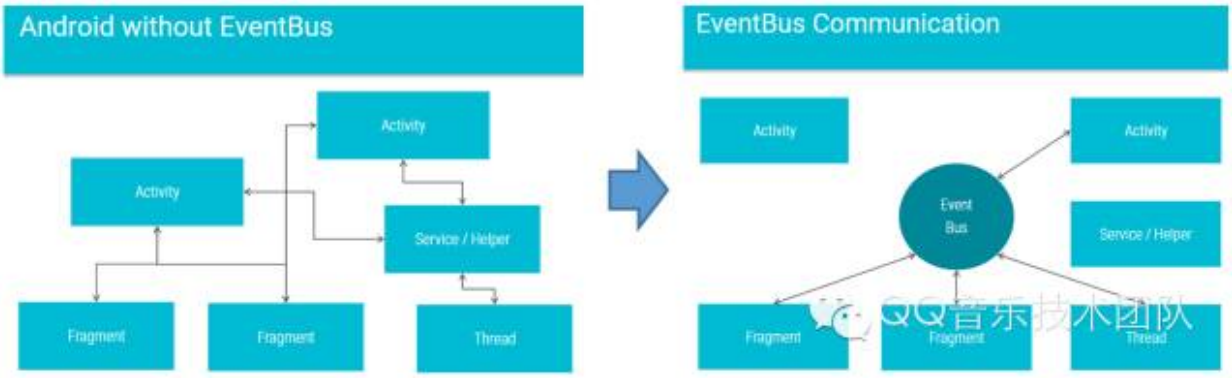
老司机教你“飙”EventBus3

2016-04-28 sangerzhong QQ音乐技术团队

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。



作为**Markus Junginger**大神耗时4年打磨、超过1亿接入量、Github 9000+ star的明星级组件，分析EventBus的文章早已是数不胜数。本文的题目是“教你飙巴士”，而这辆Bus之所以可以飙起来，是因为作者在EventBus 3中引入了**EventBusAnnotationProcessor**（注解分析生成索引）技术，大大提高了EventBus的运行效率。而分析这个加速器的资料在网上很少，因此本文会把重点放在分析这个EventBus 3的新特性上，同时分享一些踩坑经验，并结合源码分析及UML图，以直观的形式和大家一起学习EventBus 3的用法及运行原理。

1. 新手上路——使用EventBus

1.1 导入组件:

打开App的build.gradle，在dependencies中添加最新的EventBus依赖：

```
compile 'org.greenrobot:eventbus:3.0.0'
```

如果不需要索引加速的话，就可以直接跳到第二步了。而要应用最新的EventBusAnnotationProcessor则比较麻烦，因为注解解析依赖于android-apt-plugin。我们一步一步来，首先在项目gradle的dependencies中引入apt编译插件：

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

然后在App的build.gradle中应用apt插件，并设置apt生成的索引的包名和类名：

```
apply plugin: 'com.neenbedankt.android-apt'
apt {
    arguments {
        eventBusIndex "com.study.sangerzhong.studyapp.MyEventBusIndex"
    }
}
```

接着在App的dependencies中引入EventBusAnnotationProcessor：

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

这里需要注意，如果应用了EventBusAnnotationProcessor却没有设置arguments的话，编译时就会报错：

`No option eventBusIndex passed to annotation processor`

此时需要我们先编译一次，生成索引类。编译成功之后，就会发现在`\ProjectName\app\build\generated\source\apt\PakageName\`下看到通过注解分析生成的索引类，这样我们便可以在初始化EventBus时应用我们生成的索引了。

1.2 初始化EventBus

EventBus默认有一个单例，可以通过 `getDefault()` 获取，也可以通过 `EventBus.builder()` 构造自定义的EventBus，比如要应用我们生成好的索引时：

```
EventBus mEventBus = EventBus.builder().addIndex(new MyEventBusIndex()).build();
```

如果想把自定义的设置应用到EventBus默认的单例中，则可以用 `installDefaultEventBus()` 方法：

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

1.3 定义事件:

所有能被实例化为Object的实例都可以作为事件：

```
public class DriverEvent { public String info; }
```

在最新版的eventbus 3中如果用到了索引加速，事件类的修饰符必须为public，不然编译时会报错： `Subscriber method must be public`

1.4 监听事件:

首先把作为订阅事件的模块通过EventBus注册监听：

```
mEventBus.register(this);
```

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：

```
@Subscribe(threadMode = ThreadMode.POSTING, priority = 0, sticky = true)
public void handleEvent(DriverEvent event) {
    Log.d(TAG, event.info);
}
```

注解有三个参数，threadMode为回调所在的线程，priority为优先级，sticky为是否接收黏性事件。调度单位从类细化到了方法，对方法的命名也没有了要求，方便混淆代码。

但注册了监听的模块必须有一个标注了Subscribe注解方法，不然在register时会抛出异常：

`Subscriber class XXX and its super classes have no public methods with the @Subscribe annotation`

1.5 发送事件:

调用post或者postSticky即可：

```
mEventBus.post(new DriverEvent("magnet:?xt=urn:btih……"));
```

到此我们就完成了使用EventBus的学习，可以在代码中尽情地飚车了。项目接入了EventBus之后会有什么好处呢？举一个常见的用例：ViewPager中Fragment的相互通信，就不需要在容器中定义各种接口，可以直接通过EventBus来实现相互回调，这样就把逻辑从ViewPager这个容器中剥离出来，使代码阅读起来更加直观。

