link: null
title: 珠峰架构师成长计划
description: src\Subject.js
keywords: null
author: null
date: null
publisher: 珠峰架构师成长计划
stats: paragraph=124 sentences=293, words=2803

1. RxJS

- RxJS是一个响应式编程库，它使用可观察序列来组成异步和基于事件的程序
- `Observable` 是 `RxJS`中的一个对象，它可以被观察，并且可以在多次订阅之间共享
- `Observer` 是RxJS中的一个对象，它订阅 `Observable`并处理 `Observable`发布的值
- `Subscription` 表示 `Observer`订阅 `Observable`的关系
- `Operators` 是RxJS中的函数，用于在 `Observable`序列上执行各种转换和过滤操作
- `Subject` RxJS中的一个对象，它既是Observable，又是Observer。你可以使用Subject来创建一个Observable，并使用它的next，error和complete方法来发布值。 你也可以订阅Subject，并处理发布的值。
- `Schedulers` 是RxJS中的一组函数，用于控制Observable的执行
- `Observer(&#x89C2;&#x5BDF;&#x8005;)` 是由可观察对象传递的值的消费者。观察者仅仅是一组回调，每种类型的通知由可观察对象传递： `next`， `error` 和 `complete`
- 要使用 `Observer(&#x89C2;&#x5BDF;&#x8005;)`，请将其提供给可观察对象的 `subscribe`
- 观察者只是带有三个回调的对象，每种类型的通知都有一个回调，可观察对象可能传递这些通知
- `RxJS`中的观察者也可能是部分可选的的。如果不提供其中一个回调，可观察对象的执行仍然会正常进行，但是某些类型的通知将被忽略，因为观察者中没有相应的回调
- 在订阅可观察对象时，您也可以将 `next`回调作为参数提供，而不必附加到观察者对象上,在 `observable.subscribe` 内部，它将使用回调参数作为 `next`处理程序创建观察者对象
- `&#x8C03;&#x7528;`或 `&#x8BA2;&#x9605;`是一个隔离的操作：两次函数调用会触发两个单独的副作用，两次可观察对象订阅会触发两个单独的副作用。与 `&#x4E8B;&#x4EF6;&#x53D1;&#x5C04;&#x5668;&#xFF08;EventEmitters&#xFF09;`不同，事件发射器共享副作用并且无论是否存在订阅者都有急切执行，而可观察对象没有共享的执行并且是懒惰的
- `Observables` 可以使用 `new Observable` 或创建操作符创建，使用观察者订阅，执行以向观察者发送 `next` / `error` / `complete` 通知，并且可以对其执行进行处理
- Observable 的核心关注点
    - 创建 `Observables`
    - 订阅 `Observables`
    - 执行 `Observables`
    - 处理 `Observables`

```
import { Observable } from 'rxjs'
const observable = new Observable(subscriber => {
  subscriber.next(1)
  subscriber.next(2)
  subscriber.next(3)
  subscriber.complete()
})
observable.subscribe({
  next: value => console.log('next value:', value),
  complete: () => {
    console.log('complete')
  }
})
observable.subscribe(value => console.log('next value:', value))
```

- Subject 是 Observable 的一种特殊类型，它允许将值多播给许多观察者。Subject 就像 EventEmitter 每个 Subject 都是一个 Observable 和一个 Observer。您可以订阅 Subject，并且还可以调用 next 来提供值，以及 error 和 complete
- 简单来说，Subject 是一种特殊的 Observable，它既可以订阅数据流，也可以向数据流中提交数据。Subject 还具有 Observer 的特性，即可以调用 next、error 和 complete 方法

src\Subject.js

```
import { Subject } from 'rxjs';
const source = new Subject();
source.subscribe({ next: data => console.log(`Subject 第一次订阅: ${data}`) });
source.next(1);
source.next(2);
source.subscribe({ next: data => console.log(`Subject 第二次订阅: ${data}`) });
source.next(3);
source.next(4);
```

```
const express = require('express');
const cors = require('cors');
const morgan = require('morgan');
const bodyParser = require('body-parser');
const app = express();
app.use(morgan('dev'));

app.use(cors(
    {
        origin: 'http://localhost:3000',
        credentials: true
    }
));
app.use(bodyParser.json());
const users = [];
app.get('/api/user/1', (req, res) => {
     setTimeout(() => {
        res.json({ name: '张三' });
    }, 3000);
});
app.get('/api/search', (req, res) => {
    const q = req.query.q;
    const data = [];
    for (let i = 1; i 10; i++) {
        data.push(q + i);
    }
    res.json(data);
});
app.post('/api/user', (req, res) => {
   const user = req.body;
   user.id = Date.now();
   users.push(user);
   res.json(user)
});
app.delete('/api/user/1', (req, res) => {
   res.status(500).json({message:'删除失败'});
});
app.listen(8080, () => {
    console.log('server start at 8080');
});
```

