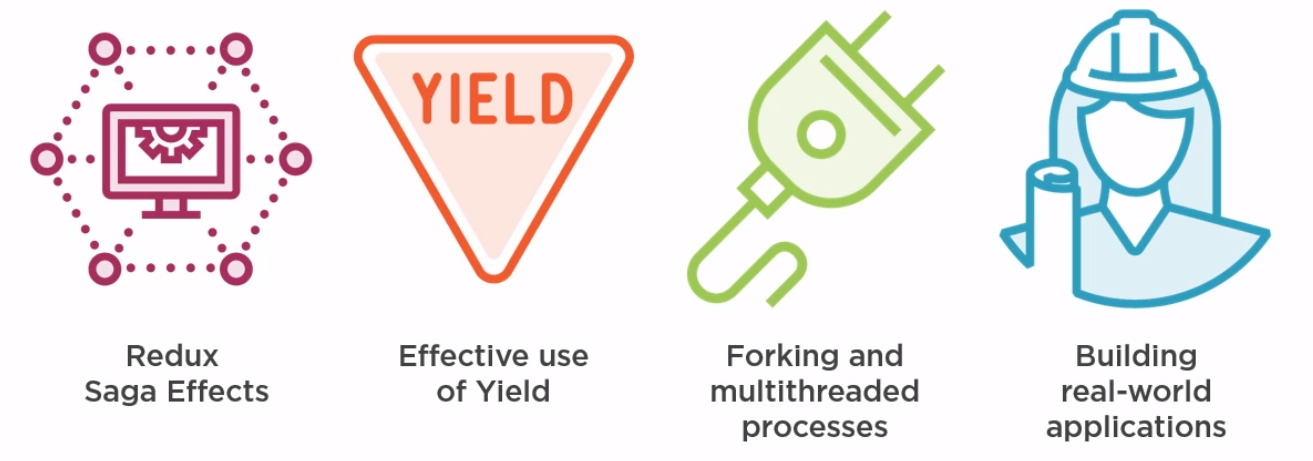
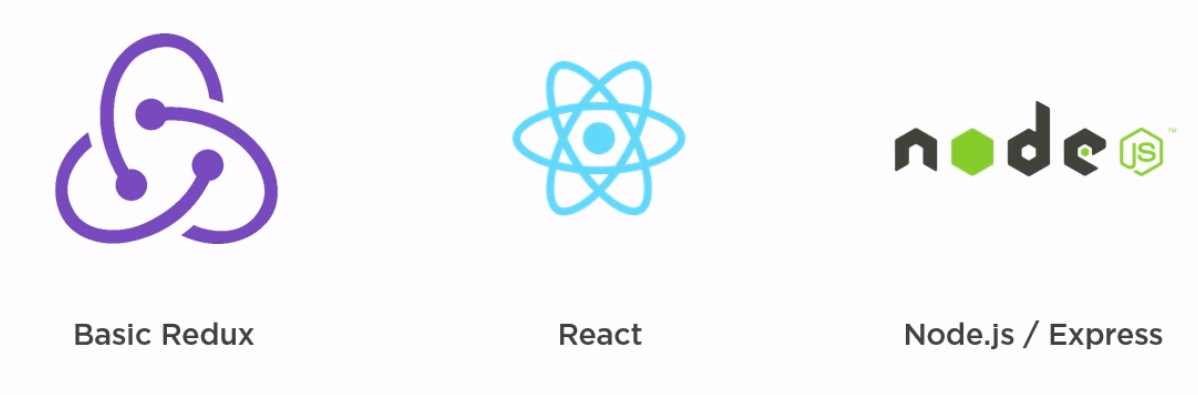
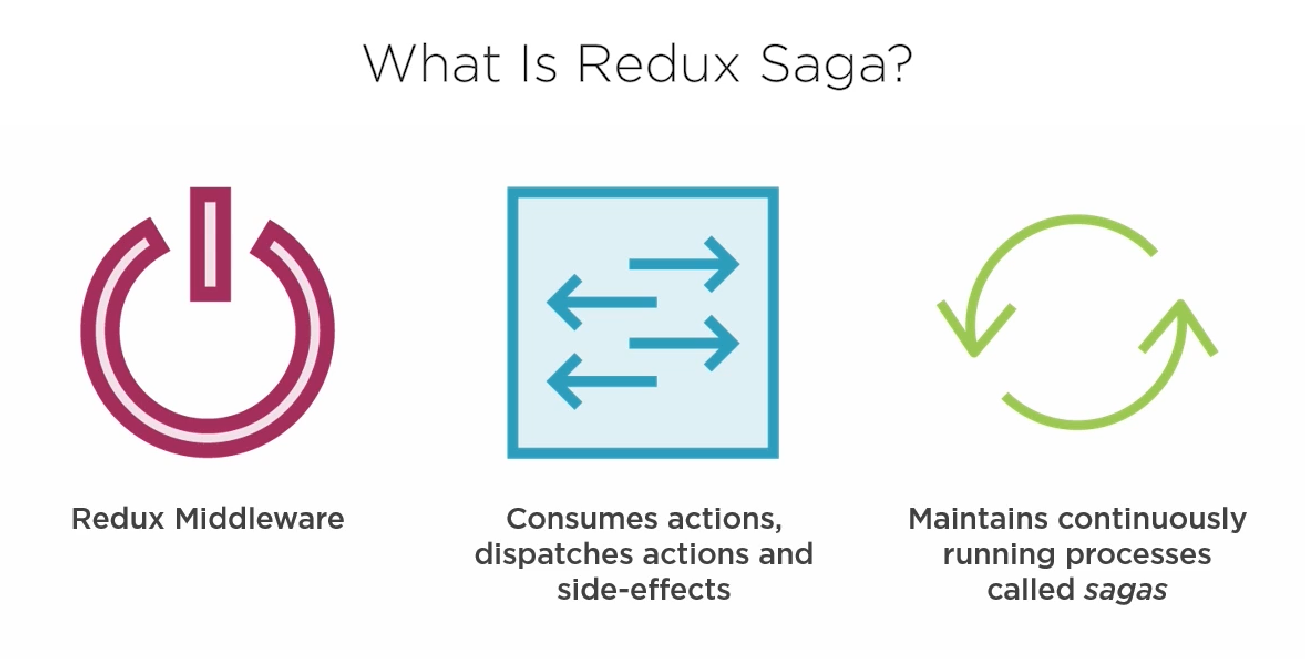
1. **Overview:**

**\_** Sẽ học :



\_ Bắt buộc :





