Redux
中间件源码分享

林佳钰

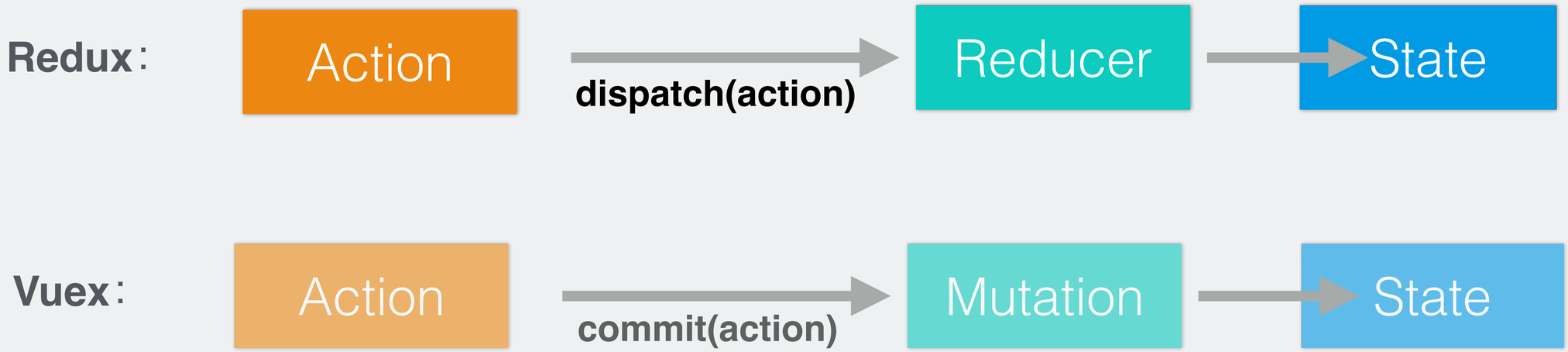
智能支付 用户前端组

Contents

1). Introduction to Redux

Redux is a **predictable state container** for JavaScript apps.

Data Flow



2). Functional Programming

In computer science, functional programming is a programming paradigm—a style of building the structure and elements of computer programs—that **treats computation as the evaluation of mathematical functions** and **avoids changing-state and mutable data**

- **Pure Function** $f(x) = 2x + 1$

- **Higher Order Functions**

  take functions as parameters & return functions as return values.
  e.g.： map, reduce, filter …..

- **Compose**

- **Currying**

2). Functional Programming

Compose

Composes functions from right to left.
This is a functional programming utility, and is included in Redux as a convenience.

$$f(x) = (x + 100) * 2 - 100$$

```
// 实现公式: f(x) = (x + 100) * 2 - 100
const add = a => a + 100;
const multiple = m => m * 2;
const subtract = s => s - 100;

// 深度嵌套函数模式 deeply nested function，将所有函数串联执行起来。
subtract(multiple(add(200)));
```

Compose

```
function compose(...funcs) {
  if (funcs.length === 0) {
    return arg => arg
  }
  if (funcs.length === 1) {
    return funcs[0]
  }
  return funcs.reduce((a, b) => (...args) => a(b(...args)))
}
```

```
// 实现公式: f(x) = (x + 100) * 2 - 100
const add = a => a + 100;
const multiple = m => m * 2;
const subtract = s => s - 100;


compose(subtract, multiple, add)(200);
```

2). Functional Programming

Currying

Currying is the technique of translating the evaluation of a function that takes **multiple arguments** into evaluating a sequence of functions, each with **a single argument**

```
// 实现公式: f(x, y, z) = (x + 100) * y - z;
const fn = (x, y, z) => (x + 100) * y - z;
fn(200, 2, 100);

// Curring实现 使用一层层包裹的单参匿名函数，来实现多参数函数的方法
const fn = x => y => z => (x + 100) * y - z;
fn(200)(2)(100);
```

