

57 | 观察者模式（下）：如何实现一个异步非阻塞的EventBus框架？

2020-03-13 王争

设计模式之美

[进入课程 >](#)

观察者模式（下）

讲述：冯永吉

时长 11:54 大小 10.91M



上一节课中，我们学习了观察者模式的原理、实现、应用场景，重点介绍了不同应用场景下，几种不同的实现方式，包括：同步阻塞、异步非阻塞、进程内、进程间的实现方式。

同步阻塞是最经典的实现方式，主要是为了代码解耦；异步非阻塞除了能实现代码解耦之外，还能提高代码的执行效率；进程间的观察者模式解耦更加彻底，一般是基于消息队列来实现，用来实现不同进程间的被观察者和观察者之间的交互。



今天，我们聚焦于异步非阻塞的观察者模式，带你实现一个类似 Google Guava EventBus 的通用框架。等你学完本节课之后，你会发现，实现一个框架也并非一件难事。

话不多说，让我们正式开始今天的学习吧！

异步非阻塞观察者模式的简易实现

上一节课中，我们讲到，对于异步非阻塞观察者模式，如果只是实现一个简易版本，不考虑任何通用性、复用性，实际上是非常容易的。

我们有两种实现方式。其中一种是：在每个 `handleRegSuccess()` 函数中创建一个新的线程执行代码逻辑；另一种是：在 `UserController` 的 `register()` 函数中使用线程池来执行每个观察者的 `handleRegSuccess()` 函数。两种实现方式的具体代码如下所示：

 复制代码

```
1 // 第一种实现方式，其他类代码不变，就没有再重复罗列
2 public class RegPromotionObserver implements RegObserver {
3     private PromotionService promotionService; // 依赖注入
4
5     @Override
6     public void handleRegSuccess(long userId) {
7         Thread thread = new Thread(new Runnable() {
8             @Override
9             public void run() {
10                 promotionService.issueNewUserExperienceCash(userId);
11             }
12         });
13         thread.start();
14     }
15 }
16
17 // 第二种实现方式，其他类代码不变，就没有再重复罗列
18 public class UserController {
19     private UserService userService; // 依赖注入
20     private List<RegObserver> regObservers = new ArrayList<>();
21     private Executor executor;
22
23     public UserController(Executor executor) {
24         this.executor = executor;
25     }
26
27     public void setRegObservers(List<RegObserver> observers) {
28         regObservers.addAll(observers);
29     }
30
31     public Long register(String telephone, String password) {
32         //省略输入参数的校验代码
33         //省略userService.register()异常的try-catch代码
34         long userId = userService.register(telephone, password);
```

```
35     for (RegObserver observer : regObservers) {
36         executor.execute(new Runnable() {
37             @Override
38             public void run() {
39                 observer.handleRegSuccess(userId);
40             }
41         });
42     }
43
44     return userId;
45 }
46 }
47
```

对于第一种实现方式，频繁地创建和销毁线程比较耗时，并且并发线程数无法控制，创建过多的线程会导致堆栈溢出。第二种实现方式，尽管利用了线程池解决了第一种实现方式的问题，但线程池、异步执行逻辑都耦合在了 `register()` 函数中，增加了这部分业务代码的维护成本。

如果我们的需求更加极端一点，需要在同步阻塞和异步非阻塞之间灵活切换，那就要不停地修改 `UserController` 的代码。除此之外，如果在项目中，不止一个业务模块需要用到异步非阻塞观察者模式，那这样的代码实现也无法做到复用。

我们知道，框架的作用有：隐藏实现细节，降低开发难度，做到代码复用，解耦业务与非业务代码，让程序员聚焦业务开发。针对异步非阻塞观察者模式，我们也可以将它抽象成框架来达到这样的效果，而这个框架就是我们这节课要讲的 `EventBus`。