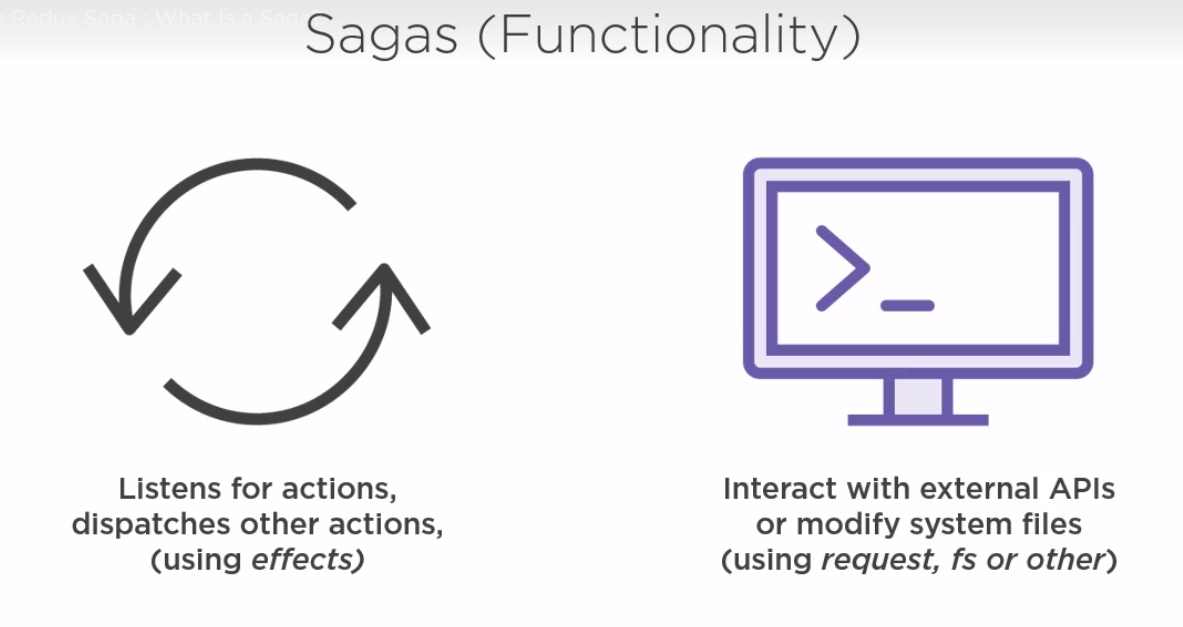
\_ Redux saga: là 1 redux middleware, có nghĩa nó cần được thêm vào redux store, nếu có nhiều middleware thì chain lại ( kết hợp lại ).

\_ Redux saga: nhận action, dispatches actions ( thực thi ) và tạo side effects. Side effects bao gồm : giao tiếp database, làm việc với api, …

**Side effect**: Ta đã biết tất cả những xử lý ở REDUCER đều phải là synchronous và pure tức chỉ là xử lý đồng bộ. Nhưng trong ứng dụng thực tế thì cần nhiều hơn vậy ví dụ như asynchronous (thực hiện một số việc như gọi một hàm AJAX để fetch dữ liệu về nhưng cần đợi kết quả chứ kết quả không trả về ngay được) hoặc là impure (thực hiện lưu, đọc dữ liệu ra bên ngoài như lưu dữ liệu ra ổ cứng hay đọc cookie từ trình duyệt… đều cần đợi kết quả). Những việc như thế trong lập trình hàm gọi nó là side effects.

**\_** Khác với redux-thunk : Redux Saga duy trì các quy trình chạy liên tục gọi là các sagas, redux saga giúp quản lý các logic của ứng dụng,

\_ **Saga:** là 1 pattern. Lắng nghe action, dispatches actions khác, sử dụng effects.



\_ Sagas ( functional programming ) : đảo ngược phiên giao dịch transaction. Sagas được thiết kế để thay thế các giao dịch đơn lẻ, lớn, khóa giao dịch.

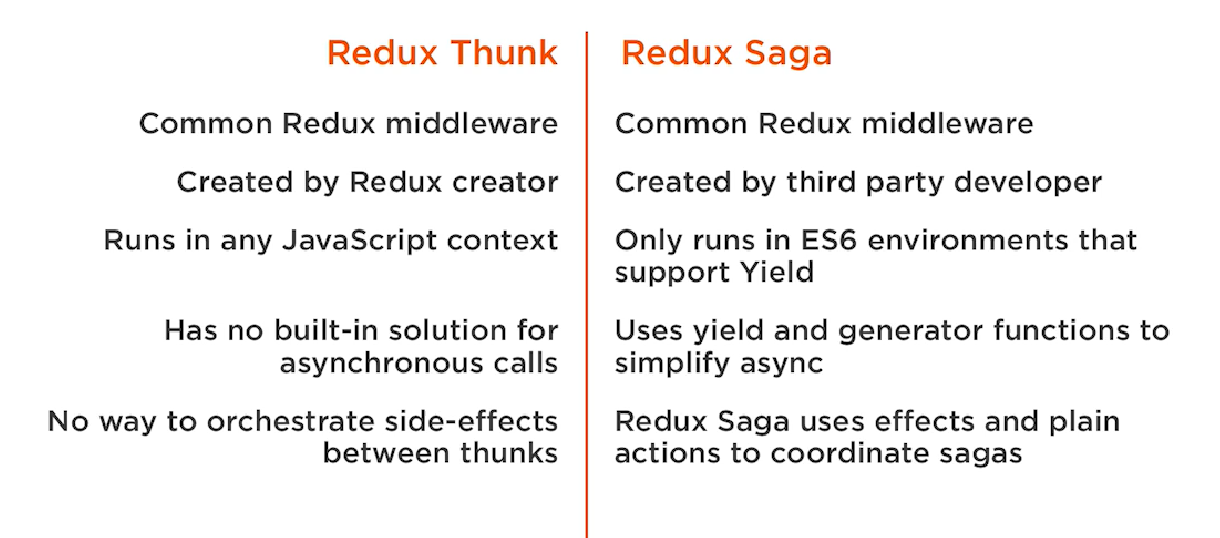
\_ **Tại sao dùng ??? Tại sao không dùng redux thunk thay thế ??? :**

+ Xử lý side effect ( Call api, db transaction ) : sử dụng **yield** & **generators**

+ Nhiều tool ( forking processes, yielding thread ) : giải quyết vấn đề trong thực tế. Đa luồng: thêm 1 tiến trình vào tiến trình khác, dừng process,

+ Học redux-sagas như là học thêm các solution để giải quyết các bài toán.

1. **Redux Sagas vs Redux Thunk :**



\_ Simplify async : đơn giản hóa xử lý async.

\_ Plain actions: sử dụng effect và action bình thường đến sagas.

