

React组件库设计

1、课程目标

2、课程大纲

3、主要内容

3.1、react组件的设计原则

有意义

通用性

无状态，无副作用

避免过度封装

单一职责

易于测试

3.2、组件文档编写

文档结构

例子：

3.3、业界成熟的组件库脚手架

Storybook

Dumi

3.4、组件库架构差异及各自使用场景

Multirepo

优点

缺点

典型案例

Monorepo

优点

缺点

典型案例

Monorepo管理工具

3.5、引入代码规范和提交规范

Eslint&Prettier

[Typescript](#)

[commitizen & commitlint & husky](#)

3.6、组件库文档

[Docz](#)

3.7、构建工具的选择

[Rollup vs Webpack](#)

[Webpack](#)

[Rollup](#)

[@umi/father](#)

3.8、编写单元测试

[使用jest:](#)

[结构](#)

[\[target\].test.js 文件常见格式](#)

[使用test-library](#)

[获取元素](#)

[断言函数](#)

[事件触发/回调处理](#)

[react-hooks-testing-library](#)

[act](#)

[单元测试编写原则](#)

3.9、版本号管理

1、课程目标

P6:

- 掌握可复用的react组件设计思路和原则；
- 能够基于业界成熟方案搭建一个react组件库；
- 能够编写高质量的组件文档。

P6+ – P7:

- 能够不使用脚手架从0-1搭建一个react组件库；
- 能够理解不同组件库架构的差异和各自使用场景；
- 能够制定标准的代码规范和提交规范；

- 能够掌握组件单元测试的编写；
- 了解组件库发布流程。

2、课程大纲

- 编写可复用高质量的react组件；
- 编写高质量的组件文档；
- 业界成熟的组件库脚手架介绍；
- 常见的组件库架构差异解析；
- 引入代码规范和提交规范；
- 构建工具选择；
- 单元测试的编写；
- 版本号规范解析。

3、主要内容

3.1、react组件的设计原则

有意义

- 命名准确，充分表意。
- 参数准确，必要的类型检查。
- 适当的注释。

通用性

- 不要耦合特殊的业务功能。
- 不要包含特定的代码处理逻辑。

▼ Bad case

TSX | 复制代码

```
1  import React from 'react'
2
3  ▼ interface CardProps {
4      title: string;
5      children: React.ReactNode;
6      install: boolean; // 某业务参数
7  }
8  ▼ function Card(props: CardProps) {
9      return (
10         <div className="card">
11             <div className="card-title">{props.title}</div>
12             <div className="card-content">{props.children}</div>
13             {
14                 props.install ?
15                     <span className="card-tag">已安装</span> :
16                     <span className="card-tag">未安装</span>
17             }
18         </div>
19     )
20 }
21
```

▼ Good case

TSX | 复制代码

```
1  import React from 'react'
2
3  ▼ interface CardProps {
4      title: string;
5      children: React.ReactNode;
6      tag: string | null;
7  }
8  ▼ function Card(props: CardProps) {
9      return (
10         <div className="card">
11             <div className="card-title">{props.title}</div>
12             <div className="card-content">{props.children}</div>
13             {props.tag && <span className="card-tag">{props.tag}</span>}
14         </div>
15     )
16 }
```

无状态，无副作用

- 状态向上层提取，尽量少用内部状态。
- 解耦IO操作。

避免过度封装

- 合理冗余。
- 避免过度抽象。

单一职责

- 一个组件只完成一个功能。
- 尽量避免不同组件间相互依赖、循环依赖。

易于测试

- 更容易的单元测试覆盖。

3.2、组件文档编写

文档结构

- 组件描述
- 使用示例及代码演示
- 组件入参描述

例子：

antdesign的文档：

组件总览

通用

Button 按钮

Icon 图标

Typography 排版

布局

Divider 分割线

Grid 栅格

Layout 布局

Space 间距

导航

Affix 钉钉

Breadcrumb 面包屑

Dropdown 下拉菜单

Menu 导航菜单

PageHeader 页头

Pagination 分页

Steps 步骤条

数据录入

AutoComplete 自动完成

Cascader 级联选择

Checkbox 多选框

Breadcrumb 面包屑

显示当前页面在系统层级结构中的位置，并能向上返回。

何时使用

- 当系统拥有超过两级以上的层级结构时；
- 当需要告知用户『你在那里』时；
- 当需要向上导航的功能时。

代码演示

Home / Application Center / Application List / An Application

基本
最简单的用法。

Application List / Application

带有图标的
图标放在文字前面。

Home > Application Center > Application List > An Application

分隔符
使用 separator=">" 可以自定义分隔符。

Location : Application Center / Application List / An Application

分隔符
使用 Breadcrumb.Separator 可以自定义分隔符。

Ant Design / Component / General / Button

基本

带有图标的

react-router V

分隔符

带下拉菜单的...