3). Redux Middleware

Middleware Format:

middleware = store => next => action => { next(action); }

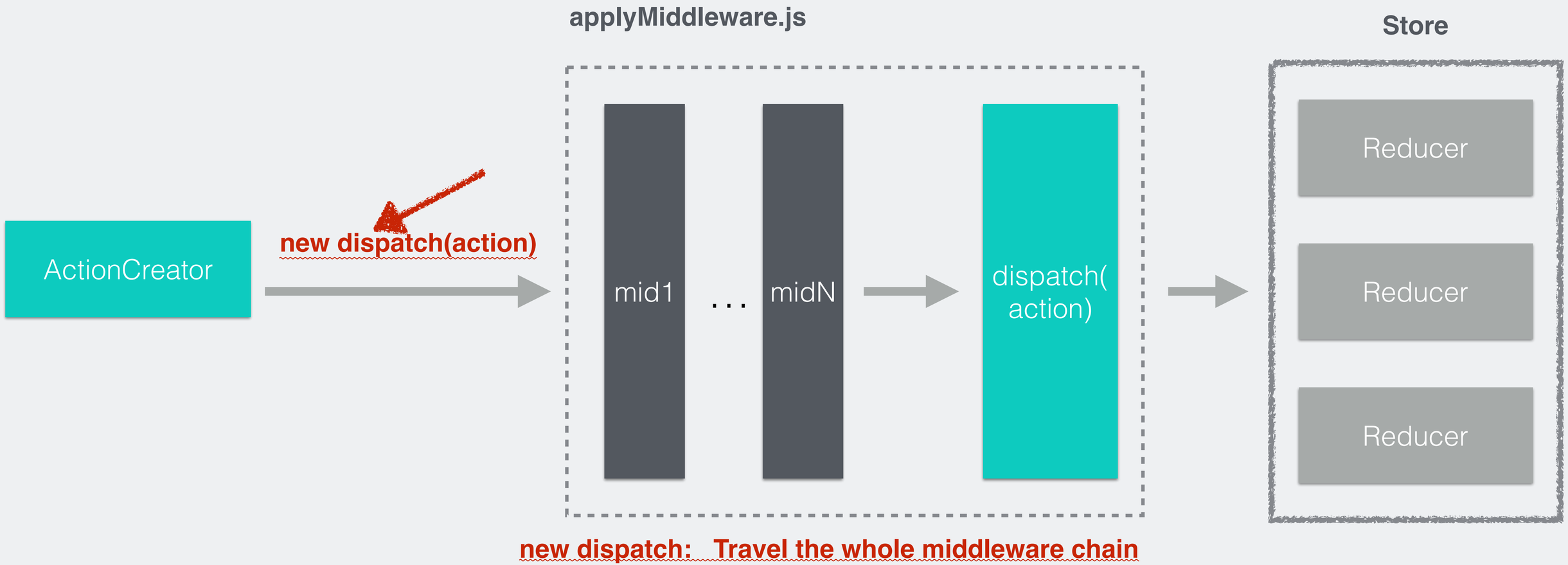
```
export default function applyMiddleware(...middlewares) {
  return (createStore) => (reducer, preloadedState, enhancer) => {
    const store = createStore(reducer, preloadedState, enhancer)
    let dispatch = store.dispatch
    let chain = []

    const middlewareAPI = {
      getState: store.getState,
      dispatch: (action) => dispatch(action)
    }
    chain = middlewares.map(middleware => middleware(middlewareAPI))
    dispatch = compose(...chain)(store.dispatch)

    return {
      ...store,
      dispatch
    }
  }
```

