转载请注明链接：http://blog.csdn.net/feather_wch/article/details/79042406

> 总结EventBus的知识点
> 分析EventBus源码的要点，包括：
>
> 1. EventBus的构造源码
> 2. 订阅者注册的源码
> 3. 事件发送的源码
> 4. 订阅者取消注册的源码

EventBus要点和源码解析

版本：2018/2/11-1

1、事件总线的作用

> 简化Activity、Fragment、Thread和Service之前的通信并且有更高的质量

2、EventBus作用和优缺点

> 1. 针对android优化的发布-订阅事件总线。
> 2. 开销小，相比于广播效率高(广播如果传递实体数据，需要序列化)
> 3. 将发送者和接受者解耦

3、EventBus三要素

> 1. Event事件
> 2. SubScriber：事件订阅者。EventBus3.0开始可以指定任意事件处理方法，只需要添加一个注解@Subscribe并且指定线程模型(默认为POSTING)
> 3. Publisher：事件发布者，直接调用EventBus的post(Object)方法

4、EventBus的四种线程模型

ThreadMode	作用	备注

ThreadMode	作用	备注
POSTING（默认）	哪个线程发布事件， 处理函数就在哪个线程处理	事件处理时要避免执行耗时操作， 会阻塞事件的传递， 甚至导致ANR
MAIN	事件在UI线程中处理	避免耗时操作
BACKGROUND	若事件在UI线程发布， 则事件在新线程处理； 若事件在子线程发布， 则直接在该线程处理	禁止UI操作
ASYNC	无论事件在哪发布， 事件都在新子线程中处理	禁止UI操作

5、EventBus3.0前的只能使用规定的消息处理方法(对应线程模型)

1. onEvent
2. onEventMainThread
3. onEventBackgroundThread
4. onEventAsync

6、EventBus的使用

1. 自定义一个事件类，如： `class MsgEvent`
2. 在需要订阅事件的地方注册事件： `EventBus.getDefault().register(this)`
3. 发送事件： `EventBus.getDefault().post(msgEvent)`
4. 处理事件

```
  @Subscribe
public void onEventMainThread(MsgEvent event)
{
    ...
}
//Since EventBus 3.0
@Subscribe (threadMode = ThreadMode.MAIN)
public void customEventHandler(MsgEvent event)
{
    ...
}
```

5. 取消事件订阅：`EventBus.getDefault().unregister(this)`

7、ProGuard需要加入EventBus相关的混淆规则

8、EventBus的粘性事件

> 是指发送事件后，再订阅该事件也可以接收到该事件(类似于粘性广播)

9、EventBus粘性事件的处理和发送

>

@Subscribe (threadMode = ThreadMode.MAIN, sticky = true)
public void customStickyEventHandler(MsgEvent event)
{
...
}

> 发送：
>
> ```java

EventBus.getDefault().postSticky(new MsgEvent("粘性事件"));

>

EventBus的构造

10、EventBus的构造方法

```
    //1. 单例模式，双重检查
```

```
public static EventBus getDefault() {
    if(defaultInstance == null) {
        Class var0 = EventBus.class;
        synchronized(EventBus.class) {
            if(defaultInstance == null) {
                defaultInstance = new EventBus();
            }
        }
    }
    return defaultInstance;
}
//2. 建造者模式
public EventBus() {
    this(DEFAULT_BUILDER);
}
private static final EventBusBuilder DEFAULT_BUILDER = new EventBusBuilder();
```

> 1. getDefault采用单例模式，使用双重检查(DLC)
> 2. EventBus的构造方法里面，通过默认EvenBusBuilder进行构造(建造者模式)
> 3. 我们可以通过构造一个EvenBusBuilder对EventBus进行配置

订阅者注册

11、EventBus的注册源码要点：

```
//EventBus的订阅者注册
public void register(Object subscriber) {
    Class subscriberClass = subscriber.getClass();
    //1. 获取订阅者所有的需要订阅的方法(SubscriberMethod中保存了订阅方法的Method对象、线程模式、
    List subscriberMethods = this.subscriberMethodFinder.findSubscriberMethods(subscriberCl
    synchronized(this) {
        Iterator var5 = subscriberMethods.iterator();

     //2. 遍历所有需要订阅的方法，并进行注册
        while(var5.hasNext()) {
            SubscriberMethod subscriberMethod = (SubscriberMethod)var5.next();
            this.subscribe(subscriber, subscriberMethod);
        }

    }
}
```

