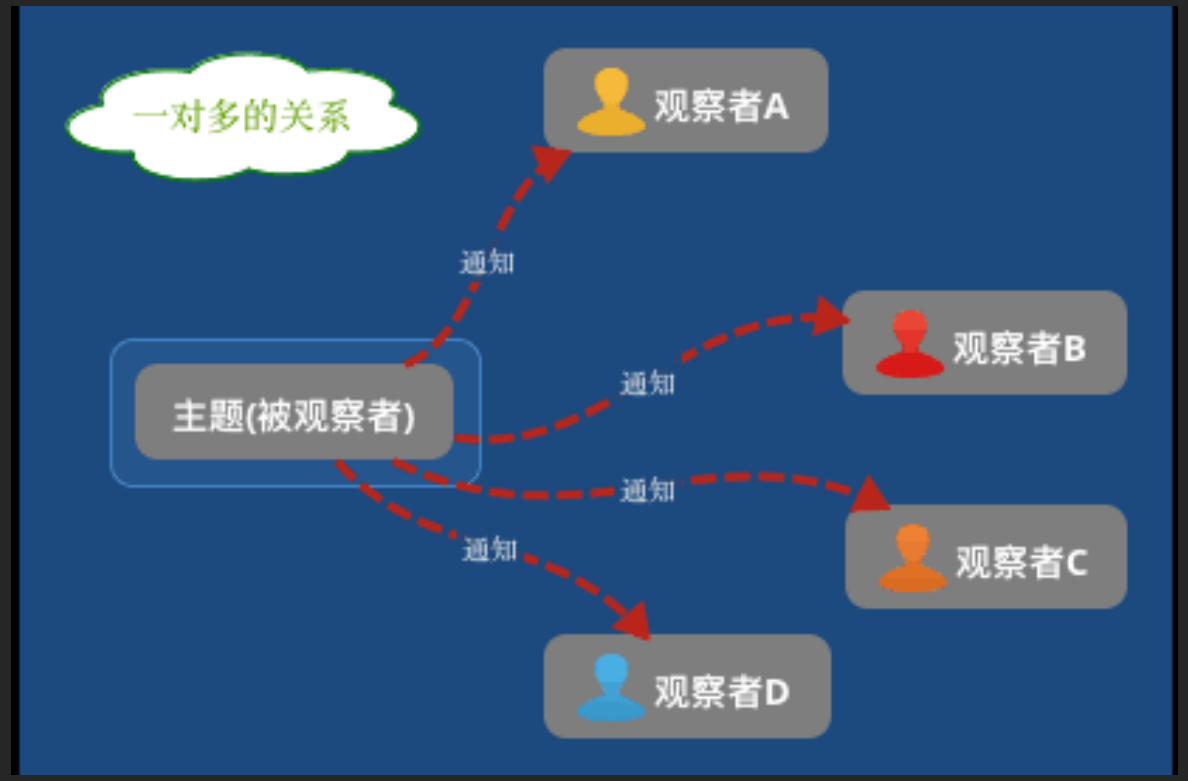

RxJS

目录

01

RxJS(Reactive Extensions for JavaScript) 是Reactive Extensions 在 JavaScript 上的实现。基于观察者模式和迭代器模式以函数式编程思维来实现的



RxJS中observable对象是一个被观察者（发布者），通过Observable对象的Subscribe函数，可以把这个发布者和观察者（observer）连接起来

发布：Observable 通过回调 next 方法向 Observer 发布事件。

01 迭代器模式体现

```ts
export interface Observer<T> {
  closed?: boolean;
  next: (value: T) => void;
  error: (err: any) => void;
  complete: () => void;
}
```

Observer 提供一个 next 方法来接收 Observable 流

complete() 当不再有新的值发出时，将触发 Observer 的 complete 方法；而在 Iterator 中，则需要在 next 的返回结果中，当返回元素 done 为 true 时，则表示 complete。

error()当在处理事件中出现异常报错时，Observer 提供 error 方法来接收错误进行统一处理；Iterator 则需要进行 try catch 包裹来处理可能出现的错误。

