

Лекція №8. Гарячі та холодні Observable. Subject. Мультикастинг.

Тип Observable використовує лінійні обчислення, тобто нічого не робить, поки на нього хтось не підпишеться. Цим він відрізняється від типу Subject, який, будучи створений, вже представляє активну роботу. Завдяки лінійності композиція об'єктів Observable не призводить до втрати даних через стан гонки без кешування. У випадку Subject це не проблема, тому що одиночне значення можна кешувати, тому якщо значення доставлено для формування композиції, воно все одно буде отримано. Але якщо потік даних необмежений, то для надання аналогічної гарантії знадобився б необмежений буфер. Тому тип Observable представляє лінійні обчислення і не починає роботу до оформлення підписки, так що композицію можна повністю створити ще до того, як розпочнеться надходження даних.

Насправді це означає дві речі:

- Сигналом до початку роботи є підписка, а не конструювання. Завдяки лінійності Observable створення об'єкта цього типу не призводить до початку виконання роботи (якщо не вважати «роботою» виділення пам'яті для самого об'єкта Observable). У цей момент лише визначається, яку роботу потрібно виконати, коли хтось підпишеться на об'єкт. Розглянемо таке визначення Observable:

```
Observable<T> someData = Observable.create((observer) => {  
  getDataFromServerWithCallback(args, data => {  
    observer.next(data);  
    observer.complete();  
  });  
});
```

Посилання someData вже існує, але функція getDataFromServerWithCallback ще не виконується. Поки що тільки оголошено обгортку Observable навколо одиниці роботи, яку ще потрібно виконати.

Ця робота почне виконуватися, коли десь буде створено підписку на об'єкт Observable:

```
someData.subscribe(x => console.log(x));
```

- Об'єкти Observable можна використовувати повторно. Той факт, що тип Observable лінійний, дозволяє використовувати один екземпляр кілька разів. Це означає, що наступна послідовність операцій є законною:

```
someData.subscribe(x => console.log('Subscriber 1:', x));
```

```
someData.subscribe(x => console.log('Subscriber 2', x));
```

Тепер ми маємо дві окремі підписки, кожна з них викликає функцію `getDataFromServerWithCallback` і породжує окремі події.

Така лінивість відрізняється від типу `Subject`, оскільки створений об'єкт `Subject` представляє вже розпочату роботу. На об'єкти `Subject` можна підписуватись кілька разів, але всі вони будуть видавати однакові результати. Якщо існує посилання на `Subject`, то робота вже виконується.

У ліновості є свої плюси та мінуси, але в `RxJS` тип `Observable` лінивий. То б то, об'єкт `Observable` не зробить нічого, якщо на нього не підписатися.

Також об'єкти `Observable` в `RxJS` поділяються на холодні та гарячі.

Холодні та гарячі `Observable` в `rxjs`

У документації сказано:

`Cold Observable` - Це потік, джерело даних якого створюється всередині конструктора `Observable`.

`Hot Observable` - Це потік, джерело даних якого створюється зовні конструктора `Observable`.

Спробуємо відобразити це у коді. Як джерело даних може бути будь-що. Наприклад, `WebSocket`, таймери, масиви тощо.

Для простоти та наочності створимо функцію `getCurrentDate`, яка повертатиме значення поточної дати та часу, та використаємо її як джерело даних.

Дотримуючись визначення, створимо холодний `Observable` і підпишемося на нього тричі у різні проміжки часу.

```
e:\Project(RxJS)\Project9\src\index.js
function coldDate() {
  return new Observable((subscriber) => {
    const value = getCurrentDate();
    subscriber.next(value);
  });
}

const observable = coldDate();
const s1 = observable.subscribe(console.log);
setTimeout(() => {
  const s2 = observable.subscribe(console.log);
}, 2000);
```

```
setTimeout(() => {  
  const s3 = observable.subscribe(console.log);  
}, 4000);
```

Запустивши код, ми побачимо, що у кожного підписника значення дати та часу своє, яке генерується в момент підписки на Observable.

Тепер подивимось, яка поведінка буде, коли observable буде гарячим.

```
function hotDate() {  
  const value = getCurrentDate();  
  return new Observable((subscriber) => {  
    subscriber.next(value);  
  });  
}  
  
const observable = hotDate();  
const s1 = observable.subscribe(console.log);  
setTimeout(() => {  
  const s2 = observable.subscribe(console.log);  
}, 2000);  
setTimeout(() => {  
  const s3 = observable.subscribe(console.log);  
}, 4000);
```

У цьому випадку значення дати і часу у всіх те, що було згенеровано ще під час створення Observable. Варто уточнити, що джерело даних для гарячого Observable може бути і поза функцією фабрики.

Від того, де ініціалізується джерело даних, залежатиме, які значення отримуватимуть підписники.

При виклику функції `subscribe` у інстансу класу `Observable` ми створюємо джерело даних у конструкторі. Значить, в `cold observable` джерело даних буде створюватися щоразу заново, тому що він створюється у конструкторі, на відміну від `hot observable`.

Основні відмінності:

Властивості потоку Hot observable

Джерело даних створюється зовні конструктора observable, тому дані в ньому можуть генеруватися, навіть якщо немає підписників.

Потік ділиться значеннями, що прийшли йому з джерела даних, з усіма своїми підписниками

```
const subject$=new Subject();
subject$.next(5);
subject$.next(6);
subject$.subscribe(value=>{ console.log(
'Subject', value)});
```

Результат:

Властивості потоку Cold observable

Джерело даних створюється всередині конструктора Observable

Джерело даних для кожного підписника своє, а значить, і значення теж свої

```
const observable$=new Observable(observer=>{
  observer.next(5);
  observer.next(6);
  observer.complete();
});
Observable$.subscribe((value)=>{ console.log('Obs
ervable', value)});
```

Результат:

Observable 5

Observable 6

При error не має значення, коли ви підписалися на Subject. Error також як і complete підписники отримають у будь-якому випадку. Після отримання error або complete підписники вже не отримуватимуть жодних повідомлень.

Чи можна зробити гарячий потік на основі холодного?

Давайте тепер перетворимо гарячий потік на холодний і навпаки. Сам потік перетворити на якийсь інший не можна, тільки створити новий на основі нього.

Створити холодний потік на основі гарячого не можна, тому що джерело знаходиться назовні і перестворювати його для кожного нового підписника ми не можемо.

