腾讯前端常考vue面试题（必备）

虚拟DOM的优劣如何?

优点:

- 保证性能下限: 虚拟DOM可以经过diff找出最小差异,然后批量进行patch,这种操作虽然比不上手动优化,但是比起粗暴的DOM操作性能要好很多,因此虚拟DOM可以保证性能下限
- 无需手动操作DOM: 虚拟DOM的diff和patch都是在一次更新中自动进行的,我们无需手动操作DOM,极大提高开发效率
- 跨平台: 虚拟DOM本质上是JavaScript对象,而DOM与平台强相关,相比之下虚拟DOM可以进行更方便地跨平台操作,例如服务器渲染、移动端开发等等

缺点:

- 无法进行极致优化: 在一些性能要求极高的应用中虚拟DOM无法进行针对性的极致优化,比如VScode采用直接手动操作DOM的方式进行极端的性能优化

如果让你从零开始写一个vue路由，说说你的思路

思路分析：

首先思考 `vue` 路由要解决的问题：用户点击跳转链接内容切换，页面不刷新。

- 借助 `hash` 或者h `istory api` 实现 `url` 跳转页面不刷新
- 同时监听 `hashchange` 事件或者 `popstate` 事件处理跳转
- 根据 `hash` 值或者 `state` 值从 `routes` 表中匹配对应 `component` 并渲染

回答范例：

一个 `SPA` 应用的路由需要解决的问题是 **页面跳转内容改变同时不刷新** ，同时路由还需要以插件形式存在，所以：

1. 首先我会定义一个 `createRouter` 函数，返回路由器实例，实例内部做几件事

- 保存用户传入的配置项

- 监听 `hash` 或者 `popstate` 事件
- 回调里根据 `path` 匹配对应路由

2. 将 `router` 定义成一个 `Vue` 插件，即实现 `install` 方法，内部做两件事

- 实现两个全局组件： `router-link` 和 `router-view` ，分别实现页面跳转和内容显示
- 定义两个全局变量： `$route` 和 `$router` ，组件内可以访问当前路由和路由器实例

你有对 Vue 项目进行哪些优化?

(1) 代码层面的优化

- v-if 和 v-show 区分使用场景
- computed 和 watch 区分使用场景
- v-for 遍历必须为 item 添加 key，且避免同时使用 v-if
- 长列表性能优化
- 事件的销毁
- 图片资源懒加载
- 路由懒加载
- 第三方插件的按需引入
- 优化无限列表性能
- 服务端渲染 SSR or 预渲染

(2) Webpack 层面的优化

- Webpack 对图片进行压缩
- 减少 ES6 转为 ES5 的冗余代码
- 提取公共代码
- 模板预编译
- 提取组件的 CSS
- 优化 SourceMap
- 构建结果输出分析
- Vue 项目的编译优化

(3) 基础的 Web 技术的优化

- 开启 gzip 压缩
- 浏览器缓存
- CDN 的使用

- 使用 Chrome Performance 查找性能瓶颈

写过自定义指令吗 原理是什么

指令本质上是装饰器，是 vue 对 HTML 元素的扩展，给 HTML 元素增加自定义功能。vue 编译 DOM 时，会找到指令对象，执行指令的相关方法。

自定义指令有五个生命周期（也叫钩子函数），分别是 bind、inserted、update、componentUpdated、unbind

```markdown
1. bind：只调用一次，指令第一次绑定到元素时调用。在这里可以进行一次性的初始化设置。
2. inserted：被绑定元素插入父节点时调用 (仅保证父节点存在，但不一定已被插入文档中)。
3. update：被绑定于元素所在的模板更新时调用，而无论绑定值是否变化。通过比较更新前后的绑定值，可以忽略不必要的
4. componentUpdated：被绑定元素所在模板完成一次更新周期时调用。
5. unbind：只调用一次，指令与元素解绑时调用。
```

原理

1.在生成 ast 语法树时，遇到指令会给当前元素添加 directives 属性

2.通过 genDirectives 生成指令代码

3.在 patch 前将指令的钩子提取到 cbs 中,在 patch 过程中调用对应的钩子

4.当执行指令对应钩子函数时，调用对应指令定义的方法

created和mounted的区别

- created:在模板渲染成html前调用，即通常初始化某些属性值，然后再渲染成视图。
- mounted:在模板渲染成html后调用，通常是初始化页面完成后，再对html的dom节点进行一些需要的操作。

