Table of Contents

redux-saga

`redux-saga` is a library that aims to make side effects (i.e. asynchronous things like data fetching and impure things like accessing the browser cache) in React/Redux applications easier and better.

The mental model is that a saga is like a separate thread in your application that's solely responsible for side effects. `redux-saga` is a redux middleware, which means this thread can be started, paused and cancelled from the main application with normal redux actions, it has access to the full redux application state and it can dispatch redux actions as well.

It uses an ES6 feature called Generators to make those asynchronous flows easy to read, write and test. *(if you're not familiar with them here are some introductory links)* By doing so, these asynchronous flows look like your standard synchronous JavaScript code. (kind of like `async` / `await`, but generators have a few more awesome features we need)

You might've used `redux-thunk` before to handle your data fetching. Contrary to redux thunk, you don't end up in callback hell, you can test your asynchronous flows easily and your actions stay pure.

Getting started

Install

```
$ npm install --save redux-saga
```

or

```
$ yarn add redux-saga
```

Alternatively, you may use the provided UMD builds directly in the `<script>` tag of an HTML page. See this section.

Usage Example

Suppose we have an UI to fetch some user data from a remote server when a button is clicked. (For brevity, we'll just show the action triggering code.)

```
class UserComponent extends React.Component {
  ...
  onSomeButtonClicked() {
    const { userId, dispatch } = this.props
    dispatch({type: 'USER_FETCH_REQUESTED', payload: {userId}})
  }
  ...
}
```

The Component dispatches a plain Object action to the Store. We'll create a Saga that watches for all `USER_FETCH_REQUESTED` actions and triggers an API call to fetch the user data.

`sagas.js`

```
import { call, put, takeEvery, takeLatest } from 'redux-saga/effects'
import Api from '...'

// worker Saga: will be fired on USER_FETCH_REQUESTED actions
function* fetchUser(action) {
   try {
      const user = yield call(Api.fetchUser, action.payload.userId);
      yield put({type: "USER_FETCH_SUCCEEDED", user: user});
   } catch (e) {
      yield put({type: "USER_FETCH_FAILED", message: e.message});
   }
}

/*
  Starts fetchUser on each dispatched `USER_FETCH_REQUESTED` action.
  Allows concurrent fetches of user.
*/
function* mySaga() {
  yield takeEvery("USER_FETCH_REQUESTED", fetchUser);
}

/*
  Alternatively you may use takeLatest.

  Does not allow concurrent fetches of user. If "USER_FETCH_REQUESTED" gets
  dispatched while a fetch is already pending, that pending fetch is cancelled
  and only the latest one will be run.
*/
function* mySaga() {
  yield takeLatest("USER_FETCH_REQUESTED", fetchUser);
}

export default mySaga;
```

To run our Saga, we'll have to connect it to the Redux Store using the `redux-saga` middleware.

`main.js`

```
import { createStore, applyMiddleware } from 'redux'
import createSagaMiddleware from 'redux-saga'

import reducer from './reducers'
import mySaga from './sagas'

// create the saga middleware
const sagaMiddleware = createSagaMiddleware()
// mount it on the Store
const store = createStore(
  reducer,
  applyMiddleware(sagaMiddleware)
)

// then run the saga
sagaMiddleware.run(mySaga)

// render the application
```

Documentation

- Introduction
- Basic Concepts
- Advanced Concepts
- Recipes
- External Resources
- Troubleshooting
- Glossary
- API Reference

Translation

- Chinese
- Chinese Traditional
- Japanese
- Korean
- Russian

Using umd build in the browser

There is also a **umd** build of `redux-saga` available in the `dist/` folder. When using the umd build `redux-saga` is available as `ReduxSaga` in the window object.

The umd version is useful if you don't use Webpack or Browserify. You can access it directly from unpkg.

The following builds are available:

- https://unpkg.com/redux-saga/dist/redux-saga.js
- https://unpkg.com/redux-saga/dist/redux-saga.min.js

Important! If the browser you are targeting doesn't support *ES2015 generators*, you must provide a valid polyfill, such as the one provided by `babel`. The polyfill must be imported before **redux-saga**:

```
import 'babel-polyfill'
// then
import sagaMiddleware from 'redux-saga'
```

Building examples from sources

```
$ git clone https://github.com/yelouafi/redux-saga.git
$ cd redux-saga
$ npm install
$ npm test
```

Below are the examples ported (so far) from the Redux repos.

Counter examples

There are three counter examples.

counter-vanilla

Demo using vanilla JavaScript and UMD builds. All source is inlined in `index.html`.

To launch the example, just open `index.html` in your browser.

> Important: your browser must support Generators. Latest versions of Chrome/Firefox/Edge are suitable.

counter

Demo using `webpack` and high-level API `takeEvery`.

```
$ npm run counter

# test sample for the generator
$ npm run test-counter
```

cancellable-counter

Demo using low-level API to demonstrate task cancellation.

```
$ npm run cancellable-counter
```

Shopping Cart example

```
$ npm run shop

# test sample for the generator
$ npm run test-shop
```

async example

```
$ npm run async

# test sample for the generators
$ npm run test-async
```

real-world example (with webpack hot reloading)

```
$ npm run real-world

# sorry, no tests yet
```

Logo

You can find the official Redux-Saga logo with different flavors in the logo directory.

Backers

Support us with a monthly donation and help us continue our activities. [Become a backer]

Sponsors

Become a sponsor and get your logo on our README on Github with a link to your site. [Become a sponsor]

redux-saga

Redux 應用程式的另一種 Side Effect 模型。代替 redux-thunk 發送的 thunk。你可以在一個地方建立 *Sagas* 來集中所有的 Side Effect 邏輯。

應用程式的邏輯會存在於 2 個地方：

- Reducers 負責 actions 之間的狀態轉變。
- Sagas 負責編排複雜/非同步的操作。

使用 Generator 函式來建立 Sagas。

> 接下來的內容，你將看見。Generators，雖然看起來比 ES7 aysnc 函式還低階（low level），但能提供像是陳述性作用（delcarative effects）、取消（cancellation）等功能。這些困難功能無法用單純的 async 函式實作出來。

此中介軟體提出了

- 可組合的抽象 **Effect**：等候某個 action；觸發 State 改變（透過分派 actions 到 store）；呼叫某個遠端服務；等不同形式的 Effects。一個 Saga 可使用熟悉的流程控制組成（if、while、for、try/catch）組合這些 Effects。
- Sage 本身就是一種 Effect。能夠過使用協調器（combinators）來組合其他 Effects。也可被其它 Sagas 呼叫，提供最大化的 Subroutines 及 Structured Programming。
- Effects 可能會以陳述方式（declaratively）所引起（yielded）。你引起的是 Effect 的描述，中介軟體會負責執行它。讓你在 Generators 內的邏輯能夠充分地進行測試。
- 你可以實作複雜的邏輯操作，橫跨多個 actions（例如：用戶入職訓練、精靈對話框、複雜遊戲規則…），這些非一般的表達。

開始入門

安裝

```
npm install redux-saga
```

創造 Saga（採用 Redux 的計數器範例）

```
import { take, put } from 'redux-saga'
// sagas/index.js

function* incrementAsync() {

  while(true) {

    // 等待每個 INCREMENT_ASYNC action
    const nextAction = yield take(INCREMENT_ASYNC)

    // delay 是範例函式
    // 回傳 Promise，會在指定的毫秒（ms）後解決（resolves）
    yield delay(1000)

    // 分派 INCREMENT_COUNTER
    yield put( increment() )
  }

}

export default [incrementAsync]
```

將 redux-saga 接到中介軟體管道

```
// store/configureStore.js
import sagaMiddleware from 'redux-saga'
import sagas from '../sagas'

export default function configureStore(initialState) {
  // Note: passing middleware as the last argument to createStore requires redux@>=3.1
.0
  return createStore(
    reducer,
    initialState,
    applyMiddleware(/* other middleware, */sagaMiddleware(...sagas))
  }
}
```

等候未來的 actions

前一個範例中，我們創造了一個 `等候未來的 actions` Saga。其中 `yield take(INCREMENT_ASYNC)` 的呼叫是一個 Sagas 如何運作的典型實例。

通常情況下，實際是由中介軟體們掌控這些 Effect 的構成，由 Action Creator 所觸發。舉例來說，redux-thunk 掌控 *thunks*，並將 `(getState, dispatch)` 作爲參數帶入，redux-promise 掌控 Promises，分派其解決後的值。redux-gen 掌控 generators，分派所有引起（yielded）的 actions 到 store 之中。這裡所有的中介軟體都有個共通點，就是 '由每個 action 呼叫' 樣式。當 action 發生時，它們將會一次又一次的被呼叫，換言之，它們的範圍由觸發它們的 *root action* 決定。

Sagas 運作方式不同，並不是由 Action Creators 所觸發，而是與你的應用程式一起並決定哪個使用者 actions 需要關注（watch）。就像是在背景執行的服務，選擇自己的邏輯進展。在上述範例中，`incrementAsync` 使用 `yield take(...)` 來拉 `INCREMENT_ASYNC` action。這是一種阻塞式呼叫，表示 Saga 不會繼續進行，直到收到符合的 action。

上述使用了 `take(INCREMENT_ASYNC)` 形式，表示正在等候 type 爲 `INCREMENT_ASYNC` 的 action。

`take` 支援幾種樣式以便約束符合的 actions。`yield take(PATTERN)` 的呼叫根據以下規則進行掌控：

- 當 PATTERN 是 undefined 或 `'*'` 時。所有新進的 actions 都會符合（例如，`take()` 將會匹配所有的 actions）
- 當 PATTERN 是 函式 時，只有當 PATTERN(action) 爲眞時，action 才會匹配。（例如，`take(action => action.entities)` 將會匹配所有有 `entities` 欄位的 action）。

- 當 PATTERN 是 字串 時，只有當 action.type === PATTERN 時才會匹配（如同上述範例的 `take(INCREMENT_ASYNC)`）
- 當 PATTERN 是 陣列 時，只有當 action.type 符合陣列中的其中一個元素時才會匹配（例如，`take([INCREMENT, DECREMENT])` 將會匹配 `INCREMENT` 或 `DECREMENT`。）

派送 actions 到 store

接收到查詢的 action 之後，Saga 觸發 `delay(1000)` 呼叫，在這個範例中，回傳了 Promise，並會在 1 秒後解決。這是個組塞式呼叫，因此 Saga 將會等候 1 秒後繼續。

延遲之後，Saga 使用 `put(action)` 函式分派了一個 `INCREMENT_COUNTER` action。相同地，這裡也將會等候分派後的結果。如果分派呼叫回傳的是一般的值，Saga 立即地恢復繼續（asap），但如果是個 Promise，則會等候 Promise 解決（或拒絕）。

常見的抽象：Effect

為了一般化，等待未來的 action；等待未來的結果，像是呼叫 `yield delay(1000)`；或者等待分派的結果，都是相同的概念。所有的案例都在引起某些 Effects 形式。

而 Saga 所做的事，實際上是將這些所有 effects 組合在一起，以便實作想要的控制流程。最簡單的方式是一個 yeidls 接著另一個 yields，循序引起 Effects。也可以使用熟悉的控制流程操作子（if、while、for）來實作複雜的控制流程。或者你想要使用 Effects 協調器來表達並發（concurrency，yield race）及平行（parallelism，yield [...]）。甚至可以引起其他的 Sagas，讓你擁有強大的 routine/subroutine 樣式。

舉例來說，`incrementAsync` 使用無窮迴圈 `while(true)` 來表示將會永遠運作於應用程式的生命週期之內。

你也可以創造有限時間的 Sagas。例如，下列 Saga 等候前 3 個 `INCREMENT_COUNTER` actions 並觸發 `showCongratulation()` action，滿足後便會結束。

```
function* onBoarding() {

  for(let i = 0; i < 3; i++)
    yield take(INCREMENT_COUNTER)

  yield put( showCongratulation() )
}
```

陳述性 Effects

Sagas Generators 可以引起多種形式的 Effects。最簡單的是引起 Promise

```
function* fetchSaga() {

  // fetch 是簡單函式
  // 回傳 Promise 將會解決 GET 回應
  const products = yield fetch('/products')

  // 分派 RECEIVE_PRODUCTS action
  yield put( receiveProducts(products) )
}
```

上述範例中， `fetch('/products')` 回傳 Promise 將會解決 GET 回應，所以 'fetch effect' 立即地執行。簡單且符合語言習慣，但是…

假設我們要測試上述 generator

```
const iterator = fetchSaga()
assert.deepEqual( iterator.next().value, ?? ) // 該期待什麼結果 ?
```

我們想要檢查 generator 第一個引起的結果，在這個案例中是執行 `fetch('/products')` 之後的結果。測試中執行實際的服務不是個可行的方式，也不是實踐的方法，所以我們需要仿製這個 fetch 服務，換言之，我們需要假的來替換眞正的 `fetch` 方法，假的並不會實際發出 GET 請求，而是用來檢查是否用了正確的參數（在這個案例中，正確的參數爲 `'/products'` ）呼叫 `fetch` 。

仿製讓測試更加困難、更不可靠。另一方面，函式單純回傳值讓測試更容易，單純使用 `equal()` 來檢查結果。這是最可靠的撰寫測試方法。

不相信？鼓勵你閱讀 Eric Elliott 的這篇文章。

> (...) `equal()`，就本質回答兩個最重要的問題，每個單元測試都必須回答，但大多數都不會回答：
>
> - 實際輸出是什麼？
> - 期望輸出是什麼？
>
> 如果你完成一個測試但沒有回答上述兩個問題，那就不是一個眞正的單元測試。你有的只是一個草率的、不完整的測試。

而我們實際所需的，只是需要確保 `fetchSaga` 引起的呼叫，其呼叫的函式以及參數是正確的。因此，此函式庫提供一些陳述性的方式來引起 Side Effects，讓 Saga 的邏輯更容易測試

```
import { call } from 'redux-saga'

function* fetchSaga() {
  const products = yield call( fetch, '/products' ) // 不會執行 effect
}
```

這裡使用了 `call(fn, ...args)` 函式。與先前範例的差異之處在於，並不會立即地執行呼叫 **fetch**，取而代之的是，`call` 創造出 **effect** 的描述。如同你在 Redux，使用 action creators 創造出 object 描述 action，將會被 Store 執行，`call` 創造出 object 描述函式呼叫。redux-saga 中介軟體負責函式呼叫並帶著解決的回應再開始 generator。

這讓我們在 Redux 環境以外也能夠容易地測試 Generator。

```
import { call } from 'redux-saga'

const iterator = fetchSaga()
assert.deepEqual(iterator.next().value, call(fetch, '/products')) // 預期的是 call(...) 值
```

現在，不再需要仿製任何事情。簡單的相等測試便足夠。

陳述式 effects 的優點是測試所有在 Saga/Generator 內的邏輯，簡單地反覆檢查迭代器的結果值，持續地對值進行單純的相等測試。真正的好處是，複雜的非同步操作不再是黑盒子，無論多複雜都可以詳細地測試，

呼叫某些物件的方法（即使用 `new` 創造），你可以用下列形式提供 `this` 上下文（context）到調用的函式

```
yield call([obj, obj.method], arg1, arg2, ...) // 如同我們使用 obj.method(arg1, arg2 ...)
```

`apply` 是個化名，用於方法調用形式

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

`call` 及 `apply` 適合在回傳 Promise 的函式。其他函式可用 `cps` 來處理 Node 風格函式（例如，`fn(...args, callback)` 其中 `callback` 是 `(error, result) => ()` 形式）。舉例來說

```
import { cps } from 'redux-saga'

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

當然你也可以測試它

```
import { cps } from 'redux-saga'

const iterator = fetchSaga()
assert.deepEqual(iterator.next().value, cps(readFile, '/path/to/file') )
```

同樣 `cps` 支援相同的方法調用形式，如同 `call` 一樣。

錯誤處理

Generator 裡面可以使用單純的 try/catch 語句來捕獲錯誤。在下列範例中，Saga 從 `api.buyProducts` 呼叫（即拒絕的 Promise）中捕獲錯誤

```
function* checkout(getState) {

  while( yield take(types.CHECKOUT_REQUEST) ) {
    try {
      const cart = getState().cart
      yield call(api.buyProducts, cart)
      yield put(actions.checkoutSuccess(cart))
    } catch(error) {
      yield put(actions.checkoutFailure(error))
    }
  }
}
```

當然並不是強迫你要用 try/catch 區塊來處理你的 API 錯誤，你也可以讓 API 服務回傳一般值並帶有錯誤旗標

```
function buyProducts(cart) {
  return doPost(...)
    .then(result => {result})
    .catch(error => {error})
}

function* checkout(getState) {
  while( yield take(types.CHECKOUT_REQUEST) ) {
    const cart = getState().cart
    const {result, error} = yield call(api.buyProducts, cart)
    if(!error)
      yield put(actions.checkoutSuccess(result))
    else
      yield put(actions.checkoutFailure(error))
  }
}
```

Effect 協調器

`yield` 陳述式非常適合用來表示非同步控制流程，一種簡單且線性的風格。但是我們同樣地需要平行運作。無法單純的撰寫

```
// 錯誤，effects 將會依序執行
const users  = yield call(fetch, '/users'),
      repose = yield call(fetch, '/repose')
```

因爲直到第 1 個呼叫解決之前，第 2 個 effect 並不會執行。取而代之，我們要寫成

```
import { call } from 'redux-saga'

