面试官: 写过『通用前端组件』吗?

前言

设计前端组件是最能考验开发者基本功的测试之一,因为调用Material design、Antd、iView 等现成组件库的 API 每个人都可以做到,但是很多人并不知道很多常用组件的设计原理。

能否设计出通用前端组件也是区分前端工程师和前端api调用师的标准之一,那么应该如何设计出一个通用组件呢?

> 下文中提到的**组件库**通常是指单个组件,而非集合的概念,集合概念的组件库是 Antd iView这种,我们所说的组件库是指集合中的单个组件,集合性质的组件库需要考虑的要更多.

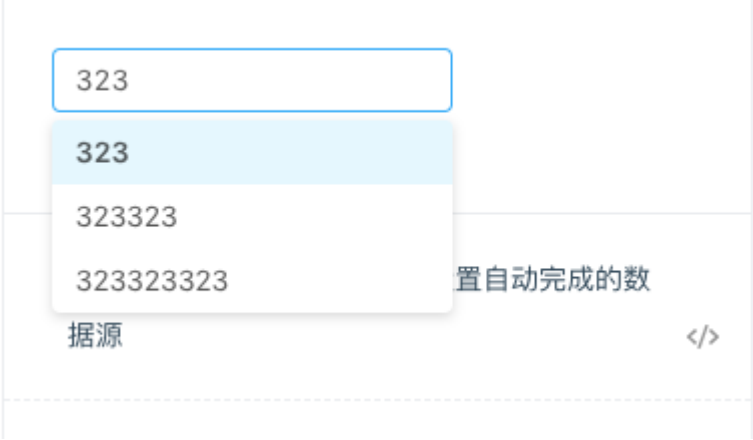
文章目录

1.前端组件库的设计原则

1.1 细粒度的考量

我们在学习设计模式的时候会遇到很多种设计原则,其中一个设计原则就是**单一职责原则**,在组件库的开发中同样适用,我们原则上一个组件只专注一件事情,单一职责的组件的好处很明显,由于职责单一就可以最大可能性地复用组件,但是这也带来一个问题,过度单一职责的组件也可能会导致过度抽象,造成组件库的碎片化。

举个例子，一个自动完成组件(AutoComplete),他其实是由 Input 组件和 Select 组件组合而成的,因此我们完全可以复用之前的相关组件,就比如 Antd 的AutoComplete组件中就复用了Select组件,同时 Calendar、 Form 等等一系列组件都复用了 Select 组件,那么Select 的细粒度就是合适的,因为 Select 保持的这种细粒度很容易被复用.



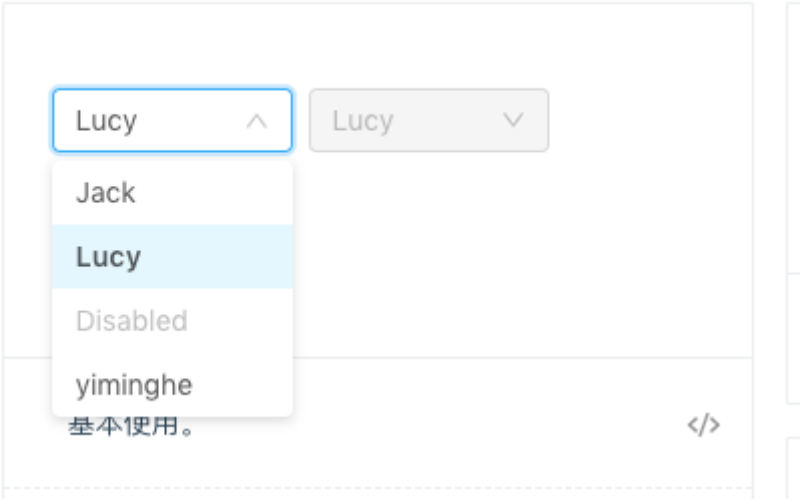
那么还有一个例子,一个徽章数组件(Badge),它的右上角会有红点提示,可能是数字也可能是 icon,他的职责当然也很单一，这个红点提示也理所当然也可以被单独抽象为一个独立组件,但是我们通常不会将他作为独立组件,因为在其他场景中这个组件是无法被复用的，因为没有类似的场景再需要小红点这个小组件了，所以作为独立组件就属于细粒度过小,因此我们往往将它作为 Badge 的内部组件,比如在 Antd 中它以ScrollNumber的名称作为Badge的内部组件存在。

所以，所谓的单一职责组件要建立在可复用的基础上，对于不可复用的单一职责组件我们仅仅作为独立组件的内部组件即可。

1.2 通用性考量

我们要设计的本身就是通用组件库,不同于我们常见的业务组件,通用组件是与业务解耦但是又服务于业务开发的,那么问题来了,如何保证组件的通用性,通用性高一定是好事吗?

