20 | 从 Redux 中间件实现原理切入，理解“面向切面编程”

2020/12/16 修言

本讲我们将结合 Redux 应用实例与 applyMiddleware 源码，对 Redux 中间件的实现原理进行分析。在此基础上，我会帮助你对“面向切面”这一经典的编程思想建立初步的认识。

认识 Redux 中间件

在分析中间件实现原理之前，我们先来认识一下中间件的用法。

中间件的引入

在第 05 讲介绍 createStore 函数时，已经简单地提过中间件——**中间件相关的信息将作为 createStore 函数的一个 function 类型的入参被传入**。这里我们简单复习一下 createStore 的调用规则，示例代码如下：

复制代码

```
// 引入 redux
import { createStore, applyMiddleware } from 'redux'
......
// 创建 store
const store = createStore(
    reducer,
    initial_state,
    applyMiddleware(middleware1, middleware2, ...)
);
```

可以看到，redux 对外暴露了 applyMiddleware 这个方法。applyMiddleware 接受任意个中间件作为入参，而它的返回值将会作为参数传入 createStore，这就是中间件的引入过程。

中间件的工作模式

中间件的引入，会为 Redux 工作流带来什么样的改变呢？这里我们以 redux-thunk 为例，从经典的“异步 Action”场景切入，一起看看中间件是如何帮我们解决问题的。

redux-thunk——经典的异步 Action 解决方案

在针对 Redux 源码主流程的分析中，我们不难看出这样一个规律——**Redux 源码中只有同步操作**，也就是说当我们 dispatch action 时，state 会被立即更新。

那如果想要在 Redux 中引入异步数据流，该怎么办呢？Redux 官方给出的建议是使用中间件来增强 createStore。支持异步数据流的 Redux 中间件有很多，其中最适合用来快速上手的应该就是 redux-thunk了。

redux-thunk 的引入和普通中间件无异，可以参考以下示例：

```
// 引入 redux-thunk
import thunkMiddleware from 'redux-thunk'
import reducer from './reducers'
// 将中间件用 applyMiddleware 包装后传入
const store = createStore(reducer, applyMiddleware(thunkMiddleware))
```

这里帮大家复习一个小小的知识点，在第 18 讲我们分析 createStore 整体源码时，曾经在 createStore 逻辑的开头见过这样一段代码：

```
// 这里处理的是没有设定初始状态的情况，也就是第一个参数和第二个参数都传 function 的情况
if (typeof preloadedState === 'function' && typeof enhancer === 'undefined') {
  // 此时第二个参数会被认为是 enhancer（中间件）
  enhancer = preloadedState;
  preloadedState = undefined;
}
```

这段代码告诉我们，在只传入两个参数的情况下，createStore 会去检查第二个参数是否是 function 类型，若是，则认为第二个参数是“enhancer”。这里的“enhancer”是“增强器”的意思，而 applyMiddleware 包装过的中间件，正是“增强器”的一种。这也就解释了为什么上面 redux-thunk 的调用示例中，applyMiddleware 调用明明是作为 createStore 的第二个参数被传入的，却仍然能够被识别为中间件信息。

redux-thunk 带来的改变非常好理解，**它允许我们以函数的形式派发一个 action**，像这样（解析在注释里）：

```
// axios 是一个用于发起异步请求的库
import axios from 'axios'
// 引入 createStore 和 applyMiddleware
import { createStore, applyMiddleware } from 'redux';
// 引入 redux-thunk
import thunk from 'redux-thunk';
// 引入 reducer
import reducer from './reducers';
// 创建一个有 thunk 中间件加持的 store 对象
const store = createStore(
  reducer,
```

```
  applyMiddleware(thunk)
);
// 用于发起付款请求，并处理请求结果。由于涉及资金，我们希望感知请求的发送和响应的返回
// 入参是付款相关的信息（包括用户账密、金额等）
// 注意 payMoney 的返回值仍然是一个函数
const payMoney = (payInfo) => (dispatch) => {
  // 付款前发出准备信号
  dispatch({ type: 'payStart' })
  fetch().then(res => { dispatch()})
  return axios.post('/api/payMoney', {
    payInfo
  })
  .then(function (response) {
    console.log(response);
    // 付款成功信号
    dispatch({ type: 'paySuccess' })
  })
  .catch(function (error) {
    console.log(error);
    // 付款失败信号
    dispatch({ type: 'payError' })
  });
}
// 支付信息，入参
const payInfo = {
  userName: xxx,
  password: xxx,
  count: xxx,
  ......
}
// dispatch 一个 action，注意这个 action 是一个函数
store.dispatch(payMoney(payInfo));
```

