30장. 작업 매니저, 생명주기 기반 컴포넌트, 페이징

30.1 작업 매니저

WorkManager는 특정 작업을 유예(Defer, 특정 시점, 특정 상황 에서 작업을 처리) 처리, 비동기 처리를 목적

```
implementation "android.arch.work:work-runtime:1.0.0-beta03"
```

30.1.1. WorkManager 작업 등록 및 실행

- Worker: 작업 내용을 가지는 추상 클래스
- WorkRequest: 작업 의뢰 내용. OneTimeWorkRequest, PeriodicWorkRequest 두 개의 클래스 제공.
- Constraints: WorkRequest의 제약 조건 명시
- WorkManager: WorkRequest를 큐에 등록, 취소하는 역할
- WorkStatus: 작업의 진행 상태

30.1 작업 매니저

Worker 정의

```
public class MyWorker extends Worker {
WorkerParameters workerParams;

public MyWorker(@NonNull Context context, @NonNull WorkerParameters workerParams) {
super(context, workerParams);
this.workerParams = workerParams;
Log.d("kkang", "MyWorker()...." + workerParams.getId() + ",
" + workerParams.getTags());
}
@NonNull
@Override
public ListenableWorker.Result doWork() {
Log.d("kkang", "MyWorker..doWork()....." + workerParams.getId() + ",
" + workerParams.getTags());
return Result.success();
}
}
```

Worker 등록

```
WorkManager workManager = WorkManager.getInstance();
OneTimeWorkRequest workRequest = new OneTimeWorkRequest.Builder(MyWorker.class).build();
workManager.enqueue(workRequest);
```

30.1 작업 매니저

30.1.2. Worker 상태 파악 및 데이터 전달

Worker 상태 파악
Worker의 작업이 성공한 것인지, 실패한 것인지, 아니면 취소된 것인지 상태 파악
getWorkInfoByIdLiveData( ) 함수에 파악하고 자 하는 Worker의 ID 값을 등록

```
workManager.getWorkInfoByIdLiveData(workRequest.getId())
.observe(this, new Observer<WorkInfo>() {
@Override
public void onChanged(@Nullable WorkInfo workInfo) {
if (workInfo != null && workInfo.getState().isFinished()) {
Log.d("kkang", "workStatus...." + workInfo.getId() + ",
" + workInfo.getState().toString());
}
}
});
```

30.1 작업 매니저

Worker에 데이터 전달

```
OneTimeWorkRequest workRequest = new OneTimeWorkRequest.Builder(MyWorker.class)
.setInputData(new Data.Builder().putString("one", "hello").putInt("two", 100)
.build()).build();
```

```
Data data = workerParams.getInputData();
String one = data.getString("one");
int two = data.getInt("two", 0);
```

Worker의 결과 데이터 획득

```
Data resultData = new Data.Builder()
.putString("result1", one + " world")
.putInt("result2", two + 10)
.build();
return Result.success(resultData);
```

```
@Override
public void onChanged(@Nullable WorkInfo workInfo) {
if (workInfo != null && workInfo.getState().isFinished()) {
Log.d("kkang", "workStatus...." + workInfo.getId() + ",
" + workInfo.getState().toString());
Data result = workInfo.getOutputData();
String result1 = result.getString("result1");
int result2 = result.getInt("result2",0);
}
}
```

30.1 작업 매니저

30.1.3. OneTimeWorkRequest와 PeriodicWorkRequest
작업 의뢰 내용을 담는 클래스는 OneTimeWorkRequest와 PeriodicWorkRequest가 제공
OneTimeWorkRequest는 단일 작업을 위한 의뢰 내용이며 PeriodicWorkRequest는 반복되는 작업을 위한 의뢰 내용

```
OneTimeWorkRequest workRequest = new OneTimeWorkRequest.Builder(MyWorker.class)
.setInitialDelay(1, TimeUnit.MINUTES)
.build();
workManager.enqueue(workRequest);
```

```
PeriodicWorkRequest workRequest = new PeriodicWorkRequest.Builder(MyWorker.class,
15, TimeUnit.MINUTES).build();
workManager.enqueue(workRequest);
```

