НЕТОЛОГИЯ

RXJS

АЛЕКСЕЙ КУЛАГИН

АЛЕКСЕЙ КУЛАГИН

СТО в студии Штрихпунктир

@alex_kulagin

ПЛАН ЗАНЯТИЯ

1. Синхронная функция, callback, Promise
2. Что делать когда данных много?
3. Паттерн Observer
4. Пример: Observer и ввод данных
5. Как нам поможет RxJS?
6. Что такое подписка и зачем нужно отписываться?
7. Пример: drag & drop
8. Пример: Стейт менеджер на RxJS

ЗАДАЧА

Перед нами поставили следующую задачу: реализовать валидацию логина пользователя при регистрации как на GitHub.

Как это выглядит: пользователь вводит логин, а мы по мере набора проверяем, свободен ли данный логин, отправляя http-запрос на сервер.

ФЛУД

Вы уже знаете примерно как это реализовать, достаточно подписаться на событие `input` и при его возникновении отправлять http-запрос. Но этот подход не очень хорош, т.к. при быстром наборе логина мы буквально «зафлудим» сервер. Почему бы нам не отправлять этот запрос с каким-то интервалом? Например, xxx мс?

Почему бы нет? Делаем в `eventListener`'а `input`'а `setTimeout`, как он сработает, отправим http-запрос.

ОТМЕНА `timeout`'ОВ

А если за это время произошло ещё одно событие `input`, то отменим предыдущий `timeout` и поставим новый. Стало сложновато, не правда ли?

А что если эту задачу можно решить буквально в пару строк?

```
fromEvent(document.getElementById('userEmail'), 'input').pipe(
  debounceTime(100),
  pluck('target', 'value'),
  map(value => value.trim()),
  filter(value => value !== ''),
  switchMap(value => getRequest(`/api/check-email?email=${value}`))
);
```

ЧТО ЭТО ТАКОЕ?

Это специальный «реактивный» подход к обработке данных с использованием библиотеки rxjs.

Но прежде чем мы с ним разберёмся, поговорим о том, как мы вообще умеем получать данные.

СИНХРОННАЯ ФУНКЦИЯ, CALLBACK, PROMISE

ПОЛУЧЕНИЕ ДАННЫХ

Мы уже умеем работать с функциями - это самый простой способ **синхронно** вычислить значение.

```
1  function calculateValue(initialData) {
2    // some code here
3  }
4
5  calculateValue(42);
```

CALLBACK

Но в JS чаще всего приходится работать с асинхронными данными из-за природы самого языка.

Для получения синхронных данных самая распространенная модель - это вызов **callback** функции - например, когда отработает `timeout`.

```
setTimeout(() => doActionAfterTimeout(), 42);
```

Или более распространенный вариант - вызвать колбэк после получения данных

```
1  const req = new XMLHttpRequest();
2
3  req.addEventListener('load', event => {
4    // работаем в колбэке с результатом
5  });
```

EventListeners

А еще мы вызываем **callback** когда добавляем обработчик событий DOM элементов. Например, так:

```
document.getElementById('submit-btn').addEventListener('click', event => {
  // обрабатываем клик по кнопке
});
```

Promise

Кроме колбеков мы еще знакомы с техникой промисов - обещание вернуть результат в будущем. Этот вариант получения данных использует, например, Fetch API.

```
fetch('/api/items')
  .then(response => response.json())
  .then(jsonData => processJsonData(jsonData));
```

ЧТО ДЕЛАТЬ КОГДА ДАННЫХ МНОГО?

В ЧЕМ ПРОБЛЕМА?

Так в чем же проблема при таком разнообразии вариантов? Проблема в самом разнообразии. А еще в ограничениях, которые каждый из подходов порождает:

- сихронные функции работают синхронно
- функции обратного вызова при большой вложенности очень трудно читать, а еще в неопытных руках они могут порождать callback hell
- `Promise` обещает вернуть значение, но только одно. Для того, чтобы заставить промис вернуть несколько значений нужно хорошенько постараться

ПРИМЕР

```
document.getElementById('submit-btn').addEventListener('click', (event) => {
  const sendValueReq = new XMLHttpRequest();
  const data = new FormData();
  data.append('value', 42);
  sendValueReq.addEventListener('load', (event) => {
    const { status } = event.target;
    if (status >= 200 && status < 300) {
      const getItemsReq = new XMLHttpRequest();
      getItemsReq.addEventListener('load', (event) => {
        const { status } = event.target;
        if (status >= 200 && status < 300) {
          process(event.target.response);
        }
      });
      getItemsReq.open('GET', '/api/items');
      getItemsReq.send();
    }
  });
  sendValueReq.open('POST', '/api/create');
  sendValueReq.send();
});
```

СВЯЗНОСТЬ

А еще эти паттерны добавляют высокую связность в код. Это значит что функция, которая, например, принимает событие знает о том, кто это событие обрабатывает.

Когда нам нужно добавить еще одно поведение при клике на эту же кнопку, нам нужно либо менять обработчик, либо добавлять еще один.

