整洁架构演进之路——京东广告投放平台实战

原创 广告研发 赵嘉铎 京东零售技术 2024年09月23日 11:10 北京

01 前言

从去年开始京东广告投放系统做了一次以领域驱动设计为思想内核的架构升级，在深入理解DDD思想的同时，我们基于广告投放业务的本质特征大胆地融入了自己的理解和改造。新架构是从设计思想到落地框架都进行了彻底的革新，涉及内容比较多，因此我们希望通过一系列文章循序渐进地阐述本次架构升级的始末。新架构并不是一日而成的，而是经过了多次架构升级的演进，因此我们将本文作为该系列的第一篇文章，先让大家通过广告投放平台的架构演进历程来了解新架构的设计初衷。

如前言所述，本文主要聚焦于广告投放系统历代代码架构的演进历程，我们也不希望本文的篇幅过于冗长，因此对新架构中具体框架及API的说明浅尝辄止，我们会在本系列接下来的数篇文章中逐步给出愈加具象的描述。

02 什么是好的代码架构

大家都清楚在当前的工作中我们所面临的主要矛盾是“越来越多的多场景化复杂业务需求与有限的研发人力之间的矛盾”。而要解决这一矛盾，就要求我们的系统能做到：**设计易拓展、代码易复用、逻辑易传承、运行更稳定**。这看起来像是一句空喊的口号，但其实每一个特性都有具体的要求：

•设计易拓展

一个好的架构应该能够实现业务与技术组件的分离，使设计者能够专注于业务流程，以填空的方式直接套用开箱即用的组件、框架和解决方案，不必进行大量的重复设计；另外好的架构也能够引导设计者完成最小子问题的正交分解，将设计者从错综复杂的上层业务逻辑中拯救出来，逐个击破，降低需求的复杂度和理解成本。

•代码易复用

一个好的架构应该有良好的分层，强调正交子模块的拆分与封装，同层原子模块之间避免互相依赖和耦合，让上层系统能够轻松实现底层业务逻辑的组合复用，以O(n1+n2+...+nm)的实现复杂度支撑O(n1*n2*...*nm)业务复杂度；另外我们的业务逻辑是建立在数据之上的，一个封装良好的代码架构在实现业务逻辑复用的同时应该有健全的数据模型维护和共享机制，避免同一个数据对象的重复查询，并能够轻松通过批量操作降低系统的I/O负载。

•逻辑易传承

我们历史上多次尝试通过维护文档的方式来建立业务知识库，但都以失败告终了。在这个过程中我们意识到业务功能都是由我们的代码承接的，它天然具备业务知识库的功能。因此一个好的代码架构不仅能够实现业务功能，而且要承担起传递业务知识的职责：当有新同学加入时，代码能够以最直接的方式帮助他快速建立起对整个业务的宏观认知，而进行具体的需求开发时，又能够按图索骥，快速定位改动点并深入了解其业务细节。

•运行更稳定

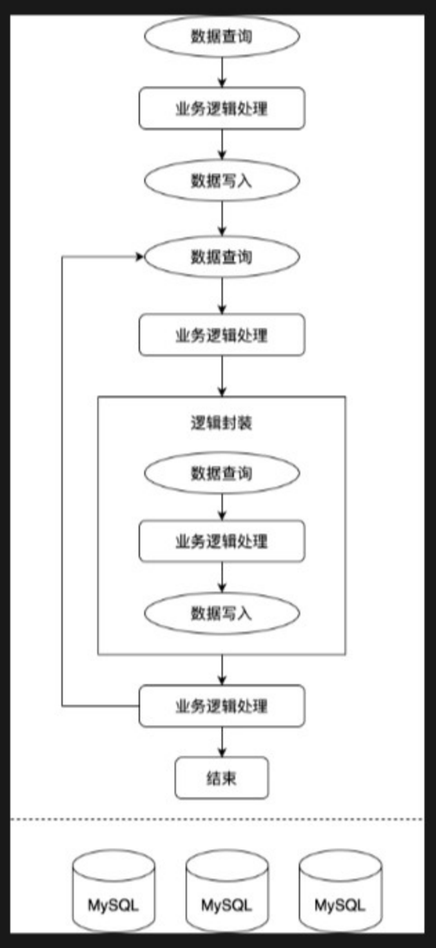
一个好的代码架构在面对多场景化的需求时，可以做到场景隔离，避免不同场景的特有逻辑之间互相耦合干扰，出现一个场景需求上线后影响其他业务场景的问题；除此之外，一个好的架构应该通过设计良好的框架和标准模板在有益的程度上对开发者的编码行为进行约束，把规范框架化，而不是过多的依赖人治和code review来实现规范统一。

