[安装 2](#_Toc32193)

[三种使用方式 2](#_Toc16715)

[在组件内部使用mobx 2](#_Toc15340)

[组件内独立class类 2](#_Toc25787)

[组件外独立class类 3](#_Toc9959)

[MobX 要点 4](#_Toc22150)

[概念与原则 5](#_Toc8566)

[概念 5](#_Toc16048)

[原则 5](#_Toc2705)

[MobX API参考 5](#_Toc9014)

[observables 5](#_Toc28573)

[创建 5](#_Toc26072)

[装饰器(Decorators) 6](#_Toc21891)

[可用装饰器列表 6](#_Toc8544)

[使用装饰器 7](#_Toc30562)

[decorate 7](#_Toc27134)

[Computed values(计算值) 7](#_Toc4082)

[用法 7](#_Toc20914)

[computed选项 7](#_Toc10957)

[错误处理 8](#_Toc22748)

[Actions(动作) 8](#_Toc9531)

[用法 8](#_Toc29799)

[Reactions(反应) & Derivations(衍生) 8](#_Toc572)

[observer 8](#_Toc24103)

[用法 8](#_Toc5077)

[autorun 9](#_Toc18795)

[用法 9](#_Toc19450)

[选项 9](#_Toc9420)

[when 9](#_Toc22916)

[用法 9](#_Toc7103)

[options 9](#_Toc6675)

[reaction 9](#_Toc23930)

[用法 10](#_Toc28624)

[options 10](#_Toc7200)

[onReactionError 10](#_Toc16999)

[用法 10](#_Toc3680)

# 安装

安装: npm install mobx --save。

React 绑定库: npm install mobx-react --save。

# 三种使用方式

## 在组件内部使用mobx

import React from 'react';

import { observable } from 'mobx';

import { observer } from 'mobx-react';

@observer

class Counter extends React.Component {

@observable count = 0;

handleInc = () => {

this.count++;

}

handleDec = () => {

this.count--;

}

render() {

return (

<div>

<p>当前值： {this.count}</p>

<button onClick={this.handleInc}>+1</button>

<button onClick={this.handleDec}>-1</button>

</div>

)

}

}

export default Counter;

## 组件内独立class类

有点像ViewModel

import React from 'react';

import { observable, action } from 'mobx';

import { observer } from 'mobx-react';

class CounterStore {

@observable count = 0;

@action inc = () => {

this.count++;

}

@action dec = () => {

this.count--;

}

}

const store = new CounterStore();

@observer

class Counter extends React.Component {

render() {

return (

<div>

<p>当前值： {store.count}</p>

<button onClick={store.inc}>+1</button>

<button onClick={store.dec}>-1</button>

</div>

)

}

}

export default Counter;

## 组件外独立class类

// CounterStore.js

import { observable, action, useStrict } from 'mobx';

useStrict(true);

class CounterStore {

@observable count = 0;

@action inc = () => {

this.count++;

}

@action dec = () => {

this.count--;

}

}

export const counterStore = new CounterStore();

// Counter.jsx

import React from 'react';

import { observer } from 'mobx-react';

import { counterStore } from './CounterStore';

@observer

class Counter extends React.Component {

render() {

return (

<div>

<p>当前值： {counterStore.count}</p>

<button onClick={counterStore.inc}>+1</button>

<button onClick={counterStore.dec}>-1</button>

</div>

)

}

}

export default Counter;