12、EventBus注册的findSubscriberMethods源码要点

```
/**
 * 获取订阅者的所有订阅方法(onEventMainThread等等)
 */
List<SubscriberMethod> findSubscriberMethods(Class<?> subscriberClass) {
    //1. 查找是否有缓存的订阅方法的集合
    List subscriberMethods = (List)METHOD_CACHE.get(subscriberClass);
    //2. 找到方法立即返回
    if(subscriberMethods != null) {
        return subscriberMethods;
    } else {
        //3. 选择采取何种方法查询订阅方法的集合(ignoreGeneratedIndex指是否忽略注解器生成的MyEv
        if(this.ignoreGeneratedIndex) {
            subscriberMethods = this.findUsingReflection(subscriberClass);
        } else {
            //4. 默认通过单例模式获取默认的EventBus对象(ignoreGeneratedIndex=false)
            subscriberMethods = this.findUsingInfo(subscriberClass);
        }

        //5.获取订阅方法集合后，放入缓存中
        if(subscriberMethods.isEmpty()) {
            throw new EventBusException("Subscriber " + subscriberClass + " and its super c
        } else {
            //放入缓存
            METHOD_CACHE.put(subscriberClass, subscriberMethods);
            return subscriberMethods;
        }
    }
}
```

13、EventBus注册的findUsingInfo源码要点

```
private List<SubscriberMethod> findUsingInfo(Class<?> subscriberClass) {
    SubscriberMethodFinder.FindState findState = this.prepareFindState();
    findState.initForSubscriber(subscriberClass);

    for(; findState.clazz != null; findState.moveToSuperclass()) {
        //1. 获取订阅者信息(默认没有忽略注解器生成的MyEventBusIndex，下面会进行判断)
        findState.subscriberInfo = this.getSubscriberInfo(findState);
        //2. 判断是否配置了MyEventBusIndex，若配置了Info不为空
        if(findState.subscriberInfo != null) {
            //4. 通过订阅者信息获得订阅方法的相关信息
            SubscriberMethod[] array = findState.subscriberInfo.getSubscriberMethods();
            SubscriberMethod[] var4 = array;
            int var5 = array.length;

            for(int var6 = 0; var6 < var5; ++var6) {
                SubscriberMethod subscriberMethod = var4[var6];
                if(findState.checkAdd(subscriberMethod.method, subscriberMethod.eventType)) {
                    findState.subscriberMethods.add(subscriberMethod);
                }
            }
        } else {
            //3. 没有配置MyEventBusIndex，会将订阅方法保存到findState中
            this.findUsingReflectionInSingleClass(findState);
        }
    }
    //5. 对findState进行回收处理并且返回订阅方法的List集合
    return this.getMethodsAndRelease(findState);
}
```

14、EventBus注册的findUsingReflectionInSingleClass源码要点

```
private void findUsingReflectionInSingleClass(SubscriberMethodFinder.FindState findState) {
    Method[] methods;
    try {
        //1. 通过反射来获得订阅者中的所有方法
        methods = findState.clazz.getDeclaredMethods();
    } catch (Throwable var12) {
        methods = findState.clazz.getMethods();
        findState.skipSuperClasses = true;
    }
    ...
    //2. 根据方法的类型、参数和注解找到订阅方法
    ...
    //3. 将找到的订阅方法的相关信息保存到findState中
    if(findState.checkAdd(method, eventType)) {
        ThreadMode threadMode = methodName1.threadMode();
        findState.subscriberMethods.add(new SubscriberMethod(method, eventType, threadMode, methodName1
    }
    ...
}
```