3). Redux Middleware

ApplyMiddleware.js

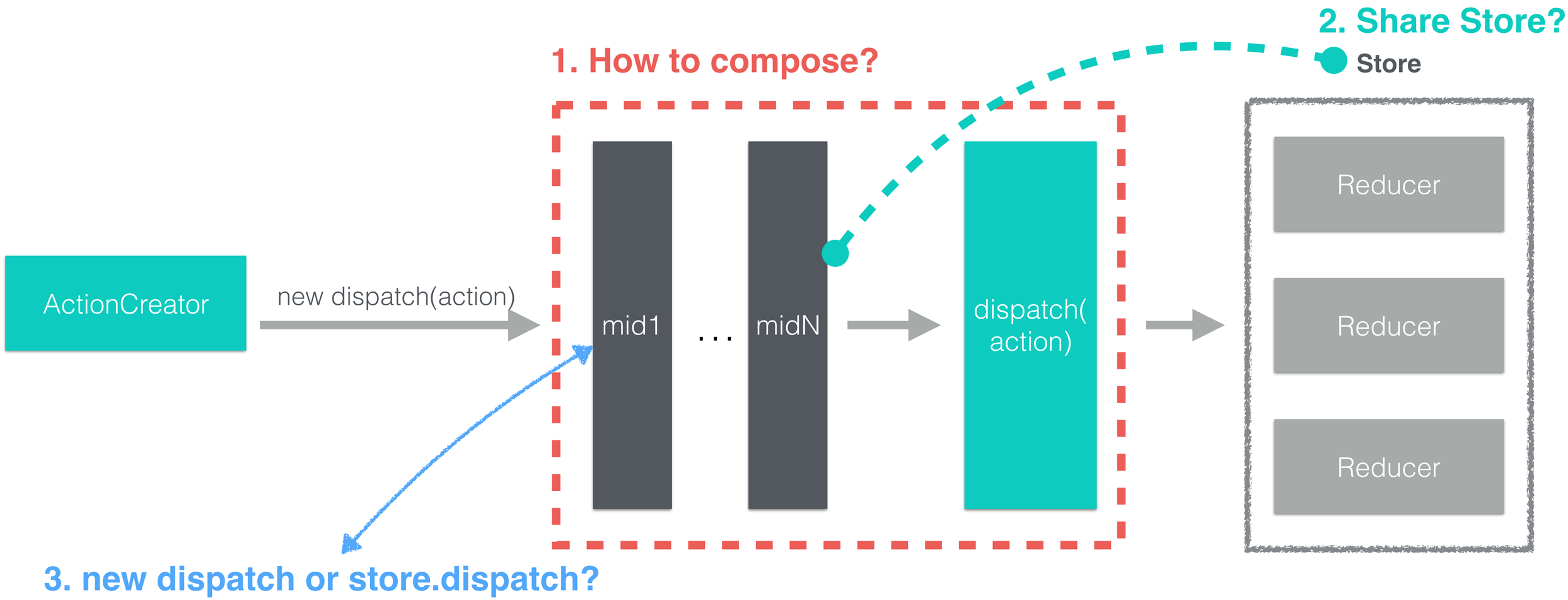
Aim:

Compose all middleware & Share one Store & Return new dispatch()



3). Redux Middleware

ApplyMiddleware.js



4. Middleware Format ? middleware = store => next => action => { next(action); }

3). Redux Middleware

1. How to compose?

Use compose(...middlewares) / deeply nested function

```
const middleware1 = action => action;
const middleware2 = action => action;
const final = action => store.dispatch(action);
/*
  1. compose(...)将所有中间件串联
  2. 定义final作为最后执行dispatch的函数
*/
compose(final, middleware2, middleware1)(action);
```

FP Compose

2. How to share store?

const koaMiddleware = async (ctx, next) => { };

```
const middleware1 = (store, action) => action;
const middleware2 = (store, action) => action;
const final = (store, action) => store.dispatch(action);
```

multiple arguments

```
// 柯里化处理参数
const middleware1 = store => action => action;
const middleware2 = store => action => action;
const final = store => action => store.dispatch(action);

// 将store保存在各个函数中 -> 循环执行处理。
const chain = [final, middleware2, middleware1].map(midItem => midItem(store));
compose(...chain)(action);
```

