> 本文转载自：菜刀文大神的GitHub

面向对象的六大原则

- **单一职责原则**

所谓职责是指类变化的原因。如果一个类有多于一个的动机被改变，那么这个类就具有多于一个的职责。而单一职责原则就是指一个类或者模块应该有且只有一个改变的原因。通俗的说，即一个类只负责一项职责，将一组相关性很高的函数、数据封装到一个类中。

- **开闭原则**

对于扩展是开放的，这意味着模块的行为是可以扩展的。当应用的需求改变时，我们可以对模块进行扩展，使其具有满足那些改变的新行为。　　对于修改是关闭的，对模块行为进行扩展时，不必改动模块的源代码。

通俗的说，尽量通过扩展的方式实现系统的升级维护和新功能添加，而不是通过修改已有的源代码。

- **里氏替换原则**

使用“抽象(Abstraction)”和“多态(Polymorphism)”将设计中的静态结构改为动态结构，维持设计的封闭性。任何基类可以出现的地方，子类一定可以出现。

在软件中将一个基类对象替换成它的子类对象，程序将不会产生任何错误和异常，反过来则不成立。在程序中尽量使用基类类型来对对象进行定义，而在运行时再确定其子类类型，用子类对象来替换父类对象。

- **依赖倒置原则**

高层次的模块不应该依赖于低层次的模块，他们都应该依赖于抽象。抽象不应该依赖于具体实现，具体实现应该依赖于抽象。
　程序要依赖于抽象接口，不要依赖于具体实现。简单的说就是要求对抽象进行编程，不要对实现进行编程，这样就降低了客户与实现模块间的耦合（各个模块之间相互传递的参数声明为抽象类型，而不是声明为具体的实现类）。

- **接口隔离原则**

一个类对另一个类的依赖应该建立在最小的接口上。其原则是将非常庞大的、臃肿的接口拆分成更小的更具体的接口。

- **迪米特原则**

又叫作最少知识原则，就是说一个对象应当对其他对象有尽可能少的了解。　　通俗地讲，一个类应该对自己需要耦合或调用的类知道得最少，不关心被耦合或调用的类的内部实现，只负责调用你提供的方法。

下面开始设计模式学习．．．

1. Singleton（单例模式）

作用：

> 保证在Java应用程序中，一个类Class只有一个实例存在。

好处：

> 由于单例模式在内存中只有一个实例，减少了内存开销。
>
> 单例模式可以避免对资源的多重占用，例如一个写文件时，由于只有一个实例存在内存中，避免对同一个资源文件的同时写操作。　　单例模式可以再系统设置全局的访问点，优化和共享资源访问。

使用情况：

> 建立目录 数据库连接的单线程操作
>
> 某个需要被频繁访问的实例对象

1.1 使用方法

第一种形式：

```
public class Singleton {

    /* 持有私有静态实例，防止被引用，此处赋值为null，目的是实现延迟加载 */
    private static Singleton instance = null;

    /* 私有构造方法，防止被实例化 */
    private Singleton() {
    }

    /* 懒汉式：第一次调用时初始Singleton，以后就不用再生成了
    静态方法，创建实例 */
    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

但是这有一个问题，不同步啊！在对据库对象进行的频繁读写操作时，不同步问题就大了。

第二种形式：

既然不同步那就给getInstance方法加个锁呗！我们知道使用synchronized关键字可以同步方法和同步代码块，所以：

```
 public static synchronized Singleton getInstance() {
     if (instance == null) {
         instance = new Singleton();
     }
     return instance;
 }
```

或是

```
public static Singleton getInstance() {
     synchronized (Singleton.class) {
         if (instance == null) {
             instance = new Singleton();
         }
     }
     return instance;
 }
```

获取Singleton实例：

```
Singleton.getInstance().方法（）
```

1.2 android中的Singleton

> 软键盘管理的 InputMethodManager

源码(以下的源码都是5.1的)：

```
205 public final class InputMethodManager {
//.........
211     static InputMethodManager sInstance;
//.........
619     public static InputMethodManager getInstance() {
620         synchronized (InputMethodManager.class) {
621             if (sInstance == null) {
622                 IBinder b = ServiceManager.getService(Context.INPUT_METHOD_SERVICE);
623                 IInputMethodManager service = IInputMethodManager.Stub.asInterface(b);
624                 sInstance = new InputMethodManager(service, Looper.getMainLooper());
625             }
626             return sInstance;
627         }
628     }
```

使用的是第二种同步代码块的单例模式（可能涉及到多线程），类似的还有 AccessibilityManager（View获得点击、焦点、文字改变等事件的分发管理，对整个系统的调试、问题定位等） BluetoothOppManager等。

当然也有同步方法的单例实现，比如：CalendarDatabaseHelper

```
307     public static synchronized CalendarDatabaseHelper getInstance(Context context) {
308         if (sSingleton == null) {
309             sSingleton = new CalendarDatabaseHelper(context);
310         }
311         return sSingleton;
312     }
```

注意Application并不算是单例模式

```
44 public class Application extends ContextWrapper implements ComponentCallbacks2 {

79     public Application() {
80         super(null);
81     }
```

在Application源码中，其构造方法是公有的，意味着可以生出多个Application实例，但为什么Application能实现一个app只存在一个实例呢？请看下面：

在ContextWrapper源码中：

