组件库演示文档开发分享

无论是element还是ant design，或者其他一些pc端的组件库，大家在使用的时候都会发现，各组件库官网都会有一个网站给出代码示例，类似这样的演示区域：

> 这个模块对于一个组件库来说是至关重要的，不仅仅是给出演示更方便使用者理解使用，也方便开发者在开发组件库的时候调试。

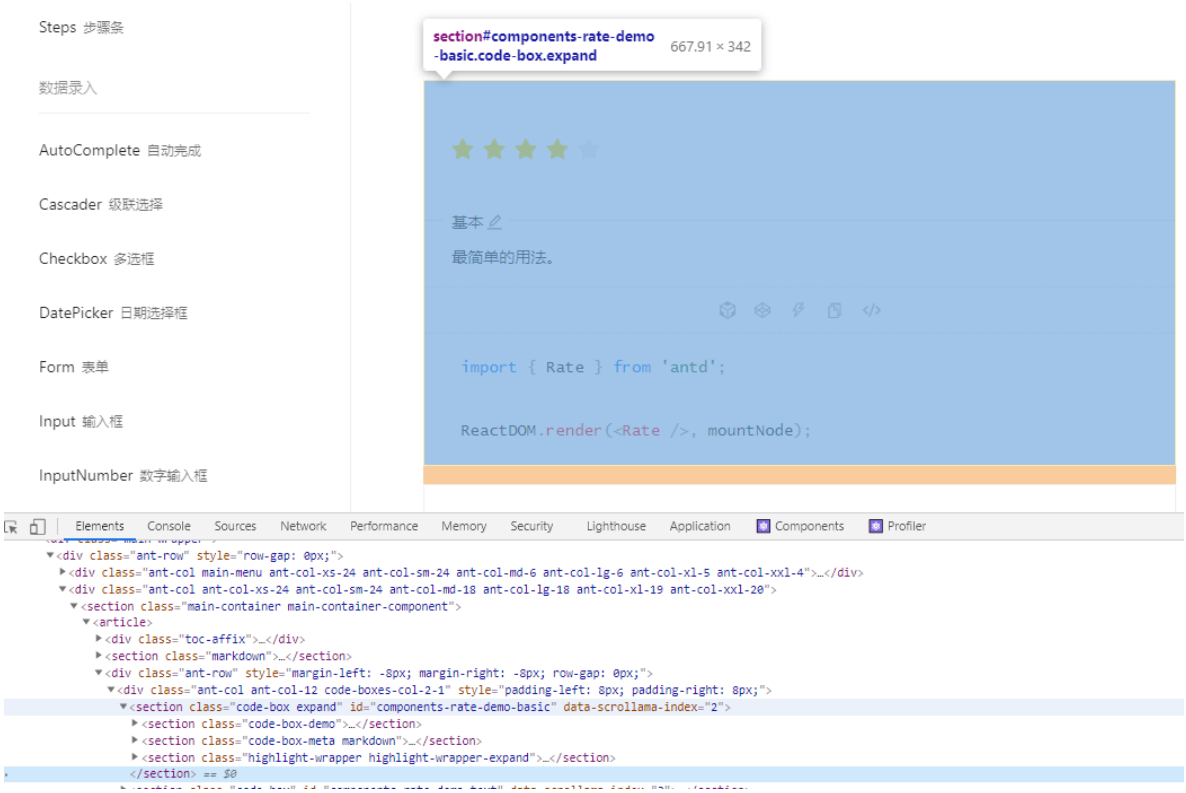
为什么不用storybook或者dumi，vuepress也行啊，两个原因：

第一，当时vue3处于rc阶段，我们的组件库是用vue3编写的，vue3更换了编译器，以上开源框架当时没更新。

第二，当时没听说过。

下面就讲一下这个演示模块是怎么开发出来的。

演示文档的构成



这是ant-design的文档截图，可以看到这个蓝色阴影的模块被很清晰地分成了三个section，分别对应的是：

1. 组件演示demo
2. 操作栏
3. 源码

首先明确一点，文档是通过markdown写出来的，markdown写文档的最大的好处就在于它有良好的词法分析支持，有很好的延展性。

组件的演示文档模块就是利用了markdown的这个特性，给markdown新增了一个自定义标识，写出我们自己解析这个标识的解析方法，解释输出成一个vue文件字符串后，再交由vue-loader解析编译生成我们熟悉的js+css，即可实现上图中在markdown文件中写两行代码渲染一整个模块的效果。

