

Table of Contents

1. 自述
2. 介绍
 - i. 动机
 - ii. 三大原则
 - iii. 先前技术
 - iv. 生态系统
 - v. 示例
3. 基础
 - i. Action
 - ii. Reducer
 - iii. Store
 - iv. 数据流
 - v. 搭配 React
 - vi. 示例：Todo List
4. 高级
 - i. 异步 Action
 - ii. 异步数据流
 - iii. Middleware
 - iv. 搭配 React Router
 - v. 示例：Reddit API
 - vi. 下一步
5. 技巧
 - i. 迁移到 Redux
 - ii. 减少样板代码
 - iii. 服务端渲染
 - iv. 编写测试
 - v. 计算衍生数据
 - vi. 实现撤销重做
6. 排错

Redux 中文文档

- 7. 词汇表
- 8. API 文档
 - i. [createStore](#)
 - ii. [Store](#)
 - iii. [combineReducers](#)
 - iv. [applyMiddleware](#)
 - v. [bindActionCreators](#)
 - vi. [compose](#)
- 9. react-redux 文档
 - i. [快速入门](#)
 - ii. [API](#)
 - iii. [排错](#)
- 10. redux-tutorial

Redux 中文文档

[gitter](#) [join chat](#)

在线 Gitbook 地址: <http://cn.redux.js.org/>

英文原版: <http://redux.js.org/>

学了这个还不尽兴? 推荐极精简的 [Redux Tutorial 教程](#)

Redux 是 JavaScript 状态容器，提供可预测化的状态管理。

可以让你构建一致化的应用，运行于不同的环境（客户端、服务器、原生应用），并且易于测试。不仅于此，它还提供 超爽的开发体验，比如有一个[时间旅行调试器可以编辑后实时预览](#)。

Redux 除了和 [React](#) 一起用外，还支持其它界面库。

它体积小精悍（只有2kB）且没有任何依赖。

评价

“Love what you’re doing with Redux”

Jing Chen, Flux 作者

“I asked for comments on Redux in FB's internal JS discussion group, and it was universally praised. Really awesome work.”

Bill Fisher, Flux 作者

“It's cool that you are inventing a better Flux by not doing Flux at all.”

André Staltz, Cycle 作者

开发经历

Redux 的开发最早开始于我在准备 React Europe 演讲[热加载与时间旅行](#)的时候，当初的目标是创建一个状态管理库，来提供最简化 API，但同时做到

行为完全可预测，因此才得以实现日志打印，热加载，时间旅行，同构应用，录制和重放，而不需要任何开发参与。

启示

Redux 由 [Flux](#) 演变而来，但受 [Elm](#) 的启发，避开了 Flux 的复杂性。不管你有没有使用过它们，只需几分钟就能上手 Redux。

安装

安装稳定版：

```
npm install --save redux
```

多数情况下，你还需要使用 [React 绑定库](#) 和 [开发者工具](#)。

```
npm install --save react-redux
npm install --save-dev redux-devtools
```

要点

应用中所有的 state 都以一个对象树的形式储存在一个单一的 *store* 中。
唯一改变 state 的办法是触发 *action*，一个描述发生什么的对象。
为了描述 action 如何改变 state 树，你需要编写 *reducers*。

就是这样！

```
import { createStore } from 'redux';

/**
 * 这是一个 reducer，形式为 (state, action) => state 的纯函数。
 * 描述了 action 如何把 state 转变成下一个 state。
 */
```

```
* state 的形式取决于你，可以是基本类型、数组、对象、
* 甚至是 Immutable.js 生成的数据结构。惟一的要点是
* 当 state 变化时需要返回全新的对象，而不是修改传入的参数。
*
* 下面例子使用 `switch` 语句和字符串来做判断，但你可以写帮助类(helper)
* 根据不同的约定（如方法映射）来判断，只要适用你的项目即可。
*/
function counter(state = 0, action) {
  switch (action.type) {
    case 'INCREMENT':
      return state + 1;
    case 'DECREMENT':
      return state - 1;
    default:
      return state;
  }
}

// 创建 Redux store 来存放应用的状态。
// API 是 { subscribe, dispatch, getState }。
let store = createStore(counter);

// 可以手动订阅更新，也可以事件绑定到视图层。
store.subscribe(() =>
  console.log(store.getState())
);

// 改变内部 state 惟一方法是 dispatch 一个 action。
// action 可以被序列化，用日记记录和储存下来，后期还可以以回放的方式执行
store.dispatch({ type: 'INCREMENT' });
// 1
store.dispatch({ type: 'INCREMENT' });
// 2
store.dispatch({ type: 'DECREMENT' });
// 1
```

你应该把要做的修改变成一个普通对象，这个对象被叫做 *action*，而不是直接修改 *state*。然后编写专门的函数来决定每个 *action* 如何改变应用的 *state*，这个函数被叫做 *reducer*。

如果你以前使用 Flux，那么你只需要注意一个重要的区别。Redux 没有 Dispatcher 且不支持多个 store。相反，只有一个单一的 store 和一个根级的 reduce 函数 (reducer)。随着应用不断变大，你应该把根级的 reducer 拆成多个小的 reducers，分别独立地操作 state 树的不同部分，而不是添加新的 stores。这就像一个 React 应用只有一个根级的组件，这个根组件又由很多小组件构成。

用这个架构开发计数器有点杀鸡用牛刀，但它的美在于做复杂应用和庞大系统时优秀的扩展能力。由于它可以用 action 追溯应用的每一次修改，因此才有强大的开发工具。如录制用户会话并回放所有 action 来重现它。

文档

- [介绍](#)
- [基础](#)
- [高级](#)
- [技巧](#)
- [排错](#)
- [词汇表](#)
- [API 文档](#)

示例

- [Counter \(source\)](#)
- [TodoMVC \(source\)](#)
- [Async \(source\)](#)
- [Real World \(source\)](#)

如果你是 NPM 新手，创建和运行一个新的项目有难度，或者不知道上面的代码应该放到哪里使用，请下载 [simplest-redux-example](#) 这个示例，它是一个集成了 React、Browserify 和 Redux 的最简化的示例项目。

交流

打开 Slack，加入 [Reactiflux](#) 中的 **#redux** 频道。

感谢

- [Elm 架构](#) 介绍了使用 reducers 来操作 state 数据；
- [Turning the database inside-out](#) 大开脑洞；
- [ClojureScript](#) 里使用 [Figwheel](#) for convincing me that re-evaluation should “just work”；
- [Webpack](#) 热模块替换；
- [Flummox](#) 教我在 Flux 里去掉样板文件和单例对象；
- [disto](#) 演示使用热加载 Stores 的可行性；
- [NuclearJS](#) 证明这样的架构性能可以很好；
- [Om](#) 普及 state 惟一原子化的思想。
- [Cycle](#) 介绍了 function 是如何在很多场景都是最好的工具；
- [React](#) 实践启迪。

贡献者

定期更新，谢谢各位辛勤贡献

- [Cam Song 会影@camsong](#)
- [Jovey Zheng@jovey-zheng](#)
- [Pandazki@pandazki](#)
- [Yuwei Wang@yuweiw823](#)
- [Desen Meng@demohi](#)
- [Arcthur@arcthur](#)
- [Doray Hong@dorayx](#)
- [Guo Cheng@guocheng](#)
- [omytea](#)
- [Fred Wang](#)
- [Amo Wu](#)
- [C. T. Lin](#)
- [钱利江](#)

- 云谦
- denvey
- 三点
- Eric Wong
- Owen Yang
- Cai Huanyu

本文档翻译流程符合 [ETC 翻译规范](#)，欢迎你来一起完善

介绍

- 动机
- 三大原则
- 先前技术
- 生态系统
- 示例

动机

随着 JavaScript 单页应用开发日趋复杂，**JavaScript 需要管理比任何时候都要多的 state（状态）**。这些 state 可能包括服务器响应、缓存数据、本地生成尚未持久化到服务器的数据，也包括 UI 状态，如激活的路由，被选中的标签，是否显示加载动效或者分页器等等。

管理不断变化的 state 非常困难。如果一个 model 的变化会引起另一个 model 变化，那么当 view 变化时，就可能引起对应 model 以及另一个 model 的变化，依次地，可能会引起另一个 view 的变化。直至你搞不清楚到底发生了什么。**state 在什么时候，由于什么原因，如何变化已然不受控制。**当系统变得错综复杂的时候，想重现问题或者添加新功能就会变得举步维艰。

如果这还不够糟糕，考虑一些来自前端开发领域的新需求，如更新调优、服务端渲染、路由跳转前请求数据等等。我们前端开发者正在经受前所未有的复杂性，[我们是时候该放弃了吗？](#)当然不是。

这里的复杂性很大程度上来自于：**我们总是将两个难以厘清的概念混淆在一起：变化和异步**。我称它们为[曼妥思和可乐](#)。如果把二者分开，能做的很好，但混到一起，就变得一团糟。一些库如 [React](#) 试图在视图层禁止异步和直接操作 DOM 来解决这个问题。美中不足的是，React 依旧把处理 state 中数据的问题留给了你。Redux就是为了帮你解决这个问题。

跟随 [Flux](#)、[CQRS](#) 和 [Event Sourcing](#) 的脚步，通过限制更新发生的时间和方式，**Redux 试图让 state 的变化变得可预测**。这些限制条件反映在 Redux 的 [三大原则](#)中。

三大原则

Redux 可以用这三个基本原则来描述：

单一数据源

整个应用的 **state** 被储存在一棵 **object tree** 中，并且这个 **object tree** 只存在于唯一一个 **store** 中。

这让同构应用开发变得非常容易。来自服务端的 state 可以在无需编写更多代码的情况下被序列化并注入到客户端中。由于是单一的 state tree，调试也变得非常容易。在开发中，你可以把应用的 state 保存在本地，从而加快开发速度。此外，受益于单一的 state tree，以前难以实现的如“撤销/重做”这类功能也变得轻而易举。

```
console.log(store.getState());  
  
{  
  visibilityFilter: 'SHOW_ALL',  
  todos: [{  
    text: 'Consider using Redux',  
    completed: true,  
  }, {  
    text: 'Keep all state in a single tree',  
    completed: false  
  }]  
}
```

State 是只读的

惟一改变 state 的方法就是触发 **action**，**action** 是一个用于描述已发生事件的普通对象。

这样确保了视图和网络请求都不能直接修改 state，相反它们只能表达想要修改的意图。因为所有的修改都被集中化处理，且严格按照一个接一个的顺序执行，因此不用担心 race condition 的出现。Action 就是普通对象而已，因此它们可以被日志打印、序列化、储存、后期调试或测试时回放出来。

```
store.dispatch({
  type: 'COMPLETE_TODO',
  index: 1
});

store.dispatch({
  type: 'SET_VISIBILITY_FILTER',
  filter: 'SHOW_COMPLETED'
});
```

使用纯函数来执行修改

为了描述 action 如何改变 state tree，你需要编写 [reducers](#)。

Reducer 只是一些纯函数，它接收先前的 state 和 action，并返回新的 state。刚开始你可以只有一个 reducer，随着应用变大，你可以把它拆成多个小的 reducers，分别独立地操作 state tree 的不同部分，因为 reducer 只是函数，你可以控制它们被调用的顺序，传入附加数据，甚至编写可复用的 reducer 来处理一些通用任务，如分页器。

```
function visibilityFilter(state = 'SHOW_ALL', action) {
  switch (action.type) {
    case 'SET_VISIBILITY_FILTER':
      return action.filter;
    default:
      return state;
  }
}

function todos(state = [], action) {
  switch (action.type) {
```

```
case 'ADD_TODO':
  return [...state, {
    text: action.text,
    completed: false
 }];
case 'COMPLETE_TODO':
  return [
    ...state.slice(0, action.index),
    Object.assign({}, state[action.index], {
      completed: true
    }),
    ...state.slice(action.index + 1)
  ]
default:
  return state;
}
}

import { combineReducers, createStore } from 'redux';
let reducer = combineReducers({ visibilityFilter, todos });
let store = createStore(reducer);
```

就是这样，现在你应该明白 Redux 是怎么回事了。

先前技术

Redux 是一个混合产物。它和一些设计模式及技术相似，但也有不同之处。让我们来探索一下这些相似与不同。

Flux

Redux 可以被看作 Flux 的一种实现吗？[是](#)，也可以说[不是](#)。

（别担心，它得到了[Flux 作者的认可](#)，如果你好奇这一点。）

Redux 的灵感来源于 Flux 的几个重要特性。和 Flux 一样，Redux 规定，将模型的更新逻辑全部集中于一个特定的层（Flux 里的 store，Redux 里的 reducer）。Flux 和 Redux 都不允许程序直接修改数据，而是用一个叫作“action”的普通对象来对更改进行描述。

而不同于 Flux，**Redux 并没有 dispatcher 的概念**。原因是它依赖纯函数来替代事件处理器。纯函数构建简单，也不需额外的实体来管理它们。你可以将这点看作这两个框架的差异或细节实现，取决于你怎么看 Flux。Flux 常常被表述为 `(state, action) => state`。从这个意义上说，Redux 无疑是 Flux 架构的实现，且得益于纯函数而更为简单。

和 Flux 的另一个重要区别，是 **Redux 设想你永远不会变动你的数据**。你可以很好地使用普通对象和数组来管理 state，而不是在多个 reducer 里变动数据，这会让你深感挫折。正确的方式是，你应该在 reducer 中返回一个新对象来更新 state，同时配合 [ES7 所提议的 object spread 语法](#) 和 [Babel](#)，或者一些库，如 [Immutable](#)，这种做法简单易行。

虽然出于性能方面的考虑，[写不纯的 reducer](#) 来变动数据在技术上是可行的，但我们并不鼓励这么做。不纯的 reducer 会使一些开发特性，如时间旅行、记录/回放或热加载不可实现。此外，在大部分实际应用中，这种数据不可变动的特性并不会带来性能问题，就像 [Om](#) 所表现的，即使对象分配失败，仍可以防止昂贵的重渲染和重计算。而得益于 reducer 的纯度，应用内

的变化更是一目了然。

Elm

Elm 是一种函数式编程语言，由 [Evan Czaplicki](#) 受 Haskell 语言的启发开发。它执行一种“model view update”的架构，更新遵循 `(state, action) => state` 的规则。从技术上说，Elm 的“updater”等同于 Redux 里的 reducer。

不同于 Redux，Elm 是一门语言，因此它在执行纯度，静态类型，不可变动性，action 和模式匹配等方面更具优势。即使你不打算使用 Elm，也可以读一读 Elm 的架构，尝试一把。基于此，有一个有趣的[使用 JavaScript 库实现类似想法](#) 的项目。Redux 应该能从中获得更多的启发！为了更接近 Elm 的静态类型，Redux 可以使用一个类似 Flow 的渐进类型解决方案。

Immutable

[Immutable](#) 是一个可实现持久数据结构的 JavaScript 库。它性能很好，并且命名符合 JavaScript API 的语言习惯。

Immutable 及类似的库都可以与 Redux 对接良好。尽可随意捆绑使用！

Redux 并不在意你如何存储 state，state 可以是普通对象，不可变对象，或者其它类型。为了从 server 端写同构应用或融合它们的 state，你可能要用到序列化或反序列化的机制。但除此以外，你可以使用任何数据存储的库，**只要它支持数据的不可变动性**。举例说明，对于 Redux state，Backbone 并无意义，因为 Backbone model 是可变的。

注意，即便你使用支持 cursor 的不可变库，也不应在 Redux 的应用中使用。整个 state tree 应被视为只读，并需通过 Redux 来更新 state 和订阅更新。因此，通过 cursor 来改写，对 Redux 来说没有意义。**而如果只是想用 cursor 把 state tree 从 UI tree 解耦并逐步细化 cursor，应使用 selector 来替代。** Selector 是可组合的 getter 函数组。具体可参考 [reselect](#)，这是一个优秀、简洁的可组合 selector 的实现。

Baobab

[Baobab](#) 是另一个流行的库，实现了数据不可变特性的 API，用以更新纯 JavaScript 对象。你当然可以在 Redux 中使用它，但两者一起使用并没有什么优势。

Baobab 所提供的大部分功能都与使用 cursors 更新数据相关，而 Redux 更新数据的唯一方法是分发一个 action。可见，两者用不同方法，解决的却是同样的问题，相互并无增益。

不同于 Immutable，Baobab 在引擎下还不能现实任何特别有效的数据结构，同时使用 Baobab 和 Redux 并无裨益。这种情形下，使用普通对象会更简便。

Rx

[Reactive Extensions](#) (和它们正在进行的 [现代化重写](#)) 是管理复杂异步应用非常优秀的方案。以外，还有致力于构建将人机交互作模拟为相互依赖的可观测变量的库。

同时使用它和 Redux 有意义么？当然！它们配合得很好。将 Redux store 视作可观察变量非常简便，例如：

```
function toObservable(store) {
  return {
    subscribe({ onNext }) {
      let dispose = store.subscribe(() => onNext(store.getState()))
      onNext(store.getState())
      return { dispose }
    }
  }
}
```

使用类似方法，你可以组合不同的异步流，将其转化为 action，再提交到

```
store.dispatch()。
```

问题在于: 在已经使用了 Rx 的情况下, 你真的需要 Redux 吗? 不一定。通过 [Rx 重新实现 Redux](#) 并不难。有人说仅需使用一两句的 `.scan()` 方法即可。这种做法说不定不错!

如果你仍有疑虑, 可以去查看 Redux 的源代码 (并不多) 以及生态系统 (例如 [开发者工具](#))。如果你无意于此, 仍坚持使用交互数据流, 可以去探索一下 [Cycle](#) 这样的库, 或把它合并到 Redux 中。记得告诉我们它运作得如何!

生态系统

Redux 是一个体小精悍的库，但它相关的内容和 API 都是精挑细选的，足以衍生出丰富的工具集和可扩展的生态系统。

如果需要关于 Redux 所有内容的列表，推荐移步至 [Awesome Redux](#)。它包含了示例、样板代码、中间件、工具库，还有很多其它相关内容。

本页将只列出由 Redux 维护者审查过的一部分内容。不要因此打消尝试其它工具的信心！整个生态发展得太快，我们没有足够的时间去关注所有内容。建议只把这些当作“内部推荐”，如果你使用 Redux 创建了很酷的内容，不要犹豫，马上发个 PR 吧。

学习 Redux

演示

- [开始学习 Redux](#) — 从作者那学习 Redux 的基础知识（30 个免费的教学视频）

示例应用

- [SoundRedux](#) — 用 Redux 构建的 SoundCloud 客户端
- [Shopping Cart \(Flux Comparison\)](#) — 用于和 Flux 框架做对比的购物车示例

教程与文章

- [redux-tutorial](#) — 逐步学习使用 Redux
- [What the Flux?! Let's Redux.](#) — Redux 介绍
- [A cartoon intro to Redux](#) — 通过卡通动画来介绍 Redux 数据流向
- [Understanding Redux](#) — 学习 Redux 的基础概念

- [Handcrafting an Isomorphic Redux Application \(With Love\)](#) — 使用数据抓取与路由分发的同构应用创建指南
- [Full-Stack Redux Tutorial](#) — 使用 Redux、React 和 Immutable 的测试优先开发指南
- [Understanding Redux Middleware](#) — Redux middleware 实践的深度指南
- [A Simple Way to Route with Redux](#) — 介绍 Redux 做路由的一种简单形式

演讲

- [Live React: Hot Reloading and Time Travel](#) — 了解 Redux 如何使用限制措施，让伴随时间旅行的热加载变得简单
- [Cleaning the Tar: Using React within the Firefox Developer Tools](#) — 了解如何从已有的 MVC 应用逐步迁移至 Redux

使用 Redux

不同框架绑定

- [react-redux](#) — React
- [ng-redux](#) — Angular
- [ng2-redux](#) — Angular 2
- [backbone-redux](#) — Backbone
- [redux-falcor](#) — Falcor
- [deku-redux](#) — Deku

中间件

- [redux-thunk](#) — 用最简单的方式搭建异步 action 构造器
- [redux-promise](#) — 遵从 FSA 标准的 promise 中间件
- [redux-rx](#) — 给 Redux 用的 RxJS 工具，包括观察变量的中间件

- [redux-logger](#) – 记录所有 Redux action 和下一次 state 的日志
- [redux-immutable-state-invariant](#) – 开发中的状态变更提醒
- [redux-analytics](#) – Redux middleware 分析
- [redux-gen](#) – Redux middleware 生成器

路由

- [redux-simple-router](#) – 保持 React Router 和 Redux 同步
- [redux-router](#) – 由 React Router 绑定到 Redux 的库

组件

- [redux-form](#) – 在 Redux 中时时持有 React 表格的 state

增强器 (Enhancer)

- [redux-batched-subscribe](#) – 针对 store subscribers 的自定义批处理与防跳请求
- [redux-history-transitions](#) – 基于独断的 action 的 history 库转换
- [redux-optimist](#) – 乐观使用将被提交或还原的 action
- [redux-undo](#) – 使 reducer 具有便捷的重做/撤销，以及 action 记录功能
- [redux-ignore](#) – 通过数组或过滤功能忽略 redux action
- [redux-recycle](#) – 在确定的 action 上重置 redux 的 state
- [redux-batched-actions](#) – 单用户通知去 dispatch 多个 action

工具集

- [reselect](#) – 受 NuclearJS 启发，有效派生数据的选择器
- [normalizr](#) – 为了让 reducers 更好的消化数据，将API返回的嵌套数据规范化
- [redux-actions](#) – 在初始化 reducer 和 action 构造器时减少样板代码 (boilerplate)
- [redux-act](#) – 生成 reducer 和 action 创建函数的库

- [redux-transducers](#) — Redux 的编译器工具
- [redux-immutablejs](#) — 将Redux 和 [Immutable](#) 整合到一起的工具
- [redux-tcomb](#) — 在 Redux 中使用具有不可变特性、并经过类型检查的 state 和 action
- [redux-mock-store](#) - 模拟 redux 来测试应用

开发者工具

- [redux-devtools](#) — 一个使用时间旅行 UI 、热加载和 reducer 错误处理器的 action 日志工具，最早演示于 [React Europe 会议](#)

社区公约

- [Flux Standard Action](#) — Flux 中 action object 的人性化标准
- [Canonical Reducer Composition](#) — 嵌套 reducer 组成的武断标准
- [Ducks: Redux Reducer Bundles](#) — 关于捆绑多个 reducer, action 类型和 action 的提案

更多

[Awesome Redux](#) 是一个包含大量与 Redux 相关的库列表。

示例

Redux 源码 中同时包含了一些示例。

复制代码时注意

如果你把 Redux 示例代码复制到其它目录，可以删除位于 `webpack.config.js` 文件尾部 “You can safely delete these lines in your project.” 注释后的代码。

Counter

运行 Counter 示例：

```
git clone https://github.com/rackt/redux.git  
  
cd redux/examples/counter  
npm install  
npm start  
  
open http://localhost:3000/
```

这个示例包含：

- 基本的 Redux 应用开发流程
- 测试代码

TodoMVC

运行 TodoMVC 示例：

```
git clone https://github.com/rackt/redux.git
```

```
cd redux/examples/todomvc  
npm install  
npm start  
  
open http://localhost:3000/
```

这个示例包含：

- Redux 中使用两个 reducer 的方法
- 嵌套数据更新
- 测试代码

Todos with Undo

运行 [todos-with-undo](#) 示例：

```
git clone https://github.com/rackt/redux.git  
  
cd redux/examples/todos-with-undo  
npm install  
npm start  
  
open http://localhost:3000/
```

这个示例包含：

- Redux 中使用两个 reducer 的方法
- 使用 [redux-undo](#) 在 Redux 中实现撤销/重做功能

异步

运行 [Async](#) 示例：

```
git clone https://github.com/rackt/redux.git  
  
cd redux/examples/async  
npm install  
npm start  
  
open http://localhost:3000/
```

这个示例包含：

- 使用 [redux-thunk](#) 处理简单的异步开发流程
- 缓存服务器响应数据，并在获取数据过程中显示加载进度条
- 缓存数据过期方法

Universal

运行 [Universal](#) 示例：

```
git clone https://github.com/rackt/redux.git  
  
cd redux/examples/universal  
npm install  
npm start  
  
open http://localhost:3000/
```

这个示例包含：

- 使用 Redux 和 React 的 [Universal rendering](#) 库
- 根据输入通过异步 fetch 来预载 state
- 从服务器端向客户端传递 state

Real World

运行 [Real World](#) 示例：

```
git clone https://github.com/rackt/redux.git  
cd redux/examples/real-world  
npm install  
npm start  
  
open http://localhost:3000/
```

这个示例包含：

- 实际应用中的 Redux 的异步处理
- 使用 [normalized](#) 结果集来缓存数据
- 用于 API 请求的自定义中间件
- 缓存服务器响应数据，并在获取数据过程中显示加载进度条
- 分页器
- 路由

Shopping Cart

运行 [Shopping Cart](#) 示例：

```
git clone https://github.com/rackt/redux.git  
cd redux/examples/shopping-cart  
npm install  
npm start  
  
open http://localhost:3000/
```

这个示例包含：

- 范式化的 state

- 利用 ID 来跟踪数据中实体
- Reducer 合成
- 与 Reducer 一同定义查询方法
- 演示如何处理回滚失败
- 基于条件的 action 发布
- 只使用 [React Redux](#) 来绑定 action 创建函数
- 条件型中间件 (日志示例)

更多

参考 [Awesome Redux](#) 获取更多示例。

基础

不要被各种关于 reducers, middleware, store 的演讲所蒙蔽 —— Redux 实际是非常简单的。如果你有 Flux 开发经验, 用起来会非常习惯。没用过 Flux 也不怕, 很容易!

