Redux Saga

Redux Saga – это middleware для Redux.

Redux Saga отвечает за вызов API, работа с браузерным API, запись в cash, запросы на сервер, работа с длительными транзакциями, сбор ошибок и прочее.

Ключевая концепция – *создать отдельный поток данных* в приложении, которые отвечают за все асинхронные операции.

Redux Saga – это *side effect manager* для Redux.

Redux Saga – это шаблон проектирования.

Принцип его работы построен на генераторах, которые пришли из ES6

Генераторы ES6

Генераторы могут порождать (yield) множество значений одно за другим, по мере необходимости. Генераторы отлично работают с перебираемыми объектами и позволяют легко создавать потоки данных.

1. Создаем функцию-генератор

function\* createGen() {

**yield 'start';** // gen.next() - вывел первое значение «start»

**const a = yield;** // gen.next() - объявил переменную a, остановился на этом шаге, жду данные для переменной а.

**console.log("a >>", a)** // gen.next() - как только получил данные для а, показал их, объявил переменную b

**const b = yield; //** gen.next() - как только получил данные для b, показал их,

**console.log("b >>", b) //** gen.next() остановился, показал сумму a + b

**yield a + b; //** gen.next() вернул «end»

**return 'end';**

**};**

1. Объявляем переменную и присваиваем ей результат выполнения функции-генератора. На данном этапе при запуске функция *возвращает генератор* (объект).

let gen = createGen()

1. Используя метод .next() получаем возвращаемые значения:

let one = gen.next()

*На выходе получаем объект:*

*{value: 'start', done: false}*

* 1. done: false
  2. value: "start"

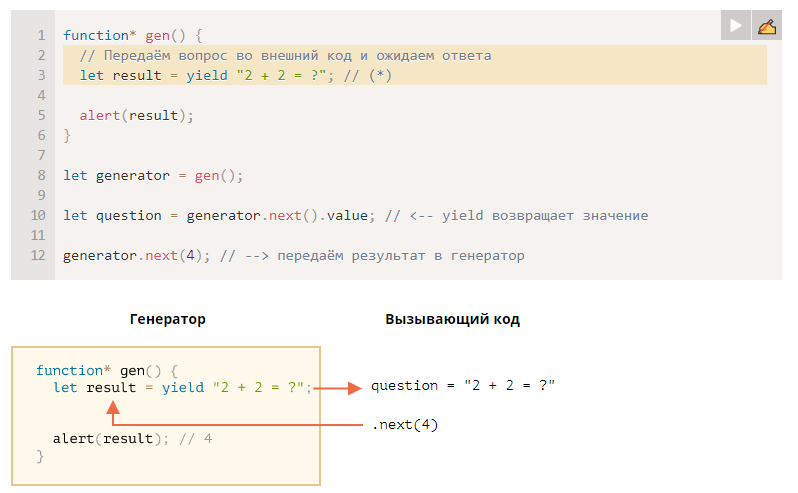
1. Когда код доходит до слова yield, она останавливает свое выполнение.

*// const a = yield; - здесь при следующем вызове next() надо передать значение переменной:*

