Android设计模式源码解析之责任链模式

> 本文为 Android 设计模式源码解析 中责任链模式分析
> Android系统版本： 4.4.4
> 分析者：Aige，分析状态：完成，校对者：SM哥，校对状态：撒丫校对中

1. 模式介绍

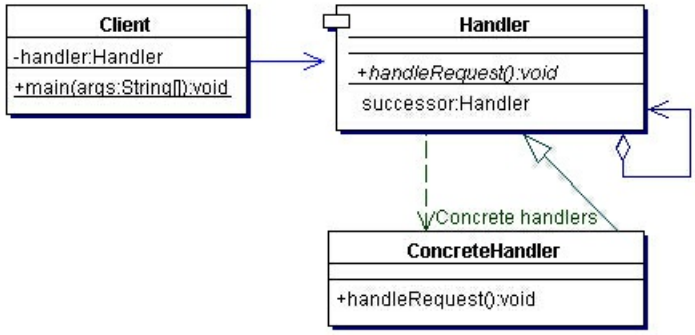
模式的定义

一个请求沿着一条“链”传递，直到该“链”上的某个处理者处理它为止。

模式的使用场景

一个请求可以被多个处理者处理或处理者未明确指定时。

2. UML类图



角色介绍

Client：客户端

Handler：抽象处理者

ConcreteHandler：具体处理者

3. 模式的简单实现

简单实现的介绍

责任链模式非常简单异常好理解，相信我它比单例模式还简单易懂，其应用也几乎无所不在，甚至可以这么说……从你敲代码的第一天起你就不知不觉用过了它最原始的裸体结构：分支语句：

```
public class SimpleResponsibility {
    public static void main(String[] args) {
        int request = (int) (Math.random() * 3);
        switch (request) {
        case 0:
            System.out.println("SMBother handle it: " + request);
            break;
        case 1:
            System.out.println("Aige handle it: " + request);
            break;
        case 2:
            System.out.println("7Bother handle it: " + request);
            break;
        default:
            break;
        }
    }
}
```

谁敢说没用过上面这种结构体的站出来我保证不打屎他，没用过swith至少if-else用过吧，if-else都没用过你怎么知道github的……上面的这段代码其实就是一种最最简单的责

任链模式，其根据request的值进行不同的处理。当然这只是个不恰当的例子来让大家尽快对责任链模式有个简单的理解，因为可能很多童鞋第一次听说这个模式，而人对未知事物总是恐惧的，为了消除大家的这种恐惧，我将大家最常见的code搬出来相信熟悉的代码对大家来说有一种亲切的感觉，当然我们实际应用中的责任链模式绝逼不是这么Mr.Simple，但是也不会复杂不到哪去。责任链模式，顾名思义，必定与责任Responsibility相关，其实质呢就像上面定义中说的那样一个请求（比如上面代码中的request值）沿着一条“链”（比如上面代码中我们的switch分支语句）传递，当某个处于“链”上的处理者（case定义的条件）处理它时完成处理。其实现实生活中关于责任者模式的例子数不胜数，最常见的就是工作中上下级之间的责任请求关系了。比如：

> 程序猿狗屎运被派出去异国出差一周，这时候就要去申请一定的差旅费了，你心里小算一笔加上各种车马费估计大概要个两三万，于是先向小组长汇报申请，可是大于一千块小组长没权利批复，于是只好去找项目主管，项目主管一看妈蛋这么狠要这么多我只能批小于五千块的，于是你只能再跑去找部门经理，部门经理看了下一阵淫笑后说没法批我只能批小于一万的，于是你只能狗血地去跪求老总，老总一看哟！小伙子心忒黑啊！老总话虽如此但还是把钱批给你了毕竟是给公司办事，到此申请处理完毕，你也可以屁颠屁颠地滚了。

如果把上面的场景应用到责任链模式，那么我们的request请求就是申请经费，组长主管经理老总们就是一个个具体的责任人他们可以对请求做出处理但是他们只能在自己的责任范围内处理该处理的请求，而程序猿只是个底层狗请求者向责任人们发起请求.....苦逼的猿。

实现源码

上面的场景我们可以使用使用如下的代码来模拟实现：

首先定义一个程序员类：

```
/**
 * 程序猿类
 *
 * @author Aige{@link https://github.com/AigeStudio}
 *
 */
public class ProgramApe {
    private int expenses;// 声明整型成员变量表示出差费用
    private String apply = "爹要点钱出差";// 声明字符串型成员变量表示差旅申请

    /*
     * 含参构造方法
     */
    public ProgramApe(int expenses) {
        this.expenses = expenses;
    }

    /*
     * 获取程序员具体的差旅费用
     */
    public int getExpenses() {
        return expenses;
    }

    /*
     * 获取差旅费申请
     */
    public String getApply() {
        return apply;
    }
}
```