分隔符

API

API

Breadcrumb

参数	说明	类型	默认值	版本
itemRender	自定义链接函数，和 react-router 配置使用	<code>(route, params, routes, paths) => ReactNode</code>	-	
params	路由的参数	<code>object</code>	-	
routes	router 的路由栈信息	<code>routes[]</code>	-	
separator	分隔符自定义	<code>ReactNode</code>	<code>/</code>	

Breadcrumb.Item

参数	说明	类型	默认值	版本
className	自定义类名	<code>string</code>	-	
dropdownProps	弹出下拉菜单的自定义配置	<code>Dropdown</code>	-	
href	链接的目的地	<code>string</code>	-	
overLay	下拉菜单的内容	<code>Menu () => Menu</code>	-	
onClick	单击事件	<code>(e:MouseEvent) => void</code>	-	

Breadcrumb.Separator

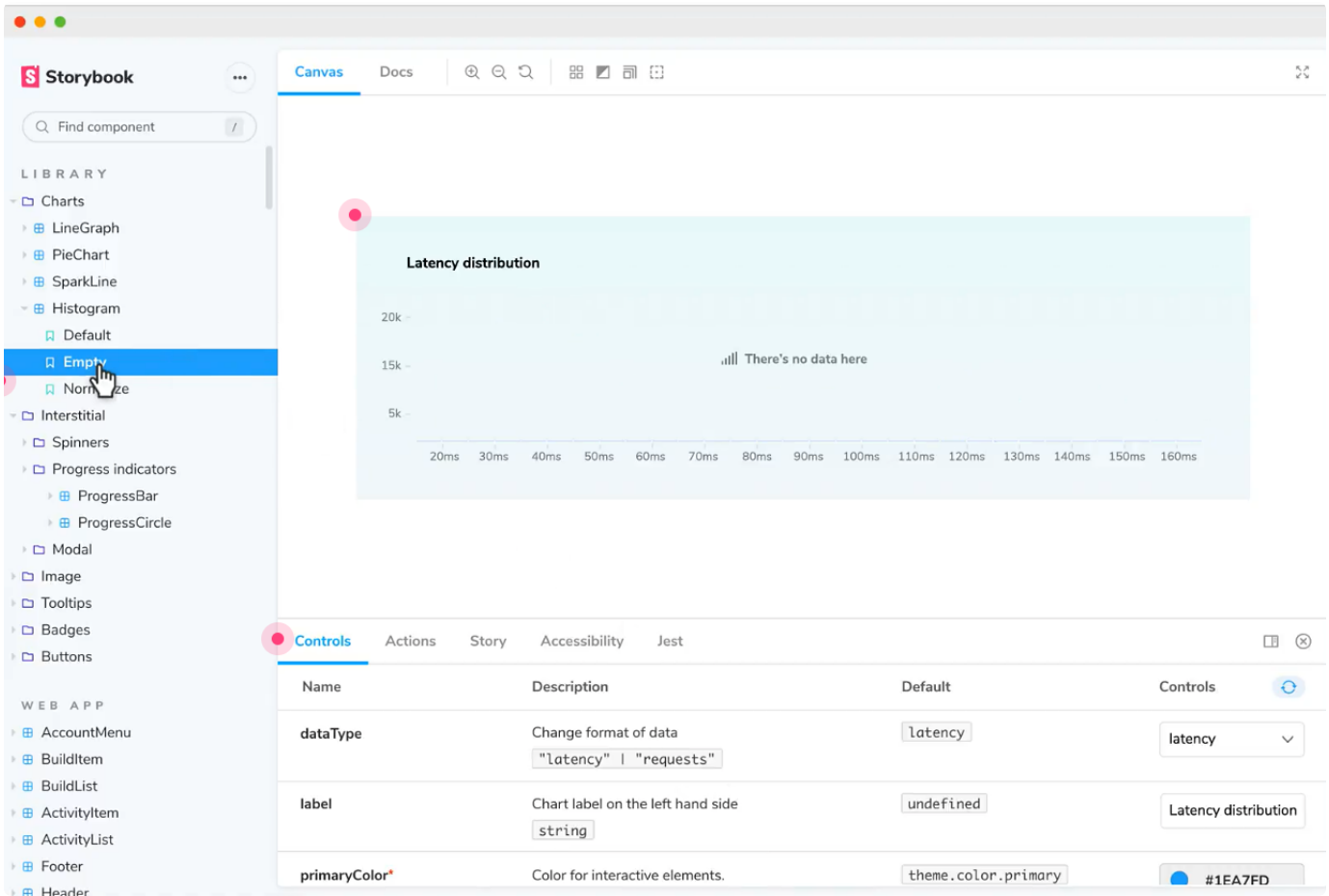
参数	说明	类型	默认值	版本
children	要显示的分隔符	<code>ReactNode</code>	<code>/</code>	

