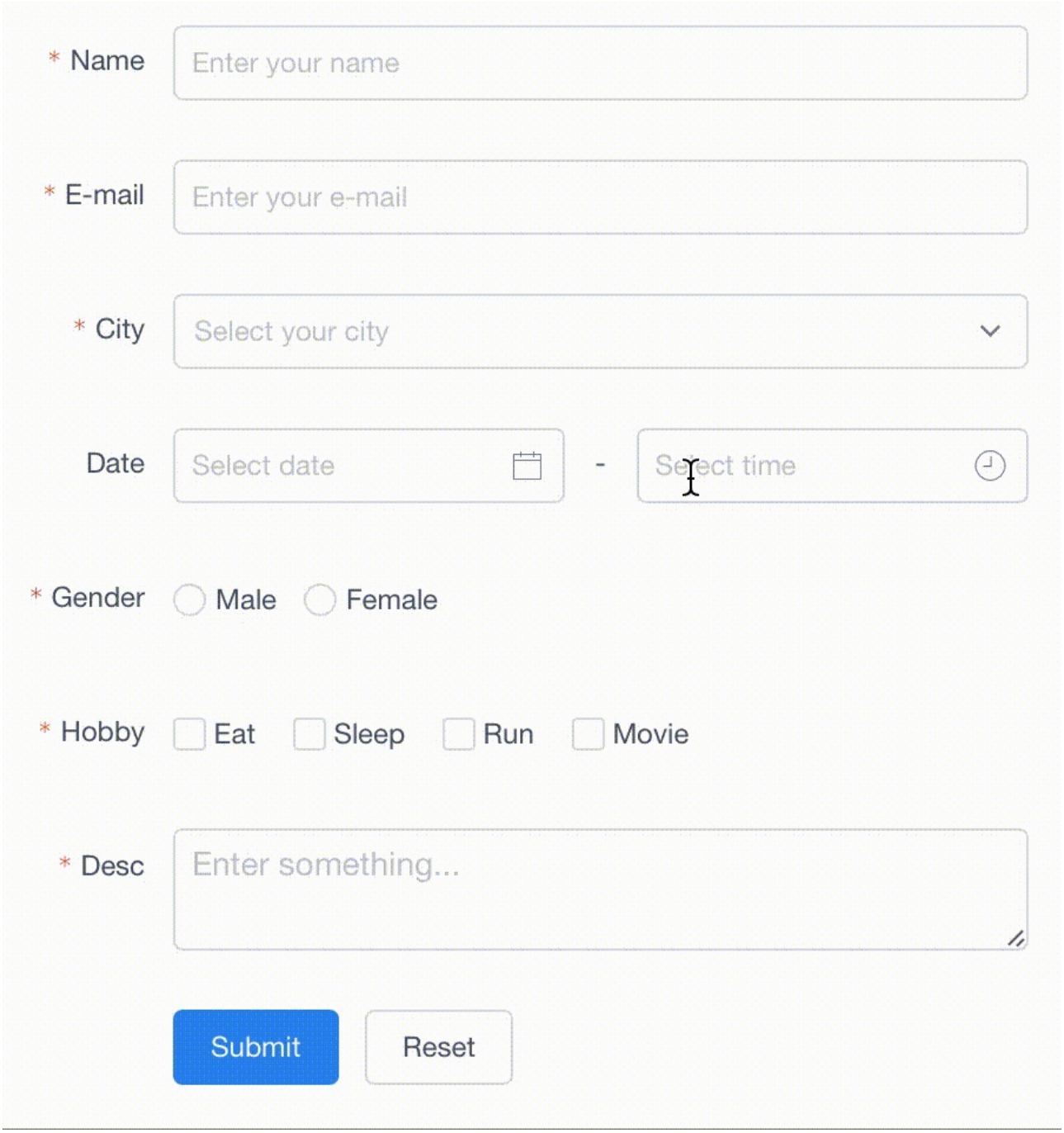
实战 1：具有数据校验功能的表单组件——Form

在第 3 节和第 4 节中，我们介绍了组件间的两种通信方法：provide / inject 和 dispatch / broadcast，前者是 Vue.js 内置的，主要用于子组件获取父组件（包括跨级）的状态；后者是自行实现的一种混合，用于父子组件（包括跨级）间通过自定义事件通信。本小节则基于这两种通信方法，来实现一个具有数据校验功能的表单组件——Form。