gen.next(10) // теперь присвоено значение a = 10

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

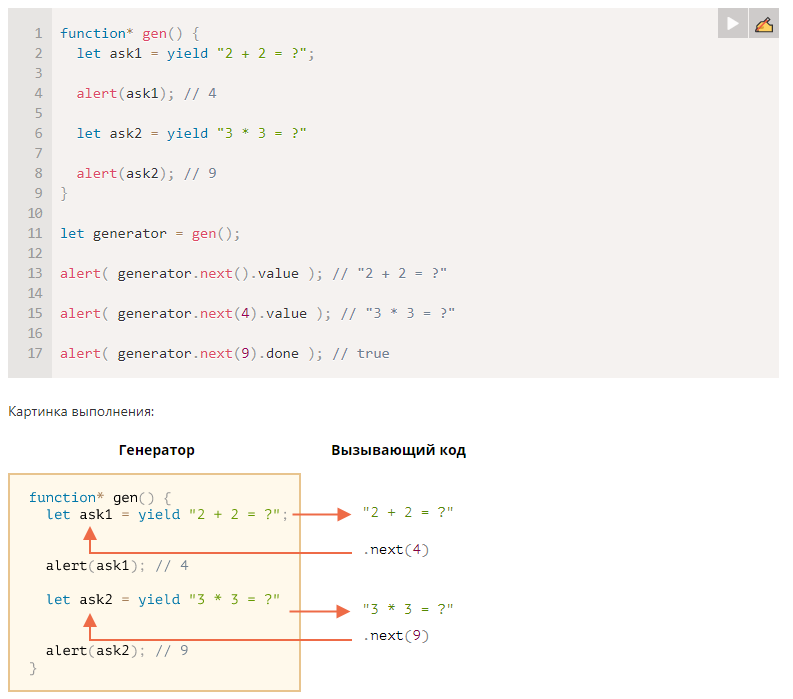
Также, прием генератором значений через yield демонстрирует следующий пример:



1. Первый вызов generator.next() – всегда без аргумента, он начинает выполнение и возвращает результат первого yield "2+2=?". На этой точке генератор приостанавливает выполнение.
2. Затем, как показано на картинке выше, результат yield переходит во внешний код в переменную question.
3. При generator.next(4) выполнение генератора возобновляется, а 4 выходит из присваивания как результат: let result = 4.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Или еще один пример:



1. Первый .next() начинает выполнение… Оно доходит до первого yield.
2. Результат возвращается во внешний код.
3. Второй .next(4) передаёт 4 обратно в генератор как результат первого yield и возобновляет выполнение.
4. …Оно доходит до второго yield, который станет результатом .next(4).
5. Третий next(9) передаёт 9 в генератор как результат второго yield и возобновляет выполнение, которое завершается окончанием функции, так что done: true.

*Получается такой «пинг-понг»: каждый next(value) передаёт в генератор значение, которое становится результатом текущего yield, возобновляет выполнение и получает выражение из следующего yield.*

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

## [generator.throw](https://learn.javascript.ru/generators" \l "generator-throw)

Для того, чтобы передать ошибку в yield, нам нужно вызвать generator.throw(err). В таком случае исключение err возникнет на строке с yield.

function\* gen() {

try {

let result = yield "2 + 2 = ?"; // (1)

alert("Выполнение программы не дойдёт до этой строки, потому что выше возникнет исключение");

} catch(e) {

alert(e); // покажет ошибку

}

}

let generator = gen();

let question = generator.next().value;

generator.throw(new Error("Ответ не найден в моей базе данных")); // (2)

С использованием конструкции try… catch:

function\* generate() {

let result = yield "2 + 2 = ?"; // Ошибка в этой строке

}

let generator = generate();

let question = generator.next().value;

try {

generator.throw(new Error("Ответ не найден в моей базе данных"));

} catch(e) {

alert(e); // покажет ошибку

}

Использование REDUX SAGA

Подключаем библиотеки:

**yarn add** **redux redux-saga react-redux**

**Подключаем Redux:**

* + - Провайдер стора:

ReactDOM.render(

    <*React.StrictMode*>

        <*Provider* store={store}>

            <*App*/>

        </*Provider*>

    </*React.StrictMode*>,

    document.getElementById('root')

);

- Создаем стор с помощью Redux

*onst* sagaMiddleWare = createSagaMiddleware();

*const* store = createStore(reducer,

    window.\_\_REDUX\_DEVTOOLS\_EXTENSION\_COMPOSE\_\_( // чтобы работал Devtools

        applyMiddleware(sagaMiddleWare)

    )

);

sagaMiddleWare.run(rootSaga);

export default store

* + - Создаем Reducer

export default *function* reducer(*state*={}, *action*) {

    return *state*

}

(см файлы в гитхабе)

Далее подключаем саму САГУ:

1. Добавление Saga Middleware. Для этого используем Middleware API, а именно createSagaMiddleware(options). Этот метод создаем Redux Middleware и соединяет саги с Redux Store:

const sagaMiddleWare = createSagaMiddleware();

1. Создаем сагу:

export default function\* rootSaga() {  
 *console*.log('Saga is ready!')  
}

1. Запускаем **КОРНЕВУЮ** сагу (далее можно создавать дочерние саги в виде дерева), только после вызова applyMiddleware (в createStore прописывается вторым параметром):

sagaMiddleWare.run(rootSaga);

Саги делятся на несколько типов:

1. SAGA **WATCHER** – следит за dispatch экшена и, когда происходит какой-либо экшен, запускает worker. *В них описывается какие-то экшены*.