А ось зробити гарячий потік на основі холодного можна. Наприклад, створити subject і підписуватися на нього. Як джерело даних візьмемо холодний observable, і значення, що надходять з нього, будемо передавати в subject.

e:\Project(RxJS)\Project9\src\index.js

```
const observable = coldDate();
const subject=new rxjs.Subject();
subject.subscribe((v)=>{ write('Subscriber 1',v);})
subject.subscribe((v)=>{ write('Subscriber 2',v);})
subject.subscribe((v)=>{ write('Subscriber 3',v);})
```

```
observable.subscribe(subject);
```

Тоді всі наші підписники отримуватимуть одні й ті самі значення. Таку ж операцію роблять і оператори мультикастингу - такі як share або shareReply:

```
const hot$ = cold$.pipe(shareReply(1))
```

Multicast – це багатоадресна розсилка, де один відправник, багато одержувачів.

Потоки, що використовують оператори мультикастингу, іноді називають теплими потоками, оскільки джерело даних у цих потоках буде створено лише у момент першої підписки. Для решти підписників він буде розшарений. І після того, як останній підписник відпишеться, джерело даних перестане видавати значення. А для нових підписників джерело даних буде створено знову з самого початку.

Об'єкти Subject

RxJS Subject є різновидом об'єктів Observable. Особливість Subject в тому, що він може надсилати дані одночасно безлічі "споживачів", які можуть реєструватися вже в процесі виконання Subject, тоді як виконання стандартного Observable здійснюється унікально для кожного його виклику.

Об'єкти RxJS Subject реалізують принцип роботи подій, підтримуючи можливість реєструвати необмежену кількість обробників даних.

Розглянемо приклад.

```
e:\Project(RxJS)\Project9\src\index.js
const sbj=new rxjs.Subject();
sbj.subscribe((v1)=> console.log(`1st:${v1}`));
sbj.next(3);
sbj.subscribe((v1)=> console.log(`2nd:${v1}`));
sbj.next(9);
```

У даному прикладі створення Subject відбувається за допомогою rxjs.Subject(). Далі реєструються обробники викликом методу subscribe(), що приймає подібно до звичайного Observable три функції: next(), error() і complete().

Але тут обробники виконуються не відразу в момент виклику subscribe(), а після звернення до методів next(), error() або complete() об'єкта.

Причому реєстрація нових "споживачів" може відбуватися будь-якої миті. Але отримувати дані вони будуть вже з наступної розсилки.

Об'єкти RxJS Subject за замовчуванням є нескінченно виконуваними, оскільки заздалегідь невідомо, коли буде виклик complete() і чи буде викликаний взагалі. Тому не варто забувати про unsubscribe().

Суб'єкт RxJS - це особливий тип Observable, який дозволяє передавати значення багатьом спостерігачам. У той час як прості Observable є одноадресними (кожен підписаний спостерігач має незалежне виконання Observable), Subject's є багатоадресними.

Subject схожий на Observable, але він може здійснювати багатоадресну розсилку багатьом спостерігачам. Subject подібні до EventEmitters: вони підтримують передачу повідомлень безлічі слухачам.

Кожен Subject є спостережуваним. Маючи Subject, ви можете підписатись на нього, надавши Observer, який почне отримувати значення. З точки зору Observer (Спостерігача) він не може знати, чи відбувається виконання від простого одноадресного Observable або від Subject.

Усередині Subject, підписка не викликає нове виконання, яке доставляє значення. Підписка просто реєструє даного спостерігача у списку спостерігачів, аналогічно тому, як це addListener зазвичай працює в інших бібліотеках та мовах.

Кожен Subject є спостерігачем. Це об'єкт із методами next(v), error(e) і complete(). Щоб передати нове значення Subject, необхідно викликати метод next(theValue), і згенерована подія буде передана спостерігачам, зареєстрованим для прослуховування Subject.

У наведеному нижче прикладі ми маємо двох спостерігачів, прикріплених до суб'єкта, і ми передаємо йому деякі значення:

```
import {Subject} from 'rxjs';
const subject = new rxjs.Subject<number>();
subject.subscribe({
  next: (v) => console.log(`observer A: ${v}`),
});
```

```
subject.subscribe({
  next: (v) => console.log(`observer B: ${v}`),
});
subject.next(1);
subject.next(2);
```

```
// Logs:
// observer A: 1
// observer B: 1
// observer A: 2
// observer B: 2
```

Оскільки Subject є також і спостерігачем, це означає, що можна надати Subject як аргумент для subscribe будь-якого Observable, як показано в прикладі нижче:

```
import {Subject, from} from 'rxjs';
const subject = new rxjs.Subject();
subject.subscribe({
  next: (v) => console.log(`observer A: ${v}`),
});
subject.subscribe({
  next: (v) => console.log(`observer B: ${v}`),
});
const observable=from([1, 2, 3]);
observable.subscribe(subject); // You can subscribe providing a Subject
// Logs:
// observer A: 1
// observer B: 1
// observer A: 2
// observer B: 2
// observer A: 3
// observer B: 3
```

Використовуючи описаний вище підхід, ми, по суті, просто перетворили одноадресне виконання Observable на багатоадресне через Subject. Це показує, що Subject

є єдиним способом зробити будь-яке виконання Observable доступним для декількох спостерігачів.

У RxJS є кілька різновидів Subject:

- 1) BehaviorSubject,
- 2) ReplaySubject,
- 3) AsyncSubject.

BehaviorSubject (Використовується часто)

BehaviorSubject зберігає у собі останнє відправлене йому значення. Так, кожному новому обробнику в момент реєстрації (виклик subscribe()) буде надіслано це значення. Також об'єкт BehaviorSubject здатний зберігати в собі початкове значення.