说说你对 proxy 的理解，Proxy 相比于 defineProperty 的优势

`Object.defineProperty()` **的问题主要有三个：**

- **不能监听数组的变化**： 无法监控到数组下标的变化，导致通过数组下标添加元素，不能实时响应
- **必须遍历对象的每个属性**： 只能劫持对象的属性，从而需要对每个对象，每个属性进行遍历，如果属性值是对象，还需要深度遍历。 `Proxy` 可以劫持整个对象，并返回一个新的对象
- **必须深层遍历嵌套的对象**

Proxy的优势如下:

- 针对对象： **针对整个对象，而不是对象的某个属性** ，所以也就不需要对 `keys` 进行遍历
- 支持数组： `Proxy` 不需要对数组的方法进行重载，省去了众多 hack，减少代码量等于减少了维护成本，而且标准的就是最好的
- `Proxy` 的第二个参数可以有 `13` 种拦截方：不限于 `apply` 、 `ownKeys` 、 `deleteProperty` 、 `has` 等等是 `Object.defineProperty` 不具备的
- `Proxy` 返回的是一个新对象,我们可以只操作新的对象达到目的,而 `Object.defineProperty` 只能遍历对象属性直接修改
- `Proxy` 作为新标准将受到浏览器厂商重点持续的性能优化，也就是传说中的新标准的性能红利

proxy详细使用点击查看(opens new window)

Object.defineProperty的优势如下:

> 兼容性好，支持 `IE9` ，而 `Proxy` 的存在浏览器兼容性问题,而且无法用 `polyfill` 磨平

defineProperty的属性值有哪些

javascript 复制代码

```
Object.defineProperty(obj, prop, descriptor)

// obj 要定义属性的对象
// prop 要定义或修改的属性的名称
// descriptor 要定义或修改的属性描述符

Object.defineProperty(obj,"name",{
  value:"poetry", // 初始值
  writable:true, // 该属性是否可写入
  enumerable:true, // 该属性是否可被遍历得到（for...in， Object.keys等）
  configurable:true, // 定该属性是否可被删除，且除writable外的其他描述符是否可被修改
  get: function() {},
  set: function(newVal) {}
})
```

相关代码如下

javascript 复制代码

```javascript
import { mutableHandlers } from "./baseHandlers"; // 代理相关逻辑
import { isObject } from "./util"; // 工具方法

export function reactive(target) {
  // 根据不同参数创建不同响应式对象
  return createReactiveObject(target, mutableHandlers);
}
function createReactiveObject(target, baseHandler) {
  if (!isObject(target)) {
    return target;
  }
  const observed = new Proxy(target, baseHandler);
  return observed;
}

const get = createGetter();
const set = createSetter();

function createGetter() {
  return function get(target, key, receiver) {
    // 对获取的值进行放射
    const res = Reflect.get(target, key, receiver);
    console.log("属性获取", key);
    if (isObject(res)) {
      // 如果获取的值是对象类型，则返回当前对象的代理对象
      return reactive(res);
    }
    return res;
  };
}
function createSetter() {
  return function set(target, key, value, receiver) {
    const oldValue = target[key];
    const hadKey = hasOwn(target, key);
    const result = Reflect.set(target, key, value, receiver);
    if (!hadKey) {
      console.log("属性新增", key, value);
    } else if (hasChanged(value, oldValue)) {
      console.log("属性值被修改", key, value);
    }
    return result;
  };
}
export const mutableHandlers = {
  get, // 当获取属性时调用此方法
```

```
  set, // 当修改属性时调用此方法
};
```

Proxy 只会代理对象的第一层，那么 Vue3 又是怎样处理这个问题的呢?

> 判断当前 Reflect.get的 返回值是否为 Object ，如果是则再通过 reactive 方法做代理， 这样就实现了深度观测。

监测数组的时候可能触发多次get/set，那么如何防止触发多次呢?

> 我们可以判断 key 是否为当前被代理对象 target 自身属性，也可以判断旧值与新值是否相等，只有满足以上两个条件之一时，才有可能执行 trigger

参考 前端进阶面试题详细解答

简述 mixin、extends 的覆盖逻辑

（1）mixin 和 extends mixin 和 extends均是用于合并、拓展组件的，两者均通过 mergeOptions 方法实现合并。

- mixins 接收一个混入对象的数组，其中混入对象可以像正常的实例对象一样包含实例选项，这些选项会被合并到最终的选项中。Mixin 钩子按照传入顺序依次调用，并在调用组件自身的钩子之前被调用。
- extends 主要是为了便于扩展单文件组件，接收一个对象或构造函数。

