

Takia-ddd 开发者指南

Takia-ddd 是什么？

Takia-ddd 是 DDD 领域驱动设计 Java 实施框架，使用 Takia 能快速地将领域驱动设计落地为异步、高并发、高吞吐量的应用系统。

Takia-ddd = Actors + Ioc/DI + AOP

Takia-ddd = DDD + DCI + Domain Events/Event Sourcing/CQRS, Takia 通过如下统一语言将 DDD 和 DCI 以及事件灵活结合在一起：

关键技术特点：

(1) DDD(Domain-Driven Development)，开发基于领域驱动设计(Domain-Driven Design)应用，提供基于内存的领域模型(in memory model)，运行时刻领域对象作为“总司令部”通过 Domain Events 驱动命令技术构架为之服务，探索了一条真正以业务对象为核心的崭新的 DDD 落地编程模型。

(2) 事件驱动架构 Event-driven Architecture(EDA),异步领域事件,并发策略,懒加载赋值(Lazy initialization or evaluation)，异步消息机制，结合 JMS 可实现大型分布式可伸缩的架构,6.4 整合入号称最快的并发框架 Disruptor。

(3) 依赖注入 DI 和 AOP 框架,类自动配对注射 autowiring/Autowired，无需指定，提高重构效率，所有类最大限度松耦合，包括框架本身的类或构件都是可替换的，提供强大定制能力;灵活简单的 AOP,没有复杂 AOP 脚本代码,可以将任何 POJO 引入 introduce 作为拦截器。

(4)命令查询分类架构 Command Query Responsibility Segregation(CQRS/CQS)，提供模型的增删改查命令流程整合，不必编写 MVC 模式中 Controller 控制器，防止新手将业务写入控制器。服务命令模式：可根据 url 参数直接激活对应的 Service 方法；提供大量数据批量查询自动分页和缓存性能优化功能。

快速入门 ABC

一步到位

直接使用 Takia 和 Spring 框架的 Annotation，写好代码即可部署运行。

注意：本案例不是 DDD 领域事件或 DCI 架构：

举例如下：

在 HelloServiceImpl 加入注解：

```
@Service("helloService")
public class HelloServiceImpl implements HelloService {
    private UserRepository userRepository;

    public HelloServiceImpl(UserRepository userRepository) {
        this.userRepository = userRepository;
    }

    public String hello(String name) {
        User user = userRepository.findUser(name);
        System.out.print("call ok");
        return "Hello, " + user.getName();
    }
}
```

HelloServiceImpl 中并没有对接口 UserRepository 实例化，只需在接口 UserRepository 的子类中加入注解@Component，如下：

```
@Component
public class UserRepositoryInMEM implements UserRepository {

}
```

如果在 Servlet 或 JSP 等客户端调用上述代码，通过 Spring 完成如下：

```
HelloService helloService = (HelloService) beanFactory.getBean("helloService");
String result = helloService.hello(myname);
```

输出：Hello XXXX

客户端代码中也没有将接口 UserRepository 子类 UserRepositoryInMEM 实例化创建后，赋给 HelloServiceImpl，这一切都是由 Takia 框架悄悄在背后实现了。

CQRS 入门

阅读本入门需要有 DDD 领域驱动设计和 CQRS 知识。

在 Takia-ddd 框架中有两个模型: Component(组件) 和 Model(领域模型). 分别以 @Component, 和 @Model 标注。

当一个 Model 被外部组件访问, 它一般是 DDD 中的聚合根实体, 因为根据 DDD 只有聚合根实体才能被外界访问引用, 外部不能直接访问聚合边界内其它对象, 必须通过聚合根, 这样聚合根才能保证聚合边界内各个对象变化的一致性。

如果一个 Model 被其他领域模型引用, 它就肯定不是聚合根, 因为聚合根之间不能直接相互引用, 它可能是聚合内一个对象, 或者是实体或者是值对象。

领域模型 Model 实例生活在 in-memory 内存缓存中, 而组件 Component 实例的生命周期是应用级别, 比如和 Web 容器相同, 一个容器内缺省是一个单例。

Component 能够用来实现 DDD 的服务 service 或其他应用管理器, 如邮件发送等。

Takia 框架也提供一种类似 Component 的 Service 类型 (标注为@Service), 它是面向外部客户端, 而不是面向内部, 可用来实现 SOA 的粗粒度大服务。

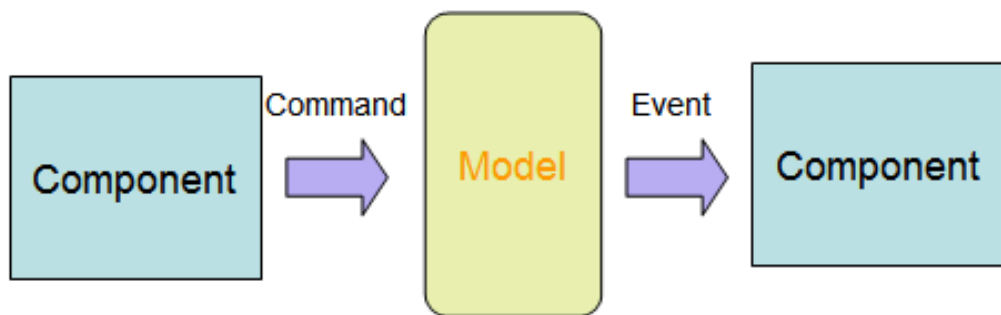
Takia 在这两个模型(Component 和 Model)之间提供四种异步并发的通讯方式, 也是一种 Producer/Consumer 模式。

1. 组件和模型 Component -----> model
2. 模型和组件 model ----->Component
3. 组件与组件 Component -----> Component
4. 模型与模型 model-----> model

当一个组件或服务 Component/Service 发送消息给领域模型 Model(也就是聚合根 aggregate root), 在 CQRS 中我们称这个消息携带的是命令 command, 当一个领域模型 model 发送消息给组件 Component, 我们称它为事件, 代表已经在领域模型中发生什么事情: :



一个命令激活聚合根模型的行为,如上面的 `startMatch` 方法,然后在这个方法执行时,一个事件也发生了,这个事件可激活其他聚合根或组件协同工作。比如让 `Repository` 组件保存模型自身等。



下面谈谈这四种通讯方式如何使用 Takia 实现:

组件与模型

Component(producer with @Component) --> Model(consumer with @Model)

在这个方式下,其实代表 CQRS 的 Command,一个命令可能来自 UI 或其他聚合根的事件,将发往聚合根实体,一个命令激活聚合根实体的一个方法行为。

这种方式下生产者和消费者 producer:consumer 只能是 1:1,一个命令只能发往一个聚合根实体模型,由这个聚合根模型根据业务规则检查命令是否有效,是否可以执行等等。

下面是一个生产者为组件的代码,使用 @Component:标注。组件要求有接口和实现

两个类。

```
package com.xxx.sample.test.command;

import com.reeham.component.ddd.annotation.model.Send;

public interface AICommand {

    @Send("CommandmaTest")
    public TestCommand ma(@Receiver BModel bModel);

}

@Component("producer")
@Introduce("message")
public class A implements AICommand {

    public TestCommand ma(BModel bModel) {
        System.out.print("send to BModel =" + bModel.getId());
        return new TestCommand(99);
    }

}
```

方法名为 "ma"是用元注解 `@Send`，对于发送一个命令，还必须在该方法的输入参数中指定该命令发往哪个具体的聚合根实体实例，所以要使用 `@Receiver`来指定，这里是 `BModel` 实例。

```
@Model
public class BModel {
    private String id;

    private int state = 100;

    public BModel(String id) {
        super();
        this.id = id;
    }

    @OnCommand("CommandmaTest")
    public void save(TestCommand testCommand) {
        this.state = testCommand.getInput() + state;
        testCommand.setOutput(state);
    }
}
```

```

    }

    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }
}

```

这里命令的生产者和消费者之间是通过主题队列 `Queue` 名为 "CommandmaTest" 进行联系，这个名词全局必须唯一，代表两者之间是 1:1，而不是 1:N 的关系。`OnCommand` 代表消费者 `consumer` 响应的方法。

整个调用过程是：生产者 `AICommand` 实例的 `ma` 方法命令消费者 `BModel` 的 `save` 方法立即执行，生产者 `AICommand` 代表一个组件 `Component`，而消费者 `BModel` 是一个 `Model`。

客户端代码：

```

AICommand a = (AICommand) beanFactory.getBean("producerforCommand");
BModel bModel = new BModel("one");
TestCommand testCommand = a.ma(bModel);
int i = 0;
long start = System.currentTimeMillis();
while (testCommand.getOutput() != 199) {
    i++;
}
long stop = System.currentTimeMillis();
Assert.assertEquals(testCommand.getOutput(), 199);
System.out.print("ok " + " " + (stop - start));

```

输出结果：

send to BModel =oneok 5

这个模式下，组件 `component` 承接来自 `UI` 或其他事件转化成的命令，也可以称为 `command handler`，如下：

`UI --->commandHandler(@Component) ---->聚合根 aggregate root`

聚合根的方法被命令激活执行，执行是单线程的，因此某个时刻只有一个线程修改模型内部状态，避免了使用锁或两阶段事务等低吞吐量低效能方式。

模型和组件

Model(producer with @Model) --> Component(consumer with @Component)

当一个聚合根实体模型接受到命令然后执行以后，它会在方法执行过程中激活 reactive 一个事件，这个事件称为领域事件 domain events，将被发往其他组件或其他聚合根实体模型。

为了实现聚合根模型作为事件的生产者，我们可以将一个组件 Component(with @Component) 注入到模型中，这样模型就作为生产者。

聚合根模型代码:

```
@Model
public class MyModel {

    private Long id;
    private String name;

    @Inject //inject the Component into this domain model
    private MyModelDomainEvent myModelDomainEvent;

    public String getName() {
        if (this.name == null) {
            DomainMessage message = myModelDomainEvent.asyncFindName(this);
            this.name = (String) message.getEventResult();
        }
        return name;
    }

    ....

}
```

我们使用了 @Inject, 将 MyModelDomainEvent 实例注入到 "MyModel" 中，而 "MyModelDomainEvent" 是一个组件，在其中我们完成事件生产者的发送方法：

```
package com.xxx.sample.test.domain.simplecase;

import com.reeham.component.ddd.annotation.Introduce;
import com.reeham.component.ddd.annotation.model.Send;
import com.reeham.component.ddd.message.DomainMessage;

@Introduce("message")
public class MyModelDomainEvent {
```

```

        @Send("MyModel.findName")
        public DomainMessage asyncFindName(MyModel myModel) {
            return new DomainMessage(myModel);
        }

        @Send("saveMyModel")
        public DomainMessage save(MyModel myModel) {
            return new DomainMessage(myModel);
        }
    }
}

```

MyModelDomainEvent 必须标注为 `@Introduce("message")`，表示引入一个拦截器叫 `message`，这是在 Takia 框架 Spring 配置文件中事先定义的。在这个事件的生产者类中有两个主题 `topic`，注意这里是主题，而不是队列 `Queue`，表示每个主题 `topic` 可以实现 `producer:consumer` 为 1:N 的事件发送。

下面看看对于领域事件的响应器也就是消费者的代码，消费者也是一个组件：

```

@Consumer("MyModel.findName")
public class FindNameListener implements DomainEventHandler {

    public void onEvent(EventDisruptor event, boolean endOfBatch) throws Exception {
        MyModel myModel = (MyModel) event.getDomainMessage().getEventSource();
        System.out.println("Asynchronous eventMessage=" + myModel.getId());
        event.getDomainMessage().setEventResult("Asynchronous eventMessage=" +
myModel.getId());
    }
}

```

FindNameListener 是使用新的元注解 `@Consumer`，注意不是 `@Component`，表示这是一个消费者组件，使用 `@Consumer`，这个类就必须继承实现接口 `DomainEventHandler`，然后在其方法 `onEvent` 中完成对某个 `topic` 的生产者响应。

如果使用 `@Component`，就必须使用 `@OnEvent` 标注你自己的对某个 `topic` 响应的方法。这是两种不同的消费者写法，一个 `topic` 可以有多个消费者，执行顺序是按照类的包名完整字符串排列。

下面是后一种消费额写法，实现生产者 `@Send("saveMyModel")` 的消费：

```

@Component("mymrepository")
@Introduce("domainCache")
public class RepositoryImp implements MyModelRepository {

    @Around
    public MyModel getModel(Long key) {

```



```

        MyModel mym = new MyModel();
        mym.setId(key);
        return mym;
    }

    @OnEvent("saveMyModel")
    public void save(MyModel myModel) {
        System.out.println("\n No.1 @OnEvent:" + this.getClass().getName());
    }
}

```

这个消费者是 `RepositoryImp`，它是 DDD 的仓储 `Repository` 实现，主要负责从仓库或者数据库还原一个完整的聚合根实体对象。

`@Introduce("domainCache")` 和 `@Around` 配合使用，将聚合根模型能保存在内存缓存中，以后凡是调用此方法，总是先检查缓存是否已经存在这个模型。这两个元注释是使用 Takia 必须的，否则命令和事件都无法正常运行。当然，可以手工调用 `com.reeham.component.ddd.dci.RoleAssigner` 的 `assignAggregateRoot` 方法将任何一个对象扮演成一个聚合根实体。

至此，我们已经知道了组件和模型之间两种通讯方式，以上两种结合起来如下调用流程。

UI ----->commandHandler(@Component) ----->聚合根

聚合根 ----->EventHandler(@Component) ----->仓储持久数据库等

在这两种方式集合情况下，聚合根实体模型其实扮演的是类似 AKKA 或 Erlang 中的 Actors 模型，同样具备以下特性：

- Share NOTHING, 没有分享
- 隔离的事件驱动处理
- 输入或输出通讯都是异步 无堵塞的消息

.组件与组件

Component(producer with @Component) --> Component(consumer with @Component)

这个模式是组件和组件之间调用方式，分两种：

- 1.依赖注入同步调用
- 2.事件异步调用

自动依赖注入是 Takia 框架早期的一个功能，适合@Service 和@Model 之间的实例自动注入，如下：

```
@Component
public class A {

    private B b;

    //通过构造器将 B 实例注入
    public A(B b){
        this.b = b;
    }
}

@Component
public class B {

}
```

当组件 B 被注入到组件 A 以后，在 A 中就可以直接同步调用 B 的方法，还有一种更加松耦合的方法，A 不再依赖 B，就是通过异步并发事件实现。

如下 A 类作为发布者：

```
package com.xxx.sample.test.event;

import com.reeham.component.ddd.annotation.model.Send;

public interface AI {

    @Send("maTest")
    public TestEvent ma();
}

@Component("producer")
@Introduce("message")
public class A implements AI {

    public TestEvent ma() {
        System.out.print("event.send.ma..");
    }
}
```

```

        return new TestEvent(99);
    }
}

```

B 是事件的消费者，它的具体消费方法需要以 `@OnEvent` 标注。

```

@Component("consumer")
public class B {

    @OnEvent("maTest")
    public void mb(TestEvent testEvent) throws Exception {
        testEvent.setResult(testEvent.getS() + 1); // the consumer return a result to the producer
        System.out.print("event.@OnEvent.mb.." + testEvent.getResult() + "\n");
        Assert.assertEquals(testEvent.getResult(), 100);
    }
}

```

方法 "maTest" 标注了 `@OnEvent`，其 topic 名称需要和 `@Send` 相同，注意，生产者的方法返回类型是 "TestEvent"，必须等同于消费者的方法输入参数，这样实现两者之间共享一个消息对象，这个消息对象可以携带任何两者需要传递分享的值对象。

如果你希望消费者也返回一个结果给生产者，那么使用 `com.reeham.component.ddd.domain.message.DomainMessage`，其中方法 `getEventResult()` 根据堵塞设置等待消费者返回结果。

测试代码如下：

```

AI a = (AI) beanFactory.getBean("producer");
TestEvent te = a.ma();
long start = System.currentTimeMillis();
while (te.getResult() != 100) {
}
long stop = System.currentTimeMillis();

Assert.assertEquals(te.getResult(), 100);
System.out.print("ok " + " " + (stop - start) + "\n");

```

输出 output:

```

[junit] event.send.ma..event.@OnEvent.mb..100
[junit] ok 31

```

模型与模型

$\text{Model}(\text{aggregate root A}) \rightarrow \text{Model}(\text{aggregate root B})$

根据 Evans 的定义，聚合根内部维持一致性，但是聚合根之间不可以直接引用，可以实现最终一致性(CAP 定理)，

当一个聚合根实体模型需要协调另外一个聚合根时，只能通过领域事件，发出事件作为下一个聚合根的输入命令。

这个模式实际可以分解为由前面三个模式组合起来：

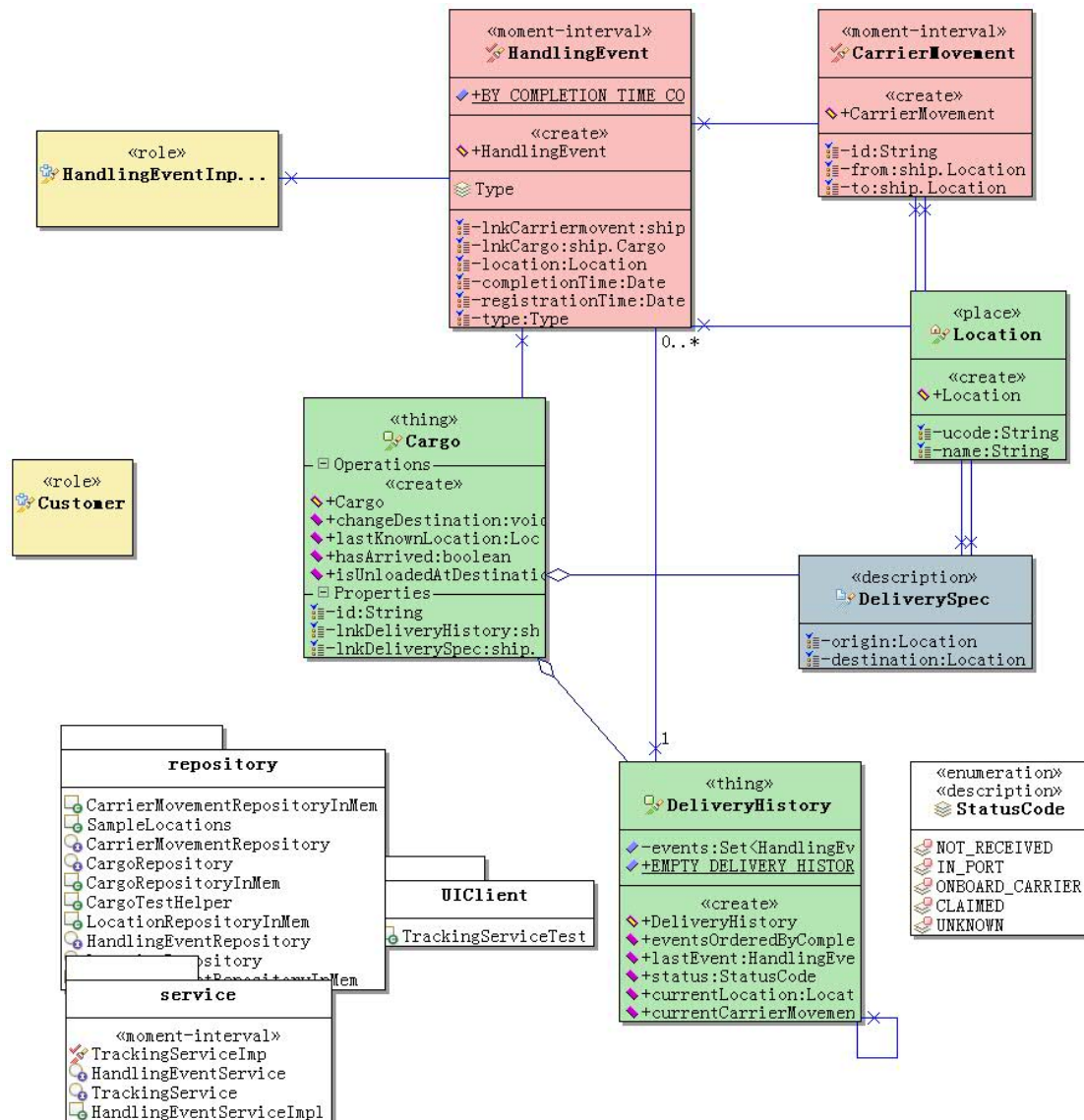
1. 聚合根 A 激活一个事件到组件 ($\text{model} \rightarrow \text{Component}$)
2. 组件将事件转为聚合根 B 的命令。 ($\text{Component} \rightarrow \text{Component}$)
3. 命令发往聚合根 B ($\text{component} \rightarrow \text{model}$)

Takia-ddd 概念

Takia 是一个 DDD 领域驱动设计框架，Domain Model + In-memory + Events，常驻内存 In-memory 的领域模型 Domain Model 通过领域事件 Domain Events 驱动技术实现各种功能，正如基因 DNA 是生命各种活动功能的核心一样，实现了以领域模型而不是数据表为核心的新的模型驱动开发架构 MDD。

领域驱动设计 DDD

当我们接到一个新项目时，使用 UML 工具，通过面向对象 DDD 的分析设计方法将其变成领域模型图，如下：



这是一个典型的 DDD 建模图,这个模型图可以直接和 Java 代码对应,比如其中 Cargo

模型的代码如下，两者是完全一一对应，可以使用 `together` 等建模工具直接转换，Takia 框架的 `@Model` 就是针对 `Cargo` 这样模型，将其运行在 Java 平台中，：

```
package ship;

@Model
public class Cargo {
    private String id;

    private ship.DeliveryHistory lnkDeliveryHistory;
    private ship.DeliverySpec lnkDeliverySpec;

    public Cargo(String trackingId, DeliverySpec deliverySpec) {
        this.id = trackingId;
        this.lnkDeliverySpec = deliverySpec;
    }

    public void changeDestination(final Location newDestination) {
        lnkDeliverySpec.setDestination(newDestination);
    }

    //跟踪货物位置
    public Location lastKnownLocation() {
        final HandlingEvent lastEvent = this.getLnkDeliveryHistory().lastEvent();
        if (lastEvent != null) {
            return lastEvent.getLocation();
        } else {
            return null;
        }
    }

    //当货物在运输抵达目的地时
    public boolean hasArrived() {
        return lnkDeliverySpec.getDestination().equals(lastKnownLocation());
    }

    //跟踪顾客货物的关键装卸事件
    public boolean isUnloadedAtDestination() {
        for (HandlingEvent event : this.getLnkDeliveryHistory().eventsOrderedByCompletionTime())
        {
            if (HandlingEvent.Type.UNLOAD.equals(event.getType())
                && hasArrived()) {
                return true;
            }
        }
        return false;
    }

    .....
}
```

当领域模型 Cargo 出来以后，下一步就是使用 Takia 框架来将其运行起来，因为 Takia 框架分为领域模型和组件技术等两个部分，Cargo 无疑属于 @Model 模型架构，我们只要给模型加上 @Model，就能让 Cargo 的对象生活在内存缓存中。

```
@Model
public class Cargo {
}
```

为什么需要 DDD?

