

# 为什么从MVC到DDD，架构的本质是什么

今天要分享的是 MVC 和 DDD 的架构本质，通过由浅入深的介绍讲解和视频带着手把手操作创建工程架构。让无论是学习 MVC 的小白码农还是希望了解更多关于 DDD 内容的老白码农，都可以学习到一点自己需要的内容。

## 一、MVC 架构

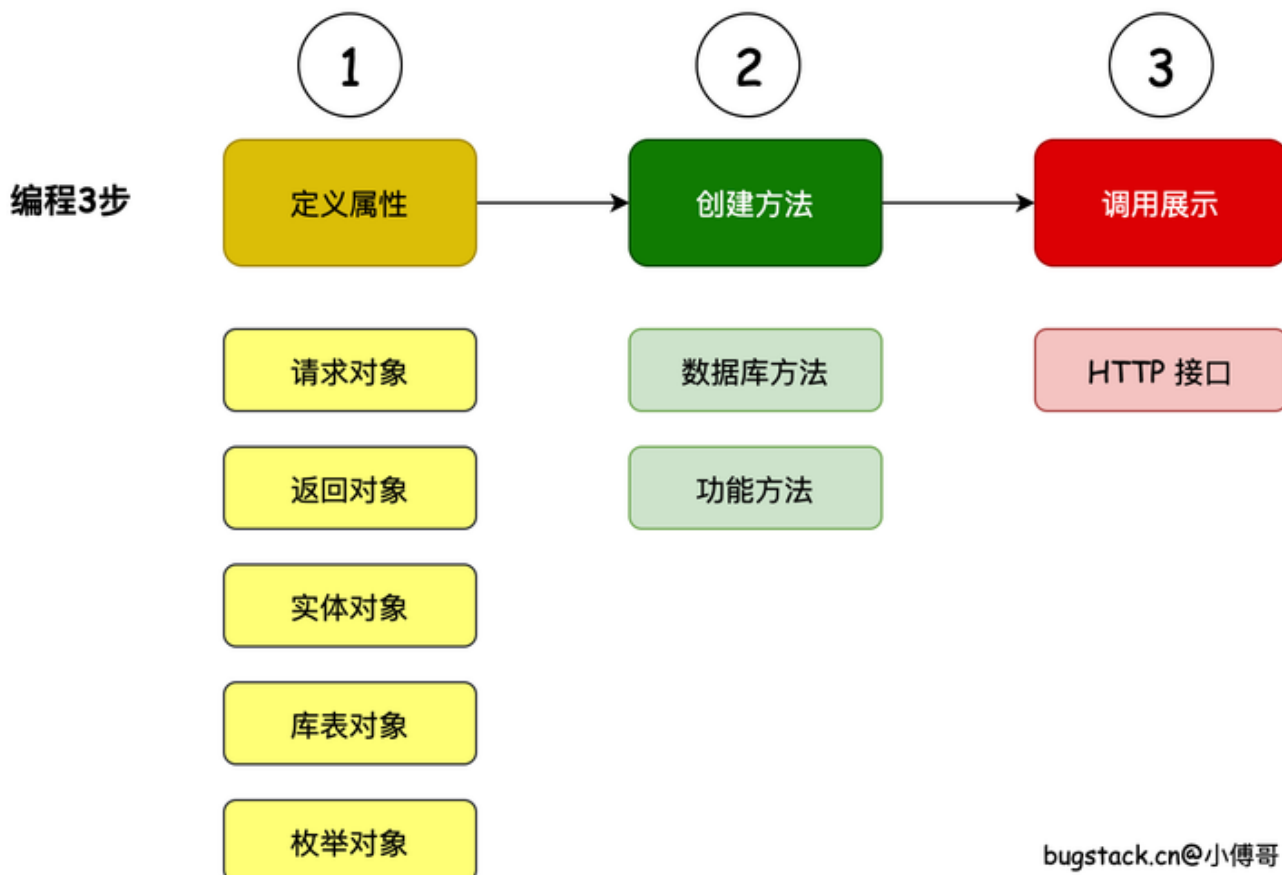
如果我们尝试把编程的复杂架构缩小到最容易理解的程度，那么编程开发其实只做3件事：“定义属性”、“创建方法”、“调用展示”。但因为同类所需的内容较多，如一系列的属性，一堆的方法实现，一组的接口封装，那么就需要合理的把这些内容分配到不同的层次中去实现，因此有了分层架构的设计。

那么本文小傅哥会向大家介绍一套MVC架构的分层设计以及如何创建使用，并提供相应的简单的案例。你可以复制这套架构在自己的场景中使用，也更能方便编程的小白可以更快的上手开发。

**注意：**此套MVC架构模型适合提供HTTP服务的工程架构，适合简单的小场景开发使用。特点：轻便、简单、学习成本低。

### 1. 编程三步

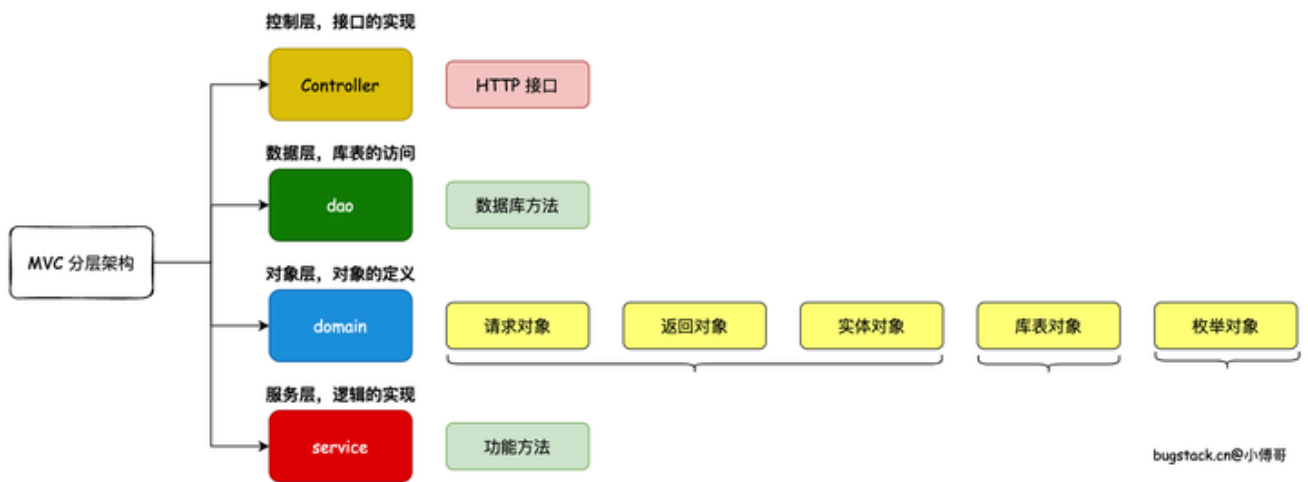
如果说你是一个特别小的玩具项目，你甚至可以把编程的3步写到一个类里。但因为你做的是正经项目，你的各种类：对象类、库表类、方法类，就会成群结队的来。如果你想把这些成群结队的类的内容，都写到一个类里去，那么就是几万行的代码了。——当然你也可以吹牛逼，你一个人做过一个项目，这项目大到啥程度呢。就是有一个类里有上万行代码。



