转载请注明链接: https://blog.csdn.net/feather_wch/article/details/88688666

MVVM-Architecture Component实战

版本号:2019-03-20(21:10)

MVVM vs MVP

1、我们是否需要将App重构为MVVM结构呢？能否继续使用MVP?

> 1. 首先: MVP没有死。MVP仍然是一种特别合理的模式
> 2. MVVM,作为一种模式，也不是一定最好的。关于MVVM的实现，Google已经提供了非常好的Example。But，需要考虑一点为什么之前MVP一直被使用呢?
>   - 因为MVP模式特别适合Android的架构，而且没有任何复杂度。
> 3. 使用MVP模式，并不意味着不能使用 `Architecture Component`中的组件
>   - 也就ViewModel有一些不合适，毕竟是 `Presenter`的替代者
> 4. 并不需要立即重构App到MVVM。 保持健壮的项目结构，比过度使用很多新技术要更重要。

2、MVVM和MVP有什么不同呢?

> 1. MVP和MVVM仅仅有一部分是不一样的:
> 2. MVP中:
>   1. Presenter通过interface和View层交互
> 3. MVMV中:
>   1. ViewModel通过观察者模式和View层交互

> 4. 注意！按照MMVM传统定义，MVP和MVMV之间的区别并不是完全符合上面的说法，但是在Android中这是最适合理解的解释。

实战: 采用ViewModel和LiveData

> 跟着敲两遍，看看注释，百分百能学会如何使用MVVM

Activity

1、Activity中通过MVVM获取到数据，自动更新UI。

> 1. 构造 `ViewModel`
> 2. 查询用户列表
> 3. 监听ViewModel中的LiveData数据，数据改变后自动更新UI。

```
public class ArchActivity extends AppCompatActivity
{
    private UserViewModel mUserViewModel;

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

        // 1. 获取UserViewModel
        mUserViewModel = ViewModelProviders.of(this, new UserViewModel.UserViewModelFactory(new
                .get(UserViewModel.class);
        // 2. 查询【用户列表】，放在这里和监听之后都是一样的，没区别。都能收到查询后的列表。
        mUserViewModel.queryUserlist();
        // 3. 监听User列表的变化
        mUserViewModel.getUserListLiveData().observe(ArchActivity.this, new Observer<List<User>
        {
            @Override
            public void onChanged(@Nullable List<User> users)
            {
                for (User user : users)
                {
                    Log.d("mvvm", "name = " + user.getName() + " age = " + user.getAge());
                }
            }
        });
        // 4. 监听错误信息的提示
        mUserViewModel.getUserListErrorMsg().observe(ArchActivity.this, new Observer<String>()
        {
            @Override
            public void onChanged(@Nullable String errorMsg)
            {
                // 开始注册的时候，会收到一个空消息
                if(!TextUtils.isEmpty(errorMsg)){
                    Toast.makeText(ArchActivity.this, errorMsg, Toast.LENGTH_SHORT).show();
                }
            }
        });
    }
}
```

契约类

2、UserMvvmContract契约类，声明了所有需要的接口

```
/**==========================================
 * @功能 契约类，声明VideModel和Model具有的功能。
 *   1. 避免出现功能的遗漏。
 *   2. 便于开发和维护
 *==========================================*/
public interface UserMvvmContract
{
    // 1、声明：ViewModel提供哪些功能
    public interface IUserViewModel{
        // 获取用户列表
        void queryUserlist();
    }
    // 2、声明：Model需要实现的功能
    public interface IUserTask{
        // 获取用户列表
        void queryUserlist(IUserCallback callback);
    }
    // 3、将Model和ViewModel解耦
    public interface IUserCallback{
        void onQueryUserListSuccess(List<User> data);
        void onQueryUserListFailed(String errmsg);
    }
}
```