02 Subscribe

```
source$ = new Observable( subscribe: observer => {
  let num = 0;
  setInterval( callback: () => observer.next(num++), ms: 1000);
});

theObserver = {
  next: val => {
      console.log(val);
    },
  error: err => console.log(err),
  complete: () => console.log('no more data'),
};

constructor() { }

ngOnInit(): void {
  const ub = this.source$.subscribe(this.theObserver);
  setTimeout( handler: () => {
    ub.unsubscribe();
  }, timeout: 4000);
```

```
export class Observable<T> implements Subscribable<T> {

  /** Internal implementation detail, do not use directly. */
  public _isScalar: boolean = false;

  /** @deprecated This is an internal implementation detail, do not use.
  source: Observable<any> | undefined;

  /** @deprecated This is an internal implementation detail, do not use.
  operator: Operator<any, T> | undefined;

  /**
   * @constructor
   * @param {Function} subscribe the function that is called when the Obs
   * initially subscribed to. This function is given a Subscriber, to whi
   * can be `next`ed, or an `error` method can be called to raise an erro
   * `complete` can be called to notify of a successful completion.
   */
  constructor(subscribe?: (this: Observable<T>, subscriber: Subscriber<T>
    if (subscribe) {
      this._subscribe = subscribe;
    }
```

创建Observable时传入的函数（rt）是在Observable订阅的时候调用的

02

在 RxJS 里面，为开发者提供了一些保障机制，来保证一个更安全的观察者。
toSubscriber来做这件事。

```
subscribe(observerOrNext?: PartialObserver<T> | ((value: T) => void) | null,
          error?: ((error: any) => void) | null,
          complete?: (() => void) | null): Subscription {

  const { operator } = this;
  const sink = toSubscriber(observerOrNext, error, complete);

  if (operator) {
    sink.add(operator.call(sink, this.source));
  } else {
    sink.add(
      this.source || (config.useDeprecatedSynchronousErrorHandling && !sink.syncErrorThrow
      this._subscribe(sink) :
      this._trySubscribe(sink)
    );
  }

  if (config.useDeprecatedSynchronousErrorHandling) {
    if (sink.syncErrorThrowable) {
      sink.syncErrorThrowable = false;
      if (sink.syncErrorThrown) {
        throw sink.syncErrorValue;
      }
    }
  }

  return sink;
}
```

实例中的Observable是没有operator的，操作符生成的Observable有这个属性，订阅的时候调用。

03 在项目中引入操作符

给Observable打补丁

使用bind或call绑定特定Observable对象

pipeable操作符

```
import 'rxjs/add/operator/take';
Observable.prototype.take = take;
```

这会将导入的操作符添加到 Observable 的原型。

```
import { take } from 'rxjs/operator/take';
import { map } from 'rxjs/operator/map';
import { of } from 'rxjs/observable/of';
map.call(take.call(of(1,2,3),  2),val => val + 2);
```

```
import { take, map } from 'rxjs/operators';
import { of } from 'rxjs/observable/of';
of(1,2,3).pipe(take(2),map(val => val + 2));
```

RxJS 提供了 pipe 辅助函数，它存在于 Observable 上,缓解了打补丁方式所带来的问题，也实现了链式调用

03 pipeable操作符介绍

pipeable 操作符可以是任何函数，但是它需要返回签名为 <T, R>(source: Observable<T>) => Observable<R> 的函数。

```
dest$ = this.source$.pipe(
  map( project: value => value * 2 )
);
```

dest$.subscribe（observe）

↓

dest$.operation.call(sink,dest$.soucrce)

