Содержание

Redux-Saga - Русский перевод

Оригинал (На английском языке): https://redux-saga.js.org/

Перевод для этой страницы взят отсюда: https://github.com/redux-saga/redux-saga/blob/master/README_ru.md

`redux-saga` - это библиотека, которая призвана упростить и улучшить выполнение сайд-эффектов (т.е. таких действий, как асинхронные операции, типа загрузки данных и "грязных" действий, типа доступа к браузерному кэшу) в React/Redux приложениях.

Можно представить это так, что сага - это как отдельный поток в вашем приложении, который отвечает за сайд-эффекты. `redux-saga` - это redux мидлвар, что означает, что этот поток может запускаться, останавливаться и отменяться из основного приложения с помощью обычных redux экшенов, оно имеет доступ к полному состоянию redux приложения и также может диспатчить redux экшены.

Библиотека использует концепцию ES6, под названием генераторы, для того, чтобы сделать эти асинхронные потоки легкими для чтения, написания и тестирования. *(если вы не знакомы с этим, здесь есть некоторые ссылки для ознакомления)* Тем самым, эти асинхронные потоки выглядят, как ваш стандартный синхронный JavaScript код. (наподобие `async` / `await`, но генераторы имеют несколько отличных возможностей, необходимых нам)

Возможно, вы уже использовали `redux-thunk`, перед тем как обрабатывать ваши выборки данных. В отличие от redux thunk, вы не оказываетесь в callback аду, вы можете легко тестировать ваши асинхронные потоки и ваши экшены остаются чистыми.

Приступая к работе

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

или

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

Альтернативно, вы можете использовать предоставленные UMD сборки напрямую в `<script>` на HTML странице.

Пример использования

Предположим, что у нас есть интерфейс для извлечения некоторых пользовательских данных с удаленного сервера при нажатии кнопки. (Для краткости, мы просто покажем код запуска экшена.)

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

Компонент диспатчит action в виде простого объекта в Store. Мы создадим сагу, которая слушает все `USER_FETCH_REQUESTED` экшены и триггерит вызовы API для извлечения пользовательских данных.

`sagas.js`

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

// worker Saga: будет запускаться на экшены типа `USER_FETCH_REQUESTED`
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});
   }
}

/*
  Запускаем `fetchUser` на каждый задиспатченый экшен `USER_FETCH_REQUESTED`.
  Позволяет одновременно получать данные пользователей.
*/
function* mySaga() {
  yield takeEvery("USER_FETCH_REQUESTED", fetchUser);
}

/*
  В качестве альтернативы вы можете использовать `takeLatest`.

  Не допускает одновременное получение данных пользователей. Если `USER_FETCH_REQUESTE
D`
  диспатчится в то время когда предыдущий запрос все еще находится в ожидании ответа,
  то этот ожидающий ответа запрос отменяется и срабатывает только последний.
*/
function* mySaga() {
  yield takeLatest("USER_FETCH_REQUESTED", fetchUser);
}

export default mySaga;
```

Для запуска нашей саги, мы подключим ее к Redux Store, используя `redux-saga` мидлвар.

`main.js`

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

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

// создаем saga мидлвар
const sagaMiddleware = createSagaMiddleware()
// монтируем его в Store
const store = createStore(
  reducer,
  applyMiddleware(sagaMiddleware)
)

// затем запускаем сагу
sagaMiddleware.run(mySaga)

// рендерим приложение
```

Базовые понятия

Использование Saga Helpers

`redux-saga` предоставляет Helper эффекты, которые врапят внутренние функции для запуска задач, когда указанные action отправляются в Store.

Helper функции построены поверх API нижнего уровня. В расширенном разделе мы увидим, как эти функции могут быть реализованы. Первая функция `takeEvery` является наиболее знакомой и обеспечивает поведение похожее на `redux-thunk`.

Давайте рассмотрим пример с AJAX.При каждом нажатии на кнопку `Fetch`, мы диспатчим action `FETCH_REQUESTED` и хотим обработать его запуском таска, который будет извлекать данные с сервера.