所以，为了不至于让一个类撑到爆，需要把黄色的对象、绿色的方法、红色的接口，都分配到不同的包结构下。这就是你编码人生中所接触到的第一个解耦操作。

## 2. 分层框架

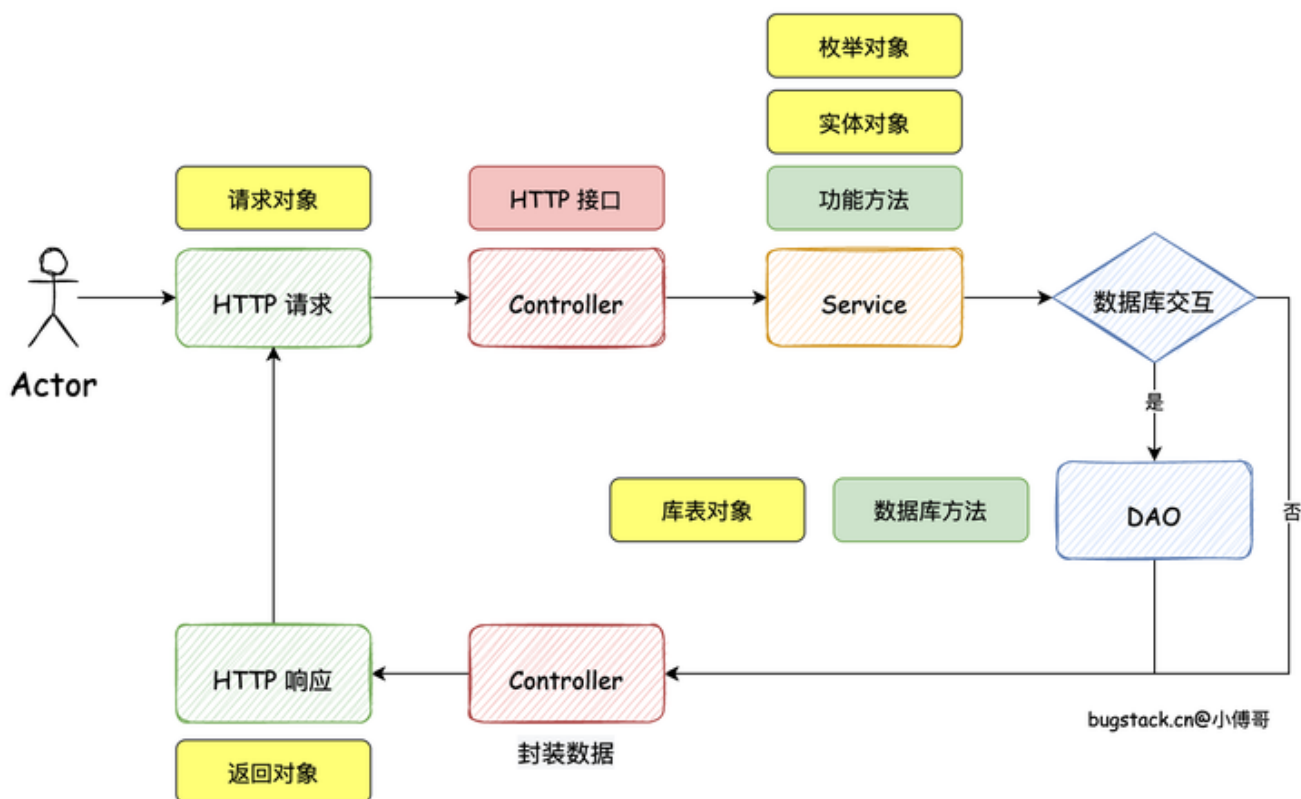
MVC 是一种非常常见且常用的分层架构，主要包括；M - mode 对象层，封装到 domain 里。V - view 展示层，但因为目前都是前后端分离的项目，几乎不会在后端项目里写 JSP 文件了。C - Controller 控制层，对外提供接口实现类。DAO 算是单独拿出来用户处理数据库操作的层。



- 如图，在 MVC 的分层架构下。我们编程3步的所需各类对象、方法、接口，都分配到 MVC 的各个层次中去。
- 因为这样分层以后，就可以很清晰明了的知道各个层都在做什么内容，也更加方便后续的维护和迭代。
- 对于一个真正的项目来说，是没有一锤子买卖的，最开始的开发远不是成本所在。最大的开发成本是后期的维护和迭代。而架构设计的意义更多的就是在解决系统的反复的维护和迭代时，如何降低成本，这也是架构分层的意义所在。

## 3. 调用流程

接下来我们再看下一套 MVC 架构中各个模块在调用时的串联关系；



- 以用户发起 HTTP 请求开始，Controller 在接收到请求后，调用由 Spring 注入到类里的 Service 方法，进入 Service 方法后有些逻辑会走数据库，有些逻辑是直接内部自己处理后就直接返回给 Controller 了。最后由 Controller 封装结果返回给 HTTP 响应。
- 同时我们也可以看到各个对象在这些请求间的一个作用，如：请求对象、库表对象、返回对象。

## 4. 架构源码

## 4.1 环境

- JDK 1.8
- Maven 3.8.6 - 下载安装maven后，本地记得配置阿里云镜像，方便快速拉取jar包。源码中 docs/maven/settings.xml 有阿里云镜像地址。
- SpringBoot 2.7.2
- MySQL 5.7 - 如果你使用 8.0 记得更改 pom.xml 中的 mysql 引用