下面的教程将会一步步教你开发简单的 Todo 应用。

- [Action](#)
- [Reducer](#)
- [Store](#)
- [数据流](#)
- [搭配 React](#)
- [示例: Todo 列表](#)

Action

首先，让我们来给 action 下个定义。

Action 是把数据从应用（译者注：这里之所以不叫 view 是因为这些数据有可能是服务器响应，用户输入或其它非 view 的数据）传到 store 的有效载荷。它是 store 数据的唯一来源。一般来说你会通过 `store.dispatch()` 将 action 传到 store。

添加新 todo 任务的 action 是这样的：

```
const ADD_TODO = 'ADD_TODO';
```

```
{
  type: ADD_TODO,
  text: 'Build my first Redux app'
}
```

Action 本质上是 JavaScript 普通对象。我们约定，action 内使用一个字符串类型的 `type` 字段来表示将要执行的动作。多数情况下，`type` 会被定义成字符串常量。当应用规模越来越大时，建议使用单独的模块或文件来存放 action。

```
import { ADD_TODO, REMOVE_TODO } from '../actionTypes';
```

样板文件使用提醒

使用单独的模块或文件来定义 action type 常量并不是必须的，甚至根本不需要定义。对于小应用来说，使用字符串做 action type 更方便些。不过，在大型应用中把它们显式地定义成常量还是利大于弊的。参照 [减少样板代码](#) 获取保持代码干净的实践经验。

除了 `type` 字段外，`action` 对象的结构完全由你自己决定。参照 [Flux 标准 Action](#) 获取关于如何组织 `action` 的建议。

这时，我们还需要再添加一个 `action type` 来标记任务完成。因为数据是存放在数组中的，所以我们通过 `index` 来标识任务。实际项目中一般会在新建内容的时候生成唯一的 ID 作为标识。

```
{  
  type: COMPLETE_TODO,  
  index: 5  
}
```

我们应该尽量减少在 `action` 中传递的数据。比如上面的例子，传递 `index` 就比把整个任务对象传过去要好。

最后，再添加一个 `action` 类型来表示当前展示的任务状态。

```
{  
  type: SET_VISIBILITY_FILTER,  
  filter: SHOW_COMPLETED  
}
```

Action 创建函数

Action 创建函数 就是生成 `action` 的方法。“`action`”和“`action 创建函数`”这两个概念很容易混在一起，使用时最好注意区分。

在 [传统的 Flux](#) 实现中，当调用 `action 创建函数` 时，一般会触发一个 `dispatch`，像这样：

```
function addTodoWithDispatch(text) {  
  const action = {  
    type: ADD_TODO,
```

```
    text
  };
  dispatch(action);
}
```

不同的是，Redux 中的 action 创建函数仅仅返回一个 action 对象。

```
function addTodo(text) {
  return {
    type: ADD_TODO,
    text
  };
}
```

这让代码更易于测试和移植。只需把 action 创建函数的结果传给 `dispatch()` 方法即可实例化 `dispatch`。

```
dispatch(addTodo(text));
dispatch(completeTodo(index));
```

或者创建一个 **被绑定的 action 创建函数** 来自动 `dispatch`：

```
const boundAddTodo = (text) => dispatch(addTodo(text));
const boundCompleteTodo = (index) => dispatch(CompleteTodo(index));
```

然后直接调用它们：

```
boundAddTodo(text);
boundCompleteTodo(index);
```

store 里能直接通过 `store.dispatch()` 调用 `dispatch()` 方法，但是多数情况下你会使用 [react-redux](#) 提供的 `connect()` 帮助器来调

用。`bindActionCreators()` 可以自动把多个 action 创建函数 绑定到 `dispatch()` 方法上。

源码

actions.js

```
/*
 * action 类型
 */

export const ADD_TODO = 'ADD_TODO';
export const COMPLETE_TODO = 'COMPLETE_TODO';
export const SET_VISIBILITY_FILTER = 'SET_VISIBILITY_FILTER';

/*
 * 其它的常量
 */

export const VisibilityFilters = {
  SHOW_ALL: 'SHOW_ALL',
  SHOW_COMPLETED: 'SHOW_COMPLETED',
  SHOW_ACTIVE: 'SHOW_ACTIVE'
};

/*
 * action 创建函数
 */

export function addTodo(text) {
  return { type: ADD_TODO, text };
}

export function completeTodo(index) {
  return { type: COMPLETE_TODO, index };
}

export function setVisibilityFilter(filter) {
  return { type: SET_VISIBILITY_FILTER, filter };
}
```

}

下一步

现在让我们 [开发一些 reducers](#) 来指定发起 action 后 state 应该如何更新。

高级用户建议

如果你已经熟悉这些基本概念且已经完成了这个示例，不要忘了看一下在 [高级教程](#) 中的 [异步 actions](#)，你将学习如何处理 AJAX 响应和如何把 action 创建函数组合成异步控制流。

Reducer

Action 只是描述了有事情发生了这一事实，并没有指明应用如何更新 state。而这正是 reducer 要做的事情。

设计 State 结构

在 Redux 应用中，所有的 state 都被保存在一个单一对象中。建议在写代码前先想一下这个对象的结构。如何才能以最简的形式把应用的 state 用对象描述出来？

以 todo 应用为例，需要保存两个不同的内容：

- 当前选中的任务过滤条件；
- 真实的任务列表。

通常，这个 state 树还需要存放其它一些数据，例如 UI 相关的 state。这样做没问题，但尽量把这些数据与 UI 相关的 state 分开。

```
{  
  visibilityFilter: 'SHOW_ALL',  
  todos: [{  
    text: 'Consider using Redux',  
    completed: true,  
  }, {  
    text: 'Keep all state in a single tree',  
    completed: false  
  }]  
}
```

处理 Reducer 关系时的注意事项

开发复杂的应用时，不可避免会有一些数据相互引用。建议你尽可能地把 state 范式化，不存在嵌套。把所有数据放到一个对象里，每个数据

以 ID 为主键，不同数据相互引用时通过 ID 来查找。把应用的 state 想像成数据库。这种方法在 [normalizr](#) 文档里有详细阐述。例如，实际开发中，在 state 里同时存放 `todosById: { id -> todo }` 和 `todos: array<id>` 是比较好的方式，本文中为了保持示例简单没有这样处理。

Action 处理

现在我们已经确定了 state 对象的结构，就可以开始开发 reducer。reducer 就是一个函数，接收旧的 state 和 action，返回新的 state。

```
(previousState, action) => newState
```

之所以称作 reducer 是因为它将被传递给

`Array.prototype.reduce(reducer, ?initialValue)` 方法。保持 reducer 纯净非常重要。永远不要在 reducer 里做这些操作：

- 修改传入参数；
- 执行有副作用的操作，如 API 请求和路由跳转；
- 调用非纯函数，如 `Date.now()` 或 `Math.random()`。

在[高级篇](#)里会介绍如何执行有副作用的操作。现在只需要谨记 reducer 一定要保持纯净。只要传入参数一样，返回必须一样。没有特殊情况、没有副作用，没有 API 请求、没有修改参数，单纯执行计算。

明白了这些之后，就可以开始编写 reducer，并让它来处理之前定义过的 [actions](#)。

我们在开始时定义默认的 state。Redux 首次执行时，state 为 `undefined`，这时候会返回默认 state。

```
import { VisibilityFilters } from './actions';
```

```
const initialState = {
  visibilityFilter: VisibilityFilters.SHOW_ALL,
  todos: []
};

function todoApp(state, action) {
  if (typeof state === 'undefined') {
    return initialState;
  }

  // 这里暂不处理任何 action,
  // 仅返回传入的 state。
  return state;
}
```

这里一个技巧是使用 [ES6 参数默认值语法](#) 来精简代码。

```
function todoApp(state = initialState, action) {
  // 这里暂不处理任何 action,
  // 仅返回传入的 state。
  return state;
}
```

现在可以处理 `SET_VISIBILITY_FILTER`。需要做的只是改变 `state` 中的 `visibilityFilter`。

```
function todoApp(state = initialState, action) {
  switch (action.type) {
    case SET_VISIBILITY_FILTER:
      return Object.assign({}, state, {
        visibilityFilter: action.filter
      });
    default:
      return state;
  }
}
```

注意:

1. 不要修改 `state`。使用 `Object.assign()` 新建了一个副本。不能这样使用 `Object.assign(state, { visibilityFilter: action.filter })`，因为它会改变第一个参数的值。你必须把第一个参数设置为空对象。也可以使用 ES7 中还在试验阶段的特性 `{ ...state, ...newState }`，参考[对象展开语法](#)。
2. 在 `default` 情况下返回旧的 `state`。遇到未知的 `action` 时，一定要返回旧的 `state`。

`Object.assign` 使用提醒

`Object.assign()` 是 ES6 特性，但多数浏览器并不支持。你要么使用 polyfill，Babel 插件，或者使用其它库如 `_.assign()` 提供的帮助方法。

`switch` 和样板代码提醒

`switch` 语句并不是严格意义上的样板代码。Flux 中真实的样板代码是概念性的：更新必须要发送、Store 必须要注册到 Dispatcher、Store 必须是对象（开发同构应用时变得非常复杂）。为了解决这些问题，Redux 放弃了 event emitters（事件发送器），转而使用纯 reducer。

很不幸到现在为止，还有很多人存在一个误区：根据文档中是否使用 `switch` 来决定是否使用它。如果你不喜欢 `switch`，完全可以自定义一个 `createReducer` 函数来接收一个事件处理函数列表，参照["减少样板代码"](#)。

处理多个 `action`

还有两个 `action` 需要处理。让我们先处理 `ADD_TODO`。

```
function todoApp(state = initialState, action) {
  switch (action.type) {
```

```

case SET_VISIBILITY_FILTER:
  return Object.assign({}, state, {
    visibilityFilter: action.filter
  });
case ADD_TODO:
  return Object.assign({}, state, {
    todos: [...state.todos, {
      text: action.text,
      completed: false
    }]
  });
default:
  return state;
}
}

```

如上，不直接修改 `state` 中的字段，而是返回新对象。新的 `todos` 对象就相当于旧的 `todos` 在末尾加上新建的 `todo`。而这个新的 `todo` 又是基于 `action` 中的数据创建的。

最后，`COMPLETE_TODO` 的实现也很好理解：

```

case COMPLETE_TODO:
  return Object.assign({}, state, {
    todos: [
      ...state.todos.slice(0, action.index),
      Object.assign({}, state.todos[action.index], {
        completed: true
      }),
      ...state.todos.slice(action.index + 1)
    ]
  });
}

```

因为我们不能直接修改却要更新数组中指定的一项数据，这里需要先把前面和后面都切开。如果经常需要这类的操作，可以选择使用帮助类 [React.addons.update](#), [updeep](#), 或者使用原生支持深度更新的库 [Immutable](#)。最后，时刻谨记永远不要在克隆 `state` 前修改它。

拆分 Reducer

目前的代码看起来有些冗余：

```
function todoApp(state = initialState, action) {
  switch (action.type) {
    case SET_VISIBILITY_FILTER:
      return Object.assign({}, state, {
        visibilityFilter: action.filter
      });
    case ADD_TODO:
      return Object.assign({}, state, {
        todos: [...state.todos, {
          text: action.text,
          completed: false
        }]
      });
    case COMPLETE_TODO:
      return Object.assign({}, state, {
        todos: [
          ...state.todos.slice(0, action.index),
          Object.assign({}, state.todos[action.index], {
            completed: true
          }),
          ...state.todos.slice(action.index + 1)
        ]
      });
    default:
      return state;
  }
}
```

上面代码能否变得更通俗易懂？这里的 `todos` 和 `visibilityFilter` 的更新看起来是相互独立的。有时 `state` 中的字段是相互依赖的，需要认真考虑，但在这个案例中我们可以把 `todos` 更新的业务逻辑拆分到一个单独的函数里：

```

function todos(state = [], action) {
  switch (action.type) {
    case ADD_TODO:
      return [...state, {
        text: action.text,
        completed: false
      }];
    case COMPLETE_TODO:
      return [
        ...state.slice(0, action.index),
        Object.assign({}, state[action.index], {
          completed: true
        }),
        ...state.slice(action.index + 1)
      ];
    default:
      return state;
  }
}

function todoApp(state = initialState, action) {
  switch (action.type) {
    case SET_VISIBILITY_FILTER:
      return Object.assign({}, state, {
        visibilityFilter: action.filter
      });
    case ADD_TODO:
    case COMPLETE_TODO:
      return Object.assign({}, state, {
        todos: todos(state.todos, action)
      });
    default:
      return state;
  }
}

```

注意 `todos` 依旧接收 `state`，但它变成了一个数组！现在 `todoApp` 只把需要更新的一部分 `state` 传给 `todos` 函数，`todos` 函数自己确定如何更新这部分数据。这就是所谓的 *reducer* 合成，它是开发 Redux 应用最基础的

模式。

下面深入探讨一下如何做 reducer 合成。能否抽出一个 reducer 来专门管理 visibilityFilter？当然可以：

```
function visibilityFilter(state = SHOW_ALL, action) {
  switch (action.type) {
    case SET_VISIBILITY_FILTER:
      return action.filter;
    default:
      return state;
  }
}
```

现在我们可以开发一个函数来做为主 reducer，它调用多个子 reducer 分别处理 state 中的一部分数据，然后再把这些数据合成一个大的单一对象。主 reducer 并不需要设置初始化时完整的 state。初始时，如果给子 reducer 传入 `undefined` 只要返回它们的默认值即可。

```
function todos(state = [], action) {
  switch (action.type) {
    case ADD_TODO:
      return [...state, {
        text: action.text,
        completed: false
      }];
    case COMPLETE_TODO:
      return [
        ...state.slice(0, action.index),
        Object.assign({}, state[action.index], {
          completed: true
        }),
        ...state.slice(action.index + 1)
      ];
    default:
      return state;
  }
}
```

```
function visibilityFilter(state = SHOW_ALL, action) {
  switch (action.type) {
    case SET_VISIBILITY_FILTER:
      return action.filter;
    default:
      return state;
  }
}

function todoApp(state = {}, action) {
  return {
    visibilityFilter: visibilityFilter(state.visibilityFilter, action),
    todos: todos(state.todos, action)
  };
}
```

注意每个 reducer 只负责管理全局 state 中它负责的一部分。每个 reducer 的 state 参数都不同，分别对应它管理的那部分 state 数据。

现在看起来好多了！随着应用的膨胀，我们已经学会把 reducer 拆分成独立文件来分别处理不同的数据域了。

最后，Redux 提供了 `combineReducers()` 工具类来做上面 `todoApp` 做的事情，这样就能消灭一些样板代码了。有了它，可以这样重构 `todoApp`：

```
import { combineReducers } from 'redux';

const todoApp = combineReducers({
  visibilityFilter,
  todos
});

export default todoApp;
```

注意上面的写法和下面完全等价：

```
export default function todoApp(state = {}, action) {
  return {
    visibilityFilter: visibilityFilter(state.visibilityFilter, action),
    todos: todos(state.todos, action)
  };
}
```

你也可以给它们设置不同的 key，或者调用不同的函数。下面两种合成 reducer 方法完全等价：

```
const reducer = combineReducers({
  a: doSomethingWithA,
  b: processB,
  c: c
});
```

```
function reducer(state = {}, action) {
  return {
    a: doSomethingWithA(state.a, action),
    b: processB(state.b, action),
    c: c(state.c, action)
  };
}
```

`combineReducers()` 所做的只是生成一个函数，这个函数来调用你的一系列 reducer，每个 reducer 根据它们的 key 来筛选出 state 中的一部分数据并处理，然后这个生成的函数所有 reducer 的结果合并成一个大的对象。没有任何魔法。

ES6 用户使用注意

`combineReducers` 接收一个对象，可以把所有顶级的 reducer 放到一个独立的文件中，通过 `export` 暴露出每个 reducer 函数，然后使用 `import * as reducers` 得到一个以它们名字作为 key 的 object：

```
import { combineReducers } from 'redux';
import * as reducers from './reducers';

const todoApp = combineReducers(reducers);
```

由于 `import *` 还是比较新的语法，为了避免困惑，我们不会在文档使用它。但在一些社区示例中你可能会遇到它们。

源码

reducers.js

```
import { combineReducers } from 'redux';
import { ADD_TODO, COMPLETE_TODO, SET_VISIBILITY_FILTER, VisibilityFilters } from './actions';
const { SHOW_ALL } = VisibilityFilters;

function visibilityFilter(state = SHOW_ALL, action) {
  switch (action.type) {
    case SET_VISIBILITY_FILTER:
      return action.filter;
    default:
      return state;
  }
}

function todos(state = [], action) {
  switch (action.type) {
    case ADD_TODO:
      return [...state, {
        text: action.text,
        completed: false
      }];
    case COMPLETE_TODO:
      return [
        ...state.slice(0, action.index),
        Object.assign({}, state[action.index], {
          completed: true
        }),
        ...state.slice(action.index + 1)
      ];
  }
}
```

```
        ...state.slice(action.index + 1)
    ];
default:
    return state;
}
}

const todoApp = combineReducers({
    visibilityFilter,
    todos
});

export default todoApp;
```

下一步

接下来会学习 [创建 Redux store](#)。store 能维持应用的 state，并在当你发起 action 的时候调用 reducer。

Store

在前面的章节中，我们学会了使用 `action` 来描述“发生了什么”，和使用 `reducers` 来根据 `action` 更新 `state` 的用法。

Store 就是把它们联系到一起的对象。Store 有以下职责：

- 维持应用的 `state`；
- 提供 `getState()` 方法获取 `state`；
- 提供 `dispatch(action)` 方法更新 `state`；
- 通过 `subscribe(listener)` 注册监听器。

再次强调一下 **Redux 应用只有一个单一的 store**。当需要拆分处理数据的逻辑时，使用 `reducer 组合` 而不是创建多个 store。

根据已有的 `reducer` 来创建 store 是非常容易的。在[前一个章节](#)中，我们使用 `combineReducers()` 将多个 `reducer` 合并成为一个。现在我们将其导入，并传递 `createStore()`。

```
import { createStore } from 'redux'
import todoApp from './reducers'
let store = createStore(todoApp)
```

`createStore()` 的第二个参数可以设置初始状态。这对开发同构应用时非常有用，可以用于把服务器端生成的 `state` 转变后在浏览器端传给应用。

```
let store = createStore(todoApp, window.STATE_FROM_SERVER);
```

发起 Actions

现在我们已经创建好了 `store`，让我们来验证一下！虽然还没有界面，我们

已经可以测试更新逻辑了。

```
import { addTodo, completeTodo, setVisibilityFilter, VisibilityFilters } from 'redux-todo';

// 打印初始状态
console.log(store.getState());

// 监听 state 更新时，打印日志
// 注意 subscribe() 返回一个函数用来注销监听器
let unsubscribe = store.subscribe(() =>
  console.log(store.getState())
);

// 发起一系列 action
store.dispatch(addTodo('Learn about actions'));
store.dispatch(addTodo('Learn about reducers'));
store.dispatch(addTodo('Learn about store'));
store.dispatch(completeTodo(0));
store.dispatch(completeTodo(1));
store.dispatch(setVisibilityFilter(VisibilityFilters.SHOW_COMPLETED));

// 停止监听 state 更新
unsubscribe();
```

可以看到 store 里的 state 是如何变化的：

```
▶ Object {visibleTodoFilter: "SHOW_ALL", todos: Array[0]}
▶ Object {visibleTodoFilter: "SHOW_ALL", todos: Array[1]}
▶ Object {visibleTodoFilter: "SHOW_ALL", todos: Array[2]}
▶ Object {visibleTodoFilter: "SHOW_ALL", todos: Array[3]}
▶ Object {visibleTodoFilter: "SHOW_ALL", todos: Array[3]}
▶ Object {visibleTodoFilter: "SHOW_ALL", todos: Array[3]}
▼ Object {visibleTodoFilter: "SHOW_COMPLETED", todos: Array[3]} ⓘ
  ▼ todos: Array[3]
    ▼ 0: Object
      completed: true
      text: "Learn about actions"
      ► __proto__: Object
    ▼ 1: Object
      completed: true
      text: "Learn about reducers"
      ► __proto__: Object
    ▼ 2: Object
      completed: false
      text: "Learn about store"
      ► __proto__: Object
      length: 3
    ► __proto__: Array[0]
  visibleTodoFilter: "SHOW_COMPLETED"
  ► __proto__: Object
```

可以看到，在还没有开发界面的时候，我们就可以定义程序的行为。而且这时候已经可以写 reducer 和 action 创建函数的测试。不需要模拟任何东西，因为它们都是纯函数。只需调用一下，对返回值做断言，写测试就是这么简单。

源码

index.js

```
import { createStore } from 'redux'
import todoApp from './reducers'

let store = createStore(todoApp)
```

下一步

在创建 todo 应用界面之前，我们先穿插学习一下[数据在 Redux 应用中如何流动的](#)。

数据流

严格的单向数据流是 Redux 架构的设计核心。

这意味着应用中所有的数据都遵循相同的生命周期，这样可以让应用变得更加可预测且容易理解。同时也鼓励做数据范式化，这样可以避免使用多个且独立的无法相互引用的重复数据。

如果这些理由还不足以令你信服，读一下 [动机](#) 和 [Flux 案例](#)，这里面有更加详细的单向数据流优势分析。虽然 Redux 就不是严格意义上的 Flux，但它们有共同的设计思想。

Redux 应用中数据的生命周期遵循下面 4 个步骤：

1. 调用 `store.dispatch(action)`。

Action 就是一个描述“发生了什么”的普通对象。比如：

```
{ type: 'LIKE_ARTICLE', articleId: 42 };  
{ type: 'FETCH_USER_SUCCESS', response: { id: 3, name: 'Mar' } };  
{ type: 'ADD_TODO', text: 'Read the Redux docs.' };
```

可以把 action 理解成新闻的摘要。如“玛丽喜欢42号文章。”或者“任务列表里添加了‘学习 Redux 文档’”。

你可以在任何地方调用 `store.dispatch(action)`，包括组件中、XHR 回调中、甚至定时器中。

2. Redux store 调用传入的 reducer 函数。

Store 会把两个参数传入 reducer：当前的 state 树和 action。例如，在这个 todo 应用中，根 reducer 可能接收这样的数据：

```
// 当前应用的 state (todos 列表和选中的过滤器)
let previousState = {
  visibleTodoFilter: 'SHOW_ALL',
  todos: [
    {
      text: 'Read the docs.',
      complete: false
    }
  ]
}

// 将要执行的 action (添加一个 todo)
let action = {
  type: 'ADD_TODO',
  text: 'Understand the flow.'
}

// render 返回处理后的应用状态
let nextState = todoApp(previousState, action);
```

注意 reducer 是纯函数。它仅仅用于计算下一个 state。它应该是完全可预测的：多次传入相同的输入必须产生相同的输出。它不应做有副作用的操作，如 API 调用或路由跳转。这些应该在 dispatch action 前发生。

3. 根 reducer 应该把多个子 reducer 输出合并成一个单一的 state 树。

根 reducer 的结构完全由你决定。Redux 原生提供 [combineReducers\(\)](#) 辅助函数，来把根 reducer 拆分成多个函数，用于分别处理 state 树的一个分支。

下面演示 [combineReducers\(\)](#) 如何使用。假如你有两个 reducer：一个是 todo 列表，另一个是当前选择的过滤器设置：

```
function todos(state = [], action) {
  // 省略处理逻辑...
  return nextState;
}
```

```
function visibleTodoFilter(state = 'SHOW_ALL', action) {
  // 省略处理逻辑...
  return nextState;
}

let todoApp = combineReducers({
  todos,
  visibleTodoFilter
})
```

当你触发 `action` 后，`combineReducers` 返回的 `todoApp` 会负责调用两个 reducer：

```
let nextTodos = todos(state.todos, action);
let nextVisibleTodoFilter = visibleTodoFilter(state.visible
```

然后会把两个结果集合并成一个 state 树：

```
return {
  todos: nextTodos,
  visibleTodoFilter: nextVisibleTodoFilter
};
```

虽然 `combineReducers()` 是一个很方便的辅助工具，你也可以选择不用；你可以自行实现自己的根 reducer！

4. Redux store 保存了根 reducer 返回的完整 state 树。

这个新的树就是应用的下一个 state！所有订阅 `store.subscribe(listener)` 的监听器都将被调用；监听器里可以调用 `store.getState()` 获得当前 state。

现在，可以应用新的 state 来更新 UI。如果你使用了 [React Redux](#) 这类的绑定库，这时就应该调用 `component.setState(newState)` 来更新。

下一步

现在你已经理解了 Redux 如何工作，是时候结合 React 开发应用了。

高级用户使用注意

如果你已经熟悉了基础概念且完成了这个教程，可以学习[高级教程](#)中的[异步数据流](#)，你将学到如何使用 middleware 在[异步 action](#) 到达 reducer 前处理它们。

搭配 React

这里需要再强调一下：Redux 和 React 之间没有关系。Redux 支持 React、Angular、Ember、jQuery 甚至纯 JavaScript。

尽管如此，Redux 还是和 [React](#) 和 [Deku](#) 这类框架搭配起来用最好，因为这类框架允许你以 state 函数的形式来描述界面，Redux 通过 action 的形式来发起 state 变化。

下面使用 React 来开发一个 todo 任务管理应用。

安装 React Redux

Redux 默认并不包含 [React 绑定库](#)，需要单独安装。

```
npm install --save react-redux
```

容器组件（Smart/Container Components）和展示组件（Dumb/Presentational Components）

Redux 的 React 绑定库包含了 [容器组件和展示组件相分离](#) 的开发思想。

明智的做法是只在最顶层组件（如路由操作）里使用 Redux。其余内部组件仅仅是展示性的，所有数据都通过 props 传入。

	容器组件	展示组件
Location	最顶层，路由处理	中间和子组件
Aware of Redux	是	否
读取数据	从 Redux 获取 state	从 props 获取数据

在这个 todo 应用中，只应有一个容器组件，它存在于组件的最顶层。在复杂的应用中，也有可能会有多个容器组件。虽然你也可以嵌套使用容器组件，但应该尽可能的使用传递 props 的形式。

设计组件层次结构

还记得当初如何 [设计 state 根对象的结构](#) 吗？现在就要定义与它匹配的界面的层次结构。其实这不是 Redux 相关的工作，[React 开发思想](#)在这方面解释的非常棒。

我们的概要设计很简单。我们想要显示一个 todo 项的列表。一个 todo 项被点击后，会增加一条删除线并标记 completed。我们会显示用户新增一个 todo 字段。在 footer 里显示一个可切换的显示全部/只显示 completed 的/只显示 incompleted 的 todos。

以下的这些组件（和它们的 props）就是从这个设计里来的：

- **AddTodo** 输入字段的输入框和按钮。
 - `onAddClick(text: string)` 当按钮被点击时调用的回调函数。
- **TodoList** 用于显示 todos 列表。
 - `todos: Array<{ text, completed }>` 形式显示的 todo 项数组。
 - `onTodoClick(index: number)` 当 todo 项被点击时调用的回调函数。
- **Todo** 一个 todo 项。
 - `text: string` 显示的文本内容。
 - `completed: boolean` todo 项是否显示删除线。
 - `onClick()` 当 todo 项被点击时调用的回调函数。
- **Footer** 一个允许用户改变可见 todo 过滤器的组件。
 - `filter: string` 当前的过滤器为： '`'SHOW_ALL'`'、'`'SHOW_COMPLETED'`' 或 '`'SHOW_ACTIVE'`'。
 - `onFilterChange(nextFilter: string)`：当用户选择不同的过滤

器时调用的回调函数。

这些全部都是展示组件。它们不知道数据是从哪里来的，或者数据是怎么变化的。你传入什么，它们就渲染什么。

如果你要把 Redux 迁移到别的上，你应该要保持这些组件的一致性。因为它们不依赖于 Redux。

直接写就是了！我们已经不用绑定到 Redux。你可以在开发过程中给出一些实验数据，直到它们渲染对了。

展示组件

这就是普通的 React 组件，所以就不在详述。直接看代码：

components/AddTodo.js

```
import React, { findDOMNode, Component, PropTypes } from 'react'

export default class AddTodo extends Component {
  render() {
    return (
      <div>
        <input type='text' ref='input' />
        <button onClick={e => this.handleClick(e)}>
          Add
        </button>
      </div>
    );
  }

  handleClick(e) {
    const node = findDOMNode(this.refs.input);
    const text = node.value.trim();
    this.props.onAddClick(text);
    node.value = '';
  }
}
```

```
AddTodo.propTypes = {
  onAddClick: PropTypes.func.isRequired
}
```

components/Todo.js

```
import React, { Component, PropTypes } from 'react';

export default class Todo extends Component {
  render() {
    return (
      <li
        onClick={this.props.onClick}
        style={{
          textDecoration: this.props.completed ? 'line-through'
            cursor: this.props.completed ? 'default' : 'pointer'
        }}>
        {this.props.text}
      </li>
    );
  }
}

Todo.propTypes = {
  onClick: PropTypes.func.isRequired,
  text: PropTypes.string.isRequired,
  completed: PropTypes.bool.isRequired
};
```

components/TodoList.js

```
import React, { Component, PropTypes } from 'react';
import Todo from './Todo';

export default class TodoList extends Component {
  render() {
```

```
return (
  <ul>
    {this.props.todos.map((todo, index) =>
      <Todo {...todo}
        key={index}
        onClick={() => this.props.onTodoClick(index)} />
    )}
  </ul>
)
}

TodoList.propTypes = {
  onTodoClick: PropTypes.func.isRequired,
  todos: PropTypes.arrayOf(PropTypes.shape({
    text: PropTypes.string.isRequired,
    completed: PropTypes.bool.isRequired
  }).isRequired).isRequired
}
```

components/Footer.js

```
import React, { Component, PropTypes } from 'react';

export default class Footer extends Component {
  renderFilter(filter, name) {
    if (filter === this.props.filter) {
      return name;
    }

    return (
      <a href="#" onClick={e => {
        e.preventDefault();
        this.props.onFilterChange(filter);
      }}>
        {name}
      </a>
    )
  }
}
```

```
render() {
  return (
    <p>
      Show:
      {' '}
      {this.renderFilter('SHOW_ALL', 'All')}
      ', '
      {this.renderFilter('SHOW_COMPLETED', 'Completed')}
      ', '
      {this.renderFilter('SHOW_ACTIVE', 'Active')}
      .
    </p>
  );
}
}

Footer.propTypes = {
  onFilterChange: PropTypes.func.isRequired,
  filter: PropTypes.oneOf([
    'SHOW_ALL',
    'SHOW_COMPLETED',
    'SHOW_ACTIVE'
  ]).isRequired
};
```

就是这些，现在开发一个笨拙型的组件 `App` 把它们渲染出来，验证下是否工作。

containers/App.js

```
import React, { Component } from 'react';
import AddTodo from '../components/AddTodo';
import TodoList from '../components/TodoList';
import Footer from '../components/Footer';

export default class App extends Component {
  render() {
    return (
      <div>
        <AddTodo
```

```
        onAddClick={text =>
          console.log('add todo', text)
        } />
      <TodoList
        todos={[{
          text: 'Use Redux',
          completed: true
        }, {
          text: 'Learn to connect it to React',
          completed: false
        }]}
        onTodoClick={todo =>
          console.log('todo clicked', todo)
        } />
      <Footer
        filter='SHOW_ALL'
        onFilterChange={filter =>
          console.log('filter change', filter)
        } />
      </div>
    );
  }
}
```

渲染 `<App />` 结果如下：

 Add

- Use Redux
- Learn to connect it to React

Show: All, Completed, Active.