这里我尝试用 redux-thunk 模拟了一个付款请求的发起 → 响应过程。

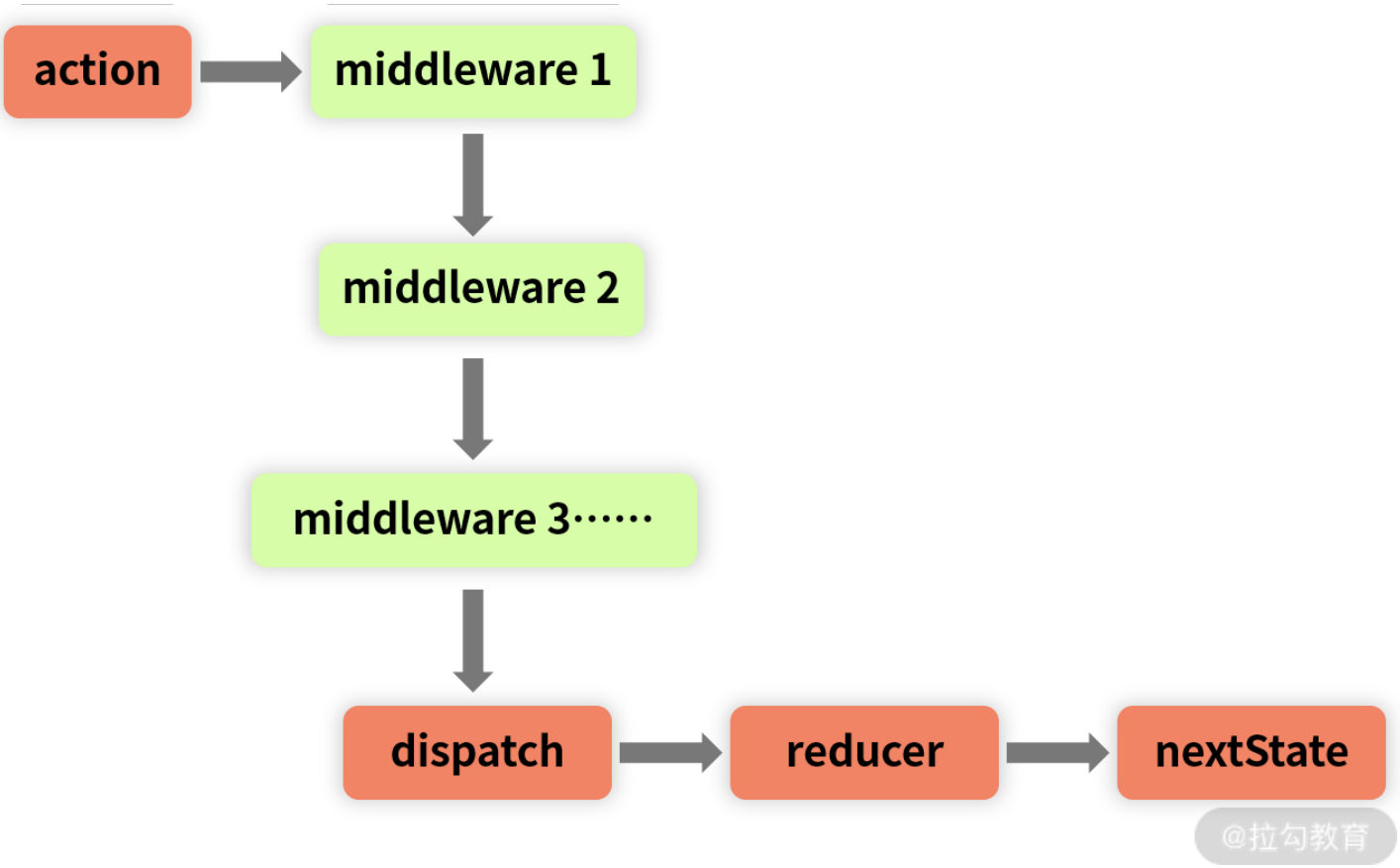
这个过程单从表面上看，和普通 Redux 调用最大的不同就是 **dispatch 的入参从 action 对象变成了一个函数**。这就不由得让人对 thunk 中间件加持下的 Redux 工作流心生好奇——**action 入参必须是一个对象**，这一点我们在第 19 讲分析 dispatch 源码时，可是亲眼见过 action 相关的数据格式强校验逻辑的！**而 thunk 中间件似乎巧妙地“绕开”了这层校验，这背后到底藏着什么玄机呢**？

