

# 초보 개발자의 RxJAVA 적용기

—

배현빈 @23기 회장

**RxJAVA**

**RxJAVA**

**= Reactive**

# Reactive Programming

Reactive Programming  
streams.

asynchronous data

리액티브 프로그래밍이다.



이하는 프로그

# Reactive

영어 - 감지됨 ▾



↔



한국어 ▾

reactive  
rē'aktiv

×

반응성  
ban-eungseong

'reactive'의 번역

형용사

반동적인  
reactive, reflex, reactionary

복고적인  
reactive

반응적인  
reactive

반응성  
반응적인  
반동적인

# 요약

## Reactive Programming

 반응적인, 실시간적인 프로그래밍

즉, 비동기적인 데이터의 흐름을 실시간으로 처리하는 프로그래밍

# 동기와 비동기

## 동기 (Synchronous)

요청이 들어온 순서에 맞게 하나씩 처리하는 방식

## 비동기 (Asynchronous)

하나의 요청에 따른 응답을 즉시 처리하지 않아도,  
그 대기 시간동안 또 다른 요청을 처리하는 방식

# Reactive Programming

리액티브 프로그래밍은 비동기적인 데이터스트림을 처리하는 프로그래밍이다.

# 데이터 스트림

시간의 흐름

작성 버튼



화면터치



데이터의 물줄기 흐름



# 실제 사례



간단한 게시물

문제점

- 게시판, 게시물, 댓글을 가져올 수 있는 API가 각각 다 다르다

# 게시글 flow



이상적인 흐름 : 동기식으로 순서대로 받아오기

## 문제점

안드로이드에서는 허니컴 이후로 메인스레드에서 네트워크 통신을 금지한다.

# NetworkOnMainThreadException

Android 개발자 > Docs > 참조

☆☆☆☆☆

## NetworkOnMainThreadException

Added in API level 11

[Kotlin](#) | [Java](#)

```
public class NetworkOnMainThreadException
extends RuntimeException

java.lang.Object
└─ java.lang.Throwable
    └─ java.lang.Exception
        └─ java.lang.RuntimeException
            └─ android.os.NetworkOnMainThreadException
```

The exception that is thrown when an application attempts to perform a networking operation on its main thread.

This is only thrown for applications targeting the Honeycomb SDK or higher. Applications targeting earlier SDK versions are allowed to do networking on their main event loop threads, but it's heavily discouraged. See the document [Designing for Responsiveness](#).

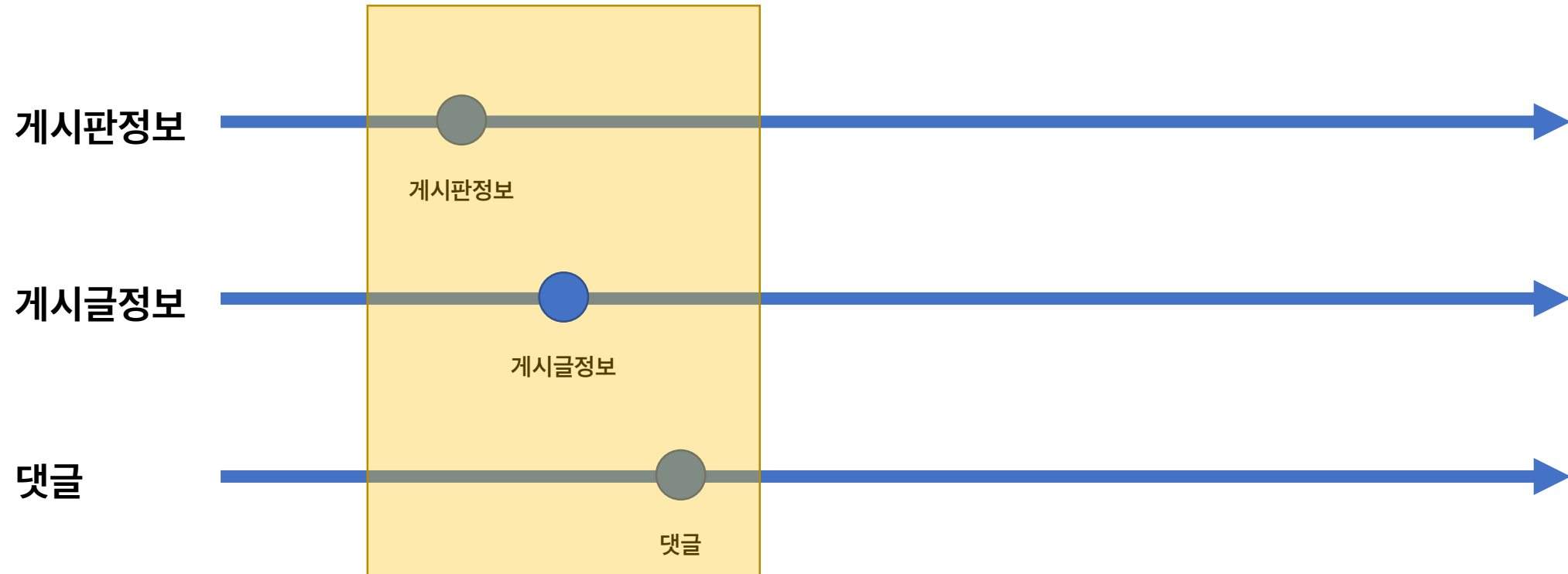
- The exception that is thrown when an application attempts to perform a networking operation on its main thread.
- 애플리케이션이 메인스레드에서 네트워킹 작업을 수행하려고 할 때 발생하는 예외입니다.

