Night Piece

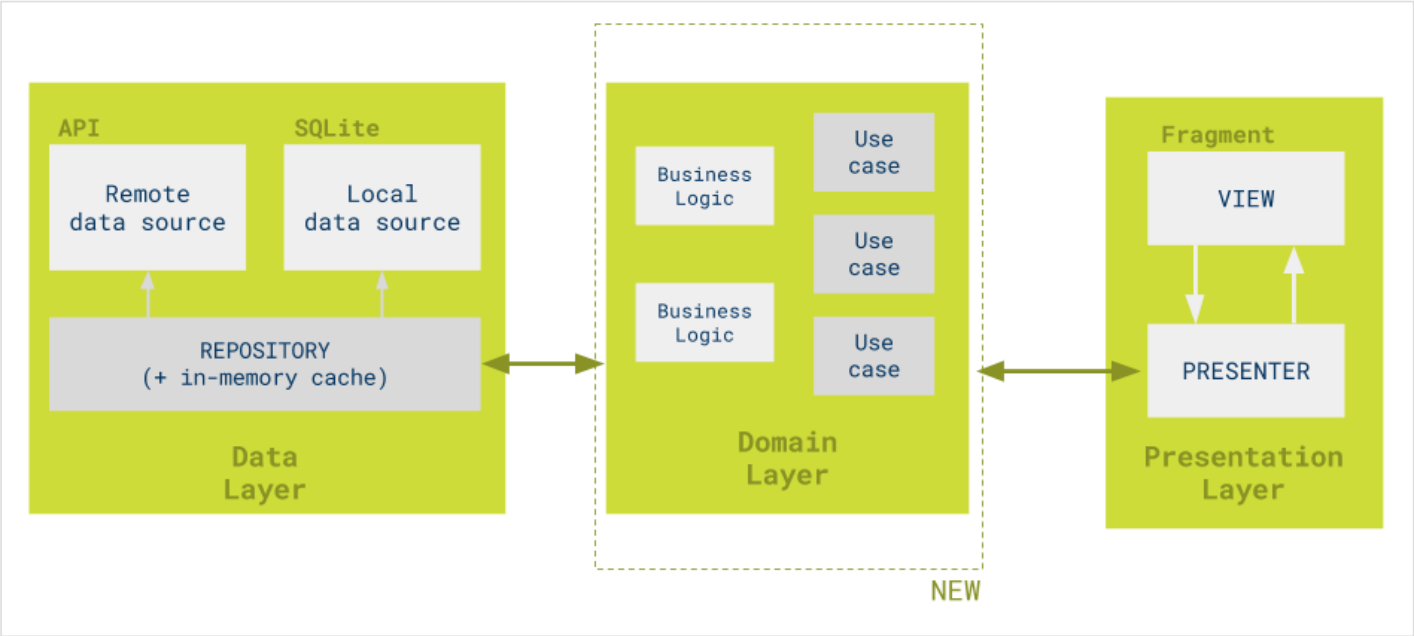
Android构架系列之二--MVP&&Clean理解与实践之实例分析

2016-05-15 | 技术

前面我们分析了MVP与Clean，本文试图以Google构架Demo的Clean分支为样本来分析一下具体的代码实现。由于Clean包含了MVP部分，所以MVP的部分一并说明。

需要强调的是这并不是Clean构架的唯一实现方式，但是其思想可以借鉴。

总体结构



分为三部分：

- 展现(Presentation)层: 核心是**MVP**，做UI控制。
- 领域(Domain)层: 核心是**UseCase** 这一层是所有的业务逻辑，这一层的类都叫做 `xxxUseCase` 或者 `xxxInteractor`（在这个Demo中都是UseCase的子类，命名都是以业务相关的动名词的形式，如GetTasks），代表了在Presentation层开发者可以执行的所有Action。
- 数据(Data)层: 核心是**Repository**，是使用数据仓库模式。

展现(Presentation)层–MVP

由以下几部分组成

1. Activity: **组合View(Fragemnt)与Presenter**，Activity不是View！Activity的 `OnCreate` 中完成3件事情。

- 构建View，这里都是Fragment。
- **生成所有Presenter用到的的UseCase**，UseCase用的UseCaseHandler,Repository：目的是方便修改注入，用Provider的方式代替注入框架，全部在Activity中注入完成，如果使用Dagger等注入框架，这里不必要。

> 请对比学习

- 生成Presenter并**双向绑定**：注意参数：注入刚才的View，和用到的所有UserCase.

