Vue3 快速上手

1.Vue3 简介

- 2020年9月18日，Vue.js 发布3.0版本，代号：One Piece（海贼王）
- 耗时2年多、**2600+次提交**、**30+个RFC**、**600+次PR**、**99位贡献者**
- github 上的 tags 地址：**https://github.com/vuejs/vue-next/releases/tag/v3.0.0**

2.Vue3带来了什么

1.性能的提升

- 打包大小减少 41%
- 初次渲染快 55%, 更新渲染快 133%
- 内存减少 54%

  ……

2.源码的升级

- 使用 `Proxy` 代替 `defineProperty` 实现响应式
- 重写虚拟 DOM 的实现和 `Tree-Shaking`

  ……

3.拥抱 TypeScript

- Vue3 可以更好的支持 TypeScript

4.新的特性

1. Composition API（组合 API）
    - `setup` 配置
    - `ref` 与 `reactive`
    - `watch` 与 `watchEffect`
    - `provide` 与 `inject`
    - ……
2. 新的内置组件
    - `Fragment`
    - `Teleport`
    - `Suspense`
3. 其他改变
    - 新的生命周期钩子
    - data 选项应始终被声明为一个函数
    - 移除 `keyCode` 支持作为 `v-on` 的修饰符
    - ……

一、创建 Vue3.0 工程

1.使用 vue-cli 创建

官方文档：**https://cli.vuejs.org/zh/guide/creating-a-project.html#vue-create**

```
## 查看@vue/cli版本，确保@vue/cli版本在4.5.0以上
vue --version
## 安装或者升级你的@vue/cli
npm install -g @vue/cli
## 创建
vue create vue_test
## 启动
cd vue_test
npm run serve
```

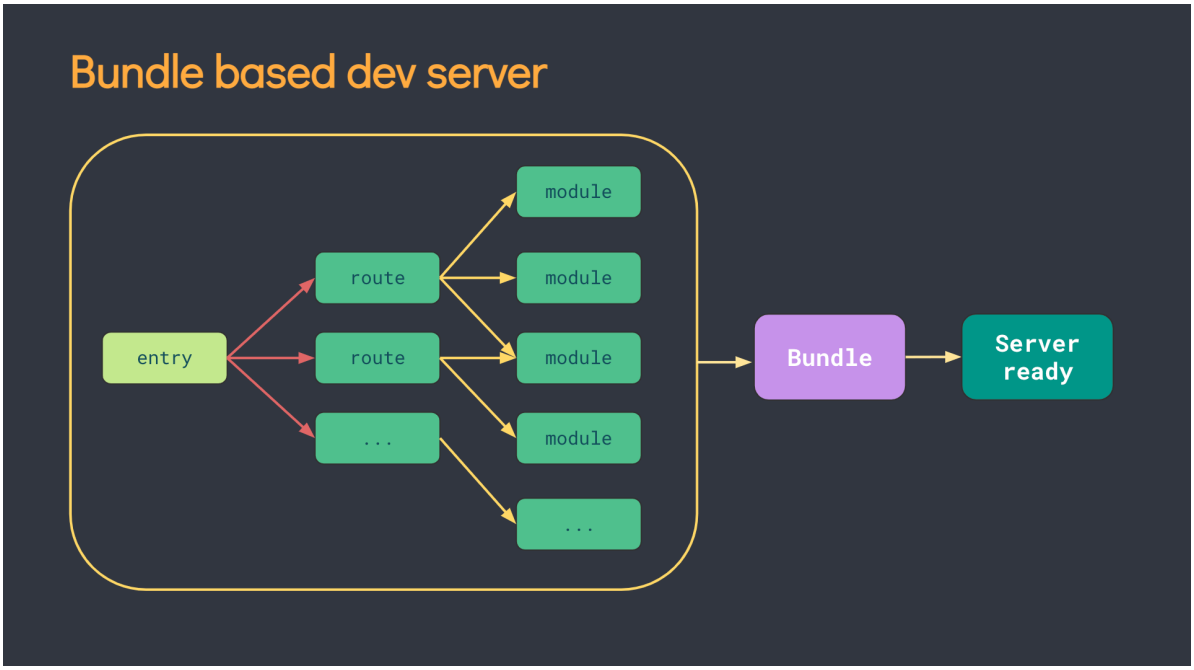
2.使用 vite 创建

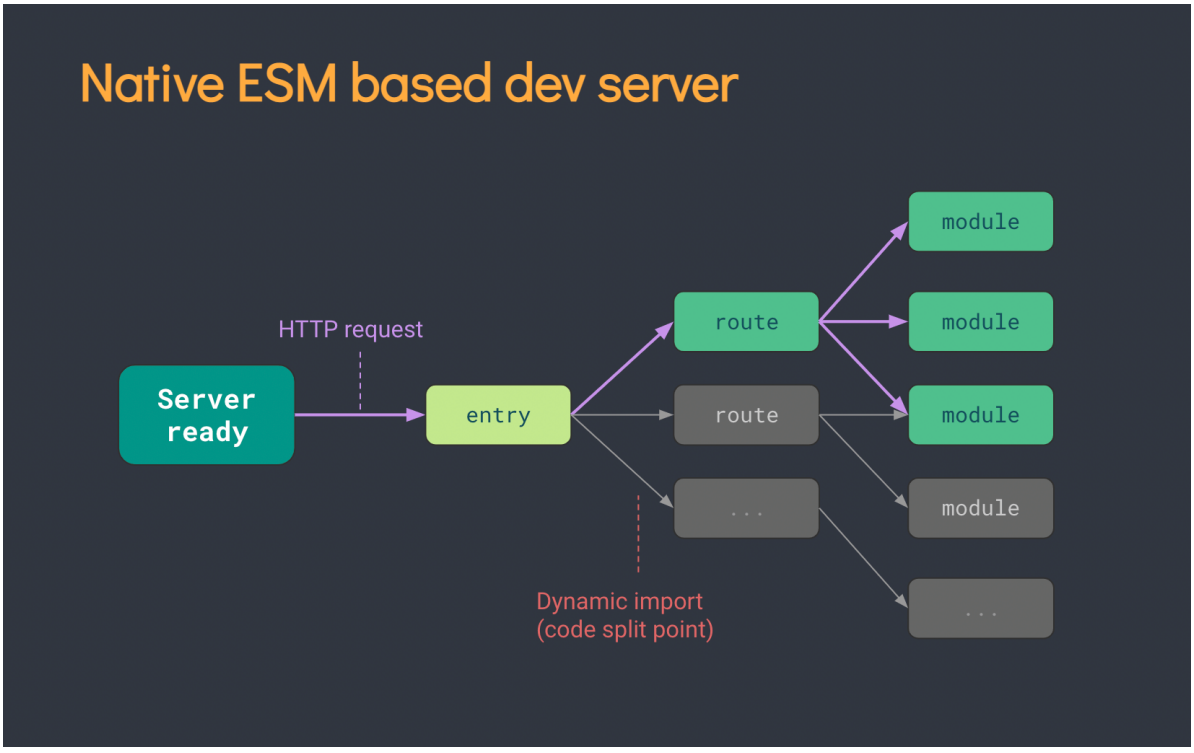
官方文档：**https://v3.cn.vuejs.org/guide/installation.html#vite**

vite 官网：**https://vitejs.cn**

- 什么是 vite？—— 新一代前端构建工具。
- 优势如下：
    - 开发环境中，无需打包操作，可快速的冷启动。

    - 轻量快速的热重载（HMR）。
    - 真正的按需编译，不再等待整个应用编译完成。
- 传统构建 与 vite 构建对比图






```
## 创建工程
npm init vite-app <project-name>
## 进入工程目录
cd <project-name>
## 安装依赖
npm install
## 运行
npm run dev
```

