## **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。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**

每次state更新时，如何记录其值

### **尝试 #1: 手动记录**

let action = addTodo('Use Redux')

console.log('dispatching', action)

store.dispatch(action)

console.log('next state', store.getState())

### **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'))

### **尝试 #3: Monkeypatching Dispatch**

重写我们的dispatch

let next = store.dispatch

store.dispatch = function dispatchAndLog(action) {

console.log('dispatching', action)

// 调用原来的 dispatch

let result = next(action)

console.log('next state', store.getState())

return result

}

### **问题: 崩溃报告**

按照如上想法，我们添加崩溃报告的处理

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

}

}}

### **尝试 #4: 隐藏 Monkeypatching**

现在我们不直接更改store.dispatch，而是返回一个新的dispatch，此时我们可以称我们的方法为“中间件”

function logger(store) {

let next = store.dispatch

return function dispatchAndLog(action) {

console.log('dispatching', action)

let result = next(action)

console.log('next state', store.getState())

return result

}

}

新建一个方法，叫做“应用中间件”，接收 store 和 中间件数组

function applyMiddlewareByMonkeypatching(store, middlewares) {

middlewares = middlewares.slice()

middlewares.reverse()

// 在每一个 middleware 中变换 dispatch 方法。

middlewares.forEach(middleware =>

// 调用中间件，每个中间件都返回一个新的dispatch

store.dispatch = middleware(store)

)

}

然后像这样应用多个 middleware：

applyMiddlewareByMonkeypatching(store, [ logger, crashReporter ])

### **尝试 #5: 移除 Monkeypatching**

ES6 的箭头函数可以使其 [柯里化](https://en.wikipedia.org/wiki/Currying" \t "https://www.redux.org.cn/docs/advanced/_blank) ，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 接收了一个参数 next ，其本身是 dispatch 函数，并返回一个新的 dispatch 函数，返回的函数会被作为下一个 middleware 的 next()

不明白[柯里化](https://en.wikipedia.org/wiki/Currying" \t "https://www.redux.org.cn/docs/advanced/_blank)函数，如下，这样写应该会明白

store => next => action => {...}

store => {

    return (next => {

        return (action => {

            ...

        })

    })

}

### **尝试 #6: “单纯”地使用 Middleware**

写一个 applyMiddleware 方法替换原来的 applyMiddlewareByMonkeypatching。

// 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()](https://www.redux.org.cn/docs/api/applyMiddleware.html) 的实现已经很接近了

### **使用Redux的applyMiddleware**

Redux的applyMiddleware原理就是我们所编写的applyMiddleware

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'))