```
50 public class ContextWrapper extends Context {
51     Context mBase;
52
53     public ContextWrapper(Context base) {
54         mBase = base;
55     }

64     protected void attachBaseContext(Context base) {
65         if (mBase != null) {
66             throw new IllegalStateException("Base context already set");
67         }
68         mBase = base;
69     }
```

ContextWrapper构造函数传入的base为null, 就算有多个Application实例，但是没有通过attach()绑定相关信息，没有上下文环境，三个字。

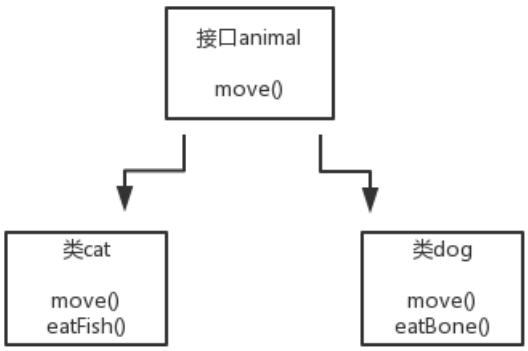
然并卵

2 . Factory（工厂模式）

> 定义一个用于创建对象的接口，让子类决定实例化哪一个类。工厂方法使一个类的实例化延迟到其子类。 **对同一个接口的实现类进行管理和实例化创建**

假设我们有这样一个需求：

动物Animal，它有行为move()。有两个实现类cat和dog。为了统一管理和创建我们设计一个工厂模式。 同时两个子类有各自的行为，Cat有eatFish()，Dog有eatBone().



结构图：

Animal接口：

```
interface animal {
    void move();
}
```

Cat类:

```
public class Cat implements Animal{

    @Override
    public void move() {
        // TODO Auto-generated method stub
        System.out.println("我是只肥猫，不爱动");
    }
    public void eatFish() {
        System.out.println("爱吃鱼");
    }
}
```

Dog类:

```
public class Dog implements Animal{

    @Override
    public void move() {
        // TODO Auto-generated method stub
        System.out.println("我是狗，跑的快");
    }
    public void eatBone() {
        System.out.println("爱吃骨头");
    }
}
```

那么现在就可以建一个工厂类（Factory.java）来对实例类进行管理和创建了.

```
public class Factory {
    //静态工厂方法
    //多处调用，不需要实例工厂类
     public static Cat produceCat() {
            return new Cat();
        }
     public static Dog produceDog() {
            return new Dog();
        }
//当然也可以一个方法，通过传入参数，switch实现
}
```

使用：

```
Animal cat = Factory.produceCat();
cat.move();
//---------------------------
Dog dog = Factory.produceDog();
dog.move();
dog.eatBone();
```

工厂模式在业界运用十分广泛，如果都用new来生成对象，随着项目的扩展，animal还可以生出许多其他儿子来，当然儿子还有儿子，同时也避免不了对以前代码的修改（比如加入后来生出儿子的实例）,怎么管理，想着就是一团糟。

```
Animal cat = Factory.produceCat();
```

这里实例化了Animal但不涉及到Animal的具体子类（减少了它们之间的偶合联系性），达到封装效果，也就减少错误修改的机会。

Java面向对象的原则，封装(Encapsulation)和分派(Delegation)告诉我们：**具体事情做得越多，越容易范错误，**

一般来说，这样的普通工厂就可以满足基本需求。但是我们如果要新增一个Animal的实现类panda，那么必然要在工厂类里新增了一个生产panda的方法。就违背了 闭包的设计原则（对扩展要开放对修改要关闭），于是有了抽象工厂模式。

2 .1 Abstract Factory(抽象工厂)

抽象工厂模式提供一个创建一系列相关或相互依赖对象的接口，而无需指定它们具体的类。　　啥意思？就是把**生产抽象成一个接口，每个实例类都对应一个工厂类**（普通工厂只有一个工厂类），**同时所有工厂类都继承这个生产接口**。

生产接口Provider：

```
interface Provider {
    Animal produce();
}
```

每个产品都有自己的工厂 CatFactory：

```
public class CatFactory implements Provider{

    @Override
    public Animal produce() {
        // TODO Auto-generated method stub
        return new Cat();
    }
}
```

DogFactory：

```
public class DogFactory implements Provider{

    @Override
    public Animal produce() {
        // TODO Auto-generated method stub
        return new Dog();
    }
}
```

产品生产：

```
Provider provider = new CatFactory();
Animal cat =provider.produce();
cat.move();
```

现在我们要加入panda，直接新建一个pandaFactory就行了，这样我们系统就非常灵活，具备了动态扩展功能。

2 .1 Android中的Factory

比如AsyncTask的抽象工厂实现：

工厂的抽象：

```
59  public interface ThreadFactory {
//省略为备注
69      Thread newThread(Runnable r);
70  }
```

产品的抽象（new Runnable就是其实现类）：

```
56   public interface Runnable {
//省略为备注
68    public abstract void run();
69 }
```

AsyncTask中工厂类的实现：