EventBus 框架功能需求介绍

`EventBus` 翻译为“事件总线”，它提供了实现观察者模式的骨架代码。我们可以基于此框架，非常容易地在自己的业务场景中实现观察者模式，不需要从零开始开发。其中，Google Guava `EventBus` 就是一个比较著名的 `EventBus` 框架，它不仅仅支持异步非阻塞模式，同时也支持同步阻塞模式

现在，我们就通过例子来看一下，Guava `EventBus` 具有哪些功能。还是上节课那个用户注册的例子，我们用 Guava `EventBus` 重新实现一下，代码如下所示：

```
1 public class UserController {
2
```

 复制代码

```

3  private UserService userService; // 依赖注入
4
5  private EventBus eventBus;
6  private static final int DEFAULT_EVENTBUS_THREAD_POOL_SIZE = 20;
7
8  public UserController() {
9      //eventBus = new EventBus(); // 同步阻塞模式
10     eventBus = new AsyncEventBus(Executors.newFixedThreadPool(DEFAULT_EVENTBUS.
11 }
12
13 public void setRegObservers(List<Object> observers) {
14     for (Object observer : observers) {
15         eventBus.register(observer);
16     }
17 }
18
19 public Long register(String telephone, String password) {
20     //省略输入参数的校验代码
21     //省略userService.register()异常的try-catch代码
22     long userId = userService.register(telephone, password);
23
24     eventBus.post(userId);
25
26     return userId;
27 }
28 }
29
30 public class RegPromotionObserver {
31     private PromotionService promotionService; // 依赖注入
32
33     @Subscribe
34     public void handleRegSuccess(long userId) {
35         promotionService.issueNewUserExperienceCash(userId);
36     }
37 }
38
39 public class RegNotificationObserver {
40     private NotificationService notificationService;
41
42     @Subscribe
43     public void handleRegSuccess(long userId) {
44         notificationService.sendInboxMessage(userId, "...");
45     }
46 }

```

利用 EventBus 框架实现的观察者模式，跟从零开始编写的观察者模式相比，从大的流程上来说，实现思路大致一样，都需要定义 Observer，并且通过 register() 函数注册


Observer，也都需要通过调用某个函数（比如，EventBus 中的 post() 函数）来给 Observer 发送消息（在 EventBus 中消息被称作事件 event）。

但在实现细节方面，它们又有些区别。基于 EventBus，我们不需要定义 Observer 接口，任意类型的对象都可以注册到 EventBus 中，通过 @Subscribe 注解来标明类中哪个函数可以接收被观察者发送的消息。

接下来，我们详细地讲一下，Guava EventBus 的几个主要的类和函数。

EventBus、AsyncEventBus


Guava EventBus 对外暴露的所有可调用接口，都封装在 EventBus 类中。其中，EventBus 实现了同步阻塞的观察者模式，AsyncEventBus 继承自 EventBus，提供了异步非阻塞的观察者模式。具体使用方式如下所示：

 复制代码

```
1 EventBus eventBus = new EventBus(); // 同步阻塞模式
2 EventBus eventBus = new AsyncEventBus(Executors.newFixedThreadPool(8)); // 异步
```

register() 函数

EventBus 类提供了 register() 函数用来注册观察者。具体的函数定义如下所示。它可以接受任何类型（Object）的观察者。而在经典的观察者模式的实现中，register() 函数必须接受实现了同一 Observer 接口的类对象。

 复制代码

```
1 public void register(Object object);
```

unregister() 函数

相对于 register() 函数，unregister() 函数用来从 EventBus 中删除某个观察者。我就不多解释了，具体的函数定义如下所示：

[📄 复制代码](#)

```
1 public void unregister(Object object);
```

post() 函数

EventBus 类提供了 post() 函数，用来给观察者发送消息。具体的函数定义如下所示：

[📄 复制代码](#)

```
1 public void post(Object event);
```

跟经典的观察者模式的不同之处在于，当我们调用 post() 函数发送消息的时候，并非把消息发送给所有的观察者，而是发送给可匹配的观察者。所谓可匹配指的是，能接收的消息类型是发送消息（post 函数定义中的 event）类型的父类。我举个例子来解释一下。