比如我们设计一个选择器(Select)组件,通常我们会设计成这样



这是一个我们最常见也最常用的选择器,但是问题是其通用性大打折扣

当我们有一个需求是长这样的时候,我们之前的选择器组件就不符合要求了,因为这个 Select 组件的最下部需要有一个可拓展的条目的按钮

这个时候我们难道要重新修改之前的选择器组件,甚至再造一个符合要求的选择器组件吗?一旦有这种情况发生,那么只能说明之前的选择器组件通用性不够,需要我们重新设计.

Antd 的 Select 组件预留了 `dropdownRender` 来进行自定义渲染,其依赖的 `rc-select` 组件中的代码如下

> Antd 依赖了大量以 `rc-` 开头的底层组件,这些组件被react-component团队(同时也就是Antd 团队)维护,其主要实现组件的底层逻辑,Antd 则是在此基础上添加Ant Design设计语言而实现的

当然类似的设计还有很多,通用性设计其实是一定意义上放弃对 DOM 的掌控,而将 DOM 结构的决定权转移给开发者, `dropdownRender` 其实就是放弃对 Select 下拉菜单中条目的掌控,Antd 的 Select 组件其实还有一个没有在文档中体现的方法 `getInputElement` 应该是对 Input 组件的自定义方法,Antd整个 Select 的组件设计非常复杂,基本将所有的 DOM 结构控制权全部暴露给了开发者,其本身只负责底层逻辑和最基本的 DOM 结构.

这是 Antd 所依赖的 re-select 最终 jsx 的结构,其 DOM 结构很简单,但是暴露了大量自定义渲染的接口给开发者.

```
复制代码
return (      <SelectTrigger        onPopupFocus={this.onPopupFocus}
onMouseEnter={this.props.onMouseEnter}        onMouseLeave=
{this.props.onMouseLeave}        dropdownAlign={props.dropdownAlign}
dropdownClassName={props.dropdownClassName}        dropdownMatchSelectWidth=
{props.dropdownMatchSelectWidth}        defaultActiveFirstOption=
{props.defaultActiveFirstOption}        dropdownMenuStyle=
{props.dropdownMenuStyle}        transitionName={props.transitionName}
animation={props.animation}        prefixCls={props.prefixCls}
dropdownStyle={props.dropdownStyle}        combobox={props.combobox}
showSearch={props.showSearch}        options={options}        multiple=
{multiple}        disabled={disabled}        visible={realOpen}
inputValue={state.inputValue}        value={state.value}        backfillValue=
{state.backfillValue}        firstActiveValue={props.firstActiveValue}
onDropdownVisibleChange={this.onDropdownVisibleChange}
getPopupContainer={props.getPopupContainer}        onMenuSelect=
{this.onMenuSelect}        onMenuDeselect={this.onMenuDeselect}
onPopupScroll={props.onPopupScroll}        showAction={props.showAction}
 ref={this.saveSelectTriggerRef}        menuItemSelectedIcon=
{props.menuItemSelectedIcon}        dropdownRender={props.dropdownRender}
  ariaId={this.ariaId}      >        <div          id={props.id}
style={props.style}          ref={this.saveRootRef}          onBlur=
{this.onOuterBlur}          onFocus={this.onOuterFocus}          className=
{classnames(rootCls)}          onMouseDown={this.markMouseDown}
onMouseUp={this.markMouseLeave}          onMouseOut={this.markMouseLeave}
  >          <div            ref={this.saveSelectionRef}
key="selection"            className={`${prefixCls}-selection
${prefixCls}-selection--${multiple ? 'multiple' : 'single'}`}
role="combobox"            aria-autocomplete="list"            aria-
haspopup="true"            aria-controls={this.ariaId}            aria-
expanded={realOpen}            {...extraSelectionProps}          >
{ctrlNode}            {this.renderClear()}
{this.renderArrow(!!multiple)}          </div        </div
</SelectTrigger    );
```

那么这么多需要自定义的地方,这个 Select 组件岂不是很难用?因为好像所有地方都需要开发者自定义,通用性设计在将 DOM 结构决定权交给开发者的同时也保留了默认结构,在开发者没有显示自定义的时候默认使用默认渲染结构,其实 Select 的基本使用很方便,如下:

```
复制代码
    <Select defaultValue="lucy" style={{ width: 120 }} disabled>      <Option
value="lucy">Lucy</Option>    </Select>
```

> 组件的形态(DOM结构)永远是千变万化的,但是其行为(逻辑)是固定的,因此通用组件的秘诀之一就是将 DOM 结构的控制权交给开发者,组件只负责行为和最基本的 DOM 结构

---

2 技术选型

2.1 css 解决方案

由于CSS 本身的众多缺陷，如书写繁琐（不支持嵌套）、样式易冲突（没有作用域概念）、缺少变量（不便于一键换主题）等不一而足。为了解决这些问题，社区里的解决方案也是出了一茬又一茬，从最早的 CSS prepocessor（SASS、LESS、Stylus）到后来的后起之秀 PostCSS，再到 CSS Modules、Styled-Components 等。