要想搞清楚这个问题，你除了需要理解 thunk 的执行逻辑，更重要的是要知道 Redux 中间件是如何工作的。

Redux 中间件是如何与 Redux 主流程相结合的?

Redux 中间件将会在 action 被分发之后、到达 reducer 之前执行，对应到工作流中，它的执行时机如下图所示：

若有多个中间件，那么 Redux 会结合它们被“安装”的先后顺序，依序调用这些中间件，这个过程如下图所示：



中间件的执行时机，允许它在状态真正发生变化之前，结合 action 的信息做一些它想做的事情。

那么中间件又是如何“绕过” dispatch 的校验逻辑的呢？其实，“绕过”dispatch 只是咱们主观上的一个使用感受。**dispatch 并非被“绕过”了，而是被“改写”了，改写它的不是别人，正是 applyMiddleware**。关于这点，我会在本文后续的源码分析环节为你深入讲解。

读到这里，对于 Redux 中间件的工作模式，你需要牢牢把握以下两点：

1. 中间件的执行时机，即 action 被分发之后、reducer 触发之前；
2. 中间件的执行前提，即 **applyMiddleware 将会对 dispatch 函数进行改写，使得 dispatch 在触发 reducer 之前，会首先执行对 Redux 中间件的链式调用。**

结合这两点，再来看 redux-thunk 的源码，一切就会豁然开朗了。

thunk 中间件到底做了什么？

redux-thunk 的源码其实非常简洁，我第一次接触时还是在 2016 年，这么多年过去了，很多事情都变了，唯一不变的是 redux-thunk，它仍然那么好懂。既然这么好懂，我们不如直接来读读看，请看 redux-thunk 的源码：

复制代码

```
// createThunkMiddleware 用于创建 thunk
function createThunkMiddleware(extraArgument) {
  // 返回值是一个 thunk，它是一个函数
  return ({ dispatch, getState }) => (next) => (action) => {
    // thunk 若感知到 action 是一个函数，就会执行 action
    if (typeof action === 'function') {
      return action(dispatch, getState, extraArgument);
    }
    // 若 action 不是一个函数，则不处理，直接放过
    return next(action);
  };
}
const thunk = createThunkMiddleware();
thunk.withExtraArgument = createThunkMiddleware;
export default thunk;
```

redux-thunk 主要做的事情，就是在拦截到 action 以后，会去检查它是否是一个函数。**若 action 是一个函数，那么 redux-thunk 就会执行它并且返回执行结果；若 action 不是一个函数，那么它就不是 redux-thunk 的处理目标，直接调用 next**，告诉 Redux “我这边的工作做完了”，工作流就可以继续往下走了。