- interval 是 RxJS 中的一个静态操作符，它会创建一个发出连续整数的 Observable，并按照指定的时间间隔发出。

```
import { interval } from 'rxjs';
const timer = interval(1000);
timer.subscribe(num => {
  console.log(num);
});
```

- bufferTime是 RxJS中的一个静态操作符，它会按照指定的时间间隔将 Observable中的值缓存在数组中，然后将这些数组作为单独的值发出

```
import { interval } from 'rxjs';
import { bufferTime } from 'rxjs/operators';
const timer = interval(500);
const bufferedTimer = timer.pipe(bufferTime(1000));
bufferedTimer.subscribe(arr => {
  console.log(arr);
});
```

- bufferCount 是 RxJS 中的一个静态操作符，它会将 Observable 中的值按照指定的数量缓存在数组中，然后将这些数组作为单独的值发出

```
import { interval } from 'rxjs';
import { bufferCount } from 'rxjs/operators';
const timer = interval(500);
const bufferedTimer = timer.pipe(bufferCount(3));
bufferedTimer.subscribe(arr => {
  console.log(arr);
});
```

- map 操作符是 RxJS 中的一个常用操作符，它可以用来对数据流中的每一项数据进行转换

```
import { of } from 'rxjs';
import { map } from 'rxjs/operators';

const source = of(1, 2, 3);
const example = source.pipe(
  map(val => val * 2)
);
const subscribe = example.subscribe(val => console.log(val));
```

- switchMap 是 RxJS 中的一个操作符，它通常用于将一个 Observable 的输出映射到另一个 Observable，并将新的 Observable的输出发送到输出流中
- switchMap 的行为非常类似于 map 操作符，但有一个重要的区别：它会取消订阅之前的 Observable，并订阅最新的 Observable。这意味着，如果有多个 Observable 输出，只会发出最新的 Observable 的输出

```
import { interval, switchMap, from, take } from 'rxjs';

const source$ = interval(1000)

.pipe(take(3));
const switch$ = source$.pipe(

    switchMap(n => from(new Promise(resolve => {
        setTimeout(() => resolve(n), 2000)
    })))
);
switch$.subscribe(n => console.log(n));
```

- mergeMap是一种内置的操作符，它可以将源 Observable的每个值映射到一个新的 Observable中，并将它们合并到一个单独的输出 Observable中。这个操作符可以用来执行多个异步操作，并将它们的结果合并到一起
- map操作符会把一个数据流转换成另一个数据流，但是它的转换函数必须是同步的，并且只能返回一个值
- mergeMap操作符也可以把一个数据流转换成另一个数据流，但是它的转换函数可以是异步的，并且可以返回多个值。它会把这些值合并成一个数据流输出

```
source$.pipe(
    mergeMap(project: function(value: T, index: number): ObservableInput, concurrent: number): Observable)
```

- source$是源Observable

- project函数接受源Observable的每个值和索引，并返回一个Observable
- concurrent参数是可选的，用于指定最多有多少个内部Observable可以并发执行

src\mergeMap.js

```
import { interval, of, mergeMap } from 'rxjs';

const source$ = interval(1000);
const merged$ = source$.pipe(

    mergeMap(n => of(n * 2))
);
merged$.subscribe(n => console.log(n));
```

- map 操作符是一个变换操作符，它会将源 Observable 中的每个值映射到一个新的值，然后将这些新值发出。
- switchMap 和 mergeMap 也是变换操作符，但它们不仅会将源 Observable 中的值映射到新的值，还会将这些值映射到新的 Observable
- switchMap 会取消之前的 Observable，并只发出最新的 Observable 中的值
- mergeMap 会同时发出所有 Observable 中的值
- takeUntil是RxJS中的一个变换操作符，它会取消订阅源 Observable，并停止发出值，直到另一个 Observable 发出值

```
import { interval } from 'rxjs';
import { takeUntil } from 'rxjs/operators';

const source$ = interval(1000);
const stop$ = new Subject();

const result$ = source$.pipe(
  takeUntil(stop$)
);

result$.subscribe(x => console.log(x));

stop$.next();
```