单独来看，并没有什么特别，现在把它和 Redux 连起来。

连接到 Redux

我们需要做出两个变化，将 `App` 组件连接到 Redux 并且让它能够 dispatch actions 以及从 Redux store 读取到 state。

首先，我们需要获取从之前安装好的 `react-redux` 提供的 `Provider`，并在渲染之前将根组件包装进 `<Provider>`。

index.js

```
import React from 'react'
import { render } from 'react-dom'
import { createStore } from 'redux'
import { Provider } from 'react-redux'
import App from './containers/App'
import todoApp from './reducers'

let store = createStore(todoApp);

let rootElement = document.getElementById('root')
render(
  <Provider store={store}>
    <App />
  </Provider>,
  rootElement
)
```

这使得我们的 `store` 能为下面的组件所用。（在内部，这个是通过 React 的 `"context"` 特性实现。）

接着，我们想要通过 `react-redux` 提供的 `connect()` 方法将包装好的组件连接到 Redux。尽量只做一个顶层的组件，或者 route 处理。从技术上来说你可以将应用中的任何一个组件 `connect()` 到 Redux store 中，但尽量避免这么做，因为这个数据流很难追踪。

任何一个从 `connect()` 包装好的组件都可以得到一个 `dispatch` 方法作为组件的 `props`，以及得到全局 `state` 中所需的任何内容。`connect()` 的唯一参数是 `selector`。此方法可以从 Redux store 接收到全局的 `state`，然后返回组件中需要的 `props`。最简单的情况下，可以返回一个初始的 `state`（例如，返回认证方法），但最好先将其进行转化。

为了使组合 `selectors` 更有效率，不妨看看 `reselect`。在这个例子中我们不会搭配 React

用到它，但它适合更大的应用。

containers/App.js

```
import React, { Component, PropTypes } from 'react';
import { connect } from 'react-redux';
import { addTodo, completeTodo, setVisibilityFilter, VisibilityFilter } from 'redux';
import AddTodo from '../components/AddTodo';
import TodoList from '../components/TodoList';
import Footer from '../components/Footer';

class App extends Component {
  render() {
    // 通过调用 connect() 注入:
    const { dispatch, visibleTodos, visibilityFilter } = this.props;
    return (
      <div>
        <AddTodo
          onAddClick={text =>
            dispatch(addTodo(text))
          } />
        <TodoList
          todos={this.props.visibleTodos}
          onTodoClick={index =>
            dispatch(completeTodo(index))
          } />
        <Footer
          filter={visibilityFilter}
          onFilterChange={nextFilter =>
            dispatch(setVisibilityFilter(nextFilter))
          } />
      </div>
    )
  }
}

App.propTypes = {
  visibleTodos: PropTypes.arrayOf(PropTypes.shape({
    text: PropTypes.string.isRequired,
    completed: PropTypes.bool.isRequired
  })),
  visibilityFilter: PropTypes.oneOf(['all', 'active', 'completed'])
}
```

```
visibilityFilter: PropTypes.oneOf([
  'SHOW_ALL',
  'SHOW_COMPLETED',
  'SHOW_ACTIVE'
]).isRequired
}

function selectTodos(todos, filter) {
  switch (filter) {
    case VisibilityFilters.SHOW_ALL:
      return todos;
    case VisibilityFilters.SHOW_COMPLETED:
      return todos.filter(todo => todo.completed);
    case VisibilityFilters.SHOW_ACTIVE:
      return todos.filter(todo => !todo.completed);
  }
}

// 基于全局 state , 哪些是我们想注入的 props ?
// 注意: 使用 https://github.com/faassen/reselect 效果更佳。
function select(state) {
  return {
    visibleTodos: selectTodos(state.todos, state.visibilityFilter),
    visibilityFilter: state.visibilityFilter
  };
}

// 包装 component , 注入 dispatch 和 state 到其默认的 connect(select)
export default connect(select)(App);
```

到此为止，迷你型的任务管理应用就开发完毕。

下一步

参照 [本示例完整](#) 来深化理解。然后就可以跳到 [高级教程](#) 学习网络请求处理和路由。

示例: Todo 列表

这是我们在[基础教程](#)里开发的迷你型的任务管理应用的完整源码。

入口文件

index.js

```
import React from 'react'
import { render } from 'react-dom'
import { createStore } from 'redux'
import { Provider } from 'react-redux'
import App from './containers/App'
import todoApp from './reducers'

let store = createStore(todoApp)

let rootElement = document.getElementById('root')
render(
  <Provider store={store}>
    <App />
  </Provider>,
  rootElement
)
```

Action 创建函数和常量

actions.js

```
/*
 * action 类型
 */

export const ADD_TODO = 'ADD_TODO';
```

```
export const COMPLETE_TODO = 'COMPLETE_TODO';
export const SET_VISIBILITY_FILTER = 'SET_VISIBILITY_FILTER'

/*
 * 其它的常量
 */

export const VisibilityFilters = {
  SHOW_ALL: 'SHOW_ALL',
  SHOW_COMPLETED: 'SHOW_COMPLETED',
  SHOW_ACTIVE: 'SHOW_ACTIVE'
};

/*
 * action 创建函数
 */

export function addTodo(text) {
  return { type: ADD_TODO, text }
}

export function completeTodo(index) {
  return { type: COMPLETE_TODO, index }
}

export function setVisibilityFilter(filter) {
  return { type: SET_VISIBILITY_FILTER, filter }
}
```

Reducers

reducers.js

```
import { combineReducers } from 'redux'
import { ADD_TODO, COMPLETE_TODO, SET_VISIBILITY_FILTER, VisibilityFilters } from './actions'
const { SHOW_ALL } = VisibilityFilters

function visibilityFilter(state = SHOW_ALL, action) {
  switch (action.type) {
```

```
case SET_VISIBILITY_FILTER:  
  return action.filter  
default:  
  return state  
}  
}  
  
function todos(state = [], action) {  
  switch (action.type) {  
    case ADD_TODO:  
      return [  
        ...state,  
        {  
          text: action.text,  
          completed: false  
        }  
      ]  
    case COMPLETE_TODO:  
      return [  
        ...state.slice(0, action.index),  
        Object.assign({}, state[action.index], {  
          completed: true  
        }),  
        ...state.slice(action.index + 1)  
      ]  
    default:  
      return state  
  }  
}  
  
const todoApp = combineReducers({  
  visibilityFilter,  
  todos  
)  
  
export default todoApp
```

容器组件

containers/App.js

```
import React, { Component, PropTypes } from 'react'
import { connect } from 'react-redux'
import { addTodo, completeTodo, setVisibilityFilter, VisibilityFilter } from '../actions'
import AddTodo from '../components/AddTodo'
import TodoList from '../components/TodoList'
import Footer from '../components/Footer'

class App extends Component {
  render() {
    // Injected by connect() call:
    const { dispatch, visibleTodos, visibilityFilter } = this.props
    return (
      <div>
        <AddTodo
          onAddClick={text =>
            dispatch(addTodo(text))
          } />
        <TodoList
          todos={visibleTodos}
          onTodoClick={index =>
            dispatch(completeTodo(index))
          } />
        <Footer
          filter={visibilityFilter}
          onFilterChange={nextFilter =>
            dispatch(setVisibilityFilter(nextFilter))
          } />
      </div>
    )
  }
}

App.propTypes = {
  visibleTodos: PropTypes.arrayOf(PropTypes.shape({
    text: PropTypes.string.isRequired,
    completed: PropTypes.bool.isRequired
  }).isRequired).isRequired,
  visibilityFilter: PropTypes.oneOf([
    'SHOW_ALL',
  ])
}
```

```
'SHOW_COMPLETED',
'SHOW_ACTIVE'
]).isRequired
}

function selectTodos(todos, filter) {
  switch (filter) {
    case VisibilityFilters.SHOW_ALL:
      return todos
    case VisibilityFilters.SHOW_COMPLETED:
      return todos.filter(todo => todo.completed)
    case VisibilityFilters.SHOW_ACTIVE:
      return todos.filter(todo => !todo.completed)
  }
}

// Which props do we want to inject, given the global state?
// Note: use https://github.com/faassen/reselect for better performance
function select(state) {
  return {
    visibleTodos: selectTodos(state.todos, state.visibilityFilter),
    visibilityFilter: state.visibilityFilter
  }
}

// 包装 component , 注入 dispatch 和 state 到其默认的 connect(select)
export default connect(select)(App)
```

展示组件

components/AddTodo.js

```
import React, { Component, PropTypes } from 'react'

export default class AddTodo extends Component {
  render() {
    return (
      <div>
```

```
        <input type='text' ref='input' />
        <button onClick={(e) => this.handleClick(e)}>
            Add
        </button>
    </div>
)
}

handleClick(e) {
    const node = this.refs.input
    const text = node.value.trim()
    this.props.onAddClick(text)
    node.value = ''
}
}

AddTodo.propTypes = {
    onAddClick: PropTypes.func.isRequired
}
```

components/Footer.js

```
import React, { Component, PropTypes } from 'react'

export default class Footer extends Component {
    renderFilter(filter, name) {
        if (filter === this.props.filter) {
            return name
        }

        return (
            <a href='#' onClick={e => {
                e.preventDefault()
                this.props.onFilterChange(filter)
            }}>
                {name}
            </a>
        )
    }

    render() {
```

```
        return (
          <p>
            Show:
            {' '}
            {this.renderFilter('SHOW_ALL', 'All')}
            ', '
            {this.renderFilter('SHOW_COMPLETED', 'Completed')}
            ', '
            {this.renderFilter('SHOW_ACTIVE', 'Active')}

            .
          </p>
        )
      }
    }

Footer.propTypes = {
  onFilterChange: PropTypes.func.isRequired,
  filter: PropTypes.oneOf([
    'SHOW_ALL',
    'SHOW_COMPLETED',
    'SHOW_ACTIVE'
  ]).isRequired
}
```

components/Todo.js

```
import React, { Component, PropTypes } from 'react'

export default class Todo extends Component {
  render() {
    return (
      <li
        onClick={this.props.onClick}
        style={{
          textDecoration: this.props.completed ? 'line-through' : 'none',
          cursor: this.props.completed ? 'default' : 'pointer'
        }>
        {this.props.text}
      </li>
    )
  }
}
```

```
}

Todo.propTypes = {
  onClick: PropTypes.func.isRequired,
  text: PropTypes.string.isRequired,
  completed: PropTypes.bool.isRequired
}
```

components/TodoList.js

```
import React, { Component, PropTypes } from 'react'
import Todo from './Todo'

export default class TodoList extends Component {
  render() {
    return (
      <ul>
        {this.props.todos.map((todo, index) =>
          <Todo {...todo}>
            key={index}
            onClick={() => this.props.onTodoClick(index)} />
        )}
      </ul>
    )
  }
}

TodoList.propTypes = {
  onTodoClick: PropTypes.func.isRequired,
  todos: PropTypes.arrayOf(PropTypes.shape({
    text: PropTypes.string.isRequired,
    completed: PropTypes.bool.isRequired
  }).isRequired).isRequired
}
```


高级

基础章节介绍了如何组织简单的 Redux 应用。在这一章节中，将要学习如何使用 AJAX 和路由。

- [异步 Action](#)
- [异步数据流](#)
- [Middleware](#)
- [搭配 React Router](#)
- [示例：Reddit API](#)
- [下一步](#)

异步 Action

在[基础教程](#)中，我们创建了一个简单的 todo 应用。它只有同步操作。每当 dispatch action 时，state 会被立即更新。

在本教程中，我们将开发一个不同的，异步的应用。它将使用 Reddit API 来获取并显示指定 reddit 下的帖子列表。那么 Redux 究竟是如何处理异步数据流的呢？

Action

当调用异步 API 时，有两个非常关键的时刻：发起请求的时刻，和接收到响应的时刻（也可能是超时）。

这两个时刻都可能会更改应用的 state；为此，你需要 dispatch 普通的同步 action。一般情况下，每个 API 请求都至少需要 dispatch 三个不同的 action：

- **一个通知 reducer 请求开始的 action。**

对于这种 action，reducer 可能会切换一下 state 中的 `isFetching` 标记。以此来告诉 UI 来显示进度条。

- **一个通知 reducer 请求成功结束的 action。**

对于这种 action，reducer 可能会把接收到的新数据合并到 state 中，并重置 `isFetching`。UI 则会隐藏进度条，并显示接收到的数据。

- **一个通知 reducer 请求失败的 action。**

对于这种 action，reducer 可能会重置 `isFetching`。或者，有些 reducer 会保存这些失败信息，并在 UI 里显示出来。

为了区分这三种 action，可能在 action 里添加一个专门的 `status` 字段作为标记位：

```
{ type: 'FETCH_POSTS' }
{ type: 'FETCH_POSTS', status: 'error', error: 'Oops' }
{ type: 'FETCH_POSTS', status: 'success', response: { ... } }
```

又或者为它们定义不同的 type：

```
{ type: 'FETCH_POSTS_REQUEST' }
{ type: 'FETCH_POSTS_FAILURE', error: 'Oops' }
{ type: 'FETCH_POSTS_SUCCESS', response: { ... } }
```

究竟使用带有标记位的同一个 action，还是多个 action type 呢，完全取决于你。这应该是你的团队共同达成的约定。使用多个 type 会降低犯错误的几率，但是如果你使用像 [redux-actions](#) 这类的辅助库来生成 action creator 和 reducer 的话，这就完全不是问题了。

无论使用哪种约定，一定要在整个应用中保持统一。
在本教程中，我们将使用不同的 type 来做。

同步 Action Creator

下面先定义几个同步的 action type 和 action creator。比如，用户可以选择要显示的 subreddit：

actions.js

```
export const SELECT_SUBREDDIT = 'SELECT_SUBREDDIT'

export function selectSubreddit(subreddit) {
  return {
    type: SELECT_SUBREDDIT,
    subreddit
}
```

```
    }  
}
```

也可以按 "刷新" 按钮来更新它：

```
export const INVALIDATE_SUBREDDIT = 'INVALIDATE_SUBREDDIT'  
  
export function invalidatesubreddit(subreddit) {  
  return {  
    type: INVALIDATE_SUBREDDIT,  
    subreddit  
  }  
}
```

这些是用户操作来控制的 action。也有另外一类 action，是由网络请求来控制。后面会介绍如何使用它们，现在，我们只是来定义它们。

当需要获取指定 subreddit 的帖子的时候，需要 dispatch REQUEST_POSTS action：

```
export const REQUEST_POSTS = 'REQUEST_POSTS'  
  
export function requestPosts(subreddit) {  
  return {  
    type: REQUEST_POSTS,  
    subreddit  
  }  
}
```

把 SELECT_SUBREDDIT 和 INVALIDATE_SUBREDDIT 分开很重要。虽然它们的发生有先后顺序，但随着应用变得复杂，有些用户操作（比如，预加载最流行的 subreddit，或者一段时间后自动刷新过期数据）后需要马上请求数据。路由变化时也可能需要请求数据，所以一开始如果把请求数据和特定的 UI 事件耦合到一起是不明智的。

最后，当收到请求响应时，我们会 dispatch `RECEIVE_POSTS`：

```
export const RECEIVE_POSTS = 'RECEIVE_POSTS'

export function receivePosts(subreddit, json) {
  return {
    type: RECEIVE_POSTS,
    subreddit,
    posts: json.data.children.map(child => child.data),
    receivedAt: Date.now()
  }
}
```

以上就是现在需要知道的所有内容。稍后会介绍如何把 dispatch action 与网络请求结合起来。

错误处理须知

在实际应用中，网络请求失败时也需要 dispatch action。虽然在本教程中我们并不做错误处理，但是这个 [真实场景的案例](#) 会演示一种实现方案。

设计 state 结构

就像在基础教程中，在功能开发前你需要 [设计应用的 state 结构](#)。在写异步代码的时候，需要考虑更多的 state，所以我们要仔细考虑一下。

这部分内容通常让初学者感到迷惑，因为选择哪些信息才能清晰地描述异步应用的 state 并不直观，还有怎么用一个树来把这些信息组织起来。

我们以最通用的案例来打头：列表。Web 应用经常需要展示一些内容的列表。比如，帖子的列表，朋友的列表。首先要明确应用要显示哪些列表。然后把它们分开储存在 state 中，这样你才能对它们分别做缓存并且在需要的时候再次请求更新数据。

"subreddit 头条" 应用会长这个样子：

```
{  
  selected subreddit: 'frontend',  
  postsBySubreddit: {  
    frontend: {  
      isFetching: true,  
      didInvalidate: false,  
      items: []  
    },  
    reactjs: {  
      isFetching: false,  
      didInvalidate: false,  
      lastUpdated: 1439478405547,  
      items: [  
        {  
          id: 42,  
          title: 'Confusion about Flux and Relay'  
        },  
        {  
          id: 500,  
          title: 'Creating a Simple Application Using React JS : A Step-by-Step Guide'  
        }  
      ]  
    }  
  }  
}
```

下面列出几个要点：

- 分开存储 subreddit 信息，是为了缓存所有 subreddit。当用户来回切换 subreddit 时，可以立即更新，同时在不需要的时候可以不请求数据。不要担心把所有帖子放到内存中（会浪费内存）：除非你需要处理成千上万条帖子，而且用户通常不会关闭标签，你不需要做任何清理。
- 每个帖子的列表都需要使用 `isFetching` 来显示进度条，`didInvalidate` 来标记数据是否过期，`lastUpdated` 来存放数据

最后更新时间，还有 `items` 存放列表信息本身。在实际应用中，你还需要存放 `fetchedPageCount` 和 `nextPageUrl` 这样分页相关的 state。

嵌套内容须知

在这个示例中，接收到的列表和分页信息是存在一起的。但是，这种做法并不适用于有互相引用的嵌套内容的场景，或者用户可以编辑列表的场景。想像一下用户需要编辑一个接收到的帖子，但这个帖子在 state tree 的多个位置重复出现。这会让开发变得非常困难。

如果你有嵌套内容，或者用户可以编辑接收到的内容，你需要把它们分开存放在 state 中，就像数据库中一样。在分页信息中，只使用它们的 ID 来引用。这可以让你始终保持数据更新。[真实场景的案例](#) 中演示了这种做法，结合 `normalizr` 来把嵌套的 API 响应数据范式化，最终的 state 看起来是这样：

```
{  
  selected subreddit: 'frontend',  
  entities: {  
    users: {  
      2: {  
        id: 2,  
        name: 'Andrew'  
      }  
    },  
    posts: {  
      42: {  
        id: 42,  
        title: 'Confusion about Flux and Relay',  
        author: 2  
      },  
      100: {  
        id: 100,  
        title: 'Creating a Simple Application Using React',  
        author: 2  
      }  
    }  
  }  
}
```

```

},
postsBySubreddit: {
  frontend: {
    isFetching: true,
    didInvalidate: false,
    items: []
  },
  reactjs: {
    isFetching: false,
    didInvalidate: false,
    lastUpdated: 1439478405547,
    items: [ 42, 100 ]
  }
}
}

```

在本教程中，我们不会对内容进行范式化，但是在一个复杂些的应用中你可能需要使用。

处理 Action

在讲 dispatch action 与网络请求结合使用细节前，我们为上面定义的 action 开发一些 reducer。

Reducer 组合须知

这里，我们假设你已经学习过 [combineReducers\(\)](#) 并理解 reducer 组合，还有 [基础章节](#) 中的 [拆分 Reducer](#)。如果还没有，请[先学习](#)。

reducers.js

```

import { combineReducers } from 'redux'
import {
  SELECT_SUBREDDIT, INVALIDATE_SUBREDDIT,
  REQUEST_POSTS, RECEIVE_POSTS
} from '../actions'

```

```
function selected subreddit(state = 'reactjs', action) {  
  switch (action.type) {  
    case SELECT_SUBREDDIT:  
      return action.subreddit  
    default:  
      return state  
  }  
}  
  
function posts(state = {  
  isFetching: false,  
  didInvalidate: false,  
  items: []  
, action) {  
  switch (action.type) {  
    case INVALIDATE_SUBREDDIT:  
      return Object.assign({}, state, {  
        didInvalidate: true  
      })  
    case REQUEST_POSTS:  
      return Object.assign({}, state, {  
        isFetching: true,  
        didInvalidate: false  
      })  
    case RECEIVE_POSTS:  
      return Object.assign({}, state, {  
        isFetching: false,  
        didInvalidate: false,  
        items: action.posts,  
        lastUpdated: action.receivedAt  
      })  
    default:  
      return state  
  }  
}  
  
function postsBySubreddit(state = {}, action) {  
  switch (action.type) {  
    case INVALIDATE_SUBREDDIT:  
    case RECEIVE_POSTS:  
    case REQUEST_POSTS:  
      return Object.assign({}, state, {
```

```
[action.subreddit]: posts(state[action.subreddit], action)
})
default:
    return state
}
}

const rootReducer = combineReducers({
    postsBySubreddit,
    selected subreddit
})

export default rootReducer
```

上面代码有两个有趣的点：

- 使用 ES6 计算属性语法，使用 `Object.assign()` 来简洁高效地更新 `state[action.subreddit]`。这个：

```
return Object.assign({}, state, {
    [action.subreddit]: posts(state[action.subreddit], action)
})
```

与下面代码等价：

```
let nextState = {}
nextState[action.subreddit] = posts(state[action.subreddit],
return Object.assign({}, state, nextState)
```

- 我们提取出 `posts(state, action)` 来管理指定帖子列表的 state。这仅仅使用 [reducer 组合](#)而已！我们还可以借此机会把 reducer 分拆成更小的 reducer，这种情况下，我们把对象内列表的更新代理到了 `posts` reducer 上。在[真实场景的案例](#)中甚至更进一步，里面介绍了如何做一个

reducer 工厂来生成参数化的分页 reducer。

记住 reducer 只是函数而已，所以你可以尽情使用函数组合和高阶函数这些特性。

异步 Action Creator

最后，如何把[之前定义](#)的同步 action creator 和 网络请求结合起来呢？标准的做法是使用 [Redux Thunk middleware](#)。要引入 `redux-thunk` 这个专门的库才能使用。我们[后面会介绍](#) middleware 大体上是如何工作的；目前，你只需要知道一个要点：通过使用指定的 middleware，action creator 除了返回 action 对象外还可以返回函数。这时，这个 action creator 就成为了 [thunk](#)。

当 action creator 返回函数时，这个函数会被 Redux Thunk middleware 执行。这个函数并不需要保持纯净；它还可以带有副作用，包括执行异步 API 请求。这个函数还可以 dispatch action，就像 dispatch 前面定义的同步 action 一样。

我们仍可以在 `actions.js` 里定义这些特殊的 thunk action creator。

actions.js

```
import fetch from 'isomorphic-fetch'

export const REQUEST_POSTS = 'REQUEST_POSTS'
function requestPosts(subreddit) {
  return {
    type: REQUEST_POSTS,
    subreddit
  }
}

export const RECEIVE_POSTS = 'RECEIVE_POSTS'
function receivePosts(subreddit, json) {
  return {
```

```
type: RECEIVE_POSTS,
subreddit,
posts: json.data.children.map(child => child.data),
receivedAt: Date.now()
}

}

// 来看一下我们写的第一个 thunk action creator!
// 虽然内部操作不同，你可以像其它 action creator 一样使用它：
// store.dispatch(fetchPosts('reactjs'))

export function fetchPosts(subreddit) {

    // Thunk middleware 知道如何处理函数。
    // 这里把 dispatch 方法通过参数的形式传给函数，
    // 以此来让它自己也能 dispatch action。

    return function (dispatch) {

        // 首次 dispatch: 更新应用的 state 来通知
        // API 请求发起了。

        dispatch(requestPosts(subreddit))

        // thunk middleware 调用的函数可以有返回值，
        // 它会被当作 dispatch 方法的返回值传递。

        // 这个案例中，我们返回一个等待处理的 promise。
        // 这并不是 redux middleware 所必须的，但这对于我们而言很方便。

        return fetch(`http://www.reddit.com/r/${subreddit}.json`)
            .then(response => response.json())
            .then(json =>

                // 可以多次 dispatch!
                // 这里，使用 API 请求结果来更新应用的 state。

                dispatch(receivePosts(subreddit, json))
            )

            // 在实际应用中，还需要
            // 捕获网络请求的异常。
    }
}
```

```
    }  
}
```

fetch 使用须知

本示例使用了 `fetch API`。它是替代 `XMLHttpRequest` 用来发送网络请求的非常新的 API。由于目前大多数浏览器原生还不支持它，建议你使用 `isomorphic-fetch` 库：

```
// 每次使用 `fetch` 前都这样调用一下  
import fetch from 'isomorphic-fetch'
```

在底层，它在浏览器端使用 `whatwg-fetch polyfill`，在服务器端使用 `node-fetch`，所以如果当你把应用改成同构时，并不需要改变 API 请求。

注意，`fetch polyfill` 假设你已经使用了 `Promise` 的 polyfill。确保你使用 `Promise polyfill` 的一个最简单的办法是在所有应用代码前启用 Babel 的 ES6 polyfill：

```
// 在应用中其它任何代码执行前调用一次  
import 'babel-polyfill'
```

我们是如何在 `dispatch` 机制中引入 Redux Thunk middleware 的呢？我们使用了 `applyMiddleware()`，如下：

index.js

```
import thunkMiddleware from 'redux-thunk'  
import createLogger from 'redux-logger'  
import { createStore, applyMiddleware } from 'redux'  
import { selectSubreddit, fetchPosts } from './actions'  
import rootReducer from './reducers'
```

```
const loggerMiddleware = createLogger()

const createStoreWithMiddleware = applyMiddleware(
  thunkMiddleware, // 允许我们 dispatch() 函数
  loggerMiddleware // 一个很便捷的 middleware, 用来打印 action 日志
)(createStore)

const store = createStoreWithMiddleware(rootReducer)

store.dispatch(selectSubreddit('reactjs'))
store.dispatch(fetchPosts('reactjs')).then(() =>
  console.log(store.getState())
)
```

thunk 的一个优点是它的结果可以再次被 dispatch:

actions.js

```
import fetch from 'isomorphic-fetch'

export const REQUEST_POSTS = 'REQUEST_POSTS'
function requestPosts(subreddit) {
  return {
    type: REQUEST_POSTS,
    subreddit
  }
}

export const RECEIVE_POSTS = 'RECEIVE_POSTS'
function receivePosts(subreddit, json) {
  return {
    type: RECEIVE_POSTS,
    subreddit,
    posts: json.data.children.map(child => child.data),
    receivedAt: Date.now()
  }
}
```

```
function fetchPosts(subreddit) {
  return dispatch => {
    dispatch(requestPosts(subreddit))
    return fetch(`http://www.reddit.com/r/${subreddit}.json`)
      .then(response => response.json())
      .then(json => dispatch(receivePosts(subreddit, json)))
  }
}

function shouldFetchPosts(state, subreddit) {
  const posts = state.postsBySubreddit[subreddit]
  if (!posts) {
    return true
  } else if (posts.isFetching) {
    return false
  } else {
    return posts.didInvalidate
  }
}

export function fetchPostsIfNeeded(subreddit) {

  // 注意这个函数也接收了 getState() 方法
  // 它让你选择接下来 dispatch 什么。

  // 当缓存的值是可用时,
  // 减少网络请求很有用。

  return (dispatch, getState) => {
    if (shouldFetchPosts(getState(), subreddit)) {
      // 在 thunk 里 dispatch 另一个 thunk!
      return dispatch(fetchPosts(subreddit))
    } else {
      // 告诉调用代码不需要再等待。
      return Promise.resolve()
    }
  }
}
```

这可以让我们逐步开发复杂的异步控制流，同时保持代码整洁如初：

index.js

```
store.dispatch(fetchPostsIfNeeded('reactjs')).then(() =>
  console.log(store.getState())
)
```

服务端渲染须知

异步 action creator 对于做服务端渲染非常方便。你可以创建一个 store, dispatch 一个异步 action creator, 这个 action creator 又 dispatch 另一个异步 action creator 来为应用的一整块请求数据, 同时在 Promise 完成和结束时才 render 界面。然后在 render 前, store 里就已经存在了需要用的 state。

[Thunk middleware](#) 并不是 Redux 处理异步 action 的唯一方式。你也可以使用 [redux-promise](#) 或者 [redux-promise-middleware](#) 来 dispatch Promise 替代函数。你也可以使用 [redux-rx](#) dispatch Observable。你甚至可以写一个自定义的 middleware 来描述 API 请求, 就像这个[真实场景的案例](#)中的做法一样。你也可以先尝试一些不同做法, 选择喜欢的, 并使用下去, 不论有没有使用到 middleware 都行。

连接到 UI

Dispatch 同步 action 与异步 action 间并没有区别, 所以就不展开讨论细节了。参照 [搭配 React](#) 获得 React 组件中使用 Redux 的介绍。参照 [示例: subreddit API](#) 来获取本例的完整代码。

下一步

阅读 [异步数据流](#) 来整理一下 异步 action 是如何适用于 Redux 数据流的。

异步数据流

如果不使用 [middleware](#) 的话，Redux 的 store 只支持 [同步数据流](#)。而这也是 [createStore\(\)](#) 所默认提供的创建方式。

你可以使用 [applyMiddleware\(\)](#) 来增强 [createStore\(\)](#)。这不是必须的，但它可以让你 [以更简便的方式来描述异步的 action](#)。

像 [redux-thunk](#) 或 [redux-promise](#) 这样支持异步的 middleware 都包装了 store 的 [dispatch\(\)](#) 方法，以此来让你 dispatch 一些除了 action 以外的其他内容，例如：函数或者 Promise。你所使用的任何 middleware 都可以以自己的方式解释你 dispatch 的任何内容，并继续传递 actions 给下一个 middleware。比如，一个 Promise middleware 能够 Promise，然后针对每个 Promise 异步地 dispatch 一对 begin/end actions。

当 middleware 链中的最后一个 middleware dispatch action 时，这个 action 必须是一个普通对象。这是 [同步式的 Redux 数据流](#) 开始的地方（译注：这里应该是指，你可以使用任意多异步的 middleware 去做你想做的事情，但是需要使用普通对象作为最后一个被 dispatch 的 action，来将处理流程带回同步方式）。

