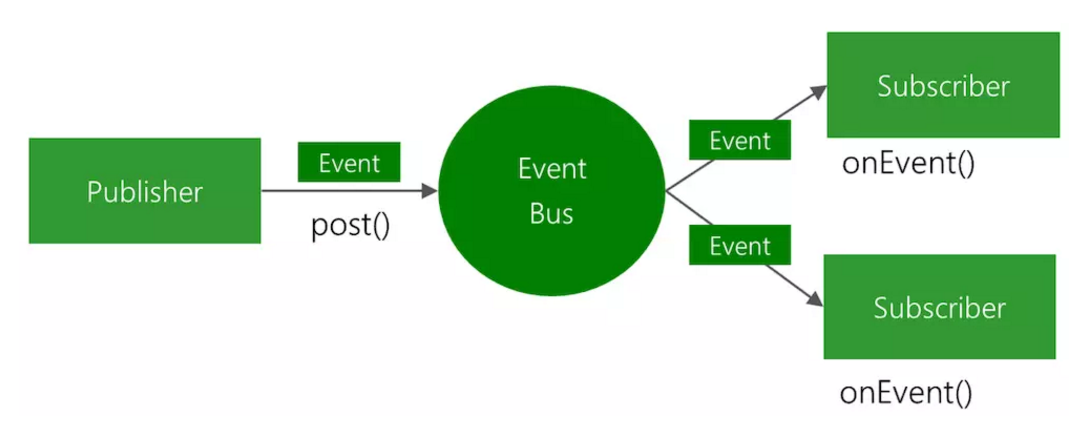
EventBus是一种用于Android的发布/订阅事件总线。它有很多优点：简化应用组件间的通信；解耦事件的发送者和接收者；避免复杂和容易出错的依赖和生命周期的问题；很快，专门为高性能优化过等等。

# 1、基础知识

EventBus使用了发布者/订阅者模式。



发布者publisher通过EventBus发布事件event，订阅者subscriber通过EventBus订阅事件。当发布者发布事件时，订阅该事件的订阅者的事件处理方法onEvent()将被调用。

# 2、准备工作

首先，在模块的 build.gradle 构建脚本中添加EventBus依赖：

dependencies {

...

compile 'org.greenrobot:eventbus:3.1.1'

}

接着，添加EventBus混淆规则。ProGuard工具混淆了方法名，并可能移除那些未被调用的方法。订阅者的事件处理方法没有被直接调用，如果开启了ProGuard工具的压缩功能，那么你必须告诉ProGuard工具保留这些订阅者方法。在模块的 proguard-rules.pro 混淆规则文件中添加如下规则：

# EventBus

-keepattributes \*Annotation\*

-keepclassmembers class \*\* {

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

}

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

# 3、基本使用

EventBus的使用非常简单，主要分为3个步骤：

定义事件。

订阅事件。

发布事件。

|  |
| --- |
| Caused by: org.greenrobot.eventbus.EventBusException: Subscriber class com.netease.  eventbus.demo.MainActivity already registered to event class java.lang.String    EventBus.getDefault().register(this);  EventBus.getDefault().register(this);  //EventBus.getDefault().isRegistered(this); //判断是否已经注册  //如果eventbus是重复注册会出错。  public synchronized boolean isRegistered(Object subscriber) {  return typesBySubscriber.containsKey(subscriber);  }  //有个map存放被注册的对象，主要判断是否存在就行了，是个mapHashMap。  private final Map<Object, List<Class<?>>> typesBySubscriber; |

## 3.1 第一步，定义事件。

事件可以是任意普通的Java对象，没有任何特殊的要求。例如：

public class MessageEvent {

private String message;

public MessageEvent(String message) {

this.message = message;

}

public String getMessage() {

return message;

}

public void setMessage(String message) {

this.message = message;

}

}

## 3.2 第二步，订阅事件。

订阅者需要定义事件处理方法（也称为订阅者方法）。当发布对应类型的事件时，该方法将被调用。EventBus 3使用 @Subscribe 注解来定义订阅者方法。方法名可以是任意合法的方法名，参数类型为订阅事件的类型。例如：

@Subscribe(threadMode = ThreadMode.MAIN)

public void onMessageEvent(MessageEvent event) {

...