# 게시글 flow - 이상



이상적인 흐름 : 동기식으로 순서대로 받아오기

# 게시글 flow - 현실



**3가지 정보를 전부 가져오기 위한 방법**

한가지 작업이 끝나고 나서 그 직후 다른 작업을 수행하거나  
3개의 데이터를 최종적으로 합치는 방법

# 기존에 안드로이드에서 Retrofit을 이용하던 방법

```
import retrofit2.Call
import retrofit2.http.GET
import retrofit2.http.Path

interface TestService {
    @GET("/v1/posts/{boardId}/{postId}")
    fun getSinglePost(@Path("boardId")boardId: Int, @Path("postId")postId: Int): Call<Post>

    @GET("/v1/board/{boardId}")
    fun getSingleBoard(@Path("boardId")boardId: Int): Call<Board>

    @GET("/v1/comment/{boardId}/{postId}")
    fun getCommentList(@Path("boardId")boardId: Int, @Path("postId")postId: Int): Call<List<Comment>>
}
```

Call<Board> : 게시판 정보를 불러오기 위한 call

Call<Post> : 게시글 정보를 불러오기 위한 call

Call<List<Comment>> : 댓글 목록을 불러오기 위한 call

# 기존에 안드로이드에서 Retrofit을 이용하던 방법

순서대로 불러오기 위해  
다음처럼 콜백

```
fun loadArticle() {  
    val testService = retrofit.create(TestService::class.java)  
    var boardCall: Call<Board> = testService.getSingleBoard(1)  
    boardCall.enqueue(object: Callback<Board> {
```

```
1  function call(win) {  
2      // for listener purpose  
3      return function() {  
4          loadLink(win, REMOTE_SRC+'/assets/css/style.css', function() {  
5              loadLink(win, REMOTE_SRC+'/lib/async.js', function() {  
6                  loadLink(win, REMOTE_SRC+'/lib/easyXDM.js', function() {  
7                      loadLink(win, REMOTE_SRC+'/lib/json2.js', function() {  
8                          loadLink(win, REMOTE_SRC+'/lib/underscore.min.js', function() {  
9                              loadLink(win, REMOTE_SRC+'/lib/backbone.min.js', function() {  
10                                 loadLink(win, REMOTE_SRC+'/dev/base_dev.js', function() {  
11                                     loadLink(win, REMOTE_SRC+'/assets/js/deps.js', function() {  
12                                         loadLink(win, REMOTE_SRC+'/src/' + win.loader_path + '/loader.js', function() {  
13                                             async.eachSeries(SCRIPTS, function(src, callback) {  
14                                                 loadScript(win, BASE_URL+src, callback);  
15                                             });  
16                                         });  
17                                     });  
18                                 });  
19                             });  
20                         });  
21                     });  
22                 });  
23             });  
24         });  
25     });  
};
```

```
use<List<Comment>>) {
```

```
    })  
}
```

