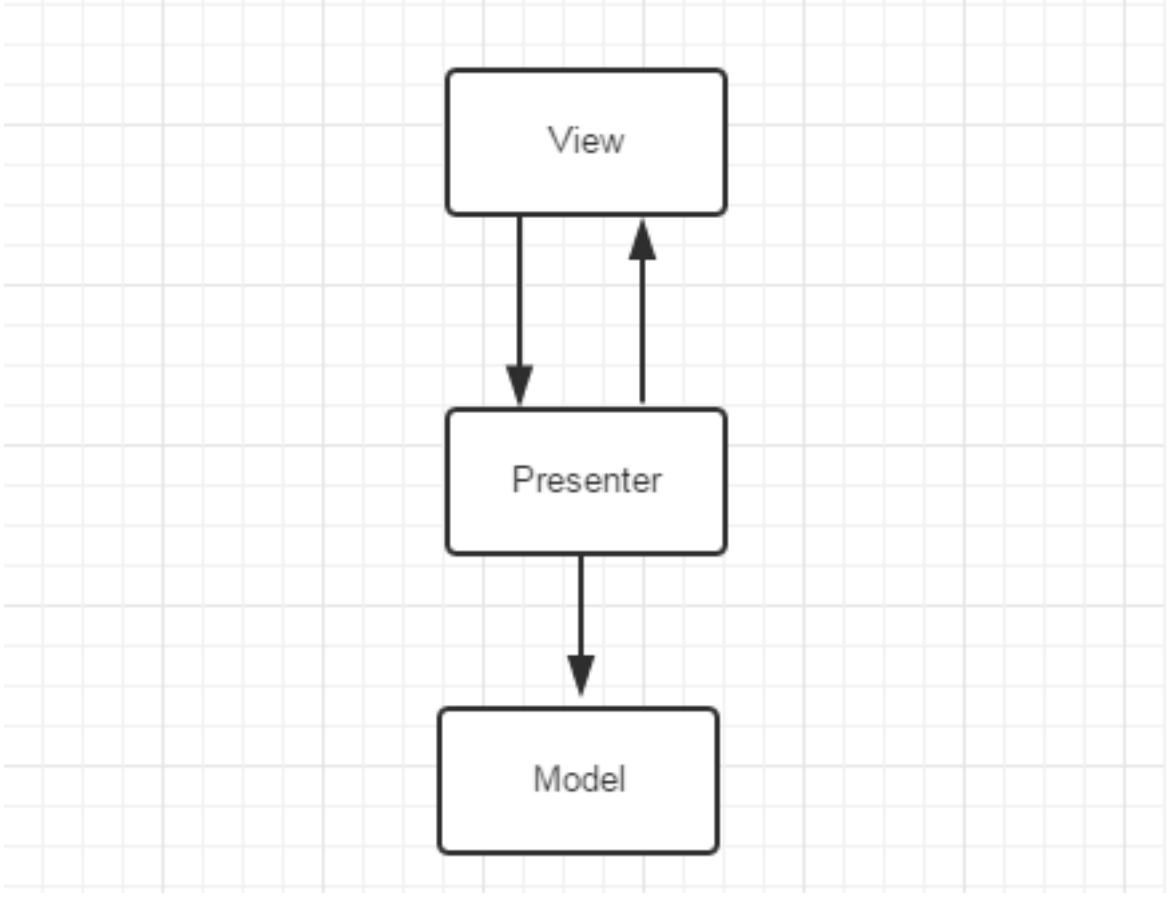
最适合android的MVP模式

MVP简介

相信大家对MVC都是比较熟悉了：M-Model-模型、V-View-视图、C-Controller-控制器，MVP作为MVC的演化版本，那么类似的MVP所对应的意义：M-Model-模型、V-View-视图、P-Presenter-表示器。 从MVC和MVP两者结合来看，Controlller/Presenter在MVC/MVP中都起着逻辑控制处理的角色，起着控制各业务流程的作用。而 MVP与MVC最不同的一点是M与V是不直接关联的也是就Model与View不存在直接关系，这两者之间间隔着的是Presenter层，其负责调控 View与Model之间的间接交互，MVP的结构图如下所示，对于这个图理解即可而不必限于其中的条条框框，毕竟在不同的场景下多少会有些出入的。在 Android中很重要的一点就是对UI的操作基本上需要异步进行也就是在MainThread中才能操作UI，所以对View与Model的切断分离是 合理的。此外Presenter与View、Model的交互使用接口定义交互操作可以进一步达到松耦合也可以通过接口更加方便地进行单元测试。