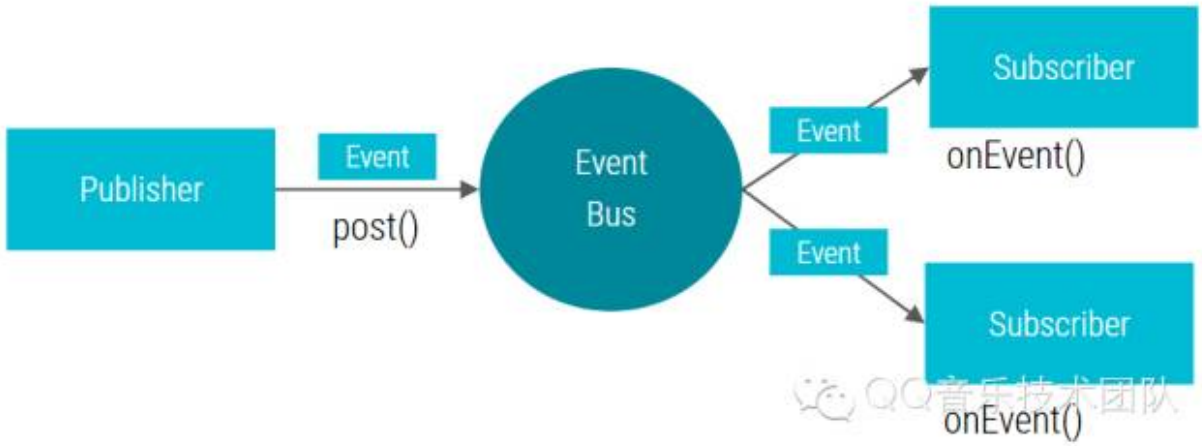
在实际项目的使用中，register和unregister通常与Activity和Fragment的生命周期相关，ThreadMode.MainThread可以很好地解决Android的界面刷新必须在UI线程的问题，不需要再回调后用Handler中转（EventBus中已经自动用Handler做了处理），黏性事件可以很好地解决post与register同时执行时的异步问题（这个在原理中会说到），事件的传递也没有序列化与反序列化的性能消耗，足以满足我们大部分情况下的模块间通信需求。

2. 变身老司机——EventBus原理分析

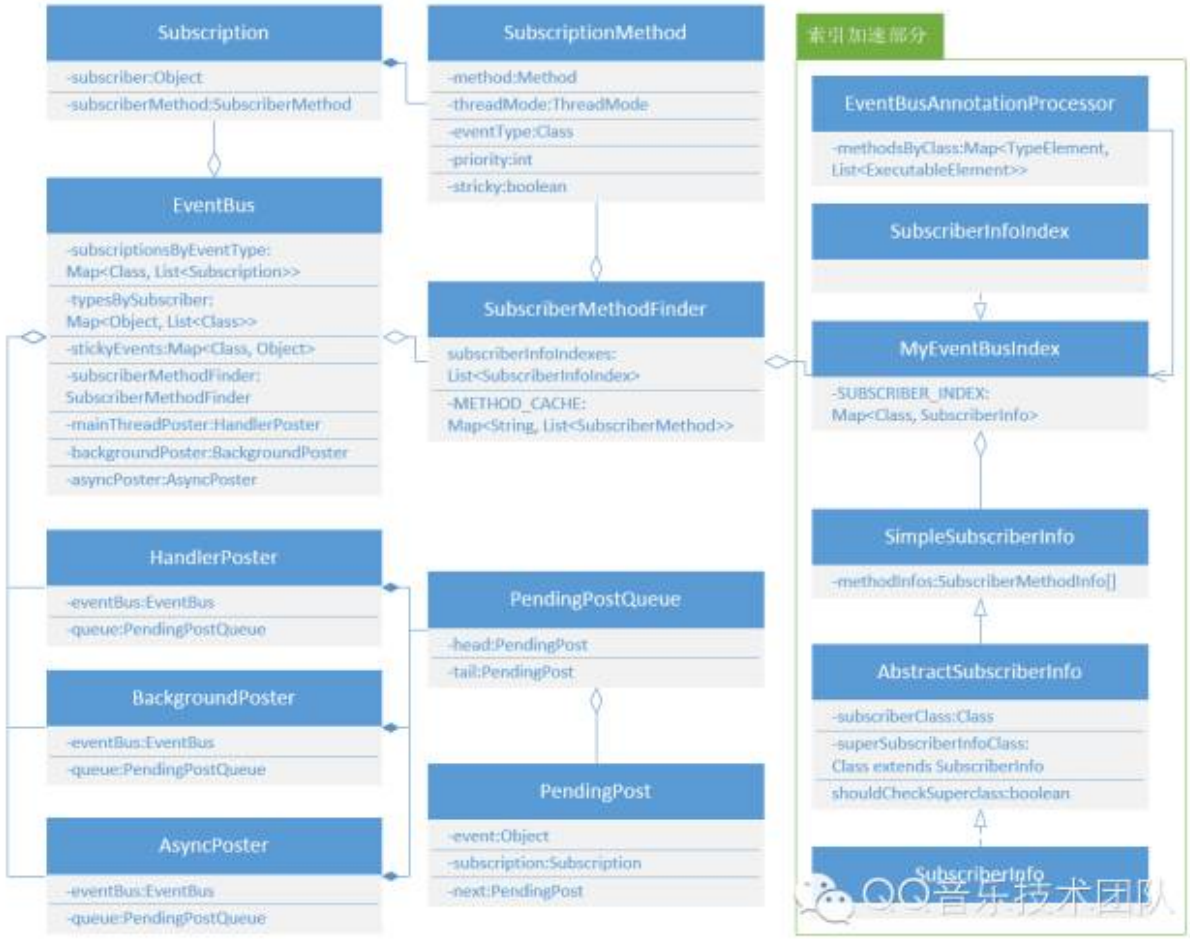
在平时使用中我们不需要关心EventBus中对事件的分发机制，但要成为能够快速排查问题的老司机，我们还是得熟悉它的工作原理，下面我们就透过UML图来学习一下。

2.1 核心架构

EventBus的核心工作机制透过作者Blog中的这张图就能很好地理解：



订阅者模块需要通过EventBus订阅相关的事件，并准备好处理事件的回调方法，而事件发布者则在适当的时机把事件post出去，EventBus就能帮我们搞定一切。在架构方面，EventBus 3与之前稍老版本有不同，我们直接看架构图：