Початкове значення визначається в момент створення RxJS BehaviorSubject.

```
e:\Project(RxJS)\Project9\src\index.js
```

```
const behaviorSubject$=new rxjs.BehaviorSubject('1');  
behaviorSubject$.subscribe((v)=>{  
  write('First subscriber',v)});
```

```
behaviorSubject$.next(2);  
behaviorSubject$.next(3);
```

```
behaviorSubject$.subscribe((v)=>{  
  write('Second subscriber',v)});
```

```
behaviorSubject$.next(4);
```

Результат:

```
First subscriber: 1
```

```
First subscriber: 2
```

```
First subscriber: 3
```

```
Second subscriber: 3
```

```
First subscriber: 4
```

```
Second subscriber: 4
```

BehaviorSubject має метод `getValue` (`rxjs/dist/types/internal`). Цей метод видає актуальний стан BehaviorSubject.


```

e:\Project(RxJS)\Project9\src\index.js
const behaviorSubject$=new rxjs.BehaviorSubject('1');
behaviorSubject$.subscribe((v)=>{
write('First subscriber',v)});

write('Current value',behaviorSubject$.getValue());
behaviorSubject$.next(2);
behaviorSubject$.next(3);

behaviorSubject$.subscribe((v)=>{
write('Second subscriber',v)});
write('Current value', behaviorSubject$.getValue());
behaviorSubject$.next(4);

write('Current value',behaviorSubject$.getValue());

```

Результат:

```

First subscriber: 1
Current value: 1
First subscriber: 2
First subscriber: 3
Second subscriber: 3
Current value: 3
First subscriber: 4
Second subscriber: 4
Current value: 4

```

ReplaySubject (Використовується рідко)

На відміну від BehaviorSubject, об'єкти ReplaySubject здатні зберігати задану кількість останніх значень, що задається при створенні об'єкта.

```

const sbj=new rxjs.ReplaySubject(2);
sbj.next(5);
sbj.next(6);

```

```
sbj.next(7);
sbj.subscribe((v)=>{ write('Subscriber 1',v);});
sbj.next(8);
sbj.next(9);
sbj.next(10);
sbj.subscribe((v)=>{ write('Subscriber 2',v);});
sbj.next(11);
```

Результат:

```
Subscriber 1: 6
Subscriber 1: 7
Subscriber 1: 8
Subscriber 1: 9
Subscriber 1: 10
Subscriber 2: 9
Subscriber 2: 10
Subscriber 1: 11
Subscriber 2: 11
```

Тобто всі нові "споживачі" відразу ж отримують по черзі всі п зазначених значень у RxJS ReplaySubject.

ReplaySubject() без параметрів отримуватиме у свої обробники всі значення всіх своїх подій. Тобто буфер за замовчуванням дорівнює нескінченності.

```
const sbj=new rxjs.ReplaySubject();
sbj.next(5);
sbj.subscribe((v)=>{ write('Subscriber 1',v);});
sbj.next(8);
sbj.subscribe((v)=>{ write('Subscriber 2',v);});
sbj.next(9);
```

Результат:

```
Subscriber 1: 5
Subscriber 1: 8
Subscriber 2: 5
Subscriber 2: 8
Subscriber 1: 9
```

Subscriber 2: 9

Конструктор об'єкта `ReplaySubject` приймає такі параметри:

```
constructor(_bufferSize?:number, _windowTime?: number, _timestampProvider?:  
TimestampProvider);
```

`_bufferSize` – розмір буфера, за умовчанням дорівнює нескінченності;

`_windowTime` – час кешування (час життя буфера). За замовчуванням дорівнює нескінченності;

`_timestampProvider` – дата, від якої починатиметься відлік життя буфера `_windowTime` (використовується дуже рідко).

Насправді, `ReplaySubject` легко замінюється звичайним `BehaviorSubject`.

AsyncSubject (Використовується рідко)

У випадку `AsyncSubject` "споживачам" передається тільки останнє значення об'єкта і тільки тоді, коли він завершить своє виконання (виклик `complete()`).

```
const sbj = new rxjs.AsyncSubject();  
sbj.subscribe(v1 => console.log(`Async: ${v1}`));  
sbj.next(7);  
sbj.next(8);  
sbj.next(9);  
setTimeout(() => sbj.complete(), 3000);
```

/*Результат у консолі (після закінчення 3 сек):

Async: 9

*/

Приклад з холодними та гарячими observable у Angular

`Notification producer` (Постачальник подій) в холодному об'єкті `Observable`, створюється самим об'єктом і тільки тоді коли хтось на нього підписується.

Для прикладу `interval()` створює холодний об'єкт, що спостерігається. Інформація створюється всередині об'єкта, що спостерігається, і для кожної нової підписки буде створюватися новий інтервал (`interval`).

`Notification producer` (Постачальник подій) у гарячих об'єктах, що спостерігаються, створюється поза об'єктом і не звертає уваги чи є підписники чи немає.

Для прикладу `fromEvent()` створює гарячий об'єкт, що спостерігається, де в якості постачальника подій виступає `DOM` і він існує незалежно від кількості підписників.

Іноді нам потрібно зробити холодний observable гарячим, візьмемо для прикладу http-запити. Розглянемо наступний приклад http-запиту в Angular.

Приклад з блоком підписника:

e:\Angular_App\HotCold\src\app\app.component.ts

```
this.user$=this.httpService.getData();
```

```
this.subscription1 = this.user$.subscribe(response => (this.name = response.name));
```

```
this.subscription2 = this.user$.subscribe(response => (this.id = response.id));
```

При цьому шаблон буде мати наступний вигляд:

```
<div>Name: {{ name }}</div>
```

```
<div>Id: {{ id }}</div>
```

У вкладці браузера network ми побачимо 2 запити до сервера, так як маємо дві підписки. Плюс ще треба і відписатись від потоків при знищенні компонента.

Статус	Метод	Домен	Файл	Ініціатор	Тип	Передано	Размер	0 мс	2.56 с
304	GET	localhost:4200	vendor.js	script	js	кешировано	2,94 МБ	53 мс	
304	GET	localhost:4200	main.js	script	js	кешировано	13,43 кБ	58 мс	
200	GET	localhost:4200	info?it=1698747457364	polyfills.js:14236 (xhr)	json	380 6	79 6	365 мс	
200	GET	jsonplaceholder.typicode.com	5	polyfills.js:14236 (xhr)	json	кешировано	507 6	0 мс	
200	GET	jsonplaceholder.typicode.com	5	polyfills.js:14236 (xhr)	json	кешировано	507 6	0 мс	
200	GET	localhost:4200	favicon.ico	FaviconLoader.sys.mjs:176 ...	vnd.micros...	кешировано	948 6	0 мс	
101	GET	localhost:4200	websocket	polyfills.js:4704 (websocket)	plain	129 6	0 6	15 мс	