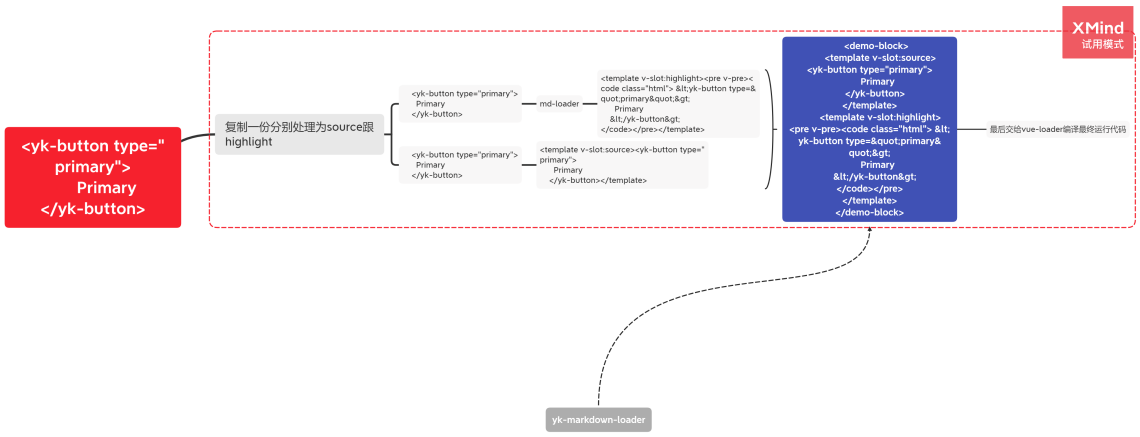
> 注：到目前为止，我一直在说演示文档模块，实际上它不是一个模块，而是一个组件，以这个ant-design的评分组件为例，这个组件负责将源码`import { Rate } from 'antd'; ReactDOM.render(<Rate />, mountNode);`渲染成两部分，一个是源码字符串，另一部分则是直接交由框架渲染器（vue/react)渲染。

‘组件模块’编写

```
<template>
  <div class="demo-container">
    <div>
      <div class="wrap">
        <div class="source">
          <slot name="source" />
        </div>
        <div v-show="toggleInfo.show" class="highlight">
          <slot name="highlight" />
        </div>
        <div class="codebox" @click="toggle">{{ toggleInfo.text }}代码</div>
      </div>
    </div>
```

```
  </div>
</template>
```

这就是我们的简化版模块结构，可以看到，class为source的部分是展示组件效果的，这一部分我们要直接交给vue-loader解析，class为codebox的部分是展示源码的，我们要先经有md-loader解析成html代码片段再展示。流程如图所示。



yk-md-loader编写

准备工作

先在打包配置中插入一段

```
module: {
  rules: [
    ...
    {
      test: /\.(en-US.md|zh-CN.md)$/,
      use: [
        'vue-loader',
        {
          loader: path.resolve(__dirname, './origin-md-loader/core.js'),
          options: { raw: true, preventExtract: false },
        },
      ],
    },
    ...
  ]
}
```

webpack中rules里面可以配置解析代码的方式，这段意思是遇到文件名中包含en-US.md或者zh-CN.md按照origin-md-loader=>vue-loader的顺序解析编译代码。

core.js

然后新建一个core.js文件，引入我们的核心插件，markdown-it，然后利用markdown-it提供的词法分析接口，新增对于`:::demo`、`:::`的自定义容器识别，例如：

md源码：

````
:::demo 对于字符型的头像，当字符串较长时，字体大小可以根据头像宽度自动调整。
```html
 <yk-button type="primary">
      Primary
    </yk-button>
```
:::
````