Form 组件概览

表单类组件在项目中会大量使用，比如输入框（Input）、单选（Radio）、多选（Checkbox）、下拉选择器（Select）等。在使用表单类组件时，也会经常用到数据校验，如果每次都写校验程序来对每一个表单控件校验，会很低效，因此需要一个能够校验基础表单控件的组件，也就是本节要完成的 Form 组件。一般的组件库都提供了这个组件，比如 iView，它能够校验内置的 15 种控件，且支持校验自定义组件，如下图所示：

（也可以在线访问本示例体验：https://run.iviewui.com/jwrqnFss (https://run.iviewui.com/jwrqnFss)）



Form 组件分为两个部分，一个是外层的 Form 表单域组件，一组表单控件只有一个 Form，而内部包含了多个 FormItem 组件，每一个表单控件都被一个 FormItem 包裹。基本的结构看起来像：

```
<i-form>
  <i-form-item>
    <i-input v-model="form.name"></i-input>
  </i-form-item>
  <i-form-item>
    <i-input v-model="form.mail"></i-input>
  </i-form-item>
</i-form>
```

Form 要用到数据校验，并在对应的 FormItem 中给出校验失败的提示，校验我们会用到一个开源库：[async-validator (https://github.com/yiminghe/async-validator)](https://github.com/yiminghe/async-validator)，基本主流的组件库都是基于它做的校验。使用它很简单，只需按要求写好一个校验规则就好，比如：

```
[
  { required: true, message: '邮箱不能为空',
trigger: 'blur' },
  { type: 'email', message: '邮箱格式不正确',
trigger: 'blur' }
]
```

这个代表要校验的数据先判断是否为空（required: true），如果为空，则提示“邮箱不能为空”，触发校验的事件为失焦（trigger: 'blur'），如果第一条满足要求，再进行第二条的验证，判断是否为邮箱格式（type: 'email'）等等，还支持自定义校验规则。更详细的用法可以参看它的文档。

接口设计

我们先使用最新的 Vue CLI 3 创建一个空白的项目（如果你还不清楚 Vue CLI 3 的用法，需要先补习一下了，可以阅读文末的扩展阅读 1），并使用 vue-router 插件，同时安装好 async-

validator 库。

在 src/components 下新建一个 form 文件夹，并初始化两个组件 form.vue 和 form-item.vue，然后初始化项目，配置路由，创建一个页面能够被访问到。

> 本节所有代码可以在 https://github.com/icarusion/vue-component-book (https://github.com/icarusion/vue-component-book) 中查看，你可以一边看源码，一边阅读本节；也可以边阅读，边动手实现一遍，遇到问题再参考完整的源码。

第 2 节我们介绍到，编写一个 Vue.js 组件，最重要的是设计好它的接口，一个 Vue.js 组件的接口来自三个部分：props、slots、events。而 Form 和 FormItem 两个组件主要做数据校验，用不到 events。Form 的 slot 就是一系列的 FormItem，FormItem 的 slot 就是具体的表单控件，比如输入框 <i-input> 。那主要设计的就是 props 了。

在 Form 组件中，定义两个 props：

- model：表单控件绑定的数据对象，在校验或重置时会访问该数据对象下对应的表单数据，类型为 Object。
- rules：表单验证规则，即上面介绍的 async-validator 所使用的校验规则，类型为 Object。

在 FormItem 组件中，也定义两个 props：

- label：单个表单组件的标签文本，类似原生的 <label> 元素，类型为 String。
- prop：对应表单域 Form 组件 model 里的字段，用于在校验或重置时访问表单组件绑定的数据，类型为 String。