先看核心类EventBus，其中subscriptionByEventType是以事件的类为key，订阅者的回调方法为value的映射关系表。也就是说EventBus在收到一个事件时，就可以根据这个事件的类型，在subscriptionByEventType中找到所有监听了该事件的订阅者及处理事件的回调方法。而typesBySubscriber则是每个订阅者所监听的事件类型表，在取消注册时可

以通过该表中保存的信息，快速删除subscriptionByEventType中订阅者的注册信息，避免遍历查找。注册事件、发送事件和注销都是围绕着这两个核心数据结构来展开。上面的Subscription可以理解为每个订阅者与回调方法的关系，在其他模块发送事件时，就会通过这个关系，让订阅者执行回调方法。

回调方法在这里被封装成了SubscriptionMethod，里面保存了在需要反射invoke方法时的各种参数，包括优先级，是否接收黏性事件和所在线程等信息。而要生成这些封装好的方法，则需要SubscriberMethodFinder，它可以在regster时得到订阅者的所有回调方法，并封装返回给EventBus。而右边的加速器模块，就是为了提高SubscriberMethodFinder的效率，会在第三章详细介绍，这里就不再啰嗦。

而下面的三个Poster，则是EventBus能在不同的线程执行回调方法的核心，我们根据不同的回调方式来看：

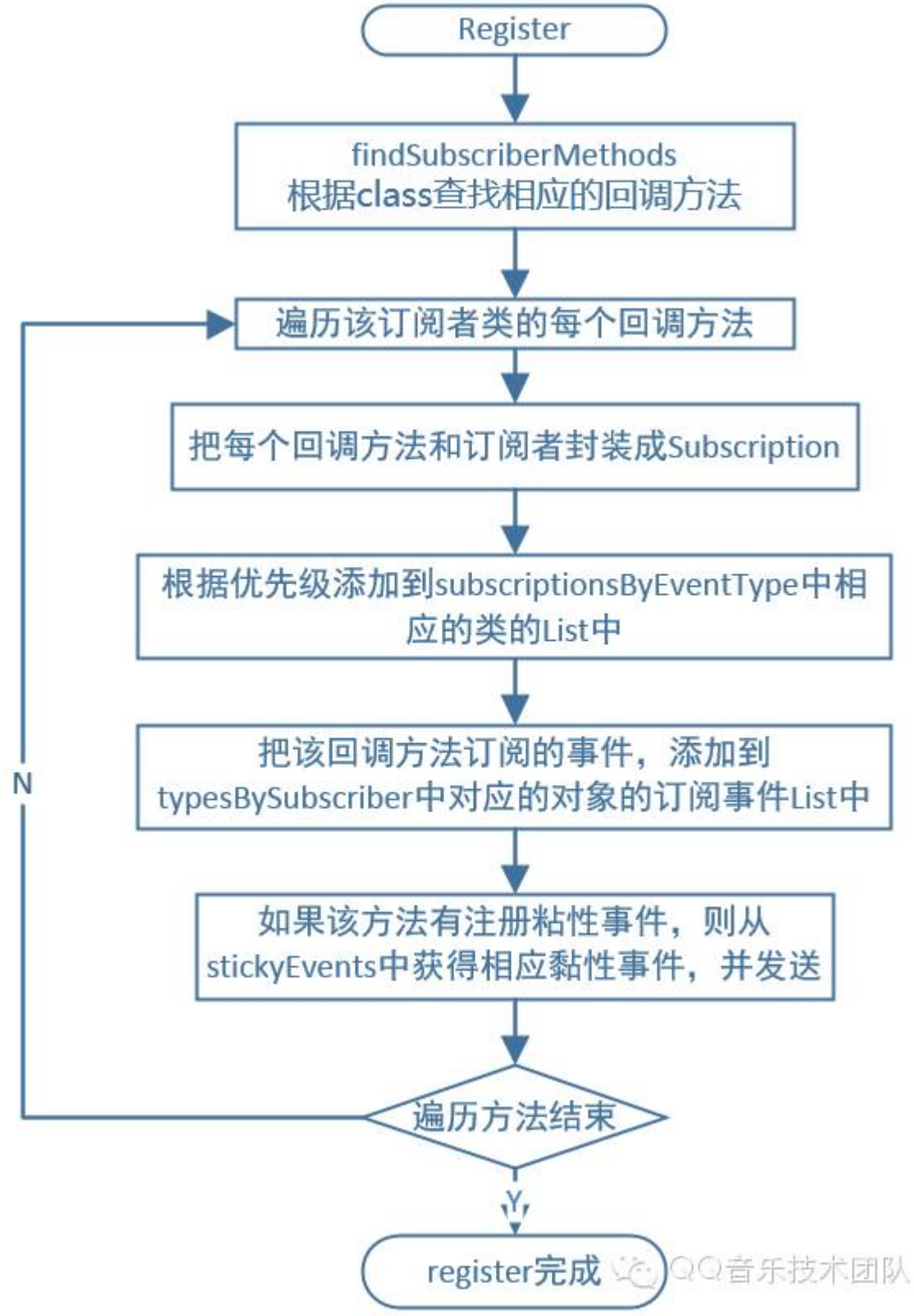
1. POSTING（在调用post所在的线程执行回调）：不需要poster来调度，直接运行。
2. MAIN（在UI线程回调）：如果post所在线程为UI线程则直接执行，否则则通过mainThreadPoster来调度。
3. BACKGROUND（在Backgroud线程回调）：如果post所在线程为非UI线程则直接执行，否则则通过backgroundPoster来调度。
4. ASYNC（交给线程池来管理）：直接通过asyncPoster调度。

可以看到，不同的Poster会在post事件时，调度相应的事件队列PendingPostQueue，让每个订阅者的回调方法收到相应的事件，并在其注册的Thread中运行。而这个事件队列是一个链表，由一个个PendingPost组成，其中包含了事件，事件订阅者，回调方法这三个核心参数，以及需要执行的下一个PendingPost。

