

转载请注明链接: https://blog.csdn.net/feather_wch/article/details/88400574

RxJava 2.x实战场景

版本号:2019-03-15(17:50)

实例参考自: [RxJava-Android-Sample](#)

@[toc]

1-后台下载，前台更新进度

1、后台进行下载任务，前台更新下载的进度

```

public class ImageMainActivity extends AppCompatActivity {

    // 1、下载按钮
    Button mDownloadBtn;
    // 2、下载进度
    TextView mProgressTxt;
    // 3、管理下游事件，Activity销毁后释放所有后台任务，避免内存泄露。
    CompositeDisposable mCompositeDisposable = new CompositeDisposable();

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_image_main);
        // 1、下载按钮
        mDownloadBtn = findViewById(R.id.download_btn);
        mDownloadBtn.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                startDownload(); // 开启下载
            }
        });
        // 2、下载进度
        mProgressTxt = findViewById(R.id.download_progress_txt);
    }

    private void startDownload(){
        // 1、模拟下载任务
        Observable<Integer> observable = Observable.create(new ObservableOnSubscribe<Integer>() {
            @Override
            public void subscribe(Observer<Integer> observableEmitter) throws Exception {
                for (int i = 0; i < 100; i++) {
                    Thread.sleep(100);
                    observableEmitter.onNext(i);
                }
                observableEmitter.onComplete();
            }
        });
        // 2、下载后在UI层展示
        DisposableObserver<Integer> disposableObserver = new DisposableObserver<Integer>() {
            @Override
            public void onNext(Integer integer) {
                mProgressTxt.setText("下载进度:" + integer + "%");
            }

            @Override
            public void onError(Throwable throwable) {

            }

            @Override
            public void onComplete() {
                mProgressTxt.setText("下载完成");
            }
        };
    }
}

```

```
// 3、订阅事件
observable.subscribeOn(Schedulers.io()) // IO处理下载
    .observeOn(AndroidSchedulers.mainThread()) // UI处理页面
    .subscribe(disposableObserver);
// 4、管理
mCompositeDisposable.add(disposableObserver);
}

@Override
protected void onDestroy() {
    super.onDestroy();
    // 释放所有后台任务
    mCompositeDisposable.clear();
}
}
```

2-按键防抖动，一段时间内只触发一次点击【buffer】

2、按键防抖动，短时间内快速点击Button，会触发多次点击的流程，通过buffer进行处理，在短时间内只触发一次点击事件。

在两秒内快速点击10次：

1. 未防抖动：点击10次Button，弹出10个Toast
2. 防抖动：点击10次Button，弹出一个Toast

```

public class ImageMainActivity extends AppCompatActivity {

    // 1、按钮
    Button mBtn;
    // 2、Subject: 用于发送点击事件
    Subject<Integer> mClickSubject;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_image_main);

        // 1、DisposableObserver
        DisposableObserver<List<Integer>> clickDisposableObserver = new DisposableObserver<List<Integer>>() {
            @Override
            public void onNext(List<Integer> integers) {
                // 如果没点击会发送空事件过来，需要过滤。
                if(integers.size() <= 0){
                    return;
                }
                Toast.makeText(ImageMainActivity.this,
                    "点击按钮次数:" + integers.size(),
                    Toast.LENGTH_SHORT).show();
            }

            @Override
            public void onError(Throwable throwable) {}
            @Override
            public void onComplete() {}
        };

        // 2、初始化Observable.
        mClickSubject = PublishSubject.create();
        /**=====
        * 3、buffer将两秒内的点击事件化为1次
        * 1. 将两秒内的点击事件缓存起来，一次性发到UI端
        * 2. 即使一次也没有手动调用onNext，还是会间隔两秒发送一个空事件过去
        *=====*/
        mClickSubject.buffer(2, TimeUnit.SECONDS)
            .subscribeOn(Schedulers.io())
            .observeOn(AndroidSchedulers.mainThread())
            .subscribe(clickDisposableObserver);

        // 4、按钮
        mBtn = findViewById(R.id.download_btn);
        mBtn.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                mClickSubject.onNext(1);
            }
        });
    }

    @Override

