从 0 - 1 实现内部组件库设计

前端组件系统设计思路及模式

设计先行原则：对于组件库实现是一方面，但具体样式的设计一定要与业务相结合。

详细设计接口：这里以 form 为例，讲解一下作为一个合格的组件库，我们需要设计哪些接口。

form 组件设计需要注意的内容：

- 校验
- API 设计

校验一般来讲，会依赖于一些基本的库，比如 async-validator，他是一个非常优秀的校验配置的 schema，允许我们通过简单配置，内部对我们输入的值进行校验。

async-validator 用法详解

async-validator 支持接受一定配置的校验规则，通过规则配置能够输出最终结果的校验关系。

```
import schema from 'async-validator'
// 校验规则
const descriptor = {
  username: [
    {
      required: true,
      message: '请填写用户名'
    },
    {
      min: 2,
      max: 20,
      message: '用户名长度为2-20'
    }
  ]
}
// 根据校验规则构造一个 validator
const validator = new schema(descriptor)
const data = {
  username: 'zhuawajiaoyu'
};
// 校验之后返回一个 promise
validator.validate(data)
  .then(() => {
    // 校验成功
  })
  .catch(({ errors, fields }) => {
  })
```

同时配置的 schema 中也支持使用自定义的 validator，通过 validator 配置，可以配置一个同步函数来进行校验。

```
var descriptor = {
  username: {
    type: "string",
    required: true,
    validator: (rule, value) => value === 'zhuawajiaoyu',
  }
}

// validator 中配置了一个校验器，校验传入的 value 是否是 zhuawajiaoyu
```

我们也可以通过 asyncValidator 配置，来接受一个 promise 进行异步的配置。

```
var descriptor = {
  username: {
    type: "string",
    required: true,
    asyncValidator: (rule, value) => {
      return new Promise((resolve, reject) => {
        if (value === 'zhuawajiaoyu') {
          return resolve();
```

```
            }
            return reject('用户名填写错误');
        });
    }
  }
}
```

async-validator 支持非常多的配置类型，以下是一些简介，具体使用时可以在文档中查阅。

- `string`：默认为字符串类型，校验的数据必须为字符串
- `number`：数字类型
- `boolean`：布尔类型
- `method`：函数类型
- `regexp`：必须是 `RegExp` 的实例，或者是个合法的正则字符串
- `integer`：必须是数字并且是整型
- `float`：必须是数字且必须是浮点数
- `array`：必须是 array，Array.isArray 返回 true
- `object`：必须是对象但不能是数组.
- `enum`：数据必须在 `enum` 配置中.
- `date`：必须是 Date 构造的合法值
- `url`：必须是 url
- `hex`：十六进制数.
- `email`：必须是 email.
- `any`：任何类型均可.

对于复杂的对象校验，我们可以通过 fields 配置，来深入的配置每一个 key 的内容

```
var descriptor = {
  address: {
    type: "object", required: true,
    fields: {
      street: {type: "string", required: true},
      city: {type: "string", required: true},
      zip: {type: "string", required: true,
len: 8, message: "invalid zip"}
    }
  },
  name: {type: "string", required: true}
}
```

async-validator 对于组件设计来说非常重要，对于要设计 Form 组件的同学来说，务必理解其中的内容

高阶组件封装 API

理解了校验之后，我们就能通过高阶组件来给基础组件上提供一些校验的方法，例如，我们通过 FormCreate 函数，接受一个函数，来返回一个新的组件。

```
function FormCreate(options) {
  const store = {};
```

```
  const form = {
    getFieldProps(fieldKey, options) {
      return {
        value: '', // 可以选择受控或者非受控
        key: fieldKey,
        onChange(e) {
          // 调用 validator 来进行校验，错误内容存入
内部状态中
        }
      }
    },
    getFieldError(fieldKey) {
      return {
        children={store[fieldKey].error}
      }
    }
  };
  return function(WrappedComponent) {
      return function() {
        return <WrappedComponent form={form} />

      }
  }
}


// 具体 Form 组件
```

```
class Form extends React.Component {
  render() {
    const form = this.props.form;
    return (
      <div>
        <input
{...form.getFieldProps('inputKey', {validator:
xxx})}></input>
        <div
{...form.getFieldError('inputKey')} />
      </div>
    )
  }
}
const FormInput = FormCreate()(Form);
```

