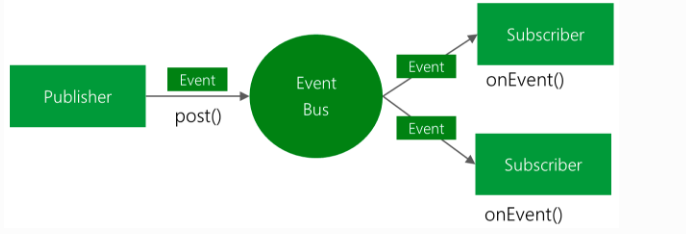
# EventBus: Events for Android

## ****EventBus是使用发布者/订阅者模式进行松散耦合的Android开源库。EventBus使得中央通信能够仅使用几行代码来解耦类 - 简化代码，消除依赖关系，加快应用程序开发速度****



## ****EventBus的好处 简化组件之间的通信、分离事件发送者和接收者、对活动，片段和后台线程表现良好、避免复杂和容易出错的依赖关系和生命周期问题、很快专门针对高性能进行优化、很小（50k jar）、具有传送线程，用户优先级等高级功能****

## ****EventBus的功能****

* **基于方便的基于注释的API：基于方便的基于注释的API：**简单地将@Subscribe注释添加到您的用户方法中。由于注释的构建时间索引，EventBus不需要在应用程序运行时进行注释反射。
* **Android主线程交付：**当与UI进行交互时，EventBus可以在主线程中传递事件，而不管事件如何发布。
* **后台线程传递：**如果您的用户长时间运行任务，EventBus还可以使用后台线程来避免UI阻塞。
* **事件和用户继承：**在EventBus中，面向对象的范例适用于事件和订阅者类。假设事件类A是B的超类。类型B的发布的事件也将被发布给对A感兴趣的用户。类似地，考虑用户类的继承。
* **跳转开始：**您可以立即开始使用默认的EventBus实例，无需配置任何内容。
* **可配置：**  要根据需要调整EventBus，可以使用构建器模式调整其行为。
* **简单而强大：**  EventBus是一个非常容易学习的API的小型库。尽管如此，您的软件架构可能会通过解耦组件获得巨大的收益：使用事件时，订阅者对发件人不了解。
* **战斗测试：** EventBus是最常用的Android库之一：数千个应用程序使用EventBus，包括非常受欢迎的应用程序。超过10亿的应用安装自己说话。
* **高性能：**特别是在Android上，性能很重要。EventBus被分析和优化了很多; 可能使其成为同类最快的解决方案。
* **基于方便的基于注释的API**  （不牺牲性能）**：**简单地将@Subscribe注释添加到您的用户方法中。由于注释的构建时间索引，EventBus不需要在应用程序运行时进行注释反射，这在Android上非常慢。
* **Android主线程交付：**当与UI进行交互时，EventBus可以在主线程中传递事件，而不管事件如何发布。
* **后台线程传递：**如果您的用户长时间运行任务，EventBus还可以使用后台线程来避免UI阻塞。
* **事件和用户继承：**在EventBus中，面向对象的范例适用于事件和订阅者类。假设事件类A是B的超类。类型B的发布的事件也将被发布给对A感兴趣的用户。类似地，考虑用户类的继承。
* **零配置：**  您可以立即开始使用代码中任何地方提供的默认EventBus实例。
* **可配置：**  要根据需要调整EventBus，可以使用构建器模式调整其行为。

# EventBus 3.0与2.x的区别

## ****包名不同****

## ****订阅方法****

* EventBus 2.x：  
  使用固定的方法名标识订阅方法，方法名也代表了threadMode

onEvent() —— POST线程，订阅方法运行在发布者所在的线程  
onEventMainThread() —— UI主线程，订阅方法运行在主线程  
onEventBackground() —— 后台线程，发布者是主线程，订阅方法运行在新开子线程；发布者是子线程，订阅方法运行在发布者所在的线程；  
onEventAsync() —— 异步线程，订阅方法运行在新开子线程，无论发布者是在哪个线程

* EventBus 3.0：  
  使用注解来标识订阅方法。同时可以指定threadMode，代表订阅方法所运行的线程；指定sticky，代表是否是粘性事件；指定priority，代表优先级。

public @interface Subscribe {

ThreadMode threadMode() default ThreadMode.POSTING;

/\*\*

\* If true, delivers the most recent sticky event (posted with

\* {@link EventBus#postSticky(Object)}) to this subscriber (if event available).

\*/

boolean sticky() default false;

/\*\* Subscriber priority to influence the order of event delivery.

\* Within the same delivery thread ({@link ThreadMode}), higher priority subscribers will receive events before

\* others with a lower priority. The default priority is 0. Note: the priority does \*NOT\* affect the order of

\* delivery among subscribers with different {@link ThreadMode}s! \*/

int priority() default 0;

}

ThreadMode.POSTING —— POST线程，订阅方法运行在发布者所在的线程（默认）  
ThreadMode.MAIN —— UI主线程，订阅方法运行在主线程  
ThreadMode.BACKGROUND —— 后台线程，发布者是主线程，订阅方法运行在新开子线程；发布者是子线程，订阅方法运行在发布者所在的线程；  
ThreadMode.ASYNC —— 异步线程，订阅方法运行在新开子线程，无论发布者是在哪个线程

sticky，与Android广播机制中的sticky概念相同。即在订阅者注册之前，便已经把事件发送出去，等到注册之后，订阅者便会收到最近发送的粘性事件。注意是**最近**，即重复发送同一种类型的粘性事件，订阅者只能收到最后一条。

priority，当事件发送后，指定多个订阅者收到事件的顺序

@Subscribe(threadMode = ThreadMode.MAIN, sticky = true， priority = 10)

public void onEvent(EventBusEvent eventBusEvent) {

tv.setText(eventBusEvent.info);