到这里，你已经在使用层面对 Redux 中间件有了足够的认知。接下来，我们就要进入源码的世界啦。

Redux 中间件机制是如何实现的

Redux 中间件是通过调用 applyMiddleware 来引入的，因此我们先看看 applyMiddleware 的源码（解析在注释里）：

复制代码

```
// applyMiddlerware 会使用“...”运算符将入参收敛为一个数组
export default function applyMiddleware(...middlewares) {
  // 它返回的是一个接收 createStore 为入参的函数
  return createStore => (...args) => {
    // 首先调用 createStore，创建一个 store
    const store = createStore(...args)
    let dispatch = () => {
      throw new Error(
        `Dispatching while constructing your middleware is not allowed. ` +
          `Other middleware would not be applied to this dispatch.`
      )
```

```
    }

    // middlewareAPI 是中间件的入参
    const middlewareAPI = {
      getState: store.getState,
      dispatch: (...args) => dispatch(...args)
    }
    // 遍历中间件数组，调用每个中间件，并且传入 middlewareAPI 作为入参，得到目标函数数组 ch
    const chain = middlewares.map(middleware => middleware(middlewareAPI))
    // 改写原有的 dispatch：将 chain 中的函数按照顺序“组合”起来，调用最终组合出来的函数，传
    dispatch = compose(...chain)(store.dispatch)

    // 返回一个新的 store 对象，这个 store 对象的 dispatch 已经被改写过了
    return {
      ...store,
      dispatch
    }
  }
}
```

在这段源码中，我们着重需要搞清楚的是以下几个问题：

1. applyMiddleware 返回了一个什么样的函数？这个函数是如何与 createStore 配合工作的？
2. dispatch 函数是如何被改写的？
3. compose 函数是如何组合中间件的？

1. applyMiddleware 是如何与 createStore 配合工作的？

先来看看 applyMiddleware 的返回值。在源码的注释中，我已经标明，它返回的是一个接收 createStore 为入参的函数。这个函数将会作为入参传递给 createStore，那么 createStore 会如何理解它呢？这里就要带你复习一下 createStore 中，enhancer 相关的逻辑了，请看下面代码：

复制代码

```
function createStore(reducer, preloadedState, enhancer) {
    // 这里处理的是没有设定初始状态的情况，也就是第一个参数和第二个参数都传 function 的情况
    if (typeof preloadedState === 'function' && typeof enhancer === 'undefined') {
        // 此时第二个参数会被认为是 enhancer（中间件）
        enhancer = preloadedState;
        preloadedState = undefined;
    }
    // 当 enhancer 不为空时，便会将原来的 createStore 作为参数传入到 enhancer 中
    if (typeof enhancer !== 'undefined') {
        return enhancer(createStore)(reducer, preloadedState);
    }
    ......
}
```

从这个代码片段中我们可以看出，一旦发现 enhancer 存在（对应到中间件场景下，enhancer 指的是 applyMiddleware 返回的函数），那么 createStore 内部就会直接 return 一个针对 enhancer 的调用。在这个调用中，第一层入参是 createStore，第二层入参是 reducer 和 preloadedState。

我们可以尝试将这个逻辑在 applyMiddleware 中对号入座一下。下面我从出入参角度简单提取了一下 applyMiddleware 的源码框架：

```
// applyMiddlerware 会使用“...”运算符将入参收敛为一个数组
export default function applyMiddleware(...middlewares) {
  // 它返回的是一个接收 createStore 为入参的函数
  return createStore => (...args) => {
    ......
  }
}
```

结合 createStore 中对 enhancer 的处理，我们可以知道，在 applyMiddleware return 出的这个函数中，createStore 这个入参对应的是 createStore 函数本身，而 args 入参则对应的是 reducer、preloadedState，这两个参数均为 createStore 函数的约定入参。