Antd 选择了 less 作为 css 的预处理方案,Bootstrap 选择了 Scss,这两种方案孰优孰劣已经争论了很多年了:

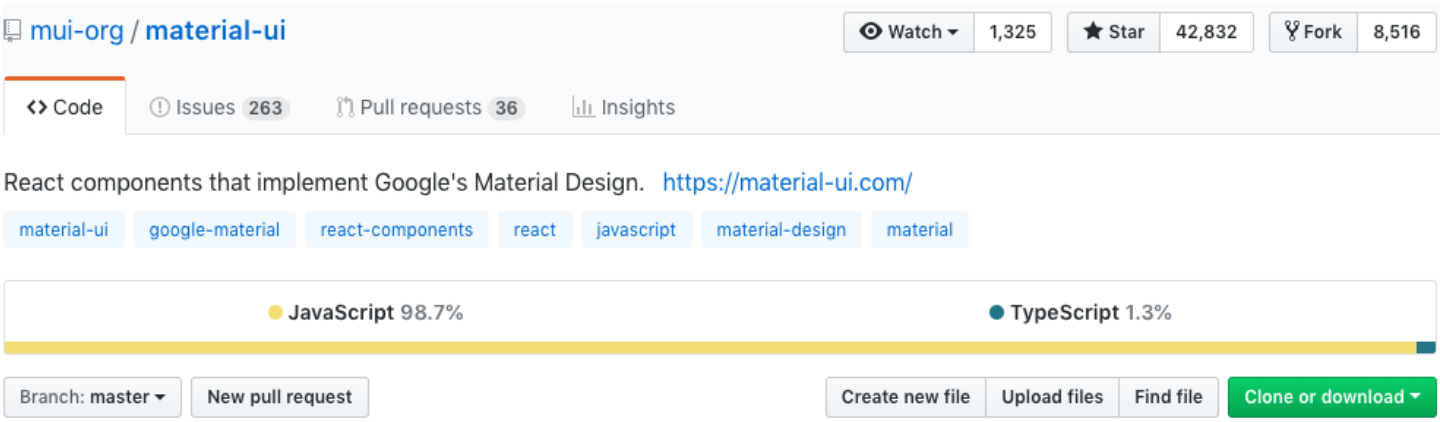
SCSS和LESS相比有什么优势?

但是不管是哪种方案都有一个很烦人的点,就是需要额外引入 css,比如 Antd 需要这样显示引入:

```
1 复制代码
2 import Button from 'antd/lib/button';import 'antd/lib/button/style';
```

为了解决这种尴尬的情况,Antd 用 Babel 插件将这种情况 Hack 掉了

而 `material-ui` 并不存在这种情况,他不需要显示引入 css,这个最流行的 React 前端组件库里面只有 js 和 ts 两种代码,并不存在 css 相关的代码,为什么呢?



他们用 `jss` 作为css-in-js 的解决方案,jsx 的引入已经将 js 和 html 耦合,css-in-js将 css 也耦合进去,此时组件便不需要显示引入 css,而是直接引用 js 即可.

这不是退化到史前前端那种写内联样式的时代了吗?

并不是,史前前端的内联样式是整个项目耦合的状态,当然要被抛弃到历史的垃圾堆中,后来的样式和逻辑分离,实际上是以页面为维度将 js css html 解耦的过程,如今的时代是组件化的时代了,jsx 已经将 js 和 html 框定到一个组件中,css 依然处于分离状态,这就导致了每次引用组件却还需要显示引入 css,css-in-js 正式彻底组件化的解决方案.

当然,我个人目前在用 styled-components,其优点引用如下:

1. 首先，styled-components 所有语法都是标准 css 语法，同时支持 scss 嵌套等常用语法，覆盖了所有 css 场景。
2. 在样式复写场景下，styled-components 支持在任何地方注入全局 css，就像写普通 css 一样
3. styled-components 支持自定义 className，两种方式，一种是用 babel 插件, 另一种方式是使用 styled.div.withConfig({ componentId: "prefix-button-container" }) 相当于添加 className="prefix-button-container"
4. className 语义化更轻松，这也是 class 起名的初衷
5. 更适合组件库使用，直接引用 import "module" 即可，否则你有三条路可以走：像 antd 一样，单独引用 css，你需要给 node_modules 添加 css-loader；组件内部直接 import css 文件，如果任何业务项目没有 css-loader 就会报错；组件使用 scss 引用，所有业务项目都要配置一份 scss-loader 给 node_modules；这三种对组件库来说，都没有直接引用来的友好
6. 当你写一套组件库，需要单独发包，又有统一样式的配置文件需求，如果这个配置文件是 js 的，所有组件直接引用，对外完全不用关注。否则，如果是 scss 配置文件，摆在面前还是三条路：每个组件单独引用 scss 文件，需要每个业务项目给 node_modules 添加 scss-loader（如果业务用了 less，还要装一份 scss 是不）；或者业务方只要使用了你的组件库，就要在入口文件引用你的 scss 文件，比如你的组件叫 button，scss 可能叫 common-css，别人听都没听过，还要查文档；或者业务方在 webpack 配置中单独引用你的 common-css，这也不科学，如果用了3个组件库，天天改 webpack 配置也很不方便。
7. 当 css 设置了一半样式，另一半真的需要 js 动态传入，你不得不 css + css-in-js 混合使用，项目久了，维护的时候发现某些 css-in-js 不变了，可以固化在 css 里，css 里固定的值又因为新去求变得可变了，你又得拿出来放在 css-in-js 里，实践过就知道有多么烦心。