ПАТТЕРН OBSERVER

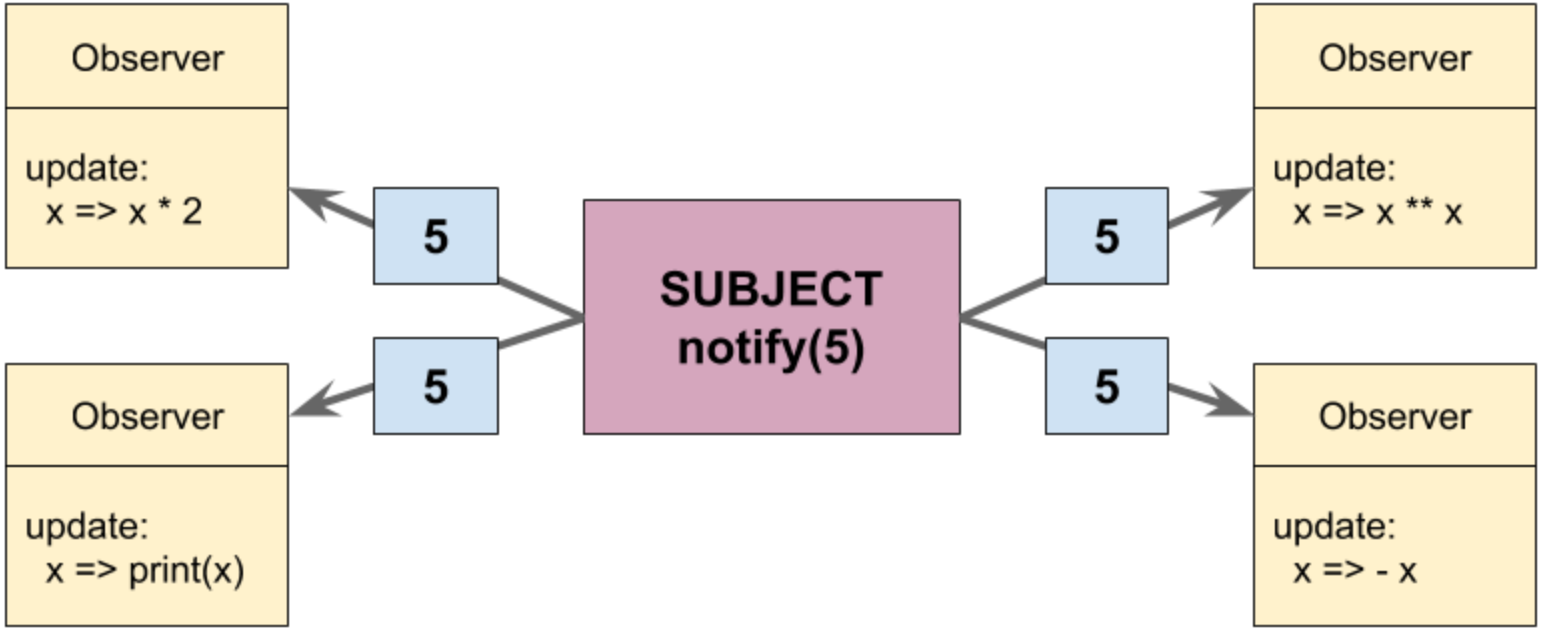
КАК ЭТО РЕШИТЬ?

Множество проблем можно решать используя общепринятые практики - паттерны. В данном случае множество проблем нам поможет решить паттерн Observer.

В этом паттерне участвуют два типа объектов:

- Subject - источник данных, которого можно оповестить о том, что пришло новое значие или изменилось актуальное значение
- Observer - наблюдатель, который получает сообщение от Subject'а о том что обновилось значение

КАК ЭТО РЕШИТЬ?



SUBJECT

Subject в данном случае является потоком, в котором постоянно появляются новые данные, а Observer'ы реагируют на каждое новое значение в потоке.

РЕАЛИЗАЦИЯ

В самом простом варианте паттерн можно организовать следующим образом:

```
const stream$ = {
  observers: [],
  subscribe(observer) {
    this.observers.push(observer);
  },
  next(value) {
    this.observers.forEach(observer => observer(value));
  }
};
```

РЕАЛИЗАЦИЯ

Но использовать один Subject на всё приложение не очень хорошая идея, поэтому лучше сделать класс для создания нового Subject'а:

```
class Subject {
  constructor() {
    this.observers = [];
  }
  subscribe(observer) {
    this.observers.push(observer);
  }
  next(value) {
    this.observers.forEach(observer => observer(value));
  }
}
```

PATTERN

Конечно мы никуда не сможем уйти от стандартных механизмов JS, таких как callback или `Promise`, но мы сможем их скрыть с помощью паттерна `Observable`.

