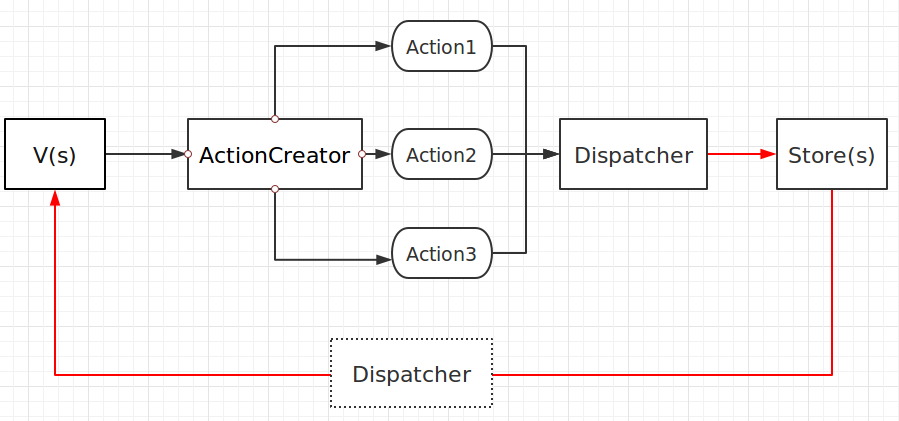
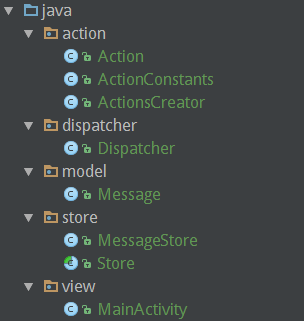
一、简单使用:



上述是大体的流程，红色的线指的是分发，一般采取EventBus、Otto这些事件分发总线，也可以使用接口回调。简单描述一下流程：

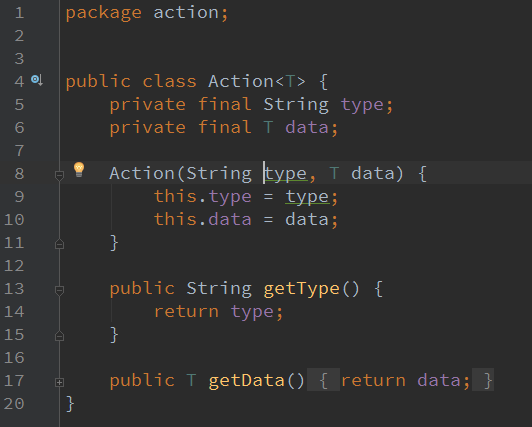
"View通过ActionCreator产生各种Action，再通过Dispatcher分发给store去处理，再通过Dispatcher分发回View"

上图中含有5个元素，没错示例的DEMO结构也分成了5部分：



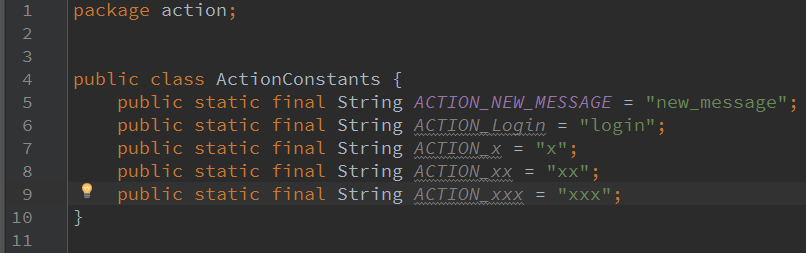
步骤一(Action):

首先Action被设计出的主要功能是标识各种操作并连接各模块的，额外还能携带数据，所以：



接着是ActionConstants常量类，一般网上的demo会继承Action类来具体实现各种Action如MessageAction,LoginAction，

而使用此常量类就不需要为每个操作去实例多一个类：



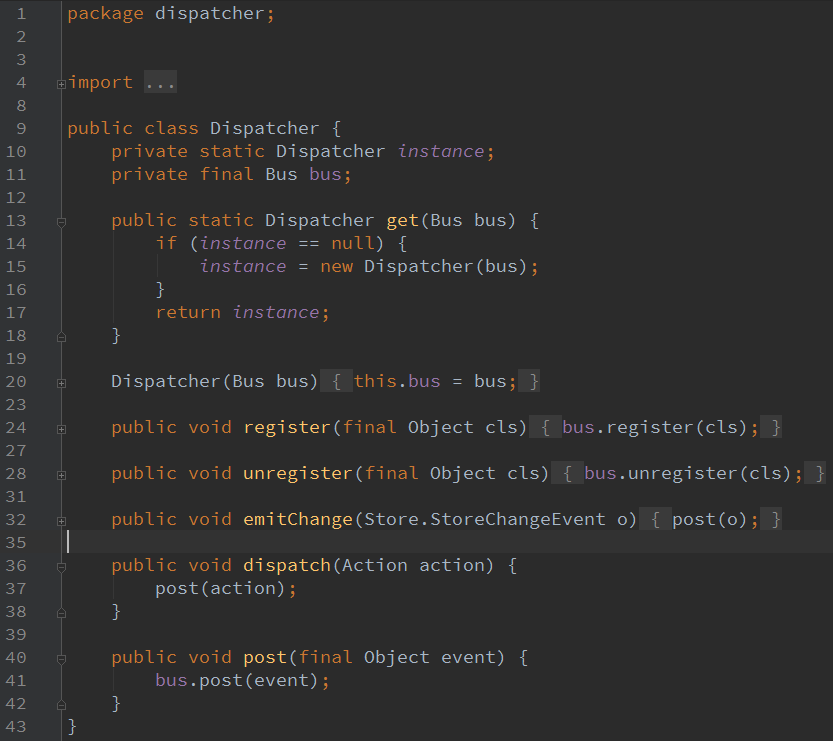
然后Action有个管理辅助类，让V去调用的，也就是上述流程图的ActionCreator，整个项目只需要一个所以写成了单例的:



步骤二(Dispatcher):

顾名思义“分发”，一般采取事件总线，整个流程有两次分发，有些demo在第一次分发的时候，采取在Dispatcher中维护一个List<store>,在store中使用事件总线，但个人觉得事件总线

作为具体的分发实现，因此应放在Dispatcher中，同时整个项目中只需用到一个所以写成了单例：



其实可以看出emitChange和dispatch 都是调用Post方法进行分发，这里是为了区分第一二次分发才这样写的。

步骤三（Model）：

Model类根据需求来进行编写，此Demo操作一个String，所以Message这个类 只有简单的一个String变量以及其get、set方法，

这里就不贴上去了。

步骤四（Store）:

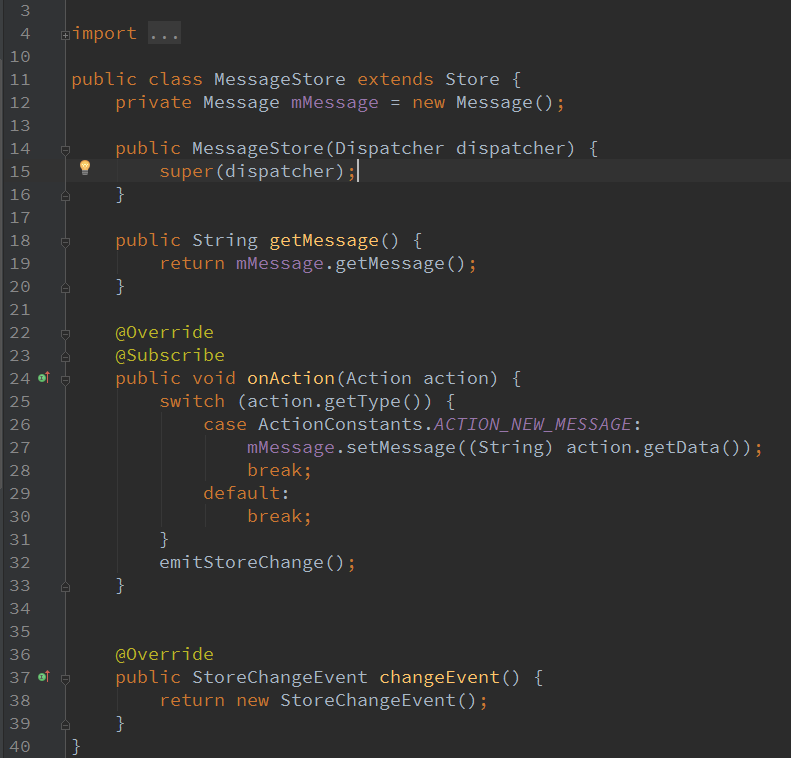
首先想，Store是通过事件总线接收数据，自身处理数据，再通过事件总线返回给View。

根据上述，Store需具备：1.引用Dispatcher（返回数据） 2.一个方法接收数据，而且Store需实现为各种操作Store，因此抽象Store最佳：



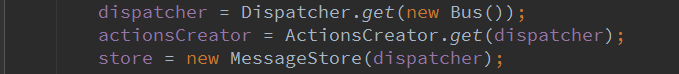
onAction即为接受数据的方法，而引用Dispatcher写在了构造方法中，同时还有个空的类StoreChangeEvent以及抽象方法changeEvent,

仅用作Store返回给View的一个标识。之所以changeEvent定义为抽象，让实现类来实现，这样一来就可以细化对UI的控制。



步骤五（View）：

接下来写View，在View中你需使用ActionsCreator，而它的构造方法需要Dispatcher,而Store实现类也需要Dispatcher,所以:



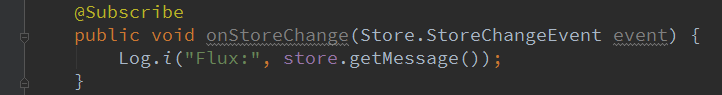
然后事件总线你要注册啊，谁接收谁注册，在整个流程中，store和view本身需要接收，所以：

使用二

在初始化这些后，你可以使用了：

使用三

发出Action后，Store处理返回：



---------------------------------------我是分割线------------------------------------------

其实这只是一个思想，实现编写处理出来的代码会因人而异，后续也试试有没啥优化改进的地方。