30.1 작업 매니저

30.1.4. Constraints를 이용한 실행 조건 명시
조건을 명시하기 위한 클래스가 Constraints
Constraints 클래스를 이용해 충전 상태, 베터리가 부족하지 않을 때, 기기가 Idle 상태일 때, 기기의 저 장공간이 부족하지 않을 때 등의 조건

네트워크 조건 명시
Constraints.Builder의 setRequiredNetworkType( ) 함수를 이용해 네트워크 상태를 지정

```
Constraints myConstraints = new Constraints.Builder()
.setRequiredNetworkType(NetworkType.CONNECTED)
.build();
OneTimeWorkRequest workRequest = new OneTimeWorkRequest.Builder(MyWorker.class)
.setConstraints(myConstraints)
.build();
```

베터리 조건 명시
Constraints.Builder의 setRequiredCharging ( ) 함수를 이용하여 베터리 충전 상태 변경에 따라서 작업을 실행

```
Constraints constraints = new Constraints.Builder()
.setRequiresCharging(true)
.build();
```

30.1 작업 매니저

30.1.5. 작업 취소

등록 취소 방법은 Worker 등록 시 부여된 ID 값으로 취소할 수도 있고 개발자가 Request에 명시한 Tag 값으로 취소할 수도 있다.

```
UUID id = workRequest.getId();
workManager.cancelWorkById(id);
```

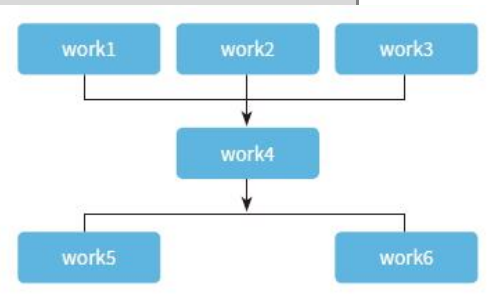
```
OneTimeWorkRequest workRequest = new OneTimeWorkRequest.Builder(MyWorker.class)
.setInitialDelay(1, TimeUnit.MINUTES)
.addTag("myJob")
.build();
workManager.enqueue(workRequest);
workManager.cancelAllWorkByTag("myJob");
```

30.1 작업 매니저

30.1.6. 작업 체이닝
각각의 작업에 순서 명시 가능

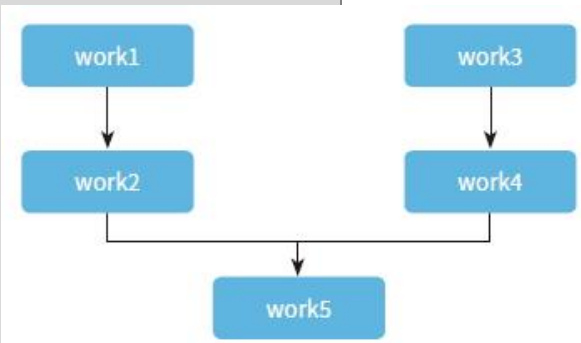
```
workManager.beginWith(request1)
.then(request2)
.enqueue();
```

```
List<OneTimeWorkRequest> beginRequest = new ArrayList();
beginRequest.add(request1);
beginRequest.add(request2);
List<OneTimeWorkRequest> lastRequest = new ArrayList();
lastRequest.add(request4);
lastRequest.add(request5);
workManager.beginWith(beginRequest)
.then(request3)
.then(lastRequest)
.enqueue();
```



30.1 작업 매니저

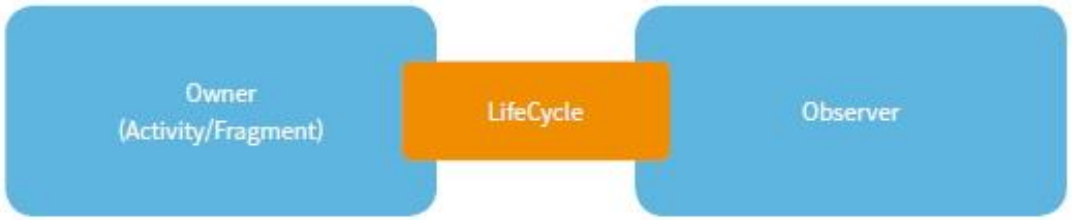
```
WorkContinuation chain1 = WorkManager.getInstance()
.beginWith(request1)
.then(request2);
WorkContinuation chain2 = WorkManager.getInstance()
.beginWith(request3)
.then(request4);
List<WorkContinuation> chainList = new ArrayList<>();
chainList.add(chain1);
chainList.add(chain2);
WorkContinuation chain3 = WorkContinuation
.combine(chainList)
.then(request5);
chain3.enqueue();
```



