Guava 事件总线(EventBus) 分析报告

吴双 2016K8009937003

一、概述

事件总线(Eventbus)是Guava中的进程内事件分发机制，用于管理事件的注册和分发。它允许组件间的发布-订阅式通信，而不需要进行显式注册。事件总线并不适用与进程间的通信。

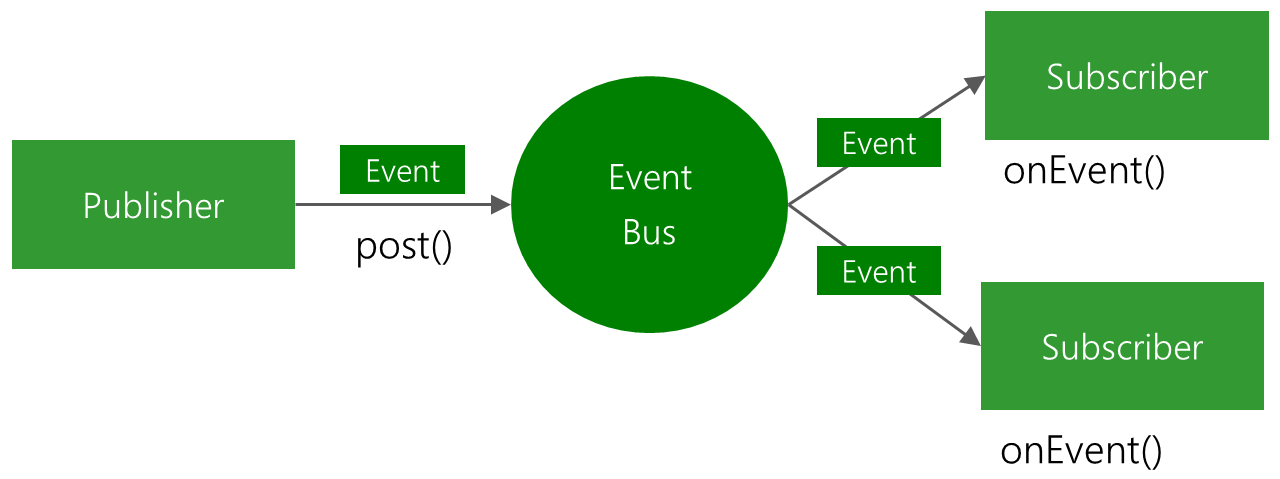


图1 Event Bus 的工作机制 （图源：<http://greenrobot.org/eventbus/>）

在EventBus中使用的描述事件分发的特定术语如下：

|  |  |
| --- | --- |
| 事件 | 可以向事件总线发布的对象 |
| 订阅 | 向事件总线注册*监听者*以接受事件的行为 |
| 监听者 | 提供一个*处理方法*，希望接受和处理事件的对象 |
| 处理方法 | 监听者提供的公共方法，事件总线使用该方法向监听者发送事件；该方法应该用Subscribe注解 |
| 发布消息 | 通过事件总线向所有匹配的监听者提供事件 |

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。其工作流程为：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。值得一提的是发布者同时也可以是订阅者。

在Guava提供的源码中，EventBus文件夹包括如下文件：

|  |  |  |
| --- | --- | --- |
| 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。

除去说明事件总线包的内容和功能的Package-info.java文件，剩下4个不小于5KB的文件是Dipatcher.java/EventBus.java/Subscriber.java/SubscriberRegistery.java，这也从侧面反映出事件的订阅和分发是事件总线的实现中的重要部分。

EventBus中某些类的关系大致如下：

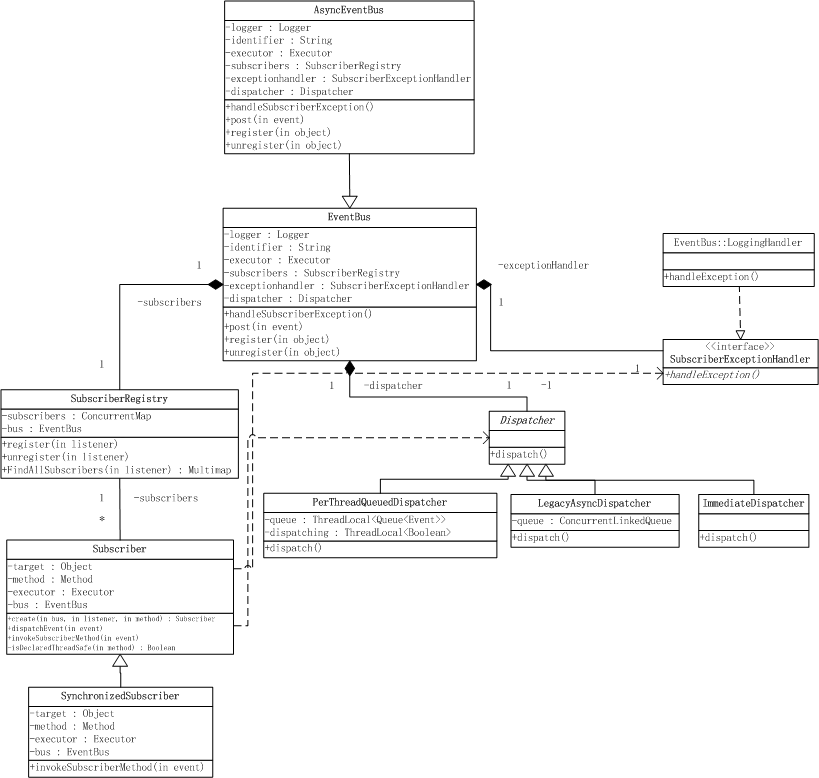


图2 EventBus中部分类的关系

二、源码分析

（一）事件总线EventBus

选择事件总线作为第一个分析对象，一是因为网上对于Eventbus.java的分析较多，进行源码分析可能比较容易入手；二是通过对于EventBus.java的分析，可以比较清楚的知道完成一次事件订阅到接受到订阅事件的大概流程。

事件总线的成员变量有：

**private** **static** **final** Logger ***logger*** = Logger.*getLogger*(EventBus.**class**.getName()); //用于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; //事件分发

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，不会修改事件注册和事件分发的重要代码，而开发者可以进行修改，保障了安全性。

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(event); //获得该事件的所有订阅者

**if** (eventSubscribers.hasNext()) {

dispatcher.dispatch(event, eventSubscribers);

} **else** **if** (!(event **instanceof** DeadEvent)) {

// the event had no subscribers and was not itself a DeadEvent

post(**new** DeadEvent(**this**, event));

}

}

（二）事件分发者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，而LegacyAsyncDispatcher是多个线程共用一个全局队列。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。ImmidiateDispatcher则不使用队列，当事件发布的时候立即进行分发。

（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。

三、设计模式分析

EventBus中运用的设计模式有单实例模式，比如EventBus类中的例外处理类logger，以及Dispatcher的子类ImmidiateDispatcher。EventBus中还运用了策略设计模式，具体体现在EventBus的post方法中使用了dispatcher的dispactch方法，但是该方法在不同的子类中有不同的实现，根据不同的需要，EventBus中的dispatcher配置为不同的Dispatcher子类。

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#](https://blog.csdn.net/u012070360/article/details/60141106)