57讲观察者模式（下）：如何实现一个异步非阻塞的EventBus框架

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：

```
// 第一种实现方式，其他类代码不变，就没有再重复罗列
public class RegPromotionObserver implements RegObserver {
  private PromotionService promotionService; // 依赖注入

  @Override
  public void handleRegSuccess(long userId) {
    Thread thread = new Thread(new Runnable() {
```

```
    Thread thread = new Thread(new Runnable() {
      @Override
      public void run() {
        promotionService.issueNewUserExperienceCash(userId);
      }
    });
    thread.start();
  }
}

// 第二种实现方式，其他类代码不变，就没有再重复罗列
public class UserController {
  private UserService userService; // 依赖注入
  private List<RegObserver> regObservers = new ArrayList<>();
  private Executor executor;

  public UserController(Executor executor) {
    this.executor = executor;
  }

  public void setRegObservers(List<RegObserver> observers) {
    regObservers.addAll(observers);
  }

  public Long register(String telephone, String password) {
    //省略输入参数的校验代码
    //省略userService.register()异常的try-catch代码
    long userId = userService.register(telephone, password);

    for (RegObserver observer : regObservers) {
      executor.execute(new Runnable() {
        @Override
        public void run() {
          observer.handleRegSuccess(userId);
        }
      });
    }

    return userId;
  }
}
```

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：

```
public class UserController {
  private UserService userService; // 依赖注入

  private EventBus eventBus;
  private static final int DEFAULT_EVENTBUS_THREAD_POOL_SIZE = 20;

  public UserController() {
    //eventBus = new EventBus(); // 同步阻塞模式
    eventBus = new AsyncEventBus(Executors.newFixedThreadPool(DEFAULT_EVENTBUS_THREAD_POOL_SIZE)); // 异步非阻塞
  }

  public void setRegObservers(List<Object> observers) {
    for (Object observer : observers) {
      eventBus.register(observer);
    }
  }

  public Long register(String telephone, String password) {
    //省略输入参数的校验代码
    //省略userService.register()异常的try-catch代码
    long userId = userService.register(telephone, password);

    eventBus.post(userId);

    return userId;
  }
}

public class RegPromotionObserver {
```

```
  private PromotionService promotionService; // 依赖注入

  @Subscribe
  public void handleRegSuccess(long userId) {
    promotionService.issueNewUserExperienceCash(userId);
  }
}

public class RegNotificationObserver {
  private NotificationService notificationService;

  @Subscribe
  public void handleRegSuccess(long userId) {
    notificationService.sendInboxMessage(userId, "...");
  }
}
```

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：

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

- 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。

```
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：