然后依次是各个大爷类：

```
/**
 * 小组长类
 *
 * @author Aige{@link https://github.com/AigeStudio}
 *
 */
```

```
 */
public class GroupLeader {

    /**
     * 处理请求
     *
     * @param ape
     *            具体的猿
     */
    public void handleRequest(ProgramApe ape) {
        System.out.println(ape.getApply());
        System.out.println("GroupLeader: Of course Yes!");
    }
}
```

```
/**
 * 项目主管类
 *
 * @author Aige{@link https://github.com/AigeStudio}
 *
 */
public class Director {
    /**
     * 处理请求
     *
     * @param ape
     *            具体的猿
     */
    public void handleRequest(ProgramApe ape) {
        System.out.println(ape.getApply());
        System.out.println("Director: Of course Yes!");
    }
}
```

```
/**
 * 部门经理类
 *
 * @author Aige{@link https://github.com/AigeStudio}
 *
 */
public class Manager {
    /**
     * 处理请求
     *
     * @param ape
     *            具体的猿
     */
    public void handleRequest(ProgramApe ape) {
        System.out.println(ape.getApply());
        System.out.println("Manager: Of course Yes!");
    }
}
```

```
/**
 * 老总类
 *
 * @author Aige{@link https://github.com/AigeStudio}
 *
 */
public class Boss {
    /**
     * 处理请求
     *
     * @param ape
     *            具体的猿
     */
    public void handleRequest(ProgramApe ape) {
        System.out.println(ape.getApply());
        System.out.println("Boss: Of course Yes!");
    }
}
```

好了，万事俱备只欠场景，现在我们模拟一下整个场景过程：

```
/**
 * 场景模拟类
 *
 * @author Aige{@link https://github.com/AigeStudio}
 *
 */
public class Client {
    public static void main(String[] args) {
        /*
         * 先来一个程序猿 这里给他一个三万以内的随机值表示需要申请的差旅费
         */
        ProgramApe ape = new ProgramApe((int) (Math.random() * 30000));

        /*
         * 再来四个老大
         */
        GroupLeader leader = new GroupLeader();
        Director director = new Director();
        Manager manager = new Manager();
        Boss boss = new Boss();

        /*
         * 处理申请
         */
        if (ape.getExpenses() <= 1000) {
            leader.handleRequest(ape);
        } else if (ape.getExpenses() <= 5000) {
            director.handleRequest(ape);
        } else if (ape.getExpenses() <= 10000) {
            manager.handleRequest(ape);
        } else {
            boss.handleRequest(ape);
        }
    }
}
```

运行一下，我的结果输出如下（注：由于随机值的原因你的结果也许与我不一样）：

> 爹要点钱出差
>
> Manager: Of course Yes!

是不是感觉有点懂了？当然上面的代码虽然在一定程度上体现了责任链模式的思想，但是确是非常terrible的。作为一个code新手可以原谅，但是对有一定经验的code+来说就不可饶恕了，很明显所有的老大都有共同的handleRequest方法而程序猿也有不同类型的，比如一个公司的php、c/c++、Android、IOS等等，所有的这些共性我们都可以将其抽象为一个抽象类或接口，比如我们的程序猿抽象父类：

```
/**
 * 程序猿抽象接口
 *
 * @author Aige{@link https://github.com/AigeStudio}
 *
 */
public abstract class ProgramApes {
    /**
     * 获取程序员具体的差旅费用
     *
     * @return 要多少钱
     */
    public abstract int getExpenses();

    /**
     * 获取差旅费申请
     *
     * @return Just a request
     */
```

```
    public abstract String getApply();
}
```

这时我们就可以实现该接口使用呆毛具现化一个具体的程序猿，比如Android猿：

```
/**
 * Android程序猿类
 *
 * @author Aige{@link https://github.com/AigeStudio}
 *
 */
public class AndroidApe extends ProgramApes {
    private int expenses;// 声明整型成员变量表示出差费用
    private String apply = "爹要点钱出差";// 声明字符串型成员变量表示差旅申请

    /*
     * 含参构造方法
     */
    public AndroidApe(int expenses) {
        this.expenses = expenses;
    }

    @Override
    public int getExpenses() {
        return expenses;
    }

    @Override
    public String getApply() {
        return apply;
    }
}
```