```
185    private static final ThreadFactory sThreadFactory = new ThreadFactory() {
186        private final AtomicInteger mCount = new AtomicInteger(1);
187
188        public Thread newThread(Runnable r) {
189            return new Thread(r, "AsyncTask #" + mCount.getAndIncrement());
190        }
191    };
```

我们可以创建另外类似的工厂，生产某种专门的线程（多线程），非常容易扩展。 当然，android中的应用还有很多（比如BitmapFactory），有兴趣的小伙伴可以去扒一扒。

3. Adapter（适配器模式）

将一个类的接口转换成客户希望的另外一个接口。

我们经常碰到要将两个没有关系的类组合在一起使用，第一解决方案是：修改各自类的接口，但是如果我们没有源代码，或者，我们不愿意为了一个应用而修改各自的接口。 怎么办?

使用Adapter，在这两种接口之间创建一个混合接口。

> 模式中的角色
>
> 需要适配的类（Adaptee）：需要适配的类。
>
> 适配器（Adapter）：通过包装一个需要适配的对象，把原接口转换成目标接口。
>
> 目标接口（Target）：客户所期待的接口。可以是**具体的或抽象的类，也可以是接口。**


```
// 需要适配的类
class Adaptee {
    public void specificRequest() {
        System.out.println("需要适配的类");
    }
}

// 目标接口
interface Target {
    public void request();
}
```

实现方式：

①、对象适配器（采用对象组合方式实现）

```
// 适配器类实现标准接口
class Adapter implements Target{
    // 直接关联被适配类
    private Adaptee adaptee;

    // 可以通过构造函数传入具体需要适配的被适配类对象
    public Adapter (Adaptee adaptee) {
        this.adaptee = adaptee;
    }

    public void request() {
        // 这里是使用委托的方式完成特殊功能
        this.adaptee.specificRequest();
    }
}

// 测试类
public class Client {
    public static void main(String[] args) {
        // 需要先创建一个被适配类的对象作为参数
        Target adapter = new Adapter(new Adaptee());
        adapter.request();
    }
}
```

如果Target不是接口而是一个具体的类的情况，这里的Adapter直接继承Target就可以了。

②、类的适配器模式（采用继承实现）

```
// 适配器类继承了被适配类同时实现标准接口
class Adapter extends Adaptee implements Target{
    public void request() {
        super.specificRequest();
    }
}

// 测试类
    public static void main(String[] args) {
        // 使用适配类
        Target adapter = new Adapter();
        adapter.request();
    }
```

如果Target和 Adaptee都是接口，并且都有实现类。 可以通过Adapter实现两个接口来完成适配。 还有一种叫PluggableAdapters,可以动态的获取几个adapters中一个。使用Reflection技术，可以动态的发现类中的Public方法。

> 优点

系统需要使用现有的类，而此类的接口不符合系统的需要。那么通过适配器模式就可以让这些功能得到**更好的复用**。 将目标类和适配者类解耦，通过引入一个适配器类重用现有的适配者类，而无需修改原有代码，**更好的扩展性**。

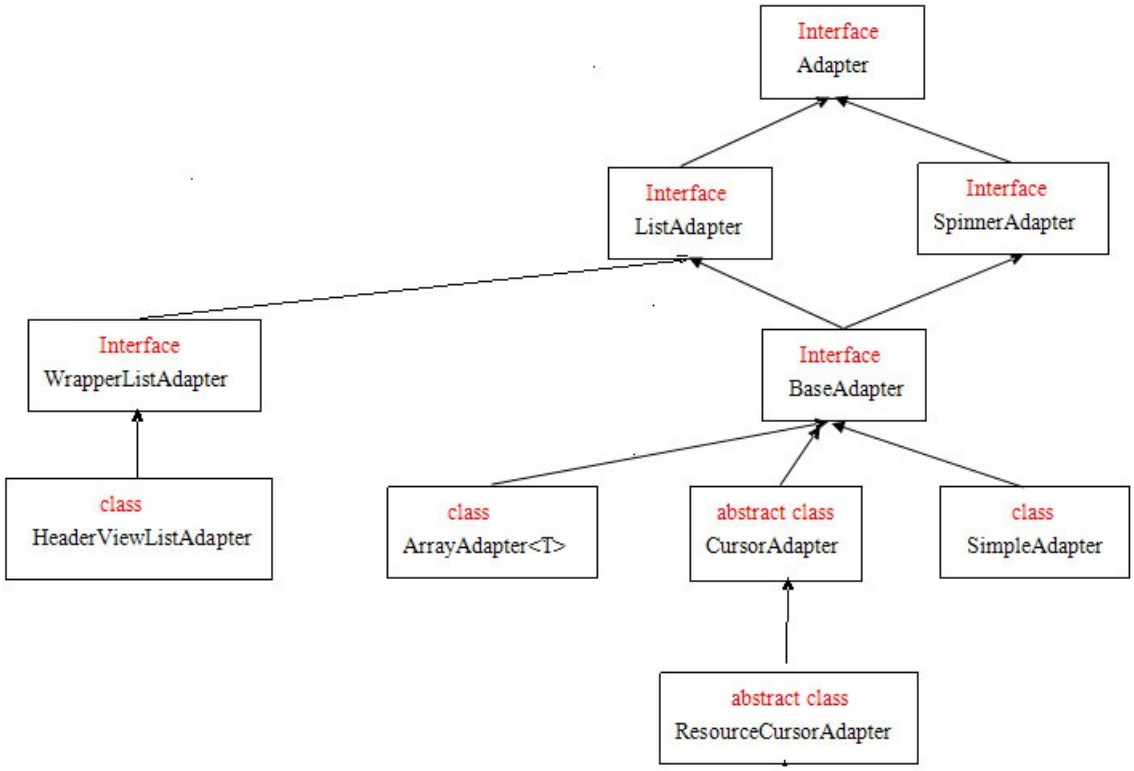
> 缺点

过多的使用适配器，会让系统非常零乱，不易整体进行把握。比如，明明看到调用的是A接口，其实内部被适配成了B接口的实现。如果不是必要，不要使用适配器，而是直接对系统进行重构。

3 .1 Android中的Adapter

android中的Adapter就有很多了，这个大家都经常用。 Adapter是AdapterView视图与数据之间的桥梁，Adapter提供对数据的访问，也负责为每一项数据产生一个对应的View。

> Adapter的继承结构



BaseAdapter的部分源码：

