以下是 **获取天气数据并更新UI项目中 Retrofit + OkHttp + RxJava 的典型用法。**

00.整体流程介绍

- 我们首先需要配置我们使用的组件的倚赖，我们导入Retrofit，gson，okHttp，rxJava。
- 现在我们处理网络请求这一块。
  - 我们知道，Retrofit依赖于Okhttp，本质是借助OkHttp完成的数据发送。所以我们要线创建一个OkHttp的单例客户端，设置它的超时重传时间，和日志拦截器，ok，我们有了OkHttp的单例客户端，就拥有了网络请求发送和接收的工具了。
  - 接下来我们创建创建Retrofit全局唯一单例，我们设置它的网络请求为OkHttp，添加Gson解析器，将JSON响应自动转换为Java对象，设置baseUrl，也就是所有API接口的相对路径将基于此URL，设置支持RxJava3的Observable返回类型，允许链式调用。
  - 现在我们还需要什么？我们需要处理GSON，对吧。我们JSON响应自动转换为Java对象，那么我们需要对应的Java对象。我们考虑我们的一个JSON响应。
    - ```
      {
        "current": {
          "temp_c": 25.0,
          "condition": {
            "text": "Sunny"
          }
        }
      }
      ```
    - 它对应的Java对象应该是?
      - ```
        public class CurrentWeather {
            @SerializedName("temp_c")
            private double temperature;

            @SerializedName("condition")
            private Condition condition;

            public double getTemperature() { return temperature; }
            public Condition getCondition() { return condition; }
        }
        ```
      - 我们需要`@SerializedName`**注解**，将JSON字段名映射到Java字段名。
  - ok，我们有了网络发送和接收的工具，以及接收后自动解析为Java对象的GSON。接下来我们需要去封装一个网络请求的API了。
    - 定义一个wearthApi接口，里面先有一个@GET("current.json")，声明这是一个HTTP GET请求，路径为`current.json`（最终URL：`BASE_URL + current.json`）

    - 内部getCurrentWeather方法，返回Observable，也就是RxJava的Observable对象，用于异步处理网络响应。同时有@Query("key")，将参数 `apiKey` 作为URL查询参数（如 `?key=YOUR_API_KEY`）
    - ```
      public interface WeatherApi {
          @GET("current.json")
          Observable<WeatherResponse> getCurrentWeather(
              @Query("key") String apiKey,
              @Query("q") String city,
              @Query("aqi") String aqi
          );
      }
      ```
    - Retrofit在运行时生成 `WeatherApi` 接口的实现类，自动处理HTTP请求和响应解析。例如：当调用 `getCurrentWeather("123", "Shanghai", "no")` 时，会生成如下请求：
    - ```
      GET https://api.weatherapi.com/v1/current.json?
      key=123&q=Shanghai&aqi=no
      ```
- 我们封装好了我们的网络层次，接下来我们需要去实际调用API获取数据了。
- 接下来我们想一下，我们要获取数据，首先不能在主线程中处理。因此我们需要一个IO线程。那么数据存储呢？我们通过ViewModel+ LiveData.
- 所以我们现在创建一个共享的ViewModel，内部有一个CurrentWeather类型的LiveData。提供一个fetchWeather方法。
    - ```
      public void fetchWeather(String city) {
          WeatherApi api =
      RetrofitClient.getRetrofit().create(WeatherApi.class);
          api.getCurrentWeather("YOUR_API_KEY", city, "no")
              .subscribeOn(Schedulers.io())
              .observeOn(AndroidSchedulers.mainThread())
              .subscribe(new Observer<WeatherResponse>() {
                  @Override
                  public void onNext(WeatherResponse response) {
                      weatherLiveData.setValue(response.getCurrent());
                  }

                  @Override
                  public void onError(Throwable error) {
                      errorLiveData.setValue("Error: " + error.getMessage());
                  }

                  @Override
                  public void onComplete() {
                      // 不需要处理完成事件
                  }
              });
      }
      ```
    - 创建Retrofit接口实例（动态代理）
    - 调用API方法获取 `Observable<WeatherResponse>`
    - 指定网络请求在IO线程执行（`subscribeOn(Schedulers.io())`）
    - 指定结果处理在主线程（`observeOn(AndroidSchedulers.mainThread())`）
      ```

    - 订阅并处理响应/错误（通过LiveData传递结果）
- ok，我们在主线程中调用这个方法，那么结果是什么？我们首先从RxJava的调度器的IO线程池中执行这个任务，去通过网络请求获取数据，并解析为CurrentWeather对象，然后通过RxJava的onNext发送给主线程，主线程在onNext方法中会通过weatherLiveData.setValue(response.getCurrent());更新ViewModel中的数据。
- 又因为我们在主线程中调用observe观察这个weatherLiveData
    - 所以，当数据更新时，ViewModel会遍历自己的观察表，调用观察者的方法，也就是这里的主线程，调用他们的回调方法，也就是这里的设置自身UI数据。
    - ```
      private void observeViewModel() {
              viewModel.getWeatherLiveData().observe(this, currentWeather -> {
                  if (currentWeather != null) {
                      tvTemperature.setText(currentWeather.getTemperature() +
      "°C");

      tvDescription.setText(currentWeather.getCondition().getDescription());
                  }
              });
      ```