定义好 props，就可以写出大概的用例了：

```
<template>
  <div>
    <i-form :model="formValidate"
:rules="ruleValidate">
      <i-form-item label="用户名" prop="name">
        <i-input v-model="formValidate.name"></i-
input>
      </i-form-item>
      <i-form-item label="邮箱" prop="mail">
        <i-input v-model="formValidate.mail"></i-
input>
      </i-form-item>
    </i-form>
  </div>
</template>
<script>
  import iForm from
'../components/form/form.vue';
  import iFormItem from '../components/form/form-
item.vue';
  import iInput from
'../components/input/input.vue';

  export default {
    components: { iForm, iFormItem, iInput },
    data () {
      return {
        formValidate: {
          name: '',
          mail: ''
        },
```

```
        ruleValidate: {
          name: [
            { required: true, message: '用户名不能为空', trigger: 'blur' }
          ],
          mail: [
            { required: true, message: '邮箱不能为空', trigger: 'blur' },
            { type: 'email', message: '邮箱格式不正确', trigger: 'blur' }
          ],
        }
      }
    }
  }
</script>
```

有两点需要注意的是：

1. 这里的 <i-input> 并不是原生的 <input> 输入框，而是一个特制的输入框组件，之后会介讲解的功能和代码；
2. <i-form-item> 的属性 prop 是字符串，所以它前面没有冒号（即不是 :prop="name"）。

当前的两个组件只是个框框，还没有实现任何功能，不过万事开头难，定义好接口，剩下的就是补全组件的逻辑，而对于使用者，知道了 props、events、slots，就已经能写出上例的使用代码了。

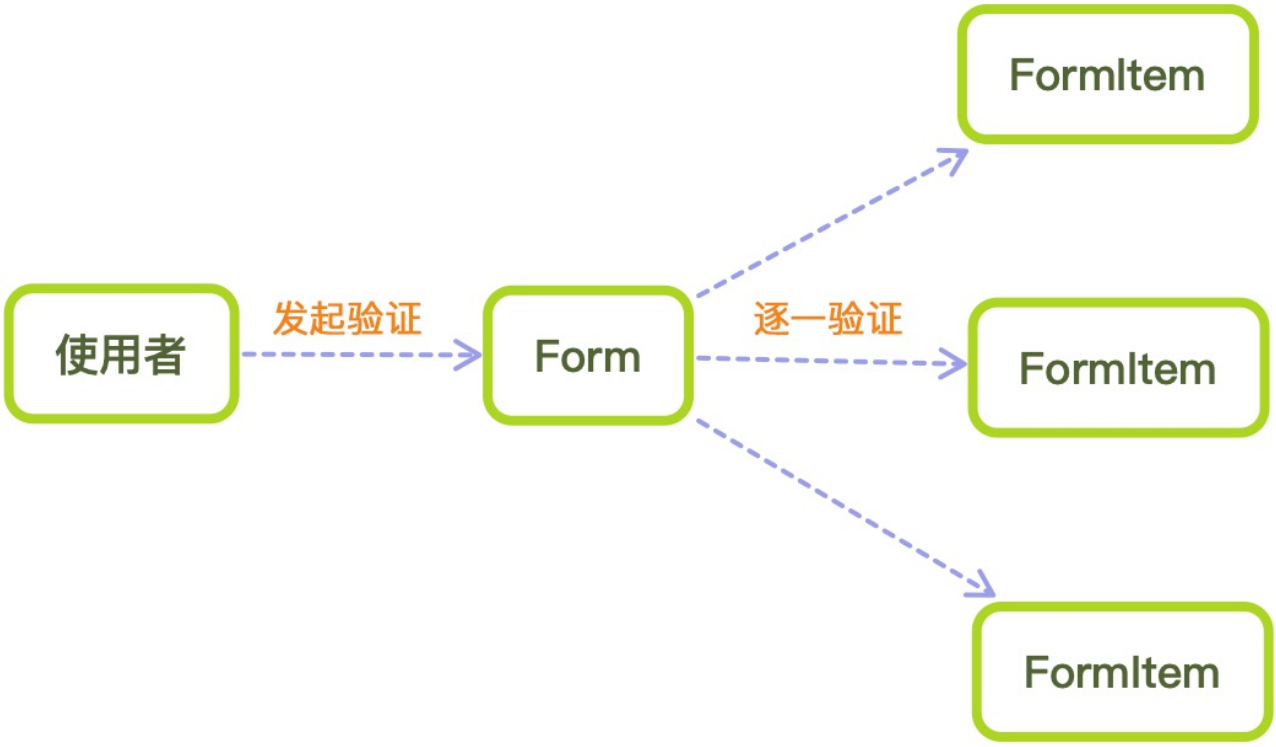
到此，Form 和 FormItem 的代码如下：

```
<!-- form.vue -->
<template>
  <form>
    <slot></slot>
  </form>
</template>
<script>
  export default {
    name: 'iForm',
    props: {
      model: {
        type: Object
      },
      rules: {
        type: Object
      }
    }
  }
</script>
```

```
<!-- form-item.vue -->
<template>
  <div>
    <label v-if="label">{{ label }}</label>
    <div>
      <slot></slot>
    </div>
  </div>
</template>
<script>
  export default {
    name: 'iFormItem',
    props: {
      label: {
        type: String,
        default: ''
      },
      prop: {
        type: String
      }
    }
  }
</script>
```