```
30public abstract class BaseAdapter implements ListAdapter, SpinnerAdapter {
31    private final DataSetObservable mDataSetObservable = new DataSetObservable();
32
33    public boolean hasStableIds() {
34        return false;
35    }
36
37    public void registerDataSetObserver(DataSetObserver observer) {
38        mDataSetObservable.registerObserver(observer);
39    }
40
41    public void unregisterDataSetObserver(DataSetObserver observer) {
42        mDataSetObservable.unregisterObserver(observer);
43    }
```

ListAdapter, SpinnerAdapter都是Target ， 数据是Adaptee ， 采用对象组合方式。

4. Chain of Responsibility（责任链模式）

使多个对象都有机会处理请求，从而避免请求的发送者和接收者之间的耦合关系。将这些对象连成一条链，并沿着这条链传递该请求，直到有一个对象处理它为止。　　发出这个请求的客户端并不知道链上的哪一个对象最终处理这个请求，这使得系统可以在不影响客户端的情况下动态地重新组织和分配责任。

编程中的小体现：

```
if(a<10){
    ...
}
else if (a<20) {
    ...
}
else if(a<30){
    ...
}
else{
    ...
}
```

程序必须依次扫描每个分支进行判断，找到对应的分支进行处理。

> 责任链模式的优点

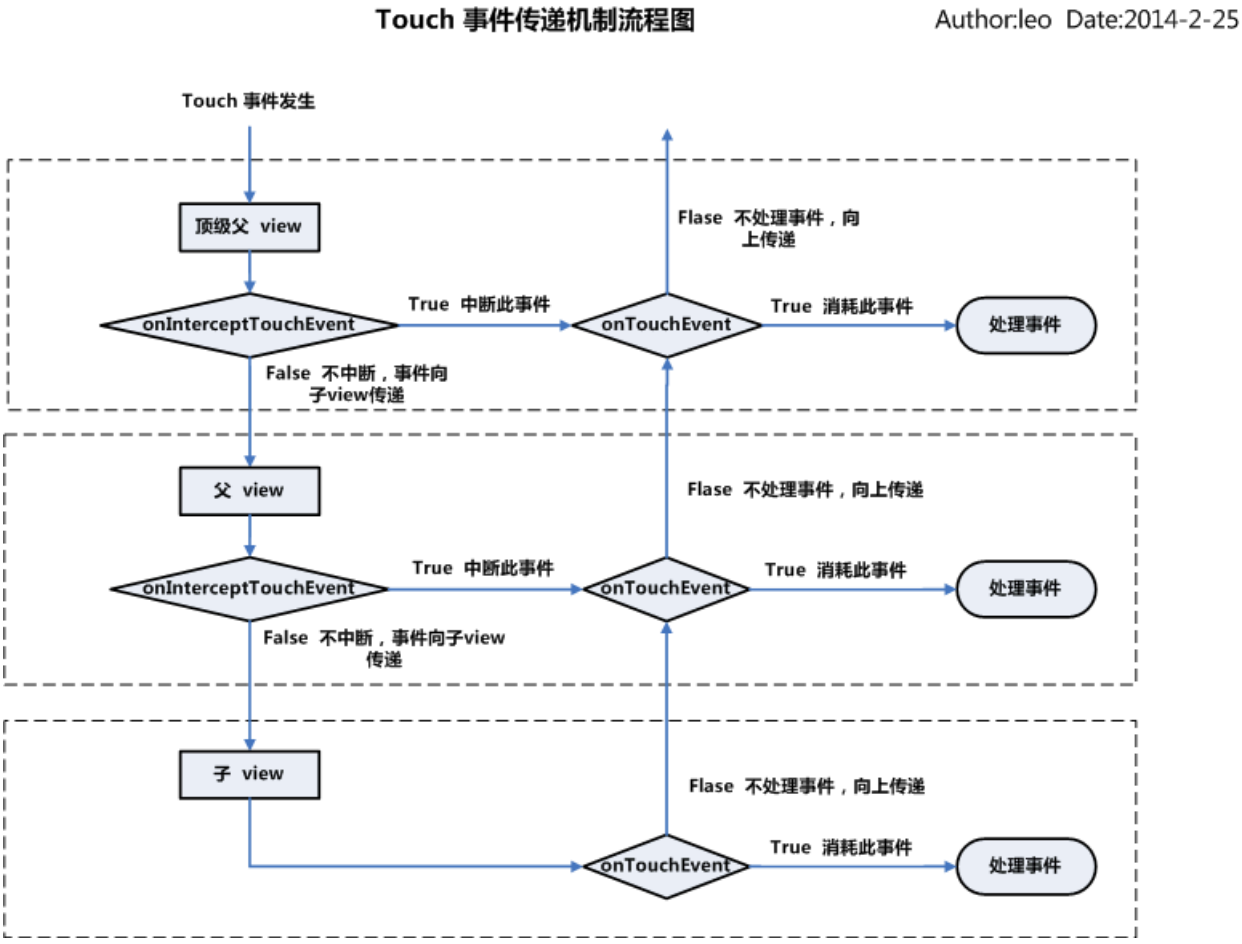
可以降低系统的耦合度（请求者与处理者代码分离），简化对象的相互连接，同时增强给对象指派职责的灵活性，增加新的请求处理类也很方便；

> 责任链模式的缺点

不能保证请求一定被接收，且对于比较长的职责链，请求的处理可能涉及到多个处理对象，系统性能将受到一定影响，而且在进行代码调试时不太方便。　　每次都是从链头开始，这也正是链表的缺点。

4 .1 Android中的Chain of Responsibility

触摸、按键等各种事件的传递



有兴趣的可以看一下这篇文章View事件分发机制源码分析，我这就不多说了。

5. Observer（观察者模式）

有时被称作发布/订阅模式，观察者模式定义了一种**一对多**的依赖关系，让多个观察者对象同时监听某一个主题对象。这个主题对象在状态发生变化时，会通知所有观察者对象，使它们能够自动更新自己。

将一个系统分割成一个一些类相互协作的类有一个不好的副作用，那就是需要维护相关对象间的一致性。我们不希望为了维持一致性而使各类紧密耦合，这样会给维护、扩展和重用都带来不便。观察者就是解决这类的耦合关系的（依赖关系并未完全解除，抽象通知者依旧依赖抽象的观察者。）。

> 观察者模式的组成

①抽象主题（Subject）

它把所有观察者对象的引用保存到一个聚集里，每个主题都可以有任何数量的观察者。抽象主题提供一个接口，可以增加和删除观察者对象。

②具体主题（ConcreteSubject）

将有关状态存入具体观察者对象；在具体主题内部状态改变时，给所有登记过的观察者发出通知。

③抽象观察者（Observer）

为所有的具体观察者定义一个接口，在得到主题通知时更新自己。

④具体观察者（ConcreteObserver）

实现抽象观察者角色所要求的更新接口，以便使本身的状态与主题状态协调。

言语苍白，上代码：

