RxJS - RxJS: Glossary And Semantics

RxJS : glossaire et sémantique

Lors de la discussion et de la documentation des observables, il est important d'avoir un langage commun et un ensemble de règles connues sur ce qui se passe. Ce document est une tentative de normaliser ces choses afin que nous puissions essayer de contrôler le langage dans nos documents, et espérons-le dans d'autres publications sur RxJS, afin que nous puissions discuter de la programmation réactive avec RxJS dans des termes cohérents.

Bien que toute la documentation de RxJS ne reflète pas cette terminologie, l'objectif de l'équipe est de s'assurer qu'elle le fait et de s'assurer que le langage et les noms autour de la bibliothèque utilisent ce document comme source de vérité et de langage unifié.

Entités majeures

Il existe des entités de haut niveau qui sont fréquemment discutées. Il est important de les définir séparément des autres concepts de niveau inférieur, car ils se rapportent à la nature de l'observable.

Consommateur

Le code qui s'abonne à l'observable. Il s'agit de la personne qui est *informée* des valeurs suivantes , des erreurs ou des achèvements .

Producteur

Tout système ou chose qui est la source de valeurs qui sont expulsées de l'abonnement observable au consommateur. Cela peut être une grande variété de choses, d'une `WebSocket`simple itération sur un `Array`. Le producteur est le plus souvent créé lors de l' action d'abonnement , et donc "possédé" par un abonnement de manière 1:1, mais ce n'est pas toujours le cas. Un producteur peut être partagé entre plusieurs abonnements, s'il est créé en dehors de l' action subscribe , auquel cas il est un-à-plusieurs, ce qui entraîne une multidiffusion .

Abonnement

Un contrat où un consommateur observe des valeurs poussées par un producteur . L'abonnement (à ne pas confondre avec la `Subscription`classe ou le type), est un processus continu qui revient à la fonction de l'observable du point de vue du Consommateur. L'abonnement commence au moment où une action d'abonnement est initiée, même avant que l' action d'abonnement ne soit terminée.

Observable

Le type principal dans RxJS. À son plus haut niveau, un observable représente un modèle pour connecter un Observer , en tant que consommateur , à un producteur , via une action subscribe , résultant en un abonnement .

Observateur

La manifestation d'un consommateur . Un type qui peut avoir certains (ou tous) gestionnaires pour chaque type de notification : next , error et complete . Le fait d'avoir les trois types de gestionnaires l'appelle généralement un "observateur", où s'il manque l'un des gestionnaires de notification, il peut être appelé un " observateur partiel" .

Actions majeures

Il existe des actions et des événements spécifiques qui se produisent entre les principales entités de RxJS et qui doivent être définis. Ces actions majeures sont les événements de plus haut niveau qui se produisent dans différentes parties de RxJS.

S'abonner

Action d'un consommateur demandant à un Observable de mettre en place un abonnement afin qu'il puisse observer un producteur . Une action d'abonnement peut se produire avec un observable via de nombreux mécanismes différents. Le mécanisme principal est la `subscribe`méthode sur la classe Observable . D'autres mécanismes incluent la `forEach`méthode , des fonctions telles que `lastValueFrom`, et `firstValueFrom`, et la `toPromise`méthode obsolète .

Finalisation

Action de nettoyer les ressources utilisées par un producteur. Ceci est garanti sur `error`, `complete`, ou en cas de désinscription. Cela ne doit pas être confondu avec la désinscription , mais cela se produit toujours lors de la désinscription.

Désinscription

L'acte d'un consommateur disant à un producteur qu'il n'est plus intéressé à recevoir des valeurs. Finalisation des causes

Observation

Un consommateur réagissant aux notifications suivantes , d' erreur ou complètes . Cela ne peut se produire *que lors* de l'abonnement .

Chaîne d'observation

Lorsqu'un observable utilise un autre observable comme producteur , une "chaîne d'observation" se met en place. C'est une chaîne d' observation telle que plusieurs observateurs se notifient de manière unidirectionnelle vers le consommateur final .

Suivant

Une valeur a été poussée au consommateur à respecter . Ne se produira que pendant l'abonnement et ne peut pas se produire après une erreur , une fin ou un désabonnement . Logiquement, cela signifie également que cela ne peut pas se produire après la finalisation .

Erreur

Le producteur a rencontré un problème et en informe le consommateur . Il s'agit d'une notification que le producteur n'enverra plus de valeurs et qu'il finalisera . Cela ne peut pas se produire après la fin , toute autre erreur ou désinscription . Logiquement, cela signifie également que cela ne peut pas se produire après la finalisation .

Complet

Le producteur informe le consommateur que c'est fait , les valeurs suivantes , sans erreur, n'enverront plus de valeurs, et il finalisera . L'achèvement ne peut pas se produire après une erreur ou un désabonnement . Complete ne peut pas être appelée deux fois. Complete , s'il se produit, se produira toujours avant finalization .

Notification

L'acte d'un producteur poussant les valeurs suivantes , les erreurs ou les complétions à un consommateur à observer . À ne pas confondre avec le `Notification`type , qui est une notification manifestée sous la forme d'un objet JavaScript.