> 可以看到上面使用了`useStrict(true);`，其作用是：使非action不能修改可观察值

# MobX 要点

任何源自应用状态的东西都应该自动地获得。

响应式应用三步骤：

1. 定义状态并使其可观察；@observable注释标记所有会随时间变化的属性

2. 创建视图以响应状态的变化

3. 更改状态；严格模式下，只能通过action更改@observable属性。

# 概念与原则

## 概念

State(状态)，状态 是驱动应用的数据。

Derivations(衍生)，任何 源自状态并且不会再有任何进一步的相互作用的东西就是衍生。

衍生以多种形式存在:用户界面、衍生数据、后端集成。

衍生类型：Computed values(计算值)、Reactions(反应)。

如果你想创建一个基于当前状态的值时，请使用 computed。

Actions(动作)，动作 是任一一段可以改变状态的代码。严格模式下只动作才可以修改状态。

## 原则

MobX 支持单向数据流，也就是动作改变状态，而状态的改变会更新所有受影响的视图。

当状态改变时，所有衍生都会进行原子级的自动更新。因此永远不可能观察到中间值。

所有衍生默认都是同步更新。这意味着例如动作可以在改变状态之后直接可以安全地检查计算值。

计算值 是延迟更新的。任何不在使用状态的计算值将不会更新，直到需要它进行副作用（I / O）操作时。 如果视图不再使用，那么它会自动被垃圾回收。

所有的计算值都应该是纯净的。它们不应该用来改变状态。

# MobX API参考

observable、 computed、 reactions、 actions

## observables

### 创建

observable(value)

observable.box(value, options?)observable.object(value, decorators?, options?)

observable.array(value, options?)

observable.map(value, options?)

@observable classProperty = value；extendObservable(this, { property: value }) 的语法糖。

extendObservable(target, properties, decorators?, options?)

#### 类型转换

Map--Observable Map

toJS();转为普通映射

toJSON();转为浅式普通对象

intercept(interceptor);拦截变化

observe(listener,fireImmediately);注册侦听器

merge(values);合并values

replace(values);替换映射全部内容；等效于.clear().merge(values);

Array--Observable Array

intercept(interceptor);拦截变化

observe(listener,fireImmediately);注册侦听器

replace(values);替换映射全部内容；

clear();清空

find(predicate:(item,index,array)=>boolean,thisArg?);

findIndex(predicate:(item,index,array)=>boolean,thisArg?);

remove(value);移除

peek();返回有所有值的数组，类似slice();

无原型对象或原型为Object.prototype--Observable Object

只观察创建observable 时存在的属性。稍后添加的属性需set 或 extendObservable。

非普通对象使用 @observable、extendObservable。

属性的 getter 会自动转变成衍生属性，就像 @computed 所做的。

observable 是自动递归到整个对象的，但不递归到非普通对象。

有原型对象、原始数据类型、函数--不变

**可封装为 Boxed Observable**

显式地调用 observable.box(value)

在类定义时使用 @observable

调用 decorate()

在类中使用 extendObservable() 来引入属性

**Boxed Observable方法**

get();返回当前值。

set(value);替换当前存储的值并通知所有观察者。

intercept(interceptor);拦截变化

observe(listener,fireImmediately);注册侦听器

# 装饰器(Decorators)

## 可用装饰器列表

observable.deep: 所有 observable 都使用的默认的装饰器。

observable.ref: 禁用自动的 observable 转换，只是创建一个 observable 引用。

observable.shallow: 与集合组合使用。将集合转换为浅 observable,集合中的值将不转变。

computed: 创建一个衍生属性, 参见 computed

action: 创建一个动作, 参见 action

action.bound: 创建有范围的动作, 参见 action

## 使用装饰器

@observable.shallow tasks = [];

const taskStore = observable({tasks: [],addTask(task) { /\* ... \*/ }},

{tasks: observable.shallow, addTask: action});

## decorate

decorate(object, decorators);无法使用@decorator语法或需要更细粒度的控制时使用

decorate(TodoList, {

todos: observable,

unfinishedTodoCount: computed,

addTodo: action.bound

})

# Computed values(计算值)

响应式的产生一个可以被其它 observer 使用的值，请使用 @computed。

如想要达到一个效果，请使用 autorun。

需被其它计算属性或 reaction使用，才会计算；如需一直计算，使用 [observe](https://cn.mobx.js.org/refguide/observe.html) 或 [keepAlive](https://github.com/mobxjs/mobx-utils" \l "keepalive" \t "https://cn.mobx.js.org/refguide/_blank) 来强制保持计算值总是处于唤醒状态。

## 用法

computed(() => expression)

computed(() => expression, (newValue) => void)

computed(() => expression, options)

