● 액티비티 ANR(Application Not Responding): 액티비티가 사용자 이벤트에 반응하지 못하는 상황. 액티비티가 화면에 출력된 상황에서 사용자 이벤트에 5초 이내 반응하지 못하면 시스템에서 액티비티를 강제로 종료한다. 즉, 액티비티 내에서 특정 업무처리 로직을 수행하는 데 시간이 오래 걸려 사용자의 이벤트를 처리하지 못하고 5초가 지나는 상황이다. 가장 대표적으로 액티비티에서 ANR 문제를 발생시키는 부분이 네트워크다. 서보 연동을 위한 네트워킹은 HTTP 프로토콜을 이용하거나 TCP/IP 프로토콜을 이용하는데, 유선 네트워킹에서는 신뢰할 수 있는 네트워크 연결을 이용하므로 빠르게 처리될 수 있지만, 모바일처럼 무선 네트워크를 사용하는 경우는 시간이 오래 걸린다고 생각해야 한다. 또한 종종 네트워킹이 안 되는 상황도 발생하므로 액티비티의 네트워킹은 모조건 ANR을 고려해서 작성해야 한다. (그렇다고 Service에 작성하기에는 시간이 오래 걸린다고 하더라도 그 작업 자체가 앱의 화면이 출력되고 있는 상황에서만 필요한 작업이며, 앱이 화면을 점유하지 못하고 다른 앱에 가려진 상황에서는 의미 없는 작업이라면 Service가 아닌 액티비티에서 처리하는 게 좋다. Service는 작성 시 많은 주의를 기울여야 하는 컴포너늩이며 자칫 잘못 작성하면 사용자에게 악성 앱이 되기 쉬운 컴포넌트이므로 불필요하게 Service를 만드는 건 좋지 않다.)

● RxJava를 이용한 ANR 해결: 액티비티에서 업무 처리를 하느라 5초 이상 사용자 이벤트를 처리하지 못하는 건, 액티비티의 수행 흐림이 있기 때문이다. 인텐트가 발생하여 액티비티가 생성되고 onCreate() 함수부터 차례대로 호출되는 액티비티의 수행 흐름으로, 시간이 오래 걸리는 작업이 있을 때는 사용자 이벤트 처리가 안 되는 것이다. 결국, ANR 문제를 해결하기 위해서는 비동기 프로그램을 구축해야 한다. 비동기 프로그램이란, 하나의 업무가 수행되는 동안 다든 업무가 대기 상태에 들어가지 않고 동시에 실행되도록 하는 프로그램을 의미한다. 비동기 프로그램 중 대표적인 것이 스레드 프로그램이며, 스레드 프로그램을 이용하면 ANR을 해결할 수 있다. 안드로이드 버전 11에서 Thread-Handler와 AsyncTask 등의 방법이 모두 deprecated 되었고, 코ᄃᆞ틴(coroutine) 사용을 권장하는데, 이는 코틀린 언어에서 제공하는 기능이다. 자바는 RxJava를 이용해서 비동기 프로그램을 구축한다.

▶ RxJava: Rx는 Reactive의 약어로, 반응형 프로그래밍(reactive programming)을 통칭하는 단어이다. 이 반응형 프로그래밍 방식을 자바 언어에서 지원하는 라이브러리가 바로 RxJava이다. 언어마다 반응형 프로그래밍을 지원하는 별도의 라이브러리가 있다(ex. 자바스크립트에서 지원하는 라이브러리는 RxJS). 반응형 프로그래밍을 비동기식 데이터 흐름을 사용한 프로그래밍이라고 설명한다. 즉, 반응형 프로그래밍은 비동기 프로그래밍을 제공하기 위해 사용하는 방식이다.

우선, RxJava를 안드로이드 build.gardle 파일의 dependencies로 설정한다.

```
implementation 'io.reactivex.rxjava3:rxandroid:3.0.0'
implementation 'io.reactivex.rxjava3:rxjava:3.0.0'
```

반응형 프로그래밍은 Observable과 Observer로 구분된다. Observable에서 어떤 업무가 처리되고 데이터가 발생하며, Observer은 그 업무처리 상태를 확인하거나, 데이터를 획득해야 하는 역할자이다. Observer가 Observable에 자신을 등록하여 Observable의 결과를 구독(subscribe)하는 구조이다. 안드로이드 프로그램에서 ANR 문제가 생기면, 코드를 Observable에 구현하고 이를 Observer로 이용하여 해결할 수 있다.

Observable은 Observable.create() 함수로 만들며, 이 함수의 매개변수로 ObservableOnSubscribe을 구현한 객체를 지정한다. ObservableOnSubscribe<Integer>처럼 제네릭으로 선언된 부분은 Observable이 발생하는 데이터 타입이다. ObservableOnSubscribe 인터페이스의 subscribe() 함수를 오버라이드하여 비동기적으로 처리되어야 하는 로직은 담는다. 그리고 Observer에게 데이터를 전달할 때는 subscribe() 함수의 매개변수로 전될된 ObservableEmitter 객체를 이용하여, 이 객체의 onNext() 함수를 이용해 전달한다.

```
Observable observable = Observable.create(new ObservableOnSubscribe<Integer>() {
    @Override
    public void subscribe(@NonNull ObservableEmitter<Integer> emitter) throws Throwable
        try {
            // 시간이 오래 걸리는 작업. <Integer> 데이터 타입.
            for (int i = 0; i < 3; i++) {
                Log.d("kkang", "observer emit " + i);
                emitter.onNext(i); // Observer에게 데이터 전달
                Thread.sleep(1000); // 1초 쉬기
            }
        } catch (Exception e) {
            e.printStackTrace();
            if (!emitter.isDisposed()) { // emitter가 바뀌지 않았다면
                emitter.onError(e); // Observer에게 에러가 발생했다는 것을 전달
            }
        } finally {
            if (!emitter.isDisposed()) { // emitter가 바뀌지 않았다면
                emitter.onComplete(); // Observer에게 끝났다는 것을 전달
            }
        }
    }
});
```

이렇게 Observable이 준비되었으면, 이제 Observable이 발생한 데이터를 획득할 Observer을 구현해야 한다. 인터페이스 Observer을 구현한 객체를 Observable의 subscribe() 함수의 매개변수로 지정하면, Observer가 Observable의 결과를 구독하게 되어 Observable에서 데이터가 발생하거나 에러가 발생하는 등의 다양한 상황에서 Observer의 알맞의 함수가 호출된다.

```
observable.subscribe(new Observer() {
    @Override
    public void onSubscrive(@NonNull Disposable d) {
        Log.d("kkang", "onSubscribe...");
    }

    @Override
    public void onNext(@androidx.annotation.NonNull Object o) {
        Log.d("kkang", "onNext..." + (Integer) o); // 넘어온 데이터도 함께 출력
    }

    @Override
    public void onError(@NonNull Throwable e) {
        Log.d("kkang", "onError"...");
    }

    @Override
    public void onComplete() {
        Log.d("kkang", "onComplete...");
    }
});
```

Resources: 깡쌤의 안드로이드 프로그래밍