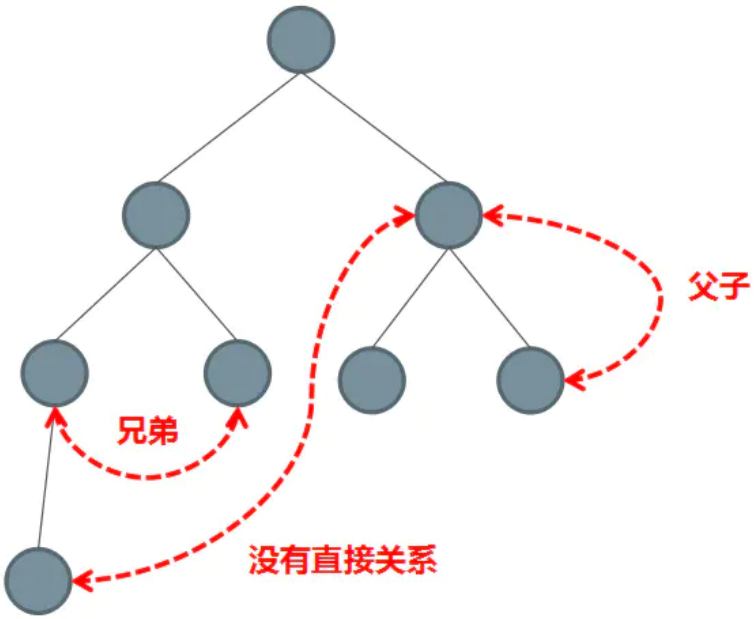
组件间通信的分类可以分成以下

- 父子组件之间的通信
- 兄弟组件之间的通信
- 祖孙与后代组件之间的通信
- 非关系组件间之间的通信

关系图:



3.4. 组件间通信的方案

整理 `vue` 中8种常规的通信方案

1. 通过 props 传递
2. 通过 $emit 触发自定义事件
3. 使用 ref
4. EventBus
5. $parent$或 root
6. attrs 与 listeners
7. Provide 与 Inject
8. Vuex

3.4.1. props传递数据

- 适用场景：父组件传递数据给子组件
- 子组件设置 `props` 属性，定义接收父组件传递过来的参数
- 父组件在使用子组件标签中通过字面量来传递值

`Children.vue`

JavaScript | 复制代码

```
props:{
    // 字符串形式
  name:String // 接收的类型参数
    // 对象形式
    age:{
        type:Number, // 接收的类型为数值
        defaule:18,  // 默认值为18
       require:true // age属性必须传递
    }
}
```

`Father.vue` 组件

JavaScript | 复制代码

```
<Children name="jack" age=18 />
```

3.4.2. $emit 触发自定义事件

- 适用场景：子组件传递数据给父组件
- 子组件通过 `$emit触发` 自定义事件， `$emit` 第二个参数为传递的数值
- 父组件绑定监听器获取到子组件传递过来的参数

`Chilfen.vue`

```javascript
this.$emit('add', good)
```

`Father.vue`

```javascript
<Children @add="cartAdd($event)" />
```

3.4.3. ref

- 父组件在使用子组件的时候设置 `ref`
- 父组件通过设置子组件 `ref` 来获取数据

父组件

```javascript
<Children ref="foo" />

this.$refs.foo  // 获取子组件实例，通过子组件实例我们就能拿到对应的数据
```

3.4.4. EventBus

- 使用场景：兄弟组件传值
- 创建一个中央事件总线 `EventBus`
- 兄弟组件通过 `$emit` 触发自定义事件， `$emit` 第二个参数为传递的数值
- 另一个兄弟组件通过 `$on` 监听自定义事件

`Bus.js`

```JavaScript
// 创建一个中央时间总线类
class Bus {
  constructor() {
    this.callbacks = {};   // 存放事件的名字
  }
  $on(name, fn) {
    this.callbacks[name] = this.callbacks[name] || [];
    this.callbacks[name].push(fn);
  }
  $emit(name, args) {
    if (this.callbacks[name]) {
      this.callbacks[name].forEach((cb) => cb(args));
    }
  }
}

// main.js
Vue.prototype.$bus = new Bus() // 将$bus挂载到vue实例的原型上
// 另一种方式
Vue.prototype.$bus = new Vue() // Vue已经实现了Bus的功能
```