我们直接通过高阶组件的形式，封装了一个内部的对象，对象内部暴露一些方法，可以最终返回组件上需要使用的 value，onChange 等等方法，onChange 的过程中我们就可以更新闭包内部的值的结果和调用校验。

上面的例子中的 Input 我们可以继续进行封装，封装之后，上层应用可以以更简单的方式来进行调用。

```
<Input fieldKey="inputKey" />
```

常见的组件级包装的 Form API 有：

- getFieldsValue/getFieldValue: 获取全部/单个组件内部的 value 值
- validateFields/validateField: 校验全部/单个组件的结果
- setFieldsValue/setFieldValue: 设置全部/单个组件的结果

一般来讲，我们的高阶组件中提供这些方法，就算是一个功能比较全面的 Form 组件。

上面的例子我们只是封装了一个具体的组件，同时，我们可以多封装一些这类的组件，最终达到更高级的效果。

```
<Form onSubmit={this.onSubmit}>
  <FormItem label="input" fieldKey="inputKey">
    <Input />
  </FormItem>
  <FormItem>
    <Select />
  </FormItem>
</Form>
```

这样封装之后，业务的调用方就能很方便的实现具体业务的书写。

其他需要注意的地方，例如 Form 中的 select，我们都知道原生的 select 样式有时不符合我们具体业务中 UI 的样式，所以需要我们对他进行定制，这时候其实我们自己模拟 div 这种下拉就更好。同理，radio 和 checkbox 我们都可以自定义他们的效果，达到更好的效果。

组件库开发、管理及调试模式

组件库一般会搭配 lerna 、 babel-plugin 或者一些可视化工具来进行设计、开发和管理。

lerna 是一个多 package 的包管理工具，地址：https://github.com/lerna/lerna

它的作用是处理多个包在有相互依赖，都需要发布的时候，通过 lerna 的一个命令，就能同时更新多个包的版本和代码。我们就能方便的组织我们的代码库结构。

它主要解决了以下几个问题：

1. 自动解决packages之间的依赖关系
2. 通过 `git` 检测文件改动，自动发布
3. 根据 `git` 提交记录，自动生成CHANGELOG

lerna 的使用也非常简单，只有两个命令 `lerna bootstrap` 和 `lerna publish` 。

lerna bootstrap 使 lerna 初始化整个项目，publish 使 lerna 来发布所有被索引的所有模块。

之后我们就可以通过例如 `BD-FORM` `BD-SELECT` 之类的包来直接引入我们的代码。

另一种优化是运行时的优化，以 `babel-plugin-import` 为例，来讲下组件库的优化。

我们使用 `import {Button} from 'antd'` 从 ant design 中引入一个 Button 组件，但是大部分情况下，我们的 webpack 等打包工具会把所有的 antd 组件打包进来，当然在后期配合 tree-shaking 可以达到优化的目的。

在不支持 tree-shaking 的打包环境下，我们就可以使用 babel 插件 import，来将它最终形如 `import Button from 'antd/Button'` 。这样，我们打包工具就只会夹在 antd/Button 文件的内容，而不会把所有 antd 加载进来。

```
import { Button } from 'antd';
ReactDOM.render(<Button>xxxx</Button>);

      ↓ ↓ ↓ ↓ ↓ ↓

var _button = require('antd/lib/button');
ReactDOM.render(<_button>xxxx</_button>);
```

配合 style 参数，还能按需加载样式 `{ "libraryName": "antd", style: "css" }`

```
import { Button } from 'antd';
ReactDOM.render(<Button>xxxx</Button>);

      ↓ ↓ ↓ ↓ ↓ ↓

var _button = require('antd/lib/button');
require('antd/lib/button/style/css');
ReactDOM.render(<_button>xxxx</_button>);
```

在我们业务实现的过程中，尤其在复杂场景，需要我们自定义一些插件来对我们最终的组件库进行一个优化，以便更好的让业务方来使用。

代码开发完成之后，我们通常需要预览，调试，并且向大家展示最终所有组件库的配置项。一般这种时候我们可以使用 storybook 等工具，来快速展示我们的组件内容，当然这类的文档工具有很多，我们可以自行选择。

https://github.com/storybookjs/storybook

组件库常见开发问题

1. 样式代码如何设计?