30.2 생명주기 기반 컴포넌트

30.2.1. 생명주기 기반 컴포넌트란?
생명주기 기반 컴포넌트(Lifecycle-aware Components)란, 액티비티의 생명주기가 복잡한 경우 이 를 처리하는 LifeCycleObserver를 따로 구성하자는 개념

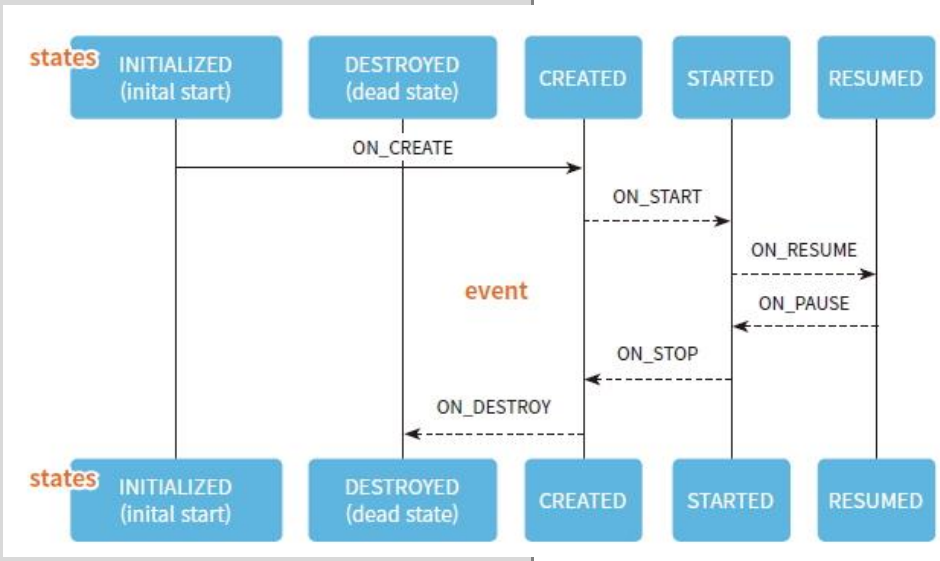
```
implementation "android.arch.lifecycle:extensions:1.1.1"
```



LifeCycle 객체에 Observer를 등록해 놓으면 액티비티나 프래그먼트 등 의 생명주기 변경 시 LifeCycle 객체가 등록된 Observer를 실행하는 구조

30.2 생명주기 기반 컴포넌트

```
public class MyLifecycleObserver implements LifecycleObserver {
@OnLifecycleEvent(Lifecycle.Event.ON_CREATE)
public void onCreate() {
Log.d("kkang", "MyLifecycleObserver....onCreate....");
}
@OnLifecycleEvent(Lifecycle.Event.ON_RESUME)
public void onResume() {
Log.d("kkang", "MyLifecycleObserver....onResume....");
}
@OnLifecycleEvent(Lifecycle.Event.ON_STOP)
public void onStop() {
Log.d("kkang", "MyLifecycleObserver....onStop....");
}
@OnLifecycleEvent(Lifecycle.Event.ON_DESTROY)
public void onDestory() {
Log.d("kkang", "MyLifecycleObserver....onDestory....");
}
}
```




```
getLifecycle().addObserver(new MyLifecycleObserver());
```

30.2 생명주기 기반 컴포넌트

30.2.2. LifecycleOwner

LifecycleOwner는 생명주기가 변경되는 클래스 자체를 지칭하며 일반적으로 액티비티, 프 래그먼트 객체
LifecycleOwner는 getLifecycle ( )이라는 추상 함수 하나를 가지는 LifecycleOwner 인터페이스를 구현한 클래스를 지칭
안드로이드 앱 개발 시 이용되는 Fragment, AppCompatActivity 클래스는 이미 이 인터페이스가 구현된 클래스

```
public class Main3Activity extends Activity implements LifecycleOwner {
private LifecycleRegistry lifecycleRegistry = new LifecycleRegistry(this);
@Override
protected void onCreate(Bundle savedInstanceState) {
super.onCreate(savedInstanceState);
lifecycleRegistry.addObserver(new MyLifecycleObserver());
}

@NonNull
@Override
public Lifecycle getLifecycle() {
return lifecycleRegistry;
}
}
```

