目录

[概念 1](#_Toc485567218)

[阻塞 1](#_Toc485567219)

[辅助函数 3](#_Toc485567220)

[takeEvery takeLatest 3](#_Toc485567221)

[effects 4](#_Toc485567222)

[call apply cps 4](#_Toc485567223)

[put 5](#_Toc485567224)

[take 6](#_Toc485567225)

[与takeEvery区别： 7](#_Toc485567226)

[fork 8](#_Toc485567227)

[join(task) 8](#_Toc485567228)

[cancel(task) 8](#_Toc485567229)

[select(selector, ...args) 9](#_Toc485567230)

[effect组合器 10](#_Toc485567231)

[race(effects) 10](#_Toc485567232)

[[...effects] (并行的 effects) 11](#_Toc485567233)

[task 12](#_Toc485567234)

[登录实例 12](#_Toc485567235)

# 概念

## 阻塞

比如，我们运行了一个无限循环的 while(true)。记住这是一个 Generator 函数，它不具备 从运行至完成的行为（run-to-completion behavior）。 Generator 将在每次迭代上阻塞以等待 action 发起。

其实就是不同yield之间要顺序执行。

## 基本逻辑

sagas 中的每个 Generator 函数被调用时，都会被传入 Redux Store 的 getState 方法作为第一个参数。

sagas 中的每个函数都必须返回一个 Generator 对象。 middleware 会迭代这个 Generator 并执行所有 yield 后的 Effect（译注：Effect 可以看作是 redux-saga 的任务单元，参考 名词解释）。

在第一次迭代里，middleware 会调用 next() 方法以取得下一个 Effect。然后 middleware 会通过下面提到的 Effects API 来执行 yield 后的 Effect。 与此同时，Generator 会暂停，直到 Effect 执行结束。当接收到执行的结果，middleware 在 Generator 里接着调用 next(result)，并将得到的结果作为参数传入。 这个过程会一直重复，直到 Generator 正常或通过抛出一些错误结束。

如果执行引发了一个错误（像提到的那些 Effect 创建器），就会调用 Generator 的 throw(error) 方法来代替。如果定义了一个 try/catch 包裹当前的 yield 指令， 那么 catch 区块将被底层 Generator runtime 调用。

# 中间件相关

## createSagaMiddleware(...sagas)

创建一个 Redux 中间件，将 Sagas 与 Redux Store 建立连接。

* sagas: Array - Generator 函数列表

返回一个中间件实例，这个实例暴露了一个 run 方法。你可以在创建saga中间件时不传入sagas，后面用这个方法来执行 Sagas。（也可以两次传入不同的sagas，不影响）

export default function configureStore(initialState) {

// 注意：redux@>=3.1.0 的版本才支持把 middleware 作为 createStore 方法的最后一个参数

return createStore(

reducer,

initialState,

applyMiddleware(/\* other middleware, \*/createSagaMiddleware(...sagas))

)

}

## middleware.run(saga, ...args)

动态执行 saga。用于 applyMiddleware 阶段之后执行 Sagas。

saga: Function: 一个 Generator 函数

args: Array: 提供给 saga 的参数 (除了 Store 的 getState 方法)

这个方法返回一个 Task 描述对象

# 辅助函数

## takeEvery takeLatest

相当于一个监听器，’\*’表示监听所有action

takeEvery 是最常见的，它提供了类似 redux-thunk 的行为

首先我们创建一个将执行异步 action 的任务：

import { call, put } from 'redux-saga/effects'

export function\* fetchData(action) {

try {

const data = yield call(Api.fetchUser, action.payload.url);

yield put({type: "FETCH\_SUCCEEDED", data});

} catch (error) {

yield put({type: "FETCH\_FAILED", error});

}

}

然后在每次 FETCH\_REQUESTED action 被发起时启动上面的任务。

import { takeEvery } from 'redux-saga'

function\* watchFetchData() {

yield\* takeEvery('FETCH\_REQUESTED', fetchData)

}

takeEvery 允许多个 fetchData 实例同时启动。在某个特定时刻，我们可以启动一个新的 fetchData 任务， 尽管之前还有一个或多个 fetchData 尚未结束。