以订单为例子，如果不采取 DDD 设计，而是通常朴素的数据表库设计，将订单设计为订单数据表，那么带来的问题是：

将实体的职责分离到不同限定场景，比如订单中有 OrderItemId, OrderId, ProductId 和 Qty，这是合乎逻辑的最初订单，后来有 MinDeliveryQty 和 PendingQty 字段，是和订单交货有关，这其实是两个概念，订单和订单的交货，但是我们把这些字段都混合在一个类中了。

混淆在一起的问题就是将来难以应付变化，因为实体的职责是各自变化的。

领域不是把实体看成铁板一块，一开始就把它分解到各种场景。下订单和订单交货交付是两个场景，它们应该有彼此独立的接口，由实体来实现。这就能够让实体和很多场景打交道，而彼此不影响。

DDD 和数据库分析设计的区别是：在数据库中它们是一个，也就是说，从 ER 模型上看，它们是一个整体，但是从领域模型角度看，它们是分离的。

传统架构与 DDD 脱节

DDD 和 Spring+Hibernate 或 JavaEE 的架构有什么区别？

其实，现有架构和 DDD 是脱节，也就是说，现有架构不能很好支撑 DDD，为什么这么说呢，因为在这样架构下，DDD 实体一直是被操作，作为方法参数传来传去，见下面伪代码演示：

```
public void myMethod(Entity entity){
.....
}
```

然后我们会有出现另外一个方法类似上面，只是其中方法代码有稍微不同：

```
public void myMethod2(Entity entity){
.....
}
```

```
public void myMethod3(Entity entity){
.....
}
```

问题来了，由于这三个方法中有一部分是共同的，当我们修改一个方法时，另外两个都要修改，万一忘记修改，就出现 BUG，这时我们很容易想把三个方法中共同部分抽象成一个方法，这里有两个方向：

首先用继承模板，其实这是坏的设计，为什么坏这里不多说。更主要的问题是：我们只是把几个功能类(服务类)合并成一个类，实体还是处于被传入被操作，如果这段操作需要事务，我们只能在这个合并类中加入事务，导致锁粒度扩大。

```
public class Service{

    public void myMethodCommon(Entity entity){
        //事务开始
        .....
        //事务结束
    }

}
```

上面这个服务类共同抽象出来的方法内加上事务后，尽管同时操作的是两个不同的实体对象，也会发生排他锁，某个时刻只能有一个实体在这个事务中被操作。

实际上，我们是要根据实体对象来进行事务，只有两个线程同时操作一个实体对象时，我们才需要事务排他锁。

所以，之前我们总是从功能行为角度考虑实现，换个不同角度，从实体角度考虑，这共同的部分是不是可以写入实体内部呢？也许从业务上讲，属于实体的行为，属于实体的职责，实体应该自己干的事情，应该有责任去做的事情，当然这其中也区分为基本职责和业务职责，结合特定业务场景的业务职责通过 DCI 来实现。

失血模型和充血模型

DDD 革命性在于：领域模型准确反映了业务语言，而传统 J2EE 或 Spring+Hibernate 等事务性编程模型只关心数据，这些数据对象除了简单 setter/getter 方法外，没有任何业务方法，被比喻成失血模型，那么领域模型这种带有业务方法的充血模型到底好在哪里？

以比赛 Match 为案例，比赛有“开始”和“结束”等业务行为，但是传统经典的方式是将“开始”和“结束”行为放在比赛的服务 Service 中，而不是放在比赛对象本身之中。如下图：

The Classic approach

```
@Entity
@Table(name="matches")
public class Match {
    @Id
    @Type(type="org.hibernate.type.UUIDCharType")
    private UUID id;

    @ManyToOne
    private Team homeTeam;

    @Column(updatable=false, insertable=false)
    private String homeTeamId;

    @ManyToOne
    private Team awayTeam;

    @Column(updatable=false, insertable=false)
    private String awayTeamId;

    @Basic
    private Date matchDate;

    @Basic
    private Date finishDate;

    private Score score;

    // getters + setters
}
```

比赛这个实体成了只有字段，或者 setter/getter 的贫血实体，这种实体大量使用在 JavaEE 或 Spring + Hibernate 中，或者是 JPA 实体或者是 Hibernate 实体，这种实体就对应 DB 表，并用 @Entity 修饰，在属性上也有 @NotNull，@Size 等。这种贫血实体其实是被数据库绑架了。

DDD 理论的创始人 Eric Evans 在 2012 年 Eric Evans 关于技术如何影响 DDD 的会谈中说：

对象概念其实很长时间已经被 J2EE 等重量框架摧残了很长时间，而且更被危险的映射到关系数据库上。

那么，比赛的开始和结束行为放在哪里了呢？被放在了另外一个类 MatchService 中：

```

@Entity
@Table(name="teams")
public class Team {
    @Id
    private String name;

    // getters + setters
}

@Embeddable
public class Score {
    @Basic
    private int homeGoals;

    @Basic
    private int awayGoals;

    // getters + setters
}

public interface MatchService {
    void createMatch(UUID matchId, String homeTeamId, String awayTeamId);

    void startMatch(UUID matchId, Date matchDate)
        throws MatchAlreadyStartedException;

    void finishMatch(
        UUID matchId, int homeGoals, int awayGoals, Date finishDate)
        throws MatchFinishedBeforeStartException, MatchAlreadyFinishedException;
}

```

这种强行将业务模型的首肢分离的做法非常类似黑客手法。

打个比喻：人虽然是由母亲生的，但是人的吃喝拉撒母亲不能替代，更不能以母爱名义肢解人的正常职责行为，如果是这样，这个人就是被母爱绑架了。

计算机或数据库技术确实伟大，但是我们不能因为用了计算机，用了数据库，用了框架，业务模型反而被技术框架给绑架。这是畸形不正常的。

提倡充血模型，实际就是让过去被肢解被黑 crack 的业务模型回归正常，当然这也会被一些先入为主或被洗过脑的程序员看成反而不正常，这更是极大可悲之处。看到领域模型代码，就看到业务需求，没有翻译没有转换，保证软件真正实现“拷贝不走样”。

所以，正常的领域模型实体应该是如下，包含比赛等应有的基本职责行为：

The final code

```
public class Match extends AggregateRoot {
    private UUID id;
    private Date matchDate;
    private boolean finished;

    public Match(UUID id, Team homeTeam, Team awayTeam) {
        apply(new MatchCreatedEvent(id, homeTeam.getName(), awayTeam.getName()));
    }

    private void handle(MatchCreatedEvent event) {
        this.id = event.getId();
        this.finished = false;
    }

    public void startMatch(Date matchDate) throws MatchAlreadyStartedException {
        if (this.matchDate != null) {
            throw new MatchAlreadyStartedException();
        }
        apply(new MatchStartedEvent(id, matchDate));
    }

    private void handle(MatchStartedEvent event) {
        this.matchDate = event.getMatchDate();
    }
}
```

DDD 最大的好处是：接触到需求第一步就是考虑领域模型，而不是将其切割成数据和行为，然后数据用数据库实现，行为使用服务实现，最后造成需求的首肢分离。DDD 让你首先考虑的是业务语言，而不是数据。重点不同导致编程世界观不同。

胖模型

上述将属于实体的行为放入实体内部，但是这样将所有行为都放入实体类中会造成胖模型，实际上这些行为只是实体在不同场景下才具有的行为，就如同你在家是儿子，有一些儿子角色行为，但是在单位是程序员角色，有编程行为，把在家里和在单位不同场景具有的行为放在一起很显然冗余。

对象的行为分两种：一种是维护对象内部字段也就是状态一致性或完整性的行为，还有一种是与外部交互调用的行为。

面向对象非常擅长显式表达状态，类 字段和属性这些都是用来定义状态的强大工具。
(banq 注：场景 事件和状态可以认为一个目标模板)。

对象的状态着重于两个方面，一个是编译时期，一个运行时期，在编译时期我们能看到对象的类定义；而在运行时刻我们可以调用对象实例的字段。

对象的行为是围绕本地状态的，不包括那些与外部协调等行为，这些本地行为只围绕对象自己内部状态进行。例如：

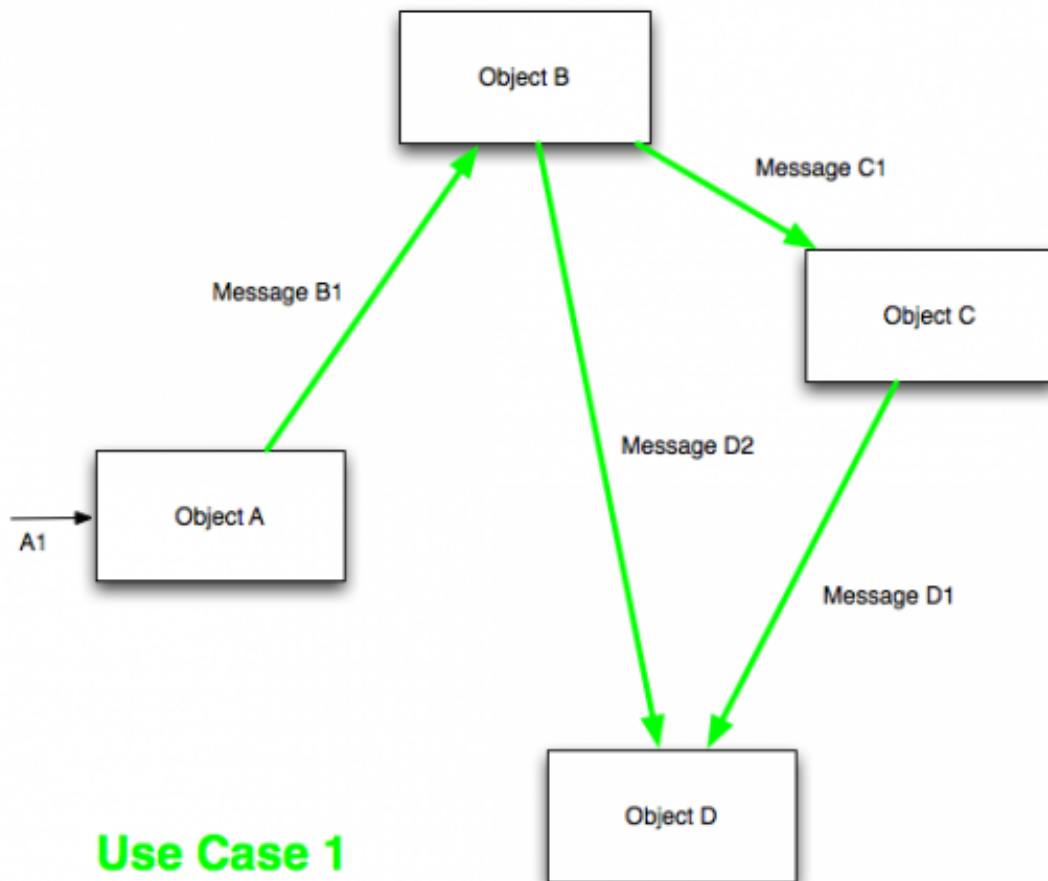
```
public class Entity {
    private int state;
    private boolean show;
    //这个方法维护 state 和 show 之间的逻辑关系，数据一致性。
    public void setState(int state){
```

```

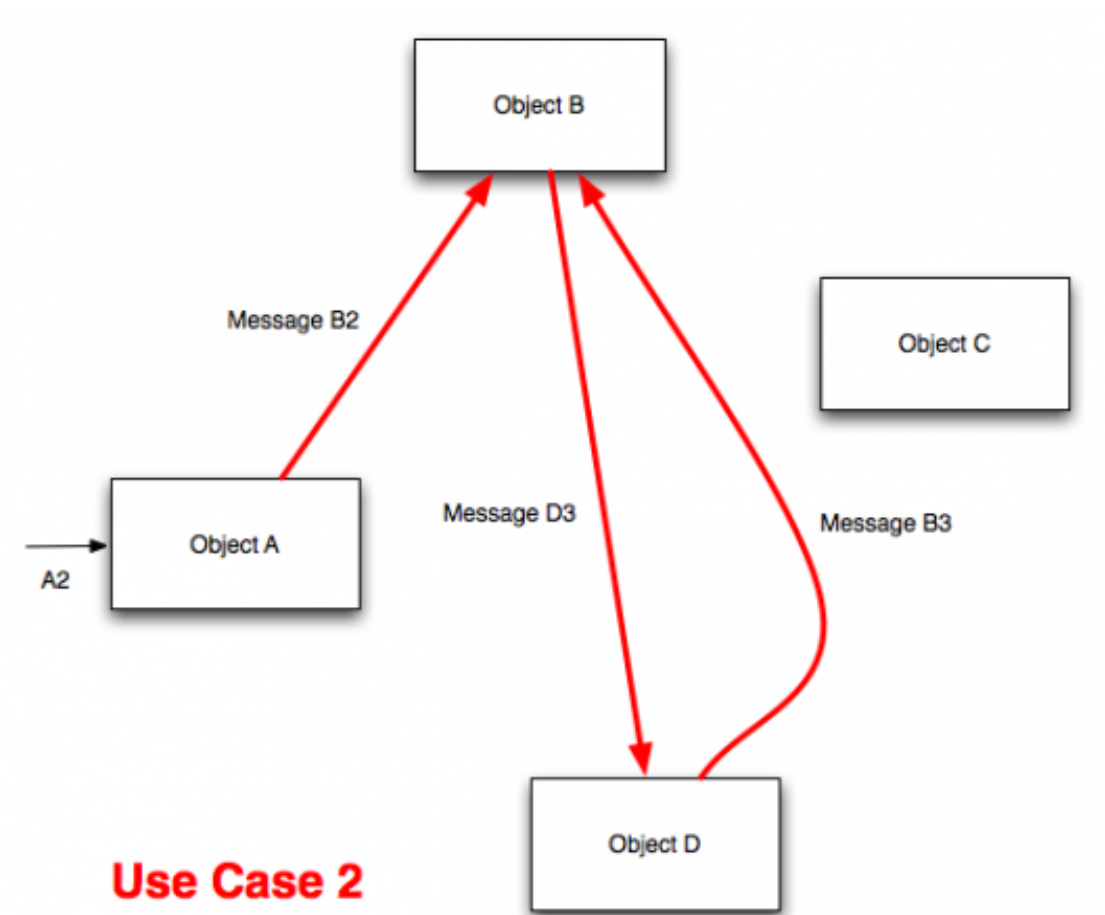
    if (this.state > 1){
        this.state = state;
        show = true;
    }
}
}

```

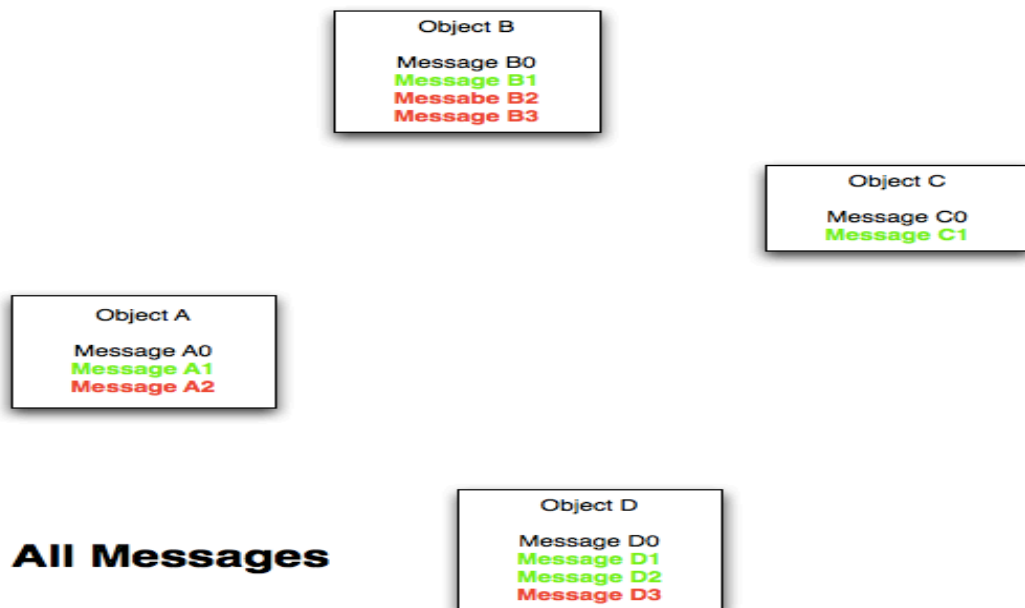
但是 OO 弱点在于无法表达与外部协调交互 collaborations 。看如下两个案例用例(最后两张图)，分别表达 A B C D 四个对象之间的相互调用。第一张图表示在传统 OO 中，我们可能将这些相互协调调用的方法专门当前类中，导致 A B C D 很多方法，这样带来问题看下篇：



在用例场景 2 下，ABCD 之间又有如下行为调用：



那么为完成上面两个用例功能，我们已经知道应该将这些行为放入实体内部，因为他们属于实体的行为，但是又属于实体之间相互调用，设计实体类图如图下



上述方式就是造成一些巨无霸的胖模型。

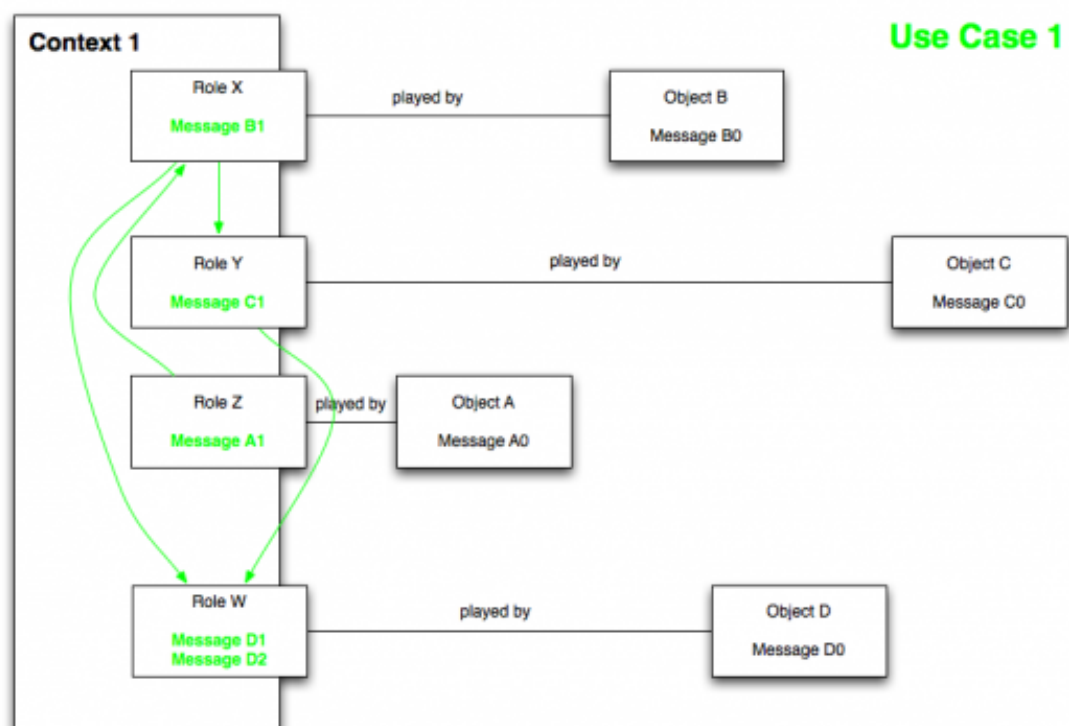
上面将很多交互方法也置于对象中的问题是：编译时刻的代码并不能反映其运行时刻发生的。源码告诉我们有四个分离的对象，分别带有许多方法(messageA0 messageB0....)。而运行时刻告诉我们：有四个对象彼此有许多交互谈话调用，其中只有很少部分和某个特定用例有关(和 User Case1 相关，或和 User Case2 相关)，因为你将所有用例相关的方法都混淆在一个对象中。

这种不匹配导致程序非常难懂，源码告诉我们一个故事，而运行时效果告诉我们是完全不同的故事。

DCI

通过引入 DCI, 数据 Data 上下文 Context 交互 Interaction(DCI):面向对象范式的演进。它是由 MVC 发明者发明的，它提供了一种分析需求的统一语言，认为领域模型在不同场景下扮演不同角色，因而具有不同的角色行为，那些丰富的行为其实是角色的，在运行时，我们将领域模型和角色集合在一起，就如同你进入家门口一刹那，你这个人角色儿子就开始绑定，直至离开家这个场景。

这样，上面 UserCase1 和 UserCase2 不同用例功能可以看成是 Domain Model 领域模型在不同场景上下文扮演不同角色发生的行为，如下图所示：



使用 DCI 的好处：

本地化：

传统 OO 要完成一个算法过程需要跨不同文件，使用 DCI，对于一个特定用例只需要

看一个文件即可。

聚焦：场景上下文只包含有对应用例需要的功能方法，你就不必在数百个方法中寻找工作。

实现“系统是什么”和“系统做什么”分离：

系统是什么：系统是什么样的？意思指所有数据对象(领域模型)和他们本地方法，通常这是系统部分最稳定的。

系统做什么：是指不同场景快速改变的行为，将系统稳定部分和经常变化的部分进行分离。DCI 提供这种分离。

* DCI 的数据类(领域模型)是告诉我们有关系统是什么。

* DCI 场景上下文 Context 告诉我们这些数据类外部相互关系是什么。

角色变得显式：

DCI 最伟大的贡献是带来了显式的角色概念，(角色类似服务或事件，是一种跨业务领域和技术架构的桥梁。)比如我出生在俄罗斯，我的名字是 Victor;我体重 65kg. 这些属性会影响一些高层次职责吗？当我回到家我扮演丈夫角色，我在单位扮演经理角色，等等。

这表明，角色没有在传统 OO 中成为第一等公民是错误的。

逻辑一致性

搞软件只要掌握两个一致性即可：

1.通过 DDD 聚合根掌握业务逻辑上一致性，保证软件实现需求；

2.通过 CAP 定理掌握数据自身的复制一致性，保证软件在技术架构包括分布式系统上准确实现。

领域模型的行为是用来保证一致性的，没有行为保证一致性的数据将是一盘散沙。领域模型的行为分为两种：

1. 保证领域模型内部逻辑一致性的行为，称为基本职责。
2. 与外部进行协调交互的行为，称为场景职责，是实体在一定场景下扮演一定角色而实施的行为。

打个比喻，人作为一个实体对象，有维持自身生命的行为，如果没有这些行为，就不是活人了；这是其基本职责；而人在家里是爸爸，承担父亲的职责；在单位是经理，可以签署文件，这些行为权利都是因为其角色使然，也就是他在单位这个业务场景，扮演的角色需要的职责。

DDD 聚合根思路，先去除不必要的关联依赖，找出高聚合，比如结合业务发现，A 和 B 是代表各自聚合的实体根。切割后分别设计，聚合根实体对象的行为应该是保证对象内部状态一致性的那些动作。

所谓逻辑一致性，也就是业务的规则 约束或校验，以日常例子说明，如果一群人的观点一致，那么我们就可以用 XX 组织 XX 帮派来称呼他们，人以群分，物以类群，人或物因为有内部一致性才归类。

具体来说，类似状态模式，如果当前进入播放状态(假设有开始 播放 暂停 停止四个

状态), 那么下一个状态只可能是暂停或停止状态, 肯定不是开始状态, 那么这种一致性判断在什么时候什么地方判断呢?