↓

src$.subscribe (new MapSubscriber(observe, project))

```
*/
export function map<T, R>(project: (value: T, index: number) => R, thisArg?: any): OperatorF
  return function mapOperation(source: Observable<T>): Observable<R> {
    if (typeof project !== 'function') {
      throw new TypeError('argument is not a function. Are you looking for `mapTo()`?');
    }
    return source.lift(new MapOperator(project, thisArg));
  };
}

lift<R>(operator?: Operator<T, R>): Observable<R> {
  const observable = new Observable<R>();
observable.source = this;
observable.operator = operator;
return observable;
}

export class MapOperator<T, R> implements Operator<T, R> {
  constructor(private project: (value: T, index: number) => R, private thisArg: any) {
  }

  call(subscriber: Subscriber<R>, source: any): any {
    return source.subscribe(new MapSubscriber(subscriber, this.project, this.thisArg));
  }
}

class MapSubscriber<T, R> extends Subscriber<T> {
  count: number = 0;
  private thisArg: any;

  constructor(destination: Subscriber<R>,
              private project: (value: T, index: number) => R,
              thisArg: any) {
    super(destination);
    this.thisArg = thisArg || this;
  }
  protected _next(value: T) {
    let result: R;
    try {
      result = this.project.call(this.thisArg, value, this.count++);
    } catch (err) {
      this.destination.error(err);
      return;
    }
    this.destination.next(result);
  }
}
```

03 pipe链式调用的实现

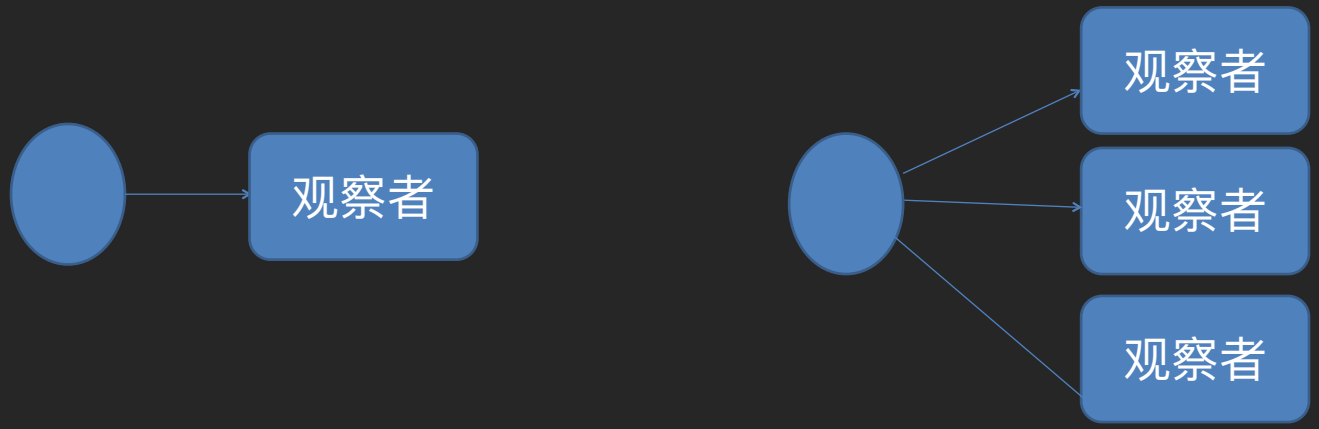
```
 */
pipe(...operations: OperatorFunction<any, any>[]): Observable<any> {
  if (operations.length === 0) {
    return this as any;
  }

  return pipeFromArray(operations)(this);
}
export function pipeFromArray<T, R>(fns: Array<UnaryFunction<T, R>>): UnaryFunction<T, R> {
  if (fns.length === 0) {
    return identity as UnaryFunction<any, any>;
  }

  if (fns.length === 1) {
    return fns[0];
  }

  return function piped(input: T): R {
    return fns.reduce((prev: any, fn: UnaryFunction<T, R>) => fn(prev), input as any);
  };
}
```

reduce函数使得operations按照它们在数组中的顺序先后执行

fn(prev)使得当前的operation接收上一个operation的执行结果作为参数

每一个operator(操作符)都被包装成了Observable，并通过source属性互相链接

04 多播



Cold Observable实现的是单播， Cold Observable 只有观察者订阅时，才开始执行发射数据流的代码。并且 Cold Observable 和 观察者只能是一对一的关系，当有多个不同的观察者时，数据流是重新完整发送的。