## 4.2 架构

- 源码: <https://gitcode.net/KnowledgePlanet/road-map/xfg-frame-mvc>
- 树形: `安装 brew install tree`IntelliJ IDEA Terminal 使用 tree`

```

├─ docs
│   └─ mvc.drawio - 架构文档
├─ pom.xml
├─ src
│   └─ main
│       └─ java
│           └─ cn
│               └─ bugstack

```

```

├── xfg
│   ├── frame
│   │   ├── Application.java
│   │   ├── common
│   │   │   ├── Constants.java
│   │   │   └── Result.java
│   │   ├── controller
│   │   │   └── UserController.java
│   │   ├── dao
│   │   │   └── IUserDao.java
│   │   ├── domain
│   │   │   ├── po
│   │   │   │   └── User.java
│   │   │   ├── req
│   │   │   │   └── UserReq.java
│   │   │   ├── res
│   │   │   │   └── UserRes.java
│   │   │   └── vo
│   │   │       └── UserInfo.java
│   │   └── service
│   │       ├── IUserService.java
│   │       └── impl
│   │           └── UserServiceImpl.java
│   └── resources
│       ├── application.yml
│       ├── mybatis
│       │   ├── config
│       │   │   └── mybatis-config.xml
│       │   └── mapper
│       │       └── User_Mapper.xml
│   └── test
│       ├── java
│       │   ├── cn
│       │   │   ├── bugstack
│       │   │   │   ├── xfg
│       │   │   │   │   ├── frame
│       │   │   │   │   │   └── test
│       │   │   │   │       └── ApiTest.java
│       │   └── test
│       └── road-map.sql

```

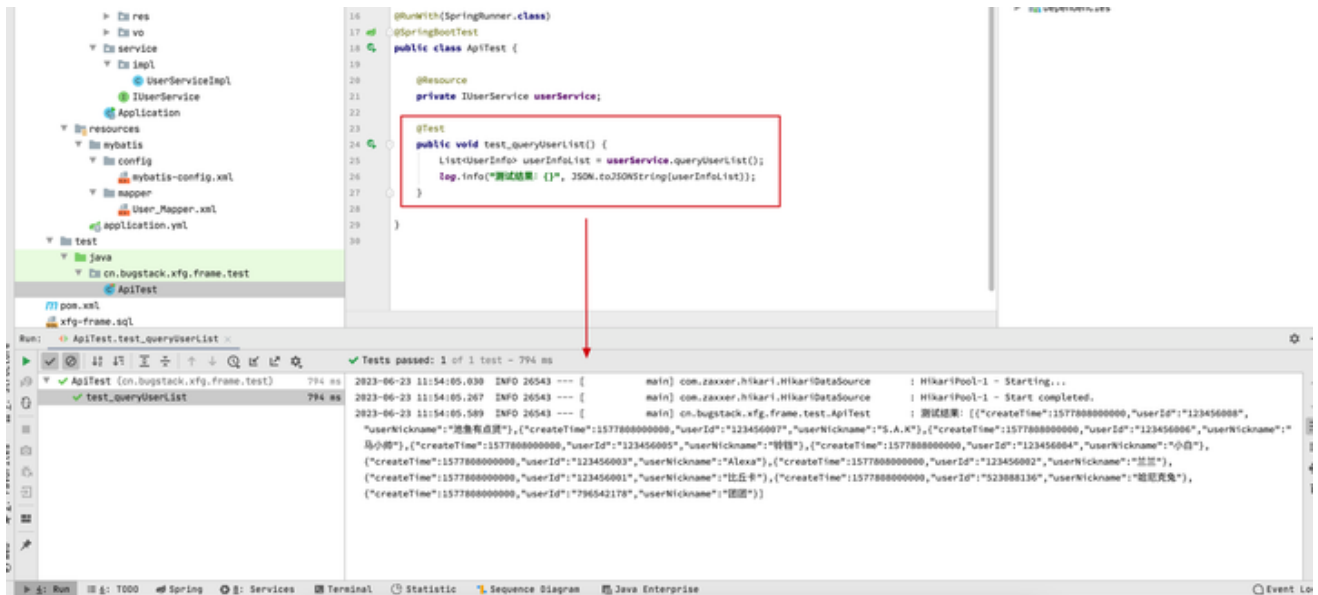
以上是整个工程架构的 tree 树形图。整个工程由 SpringBoot 驱动。

- Application.java 是启动程序的 SpringBoot 应用
- common 是额外添加的一个层，用于定义通用的类
- controller 控制层，提供接口实现。
- dao 数据库操作层
- domain 对象定义层
- service 服务实现层

## 5. 测试验证

- 首先；整个工程由 SpringBoot 驱动，提供了 road-map.sql 测试 SQL 库表语句。你可以在自己的本地mysql上进行执行。它会创建库表。

- 之后；在 application.yml 配置数据库链接信息。
- 之后就可以打开 ApiTest 进行测试了。你可以点击 Application 类的绿色箭头启动工程，使用 UserController 类提供接口的方式调用程序；<http://localhost:8089/queryUserInfo>