`Children1.vue`

```JavaScript
this.$bus.$emit('foo')
```

`Children2.vue`

```JavaScript
this.$bus.$on('foo', this.handle)
```

3.4.5. parent、root

- 通过共同祖辈 `$parent` 或者 `$root` 搭建通信桥连

兄弟组件

`this.$parent.on('add',this.add)`

另一个兄弟组件

`this.$parent.emit('add')`

3.4.6. attrs与listeners

- 适用场景：祖先传递数据给子孙
- 设置批量向下传属性 `$attrs` 和 `$listeners`
- 包含了父级作用域中不作为 `prop` 被识别 (且获取) 的特性绑定 (class 和 style 除外)。
- 可以通过 `v-bind="$attrs"` 传入内部组件

JavaScript 复制代码

```
// child：并未在props中声明foo
<p>{{$attrs.foo}}</p>

// parent
<HelloWorld foo="foo"/>
```

JavaScript 复制代码

```
// 给Grandson隔代传值，communication/index.vue
<Child2 msg="lalala" @some-event="onSomeEvent"></Child2>

// Child2做展开
<Grandson v-bind="$attrs" v-on="$listeners"></Grandson>

// Grandson使用
<div @click="$emit('some-event', 'msg from grandson')">
{{msg}}
</div>
```

3.4.7. provide 与 inject

- 在祖先组件定义 `provide` 属性，返回传递的值
- 在后代组件通过 `inject` 接收组件传递过来的值

祖先组件

JavaScript 复制代码

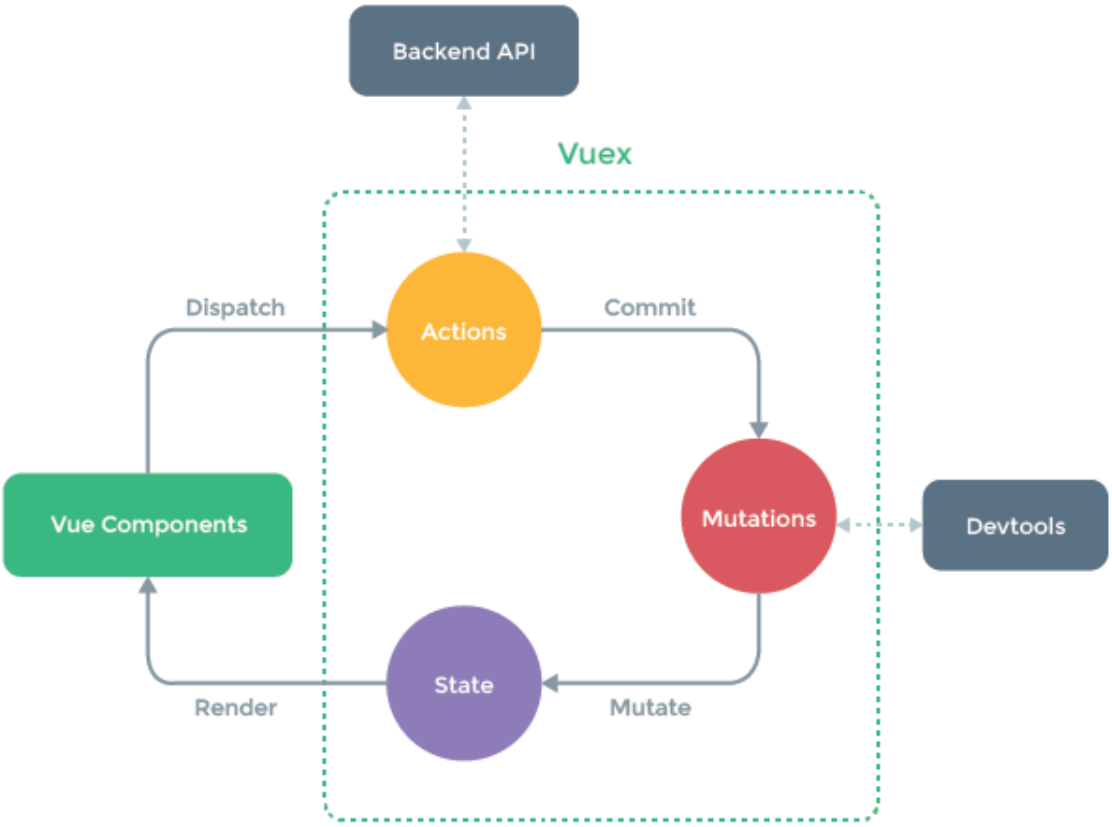
```
provide(){
    return {
        foo:'foo'
    }
}
```

