到 `this` 的使用，减少了 `this` 指向不明的情况
- 如果是小型组件，可以继续使用 `Options API` ，也是十分友好的

2. Vue3.0的设计目标是什么？做了哪些优化



2.1. 设计目标

不以解决实际业务痛点的更新都是耍流氓，下面我们来列举一下 `Vue3` 之前我们或许会面临的问题

- 随着功能的增长，复杂组件的代码变得越来越难以维护
- 缺少一种比较「干净」的在多个组件之间提取和复用逻辑的机制
- 类型推断不够友好
- `bundle` 的时间太久了

而 `Vue3` 经过长达两三年时间的筹备，做了哪些事情?

我们从结果反推

- 更小
- 更快
- TypeScript支持
- API设计一致性
- 提高自身可维护性
- 开放更多底层功能

一句话概述，就是更小更快更友好了

2.1.1. 更小

`Vue3` 移除一些不常用的 `API`

引入 `tree-shaking`，可以将无用模块“剪辑”，仅打包需要的，使打包的整体体积变小了

2.1.2. 更快

主要体现在编译方面：

- diff算法优化
- 静态提升
- 事件监听缓存
- SSR优化

2.1.3. 更友好

`vue3` 在兼顾 `vue2` 的 `options API` 的同时还推出了 `composition API`，大大增加了代码的逻辑组织和代码复用能力

这里代码简单演示下：

存在一个获取鼠标位置的函数

JavaScript | 复制代码

```
import { toRefs, reactive } from 'vue';
function useMouse(){
    const state = reactive({x:0,y:0});
    const update = e=>{
        state.x = e.pageX;
        state.y = e.pageY;
    }
    onMounted(()=>{
        window.addEventListener('mousemove',update);
    })
    onUnmounted(()=>{
        window.removeEventListener('mousemove',update);
    })

    return toRefs(state);
}
```