2.2 js 解决方案

选 Typescript ,因为巨硬大法好...

可以看看知乎问题下我的回答你为什么不用 Typescript

或者看此文TypeScript体系调研报告

---

8. **如何快速启动一个组件库项目**

组件的具体实现部分当然是组件库的核心,但是在现代前端库中其他部分也必不可少,我们需要一堆工具来辅助我们开发,例如编译工具、代码检测工具、打包工具等等。

3.1 打包工具(rollup vs webpack)

市面上打包工具数不胜数,最火爆的当然是需要*配置工程师*专门配置的webpack,但是在类库开发领域它有一个强大的对手就是 rollup。

现代市面上主流的库基本都选择了 rollup 作为打包工具，包括Angular React 和 Vue, 作为基础类库的打包工具 rollup 的优势如下:

- Tree Shaking: 自动移除未使用的代码, 输出更小的文件
- Scope Hoisting: 所有模块构建在一个函数内, 执行效率更高
- Config 文件支持通过 ESM 模块格式书写 可以一次输出多种格式:
- 模块规范: IIFE, AMD, CJS, UMD, ESM Development 与 production 版本: .js, .min.js

虽然上面部分功能已经被 webpack 实现了,但是 rollup 明显引入得更早,而Scope Hoisting更是杀手锏,由于 webpack 不得不在打包代码中构建模块系统来适应 app 开发(模块系统对于单一类库用处很小),Scope Hoisting将模块构建在一个函数内的做法更适合类库的打包.

3.2 代码检测

由于 JavaScript 各种诡异的特性和大型前端项目的出现,代码检测工具已经是前端开发者的标配了,Douglas Crockford最早于2002创造出了 JSLint,但是其无法拓展,具有极强的Douglas Crockford个人色彩,Anton Kovalyov由于无法忍受 JSLint 无法拓展的行为在2011年发布了可拓展的JSHint,一时之间JSHint成为了前端代码检测的流行解决方案.

随后的2013年,Nicholas C. Zakas鉴于JSHint拓展的灵活度不够的问题开发了全新的基于 AST 的 Lint 工具 ESLint,并随着 ES6的流行统治了前端界,ESLint 基于Esprima进行 JavaScript 解析的特性极易拓展,JSHint 在很长一段时间无法支持 ES6语法导致被 ESLint 超越.

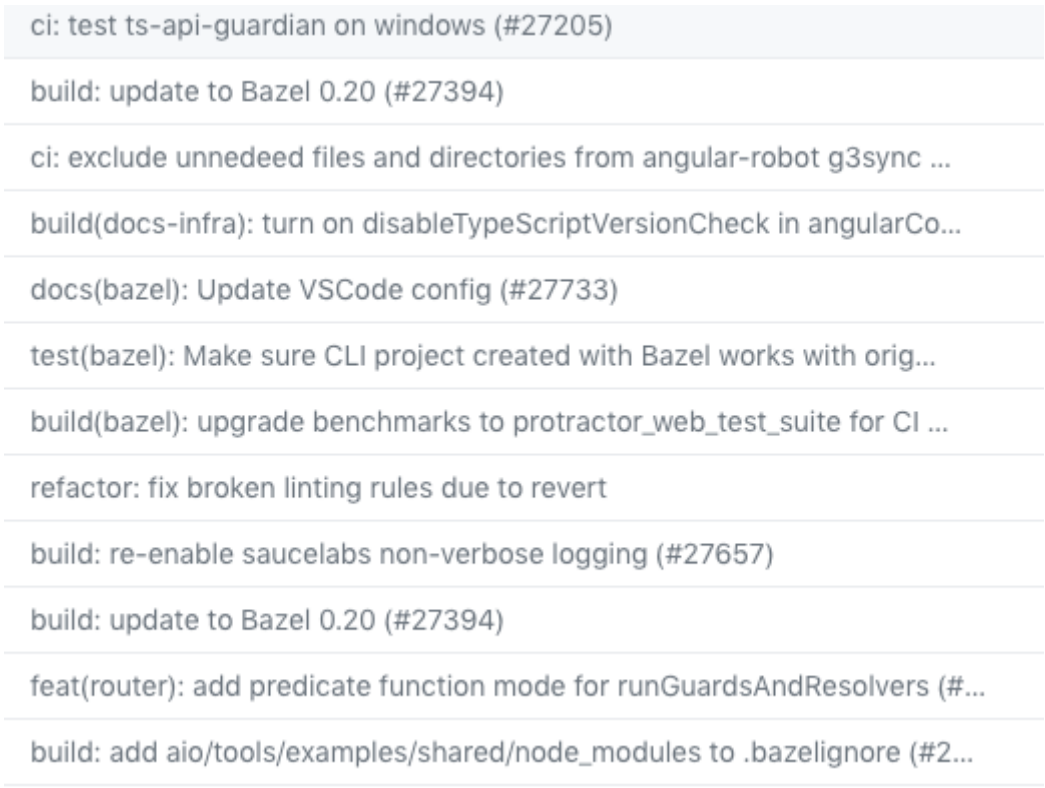
但是在 Typescript 领域 ESLint 却处于弱势地位,TSLint 的出现要比 ESLint 正式支持 Typescript 早很多,~~目前 TSLint 似乎是 TS 的事实上的代码检测工具~~.

