这是一个极简的HelloWorld应用，主要用来展示如何在Android平台架构Flux应用。并提供一些基础代码，方便开发者直接Copy这些代码到自己的工程中，省掉重新造轮子的过程。接下来会一步步的解释这个应用是如何构建的。

Demo程序是用AndroidStudio开发的，假设你已经了解Android和AndroidStudioIDE，如果你已经很熟悉Android应用的开发，看完AndroidFlux一览或许已经可以开发出基于Flux框架的应用，如果你并不熟悉Flux或者Android，务必先读完这篇文档

源码结构

本着架构即目录的思想，让我们先看一下源码结构，整个的源码结构是这样的：

```
➜  tree .
.
├── MainActivity.java
├── actions
|   ├── Action.java
|   ├── ActionsCreator.java
|   └── MessageAction.java
├── dispatcher
|   └── Dispatcher.java
├── model
|   └── Message.java
└── stores
    ├── MessageStore.java
    └── Store.java
```

这里包含4个目录和一个文件：

1. MainActivity.java Flux框架中的Controller-View部分，在Android中可以是Activity或者Fragment
2. actions Flux框架中的Action部分，存放不同类型的`XXXAction.java`和`ActionsCreator.java`文件
3. dispatcher Flux框架中的Dispatcher部分，存放`Dispatcher.java`文件，一个应用中只需要一个Dispatcher
4. model 存放各种业务逻辑相关的Model文件
5. stores Flux框架中的Stores部分，存在各种类型的`XXXStore.java`文件

创建一个Dispatcher

在AndroidFlux中Dispatcher是就是一个发布-订阅模式。Store会在这里注册自己的回调接口，Dispatcher会把Action分发到注册的Store，所以它会提供一些公有方法来注册监听和分发消息。

```
/**
 * Flux的Dispatcher模块
 * Created by ntop on 18/12/15.
 */
public class Dispatcher {
    private static Dispatcher instance;
    private final List<Store> stores = new ArrayList<>();

    public static Dispatcher get() {
        if (instance == null) {
            instance = new Dispatcher();
        }
        return instance;
    }

    Dispatcher() {}

    public void register(final Store store) {
        stores.add(store);
    }

    public void unregister(final Store store) {
        stores.remove(store);
    }

    public void dispatch(Action action) {
        post(action);
    }

    private void post(final Action action) {
        for (Store store : stores) {
            store.onAction(action);
        }
    }
}
```

Dispatcher对外仅暴露3个公有方法：

1. register(final Store store) 用来注册每个Store的回调接口
2. unregister(final Store store) 用来接触Store的回调接口
3. dispatch(Action action) 用来触发Store注册的回调接口

这里仅仅用一个`ArrayList`来管理Stores，对于一个更复杂的App可能需要精心设计数据结构来管理Stores组织和相互间的依赖关系。

创建Stores

这里使用EventBus来实现Store，EventBus的主要功能是用来给Controller-View发送`change`事件：

```
/**
 * Flux的Store模块
 * Created by ntop on 18/12/15.
 */
public abstract class Store {
    private  static final Bus bus = new Bus();

    protected Store() {
    }

    public void register(final Object view) {
        this.bus.register(view);
    }

    public void unregister(final Object view) {
        this.bus.unregister(view);
    }

    void emitStoreChange() {
        this.bus.post(changeEvent());
    }

    public abstract StoreChangeEvent changeEvent();
    public abstract void onAction(Action action);

    public class StoreChangeEvent {}
}
```

抽象的Store类，提供了一个主要的虚方法 `void onAction(Action action)` ，这个方法是注册在Dispatcher里面的回调接口，当Dispatcher有数据派发过来的时候，可以在这里处理。

下面看一下更具体的和业务相关的MessageStore类:

```
/**
 * MessageStore类主要用来维护MainActivity的UI状态
 * Created by ntop on 18/12/15.
 */
public class MessageStore extends Store {
    private static MessageStore singleton;
    private Message mMessage = new Message();

    public MessageStore() {
        super();
    }

    public String getMessage() {
        return mMessage.getMessage();
    }

    @Override
    @Subscribe
    public void onAction(Action action) {
        switch (action.getType()) {
            case MessageAction.ACTION_NEW_MESSAGE:
                mMessage.setMessage((String) action.getData());
                break;
            default:
        }
        emitStoreChange();
    }


    @Override
    public StoreChangeEvent changeEvent() {
        return new StoreChangeEvent();
    }
}
```

在这里实现了 `onAction(Action action)` 方法，并用一个 `switch` 语句来路由各种不同的Action类型。同时维护了一个结构 `Message.java` 类，这个类用来记录当前要显示的消息。Store类只能通过Dispatcher来更新（不要提供 `setter` 方法），对外仅暴露各种 `getter` 方法来获取UI状态。这里用 `String getMessage()` 方法来获取具体的消息。