至此EventBus 3的架构就分析完了，与之前EventBus老版本最明显的区别在于：分发事件的调度单位从订阅者，细化成了订阅者的回调方法。也就是说每个回调方法都有自己的优先级，执行线程和是否接收黏性事件，提高了事件分发的灵活程度，接下来我们在看核心功能的实现时更能体现这一点。

2.2 register

简单来说就是：根据订阅者的类来找回调方法，把订阅者和回调方法封装成关系，并保存到相应的数据结构中，为随后的事件分发做好准备，最后处理黏性事件：

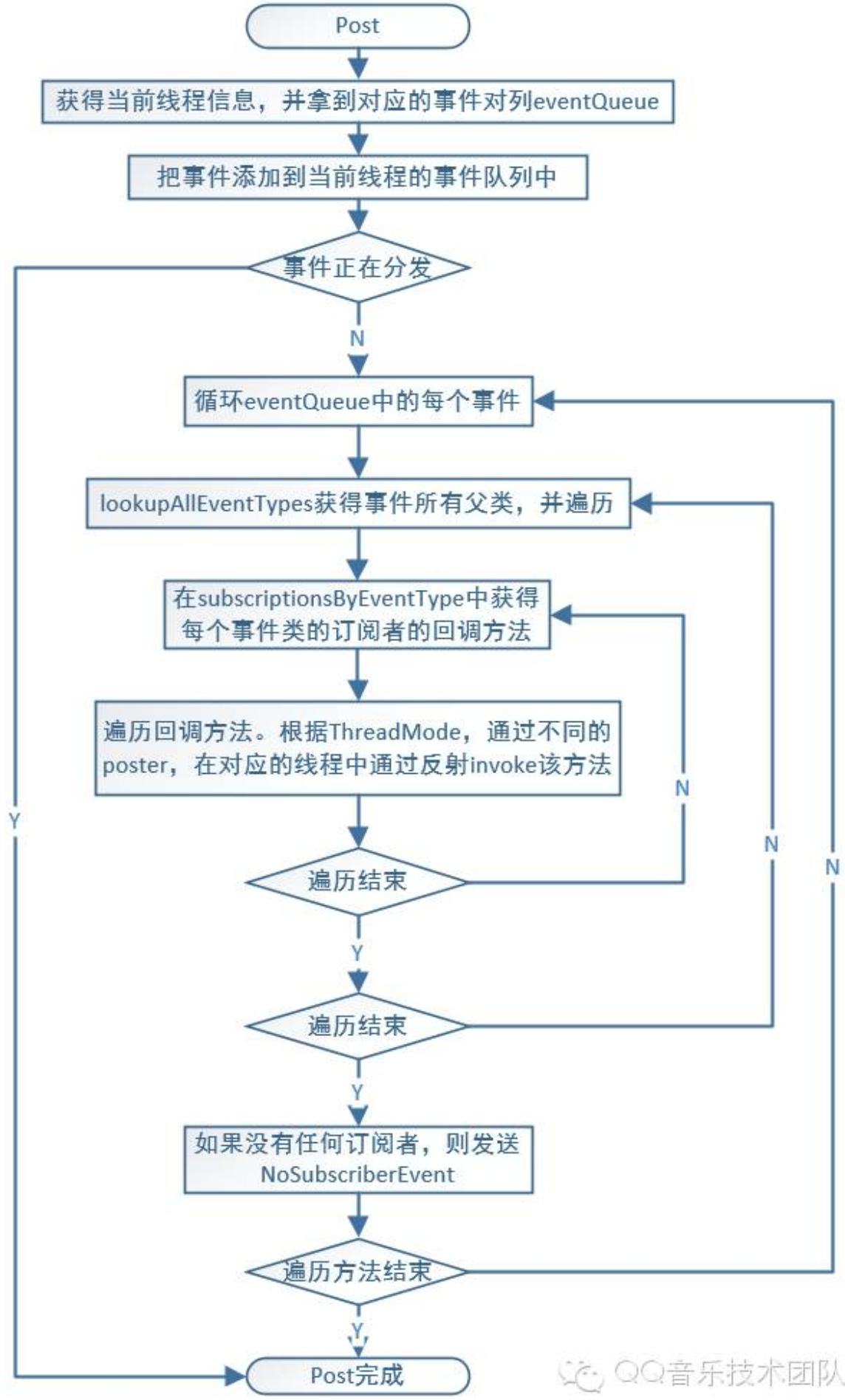


值得注意的是，老版本的EventBus是允许事件订阅者以不同的ThreadMode去监听同一个事件的，即在一个订阅者中有多个方法订阅一个事件，此时是无法保证这几个回调的先后顺序的，因为不同的线程回调是通过Handler调度的，有可能单个线程中的事件过多，事件受阻，回调则会比较慢。如今EventBus 3使用了注解来表示回调后，还可以出

现相同的ThreadMode的回调方法监听相同的事件，此时会根据注册的先后顺序，先注册先分发事件，注意不是先收到事件，收到事件的顺序还是得看poster中Handler的调度。

