Cómo Hacer Tests de Canicas

> *Este documento se refiere a cómo hacer tests de canicas sobre el **funcionamiento interno del repositorio de RxJS**, y está orientado a todo el que quiera ayudar con el mantenimiento del repositorio de RxJS. **Los usuarios de RxJS** deberían leer la guía para hacer tests de canicas para aplicaciones en lugar de este documento. La mayor diferencia es que el comportamiento del `TestScheduler` difiere entre el uso manual y el uso de la función auxiliar `testScheduler.run(callback)`.*

Los "Tests de Canicas" son tests que utilizan un Planificador Virtual especializado llamado `TestScheduler`. Nos permiten hacer test a operaciones asíncronas de forma síncrona y fiable. La "sintaxis de canicas" es un concepto que se ha adaptado a partir de enseñanzas y/o documentos de personas tal y como @jhusain, @headinthebox, @mattpodwysocki y @andrestaltz. De hecho, André Staltz inicialmente la recomendó como un DSL para crear tests unitarios, y desde entonces, se ha alterado y adoptado.

Ver También

Código de Conducta

Métodos Básicos

Los tests unitarios tienen métodos auxiliares que se han añadido para facilitar la labor de creación de tests.

`hot(marbles: string, values?: object, error?: any)` - crea un Observable 'caliente' (un Sujeto) que se comportará como si ya se estuviese "ejecutando" al comenzar el test. Una diferencia interesante es que las canicas `hot` permiten un carácter `^` para señalar la posición del '*frame* cero'. Este es el punto en el que comienza la suscripción a los Observables a los que se realiza el test.

`cold(marbles: string, values?: object, error?: any)` - crea un Observable 'frío' cuya suscripción empieza al comenienzo del test.

`expectObservable(actual: Observable<T>).toBe(marbles: string, values?: object, error?: any)` - planifica un aserto para el momento en el que se llame a la función `flush()` del `TestScheduler`. El `TestScheduler` llamará a esta función automáticamente al final del bloque `it` de Jasmine.

`expectSubscriptions(actualSubscriptionLogs: SubscriptionLog[]).toBe(subscriptionMarbles: string)` - al igual que `expectObservable`, planifica un aserto para el momento en el que se llame a la función `flush()` del `TestScheduler`. Tanto `cold()` como `hot()` retornan un Observable con una propiedad `subscriptions` del tipo `SubscriptionLog[]`. Se proporciona el parámetro `subscriptions` a `expectSubscriptions` para aseverar si este coincide con el diagrama de canicas `subscriptionsMarbles` proporcionado en `toBe()`. Los diagramas de canicas de suscripción son ligeramente diferentes a los diagramas de canicas de observables.

Valores por defecto ergonómicos para *hot* y *cold*

En ambos métodos `hot` y `cold`, los caracteres que corresponden a valores emitidos que se especifican en los diagramas de canicas se emiten como cadenas, a no ser que se le proporcione un argumento `values` al método. Por tanto:

`hot('--a--b')` emitirá `'a'` y `'b'` mientras que `hot('--a--b', { a: 1, b: 2 })` emitirá `1` y `2`.

De la misma forma, los errores que no se especifiquen llevarán por defecto la cadena `'error'`:

`hot('---#')` emitirá el error `'error'` mientras que `hot('---#', null, new SpecialError('test'))` emitirá `new SpecialError('test')`.

Sintaxis de Canicas

La sintaxis de canicas es una cadena que representa eventos que ocurren a lo largo del 'tiempo'. El primer carácter de cualquier cadena de canicas siempre representa el '*frame* cero'. Un '*frame*' es, de cierta manera, análogo a un milisegundo virtual.

`'-'` tiempo: Representa 10 '*frames*' de paso de tiempo.

`'|'` compleción: Representa la compleción con éxito de un Observable. Se trata del Productor del Observable señalando `complete()`.

`'#'` error: Representa un error finalizando el Observable. Se trata del Productor del Observable señalando `error()`.

`'a'` cualquier carácter: Todos los demás caracteres representan un valor emitido por el Productor señalando `next()`.

`'()'` agrupaciones síncronas: cuando varios eventos tienen que ocurrir en el mismo *frame* de forma síncrona, se utilizan los paréntesis para agrupar dichos eventos. Se pueden agrupar valores `next`, `complete` o `error` de esta manera. La posición de la `(` inicial determina el momento en el tiempo en el que estos valores se emiten.

`'^'` punto de suscripción: (solo para Observables calientes) muestra el punto en el que los Observables a los que se les hace el test se suscriben al Observable caliente. Es el "frame cero" para el OBservable, cada *frame* anterior al `^` será negativo.

Ejemplos

`'-'` o `'------'`: Equivale a `never()` o a un Observable que nunca emite ni llega a completarse. `|`: Equivale a `Observable.empty()`. `#`: Equivale a `Observable.throwError()`. `'--a--'`: Un Observable que espera 20 *frames*, emite un valor `a` y nunca llega a completarse. `'--a--b--|'`: En el *frame* 20 emite `a`, en el *frame* 50 emite `b` y en el frame 80, `complete`. `'--a--b--#'`: En el *frame* 20 emite `a`, en el *frame* 50 emite `b` y en el frame 80, `error`. `'-a-^-b--|'`: Un Observable caliente, en el *frame* -20 emite `a`, en el *frame* 20 emite `b` y en el *frame* 50, `complete`. `'--(abc)-|'`: En el *frame* 20 emite `a`, `b` y `c`, en el *frame* 80, `complete`. `'-----(a|)'`: En el *frame* 50 emite `a` y `complete`.

