```
D/TAG: Observable thread is : RxNewThreadScheduler-1

D/TAG: emit 1

D/TAG: After observeOn(mainThread), current thread is: main

D/TAG: After observeOn(io), current thread is : RxCachedThreadScheduler-2

D/TAG: Observer thread is :RxCachedThreadScheduler-2

D/TAG: onNext: 1
```

可以看到, 每调用一次 `observeOn()` 线程便会切换一次, 因此如果我们有类似的需求时, 便可知道如何处理了.

在RxJava中, 已经内置了很多线程选项供我们选择, 例如有

- Schedulers.io() 代表io操作的线程, 通常用于网络,读写文件等io密集型的操作
- Schedulers.computation() 代表CPU计算密集型的操作, 例如需要大量计算的操作
- Schedulers.newThread() 代表一个常规的新线程
- AndroidSchedulers.mainThread() 代表Android的主线程

这些内置的Scheduler已经足够满足我们开发的需求, 因此我们应该使用内置的这些选项, 在RxJava内部使用的是线程池来维护这些线程, 所有效率也比较高.

实践

对于我们Android开发人员来说, 经常会将一些耗时的操作放在后台, 比如网络请求或者读写文件,操作数据库等等,等到操作完成之后回到主线程去更新UI, 有了上面的这些基础, 那么现在我们就可以轻松的去做到这样一些操作.

下面来举几个常用的场景.

网络请求

Android中有名的网络请求库就那么几个, Retrofit能够从中脱颖而出很大原因就是因为它支持RxJava的方式来调用, 下面简单讲解一下它的基本用法.

要使用Retrofit,先添加Gradle配置:

```
    //retrofit
    compile 'com.squareup.retrofit2:retrofit:2.1.0'
    //Gson converter
    compile 'com.squareup.retrofit2:converter-gson:2.1.0'
    //RxJava2 Adapter
    compile 'com.jakewharton.retrofit:retrofit2-rxjava2-adapter:1.0.0'
    //okhttp
    compile 'com.squareup.okhttp3:okhttp:3.4.1'
    compile 'com.squareup.okhttp3:logging-interceptor:3.4.1'
```

随后定义Api接口:

```
public interface Api {
    @GET
    Observable<LoginResponse> login(@Body LoginRequest request);

    @GET
    Observable<RegisterResponse> register(@Body RegisterRequest request);
}
```

接着创建一个Retrofit客户端:

```
private static Retrofit create() {
            OkHttpClient.Builder builder = new OkHttpClient().newBuilder();
            builder.readTimeout(10, TimeUnit.SECONDS);
            builder.connectTimeout(9, TimeUnit.SECONDS);

            if (BuildConfig.DEBUG) {
                HttpLoggingInterceptor interceptor = new HttpLoggingInterceptor();
                interceptor.setLevel(HttpLoggingInterceptor.Level.BODY);
                builder.addInterceptor(interceptor);
            }

            return new Retrofit.Builder().baseUrl(ENDPOINT)
                    .client(builder.build())
                    .addConverterFactory(GsonConverterFactory.create())
                    .addCallAdapterFactory(RxJava2CallAdapterFactory.create())
                    .build();
}
```

发起请求就很简单了:

```
        Api api = retrofit.create(Api.class);
        api.login(request)
             .subscribeOn(Schedulers.io())               //在IO线程进行网络请求
             .observeOn(AndroidSchedulers.mainThread())  //回到主线程去处理请求结果
            .subscribe(new Observer<LoginResponse>() {
                @Override
                public void onSubscribe(Disposable d) {}

                @Override
                public void onNext(LoginResponse value) {}

                @Override
                public void onError(Throwable e) {
                    Toast.makeText(mContext, "登录失败", Toast.LENGTH_SHORT).show();
                }

                @Override
                public void onComplete() {
                    Toast.makeText(mContext, "登录成功", Toast.LENGTH_SHORT).show();
                }
            });
```

看似很完美, 但我们忽略了一点, 如果在请求的过程中Activity已经退出了, 这个时候如果回到主线程去更新UI, 那么APP肯定就崩溃了, 怎么办呢, 上一节我们说到了 `Disposable` , 说它是个开关, 调用它的 `dispose()` 方法时就会切断水管, 使得下游收不到事件, 既然收不到事件, 那么也就不会再去更新UI了. 因此我们可以在Activity中将这个 `Disposable` 保存起来, 当Activity退出时, 切断它即可.

那如果有多个 `Disposable` 该怎么办呢, RxJava中已经内置了一个容器 `CompositeDisposable` , 每当我们得到一个 `Disposable` 时就调用 `CompositeDisposable.add()` 将它添加到容器中, 在退出的时候, 调用 `CompositeDisposable.clear()` 即可切断所有的水管.

读写数据库

上面说了网络请求的例子, 接下来再看看读写数据库, 读写数据库也算一个耗时的操作, 因此我们也最好放在IO线程里去进行, 这个例子就比较简单, 直接上代码:

```
public Observable<List<Record>> readAllRecords() {
        return Observable.create(new ObservableOnSubscribe<List<Record>>() {
            @Override
            public void subscribe(ObservableEmitter<List<Record>> emitter) throws Excepti
on {
                Cursor cursor = null;
                try {
                    cursor = getReadableDatabase().rawQuery("select * from " + TABLE_NAME
, new String[]{});
                    List<Record> result = new ArrayList<>();
                    while (cursor.moveToNext()) {
                        result.add(Db.Record.read(cursor));
                    }
                    emitter.onNext(result);
                    emitter.onComplete();
                } finally {
                    if (cursor != null) {
                        cursor.close();
                    }
                }
            }
        }).subscribeOn(Schedulers.io()).observeOn(AndroidSchedulers.mainThread());
    }
```

好了本次的教程就到这里吧, 后面的教程将会教大家如何使用RxJava中强大的操作符. 通过使用这些操作符可以很轻松的做到各种吊炸天的效果. 敬请期待.