- **Presenter的状态恢复**，在Activity重建时，都是重新构建Presenter，并且只恢复Presenter中某些数据的状态。（这一步可选，只恢复使用的数据，大部分情况下并没有恢复数据，重新构建Presenter。这里的实现简单粗暴，也可以用Fragment来保持Presenter，关于Presenter的恢复问题MVP一节中有讨论）

Activity的OnCreate中代码如下

```
// 生成View
        TasksFragment tasksFragment =
                (TasksFragment) getSupportFragmentManager().findFragmentById(R.id.contentFrame);
        if (tasksFragment == null) {
            // Create the fragment
            tasksFragment = TasksFragment.newInstance();
            ActivityUtils.addFragmentToActivity(
                    getSupportFragmentManager(), tasksFragment, R.id.contentFrame);
        }
```

```
// 生成Presenter，注意参数传入了上面生成的View和用到的UseCase
// 注意：在Presenter的构造函数内部会调用View的setPresenter实现双向绑定
   mTasksPresenter = new TasksPresenter(
                Injection.provideUseCaseHandler(),
                tasksFragment,
                Injection.provideGetTasks(getApplicationContext()),
                Injection.provideCompleteTasks(getApplicationContext()),
                Injection.provideActivateTask(getApplicationContext()),
                Injection.provideClearCompleteTasks(getApplicationContext())
                );
```

```
// Presenter状态恢复
        if (savedInstanceState != null) {
            TasksFilterType currentFiltering =
                    (TasksFilterType) savedInstanceState.getSerializable(CURRENT_FILTERING_KEY);
            mTasksPresenter.setFiltering(currentFiltering);
        }
```

2. Fragment：代表View，与其他的View作用相同

```
public class TasksFragment extends Fragment implements TasksContract.View {
    public TasksFragment() {
        // Requires empty public constructor
    }

```

```
6       public static TasksFragment newInstance() {
7           // 构建Fragment的最佳实践，可以setArgument等
8           return new TasksFragment();
9       }
10
11      @Override
12      public void onCreate(@Nullable Bundle savedInstanceState) {
13          super.onCreate(savedInstanceState);
14          mListAdapter = new TasksAdapter(new ArrayList<Task>(0), mItemListener);
15      }
16
17      @Override
18      public void onResume() {
19          super.onResume();
20          // Presenter一般都会实现以下通用的方法
21          mPresenter.start();
22      }
23
24          // 双向绑定时，给Presenter使用的
25      @Override
26      public void setPresenter(@NonNull TasksContract.Presenter presenter) {
27          mPresenter = checkNotNull(presenter);
28      }
29
30      @Override
31      public void onActivityResult(int requestCode, int resultCode, Intent data) {
32           // 一些回调交给Presenter处理
33          mPresenter.result(requestCode, resultCode);
34      }
35      @Nullable
36      @Override
37      public View onCreateView(LayoutInflater inflater, ViewGroup container,
38                               Bundle savedInstanceState) {
39          View root = inflater.inflate(R.layout.addtask_frag, container, false);
40          // 这个看情况，界面中有无需要保持的数据（如一些用户输入的信息）。
41          // 由于这里没有使用Fragemnt来保持Presenter，这个也可以不加
42          // setRetainInstance(true);
43          return root;
44      }
45
46      // 其他的View接口的方法实现，给Presenter使用
47      @Override
48      public void showTasksList() {
49          getActivity().setResult(Activity.RESULT_OK);
50          getActivity().finish();
51      }
52  }
```