接着可以查看 [异步示例的完整源码](#)。

下一步

现在你对 middleware 在 Redux 中作用的例子有了初步了解，是时候应用到实际开发中了。继续阅读关于 [Middleware](#) 的详细章节。

Middleware

我们已经在[异步 Action](#)一节的示例中看到了一些 middleware 的使用。如果你使用过 Express 或者 Koa 等服务端框架, 那么应该对 *middleware* 的概念不会陌生。在这类框架中, middleware 是指可以被嵌入在框架接收请求到产生响应过程之中的代码。例如, Express 或者 Koa 的 middleware 可以完成添加 CORS headers、记录日志、内容压缩等工作。middleware 最优秀的特性就是可以被链式组合。你可以在一个项目中使用多个独立的第三方 middleware。

相对于 Express 或者 Koa 的 middleware, Redux middleware 被用于解决不同的问题, 但其中的概念是类似的。它提供的是位于 **action 被发起之后, 到达 reducer 之前的扩展点**。你可以利用 Redux middleware 来进行日志记录、创建崩溃报告、调用异步接口或者路由等等。

这个章节分为两个部分, 前面是帮助你理解相关概念的深度介绍, 而后半部分则通过[一些实例](#)来体现 middleware 的强大能力。对文章前后内容进行结合通读, 会帮助你更好的理解枯燥的概念, 并从中获得启发。

理解 Middleware

正因为 middleware 可以完成包括异步 API 调用在内的各种事情, 了解它的演化过程是一件相当重要的事。我们将以记录日志和创建崩溃报告为例, 引导你体会从分析问题到通过构建 middleware 解决问题的思维过程。

问题: 记录日志

使用 Redux 的一个益处就是它让 state 的变化过程变的可预知和透明。每当一个 action 发起完成后, 新的 state 就会被计算并保存下来。State 不能被自身修改, 只能由特定的 action 引起变化。

试想一下, 当我们的应用中每一个 action 被发起以及每次新的 state 被计算

完成时都将它们记录下来，岂不是很好？当程序出现问题时，我们可以通过查阅日志找出是哪个 action 导致了 state 不正确。

```
▼ ADD_TODO
  i dispatching: Object {type: "ADD_TODO", text: "Use Redux"}
    next state: ► Object {visibilityFilter: "SHOW_ALL", todos: Array[1]}
▼ ADD_TODO
  i dispatching: Object {type: "ADD_TODO", text: "Learn about middleware"}
    next state: ► Object {visibilityFilter: "SHOW_ALL", todos: Array[2]}
▼ COMPLETE_TODO
  i dispatching: Object {type: "COMPLETE_TODO", index: 0}
    next state: ► Object {visibilityFilter: "SHOW_ALL", todos: Array[2]}
▼ SET_VISIBILITY_FILTER
  i dispatching: Object {type: "SET_VISIBILITY_FILTER", filter: "SHOW_COMPLETED"}
    next state: ► Object {visibilityFilter: "SHOW_COMPLETED", todos: Array[2]}
```

我们如何通过 Redux 实现它呢？

尝试 #1：手动记录

最直接的解决方案就是在每次调用 `store.dispatch(action)` 前后手动记录被发起的 action 和新的 state。这称不上一个真正的解决方案，仅仅是我们理解这个问题的第一步。

注意

如果你使用 `react-redux` 或者类似的绑定库，最好不要直接在你的组件中操作 `store` 的实例。在接下来的内容中，仅仅是假设你会通过 `store` 显式地向下传递。

假设，你在创建一个 Todo 时这样调用：

```
store.dispatch(addTodo('Use Redux'))
```

为了记录这个 action 以及产生的新的 state，你可以通过这种方式记录日志：

```
let action = addTodo('Use Redux')
```

```
console.log('dispatching', action)
store.dispatch(action)
console.log('next state', store.getState())
```

虽然这样做达到了想要的效果，但是你并不想每次都这么干。

尝试 #2: 封装 Dispatch

你可以将上面的操作抽取成一个函数：

```
function dispatchAndLog(store, action) {
  console.log('dispatching', action)
  store.dispatch(action)
  console.log('next state', store.getState())
}
```

然后用它替换 `store.dispatch()`：

```
dispatchAndLog(store, addTodo('Use Redux'))
```

你可以选择到此为止，但是每次都要导入一个外部方法总归还是不太方便。

尝试 #3: Monkeypatching Dispatch

如果我们直接替换 `store` 实例中的 `dispatch` 函数会怎么样呢？Redux store 只是一个包含[一些方法](#)的普通对象，同时我们使用的是 JavaScript，因此我们可以这样实现 `dispatch` 的 monkeypatch：

```
let next = store.dispatch
store.dispatch = function dispatchAndLog(action) {
  console.log('dispatching', action)
  let result = next(action)
  console.log('next state', store.getState())
  return result
```

```
}
```

这离我们想要的已经非常接近了！无论我们在哪里发起 action，保证都会被记录。Monkeypatching 令人感觉还是不太舒服，不过利用它我们做到了我们想要的。

问题：崩溃报告

如果我们想对 `dispatch` 附加超过一个的变换，又会怎么样呢？

我脑海中出现的另一个常用的变换就是在生产过程中报告 JavaScript 的错误。全局的 `window.onerror` 并不可靠，因为它在一些旧的浏览器中无法提供错误堆栈，而这是排查错误所需的至关重要信息。

试想当发起一个 action 的结果是一个异常时，我们将包含调用堆栈，引起错误的 action 以及当前的 state 等错误信息通通发到类似于 [Sentry](#) 这样的报告服务中，不是很好吗？这样我们可以更容易地在开发环境中重现这个错误。

然而，将日志记录和崩溃报告分离是很重要的。理想情况下，我们希望他们是两个不同的模块，也可能在不同的包中。否则我们无法构建一个由这些工具组成的生态系统。（提示：我们正在慢慢了解 middleware 的本质到底是什么！）

如果按照我们的想法，日志记录和崩溃报告属于不同的模块，他们看起来应该像这样：

```
function patchStoreToAddLogging(store) {
  let next = store.dispatch
  store.dispatch = function dispatchAndLog(action) {
    console.log('dispatching', action)
    let result = next(action)
    console.log('next state', store.getState())
    return result
  }
}
```

```

function patchStoreToAddCrashReporting(store) {
  let next = store.dispatch
  store.dispatch = function dispatchAndReportErrors(action) {
    try {
      return next(action)
    } catch (err) {
      console.error('捕获一个异常!', err)
      Raven.captureException(err, {
        extra: {
          action,
          state: store.getState()
        }
      })
      throw err
    }
  }
}

```

如果这些功能以不同的模块发布，我们可以在 store 中像这样使用它们：

```

patchStoreToAddLogging(store)
patchStoreToAddCrashReporting(store)

```

尽管如此，这种方式看起来还是不够令人满意。

尝试 #4: 隐藏 Monkeypatching

Monkeypatching 本质上是一种 hack。“将任意的方法替换成你想要的”，此时的 API 会是什么样的呢？现在，让我们来看看这种替换的本质。在此之前，我们用自己的函数替换掉了 `store.dispatch`。如果我们不这样做，而是在函数中返回新的 `dispatch` 呢？

```

function logger(store) {
  let next = store.dispatch

```

```
// 我们之前的做法:  
// store.dispatch = function dispatchAndLog(action) {  
  
  return function dispatchAndLog(action) {  
    console.log('dispatching', action)  
    let result = next(action)  
    console.log('next state', store.getState())  
    return result  
  }  
}  
}
```

我们可以在 Redux 内部提供一个可以将实际的 monkeypatching 应用到 `store.dispatch` 中的辅助方法：

```
function applyMiddlewareByMonkeypatching(store, middlewares) {  
  middlewares = middlewares.slice()  
  middlewares.reverse()  
  
  // 在每一个 middleware 中变换 dispatch 方法。  
  middlewares.forEach(middleware =>  
    store.dispatch = middleware(store)  
  )  
}
```

然后像这样应用多个 middleware：

```
applyMiddlewareByMonkeypatching(store, [ logger, crashReporter ])
```

尽管我们做了很多，实现方式依旧是 monkeypatching。
因为我们仅仅是将它隐藏在我们的框架内部，并没有改变这个事实。

尝试 #5：移除 Monkeypatching

为什么我们要替换原来的 `dispatch` 呢？当然，这样我们就可以在后面直接使用 `Middleware` 了。

调用它，但是还有另一个原因：就是每一个 middleware 都可以操作（或者直接调用）前一个 middleware 包装过的 `store.dispatch`：

```
function logger(store) {
  // 这里的 next 必须指向前一个 middleware 返回的函数:
  let next = store.dispatch

  return function dispatchAndLog(action) {
    console.log('dispatching', action)
    let result = next(action)
    console.log('next state', store.getState())
    return result
  }
}
```

将 middleware 串联起来的必要性是显而易见的。

如果 `applyMiddlewareByMonkeypatching` 方法中没有在第一个 middleware 执行时立即替换掉 `store.dispatch`，那么 `store.dispatch` 将会一直指向原始的 `dispatch` 方法。也就是说，第二个 middleware 依旧会作用在原始的 `dispatch` 方法。

但是，还有另一种方式来实现这种链式调用的效果。可以让 middleware 以方法参数的形式接收一个 `next()` 方法，而不是通过 `store` 的实例去获取。

```
function logger(store) {
  return function wrapDispatchToAddLogging(next) {
    return function dispatchAndLog(action) {
      console.log('dispatching', action)
      let result = next(action)
      console.log('next state', store.getState())
      return result
    }
  }
}
```

现在是“[我们该更进一步](#)”的时刻了，所以可能会多花一点时间来让它变的更为合理一些。这些串联函数很吓人。ES6 的箭头函数可以使其 [柯里化](#)，从而看起来更舒服一些：

```
const logger = store => next => action => {
  console.log('dispatching', action)
  let result = next(action)
  console.log('next state', store.getState())
  return result
}

const crashReporter = store => next => action => {
  try {
    return next(action)
  } catch (err) {
    console.error('Caught an exception!', err)
    Raven.captureException(err, {
      extra: {
        action,
        state: store.getState()
      }
    })
    throw err
  }
}
```

这正是 **Redux middleware** 的样子。

Middleware 接收了一个 `next()` 的 `dispatch` 函数，并返回一个 `dispatch` 函数，返回的函数会被作为下一个 middleware 的 `next()`，以此类推。由于 `store` 中类似 `getState()` 的方法依旧非常有用，我们将 `store` 作为顶层的参数，使得它可以在所有 middleware 中被使用。

尝试 #6：“单纯”地使用 Middleware

我们可以写一个 `applyMiddleware()` 方法替换掉原来的 `applyMiddlewareByMonkeypatching()`。在新的 `applyMiddleware()` 中，

我们取得最终完整的被包装过的 `dispatch()` 函数，并返回一个 `store` 的副本：

```
// 警告：这只是一个“单纯”的实现方式！
// 这 *并不是* Redux 的 API.

function applyMiddleware(store, middlewares) {
  middlewares = middlewares.slice()
  middlewares.reverse()

  let dispatch = store.dispatch
  middlewares.forEach(middleware =>
    dispatch = middleware(store)(dispatch)
  )

  return Object.assign({}, store, { dispatch })
}
```

这与 Redux 中 `applyMiddleware()` 的实现已经很接近了，但是有三个重要的不同之处：

- 它只暴露一个 `store API` 的子集给 `middleware`: `dispatch(action)` 和 `getState()`。
- 它用了一个非常巧妙的方式来保证你的 `middleware` 调用的是 `store.dispatch(action)` 而不是 `next(action)`，从而使这个 `action` 会在包括当前 `middleware` 在内的整个 `middleware` 链中被正确的传递。这对异步的 `middleware` 非常有用，正如我们在[之前的章节](#)中提到的。
- 为了保证你只能应用 `middleware` 一次，它作用在 `createStore()` 上而不是 `store` 本身。因此它的签名不是 `(store, middlewares) => store`，而是 `(...middlewares) => (createStore) => createStore`。

最终的方法

这是我们刚刚所写的 middleware：

```
const logger = store => next => action => {
  console.log('dispatching', action)
  let result = next(action)
  console.log('next state', store.getState())
  return result
}

const crashReporter = store => next => action => {
  try {
    return next(action)
  } catch (err) {
    console.error('Caught an exception!', err)
    Raven.captureException(err, {
      extra: {
        action,
        state: store.getState()
      }
    })
    throw err
  }
}
```

然后是将它们引用到 Redux store 中：

```
import { createStore, combineReducers, applyMiddleware } from 'redux'

// applyMiddleware 接收 createStore()
// 并返回一个包含兼容 API 的函数。
let createStoreWithMiddleware = applyMiddleware(logger, crashReporter)

// 像使用 createStore() 一样使用它。
let todoApp = combineReducers(reducers)
let store = createStoreWithMiddleware(todoApp)
```

就是这样！现在任何被发送到 store 的 action 都会经过 `logger` 和

```
crashReporter :
```

```
// 将经过 logger 和 crashReporter 两个 middleware!
store.dispatch(addTodo('Use Redux'))
```

7个示例

如果读完上面的章节你已经觉得头都要爆了，那就想象一下把它写出来之后的样子。下面的内容会让我们放松一下，并让你的思路延续。

下面的每个函数都是一个有效的 Redux middleware。它们不是同样有用，但是至少他们一样有趣。

```
/***
 * 记录所有被发起的 action 以及产生的新的 state。
 */
const logger = store => next => action => {
  console.group(action.type)
  console.info('dispatching', action)
  let result = next(action)
  console.log('next state', store.getState())
  console.groupEnd(action.type)
  return result
}

/***
 * 在 state 更新完成和 listener 被通知之后发送崩溃报告。
 */
const crashReporter = store => next => action => {
  try {
    return next(action)
  } catch (err) {
    console.error('Caught an exception!', err)
    Raven.captureException(err, {
      extra: {
        action,
        state: store.getState()
      }
    })
  }
}
```

```

        }
    })
    throw err
}
}

/***
 * 用 { meta: { delay: N } } 来让 action 延迟 N 毫秒。
 * 在这个案例中，让 `dispatch` 返回一个取消 timeout 的函数。
 */
const timeoutScheduler = store => next => action => {
    if (!action.meta || !action.meta.delay) {
        return next(action)
    }

    let timeoutId = setTimeout(
        () => next(action),
        action.meta.delay
    )

    return function cancel() {
        clearTimeout(timeoutId)
    }
}

/***
 * 通过 { meta: { raf: true } } 让 action 在一个 rAF 循环帧中被发起。
 * 在这个案例中，让 `dispatch` 返回一个从队列中移除该 action 的函数。
 */
const rafScheduler = store => next => {
    let queuedActions = []
    let frame = null

    function loop() {
        frame = null
        try {
            if (queuedActions.length) {
                next(queuedActions.shift())
            }
        } finally {
            maybeRaf()
        }
    }
}

```

```

}

function maybeRaf() {
  if (queuedActions.length && !frame) {
    frame = requestAnimationFrame(loop)
  }
}

return action => {
  if (!action.meta || !action.meta.raf) {
    return next(action)
  }

  queuedActions.push(action)
  maybeRaf()

  return function cancel() {
    queuedActions = queuedActions.filter(a => a !== action)
  }
}

/***
 * 使你除了 action 之外还可以发起 promise。
 * 如果这个 promise 被 resolved, 他的结果将被作为 action 发起。
 * 这个 promise 会被 `dispatch` 返回, 因此调用者可以处理 rejection。
 */
const vanillaPromise = store => next => action => {
  if (typeof action.then !== 'function') {
    return next(action)
  }

  return Promise.resolve(action).then(store.dispatch)
}

/***
 * 让你可以发起带有一个 { promise } 属性的特殊 action。
 *
 * 这个 middleware 会在开始时发起一个 action, 并在这个 `promise` reso
 *
 * 为了方便起见, `dispatch` 会返回这个 promise 让调用者可以等待。
 */

```

```

const readyStatePromise = store => next => action => {
  if (!action.promise) {
    return next(action)
  }

  function makeAction(ready, data) {
    let newAction = Object.assign({}, action, { ready }, data)
    delete newAction.promise
    return newAction
  }

  next(makeAction(false))
  return action.promise.then(
    result => next(makeAction(true, { result })),
    error => next(makeAction(true, { error }))
  )
}

/**
 * 让你可以发起一个函数来替代 action。
 * 这个函数接收 `dispatch` 和 `getState` 作为参数。
 *
 * 对于（根据 `getState()` 的情况）提前退出，或者异步控制流（`dispatch`）
 * `dispatch` 会返回被发起函数的返回值。
 */
const thunk = store => next => action =>
  typeof action === 'function' ?
    action(store.dispatch, store.getState) :
    next(action)

// 你可以使用以上全部的 middleware! (当然，这不意味着你必须全都使用。)
let createStoreWithMiddleware = applyMiddleware(
  rafScheduler,
  timeoutScheduler,
  thunk,
  vanillaPromise,
  readyStatePromise,
  logger,
  crashReporter
)(createStore)
let todoApp = combineReducers(reducers)

```

```
let store = createStoreWithMiddleware(todoApp)
```

Usage with React Router

Sorry, but we're still writing this doc.

Stay tuned, it will appear in a day or two.

示例：Reddit API

这是一个[高级教程](#)的例子，包含使用 Reddit API 请求文章标题的全部源码。

入口

index.js

```
import 'babel-core/polyfill'

import React from 'react'
import { render } from 'react-dom'
import Root from './containers/Root'

render(
  <Root />,
  document.getElementById('root')
)
```

Action Creators 和 Constants

actions.js

```
import fetch from 'isomorphic-fetch'

export const REQUEST_POSTS = 'REQUEST_POSTS'
export const RECEIVE_POSTS = 'RECEIVE_POSTS'
export const SELECT_SUBREDDIT = 'SELECT_SUBREDDIT'
export const INVALIDATE_SUBREDDIT = 'INVALIDATE_SUBREDDIT'

export function selectSubreddit(subreddit) {
  return {
    type: SELECT_SUBREDDIT,
    subreddit
}
```

```
}

export function invalidateSubreddit(subreddit) {
  return {
    type: INVALIDATE_SUBREDDIT ,
    subreddit
  }
}

function requestPosts(subreddit) {
  return {
    type: REQUEST_POSTS,
    subreddit
  }
}

function receivePosts(subreddit, json) {
  return {
    type: RECEIVE_POSTS,
    subreddit,
    posts: json.data.children.map(child => child.data),
    receivedAt: Date.now()
  }
}

function fetchPosts(subreddit) {
  return dispatch => {
    dispatch(requestPosts(subreddit))
    return fetch(`http://www.reddit.com/r/${subreddit}.json`)
      .then(response => response.json())
      .then(json => dispatch(receivePosts(subreddit, json)))
  }
}

function shouldFetchPosts(state, subreddit) {
  const posts = state.postsBySubreddit[subreddit]
  if (!posts) {
    return true
  } else if (posts.isFetching) {
    return false
  } else {
```

```
        return posts.didInvalidate
    }
}

export function fetchPostsIfNeeded(subreddit) {
    return (dispatch, getState) => {
        if (shouldFetchPosts(getState(), subreddit)) {
            return dispatch(fetchPosts(subreddit))
        }
    }
}
```

Reducers

reducers.js

```
import { combineReducers } from 'redux'
import {
    SELECT_SUBREDDIT, INVALIDATE_SUBREDDIT ,
    REQUEST_POSTS, RECEIVE_POSTS
} from './actions'

function selectedSubreddit(state = 'reactjs', action) {
    switch (action.type) {
        case SELECT_SUBREDDIT:
            return action.subreddit
        default:
            return state
    }
}

function posts(state = {
    isFetching: false,
    didInvalidate: false,
    items: []
}, action) {
    switch (action.type) {
        case INVALIDATE_SUBREDDIT :
```

```
        return Object.assign({}, state, {
          didInvalidate: true
        })
      case REQUEST_POSTS:
        return Object.assign({}, state, {
          isFetching: true,
          didInvalidate: false
        })
      case RECEIVE_POSTS:
        return Object.assign({}, state, {
          isFetching: false,
          didInvalidate: false,
          items: action.posts,
          lastUpdated: action.receivedAt
        })
      default:
        return state
    }
}

function postsBySubreddit(state = { }, action) {
  switch (action.type) {
    case INVALIDATE_SUBREDDIT :
    case RECEIVE_POSTS:
    case REQUEST_POSTS:
      return Object.assign({}, state, {
        [action.subreddit]: posts(state[action.subreddit], action)
      })
    default:
      return state
  }
}

const rootReducer = combineReducers({
  postsBySubreddit,
  selectedSubreddit
})

export default rootReducer
```

Store

configureStore.js

```
import { createStore, applyMiddleware } from 'redux'
import thunkMiddleware from 'redux-thunk'
import createLogger from 'redux-logger'
import rootReducer from './reducers'

const loggerMiddleware = createLogger()

const createStoreWithMiddleware = applyMiddleware(
  thunkMiddleware,
  loggerMiddleware
)(createStore)

export default function configureStore(initialState) {
  return createStoreWithMiddleware(rootReducer, initialState)
}
```

容器型组件

containers/Root.js

```
import React, { Component } from 'react'
import { Provider } from 'react-redux'
import configureStore from '../configureStore'
import AsyncApp from './AsyncApp'

const store = configureStore()

export default class Root extends Component {
  render() {
    return (
      <Provider store={store}>
        <AsyncApp />
      </Provider>
    )
  }
}
```

```
)  
}  
}
```

containers/AsyncApp.js

```
import React, { Component, PropTypes } from 'react'  
import { connect } from 'react-redux'  
import { selectSubreddit, fetchPostsIfNeeded, invalidateSubreddit } from 'actions'  
import Picker from '../components/Picker'  
import Posts from '../components/Posts'  
  
class AsyncApp extends Component {  
  constructor(props) {  
    super(props)  
    this.handleChange = this.handleChange.bind(this)  
    this.handleRefreshClick = this.handleRefreshClick.bind(this)  
  }  
  
  componentDidMount() {  
    const { dispatch, selectedSubreddit } = this.props  
    dispatch(fetchPostsIfNeeded(selectedSubreddit))  
  }  
  
  componentWillReceiveProps(nextProps) {  
    if (nextProps.selectedSubreddit !== this.props.selectedSubreddit) {  
      const { dispatch, selectedSubreddit } = nextProps  
      dispatch(fetchPostsIfNeeded(selectedSubreddit))  
    }  
  }  
  
  handleChange(nextSubreddit) {  
    this.props.dispatch(selectSubreddit(nextSubreddit))  
  }  
  
  handleRefreshClick(e) {  
    e.preventDefault()  
  
    const { dispatch, selectedSubreddit } = this.props  
    dispatch(invalidateSubreddit(selectedSubreddit))  
    dispatch(fetchPostsIfNeeded(selectedSubreddit))  
  }  
}
```

```
}

render () {
  const { selectedSubreddit, posts, isFetching, lastUpdated } = this.props
  return (
    <div>
      <Picker value={selectedSubreddit}
              onChange={this.handleChange}
              options={[ 'reactjs', 'frontend' ]} />
      <p>
        {lastUpdated &&
         <span>
           Last updated at {new Date(lastUpdated).toLocaleTimeString(
             ' ')}
         </span>
        }
        {!isFetching &&
         <a href='#'
             onClick={this.handleRefreshClick}>
           Refresh
         </a>
        }
      </p>
      {isFetching && posts.length === 0 &&
       <h2>Loading...</h2>
      }
      {!isFetching && posts.length === 0 &&
       <h2>Empty.</h2>
      }
      {posts.length > 0 &&
       <div style={{ opacity: isFetching ? 0.5 : 1 }}>
         <Posts posts={posts} />
       </div>
      }
    </div>
  )
}

AsyncApp.propTypes = {
  selectedSubreddit: PropTypes.string.isRequired,
  posts: PropTypes.array.isRequired,
```

```
    isFetching: PropTypes.bool.isRequired,
    lastUpdated: PropTypes.number,
    dispatch: PropTypes.func.isRequired
}

function mapStateToProps(state) {
  const { selectedSubreddit, postsBySubreddit } = state
  const {
    isFetching,
    lastUpdated,
    items: posts
  } = postsBySubreddit[selectedSubreddit] || {
    isFetching: true,
    items: []
  }

  return {
    selectedSubreddit,
    posts,
    isFetching,
    lastUpdated
  }
}

export default connect(mapStateToProps)(AsyncApp)
```

展示型组件

components/Picker.js

```
import React, { Component, PropTypes } from 'react'

export default class Picker extends Component {
  render() {
    const { value, onChange, options } = this.props

    return (
      <span>
```

```

        <h1>{value}</h1>
        <select onChange={e => onChange(e.target.value)}
            value={value}>
            {options.map(option =>
                <option value={option} key={option}>
                    {option}
                </option>
            })
        </select>
    </span>
)
}
}

Picker.propTypes = {
    options: PropTypes.arrayOf(
        PropTypes.string.isRequired
    ).isRequired,
    value: PropTypes.string.isRequired,
    onChange: PropTypes.func.isRequired
}

```

components/Posts.js

```

import React, { PropTypes, Component } from 'react'

export default class Posts extends Component {
    render() {
        return (
            <ul>
                {this.props.posts.map((post, i) =>
                    <li key={i}>{post.title}</li>
                )}
            </ul>
        )
    }
}

Posts.propTypes = {
    posts: PropTypes.arrayOf(PropTypes.shape({
        title: PropTypes.string.isRequired
    }))
}

```

```
}).isRequired).isRequired  
}
```

Next Steps

Sorry, but we're still writing this doc.
Stay tuned, it will appear in a day or two.