// 正確，effects 將會平行地執行
const [users, repose]  = yield [
  call(fetch, '/users'),
  call(fetch, '/repose')
]
```

當我們引起一個陣列的 effects，generator 將會阻塞直到所有 effects 都被解決（或者一旦其中有一個被拒絕，如同 `Promise.all` 行爲）。

有時候平行發出多個任務並不希望等待所有任務都被解決，而是只需要一個 贏家：第一個被解決（或拒絕）。函式 `race` 提供了多個 effects 之間的競賽功能。

下列範例顯示 Saga 觸發了一個遠端擷取請求，並且限制該請求在 1 秒後超時。

```
import { race, take, put } from 'redux-saga'

function* fetchPostsWithTimeout() {
  while( yield take(FETCH_POSTS) ) {
    // 發出 2 個 effects 之間的競賽
    const {posts, timeout} = yield race({
      posts   : call(fetchApi, '/posts'),
      timeout : call(delay, 1000)
    })

    if(posts)
      put( actions.receivePosts(posts) )
    else
      put( actions.timeoutError() )
  }
}
```

透過 yield* 的順序性 Sagas

你可以使用內建的 `yield*` 操作子依序組合多個 sagas。讓你可以用簡單的程序式風格來依序執行 *macro-tasks*

```
function* playLevelOne(getState) { ... }

function* playLevelTwo(getState) { ... }

function* playLevelThree(getState) { ... }

function* game(getState) {

  const score1 = yield* playLevelOne(getState)
  put(showScore(score1))

  const score2 = yield* playLevelTwo(getState)
  put(showScore(score2))

  const score3 = yield* playLevelThree(getState)
  put(showScore(score3))

}
```

請注意使用 `yield*` 會導致 JavaScript 的執行期間傳播整個順序。迭代器的結果會引起巢狀迭代器。更強大的代替方式是採用更一般化的中介軟體構成機制。

組合 Sagas

雖然使用 `yield*` 提供一種語言習慣的方式來組合 Sagas。但這個方式有些限制：

- 你可能希望分開測試巢狀的 generators。這導致某些重複的測試程式碼產生以及重複執行的損耗。我們不希望執行一個巢狀 generator，只希望確保正確的參數呼叫。
- 更重要地，`yield*` 只允許順序性的任務組成，你一次只能 yield* 一個 generator。

你可以簡單地使用 `yield` 來開始一個或平行多個子任務。當引起一個呼叫到 generator，Saga 將會等候 generator 終止才開始處理，接著使用回傳值再開始（或者拋出錯誤，當錯誤來自子任務）。

```
function* fetchPosts() {
  yield put( actions.requestPosts() )
  const products = yield call(fetchApi, '/products')
  yield put( actions.receivePosts(products) )
}

function* watchFetch() {
  while ( yield take(FETCH_POSTS) ) {
    yield call(fetchPosts) // 等候 fetchPosts 任務結束
  }
}
```

引起一個巢狀 generatos 的陣列將會平行地開始所有 sub-generators 並等候全部完成。接著用所有結果值再開始

```
function* mainSaga(getState) {
  const results = yield [ call(task1), call(task2), ...]
  yield put( showResults(results) )
}
```

事實上，引起 Sagas 與引起其他 effects 並無不同（未來 actions、超時 …）。這表示你可以使用 effect 調節器結合這些 Sagas 及其他類型。

舉例來說，你希望使使用者在限定時間內完成某些遊戲

```
function* game(getState) {

  let finished
  while(!finished) {
    // 必須在 60 秒內完成
    const {score, timeout}  = yield race({
      score  : call( play, getState),
      timeout : call(delay, 60000)
    })

    if(!timeout) {
      finished = true
      yield put( showScore(score) )
    }
  }

}
```

非阻塞式的呼叫 — fork/join

`yield` 陳述式造成 generator 暫停直到引起的 effect 被解決或拒絕。如果你近一點看看這個範例

```
function* watchFetch() {
  while ( yield take(FETCH_POSTS) ) {
    yield put( actions.requestPosts() )
    const posts = yield call(fetchApi, '/posts') // 阻塞式呼叫
    yield put( actions.receivePosts(posts) )
  }
}
```

`watchFetch` generator 將會等候直到 `yield call(fetchApi, '/posts')` 結束。想像 `FETCH_POSTS` 是透過一個 `重整` 按鈕觸發。如果我們應用程式在每個 fetch 之間關閉按鈕（非並發 fetch）那這樣不會有問題，因爲我們知道不會有額外的 `FETCH_POSTS` action 觸發直到取得 `fetchApi` 呼叫的回應。

但是如果應用程式希望允許使用者點擊 `重整` 按鈕，而不需要等候目前的請求結束？

下列範例描述了一個可能的事件順序

```
UI                                watchFetch
--------------------------------------------------------
FETCH_POSTS.....................呼叫 fetchApi........... 等待解決
........................................................
........................................................
FETCH_POSTS............................................. 遺漏
........................................................
FETCH_POSTS............................................. 遺漏
................................fetchApi 回傳...........
........................................................
```

當 `watchFetch` 阻塞於 `fetchApi` 呼叫，所有在呼叫及回應之間發生的 `FETCH_POSTS` 都會被遺漏。

爲了表達非阻塞式呼叫，我們可以使用 `fork` 函式。用 `fork` 改寫前述範例的一種可能寫法是

```
import { fork, call, take, put } from 'redux-saga'

function* fetchPosts() {
  yield put( actions.requestPosts() )
  const posts = yield call(fetchApi, '/posts')
  yield put( actions.receivePosts(posts) )
}

function* watchFetch() {
  while ( yield take(FETCH_POSTS) ) {
    yield fork(fetchPosts) // 非阻塞式呼叫
  }
}
```

`fork` ，就像是 `call` ，接受 函式/generator 呼叫。

```
yield fork(func, ...args)       // 單純非同步函式 (...) -> Promise
yield fork(generator, ...args)  // Generator 函式
```

`yield fork(api)` 的結果是個 任務描述子。爲了在稍候能夠取得 forked 任務的結果，我們使用 `join` 函式

```
import { fork, join } from 'redux-saga'

function* child() { ... }

function *parent() {
  // 非阻塞式呼叫
  const task = yield fork(subtask, ...args)

  // ... 稍候
  // 現在是阻塞式呼叫，將會再開始帶著任務的結果
  const result = yield join(task)

}
```

任務物件公開幾個有益的方法

方法	回傳值
task.isRunning()	回傳 true 當任務尚未回傳，或者拋出錯誤
task.result()	任務回傳的結果。`undefined` 當任務仍然在執行中
task.error()	任務拋出的錯誤。`undefined` 當任務仍然在執行中
task.done	下列兩種其一的 Promise ● 帶有任務回傳值的解決 ● 帶有任務拋出的錯誤的拒絕

任務取消

當一個任務已經開始，你可以使用 `yield cancel(task)` 來終止執行。取消執行中任務將會導致內部拋出 `SagaCancellationException` 錯誤。

爲了看如何運作，讓我們考慮一個簡單的範例。一個背景同步功能，可以透過某些 UI 命令來開始/暫停。根據接收 `START_BACKGROUND_SYNC` action，我們將開始一個背景任務，週期性地從遠端伺服器同步某些資料。

任務持續的執行，直到 `STOP_BACKGROUND_SYNC` action 觸發。接著將會取消背景任務並且再次等待下一個 `START_BACKGROUND_SYNC` action。

```
import { take, put, call, fork, cancel, SagaCancellationException } from 'redux-saga'
import actions from 'somewhere'
import { someApi, delay } from 'somewhere'

function* bgSync() {
  try {
    while(true) {
      yield put(actions.requestStart())
      const result = yield call(someApi)
      yield put(actions.requestSuccess(result))
      yield call(delay, 5000)
    }
  } catch(error) {
    if(error instanceof SagaCancellationException)
      yield put(actions.requestFailure('Sync cancelled!'))
  }
}

function* main() {
  while( yield take(START_BACKGROUND_SYNC) ) {
    // 開始一個任務於背景執行
    const bgSyncTask = yield fork(bgSync)

    // 等候使用者的停止 action
    yield take(STOP_BACKGROUND_SYNC)
    // 使用者點選了停止。取消背景任務
    // 這會拋出 SagaCancellationException 例外到背景執行的任務
    yield cancel(bgSyncTask)
  }
}
```

`yield cancel(bgSyncTask)` 將會拋出 `SagaCancellationException` 在目前執行的任務之中。在上述範例中，例外由 `bgSync` 捕獲。注意：未被捕獲的 SagaCancellationException 不會向上冒起。如上述範例，若 `bgSync` 未捕獲取消例外，則例外將不會傳播到 `main` （因爲 `main` 已經繼續往下執行）。

取消執行中任務也會取消目前 effect，也就是取消當下的任務。

舉例來說，假設在應用程式生命週期的某個時間點，我們有個待定的呼叫鏈

```
function* main() {
  const task = yield fork(subtask)
  ...
  // 稍候
  yield cancel(task)
}

function* subtask() {
  ...
  yield call(subtask2) // 目前被此呼叫阻塞
  ...
}

function* subtask2() {
  ...
  yield call(someApi) // 目前被此呼叫阻塞
  ...
}
```

`yield cancel(task)` 將觸發取消 `subtask` ，接著觸發取消 `subtask2` 。 `SagaCancellationException` 將會拋到 `subtask2` 之中，接著拋到 `subtask` 之中。如果 `subtask` 省略對取消例外的處理，console 將會顯示顯示警告訊息來警告開發者（訊息只有當變數 `process.env.NODE_ENV` 設定為 `'development'` 的時候才會顯示）

取消例外的主要用意在於，讓被取消的任務可以執行清理邏輯。這讓應用程式不會在狀態不一致狀況下離開，在上述背景同步的範例中，透過捕獲取消例外， `bgSync` 能夠分派 `requestFailure` action 到 store。否則，store 可能留下一種不一致的狀態（例如，等候待定請求的結果）

> 很重要的一件事，請記住 `yield cancel(task)` 並不會等候被取消的任務完成（即執行 catch 內的區塊）。cancel effect 行為像是 fork。一旦 cancel 被初始化之後便會返回。一旦取消，一般情況下，清理的邏輯要盡快完成。某些情況下，清理的邏輯可能牽涉某些非同步的操作，但取消的任務是存在分開的 process，沒有辦法 rejoin 回到主要的控制流程（除了透過 Redux store 分派 actions 到其他任務。然而，這將帶領到複雜的控制流程，難以推理。更好的方式是盡可能的快速結束取消的任務）。

自動的取消

除了手動取消之外。有某些案例會自動觸發取消

1- 在 `race` effect 中。所有 race 競爭者，除了贏家，其餘皆會自動取消。

2- 在平行 effect（ `yield [...]` ）中。一旦有一個 sub-effects 被拒絕，平行 effect 將很快的被拒絕（如同 Promise.all）。這個情況下，所有其他的 sub-effects 將會自動取消。

動態啓動 Sagas — runSaga

`runSaga` 函示讓你能夠在 Redux 中介軟體環境之外開始 sagas。也讓你能夠勾到外部的 input/output，不同於 store actions。

舉例來說，你可以在伺服端開始一個 Saga

```
import serverSaga from 'somewhere'
import {runSaga, storeIO} from 'redux-saga'
import configureStore from 'somewhere'
import rootReducer from 'somewhere'

const store = configureStore(rootReducer)
runSaga(
  serverSaga(store.getState),
  storeIO(store)
).done.then(...)
```

`runSaga` 回傳一個任務物件。就像是 `fork` effect 回傳的一樣。

與取得及分派 action 到 store 不同， `runSaga` 也可以連接到其他 input/output 來源。讓你可以在 Redux 以外的世界也能夠用 sagas 的功能來實作你的控制流程。

此方法的函數簽名如下

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

參數

- `iterator: {next, throw}` : 迭代器物件，典型地使用 Generator 創造出來
- `subscribe(callback) => unsubscribe` : 換言之，接受回呼函示的函示，回傳取消訂閱的函示
    - `callback(action)` : 回呼函示（由 runSaga 提供）用來訂閱 input 事件。 `subscribe` 必須支援註冊多個訂閱者
    - `unsubscribe()` : 由 `runSaga` 使用，一旦 input 來源完成之後，用來取消訂閱（一般的回傳或拋出例外）
- `dispatch(action) => result` : 用來實現 `put` effects。每當發出 `yield put(action)` ， `dispatch` 將與 `action` 一起調用。 `dispatch` 的回傳值將用來實現 `put` effect。Promise 結果將自動地解決/拒絕。

- `monitor(sagaAction)` （optional）：用來分派所有 Saga 相關事件的回呼函示。在中介軟體的版本中，所有 actions 將被分派到 Redux store。請見 [sagaMonitor 使用範例](https://github.com/redux-saga/redux-saga/blob/master/examples/sagaMonitor.js).

`subscribe` 用來實現 `take(action)` effect。每當 `subscribe` 發出 action 到其回呼函示，所有 sagas 將被 `take(PATTERN)` 阻塞，而取得符合目前進入的 action 樣式將會再開始動作。

從原始碼組建範例

```
git clone https://github.com/redux-saga/redux-saga.git
cd redux-saga
npm install
npm test
```

下列範例（某個程度）從 Redux 存放庫移植

計數器範例

```
npm run counter

// generator 的測試
npm run test-counter
```

購物車範例

```
npm run shop

// generator 的測試
npm run test-shop
```

非同步範例

```
npm run async

// 抱歉，還沒有測試
```

真實世界範例（包含 webpack hot reloading）

```
cd examples/real-world
npm install
npm start
```

在瀏覽器使用 umd 組建

`redux-saga` 有 **umd** 組建位於 `dist/` 目錄之下。使用 `redux-saga` 的 umd 組建可以從 window 物件下的 `ReduxSaga` 取得。當你沒有使用 webpack 或 browserify 時，umd 版本是非常有用的，你可以直接從 unpkg 取得。包含下列組建：

- https://unpkg.com/redux-saga/dist/redux-saga.js
- https://unpkg.com/redux-saga/dist/redux-saga.min.js

重要！ 如果你的目標瀏覽器不支援 *es2015 generators*，你需要提供合適的 polyfill，例如，*babel* 所提供的：browser-polyfill.min.js。這個 polyfill 必須在 **redux-saga** 之前載入。

Introduction

Beginner Tutorial

Objectives of this tutorial

This tutorial attempts to introduce redux-saga in a (hopefully) accessible way.

For our getting started tutorial, we are going to use the trivial Counter demo from the Redux repo. The application is quite simple but is a good fit to illustrate the basic concepts of redux-saga without being lost in excessive details.

The initial setup

Before we start, clone the tutorial repository.

> The final code of this tutorial is located in the `sagas` branch.

Then in the command line, run:

```
$ cd redux-saga-beginner-tutorial
$ npm install
```

To start the application, run:

```
$ npm start
```

We are starting with the simplest use case: 2 buttons to `Increment` and `Decrement` a counter. Later, we will introduce asynchronous calls.

If things go well, you should see 2 buttons `Increment` and `Decrement` along with a message below showing `Counter: 0`.

> In case you encountered an issue with running the application. Feel free to create an issue on the tutorial repo.

Hello Sagas!

We are going to create our first Saga. Following the tradition, we will write our 'Hello, world' version for Sagas.

Create a file `sagas.js` then add the following snippet:

```
export function* helloSaga() {
  console.log('Hello Sagas!')
}
```

So nothing scary, just a normal function (except for the `*`). All it does is print a greeting message into the console.

In order to run our Saga, we need to:

- create a Saga middleware with a list of Sagas to run (so far we have only one `helloSaga`)
- connect the Saga middleware to the Redux store

We will make the changes to `main.js` :

```
// ...
import { createStore, applyMiddleware } from 'redux'
import createSagaMiddleware from 'redux-saga'

// ...
import { helloSaga } from './sagas'

const sagaMiddleware = createSagaMiddleware()
const store = createStore(
  reducer,
  applyMiddleware(sagaMiddleware)
)
sagaMiddleware.run(helloSaga)

const action = type => store.dispatch({type})

// rest unchanged
```

First we import our Saga from the `./sagas` module. Then we create a middleware using the factory function `createSagaMiddleware` exported by the `redux-saga` library.

Before running our `helloSaga` , we must connect our middleware to the Store using `applyMiddleware` . Then we can use the `sagaMiddleware.run(helloSaga)` to start our Saga.

So far, our Saga does nothing special. It just logs a message then exits.

Making Asynchronous calls

Now let's add something closer to the original Counter demo. To illustrate asynchronous calls, we will add another button to increment the counter 1 second after the click.

First thing's first, we'll provide an additional callback `onIncrementAsync` to the UI component.

