Table of Contents

Redux 中文文档 gitter join chat

Redux

> 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
> React 核心开发者写的 React 设计思想
>
> :arrow_down: 离线下载：pdf 格式，epub 格式，mobi 格式

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

开始之前

> 也推荐阅读你可能并不需要Redux：
> “You Might Not Need Redux”

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，但同时做到行为的完全可预测，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
```

以上基于使用 npm 来做包管理工具的情况下。

否则你可以直接在 unpkg 上访问这些文件，下载下来，或者把让你的包管理工具指向它。

一般情况下人们认为 Redux 就是一些 CommonJS 模块的集合。这些模块就是你在使用 Webpack、Browserify、或者 Node 环境时引入的。如果你想追求时髦并使用 Rollup，也是支持的。

你也可以不使用模块打包工具。`redux` 的 npm 包里 `dist` 目录包含了预编

译好的生产环境和开发环境下的 UMD 文件。可以直接使用，而且支持大部分流行的 JavaScript 包加载器和环境。比如，你可以直接在页面上的 `<script>` 标签 中引入 UMD 文件，也可以让 `Bower` 来安装。UMD 文件可以让你使用 `window.Redux` 全局变量来访问 Redux。

Redux 源文件由 ES2015 编写，但是会预编译到 CommonJS 和 UMD 规范的 ES5，所以它可以支持 任何现代浏览器。你不必非得使用 Babel 或模块打包器来使用 Redux。

附加包

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
```

需要提醒的是，和 Redux 不同，很多 Redux 生态下的包并不提供 UMD 文件，所以为了提升开发体验，我们建议使用像 Webpack 和 Browserify 这样的 CommonJS 模块打包器。

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。
惟一改变 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。

Redux 作者教你学

Redux 入门 是由 Redux 作者 Dan Abramov 讲述的包含 30 个视频的课程。它涵盖了本文档的“基础”部分，同时还有不可变（immutability）、测试、Redux 最佳实践、搭配 React 使用的讲解。**这个课程将永久免费。**

还等什么?

开始观看 30 个免费视频!

文档

- 介绍
- 基础
- 高级
- 技巧
- 常见问题
- 排错
- 词汇表
- API 文档

示例

- 原生 Counter (source)
- Counter (source)
- Todos (source)
- 可撤销的 Todos (source)
- TodoMVC (source)
- 购物车 (source)
- Tree View (source)
- 异步 (source)
- Universal (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。

作者列表

> 定期更新，谢谢各位辛勤贡献

- Cam Song 会影@camsong
- Jovey Zheng@jovey-zheng
- Pandazki@pandazki
- Shangbin Yang@rccoder
- Doray Hong@dorayx
- Yuwei Wang@yuweiw823
- Yudong@hashtree
- Arcthur@arcthur
- Desen Meng@demohi
- Zhe Zhang@zhe
- alcat2008
- Frozenme
- 姜杨军@Yogi-Jiang
- Byron Bai@happybai
- 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

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。前端开发者正在经受前所未有的复杂性，难道就这么放弃了吗？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理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。

核心概念

Redux 本身很简单。

当使用普通对象来描述应用的 state 时。例如，todo 应用的 state 可能长这样：

```
{
  todos: [{
    text: 'Eat food',
    completed: true
  }, {
    text: 'Exercise',
    completed: false
  }],
  visibilityFilter: 'SHOW_COMPLETED'
}
```

这个对象就像 “Model”，区别是它并没有 setter（修改器方法）。因此其它的代码不能随意修复它，造成难以复现的 bug。

要想更新 state 中的数据，你需要发起一个 action。Action 就是一个普通 JavaScript 对象（注意到没，这儿没有任何魔法？）用来描述发生了什么。下面是一些 action 的示例：

```
{ type: 'ADD_TODO', text: 'Go to swimming pool' }
{ type: 'TOGGLE_TODO', index: 1 }
{ type: 'SET_VISIBILITY_FILTER', filter: 'SHOW_ALL' }
```

强制使用 action 来描述所有变化带来的好处是可以清晰地知道应用中到底发生了什么。如果一些东西改变了，就可以知道为什么变。action 就像是描述发生了什么的面包屑。最终，为了把 action 和 state 串起来，开发一些函数，这就是 reducer。再次地，没有任何魔法，reducer 只是一个接收 state

和 action，并返回新的 state 的函数。 对于大的应用来说，可能很难开发这样的函数，所以我们编写很多小函数来分别管理 state 的一部分：

```
function visibilityFilter(state = 'SHOW_ALL', action) {
  if (action.type === 'SET_VISIBILITY_FILTER') {
    return action.filter;
  } else {
    return state;
  }
}

function todos(state = [], action) {
  switch (action.type) {
  case 'ADD_TODO':
    return state.concat([{ text: action.text, completed: false }]);
  case 'TOGGLE_TODO':
    return state.map((todo, index) =>
      action.index === index ?
        { text: todo.text, completed: !todo.completed } :
        todo
   )
  default:
    return state;
  }
}
```

再开发一个 reducer 调用这两个 reducer，进而来管理整个应用的 state：

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

这差不多就是 Redux 思想的全部。注意到没我们还没有使用任何 Redux 的

API。Redux 里有一些工具来简化这种模式，但是主要的想法是如何根据这些 action 对象来更新 state，而且 90% 的代码都是纯 JavaScript，没用 Redux、Redux API 和其它魔法。

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())

/* 输出
{
  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.map((todo, index) => {
        if (index === action.index) {
          return Object.assign({}, todo, {
            completed: true
          })
        }
        return todo
      })
    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，如果你想确认。）

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。正确且简便的方式是，你应该在 reducer 中返回一个新对象来更新 state， 同时配合 object spread 运算符提案 或一些库，如 Immutable。

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。 Elm 的 “updater” 与 Redux 里的 reducer 服务于相同的目的。

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。要想学习 React 和 Redux，React/Redux Links 包含了教程和不少有用的资源，Redux Ecosystem Links 则列出了 许多 Redux 相关的库及插件。

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** — 向作者学习 Redux 基础知识（30 个免费的教学视频）
- 学习 **Redux** — 搭建一个简单的图片应用，简要使用了 Redux、React Router 和 React.js 的核心思想

示例应用

- 官方示例 — 一些官方示例，涵盖了多种 Redux 技术
- SoundRedux — 用 Redux 构建的 SoundCloud 客户端
- grafgiti — 在你的 Github 的 Contributor 页上创建 graffiti
- React-lego — 如何像积木一样，一块块地扩展你的 Redux 技术栈

教程与文章

- Redux 教程
- Redux Egghead 课程笔记
- 使用 React Native 进行数据整合
- What the Flux?! Let's Redux.
- Leveling Up with React: Redux
- A cartoon intro to Redux
- Understanding Redux
- Handcrafting an Isomorphic Redux Application (With Love)
- Full-Stack Redux Tutorial
- Getting Started with React, Redux, and Immutable
- Secure Your React and Redux App with JWT Authentication
- Understanding Redux Middleware
- Angular 2 — Introduction to Redux
- Apollo Client: GraphQL with React and Redux
- Using redux-saga To Simplify Your Growing React Native Codebase
- Build an Image Gallery Using Redux Saga
- Working with VK API (in Russian)

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
- Redux: Simplifying Application State — Redux 架构介绍

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
- polymer-redux - Polymer
- ember-redux - Ember.js

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-axios-middleware — 使用 axios HTTP 客户端获取数据的 Redux 中间件
- redux-observable — Redux 的 RxJS 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-unhandled-action — 开发过程中，若 Action 未使 State 发生变化则发出警告
- redux-analytics — Redux middleware 分析
- redux-gen — Redux middleware 生成器
- redux-saga — Redux 应用的另一种副作用 model
- redux-action-tree — Redux 的可组合性 Cerebral-style 信号
- apollo-client — 针对 GraphQL 服务器及基于 Redux 的 UI 框架的缓存客户端

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
- react-redux-form — 在 React 中使用 Redux 生成表格

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 — 使 action 可稍后提交或撤销
- redux-optimistic-ui — A reducer enhancer to enable type-agnostic optimistic updates 允许对未知类型进行更新的 reducer 增强器
- redux-undo — 使 reducer 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
- redux-search — 自动 index 站点资源并实现即时搜索
- redux-electron-store — Store 增强器， 可同步不同 Electron 进程上的多个 Redux store
- redux-loop — Sequence effects purely and naturally by returning them from your reducers
- redux-side-effects — Utilize Generators for declarative yielding of side effects from your pure reducers

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返回的嵌套数据范式化
- 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
- redux-actions-assertions — Redux actions 测试断言

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
- Redux DevTools Extension — 打包了 Redux DevTools 及附加功能的 Chrome 插件

开发者工具监听器

- Log Monitor — Redux DevTools 默认监听器，提供树状视图
- Dock Monitor — A resizable and movable dock for Redux DevTools monitors
- Slider Monitor — Redux DevTools 自定义监听器，可回放被记录的 Redux action
- Inspector — Redux DevTools 自定义监听器，可筛选、区分 action，深入 state 并监测变化
- Diff Monitor — 区分不同 action 的 store 变动的 Redux Devtools 监听器
- Filterable Log Monitor — 树状可筛选视图的 Redux DevTools 监听器
- Chart Monitor — Redux DevTools 图表监听器
- Filter Actions — 可筛选 action 、可组合使用的 Redux DevTools 监听器

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

翻译

- 中文文档 — 简体中文
- 繁體中文文件 — 繁体中文
- Redux in Russian — 俄语
- Redux en Español - 西班牙语

更多

- Awesome Redux 是一个包含大量与 Redux 相关的库列表。
- React-Redux Links React、Redux、ES6 的高质量文章、教程、及相关内容列表。
- Redux Ecosystem Links Redux 相关库、插件、工具集的分类资源。

示例

Redux 源码 中同时包含了一些示例。

原生版 Counter

运行 Counter Vanilla 示例:

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

cd redux/examples/counter-vanilla
open index.html
```

该示例不需搭建系统或视图框架，展示了基于 ES5 的原生 Redux API。

Counter

运行 Counter 示例:

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

cd redux/examples/counter
npm install
npm start

open http://localhost:3000/
```

Redux 结合 React 使用的最基本示例。出于简化，当 store 发生变化，React 组件会手动重新渲染。在实际的项目中，可以使用 React 和 Redux 已绑定、且更高效的 React Redux。

该示例包含测试代码。

Todos

运行 Todos 示例：

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

cd redux/examples/todos
npm install
npm start

open http://localhost:3000/
```

深入理解在 Redux 中 state 的更新与组件是如何共同运作的例子。展示了 reducer 如何委派 action 给其它 reducer，也展示了如何使用 React Redux 从展示组件中生成容器组件。

该示例包含测试代码。

支持撤销的 Todos

运行 Todos-with-undo 示例:

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

cd redux/examples/todos-with-undo
npm install
npm start

open http://localhost:3000/
```

前一个示例的衍生。基本相同但额外展示了如何使用 Redux Undo 打包 reducer，仅增加几行代码实现撤销/重做功能。

TodoMVC

运行 TodoMVC 示例:

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

cd redux/examples/todomvc
npm install
npm start

open http://localhost:3000/
```

经典的 TodoMVC 示例。与 Todos 示例的目的相同，为了两者间比较罗列在此。

该示例包含测试代码。

购物车

运行 Shopping Cart 示例:

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

cd redux/examples/shopping-cart
npm install
npm start

open http://localhost:3000/
```

该示例展示了随着应用升级变得愈发重要的常用的 Redux 模式。尤其展示了，如何使用 ID 来标准化存储数据实体，如何在不同层级将多个 reducer 组合使用，如何利用 reducer 定义选择器以封装 state 的相关内容。该示例也展示了使用 Redux Logger 生成日志，以及使用 Redux Thunk 中间件进行 action 的条件性分发。

树状视图

运行 Tree View 示例:

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

cd redux/examples/tree-view
npm install
npm start

open http://localhost:3000/
```

该示例展示了深层嵌套树状视图的渲染，以及为了方便 reducer 更新，state 的标准化写法。优良的渲染表现，来自于容器组件细粒度的、仅针对需要渲染的 tree node 的绑定。

该示例包含测试代码。

异步

运行 Async 示例：

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

cd redux/examples/async
npm install
npm start

open http://localhost:3000/
```

该示例包含了：从异步 API 的读取操作、基于用户的输入来获取数据、显示正在加载的提示、缓存响应、以及使缓存过期失效。使用 Redux Thunk 中间件来封装异步带来的附带作用。

同构

运行 Universal 示例:

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

cd redux/examples/universal
npm install
npm start

open http://localhost:3000/
```

展示了基于 Redux 和 React 的 server rendering。怎样在服务器端准备 store 中的初始 state 并传递到客户端，使客户端中的 store 可以从现有的 state 启动。

真实场景

运行 Real World 示例：

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

cd redux/examples/real-world
npm install
npm start

open http://localhost:3000/
```

最为高阶的示例。浓缩化的设计。包含了持续性地从标准化缓存中批量获取数据实例，针对 API 调用的自定义中间件的实现，逐步渲染已加载的数据、分页器、缓存响应，展示错误信息，以及路由。同时，包含了 Redux DevTools 的使用。

更多示例

前往 Awesome Redux 获取更多 Redux 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

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 内必须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 获取更多保持代码简洁的实践经验。

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 获取关于如何构造 action 的建议。

这时，我们还需要再添加一个 action type 来表示用户完成任务的动作。因为数据是存放在数组中的，所以我们通过下标 `index` 来引用特定的任务。而实际项目中一般会在新建数据的时候生成唯一的 ID 作为数据的引用标识。

```
{
  type: TOGGLE_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 type 来表示当前的任务展示选项。

```
{
  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 Redux 中的 action 创建函数只是简单的返回一个 action:

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

```
}
```

这样做将使 action 创建函数更容易被移植和测试。

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 中只需把 action 创建函数的结果传给 `dispatch()` 方法即可发起一次 dispatch 过程。

```
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。

Action 创建函数也可以是异步非纯函数。你可以通过阅读 高级教程 中的 异步 action章节，学习如何处理 AJAX 响应和如何把 action 创建函数组合进异步控制流。因为基础教程中包含了阅读高级教程和异步 action 章节所需要的一些重要基础概念, 所以请在移步异步 action 之前, 务必先完成基础教程。

源码

actions.js

```
/*
 * action 类型
 */

export const ADD_TODO = 'ADD_TODO';
export const TOGGLE_TODO = 'TOGGLE_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 toggleTodo(index) {
  return { type: TOGGLE_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 来说明在发起 action 后 state 应该如何更新。

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，需要保存两种不同的数据：

- 当前选中的任务过滤条件；
- 完整的任务列表。

通常，这个 state 树还需要存放其它一些数据，以及一些 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，不同实体或列表间通过 ID 相互引用数据。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 就是一个纯函数，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。**只要传入参数相同，返回计算得到的下一个 state 就一定相同。没有特殊情况、没有副作用，没有 API 请求、没有变量修改，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 action。

我们将以指定 state 的初始状态作为开始。Redux 首次执行时，state 为 `undefined`，此时我们可借机设置并返回应用的初始 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。你也可以开启对ES7提案对象展开运算符的支持, 从而使用 `{ ...state, ...newState }` 达到相同的目的。

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` 须知**
>
> `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` 和样板代码须知**
>
> `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， `TOGGLE_TODO` 的实现也很好理解：

```
case TOGGLE_TODO:
  return Object.assign({}, state, {
    todos: state.todos.map((todo, index) => {
      if (index === action.index) {
        return Object.assign({}, todo, {
          completed: !todo.completed
        })
      }
      return todo
    })
  })
```

我们需要修改数组中指定的数据项而又不希望导致**突变**, 因此我们的做法是在创建一个新的数组后, 将那些无需修改的项原封不动移入, 接着对需修改的项用新生成的对象替换。(译者注：Javascript中的对象存储时均是由值和指向值的引用两个部分构成。此处**突变**指直接修改引用所指向的值, 而引用本身保持不变。) 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长：

```
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 TOGGLE_TODO:
      return Object.assign({}, state, {
        todos: state.todos.map((todo, index) => {
          if(index === action.index) {
            return Object.assign({}, todo, {
              completed: !todo.completed
            })
          }
          return todo
        })
```

```
      })
    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 TOGGLE_TODO:
      return state.map((todo, index) => {
        if (index === action.index) {
          return Object.assign({}, todo, {
            completed: !todo.completed
          })
        }
        return todo
      })
    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 TOGGLE_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，如果传入 `undefined`，子 reducer 将负责返回它们的默认值。

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

```
      return [
        ...state,
        {
          text: action.text,
          completed: false
        }
      ]
    case TOGGLE_TODO:
      return state.map((todo, index) => {
        if (index === action.index) {
          return Object.assign({}, todo, {
            completed: !todo.completed
          })
        }
        return todo
      })
    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，我们还可以将拆分后的 reducer 放到不同的文件中, 以保持其独立性并用于专门处理不同的数据域。

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再将所有 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，我们不会在本文档中使用它。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, TOGGLE_TODO, SET_VISIBILITY_FILTER, VisibilityFilt
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 TOGGLE_TODO:
      return state.map((todo, index) => {
        if (index === action.index) {
          return Object.assign({}, todo, {
            completed: !todo.completed
          })
        }
        return todo
      })
    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;
- 通过 `subscribe(listener)` 返回的函数注销监听器。

再次强调一下 **Redux 应用只有一个单一的 store**。当需要拆分数据处理逻辑时，你应该使用 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()` 的第二个参数是可选的, 用于设置 state 初始状态。这对开发同构应用时非常有用，服务器端 redux 应用的 state 结构可以与客户端保持一致, 那么客户端可以将从网络接收到的服务端 state 直接用于本地数据初始化。

```
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，我们已经可以测试数据处理逻辑了。

```
import { addTodo, toggleTodo, setVisibilityFilter, VisibilityFilters

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

// 每次 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(toggleTodo(0))
store.dispatch(toggleTodo(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: 'Mary' } }
   { 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.visibleTodoF
```

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 绑定库是基于 容器组件和展示组件相分离 的开发思想。所以建议先读完这篇文章再回来继续学习。

已经读完了？那让我们再总结一下不同点：

	展示组件	容器组件
作用	描述如何展现（骨架、样式）	描述如何运行（数据获取、状态更新）
直接使用 Redux	否	是

数据来源	props	监听 Redux state
数据修改	从 props 调用回调函数	向 Redux 派发 actions
调用方式	手动	通常由 React Redux 生成

大部分的组件都应该是展示型的，但一般需要少数的几个容器组件把它们和 Redux store 连接起来。

技术上讲你可以直接使用 `store.subscribe()` 来编写容器组件。但不建议这么做因为就无法使用 React Redux 带来的性能优化。也因此，不要手写容器组件，都是使用 React Redux 的 `connect()` 方法来生成，后面会详细介绍。

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。

展示组件

以下的这些组件（和它们的 props ）就是从这个设计里来的：

- **`TodoList`** 用于显示 todos 列表。
  - `todos: Array` 以 `{ 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。
- **`Link`** 带有 callback 回调功能的链接
    - `onClick()` 当点击链接时会触发
- **`Footer`** 一个允许用户改变可见 todo 过滤器的组件。
- **`App`** 根组件，渲染余下的所有内容。

这些组件只定义外观并不设计数据从哪里来，如果改变它。传入什么就渲染什么。如果你把代码从 Redux 迁移到别的架构，这些组件可以不做任何改动直接使用。它们并不依赖于 Redux。

容器组件

还需要一些容器组件来把展示组件连接到 Redux。例如，展示型的 `TodoList` 组件需要一个类似 `VisibleTodoList` 的容器来监听 Redux store 变化并处理如何过滤出要显示的数据。为了实现状态过滤，需要实现 `FilterLink` 的容器组件来渲染 `Link` 并在点击时触发对应的 action：

- **`VisibleTodoList`** 根据当前显示的状态来对 todo 列表进行过滤，并渲染 `TodoList`。
- **`FilterLink`** 得到当前过滤器并渲染 `Link`。
    - `filter: string` 就是当前过滤的状态

其它组件

有时很难分清到底该使用容器组件还是展示组件。例如，有时表单和函数严重耦合在一起，如这个小的组件：

- **`AddTodo`** 含有“Add”按钮的输入框

技术上讲可以把它分成两个组件，但一开始就这么做有点早。在一些非常小的组件里混用容器和展示是可以的。当业务变复杂后，如何拆分就很明显了。所以现在就使用混合型的中。

组件编码

终于开始开发组件了！先做展示组件，这样可以先不考虑 Redux。

展示组件

它们只是普通的 React 组件，所以不会详细解释。我们会使用函数式无状态组件除非需要本地 state 或生命周期函数的场景。这并不是说展示组件必须是函数 -- 只是因为这样做容易些。如果你需要使用本地 state，生命周期方法，或者性能优化，可以所它们转成 class。

components/Todo.js

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

const Todo = ({ onClick, completed, text }) => (
  <li
    onClick={onClick}
    style={{
      textDecoration: completed ? 'line-through' : 'none'
    }}
  >
    {text}
  </li>
)

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

export default Todo
```

components/TodoList.js

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

```
const TodoList = ({ todos, onTodoClick }) => (
  <ul>
    {todos.map(todo =>
      <Todo
        key={todo.id}
        {...todo}
        onClick={() => onTodoClick(todo.id)}
      />
    )}
  </ul>
)

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

export default TodoList
```

components/Link.js

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

const Link = ({ active, children, onClick }) => {
  if (active) {
    return <span>{children}</span>
  }

  return (
    <a href="#"
       onClick={e => {
         e.preventDefault()
         onClick()
       }}
    >
      {children}
```

```
    </a>
  )
}

Link.propTypes = {
  active: PropTypes.bool.isRequired,
  children: PropTypes.node.isRequired,
  onClick: PropTypes.func.isRequired
}

export default Link
```

components/Footer.js

```
import React from 'react'
import FilterLink from '../containers/FilterLink'

const Footer = () => (
  <p>
    Show:
    {" "}
    <FilterLink filter="SHOW_ALL">
      All
    </FilterLink>
    {", "}
    <FilterLink filter="SHOW_ACTIVE">
      Active
    </FilterLink>
    {", "}
    <FilterLink filter="SHOW_COMPLETED">
      Completed
    </FilterLink>
  </p>
)

export default Footer
```

components/App.js

```
import React from 'react'
import Footer from './Footer'
import AddTodo from '../containers/AddTodo'
import VisibleTodoList from '../containers/VisibleTodoList'

const App = () => (
  <div>
    <AddTodo />
    <VisibleTodoList />
    <Footer />
  </div>
)

export default App
```

容器组件

现在来创建一些容器组件把这些展示组件和 Redux 关联起来。技术上讲，容器组件就是使用 `store.subscribe()` 从 Redux state 树中读取部分数据，并通过 props 来把这些数据提供给要渲染的组件。你可以手工来开发容器组件，但建议使用使用 React Redux 库的 `connect()` 方法来生成，这个方法做了性能优化来避免很多不必要的重复渲染。（这样你就不必为了性能而手动实现 React 性能优化建议 中的 `shouldComponentUpdate` 方法。）

使用 `connect()` 前，需要先定义 `mapStateToProps` 这个函数来指定如何把当前 Redux store state 映射到展示组件的 props 中。例如，`VisibleTodoList` 需要计算传到 `TodoList` 中的 `todos`，所以定义了根据 `state.visibilityFilter` 来过滤 `state.todos` 的方法，并在 `mapStateToProps` 中使用。

