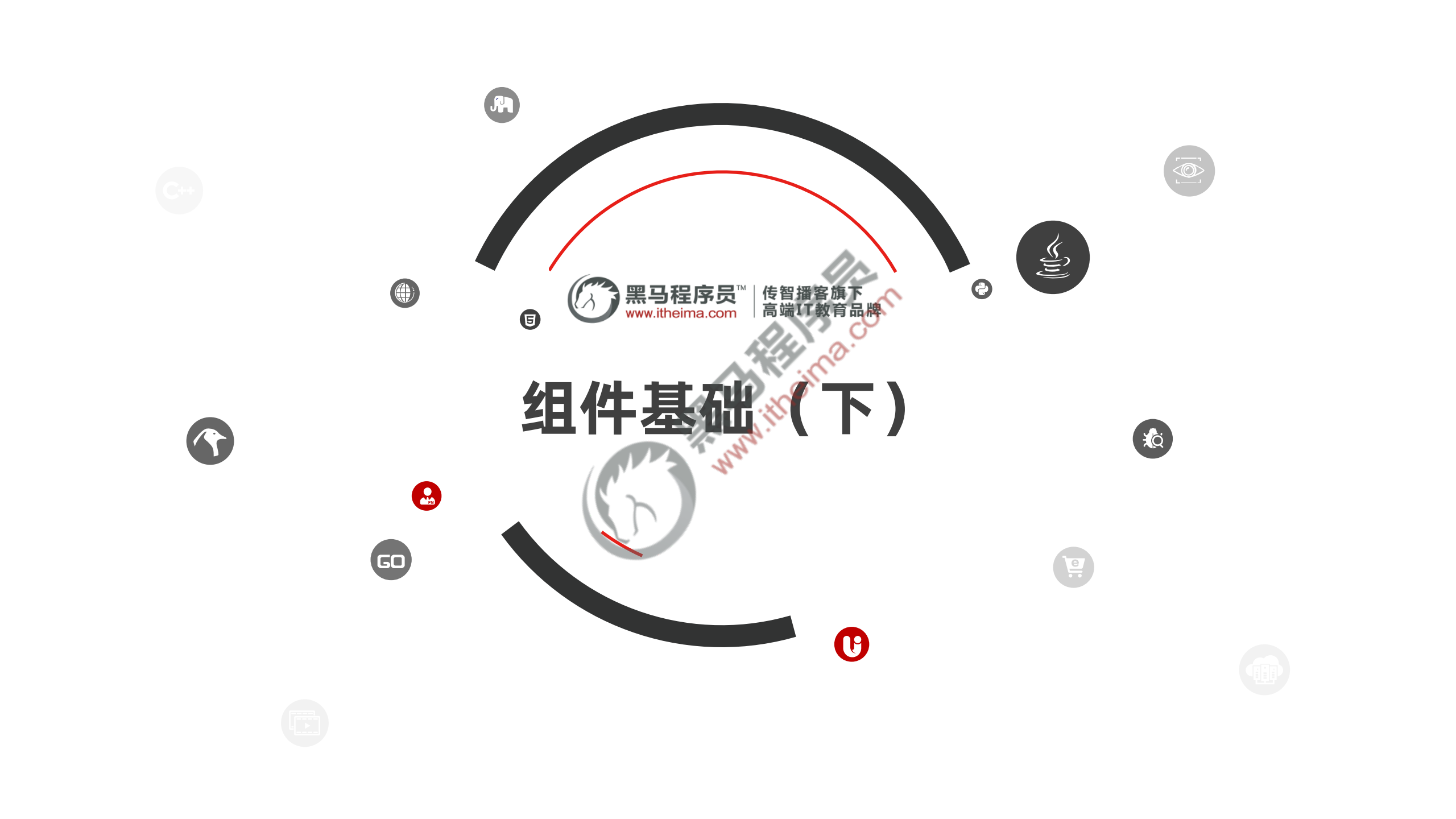


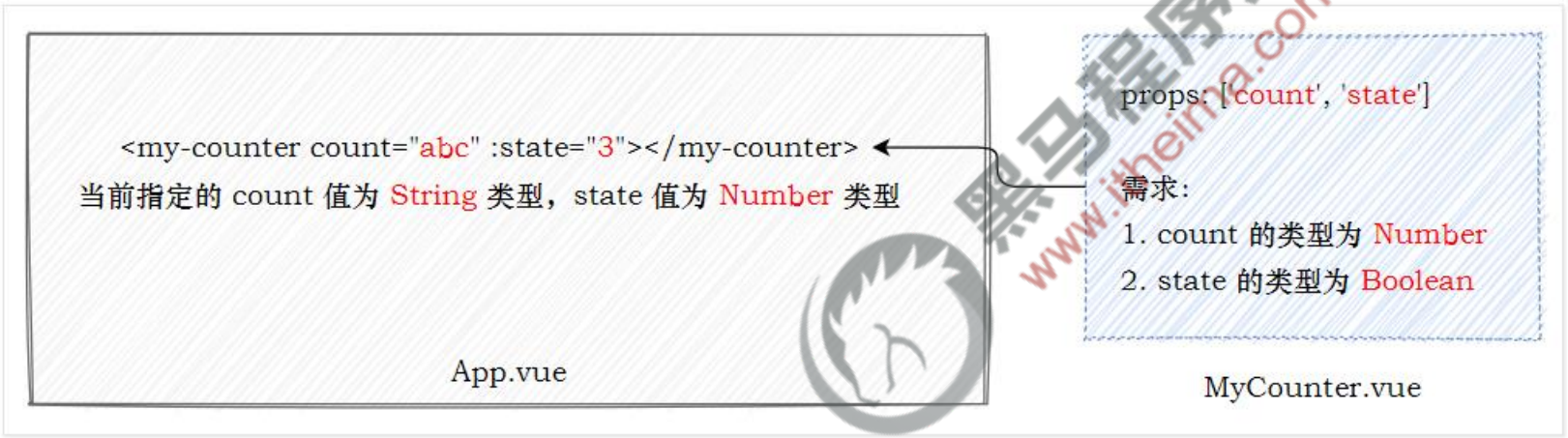
组件基础（下）

目录 Contents

props 验证

1. 什么是 props 验证

指的是：在封装组件时对外界传递过来的 props 数据进行合法性的校验，从而防止数据不合法的问题。



使用数组类型的 props 节点的缺点：无法为每个 prop 指定具体的数据类型。

props 验证

2. 对象类型的 props 节点

使用对象类型的 props 节点，可以对每个 prop 进行数据类型的校验，示意图如下：

3. props 验证

对象类型的 props 节点提供了多种数据验证方案，例如：

① 基础的类型检查

② 多个可能的类型

③ 必填项校验

④ 属性默认值

⑤ 自定义验证函数

props 验证

3.1 基础的类型检查

可以直接为组件的 prop 属性指定基础的校验类型，从而防止组件的使用者为其绑定错误类型的数据：

```
export default {
  props: { // 支持的 8 种基础类型
    propA: String,   // 字符串类型
    propB: Number,   // 数字类型
    propC: Boolean,  // 布尔值类型
    propD: Array,    // 数组类型
    propE: Object,   // 对象类型
    propF: Date,     // 日期类型
    propG: Function, // 函数类型
    propH: Symbol    // 符号类型
  },
}
```

3.2 多个可能的类型

如果某个 prop 属性值的类型不唯一，此时可以通过数组的形式，为其指定多个可能的类型，示例代码如下：

```
export default {
  props: {
    // propA 属性的值可以是“字符串”或“数字”
    propA: [String, Number],
  },
}
```

3.3 必填项校验

如果组件的某个 prop 属性是必填项，必须让组件的使用者为其传递属性的值。此时，可以通过如下的方式将其设置为必填项：

```
export default {
  props: {
    // 通过“配置对象”的形式，来定义 propB 属性的“验证规则”
    propB: {
      type: String,  // 当前属性的值必须是 String 字符串类型
      required: true // 当前属性的值是必填项，如果使用者没指定 propB 属性的值，则在终端进行警告提示
    },
  },
}
```

