必会vue面试题（附答案）

vue初始化页面闪动问题

使用vue开发时，在vue初始化之前，由于div是不归vue管的，所以我们写的代码在还没有解析的情况下会容易出现花屏现象，看到类似于{{message}}的字样，虽然一般情况下这个时间很短暂，但是还是有必要让解决这个问题的。

首先：在css里加上以下代码：

```javascript 复制代码
[v-cloak] {    display: none;}
```

如果没有彻底解决问题，则在根元素加上 `style="display: none;" :style="{display: 'block'}"`

v-show 与 v-if 有什么区别?

v-if 是**真正**的条件渲染，因为它会确保在切换过程中条件块内的事件监听器和子组件适当地被销毁和重建；也是**惰性的**：如果在初始渲染时条件为假，则什么也不做——直到条件第一次变为真时，才会开始渲染条件块。

v-show 就简单得多——不管初始条件是什么，元素总是会被渲染，并且只是简单地基于 CSS 的 “display” 属性进行切换。

所以，v-if 适用于在运行时很少改变条件，不需要频繁切换条件的场景；v-show 则适用于需要非常频繁切换条件的场景。

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

为什么Vue采用异步渲染呢?

`Vue` 是组件级更新，如果不采用异步更新，那么每次更新数据都会对当前组件进行重新渲染，所以为了性能， `Vue` 会在本轮数据更新后，在异步更新视图。核心思想 `nextTick` 。

`dep.notify()` 通知 watcher进行更新， `subs[i].update` 依次调用 watcher 的 `update` ， `queueWatcher` 将watcher 去重放入队列， nextTick（ `flushSchedulerQueue` ）在下一tick中刷新watcher队列（异步）。

keep-alive 使用场景和原理

keep-alive 是 Vue 内置的一个组件，可以实现组件缓存，当组件切换时不会对当前组件进行卸载。

- 常用的两个属性 include/exclude，允许组件有条件的进行缓存。
- 两个生命周期 activated/deactivated，用来得知当前组件是否处于活跃状态。
- keep-alive 的中还运用了 LRU(最近最少使用) 算法，选择最近最久未使用的组件予以淘汰。

能说下 vue-router 中常用的 hash 和 history 路由模式实现原理吗?

（1）hash 模式的实现原理

早期的前端路由的实现就是基于 location.hash 来实现的。其实现原理很简单，location.hash 的值就是 URL 中 # 后面的内容。比如下面这个网站，它的 location.hash 的值为 '#search'：

复制代码

```
https://www.word.com#search
```

hash 路由模式的实现主要是基于下面几个特性：

- URL 中 hash 值只是客户端的一种状态，也就是说当向服务器端发出请求时，hash 部分不会被发送；

hash 值的改变，都会在浏览器的访问历史中增加一个记录。因此我们能通过浏览器的回退、前进按钮控制hash 的切换；

- 可以通过 a 标签，并设置 href 属性，当用户点击这个标签后，URL 的 hash 值会发生改变；或者使用 JavaScript 来对 loaction.hash 进行赋值，改变 URL 的 hash 值；
- 我们可以使用 hashchange 事件来监听 hash 值的变化，从而对页面进行跳转（渲染）。

（2）history 模式的实现原理

HTML5 提供了 History API 来实现 URL 的变化。其中做最主要的 API 有以下两个：history.pushState() 和 history.repalceState()。这两个 API 可以在不进行刷新的情况下，操作浏览器的历史纪录。唯一不同的是，前者是新增一个历史记录，后者是直接替换当前的历史记录，如下所示：

javascript 复制代码

```
window.history.pushState(null, null, path);
window.history.replaceState(null, null, path);
```

history 路由模式的实现主要基于存在下面几个特性：

- pushState 和 repalceState 两个 API 来操作实现 URL 的变化；
- 我们可以使用 popstate 事件来监听 url 的变化，从而对页面进行跳转（渲染）；
- history.pushState() 或 history.replaceState() 不会触发 popstate 事件，这时我们需要手动触发页面跳转（渲染）。

参考 前端进阶面试题详细解答

Vue模版编译原理知道吗，能简单说一下吗?

