link: null
title: 珠峰架构师成长计划
description: null
keywords: null
author: null
date: null
publisher: 珠峰架构师成长计划
stats: paragraph=197 sentences=462, words=2972

1.RxJS

1.1 RxJS介绍

- RxJS 是一个使用可观察序列组合异步和基于事件的程序的库
- 它提供了一种核心类型，即 Observable，以及卫星类型（ Observer， Schedulers， Subjects）和操作符，这些操作符受到 Array 方法（map，filter，reduce，every 等）的启发，可以将异步事件处理为集合
- ReactiveX 结合了观察者模式和迭代器模式，并将函数式编程与集合相结合，以满足对理想管理事件序列的需求

1.2 基本概念

- Observable 表示可调用的未来值或事件的集合的想法
- Observer 是一组回调，知道如何监听 Observable 传递的值
- Subscription 表示 Observable 的执行，主要用于取消执行
- Operators 是纯函数，使用操作（如 map&#x3001;filter、 concat、 reduce 等）处理集合时具有函数式编程风格
- Subject 等同于 EventEmitter，是将值或事件多播到多个 Observer 的唯一方法
- Schedulers 是集中式调度程序，用于控制并发，允许我们协调计算发生在例如 setTimeout、requestAnimationFrame 或其他位置的时间

1.3 参考链接

- 官方文档 (https://rxjs.dev/)
- 入门指南 (https://rxjs.dev/guide/overview)
- 例子 (https://rxjs.dev/examples)
- 常见问题 (https://rxjs.dev/faq)
- 中文文档 (https://cn.rx.js.org/)
- github源码 (https://github.com/ReactiveX/rxjs)
- rxjs弹珠图 (https://rxmarbles.com/)
- rxjs可视化 (https://rxviz.com/)
- explorer (https://reactive.how/rxjs/explorer)

2.Observable

- Observables(&#x53EF;&#x89C2;&#x5BDF;&#x5BF9;&#x8C61;)是懒惰的多个值的 Push 集合,可观察对象是一种异步数据流，它可以在将来推送多个值。它们是 &#x60F0;&#x6027;的，因为它们不会立即开始发送值，直到有人订阅它们

2.1 Pull和Push

- Pull 和 Push 是两种不同的协议，用于描述数据生产者如何与数据消费者进行通信
- 在 Pull 系统中，消费者决定何时从数据生产者接收数据。生产者本身并不知道何时将数据传递给消费者
- 每个 JavaScript函数都是 Pull系统。函数是数据的生产者，而调用函数的代码通过 &#x62C9;&#x51FA;单个返回值从其调用中消费它

```
function* generator() {
  yield 1;
  yield 2;
  yield 3;
}

const iterator = generator();
console.log(iterator.next().value);
console.log(iterator.next().value);
console.log(iterator.next().value);
```

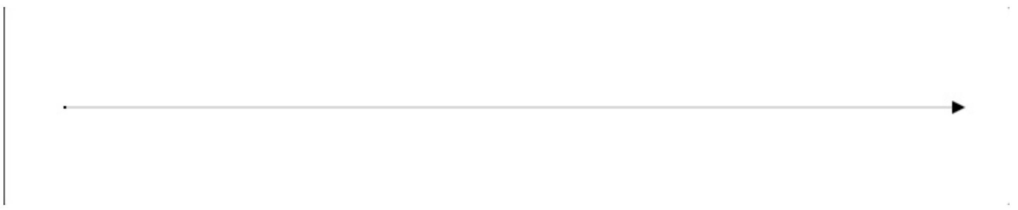
- 在 Push系统中，生产者决定何时将数据发送给消费者。消费者不知道何时会收到该数据
- Promise是 JavaScript 中最常见的推送系统。 Promise向已注册的回调函数（Consumers）提供已解析的值
- RxJS引入了可观察对象，这是一种新的 JavaScript 推送系统。可观察对象是多个值的生产者，将它们 &#x63A8;&#x9001;到 &#x89C2;&#x5BDF;&#x8005;&#xFF08;Consumers&#xFF09;