03 架构演进之路

在上一章节列举出来的特质也是评判一个架构优劣的标准，而我们新的架构方案也正是在一次次为了实现这些目标而采取的摸索中逐渐成型的。在接下来的几个章节中，我们将从最早期的代码架构开始，逐代剖析架构演进的历程，通过这种方式让大家了解每一次改进背后的设计动机和思路，从而更好地理解新架构的设计思想，也为大家推动架构向下一代演进打好基础。

第一代：没有架构的架构

最开始的时候我们的架构如下图所示，这也是我们目前最常见一种代码架构。可以看出它的特点就是“简单”，没有过多的封装和设计，平铺直叙，数据查询和业务逻辑处理互相交织，是面向数据库编程的典型案例。这种架构在早期场景单一、需求简单的阶段可以快速实现功能，没有多余的设计成本，但是随着业务的发展，系统服务的场景越来越多，这套架构就变得越来越不简单了。



问题主要体现在两个方面：

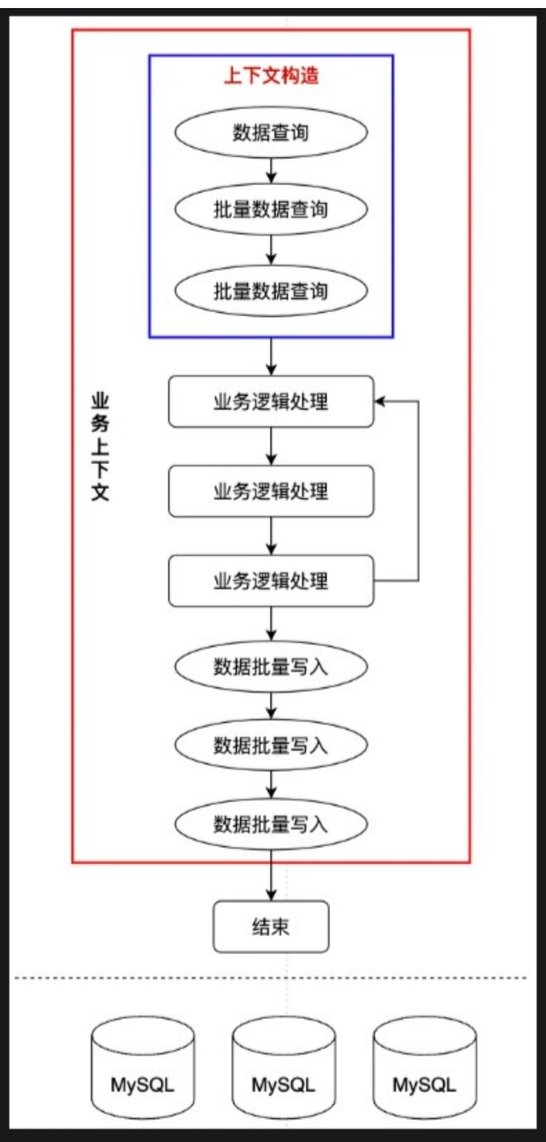
1.由于大家习惯“打补丁”式的开发，来了一个业务需求就在现有的流程中增加一个if...else分支，然后直接在新分支内实现业务逻辑。当业务流程积攒的足够冗长时，就很容易忽视前置流程已经查询好的数据对象，造成数据重复查询。同时为了实现逻辑的复用，我们开始把一些常用逻辑封装为单独的方法，然后在上层业务流程中直接调用，然而我们在封装底层方法往往会把数据的获取逻辑封装下来，这进一步加剧了数据重复查询的问题，在有循环调用的场景中这个问题会更加突出。另外这种逻辑的复用方式还会造成数据库访问碎片化，我们很难利用批量操作的优势优化系统性能。在最近刚结束的大促中，我们前期暴露的几个性能问题基本都是这种模式导致的。