简单说，Vue的编译过程就是将 `template` 转化为 `render` 函数的过程。会经历以下阶段：

- 生成AST树
- 优化
- codegen

首先解析模版，生成 AST语法树 (一种用JavaScript对象的形式来描述整个模板)。 使用大量的正则表达式对模板进行解析，遇到标签、文本的时候都会执行对应的钩子进行相关处理。

Vue的数据是响应式的，但其实模板中并不是所有的数据都是响应式的。有一些数据首次渲染后就不会再变化，对应的DOM也不会变化。那么优化过程就是深度遍历AST树，按照相关条件对树节点进行标记。这些被标记的节点(静态节点)我们就可以 跳过对它们的比对 ，对运行时的模板起到很大的优化作用。

编译的最后一步是 将优化后的AST树转换为可执行的代码 。

vue和react的区别

=> 相同点：

```markdown 复制代码
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：

```
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模版编译原理知道吗，能简单说一下吗?

简单说，Vue的编译过程就是将 `template` 转化为 `render` 函数的过程。会经历以下阶段：

- 生成AST树
- 优化
- codegen

首先解析模版，生成 `AST语法树` (一种用JavaScript对象的形式来描述整个模板)。 使用大量的正则表达式对模板进行解析，遇到标签、文本的时候都会执行对应的钩子进行相关处理。

Vue的数据是响应式的，但其实模板中并不是所有的数据都是响应式的。有一些数据首次渲染后就不会再变化，对应的DOM也不会变化。那么优化过程就是深度遍历AST树，按照相关条件对树节点进行标记。这些被标记的节点(静态节点)我们就可以 `跳过对它们的比对` ，对运行时的模板起到很大的优化作用。

编译的最后一步是 `将优化后的AST树转换为可执行的代码` 。

v-for 为什么要加 key

如果不使用 key，Vue 会使用一种最大限度减少动态元素并且尽可能的尝试就地修改/复用相同类型元素的算法。key 是为 Vue 中 vnode 的唯一标记，通过这个 key，我们的 diff 操作可以更准确、更快速

更准确：因为带 key 就不是就地复用了，在 sameNode 函数 a.key === b.key 对比中可以避免就地复用的情况。所以会更加准确。

更快速：利用 key 的唯一性生成 map 对象来获取对应节点，比遍历方式更快

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。

说一下Vue的生命周期

Vue 实例有一个完整的生命周期，也就是从开始创建、初始化数据、编译模版、挂载Dom -> 渲染、更新 -> 渲染、卸载 等一系列过程，称这是Vue的生命周期。

1. **beforeCreate（创建前）**：数据观测和初始化事件还未开始，此时 data 的响应式追踪、event/watcher 都还没有被设置，也就是说不能访问到data、computed、watch、methods上的方法和数据。
2. **created（创建后）**：实例创建完成，实例上配置的 options 包括 data、computed、watch、methods 等都配置完成，但是此时渲染得节点还未挂载到 DOM，所以不能访问到 `$el` 属性。
3. **beforeMount（挂载前）**：在挂载开始之前被调用，相关的render函数首次被调用。实例已完成以下的配置：编译模板，把data里面的数据和模板生成html。此时还没有挂载html到页面上。
4. **mounted（挂载后）**：在el被新创建的 vm.$el 替换，并挂载到实例上去之后调用。实例已完成以下的配置：用上面编译好的html内容替换el属性指向的DOM对象。完成模板中的html渲染到html 页面中。此过程中进行ajax交互。
5. **beforeUpdate（更新前）**：响应式数据更新时调用，此时虽然响应式数据更新了，但是对应的真实 DOM 还没有被渲染。
6. **updated（更新后）**：在由于数据更改导致的虚拟DOM重新渲染和打补丁之后调用。此时 DOM 已经根据响应式数据的变化更新了。调用时，组件 DOM已经更新，所以可以执行依赖于DOM的操作。然而在大多数情况下，应该避免在此期间更改状态，因为这可能会导致更新无限循环。该钩子在服务器端渲染期间不被调用。
7. **beforeDestroy（销毁前）**：实例销毁之前调用。这一步，实例仍然完全可用，`this` 仍能获取到实例。
8. **destroyed（销毁后）**：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。该钩子在服务端渲染期间不被调用。

另外还有 `keep-alive` 独有的生命周期，分别为 `activated` 和 `deactivated` 。用 `keep-alive` 包裹的组件在切换时不会进行销毁，而是缓存到内存中并执行 `deactivated` 钩子函数，命中缓存渲染后会执行 `activated` 钩子函数。

虚拟DOM实现原理?