二、常用 Composition API

官方文档: https://v3.cn.vuejs.org/guide/composition-api-introduction.html

1.拉开序幕的 setup

1. 理解：Vue3.0 中一个新的配置项，值为一个函数。
2. setup 是所有 **Composition API（组合API）** ***“表演的舞台”*** 。
3. 组件中所用到的：数据、方法等等，均要配置在 setup 中。
4. setup 函数的两种返回值：
    1. 若返回一个对象，则对象中的属性、方法, 在模板中均可以直接使用。（重点关注！）
    2. 若返回一个渲染函数：则可以自定义渲染内容。（了解）
5. 注意点：
    1. 尽量不要与 Vue2.x 配置混用
        - Vue2.x 配置（data、methos、computed...）中**可以访问到** setup 中的属性、方法。
        - 但在 setup 中**不能访问到** Vue2.x 配置（data、methos、computed...）。
        - 如果有重名, setup 优先。
    2. setup 不能是一个 async 函数，因为返回值不再是 return 的对象, 而是 promise, 模板看不到 return 对象中的属性。（**后期也可以返回一个 `Promise` 实例，但需要 `Suspense` 和异步组件的配合**）

```vue
<template>
    <h1>一个人的信息</h1>
    <h2>姓名：{{name}}</h2>
    <h2>年龄：{{age}}</h2>
    <h2>性别：{{sex}}</h2>
    <h2>a的值是：{{a}}</h2>
    <button @click="sayHello">说话(Vue3所配置的—sayHello)</button>
</template>

<script>
    setup(){
        //数据
        let name = '张三'
        let age = 18
        let a = 200

        //方法
        function sayHello(){
            alert(`我叫${name}，我${age}岁了，你好啊！`)
        }

        //返回一个对象（常用）
        return {
            name,
            age,
            sayHello,
            test2,
            a
        }

        //返回一个函数（渲染函数）
        // return ()=> h('h1','尚硅谷')
    }
```

```
</script>
```

2.ref 函数

- **作用**: 定义一个响应式的数据
- **语法**: `const xxx = ref(initValue)`
  - 创建一个包含响应式数据的**引用对象（reference 对象，简称 ref 对象）**。
  - JS 中操作数据： `xxx.value`
  - 模板中读取数据: 不需要 `.value`，直接： `<div>{{xxx}}</div>`
- **备注**:
  - 接收的数据可以是：基本类型、也可以是对象类型。
  - 基本类型的数据：响应式依然是靠 `Object.defineProperty()` 的 `get` 与 `set` 完成的。
  - 对象类型的数据：内部 ***“求助”*** 了 Vue3.0 中的一个新函数—— `reactive` 函数。

```vue
<template>
    <h1>一个人的信息</h1>
    <h2>姓名：{{name}}</h2>
    <h2>年龄：{{age}}</h2>
    <h3>工作种类：{{job.type}}</h3>
    <h3>工作薪水：{{job.salary}}</h3>
    <button @click="changeInfo">修改人的信息</button>
</template>

<script>
    import {ref} from 'vue'
    export default {
        name: 'App',
        setup(){
            //数据
            let name = ref('张三')
            let age = ref(18)
            let job = ref({
                type:'前端工程师',
                salary:'30K'
            })

            //方法
            function changeInfo(){
                // name.value = '李四'
                // age.value = 48
                console.log(job.value)
                // job.value.type = 'UI设计师'
                // job.value.salary = '60K'
                // console.log(name,age)
            }

            //返回一个对象（常用）
            return {
                name,
                age,
                job,
```

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

3.reactive 函数

- **作用**: 定义一个**对象类型**的响应式数据（基本类型不要用它，要用`ref`函数）
- **语法**：`const 代理对象= reactive(源对象)`接收一个对象（或数组），返回一个**代理对象（Proxy 的实例对象，简称 proxy 对象）**
- reactive 定义的响应式数据是“深层次的”。
- 内部基于 ES6 的 Proxy 实现，通过代理对象操作源对象内部数据进行操作。

```
<template>
    <h1>一个人的信息</h1>
    <h2>姓名：{{person.name}}</h2>
    <h2>年龄：{{person.age}}</h2>
    <h3>工作种类：{{person.job.type}}</h3>
    <h3>工作薪水：{{person.job.salary}}</h3>
    <h3>爱好：{{person.hobby}}</h3>
    <h3>测试的数据c：{{person.job.a.b.c}}</h3>
    <button @click="changeInfo">修改人的信息</button>
</template>

<script>
    import {reactive} from 'vue'
    export default {
        name: 'App',
        setup(){
            //数据
            let person = reactive({
                name:'张三',
                age:18,
                job:{
                    type:'前端工程师',
                    salary:'30K',
                    a:{
                        b:{
                            c:666
                        }
                    }
                },
                hobby:['抽烟','喝酒','烫头']
            })

            //方法
            function changeInfo(){
                person.name = '李四'
                person.age = 48
                person.job.type = 'UI设计师'
                person.job.salary = '60K'
                person.job.a.b.c = 999
```

```
                person.hobby[0] = '学习'
            }

            //返回一个对象（常用）
            return {
                person,
                changeInfo
            }
        }
    }
</script>
```

4.Vue3.0 中的响应式原理

vue2.x 的响应式

- **实现原理**：
  - **对象类型**：通过 `Object.defineProperty()` 对属性的读取、修改进行拦截（数据劫持）。
  - **数组类型**：通过重写更新数组的一系列方法来实现拦截。（对数组的变更方法进行了包裹）。

```
Object.defineProperty(data, 'count', {
    get () {},
    set () {}
})
```

- **存在问题**：
  - 新增属性、删除属性, 界面不会更新 。
  - 直接通过下标修改数组, 界面不会自动更新 。

Vue3.0 的响应式

- **实现原理**:
  - 通过 Proxy（代理）：拦截对象中任意属性的变化, 包括：属性值的读写、属性的添加、属性的删除等。
  - 通过 Reflect（反射）：对源对象的属性进行操作。
  - MDN 文档中描述的 Proxy 与 Reflect：
    - Proxy：**https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Proxy**
    - Reflect：**https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Reflect**

```
new Proxy(data, {
    // 拦截读取属性值
    get (target, prop) {
        return Reflect.get(target, prop)
    },
    // 拦截设置属性值或添加新属性
    set (target, prop, value) {
```

```
            return Reflect.set(target, prop, value)
        },
        // 拦截删除属性
        deleteProperty (target, prop) {
            return Reflect.deleteProperty(target, prop)
        }
    })

    proxy.name = 'tom'
```

