Guava-eventbus 学习报告

林涵越 2017K8009929006

第一章：从红包提醒到公众号

互联网改变了人们的生活习惯，其中最明显的一点就是：过年过节，我们的第一件事总是拿出手机，等待着每个 QQ 群主发出的那条醒目的消息：有红包，然后开始抢红包大战。也许是抢红包的紧张刺激，让我们总是习惯性忽略了一件事：QQ 是怎么做到同时通知所有人有红包的呢？

像这样的情况，在微信公众平台上更为常见。当微信公众号发送了一条推文时，微信公众平台将会把这个推文发送给所有订阅这个公众号的人。此时，大家就都会看到新的通知，看到新的推送。说的这么简单，但问题是微信公众平台是怎么让所有订阅者都收到消息的呢？

我们将这两个情况概括一下：一个对象将一个事务，传达给所有关注这个事务的对象（去进行相应的行动）。在生活中，有许多这样的情况。而在程序的设计中，这样的情况也有所涉及。既然如此，能不能用一个模块，来实现这个步骤呢？于是，我们就有了 eventbus。

事件总线(Eventbus)是 Google.Guava 提供的消息发布-订阅类库，用于管理事件的注册和分发。消息通知负责人通过 EventBus 去注册/注销观察者，最后由消息通知负责人给观察者发布消息。它允许组件间的发布-订阅式通信，而不需要进行显式注册。此外，guava 事件总线专注于进程内的数据的通信，并不适用于进程间的通信。

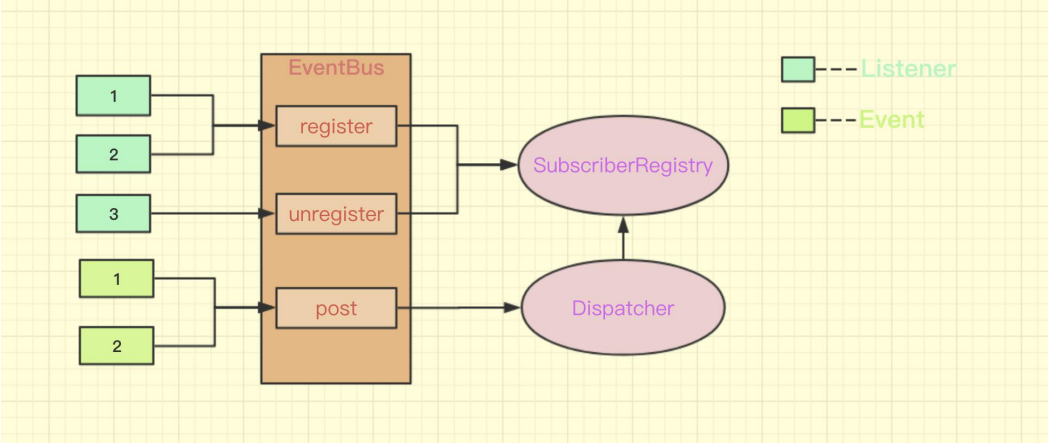

图 1 Event Bus 的工作机制 （图源：https://www.cnblogs.com/wagne/p/10188091.html）

对上图的这些英文词语做一个解释：

词语：	解释：
监听者(listener)	这是一个希望接受和处理事件的对象
事件(event)	一个事件对象，它要把它的内容传递给各个监听者
订阅 / 注册(register)	向事件总线注册监听者，使他们能够接受所对应的事件
解除订阅 / 注销(unregister)	订阅操作的反向操作，让监听者不再监听消息
发布(post)	将事件发布到事件总线中，总线将通知监听者
派发(dispatcher)	将发布的事件发送给订阅了它的监听者
订阅关系类(SubscriberRegistry)	这是一个维护了事件类型和对其感兴趣的事件的订阅者列表的类

事件总线主要包括以下几个组件：发布者、订阅者、事件总线、事件通道、事件监听器。它的工作流程为：订阅者在事件总线中注册要监听的事件，将这些订阅方法和订阅对象存储在 map 中。当发布者在特定的通道上发布一个事件时，事件总线根据事件的参数类型和 tag 找到对应的订阅者对象,最后执行订阅者对象中的方法。这些订阅方法会执行在发布者指定的线程模型中，事件总线会通知订阅者从这些特定通道上获取事件消息。当然，在 guava-eventbus 中，事件的发布者也可以是订阅者。