在Controller-View里面处理“change”事件

在Android中，Flux的Controller-View对应于Activity或者Fragment，我们需要在这里注册Strore发生改变的事件通知，以便在Store变化的时候重新绘制UI。

```
/**
 * Flux的Controller-View模块
 * Created by ntop on 18/12/15.
```

```
 */
public class MainActivity extends AppCompatActivity implements
View.OnClickListener {
    private EditText vMessageEditor;
    private Button vMessageButton;
    private TextView vMessageView;

    private Dispatcher dispatcher;
    private ActionsCreator actionsCreator;
    private MessageStore store;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        initDependencies();
        setupView();
    }

    @Override
    protected void onDestroy() {
        super.onDestroy();
        dispatcher.unregister(store);
    }

    private void initDependencies() {
        dispatcher = Dispatcher.get();
        actionsCreator = ActionsCreator.get(dispatcher);
        store = new MessageStore();
        dispatcher.register(store);
    }

    private void setupView() {
        vMessageEditor = (EditText) findViewById(R.id.message_editor);
        vMessageView = (TextView) findViewById(R.id.message_view);
        vMessageButton = (Button) findViewById(R.id.message_button);
        vMessageButton.setOnClickListener(this);
    }

    @Override
    public void onClick(View view) {
        int id = view.getId();
        if (id == R.id.message_button) {
            if (vMessageEditor.getText() != null) {

actionsCreator.sendMessage(vMessageEditor.getText().toString());
                vMessageEditor.setText(null);
            }
        }
    }
```

```
    private void render(MessageStore store) {
        vMessageView.setText(store.getMessage());
    }

    @Override
    protected void onResume() {
        super.onResume();
        store.register(this);
    }

    @Override
    protected void onPause() {
        super.onPause();
        store.unregister(this);
    }

    @Subscribe
    public void onStoreChange(Store.StoreChangeEvent event) {
        render(store);
    }
}
```

这部分的代码比较多，首先在 `onCreatre(...)` 方法中初始化了依赖和需要的UI组件。最重要的是 `onStoreChange(...)` 方法，这个方法是注册在Store中回调（使用EventBus的 `@Subscribe` 注解标识），当Store发生变化的时候会触发这个方法，我们在这里调用 `render()` 方法重绘整个界面。

创建Action

Action是简单的POJO类型，只提供两个字段： `type` 和 `data`，分别记录Action的类型和数据。注意Action一旦创建是不可更改的， 所以它的字段类型修饰为 `final` 类型。

```
public class Action<T> {
    private final String type;
    private final T data;

    Action(String type, T data) {
        this.type = type;
        this.data = data;
    }

    public String getType() {
        return type;
    }

    public T getData() {
        return data;
    }
}
```

下面是一个业务相关的Action实现：

```
public class MessageAction extends Action<String> {
    public static final String ACTION_NEW_MESSAGE = "new_message";

    MessageAction(String type, String data) {
        super(type, data);
    }
}
```

这个实现非常简单，仅仅多定义了一个Action类型字段：`public static final String ACTION_NEW_MESSAGE = "new_message"`。如你所见，Action都是这么简单的，不包含任何业务逻辑。

创建ActionCreator

ActionCreator 是Flux架构中第“四”个最重要的模块（前三：Dispatcher、Store、View），这里实际上处理很多工作，提供有一个有语义的API，构建Action，处理网络请求等。

```
/**
 * Flux的ActionCreator模块
 * Created by ntop on 18/12/15.
 */
public class ActionsCreator {

    private static ActionsCreator instance;
    final Dispatcher dispatcher;

    ActionsCreator(Dispatcher dispatcher) {
        this.dispatcher = dispatcher;
    }

    public static ActionsCreator get(Dispatcher dispatcher) {
        if (instance == null) {
            instance = new ActionsCreator(dispatcher);
        }
        return instance;
    }

    public void sendMessage(String message) {
        dispatcher.dispatch(new
MessageAction(MessageAction.ACTION_NEW_MESSAGE, message));
    }
}
```

此处提供了一个 `sendMessage(String message)` ，就像名字暗示的那样，这个方法用来发送消息（到Store）。在方法内部，会创建一个`MessageAction`来封装数据和Action类型，并通过Dispatcher发送到Store。

Model

无论是基于哪种框架的应用都需要Model模块，在这个简单的“HelloWorld”应用中，其实用一个String即可传递消息，但是为了架构的完整和更好的语义表达，定义一个Message类型封装一个String字段作为Model。

希望通过这个简单的HelloWorld应用，能够让你一窥Flux的面貌。如果你想更深入的了解在Android平台上应用Flux架构，可以查看我们的Github站点。