比如，AObserver 能接收的消息类型是 XMsg，BObserver 能接收的消息类型是 YMsg，CObserver 能接收的消息类型是 ZMsg。其中，XMsg 是 YMsg 的父类。当我们如下发送消息的时候，相应能接收到消息的可匹配观察者如下所示：

[📄 复制代码](#)

```
1 XMsg xMsg = new XMsg();
2 YMsg yMsg = new YMsg();
3 ZMsg zMsg = new ZMsg();
4 post(xMsg); => AObserver接收到消息
5 post(yMsg); => AObserver、BObserver接收到消息
6 post(zMsg); => CObserver接收到消息
```

你可能会问，每个 Observer 能接收的消息类型是在哪里定义的呢？我们来看下 Guava EventBus 最特别的一个地方，那就是 @Subscribe 注解。

@Subscribe 注解

EventBus 通过 @Subscribe 注解来标明，某个函数能接收哪种类型的消息。具体的使用代码如下所示。在 DObserver 类中，我们通过 @Subscribe 注解了两个函数 f1()、f2()。

```

1 public DObserver {
2     //...省略其他属性和方法...
3
4     @Subscribe
5     public void f1(PMsg event) { //... }
6
7     @Subscribe
8     public void f2(QMsg event) { //... }
9 }

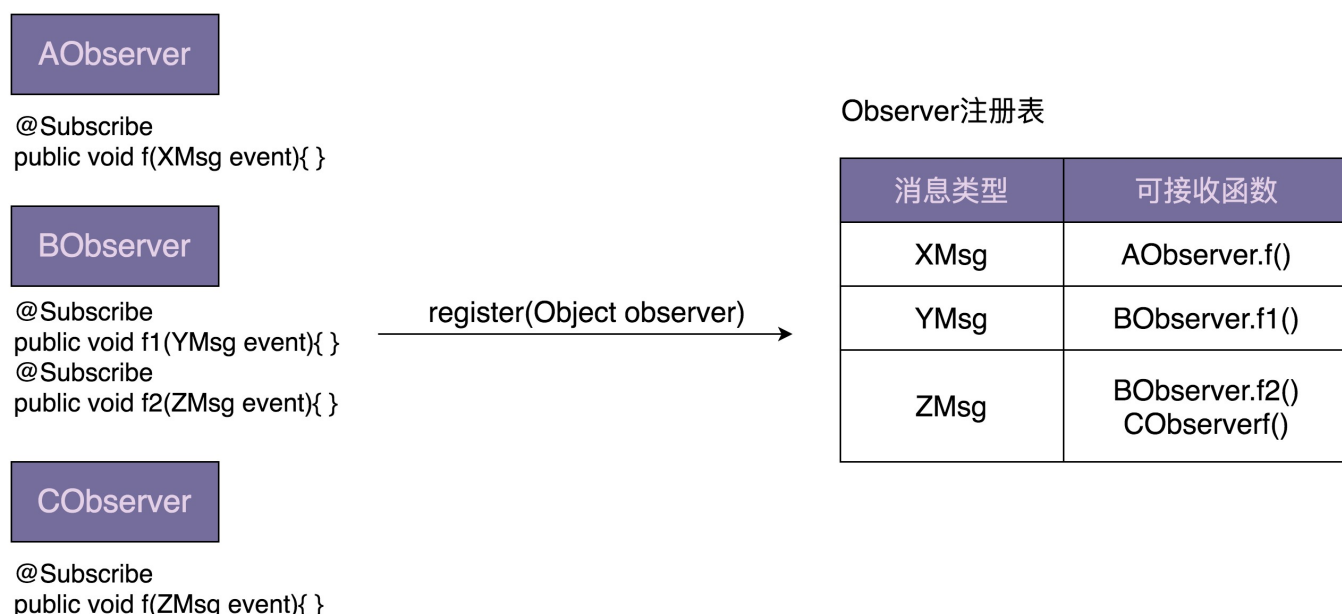
```