在 Guava 提供的源码中，EventBus 包括了如下文件：

文件名	文件大小	内容
AllowConcurrentEvent.java	2KB	用于标志一个事件订阅者 Method 类是线程安

		全的接口，需要和subscribe一起使用
AsyncEventBus.java	3KB	允许异步分发事件的事件总线
DeadEvent.java	3KB	死事件类说明和构造
Dispatcher.java	8KB	事件分发者类说明和构造
EventBus.java	10KB	事件总线类说明和构造
Package-info.java	12KB	事件总线包整体说明
Subscribe.java	2KB	用于标记一个 Method 类为事件订阅者的接口
Subscriber.java	5KB	事件订阅者类说明和构造
SubscriberExcptionContext.java	3KB	用于储存 Subscriber 报出例外的 Context 类
SubscriberExcptionHandler.java	1KB	处理 Subscriber 报出例外的接口
SubscriberRegistry.java	10KB	事件订阅者的注册

Event Bus 主要需要实现的功能有：

1.订阅者在事件总线中注册/订阅事件；

2.发布者发布事件到事件总线中的特定通道，事件总线完成事件分发。

EventBus 中部分类的关系图（简略）大致如下：

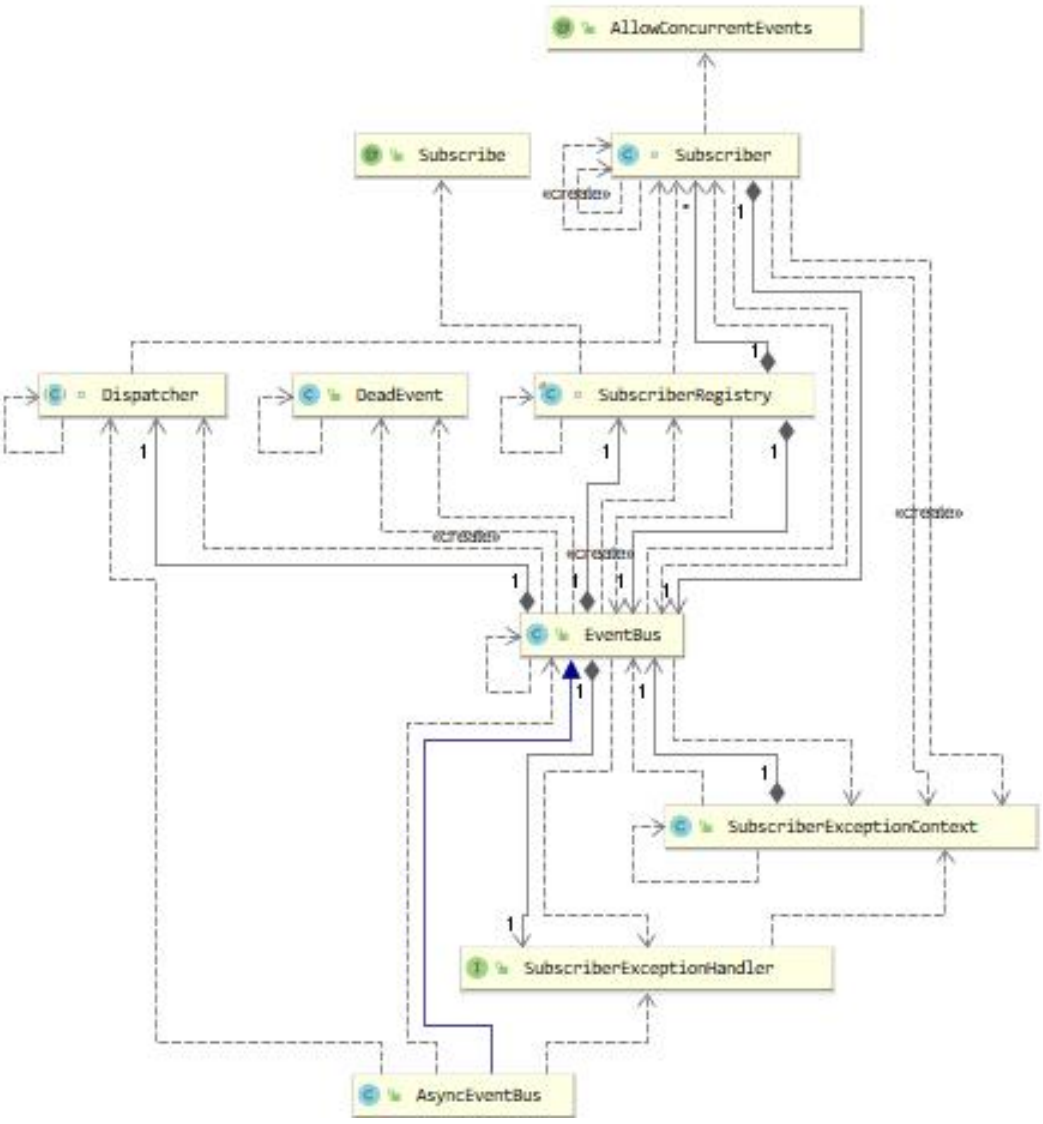

图 2 EventBus 中部分类的关系

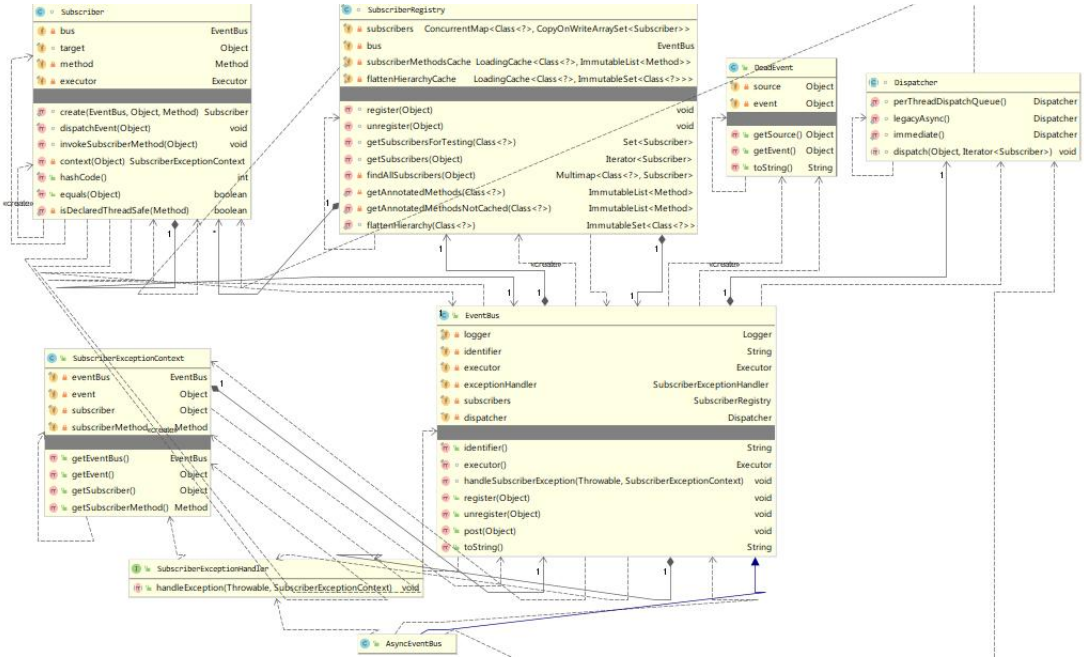

图 3 EventBus UML 类图

第二章：源码分析

（一）事件总线 EventBus

要了解事件总线的工作原理，那肯定要从事件总线 Eventbus 类开始了解起，然后往下分析它的各个部件。

事件总线的成员变量有：

```
  private static final Logger logger =
Logger.getLogger(EventBus.class.getName());  //记录事件总线的日志
  private final String identifier;             //事件总线的标识
  private final Executor executor;             //用于通知订阅者
  private final SubscriberExceptionHandler exceptionHandler;     //例
外处理
  private final SubscriberRegistry subscribers = new
SubscriberRegistry(this);
  private final Dispatcher dispatcher;    //事件派发器
```