1. **Redux saga:**

\_ Trang chủ : [**https://github.com/redux-saga/redux-saga**](https://github.com/redux-saga/redux-saga) hoặc [**https://redux-saga.js.org/**](https://redux-saga.js.org/)

\_ Cài đặt redux saga: npm install redux-saga –save hoặc yarn add saga

\_ Tích hơp vào store: ( redux ) – vì redux saga là 1 middleware

B1: import **createSagaMiddleware** từ **‘redux-saga’**

B2: Tạo middleware bằng createSagaMiddleware()

B3: Tại applyMiddlware của redux bổ sung middleware sagas.

**3.1 Tạo 1 saga đơn giản đầu tiên( không bao gồm các chức năng bất đồng bộ asynchonous ):**

\_ Tổ chức thư mục sagas ( trả về index cho store ) và chạy sử dụng redux saga middleware

\_ Kiểm chứng saga có chạy hay không ?

* 1. **Thực hiện :**

\_ Tạo thư mục sagas: chứa các saga

\_ Tạo saga: ví dụ currentUserSaga.js: export 1 function có dấu \* ở sau function

+ Là 1 generator function.

+ Tên trùng tên file ( tự quy định ).

\_ Nội dung trong function cho vào while(true) : nếu lỗi sẽ ra khỏi while

\_ Nội dung trong: **ta sẽ dừng những đoạn code trong 1 khoảng thời gian.** Sử dụng hàm tiện ích **delay** trong redux saga: import { delay } from ‘redux-saga’

\_ Ta sử dụng **yield** keyword: NHỮNG ĐOẠN CODE TIẾP THEO SẼ KHÔNG CHẠY CHO ĐẾN KHI dòng code có từ khóa **yield** này chạy xong : **yield delay(1000)** vào function và sau **yield** này ta log ra 1 đoạn thông báo.

\_ **Tạo index root saga** ( đặt trong thư mục sagas ) **:** export { **namedSaga** } from ‘…’

\_ Tạo file khởi tạo và chạy saga ( đặt cùng cấp store ) :

+ import index root saga vào. ( **import \* as sagas from ‘./sagas’)**

**+** export 1 function init saga: dùng object.values lấy tất cả giá trị của sagas được export ở index và duyệt qua và gọi sagaMiddleware.run() để chạy. ( lưu ý để không sai **scope** ta gọi hàm .run.bind(sagaMiddleware)) \*\* scope js khá quan trọng =>

**+** .bind để đảm bảo giữ đúng scope. ( Chỉ có function mới gọi bind ( là 1 prototype của func ) – dùng để xác định tham số *this* cho function )

**+**

**Export const initSagas = (sagaMiddleware) => {**

**Object.values(sagas).forEach(sagaMiddleware.run.bind(sagaMiddleware))**

**}**

// Cách viết khác: value => sagaMiddleware.run(value).bind(sagaMiddleware)

\_ Cuối cùng: tại store ta import file khởi tạo vào : import { initSagas } from ‘./…’ và gọi lại hàm init : initSagas(**sagaMiddleware**)

\_ Quan sát ta thấy là saga cứ chạy và ngăn mọi hành động render lại. Khi nào saga chạy xong mới được chạy. Từ khóa **yield**

\_ Tóm tắt:

+ Redux saga là sự dung hợp của function programming và async ES6

+ Là middleware quản lý side-effects.

+ Vượt trội về mặt business ứng dụng hơn thunk.

+ Redux saga quản lý tất cả saga và

1. **Yield:**

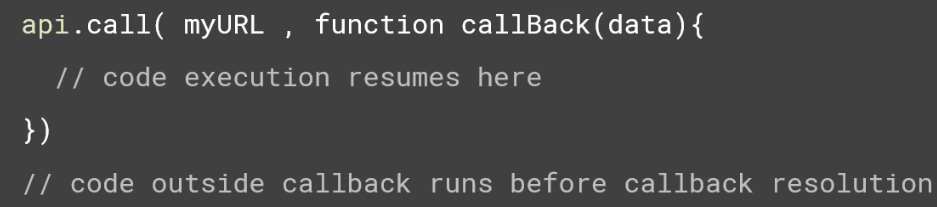
Khái niệm :

\_ Là 1 từ khóa đặc biệt, sẽ trì hoãn việc thực thi những đoạn code tiếp theo.

\_ Chỉ làm việc bên trong generator functions. Ngược lại app sẽ không chạy.

\_ Làm việc với promise và ngừng các đoạn code xung quanh lại.

Ví dụ khi ta làm việc với asynchonous ( bất đồng bộ ) và sử dụng callback function. Ta sẽ không cần **yield**. Cách truyền thống khi làm việc với js.



+ Ví dụ ta muốn sau khi gọi api thành công ta sẽ tiền hành lấy data từ server trả về và thực hiện display lên màn hình. Vậy đoạn code đó ta sẽ đặt bên trong callback. Ngược lại nếu đặt bên ngoài thì ta sẽ không đảm bảo data lấy được thành công ( mà việc này có thể sẽ chạy trước đoạn code gọi API ).

+ Vậy nếu sau khi có data ta tiếp tục gọi api, rồi cứ liên tục lặp như vậy. Code ta sẽ ( drift right nested ) như 1 bậc thang vô tận lồng nhau. Gọi là pyramid of callbacks.

+ Yield sinh ra để giải quyết issue này : Không có đoạn code nào có thể chạy trước promise này.

**let data = yield api.call(myURL);** // promise

**// display data on screen**

**+** Có nghĩa là ta sẽ đảm bảo 100% được việc có data rồi mới display. Code ta sẽ luôn luôn gọn nhẹ hơn so với cách pyramid of callbacks.

\_ Điểm mạnh & yếu :

**Điểm Mạnh**

\_ Số dòng code giảm đáng kể, ít thụt lề

\_ Dễ debug, dễ đọc code.

**Điểm Yếu**

\_ Chỉ làm việc bên trong generator functions

\_ Bắt buộc bổ sung Babel plugins.

\_ Phải xử lý lỗi 1 cách rõ ràng. ( Nếu ko ứng dụng có thể sẽ crash )

\_ Không phổ biến với người mới.

\_ Việc thực thi sẽ dừng lại nếu không xử lý error.

1. **Generator Function:**

\_ Là 1 môi trường đặc biệt để sử dụng được **yield.**

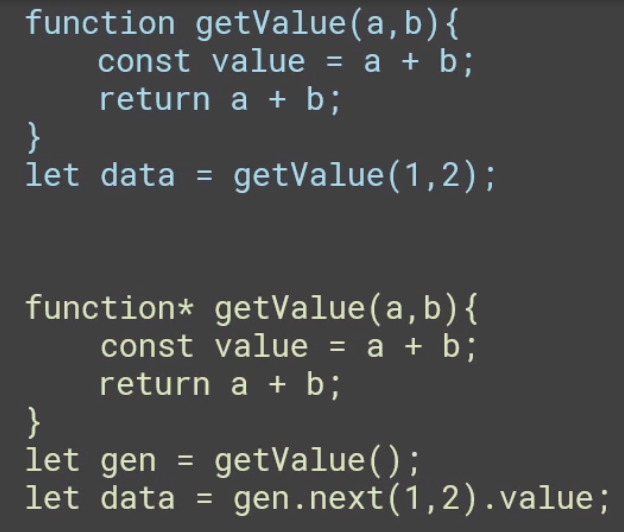
\_ Là 1 function js đặc biệt có thêm ký hiệu **\*** sau từ khóa function.