2.除了性能问题之外，由于不同业务场景的逻辑互相交织，代码分支判断逻辑缺少统一的规划，if分支层层嵌套，导致我们的代码逻辑圈复杂度不断飙升，本来应该通用的逻辑对不同场景的适配性越来越低。渐渐的，我们发现新增需求的开发越来越“不简单”了：在试图复用一段看起来相似的代码逻辑时会有很多纠结和不尽人意的地方，对代码执行流程的认知也不似以往那么清晰了，为了防止对旧的业务流程造成影响，我们开始增加更多的if分支，这反过来进一步加剧了情况的恶化，于是我们的代码中充斥着重复代码、多达5、6层的嵌套...

为了缓解旧架构中的这些问题，我们引入了上下文机制，尝试将数据的查询逻辑与业务流程分离开来，由此引出了第二代基于上下文机制的代码架构。

第二代：略有改善的上下文机制

上下文主要是为了解决数据重复查询问题引入的，思路特别朴素，就是把一个完整业务流程中要用到的全部数据提前在方法一开始就查询好，并做好校验。查询出来的数据对象保存到一个上下文对象中，这个上下文对象会贯穿整个业务流程，业务逻辑中需要用得到底层数据实体的时候统一从上下文对象中获取。



所有的数据集中在“上下文构造”步骤中查询，整个业务流程运行在上下文对象中

通过上下文的引入，我们基本上解决了数据重复查询的问题，另外我们数据的提前集中查询也有助于启发我们主动通过数据库批量查询进一步提升系统性能。而且上下文构造的过程其实也是数据校验的过程，通过上下文的提前构建，我们在一定程度上实现了预校验的逻辑，从而可以提前发现异常数据，避免写入脏数据和不必要的数据回滚操作。

上下文的引入其实并不算什么架构上的改进，它主要是解决了数据对象重复查询的问题，但是也引入了一些新的痛点，首先就是我们的数据模型中数据对象往往比较多且关系复杂，这导致我们的上下文构造逻辑十分冗长。而且同一个业务域内不同的接口使用的上下文对象中属性有较大重叠，但是也有各自的差异，因此这些上下文对象的构造逻辑又开始出现大量的重复编码或者混乱的封装。比如询量单的新建接口与修改接口对应的上下文中80%的属性是相同的，这些属性的查询和关联逻辑造成了大量的重复编码。除了重复编码问题之外，上下文机制也并没有从根本上解决多场景下业务流程差异复杂度高的问题。

第三代：数据模型与业务模型的分离

在第二代架构中我们虽然将数据对象的查询集中到了上下文构造步骤中执行，但是上下文对象的定义是和接口方法绑定的。对外暴露多少服务我们就会定义多少上下文对象，甚至不同的场景也会有各自的上下文构造逻辑，此时系统的数据模型依然隐藏在了具体的业务逻辑中。