当通过 register() 函数将 DObserver 类对象注册到 EventBus 的时候，EventBus 会根据 @Subscribe 注解找到 f1() 和 f2()，并且将两个函数能接收的消息类型记录下来（PMsg->f1，QMsg->f2）。当我们通过 post() 函数发送消息（比如 QMsg 消息）的时候，EventBus 会通过之前的记录（QMsg->f2），调用相应的函数（f2）。

手把手实现一个 EventBus 框架

Guava EventBus 的功能我们已经讲清楚了，总体上来说，还是比较简单的。接下来，我们就重复造轮子，“山寨”一个 EventBus 出来。

我们重点来看，EventBus 中两个核心函数 register() 和 post() 的实现原理。弄懂了它们，基本上就弄懂了整个 EventBus 框架。下面两张图是这两个函数的实现原理图。



Observer注册表



从图中我们可以看出，最关键的一个数据结构是 Observer 注册表，记录了消息类型和可接收消息函数的对应关系。当调用 `register()` 函数注册观察者的时候，EventBus 通过解析 `@Subscribe` 注解，生成 Observer 注册表。当调用 `post()` 函数发送消息的时候，EventBus 通过注册表找到相应的可接收消息的函数，然后通过 Java 的反射语法来动态地创建对象、执行函数。对于同步阻塞模式，EventBus 在一个线程内依次执行相应的函数。对于异步非阻塞模式，EventBus 通过一个线程池来执行相应的函数。

弄懂了原理，实现起来就简单多了。整个小框架的代码实现包括 5 个类：EventBus、AsyncEventBus、Subscribe、ObserverAction、ObserverRegistry。接下来，我们依次来看下这 5 个类。

1.Subscribe

Subscribe 是一个注解，用于标明观察者中的哪个函数可以接收消息。

[复制代码](#)

```
1 @Retention(RetentionPolicy.RUNTIME)
2 @Target(ElementType.METHOD)
3 @Beta
4 public @interface Subscribe {}
```

2.ObserverAction

ObserverAction 类用来表示 `@Subscribe` 注解的方法，其中，`target` 表示观察者类，`method` 表示方法。它主要用在 ObserverRegistry 观察者注册表中。


```

1 public class ObserverAction {
2     private Object target;
3     private Method method;
4
5     public ObserverAction(Object target, Method method) {
6         this.target = Preconditions.checkNotNull(target);
7         this.method = method;
8         this.method.setAccessible(true);
9     }
10
11     public void execute(Object event) { // event是method方法的参数
12         try {
13             method.invoke(target, event);
14         } catch (InvocationTargetException | IllegalAccessException e) {
15             e.printStackTrace();
16         }
17     }
18 }

```

3.ObserverRegistry

ObserverRegistry 类就是前面讲到的 Observer 注册表，是最复杂的一个类，框架中几乎所有的核心逻辑都在这个类中。这个类大量使用了 Java 的反射语法，不过代码整体来说都不难理解，其中，一个比较有技巧的地方是 CopyOnWriteArraySet 的使用。

CopyOnWriteArraySet，顾名思义，在写入数据的时候，会创建一个新的 set，并且将原始数据 clone 到新的 set 中，在新的 set 中写入数据完成之后，再用新的 set 替换老的 set。这样就能保证在写入数据的时候，不影响数据的读取操作，以此来解决读写并发问题。除此之外，CopyOnWriteSet 还通过加锁的方式，避免了并发写冲突。具体的作用你可以去查看一下 CopyOnWriteSet 类的源码，一目了然。

```

1 public class ObserverRegistry {
2     private ConcurrentMap<Class<?>, CopyOnWriteArraySet<ObserverAction>> registry;
3
4     public void register(Object observer) {
5         Map<Class<?>, Collection<ObserverAction>> observerActions = findAllObserver
6         for (Map.Entry<Class<?>, Collection<ObserverAction>> entry : observerAction
7             Class<?> eventType = entry.getKey();
8             Collection<ObserverAction> eventActions = entry.getValue();
9             CopyOnWriteArraySet<ObserverAction> registeredEventActions = registry.get
10             if (registeredEventActions == null) {
11                 registry.putIfAbsent(eventType, new CopyOnWriteArraySet<>());
12                 registeredEventActions = registry.get(eventType);