15、EventBus注册的subscribe(订阅者注册)源码要点

```
/**============================================================
 *  订阅者的注册
 *  @位于:  EventBus的register()
 *  @本质思想:
 *    1. 将订阅者对象添加到[订阅者对象集合]中(根据订阅方法的优先级)-进行注册
 *        [订阅者对象集合]需要根据[事件类型]添加到[按事件类型分类的总订阅者对象集合]中(subscrip
 *    2. 将事件类型添加到[事件类型集合]中
 *        [事件类型集合]需要根据[订阅者]添加到[按订阅者分类的总事件类型集合]中(typesBySubscribe
 *    3. 对粘性事件进行额外处理
 *============================================================*/
private void subscribe(Object subscriber, SubscriberMethod subscriberMethod) {
    Class eventType = subscriberMethod.eventType;
    //1. 根据订阅者(subscriber)和订阅方法(subscriberMethod)创建一个订阅对象(Subscription)
    Subscription newSubscription = new Subscription(subscriber, subscriberMethod);
    //2. 根据事件类型(EventType)获取订阅对象集合
    CopyOnWriteArrayList subscriptions = (CopyOnWriteArrayList)this.subscriptionsByEventTyp
    //3. 订阅对象集合为空，则重新创建集合，并将subscriptions根据事件类型eventType保存到subscri
    if(subscriptions == null) {
        subscriptions = new CopyOnWriteArrayList();
        this.subscriptionsByEventType.put(eventType, subscriptions);
    } else if(subscriptions.contains(newSubscription)) {
        //4. 判断订阅者是否已经被注册
        throw new EventBusException("Subscriber " + subscriber.getClass() + " already regis
    }

    int size = subscriptions.size();

    //5. 将订阅者对象添加到订阅者对象集合中
    for(int subscribedEvents = 0; subscribedEvents <= size; ++subscribedEvents) {
        if(subscribedEvents == size || subscriberMethod.priority > ((Subscription)subscript
            // 根据订阅方法的优先级进行注册
            subscriptions.add(subscribedEvents, newSubscription);
            break;
        }
    }

    //6. 通过subscriber获取事件类型集合(subscribedEvents)
    Object subscribedEvents = (List)this.typesBySubscriber.get(subscriber);
    if(subscribedEvents == null) {
        subscribedEvents = new ArrayList();
        //7. 事件类型集合为null，则新建，并根据订阅者subscriber将subscribedEvents存储到typesBy
        this.typesBySubscriber.put(subscriber, subscribedEvents);
    }

    //8. 将eventType添加到subscribedEvent中
    ((List)subscribedEvents).add(eventType);

    //9. 如果是粘性事件，从stickyEvents事件保存队列中取出该事件类型的事件发送给当前订阅者
    if(subscriberMethod.sticky) {
        if(this.eventInheritance) {
            Set stickyEvent = this.stickyEvents.entrySet();
            Iterator var9 = stickyEvent.iterator();

            while(var9.hasNext()) {
```

```
                Entry entry = (Entry)var9.next();
                Class candidateEventType = (Class)entry.getKey();
                if(eventType.isAssignableFrom(candidateEventType)) {
                    Object stickyEvent1 = entry.getValue();
                    this.checkPostStickyEventToSubscription(newSubscription, stickyEvent1);
                }
            }
        } else {
            Object var14 = this.stickyEvents.get(eventType);
            this.checkPostStickyEventToSubscription(newSubscription, var14);
        }
    }

}
```

事件的发送

16、EventBus的post方法的源码要点

```
public void post(Object event) {
    //1. PostingThreadState保存事件队列和线程状态信息
    EventBus.PostingThreadState postingState = (EventBus.PostingThreadState)this.currentPos
    //2. 获取事件队列
    List eventQueue = postingState.eventQueue;
    //3. 将当前事件插入事件队列
    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 {
            //4. 处理事件队列中所有事件，并移除该事件
            while(!eventQueue.isEmpty()) {
                this.postSingleEvent(eventQueue.remove(0), postingState);
            }
        } finally {
            postingState.isPosting = false;
            postingState.isMainThread = false;
        }
    }
}
```

17、EventBus事件发送的postSingleEvent源码要点

```
    private void postSingleEvent(Object event, EventBus.PostingThreadState postingState) throws
        Class eventClass = event.getClass();
        boolean subscriptionFound = false;
        //1. 表示是否向上查找事件的父类，默认为true(可以通过EventBuilder配置)
        if(this.eventInheritance) {
            //2. 找到所有父类事件，保存在List中
            List eventTypes = lookupAllEventTypes(eventClass);
            int countTypes = eventTypes.size();

            for(int h = 0; h < countTypes; ++h) {
                Class clazz = (Class)eventTypes.get(h);
                //3. 通过postSingleEventForEventType对事件逐一处理
                subscriptionFound |= this.postSingleEventForEventType(event, postingState, claz
            }
        } else {
            //4. 没有查找父类事件，直接处理该事件
            subscriptionFound = this.postSingleEventForEventType(event, postingState, eventClas
        }

        if(!subscriptionFound) {
            if(this.logNoSubscriberMessages) {
                Log.d(TAG, "No subscribers registered for event " + eventClass);
            }
            if(this.sendNoSubscriberEvent && eventClass != NoSubscriberEvent.class && eventClas
                this.post(new NoSubscriberEvent(this, event));
            }
        }
    }
```