```

```
protected void onDestroy() {  
    super.onDestroy();  
}  
}
```

3-关键词输入自动搜索【debounce、filter、switchMap】

1、关键词输入自动搜索，是根据用户输入的关键字即时搜索对应内容，需要解决三个问题。

1. 避免用户连续输入，导致大量无效搜索的情况。比如"123"，会依次搜索"1"、"2"、"3"。 --
- `debounce`操作符
2. 避免搜索关键字为空。--- `filter`
3. 避免旧关键字的结果覆盖了新关键字的结果。 --- `switchMap`

```

public class ImageMainActivity extends AppCompatActivity {

    // 1、关键字输入框
    EditText mSearchEt;
    // 2、Subject: 搜索的事件
    Subject<String> mSearchSubject;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_image_main);

        /**=====
        * 1、输入搜索关键词，进行搜索的逻辑处理。
        *   1. 用户连续输入内容，避免不必要的请求。debounce进行200ms内屏蔽。
        *   2. 搜索词为空的情况，进行过滤。
        *   3. 用户依次输入"old"、"new"等关键词，保证只展示最新的关键词的结果。
        *=====*/
        // 1. 初始化Subject
        mSearchSubject = PublishSubject.create();
        mSearchSubject
            // 2. 200ms内无新的搜索关键词，才真正发送事件。
            .debounce(200, TimeUnit.MILLISECONDS)
            // 3. 过滤无内容的关键字
            .filter(new Predicate<String>() {
                @Override
                public boolean test(String inputText) throws Exception {
                    return inputText.length() > 0;
                }
            })
            // 4. 只返回最后一个key的搜索结果
            .switchMap(new Function<String, ObservableSource<String>>() {
                @Override
                public ObservableSource<String> apply(final String key) throws Exception {
                    return Observable.create(new ObservableOnSubscribe<String>() {
                        @Override
                        public void subscribe(ObservableEmitter<String> observableEmitter) {
                            Log.d("feather", "搜索关键字 = " + key);

                            // 模拟网络请求的延迟
                            try {
                                Thread.sleep((long) (Math.random() * 500));
                            } catch (InterruptedException e) {
                                // 如果发送器没有被废弃，表明出现故障，上报
                                if (!observableEmitter.isDisposed()) {
                                    observableEmitter.onError(e);
                                }
                            }
                        }
                    })

                    // 返回搜索的结果
                    observableEmitter.onNext("搜索结果: " + key);
                    observableEmitter.onComplete();
                }
            }).subscribeOn(Schedulers.io()); // IO进行网络请求
    }
}

```

```

        }
    })
    .subscribeOn(Schedulers.io())
    .observeOn(AndroidSchedulers.mainThread())
    // 5. UI展示搜索结果
    .subscribe(new DisposableObserver<String>() {
        @Override
        public void onNext(String searchResult) {
            Toast.makeText(ImageMainActivity.this, searchResult, Toast.LENGTH_SHORT)
                .show();
        }

        @Override
        public void onError(Throwable throwable) {
            Toast.makeText(ImageMainActivity.this, throwable.getMessage(), Toast.LENGTH_SHORT)
                .show();
        }

        @Override
        public void onComplete() {
        }
    });

// 2、搜索框，处理输入监听器。
mSearchEt = findViewById(R.id.search_edittext);
mSearchEt.addTextChangedListener(new TextWatcher() {
    @Override
    public void beforeTextChanged(CharSequence s, int start, int count, int after) {
    }


    @Override
    public void onTextChanged(CharSequence s, int start, int before, int count) {
    }

    @Override
    public void afterTextChanged(Editable s) {
        // 发起请求处理"关键字搜索"的网络请求。
        mSearchSubject.onNext(s.toString().trim());
    }
});
}
}

