拆轮子—EventBus

『EventBus对于Android开发老司机来说肯定不会陌生，它是一个基于观察者模式的事件发布/订阅框架，开发者可以通过极少的代码去实现多个模块之间的通信，而不需要以层层传递接口的形式去单独构建通信桥梁。从而降低因多重回调导致的模块间强耦合，同时避免产生大量内部类。它拥有使用方便，性能高，接入成本低和支持多线程的优点，实乃模块解耦、代码重构必备良药。』

摘自

<https://segmentfault.com/a/1190000005089229?utm_source=tuicool&utm_medium=referral>

其实这篇链接已经基本概括使用EventBus涉及到的问题。

这篇通过分析源码，可以了解整个EB的分发过程：

<http://skykai521.github.io/2016/02/20/EventBus-3-0%E6%BA%90%E7%A0%81%E5%88%86%E6%9E%90/>

以下是我在应用EB过程中的总结：

在3.0之前，注册监听需要区分是否监听黏性（sticky）事件，监听EventBus事件的模块需要实现以onEvent开头的方法。如今改为在方法上添加注解的形式，注解配置模式，事件优先级，是否粘性事件

@Subscribe(threadMode = ThreadMode.POSTING, priority = 0, sticky = true)

public void handleEvent(DriverEvent event) {

Log.d(TAG, event.info);

}

在postToSubscription()通过不同的threadMode在不同的线程里invoke()订阅者的方法,ThreadMode共有四类:

PostThread：默认的 ThreadMode，表示在执行 Post 操作的线程直接调用订阅者的事件响应方法，不论该线程是否为主线程（UI 线程）。当该线程为主线程时，响应方法中不能有耗时操作，否则有卡主线程的风险。适用场景：对于是否在主线程执行无要求，但若 Post 线程为主线程，不能耗时的操作；

MainThread：在主线程中执行响应方法。如果发布线程就是主线程，则直接调用订阅者的事件响应方法，否则通过主线程的 Handler 发送消息在主线程中处理——调用订阅者的事件响应函数。显然，MainThread类的方法也不能有耗时操作，以避免卡主线程。适用场景：必须在主线程执行的操作；

BackgroundThread：在后台线程中执行响应方法。如果发布线程不是主线程，则直接调用订阅者的事件响应函数，否则启动唯一的后台线程去处理。由于后台线程是唯一的，当事件超过一个的时候，它们会被放在队列中依次执行，因此该类响应方法虽然没有PostThread类和MainThread类方法对性能敏感，但最好不要有重度耗时的操作或太频繁的轻度耗时操作，以造成其他操作等待。适用场景：操作轻微耗时且不会过于频繁，即一般的耗时操作都可以放在这里；

Async：不论发布线程是否为主线程，都使用一个空闲线程来处理。和BackgroundThread不同的是，Async类的所有线程是相互独立的，因此不会出现卡线程的问题。适用场景：长耗时操作，例如网络访问。

由官方文档和实际demo实践可知，EB具体响应哪个事件，只与发送的事件（本质是个对象）的类型有关。也就是说，

订阅四种事件（注解里写明模式，此处以方法名区分）

onPostingEvent、onMainEvent、onBackgroundEvent、onAsyncEvent

订阅者在接收事件时，只以参数（事件类型）为准。

现订阅以下三种方法（默认posting，优先级0，不是粘性事件）

@Subscribe(threadMode = ThreadMode.Main, priority = 0, sticky = true)

public void onMainEvent (AEvent a) {

Log.d(TAG, “main ”+a.info);

}

@Subscribe(threadMode = ThreadMode.Main, priority = 0, sticky = true)

public void onMainEvent (BEvent b) {

Log.d(TAG, “main ”+b.info);

}

@Subscribe(threadMode = ThreadMode.Async, priority = 0, sticky = true)

public void onAsyncEvent (AEvent a) {

Log.d(TAG, “async ”+a.info);

}

当某发送AEvent事件时，实际上会打印两条消息

“main ”+a.info 和 “async ”+a.info

而不是根据某发送事件时-某所处的进程来判断。

不会出现，某在main发送AEvent，接收“main ”+a.info，

而在async发送AEvent，响应“async ”+a.info 的情况。

框架设计思路：

在设计框架，BaseActivity/BaseFragment的时候，应当把订阅的事件（四种）明晰，定义相对应的四种事件，MainEvent，PostEvent，BackgroundEvent，AsyncEvent

，这样的设计，减少多次使用handler调度线程，因为EventBus已经处理过一次线程了。

如果MainEvent有多个行为，设计Mainevent事件类的时候，添加一个mainEventType参数，发送时就指定某事件交给哪个线程处理，哪个线程处理哪些不同的动作，

onMainEvent（MainEvent main）{

switch(main.getType){

case type1:

…..

default:

break;