前面我们讲过，applyMiddleware 是 enhancer 的一种，而 enhancer 的意思是“增强器”，它增强的正是 createStore 的能力。因此调用 enhancer 时，传入 createStore 及其相关的入参信息是非常必要的。

2.dispatch 函数是如何被改写的?

dispatch 函数的改写，是由下面这个代码片段完成的：

```
// middlewareAPI 是中间件的入参
const middlewareAPI = {
  getState: store.getState,
  dispatch: (...args) => dispatch(...args)
}
// 遍历中间件数组，调用每个中间件，并且传入 middlewareAPI 作为入参，得到目标函数数组 chain
const chain = middlewares.map(middleware => middleware(middlewareAPI))
// 改写原有的 dispatch：将 chain 中的函数按照顺序“组合”起来，调用最终组合出来的函数，传入 d
dispatch = compose(...chain)(store.dispatch)
```

这个代码片段做了两件事：首先以 middlewareAPI 作为入参，逐个调用传入的 middleware，获取一个由“内层函数”组成的数组 chain；然后调用 compose 函数，将 chain 中的“内层函数”逐个组合起来，并调用最终组合出来的函数。

在上面这段描述中，有两个点可能对你的理解构成障碍：

1. 什么是“内层函数”？
2. compose 函数到底是怎么组合函数的？它组合出来的又是个什么东西？

关于第 2 点，我们需要到 compose 源码中去看，这里先按下不表，咱们来说说“内层函数”在这里的含义。

首先我们需要站在函数的视角，来观察一下 thunk 中间件的源码：

复制代码

```
// createThunkMiddleware 用于创建 thunk
function createThunkMiddleware(extraArgument) {
  // 返回值是一个 thunk，它是一个函数
  return ({ dispatch, getState }) => (next) => (action) => {
    // thunk 若感知到 action 是一个函数，就会执行 action
    if (typeof action === 'function') {
      return action(dispatch, getState, extraArgument);
    }
    // 若 action 不是一个函数，则不处理，直接放过
    return next(action);
  };
}
const thunk = createThunkMiddleware();
```

thunk 中间件是 createThunkMiddleware 的返回值，createThunkMiddleware 返回的是这样的一个函数：

复制代码

```
({ dispatch, getState }) => (next) => (action) => {
  // thunk 若感知到 action 是一个函数，就会执行 action
  if (typeof action === 'function') {
    return action(dispatch, getState, extraArgument);
  }
  // 若 action 不是一个函数，则不处理，直接放过
  return next(action);
};
```

该函数的返回值仍然是一个函数，显然它是一个**高阶函数**。事实上，按照约定，所有的 Redux 中间件都必须是高阶函数。在高阶函数中，我们习惯于将原函数称为“外层函数”，将 return 出来的函数称为“内层函数”。

而 apply 中遍历 middlewares 数组，逐个调用 middleware(middlewareAPI)，无非是为了获取中间件的内层函数。

以 thunk 的源码为例，不难看出，外层函数的主要作用是获取 dispatch、getState 这两个 API，而**真正的中间件逻辑是在内层函数中包裹的**。待 `middlewares.map(middleware => middleware(middlewareAPI))` 执行完毕后，内层函数会被悉数提取至 chain 数组。接下来，我们直接拿 chain 数组开刀就行了。

提取出 chain 数组之后，applyMiddleware 做的第一件事就是将数组中的中间件逻辑 compose 起来。

那么 compose 函数又是如何工作的呢?