在一次次的改进尝试中，我们逐渐意识到多场景化的业务特性赋予我们一个动态的业务模型（或者说业务规则集），但是我们的数据模型却是静态的，数据模型的多场景化程度远小于业务规则的多场景化程度，即：同一个功能模块在不同场景下的业务规则存在差异，但却始终在操作同一套数据模型。有些同学可能会对这一结论产生质疑：在不同的场景下我们对数据对象的构造也是不同的，比如只有快车的单元下才会有关键词，京X的单元上绑定的是应用集，而直投单元上绑定的是流量包等等，这些例子是不是都说明我们的数据模型也在随业务规则一起动态变化着呢？对于这个问题我们需要“细品”一下：“数据对象属性值的设置和校验”到底是属于业务模型的范畴还是数据模型的范畴？其实我们所说的数据模型指的是实体及实体之间的关系， 不论某个产品线或计划类型是否会去设置某个子属性的值，只要我们的数据模型完成了定义，那么在任何场景下数据模型的中实体的定义及实体之间的关系都是不变的，实体只要定义出来，它会一直在那，只是某些场景下其属性值为null而已。而实体属性值的设置逻辑则是典型的业务模型的范畴。

在明确了“多场景化的动态业务模型是建立在一个相对静态的数据模型之上”这一本质之后，为了解决上下文对象构造复杂度高及重复编码的问题，我们需要做的就是数据模型的分离和下沉，为此我们引入了领域驱动设计思想中的“聚合”概念。在这篇文章里我们不需要教条地引用DDD中关于聚合的定义，它的含义可以通俗的理解为：一组关联密切且关系明确的实体或值对象的集合，一个聚合通常会支撑着一个功能极其内聚的上层业务模块。一个聚合中会定义唯一个聚合根对象，聚合根是整个聚合中实体操作的中心，聚合中的全部实体都可以通过聚合根直接或间接的访问到。聚合根通常并不难确定，比如计划聚合的聚合根自然就是Campaign实体，我们可以直接通过Campaign聚合根对象直接引用到计划下的预算、投放时段等子实体信息。

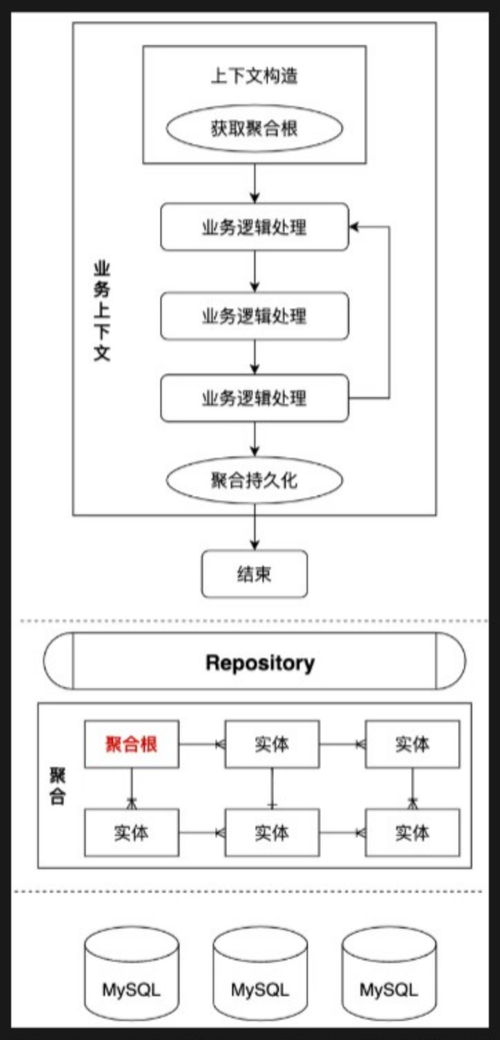
将聚合的概念落地到代码架构中我们需要做以下升级：

1.根据业务流程设计合理的数据模型，需要注意的是数据模型中的实体并不一定要与底层的库表一一对应，而是应该从业务本质出发完成实体划分和定义，另外在模型中也需要体现实体之间的关联关系。

2.在业务流程和底层数据库之间增加一个聚合层，在这一层中将第一步设计的数据模型定义为Java对象，其中实体之间的关系则转化为类与属性的关系。比如AdGroup领域对象内属性除了体现ad_group表中定义的字段之外，也定义了单元下的人群、流量包、创意列表等子实体对应的属性。