（2）mergeOptions 的执行过程

- 规范化选项（normalizeProps、normalizeInject、normalizeDirectives)
- 对未合并的选项，进行判断

javascript 复制代码

```
if (!child._base) {
  if (child.extends) {
    parent = mergeOptions(parent, child.extends, vm);
  }
  if (child.mixins) {
    for (let i = 0, l = child.mixins.length; i < l; i++) {
      parent = mergeOptions(parent, child.mixins[i], vm);
    }
  }
```

```
}
```

- 合并处理。根据一个通用 Vue 实例所包含的选项进行分类逐一判断合并，如 props、data、 methods、watch、computed、生命周期等，将合并结果存储在新定义的 options 对象里。
- 返回合并结果 options。

MVC 和 MVVM 区别

MVC

MVC 全名是 Model View Controller，是模型(model) - 视图(view) - 控制器(controller)的缩写，一种软件设计典范

- Model（模型）：是应用程序中用于处理应用程序数据逻辑的部分。通常模型对象负责在数据库中存取数据
- View（视图）：是应用程序中处理数据显示的部分。通常视图是依据模型数据创建的
- Controller（控制器）：是应用程序中处理用户交互的部分。通常控制器负责从视图读取数据，控制用户输入，并向模型发送数据

MVC 的思想：一句话描述就是 Controller 负责将 Model 的数据用 View 显示出来，换句话说就是在 Controller 里面把 Model 的数据赋值给 View。

MVVM

MVVM 新增了 VM 类

- ViewModel 层：做了两件事达到了数据的双向绑定 一是将【模型】转化成【视图】，即将后端传递的数据转化成所看到的页面。实现的方式是：数据绑定。二是将【视图】转化成【模型】，即将所看到的页面转化成后端的数据。实现的方式是：DOM 事件监听。

MVVM 与 MVC 最大的区别就是：它实现了 View 和 Model 的自动同步，也就是当 Model 的属性改变时，我们不用再自己手动操作 Dom 元素，来改变 View 的显示，而是改变属性后该属性对应 View 层显示会自动改变（对应Vue数据驱动的思想）

整体看来，MVVM 比 MVC 精简很多，不仅简化了业务与界面的依赖，还解决了数据频繁更新的问题，不用再用选择器操作 DOM 元素。因为在 MVVM 中，View 不知道 Model 的存在，Model 和 ViewModel 也观察不到 View，这种低耦合模式提高代码的可重用性

> 注意：Vue 并没有完全遵循 MVVM 的思想 这一点官网自己也有说明

那么问题来了 为什么官方要说 Vue 没有完全遵循 MVVM 思想呢?

> - 严格的 MVVM 要求 View 不能和 Model 直接通信，而 Vue 提供了$refs 这个属性，让 Model 可以直接操作 View，违反了这一规定，所以说 Vue 没有完全遵循 MVVM。

Vue 中 computed 和 watch 有什么区别?

计算属性 computed： （1）**支持缓存**，只有依赖数据发生变化时，才会重新进行计算函数；（2）计算属性内**不支持异步操作**； （3）计算属性的函数中**都有一个 get**(默认具有，获取计算属性)**和 set**(手动添加，设置计算属性)方法； （4）计算属性是自动监听依赖值的变化，从而动态返回内容。

侦听属性 watch： （1）**不支持缓存**，只要数据发生变化，就会执行侦听函数； （2）侦听属性内**支持异步操作**； （3）侦听属性的值**可以是一个对象，接收 handler 回调，deep，immediate 三个属性**； （3）监听是一个过程，在监听的值变化时，可以触发一个回调，并**做一些其他事情。**

assets和static的区别

相同点： `assets` 和 `static` 两个都是存放静态资源文件。项目中所需要的资源文件图片，字体图标，样式文件等都可以放在这两个文件下，这是相同点

不相同点： `assets` 中存放的静态资源文件在项目打包时，也就是运行 `npm run build` 时会将 `assets` 中放置的静态资源文件进行打包上传，所谓打包简单点可以理解为压缩体积，代码格式化。而压缩后的静态资源文件最终也都会放置在 `static` 文件中跟着 `index.html` 一同上传至服务器。 `static` 中放置的静态资源文件就不会要走打包压缩格式化等流程，而是直接进入打包好的目录，直接上传至服务器。因为避免了压缩直接进行上传，在打包时会提高一定的效率，但是 `static` 中的资源文件由于没有进行压缩等操作，所以文件的体积也就相对于 `assets` 中打包后的文件提交较大点。在服务器中就会占据更大的空间。