> 注: 文章成文较早,我也没想到前阵子 TS 官方钦点了 ESLint,TSLint 失宠了,面向未来的官方标配的代码检测工具肯定是 ESLint 了,但是 TSLint 目前依然被大量使用,现在仍然可以放心使用

代码检测工具是一方面,代码检测风格也需要我们做选择,市面上最流行的代码检测风格应该是 Airbnb 出品的 `eslint-config-airbnb` ,其最大的特点就是极其严格,没有给开发者任何选择的余地,当然在大型前端项目的开发中这种严格的代码风格是有利于协作的,但是作为一个类库的代码检测工具而言并不适合,所以我们选择了 `eslint-config-standard` 这种相对更为宽松的代码检测风格.

3.3 commit 规范

以下两种 commit 哪个更严谨且易于维护?



最开始使用 commit 的时候我也经常犯下图的错误,直到看到很多明星类库的 commit 才意识到自己的错误,写好 commit message 不仅有助于他人 review, 还可以有效的输出 CHANGELOG, 对项目的管理实际至关重要.

目前流行的方案是 Angular 团队的规范,其关于 head 的大致规范如下:

- type: commit 的类型
- feat: 新特性
- fix: 修改问题
- refactor: 代码重构
- docs: 文档修改
- style: 代码格式修改, 注意不是 css 修改
- test: 测试用例修改
- chore: 其他修改, 比如构建流程, 依赖管理.
- scope: commit 影响的范围, 比如: route, component, utils, build...
- subject: commit 的概述, 建议符合 50/72 formatting
- body: commit 具体修改内容, 可以分为多行, 建议符合 50/72 formatting
- footer: 一些备注, 通常是 BREAKING CHANGE 或修复的 bug 的链接.

当然规范人们不一定会遵守,我最初知道此类规范的时候也并没有严格遵循,因为人总会偷懒,直到用 `commitizen` 将此规范集成到工具流中,每个 commit 就不得不遵循规范了.

我具体参考了这篇文章: 优雅的提交你的 Git Commit Message

3.4 测试工具

业务开发中由于前端需求变动频繁的特性,导致前端对测试的要求并没有后端那么高,后端业务逻辑一旦定型变动很少,比较适合测试.

但是基础类库作为被反复依赖的模块和较为稳定的需求是必须做测试的,前端测试库也可谓是种类繁多了,经过比对之后我还是选择了目前最流行也是被三大框架同时选择了的 Jest 作为测试工具,其优点很明显:

1. 开箱即用,内置断言、测试覆盖率工具,如果你用 MoCha 那可得自己手动配置 n 多了
2. 快照功能,Jest 可以利用其特有的快照测试功能，通过比对 UI 代码生成的快照文件
3. 速度优势,Jest 的测试用例是并行执行的，而且只执行发生改变的文件所对应的测试，提升了测试速度

3.5 其它

当然以上是主要工具的选择,还有一些比如:

- 代码美化工具 prettier,解放人肉美化,同时利于不同人协作的风格一致
- 持续集成工具 travis-ci,解放人肉测试 lint,利于保证每次 push 的可靠程度

3.6 快速启动脚手架

那么以上这么多配置难道要我们每次都自己写吗?组件的具体实现才是组件库的核心,我们为什么要花这么多时间在配置上面?