技巧

这一章是关于实现应用开发中会遇到的一些典型场景和代码片段。本章内容建立在你已经学会[基础章节](#)和[高级章节](#)的基础上。

- [迁移到 Redux](#)
- [减少样板代码](#)
- [服务端渲染](#)
- [编写测试](#)
- [计算衍生数据](#)
- [实现撤销重做](#)

迁移到 Redux

Redux 不是一个单一的框架，而是一系列的约定和一些让他们协同工作的函数。你的 Redux 项目的主体代码甚至不需要使用 Redux 的 API，大部分时间你其实是在编写函数。

这让到 Redux 的双向迁移都非常的容易。我们可不想把你限制得死死的！

从 Flux 项目迁移

Reducer 抓住了 Flux Store 的本质，因此，将一个 Flux 项目逐步到 Redux 是可行的，无论你使用了 Flummox、Alt、traditional Flux 还是其他 Flux 库。

同样你也可以将 Redux 的项目通过相同的步骤迁移到上述的这些 Flux 框架。

你的迁移过程大致包含几个步骤：

- 创建一个叫做 `createFluxStore(reducer)` 的函数，通过 `reducer` 函数适配你当前项目的 Flux Store。从代码来看，这个函数很像 Redux 中 `createStore` (来源) 的实现。它的 `dispatch` 处理器应该根据不同的 `action` 来调用不同的 `reducer`，保存新的 `state` 并抛出更新事件。
- 通过创建 `createFluxStore(reducer)` 的方法来将每个 Flux Store 逐步重写为 Reducer，这个过程中你的应用中其他部分代码感知不到任何变化，仍可以和原来一样使用 Flux Store。
- 当重写你的 Store 时，你会发现你应该避免一些明显违反 Flux 模式的使用方法，例如在 Store 中请求 API、或者在 Store 中触发 `action`。一旦基于 `reducer` 来构建你的 Flux 代码，它会变得更易于理解。
- 当你所有的 Flux Store 全部基于 `reducer` 来实现时，你就可以利用

`combineReducers(reducers)` 将多个 reducer 合并到一起，然后在应用里使用这个唯一的 Redux Store。

- 现在，剩下的就只是[使用 react-redux](#) 或者类似的库来处理你的UI部分。
- 最后，你可以使用一些 Redux 的特性，例如利用 middleware 来进一步简化异步的代码。

从 Backbone 项目迁移

Backbone 的 Model 层与 Redux 有着巨大的差别，因此，我们不建议从 Backbone 项目迁移到 Redux 。如果可以的话，最好的方法是彻底重写 app 的 Model 层。不过，如果重写不可行，也可以试试使用 [backbone-redux](#) 来逐步迁移，并使 Redux 的 store 和 Backbone 的 model 层及 collection 保持同步。

减少样板代码

Redux 很大部分 [受到 Flux 的启发](#)，而最常见的关于 Flux 的抱怨是必须写一大堆的模板。而在这章技巧中，根据个人样式，团队选项，长期可维护等等因素，Redux 可以让我们自由决定代码的繁复程度。

Actions

Actions 是用来描述在 app 中发生了什么的普通对象，是描述对象变异意图的唯一途径。很重要的一点是 **必须分发的 action 对象并非模板，而是 Redux 的一个基本设计选项**。

不少框架声称自己和 Flux 很像，只不过缺少了 action 对象的概念。但可预测的是，这是从 Flux 或 Redux 的倒退。如果没有可序列化的普通对象 action，便无法记录或重演用户会话，也无法实现 [带有时间旅行的热重载](#)。如果你更喜欢直接修改数据，那你并不需要使用 Redux。

Action 一般长这样：

```
{ type: 'ADD_TODO', text: 'Use Redux' }
{ type: 'REMOVE_TODO', id: 42 }
{ type: 'LOAD_ARTICLE', response: { ... } }
```

一个约定俗成的做法是，actions 拥有一个常量 type 帮助 reducer (或 Flux 中的 Stores) 识别它们。我们建议的你使用 string 而不是 [Symbols](#) 作为 action type，因为 string 是可序列化的，而使用 Symbols 会毫无必要地使记录和重演变得困难。

在 Flux 中，传统的想法是将每个 action type 定义为 string 常量：

```
const ADD_TODO = 'ADD_TODO';
const REMOVE_TODO = 'REMOVE_TODO';
```

```
const LOAD_ARTICLE = 'LOAD_ARTICLE';
```

这么做的优势？人们通常声称常量不是必要的。对于小项目也许正确。对于大的项目，将 action types 定义为常量有如下好处：

- 帮助维护命名一致性，因为所有的 action type 汇总在同一位置。
- 有时，在开发一个新功能之前你想看到所有现存的 actions 。而你的团队里可能已经有人添加了你所需要的action，而你并不知道。
- Action types 列表在 Pull Request 中能查到所有添加，删除，修改的记录。这能帮助团队中的所有人及时追踪新功能的范围与实现。
- 如果你在导入一个 Action 常量的时候拼写错误，你会得到 undefined 。当你纳闷 action 被分发出去而什么也没发生的时候，一个拼写错误更容易被发现。

你的项目的约定取决与你自己。开始时，可能用的是 inline string，之后转为常量，也许之后将他们归为一个独立文件。Redux 不会给予任何建议，选择你自己最喜欢的。

Action Creators

另一个约定俗成的做法，通过创建函数生成 action 对象，而不是在你分发的时候内联生成它们。

例如，比起使用对象文字调用 `dispatch`：

```
// somewhere in an event handler
dispatch({
  type: 'ADD_TODO',
  text: 'Use Redux'
});
```

你其实可以在单独的文件中写一个 action creator，然后从 component 里导入：

actionCreators.js

```
export function addTodo(text) {
  return {
    type: 'ADD_TODO',
    text
  };
}
```

AddTodo.js

```
import { addTodo } from './actionCreators';

// event handler 里的某处
dispatch(addTodo('Use Redux'))
```

Action creators 总被当作模板受到批评。好吧，其实你并不非得把他们写出来！**如果你觉得更适合你的项目，你可以选用对象文字** 然而，你应该知道写 action creators 是存在某种优势的。

假设有个设计师看完我们的原型之后回来说，我们最多只允许三个 todo 。我们可以使用 [redux-thunk](#) 中间件，并添加一个提前退出，把我们的 action creator 重写成回调形式：

```
function addTodoWithoutCheck(text) {
  return {
    type: 'ADD_TODO',
    text
  };
}

export function addTodo(text) {
  // Redux Thunk 中间件允许这种形式
  // 在下面的 “异步 Action Creators” 段落中有写
  return function (dispatch, getState) {
    if (getState().todos.length === 3) {
```

```
// 提前退出
return;
}

dispatch(addTodoWithoutCheck(text));
}

}
```

我们刚修改了 `addTodo` action creator 的行为，使得它对调用它的代码完全不可见。我们不用担心去每个添加 `todo` 的地方看一看，以确认他们有了这个检查 Action creator 让你可以解耦额外的分发 action 逻辑与实际发送这些 action 的 components。当你有大量开发工作且需求经常变更的时候，这种方法十分简便易用。

生成 Action Creators

某些框架如 [Flummox](#) 自动从 action creator 函数定义生成 action type 常量。这个想法是说你不需要同时定义 `ADD_TODO` 常量和 `addTodo()` action creator。这样的方法在底层也生成了 action type 常量，但他们是隐式生成的、间接级，会造成混乱。因此我们建议直接清晰地创建 action type 常量。

写简单的 action creator 很容易容易让人厌烦，且往往最终生成多余的样板代码：

```
export function addTodo(text) {
  return {
    type: 'ADD_TODO',
    text
  }
}

export function editTodo(id, text) {
  return {
    type: 'EDIT_TODO',
    id,
    text
  }
}
```

```

}

export function removeTodo(id) {
  return {
    type: 'REMOVE_TODO',
    id
  }
}

```

你可以写一个用于生成 action creator 的函数：

```

function makeActionCreator(type, ...argNames) {
  return function(...args) {
    let action = { type }
    argNames.forEach((arg, index) => {
      action[argNames[index]] = args[index]
    })
    return action
  }
}

const ADD_TODO = 'ADD_TODO'
const EDIT_TODO = 'EDIT_TODO'
const REMOVE_TODO = 'REMOVE_TODO'

export const addTodo = makeActionCreator(ADD_TODO, 'todo')
export const editTodo = makeActionCreator(EDIT_TODO, 'id', 'todo')
export const removeTodo = makeActionCreator(REMOVE_TODO, 'id')

```

一些工具库也可以帮助生成 action creator，例如 [redux-act](#) 和 [redux-actions](#)。这些库可以有效减少你的样板代码，并紧守例如 [Flux Standard Action \(FSA\)](#) 一类的标准。

异步 Action Creators

[中间件](#) 让你在每个 action 对象分发出去之前，注入一个自定义的逻辑来解释减少样板代码

你的 action 对象。异步 action 是中间件的最常见用例。

如果没有中间件，`dispatch` 只能接收一个普通对象。因此我们必须在 components 里面进行 AJAX 调用：

actionCreators.js

```
export function loadPostsSuccess(userId, response) {
  return {
    type: 'LOAD_POSTS_SUCCESS',
    userId,
    response
  };
}

export function loadPostsFailure(userId, error) {
  return {
    type: 'LOAD_POSTS_FAILURE',
    userId,
    error
  };
}

export function loadPostsRequest(userId) {
  return {
    type: 'LOAD_POSTS_REQUEST',
    userId
  };
}
```

UserInfo.js

```
import { Component } from 'react';
import { connect } from 'react-redux';
import { loadPostsRequest, loadPostsSuccess, loadPostsFailure }

class Posts extends Component {
  loadData(userId) {
    // 调用 React Redux `connect()` 注入 props :
```

```
let { dispatch, posts } = this.props;

if (posts[userId]) {
    // 这里是被缓存的数据！啥也不做。
    return;
}

// Reducer 可以通过设置 `isFetching` 反应这个 action
// 因此让我们显示一个 Spinner 控件。
dispatch(loadPostsRequest(userId));

// Reducer 可以通过填写 `users` 反应这些 actions
fetch(`http://myapi.com/users/${userId}/posts`).then(
    response => dispatch(loadPostsSuccess(userId, response)),
    error => dispatch(loadPostsFailure(userId, error))
);
}

componentDidMount() {
    this.loadData(this.props.userId);
}

componentWillReceiveProps(nextProps) {
    if (nextProps.userId !== this.props.userId) {
        this.loadData(nextProps.userId);
    }
}

render() {
    if (this.props.isLoading) {
        return <p>Loading...</p>;
    }

    let posts = this.props.posts.map(post =>
        <Post post={post} key={post.id} />
    );

    return <div>{posts}</div>;
}

export default connect(state => ({
```

```
    posts: state.posts
  })))(Posts);
```

然而，不久就需要再来一遍，因为不同的 components 从同样的 API 端点请求数据。而且我们想要在多个components 中重用一些逻辑（比如，当缓存数据有效的时候提前退出）。

中间件让我们能写表达更清晰的、潜在的异步 action creators。 它允许我们分发普通对象之外的东西，并且解释它们的值。比如，中间件能“捕捉”到已经分发的 Promises 并把他们变为一对请求和成功/失败的 action.

中间件最简单的例子是 [redux-thunk](#). “Thunk” 中间件让你可以把 **action creators** 写成 “thunks”，也就是返回函数的函数。这使得控制被反转了：你会像一个参数一样取得 `dispatch`，所以你也能写一个多次分发的 action creator 。

注意

Thunk 只是一个中间件的例子。中间件不仅仅是关于“分发函数”的：而是关于，你可以使用特定的中间件来分发任何该中间件可以处理的东西。例子中的 Thunk 中间件添加了一个特定的行为用来分发函数，但这实际取决于你用的中间件。

用 [redux-thunk](#) 上面的代码：

actionCreators.js

```
export function loadPosts(userId) {
  // 用 thunk 中间件解释:
  return function (dispatch, getState) {
    let { posts } = getState();
    if (posts[userId]) {
      // 这里是数据缓存！啥也不做。
      return;
    }
  }
}
```

```
dispatch({
  type: 'LOAD_POSTS_REQUEST',
  userId
});

// 异步分发原味 action
fetch(`http://myapi.com/users/${userId}/posts`).then(
  response => dispatch({
    type: 'LOAD_POSTS_SUCCESS',
    userId,
    response
  }),
  error => dispatch({
    type: 'LOAD_POSTS_FAILURE',
    userId,
    error
  })
);
}

}
```

UserInfo.js

```
import { Component } from 'react';
import { connect } from 'react-redux';
import { loadPosts } from './actionCreators';

class Posts extends Component {
  componentDidMount() {
    this.props.dispatch(loadPosts(this.props.userId));
  }

  componentWillReceiveProps(nextProps) {
    if (nextProps.userId !== this.props.userId) {
      this.props.dispatch(loadPosts(nextProps.userId));
    }
  }

  render() {
    if (this.props.isLoading) {
```

```
        return <p>Loading...</p>;
    }

    let posts = this.props.posts.map(post =>
      <Post post={post} key={post.id} />
    );

    return <div>{posts}</div>;
}
}

export default connect(state => ({
  posts: state.posts
}))(Posts);
```

这样打得字少多了！如果你喜欢，你还是可以保留“原味”action creators 比如从一个容器 `loadPosts` action creator 里用到的 `loadPostsSuccess`。

最后，你可以编写你自己的中间件 你可以把上面的模式泛化，然后代之以这样的异步 action creators：

```
export function loadPosts(userId) {
  return {
    // 要在之前和之后发送的 action types
    types: ['LOAD_POSTS_REQUEST', 'LOAD_POSTS_SUCCESS', 'LOAD_POSTS_ERROR'],
    // 检查缓存（可选）：
    shouldCallAPI: (state) => !state.users[userId],
    // 进行取：
    callAPI: () => fetch(`http://myapi.com/users/${userId}/posts`),
    // 在 actions 的开始和结束注入的参数
    payload: { userId }
  };
}
```

解释这个 actions 的中间件可以像这样：

```
function callAPIMiddleware({ dispatch, getState }) {
```

```
return function (next) {
  return function (action) {
    const {
      types,
      callAPI,
      shouldCallAPI = () => true,
      payload = {}
    } = action;

    if (!types) {
      // 普通 action: 传递
      return next(action);
    }

    if (
      !Array.isArray(types) ||
      types.length !== 3 ||
      !types.every(type => typeof type === 'string')
    ) {
      throw new Error('Expected an array of three string types');
    }

    if (typeof callAPI !== 'function') {
      throw new Error('Expected fetch to be a function.');
    }

    if (!shouldCallAPI(getState())) {
      return;
    }

    const [requestType, successType, failureType] = types;

    dispatch(Object.assign({}, payload, {
      type: requestType
}));;

    return callAPI().then(
      response => dispatch(Object.assign({}, payload, {
        response: response,
        type: successType
})),
      error => dispatch(Object.assign({}, payload, {
```

```
        error: error,
        type: failureType
    }))
);
};

}
}
```

在传给 `applyMiddleware(...middlewares)` 一次以后，你能用相同方式写你的 API-调用 action creators：

```
export function loadPosts(userId) {
  return {
    types: ['LOAD_POSTS_REQUEST', 'LOAD_POSTS_SUCCESS', 'LOAD_POSTS_FAILURE'],
    shouldCallAPI: (state) => !state.users[userId],
    callAPI: () => fetch(`http://myapi.com/users/${userId}/posts`),
    payload: { userId }
  };
}

export function loadComments(postId) {
  return {
    types: ['LOAD_COMMENTS_REQUEST', 'LOAD_COMMENTS_SUCCESS', 'LOAD_COMMENTS_FAILURE'],
    shouldCallAPI: (state) => !state.posts[postId],
    callAPI: () => fetch(`http://myapi.com/posts/${postId}/comments`),
    payload: { postId }
  };
}

export function addComment(postId, message) {
  return {
    types: ['ADD_COMMENT_REQUEST', 'ADD_COMMENT_SUCCESS', 'ADD_COMMENT_FAILURE'],
    callAPI: () => fetch(`http://myapi.com/posts/${postId}/comments`),
    method: 'post',
    headers: {
      'Accept': 'application/json',
      'Content-Type': 'application/json'
    },
    body: JSON.stringify({ message })
  };
}
```

```

    },
    payload: { postId, message }
  );
}

```

Reducers

Redux 用函数描述逻辑更新减少了模版里大量的 Flux stores 。函数比对象简单，比类更简单得多。

这个 Flux store:

```

let _todos = []

const TodoStore = Object.assign({}, EventEmitter.prototype, {
  getAll() {
    return _todos
  }
})

AppDispatcher.register(function (action) {
  switch (action.type) {
    case ActionTypes.ADD_TODO:
      let text = action.text.trim()
      _todos.push(text)
      TodoStore.emitChange()
  }
})

export default TodoStore

```

用了 Redux 之后，同样的逻辑更新可以被写成 reducing function:

```

export function todos(state = [], action) {
  switch (action.type) {
    case ActionTypes.ADD_TODO:

```

```

        let text = action.text.trim()
        return [ ...state, text ]
    default:
        return state
    }
}

```

`switch` 语句 不是 真正的模版。真正的 Flux 模版是概念性的：发送更新的需求，用 Dispatcher 注册 Store 的需求，Store 是对象的需求（当你想要一个哪都能跑的 App 的时候复杂度会提升）。

不幸的是很多人仍然靠文档里用没用 `switch` 来选择 Flux 框架。如果你不爱用 `switch` 你可以用一个单独的函数来解决，下面会演示。

生成 Reducers

写一个函数将 reducers 表达为 action types 到 handlers 的映射对象。例如，如果想在 `todos` reducer 里这样定义：

```

export const todos = createReducer([], {
    [ActionTypes.ADD_TODO](state, action) {
        let text = action.text.trim();
        return [...state, text];
    }
})

```

我们可以编写下面的辅助函数来完成：

```

function createReducer(initialState, handlers) {
    return function reducer(state = initialState, action) {
        if (handlers.hasOwnProperty(action.type)) {
            return handlers[action.type](state, action);
        } else {
            return state;
        }
    }
}

```

```
}
```

不难对吧？鉴于写法多种多样，Redux 没有默认提供这样的辅助函数。可能你想要自动地将普通 JS 对象变成 Immutable 对象，以填满服务器状态的对象数据。可能你想合并返回状态和当前状态。有多种多样的方法来“获取所有” handler，具体怎么做则取决于项目中你和你的团队的约定。

Redux reducer 的 API 是 `(state, action) => state`，但是怎么创建这些 reducers 由你来定。

服务端渲染

服务端渲染一个很常见的场景是当用户（或搜索引擎爬虫）第一次请求页面时，用它来做初始渲染。当服务器接收到请求后，它把需要的组件渲染成 HTML 字符串，然后把它返回给客户端（这里统指浏览器）。之后，客户端会接手渲染控制权。

下面我们使用 React 来做示例，对于支持服务端渲染的其它 view 框架，做法也是类似的。

服务端使用 Redux

当在服务器使用 Redux 渲染时，一定要在响应中包含应用的 state，这样客户端可以把它作为初始 state。这点至关重要，因为如果在生成 HTML 前预加载了数据，我们希望客户端也能访问这些数据。否则，客户端生成的 HTML 与服务器端返回的 HTML 就会不匹配，客户端还需要重新加载数据。

把数据发送到客户端，需要以下步骤：

- 为每次请求创建全新的 Redux store 实例；
- 按需 dispatch 一些 action；
- 从 store 中取出 state；
- 把 state 一同返回给客户端。

在客户端，使用服务器返回的 state 创建并初始化一个全新的 Redux store。Redux 在服务端唯一要做的事情就是，提供应用所需的**初始 state**。

安装

下面来介绍如何配置服务端渲染。使用极简的 [Counter 计数器应用](#) 来做示例，介绍如何根据请求在服务端提前渲染 state。

安装依赖库

本例会使用 [Express](#) 来做小型的 web 服务器。还需要安装 Redux 对 React 的绑定库，Redux 默认并不包含。

```
npm install --save express react-redux
```

服务端开发

下面是服务端代码大概的样子。使用 [app.use](#) 挂载 Express middleware 处理所有请求。如果你还不熟悉 Express 或者 middleware，只需要了解每次服务器收到请求时都会调用 handleRender 函数。

server.js

```
import path from 'path';
import Express from 'express';
import React from 'react';
import { createStore } from 'redux';
import { Provider } from 'react-redux';
import counterApp from './reducers';
import App from './containers/App';

const app = Express();
const port = 3000;

// 每当收到请求时都会触发
app.use(handleRender);

// 接下来会补充这部分代码
function handleRender(req, res) { /* ... */ }
function renderFullPage(html, initialState) { /* ... */ }

app.listen(port);
```

处理请求

第一件要做的事情就是对每个请求创建一个新的 Redux store 实例。这个 store 惟一作用是提供应用初始的 state。

渲染时，使用 `<Provider>` 来包住根组件 `<App />`，以此来让组件树中所有组件都能访问到 store，就像之前的搭配 React 教程讲的那样。

服务端渲染最关键的一步是在发送响应前渲染初始的 HTML。这就要使用 `React.renderToString()`。

然后使用 `store.getState()` 从 store 得到初始 state。`renderFullPage` 函数会介绍接下来如何传递。

```
import { renderToString } from 'react-dom/server'

function handleRender(req, res) {
  // 创建新的 Redux store 实例
  const store = createStore(counterApp);

  // 把组件渲染成字符串
  const html = renderToString(
    <Provider store={store}>
      <App />
    </Provider>
  )

  // 从 store 中获得初始 state
  const initialState = store.getState();

  // 把渲染后的页面内容发送给客户端
  res.send(renderFullPage(html, initialState));
}
```

注入初始组件的 HTML 和 State

服务端最后一步就是把初始组件的 HTML 和初始 state 注入到客户端能够渲

染的模板中。如何传递 state 呢，我们添加一个 `<script>` 标签来把 `initialState` 赋给 `window.__INITIAL_STATE__`。

客户端可以通过 `window.__INITIAL_STATE__` 获取 `initialState`。

同时使用 `script` 标签来引入打包后的 js bundle 文件。之前引入的 `serve-static middleware` 会处理它的请求。下面是代码。

```
function renderFullPage(html, initialState) {
  return `
    <!doctype html>
    <html>
      <head>
        <title>Redux Universal Example</title>
      </head>
      <body>
        <div id="root">${html}</div>
        <script>
          window.__INITIAL_STATE__ = ${JSON.stringify(initialState)}
        </script>
        <script src="/static/bundle.js"></script>
      </body>
    </html>
  `
}
```

字符串插值语法须知

上面的示例使用了 ES6 的[模板字符串语法](#)。它支持多行字符串和字符串插补特性，但需要支持 ES6。如果要在 Node 端使用 ES6，参考[Babel require hook](#) 文档。你也可以继续使用 ES5。

客户端开发

客户端代码非常直观。只需要从 `window.__INITIAL_STATE__` 得到初始 state，并传给 `createStore()` 函数即可。

代码如下：

client.js

```
import React from 'react'
import { render } from 'react-dom'
import { createStore } from 'redux'
import { Provider } from 'react-redux'
import App from './containers/App'
import counterApp from './reducers'

// 通过服务端注入的全局变量得到初始 state
const initialState = window.__INITIAL_STATE__

// 使用初始 state 创建 Redux store
const store = createStore(counterApp, initialState)

render(
  <Provider store={store}>
    <App />
  </Provider>,
  document.getElementById('root')
)
```

你可以选择自己喜欢的打包工具（Webpack, Browserify 或其它）来编译并打包文件到 dist/bundle.js。

当页面加载时，打包后的 js 会启动，并调用 `React.render()`，然后会与服务端渲染的 HTML 的 `data-react-id` 属性做关联。这会把新生成的 React 实例与服务端的虚拟 DOM 连接起来。因为同样使用了来自 Redux store 的初始 state，并且 view 组件代码是一样的，结果就是我们得到了相同的 DOM。

就是这样！这就是实现服务端渲染的所有步骤。

但这样做还是比较原始的。只会用动态代码渲染一个静态的 View。下一步要做的是动态创建初始 state 支持动态渲染 view。

准备初始 State

因为客户端只是执行收到的代码，刚开始的初始 state 可能是空的，然后根据需要获取 state。在服务端，渲染是同步执行的而且我们只有一次渲染 view 的机会。在收到请求时，可能需要根据请求参数或者外部 state（如访问 API 或者数据库），计算后得到初始 state。

处理 Request 参数

服务端收到的惟一输入是来自浏览器的请求。在服务器启动时可能需要做一些配置（如运行在开发环境还是生产环境），但这些配置是静态的。

请求会包含 URL 请求相关信息，包括请求参数，它们对于做 [React Router](#) 路由时可能会有用。也可能在请求头里包含 cookies，鉴权信息或者 POST 内容数据。下面演示如何基于请求参数来得到初始 state。

server.js

```
import qs from 'qs'; // 添加到文件开头
import { renderToString } from 'react-dom/server'

function handleRender(req, res) {
  // 如果存在的话，从 request 读取 counter
  const params = qs.parse(req.query)
  const counter = parseInt(params.counter) || 0

  // 得到初始 state
  let initialState = { counter }

  // 创建新的 Redux store 实例
  const store = createStore(counterApp, initialState)

  // 把组件渲染成字符串
  const html = renderToString(
    <Provider store={store}>
      <App />
    </Provider>
  )
  res.send(html)
}
```

```

)
// 从 Redux store 得到初始 state
const finalState = store.getState()

// 把渲染后的页面发给客户端
res.send(renderFullPage(html, finalState))
}

```

上面的代码首先访问 Express 的 `Request` 对象。把参数转成数字，然后设置到初始 `state` 中。如果你在浏览器中访问 <http://localhost:3000/?counter=100>，你会看到计数器从 100 开始。在渲染后的 HTML 中，你会看到计数显示 100 同时设置进了 `__INITIAL_STATE__` 变量。

获取异步 State

服务端渲染常用的场景是处理异步 `state`。因为服务端渲染天生是同步的，因此异步的数据获取操作对应到同步操作非常重要。

最简单的做法是往同步代码里传递一些回调函数。在这个回调函数里引用响应对象，把渲染后的 HTML 发给客户端。不要担心，并没有想像中那么难。

本例中，我们假设有一个外部数据源提供计算器的初始值（所谓的把计算作为一种服务）。我们会模拟一个请求并使用结果创建初始 `state`。API 请求代码如下：

api/counter.js

```

function getRandomInt(min, max) {
  return Math.floor(Math.random() * (max - min)) + min
}

export function fetchCounter(callback) {
  setTimeout(() => {
    callback(getRandomInt(1, 100))
  }, 500)
}

```

再次说明一下，这只是一个模拟的 API，我们使用 `setTimeout` 模拟一个需要 500 毫秒的请求（实现项目中 API 请求一般会更快）。传入一个回调函数，它异步返回一个随机数字。如果你使用了基于 Promise 的 API 工具，那么要把回调函数放到 `then` 中。

在服务端，把代码使用 `fetchCounter` 包起来，在回调函数里拿到结果：

server.js

```
// 添加到 import
import { fetchCounter } from './api/counter'
import { renderToString } from 'react-dom/server'

function handleRender(req, res) {
  // 异步请求模拟的 API
  fetchCounter(apiResult => {
    // 如果存在的话，从 request 读取 counter
    const params = qs.parse(req.query)
    const counter = parseInt(params.counter) || apiResult || 0

    // 得到初始 state
    let initialState = { counter }

    // 创建新的 Redux store 实例
    const store = createStore(counterApp, initialState)

    // 把组件渲染成字符串
    const html = renderToString(
      <Provider store={store}>
        <App />
      </Provider>
    )

    // 从 Redux store 得到初始 state
    const finalState = store.getState()

    // 把渲染后的页面发给客户端
    res.send(renderFullPage(html, finalState))
  })
}
```

```
});  
}
```

因为在回调中使用了 `res.send()`，服务器会保护连接打开并在回调函数执行前不发送任何数据。你会发现每个请求都有 500ms 的延时。更高级的用法会包括对 API 请求出错进行处理，比如错误的请求或者超时。

安全注意事项

因为我们代码中很多是基于用户生成内容（UGC）和输入的，不知不觉中，提高了应用可能受攻击区域。任何应用都应该对用户输入做安全处理以避免跨站脚本攻击（XSS）或者代码注入。

我们的示例中，只对安全做基本处理。当从请求中拿参数时，对 `counter` 参数使用 `parseInt` 把它转成数字。如果不这样做，当 `request` 中有 `script` 标签时，很容易在渲染的 HTML 中生成危险代码。就像这样的：`?counter=</script><script>doSomethingBad();</script>`

在我们极简的示例中，把输入转成数字已经比较安全。如果处理更复杂的输入，比如自定义格式的文本，你应该用安全函数处理输入，比如 [validator.js](#)。

此外，可能添加额外的安全层来对产生的 `state` 进行消毒。`JSON.stringify` 可能会造成 `script` 注入。鉴于此，你需要清洗 JSON 字符串中的 HTML 标签和其它危险的字符。可能通过字符串替换或者使用复杂的库如 [serialize-javascript](#) 处理。

下一步

你还可以参考 [异步 Actions](#) 学习更多使用 `Promise` 和 `thunk` 这些异步元素来表示异步数据流的方法。记住，那里学到的任何内容都可以用于同构渲染。

如果你使用了 [React Router](#)，你可能还需要在路由处理组件中使用静态的

`fetchData()` 方法来获取依赖的数据。它可能返回 异步 action，以便你的 `handleRender` 函数可以匹配到对应的组件类，对它们均 `dispatch` `fetchData()` 的结果，在 Promise 解决后才渲染。这样不同路由需要调用的 API 请求都并置于路由处理组件了。在客户端，你也可以使用同样技术来避免在切换页面时，当数据还没有加载完成前执行路由。

编写测试

因为你写的大部分 Redux 代码都是些函数，而且大部分是纯函数，所以很好测，不需要模拟。

设置

我们建议用 [Mocha](#) 作为测试引擎。

注意因为是在 node 环境下运行，所以你不能访问 DOM。

```
npm install --save-dev mocha
```

若想结合 [Babel](#) 使用，在 `package.json` 的 `scripts` 里加入这一段：

```
{
  ...
  "scripts": {
    ...
    "test": "mocha --compilers js:babel/register --recursive",
    "test:watch": "npm test -- --watch",
  },
  ...
}
```

然后运行 `npm test` 就能单次运行了，或者也可以使用 `npm run test:watch` 在每次有文件改变时自动执行测试。

Action Creators

Redux 里的 action creators 是会返回普通对象的函数。在测试 action creators 的时候我们想要测试不仅是调用了正确的 action creator，还有是否

返回了正确的 action。

示例

```
export function addTodo(text) {
  return {
    type: 'ADD_TODO',
    text
  };
}
```

可以这样测试：

```
import expect from 'expect';
import * as actions from '../actions/TodoActions';
import * as types from '../constants/ActionTypes';

describe('actions', () => {
  it('should create an action to add a todo', () => {
    const text = 'Finish docs';
    const expectedAction = {
      type: types.ADD_TODO,
      text
    };
    expect(actions.addTodo(text)).toEqual(expectedAction);
  });
})
```

异步 Action Creators

对于使用 [Redux Thunk](#) 或其它中间件的异步 action creator，最好完全模拟 Redux store 来测试。你可以使用 [applyMiddleware\(\)](#) 和一个模拟的 store，如下所示 (可在 [redux-mock-store](#) 中找到以下代码)。也可以使用 [nock](#) 来模拟 HTTP 请求。

示例

编写测试

147

```
function fetchTodosRequest() {
  return {
    type: FETCH_TODOS_REQUEST
  }
}

function fetchTodosSuccess(body) {
  return {
    type: FETCH_TODOS_SUCCESS,
    body
  }
}

function fetchTodosFailure(ex) {
  return {
    type: FETCH_TODOS_FAILURE,
    ex
  }
}

export function fetchTodos() {
  return dispatch => {
    dispatch(fetchTodosRequest())
    return fetch('http://example.com/todos')
      .then(res => res.json())
      .then(json => dispatch(fetchTodosSuccess(json.body)))
      .catch(ex => dispatch(fetchTodosFailure(ex)))
  }
}
```

可以这样测试:

```
import expect from 'expect'
import { applyMiddleware } from 'redux'
import thunk from 'redux-thunk'
import * as actions from '../actions/counter'
import * as types from '../constants/ActionTypes'
import nock from 'nock'
```

```
const middlewares = [ thunk ]  
  
/**  
 * 使用中间件模拟 Redux store。  
 */  
function mockStore(getState, expectedActions, done) {  
  if (!Array.isArray(expectedActions)) {  
    throw new Error('expectedActions should be an array of expected actions')  
  }  
  if (typeof done !== 'undefined' && typeof done !== 'function') {  
    throw new Error('done should either be undefined or a function')  
  }  
  
  function mockStoreWithoutMiddleware() {  
    return {  
      getState() {  
        return typeof getState === 'function' ?  
          getState() :  
          getState  
      },  
  
      dispatch(action) {  
        const expectedAction = expectedActions.shift()  
  
        try {  
          expect(action).toEqual(expectedAction)  
          if (done && !expectedActions.length) {  
            done()  
          }  
          return action  
        } catch (e) {  
          done(e)  
        }  
      }  
    }  
  }  
  
  const mockStoreWithMiddleware = applyMiddleware(  
    ...middlewares  
)  
  (mockStoreWithoutMiddleware)  
  
  return mockStoreWithMiddleware()
```

```
}

describe('async actions', () => {
  afterEach(() => {
    nock.cleanAll()
  })

  it('creates FETCH_TODOS_SUCCESS when fetching todos has been done', () => {
    nock('http://example.com/')
      .get('/todos')
      .reply(200, { todos: ['do something'] })

    const expectedActions = [
      { type: types.FETCH_TODOS_REQUEST },
      { type: types.FETCH_TODOS_SUCCESS, body: { todos: ['do something'] } }
    ]
    const store = mockStore({ todos: [] }, expectedActions, done)
    store.dispatch(actions.fetchTodos())
  })
})
```