Приклад з блоком async pipe:

e:\Angular_App\HotCold\src\app\app.component.ts

```
this.user$=this.httpService.getData();
```

```
this.name$ = this.user$.pipe(map(user => user.name));
```

```
this.id$ = this.user$.pipe(map(user => user.id));
```

і ми відображаємо ім'я та id користувача в шаблоні використовуючи пайп async (які розташовані у різних місцях, щоб неможливо було обробити в одному пайпі async).

```
<div>Name: {{ name$ | async }}</div>
```

```
<div>Id: {{ id$ | async }}</div>
```

Результат:

Статус	Метод	Домен	Файл	Ініціатор	Тип	Передано	Размер	0 мс	5,12 с
304	GET	localhost:4200	vendor.js	script	js	кешировано	0 6	39 мс	
200	GET	localhost:4200	main.js	script	js	13,75 кБ	13,45 кБ	3 мс	
200	GET	localhost:4200	info?it=1698682864777	polyfills.js:14236 (xhr)	json	380 6	79 6	6 мс	
101	GET	localhost:4200	websocket	polyfills.js:4704 (websocket)	plain	129 6	0 6	15 мс	
200	GET	jsonplaceholder.typicode.com	5	polyfills.js:14236 (xhr)	json	кешировано	507 6	0 мс	
200	GET	jsonplaceholder.typicode.com	5	polyfills.js:14236 (xhr)	json	кешировано	507 6	0 мс	
200	GET	localhost:4200	favicon.ico	FaviconLoader.sys.mjs:176 ...	vnd.micros...	кешировано	948 6	0 мс	

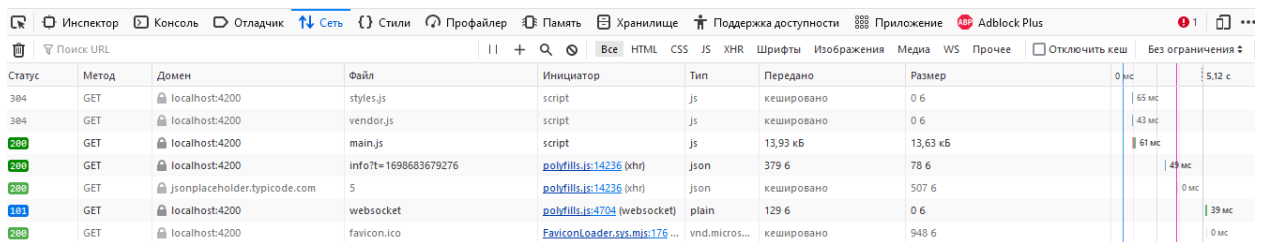
12 запросов | 16,14 кБ / 14,25 кБ передано | Передано за: 5,28 с | DOMContentLoaded: 829 мс | load: 4,49

У вкладці браузера network ми побачимо також 2 запити. Причина цього полягає в тому, що Angular's Http створює холодний observable так, що кожен новий підписник еквівалентний новому запиту. На практиці не вигідно робити кілька схожих запитів. Тому, не вигідно використовувати async пайп двічі тому що це призведе до кількох підписок замість одної.

Численні підписки можуть бути небезпечними. Нерідко подвійна підписка призводить до того, що наш компонент виконує два ідентичні http-запити замість одного, або відправляє ідентичні дії вашому сховищу станів, або здійснює важкі розрахунки більше одного разу.

Для вирішення цієї проблеми потрібно використати оператори share() або publish(), refCount():

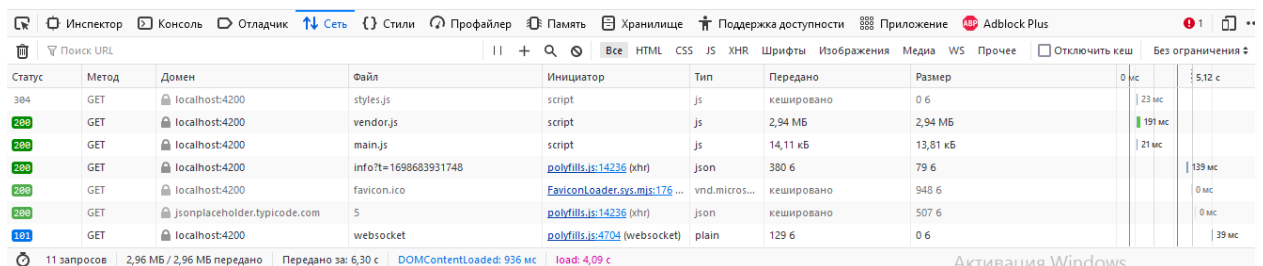
```
this.user$=this.httpService.getData().pipe(share());  
  
this.name$ = this.user$.pipe(map(user => user.name));  
  
this.id$ = this.user$.pipe(map(user => user.id));
```



Статус	Метод	Домен	Файл	Ініціатор	Тип	Передано	Размер	0 мс	5,12 с
304	GET	localhost:4200	styles.js	script	js	кешировано	0 6	65 мс	
304	GET	localhost:4200	vendor.js	script	js	кешировано	0 6	43 мс	
200	GET	localhost:4200	main.js	script	js	13,93 кБ	13,63 кБ	61 мс	
200	GET	localhost:4200	info?t=1698683679276	polyfills.js:14236 (xhr)	json	379 6	78 6	49 мс	
200	GET	jsonplaceholder.typicode.com	5	polyfills.js:14236 (xhr)	json	кешировано	507 6	0 мс	
101	GET	localhost:4200	websocket	polyfills.js:4704 (websocket)	plain	129 6	0 6	39 мс	
200	GET	localhost:4200	favicon.ico	FaviconLoader.sys.mjs:176 ...	vnd.micros...	кешировано	948 6	0 мс	

або

```
this.user$=this.httpService.getData().pipe(publish(),refCount());  
  
this.name$ = this.user$.pipe(map(user => user.name));  
  
this.id$ = this.user$.pipe(map(user => user.id));
```



Статус	Метод	Домен	Файл	Ініціатор	Тип	Передано	Размер	0 мс	5,12 с
304	GET	localhost:4200	styles.js	script	js	кешировано	0 6	23 мс	
200	GET	localhost:4200	vendor.js	script	js	2,94 МБ	2,94 МБ	191 мс	
200	GET	localhost:4200	main.js	script	js	14,11 кБ	13,81 кБ	21 мс	
200	GET	localhost:4200	info?t=1698683931748	polyfills.js:14236 (xhr)	json	380 6	79 6	139 мс	
200	GET	localhost:4200	favicon.ico	FaviconLoader.sys.mjs:176 ...	vnd.micros...	кешировано	948 6	0 мс	
200	GET	jsonplaceholder.typicode.com	5	polyfills.js:14236 (xhr)	json	кешировано	507 6	0 мс	
101	GET	localhost:4200	websocket	polyfills.js:4704 (websocket)	plain	129 6	0 6	39 мс	

11 запитів 2,96 МБ / 2,96 МБ передано Передано за: 6,30 с DOMContentLoaded: 936 мс load: 4,09 с Активация Windows

Оператори share(), publish() та інші multicasting-оператори змушують холодні Observable вести себе як гарячі.