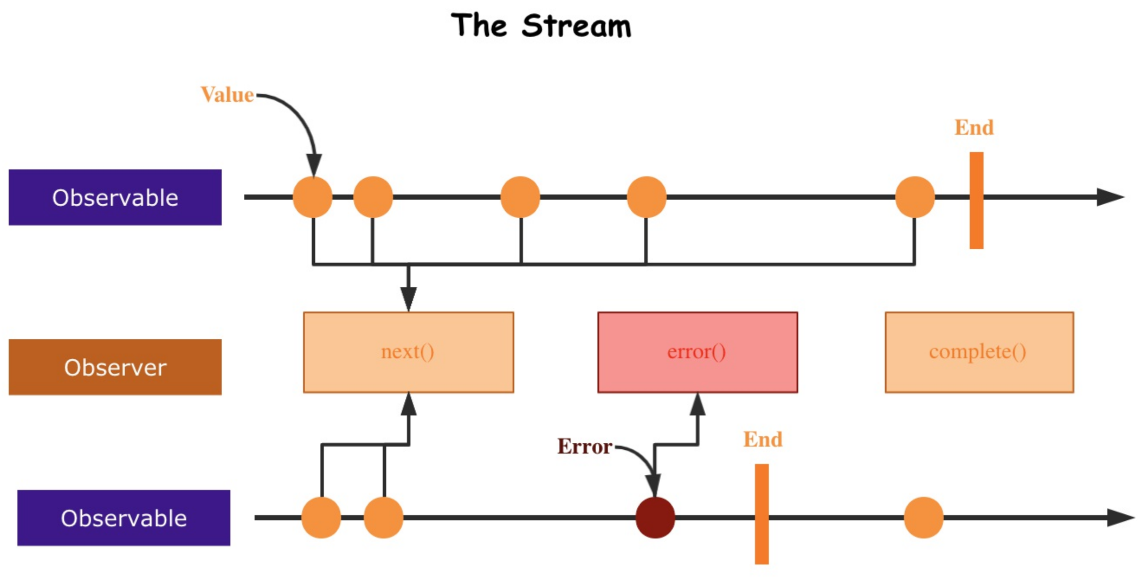
```
button.addEventListener('click', function() {
  console.log('Button was clicked!');
});
```

生产者 消费者 Pull 被动：在请求时生成数据 主动：决定何时请求数据 Push 主动：以自己的速度生成数据 被动：对收到的数据做出反应

2.2 Stream(流)

- 流是随着时间变化的值序列





2.3 Observable

- 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

2.4 src\index.js

src\index.js

```
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))
```

2.5 rxjs\index.js

src\rxjs\index.js

```
export { Observable } from './internal/Observable';
```

2.6 Observable.js

src\rxjs\internal\Observable.js

```

import { Subscriber } from './Subscriber';

export class Observable {
    constructor(subscribe) {
        if (subscribe) {
            this._subscribe = subscribe;
        }
    }
    subscribe(observerOrNext) {
        const subscriber = new Subscriber(observerOrNext);
        this._subscribe(subscriber)
        return subscriber;
    }
}
```

src\rxjs\internal\Subscriber.js

```
import { isFunction } from './util/isFunction';
export class Subscriber {
    isStopped = false;
    constructor(observerOrNext) {
        let observer;
        if (isFunction(observerOrNext)) {
            observer = {
                next: observerOrNext
            };
        } else {
            observer = observerOrNext;
        }
        this.destination = observer;
    }
    next(value) {
        if (!this.isStopped) {
            this.destination.next(value);
        }
    }
    complete() {
        if (!this.isStopped) {
            this.isStopped = true;
            this.destination.complete?.();
        }
    }
}
```

2.8 isFunction.js

src\rxjs\internal\util\isFunction.js

```
export function isFunction(value) {
  return typeof value === 'function';
}
```

3. of

- RxJS 的 `of` 操作符允许你创建一个 `Observable`，它发出一组项目，然后完成
- 你可以使用它来将任何值发送到一个 `Observable` 中，例如，你可以使用它来将一个数字数组转换为 `Observable`
- `of` 操作符是同步的，意味着它会立即发出所有的值，并立即完成,如果你需要异步发出值，你可以使用 from 操作符
- RxJS 的 `from` 操作符允许你将多种不同的数据类型转换为 Observable，包括数组、类数组对象(如arguments对象)、迭代器和可观察对象
- `from` 操作符是异步的，意味着它会在内部使用内置的调度

3.1 src\index.js

src\index.js

```
import { of, from } from './rxjs'
const arrayLike = of(1, 2, 3)
arrayLike.subscribe({
  next: value => console.log(`arrayLike:`, value),
  complete: () => console.log('arrayLike done'),
});

const promiseLike = from(Promise.resolve(4))

promiseLike.subscribe({
  next: value => console.log(`promiseLike:`, value),
  complete: () => console.log('promiseLike done'),
});
```

3.2 rxjs\index.js

src\rxjs\index.js

```
export { Observable } from './internal/Observable';
+export { of } from './internal/observable/of';
+export { from } from './internal/observable/from';
```