Это позволит нам решить сразу несколько проблем:

- уменьшит связность кода - мы сможем добавлять независимые обработчики на наш источник данных
- позволит подменять реализацию получения данных не меняя при этом потребителей

ПРИМЕР: OBSERVER И ВВОД ДАННЫХ

ПРИМЕР

Попробуем применить это на практике. Для создания Subject'а будем использовать наш класс

Для начала созданим объект Subject:

```
// создаем источник
const stream$ = new Subject();
```

Дальше добавим обработчик события ввода текста в поле - когда значение меняется отправляем его в stream:

```
document
  .getElementById('userEmail')
  .addEventListener('input', e => stream$.next(e));
```

ПРИМЕР

Теперь добавим обработчики нового значения в потоке:

```
stream$.subscribe(event => sendData(event.target.value));
stream$.subscribe(event => processAnalytics(event.type, event.target));
```

Плюс такого подхода в том, что мы буквально в рантайме можем добавлять обработчики. Например, добавим логгер если мы находимся в режиме разработки:

```
if (productionMode === false) {
  stream$.subscribe(event => log(event));
}
```

КАК НАМ ПОМОЖЕТ RXJS?

КАК НАМ ПОМОЖЕТ RXJS?

`RxJS` - это готовая библиотека для работы с потоками.

Ее можно установить как через npm

```
npm install rxjs
```

Или можно подключить через CDN - тогда глобально будет доступна переменная `rxjs`:

```
<script src='https://unpkg.com/rxjs/bundles/rxjs.umd.min.js'></script>
```

МЕНЯЕМ ПРИВЫЧНЫЕ ПОДХОДЫ

Для того, чтобы превратить знакомые нам подходы в потоки, у rxjs есть уже готовые функции.

Например, чтобы начать слушать событие на DOM элементе используется функция `fromEvent`:

```
import { fromEvent } from 'rxjs';

userEmail$ = fromEvent(document.getElementById('userEmail'), 'input');
```

ЧТО ТАКОЕ ПОДПИСКА И ЗАЧЕМ НУЖНО ОТПИСЫВАТЬСЯ?

ПОТОКИ

Здесь мы получили поток `userEmail$` (знак `$` не обязателен, это соглашение для обозначения переменных потока - stream). Для того, чтобы начать получать из потока данные нужно на него подписаться с помощью метода `subscribe` - он принимает три функции:

1. Функция для обработки нового значения в потоке
2. Функция, которая вызывается когда в потоке происходит ошибка
3. Функция, которая вызывается когда поток завершается

ПОТОКИ

Каждая из этих функций необязательна. Также их можно сгруппировать в объект вида:

```
{
  next: (value) => {},
  error: (err) => {},
  complete: () => {}
}
```

Выведем значения в консоль:

```
userEmail$.subscribe(value => console.log(value));
```

Интересно то, что rxjs не добавит eventListener, пока мы не подпишемся на поток. Также важно не забыть отписываться от подписок.

ЗАЧЕМ НУЖНА ОТПИСКА?

Отписка нужна тогда, когда нам больше не нужны данные в потоке. Например, больше не нужно получать данные из формы.

Для того, чтобы отписаться нужно использовать объект подписки, который мы получаем когда выполняем `subscribe`:

```
const subscription = userEmail$.subscribe(value => console.log(value));

setTimeout(() => subscription.unsubscribe(), 4200);
```

После отписки должны освобождаться все ресурсы.

КАК СДЕЛАТЬ OBSERVABLE PROMISE

Для того, чтобы `Promise` превратить в `Observable` rxjs предоставляет функцию `from`. Предположим у нас есть Promise, который возвращает Fetch API. Сделаем из него поток:

```
const apiPromise = fetch('https://jsonplaceholder.typicode.com/posts');
from(apiPromise).subscribe(result => console.log(result.json()));
```

ЧТО ЕЩЕ ПОЛЕЗНОГО ДАЕТ RXJS?

У rxjs из коробки есть множество функций для создания потоков:

- `timer` - аналог `setTimeout`
- `interval` - аналог `setInterval`
- `bindCallback` - создает поток для обработки функции с callback-ом
- `range` - создает поток, в котором появляются числа в заданном интервале
- `from` - помимо промиса может принимать массив значений и по одному отправляет их в поток
- `of` - создает из значения поток из своих агрументов

ФУНКЦИИ СОЗДАНИЯ ПОТОКОВ

```
1  from([1, 2, 3]).subscribe(v => console.log(v));
2  // -1-2-3-|
3
4  of(1, 2, 3).subscribe(v => console.log(v));
5  // -1-2-3-|
6
7  of([1, 2, 3]).subscribe(v => console.log(v));
8  // -[1,2,3]-|
9
10 range(1, 3).subscribe(v => console.log(v));
11 // -1-2-3-|
```

PIPE

Часто бывает удобно преобразовывать значения в потоке. Например, в поток попадает событие элемента `HTMLInputElement`, а нам нужно, чтобы дальше проходило значение.

Для этого используются функции - операторы.