如果我们只想得到最新那个请求的响应（例如，始终显示最新版本的数据）。我们可以使用 takeLatest辅助函数。

import { takeLatest } from 'redux-saga'

function\* watchFetchData() {

yield\* takeLatest('FETCH\_REQUESTED', fetchData)

}

和 takeEvery 不同，在任何时刻 takeLatest 只允许执行一个 fetchData 任务。并且这个任务是最后被启动的那个。 如果之前已经有一个任务在执行，那之前的这个任务会自动被取消。

# effects

## call apply cps

简单来说就是对函数进行了一层封装，方便测试。

call(fn, ...args)

call([context, fn], ...args)

apply(context, fn, args)

cps(fn, ...args)

import { call } from 'redux-saga/effects'

function\* fetchProducts() {

const products = yield call(Api.fetch, '/products')

// ...

}

测试

import { call } from 'redux-saga/effects'

import Api from '...'

const iterator = fetchProducts()

// expects a call instruction

assert.deepEqual(

iterator.next().value,

call(Api.fetch, '/products'),

"fetchProducts should yield an Effect call(Api.fetch, './products')"

)

这些 声明式调用（declarative calls） 的优势是，我们可以通过简单地遍历 Generator 并**在 yield 后的成功的值上面做一个 deepEqual 测试**， 就能测试 Saga 中所有的逻辑。这是一个真正的好处，因为复杂的异步操作都不再是黑盒，你可以详细地测试操作逻辑，不管它有多么复杂。

call 同样支持**调用对象方法**，你可以使用以下形式，为调用的函数提供一个 this 上下文：

yield call([obj, obj.method], arg1, arg2, ...) // 如同 obj.method(arg1, arg2 ...)

apply 提供了另外一种调用的方式：

yield **apply**(obj, obj.method, [arg1, arg2, ...])

call 和 apply 非常适合返回 Promise 结果的函数。另外一个函数**cps** 可以用来处理 Node 风格的函数 （例如，fn(...args, callback) 中的 callback 是 (error, result) => () 这样的形式，cps 表示的是延续传递风格（Continuation Passing Style））。

举个例子：

import { cps } from 'redux-saga'

const content = yield cps(readFile, '/path/to/file')

当然你也可以像测试 call 一样测试它：

import { cps } from 'redux-saga/effects'

## put

对dispatch进行了封装，目的和call类似都是为了方便测试。

import { call, put } from 'redux-saga/effects'

//...

function\* fetchProducts() {

const products = yield call(Api.fetch, '/products')

// 创建并 yield 一个 dispatch Effect

yield put({ type: 'PRODUCTS\_RECEIVED', products })

}

测试

const iterator = fetchProducts()

// 期望一个 call 指令

assert.deepEqual(

iterator.next().value,

call(Api.fetch, '/products'),

"fetchProducts should yield an Effect call(Api.fetch, './products')"

)

// 创建一个假的响应对象

const products = {}

// 期望一个 dispatch 指令

assert.deepEqual(

iterator.next(products).value,

put({ type: 'PRODUCTS\_RECEIVED', products }),

"fetchProducts should yield an Effect put({ type: 'PRODUCTS\_RECEIVED', products })"

)

## take

创建一条 Effect 描述信息，指示 middleware 等待 Store 上指定的 action。 Generator 会暂停，直到一个与pattern 匹配的 action 被发起。

用以下规则来解释 pattern：

* 如果调用 take 时参数为空，或者传入 '\*'，那将会匹配所有发起的 action（例如，take() 会匹配所有的 action）。
* 如果是一个函数，action 会在 pattern(action) 返回为 true 时被匹配（例如，take(action =>action.entities) 会匹配那些 entities 字段为真的 action）。
* 如果是一个字符串，action 会在 action.type === pattern 时被匹配（例如，take(INCREMENT\_ASYNC)）。
* 如果参数是一个数组，会针对数组所有项，匹配与 action.type 相等的 action（例如，take([INCREMENT, DECREMENT]) 会匹配 INCREMENT 或 DECREMENT 类型的 action）。
* import { take, put } from 'redux-saga/effects'
* function\* watchFirstThreeTodosCreation() {
* for(let i = 0; i < 3; i++) {
* const action = yield take('TODO\_CREATED')
* }
* yield put({type: 'SHOW\_CONGRATULATION'})
* }