```
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：

```
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）类型的子类。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：

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

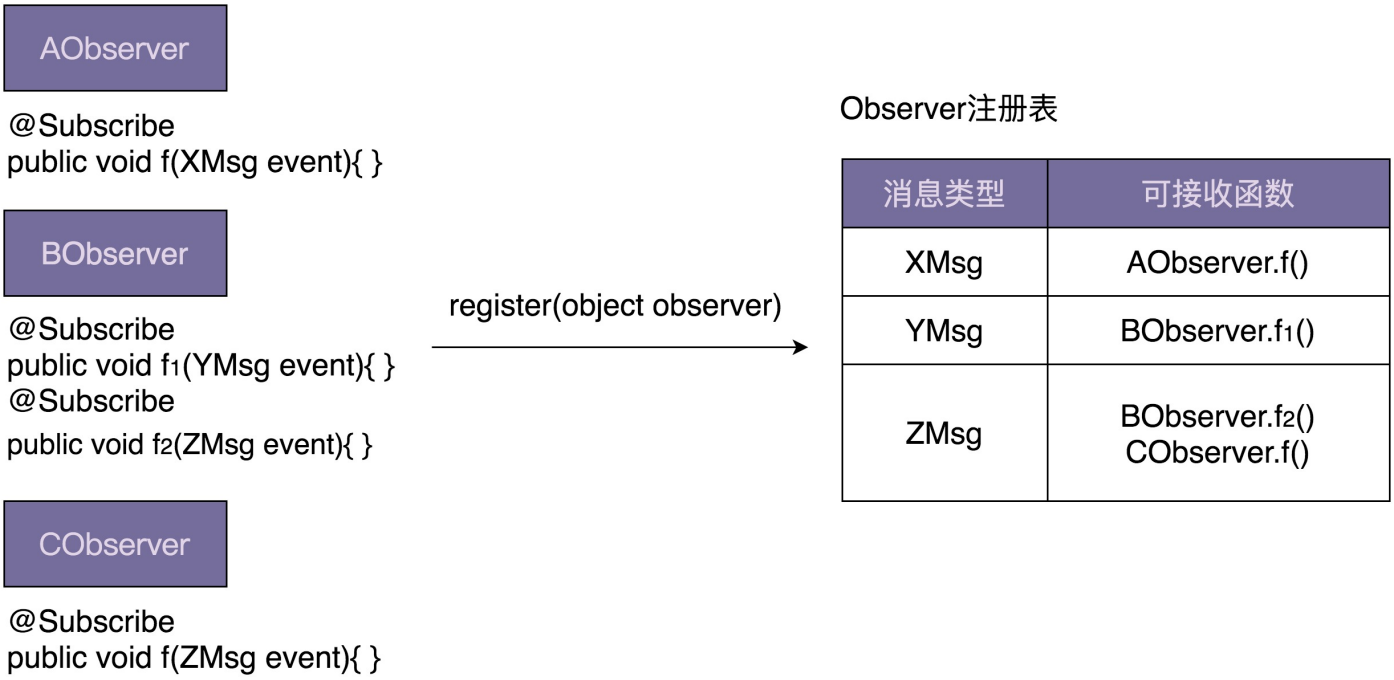
```
public DObserver {
  //...省略其他属性和方法...

  @Subscribe
  public void f1(PMsg event) { //... }

  @Subscribe
  public void f2(QMsg event) { //... }
}
```

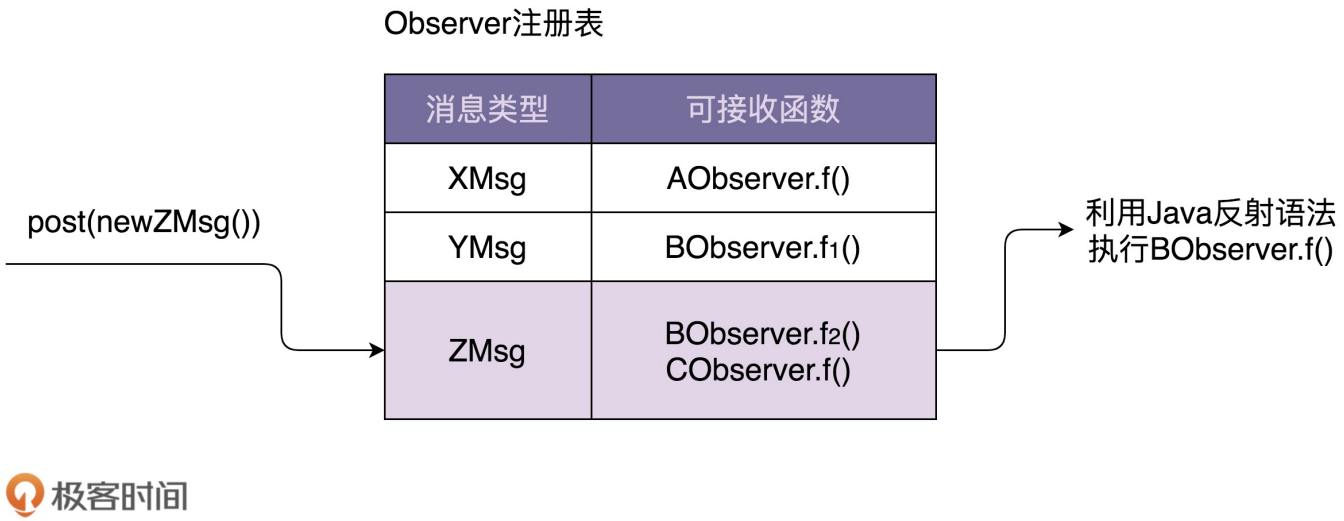
当通过register()函数将DObserver 类对象注册到EventBus的时候，EventBus会根据@Subscribe注解找到f1()和f2()，并且将两个函数能接收的消息类型记录下来（PMsg->f1，QMsg->f2）。当我们通过post()函数发送消息（比如QMsg消息）的时候，EventBus会通过之前的记录（QMsg->f2），调用相应的函数（f2）。

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。





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。

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
@Beta
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。