1. SAGA **WORKER –** выполняет бизнес-логику (запрос/таймаут/запись в кэш и пр.). Запускается в зависимости от какого-либо выполненного экшена. *В них описываем бизнес-логику приложения. При тестировании саг, именно их тестируют.*
2. **ROOT** SAGA – всего лишь запускает watcher:

export default function\* rootSaga() {  
 *console*.log('Saga is ready!')  
 yield watchClickSaga();  
}

эффекты:

ПЕРЕД КАЖДЫМ ЭФФЕКТОМ САГИ НУЖЕН yield !!!

Добавляем импорт, например:

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

**Эффекты** – вспомогательные функции, которые создают простые объекты. Эти объекты содержат инструкции, которые выполняются самой reduxSagaMiddleware. По сути, эффект прокидывает в Redux Saga инструкции, которые ей нужно выполнять.

take() – ждать выполнение указанного действия. Т.е. жди какой-либо диспатч в приложении, и после диспатча можно выполнить инструкцию.

export function\* watchClickSaga() {  
 yield take('CLICK'); // значит: пока не выполнится CLICK код ниже выполнен не будет.  
 *console*.log('mouse click')  
}

Для демонстрации добавим кнопку в приложении. При клике на нее в консоли будет появляться сообщение mouse click, но только один раз!

function App() {  
  
 const store = useSelector(*store* => *store*);  
 const dispatch = useDispatch();  
  
 return (  
 <div>  
 **Redux saga** <button onClick={() => dispatch({type: 'CLICK'})}> **CLICK ME**</button>  
 </div>  
 );  
}

Для того, чтобы изменения диспатчились при каждом клике, нужно зациклить генератор с помощью конструкции while:

export function\* watchClickSaga() {  
 while (true) {  
 yield take('CLICK'); // значит: пока не выполнится CLICK код ниже выполнен не будет.  
 *console*.log('mouse click')  
 }  
}

Но правильнее будет перенести бизнес-логику в worker, из watcher вызываем worker при этом :

export function\* workerSaga() {  
 *console*.log('mouse click')  
}  
  
export function\* watchClickSaga() {  
 while (true) {  
 yield take('CLICK'); // значит: пока не выполнится CLICK код ниже выполнен не будет.  
 yield workerSaga();  
 }  
}

Чтобы не писать каждый раз циклы или конструкции типа while, есть несколько удобных эффектов:

takeEvery() – ждать выполнение указанного действия. Т.е. жди какой-либо диспатч в приложении, и после диспатча можно выполнить инструкцию.

СТО РАЗ КЛИКНУЛ – СТО РАЗ СРАБОТАЛО

Импортируем ее из редакс-саги:

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

Имплементим ее, передаем ей ACTION и ФУНКЦИЮ, которая будет вызвана при action:

export *function*\* watchClickSaga() {

    // while (true) {

    //     yield take('CLICK'); // значит: пока не выполнится CLICK код ниже выполнен не будет.

    //     yield workerSaga();

    // }

    yield takeEvery('CLICK', workerSaga) // вместо кода выше

}

takeLatest() – *автоматически отменяет любую предыдущую задачу саги, запущенную ранее, если она все еще выполняется.* Т.е. из нескольких асинхронных кликов или действий, вызовется только последний. Для демонстрации добавим промис, который будет ждать. В примере ниже сработает только последний console.log. СТО РАЗ КЛИКНУЛ – СРАБОТАЛ ПОСЛЕДНИЙ

*const* wait = (*t*) *=>* new *Promise*((resolve *=>* {

    setTimeout(resolve, *t*)

}))

export *function*\* workerSaga() {

    yield wait(1000)

    console.log('mouse click')

}

export *function*\* watchClickSaga() {

    // while (true) {

    //     yield take('CLICK'); // значит: пока не выполнится CLICK код ниже выполнен не будет.