```
const Counter = ({ value, onIncrement, onDecrement, onIncrementAsync }) =>
  <div>
    {' '}
    <button onClick={onIncrementAsync}>
      Increment after 1 second
    </button>
    <hr />
    <div>
      Clicked: {value} times
    </div>
  </div>
```

Next we should connect the `onIncrementAsync` of the Component to a Store action.

We will modify the `main.js` module as follows

```
function render() {
  ReactDOM.render(
    <Counter
      value={store.getState()}
      onIncrement={() => action('INCREMENT')}
      onDecrement={() => action('DECREMENT')}
      onIncrementAsync={() => action('INCREMENT_ASYNC')} />,
    document.getElementById('root')
  )
}
```

Note that unlike in redux-thunk, our component dispatches a plain object action.

Now we will introduce another Saga to perform the asynchronous call. Our use case is as follows:

> On each `INCREMENT_ASYNC` action, we want to start a task that will do the following
>
> - Wait 1 second then increment the counter

Add the following code to the `sagas.js` module:

```
import { delay } from 'redux-saga'
import { put, takeEvery } from 'redux-saga/effects'

// Our worker Saga: will perform the async increment task
export function* incrementAsync() {
  yield delay(1000)
  yield put({ type: 'INCREMENT' })
}

// Our watcher Saga: spawn a new incrementAsync task on each INCREMENT_ASYNC
export function* watchIncrementAsync() {
  yield takeEvery('INCREMENT_ASYNC', incrementAsync)
}
```

Time for some explanations.

We import `delay` , a utility function that returns a Promise that will resolve after a specified number of milliseconds. We'll use this function to *block* the Generator.

Sagas are implemented as Generator functions that *yield* objects to the redux-saga middleware. The yielded objects are a kind of instruction to be interpreted by the middleware. When a Promise is yielded to the middleware, the middleware will suspend the Saga until the Promise completes. In the above example, the `incrementAsync` Saga is suspended until the Promise returned by `delay` resolves, which will happen after 1 second.

Once the Promise is resolved, the middleware will resume the Saga, executing code until the next yield. In this example, the next statement is another yielded object: the result of calling `put({type: 'INCREMENT'})` , which instructs the middleware to dispatch an `INCREMENT` action.

`put` is one example of what we call an *Effect*. Effects are simple JavaScript objects which contain instructions to be fulfilled by the middleware. When a middleware retrieves an Effect yielded by a Saga, the Saga is paused until the Effect is fulfilled.

So to summarize, the `incrementAsync` Saga sleeps for 1 second via the call to `delay(1000)` , then dispatches an `INCREMENT` action.

Next, we created another Saga `watchIncrementAsync` . We use `takeEvery` , a helper function provided by `redux-saga` , to listen for dispatched `INCREMENT_ASYNC` actions and run `incrementAsync` each time.

Now we have 2 Sagas, and we need to start them both at once. To do that, we'll add a `rootSaga` that is responsible for starting our other Sagas. In the same file `sagas.js` , add the following code:

```
// single entry point to start all Sagas at once
export default function* rootSaga() {
  yield [
    incrementAsync(),
    watchIncrementAsync()
  ]
}
```

This Saga yields an array with the results of calling our two sagas, `helloSaga` and `watchIncrementAsync`. This means the two resulting Generators will be started in parallel. Now we only have to invoke `sagaMiddleware.run` on the root Saga in `main.js`.

```
// ...
import rootSaga from './sagas'

const sagaMiddleware = createSagaMiddleware()
const store = ...
sagaMiddleware.run(rootSaga)

// ...
```

Making our code testable

We want to test our `incrementAsync` Saga to make sure it performs the desired task.

Create another file `sagas.spec.js`:

```
import test from 'tape';

import { incrementAsync } from './sagas'

test('incrementAsync Saga test', (assert) => {
  const gen = incrementAsync()

  // now what ?
});
```

`incrementAsync` is a generator function. When run, it returns an iterator object, and the iterator's `next` method returns an object with the following shape

```
gen.next() // => { done: boolean, value: any }
```

The `value` field contains the yielded expression, i.e. the result of the expression after the `yield`. The `done` field indicates if the generator has terminated or if there are still more 'yield' expressions.

In the case of `incrementAsync`, the generator yields 2 values consecutively:

1. `yield delay(1000)`
2. `yield put({type: 'INCREMENT'})`

So if we invoke the next method of the generator 3 times consecutively we get the following results:

```
gen.next() // => { done: false, value: <result of calling delay(1000)> }
gen.next() // => { done: false, value: <result of calling put({type: 'INCREMENT'})> }
gen.next() // => { done: true, value: undefined }
```

The first 2 invocations return the results of the yield expressions. On the 3rd invocation since there is no more yield the `done` field is set to true. And since the `incrementAsync` Generator doesn't return anything (no `return` statement), the `value` field is set to `undefined`.

So now, in order to test the logic inside `incrementAsync`, we'll simply have to iterate over the returned Generator and check the values yielded by the generator.

```
import test from 'tape';

import { incrementAsync } from './sagas'

test('incrementAsync Saga test', (assert) => {
  const gen = incrementAsync()

  assert.deepEqual(
    gen.next(),
    { done: false, value: ??? },
    'incrementAsync should return a Promise that will resolve after 1 second'
  )
});
```

The issue is how do we test the return value of `delay`? We can't do a simple equality test on Promises. If `delay` returned a *normal* value, things would've been easier to test.

Well, `redux-saga` provides a way to make the above statement possible. Instead of calling `delay(1000)` directly inside `incrementAsync`, we'll call it *indirectly*:

```
// ...
import { delay } from 'redux-saga'
import { put, call, takeEvery } from 'redux-saga/effects'

export function* incrementAsync() {
  // use the call Effect
  yield call(delay, 1000)
  yield put({ type: 'INCREMENT' })
}
```

Instead of doing `yield delay(1000)`, we're now doing `yield call(delay, 1000)`. What's the difference?

In the first case, the yield expression `delay(1000)` is evaluated before it gets passed to the caller of `next` (the caller could be the middleware when running our code. It could also be our test code which runs the Generator function and iterates over the returned Generator). So what the caller gets is a Promise, like in the test code above.

In the second case, the yield expression `call(delay, 1000)` is what gets passed to the caller of `next`. `call` just like `put`, returns an Effect which instructs the middleware to call a given function with the given arguments. In fact, neither `put` nor `call` performs any dispatch or asynchronous call by themselves, they simply return plain JavaScript objects.

```
put({type: 'INCREMENT'}) // => { PUT: {type: 'INCREMENT'} }
call(delay, 1000)        // => { CALL: {fn: delay, args: [1000]}}
```

What happens is that the middleware examines the type of each yielded Effect then decides how to fulfill that Effect. If the Effect type is a `PUT` then it will dispatch an action to the Store. If the Effect is a `CALL` then it'll call the given function.

This separation between Effect creation and Effect execution makes it possible to test our Generator in a surprisingly easy way:

```
import test from 'tape';

import { put, call } from 'redux-saga/effects'
import { delay } from 'redux-saga'
import { incrementAsync } from './sagas'

test('incrementAsync Saga test', (assert) => {
  const gen = incrementAsync()

  assert.deepEqual(
    gen.next().value,
    call(delay, 1000),
    'incrementAsync Saga must call delay(1000)'
  )

  assert.deepEqual(
    gen.next().value,
    put({type: 'INCREMENT'}),
    'incrementAsync Saga must dispatch an INCREMENT action'
  )

  assert.deepEqual(
    gen.next(),
    { done: true, value: undefined },
    'incrementAsync Saga must be done'
  )

  assert.end()
});
```

Since `put` and `call` return plain objects, we can reuse the same functions in our test code. And to test the logic of `incrementAsync` , we simply iterate over the generator and do `deepEqual` tests on its values.

In order to run the above test, run:

```
$ npm test
```

which should report the results on the console.

Background on the Saga concept

WIP

For now, here are some useful links.

External links

- Applying the Saga Pattern (Youtube video) By Caitie McCaffrey
- Original paper By Hector Garcia-Molina & Kenneth Salem
- A Saga on Sagas from MSDN site

Basic Concepts

Using Saga Helpers

`redux-saga` provides some helper effects wrapping internal functions to spawn tasks when some specific actions are dispatched to the Store.

The helper functions are built on top of the lower level API. In the advanced section, we'll see how those functions can be implemented.

The first function, `takeEvery` is the most familiar and provides a behavior similar to `redux-thunk` .

Let's illustrate with the common AJAX example. On each click on a Fetch button we dispatch a `FETCH_REQUESTED` action. We want to handle this action by launching a task that will fetch some data from the server.

First we create the task that will perform the asynchronous action:

```
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})
   }
}
```

To launch the above task on each `FETCH_REQUESTED` action:

```
import { takeEvery } from 'redux-saga/effects'

function* watchFetchData() {
  yield takeEvery('FETCH_REQUESTED', fetchData)
}
```

In the above example, `takeEvery` allows multiple `fetchData` instances to be started concurrently. At a given moment, we can start a new `fetchData` task while there are still one or more previous `fetchData` tasks which have not yet terminated.

If we want to only get the response of the latest request fired (e.g. to always display the latest version of data) we can use the `takeLatest` helper:

```
import { takeLatest } from 'redux-saga/effects'

function* watchFetchData() {
  yield takeLatest('FETCH_REQUESTED', fetchData)
}
```

Unlike `takeEvery`, `takeLatest` allows only one `fetchData` task to run at any moment. And it will be the latest started task. If a previous task is still running when another `fetchData` task is started, the previous task will be automatically cancelled.

If you have multiple Sagas watching for different actions, you can create multiple watchers with those built-in helpers which will behave like there was `fork` used to spawn them (we'll talk about `fork` later. For now consider it to be an Effect that allows us to start multiple sagas in the background)

For example:

```
import { takeEvery } from 'redux-saga'

// FETCH_USERS
function* fetchUsers(action) { ... }

// CREATE_USER
function* createUser(action) { ... }

// use them in parallel
export default function* rootSaga() {
  yield takeEvery('FETCH_USERS', fetchUsers)
  yield takeEvery('CREATE_USER', createUser)
}
```

Declarative Effects

In `redux-saga` , Sagas are implemented using Generator functions. To express the Saga logic we yield plain JavaScript Objects from the Generator. We call those Objects *Effects*. An Effect is simply an object which contains some information to be interpreted by the middleware. You can view Effects like instructions to the middleware to perform some operation (invoke some asynchronous function, dispatch an action to the store).

To create Effects, you use the functions provided by the library in the `redux-saga/effects` package.

In this section and the following, we will introduce some basic Effects. And see how the concept allows the Sagas to be easily tested.

Sagas can yield Effects in multiple forms. The simplest way is to yield a Promise.

For example suppose we have a Saga that watches a `PRODUCTS_REQUESTED` action. On each matching action, it starts a task to fetch a list of products from a server.

```
import { takeEvery } from 'redux-saga/effects'
import Api from './path/to/api'

function* watchFetchProducts() {
  yield takeEvery('PRODUCTS_REQUESTED', fetchProducts)
}

function* fetchProducts() {
  const products = yield Api.fetch('/products')
  console.log(products)
}
```

In the example above, we are invoking `Api.fetch` directly from inside the Generator (In Generator functions, any expression at the right of `yield` is evaluated then the result is yielded to the caller).

`Api.fetch('/products')` triggers an AJAX request and returns a Promise that will resolve with the resolved response, the AJAX request will be executed immediately. Simple and idiomatic, but...

Suppose we want to test generator above:

```
const iterator = fetchProducts()
assert.deepEqual(iterator.next().value, ??) // what do we expect ?
```

We want to check the result of the first value yielded by the generator. In our case it's the result of running `Api.fetch('/products')` which is a Promise . Executing the real service during tests is neither a viable nor practical approach, so we have to *mock* the `Api.fetch` function, i.e. we'll have to replace the real function with a fake one which doesn't actually run the AJAX request but only checks that we've called `Api.fetch` with the right arguments (`'/products'` in our case).

Mocks make testing more difficult and less reliable. On the other hand, functions that simply return values are easier to test, since we can use a simple `equal()` to check the result. This is the way to write the most reliable tests.

Not convinced? I encourage you to read Eric Elliott's article:

> (...) `equal()`, by nature answers the two most important questions every unit test must answer, but most don't:
>
> - What is the actual output?
> - What is the expected output?
>
> If you finish a test without answering those two questions, you don't have a real unit test. You have a sloppy, half-baked test.

What we actually need is just to make sure the `fetchProducts` task yields a call with the right function and the right arguments.

Instead of invoking the asynchronous function directly from inside the Generator, **we can yield only a description of the function invocation**. i.e. We'll simply yield an object which looks like

```
// Effect -> call the function Api.fetch with `./products` as argument
{
  CALL: {
    fn: Api.fetch,
    args: ['./products']
  }
}
```

Put another way, the Generator will yield plain Objects containing *instructions*, and the `redux-saga` middleware will take care of executing those instructions and giving back the result of their execution to the Generator. This way, when testing the Generator, all we need to do is to check that it yields the expected instruction by doing a simple `deepEqual` on the yielded Object.

For this reason, the library provides a different way to perform asynchronous calls.

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

function* fetchProducts() {
  const products = yield call(Api.fetch, '/products')
  // ...
}
```

We're using now the `call(fn, ...args)` function. **The difference from the preceding example is that now we're not executing the fetch call immediately, instead, `call` creates a description of the effect**. Just as in Redux you use action creators to create a plain object describing the action that will get executed by the Store, `call` creates a plain object describing the function call. The redux-saga middleware takes care of executing the function call and resuming the generator with the resolved response.

This allows us to easily test the Generator outside the Redux environment. Because `call` is just a function which returns a plain Object.

```
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')"
)
```

Now we don't need to mock anything, and a simple equality test will suffice.

The advantage of those *declarative calls* is that we can test all the logic inside a Saga by simply iterating over the Generator and doing a `deepEqual` test on the values yielded successively. This is a real benefit, as your complex asynchronous operations are no longer black boxes, and you can test in detail their operational logic no matter how complex it is.

`call` also supports invoking object methods, you can provide a `this` context to the invoked functions using the following form:

```
yield call([obj, obj.method], arg1, arg2, ...) // as if we did obj.method(arg1, arg2 .
..)
```

`apply` is an alias for the method invocation form

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

`call` and `apply` are well suited for functions that return Promise results. Another function `cps` can be used to handle Node style functions (e.g. `fn(...args, callback)` where `callback` is of the form `(error, result) => ()`). `cps` stands for Continuation Passing Style.

For example:

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

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

And of course you can test it just like you test `call` :

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

const iterator = fetchSaga()
assert.deepEqual(iterator.next().value, cps(readFile, '/path/to/file') )
```

`cps` also supports the same method invocation form as `call` .

Dispatching actions to the store

Taking the previous example further, let's say that after each save, we want to dispatch some action to notify the Store that the fetch has succeeded (we'll omit the failure case for the moment).

We could pass the Store's `dispatch` function to the Generator. Then the Generator could invoke it after receiving the fetch response:

```
// ...

function* fetchProducts(dispatch) {
  const products = yield call(Api.fetch, '/products')
  dispatch({ type: 'PRODUCTS_RECEIVED', products })
}
```

However, this solution has the same drawbacks as invoking functions directly from inside the Generator (as discussed in the previous section). If we want to test that `fetchProducts` performs the dispatch after receiving the AJAX response, we'll need again to mock the `dispatch` function.

Instead, we need the same declarative solution. Just create an Object to instruct the middleware that we need to dispatch some action, and let the middleware perform the real dispatch. This way we can test the Generator's dispatch in the same way: by just inspecting the yielded Effect and making sure it contains the correct instructions.

The library provides, for this purpose, another function `put` which creates the dispatch Effect.

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

function* fetchProducts() {
  const products = yield call(Api.fetch, '/products')
  // create and yield a dispatch Effect
  yield put({ type: 'PRODUCTS_RECEIVED', products })
}
```

Now, we can test the Generator easily as in the previous section

```
import { call, put } 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')"
)

// create a fake response
const products = {}

// expects a dispatch instruction
assert.deepEqual(
  iterator.next(products).value,
  put({ type: 'PRODUCTS_RECEIVED', products }),
  "fetchProducts should yield an Effect put({ type: 'PRODUCTS_RECEIVED', products })"
)
```

Note how we pass the fake response to the Generator via its `next` method. Outside the middleware environment, we have total control over the Generator, we can simulate a real environment by simply mocking results and resuming the Generator with them. Mocking data is a lot simpler than mocking functions and spying calls.

Error handling

In this section we'll see how to handle the failure case from the previous example. Let's suppose that our API function `Api.fetch` returns a Promise which gets rejected when the remote fetch fails for some reason.

We want to handle those errors inside our Saga by dispatching a `PRODUCTS_REQUEST_FAILED` action to the Store.

We can catch errors inside the Saga using the familiar `try/catch` syntax.

```
import Api from './path/to/api'
import { call, put } from 'redux-saga/effects'

// ...

function* fetchProducts() {
  try {
    const products = yield call(Api.fetch, '/products')
    yield put({ type: 'PRODUCTS_RECEIVED', products })
  }
  catch(error) {
    yield put({ type: 'PRODUCTS_REQUEST_FAILED', error })
  }
}
```

In order to test the failure case, we'll use the `throw` method of the Generator

```
import { call, put } 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')"
)

// create a fake error
const error = {}

// expects a dispatch instruction
assert.deepEqual(
  iterator.throw(error).value,
  put({ type: 'PRODUCTS_REQUEST_FAILED', error }),
  "fetchProducts should yield an Effect put({ type: 'PRODUCTS_REQUEST_FAILED', error }
)"
)
```

In this case, we're passing the `throw` method a fake error. This will cause the Generator to break the current flow and execute the catch block.

Of course, you're not forced to handle your API errors inside `try` / `catch` blocks. You can also make your API service return a normal value with some error flag on it. For example, you can catch Promise rejections and map them to an object with an error field.

```
import Api from './path/to/api'
import { call, put } from 'redux-saga/effects'

function fetchProductsApi() {
  return Api.fetch('/products')
    .then(response => ({ response }))
    .catch(error => ({ error }))
}

function* fetchProducts() {
  const { response, error } = yield call(fetchProductsApi)
  if (response)
    yield put({ type: 'PRODUCTS_RECEIVED', products: response })
  else
    yield put({ type: 'PRODUCTS_REQUEST_FAILED', error })
}
```

A common abstraction: Effect

To generalize, triggering Side Effects from inside a Saga is always done by yielding some declarative Effect. (You can also yield Promise directly, but this will make testing difficult as we saw in the first section.)

What a Saga does is actually compose all those Effects together to implement the desired control flow. The simplest example is to sequence yielded Effects by just putting the yields one after another. You can also use the familiar control flow operators (`if` , `while` , `for`) to implement more sophisticated control flows.