2.3 of.js

src\rxjs\internal\observable\of.js

```
import { from } from './from';
export function of(...args) {
  return from(args);
}
```

3.4 from.js

src\rxjs\internal\observable\from.js

```
import { innerFrom } from './innerFrom';
export function from(input) {
  return innerFrom(input)
}
```

3.5 innerFrom.js

src\rxjs\internal\observable\innerFrom.js

```
import { isArrayLike } from '../util/isArrayLike';
import { isPromise } from '../util/isPromise';
import { Observable } from '../Observable';
export function innerFrom(input) {
  if (input instanceof Observable) {
    return input;
  }
  if (input != null) {
    if (isArrayLike(input)) {
      return fromArrayLike(input);
    }
    if (isPromise(input)) {
      return fromPromise(input);
    }
  }
}
export function fromArrayLike(array) {
  return new Observable(subscriber => {
    for (let i = 0; i < array.length; i++) {
      subscriber.next(array[i]);
    }
    subscriber.complete();
  });
}
export function fromPromise(promise) {
  return new Observable(subscriber => {
    promise.then(value => {
      subscriber.next(value);
      subscriber.complete();
    });
  });
}
```

3.6 isArrayLike.js

src\rxjs\internal\util\isArrayLike.js

```
export const isArrayLike = x => x && typeof x.length === 'number' && typeof x !== 'function';
```

3.7 isPromise.js

src\rxjs\internal\util\isPromise.js

```
import { isFunction } from "./isFunction";
export function isPromise(value) {
  return isFunction(value?.then);
}
```

4. fromEvent

- RxJS 的 fromEvent 函数允许你将浏览器事件转换为 Observable。它接受两个参数：
  - 第一个参数是事件目标，例如 DOM 元素或 window对象
  - 第二个参数是事件名称，例如 click或 scroll

4.1 src\index.js

src\index.js

```
import { fromEvent } from './rxjs'
const source = fromEvent(document, 'click');
const subscriber = source.subscribe(console.log)
setTimeout(() => {
  subscriber.unsubscribe();
}, 1000)
```

4.2 rxjs\index.js

src\rxjs\index.js

```
export { Observable } from './internal/Observable';
export { of } from './internal/observable/of';
export { from } from './internal/observable/from';
+export { fromEvent } from './internal/observable/fromEvent';
```

4.3 fromEvent.js

src\rxjs\internal\observable\fromEvent.js

```
import { Observable } from '../Observable';
export function fromEvent(target, eventName) {
    return new Observable(subscriber => {
        const handler = (...args) => subscriber.next(...args);
        target.addEventListener(eventName, handler);
        return () => target.removeEventListener(eventName, handler);
    });
}
```

src\rxjs\internal\Subscriber.js

```
import { isFunction } from './util/isFunction';
+import { Subscription } from './Subscription';
+export class Subscriber extends Subscription {
    isStopped = false;
    constructor(observerOrNext) {
        super();
        let observer;
        if (isFunction(observerOrNext)) {
            observer = {
                next: observerOrNext
            };
        } else {
            observer = observerOrNext;
        }
        this.destination = observer;
    }
    next(value) {
        if (!this.isStopped) {
            this.destination.next(value);
        }
    }
    complete() {
        if (!this.isStopped) {
            this.isStopped = true;
            this.destination.complete?.();
        }
    }
}
```

4.5 Subscription.js

src\rxjs\internal\Subscription.js

```
export class Subscription {
    _finalizers = []
    unsubscribe() {
        const { _finalizers } = this;
        if (_finalizers) {
            for (const finalizer of _finalizers) {
                finalizer();
            }
        }
    }
    add(teardown) {
        this._finalizers.push(teardown);
    }
}
```