```
//抽象观察者
public interface Observer
{
  public void update(String str);

}
```

```
//具体观察者
public class ConcreteObserver implements Observer{
    @Override
    public void update(String str) {
        // TODO Auto-generated method stub
        System.out.println(str);
    }
}
```

```
//抽象主题
public interface Subject
{
    public void addObserver(Observer observer);
    public void removeObserver(Observer observer);
    public void notifyObservers(String str);
}
```

```
//具体主题
public class ConcreteSubject implements Subject{
    // 存放观察者
    private List<Observer> list = new ArrayList<Observer>();
    @Override
    public void addObserver(Observer observer) {
        // TODO Auto-generated method stub
        list.add(observer);
    }

    @Override
    public void removeObserver(Observer observer) {
        // TODO Auto-generated method stub
        list.remove(observer);
    }

    @Override
    public void notifyObservers(String str) {
        // TODO Auto-generated method stub
        for(Observer observer:list){
            observer.update(str);
        }
    }
}
```

下面是测试类：

```
/**
 * @author fanrunqi
 */
public class Test {
     public static void main(String[] args) {
        //一个主题
         ConcreteSubject eatSubject = new ConcreteSubject();
         //两个观察者
         ConcreteObserver personOne = new ConcreteObserver();
         ConcreteObserver personTwo = new ConcreteObserver();
         //观察者订阅主题
         eatSubject.addObserver(personOne);
         eatSubject.addObserver(personTwo);

         //通知开饭了
         eatSubject.notifyObservers("开饭啦");
    }
}
```

> “关于代码你有什么想说的？”“没有，都在代码里了”“（⊙ｏ⊙）哦．．．．．”

5 .1 Android中的Observer

观察者模式在android中运用的也比较多，最熟悉的ContentObserver。

① 抽象类ContentResolver中（Subject）

注册观察者：

```
1567    public final void registerContentObserver(Uri uri, boolean notifyForDescendents,
1568            ContentObserver observer, int userHandle)
```

取消观察者：

```
1583    public final void unregisterContentObserver(ContentObserver observer)
```

抽象类ContentObserver中（Observer）

```
94     public void onChange(boolean selfChange) {
95         // Do nothing.  Subclass should override.
96     }

144    public void onChange(boolean selfChange, Uri uri, int userId) {
145        onChange(selfChange, uri);
146    }