30.2 생명주기 기반 컴포넌트

ProcessLifecycleOwner
LifecycleOwner는 개별 액티비티, 프래그먼트의 생명주기를 감지하기 위해 사용.
애 플리케이션을 다시 시작하거나(resume) 멈추는(pause) 상황을 감지해야 할 때 ProcessLifecycleOwner를 이용

```
ProcessLifecycleOwner.get().getLifecycle().addObserver(new MyLifecycleObserver3());
```

30.2 생명주기 기반 컴포넌트

30.2.3. LifecycleObserver : 라이브데이터 이용
LifecycleObserver의 작업에 의한 데이터를 액티비티 등에서 이용해야 할 때, 라이 브데이터(LiveData)를 이용

```
public class MyLifecycleObserver4 extends LiveData<String> implements LifecycleObserver {
@OnLifecycleEvent(Lifecycle.Event.ON_CREATE)
public void onCreate() {
setValue("observer oncreate data");
}
@OnLifecycleEvent(Lifecycle.Event.ON_RESUME)
public void onResume() {
setValue("observer onResume data");
}
}
```

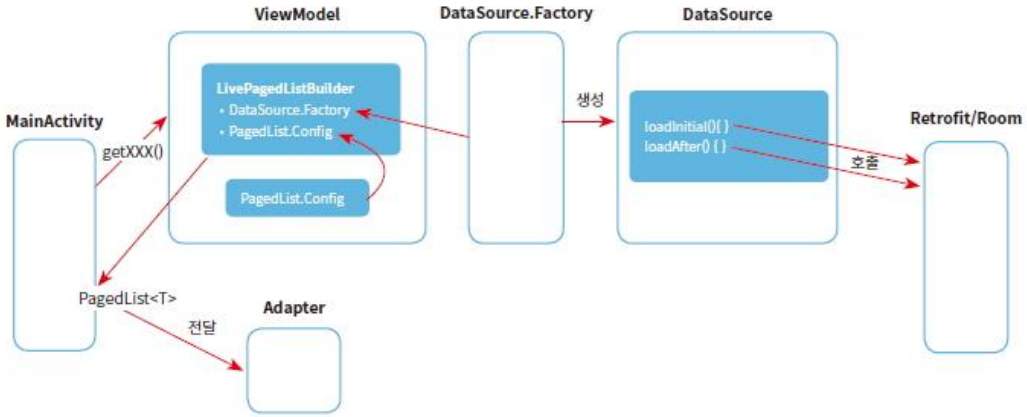
```
public class Test4Activity extends AppCompatActivity {
MyLifecycleObserver4 liveData;
@Override
protected void onCreate(Bundle savedInstanceState) {
super.onCreate(savedInstanceState);
setContentView(R.layout.activity_test4);

liveData = new MyLifecycleObserver4();
liveData.observe(this, result -> {
Log.d("kkang", "activity...." + result);
});
getLifecycle().addObserver(liveData);
}
}
```

30.3 페이징

30.3.1. 페이징이란?
페이징(Paging)이란, RecyclerView에서 페이징 처리를 조금 더 쉽고 효율적으로 작성하게 하기 위한 라이브러리

```
implementation 'android.arch.paging:runtime:1.0.0'
```



30.3 페이징

DataSource
RecyclerView를 위한 데이터 획득 역할.
데이터는 Retrofit을 이용한 네트워크 데이터일 수 도 있고, 룸(Room)을 이용한 데이터베이스 데이터일 수도 있다.

DataSource.Factory
이름 그대로 DataSource를 생성해주는 역할.
DataSource를 앱 내에서 여러 개 가지고 있고 DataSource.Factory에서 그 순간 이용하고자 하는 DataSource를 생성 이용

PageListBuilder
페이징을 위한 데이터를 획득하는 곳.
PageListBuilder에게 DataSource.Factory와 PagedList.Config를 알려주면 PagedList.Config에 설정된 내용대로 DataSource.Factory를 이용해 데이터를 획득

PagedList<T>
PageListBuilder에 의해 획득한 결과 데이터.
PagedList를 RecyclerView의 Adapter에 전 달하면 화면에 출력.