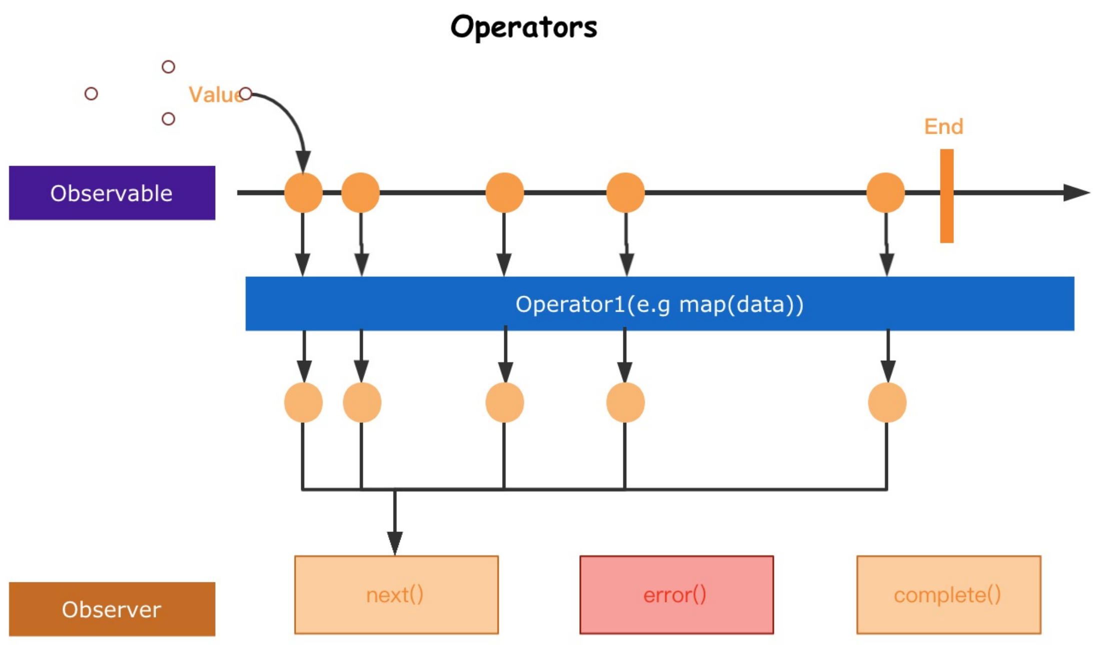
4.6 Observable.js

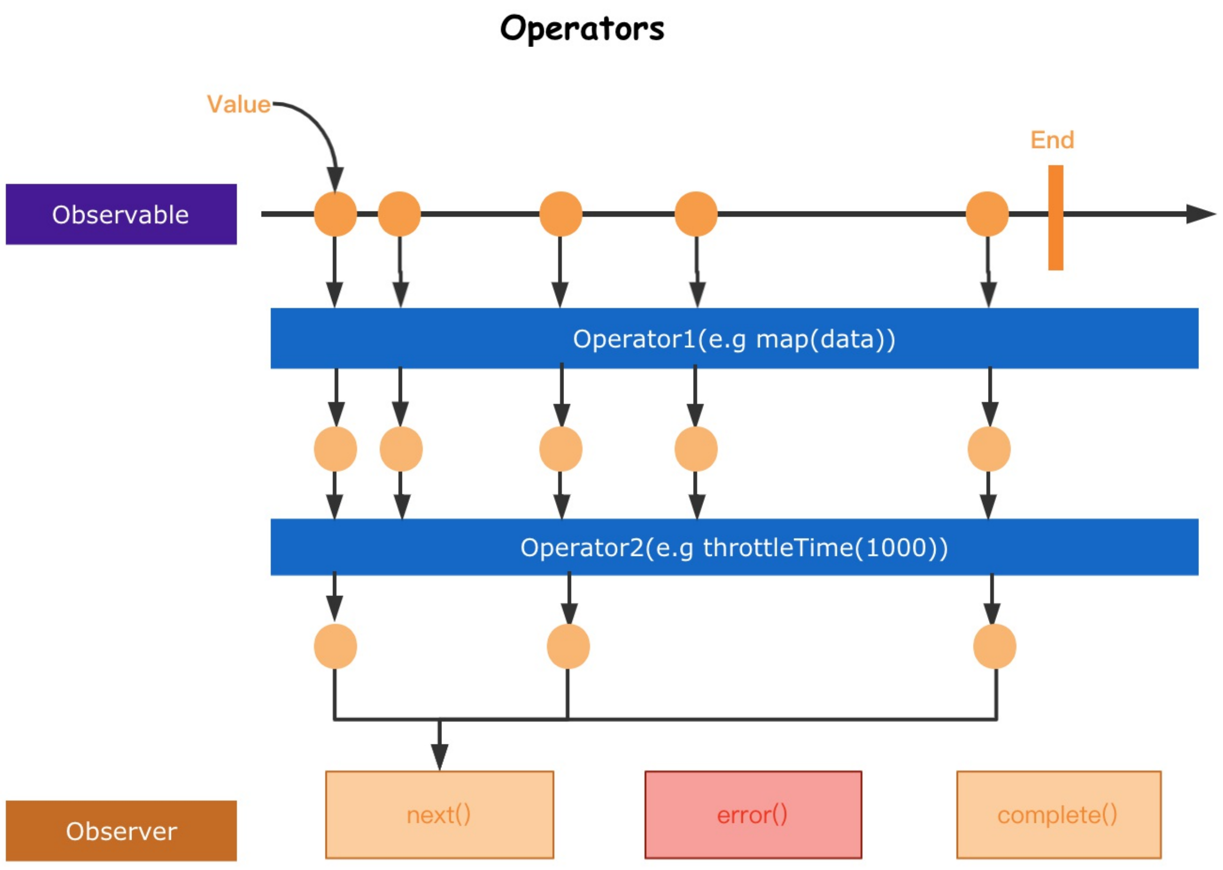
src\rxjs\internal\Observable.js

```
import { Subscriber } from './Subscriber';
export class Observable {
    constructor(subscribe) {
        if (subscribe) {
            this._subscribe = subscribe;
        }
    }
    subscribe(observerOrNext) {
        const subscriber = new Subscriber(observerOrNext);
+       const teardown = this._subscribe(subscriber)
+       subscriber.add(teardown)
        return subscriber;
    }
}
```