# RxJAVA를 이용한 개선

정보를 전부 가져오기 위한 방법

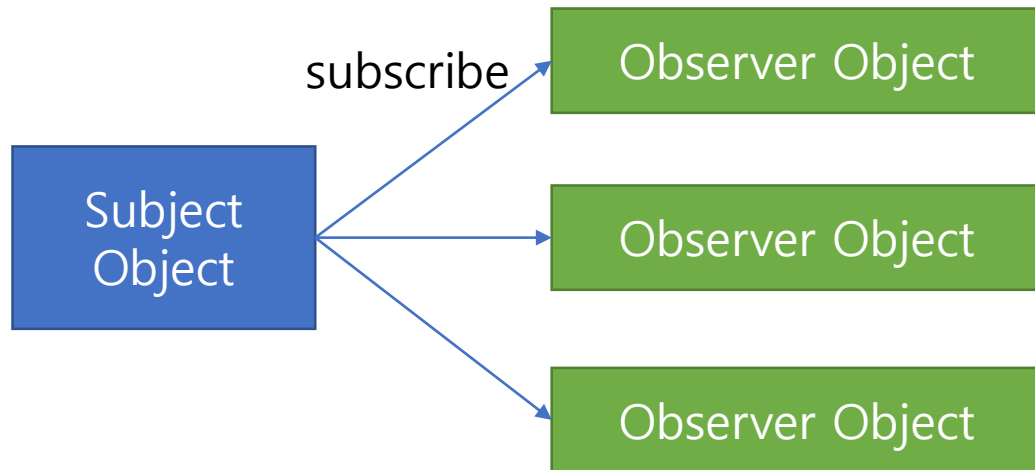
한가지 작업이 끝나고 나서 그 직후 다른 작업을 수행하거나  
3개의 데이터를 최종적으로 합치는 방법

# HOW?



# Observer

주체 객체(Subject Object)의 상태에 변화가 생기면  
관찰 객체(Observer Object)에서도 변화가 생긴다.



# RxJAVA의 observable

말 그대로 observe 할 수 있는 객체  
각 작업을 observable로 만들어놓고 구독 (subscribe)할 수 있다.



각 작업을 subscribe하여 순서대로 진행하거나  
여러 개의 작업을 나중에 완료한 후 합칠 수 있다.

# RxJAVA를 기반으로 변경하기

```
import io.reactivex.rxjava3.core.Observable
import retrofit2.http.GET
import retrofit2.http.Path

interface TestService {
    @GET("/v1/posts/{boardId}/{postId}")
    fun getSinglePost(@Path("boardId")boardId: Int, @Path("postId")postId: Int): Observable<Post>

    @GET("/v1/board/{boardId}")
    fun getSingleBoard(@Path("boardId")boardId: Int): Observable<Board>

    @GET("/v1/comment/{boardId}/{postId}")
    fun getCommentList(@Path("boardId")boardId: Int, @Path("postId")postId: Int):
    Observable<List<Comment>>
}
```

서비스의 리턴값이 Call<T>에서 Observable<T>로 변경된 것을 알 수 있음

# 1) 순서대로 진행

```
fun loadArticle() {  
    val testService = retrofit.create(TestService::class.java)  
    testService.getSingleBoard(1)  
        .subscribeOn(Schedulers.io())  
        .observeOn(AndroidSchedulers.mainThread())  
        .flatMap { board ->  
            testService.getSinglePost(1, 1)  
        }  
        .flatMap { post: Post? ->  
            testService.getCommentList(1, 1)  
        }  
        .subscribe({commentList ->  
            //작업 완료  
        }, {t ->  
            Log.e(TAG, t.message)  
        })  
}
```

flatMap() : 가공한 데이터를 Observable<T> 형태로 리턴  
코드의 가독성이 상당히 개선된 것을 볼 수 있다.

## 2) 여러개의 작업을 진행한 뒤 한꺼번에 가져오기

```
fun loadArticle() {  
    val testService = retrofit.create(TestService::class.java)  
    Observable.combineLatest(testService.getSingleBoard(1),  
        testService.getSinglePost(1, 1),  
        testService.getCommentList(1, 1),  
        Function3 { board: Board, post: Post, commentList: List<Comment> ->  
            Article(board, post, commentList)  
        })  
        .subscribeOn(Schedulers.io())  
        .observeOn(AndroidSchedulers.mainThread())  
        .subscribe({ article ->  
            Log.d(TAG, "boardId : ${article.board.id}")  
            Log.d(TAG, "postId : ${article.post.id}")  
            Log.d(TAG, "title : ${article.post.title}")  
            Log.d(TAG, "first comment : ${article.commentList[0].content}")  
        }, { t ->  
            Log.e(TAG, t.message)  
        })  
}
```

제일 마지막의 subscribe() 메소드는 모든 데이터를 받아온 뒤 실행하게 된다.

# Observable의 구현체

- **Observable**
  - 가장 기본적인 Observer패턴의 구현체
- **Single**
  - Observable의 특수한 형태
  - Observable 클래스는 데이터를 무한하게 발행할 수 있지만 Single 클래스는 오직 1개의 데이터만 발행하도록 한정
  - 데이터 하나가 발행과 동시에 존재
- **Maybe**
  - Single과 마찬가지로 최대 데이터 하나를 가질 수 있지만 데이터 발행 없이 바로 데이터 발생을 완료할 수 있다.
- **Completable**
  - 비동기 처리 후 반환되는 결과가 없는 경우 사용

**감사합니다.**

—