}

没有必要onMainEvent（A），onMainEvent（B），onMainEvent（C）

这样要写三个类，又都是Main进程处理，代码冗余。

拆轮子—注解Annotation

注解demo解析，这篇很直接的讲了怎样去看，去定义一个注解，暴力直白。

<http://www.cnblogs.com/yydcdut/p/4646454.html>

这个循序渐进，由浅入深的从含义到调用，虽稍显啰嗦，但是很值得慢慢看。

<http://blog.csdn.net/lmj623565791/article/details/43452969>

我用我的中二理解来描述注解。

程序员---魔法师；

代码里的method、function、方法---武器；

元Annotation---元素；

@Retention, @Target, @Inherited, @Documented

标准Annotation---基础附魔；

Override, Deprecated, SuppressWarnings

自定义Annotation---自定义附魔；

众所周知，在打造武器的时候，附魔总是能够给武器加上不可思议的效果。

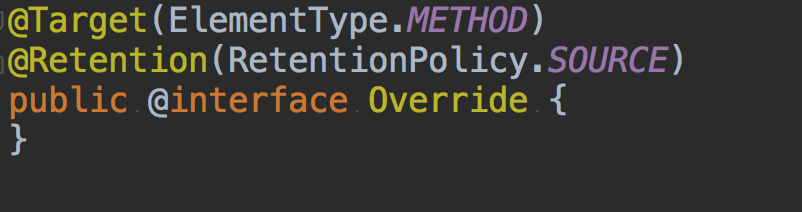
而附魔受附魔元素，附魔环境的影响，

通过组合不同的元素（甚至不同形态的元素，如水和冰）创造不同的附魔效果，定义附魔环境，

最后喊出法术名称，给武器附魔。

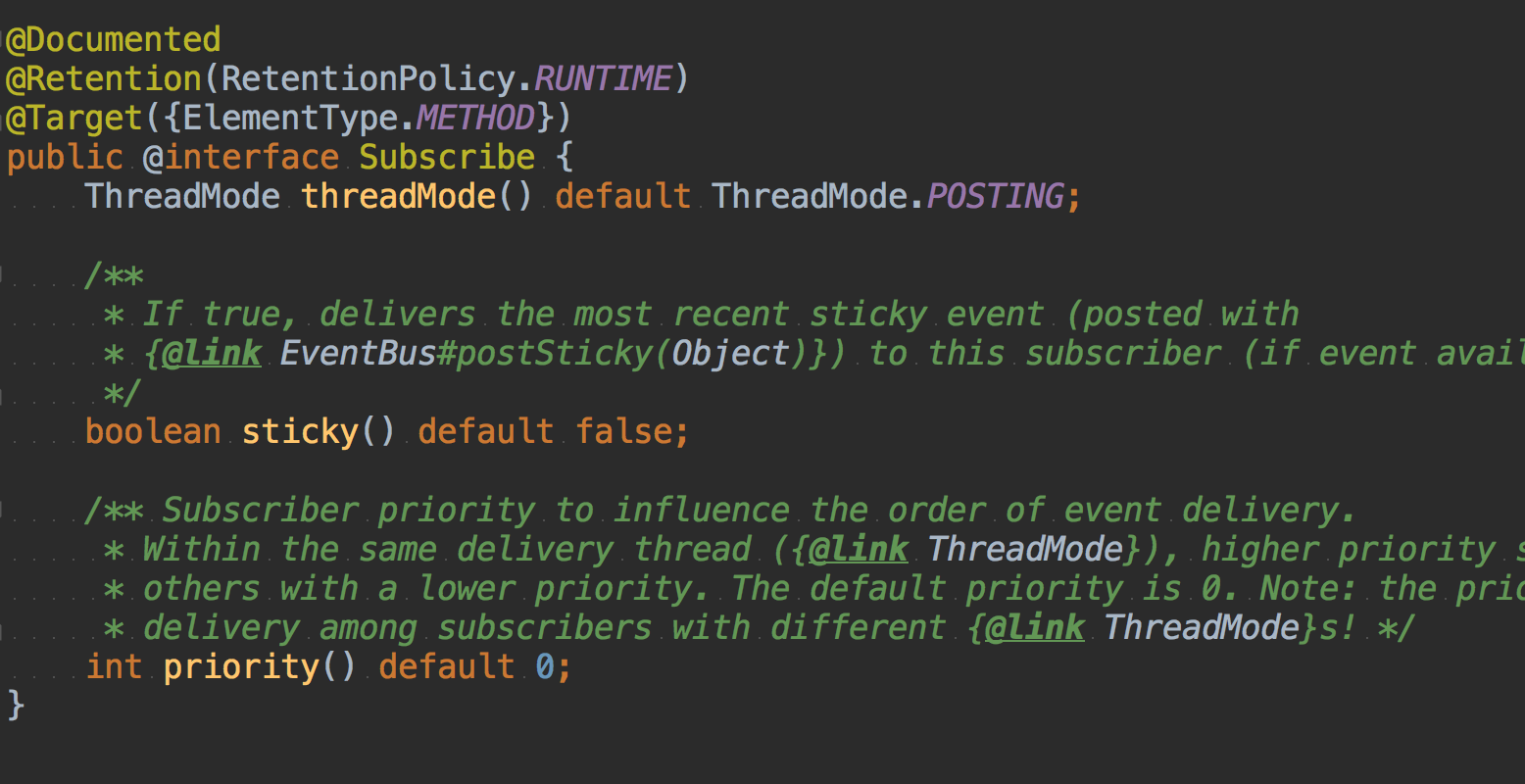
-----------------------------------0：附魔解析-------------------------------