5.reactive 对比 ref

- **从定义数据角度对比**：
  - ref 用来定义：**基本类型数据**。
  - reactive 用来定义：**对象（或数组）类型数据**。
  - **备注**：ref 也可以用来定义**对象（或数组）类型数据**，它内部会自动通过 `reactive` 转为**代理对象**。
- **从原理角度对比**：
  - ref 通过 `Object.defineProperty()` 的 `get` 与 `set` 来实现响应式（数据劫持）。
  - reactive 通过使用 **Proxy** 来实现响应式（数据劫持），并通过 **Reflect** 操作**源对象**内部的数据。
- **从使用角度对比**：
  - ref 定义的数据：操作数据**需要** `.value`，读取数据时模板中直接读取**不需要** `.value`。
  - reactive 定义的数据：操作数据与读取数据：**均不需要** `.value`。

6.setup 的两个注意点

- setup 执行的时机
  - 在 beforeCreate 之前执行一次，this 是 undefined。
- setup 的参数
  - props：值为对象，包含：组件外部传递过来，且组件内部声明接收了的属性。
  - context：上下文对象
    - attrs: 值为对象，包含：组件外部传递过来，但没有在 props 配置中声明的属性, 相当于 `this.$attrs`。
    - slots: 收到的插槽内容, 相当于 `this.$slots`。
    - emit: 分发自定义事件的函数, 相当于 `this.$emit`。

7.计算属性与监视

1.computed 函数

- 与 Vue2.x 中 computed 配置功能一致
- 写法

```
import {computed} from 'vue'

setup(){
    ...
    //计算属性——简写
    let fullName = computed(()=>{
        return person.firstName + '-' + person.lastName
    })
    //计算属性——完整
    let fullName = computed({
        get(){
            return person.firstName + '-' + person.lastName
        },
        set(value){
            const nameArr = value.split('-')
            person.firstName = nameArr[0]
            person.lastName = nameArr[1]
        }
    })
}
```

2.watch 函数

- 与 Vue2.x 中 watch 配置功能一致
- 两个小“坑”：
  - 监视 `reactive` 定义的响应式数据时：`oldValue` 无法正确获取、强制开启了深度监视（`deep` 配置失效）。
  - 监视 `reactive` 定义的响应式数据中**某个属性**时：`deep` 配置有效。

```
//情况一：监视ref定义的响应式数据
watch(sum,(newValue,oldValue)=>{
    console.log('sum变化了',newValue,oldValue)
},{immediate:true})

//情况二：监视多个ref定义的响应式数据
watch([sum,msg],(newValue,oldValue)=>{
    console.log('sum或msg变化了',newValue,oldValue)
})

/* 情况三：监视reactive定义的响应式数据
            若watch监视的是reactive定义的响应式数据，则无法正确获得oldValue！！
            若watch监视的是reactive定义的响应式数据，则强制开启了深度监视
*/
watch(person,(newValue,oldValue)=>{
    console.log('person变化了',newValue,oldValue)
},{immediate:true,deep:false}) //此处的deep配置不再奏效

//情况四：监视reactive定义的响应式数据中的某个属性
watch(()=>person.job,(newValue,oldValue)=>{
```

```
        console.log('person的job变化了',newValue,oldValue)
    },{immediate:true,deep:true})

    //情况五：监视reactive定义的响应式数据中的某些属性
    watch([()⇒person.job,()⇒person.name],(newValue,oldValue)⇒{
        console.log('person的job变化了',newValue,oldValue)
    },{immediate:true,deep:true})

    //特殊情况
    watch(()⇒person.job,(newValue,oldValue)⇒{
        console.log('person的job变化了',newValue,oldValue)
    },{deep:true}) //此处由于监视的是reactive素定义的对象中的某个属性，所以deep配置有效
```

watch 时的 value 问题：

- watch 监视 ref 所定义的基本数据类型时，不需要加 `.value`，监视 ref 定义的对象时需要加 `.value` 或者配置深度监视 `deep: true`