### 与takeEvery区别：

主动拉取action

这样的反向控制让我们能够实现一些使用传统的 push 方法做非常规事情的控制流。

主动拉取 action 的另一个好处是我们可以使用熟悉的同步风格来描述我们的控制流。举个例子，假设我们希望实现一个这样的登录控制流，有两个 action 分别是 LOGIN 和 LOGOUT。 使用 takeEvery（或 redux-thunk）我们必须要写两个分别的任务（或 thunks）：一个用于 LOGIN，另一个用于 LOGOUT。

结果就是我们的逻辑现在分开在两个地方了。别人为了阅读我们的代码搞明白这是怎么回事，他必须阅读两个处理函数的源代码并且要在两处逻辑之间建立连接。 这意味着他必须通过在心中重新排列放在几个不同地方的代码逻辑获得正确的排序，从而在脑中重建控制流模型。

使用拉取（pull）模式，我们可以在同一个地方写控制流。

function\* loginFlow() {

while(true) {

yield take('LOGIN')

// ... perform the login logic

yield take('LOGOUT')

// ... perform the logout logic

}

}

与重复处理相同的 action 不同，loginFlow Saga 更好理解，因为序列中的 actions 是期望之中的。它知道LOGIN action 后面应该始终跟着一个 LOGOUT action。LOGOUT 也应该始终跟在一个 LOGIN 后面（一个好的 UI 程序应该始终强制执行顺序一致的 actions，通过隐藏或禁用意料之外的 action）。

## fork

fork(fn, ...args)

fork([context, fn], ...args)

支持为 fn 指定 this 上下文（调用对象方法）。功能类似call，cps,但是无阻塞。

创建一条 Effect 描述信息，指示 middleware 以 无阻塞调用 方式执行 fn。

#### 参数

* fn: Function - 一个 Generator 函数, 或者返回 Promise 的普通函数
* args: Array - 一个数组，作为 fn 的参数

**注意**

fork 类似于 call，可以用来调用普通函数和 Generator 函数。但 fork 的调用是无阻塞的，在等待fn 返回结果时，middleware 不会暂停 Generator。 相反，一旦 fn 被调用，Generator 立即恢复执行。

fork 与 race 类似，是一个中心化的 Effect，管理 Sagas 间的并发。

yield fork(fn ...args) 的结果是一个 [Task](http://leonshi.com/redux-saga-in-chinese/docs/api/index.html#task) 对象 —— 一个具备某些有用的方法和属性的对象。

## join(task)

指示 middleware 等待之前的 fork 任务返回结果。

task: Task - 之前的 fork 指令返回的 Task 对象

## cancel(task)

创建一条 Effect 描述信息，指示 middleware 取消之前的 fork 任务。

取消执行中的 Generator，middleware 将会抛出一个 SagaCancellationException 的错误。

取消会向下传播。当取消 Generator 时，middleware 会同时取消当前 Effect（阻塞当前 Generator 的 Effect）。 如果当前 Effect 调用了另一个 Generator，那这个 Generator 也会被取消。

被取消的 Generator 可以捕获 SagaCancellationException 错误，以便在它结束前执行某些清理逻辑（例如，如果 Generator 正处于 AJAX 调用，清除 state 中的 isFetching 标识）。

注意，未被捕获的 SagaCancellationException 不会向上冒泡，如果 Generator 不处理取消异常，异常将不会冒泡至它的父级 Generator。

cancel 是一个无阻塞 Effect。也就是说，Generator 将在取消异常被抛出后立即恢复。

对于返回 Promise 结果的函数，你可以通过给 promise 附加一个 [CANCEL] 来插入自己的取消逻辑。

import { fork, cancel, CANCEL } from 'redux-saga/effects'

function myApi() {

const promise = myXhr(...)

promise[CANCEL] = () => myXhr.abort()

return promise

}

function\* mySaga() {

const task = yield fork(myApi)

// ... 过一会儿

// 将会调用 myApi 上的 promise[CANCEL]

yield cancel(task)

}

## select(selector, ...args)

获取state。比如select(state => state);

作用：将需要getState的地方抽取成一个一个单独的文件（selector）（如果不这样，当我们需要获取state时，需要将顶层的getState一层层传下来。），这本身只是一个封装函数，重要的是我们自己写的selector。