We saw that using Effects like `call` and `put` , combined with high-level APIs like `takeEvery` allows us to achieve the same things as `redux-thunk` , but with the added benefit of easy testability.

But `redux-saga` provides another advantage over `redux-thunk` . In the Advanced section you'll encounter some more powerful Effects that let you express complex control flows while still allowing the same testability benefit.

Advanced

In this section, we'll dig into more powerful Effects provided by the library.

Pulling future actions

Until now we've used the helper effect `takeEvery` in order to spawn a new task on each incoming action. This mimics somewhat the behavior of redux-thunk: each time a Component, for example, invokes a `fetchProducts` Action Creator, the Action Creator will dispatch a thunk to execute the control flow.

In reality, `takeEvery` is just a wrapper effect for internal helper function built on top of the lower level and more powerful API. In this section we'll see a new Effect, `take`, which makes it possible to build complex control flow by allowing total control of the action observation process.

A simple logger

Let's take a simple example of a Saga that watches all actions dispatched to the store and logs them to the console.

Using `takeEvery('*')` (with the wildcard `*` pattern) we can catch all dispatched actions regardless of their types.

```
import { select, takeEvery } from 'redux-saga/effects'

function* watchAndLog() {
  yield takeEvery('*', function* logger(action) {
    const state = yield select()

    console.log('action', action)
    console.log('state after', state)
  })
}
```

Now let's see how to use the `take` Effect to implement the same flow as above

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

function* watchAndLog() {
  while (true) {
    const action = yield take('*')
    const state = yield select()

    console.log('action', action)
    console.log('state after', state)
  }
}
```

The `take` is just like `call` and `put` we saw earlier. It creates another command object that tells the middleware to wait for a specific action. The resulting behavior of the `call` Effect is the same as when the middleware suspends the Generator until a Promise resolves. In the `take` case it'll suspend the Generator until a matching action is dispatched. In the above example `watchAndLog` is suspended until any action is dispatched.

Note how we're running an endless loop `while (true)`. Remember this is a Generator function, which doesn't have a run-to-completion behavior. Our Generator will block on each iteration waiting for an action to happen.

Using `take` has a subtle impact on how we write our code. In the case of `takeEvery` the invoked tasks have no control on when they'll be called. They will be invoked again and again on each matching action. They also have no control on when to stop the observation.

In the case of `take` the control is inverted. Instead of the actions being *pushed* to the handler tasks, the Saga is *pulling* the action by itself. It looks as if the Saga is performing a normal function call `action = getNextAction()` which will resolve when the action is dispatched.

This inversion of control allows us to implement control flows that are non-trivial to do with the traditional *push* approach.

As a simple example, suppose that in our Todo application, we want to watch user actions and show a congratulation message after the user has created his first three todos.

```
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'})
}
```

Instead of a `while (true)` we're running a `for` loop which will iterate only three times. After taking the first three `TODO_CREATED` actions, `watchFirstThreeTodosCreation` will cause the application to display a congratulation message then terminate. This means the Generator will be garbage collected and no more observation will take place.

Another benefit of the pull approach is that we can describe our control flow using a familiar synchronous style. For example, suppose we want to implement a login flow with 2 actions `LOGIN` and `LOGOUT`. Using `takeEvery` (or `redux-thunk`) we'll have to write 2 separate tasks (or thunks): one for `LOGIN` and the other for `LOGOUT`.

The result is that our logic is now spread in 2 places. In order for someone reading our code to understand what's going on, he has to read the source of the 2 handlers and make the link between the logic in both. It means he has to rebuild the model of the flow in his head by rearranging mentally the logic placed in various places of the code in the correct order.

Using the pull model we can write our flow in the same place instead of handling the same action repeatedly.

```
function* loginFlow() {
  while (true) {
    yield take('LOGIN')
    // ... perform the login logic
    yield take('LOGOUT')
    // ... perform the logout logic
  }
}
```

The `loginFlow` Saga more clearly conveys the expected action sequence. It knows that the `LOGIN` action should always be followed by a `LOGOUT` action and that `LOGOUT` is always followed by a `LOGIN` (a good UI should always enforce a consistent order of the actions, by hiding or disabling unexpected action).

Non-blocking calls

In the previous section, we saw how the `take` Effect allows us to better describe a non-trivial flow in a central place.

Revisiting the login flow example:

```
function* loginFlow() {
  while (true) {
    yield take('LOGIN')
    // ... perform the login logic
    yield take('LOGOUT')
    // ... perform the logout logic
  }
}
```

Let's complete the example and implement the actual login/logout logic. Suppose we have an API which permits us to authorize the user on a remote server. If the authorization is successful, the server will return an authorization token which will be stored by our application using DOM storage (assume our API provides another service for DOM storage).

When the user logs out, we'll simply delete the authorization token stored previously.

First try

So far we have all needed Effects in order to implement the above flow. We can wait for specific actions in the store using the `take` Effect. We can make asynchronous calls using the `call` Effect. Finally, we can dispatch actions to the store using the `put` Effect.

So let's give it a try:

> Note: the code below has a subtle issue. Make sure to read the section until the end.

```
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) {
    yield put({type: 'LOGIN_ERROR', error})
  }
}

function* loginFlow() {
  while (true) {
    const {user, password} = yield take('LOGIN_REQUEST')
    const token = yield call(authorize, user, password)
    if (token) {
      yield call(Api.storeItem, {token})
      yield take('LOGOUT')
      yield call(Api.clearItem, 'token')
    }
  }
}
```

First we created a separate Generator `authorize` which will perform the actual API call and notify the Store upon success.

The `loginFlow` implements its entire flow inside a `while (true)` loop, which means once we reach the last step in the flow (`LOGOUT`) we start a new iteration by waiting for a new `LOGIN_REQUEST` action.

`loginFlow` first waits for a `LOGIN_REQUEST` action. Then retrieves the credentials in the action payload (`user` and `password`) and makes a `call` to the `authorize` task.

As you noted, `call` isn't only for invoking functions returning Promises. We can also use it to invoke other Generator functions. In the above example, **`loginFlow` will wait for authorize until it terminates and returns** (i.e. after performing the api call, dispatching the action and then returning the token to `loginFlow`).

If the API call succeeds, `authorize` will dispatch a `LOGIN_SUCCESS` action then return the fetched token. If it results in an error, it'll dispatch a `LOGIN_ERROR` action.

If the call to `authorize` is successful, `loginFlow` will store the returned token in the DOM storage and wait for a `LOGOUT` action. When the user logouts, we remove the stored token and wait for a new user login.

In the case of `authorize` failed, it'll return an undefined value, which will cause `loginFlow` to skip the previous process and wait for a new `LOGIN_REQUEST` action.

Observe how the entire logic is stored in one place. A new developer reading our code doesn't have to travel between various places in order to understand the control flow. It's like reading a synchronous algorithm: steps are laid out in their natural order. And we have functions which call other functions and wait for their results.

But there is still a subtle issue with the above approach

Suppose that when the `loginFlow` is waiting for the following call to resolve:

```
function* loginFlow() {
  while (true) {
    // ...
    try {
      const token = yield call(authorize, user, password)
      // ...
    }
    // ...
  }
}
```

The user clicks on the `Logout` button causing a `LOGOUT` action to be dispatched.

The following example illustrates the hypothetical sequence of the events:

```
UI                              loginFlow
--------------------------------------------------------
LOGIN_REQUEST...................call authorize.......... waiting to resolve
........................................................
........................................................
LOGOUT.................................................. missed!
........................................................
................................authorize returned...... dispatch a `LOGIN_SUCCESS`!!
........................................................
```

When `loginFlow` is blocked on the `authorize` call, an eventual `LOGOUT` occurring in between the call and the response will be missed, because `loginFlow` hasn't yet performed the `yield take('LOGOUT')`.

The problem with the above code is that `call` is a blocking Effect. i.e. the Generator can't perform/handle anything else until the call terminates. But in our case we do not only want `loginFlow` to execute the authorization call, but also watch for an eventual `LOGOUT` action that may occur in the middle of this call. That's because `LOGOUT` is *concurrent* to the `authorize` call.

So what's needed is some way to start `authorize` without blocking so `loginFlow` can continue and watch for an eventual/concurrent `LOGOUT` action.

To express non-blocking calls, the library provides another Effect: `fork`. When we fork a *task*, the task is started in the background and the caller can continue its flow without waiting for the forked task to terminate.

So in order for `loginFlow` to not miss a concurrent `LOGOUT`, we must not `call` the `authorize` task, instead we have to `fork` it.

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

function* loginFlow() {
  while (true) {
    ...
    try {
      // non-blocking call, what's the returned value here ?
      const ?? = yield fork(authorize, user, password)
      ...
    }
    ...
  }
}
```

The issue now is since our `authorize` action is started in the background, we can't get the `token` result (because we'd have to wait for it). So we need to move the token storage operation into the `authorize` task.

```
import { fork, call, take, 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})
    yield call(Api.storeItem, {token})
  } catch(error) {
    yield put({type: 'LOGIN_ERROR', error})
  }
}

function* loginFlow() {
  while (true) {
    const {user, password} = yield take('LOGIN_REQUEST')
    yield fork(authorize, user, password)
    yield take(['LOGOUT', 'LOGIN_ERROR'])
    yield call(Api.clearItem, 'token')
  }
}
```

We're also doing `yield take(['LOGOUT', 'LOGIN_ERROR'])`. It means we are watching for 2 concurrent actions:

- If the `authorize` task succeeds before the user logs out, it'll dispatch a `LOGIN_SUCCESS` action, then terminate. Our `loginFlow` saga will then wait only for a future `LOGOUT` action (because `LOGIN_ERROR` will never happen).
- If the `authorize` fails before the user logs out, it will dispatch a `LOGIN_ERROR` action, then terminate. So `loginFlow` will take the `LOGIN_ERROR` before the `LOGOUT` then it will enter in a another `while` iteration and will wait for the next `LOGIN_REQUEST` action.
- If the user logs out before the `authorize` terminate, then `loginFlow` will take a `LOGOUT` action and also wait for the next `LOGIN_REQUEST`.

Note the call for `Api.clearItem` is supposed to be idempotent. It'll have no effect if no token was stored by the `authorize` call. `loginFlow` makes sure no token will be in the storage before waiting for the next login.

But we're not yet done. If we take a `LOGOUT` in the middle of an API call, we have to **cancel** the `authorize` process, otherwise we'll have 2 concurrent tasks evolving in parallel: The `authorize` task will continue running and upon a successful (resp. failed) result, will dispatch a `LOGIN_SUCCESS` (resp. a `LOGIN_ERROR`) action leading to an inconsistent state.

In order to cancel a forked task, we use a dedicated Effect `cancel`

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

// ...

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')
  }
}
```

`yield fork` results in a Task Object. We assign the returned object into a local constant `task`. Later if we take a `LOGOUT` action, we pass that task to the `cancel` Effect. If the task is still running, it'll be aborted. If the task has already completed then nothing will happen and the cancellation will result in a no-op. And finally, if the task completed with an error, then we do nothing, because we know the task already completed.

We are *almost* done (concurrency is not that easy; you have to take it seriously).

Suppose that when we receive a `LOGIN_REQUEST` action, our reducer sets some `isLoginPending` flag to true so it can display some message or spinner in the UI. If we get a `LOGOUT` in the middle of an API call and abort the task by simply *killing it* (i.e. the task is stopped right away), then we may end up again with an inconsistent state. We'll still have `isLoginPending` set to true and our reducer will be waiting for an outcome action (`LOGIN_SUCCESS` or `LOGIN_ERROR`).

Fortunately, the `cancel` Effect won't brutally kill our `authorize` task, it'll instead give it a chance to perform its cleanup logic. The cancelled task can handle any cancellation logic (as well as any other type of completion) in its `finally` block. Since a finally block execute on any type of completion (normal return, error, or forced cancellation), there is an Effect `cancelled` which you can use if you want handle cancellation in a special way:

```
import { take, call, put, cancelled } 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})
    yield call(Api.storeItem, {token})
    return token
  } catch(error) {
    yield put({type: 'LOGIN_ERROR', error})
  } finally {
    if (yield cancelled()) {
      // ... put special cancellation handling code here
    }
  }
}
```

You may have noticed that we haven't done anything about clearing our `isLoginPending` state. For that, there are at least two possible solutions:

- dispatch a dedicated action `RESET_LOGIN_PENDING`
- more simply, make the reducer clear the `isLoginPending` on a `LOGOUT` action

Running Tasks In Parallel

The `yield` statement is great for representing asynchronous control flow in a simple and linear style, but we also need to do things in parallel. We can't simply write:

```
// wrong, effects will be executed in sequence
const users  = yield call(fetch, '/users'),
      repos = yield call(fetch, '/repos')
```

Because the 2nd effect will not get executed until the first call resolves. Instead we have to write:

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

// correct, effects will get executed in parallel
const [users, repos]  = yield [
  call(fetch, '/users'),
  call(fetch, '/repos')
]
```

When we yield an array of effects, the generator is blocked until all the effects are resolved or as soon as one is rejected (just like how `Promise.all` behaves).

Starting a race between multiple Effects

Sometimes we start multiple tasks in parallel but we don't want to wait for all of them, we just need to get the *winner*: the first one that resolves (or rejects). The `race` Effect offers a way of triggering a race between multiple Effects.

The following sample shows a task that triggers a remote fetch request, and constrains the response within a 1 second timeout.

```
import { race, take, put } from 'redux-saga/effects'
import { delay } from 'redux-saga'

function* fetchPostsWithTimeout() {
  const {posts, timeout} = yield race({
    posts: call(fetchApi, '/posts'),
    timeout: call(delay, 1000)
  })

  if (posts)
    put({type: 'POSTS_RECEIVED', posts})
  else
    put({type: 'TIMEOUT_ERROR'})
}
```

Another useful feature of `race` is that it automatically cancels the loser Effects. For example, suppose we have 2 UI buttons:

- The first starts a task in the background that runs in an endless loop `while (true)` (e.g. syncing some data with the server each x seconds).
- Once the background task is started, we enable a second button which will cancel the task

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

function* backgroundTask() {
  while (true) { ... }
}

function* watchStartBackgroundTask() {
  while (true) {
    yield take('START_BACKGROUND_TASK')
    yield race({
      task: call(backgroundTask),
      cancel: take('CANCEL_TASK')
    })
  }
}
```

In the case a `CANCEL_TASK` action is dispatched, the `race` Effect will automatically cancel `backgroundTask` by throwing a cancellation error inside it.

Sequencing Sagas via `yield*`

You can use the builtin `yield*` operator to compose multiple Sagas in a sequential way. This allows you to sequence your *macro-tasks* in a simple procedural style.

```
function* playLevelOne() { ... }

function* playLevelTwo() { ... }

function* playLevelThree() { ... }

function* game() {
  const score1 = yield* playLevelOne()
  yield put(showScore(score1))

  const score2 = yield* playLevelTwo()
  yield put(showScore(score2))

  const score3 = yield* playLevelThree()
  yield put(showScore(score3))
}
```

Note that using `yield*` will cause the JavaScript runtime to *spread* the whole sequence. The resulting iterator (from `game()`) will yield all values from the nested iterators. A more powerful alternative is to use the more generic middleware composition mechanism.

Composing Sagas

While using `yield*` provides an idiomatic way of composing Sagas, this approach has some limitations:

- You'll likely want to test nested generators separately. This leads to some duplication in the test code as well as the overhead of the duplicated execution. We don't want to execute a nested generator but only make sure the call to it was issued with the right argument.
- More importantly, `yield*` allows only for sequential composition of tasks, so you can only `yield*` to one generator at a time.

You can simply use `yield` to start one or more subtasks in parallel. When yielding a call to a generator, the Saga will wait for the generator to terminate before progressing, then resume with the returned value (or throws if an error propagates from the subtask).

```
function* fetchPosts() {
  yield put(actions.requestPosts())
  const products = yield call(fetchApi, '/products')
  yield put(actions.receivePosts(products))
}

function* watchFetch() {
  while (yield take(FETCH_POSTS)) {
    yield call(fetchPosts) // waits for the fetchPosts task to terminate
  }
}
```

Yielding to an array of nested generators will start all the sub-generators in parallel, wait for them to finish, then resume with all the results

```
function* mainSaga(getState) {
  const results = yield [call(task1), call(task2), ...]
  yield put(showResults(results))
}
```

In fact, yielding Sagas is no different than yielding other effects (future actions, timeouts, etc). This means you can combine those Sagas with all the other types using the effect combinators.

For example, you may want the user to finish some game in a limited amount of time:

```
function* game(getState) {
  let finished
  while (!finished) {
    // has to finish in 60 seconds
    const {score, timeout} = yield race({
      score: call(play, getState),
      timeout: call(delay, 60000)
    })

    if (!timeout) {
      finished = true
      yield put(showScore(score))
    }
  }
}
```

Task cancellation

We saw already an example of cancellation in the Non blocking calls section. In this section we'll review cancellation in more detail.

Once a task is forked, you can abort its execution using `yield cancel(task)`.

To see how it works, let's consider a simple example: A background sync which can be started/stopped by some UI commands. Upon receiving a `START_BACKGROUND_SYNC` action, we fork a background task that will periodically sync some data from a remote server.

The task will execute continually until a `STOP_BACKGROUND_SYNC` action is triggered. Then we cancel the background task and wait again for the next `START_BACKGROUND_SYNC` action.