- withLatestFrom 是 RxJS 中的一个变换操作符，它会在源 Observable 发出值时，取最新的值从另一个 Observable 中发出

```
first$:  -----0-----1-----2-----3-----4-----5|
second$: ----------------0----------------1|
result$: ----------------[2,0]--[3,0]-[4,0]-[5,1]|
```

withLatestFrom.js

```
import {interval } from 'rxjs';
import { withLatestFrom } from 'rxjs/operators';
const first$ = interval(1000);
const second$ = interval(3000);
const result$ = first$.pipe(
  withLatestFrom(second$)
);
result$.subscribe(([first, second]) => console.log(first, second));
```

- debounceTime 操作符会在指定的时间内忽略掉源 Observable 发出的新的值，如果在这段时间内源 Observable 再次发出值，则会重新计算这段时间。

debounceTime.js

```
import { debounceTime } from 'rxjs/operators';

input$.pipe(debounceTime(500)).subscribe(val => {

  search(val);
});
```

- debounce 和 debounceTime 操作符类似，但是它允许你使用自定义函数来决定忽略掉源 Observable 发出的值的时间

```
import { debounce, timer } from 'rxjs';
input$.pipe(debounce(() => timer(500))).subscribe(val => {

  search(val);
});
```

- lastValueFrom 是一种 RxJS 操作符，它是用于 Observable 序列的。它会返回一个新的 Observable，该 Observable 在源 Observable 完成（即不再发出任何项）时发出源 Observable 的最后一个值。
- 如果源 Observable 从未完成，则 lastValue Observable 将永远不会发出任何值。因此，通常需要与其他操作符（例如 take 或 takeUntil）结合使用，以便在源 Observable 完成之前终止观察

```
import { lastValueFrom,interval,take } from 'rxjs';
const source = interval(1000);
const lastValue = lastValueFrom(source.pipe(
    take(5),
));
lastValue.then(console.log);
```

- RxJS中的share方法是一种可以让多个观察者订阅同一个Observable的方法。它通常用于避免在多个观察者之间重复执行相同的数据请求
- 假设你有一个Observable，它执行一个HTTP请求以获取数据。如果你在不同的组件中订阅了这个Observable，它就会执行多次HTTP请求，这样可能会导致性能问题
- 这时，你就可以使用share方法来共享这个Observable，以避免多次执行相同的HTTP请求。具体来说，share方法会将Observable转换成一个ConnectableObservable，它可以让多个观察者订阅同一个Observable，但是实际上只会执行一次数据请求

```
import { share } from 'rxjs/operators';
import { fromFetch } from 'rxjs/fetch';
const sharedObservable = fromFetch('http://localhost:8080/api/user/1')
.pipe(share())
sharedObservable.subscribe(res=>res.json().then(res=>console.log(res)));
sharedObservable.subscribe(res=>res.json().then(res=>console.log(res)));
```

- fromFetch 操作符允许你从给定的 URL 中获取资源，并将获取到的资源作为一个 Observable 发出

```
fromFetch('http://localhost:8080/api/user/1')
  .pipe(
    switchMap(response => {
      if (response.ok) {
        return response.json();
      } else {
        throw new Error('Api request failed');
      }
    }),
  )
  .subscribe({
    next: response => console.log(response),
    error: error => console.error(error),
  });
```

- merge 操作符允许你将多个 Observable 合并成一个 Observable，并将这些 Observable 中的值按时间顺序依次发出

```
const first = of(1, 2, 3);
const second = of(4, 5, 6);
merge(first, second).subscribe(value => console.log(value));
```

- catchError和 throwError都是用来处理 Observable中发生的错误的
- catchError操作符允许你捕获一个 Observable中发生的错误，并返回一个新的 Observable来取代原来的 Observable
- catchError 的方法是将它作为 Observable 的链式调用的一部分，并传入一个回调函数作为参数。回调函数接收一个错误对象作为参数，并返回一个新的 Observable。这个新的 Observable 将会取代原来的 Observable，并继续执行后续的操作。
- throwError 操作符则是用来显式地抛出一个错误的。它返回一个不包含任何值的 Observable，并立即终止。通常，你可能会使用 throwError 来表示一个不可恢复的错误，例如网络连接