3.4 属性默认值

在封装组件时，可以为某个 prop 属性指定默认值。示例代码如下：

```
export default {
  props: {
    // 通过“配置对象”的形式，来定义 propC 属性的“验证规则”
    propC: {
      type: Number,
      default: 100  // 如果使用者没有指定 propC 的值，则 propC 属性的默认值为 100
    },
  },
}
```

props 验证

3.5 自定义验证函数

在封装组件时，可以为 prop 属性指定自定义的验证函数，从而对 prop 属性的值进行更加精确的控制：

```
export default {
  props: {
    // 通过“配置对象”的形式，来定义 propD 属性的“验证规则”
    propD: {
      // 通过 validator 函数，对 propD 属性的值进行校验，“属性的值”可以通过形参 value 进行接收
      validator(value) {
        // propD 属性的值，必须匹配下列字符串中的一个
        // validator 函数的返回值为 true 表示验证通过，false 表示验证失败
        return ['success', 'warning', 'danger'].indexOf(value) !== -1
      }
    },
  },
}
```

目录 Contents

计算属性

1. 什么是计算属性

计算属性本质上就是一个 function 函数，它可以实时监听 data 中数据的变化，并 return 一个计算后的新值，供组件渲染 DOM 时使用。

计算属性

2. 如何声明计算属性

计算属性需要以 function 函数的形式声明到组件的 computed 选项中，示例代码如下：

```
<input type="text" v-model.number="count" />
<p>{{ count }} 乘以 2 的值为：{{ plus }}</p>

data() {
  return { count: 1 }
},
computed: {
  plus() { // 计算属性：监听 data 中 count 值的变化，自动计算出 count * 2 之后的新值
    return this.count * 2
  },
}
```

注意：计算属性侧重于得到一个计算的结果，因此计算属性中必须有 return 返回值！

计算属性

3. 计算属性的使用注意点

① 计算属性必须定义在 computed 节点中

② 计算属性必须是一个 function 函数

③ 计算属性必须有返回值

④ 计算属性必须当做普通属性使用

计算属性

4. 计算属性 vs 方法

相对于方法来说，计算属性会缓存计算的结果，只有计算属性的依赖项发生变化时，才会重新进行运算。因此计算属性的性能更好：

```
computed: {
  plus() { // 计算属性的计算结果会被缓存，性能好
    console.log('计算属性被执行了')
    return this.count * 2
  },
},
methods: {
  plus() { // 方法的计算结果无法被缓存，性能低
    console.log('方法被执行了')
    return this.count * 2
  }
},
```