```
import { take, put, call, fork, cancel, cancelled } from 'redux-saga/effects'
import { delay } from 'redux-saga'
import { someApi, actions } from 'somewhere'

function* bgSync() {
  try {
    while (true) {
      yield put(actions.requestStart())
      const result = yield call(someApi)
      yield put(actions.requestSuccess(result))
      yield call(delay, 5000)
    }
  } finally {
    if (yield cancelled())
      yield put(actions.requestFailure('Sync cancelled!'))
  }
}

function* main() {
  while ( yield take(START_BACKGROUND_SYNC) ) {
    // starts the task in the background
    const bgSyncTask = yield fork(bgSync)

    // wait for the user stop action
    yield take(STOP_BACKGROUND_SYNC)
    // user clicked stop. cancel the background task
    // this will cause the forked bgSync task to jump into its finally block
    yield cancel(bgSyncTask)
  }
}
```

In the above example, cancellation of `bgSyncTask` will cause the Generator to jump to the finally block. Here you can use `yield cancelled()` to check if the Generator has been cancelled or not.

Cancelling a running task will also cancel the current Effect where the task is blocked at the moment of cancellation.

For example, suppose that at a certain point in an application's lifetime, we have this pending call chain:

```
function* main() {
  const task = yield fork(subtask)
  ...
  // later
  yield cancel(task)
}

function* subtask() {
  ...
  yield call(subtask2) // currently blocked on this call
  ...
}

function* subtask2() {
  ...
  yield call(someApi) // currently blocked on this call
  ...
}
```

`yield cancel(task)` triggers a cancellation on `subtask`, which in turn triggers a cancellation on `subtask2`.

So we saw that Cancellation propagates downward (in contrast returned values and uncaught errors propagates upward). You can see it as a *contract* between the caller (which invokes the async operation) and the callee (the invoked operation). The callee is responsible for performing the operation. If it has completed (either success or error) the outcome propagates up to its caller and eventually to the caller of the caller and so on. That is, callees are responsible for *completing the flow*.

Now if the callee is still pending and the caller decides to cancel the operation, it triggers a kind of a signal that propagates down to the callee (and possibly to any deep operations called by the callee itself). All deeply pending operations will be cancelled.

There is another direction where the cancellation propagates to as well: the joiners of a task (those blocked on a `yield join(task)`) will also be cancelled if the joined task is cancelled. Similarly, any potential callers of those joiners will be cancelled as well (because they are blocked on an operation that has been cancelled from outside).

Testing generators with fork effect

When `fork` is called it starts the task in the background and also returns task object like we have learned previously. When testing this we have to use utility function `createMockTask`. Object returned from this function should be passed to next `next` call after fork test. Mock task can then be passed to `cancel` for example. Here is test for `main` generator which is on top of this page.

```
import { createMockTask } from 'redux-saga/utils';

describe('main', () => {
  const generator = main();

  it('waits for start action', () => {
    const expectedYield = take(START_BACKGROUND_SYNC);
    expect(generator.next().value).to.deep.equal(expectedYield);
  });

  it('forks the service', () => {
    const expectedYield = fork(bgSync);
    expect(generator.next().value).to.deep.equal(expectedYield);
  });

  it('waits for stop action and then cancels the service', () => {
    const mockTask = createMockTask();

    const expectedTakeYield = take(STOP_BACKGROUND_SYNC);
    expect(generator.next(mockTask).value).to.deep.equal(expectedTakeYield);

    const expectedCancelYield = cancel(mockTask);
    expect(generator.next().value).to.deep.equal(expectedCancelYield);
  });
});
```

You can also use mock task's functions `setRunning`, `setResult` and `setError` to set mock task's state. For example `mockTask.setRunning(false)`.

Note

It's important to remember that `yield cancel(task)` doesn't wait for the cancelled task to finish (i.e. to perform its finally block). The cancel effect behaves like fork. It returns as soon as the cancel was initiated. Once cancelled, a task should normally return as soon as it finishes its cleanup logic.

Automatic cancellation

Besides manual cancellation there are cases where cancellation is triggered automatically

1. In a `race` effect. All race competitors, except the winner, are automatically cancelled.
2. In a parallel effect (`yield [...]`). The parallel effect is rejected as soon as one of the sub-effects is rejected (as implied by `Promise.all`). In this case, all the other sub-effects are automatically cancelled.

redux-saga's fork model

In `redux-saga` you can dynamically fork tasks that execute in the background using 2 Effects

- `fork` is used to create *attached forks*
- `spawn` is used to create *detached forks*

Attached forks (using `fork`)

Attached forks remains attached to their parent by the following rules

Completion

- A Saga terminates only after
  - It terminates its own body of instructions
  - All attached forks are themselves terminated

For example say we have the following

```
import { delay } from 'redux-saga'
import { fork, call, put } from 'redux-saga/effects'
import api from './somewhere/api' // app specific
import { receiveData } from './somewhere/actions' // app specific

function* fetchAll() {
  const task1 = yield fork(fetchResource, 'users')
  const task2 = yield fork(fetchResource, 'comments')
  yield call(delay, 1000)
}

function* fetchResource(resource) {
  const {data} = yield call(api.fetch, resource)
  yield put(receiveData(data))
}

function* main() {
  yield call(fetchAll)
}
```

`call(fetchAll)` will terminate after:

- The `fetchAll` body itself terminates, this means all 3 effects are performed. Since `fork` effects are non blocking, the task will block on `call(delay, 1000)`
- The 2 forked tasks terminate, i.e. after fetching the required resources and putting the corresponding `receiveData` actions

So the whole task will block until a delay of 1000 millisecond passed *and* both `task1` and `task2` finished their business.

Say for example, the delay of 1000 milliseconds elapsed and the 2 tasks hasn't yet finished, then `fetchAll` will still wait for all forked tasks to finish before terminating the whole task.

The attentive reader might have noticed the `fetchAll` saga could be rewritten using the parallel Effect

```
function* fetchAll() {
  yield [
    call(fetchResource, 'users'),     // task1
    call(fetchResource, 'comments'),  // task2,
    call(delay, 1000)
  ]
}
```

In fact, attached forks shares the same semantics with the parallel Effect:

- We're executing tasks in parallel
- The parent will terminate after all launched tasks terminate

And this applies for all other semantics as well (error and cancellation propagation). You can understand how attached forks behave by simply considering it as a *dynamic parallel* Effect.

Error propagation

Following the same analogy, Let's examine in detail how errors are handled in parallel Effects

for example, let's say we have this Effect

```
yield [
  call(fetchResource, 'users'),
  call(fetchResource, 'comments'),
  call(delay, 1000)
]
```

The above effect will fail as soon as any one of the 3 child Effects fails. Furthermore, the uncaught error will cause the parallel Effect to cancel all the other pending Effects. So for example if `call(fetchResource), 'users')` raises an uncaught error, the parallel Effect will cancel the 2 other tasks (if they are still pending) then aborts itself with the same error from the failed call.

Similarly for attached forks, a Saga aborts as soon as

- Its main body of instructions throws an error
- An uncaught error was raised by one of its attached forks

So in the previous example

```
//... imports

function* fetchAll() {
  const task1 = yield fork(fetchResource, 'users')
  const task2 = yield fork(fetchResource, 'comments')
  yield call(delay, 1000)
}

function* fetchResource(resource) {
  const {data} = yield call(api.fetch, resource)
  yield put(receiveData(data))
}

function* main() {
  try {
    yield call(fetchAll)
  } catch (e) {
    // handle fetchAll errors
  }
}
```

If at a moment, for example, `fetchAll` is blocked on the `call(delay, 1000)` Effect, and say, `task1` failed, then the whole `fetchAll` task will fail causing

- Cancellation of all other pending tasks. This includes:
  - The *main task* (the body of `fetchAll`): cancelling it means cancelling the current Effect `call(delay, 1000)`
  - The other forked tasks which are still pending. i.e. `task2` in our example.
- The `call(fetchAll)` will raise itself an error which will be caught in the `catch` body of `main`

Note we're able to catch the error from `call(fetchAll)` inside `main` only because we're using a blocking call. And that we can't catch the error directly from `fetchAll`. This a rule of thumb, **you can't catch errors from forked tasks**. A failure in an attached fork will cause the forking parent to abort (Just like there is no way to catch an error *inside* a parallel Effect, only from outside by blocking on the parallel Effect).

Cancellation

Cancelling a Saga causes the cancellation of:

- The *main task* this means cancelling the current Effect where the Saga is blocked
- All attached forks that are still executing

WIP

Detached forks (using `spawn`)

Detached forks live in their own execution context. A parent doesn't wait for detached forks to terminate. Uncaught errors from spawned tasks are not bubbled up to the parent. And cancelling a parent doesn't automatically cancel detached forks (you need to cancel them explicitly).

In short, detached forks behave like root Sagas started directly using the `middleware.run` API.

WIP

Concurrency

In the basics section, we saw how to use the helper effects `takeEvery` and `takeLatest` in order to manage concurrency between Effects.

In this section we'll see how those helpers could be implemented using the low-level Effects.

`takeEvery`

```
function* takeEvery(pattern, saga, ...args) {
  const task = yield fork(function* () {
    while (true) {
      const action = yield take(pattern)
      yield fork(saga, ...args.concat(action))
    }
  })
  return task
}
```

`takeEvery` allows multiple `saga` tasks to be forked concurrently.

`takeLatest`

```
function* takeLatest(pattern, saga, ...args) {
  const task = yield fork(function* () {
    let lastTask
    while (true) {
      const action = yield take(pattern)
      if (lastTask)
        yield cancel(lastTask) // cancel is no-op if the task has already terminated

      lastTask = yield fork(saga, ...args.concat(action))
    }
  })
  return task
}
```

`takeLatest` doesn't allow multiple Saga tasks to be fired concurrently. As soon as it gets a new dispatched action, it cancels any previously-forked task (if still running).

`takeLatest` can be useful to handle AJAX requests where we want to only have the response to the latest request.

Testing Sagas

Effects return plain javascript objects

Those objects describe the effect and redux-saga is in charge to execute them.

This makes testing very easy because all you have to do is compare that the object yielded by the saga describe the effect you want.

Basic Example

```
console.log(put({ type: MY_CRAZY_ACTION }));

/*
{
  @@redux-saga/IO': true,
  PUT: {
    channel: null,
    action: {
      type: 'MY_CRAZY_ACTION'
    }
  }
}
 */
```

Testing a saga that wait for a user action and dispatch

```
const CHOOSE_COLOR = 'CHOOSE_COLOR';
const CHANGE_UI = 'CHANGE_UI';

const chooseColor = (color) => ({
  type: CHOOSE_COLOR,
  payload: {
    color,
  },
});

const changeUI = (color) => ({
  type: CHANGE_UI,
  payload: {
    color,
  },
});


function* changeColorSaga() {
  const action = yield take(CHOOSE_COLOR);
  yield put(changeUI(action.payload.color));
}

test('change color saga', assert => {
  const gen = changeColorSaga();

  assert.deepEqual(
    gen.next().value,
    take(CHOOSE_COLOR),
    'it should wait for a user to choose a color'
  );

  const color = 'red';
  assert.deepEqual(
    gen.next(chooseColor(color)).value,
    put(changeUI(color)),
    'it should dispatch an action to change the ui'
  );

  assert.deepEqual(
    gen.next().done,
    true,
    'it should be done'
  );

  assert.end();
});
```

Another great benefit is that your tests are also your doc! They describe everything that should happen.

Branching Saga

Sometimes your saga will have different outcomes. To test the different branches without repeating all the steps that lead to it you can use the utility function **cloneableGenerator**

```
const CHOOSE_NUMBER = 'CHOOSE_NUMBER';
const CHANGE_UI = 'CHANGE_UI';
const DO_STUFF = 'DO_STUFF';

const chooseNumber = (number) => ({
  type: CHOOSE_NUMBER,
  payload: {
    number,
  },
});

const changeUI = (color) => ({
  type: CHANGE_UI,
  payload: {
    color,
  },
});

const doStuff = () => ({
  type: DO_STUFF,
});


function* doStuffThenChangeColor() {
  yield put(doStuff());
  yield put(doStuff());
  const action = yield take(CHOOSE_NUMBER);
  if (action.payload.number % 2 === 0) {
    yield put(changeUI('red'));
  } else {
    yield put(changeUI('blue'));
  }
}

import { put, take } from 'redux-saga/effects';
import { cloneableGenerator } from 'redux-saga/utils';

test('doStuffThenChangeColor', assert => {
  const data = {};
  data.gen = cloneableGenerator(doStuffThenChangeColor)();

  assert.deepEqual(
    data.gen.next().value,
    put(doStuff()),
    'it should do stuff'
  );
  assert.deepEqual(
    data.gen.next().value,
    put(doStuff()),
    'it should do stuff'
  );

  assert.deepEqual(
    data.gen.next().value,
    take(CHOOSE_NUMBER),
    'should wait for the user to give a number'
  );

  assert.test('user choose an even number', a => {
    // cloning the generator before sending data
    data.clone = data.gen.clone();
    a.deepEqual(
      data.gen.next(chooseNumber(2)).value,
      put(changeUI('red')),
      'should change the color to red'
    );

    a.equal(
      data.gen.next().done,
      true,
      'it should be done'
    );

    a.end();
  });

  assert.test('user choose an odd number', a => {
    a.deepEqual(
      data.clone.next(chooseNumber(3)).value,
      put(changeUI('blue')),
      'should change the color to blue'
    );

    a.equal(
      data.clone.next().done,
      true,
      'it should be done'
    );

    a.end();
  });
});
```

See also: Task cancellation for testing fork effects

See also: Repository Examples:

https://github.com/redux-saga/redux-saga/blob/master/examples/counter/test/sagas.js

https://github.com/redux-saga/redux-saga/blob/master/examples/shopping-cart/test/sagas.js

Connecting Sagas to external Input/Output

We saw that `take` Effects are resolved by waiting for actions to be dispatched to the Store. And that `put` Effects are resolved by dispatching the actions provided as argument.

When a Saga is started (either at startup or later dynamically), the middleware automatically connects its `take` / `put` to the store. The 2 Effects can be seen as a sort of Input/Output to the Saga.

`redux-saga` provides a way to run a Saga outside of the Redux middleware environment and connect it to a custom Input/Output.

```
import { runSaga } from 'redux-saga'

function* saga() { ... }

const myIO = {
  subscribe: ..., // this will be used to resolve take Effects
  dispatch: ...,  // this will be used to resolve put Effects
  getState: ...,  // this will be used to resolve select Effects
}

runSaga(
  saga(),
  myIO
)
```

For more info, see the API docs.

Using Channels

Until now we've used the `take` and `put` effects to communicate with the Redux Store. Channels generalize those Effects to communicate with external event sources or between Sagas themselves. They can also be used to queue specific actions from the Store.

In this section, we'll see:

- How to use the `yield actionChannel` Effect to buffer specific actions from the Store.
- How to use the `eventChannel` factory function to connect `take` Effects to external event sources.
- How to create a channel using the generic `channel` factory function and use it in `take` / `put` Effects to communicate between two Sagas.

Using the `actionChannel` Effect

Let's review the canonical example:

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

function* watchRequests() {
  while (true) {
    const {payload} = yield take('REQUEST')
    yield fork(handleRequest, payload)
  }
}

function* handleRequest(payload) { ... }
```

The above example illustrates the typical *watch-and-fork* pattern. The `watchRequests` saga is using `fork` to avoid blocking and thus not missing any action from the store. A `handleRequest` task is created on each `REQUEST` action. So if there are many actions fired at a rapid rate there can be many `handleRequest` tasks executing concurrently.

Imagine now that our requirement is as follows: we want to process `REQUEST` serially. If we have at any moment four actions, we want to handle the first `REQUEST` action, then only after finishing this action we process the second action and so on...

So we want to *queue* all non-processed actions, and once we're done with processing the current request, we get the next message from the queue.

Redux-Saga provides a little helper Effect `actionChannel`, which can handle this for us. Let's see how we can rewrite the previous example with it:

```
import { take, actionChannel, call, ... } from 'redux-saga/effects'

function* watchRequests() {
  // 1- Create a channel for request actions
  const requestChan = yield actionChannel('REQUEST')
  while (true) {
    // 2- take from the channel
    const {payload} = yield take(requestChan)
    // 3- Note that we're using a blocking call
    yield call(handleRequest, payload)
  }
}

function* handleRequest(payload) { ... }
```

The first thing is to create the action channel. We use `yield actionChannel(pattern)` where pattern is interpreted using the same rules we mentioned previously with `take(pattern)`. The difference between the 2 forms is that `actionChannel` **can buffer incoming messages** if the Saga is not yet ready to take them (e.g. blocked on an API call).

Next is the `yield take(requestChan)`. Besides usage with a `pattern` to take specific actions from the Redux Store, `take` can also be used with channels (above we created a channel object from specific Redux actions). The `take` will block the Saga until a message is available on the channel. The take may also resume immediately if there is a message stored in the underlying buffer.

The important thing to note is how we're using a blocking `call`. The Saga will remain blocked until `call(handleRequest)` returns. But meanwhile, if other `REQUEST` actions are dispatched while the Saga is still blocked, they will queued internally by `requestChan`. When the Saga resumes from `call(handleRequest)` and executes the next `yield take(requestChan)`, the take will resolve with the queued message.

By default, `actionChannel` buffers all incoming messages without limit. If you want a more control over the buffering, you can supply a Buffer argument to the effect creator. Redux-Saga provides some common buffers (none, dropping, sliding) but you can also supply your own buffer implementation. See API docs for more details.