MVP结构图

关于android的MVP模式其实一直没有一个统一的实现方式，不同的人由于个人理解的不同，进而产生了很多不同的实现方式，其实很难去说哪一种更好，哪一种不好，针对不同的场合，不同的实现方式都有各自的优缺点。

而我采用的MVP是Google提出的一种MVP实现方式，个人认为这种方式实现简单，更加适合android项目。传统的MVP中Model起着处理具体逻辑的功能，Presenter起着隔离和解耦的作用，而在Google的MVP中弱化了Model，Model的逻辑由Presenter来实现，spManager、dbManager可以看做是Model，由Fragment实现View实现视图的变化，Activity作为一个全局的控制者，负责创建view以及presenter实例，并将二者联系起来。

实现步骤

1.BasePresenter

```
public interface BasePresenter {
    void start();
}
```

2.BaseView

```
public interface BaseView<P extends BasePresenter> {
    void setPresenter(P presenter);
}
```

两个接口分别作为Presenter和View的基类，仅定义了最基本的方法，具体页面的view和presenter则分别定义继承的接口，添加属于自己页面的方法。

3.Contract 契约类

这是Google MVP与其他实现方式的不同之一，契约类用于定义同一个界面的view和presenter的接口，通过规范的方法命名或注释，可以清晰的看到整个页面的逻辑。

```
public interface SampleContract {

    interface Presenter extends BasePresenter {
        //获取数据
        void getData(App app, int userId);
        //检查数据是否有效
        void checkData();
        //删除消息
        void deleteMsg(App app, int msgId);
```

```
        ...
    }

    interface View extends BaseView<Presenter> {
        //显示加载中
        void showLoading();
        //刷新界面
        void refreshUI(MessageListEntity.CategoryData data);
        //显示错误界面
        void showError();
        ...
    }
}
```

4.具体的Impl类

Fragment实现View接口，这里使用Google推荐的创建Fragment实例的方法newInstance()，将fragment必备的参数传入。

```
public class SampleFragment extends BaseFragment implements SampleContract.

    private static final String ARG_PARAM1 = "param1";
    private static final String ARG_PARAM2 = "param2";

    private String mParam1;
    private String mParam2;

    private SampleContract.Presenter mPresenter;


    /**
     * Use this factory method to create a new instance of
     * this fragment using the provided parameters.
     *
     * @param param1 Parameter 1.
     * @param param2 Parameter 2.
     * @return A new instance of fragment SampleFragment.
     */
    // TODO: Rename and change types and number of parameters
    public static SampleFragment newInstance(String param1, String param2)
        SampleFragment fragment = new SampleFragment();
        Bundle args = new Bundle();
        args.putString(ARG_PARAM1, param1);
        args.putString(ARG_PARAM2, param2);
        fragment.setArguments(args);
        return fragment;
```

```
}

@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    if (getArguments() != null) {
        mParam1 = getArguments().getString(ARG_PARAM1);
        mParam2 = getArguments().getString(ARG_PARAM2);
    }
}

@Override
public View onCreateView(LayoutInflater inflater, ViewGroup container,
                         Bundle savedInstanceState) {
    TextView textView = new TextView(getActivity());
    textView.setText(R.string.hello_blank_fragment);
    return textView;
}

@Override
public void setPresenter(SampleContract.Presenter presenter) {
    mPresenter = presenter;
}
```

```
    @Override
    public void refreshUI(MessageListEntity.CategoryData data) {
        //change UI
    }

    @Override
    public void showError() {
        //change UI
    }
}
```

Presenter实现类，提供一个参数为对应View的构造器，持有View的引用，并调用View的setPresenter()方法，让View也持有Presenter的引用，方便View调用Presenter的方法。

```
public class SamplePresenterImpl implements SampleContract.Presenter {

    private SampleContract.View mView;

    public SamplePresenterImpl(SampleContract.View mView) {
        this.mView = mView;
        mView.setPresenter(this);
```

```
    }

    ...
}
```