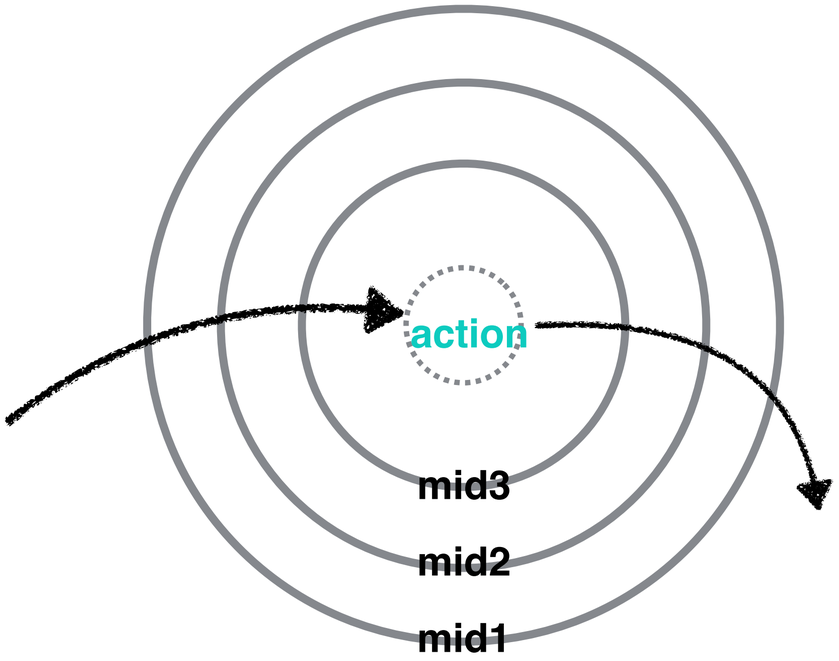
single arguments

FP Currying

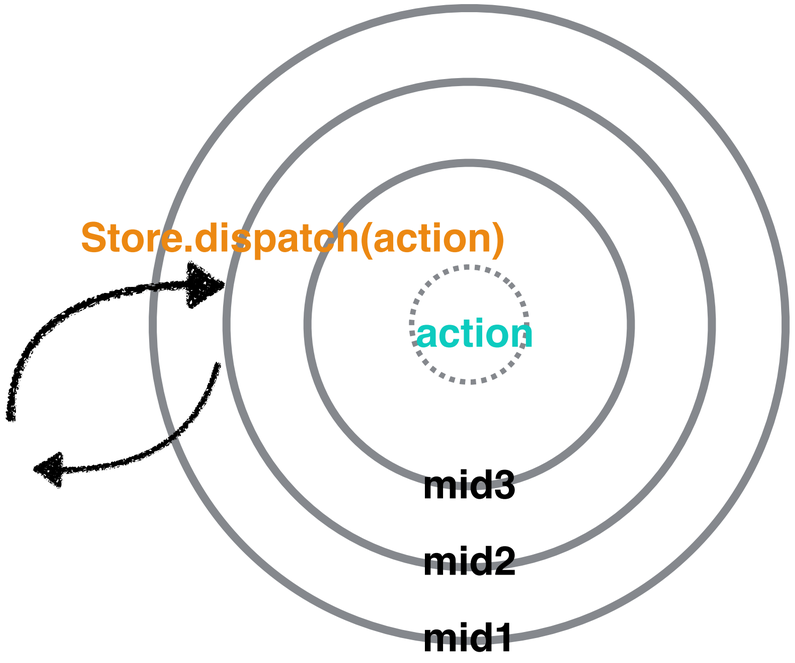
3. New dispatch or Store.dispatch?

