Android设计模式源码解析之桥接模式

> 本文为 Android 设计模式源码解析 中 桥接模式 分析
> Android系统版本： 4.2
> 分析者：shen0834，分析状态：未完成，校对者：Mr.Simple，校对状态：完成

模式介绍

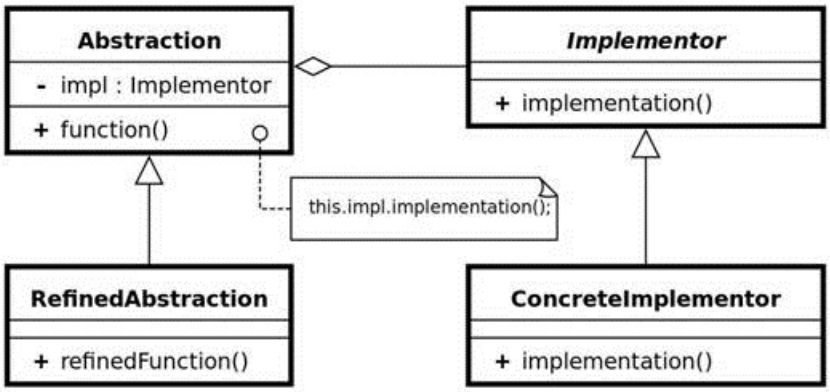
模式的定义

将抽象部分与实现部分分离，使它们都可以独立的变化。

模式的使用场景

- 如果一个系统需要在构件的抽象化角色和具体化角色之间添加更多的灵活性，避免在两个层次之间建立静态的联系。
- 设计要求实现化角色的任何改变不应当影响客户端，或者实现化角色的改变对客户端是完全透明的。
- 需要跨越多个平台的图形和窗口系统上。
- 一个类存在两个独立变化的维度，且两个维度都需要进行扩展。

UML类图



角色介绍

- 抽象化(Abstraction)角色：抽象化给出的定义，并保存一个对实现化对象的引用。 修正抽象化(Refined Abstraction)角色：扩展抽象化角色，改变和修正父类对抽象化的定义。
- 实现化(Implementor)角色：这个角色给出实现化角色的接口，但不给出具体的实现。必须指出的是，这个接 口不一定和抽象化角色的接口定义相同，实际上，这两个接口可以非常不一样。实现化角色应当只给出底层操作，而抽象化角色应当只给出基于底层操作的更高一层的操作。
- 具体实现化(ConcreteImplementor)角色：这个角色给出实现化角色接口的具体实现。

模式的简单实现

介绍

其实Java的虚拟机就是一个很好的例子，在不同平台平台上，用不同的虚拟机进行实现，这样只需把Java程序编译成符合虚拟机规范的文件，且只用编译一次，便在不同平台上都能工作。 但是这样说比较抽象，用一个简单的例子来实现bridge模式。

编写一个程序，使用两个绘图的程序的其中一个来绘制矩形或者原型，同时，在实例化矩形的时候，它要知道使用绘图程序1（DP1）还是绘图程序2（DP2）。

(ps:假设dp1和dp2的绘制方式不一样，它们是用不同方式进行绘制，示例代码，不讨论

过多细节)

实现源码

```
    首先是两个绘图程序dp1,dp2
//具体的绘图程序类dp1
public class DP1 {

    public void draw_1_Rantanle(){
        System.out.println("使用DP1的程序画矩形");
    }

    public void draw_1_Circle(){
        System.out.println("使用DP1的程序画圆形");
    }
}
//具体的绘图程序类dp2
public class DP2 {

    public void drawRantanle(){
        System.out.println("使用DP2的程序画矩形");
    }

    public void drawCircle(){
        System.out.println("使用DP2的程序画圆形");
    }

}
接着 抽象的形状Shape和两个派生类：矩形Rantanle和圆形Circle
//抽象化角色Abstraction
abstract class Shape {
    //持有实现的角色Implementor(作图类)
    protected Drawing myDrawing;

    public Shape(Drawing drawing) {
        this.myDrawing = drawing;
    }

    abstract public void draw();

    //保护方法drawRectangle
    protected void drawRectangle(){
       //this.impl.implmentation()
        myDrawing.drawRantangle();
    }

    //保护方法drawCircle
    protected void drawCircle(){
        //this.impl.implmentation()
        myDrawing.drawCircle();
    }
}
//修正抽象化角色Refined Abstraction(矩形)
public class Rantangle extends Shape{
    public Rantangle(Drawing drawing) {
        super(drawing);
    }

    @Override
    public void draw() {
        drawRectangle();
    }
}
//修正抽象化角色Refined Abstraction(圆形)
public class Circle extends Shape {
    public Circle(Drawing drawing) {
        super(drawing);
    }
    @Override
```

```
    public void draw() {
        drawCircle();
    }
}
最后，我们的实现绘图的Drawing和分别实现dp1的V1Drawing和dp2的V2Drawing
//实现化角色Implementor
//implmentation两个方法，画圆和画矩形
public interface Drawing {
    public void drawRantangle();
    public void drawCircle();
}
//具体实现化逻辑ConcreteImplementor
//实现了接口方法，使用DP1进行绘图
public class V1Drawing implements Drawing{

    DP1 dp1;

    public V1Drawing() {
        dp1 = new DP1();
    }
    @Override
    public void drawRantangle() {
        dp1.draw_1_Rantanle();
    }
    @Override
    public void drawCircle() {
        dp1.draw_1_Circle();
    }
}
//具体实现化逻辑ConcreteImplementor
//实现了接口方法，使用DP2进行绘图
public class V2Drawing implements Drawing{
    DP2 dp2;

    public V2Drawing() {
        dp2 = new DP2();
    }

    @Override
    public void drawRantangle() {
        dp2.drawRantanle();
    }
    @Override
    public void drawCircle() {
        dp2.drawCircle();
    }
}
```