Hot Observable实现的是多播，Hot Observable 无论有没有 观察者 订阅，事件始终都会发生。当 Hot Observable 有多个订阅者时，它可以确保每个观察者接收到的数据绝对相等。

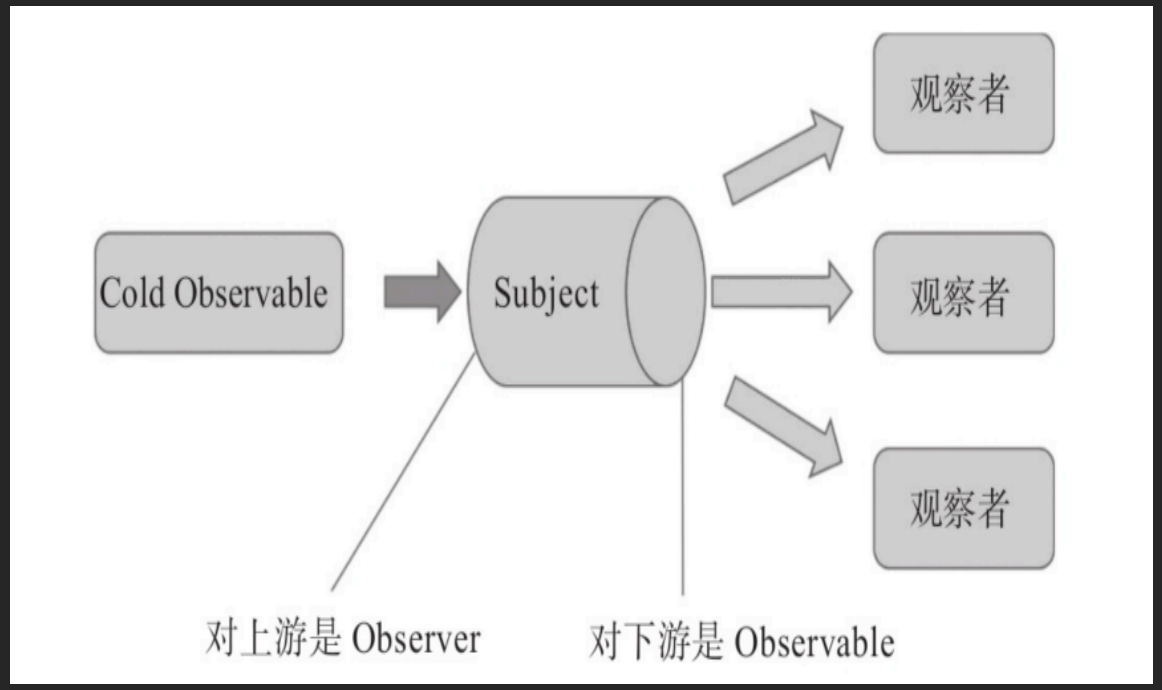
04

Subject、 BehaviorSubject、 ReplaySubject实现了 Observable 的多播。

```
export class Subject<T> extends Observable<T> implements SubscriptionLike {

  [rxSubscriberSymbol]() {
    return new SubjectSubscriber(this);
  }

  observers: Observer<T>[] = [];

  next(value: T) {
    if (this.closed) {
      throw new ObjectUnsubscribedError();
    }
    if (!this.isStopped) {
      const { observers } = this;
      const len = observers.length;
      const copy = observers.slice();
      for (let i = 0; i < len; i++) {
        copy[i].next(value!);
      }
    }
  }
  _subscribe(subscriber: Subscriber<T>): Subscription {
    if (this.closed) {
      throw new ObjectUnsubscribedError();
    } else if (this.hasError) {
      subscriber.error(this.thrownError);
      return Subscription.EMPTY;
    } else if (this.isStopped) {
      subscriber.complete();
      return Subscription.EMPTY;
    } else {
      this.observers.push(subscriber);
      return new SubjectSubscription(this, subscriber);
    }
  }
```

Subject对象即是一个Observable也是一个observer, 当订阅Subject时,Subject会将该观察者添加到内部的观察者列表中。当调用 Subject 的 next(value) 时，它会遍历观察者列表并将 value 传递给 next 方法。要想从 subject 的观察者列表中移除该观察者，只需简单调用subscription的unsubscribe方法即可。