```


```

13     }
14     registeredEventActions.addAll(eventActions);
15 }
16 }
17
18 public List<ObserverAction> getMatchedObserverActions(Object event) {
19     List<ObserverAction> matchedObservers = new ArrayList<>();
20     Class<?> postedEventType = event.getClass();
21     for (Map.Entry<Class<?>, CopyOnWriteArraySet<ObserverAction>> entry : regi
22         Class<?> eventType = entry.getKey();
23         Collection<ObserverAction> eventActions = entry.getValue();
24         if (postedEventType.isAssignableFrom(eventType)) {
25             matchedObservers.addAll(eventActions);
26         }
27     }
28     return matchedObservers;
29 }
30
31 private Map<Class<?>, Collection<ObserverAction>> findAllObserverActions(Object
32     Map<Class<?>, Collection<ObserverAction>> observerActions = new HashMap<>();
33     Class<?> clazz = observer.getClass();
34     for (Method method : getAnnotatedMethods(clazz)) {
35         Class<?>[] parameterTypes = method.getParameterTypes();
36         Class<?> eventType = parameterTypes[0];
37         if (!observerActions.containsKey(eventType)) {
38             observerActions.put(eventType, new ArrayList<>());
39         }
40         observerActions.get(eventType).add(new ObserverAction(observer, method))
41     }
42     return observerActions;
43 }
44
45 private List<Method> getAnnotatedMethods(Class<?> clazz) {
46     List<Method> annotatedMethods = new ArrayList<>();
47     for (Method method : clazz.getDeclaredMethods()) {
48         if (method.isAnnotationPresent(Subscribe.class)) {
49             Class<?>[] parameterTypes = method.getParameterTypes();
50             Preconditions.checkArgument(parameterTypes.length == 1,
51                 "Method %s has @Subscribe annotation but has %s parameters."
52                 + "Subscriber methods must have exactly 1 parameter.",
53                 method, parameterTypes.length);
54             annotatedMethods.add(method);
55         }
56     }
57     return annotatedMethods;
58 }
59 }

```

4.EventBus

EventBus 实现的是阻塞同步的观察者模式。看代码你可能会有些疑问，这明明就用到了线程池 Executor 啊。实际上，MoreExecutors.directExecutor() 是 Google Guava 提供的工具类，看似是多线程，实际上是单线程。之所以要这么实现，主要还是为了跟 AsyncEventBus 统一代码逻辑，做到代码复用。

 复制代码

```
1 public class EventBus {
2     private Executor executor;
3     private ObserverRegistry registry = new ObserverRegistry();
4
5     public EventBus() {
6         this(MoreExecutors.directExecutor());
7     }
8
9     protected EventBus(Executor executor) {
10         this.executor = executor;
11     }
12
13     public void register(Object object) {
14         registry.register(object);
15     }
16
17     public void post(Object event) {
18         List<ObserverAction> observerActions = registry.getMatchedObserverActions(event);
19         for (ObserverAction observerAction : observerActions) {
20             executor.execute(new Runnable() {
21                 @Override
22                 public void run() {
23                     observerAction.execute(event);
24                 }
25             });
26         }
27     }
28 }
```

5.AsyncEventBus

有了 EventBus，AsyncEventBus 的实现就非常简单了。为了实现异步非阻塞的观察者模式，它就不能再继续使用 MoreExecutors.directExecutor() 了，而是需要在构造函数中，由调用者注入线程池。

 复制代码

```
1 public class AsyncEventBus extends EventBus {
2     public AsyncEventBus(Executor executor) {
```

```
3     super(executor);  
4 }  
5 }
```

