EventBus解析

- 使用简介
  1. 注册

     ```
     EventBus.getDefault().register(this);
     ```

  2. 响应事件订阅方法

     ```
     @Subscribe(threadMode = ThreadMode.BACKGROUND, sticky = true, priority = 100)
     public void test(String str) {

     }
     ```

  3. 发送事件

     ```
     EventBus.getDefault().post("str");
     EventBus.getDefault().postSticky("str");
     ```

  4. 解除注册

     ```
     EventBus.getDefault().unregister(this);
     ```

- 源码解析
  - 通过EventBus.getDefault()获取到EventBus对象

  ```
  // 双重检查的单例模式，保证得到同一个实例
  public static EventBus getDefault() {
          if (defaultInstance == null) {
              synchronized (EventBus.class) {
                  if (defaultInstance == null) {
                      defaultInstance = new EventBus();
                  }
              }
          }
          return defaultInstance;
  }
  ```

  - 注册：
    1. 获得订阅者的class对象

       ```
       public void register(Object subscriber) {
               // 1.
               Class<?> subscriberClass = subscriber.getClass();
               ...
       }
       ```

2. 获得这个订阅者都订阅了什么事件（包括ThreadMode、eventType、priority、sticky等信息）

```
public void register(Object subscriber) {
        Class<?> subscriberClass = subscriber.getClass();
        // 2.
        List<SubscriberMethod> subscriberMethods =
subscriberMethodFinder.findSubscriberMethods(subscriberClass);
        ...
}
```

3. 根据第二步得到的 该订阅者订阅事件 的响应函数，循环每个响应函数

```
public void register(Object subscriber) {
        Class<?> subscriberClass = subscriber.getClass();
        List<SubscriberMethod> subscriberMethods =
subscriberMethodFinder.findSubscriberMethods(subscriberClass);
        // 3.
        synchronized (this) {
            for (SubscriberMethod subscriberMethod : subscriberMethods) {
                subscribe(subscriber, subscriberMethod);
            }
        }
}
```

- 循环体中的subscribe():

  1. 获得订阅的事件的事件类型（就这订阅的是个啥事件）

```
private void subscribe(Object subscriber, SubscriberMethod
subscriberMethod) {
        //1.
        Class<?> eventType = subscriberMethod.eventType;

        ...
}
```

  2. 根据这个事件的事件类型，得到该事件的所有订阅者信息，根据优先级把当前订阅者的信息插入到订阅者队列中

```
private void subscribe(Object subscriber, SubscriberMethod
subscriberMethod) {
        Class<?> eventType = subscriberMethod.eventType;
        // 2.
        // 该订阅者对象
        Subscription newSubscription = new Subscription(subscriber,
subscriberMethod);
        // 该事件的所有订阅者信息
        CopyOnWriteArrayList<Subscription> subscriptions =
subscriptionsByEventType.get(eventType);
        if (subscriptions == null) {
            // 该事件之前没有订阅者，new一个订阅者队列
```

```
                subscriptions = new CopyOnWriteArrayList<>();
                subscriptionsByEventType.put(eventType, subscriptions);
            } else {
                // 该事件之前已经有订阅者了。检查当前订阅者是否已经在订阅者队列中
                if (subscriptions.contains(newSubscription)) {
                    throw new EventBusException("Subscriber " +
subscriber.getClass() + " already registered to event "
                            + eventType);
                }
            }
            // 根据优先级把当前订阅者的信息插入到订阅者队列中
            int size = subscriptions.size();
            for (int i = 0; i <= size; i++) {
                if (i == size || subscriberMethod.priority >
subscriptions.get(i).subscriberMethod.priority) {
                    subscriptions.add(i, newSubscription);
                    break;
                }
            }

            ...
    }
```

3. 得到当前订阅者对象订阅的所有事件，将订阅者和他的所有事件保存到typesBySubscriber里，用于后续取消订阅

```
    private void subscribe(Object subscriber, SubscriberMethod
subscriberMethod) {
            Class<?> eventType = subscriberMethod.eventType;
            Subscription newSubscription = new Subscription(subscriber,
subscriberMethod);
            CopyOnWriteArrayList<Subscription> subscriptions =
subscriptionsByEventType.get(eventType);
            if (subscriptions == null) {
                subscriptions = new CopyOnWriteArrayList<>();
                subscriptionsByEventType.put(eventType, subscriptions);
            } else {
                if (subscriptions.contains(newSubscription)) {
                    throw new EventBusException("Subscriber " +
subscriber.getClass() + " already registered to event "
                            + eventType);
                }
            }
            int size = subscriptions.size();
            for (int i = 0; i <= size; i++) {
                if (i == size || subscriberMethod.priority >
subscriptions.get(i).subscriberMethod.priority) {
                    subscriptions.add(i, newSubscription);
                    break;
                }
            }

            // 3.
```

```
        // 得到当前订阅者对象订阅的所有事件
        List<Class<?>> subscribedEvents =
typesBySubscriber.get(subscriber);
        // 将当前事件保存到typesBySubscriber里
        if (subscribedEvents == null) {
            subscribedEvents = new ArrayList<>();
            typesBySubscriber.put(subscriber, subscribedEvents);
        }
        subscribedEvents.add(eventType);

        ...
}
```