Оператор `pluck` принимает набор ключей, которые нужно вытащить из вложенных объектов:

```
pluck('target', 'value') // вытаскивает event.target.value
```

То же самое можно достичь с помощью более общего оператора `map`, который очень похож на метод `Array.map` - преобразует каждый элемент потока во что-то другое:

```
map(event => Number(event.target.value))
```

PIPE

Операторы можно последовательно комбинировать в потоке с помощью метода `pipe`. Например, так:

```
fromEvent(element, 'input').pipe(
  pluck('target', 'value'),
  map(value => Number(value)),
  filter(value => !isNaN(value))
)
```

В данном примере `filter` - оператор, который решает попадет ли значение дальше в поток. Он получает в качестве аргумента функцию, которая выполняется для каждого элемента потока. Если функция возвращает true, то значение проходит дальше, иначе игнорируется.

PIPE

Например, мы можем создать поток от 1 до 10, пропустить значения через `pipe` и с помощью оператора `filter` сделать так, чтобы в `subscribe` пришли только четные значения:

```
1  import { range } from 'rxjs';
2  import { filter } from 'rxjs';
3
4  range(1, 10)
5    .pipe(filter(value => value % 2))
6    .subscribe(v => console.log(v));
7  // -2-4-6-8-10-|
```

PIPE

Использование пайпов позволяет выносить отдельные части логики в самостоятельные функции и подключать их по необходимости. Например, можно легко сделать тротлинг значений при вводе данных в поле.

```
const input = document.getElementById('email');
fromEvent(input, 'input')
  .pipe(
    debounceTime(100),
    pluck('target', 'value'),
    filter(value => value.trim() !== ''),
    switchMap(value =>
      getRequest(`/api/check-email?email=${value}`).pipe(
        map(data => data.available),
        catchError(err => of(false))
      )
    )
  )
  .subscribe(value => {
    input.classList.remove('valid');
    input.classList.remove('error');
    input.classList.add(value ? 'valid' : 'error');
  });
```

CUSTOM OPERATORS

Помимо встроенных функций и операторов можно создавать свои.

Для того, чтобы создать поток с собственной логикой используется конструктор `Observable`.

`Observable` - это источник данных в RxJS, в котором определяется как данные будут попадать к подписчикам. Все функции, рассмотренные ранее, возвращают `Observable`.

Конструктор принимает функцию, которая описывает, как данные попадают в поток.

CUSTOM OPERATORS

У этой функции только один аргумент - объект `observer`, у которого есть три метода:

- `observer.next(value)` - оповестить подписчика о новом значении
- `observer.error(err)` - оповестить подписчика об ошибке
- `observer.complete()` - оповестить подписчика что значения в потоке закончились

После вызова `error` или `complete` подписчик больше не получит значений, даже если мы их попытаемся отправить.

ПРИМЕР

Для примера создадим поток, который выдает подписчику три случайных числа.

```
const stream$ = new Observable(observer => {
  setTimeout(() => observer.next(Math.random()), 100);
  setTimeout(() => observer.next(Math.random()), 200);
  setTimeout(() => observer.next(Math.random()), 300);
});
```

SUBSCRIBE

Итак мы объявили поток. Но функция, которая генерирует данные, не вызовется пока мы не подпишемся на этот поток. Это значит что функция, которую мы передали как аргумент, вызовется только когда мы подпишемся на поток.

Важное замечание: запомните, что пока вы не подписались, ничего происходить не будет!

UNSUBSCRIBE

Для того, чтобы при отписке освободить ресурсы (в примере отменить таймауты) нужно из функции, которая передается в new `Observable`, вернуть другую функцию, которая вызовется при вызове `unsubscribe`:

```
const stream$ = new Observable(observer => {
  const timer = setTimeout(() => observer.next(Math.random()), 100);

  return () => {
    clearTimeout(timer);
  };
});
```

Важное замечание: `Observable` является unicast потоком. Это значит, что для каждой подписки может быть только один потребитель. То есть при каждой подписки на поток функция заданная в конструкторе выполнится заново.

UNSUBSCRIBE

Что это значит на практике?

Рассмотрим пример, когда `Observable` генерирует одно случайное число:

```
const random$ = new Observable(observer => {
  observer.next(Math.floor(Math.random() * 100));
});

random$.subscribe(v => console.log(v)); // 42
random$.subscribe(v => console.log(v)); // 36
```

Для каждой из подписок сгенерируется свое число.

ОБРАБОТКА И ГЕНЕРАЦИЯ ОШИБОК

Иногда случается так, что в потоке появляется ошибка - обращение к несуществующему свойству, недопустимая операция, вызов несуществующей функции и т.п.

В этом случае поток полностью завершается и выполняется функция-обработчик ошибки в `subscribe`:

```
erroredStream$.subscribe(null, err => console.log(err));
```

Также бывает необходимо принудительно завершить поток с ошибкой. В этом случае достаточно выбросить обычное исключение - `throw new Error('error message')` и этот объект попадет в обработчик ошибки.

ОБРАБОТКА И ГЕНЕРАЦИЯ ОШИБОК

Другой сценарий - когда в потоке произошла ошибка нужно пустить вместо оригинального потока замещающий поток. Для этого используется оператор `catchError`.

Этот оператор внутри подписывается на входящий поток. Пока этот поток работает нормально оператор `catchError` пропускает значения в исходящий поток.