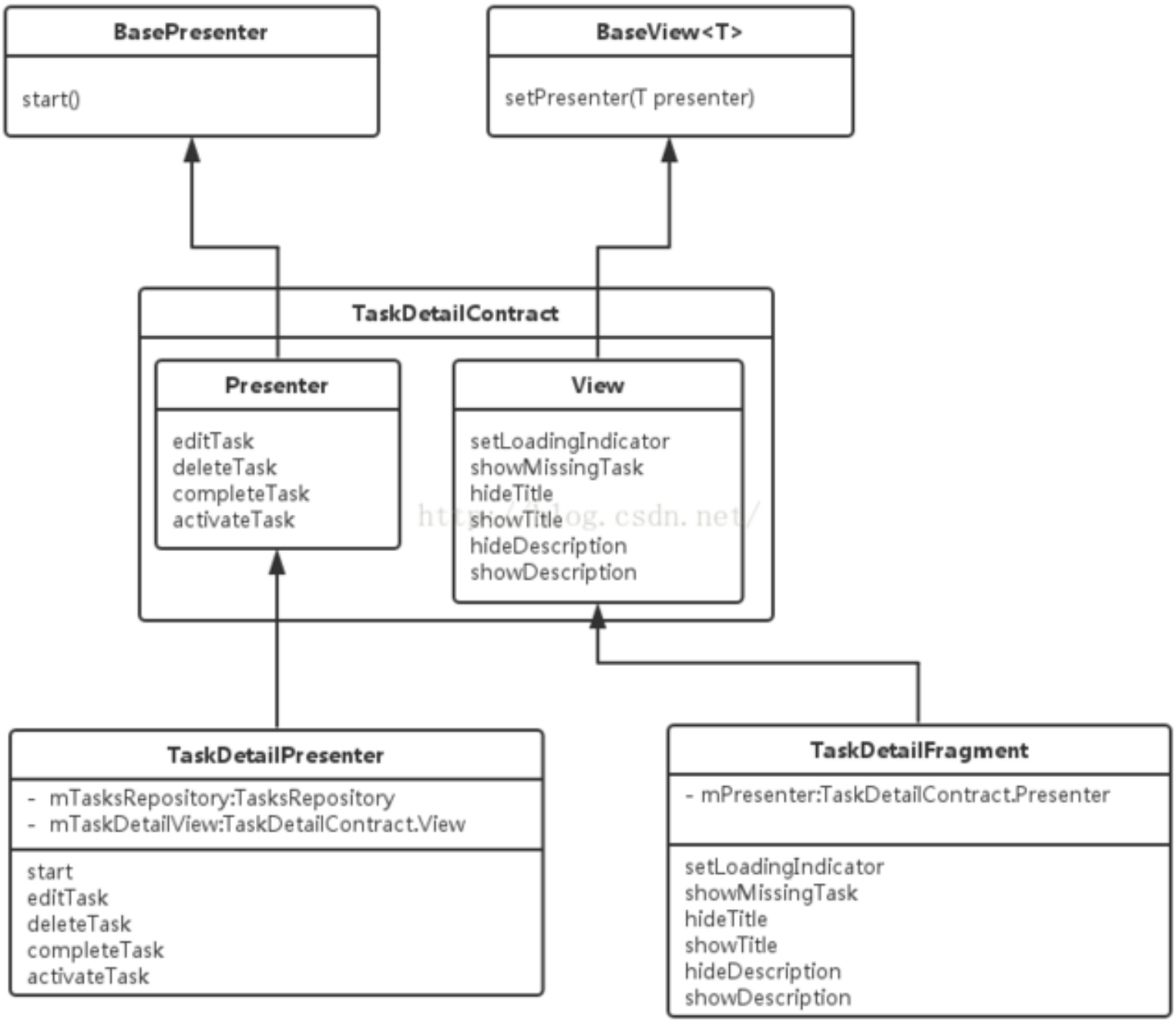
5.最后就是Activity

创建view以及presenter实例，并将二者联系起来。

```
public class SampleActivity extends BaseActivity {

    public static final String FRAGMENT_TAG = "fragment_tag";

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

    private void init() {
        //初始化view
        FragmentManager fragmentManager = getSupportFragmentManager();
        SampleFragment fragment = (SampleFragment) fragmentManager.findFrag
        if (fragment == null) {
            fragment = SampleFragment.newInstance(param1, param2);
            fragmentManager.beginTransaction().add(R.id.fl_container, fragm
        }

        //初始化presenter
        new SamplePresenterImpl(fragment);
    }
}
```

下图是Google官方Demo：todo-mvp模式的架构图

从结构方面观察一下MVP模式



20160510110516376.png

为什么使用MVP?