```
public class ObserverAction {
  private Object target;
  private Method method;

  public ObserverAction(Object target, Method method) {
    this.target = Preconditions.checkNotNull(target);
    this.method = method;
    this.method.setAccessible(true);
  }

  public void execute(Object event) { // event是method方法的参数
    try {
      method.invoke(target, event);
    } catch (InvocationTargetException | IllegalAccessException e) {
      e.printStackTrace();
    }
  }
}
```

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。

```
public class ObserverRegistry {
  private ConcurrentMap<Class<?>, CopyOnWriteArraySet<ObserverAction>> registry = new ConcurrentHashMap<>();
```

```
private ConcurrentMap<Class<?>, CopyOnWriteArraySet<ObserverAction>> registry = new ConcurrentHashMap<>();

public void register(Object observer) {
  Map<Class<?>, Collection<ObserverAction>> observerActions = findAllObserverActions(observer);
  for (Map.Entry<Class<?>, Collection<ObserverAction>> entry : observerActions.entrySet()) {
    Class<?> eventType = entry.getKey();
    Collection<ObserverAction> eventActions = entry.getValue();
    CopyOnWriteArraySet<ObserverAction> registeredEventActions = registry.get(eventType);
    if (registeredEventActions == null) {
      registry.putIfAbsent(eventType, new CopyOnWriteArraySet<>());
      registeredEventActions = registry.get(eventType);
    }
    registeredEventActions.addAll(eventActions);
  }
}

public List<ObserverAction> getMatchedObserverActions(Object event) {
  List<ObserverAction> matchedObservers = new ArrayList<>();
  Class<?> postedEventType = event.getClass();
  for (Map.Entry<Class<?>, CopyOnWriteArraySet<ObserverAction>> entry : registry.entrySet()) {
    Class<?> eventType = entry.getKey();
    Collection<ObserverAction> eventActions = entry.getValue();
    if (eventType.isAssignableFrom(postedEventType)) {
      matchedObservers.addAll(eventActions);
    }
  }
  return matchedObservers;
}

private Map<Class<?>, Collection<ObserverAction>> findAllObserverActions(Object observer) {
  Map<Class<?>, Collection<ObserverAction>> observerActions = new HashMap<>();
  Class<?> clazz = observer.getClass();
  for (Method method : getAnnotatedMethods(clazz)) {
    Class<?>[] parameterTypes = method.getParameterTypes();
    Class<?> eventType = parameterTypes[0];
    if (!observerActions.containsKey(eventType)) {
      observerActions.put(eventType, new ArrayList<>());
    }
    observerActions.get(eventType).add(new ObserverAction(observer, method));
  }
  return observerActions;
}

private List<Method> getAnnotatedMethods(Class<?> clazz) {
```

```
    List<Method> annotatedMethods = new ArrayList<>();
    for (Method method : clazz.getDeclaredMethods()) {
      if (method.isAnnotationPresent(Subscribe.class)) {
        Class<?>[] parameterTypes = method.getParameterTypes();
        Preconditions.checkArgument(parameterTypes.length == 1,
                "Method %s has @Subscribe annotation but has %s parameters."
                        + "Subscriber methods must have exactly 1 parameter.",
                method, parameterTypes.length);
        annotatedMethods.add(method);
      }
    }
    return annotatedMethods;
  }
}
```

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。

```
public class EventBus {
  private Executor executor;
  private ObserverRegistry registry = new ObserverRegistry();

  public EventBus() {
    this(MoreExecutors.directExecutor());
  }

  protected EventBus(Executor executor) {
    this.executor = executor;
  }

  public void register(Object object) {
    registry.register(object);
  }

  public void post(Object event) {
    List<ObserverAction> observerActions = registry.getMatchedObserverActions(event);
    for (ObserverAction observerAction : observerActions) {
      executor.execute(new Runnable() {
        @Override
        public void run() {
          observerAction.execute(event);
        }
      });
    }
  }
}
```

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。