计算属性

5. 计算属性案例

案例需求，使用计算属性动态计算：

① 已勾选的商品总个数

② 已勾选的商品总价

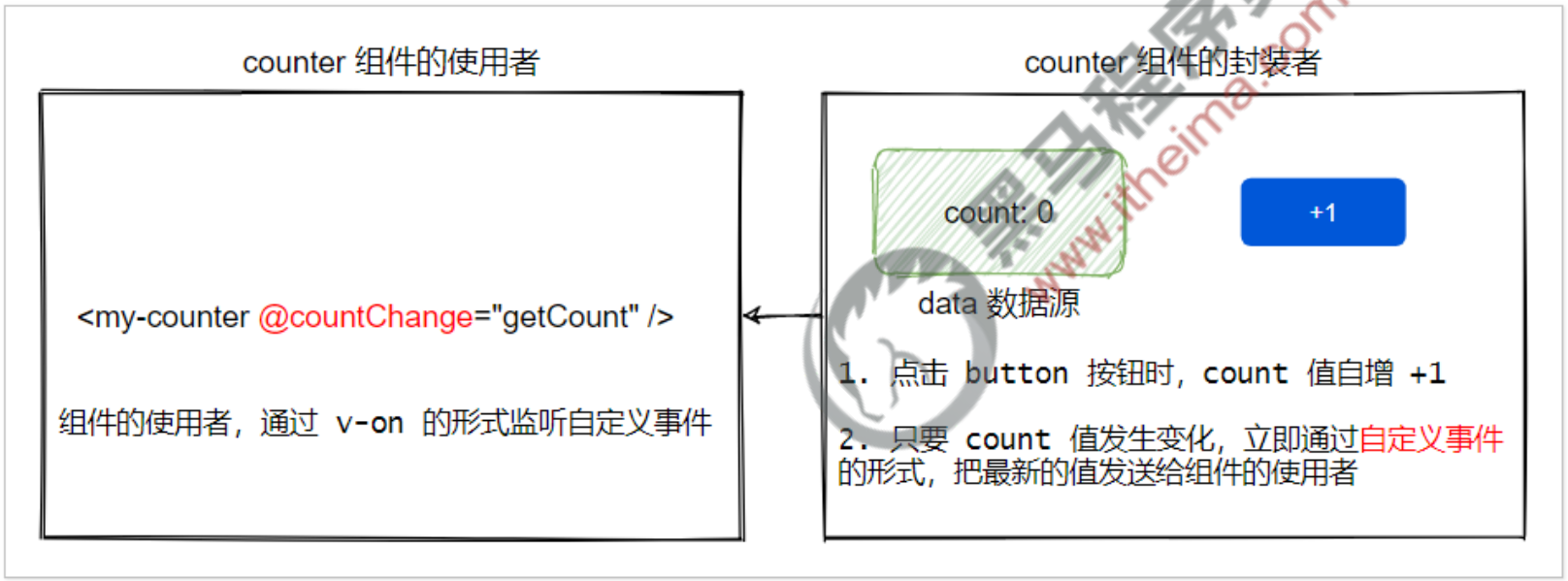
③ 结算按钮的禁用状态

目录 Contents

自定义事件

1. 什么是自定义事件

在封装组件时，为了让组件的使用者可以监听到组件内状态的变化，此时需要用到组件的自定义事件。



自定义事件

2. 自定义事件的 3 个使用步骤

在封装组件时：

① 声明自定义事件

② 触发自定义事件

在使用组件时：

③ 监听自定义事件

自定义事件

2.1 声明自定义事件

开发者为自定义组件封装的自定义事件，必须事先在 emits 节点中声明，示例代码如下：

```
<template>
  <h3>Counter组件</h3>
  <button>+1</button>
</template>

<script>
export default {
  // my-counter 组件的自定义事件，必须事先声明到 emits 节点中
  emits: ['change'],
}
</script>
```

自定义事件

2.2 触发自定义事件

在 emits 节点下声明的自定义事件，可以通过 this.$emit('自定义事件的名称') 方法进行触发，示例代码如下：

```vue
<template>
  <h3>Counter组件</h3>
  <button @click="onBtnClick">+1</button>
</template>

<script>
export default {
  emits: ['change'],
  methods: {
    onBtnClick() {
      this.$emit('change') // 当点击 +1 按钮时，调用 this.$emit() 方法，触发自定义的 change 事件
    },
  },
}
</script>
```

2.3 监听自定义事件

在使用自定义的组件时，可以通过 v-on 的形式监听自定义事件。示例代码如下：

```
<!-- 使用 v-on 指令绑定事件监听 -->
<my-counter @change="getCount"></my-counter>

methods: {
  getCount() {
    console.log('监听到了 count 值的变化！')
  }
}
```