在这个示例中，图形Shape类有两种类型，圆形和矩形，为了使用不同的绘图程序绘制图形，把实现的部分进行了分离，构成了Drawing类层次结构，包括V1Drawing和V2Drawing。在具体实现类中，V1Drawing控制着DP1程序进行绘图，V2Drawing控制着DP2程序进行绘图，以及保护的方法drawRantangle,drawCircle(Shape类中) 。

Android源码中的模式实现

在Android中也运用到了Bridge模式，我们使用很多的ListView和BaseAdpater其实就是Bridge模式的运行，很多人会问这个不是Adapter模式，接下来根据源码来分析。

首先ListAdapter.java：

```
public interface ListAdapter extends Adapter{
    //继承自Adapter，扩展了自己的两个实现方法
      public boolean areAllItemsEnabled();
      boolean isEnabled(int position);
}
```

这里先来看一下父类AdapterView。

```
public abstract class AdapterView<T extends Adapter> extends ViewGroup {
    //这里需要一个泛型的Adapter
      public abstract T getAdapter();
```

```
    public abstract void setAdapter(T adapter);
}
```

接着来看ListView的父类AbsListView，继承自AdapterView

```
public abstract class AbsListView extends AdapterView<ListAdapter>
    //继承自AdapterView,并且指明了T为ListAdapter
    /**
     * The adapter containing the data to be displayed by this view
     */
    ListAdapter mAdapter;
    //代码省略
    //这里实现了setAdapter的方法，实例对实现化对象的用
    public void setAdapter(ListAdapter adapter) {
        //这的adapter是从子类传入上来，也就是listview，拿到了具体实现化的对象
        if (adapter != null) {
            mAdapterHasStableIds = mAdapter.hasStableIds();
            if (mChoiceMode != CHOICE_MODE_NONE && mAdapterHasStableIds &&
                    mCheckedIdStates == null) {
                mCheckedIdStates = new LongSparseArray<Integer>();
            }
        }
        if (mCheckStates != null) {
            mCheckStates.clear();
        }
        if (mCheckedIdStates != null) {
            mCheckedIdStates.clear();
        }
    }
```

大家都知道，构建一个listview，adapter中最重要的两个方法，getCount()告知数量，getview()告知具体的view类型，接下来看看AbsListView作为一个视图的集合是如何来根据实现化对象adapter来实现的具体的view呢？

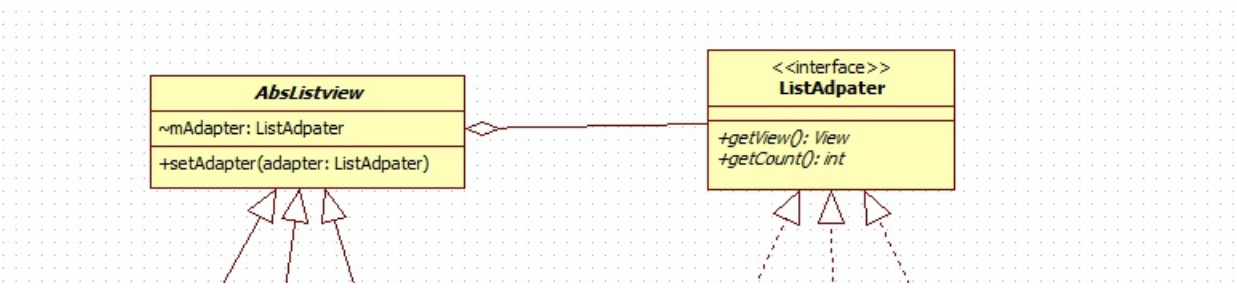
```
   protected void onAttachedToWindow() {
        super.onAttachedToWindow();

            //省略代码，
            //这里在加入window的时候，getCount()确定了集合的个数
            mDataChanged = true;
            mOldItemCount = mItemCount;
            mItemCount = mAdapter.getCount();
        }
    }
```