```

4-轮询向服务器发起请求【intervalRange、repeatWhen】

1、定时向服务器发起请求，两种场景：

- 
1. 间隔固定时间，发起请求
 2. 间隔不等时间，发起请求


```

public class ImageMainActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_image_main);
        // 固定时延-轮询按钮
        findViewById(R.id.fixed_polling_btn).setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                startFixedPolling();
            }
        });
        // 变长时延-轮询按钮
        findViewById(R.id.variable_polling_btn).setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                startVariablePolling();
            }
        });
    }

    // 固定时延-轮询
    private void startFixedPolling(){
        // 1、固定时延轮询
        Observable.intervalRange(0,    // 第一个数据的值
                                100,  // 发送多少个数据
                                200,  // 间隔多久才发送第一个数据
                                400,  // 两项数据之间的间隔
                                TimeUnit.MILLISECONDS) // 时间单位
        .subscribeOn(Schedulers.io())
        .observeOn(AndroidSchedulers.mainThread())
        // 2、模拟发起请求
        .doOnNext(new Consumer<Long>() {
            @Override
            public void accept(Long aLong) throws Exception {
                // 发起网络请求
                Toast.makeText(ImageMainActivity.this, aLong+"", Toast.LENGTH_SHORT).show();
            }
        })
        .observeOn(AndroidSchedulers.mainThread())
        .subscribe(new DisposableObserver<Long>() {
            @Override
            public void onNext(Long aLong) {

            }

            @Override
            public void onError(Throwable throwable) {

            }

            @Override
            public void onComplete() {

            }
        })
    }
}

```

```

    });
}

// 计数器用于变长
private int mRepeatCount = 0;

// 变长时延-轮询
private void startVariablePolling(){

    Observable.just(0L)
        // 1、重新订阅后，触发该回调
        .observeOn(AndroidSchedulers.mainThread())
        .doOnComplete(new Action() {
            @Override
            public void run() throws Exception {
                //
                Toast.makeText(ImageMainActivity.this, mRepeatCount + "",
                    Toast.LENGTH_SHORT).show();
            }
        })
        // 2、每次完成后都重订阅
        .repeatWhen(new Function<Observable<Object>, ObservableSource<?>>() {
            @Override
            public ObservableSource<?> apply(Observable<Object> objectObservable) throws
            {
                return objectObservable.flatMap(new Function<Object, Observable<Long>>() {
                    {
                        @Override
                        public Observable<Long> apply(Object o) throws Exception
                        {
                            // 2. Timer间隔目标时间进行调用
                            return Observable.timer(++mRepeatCount * 200, TimeUnit.MILLIS);
                        }
                    }
                }));
            }
        }).subscribeOn(Schedulers.io()).observeOn(AndroidSchedulers.mainThread())
        .subscribe(new Consumer<Long>() {
            @Override
            public void accept(Long aLong) throws Exception {
                // 定时器到的时候都会回调
                Toast.makeText(ImageMainActivity.this, mRepeatCount + "",
                    Toast.LENGTH_SHORT).show();
            }
        }));
}
}

```

5-数据绑定的TextView(数据的改变会自动更新TextView) 【PublishProcessor】

1、TextView因为Model层数据的改变，自动更新本身显示的内容。

1. 利用 PublishProcessor 实现

```
public class ImageMainActivity extends AppCompatActivity {

    // 1. 自动更新的控件
    TextView mTextView;
    // 2. 通知TextView进行更新
    PublishProcessor<String> mPublishProcessor;
    // 3. 避免内存泄露
    Disposable mDisposable;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_image_main);

        // 1、根据数据改变自动更新的TextView
        mTextView = findViewById(R.id.databinding_txt);

        // 2、接收数据的改变
        mPublishProcessor = PublishProcessor.create();
        mDisposable = mPublishProcessor.subscribe(new Consumer<String>() {
            @Override
            public void accept(String text) throws Exception {
                mTextView.setText(text);
            }
        });

        // 3、数据的改变，会更新TextView展示的内容
        ((EditText)findViewById(R.id.search_edittext)).addTextChangedListener(new TextWatcher() {
            @Override
            public void beforeTextChanged(CharSequence s, int start, int count, int after) {}
            @Override
            public void onTextChanged(CharSequence s, int start, int before, int count) {}
            @Override
            public void afterTextChanged(Editable s) {
                mPublishProcessor.onNext(s.toString().trim());
            }
        });
    }

    @Override
    protected void onDestroy() {
        super.onDestroy();
        mDisposable.dispose();
    }
}
```