在新建一个 EventBus 后一共有 4 种初始化方法：传入缺省参数；传入 identifier；传入 exceptionhadler；传入 identifier / executor / exceptionhandler / dispatcher。前面三种方法修饰符都是 public，只有最后一种修饰符是 default。方法注释符的不同有两个原因：一方面，使用者在使用事件总线的时候，并不需要关心事件是如何分发的，因此这一部分并不需要开放修改器方法给使用者；另一方面，这限制了使用者在使用事件总线时候只能修改 identifier 和 exceptionhandler，因此不会修改到事件注册和事件分发的重要代码，使他们只能供开发者修改，使得代码更安全。

EventBus 中有另外 3 个重要的 public 方法：register/unregister/post，其中 register 和 unregister 都是直接调用了 subscriber 中的对应方法，而 post 方法的代码如下：

```
  public void post(Object event) {
    Iterator<Subscriber> eventSubscribers =
                                        subscribers.getSubscribers(even
                                        t);           //获得该事件的所有订
                                        阅者
    if (eventSubscribers.hasNext()) {
      dispatcher.dispatch(event, eventSubscribers);      //根据派发器进
行派发
    } else if (!(event instanceof DeadEvent)) {
      // the event had no subscribers and was not itself a DeadEvent
      post(new DeadEvent(this, event));
    }
  }
```

（二）事件派发 Dispatcher

Dispatcher 类是一个抽象类，这是因为在 Dispatcher 中包含了抽象方法 dispatch：

```
  /** Dispatches the given {@code event} to the given {@code subscribers}.
*/
  abstract void dispatch(Object event, Iterator<Subscriber>
subscribers);
```

根据注释可以知道，dispatch 这个抽象方法的使用场景是对给定的事件和订阅者进行特定的一对一事件分发。之所以把这个方法定义为抽象方法，是因为在不同的子类中对于该方法的实现不相同。在子类 perThreadQueuedDispatcher 中，dispatch 实现方法如下：

```
void dispatch(Object event, Iterator<Subscriber> subscribers) {
  checkNotNull(event);
  checkNotNull(subscribers);
  Queue<Event> queueForThread = queue.get();
  queueForThread.offer(new Event(event, subscribers));

  if (!dispatching.get()) {
    dispatching.set(true);
    try {
      Event nextEvent;
      while ((nextEvent = queueForThread.poll()) != null) {
        while (nextEvent.subscribers.hasNext()) {
          nextEvent.subscribers.next().dispatchEvent(nextEvent.event);
        }
      }
    } finally {
      dispatching.remove();
      queue.remove();
    }
  }
}
```

LegacyAsyncDispatcher 中，dispatch 方法实现如下：

```
    @Override
    void dispatch(Object event, Iterator<Subscriber> subscribers) {
      checkNotNull(event);
      while (subscribers.hasNext()) {
        queue.add(new EventWithSubscriber(event, subscribers.next()));
      }

      EventWithSubscriber e;
      while ((e = queue.poll()) != null) {
        e.subscriber.dispatchEvent(e.event);
```

```
  }
}
```

而在另一子类 ImmidiateDispatcher 中，dispatch 方法实现如下：

```
@Override
void dispatch(Object event, Iterator<Subscriber> subscribers) {
  checkNotNull(event);
  while (subscribers.hasNext()) {
    subscribers.next().dispatchEvent(event);
  }
   }
```

在 Dispacher.java 文件中一共有 3 中 Dispatcher 子类：PerThreadQueueDispatcher / LegacyAsyncDispatcher / ImmidiateDispatcher。

前两个子类都需要队列存储事件，其中 PerThreadQueueDispatcher 是一个线程对应一个队列，这是 eventbus 默认的一种派发方式。而 LegacyAsyncDispatcher 则是多个线程共用一个全局队列，它是 eventbus 的子类 Asynceventbus 的派发方式。发布事件时，事件会被存储到队列中，dispatcher 按照队列中事件的顺序进行事件分发。相比于普通的派发，它在多线程下是一种较为均衡的解决方案。如 A 线程，B 线程都请求派发，它不会先派发 A 再派发 B，而是 AB 混在一起派发。ImmidiateDispatcher 则不使用队列，当事件发布的时候立即进行分发，这是个同步派发。