```
const getVisibleTodos = (todos, filter) => {
  switch (filter) {
    case 'SHOW_ALL':
      return todos
    case 'SHOW_COMPLETED':
      return todos.filter(t => t.completed)
```

```
    case 'SHOW_ACTIVE':
      return todos.filter(t => !t.completed)
  }
}

const mapStateToProps = (state) => {
  return {
    todos: getVisibleTodos(state.todos, state.visibilityFilter)
  }
}
```

除了读取 state，容器组件还能分发 action。类似的方式，可以定义 `mapDispatchToProps()` 方法接收 `dispatch()` 方法并返回期望注入到展示组件的 props 中的回调方法。例如，我们希望 `VisibleTodoList` 向 `TodoList` 组件中注入一个叫 `onTodoClick` 的 props 中，还希望 `onTodoClick` 能分发 `TOGGLE_TODO` 这个 action：

```
const mapDispatchToProps = (dispatch) => {
  return {
    onTodoClick: (id) => {
      dispatch(toggleTodo(id))
    }
  }
}
```

最后，使用 `connect()` 创建 `VisibleTodoList`，并传入这两个函数。

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

const VisibleTodoList = connect(
  mapStateToProps,
  mapDispatchToProps
)(TodoList)

export default VisibleTodoList
```

这就是 React Redux API 的基础，但还漏了一些快捷技巧和强大的配置。建议你仔细学习 它的文档。如果你担心 `mapStateToProps` 创建新对象太过频繁，可以学习如何使用 reselect 来 计算衍生数据。

其它容器组件定义如下：

containers/FilterLink.js

```
import { connect } from 'react-redux'
import { setVisibilityFilter } from '../actions'
import Link from '../components/Link'

const mapStateToProps = (state, ownProps) => {
  return {
    active: ownProps.filter === state.visibilityFilter
  }
}

const mapDispatchToProps = (dispatch, ownProps) => {
  return {
    onClick: () => {
      dispatch(setVisibilityFilter(ownProps.filter))
    }
  }
}

const FilterLink = connect(
  mapStateToProps,
  mapDispatchToProps
)(Link)

export default FilterLink
```

containers/VisibleTodoList.js

```
import { connect } from 'react-redux'
import { toggleTodo } from '../actions'
import TodoList from '../components/TodoList'
```

```
const getVisibleTodos = (todos, filter) => {
  switch (filter) {
    case 'SHOW_ALL':
      return todos
    case 'SHOW_COMPLETED':
      return todos.filter(t => t.completed)
    case 'SHOW_ACTIVE':
      return todos.filter(t => !t.completed)
  }
}

const mapStateToProps = (state) => {
  return {
    todos: getVisibleTodos(state.todos, state.visibilityFilter)
  }
}

const mapDispatchToProps = (dispatch) => {
  return {
    onTodoClick: (id) => {
      dispatch(toggleTodo(id))
    }
  }
}

const VisibleTodoList = connect(
  mapStateToProps,
  mapDispatchToProps
)(TodoList)

export default VisibleTodoList
```

其它组件

containers/AddTodo.js

```
import React from 'react'
import { connect } from 'react-redux'
import { addTodo } from '../actions'
```

```
let AddTodo = ({ dispatch }) => {
  let input

  return (
    <div>
      <form onSubmit={e => {
        e.preventDefault()
        if (!input.value.trim()) {
          return
        }
        dispatch(addTodo(input.value))
        input.value = ''
      }}>
        <input ref={node => {
          input = node
        }} />
        <button type="submit">
          Add Todo
        </button>
      </form>
    </div>
  )
}
AddTodo = connect()(AddTodo)

export default AddTodo
```

传入 Store

所有容器组件都可以访问 Redux store，所以可以手动监听它。一种方式是把它以 props 的形式传入到所有容器组件中。但这太麻烦了，因为必须要用 `store` 把展示组件包裹一层，仅仅是因为恰好在组件树中渲染了一个容器组件。

建议的方式是使用指定的 React Redux 组件 `<Provider>` 来 魔法般的 让所有容器组件都可以访问 store，而不必显示地传递它。只需要在渲染根组件时使用即可。

index.js

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

let store = createStore(todoApp)

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

下一步

参照 本完整示例 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, VisibilityFi
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
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 performan
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)(A
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' : 'no
          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 subreddit 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 请求都需要 dispatch 至少三种 action：

- **一种通知 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 来显示加载界面。

- **一种通知 reducer 请求成功的 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 则会隐藏加载界面，并显示接收到的数据。

- **一种通知 reducer 请求失败的 action。**

  对于这种 action，reducer 可能会重置 `isFetching`。另外，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 会降低犯错误的机率，但是如果你使用像 redux-actions 这类的辅助库来生成 action 创建函数和 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 创建函数（Action Creator）

下面先定义几个同步的 action 类型 和 action 创建函数。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 `REQUEST_POSTS` 和 `SELECT_SUBREDDIT` 或 `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。

"Reddit 头条" 应用会长这个样子：

```
{
  selectedsubreddit: '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 and Fl
        }
      ]
    }
  }
}
```

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，同时用户还很少关闭标签页，否则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：

```
{
  selectedsubreddit: '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 JS and
        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 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
```

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], acti
  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 创建函数

最后，如何把之前定义的同步 action 创建函数和 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 创建函数除了返回 action 对象外还可以返回函数。这时，这个 action 创建函数就成为了 thunk。

当 action 创建函数返回函数时，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 创建函数。

`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 创建函数!
// 虽然内部操作不同，你可以像其它 action 创建函数 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.subreddit.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 store = createStore(
  rootReducer,
  applyMiddleware(
    thunkMiddleware, // 允许我们 dispatch() 函数
```

```
    loggerMiddleware // 一个很便捷的 middleware, 用来打印 action 日志
  )
)

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 创建函数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 创建函数，这个 action 创建函数又 dispatch 另一个异步 action 创建函数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：

- 你可以使用 redux-promise 或者 redux-promise-middleware 来 dispatch Promise 替代函数。
- 你可以使用 redux-observable 来 dispatch Observable。
- 你可以使用 redux-saga 中间件来创建更加复杂的异步 action。
- 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：Reddit 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

默认情况下，`createStore()` 所创建的 Redux store 没有使用 middleware，所以只支持 同步数据流。

你可以使用 `applyMiddleware()` 来增强 `createStore()`。虽然这不是必须的，但是它可以帮助你用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析你 dispatch 的任何内容，并继续传递 actions 给下一个 middleware。比如，支持 Promise 的 middleware 能够拦截 Promise，然后为每个 Promise 异步地 dispatch 一对 begin/end actions。

当 middleware 链中的最后一个 middleware 开始 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

现在通过例子你对 middleware 在 Redux 中作用有了初步了解，是时候应用到实际开发中并自定义 middleware 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

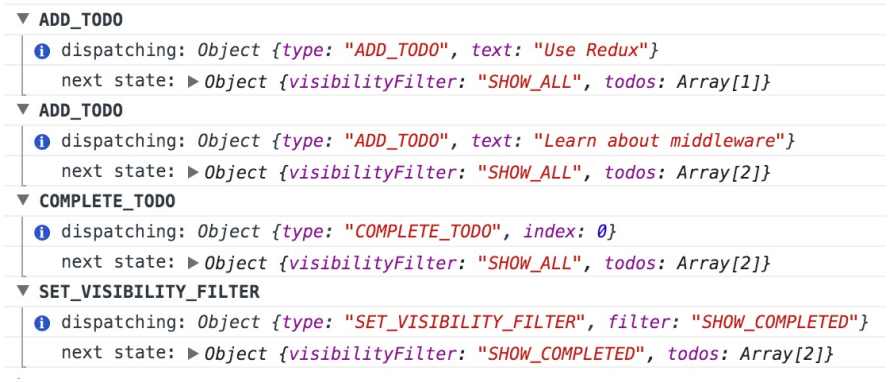
正因为 middleware 可以完成包括异步 API 调用在内的各种事情，了解它的演化过程是一件相当重要的事。我们将以记录日志和创建崩溃报告为例，引导你体会从分析问题到通过构建 middleware 解决问题的思维过程。

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。



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是够令人满意。

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 串连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：这只是一种“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`。

由于在使用之前需要先应用方法到 `createStore()` 之上有些麻烦，`createStore()` 也接受将希望被应用的函数作为最后一个可选参数传入。

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'

let todoApp = combineReducers(reducers)
let store = createStore(
  todoApp,
  // applyMiddleware() 告诉 createStore() 如何处理中间件
  applyMiddleware(logger, crashReporter)
)
```

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` resolve
 *
 * 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 todoApp = combineReducers(reducers)
let store = createStore(
  todoApp,
  applyMiddleware(
    rafScheduler,
    timeoutScheduler,
    thunk,
    vanillaPromise,
    readyStatePromise,
    logger,
    crashReporter
  )
```

```
  )
```

搭配 React Router

现在你想在你的 Redux 应用中使用路由功能，可以搭配使用 React Router 来实现。 Redux 和 React Router 将分别成为你数据和 URL 的事实来源（the source of truth）。 在大多数情况下， **最好** 将他们分开，除非你需要时光旅行和回放 action 来触发 URL 改变。

安装 React Router

可以使用 npm 来安装 `React Router`。本教程基于 `react-router@^2.7.0` 。

```
npm install --save react-router
```

配置后备(fallback) URL

在集成 React Router 之前，我们需要配置一下我们的开发服务器。 显然，我们的开发服务器无法感知配置在 React Router 中的 route。 比如：你想访问并刷新 `/todos`，由于是一个单页面应用，你的开发服务器需要生成并返回 `index.html`。 这里，我们将演示如何在流行的开发服务器上启用这项功能。

> ### 使用 Create React App 须知
>
> 如果你是使用 Create React App （你可以点击这里了解更多，译者注）工具来生成项目，会自动为你配置好后备(fallback) URL。

配置 Express

如果你使用的是 Express 来返回你的 `index.html` 页面，可以增加以下代码到你的项目中：

```
app.get('/*', (req,res) => {
  res.sendfile(path.join(__dirname, 'index.html'))
})
```

配置 WebpackDevServer

如果你正在使用 WebpackDevServer 来返回你的 `index.html` 页面， 你可以增加如下配置到 webpack.config.dev.js：

```
devServer: {
  historyApiFallback: true,
}
```

连接 React Router 和 Redux 应用

在这一章，我们将使用 Todos 作为例子。我们建议你在阅读本章的时候，先将仓库克隆下来。

首先，我们需要从 React Router 中导入 `<Router />` 和 `<Route />`。代码如下：

```
import { Router, Route, browserHistory } from 'react-router';
```

在 React 应用中，通常你会用 `<Router />` 包裹 `<Route />`。 如此，当 URL 变化的时候， `<Router />` 将会匹配到指定的路由，然后渲染路由绑定的组件。 `<Route />` 用来显式地把路由映射到应用的组件结构上。 用 `path` 指定 URL，用 `component` 指定路由命中 URL 后需要渲染的那个组件。

```
const Root = () => (
  <Router>
    <Route path="/" component={App} />
```

```
    </Router>
  );
```

另外，在我们的 Redux 应用中，我们仍将使用 `<Provider />`。 `<Provider />` 是由 React Redux 提供的高阶组件，用来让你将 Redux 绑定到 React（详见 搭配 React）。

然后，我们从 React Redux 导入 `<Provider />`：

```
import { Provider } from 'react-redux';
```

我们将用 `<Provider />` 包裹 `<Router />`，以便于路由处理器可以访问 `store`（暂时未找到相关中文翻译，译者注）。

```
const Root = ({ store }) => (
  <Provider store={store}>
    <Router>
      <Route path="/" component={App} />
    </Router>
  </Provider>
);
```

现在，如果 URL 匹配到 '/'，将会渲染 `<App />` 组件。此外，我们将在 '/' 后面增加参数 `(:filter)`，当我们尝试从 URL 中读取参数 `(:filter)`，需要以下代码：

```
<Route path="/(:filter)" component={App} />
```

也许你想将 '#' 从 URL 中移除（例如：`http://localhost:3000/#/?_k=4sbb0i`）。你需要从 React Router 导入 `browserHistory` 来实现：

```
import { Router, Route, browserHistory } from 'react-router';
```

然后将它传给 `<Router />` 来移除 URL 中的 '#':

```
<Router history={browserHistory}>
  <Route path="/(:filter)" component={App} />
</Router>
```

只要你不需要兼容古老的浏览器，比如IE9，你都可以使用 `browserHistory` 。

`components/Root.js`

```
import React, { PropTypes } from 'react';
import { Provider } from 'react-redux';
import { Router, Route, browserHistory } from 'react-router';
import App from './App';

const Root = ({ store }) => (
  <Provider store={store}>
    <Router history={browserHistory}>
      <Route path="/(:filter)" component={App} />
    </Router>
  </Provider>
);

Root.propTypes = {
  store: PropTypes.object.isRequired,
};

export default Root;
```

通过 React Router 导航

React Router 提供了 `<Link />` 来实现导航功能。 下面将举例演示。现在，修改我们的容器组件 `<FilterLink />` ， 这样我们就可以使用 `<FilterLink />` 来改变 URL。你可以通过 `activeStyle` 属性来指定激活状态的样式。

containers/FilterLink.js

```
import React from 'react';
import { Link } from 'react-router';

const FilterLink = ({ filter, children }) => (
  <Link
    to={filter === 'all' ? '' : filter}
    activeStyle={{
      textDecoration: 'none',
      color: 'black'
    }}
  >
    {children}
  </Link>
);

export default FilterLink;
```

components/Footer.js

```
import React from 'react'
import FilterLink from '../containers/FilterLink'

const Footer = () => (
  <p>
    Show:
    {" "}
    <FilterLink filter="all">
      All
    </FilterLink>
    {", "}
    <FilterLink filter="active">
      Active
    </FilterLink>
    {", "}
    <FilterLink filter="completed">
      Completed
    </FilterLink>
```

```
    </p>
  );

  export default Footer
```

这时，如果你点击 `<FilterLink />`，你将看到你的 URL 在 `'/complete'`， `'/active'`， `'/'` 间切换。 甚至还支持浏览的回退功能，可以从历史记录中找到之前的 URL 并回退。

从 URL 中读取数据

现在，即使 URL 改变，todo 列表也不会被过滤。 这是因为我们是在 `<VisibleTodoList />` 的 `mapStateToProps()` 函数中过滤的。 这个目前仍然是和 `state` 绑定，而不是和 URL 绑定。 `mapStateToProps` 的第二可选参数 `ownProps`，这个是一个传递给 `<VisibleTodoList />` 所有属性的对象。

containers/VisibleTodoList.js

```
const mapStateToProps = (state, ownProps) => {
  return {
    todos: getVisibleTodos(state.todos, ownProps.filter) // 以前是 get
  };
};
```

目前我们还没有传递任何参数给 `<App />`，所以 `ownProps` 依然是一个空对象。 为了能够根据 URL 来过滤我们的 todo 列表，我们需要向 `<VisibleTodoList />` 传递URL参数。

之前我们写过：`<Route path="/(:filter)" component={App} />`，这使得可以在 `App` 中获取 `params` 的属性。

`params` 是一个包含 url 中所有指定参数的对象。 例如：如果我们访问 *`localhost:3000/completed`*，那么 *`params`* 将等价于 *`{ filter: 'completed'`*

`}`。现在，我们可以在 `<App />` 中读取 *URL* 参数了。

注意，我们将使用 ES6 的解构赋值来对 `params` 进行赋值，以此传递给 `<VisibleTodoList />`。

`components/App.js`

```
const App = ({ params }) => {
  return (
    <div>
      <AddTodo />
      <VisibleTodoList
        filter={params.filter || 'all'}
      />
      <Footer />
    </div>
  );
};
```

下一步

现在你已经知道如何实现基础的路由，接下来你可以阅读 React Router API 来学习更多知识。

> **其它路由库注意点**
>
> *Redux Router* 是一个实验性质的库，它使得你的 URL 的状态和 redux store 内部状态保持一致。它有和 React Router 一样的 API，但是它的社区支持比 react-router 小。
>
> *React Router Redux* 将你的 redux 应用和 react-router 绑定在一起，并且使它们保持同步。如果没有这层绑定，你将不能通过时光旅行来回放 action。除非你需要这个，不然 React-router 和 Redux 完全可以分开操作。

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-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(`https://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()

export default function configureStore(preloadedState) {
  return createStore(
    rootReducer,
    preloadedState,
    applyMiddleware(
      thunkMiddleware,
      loggerMiddleware
    )
  )
}
```

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 } f
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 } = thi
    return (
      <div>
        <Picker value={selectedSubreddit}
                onChange={this.handleChange}
                options={[ 'reactjs', 'frontend' ]} />
        <p>
          {lastUpdated &&
            <span>
              Last updated at {new Date(lastUpdated).toLocaleTimeStri
              {' '}
            </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.array.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
- 使用对象展开运算符
- 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回上述的这些 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。

使用对象展开运算符（Object Spread Operator）

从不直接修改 state 是 Redux 的核心理念之一，所以你会发现自己总是在使用 `Object.assign()` 创建对象拷贝，而拷贝中会包含新创建或更新过的属性值。在下面的 `todoApp` 示例中，`Object.assign()` 将会返回一个新的 `state` 对象，而其中的 `visibilityFilter` 属性被更新了：

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

尽管这样可行，但 `Object.assign()` 冗长的写法会迅速降低 reducer 的可读性。

一个可行的替代方案是使用 ES7 提案的 对象展开运算符。该提案让你可以通过展开运算符 (`...`) ，以更加简洁的形式将一个对象的可枚举属性拷贝至另一个对象。对象展开运算符在概念上与 ES6 的 数组展开运算符 相似。 我们试着用这种方式简化 `todoApp` ：

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

```
  }
```

当你在组合复杂对象时, 使用对象展开运算符带来的好处将更加突出。例如下面的 `getAddedIds` 将一个 `id` 数组转换为一个对象数组, 而这些对象的内容是由 `getProduct` 和 `getQuantity` 的结果组合而成。

```
return getAddedIds(state.cart).map(id => Object.assign(
  {},
  getProduct(state.products, id),
  {
    quantity: getQuantity(state.cart, id)
  }
))
```

运用对象扩展运算符简化上面的 `map` 调用:

```
return getAddedIds(state.cart).map(id => ({
  ...getProduct(state.products, id),
  quantity: getQuantity(state.cart, id)
}))
```

目前对象展开运算符提案还处于 ECMAScript Stage 3 草案阶段, 若你想在产品中使用它得依靠转换编译器, 如 Babel。 你可以使用 `es2015` 预设值, 安装 `babel-plugin-transform-object-rest-spread` 并将其单独添加到位于 `.babelrc` 的 `plugins` 数组中。

```
{
  "presets": ["es2015"],
  "plugins": ["transform-object-rest-spread"]
}
```

注意这仍然是一个试验性的语言特性, 在将来可能发生改变。不过一些大型项目如 React Native 已经在广泛使用它。所以我们大可放心, 即使真的发生改

变, 也应该会有自动化的迁移方案。

缩减样板代码

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样板代码。在这章中，我们将考虑 Redux 如何根据个人风格，团队偏好，长期可维护性等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，并且是描述突变数据意图的唯一途径。很重要的一点是 **不得不 dispatch 的 action 对象并非是一个样板代码，而是 Redux 的一个** 基本设计选择.

不少框架声称自己和 Flux 很像，只不过缺少了 action 对象的概念。在可预测性方面，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，action 拥有一个不变的 type 帮助 reducer (或 Flux 中的 Stores) 识别它们。我们建议你使用 string 而不是 符号（Symbols） 作为 action type ，因为 string 是可序列化的，并且使用符号会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';
```

这么做的优势是什么？**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 import 一个 Action 常量的时候拼写错了，你会得到 `undefined` 。在 dispatch 这个 action 的时候，Redux 会立即抛出这个错误，你也会马上发现错误。

你的项目约定取决与你自己。开始时，可能在刚开始用内联字符串（inline string），之后转为常量，也许再之后将他们归为一个独立文件。Redux 在这里没有任何建议，选择你自己最喜欢的。

Action Creators

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

例如，不是使用对象字面量调用 `dispatch` ：

```
// event handler 里的某处
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 import：

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 总被当作样板代码受到批评。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，你可以选用对象字面量** 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。

Action Creators 生成器

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 dispatch 出去之前，注入一个自定义的逻辑来解释你的 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 } from

class Posts extends Component {
  loadData(userId) {
    // 调用 React Redux `connect()` 注入的 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` 响应这个 action
    // 因此让我们显示一个 Spinner 控件。
    dispatch(loadPostsRequest(userId));

    // Reducer 可以通过填写 `users` 响应这些 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 dispatch 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 dispatch 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 重写上面的代码：

`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_F
    // 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 next => action => {
    const {
```

```
    types,
    callAPI,
    shouldCallAPI = () => true,
    payload = {}
  } = action

  if (!types) {
    // Normal action: pass it on
    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 callAPI 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,
      type: successType
    })),
    error => dispatch(Object.assign({}, payload, {
      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_F
    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_C
    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_COMMEN
    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 reducer 用函数描述逻辑更新减少了样板代码里大量的 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 真正的样板代码。真正的 Flux 样板代码是概念性的：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。

Reducers 生成器

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**惟一**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。

另外，如果有使用 ES6 和 JSX 语法，需要使用 Babel (对应示例this example of a Node Server with Babel) 和 React preset。

`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, preloadedState) { /* ... */ }
```

```
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 preloadedState = store.getState();

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

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 `preloadedState` 赋给 `window.__INITIAL_STATE__`。

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

同时使用 script 标签来引入打包后的 js bundle 文件。这是打包工具输出的客户端入口文件，以静态文件或者 URL 的方式实现服务端开发中的热加载。下面是代码。

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

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 preloadedState = window.__INITIAL_STATE__

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

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 preloadedState = { counter }

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

  // 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))
}
```

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（实际项目中 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 preloadedState = { counter }

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

    // 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 Jest 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 jest
```

若想结合 Babel 使用，你需要安装 `babel-jest`：

```
npm install --save-dev babel-jest
```

并且在 `.babelrc` 中启用 ES2015 的功能：

```
{
  "presets": ["es2015"]
}
```

然后，在 `package.json` 的 `scripts` 里加入这一段：

```
{
  ...
  "scripts": {
    ...
    "test": "jest",
    "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 创建函数 (Action Creators)