建议： 将项目中 `template` 需要的样式文件js文件等都可以放置在 `assets` 中，走打包这一流程。减少体积。而项目中引入的第三方的资源文件如 `iconfoont.css` 等文件可以放置在 `static` 中，因为这些引入的第三方文件已经经过处理，不再需要处理，直接上传。

用VNode来描述一个DOM结构

虚拟节点就是用一个对象来描述一个真实的DOM元素。首先将 `template` （真实DOM）先转成 `ast` ， `ast` 树通过 `codegen` 生成 `render` 函数， `render` 函数里的 `_c` 方法将它转为虚拟dom

Vue的性能优化有哪些

（1）编码阶段

- 尽量减少data中的数据，data中的数据都会增加getter和setter，会收集对应的watcher
- v-if和v-for不能连用
- 如果需要使用v-for给每项元素绑定事件时使用事件代理
- SPA 页面采用keep-alive缓存组件
- 在更多的情况下，使用v-if替代v-show
- key保证唯一
- 使用路由懒加载、异步组件
- 防抖、节流
- 第三方模块按需导入
- 长列表滚动到可视区域动态加载
- 图片懒加载

（2）SEO优化

- 预渲染
- 服务端渲染SSR

（3）打包优化

- 压缩代码
- Tree Shaking/Scope Hoisting
- 使用cdn加载第三方模块
- 多线程打包happypack
- splitChunks抽离公共文件
- sourceMap优化

（4）用户体验

- 骨架屏
- PWA
- 还可以使用缓存(客户端缓存、服务端缓存)优化、服务端开启gzip压缩等。

Vue.extend 作用和原理

官方解释：Vue.extend 使用基础 Vue 构造器，创建一个“子类”。参数是一个包含组件选项的对象。

其实就是一个子类构造器 是 Vue 组件的核心 api 实现思路就是使用原型继承的方法返回了 Vue 的子类 并且利用 mergeOptions 把传入组件的 options 和父类的 options 进行了合并

相关代码如下

javascript 复制代码

```javascript
export default function initExtend(Vue) {
  let cid = 0; //组件的唯一标识
  // 创建子类继承Vue父类 便于属性扩展
  Vue.extend = function (extendOptions) {
    // 创建子类的构造函数 并且调用初始化方法
    const Sub = function VueComponent(options) {
      this._init(options); //调用Vue初始化方法
    };
    Sub.cid = cid++;
    Sub.prototype = Object.create(this.prototype); // 子类原型指向父类
    Sub.prototype.constructor = Sub; //constructor指向自己
    Sub.options = mergeOptions(this.options, extendOptions); //合并自己的options和父类的options
    return Sub;
  };
}
```

常见的事件修饰符及其作用

- `.stop`：等同于 JavaScript 中的 `event.stopPropagation()` ，防止事件冒泡；
- `.prevent` ：等同于 JavaScript 中的 `event.preventDefault()` ，防止执行预设的行为（如果事件可取消，则取消该事件，而不停止事件的进一步传播）；
- `.capture` ：与事件冒泡的方向相反，事件捕获由外到内；
- `.self` ：只会触发自己范围内的事件，不包含子元素；
- `.once` ：只会触发一次。

Vue.js的template编译

简而言之，就是先转化成AST树，再得到的render函数返回VNode（Vue的虚拟DOM节点），详细步骤如下：

> 首先，通过compile编译器把template编译成AST语法树（abstract syntax tree 即 源代码的抽象语法结构的树状表现形式），compile是createCompiler的返回值，createCompiler是用以创建编译器的。另外compile还负责合并option。

> 然后，AST会经过generate（将AST语法树转化成render funtion字符串的过程）得到render函数，render的返回值是VNode，VNode是Vue的虚拟DOM节点，里面有（标签名、子节点、文本等等）