后代组件

```
JavaScript | 复制代码
1  inject:['foo'] // 获取到祖先组件传递过来的值
```

3.4.8. `vuex`

- 适用场景: 复杂关系的组件数据传递
- `Vuex` 作用相当于一个用来存储共享变量的容器



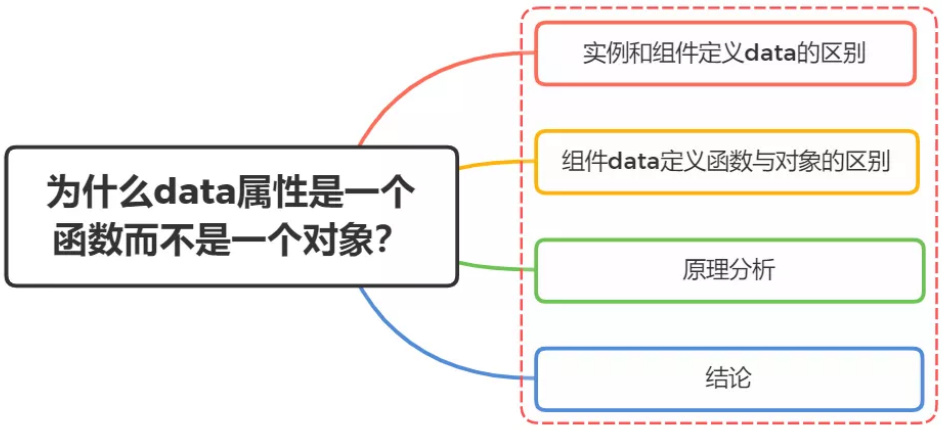
- `state` 用来存放共享变量的地方
- `getter`，可以增加一个 `getter` 派生状态，(相当于 `store` 中的计算属性)，用来获得共享变量的值
- `mutations` 用来存放修改 `state` 的方法。
- `actions` 也是用来存放修改state的方法，不过 `action` 是在 `mutations` 的基础上进行。常

用来做一些异步操作

3.5. 小结

- 父子关系的组件数据传递选择 `props` 与 `$emit` 进行传递，也可选择 `ref`
- 兄弟关系的组件数据传递可选择 `$bus`，其次可以选择 `$parent` 进行传递
- 祖先与后代组件数据传递可选择 `attrs` 与 `listeners` 或者 `Provide` 与 `Inject`
- 复杂关系的组件数据传递可以通过 `vuex` 存放共享的变量

4. 为什么data属性是一个函数而不是一个对象?



4.1. 实例和组件定义data的区别

`vue` 实例的时候定义 `data` 属性既可以是一个对象，也可以是一个函数

```javascript
const app = new Vue({
    el:"#app",
    // 对象格式
    data:{
        foo:"foo"
    },
    // 函数格式
    data(){
        return {
            foo:"foo"
        }
    }
 })
```

组件中定义 `data` 属性，只能是一个函数

如果为组件 `data` 直接定义为一个对象

```javascript
Vue.component('component1',{
    template:`<div>组件</div>`,
    data:{
        foo:"foo"
    }
})
```

则会得到警告信息

```
[Vue warn]: The "data" option should be a function that returns a per-instance value in component definitions.   vue.js:634
```

警告说明：返回的 `data` 应该是一个函数在每一个组件实例中

4.2. 组件data定义函数与对象的区别

上面讲到组件 `data` 必须是一个函数，不知道大家有没有思考过这是为什么呢?