3.上层的业务流程对聚合中实体的访问和修改都是通过聚合根实现的，而要想获取聚合根则必须通过聚合层暴露出来的Repository接口。

第三步提到的Repository层接口是完全面向数据模型定义的，几乎与业务无关，通常不会为某个特殊的业务场景定义专用的数据查询或写入方法，它定义的都是通用的数据访问接口，让上层业务以声明式的方法获取所需的聚合根对象（或集合）。Repository的将数据对象的查询和实体关系的组装逻辑屏蔽在其接口实现中，上层业务不需要再次执行聚合根下子实体对象的查询和关联逻辑。



引入聚合后上下文的构造和数据的写入流程得以极大地简化

从上图可以看出，由于上下文中的很多数据对象都被转移到了聚合中，之前繁琐的数据查询和关联逻辑被分离下沉到了Repository的实现中，业务模型中不同的服务接口可以直接复用Repository中沉淀的数据查询和组装逻辑，上下文构造得以极大的精简，重复编码问题也得到了根本性的解决，体现了我们架构目标中“代码易复用”的要求。

除了更加灵活和优雅的复用数据查询和组装逻辑之外，聚合的引入让我们实现了数据模型和业务模型的分离，聚合层几乎与业务流程无关，直接体现数据模型的完整全貌。当有新同学加入的时候，可以通过阅读聚合层代码获取最全、最准确的数据模型定义，不再需要从代码中四处搜集对象关联关系的蛛丝马迹，这体现了我们架构目标中“逻辑易传承”的要求。

```
// 降级后的报表查询，同时获取单元主体、单元日预算、应用集、出价及所属的计划信息
AdGroupDomainQuery adGroupDomainQuery = AdGroupDomainQuery.builder()
        .pin(loginPin)
        .userId(loginUserId)
        .campaignId(command.getCampaignId())
        .mediaResourceType(command.getMediaResourceType())
        .ids(command.getGroupIds())
        .campaignType(command.getCampaignType())
        .businessType(command.getBusinessType())
        .startTime(command.getEnclosedStartTime())
        .endTime(command.getEnclosedEndTime())
        .name(command.getName())
        .nameLike(command.getNameLike())
        .status(command.getShowStatus())
        .page(command.getPage())
        .pageSize(command.getPageSize())
        .build()
        .addSubEntityQuery(AdGroup.class, AdGroupDomainQuery.NON())
        .addSubEntityQuery(BudgetInfo.class, null)
        .addSubEntityQuery(TrafficStrategyInfo.class, null)
        .addSubEntityQuery(BiddingInfo.class, null)
        .addSubEntityQuery(DeliverySchedule.class, null)
        .addSubEntityQuery(Campaign.class, new CampaignDomainQuery().addSubEntityQuery(MapBuilder.of(
                Campaign.class, CampaignDomainQuery.NON(),
                MarketTargetInfo.class, CampaignDomainQuery.NON())));
Response<PageData<AdGroup>> adGroupPageData = adGroupRepository.findPage(adGroupDomainQuery);
if (adGroupPageData.isFailure()) {
    log.error("==> Fail to query group info with param: {}, info: {}", JSON.toJSONString(adGroupDomainQuery), adGroupPageData.getMsg());
    return Response.fail(adGroupPageData.getMsgOrElse("未知异常"));
}
if (adGroupPageData.getData() == null || CollectionsJ.isEmpty(adGroupPageData.getData().getData())) {
    log.warn("==> No page data found with query: {}", JSON.toJSONString(adGroupDomainQuery));
    return Response.success();
}
```