}

订阅者还需要在总线上注册，并在不需要时在总线上注销。只有订阅者注册了，它们才会收到事件。在Android中，可以根据Activity或者Fragment的生命周期来注册和注销。例如：

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_main);

initContentView();

// 注册订阅者

EventBus.getDefault().register(this);

}

@Override

protected void onDestroy() {

super.onDestroy();

// 注销订阅者

EventBus.getDefault().unregister(this);

}

需要注意的是：一般在onCreate()方法中进行注册订阅。在onDestory()方法中进行解除订阅。

## 3.3 第三步，发布事件。

在需要的地方发布事件，所有订阅了该类型事件并已注册的订阅者将收到该事件。例如：

// 发布事件

EventBus.getDefault().post(new MessageEvent("Hello EventBus!"));

## 3.4一个例子

订阅事件的代码如下所示：

public class MainActivity extends AppCompatActivity implements View.OnClickListener {

private static final String TAG = "MainActivity";

private TextView mTvMessage;

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_main);

initContentView();

// 注册订阅者

EventBus.getDefault().register(this);

}

private void initContentView() {

Button btnStart = findViewById(R.id.btn\_main\_start\_activity);

mTvMessage = findViewById(R.id.tv\_main\_message);

btnStart.setOnClickListener(this);

}

@Override

public void onClick(View v) {

if (v.getId() == R.id.btn\_main\_start\_activity) {

SecondActivity.start(this);

}

}

@Subscribe(threadMode = ThreadMode.MAIN)

public void onMessageEvent(MessageEvent event) {

Log.i(TAG, "message is " + event.getMessage());

// 更新界面

mTvMessage.setText(event.getMessage());

}

@Override

protected void onDestroy() {

super.onDestroy();

// 注销订阅者

EventBus.getDefault().unregister(this);

}

}

MainActivity订阅了MessageEvent事件。当接收到MessageEvent事件时，订阅者方法将打印日志消息，并更新界面上的TextView。

发布事件的代码如下所示：

public class SecondActivity extends AppCompatActivity implements View.OnClickListener {

public static void start(Context context) {

Intent intent = new Intent(context, SecondActivity.class);

context.startActivity(intent);

}

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_second);

initContentView();

}

private void initContentView() {

findViewById(R.id.btn\_second\_post\_event).setOnClickListener(this);

}

@Override

public void onClick(View v) {

if (v.getId() == R.id.btn\_second\_post\_event) {

// 发布事件

EventBus.getDefault().post(new MessageEvent("Hello EventBus!"));

}

}

}

当点击发布事件的按钮时，SecondActivity将发布一个MessageEvent事件。

运行应用。点击MainActivity界面上的启动活动按钮来启动SecondActivity，然后点击SecondActivity界面上的发布事件按钮来发布事件。最后，回退到MainActivity，可以看到界面上的TextView的内容已经更新为"Hello EventBus!"，并且应用打印出如下信息：

12-27 20:59:18.919 24705-24705/com.github.cyc.eventbus.basicusedemo I/MainActivity: message is Hello EventBus!