@computed({equals: compareFn}) get classProperty() { return expression; }

@computed get classProperty() { return expression; }

decorate(object, {property: computed})

observable.object、extendObservable 会自动将getter属性推导成计算属性

## computed选项

name: string, 在 spy 和 MobX 开发者工具中使用的调试名称

context: 在提供的表达式中使用的 this。一个对象？

equals: (value, value) => boolean，默认值是 comparer.default(===) 。

requiresReaction: boolean 在重新计算衍生属性之前，等待追踪的 observables 值发生变化。对于非常昂贵的计算值，推荐设置成 true。

keepAlive: 如果没有任何人观察到，则不要使用此计算值。容易导致内存泄漏。

get: () => value， 重载计算属性的 getter

set: (value) => void， 重载计算属性的 setter。第二个参数为函数，则将其作为setter。

## 错误处理

如果计算值在其计算期间抛出异常，则此异常将捕获并在读取其值时重新抛出。

强烈建议始终抛出“错误”，以便保留原始堆栈跟踪。

# Actions(动作)

当启用严格模式时，需要强制使用 action。

只在(最外层的)动作完成后通知计算值和反应。中间值不可见。

不支持使用 setters 的 @action 装饰器，但计算属性的 setters 是自动的动作。

## 用法

action(fn)

action(name, fn)

@action classMethod

@action(name) classMethod name为绑定的对象

@action boundClassMethod = (args) => { body }

@action(name) boundClassMethod = (args) => { body }

@action.bound boundClassMethod(args) { body } 不可与箭头函数一起使用

runInAction(name?, fn),action(name, fn)()的语法糖，用于一次性动作。

# Reactions(反应) & Derivations(衍生)

计算值 是自动响应状态变化的值。 反应 是自动响应状态变化的副作用。

## observer

包裹 React 组件的高阶组件。render中使用的observable变化时自动重新渲染。

### 用法

observer(React.createClass({ ... }))

observer((props, context) => ReactElement)

observer(class MyComponent extends React.Component { ... })

@observer class MyComponent extends React.Component { ... }

## autorun

追踪在sideEffect运行期间访问过的 observable 的状态。 将来如果有其中一个已使用的observable 发生变化，同样的sideEffect会再运行一遍。 autorun 返回一个清理函数用来取消副作用。

### 用法

autorun(() => { sideEffect }, options)

### 选项

name?: string: 用于识别和调试的名称

delay?: number: 使副作用延迟和防抖的时间。默认为 0

onError?: (error) => void: 如果 autorun 函数抛出异常，则触发错误处理函数

scheduler?: (callback) => void: 设置自定义调度器以决定如何调度 autorun 函数的重新运行

## when

condition 表达式会自动响应任何它所使用的 observable。 一旦表达式返回的是真值，副作用函数便会立即调用，但只会调用一次。

### 用法

when(() => condition, () => { sideEffect }, options)

### options

name?: string: 用于识别和调试的名称

onError?: (error) => void: 如果 断言函数 或 副作用函数 函数抛出异常，则触发错误处理函数

timeout: number 以毫秒为单位的延迟，之后将触发 onError 处理函数，以通知在指定时间内未满足条件

## reaction

追踪 observable 方面给予了更细粒度的控制。 它接收两个函数，第一个是追踪并返回数据，该数据用作第二个函数，也就是副作用的输入。 与 'autorun' 不同的是副作用起初不会运行，并且在执行副作用时访问的任何 observable 都不会被追踪。

### 用法

reaction(() => data, data => { sideEffect }, options)

### options

fireImmediately?: boolean: 在触发 副作用函数 之前等待变化。默认为 false

delay?: number: 使副作用延迟和防抖的时间。默认为 0

equals. 自定义相等函数来确定 expr 函数是否与之前的结果不同

## onReactionError

附加一个全局错误监听器，对于从 reaction 抛出的每个错误都会调用该错误监听器。 它可以用来监控或者测试。

### 用法

onReactionError(handler: (error: any, derivation) => void)