ViewModel

3、ViewModel层利用LiveData存储数据，内部通过Model层实际请求数据

```
/**=======================================
 * @功能 LiveData存储数据，借助Model(Task)查询数据
 *=======================================*/
public class UserViewModel extends ViewModel implements UserMvvmContract.IUserViewModel, UserMv
{
    /**=============================================
     * 1、LiveData部分：只有get方法
     *=============================================*/
    private MutableLiveData<List<User>> mUserListLiveData;
    private MutableLiveData<String> mUserListErrorMsg;
    private UserMvvmContract.IUserTask mUserTask;
    // User列表
    public MutableLiveData<List<User>> getUserListLiveData()
    {
        if(mUserListLiveData == null){
            mUserListLiveData = new MutableLiveData<>();
        }
        return mUserListLiveData;
    }
    // User列表，请求错误，错误提示
    public MutableLiveData<String> getUserListErrorMsg()
    {
        if(mUserListErrorMsg == null){
            mUserListErrorMsg = new MutableLiveData<>();
        }
        return mUserListErrorMsg;
    }
    /**=======================================
     * 2、构造部分
     *   1. 需要实际的UserTask
     *   2. 需要ViewModel构造器
     *=================================*/
    public UserViewModel(UserMvvmContract.IUserTask userTask)
    {
        mUserTask = userTask;
    }
    // 构造带参数的ViewModel
    public static class UserViewModelFactory extends ViewModelProvider.NewInstanceFactory{
        private UserMvvmContract.IUserTask mUserTask;
        public UserViewModelFactory(UserMvvmContract.IUserTask userTask)
        {
            mUserTask = userTask;
        }
        @NonNull
        @Override
        public <T extends ViewModel> T create(@NonNull Class<T> modelClass)
        {
            return (T) new UserViewModel(mUserTask);
        }
    }

    /**=================================
     * 3、UserViewModel实际执行任务的部分
     *     1. 实现自：IUserViewModel
```

```
     *==================================*/
    @Override
    public void queryUserlist()
    {
        mUserTask.queryUserlist(this);
    }

    /**==================================
     * 4、根据请求结果，更改LiveData数据。
     *    1. 回调接口实现自: IUserCallback
     *==================================*/
    // 1. 获取到新的用户列表，更新数据。
    @Override
    public void onQueryUserListSuccess(List<User> data)
    {
        getUserListLiveData().postValue(data); // 设置User列表
        getUserListErrorMsg().postValue(""); // 清空错误信息
    }

    // 2. 获取用户列表，失败。更新错误提示信息。
    @Override
    public void onQueryUserListFailed(String errmsg)
    {
        getUserListErrorMsg().postValue(errmsg); // 设置错误信息
    }
}
```

Model

4、Model层进行实际的数据请求，可以通过网络、数据库获取

> UserTask.java

```
public class UserTask implements UserMvvmContract.IUserTask
{
    @Override
    public void queryUserlist(UserMvvmContract.IUserCallback callback)
    {
        ArrayList<User> users = new ArrayList<>();
        users.add(new User("Faker", 17));
        users.add(new User("Rookie", 20));
        users.add(new User("The Shy", 18));
        users.add(new User("JK", 17));
        users.add(new User("Ning", 23));
        users.add(new User("Blue", 20));
        callback.onQueryUserListSuccess(users);
    }
}
```

> User.java

```
public class User
{
    private String mName;
    private int mAge;

    public User(String name, int age)
    {
        mName = name;
        mAge = age;
    }

    public String getName()
    {
        return mName == null ? "" : mName;
    }

    public void setName(String name)
    {
        mName = name;
    }

    public int getAge()
    {
        return mAge;
    }

    public void setAge(int age)
    {
        mAge = age;
    }
}
```

MVP

优点

1、MVP的优点

> 1. 利用Presenter将Model和View完全解耦
> 2. View的变更不会影响Presenter
> 3. Presenter可以复用

缺点

2、MVP具有哪些弊病?

> 1. 粒度不好控制，需要创建非常多的类和接口
> 2. 重用Presenter会实现过多不需要的接口

3. Presenter和View间通过 `Interface` 进行通信，且非常频繁，一旦View的数据发生变化，就需要更改对应的接口。

3、如何解决MVP的弊端

1. 硬解决:
    - 通过 `Template` 自动生成需要的类和接口，减少复制粘贴的冗余工作量
2. 软解决:
    1. 过于简单的逻辑没必要使用Presenter，可以直接处理
    2. 将类似的业务逻辑划分到一起，充分利用Presenter和Model可以重用的特点。

主流实现方式

4、MVP架构主流实现方式有两种

1. 第一种:【使用最多】将 `Activity`、`Fragment` 作为View，抽象出一个 `Presenter`层
2. 第二种: 将 `Activity`、`Fragment` 作为Presenter，抽象出一个 `View`层

5、第一种MVP框架的缺点

1. 不可以直接和Activity、Fragment的生命周期做绑定

6、为什么要和Activity、Fragment的生命周期做绑定?

TheMVP

7、阿里框架TheMVP是典型的第二种MVP框架：抽象出View层

1. 将Activity和Fragment作为Presenter
2. 将UI操作抽到 `Delegate` 中，作为View层。

8、TheMVP的优点

1. 能少写很多类
2. 可以直接和Activity、Fragment的生命周期做绑定
3. 方便重用View【然而现实中View层变化很频繁，大多是重用Presenter】

9、TheMVP的缺点

1. 不能重用Presenter
2. 对Presenter的实现类有限制，必须是Activity、Fragment
3. Adapter、Dialog就必须根据其特性重写对应的Presenter基类
4. Presenter基类继承了Activity或者Fragment，如果需要通过继承使用其他Activity、Fragment就需要修改Presenter基类。

参考资料

1. MVVM with architecture components: a step by step guideline for MVP lovers
2. antoniolg's MVVM Example
3. antoniolg's MVP Example