For example if you want to handle only the most recent five items you can use:

```
import { buffers } from 'redux-saga'
import { actionChannel } from 'redux-saga/effects'

function* watchRequests() {
  const requestChan = yield actionChannel('REQUEST', buffers.sliding(5))
  ...
}
```

Using the `eventChannel` factory to connect to external events

Like `actionChannel` (Effect), `eventChannel` (a factory function, not an Effect) creates a Channel for events but from event sources other than the Redux Store.

This simple example creates a Channel from an interval:

```
import { eventChannel, END } from 'redux-saga'

function countdown(secs) {
  return eventChannel(emitter => {
      const iv = setInterval(() => {
        secs -= 1
        if (secs > 0) {
          emitter(secs)
        } else {
          // this causes the channel to close
          emitter(END)
        }
      }, 1000);
      // The subscriber must return an unsubscribe function
      return () => {
        clearInterval(iv)
      }
    }
  )
}
```

The first argument in `eventChannel` is a *subscriber* function. The role of the subscriber is to initialize the external event source (above using `setInterval`), then routes all incoming events from the source to the channel by invoking the supplied `emitter`. In the above example we're invoking `emitter` on each second.

> Note: You need to sanitize your event sources as to not pass null or undefined through the event channel. While it's fine to pass numbers through, we'd recommend structuring your event channel data like your redux actions. `{ number }` over `number`.

Note also the invocation `emitter(END)`. We use this to notify any channel consumer that the channel has been closed, meaning no other messages will come through this channel.

Let's see how we can use this channel from our Saga. (This is taken from the cancellable-counter example in the repo.)

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

// creates an event Channel from an interval of seconds
function countdown(seconds) { ... }

export function* saga() {
  const chan = yield call(countdown, value)
  try {
    while (true) {
      // take(END) will cause the saga to terminate by jumping to the finally block
      let seconds = yield take(chan)
      console.log(`countdown: ${seconds}`)
    }
  } finally {
    console.log('countdown terminated')
  }
}
```

So the Saga is yielding a `take(chan)`. This causes the Saga to block until a message is put on the channel. In our example above, it corresponds to when we invoke `emitter(secs)`. Note also we're executing the whole `while (true) {...}` loop inside a `try/finally` block. When the interval terminates, the countdown function closes the event channel by invoking `emitter(END)`. Closing a channel has the effect of terminating all Sagas blocked on a `take` from that channel. In our example, terminating the Saga will cause it to jump to its `finally` block (if provided, otherwise the Saga simply terminates).

The subscriber returns an `unsubscribe` function. This is used by the channel to unsubscribe before the event source complete. Inside a Saga consuming messages from an event channel, if we want to *exit early* before the event source complete (e.g. Saga has been cancelled) you can call `chan.close()` to close the channel and unsubscribe from the source.

For example, we can make our Saga support cancellation:

```
import { take, put, call, cancelled } from 'redux-saga/effects'
import { eventChannel, END } from 'redux-saga'

// creates an event Channel from an interval of seconds
function countdown(seconds) { ... }

export function* saga() {
  const chan = yield call(countdown, value)
  try {
    while (true) {
      let seconds = yield take(chan)
      console.log(`countdown: ${seconds}`)
    }
  } finally {
    if (yield cancelled()) {
      chan.close()
      console.log('countdown cancelled')
    }
  }
}
```

Here is another example of how you can use event channels to pass WebSocket events into your saga (e.g.: using socket.io library). Suppose you are waiting for a server message `ping` then reply with a `pong` message after some delay.

```
import { take, put, call, apply } from 'redux-saga/effects'
import { eventChannel, delay } from 'redux-saga'
import { createWebSocketConnection } from './socketConnection'

// this function creates an event channel from a given socket
// Setup subscription to incoming `ping` events
function createSocketChannel(socket) {
  // `eventChannel` takes a subscriber function
  // the subscriber function takes an `emit` argument to put messages onto the channel
  return eventChannel(emit => {

    const pingHandler = (event) => {
      // puts event payload into the channel
      // this allows a Saga to take this payload from the returned channel
      emit(event.payload)
    }

    // setup the subscription
    socket.on('ping', pingHandler)

    // the subscriber must return an unsubscribe function
    // this will be invoked when the saga calls `channel.close` method
    const unsubscribe = () => {
      socket.off('ping', pingHandler)
    }

    return unsubscribe
  })
}

// reply with a `pong` message by invoking `socket.emit('pong')`
function* pong(socket) {
  yield call(delay, 5000)
  yield apply(socket, socket.emit, ['pong']) // call `emit` as a method with `socket`
as context
}

export function* watchOnPings() {
  const socket = yield call(createWebSocketConnection)
  const socketChannel = yield call(createSocketChannel, socket)

  while (true) {
    const payload = yield take(socketChannel)
    yield put({ type: INCOMING_PONG_PAYLOAD, payload })
    yield fork(pong, socket)
  }
}
```

> Note: messages on an eventChannel are not buffered by default. You have to provide a buffer to the eventChannel factory in order to specify buffering strategy for the channel (e.g. `eventChannel(subscriber, buffer)`). See the API docs for more info.

Using channels to communicate between Sagas

Besides action channels and event channels. You can also directly create channels which are not connected to any source by default. You can then manually `put` on the channel. This is handy when you want to use a channel to communicate between sagas.

To illustrate, let's review the former example of request handling.

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

function* watchRequests() {
  while (true) {
    const {payload} = yield take('REQUEST')
    yield fork(handleRequest, payload)
  }
}

function* handleRequest(payload) { ... }
```

We saw that the watch-and-fork pattern allows handling multiple requests simultaneously, without limit on the number of worker tasks executing concurrently. Then, we used the `actionChannel` effect to limit the concurrency to one task at a time.

So let's say that our requirement is to have a maximum of three tasks executing at the same time. When we get a request and there are less than three tasks executing, we process the request immediately, otherwise we queue the task and wait for one of the three *slots* to become free.

Below is an example of a solution using channels:

```
import { channel } from 'redux-saga'
import { take, fork, ... } from 'redux-saga/effects'

function* watchRequests() {
  // create a channel to queue incoming requests
  const chan = yield call(channel)

  // create 3 worker 'threads'
  for (var i = 0; i < 3; i++) {
    yield fork(handleRequest, chan)
  }

  while (true) {
    const {payload} = yield take('REQUEST')
    yield put(chan, payload)
  }
}

function* handleRequest(chan) {
  while (true) {
    const payload = yield take(chan)
    // process the request
  }
}
```

In the above example, we create a channel using the `channel` factory. We get back a channel which by default buffers all messages we put on it (unless there is a pending taker, in which the taker is resumed immediately with the message).

The `watchRequests` saga then forks three worker sagas. Note the created channel is supplied to all forked sagas. `watchRequests` will use this channel to *dispatch* work to the three worker sagas. On each `REQUEST` action the Saga will simply put the payload on the channel. The payload will then be taken by any *free* worker. Otherwise it will be queued by the channel until a worker Saga is ready to take it.

All the three workers run a typical while loop. On each iteration, a worker will take the next request, or will block until a message is available. Note that this mechanism provides an automatic load-balancing between the 3 workers. Rapid workers are not slowed down by slow workers.

Recipes

Throttling

You can throttle a sequence of dispatched actions by using a handy built-in `throttle` helper. For example, suppose the UI fires an `INPUT_CHANGED` action while the user is typing in a text field.

```
import { throttle } from 'redux-saga/effects'

function* handleInput(input) {
  // ...
}

function* watchInput() {
  yield throttle(500, 'INPUT_CHANGED', handleInput)
}
```

By using this helper the `watchInput` won't start a new `handleInput` task for 500ms, but in the same time it will still be accepting the latest `INPUT_CHANGED` actions into its underlaying `buffer`, so it'll miss all `INPUT_CHANGED` actions happening in-between. This ensures that the Saga will take at most one `INPUT_CHANGED` action during each period of 500ms and still be able to process trailing action.

Debouncing

To debounce a sequence, put the built-in `delay` helper in the forked task:

```
import { delay } from 'redux-saga'

function* handleInput(input) {
  // debounce by 500ms
  yield call(delay, 500)
  ...
}

function* watchInput() {
  let task
  while (true) {
    const { input } = yield take('INPUT_CHANGED')
    if (task) {
      yield cancel(task)
    }
    task = yield fork(handleInput, input)
  }
}
```

The `delay` function implements a simple debounce using a Promise.

```
const delay = (ms) => new Promise(resolve => setTimeout(resolve, ms))
```

In the above example `handleInput` waits for 500ms before performing its logic. If the user types something during this period we'll get more `INPUT_CHANGED` actions. Since `handleInput` will still be blocked in the `delay` call, it'll be cancelled by `watchInput` before it can start performing its logic.

Example above could be rewritten with redux-saga `takeLatest` helper:

```
import { delay } from 'redux-saga'

function* handleInput({ input }) {
  // debounce by 500ms
  yield call(delay, 500)
  ...
}

function* watchInput() {
  // will cancel current running handleInput task
  yield takeLatest('INPUT_CHANGED', handleInput);
}
```

Retrying XHR calls

To retry a XHR call for a specific amount of times, use a for loop with a delay:

```
import { delay } from 'redux-saga'

function* updateApi(data) {
  for(let i = 0; i < 5; i++) {
    try {
      const apiResponse = yield call(apiRequest, { data });
      return apiResponse;
    } catch(err) {
      if(i < 5) {
        yield call(delay, 2000);
      }
    }
  }
  // attempts failed after 5x2secs
  throw new Error('API request failed');
}

export default function* updateResource() {
  while (true) {
    const { data } = yield take('UPDATE_START');
    try {
      const apiResponse = yield call(updateApi, data);
      yield put({
        type: 'UPDATE_SUCCESS',
        payload: apiResponse.body,
      });
    } catch (error) {
      yield put({
        type: 'UPDATE_ERROR',
        error
      });
    }
  }
}
```

In the above example the `apiRequest` will be retried for 5 times, with a delay of 2 seconds in between. After the 5th failure, the exception thrown will get caught by the parent saga, which will dispatch the `UPDATE_ERROR` action.

If you want unlimited retries, then the `for` loop can be replaced with a `while (true)`. Also instead of `take` you can use `takeLatest`, so only the last request will be retried. By adding an `UPDATE_RETRY` action in the error handling, we can inform the user that the update was not successfull but it will be retried.

```
import { delay } from 'redux-saga'

function* updateApi(data) {
  while (true) {
    try {
      const apiResponse = yield call(apiRequest, { data });
      return apiResponse;
    } catch(error) {
      yield put({
        type: 'UPDATE_RETRY',
        error
      })
      yield call(delay, 2000);
    }
  }
}

function* updateResource({ data }) {
  const apiResponse = yield call(updateApi, data);
  yield put({
    type: 'UPDATE_SUCCESS',
    payload: apiResponse.body,
  });
}

export function* watchUpdateResource() {
  yield takeLatest('UPDATE_START', updateResource);
}
```

Undo

The ability to undo respects the user by allowing the action to happen smoothly first and foremost before assuming they don't know what they are doing. GoodUI The redux documentation describes a robust way to implement an undo based on modifying the reducer to contain `past` , `present` , and `future` state. There is even a library redux-undo that creates a higher order reducer to do most of the heavy lifting for the developer.

However, this method comes with it overheard from storing references to the previous state(s) of the application.

Using redux-saga's `delay` and `race` we can implement a simple, one-time undo without enhancing our reducer or storing the previous state.

```
import { take, put, call, spawn, race } from 'redux-saga/effects'
import { delay } from 'redux-saga'
import { updateThreadApi, actions } from 'somewhere'

function* onArchive(action) {

  const { threadId } = action
  const undoId = `UNDO_ARCHIVE_${threadId}`

  const thread = { id: threadId, archived: true }

  // show undo UI element, and provide a key to communicate
  yield put(actions.showUndo(undoId))

  // optimistically mark the thread as `archived`
  yield put(actions.updateThread(thread))

  // allow the user 5 seconds to perform undo.
  // after 5 seconds, 'archive' will be the winner of the race-condition
  const { undo, archive } = yield race({
    undo: take(action => action.type === 'UNDO' && action.undoId === undoId),
    archive: call(delay, 5000)
  })

  // hide undo UI element, the race condition has an answer
  yield put(actions.hideUndo(undoId))

  if (undo) {
    // revert thread to previous state
    yield put(actions.updateThread({ id: threadId, archived: false }))
  } else if (archive) {
    // make the API call to apply the changes remotely
    yield call(updateThreadApi, thread)
  }
}

function* main() {
  while (true) {
    // wait for an ARCHIVE_THREAD to happen
    const action = yield take('ARCHIVE_THREAD')
    // use spawn to execute onArchive in a non-blocking fashion, which also
    // prevents cancelation when main saga gets cancelled.
    // This helps us in keeping state in sync between server and client
    yield spawn(onArchive, action)
  }
}
```

External Resources

Articles on Generators

- The Definitive Guide to the JavaScript Generators by Gajus Kuizinas
- The Basics Of ES6 Generators by Kyle Simpson
- ES6 generators in depth by Axel Rauschmayer

Articles on redux-saga

- Redux nowadays: From actions creators to sagas by Riad Benguella
- Managing Side Effects In React + Redux Using Sagas by Jack Hsu
- Using redux-saga To Simplify Your Growing React Native Codebase by Steve Kellock
- Master Complex Redux Workflows with Sagas by Brandon Konkle
- Handling async in Redux with Sagas by Niels Gerritsen
- Tips to handle Authentication in Redux by Mattia Manzati
- Build an Image Gallery Using React, Redux and redux-saga by Joel Hooks
- Async Operations using redux saga by Andrés Mijares
- Introduction to Redux Saga by Matt Granmoe
- Vuex meets Redux-saga by Illya Klymov

Addons

- redux-saga-sc – Provides sagas to easily dispatch redux actions over SocketCluster websockets
- redux-form-saga – An action creator and saga for integrating Redux Form and Redux Saga
- redux-electron-enhancer – Redux store which synchronizes between instances in multiple process
- eslint-plugin-redux-saga - ESLint rules that help you to write error free sagas
- redux-saga-router - Helper for running sagas in response to route changes.
- vuex-redux-saga - Bridge between Vuex and Redux-Saga

Troubleshooting

App freezes after adding a saga

Make sure that you `yield` the effects from the generator function.

Consider this example:

```
import { take } from 'redux-saga/effects'

function* logActions() {
  while (true) {
    const action = take() // wrong
    console.log(action)
  }
}
```

It will put the application into an infinite loop because `take()` only creates a description of the effect. Unless you `yield` it for the middleware to execute, the `while` loop will behave like a regular `while` loop, and freeze your application.

Adding `yield` will pause the generator and return control to the Redux Saga middleware which will execute the effect. In case of `take()`, Redux Saga will wait for the next action matching the pattern, and only then will resume the generator.

To fix the example above, simply `yield` the effect returned by `take()`:

```
import { take } from 'redux-saga/effects'

function* logActions() {
  while (true) {
    const action = yield take() // correct
    console.log(action)
  }
}
```

My Saga is missing dispatched actions

Make sure the Saga is not blocked on some effect. When a Saga is waiting for an Effect to resolve, it will not be able to take dispatched actions until the Effect is resolved.

For example, consider this example

```
function watchRequestActions() {
  while (true) {
    const {url, params} = yield take('REQUEST')
    yield call(handleRequestAction, url, params) // The Saga will block here
  }
}

function handleRequestAction(url, params) {
  const response = yield call(someRemoteApi, url, params)
  yield put(someAction(response))
}
```

When `watchRequestActions` performs `yield call(handleRequestAction, url, params)`, it'll wait for `handleRequestAction` until it terminates an returns before continuing on the next `yield take`. For example suppose we have this sequence of events

```
UI                      watchRequestActions             handleRequestAction
-----------------------------------------------------------------------------
.......................take('REQUEST').......................................
dispatch(REQUEST)......call(handleRequestAction).......call(someRemoteApi)... Wait ser
ver resp.
.............................................................................
.............................................................................
dispatch(REQUEST)............................................................ Action m
issed!!
.............................................................................
.............................................................................
.......................................................put(someAction).......
.......................take('REQUEST')....................................... saga is
resumed
```

As illustrated above, when a Saga is blocked on a **blocking call** then it will miss all the actions dispatched in-between.

To avoid blocking the Saga, you can use a **non-blocking call** using `fork` instead of `call`

```
function watchRequestActions() {
  while (true) {
    const {url, params} = yield take('REQUEST')
    yield fork(handleRequestAction, url, params) // The Saga will resume immediately
  }
}
```

Glossary

This is a glossary of the core terms in Redux Saga.

Effect

An effect is a plain JavaScript Object containing some instructions to be executed by the saga middleware.

You create effects using factory functions provided by the redux-saga library. For example you use `call(myfunc, 'arg1', 'arg2')` to instruct the middleware to invoke `myfunc('arg1', 'arg2')` and return the result back to the Generator that yielded the effect

Task

A task is like a process running in background. In a redux-saga based application there can be multiple tasks running in parallel. You create tasks by using the `fork` function

```
function* saga() {
  ...
  const task = yield fork(otherSaga, ...args)
  ...
}
```

Blocking/Non-blocking call

A Blocking call means that the Saga yielded an Effect and will wait for the outcome of its execution before resuming to the next instruction inside the yielding Generator.

A Non-blocking call means that the Saga will resume immediately after yielding the Effect.