很显然应该是在触发状态改变之前的动作行为中判断, 那么这些动作就不能放在领域模型以外了, 这也就是失血模型的根本问题所在。

除此保证内部一致性以外的动作方法可以不用放在领域模型内部, 这些和业务场景有关的交互行为可以在服务中, 也可以使用 DCI 将接口注入领域模型中, 还可以用消息或事件实现。也就是说, 用消息来实现交互, 不管这种交互是由事件引起的, 还是领域模型对技术架构发出的一种命令。

DDD CQRS 和 Event Sourcing 的案例:

以比赛 **Match** 代码为例子:

```
public class Match {

    private String id;

    private Date matchDate;

    private Team teams[] = new Team[2];

    private boolean finished;

    @Inject
    public EventSourcing es;

    //开始比赛方法保证比赛本身的内部一致性
    public void startMatch(Date matchDate) {
        //如果没有比赛时间 比赛是无法开始 这是一致性校验
        if (this.matchDate != null)
            System.err.print("the match has started");
        es.started(new MatchStartedEvent(this.id, matchDate));
    }

    //结束比赛方法也是需要保证比赛内部逻辑一致
    public void finishWithScore(Score score, Date matchDate) {
        if (this.matchDate == null)
            System.err.print("the match has not started");
        //如果比赛已经结束, 再次结束肯定破坏比赛状态
        if (finished)
            System.err.print("the match has finished");
        es.finished(new MatchFinishedEvent(this.id, matchDate, score.getHomeGoals(),
```



```
score.getAwayGoals());
    }

...
}
```

比赛对象的字段有：比赛有开始时间 开始状态 结束时间 结束状态。

这些字段之间有约束规则，形成逻辑一致性，才能真正形成比赛这个概念：

1. 比赛必须有开始时间和结束时间，不能为空。
2. 比赛开始后，只能结束，不能再开始。
3. 比赛结束后，一切都 Over，不能再有结束或开始动作。

在 Match 这个聚合根内部加入这些维持自身一致性的行为，从而保证 Match 状态的逻辑一致性，如果这些行为放在 Match 外部去实现，造成 Match 内部字段数值混乱，没有可控性，这和直接操作数据表没有什么两样。

CQRS 架构

CQRS 是为了更好将 DDD 与现有技术架构结合而推出的一种分离架构。领域专家强调领域模型要充血，不能被技术架构绑架，我们当然应该首先满足，但是数据库成熟强大的查询能力我们也不能走极端弃之不用，因为我们的系统不但要满足领域专家，而且还有用户要求的各种复杂查询功能。

我们进行仔细分析就会发现：领域模型的各种职责行为是靠用户命令触发的，没有用户的命令，这些领域模型的职责不会被激活，就像地雷虽然有爆炸功能，但是没有点爆命令也无法起爆啊。

而用户需要的各种数据组合是一种查询，这些可以依靠数据库查询能力或一些大数据框架来进行并发计算。

这两种路线其实不矛盾，于是 Command 和 Query 分离，领域专家和用户两个需求都能满足，皆大欢喜。至于为什么引入事件，因为 Command 和 Query 是两条路线，command 经常导致状态变化，这种变化要及时反应到 Query 中，通过异步事件将两者同步。

详细讨论见：**DDD CQRS 和 Event Sourcing 的案例：足球比赛**

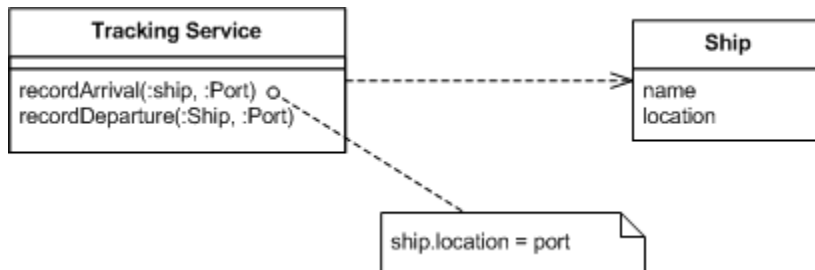
场景 事件与状态

上面 DCI 帮助我们减轻了胖模型可能，那么在具体实施又带来一个问题，领域模型注入到角色中，还是领域模型主动扮演角色呢？如果是前者，这 and 传统 SOA 架构中，使用服务来调用领域模型非常类似，这时领域模型只是被操作，而没有发挥主动的支配地位。

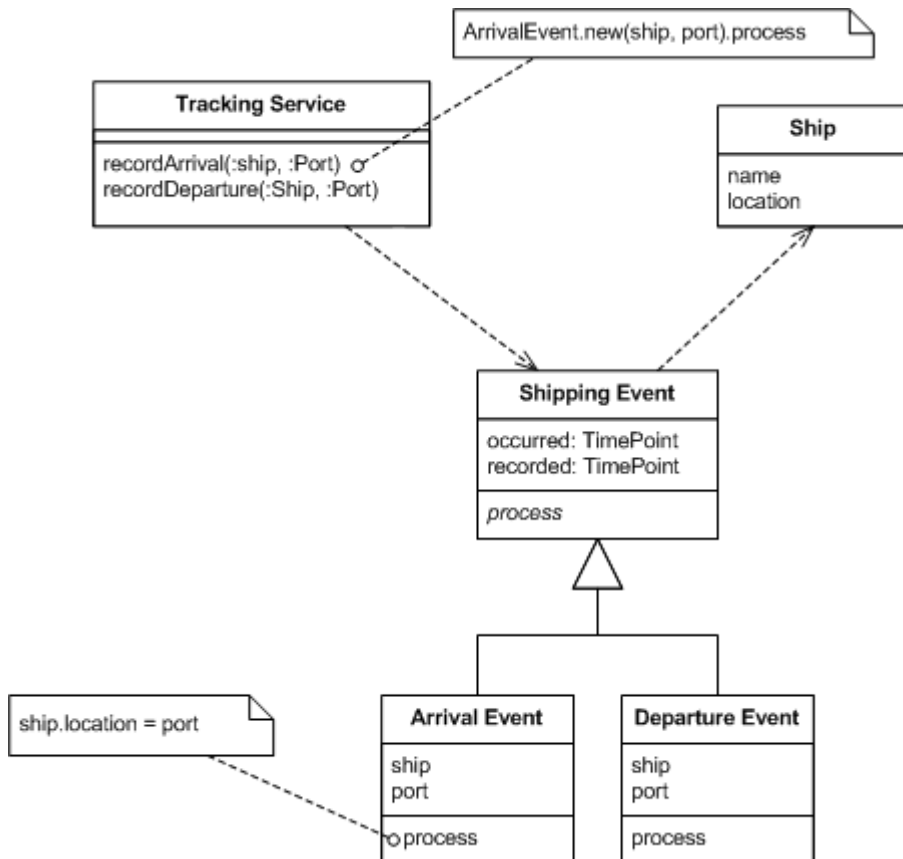
另外从代码可读性上看，要了解一个领域模型扮演了多少角色也不是一件轻松的事情，如果我们在一个领域模型中能看到其所有可能扮演的角色，这样对象阅读将非常方便。

那么领域模型如何发出指令呢？通过事件和消息。

在 MartinFowler 的 Event Sourcing 文章谈到，将导致状态变化的所有事件提取出来以货运为案例：



原来架构是一个服务和一个实体，引入事件后：

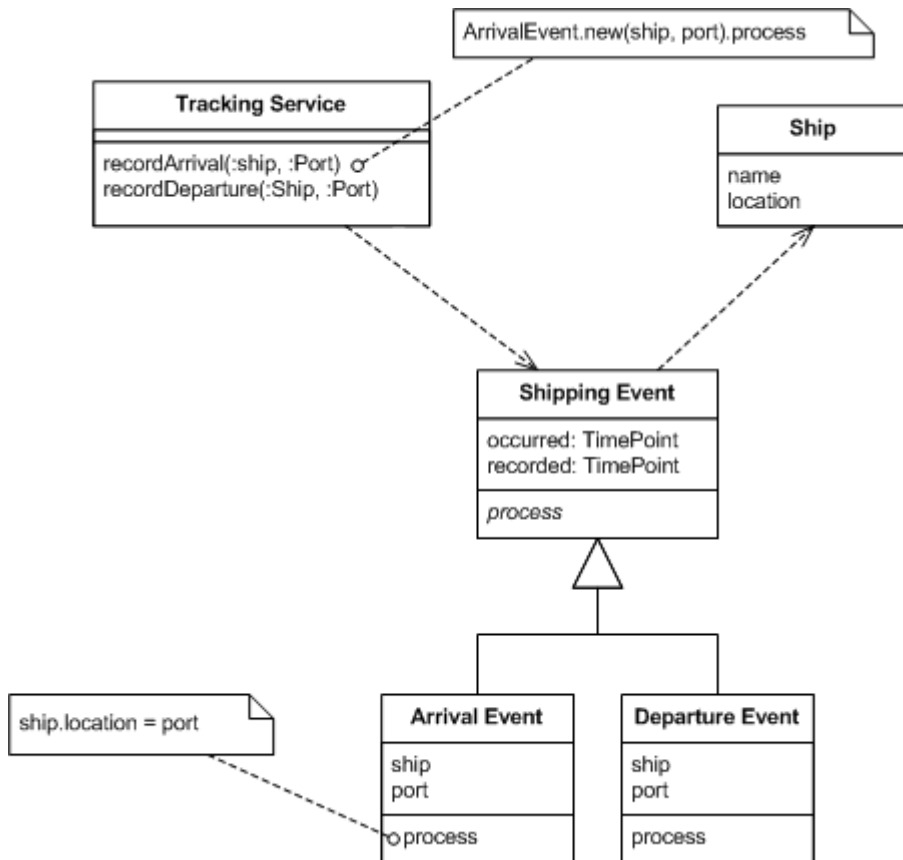


那么正如大部分系统都可以引入服务这个概念，“服务”这个概念既是业务也是技术概念，成为业务和技术架构的桥梁，谈到服务，业务人员知道它在业务上代表什么，技术人员知道如何实现它。

而“事件”也是一种和“服务”类似的介于技术和业务之间桥梁的术语，通过引入事件，应该不会对业务领域包括实体有伤害作用，也不会感觉它是天外来客或神仙姐姐，否则服务这个概念也应该是了。

为什么需要事件？

以货运为案例，如下图：



有些人会误解 ShippingEvent 为实体，事件如果是一个实体，变成事件的记录了(事件变成了状态)，并不能代表正在发生的事件。

而我们根据前面统一语言，设计如下：

```
public Class Ship{

    @Autowired
    ShippingRole shippingRole;

    ....
}
```

Ship 是 data model 也就是领域模型 ,ShippingRole 是角色，由角色 ShippingRole 发出各种装船事件 比如到达 卸货等事件。

有些 DCI 实现是将 Ship 注入到 ShippingRole 中，我个人认为对阅读代码理解不利，你要了解领域模型的所有功能，必须一个个找它被注入到了那些角色场景中，这和 SOA 下被服务类操作使用有什么区别呢？

将领域模型的所有可能扮演的角色陈列在领域模型的，而角色的行为都分离到角色中，这样有助于我们只要打开领域模型类的代码，大概知道这个模型实现哪些功能？

以上是从业务角色看实现的 **ShippingRole** 角色，那么从技术架构角度看，领域模型还有持久化自己等等，按照职责驱动开发方法，无论是业务要求或技术要求，这些都是领域模型应该做的事情，应该承担的职责，难道业务上干的事情在领域模型中，技术要要求干的就不让领域模型干了？

由此，我们从对象职责这个角度综合业务和技术要求，得出领域模型如下：

```
public Class Ship{

    @Autowired
    LazyOperatorRole lazyOperatorRole;

    @ Autowired
    ShippingRole shippingRole;

    .....
}
```

其中 **LazyOperatorRole** 角色负责对实体 **Ship** 的其他次要字段根据需要进行加载，或者根据需要进行数据库持久保存。

其实，这里 **LazyOperatorRole** 和 **ShippingRole** 也是 DDD 中强调的 **Bounded Context**，因为谈到 **Context**，总是相伴角色的，比如 DCI 中谈到 **Context**，隐式地其实围绕角色 **Role**。

我们在实体中通过引入符合一定 **BoundedContext** 的角色，将这些 **BoundedContext** 下发生的行为分离进入角色，由角色通过发送事件给外界，如果不使用事件发送，这些角色行为实现不管是业务行为还是技术行为，总是不可预期的，它们可能是技术和业务混合在一起调用，比如查个数据库，发邮件，关联一下字段等等，这些复杂行为反而会弄脏领域模型，而只有通过事件引入，才能分离领域模型和其生存的环境。

所以，事件的引入已经不只是对业务有好处(业务本身就有事件如 **shipping** 事件)，而且对架构有好处(如存储事件)，更主要保证领域模型不会被太多细节包括技术细节覆盖，突出显示其作为系统核心，其代表用户需求的中心地位。

面向 DDD 的事件驱动架构

为了更好地突出应用需求为主，Takia 框架采取面向 DDD 的主要设计思想，主要特点是常驻内存 **in-memory** 的领域模型向技术架构发送事件消息，驱动技术架构为之服务，如同人体的 **DNS** 是人体各种活动的主要控制者和最高司令，领域模型是一个软件系统的最高司令。

Takia 有五种模型组件，如下：

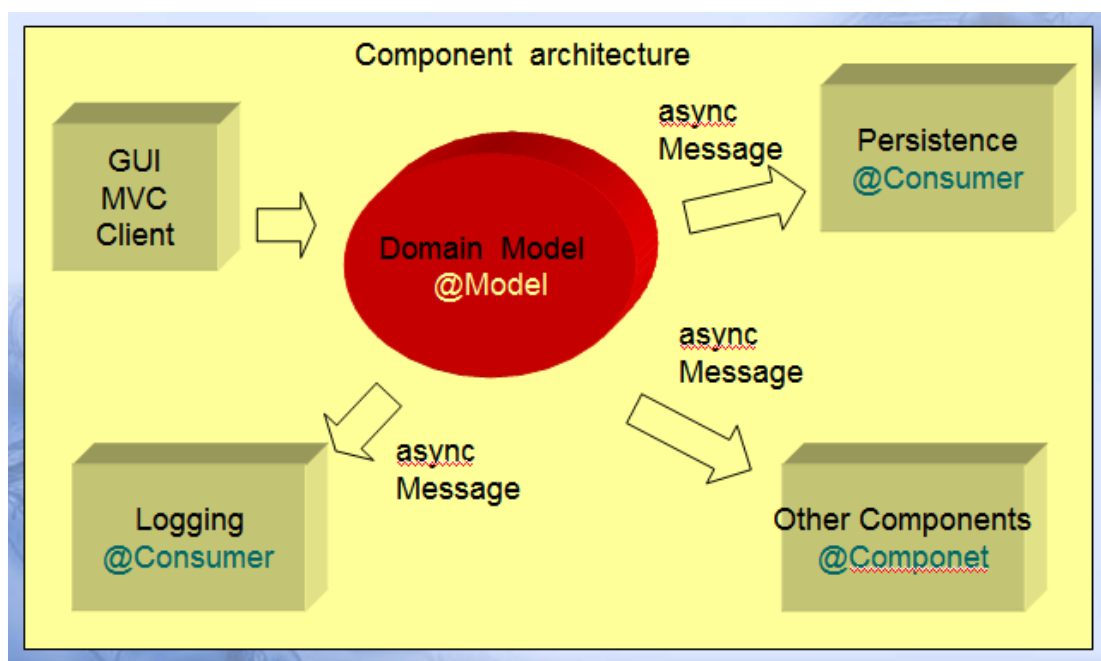
- 一. 实体聚合根对象 元注释 `@Model`;
 - 二. 服务 Service 元注释 `@Service`;
 - 三. 普通类组件构件 `@Component`;
 - 四. 生产者 Producer-消费者模型 `@Send @Consumer`;
 - 五. 拦截器 `@Interceptor`, 首先需要导入点 `@Introduce`;
- 所有都在 `com.reeham.component.ddd.annotation.*`包中。

这些模型组件其实有划分为两大类：业务和技术：

常驻内存`@Model` 领域模型，包括实体模型 值对象和领域服务，与技术架构无关。
相当于鱼；生存空间是缓冲器中

`@Component` 技术组件架构，用以支撑领域模型在计算机系统运行的支撑环境，相当于鱼生活的水。空间在 Context container,例如 `ServletContext` 中。

两者以 Domain Events 模式交互方式：领域模型向技术组件发出异步命令。



实体模型

根据 DDD 方法，需求模型分为实体模型 值对象和领域服务三种，实际需求经常被划分为不同的对象群，如 Cargo 对象群就是 Cargo 为根对象，后面聚合了一批与其联系非常紧密的子对象如值对象等，例如轿车为根对象，而马达则是轿车这个对象群中的一个实体子对象。

在 Takia 框架中，实体根模型通常以`@Model`标识，并且要求有一个唯一的标识主键，你可以看成和数据表的主键类似，它是用来标识这个实体模型为唯一的标志，也可以使用 Java 对象的 `HashCode` 来标识。

Takia 框架是实体模型的主键标识有两个用处：

首先是用来缓存实体模型，目前缓冲器是使用 `EHcache`，可以无缝整合分布式云缓存

Terracotta 来进行伸缩性设计。

只要标识@Model 的实体,在表现层 Jsp 页面再次使用时,将自动直接从缓存中获得,因为在中间业务层和表现层之间 Takia 框架存在一个缓存拦截器 CacheInterceptor,见框架的 Spring 配置文件中配置。

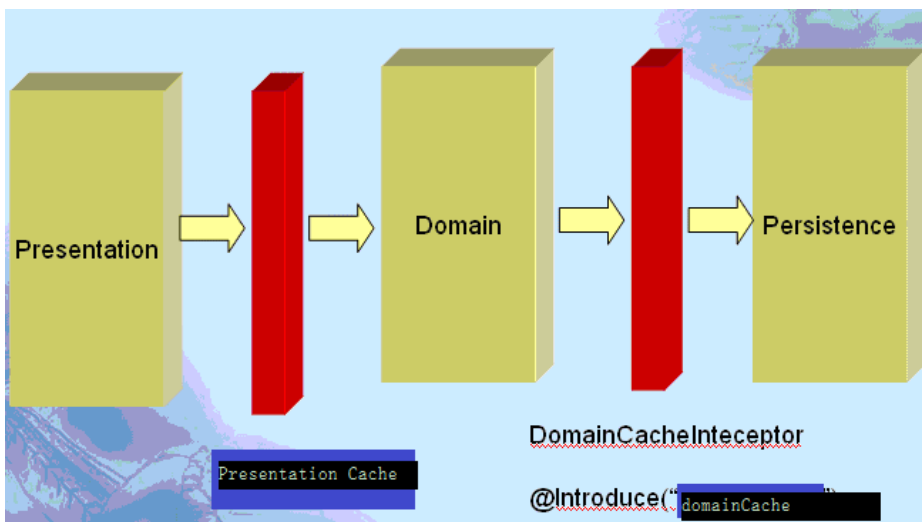
为了能够在业务层能够使用缓存中的模型,需要在业务层后面的持久层或 Repository 中手工加入缓存的 Annotation 标签,如下:

```
@Component("mymrepository")
@Introduce("domainCache")
public class RepositoryImp implements MyModel {

    @Around
    public MyModel getModel(Long key) {
        MyModel mym = new MyModel();
        mym.setId(key);
        return mym;
    }
}
```

Auto cache MyModel

实体模型的缓存架构如下:



注意: 这里可能是 Takia 框架的一个创新或重要点,在 Takia1.0 之前实际上没有对模型进行突出的支持,就象画个圈,圈子外面基本都就绪,圈子里面留白,这个圈子就是 Domain Model,原来因为考虑持久层 Hibernate 等 ORM 巨大影响力,就连 Spring 也是将 Model 委托给 Hibernate 处理,自己不做模型层,但是当 NoSQL 运动蓬勃发展,DDD 深入理解,则找到一个方式可以介入模型层,同时又不影响任一持久层框架的接入。

Takia 框架通过在持久层的获得模型对象方法上加注释的办法,既将模型引入了内存缓存,又实现了向这个模型注射必要的领域事件 Domain Events。

事件 Event Sourcing

Takia 的最大特点是领域模型驱动技术架构;

如果说普通编程缺省是顺序运行,那么事件模型缺省是并行运行,两者有基本思路的不同。

Event Sourcing 适合将复杂业务领域和复杂技术架构实现分离的不二之选。实现业务和技术的松耦合，业务逻辑能够与技术架构解耦，将”做什么”和”怎么做”分离

事件模型也是一种 EDA 架构 Event-driven Architecture，可以实现异步懒惰加载 Asynchronous Lazy-load 类似函数式语言的懒功能，只有使用时才真正执行。

具有良好的可伸缩性和可扩展性，通过与 JMS 等消息系统结合，可以在多核或多个服务器之间弹性扩展。

事件模型也是适合 DDD 落地的最佳解决方案之一。领域模型是充血模型，类似人类的 DNA，是各种重要事件导向的开关。用户触发事件，事件直接激活领域模型的方法函数，再通过异步事件驱动技术活动，如存储数据库或校验信用卡有效性等。

Takia 框架 1.0 实现了 Domain Model + In-memory + Events.，2001 年 Martin fowler 在其文章 LMAX 架构 推荐 In-memory + Event Sourcing 架构。以下是该文的精选摘要，从一个方面详细说明了事件模型的必要性：

内存中的领域模型处理业务逻辑，产生输出事件，整个操作都是在内存中，没有数据库或其他持久存储。将所有数据驻留在内存中有两个重要好处：首先是快，没有 IO，也没有事务，其次是简化编程，没有对象/关系数据库的映射，所有代码都是使用 Java 对象模型。

使用基于内存的模型有一个重要问题：万一崩溃怎么办？电源掉电也是可能发生的，“事件”(Event Sourcing)概念是问题解决的核心，业务逻辑处理器的状态是由输入事件驱动的，只要这些输入事件被持久化保存起来，你就总是能够在崩溃情况下，根据事件重演重新获得当前状态。

事件方式是有价值的因为它允许处理器可以完全在内存中运行，但它有另一种用于诊断相当大的优势：如果出现一些意想不到的行为，事件副本们能够让他们在开发环境重放生产环境的事件，这就容易使他们能够研究和发现出在生产环境到底发生了什么事。

这种诊断能力延伸到业务诊断。有一些企业的任务，如在风险管理，需要大量的计算，但是不处理订单。一个例子是根据其目前的交易头寸的风险状况排名前 20 位客户名单，他们就可以切分到复制好的领域模型中进行计算，而不是在生产环境中正在运行的领域模型，不同性质的领域模型保存在不同机器的内存中，彼此不影响。

LMAX 团队同时还推出了开源并发框架 Disruptor，他们通过测试发现通常的并发模型如 Actor 模型是有瓶颈的，所以他们采取 Disruptor 这样无锁框架，采取了配合多核 CPU 高速缓冲策略，而其他策略包括 JDK 一些带锁都是有性能陷阱的: JVM 伪共享。

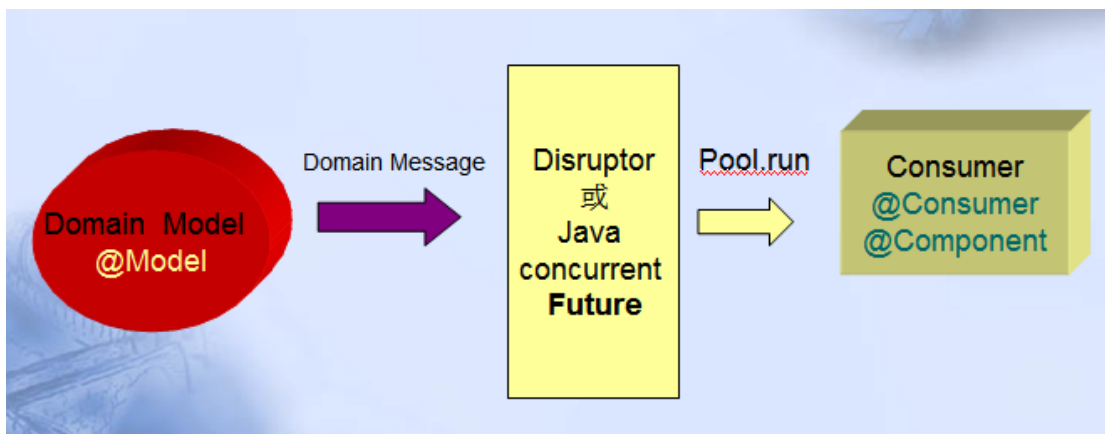
Takia 的领域事件是基于号称最快的并发框架 Disruptor 开发的，因此 Takia 的事件是并行并发模型，不同于普通编程是同一个线程内的顺序执行模型。