18、EventBus事件发送的postSingleEventForEventType源码要点

```
/**==============================
 *  按照事件类型post事件
 *==============================*/
private boolean postSingleEventForEventType(Object event, EventBus.PostingThreadState posti
    CopyOnWriteArrayList subscriptions;
    synchronized(this) {
        //1. 从[按事件类型分类的总订阅对象集合]中获取订阅对象集合(与该事件对应)
        subscriptions = (CopyOnWriteArrayList)this.subscriptionsByEventType.get(eventClass)
    }

    if(subscriptions != null && !subscriptions.isEmpty()) {
        Iterator var5 = subscriptions.iterator();

        //2. 遍历订阅对象集合，分别处理
        while(var5.hasNext()) {
            Subscription subscription = (Subscription)var5.next();
            //3. postingState获得事件和订阅对象
            postingState.event = event;
            postingState.subscription = subscription;
            boolean aborted = false;

            try {
                //4. 对事件进行处理
                this.postToSubscription(subscription, event, postingState.isMainThread);
                aborted = postingState.canceled;
            } finally {
                postingState.event = null;
                postingState.subscription = null;
                postingState.canceled = false;
            }

            if(aborted) {
                break;
            }
        }

        return true;
    } else {
        return false;
    }
}
```

19、EventBus事件发送的postToSubscription源码要点

```
    /**==================================
     *        发送给订阅对象
     *  @要点：
     *     1. invokeSubscriber()-通过反射直接运行订阅的方法
     *     2. mainThreadPoster.enqueue()-是将订阅事件添加到主线程队列中
     *           类型为HandlerPoster，继承自Handler，通过Handler将订阅方法却环岛主线程执行。
     *     3. backgroundPoster.enqueue()-新开子线程处理
     *     4. asyncPoster.enqueue()-新开子线程处理
     *==================================*/
    private void postToSubscription(Subscription subscription, Object event, boolean isMainThre
        //1. 取出订阅方法的线程模式[subscription.subscriberMethod.threadMode]
        switch(subscription.subscriberMethod.threadMode) {
            //2. 根据模式分别处理
            case POSTING:
                //3. 与事件发布处在同一线程
                this.invokeSubscriber(subscription, event);
                break;
            case MAIN:
                //4. 在UI线程
                if(isMainThread) {
                    this.invokeSubscriber(subscription, event);
                } else {
                    this.mainThreadPoster.enqueue(subscription, event);
                }
                break;
            case BACKGROUND:
                //5. 事件发布处若在UI线程，则新开子线程处理。若在子线程，则在该线程处理
                if(isMainThread) {
                    this.backgroundPoster.enqueue(subscription, event);
                } else {
                    this.invokeSubscriber(subscription, event);
                }
                break;
            case ASYNC:
                //6. 无论是否在UI线程，均新开子线程处理
                this.asyncPoster.enqueue(subscription, event);
                break;
            default:
                throw new IllegalStateException("Unknown thread mode: " + subscription.subscrib
        }

    }
```

订阅者取消注册

20、订阅者取消注册的unregister源码要点

```
public synchronized void unregister(Object subscriber) {
    //1. 通过订阅者(subscriber)从[按订阅者分类的 事件类型集合的总集合中]中获取相应[事件类型集合
    List subscribedTypes = (List)this.typesBySubscriber.get(subscriber);
    if(subscribedTypes != null) {
        Iterator var3 = subscribedTypes.iterator();

        while(var3.hasNext()) {
            Class eventType = (Class)var3.next();
            //2. 通过[事件类型]在[订阅对象集合]中移除该订阅者的订阅对象
            this.unsubscribeByEventType(subscriber, eventType);
        }
        //3. [事件类型集合的总集合]中移除与订阅者相关的事件类型集合
        this.typesBySubscriber.remove(subscriber);
    } else {
        Log.w(TAG, "Subscriber to unregister was not registered before: " + subscriber.get(
    }
}
```

21、订阅者取消注册的unsubscribeByEventType源码要点

```
private void unsubscribeByEventType(Object subscriber, Class<?> eventType) {
    //1. 通过事件类型获取订阅对象集合
    List subscriptions = (List)this.subscriptionsByEventType.get(eventType);
    if(subscriptions != null) {
        int size = subscriptions.size();

        for(int i = 0; i < size; ++i) {
            //2. 移除与订阅者(subscriber)相关的订阅对象
            Subscription subscription = (Subscription)subscriptions.get(i);
            if(subscription.subscriber == subscriber) {
                subscription.active = false;
                subscriptions.remove(i);
                --i;
                --size;
            }
        }
    }
}
```