\_ Gọi hàm này trả về 1 đối tượng generator.

\_ Khi gọi generator function, đoạn code của nó thực ra là không chạy, nó chỉ chạy khi ta gọi phương thức **next()** trong generator object.

\_ Có thể return hoặc **yield** nhiều giá trị ( khác so với function thường )

\_ VD :



**gen:** Là 1 đối tượng generator.

Sau đó generator function được thực thi ta cần gọi next ( …params ) và .value để lấy giá trị.

\_ Tạo 1 generator function đơn giản :

var generatorFunction = function\*() { return 5 }

generatorFunction() // thực thi

generatorFunction().next() // lấy object generator

generatorFunction().next().value // lấy valeu của object generator

\_ Tạo 1 generator function có yield

var generator = function\*(){

**yield 1;**

**yield 2;**

**… 3, 4, 5**

**Return 5;**

}

var obj = generator();

obj.next();

obj.next().value

+ Kết quả ta sẽ nhận được giá trị đầu tiên của yield **Object { value: 1, done: false )**

+ Gọi lần lượt next như vậy ta sẽ được Object của yield cuối cùng và **done** là true.

\_ Viết 1 generator function xử dụng asynchonous : ( sau 1s trả về kết quả 999 )

var generator = function\*() {

yield new Promise( r=> setTimeout(r, 1000) );

return 999;

}

var obj = generator();

obj.next().value.then(v => console.log(obj.next().value))

**\_ Kết hợp Yield & Promises:**

+ Gọi hàm kèm theo từ khóa **yield** phải trả về 1 promise ( hoặc object, đối tượng hợp lệ khác )

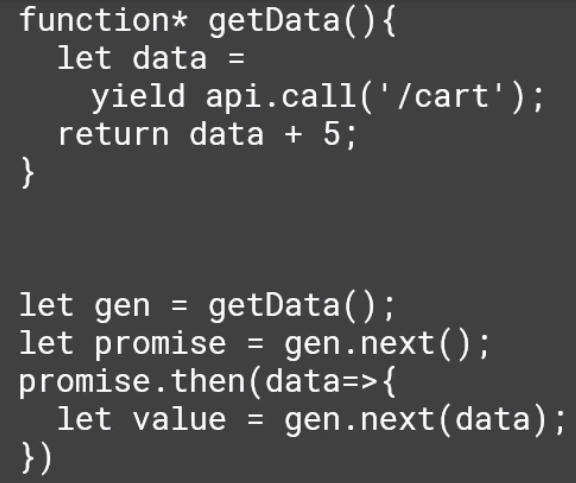
+ Code sẽ được thực thi tiếp tục khi promise đã resolved.

+ Nếu promise quăng lỗi, code sẽ dừng tại **yield** đó.

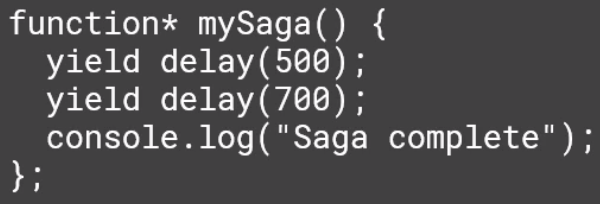
\_ Wrapping Generators:

+ Khi sử dụng **yield** một promise phải được gọi thủ công.

+ Redux saga bao bọc generators 1 cách tự động. ( khi chạy generator trong redux saga, nó sẽ tự wrap generator và tự động resolves promise ).



\_ Còn trong Sagas thì tự động :



\_ Ví dụ :

var generator = function\*() {

var data1= yield delay(1000, 1);

console.log(‘step 1’);

var data2= yield delay(1000, 1);

console.log(‘step 2’);

return data 1 + data2;

}

// let obj = generator().next()

// liên tục để lấy giá trị

+ Để lấy được giá trị, object ta chỉ cần gọi hàm **next**.

+ Khi ta gọi thủ công để lấy giá trị

+ Nhưng nếu ở 1 step nào đó lỗi. Ta sẽ không lấy được kết quả cuối cùng. Hoặc nói cách khác ta muốn gọi xong api => có data để xử lý logic. Và lời gọi phải được thực hiện tự động, chứ không gọi lần lượt thì ta sẽ nhận được ở yield 1 hoặc 2 sẽ là undefined.

+ Sử dụng phương thức **run** của redux-saga :

**run(generator)**

+ Ta sẽ tạo 1 object wrapper : **var wrapped = co.wrap(generator)**

+ Khi ta gọi thực thi **wrapped** function thì nó sẽ tự động chạy tất cả các yield promise và resolve tất cả và trả về giá trị.

**wrapper.then(v => console.log(‘Đã resolve tất cả: ’, v));**

\_ Tóm lại:

+ Yield là từ khóa đặc biệt chỉ xuất hiện trong generator functions.

+ Yield thường được thực hiện bằng việc gọi 1 hàm trả về promise.

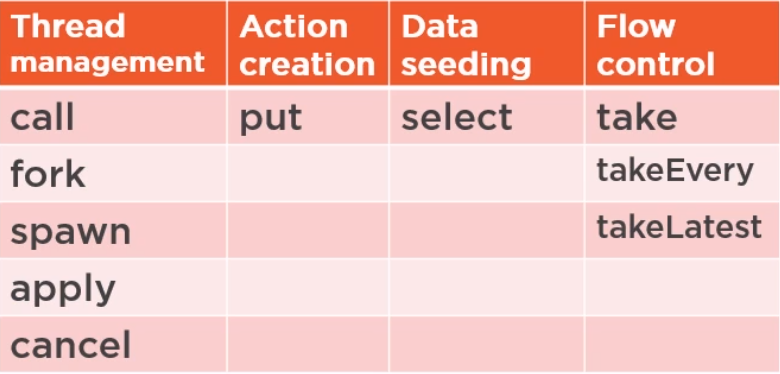
+ Redux-saga bao bọc generators để quản lý promise đơn giản hơn.