（三）事件注册者 Subscriber

Subscriber 类是一个默认类，成员变量有：

```
/** The event bus this subscriber belongs to. */
 @Weak private EventBus bus;

 /** The object with the subscriber method. */
 @VisibleForTesting final Object target;
```

```
    /** Subscriber method. */
    private final Method method;

    /** Executor to use for dispatching events to this
  subscriber. */
    private final Executor executor;
```

其中 executor 使用的是 bus 中的 executor，用于接收来自 EventBus 的事件。

在 SubscriberRegistry 中 Subscriber 的创建并不是直接使用默认的创建方式 new Subscriber(parameters)，而是需要调用 Create 方法，代码如下：

```
  /** Creates a {@code Subscriber} for {@code method} on {@code listener}.
*/
  static Subscriber create(EventBus bus, Object listener, Method method)
{
    return isDeclaredThreadSafe(method)
        ? new Subscriber(bus, listener, method)
        : new SynchronizedSubscriber(bus, listener, method);
  }
```

Create 和 isDeclaredThreadSafe 这两个方法都是静态方法，原因是在使用这两个方法创建 subscriber 时，subscriber 类很有可能没有实例化，而静态方法在没有实例化的时候也可以使用。

如果该线程被标记为安全的（使用 AllowConcurrentEvents 方法），则实例化一个新的 Subscriber，否则实例化一个 SynchronizedSubscriber。两者的区别在于 invokeSubscriberMethod 方法，SynchronizedSubscriber 会增加 synchronized 关键字，确保同一时刻只有一个进程使用该方法，代码如下：

```
    @Override
    void invokeSubscriberMethod(Object event) throws
```

```
InvocationTargetException {
      synchronized (this) {
        super.invokeSubscriberMethod(event);
      }
    }
```

在实际使用的过程中，可能会出现同一 object 多次对同一 method 注册 subscriber 的情况，因此 subscriber 类中有一个 equals 方法用于判断两个 object 是否相同。

```
  @Override
  public final boolean equals(@Nullable Object obj) {
    if (obj instanceof Subscriber) {
      Subscriber that = (Subscriber) obj;
      // Use == so that different equal instances will still receive
events.
      // We only guard against the case that the same object is registered
      // multiple times
      return target == that.target && method.equals(that.method);
    }
    return false;
      }
```

Subscriber 的例外处理不在 Subscriber 中，而是在 EventBus 中，因为在通常情况下默认 Subscriber 是不会产生例外的，如果产生例外，需要反馈给 EventBus 进行处理并写入日志。这并不是一个常见的例外处理方法，主要的目的是为了找出问题所在。

（四）SubscriberRegistry

SubscriberRegistry 是一个 final 类。它有两个成员变量：Subscribers 和 bus。

Subscribers 使用 ConcurrentMap 存储已经注册的 class-Subscriber 对，而 bus 是对该 SubscriberRegistry 所在 EventBus 的引用。

```
  /*All registered subscribers, indexed by event type.
```

```
   * <p>The {@link CopyOnWriteArraySet} values make it easy and
relatively lightweight to get an
   * immutable snapshot of all current subscribers to an event without
any locking.
   */
  private final ConcurrentMap<Class<?>,
CopyOnWriteArraySet<Subscriber>> subscribers =
Maps.newConcurrentMap();
  /** The event bus this registry belongs to. */
  @Weak private final EventBus bus;
```