Takia 的领域事件是一种异步模式 + 生产者-消费者模式。主题 topic 和 Queue 队列两种。领域模型是生产者；消费者有两种：

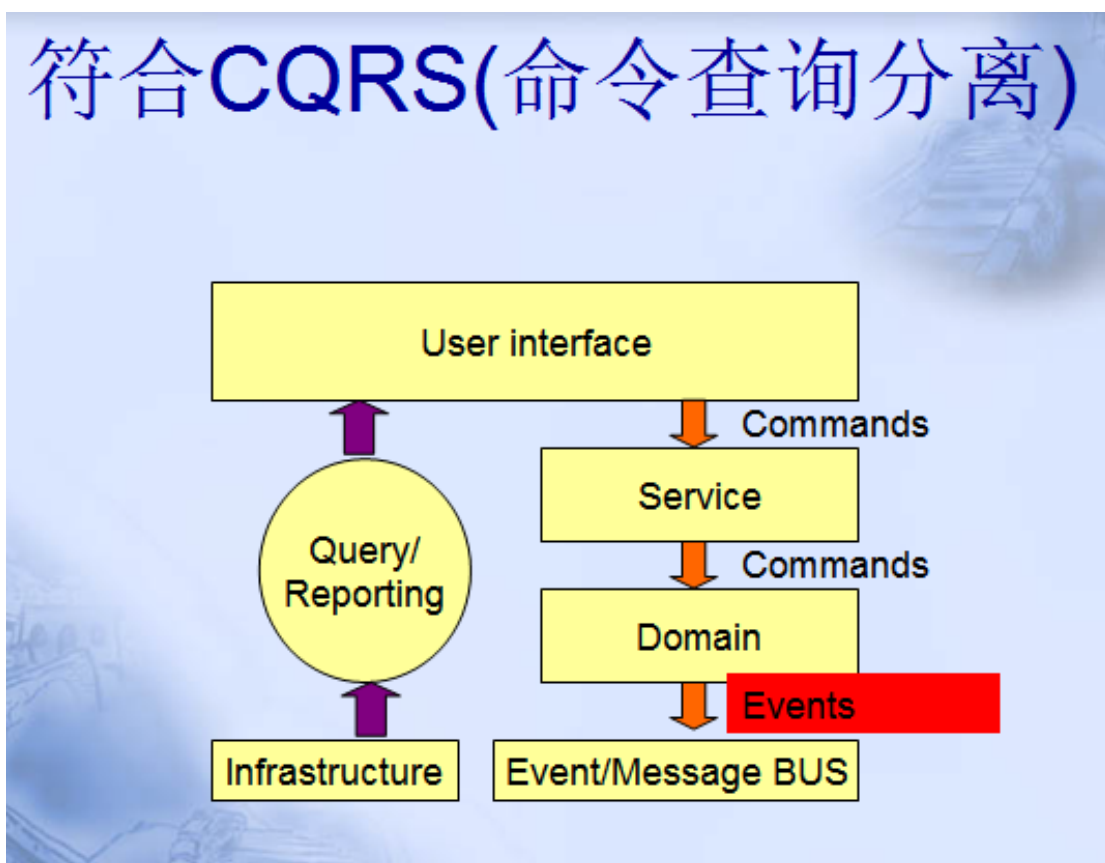
.@Consumer; 可以实现 1:N 多个，内部机制使用号称最快的并发框架 Disruptor 实现。适合轻量；小任务；原子性；无状态。

.@Componet; 直接使用普通组件类作为消费者，使用 jdk future 机制，只能 1:1，适合大而繁重的任务，有状态，单例。

Domain Events 实现机制如下：



Takia 的事件模型还是一种 CQRS 架构，可以实现模型的修改和查询分离，也就是读写分离的架构：



无堵塞的并发编程

顺序编程和并发编程是两种完全不同的编程思路，堵塞 Block 是顺序编程的家常便饭，常常隐含在顺序过程式编程中难以发现，最后，成为杀死系统的罪魁祸首；但是在并发编程中，堵塞却成为一个目标非常暴露的敌人，堵塞成为并发不可调和绝对一号公敌。

因为无堵塞所以快，这已经成为并发的一个基本特征。

过去我们都习惯了在一个线程下的顺序编程，比如，我们写一个 Jsp(PHP 或 ASP)实

际都是在一个线程

下运行，以 google 的 adsense.Jsp 为例子：

```
<%  
//1.获得当前时间  
long googleDt = System.currentTimeMillis();  
//2.创建一个字符串变量  
StringBuilder googleAdUrlStr = new StringBuilder(PAGEAD);  
//3.将当前时间加入到字符串中  
googleAdUrlStr.append("&dt=").append(googleDt);  
//4.以字符串形成一个 URL  
URL googleAdUrl = new URL(googleAdUrlStr.toString());  
%>
```

以上 JSP 中 4 步语句实际是在靠一个线程依次顺序执行的，如果这四步中有一步执行得比较慢，也就是我们所称的是堵塞，那么无疑会影响四步的最后执行时间，这就象乌龟和兔子过独木桥，整体效能将被跑得慢的乌龟降低了。

过去由于是一个 CPU 处理指令，使得顺序编程成为一种被迫的自然方式，以至于我们已经习惯了顺序运行的思维；但是如今是双核或多核时代，我们为什么不能让两个 CPU 或多个 CPU 同时做事呢？

如果两个 CPU 同时运行上面代码会有什么结果？首先，我们要考虑两个 CPU 是否能够同时运行这段逻辑呢？

考虑到第三步是依赖于前面两步，而第二步是不依赖第一步的，因此，第一步和第二步可以交给两个 CPU 同时去执行，然后在第三步这里堵塞等待，将前面两步运行的结果在此组装。

很显然，这四步中由于第三步的堵塞等待，使得两个 CPU 并行计算汇聚到这一步又变成了瓶颈，从而并不能充分完全发挥两个 CPU 并行计算的性能。

我们把这段 JSP 的第三步代码堵塞等待看成是因为业务功能必须要求的顺序编程，无论前面你们如何分开跑得快，到这里就必须合拢一个个过独木桥了。

但是，在实际技术架构中，我们经常也会因为非业务原因设置人为设置各种堵塞等待，这样的堵塞就成为并行的敌人了，比如

我们经常有(特别是 Socket 读取)

```
While(true){  
    .....  
}
```

这样的死循环，无疑这种无限循环是一种堵塞，非常耗费 CPU，它无疑成为并行的敌人。

比如 JDK 中 `java.concurrent. BlockingQueue` `LinkedBlockingQueue`，也是一种堵塞式的所谓并行包，这些并行功能必须有堵塞存在的前提下才能完成并行运行，很显然是一种伪并行。

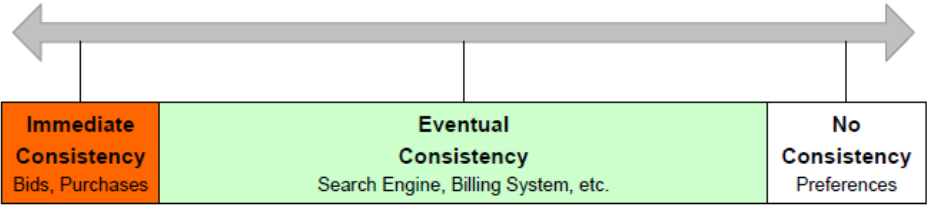
由于各种技术上的堵塞存在，包括多线程中锁机制，也是一种堵塞，因为锁机制在某个时刻只允许一个线程进行修改操作，所以，并发框架 `Disruptor` 可以自豪地说：无锁，所以很快。

现在非常流行事件编程模型，如 `Event Sourcing` 或 `Domain Events` 或 `Actor` 等等，事

件编程实际是一种无堵塞的并行编程，因为事件这个词语本身有业务模型上概念，也有技术平台上的一个规范，谈到事件，领域专家明白如同电话铃事件发生就要接，程序员也能明白只要有事件，CPU 就要立即处理(特别是紧急事件)，而且事件发生在业务上可能是随机的，因此事件之间难以形成互相依赖，这就不会强迫技术上发生前面 Jsp 页面的第三步堵塞等待。

因此，在事件模型下，缺省编程思维习惯是并发并行的，如果说过去我们缺省的是进行一个线程内的顺序编程，那么现在我们是多线程无锁无堵塞的并发编程，这种习惯的改变本身也是一种思维方式的改变。

在缺省大多数是并发编程的情况下，我们能将业务上需要的顺序执行作为一种特例认真小心对待，不要让其象癌细胞一样扩散。我们发现这种业务上的顺序通常表现为一种高一一致性追求，更严格的一种事务性，每一步依赖上一步，下一步失败，必须将上一步回滚，这种方式是多核 CPU 克星，也是分布式并行计算的死穴。值得庆幸的是这种高一一致性的顺序编程在大部分系统中只占据很小一部分，下图是电子商务 EBay 将他们的高一致性局限在小部分示意图：



由此可见，过去我们实现的顺序编程，实际上是我们把一种很小众的编程方式进行大规模推广，甚至作为缺省的编程模式，结果导致 CPU 闲置，吞吐量上不去同时，CPU 负载也上不去，CPU 出工不出力，如同过去计划经济时代的人员生产效率。

据统计，在一个堵塞或有锁的系统中，等待线程将会闲置，这会消耗很多系统资源。消耗资源的公式如下：

$$\text{闲置的线程数} = (\text{arrival_rate} * \text{processing_time})$$

如果 arrival_rate(访问量达)很高，闲置线程数将会很大。堵塞系统是在一个没有效率的方式下运行，无法通过提高 CPU 负载获得高吞吐量。

避免锁或同步锁有多种方式，volatile 是一种方式，主要的不变性设计，比如设计成 DDD 的值对象那种，要变就整个更换，就不存在对值对象内部共享操作；还有克隆原型，比如模型对象克隆自己分别传给并发的多个事件处理；用 Visibility 使资料对所有线程可见等等；

最彻底的方式就是使用无锁 Queue 队列的事件或消息模型，Disruptor 采取的是一个类似左轮手枪圆环 RingBuffer 设计，这样既能在两个线程之间共享状态，又实现了无锁，比较巧妙，业界引起震动。

当然 Scala 的那种 Share nothing 的 Actor 模型也是一种无锁并发模型，不过奇怪的是，发明 Disruptor 的 LMAX 团队首先使用过 Actor 模型，但是并发测试发现还是有性能瓶颈的，所以，他们才搞了一个 Disruptor。

领域模型发出事件，事件由 Disruptor 的 RingBuffer 传递给另外一个线程(事件消费者)；实现两个线程之间数据传递和共享。

当事件消费者线程处理完毕，将结果放入另外一个 RingBuffer，供原始线程(事件生产者)读取或堵塞读取，是否堵塞取决于是否需要顺序了。

非堵塞并发使用模式如下：

发出调用以后不必立即使用结果，直至以后需要时才使用它。发出调用后不要堵塞在原地等待处理结果，而是去做其他事情，直至你需要使用这个处理结果时才去获取结果。

被调用者将会返回一个“future”句柄，调用者能够获得这个句柄后做其他事情，其他事情处理完毕再从这个句柄中获得被调用者的处理结果。

Takia 为推广适合多核 CPU 的无堵塞并发编程范式进行了探索，使用了 Domain Events 和 DCI 等不同抽象层次对并发编程进行了封装，从而降低开发者使用并发编程的难度。

DCI 实现代码

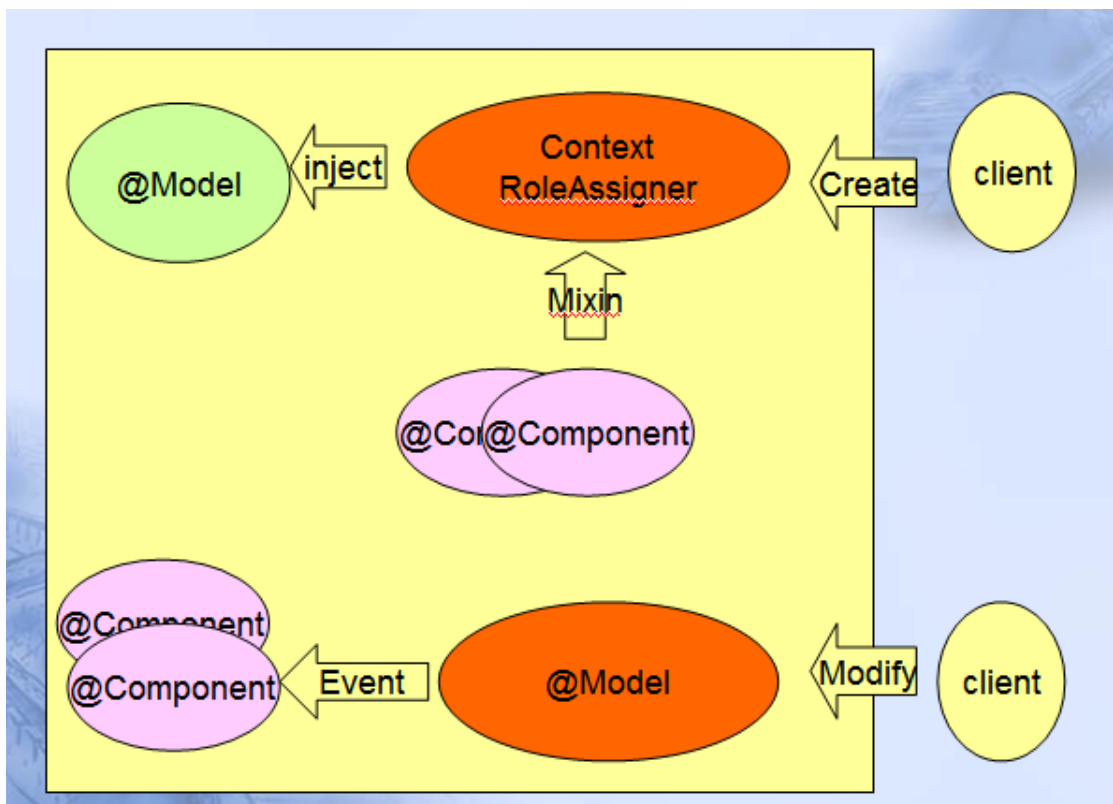
DCI：数据 Data，上下文(场景)Context，交互 Interactions 是由 MVC 发明者 [Trygve Reenskaug](#) 发明的。其核心思想是：

让我们的核心模型更加简单，只有数据和基本行为。业务逻辑等交互行为在角色模型中在运行时的场景，将角色的交互行为注射到数据中。

Takia 框架提供了两种 DCI 风格实现：一种是将 DomainEvents 对象注射进入当前模型；还有一种是直接将数据模型和接口混合。这两种方式适合不同场景。

如果我们已经 Hold 住了一个领域对象，那么就直接通过其发出领域事件实现功能；比如模型的修改。在这种模式下，发出领域事件的领域模型本身已经隐含了场景。事件代表场景出头牵线。

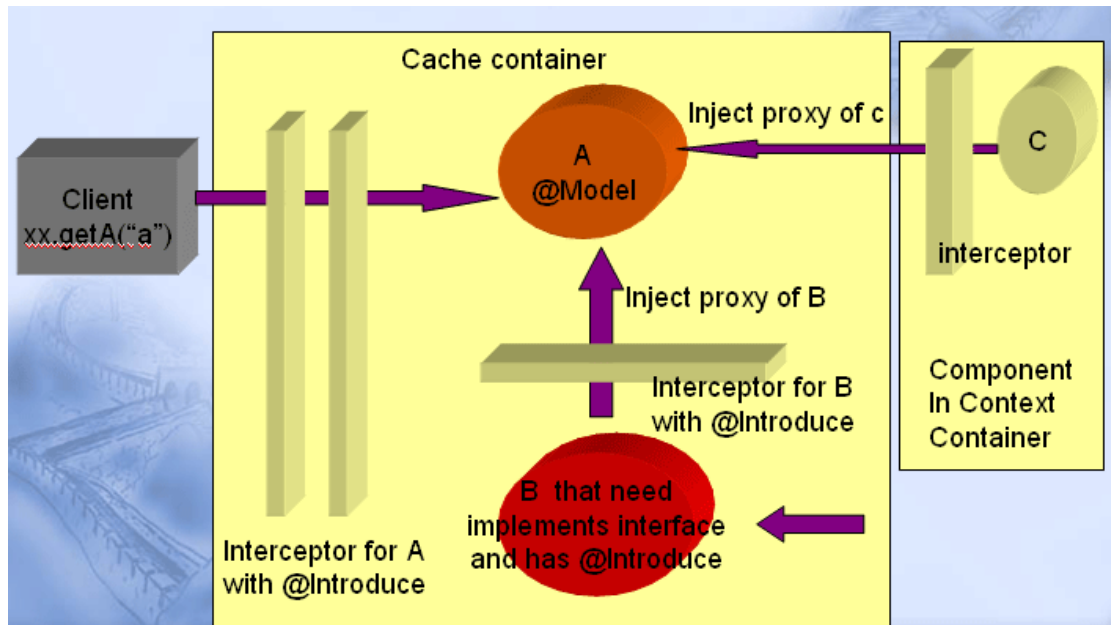
否则，我们显式创建一个上下文 Context，在其中通过 RoleAssigner 将角色接口注入到领域对象中。比如模型新增创建或删除。(对象不能自己创建自己)。



具体实现可见下面专门 DCI 案例章节。

依赖注入 DI 实现代码

(一) @Model: 模型中可以通过字段的@Inject 将其他类注射进入, 包括@Component 类。被注射的类如果有@Introduce, 将再次引入拦截器。



```
@Model
public class MyModel {

    private Long id;
    private String name;

    @Inject
    private MyModelDomainEvent myModelDomainEvent;

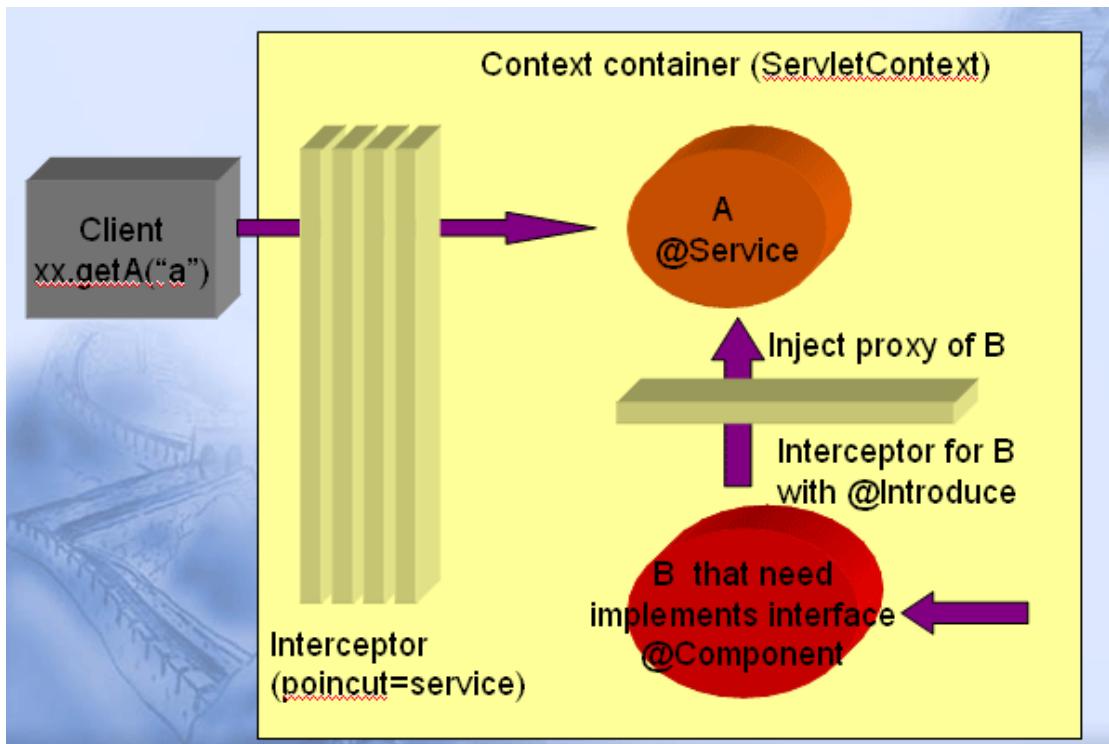
    @Inject
    private MyModelService myModelServiceCommand;

    @Introduce("message")
    public class MyModelDomainEvent {

        @Send("MyModel.findName")
        public ModelMessage asyncFindName(MyModel myModel) {
            return new ModelMessage(myModel);
        }
    }
}

// invoke
MyModel --> @Introduce(message) --> MyModelDomainEvent
```

(二) @Component: 技术架构中组件可以通过构造器直接注射, 被注射的类如果有@Introduce, 将再次引入拦截器。



AOP 拦截器

Takia 框架拦截器基于 Spring 实现，主要由 `MessageInterceptor` 和 `DomainCacheInterceptor` 两个实现。

框架内部组件机制

Takia 内部组件都是基于 Spring 管理的标准组件，请参考 Spring 相关文档。

事件模型实现

本章主要介绍如何使用 Takia 框架实现领域事件 `Domain Events`，从这些事件实现中可以发现事件模型的优点和特点。

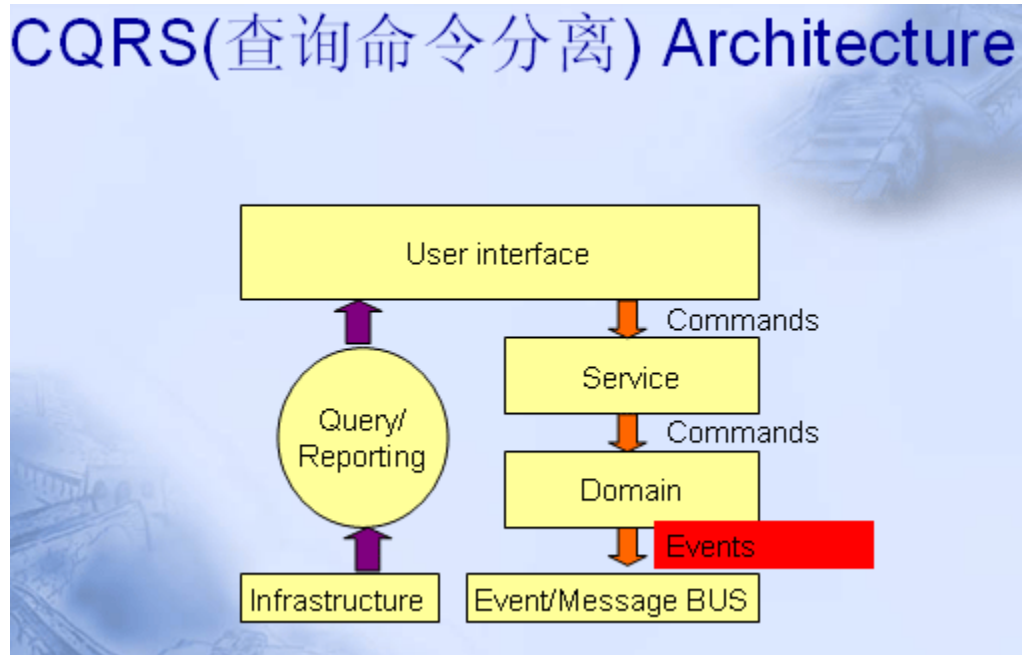
Domain Events 开发

在 Evans DDD 实现过程中，经常会碰到实体和服务 `Service` 以及 `Repository` 交互过程，这个交互过程的实现是一个难点，也是容易造成贫血模型的主要途径。

领域模型中只有业务，没有计算机软件架构和技术。不要将和技术相关的服务和功能组件注射到实体模型中，例如数据库 `Dao` 等操作。由领域模型通过 `Domain Events` 机制指挥 `Domain` 服务和其他功能组件，包括数据库操作。