- 虚拟DOM本质上是JavaScript对象,是对真实DOM的抽象
- 状态变更时，记录新树和旧树的差异
- 最后把差异更新到真正的dom中

v-show 与 v-if 有什么区别?

v-if 是**真正**的条件渲染，因为它会确保在切换过程中条件块内的事件监听器和子组件适当地被销毁和重建；也是**惰性的**：如果在初始渲染时条件为假，则什么也不做——直到条件第一次变为真时，才会开始渲染条件块。

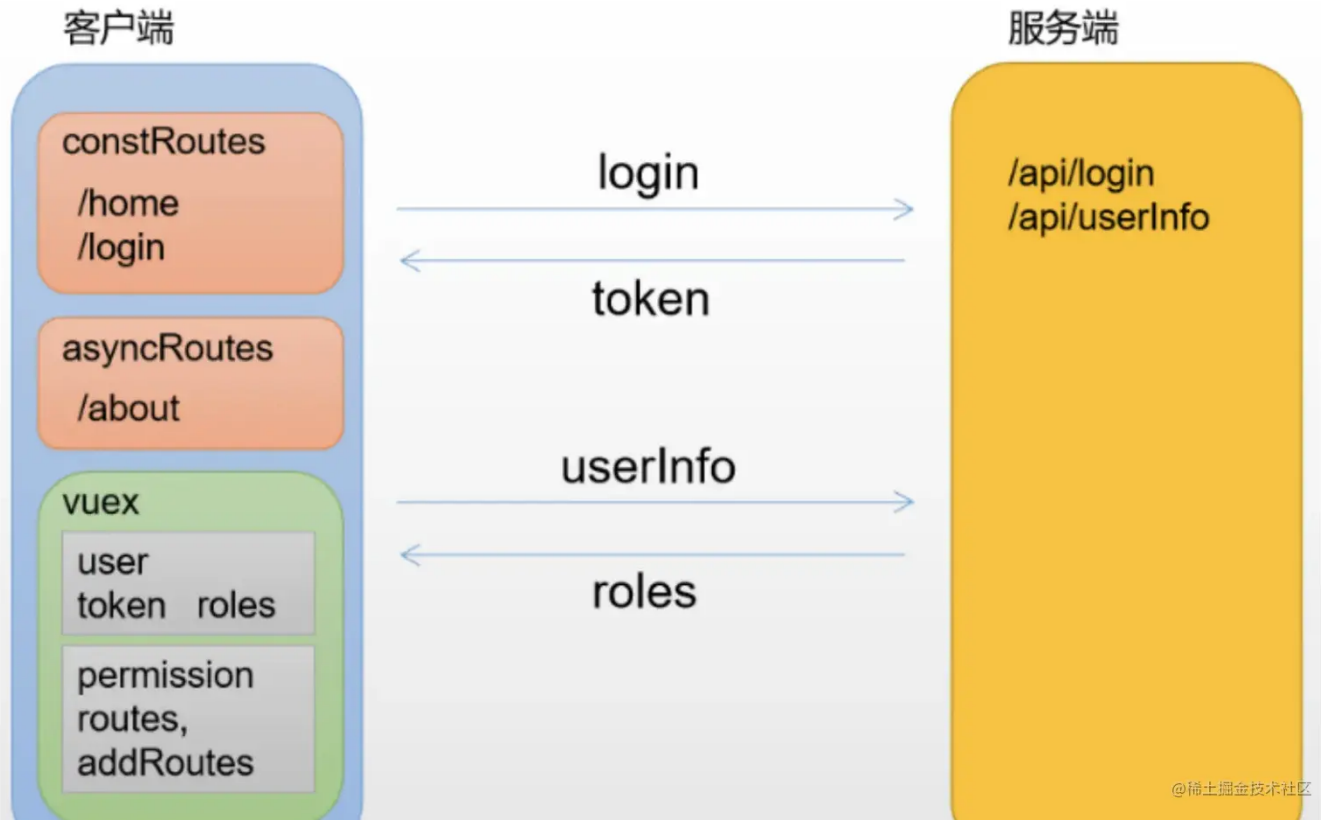
v-show 就简单得多——不管初始条件是什么，元素总是会被渲染，并且只是简单地基于 CSS 的 “display” 属性进行切换。