# 4、线程模式

EventBus支持订阅者方法在不同于发布事件所在线程的线程中被调用。你可以使用线程模式来指定调用订阅者方法的线程。EventBus总共支持5种线程模式：

ThreadMode.POSTING 订阅者方法将在发布事件所在的线程中被调用。这是 默认的线程模式。事件的传递是同步的，一旦发布事件，所有该模式的订阅者方法都将被调用。这种线程模式意味着最少的性能开销，因为它避免了线程的切换。因此，对于不要求是主线程并且耗时很短的简单任务推荐使用该模式。使用该模式的订阅者方法应该快速返回，以避免阻塞发布事件的线程，这可能是主线程。

ThreadMode.MAIN 订阅者方法将在主线程（UI线程）中被调用。因此，可以在该模式的订阅者方法中直接更新UI界面。如果发布事件的线程是主线程，那么该模式的订阅者方法将被直接调用。使用该模式的订阅者方法必须快速返回，以避免阻塞主线程。

ThreadMode.MAIN\_ORDERED 订阅者方法将在主线程（UI线程）中被调用。因此，可以在该模式的订阅者方法中直接更新UI界面。事件将先进入队列然后才发送给订阅者，所以发布事件的调用将立即返回。这使得事件的处理保持严格的串行顺序。使用该模式的订阅者方法必须快速返回，以避免阻塞主线程。

（

ThreadMode: MAIN：订阅者(Subscriber) 在主线程中执行响应事件的方法,如果 发布者(Publisher) post 事件是主线程,则直接调用响应事件的方法,如果 post 的是子线程,则加入到主线程的消息循环队列中执行响应事件的方法。

ThreadMode: MAIN\_ORDERED： 订阅者(Subscriber) 在主线程中执行响应事件的方法 和 ThreadMode: MAIN 区别在于,不管 发布者(Publisher) post 事件是什么线程 ,MAIN\_ORDERED会把事件加入到主线程的消息循环队列中执行,而不会直接调用处理消息的方法。

）

ThreadMode.BACKGROUND 订阅者方法将在后台线程中被调用。如果发布事件的线程不是主线程，那么订阅者方法将直接在该线程中被调用。如果发布事件的线程是主线程，那么将使用一个单独的后台线程，该线程将按顺序发送所有的事件。使用该模式的订阅者方法应该快速返回，以避免阻塞后台线程。

ThreadMode.ASYNC 订阅者方法将在一个单独的线程中被调用。因此，发布事件的调用将立即返回。如果订阅者方法的执行需要一些时间，例如网络访问，那么就应该使用该模式。避免触发大量的长时间运行的订阅者方法，以限制并发线程的数量。EventBus使用了一个线程池来有效地重用已经完成调用订阅者方法的线程。

# 5、粘性事件

如果先发布了事件，然后有订阅者订阅了该事件，那么除非再次发布该事件，否则订阅者将永远接收不到该事件。此时，可以使用粘性事件。发布一个粘性事件之后，EventBus将在内存中缓存该粘性事件。当有订阅者订阅了该粘性事件，订阅者将接收到该事件。

订阅和发布一个粘性事件的示例代码如下所示：

// 订阅粘性事件

@Subscribe(sticky = true)

public void onMessageEvent(MessageEvent event) {

...

}

// 发布粘性事件

EventBus.getDefault().postSticky(new MessageEvent("Hello EventBus!"));

发布一个粘性事件之后，EventBus将一直缓存该粘性事件。如果想要移除粘性事件，那么可以使用如下方法：

// 移除指定的粘性事件

removeStickyEvent(Object event);

// 移除指定类型的粘性事件

removeStickyEvent(Class<T> eventType);

// 移除所有的粘性事件

removeAllStickyEvents();

# 6、事件优先级

EventBus支持在定义订阅者方法时指定事件传递的优先级。默认情况下，订阅者方法的事件传递优先级为0。数值越大，优先级越高。在相同的线程模式下，更高优先级的订阅者方法将优先接收到事件。注意：优先级只有在相同的线程模式下才有效。

指定事件传递优先级的示例代码如下所示：

@Subscribe(priority = 1)

public void onMessageEvent(MessageEvent event) {

...