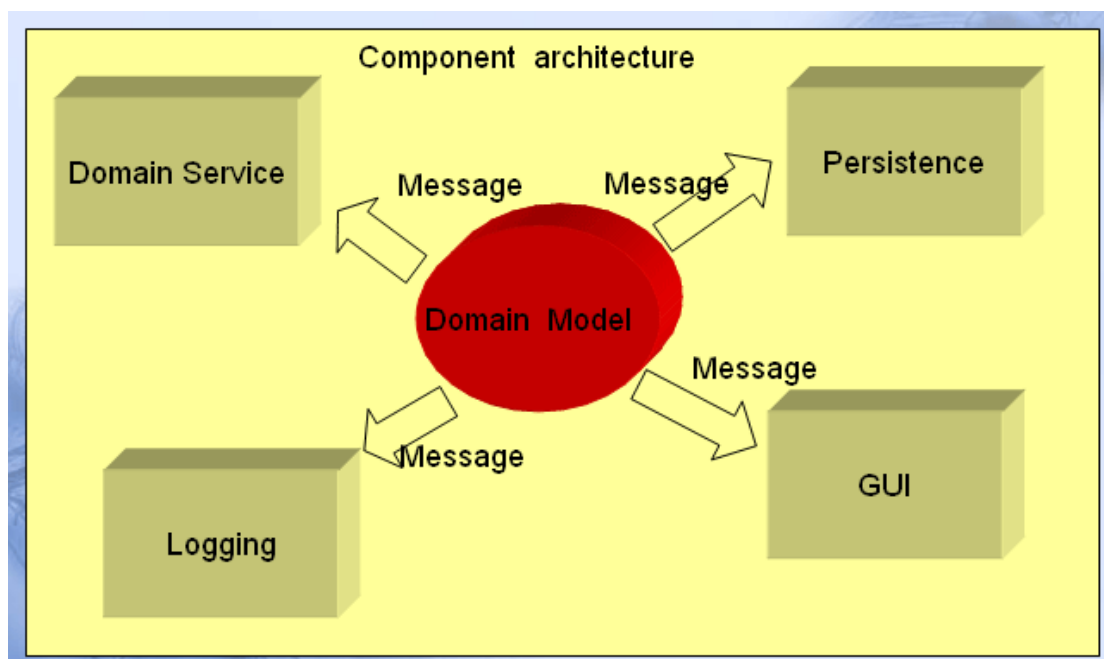
<http://jonathan-oliver.blogspot.com/search/label/DDD>

使用 DDD，常常需要命令查询分离模式 CQS 架构，如下图：



图中 Domain 发出的 Events 就称为 Domain Events, 如果说 CQS 架构提出将 Command 命令和查询分离的模式，那么随着 Command 命令产生 Domain Events 则提出业务模型和技术架构分离的解决方案。

Takia 提供的异步观察者模式为 Domain Event 实现提供更优雅的方案。



让模型常驻内存

在开始 DomainEvents 开始之前，必须保证领域模型在内存中，分两步：

1. 使用@Model 标注你的领域模型，如 User 是从需求中分析出的一个领域模型类：

@Model

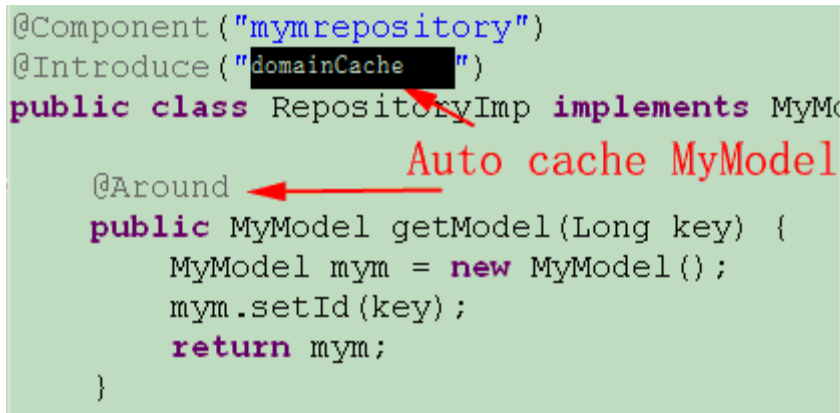
```
public class User {  
    private String userId;  
    private String name;  
    ....  
}
```

源码见：

当领域模型类使用@Model 标注后，意味着这个模型将会驻留在内存中 in memory, 缺省 Takia 框架支持的是 Ehcache. 你也可以设置为 Key-value store 或其他内存数据网格 IMDG。

2. 虽然第一步标注了，但是模型不会自动驻留内存，还需要你在自己的应用代码中显式表明一下，方法是：在仓储 Respository 层，也就是封装了数据库 SQL 操作的层面，因为我们的领域模型一般都是从数据库获得，在仓储层进行组装，将一个数据表数据转换成领域对象，那么在这个转换性质的类和方法上要加一个元注释@Introduce(“domainCache”)和@Around。

这一步确保领域对象每次加载在内存都是唯一的，In-memory。这对于后面使用 domain events 是必须关键重要步骤。



```
@Component("mymrepository")  
@Introduce("domainCache")  
public class RepositoryImp implements MyModel {  
    @Around  
    public MyModel getModel(Long key) {  
        MyModel mym = new MyModel();  
        mym.setId(key);  
        return mym;  
    }  
}
```

Auto cache MyModel

下面开始 Domain Events 开发步骤，分两步：

Domain Events 自 2.0 版本以后，有两种使用方式，一种是 1.0 版本以前的使用 JDK 并发包 futuretask 实现的方式；一种是 2.0 版本以后新增引入最快并发框架 Disruptor 的 @Consuner 方式。首先谈一下这种新的方式：

消息生产者开发

第一步：消息发送者的开发，与 2.0 版本以前类似：

首先：使用 @Model 和 @Introduce(“message”)标注实体类。

然后：使用 @Send(“mytopic”) 标注该实体中的发送方法。

如下：


```

@Model
@Introduce("message")
public class DomainEvent {
    private Long id;
    private String name;
    @Send("mychannel")
    public DomainMessage myMethod() {
        DomainMessage em =
            new DomainMessage(this.name);
        return em;
    }
}

```

注意点: @Introduce(“message”)中“message”值表示引入 Takia 配置 Spring 中消息拦截器: com.reeham.component.ddd.message.MessageInterceptor, 通过该配置让该模型类成为消息生产者或发送者。

@ Send(“mytopic”):中的“mytopic”是消息 topic 名称, 可自己取, 但是和消费者的标注 @Consumer(“mytopic”)或@Componet(“mytopic”)是一致的, 表示生产者将消息发往这个 topic 中;

在@send 标注的方法中, 你还需要将你要传送给消息消费者使用的数据打包进入 DomainMessag 对象, 该方法的返回类型必须是 DomainMessag.

消息消费者开发

第二步: 消息消费者开发, 分两种, @Consumer 和@Component

@Consumer; 可以实现 1:N 多个, 内部机制使用号称最快的并发框架 Disruptor 实现。适合轻量; 小任务; 原子性; 无状态。

@Componet; 直接使用普通组件类作为消费者, 使用 jdk future 机制, 只能 1:1, 适合大而繁重的任务, 有状态, 单例。

@Consumer 和@Component 大部分情况下可互换使用。

@Consumer 方式:

标注消费者 @Consumer("mytopic"); 消费者类必须实现接口 com.reeham.component.ddd.message.DomainEventHandler。

```

@Consumer("mychannel")
public class MyDomainEventHandler implements DomainEventHandler
{
    public void onEvent(EventDisruptor event, boolean endOfBatch)
        throws Exception {
        System.out.println("DomainEventHandler action " +
            event.getDomainMessage().getEventSource());
    }
}

```

如果有多个消费者订阅了同一个主题 Topic, 那么这些消费者之间的运行顺序是按照

他们的类名字母先后的。如 AEventHandler 先于 BEventHandler 先于 CEvent...等。

消费者的主要内容写在 onEvent 中，可通过 event.getDomainMessage()获得生产者你要传递的数据。

以上两个步骤的领域事件@Consumer 消费者开发完成。

返回事件处理结果

如果要返回事件的处理结果，在 DomainEventHandler 的 onEvent 方法 将结果设置如 event 的 DomainMessage 中，如下：

```
void onEvent(EventDisruptor event, boolean endOfBatch) throws Exception {  
    //返回结果    “eventMessage=hello”  
    event.getDomainMessage().setEventResult("eventMessage=" + myModel.getId());  
}
```

Takia 客户端可以是在 Web 的 Servlet 或 Jsp 中调用，也可以在 Application 调用，调用以上领域事件代码如下：

```
IServiceSample serviceSample = (IServiceSample) beanFactory.getBean("serviceSample");  
//返回结果    “eventMessage=hello”  
String res = (String) serviceSample.eventPointEntry("hello");  
Assert.assertEquals(res, "eventMessage=hello");
```

@Component 消费者

下面谈一下 1.0 以前版本但是 2.0 以后延续使用的 1:1 队列@Component 方式的开发，@Component 方式适合大任务，繁重计算，不必返回结果，是单例，而@Consumer 非线程方式：每次启动都启动新的实例线程。

1. 创建模型类如 UserModel(需要@Model 标注)，在 UserModel 引入自己的 UserDomainEvents

@Autowired

private UserDomainEvents userDomainEvents;

2. 创建 UserDomainEvents 类

3. 创建 com.reeham.component.ddd.message.MessageListener 实现子类

如下图所示步骤：

```

@Model
public class UserModel {

    private String userId;
    private String name;

    private UserCountValueObject ageVO;

    @Inject
    private UserDomainEvents userDomainEvents;

    public void update(UserModel userParameter) {
        this.name = userParameter.getName();

        ageVO.preloadData();
        userDomainEvents.save(this);
    }
}

@Introduce("message")
public class UserDomainEvents {

    @Send("saveUser")
    public DomainMessage save(UserModel user) {
        return new DomainMessage(user);
    }
}

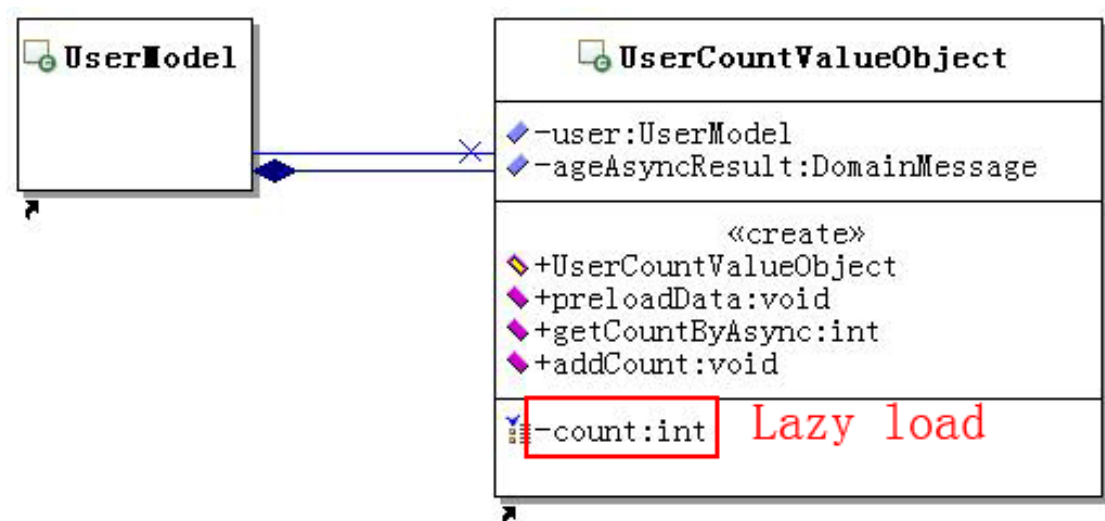
@Component("computeCount")
public class ComputeCountListener implements MessageListener {

    public void action(DomainMessage message) {
        UserModel user = (UserModel) message.getEvent();
        int age = userRepository.getAge(user.getUserId());
    }
}

```

使用 Domain Events 可实现异步懒加载机制，对模型中任何字段值根据需从数据库加载，即用即取，这种方式有别于 Hibernate 的惰加载 lazy load 机制，更加灵活。

如下图，我们为了实现 UserCount 值对象中 count 数据的懒加载，将其封装到一个值对象中，这个值对象其他字段和方法都是为懒加载服务的。



我们看看 getCountByAsync 方法内部：

```

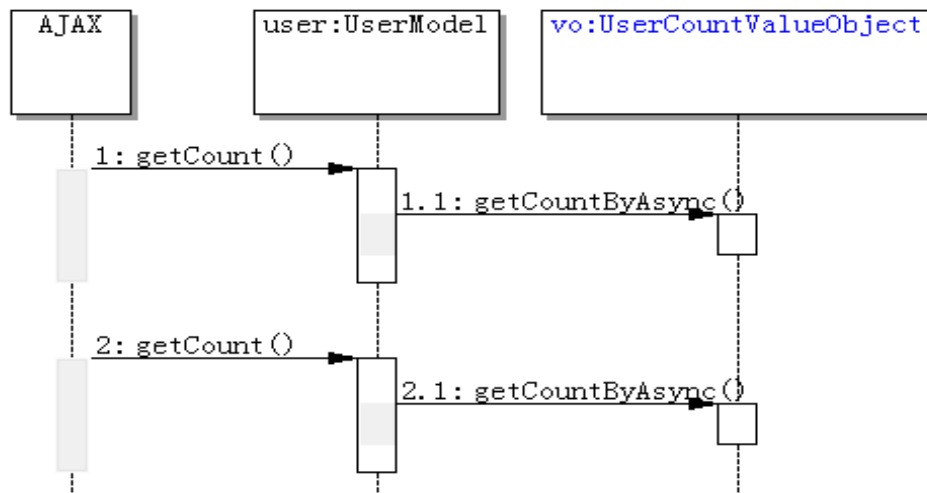
public int getCountByAsync() {
    if (count == -1) { // lazy load
        if (ageAsyncResult == null)
            ageAsyncResult = user.getUserDomainEvents().computeCount(user);
        else
            count = (Integer) ageAsyncResult.getEventResult();
    }
    return count;
}

```

Count 初始值是-1，如果该方法第一次调用，count 必为-1，那么 ageAsyncResult 也应该为空，这时就激活 computeCount 这个耗时耗 CPU 的任务，当第二次访问 getCountByAsync 时，ageAsyncResult 应该不为空，因为耗时任务应该计算完毕，如果还没有，会在这里等待。一旦获得计算结果，count 就不会为-1。

UserCount 值对象随同 UserModel 生活在内存中，因此，这样耗时任务计算只需要一次，以后，基本直接从对象的字段 count 直接获得。

调用顺序图如下：



懒加载 Laziness

Account 中有一个计算该用户发帖总数字段 messageCount:

```

public class Account{

    private int messageCount;

    public int getMessageCount(){
        return messageCount;
    }

}

```

这个 messageCount 是通过查询该用户所有发帖数查询出来的，也许你会问，为什么不将用户发帖总数设置为数据表一个字段，这样用户发帖时，更新这个字段，这样只要直接查询这个字段就可以得到 messageCount？

没有设立专门持久字段的原因如下：

1. 从模型设计角度看：messageCount 字段归属问题，messageCount 其实是一个表示 Account 和 Message 两个实体的关系关联统计字段，但是这种关联关系不属于高聚合那种组合关系的关联，不是 Account 和 Message 的必备属性，根据 DDD 的高聚合低关联原则，能不要的关联就不要，因此放在哪里都不合适。

2. 从性能可伸缩性角度看：如果我们将 messageCount 放在第三方专门关联表中，那么用户增加新帖子，除了对 Message 对应的表操作外，还需要对这个关联表操作，而且必须是事务的，事务是反伸缩性的，性能差，如果象 messageCount 各种统计字段很多，跨表事务边界延长，这就造成整体性能下降。

3. 当然，如果将 messageCount 硬是作为 Account 表字段，由于整个软件的业务操作都是 Account 操作的，是不是将其他业务统计如 threadCount 等等都放到 Account 表中呢？这会造成 Account 表变大，最终也会影响性能。

那么 messageCount 每次都通过查询 Message 对应表中用户所有发帖数获得，这也会导致性能差，表中数据越多，这种查询就越费 CPU。

使用缓存，因为 Account 作为模型被缓存，那么其 messageCount 值将只有第一次创建 Account 执行查询，以后就通过缓存中 Account 可直接获得。

所以，根据 DDD，在 AccountRepository 或 AccountFactory 实现数据表和实体 Account 的转换，Account 中的值都是在这个类中通过查询数据表获得的。

当访问量增加时，这里又存在一个性能问题，虽然一个 Account 创建时，messageCount 查询耗时可能觉察不出，但是如果是几十几百个 Account 第一次创建，总体性能损耗也是比较大的，鉴于我们对可伸缩性无尽的追求，这里还是有提升余地。

从设计角度看，由于 messageCount 不是 Account 的必备字段，因此，不是每次创建 Account 时都要实现 messageCount 的赋值，可采取即用即查方式。所以，我们需要下面设计思路：

```
public class Account{

    private int messageCount = -1;

    public int getMessageCount(){
        if(messageCount == -1)
            //第一次使用时即时查询数据表
            return messageCount;
        }
    }
```

怎么实现这个功能呢？使用 Hibernate 的懒加载？使用 Lazy load 需要激活 Open

Session In View，否则如果 Session 关闭了，这时客户端需要访问 messageCount，就会抛 lazy Exception 错误，但是 OSIV 只能在一个请求响应范围打开，messageCount 访问可能不是在这次请求中访问，有可能在后面请求或其他用户请求访问，所以，这个懒加载必须是跨 Session，是整个应用级别的。

实际上，只要 Account 保存在缓存中，对象和它的字段能够跨整个应用级别，这时，只要在 messageCount 被访问即时查询数据表，就能实现我们目标，其实如此简单问题，因为考虑 Hibernate 等 ORM 框架特点反而变得复杂，这就是 DDD 一直反对的技术框架应该为业务设计服务，而不能成为束缚和障碍，这也是一山不容二虎的一个原因。

这带来一个问题，如何在让 Account 这个实体对象中直接查询数据库呢？是不是直接将 AccountRepository 或 AccountDao 注射到 Account 这个实体呢？由于 AccountDao 等属于技术架构部分，和业务 Account 没有关系，只不过是支撑业务运行的环境，如果将这么多计算机技术都注射到业务模型中，弄脏了业务模型，使得业务模型必须依赖特定的技术环境，这实际上就不是 POJO 了，POJO 定义是不依赖任何技术框架或环境。

POJO 是 Martin Fowler 提出的，为了找到解决方式，我们还是需要从他老人家方案中找到答案，模型事件以及 Event 模式也是他老人家肯定的，这里 Account 模型只需要向技术环境发出一个查询 Event 指令也许就可以。

那么，我们就引入一个 Domain Events 对象吧，以后所有与技术环境的指令交互都通过它来实现，具体实现中，由于异步 Message 是目前我们已知架构中最松耦合的一种方案，所以，我们将异步 Message 整合 Domain Events 实现，应该是目前我们知识水平能够想得到的最好方式之一，当然不排除以后有更好方式，目前 Takia1.0 已经整合了 Domain Events + 异步消息机制，我们就可以直接来使用。

这样，Account 的 messageCount 即用即查就可以使用 Domain Events + 异步消息实现：

```
public int getMessageCount(){
    if (messageCount == -1) {
        if (messageCountAsyncResult == null) {
            //向技术环境发出查询获得 messageCount 值的命令，
            //这个命令是在另外新线程实现，因此结果不一定立即返回
            messageCountAsyncResult =
domainEvents.computeAccountMessageCount(account.getUserIdLong());
        } else {
            //当客户端再次调用本方法时，可以获得查询结果，
            //如果查询过程很慢，还是没有完成，会在这里堵塞等待，但概率很小
            messageCount = (Integer) messageCountAsyncResult.getEventResult();
        }
    }
}
```

messageCount 最后获得，需要通过两次调用 getMessageCount 方法，第一次是激活异步查询，第二次是获得查询结果，在 B/S 架构中，一般第二次查询是由浏览器再次发出请求，这浏览器服务器一来一回的时间，异步查询一般基本都已经完成，这就是充分利用 B/S 架构的时间差，实现高效率的并行计算。

所以，并不是一味死用同步就能提高性能，可伸缩性不一定是指单点高性能，而是指整个系统的高效率，利用系统之间传送时间差，实现并行计算也是一种架构思路。这种思

考思路在实际生活中也经常会发生。

最后，关于 `messageCount` 还有一些有趣结尾，如果浏览器不再发第二次请求，那么浏览器显示 `Account` 的 `messageCount` 就是 -1，我们可以做成不显示，也就看不到 `Account` 的发帖总数，如果你的业务可以容忍这个情况，比如目前这个论坛就可以容忍这种情况存在，`Account` 的 `messageCount` 第一次查询会看不到，以后每次查询就会出现，因为 `Account` 一直在缓存内存中。

如果你的业务不能容忍，那么就在浏览器中使用 AJAX 再次发出对 `getMessageCount` 的二次查询，那么用户就会每次

都会看到用户的发帖总数，`Forum` 这个论坛的标签关注人数就是采取这个技术实现的。这样浏览器异步和服务器端异步完美结合在一起，整个系统向异步高可伸缩性迈进一大步。

更进一步，有了 `messageCount` 异步查询，如何更新呢？当用户发帖时，直接对内存缓存中 `Account` 更新加一就可以，这样，模型操作和数据表操作在 DDD + 异步架构中完全分离了，数据表只起到存储作用(`messageCount` 甚至没有专门的存储数据表字段)，这和流行的 NoSQL 架构是同一个思路。

由于针对 `messageCount` 有一些专门操作，我们就不能直接在 `Account` 中实现这些操作，可以使用一个专门值对象实现。如下::

```
public class AccountMessageVO {

    private int messageCount = -1;

    private DomainMessage messageCountAsyncResult;

    private String userId;

    //值对象不能引用其他对象，值对象都是值，不能有引用
    //值对象是树的叶子，可能会被任何架构 hold 住，如果引用
    //其他对象，那些对象资源会被 Hold 住，造成 Memory Leak
    //private Account account

    public AccountMessageVO(Account account) {
        super();
        .....
    }

    public int getMessageCount(DomainEvents domainEvents) {
        if (messageCount == -1) {
            if (messageCountAsyncResult == null) {
                messageCountAsyncResult =
                domainEvents.computeAccountMessageCount(account.getUserIdLong());
            } else {
                messageCount = (Integer) messageCountAsyncResult.getEventResult();
            }
        }
        return messageCount;
    }

    public void update(int count) {
        if (messageCount != -1) {
```

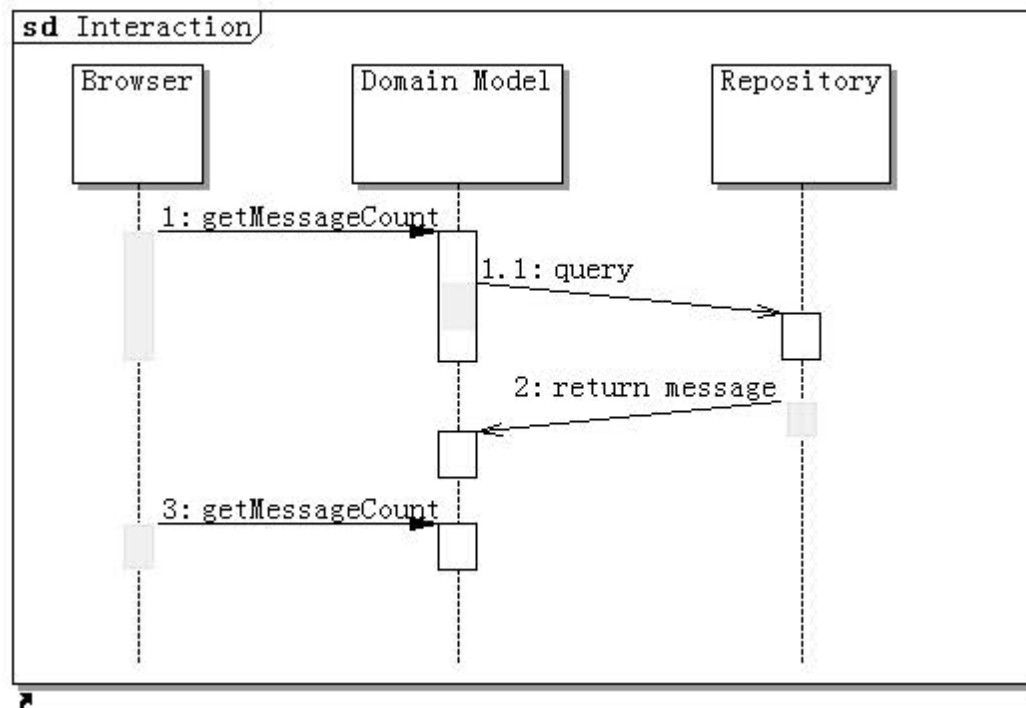
```

        messageCount = messageCount + count;
    }

}
}

