Observable 与 RxJS（响应式编程）

RxJS 是一个库，它通过使用 Observable 序列来编写异步和基于事件的程序。它提供了一个核心类型 Observable，附属类型（Observer、Schedulers、Subjects）和受 [Array#extras] 启发的操作符（map、filter、reduce、every, 等等），这些数组操作符可以把异步事件作为集合来处理

基本概念

在 RxJS 中用来解决异步事件管理的的基本概念是：

- Observable（可观察对象）：表示一个概念，这个概念是一个可调用的未来值或事件的集合
- Observer（观察者）：一个回调函数的集合，它知道如何去监听由 Observable 提供的值
- Subscription（订阅）：表示 Observable 的执行，主要用于取消 Observable 的执行
- Operators（操作符）： 采用函数式编程风格的纯函数（pure function），使用像 map、filter、concat、flatMap 等这样的操作符来处理集合
- Subject（主体）： 相当于 EventEmitter，并且是将值或事件多路推送给多个 Observer 的唯一方式
- Schedulers（调度器）：用来控制并发并且是中央集权的调度员，允许我们在发生计算时进行协调，例如 setTimeout 或 requestAnimationFrame 或其他

> 纯函数：如果函数的调用参数相同，则永远返回相同的结果。它不依赖于程序执行期间函数外部任何状态或数据的变化，必须只依赖于其输入参数

创建可观察对象

Observable 可以使用 create 来创建，Rx.Observable.create 是 Observable 构造函数的别名，它接收一个参数：subscribe 函数。 但通常我们使用所谓的创建操作符， 像 of、from、interval、等等

```
// 构造函数
const observable = new Observable((subscriber) => {
  try {
    subscriber.next(1);
    subscriber.next(2);
    subscriber.next(3);
    subscriber.complete();
  } catch (err) {
    subscriber.error(err);
  }
});

// 创建操作符
const observable = of(1, 2, 3);
const observable = from([1, 2, 3]);
const observable = fromEvent(document.body, 'click');
```

订阅

只有当有人订阅 Observable 的实例时，它才会开始发布值。 订阅时要先调用该实例的 subscribe() 方法，并把一个观察者对象传给它，用来接收通知

```
observable.subscribe(
  (res) => console.log(res),
  (err) => console.error(err),
  () => console.log('complete')
);
```

执行

Observable 是惰性运算，只有在每个观察者订阅后才会执行。随着时间的推移，执行会以同步或异步的方式产生多个值

```
const observable = new Observable((subscriber) => {
  // 以下 try-catch 即是执行
  try {
    subscriber.next(1);
    subscriber.next(2);
    subscriber.next(3);
    subscriber.complete();
  } catch (err) {
    subscriber.error(err);
  }
});
```

清理

因为 Observable 执行 **可能** 会是无限的，并且观察者通常希望能在有限的时间内中止执行，所以我们需要一个 API 来取消执行。因为每个执行都是其对应观察者专属的，一旦观察者完成接收值，它必须要一种方法来停止执行，以避免浪费计算能力或内存资源

> 一般的 Observable 也有可能在内部执行 complete() 或者 error() 终止 Observable

```
var observable = from([10, 20, 30]);
var subscription = observable.subscribe((res) => console.log(res));
// 在不需要的时候，取消观察
subscription.unsubscribe();
```

Subject（主体）

Subject 是一种特殊类型的 Observable，它允许将值多播给多个观察者，所以 Subject 是多播的，而普通的 Observable 是单播的（每个已订阅的观察者都拥有 Observable 的独立执行）

```
var subject = new Subject();

subject.subscribe({
  next: (v) => console.log('observerA: ' + v),
});
subject.subscribe({
  next: (v) => console.log('observerB: ' + v),
});

subject.next(1);
subject.next(2);
```

Subject（主体）的三类变体：

- BehaviorSubject

  BehaviorSubject 有一个“当前值”的概念，它保存了发送给消费者的最新值，并且当有新的观察者订阅时，会立即从 BehaviorSubject 那接收到“当前值”