常见的组件库，我们的样式都是单独写在组件的文件中，最终所有的样式会进行一个合并打包，这就造成了我们在调用组件库的时候，还需要额外引入组件的样式。

这种方式对组件库的使用来说不算特别友好，同时全局的样式业务方在调用时容易覆盖。

例如，我们在使用 antd 的时候，必须引入它内部组件的样式，当然我们也可以按照组件来手动引入，不过那样确实使用起来很麻烦。

```
@import '~antd/dist/antd.css';
```

另一种方式，就是我们使用 css-in-js 方案，使用了这种方案之后，业务方在调用组件时，只需要引入一个 js 组件即可，组件的样式已经内连到了我们的组件 DOM 结构上。

```
import styled from 'styled-components';
const Title = styled.h1`
  font-size: 1.5em;
  text-align: center;
  color: palevioletred;
`;
<Title>Hello World</Title>

// 处理了样式集合为了 class
<h1 style="sc-AxjAm laiDrE"></h1>
<h1 style={{textAlien: 'center'}}></h1>
```

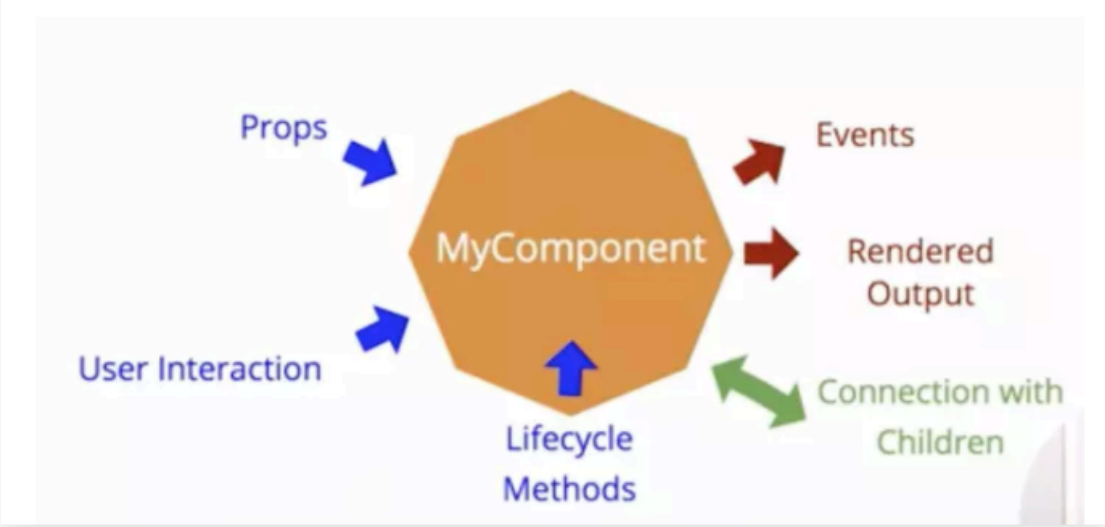
这样做的好处当然就是业务方比较轻松，因为不需要做太多额外的工作，但是这样做也有弊端，那就是组件库设计的过程中，有的必须有足够的弹性，能够应付部分业务需求对样式的定制，往往有些定制的需求，导致业务方在使用的时候需要使用 important，在设计时我们需要注意这种情况。

2. 主题、国际化、可访问性设计和单元测试的设计

主题、国际化、可访问性则需要我们在组件库中提前想好相关的设计，标签上预留对应的接口。同时也需要增加一些运行时配置项，保证可以通过接口进行获取更新。

大部分主题可以通过预处理框架来实现，比如 less、sass 等预处理框架，提前留好了变量和函数，只需要在业务使用的过程中，覆盖这些变量即可，这样编译出来的样式文件主题部分的样式也会改变。

在具体的实现中，组件库属于基础架构部分，所以对于组件库的发布和上线需要额外小心，一旦某个版本产生了问题，业务引入后可能会产生非常严重的后果。所以我们需要尽量写一部分单元测试，来保证代码发布过程中的 UI 一致和功能一致。



3. 代码与工程化问题

对于我们的组件代码来说，最好是将所有的源码进行编译，同时为了最终的代码大小和版本不会冲突，对于框架性质的东西，需要我们设置 peerDependiencies，这样最终我们的包内部不会出现版本依赖之类的问题。

同时对于代码静态检查，我们也可以使用例如 eslint 等工具，保证我们的静态代码的规范统一，不会出现一些低级 bug，同时也保证了代码格式的统一性。