Reducers

Reducer 把 action 应用到之前的 state，并返回新的 state。测试如下。

示例

```
import { ADD_TODO } from '../constants/ActionTypes'

const initialState = [
  {
    text: 'Use Redux',
    completed: false,
    id: 0
  }
]

export default function todos(state = initialState, action) {
```

```
switch (action.type) {
  case ADD_TODO:
    return [
      {
        id: state.reduce((maxId, todo) => Math.max(todo.id, maxId), -1),
        completed: false,
        text: action.text
      },
      ...state
    ]

  default:
    return state
}
}
```

可以这样测试：

```
import expect from 'expect'
import reducer from '../reducers/todos'
import * as types from '../constants/ActionTypes'

describe('todos reducer', () => {
  it('should return the initial state', () => {
    expect(
      reducer(undefined, {})
    ).toEqual([
      {
        text: 'Use Redux',
        completed: false,
        id: 0
      }
    ])
  })

  it('should handle ADD_TODO', () => {
    expect(
      reducer([], {
        type: types.ADD_TODO,
        text: 'Learn Redux'
      })
    ).toEqual([
      {
        text: 'Learn Redux',
        completed: false,
        id: 1
      }
    ])
  })
})
```

```
        text: 'Run the tests'
    })
).toEqual(
[
{
    text: 'Run the tests',
    completed: false,
    id: 0
}
])
)

expect(
reducer(
[
{
    text: 'Use Redux',
    completed: false,
    id: 0
}
],
{
    type: types.ADD_TODO,
    text: 'Run the tests'
}
)
).toEqual(
[
{
    text: 'Run the tests',
    completed: false,
    id: 1
},
{
    text: 'Use Redux',
    completed: false,
    id: 0
}
]
)
})
```

Components

React components 的优点是，一般都很小且依赖于 props。因此测试起来很简便。

首先，安装 [React Test Utilities](#):

```
npm install --save-dev react-addons-test-utils
```

要测 components，我们要创建一个叫 `setup()` 的辅助方法，用来把模拟过的（stubbed）回调函数当作 props 传入，然后使用 [React 浅渲染](#) 来渲染组件。这样就可以依据“是否调用了回调函数”的断言来写独立的测试。

示例

```
import React, { PropTypes, Component } from 'react'
import TodoTextInput from './TodoTextInput'

class Header extends Component {
  handleSave(text) {
    if (text.length !== 0) {
      this.props.addTodo(text)
    }
  }

  render() {
    return (
      <header className='header'>
        <h1>todos</h1>
        <TodoTextInput newTodo={true}
                      onSave={this.handleSave.bind(this)}
                      placeholder='What needs to be done?' />
      </header>
    )
  }
}
```

```
Header.propTypes = {
  addTodo: PropTypes.func.isRequired
}

export default Header
```

可以这样测试：

```
import expect from 'expect'
import React from 'react'
import TestUtils from 'react-addons-test-utils'
import Header from '../components/Header'
import TodoTextInput from '../components/TodoTextInput'

function setup() {
  let props = {
    addTodo: expect.createSpy()
  }

  let renderer = TestUtils.createRenderer()
  renderer.render(<Header {...props} />)
  let output = renderer.getRenderOutput()

  return {
    props,
    output,
    renderer
  }
}

describe('components', () => {
  describe('Header', () => {
    it('should render correctly', () => {
      const { output } = setup()

      expect(output.type).toBe('header')
      expect(output.props.className).toBe('header')
    })
  })
})
```

```
let [ h1, input ] = output.props.children

expect(h1.type).toBe('h1')
expect(h1.props.children).toBe('todos')

expect(input.type).toBe(TodoTextInput)
expect(input.props.newTodo).toBe(true)
expect(input.props.placeholder).toBe('What needs to be done?')

it('should call addTodo if length of text is greater than 0',
  () => {
    const { output, props } = setup()
    let input = output.props.children[1]
    input.props.onSave('')
    expect(props.addTodo.calls.length).toBe(0)
    input.props.onSave('Use Redux')
    expect(props.addTodo.calls.length).toBe(1)
  })
})
```

setState() 异常修复

浅渲染目前的问题是 `如果调用 setState 便抛异常`. React 貌似想要的是，如果想要使用 `setState`，DOM 就一定要存在（但测试运行在 node 环境下，是没有 DOM 的）。要解决这个问题，我们用了 jsdom，为了在 DOM 无效的时候，React 也不抛异常。按下面方法 `设置它`：

```
npm install --save-dev jsdom
```

然后，在测试目录中创建 `setup.js` 文件：

```
import { jsdom } from 'jsdom'

global.document = jsdom('
<!doctype html>
<html>
<body></body>
</html>')
global.window = document.defaultView
```

```
global.navigator = global.window.navigator
```

重要的一点，在引入 React 之前，这段代码应被评估。因此，要在 package.json 中加入 `--require ./test/setup.js`，以更改 mocha 命令。

```
{
  ...
  "scripts": {
    ...
    "test": "mocha --compilers js:babel/register --recursive --I",
  },
  ...
}
```

连接组件

如果你使用了 [React Redux](#)，可能你也同时在使用类似 `connect()` 的 [higher-order components](#)，将 Redux state 注入到常见的 React 组件中。

请看这个 `App` 组件：

```
import { connect } from 'react-redux'

class App extends Component { /* ... */ }

export default connect(mapStateToProps)(App)
```

在单元测试中，一般会这样导入 `App` 组件

```
import App from './App'
```

但是，当这样导入时，实际上持有的是 `connect()` 返回的包装过组件，而不是 `App` 组件本身。如果想测试它和 Redux 间的互动，好消息是可以使用一个专为单元测试创建的 `store`，将它包装在 `<Provider>` 中。但有时我们仅仅是想测试组件的渲染，并不想要这么一个 Redux `store`。

想要不和装饰件打交道而测试 `App` 组件本身，我们建议你同时导出未包装的组件：

```
import { connect } from 'react-redux'

// 命名导出未连接的组件（测试用）
export class App extends Component { /* ... */ }

// 默认导出已连接的组件（app 用）
export default connect(mapStateToProps)(App)
```

鉴于默认导出的依旧是包装过的组件，上面的导入语句会和之前一样工作，不需要更改应用中的代码。不过，可以这样在测试文件中导入没有包装的 `App` 组件：

```
// 注意花括号：抓取命名导出，而不是默认导出
import { App } from './App'
```

如果两者都需要：

```
import ConnectedApp, { App } from './App'
```

在 `app` 中，仍然正常地导入：

```
import App from './App'
```

只在测试中使用命名导出。

混用 ES6 模块和 CommonJS 的注意事项

如果在应用代码中使用 ES6，但在测试中使用 ES5，Babel 会通过其 `interop` 的机制处理 ES6 的 `import` 和 CommonJS 的 `require` 的转换，使这两个模块的格式各自运作，但其行为依旧有 [细微的区别](#)。如果在默认导出的附近增加另一个导出，将导致无法默认导出 `require('./App')`。此时，应代以 `require('./App').default`。

中间件

中间件函数会对 Redux 中 `dispatch` 的调用行为进行封装。因此，需要通过模拟 `dispatch` 的调用行为来测试。

示例

```
import expect from 'expect'
import * as types from '../constants/ActionTypes'
import singleDispatch from '../../../../../middleware/singleDispatch'

const createFakeStore = fakeData => ({
  getState() {
    return fakeData
  }
})

const dispatchWithStoreOf = (storeData, action) => {
  let dispatched = null
  const dispatch = singleDispatch(createFakeStore(storeData))(action)
  dispatch(action)
  return dispatched
};

describe('middleware', () => {
  it('should dispatch if store is empty', () => {
    const action = {
      type: types.ADD_TODO
    }
  })
});
```

```
expect(
  dispatchWithStoreOf({}, action)
).toEqual(action)
})

it('should not dispatch if store already has type', () => {
  const action = {
    type: types.ADD_TODO
  }

  expect(
    dispatchWithStoreOf({
      [types.ADD_TODO]: 'dispatched'
    }, action)
  ).toNotExist()
})
})
```

词汇表

- [React Test Utils](#): React 测试工具。
- [jsdom](#): 一个 JavaScript 的内建 DOM 。 Jsdom 不使用浏览器也能跑测试。
- [浅渲染 \(shallow renderer\)](#) :浅渲染的中心思想是， 初始化一个组件然后得到它的 `render` 方法作为结果， 渲染深度仅一层， 而非递归渲染整个 DOM 。浅渲染的结果是一个 [ReactElement](#)， 意味着我们可以访问它的 `children` 和 `props`， 且测试它本身是否工作正常。同时也意味着， 更改一个子组件不会影响到其父组件的测试。

计算衍生数据

[Reselect](#) 库可以创建可记忆的(Memoized)、可组合的 **selector** 函数。 Reselect selectors 可以用来高效地计算 Redux store 里的衍生数据。

可记忆的 Selectors 初衷

首先访问 [Todos 列表示例](#):

containers/App.js

```
import React, { Component, PropTypes } from 'react';
import { connect } from 'react-redux';
import { addTodo, completeTodo, setVisibilityFilter, VisibilityFilter } from '../actions';
import AddTodo from '../components/AddTodo';
import TodoList from '../components/TodoList';
import Footer from '../components/Footer';

class App extends Component {
  render() {
    // 通过 connect() 注入:
    const { dispatch, visibleTodos, visibilityFilter } = this.props;
    return (
      <div>
        <AddTodo
          onAddClick={text =>
            dispatch(addTodo(text))
          } />
        <TodoList
          todos={this.props.visibleTodos}
          onTodoClick={index =>
            dispatch(completeTodo(index))
          } />
        <Footer
          filter={visibilityFilter}
          onFilterChange={nextFilter =>
            dispatch(setVisibilityFilter(nextFilter))
          } />
      </div>
    );
  }
}

export default connect(state => ({ visibleTodos, visibilityFilter }))(App);
```

```
        } />
      </div>
    );
}
}

App.propTypes = {
  visibleTodos: PropTypes.arrayOf(PropTypes.shape({
    text: PropTypes.string.isRequired,
    completed: PropTypes.bool.isRequired
})),
  visibilityFilter: PropTypes.oneOf([
    'SHOW_ALL',
    'SHOW_COMPLETED',
    'SHOW_ACTIVE'
]).isRequired
};

function selectTodos(todos, filter) {
  switch (filter) {
  case VisibilityFilters.SHOW_ALL:
    return todos;
  case VisibilityFilters.SHOW_COMPLETED:
    return todos.filter(todo => todo.completed);
  case VisibilityFilters.SHOW_ACTIVE:
    return todos.filter(todo => !todo.completed);
  }
}

function select(state) {
  return {
    visibleTodos: selectTodos(state.todos, state.visibilityFilter),
    visibilityFilter: state.visibilityFilter
  };
}

// 打包组件，注入 dispatch 和 state
export default connect(select)(App);
```

上面的示例中，`select` 调用了 `selectTodos` 来计算 `visibleTodos`。运

行没问题，但有一个缺点：每当组件更新时都会重新计算 `visibleTodos`。如果 state tree 非常大，或者计算量非常大，每次更新都重新计算可能会带来性能问题。Reselect 能帮你省去这些没必要的重新计算。

创建可记忆的 Selector

我们需要一个可记忆的 selector 来替代这个 `select`，只在 `state.todos` or `state.visibilityFilter` 变化时重新计算 `visibleTodos`，而在其它部分（非相关）变化时不做计算。

Reselect 提供 `createSelector` 函数来创建可记忆的 selector。`createSelector` 接收一个 `input-selectors` 数组和一个转换函数作为参数。如果 state tree 的改变会引起 `input-selector` 值变化，那么 selector 会调用转换函数，传入 `input-selectors` 作为参数，并返回结果。如果 `input-selectors` 的值和前一次的一样，它将会直接返回前一次计算的数据，而不会再调用一次转换函数。

定义一个可记忆的 selector `visibleTodosSelector` 来替代 `select`：

`selectors/TodoSelectors.js`

```
import { createSelector } from 'reselect';
import { VisibilityFilters } from './actions';

function selectTodos(todos, filter) {
  switch (filter) {
    case VisibilityFilters.SHOW_ALL:
      return todos;
    case VisibilityFilters.SHOW_COMPLETED:
      return todos.filter(todo => todo.completed);
    case VisibilityFilters.SHOW_ACTIVE:
      return todos.filter(todo => !todo.completed);
  }
}

const visibilityFilterSelector = (state) => state.visibilityFilter;
const todosSelector = (state) => state.todos;
```

```

export const visibleTodosSelector = createSelector(
  [visibilityFilterSelector, todosSelector],
  (visibilityFilter, todos) => {
    return {
      visibleTodos: selectTodos(todos, visibilityFilter),
      visibilityFilter
    };
  }
);

```

在上例中，`visibilityFilterSelector` 和 `todosSelector` 是 input-selector。因为他们并不转换数据，所以被创建成普通的非记忆的 selector 函数。但是，`visibleTodosSelector` 是一个可记忆的 selector。他接收 `visibilityFilterSelector` 和 `todosSelector` 为 input-selector，还有一个转换函数来计算过滤的 todos 列表。

组合 Selector

可记忆的 selector 自身可以作为其它可记忆的 selector 的 input-selector。下面的 `visibleTodosSelector` 被当作另一个 selector 的 input-selector，来进一步通过关键字 (keyword) 过滤 todos。

```

const keywordSelector = (state) => state.keyword

const keywordFilterSelector = createSelector(
  [ visibleTodosSelector, keywordSelector ],
  (visibleTodos, keyword) => visibleTodos.filter(
    todo => todo.indexOf(keyword) > -1
  )
)

```

连接 Selector 和 Redux Store

如果你在使用 react-redux，你可以使用 connect 来连接可记忆的 selector 和计算衍生数据

Redux store。

containers/App.js

```
import React, { Component, PropTypes } from 'react'
import { connect } from 'react-redux'
import { addTodo, completeTodo, setVisibilityFilter } from '../actions'
import AddTodo from '../components/AddTodo'
import TodoList from '../components/TodoList'
import Footer from '../components/Footer'
import { visibleTodosSelector } from '../selectors/todoSelectors'

class App extends Component {
  render() {
    // Injected by connect() call:
    const { dispatch, visibleTodos, visibilityFilter } = this.props
    return (
      <div>
        <AddTodo
          onAddClick={text =>
            dispatch(addTodo(text))
          } />
        <TodoList
          todos={this.props.visibleTodos}
          onTodoClick={index =>
            dispatch(completeTodo(index))
          } />
        <Footer
          filter={visibilityFilter}
          onFilterChange={nextFilter =>
            dispatch(setVisibilityFilter(nextFilter))
          } />
      </div>
    )
  }
}

App.propTypes = {
  visibleTodos: PropTypes.arrayOf(PropTypes.shape({
    text: PropTypes.string.isRequired,
    completed: PropTypes.bool.isRequired
  }))
}
```

```
}),
visibilityFilter: PropTypes.oneOf([
  'SHOW_ALL',
  'SHOW_COMPLETED',
  'SHOW_ACTIVE'
]).isRequired
}

// 把 selector 传递给连接的组件
export default connect(visibleTodosSelector)(App)
```

实现撤销历史

在应用中内建撤销和重做功能往往需要开发者有意识的做出一些努力。对于经典的 MVC 框架来说这不是一个简单的问题，因为你需要通过克隆所有相关的 model 来追踪每一个历史状态。此外，你需要关心整个撤销堆栈，因为用户初始化的更改也应该是可撤销的。

这意味着在一个 MVC 应用中实现撤销和重做，通常迫使你用一些类似于 [Command](#) 的特殊的数据修改模式来重写应用中的部分代码。

而在 Redux 中，实现撤销历史却是轻而易举的。有以下三个原因：

- 你想要跟踪的 state 子树不会包含多个模型 (models—just) 。
- state 是不可变的，所有修改已经被描述成分离的 action，而这些 action 与预期的撤销堆栈模型很接近了。
- reducer 的签名 `(state, action) => state` 让它可以自然的实现 “reducer enhancers” 或者 “higher order reducers”。它们可以让你在为 reducer 添加额外的功能时保持这个签名。撤销历史就是一个典型的应用场景。

在动手之前，确认你已经阅读过[基础教程](#)并且良好掌握了 [reducer 合成](#)。本文中的代码会构建于 [基础教程](#) 的示例之上。

文章的第一部分，我们将会解释实现撤销和重做功能所用到的基础概念。

在第二部分中，我们会展示如何使用 [Redux Undo](#) 库来无缝地实现撤销和重做。



Show: All, Completed, Active.

Undo Redo

理解撤消历史

设计状态结构

撤消历史也是你的应用 state 的一部分，我们没有任何原因通过不同的方式实现它。无论 state 如何随着时间不断变化，当你实现撤消和重做这个功能时，你就必须追踪 state 在不同时刻的历史记录。

例如，一个计数器应用的 state 结构看起来可能是这样：

```
{  
  counter: 10  
}
```

如果我们希望在这样一个应用中实现撤消和重做的话，我们必须保存更多的 state 以解决下面几个问题：

- 撤消或重做留下了哪些信息？
- 当前的状态是什么？
- 撤销堆栈中过去（和未来）的状态是什么？

这是一个对于 state 结构的修改建议，可以回答上述问题的：

```
{
```

```
counter: {  
  past: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9],  
  present: 10,  
  future: []  
}  
}
```

现在，如果我们按下“撤消”，我们希望恢复到过去的状态：

```
{  
  counter: {  
    past: [0, 1, 2, 3, 4, 5, 6, 7, 8],  
    present: 9,  
    future: [10]  
  }  
}
```

再来一次：

```
{  
  counter: {  
    past: [0, 1, 2, 3, 4, 5, 6, 7],  
    present: 8,  
    future: [9, 10]  
  }  
}
```

当我们按下“重做”，我们希望往未来状态移动一步：

```
{  
  counter: {  
    past: [0, 1, 2, 3, 4, 5, 6, 7, 8],  
    present: 9,  
    future: [10]  
  }  
}
```

最终，如果处于撤销堆栈中，用户发起了一个操作（例如，减少计数），我们将会丢弃所有未来的信息：

```
{  
  counter: {  
    past: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9],  
    present: 8,  
    future: []  
  }  
}
```

有趣的一点是，我们在撤销堆栈中保存数字，字符串，数组或是对象都没有关系。整个结构始终完全一致：

```
{  
  counter: {  
    past: [0, 1, 2],  
    present: 3,  
    future: [4]  
  }  
}
```

```
{  
  todos: {  
    past: [  
      [],  
      [{ text: 'Use Redux' }],  
      [{ text: 'Use Redux', complete: true }]  
    ],  
    present: [{ text: 'Use Redux', complete: true }, { text: 'Im  
future: [  
      [{ text: 'Use Redux', complete: true }, { text: 'Implement  
    ]  
  }  
}
```

它看起来通常都是这样：

```
{  
  past: Array<T>,  
  present: T,  
  future: Array<T>  
}
```

我们可以保存单一的顶层历史记录：

```
{  
  past: [  
    { counterA: 1, counterB: 1 },  
    { counterA: 1, counterB: 0 },  
    { counterA: 0, counterB: 0 }  
  ],  
  present: { counterA: 2, counterB: 1 },  
  future: []  
}
```

也可以分离的历史记录，用户可以独立地执行撤销和重做操作：

```
{  
  counterA: {  
    past: [1, 0],  
    present: 2,  
    future: []  
  },  
  counterB: {  
    past: [0],  
    present: 1,  
    future: []  
  }  
}
```

接下来我们将会看到如何选择合适的撤销和重做的颗粒度。

设计算法

无论何种特定的数据类型，重做历史记录的 state 结构始终一致：

```
{  
  past: Array<T>,  
  present: T,  
  future: Array<T>  
}
```

让我们讨论一下如何通过算法来操作上文所述的 state 结构。我们可以定义两个 action 来操作该 state： UNDO 和 REDO 。在 reducer 中，我们希望以如下步骤处理这两个 action：

处理 Undo

- 移除 past 中的最后一个元素。
- 将上一步移除的元素赋予 present 。
- 将原来的 present 插入到 future 的最前面。

处理 Redo

- 移除 future 中的第一个元素。
- 将上一步移除的元素赋予 present 。
- 将原来的 present 追加到 past 的最后面。

处理其他 Action

- 将当前的 present 追加到 past 的最后面。
- 将处理完 action 所产生的新的 state 赋予 present 。
- 清空 future 。

第一次尝试：编写 Reducer

```
const initialState = {
  past: [],
  present: null, // (?) 我们如何初始化当前状态?
  future: []
};

function undoable(state = initialState, action) {
  const { past, present, future } = state;

  switch (action.type) {
    case 'UNDO':
      const previous = past[past.length - 1];
      const newPast = past.slice(0, past.length - 1);
      return {
        past: newPast,
        present: previous,
        future: [present, ...future]
      };
    case 'REDO':
      const next = future[0];
      const newFuture = future.slice(1);
      return {
        past: [...past, present],
        present: next,
        future: newFuture
      };
    default:
      // (?) 我们如何处理其他 action?
      return state;
  }
}
```

这个实现是无法使用的，因为它忽略了下面三个重要的问题：

- 我们从何处获取初始的 `present` 状态？我们无法预先知道它。
- 当外部 `action` 被处理完毕后，我们在哪里完成将 `present` 保存到 `past` 的工作？

- 我们如何将 `present` 状态的控制委托给一个自定义的 reducer?

看起来 reducer 并不是正确的抽象方式，但是我们已经非常接近了。

遇见 Reducer Enhancers

你可能已经熟悉 [higher order function](#) 了。如果你使用过 React，也应该熟悉 [higher order component](#)。对于 reducer 来说，也有一种对应的实现模式。

一个 **reducer enhancer**（或者一个 **higher order reducer**）作为一个函数，接收 reducer 作为参数，并返回一个新的 reducer，这个新的 reducer 可以处理新的 action，或者维护更多的 state，亦或者将它无法处理的 action 委托给原始的 reducer 处理。这不是什么新的模式技术（pattern—technically），[`combineReducers\(\)`](#) 就是一个 reducer enhancer，因为它同样接收多个 reducer 并返回一个新的 reducer。

这是一个没有任何额外功能的 reducer enhancer 的示例：

```
function doNothingWith(reducer) {
  return function (state, action) {
    // 仅仅是调用被传入的 reducer
    return reducer(state, action);
  };
}
```

一个可以组合 reducer 的 reducer enhancer 看起来应该像这样：

```
function combineReducers(reducers) {
  return function (state = {}, action) {
    return Object.keys(reducers).reduce((nextState, key) => {
      // 调用每一个 reducer，并将由它管理的部分 state 传给它
      nextState[key] = reducers[key](state[key], action);
      return nextState;
    }, {});
  };
}
```

第二次尝试：编写 Reducer Enhancer

现在我们对 reducer enhancer 有了更深的了解，我们可以明确所谓的 可撤销 到底是什么：

```
function undoable(reducer) {
  // 以一个空的 action 调用 reducer 来产生初始的 state
  const initialState = {
    past: [],
    present: reducer(undefined, {}),
    future: []
  };

  // 返回一个可以执行撤销和重做的新的reducer
  return function (state = initialState, action) {
    const { past, present, future } = state;

    switch (action.type) {
      case 'UNDO':
        const previous = past[past.length - 1];
        const newPast = past.slice(0, past.length - 1);
        return {
          past: newPast,
          present: previous,
          future: [present, ...future]
        };
      case 'REDO':
        const next = future[0];
        const newFuture = future.slice(1);
        return {
          past: [...past, present],
          present: next,
          future: newFuture
        };
      default:
        // 将其他 action 委托给原始的 reducer 处理
        const newPresent = reducer(present, action);
        if (present === newPresent) {
```

```
        return state;
    }
    return {
        past: [...past, present],
        present: newPresent,
        future: []
    };
}
};
```

我们现在可以将任意的 reducer 通过这个拥有可撤销能力的 reducer enhancer 进行封装，从而让它们可以处理 UNDO 和 REDO 这两个 action。

```
// 这是一个 reducer。
function todos(state = [], action) {
  /* ... */
}

// 处理完成之后仍然是一个 reducer!
const undoableTodos = undoable(todos);

import { createStore } from 'redux';
const store = createStore(undoableTodos);

store.dispatch({
  type: 'ADD_TODO',
  text: 'Use Redux'
});

store.dispatch({
  type: 'ADD_TODO',
  text: 'Implement Undo'
});

store.dispatch({
  type: 'UNDO'
});
```

还有一个重要注意点：你需要记住当你恢复一个 state 时，必须把 `.present` 追加到它上面。你也不能忘了需要通过检查 `.past.length` 和 `.future.length` 来决定撤销和重做按钮是否可用。

你可能听说过 Redux 受 Elm 架构 影响颇深。所以这个示例与 [elm-undo-redo package](#) 十分相似也不会太令人吃惊。

使用 Redux Undo

以上这些都非常有用，但是有没有一个库能帮助我们实现 可撤销 功能，而不是由我们自己编写呢？当然有！来看看 [Redux Undo](#)，它可以为你的 Redux 状态树中的任何部分提供撤销和重做功能。

在这个部分中，你会学到如何让 [示例：Todo List](#) 拥有可撤销的功能。你可以在 [todos-with-undo](#) 找到完整的源码。

安装

首先，你必须先执行

```
npm install --save redux-undo
```

这一步会安装一个提供 可撤销 功能的 reducer enhancer 的库。

封装 Reducer

你需要通过 `undoable` 函数强化你的 reducer。例如，如果使用了 `combineReducers()`，你的代码看起来应该像这样：

`reducers.js`

```
import undoable, { distinctState } from 'redux-undo';
```

```
/* ... */

const todoApp = combineReducers({
  visibilityFilter,
  todos: undoable(todos, { filter: distinctState() })
});
```

`distinctState()` 过滤器将会忽略那些没有引起 state 变化的 action。还有一些[其他选项](#)来配置你可撤销的 reducer，例如为撤销和重做动作指定 action 的类型。

你可以在 reducer 合并层次中的任何级别对一个或多个 reducer 执行 `undoable`。由于 `visibilityFilter` 的变化并不会影响撤销历史，我们选择只对 `todos` reducer 进行封装，而不是整个顶层的 reducer。

更新 Selector

现在 `todos` 相关的 state 看起来应该像这样：

```
{
  visibilityFilter: 'SHOW_ALL',
  todos: {
    past: [
      [],
      [{ text: 'Use Redux' }],
      [{ text: 'Use Redux', complete: true }]
    ],
    present: [{ text: 'Use Redux', complete: true }, { text: 'Implement' }],
    future: [
      [{ text: 'Use Redux', complete: true }, { text: 'Implement' }]
    ]
  }
}
```

这意味着你必须要通过 `state.todos.present` 操作 state，而不是原来的 `state.todos`：

containers/App.js

```
function select(state) {
  const presentTodos = state.todos.present;
  return {
    visibleTodos: selectTodos(presentTodos, state.visibilityFilter),
    visibilityFilter: state.visibilityFilter
  };
}
```

为了确认撤销和重做按钮是否可用，你必须检查 `past` 和 `future` 数组是否为空：

containers/App.js

```
function select(state) {
  return {
    undoDisabled: state.todos.past.length === 0,
    redoDisabled: state.todos.future.length === 0,
    visibleTodos: selectTodos(state.todos.present, state.visibilityFilter),
    visibilityFilter: state.visibilityFilter
  };
}
```

添加按钮

现在，你需要做的全部事情就只是为撤销和重做操作添加按钮了。

首先，你需要从 `redux-undo` 中导入 `ActionCreators`，并将他们传递给 `Footer` 组件：

containers/App.js

```
import { ActionCreators } from 'redux-undo';

/* ... */

class App extends Component {
  render() {
    const { dispatch, visibleTodos, visibilityFilter } = this.props;
    return (
      <div>
        {/* ... */}
        <Footer
          filter={visibilityFilter}
          onFilterChange={nextFilter => dispatch(setVisibilityFilter(nextFilter))}
          onUndo={() => dispatch(ActionCreators.undo())}
          onRedo={() => dispatch(ActionCreators.redo())}
          undoDisabled={this.props.undoDisabled}
          redoDisabled={this.props.redoDisabled} />
      </div>
    );
  }
}
```

在 footer 中渲染它们:

components/Footer.js

```
export default class Footer extends Component {

  /* ... */

  renderUndo() {
    return (
      <p>
        <button onClick={this.props.onUndo} disabled={this.props.undoDisabled}>Undo</button>
        <button onClick={this.props.onRedo} disabled={this.props.redoDisabled}>Redo</button>
      </p>
    );
  }
}
```

```
render() {
  return (
    <div>
      {this.renderFilters()}
      {this.renderUndo()}
    </div>
  );
}
}
```

