Sử dụng Middleware để có thể chạy được logic bất đồng bộ

Bởi chính Redux, một *store* không hề biết bất cứ thứ gì về **logic bất đồng bộ**. Nó chỉ biết cách để dispatch một hành động đồng bộ, cập nhật trạng thái bằng cách gọi tới hàm root của reducer, và thông báo cho UI là có cái gì đó đã thay đổi. Bất kì sự bất đồng bộ nào đều phải xảy ra bên ngoài *store*

Nhưng, sẽ ra sao nếu như chúng ta muốn có các **logic bất đồng bộ (async logic)** tương tác với *store* bởi dispatch hoặc là kiểm tra trạng thái hiện tại của *store*? Đó là cách màn **Redux middleware** hướng tới. Chúng mở rộng *store* và cho phép bạn:

- Thực thi các logic mở rộng khi bất kì action nào được dispatch (như là ghi vào *action* / *state*)
- Tạm dừng, điều chỉnh, delay, thay thế hoặc halt một hành động được dispatch
- Viết những đoạn code mở rộng có thể truy cập vào dispatch và getState
- Dạy *dispath* các để chấp nhận những giá trị khác bên cạnh các *plain action objects* như là một hàm, một promise bằng cách ngăn chúng lại và thực hiện dispatch các *action object* thay thế

Lý do phổ biến nhất để sử dụng middleware đó là cho phép tạo ra nhiều kiểu *async logic* để có thể tương tác được với *store*. Điều này cho phép bạn có thể những đoạn code có khả năng dispatch các *action* và kiểm tra *state* của store, trong khi đó vẫn giữ cho logic ngăn cách với UI

Có nhiều loại *async middleware* dành cho Redux, và mỗi lại cho phép bạn viết logic theo các cú pháp khác nhau. Các *async middleware* phổ biến nhất là

- **redux-thunk**, thứ cho phép bạn viết các *plain function* mà hướng tới chứ các *async logic*
- **redux-saga**, thứ sử dụng để sinh ra các hàm và trả về các miêu tả về hành vi nhờ vậy chúng có thể thực thi bởi *middleware*
- **redux-observable**, sử dụng thuw viện *RxJS observable* đẻ tạo ra chuỗi các hàm xử lý các *action*

NOTE: Với mỗi thư viện được sử dụng trong các usecase khác nhau và có sự đánh dổi

Nêu bạn cần phải viết các logic kéo dữ liệu, chúng tôi khuyên khích bạn sử dụng *Redux Thunk middleware* như là giải pháp tiêu chuẩn, nó vừa đủ cho hầu hết các usecase thông tường. Thêm vào đó sử dụng cú pháp **async/await** trong *thunk* làm cho chúng dễ đọc hơn

Redux Toolkit *configureStore* mặc định tự động thiết lập thunk middleware, nhờ vậy bạn có thể bắt đầu ngay vào việc viết các *thunk* như là một phần của app

Định nghĩa các Async Logic trong các Slice

Redux Tookit hiện tại không cung cấp bất kì API đặc biệt nào hoặc là cú pháp để có thể viết các hàm *thunk*. Cụ thể, chúng không được định nghĩa như là một phần của *createSlice()*. Bạn phải viết chúng riêng biệt với logic của các *reducer*, chính xác hơn là giống như các viết một *plain Redux code*

Thunk thông thường dispatch các *plain action*, ví dụ như là **dispatch(dataLoaded(response.data))**

Nhiều Redux app có cấu trúc code của họ sử dungj giải pháp *folder-by-type*. Trong cáu trúc đó, thunk action creator thường được định nghĩa trong các file *action*, song song với các *plain action creator*

Bởi vì chúng ta không thể phân tác các file *action*, nó làm cho .... viết các thunk hướng tới các file *slice*

Một file slice thông thường chứa thunks sẽ trông như sau

```
const usersSlice = createSlice({
        name : 'users',
        initialState : {
                loading : 'idle',
                users : [],
        },
```

```
        reducers : {
                usersLoading(state, action) {
                        // Sử dụng giải pháp 'State Machine' dành cho các
trạng thái loading thay vì sử dụng các biến boolean
                        if (state.loading === 'idle') {
                                state.loading = 'pending'
                        }
                },
                usersReceived(state, action) {
                        if (state.loading === 'pending') {
                                state.loading = 'idle'
                                state.users = action.payload
                        }
                }
        },
})

// Destructure and export the plain action creators
export const { usersLoading, usersReceived } = usersSlice.actions

// Định nghĩa một thung để dispatch các *action creator*
const fetchUsers = () => async (dispatch) => {
        dispatch(usersLoading())
        const response = await usersAPI.fetchAll();
        dispatch(usersReceived(response.data));
}
```