}

## ****参数配置的颗粒度****

* EventBus 2.x：  
  priority 和 stickyness 仅仅以类为单位进行配置，迫使该类中所有订阅方法不得不共享同一配置
* EventBus 3.0 ：  
  priority 和 stickyness 以方法为单位进行配置，颗粒度更细，显然更加灵活

## ****性能提升****

* EventBus 2.x：  
  运行时，使用反射的方式来查找回调方法，性能比较慢
* EventBus 3.0：  
  提供了EventBusAnnotationProcessor注解处理器，可以在编译期读取@Subscribe()注解并且解析其中所包含的信息，以供运行时直接使用，从而运行时的性能大大提高了

# EventBus 的使用

## ****3.1定义events，如果你需要更多的扩展性，可以考虑加入泛型:****

public class EventBusEvent {

public final String info;

public EventBusEvent (String Info) {

this.info = Info;

}

}

## ****3.2准备 subscribers**** 订阅者实现了事件处理方法，在事件被发送时，它将被调用。这相当于经典观察者模式中的update()方法。在EventBus3.0中，在你的订阅方法上加上注解, 可以指定一个thread mode。并且，方法必须是public权限，其方法参数有且只能有一个，另外类型必须为第一步定义好的事件类型: 3.0版本 @Subscribe(threadMode = ThreadMode.MAIN)

public void onEvent(EventBusEvent eventBusEvent) {

tv.setText(eventBusEvent.info);

}

//2.x版本

public void onEvent(EventBusEvent eventBusEvent) {}

public void onEventMainThread(EventBusEvent eventBusEvent) {}

public void onEventBackgroundThread(EventBusEvent eventBusEvent) {}

同时，register 和 unregister 你的 subscriber。仅仅当订阅者被注册后，它们才能接收到事件。对于Android , activity 和 fragment 一般情况下应该依据生命周期来register，大部分情况下，可以放在onStart/onStop中:

@Override

public void onStart() {

super.onStart();

EventBus.getDefault().register(this);

}

@Override

public void onStop() {

super.onStop();

EventBus.getDefault().unregister(this);

}

## **3.3 这相当于经典观察者模式中的notifyObservers()方法。你可以在你**的代码中的任何地方发送事件，所有当前被注册的并且匹配该事件类型的**订阅者，都将接收到它。**

EventBus.getDefault().post(new EventBusEvent("Wild Android Monster"));

## ****3.4线程间的传递（ThreadMode）**** 在EventBus的事件处理函数中需要指定线程模型，即指定事件处理函数运行所在的想线程。在上面我们已经接触到了EventBus的四种线程模型。在EventBus中的观察者通常有四种线程模型，分别是PostThread（默认）、MainThread、BackgroundThread与Async

* PostThread：如果使用事件处理函数指定了线程模型为PostThread，那么该事件在哪个线程发布出来的，事件处理函数就会在这个线程中运行，也就是说发布事件和接收事件在同一个线程。在线程模型为PostThread的事件处理函数中尽量避免执行耗时操作，因为它会阻塞事件的传递，甚至有可能会引起ANR。
* MainThread：如果使用事件处理函数指定了线程模型为MainThread，那么不论事件是在哪个线程中发布出来的，该事件处理函数都会在UI线程中执行。该方法可以用来更新UI，但是不能处理耗时操作。
* BackgroundThread：如果使用事件处理函数指定了线程模型为BackgroundThread，那么如果事件是在UI线程中发布出来的，那么该事件处理函数就会在新的线程中运行，如果事件本来就是子线程中发布出来的，那么该事件处理函数直接在发布事件的线程中执行。在此事件处理函数中禁止进行UI更新操作。
* Async：如果使用事件处理函数指定了线程模型为Async，那么无论事件在哪个线程发布，该事件处理函数都会在新建的子线程中执行。同样，此事件处理函数中禁止进行UI更新操作。
* @Subscribe(threadMode = ThreadMode.PostThread)
* public void onMessageEventPostThread(MessageEvent messageEvent) {
* Log.e("PostThread", Thread.currentThread().getName());
* }
* @Subscribe(threadMode = ThreadMode.MainThread)
* public void onMessageEventMainThread(MessageEvent messageEvent) {
* Log.e("MainThread", Thread.currentThread().getName());
* }
* @Subscribe(threadMode = ThreadMode.BackgroundThread)
* public void onMessageEventBackgroundThread(MessageEvent messageEvent) {
* Log.e("BackgroundThread", Thread.currentThread().getName());
* }
* @Subscribe(threadMode = ThreadMode.Async)
* public void onMessageEventAsync(MessageEvent messageEvent) {
* Log.e("Async", Thread.currentThread().getName());
* }

分别使用上面四个方法订阅同一事件，打印他们运行所在的线程。首先我们在UI线程中发布一条MessageEvent的消息，看下日志打印结果是什么。

findViewById(R.id.send).setOnClickListener(new View.OnClickListener() {

@Override

public void onClick(View v) {

Log.e("postEvent", Thread.currentThread().getName());

EventBus.getDefault().post(new MessageEvent());

}

});