在我们定义好一个组件的时候， `vue` 最终都会通过 `Vue.extend()` 构成组件实例

这里我们模仿组件构造函数，定义 `data` 属性，采用对象的形式

```javascript
function Component(){

}
Component.prototype.data = {
  count : 0
}
```

创建两个组件实例

```plaintext
const componentA = new Component()
const componentB = new Component()
```

修改 `componentA` 组件 `data` 属性的值， `componentB` 中的值也发生了改变

```javascript
console.log(componentB.data.count)  // 0
componentA.data.count = 1
console.log(componentB.data.count)  // 1
```

产生这样的原因这是两者共用了同一个内存地址， `componentA` 修改的内容，同样对 `componentB` 产生了影响

如果我们采用函数的形式，则不会出现这种情况（函数返回的对象内存地址并不相同）

```javascript
function Component(){
  this.data = this.data()
}
Component.prototype.data = function (){
    return {
      count : 0
    }
}
```

修改 `componentA` 组件 `data` 属性的值， `componentB` 中的值不受影响

```JavaScript
console.log(componentB.data.count)  // 0
componentA.data.count = 1
console.log(componentB.data.count)  // 0
```

`vue` 组件可能会有很多个实例，采用函数返回一个全新 `data` 形式，使每个实例对象的数据不会受到其他实例对象数据的污染

4.3. 原理分析

首先可以看看 `vue` 初始化 `data` 的代码， `data` 的定义可以是函数也可以是对象

源码位置： `/vue-dev/src/core/instance/state.js`

```JavaScript
function initData (vm: Component) {
  let data = vm.$options.data
  data = vm._data = typeof data === 'function'
    ? getData(data, vm)
    : data || {}
    ...
}
```

`data` 既能是 `object` 也能是 `function` ，那为什么还会出现上文警告呢?

别急，继续看下文

组件在创建的时候，会进行选项的合并

源码位置： `/vue-dev/src/core/util/options.js`

自定义组件会进入 `mergeOptions` 进行选项合并

JavaScript | 复制代码

```javascript
Vue.prototype._init = function (options?: Object) {
    ...
    // merge options
    if (options && options._isComponent) {
      // optimize internal component instantiation
      // since dynamic options merging is pretty slow, and none of the
      // internal component options needs special treatment.
      initInternalComponent(vm, options)
    } else {
      vm.$options = mergeOptions(
        resolveConstructorOptions(vm.constructor),
        options || {},
        vm
      )
    }
    ...
  }
```

定义 `data` 会进行数据校验

源码位置：`/vue-dev/src/core/instance/init.js`

这时候 `vm` 实例为 `undefined`，进入 `if` 判断，若 `data` 类型不是 `function`，则出现警告提示

JavaScript | 复制代码

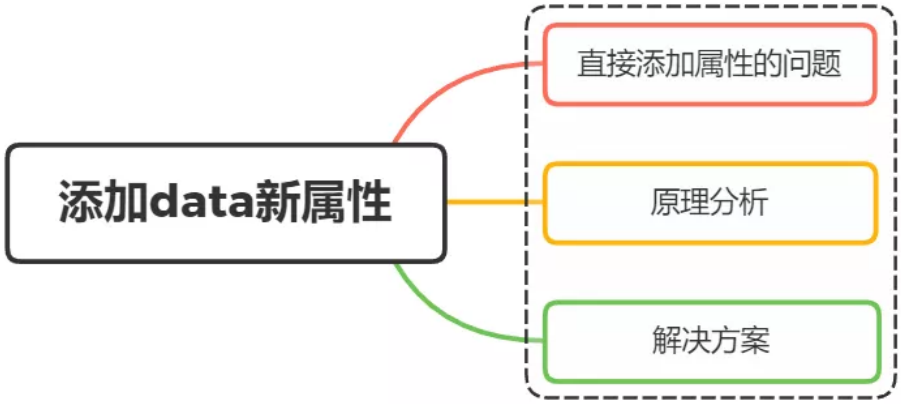
```
strats.data = function (
  parentVal: any,
  childVal: any,
  vm?: Component
): ?Function {
  if (!vm) {
    if (childVal && typeof childVal !== "function") {
      process.env.NODE_ENV !== "production" &&
        warn(
          'The "data" option should be a function ' +
            "that returns a per-instance value in component " +
            "definitions.",
          vm
        );

      return parentVal;
    }
    return mergeDataOrFn(parentVal, childVal);
  }
  return mergeDataOrFn(parentVal, childVal, vm);
};
```