For example

```
function* saga() {
  yield take(ACTION)              // Blocking: will wait for the action
  yield call(ApiFn, ...args)      // Blocking: will wait for ApiFn (If ApiFn returns a
 Promise)
  yield call(otherSaga, ...args)  // Blocking: will wait for otherSaga to terminate

  yield put(...)                   // Non-Blocking: will dispatch within internal sche
duler

  const task = yield fork(otherSaga, ...args)  // Non-blocking: will not wait for othe
rSaga
  yield cancel(task)                           // Non-blocking: will resume immediately

  // or
  yield join(task)                              // Blocking: will wait for the task to
 terminate
}
```

Watcher/Worker

refers to a way of organizing the control flow using two separate Sagas

- The watcher: will watch for dispatched actions and fork a worker on every action
- The worker: will handle the action and terminate

example

```
function* watcher() {
  while (true) {
    const action = yield take(ACTION)
    yield fork(worker, action.payload)
  }
}

function* worker(payload) {
  // ... do some stuff
}
```

API Reference

Cheatsheets

Middleware API

`createSagaMiddleware(options)`

Creates a Redux middleware and connects the Sagas to the Redux Store

- `options: Object` - A list of options to pass to the middleware. Currently supported options are:

  - `sagaMonitor` : SagaMonitor - If a Saga Monitor is provided, the middleware will deliver monitoring events to the monitor.

  - `emitter` : Used to feed actions from redux to redux-saga (through redux middleware). Emitter is a higher order function, which takes a builtin emitter and returns another emitter.

    Example

    In the following example we create an emitter which "unpacks" array of actions and emits individual actions extracted from the array.

```
createSagaMiddleware({
  emitter: emit => action => {
   if (Array.isArray(action)) {
     action.forEach(emit);
     return
   }
   emit(action);
  }
});
```

- `logger` : Function - defines a custom logger for the middleware. By default, the middleware logs all errors and warnings to the console. This option tells the middleware to send errors/warnings to the provided logger instead. The logger is called with the params `(level, ...args)` . The 1st indicates the level of the log ('info', 'warning' or 'error'). The rest corresponds to the following arguments (You can use `args.join(' ') to concatenate all args into a single StringS`).
- `onError` : Function - if provided, the middleware will call it with uncaught errors from Sagas. useful for sending uncaught exceptions to error tracking services.

Example

Below we will create a function `configureStore` which will enhance the Store with a new method `runSaga` . Then in our main module, we will use the method to start the root Saga of the application.

configureStore.js

```
import createSagaMiddleware from 'redux-saga'
import reducer from './path/to/reducer'

export default function configureStore(initialState) {
  // Note: passing middleware as the last argument to createStore requires redux@>=3.1
.0
  const sagaMiddleware = createSagaMiddleware()
  return {
    ...createStore(reducer, initialState, applyMiddleware(/* other middleware, */sagaM
iddleware)),
    runSaga: sagaMiddleware.run
  }
}
```

main.js

```
import configureStore from './configureStore'
import rootSaga from './sagas'
// ... other imports

const store = configureStore()
store.runSaga(rootSaga)
```

Notes

See below for more information on the `sagaMiddleware.run` method.

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

Dynamically run `saga`. Can be used to run Sagas **only after** the `applyMiddleware` phase.

- `saga: Function` : a Generator function
- `args: Array<any>` : arguments to be provided to `saga`

The method returns a Task descriptor.

Notes

`saga` must be a function which returns a Generator Object. The middleware will then iterate over the Generator and execute all yielded Effects.

`saga` may also start other sagas using the various Effects provided by the library. The iteration process process described below is also applied to all child sagas.

In the first iteration, the middleware invokes the `next()` method to retrieve the next Effect. The middleware then executes the yielded Effect as specified by the Effects API below. Meanwhile, the Generator will be suspended until the effect execution terminates. Upon receiving the result of the execution, the middleware calls `next(result)` on the Generator passing it the retrieved result as an argument. This process is repeated until the Generator terminates normally or by throwing some error.

If the execution results in an error (as specified by each Effect creator) then the `throw(error)` method of the Generator is called instead. If the Generator function defines a `try/catch` surrounding the current yield instruction, then the `catch` block will be invoked by the underlying Generator runtime. The runtime will also invoke any corresponding finally block.

In the case a Saga is cancelled (either manually or using the provided Effects), the middleware will invoke `return()` method of the Generator. This will cause the Generator to skip directly to the finally block.

Saga Helpers

> Note: the following functions are helper functions built on top of the Effect creators below.

`takeEvery(pattern, saga, ...args)`

Spawns a `saga` on each action dispatched to the Store that matches `pattern`.

- `pattern: String | Array | Function` - for more information see docs for `take(pattern)`
- `saga: Function` - a Generator function
- `args: Array<any>` - arguments to be passed to the started task. `takeEvery` will add the incoming action to the argument list (i.e. the action will be the last argument provided to `saga`)

Example

In the following example, we create a simple task `fetchUser`. We use `takeEvery` to start a new `fetchUser` task on each dispatched `USER_REQUESTED` action:

```
import { takeEvery } from `redux-saga/effects`

function* fetchUser(action) {
  ...
}

function* watchFetchUser() {
  yield takeEvery('USER_REQUESTED', fetchUser)
}
```

Notes

`takeEvery` is a high-level API built using `take` and `fork`. Here is how the helper could be implemented using the low-level Effects

```
function* takeEvery(pattern, saga, ...args) {
  const task = yield fork(function* () {
    while (true) {
      const action = yield take(pattern)
      yield fork(saga, ...args.concat(action))
    }
  })
  return task
}
```

`takeEvery` allows concurrent actions to be handled. In the example above, when a `USER_REQUESTED` action is dispatched, a new `fetchUser` task is started even if a previous `fetchUser` is still pending (for example, the user clicks on a `Load User` button 2 consecutive times at a rapid rate, the 2nd click will dispatch a `USER_REQUESTED` action while the `fetchUser` fired on the first one hasn't yet terminated)

`takeEvery` doesn't handle out of order responses from tasks. There is no guarantee that the tasks will termiate in the same order they were started. To handle out of order responses, you may consider `takeLatest` below.

`takeLatest(pattern, saga, ...args)`

Spawns a `saga` on each action dispatched to the Store that matches `pattern`. And automatically cancels any previous `saga` task started previous if it's still running.

Each time an action is dispatched to the store. And if this action matches `pattern`, `takeLatest` starts a new `saga` task in the background. If a `saga` task was started previously (on the last action dispatched before the actual action), and if this task is still running, the task will be cancelled.

- `pattern: String | Array | Function` - for more information see docs for `take(pattern)`
- `saga: Function` - a Generator function
- `args: Array<any>` - arguments to be passed to the started task. `takeLatest` will add the incoming action to the argument list (i.e. the action will be the last argument provided to `saga`)

Example

In the following example, we create a simple task `fetchUser`. We use `takeLatest` to start a new `fetchUser` task on each dispatched `USER_REQUESTED` action. Since `takeLatest` cancels any pending task started previously, we ensure that if a user triggers multiple consecutive `USER_REQUESTED` actions rapidly, we'll only conclude with the latest action

```
import { takeLatest } from `redux-saga/effects`

function* fetchUser(action) {
  ...
}

function* watchLastFetchUser() {
  yield takeLatest('USER_REQUESTED', fetchUser)
}
```

Notes

`takeLatest` is a high-level API built using `take` and `fork`. Here is how the helper could be implemented using the low-level Effects

```
function* takeLatest(pattern, saga, ...args) {
  const task = yield fork(function* () {
    let lastTask
    while (true) {
      const action = yield take(pattern)
      if (lastTask)
        yield cancel(lastTask) // cancel is no-op if the task has already terminated

      lastTask = yield fork(saga, ...args.concat(action))
    }
  })
  return task
}
```

`throttle(ms, pattern, saga, ...args)`

Spawns a `saga` on an action dispatched to the Store that matches `pattern`. After spawning a task it's still accepting incoming actions into the underlaying `buffer`, keeping at most 1 (the most recent one), but in the same time holding up with spawning new task for `ms` miliseconds (hence it's name - `throttle`). Purpose of this is to ignore incoming actions for a given period of time while processing a task.

- `ms: Number` - length of a time window in miliseconds during which actions will be ignored after the action starts processing
- `pattern: String | Array | Function` - for more information see docs for `take(pattern)`
- `saga: Function` - a Generator function

- `args: Array<any>` - arguments to be passed to the started task. `throttle` will add the incoming action to the argument list (i.e. the action will be the last argument provided to `saga`)

Example

In the following example, we create a simple task `fetchAutocomplete` . We use `throttle` to start a new `fetchAutocomplete` task on dispatched `FETCH_AUTOCOMPLETE` action. However since `throttle` ignores consecutive `FETCH_AUTOCOMPLETE` for some time, we ensure that user won't flood our server with requests.

```
import { throttle } from `redux-saga/effects`

function* fetchAutocomplete(action) {
  const autocompleteProposals = yield call(Api.fetchAutocomplete, action.text)
  yield put({type: 'FETCHED_AUTOCOMPLETE_PROPOSALS', proposals: autocompleteProposals}
)
}

function* throttleAutocomplete() {
  yield throttle(1000, 'FETCH_AUTOCOMPLETE', fetchAutocomplete)
}
```

Notes

`throttle` is a high-level API built using `take` , `fork` and `actionChannel` . Here is how the helper could be implemented using the low-level Effects

```
function* throttle(ms, pattern, task, ...args) {
  const throttleChannel = yield actionChannel(pattern, buffers.sliding(1))

  while (true) {
    const action = yield take(throttleChannel)
    yield fork(task, ...args, action)
    yield call(delay, ms)
  }
}
```

Effect creators

> Notes:
>
> - Each function below returns a plain JavaScript object and does not perform any execution.
> - The execution is performed by the middleware during the Iteration process described above.
> - The middleware examines each Effect description and performs the appropriate action.

`take(pattern)`

Creates an Effect description that instructs the middleware to wait for a specified action on the Store. The Generator is suspended until an action that matches `pattern` is dispatched.

`pattern` is interpreted using the following rules:

- If `take` is called with no arguments or `'*'` all dispatched actions are matched (e.g. `take()` will match all actions)

- If it is a function, the action is matched if `pattern(action)` is true (e.g. `take(action => action.entities)` will match all actions having a (truthy) `entities` field.)

  > Note: if the pattern function has `toString` defined on it, `action.type` will be tested against `pattern.toString()` instead. This is useful if you're using an action creator library like redux-act or redux-actions.

- If it is a String, the action is matched if `action.type === pattern` (e.g. `take(INCREMENT_ASYNC)`

- If it is an array, each item in the array is matched with beforementioned rules, so the mixed array of strings and function predicates is supported. The most common use case is an array of strings though, so that `action.type` is matched against all items in the array (e.g. `take([INCREMENT, DECREMENT])` and that would match either actions of type `INCREMENT` or `DECREMENT`).

The middleware provides a special action `END`. If you dispatch the END action, then all Sagas blocked on a take Effect will be terminated regardless of the specified pattern. If the terminated Saga has still some forked tasks which are still running, it will wait for all the child tasks to terminate before terminating the Task.

`take.maybe(pattern)`

Same as `take(pattern)` but does not automatically terminate the Saga on an `END` action. Instead all Sagas blocked on a take Effect will get the `END` object.

Notes

`take.maybe` got it name from the FP analogy - it's like instead of having a return type of `ACTION` (with automatic handling) we can have a type of `Maybe(ACTION)` so we can handle both cases:

- case when there is a `Just(ACTION)` (we have an action)
- the case of `NOTHING` (channel was closed*). i.e. we need some way to map over `END`

- internally all `dispatch` ed actions are going through the `stdChannel` which is geting closed when `dispatch(END)` happens

`take(channel)`

Creates an Effect description that instructs the middleware to wait for a specified message from the provided Channel. If the channel is already closed, then the Generator will immediately terminate following the same process described above for `take(pattern)` .

`take.maybe(channel)`

Same as `take(channel)` but does not automatically terminate the Saga on an `END` action. Instead all Sagas blocked on a take Effect will get the `END` object. See more here

`put(action)`

Creates an Effect description that instructs the middleware to dispatch an action to the Store. This effect is non-blocking and any errors that are thrown downstream (e.g. in a reducer) will not bubble back into the saga.

- `action: Object` - see Redux `dispatch` documentation for complete info

`put.resolve(action)`

Just like `put` but the effect is blocking (if promise is returned from `dispatch` it will wait for its resolution) and will bubble up errors from downstream.

- `action: Object` - see Redux `dispatch` documentation for complete info

`put(channel, action)`

Creates an Effect description that instructs the middleware to put an action into the provided channel.

- `channel: Channel` - a `Channel` Object.
- `action: Object` - see Redux `dispatch` documentation for complete info

This effect is blocking if the put is *not* buffered but immediately consumed by takers. If an error is thrown in any of these takers it will bubble back into the saga.

`call(fn, ...args)`

Creates an Effect description that instructs the middleware to call the function `fn` with `args` as arguments.

- `fn: Function` - A Generator function, or normal function which either returns a Promise as result, or any other value.

- `args: Array<any>` - An array of values to be passed as arguments to `fn`

Notes

`fn` can be either a *normal* or a Generator function.

The middleware invokes the function and examines its result.

If the result is an Iterator object, the middleware will run that Generator function, just like it did with the startup Generators (passed to the middleware on startup). The parent Generator will be suspended until the child Generator terminates normally, in which case the parent Generator is resumed with the value returned by the child Generator. Or until the child aborts with some error, in which case an error will be thrown inside the parent Generator.

If the result is a Promise, the middleware will suspend the Generator until the Promise is resolved, in which case the Generator is resumed with the resolved value. or until the Promise is rejected, in which case an error is thrown inside the Generator.

If the result is not an Iterator object nor a Promise, the middleware will immediately return that value back to the saga, so that it can resume its execution synchronously.

When an error is thrown inside the Generator. If it has a `try/catch` block surrounding the current `yield` instruction, the control will be passed to the `catch` block. Otherwise, the Generator aborts with the raised error, and if this Generator was called by another Generator, the error will propagate to the calling Generator.

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

Same as `call(fn, ...args)` but supports passing a `this` context to `fn` . This is useful to invoke object methods.

`apply(context, fn, [args])`

Alias for `call([context, fn], ...args)` .

`cps(fn, ...args)`

Creates an Effect description that instructs the middleware to invoke `fn` as a Node style function.

- `fn: Function` - a Node style function. i.e. a function which accepts in addition to its arguments, an additional callback to be invoked by `fn` when it terminates. The callback accepts two parameters, where the first parameter is used to report errors while the second is used to report successful results
- `args: Array<any>` - an array to be passed as arguments for `fn`

Notes

The middleware will perform a call `fn(...arg, cb)` . The `cb` is a callback passed by the middleware to `fn` . If `fn` terminates normally, it must call `cb(null, result)` to notify the middleware of a successful result. If `fn` encounters some error, then it must call `cb(error)` in order to notify the middleware that an error has occurred.

The middleware remains suspended until `fn` terminates.

`cps([context, fn], ...args)`

Supports passing a `this` context to `fn` (object method invocation)

`fork(fn, ...args)`

Creates an Effect description that instructs the middleware to perform a *non-blocking call* on `fn`

Arguments

- `fn: Function` - A Generator function, or normal function which returns a Promise as result
- `args: Array<any>` - An array of values to be passed as arguments to `fn`

returns a Task object.

Note

`fork`, like `call`, can be used to invoke both normal and Generator functions. But, the calls are non-blocking, the middleware doesn't suspend the Generator while waiting for the result of `fn`. Instead as soon as `fn` is invoked, the Generator resumes immediately.

`fork`, alongside `race`, is a central Effect for managing concurrency between Sagas.

The result of `yield fork(fn ...args)` is a Task object. An object with some useful methods and properties.

All forked tasks are *attached* to their parents. When the parent terminates the execution of its own body of instructions, it will wait for all forked tasks to terminate before returning.

Errors from child tasks automatically bubble up to their parents. If any forked task raises an uncaught error, then the parent task will abort with the child Error, and the whole Parent's execution tree (i.e. forked tasks + the *main task* represented by the parent's body if it's still running) will be cancelled.

Cancellation of a forked Task will automatically cancel all forked tasks that are still executing. It'll also cancel the current Effect where the cancelled task was blocked (if any).

If a forked task fails *synchronously* (ie: fails immediately after its execution before performing any async operation), then no Task is returned, instead the parent will be aborted as soon as possible (since both parent and child executes in parallel, the parent will abort as soon as it takes notice of the child failure).

To create *detached* forks, use `spawn` instead.

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

Supports invoking forked functions with a `this` context

`spawn(fn, ...args)`

Same as `fork(fn, ...args)` but creates a *detached* task. A detached task remains independent from its parent and acts like a top-level task. The parent will not wait for detached tasks to terminate before returning and all events which may affect the parent or the detached task are completely independents (error, cancellation).

`spawn([context, fn], ...args)`

Supports spawning functions with a `this` context

`join(task)`

Creates an Effect description that instructs the middleware to wait for the result of a previously forked task.

- `task: Task` - A Task object returned by a previous `fork`

Notes

`join` will resolve to the same outcome of the joined task (success or error). If the joined the task is cancelled, the cancellation will also propagate to the Saga executing the join effect effect. Similarly, any potential callers of those joiners will be cancelled as well.

`join(...tasks)`

Creates an Effect description that instructs the middleware to wait for the results of previously forked tasks.

- `tasks: Array<Task>` - A Task is the object returned by a previous `fork`

Notes

It simply wraps automatically array of tasks in join effects, so it becomes roughly equivalent of `yield tasks.map(t => join(t))`.

`cancel(task)`

Creates an Effect description that instructs the middleware to cancel a previously forked task.

- `task: Task` - A Task object returned by a previous `fork`

Notes

To cancel a running task, the middleware will invoke `return` on the underlying Generator object. This will cancel the current Effect in the task and jump to the finally block (if defined).