打印结果如下：

log --> E/postEvent﹕ main

log --> E/PostThread﹕ main

log --> E/Async﹕ pool-1-thread-1

log --> E/MainThread﹕ main

log --> E/BackgroundThread﹕ pool-1-thread-2

从日志打印结果可以看出，如果在UI线程中发布事件，则线程模型为PostThread的事件处理函数也执行在UI线程，与发布事件的线程一致。线程模型为Async的事件处理函数执行在名字叫做pool-1-thread-1的新的线程中。而MainThread的事件处理函数执行在UI线程，BackgroundThread的时间处理函数执行在名字叫做pool-1-thread-2的新的线程中。

我们再看看在子线程中发布一条MessageEvent的消息时，会有什么样的结果。

findViewById(R.id.send).setOnClickListener(new View.OnClickListener() {

@Override

public void onClick(View v) {

new Thread(new Runnable() {

@Override

public void run() {

Log.e("postEvent", Thread.currentThread().getName());

EventBus.getDefault().post(new MessageEvent());

}

}).start();

}

});

打印结果如下：

log --> E/postEvent﹕ Thread-125

log --> E/PostThread﹕ Thread-125

log --> E/BackgroundThread﹕ Thread-125

log --> E/Async﹕ pool-1-thread-1

log --> E/MainThread﹕ main

从日志打印结果可以看出，如果在子线程中发布事件，则线程模型为PostThread的事件处理函数也执行在子线程，与发布事件的线程一致（都是Thread-125）。BackgroundThread事件模型也与发布事件在同一线程执行。Async则在一个名叫pool-1-thread-1的新线程中执行。MainThread还是在UI线程中执行。

## 3.5黏性事件 事件发布后，有些事件携带有兴趣的信息。例如，事件表明一些初始化完成。或者如果您有一些传感器或位置数据，并且要保持最近的值。而不是实现自己的缓存，您可以使用粘性事件。所以EventBus将某个类型的最后一个粘性事件保留在内存中。然后可以将粘性事件传递给用户或明确查询。因此，您不需要任何特殊的逻辑来考虑已经可用的数据。

### 3.5.1发送黏性事件：

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

### 3.5.2注册订阅

**现在开始一个新的活动。在注册期间，所有粘性订阅者方法将立即获得以前发布的粘性事件：**

|  |  |
| --- | --- |
| 2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | @Override  public void onStart() {      super.onStart();      EventBus.getDefault().register(this);  }    // UI updates must run on MainThread  @Subscribe(sticky = true, threadMode = ThreadMode.MAIN)  public void onEvent(MessageEvent event) {      textField.setText(event.message);  }    @Override  public void onStop() {      EventBus.getDefault().unregister(this);      super.onStop(); |

### 3.5.3 手动获取和删除粘性事件 如您所见，最后一个粘性事件在注册时自动发送到匹配的订阅者。但有时手动检查粘性事件可能会更方便。也可能需要删除（消费）粘性事件，以便它们不再被传送。例：

方法 removeStickyEvent重载：当您传入类时，它将返回以前持有的粘性事件。使用这种变体，我们可以改进前面的例子：

## **3.6 配置**

EventBusBuilder用来配置EventBus。比如，如果一个提交的事件没有订阅者，可以使EventBus保持安静。

EventBus eventBus = EventBus.builder().logNoSubscriberMessages(false)

.sendNoSubscriberEvent(false).build()

另一个例子是当一个订阅者抛出一个异常的失败。注意：默认情况下，EventBus捕获异常从onEvent方法中抛出并且发出一个SubscriberExceptionEvent ，这个事件可以不必处理。

EventBus eventBus = EventBus.builder().throwSubscriberException(true).build();

**配置默认EventBus实例**使用EventBus.getDefault()是一种简单的方法来获取共享的EventBus实例。EventBusBuilder也可以使用installDefaultEventBus()方法来配置这个默认的实例。比如，当在onEvent方法中发生异常的时候，可以配置默认的EventBus实例来重新抛出异常。建议在使用DEBUG模式的时候这么使用，因为这样app会因为这个异常而崩溃。

EventBus.builder().throwSubscriberException(BuildConfig.DEBUG)

.installDefaultEventBus();

注意：只有在默认EventBus实例在第一次使用之前这么配置一次。后续调用installDefaultEventBus() 会抛出异常。这确保应用程序的行为一致。可以在Application类中配置默认的EventBus。

## **3.7 用户索引(**Subscriber Index**)**

用户索引是EventBus 3的一个新功能。它是可选的优化，可加速初始用户注册。

订阅者索引可以在构建时使用EventBus注释处理器创建。虽然不需要使用索引，但建议在Android上获得最佳性能。

索引前提条件  
请注意，只有@Subscriber方法可以索引，用户和事件类是公共的。另外，由于Java注释处理本身的技术限制，@Subscribe 注释在匿名类中无法识别。

当EventBus不能使用索引时，它将在运行时自动回退到反射。因此，它仍然可以工作，只是稍慢一些。

### 3.7.1使用annotationProcessor

如果您不使用Android Gradle Plugin 2.2.0或更高版本，请使用android-apt的配置。

要启用索引生成，您需要使用annotationProcessor  属性将EventBus注释处理器添加到构建中 。还设置一个参数 eventBusIndex来指定要生成的索引的完全限定类。例如，将以下部分添加到您的Gradle构建脚本中：

android {

    defaultConfig {

        javaCompileOptions {

            annotationProcessorOptions {

                arguments = [ eventBusIndex : 'com.example.myapp.MyEventBusIndex' ]

            }

        }

    }

}

dependencies {

    compile 'org.greenrobot:eventbus:3.0.0'

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

}

### 3.7.2使用android-apt

如果上述不适用于您，则可以使用[android-apt](https://bitbucket.org/hvisser/android-apt)  Gradle插件将EventBus注释处理器添加到您的构建中。将以下部分添加到您的Gradle构建脚本中：

buildscript {

    dependencies {

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

    }

}

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

dependencies {

    compile 'org.greenrobot:eventbus:3.0.0'