129    public void onChange(boolean selfChange, Uri uri) {
130        onChange(selfChange);
131    }
```

观察特定Uri引起的数据库的变化，继而做一些相应的处理（最终都调用的第一个函数）.

② DataSetObserver，其实这个我们一直在用，只是没意识到。

我们再看到BaseAdapter的部分源码：

```
37    public void registerDataSetObserver(DataSetObserver observer) {
38        mDataSetObservable.registerObserver(observer);
39    }
40
41    public void unregisterDataSetObserver(DataSetObserver observer) {
42        mDataSetObservable.unregisterObserver(observer);
43    }
```

上面两个方法分别向向BaseAdater注册、注销一个DataSetObserver实例。

DataSetObserver 的源码：

```
24  public abstract class DataSetObserver {

29      public void onChanged() {
30          // Do nothing
31      }

38      public void onInvalidated() {
39          // Do nothing
40      }
41  }
```

DataSetObserver就是一个观察者，它一旦发现BaseAdapter内部数据有变量，就会通过回调方法DataSetObserver.onChanged和DataSetObserver.onInvalidated来通知DataSetObserver的实现类。

6．Builder（建造者模式）

建造者模式：是将一个复杂的对象的构建与它的表示分离（同构建不同表示），使得同样的构建过程可以创建不同的表示。

> 一个人活到70岁以上，都会经历这样的几个阶段：婴儿，少年，青年，中年，老年。并且每个人在各个阶段肯定是不一样的，世界上不存在两个人在人生的这5个阶段的生活完全一样，但是活到70岁以上的人，都经历了这几个阶段是肯定的。实际上这是一个比较经典的建造者模式的例子了。

将复杂的内部创建封装在内部，对于外部调用的人来说，只需要传入建造者和建造工具，对于内部是如何建造成成品的，调用者无需关心。

建造者模式通常包括下面几个角色：

① Builder：一个抽象接口，用来规范产品对象的各个组成成分的建造。

② ConcreteBuilder：实现Builder接口，针对不同的商业逻辑，具体化复杂对象的各部分的创建，在建造过程完成后，提供产品的实例。

③ Director：指导者，调用具体建造者来创建复杂对象的各个部分，不涉及具体产品的信息，只负责保证对象各部分完整创建或按某种顺序创建。

④ Product：要创建的复杂对象。

> 与抽象工厂的区别：在建造者模式里，有个指导者，由指导者来管理建造者，用户是与指导者联系的，指导者联系建造者最后得到产品。即建造模式可以强制实行一种分步骤进行的建造过程。

Product和产品的部分Part接口

```
public interface Product { }
public interface Part { }
```

Builder：

```
public interface Builder {
      void buildPartOne();
      void buildPartTwo();

      Product getProduct();
  }
```

ConcreteBuilder:

```
//具体建造工具
   public class ConcreteBuilder implements Builder {
      Part partOne, partTwo;

      public void buildPartOne() {
         //具体构建代码
      };
      public void buildPartTwo() {
         //具体构建代码
      };
       public Product getProduct() {
         //返回最后组装的产品
      };
   }
```

Director :

```
  public class Director {
      private Builder builder;

      public Director( Builder builder ) {
         this.builder = builder;
      }
      public void construct() {
         builder.buildPartOne();
         builder.buildPartTwo();
      }
   }
```

建造：

```
ConcreteBuilder builder = new ConcreteBuilder();
Director director = new Director(builder);
//开始各部分建造
director.construct();
Product product = builder.getResult();
```

> 优点：

客户端不必知道产品内部组成的细节。

具体的建造者类之间是相互独立的，对系统的扩展非常有利。

由于具体的建造者是独立的，因此可以对建造过程逐步细化，而不对其他的模块产生任何影响。

> 使用场合：

创建一些复杂的对象时，这些对象的内部组成构件间的建造**顺序是稳定**的，但是对象的内**部组成构件面临着复杂的变化**。

要创建的复杂对象的算法，独立于该对象的组成部分，也独立于组成部分的装配方法时。

6 .1 Android中的Builder

android中的Dialog就使用了Builder Pattern，下面来看看AlertDialog的部分源码。

```
371     public static class Builder {
372         private final AlertController.AlertParams P;
373         private int mTheme;

393         public Builder(Context context, int theme) {
394             P = new AlertController.AlertParams(new ContextThemeWrapper(
395                     context, resolveDialogTheme(context, theme)));
396             mTheme = theme;
397         }
```

AlertDialog的Builder 是一个静态内部类，没有定义Builder 的抽象接口。　　对AlertDialog设置的属性会保存在Build类的成员变量P（AlertController.AlertParams）中。

Builder类中部分方法：

```
416         public Builder setTitle(int titleId) {
417             P.mTitle = P.mContext.getText(titleId);
418             return this;
419         }
```

```
452         public Builder setMessage(int messageId) {
453             P.mMessage = P.mContext.getText(messageId);
454             return this;
455         }
```

```
525         public Builder setPositiveButton(CharSequence text, final OnClickListener listener) {
526             P.mPositiveButtonText = text;
527             P.mPositiveButtonListener = listener;
528             return this;
529         }
```

而show()方法会返回一个结合上面设置的dialog实例