4.4. 结论

- 根实例对象 `data` 可以是对象也可以是函数（根实例是单例），不会产生数据污染情况
- 组件实例对象 `data` 必须为函数，目的是为了防止多个组件实例对象之间共用一个 `data`，产生数据污染。采用函数的形式， `initData` 时会将其作为工厂函数都会返回全新 `data` 对象

5. 动态给vue的data添加一个新的属性时会发生什么？怎样解决？



5.1. 直接添加属性的问题

我们从一个例子开始

定义一个 `p` 标签，通过 `v-for` 指令进行遍历

然后给 `botton` 标签绑定点击事件，我们预期点击按钮时，数据新增一个属性，界面也 新增一行

HTML 复制代码

```html
<p v-for="(value,key) in item" :key="key">
    {{ value }}
</p>
<button @click="addProperty">动态添加新属性</button>
```

实例化一个 `vue` 实例，定义 `data` 属性和 `methods` 方法

JavaScript 复制代码

```javascript
const app = new Vue({
    el:"#app",
    data:()=>{
        item:{
            oldProperty:"旧属性"
        }
    },
    methods:{
        addProperty(){
            this.items.newProperty = "新属性"  // 为items添加新属性
            console.log(this.items)  // 输出带有newProperty的items
        }
    }
})
```

点击按钮，发现结果不及预期，数据虽然更新了（`console`打印出了新属性），但页面并没有更新

5.2. 原理分析

为什么产生上面的情况呢?

下面来分析一下

`vue2`是用过`Object.defineProperty`实现数据响应式

JavaScript 复制代码

```
const obj = {}
Object.defineProperty(obj, 'foo', {
        get() {
            console.log(`get foo:${val}`);
            return val
        },
        set(newVal) {
            if (newVal !== val) {
                console.log(`set foo:${newVal}`);
                val = newVal
            }
        }
    })
}
```

当我们访问`foo`属性或者设置`foo`值的时候都能够触发`setter`与`getter`

JavaScript 复制代码

```
obj.foo
obj.foo = 'new'
```

但是我们为`obj`添加新属性的时候，却无法触发事件属性的拦截

JavaScript 复制代码

```
obj.bar  = '新属性'
```

原因是一开始`obj`的`foo`属性被设成了响应式数据，而`bar`是后面新增的属性，并没有通过`Object.defineProperty`设置成响应式数据

5.3. 解决方案

`Vue` 不允许在已经创建的实例上动态添加新的响应式属性

若想实现数据与视图同步更新，可采取下面三种解决方案：

- Vue.set()
- Object.assign()
- $forcecUpdated()

5.3.1. Vue.set()

Vue.set(target, propertyName/index, value)

参数

- `{Object | Array} target`
- `{string | number} propertyName/index`
- `{any} value`

返回值：设置的值

通过 `Vue.set` 向响应式对象中添加一个 `property`，并确保这个新 `property` 同样是响应式的，且触发视图更新

关于 `Vue.set` 源码（省略了很多与本节不相关的代码）

源码位置：`src\core\observer\index.js`

JavaScript | 复制代码

```
function set (target: Array<any> | Object, key: any, val: any): any {
  ...
  defineReactive(ob.value, key, val)
  ob.dep.notify()
  return val
}
```

这里无非再次调用 `defineReactive` 方法，实现新增属性的响应式

关于 `defineReactive` 方法，内部还是通过 `Object.defineProperty` 实现属性拦截

大致代码如下：

```javascript
function defineReactive(obj, key, val) {
    Object.defineProperty(obj, key, {
        get() {
            console.log(`get ${key}:${val}`);
            return val
        },
        set(newVal) {
            if (newVal !== val) {
                console.log(`set ${key}:${newVal}`);
                val = newVal
            }
        }
    })
}
```