所以，v-if 适用于在运行时很少改变条件，不需要频繁切换条件的场景；v-show 则适用于需要非常频繁切换条件的场景。

Vue要做权限管理该怎么做？控制到按钮级别的权限怎么做?

分析

- 综合实践题目，实际开发中经常需要面临权限管理的需求，考查实际应用能力。
- 权限管理一般需求是两个：页面权限和按钮权限，从这两个方面论述即可。



思路

- 权限管理需求分析：页面和按钮权限
- 权限管理的实现方案：分后端方案和前端方案阐述
- 说说各自的优缺点

回答范例

1. 权限管理一般需求是页面权限和按钮权限的管理

2. 具体实现的时候分后端和前端两种方案：

- **前端方案** 会把所有路由信息在前端配置，通过路由守卫要求用户登录，用户登录后根据角色过滤出路由表。比如我会配置一个 `asyncRoutes` 数组，需要认证的页面在其路由的 `meta` 中添加一个 `roles` 字段，等获取用户角色之后取两者的交集，若结果不为空则说明可以访问。此过滤过程结束，剩下的路由就是该用户能访问的页面，最后通过 `router.addRoutes(accessRoutes)` 方式动态添加路由即可

- **后端方案** 会把所有页面路由信息存在数据库中，用户登录的时候根据其角色查询得到其能访问的所有页面路由信息返回给前端，前端再通过 `addRoutes` 动态添加路由信息

- 按钮权限的控制通常会 实现一个指令 ，例如 v-permission ，将按钮要求角色通过值传给v-permission 指令，在指令的 moutned 钩子中可以判断当前用户角色和按钮是否存在交集，有则保留按钮，无则移除按钮

3. 纯前端方案的优点是实现简单，不需要额外权限管理页面，但是维护起来问题比较大，有新的页面和角色需求就要修改前端代码重新打包部署；服务端方案就不存在这个问题，通过专门的角色和权限管理页面，配置页面和按钮权限信息到数据库，应用每次登陆时获取的都是最新的路由信息，可谓一劳永逸！

可能的追问

1. 类似 Tabs 这类组件能不能使用 v-permission 指令实现按钮权限控制?

html 复制代码

```html
<el-tabs>
  <el-tab-pane label="用户管理" name="first">用户管理</el-tab-pane>
    <el-tab-pane label="角色管理" name="third">角色管理</el-tab-pane>
</el-tabs>
```

2. 服务端返回的路由信息如何添加到路由器中?

javascript 复制代码

```javascript
// 前端组件名和组件映射表
const map = {
  //xx: require('@/views/xx.vue').default // 同步的方式
  xx: () => import('@/views/xx.vue') // 异步的方式
}
// 服务端返回的asyncRoutes
const asyncRoutes = [
  { path: '/xx', component: 'xx',... }
]
// 遍历asyncRoutes，将component替换为map[component]
function mapComponent(asyncRoutes) {
  asyncRoutes.forEach(route => {
    route.component = map[route.component];
    if(route.children) {
      route.children.map(child => mapComponent(child))
    }
    })
}
mapComponent(asyncRoutes)
```

对前端路由的理解

在前端技术早期，一个 url 对应一个页面，如果要从 A 页面切换到 B 页面，那么必然伴随着页面的刷新。这个体验并不好，不过在最初也是无奈之举——用户只有在刷新页面的情况下，才可以重新去请求数据。