Inside the finally block, you can execute any cleanup logic or dispatch some action to keep the store in a consistent state (e.g. reset the state of a spinner to false when an ajax request is cancelled). You can check inside the finally block if a Saga was cancelled by issuing a `yield cancelled()`.

Cancellation propagates downward to child sagas. When cancelling a task, the middleware will also cancel the current Effect (where the task is currently blocked). If the current Effect is a call to another Saga, it will be also cancelled. When cancelling a Saga, all *attached forks* (sagas forked using `yield fork()`) will be cancelled. This means that cancellation effectively affects the whole execution tree that belongs to the cancelled task.

`cancel` is a non-blocking Effect. i.e. the Saga executing it will resume immediately after performing the cancellation.

For functions which return Promise results, you can plug your own cancellation logic by attaching a `[CANCEL]` to the promise.

The following example shows how to attach cancellation logic to a Promise result:

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

function myApi() {
  const promise = myXhr(...)

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

function* mySaga() {

  const task = yield fork(myApi)

  // ... later
  // will call promise[CANCEL] on the result of myApi
  yield cancel(task)
}
```

`cancel(...tasks)`

Creates an Effect description that instructs the middleware to cancel previously forked tasks.

- `tasks: Array<Task>` - A Task is the object returned by a previous `fork`

Notes

It simply wraps automatically array of tasks in cancel effects, so it becomes roughly equivalent of `yield tasks.map(t => cancel(t))` .

`cancel()`

Creates an Effect description that instructs the middleware to cancel a task in which it has been yielded (self cancellation). It allows to reuse destructor-like logic inside a `finally` blocks for both outer (`cancel(task)`) and self (`cancel()`) cancellations.

Example

```
function* deleteRecord({ payload }) {
  try {
    const { confirm, deny } = yield call(prompt);
    if (confirm) {
      yield put(actions.deleteRecord.confirmed())
    }
    if (deny) {
      yield cancel()
    }
  } catch(e) {
    // handle failure
  } finally {
    if (yield cancelled()) {
      // shared cancellation logic
      yield put(actions.deleteRecord.cancel(payload))
    }
  }
}
```

`select(selector, ...args)`

Creates an effect that instructs the middleware to invoke the provided selector on the current Store's state (i.e. returns the result of `selector(getState(), ...args)`).

- `selector: Function` - a function `(state, ...args) => args` . It takes the current state and optionally some arguments and returns a slice of the current Store's state
- `args: Array<any>` - optional arguments to be passed to the selector in addition of `getState` .

If `select` is called without argument (i.e. `yield select()`) then the effect is resolved with the entire state (the same result of a `getState()` call).

> It's important to note that when an action is dispatched to the store, the middleware first forwards the action to the reducers and then notifies the Sagas. This means that when you query the Store's State, you get the State **after** the action has been applied. However, this behavior is only guaranteed if all subsequent middlewares call `next(action)` synchronously. If any subsequent middleware calls `next(action)` asynchronously (which is unusual but possible), then the sagas will get the state from **before** the action is applied. Therefore it is recommended to review the source of each subsequent middleware to ensure it calls `next(action)` synchronously, or else ensure that redux-saga is the last middleware in the call chain.

Notes

Preferably, a Saga should be autonomous and should not depend on the Store's state. This makes it easy to modify the state implementation without affecting the Saga code. A saga should preferably depend only on its own internal control state when possible. But sometimes, one could find it more convenient for a Saga to query the state instead of maintaining the needed data by itself (for example, when a Saga duplicates the logic of invoking some reducer to compute a state that was already computed by the Store).

For example, suppose we have this state shape in our application:

```
state = {
  cart: {...}
}
```

We can create a *selector*, i.e. a function which knows how to extract the `cart` data from the State:

`./selectors`

```
export const getCart = state => state.cart
```

Then we can use that selector from inside a Saga using the `select` Effect:

`./sagas.js`

```
import { take, fork, select } from 'redux-saga/effects'
import { getCart } from './selectors'

function* checkout() {
  // query the state using the exported selector
  const cart = yield select(getCart)

  // ... call some API endpoint then dispatch a success/error action
}

export default function* rootSaga() {
  while (true) {
    yield take('CHECKOUT_REQUEST')
    yield fork(checkout)
  }
}
```

`checkout` can get the needed information directly by using `select(getCart)`. The Saga is coupled only with the `getCart` selector. If we have many Sagas (or React Components) that needs to access the `cart` slice, they will all be coupled to the same function `getCart`. And if we now change the state shape, we need only to update `getCart`.

`actionChannel(pattern, [buffer])`

Creates an effect that instructs the middleware to queue the actions matching `pattern` using an event channel. Optionally, you can provide a buffer to control buffering of the queued actions.

`pattern:` - see API for `take(pattern)` `buffer: Buffer` - a Buffer object

Example

The following code creates channel to buffer all `USER_REQUEST` actions. Note that even the Saga maybe blocked on the `call` effect. All actions that come while it's blocked are automatically buffered. This causes the Saga to execute the API calls one at a time

```
import { actionChannel, call } from 'redux-saga/effects'
import api from '...'

function* takeOneAtMost() {
  const chan = yield actionChannel('USER_REQUEST')
  while (true) {
    const {payload} = yield take(chan)
    yield call(api.getUser, payload)
  }
}
```

`flush(channel)`

Creates an effect that instructs the middleware to flush all buffered items from the channel. Flushed items are returned back to the saga, so they can be utilized if needed.

- `channel: Channel` - a `Channel` Object.

Example

```
function* saga() {
  const chan = yield actionChannel('ACTION')

  try {
    while (true) {
      const action = yield take(chan)
      // ...
    }
  } finally {
    const actions = yield flush(chan)
    // ...
  }

}
```

`cancelled()`

Creates an effect that instructs the middleware to return whether this generator has been cancelled. Typically you use this Effect in a finally block to run Cancellation specific code

Example

```
function* saga() {
  try {
    // ...
  } finally {
    if (yield cancelled()) {
      // logic that should execute only on Cancellation
    }
    // logic that should execute in all situations (e.g. closing a channel)
  }
}
```

Effect combinators

`race(effects)`

Creates an Effect description that instructs the middleware to run a *Race* between multiple Effects (this is similar to how `Promise.race([...])` behaves).

`effects: Object` - a dictionary Object of the form {label: effect, ...}

Example

The following example runs a race between two effects:

1. A call to a function `fetchUsers` which returns a Promise
2. A `CANCEL_FETCH` action which may be eventually dispatched on the Store

```
import { take, call, race } from `redux-saga/effects`
import fetchUsers from './path/to/fetchUsers'

function* fetchUsersSaga {
  const { response, cancel } = yield race({
    response: call(fetchUsers),
    cancel: take(CANCEL_FETCH)
  })
}
```

If `call(fetchUsers)` resolves (or rejects) first, the result of `race` will be an object with a single keyed object `{response: result}` where `result` is the resolved result of `fetchUsers`.

If an action of type `CANCEL_FETCH` is dispatched on the Store before `fetchUsers` completes, the result will be a single keyed object `{cancel: action}`, where action is the dispatched action.

Notes

When resolving a `race`, the middleware automatically cancels all the losing Effects.

`[...effects] (parallel effects)`

Creates an Effect description that instructs the middleware to run multiple Effects in parallel and wait for all of them to complete.

Example

The following example runs two blocking calls in parallel:

```
import { fetchCustomers, fetchProducts } from './path/to/api'

function* mySaga() {
  const [customers, products] = yield [
    call(fetchCustomers),
    call(fetchProducts)
  ]
}
```

Notes

When running Effects in parallel, the middleware suspends the Generator until one of the following occurs:

- All the Effects completed with success: resumes the Generator with an array containing the results of all Effects.
- One of the Effects was rejected before all the effects complete: throws the rejection error inside the Generator.

Interfaces

Task

The Task interface specifies the result of running a Saga using `fork` , `middleware.run` or `runSaga` .

method	return value
task.isRunning()	true if the task hasn't yet returned or thrown an error
task.isCancelled()	true if the task has been cancelled
task.result()	task return value. \`undefined\` if task is still running
task.error()	task thrown error. \`undefined\` if task is still running
task.done	a Promise which is either: • resolved with task's return value • rejected with task's thrown error
task.cancel()	Cancels the task (If it is still running)

Channel

A channel is an object used to send and receive messages between tasks. Messages from senders are queued until an interested receiver request a message, and registered receiver is queued until a message is available.

Every channel has an underlying buffer which defines the buffering strategy (fixed size, dropping, sliding)

The Channel interface defines 3 methods: `take` , `put` and `close`

`Channel.take(callback):` used to register a taker. The take is resolved using the following rules

- If the channel has buffered messages, then `callback` will be invoked with the next message from the underlying buffer (using `buffer.take()`)
- If the channel is closed and there are no buffered messages, then `callback` is invoked with `END`
- Otherwise `callback` will be queued until a message is put into the channel

`Channel.put(message):` Used to put message on the buffer. The put will be handled using the following rules

- If the channel is closed, then the put will have no effect.
- If there are pending takers, then invoke the oldest taker with the message.
- Otherwise put the message on the underlying buffer

`Channel.flush(callback):` Used to extract all buffered messages from the channel. The flush is resolved using the following rules

- If the channel is closed and there are no buffered messages, then `callback` is invoked with `END`
- Otherwise `callback` is invoked with all buffered messages.

`Channel.close():` closes the channel which means no more puts will be allowed. All pending takers will be invoked with `END` .

Buffer

Used to implement the buffering strategy for a channel. The Buffer interface defines 3 methods: `isEmpty` , `put` and `take`

- `isEmpty()` : returns true if there are no messages on the buffer. A channel calls this method whenever a new taker is registered
- `put(message)` : used to put new message in the buffer. Note the Buffer can chose to not

store the message (e.g. a dropping buffer can drop any new message exceeding a given limit)
- `take()` used to retrieve any buffered message. Note the behavior of this method has to be consistent with `isEmpty`

SagaMonitor

Used by the middleware to dispatch monitoring events. Actually the middleware dispatches 5 events:

- When an effect is triggered (via `yield someEffect`) the middleware invokes `sagaMonitor.effectTriggered`
- If the effect is resolved with success the middleware invokes `sagaMonitor.effectResolved`
- If the effect is rejected with an error the middleware invokes `sagaMonitor.effectRejected`
- If the effect is cancelled the middleware invokes `sagaMonitor.effectCancelled`
- Finally, the middleware invokes `sagaMonitor.actionDispatched` when a Redux action is dispatched.

Below the signature for each method

- `effectTriggered(options)` : where options is an object with the following fields
    - `effectId` : Number - Unique ID assigned to the yielded effect
    - `parentEffectId` : Number - ID of the parent Effect. In the case of a `race` or `parallel` effect, all effects yielded inside will have the direct race/parallel effect as a parent. In case of a top-level effect, the parent will be the containing Saga
    - `label` : String - In case of a `race` effect, all child effects will be assigned as label the corresponding keys of the object passed to `race`
    - `effect` : Object - the yielded effect itself
- `effectResolved(effectId, result)`
    - `effectId` : Number - The ID of the yielded effect
    - `result` : any - The result of the successful resolution of the effect. In case of `fork` or `spawn` effects, the result will be a `Task` object.
- `effectRejected(effectId, error)`
    - `effectId` : Number - The ID of the yielded effect

  - `error` : any - Error raised with the rejection of the effect

- `effectCancelled(effectId)`
  - `effectId` : Number - The ID of the yielded effect
- `actionDispatched(action)`
  - `action` : Object - The dispatched Redux action. If the action was dispatched by a Saga then the action will have a property `SAGA_ACTION` set to true (`SAGA_ACTION` can be imported from `redux-saga/utils`).

External API

`runSaga(iterator, options)`

Allows starting sagas outside the Redux middleware environment. Useful if you want to connect a Saga to external input/output, other than store actions.

`runSaga` returns a Task object. Just like the one returned from a `fork` effect.

- `iterator: {next, throw}` - an Iterator object, Typically created by invoking a Generator function
- `options: Object` - currently supported options are:
  - `subscribe(callback): Function` - A function which accepts a callback and returns an `unsubscribe` function
    - `callback(input): Function` - callback(provided by runSaga) used to subscribe to input events. `subscribe` must support registering multiple subscriptions.
      - `input: any` - argument passed by `subscribe` to `callback` (see Notes below)
  - `dispatch(output): Function` - used to fulfill `put` effects.
    - `output: any` - argument provided by the Saga to the `put` Effect (see Notes below).
  - `getState(): Function` - used to fulfill `select` and `getState` effects
  - `sagaMonitor` : SagaMonitor - see docs for `createSagaMiddleware(options)`
  - `logger` : `Function` - see docs for `createSagaMiddleware(options)`
  - `onError` : `Function` - see docs for `createSagaMiddleware(options)`

Notes

The `{subscribe, dispatch}` is used to fulfill `take` and `put` Effects. This defines the Input/Output interface of the Saga.

`subscribe` is used to fulfill `take(PATTERN)` effects. It must call `callback` every time it has an input to dispatch (e.g. on every mouse click if the Saga is connected to DOM click events). Each time `subscribe` emits an input to its callbacks, if the Saga is blocked on a `take` effect, and if the take pattern matches the currently incoming input, the Saga is resumed with that input.

`dispatch` is used to fulfill `put` effects. Each time the Saga emits a `yield put(output)`, `dispatch` is invoked with output.

Utils

`channel([buffer])`

A factory method that can be used to create Channels. You can optionally pass it a buffer to control how the channel buffers the messages.

By default, if no buffer is provided, the channel will queue incoming messages up to 10 until interested takers are registered. The default buffering will deliver message using a FIFO strategy: a new taker will be delivered the oldest message in the buffer.

`eventChannel(subscribe, [buffer], [matcher])`

Creates channel that will subscribe to an event source using the `subscribe` method. Incoming events from the event source will be queued in the channel until interested takers are registered.

- `subscribe: Function` used to subscribe to the underlying event source. The function must return an unsubscribe function to terminate the subscription.
- `buffer: Buffer` optional Buffer object to buffer messages on this channel. If not provided messages will not buffered on this channel.
- `matcher: Function` optional predicate function (`any => Boolean`) to filter incoming messages. Only messages accepted by the matcher will be put on the channel.

To notify the channel that the event source has terminated, you can notify the provided subscriber with an `END`

Example

In the following example we create an event channel that will subscribe to a `setInterval`

```
const countdown = (secs) => {
  return eventChannel(emitter => {
      const iv = setInterval(() => {
        console.log('countdown', secs)
        secs -= 1
        if (secs > 0) {
          emitter(secs)
        } else {
          emitter(END)
          clearInterval(iv)
          console.log('countdown terminated')
        }
      }, 1000);
      return () => {
        clearInterval(iv)
        console.log('countdown cancelled')
      }
    }
  )
}
```

`buffers`

Provides some common buffers

- `buffers.none()` : no buffering, new messages will be lost if there are no pending takers
- `buffers.fixed(limit)` : new messages will be buffered up to `limit` . Overflow will raises an Error. Omitting a `limit` value will result in a limit of 10.
- `buffers.expanding(initialSize)` : like `fixed` but Overflow will cause the buffer to expand dynamically.
- `buffers.dropping(limit)` : same as `fixed` but Overflow will silently drop the messages.
- `buffers.sliding(limit)` : same as `fixed` but Overflow will insert the new message at the end and drop the oldest message in the buffer.

`delay(ms, [val])`

Returns a Promise that will resolve after `ms` milliseconds with `val` .

`cloneableGenerator(generatorFunc)`

Takes a generator function (function*) and returns a generator function. All generators instanciated from this function will be cloneable. For testing purpose only.

Example

This is useful when you want to test different branch of a saga without having to replay the actions that lead to it.

```
function* oddOrEven() {
  // some stuff are done here
  yield 1;
  yield 2;
  yield 3;

  const userInput = yield 'enter a number';
  if (userInput % 2 === 0) {
    yield 'even';
  } else {
    yield 'odd'
  }
}

test('my oddOrEven saga', assert => {
  const data = {};
  data.gen = cloneableGenerator(oddOrEven)();

  assert.equal(
    data.gen.next().value,
    1,
    'it should yield 1'
  );

  assert.equal(
    data.gen.next().value,
    2,
    'it should yield 2'
  );

  assert.equal(
    data.gen.next().value,
    3,
    'it should yield 3'
  );

  assert.equal(
    data.gen.next().value,
    'enter a number',
    'it should ask for a number'
  );
  assert.test('even number is given', a => {
    // we make a clone of the generator before giving the number;
    data.clone = data.gen.clone();

    a.equal(
      data.gen.next(2).value,
      'even',
      'it should yield "event"'
    );

    a.equal(
      data.gen.next().done,
      true,
      'it should be done'
    );

    a.end();
  });

  assert.test('odd number is given', a => {

    a.equal(
      data.clone.next(1).value,
      'odd',
      'it should yield "odd"'
    );

    a.equal(
      data.clone.next().done,
      true,
      'it should be done'
    );

    a.end();
  });

  assert.end();
});
```

`createMockTask()`

Returns an object that mocks a task. For testing purposes only. See Task Cancellation docs for more information.)

Cheatsheets

Blocking / Non-blocking

Name	Blocking
takeEvery	No
takeLatest	No
throttle	No
take	Yes
take(channel)	Sometimes (see API reference)
take.maybe	Yes
put	No
put.resolve	Yes
put(channel, action)	No
call	Yes
apply	Yes
cps	Yes
fork	No
spawn	No
join	Yes
cancel	Yes
select	No
actionChannel	No
flush	Yes
cancelled	Yes
race	Yes
[...effects]	Blocks only if there is a blocking effect in the array