在 Form 中缓存 FormItem 实例

Form 组件的核心功能是数据校验，一个 Form 中包含了多个 FormItem，当点击提交按钮时，要逐一对每个 FormItem 内的表单组件校验，而校验是由使用者发起，并通过 Form 来调用每一个 FormItem 的验证方法，再将校验结果汇总后，通过 Form 返回出去。大致的流程如下图所示：



因为要在 Form 中逐一调用 FormItem 的验证方法，而 Form 和 FormItem 是独立的，需要预先将 FormItem 的每个实例缓存在 Form 中，这个操作就需要用到第 4 节的组件通信方法。当每个 FormItem 渲染时，将其自身（this）作为参数通过 dispatch 派发到 Form 组件中，然后通过一个数组缓存起来；同理当 FormItem 销毁时，将其从 Form 缓存的数组中移除。相关代码如下：

```
// form-item.vue，部分代码省略

import Emitter from '../../mixins/emitter.js';

export default {
  name: 'iFormItem',
  mixins: [ Emitter ],
  // 组件渲染时，将实例缓存在 Form 中
  mounted () {
    // 如果没有传入 prop，则无需校验，也就无需缓存
    if (this.prop) {
      this.dispatch('iForm', 'on-form-item-add',
this);
    }
  },
  // 组件销毁前，将实例从 Form 的缓存中移除
  beforeDestroy () {
    this.dispatch('iForm', 'on-form-item-remove',
this);
  }
}
```

注意，Vue.js 的组件渲染顺序是由内而外的，所以 FormItem 要先于 Form 渲染，在 FormItem 的 mounted 触发时，我们向 Form 派发了事件 on-form-item-add，并将当前 FormItem 的实例（this）传递给了 Form，而此时，Form 的 mounted 尚未触发，因为 Form 在最外层，如果在 Form 的 mounted 里监听事件，是不可以的，所以要在其 created 内监听自定义事件，Form 的 created 要先于 FormItem 的 mounted。所以 Form 的相关代码为：

```
// form.vue, 部分代码省略
export default {
  name: 'iForm',
  data () {
    return {
      fields: []
    };
  },
  created () {
    this.$on('on-form-item-add', (field) => {
      if (field) this.fields.push(field);
    });
    this.$on('on-form-item-remove', (field) => {
      if (field.prop)
this.fields.splice(this.fields.indexOf(field),
1);
    });
  }
}
```

定义的数据 fields 就是用来缓存所有 FormItem 实例的。

触发校验

Form 支持两种事件来触发校验：

- **blur**：失去焦点时触发，常见的有输入框失去焦点时触发校验；
- **change**：实时输入时触发或选择时触发，常见的有输入框实时输入时触发校验、下拉选择器选择项目时触发校验等。