3. compose 源码解读：函数的合成

函数合成（组合函数）并不是 Redux 的专利，而是函数式编程中一个通用的概念。因此在 Redux 源码中，compose 函数是作为一个独立文件存在的，它具备较强的工具属性。

我们还是先通过阅读源码，来弄清楚 compose 到底都做了什么。以下是 compose 的源码（解析在注释里）：

复制代码

```
// compose 会首先利用"..."运算符将入参收敛为数组格式
export default function compose(...funcs) {
  // 处理数组为空的边界情况
  if (funcs.length === 0) {
    return arg => arg
  }

  // 若只有一个函数，也就谈不上组合，直接返回
  if (funcs.length === 1) {
    return funcs[0]
  }
  // 若有多个函数，那么调用 reduce 方法来实现函数的组合
  return funcs.reduce((a, b) => (...args) => a(b(...args)))
}
```

其实整段源码中值得你细细品味的只有最后这一行代码：

复制代码

```
// 若有多个函数，那么调用 reduce 方法来实现函数的组合
return funcs.reduce((a, b) => (...args) => a(b(...args)))
```

这行代码告诉我们，函数组合是通过调用数组的 reduce 方法来实现的。

reduce 方法是 JS 数组中一个相对基础的概念，这里我们不再展开讲解，需要复习的同学请狠狠地点击这里。

reducer 方法的特点是，会对数组中的每个元素执行我们指定的函数逻辑，并将其结果汇总为单个返回值。因此对于这样的一个 compose 调用来说：

```
compose(f1, f2, f3, f4)
```

复制代码

它会把函数组合为这种形式：

```
(...args) =>  f1(f2(f3(f4(...args))))
```

复制代码

如此一来，f1、f2、f3、f4 这 4 个中间件的内层逻辑就会被组合到一个函数中去，当这个函数被调用时，f1、f2、f3、f4 将会按照顺序被依次调用。这就是“函数组合”在此处的含义。

加餐：中间件与面向切面编程

中间件这个概念并非 Redux 的专利，它在软件领域由来已久，大家所熟知的 Koa、Express 这些 Node 框架中也都不乏对中间件的应用。那么为什么中间件可以流行？为什么我们的应用需要中间件呢？这里，我就以 Redux 中间件机制为例，简单和你聊聊中间件背后的“面向切面”编程思想。

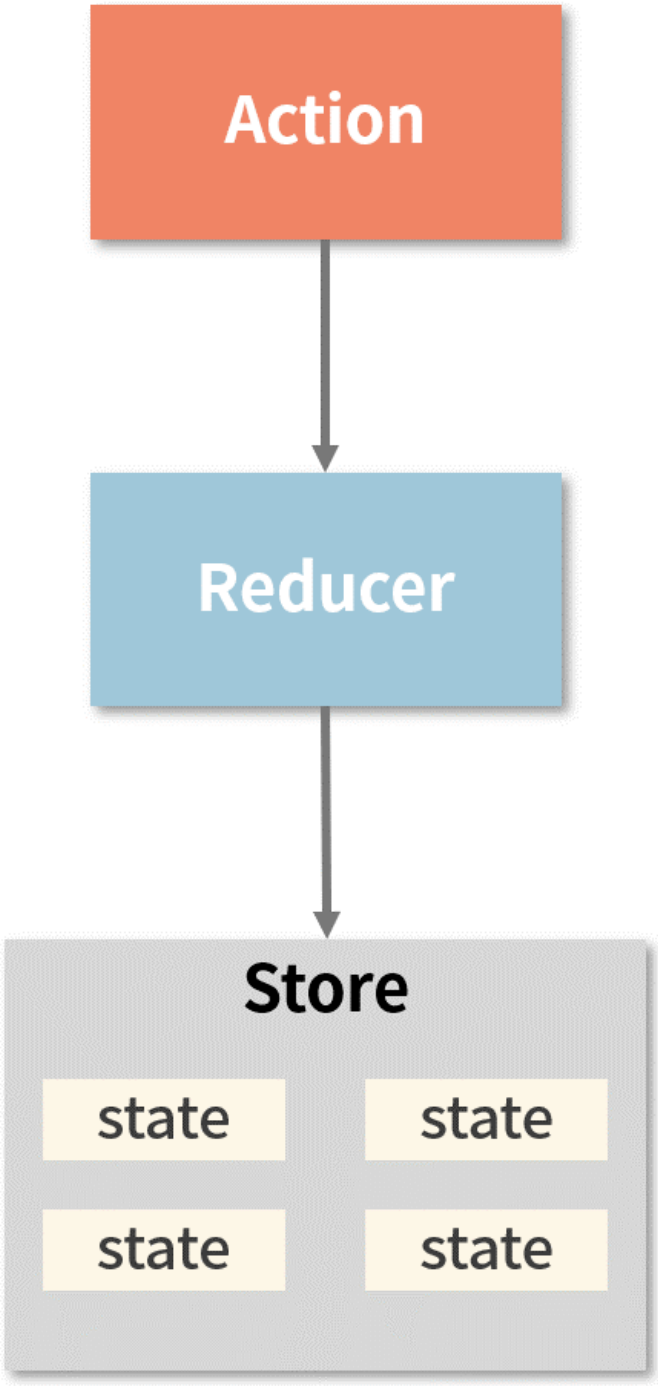
AOP（面向切面）这个概念可能很多同学都不太了解，大家相对熟悉的应该是 OOP（面向对象）。而 AOP 的存在，恰恰是为了解决 OOP 的局限性，我们可以将 AOP 看作是对 OOP 的一种补充。