6-指数退避策略，在请求重试中的使用【retryWhen】

1、什么是指数退避策略

1. 根据输出的反馈(请求失败), 动态调整重试请求的等待时间
2. 一般请求失败时, 重新请求的等待时间为(2000ms), 如果再次失败, 则延长重试等待的时间(等待时间翻倍 = 4000ms)
3. 此外还有失败重新请求的次数, 失败过多则不继续重试。

2、采用指数退避策略, 进行请求的重试

```

public class ImageMainActivity extends AppCompatActivity {

    private static final String ERROR_MSG_RETRY = "error_msg_retry";

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_image_main);

        // 1、根据数据改变自动更新的TextView
        findViewById(R.id.retry_when_btn).setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                startRetryRequest();
            }
        });
    }

    @Override
    protected void onDestroy() {
        super.onDestroy();
    }

    private int mRetryCount = 0;

    private void startRetryRequest() {
        // 1、模拟网络请求，每次都失败。会进行重新订阅，subscribe()会重新执行
        Observable
            .create(new ObservableOnSubscribe<String>() {
                @Override
                public void subscribe(ObservableEmitter<String> observableEmitter) throws Exception {
                    // 每次都失败，每次都延长重试间隔。100,200,400,800,1600
                    observableEmitter.onError(new Throwable(ERROR_MSG_RETRY));
                }
            })
        // 2、请求失败后，onError()前调用。用于打印Toast。
        .observeOn(AndroidSchedulers.mainThread())
        .doOnError(new Consumer<Throwable>() {
            @Override
            public void accept(Throwable throwable) throws Exception {
                Toast.makeText(ImageMainActivity.this, "重新发起请求：" + mRetryCount *
            }
        })
        // 3、重试。onError()后才会触发retryWhen的重试操作。
        .retryWhen(new Function<Observable<Throwable>, ObservableSource<?>>() {
            @Override
            public ObservableSource<?> apply(Observable<Throwable> throwableObservable) {
                return throwableObservable.flatMap(new Function<Throwable, ObservableSource<?>>() {
                    @Override
                    public ObservableSource<?> apply(Throwable throwable) throws Exception {
                        // 3、根据时间进行重试操作。
                        String errorMsg = throwable.getMessage();

```

```

        if (ERROR_MSG_RETRY.equals(errorMsg)) {
            return Observable.timer(++mRetryCount * 200, TimeUnit.MILLIS);
        }
        // 4、其他异常
        return Observable.error(throwable);
    }
}
});
}
}).subscribeOn(Schedulers.io())
.observeOn(AndroidSchedulers.mainThread())
.subscribe(new Observer<String>() {
    @Override
    public void onSubscribe(Disposable disposable) {

    }

    @Override
    public void onNext(String s) {

    }

    @Override
    public void onError(Throwable throwable) {
        Toast.makeText(ImageMainActivity.this, throwable.getMessage(), Toast.LENGTH_SHORT).show();
    }

    @Override
    public void onComplete() {

    }
});
}
}