- ReplaySubject

  ReplaySubject 类似于 BehaviorSubject，它可以发送旧值给新的订阅者，但它还可以记录 Observable 执行的一部分

- AsyncSubject

  AsyncSubject 只有当 Observable 执行完成时（执行 complete()），它才会将执行的最后一个值发送给观察者

操作符

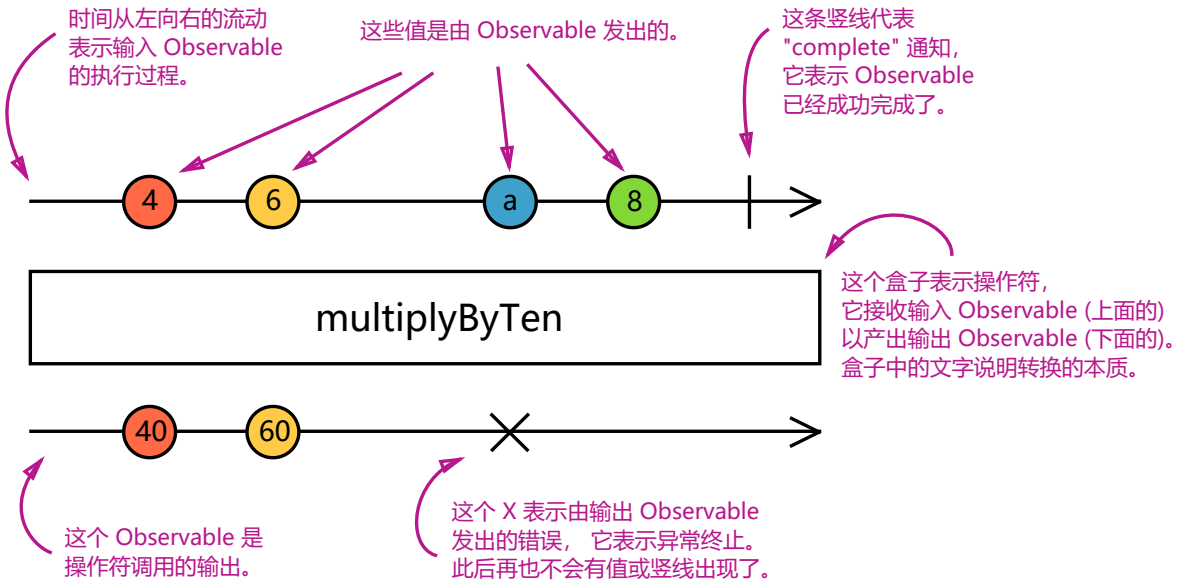
操作符是 Observable 类型上的方法，比如 .map(...)、.filter(...)、.merge(...)，等等。当操作符被调用时，它们不会改变已经存在的 Observable 实例。相反，它们返回一个新的 Observable ，它的 subscription 逻辑基于第一个 Observable

> 操作符是函数，它基于当前的 Observable 创建一个新的 Observable。这是一个无副作用的操作：前面的 Observable 保持不变

操作符本质上是一个纯函数 (pure function)，它接收一个 Observable 作为输入，并生成一个新的 Observable 作为输出

Marble diagrams (弹珠图)

要解释操作符是如何工作的，文字描述通常是不足以描述清楚的。许多操作符都是跟时间相关的，它们可能会以不同的方式延迟(delay)、取样(sample)、节流(throttle)或去抖动值(debonce)。图表通常是更适合的工具。弹珠图是操作符运行方式的视觉表示，其中包含输入 Obserable(s) (输入可能是多个 Observable)、操作符及其参数和输出 Observable