4. 判断是否接受粘性事件，如果接受，就取出该事件post给当前订阅者

```
private void subscribe(Object subscriber, SubscriberMethod
subscriberMethod) {
        Class<?> eventType = subscriberMethod.eventType;
        Subscription newSubscription = new Subscription(subscriber,
subscriberMethod);
        CopyOnWriteArrayList<Subscription> subscriptions =
subscriptionsByEventType.get(eventType);
        if (subscriptions == null) {
            subscriptions = new CopyOnWriteArrayList<>();
            subscriptionsByEventType.put(eventType, subscriptions);
        } else {
            if (subscriptions.contains(newSubscription)) {
                throw new EventBusException("Subscriber " +
subscriber.getClass() + " already registered to event "
                        + eventType);
            }
        }
        int size = subscriptions.size();
        for (int i = 0; i <= size; i++) {
            if (i == size || subscriberMethod.priority >
subscriptions.get(i).subscriberMethod.priority) {
                subscriptions.add(i, newSubscription);
                break;
            }
        }
        List<Class<?>> subscribedEvents =
typesBySubscriber.get(subscriber);
        if (subscribedEvents == null) {
            subscribedEvents = new ArrayList<>();
            typesBySubscriber.put(subscriber, subscribedEvents);
        }
        subscribedEvents.add(eventType);

        // 4.
        // 如果接收sticky事件,立即分发sticky事件
        if (subscriberMethod.sticky) {
            if (eventInheritance) {
                Set<Map.Entry<Class<?>, Object>> entries =
stickyEvents.entrySet();
                for (Map.Entry<Class<?>, Object> entry : entries) {
                    Class<?> candidateEventType = entry.getKey();
                    if
(eventType.isAssignableFrom(candidateEventType)) {
                        Object stickyEvent = entry.getValue();

 checkPostStickyEventToSubscription(newSubscription, stickyEvent);
                    }
                }
            } else {
                Object stickyEvent = stickyEvents.get(eventType);
                checkPostStickyEventToSubscription(newSubscription,
stickyEvent);
            }
        }
    }
}
```

- 发送事件
    1. 将要发送的事件add入当前线程的事件队列中
    2. 检查当前线程是否在分发事件，在分发的话就没他什么事儿了。要是没在分发，那就要分发啦。
        - 检查没在分发后，循环体中调用postSingleEvent()去分发事件队列的每个事件
        - 在postSingleEvent()中：

          检查该事件是否有继承父类->

          如果有继承父类，得到当前事件的所有父类和接口，循环地调用postSingleEventForEventType()去分发他们（所有的父类和接口）->

          如果他们为空（没有任何订阅者）,发送NoSubscriberEvent（）->

          如果没有继承父类，直接调用postSingleEventForEventType()去分发当前这一个事件。
        - 在postSingleEventForEventType()中：获取所有订阅了这个事件的订阅者列表，在postToSubscription()里去分发
        - 在postToSubscription()通过不同的threadMode在不同的线程里调用invoke()订阅者的方法
- 解除注册
    1. 取出这个订阅者订阅的事件类型

```
public synchronized void unregister(Object subscriber) {
        // 1.
        List<Class<?>> subscribedTypes =
typesBySubscriber.get(subscriber);
        ...
}
```

    2. 分别解除每个事件

```
public synchronized void unregister(Object subscriber) {
        List<Class<?>> subscribedTypes =
typesBySubscriber.get(subscriber);
        // 2.
        if (subscribedTypes != null) {
            for (Class<?> eventType : subscribedTypes) {
                unsubscribeByEventType(subscriber, eventType);
            }
            ...
        }
        ...
}
```

- 循环体中的unsubscribeByEventType():

1. 取出这个事件的订阅者列表

```
private void unsubscribeByEventType(Object subscriber, Class<?>
eventType) {
        // 1.
        List<Subscription> subscriptions =
subscriptionsByEventType.get(eventType);
        ...
}
```

2. 在订阅者列表里找到该订阅者（要解除注册的那个订阅者），将它移出队列

```
private void unsubscribeByEventType(Object subscriber, Class<?>
eventType) {
        List<Subscription> subscriptions =
subscriptionsByEventType.get(eventType);
        // 2.
        if (subscriptions != null) {
            int size = subscriptions.size();
            for (int i = 0; i < size; i++) {
                Subscription subscription = subscriptions.get(i);
                if (subscription.subscriber == subscriber) {
                    subscription.active = false;
                    subscriptions.remove(i);
                    i--;
                    size--;
                }
            }
        }
    }
```

3. 从typesBySubscriber中移除当前订阅者（包括他订阅的所有事件）

```
public synchronized void unregister(Object subscriber) {
        List<Class<?>> subscribedTypes =
typesBySubscriber.get(subscriber);
        if (subscribedTypes != null) {
            for (Class<?> eventType : subscribedTypes) {
                unsubscribeByEventType(subscriber, eventType);
            }
            // 3.
            typesBySubscriber.remove(subscriber);
        } else {
            Log.w(TAG, "Subscriber to unregister was not registered
before: " + subscriber.getClass());
        }
}
```