设置通用的查询条件，并且指定需要获取聚合根下的哪些子实体
并且可以为子实体的查询设置嵌套过滤逻辑

通过Repository中的公共接口一行代码获取全部数据，
数据对象之间的关联关系也已经组装好

RE降级后的补数逻辑一直是一件令人头痛的事情，聚合的引入可以极大地简化这一流程

本文主要探讨的是我们引入聚合的动机，关于数据模型的设计、Repository接口的实现和使用相关的实战内容只是点到为止，关于这部分的详细内容属于多体系架构中的数据模型管理体系，我们将在该体系的设计中进行深入的探讨。其实就我个人的实践经验而言，在实现架构升级所作出的众多尝试中，聚合的引入是给我带来幸福感最强的一项改进，但是我始终没能找到一种合适的表达方式将我之所感无所保留地传递给大家，所言之语总是苍白，或许聚合引入带来的收益只有让大家在实践中去亲身感受了。另外熟悉领域驱动设计的同学可能已经从上面的设计中嗅到了一丝DDD的味道，但是可能又会觉得没有那么DDD，关于这个问题限于当前陈述上下文的原因还不好直接给予解答，容笔者在这里卖个关子，在后面的系列文章中我们会详细阐明这种设计的细节和考量。

第四代：领域能力拆分与编排

通过引入聚合我们基本上解决了数据查询逻辑复用的问题，但是由于多平台、多维度和多场景化带来的业务复杂度的问题却依然存在。而解决这个问题的基本思路其实祖师爷已经给我们准备好了，那就是组合复用原则。

作为一个典型的2B的平台，我们的业务特点就是流程冗长复杂，一个业务流程通常由多个流程节点组成，比如单元新建流程，可以分为：基础信息设置、单元名称设置、投放周期设置、投放位置设置、定向设置、出价设置、关键词设置等多个节点组成。这些节点再叠加上不同产品线（展位、快车、触点）、站外不同媒体（头、腾、百、快、京X）、不同的投放平台（京准通、流量货币化、京易投）以及不同的站点（国内、泰国、印尼、出海）等多维度的业务场景，就使系统具备了

O(n1∗n2∗...∗nm)业务复杂度，其中nx为不同业务细分维度下的场景复杂度，而组合复用原则就是专门为解决这一问题而生的。

组合复用原则强调复杂问题的拆分，拆分出来的最小子问题可以互不干扰地进行独立的迭代。在此基础上，上层模块可以通过对最小子问题的组合编排实现一项完整的业务功能。由于最小子问题之间彼此正交，我们独立维护各个最小子问题的编码复杂度就可以降级为O(n1+n2+...+nm)。基于该思想，我们在新架构中引入了领域能力拆分与编排机制。

领域能力的识别与拆分

在新架构中我们会将一个完整的业务流程正交分解为多个“能力节点”。这里所说的“正交分解”是指拆分出来的各个子模块之间互不干扰，可以独立进行迭代。举个例子来说，在早期大家进行能力梳理的时候，有同学从单元新建流程中拆分出了“出价信息校验”和“出价设置”两个能力节点，这其实是不合理的。因为出价信息的校验和出价属性的设置并不正交，他们互相依赖，我们应该这两段逻辑合并到一起，抽象为一个“出价设置”节点。

能力节点主要定义了系统中各个原子模块的功能范围。一般来说，一个能力节点通常包含一个能力门面和0到多个能力实例。能力门面并不承接具体的业务逻辑，它的作用是对外暴露统一的调用入口及请求转发，具体的业务逻辑则由能力门面下的能力实例承接。比如出价设置节点下会按照出价类型划分为：手动出价、tCPA智能出价、MC智能出价、eCPC智能出价几个具体的领域能力实例，而在人群定向设置节点下则有京选店铺人群设置、乐高人群设置和自定义人群设置几个领域能力实例。