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

}

apt {

    arguments {

        eventBusIndex "com.example.myapp.MyEventBusIndex"

    }

}

当您下次构建项目（没有错误）时，将为您生成使用eventBusIndex指定的类 。然后当设置EventBus传递它如下所示

EventBus eventBus = EventBus.builder().addIndex(new MyEventBusIndex()).build();

或者，如果要在整个应用程序中使用默认实例：

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

// Now the default instance uses the given index. Use it like this:

EventBus eventBus = EventBus.getDefault();

### 3.7.3加入索引

您可以将相同的原则应用于作为库的一部分（而不是最终应用程序）的代码。这样，您可能会有多个索引类，您可以在EventBus设置期间添加所有索引类，例如：

EventBus eventBus = EventBus.builder()

    .addIndex(new MyEventBusAppIndex())

    .addIndex(new MyEventBusLibIndex()).build();

# EventBus源码解析

## **知识补充**

### **android-apt是什么?**

android-apt 是一个Gradle插件，协助Android Studio 处理annotation processors, 它有两个目的：

1. 允许配置只在编译时作为注解处理器的依赖，而不添加到最后的APK或library
2. 设置源路径，使注解处理器生成的代码能被Android Studio正确的引用

1.添加android-apt到构建脚本中

使用该插件，添加如下到你的构建脚本中  
//配置在Project下的build.gradle中

buildscript {

repositories {

mavenCentral()

}

dependencies {

//替换成最新的 gradle版本

classpath 'com.android.tools.build:gradle:1.3.0'

//替换成最新android-apt版本

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

}

}

//配置到Module下的build.gradle中

apply plugin: 'com.android.application'

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

2.使用annotationProcessor

android {

defaultConfig {

javaCompileOptions {

annotationProcessorOptions {

arguments = [eventBusIndex:'com.monster.android.wild.MyEventBusIndex']

}

}

}

}

dependencies {

compile 'org.greenrobot:eventbus:3.0.0'

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

}

传递处理器参数

有些注解处理器需要传递参数，你能通过使用apt.arguments达到此目的，如下是一个向AndroidAnnotations传递参数的例子

apt {

arguments {

resourcePackageName android.defaultConfig.applicationId

androidManifestFile variant.outputs[0]?.processResources?.manifestFile

}

}

### 自定义注解

注解(Annotation)相当于一种标记，在程序中加入注解就等于为程序打上某种标记，没有加，则等于没有任何标记，以后，javac编译器、开发工具和其他程序可以通过反射来了解你的类及各种元素上有无何种标记，看你的程序有什么标记，就去干相应的事，标记可以加在包、类，属性、方法，方法的参数以及局部变量上。

@Retention(RetentionPolicy.RUNTIME)

14 //Retention注解决定MyAnnotation注解的生命周期

15 @Target( { ElementType.METHOD, ElementType.TYPE })

16 //Target注解决定MyAnnotation注解可以加在哪些成分上，如加在类身上，或者属性身上，或者方法身上等成分

17 /\*

18 \* @Retention(RetentionPolicy.SOURCE)

19 \* 这个注解的意思是让MyAnnotation注解只在java源文件中存在，编译成.class文件后注解就不存在了

20 \* @Retention(RetentionPolicy.CLASS)

21 \* 这个注解的意思是让MyAnnotation注解在java源文件(.java文件)中存在，编译成.class文件后注解也还存在，

22 \* 被MyAnnotation注解类标识的类被类加载器加载到内存中后MyAnnotation注解就不存在了

23 \*/

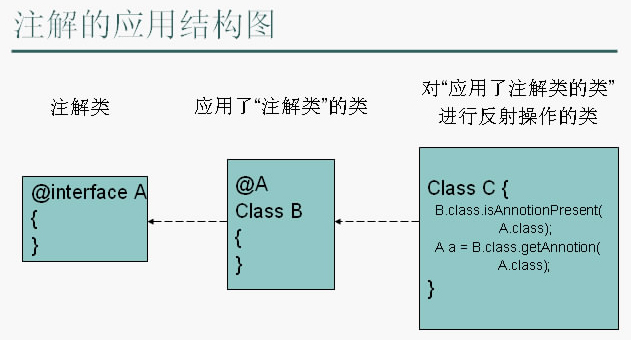
24 /\*

25 \* 这里是在注解类MyAnnotation上使用另一个注解类，这里的Retention称为元注解。

26 \* Retention注解括号中的"RetentionPolicy.RUNTIME"意思是让MyAnnotation这个注解的生命周期一直程序运行时都存在

27 \*/

28 public @interface MyAnnotation {}



　　注解就相当于一个你的源程序要调用一个类，在源程序中应用某个注解，得事先准备好这个注解类。就像你要调用某个类，得事先开发好这个类。

### ****ThreadLocal****

早在JDK 1.2的版本中就提供[**Java**](http://lib.csdn.net/base/javase).lang.ThreadLocal，ThreadLocal为解决多线程程序的并发问题提供了一种新的思路。使用这个工具类可以很简洁地编写出优美的多线程程序。

　　当使用ThreadLocal维护变量时，ThreadLocal为每个使用该变量的线程提供独立的变量副本，所以每一个线程都可以独立地改变自己的副本，而不会影响其它线程所对应的副本。