自定义事件

3. 自定义事件传参

在调用 this.$emit() 方法触发自定义事件时，可以通过第 2 个参数为自定义事件传参，示例代码如下：

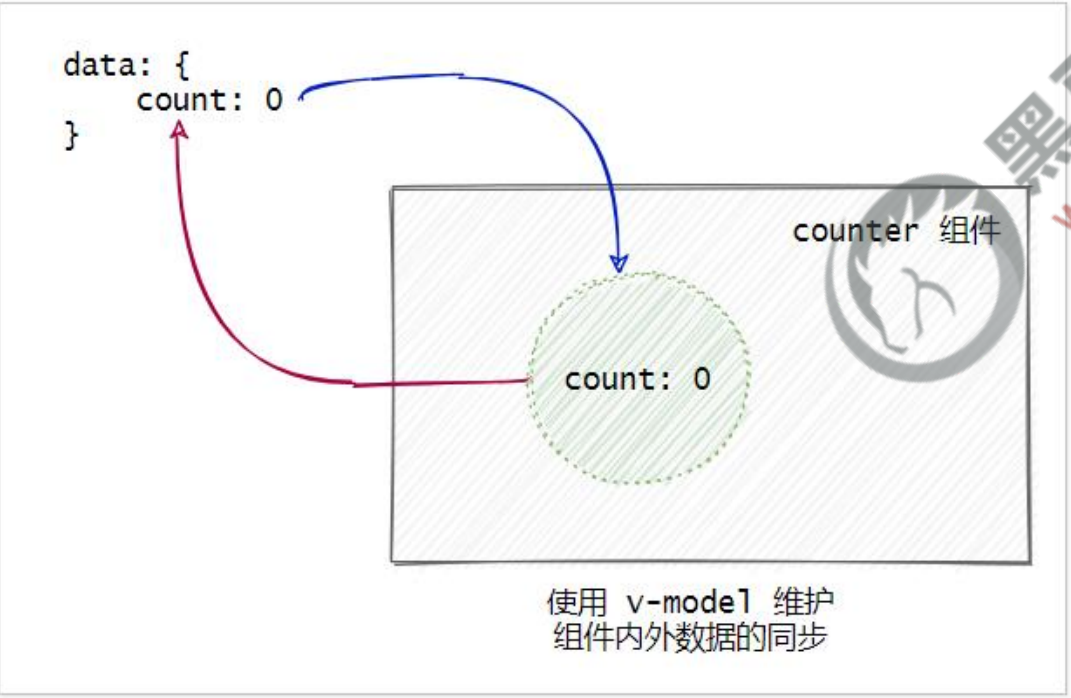
```vue
<template>
  <h3>Counter组件</h3>
  <button @click="onBtnClick">+1</button>
</template>

<script>
export default {
  emits: ['change'],
  methods: {
    onBtnClick() {
      this.$emit('change', this.count) // 触发自定义事件时，通过第二个参数传参
    },
  },
}
</script>
```

目录 Contents

组件上的 v-model

1. 为什么需要在组件上使用 v-model

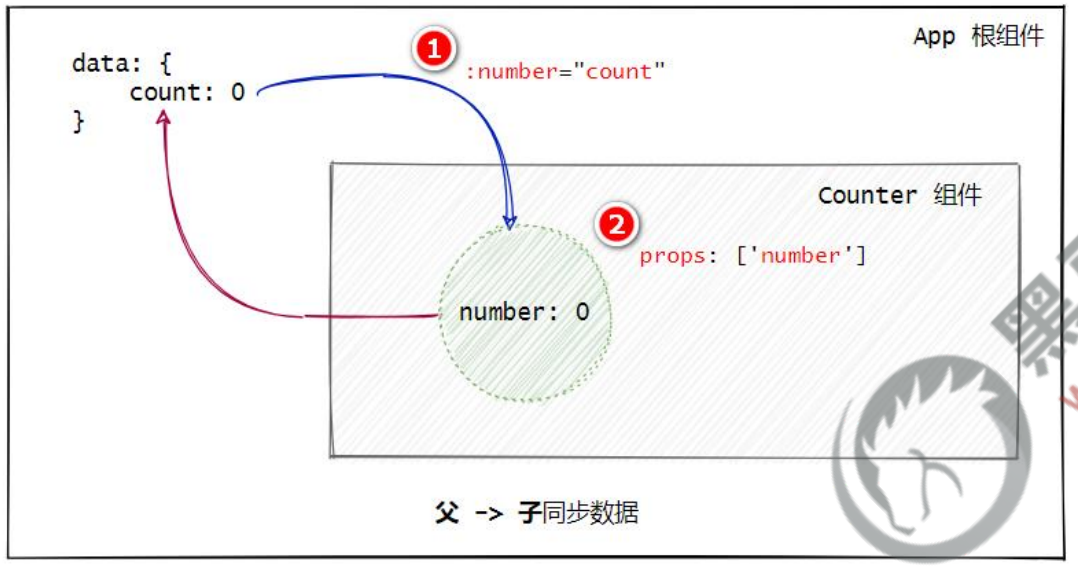
v-model 是双向数据绑定指令，当需要维护组件内外数据的同步时，可以在组件上使用 v-model 指令。示意图如下：