Сначала мы создаем задачу, которая будет выполнять асинхронное действие (**Api.fetchUser**):

```
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})
   }
}
```

Чтобы таск запускался для каждого action `FETCH_REQUESTED`, напишем:

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

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

В приведенном выше примере `takeEvery` позволяет одновременно запускать несколько экземпляров `fetchData` одновременно. В данный момент мы можем запустить новый таск `fetchData`, пока есть все ещё запущеные таски `fetchData`, которые еще не завершены.

Если мы хотим получить ответ только от последнего запроса (например, чтобы всегда показывать последнюю версию данных), мы можем использовать helper `takeLatest`:

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

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

В отличие от `takeEvery`, `takeLatest` позволяет запустить только один таск `fetchData` (в одно время). Если предыдущая задача все еще запущена пока другой таск `fetchData` все ещё работает, то тогда предыдущая задача будет автоматически отменена.

Если у вас есть несколько сагов, которые наблюдают за различными action, вы можете создать несколько наблюдателей с помощью helper функции, которые будут вести себя так, будто их создали через `fork` (мы поговорим о `fork` позже. На данный момент будем просто счиать что этот Effect позволяет нам запускать несколько саг в фоновом режиме)

Пример:

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

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

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

// запускает их параллельно
export default function* rootSaga() {
  yield takeEvery('FETCH_USERS', fetchUsers)
  yield takeEvery('CREATE_USER', createUser)
}
```

Декларативные эффекты

В `redux-saga` саги реализованы с использованием функций генераторов. Чтобы выразить логику сагов, мы используя ключевоео слово `yield` и передаем простые объекты JavaScript из генератора. Мы называем их объ*екты Effects.* Effect это простой объект, который содержит информацию, которая должна интерпретироваться с помощью middleware. Вы можете просматривать эффекты, как инструкции в middleware которые выполняют какую-то операцию (вызовите любую асинхронную функцию и диспачните её в ваш Store).

Чтобы создать effects, вы можете используете функции которые находятся в пакете `redux-saga/effects` .

В этом разделе и ниже мы раскажем об основных Effects-ах. И рассмотрим, как эта концепция позволяет легко тестировать вашии саги.

Саги могут отдавать Effect в нескольких формах. Самый простой способ используя ключивое слово `yield` вернуть `Promise` .

Например, предположим, что у нас есть сага, которая следит за action `PRODUCT_REQUESTED` . При каждом диспатче action `PRODUCT_REQUESTED` , запускается задача которая получает список продуктов с сервера.

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

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

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

В приведенном выше примере мы вызываем `Api.fetch` непосредственно изнутри генератора (в функциях генераторах, любое выражение справа от `yield` вычисляется, и только потом отдается).

`Api.fetch('/products')` запускает AJAX-запрос и возвращает **Promise** that will resolve with the resolved response, the AJAX request will be executed immediately.

Просто, но...

Предположим, мы хотим создать тест для генератор выше:

```
const iterator = fetchProducts()
assert.deepEqual(iterator.next().value, ??) // что мы ожидаем?
```

Мы хотим проверить результат первого значения, полученного из генератора. В нашем случае это результат выполнения `Api.fetch('/products')`, который является Promise-ом.

Выполнение реального сервиса во время тестов не является ни жизнеспособным, ни практическим подходом, поэтому приходится мокать результат функции `Api.fetch`, т. е. мы должны заменить реальные функции фейковыми, которая на самом деле не делают AJAX-запрос, а только проверяет, что `Api.fetch` с правильными аргументами (`'/products'` в нашем случае).

Моки делают тестирование более сложным и менее надежным. С другой стороны, функции которая просто возвращают значения легче проверить, так как мы можем простую использовать `equal()`, чтобы проверить результат.

Это способ позволяет написать самые надежные тесты.

Не верите? Советую вам прочитать 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?
>
> Если вы закончите тест, не отвечая на эти два вопроса, у вас нет реального модульного теста. У вас будут только неаккуратные тесты.

Нам нужно просто чтобы убедиться, что таск `fetchProducts` вызывает с использованием yield правильную функцию с правильным набором аргументов.

Вместо вызова асинхронной функции непосредственно из Генератора, **мы можем вернуть только описание вызова функции.** т. е. мы просто вернем объект, который выглядит следующим образом