- 如果你正常获取了这样的结果信息，那么说明你已经启动成功。接下来就可以对照着MVC的结构进行学习，以及使用这样的工程结构开发自己的项目。

## 二、DDD 架构

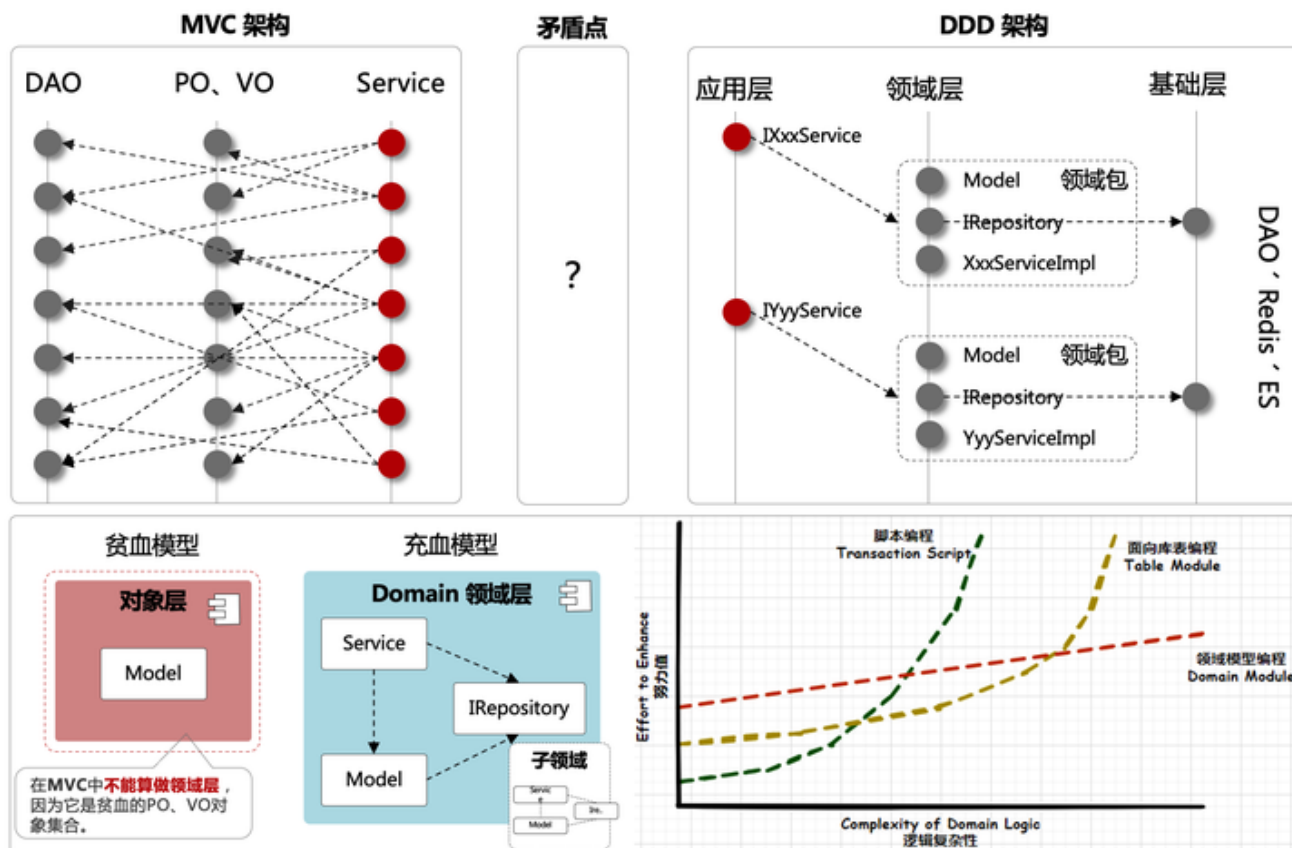
从最早接触 DDD 架构，到后来用 DDD 架构不断的承接项目开发，一次次在项目开发中的经验积累。对 DDD 有了不少的理解。DDD 是一种思想，落地的形态和结构会有不同的方式，甚至在编码上也会有风格的差异。但终期目标就一个；“提供代码的可维护性，降低迭代开发成本。”也是康威定律所述：“任何组织在设计一套系统时，所交付的设计方案在结构上都与该组织的沟通结构保持一致。”

但 DDD 与 MVC 相比的概率较多，贸然用理论驱动代码开发，会让整个工程变得非常混乱，甚至可能虽然是用的 DDD 但最后写出来了一片四不像的 MVC 代码。所以对于程序员来说，先能上手一个工程，在从工程了解理论会更加容易。为此小傅哥想以此文，通过实战编码的方式向大家分享 DDD 架构，并能让大家上手的 DDD 架构。

### 1. 问题碰撞

你用 MVC 写代码，遇到过最大的问题是什么？

简单、容易、好理解，是 MVC 架构的特点，但也正因为简单的分层逻辑，在适配较复杂的场景并且需要长周期的维护时，代码的迭代成本就会越来越高。如图；

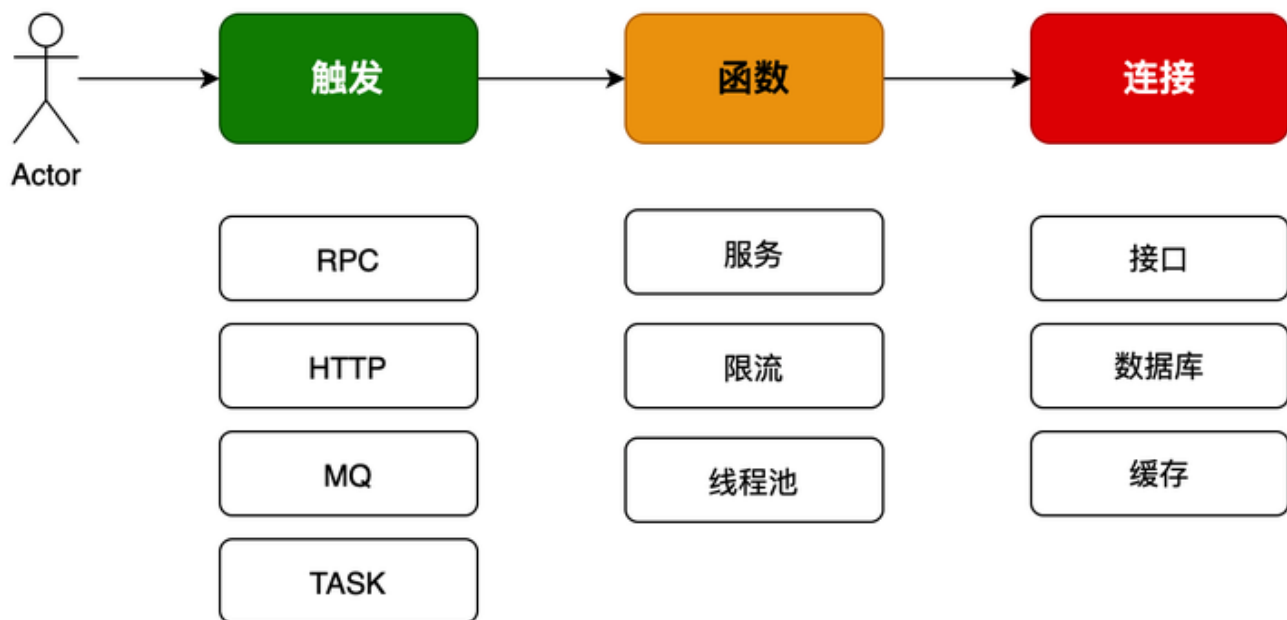


康威定律：任何组织在设计一套系统时，所交付的设计方案在结构上都与该组织的沟通结构保持一致。

- 如果你接触过较大型且已经长期维护项目的 MVC 架构，你就会发现这里的 DAO、PO、VO 对象，在 Service 层相互调用。那么长期开发后，就导致了各个 PO 里的属性字段数量都被撑的特别大。这样的开发方式，将“状态”、“行为”分离到不同的对象中，代码的意图渐渐模糊，膨胀、臃肿和不稳定的架构，让迭代成本增加。
- 而 DDD 架构首先以解决此类问题为主，将各个属于自己领域范围内的行为和逻辑封装到自己的领域包下处理。这也是 DDD 架构设计的精髓之一。它希望在分治层面合理切割问题空间为更小规模若干子问题，而问题越小就容易被理解和处理，做到高内聚低耦合。这也是康威定律所提到的，解决复杂场景的设计主要分为：分治、抽象和知识。