我们只需要调用这个函数，即可获取 `x` 、 `y` 的坐标，完全不用关注实现过程

试想一下，如果很多类似的第三方库，我们只需要调用即可，不必关注实现过程，开发效率大大提高

同时， `VUE3` 是基于 `typescipt` 编写的，可以享受到自动的类型定义提示

2.2. 优化方案

`vue3` 从很多层面都做了优化，可以分成三个方面：

- 源码
- 性能
- 语法 API

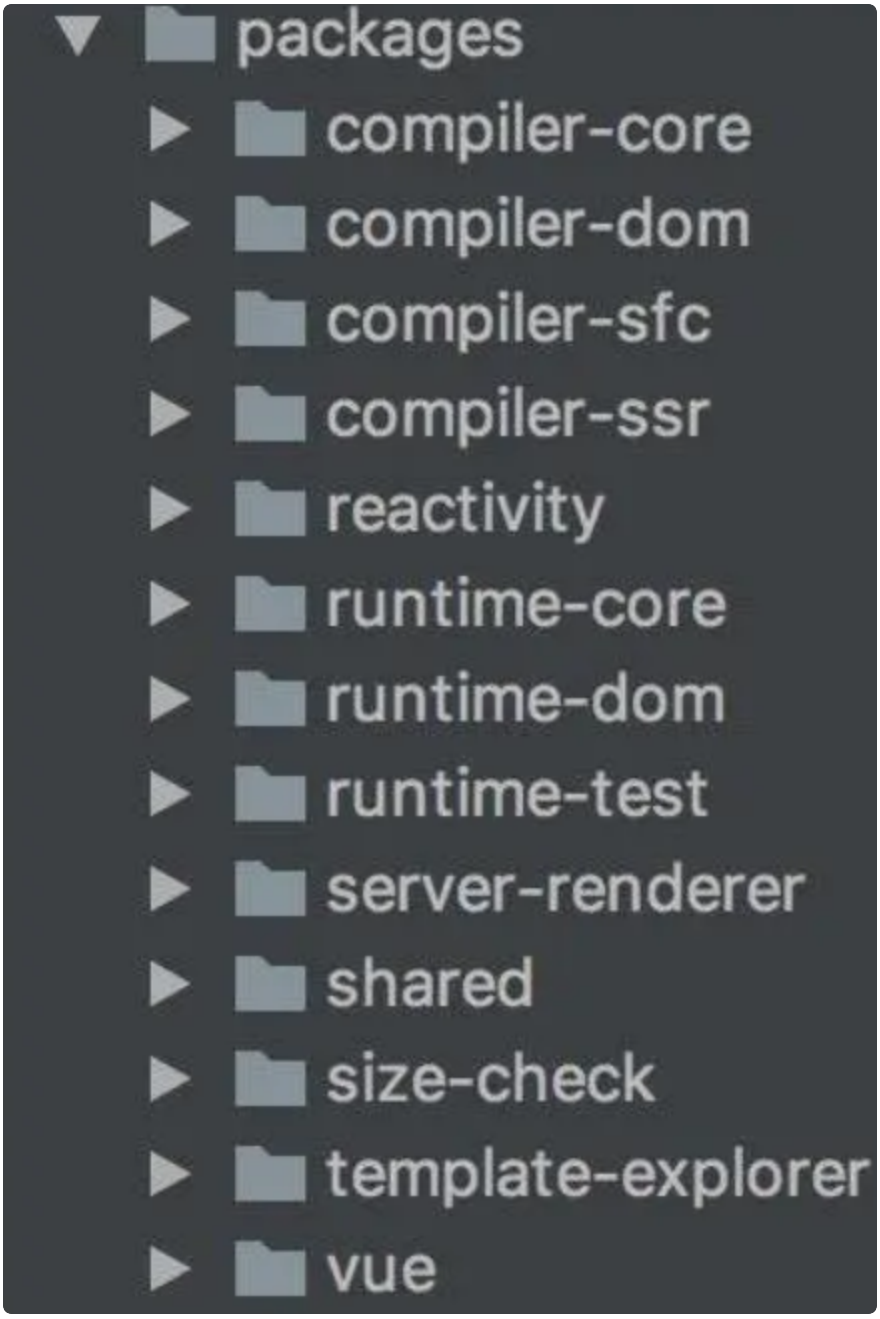
2.2.1. 源码

源码可以从两个层面展开：

- 源码管理
- TypeScript

2.2.1.1. 源码管理

`vue3` 整个源码是通过 `monorepo` 的方式维护的，根据功能将不同的模块拆分到 `packages` 目录下面不同的子目录中



这样使得模块拆分更细化，职责划分更明确，模块之间的依赖关系也更加明确，开发人员也更容易阅读、理解和更改所有模块源码，提高代码的可维护性

另外一些 `package`（比如 `reactivity` 响应式库）是可以独立于 `Vue` 使用的，这样用户如果只想使用 `Vue3` 的响应式能力，可以单独依赖这个响应式库而不用去依赖整个 `Vue`

2.2.1.2. TypeScript

`Vue3` 是基于 `typeScript` 编写的，提供了更好的类型检查，能支持复杂的类型推导

2.2.2. 性能

`vue3` 是从什么哪些方面对性能进行进一步优化呢?

- 体积优化
- 编译优化
- 数据劫持优化

这里讲述数据劫持:

在 `vue2` 中，数据劫持是通过 `Object.defineProperty` ，这个 API 有一些缺陷，并不能检测对象属性的添加和删除

JavaScript | 复制代码

```
Object.defineProperty(data, 'a',{
  get(){
    // track
  },
  set(){
    // trigger
  }
})
```

尽管 `Vue` 为了解决这个问题提供了 `set` 和 `delete` 实例方法，但是对于用户来说，还是增加了一定的心智负担

同时在面对嵌套层级比较深的情况下，就存在性能问题

JavaScript | 复制代码

```
default {
  data: {
    a: {
      b: {
          c: {
          d: 1
        }
      }
    }
  }
}
```

相比之下， `vue3` 是通过 `proxy` 监听整个对象，那么对于删除还是监听当然也能监听到

同时 `Proxy` 并不能监听到内部深层次的对象变化，而 `Vue3` 的处理方式是在 `getter` 中去递归响应式，这样的好处是真正访问到的内部对象才会变成响应式，而不是无脑递归

2.2.3. 语法 API

这里当然说的就是 `composition API`，其两大显著的优化：

- 优化逻辑组织
- 优化逻辑复用

2.2.3.1. 逻辑组织

一张图，我们可以很直观地感受到 `Composition API` 在逻辑组织方面的优势

相同功能的代码编写在一块，而不像 `options API` 那样，各个功能的代码混成一块