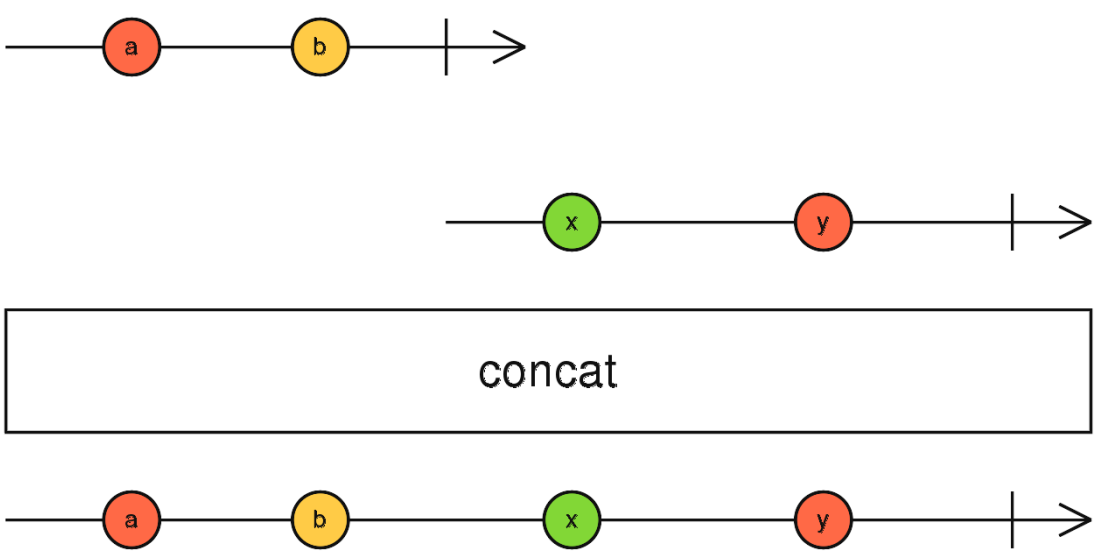


常用操作符

类别	操作
创建	from,fromEvent,fromPromise, of
组合	combineLatest, concat, merge, startWith , withLatestFrom, zip
过滤	debounceTime, distinctUntilChanged, filter, take, takeUntil
转换	bufferTime, concatMap, map, mergeMap, scan, switchMap
工具	tap
多播	share

