EventBus学习

03 关于EventBus的概述

# 1.三要素

* Event 事件。它可以是任意类型。
* Subscriber 事件订阅者。在EventBus3.0之前我们必须定义以onEvent开头的那几个方法，分别是onEvent、onEventMainThread、onEventBackgroundThread和onEventAsync，而在3.0之后事件处理的方法名可以随意取，不过需要加上注解@subscribe()，并且指定线程模型，默认是POSTING。
* Publisher 事件的发布者。我们可以在任意线程里发布事件，一般情况下，使用EventBus.getDefault()就可以得到一个EventBus对象，然后再调用post(Object)方法即可。

**DevRing & Demo地址**：<https://github.com/LJYcoder/DevRing>

学习/参考地址：   
<http://blog.csdn.net/itachi85/article/details/52205464>   
<http://blog.csdn.net/Tencent_Bugly/article/details/51354693>   
<http://blog.csdn.net/qq_28746251/article/details/51476389>

# 2.前言

EventBus是一个基于发布/订阅的事件总线（数据通信框架），它简化了组件之间、线程之间的数据通信操作，并且耦合度低、开销小。   
3.0版本后，使用注解来声明订阅者函数及其相关属性，使得操作流程更加便捷，还提供index帮助提升其性能。

（如果你不喜欢用EventBus，而想用RxJava自己封装一个RxBus来实现通信，   
可以参考<http://www.jianshu.com/p/3a3462535b4d>）

# 3.介绍

下面从 **配置、基本使用、粘性事件、使用index优化、简单封装、混淆** 这几个部分来介绍EventBus。

## 1. 配置

在Module下的build.gradle中添加

//EventBus

compile 'org.greenrobot:eventbus:3.0.0'

annotationProcessor 'org.greenrobot:eventbus-annotation-processor:3.0.1'//用于eventbus开启Index加速

## 2. 基本使用

使用步骤分为定义事件、订阅事件、发送事件、处理事件、取消订阅五步

## 2.1 定义事件

先定义一个你打算发送的事件类，里面添加你要发送的数据变量。   
变量的类型除了基本数据类型，也可以是自定义的实体类。

public class MovieEvent {

private int count;

public MovieEvent(int count) {

this.count = count;

}

public int getCount() {

return count;

}

public void setCount(int count) {

this.count = count;

}

}

## 2.2 订阅事件

在你要接收事件的地方订阅事件：

public class MovieActivity{

....

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

//确保之前未订阅过，再调用订阅语句，以免报错

if(!EventBus.getDefault().isRegistered(subscriber)){

EventBus.getDefault().register(subscriber);

}

}

....

}

## 2.3 发送事件

在你要发送事件的地方，调用

EventBus.getDefault().post(new MovieEvent(1));

这里需要注意，发送的事件是属于引用传递，也就是说，发送事件后，你在事件处理函数中对接收到的事件进行了修改，那么发送源头的事件也会跟着改变。所以如果不想影响到发送源头的数据，建议new对象后再发送。

另外，EventBus提供了一个方法用于发送粘性事件，粘性事件相关内容会在下面另外介绍。

EventBus.getDefault().postSticky(new MovieEvent(1));

## 2.4 处理事件

在接收事件的地方添加处理事件的方法，请与订阅事件方法处于同一个类下，以保证能成功订阅事件

public class MovieActivity{

....

//声明处理事件的方法

@Subscribe

public void handlerEvent(MovieEvent event) {

//处理事件

int count = event.getCount();

...

}

....

}

你可以自定义处理事件方法的名称，但必须加上@Subscribe注解来声明该方法为事件接收处理方法。   
方法的参数用于指定你要接收事件类型。比如参数为MovieEvent event，则表示接收类型为MovieEvent的事件，其他类型的事件将不会接收到。

另外@Subscribe里面可以对 **处理事件时所在的线程、事件接收的优先级、是否为粘性事件** 进行设置

* 处理事件时所在的线程

@Subscribe(threadMode = 线程类型)

* 1

线程类型有以下四种选择：   
1. ThreadMode.POSTING：默认的类型。表示处理事件时所在的线程将会与事件发送所在的线程一致，也就是两者的执行都处于同一个线程。   
2. ThreadMode.MAIN：表示处理事件时所在的线程将切换为UI主线程。如果发送事件所在的线程本来就是UI主线程，则不会切换。   
3. ThreadMode.BACKGROUND：表示处理事件时所在的线程将切换为后台线程。如果发送事件所在的线程本来就是后台线程，则不会切换。   
4. ThreadMode.ASYNC：表示处理事件时所在的线程将会切换为一个新建的独立子线程。