从上述代码中，我们可以得到几点信息：

- 在View的生命周期中调用对应的Presenter方法。
- **View与Presenter的绑定时机：**这里的View(Fragment)比较被动，通过在Presenter的构造函数中调用View接口的setPresnter方法注入Presenter，实现双向绑定。
- Fragment没有履行Presenter保持的职责，他只负责保持界面的数据（如果有必要，参考 `AddEditTaskFragment.java` ）。

> 之所以这样，一部分原因是由Activity来管理数据恢复这些事情，职责清晰。

3. Presenter类

特点如下

- 实现了xxxContract.Presenter接口，包括该接口的父接口BasePresenter中定义的生命周期映射（只有 `void start()` 方法一般在View的 `onResume()` 中调用）。
- 暴露了的接口要明确。大部分暴露的接口都是View使用的操作（由用户行为触发）与Activity用到的功能（数据保持恢复型操作）。**如何定义，定义什么接口具体查看Contract**
- 构造函数中与Fragment绑定，setPresenter
- 一个Presenter中**含有多个UseCase**
- 一个对外接口可以单独运行一个UseCase或者**组合运行多个UseCase，嵌套调用。**
- 可能有 `public void result(int requestCode, int resultCode)` 接口，映射了Fragment(不是Activity)的onActivityResult方法，处理回调。
- 额外的还有数据获取与恢复接口给Activity调用
- 接口中对View传来的**原始数据**进行处理。如判空等，在Presenter中，如果是null，直接调用View告知用户。而不是把这些值向下传入Domain层。**原则：异常输入越早处理约好**

```
1          public class TaskDetailPresenter implements TaskDetailContract.Presenter {
2
3      private final TaskDetailContract.View mTaskDetailView;
4      private final UseCaseHandler mUseCaseHandler;
5      // 含有多个UseCase
6      private final GetTask mGetTask;
7      private final CompleteTask mCompleteTask;
8      private final ActivateTask mActivateTask;
9      private final DeleteTask mDeleteTask;
10
11     @Nullable
12     private String mTaskId;
13
14     public TaskDetailPresenter(@NonNull UseCaseHandler useCaseHandler,
15             @Nullable String taskId,
16             @NonNull TaskDetailContract.View taskDetailView,
17             @NonNull GetTask getTask,
18             @NonNull CompleteTask completeTask,
19             @NonNull ActivateTask activateTask,
20             @NonNull DeleteTask deleteTask) {
21         mTaskId = taskId;
22         // 这些判空也是尽早发现问题的思想
23         mUseCaseHandler = checkNotNull(useCaseHandler, "useCaseHandler cannot be null!");
24         mTaskDetailView = checkNotNull(taskDetailView, "taskDetailView cannot be null!");
25         mGetTask = checkNotNull(getTask, "getTask cannot be null!");
26         mCompleteTask = checkNotNull(completeTask, "completeTask cannot be null!");
27         mActivateTask = checkNotNull(activateTask, "activateTask cannot be null!");
28         mDeleteTask = checkNotNull(deleteTask, "deleteTask cannot be null!");
29         mTaskDetailView.setPresenter(this);
30     }
31
32     // 抽象了一下，几乎所有的Presenter都有启动的那一刻，启动后可能是获取数据（绝大多数），或者其他操作。

33     @Override
34     public void start() {
35         openTask();
36     }
```

```
37      // 这个很有意思，把Fragment的onActivityResult的值直接传递到Presenter中处理
38      @Override
39      public void result(int requestCode, int resultCode) {
40          // If a task was successfully added, show snackbar
41          if (AddEditTaskActivity.REQUEST_ADD_TASK == requestCode
42                  && Activity.RESULT_OK == resultCode) {
43              mTasksView.showSuccessfullySavedMessage();
44          }
45      }
46
47      private void openTask() {
48      // 这里是输入的异常处理，越早越好，不要向下传再抛回来
49          if (mTaskId == null || mTaskId.isEmpty()) {
50              mTaskDetailView.showMissingTask();
51              return;
52          }
53
54          mTaskDetailView.setLoadingIndicator(true);
55
56          mUseCaseHandler.execute(mGetTask, new GetTask.RequestValues(mTaskId),
57                  new UseCase.UseCaseCallback<GetTask.ResponseValue>() {
58                      @Override
59                      public void onSuccess(GetTask.ResponseValue response) {
60                          Task task = response.getTask();
61
62                          // The view may not be able to handle UI updates anymore
63                          if (!mTaskDetailView.isActive()) {
64                              return;
65                          }
66                          mTaskDetailView.setLoadingIndicator(false);
67                          if (null == task) {
68                              mTaskDetailView.showMissingTask();
69                          } else {
70                              showTask(task);
71                          }