1. **Redux Saga Effects:** put, select, take, takeLatest, takeEvery, call, fork, …

\_ Tìm hiểu tất cả redux-saga effect và cancel ( dùng dừng threads cha, con đang chạy )

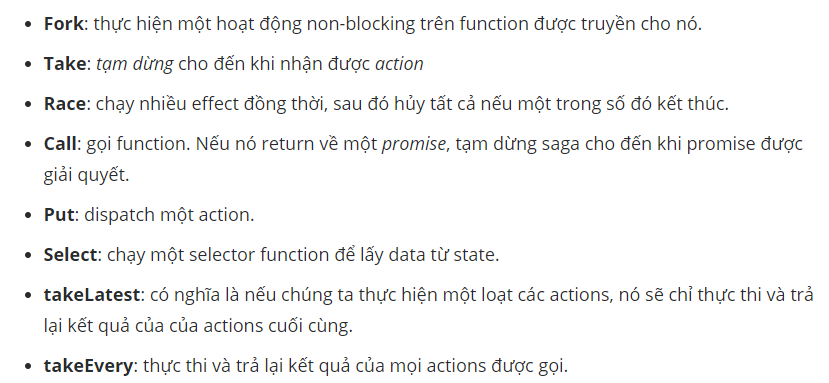


\_ Còn gọi là saga helpers.

\_ Cột 1 – quản lý thread : kết quả là tạo ra thread mới hoặc nhường lại cho thread cũ.

\_ Cột 2 - Action creation : là 1 effect cho việc tạo ra các actions ( dispatch 1 action ).

\_ Cột 3 – Data seeding : là 1 effect cho việc lấy data ( state ) từ app ( select ) .

\_ Cột 4 - Flow-control : dùng dừng đoạn code hoặc hướng sang đoạn code mới dựa trên action được dispatched.   


**6.1: Take – tạm dừng đến khi nhận action**

\_ Dùng dể chờ đợi ( lắng nghe ) một action xảy ra, tham số đầu vào là 1 action ( thường là action types – constants ). Lưu ý : để ràng buộc, kích hoạt một quá trình khi action được dispatched.

\_ Là 1 effect ( 1 helper, 1 operator) : dùng để dừng tại 1 dòng code.

\_ Code chạy tiếp chạy khi ta dispatch 1 action cụ thể.

\_ Affect này không bổ sung thêm thread. Chỉ duy nhât 1 thread – nhiều action không dẫn đến nhiều response.

\_ Bất kỳ thuộc tính nào của action được gửi đi như là yield.

\_ VD dừng code và chỉ chạy khi action đc gọi :

var mySaga = function\*() {

console.log(‘Saga start…’);

const state = yield effects.**take**(‘SET\_STATE’);

console.log(‘State: ’, state);

}

run(mySaga);

* Để code chạy tiếp :

**dispatch({ type: ‘SET\_STATE’, value: 1 });**

\_ Thực tế : nhận kết quả mỗi khi 1 action khác được dispatch

VD : const { items } = yield take(SET\_CART\_ITEMS)

=> Mỗi khi SET\_CART\_ITEMS ta lấy được items.

* 1. **Put :**

\_ Thực thi dispatch 1 action ngay ập tức.

\_ Code thực thi sẽ không dừng lại. ( yield dừng )

\_ Giống như gọi dispatch trong redux-thunk hoặc cách gọi bình thường.

\_ Ví dụ : Dừng code ( dùng take ) cho đến khi action được gọi, được dispatched ( dùng put ).

var mySaga = function\*() {

console.log(‘Saga start…’);

const state = yield effects.**take**(‘SET\_STATE’);

console.log(‘State: ’, state);

}

run(mySaga);

var putSaga = function\*() {

yield effects.put({ type: ‘SET\_STATE’, value: 999 });

}

run(putSaga);

* 1. **Call :**

\_ Giống nhưgọi trực tiếp 1 function ( thức thi function ). Sử dụng để testing. Trả về 1 promise và sẽ **tạm dừng** saga cho đến khi promise resolved.

\_ Thực tế : call api và nhận vào promise resolved.

\_ Ví dụ thực tế: currentUserSaga => action : GET\_CURRENT\_USER

+ Dừng code thực thi cho đến khi GET\_CURRENT\_USER được gọi

**+ currentUserSaga** lấy thông tin user và chờ action **GET\_CURRENT\_USER\_INFO** và thông tin current chỉ cần mỗi khi fetched. => dùng **take**

+ Lấy thông tin user = việc gọi api => dùng call.

+ Cuối cùng dùng PUT để dispatch action lưu thông tin vào store.

export function\* currentUserSaga(){

const { id } = yield take(GET\_CURRENT\_USER\_INFO); // chờ lấy được id

const response = yield call (fetch, API );

*const data = yield apply( response, response.json );*

yield put(setCurrentUser(data)); // setCurrentUser là action creator.

}

* 1. **Fork:**

\_ Gọi những method cụ thể ( như **call** – tham số đầu vào là 1 method )

\_ Không thể truy cập yielded variables

\_ Lời gọi tiếp theo sẽ không bị dừng thực thi.

\_ Sử dụng khi: bắt đầu 1 task **non-blocking.**

VD :

Function \*fn(){

While(true){

Console.log(“abc”);

Yield delay(1000);

}

}

Const saga = function\*(){

While(true){

Yield effects.fork(fn);

Yield delay(500);

}

}

\_ Thường dùng để chĩa ra các nhánh khác nhau để thực thi tiếp.

\_ Thực tế : mỗi khi items trong giỏ hàng thay đổi ta loại lại data của từng item tương ứng

+ Ta lấy id current user.

+ Gọi api lấy cart theo user.

+ Lưu cart item vào store bằng put.

+ Mỗi khi item cart thay đổi, được set ta lấy danh sách item

+ Tương ứng mỗi item ta rẽ các nhánh để tiến hành load thông tin.

+ Gọi api lấy thông tin tương ứng từng item.

+ Sau khi có thông tin ta put action để lưu thông tin vào store.

VD :

export function\* fetchCartSaga(){

const { id } = yield take(SET\_CURRENT\_USER) // Mỗi khi set current user được gọi ta lấy id ( thường SET CURRENT USER sau khi log in )

const resp = yield fetch(API);

const cartitems = yield resp.json();

yield put(setCartItems(items)));

}

export function\* itemDetailsSaga(){

const {items } = yield take(SET\_CART\_ITEMS); // Có nghĩa là setCartItems(items) gọi thì dòng này sẽ thực hiện

yield items.map(item => fork(loadItemDetails, item));

}

function\* loadItemDetail(item){

const { id } = item;

const resp = yield fetch(“API”);