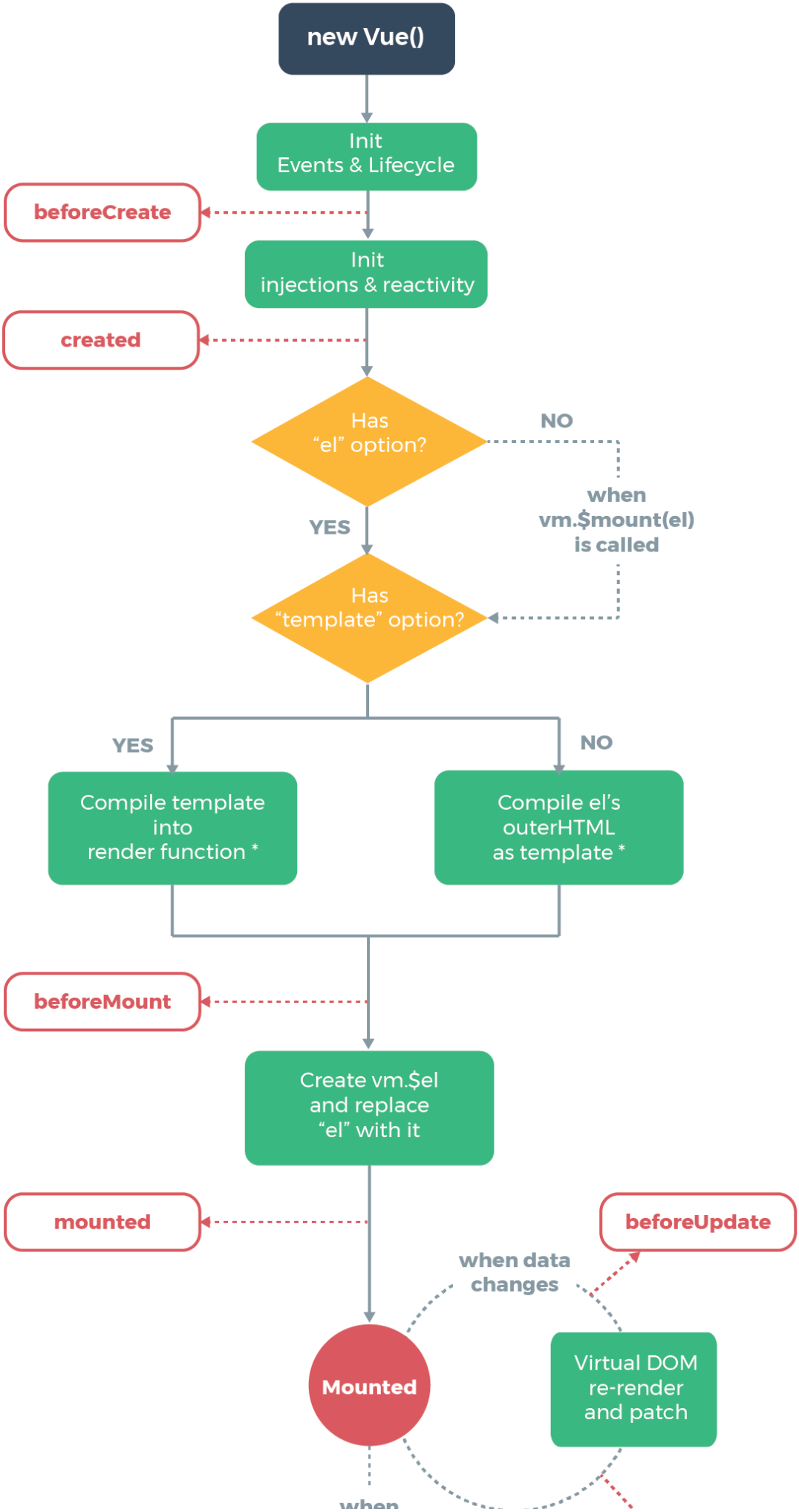
3.watchEffect 函数

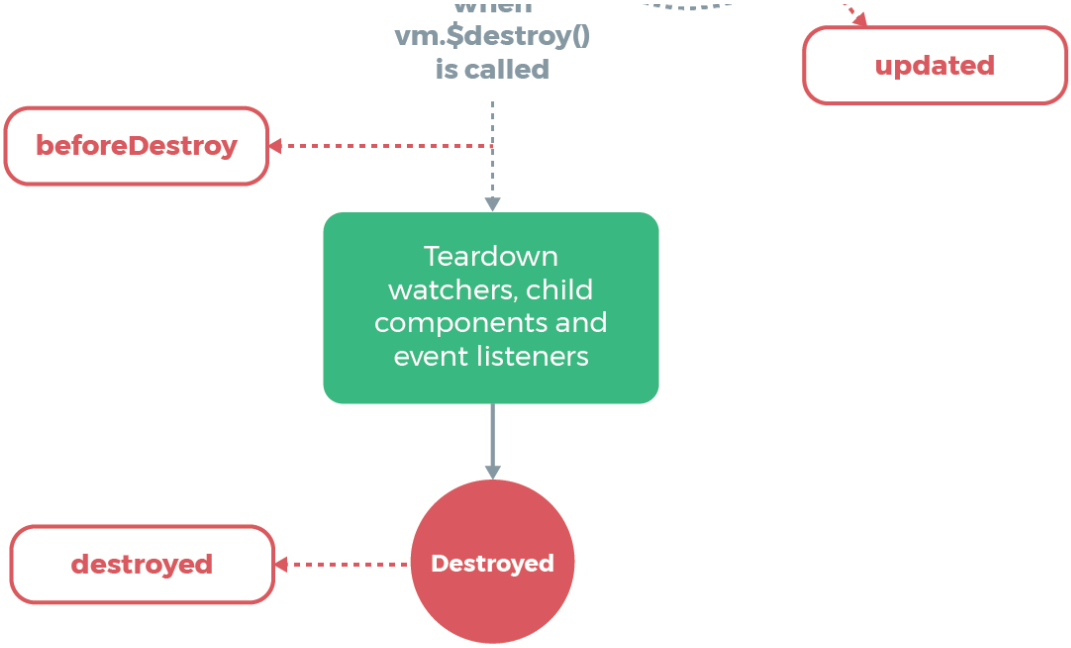
- `watch` 的套路是：既要指明监视的属性，也要指明监视的回调。
- `watchEffect` 的套路是：不用指明监视哪个属性，监视的回调中用到哪个属性，那就监视哪个属性。
- `watchEffect` 有点像 `computed`：
  - 但 `computed` 注重的计算出来的值（回调函数的返回值），所以必须要写返回值。
  - 而 `watchEffect` 更注重的是过程（回调函数的函数体），所以不用写返回值。

```
//watchEffect所指定的回调中用到的数据只要发生变化，则直接重新执行回调。
watchEffect(()⇒{
    const x1 = sum.value
    const x2 = person.age
    console.log('watchEffect配置的回调执行了')
})
```