　　从线程的角度看，目标变量就象是线程的本地变量，这也是类名中“Local”所要表达的意思。

　　所以，在Java中编写线程局部变量的代码相对来说要笨拙一些，因此造成线程局部变量没有在Java开发者中得到很好的普及。

## EventBus注册与反注册流程源码分析

public void register(Object subscriber) {

Class<?> subscriberClass = subscriber.getClass(); //获得订阅者的class

List<SubscriberMethod> subscriberMethods = subscriberMethodFinder.findSubscriberMethods(subscriberClass); // 查找订阅者类的所有订阅方法

synchronized (this) {

for (SubscriberMethod subscriberMethod : subscriberMethods) {

subscribe(subscriber, subscriberMethod);

}

}

}

从上述代码可以看出，注册的大致流程为：

1. 查找到订阅者类的所有订阅方法
2. 注册订阅信息  
   接下来我们来详细分析通过SubscriberMethodFinder来查找订阅方法的过程，在介绍之前我们先详细看一下SubscriberMethod

public class SubscriberMethod {

final Method method; // 订阅方法

final ThreadMode threadMode; // 标识在哪个线程执行，有POSTING，MAIN,BACKGROUND,ASYNC 四种模式

final Class<?> eventType; // 事件类

final int priority; // 优先级

final boolean sticky; // 是否是粘性事件

/\*\* Used for efficient comparison \*/

String methodString;

....

}

SubscriberMethod 包含了订阅方法相关的所有信息，接下来我们看SubscriberMethodFinder中是如何查找事件类中的所有订阅方法信息的

List<SubscriberMethod> findSubscriberMethods(Class<?> subscriberClass) {

List<SubscriberMethod> subscriberMethods = METHOD\_CACHE.get(subscriberClass); //首先从METHOD\_CACHE中读取缓存下来的订阅方法列表

if (subscriberMethods != null) {

return subscriberMethods;

}

// ignoreGeneratedIndex属性表示是否忽略注解器生成的MyEventBusIndex。如何生成MyEventBusIndex类以及他的使用，可以本文中apt部分的详细说明。ignoreGeneratedIndex的默认值为false，可以通过EventBusBuilder来设置它的值

if (ignoreGeneratedIndex) {

// 利用反射来获取订阅类中所有订阅方法信息

subscriberMethods = findUsingReflection(subscriberClass);

} else {

// 从注解器生成的MyEventBusIndex类中获得订阅类的订阅方法信息

subscriberMethods = findUsingInfo(subscriberClass);

}

if (subscriberMethods.isEmpty()) {

throw new EventBusException("Subscriber " + subscriberClass

+ " and its super classes have no public methods with the @Subscribe annotation");

} else {

METHOD\_CACHE.put(subscriberClass, subscriberMethods);

return subscriberMethods;

}

}

从上面代码可以看出，SubscriberMethodFinder获得订阅方法信息列表的步骤为：

1. 首先从订阅方法缓存中获取订阅类的订阅方法，如果没有则进入步骤2
2. 通过ignoreGeneratedIndex属性判断通过以下两种方式中的一种来查找订阅方法，这两种方式为  
   -如果ignoreGeneratedIndex为FALSE，通过EventBusAnnotationProcessor（注解处理器）生成的MyEventBusIndex中获取  
   -否则通过反射来获取订阅类中订阅方法信息  
   关于EventBusAnnotationProcessor注解处理器在本文中apt部分的介绍中有详细阐述，此处不再赘述。我们详细解读一下通过反射来获取订阅类中订阅方法信息列表的源代码
3. private List<SubscriberMethod> findUsingReflection(Class<?> subscriberClass) {
4. FindState findState = prepareFindState();
5. findState.initForSubscriber(subscriberClass); // 初始化FindState
6. while (findState.clazz != null) {
7. findUsingReflectionInSingleClass(findState);// 寻找某个类中的所有事件响应方法
8. findState.moveToSuperclass(); //继续寻找当前类父类中注册的事件响应方法
9. }
10. return getMethodsAndRelease(findState);
11. }

分析以上源码，具体步骤如下：

1. 通过反射得到当前 class 的所有方法
2. 过滤掉不是 public 和是 abstract、static、bridge、synthetic 的方法
3. 找出所有参数只有一个的方法
4. 找出被Subscribe注解的方法
5. 把method 方法和 事件类型eventtype添加到 findState 中
6. 把 method 方法、事件类型、threadMode、priority、sticky 封装成 SubscriberMethod 对象，然后添加到 findState.subscriberMethods  
   以上为查找订阅者类中所有订阅方法信息的详细过程，查找完订阅类中所有订阅方法信息后，遍历所有订阅方法信息，继续进行注册的步骤
7. private void subscribe(Object subscriber, SubscriberMethod subscriberMethod) {
8. Class<?> eventType = subscriberMethod.eventType; // 获取订阅方法的事件类
9. Subscription newSubscription = new Subscription(subscriber, subscriberMethod); // 创建订阅类
10. CopyOnWriteArrayList<Subscription> subscriptions = subscriptionsByEventType.get(eventType); // 获取订阅了此事件类的所有订阅者信息列表，如果不存在则将其加入订阅者信息列表，如果已经存在此订阅者信息则抛出已注册的异常
11. if (subscriptions == null) {
12. subscriptions = new CopyOnWriteArrayList<>();
13. subscriptionsByEventType.put(eventType, subscriptions);
14. } else {
15. if (subscriptions.contains(newSubscription)) {
16. throw new EventBusException("Subscriber " + subscriber.getClass() + " already registered to event "
17. + eventType);
18. }
19. }
20. int size = subscriptions.size();
21. for (int i = 0; i <= size; i++) {
22. if (i == size || subscriberMethod.priority > subscriptions.get(i).subscriberMethod.priority) {
23. subscriptions.add(i, newSubscription);
24. break;
25. }
26. } // 将订阅者信息按照优先级加入到订阅者信息列表中
27. List<Class<?>> subscribedEvents = typesBySubscriber.get(subscriber); // 获取此订阅者实例订阅的所有事件类的列表
28. if (subscribedEvents == null) {
29. subscribedEvents = new ArrayList<>();
30. typesBySubscriber.put(subscriber, subscribedEvents);
31. }
32. subscribedEvents.add(eventType); // 将此事件类加入 订阅者事件类列表中
33. // 判断是否是粘性事件，对粘性事件立即进行分发
34. if (subscriberMethod.sticky) {
35. if (eventInheritance) {
36. // 所有事件类的子类都要处理
37. // Existing sticky events of all subclasses of eventType have to be considered.
38. // Note: Iterating over all events may be inefficient with lots of sticky events,
39. // thus data structure should be changed to allow a more efficient lookup
40. // (e.g. an additional map storing sub classes of super classes: Class -> List<Class>).
41. Set<Map.Entry<Class<?>, Object>> entries = stickyEvents.entrySet();
42. for (Map.Entry<Class<?>, Object> entry : entries) {
43. Class<?> candidateEventType = entry.getKey();
44. if (eventType.isAssignableFrom(candidateEventType)) {
45. Object stickyEvent = entry.getValue();
46. checkPostStickyEventToSubscription(newSubscription, stickyEvent);
47. }
48. }
49. } else {
50. // 如果当前粘性事件集合中存在此事件类对应的粘性事件对象，则进入事件分发流程
51. Object stickyEvent = stickyEvents.get(eventType);
52. checkPostStickyEventToSubscription(newSubscription, stickyEvent);
53. }
54. }
55. }