就是这样！在[示例文件夹](#)下执行 `npm install` 和 `npm start` 试试看吧！

排错

这里会列出常见的问题和对应的解决方案。

虽然使用 React 做示例，但是即使你使用了其它库，这些问题和解决方案仍然对你有所帮助。

dispatch action 后什么也没有发生

有时，你 dispatch action 后，view 却没有更新。这是为什么呢？可能有下面几种原因。

永远不要直接修改 reducer 的参数

如果你想修改 Redux 给你传入的 state 或 action，请住手！

Redux 假定你永远不会修改 reducer 里传入的对象。任何时候，你都应该返回一个新的 **state** 对象。即使你没有使用 [Immutable](#) 这样的库，也要保证做到不修改对象。

不变性（Immutability）可以让 [react-redux](#) 高效的监听 state 的细粒度更新。它也让 [redux-devtools](#) 能提供“时间旅行”这类强大特性。

例如，下面的 reducer 就是错误的，因为它改变了 state：

```
function todos(state = [], action) {
  switch (action.type) {
    case 'ADD_TODO':
      // 错误！这会改变 state.actions。
      state.push({
        text: action.text,
        completed: false
      })
      return state
    case 'COMPLETE_TODO':
      // 错误！这会改变 state[action.index]。
```

```
        state[action.index].completed = true
    return state
default:
    return state
}
}
```

应该重写成这样：

```
function todos(state = [], action) {
  switch (action.type) {
    case 'ADD_TODO':
      // 返回新数组
      return [
        ...state,
        {
          text: action.text,
          completed: false
        }
      ]
    case 'COMPLETE_TODO':
      // 返回新数组
      return [
        ...state.slice(0, action.index),
        // 修改之前复制数组
        Object.assign({}, state[action.index], {
          completed: true
        }),
        ...state.slice(action.index + 1)
      ]
    default:
      return state
  }
}
```

虽然需要写更多代码，但是让 Redux 变得可具有可预测性和高效。如果你想减少代码量，你可以用一些辅助方法类似 `React.addons.update` 来让这样的不可变转换操作变得更简单：

```
// 修改前
return [
  ...state.slice(0, action.index),
  Object.assign({}, state[action.index], {
    completed: true
}),
  ...state.slice(action.index + 1)
]

// 修改后
return update(state, {
  [action.index]: {
    completed: {
      $set: true
    }
  }
})
```

最后，如果需要更新 object，你需要使用 Underscore 提供的 `_.extend` 方法，或者更好的，使用 `Object.assign` 的 polyfill

要注意 `Object.assign` 的使用方法。例如，在 reducer 里不要这样使用 `Object.assign(state, newData)`，应该用 `Object.assign({}, state, newData)`。这样它才不会覆盖以前的 `state`。

你也可以通过使用 [Babel transform-object-rest-spread 插件](#)来开启 ES7 对象的 spread 操作：

```
// 修改前:
return [
  ...state.slice(0, action.index),
  Object.assign({}, state[action.index], {
    completed: true
}),
  ...state.slice(action.index + 1)
]

// 修改后:
```

```
return [
  ...state.slice(0, action.index),
  { ...state[action.index], completed: true },
  ...state.slice(action.index + 1)
]
```

注意还在实验阶段的特性注定经常改变，最好不要在大的项目里过多依赖它们。

不要忘记调用 `dispatch(action)`

如果你定义了一个 action 创建函数，调用它并不会自动 dispatch 这个 action。比如，下面的代码什么也不会做：

`TodoActions.js`

```
export function addTodo(text) {
  return { type: 'ADD_TODO', text }
}
```

`AddTodo.js`

```
import { Component } from 'react'
import { addTodo } from './TodoActions'

class AddTodo extends Component {
  handleClick() {
    // 不起作用!
    addTodo('Fix the issue')
  }

  render() {
    return (
      <button onClick={() => this.handleClick()}>
        Add
      </button>
    )
  }
}
```

```

    }
}

```

它不起作用是因为你的 action 创建函数只是一个返回 action 的函数而已。你需要手动 dispatch 它。我们不能在定义时把 action 创建函数绑定到指定的 Store 上，因为应用在服务端渲染时需要为每个请求都对应一个独立的 Redux store。

解法是调用 `store` 实例上的 `dispatch()` 方法。

```

handleClick() {
  // 生效! (但你需要先以某种方式拿到 store)
  store.dispatch(addTodo('Fix the issue'))
}

```

如果组件的层级非常深，把 `store` 一层层传下去很麻烦。因此 `react-redux` 提供了 `connect` 这个 [高阶组件](#)，它除了可以帮你监听 Redux store，还会把 `dispatch` 注入到组件的 `props` 中。

修复后的代码是这样的：

AddTodo.js

```

import React, { Component } from 'react'
import { connect } from 'react-redux'
import { addTodo } from './TodoActions'

class AddTodo extends Component {
  handleClick() {
    // 生效!
    this.props.dispatch(addTodo('Fix the issue'))
  }

  render() {
    return (
      <button onClick={() => this.handleClick()}>

```

```
        Add
      </button>
    )
}
}

// 除了 state, `connect` 还把 `dispatch` 放到 props 里。
export default connect()(AddTodo)
```

如果你想的话也可以把 `dispatch` 手动传给其它组件。

其它问题

在 Discord [Reactiflux](#) 里的 **redux** 频道里提问，或者[提交一个 issue](#)。
如果问题终于解决了，请把解法[写到文档里](#)，以便别人遇到同样问题时参考。

词汇表

这是 Redux 的核心概念词汇表以及这些核心概念的类型签名。这些类型使用了[流标注法](#)进行记录。

State

```
type State = any
```

State (也称为 *state tree*) 是一个宽泛的概念，但是在 Redux API 中，通常是指一个唯一的 state 值，由 store 管理且由 `getState()` 方法获得。它表示了 Redux 应用的全部状态，通常为一个多层嵌套的对象。

约定俗成，顶层 state 或为一个对象，或像 Map 那样的键-值集合，也可以是任意的数据类型。然而你应尽可能确保 state 可以被序列化，而且不要把什么数据都放进去，导致无法轻松地把 state 转换成 JSON。

Action

```
type Action = Object
```

Action 是一个普通对象，用来表示即将改变 state 的意图。它是将数据放入 store 的唯一途径。无论是从 UI 事件、网络回调，还是其他诸如 WebSocket 之类的数据源所获得的数据，最终都会被 dispatch 成 action。

约定俗成，action 必须拥有一个 `type` 域，它指明了需要被执行的 action type。Type 可以被定义为常量，然后从其他 module 导入。比起用 [Symbols](#) 表示 `type`，使用 String 是更好的方法，因为 string 可以被序列化。

除了 `type` 之外，action 对象的结构其实完全取决于你自己。如果你感兴趣

的话, 请参考 [Flux Standard Action](#), 了解如何构建 action。

还有就是请看后面的 [异步 action](#)。

Reducer

```
type Reducer<S, A> = (state: S, action: A) => S
```

Reducer (也称为 *reducing function*) 函数接受两个参数: 之前累积运算的结果和当前被累积的值, 返回的是一个新的累积结果。该函数把一个集合归并成一个单值。

Reducer 并不是 Redux 特有的函数 —— 它是函数式编程中的一个基本概念, 甚至大部分的非函数式语言比如 JavaScript, 都有一个内置的 `reduce` API。对于 JavaScript, 这个 API 是 [`Array.prototype.reduce\(\)`](#)。

在 Redux 中, 累计运算的结果是 `state` 对象, 而被累积的值是 `action`。Reducer 由上次累积的结果 `state` 与当前被累积的 `action` 计算得到一个新 `state`。这些 Reducer 必须是纯函数, 而且当输入相同时返回的结果也会相同。它们不应该产生任何副作用。正因如此, 才使得诸如热重载和时间旅行这些很棒的功能成为可能。

Reducer 是 Redux 之中最重要的概念。

不要在 reducer 中有 API 调用

dispatch 函数

```
type BaseDispatch = (a: Action) => Action
type Dispatch = (a: Action | AsyncAction) => any
```

dispatching function (或简言之 *dispatch function*) 是一个接收 `action` 或者[异步 action](#) 的函数。

步 `action` 的函数，该函数要么往 store 分发一个或多个 `action`，要么不分发任何 `action`。

我们必须分清一般的 `dispatch function` 以及由 `store` 实例提供的没有 `middleware` 的 base `dispatch` function 之间的区别。

Base dispatch function 总是同步地把 `action` 与上一次从 `store` 返回的 `state` 发往 `reducer`，然后计算出新的 `state`。它期望 `action` 会是一个可以被 `reducer` 消费的普通对象。

`Middleware` 封装了 base dispatch function，允许 dispatch function 处理 `action` 之外的异步 `action`。Middleware 可以改变、延迟、忽略 `action` 或异步 `action`，也可以在传递给下一个 middleware 之前对它们进行解释。获取更多信息请往后看。

Action Creator

```
type ActionCreator = (...args: any) => Action | AsyncAction
```

Action Creator 很简单，就是一个创建 `action` 的函数。不要混淆 `action` 和 `action creator` 这两个概念。`Action` 是一个信息的负载，而 `action creator` 是一个创建 `action` 的工厂。

调用 `action creator` 只会生产 `action`，但不分发。你需要调用 `store` 的 `dispatch` function 才会引起变化。有时我们讲 *bound action creator*，是指一个函数调用了 `action creator` 并立即将结果分发给一个特定的 `store` 实例。

如果 `action creator` 需要读取当前的 `state`、调用 API、或引起诸如路由变化等副作用，那么它应该返回一个异步 `action` 而不是 `action`。

异步 Action

```
type AsyncAction = any
```

异步 *action* 是一个发给 *dispatching* 函数的值，但是这个值还不能被 *reducer* 消费。在发往 base `dispatch()` function 之前，`middleware` 会把异步 *action* 转换成一个或一组 *action*。异步 *action* 可以有多种 type，这取决于你所使用的 *middleware*。它通常是 Promise 或者 thunk 之类的异步原生数据类型，虽然不会立即把数据传递给 *reducer*，但是一旦操作完成就会触发 *action* 的分发事件。

Middleware

```
type MiddlewareAPI = { dispatch: Dispatch, getState: () => State }
type Middleware = (api: MiddlewareAPI) => (next: Dispatch) => D:
```

Middleware 是一个组合 `dispatch function` 的高阶函数，返回一个新的 `dispatch function`，通常将 `异步 actions` 转换成 *action*。

Middleware 利用复合函数使其可以组合其他函数，可用于记录 *action* 日志、产生其他诸如变化路由的副作用，或将异步的 API 调用变为一组同步的 *action*。

请见 `applyMiddleware(...middlewares)` 获取 *middleware* 的详细内容。

Store

```
type Store = {
  dispatch: Dispatch
  getState: () => State
  subscribe: (listener: () => void) => () => void
  replaceReducer: (reducer: Reducer) => void
};
```

Store 维持着应用的 state tree 对象。因为应用的构建发生于 reducer，所以一个 Redux 应用中应当只有一个 Store。

- `dispatch(action)` 是上述的 base dispatch function。
- `getState()` 返回当前 store 的 state。
- `subscribe(listener)` 注册一个 state 发生变化时的回调函数。
- `replaceReducer(nextReducer)` 可用于热重载和代码分割。通常你不需要用到这个 API。

详见完整的 [store API reference](#)。

Store Creator

```
type StoreCreator = (reducer: Reducer, initialState: ?State) =>
```

Store creator 是一个创建 Redux store 的函数。就像 dispatching function 那样，我们必须分清通过 `createStore(reducer, initialState)` 由 Redux 导出的 base store creator 与从 store enhancer 返回的 store creator 之间的区别。

Store enhancer

```
type StoreEnhancer = (next: StoreCreator) => StoreCreator
```

Store enhancer 是一个组合 store creator 的高阶函数，返回一个新的强化过的 store creator。这与 middleware 相似，它也允许你通过复合函数改变 store 接口。

Store enhancer 与 React 的高阶 component 概念一致，通常也会称为

“component enhancers”。

因为 store 并非实例，更像是一个函数集合的普通对象，所以可以轻松地创建副本，也可以在不改变原先的 store 的条件下修改副本。在 [compose](#) 文档中有一个示例演示了这种做法。

大多数时候你基本不用编写 store enhancer，但你可能会在 [developer tools](#) 中用到。正因为 store enhancer，应用程序才有可能察觉不到“时间旅行”。有趣的是，[Redux middleware 本身的实现](#)就是一个 store enhancer。

API 文档

Redux 的 API 非常少。Redux 定义了一系列的约定 (contract) 来让你来实现 (例如 [reducers](#))，同时提供少量辅助函数来把这些约定整合到一起。

这一章会介绍所有的 Redux API。记住，Redux 只关心如何管理 state。在实际的项目中，你还需要使用 UI 绑定库如 [react-redux](#)。

顶级暴露的方法

- [createStore\(reducer, \[initialState\]\)](#)
- [combineReducers\(reducers\)](#)
- [applyMiddleware\(...middlewares\)](#)
- [bindActionCreators\(actionCreators, dispatch\)](#)
- [compose\(...functions\)](#)

Store API

- [Store](#)
 - [getState\(\)](#)
 - [dispatch\(action\)](#)
 - [subscribe\(listener\)](#)
 - [getReducer\(\)](#)
 - [replaceReducer\(nextReducer\)](#)

引入

上面介绍的所有函数都是顶级暴露的方法。都可以这样引入：

ES6

```
import { createStore } from 'redux';
```

ES5 (CommonJS)

```
var createStore = require('redux').createStore;
```

ES5 (UMD build)

```
var createStore = Redux.createStore;
```

createStore(reducer, [initialState])

创建一个 Redux [store](#) 来以存放应用中所有的 state。

应用中应有且仅有一个 store。

参数

1. `reducer` (*Function*): 接收两个参数, 分别是当前的 state 树和要处理的 `action`, 返回新的 state 树。
2. `[initialState] (any)`: 初始时的 state。在同构应用中, 你可以决定是否把服务端传来的 state 水合 (hydrate) 后传给它, 或者从之前保存的用户会话中恢复一个传给它。如果你使用 [combineReducers](#) 创建 `reducer`, 它必须是一个普通对象, 与传入的 keys 保持同样的结构。否则, 你可以自由传入任何 `reducer` 可理解的内容。

返回值

([Store](#)): 保存了应用所有 state 的对象。改变 state 的唯一方法是 [dispatch](#) action。你也可以 [subscribe](#) 监听 state 的变化, 然后更新 UI。

示例

```
import { createStore } from 'redux'

function todos(state = [], action) {
  switch (action.type) {
    case 'ADD_TODO':
      return state.concat([ action.text ])
    default:
      return state
  }
}
```

```

let store = createStore(todos, [ 'Use Redux' ])

store.dispatch({
  type: 'ADD_TODO',
  text: 'Read the docs'
})

console.log(store.getState())
// [ 'Use Redux', 'Read the docs' ]

```

小贴士

- 应用中不要创建多个 store! 相反, 使用 `combineReducers` 来把多个 reducer 创建成一个根 reducer。
- 你可以决定 state 的格式。你可以使用普通对象或者 `Immutable` 这类的实现。如果你不知道如何做, 刚开始可以使用普通对象。
- 如果 state 是普通对象, 永远不要修改它! 比如, reducer 里不要使用 `Object.assign(state, newData)`, 应该使用 `Object.assign({}, state, newData)`。这样才不会覆盖旧的 state。也可以使用 Babel 阶段 1 中的 `ES7 对象的 spread 操作` 特性中的 `return { ...state, ...newData }`。
- 对于服务端运行的同构应用, 为每一个请求创建一个 store 实例, 以此让 store 相隔离。dispatch 一系列请求数据的 action 到 store 实例上, 等待请求完成后再在服务端渲染应用。
- 当 store 创建后, Redux 会 dispatch 一个 action 到 reducer 上, 来用初始的 state 来填充 store。你不需要处理这个 action。但要记住, 如果第一个参数也就是传入的 state 如果是 `undefined` 的话, reducer 应该返回初始的 state 值。

Store

Store 就是用来维持应用所有的 state 树 的一个对象。改变 store 内 state 的惟一途径是对它 dispatch 一个 action。

Store 不是类。它只是有几个方法的对象。要创建它，只需要把根部的 reducing 函数 传递给 createStore 。

Flux 用户使用注意

如果你以前使用 Flux，那么你只需要注意一个重要的区别。Redux 没有 Dispatcher 且不支持多个 store。相反，只有一个单一的 store 和一个根级的 reduce 函数 (reducer) 。随着应用不断变大，你应该把根级的 reducer 拆成多个小的 reducers，分别独立地操作 state 树的不同部分，而不是添加新的 stores。这就像一个 React 应用只有一个根级的组件，这个根组件又由很多小组件构成。

Store 方法

- `getState()`
- `dispatch(action)`
- `subscribe(listener)`
- `replaceReducer(nextReducer)`

Store 方法

getState()

返回应用当前的 state 树。

它与 store 的最后一个 reducer 返回值相同。

返回值

(any): 应用当前的 state 树。

dispatch(action)

分发 action。这是触发 state 变化的惟一途径。

会使用当前 `getState()` 的结果和传入的 `action` 以同步方式的调用 store 的 `reduce` 函数。返回值会被作为下一个 state。从现在开始，这就成为了 `getState()` 的返回值，同时变化监听器(change listener)会被触发。

Flux 用户使用注意

当你在 `reducer` 内部调用 `dispatch` 时，将会抛出错误提示“Reducers may not dispatch actions. (Reducer 内不能 dispatch action)”。这相当于 Flux 里的“Cannot dispatch in a middle of dispatch (dispatch 过程中不能再 dispatch)”，但并不会引起对应的错误。在 Flux 里，当 Store 处理 action 和触发 update 事件时，`dispatch` 是禁止的。这个限制并不好，因为他限制了不能在生命周期回调里 `dispatch action`，还有其它一些本来很正常的地方。

在 Redux 里，只会在根 `reducer` 返回新 state 结束后再会调用事件监听器，因此，你可以在事件监听器里再做 `dispatch`。惟一使你不能在 `reducer` 中途 `dispatch` 的原因是确保 `reducer` 没有副作用。如果 `action` 处理会产生副作用，正确的做法是使用异步 `action 创建函数`。

参数

1. `action (Object)`: 描述应用变化的普通对象。Action 是把数据传入 store 的惟一途径，所以任何数据，无论来自 UI 事件，网络回调或者是其它资源如 WebSockets，最终都应该以 `action` 的形式被 `dispatch`。按照约定，`action` 具有 `type` 字段来表示它的类型。`type` 也可被定义为常量或者是从其它模块引入。最好使用字符串，而不是 `Symbols` 作为 `action`，因为字符串是可以被序列化的。除了 `type` 字段外，`action` 对象的结构完全取决于你。参照 `Flux 标准 Action` 获取如何组织 `action` 的

建议。

返回值

(Object[†]): 要 dispatch 的 action。

注意

[†] 使用 `createStore` 创建的“纯正” store 只支持普通对象类型的 action，而且会立即传到 reducer 来执行。

但是，如果你用 `applyMiddleware` 来套住 `createStore` 时，middleware 可以修改 action 的执行，并支持执行 dispatch `intent`（意图）。Intent 一般是异步操作如 Promise、Observable 或者 Thunk。

Middleware 是由社区创建，并不会同 Redux 一起发行。你需要手动安装 `redux-thunk` 或者 `redux-promise` 库。你也可以创建自己的 middleware。

想学习如何描述异步 API 调用？看一下 action 创建函数里当前的 state，执行一个有副作用的操作，或者以链式操作执行它们，参照 `applyMiddleware` 中的示例。

示例

```
import { createStore } from 'redux'
let store = createStore(todos, [ 'Use Redux' ])

function addTodo(text) {
  return {
    type: 'ADD_TODO',
    text
  }
}

store.dispatch(addTodo('Read the docs'))
store.dispatch(addTodo('Read about the middleware'))
```

subscribe(listener)

添加一个变化监听器。每当 dispatch action 的时候就会执行， state 树中的一部分可能已经变化。你可以在回调函数里调用 `getState()` 来拿到当前 state。

这是一个底层 API。多数情况下，你不会直接使用它，会使用一些 React（或其它库）的绑定。如果你想让回调函数执行的时候使用当前的 state，你可以 [把 store 转换成一个 Observable](#) 或者写一个定制的 [observeStore 工具](#)。

如果需要解绑这个变化监听器，执行 `subscribe` 返回的函数即可。

参数

1. `listener (Function)`: 每当 dispatch action 的时候都会执行的回调。state 树中的一部分可能已经变化。你可以在回调函数里调用 `getState()` 来拿到当前 state。store 的 reducer 应该是纯函数，因此你可能需要对 state 树中的引用做深度比较来确定它的值是否有变化。

返回值

`(Function)`: 一个可以解绑变化监听器的函数。

示例

```
function select(state) {
  return state.some.deep.property
}

let currentValue
function handleChange() {
  let previousValue = currentValue
  currentValue = select(store.getState())

  if (previousValue !== currentValue) {
```

```
        console.log('Some deep nested property changed from', previousValue)
    }
}

let unsubscribe = store.subscribe(handleChange)
handleChange()
```

replaceReducer (nextReducer)

替换 store 当前用来计算 state 的 reducer。

这是一个高级 API。只有在你需要实现代码分隔，而且需要立即加载一些 reducer 的时候才可能会用到它。在实现 Redux 热加载机制的时候也可能会用到。

参数

1. `reducer` (*Function*) store 会使用的下一个 reducer。

combineReducers(reducers)

随着应用变得复杂，需要对 [reducer 函数](#) 进行拆分，拆分后的每一块独立负责管理 [state](#) 的一部分。

`combineReducers` 辅助函数的作用是，把一个由多个不同 `reducer` 函数作为 `value` 的 `object`，合并成一个最终的 `reducer` 函数，然后就可以对这个 `reducer` 调用 [createStore](#)。

合并后的 `reducer` 可以调用各个子 `reducer`，并把它们的结果合并成一个 `state` 对象。`state` 对象的结构由传入的多个 `reducer` 的 `key` 决定。

最终，`state` 对象的结构会是这样的：

```
{  
  reducer1: ...  
  reducer2: ...  
}
```

通过为传入对象的 `reducer` 命名不同来控制 `state key` 的命名。例如，你可以调用 `combineReducers({ todos: myTodosReducer, counter: myCounterReducer })` 将 `state` 结构变为 `{ todos, counter }`。

通常的做法是命名 `reducer`，然后 `state` 再去分割那些信息，因此你可以使用 ES6 的简写方法：`combineReducers({ counter, todos })`。这与 `combineReducers({ counter: counter, todos: todos })` 一样。

Flux 用户使用须知

本函数可以帮助你组织多个 `reducer`，使它们分别管理自身相关联的 `state`。类似于 Flux 中的多个 `store` 分别管理不同的 `state`。在 Redux 中，只有一个 `store`，但是 `combineReducers` 让你拥有多个 `reducer`，同时保持各自负责逻辑块的独立性。

参数

1. `reducers (Object)`: 一个对象，它的值（`value`）对应不同的 `reducer` 函数，这些 `reducer` 函数后面会被合并成一个。下面会介绍传入 `reducer` 函数需要满足的规则。

之前的文档曾建议使用 ES6 的 `import * as reducers` 语法来获得 `reducer` 对象。这一点造成了很多疑问，因此现在建议在 `reducers/index.js` 里使用 `combineReducers()` 来对外输出一个 `reducer`。下面有示例说明。

返回值

(*Function*): 一个调用 `reducers` 对象里所有 `reducer` 的 `reducer`，并且构造一个与 `reducers` 对象结构相同的 `state` 对象。

注意

本函数设计的时候有点偏主观，就是为了避免新手犯一些常见错误。也因些我们故意设定一些规则，但如果你自己手动编写根 `reducer` 时并不需要遵守这些规则。

每个传入 `combineReducers` 的 `reducer` 都需满足以下规则：

- 所有未匹配到的 `action`，必须把它接收到的第一个参数也就是那个 `state` 原封不动返回。
- 永远不能返回 `undefined`。当过早 `return` 时非常容易犯这个错误，为了避免错误扩散，遇到这种情况时 `combineReducers` 会抛异常。
- 如果传入的 `state` 就是 `undefined`，一定要返回对应 `reducer` 的初始 `state`。根据上一条规则，初始 `state` 禁止使用 `undefined`。使用 ES6 的默认参数值语法来设置初始 `state` 很容易，但你也可以手动检查第一个参数是否为 `undefined`。

虽然 `combineReducers` 自动帮你检查 reducer 是否符合以上规则，但你也应该牢记，并尽量遵守。

示例

reducers/todos.js

```
export default function todos(state = [], action) {
  switch (action.type) {
    case 'ADD_TODO':
      return state.concat([action.text])
    default:
      return state
  }
}
```

reducers/counter.js

```
export default function counter(state = 0, action) {
  switch (action.type) {
    case 'INCREMENT':
      return state + 1
    case 'DECREMENT':
      return state - 1
    default:
      return state
  }
}
```

reducers/index.js

```
import { combineReducers } from 'redux'
import todos from './todos'
import counter from './counter'

export default combineReducers({
  todos,
```

```
    counter
})
```

App.js

```
import { createStore } from 'redux'
import reducer from './reducers/index'

let store = createStore(reducer)
console.log(store.getState())
// {
//   counter: 0,
//   todos: []
// }

store.dispatch({
  type: 'ADD_TODO',
  text: 'Use Redux'
})
console.log(store.getState())
// {
//   counter: 0,
//   todos: [ 'Use Redux' ]
// }
```

小贴士

- 本方法只是起辅助作用！你可以自行实现[不同功能的](#) `combineReducers`，甚至像实现其它函数一样，明确地写一个根 `reducer` 函数，用它把子 `reducer` 手动组装成 `state` 对象。
- 在 `reducer` 层级的任何一级都可以调用 `combineReducers`。并不是一定要在最外层。实际上，你可以把一些复杂的子 `reducer` 拆分成单独的孙子级 `reducer`，甚至更多层。

applyMiddleware(...middlewares)

使用包含自定义功能的 middleware 来扩展 Redux 是一种推荐的方式。

Middleware 可以让你包装 store 的 `dispatch` 方法来达到你想要的目的。同时，middleware 还拥有“可组合”这一关键特性。多个 middleware 可以被组合到一起使用，形成 middleware 链。其中，每个 middleware 都不需要关心链中它前后的 middleware 的任何信息。

Middleware 最常见的使用场景是无需引用大量代码或依赖类似 Rx 的第三方库实现异步 actions。这种方式可以让你像 `dispatch` 一般的 actions 那样 `dispatch` 异步 actions。

例如，[redux-thunk](#) 支持 `dispatch` function，以此让 action creator 控制反转。被 `dispatch` 的 function 会接收 `dispatch` 作为参数，并且可以异步调用它。这类的 function 就称为 *thunk*。另一个 middleware 的示例是 [redux-promise](#)。它支持 `dispatch` 一个异步的 [Promise](#) action，并且在 `Promise` `resolve` 后可以 `dispatch` 一个普通的 action。

Middleware 并不需要和 [createStore](#) 绑在一起使用，也不是 Redux 架构的基础组成部分，但它带来的益处让我们认为有必要在 Redux 核心中包含对它的支持。因此，虽然不同的 middleware 可能在易用性和用法上有所不同，它仍被作为扩展 `dispatch` 的唯一标准的方式。

参数

- `...middlewares (arguments)`: 遵循 Redux *middleware API* 的函数。每个 middleware 接受 `Store` 的 `dispatch` 和 `getState` 函数作为命名参数，并返回一个函数。该函数会被传入被称为 `next` 的下一个 middleware 的 `dispatch` 方法，并返回一个接收 `action` 的新函数，这个函数可以直接调用 `next(action)`，或者在其他需要的时刻调用，甚至根本不去调用它。调用链中最后一个 middleware 会接受真实的 store 的 `dispatch` 方法作为 `next` 参数，并借此结束调用链。所以，

middleware 的函数签名是 `({ getState, dispatch }) => next => action`。

返回值

(Function) 一个应用了 middleware 后的 store enhancer。这个 store enhancer 就是一个函数，并且需要应用到 `createStore`。它会返回一个应用了 middleware 的新的 `createStore`。

示例: 自定义 Logger Middleware

```
import { createStore, applyMiddleware } from 'redux'
import todos from './reducers'

function logger({ getState }) {
  return (next) => (action) => {
    console.log('will dispatch', action)

    // 调用 middleware 链中下一个 middleware 的 dispatch。
    let returnValue = next(action)

    console.log('state after dispatch', getState())

    // 一般会是 action 本身，除非
    // 后面的 middleware 修改了它。
    return returnValue
  }
}

let createStoreWithMiddleware = applyMiddleware(logger)(createStore)
let store = createStoreWithMiddleware(todos, [ 'Use Redux' ])

store.dispatch({
  type: 'ADD_TODO',
  text: 'Understand the middleware'
})
// (将打印如下信息:)
// will dispatch: { type: 'ADD_TODO', text: 'Understand the middleware' }
// state after dispatch: [ 'Use Redux', 'Understand the middleware' ]
```