```

7-账号、密码、邮箱同时验证通过才允许进行后续的注册行为【combineLatest】

1、很多注册页面或者登陆页面，需要用户输入的账号和密码同时满足需求，才允许进行后续的操作。需要满足同时监听多个Observable，但是在统一的地方一起验证。

1. 利用 combineLatest 在账号/密码都起码输入过一次后，进行统一验证。

```

public class ImageMainActivity extends AppCompatActivity {

    Button mButton;
    CompositeDisposable mCompositeDisposable = new CompositeDisposable();

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_image_main);

        // 1、注册按钮
        mButton = findViewById(R.id.button);

        // 2、账号和密码的Subject
        PublishSubject accountPublishSubject = PublishSubject.create();
        PublishSubject passwordPublishSubject = PublishSubject.create();
        DisposableObserver disposableObserver;
        /**=====
        * 3、当任意一个Observable发射数据之后，会去取其它Observable最近一次发射的数据。
        * 1. 验证账号和密码的合法性
        *=====*/
        Observable.combineLatest(accountPublishSubject,
            passwordPublishSubject,
            new BiFunction<String, String, Boolean>() {
                @Override
                public Boolean apply(String account, String password) throws Exception {
                    // 账号需要长度大于5
                    // 密码需要长度大于8
                    return account.length() > 5 && password.length() > 8;
                }
            }).subscribe((disposableObserver = new DisposableObserver<Boolean>() {
                @Override
                public void onNext(Boolean value) {
                    mButton.setText(value ? "登陆" : "账号密码不合法");
                    mButton.setClickable(value); // 是否可以点击
                }
                @Override
                public void onError(Throwable throwable) {}
                @Override
                public void onComplete() {}
            }));

        // 4、统一管理，防止内存泄露。
        mCompositeDisposable.add(disposableObserver);

        // 5、账号/密码输入框设置监听器
        ((EditText) findViewById(R.id.account_edittext))
            .addTextChangedListener(new EditTextWatcher(accountPublishSubject));
        ((EditText) findViewById(R.id.password_edittext))
            .addTextChangedListener(new EditTextWatcher(passwordPublishSubject));
    }
}

```

```

/**=====
 * @功能 发出EditText输入的内容
 * =====*/
class EditTextWatcher implements TextWatcher {

    PublishSubject mPublishSubject;
    public EditTextWatcher(PublishSubject publishSubject) {
        this.mPublishSubject = publishSubject;
    }
    @Override
    public void beforeTextChanged(CharSequence s, int start, int count, int after) {}
    @Override
    public void onTextChanged(CharSequence s, int start, int before, int count) {}
    @Override
    public void afterTextChanged(Editable s) {
        mPublishSubject.onNext(s.toString());
    }
}

// 避免内存泄露
@Override
protected void onDestroy() {
    super.onDestroy();
    mCompositeDisposable.clear();
}
}

```

8-先从硬盘获取缓存，再从网络获取数据。

concat

1、concat先获取缓存的数据，再去请求网络数据。

1. 本地缓存: 500ms
2. 网络数据: 2000ms
3. 先500ms + 再2000ms = 2500ms


```

public class ImageMainActivity extends AppCompatActivity {

    CompositeDisposable mCompositeDisposable = new CompositeDisposable();

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_image_main);

        findViewById(R.id.button).setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                // 1、接收到缓存or网络数据
                DisposableObserver disposableObserver = getDisposableObserver();
                mCompositeDisposable.add(disposableObserver); // 统一管理，防止内存泄露。

                Observable<List<String>> observable = Observable.concat(
                    getCacheDataObservable().subscribeOn(Schedulers.io()),
                    getNetworkDataObservable().subscribeOn(Schedulers.io()));
                // 2、进行数据的查询
                observable.observeOn(AndroidSchedulers.mainThread())
                    .subscribe(disposableObserver);
            }
        });
    }

    // 避免内存泄露
    @Override
    protected void onDestroy() {
        super.onDestroy();
        mCompositeDisposable.clear();
    }

    private Observable<List<String>> getCacheDataObservable(){
        return Observable.create(new ObservableOnSubscribe<List<String>>() {
            @Override
            public void subscribe(ObservableEmitter<List<String>> observableEmitter) throws Exception {
                Log.d("feather", "开始查询【缓存】数据");
                Thread.sleep(500);
                Log.d("feather", "开始加载【缓存】数据");
                ArrayList cacheDatas = new ArrayList();
                cacheDatas.add("1 cache");
                cacheDatas.add("2 cache");
                cacheDatas.add("3 cache");
                observableEmitter.onNext(cacheDatas);
                observableEmitter.onComplete();
            }
        });
    }

    private Observable<List<String>> getNetworkDataObservable(){
        return Observable.create(new ObservableOnSubscribe<List<String>>() {