    //     yield workerSaga();

    // }

    yield takeLatest('CLICK', workerSaga)

}

takeLeading() – работает наоборот от takeLatest. Выполняет первое событие, остальные игнорирует. *Т.е. автоматически отменяет любую следующую задачу саги, запущенную позднее, если первая запущенная еще выполняется. СТО РАЗ КЛИКНУЛ – СРАБОТАЛ ТОЛЬКО ПЕРВЫЙ, ЕСЛИ ЕЩЕ ВЫПОЛНЯЕТСЯ .* В примере ниже будет резолвиться только первый промис:

*const* wait = (*t*) *=>* new *Promise*((resolve *=>* {

    setTimeout(resolve, *t*)

}))

export *function*\* workerSaga() {

    yield wait(1000)

    console.log('mouse click')

}

export *function*\* watchClickSaga() {

    // while (true) {

    //     yield take('CLICK'); // значит: пока не выполнится CLICK код ниже выполнен не будет.

    //     yield workerSaga();

    // }

    yield takeLeading('CLICK', workerSaga)

}

Запросы на сервер:

Будем использовать сервис starwars.api

Чтобы отправить запрос на сервер мы **внутри WORKER** пишем следующее:

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

async *function* getPeople () {

*const* request = await fetch('http://swapi.dev/api/people')

*const* data = request.json();

    return data

}

export *function*\* workerSaga() {

*const* data = yield getPeople()

    console.log(data)

}

export *function*\* watchClickSaga() {

    yield takeEvery('CLICK', workerSaga)

}

export default *function*\* rootSaga() {

    console.log('Saga is ready!')

    yield watchClickSaga();

}

Далее закинем информацию от сервера в стор. Для этого в сторе создадим initial state и action:

*const* initial = {

  people: []

}

export default *function* reducer (*state* = initial, *action*) {

  switch (*action*.type) {

    case 'SET\_PEOPLE':

      return { ...*state*, people: [...*state*.people, *action*.payload] }

    default:

      return *state*

  }

}

Далее будем диспатчить людей в стор. Для этого будем использовать эффект

put() – вызывает dispatch с переданным action (диспатчит экшен в редьюсер)

export *function*\* workerSaga() {

*const* data = yield getPeople()

    yield put({type: "SET\_PEOPLE", payload: data})

}

Но данный код сложно тестировать. Поэтому в Redux saga есть свой эффект call

call() – выполняет переданную функцию. Если функция вернет Promise, то приостанавливает сагу до тех пор, пока Promise не зарезолвится.

В call() можно передать *простую функцию, асинхронную функцию, другой генератор.*

***Вторым параметром***можно передать***аргументы***для вызываемой функции

export *function*\* workerSaga() {

*const* data = yield call(getPeople, “Mike”)

    yield put({type: "SET\_PEOPLE", payload: data.results})

}

Поочередная загрузка:

В примере ниже мы получаем два ответа от сервера по очереди. Пока первый не завершится, второй запрос не отправится. При этом все эффекты саги нужно разделять на

**БЛОКИРУЮЩИЕ ЭФФЕКТЫ (take, call)** – тем или иным способом блокируют выполнение кода

**НЕБЛОКИРУЮЩИЕ ЭФФЕКТЫ (fork, spawn, select)**

*ЗАПРОСЫ ВЫПОЛНЯЮТСЯ ПОСЕДОВАТЕЛЬНО,* ***СИНХРОННО***

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

async *function* swapiGet (*pattern*) {

*const* request = await fetch(`http://swapi.dev/api/${*pattern*}`)

*const* data = request.json()

  return data

}

export *function* \* workerSaga () {

*const* people = yield call(swapiGet, 'people') // первый запрос. Call блокирует выполнение кода, пока не зарезолвится промис

  console.log('people', people)