* 优先级

@Subscribe(priority = 100)

priority用于指定接收事件的优先级，默认值为0。   
优先级高的事件处理函数将先收到发送的事件，你可以在优先级高的事件处理函数中拦截事件，不让它继续往下传递，拦截方法如下

EventBus.getDefault().cancelEventDelivery(event);

* 粘性事件

@Subscribe(sticky = true)

sticky用来声明是否接收订阅前就已发出的粘性事件，默认值为false，具体介绍请看后面的“粘性事件”

### 2.5 取消订阅

在退出或者不需要接收事件时，取消订阅

public class MovieActivity{

....

@Override

protected void onDestroy() {

super.onDestroy();

//取消订阅

if (EventBus.getDefault().isRegistered(subscriber)) {

EventBus.getDefault().unregister(subscriber);

}

}

....

}

## 3. 粘性事件

一般我们的使用流程为：普通事件：订阅事件—》发送事件—》接收处理事件。   
那如果现在希望：粘性消息：发送事件—》订阅事件—》接收处理事件，   
EventBus提供的粘性事件便可实现这一场景。

### 3.1 使用步骤

使用步骤和普通事件的基本一样，但事件的发送调用的是postSticky(event)，事件处理函数需声明@Subscriber(sticky = true)。

//事件发送方

//发送粘性事件

EventBus.getDefault().postSticky(new MovieEvent(1));

//事件接收处理方

//发送完粘性事件后再进行订阅事件

EventBus.getDefault().register(this);//注册事件接收

//接收处理订阅前发出的粘性事件

@Subscribe(threadMode = ThreadMode.MAIN, sticky = true)

public void handleEvent(MovieEvent event) {

//处理事件

int count = event.getCount();

}

### 3.2 使用场景

这里举一个使用场景：Activity间跳转传值。参考自[http://www.cnblogs.com/ldq2016/p/5387444.html](http://www.cnblogs.com/ldq2016/p/5387444.html" \t "_blank)   
我们平时都是使用Intent携带数据来实现，如果要传递的是自定义的实体类，还需要进行序列化操作。下面大致演示如何使用EventBus的粘性事件来实现这一场景。

ActivityA跳转到ActivityB，并将Movie对象传递过去。

//ActivityA中的代码

Movie movie = new Movie();

//发送粘性事件，传送movie

EventBus.getDefault().postSticky(movie);

//跳转到ActivityB

startActivity(new Intent(this, ActivityB.class));

//ActivityB中的代码

//订阅事件

EventBus.getDefault().register(this);

//获取订阅前ActivityA发送的粘性事件

@Subscribe(sticky = true)

public void getDataFromOtherActivity(Movie movie) {

//得到ActivityA的Movie对象，进行具体操作。

}

如果大家还有其他使用场景，欢迎留言分享~

## 3.3 补充

1）EventBus仅保存最新发送的粘性事件。   
2）手动获取、移除粘性事件

//手动获取粘性事件

MovieEvent movieEvent = EventBus.getDefault().getStickyEvent(MovieEvent.class);

if(movieEvent != null) {

//移除粘性事件

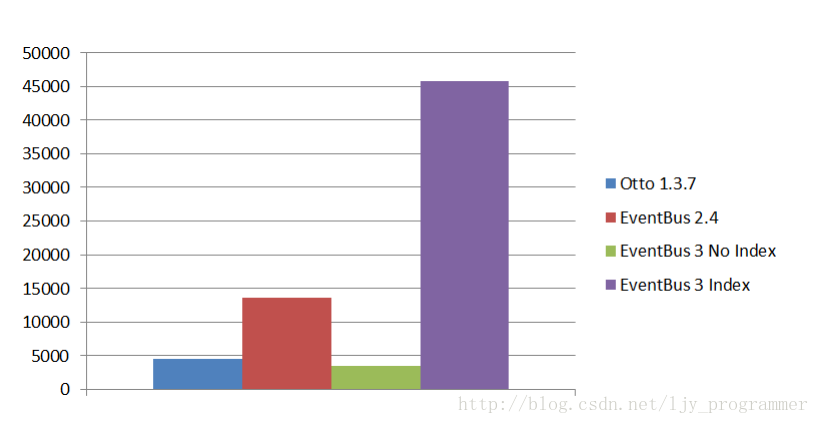
EventBus.getDefault().removeStickyEvent(movieEvent);