```
// Effect -> вызывает функцию Api.fetch с аргументом `./products`
{
  CALL: {
    fn: Api.fetch,
    args: ['./products']
  }
}
```

Иными словами, генератор будет отдавать простые объекты, содержащие инструкции и middleware `redux-saga` будет заботиться о выполнении этих поручений и результат их выполнения для генератора. Таким образом, при тестировании Генератора, все, что нам нужно сделать, это проверить, что он отдает ожидаемую инструкцию, просто вызывая `deepEqual` на данном объекте.

По этой причине библиотека предоставляет другой способ выполнения асинхронных вызовов.

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

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

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')"
)
```

Теперь мы используем функцию `call(fn, ...args)`. **Отличие от предыдущего примера заключается в том, что теперь мы не будем получать данные используя `fetch` сразу, вместо этого мы вызовем `call` для создания описания Effect-а.**

Так же, как в Redux вы используете action creator для создания простого объекта, описывающего действие, которое будет выполняться Store, `call` создает простой объект, описывающий вызов функции. Middleware в Redux-saga заботится о выполнении вызова функции и возобновлении работы генератора с полученым ответом.

Это позволяет легко проверить генератор вне среды Redux-а. Потому что `call` это просто функция, которая возвращает простой объект.

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

const iterator = fetchProducts()

// ожидается вызов
assert.deepEqual(
  iterator.next().value,
  call(Api.fetch, '/products'),
  "fetchProducts should yield an Effect call(Api.fetch, './products')"
)
```

Теперь нам не нужно мокать какие-либо данные и нам достаточно простой проверки полученных данных на равенство.

Преимущество этих *декларативных вызовов* состоит в том, что мы можем проверить всю логику внутри саги, просто проводя итерацию генератора и выполняя `deepEqual` с значением полученным последовательно из генератора. Это реальная выгода, так как ваши сложные асинхронные операции больше не являются черными ящиками, и вы можете детально проверить их логику независимо от того, насколько она сложна.

`call` также поддерживает вызов объектных методов, вы можете представить `this` в контексте вызываемой функции, используя следующую форму:

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

`apply` является алиасом для формы вызова метода

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

`call` и `apply` хорошо подходят для функций, которые возвращают результаты в виде Promise.

Другая функция `cps` может использоваться для обработки функций в стиле Node (например, `fn(...args, callback)` , где `callback` имеет вид `(error, result) => ()`).

Пример:

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

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

И, конечно, вы можете написать тест для него также же, как бы вы написали для `call`:

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

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

`cps` также поддерживает ту же форму вызова метода, что и `call`.

Диспатчим action в Store

Следуя предыдущему примеру, предположим, что после каждого сохранения мы хотим отправить какое-либо действие, чтобы уведомить Store о том, что получение данных прошло успешно (в данный момент мы не будем думать о возможных ошибках при получении данных).

Мы могли бы поместить функцию `dispatch` в генератор и затем генератор может вызвать его (*dispatch*) после получения ответа:

```
// ...

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

Однако это решение имеет те же недостатки, что и функции вызываемые непосредственно изнутри генератора (как было сказано в предыдущем разделе). Если мы хотим проверить, что `fetchProducts` выполняет отправку после получения ответа от AJAX, то нам придется опять создавать mock-функцию `dispatch`.

Вместо этого нам нужно декларативное решение. Просто создайте объект что бы указать middleware что нам нужно диспачить action и позвольте middleware выполнить настоящию отправку[1]. Таким образом, мы можем проверить отправку генератора таким же образом: просто проверив полученный эффект через полученый от генератора и убедится, что он содержит правильные инструкции.

Для этой цели библиотека предоставляет функцию `put` который диспатчит Effect.