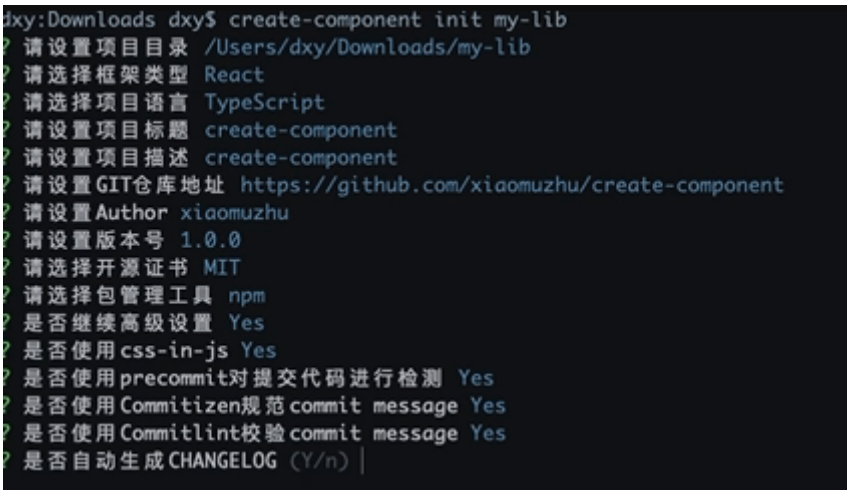
我们在建立 APP 项目时通常会用到框架官方提供的脚手架,比如 React 的 create-react-app,Angular 的 Angular-Cli 等等,那么能不能有一个专门用于组件开发的快速启动的脚手架呢?

有的,我最近开发了一款快速启动组件库开发的命令行工具--create-component

利用

```
1 复制代码
2 create-component init <name>
```

来快速启动项目,我们提供了丰富的可选配置,只要你做好技术选型后,根据提示去选择配置即可,create-component 会自动根据配置生成脚手架,其灵感就来源于 vue-cli和 Angular-cli.



---

4. 如何设计一个轮播图组件

说了很多理论,那么实战如何呢?设计一个通用组件试试吧!

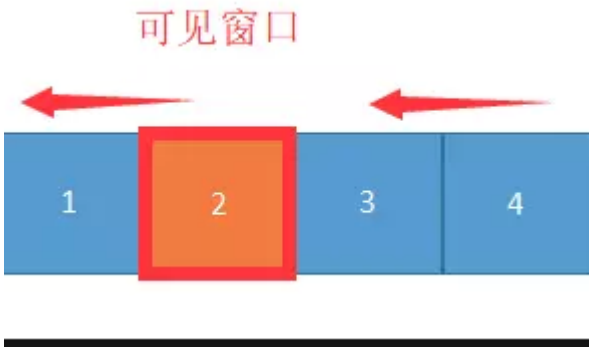
4.1 轮播图基本原理

轮播图(Carousel),在 Antd 中被称为走马灯,可能是前端开发者最常见的组件之一了,不管是在 PC 端还是在移动端我们总能见到他的身影.

那么我们通常是如何使用轮播图的呢?Antd 的代码如下

```
复制代码
  <Carousel    <div<h31</h3</div    <div<h32</h3</div    <div<h33</h3</div
<div<h34</h3</div  </Carousel
```

问题是我们在 `Carousel` 中放入了四组 `div` 为什么一次只显示一组呢?



图中被红框圈住的为可视区域,可视区域的位置是固定的,我们只需要移动后面 `div` 的位置就可以做到1 2 3 4四个子组件轮播的效果,那么子组件2目前在可视区域是可以被看到的,1 3 4应该被隐藏,这就需要我们设置overflow 属性为 hidden来隐藏非可视区域的子组件.

> 复制查看动图: images2015.cnblogs.com/blog/979044…

因此就比较明显了,我们设计一个可视窗口组件 `Frame` ,然后将四个 `div` 共同放入幻灯片组合组件 `SlideList` 中,并用 `SlideItem` 分别将 `div` 包裹起来,实际代码应该是这样的:

```
复制代码
 <Frame    <SlideList        <SlideItem            <div<h31</h3</div
</SlideItem        <SlideItem            <div<h32</h3</div          </SlideItem
        <SlideItem            <div<h33</h3</div          </SlideItem
<SlideItem            <div<h34</h3</div          </SlideItem    </SlideList
</Frame
```

我们不断利用 `translateX` 来改变 `SlideList` 的位置来达到轮播效果,如下图所示,每次轮播的触发都是通过改变 `transform: translateX()` 来操作的

4.2 轮播图基础实现

搞清楚基本原理那么实现起来相对容易了,我们以移动端的实现为例,来实现一个基础的移动端轮播图.

首先我们要确定**可视窗口**的宽度,因为我们需要这个宽度来计算出 `SlideList` 的长度(`SlideList` 的长度通常是可视窗口的倍数,比如要放三张图片,那么 `SlideList` 应该为可视窗口的至少3倍),不然我们无法通过 `translateX` 来移动它.