8.生命周期

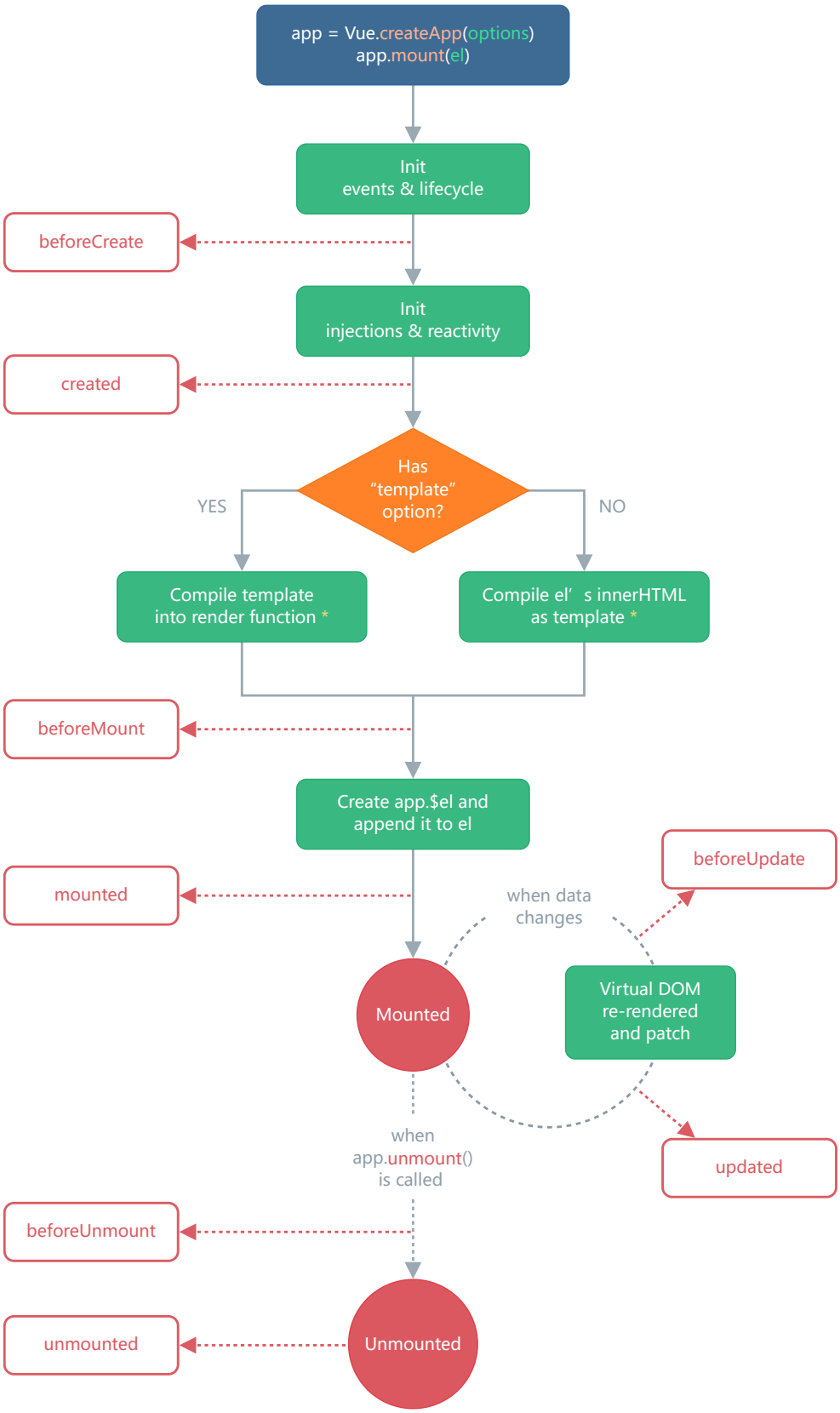
vue2.x 的生命周期


new Vue()
Init
Events & Lifecycle
beforeCreate
Init
injections & reactivity
created
Has
“el” option?
NO
when
vm.$mount(el)
is called
YES
Has
“template” option?
YES
NO
Compile template
into
render function *
Compile el’s
outerHTML
as template *
beforeMount
Create vm.$el
and replace
“el” with it
mounted
beforeUpdate
when data
changes
Mounted
Virtual DOM
re-render
and patch
when



* template compilation is performed ahead-of-time if using
a build step, e.g. single-file components

vue3.0 的生命周期



* Template compilation is performed ahead-of-time if using a build step, e.g., with single-file components.

- Vue3.0 中可以继续使用 Vue2.x 中的生命周期钩子，但有有两个被更名：
  - `beforeDestroy` 改名为 `beforeUnmount`

    - `destroyed` 改名为 `unmounted`
- Vue3.0 也提供了 Composition API 形式的生命周期钩子，与 Vue2.x 中钩子对应关系如下：
    - `beforeCreate` ==> `setup()`
    - `created` ==> `setup()`
    - `beforeMount` ==> `onBeforeMount`
    - `mounted` ==> `onMounted`
    - `beforeUpdate` ==> `onBeforeUpdate`
    - `updated` ==> `onUpdated`
    - `beforeUnmount` ==> `onBeforeUnmount`
    - `unmounted` ==> `onUnmounted`

9.自定义 hook 函数

- 什么是 hook? —— 本质是一个函数，把 setup 函数中使用的 Composition API 进行了封装。
- 类似于 vue2.x 中的 mixin。
- 自定义 hook 的优势: 复用代码, 让 setup 中的逻辑更清楚易懂。

`src/hooks/usePoint.js`

```
import {reactive,onMounted,onBeforeUnmount} from 'vue'
export default function (){
    //实现鼠标“打点”相关的数据
    let point = reactive({
        x:0,
        y:0
    })

    //实现鼠标“打点”相关的方法
    function savePoint(event){
        point.x = event.pageX
        point.y = event.pageY
        console.log(event.pageX,event.pageY)
    }

    //实现鼠标“打点”相关的生命周期钩子
    onMounted(()=>{
        window.addEventListener('click',savePoint)
    })

    onBeforeUnmount(()=>{
        window.removeEventListener('click',savePoint)
    })

    return point
}
```

在其他组件中引入：`import usePoint from '../hooks/usePoint'`，然后使用 `let point = usePoint()`，最后返回即可 `return {point}`

10.toRef 和 toRefs

- **作用**：创建一个 ref 对象，其 value 值指向另一个对象中的某个属性。
- **语法**：`const name = toRef(person,'name')`
- **应用**： 要将响应式对象中的某个属性单独提供给外部使用时 。
- **扩展**：`toRefs` 与 `toRef` 功能一致，但 可以批量创建多个 ref 对象，但只考虑对象第一层的数据 ，语法：`toRefs(person)`
- toRefs 的返回值应为：

```
return {
    ...toRefs(person)
}
```

三、其它 Composition API

1.shallowReactive 与 shallowRef

- `shallowReactive`： 只处理对象最外层属性的响应式（浅响应式） 。
- `shallowRef`： 只处理基本数据类型的响应式, 不进行对象的响应式处理 。
- **什么时候使用?**
  - 如果有一个对象数据，结构比较深, 但变化时只是外层属性变化 ==> `shallowReactive`。
  - 如果有一个对象数据，后续功能不会修改该对象中的属性，而是生新的对象来替换 ==> `shallowRef`。

2.readonly 与 shallowReadonly

- `readonly`：让一个响应式数据变为只读的（深只读） 。
- `shallowReadonly`： 让一个响应式数据变为只读的（浅只读） 。
- **应用场景**: 不希望数据被修改时 。

3.toRaw 与 markRaw

- `toRaw`：
  - **作用**：将一个由 `reactive` 生成的**响应式对象**转为**普通对象**。
  - **使用场景**： 用于读取响应式对象对应的普通对象，对这个普通对象的所有操作，不会引起页面更新 。
- `markRaw`：
  - **作用**： 标记一个对象，使其永远不会再成为响应式对象 。
  - **应用场景**:
    1. 有些值不应被设置为响应式的，例如复杂的第三方类库等 。
    2. 当渲染具有不可变数据源的大列表时，跳过响应式转换可以提高性能 。