```
991         public AlertDialog show() {
992             AlertDialog dialog = create();
993             dialog.show();
994             return dialog;
995         }
996     }
997
998 }
```

```
972         public AlertDialog create() {
973             final AlertDialog dialog = new AlertDialog(P.mContext, mTheme, false);
974             P.apply(dialog.mAlert);
975             dialog.setCancelable(P.mCancelable);
976             if (P.mCancelable) {
977                 dialog.setCanceledOnTouchOutside(true);
978             }
979             dialog.setOnCancelListener(P.mOnCancelListener);
980             dialog.setOnDismissListener(P.mOnDismissListener);
981             if (P.mOnKeyListener != null) {
982                 dialog.setOnKeyListener(P.mOnKeyListener);
983             }
984             return dialog;
985         }
```

简单建造：

```
new AlertDialog.Builder(context)
 .setTitle("标题")
 .setMessage("消息框")
 .setPositiveButton("确定", null)
 .show();
```

7 . Memento（备忘录模式）

备忘录模式又叫做快照模式(Snapshot Pattern)或Token模式，是对象的行为模式。

备忘录对象是一个用来存储另外一个对象内部状态的快照的对象。备忘录模式的用意是在不破坏封装的条件下，将一个对象的状态捕捉(Capture)住，并外部化，存储起来，从而可以在将来合适的时候把这个对象还原到存储起来的状态。备忘录模式常常与命令模式和迭代子模式一同使用。

备忘录模式所涉及的角色有三个：

① Originator(发起人)： 负责创建一个备忘录Memento，用以记录当前时刻它的内部状态，并可使用备忘录恢复内部状态。Originator可根据需要决定Memento存储Originator的哪些内部状态。

② Memento(备忘录)： 负责存储Originnator对象的内部状态，并可防止Originator以外的其他对象访问备忘录Memento，备忘录有两个接口，Caretaker只能看到备忘录的窄接口，它只能将备忘录传递给其他对象。

③、 Caretaker(管理者):负责保存好备忘录Memento，不能对备忘录的内容进行操作或检查。


```
public class Originator {

    private String state;
    /**
     * 工厂方法，返回一个新的备忘录对象
     */
    public Memento createMemento(){
        return new Memento(state);
    }
    /**
     * 将发起人恢复到备忘录对象所记载的状态
     */
    public void restoreMemento(Memento memento){
        this.state = memento.getState();
    }

    public String getState() {
        return state;
    }

    public void setState(String state) {
        this.state = state;
        System.out.println("当前状态：" + this.state);
    }

}
```

```
public class Memento {

    private String state;

    public Memento(String state){
        this.state = state;
    }

    public String getState() {
        return state;
    }

    public void setState(String state) {
        this.state = state;
    }

}
```

```
public class Caretaker {

    private Memento memento;
    /**
     * 备忘录的取值方法
     */
    public Memento retrieveMemento(){
        return this.memento;
    }
    /**
     * 备忘录的赋值方法
     */
    public void saveMemento(Memento memento){
        this.memento = memento;
    }
}
```

使用：

```
Originator o = new Originator();
Caretaker c = new Caretaker();
//改变负责人对象的状态
o.setState("On");
//创建备忘录对象，并将发起人对象的状态储存起来
 c.saveMemento(o.createMemento());
//修改发起人的状态
o.setState("Off");
//恢复发起人对象的状态
 o.restoreMemento(c.retrieveMemento());
```

不需要了解对象的内部结构的情况下备份对象的状态，方便以后恢复。

7.1 Android中的Memento

Activity的onSaveInstanceState和onRestoreInstanceState就是通过Bundle（相当于备忘录对象）这种序列化的数据结构来存储Activity的状态，至于其中存储的数据结构，这两个方法不用关心。

还是看一下源码：

```
1365    protected void onSaveInstanceState(Bundle outState) {
1366        outState.putBundle(WINDOW_HIERARCHY_TAG, mWindow.saveHierarchyState());
1367        Parcelable p = mFragments.saveAllState();
1368        if (p != null) {
1369            outState.putParcelable(FRAGMENTS_TAG, p);
1370        }
1371        getApplication().dispatchActivitySaveInstanceState(this, outState);
1372    }
```

```
1019    protected void onRestoreInstanceState(Bundle savedInstanceState) {
1020        if (mWindow != null) {
1021            Bundle windowState = savedInstanceState.getBundle(WINDOW_HIERARCHY_TAG);
1022            if (windowState != null) {
1023                mWindow.restoreHierarchyState(windowState);
1024            }
1025        }
1026    }
```

8. Prototype（原型模式）

原型模式，能快速克隆出一个与已经存在对象类似的另外一个我们想要的新对象。　　工作原理是：通过将一个原型对象传给那个要发动创建的对象，这个要发动创建的对象通过请求原型对象拷贝它们自己来实施创建。

分为深拷贝和浅拷贝。深拷贝就是把对象里面的引用的对象也要拷贝一份新的对象，并将这个新的引用对象作为拷贝的对象引用（多读两遍）。

一般使用原型模式有个明显的特点，就是实现cloneable的clone()方法。

在Intent源码中：