Mẫu thiết kế kéo dữ liệu Redux

Kéo dữ liệu logic dành cho Redux lường theo mẫu có thể dự đoán được:

- Một hành động *bắt đầu (start)* được dispatch trước khi một request biểu thị rằng request đang trong quá trình xử lý. Điều này có thể sử dụng để track trạng thái loading, để cho phép bỏ qua các request trùng lặp hoặc hiển thị loading indicator ở UI
- Một request bất đồng bộ được tạo
- Tùy thuộc vào kết quả của request, *async logic* dispatch cả hành động *success* chứa dữ liệu của quả request, hoặc là một hành động *failure* chứa thông tin về lỗi. *Reducer logic* xóa trạng thái loading ở cả 2 trường hợp trên, và xử lý kết của

của dữ liệu trong trường hợp thành công, hoặc lưu lại các giá trị lỗi để có thể hiển thị

Những bước trên thì không yêu cầu nhưng được khuyến khích trong *Redux tutorial* như là một mẫu gợi ý

Các triển khai thông thường như sau:

```
const getRepoDetaisStarted = () => ({
        type : "repoDetails/fetchStarted"
})

const getRepoDetailsSuccess = (repoDetails) => ({
        type : "repoDetails/fetchSucceeded",
        payload : repoDetails
})
const getRepoDetailsFailed = (error) => ({
        type : "repoDetails/fetchFailed",
        error
})
const fetchIssuesCount = (org, repo) => async dispatch => {
        dispatch(getRepoDetailsStarted())
        try {
                const repoDetails = await getRepodetails(org, repo)
                dispatch(getRepoDetailsSuccess(repoDetails))
        } catch (err) {
                dispatch(getRepoDetailsFailed(err.toString()));
        }
}
```

Tuy nhiên, viết code như trên thì là thảm họa. Mỗi loại phân biệt của request lặp lại các triển khia tương tự nhau:

- Type action duy nhất cần được định nghĩa cho 3 trường hợp khác nhau
- Mỗi type action thường tương ứng với một action creator
- Một thunk phải được việt để dispath đúng hành động trong chuỗi

createAsyncThunk trùy tượng mẫu này bằng các sinh sẵn các action type và các action creator và sinh sẵn luông thunk để có thể dispatch các hành động này

Async Request with createAsyncThunk

Như là một dev, bạn chắc là hầu hết các lô âu với logic cần được tạo một API request, action type naame hiển thị trong Redux action history log, và các mà reducer của bạn nên sử lý các data được kéo về. Các chi tiết lạp đi lạp lại của việc xác định nhiều loại hành động và gửi các hành động thoe đúng trình tự không phải là điều quan trọng

createAsyncThunk tối giản hóa quá trình này, bạn chỉ cần cung cấp một *string* dành do tiền tố của action type và một payload creator callback để chạy logic async thực sự và sau đó return về một promise như là kết quả. Trong khi trả về, createAsyncThunk sẽ cho bạn một thunk mà kiểm soát việc dispatch các hành động đúng dựa trên promise mà bạn trả về, và action type mà bạn muốn handle trong reducer của bạn

```
import { createAsyncThunk, createSlice } from '@reduxjs/toolkit';
import { userAPI } from './userAPI';

// Đầu tiên là tạo thunk
const fetchUserById = createAsyncThunk(
        // string dành cho tiền tố của actionType
        'users/fetchByIdStatus',

        // payload creator callback
        async (userId, thunkAPI) => {
                const response = await userAPI.fetchById(userId);
                return response.data
        }
)

// Sau đó mình sẽ thực hiện handle hành động trong reducer
const usersSlice = createSlice({
        name : 'users',
        initialState : {
                entities : [],
                loading : 'idle',
        },
        reducers : {
                // các logic reducer tiêu chuẩn, được tự động khởi tạo các
```

```
action type cho mỗi reducer
        },
        extraReducers : (builder) => {
                // Thêm reducer dành cho các action type mở rộng ở đây,
handle trạng thái loading nếu cần thiết
                builder.addCase(fetchUserById.fulfilled, (state, action)
=> {
                        // Thêm user vào mảng state
                        state.entities.push(action.payload)
                })
        }
,
})

// Sau đó, dispath thunk khi cần bên trong app
dispatch(fetchUserById(123));
```

Thunk action creator chấp nhật một tham số đơn, luôn được pass như là tham số đầu tiên tới *payload creator callback*

Payload creator sẽ nhận một thunkAPI chứa các tham số thông thường được pass và Redux thunk function

```
interface ThunkAPI {
        dispatch : Function
        getState : Function
        extra? : any
        requestId : string
        signal : AbortSignal
}
```