```

```

@Override
public void subscribe(ObservableEmitter<List<String>> observableEmitter) throws Exception {
    Log.d("feather", "开始查询【网络】数据");
    Thread.sleep(2000);
    Log.d("feather", "开始加载【网络】数据");
    ArrayList netDatas = new ArrayList();
    netDatas.add("1 net");
    netDatas.add("2 net");
    netDatas.add("3 net");
    observableEmitter.onNext(netDatas);
    observableEmitter.onComplete();
}
});
}

private DisposableObserver<List<String>> getDisposableObserver(){
    return new DisposableObserver<List<String>>() {
        @Override
        public void onNext(List<String> strings) {
            for (String string : strings) {
                Log.d("feather", string);
            }
        }
        @Override
        public void onError(Throwable throwable) {
            Log.d("feather", "onError = " + throwable.getMessage());
        }
        @Override
        public void onComplete() {
            Log.d("feather", "onComplete");
        }
    };
}
}
}

```

concatEager

2、concatEager让本地缓存和网络请求同时进行

1. 能够同步进行 读取本地缓存 、 请求网络数据 的操作。 但是后者会等待前者完成
2. 例如 读取缓存时间很长 = 2000ms , 网络请求500ms ,最终时间 2000ms

```
// 1、接收到缓存or网络数据
DisposableObserver disposableObserver = getDisposableObserver();
mCompositeDisposable.add(disposableObserver); // 统一管理，防止内存泄露。
// 2、concatEager需要列表
List<Observable<List<String>>> observableList = new ArrayList<>();
observableList.add(getCacheDataObservable().subscribeOn(Schedulers.io()));
observableList.add(getNetworkDataObservable().subscribeOn(Schedulers.io()));
// 3、concatEager
Observable<List<String>> observable = Observable.concatEager(observableList);
// 4、进行数据的查询
observable.observeOn(AndroidSchedulers.mainThread())
    .subscribe(disposableObserver);
```

merge

3、merge因为前后无序，可能会导致缓存数据覆盖网络数据

```
DisposableObserver disposableObserver = getDisposableObserver();
mCompositeDisposable.add(disposableObserver); // 统一管理，防止内存泄露。
// 查询缓存和网络数据
Observable.merge(
    getCacheDataObservable().subscribeOn(Schedulers.io()),
    getNetworkDataObservable().subscribeOn(Schedulers.io()))
    .observeOn(AndroidSchedulers.mainThread())
    .subscribe(disposableObserver);
```

takeUntil、merge、publish

4、实现代码如下:

1. takeUntil: 保证在网络数据返回后，缓存数据不会再发送
2. merge: 缓存请求、网络请求同步进行
3. publish: 避免takeUntil、merge导致的多次订阅的问题(会多次请求数据)

```

public class ImageMainActivity extends AppCompatActivity {

    CompositeDisposable mCompositeDisposable = new CompositeDisposable();

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_image_main);

        findViewById(R.id.button).setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                // 1、统一管理，防止内存泄露。
                DisposableObserver<List<String>> disposableObserver = getDisposableObserver();
                mCompositeDisposable.add(disposableObserver);
                // 2、请求网络数据Observable
                final Observable<List<String>> network = getNetworkDataObservable().subscribeOn(
                    Observable.<List<String>> observable = network.publish(new Function<Observable<List<String>>,
                        @Override
                        public ObservableSource<List<String>> apply(Observable<List<String>> network) {
                            // 3、缓存数据takeUntil在网络数据返回后，就不会发出任何数据了。
                            Observable<List<String>> cache = getCacheDataObservable().subscribeOn(Schedulers.io());
                            // 4、缓存、网络同步开始请求
                            return Observable.merge(network, cache);
                        }
                    ));
                // 5、开始订阅
                observable
                    .subscribeOn(Schedulers.io())
                    .observeOn(AndroidSchedulers.mainThread())
                    .subscribe(disposableObserver);
            }
        });
    }

    // 避免内存泄露
    @Override
    protected void onDestroy() {
        super.onDestroy();
        mCompositeDisposable.clear();
    }

    private Observable<List<String>> getCacheDataObservable(){
        return Observable.create(new ObservableOnSubscribe<List<String>>() {
            @Override
            public void subscribe(ObserverEmitter<List<String>> observableEmitter) throws InterruptedException {
                try{
                    Log.d("feather", "开始查询【缓存】数据");
                    Thread.sleep(1000);
                    Log.d("feather", "开始加载【缓存】数据");
                    ArrayList cacheDatas = new ArrayList();
                    cacheDatas.add("1 cache");
                } catch (Exception e) {
                    e.printStackTrace();
                }
                observableEmitter.onNext(cacheDatas);
            }
        });
    }
}