同样的，所有的老大都有一个批复经费申请的权利，我们把这个权利抽象为一个IPower接口：

```
/**
 * 老大们的权利接口
 *
 * @author Aige{@link https://github.com/AigeStudio}
 *
 */
public interface IPower {
    /**
     * 处理请求
     *
     * @param ape
     *            具体的猿
     */
    public void handleRequest(ProgramApe ape);
}
```

然后让所有的老大们实现该接口即可其它不变，而场景类Client中也只是修改各个老大的引用类型为IPower而已，具体代码就不贴了，运行效果也类似。

然而上面的代码依然问题重重，为什么呢？大家想想，当程序猿发出一个申请时却是在场景类中做出判断决定的……然而这个职责事实上应该由老大们来承担并作出决定，上面的代码搞反了……既然知道了错误，那么我们就来再次重构一下代码：

把所有老大抽象为一个leader抽象类，在该抽象类中实现处理逻辑：

```
/**
 * 领导人抽象类
 *
 * @author Aige{@link https://github.com/AigeStudio}
 *
 */
public abstract class Leader {
    private int expenses;// 当前领导能批复的金额
    private Leader mSuperiorLeader;// 上级领导
```

```
    /**
     * 含参构造方法
     *
     * @param expenses
     *            当前领导能批复的金额
     */
    public Leader(int expenses) {
        this.expenses = expenses;
    }

    /**
     * 回应程序猿
     *
     * @param ape
     *            具体的程序猿
     */
    protected abstract void reply(ProgramApe ape);

    /**
     * 处理请求
     *
     * @param ape
     *            具体的程序猿
     */
    public void handleRequest(ProgramApe ape) {
        /*
         * 如果说程序猿申请的money在当前领导的批复范围内
         */
        if (ape.getExpenses() <= expenses) {
            // 那么就由当前领导批复即可
            reply(ape);
        } else {
            /*
             * 否则看看当前领导有木有上级
             */
            if (null != mSuperiorLeader) {
                // 有的话简单撒直接扔给上级处理即可
                mSuperiorLeader.handleRequest(ape);
            } else {
                // 没有上级的话就批复不了老……不过在这个场景中总会有领导批复的淡定
                System.out.println("Goodbye my money......");
            }
        }
    }

    /**
     * 为当前领导设置一个上级领导
     *
     * @param superiorLeader
     *            上级领导
     */
    public void setLeader(Leader superiorLeader) {
        this.mSuperiorLeader = superiorLeader;
    }
}
```

这么一来，我们的领导老大们就有了实实在在的权利职责去处理底层苦逼程序猿的请求。OK，接下来要做的事就是让所有的领导继承该类：

```
/**
 * 小组长类
 *
 * @author Aige{@link https://github.com/AigeStudio}
 *
 */
public class GroupLeader extends Leader {

    public GroupLeader() {
        super(1000);
    }
```

```
    }

    @Override
    protected void reply(ProgramApe ape) {
        System.out.println(ape.getApply());
        System.out.println("GroupLeader: Of course Yes!");
    }
}
```

```
/**
 * 项目主管类
 *
 * @author Aige{@link https://github.com/AigeStudio}
 *
 */
public class Director extends Leader{
    public Director() {
        super(5000);
    }

    @Override
    protected void reply(ProgramApe ape) {
        System.out.println(ape.getApply());
        System.out.println("Director: Of course Yes!");
    }
}
```

```
/**
 * 部门经理类
 *
 * @author Aige{@link https://github.com/AigeStudio}
 *
 */
public class Manager extends Leader {
    public Manager() {
        super(10000);
    }

    @Override
    protected void reply(ProgramApe ape) {
        System.out.println(ape.getApply());
        System.out.println("Manager: Of course Yes!");
    }
}
```

```
/**
 * 老总类
 *
 * @author Aige{@link https://github.com/AigeStudio}
 *
 */
public class Boss extends Leader {
    public Boss() {
        super(40000);
    }

    @Override
    protected void reply(ProgramApe ape) {
        System.out.println(ape.getApply());
        System.out.println("Boss: Of course Yes!");
    }
}
```