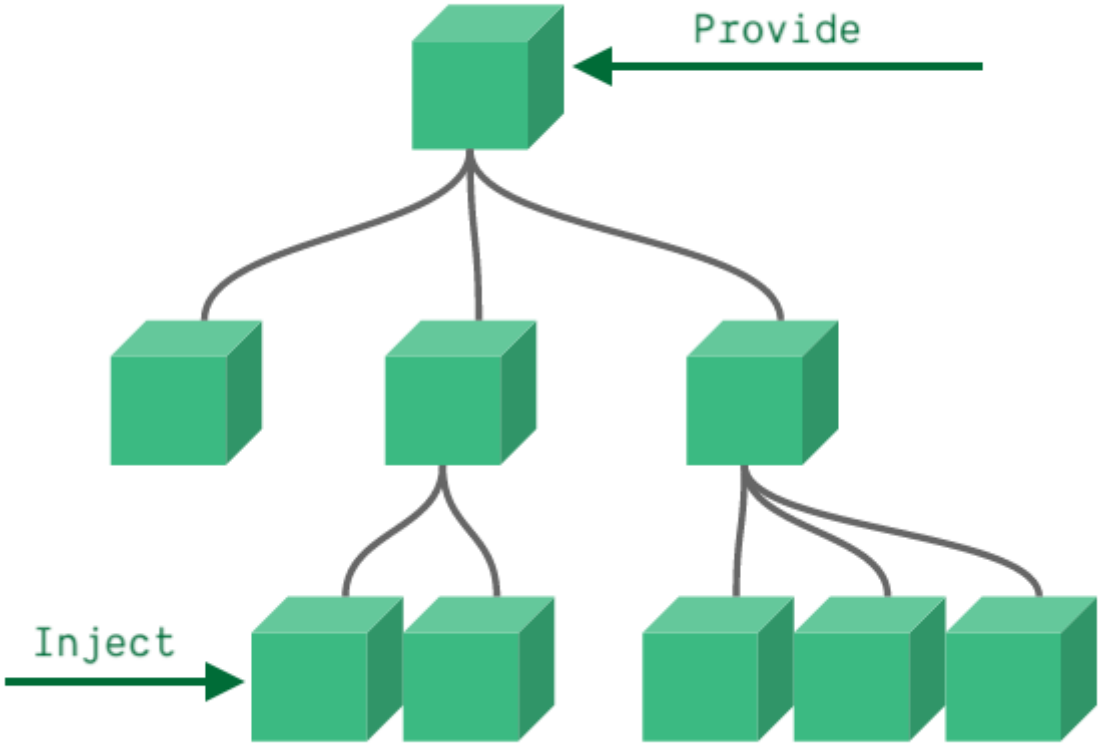
4.customRef

- **作用**：创建一个自定义的 ref，并对其依赖项跟踪和更新触发进行显式控制 。
- **实现防抖效果**：

```
<template>
    <input type="text" v-model="keyword">
    <h3>{{keyword}}</h3>
</template>

<script>
    import {ref,customRef} from 'vue'
    export default {
        name:'Demo',
        setup(){
            // let keyword = ref('hello') //使用Vue准备好的内置ref
            //自定义一个myRef
            function myRef(value,delay){
                let timer
                //通过customRef去实现自定义
                return customRef((track,trigger)=>{
                    return{
                        get(){
                            track() //告诉Vue这个value值是需要被“追踪”的，在return 之前调用
                            return value
                        },
                        set(newValue){
                            clearTimeout(timer)
                            timer = setTimeout(()=>{
                                value = newValue
                                trigger() //告诉Vue去更新界面
                            },delay)
                        }
                    }
                })
            }
            let keyword = myRef('hello',500) //使用程序员自定义的ref
            return {
                keyword
            }
        }
    }
</script>
```

5.provide 与 inject



- **作用**：实现**祖与后代组件间**通信
- **套路**：父组件有一个 `provide` 选项来提供数据，后代组件有一个 `inject` 选项来开始使用这些数据
- **具体写法**：
  1. **祖组件中**：

```
setup(){
    ......
    let car = reactive({name:'奔驰',price:'40万'})
    provide('car',car)
    ......
}
```

  2. **后代组件中**：

```
setup(props,context){
    ......
    const car = inject('car')
    return {car}
    ......
}
```

6.响应式数据的判断

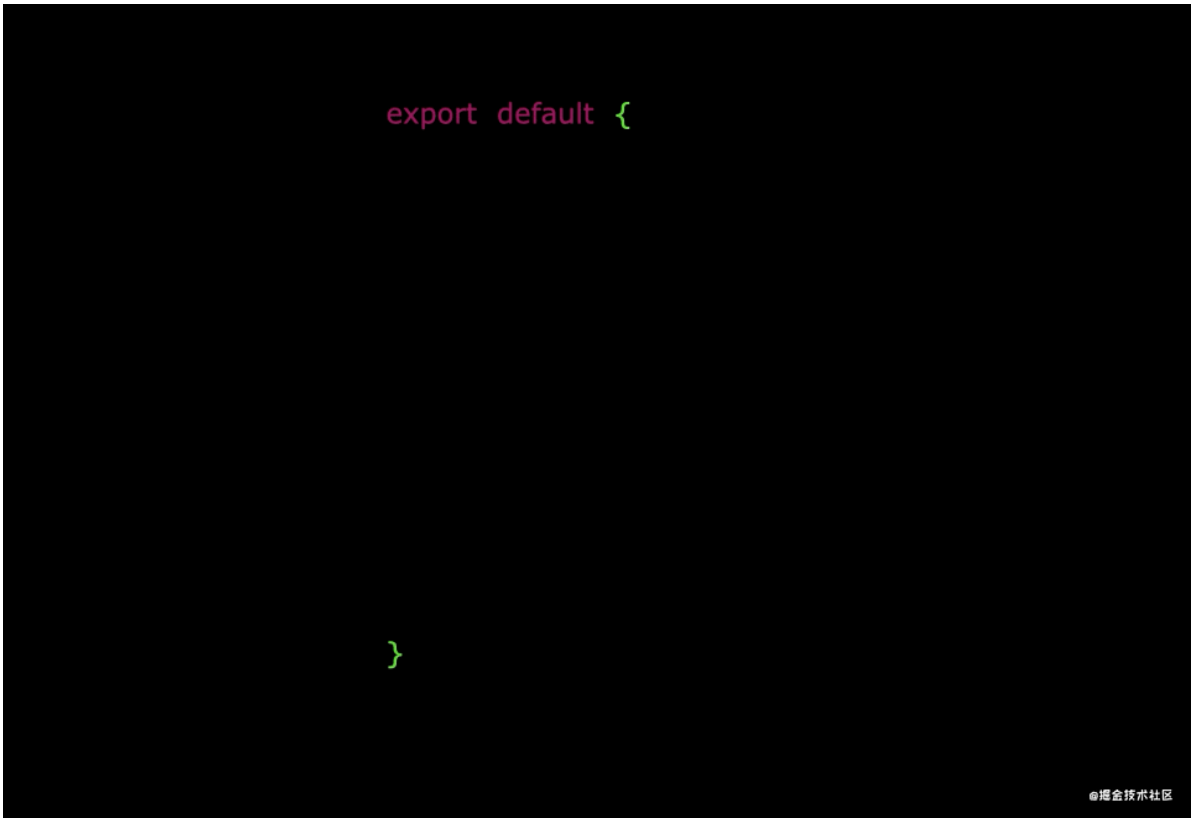
- `isRef`：检查一个值是否为一个 `ref` 对象
- `isReactive`：检查一个对象是否是由 `reactive` 创建的响应式代理
- `isReadonly`：检查一个对象是否是由 `readonly` 创建的只读代理