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

function* fetchProducts() {
  const products = yield call(Api.fetch, '/products')
  // создаем и диспатчим Effect
  yield put({ type: 'PRODUCTS_RECEIVED', products })
}
```

Теперь мы можем легко написать тест для генератора, как в предыдущем разделе

```
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')"
)

// создаем фейковый ответ с сервера
const products = {}

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

Обратите внимание, как мы передаем фейковый ответ генератору с помощью метода `next`. Во вне middleware мы имеем полный контроль над генератором, мы можем имитировать реальную среду с простым фейковым результатом и и возабновить генератор с ними. Mocking данные намного проще, чем mock-функции и [spying calls].

> [1]. Just create an Object to instruct the middleware that we need to dispatch some action, and let the middleware perform the real dispatch. ↩

Базовая абстракция: Effect

~~Чтобы обобщить инициирование сайд эффектов внутри саги всегда [yielding] декларативный эффект.~~ (Вы также можете используя `yield` чтобы вернуть `Promise` напрямую, но это затруднит тестирование, как мы видели в первом разделе.)

Saga составляет все эти эффекты вместе для реализации желаемого потока управления. Простейшим примером является последовательность Effect-ов возващаемых используя ключевое слово `yield` : просто поместив инструктции с использованием ключевого слова `yield` в генератор которые будут следовать один за другим. Вы также можете использовать знакомые операторы потока управления (`if` , `while` , `for`) для реализации более сложных потоков управления.

Мы видели, что использование Effect-ов, таких как `call` и `put` , в сочетании с API высокого уровня, таких как `takeEvery` , позволяет нам достичь тех же вещей, что и `redux-thunk,` но с дополнительным преимуществом легкого тестирования.

Но `redux-saga` дает еще одно преимущество перед `redux-thunk` . В дополительном разделе вы столкнетесь с более мощными Effect-ами, которые позволят вам выражать сложные потоки управления, сохраняя при этом приемущество легкого написание тестов.

Получение будущих действий

До сих пор мы использовали helper Effect `takeEvery`, чтобы создать новый таск для каждого входящего action. Это немного напоминает поведение `redux-thunk`: например, каждый раз, когда компонент, вызывает `fetchProducts` Action Creator, Action Creator отправляет thunk для выполнения потока управления.

В действительности, `takeEvery` - всего лишь враппер для внутренней вспомогательной функции, созданной на основе более низкого уровня и более мощного API. В этом разделе мы увидим новый Effect `take`, который позволит построить сложный поток управления, позволяя полностью контролировать процесс наблюдения за action.

Простой логгер action-ов

Давайте рассмотрим простой пример Саги, которая следит за всеми действиями, отправленными в Store, и выводит их в консоль.

Используя `takeEvery('*')` (с аргументом `*`) мы можем отловить все отправленные action независимо от их типов.

```
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)
  })
}
```

Теперь давайте посмотрим, как использовать Effect `take` для реализации того же потока, что и выше

```
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)
  }
}
```

Effect `take` похож на `call` и на `put` который мы видили раньше. Он создает еще один командный объект,[1] который сообщает middleware, что-бы тот ждал определенного действия.

Результирующее поведение Effect-а `call` аналогично тому, когда midleware приостанавливает генератор до того пока `Promise` не вернет результат. В приведенном выше примере `watchAndLog` приостанавливается до тех пор пока не будет диспачено какое нибудь действие.

Обратите внимание на то, как мы запускаем бесконечный цикл `while (true)`. Помните, что это функция Генератор, которая работает не также как обычные функции. Наш генератор будет останавливатся на каждой итерации в ожидании пока не произайдет диспатч.

Использование `take` оказывает тонкое влияние на то, как мы пишем наш код. В случае `takeEvery` вызываемые таски не контролируют, когда они будут вызваны. Они будут вызываться снова и снова для каждого соответствующего action. Они также не контролируют, когда остановить наблюдение.

В случае с `take` управление - инвертируется. Instead of the actions being *pushed* to the handler tasks, the Saga is *pulling* the action by itself. Это выглядит так, как будто Saga выполняет нормальную функцию `action = getNextAction()`, которая будет решать, когда action будет диспатчен.