注意：在使用 `Breadcrumb.Separator` 时，其父组件的分隔符必须设置为 `separator=""`，否则会出现父组件默认的分隔符。

3.3、业界成熟的组件库脚手架

随着前端技术发展，组件库已成为前端必要的基础设施之一，业界也涌现了一些成熟的组件库脚手架，下面为大家介绍两个生态完善，开箱即用的脚手架：

Storybook



Dumi

dumi由蚂蚁金服的umi团队开源，广泛应用于阿里生态的各开源前端仓库，其特点是构建部署方便、能够快速生成界面美观的文档，配套生态完善。且有中文文档，排查问题方便。

dumi

为组件开发场景而生的文档工具

快速上手



开箱即用

考究的默认配置和约定的目录结构，帮助开发者零成本上手，让所有注意力都能放在文档编写和组件开发上



为组件开发而生

丰富的 Markdown 扩展，不止于渲染组件 demo，使得组件的文档不仅易于编写、管理，还好看、好用



主题系统

渐进式的自定义主题能力，小到扩展自己的 Markdown 标签，大到自定义完整主题包，全由你定



API 自动生成

可基于 TypeScript 类型定义自动生成组件 API，组件永远『表里如一』



移动端组件库研发
















安装主题包即可快速应用移动端组件研发能力，内置移动端高清渲染方案



资产数据化能力

一行命令将组件资产数据化，标准化的资产数据可与下游生产力工具串联

谁在使用

 ahooks	 alibaba	 ant-design	 Dooringx	 Family
 GGEditor	 Graphin	 Issues-helper	 LightProxy	 pinyin
				

以上内容是react组件库的基本篇，结合以上内容我们就可以搭建出一个基本的react组件库了。接下来我们深入组件库细节，从0到1搭建我们自己的组件库。

3.4、组件库架构差异及各自使用场景

Multirepo

一个仓库内只一个项目，以一个npm包发布，适用于基础组件库。

优点

- 项目简单，调试安装比较方便。

缺点

- 项目庞大时构建和发布耗时长。
- 组件库使用时需整体引入，造成一定的资源浪费。（可通过es module方式解决）

典型案例

- antdesign

Monorepo

一个仓库内管理多个项目，以多个npm包方式发布，依赖集中管理，npm包版本可以集中管理，也可以单独管理。通常适用于有一定关联的组件，但各组件需要支持单独的npm包发布和安装。

优点

- 共同依赖统一管理，版本控制更加容易，依赖管理会变的方便。
- 支持组件的单独发布和单独构建。
- 使用时可以单独引入。

缺点

- 项目搭建复杂度高。

典型案例

- [react](#)

Monorepo管理工具

- lerna
- yarn workspace
- pnpm

接下来，我们开始基于lerna，0到1搭建一个组件库：

3.5、引入代码规范和提交规范

Eslint&Prettier

一个高质量的组件库，eslint和prettier是必须的，能够帮助我们统一整个仓库的代码规范。

常用的eslint配置：

▼ TSX | 复制代码

```
1  "eslint:recommended",
2  "plugin:react/recommended",
3  "plugin:react-hooks/recommended",
4  // 如果使用ts
5  "plugin:@typescript-eslint/eslint-recommended",
6  "plugin:@typescript-eslint/recommended",
7  "prettier"
```

也可以使用业界成熟的eslint配置：

- [@umijs/fabric](#)

▼ .eslintrc.js

TSX | 复制代码

```
1 ▾ module.exports = {  
2   extends: [require.resolve('@umijs/fabric/dist/eslint')],  
3 };
```

▼ .stylelintrc.js

TSX | 复制代码

```
1 ▾ module.exports = {  
2   extends: [require.resolve('@umijs/fabric/dist/stylelint')],  
3 };
```