01.网络请求流程

- **依赖配置（build.gradle）**
    - ```
      dependencies {
          // Retrofit & OkHttp
          implementation 'com.squareup.retrofit2:retrofit:2.9.0'
          implementation 'com.squareup.retrofit2:converter-gson:2.9.0'
          implementation 'com.squareup.okhttp3:logging-interceptor:4.9.1'

          // RxJava
          implementation 'io.reactivex.rxjava3:rxandroid:3.0.0'
          implementation 'io.reactivex.rxjava3:rxjava:3.0.0'
          implementation 'com.squareup.retrofit2:adapter-rxjava3:2.9.0'
      }
      ```
- 网络层封装（核心代码）
    - 创建带日志拦截器的OkHttpClient
        - ```
          public class NetworkClient {
              private static final int TIMEOUT = 30;

              public static OkHttpClient getOkHttpClient() {
                  return new OkHttpClient.Builder()
                      .connectTimeout(TIMEOUT, TimeUnit.SECONDS)
                      .readTimeout(TIMEOUT, TimeUnit.SECONDS)
                      .addInterceptor(new
          HttpLoggingInterceptor().setLevel(Level.BODY))
                      .build();
              }
          }
          ```
          ```

- 创建Retrofit实例
组件	作用
`BASE_URL`	定义所有API请求的基础URL（如 `https://api.weatherapi.com/v1/`）
`Retrofit.Builder()`	构建Retrofit实例的工厂类
`.baseUrl(BASE_URL)`	设置所有API接口的相对路径将基于此URL（如 `current.json` 会拼接为完整URL）
`.client(NetworkClient.getOkHttpClient())`	配置OkHttp客户端实例（负责实际网络请求）
`.addConverterFactory(GsonConverterFactory.create())`	添加Gson解析器，将JSON响应自动转换为Java对象（如 `WeatherResponse`）
`.addCallAdapterFactory(RxJava3CallAdapterFactory.create())`	支持RxJava3的Observable返回类型，允许链式调用
单例模式	全局唯一Retrofit实例，避免重复创建浪费资源
  - ```
    public class RetrofitClient {
        private static final String BASE_URL =
    "https://api.weatherapi.com/v1/";

        private static Retrofit retrofit;

        public static Retrofit getRetrofit() {
            if (retrofit == null) {
                retrofit = new Retrofit.Builder()
                    .baseUrl(BASE_URL)
                    .client(NetworkClient.getOkHttpClient())
                    .addConverterFactory(GsonConverterFactory.create())

    .addCallAdapterFactory(RxJava3CallAdapterFactory.create())
                    .build();
            }
            return retrofit;
        }
    }
    ```
- 定义天气接口API
注解/参数	作用
`@GET("current.json")`	声明这是一个HTTP GET请求，路径为 `current.json`（最终URL：`BASE_URL + current.json`）
`@Query("key")`	将参数 `apiKey` 作为URL查询参数（如 `?key=YOUR_API_KEY`）
`Observable<WeatherResponse>`	返回RxJava的Observable对象，用于异步处理网络响应
    ```

```
public interface WeatherApi {
    @GET("current.json")
    Observable<WeatherResponse> getCurrentWeather(
        @Query("key") String apiKey,
        @Query("q") String city,
        @Query("aqi") String aqi
    );
}
```

- 定义数据模型
  - **作用**：映射API返回的JSON数据顶层结构。
  - **JSON示例**：

```
{
  "current": {
    "temp_c": 25.0,
    "condition": {
      "text": "Sunny"
    }
  }
}
```

```
public class CurrentWeather {
    @SerializedName("temp_c")
    private double temperature;

    @SerializedName("condition")
    private Condition condition;

    public double getTemperature() { return temperature; }
    public Condition getCondition() { return condition; }
}
```

  - **作用**：映射 `current` 对象，包含温度和天气状况。
  - **字段映射**：
    - `temp_c` → `temperature`
    - `condition` → `condition`（嵌套对象）

```
public class WeatherResponse {
    @SerializedName("current")
    private CurrentWeather current;

    // Getters
    public CurrentWeather getCurrent() {
        return current;
    }
}

public class CurrentWeather {
    @SerializedName("temp_c")
    private double temperature;
```

```
    @SerializedName("condition")
    private Condition condition;

    // Getters
    public double getTemperature() {
        return temperature;
    }

    public Condition getCondition() {
        return condition;
    }
}

public class Condition {
    @SerializedName("text")
    private String description;