## 2. 简化解理解

在给大家讲解 MVC 架构的时候，小傅哥提到了一个简单的开发模型。开发代码可以理解为：“定义属性 -> 创建方法 -> 调用展示”但这个模型结构过于简单，不太适合运用了各类分布式技术栈以及更多逻辑的 DDD 架构。所以在 DDD 这里，我们把开发代码可以抽象为：“触发 -> 函数 -> 连接”如图；

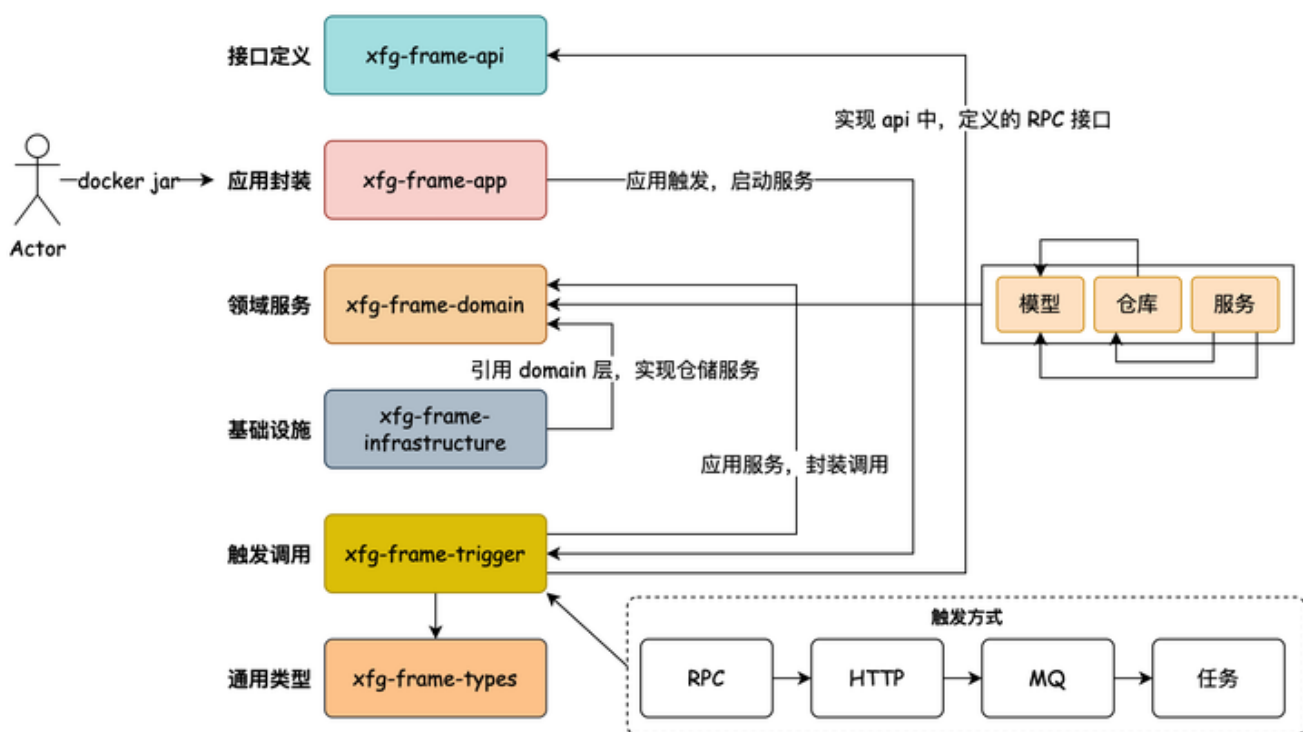


- DDD 架构常用于微服务场景，因此也一个系统的调用方式就不只是 HTTP 还包括；RPC 远程、MQ 消息、TASK 任务，因此这些种方式都可以理解为触发。
- 通过触发调用函数方法，我们这里可以把各个服务都当成一个函数方法来看。而函数方法通过连接，调用到其他的接口、数据库、缓存来完成函数逻辑。

接下来，小傅哥在带着大家把这些所需的模块，拆分到对应的DDD系统架构中。

### 3. 架构分层

如下是 DDD 架构的一种分层结构，也可以有其他种方式，核心的重点在于适合你所在场景的业务开发。以下的分层结构，是小傅哥在使用 DDD 架构多种的方式开发代码后，做了简化和处理的。右侧的连线是各个模块的依赖关系。接下来小傅哥就给大家做一下模块的介绍。





- **接口定义 - xfg-frame-api**: 因为微服务中引用的 RPC 需要对外提供接口的描述信息, 也就是调用方在使用的時候, 需要引入 jar 包, 让调用方好能依赖接口的定义做代理。
- **应用封装 - xfg-frame-app**: 这是应用启动和配置的一层, 如一些 aop 切面或者 config 配置, 以及打包镜像都是在这一层处理。你可以把它理解为专门为了启动服务而存在的。
- **领域封装 - xfg-frame-domain**: 领域模型服务, 是一个非常重要的模块。无论怎么做DDD的分层架构, domain 都是肯定存在的。在一层中会有一个个细分的领域服务, 在每个服务包中会有【模型、仓库、服务】这样3部分。
- **仓储服务 - xfg-frame-infrastructure**: 基础层依赖于 domain 领域层, 因为在 domain 层定义了仓储接口需要在基础层实现。这是依赖倒置的一种设计方式。
- **领域封装 - xfg-frame-trigger**: 触发器层, 一般也被叫做 adapter 适配器层。用于提供接口实现、消息接收、任务执行等。所以对于这样的操作, 小傅哥把它叫做触发器层。
- **类型定义 - xfg-frame-types**: 通用类型定义层, 在我们的系统开发中, 会有很多类型的定义, 包括; 基本的 Response、Constants 和枚举。它会被其他的层进行引用使用。
- **领域编排【可选】 - xfg-frame-case**: 领域编排层, 一般对于较大且复杂的的项目, 为了更好的防腐和提供通用的服务, 一般会添加 case/application 层, 用于对 domain 领域的逻辑进行封装组合处理。