30.3 페이징

30.3.2. DataSource
실제 데이터를 획득하는 역할을 합니다. 데이터는 네트워크 데이터일 수도 있고 데이터베이스 데 이터일 수도 있다.
DataSource는 추상 클래스이며 이를 구현한 PageKeyedDataSource, ItemKeyedDataSource, PositionalDataSoruce 중 하나를 상속받아 작성.

```
public class MyDataSource extends PageKeyedDataSource<Long, Item> {
…
//초기 작업 딱 한 번 호출..
@Override
public void loadInitial(@NonNull LoadInitialParams<Long> params,
@NonNull LoadInitialCallback<Long, Item> callback) {
…
callback.onResult(response.body().articles, null, 2L);
…
}
@Override
public void loadBefore(@NonNull LoadParams<Long> params,
@NonNull LoadCallback<Long, Item> callback) {
}
@Override
public void loadAfter(@NonNull LoadParams<Long> params,
@NonNull LoadCallback<Long, Item> callback) {
…
callback.onResult(response.body().articles, nextKey);
}
}
```

30.3 페이징

- ItemKeyedDataSource<Key, Value>: 페이지의 항목 정보를 이용해서 다음 페이지에 대한 정보를 결정
- PageKeyedDataSource<Key, Value>: 페이지 번호에 대한 정보를 유지해 그다음 페이지 번호를 증감시켜 결정
- PositionalDataSoruce<T>: 어디부터 어디까지의 데이터를 고정시켜 획득하는 경우

```
public class TestMyItemDataSource extends ItemKeyedDataSource<Long, Item> {
@Override
public void loadAfter(@NonNull LoadParams<Long> params,
@NonNull LoadCallback<Item> callback) {
callback.onResult(response.body().articles);
}

@NonNull
@Override
public Long getKey(@NonNull Item item) {
Log.i(TAG, "getKey " + item.id);
return item.id;
}
}
```

30.3 페이징

30.3.3. DataSource.Factory & PageListBuilder
DataSource. Factory는 이름 그대로 DataSource 객체를 만들어 주는 역할

```
public class MyDataFactory extends DataSource.Factory {
private MyDataSource dataSource;
…

@Override
public DataSource create() {
dataSource = new MyDataSource(application);
return dataSource;
}
}
```

PageListBuilder에 DataSource.Factory 를 등록

```
public class MyViewModel extends ViewModel {
…
MyDataFactory dataFactory = new MyDataFactory(application);
PagedList.Config pagedListConfig = (new PagedList.Config.Builder())
.setInitialLoadSizeHint(3)
.setPageSize(5).build();
itemLiveData = (new LivePagedListBuilder(dataFactory, pagedListConfig))
.build();
}
```

30.3 페이징

30.3.4. PagedListAdapter
RecyclerView의 Adapter에 전달해 출력.
페이징을 이용하기 위한 Adapter는 RecyclerView. Adapter를 상속받아 작성하지 않고 PagedListAdapter를 상속받아 작성

```
public class MyListAdapter extends PagedListAdapter<Item, RecyclerView.ViewHolder> {
public static DiffUtil.ItemCallback<Item> DIFF_CALLBACK =
new DiffUtil.ItemCallback<Item>() {
@Override
public boolean areItemsTheSame(@NonNull Item oldItem, @NonNull Item newItem) {
return oldItem.id == newItem.id;
}

@Override
public boolean areContentsTheSame(@NonNull Item oldItem, @NonNull Item newItem) {
return oldItem.equals(newItem);
}
};
public MyListAdapter(Context context) {
super(DIFF_CALLBACK);
}
@Override
public void onBindViewHolder(@NonNull RecyclerView.ViewHolder holder, int position) {
// PagedListAdapter에서 제공하는 함수.. T getItem(int position)
Item article = getItem(position);
...
}
}
```

Step by Step 실습 – 30-1. 페이징

- 그레이들 작업
- AndroidManifest.xml
- ArticleDAO.java
- MyDataSource.java
- MyDataFactory.java
- MyViewModel.java
- MyListAdapter.java