```

调用顺序图如下：



下面再以帖子的附件或图片来说明懒加载的应用：

从业务上看 FourmMessage 和其附件 Attachment 并不存在严格的聚合关系，不是彼此非缺不可的，没有必要在 FourmMessage 中包含或引用一个 Attachment。

Attachment 是一个值对象，而非实体，从业务上讲可能不明显，但是因为在技术上我们要实现 Attachment 内部的懒加载，懒加载即用即加载，不用就消失的这个功能本身特点决定了它必须是一个值对象。

上面是从 Attachment 的加载也就是读取角度考虑他们之间的关系，如果需要更新修改一个 ForumMessage 的 Attachment 如何办？

我们知道更新一个模型需要涉及到 UI 的模型，在 Takia 中 UI 模型使用 Form 实现，ForumMessage 的 UI 模型是 MessageForm。

从写入流程来看，用户提交修改后的表单，也就是一个新的 MessageForm，(当然，修改之前的显示原有 ForumMessage 属于读流程，实现思路前面一样)，在 MessageForm 中有 Attachment 的 set 方法：

```

//第一步：用户提交修改表单，更新了附件，首先激活这个方法
public void setAttachment(AttachmentVO attachment) {
    this.attachment = attachment;
}

```

//第二步: Takia 框架将 MessageForm 拷贝到 ForumMessage 时, 调用这个方法
//然后将 Attachment 通过 ForumMessage 的 setAttachment 方法给予 ForumMessage

```
public AttachmentVO getAttachment() {  
    return attachment;  
}
```

用户提交表单后, 新的 Attachment 从 MessageForm 传递到了一个新的 ForumMessage 中, ForumMessage 代码如下:

```
//从 MessageForm 取出的 attachment 被 Takia 框架调用这个方法赋值  
public void setAttachment(AttachmentVO attachment) {  
    attachment.setMessageId(messageId);  
    //呼唤仓储角色来实现仓储更新  
    repositoryRole.saveUploadFiles(attachment);  
}
```

//懒加载 读取调用这个方法

```
public AttachmentVO getAttachment() {  
    return new AttachmentVO(this.messageId, this.lazyLoaderRole);  
}
```

下面看看关键的 AttachmentVO 这个值对象的内部:

```
public class AttachmentVO extends LazyLoader {  
    //值对象中只能使用值, 不能直接引用ForumMessage  
    private long messageId;  
  
    // AttachmentVO缺省是用于读取, 加载角色是其缺省配置  
    private final LazyLoaderRole lazyLoaderRole;  
  
    // for upload files lazyload  
    private volatile Collection uploadFiles;  
  
    private boolean reload;  
  
    public AttachmentVO(long messageId, LazyLoaderRole lazyLoaderRole)  
    {  
        super();  
        this.messageId = messageId;  
        this.lazyLoaderRole = lazyLoaderRole;  
        this.uploadFiles = new ArrayList();  
    }  
  
    public long getMessageId() {
```



```

        return messageId;
    }

    public void setMessageId(long messageId) {
        this.messageId = messageId;
    }

    public void importUploadFiles(Collection uploadFiles) {
        this.uploadFiles = uploadFiles;
    }

    public void updateUploadFiles(Collection uploadFiles) {
        this.uploadFiles = uploadFiles;
        super.preload();
        this.reload = true;
    }

    public Collection exportUploadFiles() {
        return this.uploadFiles;
    }

    public Collection getUploadFiles() {
        if (reload) {
            this.uploadFiles = (Collection) super.loadResult();
            reload = false;
        }
        return uploadFiles;
    }

    public void preloadUploadFileDatas() {
        if (uploadFiles == null || uploadFiles.size() == 0)
            return;
        for (Object o : uploadFiles) {
            UploadFile uploadFile = (UploadFile) o;
            //懒加载附件列表中元素的附件数据。
            uploadFile.preloadData();
        }
    }

    @Override
    //实现懒加载模板必须完成的方法，获取ForuMessage的所有附件列表
    //列表元素中也没有真正加载图片等附件数据，这里也会设置一个懒加载
    public DomainMessage getDomainMessage() {
        return lazyLoaderRole.loadUploadFiles(Long.toString(messageId));
    }

```

```
}  
  
}
```

下面看看懒加载模板 LazyLoader 的实现原理，可以重用多个需要懒加载的场景，AttachmentVO 是继承这个模板的：

```
public abstract class LazyLoader {  
  
    //  
    protected volatile DomainMessage domainMessage;  
  
    //第一次调用这个方法，激活懒加载，但是不重视获得加载的结果  
    public void preload() {  
        try {  
            if (domainMessage == null)  
                domainMessage = getDomainMessage();  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
    }  
  
    //第二次调用这个方法，获得懒加载激活后产生的结果  
    public Object loadResult() {  
        Object loadedResult = null;  
        try {  
            if (domainMessage == null)  
                preload();  
            loadedResult = domainMessage.getEventResult();  
            if (loadedResult != null)  
                domainMessage = null;  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
        return loadedResult;  
    }  
  
    //具体实现类需要实现的模板方法，返回一个事件消息  
    public abstract DomainMessage getDomainMessage();  
}
```

即用即加载

比如图片，只有显示图片时才会需要图片的二进制数据，这个数据比较大，所以，一般从持久层加载图片时，只加载图片其他核心文字信息，如图片 ID，名称等，当需要显示时，再加载二进制数据输出真正图片。

```
public byte[] getData() {
    byte[] bytes = super.getData();
    if (bytes != null)
        return bytes;
    preloadData();
    bytes = (byte[]) imgDataAsyncResult.getEventResult();
    this.setData(bytes);
    return bytes;
}

//预加载 可在 JSP 即将显示图片之前发出事件激活该方法
public void preloadData() {
    if (imgDataAsyncResult == null && domainEvents != null)
        imgDataAsyncResult = this.domainEvents.loadUploadEntity(this);
}
```

传统意义上，懒加载和异步都是好像不被人接受的，会带来比较差的性能，高延迟性，属于边缘技术，这其实是被误导了：并发策略可以解决延迟

懒加载和异步代表的并发策略实际是一种潮流趋势，特别是作为并行计算语言 Scala 和 erlang 的新亮点：函数式编程 functional programming 的特点

Takia 框架实现了领域模型层的懒加载和异步，可以完全克服 Hibernate 等 ORM 框架带来的 Lazy 懒加载错误问题。

即用即加载特点

Takia 即用即加载和 Hibernate 的延迟加载或 AJAX 的异步加载是有区别的，AJAX 的异步加载会发出很多 Http 请求；而 Hibernate 等延迟加载是无法跨请求的，只能在一个请求响应发往客户端结束前全部完成。

而基于 Takia+disruptor 的即用即加载是跨请求的，也就是说，只有当第一次输出 JSP 页面时，需要访问模型对象的 getXXX 方法时，才会从数据库中加载。

当然，所有这些都必须以 DDD 模型对象为中心设计，这里也涉及到聚合根的设计。可以设计两种聚合根，一个负责对外，一个负责对内。

所谓对外，打个比喻，当我们看到一个球时，我们这时处于球的外部，看到的是球的表面特征，比如球面光滑等的；而当我们走到球内部，看到球里面有很多房子，这是对内。

我们人观察任何事物都有这样内外分别，那么在访问模型对象时，也有内外之分，这时聚合根就要负责其内外分别的职责。

比如 DDD 书籍中 Car 和发动机两个都是聚合根，发动机是提供 Car 内部动力，而 Car 则是一个集合概念，负责对外，我们首先看到的是 CAR，走到 Car 内部，我们才感受到

发动机。

但是 Car 和发动机的区别是：发动机你可以看到一个具体机器，而 Car 则没有具体形态，它内部是由发动机 车轮 车厢等组成，只有你走出 Car 内部，才看到 Car 的形态，这时 Car 其实是一种外表边界而已。

所以，一般重建聚合根时，我们可以倾向于先构建一个框架聚合根 Car，然后由 Car 负责加载重建其内部一个个元素组件。

Forum 论坛聚合根也是由 Thread 这个组合体和 RootMessage 这两个聚合根，因此加载一个帖子时，首先加载一个 Thread，其他部分加载看情况而定，如果是要看帖子内部内容，届时显示时会加载与内容显示相关的部件；而如果只是想看看很多 Thread 列表，从外部看 Thread，那届时就会加载 Thread 与外部特征有关的数据。这和从外部看球与走到球内部的道理是一样的。

顺序执行

Takia 框架的事件机制是并发的多线程，好处是能充分应用 CPU，提高性能，带来的问题是如果业务上希望事件是顺序执行的，如何实现呢？

通过领域模型发出事件指挥技术架构为之服务，这些事件是异步的，如果在第一个事件响应还没有处理完成，第二个事件就要基于第一事件的处理结果进行进一步处理，如何协调他们的前后一致的关系？

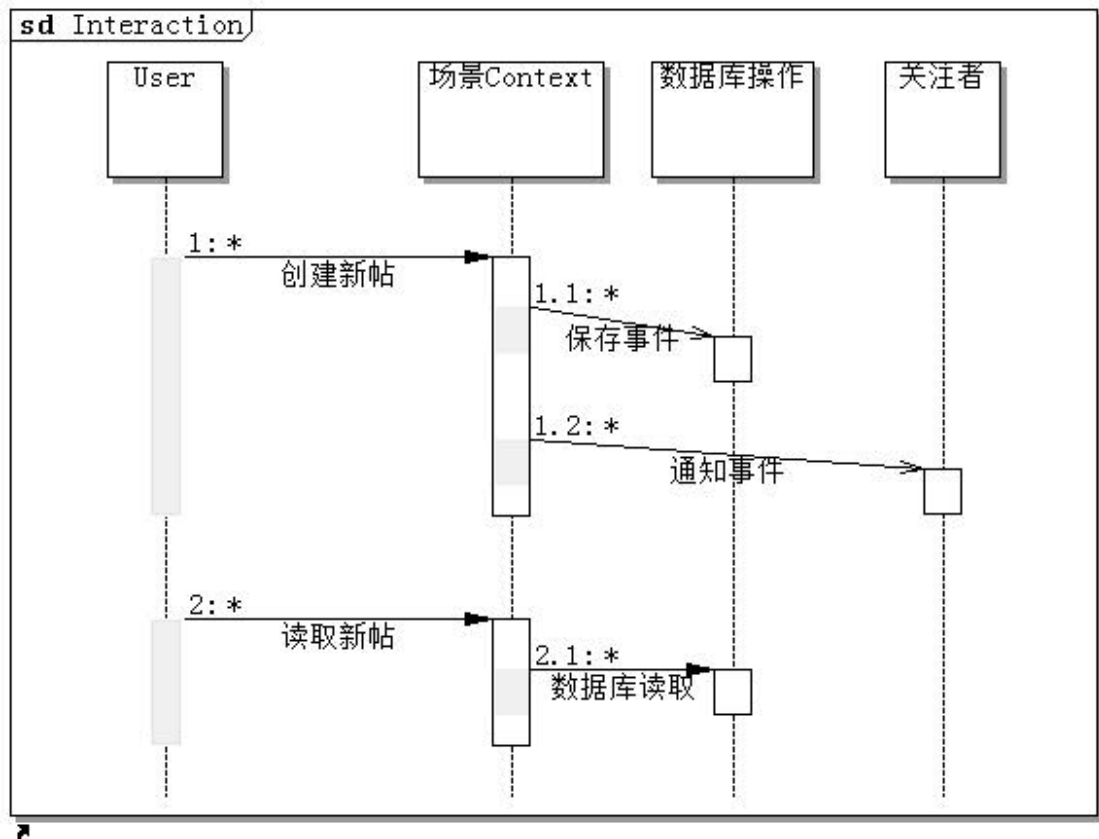
如果说我们普通编程是在一个线程内进行顺序同步编程，那么 Takia 框架的事件编程实际是一种并行异步编程模型，普通编程需要花力气让程序并行运行；而事件编程则要注意让其顺序执行。

Takia 框架事件模型提供两种顺序执行的方式：

1. 根据事件订阅者类的名称顺序执行，比如 ADomainHandler 在 BDomainHandler 之前执行，BDomainHandler 在 CDomainHandler 之前执行。
2. 将上一个事件的处理结果传入下一个事件，然后进行堵塞等待方式。下面主要介绍这种方式。

以 Forum 中帖子创建为案例，当用户提交新的帖子时，发出领域事件进行数据库保存，用户然后就会立即再次从数据库读取这个新帖子，而在另外一个线程中数据库保存还没有完成，那么从数据库是无法读到这个新帖子的。

我们将用户提交新帖子后的处理顺序图如下：



用户有两个动作：创建新帖后再读取新帖，这两个动作次序是有先后次序的，否则读取新帖就无法正确读取。

所以，我们希望这个过程是一个顺序执行的：

创建新帖；然后发出保存事件，数据库保存完成新帖后；再执行通知事件；如果在这一过程中，用户又几乎同时发出读取新帖，那么一定要等待创建新帖的过程全部完成才能执行。

为了实现这种顺序执行过程，我们使用 Takia 框架的 DomainMessage 的中一种堵塞读取方法，将发出保存事件的 DomainMessage 传递给下一个通知事件，然后在通知事件激活后堵塞读取上一次事件结果 DomainMessage。

```

RepositoryRoleIF repositoryRole = (RepositoryRoleIF) roleAssinger.assign(forumMessage, new
RepositoryRole());
//获得数据库保存的事件结果
DomainMessage domainMessage = repositoryRole.addTopicMessage(forumMessage);
//将事件结果保留，以便读取新帖时使用
transactions.put(forumMessage.getMessageId(), domainMessage);

ThreadRoleIF threadRole = (ThreadRoleIF) roleAssinger.assign(forumMessage, new ThreadRole());
//将数据库保存的事件结果作为通知事件的输入参数
threadRole.aftercreateAction(domainMessage);
  
```

在通知事件的订阅者中代码如下：

```

//获得传入的输入参数，也就是数据库保存事件结果
DomainMessage lastStepMessage = (DomainMessage) event.getDomainMessage().getEventSource();
//对数据库保存事件进行堵塞读取，一直等待数据库保存完成
Object lastStepOk = lastStepMessage.getEventResult();
  
```

```

if (lastStepOk != null) { //如果数据库保存完成
    ForumMessage forumMessage = (ForumMessage) lastStepOk;
    boolean isReplyNotifyForAuthor = forumMessage.isReplyNotify();
    forumMessage = forumFactory.getMessage(forumMessage.getMessageId());
    forumMessage.getForum().addNewThread(forumMessage);

    messageNotifyAction(isReplyNotifyForAuthor, forumMessage);
}

```

在通知事件的实现中，将对上次发出的事件处理结果进行堵塞读取处理结果，如果没有处理结果，就一直等待直至有结果。

那么在上次数据库保存的事件处理中如何表示自己的事件处理完成，有了处理结果呢？

实际上，关键是，在数据库保存事件的订阅者中，只要在处理保存数据库完成后，将一个非空对象赋值给 `DomainMessage` 即可。如下：

```
domainMessage.setEventResult(forumMessage);
```

因为 `forumMessage` 是非空，当调用 `DomainMessage` 的 `setEventResult` 方法时，此方法中有一个新的 `Disruptor` 队列，而通知事件中堵塞读取等待的就是这个新的专门用来传递处理结果的 `Disruptor` 队列，当 `Disruptor` 队列消息时，堵塞读取立即被激活，获得 `forumMessage`，被堵塞的下一个事件就能够知道上一步事件已经处理完成。

在 `Forum` 中，数据库保存事件订阅者是在 `MessageTransactionPersistence` 中使用 `@OnEvent` 方法标注实现的。

```

@OnEvent("addTopicMessage")
public ForumMessage insertTopicMessage(ForumMessage forumMessage) throws Exception {
    logger.debug("enter createTopicMessage");
    try {
        jtaTransactionUtil.beginTransaction();
        messageRepository.createTopicMessage(forumMessage);
        logger.debug("createTopicMessage ok!");
        jtaTransactionUtil.commitTransaction();
    } catch (Exception e) {
        jtaTransactionUtil.rollback();
        String error = e + " createTopicMessage forumMessageId=" + forumMessage.getMessageId();
        logger.error(error);
        throw new Exception(error);
    }
    //返回一个非空对象，表示当前处理过程完成
    return forumMessage;
}

```

由于事件订阅者没有直接使用原始的事件订阅者方式，也就是使用 `@Consumer("addTopicMessage")`，以及实现接口 `com.reeham.component.ddd.message.DomainEventHandler`。

而是使用 `Takia` 框架更为灵活的 `@OnEvent`，这时只要被标注 `@OnEvent` 的方法必须返回一个非空对象，那么在 `Takia` 框架中有一个缺省的 `DomainEventHandler` 实现类，就会将 `@OnEvent` 标注的方法返回值自动执行 `DomainMessage` 的 `setEventResult` 方法。

由此可见，只要我们在下一个事件处理之前，对上一个事件处理结果进行堵塞等待，

就能确保事务顺序依次执行。

下面谈谈读取新帖这个事件如何堵塞读取第一个处理事件呢？

我们注意到，在将第一个事件数据库保存事件处理结果传入第二个事件之前，我们已经将第一次事件处理结果保存：

```
transactions.put(forumMessage.getMessageId(), domainMessage);
```

那么，当有读取新帖这中随时可能发生的事件时，我们只要从保存的第一次事件处理结果中也是堵塞读取确认第一个事件处理结束再继续：

```
public boolean isTransactionOk(Long messageId) {
    if (transactions.size() == 0)
        return true;
    if (!transactions.containsKey(messageId)) {
        return true;
    }
    //获得第一次事件
    DomainMessage message = transactions.get(messageId);
    //堵塞读取第一次事件处理结果
    if (message.getEventResult() != null) {
        transactions.remove(messageId);
        return true;
    }
    return false;
}
```

DCI 实现

DCI：数据 Data, 场景 Context, 交互 Interactions 是由 MVC 发明者 Trygve Reenskaug 发明的。见 DCI 架构是什么？DCI 让我们的核心模型更加简单，只有数据和基本行为。业务逻辑等交互行为在角色模型中 在运行时的场景，将角色的交互行为注射到数据中。

Takia 框架提供了两种 DCI 实现风格：没有领域事件的纯粹注入；另外一种领域事件的注入；这两种模型适合不同的场景：

如果我们已经 Hold 住了一个领域对象，那么就直接通过其发出领域事件实现功能；比如模型的修改。否则，我们创建一个上下文 Context, 在其中通过 RoleAssigner 将角色接口注入到领域对象中。比如模型新增创建或删除。(对象不能自己创建自己)。

下面谈谈这两种 DCI 模型的实现：

角色分配场景

数据模型如下：

```

@Model
public class MyModel {
    private Long id;
    private String name;
    No domain events with @Inject
    public Long getId() {
        return id;
    }
}

```

DCI 的 Role 实现：角色必须有一个接口，否则无法实现 Mixi 注入模型的功能，这是实现 DCI 必要条件。

```

@Introduce("message")
public class RepositoryRole implements RepositoryRoleIF {
    // interface needed

    @Send("save")
    public DomainMessage save(MyModel myModel) {
        return new DomainMessage(myModel);
    }
}

```

Takia 框架提供了一种角色分配器：com.reeham.component.ddc.dci.RoleAssigner 是一个角色分配器，可以向任何模型中注入任何带有接口 (Mixin)的实现类。

当使用 RoleAssigner, 我们就没有必要从带有元注释 @Introduce(“domainCache”) 和 @Around 的仓储中首先获得一个模型对象，这是模型对象从一个主动核心让位于包含有 RoleAssigner 的 Context 场景上下文，模型成为被注射的被动对象了；而在领域事件驱动中，模型是发出领域事件，是一种主动核心对象。

RoleAssigner 可以手工对任何一个模型对象从外部进行事件注入或角色分配。

下面是使用角色分配器实现的 DCI 场景代码：

```

@Component("repositoryContext")
public class RepositoryContext {
    private final RoleAssigner roleAssigner;

    public RepositoryContext(RoleAssigner roleAssigner) {
        this.roleAssigner = roleAssigner;
    }

    public void interact() {
        MyModel myModel = new MyModel();
        RepositoryRoleIF r =
            (RepositoryRoleIF)roleAssigner.assign(myModel, new RepositoryRole());
        r.save(myModel);
    }
}

```

注意这个 RepositoryContext 必须标注以@Component，表示其生存在组件边界中，

Takia 框架有两个边界：

@Model 的模型对象是生活在缓存内存中；而@Component/@Service 生活在应用容器中，比如 Web 容器的 ServletContext 中。

很多情况下，在模型对象还没有生活到内存中时，应用容器总是首先存在，应用容器如同天地，万物之母。

比如模型对象的新增创建时，由于这个模型对象在数据库仓储中还不存在，因此在这种情况下，我们不能 Hold 住还没有创建的模型对象，这时模型对象内部的领域事件就派不上用处。

但是应用容器以及存在，因此，我们可以使用已经生存在应用容器中的 com.reeham.component.ddd.dci.RoleAssigner 对新创建的模型进行功能注入，让这个从页面提交过来的新的模型对象具有能够自己持久化(save())或其他必要的业务功能。

这样，我们通过获取模型时的自动注入和 RoleAssigner 的手工注入两种手段，灵活地根据不同上下文实现风格的 DCI。

这种以角色分配器为主要的场景模式的 DCI 比较适合脱离领域模型内部的外部操作，Takia 框架认为世界基本上有三个部分组成：边界内的事物内部；边界；边界外的事物外部。事物的创建删除一般由事物外部作用，因为一个事物不可能在自己内部创建自己，因为它自己还不存在。

角色分配器的注入 DCI 比较适合模型的创建或删除，在开源 Forum 的 MessageKernel, ForumMessage 的创建删除是利用角色分配器实现，而 ForumMessage 的修改则是利用模型的领域事件实现。

领域事件的 DCI

这种模式的 DCI 实现特点是隐含了场景，凡是领域模型发出事件的地方就是一种 DCI 中的场景上下文。这时领域模型占据了核心位置，场景是被隐含在领域模型内部，这不同于上一种角色分配器中场景和领域模型是分离的。

下图是领域模型在运行时领域事件被 Takia 框架自动注入的原理图，这个运行时是指，我们从 Repository 仓储(带有元注释 @Introduce(“domainCache”) 和 @Around 的仓储)获得领域模型对象时发生的。

```

@Model
public class MyModel {

    private Long id;
    private String name;

    @Inject
    private MyModelDomainEvent myModelDomainEvent;

    @Inject
    private MyModelService myModelServiceCommand;
}

```

Inject

```

@Introduce("message")
public class MyModelDomainEvent {

    @Send("MyModel.findName")
    public ModelMessage asyncFindName(MyModel myModel) {
        return new ModelMessage(myModel);
    }
}

```

Introduce a interceptor

MyModel $\xrightarrow{\text{invoke}}$ @Introduce(message) $\xrightarrow{\text{invoke}}$ MyModelDomainEvent

上图中领域事件 MyDomainEvent 的注入是在如下代码被调用时发生的：

```

@Component("mymrepository")
@Introduce("domainCache")
public class RepositoryImp implements MyModelRepository {

    @Around
    public MyModel getModel(Long key) {
        MyModel mym = new MyModel();
        mym.setId(key);
        return mym;
    }
}

```

Auto cache MyModel

也就是当如下场景执行时，将会发生事件注入：

```

//当从仓储 RepositoryImp 获得模型对象时，事件被注入到 myModel 对象中
MyModel myModel = repository.getModel(new Long(100));

//通过模型发出事件。
myModel.myModelDomainEvent.asyncFindName(this);

```

上图中,不但可以注射 MyModelDomainEvent 到模型 MyModel 中,偶尔也可以将 Takia 框架中任何一个以 @component 或 @Service 标注的组件如 MyModelService 注射到 MyModel 中,这种方式相比 DomainEvents 的异步,这是同步方式。