72                      }
73
74                      @Override
75                      public void onError() {
76                          // The view may not be able to handle UI updates anymore
77                          if (!mTaskDetailView.isActive()) {
78                              return;
79                          }
80                          mTaskDetailView.showMissingTask();
81                      }
82                  });
83      }
84
85      // 这些暴露的接口都是以用户动作触发为单位的！
86      @Override
87      public void editTask() {
88      // 这里是输入的异常处理，越早越好，不要向下传再抛回来
89          if (mTaskId == null || mTaskId.isEmpty()) {
90              mTaskDetailView.showMissingTask();
91              return;
92          }
93          mTaskDetailView.showEditTask(mTaskId);
94      }
95
```

```
96      @Override
97      public void deleteTask() {
98          mUseCaseHandler.execute(mDeleteTask, new DeleteTask.RequestValues(mTaskId),
99                  new UseCase.UseCaseCallback<DeleteTask.ResponseValue>() {
100                     @Override
101                     public void onSuccess(DeleteTask.ResponseValue response) {
102                         mTaskDetailView.showTaskDeleted();
103                     }
104
105                     @Override
106                     public void onError() {
107                         // Show error, log, etc.
108                     }
109                 });
110     }
111 // 这些暴露的接口都是以用户动作触发为单位的！
112     @Override
113     public void completeTask() {
114         if (mTaskId == null || mTaskId.isEmpty()) {
115             mTaskDetailView.showMissingTask();
116             return;
117         }
118
119         mUseCaseHandler.execute(mCompleteTask, new CompleteTask.RequestValues(mTaskId),
120                 new UseCase.UseCaseCallback<CompleteTask.ResponseValue>() {
121                     @Override
122                     public void onSuccess(CompleteTask.ResponseValue response) {
123                         mTaskDetailView.showTaskMarkedComplete();
124                     }
125
126                     @Override
127                     public void onError() {
128                         // Show error, log, etc.
129                     }
130                 });
131     }
132 // 这些暴露的接口都是以用户动作触发为单位的！
133     @Override
134     public void activateTask() {
135         if (mTaskId == null || mTaskId.isEmpty()) {
136             mTaskDetailView.showMissingTask();
137             return;
138         }
139         mUseCaseHandler.execute(mActivateTask, new ActivateTask.RequestValues(mTaskId),
140                 new UseCase.UseCaseCallback<ActivateTask.ResponseValue>() {
141                     @Override
142                     public void onSuccess(ActivateTask.ResponseValue response) {
143                         mTaskDetailView.showTaskMarkedActive();
144                     }
145
146                     @Override
147                     public void onError() {
148                         // Show error, log, etc.
149                     }
150                 });

151     }
152
153     private void showTask(Task task) {
154         String title = task.getTitle();
```

```
155          String description = task.getDescription();
156
157          if (title != null && title.isEmpty()) {
158              mTaskDetailView.hideTitle();
159          } else {
160              mTaskDetailView.showTitle(title);
161          }
162
163          if (description != null && description.isEmpty()) {
164              mTaskDetailView.hideDescription();
165          } else {
166              mTaskDetailView.showDescription(description);
167          }
168          mTaskDetailView.showCompletionStatus(task.isCompleted());
169      }
170
171      // 这两个方法比较特别，是Avtivity保存与恢复数据使用的，不是用户操作
172      @Override
173      public void setFiltering(TasksFilterType requestType) {
174          mCurrentFiltering = requestType;
175      }
176
177      @Override
178      public TasksFilterType getFiltering() {
179          return mCurrentFiltering;
180      }
181  }
```