5. map&filter

- `map`操作符允许你对 `Observable`中的每个值进行转换，并返回一个新的 `Observable`。它接受一个函数作为参数，该函数定义如何转换每个值。
- `filter`操作符允许你选择性地过滤 `Observable` 中的值。它接受一个函数作为参数，该函数定义如何过滤值





5.1 Operators

- 在Rx中 `Observable`，控制流的状态,是它的基石，但最有用的是它的 `operator`， `operator`允许复杂的异步代码以声明的方式进行轻松组合的基础单元。 `operator`主要作用是操作、组合流中的数据
- 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。订阅输出 Observable 同样会订阅输入 Observable
- 操作符类型
    - Creation Operators 创建操作符,它们用于创建新的 Observable。这些操作符可以从各种不同的数据源（如数组、对象、Promise 等）创建 Observable，并可以控制 Observable 的行为（如发出值的频率、顺序等）,像 `of`、 `from`、 `timer`、 `interval`和 `fromEvent`等
    - Transformation Operators 转换操作符是 RxJS 中的一类特殊的操作符，它们用于将输入 Observable 转换为新的输出 Observable。这些操作符可以对输入 Observable 中的值进行转换、过滤、合并等操作，以便在输出 Observable 中呈现出所需的信息。像 `map`等

  - Combination Operators 组合操作符用于将多个 `Observable` 合并成一个新的 `Observable`。这些操作符可以帮助你创建复杂的数据流，并控制它们之间的关系,像 `merge`和 `concat`等
  - Filtering Operators 过滤操作符用于过滤输入 Observable 中的值，只返回符合特定条件的值。这些操作符可以帮助你创建精确的数据流，并且非常实用。像 `filter`等
  - Multicasting Operators 多播操作符用于将单个 `Observable`共享给多个观察者（Observer）。这些操作符可以帮助你控制 Observable 的行为，并有效地利用资源。像 `share`等

- rxmarbles (https://rxmarbles.com/)

5.2 src\index.js

src\index.js

```
import { of, map, filter } from './rxjs'
const subscriber = of(1, 2, 3)
  .pipe(map(val => val * 2))
  .pipe(filter(val => val > 3))
  .pipe(map(data => data + 1))
subscriber.subscribe(console.log)
```

5.3 rxjs\index.js

src\rxjs\index.js

```
export { Observable } from './internal/Observable';
export { of } from './internal/observable/of';
export { from } from './internal/observable/from';
export { fromEvent } from './internal/observable/fromEvent';
+export { filter } from './internal/operators/filter';
+export { map } from './internal/operators/map';
```

5.4 Observable.js

src\rxjs\internal\Observable.js

```
import { Subscriber } from './Subscriber';
export class Observable {
    constructor(subscribe) {
        if (subscribe) {
            this._subscribe = subscribe;
        }
    }
    subscribe(observerOrNext) {
        const subscriber = new Subscriber(observerOrNext);
        const teardown = this._subscribe(subscriber)
        subscriber.add(teardown)
        return subscriber;
    }
+   pipe(operation) {
+       return operation(this);
+   }
}
```

5.5 map.js

src\rxjs\internal\operators\map.js

```
import { Observable } from '../Observable';
export function map(project) {
  return source => {
    const observable = new Observable(function (subscriber) {
      return source.subscribe({
        ...subscriber,
        next: value => {
          subscriber.next(project(value));
        }
      });
    });
    return observable;
  };
}
```

5.6 filter.js

src\rxjs\internal\operators\filter.js

```
import { Observable } from '../Observable';
export function filter(predicate) {
  return source => {
    const observable = new Observable(function (subscriber) {
      return source.subscribe({
        ...subscriber,
        next: value => {
          predicate(value) && subscriber.next(value)
        }
      });
    });
    return observable;
  };
}
```