Гарячі observable є multicast тому що всі підписники отримують інформацію з одного й того самого джерела подій (producer).

Холодні observable є unicast тому що кожен підписник отримує інформацію з різних джерел подій (producer) (то б то, під час створення підписки producer створюється для кожної підписки окремо).

multicast()

У RxJS є multicast() оператор, який приймає Subject або Subject factory і повертає так званий ConnectableObservable. Subject переданий як аргумент працює як посередник у multicast Observable. Він просто передає інформацію з вихідного Observable всім підписникам. ConnectableObservable - це звичайний Observable, але він не підписується до джерела доки не буде викликаний connect() method.

Приклад:

```
this.user$ = this.http.get(`api/user/1`).pipe(multicast(new Subject()));
```

Просто так це не запрацює, тому що нам потрібно викликати метод connect() вручну.

```
this.name$ = this.user$.pipe(map(user => user.name));
```

```
this.age$ = this.user$.pipe(map(user => user.age));
```

```
this.user$.connect();
```

Або в нашому Angular додатку з пайпами:

e:\Angular_App\HotCold\src\app\app.component.ts

```
this.user$=this.httpService.getData().pipe(multicast(new Subject()));
```

```
this.name$ = this.user$.pipe(map(user => user.name));
```

```
this.id$ = this.user$.pipe(map(user => user.id));
```

```
(this.user$ as ConnectableObservable<any>).connect();
```

Після цього ми отримаємо схожу поведінку, буде лише один запит замість двох. Підключення (Connecting) та відключення (disconnecting) вручну може бути складним для реалізації, тому є оператор — refCount() який автоматично підключатиметься (connect()) при першій підписці, зберігаючи кількість підписників та Subject-підключених до джерела подій до тих пір, поки будуть залишатися підписники. Коли не залишиться підписників, Subject буде відключено від джерела подій (Source).

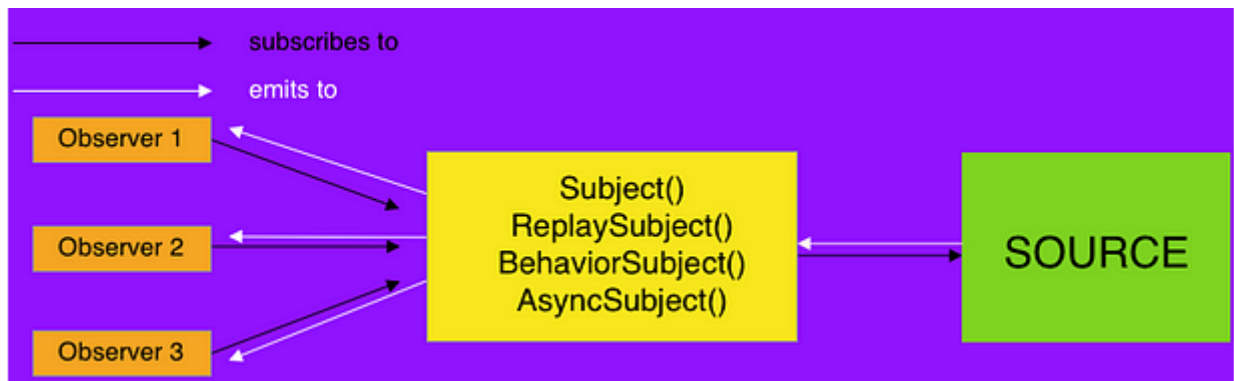
Джерелом подій (SourceObservable) у нашому прикладі виступає Observable, що повертається методом this.http.get();

Subject — внутрішній subject передано як аргумент multicast();

Subscriptions or observers — `this.name$` та `this.id$` та інші observable які підписані на Subject.

В даному випадку, всі підписники будуть підписані на Subject() (переданий в multicast) і Subject() сам підпишиться на виклик http. Коли Observable поверне результат запиту, Subject() отримає результат і поділиться ним з підписниками.

Основна ідея мультикастингу (multicasting) полягає в тому, що Subject підписується на джерело подій (Source) і безліч спостерігачів (Observers) підписуються на нього (Subject).



How multicasting works in RXJS

Модифікуємо наш код для використання `refCount()`, так нам не доведеться вручну коннектитися (викликати метод `connect`).

```
this.user$=this.httpService.getData().pipe(multicast(new Subject()), refCount());  
this.name$ = this.user$.pipe(map(user => user.name));  
this.id$ = this.user$.pipe(map(user => user.id));
```

Тепер нам не треба викликати вручну `connect()` і турбуватися про відписки(`disconnecting`). `refCount()` буде приєднувати (`connect`) Subject до джерела подій (Source) за першої підписки на нього і відпишеться коли не буде спостерігачів. Раніше ми використовували комбінацію `publish()`, `refCount()` вона буде еквівалентна такому запису `multicast(new Subject()), refCount()`.

publish()

У RxJS є оператор `publish()` і якщо подивимося на його вихідний код то побачимо, що він використовує `multicast` з `Subject()`

```
publish() === multicast(new Subject())
```

`publish()` опціонально приймає `selector function` і це фактично змінює поведінку оператора і це заслуговує на окрему статтю.

Так що коли ми використовуємо `publish()` ми насправді використовуємо вже відомий метод `multicast()` з `Subject()` і нам так само потрібно вручну подбати про з'єднання та від'єднання, або використовувати `refCount()` для автоматизації цього процесу.