4. Contract–接口定义

   这个类是demo的特色，把一个业务的**展现层与领域层之间的接口**归类到一个类中十分清晰

   - View层的操作（往往由用户触发）
     - 编辑
     - 添加
     - 删除
     - 点击
     - 下拉。。。
   - View的生命周期映射、抽象
     - onResume – `void start()`
     - onPause
     - onDestroy。。。
     - **`void result(int requestCode, int resultCode);`**
   - 数据存储恢复（这个demo是Activity使用）
     - onSaveInstance – `void setFiltering(TasksFilterType requestType);`
     - onRestoreInstance – `TasksFilterType getFiltering();`

```
1  public interface AddEditTaskContract {
2  // view层接口,从extends BaseView<Presenter> 就看出来依赖
3      interface View extends BaseView<Presenter> {
4
5          void showEmptyTaskError();
```

```
        void showTasksList();

        void setTitle(String title);

        void setDescription(String description);

        boolean isActive();
    }
// presenter接口
    interface Presenter extends BasePresenter {

        void saveTask(String title, String description);

        void populateTask();
    }
}
```

领域(Domain)层–UseCase

调用领域层的代码都是在展现层的Presenter类中。

UseCase的外部特点：

- 独立性，可复用，一个业务定义的UseCase可以被其他业务单独使用

> 实例： `TaskDetailPresenter` 与 `TasksPresenter` 都使用了 `CompleteTask`

- 命名直观，表示其功能
- **一个UseCase外而言只执行一个任务**，既一个request一个reponse，没有多个方法暴露
- Presentation层的调用者使用命令模式执行UseCase
  - 单独运行一个UseCase
  - 组合运行多个UseCase：嵌套调用
- 使用命令模式 一个执行器参考 `UseCaseHandler` ， 参数是UseCase（命令），Request（输入参数）与Response（输出结果）。 **UseCaseHandler也是在Activty中构造传入Presenter的。**
- **注意传参的方式**，Request与Response都是定义在UseCase中的内部类，用它们来包裹传递的值，不是使用 `new xxxUseCase(param1,param2).execute(callback)` 的样式，或者 `new xxxUseCase().execute(param1,param2,callback)`

实例代码如下

```
public void clearCompletedTasks() {
        mUseCaseHandler.execute(mClearCompleteTasks, new ClearCompleteTasks.RequestValues(),
                new UseCase.UseCaseCallback<ClearCompleteTasks.ResponseValue>() {
                    @Override
                    public void onSuccess(ClearCompleteTasks.ResponseValue response) {
                        mTasksView.showCompletedTasksCleared();
                        loadTasks(false, false);
                    }

                    @Override
                    public void onError() {
```

```
                        mTasksView.showLoadingTasksError();
                    }
                });
    }
```

UseCase的内部实现

- **UseCase内部没有调用其他UseCase，组合由Presenter完成**，UseCase之间不可以互相调用？？？

> demo中是这样的，实际开发中有这个需求吗？还是合理划分UseCase就可以了？，尤其是一个UseCase只有执行一个execute，如果一个复杂的UseCase有多个可以复用的任务组成，难道逻辑放到Presenter中？虽然理论上移动端不应该有如此复杂的业务逻辑。展示逻辑（如分页）在Presenter中没有问题。