Redux 里的 action 创建函数是会返回普通对象的函数。在测试 action 创建函数的时候我们想要测试是否调用了正确的 action 创建函数，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 * 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 创建函数

对于使用 Redux Thunk 或其它中间件的异步 action 创建函数，最好完全模拟 Redux store 来测试。 你可以使用 redux-mock-store 把 middleware 应用到模拟的 store。也可以使用 nock 来模拟 HTTP 请求。

示例

```
import fetch from 'isomorphic-fetch';

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 configureMockStore from 'redux-mock-store'
import thunk from 'redux-thunk'
import * as actions from '../../actions/TodoActions'
import * as types from '../../constants/ActionTypes'
import nock from 'nock'
import expect from 'expect' // 你可以使用任何测试库

const middlewares = [ thunk ]
const mockStore = configureMockStore(middlewares)

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, { body: { todos: ['do something'] }})

    const expectedActions = [
      { type: types.FETCH_TODOS_REQUEST },
      { type: types.FETCH_TODOS_SUCCESS, body: { todos: ['do somethin
    ]
    const store = mockStore({ todos: [] })

    return store.dispatch(actions.fetchTodos())
      .then(() => { // 异步 actions 的返回
        expect(store.getActions()).toEqual(expectedActions)
      })
  })
})
```

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),
          completed: false,
          text: action.text
        },
        ...state
      ]

    default:
      return state
  }
}
```

可以这样测试:

```
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: '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 Enzyme 。 Enzyme 底层使用了 React Test Utilities ，但是更方便、更易读，而且更强大。

```
npm install --save-dev enzyme
```

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 React from 'react'
import { shallow } from 'enzyme'
import Header from '../../components/Header'

function setup() {
  const props = {
    addTodo: jest.fn()
  }

  const enzymeWrapper = shallow(<Header {...props} />)

  return {
    props,
    enzymeWrapper
```

```
  }
}

describe('components', () => {
  describe('Header', () => {
    it('should render self and subcomponents', () => {
      const { enzymeWrapper } = setup()

      expect(enzymeWrapper.find('header').hasClass('header')).toBe(tr

      expect(enzymeWrapper.find('h1').text()).toBe('todos')

      const todoInputProps = enzymeWrapper.find('TodoTextInput').prop
      expect(todoInputProps.newTodo).toBe(true)
      expect(todoInputProps.placeholder).toEqual('What needs to be do
    })

    it('should call addTodo if length of text is greater than 0', ()
      const { enzymeWrapper, props } = setup()
      const input = enzymeWrapper.find('TodoTextInput')
      input.props().onSave('')
      expect(props.addTodo.mock.calls.length).toBe(0)
      input.props().onSave('Use Redux')
      expect(props.addTodo.mock.calls.length).toBe(1)
    })
  })
})
```

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(mapDispatchToProps)(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 * 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))(actionA
    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

- Enzyme：Enzyme 是一个 React 的 JavaScript 测试工具，能够让断言、操作以及遍历你的 React 组件的输出变得更简单。
- React Test Utils： React 测试工具。被 Enzyme 所使用。
- 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 `渲染` 方法作为结果，渲染深度仅一层，而非递归渲染整个

DOM 。浅渲染对单元测试很有用， 你只要测试某个特定的组件，而不包括它的子组件。这也意味着，更改一个子组件不会影响到其父组件的测试。如果要测试一个组件和它所有的子组件，可以用 Enzyme's mount() method ， 也就是完全 DOM 渲染来实现。

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/VisibleTodoList.js`

```
import { connect } from 'react-redux'
import { toggleTodo } from '../actions'
import TodoList from '../components/TodoList'

const getVisibleTodos = (todos, filter) => {
  switch (filter) {
    case 'SHOW_ALL':
      return todos
    case 'SHOW_COMPLETED':
      return todos.filter(t => t.completed)
    case 'SHOW_ACTIVE':
      return todos.filter(t => !t.completed)
  }
}

const mapStateToProps = (state) => {
  return {
    todos: getVisibleTodos(state.todos, state.visibilityFilter)
  }
}

const mapDispatchToProps = (dispatch) => {
  return {
    onTodoClick: (id) => {
      dispatch(toggleTodo(id))
    }
```

```
  }
}

const VisibleTodoList = connect(
  mapStateToProps,
  mapDispatchToProps
)(TodoList)

export default VisibleTodoList
```

上面的示例中，`mapStateToProps` 调用了 `getVisibleTodos` 来计算 `todos`。运行没问题，但有一个缺点：每当组件更新时都会重新计算 `todos`。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 `getVisibleTodos`，只在 `state.todos` or `state.visibilityFilter` 变化时重新计算 `todos`，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 `getVisibleTodos` 来替代上面的无记忆版本：

`selectors/index.js`

```
import { createSelector } from 'reselect'

const getVisibilityFilter = (state) => state.visibilityFilter
const getTodos = (state) => state.todos
```

```
export const getVisibleTodos = createSelector(
  [ getVisibilityFilter, getTodos ],
  (visibilityFilter, todos) => {
    switch (visibilityFilter) {
      case 'SHOW_ALL':
        return todos
      case 'SHOW_COMPLETED':
        return todos.filter(t => t.completed)
      case 'SHOW_ACTIVE':
        return todos.filter(t => !t.completed)
    }
  }
)
```

在上例中，`getVisibilityFilter` 和 `getTodos` 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，`getVisibleTodos` 是一个可记忆的 selector。他接收 `getVisibilityFilter` 和 `getTodos` 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 `getVisibleTodos` 被当作另一个 selector 的 input-selector，来进一步通过关键字（keyword）过滤 todos。

```
const getKeyword = (state) => state.keyword

const getVisibleTodosFilteredByKeyword = createSelector(
  [ getVisibleTodos, getKeyword ],
  (visibleTodos, keyword) => visibleTodos.filter(
    todo => todo.text.indexOf(keyword) > -1
  )
)
```

连接 Selector 和 Redux Store

如果你在使用 React Redux，你可以在 `mapStateToProps()` 中当正常函数来调用 selectors

`containers/VisibleTodoList.js`

```
iimport { connect } from 'react-redux'
import { toggleTodo } from '../actions'
import TodoList from '../components/TodoList'
import { getVisibleTodos } from '../selectors'

const mapStateToProps = (state) => {
  return {
    todos: getVisibleTodos(state)
  }
}

const mapDispatchToProps = (dispatch) => {
  return {
    onTodoClick: (id) => {
      dispatch(toggleTodo(id))
    }
  }
}

const VisibleTodoList = connect(
  mapStateToProps,
  mapDispatchToProps
)(TodoList)

export default VisibleTodoList
```

在 selectors 中访问 React Props

> 现在假使我们要支持一个新功能：支持多个 Todo 列表新功能。为了简洁起见，省略了实现这个工程会遇到的与本节不相关的内容（reducers 的变化、组件、Actions 等）

到目前为止，我们只看到 selector 接收 Redux store state 作为参数，然而，selector 也可以接收 props。

这儿有一个 `App` 的组件，它渲染了三个叫做 `VisibleTodoList` 的子组件，每个组件都带一个 `listId` 的 prop;

components/App.js

```
import React from 'react'
import Footer from './Footer'
import AddTodo from '../containers/AddTodo'
import VisibleTodoList from '../containers/VisibleTodoList'

const App = () => (
  <div>
    <VisibleTodoList listId="1" />
    <VisibleTodoList listId="2" />
    <VisibleTodoList listId="3" />
  </div>
)
```

每个 `VisibleTodoList` 容器根据 `listId` props 的值选择不同的 state 切片，让我们修改 `getVisibilityFilter` 和 `getTodos` 来接收 props。

selectors/todoSelectors.js

```
import { createSelector } from 'reselect'

const getVisibilityFilter = (state, props) =>
  state.todoLists[props.listId].visibilityFilter

const getTodos = (state, props) =>
  state.todoLists[props.listId].todos

const getVisibleTodos = createSelector(
  [ getVisibilityFilter, getTodos ],
  (visibilityFilter, todos) => {
```

```
    switch (visibilityFilter) {
      case 'SHOW_COMPLETED':
        return todos.filter(todo => todo.completed)
      case 'SHOW_ACTIVE':
        return todos.filter(todo => !todo.completed)
      default:
        return todos
    }
  }
)

export default getVisibleTodos
```

`props` 可以通过 `mapStateToProps` 传递给 `getVisibleTodos`：

```
const mapStateToProps = (state, props) => {
  return {
    todos: getVisibleTodos(state, props)
  }
}
```

现在，`getVisibleTodos` 可以访问 `props`，一切看上去都是如此的美好。

但是这儿有一个问题!

使用带有多个 `visibleTodoList` 容器实例的 `getVisibleTodos` selector 不能使用函数记忆功能。

containers/VisibleTodoList.js

```
import { connect } from 'react-redux'
import { toggleTodo } from '../actions'
import TodoList from '../components/TodoList'
import { getVisibleTodos } from '../selectors'

const mapStateToProps = (state, props) => {
  return {
```

```
    // 警告：下面的 selector 不会正确记忆
    todos: getVisibleTodos(state, props)
  }
}

const mapDispatchToProps = (dispatch) => {
  return {
    onTodoClick: (id) => {
      dispatch(toggleTodo(id))
    }
  }
}

const VisibleTodoList = connect(
  mapStateToProps,
  mapDispatchToProps
)(TodoList)

export default VisibleTodoList
```

用 `createSelector` 创建的 selector 只有在参数集与之前的参数集相同时才会返回缓存的值。如果我们交替的渲染 `VisibleTodoList listId="1" />` 和 `VisibleTodoList listId="2" />`，共享的 selector 将交替的接收 `listId: 1` 和 `listId: 2`。这会导致每次调用时传入的参数不同，因此 selector 将始终重新计算而不是返回缓存的值。我们将在下一节了解如何解决这个限制。

跨多组件的共享 Selector

> 这节中的例子需要 React Redux v4.3.0 或者更高的版本

为了跨越多个 `VisibleTodoList` 组件共享 selector，**于此同时**正确记忆。每个组件的实例需要有拷贝 selector 的私有版本。

我们创建一个 `makeGetVisibleTodos` 的函数，在每个调用的时候返回一个 `getVisibleTodos` selector 的新拷贝。

selectors/todoSelectors.js

```
import { createSelector } from 'reselect'

const getVisibilityFilter = (state, props) =>
  state.todoLists[props.listId].visibilityFilter

const getTodos = (state, props) =>
  state.todoLists[props.listId].todos

const makeGetVisibleTodos = () => {
  return createSelector(
    [ getVisibilityFilter, getTodos ],
    (visibilityFilter, todos) => {
      switch (visibilityFilter) {
        case 'SHOW_COMPLETED':
          return todos.filter(todo => todo.completed)
        case 'SHOW_ACTIVE':
          return todos.filter(todo => !todo.completed)
        default:
          return todos
      }
    }
  )
}
```

我们还需要一种每个容器访问自己私有 selector 的方式。`connect` 的 `mapStateToProps` 函数可以帮助我们。

如果 `connect` 的 `mapStateToProps` 返回的不是一个对象而是一个函数，他将被用做为每个容器的实例创建一个单独的 `mapStateToProps` 函数。

下面例子中的 `makeMapStateToProps` 创建一个新的 `getVisibleTodos` selectors，返回一个独占新 selector 的权限的 `mapStateToProps` 函数。

```
const makeMapStateToProps = () => {
  const getVisibleTodos = makeGetVisibleTodos()
  const mapStateToProps = (state, props) => {
    return {
      todos: getVisibleTodos(state, props)
```

```
    }
  }
  return mapStateToProps
}
```

如果我们通过 `makeMapStateToProps` 来 `connect`，`VisibleTodosList` 容器的每个组件都会拥有含私有 `getVisibleTodos` selector 的 `mapStateToProps`。不论 `VisibleTodosList` 容器的展现顺序如何，记忆功能都会正常工作。

container/VisibleTodosList.js

```
import { connect } from 'react-redux'
import { toggleTodo } from '../actions'
import TodoList from '../components/TodoList'
import { makeGetVisibleTodos } from '../selectors'

const makeMapStateToProps = () => {
  const getVisibleTodos = makeGetVisibleTodos()
  const mapStateToProps = (state, props) => {
    return {
      todos: getVisibleTodos(state, props)
    }
  }
  return mapStateToProps
}

const mapDispatchToProps = (dispatch) => {
  return {
    onTodoClick: (id) => {
      dispatch(toggleTodo(id))
    }
  }
}

const VisibleTodoList = connect(
  makeMapStateToProps,
  mapDispatchToProps
)(TodoList)
```

```
export default VisibleTodoList
```

下一步

查看 官方文档 和 FAQ。当因为太多的衍生计算和重复渲染导致出现性能问题时，大多数的 Redux 项目会开始使用 Reselect。所以在你创建一个大型项目的时候确保你对 reselect 是熟悉的。你也可以去研究他的 源码，这样你就不认为他是黑魔法了。

实现撤销历史

在应用中构建撤销和重做功能往往需要开发者刻意地付出一些精力。对于经典的 MVC 框架来说，这不是一个简单的问题，因为你需要克隆所有相关的 model 来追踪每一个历史状态。此外，你需要考虑整个撤销堆栈，因为用户的初始更改也是可撤销的。

这意味着在 MVC 应用中实现撤销和重做功能时，你不得不使用一些类似于 Command 的特殊的数据修改模式来重写你的应用代码。

然而你可以用 Redux 轻而易举地实现撤销历史，因为以下三个原因：

- 不存在多个模型的问题，你需要关心的只是 state 的子树。
- state 是不可变数据，所有修改被描述成独立的 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` 可以自然地实现 “reducer enhancers” 或者 “higher order reducers”。它们在你为 reducer 添加额外的功能时保持着这个签名。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。

理解撤销历史

设计状态结构

撤销历史也是应用 state 的一部分，我们没有必要以不同的方式实现它。当你实现撤销和重做这个功能时，无论 state 如何随着时间不断变化，你都需要追踪 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销和重做的话，我们必须保存更多的 state 以解决下面几个问题：

- 撤销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？

为此我们对 state 结构做了以下修改以便解决上述问题：

```
{
```

```
  counter: {
    past: [ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 ],
    present: 10,
    future: []
  }
}
```

现在，如果按下“撤销”，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 ]
  }
}
```

再按一次：

```
{
  counter: {
    past: [ 0, 1, 2, 3, 4, 5, 6, 7 ],
    present: 8,
    future: [ 9, 10 ]
  }
}
```

当我们按下“重做”，我们希望往未来的状态移动一步：

```
{
  counter: {
    past: [ 0, 1, 2, 3, 4, 5, 6, 7, 8 ],
    present: 9,
    future: [ 10 ]
  }
}
```

最终，当处于撤销堆栈中时，用户发起了一个操作（例如，减少计数），那么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的是数字、字符串、数组或是对象都不重要，因为整个结构始终保持一致：

```
{
  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: 'Implem
    future: [
      [ { text: 'Use Redux', complete: true }, { text: 'Implement Und
    ]
  }
}
```

它看起来通常都是这样：

```
{
  past: Array<T>,
  present: T,
  future: Array<T>
}
```

我们可以在顶层保存单一的历史记录：

```
{
  past: [
    { counterA: 1, counterB: 1 },
    { counterA: 1, counterB: 0 },
    { counterA: 0, counterB: 0 }
  ],
  present: { counterA: 2, counterB: 1 },
  future: []
}
```

也可以分离历史记录，这样我们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合适地分离撤销和重做。

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。
- 当处理完外部的 action 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。

初识 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。我们把这种模式加工一下，将其运用到 reducers。

reducer enhancer（或者 **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。这不是什么新模式，`combineReducers()` 也是 reducer enhancer，因为它同样接收多个 reducer 并返回一个新的 reducer。

这是一个没有任何功能的 reducer enhancer 示例：

```
function doNothingWith(reducer) {
  return function (state, action) {
    // 仅仅调用传入的 reducer
    return reducer(state, action)
  };
}
```

一个组合其他 reducer 的 reducer enhancer 看起来类似于这样：

```
function combineReducers(reducers) {
  return function (state = {}, action) {
    return Object.keys(reducers).reduce((nextState, key) => {
      // 调用每一个 reducer 并将其管理的部分 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 封装到 `可撤销` 的 reducer enhancer，从而处理 `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` 追加到当前的 state 上。你也不能忘了通过检查 `.past.length` 和 `.future.length` 确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，所以不必惊讶于这个示例与 elm-undo-redo package 如此相似。

使用 Redux Undo

以上这些信息都非常有用，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，如果之前导出的是 todos reducer，那么现在你需要把这个 reducer 传给 `undoable()` 然后把计算结果导出：

`reducers/todos.js`

```
import undoable, { distinctState } from 'redux-undo'
```

```
/* ... */

const todos = (state = [], action) => {
  /* ... */
}

const undoableTodos = undoable(todos, {
  filter: distinctState()
})

export default undoableTodos
```

这里的 `distinctState()` 过滤器会忽略那些没有引起 state 变化的 actions，可撤销的 reducer 还可以通过其他选择进行配置，例如为撤销和重做的 action 设置 action type。

值得注意的是虽然这与调用 `combineReducers()` 的结果别无二致，但是现在的 `todos` reducer 可以传递给 Redux Undo 增强的 reducer。

`reducers/index.js`

```
import { combineReducers } from 'redux'
import todos from './todos'
import visibilityFilter from './visibilityFilter'

const todoApp = combineReducers({
  todos,
  visibilityFilter
})

export default todoApp
```

你可以在 reducer 合并层次中的任何层级对一个或多个 reducer 执行 `undoable`。我们只对 `todos` reducer 进行封装而不是整个顶层的 reducer，这样 `visibilityFilter` 引起的变化才不会影响撤销历史。

更新 Selectors

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: 'Implem
    future: [
      [ { text: 'Use Redux', complete: true }, { text: 'Implement Und
    ]
  }
}
```

这意味着你必须通过 `state.todos.present` 访问 state 而不是原来的 `state.todos`：

`containers/VisibleTodoList.js`

```
const mapStateToProps = (state) => {
  return {
    todos: getVisibleTodos(state.todos.present, state.visibilityFilte
  }
}
```

添加按钮

现在只剩下给撤销和重做的 action 添加按钮。

首先，为这些按钮创建一个名为 `UndoRedo` 的容器组件。由于展示部分非常简单，我们不再需要把它们分离到单独的文件去：

containers/UndoRedo.js

```
import React from 'react'

/* ... */

let UndoRedo = ({ canUndo, canRedo, onUndo, onRedo }) => (
  <p>
    <button onClick={onUndo} disabled={!canUndo}>
      Undo
    </button>
    <button onClick={onRedo} disabled={!canRedo}>
      Redo
    </button>
  </p>
)
```

你需要使用 React Redux 的 connect 函数生成容器组件，然后检查 `state.todos.past.length` 和 `state.todos.future.length` 来判断是否启用撤销和重做按钮。你不再需要给撤销和重做编写 action creators 了，因为 Redux Undo 已经提供了这些 action creators：

containers/UndoRedo.js

```
/* ... */

import { ActionCreators as UndoActionCreators } from 'redux-undo'
import { connect } from 'react-redux'

/* ... */

const mapStateToProps = (state) => {
  return {
    canUndo: state.todos.past.length > 0,
    canRedo: state.todos.future.length > 0
```

```
  }
}

const mapDispatchToProps = (dispatch) => {
  return {
    onUndo: () => dispatch(UndoActionCreators.undo()),
    onRedo: () => dispatch(UndoActionCreators.redo())
  }
}

UndoRedo = connect(
  mapStateToProps,
  mapDispatchToProps
)(UndoRedo)

export default UndoRedo
```

现在把这个 `UndoRedo` 组件添加到 `App` 组件：

components/App.js

```
import React from 'react'
import Footer from './Footer'
import AddTodo from '../containers/AddTodo'
import VisibleTodoList from '../containers/VisibleTodoList'
import UndoRedo from '../containers/UndoRedo'

const App = () => (
  <div>
    <AddTodo />
    <VisibleTodoList />
    <Footer />
    <UndoRedo />
  </div>
)

export default App
```

就是这样！在示例文件夹下执行 `npm install` 和 `npm start` 试试看吧！

子应用隔离

考虑一下这样的场景：有一个大应用（对应 `<BigApp>` 组件）包含了很多小的“子应用”（对应 `SubApp` 组件）：

```
import React, { Component } from 'react'
import SubApp from './subapp'

class BigApp extends Component {
  render() {
    return (
      <div>
        <SubApp />
        <SubApp />
        <SubApp />
      </div>
    )
  }
}
```

这些 `<SubApp>` 是完全独立的。它们并不会共享数据或 action，也互不可见且不需要通信。

这时最好的做法是不要把它混入到标准 Redux 的 reducer 组件中。 对于一般型的应用，还是建议使用 reducer 组件。但对于 “应用集合”，“仪表板”，或者企业级软件这些把多个本来独立的工具凑到一起打包的场景，可以试下子应用的方案。

子应用的方案还适用于有多个产品或垂直业务的大团队。小团队可以独立发布子应用或者互相独立于自己的“应用壳”中。

下面是 connect 过的子应用的根组件。 像其它组件一样，它还可以渲染更多子组件，connect 或者没有 connect 的都可以。通常只要把它使用 `<Provider>` 渲染就够了。

```
class App extends Component { ... }
export default connect(mapStateToProps)(App)
```

但是，如果不想让外部知道子应用的组件是 Redux 应用的话，可以不调用 `ReactDOM.render(<Provider><App /></Provider>)` 。

或者可以在“大应用”中同时运行它的多个实例呢，还能保证每个在黑盒里运行，外界对 Redux 无感知。

为了使用 React API 来隐藏 Redux 的痕迹，在组件的构造方法里初始化 store 并把它包到一个特殊的组件中：

```
import React, { Component } from 'react'
import { Provider } from 'react-redux'
import reducer from './reducers'
import App from './App'

class SubApp extends Component {
  constructor(props) {
    super(props)
    this.store = createStore(reducer)
  }

  render() {
    return (
      <Provider store={this.store}>
        <App />
      </Provider>
    )
  }
}
```

这样的话每个实例都是独立的。

如果应用间需要共享数据，不 推荐使用这个模式。 但是，如果大应用完全不需要访问子应用内部数据的话非常有用， 同时我们还想把 Redux 作为一种

内部细节实现方式对外部隐藏。 每个组件实例都有它自己的 store，所以它们彼此是不可见的。

组织 Reducer

作为核心概念， Redux 真的是一种十分简单的设计模式：所有你“写”的逻辑都集中在一个单独的函数中，并且执行这些逻辑的唯一方式就是传给 Redux 一个能够描述当时情景的普通对象（plain object）。Redux store 调用这些逻辑函数，并传入当前的 state tree 以及这些描述对象，返回新的 state tree，接着 Redux store 便开始通知这些订阅者（subscriber）state tree 已经改变了。