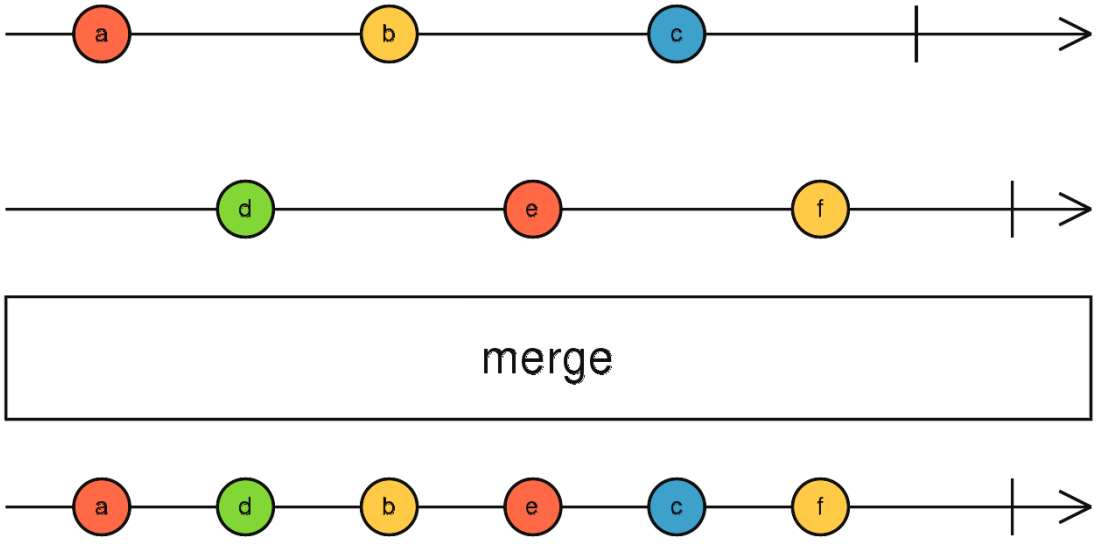
- concat

  通过顺序地发出多个 Observables 的值将它们连接起来，一个接一个的



- merge

  通过把多个 Observables 的值混合到一个 Observable 中来将其打平

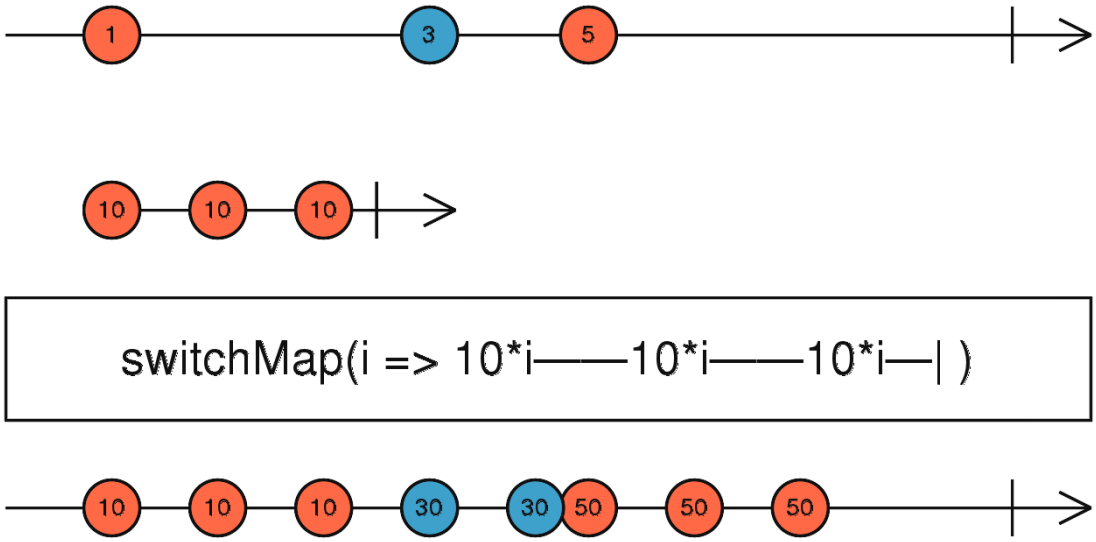


- switchMap

将每个源值投射成 Observable，该 Observable 会合并到输出 Observable 中， 并且只发出最新投射的 Observable 中的值

> 将每个值映射成 Observable ， 然后使用 switch 打平所有的内部 Observables

返回的 Observable 基于应用一个函数来发送项，该函数提供给源 Observable 发出的每个项， 并返回一个所谓的“内部” Observable，每次观察到这些内部 Observables 的其中一个时，输出 Observable 将开始发出该内部 Observable 所发出的项。当发出一个新的内部 Observable 时，switchMap 会停止发出先前发出的内部 Observable 并开始发出新的内部 Observable 的值



响应式表单

响应式表单提供了一种模型驱动的方式来处理表单输入，其中的值会随时间而变化。本文会向你展示如何创建和更新基本的表单控件，接下来还会在一个表单组中使用多个控件，验证表单的值，以及创建动态表单，也就是在运行期添加或移除控件

providedIn 与 NgModule

当你把服务提供者添加到应用的根注入器中时，它就在整个应用程序中可用了。 另外，这些服务提供者也同样对整个应用中的类是可用的 —— 只要它们有供查找用的服务令牌

你应该始终在根注入器中提供这些服务 —— 除非你希望该服务只有在消费方要导入特定的 @NgModule 时才生效

也可以规定某个服务只有在特定的 @NgModule 中提供。比如，如果你希望只有当消费方导入了你创建的 UserModule 时才让 UserService 在应用中生效，那就可以指定该服务要在该模块中提供：

```
import { Injectable } from '@angular/core';
import { UserModule } from './user.module';

@Injectable({
  providedIn: UserModule,
})
export class UserService {}
```

上面的例子展示的就是在模块中提供服务的首选方式。之所以推荐该方式，是因为当没有人注入它时，该服务就可以被摇树优化掉。如果没办法指定哪个模块该提供这个服务，你也可以在那个模块中为该服务声明一个提供者：

```
import { NgModule } from '@angular/core';
import { UserService } from './user.service';

@NgModule({
  providers: [UserService],
})
export class UserModule {}
```

单例服务

在 Angular 中有两种方式来生成单例服务：

- 把 @Injectable() 中的 providedIn 属性设置为 "root"
- 把该服务包含在 AppModule 或某个只会被 AppModule 导入的模块中

```
import { Injectable } from '@angular/core';

@Injectable({
  providedIn: 'root',
})
export class UserService {}
```