catchError

```
import { Observable ,of} from 'rxjs';
import { catchError } from 'rxjs/operators';
const source$ = new Observable(subscriber=>{
    setTimeout(()=>{
        subscriber.error(new Error('发生了错误'));
    },1000);
});
source$.pipe(
    catchError(error => of('正常值')),
).subscribe({
    next: value => console.log('next',value),
    error: error => console.error('error',error),
    complete: () => console.log('complete'),
});
```

```
import { Observable ,of, throwError} from 'rxjs';
import { catchError } from 'rxjs/operators';
const source$ = new Observable(subscriber=>{
    subscriber.error({success:false});
});
source$.pipe(
    catchError(error => {

        return throwError(()=>error);
    }),
).subscribe({
    next: value => console.log('next',value),
    error: error => console.error('error',error),
    complete: () => console.log('complete'),
});
```

- filter 操作符允许你只发出源 Observable 中满足特定条件的值

```
import { of } from 'rxjs';
of(1, 2, 3, 4, 5)
  .pipe(
    filter(value => value % 2 === 0),
  )
  .subscribe(value => console.log(value));
```

- throwIfEmpty操作符用于在源 Observable 完成后，如果没有发出任何值，就抛出一个错误

```
import { Observable, throwIfEmpty } from 'rxjs';
const source$ = new Observable(subscriber => {
    subscriber.next(1);
    subscriber.complete();
});
source$
    .pipe(throwIfEmpty())
    .subscribe({
        next: user => console.log(user),
        error: error => console.error(error),
    })
```

2.缓存

src\bufferTime.js

```
import { interval, bufferTime } from 'rxjs';

const messageBox = document.getElementById('messageBox');
const source$ = interval(1000);
source$.pipe(bufferTime(2000))
    .subscribe((messages) => {
        messageBox.innerHTML += messages.map(item => `Message ${item}`)
            .join('\n')
    })
```

src\bufferCount.js

```
import { interval, bufferCount } from 'rxjs';
const messageBox = document.getElementById('messageBox');
const source$ = interval(1000);

source$.pipe(bufferCount(3))
    .subscribe((messages) => {
        messageBox.innerHTML += messages.map(item => `Message ${item}`)
            .join('\n')
    })
```

3.拖拽

public\index.html

```
<html lang="en">
<head>
  <meta charset="utf-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1" />
  <meta name="theme-color" content="#000000" />
  <meta name="description" content="Web site created using create-react-app" />
  <title>React Apptitle>
  <style>
    #draggable {
      width: 100px;
      height: 100px;
      background-color: red;
      position: absolute;
      top: 0;
      left: 0;
    }
  style>
head>
<body>
  <ul id="messageBox">ul>
  <div id="draggable">div>
body>
html>
```

```
import { fromEvent } from 'rxjs';
import { withLatestFrom, takeUntil,switchMap} from 'rxjs/operators';
function startDragging(element) {
    const mouseDown = fromEvent(element, 'mousedown')
    const mouseUp = fromEvent(document, 'mouseup')
    const mouseMove = fromEvent(document, 'mousemove')
    mouseDown.pipe(
        switchMap(() => mouseMove.pipe(takeUntil(mouseUp))),
        withLatestFrom(mouseDown, (moveEvent, downEvent) => {
            return {
                left: moveEvent.clientX - downEvent.offsetX,
                top: moveEvent.clientY - downEvent.offsetY
            }
        })
    ).subscribe(({left,top}) => {
        element.style.left = left + 'px';
        element.style.top = top + 'px';
    })
}

const draggable = document.getElementById('draggable');
startDragging(draggable);
```

4.并发请求

src\multiRequest.js

```
import { mergeMap, from } from 'rxjs';
/**
  实现一个批量并发请求函数 request(urls, concurrent)，要求如下：
  1.要求最大并发数 concurrent
  2.每当有一个请求返回，就进行新的请求
  3.所有请求完成后，结果按照urls里面的顺序依次打出
 */
function fetchData(url) {
  return new Promise(resolve => setTimeout(() => resolve(url), 3000));
}
const urls = [
  '/api/user/1',
  '/api/user/2',
  '/api/user/3'
];
const start = Date.now();
function request(urls, concurrent) {
  from(urls)
    .pipe(mergeMap(fetchData, concurrent))
    .subscribe(val => {
      console.log(`耗时: ${parseInt((Date.now() - start) / 1000)}s`);
      console.log(val)
    });
}
request(urls, 2)
```