- UseCase内部的 `executeUseCase()` 覆写，实现真正的业务逻辑。
- 内部类定义Request与Reponse，包裹传递的实体。
- **没有在UseCase内的变量缓存数据**
- 执行器executeUseCase默认在在非UI线程执行UseCase，但是CallBack会回到UI线程，参考 `UseCaseHandler.java`

```
// UseCase泛型参数就是命令模式的几个参数
public class GetTasks extends UseCase<GetTasks.RequestValues, GetTasks.ResponseValue> {

        // 注意：无变量缓存
    private final TasksRepository mTasksRepository;

    private final FilterFactory mFilterFactory;


    public GetTasks(@NonNull TasksRepository tasksRepository, @NonNull FilterFactory filterFactory) {
        mTasksRepository = checkNotNull(tasksRepository, "tasksRepository cannot be null!");
        mFilterFactory = checkNotNull(filterFactory, "filterFactory cannot be null!");
    }

    @Override
    protected void executeUseCase(final RequestValues values) {
        if (values.isForceUpdate()) {
            mTasksRepository.refreshTasks();
        }

        mTasksRepository.getTasks(new TasksDataSource.LoadTasksCallback() {
            @Override
            public void onTasksLoaded(List<Task> tasks) {
            // 纯的业务逻辑，每一次都从数据仓库重新获取过滤
                TasksFilterType currentFiltering = values.getCurrentFiltering();
                TaskFilter taskFilter = mFilterFactory.create(currentFiltering);

                List<Task> tasksFiltered = taskFilter.filter(tasks);
                ResponseValue responseValue = new ResponseValue(tasksFiltered);
                // 这种通知方式getUseCaseCallback的被封装了
                getUseCaseCallback().onSuccess(responseValue);
            }

            @Override
            public void onDataNotAvailable() {
             // 这种通知方式getUseCaseCallback的被封装了
                getUseCaseCallback().onError();
```

```
            }
        });

    }
        // 注意这两个类UseCase.RequestValues与UseCase.ResponseValue是空的接口，子类设计也是比较自由的
    public static final class RequestValues implements UseCase.RequestValues {

        private final TasksFilterType mCurrentFiltering;
        private final boolean mForceUpdate;

        public RequestValues(boolean forceUpdate, @NonNull TasksFilterType currentFiltering) {
            mForceUpdate = forceUpdate;
            mCurrentFiltering = checkNotNull(currentFiltering, "currentFiltering cannot be null!");
        }

        public boolean isForceUpdate() {
            return mForceUpdate;
        }

        public TasksFilterType getCurrentFiltering() {
            return mCurrentFiltering;
        }
    }

    public static final class ResponseValue implements UseCase.ResponseValue {

        private final List<Task> mTasks;

        public ResponseValue(@NonNull List<Task> tasks) {
            mTasks = checkNotNull(tasks, "tasks cannot be null!");
        }

        public List<Task> getTasks() {
            return mTasks;
        }
    }
}
```

数据(Data)层–Repository模式

领域层从数据仓库获取接口，

Repository的外部特点

- 领域层直接持有数据层的类 `TasksRepository` 而非 `TasksDataSource` 接口。

> 虽然持有TasksRepository，不影响测试（本质上它就是个门面，如果测试在注入时替换内部Source就行，参考下面代码），但是很奇怪。我觉得持有TasksDataSource没有问题，可能是TasksRepository语意更清晰。

- 单例设计，很好理解
- **有同步方法，也有异步方法**。但是没必要用异步的，同步即可。TasksDataSource中有一些异步的Callback接口，README中都说了没有必要。。。