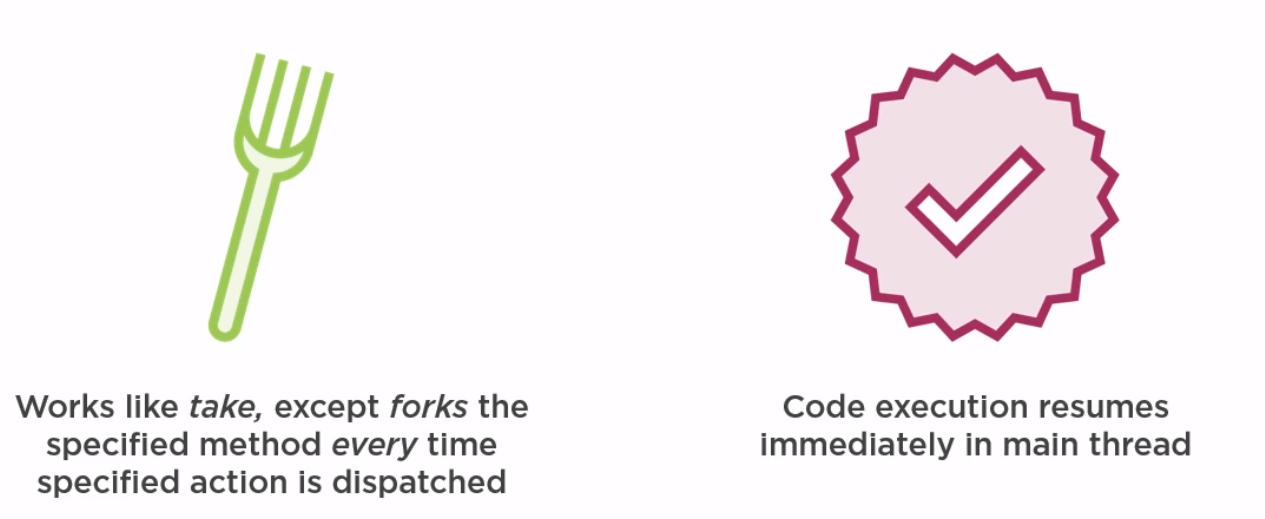
const data = yield resp.json();

const info = data[0];

yield put(setItemDetail(info));

}

**6.5 Take Every :**



\_ Giống như *take,* ngoại trừ forks những phương thức cụ thể mỗi khi action cụ thể được dispatched.

\_ Kết hợp giữa take và fork. Thay vì mỗi lần nhận 1 action, mỗi lần nó forks 1 new child process để xử lý.

\_ Không giống take, nhưng giống fork. Khi gọi **takeEvery**, main thread vẫn tiếp tục được thực thi.

\_ Là 1 phần của fork.

VD :

+ Tạo saga gọi method mỗi khi 1 action được dispatch.

+ Note làm thế nào multiple threads được tạo.

+ Note luồng mà takeEvery được gọi sẽ bắt đầu ngay lập tức.

let process = function\* (){

while(true){

log(“process loop”);

yield delay(1000);

}

}

let saga = function\*(){

yield effects.takeEvery(“START”, process);

log(“Saga end”);

}

Run(saga);

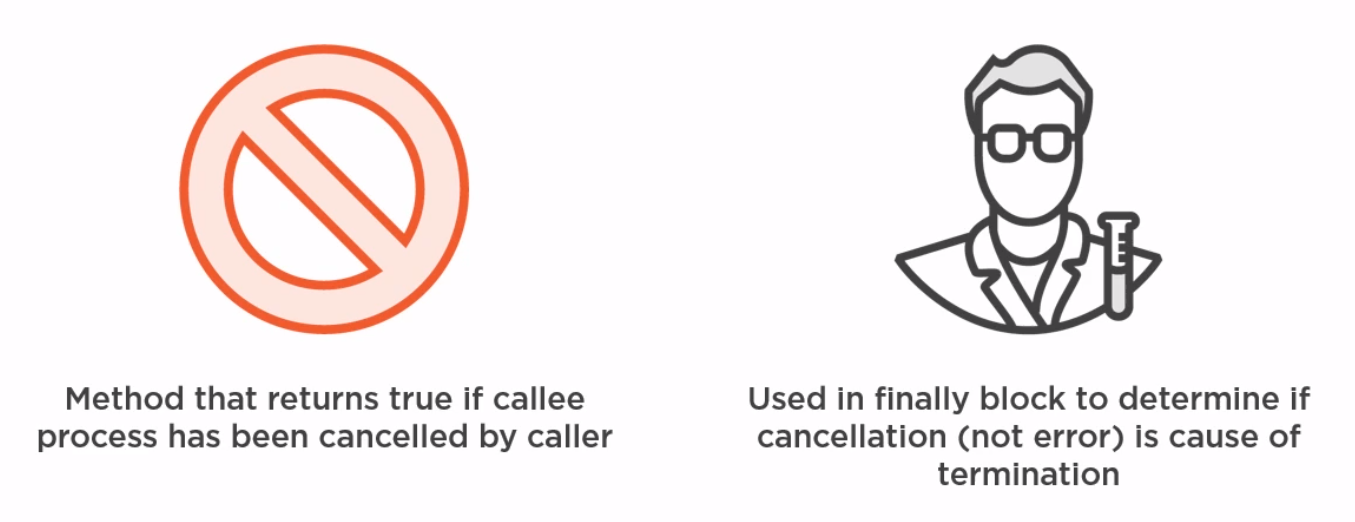
. Kết quả : log saga end. Nhưng saga vẫn đang chờ action START. Mỗi khi dispatch action START => sẽ chạy.

* 1. **Cancel & Cancelled:**

\_ Cancel :



\_ Cancelled :



VD

\_ Tạo 1 forked process, sau đó cancel nó sau 1 khoảng delay.

\_ Lưu ý làm thế nào forked process chạy cho đến khi hủy.

\_ Lưu ý khi nào finally block được gọi.

\_ Cancelled phải được gọi cùng với từ khóa YIELD.

Let process = function \* (){

try{

while(true) {

log(“process looped”);

yield delay(500);

}

} finally {

Const cancelled = yield effects.cancelled();

Log(“cancelled”, cancelled);

}

}

Let saga = fnction\*(){

Let forked = yield effects.fork(process);

Yield delay(5000);

Yield effects.cancel(forked);

Console.log(“DONE!”);

}

RUN(SAGA);

=>Kết quả loop 10 lần và stop.