词法分析：

```
// Tokens
[
  Token {
    type: 'container_demo_open',
    tag: 'div',
    attrs: null,
    map: [ 0, 6 ],
    nesting: 1,
    level: 0,
    children: null,
    content: '',
    markup: ':::',
    info: 'demo 对于字符型的头像，当字符串较长时，字体大小可以根据头像宽度自动调整。',
    meta: null,
    block: true,
    tag: 'div',
    attrs: null,
    map: [ 0, 6 ],
    nesting: 1,
    level: 0,
    children: null,
    content: '',
    markup: ':::',
    info: 'demo 对于字符型的头像，当字符串较长时，字体大小可以根据头像宽度自动调整。',
    meta: null,
    block: true,
    hidden: false
  },
  Token {
    type: 'fence',
    tag: 'code',
    attrs: null,
    map: [ 1, 6 ],
    nesting: 0,
    level: 1,
    children: null,
    content: ' <yk-button type="primary">\n      Primary\n    </yk-button>\n',
    markup: '```',
    info: 'html',
    meta: null,
    block: true,
    hidden: false
  },
  Token {
    type: 'container_demo_close',
```

```
    tag: 'div',
    attrs: null,
    map: null,
    nesting: -1,
    level: 0,
    children: null,
    content: '',
    markup: ':::',
    info: '',
    meta: null,
    block: true,
    hidden: false
  }
]
```

对于这段词法分析结果，我们可以看到，对于md中的自定义容器:::语法的支持，所以我们只需要找到info中带有demo即可分析出哪段代码是包含在自定义容器里面的，再取其中的fence代码，即可拿到我们想要的代码片段。然后按照流程解析

```
//core.js
    let markdown = require('markdown-it');
    const mdContainer = require('markdown-it-container');
    ...
    markdown(opts.preset, opts)
    .use(mdContainer, 'demo', {
        validate(params) {
            return params.trim().match(/^demo\s*(.*)$/);
        },
        render(tokens, idx) {
            const m = tokens[idx].info.trim().match(/^demo\s*(.*)$/);
            if (tokens[idx].nesting === 1) {
            const description = m && m.length > 1 ? m[1] : '';
            const content = tokens[idx + 1].type === 'fence' ? tokens[idx +
1].content : '';
            return `<demo-block>
            ${
            description
                ? `<template v-slot:description><div>${md.render(description)}
</div></template>`
                : ''
            }
            <!--yk-design-demo: ${content}:yk-design-demo-->
            `;
        }
      return '</demo-block>';
    },
    });
    .use(md => {
      const defaultRender = md.renderer.rules.fence;
      md.renderer.rules.fence = (tokens, idx, options, env, self) => {
        console.log('tag', tokens)
        const token = tokens[idx];
        // 判断该 fence 是否在 :::demo 内
        const prevToken = tokens[idx - 1]
        const isInDemoContainer =
          prevToken && prevToken.nesting === 1 &&
prevToken.info.trim().match(/^demo\s*(.*)$/);
```

```
        if (token.info.trim() === 'html' && isInDemoContainer) {
          return `<template v-slot:highlight><pre v-pre><code
class="html">${md.utils.escapeHtml(
            token.content,
          )}</code></pre></template>`;
        }
        return defaultRender(tokens, idx, options, env, self);
      };
    });
    ...
    let content = parser.render(source);
```

content内容：

```
<demo-block>
        <template v-slot:description><div><p>对于字符型的头像，当字符串较长时，字体大小
可以根据头像宽度自动调整。</p>
</div></template>
        <!--yk-design-demo:  <yk-button type="primary">
      Primary
    </yk-button>
:yk-design-demo-->
        <template v-slot:highlight><pre v-pre><code class="html"> &lt;yk-button
type=&quot;primary&quot;&gt;
      Primary
    &lt;/yk-button&gt;
</code></pre></template></demo-block>
```