最后，更新我们的场景类，将其从责任人的角色中解放出来：

```
/**
 * 场景模拟类
 *
 * @author Aige{@link https://github.com/AigeStudio}
 *
 */
```

```
public class Client {
    public static void main(String[] args) {
        /*
         * 先来一个程序猿 这里给他一个三万以内的随机值表示需要申请的差旅费
         */
        ProgramApe ape = new ProgramApe((int) (Math.random() * 30000));

        /*
         * 再来四个老大
         */
        Leader leader = new GroupLeader();
        Leader director = new Director();
        Leader manager = new Manager();
        Leader boss = new Boss();

        /*
         * 设置老大的上一个老大
         */
        leader.setLeader(director);
        director.setLeader(manager);
        manager.setLeader(boss);

        // 处理申请
        leader.handleRequest(ape);
    }
}
```

运行三次，下面是三次运行的结果（注：由于随机值的原因你的结果也许与我不一样）：

> 爹要点钱出差
>
> Boss: Of course Yes!
>
> ---
>
> 爹要点钱出差
>
> Director: Of course Yes!
>
> ---
>
> 爹要点钱出差
>
> Boss: Of course Yes!

总结

OK，这样我们就将请求和处理分离开来，对于程序猿来说，不需要知道是谁给他批复的钱，而对于领导们来说，也不需要确切地知道是批给哪个程序猿，只要根据自己的责任做出处理即可，由此将两者优雅地解耦。

Android源码中的模式实现

Android中关于责任链模式比较明显的体现就是在事件分发过程中对事件的投递，其实严格来说，事件投递的模式并不是严格的责任链模式，但是其是责任链模式的一种变种体现，在ViewGroup中对事件处理者的查找方式如下：

```
@Override
public boolean dispatchTouchEvent(MotionEvent ev) {
    // 省略两���代码……

    boolean handled = false;
    if (onFilterTouchEventForSecurity(ev)) {

        // 省略N行代码……

        /*
         * 如果事件未被取消并未被拦截
         */
        if (!canceled && !intercepted) {
            /*
             * 如果事件为起始事件
             */
```

```
        if (actionMasked == MotionEvent.ACTION_DOWN
                || (split && actionMasked == MotionEvent.ACTION_POINTER_DOWN)
                || actionMasked == MotionEvent.ACTION_HOVER_MOVE) {

            // 省掉部分逻辑.....

            final int childrenCount = mChildrenCount;

            /*
             * 如果TouchTarget为空并且子元素不为0
             */
            if (newTouchTarget == null && childrenCount != 0) {
                final float x = ev.getX(actionIndex);
                final float y = ev.getY(actionIndex);

                final View[] children = mChildren;

                final boolean customOrder = isChildrenDrawingOrderEnabled();

                /*
                 * 遍历子元素
                 */
                for (int i = childrenCount - 1; i >= 0; i--) {
                    final int childIndex = customOrder ?
                            getChildDrawingOrder(childrenCount, i) : i;
                    final View child = children[childIndex];

                    /*
                     * 如果这个子元素无法接收Pointer Event或这个事件点压根就没有落在子元素的边界范围内
                     */
                    if (!canViewReceivePointerEvents(child)
                            || !isTransformedTouchPointInView(x, y, child, null)) {
                        // 那么就跳出该次循环继续遍历
                        continue;
                    }

                    // 找到Event该由哪个子元素持有
                    newTouchTarget = getTouchTarget(child);


                    if (newTouchTarget != null) {
                        newTouchTarget.pointerIdBits |= idBitsToAssign;
                        break;
                    }

                    resetCancelNextUpFlag(child);

                    /*
                     * 投递事件执行触摸操作
                     * 如果子元素还是一个ViewGroup则递归调用重复此过程
                     * 如果子元素是一个View那么则会调用View的dispatchTouchEvent并最终由onTouchEvent处理
                     */
                    if (dispatchTransformedTouchEvent(ev, false, child, idBitsToAssign)) {
                        mLastTouchDownTime = ev.getDownTime();
                        mLastTouchDownIndex = childIndex;
                        mLastTouchDownX = ev.getX();
                        mLastTouchDownY = ev.getY();
                        newTouchTarget = addTouchTarget(child, idBitsToAssign);
                        alreadyDispatchedToNewTouchTarget = true;
                        break;
                    }
                }
            }

            /*
             * 如果发现没有子元素可以持有该次事件
             */
            if (newTouchTarget == null && mFirstTouchTarget != null) {
                newTouchTarget = mFirstTouchTarget;
                while (newTouchTarget.next != null) {
```

```
                    while (newTouchTarget.next != null) {
                        newTouchTarget = newTouchTarget.next;
                    }
                    newTouchTarget.pointerIdBits |= idBitsToAssign;
                }
            }
        }

        // 省去不必要代码……
    }

    // 省去一行代码……

    return handled;
}
```