至此，我们用了不到 200 行代码，就实现了一个还算凑活能用的 EventBus，从功能上来讲，它跟 Google Guava EventBus 几乎一样。不过，如果去查看 [Google Guava EventBus 的源码](#)，你会发现，在实现细节方面，相比我们现在的实现，它其实做了很多优化，比如优化了在注册表中查找消息可匹配函数的算法。如果有时间的话，建议你去读一下它的源码。

重点回顾

好了，今天的内容到此就讲完了。我们来一块总结回顾一下，你需要重点掌握的内容。

框架的作用有：隐藏实现细节，降低开发难度，做到代码复用，解耦业务与非业务代码，让程序员聚焦业务开发。针对异步非阻塞观察者模式，我们也可以将它抽象成框架来达到这样的效果，而这个框架就是我们这节课讲的 EventBus。EventBus 翻译为“事件总线”，它提供了实现观察者模式的骨架代码。我们可以基于此框架，非常容易地在自己的业务场景中实现观察者模式，不需要从零开始开发。

很多人觉得做业务开发没有技术挑战，实际上，做业务开发也会涉及很多非业务功能的开发，比如今天讲到的 EventBus。在平时的业务开发中，我们要善于抽象这些非业务的、可复用的功能，并积极地把它们实现成通用的框架。

课堂讨论

在今天内容的第二个模块“EventBus 框架功能需求介绍”中，我们用 Guava EventBus 重新实现了 UserController，实际上，代码还是不够解耦。UserController 还是耦合了很多跟观察者模式相关的非业务代码，比如创建线程池、注册 Observer。为了让 UserController 更加聚焦在业务功能上，你有什么重构的建议吗？

欢迎留言和我分享你的想法。如果有收获，也欢迎你把这篇文章分享给你的朋友。

本周热门直播

- 没有代码洁癖的程序员，是不是好程序员？
- 如何成为一名“面霸”？
- 大厂面试问的那些冷门问题，在工作中真就不会用到吗？
- 如何才能学好纷繁复杂的 Spring 技术栈？
- 别焦虑，你得想自己怎么做才能成为“团队骨干”



微信扫码，进入直播观众席>>>

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 56 | 观察者模式（上）：详解各种应用场景下观察者模式的不同实现方式

下一篇 58 | 模板模式（上）：剖析模板模式在JDK、Servlet、JUnit等中的应用

精选留言 (27)

写留言



小文同学

2020-03-13

Guava EventBus 对我来说简直是一份大礼。里面解耦功能使本来的旧项目又不可维护逐渐转化为可维护。

EventBus作为一个总线，还考虑了递归传送事件的问题，可以选择广度优先传播和深度优先传播，遇到事件死循环的时候还会报错。Guava的项目对这个模块的封装非常值得我们...
展开 ∨



15



下雨天

2020-03-13

课后题：

代理模式，使用一个代理类专门来处理EventBus相关逻辑。作用：

- 1.将业务与非业务逻辑分离
- 2.后续替换EventBus实现方式直接改写代理类，满足拓展需求

展开 ∨



7



辣么大

2020-03-13

重构使用代理模式，将非业务代码放到代理类中。

另外试了争哥讲的EventBut类，在定义观察者的入参要修改成*Long*类型，如果使用long，这个方法是无法注册的，代码执行收不到通知。应该是ObserverRegistry类需要完善一下。

@Subscribe...

展开 ∨



2



饭

2020-03-13

老师，我们主要做物流方面的业务系统，类似仓储，港口这样的，流程繁杂。平时主要就是写增删改查，然后通过一个状态字段变化控制流程，所有业务代码流程中每一步操作都写满了各种状态验证，判断。后期稍微需求变动一点点，涉及到状态改动，要调整流程的话，都是一场灾难。针对我们这种系统，有办法将流程状态解耦出来吗？今天看到这篇事件总线的文章，好像看到希望，但是没想清具体怎么操作。不知道老师怎么看

展开 ∨



1



Heaven

2020-03-13

对于这个问题,在UserController中,我们应该只保留post函数() 发送的相关逻辑,而将注册Observer,初始化EventBus相关逻辑剔除,如果非要使用EventBus来实现的话,我们需要有人帮我们去进行注册和初始化,这时候就可以立马想到之前讲的工厂模式的DI框架,我们可以让所有观察者都被DI框架所管理,并且对EventBus创建一个装饰器类,在这个装饰器类中,由开发者选择注入线程池实现异步发送还是直接使用同步发送的,并且在init函数中 从DI框架管...