Notions majeures

Une partie de ce dont nous discutons est conceptuelle. Ce sont principalement des traits communs de comportements qui peuvent se manifester dans des observables ou dans des systèmes réactifs basés sur le push.

Multidiffusion

L'acte d'un producteur observé par de **nombreux** consommateurs .

Monodiffusion

L'acte d'un producteur observé par un **seul** consommateur . Un observable est "unicast" lorsqu'il ne connecte qu'un seul producteur à un seul consommateur . Unicast ne signifie pas nécessairement "froid" .

Froid

Un observable est "froid" lorsqu'il crée un nouveau producteur lors de l'abonnement pour chaque nouvel abonnement . En conséquence, les observables "froids" sont *toujours* unicast , étant un producteur observé par un consommateur . Les observables froids peuvent être rendus chauds mais pas l'inverse.

Chaud

Un observable est "chaud", lorsque son producteur a été créé en dehors du contexte de l' action subscribe . Cela signifie que l'observable "chaud" est presque toujours multicast . Il est possible qu'un observable "chaud" soit encore *techniquement* unicast, s'il est conçu pour n'autoriser qu'un seul abonnement à la fois, cependant, il n'y a pas de mécanisme simple pour cela dans RxJS, et le scénario est peu probable. Aux fins de la discussion, tous les observables "chauds" peuvent être supposés être multicast . Les observables chauds ne peuvent pas être rendus froids .

Pousser

Observables are a push-based type. That means rather than having the consumer call a function or perform some other action to get a value, the consumer receives values as soon as the producer has produced them, via a registered next handler.

Pull

Pull-based systems are the opposite of push-based. In a pull-based type or system, the consumer must request each value the producer has produced manually, perhaps long after the producer has actually done so. Examples of such systems are Functions and Iterators

Minor Entities

Operator

A factory function that creates an operator function. Examples of this in rxjs are functions like `map` and `mergeMap`, which are generally passed to `pipe`. The result of calling many operators, and passing their resulting operator functions into pipe on an observable source will be another observable, and will generally not result in subscription.

Operator Function

A function that takes an observable, and maps it to a new observable. Nothing more, nothing less. Operator functions are created by operators. If you were to call an rxjs operator like map and put the return value in a variable, the returned value would be an operator function.

Operation

An action taken while handling a notification, as set up by an operator and/or operator function. In RxJS, a developer can chain several operator functions together by calling operators and passing the created operator functions to the `pipe` method of `Observable`, which results in a new observable. During subscription to that observable, operations are performed in an order dictated by the observation chain.

Stream

A "stream" or "streaming" in the case of observables, refers to the collection of operations, as they are processed during a subscription. This is not to be confused with node Streams, and the word "stream", on its own, should be used *sparingly* in documentation and articles. Instead, prefer observation chain, operations, or subscription. "Streaming" is less ambiguous, and is fine to use given this defined meaning.

Source

An observable or valid observable input having been converted to an observable, that will supply values to another observable, either as the result of an operator or other function that creates one observable as another. This source, will be the producer for the resulting observable and all of its subscriptions. Sources may generally be any type of observable.

Observable Inputs

An "observable input" (defined as a type here), is any type that can be easily converted to an Observable. Observable Inputs may sometimes be referred to as "valid observable sources".

Notifier

An observable that is being used to notify another observable that it needs to perform some action. The action should only occur on a next notification, and never on error or complete. Generally, notifiers are used with specific operators, such as `takeUntil`, `buffer`, or `delayWhen`. A notifier may be passed directly, or it may be returned by a callback.

Inner Source

One, of possibly many sources, which are subscribed to automatically within a single subscription to another observable. Examples of an "inner source" include the observable inputs returned by the mapping function in a mergeMap operator. (e.g. `source.pipe(mergeMap(value => createInnerSource(value)))`, where `createInnerSource` returns any valid observable input).

Partial Observer

An observer that lacks all necessary notification handlers. Generally these are supplied by user-land consumer code. A "full observer" or "observer" would simply be an observer that has all notification handlers.

Other Concepts

Unhandled Errors

An "unhandled error" is any error that is not handled by a consumer-provided function, which is generally provided during the subscribe action. If no error handler was provided, RxJS will assume the error is "unhandled" and rethrow the error on a new callstack to prevent "producer interference".

Producer Interference

Producer interference happens when an error is allowed to unwind the RxJS callstack during notification. When this happens, the error could break things like for-loops in upstream sources that are notifying consumers during a multicast. That would cause the other consumers in that multicast to suddenly stop receiving values without logical explanation. As of version 6, RxJS goes out of its way to prevent producer interference by ensuring that all unhandled errors are thrown on a separate callstack.

Upstream And Downstream

L'ordre dans lequel les notifications sont traitées par les opérations dans un flux a une directionnalité. « En amont » fait référence à une opération qui a déjà été traitée avant l' opération en cours , et « En aval » fait référence à une opération qui *sera* traitée *après* l' opération en cours . Voir aussi : Streaming .