Эта инверсия управления позволяет нам реализовать потоки управления, которые нетривиальны для традиционного подхода *push*.

В качестве простого примера предположим, что в нашем приложении **Todo** мы хотим посмотреть действия пользователя и показать поздравительное сообщение после того, как пользователь создал свои первые три заметки.

```
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 two actions `LOGIN` and `LOGOUT`. Using `takeEvery` (or `redux-thunk`) we'll have to write two separate tasks (or thunks): one for `LOGIN` and the other for `LOGOUT`.

The result is that our logic is now spread in two places. In order for someone reading our code to understand it, they would have to read the source of the two handlers and make the link between the logic in both in their head. In other words, it means they would have to rebuild the model of the flow in their 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')
    // ... выполняем логику авторизации
    yield take('LOGOUT')
    // ... выполняем логику выхода
  }
}
```

`loginFlow` более четко передает ожидаемую последовательность действий. Он знает, что за action `LOGIN` всегда следует действие `LOGOUT` и что `LOGOUT` всегда следует `LOGIN` (хороший пользовательский интерфейс должен всегда обеспечивать последовательный порядок действий, скрывая или отключая неожиданное фсешщт).

> [1]. It creates another command object ↩

Не блокирующие вызовы

В предыдущем разделе мы видели, как Effect `take` позволяет лучше всего описать нетривиальный поток в центральном месте.

Посмотрим ещё раз примера потока входа:

```
function* loginFlow() {
  while (true) {
    yield take('LOGIN')
    // ... выполнить логику входа
    yield take('LOGOUT')
    // ... выполнить логику выхода
  }
}
```

Давайте завершим пример и реализуем логику входа и выхода. Допустим, у нас есть API, который позволяет авторизовать пользователя на удаленном сервере. Если авторизация выполнена успешно, сервер вернет авторизационный токен, который будет сохранен нашим приложением с помощью хранилища DOM (предположим, что наш API предоставляет другой сервис для хранения DOM).

Когда пользователь выйдет из системы, мы просто удалим ранее сохраненный токен авторизации.

Первая попытка

Пока что у нас есть все необходимые эффекты для реализации вышеуказанного потока. Мы можем ждать конкретных действий в store, используя Effect `take`. С помощью `call` Effect можно выполнять асинхронные вызовы.

Наконец, мы можем отправлять действия в Store с помощью Effect `put`.

Давайте попробуем:

> 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:

```
iimport { 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

Запуск тасков параллельно

Ключевое слово `yield` отлично подходит для представления асинхронного потока управления в простом и линейном стиле, но нам также нужно выполнять параллельно что-то ещё.

Мы не можем просто написать:

```
// неправильно, Effect-ы будут выполняться последовательно
const users  = yield call(fetch, '/users'),
      repos = yield call(fetch, '/repos')
```

Потому что 2-й Effect не будет выполнен, пока не будет получен результат первого вызова. Вместо этого мы должны написать так:

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

// правильно, Effect-ы будут выполняться параллельно
const [users, repos]  = yield all([
  call(fetch, '/users'),
  call(fetch, '/repos')
])
```

Когда мы возвращаем массив используя `yield` таким образом, то, генератор блокируется до тех пор, пока все Effect-ы не будут завершены или пока хотя бы один не вернет reject (Promise.reject) (работает также как `Promise.all`[1]).

> [1]. *Метод Promise.all возвращает обещание, которое выполнится тогда, когда будут выполнены все обещания, переданные в виде перечисляемого аргумента, или отклонено любое из переданных обещаний:* https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Promise/all ↩

Запуск гонки между несколькими Effect-ами

Иногда мы запускаем несколько задач параллельно, но мы не хотим ждать их всех, нам просто нужно получить победителя: первый, который вернет результат (или исключение / Promise.reject).

Effect `race` запускает гонку между несколькими Effect-ами.