2.3 post

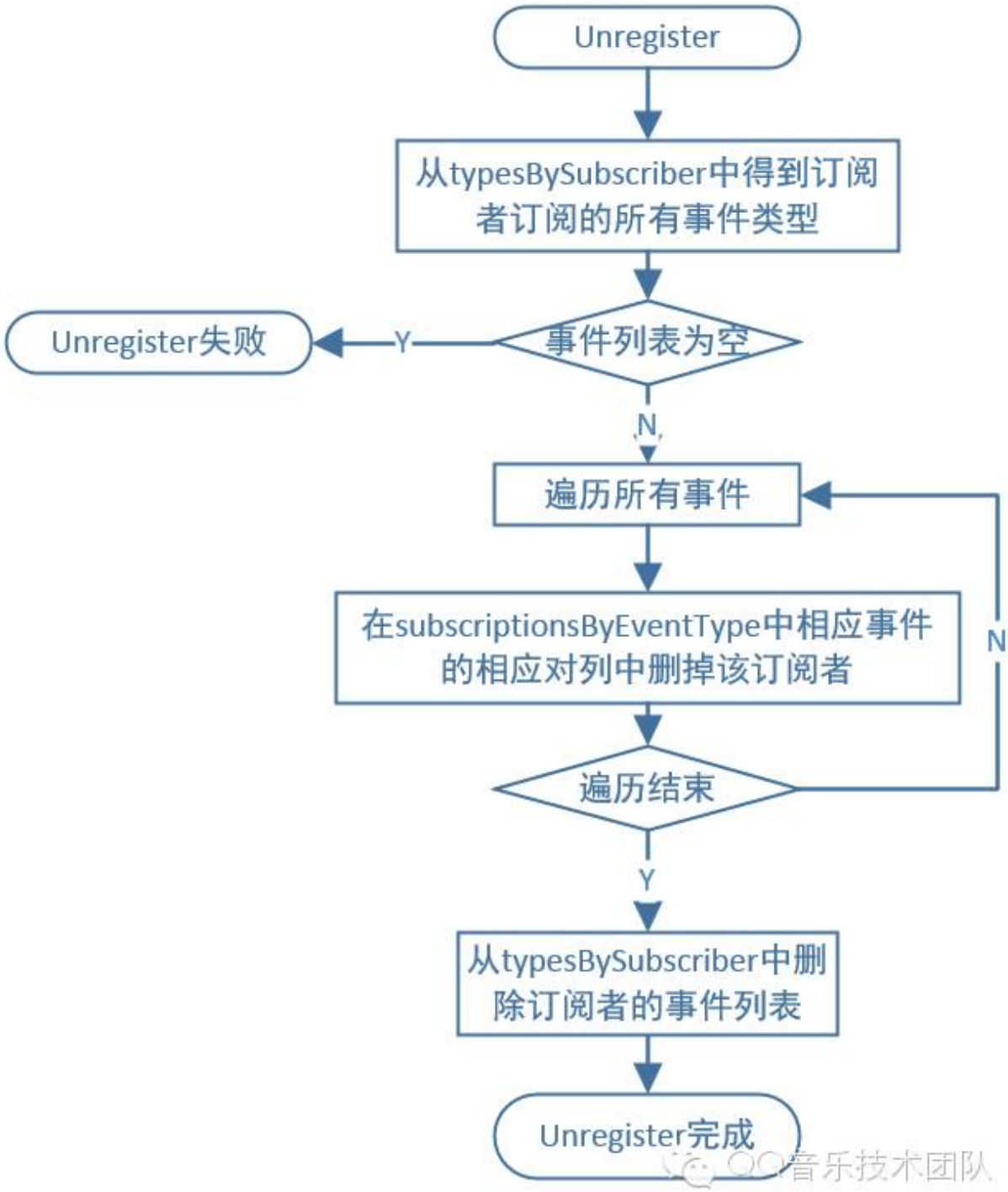
总的来说就是分析事件，得到所有监听该事件的订阅者的回调方法，并利用反射来invoke方法，实现回调：


Post
获得当前线程信息，并拿到对应的事件对列eventQueue
把事件添加到当前线程的事件队列中
事件正在分发
N
循环eventQueue中的每个事件
lookupAllEventTypes获得事件所有父类，并遍历
在subscriptionsByEventType中获得每个事件类的订阅者的回调方法
遍历回调方法。根据ThreadMode，通过不同的poster，在对应的线程中通过反射invoke该方法
遍历结束
N
Y
遍历结束
N
Y
如果没有任何订阅者，则发送NoSubscriberEvent
遍历方法结束
N
Y
Y
Post完成
QQ音乐技术团队

这里就能看到poster的调度事件功能，同时可以看到调度的单位细化成了Subscription，即每一个方法都有自己的优先级和是否接收黏性事件。在源代码中为了保证post执行不会出现死锁，等待和对同一订阅者发送相同的事件，增加了很多线程保护锁和标志位，值得我们每个开发者学习。