}

3）   
发送粘性事件后，对于在发送前就已经订阅事件的订阅者，它们都会收到类型相符的粘性事件，不管它们的事件处理方法是否声明为sticky=true。 而对于在发送后才进行订阅事件的订阅者，其事件处理方法必须声明为sticky=true才能收到类型相符的粘性事件。

## 4. 使用index优化

在3.0版本之前，为了保证性能，EventBus在遍历寻找订阅者的回调方法时使用反射而不是注解，而在3.0版本由于采用注解，从下图可以看到，其性能比2.4版本要下降很多。   
为了在使用注解的情况下保证高性能，EventBus提供了通过开启Index来提升性能的方法，从下图可以看到，开启了Index之后，其性能提高了许多倍。



下面介绍如何生成和开启索引。

### 4.1 生成索引

网上很多关于Index的文章中，是通过添加以下配置来生成的

//project下的build.gradle文件

buildscript {

dependencies {

classpath 'com.neenbedankt.gradle.plugins:android-apt:1.8'

}

}

//module下的build.gradle文件

apply plugin: 'com.neenbedankt.android-apt'

apt {

arguments {

eventBusIndex "com.dev.base.MyEventBusIndex"

}

}

dependencies {

compile 'org.greenrobot:eventbus:3.0.0'

apt 'org.greenrobot:eventbus-annotation-processor:3.0.1'

}

但是，如果你的项目中也使用了ButterKnife库，那么添加上面的配置后会导致ButterKnife无法正常工作；另外，android-apt的作者已在官网发表声明表示后续将不会继续维护android-apt，所以并不推荐这个方式实现，而是使用Android推出的官方插件annotationProcessor来替代apt。   
通过annotationProcessor来设置生成index的配置会更加便捷，如下：

//module下的build.gradle文件

android{

defaultConfig {

//...省略其他配置

javaCompileOptions {

annotationProcessorOptions {

arguments = [ eventBusIndex : "com.dev.base.MyEventBusIndex" ]

}

}

}

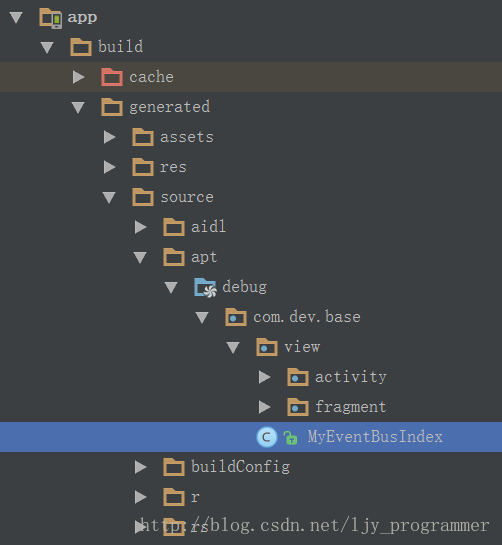
}

dependencies {

compile 'org.greenrobot:eventbus:3.0.0'

annotationProcessor 'org.greenrobot:eventbus-annotation-processor:3.0.1'

}

添加以上配置后，编译 **运行** 项目后，如果在\build\generated\source\apt\debug\项目包名\下生成了你指定的Index类，则表示生成index成功，如下图所示。   


### 4.2 开启索引

生成index之后，在Appliction中调用addIndex()方法开启即可。

EventBus.builder().addIndex(new MyEventBusIndex()).installDefaultEventBus();

## 5. 简单封装

public class EventBusManager {

//开启Index加速

public static void openIndex() {

EventBus.builder().addIndex(new MyEventBusIndex()).installDefaultEventBus();

}

//订阅事件

public static void register(Object subscriber) {

if(!EventBus.getDefault().isRegistered(subscriber)){

EventBus.getDefault().register(subscriber);

}

}

//取消订阅

public static void unregister(Object subscriber) {

if (EventBus.getDefault().isRegistered(subscriber)) {

EventBus.getDefault().unregister(subscriber);

}

}

//终止事件继续传递

public static void cancelDelivery(Object event) {

EventBus.getDefault().cancelEventDelivery(event);

}

//获取保存起来的粘性事件

public static <T> T getStickyEvent(Class<T> classType){

return EventBus.getDefault().getStickyEvent(classType);

}

//删除保存中的粘性事件

public static void removeStickyEvent(Object event) {

EventBus.getDefault().removeStickyEvent(event);

}

//发送事件

public static void postEvent(Object event){

EventBus.getDefault().post(event);