6. pipe

- Observable 对象有一个名为 pipe 的方法，允许你将多个操作符链接在一起。它可以让你在单个表达式中执行复杂的数据处理流程。

6.1 src\index.js

src\index.js

```
import { of, map, filter } from './rxjs'
const subscriber = of(1, 2, 3)
    .pipe(
        map(val => val * 2),
        filter(val => val > 3),
        map(data => data + 1)
    )
subscriber.subscribe(console.log)
```

6.2 Observable.js

src\rxjs\internal\Observable.js

```
import { Subscriber } from './Subscriber';
+import { pipeFromArray } from './util/pipe';
export class Observable {
    constructor(subscribe) {
        if (subscribe) {
            this._subscribe = subscribe;
        }
    }
    subscribe(observerOrNext) {
        const subscriber = new Subscriber(observerOrNext);
        const teardown = this._subscribe(subscriber)
        subscriber.add(teardown)
        return subscriber;
    }
+   pipe(...operations) {
+       return pipeFromArray(operations)(this);
+   }
}
```

6.3 pipe.js

src\rxjs\internal\util\pipe.js

```
import { identity } from './identity';
export function pipeFromArray(fns) {
    if (fns.length === 0) {
        return identity;
    }
    if (fns.length === 1) {
        return fns[0];
    }
    return function piped(input) {
        return fns.reduce((prev, fn) => fn(prev), input);
    };
}
```

6.4 identity.js

src\rxjs\internal\util\identity.js

```
export function identity(x) {
  return x;
}
```

7.asyncScheduler

- 使用 setTimeout（task，duration）调度任务
- async 调度器通过将任务放在 JavaScript 事件循环队列中异步地调度任务。它最适用于延迟任务的执行或定期执行任务

7.1 src\index.js

src\index.js

```
import { asyncScheduler } from './rxjs'
function task(state) {
  console.log('state: ', state);
  if (state < 5) {
    this.schedule(state + 1, 1000);
  }
}
asyncScheduler.schedule(task, 1000, 0);
```

7.2 rxjs\index.js

src\rxjs\index.js

```
export { Observable } from './internal/Observable';
export { of } from './internal/observable/of';
export { from } from './internal/observable/from';
export { fromEvent } from './internal/observable/fromEvent';
export { filter } from './internal/operators/filter';
export { map } from './internal/operators/map';
+export { asyncScheduler } from './internal/scheduler/async';
```

7.3 Scheduler.js

src\rxjs\internal\Scheduler.js

```
export class Scheduler {
  constructor(schedulerActionCtor) {
    this.schedulerActionCtor = schedulerActionCtor;
  }
  schedule(work, delay = 0, state) {
    return new this.schedulerActionCtor(work).schedule(state, delay);
  }
}
```

7.4 AsyncAction.js

src\rxjs\internal\scheduler\AsyncAction.js

```
export class AsyncAction {
  pending = false;
  constructor(work) {
    this.work = work;
  }
  schedule(state, delay = 0) {
    this.state = state;
    this.delay = delay;
    if (this.id != null) {
      this.id = this.recycleAsyncId(this.id);
    }
    this.pending = true;
    this.id = this.requestAsyncId(delay);
    return this;
  }
  requestAsyncId(delay = 0) {
    return setInterval(this.execute.bind(this), delay);
  }
  execute() {
    this.pending = false;
    this.work(this.state);
    if (this.pending === false && this.id !== null) {
      this.id = this.recycleAsyncId(this.id);
    }
  }
  recycleAsyncId(id) {
    if (id !== null) {
      clearInterval(id);
    }
    return null;
  }
}
```

7.5 async.js

src\rxjs\internal\scheduler\async.js

```
import { AsyncAction } from './AsyncAction';
import { Scheduler } from '../Scheduler';
export const asyncScheduler = new Scheduler(AsyncAction);
```

8.timer

- `timer` 函数是一个工厂函数，可以创建一个发出数字的 `Observable`，每个数字增加 1。它接受两个参数：起始值和间隔时间

8.1 src\index.js

src\index.js

```
import { timer } from './rxjs'
timer(1000).subscribe(() => console.log('timer'))
```

8.2 rxjs\index.js

src\rxjs\index.js

```
export { Observable } from './internal/Observable';
export { of } from './internal/observable/of';
export { from } from './internal/observable/from';
export { fromEvent } from './internal/observable/fromEvent';
export { filter } from './internal/operators/filter';
export { map } from './internal/operators/map';
export { asyncScheduler } from './internal/scheduler/async';
+export { timer } from './internal/observable/timer';
```