比如标准注解（标准附魔）Override的定义是：



代表的含义，这是一个不被继承的，用于描述方法的，源文件保留，不写进Doc的注解。

EventBus的Subscribe也是注解，定义如下



代表的意思这是一个写入doc，运行时保留，描述方法，不可继承的注解。

----------------------------1.元素之力------------------------

元素@Documented 是否会保存到 Javadoc 文档中，默认否（就是不写）。

元素@Retention 保留时间，默认为 CLASS。

作用：表示需要在什么级别保存该注释信息，用于描述注解的生命周期（即：被描述的注解在什么范围内有效）

取值（RetentionPolicy）有：

　　　　1.SOURCE:在源文件中有效（即源文件保留），SOURCE 大都为 Mark Annotation，这类 Annotation 大都用来校验，比如 Override, Suppress Warnings；

　　　　2.CLASS:在class文件中有效（即class保留）；

　　　　3.RUNTIME:在运行时有效（即运行时保留）。

@Target 可以用来修饰哪些程序元素，默认未标注，表示可修饰所有。

作用：用于描述注解的使用范围（即：被描述的注解可以用在什么地方）

取值(ElementType)有：

　　　　1.CONSTRUCTOR:用于描述构造器；

　　　　2.FIELD:用于描述域；

　　　　3.LOCAL\_VARIABLE:用于描述局部变量；

　　　　4.METHOD:用于描述方法；

　　　　5.PACKAGE:用于描述包；

　　　　6.PARAMETER:用于描述参数；

　　　　7.TYPE:用于描述类、接口(包括注解类型) 或enum声明。

@Inherited 是否可以被继承，默认为 false。

当一个 @Inherited类型标注的注解的Retention取值是RetentionPolicy.RUNTIME，则反射API增强了这种继承性。

如果我们使用java.lang.reflect去查询一个@Inherited类型的注解，反射代码检查将展开如下工作：检查class和其父类，直到发现指定的注解类型被发现，或者到达类继承结构的顶层。

--------------------2.法术名------------

@interface修饰的就是注解的名称。

----------------------3.附魔环境-------------------------------

public @interface 注解名 {定义体}

在定义体里写方法名，实际上定义体内的方法名是注解的参数名。

定义体的返回值支持

1.所有基本数据类型（int,float,boolean,byte,double,char,long,short）；

2.String类型；

3.Class类型；

4.enum类型；

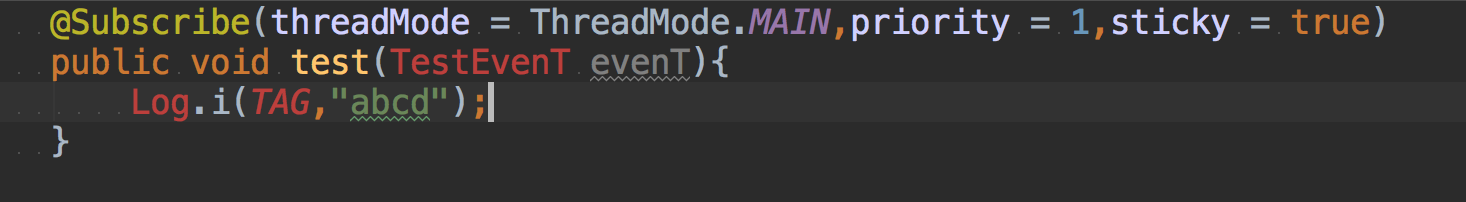
5.Annotation类型；

6.以上所有类型的一维数组。

定义public@燃烧效果

{ int 火元素 default 500°；助燃体Class default 氢气；enum 氧气}

---------------------------4.释放技能------------------------------



在满足runtime，method时，

对handleEvent方法附上了一个优先级为0，粘性的subscribe魔法效果。

---------------------------------5.技能效果--------------------------

为什么要给方法加上注解呢？---为什么要给武器附魔呢？

答案很简单嘛，调用更快---武器更强。

拆轮子—单例模式

拆轮子—工厂模式

拆轮子—OkHttp