- 接口中的方法定义与存储的数据相关，如添加一个todo任务，删除一个todo任务，获取所有的todo任务

Repository的内部实现

- **内部有缓存**，单仅仅是原始数据缓存，使用HashMap实现，比较简单。
- Repository模式类似与装饰模式， TasksRepository 暴露的接口只负责获取到数据，而不论数据的来源是哪里（可能是内存，网络，数据库）
- TasksRepository 的内部设计会引用多个来源TasksDataSource，他们也都实现了 TasksRepository 接口。如果需要测试，直接用fake的TasksDataSource替代真实的source即可。

```
// 注意接口设计TasksDataSource与下面mTasksRemoteDataSource等相同
public class TasksRepository implements TasksDataSource {

    private static TasksRepository INSTANCE = null;

    private final TasksDataSource mTasksRemoteDataSource;

    private final TasksDataSource mTasksLocalDataSource;

    // 缓存
    Map<String, Task> mCachedTasks;
        // 缓存 数据脏了
    boolean mCacheIsDirty = false;

    // Prevent direct instantiation.
    private TasksRepository(@NonNull TasksDataSource tasksRemoteDataSource,
                            @NonNull TasksDataSource tasksLocalDataSource) {
        mTasksRemoteDataSource = checkNotNull(tasksRemoteDataSource);
        mTasksLocalDataSource = checkNotNull(tasksLocalDataSource);
    }

        // 这里有些特殊：getInstance的参数是source，RemoteDataSource与LocalDataSource可以替换成fake的source
        // 注意：缓存是内置的，没有用外面的
    public static TasksRepository getInstance(TasksDataSource tasksRemoteDataSource,
                                              TasksDataSource tasksLocalDataSource) {
        if (INSTANCE == null) {
            INSTANCE = new TasksRepository(tasksRemoteDataSource, tasksLocalDataSource);
        }
        return INSTANCE;
    }


    public static void destroyInstance() {
        INSTANCE = null;
    }


        // 数据获取逻辑，可能是从任何地方获取的数据
    @Override
    public void getTasks(@NonNull final LoadTasksCallback callback) {
        checkNotNull(callback);

        // Respond immediately with cache if available and not dirty
        if (mCachedTasks != null && !mCacheIsDirty) {
            callback.onTasksLoaded(new ArrayList<>(mCachedTasks.values()));
            return;
```

```
        }

        if (mCacheIsDirty) {
            // If the cache is dirty we need to fetch new data from the network.
            getTasksFromRemoteDataSource(callback);
        } else {
            // Query the local storage if available. If not, query the network.
            mTasksLocalDataSource.getTasks(new LoadTasksCallback() {
                @Override
                public void onTasksLoaded(List<Task> tasks) {
                    refreshCache(tasks);
                    callback.onTasksLoaded(new ArrayList<>(mCachedTasks.values()));
                }

                @Override
                public void onDataNotAvailable() {
                    getTasksFromRemoteDataSource(callback);
                }
            });
        }
    }

    @Override
    public void saveTask(@NonNull Task task) {
        checkNotNull(task);
        mTasksRemoteDataSource.saveTask(task);
        mTasksLocalDataSource.saveTask(task);

        // Do in memory cache update to keep the app UI up to date
        if (mCachedTasks == null) {
            mCachedTasks = new LinkedHashMap<>();
        }
        mCachedTasks.put(task.getId(), task);
    }
}
```

数据实体

三层的数据Entity与Clean原文中不同

特点：

- 公用，**三层通用了一个数据Model**–Task。减少了Clean构架的三层数据模型之间的转换

总结

仅仅讨论Demo的不完善的地方：

- 没有考虑P层的Presenter的保持
- Domain层没有负责的业务逻辑，没有多UseCase相互调用的例子
- Domain数据处理简单没有性能问题。没有缓存
- 没有Notify机制的示例。都是一个request一个reponse的简单请求。

优点：

- Activty与Fragment职责明确
- Contract设计
- 轻客户端思想，Domain尽量简单（与上面对应，哈哈）
- UseCase的简单设计思想，使得UseCese可以在其他模块复用（参考GetTasks用例）
- Domain的命令模式，设计可以参考
- 数据仓库的设计（缓存和多Source思想）

争论：

- StatisticsPresenter中的统计逻辑位置是否有问题？在主线程？为什么不用一个UseCase？

Android # 主框架

‹ Android构架系列之二--MVP&&Clean理解与实践之Clean

Android构架系列之二--MVP&&Clean理解与实践之问题解答与总结 ›