后来，改变发生了——Ajax 出现了，它允许人们在不刷新页面的情况下发起请求；与之共生的，还有“不刷新页面即可更新页面内容”这种需求。在这样的背景下，出现了 **SPA（单页面应用）**。

SPA极大地提升了用户体验，它允许页面在不刷新的情况下更新页面内容，使内容的切换更加流畅。但是在 SPA 诞生之初，人们并没有考虑到“定位”这个问题——在内容切换前后，页面的 URL 都是一样的，这就带来了两个问题：

- SPA 其实并不知道当前的页面“进展到了哪一步”。可能在一个站点下经过了反复的“前进”才终于唤出了某一块内容，但是此时只要刷新一下页面，一切就会被清零，必须重复之前的操作、才可以重新对内容进行定位——SPA 并不会“记住”你的操作。
- 由于有且仅有一个 URL 给页面做映射，这对 SEO 也不够友好，搜索引擎无法收集全面的信息

为了解决这个问题，前端路由出现了。

前端路由可以帮助我们在仅有一个页面的情况下，“记住”用户当前走到了哪一步——为 SPA 中的各个视图匹配一个唯一标识。这意味着用户前进、后退触发的新内容，都会映射到不同的 URL 上去。此时即便他刷新页面，因为当前的 URL 可以标识出他所处的位置，因此内容也不会丢失。

那么如何实现这个目的呢？首先要解决两个问题：

- 当用户刷新页面时，浏览器会默认根据当前 URL 对资源进行重新定位（发送请求）。这个动作对 SPA 是不必要的，因为我们的 SPA 作为单页面，无论如何也只会有一个资源与之对应。此时若走正常的请求-刷新流程，反而会使用户的前进后退操作无法被记录。
- 单页面应用对服务端来说，就是一个URL、一套资源，那么如何做到用“不同的URL”来映射不同的视图内容呢？

从这两个问题来看，服务端已经完全救不了这个场景了。所以要靠咱们前端自力更生，不然怎么叫“前端路由”呢？作为前端，可以提供这样的解决思路：

- 拦截用户的刷新操作，避免服务端盲目响应、返回不符合预期的资源内容。把刷新这个动作完全放到前端逻辑里消化掉。

- 感知 URL 的变化。这里不是说要改造 URL、凭空制造出 N 个 URL 来。而是说 URL 还是那个 URL，只不过我们可以给它做一些微小的处理——这些处理并不会影响 URL 本身的性质，不会影响服务器对它的识别，只有我们前端感知的到。一旦我们感知到了，我们就根据这些变化、用 JS 去给它生成不同的内容。

子组件可以直接改变父组件的数据吗?

子组件不可以直接改变父组件的数据。这样做主要是为了维护父子组件的单向数据流。每次父级组件发生更新时，子组件中所有的 prop 都将会刷新为最新的值。如果这样做了，Vue 会在浏览器的控制台中发出警告。

Vue提倡单向数据流，即父级 props 的更新会流向子组件，但是反过来则不行。这是为了防止意外的改变父组件状态，使得应用的数据流变得难以理解，导致数据流混乱。如果破坏了单向数据流，当应用复杂时，debug 的成本会非常高。

只能通过 `$emit` 派发一个自定义事件，父组件接收到后，由父组件修改。

生命周期钩子是如何实现的

Vue 的生命周期钩子核心实现是利用发布订阅模式先把用户传入的的生命周期钩子订阅好（内部采用数组的方式存储）然后在创建组件实例的过程中会一次执行对应的钩子方法（发布）

相关代码如下

javascript 复制代码

```javascript
export function callHook(vm, hook) {
  // 依次执行生命周期对应的方法
  const handlers = vm.$options[hook];
  if (handlers) {
    for (let i = 0; i < handlers.length; i++) {
      handlers[i].call(vm); //生命周期里面的this指向当前实例
    }
  }
}

// 调用的时候
Vue.prototype._init = function (options) {
  const vm = this;
  vm.$options = mergeOptions(vm.constructor.options, options);
  callHook(vm, "beforeCreate"); //初始化数据之前
  // 初始化状态
  initState(vm);
```

```
  callHook(vm, "created"); //初始化数据之后
  if (vm.$options.el) {
    vm.$mount(vm.$options.el);
  }
};
```