以上两个事件，都是有具体的表单组件来触发的，我们先来编写一个简单的输入框组件 i-input。在 components 下新建目录 input，并创建文件 input.vue：

```
<!-- input.vue -->
<template>
  <input
         type="text"
         :value="currentValue"
         @input="handleInput"
         @blur="handleBlur"
         />
</template>
<script>
  import Emitter from '../../mixins/emitter.js';

  export default {
    name: 'iInput',
    mixins: [ Emitter ],
    props: {
      value: {
        type: String,
        default: ''
      },
    },
    data () {
      return {
        currentValue: this.value
      }
    },
    watch: {
      value (val) {
        this.currentValue = val;
```

```
      }
    },
    methods: {
      handleInput (event) {
        const value = event.target.value;
        this.currentValue = value;
        this.$emit('input', value);
        this.dispatch('iFormItem', 'on-form-
change', value);
      },
      handleBlur () {
        this.dispatch('iFormItem', 'on-form-
blur', this.currentValue);
      }
    }
  }
</script>
```

Input 组件中，绑定在 <input> 元素上的原生事件 @input，每当输入一个字符，都会调用句柄 handleInput，并通过 dispatch 方法向上级的 FormItem 组件派发自定义事件 on-form-change；同理，绑定的原生事件 @blur 会在 input 失焦时触发，并传递事件 on-form-blur。

基础组件有了，接下来要做的，是在 FormItem 中监听来自 Input 组件派发的自定义事件。这里可以在 mounted 中监听，因为你的手速远赶不上组件渲染的速度，不过在 created 中监听也是没任何问题的。相关代码如下：

```
// form-item.vue，部分代码省略
export default {
  methods: {
    setRules () {
      this.$on('on-form-blur', this.onFieldBlur);
      this.$on('on-form-change',
this.onFieldChange);
    },
  },
  mounted () {
    if (this.prop) {
      this.dispatch('iForm', 'on-form-item-add',
this);
      this.setRules();
    }
  }
}
```

通过调用 setRules 方法，监听表单组件的两个事件，并绑定了句柄函数 onFieldBlur 和 onFieldChange，分别对应 blur 和 change 两种事件类型。当 onFieldBlur 或 onFieldChange 函数触发时，就意味着 FormItem 要对**当前的数据**进行一次校验。当前的数据，指的就是通过表单域 Form 中定义的 props：model，结合当前 FormItem 定义的 props：prop 来确定的数据，可以回顾上文写过的用例。

因为 FormItem 中只定义了数据源的某个 key 名称（即属性 prop），要拿到 Form 中 model 里的数据，需要用到第 3 节的通信方法 provide / inject。所以在 Form 中，把整个实例（this）向下提供，并在 FormItem 中注入：

```
// form.vue, 部分代码省略
export default {
  provide() {
    return {
      form : this
    };
  }
}
```

```
// form-item.vue, 部分代码省略
export default {
  inject: ['form']
}
```

准备好这些，接着就是最核心的校验功能了。blur 和 change 事件都会触发校验，它们调用同一个方法，只是参数不同。相关代码如下：

```
// form-item.vue, 部分代码省略
import AsyncValidator from 'async-validator';

export default {
  inject: ['form'],
  props: {
    prop: {
      type: String
    },
  },
  data () {
    return {
      validateState: '',  // 校验状态
      validateMessage: '',  // 校验不通过时的提示信息
    }
```

```
  },
  computed: {
    // 从 Form 的 model 中动态得到当前表单组件的数据
    fieldValue () {
      return this.form.model[this.prop];
    }
  },
  methods: {
    // 从 Form 的 rules 属性中，获取当前 FormItem 的
校验规则
    getRules () {
      let formRules = this.form.rules;

      formRules = formRules ?
formRules[this.prop] : [];

      return [].concat(formRules || []);
    },
    // 只支持 blur 和 change，所以过滤出符合要求的
rule 规则
    getFilteredRule (trigger) {
      const rules = this.getRules();
      return rules.filter(rule => !rule.trigger
|| rule.trigger.indexOf(trigger) !== -1);
    },
    /**
     * 校验数据
     * @param trigger 校验类型
     * @param callback 回调函数
     */
    validate(trigger, callback = function () {})
{
      let rules = this.getFilteredRule(trigger);
```

```
      if (!rules || rules.length === 0) {
        return true;
      }

      // 设置状态为校验中
      this.validateState = 'validating';

      // 以下为 async-validator 库的调用方法
      let descriptor = {};
      descriptor[this.prop] = rules;

      const validator = new
AsyncValidator(descriptor);
      let model = {};

      model[this.prop] = this.fieldValue;

      validator.validate(model, { firstFields:
true }, errors => {
        this.validateState = !errors ? 'success'
: 'error';
        this.validateMessage = errors ?
errors[0].message : '';

        callback(this.validateMessage);
      });
    },
    onFieldBlur() {
      this.validate('blur');
    },
    onFieldChange() {
      this.validate('change');
```

```
    }
  }
}
```