2.4 unregister

注销就比较简单了，把在注册时往两个数据结构中添加的订阅者信息删除即可：

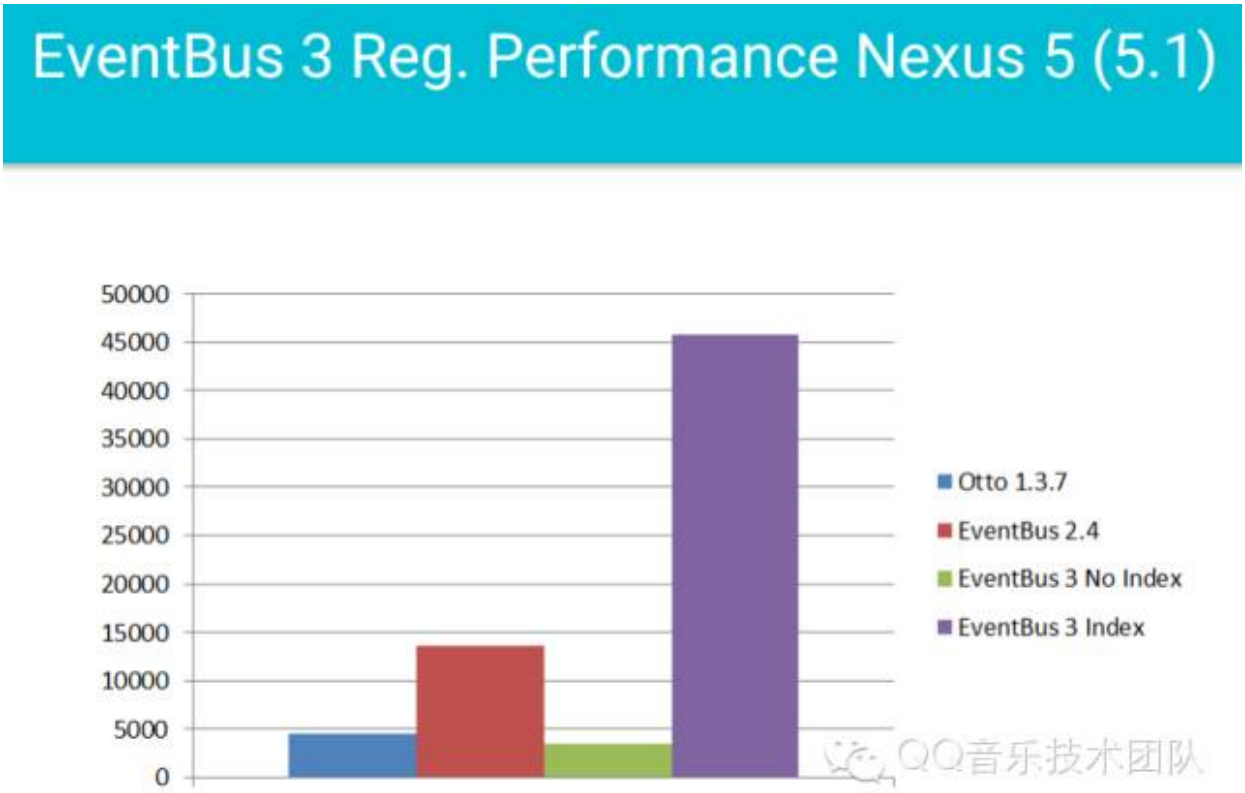


上面经常会提到黏性事件，为什么要有这个设计呢？这里举个例子：我想在登陆成功后自动播放歌曲，而登陆和监听登陆监听是同时进行的。在这个前提下，如果登陆流程走得特别快，在登陆成功后播放模块才注册了监听。此时播放模块便会错过了【登陆成功】的事件，出现“虽然登陆成功了，回调却没执行”的情况。而如果【登陆成功】这个事件是一个黏性事件的话，那么即使我后来才注册了监听（并且回调方法设置为监听黏性事件），则回调就能在注册的那一刻被执行，这个问题就能被优雅地解决，而不需要额外去定义其他标志位。

至此大家对EventBus的运行原理应该有了一定的了解，虽然看起来像是一个复杂耗时的自动机，但大部分时候事件都是一瞬间就能分发到位的，而大家关心的性能问题反而是发生在注册EventBus的时候，因为需要遍历监听者的所有方法去找到回调的方法。作者也提到运行时注解的性能在Android上并不理想，为了解决这个问题，作者才会以索引的方式去生成回调方法表（下一章会详细介绍）。而EventBus源码分析的文章早已是数不胜数，这里就不再大段大段地贴代码了，主要以类图和流程图的形式让大家直观地了解EventBus3的整体架构及核心功能的实现原理，把源码分析留到后面介绍EventBusAnnotationProcessor中再进行。大家如果想要深入学习EventBus 3的话，在本文结尾的参考文章中有很多写得很棒的源码分析。

3. 涡轮引擎——索引加速

在EventBus 3的介绍中，作者提到以前的版本为了保证性能，在遍历寻找订阅者的回调方法时使用反射而不是注解。但现在却能在使用注解的前提下，大幅度提高性能，同时作者在博客中放出了这张对比图：



可以看到在性能方面，EventBus 3由于使用了注解，比起使用反射来遍历方法的2.4版本逊色不少。但开启了索引后性能像打了鸡血一样，远远超出之前的版本。这里我们就来分析一下这个提高EventBus性能的“涡轮引擎”。（下面的源码分析为了方便阅读，添加了部分注释，并删减了部分源码，如果有疑问的话可以到官方的github上查看原版源码）

首先我们知道，索引是在初始化EventBus时通过`EventBusBuilder.addIndex(SubscriberInfoIndex index)`方法传进来的，我们就先看看这个方法：

```
public EventBusBuilder addIndex(SubscriberInfoIndex index) {
    if(subscriberInfoIndexes == null) {
        subscriberInfoIndexes = new ArrayList<>();
    }
    subscriberInfoIndexes.add(index);
    return this;
}
```

可以看到，传进来的索引信息会保存在subscriberInfoIndexes这个List中，后续会通过EventBusBuilder传到相应EventBus的SubscriberMethodFinder实例中。我们先来分析SubscriberInfoIndex这个参数：

```
public interface SubscriberInfoIndex {
    SubscriberInfo getSubscriberInfo(Class<?> subscriberClass);
}
```

可见索引只需要做一件事情——就是能拿到订阅者的信息。而实现这个接口的类如果我们没有编译过，是找不到的。这里就得看我们在一开始在配置gradle时导入的EventBusAnnotationProcessor：

```
@SupportedAnnotationTypes("org.greenrobot.eventbus.Subscribe")
@SupportedOptions(value = {"eventBusIndex", "verbose"})
public class EventBusAnnotationProcessor extends AbstractProcessor {
    /** Found subscriber methods for a class (without superclasses). 被注解表示的方
    private final ListMap<TypeElement, ExecutableElement> methodsByClass = new Li
    private final Set<TypeElement> classesToSkip = new HashSet<>(); // checkHasEr

    @Override
    public boolean process(Set<? extends TypeElement> annotations, RoundEnvironme
        Messager messager = processingEnv.getMessager();
        try {
            String index = processingEnv.getOptions().get(OPTION_EVENT_BUS_INDEX)
            if (index == null) { // 如果没有在gradle中配置apt的argument，编译就会在这
                messager.printMessage(Diagnostic.Kind.ERROR, "No option " + OPTIO
                        " passed to annotation processor");
                return false;
            }
            /** ... */
            collectSubscribers(annotations, env, messager); // 根据注解拿到所有订阅
```

```
            checkForSubscribersToSkip(messager, indexPackage); // 筛掉不符合规则的
            if (!methodsByClass.isEmpty()) {
                createInfoIndexFile(index); // 生成索引类
            }
            /** 打印错误 */
    }

    /** 下面这些方法就不再贴出具体实现了，我们了解它们的功能就行 */
    private void collectSubscribers // 遍历annotations，找出所有被注解标识的方法，以
    private boolean checkHasNoErrors // 过滤掉static，非public和参数大于1的方法
    private void checkForSubscribersToSkip // 检查methodsByClass中的各个类，是否存在
    /** 下面这三个方法会把methodsByClass中的信息写到相应的类中 */
    private void writeCreateSubscriberMethods
    private void createInfoIndexFile
    private void writeIndexLines
}
```