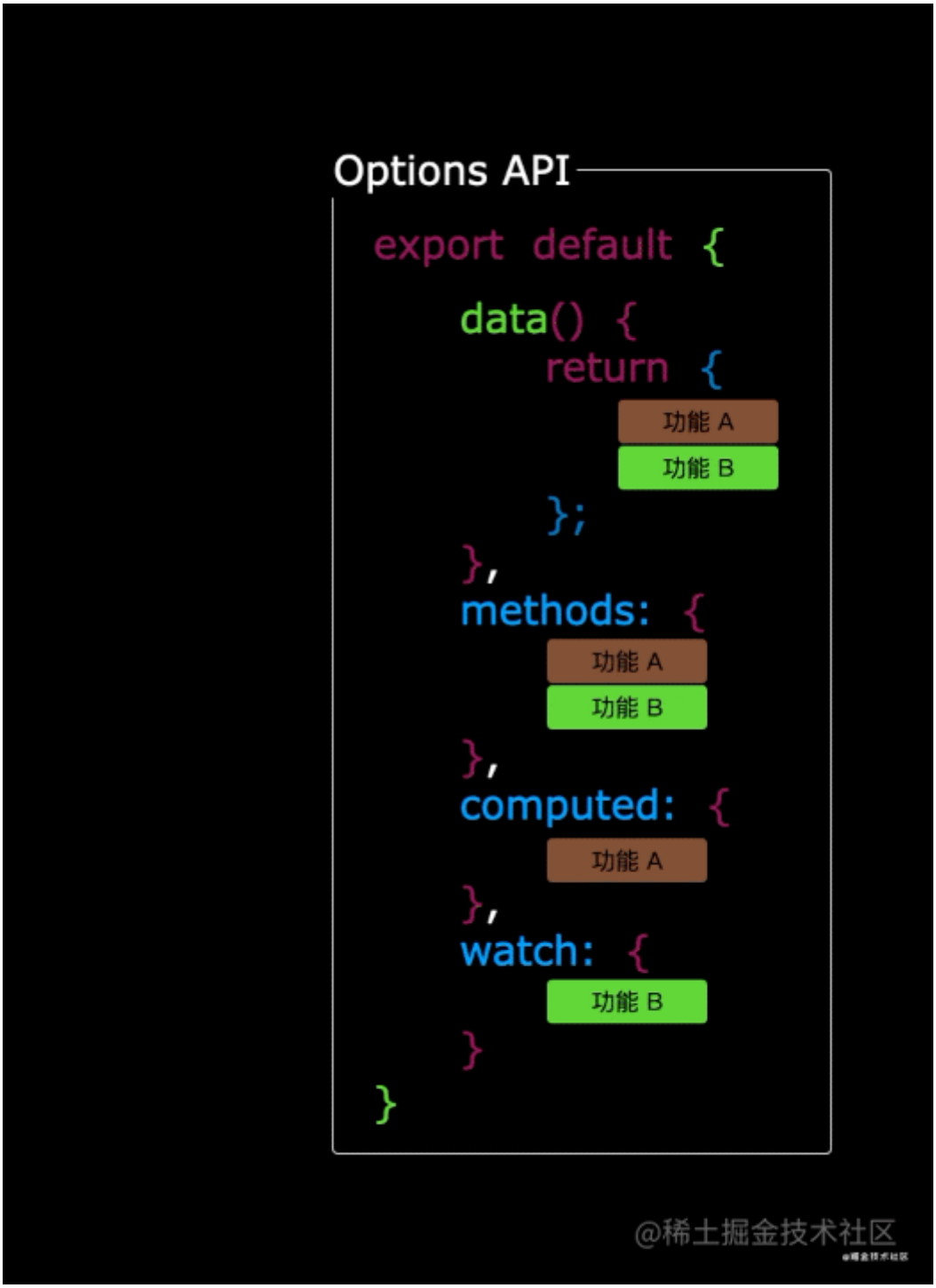
- `isProxy`：检查一个对象是否是由 `reactive` 或者 `readonly` 方法创建的代理

四、Composition API 的优势

1.Options API 存在的问题

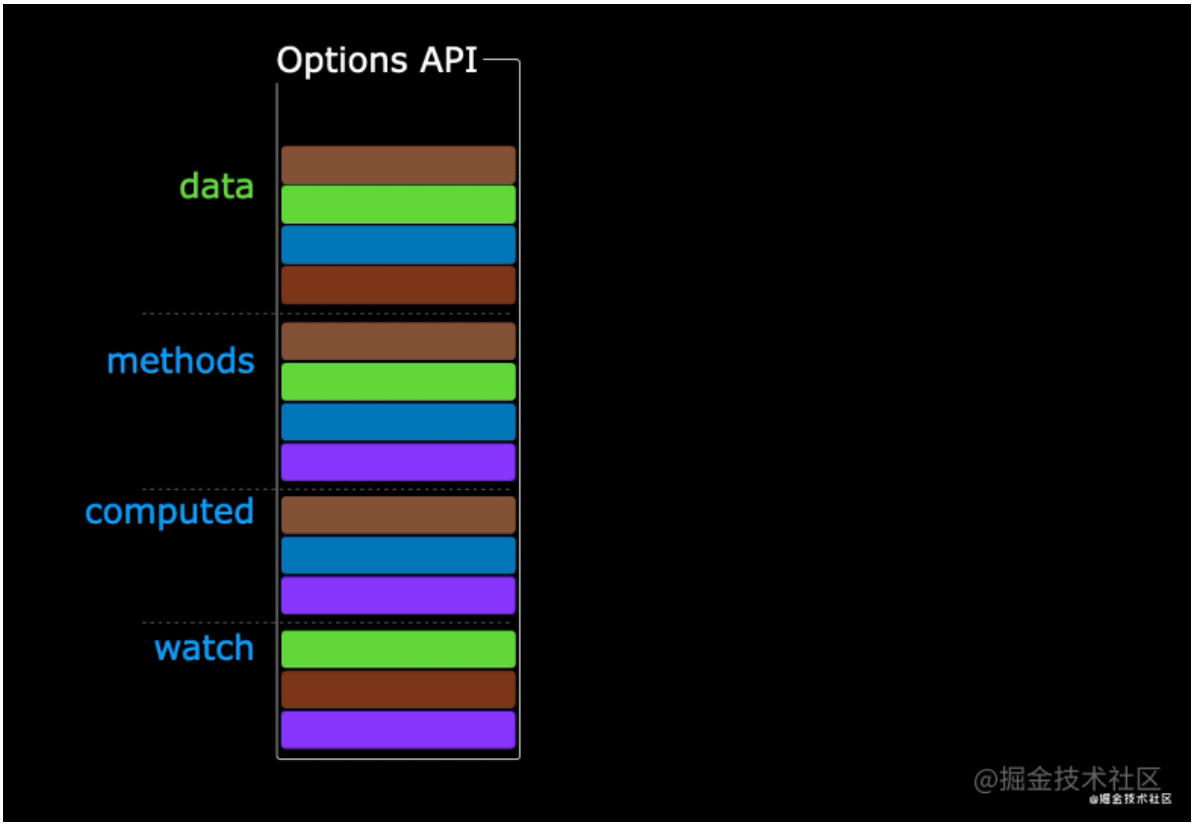
使用传统 OptionsAPI 中，新增或者修改一个需求，就需要分别在 data，methods，computed 里修改 。