Create store with new dispatch

```
// 将store保存在各个函数中 -> 循环执行处理。
const chain = [final, middleware2, middleware1].map(midItem => midItem(store));
```



正常中间件流程



如果有中间件调用dispatch方法会导致死循环

3. New dispatch or Store.dispatch?

```
// 这就是为什么在给所有middleware，共享Store的时候，会重新定义一遍getState和dispatch方法。
const middlewareAPI = {
  getState: store.getState,
  dispatch: (action) => dispatch(action)
}
chain = middlewares.map(middleware => middleware(middlewareAPI))
dispatch = compose(...chain)(store.dispatch)
```

3). Redux Middleware

4. Middleware Format?

middleware = store => next => action => { next(action); }

?

```
// 柯里化处理参数
const middleware1 = store => next => action => { log(1); next(action)};
const middleware2 = store => next => action => { log(2); next(action)};

// 中间件串联
const chain = [middleware1, middleware2 ].map(midItem => midItem({
  dispatch: (action) => store.dispatch(action)}));


// compose(...chain)会形成一个调用链，next指代下一个函数的注册，如果执行到了最后next就是原生的store.dispatch方法
dispatch = compose(...chain)(store.dispatch);
```

4. Middleware Format?

```
chain = [ mid1, mid2 ];

function mid1 (next) {
  return function (action) {
    next(action);
  }
}

function mid2 (next) {
  return function (action) {
    next(action);
  }
}
```

compose(...chain)

```
function mid2(mid1) {
  return function (action) {
    mid1(action);
  }
}
```

next用来注册下一个需要被执行的中间件
最后执行的是原生**store.dispatch**方法

applyMiddleware.js

3). Redux Middleware

```
export default function applyMiddleware(...middlewares) {
  return (createStore) => (reducer, preloadedState, enhancer) => {
    const store = createStore(reducer, preloadedState, enhancer)
    let dispatch = store.dispatch
    let chain = []

    const middlewareAPI = {
      getState: store.getState,
      dispatch: (action) => dispatch(action)
    }
    chain = middlewares.map(middleware => middleware(middlewareAPI))
    dispatch = compose(...chain)(store.dispatch)

    return {
      ...store,
      dispatch
    }
  }
}
```

2. Share store

1. compose

3. new dispatch

4). Koa Middleware

Register a middleware

```
const go = async (ctx, next) => {
  console.log('gogogogo LeiOLeiOLei');
  await next();
};
app.use(go);
```

```
use(fn) {
  if (typeof fn !== 'function') throw new TypeError('middleware must be a function!');
  if (isGeneratorFunction(fn)) {
    deprecate('Support for generators will be removed in v3. ' +
              'See the documentation for examples of how to convert old middleware ' +
              'https://github.com/koajs/koa/blob/master/docs/migration.md');
    fn = convert(fn);
  }
  debug('use %s', fn._name || fn.name || '-');
  this.middleware.push(fn);
  return this;
}
```

4). Koa Middleware

```
listen() {
  debug('listen');
  const server = http.createServer(this.callback());
  return server.listen.apply(server, arguments);
}
```

```
callback() {
  const fn = compose(this.middleware);
  console.log('callback - fn', fn);

  if (!this.listeners('error').length) this.on('error', this.onerror);

  const handleRequest = (req, res) => {
    res.statusCode = 404;
    const ctx = this.createContext(req, res);
    const onerror = err => ctx.onerror(err);
    const handleResponse = () => respond(ctx);
    onFinished(res, onerror);
    return fn(ctx).then(handleResponse).catch(onerror);
  };

  return handleRequest;
}
```

4). koa-compose

Register a middleware

await next();

```
function compose (middleware) {
  if (!Array.isArray(middleware)) throw new TypeError('Middleware stack must be an array!')
  for (const fn of middleware) {
    if (typeof fn !== 'function') throw new TypeError('Middleware must be composed of functions!')
  }

  /**
   * @param {Object} context
   * @return {Promise}
   * @api public
   */

  return function (context, next) {
    // last called middleware #
    let index = -1
    return dispatch(0)
    function dispatch (i) {
      if (i <= index) return Promise.reject(new Error('next() called multiple times'))
      index = i
      let fn = middleware[i]

      if (i === middleware.length) fn = next
      if (!fn) return Promise.resolve()
      try {
        return Promise.resolve(fn(context, function next () {
          return dispatch(i + 1)
        }))
      } catch (err) {
        return Promise.reject(err)
      }
    }
  }
}
```

Q & A