模型注入功能实际上有助于实现 DCI 架构,因为 DCI 架构关键是将角色注入数据模型中。而事件的发布者也就是生产者其实就是一个具有业务行为的角色。

为更清楚说明 DCI,下面以 Takia 案例说明。

领域模型是 DCI 的 Data,只有数据和基本行为,更简单,但注意不是只有 setter/getter 的贫血模型。如下:

```
@Model
public class UserModel {

    private String userId;
    private String name;
    //注意 这是一个带 Domain Events 注入的
    @Inject
    private ComputerRole computerRole;
```

Domain Events 事件或消息的生产者也就是 DCI 中的角色 Role,比如我们有一个专门进行计数计算的角色,实际上真正计算核心因为要使用关系数据库等底层技术架构,并不真正在此实现,而是依托消息消费者 @Consumer 实现,那么消息生产者可以看出是一个接口,消费者看成是接口的实现:

```
@Introduce("message")
public class ComputerRole {

    @Send("computeCount")
    public DomainMessage computeCount(UserModel user) {
        return new DomainMessage(user);
    }

    @Send("saveUser")
    public DomainMessage save(UserModel user) {
        return new DomainMessage(user);
    }

}
```

DCI 第三个元素是场景 Context,在这个场景下,ComputeRole 将被注入到模型 UserModel 中,实现计数计算的业务功能:

```
public class ComputeContext {
```

```

private DomainMessage ageAsyncResult;

public void preloadData(UserModel user) {
    if (ageAsyncResult == null)
        ageAsyncResult = user.getUserDomainEvents().computeCount(user);
}

public int loadCountNow(UserModel user) {
    preloadData(user);
    return (Integer) ageAsyncResult.getEventResult();
}

public int loadCountByAsync(UserModel user) {
    if (ageAsyncResult == null)
        ageAsyncResult = user.getUserDomainEvents().computeCount(user);
    else if (ageAsyncResult != null)
        return (Integer) ageAsyncResult.getEventResult();
    return -1;
}
}

```

上述是 UserModel 中有一个 @Inject:

```

@Inject
private ComputerRole computerRole;

```

模型驱动开发(MDD)案例

DDD 是领域驱动设计(Domain-Driven Design)的简称, DDD 是一种分析设计建模方法, 它倡导统一语言, 提出了实体和值对象 以及聚合根等概念, 借助 DDD 我们能够在结构理清需求中领域模型。

DCI: Data 数据模型, Context 上下文或场景, Interactions 交互行为是一种新的编程范式, 由 MVC 发明人 Trygve Reenskaug 提出。

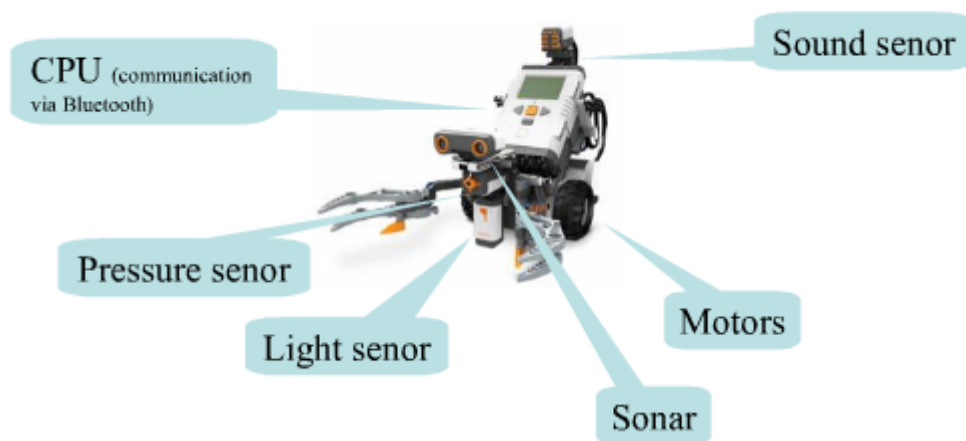
DCI 的关键是:

1. 要让核心模型非常瘦.
2. 逻辑或行为应该放在角色这个类中

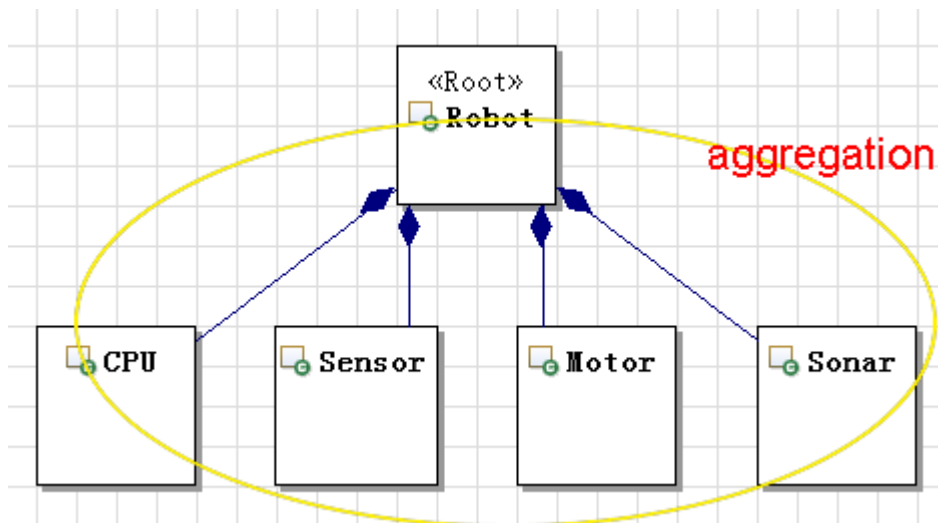
Event Sourcing 是由 Martin Fowler 提出, 是将业务领域精髓 (尤其是最复杂的) 与技术平台的复杂性实现脱钩的天作之合。

下面我们将演示如何将上述三者在实践中结合在一起?

以机器人 Robot 这个需求为案例, 下图是 Robot 描述, 我们根据这种图通过 DDD 建模。



从上面这张图中，我们根据 DDD 的实体聚合根等定义，得出如下类图：



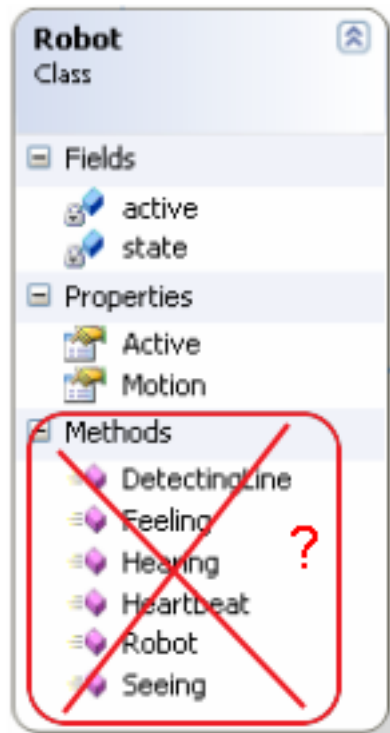
可见，Robot 作为一个聚合集合的根实体，它有四个重要部分组成，这些组成部分与 Robot 形成一种高凝聚的组合关系。缺一不可。

有了这样的结构关系，我们还将细化方法行为，**根据 对象的责任与职责**，也就是职责驱动开发方法论，它提出一种角色职责的模型：roles-and-responsibilities，见 Object Design: Roles, Responsibilities, and Collaborations)，什么是职责呢？职责就是那些 knowing what; doing what; deciding what.

那么 Robot 如果作为一个智能机器人，它应该有哪些职责呢？

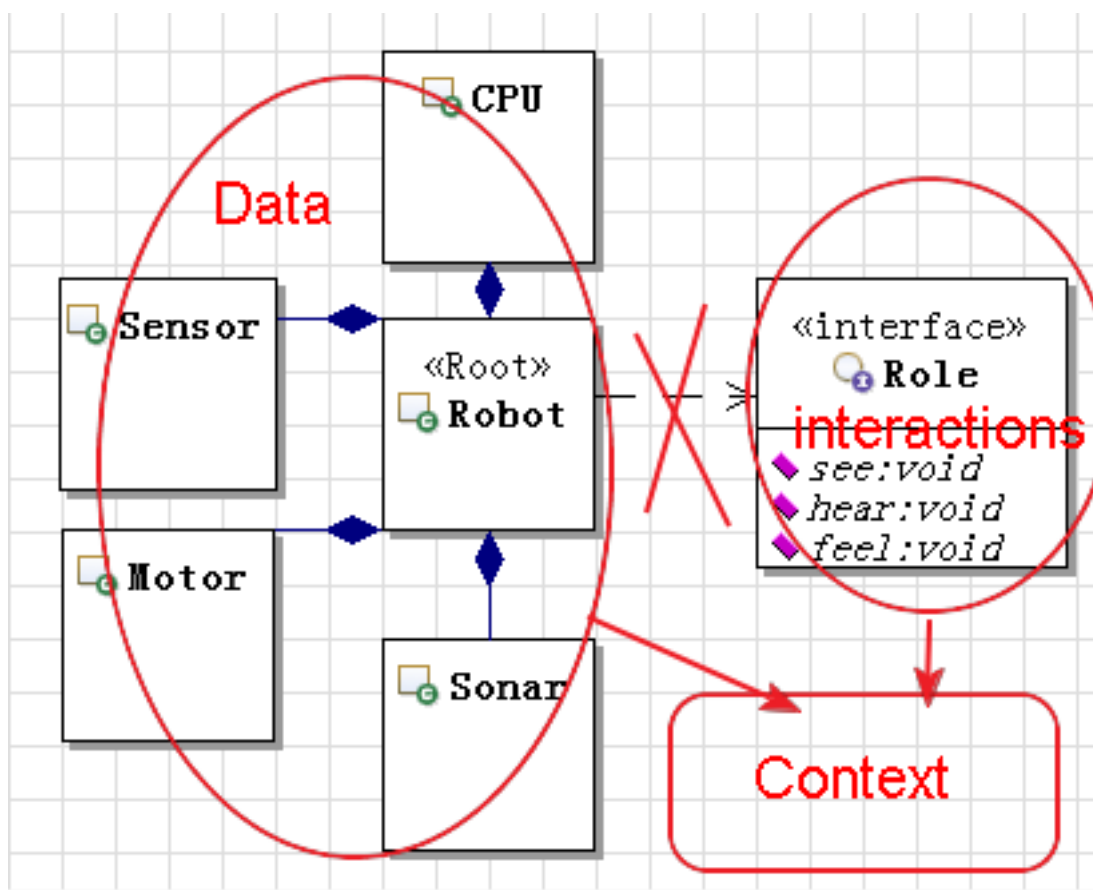
Robot 的职责功能是能听，能看或能感觉，听 看 感觉这些都是其作为一个智能机器人角色的职责。

那么，下一步关键是，如何实现这些职责呢？是否是将这些职责设计作为 Robot 实体类的方法呢？如下：



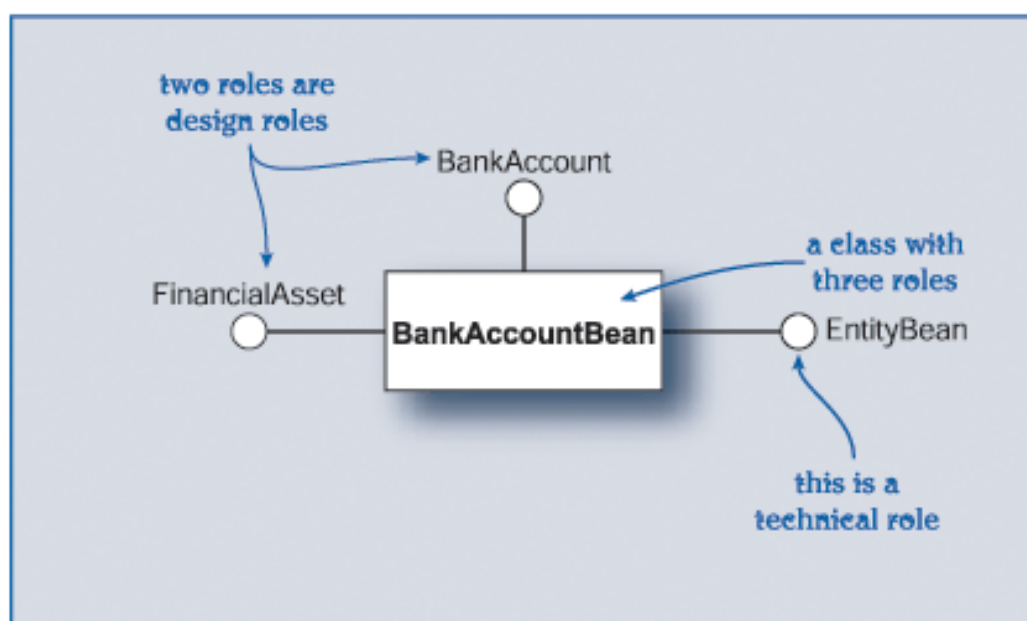
这样设计以后，可能会导致 **Robot** 实体类非常臃肿，是一个庞大的对象，这有违背 DCI 要旨。

DCI 认为要保持模型的精简，听 看 感觉这些行为是 **Robot** 作为一个智能机器人，不是普通机器人，这样一个特殊角色具备的职责，应此，应该将这些行为放入一个叫智能机器人的角色中。当在运行时需要的场景 **context** 时，我们将这个角色中的职责行为注入到精简的数据模型中，如下图：



这样的例子很多，一个人在家是父亲，在单位是经理，父亲和经理都是角色，是不是要将这些角色行为比如签字 烧菜这些和具体业务场景有关的职责放入“人”这个类中呢？显然不是。

又比如银行账户 Account 有三种角色：两个是设计角色 BankAccount 银行账户 和 FinancialAsset 理财账户，另外一个角色是技术角色，它又是 EJB 的实体 Bean 专门用来实现持久化保存。



那么如何将上面 DCI 设计或职责驱动落实为代码呢？特别是 Robot 实体类和角色智能机器人的行为如何在运行时场景结合呢？这非常类似桥模式：

```
public String hello(String id) {
    Robot robot = robotRepository.find(id);
    //将角色智能机器人 IntelligentRole 的行为注入到 Robot 数据对象中
    IntelligentRole intelligentRobot = (IntelligentRole) roleAssigner.assign(robot, new IntelligentRobot());
    //得到一个混和 robot 将具有听 看 感觉等能力行为
    return "Hello, " + intelligentRobot.hear();
}
```

类似 Account 的实体持久化角色，， Robot 也有一个保存自己到数据库的技术职责， Robot 保存自己应该是首先由自己发出这样意愿，而不是被保存，是主动保存，其次保存数据库这个动作耗时，影响性能，因此，我们使用领域事件 Domain Events 来间接实现。

一个 PublisherRole 是保存事件发送者 Robot 可以扮演这样一个角色发出保存事件。：

```
@Introduce("message")
public class PublisherRoleImp implements PublisherRole {

    @Send("saveme")
    public DomainMessage remember(Robot robot) {
        return new DomainMessage(robot);
    }
}
```

保存事件的接受方就是 DDD 中定义的仓储 Respository:

```
@Component
@Introduce("domainCache")
public class RobotRepositoryInMEM implements RobotRepository {
    .....

    //保存事件的订阅者 真正实现数据库保存
    @OnEvent("saveme")
    public void save(Robot robot) {
        memDB.put(robot.getId(), robot);
    }
    .....
}
```


.那么 Robot 在什么时候扮演事件发送者发出保存自己的命令呢？可以在任何时候，下面是一个 context:

```
public void save(Robot robot) {  
    PublisherRole publisher = (PublisherRole)  
roleAssigner.assign(robot, new PublisherRoleImp());  
    publisher.remember(robot);  
}
```

至此，我们通过机器人 Robot 案例展示了 DDD DCI 和事件模型等分析设计实现的过程，当然复杂项目将比这个过程更加复杂，需要敏捷迭代，精炼出符合客观规律的核心模型。

不变性设计原则

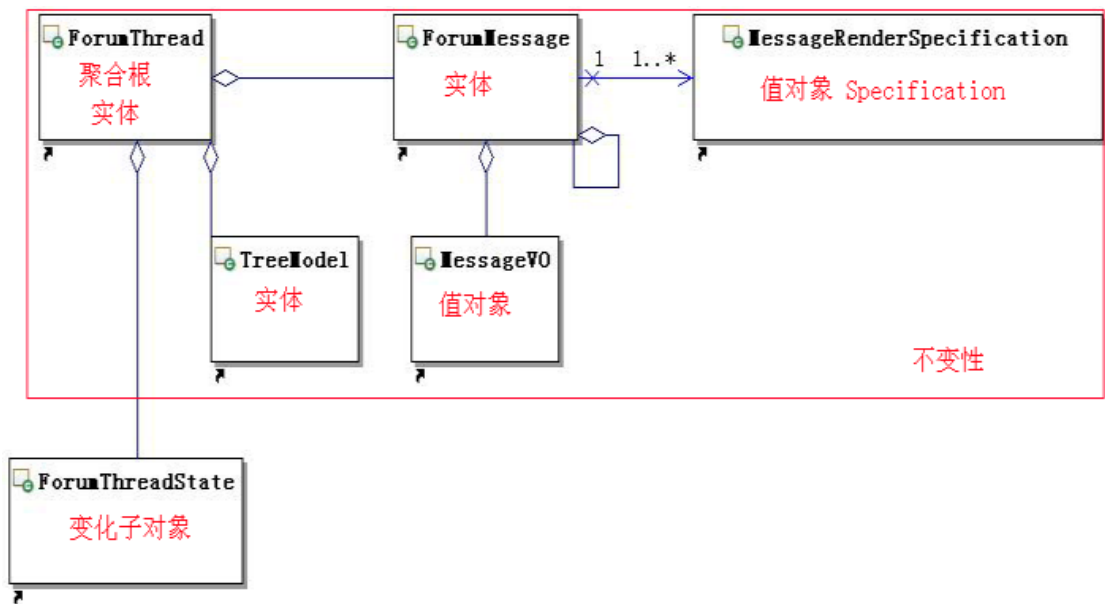
不变性是统领业务分析和高性能架构重要法门，通过业务上不变性分析设计，可以实现代码运行的并发高性能和高扩展性。

不可变性是一种抽象，它并不在自然界中存在，世界是可变的，持续不断变化。所以数据结构是可变的，他们是描述真实世界某段时间内的状态。而状态经常会被修改，如下面情况：

- 1.状态被并发线程同时修改
- 2.多个用户对一个共享对象(状态)进行冲突性修改。
- 3.用户提供了无效数据，应该被回滚。

在自然可变的模型下，会在上面情况下发生不一致性或数据破碎 **crushing**，显然可变性很容易导致错误。我们需要的是一种新视野，新角度，一种事务 **transaction** 视野，当你从事务性程序中看世界时，一切都是不可变的。这和**删除与引用透明**中功能重复执行不可变原理非常类似，体现逻辑的线性特征。

实战 DDD(Domain-Driven Design 领域驱动设计:Evans DDD)中提出状态是一种值对象，而值对象的特征是不可变性，如果发生变化，就全部替换。



值对象状态的这种不可变性还为我们进行海量数据处理提供了一种优雅的架构：如何打败 CAP 定理？

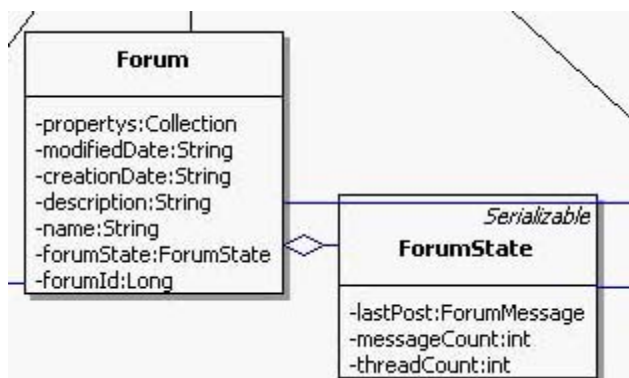
数据有两个重要属性：首先数据是基于时间的，数据是表达一段时间内一个逻辑为真的事实。另外一个属性是数据本质上是不可变的，因为和时间有关，我们是不能回到过去改变数据的真实性。

我们把基于时间的数据准确称为状态，状态是表达一段时间内一个逻辑为真的事实，状态是不可变的，因为我们不能回到过去改变状态。

针对状态的不可变性，在应用中需要针对状态改变进行一种最佳实践，也就是说：如果状态发生改变，我们只要重新构建一个新的状态对象，而不是在原有状态对象内部去修改字段。

根据这个宗旨，对 Forum 中没有对状态做到不变性替换的问题进行了重构。

以 Forum 的 ForumState 为案例，FormThread 的 ForumThreadState 也是同理，如下：



下面是根据不变性新设计的 ForumState:

```
public class ForumState {
    private final AtomicLong threadCount;

    /**
     * the number of messages in the thread. This includes the root message. So,
     * to find the number of replies to the root message, subtract one from the
     * answer of this method.
     */
    private final AtomicLong messageCount;

    private final ForumMessage lastPost;

    private final Forum forum;

    private final SubscribedState subscribedState;

    public ForumState(Forum forum, ForumMessage lastPost, long messageCount, long threadCount) {
        super();
        this.forum = forum;
        this.lastPost = lastPost;
        this.messageCount = new AtomicLong(messageCount);
        this.threadCount = new AtomicLong(threadCount);
        this.subscribedState = new SubscribedState(new ForumSubscribed(forum));
    }
}
```

与以前的 ForumState 区别主要是：将其中所有字段都加了 final，表示不能修改，构建 ForumState 时就必须指定其中的数据值。Java 中体现不可变性的特点主要是 final 和构造函数，ForumState 的就没有了所有 setXXXX 方法，不能对 ForumState 内部单独修改，强迫替换整个对象。

有了不可变性的值对象，根据 DDD 设计要则，还需要一个工厂来维护这种不变性，也就是 ForumState 的创新 new 需要通过工厂专门统一实现，不是任意地随意 new 创建的。

专门创建一个 FormStateFactory 负责 FormState 的生命周期维护：

```
public interface ForumStateFactory {

    void init(Forum forum, ForumMessage lastPost);

    void addNewMessage(Forum forum, ForumMessage newLastPost);

    void addNewThread(Forum forum, ForumMessage newLastPost);

    void updateMessage(Forum forum, ForumMessage forumMessage);

}
```