## 4. 架构源码

### 4.1 环境

- JDK 1.8
- Maven 3.8.6
- SpringBoot 2.7.2
- MySQL 5.7 - 如果你使用 8.0 记得更改 pom.xml 中的 mysql 引用

### 4.2 架构

- **源码**: <https://gitcode.net/KnowledgePlanet/road-map/xfg-frame-ddd>
- **树形**: 安装 brew install tree ``IntelliJ IDEA Terminal 使用 tree

```
.
├── README.md
├── docs
│   ├── dev-ops
│   │   ├── environment
│   │   │   └── environment-docker-compose.yml
│   │   ├── siege.sh
│   │   └── skywalking
│   │       └── skywalking-docker-compose.yml
│   ├── doc.md
│   ├── sql
│   │   └── road-map.sql
│   └── xfg-frame-ddd.drawio
├── pom.xml
├── xfg-frame-api
│   ├── pom.xml
│   ├── src
│   │   └── main
│   │       ├── java
│   │       │   └── cn
│   │       └── bugstack
```



```

├── xfg
│   ├── frame
│   │   ├── api
│   │   │   ├── IAccountService.java
│   │   │   ├── IRuleService.java
│   │   │   ├── model
│   │   │   │   ├── request
│   │   │   │   │   ├── DecisionMatterRequest.java
│   │   │   │   ├── response
│   │   │   │   │   ├── DecisionMatterResponse.java
│   │   │   └── package-info.java
│   └── xfg-frame-api.iml
├── xfg-frame-app
│   ├── Dockerfile
│   ├── build.sh
│   ├── pom.xml
│   ├── src
│   │   ├── main
│   │   │   ├── bin
│   │   │   │   ├── start.sh
│   │   │   │   └── stop.sh
│   │   │   ├── java
│   │   │   │   ├── cn
│   │   │   │   │   ├── bugstack
│   │   │   │   │   │   ├── xfg
│   │   │   │   │   │   │   ├── frame
│   │   │   │   │   │   │   │   ├── Application.java
│   │   │   │   │   │   │   │   ├── aop
│   │   │   │   │   │   │   │   │   ├── RateLimiterAop.java
│   │   │   │   │   │   │   │   │   └── package-info.java
│   │   │   │   │   │   │   └── config
│   │   │   │   │   │   │   │   ├── RateLimiterAopConfig.java
│   │   │   │   │   │   │   │   ├── RateLimiterAopConfigProperties.java
│   │   │   │   │   │   │   │   ├── ThreadPoolConfig.java
│   │   │   │   │   │   │   │   ├── ThreadPoolConfigProperties.java
│   │   │   │   │   │   │   └── package-info.java
│   │   │   └── resources
│   │   │   │   ├── application-dev.yml
│   │   │   │   ├── application-prod.yml
│   │   │   │   ├── application-test.yml
│   │   │   │   ├── application.yml
│   │   │   │   ├── logback-spring.xml
│   │   │   │   ├── mybatis
│   │   │   │   │   ├── config
│   │   │   │   │   │   ├── mybatis-config.xml
│   │   │   │   │   └── mapper
│   │   │   │   │   │   ├── RuleTreeNodeLine_Mapper.xml
│   │   │   │   │   │   ├── RuleTreeNode_Mapper.xml
│   │   │   │   │   └── RuleTree_Mapper.xml
│   │   └── test
│   │       ├── java
│   │       │   ├── cn
│   │       │   │   ├── bugstack

```

```

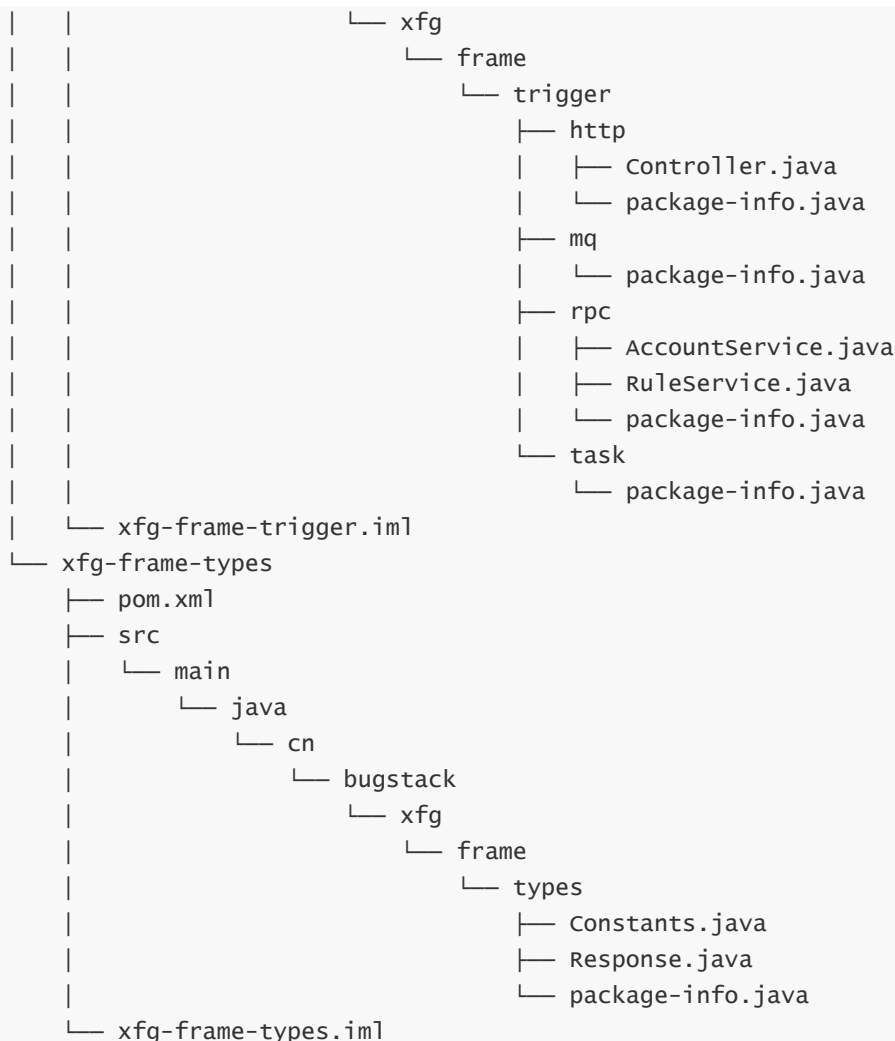
graph TD
    xfg --> frame
    frame --> test
    test --> ApiTestJava[ApiTest.java]
    xfg --> xfgFrameAppIml[xfg-frame-app.iml]
    xfg --> xfgFrameDddIml[xfg-frame-ddd.iml]
    xfg --> xfgFrameDomain
    xfgFrameDomain --> pomXml[pom.xml]
    xfgFrameDomain --> src
    src --> main
    main --> java
    java --> cn
    cn --> bugstack
    bugstack --> xfg
    xfg --> frame
    frame --> domain
    domain --> order
    order --> model
    model --> aggregates
    aggregates --> OrderAggregateJava[OrderAggregate.java]
    model --> entity
    entity --> OrderItemEntityJava[OrderItemEntity.java]
    entity --> ProductEntityJava[ProductEntity.java]
    entity --> packageInfoJava[package-info.java]
    entity --> valobj
    valobj --> OrderIdVOJava[OrderIdVO.java]
    valobj --> ProductDescriptionVOJava[ProductDescriptionVO.java]
    valobj --> ProductNameVOJava[ProductNameVO.java]
    order --> repository
    repository --> IOrderRepositoryJava[IOrderRepository.java]
    repository --> packageInfoJava
    order --> service
    service --> OrdersServiceJava[OrdersService.java]
    service --> packageInfoJava
    domain --> rule
    rule --> model
    model --> aggregates
    aggregates --> TreeRuleAggregateJava[TreeRuleAggregate.java]
    model --> entity
    entity --> DecisionMatterEntityJava[DecisionMatterEntity.java]
    entity --> EngineResultEntityJava[EngineResultEntity.java]
    model --> packageInfoJava
    model --> valobj
    valobj --> TreeNodeLineVOJava[TreeNodeLineVO.java]
    valobj --> TreeNodeVOJava[TreeNodeVO.java]
    valobj --> TreeRootVOJava[TreeRootVO.java]
    rule --> repository
    repository --> IRuleRepositoryJava[IRuleRepository.java]
    repository --> packageInfoJava
    rule --> service
    service --> engine
    engine --> EngineBaseJava[EngineBase.java]
    engine --> EngineConfigJava[EngineConfig.java]