至此便揭开了索引生成的秘密，是在编译时apt插件通过EventBusAnnotationProcessor分析注解，并利用注解标识的相关类的信息去生成相关的类。writeCreateSubscriberMethods中调用了很多IO函数，很容易理解，这里就不贴了，我们直接看生成出来的类：

```
/** This class is generated by EventBus, do not edit. */
public class MyEventBusIndex implements SubscriberInfoIndex {
private static final Map<Class<?>, SubscriberInfo> SUBSCRIBER_INDEX;
    static {
        SUBSCRIBER_INDEX = new HashMap<Class<?>, SubscriberInfo>();
        // 每有一个订阅者类，就调用一次putIndex往索引中添加相关的信息
        putIndex(new SimpleSubscriberInfo(com.study.sangerzhong.studyapp.ui.MainA
            new SubscriberMethodInfo("onEvent", com.study.sangerzhong.studyapp.ui
            // 类中每一个被Subscribe标识的方法都在这里添加进来
        }));
    }
    // 下面的代码就是EventBusAnnotationProcessor中写死的了
    private static void putIndex(SubscriberInfo info) {
        SUBSCRIBER_INDEX.put(info.getSubscriberClass(), info);
    }

    @Override
    public SubscriberInfo getSubscriberInfo(Class<?> subscriberClass) {
        SubscriberInfo info = SUBSCRIBER_INDEX.get(subscriberClass);
        if (info != null) {
            return info;
        } else {
            return null;
        }
```

```
    }
}
```

可见，子类中hardcode了所有注册了EventBus的类中被Subscribe注解标识的方法信息，包括方法名、方法参数类型等信息。并把这些信息封装到SimpleSubscriberInfo中，我们拿到的索引其实就是以订阅者的类为Key、SimpleSubscriberInfo为value的哈希表。而这些hardcode都是在编译的时候生成的，避免了在在EventBus.register()时才去遍历查找生成，从而把在注册时需要遍历订阅者所有方法的行为，提前到在编译时完成了。

索引的生成我们已经明白了，那么它是在哪里被应用的呢？我们记得在注册时，EventBus会通过SubscriberMethodFinder来遍历注册对象的Class的所有方法，筛选出符合规则的方法，并作为订阅者在接收到事件时执行的回调，我们直接来看看`SubscriberMethodFinder.findSubscriberMethods()`这个核心方法：

```
List<SubscriberMethod> findSubscriberMethods(Class<?> subscriberClass) {
    List<SubscriberMethod> subscriberMethods = METHOD_CACHE.get(subscriberClass);
    if (subscriberMethods != null) {
        return subscriberMethods; // 先去方法缓存里找，找到直接返回
    }
    if (ignoreGeneratedIndex) { // 是否忽略设置的索引
        subscriberMethods = findUsingReflection(subscriberClass);
    } else {
        subscriberMethods = findUsingInfo(subscriberClass);
    }
    /** 把找到的方法保存到METHOD_CACHE里并返回，找不到直接抛出异常 */
}
```

可以看到其中findUsingInfo()方法就是去索引中查找订阅者的回调方法，我们戳进去看看这个方法的实现：

```
private List<SubscriberMethod> findUsingInfo(Class<?> subscriberClass) {
    // 最新版的EventBus3中，寻找方法时所需的临时变量都被封装到了FindState这个静态内部类
    FindState findState = prepareFindState(); // 到对象池中取得上下文，避免频繁创造对
    findState.initForSubscriber(subscriberClass); // 初始化寻找方法的上下文
    while (findState.clazz != null) { // 子类找完了，会继续去父类中找
        findState.subscriberInfo = getSubscriberInfo(findState); // 获得订阅者类的
        if (findState.subscriberInfo != null) { // 上一步能拿到相关信息的话，就开始找
            SubscriberMethod[] array = findState.subscriberInfo.getSubscriberMeth
            for (SubscriberMethod subscriberMethod : array) {
                if (findState.checkAdd(subscriberMethod.method, subscriberMethod.
                    // checkAdd是为了避免在父类中找到的方法是被子类重写的，此时应该保证
                    findState.subscriberMethods.add(subscriberMethod);
                }
```

```
            }
        } else { // 索引中找不到，降级成运行时通过注解和反射去找
            findUsingReflectionInSingleClass(findState);
        }
        findState.moveToSuperclass(); // 上下文切换成父类
    }
    return getMethodsAndRelease(findState); // 找完后，释放FindState进对象池，并返回
}
```

可以看到EventBus中在查找订阅者的回调方法时是能处理好继承关系的，不仅会去遍历父类，而且还会避免因为重写方法导致执行多次回调。其中需要关心的是getSubscriberInfo()是如何返回索引数据的，我们继续深入：

```
private SubscriberInfo getSubscriberInfo(FindState findState) {
    if (findState.subscriberInfo != null && findState.subscriberInfo.getSuperSubs
        SubscriberInfo superclassInfo = findState.subscriberInfo.getSuperSubscrib
        if (findState.clazz == superclassInfo.getSubscriberClass()) { // 确定是所需
            return superclassInfo;
        }
    }
    if (subscriberInfoIndexes != null) { // 从我们传进来的subscriberInfoIndexes中获
        for (SubscriberInfoIndex index : subscriberInfoIndexes) {
            SubscriberInfo info = index.getSubscriberInfo(findState.clazz);
            if (info != null) { return info; }
        }
    }
    return null;
}
```

可见就在这个方法里面，应用到了我们生成的索引，避免我们需要在findSubscriberMethods时去调用耗时的findUsingReflection方法。在实际使用中，Nexus6上一个Activity注册EventBus需要10毫秒左右，而使用了索引后能降低到3毫秒左右，效果非常明显。虽然这个索引的实现逻辑有点绕，而且还存在一些坑（比如后面讲到的混淆问题），但实现的手段非常巧妙，尤其是“把耗时的操作在编译的时候完成”和“用对象池减少创建对象的性能开销”的思想值得我们开发者借鉴。