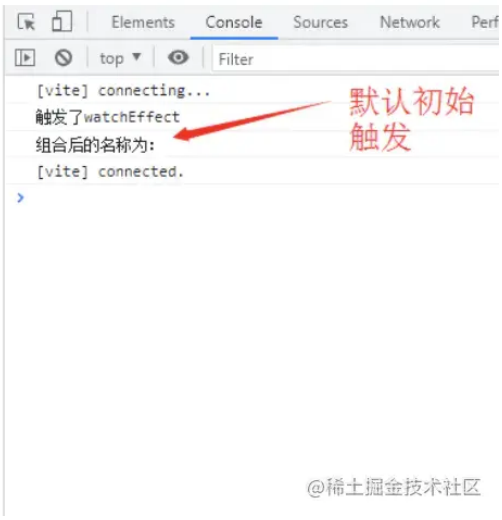
vue3中 watch、watchEffect区别

- `watch` 是惰性执行，也就是只有监听的值发生变化的时候才会执行，但是 `watchEffect` 不同，每次代码加载 `watchEffect` 都会执行（忽略 `watch` 第三个参数的配置，如果修改配置项也可以实现立即执行）
- `watch` 需要传递监听的对象， `watchEffect` 不需要
- `watch` 只能监听响应式数据： `ref` 定义的属性和 `reactive` 定义的对象，如果直接监听 `reactive` 定义对象中的属性是不允许的（会报警告），除非使用函数转换一下。其实就是官网上说的监听一个 `getter`
- `watchEffect` 如果监听 `reactive` 定义的对象是不起作用的，只能监听对象中的属性

看一下 `watchEffect` 的代码

```html 复制代码
<template>
<div>
  请输入firstName：
  <input type="text" v-model="firstName">
</div>
<div>
  请输入lastName：
  <input type="text" v-model="lastName">
</div>
<div>
  请输入obj.text：
  <input type="text" v-model="obj.text">
</div>
 <div>
 【obj.text】 {{obj.text}}
 </div>
```

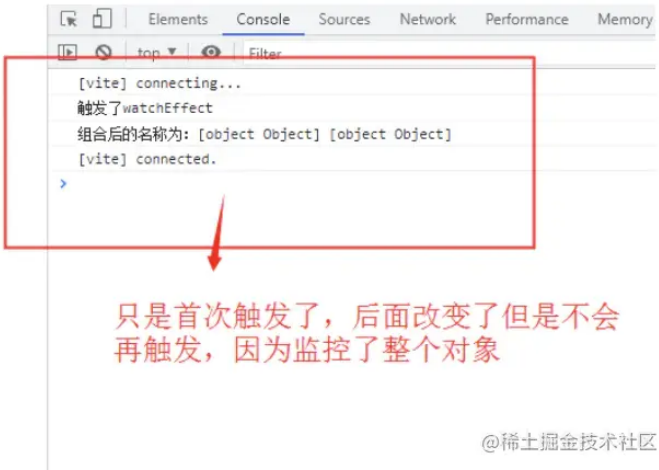
```
</template>

<script>
import {ref, reactive, watch, watchEffect} from 'vue'
export default {
  name: "HelloWorld",
  props: {
    msg: String,
  },
  setup(props,content){
    let firstName = ref('')
    let lastName = ref('')
    let obj= reactive({
      text:'hello'
    })
    watchEffect(()=>{
      console.log('触发了watchEffect');
      console.log(`组合后的名称为：${firstName.value}${lastName.value}`)
    })
    return{
      obj,
      firstName,
      lastName
    }
  }
};
</script>
```



改造一下代码

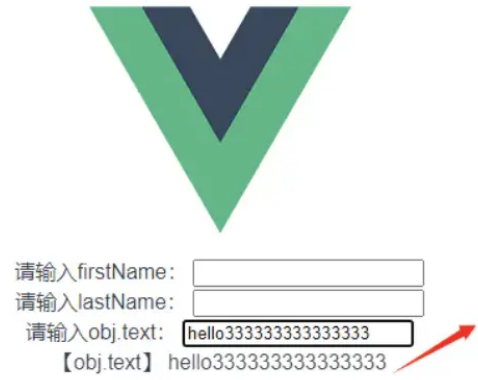
javascript 复制代码

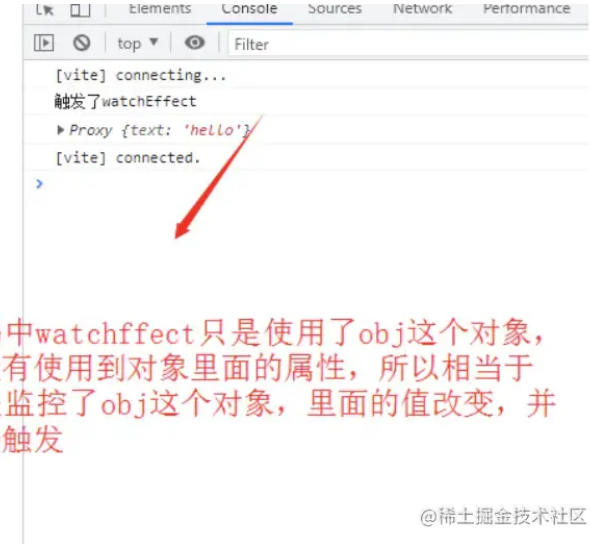
```
watchEffect(()=>{
  console.log('触发了watchEffect');
  // 这里我们不使用firstName.value/lastName.value ，相当于是监控整个ref,对应第四点上面的结论
  console.log(`组合后的名称为：${firstName}${lastName}`)
})
```



javascript 复制代码

```
watchEffect(()=>{
  console.log('触发了watchEffect');
  console.log(obj);
})
```