Через те, що зазвичай `publish()` використовується разом з `refCount()`, то в rxjs є схожий оператор який використовує `refCount()` всередині себе і веде себе схоже (з комбінацією `publish` і `refCount`). Це `share()`.

share()

```
share() === multicast(() => new Subject()).refCount()
```

У першому прикладі ми бачимо, що `share()` робить те саме, що `publish()`, `refCount()` і в більшості випадків вони однакові. `share()` — це оператор, який використовує `refCount()` всередині себе, так що нам не потрібно викликати його. `share()` як і `publish()` використовує `multicast()` але вся різниця в аргументах переданих в `multicast()`.

`publish()` використовує екземпляр `Subject` - `multicast(new Subject())`

`share()` використовує функцію-фабрику, яка повертає екземпляр `Subject` — `multicast(() => new Subject()).refCount()`

Це єдина причина чому не можна сказати, що `share()` те саме, що `publish()` + `refCount()`. Ця різниця є причиною різної поведінки для запізнілих підписників (late subscribers) коли джерело (Source) стало завершеним (completed).

Відмінності між share() та publish() + refCount()

Вони обидва використовують `refCount()` для управління підписниками, проте:

`publish()` + `refCount()` - поки є хоча б один підписник на джерело (Subject), він видаватиме значення. Після того, як не залишиться підписників, джерело (Subject) буде відключено від вихідного джерела (Source). Якщо джерело (Source) було завершено, то всі нові підписники будуть отримувати “completed”, але якщо джерело (Source) не було завершено, він буде перепідключений до вихідного джерела (Source).

`share()` - поки є хоча б один підписник на джерело (Subject), він видаватиме значення. Після того, як не залишиться підписників, джерело (Subject) буде відключено від вихідного джерела (Source). Для будь-якого нового підписника, неважливо чи було завершено потік (Source) чи ні, він буде підписаний до джерела (Source) знову використовуючи `new Subject`.

Різниця дуже тонка, але важлива. Даваймо модифікуємо наш код, додавши кнопку, яка буде оновлювати інформацію користувача. При натисканні на неї дані будуть повторно завантажені із сервера.

Спочатку даваймо використаємо `share()`:

```
ngOnInit() {
```



```

    this.user$ = this.http.get(`api/user/1`).pipe(
      share()
    )
    this.name$ = this.user$.pipe(
      map(user => user.name)
    );
    this.age$ = this.user$.pipe(
      map(user => user.age)
    );
  }
  update() {
    this.name$ = this.user$.pipe(
      map(user => user.name + 'update')
    );
    this.age$ = this.user$.pipe(
      map(user => user.age + 'updated')
    );
  }
}

```

Коли при ініціалізації ми завантажили дані, `refCount()` порахує всі посилання. У нас є 2 посилання на `Subject`. Як тільки ми отримаємо дані з сервера, `Subject` отримає дані з джерела `Source` і завершиться (`complete`). Обидва підписника отримують дані з `Subject` і завершаться, що означає, що `refCount()` поверне 0. У такому випадку `Subject` буде відкріплений від джерела (`Source`). Коли ми виконаємо метод `update()`, `new Subject()` еземпляр буде створено та підписано на джерело(`Source`). Отже, кожен виклик `update()` буде надсилати запит на сервер.

Тепер розглянемо той самий приклад з `publish()`, `refCount()`

```

this.user$ = this.http.get(`api/user/1`).pipe(
  publish(),
  refCount()
);

```

Знову ми будемо мати `refCount()= 2` і як тільки `Source` запускає подію лічильник стане 0. Але коли ми виконаємо метод `update()` то нічого не відбудеться (не будуть повторно виконані запити до сервера). Як було написано раніше нові підписники будуть отримувати повідомлення про завершення джерела, якщо той завершиться.

Причина чому вони поводяться так у `multicast()` полягає в тому, що `publish()` використовує екземпляр `Subject`, коли він завершується, він буде теж завершений, так що кожен новий підписник отримає тільки повідомлення про завершення.

`share()` використовує фабричний метод, який повертає екземпляр `Subject`. Коли джерело завершується, `Subject` буде теж завершено, АЛЕ для нових підписників буде створено новий екземпляр `Subject`, який буде підписаний на основне джерело подій.

Лекція №9. Мультикастинг (продовження). Multicast() з різними типами subject.

Досі ми говорили про multicast, що використовує Subject. Є ще кілька типів Subject - ReplaySubject, BehaviorSubject та AsyncSubject. Передача мультикасту (multicast) різних Subject поверне ConnectableObservable, АЛЕ їх поведінка відрізнятиметься.

Перше, що ми розберемо буде ReplaySubject(n), він приймає число як аргумент, який є кількістю значень у буфері. Для будь-якого нового підписника він буде відтворювати n-у кількість значень.

Якщо ми передамо ReplaySubject(n) в multicast(), то кожен новий підписник буде отримувати n програних значень.

publishReplay()

```
publishReplay() === multicast(new ReplaySubject())
```

publishReplay() повертає ConnectableObservable так що нам потрібно використовувати connect() або refCount() для керування з'єднаннями (connections). Давайте модифікуємо наш приклад, так що кожен новий підписник отримуватиме значення з буфера. Коли ми клацаємо по update() ми не хочемо отримувати нові значення, але хочемо отримати закешовані значення.

```
this.user$ = this.http.get(`api/user/1`).pipe(  
  publishReplay(1),  
  refCount()  
);
```

Усі підписники ReplaySubject-а перед завершенням потоку отримають значення (у нашому випадку лише 1 значення оскільки http-емітить (emits) значення один раз). Для всіх нових підписників ReplaySubject повторюватиме N буферизованих значень.

Через те, що ми використовуємо publishReplay() з refCount(), є дуже схожий оператор, який використовує механізм підрахунку посилок усередині себе і веде себе схоже. Це shareReplay()

shareReplay()

shareReplay() – оператор, який може вести себе схожим з publishReplay() + refCount() чином, але він залежить від того, як ми його використовуємо.