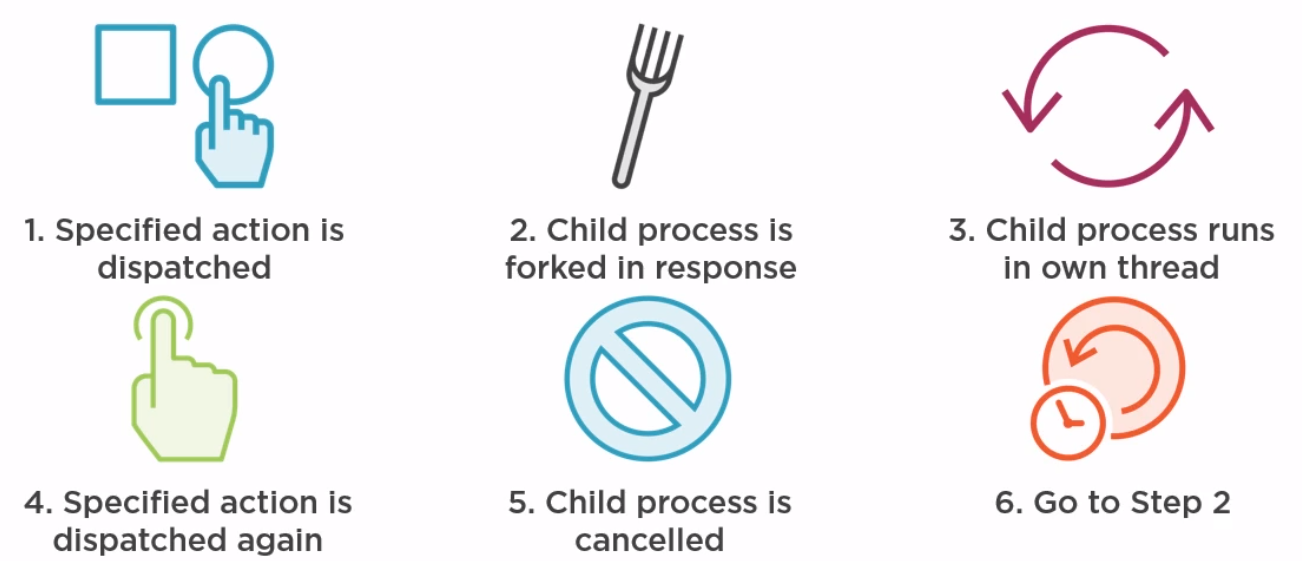
**6.6 TakeLatest :**

\_ Phức tạp nhất, mạnh mẽ nhất.

\_ Sự kết hợp của fork, takeEvery, và cancel.

\_ Sẽ forks child process ( rẽ nhánh sang 1 tiến trình con để xử lý ) mỗi lần action được thực hiện. Trong khi vẫn giữ chính xác 1 instance của child process đang chạy. ( singleton ).

VD : hành động nhấn nút submit nhiều lần chỉ lấy lần cuối cùng.



\_ Đầu tiên ta dispatch action khi click vào button

\_ Thứ 2 : Process con được rẽ ra.

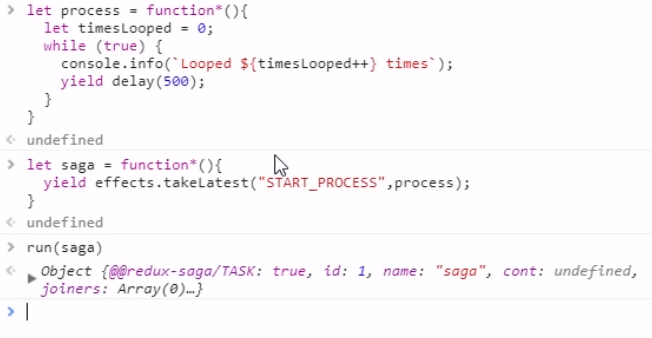
\_ Thứ 3 : Child process chạy trong thread của chúng ta.

\_ Thứ 4 : Dispatch action đó lại lần nữa

\_ Thứ 5 : Process bị cancelled.

\_ Cuối cùng: Quay lại bước 2.

VD :



* Tiến hành dispatch START\_PROCESS
* Quan sát
* Dispatch lần nữa sẽ console.log ra lại từ đầu. ( loop trước đó đã stop )

\_ Áp dụng vào project :

+ Tăng số lượng item trong giỏ hàng. ( hoặc giảm ), Mỗi lần user add hoặc remove item, process đã tồn tại sẽ cancelled và 1 process mới được forked.

\_ Demo :

export function\* **itemQuantitySaga**(){

yield [

takeLatest(DECREASE\_ITEM\_QUANTITY, handleDecreseItemQuantity),

takeLatest(INCREASE\_ITEM\_QUANTITY, handleIncreaseItemQuantity)

]

}

export function **handleIecreseItemQuantity** ({id}){

yield put(setItemQuantityFetchStatus(FETCHING)); // dispatch action

const user = yield select(currentUserSelector); // get state dùng select

const resp = yield call(fetch, api); // gọi api

if (resp.status !== 200){

yield put( decreaseItemQuantity(id, true)); // hết trong kho => giảm lại

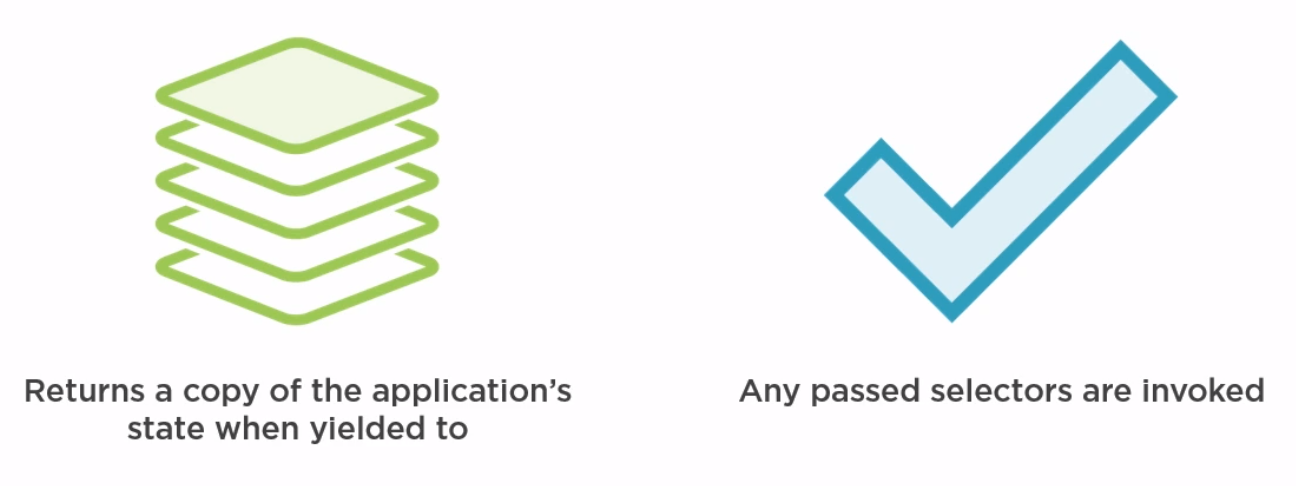
}

yield put(setItemQuantityFetchStatus(FETCHED));

}

Tương tự Decrease. Nhưng không tăng lại khi giảm. ( trường hợp status !== 200)