Redux 设置了一些基本的限制来保证这些逻辑函数的正常工作，就像 Reducers 里面描述的一样，它必须有类似 `(previousState, action) => newState` 这样的结构，它们通常被称作 **reducer 函数**，并且必须是**纯**函数和可预测的。

在这之后，只要遵循这些基本的规则，Redux 就不会关心你在这些 reducer 函数中是如何组织逻辑的。这既能带来会多的自由，也会导致很多的困惑。不过在写这些 reducer 的时候，也会有很多的常见的模式以及很多需要注意的相关信息与概念。而随着应用规模逐渐变大，这些模式在管理这些错综复杂的 reducer 时，处理真实世界的数据时，以及优化 UI 性能时都起着至关重要的作用。

写 Reducer 时必要的概念

这些概念中的一部分，可能已经在别的 Redux 文档中描述过了。其他的概念也都是些比较普通的或者可以适用于 Redux 外的，这里有许多文章来详细的解释这些概念。这些概念和技巧是能写出符合 Solid 原则的 Redux reducer 逻辑的基础。

深入的理解这些概念是你要学习更高级的 Redux 技术之前必不可少的事情。这里有一个推荐的阅读列表。

必要的概念

另外还值得注意的是，在特定的应用特定的架构下，这些建议可能也不是非常的适合。举个例子，如果一个应用使用了 Immutable.js Map 来存储数据，那么它组织 reducer 逻辑的时候就可能和用普通对象存储数据的情况不一样。 这些文档主要假设我们使用的都是 Javascript 普通对象，但即使你使用一些其他的工具，这里的很多规则其实依然适用。

Reducer 概念和技巧

Reducer 基础概念

就像 Reducers 中描述的一样，一个 Redux reducer 函数需要具备：

- 应该有类似 `(previousState, action) => newState` 特征的函数，函数的类型与 Array.prototype.reduce(reducer, ?initialValue) 这个函数很相似。
- 应该是"纯"函数，纯函数意味着不能突变（原文mutate，意指直接修改引用所指向的值）它的参数，如果在函数中执行 API 调用，或者在函数外部修改值，又或者调用一个非纯函数比如 `Date.now()` 或 `Math.random()`，那么就会带来一些副作用。这意味着 state 的更新应该在**"不可变（immutable）"**的理念下完成，这就是说**总是去返回一个新的更新后的对象**，而不是直接去修改原始的 state tree。

> **关于不可变（immutability）和突变（mutation）以及副作用**
>
> 突变是一种不鼓励的做法，因为它通常会打乱调试的过程，以及 React Redux 的 `connect` 函数：
>
> - 对于调试过程, Redux DevTools 期望重放 action 记录时能够输出 state 值，而不会改变任何其他的状态。**突变或者异步行为会产生一些副作用，可能使调试过程中的行为被替换，导致破坏了应用。**
> - 对于 React Redux `connect` 来说，为了确定一个组件（component）是否需要更新，它会检查从 `mapStateToProps` 中返回的值是否发生改变。为了提升性能，`connect` 使用了一些依赖于不可变 state 的方法。并且使用浅引用（shallow reference）来检测状态的改变。这意味着**直接修改对象或者数组是不会被检测到的并且组件不会被重新渲染。**
>
> 其他的副作用像在 reducer 中生成唯一的 ID 或者时间戳时也会导致代码的不可预测并且难以调试和测试。

因为上面这些规则，在去学习具体的组织 Redux reducer 的技术之前，了解并完全理解下面这些核心概念是十分重要的。

Redux Reducer 基础

核心概念:

- 理解 state 和 state shape
- 通过拆分 state 来确定各自的更新职责（**reducer 组合**）
- 高阶 reducers
- 定义 reducer 的初始化状态

阅读列表:

- Redux 文档: Reducer
- Redux 文档: Reducer 样板代码
- Redux 文档: 实现撤销历史
- Redux 文档: `combineReducers`
- 高阶 Reducer 的力量
- Stack Overflow: Store 初始化 state 和 `combineReducers`
- Stack Overflow: State 键的名称与 `combineReducers`

纯函数和副作用

核心概念:

- 副作用
- 纯函数
- 如何理解组合函数

Reading List:

阅读列表:

- 关于函数式编程的一点儿小想法
- 理解程序的副作用
- 学习 Javascript 中的函数式编程
- 使用纯函数编程的理由

不可变数据的管理

核心概念：

- 可变与不可变
- 安全地以不可变的方式更新对象和数组
- 避免在函数和语句中突变 state

阅读列表

- 在 React 中使用 Immutable 特性的优缺点
- Javascript 和 Immutable 特性
- 使用 ES6 的 Immutable 数据及其延伸
- Immutable 数据从零开始
- Redux 文档: 使用对象展开符

范式化数据

核心概念：

- 数据库的组织结构
- 拆分相关/嵌套数据到单独的表中
- 为每个被赋值的对象都存储一个单独的标识
- 通过 ID 引用对象
- 通过对象 ID 来查找表，通过一组 ID 来记录顺序
- 通过关系来联系各个对象

阅读列表：

- 用简单的英语介绍数据库范式化
- Redux 惯用法: 范式化 State Shape
- 范式化文档
- 让 Redux 变得更干净：范式化
- 查询 Redux Store
- 维基百科: 关联实体

- 数据库设计: 多对多
- 当组织你的应用 State 时避免不必要的复杂度

Reducer 和 State 的基本结构

Reducer 的基本结构

首先必须明确的是，整个应用只有一个**单一的 reducer 函数**：这个函数是传给 `createStore` 的第一个参数。一个单一的 reducer 最终需要做以下几件事：

- reducer 第一次被调用的时候，`state` 的值是 `undefined`。reducer 需要在 action 传入之前提供一个默认的 state 来处理这种情况。
- reducer 需要先前的 state 和 dispatch 的 action 来决定需要做什么事。
- 假设需要更改数据，应该用更新后的数据创建新的对象或数组并返回它们。
- 如果没有什么更改，应该返回当前存在的 state 本身。

写 reducer 最简单的方式是把所有的逻辑放在一个单独的函数声明中，就像这样：

```
function counter(state, action) {
    if (typeof state === 'undefined') {
         state = 0; // 如果 state 是 undefined，用这个默认值初始化 store
     }
     if (action.type === 'INCREMENT') {
         return state + 1;
     }
     else if (action.type === 'DECREMENT') {
         return state - 1;
     }
     else {
         return state; // 未识别 action 会经过这里
     }
}
```

这个简单的函数满足上面提到的所有基本要求。在最开始会返回一个默认的值初始化 store；根据 action 的 type 决定 state 是哪种类型的更新，最后返回新的 state；如果没有什么要发生，会返回先前的 state。

这里有一些对这个 reducer 的简单调整。首先，重复的 `if/else` 语句看上去是很烦人的，可以使用 `switch` 语句代替他。其次，我们可以使用 ES6 的默认参数来处理初始 state 不存在的情况。有了这些变化，reducer 看上去会长成这样：

```
function counter(state = 0, action) {
  switch (action.type) {
    case 'INCREMENT':
      return state + 1;
    case 'DECREMENT':
      return state - 1;
    default:
      return state;
  }
}
```

这是典型 Redux reducer 的基本结构。

State 的基本结构

Redux 鼓励你根据要管理的数据来思考你的应用程序。数据就是你应用的 state，state 的结构和组织方式通常会称为 "shape"。在你组织 reducer 的逻辑时，state 的 shape 通常扮演一个重要的角色。

Redux state 中顶层的状态树通常是一个普通的 JavaScript 对象（当然也可以是其他类型的数据，比如：数字、数据或者其他专门的数据结构，但大多数库的顶层值都是一个普通对象）。在顶层对象中组织数据最常见的方式是将数据划分为子树，每个顶层的 key 对应着和特定域或者切片相关联的数据。例如，Todo 应用的 state 通常长这样：

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

在这个例子中，`todos` 和 `visibilityFilter` 都是 state 的顶层 Key，他们分别代表着一个某个特定概念的数据切片。

大多数应用会处理多种数据类型，通常可以分为以下三类：

- 域数据（Domain data）：应用需要展示、使用或者修改的数据（比如 从服务器检索到的所有 todos
- 应用状态（App state）：特定于应用某个行为的数据（比如 “Todo #5 是现在选择的状态”，或者 “正在进行一个获取 Todos 的请求”）
- UI 状态（UI state）：控制 UI 如何展示的数据（比如 “编写 TODO 模型的弹窗现在是展开的”）

Store 代表着应用核心，因此应该用域数据（Domain data）和应用状态数据（App state）定义 State，而不是用 UI 状态（UI state）。举个例子，`state.leftPane.todoList.todos` 这样的结构就是一个坏主意，因为整个应用的核心是 “todos” 而不仅仅是 UI 的一个模块。`todos` 这个切片才应该是 state 结构的顶层。

UI 树和状态树之间很少有 1 对 1 的关系。除非你想明确的跟踪你的 Redux Store 中存储的 UI 数据的各个方面，但即使是这样，UI 数据的结构和域数据的结构也是不一样的。

一个典型的应用 state 大致会长这样：

```
{
    domainData1 : {},
    domainData2 : {},
    appState1 : {},
    appState2 : {},
    ui : {
        uiState1 : {},
        uiState2 : {},
    }
}
```

拆分 Reducer 逻辑

对于任何一个有意义的应用来说，将所有的更新逻辑都放入到单个 reducer 函数中都将会让程序变得不可维护。虽然说对于一个函数应该有多长没有准确的规定，但一般来讲，函数应该比较短，并且只做一件特定的事。因此，把很长的，同时负责很多事的代码拆分成容易理解的小片段是一个很好的编程方式。

因为 Redux reducer 也仅仅是一个函数，上面的概念也适用。你可以将 reducer 中的一些逻辑拆分出去，然后在父函数中调用这个新的函数。

这些新的函数通常分为三类：

1. 一些小的工具函数，包含一些可重用的逻辑片段
2. 用于处理特定情况下的数据更新的函数，参数除了 `(state, action)` 之外，通常还包括其它参数
3. 处理给定 state 切片的所有更新的函数，参数格式通常为 `(state, action)`

为了清楚起见，这些术语将用于区分不同类型的功能和不同的用例：

- **reducer:** 任何符合 `(state, action) -> newState` 格式的函数（即，可以用做 `Array.reducer` 参数的任何函数）。
- **root reducer:** 通常作为 `createStore` 第一个参数的函数。他是唯一的一个在所有的 reducer 函数中必须符合 `(state, action) -> newState` 格式的函数。
- **slice reducer:** 一个负责处理状态树中一块切片数据的函数，通常会作为 `combineReducers` 函数的参数。
- **case function:** 一个负责处理特殊 action 的更新逻辑的函数。可能就是一个 reducer 函数，也可能需要其他参数才能正常工作。
- **higher-order reducer:** 一个以 reducer 函数作为参数，且/或返回一个新的 reducer 函数的函数（比如： `combineReducers`，`redux-undo`）。

在各种讨论中 “sub-reducer” 这个术语通常表示那些不是 root reducer 的任何函数，但这个表述并不是很精确。一些人认为应该表示 "业务逻辑（business login）" （与应用程序特定行为相关的功能）或者 “工具函数（utility functions）”（非应用程序特定的通用功能）。

将复杂的环境分解为更小，更易于理解的过程就是术语中的 函数分解 (functional decomposition)。这个术语可以用在任何代码中。在 Redux 中，使用第三个方法来构造 reducer 逻辑是非常普遍的，即更新逻辑被委托在基于 state 切片的的其他函数中。Redux 将这个概念称为 **reducer composition**，带目前为止，这个方法是构建 reducer 逻辑最常用的方法。事实上， Redux 包含一个 `combineReducers()` 的工具函数，它专门抽象化基于 state 切片的其他 reducer 函数的工作过程。但是你必须明确的是，这并不是唯一模式。实际上，完全可以用所有的三种方法拆分逻辑，通常情况下，这也是一个好主意。Refactoring Reducers 章节会演示一些实例。

使用函数分解（Functional Decomposition）和 Reducer 组合（Reducer Composition）重构 Reducer

看看不同类型的 `sub-reducer` 和如何把他们组合在一起的例子是很有用的。现在让我们看看如何将一个大型的单个 reducer 重构为多个比较小的函数的组合。

> 注意： 为了说明重构的概念和过程而不是为了编写简洁的代码，这个例子是特意以冗长的风格编写的

初遇 Reducer

让我们看看初始 reducer 长什么样：

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


function appReducer(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.concat({
                    id: action.id,
                    text: action.text,
                    completed: false
                })
            });
```

```
        }
        case 'TOGGLE_TODO' : {
            return Object.assign({}, state, {
                todos : state.todos.map(todo => {
                    if (todo.id !== action.id) {
                      return todo;
                    }

                    return Object.assign({}, todo, {
                        completed : !todo.completed
                    })
                  })
            });
        }
        case 'EDIT_TODO' : {
            return Object.assign({}, state, {
                todos : state.todos.map(todo => {
                    if (todo.id !== action.id) {
                      return todo;
                    }

                    return Object.assign({}, todo, {
                        text : action.text
                    })
                  })
            });
        }
        default : return state;
    }
}
```

这个函数非常短，但已经开始变得比较复杂。我们在处理两个不同的区域（filtering 和 todo 列表），嵌套使得更新逻辑难以阅读，并且会让我们不清楚到底是什么跟什么。

提取工具函数（Extracting Utility Functions）

第一步是写一个返回更新了相应区域的新对象。这儿还有一个重复的逻辑是在更新数组中的特定项目，我们也可以将他提成一个函数。

```
function updateObject(oldObject, newValues) {
    // 用空对象作为第一个参数传递给 Object.assign，以确保是复制数据，而不是
    return Object.assign({}, oldObject, newValues);
}

function updateItemInArray(array, itemId, updateItemCallback) {
    const updatedItems = array.map(item => {
        if(item.id !== itemId) {
            // 因为我们只想更新一个项目，所以保留所有的其他项目
            return item;
        }

        // 使用提供的回调来创建新的项目
        const updatedItem = updateItemCallback(item);
        return updatedItem;
    });

    return updatedItems;
}

function appReducer(state = initialState, action) {
    switch(action.type) {
        case 'SET_VISIBILITY_FILTER' : {
            return updateObject(state, {visibilityFilter : action.fil
        }
        case 'ADD_TODO' : {
            const newTodos = state.todos.concat({
                id: action.id,
                text: action.text,
                completed: false
            });

            return updateObject(state, {todos : newTodos});
        }
        case 'TOGGLE_TODO' : {
            const newTodos = updateItemInArray(state.todos, action.id
                return updateObject(todo, {completed : !todo.complete
            });

            return updateObject(state, {todos : newTodos});
        }
        case 'EDIT_TODO' : {
```

```
            const newTodos = updateItemInArray(state.todos, action.id
                return updateObject(todo, {text : action.text});
            });

            return updateObject(state, {todos : newTodos});
        }
        default : return state;
    }
}
```

这样就减少了重复，使得代码的可读性更高。

提取 case reducer

接下来，把特殊逻辑封装成对应的函数：

```
// 省略了内容
function updateObject(oldObject, newValues) {}
function updateItemInArray(array, itemId, updateItemCallback) {}


function setVisibilityFilter(state, action) {
    return updateObject(state, {visibilityFilter : action.filter });
}

function addTodo(state, action) {
    const newTodos = state.todos.concat({
        id: action.id,
        text: action.text,
        completed: false
    });

    return updateObject(state, {todos : newTodos});
}

function toggleTodo(state, action) {
    const newTodos = updateItemInArray(state.todos, action.id, todo =
        return updateObject(todo, {completed : !todo.completed});
    });
```

```
    return updateObject(state, {todos : newTodos});
}

function editTodo(state, action) {
    const newTodos = updateItemInArray(state.todos, action.id, todo =
        return updateObject(todo, {text : action.text});
    });

    return updateObject(state, {todos : newTodos});
}

function appReducer(state = initialState, action) {
    switch(action.type) {
        case 'SET_VISIBILITY_FILTER' : return setVisibilityFilter(sta
        case 'ADD_TODO' : return addTodo(state, action);
        case 'TOGGLE_TODO' : return toggleTodo(state, action);
        case 'EDIT_TODO' : return editTodo(state, action);
        default : return state;
    }
}
```

现在很清楚每个 `case` 发生了什么。我们也可以看到一些模式的雏形。

按域拆分数据（Separating Data Handling by Domain）

目前的 Reducer 仍然需要关心程序中所有不同的 case。下面尝试把 filter 逻辑和 todo 逻辑分离：

```
// 省略了内容
function updateObject(oldObject, newValues) {}
function updateItemInArray(array, itemId, updateItemCallback) {}



function setVisibilityFilter(visibilityState, action) {
    // 从技术上将，我们甚至不关心之前的状态
    return action.filter;
}
```

```
function visibilityReducer(visibilityState = 'SHOW_ALL', action) {
    switch(action.type) {
        case 'SET_VISIBILITY_FILTER' : return setVisibilityFilter(vis
        default : return visibilityState;
    }
};


function addTodo(todosState, action) {
    const newTodos = todosState.concat({
        id: action.id,
        text: action.text,
        completed: false
    });

    return newTodos;
}

function toggleTodo(todosState, action) {
    const newTodos = updateItemInArray(todosState, action.id, todo =>
        return updateObject(todo, {completed : !todo.completed});
    });

    return newTodos;
}

function editTodo(todosState, action) {
    const newTodos = updateItemInArray(todosState, action.id, todo =>
        return updateObject(todo, {text : action.text});
    });

    return newTodos;
}

function todosReducer(todosState = [], action) {
    switch(action.type) {
        case 'ADD_TODO' : return addTodo(todosState, action);
        case 'TOGGLE_TODO' : return toggleTodo(todosState, action);
        case 'EDIT_TODO' : return editTodo(todosState, action);
        default : return todosState;
    }
```

```
}

function appReducer(state = initialState, action) {
    return {
        todos : todosReducer(state.todos, action),
        visibilityFilter : visibilityReducer(state.visibilityFilter,
    };
}
```

我们注意到，两个 reducer 分别关心 state 中的不同的部分。都只需要把自身关心的数据作为参数，不再需要返回复杂的嵌套型 state 对象了，代码变得更简单。

减少样板代码

马上就大功告成了。因为很多人不喜欢使用 switch 这种语法结构，创建一个 action 到 case 查找表示非常通用的做法。可以使用 缩减样板代码 中提到的 `createReducer` 函数减少样板代码。

```
// 省略了内容
function updateObject(oldObject, newValues) {}
function updateItemInArray(array, itemId, updateItemCallback) {}

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


// 省略了内容
function setVisibilityFilter(visibilityState, action) {}

const visibilityReducer = createReducer('SHOW_ALL', {
```

```
    'SET_VISIBILITY_FILTER' : setVisibilityFilter
});

// 省略了内容
function addTodo(todosState, action) {}
function toggleTodo(todosState, action) {}
function editTodo(todosState, action) {}

const todosReducer = createReducer([], {
    'ADD_TODO' : addTodo,
    'TOGGLE_TODO' : toggleTodo,
    'EDIT_TODO' : editTodo
});

function appReducer(state = initialState, action) {
    return {
        todos : todosReducer(state.todos, action),
        visibilityFilter : visibilityReducer(state.visibilityFilter,
    };
}
```

通过切片组合 Reducer（Combining Reducers by Slice）

最后一步了，使用 Redux 中 `combineReducers` 这个工具函数去把管理每个 state 切片的逻辑组合起来，形成顶层的 reducer。最终变成这样：

```
// 可重用的工具函数

function updateObject(oldObject, newValues) {
    // 将空对象作为第一个参数传递给 Object.assign，以确保只是复制数据，而不
    return Object.assign({}, oldObject, newValues);
}

function updateItemInArray(array, itemId, updateItemCallback) {
    const updatedItems = array.map(item => {
        if(item.id !== itemId) {
            // 因为我们只想更新一个项目，所以保留所有的其他项目
            return item;
        }
```

```
          // 使用提供的回调来创建新的项目
          const updatedItem = updateItemCallback(item);
          return updatedItem;
      });

      return updatedItems;
}

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


// 处理特殊 case 的 Handler ("case reducer")
function setVisibilityFilter(visibilityState, action) {
    // 从技术上将，我们甚至不关心之前的状态
    return action.filter;
}

// 处理整个 state 切片的 Handler ("slice reducer")
const visibilityReducer = createReducer('SHOW_ALL', {
    'SET_VISIBILITY_FILTER' : setVisibilityFilter
});

// Case reducer
function addTodo(todosState, action) {
    const newTodos = todosState.concat({
        id: action.id,
        text: action.text,
        completed: false
    });

    return newTodos;
}
```

```
// Case reducer
function toggleTodo(todosState, action) {
    const newTodos = updateItemInArray(todosState, action.id, todo =>
        return updateObject(todo, {completed : !todo.completed});
    });

    return newTodos;
}

// Case reducer
function editTodo(todosState, action) {
    const newTodos = updateItemInArray(todosState, action.id, todo =>
        return updateObject(todo, {text : action.text});
    });

    return newTodos;
}

// Slice reducer
const todosReducer = createReducer([], {
    'ADD_TODO' : addTodo,
    'TOGGLE_TODO' : toggleTodo,
    'EDIT_TODO' : editTodo
});