5.竞态

- 前端请求后端接口时的竞态问题是指，当前端同时发起多个请求时，可能会导致意料之外的结果
- 例如，假设你正在使用 JavaScript 的 fetch 函数发起多个请求，如果在前一个请求完成之前就发起了新的请求，则会发生竞态问题。这是因为，在前一个请求完成之前，后端接口可能会收到并处理多个请求，导致数据更新的顺序不确定

```
import { Subject, switchMap } from 'rxjs';
function fetchData(id) {
    return new Promise(resolve => setTimeout(() => resolve(id), 1000 * id));
}
const search = new Subject()
search.pipe(switchMap(fetchData))
    .subscribe(console.log);
search.next(3);
setTimeout(() => {
    search.next(1);
}, 1000);
```

6.suggests

```
import { fromEvent, of, timer } from 'rxjs';
import { debounce, debounceTime, switchMap } from 'rxjs/operators';

const inputElement = document.querySelector('#keyword');
const wordsElement = document.querySelector('#words');

const input$ = fromEvent(inputElement, 'input');
input$.subscribe(event => { console.log(event) });

const search$ = input$.pipe(
    debounceTime(100),

    debounce((event) => event.target.value.length > 3 ? of(event) : timer(3000)),
    switchMap(event => fetch(`http://localhost:8080/api/search?q=${event.target.value}`))
);

search$.subscribe(response => {
    response.json().then(data => {

        wordsElement.innerHTML = data.map(item => `${item}`).join('');
    });
});
```

7.fetch封装

src\index.js

```
import {http} from './fetch/http';
http.request({
    url: 'http://localhost:8080/api/user/1',
    method: 'GET'
}).then(response => {
    console.log(response)
})
```

src\fetch\http.js

```
import { lastValueFrom, share } from 'rxjs';
import { fromFetch } from 'rxjs/fetch';
export class Http {
  request(options) {
    return  lastValueFrom(
      fromFetch(options.url, options)
      .pipe(share())
    )
  }
}
export const http = new Http();
```

src\index.js

```
import { http } from './fetch/http';
http.post('http://localhost:8080/api/user', { name: 'zhangsan' })
    .then(res => res.json())
    .then(response => {
        console.log(response)
    })
```

src\fetch\http.js

```
import { lastValueFrom, share } from 'rxjs';
import { fromFetch } from 'rxjs/fetch';
import { getUrlFromOptions, getInitFromOptions } from './utils';
export class Http {
  request(options) {
    const url = getUrlFromOptions(options);
    const init = getInitFromOptions(options);
    return  lastValueFrom(
      fromFetch(url, init)
      .pipe(share())
    )
  }
+ delete(url) {
+   return this.request({ method: 'DELETE', url });
+ }
+ put(url, data) {
+   return this.request({ method: 'PUT', url, data, headers: { "Content-Type": "application/json" } });
+ }
+ post(url, data) {
+   return this.request({ method: 'POST', url, data, headers: { "Content-Type": "application/json" } });
+ }
}
export const http = new Http();
```

src\fetch\utils.js

```
export function getUrlFromOptions(options) {
    let { url, params={}, method='GET' } = options;
    let queryString = "";
    params = Object.keys(params).reduce((filteredParams, key) => {
        if (params[key] !== "") {
            filteredParams[key] = params[key];
        }
        return filteredParams;
    }, {});
    queryString = Object.keys(params).map(key => {
        return encodeURIComponent(key) + "=" + encodeURIComponent(params[key]);
    }).join("&");
    if (method === "GET" || method === "DELETE") {
        url += "?" + queryString;
    }
    return url;
}

export function getInitFromOptions(options) {
    let method = options.method || 'GET';
    let headers = options.headers || {};
    let isJSONBody = headers['Content-Type'] === 'application/json';
    let body = options.data;
    if(body&& Object.keys(body).length>0){
        if (isJSONBody) {
            body = JSON.stringify(body);
        } else {
            body = new URLSearchParams(body);
        }
    }
    let credentials = options.credentials || 'omit';
    return { method, headers, body, credentials };
}
```

src\index.js

```
import { http } from './fetch/http';
http.post('http://localhost:8080/api/user', { name: 'zhangsan' })
    .then(response => {
        console.log(response.data)
    })
```