}

你可以在高优先级的订阅者方法接收到事件之后取消事件的传递。此时，低优先级的订阅者方法将不会接收到该事件。注意： 订阅者方法只有在线程模式为ThreadMode.POSTING时，才可以取消一个事件的传递。

取消事件传递的示例代码如下所示：

@Subscribe(threadMode = ThreadMode.POSTING, priority = 1)

public void onMessageEvent(MessageEvent event) {

...

// 取消事件传递

EventBus.getDefault().cancelEventDelivery(event);

}

# 7、订阅者索引类

默认情况下，EventBus在查找订阅者方法时采用的是反射。订阅者索引是EventBus 3的一个新特性。它可以加速订阅者的注册，是一个可选的优化。订阅者索引的原理是：使用EventBus的注解处理器在应用构建期间创建订阅者索引类，该类包含了订阅者和订阅者方法的相关信息。EventBus官方推荐在Android中使用订阅者索引以获得最佳的性能。

要开启订阅者索引的生成，你需要在构建脚本中使用annotationProcessor属性将EventBus的注解处理器添加到应用的构建中，还要设置一个eventBusIndex参数来指定要生成的订阅者索引的完全限定类名。

我们在前面的基本使用的那个例子上进行修改。首先，修改模块下的build.gradle构建脚本：

android {

defaultConfig {

...

javaCompileOptions {

annotationProcessorOptions {

arguments = [eventBusIndex:

'com.github.cyc.eventbus.subscriberindexdemo.MyEventBusIndex']

}

}

}

...

}

dependencies {

...

compile 'org.greenrobot:eventbus:3.1.1'

annotationProcessor 'org.greenrobot:eventbus-annotation-processor:3.1.1'

}

然后，build一下工程。EventBus注解处理器将为你生成一个订阅者索引类。

最后，在应用自定义的Application类的onCreate()方法中将订阅者索引类添加到EventBus中，并将该EventBus设置成默认的EventBus。示例代码如下所示：

public class MyApplication extends Application {

@Override

public void onCreate() {

super.onCreate();

// 配置EventBus

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

}

}

# 8、总结

EventBus是一种用于Android的发布/订阅事件总线。使用EventBus可以简化应用组件间的通信，可以解耦事件的发送者和接收者。

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。

1. 单例类：eventbus是基于注册监听的方式来运行的，因此，首先需要将eventbus，然后才会有事件及监听者。新建eventbus或者AsyncEventBus。
2. 订阅：注册监听者

底层就是将类eventListener中所有注解有Subscribe的方法与其Event对放在一个map中（一个event可以对应多个Subscribe的方法）。

（3）事件发送：执行指定事件类型的订阅者（包含了method），从订阅者中获取指定事件的订阅者，然后按照规则（同步、异步）执行指定的方法。

## 8.1 Subscribe注解

EventBus3.0 开始用Subscribe注解配置事件订阅方法，不再使用方法名了，例如：

@Documented

@Retention(RetentionPolicy.RUNTIME)

@Target({ElementType.METHOD})

public @interface Subscribe {

// 指定事件订阅方法的线程模式，即在那个线程执行事件订阅方法处理事件，默认为POSTING

ThreadMode threadMode() default ThreadMode.POSTING;

// 是否支持粘性事件，默认为false

boolean sticky() default false;

// 指定事件订阅方法的优先级，默认为0，如果多个事件订阅方法可以接收相同事件的，则优先级高的先接收到事件

int priority() default 0;

}

所以在使用Subscribe注解时可以根据需求指定threadMode、sticky、priority三个属性。

其中threadMode属性有如下几个可选值：

• ThreadMode.POSTING，默认的线程模式，在那个线程发送事件就在对应线程处理事件，避免了线程切换，效率高。

• ThreadMode.MAIN，如在主线程（UI线程）发送事件，则直接在主线程处理事件；如果在子线程发送事件，则先将事件入队列，然后通过 Handler 切换到主线程，依次处理事件。