// 顶层 reducer
const appReducer = combineReducers({
    visibilityFilter : visibilityReducer,
    todos : todosReducer
});
```

现在我们有了分离集中 reducer 的例子：像 `updateObject` 和 `createReducer` 一样的工具函数，像 `setVisibilityFilter` 和 `addTodo` 一样的处理器（Handler），像 `visibilityReducer` 和 `todosReducer` 一样的处理单个切片数据的 Handler。`appReducer` 可以被当作是顶层 reducer。

这个例子中最后的结果看上去比原始的版本更长，这主要是因为工具函数的提取，注释的添加和一些为了清楚起见的故意冗长（比如单独的 return 语句）。单独的看每个功能，他们承担的责任更小，意图也更加清楚。在真正

的应用中，这些函数将会分到单独的文件中，比
如：`reducerUtilities.js`，`visibilityReducer.js`，`todosReudcer.js` 和
`rootReducer.js`。

使用 `combineReducers`

核心概念

基于 Redux 的应用程序中最常见的 state 结构是一个简单的 JavaScript 对象，它最外层的每个 key 中拥有特定域的数据。类似地，给这种 state 结构写 reducer 的方式是分拆成多个 reducer，拆分之后的 reducer 都是相同的结构（state, action），并且每个函数独立负责管理该特定切片 state 的更新。多个拆分之后的 reducer 可以响应一个 action，在需要的情况下独立的更新他们自己的切片 state，最后组合成新的 state。

这个模式是如此的通用，Redux 提供了 `combineReducers` 去实现这个模式。这是一个高阶 Reducer 的示例，他接收一个拆分后 reducer 函数组成的对象，返回一个新的 Reducer 函数。

在使用 `combineReducer` 的时候你需要重点注意下面几个方法：

- 首先，`combineReducer` 只是一个工具函数，用于简化编写 Redux reducer 时最常见的场景。你没有必要一定在你的应用程序中使用它，他不会处理每一种可能的场景。你完全可以不使用它来编写 reducer，或者对于 `combinerReducer` 不能处理的情况编写自定义的 reducer。（参见 combineReducers 章节中的例子和建议）
- 虽然 Redux 本身并不管你的 state 是如何组织的，但是 combineReducer 强制地约定了几个规则来帮助使用者们避免常见的错误（参见 combineReducer）
- 一个常见的问题是 Reducer 在 dispatch action 的时候是否调用了所有的 reducer。当初你可能觉得“不是”，因为真的只有一个根 reducer 函数啊。但是 `combineReducer` 确实有着这样的特殊效果。在生成新的 state 树时，`combinerReducers` 将调用每一个拆分之后的 reducer 和与当前的 Action，如果有需要的话会使得每一个 reducer 有机会响应和更新拆分后的 state。所以，在这个意义上，`combineReducers` 会调用所有的 reducer，严格来说是它包装的所有 reducer。

- 你可以在任何级别的 reducer 中使用 `combineReducer`，不仅仅是在创建根 reducer 的时候。在不同的地方有多个组合的 reducer 是非常常见的，他们组合到一起来创建根 reducer。

定义 State 结构

这里有两种方式来定义 Store state 的初始结构和内容。首先，`createStore` 函数可以将 `preloadedState` 作为第二个参数。这主要用于初始化那些在其他地方有持久化存储的 state，例如浏览器的 localStorage，另外一种方式是当 state 是 `undefined` 的时候返回 initial state。这两种方法在 初始化 state 章节 中有着更加详细的描述，但是在使用 `combineReducers` 的时候需要注意其他的一些问题。

`combineReducers` 接收拆分之后的 reducer 函数组成的对象，并且创建出具有相同键对应状态对象的函数。这意味着如果没有给 `createStore` 提供预加载 state，输出 state 对象的 key 将由输入的拆分之后 reducer 组成对象的 key 决定。这些名称之间的相关性并不总是显而易见的，尤其是在使用 ES6 的时候（如默认模块搭配出和对象字面量的简写方向时）。

这儿有一些如何用 ES6 中对象字面量简写方式使用 `combineReducers` 的例子。

```
// reducers.js
export default theDefaultReducer = (state = 0, action) => state;

export const firstNamedReducer = (state = 1, action) => state;

export const secondNamedReducer = (state = 2, action) => state;


// rootReducer.js
import {combineReducers, createStore} from "redux";

import theDefaultReducer, {firstNamedReducer, secondNamedReducer} fro
```

```
// 使用 ES6 的对象字面量简写方式定义对象结构
const rootReducer = combineReducers({
    theDefaultReducer,
    firstNamedReducer,
    secondNamedReducer
});

const store = createStore(rootReducer);
console.log(store.getState());
// {theDefaultReducer : 0, firstNamedReducer : 1, secondNamedReducer
```

因为我们使用了 ES6 中的对象字面量简写方式，在最后的 state 中 key 的名字和 import 进来的变量的名字一样，这可能并不是经常期望的，经常会对不熟悉 ES6 的人造成困惑。

同样的，结果的名字也有点奇怪，在 state 中 key 的名字包含 “reducer” 这样的词通常不是一个好习惯，key 应该反映他们特有域或者数据类型。这意味着我们应该明确拆分之后 reducer 对象中 key 的名称，定义输出 state 对象中的 key，或者在使用对象字面量简写方式的时候，仔细的重命名的拆分之后的 reducer 以设置 key。

一个比较好用的使用示例如下：

```
import {combineReducers, createStore} from "redux";

// 将 default import 进来的名称重命名为任何我们想要的名称。我们也可以重命名
import defaultState, {firstNamedReducer, secondNamedReducer as second

const rootReducer = combineReducers({
    defaultState,                   // key 的名称和 default export 的名
    firstState : firstNamedReducer, // key 的名字是单独取的，而不是变量的
    secondState,                    // key 的名称和已经被重命名过的 exp
});

const reducerInitializedStore = createStore(rootReducer);
console.log(reducerInitializedStore.getState());
// {defaultState : 0, firstState : 1, secondState : 2}
```

这种 state 的结构恰好能反应所涉及的数据，因为我们特别的设置了我们传递给 `combineReducers` 的 key。

`combineReducers` 进阶

Redux 引入了非常实用的 `combineReducers` 工具函数，但我们却粗暴地将它限制于单一的应用场景：把不同片段的 state 的更新工作委托给一个特定的 reducer，以此更新由普通的 JavaScript 对象构成的 state 树。它不解决 Immutable.js Maps 所构建的 state tree，也不会把其余部分的 state 作为额外参数传递给 reducer 或者排列 reducer 的调用顺序，它同样不关心 reducer 如何工作。

于是一个常见问题出现了，“`combineReducers` 如何处理这些应用场景呢？”通常给出的回答只是“你不能这么做，你可能需要通过其他方式解决”。**一旦你突破 `combineReducers` 的这种限制，就是创建各色各样的“自定义” reducer 的时候了**，不管是为了解决一次性场景的特殊 reducer，还是能够被广泛复用的 reducer。本文为几种典型的应用场景提供了建议，但你也可以自由发挥。

结合 Immutable.js 对象使用 reducers

由于目前 `combineReducers` 只能处理普通的 JavaScript 对象，对于把 Immutable.js Map 对象作为顶层 state 树的应用程序来说，可能无法使用 `combineReducers` 管理应用状态。因为很多开发者采用了 Immutable.js，所以涌现了大量提供类似功能的工具，例如 redux-immutable。这个第三方包实现了一个能够处理 Immutable Map 数据而非普通的 JavaScript 对象的 `combineReducers`。

不同 reducers 之间共享数据

类似地，如果 `sliceReducerA` 为了处理特殊的 action 正好需要来自 `sliceReducerB` 的部分 state 数据，或者 `sliceReducerB` 正好需要全部的 state 作为参数，单单就 `combineReducers` 是无法解决这种问题的。可以这样来解决：把所需数据当额外参数的形式传递给自定义函数，例如：

```
function combinedReducer(state, action) {
    switch(action.type) {
        case "A_TYPICAL_ACTION" : {
            return {
                a : sliceReducerA(state.a, action),
                b : sliceReducerB(state.b, action)
            };
        }
        case "SOME_SPECIAL_ACTION" : {
            return {
                // 明确地把 state.b 作为额外参数进行传递
                a : sliceReducerA(state.a, action, state.b),
                b : sliceReducerB(state.b, action)
            }
        }
        case "ANOTHER_SPECIAL_ACTION" : {
            return {
                a : sliceReducerA(state.a, action),
                // 明确地把全部的 state 作为额外参数进行传递
                b : sliceReducerB(state.b, action, state)
            }
        }
        default: return state;
    }
}
```

另一种解决“共享片段数据更新” (shared-slice updates) 问题的简单方法是，给 action 添加额外数据。可以通过 thunk 函数或者类似的方法轻松实现，如下：

```
function someSpecialActionCreator() {
    return (dispatch, getState) => {
        const state = getState();
        const dataFromB = selectImportantDataFromB(state);

        dispatch({
            type : "SOME_SPECIAL_ACTION",
            payload : {
                dataFromB
```

```
            }
        });
    }
}
```

因为 B 的数据已经存在于 action 中，所以它的父级 reducer 不需要做任何特殊的处理就能把数据暴露给 `sliceReducerA`。

第三种方法是：使用 `combineReducers` 组合 reducer 来处理“简单”的场景，每个 reducer 依旧只更新自己的数据；同时新加一个 reducer 来处理多块数据交叉的“复杂”场景；最后写一个包裹函数依次调用这两类 reducer 并得到最终结果：

```
const combinedReducer = combineReducers({
    a : sliceReducerA,
    b : sliceReducerB
});

function crossSliceReducer(state, action) {
    switch(action.type) {
        case "SOME_SPECIAL_ACTION" : {
            return {
                // 明确地把 state.b 作为额外参数进行传递
                a : handleSpecialCaseForA(state.a, action, state.b),
                b : sliceReducerB(state.b, action)
            }
        }
        default : return state;
    }
}

function rootReducer(state, action) {
    const intermediateState = combinedReducer(state, action);
    const finalState = crossSliceReducer(intermediateState, action);
    return finalState;
}
```

已经有一个库 reduce-reducers 可以简化以上操作流程。它接收多个 reducer 然后对它们依次执行 `reduce()` 操作，并把产生的中间值依次传递给下一个 reducer：

```
// 与上述手动编写的 `rootReducer` 一样
const rootReducer = reduceReducers(combinedReducers, crossSliceReduce
```

值得注意的是，如果你使用 `reduceReducers` 你应该确保第一个 reducer 能够定义初始状态的 state 数据，因为后续的 reducers 通常会假定 state 树已经存在，也就不会为此提供默认状态。

更多建议

再次强调，Reducer 只是普通的函数，明确这一概念非常重要。`combineReducers` 虽然实用也只是冰山一角。除了用 switch 语句编写函数，还可以用条件逻辑；函数不仅可以彼此组合，也可以调用其他函数。也许你可能需要这样的一个 reducer，它既能够重置 state，也能够响应特定的 action。你可以这样做：

```
const undoableFilteredSliceA = compose(undoReducer, filterReducer("AC
const rootReducer = combineReducers({
    a : undoableFilteredSliceA,
    b : normalSliceReducerB
});
```

注意 `combineReducers` 无需知道也不关心任何一个负责管理 `a` 数据的 reducer。所以我们并不需要像以往一样修改 `combineReducers` 来实现撤销功能 —— 我们只需把各种函数组合成一个新函数即可。

另外 `combineReducers` 只是 Redux 内置的 reducer 工具，大量形式各异的可复用的第三方 reducer 工具层出不穷。在 Redux Addons 目录中列举了很

多可供使用的第三方工具。也许这些工具解决不了你的应用场景，但你随时都可以实现一个能够满足你需求的工具函数。

State 范式化

事实上，大部分程序处理的数据都是嵌套或互相关联的。例如，一个博客中有多篇文章，每篇文章有多条评论，所有的文章和评论又都是由用户产生的。这种类型应用的数据看上去可能是这样的：

```
const blogPosts = [
    {
        id : "post1",
        author : {username : "user1", name : "User 1"},
        body : "......",
        comments : [
            {
                id : "comment1",
                author : {username : "user2", name : "User 2"},
                comment : ".....",
            },
            {
                id : "comment2",
                author : {username : "user3", name : "User 3"},
                comment : ".....",
            }
        ]
    },
    {
        id : "post2",
        author : {username : "user2", name : "User 2"},
        body : "......",
        comments : [
            {
                id : "comment3",
                author : {username : "user3", name : "User 3"},
                comment : ".....",
            },
            {
                id : "comment4",
                author : {username : "user1", name : "User 1"},
                comment : ".....",
```

```
            },
            {
                id : "comment5",
                author : {username : "user3", name : "User 3"},
                comment : ".....",
            }
        ]
    }
    // and repeat many times
]
```

上面的数据结构比较复杂，并且有部分数据是重复的。这里还存在一些让人关心的问题：

- 当数据在多处冗余后，需要更新时，很难保证所有的数据都进行更新。
- 嵌套的数据意味着 reducer 逻辑嵌套更多、复杂度更高。尤其是在打算更新深层嵌套数据时。
- 不可变的数据在更新时需要状态树的祖先数据进行复制和更新，并且新的对象引用会导致与之 connect 的所有 UI 组件都重复 render。尽管要显示的数据没有发生任何改变，对深层嵌套的数据对象进行更新也会强制完全无关的 UI 组件重复 render

正因为如此，在 Redux Store 中管理关系数据或嵌套数据的推荐做法是将这一部分视为数据库，并且将数据按范式化存储。

设计范式化的 State

范式化的数据包含下面几个概念：

- 任何类型的数据在 state 中都有自己的 “表”。
- 任何 “数据表” 应将各个项目存储在对象中，其中每个项目的 ID 作为 key，项目本身作为 value。
- 任何对单个项目的引用都应该根据存储项目的 ID 来完成。
- ID 数组应该用于排序。

上面博客示例中的 state 结构范式化之后可能如下：

```
{
    posts : {
        byId : {
            "post1" : {
                id : "post1",
                author : "user1",
                body : "......",
                comments : ["comment1", "comment2"]
            },
            "post2" : {
                id : "post2",
                author : "user2",
                body : "......",
                comments : ["comment3", "comment4", "comment5"]
            }
        }
        allIds : ["post1", "post2"]
    },
    comments : {
        byId : {
            "comment1" : {
                id : "comment1",
                author : "user2",
                comment : ".....",
            },
            "comment2" : {
                id : "comment2",
                author : "user3",
                comment : ".....",
            },
            "comment3" : {
                id : "comment3",
                author : "user3",
                comment : ".....",
            },
            "comment4" : {
                id : "comment4",
                author : "user1",
                comment : ".....",
```

```
            },
            "comment5" : {
                id : "comment5",
                author : "user3",
                comment : ".....",
            },
        },
        allIds : ["comment1", "comment2", "comment3", "commment4", "c
    },
    users : {
        byId : {
            "user1" : {
                username : "user1",
                name : "User 1",
            }
            "user2" : {
                username : "user2",
                name : "User 2",
            }
            "user3" : {
                username : "user3",
                name : "User 3",
            }
        },
        allIds : ["user1", "user2", "user3"]
    }
}
```

这种 state 在结构上更加扁平。与原始的嵌套形式相比，有下面几个地方的改进：

- 每个数据项只在一个地方定义，如果数据项需要更新的话不用在多处改变
- reducer 逻辑不用处理深层次的嵌套，因此看上去可能会更加简单
- 检索或者更新给定数据项的逻辑变得简单与一致。给定一个数据项的 type 和 ID，不必挖掘其他对象而是通过几个简单的步骤就能查找到它。
- 每个数据类型都是唯一的，像改评论这样的更新仅仅需要状态树中 “comment > byId > comment” 这部分的复制。这也就意味着在 UI 中只

> 有数据发生变化的一部分才会发生更新。与之前的不同的是，之前嵌套形式的结构需要更新整个 comment 对象，post 对象的父级，以及整个 post 对象的数组。这样就会让所有的 Post 组件和 Comment 组件都再次渲染。

需要注意的是，范式化的 state 意味更多的组件被 connect，每个组件负责查找自己的数据，这和小部分的组件被 connect，然后查找大部分的数据再进行向下传递数据是恰恰相反的。事实证明，connect 父组件只需要将数据项的 Id 传递给 connect 的子对象是在 Redux 应用中优化 UI 性能的良好模式，因此保持范式化的 state 在提高性能方面起着关键作用。

组织 State 中的范式化数据

一个典型的应用中通常会有相关联的数据和无关联数据的混合体。虽然我们对这种不同类型的数据应该如何组织没有一个单一的规则，但常见的模式是将关系 “表” 放在一个共同的父 key 中，比如：“entities”。通过这种模式组织的 state 看上去长得像这样：

```
{
    simpleDomainData1: {....},
    simpleDomainData2: {....}
    entities : {
        entityType1 : {....},
        entityType2 : {....}
    }
    ui : {
        uiSection1 : {....},
        uiSection2 : {....}
    }
}
```

这样可以通过多种方式进行扩展。比如一个对 entities 要进行大量编辑的应用可能希望在 state 中保持两组 “表”，一个用于存储 “当前”（current） 的项目，一个用于存储 “正在进行中”(work-in-progress) 的项目。当数据项在被编辑的时候，其值可以被复制到 “正在进行中” 的那个表中，任何更新他的动作

都将在 “正在进行中” 的表中工作，编辑表单被该组数据控制着，UI 仍然被原始数据控制着。表单的 “重置” 通过移除 “正在进行中” 表的数据项然后从 “当前” 表中复制原始数据到 “正在进行中” 表中就能轻易做到，表单的 “应用” 通过把 “正在进行中” 表的数据复制到 “当前” 表中就能实现。

表间关系

因为我们将 Redux Store 视为数据库，所以在很多数据库的设计规则在这里也是适用的。例如，对于多对多的关系，可以设计一张中间表存储相关联项目的 ID（经常被称作 “连接表” 或者 “关联表”）。为了一致起见，我们还会使用相同的 `byId` 和 `allIds` 用于实际的数据项表中。

```
{
    entities: {
        authors : { byId : {}, allIds : [] },
        books : { byId : {}, allIds : [] },
        authorBook : {
            byId : {
                1 : {
                    id : 1,
                    authorId : 5,
                    bookId : 22
                },
                2 : {
                    id : 2,
                    authorId : 5,
                    bookId : 15,
                }
                3 : {
                    id : 3,
                    authorId : 42,
                    bookId : 12
                }
            },
            allIds : [1, 2, 3]

        }
    }
```

```
  }
```

像 “查找这个作者所有的书” 这个操作可以通过在连接表上一个单一的循环来实现。相对于应用中一般情况下数据量和 JavaScript 引擎的运行速度，在大多数情况下，这样操作的性能是足够好的。

嵌套数据范式化

因为 API 经常以嵌套的形式发送返回数据，所以该数据需要在引入状态树之前转化为规范化形态。Normalizr 库可以帮助你实现这个。你可以定义 schema 的类型和关系，将 schema 和响应数据提供给 Normalizr，他会输出响应数据的范式化变换。输出可以放在 action 中，用于 store 的更新。有关其用法的更多详细信息，请参阅 Normalizr 文档。

管理范式化数据

如 范式化数据 章节所提及的，我们经常使用 Normaizr 库将嵌套式数据转化为适合集成到 store 中的范式化数据。但这并不解决针对范式化的数据进一步更新后在应用的其他地方使用的问题。根据喜好有很多种方法可供使用。下面展示一个像文章添加评论的示例。

标准方法

简单合并

一种方法是将 action 的内容合并到现有的 state。在这种情况下，我们需要一个对数据的深拷贝（非浅拷贝）。Lodash 的 `merge` 方法可以帮我们处理这个：

```
import merge from "lodash/object/merge";

function commentsById(state = {}, action) {
    switch(action.type) {
        default : {
           if(action.entities && action.entities.comments) {
               return merge({}, state, action.entities.comments.byId)
           }
           return state;
        }
    }
}
```

这样做会让 reducer 保持最小的工作量，但需要 action creator 在 action dispatch 之前做大量的工作来将数据转化成正确的形态。在删除数据项时这种方式也是不适合的。

reducer 切片组合

如果我们有一个由切片 reducer 组成的嵌套数据，每个切片 reducer 都需要知道如何响应这个 action。因为我们需要让 action 囊括所有相关的数据。譬如更新相应的 Post 对象需要生成一个 comment 的 id，然后使用 id 作为 key 创建一个新的 comment 对象，并且让这个 comment 的 id 包括在所有的 comment id 列表中。下面是一个如何组合这样数据的例子：

> 译者注：结合上章节中范式化之后的 state 阅读

```
// actions.js
function addComment(postId, commentText) {
    // 为这个 comment 生成一个独一无二的 ID
    const commentId = generateId("comment");

    return {
        type : "ADD_COMMENT",
        payload : {
            postId,
            commentId,
            commentText
        }
    };
}


// reducers/posts.js
function addComment(state, action) {
    const {payload} = action;
    const {postId, commentId} = payload;

    // 查找出相应的文章，简化其余代码
    const post = state[postId];

    return {
        ...state,
        // 用新的 comments 数据更新 Post 对象
        [postId] : {
             ...post,
```

```
                comments : post.comments.concat(commentId)
            }
        };
    }

    function postsById(state = {}, action) {
        switch(action.type) {
            case "ADD_COMMENT" : return addComment(state, action);
            default : return state;
        }
    }

    function allPosts(state = [], action) {
        // 省略，这个例子中不需要它
    }

    const postsReducer = combineReducers({
        byId : postsById,
        allIds : allPosts
    });


    // reducers/comments.js
    function addCommentEntry(state, action) {
        const {payload} = action;
        const {commentId, commentText} = payload;

        // 创建一个新的 Comment 对象
        const comment = {id : commentId, text : commentText};

        // 在查询表中插入新的 Comment 对象
        return {
            ...state,
            [commentId] : comment
        };
    }

    function commentsById(state = {}, action) {
        switch(action.type) {
            case "ADD_COMMENT" : return addCommentEntry(state, action);
            default : return state;
        }
```

```
}


function addCommentId(state, action) {
    const {payload} = action;
    const {commentId} = payload;
    // 把新 Comment 的 ID 添加在 all IDs 的列表后面
    return state.concat(commentId);
}

function allComments(state = [], action) {
    switch(action.type) {
        case "ADD_COMMENT" : return addCommentId(state, action);
        default : return state;
    }
}

const commentsReducer = combineReducers({
    byId : commentsById,
    allIds : allComments
});
```

这个例子之所有有点长，是因为它展示了不同切片 reducer 和 case reducer 是如何配合在一起使用的。注意这里对 “委托” 的理解。postById reducer 切片将工作委拖给 addComment，addComment 将新的评论 id 插入到相应的数据项中。同时 commentsById 和 allComments 的 reducer 切片都有自己的 case reducer，他们更新评论查找表和所有评论 id 列表的表。

其他方法

基于任务的更新

reducer 仅仅是个函数，因此有无数种方法来拆分这个逻辑。使用切片 reducer 是最常见，但也可以在更面向任务的结构中组织行为。由于通常会涉及到更多嵌套的更新，因此常常会使用 dot-prop-immutable、object-path-immutable 等库实现不可变更新。

```
import posts from "./postsReducer";
import comments from "./commentsReducer";
import dotProp from "dot-prop-immutable";
import {combineReducers} from "redux";
import reduceReducers from "reduce-reducers";

const combinedReducer = combineReducers({
    posts,
    comments
});


function addComment(state, action) {
    const {payload} = action;
    const {postId, commentId, commentText} = payload;

    // State here is the entire combined state
    const updatedWithPostState = dotProp.set(
        state,
        `posts.byId.${postId}.comments`,
        comments => comments.concat(commentId)
    );

    const updatedWithCommentsTable = dotProp.set(
        updatedWithPostState,
        `comments.byId.${commentId}`,
        {id : commentId, text : commentText}
    );

    const updatedWithCommentsList = dotProp.set(
        updatedWithCommentsTable,
        `comments.allIds`,
        allIds => allIds.concat(commentId);
    );

    return updatedWithCommentsList;
}

const featureReducers = createReducer({}, {
    ADD_COMMENT : addComment,
};
```

```
const rootReducer = reduceReducers(
    combinedReducer,
    featureReducers
);
```

这种方法让 `ADD_COMMENT` 这个 case 要干哪些事更加清楚，但需要更新嵌套逻辑和对特定状态树的了解。最后这取决于你如何组织 reducer 逻辑，或许你根本不需要这样做。

Redux-ORM

Redux-ORM 库提供了一个非常有用的抽象层，用于管理 Redux store 中存储的范式化数据。它允许你声明 Model 类并且定义他们之间的关系。然后它可以为你的数据类型生成新“表”，充当用于查找数据的特殊选择器工具，并且对数据执行不可变更新。

有几种方法可以用 Redux-ORM 执行数据更新。首选，Redux-ORM 文档建议在每个 Model 子类上定义 reducer 函数，然后将自动生成的组合 reducer 函数放到 store 中：

