57-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() {
      @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
  }

  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()。

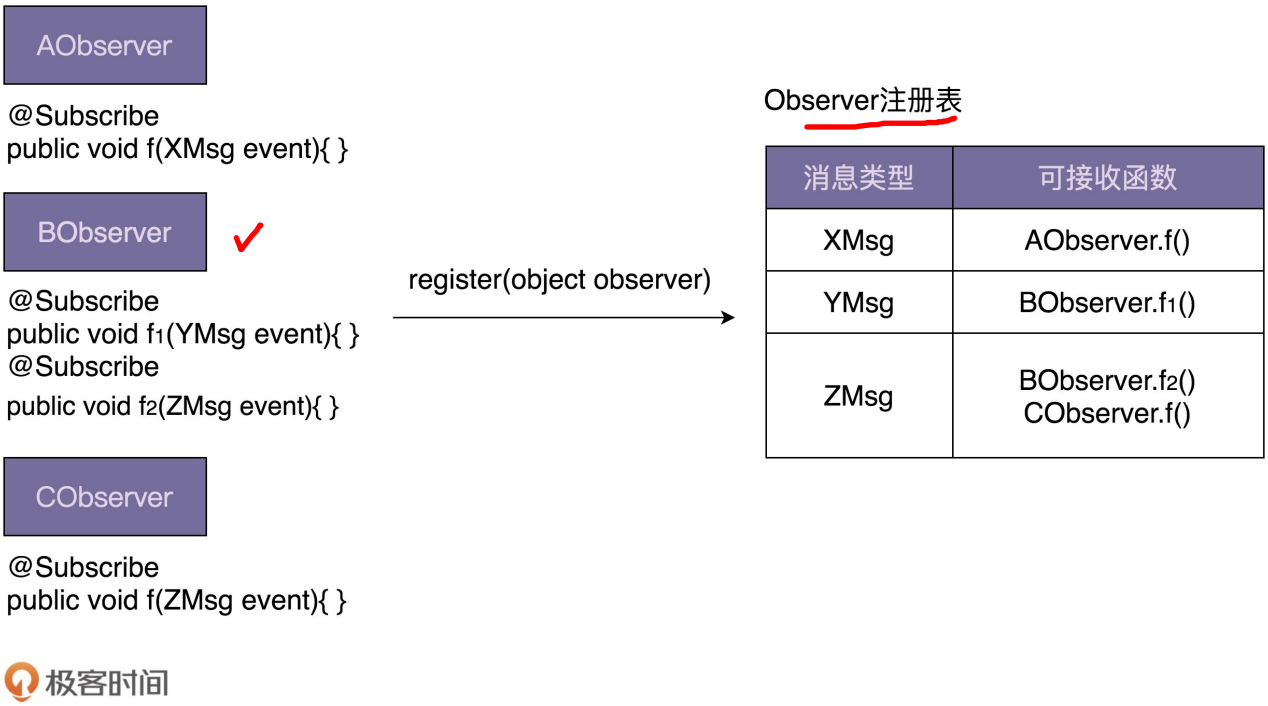
```
public DObserver {
  //...省略其他属性和方法...

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

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

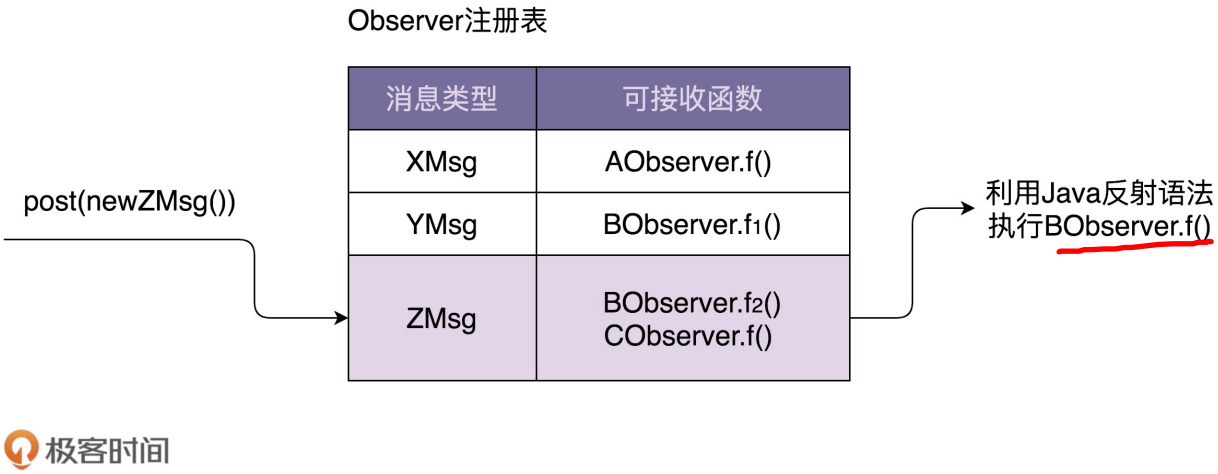
当通过register()函数将DObserver 类对象注册到EventBus的时候，EventBus会根据@Subscribe注解找到f1()和f2()，并且将两个函数能接收的消息类型记录下来（PMsg->f1，QMsg->f2）。当我们通过post()函数发送消息（比如QMsg消息）的时候，EventBus会通过之前的记录（QMsg->f2），调用相应的函数（f2）。

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。





极客时间

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<>()

  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：

- 孙志强 2020-03-13 08:19:31
  EventBus和Spring里的事件机制好像

- Liam 2020-03-13 08:12:53
  生产消费者模式可实现，eventbus管理一个消息队列，观察者自己注册到evenbus，业务发消息到队列后，取出来给匹配的观察者执行

- 黄林晴 2020-03-13 07:59:35
  打卡

- rayjun 2020-03-13 07:35:25
  EventBus可以使用依赖注入的方式注入进来

- Jeff.Smile 2020-03-13 07:09:53
  在例子中当eventbus调用post传递的参数中是long userId,而两个observer被subcriber注解的方法参数都一样，此时这两个方法都会被调用到吗？