До версії RxJs 6.4.0 механізм підрахунку посилок у shareReplay() працював трохи інакше. Починаючи з 6.4.0 можна явно передати аргумент для shareReplay() щоб той використовував “нормальний” механізм підрахунку посилок.

```
shareReplay({refCount: true}) (RXJS 6.4.0 або новіший)
```

`refCount: true` означає, що `shareReplay()` буде використовувати механізм підрахунку посилань, схожий на `refCount()`. У такому випадку `shareReplay({refCount: true})` буде те саме, що `publishReplay() + refCount()`. Давайте модифікуємо наш приклад із `shareReplay`.

```
this.user$ = this.http.get(`api/user/1`).pipe(  
  shareReplay({refCount: true, bufferSize: 1})  
);
```

У даному випадку, нам не потрібно більше використовувати `refCount()`, тому що `shareReplay({refCount: true})` використовує власний механізм підрахунку посилань. Результат буде однаковим з `publishReplay(1) + refCount()`. Всі підписники `ReplaySubject`-а отримуватимуть події щоразу, коли він їх видає. Усі нові підписники отримуватимуть `N` буферизованих значень. Перед тим як говорити про інші способи використання `shareReplay`, погляньмо на їх відмінності.

Відмінності між `shareReplay({refCount: true})` та `publishReplay() + refCount()`

Вони обидва використовують `ReplaySubject()`, але `shareReplay()` не використовує `multicast()`.

`publishReplay(n) + refCount()` — поки існує хоча б один підписник джерела, `ReplaySubject` буде видавати значення, якщо підписників немає, `ReplaySubject` буде відключений від джерела. Будь-який новий підписник отримає останні `N` значень із `ReplaySubject` і повторно перепідпишеться на джерело, використовуючи той самий `ReplaySubject`, якщо джерело ще не завершено.

`shareReplay({refCount: true, bufferSize: n})` - доки існує хоча б один підписник, `ReplaySubject` буде видавати значення, якщо підписників немає, `ReplaySubject` буде відключений від джерела. Для нових підписників, якщо джерело завершилося, він буде видавати останні `N` значень з `ReplaySubject`, але якщо джерело не завершилося, він знову підписуватиметься на джерело, використовуючи новий `ReplaySubject`.

Для того щоб побачити різницю, будемо використовувати `interval()` для того щоб нові підписники не були завершені:

```
this.source = interval(1000).pipe(  
  publishReplay(1),  
  refCount()  
);  
  
const sub1 = this.source.subscribe(x => console.log('sub 1', x));  
const sub2 = this.source.subscribe(x => console.log('sub 2', x));  
setTimeout(() => {
```

```

sub1.unsubscribe();
sub2.unsubscribe();
}, 2000);

```

У нас є 2 підписки на ReplaySubject, sub1 та sub2. Через 2 секунди обидва відписуються від Subject. Через те, що ми використовуємо refCount(), то коли не буде підписників (тобто лічильник посилань видасть нуль) ReplaySubject буде відключений від джерела подій (Source). Поки що ми побачимо в консолі.

```

sub 1-0
sub 2-0
sub 1-1
sub 2-1

```

Припустимо, що ми створили нову підписку на ReplaySubject, клікнувши на кнопку (після того як refCount видав нуль)

```

newSub() {
  const sub3 = this.source.subscribe(x => console.log('sub 3', x));
}

```

Коли newSub() отримає виконання, sub3 поверне останнє буферизоване значення з ReplaySubject (яке дорівнює 1) і перевірить завершеність Source. Якщо завершено, sub3 теж буде завершено. Як би там не було через те, що ми використовуємо interval(), Source не буде завершено і внутрішній ReplaySubject буде знову перепідписаний на існуючий Subject. Результат буде наступним.

```

sub 1-0
sub 2-0
sub 1-1
sub 2-1
/**** execution of newSub() ****/
sub 3- 1 <- replayed value
sub 3-0 <- new subscription
sub 3-1
sub 3-2
...

```

Внутрішній ReplaySubject повторює буферизовані значення для нових спостерігачів(observers) і завершує їх, або перепідписується на Source залежно від статусу Source.

Тепер повторимо приклад з interval() використовуючи shareReplay({refCount:true})

```

this.source = interval(1000).pipe(
  shareReplay({refCount: true, bufferSize: 1})
);
const sub1 = this.source.subscribe(x => console.log('sub 1', x));
const sub2 = this.source.subscribe(x => console.log('sub 2', x));
setTimeout(() => {
  sub1.unsubscribe();
  sub2.unsubscribe();
}, 2000);
//execute newSub() after sub1 and sub2 unsubscribe
newSub() {
  const sub3 = this.source.subscribe(x => console.log('sub 3', x));
}


```

shareReplay() не використовує multicast але він використовує внутрішню фабричну функцію і якщо ми використовуємо його з `refCount: true` і `refCount` повертає нуль, для будь-якого нового підписника, за умови завершення Source він повертатиме буферизовані значення та емітуватиме(`emit`) повідомлення про завершення (`completed`). Якщо Source не було завершено для кожного нового підписника буде створено новий `ReplaySubject` і підписка на Source. В результаті, після запуску коду вище та виконання `newSub()` ми побачимо.

```

sub 1-0
sub 2-0
sub 1-1
sub 2-1
/**** execution of newSub() ****/
sub 3-0 <- new subscription
sub 3-1
sub 3-2
...

```

Як можна помітити, що для `sub3` значення не повторюється. Причина в тому, що при відписці (`unsubscribe`) `sub1` і `sub2`, `refCount` поверне нуль і якщо Source завершено всі нові підписники як і `sub3` будуть отримувати всі буферизовані значення і статус `completed`, але так як ми використовуємо інтервал, Source не буде завершений, `ReplaySubject` буде знищений і новий підписник буде створювати новий екземпляр `ReplaySubject` та підписуватись на Source заново.

shareReplay() без refCount

До цього часу ми використовували shareReplay з refCount: true.

Ми можемо використовувати shareReplay з refCount встановленим у false або взагалі не встановленим (not set at all) і визначити тільки розмір буфера - наприклад shareReplay({refCount: false, bufferSize: 3}) та shareReplay(3) будуть еквівалентні. Це означає, що ReplaySubject буде публікувати останні 3 значення і refCount буде false. Це не означає, що не буде механізму підрахунку посилань, він просто буде інакше поводитися.