虚拟DOM的解析过程

虚拟DOM的解析过程：

- 首先对将要插入到文档中的 DOM 树结构进行分析，使用 js 对象将其表示出来，比如一个元素对象，包含 TagName、props 和 Children 这些属性。然后将这个 js 对象树给保存下来，最后再将 DOM 片段插入到文档中。
- 当页面的状态发生改变，需要对页面的 DOM 的结构进行调整的时候，首先根据变更的状态，重新构建起一棵对象树，然后将这棵新的对象树和旧的对象树进行比较，记录下两棵树的的差异。
- 最后将记录的有差异的地方应用到真正的 DOM 树中去，这样视图就更新了。

Vue是如何收集依赖的?

在初始化 Vue 的每个组件时，会对组件的 data 进行初始化，就会将由普通对象变成响应式对象，在这个过程中便会进行依赖收集的相关逻辑，如下所示：

javascript 复制代码

```
function defieneReactive (obj, key, val){
  const dep = new Dep();
  ...
  Object.defineProperty(obj, key, {
    ...
    get: function reactiveGetter () {
      if(Dep.target){
        dep.depend();
        ...
      }
      return val
    }
    ...
  })
}
```

以上只保留了关键代码，主要就是 `const dep = new Dep()` 实例化一个 Dep 的实例，然后在 get 函数中通过 `dep.depend()` 进行依赖收集。 **（1）Dep** Dep是整个依赖收集的核心，其关键代码如下：

javascript 复制代码

```
class Dep {
  static target;
  subs;

  constructor () {
    ...
    this.subs = [];
  }
  addSub (sub) {
    this.subs.push(sub)
  }
  removeSub (sub) {
    remove(this.sub, sub)
  }
  depend () {
    if(Dep.target){
      Dep.target.addDep(this)
    }
  }
  notify () {
    const subs = this.subds.slice();
    for(let i = 0;i < subs.length; i++){
      subs[i].update()
    }
  }
}
```

Dep 是一个 class ，其中有一个关 键的静态属性 static，它指向了一个全局唯一 Watcher，保证了同一时间全局只有一个 watcher 被计算，另一个属性 subs 则是一个 Watcher 的数组，所以 Dep 实际上就是对 Watcher 的管理，再看看 Watcher 的相关代码 ：

（2）Watcher

javascript 复制代码

```
class Watcher {
  getter;
  ...
  constructor (vm, expression){
    ...
    this.getter = expression;
    this.get();
```

```
  }
  get () {
    pushTarget(this);
    value = this.getter.call(vm, vm)
    ...
    return value
  }
  addDep (dep){
        ...
    dep.addSub(this)
  }
  ...
}
function pushTarget (_target) {
  Dep.target = _target
}
```

Watcher 是一个 class，它定义了一些方法，其中和依赖收集相关的主要有 get、addDep 等。

（3）过程

在实例化 Vue 时，依赖收集的相关过程如下：初始化状态 initState，这中间便会通过 defineReactive 将数据变成响应式对象，其中的 getter 部分便是用来依赖收集的。初始化最终会走 mount 过程，其中会实例化 Watcher，进入 Watcher 中，便会执行 this.get() 方法，

javascript 复制代码

```
updateComponent = () => {
  vm._update(vm._render())
}
new Watcher(vm, updateComponent)
```

get 方法中的 pushTarget 实际上就是把 Dep.target 赋值为当前的 watcher。

this.getter.call（vm，vm），这里的 getter 会执行 vm._render() 方法，在这个过程中便会触发数据对象的 getter。那么每个对象值的 getter 都持有一个 dep，在触发 getter 的时候会调用 dep.depend() 方法，也就会执行 Dep.target.addDep(this)。刚才 Dep.target 已经被赋值为 watcher，于是便会执行 addDep 方法，然后走到 dep.addSub() 方法，便将当前的 watcher 订阅到这个数据持有的 dep 的 subs 中，这个目的是为后续数据变化时候能通知到哪些 subs 做准备。所以在 vm._render() 过程中，会触发所有数据的 getter，这样便已经完成了一个依赖收集的过程。