在 FormItem 的 validate() 方法中，最终做了两件事：

1. 设置了当前的校验状态 validateState 和校验不通过提示信息 validateMessage（通过值为空）；
2. 将 validateMessage 通过回调 callback 传递给调用者，这里的调用者是 onFieldBlur 和 onFieldChange，它们只传入了第一个参数 trigger，callback 并未传入，因此也不会触发回调，而这个回调主要是给 Form 用的，因为 Form 中可以通过提交按钮一次性校验所有的 FormItem（后文会介绍）这里只是表单组件触发事件时，对当前 FormItem 做校验。

除了校验，还可以对当前数据进行重置。重置是指将表单组件的数据还原到最初绑定的值，而不是清空，因此需要预先缓存一份初始值。同时我们将校验信息也显示在模板中，并加一些样式。相关代码如下：

```
<!-- form-item.vue, 部分代码省略 -->
<template>
  <div>
    <label v-if="label" :class="{ 'i-form-item-
label-required': isRequired }">{{ label }}
</label>
    <div>
      <slot></slot>
      <div v-if="validateState === 'error'"
class="i-form-item-message">{{ validateMessage }}
</div>
    </div>
  </div>
</template>
```

```
<script>
  export default {
    props: {
      label: {
        type: String,
        default: ''
      },
      prop: {
        type: String
      },
    },
    data () {
      return {
        isRequired: false,  // 是否为必填
        validateState: '',  // 校验状态
        validateMessage: '',  // 校验不通过时的提示信
息
      }
    },
    mounted () {
      // 如果没有传入 prop，则无需校验，也就无需缓存
      if (this.prop) {
        this.dispatch('iForm', 'on-form-item-
add', this);

        // 设置初始值，以便在重置时恢复默认值
        this.initialValue = this.fieldValue;

        this.setRules();
      }
    },
    methods: {
      setRules () {
```

```
        let rules = this.getRules();
        if (rules.length) {
          rules.every((rule) => {
            // 如果当前校验规则中有必填项，则标记出来
            this.isRequired = rule.required;
          });
        }

        this.$on('on-form-blur',
this.onFieldBlur);
        this.$on('on-form-change',
this.onFieldChange);
      },
      // 从 Form 的 rules 属性中，获取当前 FormItem
的校验规则
      getRules () {
        let formRules = this.form.rules;

        formRules = formRules ?
formRules[this.prop] : [];

        return [].concat(formRules || []);
      },
      // 重置数据
      resetField () {
        this.validateState = '';
        this.validateMessage = '';

        this.form.model[this.prop] =
this.initialValue;
      },
    }
  }
```

```
</script>
<style>
  .i-form-item-label-required:before {
    content: '*';
    color: red;
  }
  .i-form-item-message {
    color: red;
  }
</style>
```

至此，FormItem 代码已经完成，不过它只具有单独校验的功能，也就是说，只能对自己的一个表单组件验证，不能对一个表单域里的所有组件一次性全部校验。而实现全部校验和全部重置的功能，要在 Form 中完成。

上文已经介绍到，在 Form 组件中，预先缓存了全部的 FormItem 实例，自然也能在 Form 中调用它们。通过点击提交按钮全部校验，或点击重置按钮全部重置数据，只需要在 Form 中，逐一调用缓存的 FormItem 实例中的 validate 或 resetField 方法。相关代码如下：

```
// form.vue，部分代码省略
export default {
  data () {
    return {
      fields: []
    };
  },
  methods: {
    // 公开方法：全部重置数据
    resetFields() {
      this.fields.forEach(field => {
```

```
        field.resetField();
      });
    },
    // 公开方法：全部校验数据，支持 Promise
    validate(callback) {
      return new Promise(resolve => {
        let valid = true;
        let count = 0;
        this.fields.forEach(field => {
          field.validate('', errors => {
            if (errors) {
              valid = false;
            }
            if (++count === this.fields.length) {
              // 全部完成
              resolve(valid);
              if (typeof callback === 'function')
{
                callback(valid);
              }
            }
          });
        });
      });
    }
  },
}
```