文章一开始并没有说MVP的好处，而是先介绍了如何实现MVP，在这里我会通过分析来体现MVP的好处。

1.解耦

毋庸置疑这是最大的好处，通过上述例子可以看到一个页面被分成了两个部分，一个是视图的逻辑，另一个是业务逻辑。两者持有的引用都是对方的接口，因此可以随意地替换实现（页面大修改），并且单独修改view或者presenter的逻辑并不会影响另一方(小修改)。

2.代码清晰

以前的MVC模式，当一个页面的逻辑足够复杂，就会出现一个activity上千行的情况，在这样的一个类中想定位一个bug是十分困难的，有时候自己的敲的代码看着都像别人的代码，得重头捋一捋才能定位，相当耗时。而用MVP模式，一个页面的逻辑都可以从Contract类中直观的看到有哪些，找到对应逻辑的方法再进入实现类中找到问题所在，效率上不可同日而语。

3.灵活

好多MVP的实现都是用Activity来实现View，这里使用Fragment便是出于灵活的考虑，Fragment可以复用，可以替换。面对多变的需求可以更从容的应对，例如：一个页面原本是一个列表，后来要求这个页面显示2个Tab，原来的列表变为其中一个Tab。类似的情况可能很常见，如果是用Activity实现的，可能要修改activity类中的很多代码，而原本就使用Fragment和MVP的话，仅需添加一个Fragment，修改Activity假如切换Tab的逻辑，而不用修改第一个Fragment的任何逻辑，这更符合OOP的编程思想。

4.简化

开头说过，传统MVP中Model起着处理具体逻辑的功能，Presenter起着隔离和解耦的作用。这种方式实现的MVP中Presenter看上去更像一个代理类，仅仅是不让View直接访问Model。虽然这样做解耦的程度更高，但实际项目中，一个页面逻辑的修改是非常少的，仅仅在产品修改需求是才会发生，大部分情况下仅仅是修复bug之类的小修改，并需要这样彻底的解耦。从另一个方面来说，一个页面的视图逻辑和业务逻辑本就是一体的。因此，Google的MVP弱化的Model的存在，让Presenter代替传统意义上的Model，减少了因使用MVP而剧增的类。这种方式相比不适用MVP仅从1个activity，增加到了1个activity，1个fragment（view），1个presenter，1个contract（如果有传统意义上的Model，还会有1个Model的接口和1个实现类）。

注意点！

1.一段逻辑到底是放在View还是放在Presenter？其实从定义我们也可以知道View只处理界面的逻辑，任何逻辑判断都应该在presenter中实现。但在实际过程中很容易出现View处理逻辑的情况，例如：网络请求返回一些数据，需要对数据进行删选或处理，处理的过程很容易发生在View层，这是需要极力避免的情况，所有不涉及界面的判断都要放到presenter层。

2.类名、方法名的规范。页面的契约类都以contract结尾，presenter的实现类都以PresenterImpl结尾。View和Presenter接口中的方法需要见名知意，意义模糊的可以适当添加注释。规范的命名有助于后续的修改和他人的维护。

3.子线程回调。Presenter中处理子线程任务完成后，一般会回到主线程调用View的方法刷新UI，但如果此时activity已经销毁，并且没有取消子线程的任务（例如网络请求），此时去调用View的方法很容易发生空指针，应该在调用之前判断一下，因此建议view增加一个isActive()方法，用于判断当前view是否可见：

```
public interface BaseView<P extends BasePresenter> {
    void setPresenter(P presenter);
    boolean isActive();
}
```

在fragment中只需这样实现即可：

```
    @Override
    public boolean isActive() {
        return isAdded();
    }
```

除此之外，如果子线程的任务会持续很久，或者由于网络等额外因素导致子线程耗时过久，但此时activity其实已经destroy了，presenter的子线程还在运行则不会被GC，并且presenter持有了fragment（view），导致了内存泄露。解决这个问题可以通过弱应用的方式解决，下是实现方式：

```
public abstract class BasePresenterImpl<V extends BaseView> implements Base

    protected WeakReference<V> mView;

    public BasePresenterImpl(V view) {
        mView = new WeakReference<V>(view);
        view.setPresenter(this);
    }

    protected boolean isViewActive() {
        return mView != null && mView.get().isActive();
    }
```

```
    public void detachView() {
        if (mView != null) {
            mView.clear();
            mView = null;
        }
    }

}
```

view也可以抽象abstract类。