HTTP 拦截器

借助拦截机制，你可以声明一些拦截器，它们可以检查并转换从应用中发给服务器的 HTTP 请求。这些拦截器还可以在返回应用的途中检查和转换来自服务器的响应。多个拦截器构成了请求/响应处理器的双向链表

拦截器可以用一种常规的、标准的方式对每一次 HTTP 的请求/响应任务执行从认证到记日志等很多种隐式任务

要实现拦截器，就要实现一个实现了 HttpInterceptor 接口中的 intercept() 方法的类

```
import { Injectable } from '@angular/core';
import {
  HttpEvent,
  HttpInterceptor,
  HttpHandler,
  HttpRequest,
} from '@angular/common/http';

import { Observable } from 'rxjs';

@Injectable()
export class NoopInterceptor implements HttpInterceptor {
  intercept(
    req: HttpRequest<any>,
    next: HttpHandler
  ): Observable<HttpEvent<any>> {
    return next.handle(req);
```

```
  }
}
```

由于在 AppModule 中导入了 HttpClientModule，导致本应用在其根注入器中提供了 HttpClient。所以你也同样要在 AppModule 中提供这些拦截器

```
{ provide: HTTP_INTERCEPTORS, useClass: NoopInterceptor, multi: true }
```

HTTP 服务代理

项目根文件夹下新建代理配置 json 文件（proxy.config.json）：

```
{
  "/api": {
    "target": "http://localhost:8000",
    "secure": false,
    "logLevel": "debug",
    "changeOrigin": true
  }
}
```

在 package.json 中使用代理：

```
{
  "scripts": {
    "ng": "ng",
    "start": "ng serve --proxy-config proxy.config.json",
    "build": "ng build",
    "test": "ng test",
    "lint": "ng lint",
    "e2e": "ng e2e"
  }
}
```

路由守卫

守卫可以用同步的方式返回一个布尔值，但在很多情况下，守卫无法用同步的方式给出答案。 守卫可能会向用户问一个问题、把更改保存到服务器，或者获取新数据，而这些都是异步操作。因此，路由的守卫可以返回一个 Observable 或 Promise，并且路由器会等待这个可观察对象被解析为 true 或 false

路由器可以支持多种守卫接口：

- 用 CanActivate 来处理导航到某路由的情况
- 用 CanActivateChild 来处理导航到某子路由的情况
- 用 CanDeactivate 来处理从当前路由离开的情况
- 用 Resolve 在路由激活之前获取路由数据
- 用 CanLoad 来处理异步导航到某特性模块的情况

```
import { Injectable } from '@angular/core';
import {
  CanActivate,
  Router,
```

```
  ActivatedRouteSnapshot,
  RouterStateSnapshot,
  CanActivateChild,
  UrlTree,
} from '@angular/router';
import { AuthService } from './auth.service';

@Injectable({
  providedIn: 'root',
})
export class AuthGuard implements CanActivate, CanActivateChild {
  constructor(private authService: AuthService, private router: Router) {}

  canActivate(
    route: ActivatedRouteSnapshot,
    state: RouterStateSnapshot
  ): true | UrlTree {
    let url: string = state.url;

    return this.checkLogin(url);
  }

  canActivateChild(
    route: ActivatedRouteSnapshot,
    state: RouterStateSnapshot
  ): true | UrlTree {
    return this.canActivate(route, state);
  }

  /* . . . */
}
```

```
const adminRoutes: Routes = [
  {
    path: 'admin',
    component: AdminComponent,
    canActivate: [AuthGuard],
    children: [
      {
        path: '',
        canActivateChild: [AuthGuard],
        children: [
          { path: 'crises', component: ManageCrisesComponent },
          { path: 'heroes', component: ManageHeroesComponent },
          { path: '', component: AdminDashboardComponent },
        ],
      },
    ],
  },
];

@NgModule({
  imports: [RouterModule.forChild(adminRoutes)],
  exports: [RouterModule],
})
export class AdminRoutingModule {}
```