За замовчуванням refCount встановлено як false у shareReplay().

refCount: false означає, що при першій підписці на ReplaySubject він буде підписаний на Source. Але він не відключить ReplaySubject від Source, в якому не залишилося підписників на ReplaySubject. Давайте змінимо наш приклад знову виставивши refCount у значення false:

```
this.source = interval(1000).pipe(  
  shareReplay({refCount: false, bufferSize: 2})  
  //or just shareReplay(2)  
);  
  
const sub1 = this.source.subscribe(x => console.log('sub 1', x));  
const sub2 = this.source.subscribe(x => console.log('sub 2', x));  
  
setTimeout(() => {  
  sub1.unsubscribe();  
  sub2.unsubscribe();  
}, 2000);  
  
setTimeout(() => {  
  const sub3 = this.source.subscribe(x => console.log('sub 3', x));  
}, 4000);
```

sub1 і sub2 підписані на ReplaySubject та ReplaySubject підписується на внутрішній Source. Після 2 секунд sub1 і sub2 буде відписано, але в цьому випадку ReplaySubject не буде відписано від Source. Source буде продовжувати публікувати значення навіть якщо немає підписників, щоб обробити його значення. Після 4 секунд новий підписник підпишеться на ReplaySubject і отримає останні 2 значення буфера і продовжить отримувати значення з Source. Результат буде наступним.

```
sub 1-0  
sub 2-0  
sub 1-1  
sub 2-1
```

```
/**** after 4 seconds ****/
```

```
sub 3-2 <- replayed values
```

```
sub 3-3 <-
```

```
sub 3-4 <- continues receiving values
```

```
sub 3-5
```

...

sub1 і sub2 підписані, друкують значення і після двох секунд відписуються, але через те, що Source ще не завершено (completed), ReplaySubject буде отримувати дані з Source і тому після 4 секунд sub3 підписується на ReplaySubject і він отримає не 0 і 1 як значення з буфера, але 2 і 3, тому що ReplaySubject-у вдалося отримати нові дані з Source і поновити буфер. Єдиний випадок, коли ReplaySubject відпишеться від Source це коли Source завершено або завершено помилкою. Кожен новий підписник у такому разі отримуватиме відтворювані значення та завершиться.

shareReplay(n) — Не важливо чи є активні підписки чи ні, ReplaySubject продовжуватиме публікувати значення та зберігати з'єднання з Source доки той не буде завершений або видасть помилку. Кожен новий підписник (Subscriber) отримуватиме останні N значень. Якщо Source ще не завершився, нові підписники продовжуватимуть отримувати значення з нього (Source).

publishBehavior()

publishBehavior() використовує multicast з іншим subject, з BehaviorSubject.

```
publishBehavior() === multicast(new BehaviorSubject())
```

publishBehavior() поверне ConnectableObservable який нам потрібно використовувати для refCount() або підключатися вручну.

publishBehavior() приймає значення за замовчуванням як параметр і публікуватиме його для всіх підписників якщо Source не публікував. Розглянемо приклад:

```
this.source = interval(1000).pipe(
```

```
publishBehavior(47),
```

```
refCount()
```

```
);
```

```
const sub1 = this.source.subscribe(x => console.log('sub 1', x));
```

```
const sub2 = this.source.subscribe(x => console.log('sub 2', x));
```

```
setTimeout(() => {
```

```
  sub1.unsubscribe();
```

```
  sub2.unsubscribe();
```

```
}, 2000);
```



```

setTimeout(() => {
  const sub3 = this.source.subscribe(x => console.log('sub 3', x));
}, 4000);

```

Результат буде наступним

```

sub 1-47 <- default values
sub 2-47 <-
sub 1-0
sub 2-0
sub 1-1
sub 2-1
/**** after 4 seconds ****/
sub 3- 1 <- last buffered value
sub 3-1
sub 3-2
...

```

Через те що interval асинхронний, коли sub1 і sub2 підписуються на BehaviorSubject, в цей час Source ще не опублікував значення, так що sub1 і sub2 буде діставати значення за замовчуванням з BehaviorSubject. Через 2 секунди sub1 і sub2 будуть відписані від BehaviorSubject та BehaviorSubject відпишеться від Source. Через 4 секунди sub3 буде підписаний на BehaviorSubject через те, що Source ще не був завершений, sub3 буде отримувати останнє опубліковане значення і перепідпишеться на Source використовуючи той же (name) BehaviorSubject.

publishBehavior(default_value) — коли підписка на BehaviorSubject створена до того як Source опублікував значення, BehaviorSubject передасть значення за замовчуванням підписнику. Поки є хоча б один підписник на Source BehaviorSubject публікуватиме значення. Коли не залишиться підписників BehaviorSubject буде відключено від Source. Якщо Source ще не було завершено, нові підписники будуть отримувати останнє значення від BehaviorSubject і перепідписуватись на source користуючись тим самим BehaviorSubject. Якщо Source було завершено, всі нові підписники будуть отримувати статус завершення.

publishLast()

publishLast() використовує multicast с AsyncSubject

```
publishLast() === multicast(new AsyncSubject())
```

Як і всі multicast-оператори publishLast() краще використовувати з refCount().

AsyncSubject не буде публікувати значення, якщо підписаний до завершення. Після завершення він опублікує останнє значення.

```
this.source = interval(1000).pipe(
  take(2),
  publishLast(),
  refCount()
);
const sub1 = this.source.subscribe(x => console.log('sub 1', x));
const sub2 = this.source.subscribe(x => console.log('sub 2', x));
setTimeout(() => {
  const sub3 = this.source.subscribe(x => console.log('sub 3', x));
}, 7000);
```

Через те, що interval це нескінченний observable, то ми використовуємо take(2). Після чого буде згенеровано 2 значення та потік завершиться. Нижче продемонстровано результат.

```
sub 1-1 //completed
sub 2-1 //completed
/**** after 7 seconds ****/
sub 3-1 //completed
```

Коли sub1 і sub2 підписуються на AsyncSubject вони не будуть отримувати значення, поки Source не завершиться. Коли Source завершується AsyncSubject передаватиме останнє значення всім спостерігачам (observers) і завершиться. Після 7 секунд sub3 підписується на AsyncSubject і через те, що він завершений sub3 отримає останнє значення і теж завершиться.

publishLast() — Не важливо як багато підписників підключених (connected) до AsyncSubject, він не буде видавати значення поки джерело (Source) не завершено(completes). Коли джерело (Source) змінило статус на завершення AsyncSubject також змінює статус на завершення і видає останнє значення і повідомлення про завершення ('complete') всіх поточних і нових підписників.