示例: 使用 Thunk Middleware 来做异步 Action

```
import { createStore, combineReducers, applyMiddleware } from 'redux'
import thunk from 'redux-thunk'
import * as reducers from './reducers'

// 调用 applyMiddleware, 使用 middleware 增强 createStore:
let createStoreWithMiddleware = applyMiddleware(thunk)(createStore)

// 像原生 createStore 一样使用。
let reducer = combineReducers(reducers)
let store = createStoreWithMiddleware(reducer)

function fetchSecretSauce() {
  return fetch('https://www.google.com/search?q=secret+sauce')
}

// 这些是你已熟悉的普通 action creator。
// 它们返回的 action 不需要任何 middleware 就能被 dispatch。
// 但是，他们只表达「事实」，并不表达「异步数据流」

function makeASandwich(forPerson, secretSauce) {
  return {
    type: 'MAKE_SANDWICH',
    forPerson,
    secretSauce
  }
}

function apologize(fromPerson, toPerson, error) {
  return {
    type: 'APOLOGIZE',
    fromPerson,
    toPerson,
    error
  }
}

function withdrawMoney(amount) {
```

```
return {
  type: 'WITHDRAW',
  amount
}
}

// 即使不使用 middleware, 你也可以 dispatch action:
store.dispatch(withdrawMoney(100))

// 但是怎样处理异步 action 呢,
// 比如 API 调用, 或者是路由跳转?

// 来看一下 thunk。
// Thunk 就是一个返回函数的函数。
// 下面就是一个 thunk。

function makeASandwichWithSecretSauce(forPerson) {

  // 控制反转!
  // 返回一个接收 `dispatch` 的函数。
  // Thunk middleware 知道如何把异步的 thunk action 转为普通 action.

  return function (dispatch) {
    return fetchSecretSauce().then(
      sauce => dispatch(makeASandwich(forPerson, sauce)),
      error => dispatch(apologize('The Sandwich Shop', forPerson))
    )
  }
}

// Thunk middleware 可以让我们像 dispatch 普通 action
// 一样 dispatch 异步的 thunk action.

store.dispatch(
  makeASandwichWithSecretSauce('Me')
)

// 它甚至负责回传 thunk 被 dispatch 后返回的值,
// 所以可以继续串连 Promise, 调用它的 .then() 方法。

store.dispatch(
  makeASandwichWithSecretSauce('My wife')
)
```

```

).then(() => {
  console.log('Done!')
})

// 实际上，可以写一个 dispatch 其它 action creator 里
// 普通 action 和异步 action 的 action creator,
// 而且可以使用 Promise 来控制数据流。

function makeSandwichesForEverybody() {
  return function (dispatch, getState) {
    if (!getState().sandwiches.isShopOpen) {

      // 返回 Promise 并不是必须的，但这是一个很好的约定，
      // 为了让调用者能够在异步的 dispatch 结果上直接调用 .then() 方法

      return Promise.resolve()
    }

    // 可以 dispatch 普通 action 对象和其它 thunk,
    // 这样我们就可以在一个数据流中组合多个异步 action.

    return dispatch(
      makeASandwichWithSecretSauce('My Grandma')
    ).then(() =>
      Promise.all([
        dispatch(makeASandwichWithSecretSauce('Me')),
        dispatch(makeASandwichWithSecretSauce('My wife'))
      ])
    ).then(() =>
      dispatch(makeASandwichWithSecretSauce('Our kids'))
    ).then(() =>
      dispatch(getState().myMoney > 42 ?
        withdrawMoney(42) :
        apologize('Me', 'The Sandwich Shop')
      )
    )
  }
}

// 这在服务端渲染时很有用，因为我可以等到数据
// 准备好后，同步的渲染应用。

```

```
import { renderToString } from 'react-dom/server'

store.dispatch(
  makeSandwichesForEverybody()
).then(() =>
  response.send(renderToString(<MyApp store={store} />))
)

// 也可以在任何导致组件的 props 变化的时刻
// dispatch 一个异步 thunk action.

import { connect } from 'react-redux'
import { Component } from 'react'

class SandwichShop extends Component {
  componentDidMount() {
    this.props.dispatch(
      makeASandwichWithSecretSauce(this.props.forPerson)
    )
  }

  componentWillReceiveProps(nextProps) {
    if (nextProps.forPerson !== this.props.forPerson) {
      this.props.dispatch(
        makeASandwichWithSecretSauce(nextProps.forPerson)
      )
    }
  }

  render() {
    return <p>{this.props.sandwiches.join('mustard')}</p>
  }
}

export default connect(
  state => ({
    sandwiches: state.sandwiches
  })
)(SandwichShop)
```

小贴士

- Middleware 只是包装了 store 的 `dispatch` 方法。技术上讲，任何 middleware 能做的事情，都可能通过手动包装 `dispatch` 调用来实现，但是放在同一个地方统一管理会使整个项目的扩展变的容易得多。
- 如果除了 `applyMiddleware`，你还用了其它 store enhancer，一定要把 `applyMiddleware` 放到组合链的前面，因为 middleware 可能会包含异步操作。比如，它应该在 `redux-devtools` 前面，否则 DevTools 就看不到 Promise middleware 里 `dispatch` 的 action 了。
- 如果你想有条件地使用 middleware，记住只 import 需要的部分：

```
let middleware = [ a, b ]
if (process.env.NODE_ENV !== 'production') {
  let c = require('some-debug-middleware')
  let d = require('another-debug-middleware')
  middleware = [ ...middleware, c, d ]
}
const createStoreWithMiddleware = applyMiddleware(...middlew
```

这样做有利于打包时去掉不需要的模块，减小打包文件大小。

- 有想过 `applyMiddleware` 本质是什么吗？它肯定是比 middleware 还强大的扩展机制。实际上，`applyMiddleware` 只是被称为 Redux 最强大的扩展机制的 `store enhancers` 中的一个范例而已。你不太可能需要实现自己的 store enhancer。另一个 store enhancer 示例是 `redux-devtools`。Middleware 并没有 store enhancer 强大，但开发起来却是更容易的。
- Middleware 听起来比实际难一些。真正理解 middleware 的唯一办法是了解现有的 middleware 是如何工作的，并尝试自己实现。需要的功能可能错综复杂，但是你会发现大部分 middleware 实际上很小，只有 10 行左右，是通过对它们的组合使用来达到最终的目的。

bindActionCreators(actionCreator, dispatch)

把 `action creators` 转成拥有同名 `keys` 的对象，但使用 `dispatch` 把每个 `action creator` 包围起来，这样可以直接调用它们。

一般情况下你可以直接在 `Store` 实例上调用 `dispatch`。如果你在 React 中使用 Redux，`react-redux` 会提供 `dispatch`。

惟一使用 `bindActionCreators` 的场景是当你需要把 `action creator` 往下传到一个组件上，却不想让这个组件觉察到 Redux 的存在，而且不希望把 Redux store 或 `dispatch` 传给它。

为方便起见，你可以传入一个函数作为第一个参数，它会返回一个函数。

参数

1. `actionCreators` (*Function or Object*): 一个 `action creator`，或者键值是 `action creators` 的对象。
2. `dispatch` (*Function*): 一个 `dispatch` 函数，由 `Store` 实例提供。

返回值

(*Function or Object*): 一个与原对象类似的对象，只不过这个对象中的的每个函数值都可以直接 `dispatch action`。如果传入的是一个函数，返回的也是一个函数。

示例

TodoActionCreators.js

```
export function addTodo(text) {
```

```
return {
  type: 'ADD_TODO',
  text
};

export function removeTodo(id) {
  return {
    type: 'REMOVE_TODO',
    id
  };
}
```

SomeComponent.js

```
import { Component } from 'react';
import { bindActionCreators } from 'redux';
import { connect } from 'react-redux';

import * as TodoActionCreators from './TodoActionCreators';
console.log(TodoActionCreators);
// {
//   addTodo: Function,
//   removeTodo: Function
// }

class TodoListContainer extends Component {
  componentDidMount() {
    // 由 react-redux 注入:
    let { dispatch } = this.props;

    // 注意: 这样做行不通:
    // TodoActionCreators.addTodo('Use Redux');

    // 你只是调用了创建 action 的方法。
    // 你必须要 dispatch action 而已。

    // 这样做行得通:
    let action = TodoActionCreators.addTodo('Use Redux');
    dispatch(action);
  }
}
```

```
render() {
  // 由 react-redux 注入:
  let { todos, dispatch } = this.props;

  // 这是应用 bindActionCreators 比较好的场景:
  // 在子组件里, 可以完全不知道 Redux 的存在。

  let boundActionCreators = bindActionCreators(TodoActionCreators,
    console.log(boundActionCreators));
  // {
  //   addTodo: Function,
  //   removeTodo: Function
  // }

  return (
    <TodoList todos={todos}
      {...boundActionCreators} />
  );
}

// 一种可以替换 bindActionCreators 的做法是直接把 dispatch 函数
// 和 action creators 当作 props
// 传递给子组件
// return <TodoList todos={todos} dispatch={dispatch} />;
}

}

export default connect(
  state => ({ todos: state.todos })
)(TodoListContainer)
```

小贴士

- 你或许要问：为什么不直接把 action creators 绑定到 store 实例上，就像传统 Flux 那样？问题是这样做的话如果开发同构应用，在服务端渲染时就不行了。多数情况下，你 每个请求都需要一个独立的 store 实例，这样你可以为它们提供不同的数据，但是在定义的时候绑定 action creators，你就可以使用一个唯一的 store 实例来对应所有请求了。

- 如果你使用 ES5，不能使用 `import * as` 语法，你可以把 `require('./TodoActionCreators')` 作为第一个参数传给 `bindActionCreators`。惟一要考虑的是 `actionCreators` 的参数全是函数。模块加载系统并不重要。

compose(...functions)

从右到左来组合多个函数。

这是函数式编程中的方法，为了方便，被放到了 Redux 里。当需要把多个 [store 增强器](#) 依次执行的时候，需要用到它。

参数

1. (*arguments*): 需要合成的多个函数。每个函数都接收一个函数作为参数，然后返回一个函数。

返回值

(*Function*): 从右到左把接收到的函数合成后的最终函数。

示例

下面示例演示了如何使用 `compose` 增强 [store](#)，这个 `store` 与 [applyMiddleware](#) 和 [redux-devtools](#) 一起使用。

```
import { createStore, combineReducers, applyMiddleware, compose
import thunk from 'redux-thunk';
import * as reducers from '../reducers/index';

let reducer = combineReducers(reducers);
let middleware = [thunk];

let finalCreateStore;

// 生产环境中，我们希望只使用 middleware。
// 而在开发环境中，我们还希望使用一些 redux-devtools 提供的一些 store †
// UglifyJS 会在构建过程中把一些不会执行的死代码去除掉。

if (process.env.NODE_ENV === 'production') {
  finalCreateStore = applyMiddleware(...middleware)(createStore
```

```
    } else {
      finalCreateStore = compose(
        applyMiddleware(...middleware),
        require('redux-devtools').devTools(),
        require('redux-devtools').persistState(
          window.location.href.match(/[^&]+debug_session=([^&]+)\b/)
        ),
        createStore
      );
    }

    // 不使用 compose 来写是这样子：
    //
    // finalCreateStore =
    //   applyMiddleware(middleware)(
    //     devTools()(
    //       persistState(window.location.href.match(/[^&]+debug_se
    //     createStore
    //   )
    // )
    // );
  }

  let store = finalCreateStore(reducer);
}
```

小贴士

- `compose` 做的只是让你不使用深度右括号的情况下写深度嵌套的函数。不要觉得它很复杂。

React Redux

译者注：本库并不是 Redux 内置，需要单独安装。因为一般会和 Redux 一起使用，所以放到一起翻译

[Redux](#) 官方提供的 React 绑定库。
具有高效且灵活的特性。

安装

React Redux 依赖 [React 0.14](#) 或更新版本。

```
npm install --save react-redux
```

你需要使用 [npm](#) 作为包管理工具，配合 [Webpack](#) 或 [Browserify](#) 作为模块打包工具来加载 [CommonJS](#) 模块。

如果你不想使用 [npm](#) 和模块打包工具，只想打包一个 [UMD](#) 文件来提供 [ReactRedux](#) 全局变量，那么可以使用 [cdnjs](#) 上打包好的版本。但对于非常正式的项目并不建议这么做，因为和 Redux 一起工作的大部分库都只有 [npm](#) 才能提供。

React Native

在 [React Native](#) 支持 [React 0.14](#) 前，你需要使用 [React Redux 3.x](#) 分支和对应文档。

文档

- [快速入门](#)

- API
 - `<Provider store>`
 - `connect([mapStateToProps], [mapDispatchToProps], [mergeProps], [options])`
- 排错

快速开始

本库深受 [分离容器组件和展示组件](#) 思想启发。

在应用中，只有最顶层组件是对 Redux 可知（例如路由处理）这是很好的。所有它们的子组件都应该是“笨拙”的，并且是通过 props 获取数据。

	容器组件	展示组件
位置	最顶层，路由处理	中间和子组件
使用 Redux	是	否
读取数据	从 Redux 获取 state	从 props 获取数据
修改数据	向 Redux 发起 actions	从 props 调用回调函数

不使用 Redux 的展示组件

让我们看下，我们拥有一个 `<Counter />` 的展示组件，它有一个通过 props 传过来的值，和一个函数 `onIncrement`，当你点击“Increment”按钮时就会调用这个函数：

```
import { Component } from 'react';

export default class Counter extends Component {
  render() {
    return (
      <button onClick={this.props.onIncrement}>
        {this.props.value}
      </button>
    );
  }
}
```

容器组件使用 `connect()` 方法连接 Redux

以下为你说明如何连接到 Redux Store。

我们用 `react-redux` 提供的 `connect()` 方法将“笨拙”的 `Counter` 转化成容器组件。`connect()` 允许你从 Redux store 中指定准确的 state 到你想要获取的组件中。这让你能获取到任何级别颗粒度的数据。

`containers/CounterContainer.js`

```
import { Component } from 'react';
import { connect } from 'react-redux';

import Counter from '../components/Counter';
import { increment } from '../actionsCreators';

// 哪些 Redux 全局的 state 是我们组件想要通过 props 获取的?
function mapStateToProps(state) {
  return {
    value: state.counter
  };
}

// 哪些 action 创建函数是我们想要通过 props 获取的?
function mapDispatchToProps(dispatch) {
  return {
    onIncrement: () => dispatch(increment())
  };
}

export default connect(
  mapStateToProps,
  mapDispatchToProps
)(Counter);

// 你可以传递一个对象，而不是定义一个 `mapDispatchToProps`:
// export default connect(mapStateToProps, CounterActionCreators);

// 或者如果你想省略 `mapDispatchToProps`，你可以通过传递一个 `dispatch`:
// export default connect(mapStateToProps)(Counter);

// 想看到更多的方法，详细的 connect() 示例如下。
```

作为一个展示组件，无论是在同一个文件中调用 `connect()` 方法，还是分开调用，都取决于你。你应该多考虑的是，是否重用这个组件来绑定不同数据。

嵌套

在你的应用任何层次中，你可以拥有很多使用 `connect()` 的组件，甚至你可以把它们嵌套使用。的确如此，但是更好的做法是只在最顶层的组件中使用 `connect()`，例如路由处理，这使得应用中的数据流是保持可预知的。

修饰器的支持

你可能会注意到，我们在调用 `connect()` 方法的时候使用了两个括号。这个叫作局部调用，并且这允许开发者使用 ES7 提供的修饰语法：

```
// 这是还不稳定的语法！这可能在实际的应用中被修改或摒弃。  
@connect(mapStateToProps)  
export default class CounterContainer { ... }
```

不要忘了修饰器还在实验中！在以下的示范中，它们被提取出来，作为一个可以在任何地方调用的函数例子。

额外的灵活性

这是最基础的用法，但 `connect()` 也支持很多其他的模式：通过传递一个普通的 `dispatch()` 方法，绑定多个 `action` 创建函数，把它们传递到一个 `action prop` 中，选择一部分 `state` 和绑定的 `action` 创建函数依赖到 `props` 中，等等。了解更多请看下面的 `connect()` 文档。

注入 Redux Store

最后，我们实际上是怎么连接到 Redux store 的呢？我们需要在根组件中创建这个 store。对于客户端应用而言，根组件是一个很好的地方。对于服务端渲染而言，你可以在处理请求中完成这个。

关键是从 React Redux 将整个视图结构包装进 `<Provider>`。

```
import ReactDOM from 'react-dom';
import { Component } from 'react';
import { Provider } from 'react-redux';

class App extends Component {
  render() {
    // ...
  }
}

const targetEl = document.getElementById('root');

ReactDOM.render(
  <Provider store={store}>
    <App />
  </Provider>,
  targetEl
);
```

API

<Provider store>

<Provider store> 使组件层级中的 `connect()` 方法都能够获得 Redux store。正常情况下，你的根组件应该嵌套在 <Provider> 中才能使用 `connect()` 方法。

如果你真的不想把根组件嵌套在 <Provider> 中，你可以把 `store` 作为 `props` 传递到每一个被 `connect()` 包装的组件，但是我们只推荐您在单元测试中对 `store` 进行伪造 (stub) 或者在非完全基于 React 的代码中才这样做。正常情况下，你应该使用 <Provider>。

属性

- `store` ([Redux Store](#)): 应用程序中唯一的 Redux store 对象
- `children` ([ReactElement](#)) 组件层级的根组件。

例子

React

```
ReactDOM.render(  
  <Provider store={store}>  
    <MyRootComponent />  
  </Provider>,  
  rootEl  
>);
```

React Router 0.13

```
Router.run(routes, Router.HistoryLocation, (Handler, routerState)  
  ReactDOM.render(  
    <Provider store={store}>
```

```
    {/* 注意这里的 "routerState": 该变量应该传递到子组件 */}
    <Handler routerState={routerState} />
  </Provider>,
  document.getElementById('root')
);
});
```

React Router 1.0

```
ReactDOM.render(
  <Provider store={store}>
    <Router history={history}>...</Router>
  </Provider>,
  targetEl
);
```

**connect([mapStateToProps],
[mapDispatchToProps], [mergeProps],
[options])**

连接 React 组件与 Redux store。

连接操作不会改变原来的组件类，反而返回一个新的已与 Redux store 连接的组件类。

参数

- [`mapStateToProps(state, [ownProps]): stateProps`] (*Function*): 如果定义该参数，组件将会监听 Redux store 的变化。任何时候，只要 Redux store 发生改变，`mapStateToProps` 函数就会被调用。该回调函数必须返回一个纯对象，这个对象会与组件的 props 合并。如果你省略了这个参数，你的组件将不会监听 Redux store。如果指定了该回调函数中的第二个参数 `ownProps`，则该参数的值为传递到组件的 props，而且只要组件接收到新的 props，`mapStateToProps` 也会被调用。

- [`mapDispatchToProps(dispatch, [ownProps]): dispatchProps`] (*Object or Function*): 如果传递的是一个对象，那么每个定义在该对象的函数都将被当作 Redux action creator，而且这个对象会与 Redux store 绑定在一起，其中所定义的方法名将作为属性名，合并到组件的 props 中。如果传递的是一个函数，该函数将接收一个 `dispatch` 函数，然后由你来决定如何返回一个对象，这个对象通过 `dispatch` 函数与 action creator 以某种方式绑定在一起（提示：你也许会用到 Redux 的辅助函数 `bindActionCreators()`）。如果你省略这个 `mapDispatchToProps` 参数，默认情况下，`dispatch` 会注入到你的组件 props 中。如果指定了该回调函数中第二个参数 `ownProps`，该参数的值为传递到组件的 props，而且只要组件接收到新 props，`mapDispatchToProps` 也会被调用。
- [`mergeProps(stateProps, dispatchProps, ownProps): props`] (*Function*): 如果指定了这个参数，`mapStateToProps()` 与 `mapDispatchToProps()` 的执行结果和组件自身的 `props` 将传入到这个回调函数中。该回调函数返回的对象将作为 `props` 传递到被包装的组件中。你也许可以用这个回调函数，根据组件的 `props` 来筛选部分的 state 数据，或者把 `props` 中的某个特定变量与 action creator 绑定在一起。如果你省略这个参数，默认情况下返回 `Object.assign({}, ownProps, stateProps, dispatchProps)` 的结果。
- [`options`] (*Object*) 如果指定这个参数，可以定制 connector 的行为。
 - [`pure = true`] (*Boolean*): 如果为 `true`，connector 将执行 `shouldComponentUpdate` 并且浅对比 `mergeProps` 的结果，避免不必要的更新，前提是当前组件是一个“纯”组件，它不依赖于任何的输入或 state 而只依赖于 props 和 Redux store 的 state。默认值为 `true`。
 - [`withRef = false`] (*Boolean*): 如果为 `true`，connector 会保存一个对被包装组件实例的引用，该引用通过 `getWrappedInstance()` 方法获得。默认值为 `false`

返回值

根据配置信息，返回一个注入了 state 和 action creator 的 React 组件。

静态属性

- `WrappedComponent` (*Component*): 传递到 `connect()` 函数的原始组件类。

静态方法

组件原来的静态方法都被提升到被包装的 React 组件。

实例方法

`getWrappedInstance(): ReactComponent`

仅当 `connect()` 函数的第四个参数 `options` 设置了 `{ withRef: true }` 才返回被包装的组件实例。

备注

- 函数将被调用两次。第一次是设置参数，第二次是组件与 Redux store 连接：`connect(mapStateToProps, mapDispatchToProps, mergeProps)(MyComponent)`。
- `connect` 函数不会修改传入的 React 组件，返回的是一个新的已与 Redux store 连接的组件，而且你应该使用这个新组件。
- `mapStateToProps` 函数接收整个 Redux store 的 state 作为 props，然后返回一个传入到组件 props 的对象。该函数被称之为 **selector**。参考使用 [reselect](#) 高效地组合多个 **selector**，并对 [收集到的数据进行处理](#)。

Examples 例子

只注入 `dispatch`，不监听 `store`

```
export default connect()(TodoApp);
```

注入 `dispatch` 和全局 `state`

不要这样做！这会导致每次 `action` 都触发整个 `TodoApp` 重新渲染，你做的所有性能优化都将付之东流。

最好在多个组件上使用 `connect()`，每个组件只监听它所关联的部分 `state`。

```
export default connect(state => state)(TodoApp);
```

注入 `dispatch` 和 `todos`

```
function mapStateToProps(state) {
  return { todos: state.todos };
}
```

```
export default connect(mapStateToProps)(TodoApp);
```

注入 `todos` 和所有 `action creator` (`addTodo`, `completeTodo`, ...)

```
import * as actionCreators from './actionCreators';

function mapStateToProps(state) {
  return { todos: state.todos };
}

export default connect(mapStateToProps, actionCreators)(TodoApp);
```

注入 `todos` 并把所有 `action creator` 作为 `actions` 属性也注入组件中

```
import * as actionCreators from './actionCreators';
import { bindActionCreators } from 'redux';

function mapStateToProps(state) {
```

```
    return { todos: state.todos };
}

function mapDispatchToProps(dispatch) {
  return { actions: bindActionCreators(actionCreators, dispatch) };
}

export default connect(mapStateToProps, mapDispatchToProps)(TodosList);
```

注入 `todos` 和指定的 `action creator` (`addTodo`)

```
import { addTodo } from './actionCreators';
import { bindActionCreators } from 'redux';

function mapStateToProps(state) {
  return { todos: state.todos };
}

function mapDispatchToProps(dispatch) {
  return bindActionCreators({ addTodo }, dispatch);
}

export default connect(mapStateToProps, mapDispatchToProps)(TodosList);
```

注入 `todos` 并把 `todoActionCreators` 作为 `todoActions` 属性、 `counterActionCreators` 作为 `counterActions` 属性注入到组件中

```
import * as todoActionCreators from './todoActionCreators';
import * as counterActionCreators from './counterActionCreators';
import { bindActionCreators } from 'redux';

function mapStateToProps(state) {
  return { todos: state.todos };
}

function mapDispatchToProps(dispatch) {
  return {
```

```
    todoActions: bindActionCreators(todoActionCreators, dispatch),
    counterActions: bindActionCreators(counterActionCreators, dispatch)
};

export default connect(mapStateToProps, mapDispatchToProps)(TodosList);
```

注入 `todos` 并把 `todoActionCreators` 与 `counterActionCreators` 一同作为 `actions` 属性注入到组件中

```
import * as todoActionCreators from './todoActionCreators';
import * as counterActionCreators from './counterActionCreators';
import { bindActionCreators } from 'redux';

function mapStateToProps(state) {
  return { todos: state.todos };
}

function mapDispatchToProps(dispatch) {
  return {
    actions: bindActionCreators(Object.assign({}, todoActionCreators,
      counterActionCreators), dispatch)
  };
}

export default connect(mapStateToProps, mapDispatchToProps)(TodosList);
```

注入 `todos` 并把所有的 `todoActionCreators` 和 `counterActionCreators` 作为 `props` 注入到组件中

```
import * as todoActionCreators from './todoActionCreators';
import * as counterActionCreators from './counterActionCreators';
import { bindActionCreators } from 'redux';

function mapStateToProps(state) {
  return { todos: state.todos };
}
```

```
function mapDispatchToProps(dispatch) {
  return bindActionCreators(Object.assign({}, todoActionCreators), dispatch)
}

export default connect(mapStateToProps, mapDispatchToProps)(TodoApp);
```

根据组件的 `props` 注入特定用户的 `todos`

```
import * as actionCreators from './actionCreators';

function mapStateToProps(state, ownProps) {
  return { todos: state.todos[ownProps.userId] };
}

export default connect(mapStateToProps)(TodoApp);
```

根据组件的 `props` 注入特定用户的 `todos` 并把 `props.userId` 传入到 `action` 中

```
import * as actionCreators from './actionCreators';

function mapStateToProps(state) {
  return { todos: state.todos };
}

function mergeProps(stateProps, dispatchProps, ownProps) {
  return Object.assign({}, ownProps, {
    todos: stateProps.todos[ownProps.userId],
    addTodo: (text) => dispatchProps.addTodo(ownProps.userId, text)
  });
}

export default connect(mapStateToProps, actionCreators, mergeProps)(TodoApp);
```


排错

开始之前，一定你已经学习 [Redux 排错](#)。

View 不更新的问题

阅读上面的链接。简而言之，

- Reducer 永远不应该更改原有 state，应该始终返回新的对象，否则，React Redux 觉察不到数据变化。
- 确保你使用了 `connect()` 的 `mapDispatchToProps` 参数或者 `bindActionCreators` 来绑定 action creator 函数，你也可以手动调用 `dispatch()` 进行绑定。直接调用 `MyActionCreators.addTodo()` 并不会起任何作用，因为它只会返回一个 action 对象，并不会 *dispatch* 它。

React Router 0.13 的 route 变化中，view 不更新

如果你正在使用 React Router 0.13，你可能会[碰到这样的问题](#)。解决方法很简单：当使用 `<RouteHandler>` 或者 `Router.run` 提供的 `Handler` 时，不要忘记传递 router state。

根 View：

```
Router.run(routes, Router.HistoryLocation, (Handler, routerState) =>
  ReactDOM.render(
    <Provider store={store}>
      {/* 注意这里的 "routerState" */}
      <Handler routerState={routerState} />
    </Provider>,
    document.getElementById('root')
  );
});
```

嵌套 view:

```
render() {
  // 保持这样传递下去
  return <RouteHandler routerState={this.props.routerState} />;
}
```

很方便地，这样你的组件就能访问 router 的 state 了！当然，你可以将 React Router 升级到 1.0，这样就不会有此问题了。（如果还有问题，联系我们！）

Redux 外部的一些东西更新时，view 不更新

如果 view 依赖全局的 state 或是 [React “context”](#)，你可能发现那些使用 `connect()` 进行修饰的 view 无法更新。

这是因为，默认情况下 `connect()` 实现了 [shouldComponentUpdate](#)，它假定在 `props` 和 `state` 一样的情况下，组件会渲染出同样的结果。这与 React 中 [PureRenderMixin](#) 的概念很类似。

这个问题的最好的解决方案是保持组件的纯净，并且所有外部的 state 都应通过 `props` 传递给它们。这将确保组件只在需要重新渲染时才会重新渲染，这将大大地提高了应用的速度。

当不可抗力导致上述解法无法实现时（比如，你使用了严重依赖 React context 的外部库），你可以设置 `connect()` 的 `pure: false` 选项：

```
function mapStateToProps(state) {
  return { todos: state.todos };
}

export default connect(mapStateToProps, null, null, {
  pure: false
})
```

```
})(TodoApp);
```

这样就表示你的 `TodoApp` 不是纯净的，只要父组件渲染，自身都会重新渲染。注意，这会降低应用的性能，所以只有在别无他法的情况下才使用它。

在 `context` 或 `props` 中都找不到 “store”

如果你有 `context` 的问题，

1. 确保你没有引入多个 React 实例 到页面上。
2. 确保你没有忘记将根组件包装进 `<Provider>`。
3. 确保你运行的 React 和 React Redux 是最新版本。

Invariant Violation:

`addComponentAsRefTo(...)`: 只有 `ReactOwner` 才有 `refs`。这通常意味着你在一个没有 `owner` 的组件中添加了 `ref`

如果你在 web 中使用 React，就通常意味着你[引用了两遍 React](#)。按照这个链接解决即可。