在 OOP 模式下，当我们想要拓展一个类的逻辑时，最常见的思路就是继承：class A 继承 class B，class B 继承 class C……这样一层一层将逻辑向下传递。

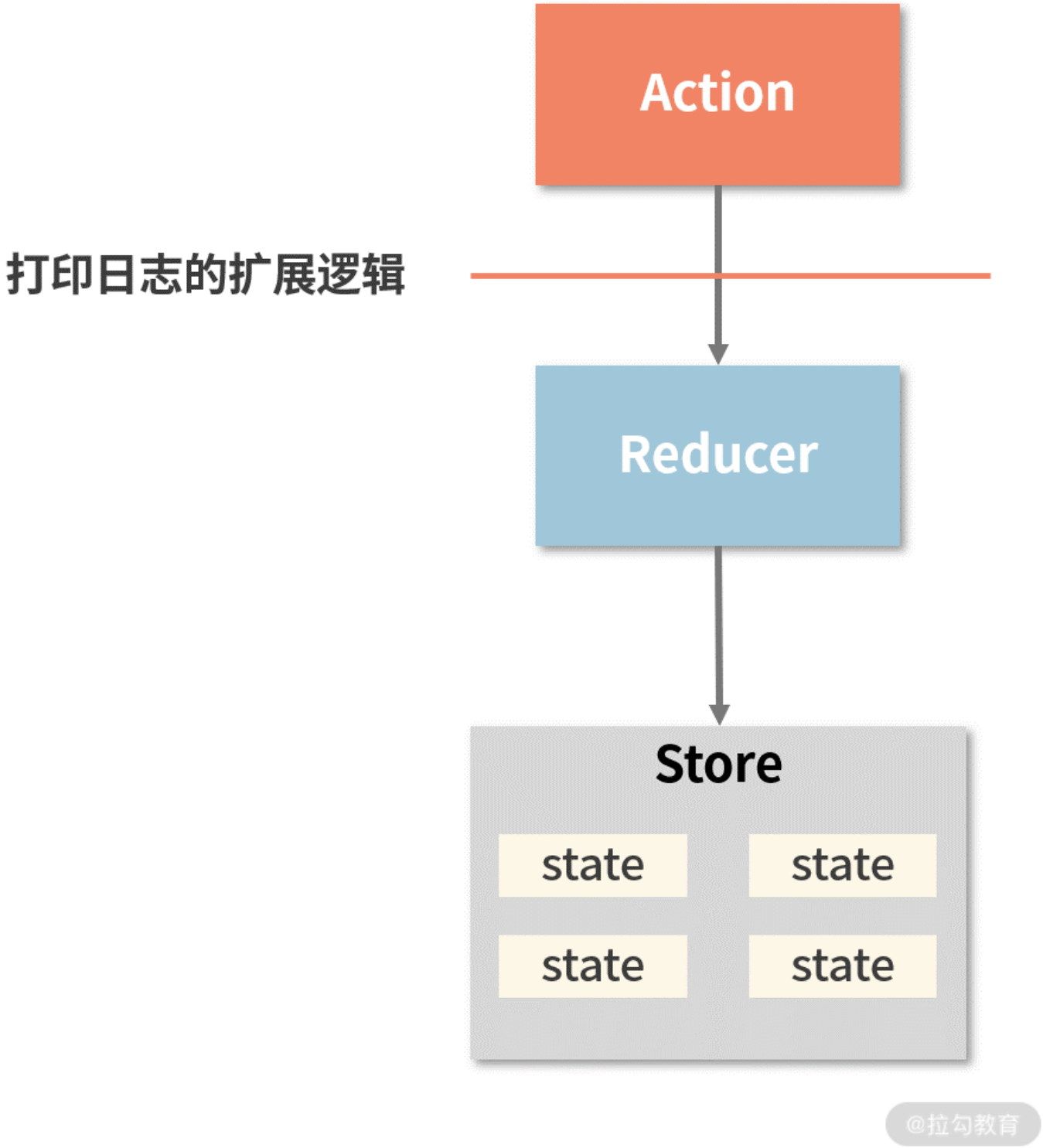
当我们想要为某几个类追加一段共同的逻辑时，可以通过修改它们共同的父类来实现，这无疑会使得公共类越来越臃肿，可我们也确实没有什么更好的办法——总不能任这些公共逻辑散落在不同的业务逻辑里吧？那将会引发更加严重的代码冗余及耦合问题。