8.3 timer.js

src\rxjs\internal\observable\timer.js

```
import { Observable } from '../Observable';
import { asyncScheduler } from '../scheduler/async';
export function timer(dueTime = 0, scheduler = asyncScheduler) {
  return new Observable(subscriber => {
    let n = 0;
    return scheduler.schedule(function () {
      subscriber.next(n++);
    }, dueTime);
  });
}
```

9.interval

- `interval` 函数是一个工厂函数，可以创建一个发出数字的 `Observable`，每个数字增加 1。它接受一个间隔时间参数，表示每次发送之间的时间间隔
- `interval` 函数会一直发送数字，直到你取消订阅。你可以使用 `take` 操作符限制发送的数字数量

9.1 src\index.js

src\index.js

```
import { interval } from './rxjs'
interval(1000).subscribe((v) => console.log(v))
```

9.2 rxjs\index.js

src\rxjs\index.js

```
export { Observable } from './internal/Observable';
export { of } from './internal/observable/of';
export { from } from './internal/observable/from';
export { fromEvent } from './internal/observable/fromEvent';
export { filter } from './internal/operators/filter';
export { map } from './internal/operators/map';
export { asyncScheduler } from './internal/scheduler/async';
export { timer } from './internal/observable/timer';
+export { interval } from './internal/observable/interval';
```

9.3 interval.js

src\rxjs\internal\observable\interval.js

```
import { asyncScheduler } from '../scheduler/async';
import { timer } from './timer';
export function interval(period = 0, scheduler = asyncScheduler) {
  return timer(period, period, scheduler);
}
```

9.4 timer.js

src\rxjs\internal\observable\timer.js

```
import { Observable } from '../Observable';
import { asyncScheduler } from '../scheduler/async';
+export function timer(dueTime = 0, interval, scheduler = asyncScheduler) {
  return new Observable(subscriber => {
    let n = 0;
    return scheduler.schedule(function () {
      subscriber.next(n++);
+     if (interval >= 0) {
+       this.schedule(undefined, interval);
+     } else {
+       subscriber.complete();
+     }
    }, dueTime);
  });
}
```

10.take

- take 操作符会从 Observable 中取出前 N 个值，然后完成。它是一个过滤操作符，可以用来限制 Observable 发送的值的数量
- take 操作符会在 Observable 发送完 N 个值之后立即完成，因此你不需要使用 unsubscribe 方法取消订阅

10.1 src\index.js

src\index.js

```
import { interval, take } from './rxjs'
interval(500)
    .pipe(take(3))
    .subscribe(console.log)
```

10.2 rxjs\index.js

src\rxjs\index.js

```
export { Observable } from './internal/Observable';
export { of } from './internal/observable/of';
export { from } from './internal/observable/from';
export { fromEvent } from './internal/observable/fromEvent';
export { filter } from './internal/operators/filter';
export { map } from './internal/operators/map';
export { asyncScheduler } from './internal/scheduler/async';
export { timer } from './internal/observable/timer';
export { interval } from './internal/observable/interval';
+export { take } from './internal/operators/take';
```

10.3 take.js

src\rxjs\internal\operators\take.js

```
import { Observable } from '../Observable';
export function take(count) {
  return source => {
    let seen = 0;
    const observable = new Observable(function (subscriber) {
      return source.subscribe({
        ...subscriber,
        next: value => {
          seen++;
          if (seen if (seen >= count) {
              subscriber.complete();
            }
          }
        }
      });
    });
    return observable;
  };
}
```

11.Subject

- 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

11.1 Cold Observable和Hot Observable

- Hot Observable 和 Cold Observable是指两种不同类型的 Observable，它们在执行时的行为有所不同
- Cold Observable 是一种会在每个观察者订阅时重新开始发出数据的 Observable。每个观察者都有自己的数据流，即使多个观察者订阅同一个 Cold Observable，它们也会收到完全独立的数据流。例如，当你订阅一个 Cold Observable 时，它会从头开始发出数据，不会丢失任何信息
- Hot Observable是一种在发出数据时无论是否有观察者订阅都会继续发出数据的 Observable。每个观察者都会收到相同的数据流，并且会收到所有之前发出的数据。例如，当你订阅一个 Hot Observable时，它可能会丢失一些信息，因为它在你订阅之前就已经开始发出数据了
- 总的来说， Cold Observable 适用于那些需要每个观察者都收到完整数据流的场景，而 Hot Observable 适用于那些数据流是连续不断的，不需要每个观察者都收到完整数据流的场景