这样我们就完成了第一步，我们的content内容就是一个我们想要的东西，是一段vue的模板文件，但这边我添加了一对标识，`<!--yk-design-demo:`，`:yk-design-demo-->`。

为什么要这样做呢？如果只是做到这样输出，是远远达不到使用要求的。

考虑下面这种场景，如果你在md中这样写了一段文本：

````
:::demo 对于字符型的头像，当字符串较长时，字体大小可以根据头像宽度自动调整。
```html
 <yk-button type="primary" @click="onClick">
      Primary
    </yk-button>
  <script>
export default {
  methods: {
    onClick() {
      console.log('time', Date.now());
    },
  },
}
</script>
```
:::
````

按照我们的词法解析规则，content就会被解析成如下内容：

```
<demo-block>
        <template v-slot:description><div><p>对于字符型的头像，当字符串较长时，字体大小可以根据头像宽度自动调整。</p>
</div></template>
        <!--yk-design-demo:  <yk-button type="primary">
      Primary
    </yk-button>
      <script>
export default {
  methods: {
    onClick() {
      console.log('time', Date.now());
    },
  },
}
</script>

:yk-design-demo-->
        <template v-slot:highlight><pre v-pre><code class="html"> &lt;yk-button type=&quot;primary&quot;&gt;
      Primary
    &lt;/yk-button&gt;
</code></pre></template></demo-block>
```

很明显，这样的模板文件是不被vue语法所接受的，所以，我从设定的标识入手，开始做script标签的解析，让我们的md文件也可以正常的用vue语法写js。

解析script标签

这个很简单，直接解析content内容，截取一对script标签即可：

```
  //core.js
  const { stripScript, stripTemplate, genInlineComponentText } =
require('./utils');
  ...
  const startTag = '<!--yk-design-demo:';
  const startTagLen = startTag.length;
  const endTag = ':yk-design-demo-->';
  const endTagLen = endTag.length;

  let componenetsString = '';
  let importStr = '';
  let id = 0; // demo 的 id
  let output = []; // 输出的内容
  let start = 0; // 字符串开始位置

  let commentStart = content.indexOf(startTag);
  let commentEnd = content.indexOf(endTag, commentStart + startTagLen);
  while (commentStart !== -1 && commentEnd !== -1) {
    output.push(content.slice(start, commentStart));

    const commentContent = content.slice(commentStart + startTagLen,
commentEnd);
    const html = stripTemplate(commentContent);
    const script = stripScript(commentContent);
    let demoComponentContent = contentArr[1];
```

```
    const demoComponentName = `yk-design-demo${id}`;
    output.push(`<template v-slot:source><${demoComponentName} /></template>`);
    componenetsString += `${JSON.stringify(demoComponentName)}:
${demoComponentContent},`;

    // 重新计算下一次的位置
    id++;
    start = commentEnd + endTagLen;
    commentStart = content.indexOf(startTag, start);
    commentEnd = content.indexOf(endTag, commentStart + startTagLen);
  }
```

```
// utils
function stripScript(content) {
  const result = content.match(/<(script)>([\s\S]+)<\/\1>/);
  return result && result[2] ? result[2].trim() : '';
}

function stripStyle(content) {
  const result = content.match(/<(style)\s*>([\s\S]+)<\/\1>/);
  return result && result[2] ? result[2].trim() : '';
}

// 编写例子时不一定有 template。所以采取的方案是剔除其他的内容
function stripTemplate(content) {
  content = content.trim();
  if (!content) {
    return content;
  }
  return content.replace(/<(script|style)[\s\S]+<\/\1>/g, '').trim();
}
```