}

//发送粘性事件

public static void postStickyEvent(Object event) {

EventBus.getDefault().postSticky(event);

}

}

DevRing/Demo中已对EventBus进行了封装，在Activity和Fragment中，只需重写isUseEventBus()方法并返回true，则会自动对该页面进行订阅和解除订阅的操作，详情请查阅代码。

## 6. 混淆

在proguard-rules.pro文件中添加以下内容进行混淆配置

#EventBus开始

-keepattributes \*Annotation\*

#如果使用了EventBus index进行优化加速，就必须加上这个

-keepclassmembers class \*\* {

@org.greenrobot.eventbus.Subscribe <methods>;

}

-keep enum org.greenrobot.eventbus.ThreadMode { \*; }

#如果使用了Async类型的线程

-keepclassmembers class \* extends org.greenrobot.eventbus.util.ThrowableFailureEvent {

<init>(java.lang.Throwable);

}

#EventBus结束

# 7. 下面介绍EventBus的原理。

源码大家可以根据原理自己观摩学习，对于砖家，你可以直接搬砖使用，但学习源码，能让你更懂得如何搬砖。各人自由发挥吧。  
1、想成为订阅者，需要进行注册

EventBus.getDefault().register(this);

那么先从getDefault()方法走起，定睛一看，so easy,老司机都懂，不解释。

public static EventBus getDefault() {

if (defaultInstance == null) {

synchronized (EventBus.class) {

if (defaultInstance == null) {

defaultInstance = new EventBus();

}

}

}

return defaultInstance;

}

获取实例直接调用了EventBus构造方法

public EventBus() {

this(DEFAULT\_BUILDER);

}

private final Map<Class<?>, CopyOnWriteArrayList<Subscription>> subscriptionsByEventType;

private final Map<Object, List<Class<?>>> typesBySubscriber;

private final Map<Class<?>, Object> stickyEvents;

private final HandlerPoster mainThreadPoster;

private final BackgroundPoster backgroundPoster;

private final AsyncPoster asyncPoster;

EventBus(EventBusBuilder builder) {

//key为订阅事件 value为订阅这个事件的所有订阅者的CopyOnWriteArrayList

subscriptionsByEventType = new HashMap<>();

//以订阅者的类为key，以event事件类为value，在进行register或unregister的时候，会操作这HashMap。

typesBySubscriber = new HashMap<>();

//粘性事件的集合

stickyEvents = new ConcurrentHashMap<>();

//事件处理

mainThreadPoster = new HandlerPoster(this, Looper.getMainLooper(), 10);

backgroundPoster = new BackgroundPoster(this);

asyncPoster = new AsyncPoster(this);

......

}

register()方法：

public void register(Object subscriber) {

Class<?> subscriberClass = subscriber.getClass();

List<SubscriberMethod> subscriberMethods = subscriberMethodFinder.findSubscriberMethods(subscriberClass);

synchronized (this) {

for (SubscriberMethod subscriberMethod : subscriberMethods) {

subscribe(subscriber, subscriberMethod);

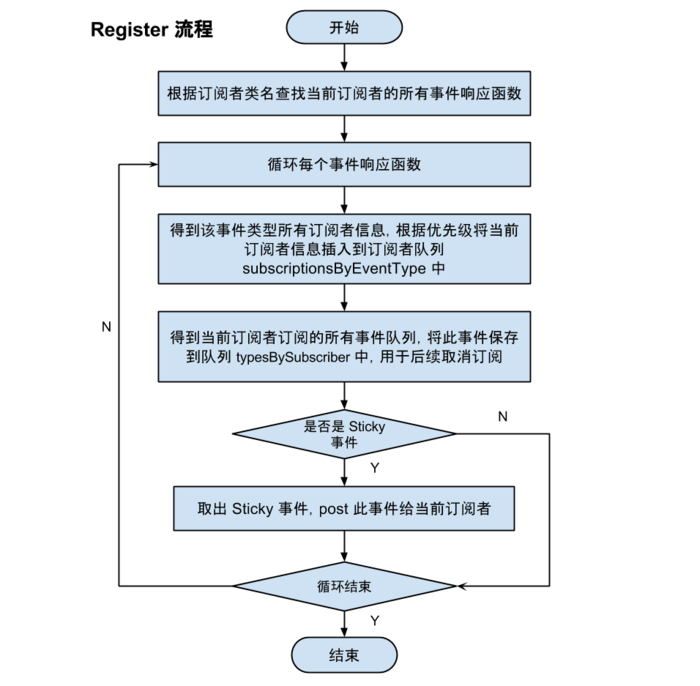
}

}

}

注册的时候通过subscriberMethodFinder的findSubscriberMethods方法去查找和缓存订阅者订阅了哪些事件.返回一个SubscriberMethod（SubscriberMethod里包含了这个方法的Method对象,以及将来响应订阅是在哪个线程的ThreadMode,以及订阅的事件类型eventType,以及订阅的优先级priority,以及是否接收粘性sticky事件的boolean值）对象的List。