11.1.1 Cold Observable

- 推送值的生产者 producer来自 Observable内部。将会推送什么样的值在 Observable创建时被定义下来，不会改变
- producer与 observer是一对一的关系，即是unicast (单播)的
- 当有 observer订阅时， producer会把预先定义好的若干值依次推送给每个 observer
- Cold Observable每次订阅后就只会有一个观察者， 下一个观察者要进行订阅时是一次新的数据流程，因此 Cold Observable与 observer是一对一关系

- 数据流的 operators基本上都是属于 Cold Observable

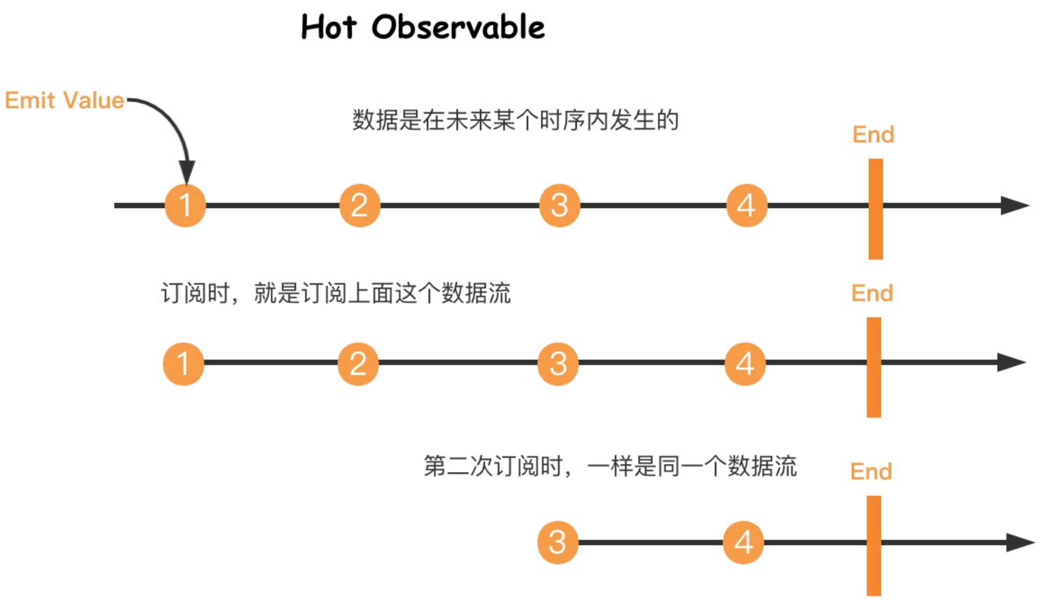
```
import { Observable } from './rxjs';
const source = new Observable(subscriber => {
    subscriber.next(1);
    subscriber.next(2);
    subscriber.next(3);
    subscriber.complete();
});

source.subscribe(data => console.log(`subscriberA: ${data}`));

source.subscribe(data => console.log(`subscriberB: ${data}`));
```

11.1.2 Hot Observable

-
    - 推送值的 producer来自 observable外部，何时推送以及推送什么样的值在创建时都是未知的。 producer与 observer是一对多的关系，即 **multicast** (多播)的
- 每当有 observer订阅时，会将 observer注册到观察者列表中
- 当外部的 producer被触发或执行时，会将值同时推送给所有的 observer


Hot Observable
Emit Value
数据是在未来某个时序内发生的
End
1
2
3
4
订阅时，就是订阅上面这个数据流
End
1
2
3
4
第二次订阅时，一样是同一个数据流
End
3
4

```
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);
```

11.1 src\index.js

src\index.js

```
import { Subject } from './rxjs'
const subject = new Subject();

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

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

11.2 rxjs\index.js

src\rxjs\index.js

```
export { Observable } from './internal/Observable';
export { of } from './internal/observable/of';
export { from } from './internal/observable/from';
export { fromEvent } from './internal/observable/fromEvent';
export { filter } from './internal/operators/filter';
export { map } from './internal/operators/map';
export { asyncScheduler } from './internal/scheduler/async';
export { timer } from './internal/observable/timer';
export { interval } from './internal/observable/interval';
export { take } from './internal/operators/take';
+export { Subject } from './internal/Subject';
```

11.3 Subject.js

src\rxjs\internal\Subject.js

```
import { Subscriber } from './Subscriber';
export class Subject extends Subscriber {
    observers = []
    subscribe(subscriber) {
        const { observers } = this;
        observers.push(subscriber);
    }
    next(value) {
        const copy = this.observers.slice();
        for (const observer of copy) {
            observer.next(value);
        }
    }
    complete() {
        const { observers } = this;
        while (observers.length) {
            observers.shift().complete?.();
        }
    }
}
```