В следующем примере показан таск, который запускает запрос удаленной выборки и ограничивает ответ в течение 1 секунды.

```
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'})
}
```

Еще одна полезная особенность `race` состоит в том, что она автоматически отменяет Effect-ы проигравших. Например, предположим, что у нас есть две кнопки:

- Первый запускает задачу в фоновом режиме, которая выполняется в бесконечном цикле `while (true)` (например, синхронизация данных с сервером каждые x секунд).
- После запуска фоновой задачи мы включаем вторую кнопку, которая отменяет задачу

```
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')
    })
  }
}
```

В случае диспатча action `CANCEL_TASK` , Effect `race` автоматически отменяет `backgroundTask` , выбрасывая в нем исключение.

Последовательность Saga через `yield*`

Вы можете использовать встроенный оператор `yield*` для создания последовательности сагов. Это позволяет вам последовательно выполнять макрозадания в простой процедурном стиле.

```
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))
}
```

Обратите внимание, что использование `yield*` приведет к тому, что среда выполнения JavaScript распространит всю последовательность.

Получившийся итератор (из `game()`) даст все значения из вложенных итераторов.

`yield*` - более мощная альтернатива использующая более общий механизм составления промежуточного слоя1.

> 1. A more powerful alternative is to use the more generic middleware composition mechanism ↩

Составление Saga-ов

Хотя использование `yield*` обеспечивает идиоматический способ составления Saga-ов, этот подход имеет некоторые ограничения:

- Вероятно, вы захотите протестировать вложенные генераторы отдельно.Это приводит к некоторому дублированию в тестовом коде, а также приводит к накладным расходам на дублирующийся код. Мы не хотим выполнять вложенный генератор, а только хотим убедитесь, что он был вызван с правильным аргументом.
- Что еще более важно, `yield*` принимает в качестве аргумента только последовательность тасков, поэтому вы можете передавать `yield*` только один генератор за раз.

Вы можете просто использовать `yield`, чтобы параллельно запускать одну или несколько подзадач.

При вызове генератора, Saga будет ждать завершения работы генератора перед продолжением задачи, а затем возабновится с возвращенным значением (или выбрасывает исключение, если ошибка распространяется от подзадачи).

```
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) // запускает и ожидает завершения таска fetchPosts
  }
}
```

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 all([call(task1), call(task2), ...])
  yield put(showResults(results))
}
```

На самом деле, yield Saga-ов не отличается от yield других Effect-ов (actions, таймауты, т.д.).

Например, вам может понадобавится, чтобы пользователь закончил игру в течение ограниченного периода времени:

```
function* game(getState) {
  let finished
  while (!finished) {
    // есть всего 60 секунд что бы закончить игру
    const {score, timeout} = yield race({
      score: call(play, getState),
      timeout: call(delay, 60000) // 60000 - 60 секунд
    })

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

Примеры тестирования Saga-ов

Effect-ы возвращают простые объекты JavaScript

Эти объекты описывают Effect-ы, и `redux-saga` отвечает за их выполнение.

Это делает тестирование очень простым, потому что все, что вам нужно сделать, это сравнить, что объект, полученный сагой, с нужным вам объектом.

Основной пример

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

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

Тестирование саги, ожидающей action от пользователя и диспачит action `CHANGE_UI`

```
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();
});
```

Еще одно большое преимущество в том, что ваши тесты также являются вашим документом[1]! Они описывают все, что должно произойти.

Ветление Saga

Иногда ваша сага будет иметь разные результаты. Чтобы протестировать разные ветви без повторения всех шагов, вы можете использовать функцию **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();
  });
});
```

См. Также: Task cancellation for testing fork effects

См. Также: Примеры:

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

> [1]. Another great benefit is that your tests are also your doc ↩

Подключение Saga-ов к внешнему входу/выходу

Мы видели, что Effect `take` выполняется путем ожидания action которые были диспатчены в Store. И то что Effect `put` выполняется путем отправки диспатча action указаного в качестве аргумента.

Когда запущена Saga (либо при запуске, либо позже динамически), middleware автоматически связывает `take` / `put` с Store. Эти 2 Effect-а можно рассматривать как своего рода вход/выход для саги.

`redux-saga` поддерживает способ запуска Saga-ов вне среды Redux middleware и подключения к пользовательскому 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(
  myIO
  saga,
)
```