接着回到subscribe(subscriber, subscriberMethod)中去,通过这个方法,我们就完成了注册。具体代码自行去观摩吧。



2、事件的发送Post

/\*\* Posts the given event to the event bus. \*/

public void post(Object event) {

//获取当前线程信息

//currentPostingThreadState是一个ThreadLocal

//ThreadLocal 是一个线程内部的数据存储类，通过它可以在指定的线程中存储数据，

PostingThreadState postingState = currentPostingThreadState.get();

List<Object> eventQueue = postingState.eventQueue;

//把事件加入线程队列中

eventQueue.add(event);

if (!postingState.isPosting) {

postingState.isMainThread = Looper.getMainLooper() == Looper.myLooper();

postingState.isPosting = true;

if (postingState.canceled) {

throw new EventBusException("Internal error. Abort state was not reset");

}

try {

while (!eventQueue.isEmpty()) {

//不断从队列中获取发送单个事件

postSingleEvent(eventQueue.remove(0), postingState);

}

} finally {

postingState.isPosting = false;

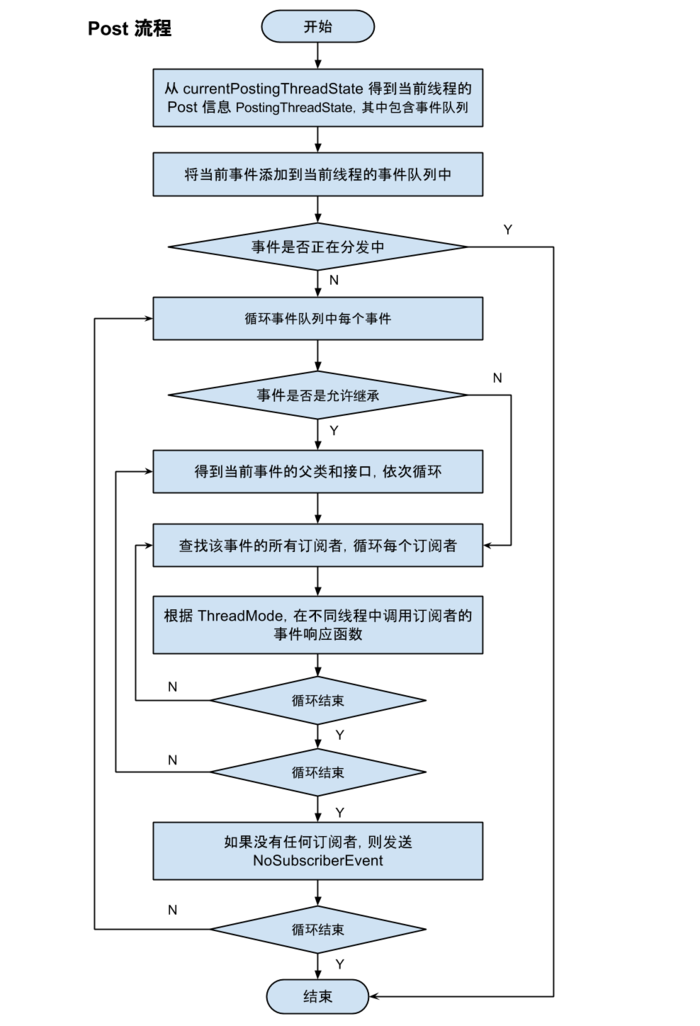
postingState.isMainThread = false;

}

}

}

大致原理是通过ThreadLocal存储指定线程的数据，数据在当前线程中的事件队列中不停往外分发。



关于EventBuse和广播以及RxBus

本地广播是相对消耗时间、空间最多的一种方式，但是大家都知道，广播是四大组件之一，许多系统级的事件都是通过广播来通知的，比如说网络的变化、电量的变化，短信发送和接收的状态，所以，如果与android系统进行相关的通知，还是要选择本地广播；在BroadcastReceiver的 onReceive方法中，可以获得Context 、intent参数，这两个参数可以调用许多的sdk中的方法，而eventbus获得这两个参数相对比较困难；