7如果当前订阅信息没有注册过，则按照优先级加入到订阅信息集合 subscriptionsByEventType(subscriptionsByEventType以事件类为key，以订阅信息列表为value的订阅信息集合)。

8将事件类型加入到订阅事件类型集合 typesBySubscriber(typesBySubscriber 以订阅者对象为key，以订阅事件类型为value的订阅类型信息集合)。

9如果当前订阅的为粘性事件，则进入事件分发的流程。  
Map<Class<?>, Object> stickyEvents; 以事件类为key，以事件对象为value的粘性事件集合  
Subscription是订阅信息类，包含订阅者对象，订阅方法，是否处于激活状态等

final class Subscription {

final Object subscriber; //　订阅者对象

final SubscriberMethod subscriberMethod; // 订阅方法

/\*\*

\* Becomes false as soon as {@link EventBus#unregister(Object)} is called, which is checked by queued event delivery

\* {@link EventBus#invokeSubscriber(PendingPost)} to prevent race conditions.

\*/

volatile boolean active; // 是否处于激活状态

Subscription(Object subscriber, SubscriberMethod subscriberMethod) {

this.subscriber = subscriber;

this.subscriberMethod = subscriberMethod;

active = true;

}

@Override

public boolean equals(Object other) {

if (other instanceof Subscription) {

Subscription otherSubscription = (Subscription) other;

return subscriber == otherSubscription.subscriber

&& subscriberMethod.equals(otherSubscription.subscriberMethod);

} else {

return false;

}

}

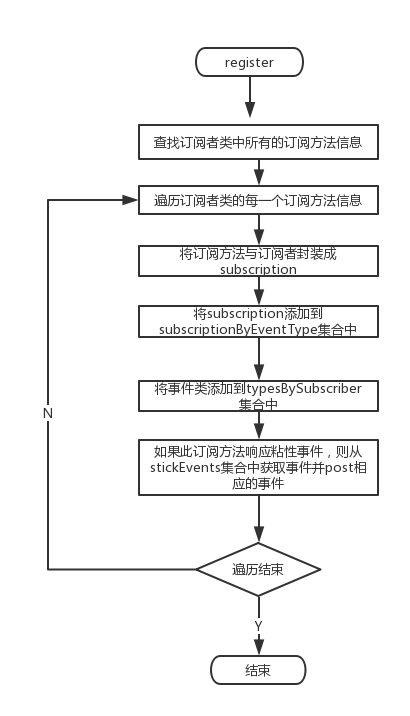
@Override

public int hashCode() {

return subscriber.hashCode() + subscriberMethod.methodString.hashCode();

}

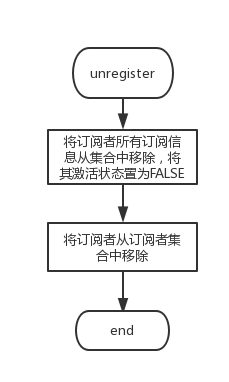
}

注册的流程图如下  


反注册的流程为：

1. 将订阅者的订阅信息从订阅信息集合中移除，同时将订阅信息激活状态置为FALSE
2. 将订阅者信息从订阅者订阅类型集合中移除

注册的流程图如下：



## EventBus Post分析 先简单的描述下大体过程： 从EventBus的总线LIst中找出订阅了这个event的方法Subscription，然后根据Subscription中的method指定的不同线程信息，将这个方法放置在相应线程中调用。 先从入口post方法看起：

private final ThreadLocal<PostingThreadState> currentPostingThreadState = new ThreadLocal<PostingThreadState>() {

@Override

protected PostingThreadState initialValue() {

return new PostingThreadState();

}

};

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

public void post(Object event) {

//currentPostingThreadState是一个ThreadLocal，他的特点是获取当前线程一份独有的变量数据，不受其他线程影响。

PostingThreadState postingState = currentPostingThreadState.get();

//postingState就是获取到的线程独有的变量数据

List<Object> eventQueue = postingState.eventQueue;