• ThreadMode.MAIN\_ORDERED，无论在那个线程发送事件，都先将事件入队列，然后通过 Handler 切换到主线程，依次处理事件。

• ThreadMode.BACKGROUND，如果在主线程发送事件，则先将事件入队列，然后通过线程池依次处理事件；如果在子线程发送事件，则直接在发送事件的线程处理事件。

• ThreadMode.ASYNC，无论在那个线程发送事件，都将事件入队列，然后通过线程池处理。

## 8.2 注册事件订阅方法

注册：查找+注册

反注册：查找+删除

注册事件的方式如下：

EventBus.getDefault().register(this);

其中getDefault()是一个单例方法，保证当前只有一个EventBus实例：

public static EventBus getDefault() {

if (defaultInstance == null) {

synchronized (EventBus.class) {

if (defaultInstance == null) {

defaultInstance = new EventBus();

}

}

}

return defaultInstance;

}

有了EventBus的实例就可以进行注册了：

public void register(Object subscriber) {

// 得到当前要注册类的Class对象

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

// 根据Class查找当前类中订阅了事件的方法集合，即使用了Subscribe注解、有public修饰符、一个参数的方法

// SubscriberMethod类主要封装了符合条件方法的相关信息：

// Method对象、线程模式、事件类型、优先级、是否是粘性事等

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

synchronized (this) {

// 循环遍历订阅了事件的方法集合，以完成注册

for (SubscriberMethod subscriberMethod : subscriberMethods) {

subscribe(subscriber, subscriberMethod);

}

}

}

先查找：找到符合条件的方法；

再注册：遍历方法

// 循环遍历当前类的方法，筛选出符合条件的

// 获得方法的修饰符

// 如果是public类型，但非abstract、static等

// 获得当前方法所有参数的类型

// 如果当前方法只有一个参数

// 如果当前方法使用了Subscribe注解

// 得到该参数的类型

// 得到Subscribe注解的threadMode属性值，即线程模式，以及是否是粘性事件

// 创建一个SubscriberMethod对象，并添加到subscriberMethods集合

我们已经找到了当前注册类及其父类中订阅事件的方法的集合。

把之前找到的符合的方法及方法信息、方法所在的对象保存到集合。

## 8.3 事件的处理

post发布事件：发布+查找同类型的事件的方法+调用方法（线程切换+反射）

private void postToSubscription(Subscription subscription, Object event, boolean isMainThread) {

// 判断订阅事件方法的线程模式

switch (subscription.subscriberMethod.threadMode) {

// 默认的线程模式，在那个线程发送事件就在那个线程处理事件

case POSTING:

invokeSubscriber(subscription, event);

break;

// 在主线程处理事件

case MAIN:

// 如果在主线程发送事件，则直接在主线程通过反射处理事件

if (isMainThread) {

invokeSubscriber(subscription, event);

} else {

// 如果是在子线程发送事件，则将事件入队列，通过Handler切换到主线程执行处理事件

// mainThreadPoster 不为空

mainThreadPoster.enqueue(subscription, event);

}

break;

// 无论在那个线程发送事件，都先将事件入队列，然后通过 Handler 切换到主线程，依次处理事件。

// mainThreadPoster 不为空

case MAIN\_ORDERED:

if (mainThreadPoster != null) {

mainThreadPoster.enqueue(subscription, event);

} else {

invokeSubscriber(subscription, event);

}

break;

case BACKGROUND:

// 如果在主线程发送事件，则先将事件入队列，然后通过线程池依次处理事件

if (isMainThread) {

backgroundPoster.enqueue(subscription, event);

} else {

// 如果在子线程发送事件，则直接在发送事件的线程通过反射处理事件

invokeSubscriber(subscription, event);

}

break;

// 无论在那个线程发送事件，都将事件入队列，然后通过线程池处理。

case ASYNC:

asyncPoster.enqueue(subscription, event);

break;

default:

throw new IllegalStateException("Unknown thread mode: " + subscription.subscriberMethod.threadMode);

}

}