*const* planets = yield call(swapiGet, 'planets') // второй запрос. Call блокирует выполнение кода, пока не зарезолвится промис

  console.log('planets', planets)

  yield put({ type: 'SET\_PEOPLE', payload: people.results })

  yield put({ type: 'SET\_PLANETS', payload: planets.results })

}

export *function* \* watchClickSaga () {

  yield takeEvery('LOAD\_DATA', workerSaga)

}

export default *function* \* rootSaga () {

  console.log('Saga is ready!')

  yield watchClickSaga()

}

Если надо, чтобы запросы (сайд-эффекты) выполнялись **АСИНХРОННО**, будем использовать эффект fork.

Fork() – эффект, который указываем middleware выполнить неблокирующий вызов переданной функции. *Управляет параллелизмами между сагами!*

Для демонстрации будем использовать сразу две саги!

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

async *function* swapiGet (*pattern*) {

*const* request = await fetch(`http://swapi.dev/api/${*pattern*}`)

*const* data = request.json()

  return data

}

export *function*\* loadPeople() { // первая сага

*const* people = yield call(swapiGet, 'people')

    console.log('people', people)

    yield put({ type: 'SET\_PEOPLE', payload: people.results })

}

export *function*\* loadPlanets() { // вторая сага

*const* planets = yield fork(swapiGet, 'planets')

    console.log('planets', planets)

    yield put({ type: 'SET\_PLANETS', payload: planets.results })

}

// WORKER SAGA

export *function* \* workerSaga () {

  yield fork(loadPeople) // выполняется одновременно. Если поменять на call, то выполняться будет последовательно

  yield fork(loadPlanets) // выполняется одновременно. Если поменять на call, то выполняться будет последовательно

}

// WATCHER SAGA

export *function* \* watchClickSaga () {

  yield takeEvery('LOAD\_DATA', workerSaga)

}

// ROOT SAGA

export default *function* \* rootSaga () {

  console.log('Saga is ready!')

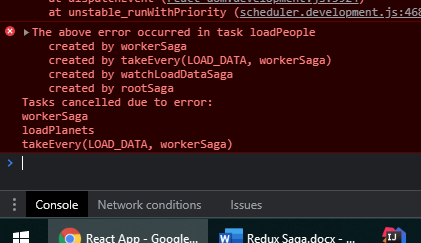
  yield fork(watchClickSaga) // добавили здесь fork, просто так правильнее, поведение программы не поменяется.

}

Особенности fork().

Любая **форкнутая** функция прикреплена к своему родителю, поэтому ошибка, которая возникает в дочернем элементе, она всплывает к родителям и может быть перехвачена в родительской. Также при этом родительская задача блокируется и больше не будет выполнена.

*В ошибке в консоли можно увидеть родителя и потомков, а также отмененные таски.*

**

*Чтобы ошибки не всплывали к родителю, нужно использовать другой оператор – spawn()*

spawn() – создает параллельную задачу в корне саги, сам процесс при этом не привязан к родителю.

*Применим вместо call эффект spawn(). Теперь ошибка в одной саге не вызовет остановку работы второй саги.*

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

async *function* swapiGet (*pattern*) {

*const* request = await fetch(`http://swapi.dev/api/${*pattern*}`)

*const* data = request.json()

  return data

}

export *function*\* loadPeople() {

    throw new *Error*()

*const* people = yield call(swapiGet, 'people')

    console.log('people', people)

    yield put({ type: 'SET\_PEOPLE', payload: people.results })

}

export *function*\* loadPlanets() {

*const* planets = yield fork(swapiGet, 'planets')

    console.log('planets', planets)

    yield put({ type: 'SET\_PLANETS', payload: planets.results })

}

// WORKER SAGA

export *function* \* workerSaga () {

  yield spawn(loadPlanets)

  yield spawn(loadPeople)

}

// WATCHER SAGA

export *function* \* watchLoadDataSaga () {

  yield takeEvery('LOAD\_DATA', workerSaga)

}

// ROOT SAGA

export default *function* \* rootSaga () {

  console.log('Saga is ready!')

  yield fork(watchLoadDataSaga)

}

Оператор join().