▼ .prettierrc.js

TSX | 复制代码

```
1  const fabric = require('@umijs/fabric');  
2  
3 ▾ module.exports = {  
4   ...fabric.prettier,  
5 };
```

以上是eslint的配置，需要我们手动执行，万一我们忘记手动执行怎么办？

我们可以使用lint-staged：

staged 是 Git 里的概念，表示暂存区，lint-staged 表示只检查并矫正暂存区中的文件。一来提高校验效率，二来可以为老的项目带去巨大的方便。

▼ package.json

JavaScript | 复制代码

```
1 ▾ "lint-staged": {  
2 ▾   "*.tsx": [  
3     "eslint --fix",  
4     "git add"  
5   ]  
6 },
```

Typescript

推荐大家在项目中使用typescript，良好的类型定义也是一个必须标准。

这里给大家贴一份常用的tsconfig.json配置。

▼ tsconfig.json

TSX | 复制代码

```
1  {
2    "compilerOptions": {
3      "outDir": "dist",
4      "module": "commonjs",
5      "target": "es5",
6      "lib": ["esnext", "dom"],
7      "baseUrl": ".",
8      "jsx": "react",
9      "resolveJsonModule": true,
10     "allowSyntheticDefaultImports": true,
11     "moduleResolution": "node",
12     "forceConsistentCasingInFileNames": true,
13     "noImplicitReturns": true,
14     "suppressImplicitAnyIndexErrors": true,
15     "noUnusedLocals": true,
16     "experimentalDecorators": true,
17     "strict": true,
18     "skipLibCheck": true,
19     "declaration": true
20   },
21   "exclude": [
22     "node_modules",
23     "build",
24     "dist",
25   ],
26   "include": ["src/**/*.ts"]
27 }
```

commitizen & commitlint & husky

commitizen帮助我们自动生成统一格式的提交前缀，能够在多人协作开发时，保持统一格式的提交记录。

commitizen有很多提交规则，由于我们使用lerna搭建项目。所以使用cz-lerna-changelog规则：

package.json

Shell

复制代码

```
1  "scripts": {
2    "c": "git-cz"
3  },
4  "config": {
5    "commitizen": {
6      "path": "./node_modules/cz-lerna-changelog"
7    }
8  },
```

commitlint 能够帮我吗检查错误格式的commit提交。

commitlint.config.js

JavaScript

复制代码

```
1  module.exports = { extends: ['@commitlint/config-conventional'] }
```

husky能够拦截格式错误的commit提交。

package.json

Shell

复制代码

```
1  {
2    "husky": {
3      "hooks": {
4        "commit-msg": "commitlint -E HUSKY_GIT_PARAMS"
5      }
6    }
7  }
```

3.6、组件库文档

Docz

docz的使用方法很简单，在安装完后，我们在package.json中加入如下命令，就可以使用了。

▼ package.json

Shell | 复制代码

```
1  "scripts": {
2    "docz:dev": "docz dev",
3    "docz:build": "docz build",
4    "docz:serve": "docz build && docz serve"
5  },
```

3.7、构建工具的选择

Rollup vs Webpack

Webpack

- **代码分割**：Webpack可以将你的 app 分割成许多个容易管理的分块，这些分块能够在用户使用你的 app 时按需加载。这意味着你的用户可以有更快的交互体验
- **静态资源导入**：图片、CSS 等静态资源可以直接导入到你的 app 中，就和其它的模块、节点一样能够进行依赖管理。

Rollup

- **Tree Shaking**：是rollup提出的一个特性，利用的es6模块的静态特性对导入的模块进行分析，只抽取使用到的方法，从而减小打包体积。
- 配置使用简便，生成的代码相对于Webpack更简洁。
- 可以指定生成生产中使用的各种不同的模块（amd,commonjs,es,umd）。

@umi/father

基于rollup，配置简单，支持多种架构的组件库打包。

最简单的配置：

▼ .fatherrc.js

JavaScript | 复制代码

```
1  export default {
2    entry: 'src/index.js',
3  }
```

Monorepo配置：

```
1  import { readdirSync } from 'fs';
2  import { join } from 'path';
3
4  const pkgs = readdirSync(join(__dirname, 'packages')).filter(
5    pkg => pkg.charAt(0) !== '.' && ![].includes(pkg),
6  );
7
8  export default {
9    target: 'node',
10    cjs: { type: 'babel', lazy: true },
11    pkgs: [...pkgs],
12  };
13
```