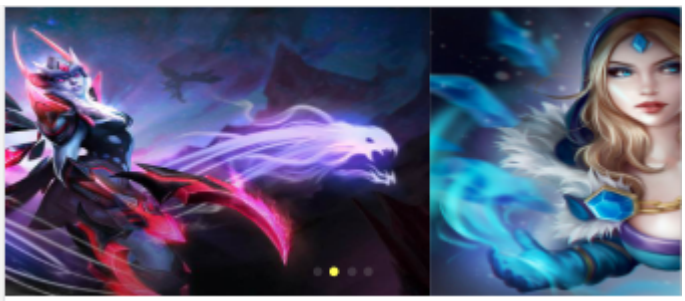
我们通过 `getBoundingClientRect` 来获取可视区域真实的长度, `SlideList` 的长度那么为:

`slideListWidth = (len + 2) * width` (len 为传入子组件的数量,width 为可视区域宽度)

至于为什么要 `+2` 后面会提到.

- 复制代码
  /**设置轮播区域尺寸@param x */ private setSize(x?: number) { const { width } = this.frameRef.current!.getBoundingClientRect() const len = React.Children.count(this.props.children) const total = len + 2 this.setState({ slideItemWidth: width, slideListWidth: total * width, total, translateX: -width * this.state.currentIndex, startPositionX: x !== undefined ? x : 0, }) }

获取到了总长度之后如何实现轮播呢?我们需要根据用户反馈来触发轮播,在移动端通常是通过手指滑动来触发轮播,这就需要三个事件 `onTouchStart` `onTouchMove` `onTouchEnd` .



`onTouchStart` 顾名思义是在手指触摸到屏幕时触发的事件,在这个事件里我们只需要记录下手指触摸屏幕的横轴坐标 x 即可,因为我们会通过其横向滑动的距离大小来判断是否触发轮播

- 复制代码
  /**处理触摸起始时的事件 *@private@param {React.TouchEvent} e@memberof Carousel */ private onTouchStart(e: React.TouchEvent) { clearInterval(this.autoPlayTimer) // 获取起始的横轴坐标 const { x } = getPosition(e) this.setSize(x) this.setState({ startPositionX: x, }) }

`onTouchMove` 顾名思义是处于滑动状态下的事件,此事件在 `onTouchStart` 触发后, `onTouchEnd` 触发前,在这个事件中我们主要做两件事,一件事是判断滑动方向,因为用户可能向左或者向右滑动,另一件事是让轮播图跟随手指移动,这是必要的用户反馈.

- 复制代码
  /**当触摸滑动时处理事件 *@private@param {React.TouchEvent} e@memberof Carousel */ private onTouchMove(e: React.TouchEvent) { const { slideItemWidth, currentIndex, startPositionX } = this.state const { x } = getPosition(e) const deltaX = x - startPositionX // 判断滑动方向 const direction = deltaX > 0 ? 'right' : 'left' this.setState({ direction, moveDeltaX: deltaX, // 改变translateX来达到轮播组件跟随手指移动的效果 translateX: -(slideItemWidth * currentIndex) + deltaX, }) }

`onTouchEnd` 顾名思义是滑动完毕时触发的事件,在此事件中我们主要做一个件事情,就是判断是否触发轮播,我们会设置一个阈值 `threshold` ,当滑动距离超过这个阈值时才会触发轮播,毕竟没有阈值的话用户稍微触碰轮播图就造成轮播,误操作会造成很差的用户体验.

- 复制代码
  /**滑动结束处理的事件 *@private@memberof Carousel */ private onTouchEnd() { this.autoPlay() const { moveDeltaX, slideItemWidth, direction } = this.state const threshold = slideItemWidth * THRESHOLD_PERCENTAGE // 判断是否轮播 const moveToNext = Math.abs(moveDeltaX) > threshold if (moveToNext) { // 如果轮播触发那么进行轮播操作 this.handleSwipe(direction!) } else { // 轮播不触发,那么轮播图回到原位 this.handleMisoperation() } }

4.3 轮播图的动画效果

我们常见的轮播图肯定不是生硬的切换,一般在轮播中会有一个渐变或者缓动的动画,这就需要我们加入动画效果.

我们制作动画通常有两个选择,一个是用 css3自带的动画效果,另一个是用浏览器提供的 requestAnimationFrame API

孰优孰劣?css3简单易用上手快,兼容性好, `requestAnimationFrame` 灵活性更高,能实现 css3实现不了的动画,比如众多缓动动画 css3都束手无策,因此我们毫无疑问地选择了 `requestAnimationFrame` .

> 双方对比请看张鑫旭大神的CSS3动画那么强，requestAnimationFrame还有毛线用?