```
public class AsyncEventBus extends EventBus {
  public AsyncEventBus(Executor executor) {
    super(executor);
  }
}
```

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。

精选留言

小文同学

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去阅读，复杂的都在里头，外面极为易用，我拷贝了一份EventBus的代码进行修改以适配自己的项目，发觉里面的构造都极为精密巧妙，像一个机械钟表一样，自己都下不了手，觉得不小心就是弄坏了。

跟随真正优秀的工程师，并阅读其写出来的代码让人受益匪浅。

2020-03-13 14:19

下雨天

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

2020-03-13 09:51

辣么大

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

public void handleRegSuccess(Long userId) {

System.out.println("handleRegSuccess...");

promotionService.issueNewUserExperienceCash(userId);

}

代码见：https://github.com/gdhucoder/Algorithms4/tree/master/designpattern/u57

2020-03-13 22:11

Heaven

对于这个问题,在UserCntroller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框架管理的对象池中拿出所有标有@Subscribe注解的类,保存到ObserverRegistry中,对于所有需要使用EventBus的类,注入这个装饰器类即可,设计的好,甚至可以做到其他依赖代码都不用改一点

2020-03-13 11:27

blacknhole

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?

2020-03-15 02:14

陈玉群

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。

2020-03-14 23:09

Frank

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，如果要实现进程内的观察者使用EventBus最为方便。从JDK9之后，java.util.Observable&Observer已被标记为废弃，建议使用Java Beans规范中的事件模式和java.util.concurrent.Flow API。

2020-03-14 22:16

cricket1981

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

2020-03-14 10:04

爱麻将

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。

2020-03-14 01:32

小晏子

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()
}

public class UserControllerService implement iController {
private string telphone;
private string password;

public Long register() {
long userId = userService.register(telephone, password);
return userId;
}
}

public class EventController {
private iController iService;

private EventBus eventBus;
private static final int DEFAULT_EVENTBUS_THREAD_POOL_SIZE = 20;

public EventController() {
```

eventBus = new AsyncEventBus(Executors.newFixedThreadPool(DEFAULT_EVENTBUS_THREAD_POOL_SIZE)); // 异步非阻塞模式
}

public void setRegObservers(List<Object> observers) {
for (Object observer : observers) {
eventBus.register(observer);
}
}

public void SendMessage() {
object msg = iService.register()
eventBus.post(msg)
}

}
2020-03-13 23:49

hanazawakana
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
2020-03-13 19:06

1012
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
2020-03-13 14:50

饭
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
2020-03-13 13:24

test
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。
2020-03-13 13:10

让爱随风

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

感觉这个是不是不对啊，感觉应该是：
XMsg xMsg = new XMsg();
YMsg yMsg = new YMsg();
ZMsg zMsg = new ZMsg();
post(xMsg); => AObserver 接收到消息
post(yMsg); => AObserver，BObserver接收到消息
post(zMsg); => AObserver，BObserver，CObserver接收到消息

2020-03-13 12:03

Eden Ma
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.
2020-03-13 11:55

Ken张云忠

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逻辑封装在进该类型,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.

2020-03-13 11:32

守拙

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?

使用EventBus后, setRegObservers()方法无需在UserController中调用了, 每个Observer的构造器中EventBus.register(this)就可以了. EventBus的意义就在于将Observable与Observer彻底解耦.

EventBus作为系统中唯一的组件, 可以设计成单例模式. Observable可以直接通过EventBus.getDefault().post(XXEvent())的方式使用, Observable无需依赖注入.

线程池的创建可以使用Builder模式为EventBus配置. 在Application进程初始化时, 即配置EventBus的线程池. 如此Observable就可以无需考虑线程池的配置.

EventBus可以提供unregister()以便observer生命周期管理.

2020-03-13 10:54

,

课后题:可以将EventBus和AsyncEventBus存入spring容器中,使用前先将ObServer注册进去,之后使用的时候只要依赖注入就可以了

2020-03-13 09:50

gogo

看了下google EventBus源码，是标注了@Beta的，能用于生产环境吗?

2020-03-13 09:45