怎么办呢？“面向切面”来救场！

既然是面向“切面”，那么首先我们要搞清楚什么是“切面”。切面是一个相对于执行流程来说的概念，以 Redux 为例，它的工作流自上而下应该是这样的，如下图所示：



此时考虑这样一个需求：要求在每个 Action 被派发之后，打出一个 console.log 记录“action 被派发了”这个动作，也就是我们常说的“日志追溯”。这个需求的**通用性很强、业务属性很弱**，因此不适合与任何的业务逻辑耦合在一起。那我们就可以以 **“切面”这种形式，把它与业务逻辑剥离开来**：扩展功能在工作流中的执行节点，可以视为一个单独“切点”；我们把扩展功能的逻辑放到这个“切点”上来，形成的就是一个可以拦截前序逻辑的“切面”，如下图所示：



“切面”与业务逻辑是分离的，因此 AOP 是一种典型的 **“非侵入式”的逻辑扩充思路**。

在日常开发中，像“日志追溯”“异步工作流处理”“性能打点”这类和业务逻辑关系不大的功能，我们都可以考虑把它们抽到“切面”中去做。

面向切面编程带来的利好是非常明显的。从 Redux 中间件机制中，不难看出，面向切面思想在很大程度上提升了我们组织逻辑的灵活度与干净度，帮助我们规避掉了逻辑冗余、逻辑耦合这类问题。通过将

“切面”与业务逻辑剥离，开发者能够专注于业务逻辑的开发，并通过“**即插即用**”的方式自由地组织自己想要的扩展功能。

总结

在这一讲，我们首先以 redux-thunk 中间件为例，从“异步工作流”场景切入，认识了 Redux 中间件的工作模式。随后，结合 applyMiddleware 源码，对 Redux 中间件的整个执行机制进行了细致深入的分析，并在文末引入了对“面向切面”这一编程思想的介绍。

行文至此，整个由 Redux 所牵出的核心知识体系也已经一览无余地呈现在你面前，相信你对 Redux 的理解又上了一个台阶。

专栏的下一讲，我将以 React 的另一个“好帮手” React-Router 为切入点，为你讲解前端路由相关的知识，不见不散。