```
// models.js
import {Model, many, Schema} from "redux-orm";

export class Post extends Model {
  static get fields() {
    return {
      // 定义一个多边关系 - 一个 Post 可以有多个 Comments,
      // 字段名是 “comments”
      comments : many("Comment")
    };
  }

  static reducer(state, action, Post) {
    switch(action.type) {
      case "CREATE_POST" : {
        // 排队创建一个 Post 实例
        Post.create(action.payload);
```

```
        break;
      }
      case "ADD_COMMENT" : {
        const {payload} = action;
        const {postId, commentId} = payload;
        // 排队增加一个 Comment ID 和 Post 实例的联系
        Post.withId(postId).comments.add(commentId);
        break;
      }
    }

    // Redux-ORM 将在返回后自动应用排队的更新
  }
}
Post.modelName = "Post";

export class Comment extends Model {
  static get fields() {
    return {};
  }

  static reducer(state, action, Comment) {
    switch(action.type) {
      case "ADD_COMMENT" : {
        const {payload} = action;
        const {commentId, commentText} = payload;

        // 排队创建一个 Comment 实例
        Comment.create({id : commentId, text : commentText});
        break;
      }
    }

    // Redux-ORM 将在返回后自动应用排队的更新
  }
}
Comment.modelName = "Comment";

// 创建 Schema 实例，然后和 Post、Comment 数据模型挂钩起来
export const schema = new Schema();
schema.register(Post, Comment);
```

```
// main.js
import { createStore, combineReducers } from 'redux'
import {schema} from "./models";

const rootReducer = combineReducers({
  // 插入 Redux-ORM 自动生成的 reducer，这将
  // 初始化一个数据模型“表”，并且和我们在
  // 每个 Model 子类中定义的 reducer 逻辑挂钩起来
  entities : schema.reducer()
});

// dispatch 一个 action 以创建一个 Post 实例
store.dispatch({
  type : "CREATE_POST",
  payload : {
    id : 1,
    name : "Test Post Please Ignore"
  }
});

// dispath 一个 action 以创建一个 Comment 实例作为上个 Post 的子元素
store.dispatch({
  type : "ADD_COMMENT",
  payload : {
    postId : 1,
    commentId : 123,
    commentText : "This is a comment"
  }
});
```

Redux-ORM 库维护要应用的内部更新队列。这些更新是不可变更新，这个库简化了这个更新过程。

使用 Redux-ORM 的另一个变化是用一个单一的 case reducer 作为抽象层。

```
import {schema} from "./models";

// 假设这个 case reducer 正在我们的“entities”切片 reducer 使用，
// 并且我们在 Redux-ORM 的 Model 子类上没有定义 reducer
```

```
function addComment(entitiesState, action) {
    const session = schema.from(entitiesState);
    const {Post, Comment} = session;
    const {payload} = action;
    const {postId, commentId, commentText} = payload;

    const post = Post.withId(postId);
    post.comments.add(commentId);

    Comment.create({id : commentId, text : commentText});

    return session.reduce();
}
```

总之，Redux-ORM 提供了一组非常有用的抽象，用于定义数据类型之间的关系，在我们的 state 中创建了一个 “表”，检索和反规划关系数据，以及将不可变更新应用于关系数据。

Reducer 逻辑复用

随着应用程序的增长，在 reducer 逻辑中开始出现一些常见的模式。你可能会发现一部分 reducer 逻辑对于不同类型的数据做着相同的工作，你想通过对每种数据类型复用相同的公共逻辑来减少重复的代码。或者，你可能想要在 store 中处理某个类型的数据的多个”实例“。然而，Redux store 采用全局结构的设计本身就是一种折衷：优点是易于追踪应用程序的整体状态，但是，也可能更难的”命中“那些需要更新特定一部分状态的 action，特别是当你使用了 `combineReducers` 。

例如，假设想在程序中追踪多个计数器，分别命名为A，B，和C。定义初始的 `counter` reducer，然后使用 `combineReducers` 去设置状态。

```
function counter(state = 0, action) {
    switch (action.type) {
        case 'INCREMENT':
            return state + 1;
        case 'DECREMENT':
            return state - 1;
        default:
            return state;
    }
}

const rootReducer = combineReducers({
    counterA : counter,
    counterB : counter,
    counterC : counter
});
```

不幸的是，这样设置有一个问题。因为 `combineReducers` 将会使用相同的action调用每一个 reducer，发送 `{type:'INCREMENT'}` 实际上将会导致所有三个计数器的值被增加，而不仅仅是其中一个。我们需要一些方法去封装 `counter` 的逻辑，以此来保证只有我们关心的计数器被更新。

使用高阶 Reducer 来定制行为

正如在 Reducer 逻辑拆分 定义的那样，高阶 reducer 是一个接收 reducer 函数作为参数，并返回新的 reducer 函数的函数。它也可以被看作成一个“reducer 工厂”。`combineReducers` 就是一个高阶 reduce 的例子。我们可以使用这种模式来创建特定版本的 reducer 函数，每个版本只响应特定的 action。

创建特定的 reducer 有两种最常见的方式，一个是使用给定的前缀或者后缀生成新的 action 常量，另一个是在 action 对象上附加额外的信息。下面是它们大概的样子：

```
function createCounterWithNamedType(counterName = '') {
    return function counter(state = 0, action) {
        switch (action.type) {
            case `INCREMENT_${counterName}`:
                return state + 1;
            case `DECREMENT_${counterName}`:
                return state - 1;
            default:
                return state;
        }
    }
}

function createCounterWithNameData(counterName = '') {
    return function counter(state = 0, action) {
        const {name} = action;
        if(name !== counterName) return state;

        switch (action.type) {
            case `INCREMENT`:
                return state + 1;
            case `DECREMENT`:
                return state - 1;
            default:
                return state;
        }
```

```
    }
}
```

现在我们应该可以使用它们任何一个去生成我们特定的计数器 reducer，然后发送 action，并只会影响关心的那部分的 state：

```
const rootReducer = combineReducers({
    counterA : createCounterWithNamedType('A'),
    counterB : createCounterWithNamedType('B'),
    counterC : createCounterWithNamedType('C'),
});

store.dispatch({type : 'INCREMENT_B'});
console.log(store.getState());
// {counterA : 0, counterB : 1, counterC : 0}
```

我们在某种程度上也可以改变这个方法，创建一个更加通用的高阶 reducer，它可以接收一个给定的 reducer，一个名字或者标识符：

```
function counter(state = 0, action) {
    switch (action.type) {
        case 'INCREMENT':
            return state + 1;
        case 'DECREMENT':
            return state - 1;
        default:
            return state;
    }
}

function createNamedWrapperReducer(reducerFunction, reducerName) {
    return (state, action) => {
        const {name} = action;
        const isInitializationCall = state === undefined;
        if(name !== reducerName && !isInitializationCall) return stat

        return reducerFunction(state, action);
    }
```

```
}

const rootReducer = combineReducers({
    counterA : createNamedWrapperReducer(counter, 'A'),
    counterB : createNamedWrapperReducer(counter, 'B'),
    counterC : createNamedWrapperReducer(counter, 'C'),
});
```

甚至还可以写一个通用的高阶 reducer 过滤器：

```
function createFilteredReducer(reducerFunction, reducerPredicate) {
    return (state, action) => {
        const isInitializationCall = state === undefined;
        const shouldRunWrappedReducer = reducerPredicate(action) || i
        return shouldRunWrappedReducer ? reducerFunction(state, actio
    }
}

const rootReducer = combineReducers({
    // 检查后缀
    counterA : createFilteredReducer(counter, action => action.type.e
    // 检查 action 中的额外数据
    counterB : createFilteredReducer(counter, action => action.name =
    // 响应所有的 'INCREMENT' action，但不响应 'DECREMENT'
    counterC : createFilteredReducer(counter, action => action.type =
};
```

这些基本的模式可以让你在 UI 内处理一个智能连接的 component 的多个实例。对于像分页或者排序这些通用的功能，可以复用相同的逻辑。

不可变更新模式

在 redux 基本概念的不可变数据管理 中给出一些示例，演示了不可变的基本更新操作，例如，更新一个对象中一个字段，或者，在数组的末尾增加一个数据。然而，reducer 经常需要综合使用这些基本操作去处理更加复杂的任务。下面是一些你可能必须去实现的常见任务的例子。

更新嵌套的对象

更新嵌套数据的关键是必须适当地复制和更新嵌套的每个级别。这往往是那些学习 redux 一个难以理解的概念，当试图更新嵌套对象的时候，有一些具体的问题会经常出现。这些意外的导致了直接变化，应该被避免。

常见错误 #1：指向同一对象的新变量

定义一个新变量不会创建一个新的实际对象，它只创建另一个引用到同一个对象。这个错误的示例如下：

```
function updateNestedState(state, action) {
    let nestedState = state.nestedState;
    // 错误：这将导致直接修改已经存在的对象引用-不要这么做!
    nestedState.nestedField = action.data;

    return {
        ...state,
        nestedState
    };
}
```

这个函数正确返回了顶层状态对象的浅复制，但是变量 `nestedState` 依然指向已经存在的对象，这个状态被直接修改了。

常见错误 #2：仅仅在一个层级上做浅复制

这个错误的另外一个常见版本的如下所示：

```
function updateNestedState(state, action) {
    // 问题：这仅仅做了浅复制！
    let newState = {...state};

    // 错误：nestedState 仍然是同一个对象！
    newState.nestedState.nestedField = action.data;

    return newState;
}
```

做一个顶层的浅复制是不够的 - `nestedState` 对象也应该被复制。

正确方法：复制嵌套数据的所有层级

不幸的是，正确地使用不变的更新去深度嵌套状态的过程很容易变得冗长难读。 更新 `ate.first.second[someId].fourth` 的示例大概如下所示：

```
function updateVeryNestedField(state, action) {
    return {
        ....state,
        first : {
            ...state.first,
            second : {
                ...state.first.second,
                [action.someId] : {
                    ...state.first.second[action.someId],
                    fourth : action.someValue
                }
            }
        }
    }
}
```

显然，每一层嵌套使得阅读更加困难，并给了更多犯错的机会。这是其中一个原因，鼓励你保持状态扁平，尽可能构建 reducer。

在数组中插入和删除数据

通常，一个 Javascript 数组中内容使用变化的函数来修改，例如，`push`，`unshift`，`shift`。因为我们不想在 reducer 中直接修改状态，这些通常应该被避免。正因如此，你可能会看到 “插入” 和 “删除” 的行为如下所示：

```
function insertItem(array, action) {
    return [
        ...array.slice(0, action.index),
        action.item,
        ...array.slice(action.index)
    ]
}

function removeItem(array, action) {
    return [
        ...array.slice(0, action.index),
        ...array.slice(action.index + 1)
    ];
}
```

但是，请记住，关键是原始内存中的引用没有被修改。**只要首先我们做了复制，我们就可以安全的变化这个复制。** 请注意，这个对于数组和对象都是正确的，但嵌套的数据仍然必须使用相同的规则更新。

这意味着我们也可以编写插入和删除函数如下所示：

```
function insertItem(array, action) {
    let newArray = array.slice();
    newArray.splice(action.index, 0, action.item);
    return newArray;
}

function removeItem(array, action) {
    let newArray = array.slice();
    newArray.splice(action.index, 1);
    return newArray;
```

```
}
```

删除函数也可以是这样：

```
function removeItem(array, action) {
    return array.filter( (item, index) => index !== action.index);
}
```

在一个数组中更新一个项目

更新数组的一项可以使用 `Array.map`，返回我们想要更新那项的一个新值，和其他项原先的值：

```
function updateObjectInArray(array, action) {
    return array.map( (item, index) => {
        if(index !== action.index) {
            // 这不是我们关心的项-保持原来的值
            return item;
        }

        // 否则，这是我们关心的-返回一个更新的值
        return {
            ...item,
            ...action.item
        };
    });
}
```

不可变更新工具库

因为编写不可变的更新代码可能变得乏味，所以有许多工具程序库试图抽象出这个过程。这些库在 API 和用法上有所不同，但都试图提供一种更短和更简洁的方式来编写这些更新。有些，像 dot-prop-immutable ，使用字符串路

径作为命令：

```
state = dotProp.set(state, `todos.${index}.complete`, true)
```

其他的，例如 immutability-helper （现在过时的 React 不可变助手插件的一个复制），使用嵌套数据和助手函数：

```
var collection = [1, 2, {a: [12, 17, 15]}];
var newCollection = update(collection, {2: {a: {$splice: [[1, 1, 13,
```

这些可以有效的替代了手写不可变更新逻辑。

许多不可变更新工具的列表可以在 Immutable Data#Immutable Update Utilities 的 Redux Addons Catalog 部分找到。

初始化 State

主要有两种方法来初始化应用的 state 。

可以使用 `createStore` 方法中的第二个可选参数 `preloadedState` 。

也可以在 reducer 中为 `undefined` 的 state 参数指定的默认的初始值。这个可以通过在 reducer 中添加一个明确的检查来完成，也可以使用 ES6 中默认参数的语法 `function myReducer(state = someDefaultValue, action)`

这两种方法是怎么相互影响的也许并不总是很清楚，不过幸运的是，这个过程遵循着一些可预见的规则，这里来说明它们是如何联系在一起的。

概要

如果不使用 `combineReducers()` 或者类似的代码，那么 `preloadedState` 总是会优先于在 reducer 里面使用 `state = ...` ，因为 `state` 传到 reducer 里的**是** `preloadedState` 的 state **而不是** `undefined` ，所以 ES6 的默认参数语法并不起作用。

如果使用 `combineReducers()` 方法，那么这里的行为就会有一些细微的差别了。那些指定了 `preloadedState` 的 reducer 会接收到那些对应的 state。而其他的 reducer 将会接收到 `undefined` 并因此回到了 `state = ...` 这里去获取指定的默认值。

通常情况下，通过 `preloadedState` 指定的 state 要优先于通过 reducer 指定 state。这样可以使通过 reducer 默认参数指定初始数据显得更加的合理，并且当你从一些持久化的存储器或服务器更新 store 的时候，允许你更新已存在的数据（全部或者部分）。

深度

单一简单的 Reducer

首先，让我们举一个单独的 reducer 的例子，并且不使用 `combineReducers()` 方法。

那么你的 reducer 可能像下面这样：

```
function counter(state = 0, action) {
  switch (action.type) {
  case 'INCREMENT': return state + 1;
  case 'DECREMENT': return state - 1;
  default: return state;
  }
}
```

现在让我们创建一个 store。

```
import { createStore } from 'redux';
let store = createStore(counter);
console.log(store.getState()); // 0
```

初始的状态是 0 。为什么呢？因为 `createStore` 的第二个参数是 `undefined`。这是 `state` 第一次传到了 reducer 当中。Redux 会发起（dispatch）一个“虚拟”的 action 来填充这个 state 。所以 `counter` reducer 里获得的 `state` 等于 `undefined`。**实际上这相当于触发了那个默认参数的特性**。因此，现在 `state` 的值是设定的默认参数 `0` ，这个 state (`0`) 是这里 reducer 的返回值。

接下来让我们举一个不同场景的例子：

```
import { createStore } from 'redux';
let store = createStore(counter, 42);
console.log(store.getState()); // 42
```

为什么这次这里的值是 `42` 而不是 `0` 呢？因为 `createStore` 的第二个参数是 `42` 。这个参数会赋给 state 并伴随着一个虚拟 action 一起传给 reducer。**这次，传给 reducer 的 `state` 不再是 `undefined` (是 `42`)，所以 ES6 的默认参数特性没有起作用。**所以这里的 `state` 是 `42`， `42` 是这个 reducer 的返回值。

组合多个 Reducers

现在让我们举一个使用 `combineReducers()` 的例子。

你有两个 reducers：

```
function a(state = 'lol', action) {
  return state;
}

function b(state = 'wat', action) {
  return state;
}
```

这个组合的 reducer 通过 `combineReducers({ a, b })` 生成，就像下面这样：

```
// const combined = combineReducers({ a, b })
function combined(state = {}, action) {
  return {
    a: a(state.a, action),
    b: b(state.b, action)
  };
}
```

如果我们调用 `createStore` 方法并且不使用 `preloadedState` 参数，它将会把 `state` 初始化成 `{}` 。 因此，传入 reducer 的 `state.a` 和 `state.b` 的值将会是 `undefined` **不论是 `a` 还是 `b` 的值都是 `undefined` ，这时如果指

定 `state` 的默认值，那么 reducer 将会返回这个默认值。 这就是第一次请求这个组合的 reducer 时返回 state `{ a: 'lol', b: 'wat'}` 的原因。

```
import { createStore } from 'redux';
let store = createStore(combined);
console.log(store.getState()); // { a: 'lol', b: 'wat' }
```

接下来让我们举一个不同的场景的例子：

```
import { createStore } from 'redux';
let store = createStore(combined, { a: 'horse' });
console.log(store.getState()); // { a: 'horse', b: 'wat' }
```

现在我们指定了 `createStore()` 的 `preloadedState` 参数。现在这个组合的 reducer 返回的 state 结合了我们指定给 `a` 的默认值 `horse` 以及通过 reducer 本身默认参数指定的 `b` 的值 `'wat'` 。

让我们重新看看这个组合的 reducer 做了什么：