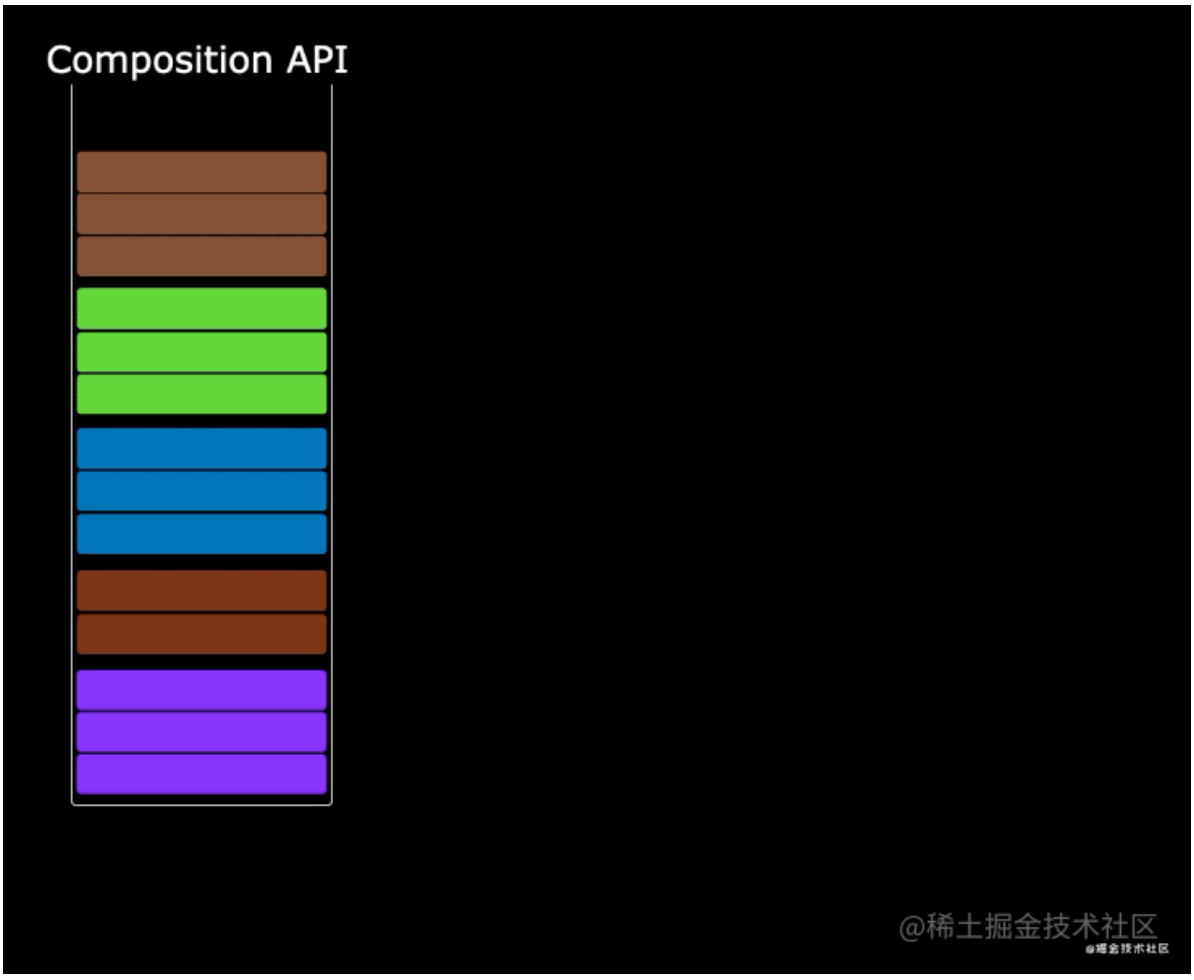




2.Composition API 的优势

我们可以更加优雅的组织我们的代码，函数。让相关功能的代码更加有序的组织在一起。





五、新的组件

1.Fragment 组件

- 在 Vue2 中: **组件必须有一个根标签**
- 在 Vue3 中: **组件可以没有根标签**，内部会将多个标签包含在一个 Fragment 虚拟元素中
- **好处**：减少标签层级, 减小内存占用

2.Teleport 组件

- 什么是 Teleport? —— `Teleport` 是一种能够将我们的**组件 html 结构**移动到指定位置的技术。

```
<teleport to="移动位置">
    <div v-if="isShow" class="mask">
        <div class="dialog">
            <h3>我是一个弹窗</h3>
            <button @click="isShow = false">关闭弹窗</button>
        </div>
    </div>
</teleport>
```

3.Suspense 组件

- 等待异步组件时渲染一些额外内容，让应用有更好的用户体验
- **使用步骤**：
  - **异步引入组件**

```
// import Child from './components/Child' // 静态引入
import {defineAsyncComponent} from 'vue'
const Child = defineAsyncComponent(()=>import('./components/Child.vue'))
// 异步引入（动态引入）
```

  - **使用`Suspense`包裹组件，并配置好`default` 与 `fallback`**

```
<template>
    <div class="app">
        <h3>我是App组件</h3>
        <Suspense>
            <template v-slot:default>
                <Child/>
            </template>
            <template v-slot:fallback>
                <h3>加载中.....</h3>
            </template>
        </Suspense>
    </div>
</template>
```

六、其他

1.全局 API 的转移

- Vue 2.x 有许多全局 API 和配置。
  - 例如：**注册全局组件、注册全局指令等。**

```
//注册全局组件
Vue.component('MyButton', {
  data: () => ({
    count: 0
  }),
  template: '<button @click="count++">Clicked {{ count }} times.
</button>'
})

//注册全局指令
Vue.directive('focus', {
  inserted: el => el.focus()
}
```

- Vue3.0 中对这些 API 做出了调整：
  - **将全局的API，即：`Vue.xxx` 调整到应用实例（`app`）上**

2.x 全局 API（`Vue`）	3.x 实例 API（`app`）
Vue.config.xxxx	app.config.xxxx
Vue.config.productionTip	**移除**
Vue.component	app.component
Vue.directive	app.directive
Vue.mixin	app.mixin
Vue.use	app.use
Vue.prototype	app.config.globalProperties

2.其他改变

- `data 选项应始终被声明为一个函数`。
- **过度类名的更改**：
  - **Vue2.x 写法**

```
.v-enter,
.v-leave-to {
  opacity: 0;
}
.v-leave,
.v-enter-to {
  opacity: 1;
}
```

- **Vue3.x 写法**

```
.v-enter-from,
.v-leave-to {
  opacity: 0;
}

.v-leave-from,
.v-enter-to {
  opacity: 1;
}
```

- **移除 `keyCode` 作为 `v-on` 的修饰符，同时也不再支持 `config.keyCodes`**
- **移除 `v-on.native` 修饰符**
  - **父组件中绑定事件**

```
<my-component
  v-on:close="handleComponentEvent"
  v-on:click="handleNativeClickEvent"
/>
```

  - **子组件中声明自定义事件**

```
<script>
  export default {
    emits: ['close']
  }
</script>
```

- **移除**过滤器（`filter`）

> 过滤器虽然这看起来很方便，但它需要一个自定义语法，打破大括号内表达式是 “只是 JavaScript” 的假设，这不仅有学习成本，而且有实现成本！**建议用** 方法调用 **或** 计算属性 **去替换过滤器。**

- ……