//把post的事件添加到事件队列

eventQueue.add(event);

// 如果没有处在事件发布状态，那么开始发送事件并一直保持发布状态

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()) {

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

}

} finally {

postingState.isPosting = false;

postingState.isMainThread = false;

}

}

}

/\*\* For ThreadLocal, much faster to set (and get multiple values). \*/

final static class PostingThreadState {

//事件队列

final List<Object> eventQueue = new ArrayList<Object>();

boolean isPosting;

boolean isMainThread;

Subscription subscription;

Object event;

boolean canceled;

}

在 post方法中，大致流程为 :

* 首先根据 currentPostingThreadState 获取当前线程状态 postingState 。currentPostingThreadState 其实就是一个 ThreadLocal 类的对象，不同的线程根据自己独有的索引值可以得到相应属于自己的 postingState 数据。
* 然后把事件 event 加入到 eventQueue 队列中排队。
* 循环遍历 eventQueue ，取出事件发送事件。发送单个事件是调用 postSingleEvent(Object event, PostingThreadState postingState) 方法。

我们跟进去看postSingleEvent方法源码：  
private void postSingleEvent(Object event, PostingThreadState postingState) throws Error {

// 得到事件的类型

Class<?> eventClass = event.getClass();

// 是否找到订阅者

boolean subscriptionFound = false;

// 如果支持事件继承，默认为支持

if (eventInheritance) {

// 查找 eventClass 的所有父类和接口

List<Class<?>> eventTypes = lookupAllEventTypes(eventClass);

int countTypes = eventTypes.size();

for (int h = 0; h < countTypes; h++) {

Class<?> clazz = eventTypes.get(h);

// 依次向 eventClass 的父类或接口的订阅方法发送事件

// 只要有一个事件发送成功，返回 true ，那么 subscriptionFound 就为 true

subscriptionFound |= postSingleEventForEventType(event, postingState, clazz);

}

} else {

// 发送事件

subscriptionFound = postSingleEventForEventType(event, postingState, eventClass);

}

// 如果没有订阅者

if (!subscriptionFound) {

if (logNoSubscriberMessages) {

Log.d(TAG, "No subscribers registered for event " + eventClass);

}

if (sendNoSubscriberEvent && eventClass != NoSubscriberEvent.class &&

eventClass != SubscriberExceptionEvent.class) {

// 发送 NoSubscriberEvent 事件，可以自定义接收

post(new NoSubscriberEvent(this, event));

}

}

}

在postSingleEvent方法中，事件发送根据 eventInheritance 分成两种，大致流程为：

* eventInheritance 默认为true，支持事件继承：得到 eventClass 的所有父类和接口，然后循环依次发送事件；
* eventInheritance 默认为false,不支持事件继承：直接发送eventClass 事件。
* 若找不到订阅者，默认会发送 NoSubscriberEvent 事件。开发者可以自定义订阅方法接收这个事件。

关于事件发送的具体操作进一步到 postSingleEventForEventType方法中去看：

private boolean postSingleEventForEventType(Object event, PostingThreadState postingState, Class<?> eventClass) {

CopyOnWriteArrayList<Subscription> subscriptions;

synchronized (this) {

// 得到订阅者

subscriptions = subscriptionsByEventType.get(eventClass);

}

if (subscriptions != null && !subscriptions.isEmpty()) {

// 依次遍历订阅者

for (Subscription subscription : subscriptions) {

postingState.event = event;

postingState.subscription = subscription;

boolean aborted = false;

try {

// 发送事件

postToSubscription(subscription, event, postingState.isMainThread);

// 是否被取消了

aborted = postingState.canceled;

} finally {

postingState.event = null;

postingState.subscription = null;

postingState.canceled = false;

}

// 如果被取消，则跳出循环

if (aborted) {

break;

}

}

return true;

}

return false;

}

postSingleEventForEventType方法主要做的事情：

* 从订阅者注册列表中取出eventClass事件对应的订阅者列表，
* 遍历了订阅者，然后依次调用 postToSubscription 方法发送事件。

具体的事件发送操作又到了postToSubscription 方法中。

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

// 根据不同的线程模式执行对应

switch (subscription.subscriberMethod.threadMode) {

case POSTING: // 和发送事件处于同一个线程

invokeSubscriber(subscription, event);

break;

case MAIN: // 主线程

if (isMainThread) {

invokeSubscriber(subscription, event);

} else {

mainThreadPoster.enqueue(subscription, event);

}

break;

case BACKGROUND: // 子线程

if (isMainThread) {

backgroundPoster.enqueue(subscription, event);

} else {

invokeSubscriber(subscription, event);

}

break;

case ASYNC: // 和发送事件处于不同的线程

asyncPoster.enqueue(subscription, event);

break;

default: // 抛出异常

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

}

}

void invokeSubscriber(Subscription subscription, Object event) {

try {

// 通过反射执行订阅方法

subscription.subscriberMethod.method.invoke(subscription.subscriber, event);

} catch (InvocationTargetException e) {

handleSubscriberException(subscription, event, e.getCause());

} catch (IllegalAccessException e) {

throw new IllegalStateException("Unexpected exception", e);