* 1. **Select :.**

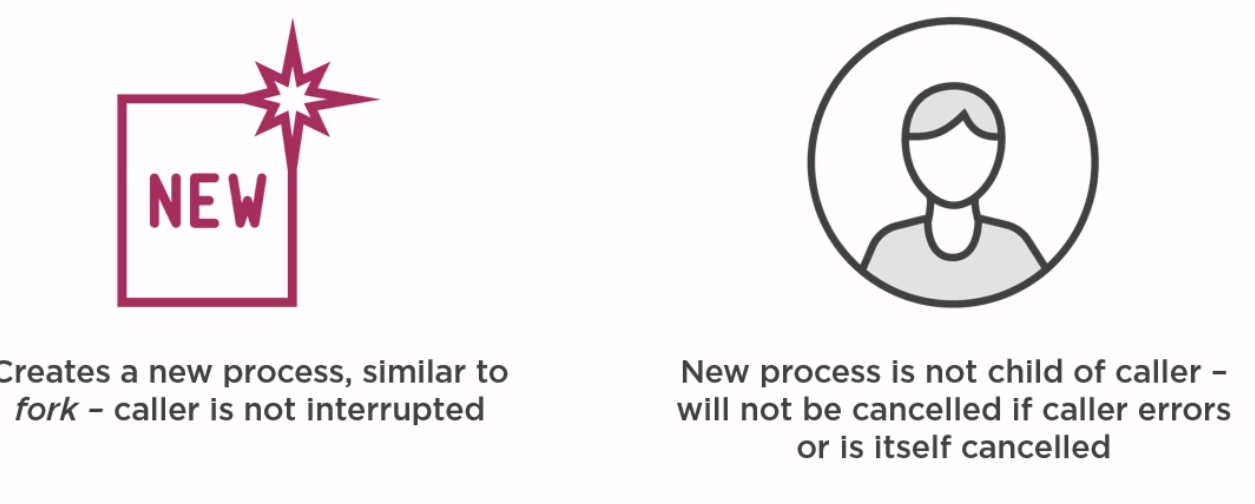


* 1. **Spawn :** Ít phổ biến

\_ Tạo 1 new process, giống như fork.

\_ Người gọi không bị gián đoạn.

\_ New process không phải child của caller, sẽ không bị cancelled nếu lời gọi errors hoặc nó tự cancelled.



VD : tạo saga gọi 1 process và lỗi sau đó. Dùng Spawn nó sẽ tiếp tục run chứ ko dừng lại. Còn fork nếu lỗi là sẽ dừng lại.



Run(saga). Theo dõi sau đ1o thay Spawn = Fork.

* 1. **All**

\_ Kết hợp nhiều lệnh vào một.

\_ Code thực thi bắt đầu khi tất cả actions được dispatched. ( theo thứ tự bất kỳ )

VD : itemPriceSaga:

export function\* itemPriceSaga(){

const [{user}, {items}] = yield all([

take(SET\_CURRENT\_USER),

take(SET\_CART\_ITEMS)

]);

yield items.map(item => call(fetchItemPrice, item.id, user.country));

}

function\* fetchItemPrice(id, currency){

const resp = yield fetch(API);

const json = yield response.json();

const price = json[0].price;

yield put(setItemPrice(id, price));

}

\_ Tóm tắt tất cả effect :

+ Effects tạo ra plain objects – redux saga phiên dịch chúng và thực thi processes.

+ **Take, TakeEvery, TakeLatest** : chờ, lắng nghe 1 action để tạo 1 new process.

+ **Call, Fork, Spawn** : tạo ra các loại khác nhau của new processes.

+ **Forked processes** bị hủy khi parent của nó bị hủy hoặc lỗi.

+ **Take, Call** dừng ( *pause* ) những sự thực thi của caller process.

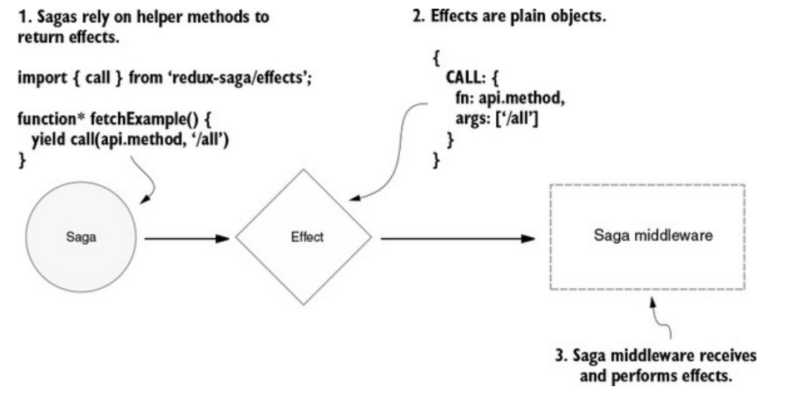
Redux In Action

1 Import **form** từ gói redux-saga / effects package

2 Rẽ nhánh mỗi người theo dõi cho phép rootSaga chuyển sang phần tiếp theo.

3 Mỗi người quan sát cũng là một generator.

Fork đang làm gì ở đây? Khi rootSaga thực hiện, nó sẽ tạm dừng ở mọi câu lệnh có từ khóa **yield** cho đến khi side effect hoàn tất. Tuy nhiên, phương pháp **fork** cho phép rootSaga chuyển sang yield tiếp theo mà không có quyết định. Mỗi loại **fork** này được cho là không bị chặn. Việc triển khai này có ý nghĩa, bởi vì bạn muốn khởi động tất cả người theo dõi khi khởi tạo, không chỉ là người đầu tiên trong danh sách.



Phương thức **call** được sử dụng ở đây tương tự với chức năng **call** của JavaScript. Bạn sẽ sớm sử dụng lại nó để chỉ định yêu cầu AJAX để tìm nạp các tác vụ. Một khi bạn sử dụng các phương thức reduxsaga cụ thể để tạo ra một effect, saga middleware sẽ xử lý và thực hiện các side effect cần thiết bên ngoài View. Bạn sẽ tìm hiểu thêm về các phương pháp này trong việc triển khai nhiệm vụ tìm nạp tiếp theo.

1 Imports take from the effects package

2 Người theo dõi sử dụng vòng lặp vô hạn để xử lý các hành động thường xuyên khi cần.

3 Take chờ đợi một loại hành động nhất định trước khi cho phép saga tiến hành.

\_ Sau khi xử lý side effect cần dispatch những action mới. Dùng **put.** Put nhận vào các action bạn muốn gửi đến middleware và reducer.

\_ Call là một phương thức chặn để xác định là gửi request AJAX.

\_ Sau đó mới yêu cầu thành công hoặc thất bại thì dùng put để dispatch action.