这样，我们直接将script标签截取并放到了content末，那么我们输出的就是这样一个内容：

```
<template>
<demo-block>
        <template v-slot:description><div><p>对于字符型的头像，当字符串较长时，字体大小
可以根据头像宽度自动调整。</p>
</div></template>
      <yk-button type="primary">
      Primary
    </yk-button>
        <template v-slot:highlight><pre v-pre><code class="html"> &lt;yk-button
type=&quot;primary&quot;&gt;
      Primary
    &lt;/yk-button&gt;
</code></pre></template></demo-block>
</template>
      <script>
export default {
  methods: {
    onClick() {
      console.log('time', Date.now());
    },
  },
```

```
}
</script>
```

直接返回并交给下一阶段，即vue-loader解析即可，至此，看起来我们的md-loader已经圆满完成了，可以直接将一个md文件解析编译输出为一个vue文件，再交由vue-loader编译输出。但仔细想想，就会发现这样的loader还远远谈不上可用，真正要做出一个方便可用的loader，还有一些情况要考虑，再次试想这样一个例子，这是在生产环境中DatePicker.zh-CN.md中的截取的一小段：

````
### 定制日期单元格
:::demo 使用 dateRender 可以自定义日期单元格的内容和样式。
```html
  <div>
    <a-date-picker>
      <template  v-slot:dateRender="{current, today}">
        <div class="yeahka-calendar-date" :style="getCurrentStyle(current,
today)">
          {{ current.date() }}
        </div>
      </template>
    </a-date-picker>
    <a-range-picker>
      <template  v-slot:dateRender="{current}">
        <div class="yeahka-calendar-date" :style="getCurrentStyle(current)">
          {{ current.date() }}
        </div>
      </template>
    </a-range-picker>
    <a-week-picker>
      <template  v-slot:dateRender="{current}">
        <div class="yeahka-calendar-date" :style="getCurrentStyle(current)">
          {{ current.date() }}
        </div>
      </template>
    </a-week-picker>
  </div>
<script>
export default {
  methods: {
    getCurrentStyle(current, today) {
      const style = {};
      if (current.date() === 1) {
        style.border = '1px solid #1890ff';
        style.borderRadius = '50%';
      }
      return style;
    },
  },
}
</script>

```
:::

### 不可选择日期和时间
:::demo 可用 disabledDate 和 disabledTime 分别禁止选择部分日期和时间，其中
disabledTime 需要和 showTime 一起使用。
````

```html
  <div>
    <a-date-picker
      format="YYYY-MM-DD HH:mm:ss"
      :disabled-date="disabledDate"
      :disabled-time="disabledDateTime"
      :show-time="{ defaultValue: moment('00:00:00', 'HH:mm:ss') }"
    />
    <br />
    <a-month-picker :disabled-date="disabledDate" placeholder="Select month" />
    <br />
    <a-range-picker
      :disabled-date="disabledDate"
      :disabled-time="disabledRangeTime"
      :show-time="{
        hideDisabledOptions: true,
        defaultValue: [moment('00:00:00', 'HH:mm:ss'), moment('11:59:59',
'HH:mm:ss')],
      }"
      format="YYYY-MM-DD HH:mm:ss"
    />
  </div>
<script>
import moment from 'moment';
export default {
  methods: {
    moment,
    range(start, end) {
      const result = [];
      for (let i = start; i < end; i++) {
        result.push(i);
      }
      return result;
    },

    disabledDate(current) {
      // Can not select days before today and today
      return current && current < moment().endOf('day');
    },

    disabledDateTime() {
      return {
        disabledHours: () => this.range(0, 24).splice(4, 20),
        disabledMinutes: () => this.range(30, 60),
        disabledSeconds: () => [55, 56],
      };
    },

    disabledRangeTime(_, type) {
      if (type === 'start') {
        return {
          disabledHours: () => this.range(0, 60).splice(4, 20),
          disabledMinutes: () => this.range(30, 60),
          disabledSeconds: () => [55, 56],
        };
      }
      return {
        disabledHours: () => this.range(0, 60).splice(20, 4),
```

```
        disabledMinutes: () => this.range(0, 31),
        disabledSeconds: () => [55, 56],
      };
    },
  },
}
</script>
```
:::

看看会有什么问题?