ОБРАБОТКА И ГЕНЕРАЦИЯ ОШИБОК

Когда во входящем потоке происходит ошибка он переключается на другой поток. Для того, что получить этот поток в `catchError` передается функция, которая его создаст. Например:

```
const erroredStream$ = new Observable(observer => {
  observer.emit(1);
  observer.emit(2);
  observer.emit(3);
  observer.error('out of boundries');
});

erroredStream$
  .pipe(
    catchError(err => {
      // err === 'out of boundries'
      // тут erroredStream$ завершился с ошибкой и больше не будет выдавать значения
      // но мы можем отдать замещающий поток
      return of(4, 5, 6);
    })
  )
  .subscribe(
    value => console.log('value', value),
    err => console.error('err', err)
  );
// 1-2-3-4-5-6-|
```

ОБРАБОТКА И ГЕНЕРАЦИЯ ОШИБОК

Иногда бывает так, что в `catchError` нужно проверить тип ошибки или просто ее залогировать и прокинуть ошибку дальше.

Для этого недостаточно просто вызвать `throw new Error()`. Нужно вернуть поток, который генерирует ошибку. Для этого используется функция rxjs `throwError`.

```
fromEvent(inputEl, 'input').pipe(
  switchMap(
    event => createObservableRequest(event.target.value)
    .pipe(
      catchError(err => {
        if (err.status > 500) {
          return throwError(new Error('internal server error'));
        }
        if (err.status > 400) {
          return throwError(new Error('invalid request'));
        }
        return of(null);
      })
    )
  )
)
```

ОБРАБОТКА И ГЕНЕРАЦИЯ ОШИБОК

Тут важно, что `catchError` «крепится» именно к запросу **внутри** `switchMap`. Это сделано потому, что нужно обработать ошибку запроса.

И если это не ошибка с кодом 4xx или 5xx, запрос как бы вернет значение `null`.

Если `catchError` вынести в основной `pipe` в любом случае после ошибки мы больше не получим значений из `inputEl`.

`Subject` в RXJS

`Subject` - это специальный тип `Observable` потока, который позволяет иметь несколько подписчиков.

`Subject` очень похож на `EventEmitter` из Node.js - он тоже хранит внутри список подписчиков.

`Subject` - это не только `Observable` поток (на него можно подписаться), но и `Observer` - у него есть методы `next`, `error` и `complete`.

Subject

```
1  import { Subject } from 'rxjs';
2
3  const stream$ = new Subject();
4
5  stream$.subscribe(v => console.log(`observerA: ${v}`));
6  stream$.subscribe(v => console.log(`observerB: ${v}`));
7
8  stream$.next(1);
9  stream$.next(2);
10
11 // Logs:
12 // observerA: 1
13 // observerB: 1
14 // observerA: 2
15 // observerB: 2
```

Subject

Для лучшего понимания перепишем пример со случайными числами так, чтобы все подписчики получили одинаковое значение:

```
const randomUnicast$ = new Observable(observer => {
  observer.next(Math.floor(Math.random() * 100));
});

const random$ = new Subject();

random$.subscribe(v => console.log(v)); // 42
random$.subscribe(v => console.log(v)); // 42

randomUnicast$.subscribe(random$);
```