```

```

        cacheDatas.add("2 cache");
        cacheDatas.add("3 cache");
        observableEmitter.onNext(cacheDatas);
    }catch (Exception e){
        if(!observableEmitter.isDisposed()){
            observableEmitter.onError(new Throwable("缓存数据加载失败"));
        }
    }
}

});
}

private Observable<List<String>> getNetworkDataObservable(){
    return Observable.create(new ObservableOnSubscribe<List<String>>() {
        @Override
        public void subscribe(ObservableEmitter<List<String>> observableEmitter) throws Int
            try{
                Log.d("feather", "开始查询【网络】数据");
                Thread.sleep(2000);
                Log.d("feather", "开始加载【网络】数据");
                ArrayList netDatas = new ArrayList();
                netDatas.add("1 net");
                netDatas.add("2 net");
                netDatas.add("3 net");
                observableEmitter.onNext(netDatas);
            }catch (Exception e){
                if(!observableEmitter.isDisposed()){
                    observableEmitter.onError(new Throwable("网络数据加载失败"));
                }
            }
        }
    });
}

private DisposableObserver<List<String>> getDisposableObserver(){
    return new DisposableObserver<List<String>>() {
        @Override
        public void onNext(List<String> strings) {
            for (String string : strings) {
                Log.d("feather", string);
            }
        }
        @Override
        public void onError(Throwable throwable) {
            Log.d("feather", "onError = " + throwable.getMessage());
        }
        @Override
        public void onComplete() {
            Log.d("feather", "onComplete");
        }
    };
}
}

```

9-延迟1s执行一个任务/周期性执行一个任务/执行一个任务 延迟1s后执行另一个任务【timer、interval、delay】

timer

1、使用timer在延迟1秒后执行一个任务

```
/**=====
 * @function 使用timer延时1s后执行任务
 *=====*/
private void startBytime(){
    Observable.timer(1000, TimeUnit.MILLISECONDS)
        .subscribeOn(Schedulers.io())
        .observeOn(AndroidSchedulers.mainThread())
        .subscribe(new DisposableObserver<Long>() {
            @Override
            public void onNext(Long value) {
                Log.d("feather", "value = " + value);
            }
            @Override
            public void onError(Throwable throwable) {
                Log.d("feather", throwable.getMessage());
            }
            @Override
            public void onComplete() {
                Log.d("feather", "onComplete");
            }
        });
}
```

interval

1、每隔1s执行一次任务，第一个任务前也有1s的间隔

1. 第一个任务执行前，间隔2000ms
2. 后续每隔任务的执行周期是，1000ms

```

private void startByInterval(){
    Observable.interval(2000, 1000, TimeUnit.MILLISECONDS)
        .subscribeOn(Schedulers.io())
        .observeOn(AndroidSchedulers.mainThread())
        .subscribe(new DisposableObserver<Long>() {
            @Override
            public void onNext(Long value) {
                Log.d("feather", "value = " + value);
            }
            @Override
            public void onError(Throwable throwable) {
                Log.d("feather", throwable.getMessage());
            }
            @Override
            public void onComplete() {
                Log.d("feather", "onComplete");
            }
        });
}

```

2、周期性1s的间隔执行一个任务，只执行五次

1. interval + take 组合

```

private void startByInterval(){
    Observable.interval(2000, 1000, TimeUnit.MILLISECONDS)
        .take(5) // 【重点】
        .subscribeOn(Schedulers.io())
        .observeOn(AndroidSchedulers.mainThread())
        .subscribe(new DisposableObserver<Long>() {
            @Override
            public void onNext(Long value) {
                Log.d("feather", "value = " + value);
            }
            @Override
            public void onError(Throwable throwable) {
                Log.d("feather", throwable.getMessage());
            }
            @Override
            public void onComplete() {
                Log.d("feather", "onComplete");
            }
        });
}