创建一条 Effect 描述信息，指示 middleware **调用提供的选择器**获取 Store state 上的数据

* selector: Function - 一个 (state, ...args) => args 函数. 通过当前 state 和一些可选参数，返回当前 Store state 上的部分数据。
* args: Array - 可选参数，传递给选择器（附加在 getState 后）

如果 select 调用时参数为空（即 yield select()），那 effect 会取得整个的 state（和调用 getState() 的结果一样）。

在发起 action 到 store 时，middleware 首先会转发 action 到 reducers 然后通知 Sagas。这意味着，当你查询 Store 的 state， 你获取的是 action 被处理之后的 state。

import { take, fork, select } from 'redux-saga/effects'

import { getCart } from './selectors'

function\* checkout() {

// 使用选择器查询 state

**const cart = yield select(getCart)**

// ... 调用某些 Api 然后发起一个 success/error action

}

export default function\* rootSaga() {

while(true) {

yield take('CHECKOUT\_REQUEST')

// 不再需要传递 getState 参数了

yield fork(checkout)

}

}

# effect组合器

## race(effects)

创建一条 Effect 描述信息，指示 middleware 在多个 Effect 之间执行一个 race（类似 Promise.race([...]) 的行为，只要有一个状态改变，race的状态就跟着改变）。

effects: Object - 一个 {label: effect, ...} 形式的字典对象(键值对)（可以直接call，也可以take监听）

在 resolve race 的时候，middleware 会自动取消所有失效的 Effect（译注：即剩余的 Effect）。

function\* fetchUsersSaga {

const { response, cancel } = yield race({

response: call(fetchUsers),

cancel: take(CANCEL\_FETCH)

})

}

## [...effects] (并行的 effects)

创建一条 Effect 描述信息，指示 middleware 并行执行多个 Effect，并等待所有 Effect 完成。

function\* mySaga() {

const [customers, products] = yield [

call(fetchCustomers),

call(fetchProducts)

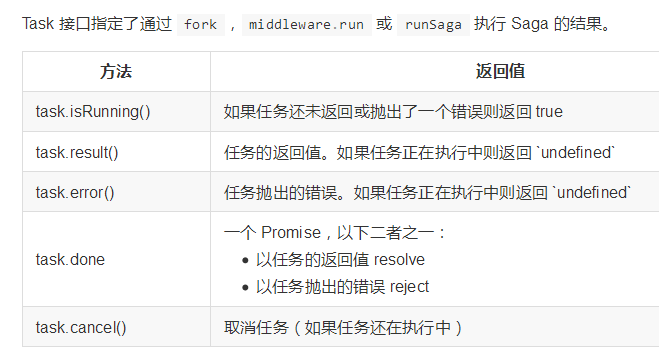
]

}

并行执行多个 Effect 时，middleware 会暂停 Generator，直到以下情况之一：

* 所有 Effect 成功完成：Generator 恢复并返回一个包含所有 Effect 结果的数组。
* 在所有 Effect 完成之前，有一个 Effect 被 reject 了：Generator 抛出 reject 错误。

# task



# 外部api

## runSaga

runSaga(iterator, {subscribe, dispatch, getState}, [monitor])

## 登录实例

import { isCancelError } from 'redux-saga'

import { take, call, put } from 'redux-saga/effects'

import Api from '...'

function\* authorize(user, password) {

try {

const token = yield call(Api.authorize, user, password)

yield put({type: 'LOGIN\_SUCCESS', token})

return token

} catch(error) {

if(!isCancelError(error))

yield put({type: 'LOGIN\_ERROR', error})

}

}

function\* loginFlow() {

while(true) {

const {user, password} = yield take('LOGIN\_REQUEST')

// fork return a Task object

const task = yield fork(authorize, user, password)

const action = yield take(['LOGOUT', 'LOGIN\_ERROR'])

if(action.type === 'LOGOUT')

yield cancel(task)

yield call(Api.clearItem('token'))

}

你可能已经注意到了，我们仍然没有做任何与清除 isLoginPending 状态相关的事情。 对于这一点，有 2 个可能的解决方案（或其他的）。

* 发起一个指定的 action RESET\_LOGIN\_PENDING。
* 或者更简单，让 reducer 收到 LOGOUT action 时清除 isLoginPending。