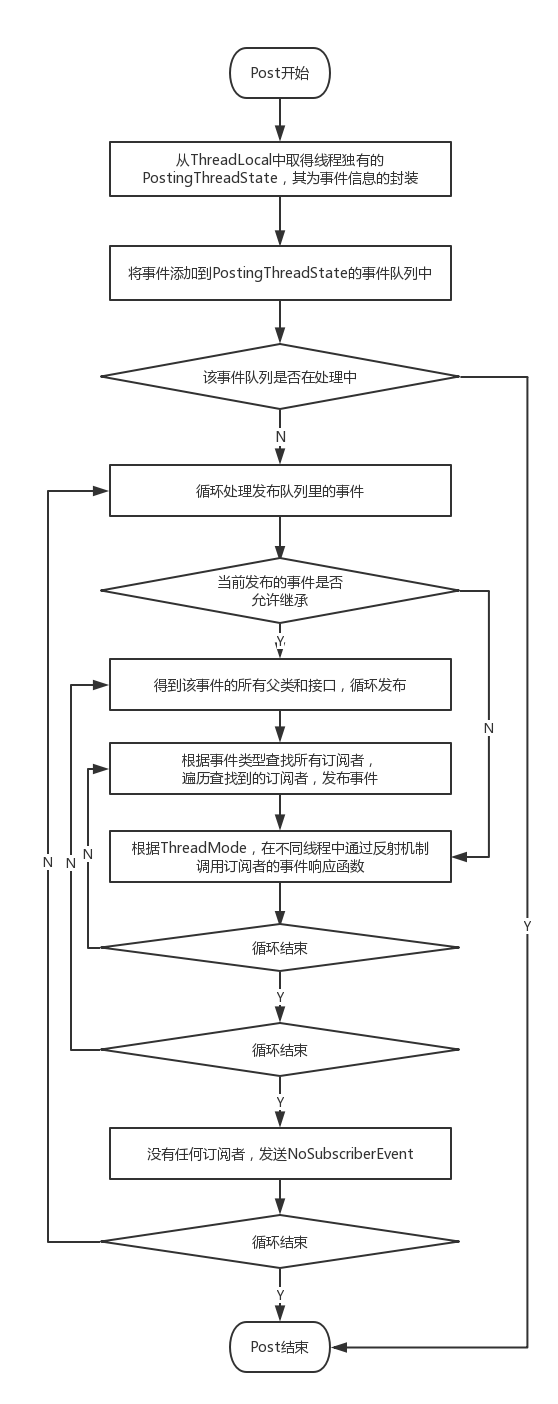
}

}

postToSubscription 方法中，根据 threadMode 不同将事件按发送线程模式 共分为四种：

* （POSTING）同一个线程：表示订阅方法所处的线程和发布事件的线程是同一个线程；
* （MAIN）主线程：如果发布事件的线程是主线程，那么直接执行订阅方法；否则利用 Handler 回调主线程来执行；
* （BACKGROUND）子线程：如果发布事件的线程是主线程，那么调用线程池中的子线程来执行订阅方法；否则直接执行；
* （ASYNC）异步线程：无论发布事件执行在主线程还是子线程，都利用一个异步线程来执行订阅方法。  
  这四种线程模式其实最后都会调用 invokeSubscriber(Subscription subscription, Object event) 方法通过反射来调用注册的方法，从而实现了将事件event发送的了对应的注册者方法里处理。

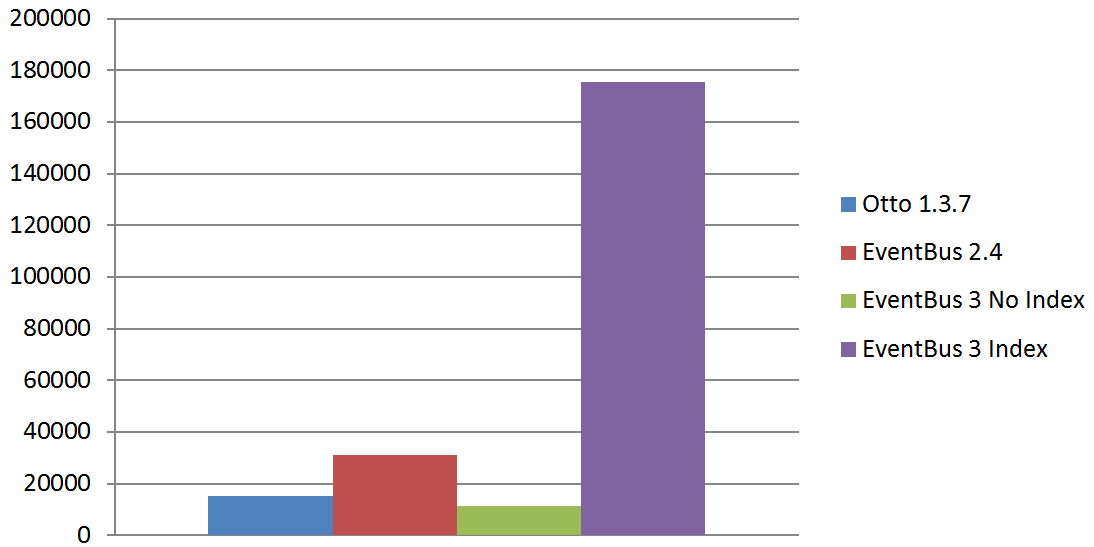
整个代码流程整体是线性，最后附上流程图：



# 总结

1.使用Subscriber Index

Subscriber Index是EventBus 3.0新引入的功能，它能大幅提升**初始**注册过程的效率。其原理其实很简单，就是把运行时期做的事情放到了编译时期去做，从而提升了运行时期的效率。大家可以从EventBus作者的博客中的对比图得到感性的认识。 可以看到，开启了索引的3.0，其注册速度比2.4快了3到6倍。



2．事件分类 要分开 加快查找订阅者的速度

subscriptionsByEventType = new HashMap<>();//以事件类的class对象为键值，记录注册方法信息，值为一个Subscription的列表