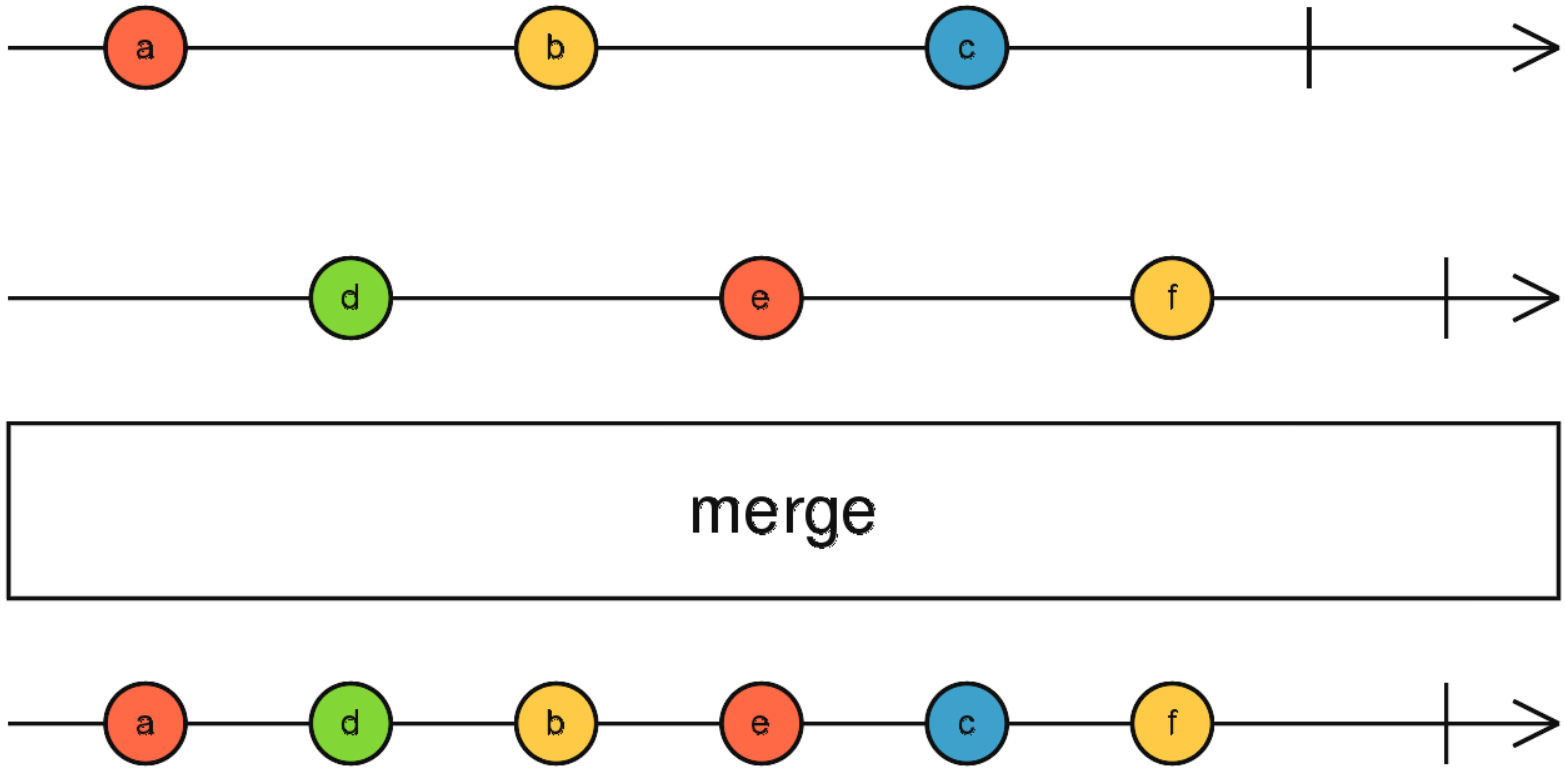
HOO - OBSERVABLE ВЫСШЕГО ПОРЯДКА

Часто бывает так, что в потоке нужно переключиться на другой поток и слить два потока в один. Например нужно сделать два http запроса и объединить их результат.

Для этого в rxjs есть специальные функции, которые принимают в качестве аргумента `Observable`.