Sintaxis de Canicas de Suscripción

La sintaxis de las canicas de suscripción es ligeramente distinta a la sintaxis convencional de canicas. Representa los puntos de suscripción y de cancelación de suscripción que ocurren en

el tiempo. No debería haber ningún otro tipo de evento representado en este tipo de diagrama.

`'-'` tiempo: Representa 10 '*frames*' de tiempo.

`'^'` punto de suscripción: Muestra el punto en el tiempo en el que se realiza una suscripción.

`'!'` punto de cancelación de suscripción: Muestra el punto en el tiempo en el que se cancela una suscripción.

En cada diagrama de canicas de suscripción debería haber como mucho un punto `^` y como mucho un punto `!`. Además de estos dos caracteres, el carácter `-` es el único permitido en un diagrama de canicas de suscripción.

Ejemplos

`'-'` or `'------'`: No se ha realizado ninguna suscripción. `'--^--'`: Se ha realizado una suscripción después de haber pasado 20 *frames* de tiempo, y dicha suscripción no se ha llegado a cancelar. `'--^--!-'`: En el *frame* 20 se ha realizado una suscripción, que se ha cancelado en el *frame* 50.

Anatomía de un Test

Un test básico podría tener el siguiente aspecto:

```
const e1 = hot("----a--^--b-------c--|");
const e2 = hot("---d-^--e---------f-----|");
const expected = "---(be)----c-f-----|";

expectObservable(e1.merge(e2)).toBe(expected);
```

Copy

Los caracteres `^` de un Observable `hot` siempre deben estar alineados.

El primer carácter de un Observable `cold` o de un Observable `expected` siempre deben estar alineados entre ellos, y también con el `^` de un Observable caliente.

Se deben utilizar los valores de emisión por defecto siempre que sea posible. Se especifican `values` cuando sea necesario.

Un test de ejemplo con valores especificados:

```
const values = {
  a: 1,
  b: 2,
  c: 3,
  d: 4,
  x: 1 + 3, // a + c
  y: 2 + 4, // b + d
};
const e1 = hot("---a---b---|", values);
const e2 = hot("-----c---d---|", values);
const expected = "-----x---y---|";

expectObservable(
  e1.zip(e2, function (x, y) {
    return x + y;
  })
).toBe(expected, values);
```

Copy

Se utiliza el mismo hash para buscar cualquier valor, ya que así se asegura que múltiples usos del mismo carácter tengan el mismo valor.

Se debe hacer que los valores del resultado sean lo más obviamente representativos que se pueda. Al fin y al cabo, se están realizando tests, y la claridad es más importante que la eficiencia, por lo que: `x: 1 + 3, // a + c` es mejor que simplemente `x: 4`. El primero transmite *por qué* el resultado es 4, mientras que el segundo no.

Un test de ejemplo con asertos de suscripción:

```
const x = cold(        '--a---b---c--|');
const xsubs =    '------^-------!';
const y = cold(                '---d--e---f---|');
const ysubs =    '--------------^-------------!';
const e1 = hot(  '------x-------y------|', { x: x, y: y });
const expected = '--------a---b----d--e---f---|';

expectObservable(e1.switch()).toBe(expected);
expectSubscriptions(x.subscriptions).toBe(xsubs);
expectSubscriptions(y.subscriptions).toBe(ysubs);
```

Se debe alinear el comienzo de los diagramas `xsubs` y `ysubs` con el diagrama `expected`.

Nótese como se cancela la suscripción al Observable frío `x` al mismo tiempo que `e1` emite el valor `y`.

En la mayoría de los casos será innecesario comprobar los puntos de suscripción y de cancelación de suscripción, ya que o serán obvios o bien estarán implícitos en el diagrama `expected`. En estos casos no es necesario realizar asertos de suscripción. En los casos en los que haya suscripciones internas u Observables fríos con varios suscriptores, los asertos de suscripción sí que resultarán útiles.

Cómo Generar diagramas de canicas PNF a partir de tests

Normalmente, cada caso de test en Jasmine se escribe como `it('should do something', function () { /* ... */ })`. Para indicar que se quiere generar un diagrama PNG en un caso de test concreto, se debe utilizar la función `asDiagram(label)`, de la siguiente manera:

```
it.asDiagram('zip')('should zip by concatenating', function () {
  const e1 =    hot('---a---b---|');
  const e2 =    hot('-----c---d---|');
  const expected =  '-----x---y---|';
  const values = { x: 'ac', y: 'bd' };

  const result = e1.zip(e2, function(x, y) { return String(x) + String(y); });

  expectObservable(result).toBe(expected, values);
});
```

Al ejecutar `npm run tests2png`, este caso de test se analizará sintácticamente y se generará un fichero PNG `zip.png` (el nombre del fichero viene determinado por `${operatorLabel}.png`) en la carpeta `img/`.