虽然说 Vue.js 的 API 只来自 prop、event、slot 这三个部分，但一些场景下，需要通过 ref 来访问这个组件，调用它的一些内置方法，比如上面的 validate 和 resetFields 方法，就需要使用者来主动调用。

resetFields 很简单，就是通过循环逐一调用 FormItem 的 resetField 方法来重置数据。validate 稍显复杂，它支持两种使用方法，一种是普通的回调，比如：

```
<template>
  <div>
    <i-form ref="form"></i-form>
    <button @click="handleSubmit">提交</button>
  </div>
</template>
<script>
  export default {
    methods: {
      handleSubmit () {
        this.$refs.form.validate((valid) => {
          if (valid) {
            window.alert('提交成功');
          } else {
            window.alert('表单校验失败');
          }
        })
      }
    }
  }
</script>
```

同时也支持 Promise，例如：

```
handleSubmit () {
  const validate = this.$refs.form.validate();

  validate.then((valid) => {
    if (valid) {
      window.alert('提交成功');
    } else {
      window.alert('表单校验失败');
    }
  })
}
```

在 Form 组件定义的 Promise 中，只调用了 resolve(valid)，没有调用 reject()，因此不能直接使用 .catch() ，不过聪明的你稍作修改，肯定能够支持到！

完整的用例如下：

```
<template>
  <div>
    <h3>具有数据校验功能的表单组件--Form</h3>
    <i-form ref="form" :model="formValidate"
:rules="ruleValidate">
      <i-form-item label="用户名" prop="name">
        <i-input v-model="formValidate.name"></i-
input>
      </i-form-item>
      <i-form-item label="邮箱" prop="mail">
        <i-input v-model="formValidate.mail"></i-
input>
      </i-form-item>
    </i-form>
    <button @click="handleSubmit">提交</button>
```

```
    <button @click="handleReset">重置</button>
  </div>
</template>
<script>
  import iForm from
'../components/form/form.vue';
  import iFormItem from '../components/form/form-
item.vue';
  import iInput from
'../components/input/input.vue';

  export default {
    components: { iForm, iFormItem, iInput },
    data () {
      return {
        formValidate: {
          name: '',
          mail: ''
        },
        ruleValidate: {
          name: [
            { required: true, message: '用户名不能
为空', trigger: 'blur' }
          ],
          mail: [
            { required: true, message: '邮箱不能为
空', trigger: 'blur' },
            { type: 'email', message: '邮箱格式不正
确', trigger: 'blur' }
          ],
        }
      }
    },
```

```
    methods: {
      handleSubmit () {
        this.$refs.form.validate((valid) => {
          if (valid) {
            window.alert('提交成功');
          } else {
            window.alert('表单校验失败');
          }
        })
      },
      handleReset () {
        this.$refs.form.resetFields();
      }
    }
  }
</script>
```

运行效果：

完整的示例源码可通过 GitHub 查看：

https://github.com/icarusion/vue-component-book (https://github.com/icarusion/vue-component-book)

> 项目基于 Vue CLI 3 构建，下载安装依赖后，通过 npm run serve 可访问。

结语

组件最终的效果看起来有点 “low”，但它实现的功能却不简单。通过这个实战，你或许已经感受到本小册一开始说的，组件写到最后，都是在拼 JavaScript 功底。的确，Vue.js 组件为我们提供了一种新的代码组织形式，但归根到底，是离不开 JS 的。

这个实战，你应该对独立组件间的通信用法有进一步的认知了吧，不过，这还不是组件通信的终极方案，下一节，我们就来看看适用于任何场景的组件通信方案。

注：本节部分代码参考 iView (https://github.com/iview/iview/tree/2.0/src/components/fo

扩展阅读

- 一份超级详细的Vue-cli3.0使用教程 (https://juejin.im/post/5bdec6e8e51d4505327a8952)