3.8、编写单元测试

jest是我们常用的单元测试框架。

test-library专注于测试react组件，与之配套的还有[react-hooks-testing-library](#)，专门用来测试react-hooks。

使用jest：

结构

编写单元测试所涉及的文件应存放于以下两个目录：

- mocks/：模拟文件目录
- [name].mock.json：【例】单个模拟文件
- tests/：单元测试目录
- [target].test.js：【例】单个单元测试文件，[target]与目标文件名保持一致，当目标文件名为index时，采用其上层目录或模块名。

[target].test.js 文件常见格式

```
1  const thirdPartyModule = require('thrid-party-module')
2
3  describe('@fe/module-name' () => {
4    const mocks = {}
5
6    beforeAll(() => {})
7
8    beforeEach(() => {})
9
10   test('描述行为', () => {
11     mocks.fake.mockReturnValue('控制模拟行为的代码置于最上方')
12
13     const target = require('../target.js')
14     const result = target.foo('执行目标待测功能')
15
16     expcet(result).toBe('断言置于最下方')
17   })
18 })
19
20
```

保证每个describe内部只有mock对象、生命周期钩子函数和test函数，将模拟对象都添加到mocks对象的适当位置，将初始化操作都添加到适当的生命周期函数中。

使用test-library

React测试库是一组能让你不依赖React组件具体实现对他们进行测试的辅助工具。它让重构工作变得轻而易举，还会推动你拥抱有关无障碍的最佳实现。React测试库并不是Jest的替代方案，因为他们需要彼此，并且有不同的分工。

1. 利用react测试库渲染APP组件
2. 利用react测试库获取元素
3. 利用Jest来进行写测试用例和断言

```

1  import { render, screen } from '@testing-library/react';
2  import App from './App';
3  test('renders learn react link', () => {
4    render(<App />);
5    const linkElement = screen.getByText(/learn react/i);
6    expect(linkElement).toBeInTheDocument();
7  });

```

获取元素

- `getByRole` `<div role="alert"></div>`
- `getByLabelText`: `<label for="search" />`
- `getByPlaceholderText`: `<input placeholder="Search" />`
- `getByAltText`: ``
- `getByDisplayValue`: `<input value="JavaScript">`
- `getByTestId`: `<any data-testid="xxx">`

除此之外，还有`queryByxxx`和`findByxxx`的查询函数，什么时候用`get`/`query`/`find`?，需要了解它们的不同。`getBy`返回元素或者错误。`getBy`在查找不到元素时返回错误，这是非常方便的，有助于我们在开发的过程中尽早的发现自己的用例发生了错误。`findBy`用于查询一个在异步之后会被最终渲染的元素。

- `queryByText`/`findByText`
- `queryByRole`/`findByRole`
- `queryByLabelText`/`findByLabelText`
- `queryByPlaceholderText`/`findByPlaceholderText`
- `queryByAltText`/`findByAltText`
- `queryByDisplayValue`/`findByDisplayValue`

断言函数

正常情况下，这些断言函数来自Jest，但是React测试库拓展了，加入了一些自己的断言函数。

- `toBeDisabled`
- `toBeEnabled`
- `toBeEmpty`
- `toBeEmptyDOMElement`
- `toBeInTheDocument`
- `toBeInvalid`

- toBeRequired
- toBeValid
- toBeVisible
- toContainElement
- toContainHTML
- toHaveAttribute
- toHaveClass
- toHaveFocus
- toHaveFormValues
- toHaveStyle
- toHaveTextContent
- toHaveValue
- toHaveDisplayValue
- toBeChecked
- toBePartiallyChecked
- toHaveDescription

事件触发/回调处理

我们可以通过React测试库的fireEvent去模拟用户交互行为：（输入文字到Input框内）

Search组件：

```
1 function Search({ value, onChange, children }) {
2   return (
3     <div>
4       <label htmlFor="search">{children}</label>
5       <input
6         id="search"
7         type="text"
8         role="textbox"
9         value={value}
10        onChange={onChange}
11      >
12    </div>
13  );
14 }
```

TypeScript | 复制代码

我们想要测试当我们在Search的Input框内输入值时，onChange是否有按预期的被调用，则需要通过jest给我们提供的fn函数：

```
1 describe('Search', () => {
2   test('calls the onChange callback handler', () => {
3     const onChange = jest.fn();
4
5     render(
6       <Search value="" onChange={onChange}>
7         </Search>
8     );
9
10    fireEvent.change(screen.getByRole('textbox'), {
11      target: { value: 'Javascript' },
12    });
13
14    expect(onChange).toHaveBeenCalledTimes(1);
15  })
16 })
```