5.3.2. Object.assign()

直接使用 `Object.assign()` 添加到对象的新属性不会触发更新

应创建一个新的对象，合并原对象和混入对象的属性

```javascript
this.someObject = Object.assign({},this.someObject,{newProperty1:1,newProperty2:2 ...})
```

5.3.3. $forceUpdate

如果你发现你自己需要在 `Vue` 中做一次强制更新，99.9% 的情况，是你在某个地方做错了事

`$forceUpdate` 迫使 `Vue` 实例重新渲染

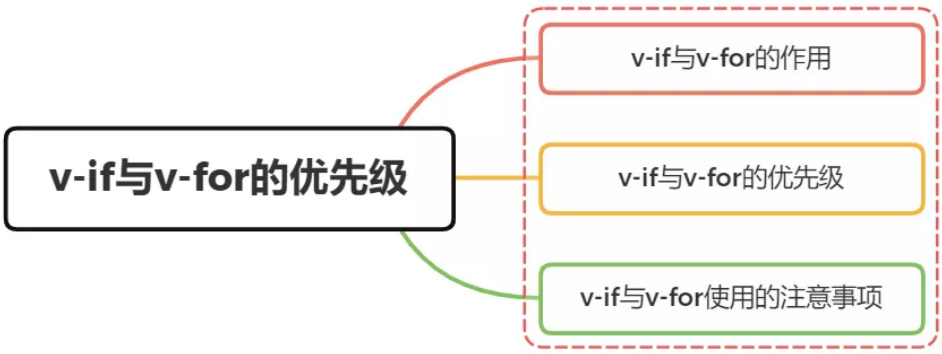
PS：仅仅影响实例本身和插入插槽内容的子组件，而不是所有子组件。

5.4. 小结

- 如果为对象添加少量的新属性，可以直接采用 `Vue.set()`
- 如果需要为新对象添加大量的新属性，则通过 `Object.assign()` 创建新对象
- 如果你实在不知道怎么操作时，可采取 `$forceUpdate()` 进行强制刷新 (不建议)

PS：`vue3` 是用过 `proxy` 实现数据响应式的，直接动态添加新属性仍可以实现数据响应式

6. v-if和v-for的优先级是什么?



6.1. 作用

`v-if` 指令用于条件性地渲染一块内容。这块内容只会在指令的表达式返回 `true` 值的时候被渲染

`v-for` 指令基于一个数组来渲染一个列表。`v-for` 指令需要使用 `item in items` 形式的特殊语法，其中 `items` 是源数据数组或者对象，而 `item` 则是被迭代的数组元素的别名

在 `v-for` 的时候，建议设置 `key` 值，并且保证每个 `key` 值是独一无二的，这便于 `diff` 算法进行优化

两者在用法上

JavaScript | 复制代码

```
<Modal v-if="isShow" />

<li v-for="item in items" :key="item.id">
    {{ item.label }}
</li>
```

6.2. 优先级

`v-if` 与 `v-for` 都是 `vue` 模板系统中的指令

在 `vue` 模板编译的时候，会将指令系统转化成可执行的 `render` 函数

6.2.1. 示例

编写一个 `p` 标签，同时使用 `v-if` 与 `v-for`

HTML 复制代码

```html
<div id="app">
    <p v-if="isShow" v-for="item in items">
        {{ item.title }}
    </p>
</div>
```

创建 `vue` 实例，存放 `isShow` 与 `items` 数据

JavaScript 复制代码

```javascript
const app = new Vue({
  el: "#app",
  data() {
    return {
      items: [
        { title: "foo" },
        { title: "baz" }]
    }
  },
  computed: {
    isShow() {
      return this.items && this.items.length > 0
    }
  }
})
```

模板指令的代码都会生成在 `render` 函数中，通过 `app.$options.render` 就能得到渲染函数

JavaScript 复制代码

```javascript
ƒ anonymous() {
  with (this) { return
    _c('div', { attrs: { "id": "app" } },
    _l((items), function (item)
    { return (isShow) ? _c('p', [_v("\n" + _s(item.title) + "\n")]) : _e()
}), 0) }
}
```

`_l` 是 `vue` 的列表渲染函数，函数内部都会进行一次 `if` 判断

初步得到结论： `v-for` 优先级是比 `v-if` 高