Часто возникают ситуации, когда нужно дождаться ответ от сервера задиспатчить изменения. Но часто бывает, когда нужно получить результат и сделать что-то еще. Для этого используется оператор join.

Join – заблокировать неблокирующую задачу и получить ее результат.

Например, пока не выполнится loadPeople, console log не будет

import { takeEvery, put, call, fork, spawn, join } from 'redux-saga/effects'

async *function* swapiGet (*pattern*) {

*const* request = await fetch(`http://swapi.dev/api/${*pattern*}`)

*const* data = request.json()

  return data

}

export *function*\* loadPeople() {

*const* people = yield call(swapiGet, 'people')

    console.log('people', people)

    yield put({ type: 'SET\_PEOPLE', payload: people.results })

}

export *function*\* loadPlanets() {

*const* planets = yield fork(swapiGet, 'planets')

    console.log('planets', planets)

    yield put({ type: 'SET\_PLANETS', payload: planets.results })

}

// WORKER SAGA

export *function* \* workerSaga () {

*const* task = yield fork(loadPeople)

  yield spawn(loadPlanets)

*const* people = yield join(task) // в этом случае  join является блокирующим вызовом

  console.log('join people', people)

}

// WATCHER SAGA

export *function* \* watchLoadDataSaga () {

  yield takeEvery('LOAD\_DATA', workerSaga)

}

// ROOT SAGA

export default *function* \* rootSaga () {

  console.log('Saga is ready!')

  yield fork(watchLoadDataSaga)

}

Оператор select().

Select() – получить данные из store, аналог useSelect() или mapStateToProps из connect.

*const* store = yield select(*store* *=>* *store*)

Например:

import { takeEvery, put, call, fork, spawn, join } from 'redux-saga/effects'

async *function* swapiGet (*pattern*) {

*const* request = await fetch(`http://swapi.dev/api/${*pattern*}`)

*const* data = request.json()

  return data

}

export *function*\* loadPeople() {

*const* people = yield call(swapiGet, 'people')

    console.log('people', people)

    yield put({ type: 'SET\_PEOPLE', payload: people.results })

}

export *function*\* loadPlanets() {

*const* planets = yield fork(swapiGet, 'planets')

    console.log('planets', planets)

    yield put({ type: 'SET\_PLANETS', payload: planets.results })

}

// WORKER SAGA

export *function* \* workerSaga () {

*const* task = yield fork(loadPeople)

  yield spawn(loadPlanets)

*const* people = yield join(task) // в этом случае  join является блокирующим вызовом

*const* store = yield select(*store* *=>* *store*)

  console.log('join people', people)

}

// WATCHER SAGA

export *function* \* watchLoadDataSaga () {

  yield takeEvery('LOAD\_DATA', workerSaga)

}

// ROOT SAGA

export default *function* \* rootSaga () {

  console.log('Saga is ready!')

  yield fork(watchLoadDataSaga)

}

Лучшей практикой является держать отдельно store от saga.

КОМБИНАТОРЫ ЭФФЕКТОВ

В Проекте используются комбинаторы эффектов all.

all – комбинатор, который говорит middlware запустить несколько эффектов сразу. Есть два вида.

- all([…effects]) - массив эффектов *он как Promise.all*, эффекты *выполняются параллельно*, будет ждать *завершения всех*.

Запускаются два блокирующих вызова параллельно:

*import* { fetchCustomers, fetchProducts } *from* './path/to/api'

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

*function*\* mySaga() {

*const* [customers, products] = *yield* all([

call(fetchCustomers),

call(fetchProducts)

])

}

- all(effects) - эффекты *выполняются параллельно*, есть доступ к объекту.

*import* { fetchCustomers, fetchProducts } *from* './path/to/api'

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

*function*\* mySaga() {

*const* { customers, products } = *yield* all({

customers: call(fetchCustomers),

products: call(fetchProducts)

})

}