可以看到onChange通过fireEvent触发的情况下，只调用了一次，这个时候，我们可以使用userEvent去替代fireEvent，比起fireEvent，userEvent更加的贴近人类的交互行为，在输入文字的时候，可以看到onChange会被调用多次（这是因为userEvent更加模拟了人类的键盘输入，keyDown等）

```
1 describe('Search', async () => {
2   test('calls the onChange callback handler', async () => {
3     const onChange = jest.fn();
4
5     render(
6       <Search value="" onChange={onChange}>
7         Search:
8       </Search>
9     );
10
11    await userEvent.type(screen.getByRole('textbox'), 'JavaScript');
12
13    expect(onChange).toHaveBeenCalledTimes(10);
14  })
15 })
```

react-hooks-testing-library

[react-hooks-testing-library](#)，是一个专门用来测试React hook的库。我们知道虽然hook是一个函数，可是我们不能用测试普通函数的方法来测试它们，因为它们的实际运行会涉及到很多React运行时（runtime）的东西，react-hooks-testing-library的库来允许我们像测试普通函数一样测试我们定义的hook，这个库其实背后也是将我们定义的hook运行在一个TestComponent里面，只不过它封装了一些简易的API来简化我们的测试。

renderHook

- renderHook用来渲染hook的，它会在调用的时候渲染一个专门用来测试的TestComponent来使用我们的hook。renderHook的函数签名是renderHook(callback, options?)，它的第一个参数是一个callback函数，这个函数会在TestComponent每次被重新渲染的时候调用，因此我们可以在这个函数里面调用我们想要测试的hook。
- renderHook的返回值是RenderHookResult对象，这个对象会有下面这些属性：
 - result: result是一个对象，它包含两个属性，一个是current，它保存的是renderHookcallback的返回值，另外一个属性是error，它用来存储hook在render过程中出现的任何错误。
 - rerender: rerender函数是用来重新渲染TestComponent的，它可以接收一个newProps作为参数，这个参数会作为组件重新渲染时的props值，同样renderHook的callback函数也会使用这个新的props来重新调用。
 - unmount: unmount函数是用来卸载TestComponent的，它主要用来覆盖一些useEffect cleanup函数的场景。

act

我们知道组件状态更新的时候（setState），组件需要被重新渲染，而这个重渲染是需要React进行调度的，因此是个异步的过程，我们可以通过使用act函数将所有会更新到组件状态的操作封装在它的callback里面来保证act函数执行完之后我们定义的组件已经完成了重新渲染。

```
1 // somewhere/useCounter.js
2 import { useState, useCallback } from 'react'
3
4 function useCounter() {
5   const [count, setCount] = useState(0)
6
7   const increment = useCallback(() => setCount(x => x + 1), [])
8   const decrement = useCallback(() => setCount(x => x - 1), [])
9
10  return {count, increment, decrease}
11 }
12
13 describe('decrement', () => {
14   it('decrease counter by 1', () => {
15     const { result } = renderHook(() => useCounter())
16
17     act(() => {
18       result.current.decrement()
19     })
20
21     expect(result.current.count).toBe(-1)
22   })
23 })
```

单元测试编写原则

- 每个单元测试应该有个好名字
- 将内部逻辑与外部请求分开测试
- 要保证单元测试的外部环境尽量和实际使用时是一致的
- 对服务边界（interface）的输入和输出进行严格验证
- 用断言来代替原生的报错函数
- 避免随机结果
- 尽量避免断言时间的结果
- 测试用例之间相互隔离，不要相互影响
- 原子性，所有的测试只有两种结果：成功和失败
- 避免测试中的逻辑，即不该包含if、switch、for、while等
- 不要保护起来，try...catch...
- 每个用例只测试一个关注点
- 3A策略：arrange, action, assert

3.9、版本号管理

我们的组件库使用npm发布，版本号规范也使用npm标准的semver规范。

版本格式：主版本号.次版本号.修订号，版本号递增规则如下：

1. 主版本号：当你做了不兼容的 API 修改，
2. 次版本号：当你做了向下兼容的功能性新增，
3. 修订号：当你做了向下兼容的问题修正。

可以使用lerna version交互式的选择你的版本号。