04



Subject与上下游的关系

多播操作符 multicast

```
source$ = interval(1000);
tick$ = this.source$.pipe(
  take(5),
  multicast(new Subject()),
  refCount()
);
```

能够以上游的Observable为数据源,利用Subject 产生一个新的HotObservable对象

05 高级多播功能 BehaviorSubject

```
 */
export class BehaviorSubject<T> extends Subject<T> {

  constructor(private _value: T) {
    super();
  }

  get value(): T {
    return this.getValue();
  }

  /** @deprecated This is an internal implementation detail, do not us
  _subscribe(subscriber: Subscriber<T>): Subscription {
    const subscription = super._subscribe(subscriber);
    if (subscription && !(<SubscriptionLike>subscription).closed) {
      subscriber.next(this._value);
    }
    return subscription;
  }

  getValue(): T {
    if (this.hasError) {
      throw this.thrownError;
    } else if (this.closed) {
      throw new ObjectUnsubscribedError();
    } else {
      return this._value;
    }
  }

  next(value: T): void {
    super.next( value: this._value = value);
  }
}
```

业务场景： 提供默认数据或者获取一些数据并想让应用记住最新获取的数据

05 高级多播功能 ReplaySubject 和publishReplay

ReplaySubject: 可以给新订阅者发送“旧”数据的，有_events属性存储一些旧的数据，当有新的观察者订阅时，首先会“重播”_events内存储的数据给新来的观察者，创建 ReplaySubject 时，可以指定存储的数据量以及数据的过期时间。

```
export function publishReplay<T, R>(bufferSize?: number,
                                     windowTime?: number,
                                     selectorOrScheduler?: SchedulerLike | Operat
                                     scheduler?: SchedulerLike): UnaryFunction<Ob

  if (selectorOrScheduler && typeof selectorOrScheduler !== 'function') {
    scheduler = selectorOrScheduler;
  }

  const selector = typeof selectorOrScheduler === 'function' ? selectorOrSchedul
  const subject = new ReplaySubject<T>(bufferSize, windowTime, scheduler);

  return (source: Observable<T>) => multicast(() => subject, selector!)(source)
}
```

publishReplay操作符 是直接调用multicast操作符，只不过使用的Subject对象不同， 使用的是返回值为ReplaySubject函数的

业务场景：获取一些数据并想让应用记住最新获取的数据，同时获取的内容可能只在一段时间内是有效的，并且在保留足够的时间后会清除缓存。

05 高级多播操作符shareReplay

业务场景：当有副作用或繁重的计算时，不希望在多个订阅者之间重复执行 或者流的后来订阅者也需要访问之前发出的值

```
tick$ = this.source$.pipe(
  shareReplay(1)
)
```

=

```
tick$ = this.source$.pipe(
  publishReplay(1),
  refCount()
);
```

```
function shareReplayOperator<T>({
  bufferSize = Infinity,
  windowTime = Infinity,
  refCount: useRefCount,
  scheduler
}: ShareReplayConfig) {
  let subject: ReplaySubject<T> | undefined;
  let refCount = 0;
  let subscription: Subscription | undefined;
  let hasError = false;

  return function shareReplayOperation(this: Subscriber<T>, source: Observable<T>) {
    refCount++;
    if (!subject || hasError) {
      hasError = false;
      subject = new ReplaySubject<T>(bufferSize, windowTime, scheduler);
      subscription = source.subscribe( observer: {
        next(value) { subject.next(value); },
        error(err) {
          hasError = true;
          subject!.error(err);
        },
        complete() {
          subscription = undefined;
          subject!.complete();
        },
      });
    }

    const innerSub = subject.subscribe(this);
    this.add(() => {
      refCount--;
      innerSub.unsubscribe();
      if (subscription && useRefCount && refCount === 0) {
        subscription.unsubscribe();
        subscription = undefined;
        subject = undefined;
      }
    });
  };
```


Q&A