**Redux中间件背景：**

Redux本身提供的Api与功能并不多，但是它提供了一个中间件（插件）机制，可以使用第三方提供的中间件或自己编写一个中间件来对Redux的功能进行增强，比如可以使用redux-logger这个中间件来记录action以及每次action前后的state、使用redux-undo来取消/重做action、使用redux-persist-store来对store进行持久， 使用redux-thunk来进行异步化action，等等。

在redux中实现中间件的本质的目的是提供第三方插件的模式，自定义拦截action->reducer的过程。变为action->middlewares->reducer. 这种机制可以让我们改变数据流，实现异步action, action过滤，日志输出，异常报告等功能。

Redux提供中间件，中间件的实质是一个方法，该方法接受一个对象为参数 ({dispatch, getState}). 该方法将返回一些系列的方法。最深层的方法表明中间件的实际逻辑，最深层的方法一般包含两种情况。第一种是中间件自行处理该action,不移交给下一个中间。这最表明中间件的用途。第二种是会包含语句next(action)，表示在某些情况下将action传递给下一个中间件，此中间件自身不做任何特殊处理。最深层的方法，将接受的参数为action。该action就是未来我们store.dispatch(action)中的action.

例如：  
reactTrunk中间件实现源码：

return ({dispatch, getState}) => next => action => {

if (typeof action === 'function') {

return action (dispatch, getState, extraArgument); //中间件自身自行处理的情况，也表明该中间件的主要用处是处理store.dispatch(action), action为function的情况。在action ==== function的情况下不调用next(action)代表此时reactTrunk中间件将自行处理，不再移交给下个中间件。action === function, 所以未来我们可以在调用store.dispatch方法时，如果想应用这个中间件，那么传入action参数必定为一个方法，并且基于回调函数的原理，回掉函数的参数由调用时决定，所以在action参数传入的方法第一个参数为dispatch, 第二个参数为getState.

}

return next(action); //next(action)传递给下个中间件

};

例如：

自定义的multiAction中间件实现源码：

export const multiActions = ({dispatch, getState}) => next => action => {

if (Array.isArray(action)) {

action.forEach(a => next(a)) //中间件自身自行处理的情况，也表明该中间件的主要用处是处理store.dispatch(action), action为数组的情况。在action是数组的情况下不调用next(action),代表此时multiAction中间件将自行处理。不再移交给下个中间件。它的处理方式是轮询数组，对每个数组对象都移交给下一个中间件。

}

else{

next(action)

}

}

P.S Redux的最初设计，action只能为普通的JS对象，然后通过reducer进行处理。从上述两个例子可以看出，在使用中间件以后，通过在中间件的最深层函数中的逻辑，可以达到根据不同的情况，中间件可以自身处理action不传递给下个中间件，或者只是普通移交给下一个中间件，也可以先执行额外的逻辑再移交给下个中间件。并且我们可以在store.dispatch(action)中，为action使用更为复杂的数据结构（因为中间件会进行处理）。

**使用applyMiddleware挂载中间件：**

Redux提供了一个叫applyMiddleware的方法，用于挂载一个或多个中间件。

在createStore的applyMiddleware中传入该中间件方法作为参数，进行挂载。如果有多个中间件方法，则传入applyMiddleware方法的顺序便为应用中间件们的顺序。

例如我们挂载一个第三方中间件react-trunk：

import {createStore, applyMiddleware} from “redux”;

const store = createStore (counter, applyMiddleware(ReduxThunk.default));

原来的store.dispatch方法只能接收一个普通的action对象作为参数，当我们加入了ReduxThunk这个中间件之后，store.dispatch还可以接收一个方法作为参数，这个方法会接收到两个参数，第一个是dispatch，等同于store.dispatch，第二个是getState，等同于store.getState，也就是说，现在可以这样来触发INCREASE：

store.dispatch((dispatch, getState) => dispatch({type: 'INCREASE'}));

依托于ReduxThunk实现一个异步action操作：

<div>

<p>Count: <span id="value">0</span></p>

<button id="btn\_increase">+ 1</button>

<button id="btn\_async\_increase">+ 1 async</button>

<button id="btn\_decrease">- 1</button>

</div>

document. getElementById('btn\_async\_increase').addEventListener('click', function () {

store.dispatch((dispatch, getState) => { //store.dispatch接受一个方法作为参数，该方法接受dispatch, getState参数。

setTimeout(() => {

dispatch ({type: 'INCREASE'});

}, 1000);

});

});

**解析中间件ReduxThunk源码：**

function createThunkMiddleware(extraArgument) {

return ({dispatch, getState}) => next => action => {//createThunkMiddleware方法返回一个中间件，也就是返回一个方法。该中间件接受 ({dispatch, getState}) 参数并且返回一系列的方法。在一系列方法的最深层，调用next(action)传递action到下一个中间件。

if (typeof action === 'function') {

return action (dispatch, getState, extraArgument);

}

return next(action);

};

}

const thunk = createThunkMiddleware();

thunk.withExtraArgument = createThunkMiddleware;

export default thunk; //默认导出createThunkMiddleware方法的返回结果

其中({dispatch, getState}) => (next) => (action) => {} 的结构是ES6的箭头函数写法，可以变得更加简洁。如果不使用es6写法，则扩展下来是：

function ({dispatch, getState})

return function(next) {

return function(action) {….}

}

}

P.S Redux的dataStore可以使用多个中间件。Action将从这些中间件形成的中间链中一个接一个的处理。当一个中间处理完action后，将通过调用next方法传给下一个中间件。最后传给dataStore的Dispatch方法。