    // Getter
    public String getDescription() {
        return description;
    }
}
```

02.ViewModel中发起请求（MVVM架构）

- 成员变量
  - ```
    private final CompositeDisposable disposables = new
    CompositeDisposable();
    private final MutableLiveData<CurrentWeather> weatherLiveData = new
    MutableLiveData<>();
    private final MutableLiveData<String> errorLiveData = new
    MutableLiveData<>();
    ```
变量	作用
`CompositeDisposable`	统一管理RxJava订阅，防止内存泄漏
`MutableLiveData<CurrentWeather>`	存储天气数据，通过LiveData实现观察者模式
`MutableLiveData<String>`	存储错误信息，实现错误信息的观察与处理
- fetchWeather()方法
  - ```
    public void fetchWeather(String city) {
        WeatherApi api =
    RetrofitClient.getRetrofit().create(WeatherApi.class);
        api.getCurrentWeather("YOUR_API_KEY", city, "no")
            .subscribeOn(Schedulers.io())
            .observeOn(AndroidSchedulers.mainThread())
            .subscribe(new Observer<WeatherResponse>() {
                @Override
                public void onNext(WeatherResponse response) {
                    weatherLiveData.setValue(response.getCurrent());
    ```

```
            }

            @Override
            public void onError(Throwable error) {
                errorLiveData.setValue("Error: " + error.getMessage());
            }

            @Override
            public void onComplete() {
                // 不需要处理完成事件
            }
        });
}
```

- 创建Retrofit接口实例（动态代理）
- 调用API方法获取 `Observable<WeatherResponse>`
- 指定网络请求在IO线程执行（`subscribeOn(Schedulers.io())`）
- 指定结果处理在主线程（`observeOn(AndroidSchedulers.mainThread())`）
- 订阅并处理响应/错误（通过LiveData传递结果）
-

```
public class WeatherViewModel extends ViewModel {
    private final CompositeDisposable disposables = new
CompositeDisposable();
    private final MutableLiveData<CurrentWeather> weatherLiveData = new
MutableLiveData<>();
    private final MutableLiveData<String> errorLiveData = new
MutableLiveData<>();

    public void fetchWeather(String city) {
        WeatherApi api =
RetrofitClient.getRetrofit().create(WeatherApi.class);
        api.getCurrentWeather("YOUR_API_KEY", city, "no")
            .subscribeOn(Schedulers.io())
            .observeOn(AndroidSchedulers.mainThread())
            .subscribe(
                response ->
weatherLiveData.setValue(response.getCurrent()),
                error -> errorLiveData.setValue("Error: " +
error.getMessage())
            );
    }

    public LiveData<CurrentWeather> getWeatherLiveData() {
        return weatherLiveData;
    }

    public LiveData<String> getErrorLiveData() {
        return errorLiveData;
    }

    @Override
    protected void onCleared() {
```

```
        disposables.clear(); // 防止内存泄漏
        super.onCleared();
    }
}
```

03.主线程中观察数据并更新UI

- 但是
  - **ViewModelProvider**：通过AndroidX的 `ViewModelProvider` 获取ViewModel实例。
  - 在observeViewModel方法中，通过LiveData的observe监听数据变化，当数据变化时，从LiveData中读取数据，更新UI。
  - 调用viewModel.fetchWeather("Shanghai");开始实际的网络请求过程。
- **观察者模式实现**：
  - `LiveData` 内部维护一个观察者列表。
  - 当调用 `setValue()` 时，遍历所有观察者并调用其 `onChanged()` 方法。
  - `Activity` 中通过 `observe()` 注册的 Lambda 即为观察者的 `onChanged()` 实现。
  - ```
    public class WeatherActivity extends AppCompatActivity {
        private WeatherViewModel viewModel;
        private TextView tvTemperature, tvDescription;

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

            tvTemperature = findViewById(R.id.tv_temperature);
            tvDescription = findViewById(R.id.tv_description);

            viewModel = new
    ViewModelProvider(this).get(WeatherViewModel.class);
            observeViewModel();
            viewModel.fetchWeather("Shanghai");
        }

        private void observeViewModel() {
            viewModel.getWeatherLiveData().observe(this, currentWeather -> {
                if (currentWeather != null) {
                    tvTemperature.setText(currentWeather.getTemperature() +
    "°C");

    tvDescription.setText(currentWeather.getCondition().getDescription());
                }
            });

            viewModel.getErrorLiveData().observe(this, errorMsg -> {
                Toast.makeText(this, errorMsg, Toast.LENGTH_LONG).show();
            });
        }
    }
    ```
    ```

04.关键点说明

- **线程调度**
  - `subscribeOn(Schedulers.io())`：确保网络请求在IO线程池执行
  - `observeOn(AndroidSchedulers.mainThread())`：结果处理在主线程更新UI
- **生命周期管理**
  - 使用 `ViewModel` + `LiveData` 避免内存泄漏
  - `CompositeDisposable` 统一管理订阅
- **错误处理**
  - 分离成功和错误回调，通过 `LiveData` 通知UI层
  - 避免在 `onError` 中直接操作UI组件
- **可维护性**
  - 网络层与业务层分离，便于替换底层实现（如切换为Ktor）
  - 数据模型与API响应严格对应，利用Gson自动解析