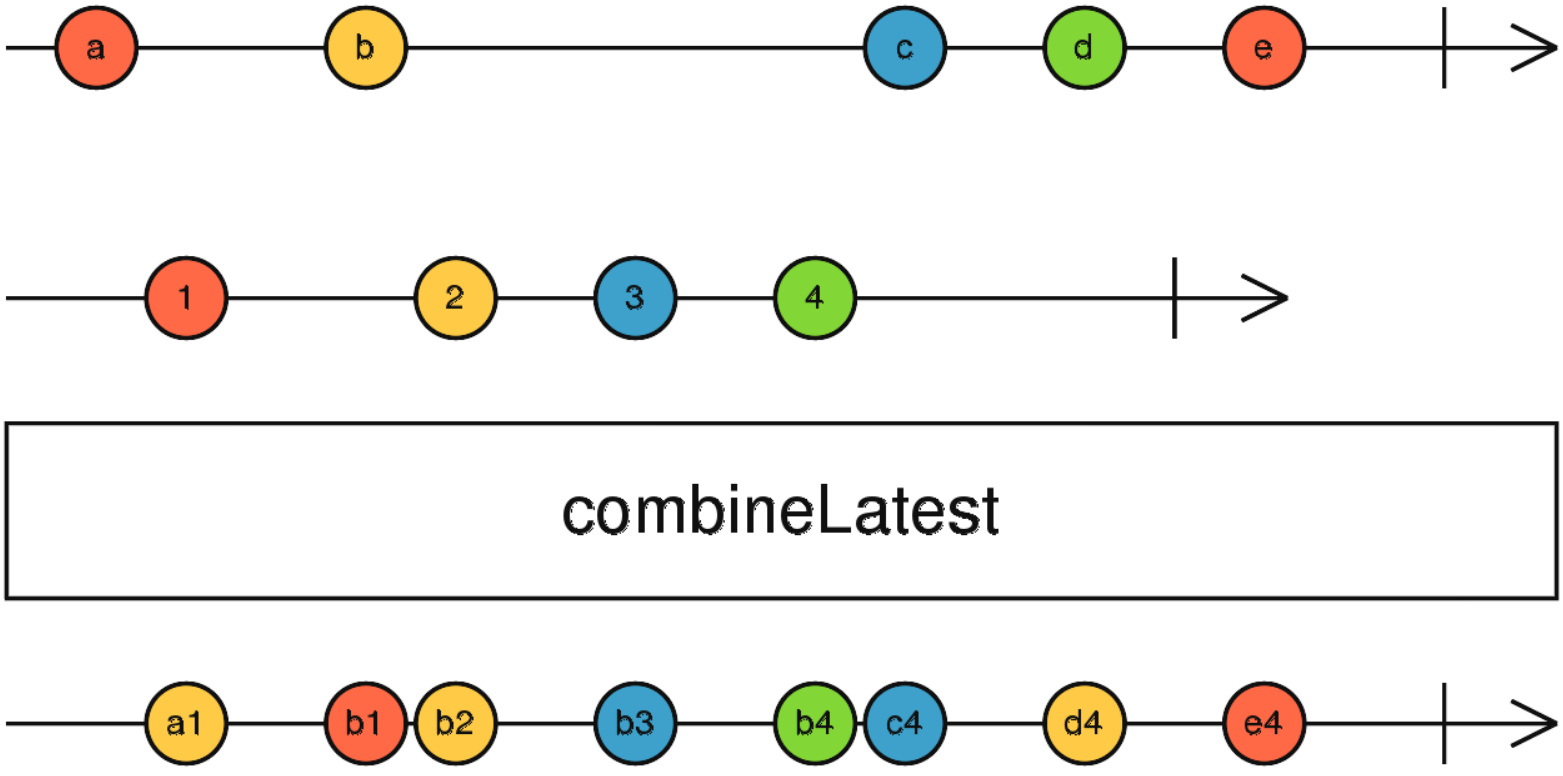
`merge()`

Оператор `merge` принимает любое количество `Observable` и эмитит в общий поток.



`combineLatest()`

Оператор `combineLatest` принимает любое количество `Observable` и эмитит в общий поток массив с последним значением каждого потока.



*Map

Помимо подобных функций существуют операторы, которые позволяют переключить поток внутри pipe:

- `mergeMap` : выполняет все параллельно
- `concatMap` : выполняет все последовательно
- `switchMap` : выполняет новый, а предыдущий отменяет
- `exhaustMap` : игнорирует все новые, пока предыдущий не завершится

ПРИМЕР

Для того, чтобы посмотреть, как это работает, сделаем специальную функцию. Она должна принимать число и возвращать `Observable`.

При подписке на этот `Observable` должен начаться запрос к серверу, а при отписке прерваться если он еще не выполнился.

```
import { Observable } from 'rxjs';
function getRequest(url) {
  return new Observable(observer => {
    const controller = new AbortController();

    fetch(url, {
      signal: controller.signal
    })
      .then(res => res.json())
      .then((data) => {
        observer.next(data);
        observer.complete();
      })
      .catch(err => observer.error(err));

    return () => controller.abort();
  });
}
```

ABORTCONTROLLER

`AbortController` - объект, позволяющий прерывать запросы, производимые с помощью Fetch API. Описан в спецификации DOM.

Напоминаем, что в XHR для этого существует метод `abort`.

mergeMap()

Создает `Observable` для каждого входящего значения и результы всех потоков объединяет в один без учета очереди:

```
range(1, 3)
  .pipe(mergeMap(id => getRequest(`https://jsonplaceholder.typicode.com/posts/${id}`)))
  .subscribe(value => console.log(v));
```

Параллельно выполнит три запроса. В консоль выведутся результаты всех запросов, но порядок не гарантирован.

concatMap()

Создает `Observable` для каждого входящего значения и ставит его в очередь. Когда один `Observable` завершится подписывается на следующий. Таким образом гарантирует порядок значений:

```
range(1, 3)
  .pipe(concatMap(id => getRequest(`https://jsonplaceholder.typicode.com/posts/${id}`)))
  .subscribe(value => console.log(v));
```

1. Когда придет значение `id=1` начнет новый запрос.
2. Когда придет значение `id=2` дождется пока придет ответ от запроса `id=1` и начнет новый запрос.
3. Когда придет значение `id=3` дождется пока придет ответ от запроса `id=2` и начнет новый запрос.

Таким образом в консоль будут выведены три объекта по порядку.

switchMap()

Создает `Observable` для входящего значения и отписывается от потока, созданного для предыдущего значения:

```
range(1, 3)
  .pipe(switchMap(id => getRequest(`https://jsonplaceholder.typicode.com/posts/${id}`)))
  .subscribe(value => console.log(v));
```

Начнет запрос для `id=1`, но не дождется запроса из-за нового значения 2 и отменит запрос. Аналогично для `id=2`.

Для `id=3` следующего значения нет, поэтому запрос выполнится и в консоль выведется только результат последнего запроса.

exaustMap()

Создает `Observable` для входящего значения. Пока этот `Observable` не завершится игнорирует все новые значения:

```
range(1, 3)
  .pipe(exaustMap(id => getRequest(`https://jsonplaceholder.typicode.com/posts/${id}`)))
  .subscribe(value => console.log(v));
```

1. Когда придет значение `id=1` начнет новый запрос.
2. Значение `id=2` будет проигнорировано, потому что выполняется запрос для `id=1`.
3. Значение `id=3` будет проигнорировано, потому что выполняется запрос для `id=1`.