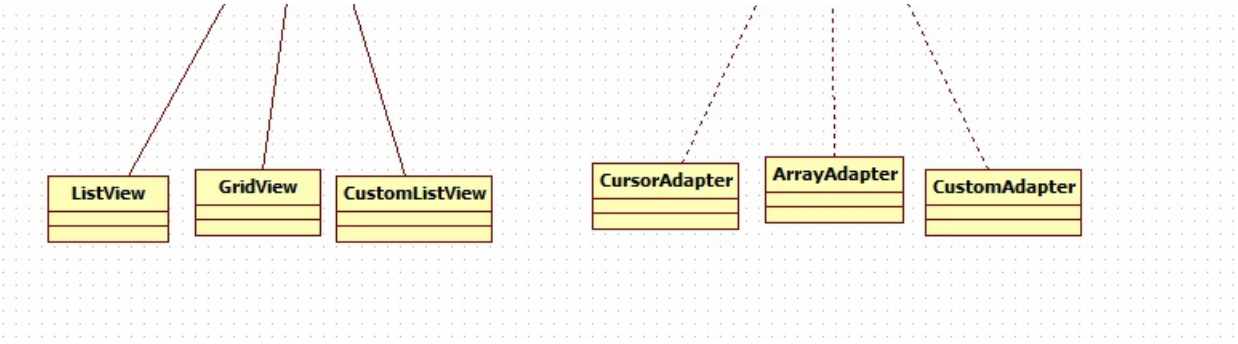
接着来看

```
View obtainView(int position, boolean[] isScrap) {
        //代码省略
       //这里根据位置显示具体的view,return的child是从持有的实现对象mAdapter里面的具体实现的
       //方法getview来得到的。
        final View child = mAdapter.getView(position, scrapView, this);
        //代码省略
        return child;
    }
```

接下来在ListView中，onMeasure调用了obtainView来确定宽高，在扩展自己的方法来排列这些view。知道了

这些以后，我们来画一个简易的UML图来看下:





对比下GOF的上图，是不是发现很像呢？实际上最开始研究Adapter模式的时候,越看越不对啊，于是整理结构，画了UML发现这更像是一个bridge模式，那时候对设计模式也是模模糊糊的，于是静下来研究。抽象化的角色一个视图的集合AdapterView，它扩展了AbsListView，AbsSpinner，接下来他们分别扩展了ListView，GridView，Spinner,Gallery，用不同方式来展现这些ItemViews，我们继续扩展类似ListView的PulltoRefreshView等等。而实现化角色Adapter扩展了ListAdpater,SpinnerAdapter，接着具体的实现化角色BaseAdapter实现了他们，我们通过继承BaseAdapter又实现了我们各式各样的ItemView。

杂谈

这里就是Android工程师的牛X之处了，用一个bridge和adapter来解决了一个大的难题。试想一下，视图的排列方式是无穷尽，是人们每个人开发的视图也是无穷尽的。如果你正常开发，你需要多少类来完成呢？而Android把最常用用的展现方式全部都封装了出来，而在实现角色通过Adapter模式来应变无穷无尽的视图需要。抽象化了一个容器使用适配器来给容器里面添加视图，容器的形状(或理解为展现的方式)以及怎么样来绘制容器内的视图，你都可以独自的变化，双双不会干扰，真正的脱耦，就要最开始说的那样：“将抽象部分与实现部分分离，使它们都可以独立的变化。”

从上面的两个案例，我们可以看出，我们在两个解决方案中都用到bridge和adapter模式，那是因为我们必须使用给定的绘图程序(adapter适配器)，绘图程序(adapter适配器)有已经存在的接口必须要遵循，因此需要使用Adapter进行适配，然后才能用同样的方式处理他们,他们经常一起使用，并且相似，但是Adapter并不是Bridge的一部分。

优点与缺点

实现与使用实现的对象解耦，提供了可扩展性，客户对象无需担心操作的实现问题。 如果你采用了bridge模式，在处理新的实现将会非常容易。你只需定义一个新的具体实现类，并且实现它就好了，不需要修改任何其他的东西。但是如果你出现了一个新的具体情况，需要对实现进行修改时，就得先修改抽象的接口，再对其派生类进行修改，但是这种修改只会存在于局部,并且这种修改将变化的英雄控制在局部，并且降低了出现副作用的风险，而且类之间的关系十分清晰，如何实现一目了然。