```
4084    @Override
4085    public Object clone() {
4086        return new Intent(this);
4087    }
```

这里Intent通过实现Cloneable接口来实现原型拷贝。

9. Strategy（策略模式）

定义：有一系列的算法，将每个算法封装起来（每个算法可以封装到不同的类中），各个算法之间可以替换，策略模式让算法独立于使用它的客户而独立变化。

举例：　　一个影碟机，你往里面插什么碟子，就能放出什么电影。　　属性动画，设置不同的插值器对象，就可以得到不同的变化曲线。　　返回值解析，传入什么样的解析器，就可以把二进制数据转换成什么格式的数据，比如String、Json、XML。

策略模式其实就是多态的一个淋漓精致的体现。

在android中不同Animation动画的实现，主要是依靠Interpolator的不同而实现的。

```
401     public void setInterpolator(Interpolator i) {
402         mInterpolator = i;
403     }
```

10. Template（模板模式）

定义：定义一个操作中的算法框架，而将一些步骤延迟到子类中，使得子类可以不改变一个算法的结构即可重定义该算法的某些特定的步骤。 **实现流程已经确定，实现细节由子类完成。** 生命周期对于我们都不陌生，它就是典型的Template模式，在具体流程确定的情况下，至于我们要复写生命周期那些方法，实现那些功能由继承activity的子类去具体实现。

关键在于必须有具体的执行流程，比如AsyncTask。

11. Proxy（代理模式）

定义：为其他对象提供一种代理以控制对这个对象的访问。 代理： 在出发点到目的地之间有一道中间层。

应用：Android跨进程通信方式 ，建议去了解一下Binder机制。

12. Interpreter（解释器模式）

定义语言的文法，并且建立一个解释器来解释该语言中的句子。

比如Android中通过PackageManagerService来解析AndroidManifest.xml中定义的Activity、service等属性。

13. State（状态模式）

行为是由状态来决定的，不同状态下有不同行为。

注意：状态模式的行为是平行的、不可替换的，策略模式的行为是彼此独立可相互替换的。

体现：不同的状态执行不同的行为，当WIFI开启时，自动扫描周围的接入点，然后以列表的形式展示；当wifi关闭时则清空。

14. Command（命令模式）

我们有很多命令，把它们放在一个下拉菜单中，用户通过先选择菜单再选择具体命令，这就是Command模式。

本来用户(调用者)是直接调用这些命令的，在菜单上打开文档，就直接指向打开文档的代码，使用Command模式，就是在这两者之间增加一个中间者，将这种直接关系拗断，同时两者之间都隔离,基本没有关系了。 显然这样做的好处是符合封装的特性，降低耦合度，有利于代码的健壮性 可维护性 还有复用性。Command是将对行为进行封装的典型模式，Factory是将创建进行封装的模式。

android底层逻辑对事件的转发处理就用到了Command模式。

15. Iterator（迭代模式）

提供一种方法顺序访问一个容器对象中的各个元素，而不需要暴露该对象的内部表示。

应用：

在Java中的Iterator类。

Android中的 Cursor。

```
cursor.moveToFirst();
```

16. Composite（组合模式）

将对象以树形结构组织起来，以达成“部分－整体” 的层次结构，使得客户端对单个对象和组合对象的使用具有一致性。

Android中View的结构是树形结构，每个ViewGroup包含一系列的View，而ViewGroup本身又是View。这是Android中非常典型的组合模式。

17. Flyweight（共享模式/享元模式）

定义：避免大量拥有相同内容的小类的开销(如耗费内存)，使大家共享一个类(元类)。

面向对象语言的原则就是一切都是对象，但是如果真正使用起来，有时对象数可能显得很庞大，比如，字处理软件，如果以每个文字都作为一个对象，几千个字，对象数就是几千，无疑耗费内存，那么我们还是要"求同存异"，找出这些对象群的共同点，设计一个元类，封装可以被共享的类，另外，还有一些特性是取决于应用(context)，是不可共享的，这也Flyweight中两个重要概念内部状态intrinsic和外部状态extrinsic之分。

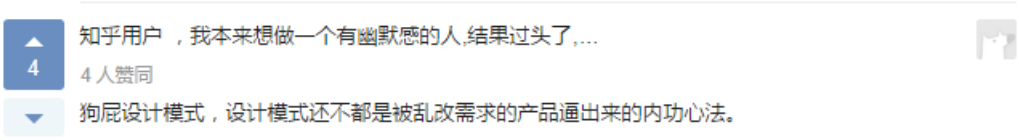
说白点，就是先捏一个的原始模型，然后随着不同场合和环境，再产生各具特征的具体模型，很显然，在这里需要产生不同的新对象，所以Flyweight模式中常出现Factory模式。Flyweight的内部状态是用来共享的，Flyweight factory负责维护一个Flyweight pool(模式池)来存放内部状态的对象。

Flyweight模式是一个提高程序效率和性能的模式，会大大加快程序的运行速度。应用场合很多：比如你要从一个数据库中读取一系列字符串，这些字符串中有许多是重复的，那么我们可以将这些字符串储存在Flyweight池(pool)中。

在Android线程通信中，每次获取Message时调Message.obtain()其实就是从消息池中取出可重复使用的消息，避免产生大量的Message对象。

最后

> 那么问题来了，什么是设计模式？



> 设计模式是前辈、大牛在实际编程中对遇到的问题解决方案的抽象。