SubscriberRegistry中最为重要的2个方法是register和unregister。

(1)Register方法

Register方法首先会使用FindAllSubscribers方法。FindAllSubscribers会访问SubscriberRegistry类中的***subscriberMethodsCache***，它用于存储一个类到该类及其所有父类中有@Subscribe注解的方法（其实就是event）的mapping，遍历该缓存并且为每个已经注解的方法创造一个Subscriber，最后返回所有的subscriber。这里创建Subscriber 使用的是create方法。具体代码如下：

```
  /**
   * Returns all subscribers for the given listener grouped by the type
of event they subscribe to.
   */
  private Multimap<Class<?>, Subscriber> findAllSubscribers(Object
listener) {
    Multimap<Class<?>, Subscriber> methodsInListener =
HashMultimap.create();
    Class<?> clazz = listener.getClass();
    for (Method method : getAnnotatedMethods(clazz)) {
      Class<?>[] parameterTypes = method.getParameterTypes();
      Class<?> eventType = parameterTypes[0];
      methodsInListener.put(eventType, Subscriber.create(bus, listener,
method));
    }
    return methodsInListener;
  }
```

获得所有的 subscriber 后还需要把这些 subscriber 加入 CopyonWriteArraySet 中，因为 subscribers 存储的是 event 到 CopyonWriteArraySet 的映射。如果 subscribers 中没有该 event，会新建一个映射。具体代码如下：

```
  /** Registers all subscriber methods on the given listener object. */
  void register(Object listener) {
    Multimap<Class<?>, Subscriber> listenerMethods =
findAllSubscribers(listener);

    for (Entry<Class<?>, Collection<Subscriber>> entry :
listenerMethods.asMap().entrySet()) {
      Class<?> eventType = entry.getKey();
      Collection<Subscriber> eventMethodsInListener =
entry.getValue();

      CopyOnWriteArraySet<Subscriber> eventSubscribers =
subscribers.get(eventType);

      if (eventSubscribers == null) {
        CopyOnWriteArraySet<Subscriber> newSet = new
CopyOnWriteArraySet<>();
        eventSubscribers =

MoreObjects.firstNonNull(subscribers.putIfAbsent(eventType, newSet),
newSet);
      }

      eventSubscribers.addAll(eventMethodsInListener);
    }
  }
```

(2)unregister 方法：

Unregister 方法也是首先使用 FindAllSubscribers 找到 listener 的所有 subscriber，然后找到对应的 CopyonWriteSet。如果该集合已经是空集或者 removeall 操作不成功（这意味着至少有 1 个 subscriber 被删除了）则会报错。

第三章：设计模式分析

EventBus 作为一个处理一对多的工具，自然一定会涉及到行为模式中的 observer 观察者模式。只不过相比于传统的 observer 方法，eventbus 采用的是基于 java 注解的“隐式接口”的方法，以此来绑定订阅者。

除此之外，Eventbus 的 Dispatcher 的设计中运用了 Strategy 策略模式。它提供了一个抽象方法接口 dispatch，不同的策略重写他们自己的 dispatch 方法来实现他们的策略，而 eventbus 则根据 dispatcher 的不同，执行不同的 dispatch 方法。

另外，在 ImmediateDispatcher 中，还用到了 Singleton 单例模式。这是一个同步的派发器，两个派发器同时存在必然导致派发的混乱。因此它内部用一个属性来维护它的唯一实例。而使用 getInstance 来返回单例。由于根据策略选择的不同，我们不一定需要一个单例，因此它使用了静态内部类的方法，在使用了 get 方法的时候才被创建，节省内存。

第四章：总结与展望

很高兴这学期能学习面向对象程序设计这门课程。对于从未接触过这种编程方法的我来说，一开始是一个挺大的挑战。刚开始看源码时两眼发懵，一头雾水，后来在老师的讲解下终于逐渐有了一些概念，再通过不断的查找资料，这才把 eventbus 给读完。虽然说这只是一个 C 级的小项目，但它的设计相当巧妙，而且它的应用范围，也绝不仅仅是 C 级。

感谢王伟老师和唐震助教这学期对我们的指导和帮助，感谢刘骐鸣同学让我选择了这门课程，并给予了我帮助，感谢吴双学姐在设计模式的分析中给我的帮助。

参考文献链接：

1. https://www.cnblogs.com/moonandstar08/p/5651793.html

2. https://blog.csdn.net/feelwing1314/article/details/80335164

3. http://ifeve.com/google-guava-eventbus/

4. https://blog.csdn.net/u012070360/article/details/60141106#

5. https://www.cnblogs.com/wagne/p/10188091.html