过去 ForumState 是直接进行修改其中的字段，如下：

```
public synchronized void addNewMessage(Forum forum) {
    forumState.addMessageCount();
    forumState.setLastPost(forumMessageReply);
    forumMessageReply.setForum(this);
    this.publisherRole.subscriptionNotify()
}
```

```
local: addNewMessage(ForumMessageReply)
public void addNewMessage(ForumMessageReply forumMessageReply) {
    forumStateManager.addNewMessage(this, forumMessageReply);
    this.publisherRole.subscriptionNotify()
}
```

forumState.setLasPost 这些方法是直接修改，现在重构成使用 ForumStateFactory 的 addNewMessage 来实现，其内部方法具体代码如下：

```
public void addNewMessage(Forum forum, ForumMessage newLastPost) {
    try {
        long newMessageCount = forum.getForumState().addMessageCount();
        //重新创建新的 ForumState
        ForumState forumState = new ForumState(forum, newLastPost, newMessageCount,
forum.getForumState().getThreadCount());
        //替换 Forum 原来旧的 forumState
        forum.setForumState(forumState);
        newLastPost.setForum(forum);
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

通过这种不变性的重构，好处是去除了原来 addNewMessage 方法的 synchronized 同步锁，通过不变性规避了共享锁的争夺，从而获得了更好的并发性能。

借助 Takia 的 ES 模型，通过稍微复杂点的不可变+Domain Events 实现类似 Scala 的不可变+Actor 的并发编程

需要注意的是：不同的事件引起不同的状态，如读取事件引发读状态的变化，比如帖子的浏览数；而写事件引发写状态的变化；越是频繁发生的事件，其不变性的时间周期 Scope 就越短，实现的手段就有区别。

帖子浏览数是帖子的一种读取次数状态，频繁发生，如果每次读取事件发生，我们象上面采取值对象不变性原则，每次都构造一个新的值对象，无疑会有大量垃圾临时对象产生，从而频繁触发 JVM 的垃圾回收机制，那么在这种情况下，我们可以采取其他并发措施，比如使用 JDK 的原子类型 AtomicLong 等，通过其提供的自增获取的原子功能实现并发。

还有一种状态是结构关系状态，比如帖子之间的相互回复关系可以组成一个二叉树的模型，如果有新帖写事件发生，会增加一个新帖 ID 在这个树结构中。这是一种难以避免的在原来数据上进行修改情况。

最佳实践

下面以 Forum 为案例,结合 DDD + DCI + Event Sourcing 介绍一下实战的最佳事件,方便大家能够更方便使用框架。

经过 DDD 分析设计,领域划分,找出核心领域,在核心领域分辨出实体和值对象,这些实体和值对象之间有高度依赖的聚合关系,也有普通松散的关联关系,如何表达这样对象之间关系,聚合根显得很重要。

聚合根一般是有实体充当,一个对象群中可以有多个聚合根,最好两个,太多无法达到管理作用,聚合根实际是实体在结构上充当的管理角色,实体在行为场景下也扮演有各种交互能力的角色(DCI)。

在一个聚合的对象群中,聚合根如同树形数据结构的根,而值对象如同树形的叶子,其他实体可能是叶或枝,可以下面带子对象,也可以不带,但是值对象后面是绝不会带子对象的,因为值对象内部都是值,不会对其他对象进行引用,例如下面 SampleVO 就不是严格的值对象:

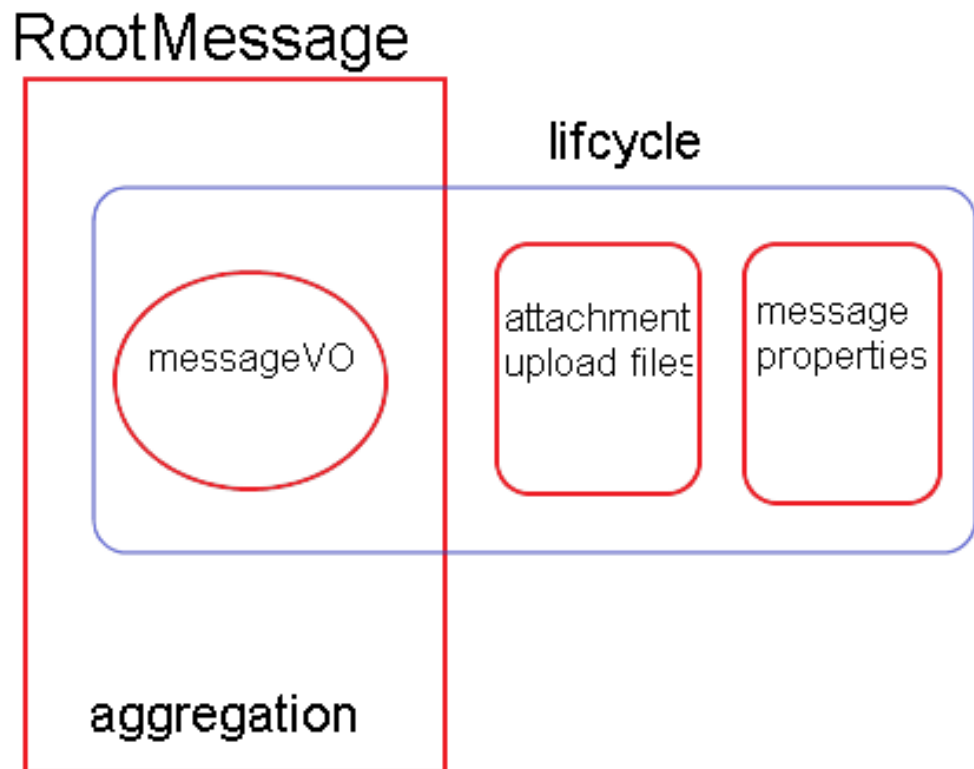
```
Class SampleVO{
    private final int id;
    private final A a;    //对其他对象引用,表示下面有分支
    private final String name;
}
```

如果非得对其他对象进行引用,使用那个对象的 ID 即可:

```
Class SampleVO{
    private final int id;
    private final int aId;    //A 对象的标识值
    private final String name;
}
```

值对象在 Java 中一个特征是字段需要加上 final,一旦赋值不能修改,如果你有需要修改的字段,那么换个角度考虑,那个字段可能是代表一个状态,状态性质数据属于实体对象。

下面是 Forum 经过分析后的类图:



RootMessage 是聚合根，当然 ForumThread 也可以看成聚合根，正如 Car 是汽车对象的聚合根，而发动机也是汽车的聚合根一样，Car 和 Thread 概念都是一种集合概念，也就是说，我们看到的 Cat 或 Thread 都是由部分组成的，并没有一个独立元素组成的，thread 是根贴和一系列回帖的组合，Car 是发动机和车轮等等组合。

messageVO 是 RootMessage 的聚合部分，其带把帖子的发言内容，帖子如果没有内容，就不叫帖子，两者之间是非常强烈的聚合关系。

而帖子上传文件和帖子属性这两个对象则是帖子可有可无的，与帖子 message 之间是一种松散的关联关系。但是它们的生命周期和 messageVO 是一致的，用户发言，填写内容后，上传文件都是同时产生，发言后，也希望两者都能立即看到，有一定的一致性要求。

从功能上看，用户发帖，也可能修改贴，我们从这两个角度分别阐述上述关联对象的不同要求。

用户发帖，实际创建一个新的帖子实体，用户提交表单后，希望立即看到他的发布结果，其他用户看到有新帖，也希望不但看到帖子内容，还有它的上传文件，而不希望只看到帖子内容，上传文件被告知耐心等待。

因此，在用户一致性来看，MessageVO 和 Attachment 是高一致性的，那么我们在创建 MessageVO，包括命令仓储储存 MessageVO 时，同时也要一同存储 Attachment。如果这里面引入异步，先存储 MessageVO，然后从仓储加载出 Message，在其中填写 Attachment，这种情况容易被中间劈腿，但你加载只有 MessageVO 的 Message 时，还没来得及填写 Attachment 时，其他用户已经取出 Message 显示了，结果看到没有附件 Attachment 的帖子。违反业务要求。

因此，在 Message 仓储中，我们将其聚合的对象 MessageVO 和关联的重要对象

Attachment 和其他业务上要求高一致性的关联对象一同存储，当成一个完整的一致的“聚合边界”内对象群：

```
//存储 MessageVO
messageDaoFacade.getMessageDao().createMessage(forumMessagePostDTO);

//存储上传文件
uploadRepository.saveAllUploadFiles(forumMessagePostDTO.getMessageId().toString(),
forumMessagePostDTO.getAttachment().getUploadFiles());

//存储帖子属性
updateMessageProperties(forumMessagePostDTO);

//存储帖子的关键词
tagRepository.saveTagTitle(forumThread.getThreadId(),
forumMessagePostDTO.getMessageVO().getTagTitle());
```

下面的问题是，在帖子被创建时需要如此高一致性，那么在帖子修改时是否也需要这样呢？

因为修改对象时，我们常常只修改其中一两个字段，如果因为修改一个字段，将其他无关的字段和子对象都一同修改更新，无疑是浪费资源的，耗费很多精力去完成聚合根其他子对象在修改时的操作，比如有的修改是 **merge** 融合，即如果原来有相同值，替代，如果没有，则和原来值合成一个集合共同存储，比如 **message** 属性 **properties**，帖子中属性如果这次有修改的则更换成新的值，如果没有修改，也不能将原来的丢弃，它不是一个 **Replace** 完全替代，而是有 **append** 的合并操作。

帖子修改时，我们内存缓存中肯定已经存在这个帖子对象，这样我们就直接修改内存帖子对象即可，然后在帖子对象内部，负责自己与仓储之间的一致性。

所以，修改实体时，需要完成两个部分工作：

1. 修改自身
2. 通过发出领域事件，修改仓储。

注意：

发出事件时，传送的对象必须是值对象，也就是不能再包含其他对象，也可以是构建的一个新对象，以为发出的事件发出后就会消失，是一种即用即完的概念，否则也容易造成内存泄漏。事件监听器处理完毕后，发出传送的对象可以是任何对象。

我们知道，在 Takia 框架中，一个领域对象有两种创建方式：

1. 从后端仓储中创建，生存在内存缓存中，可以发出领域事件
2. 从前端由用户提交表单创建，这部分无缓存，也不能发出领域事件，只能作为 DTO 赋值使用。

以 Attachment 上传对象集合为例，如何实现修改？

当前端输入文字，以及上传了新的文件，这是来自两个不同来源，我们都要组合在同一个领域对象中，因为这时领域对象是作为输入参数 DTO 使用的。

比如帖子修改时，ForumMessage 虽然是一个领域实体，但是也作为前端传入参数的 DTO，其内部值是由 Takia 框架通过 MessageForm 进行对应 **getXXX** 和 **setXXX** 复制而来。

它们之间复制关系如下：

1. 当显示在用户界面，以供下步修改时，这时显示的 FormMessage 来自缓存或仓储。这时 FormMessage 的 getXXX 对应 MessagForm 的 setXXX，注意 set 和 get 后面的 XXX 一定完全相同才能复制，这和 set/get 操作的字段 xxx 没有关系。
2. 当用户提交已经修改后的表单时，用户提交新数据都在 MessageForm 中，MessageForm 的 getXXX 对应于 FormMessage 的 setXXX，也必须 XXX 两个都相同，包括大小写才可以赋值拷贝，这样包含用户提交新数据的 FormMessage 被打包成 EventModel 传入 Service 中。
因此，在 Service 我们只要通过 EventModel.getModelIF() 就可以获得用户提交的表单内容。

现在，ForumMessage 的 update 方法已经接受到了前端提交修改新命令，update 方法是 ForumMessage 的作为 DTO 的新对象，取名 forumMessagePostDTO 等。

在 update 需要做的就是，将 forumMessagePostDTO 替换本身自己，至于怎么替换，就要看业务功能，比如上传文件需要进行完全替换，而帖子属性需要进行 merge 融合；而帖子可能被从一个论坛版块迁移到另外一个，那么 ForumMessage 其对应的 Form 则要更换。

根据不同业务功能和子对象性质不同，有不同的实现方法，比如上传文件在读写两个方面考虑，读时需要懒加载，写时是完全替换；而帖子属性则是读时需要懒加载，写时是 Merge 融合。至于为什么需要懒加载，就看其读取仓储是否费时费力，从数据库加载大文件很耗时，而帖子属性如果有很多，一连串加载也会耗时。而加载 FormMessage 的一个新的 Forum，迁移版块则不会耗时，不需要懒加载，直接发出领域事件即可。

对于需要懒加载，我们设计专门一个对象，继承 LazyLoader 即可，前面懒加载章节有专门描述。下面我们进行实战的优化修改考虑。

对于上传文件创建一个新的值对象 AttachmentsVO，继承 LazyLoader，在其 get 方法实现懒加载：

```
public class AttachmentsVO extends LazyLoader {

    private final long messageId;
    //懒加载角色
    private LazyLoaderRole lazyLoaderRole;

    // for upload files lazyload
    private volatile Collection uploadFiles;

    // 构造函数专门针对读取 激活懒加载for read or load
    public AttachmentsVO(long messageId, LazyLoaderRole lazyLoaderRole)
    {
        super();
        this.messageId = messageId;
        this.lazyLoaderRole = lazyLoaderRole;
    }
}
```

```

//该构造函数专门针对写，也就是本对象作为前端传入的DTO for be written
//适合帖子被创建时使用
public AttachmentsVO(long messageId, Collection uploadFiles) {
    super();
    this.messageId = messageId;
    this.uploadFiles = uploadFiles;
}

//当上传文件需要被修改替换，使用该方法，使用于帖子修改
public void setUploadFiles(Collection uploadFiles) {
    // this.uploadFiles = uploadFiles;
    this.uploadFiles = null;
}

//关键方法，通过该方法获得上传图片的懒加载
public Collection getUploadFiles() {
    if (this.uploadFiles == null && lazyLoaderRole != null) {
        // return result cannot be null, can be a ArrayList that
isEmpty()
        // blocked until return result, display attachment lifecycle
is
        // same as messageVO;
        this.uploadFiles = (Collection) super.loadBlockedResult();
    }
    return uploadFiles;
}

```

从上面看出，我们通过 AttachmentsVO 的两个不同构造函数，让 AttachmentsVO 适用于两个不同的流程读和写。

为什么要区分读和写呢？这是因为读时使用懒加载，而写时则不需要，AttachmentsVO 这时只是一个实实在在的 DTO，作为数据携带者，将其中数据传送到即可。当 AttachmentsVO 作为一个被读取时，其是和其父对象 ForuMessage 驻留在内存缓存中的，如果 ForuMessage 的 update 被触发，需要对其自身进行修改时，就要调用 AttachmentsVO 的 setUploadFiles 方法，该方法内部并没有品尝的 this.xxx = xxx 操作，而是将 this.xxx 清空，下次调用 getXXX 时，发现 this.xxx 为空，将激活懒加载进行加载。

所以，AttachmentsVO 在读取时，实际自身要维护自己和仓储一致性。

如果换成 MessagePropertyVO 也就是帖子属性时，这里 setXXX 方法根据业务不同有区别，属性修改是 merge 融合，和原来的值进行修改合并，那么这里 setXXX 方法就要进行这种实现，然后通过 getXXX 获得的是进行融合后的结果，这个结果通过领域事件发送到仓储更新新结果进行更新数据库即可。

总结如下：

第一.：在 Takia 框架应用中，一个实体模型有两个创建地点，在仓储被创建后，将数据库数据赋值，然后被缓存，也就是变成内存中实体对象；还有一个创建地点在表现层，

用来作为数据对象 DTO，主要是拷贝用户提交的表单数据 Form。

为了表达一个实体类这两种用法，从实体聚合根到子对象都要提供两个构造函数，比如聚合根 ForumMessage 的构造函数：

```
// created from repository that will be in memory, it is Entity
public ForumMessage(Long messageId) {
    this.messageId = messageId;
    this.messageVO = new MessageVO();
}

// created in UI, catch the messageForm's copy data, it is DTO.
public ForumMessage() {
    this.messageVO = new MessageVO();
    this.account = new Account();
    this.attachmentsVO = new AttachmentsVO(new Long(0), new
ArrayList());
    this.messagePropertysVO = new MessagePropertysVO(new Long(0), new
ArrayList());
}
```

带有 MessageId 的是表示是才仓储创建的正常的实体对象，其中子对象采取懒加载赋值，即使用时才赋值。而无参数构造函数的，一般是临时创建，传送数据使用，在其中要将一些子对象都立即要赋值，防止使用或从 MessageForm 互相拷贝数据时出现 InvalidatTargetException 错误。

第二：领域事件在聚合根内部的使用：

如果修改一个对象时，需要对其关联的子对象进行修改，我们都需要在使用领域事件进行修改命令的响应处理，这是一个写命令的处理；

对于读取查询，我们要考虑使用领域事件进行子对象的懒加载，虽然造成一定延迟，但是查询性能提高很多。

当然，如果这两种功能一个对象都需要，那么就要考虑以谁为主，如 AttachmentsVO 以读为主，通过提供两个构造函数，分别实现读的懒加载功能，以及写时作为数据携带 DTO 的功能。

为了显式分辨读和写，建议不要在一个对象中混同两个领域事件角色来分别操作读和写，容易混淆，写入操作角色可直接其聚合根来完成写时的领域事件触发。

第三：对象行为分对内和对外两种。

以上是针对聚合根实体内部关联对象内部修改，对内行为必须由实体自己的方法完成。

如果是其他实体帖子与外部交互行为，对外行为则不能由实体内，必须放在其场景下，可委托实体担任的角色完成。根据 DCI 在场景中实现，比如帖子的创建，是由外部对其创建，这时其还没诞生，不可能对其内部进行操作，那么创建帖子过程很显然在创建场景中实现，创建过程包括其聚合边界生命周期要求一致的其他对象创建。

第四：同步和异步根据一致性高低要求选择。

从领域事件角度来看，领域事件由于是异步，不可能做到在同一个线程内执行逐步一

行顺序代码那样严格先来后到，但是也不可能特别慢，可以用来实现一些业务一致性要求不是很严格的应用。比如帖子修改时帖子从一个论坛搬迁到另外一个论坛，这时我们只要将帖子当前论坛清空，下次根据帖子读取其所在论坛时，使用一个判断语句，如果为空，立即通过领域事件加载，并且通过 **Blocking** 获得其结果。这个 **Blocking** 等待时间可以调整，确实是 1 秒，如果没有获得，显示空也可以，用户也不会太在意，等下次刷新页面时就一般是正确结果了。

对于高一致性的业务，使用 **DCI** 实现同步，其中也可以使用领域事件进行并行操作；对于不是很要求高一致性的，通过懒加载或模型自己发出领域事件进行实现操作。

因为模型自己是无法直接和仓储等技术架构交互的，如果需要交互则使用 **DCI**，否则其行为肯定是有关模型自己内部维护结构性或数据一致性等操作，则可以通过懒加载或模型内部发出领域事件进行操作。

这里有一个疑问，如果模型内部的操作也需要严格同步，怎么办呢？这种情况几乎不存在，因为严格同步就是更新自己，自己在内存中，更新自己是最快的，其他用户获得的也是留住内存中自己，可以使用同步锁对对象中一些方法进行严格的事务操作。

Takia 框架高级使用

在这一章，主要介绍在深入使用 Takia 框架中碰到的一些问题和解决方案，从而达到更加灵活地使用 Takia 框架。

内嵌对象(Embedded Object)缓存设计

请看下面这段代码：

```
public class Category extends Model{
    String Id;
    Product product;    //内嵌包含了一个 Product 对象
}
```

Category 这个 Model 内嵌了 Product 这个 Model，属于一种关联关系。

目前 Takia 框架提供的缺省缓存是扁平式，不是嵌入式或树形的（当然也可以使用 JbossCache 等树形缓存替代），因此，一个 ModelB 对象（如 Product）被嵌入到另外一个 ModelA 对象（如 Category）中，那么这个 ModelB 对象随着 ModelA 对象被 Takia 框架一起缓存。

假设实现 ModelA 已经在缓存中，如果客户端从缓存直接获取 ModelB，缓存由于不知道 ModelB 被缓存到 ModelA 中（EJB3 实体 Bean 中是通过 Annotation 标注字段），那么缓存可能告知客户端没有 ModelB 缓存。那么有可能客户端会再向缓存中加一个新的 ModelB，这样同一个 ID 可能有两份 ModelB，当客户端直接调用的 ModelB 进行其中字段更新，那么包含在 ModelA 中的 ModelB 可能未被更新（因为 ModelA 没有字段更新）。

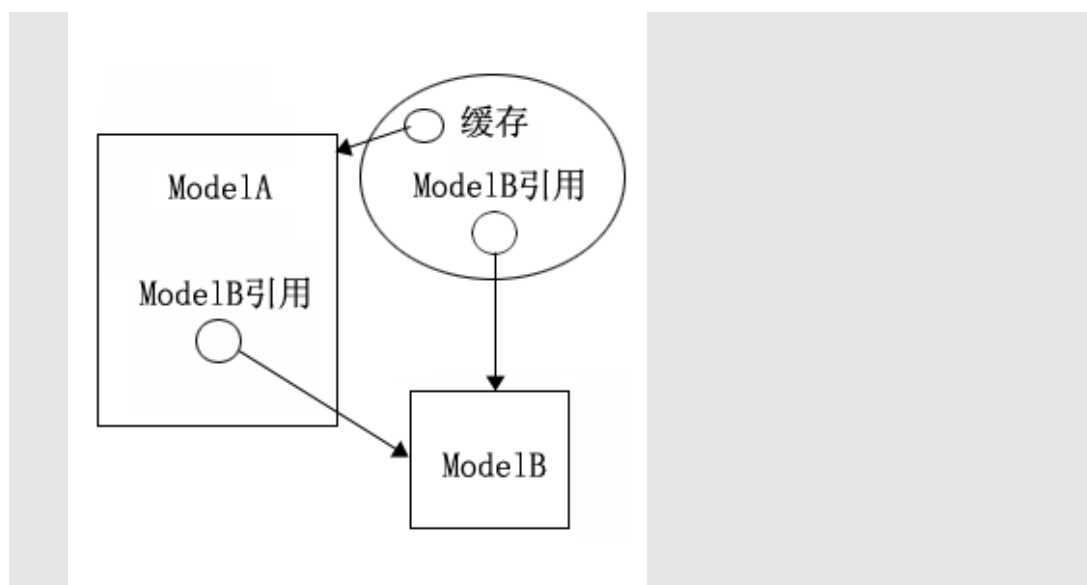
有两种解决方案：

第一，最直接方式，通过手工清除 ModelA 缓存方式来更新，或者耐心等待 ModelA 缓存自动更新。手工清除缓存见下章。

注意：下面这种做法将也会导致不一致现象发生：

在 DAO 层读取数据库。生成 ModelA 时，直接读取数据库将 ModelB 充填。

第二.在进行 ModelA 和 ModelB 的相关操作服务设计时，就要注意保证这两种情况下 ModelB 指向的都是同一个。如下图：



为达到这个目的，只要在 Service 层和 Dao 层之间加一个缓存 Decorator，服务层向 Dao 层调用的任何 Model 对象都首先经过缓存检查，缓存中保存的 ModelA 中的 ModelB 是一个只有 ModelB 主键的空对象，在服务层 getModelA 方法中，再对 ModelA 的 ModelB 进行充实，填满，根据 ModelA 中的 ModelB 的主键，首先再到缓存查询，如果有，则将缓存中 ModelB 充填到 ModelA 的 ModelB 中，这样上图目的就可以实现了。

Forum/ForumMessage 都属于这种情况。

Model 缓存使用

Takia 框架通过两种方式使用 Model 缓存：

CRUD 框架内部使用，如果你使用 Takia 框架提供的 CRUD 功能，那么其已经内置 Model 缓存，而且会即时清除缓存。

通过 DomainCacheInterceptor 缓存拦截器，如果你不使用 Takia 框架的 CRUD 功能，缓存拦截器功能将激活，在向 Service 获取 Model 之前，首先查询当前缓存器中是否存在该 Model，如果有从缓存中获取。当你的 Model 中数值更改后，必须注意自己需要手工清除该 Model 缓存，清除方法如下介绍。

Takia 框架除了提供单个 Model 缓存外，还在持久层 Dao 层提供了查询条件的缓存，例如如果你是根据某个字段按照如何排列等条件进行查询，这个查询条件将被缓存，这样，下次如果有相同查询条件，该查询条件将被提出，与其相关的满足查询条件一些结果（如符合条件总数等）也将被从缓存中提出，节省翻阅数据库的性能开销。

手工访问缓存

在一般情况下，前台表现层通过 getService 方法访问服务层一个服务，然后通过该服务 Service 获得一个 Model，这种情况 Takia 框架的缓存拦截器将自动首先从缓存读取。

但是，有时我们在服务层编码时，需要获得一个 Model，在这种情况下，Takia 框架的缓存拦截器就不起作用，这时可能我们需要手工访问缓存。

因为所有服务类 POJO 都属于 Takia 框架的容器内部组件，这实际是在容器内访问容



器组件的问题。

使用 `ModelContainer` 可以手动查询模型缓存信息。

手工清除缓存

注意：手工清除缓存不是必要的，因为缓存中对象是有存在周期的，这在 `Cache` 中设置的，过一段时间缓存将自动清除那些超过配置时间不用的对象，这样你修改的数据将从数据库重新加载。如果你等不及这些内在变化，可以手工处理：

有两种情况需要手工清除缓存，首先，在持久层的 `Dao` 类中，总是需要手工清除查询条件的缓存（注意不是 `Model` 缓存，是查询条件的缓存），只要在相应的增删改方法中调用 `ModelContainer` 的相关方法既可。

配置 ehcache 作为缓存

请参考 Spring 的 Ehcache 集成文档。

配置 guavacache 作为缓存

请参考 Takia 的 guavacache 集成配置。

配置 memcache 作为缓存

请参考 Takia 的 memcache 集成配置。

全文完