想用 `requestAnimationFrame` 实现缓动效果就需要特定的缓动函数,下面就是典型的缓动函数

- 复制代码
  type tweenFunction = (t: number, b: number, _c: number, d: number) => numberconst easeInOutQuad: tweenFunction = (t, b, _c, d) => { const c = _c - b; if ((t /= d / 2) < 1) { return c / 2t * t + b; } else { return -c / 2((--t) * (t - 2) - 1) + b; }}

缓动函数接收四个参数,分别是:

- t: 时间
- b:初始位置
- _c:结束的位置

- d:速度

通过这个函数我们能算出每一帧轮播图所在的位置, 如下:

```
(30) [-375, -375.8333333333333, -378.3333333333333, -382.5, -388.3333333333333, -395.8333333333333, -405, -415.8333333333333, -428.3333333333333, -442.5, -458.3333333333337, -475.83333333333337, -495, -515.8333333333334, -538.3333333333334, -562.5000000000001, -586.6666666666666, -609.1666666666667, -630, -649.1666666666667, -666.6666666666667, -682.5, -696.6666666666667, -709.1666666666667, -720, -729.1666666666667, -736.6666666666667, -742.5, -746.6666666666666, -749.1666666666667]
  length: 0
  __proto__: Array(0)
```

在获取每一帧对应的位置后,我们需要用 `requestAnimationFrame` 不断递归调用依次移动位置,我们不断调用 `animation` 函数是其触发函数体内的 `this.setState({ translateX: tweenQueue[0], })` 来达到移动轮播图位置的目的,此时将这数组内的30个位置依次快速执行就是一个缓动动画效果.

- 复制代码

  /**递归调用,根据轨迹运动 *@private@param {number[]} tweenQueue@param {number} newIndex@memberof Carousel */ private animation(tweenQueue: number[], newIndex: number) { if (tweenQueue.length < 1) { this.handleOperationEnd(newIndex) return } this.setState({ translateX: tweenQueue[0], }) tweenQueue.shift() this.rafId = requestAnimationFrame(() => this.animation(tweenQueue, newIndex)) }

但是我们发现了一个问题,当我们移动轮播图到最后的时候,动画出现了问题,*当我们向左滑动最后一个轮播图 `div4` 时,这种情况下应该是图片向左滑动,然后第一张轮播图 `div1` 进入可视区域,但是反常的是图片快速向右滑动 `div1` 出现在可是区域...*

因为我们此时将位置4设置为了位置1,这样才能达到不断循环的目的,但是也造成了这个副作用,图片行为与用户行为产生了相悖的情况(用户向左划动,图片向右走).

目前业界的普遍做法是将图片首尾相连,例如图片1前面连接一个图片4,图片4后跟着一个图片1,这就是为什么之前计算长度时要 `+2`

`slideListWidth = (len + 2) * width` (len 为传入子组件的数量,width 为可视区域宽度)

当我们移动图片4时就不会出现上述向左滑图片却向右滑的情况,因为真实情况是:

`图片4 -- 滑动为 -> 伪图片1` 也就是位置 5 变成了位置 6

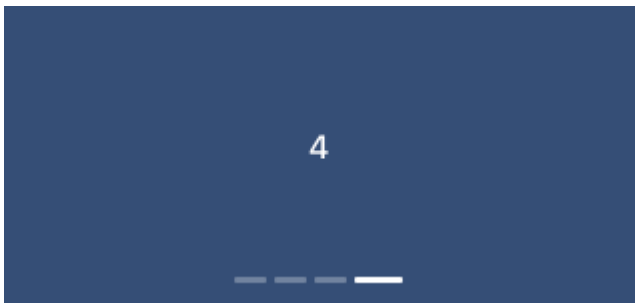
当动画结束之后,我们迅速把 `伪图片1` 的位置设置为 `真图片1` ,这其实是个障眼法,也就是说动画执行过程中实际上是 `图片4` 到 `伪图片1` 的过程,当结束后我们偷偷把 `伪图片1` 换成 `真图片1` ,因为两个图一模一样,所以这个转换的过程用户根本看不出来...

如此一来我们就可以实现无缝切换的轮播图了

4.4 改进方向

我们实现了轮播图的基本功能,但是其通用性依然存在缺陷:

1. 提示点的自定义: 我的实现是一个小点,而 antd 是用的条,这个地方完全可以将 dom 结构的决定权交给开发者.



1. 方向的自定义: 本轮播图只有水平方向的实现,其实也可以有纵向轮播

1. 多张轮播:除了单张轮播也可以多张轮播

以上都是可以对轮播图进行拓展的方向,相关的还有性能优化方面

我们的具体代码中有一个相关实现,我们的轮播图其实是有自动轮播功能的,但是很多时候页面并不在用户的可视页面中,我们可以根据是否页面被隐藏来取消定时器终止自动播放.

```
// 监听 document,如果处于隐藏状态,那么取消定时器
document.addEventListener('visibilitychange', () => {
  const isHidden = document.hidden
  if (isHidden) {
    clearInterval(this.autoPlayTimer)
  } else {
    this.autoPlay()
  }
})
```