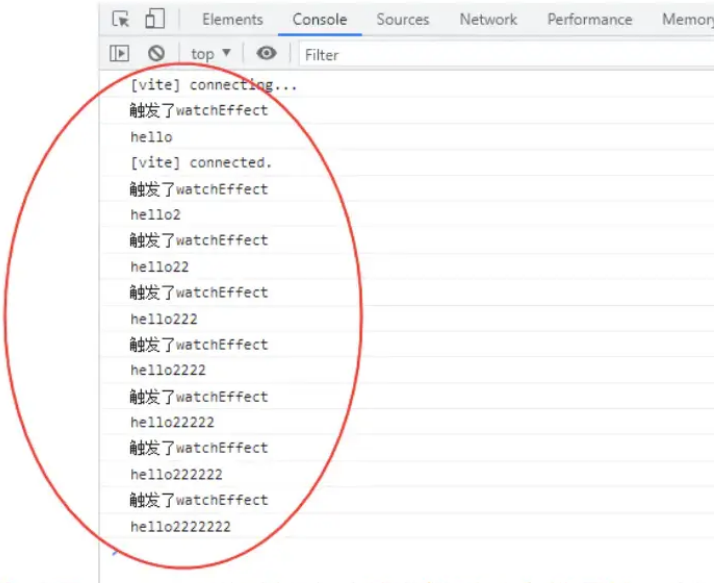


代码中watchffect只是使用了obj这个对象，并没有使用到对象里面的属性，所以相当于只是监控了obj这个对象，里面的值改变，并不会触发

稍微改造一下

javascript 复制代码

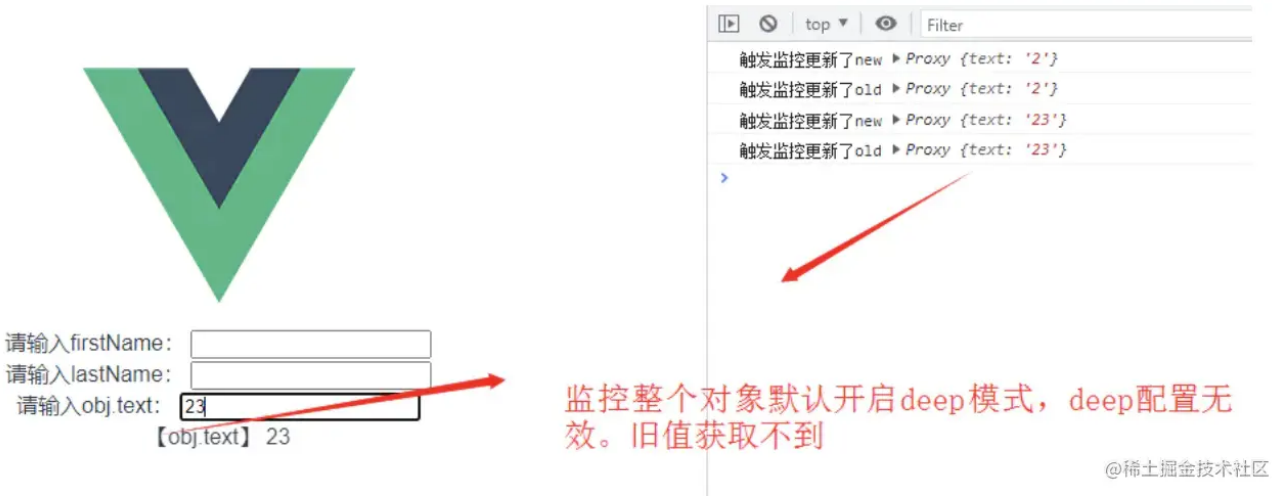
```
let obj = reactive({
  text:'hello'
})
watchEffect(()=>{
  console.log('触发了watchEffect');
  console.log(obj.text);
})
```



由于使用了obj.text所以相当于是监控了对象的属性，所以改变就触发了watchEffect

再看一下watch的代码，验证一下

```javascript
let obj= reactive({
  text:'hello'
})
// watch是惰性执行， 默认初始化之后不会执行，只有值有变化才会触发，可通过配置参数实现默认执行
watch(obj, (newValue, oldValue) => {
  // 回调函数
  console.log('触发监控更新了new',  newValue);
  console.log('触发监控更新了old',  oldValue);
},{
  // 配置immediate参数，立即执行，以及深层次监听
  immediate: true,
  deep: true
})
```