因此广播相对于其他的方式而言，广播是重量级的，消耗资源较多的方式。他的优势体现在与sdk连接紧密，如果需要同 android 交互的时候，广播的便捷性会抵消掉它过多的资源消耗，但是如果不同android交互，或者说，只做很少的交互，使用广播是一种浪费；

EventBus 作为 Android 开发中常用的框架，拥有着许多优点：

* 调度灵活。不依赖于 Context，使用时无需像广播一样关注 Context 的注入与传递。父类对于通知的监听和处理可以继承给子类，这对于简化代码至关重要；通知的优先级，能够保证 Subscriber 关注最重要的通知；粘滞事件（sticky events）能够保证通知不会因 Subscriber 的不在场而忽略。可继承、优先级、粘滞，是 EventBus 比之于广播、观察者等方式最大的优点，它们使得创建结构良好组织紧密的通知系统成为可能。
* 使用简单。EventBus 的 Subscriber 注册非常简单，调用 eventBus 对象的 register 方法即可，如果不想创建 eventBus 还可以直接调用静态方法 EventBus.getDefault() 获取默认实例，Subscriber 接收到通知之后的操作放在 onEvent 方法里就行了。成为 Publisher 的过程就更简单了，只需要调用合适的 eventBus（自己创建的或是默认的）的 post 方法即可。
* 快速且轻量。作为 github 的明星项目之一， EventBus 的源代码中许多技巧来改善性能，

eventbus的缺点是他的代码逻辑不是很清楚，在 Subscriber 注册的时候，Subscriber 中的方法会被遍历查找以 onEvent 开头的 public 方法。这将带来一些问题，一旦对代码进行混淆，就无法查找到了。好消息是 EventBus 已经打算使用注解来实现，这应该能够解决代码混淆的问题。

但有一个缺点是观察者独有的，那就是**观察者可能会造成接口的膨胀。**特别是当程序要求大量形式各异的通知，而程序员有没有做出良好的抽象时，代码中会包含大量的接口，接口数量的增长又会带来命名、注释等等一大堆问题。本质上说观察者要求程序员从零开始实现事件的产生、分发与处理过程，这就要求参与者必须对整个通知过程有着良好的理解。当程序代码适量时，这是一个合理的要求，然而当程序太大时，这将成为一种负担。

# 8.关于EventBus，广播，RxBus对比

本篇是对Android广播、观察者模式、EventBus、RxBus四者的一些看法

## 1、Android全局广播Broadcast

首先全局广播是重量级别的，并且会消耗很多资源，但是可以跨进程通信，通过以上优缺点可以知道只有一种情况下才会用到它，**跨进程**的时候，这个特点也是其他方案不能达到的。

## 2、Android本地广播Broadcast

Local Broadcast也是会消耗很多资源的，但是相比全局广播要轻量一些，他的最大优点是**可以拿到Context、Intent等和Android系统紧密相关的上下文**，这样就方便数据的传递和接受。

3、观察者模式

观察者观察一个主题的时候，就会把自己添加到主题的List里面，然后主题更新的时候会遍历主题List中的观察者，调用观察者模式中的update函数来更新自己，最后离开的时候要从主题中移除观察者，防止引用造成的内存泄漏。[网络上面观察者模式的例子](https://link.jianshu.com?t=http:/www.cnblogs.com/mengdd/archive/2013/02/07/2908929.html" \t "_blank)，由此可以看出**观察者模式中观察者是要实现主题更新时候的接口**，其实这就是和EventBus的不同，耦合性比EventBus大。

4、EventBus

EventBus中执行

EventBus.getDefault().register(this);

的时候是让EventBus扫描当前类，把onEvent方法记录在Map里面，参数为key，方法为value，然后EventBus.getDefault().post(...)的时候，通过post里面的type来查找Map中对应的value，然后通过反射来执行我们的方法，最后别忘了在最后

EventBus.getDefault().unregister(this);

注销EventBus防止内存泄漏。  
好多人说是观察者模式，但是仔细查看可以知道和观察者模式最大的不同是，**主题在要更新数据的时候是通过反射来执行动作的**  
个人感觉EventBus的缺点是：代码阅读性降低、不能跨进程、不能混淆，但是优点很多，比观察者模式耦合性更低，比广播更轻量。

5、RxBus

原理和EventBus一样，但是如果项目中用了RxJava，就可以选择这个强于EventBus。