```
// const combined = combineReducers({ a, b })
function combined(state = {}, action) {
  return {
    a: a(state.a, action),
    b: b(state.b, action)
  };
}
```

在这个案例中， `state` 是我们指定的，所以它并没有被赋值为 `{}` 。这是一个 `a` 的值是 `'horse'` 的对象，但是没有 `b` 的值。这就是为什么 `a` reducer 收到了 `'horse'` 的 state 并高兴地返回它，但 `b` reducer 却得到了 `undefined` state 并因此返回了它自己设定的默认值（在这个例子里是 `'wat'` ）。这就是我们如何得到的 `{ a: 'horse', b: 'wat' }` 这个结果。

总结

综上所述，如果你遵守 Redux 的约定并且要让 reducer 中 `undefined` 的 `state` 参数返回初始 state （最简单的实现方法就是利用 ES6 的默认参数来指定 state），那么对于组合多个 reducers 的情况，这将是一个很有用的做法，**他们会优先选择通过 `preloadedState` 参数传到 `createStore()` 的对象中的相应值，但是如果你不传任何东西，或者没设置相应的字段，那么 reducer 就会选择指定的默认 `state` 参数来取代**。这样的方法效果很好，因为他既能用来初始化也可以用来更新现有的数据，不过如果数据没有保护措施的话，这样做也会使一些独立的 reducer 的 state 被重新赋值。当然你可以递归地使用这个模式，比如你可以在多个层级上使用 `combineReducers()` 方法，或者甚至手动的组合这些 reducer 并且传入对应部分的 state tree。

Redux 常见问题

目录

- **代码结构**
  - 文件结构应该是什么样？项目中该如何对 action 创建函数和 reducer 分组？ selector 又该放在哪里？
  - 如何将逻辑在 reducer 和 action 创建函数之间划分？ “业务逻辑” 应该放在哪里？
- **性能**
  - 考虑到性能和架构， Redux “可扩展性” 如何？
  - 每个 action 都调用 “所有的 reducer” 会不会很慢？
  - 在 reducer 中必须对 state 进行深拷贝吗？拷贝 state 不会很慢吗？
  - 怎样减少 store 更新事件的数量？
  - 仅有 “一个 state 树” 会引发内存问题吗？分发多个 action 会占用内存空间吗？
- **React Redux**
  - 为何组件没有被重新渲染、或者 mapStateToProps 没有运行？
  - 为何组件频繁的重新渲染？
  - 怎样使 mapStateToProps 执行更快？
  - 为何不在被连接的组件中使用 this.props.dispatch ？
  - 应该只连接到顶层组件吗，或者可以在组件树中连接到不同组件吗？
- **其它**
  - 有 “真实存在” 且很庞大的 Redux 项目吗？
  - 如何在 Redux 中实现鉴权？

Redux 常见问题：综合

目录

综合

何时使用 Redux?

React 早期贡献者之一 Pete Hunt 说：

> 你应当清楚何时需要 Flux。如果你不确定是否需要它，那么其实你并不需要它。

Redux 的创建者之一 Dan Abramov 也曾表达过类似的意思:

> 我想修正一个观点：当你在使用 React 遇到问题时，才使用 Redux。

一般而言，如果随着时间的推移，数据处于合理的变动之中、需要一个单一的数据源、在 React 顶层组件 state 中维护所有内容的办法已经无法满足需求，这个时候就需要使用 Redux 了。

在打算使用 Redux 的时候进行权衡是非常重要的。它从设计之初就不是为了编写最短、最快的代码，他是为了解决 “当有确定的状态发生改变时，数据从哪里来” 这种可预测行为的问题的。它要求你在应用程序中遵循特定的约定：应用的状态需要存储为纯数据的格式、用普通的对象描述状态的改变、用不可更新的纯函数式方式来处理状态变化。这也成了抱怨是“样板代码”的来源。这些约束需要开发人员一起来努力维护，但也打开了一扇扇可能的大门（比如：数据持久性、同步）。

如果你只是刚开始学习 React，你应该首先专注于 React，然后再看看 Redux 是否适合于你的应用。

最后需要说明的是：Redux 仅仅是个工具。它是一个伟大的工具，经常有一个很棒的理由去使用它，但也有很多的理由不去使用它。时刻注意对你的工具做出明确的决策，并且权衡每个决策带来的利弊。

补充资料

文档

- Introduction: Motivation

文章

- React How-To
- You Might Not Need Redux
- The Case for Flux

讨论

- Twitter: Don't use Redux until...
- Twitter: Redux is designed to be predictable, not concise
- Twitter: Redux is useful to eliminate deep prop passing
- Stack Overflow: Why use Redux over Facebook Flux?
- Stack Overflow: Why should I use Redux in this example?
- Stack Overflow: What could be the downsides of using Redux instead of Flux?
- Stack Overflow: When should I add Redux to a React app?

Redux 只能搭配 React 使用?

Redux 能作为任何 UI 层的 store。通常是与 React 或 React Native 搭配使用，但是也可以绑定 Angular、 Angular 2、 Vue、 Mithril 等框架使用。Redux 提供的订阅机制，可以与任何代码集成。这就是说，在结合 UI 随

state 变化的声明式视图时（如 React 或者其他相似的库），Redux 就发挥它的最大作用。

Redux 需要特殊的编译工具支持吗?

Redux 写法遵循 ES6 语法，但在发布时被 Webpack 和 Babel 编译成了 ES5，所以在使用时可以忽略 JavaScript 的编译过程。 Redux 也提供了 UMD 版本，可以直接使用而不需要任何编译过程。counter-vanilla 示例用 `<script>` 标签的方式展示了 Redux 基本的 ES5 用法。正如相关 pull request 中的说法：

> Counter Vanilla 例子意图是消除 Redux 需要 Webpack、 React、 热重载、 sagas、 action 创建函数、 constants、 Babel、 npm、 CSS 模块化、 decorators、 fluent Latin、 Egghead subscription、 博士学位或者需要达到 Exceeds Expectations O.W.L. 这一级别的荒谬观点。
>
> 仅仅是 HTML， 一些 `<script>` 标签，和简单的 DOM 操作而已。

Redux 常见问题：Reducer

目录

Reducer

如何在 reducer 之间共享 state? `combineReducers` 是必须的吗?

Redux store 推荐的结构是将 state 对象按键值切分成 “层”（slice） 或者 “域”（domain），并提供独立的 reducer 方法管理各自的数据层。就像 Flux 模式中的多个独立 store 一样， Redux 为此还提供了 `combineReducers` 工具来简化该模型。应当注意的是， `combineReducers` **不是** 必须的，它仅仅是通过简单的 JavaScript 对象作为数据，让 state 层能与 reducer 一一关联的函数而已。

许多用户想在 reducer 之间共享数据，但是 `combineReducers` 不允许此种行为。有许多可用的办法：

- 如果一个 reducer 想获取其它 state 层的数据，往往意味着 state 树需要重构，需要让单独的 reducer 处理更多的数据。
- 你可能需要自定义方法去处理这些 action，用自定义的顶层 reducer 方法替换 `combineReducers`。你可以使用类似于 reduce-reducers 的工具运行 `combineReducers` 去处理尽可能多的 action，同时还要为存在 state 交叉部分的若干 action 执行更专用的 reducer。
- 类似于 `redux-thunk` 的 异步 action 创建函数 能通过 `getState()` 方法获取所有的 state。 action 创建函数能从 state 中检索到额外的数据并传入 action，所以 reducer 有足够的信息去更新所维护的 state 层。

只需牢记 reducer 仅仅是函数，可以随心所欲的进行划分和组合，而且也推荐将其分解成更小、可复用的函数 (“reducer 合成”)。按照这种做法，如果子 reducer 需要一些参数时，可以从父 reducer 传入。你只需要确保他们遵循 reducer 的基本准则：`(state, action) => newState`，并且以不可变的方式更新 state，而不是直接修改 state。

补充资料

文档

- API: combineReducers
- Recipes: Reducers 基础结构

讨论

- #601: A concern on combineReducers, when an action is related to multiple reducers
- #1400: Is passing top-level state object to branch reducer an anti-pattern?
- Stack Overflow: Accessing other parts of the state when using combined reducers?
- Stack Overflow: Reducing an entire subtree with redux combineReducers
- Sharing State Between Redux Reducers

处理 action 必须用 `switch` 语句吗?

不是。在 reducer 里面你可以使用任何方法响应 action。`switch` 语句是最常用的方式，当然你也可以用 `if`、功能查找表、创建抽象函数等。事实上，虽然 Redux 要求每个 action 对象都有一个 `type` 的字段，但是你的 reducer 逻辑不必一定要依赖它做处理。也就是说，标准方法肯定是用基于 `type` 的 `switch` 语句或者查找表。

补充资料

文档

- Recipes: Reducing Boilerplate

讨论

- #883: take away the huge switch block
- #1167: Reducer without switch

Redux 常见问题：组织 State

目录

组织 State

必须将所有 state 都维护在 Redux 中吗？ 可以用 React 的 `setState()` 方法吗?

没有 “标准”。有些用户选择将所有数据都在 Redux 中维护，那么在任何时刻，应用都是完全有序及可控的。也有人将类似于“下拉菜单是否打开”的非关键或者 UI 状态，在组件内部维护。适合自己的才是最好的。

使用局部组件状态是更好的。作为一名开发者，应该决定使用何种 state 来组装你的应用，每个 state 的生存范围是什么。在两者之间做好平衡，然后就去做吧。

这里有一些将怎样的数据放入 Redux 的经验法则：

- 应用的其他部分是否关心这个数据?
- 是否需要根据需要在原始数据的基础上创建衍生数据?
- 相同的数据是否被用作驱动多个组件?
- 能否将状态恢复到特定时间点（在时光旅行调试的时候）？
- 是否要缓存数据（比如：数据存在的情况下直接去使用它而不是重复去请求他）？

有许多开源组件实现了各式各样在 Redux store 存储独立组件状态的替代方法，比如 redux-ui、 redux-component、 redux-react-local等等。还可以将 Redux 的原则和 reducers 的概念应用到组件层面，按照 `this.setState((previousState) => reducer(previousState, someAction))` 的情形。

补充资料

文章

- You Might Not Need Redux
- Finding `state`'s place with React and Redux.
- A Case for setState
- How to handle state in React. The missing FAQ.
- Where to Hold React Component Data: state, store, static, and this
- The 5 Types Of React Application State

讨论

- #159: Investigate using Redux for pseudo-local component state
- #1098: Using Redux in reusable React component
- #1287: How to choose between Redux's store and React's state?
- #1385: What are the disadvantages of storing all your state in a single immutable atom?
- Twitter: Should I keep something in React component state?
- Twitter: Using a reducer to update a component
- React Forums: Redux and global state vs local state
- Reddit: "When should I put something into my Redux store?"
- Stack Overflow: Why is state all in one place, even state that isn't global?
- Stack Overflow: Should all component state be kept in Redux store?

库

- Redux Addons Catalog: Component State

可以将 store 的 state 设置为函数、promise或者其它非序列化值吗?

强烈推荐只在 store 中维护普通的可序列化对象、数组以及基本数据类型。虽然从 技术 层面上将非序列化项保存在 store 中是可行的，但这样会破坏 store 内容持久化和恢复能力，以及会干扰时间旅行。

如果你不关心数据持久化和时间旅行，那么完全欢迎把不可以持久化的数据放入 Redux 的 Store 中存储。最终，他是你的应用程序，如何实现完全取决于你自己。与其他很多 Redux 的事情一样，你需要明白权衡所需。

补充资料

讨论

- #1248: Is it ok and possible to store a react component in a reducer?
- #1279: Have any suggestions for where to put a Map Component in Flux?
- #1390: Component Loading
- #1407: Just sharing a great base class
- #1793: React Elements in Redux State

如何在 state 中组织嵌套及重复数据?

当数据存在 ID、嵌套或者关联关系时，应当以 “范式化” 形式存储：对象只能存储一次，ID 作为键值，对象间通过 ID 相互引用。将 store 类比于数据库，每一项都是独立的 “表”。normalizr 、 redux-orm 此类的库能在管理规范化数据时提供参考和抽象。

补充资料

文档

- Advanced: Async Actions

- Examples: Real World example
- Recipes: Structuring Reducers - Prerequisite Concepts
- Recipes: Structuring Reducers - Normalizing State Shape
- Examples: Tree View

文章

- High-Performance Redux
- https://medium.com/@adamrackis/querying-a-redux-store-37db8c7f3b0f

讨论

- #316: How to create nested reducers?
- #815: Working with Data Structures
- #946: Best way to update related state fields with split reducers?
- #994: How to cut the boilerplate when updating nested entities?
- #1255: Normalizr usage with nested objects in React/Redux
- #1269: Add tree view example
- #1824: Normalising state and garbage collection
- Twitter: state shape should be normalized
- Stack Overflow: How to handle tree-shaped entities in Redux reducers?
- Stack Overflow: How to optimize small updates to props of nested components in React + Redux?

Redux 常见问题：创建 Store

目录

创建 Store

可以创建多个 store 吗，应该这么做吗？能在组件中直接引用 store 并使用吗?

Flux 原始模型中一个应用有多个 “store”，每个都维护了不同维度的数据。这样导致了类似于一个 store “等待” 另一 store 操作的问题。Redux 中将 reducer 分解成多个小而美的 reducer，进而切分数据域，避免了这种情况的发生。

正如上述问题所述，“可能” 在一个页面中创建多个独立的 Redux store，但是预设模式中只会有一个 store。仅维持单个 store 不仅可以使用 Redux DevTools，还能简化数据的持久化及深加工、精简订阅的逻辑处理。

在 Redux 中使用多个 store 的理由可能包括：

- 对应用进行性能分析时，解决由于过于频繁更新部分 state 引起的性能问题。
- 在更大的应用中 Redux 只是作为一个组件，这种情况下，你也许更倾向于为每个根组件创建单独的 store。

然而，创建新的 store 不应成为你的第一反应，特别是当你从 Flux 背景迁移而来。首先尝试组合 reducer，只有当它无法解决你的问题时才使用多个 store。

类似的，虽然你 能 直接导入并获取 store 实例，但这并非 Redux 的推荐方式。当你创建 store 实例并从组件导出，它将变成一个单例。这意味着将很难把 Redux 应用封装成一个应用的子组件，除非这是必要的，或者为了实现服务端渲染需要为每一个请求创建单独的 store 实例。

借助 React Redux，由 `connect()` 生成的包装类实际上会检索存在的 `props.store`，但还是推荐将根组件包装在 `<Provider store={store}>` 中，这样传递 store 的任务都交由 React Redux 处理。这种方式，我们不用考虑 store 模块的导入、 Redux 应用的封装，后期支持服务器渲染也将变得更为简便。

补充资料

文档

- API: Store

讨论

- #1346: Is it bad practice to just have a 'stores' directory?
- Stack Overflow: Redux multiple stores, why not?
- Stack Overflow: Accessing Redux state in an action creator
- Gist: Breaking out of Redux paradigm to isolate apps

在 store enhancer 中可以存在多个 middleware 链吗？ 在 middleware 方法中，`next` 和 `dispatch` 之间区别是什么?

Redux middleware 就像一个链表。每个 middleware 方法既能调用 `next(action)` 传递 action 到下一个 middleware，也可以调用

`dispatch(action)` 重新开始处理，或者什么都不做而仅仅终止 action 的处理进程。

创建 store 时， `applyMiddleware` 方法的入参定义了 middleware 链。定义多个链将无法正常执行，因为它们的 `dispatch` 引用显然是不一样的，而且不同的链也无法有效连接到一起。

补充资料

文档

- Advanced: Middleware
- API: applyMiddleware

讨论

- #1051: Shortcomings of the current applyMiddleware and composing createStore
- Understanding Redux Middleware
- Exploring Redux Middleware

怎样只订阅 state 的一部分变更？如何将分发的 action 作为订阅的一部分？

Redux 提供了独立的 `store.subscribe` 方法用于通知监听器 store 的变更信息。监听器的回调方法并没有把当前的 state 作为入参，它仅仅代表了 有些数据 被更新。订阅者的逻辑中调用 `getState()` 获取当前的 state 值。

这个 API 是没有依赖及副作用的底层接口，可以用于创建高阶订阅者逻辑。类似 React Redux 的 UI 绑定能为所有连接的组件都创建订阅。也可以用于编写智能的新旧 state 比对方法，从而在某些内容变化时执行额外的逻辑处理。示例 redux-watch 和 redux-subscribe 提供不同的方式用于指定订阅及处理变更。

新的 state 没有传递给监听者，目的是简化 store enhancer 的实现，比如

Redux DevTools。此外，订阅者旨在响应 state 值本身，而非 action。当 action 很重要且需要特殊处理时，使用 middleware。

补充资料

文档

- Basics: Store
- API: Store

讨论

- #303: subscribe API with state as an argument
- #580: Is it possible to get action and state in store.subscribe?
- #922: Proposal: add subscribe to middleware API
- #1057: subscribe listener can get action param?
- #1300: Redux is great but major feature is missing

库

- Redux Addons Catalog: Store Change Subscriptions

Redux 常见问题：Action

目录

Actions

为何 `type` 必须是字符串，或者至少可以被序列化？为什么 action 类型应该作为常量?

和 state 一样，可序列化的 action 使得若干 Redux 的经典特性变得可能，比如时间旅行调试器、录制和重放 action。若使用 `Symbol` 等去定义 `type` 值，或者用 `instanceof` 对 action 做自检查都会破坏这些特性。字符串是可序列化的、自解释型，所以是更好的选择。注意，如果 action 目的是在 middleware 中处理，那么使用 Symbols、 Promises 或者其它非可序列化值也是 可以 的。 action 只有当它们正真到达 store 且被传递给 reducer 时才需要被序列化。

因为性能原因，我们无法强制序列化 action，所以 Redux 只会校验 action 是否是普通对象，以及 `type` 是否定义。其它的都交由你决定，但是确保数据是可序列化将对调试以及问题的重现有很大帮助。

封装并集聚公共代码是程序规划时的核心概念。虽然可以在任何地方手动创建 action 对象、手动指定 `type` 值，定义常量的方式使得代码的维护更为方便。如果将常量维护在单独的文件中，在 `import` 时校验，能避免偶然的拼

写错误。

补充资料

文档

- Reducing Boilerplate

Discussion

- #384: Recommend that Action constants be named in the past tense
- #628: Solution for simple action creation with less boilerplate
- #1024: Proposal: Declarative reducers
- #1167: Reducer without switch
- Stack Overflow: Why do you need 'Actions' as data in Redux?
- Stack Overflow: What is the point of the constants in Redux?

是否存在 reducer 和 action 之间的一对一映射?

不存在。建议的方式是编写独立且很小的 reducer 方法去更新指定的 state 部分，这种模式被称为 “reducer 合成”。一个指定的 action 也许被它们中的全部、部分、甚至没有一个处理到。这种方式把组件从实际的数据变更中解耦，一个 action 可能影响到 state 树的不同部分，对组件而言再也不必知道这些了。有些用户选择将它们紧密绑定在一起，就像 “ducks” 文件结构，显然是没有默认的一对一映射。所以当你想在多个 reducer 中处理同一个 action 时，应当避免此类结构。

补充资料

文档

- Basics: Reducers
- Recipes: Structuring Reducers

讨论

- Twitter: most common Redux misconception
- #1167: Reducer without switch
- Reduxible #8: Reducers and action creators aren't a one-to-one mapping
- Stack Overflow: Can I dispatch multiple actions without Redux Thunk middleware?

怎样表示类似 AJAX 请求的 “副作用”？为何需要 “action 创建函数”、“thunks” 以及 “middleware” 类似的东西去处理异步行为？

这是一个持久且复杂的话题，针对如何组织代码以及采用何种方式有很多的观点。

任何有价值的 web 应用都必然要执行复杂的逻辑，通常包括 AJAX 请求等异步工作。这类代码不再是针对输入的纯函数，与第三方的交互被认为是 “副作用”。

Redux 深受函数式编程的影响，创造性的不支持副作用的执行。尤其是 reducer， 必须 是符合 `(state, action) => newState` 的纯函数。然而，Redux 的 middleware 能拦截分发的 action 并添加额外的复杂行为，还可以添加副作用。

Redux 建议将带副作用的代码作为 action 创建过程的一部分。因为该逻辑能 在 UI 组件内执行，那么通常抽取此类逻辑作为可重用的方法都是有意义的，因此同样的逻辑能被多个地方调用，也就是所谓的 action 创建函数。

最简单也是最常用的方法就是使用 Redux Thunk middleware，这样就能用更为复杂或者异步的逻辑书写 action 创建函数。另一个被广泛使用的方法是 Redux Saga，你可以用 generator 书写类同步代码，就像在 Redux 应用中使用 “后台线程” 或者 “守护进程”。还有一个方法是 Redux Loop，它允许 reducer 以声明副作用的方式去响应 state 变化，并让它们分别执行，从而反转了进程。除此之外，还有 许多 其它开源的库和理念，都有各自针对副作用

的管理方法。

补充资料

文档

- Advanced: Async Actions
- Advanced: Async Flow
- Advanced: Middleware

文章

- Redux side effects and you
- Pure functionality and side effects in Redux
- From Flux to Redux: Async Actions the easy way
- React/Redux Links: "Redux Side Effects" category
- Gist: Redux-Thunk examples

讨论

- #291: Trying to put API calls in the right place
- #455: Modeling side effects
- #533: Simpler introduction to async action creators
- #569: Proposal: API for explicit side effects
- #1139: An alternative side effect model based on generators and sagas
- Stack Overflow: Why do we need middleware for async flow in Redux?
- Stack Overflow: How to dispatch a Redux action with a timeout?
- Stack Overflow: Where should I put synchronous side effects linked to actions in redux?
- Stack Overflow: How to handle complex side-effects in Redux?
- Stack Overflow: How to unit test async Redux actions to mock ajax response
- Stack Overflow: How to fire AJAX calls in response to the state changes with Redux?
- Reddit: Help performing Async API calls with Redux-Promise

Middleware.
- Twitter: possible comparison between sagas, loops, and other approaches

是否应该在 action 创建函数中连续分发多个 action?

关于如何构建 action 并没有统一的规范。使用类似 Redux Thunk 的异步 middleware 支持了更多的场景，比如分发连续多个独立且相关联的 action、分发 action 指示 AJAX 请求的阶段、 根据 state 有条件的分发 action、甚至分发 action 并随后校验更新的 state。

通常，明确这些 action 是关联还是独立，是否应当作为一个 action。评判当前场景影响因素的同时，还需根据 action 日志权衡 reducer 的可读性。例如，一个包含新 state 树的 action 会使你的 reducer 只有一行，副作用是没有任何历史表明 为什么 发生了变更，进而导致调试异常困难。另一方面，如果为了维持它们的粒状结构（granular），在循环中分发 action，这表明也许需要引入新的 acton 类型并以不同的方式去处理它。

避免在同一地方连续多次以同步的方式进行分发，其性能问题是值得担忧的。有许多插件和方法可以批处理调度。

补充资料

文档

- FAQ: Performance - Reducing Update Events

讨论

- #597: Valid to dispatch multiple actions from an event handler?
- #959: Multiple actions one dispatch?
- Stack Overflow: Should I use one or several action types to represent this async action?

- Stack Overflow: Do events and actions have a 1:1 relationship in Redux?
- Stack Overflow: Should actions be handled by reducers to related actions or generated by action creators themselves?

Redux 常见问题：代码结构

目录

代码结构

文件结构应该是什么样？项目中该如何对 action 创建函数和 reducer 分组？ selector 又该放在哪里？

因为 Redux 只是数据存储的库，它没有关于工程应该被如何组织的直接主张。然后，有一些被大多数 Redux 开发者所推荐的模式：

- Rails-style：“actions”、“constants”、“reducers”、“containers” 以及 “components” 分属不同的文件夹
- Domain-style：为每个功能或者域创建单独的文件夹，可能会为某些文件类型创建子文件夹
- “Ducks”：类似于 Domain-style，但是明确地将 action、 reducer 绑定在一起，通常将它们定义在同一文件内。

推荐做法是将 selector 与 reducer 定义在一起并输出，并在 reducer 文件中与知道 state 树真实形状的代码一起被重用（例如在 `mapStateToProps` 方法、异步 action 创建函数，或者 saga）。

不管代码在你的磁盘上是如何存放的，必须记住的是 action 和 reducer 不应该单独考虑。在一个文件夹中定义的 reducer 可以响应另一个文件夹中定义的 action 是非常常见的（甚至是鼓励的）。

补充资料

文档

- FAQ: Actions - "1:1 mapping between reducers and actions?"

文章

- How to Scale React Applications(accompanying talk:Scaling React Applications)
- Redux Best Practices
- Rules For Structuring (Redux) Applications
- A Better File Structure for React/Redux Applications
- Organizing Large React Applications
- Four Strategies for Organizing Code
- Encapsulating the Redux State Tree
- Redux Reducer/Selector Asymmetry
- Modular Reducers and Selectors
- My journey towards a maintainable project structure for React/Redux
- React/Redux Links: Architecture - Project File Structure

讨论

- #839: Emphasize defining selectors alongside reducers
- #943: Reducer querying
- React Boilerplate #27: Application Structure
- Stack Overflow: How to structure Redux components/containers
- Twitter: There is no ultimate file structure for Redux

如何将逻辑在 reducer 和 action 创建函数之间划分？“业务逻辑”应该放在哪里？

关于逻辑的哪个部分应该放在 reducer 或者 action 创建函数中，没有清晰的答案。一些开发者喜欢 “fat” action 创建函数，“thin” reducer 仅仅从 action

拿到数据并绑定到 state 树。其他人的则强调 action 越简单越好，尽量减少在 action 创建函数中使用 `getState()` 方法。

下面的评论恰如其分的概括了这两种分歧：

> 问题是什么在 action 创建函数中、什么在 reducer 中，就是关于 fat 和 thin action 创建函数的选择。如果你将逻辑都放在 action 创建函数中，最终用于更新 state 的 action 对象就会变得 fat，相应的 reducer 就变得纯净、简洁。因为只涉及很少的业务逻辑，将非常有利于组合。 如果你将大部分逻辑置于 reducer 之中，action 将变得精简、美观，大部分数据逻辑都在一个地方维护，但是 reducer 由于引用了其它分支的信息，将很难组合。最终的 reducer 会很庞大，而且需要从更高层的 state 获取额外信息。

当你从这两种极端情况中找到一个平衡时，就意味着你已经掌握了 Redux。

补充资料

文章

- Where do I put my business logic in a React/Redux application?
- How to Scale React Applications

讨论

- #1165: Where to put business logic / validation?
- #1171: Recommendations for best practices regarding action-creators, reducers, and selectors
- Stack Overflow: Accessing Redux state in an action creator?

Redux 常见问题：性能

目录

性能

考虑到性能和架构， Redux “可扩展性” 如何?

没有一个明确的答案，在大多数情况下都不需要考虑该问题。

Redux 所做的工作可以分为以下几部分：在 middleware 和 reducer 中处理 action （包括对象复制及不可变更新）、 action 分发之后通知订阅者、根据 state 变化更新 UI 组件。虽然在一些复杂场景下，这些都 可能 变成一个性能问题，但 Redux 本质上并没有任何慢或者低效的实现。实际上，React Redux 已经做了大量的优化工作减少不必要的重复渲染，React Redux v5 相比之前的版本有着显著的改进。

与其他库相比，Redux 可能没有那么快。为了更大限度的展示 React 的渲染性能，state 应该以规范化的结构存储，许多单独的组件应该直接连接到 store，连接的列表组件应该将项目 ID 传给子列表（允许列表项通过 ID 查找数据）。这使得要进行渲染的量最小化。使用带有记忆功能的 selector 函数也对性能有非常大的帮助。

考虑到架构方面，事实证据表明在各种项目及团队规模下，Redux 都表现出

色。Redux 目前正被成百上千的公司以及更多的开发者使用着，NPM 上每月都有几十万的安装量。有一位开发者这样说：

> 规模方面，我们大约有500个 action 类型、400个 reducer、150个组件、5个 middleware、200个 action、2300个测试案例。

补充资料

文档

- Recipes: Structuring Reducers - state 范式化

文章

- How to Scale React Applications (accompanying talk: Scaling React Applications)
- High-Performance Redux
- Improving React and Redux Perf with Reselect
- Encapsulating the Redux State Tree
- React/Redux Links: Performance - Redux

讨论

- #310: Who uses Redux?
- #1751: Performance issues with large collections
- React Redux #269: Connect could be used with a custom subscribe method
- React Redux #407: Rewrite connect to offer an advanced API
- React Redux #416: Rewrite connect for better performance and extensibility
- Redux vs MobX TodoMVC Benchmark: #1
- Reddit: What's the best place to keep the initial state?
- Reddit: Help designing Redux state for a single page app
- Reddit: Redux performance issues with a large state object?
- Reddit: React/Redux for Ultra Large Scale apps

- Twitter: Redux scaling
- Twitter: Redux vs MobX benchmark graph - Redux state shape matters
- Stack Overflow: How to optimize small updates to props of nested components?
- Chat log: React/Redux perf - updating a 10K-item Todo list
- Chat log: React/Redux perf - single connection vs many connections

每个 action 都调用 “所有的 reducer” 会不会很慢?

我们应当清楚的认识到 Redux store 只有一个 reducer 方法。 store 将当前的 state 和分发的 action 传递给这个 reducer 方法，剩下的就让 reducer 去处理。

显然，在单独的方法里处理所有的 action 仅从方法大小及可读性方面考虑，就已经很不利于扩展了，所以将实际工作分割成独立的方法并在顶层的 reducer 中调用就变得很有意义。尤其是目前的建议模式中推荐让单独的子 reducer 只负责更新特定的 state 部分。 `combineReducers()` 和 Redux 搭配的方案只是许多实现方式中的一种。强烈建议尽可能保持 store 中 state 的扁平化和范式化，至少你可以随心所欲的组织你的 reducer 逻辑。

即使你在不经意间已经维护了许多独立的子 reducer，甚至 state 也是深度嵌套，reducer 的速度也并不构成任何问题。JavaScript 引擎有足够的能力在每秒运行大量的函数调用，而且大部门的子 reducer 只是使用 `switch` 语句，并且针对大部分 action 返回的都是默认的 state。

如果你仍然关心 reducer 的性能，可以使用类似 redux-ignore 和 reduxr-scoped-reducer 的工具，确保只有某几个 reducer 响应特定的 action。你还可以使用 redux-log-slow-reducers 进行性能测试。

补充资料

讨论

- #912: Proposal: action filter utility

- #1303: Redux Performance with Large Store and frequent updates
- Stack Overflow: State in Redux app has the name of the reducer
- Stack Overflow: How does Redux deal with deeply nested models?

在 reducer 中必须对 state 进行深拷贝吗？拷贝 state 不会很慢吗?

以不可变的方式更新 state 意味着浅拷贝，而非深拷贝。相比于深拷贝，浅拷贝更快，因为只需复制很少的字段和对象，实际的底层实现中也只是移动了若干指针而已。

因此，你需要创建一个副本，并且更新受影响的各个嵌套的对象层级即可。尽管上述动作代价不会很大，但这也是为什么需要维护范式化及扁平化 state 的又一充分理由。

> Redux 常见的误解： 需要深拷贝 state。实际情况是：如果内部的某些数据没有改变，继续保持统一引用即可。

补充资料

文档

- Recipes: Structuring Reducers - Prerequisite Concepts
- Recipes: Structuring Reducers - Immutable Update Patterns

讨论

- #454: Handling big states in reducer
- #758: Why can't state be mutated?
- #994: How to cut the boilerplate when updating nested entities?
- Twitter: common misconception - deep cloning
- Cloning Objects in JavaScript

怎样减少 store 更新事件的数量?

Redux 在 action 分发成功（例如，action 到达 store 被 reducer 处理）后通知订阅者。在有些情况下，减少订阅者被调用的次数会很有用，特别在当 action 创建函数分发了一系列不同的 action 时。

如果你在使用 React，你可以写在 `ReactDOM.unstable_batchedUpdates()` 以提高同步分发的性能，但这个 API 是实验性质的，可能会在以后的版本中移除，所以也不要过度依赖它。可以看看一些第三方的实现 redux-batched-subscribe（一个高级的 reducer，可以让你单独分发几个 action）、redux-batched-subscribe（一个 store 增强器，可以平衡多个分发情况下订阅者的调用次数）和 redux-batched-actions（一个 store 增强器，可以利用单个订阅提醒的方式分发一系列的 action）。

补充资料

讨论

- #125: Strategy for avoiding cascading renders
- #542: Idea: batching actions
- #911: Batching actions
- #1813: Use a loop to support dispatching arrays
- React Redux #263: Huge performance issue when dispatching hundreds of actions

库

- Redux Addons Catalog: Store - Change Subscriptions

仅有 “一个 state 树” 会引发内存问题吗？分发多个 action 会占用内存空间吗？

首先，在原始内存使用方面，Redux 和其它的 JavaScript 库并没有什么不同。唯一的区别就是所有的对象引用都嵌套在同一棵树中，而不是像类似于 Backbone 那样保存在不同的模型实例中。第二，与同样的 Backbone 应用相比，典型的 Redux 应用可能使用 更少 的内存，因为 Redux 推荐使用普通

的 JavaScript 对象和数组，而不是创建模型和集合实例。最后，Redux 仅维护一棵 state 树。不再被引用的 state 树通常都会被垃圾回收。

Redux 本身不存储 action 的历史。然而，Redux DevTools 会记录这些 action 以便支持重放，而且也仅在开发环境被允许，生产环境则不会使用。

补充资料

文档

- Docs: Async Actions

讨论

- Stack Overflow: Is there any way to "commit" the state in Redux to free memory?
- Reddit: What's the best place to keep initial state?

Redux 常见问题：React Redux

目录

React Redux

为何组件没有被重新渲染、或者 mapStateToProps 没有运行?

目前来看，导致组件在 action 分发后却没有被重新渲染，最常见的原因是对 state 进行了直接修改。Redux 期望 reducer 以 “不可变的方式” 更新 state，实际使用中则意味着复制数据，然后更新数据副本。如果直接返回同一对象，即使你改变了数据内容，Redux 也会认为没有变化。类似的，React Redux 会在 `shouldComponentUpdate` 中对新的 props 进行浅层的判等检查，以期提升性能。如果所有的引用都是相同的，则返回 `false` 从而跳过此次对组件的更新。

需要注意的是，不管何时更新了一个嵌套的值，都必须同时返回上层的任何数据副本给 state 树。如果数据是 `state.a.b.c.d`，你想更新 `d`，你也必须返回 `c`、`b`、`a` 以及 `state` 的拷贝。state 树变化图 展示了树的深层变化为何需要改变途经的结点。

“以不可变的方式更新数据” 并 不 代表你必须使用 Immutable.js, 虽然是很好的选择。你可以使用多种方法，达到对普通 JS 对象进行不可变更新的目的：

- 使用类似于 `Object.assign()` 或者 `_.extend()` 的方法复制对象，`slice()` 和 `concat()` 方法复制数组。
- ES6 数组的 spread sperator（展开运算符），JavaScript 新版本提案中类似的对象展开运算符。
- 将不可变更新逻辑包装成简单方法的工具库。

补充资料

文档

- Troubleshooting
- React Redux: Troubleshooting
- Recipes: Using the Object Spread Operator
- Recipes: Structuring Reducers - Prerequisite Concepts
- Recipes: Structuring Reducers - Immutable Update Patterns

文章

- Pros and Cons of Using Immutability with React
- React/Redux Links: Immutable Data

讨论

- #1262: Immutable data + bad performance
- React Redux #235: Predicate function for updating component
- React Redux #291: Should mapStateToProps be called every time an action is dispatched?
- Stack Overflow: Cleaner/shorter way to update nested state in Redux?
- Gist: state mutations

为何组件频繁的重新渲染?

React Redux 采取了很多的优化手段，保证组件直到必要时才执行重新渲染。一种是对 `mapStateToProps` 和 `mapDispatchToProps` 生成后传入 `connect` 的 props 对象进行浅层的判等检查。遗憾的是，如果当

`mapStateToProps` 调用时都生成新的数组或对象实例的话，此种情况下的浅层判等不会起任何作用。一个典型的示例就是通过 ID 数组返回映射的对象引用，如下所示：