```

delay

3、先执行第一个任务，等待2s后，再执行第二个任务

```

private void startByDelay(){
    Observable.just(0)
        // 1、立即执行一个任务
        .doOnNext(new Consumer<Integer>() {
            @Override
            public void accept(Integer value) throws Exception {
                Log.d("feather", "first value = " + value);
            }
        })
        // 2、延迟下游事件的接收。在2000ms后才发送onNext，去执行后续的任务。
        .delay(2000, TimeUnit.MILLISECONDS)
        .subscribeOn(Schedulers.io())
        .observeOn(AndroidSchedulers.mainThread())
        // 3、执行第二个任务
        .subscribe(new DisposableObserver<Integer>() {
            @Override
            public void onNext(Integer value) {
                Log.d("feather", "value = " + value);
            }
            @Override
            public void onError(Throwable throwable) {
                Log.d("feather", throwable.getMessage());
            }
            @Override
            public void onComplete() {
                Log.d("feather", "onComplete");
            }
        });
}

```

10-Activity销毁后重建，后台任务不中断，能继续展示进度【ConnectableObservable、publish、connect】

1、Activity可能会因为横竖屏切换等原因导致重建，需要保持Fragment中的后台任务不会中断，并且在重建之后继续展示任务进度

1. Fragment使用 `setRetainInstance(true)`; ---避免Fragment的重建，并且只回调一次 `onCreate()`方法
2. 利用 `ConnectableObservable` , `Hot Observable` 持续的发出数据，在重建后，还能继续接收数据。


```

public class WorkFragment extends Fragment {

    private static final String TAG = "wchf";

    TextView mProgressText;

    ConnectableObservable<Integer> mConnectableObservable;
    DisposableObserver mDisposableObserver;

    @Override
    public void onCreate(@Nullable Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        Log.d(TAG, "onCreate");

        /**=====
        * @function 用于只设置一个热被观察者，持续发送数据出去。
        * 设置true后，Activity的重新创建不会导致Fragment的重建：
        * 1. 不会再触发onCreate
        * 2. 会触发onAttach
        * 3. 会触发onActivityCreated等等方法
        *=====*/
        setRetainInstance(true);

        // 1、创建普通的Observable
        mConnectableObservable = Observable.create(new ObservableOnSubscribe<Integer>() {
            @Override
            public void subscribe(ObservableEmitter<Integer> observableEmitter) throws
                try{
                    for (int i = 0; i < 100; i++) {
                        Thread.sleep(100);
                        observableEmitter.onNext(i);
                    }
                    observableEmitter.onComplete();
                }catch (Exception e){
                    e.printStackTrace();
                }
            }
        })
        .subscribeOn(Schedulers.io())
        .observeOn(AndroidSchedulers.mainThread())
        // 2、通过publish，将Cold Observable转换为Hot Observable
        .publish();
        // 3、下游观察者，展示下载进度
        mDisposableObserver = new DisposableObserver<Integer>() {
            @Override
            public void onNext(Integer integer) {
                mProgressText.setText("下载进度:" + integer + "%");
            }
            @Override
            public void onError(Throwable throwable) { }
            @Override
            public void onComplete() {
                mProgressText.setText("下载完成");
            }
        }
    }
}

```

```

};
// 4、订阅：监听下载进度
mConnectableObservable.subscribe(mDisposableObserver);
}
@Override
public View onCreateView(LayoutInflater inflater, ViewGroup container,
                          Bundle savedInstanceState) {
    Log.d(TAG, "onCreateView");
    View view = inflater.inflate(R.layout.fragment_work, container, false);
    view.findViewById(R.id.back_work_btn).setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            // 5、开启后台任务。---开始持续发送数据。
            mConnectableObservable.connect();
        }
    });
    mProgressText = view.findViewById(R.id.back_work_progress_txt);
    return view;
}
@Override
public void onResume() {
    super.onResume();
    Log.d(TAG, "onResume");
}
@Override
public void onDestroy() {
    Log.d(TAG, "onDestroy");
    super.onDestroy();
    if(mDisposableObserver != null && !mDisposableObserver.isDisposed()){
        mDisposableObserver.dispose();
    }
}
}
}

```

参考资料

1. [RxJava-Android-Sample](#)
2. [RxJava2 实战知识梳理](#)