再来看看dispatchTransformedTouchEvent方法是如何调度子元素dispatchTouchEvent方法的：

```
private boolean dispatchTransformedTouchEvent(MotionEvent event, boolean cancel,
        View child, int desiredPointerIdBits) {
    final boolean handled;

    final int oldAction = event.getAction();

    /*
     * 如果事件被取消
     */
    if (cancel || oldAction == MotionEvent.ACTION_CANCEL) {
        event.setAction(MotionEvent.ACTION_CANCEL);

       /*
        * 如果没有子元素
        */
        if (child == null) {
            // 那么就直接调用父类的dispatchTouchEvent注意这里的父类终会为View类
            handled = super.dispatchTouchEvent(event);
        } else {
            // 如果有子元素则传递cancle事件
            handled = child.dispatchTouchEvent(event);
        }
        event.setAction(oldAction);
        return handled;
    }

    /*
     * 计算即将被传递的点的数量
     */
    final int oldPointerIdBits = event.getPointerIdBits();
    final int newPointerIdBits = oldPointerIdBits & desiredPointerIdBits;

    /*
     * 如果事件木有相应的点那么就丢弃该次事件
     */
    if (newPointerIdBits == 0) {
        return false;
    }

    // 声明临时变量保存坐标转换后的MotionEvent
    final MotionEvent transformedEvent;

    /*
     * 如果事件点的数量一致
     */
    if (newPointerIdBits == oldPointerIdBits) {
        /*
         * 子元素为空或子元素有一个单位矩阵
         */
        if (child == null || child.hasIdentityMatrix()) {
            /*
             * 再次区分子元素为空的情况
```

```
            * 将状态分子元素为空的情况
             */
            if (child == null) {
                // 为空则调用父类dispatchTouchEvent
                handled = super.dispatchTouchEvent(event);
            } else {
                // 否则尝试获取xy方向上的偏移量（如果通过scrollTo或scrollBy对子视图进行滚动的话）
                final float offsetX = mScrollX - child.mLeft;
                final float offsetY = mScrollY - child.mTop;

                // 将MotionEvent进行坐标变换
                event.offsetLocation(offsetX, offsetY);

                // 再将变换后的MotionEvent传递给子元素
                handled = child.dispatchTouchEvent(event);

                // 复位MotionEvent以便之后再次使用
                event.offsetLocation(-offsetX, -offsetY);
            }

            // 如果通过以上的逻辑判断当前事件被持有则可以直接返回
            return handled;
        }
        transformedEvent = MotionEvent.obtain(event);
    } else {
        transformedEvent = event.split(newPointerIdBits);
    }

    /*
     * 下述雷同不再累赘
     */
    if (child == null) {
        handled = super.dispatchTouchEvent(transformedEvent);
    } else {
        final float offsetX = mScrollX - child.mLeft;
        final float offsetY = mScrollY - child.mTop;
        transformedEvent.offsetLocation(offsetX, offsetY);
        if (! child.hasIdentityMatrix()) {
            transformedEvent.transform(child.getInverseMatrix());
        }

        handled = child.dispatchTouchEvent(transformedEvent);
    }

    transformedEvent.recycle();
    return handled;
}
```

ViewGroup事件投递的递归调用就类似于一条责任链，一旦其寻找到责任者，那么将由责任者持有并消费掉该次事件，具体的体现在View的onTouchEvent方法中返回值的设置（这里介于篇幅就不具体介绍ViewGroup对事件的处理了），如果onTouchEvent返回false那么意味着当前View不会是该次事件的责任人将不会对其持有，如果为true则相反，此时View会持有该事件并不再向外传递。

4. 杂谈

世界不是完美的，所以不会有完美的事物存在。就像所有的设计模式一样， 有优点优缺点，但是总的来说优点必定大于缺点或者说缺点相对于优点来说更可控。责任链模式也一样，有点显而易见，可以对请求者和处理者关系的解耦提高代码的灵活性，比如上面我们的例子中如果在主管和经理之间多了一个总监，那么总监可以批复小于7500的经费，这时候根据我们上面重构的模式，仅需新建一个总监类继承Leader即可其它所有的存在类都可保持不变。责任链模式的最大缺点是对链中责任人的遍历，如果责任人太多那么遍历必定会影响性能，特别是在一些递归调用中，要慎重。