- 监控整个 `reactive` 对象，从上面的图可以看到 `deep` 实际默认是开启的，就算我们设置为 `false` 也还是无效。而且旧值获取不到。
- 要获取旧值则需要监控对象的属性，也就是监听一个 `getter` ，看下图

```
let firstName = ref("");
let lastName = ref("");
let obj = reactive({
  text: "hello",
});

watch(
  obj.text,
  (newValue, oldValue) => {
    // 回调函数
    console.log("触发监控更新了new", newValue);
    console.log("触发监控更新了old", oldValue);
  },
  {
    immediate:true,
    deep: false
  }
);

watch(
  () =>
  obj.text,
  (newValue, oldValue) => {
    // 回调函数
    console.log("触发监控更新了new", newValue);
    console.log("触发监控更新了old", oldValue);
  },
  {
    immediate: true,
    deep: true,
  }
);

return {
  obj,
  firstName,
  lastName,
};
```

错误写法，会报警告，这不是监控一个对象的get

正确写法

总结

- 如果定义了 `reactive` 的数据，想去使用 `watch` 监听数据改变，则无法正确获取旧值，并且 `deep` 属性配置无效，自动强制开启了深层次监听。
- 如果使用 `ref` 初始化一个对象或者数组类型的数据，会被自动转成 `reactive` 的实现方式，生成 `proxy` 代理对象。也会变得无法正确取旧值。

- 用任何方式生成的数据，如果接收的变量是一个 `proxy` 代理对象，就都会导致 `watch` 这个对象时，`watch` 回调里无法正确获取旧值。
- 所以当大家使用 `watch` 监听对象时，如果在不需要使用旧值的情况，可以正常监听对象没关系；但是如果当监听改变函数里面需要用到旧值时，只能监听 对象.xxx`属性 的方式才行

watch和watchEffect异同总结

体验

`watchEffect` 立即运行一个函数，然后被动地追踪它的依赖，当这些依赖改变时重新执行该函数

javascript 复制代码

```
const count = ref(0)

watchEffect(() => console.log(count.value))
// -> logs 0

count.value++
// -> logs 1
```

`watch` 侦测一个或多个响应式数据源并在数据源变化时调用一个回调函数

javascript 复制代码

```
const state = reactive({ count: 0 })
watch(
  () => state.count,
  (count, prevCount) => {
    /* ... */
  }
)
```

回答范例

1. `watchEffect` 立即运行一个函数，然后被动地追踪它的依赖，当这些依赖改变时重新执行该函数。 `watch` 侦测一个或多个响应式数据源并在数据源变化时调用一个回调函数
2. `watchEffect(effect)` 是一种特殊 `watch` ，传入的函数既是依赖收集的数据源，也是回调函数。如果我们不关心响应式数据变化前后的值，只是想拿这些数据做些事情，那么 `watchEffect` 就是我们需要的。 `watch` 更底层，可以接收多种数据源，包括用于依赖收集的 `getter` 函数，因此它完全可以实现 `watchEffect` 的功能，同时由于可以指定 `getter` 函数，依赖可以控制的更精确，还能获取数据变化前后的值，因此如果需要这些时我们会使用 `watch`

3. `watchEffect` 在使用时，传入的函数会立刻执行一次。 `watch` 默认情况下并不会执行回调函数，除非我们手动设置 `immediate` 选项
4. 从实现上来说， `watchEffect(fn)` 相当于 `watch(fn,fn,{immediate:true})`

`watchEffect` 定义如下

```javascript 复制代码
export function watchEffect(
  effect: WatchEffect,
  options?: WatchOptionsBase
): WatchStopHandle {
  return doWatch(effect, null, options)
}
```

`watch` 定义如下

```javascript 复制代码
export function watch<T = any, Immediate extends Readonly<boolean> = false>(
  source: T | WatchSource<T>,
  cb: any,
  options?: WatchOptions<Immediate>
): WatchStopHandle {
  return doWatch(source as any, cb, options)
}
```

很明显 `watchEffect` 就是一种特殊的 `watch` 实现。

Vue中的key到底有什么用?

`key` 是为Vue中的vnode标记的唯一id,通过这个key,我们的diff操作可以更准确、更快速

diff算法的过程中,先会进行新旧节点的首尾交叉对比,当无法匹配的时候会用新节点的 `key` 与旧节点进行比对,然后超出差异.