```

```

| | | | | EngineFilter.java
| | | | | └─ impl
| | | | |     └─ RuleEngineHandle.java
| | | | | └─ logic
| | | | |     └─ BaseLogic.java
| | | | |     └─ LogicFilter.java
| | | | |         └─ impl
| | | | |             └─ UserAgeFilter.java
| | | | |             └─ UserGenderFilter.java
| | | | | └─ package-info.java
| | | └─ user
| | |     └─ model
| | |         └─ valobj
| | |             └─ UserVO.java
| | |     └─ repository
| | |         └─ IUserRepository.java
| | |     └─ service
| | |         └─ UserService.java
| | |         └─ impl
| | |             └─ UserServiceImpl.java
| | └─ xfg-frame-domain.iml
└─ xfg-frame-infrastructure
    └─ pom.xml
    └─ src
        └─ main
            └─ java
                └─ cn
                    └─ bugstack
                        └─ xfg
                            └─ frame
                                └─ infrastructure
                                    └─ dao
                                        └─ IUserDao.java
                                        └─ RuleTreeDao.java
                                        └─ RuleTreeNodeDao.java
                                        └─ RuleTreeNodeLineDao.java
                                    └─ package-info.java
                                    └─ po
                                        └─ RuleTreeNodeLineVO.java
                                        └─ RuleTreeNodeVO.java
                                        └─ RuleTreeVO.java
                                        └─ UserPO.java
                                    └─ repository
                                        └─ RuleRepository.java
                                        └─ UserRepository.java
                                └─ xfg-frame-infrastructure.iml
└─ xfg-frame-trigger
    └─ pom.xml
    └─ src
        └─ main
            └─ java
                └─ cn
                    └─ bugstack

```



以上是整个工程架构的 tree 树形图。整个工程由 xfg-frame-app 模的 SpringBoot 驱动。这里小傅哥在 domain 领域模型下提供了 order、rule、user 三个领域模块。并在每个模块下提供了对应的测试内容。这块是整个模型的重点，其他模块都可以通过测试看到这里的调用过程。

## 4.3 领域

一个领域模型中包含3个部分；model、repository、service 三部分；

- model 对象的定义
- repository 仓储的定义
- service 服务实现

以上3个模块，一般也是大家在使用 DDD 时候最不容易理解的分层。比如 model 里还分为；valobj - 值对象、entity 实体对象、aggregates 聚合对象；

- **值对象**：表示没有唯一标识的业务实体，例如商品的名称、描述、价格等。
- **实体对象**：表示具有唯一标识的业务实体，例如订单、商品、用户等；
- **聚合对象**：是一组相关的实体对象的根，用于保证实体对象之间的一致性和完整性；

关于model中各个对象的拆分，尤其是聚合的定义，会牵引着整个模型的设计。当然你可以在初期使用 DDD 的时候不用过分在意领域模型的设计，可以把整个 domain 下的一个个包当做充血模型结构，这样编写出来的代码也是非常适合维护的。

## 4.4 环境(开发/测试/上线)

源码: `xfg-frame-ddd/pom.xml`

```
<profile>
  <id>dev</id>
  <activation>
    <activeByDefault>true</activeByDefault>
  </activation>
  <properties>
    <profileActive>dev</profileActive>
  </properties>
</profile>
<profile>
  <id>test</id>
  <properties>
    <profileActive>test</profileActive>
  </properties>
</profile>
<profile>
  <id>prod</id>
  <properties>
    <profileActive>prod</profileActive>
  </properties>
</profile>
```

- 定义环境; 开发、测试、上线。

































源码: `xfg-frame-app/application.yml`

```
spring:
  config:
    name: xfg-frame
  profiles:
    active: dev # dev、test、prod
```

- 除了 pom 的配置, 还需要在 application.yml 中指定环境。这样就可以对应的加载到; `application-dev.yml`、`application-prod.yml`、`application-test.yml` 这样就可以很方便的加载对应的配置信息了。尤其是各个场景中切换会更加方便。

