

课程目标



Prop & Data



计算属性和侦听器



数组操作

Prop和Data

Data

```
data() {  
  return {  
    classmates: [  
      { id: 1, name: "Han Meimei" },  
      { id: 2, name: "Li Lei" },  
      { id: 3, name: "Lily" }  
    ]  
  };  
},
```

vue中的data:当vue对象创建的时候，会将data函数返回对象中所有属性都放入响应式系统当中。

data为什么是函数？因为只有返回一个生产data的函数，这个组件产生的每一个实例才能维持一份被返回对象的独立的拷贝，让各个组件实例维护各自的数据，而单纯的写成对象形式，就使得所有组件实例共用了一份data，就会造成一个变了全都会变的结果

单向数据流

vue从2.x开始是一个单向数据流,为什么呢？因为父组件props的更新，会向下流入子组件。但是反过来在子组件修改props则不行。这样的做的目的是防止子组件意外的修改父组件的状态，导致应用的状态无法理解。为什么这么说呢？js的对象是通过引入传入的，所以在对于一个数组或者对象类型的Prop来说，在子组件中去改变props对象或者数据本身，将

影响父组件之间的状态，如果组件层级比较多时，将定位不到是哪个层级的组件修改了 props 数据。

Prop

父组件向子组件

Prop 属性校验

```
1  export default {
2    props: {
3      parentName: {
4        type: String,
5        required: true,
6        default: "banjitino",
7        validator(value) {
8          const nameEnums = ["banjitino", "Sam"];
9          return nameEnums.indexOf(value) !== -1;
10       }
11     }
12   }
13 };
```

规则	备注
type	1. 类型检查，value为对应类型的构造函数 2. null 和 undefined 会通过任何类型验证 3. 多种类型传入数组 [String, Array]
required	是否必填
default	默认值，默认值为对象或者数组的默认值必须从一个工厂函数获取
validator	自定义校验函数，返回true为通过，false为不通过

计算属性和侦听器

计算属性

```
1 <template>
2   <div class="hello">
3     <div v-for="(p, index) in classmates" :key="p.id">
4       {{ `${index}.${p.name}` }}:{{p.count}}
5       <button @click="add(index)">+</button>
6       <button @click="del(index)">-</button>
7     </div>
8     <div>count: {{ count }}</div>
9   </div>
10 </template>
11 <script>
12 export default {
13   data() {
14     return {
15       classmates: [
16         { id: 1, name: "Han Meimei", count: 0 },
17         { id: 2, name: "Li Lei", count: 0 },
18         { id: 3, name: "Lily", count: 0 }
19       ]
20     };
21   },
22   computed: {
23     count() {
24       return this.classmates.reduce((p, c) => {
25         p = p + c.count;
26         return p;
27       }, 0);
28     }
29   },
30   methods: {
31     add(i) {
32       this.classmates[i].count += 1;
33     },
34     del(i) {
35       this.classmates[i].count -= 1;
36     }
37   }
38 }
```

直接在count里面添加的一个计算属性
无需在data里面添加。

- 计算属性是基于其内部的响应式依赖进行缓存的(上图的依赖就是this.classmates)
- 只有相关响应式依赖发生改变时才会重新赋值。

上述使用计算属性，也可以使用一个方法，然后在模板中调用

```

    <button @click="add(index)">+</button>
    <button @click="del(index)">-</button>
  </div>
  <div>count: {{ count() }}</div>
</div>
</template>
<script>
export default {
  data() {
    return {
      classmates: [
        { id: 1, name: "Han Meimei", count: 0 },
        { id: 2, name: "Li Lei", count: 0 },
        { id: 3, name: "Lily", count: 0 }
      ]
    };
  },
  // computed: {
  //   count() {
  //     return this.classmates.reduce((p, c) => {
  //       p = p + c.count;
  //       return p;
  //     }, 0);
  //   }
  // },
  methods: {
    count() {
      return this.classmates.reduce((p, c) => {
        p = p + c.count;
        return p;
      }, 0);
    },
    add(i) {
      this.classmates[i].count += 1;
    },
    del(i) {
      this.classmates[i].count -= 1;
    }
  }
}

```

方法调用

- 方法是无缓存
- 每当触发重新渲染的时候，调用方法总会再次执行函数。

从两个方式方法的比较 计算属性是依赖进行缓存的，而方法是无缓存的。性能上计算属性方式更好。

计算属性的一般用于计算，我们也可以将一些不改动的常量内容，放入计算属性中，让其缓存。

```
1  computed: {
2    constant: function() {
3      return CONSTANT;
4    }
5  },
```

常量

侦听器

watch函数:

```
1  <template>
2    <div>
3      {{ count }}
4      <button @click="count = count + 1">add</button>
5    </div>
6  </template>
7  <script>
8    const log = index => index;
9    export default {
10     data() {
11       return {
12         count: 0
13       };
14     },
15     watch: {
16       count() {
17         log(this.count);
18       }
19     }
20   };
21 </script>
```

computed和watch都可以监听数据的变化，那有什么区别呢？computed是依赖数据的一个变化然后进行计算的，如果这个计算量比较大，比较消耗内存，那会阻塞页面渲染。就不建议用computed。这时候在数据变化执行异步操作或者开销比较大的操作，我们需要使用watch函数

数组操作

查看下面数组的操作，出现以下问题。

```
1  data() {
2    return {
3      classmates: [
4        { id: 1, name: "Han Meimei" },
5        { id: 2, name: "Li Lei" },
6        { id: 3, name: "Lily" }
7      ]
8    };
9  },
10 methods: {
11   change() {
12     this.classmates[2] = { id: 4, name: "Wang" }
13   }
14 }
```

修改数组，并不会发生视图的刷新

查看vue2.0的源码我们可以到，我们的响应式是Object.defineProperty() 对data属性进行一个遍历，并把它转化为get和set，get和set对于用户来说是不可见的，但是其内部是可以追踪依赖的，在属性访问和改变的时候通知变更。这导致了vue在数据操作的局限性。

- 不能检测对象属性的添加和删除
- 不能检测到数组长度变化(通过改变length而增加的长度不能检测到)
- 不是因为defineProperty的局限性，而是出于性能考量的，不会对数组每个元素监听(为什么要这么设计呢？数组一般是用来遍历一些列表的，那如果这个列表是一个海量的数据，大家都知道defineProperty是需要的一个前置依赖信息的收集的，所以一般都要先在data定义好，这个依赖收集就会变得很庞大，这个就会比较消耗性能)

解决方法:

通过全局set() 方法添加修改

```
1 change() {  
2   // this.classmates[2] = { id: 4, name: "Wang" };  
3   Vue.set(this.classmates, "2", { id: 4, name: "Wang" });  
4  
5   this.$set(this.classmates, "2", { id: 4, name: "Wang" });  
6 }
```

通过的全局delete()方法删除

```
1 change() {  
2   Vue.delete(this.classmates, "2");  
3  
4   this.$delete(this.classmates, "2");  
5 }
```

以上是全局的几个修改方法。不过vue对数组操作的相关方法进行了一次代理包装，在代理包装的过程中其实就是把这些函数加入了响应式的跟踪依赖，所以可以正常的响应式数据绑定。下面操作的数组的方法加入了响应式的跟踪依赖。

- push()
- pop()
- shift()
- unshift()
- splice()
- sort()
- reverse()