4. 驾驶宝典——踩坑与经验

4.1 混淆问题

混淆作为版本发布必备的流程，经常会闹出很多奇奇怪怪的问题，且不方便定位，尤其

是EventBus这种依赖反射技术的库。通常情况下都会把相关的类和回调方法都keep住，但这样其实会留下被人反编译后破解的后顾之忧，所以我们的目标是keep最少的代码。

首先，因为EventBus 3弃用了反射的方式去寻找回调方法，改用注解的方式。作者的意思是在混淆时就不用再keep住相应的类和方法。但是我们在运行时，却会报`java.lang.NoSuchFieldError: No static field POSTING`。网上给出的解决办法是keep住所有eventbus相关的代码：

```
-keep class de.greenrobot.** {*;}
```

其实我们仔细分析，可以看到是因为在SubscriberMethodFinder的findUsingReflection方法中，在调用Method.getAnnotation()时获取ThreadMode这个enum失败了，所以我们只需要keep住这个enum就可以了（如下）。

```
-keep public enum org.greenrobot.eventbus.ThreadMode { public static *; }
```

这样就能正常编译通过了，但如果使用了索引加速，是不会有上面这个问题的。因为在找方法时，调用的不是findUsingReflection，而是findUsingInfo。但是使用了索引加速后，编译后却会报新的错误：`Could not find subscriber method in XXX Class. Maybe a missing ProGuard rule?`

这就很好理解了，因为生成索引GeneratedSubscriberIndex是在代码混淆之前进行的，混淆之后类名和方法名都不一样了（上面这个错误是方法无法找到），得keep住所有被Subscribe注解标注的方法：

```
-keepclassmembers class * {
    @de.greenrobot.event.Subscribe <methods>;
}
```

所以又倒退回了EventBus2.4时不能混淆onEvent开头的方法一样的处境了。所以这里就得权衡一下利弊：使用了注解不用索引加速，则只需要keep住EventBus相关的代码，现有的代码可以正常的进行混淆。而使用了索引加速的话，则需要keep住相关的方法和类。

4.2 跨进程问题

目前EventBus只支持跨线程，而不支持跨进程。如果一个app的service起到了另一个进程中，那么注册监听的模块则会收不到另一个进程的EventBus发出的事件。这里可以考

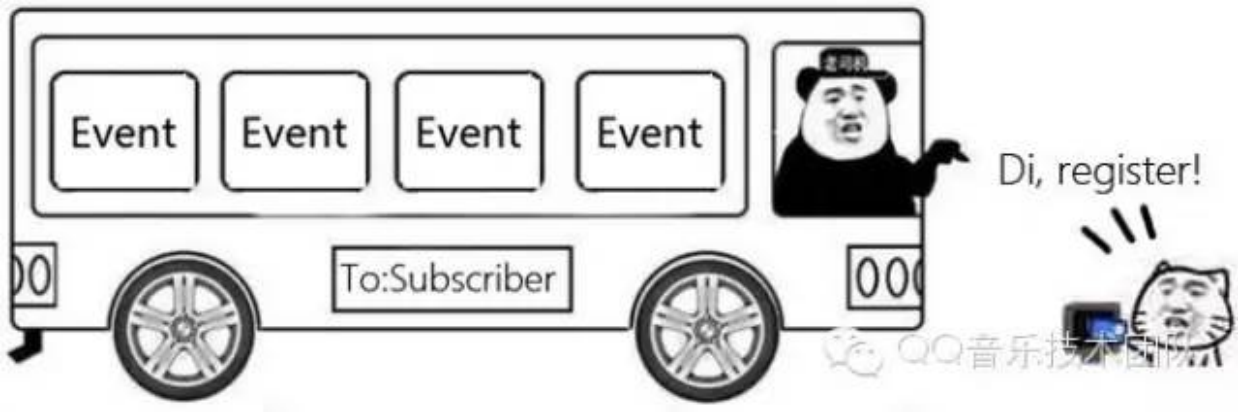
虑利用IPC做映射表，并在两个进程中各维护一个EventBus，不过这样就要自己去维护register和unregister的关系，比较繁琐，而且这种情况下通常用广播会更加方便，大家可以思考一下有没有更优的解决方案。

4.3 事件环路问题

在使用EventBus时，通常我们会把两个模块相互监听，来达到一个相互回调通信的目的。但这样一旦出现死循环，而且如果没有相应的日志信息，很难定位问题。所以在使用EventBus的模块，如果在回调上有环路，而且回调方法复杂到了一定程度的话，就要考虑把接收事件专门封装成一个子模块，同时考虑避免出现事件环路。

5. 车神之路——写在最后

当然，EventBus并不是重构代码的唯一之选。作为观察者模式的“同门师兄弟”——RxJava，作为功能更为强大的响应式编程框架，可以轻松实现EventBus的事件总线功能（RxBus）。但毕竟大型项目要接入RxJava的成本高，复杂的操作符需要开发者投入更多的时间去学习。所以想在成熟的项目中快速地重构、解耦模块，EventBus依旧是我们的不二之选。



本文总结了EventBus 3的使用方法，运行原理和一些新特性，让大家能直观地看到这个组件的优点和缺点，同时让大家在考虑是否在项目中引入EventBus时心里有个底。最后感谢Markus Junginger大神开源了如此实用的组件，以及组内同事在笔者探究EventBus原理时提供的帮助，希望大家在看完本文后都能有所收获，成为NB的Android开发老司机。

参考：

1. EventBus 3 Beta

2. Markus Junginger - EventBus 3 beta announced at droidcon
3. Skykai - EventBus 3.0 源代码分析
4. yydcdut - EventBus3.0源码解析
5. TmWork - EventBus源码解析
6. 鸿洋 - Android打造编译时注解解析框架
7. YoKey - 用RxJava实现事件总线(Event Bus)