## 4.5 切面

一个工程开发中, 有时候可能会有很多的统一切面和启动配置的处理, 这些内容都可以在 xfg-frame-app 完成。

- ▼  xfg-frame-app
  - ▼  src
    - ▼  main
      - ▼  bin
        -  start.sh
        -  stop.sh
      - ▼  java
        - ▼  cn.bugstack.xfg.frame
          - ▼  aop
            -  package-info.java
            -  RateLimiterAop
          - ▼  config
            -  package-info.java
            -  RateLimiterAopConfig
            -  RateLimiterAopConfigProperties
            -  ThreadPoolConfig
            -  ThreadPoolConfigProperties
      -  Application
    - ▼  resources
      - ▼  mybatis
        - ▶  config
        - ▶  mapper
      -  application.yml
      -  application-dev.yml
      -  application-prod.yml
      -  application-test.yml
      -  logback-spring.xml
    - ▶  test
    -  build.sh
    -  Dockerfile
    -  pom.xml
    -  xfg-frame-app.iml

源码: `cn.bugstack.xfg.frame.aop.RateLimiterAop`

```
@Slf4j
@Aspect
public class RateLimiterAop {

    private final long timeout;
    private final double permitsPerSecond;
    private final RateLimiter limiter;

    public RateLimiterAop(double permitsPerSecond, long timeout) {
        this.permitsPerSecond = permitsPerSecond;
        this.timeout = timeout;
        this.limiter = RateLimiter.create(permitsPerSecond);
    }

    @Pointcut("execution(* cn.bugstack.xfg.frame.trigger...*(..))")
    public void pointCut() {
    }

    @Around(value = "pointCut()", argNames = "jp")
    public Object around(ProceedingJoinPoint jp) throws Throwable {
        boolean tryAcquire = limiter.tryAcquire(timeout, TimeUnit.MILLISECONDS);
        if (!tryAcquire) {
            Method method = getMethod(jp);
            log.warn("方法 {}.{} 请求已被限流, 超过限流配置[{}]/秒",
method.getDeclaringClass().getCanonicalName(), method.getName(), permitsPerSecond);
            return Response.<Object>builder()
                .code(Constants.ResponseCode.RATE_LIMITER.getCode())
                .info(Constants.ResponseCode.RATE_LIMITER.getInfo())
                .build();
        }
        return jp.proceed();
    }

    private Method getMethod(JoinPoint jp) throws NoSuchMethodException {
        Signature sig = jp.getSignature();
        MethodSignature methodSignature = (MethodSignature) sig;
        return jp.getTarget().getClass().getMethod(methodSignature.getName(),
methodSignature.getParameterTypes());
    }
}
```

## 使用

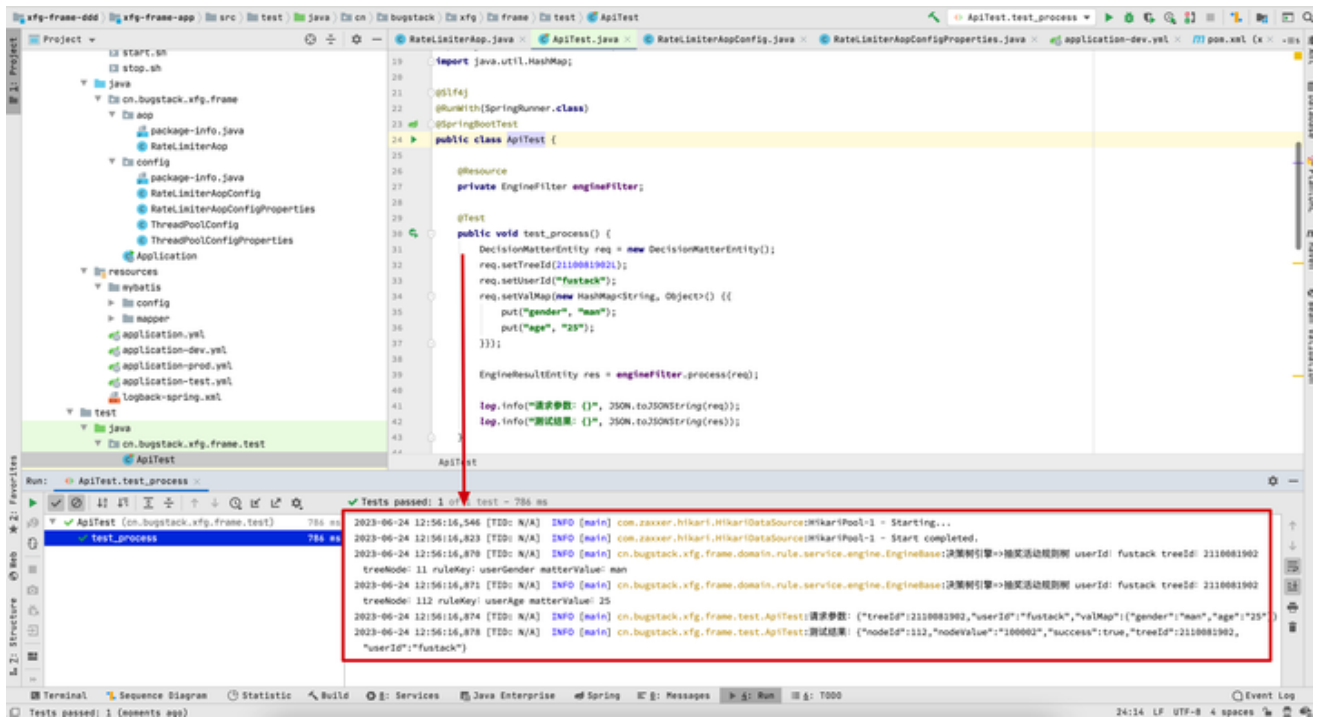
```
# 限流配置
rate-limiter:
  permits-per-second: 1
  timeout: 5
```

- 这样你所有的通用配置, 又和业务没有太大的关系的, 就可以直接写到这里了。—— 具体可以参考代码。



## 5. 测试验证

- 首先；整个工程由 SpringBoot 驱动，提供了 road-map.sql 测试 SQL 库表语句。你可以在自己的本地mysql上进行执行。它会创建库表。
- 之后；在 application.yml 配置数据库链接信息。
- 之后就可以打开 ApiTest 进行测试了。你可以点击 Application 类的绿色箭头启动工程，使用触发器里的接口调用测试，或者单元测试RPC接口，小傅哥也提供了泛化调用的方式。



- 如果你正常获取了这样的结果信息，那么说明你已经启动成功。接下来就可以对照着DDD的结构进行学习，以及使用这样的工程结构开发自己的项目。