> diff程可以概括为：oldCh和newCh各有两个头尾的变量StartIdx和EndIdx，它们的2个变量相互比较，一共有4种比较方式。如果4种比较都没匹配，如果设置了key，就会用key进行比较，在比较的过程中，变量会往中间靠，一旦StartIdx>EndIdx表明oldCh和newCh至少有一个已经遍历完了，就会结束比较,这四种比较方式就是首、尾、旧尾新头、旧头新尾.

- 准确: 如果不加 `key` ,那么vue会选择复用节点(Vue的就地更新策略),导致之前节点的状态被保留下来,会产生一系列的bug.
- 快速: key的唯一性可以被Map数据结构充分利用,相比于遍历查找的时间复杂度O(n),Map的时间复杂度仅仅为O(1).

Vue data 中某一个属性的值发生改变后，视图会立即同步执行重新渲染吗?

不会立即同步执行重新渲染。Vue 实现响应式并不是数据发生变化之后 DOM 立即变化，而是按一定的策略进行 DOM 的更新。Vue 在更新 DOM 时是异步执行的。只要侦听到数据变化，Vue 将开启一个队列，并缓冲在同一事件循环中发生的所有数据变更。

如果同一个watcher被多次触发，只会被推入到队列中一次。这种在缓冲时去除重复数据对于避免不必要的计算和 DOM 操作是非常重要的。然后，在下一个的事件循环tick中，Vue 刷新队列并执行实际（已去重的）工作。

vue和react的区别

=> 相同点:

```markdown
1. 数据驱动页面，提供响应式的试图组件
2. 都有virtual DOM,组件化的开发，通过props参数进行父子之间组件传递数据，都实现了webComponents规范
3. 数据流动单向，都支持服务器的渲染SSR
4. 都有支持native的方法，react有React native， vue有wexx
```

=> 不同点:

```css
1.数据绑定：Vue实现了双向的数据绑定，react数据流动是单向的
2.数据渲染：大规模的数据渲染，react更快
3.使用场景：React配合Redux架构适合大规模多人协作复杂项目，Vue适合小快的项目
4.开发风格：react推荐做法jsx + inline style把html和css都写在js了
    vue是采用webpack + vue-loader单文件组件格式，html, js, css同一个文件
```

Vue 的生命周期方法有哪些 一般在哪一步发请求

beforeCreate 在实例初始化之后，数据观测(data observer) 和 event/watcher 事件配置之前被调用。在当前阶段 data、methods、computed 以及 watch 上的数据和方法都不能被访问

created 实例已经创建完成之后被调用。在这一步，实例已完成以下的配置：数据观测(data observer)，属性和方法的运算， watch/event 事件回调。这里没有 *el*, 如果非要想与*Dom*进行交互，可以通过*vm*.nextTick 来访问 Dom

beforeMount 在挂载开始之前被调用：相关的 render 函数首次被调用。

mounted 在挂载完成后发生，在当前阶段，真实的 Dom 挂载完毕，数据完成双向绑定，可以访问到 Dom 节点

beforeUpdate 数据更新时调用，发生在虚拟 DOM 重新渲染和打补丁（patch）之前。可以在这个钩子中进一步地更改状态，这不会触发附加的重渲染过程

updated 发生在更新完成之后，当前阶段组件 Dom 已完成更新。要注意的是避免在此期间更改数据，因为这可能会导致无限循环的更新，该钩子在服务器端渲染期间不被调用。

beforeDestroy 实例销毁之前调用。在这一步，实例仍然完全可用。我们可以在这时进行善后收尾工作，比如清除计时器。

destroyed Vue 实例销毁后调用。调用后，Vue 实例指示的所有东西都会解绑定，所有的事件监听器会被移除，所有的子实例也会被销毁。 该钩子在服务器端渲染期间不被调用。

activated keep-alive 专属，组件被激活时调用

deactivated keep-alive 专属，组件被销毁时调用

> 异步请求在哪一步发起?

可以在钩子函数 created、beforeMount、mounted 中进行异步请求，因为在这三个钩子函数中，data 已经创建，可以将服务端端返回的数据进行赋值。

如果异步请求不需要依赖 Dom 推荐在 created 钩子函数中调用异步请求，因为在 created 钩子函数中调用异步请求有以下优点：

- 能更快获取到服务端数据，减少页面 loading 时间；
- ssr 不支持 beforeMount 、mounted 钩子函数，所以放在 created 中有助于一致性；