展开 ∨



1



Jeff.Smile

2020-03-13

在例子中当eventbus调用post传递的参数中是long userId,而两个observer被subscriber注解的方法参数都一样，此时这两个方法都会被调用到吗？

展开 ∨



右耳听海

2020-03-17

给老师点赞，虽然很早就接触了eventbus，但一直没明白这个的设计思想，现在有种醍醐灌顶的感觉



Geek_76616d

2020-03-16

对Guava EventBus相见恨晚啊

展开 ∨



blackhole

2020-03-15

提个问题：

文中“所谓可匹配指的是，能接收的消息类型是发送消息（post 函数定义中的 event）类型的子类”这话似乎有问题，应该是父类吧？

展开 ∨



陈玉群

2020-03-14

争哥，在EventBus 框架功能需求介绍里面，如果XMsg 是 YMsg 的父类，则post(xMsg); => AObserver、BObserver接收到消息，这个地方应该是如果XMsg 是 YMsg 的子类。



Frank

2020-03-14

为了让 UserController 更加聚焦在业务功能上，我的想法是将耦合的EventBus代码抽取出来形成一个单独的服务类，通过注入的方式注入到UserController类中使用。这样使其两者的职责单一，而新抽取出来的服务类可被其他业务场景复用。

今天也加深了对Guava EventBus的认识，虽然之前专栏也介绍过这个类库的使用。结合Jdk提供的java.util.Observable&Observer观察者模式API，与EventBus进行比对，如果...

展开 ∨



cricket1981

2020-03-14

`public void handleRegSuccess(long userId)` 方法签名中的`long`类型应该改成`Long`类型，不然`SubscriberRegistry.getSubscribers(Object event)`会匹配不上类型

展开 ▾



爱麻将

2020-03-14

最近公司做了个业务系统架构重构，套用了其它公司的业务架构，架构与业务耦合的太紧，做起来非常痛苦，越来越觉得跟争哥写的专栏相违背。

展开 ▾



小晏子

2020-03-13

我的想法比较直接，将`UserController`中的业务代码提出来放在接口的实现类中，这个`UserController`可以改名为`EventController`，然后这个接口实现类注入到这个`EventController`中，这样业务逻辑和控制逻辑就分离了，示例如下：

```
interface iController {  
    object register()...
```

展开 ▾



hanazawakana

2020-03-13

单独用一个工具类来处理eventbus相关的注册和post操作。然后通过依赖注入传给`UserController`



1012

2020-03-13

`UserController` 耦合的跟观察者模式相关的非业务代码可以使用代理模式进行重构



test

2020-03-13

课堂讨论：装饰器模式修饰`UserController`，在装饰器类里面创建线程池，注册`Observer`。





让爱随风

2020-03-13

```
XMsg xMsg = new XMsg();  
YMsg yMsg = new YMsg();  
ZMsg zMsg = new ZMsg();  
post(xMsg); => AObserver、BObserver接收到消息...
```

展开 ▾



Eden Ma

2020-03-13

使用单例作为通知中心将创建线程和注册observer的代码放在里面,将被观察者状态注入到单例类,进而通知观察者.

展开 ▾



Ken张云忠

2020-03-13

UserController 还是耦合了很多跟观察者模式相关的非业务代码,比如创建线程池、注册Observer。为了让 UserController 更加聚焦在业务功能上,你有什么重构的建议吗?
创建一个UserSubject类,将线程创建和注册Observer逻辑封装在该类型,再通过依赖注入方式注入到UserController,最后UserController只需UserSubject的post函数就可以发送消息了.

展开 ▾