```
const mapStateToProps = (state) => {
  return {
    objects: state.objectIds.map(id => state.objects[id])
  }
}
```

尽管每次数组内都包含了同样的对象引用，数组本身却指向不同的引用，所以浅层判等的检查结果会导致 React Redux 重新渲染包装的组件。

这种额外的重新渲染也可以避免，使用 reducer 将对象数组保存到 state，利用 Reselect 缓存映射的数组，或者在组件的 `shouldComponentUpdate` 方法中，采用 `_.isEqual` 等对 props 进行更深层次的比较。注意在自定义的 `shouldComponentUpdate()` 方法中不要采用了比重新渲染本身更为昂贵的实现。可以使用分析器评估方案的性能。

对于独立的组件，也许你想检查传入的 props。一个普遍存在的问题就是在 render 方法中绑定父组件的回调，比如 `<Child onClick={this.handleClick.bind(this)} />`。这样就会在每次父组件重新渲染时重新生成一个函数的引用。所以只在父组件的构造函数中绑定一次回调是更好的做法。

补充资料

文档

- FAQ: Performance - Scaling

文章

- A Deep Dive into React Perf Debugging
- React.js pure render performance anti-pattern

- Improving React and Redux Performance with Reselect
- Encapsulating the Redux State Tree
- React/Redux Links: React/Redux Performance

讨论

- Stack Overflow: Can a React Redux app scale as well as Backbone?

库

- Redux Addons Catalog: DevTools - Component Update Monitoring

怎样使 `mapStateToProps` 执行更快?

尽管 React Redux 已经优化并尽量减少对 `mapStateToProps` 的调用次数，加快 `mapStateToProps` 执行并减少其执行次数仍然是非常有价值的。普遍的推荐方式是利用 Reselect 创建可记忆（memoized）的 “selector” 方法。这样，selector 就能被组合在一起，并且同一管道（pipeline）后面的 selector 只有当输入变化时才会执行。意味着你可以像筛选器或过滤器那样创建 selector，并确保任务的执行时机。

补充资料

文档

- Recipes: Computed Derived Data

文章

- Improving React and Redux Performance with Reselect

讨论

- #815: Working with Data Structures
- Reselect #47: Memoizing Hierarchical Selectors

为何不在被连接的组件中使用 `this.props.dispatch`？

`connect()` 方法有两个主要的参数，而且都是可选的。第一个参数 `mapStateToProps` 是个函数，让你在数据变化时从 store 获取数据，并作为 props 传到组件中。第二个参数 `mapDispatchToProps` 依然是函数，让你可以使用 store 的 `dispatch` 方法，通常都是创建 action 创建函数并预先绑定，那么在调用时就能直接分发 action。

如果在执行 `connect()` 时没有指定 `mapDispatchToProps` 方法，React Redux 默认将 `dispatch` 作为 prop 传入。所以当你指定方法时， `dispatch` 将 不 会自动注入。如果你还想让其作为 prop，需要在 `mapDispatchToProps` 实现的返回值中明确指出。

补充资料

文档

- React Redux API: connect()

讨论

- React Redux #89: can i wrap multi actionCreators into one props with name?
- React Redux #145: consider always passing down dispatch regardless of what mapDispatchToProps does
- React Redux #255: this.props.dispatch is undefined if using mapDispatchToProps
- Stack Overflow: How to get simple dispatch from this.props using connect w/ Redux?

应该只连接到顶层组件吗，或者可以在组件树中连接到不同组件吗?

早期的 Redux 文档中建议只在组件树顶层附近连接若干组件。然而，时间和经验都表明，这需要让这些组件非常了解它们子孙组件的数据需求，还导致它们会向下传递一些令人困惑的 props。

目前的最佳实践是将组件按照 “展现层（presentational）” 或者 “容器（container）” 分类，并在合理的地方抽象出一个连接的容器组件：

> Redux 示例中强调的 “在顶层保持一个容器组件” 是错误的。不要把这个当做准则。让你的展现层组件保持独立。然后创建容器组件并在合适时进行连接。当你感觉到你是在父组件里通过复制代码为某些子组件提供数据时，就是时候抽取出一个容器了。只要你认为父组件过多了解子组件的数据或者 action，就可以抽取容器。

总之，试着在数据流和组件职责间找到平衡。

补充资料

文档

- Basics: Usage with React

文章

- Presentational and Container Components
- High-Performance Redux
- React/Redux Links: Architecture - Redux Architecture
- React/Redux Links: Performance - Redux Performance

讨论

- Twitter: emphasizing “one container” was a mistake
- #419: Recommended usage of connect
- #756: container vs component?
- #1176: Redux+React with only stateless components
- Stack Overflow: can a dumb component use a Redux container?

Redux 常见问题：其它

目录

其他

有 “真实存在” 且很庞大的 Redux 项目吗?

存在，并且有很多，比如：

- Twitter's mobile site
- Wordpress's new admin page
- Firefox's new debugger
- Mozilla's experimental browser testbed
- The HyperTerm terminal application

很多，真的有很多！

补充资料

文档

- Introduction: Examples

讨论

- Reddit: Large open source react/redux projects?
- HN: Is there any huge web application built using Redux?

如何在 Redux 中实现鉴权?

在任何真正的应用中，鉴权都必不可少。当考虑鉴权时须谨记：不管你怎样组织应用，都并不会改变什么，你应当像实现其它功能一样实现鉴权。这实际上很简单：

1. 为 `LOGIN_SUCCESS`、`LOGIN_FAILURE` 等定义 action 常量。
2. 创建接受凭证的 action 创建函数，凭证是指示身份验证成功与否的标志、一个令牌、或者作为负载的错误信息。
3. 使用 Redux Thunk middleware 或者其它适合于触发网络请求（请求 API，如果是合法鉴权则返回令牌）的 middleware 创建一个异步的 action 创建函数。之后在本地存储中保存令牌或者给用户一个非法提示。可以通过执行上一步的 action 创建函数达到此效果。
4. 为每个可能出现的鉴权场景（`LOGIN_SUCCESS`、`LOGIN_FAILURE` 等）编写独立的 reducer。

补充资料

文章

- Authentication with JWT by Auth0
- Tips to Handle Authentication in Redux

例子

- react-redux-jwt-auth-example

讨论

- redux-auth

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.map((todo, index) => {
        if (index === action.index) {
          // 修改之前复制数组
          return Object.assign({}, todo, {
            completed: true
          })
        }
        return todo
      })
    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.map((todo, index) => {
  if (index === action.index) {
    return Object.assign({}, todo, {
      completed: true
    })
  }
  return todo
})

// 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 对象操作符 所述的使用更多简洁的语法：

```
// 修改前：
return state.map((todo, index) => {
  if (index === action.index) {
    return Object.assign({}, todo, {
      completed: true
    })
  }
  return todo
})
```

```
// 修改后:
return state.map((todo, index) => {
  if (index === action.index) {
    return { ...todo, completed: true }
  }
  return todo
})
```

注意还在实验阶段的特性会经常改变。

同时要注意那些需要复制的深层嵌套的 state 对象。而 `_.extend` 和 `Object.assign` 只能提供浅层的 state 复制。在 更新嵌套的对象 章节中会教会你如何处理嵌套的 state 对象。

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 React, { 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。

确保 mapStateToProps 是正确的

你可能正确地 diaptching 一个 action 并且将它提到了 reducer 中，但是对应的 state 却没有通过 props 正确地传输。

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，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) => Dispatc
```

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
```

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, [preloadedState], [enhancer])
- 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, [preloadedState], enhancer)`

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. [`preloadedState`] *(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。

3. `enhancer` *(Function)*: 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。

返回值

(*`Store`*): 保存了应用所有 state 的对象。改变 state 的惟一方法是 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`。如果可以的话，也可以使用 对象拓展操作符（object spread spread operator 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。

- 要使用多个 store 增强器的时候，你可能需要使用 compose

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）”。这就相当于 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 的原因是要确保 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。

你可以在变化监听器里面进行 `dispatch()`，但你需要注意下面的事项：

1. 监听器调用 `dispatch()` 仅仅应当发生在响应用户的 actions 或者特殊的条件限制下（比如： 在 store 有一个特殊的字段时 dispatch action）。虽然没有任何条件去调用 `dispatch()` 在技术上是可行的，但是随着每次 `dispatch()` 改变 store 可能会导致陷入无穷的循环。
2. 订阅器（subscriptions） 在每次 `dispatch()` 调用之前都会保存一份快照。当你在正在调用监听器（listener）的时候订阅(subscribe)或者去掉订阅（unsubscribe），对当前的 `dispatch()` 不会有任何影响。但是对于下一次的 `dispatch()`，无论嵌套与否，都会使用订阅列表里最近的一次快照。
3. 订阅器不应该注意到所有 state 的变化，在订阅器被调用之前，往往由于嵌套的 `dispatch()` 导致 state 发生多次的改变。保证所有的监听器都注册在 `dispatch()` 启动之前，这样，在调用监听器的时候就会传入监听器所存在时间里最新的一次 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', previousVal
  }
}

let unsubscribe = store.subscribe(handleChange)
unsubscribe()
```

`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 redcuer 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。即使你通过 `Redux.createStore(combineReducers(...), initialState)` 指定初始 state，`combineReducers` 也会尝试通过传递 `undefined` 的 `state` 来检测你的 reducer 是否符合规则。因此，即使你在代码中不打算实际接收值为 `undefined` 的 `state`，也必须保证你的 reducer 在接收到 `undefined` 时能够正常工作。

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 的签名是 `createStore => createStore`，但是最简单的使用方法就是直接作为最后一个 `enhancer` 参数传递给 `createStore()` 函数。

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 store = createStore(
  todos,
  [ 'Use Redux' ],
  applyMiddleware(logger)
)

store.dispatch({
  type: 'ADD_TODO',
  text: 'Understand the middleware'
})
```

```
// (将打印如下信息:)
// will dispatch: { type: 'ADD_TODO', text: 'Understand the middlewar
// 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'

let reducer = combineReducers(reducers)
// applyMiddleware 为 createStore 注入了 middleware:
let store = createStore(reducer, applyMiddleware(thunk))

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, err
    )
  }
}

// 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 store = createStore(
  reducer,
  preloadedState,
  applyMiddleware(...middleware)
)
```

  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 enhancer 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。

- 想要使用多个 store enhancer，可以使用 `compose()` 方法。

`bindActionCreators(actionCreators, 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` 函数让你直接调用它 。

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作为 actionCreators，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。预计每个函数都接收一个参数。它的返回值将作为一个参数提供给它左边的函数，以此类推。例外是最右边的参数可以接受多个参数，因为它将为由此产生的函数提供签名。（译者注：`compose(funcA, funcB, funcC)` 形象为 `compose(funcA(funcB(funcC())))`）

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 } fro
import thunk from 'redux-thunk'
import DevTools from './containers/DevTools'
import reducer from '../reducers/index'

const store = createStore(
  reducer,
  compose(
    applyMiddleware(thunk),
    DevTools.instrument()
  )
```

```
)
```

小贴士

- `compose` 做的只是让你在写深度嵌套的函数时，避免了代码的向右偏移（译者注：可以参考上述的译者注）。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

从 React Native 0.18 发布之后，4.x 版本的 React Redux 能搭配 React Native 一起开发。如果你在使用 4.x 版本的 React Redux 和 React Native 一起开发遇到问题时，请先运行 `npm ls react` 确保你的 `node_modules` 中没有 React 的复制品。我们建议你使用 `npm@3.x` 来更好地规避这类问题。

如果你使用的是旧版本的 React Native 遇到这个问题，你可能需要继续使用

React Redux 3.x 和对应文档去解决。

文档

- Redux：搭配 React
- API
  - `<Provider store>`
  - `connect([mapStateToProps], [mapDispatchToProps], [mergeProps], [options])`
- 排错

它是如何工作的?

我们在 readthesource 中的一段有深入讨论到。
尽情享用吧!

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 `connet()` 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

Vanilla React

```
ReactDOM.render(
  <Provider store={store}>
    <MyRootComponent />
  </Provider>,
  rootEl
)
```

React Router

```
ReactDOM.render(
  <Provider store={store}>
    <Router history={history}>
```

```
        <Route path="/" component={App}>
          <Route path="foo" component={Foo}/>
          <Route path="bar" component={Bar}/>
        </Route>
      </Router>
    </Provider>,
    document.getElementById('root')
  )
```

`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（例如，当 props 接收到来自父组件一个小小的改动，那么你所使用的 ownProps 参数，mapStateToProps 都会被重新计算）。

  > 注意：在高级章节中，你需要更好地去控制渲染的性能，所用到的 `mapStateToProps()` 会返回一个函数。但在这个例子中，这个函数将被 `mapStateToProps()` 在独有的组件实例中调用。这样就允许你在每一个实例中去记录。你可以参考 #279 去测试和了解其中的详细内容。但在绝大多数的应用中不会用到。

- [`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。

  > 注意：在高级章节中，你需要更好地去控制渲染的性能，所用到的 `mapStateToProps()` 会返回一个函数。但在这个例子中，**这个**函数将被 `mapStateToProps()` 在独有的组件实例中调用。这样就允许你在每一个实例中去记录。你可以参考 #279 去测试和了解其中的详细内容。但在绝大多数的应用中不会用到。

- [`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` 。

> 注意：如果定义一个包含强制性参数函数（这个函数的长度为 1）时， `ownProps` **不会传到** `mapStateToProps` 和 `mapDispatchToProps` 中。举个例子，如下这样定义一个函数时将不会接收到 `ownProps` 作为第二个参数。
>
> ```
> function mapStateToProps(state) {
>   console.log(state); // state
>   console.log(arguments[1]); // undefined
> }
> ```
>
> ```
> const mapStateToProps = (state, ownProps = {}) => {
>   console.log(state); // state
>   console.log(ownProps); // undefined
> }
> ```
>
> 当函数没有强制性的参数或两个参数时**将接收到** `ownProps` 。
>
> ```
> const mapStateToProps = (state, ownProps) => {
>   console.log(state); // state
>   console.log(ownProps); // ownProps
> }
> ```
>
> ```
> function mapStateToProps() {
>   console.log(arguments[0]); // state
>   console.log(arguments[1]); // ownProps
> }
> ```

```
const mapStateToProps = (...args) => {
  console.log(args[0]); // state
  console.log(args[1]); // ownProps
}
```

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)
```

注入全部没有订阅 store 的 action creators (`addTodo`, `completeTodo`, ...)

```
import * as actionCreators from './actionCreators'

export default connect(null, actionCreators)(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

```
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 (`addTodo` , `completeTodo` , ...) 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)(TodoApp)
```

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)(TodoApp)
```

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, dispatc
  }
}

export default connect(mapStateToProps, mapDispatchToProps)(TodoApp)
```

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,
  }
```

```
}

export default connect(mapStateToProps, mapDispatchToProps)(TodoApp)
```

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, cou
}

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)(T
```

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。