Таким образом в консоль будет выведен только результат первого запроса.

ПРИМЕР: DRAG & DROP

ПРИМЕР DRAG & DROP

```
import { fromEvent } from 'rxjs';
import { takeUntil, switchMap, map } from 'rxjs/operators';

const draggable = (el) => {
  const startDrag$ = fromEvent(el, 'mousedown');
  const moveDrag$ = fromEvent(document, 'mousemove');
  const endDrag$ = fromEvent(el, 'mouseup');

  return startDrag$.pipe(
    switchMap((event) => {
      event.stopPropagation();
      const diffX = el.offsetLeft - event.clientX;
      const diffY = el.offsetTop - event.clientY;
      return moveDrag$.pipe(
        map((event) => {
          const { clientX, clientY } = event;
          return {
            x: clientX + diffX,
            y: clientY + diffY
          };
        }),
        takeUntil(endDrag$),
      );
    })
  );
};

draggable(draggableEl).subscribe(coord => {
  draggableEl.style.top = `${coord.y}px`;
  draggableEl.style.left = `${coord.x}px`;
});
```

ПРИМЕР: STATE MANAGER НА RXJS

ПРИМЕР STATE MANAGER'A НА RXJS

State Manager - это способ управления состоянием страницы.

State Manager хранит в себе текущее состояние и в ответ на команды с помощью единой функции-редьюсера высчитывает новое состояние.

СПИСОК ДЕЙСТВИЙ

Для начала определим список действий, которые могут изменять состояние:

```
const Actions = {
  Increment: 'INCREMENT',
  Decrement: 'DECREMENT',
  Reset: 'RESET'
};
```

Само действие это объект состощий из двух полей:

- type - обязательное, поясняет что произошло
- payload - необязательное, дает дополнительные данные

```
{
  'type': 'INCREMENT',
  'payload': 5
}
```

REDUCER

Reducer - это чистая функция, которая принимает текущее состояние и действие и возвращающая новое состояние. Так как функция чистая, то исходное состояние не меняется.

```
function reduce(state, action) {
  switch (action.type) {
    case Actions.Increment:
      return { ...state, counter: state.counter + action.payload };
    case Actions.Decrement:
      return { ...state, counter: state.counter - action.payload };
    case Actions.Reset:
      return { ...state, counter: 0 };
    default:
      return state;
  }
}
```

STORE

Store - это класс, который создает поток событий и превращает его в поток состояний.

```
class Store {
  constructor() {
    this.actions$ = new Subject();
    this.state$ = this.actions$.asObservable().pipe(
      startWith({ type: '__INITIALIZATION__' }),
      scan((state, action) => reduce(state, action), { counter: 0 }),
      share()
    );
  }

  dispatch(type, payload = null) {
    this.actions$.next({ type, payload });
  }

  inc(value = null) {
    this.dispatch(Actions.Increment, value);
  }
  dec(value = null) {
    this.dispatch(Actions.Decrement, value);
  }
  reset() {
    this.dispatch(Actions.Reset);
  }
}
```

СОЕДИНЯЕМ STORE С ИНТЕРФЕЙСОМ

После того, как мы создали объект Store нам нужно подписаться на изменение его состояния.

```
const store = new Store();

store.state$
  .pipe(
    pluck('counter'),
    distinctUntilChanged()
  )
  .subscribe(value => {
    document.getElementById('counterValue').text = value;
  });
```

СОЕДИНЯЕМ STORE С ИНТЕРФЕЙСОМ

На события в интерфейсе реагируем вызовом методов Store:

```
const step$ = fromEvent(document.getElementById('stepInput'), 'input').pipe(
  map(e => Number(e.target.value) || null),
  startWith(null)
);

fromEvent(document.getElementById('incButton'), 'click')
  .pipe(withLatestFrom(step$))
  .subscribe(([event, step]) => store.inc(step));

fromEvent(document.getElementById('decButton'), 'click')
  .pipe(withLatestFrom(step$))
  .subscribe(([event, step]) => store.dec(step));

fromEvent(document.getElementById('resetButton'), 'click').subscribe(
  ([event, step]) => store.reset()
);
```

STATE MANAGER

Таким образом мы получили "общее состояние страницы", на которое могут подписываться различные виджеты.

ИТОГИ

ИТОГИ

Сегодня мы разобрали библиотеку rxjs и посмотрели, какие подходы она предлагает для решения проблем работы с данными.

Кроме того, обсудили подходы к управлению состоянием страницы, которые сейчас используются в том числе в современных фреймворках.

RXJS V5 VS V6

Обратите внимание, что многие руководства в сети ориентируются на 5-ую версию rxjs, в то время как мы с вами используем 6-ую.

Между этими версиями есть достаточно существенные различия в синтаксисе.

RXJS MARBLES

RxJS Marbles - полезный сайт с интерактивными диаграммами, позволяющий понять работу операторов.

НЕТОЛОГИЯ

Спасибо за внимание!

Время задавать вопросы 🙂

АЛЕКСЕЙ КУЛАГИН

@alex_kulagin