- 外界数据的变化会自动同步到 counter 组件中
- counter 组件中数据的变化，也会自动同步到外界

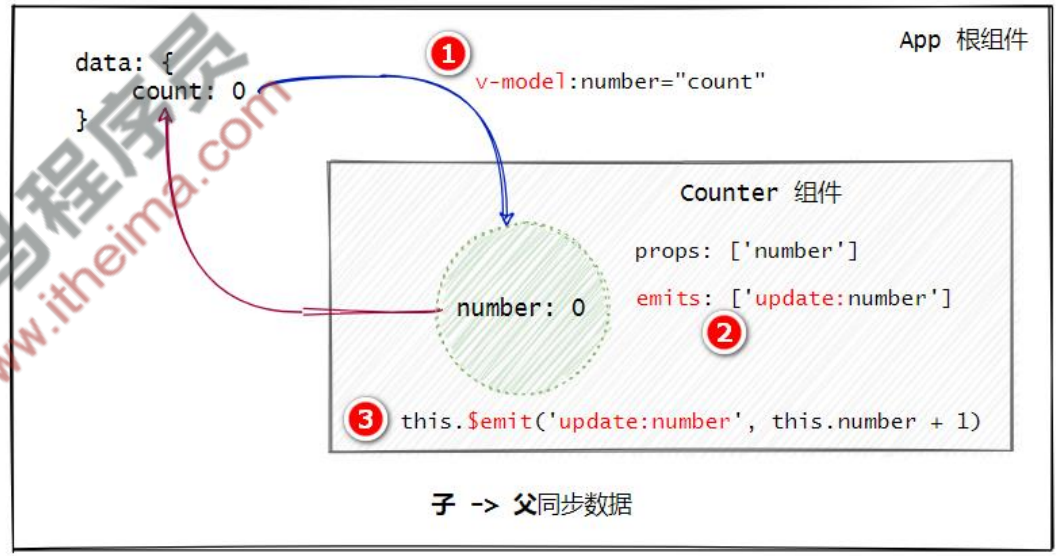
组件上的 v-model

2. 在组件上使用 v-model 的步骤



① 父组件通过 v-bind: 属性绑定的形式，把数据传递给子组件

② 子组件中，通过 props 接收父组件传递过来的数据



① 在 v-bind: 指令之前添加 v-model 指令

② 在子组件中声明 emits 自定义事件，格式为 update:xxx

③ 调用 $emit() 触发自定义事件，更新父组件中的数据

目录 Contents

任务列表案例

1. 案例效果

任务列表案例

2. 用到的知识点

① vite 创建项目

② 组件的封装与注册

③ props

④ 样式绑定

⑤ 计算属性

⑥ 自定义事件

⑦ 组件上的 v-model

任务列表案例

3. 整体实现步骤

① 使用 vite 初始化项目

② 梳理项目结构

③ 封装 todo-list 组件

④ 封装 todo-input 组件

⑤ 封装 todo-button 组件

① 能够知道如何对 props 进行验证

- 数组格式、对象格式
- type、default、required、validator

② 能够知道如何使用计算属性

- computed 节点、必须 return 一个结果、缓存计算结果

③ 能够知道如何为组件绑定自定义事件

- v-on 绑定自定义事件、emits、$emit()

④ 能够知道如何在组件上使用 v-model

- 应用场景：实现组件内外的数据同步
- v-model:props名称、emits、$emit('update:props名称')

黑马程序员

www.itheima.com

传智播客旗下高端IT教育品牌