src\fetch\http.js

```
+import { lastValueFrom, share ,mergeMap} from 'rxjs';
import { fromFetch } from 'rxjs/fetch';
import { getUrlFromOptions, getInitFromOptions } from './utils';
export class Http {
  request(options) {
    const url = getUrlFromOptions(options);
    const init = getInitFromOptions(options);
    return  lastValueFrom(
      fromFetch(url, init)
      .pipe(
+       mergeMap(async response => {
+         if (response.ok) {
+           return {data: await response.json(),status: response.status};
+         } else {
+           return Promise.reject({data: await response.json(),status: response.status});
+         }
+       }),
        share())
    )
  }
  post(url, data) {
    return this.request({ method: 'POST', url, data, headers: { "Content-Type": "application/json" } });
  }
  delete(url) {
    return this.request({ method: 'DELETE', url });
  }
  put(url, data) {
    return this.request({ method: 'PUT', url, data, headers: { "Content-Type": "application/json" } });
  }
  get(url, params) {
    return this.request({ method: 'GET', url, params });
  }
}
export const http = new Http();
```

src\index.js

```
import { http } from './fetch/http';
+http.delete('http://localhost:8080/api/user/1', { name: 'zhangsan' })
    .then(response => {
        console.log(response.data)
+    },error=>console.error(error))
```

src\fetch\http.js

```
import { lastValueFrom, share, mergeMap, filter, merge } from 'rxjs';
import { fromFetch } from 'rxjs/fetch';
import { getUrlFromOptions, getInitFromOptions } from './utils';
export class Http {
  request(options) {
    const url = getUrlFromOptions(options);
    const init = getInitFromOptions(options);
+   const fetchStream = fromFetch(url, init).pipe(share())
+   const successStream = fetchStream.pipe(
+     filter(response => response.ok),
+     mergeMap(async response => {
+       return { data: await response.json(), status: response.status };
+     }),
+   );
+   const failureStream = fetchStream.pipe(
+     filter(response => !response.ok),
+     mergeMap(async response => {
+       return Promise.reject({ error: await response.json(), status: response.status })
+     }),
+   );
+   const mergedStream = merge(successStream, failureStream)
+   return lastValueFrom(mergedStream);
  }
  delete(url) {
    return this.request({ method: 'DELETE', url });
  }
  put(url, data) {
    return this.request({ method: 'PUT', url, data, headers: { "Content-Type": "application/json" } });
  }
  post(url, data) {
    return this.request({ method: 'POST', url, data, headers: { "Content-Type": "application/json" } });
  }
}
export const http = new Http();
```

src\index.js

```
import { http } from './fetch/http';
http.delete('http://localhost:8080/api/user/1')
    .then(response => {
        console.log(response.data)
    },error=>console.error(error))
```

src\fetch\http.js

```
+import { lastValueFrom, share, mergeMap, filter, merge,catchError,throwError,takeUntil,throwIfEmpty,Subject } from 'rxjs';
import { fromFetch } from 'rxjs/fetch';
import { getUrlFromOptions, getInitFromOptions } from './utils';
export class Http {
+ cancleRequests = new Subject();
+ cancel(requestId) {
+   this.cancleRequests.next(requestId);
+ }
  request(options) {
    const url = getUrlFromOptions(options);
    const init = getInitFromOptions(options);
    const fetchStream = fromFetch(url, init).pipe(share())
    const successStream = fetchStream.pipe(
      filter(response => response.ok),
      mergeMap(async response => {
        return { data: await response.json(), status: response.status };
      }),
    );
    const failureStream = fetchStream.pipe(
      filter(response => !response.ok),
      mergeMap(async response => {
        return Promise.reject({ error: await response.json(), status: response.status })
      }),
    );
+   const mergedStream = merge(successStream, failureStream).pipe(
+     catchError(error => throwError(()=>({...error,url}))),
+     takeUntil(
+       this.cancleRequests.pipe(
+         filter(requestId => options.requestId === requestId))
+     )
+     ,throwIfEmpty(() => ({
+       type: 'cancel',
+       cancelled: true,
+       data: null,
+       status: -1,
+       statusText: '请求被取消',
+       config: options
+     }))
+   );
+   return lastValueFrom(mergedStream);
  }
  delete(url) {
    return this.request({ method: 'DELETE', url });
  }
  put(url, data) {
    return this.request({ method: 'PUT', url, data, headers: { "Content-Type": "application/json" } });
  }
  post(url, data) {
    return this.request({ method: 'POST', url, data, headers: { "Content-Type": "application/json" } });
  }
  get(url, params) {
    return this.request({ method: 'GET', url, params });
  }
}
export const http = new Http();
```