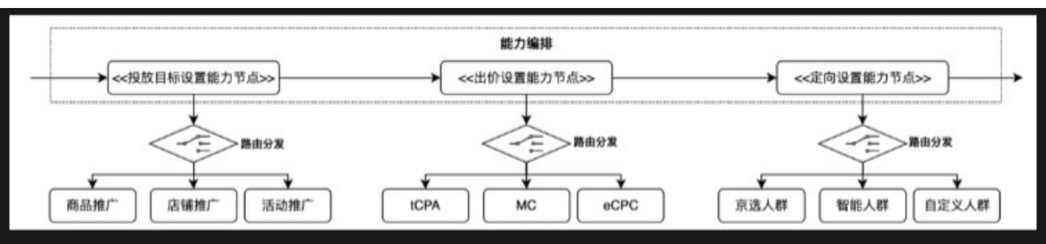
能力编排与请求路由

将整个系统划分为多个独立的能力节点之后，接下来就需要通过能力编排将这些能力节点串联到一起组装成一个完成的服务。如下图所示，所谓的能力编排就是将业务流程中所需要的原子模块对应的能力节点串联起来，定义好他们之间数据传递的方式和编排规则。需要注意的是，能力编排操作的是能力节点而不是能力实例，在处理服务请求时，每一个能力节点负责将请求路由到正确的领域能力实例中进行处理。之所以这样设计是因为我们的业务流程相对稳定，系统对外提供的服务流程中业务节点及节点间的执行顺序很少会发生变化，需求迭代往往是对某个能力节点进行横向的拓展，也就是对具体的领域能力实例进行增删或者修改。通过能力节点的抽象及路由机制的引入，我们将动态变化着的部分从相对稳定的业务流程中分离出去，从而保障核心流程的稳定性不被频繁变化着的需求所影响，这一点与我们当时做数据模型与业务模型分离的动机是一致的，本质上都是在隔离变化。

```
@Override
public void registerDomainAbilities(DomainAbilityOrchestrator<CampaignCreateCommand, CampaignCreationContext> abilityOrchestrator) {
    abilityOrchestrator.abilityNodeRegistrar()
            .registerSingle(baseInfoAssembleAbility)
                    .required()
                    .transferCtxBeforeStep(EnumExecutorStepType.INIT_CONTEXT, EnumExecutorStepType.BIZ_LOGIC_PROCESS)
            .done()
            .required(userMediaStatusCheckAbility)
            .required(campaignUpperLimitCheckAbility)
            .required(campaignScheduleConfigAbility)
            .required(campaignNameConfigAbility)
            .required(campaignTargetTypeAbility)
            .choose()
                    .when((srvCmd, srvCtx) -> srvCmd.getBudgetOptimizeSwitch() != null && EnumUtils.equals(EnumCampaignType.ZT_TIKTOK, srvCmd.getCampaignType()))
                    .required(campaignBudgetOptAbility)
                    .endWhen()
            .endChoose()
            .registerSingle(campaignBudgetConfigAbility)
                    .transferCtxBeforeStep(EnumExecutorStepType.INIT_CONTEXT)
                    .ctxTransfer((srvCtx, abtCmd, abtCtx) -> {
                        abtCtx.setBudgetSettable(srvCtx.getCampaign());
                        abtCtx.setBudgetOptimizeSwitch(srvCtx.getCommand().getBudgetOptimizeSwitch());
                    })
            .done()
            .choose()
                    .when(((srvCmd, srvCtx) -> EnumUtils.equals(EnumCampaignType.ZT_TIKTOK, srvCmd.getCampaignType())))
                    .required(campaignAddSyncAbilityExecutor)
                    .endWhen()
            .endChoose()
            .register();
}
```


一个能力编排示例

除了能力编排框架之外，能力实例的路由机制也是实现复杂度降维的关键。如下图所示，路由机制通过将能力门面及门面下用于承接不同场景下具体业务规则的能力实例打包到一起，同时也将原子业务模块内的场景复杂度封装屏蔽在了模块内部，使上层的业务流程定义只需要关注一次完整的请求需要使用哪些原子业务模块（也就是能力节点），而无需关注这个节点下具体的能力实例，当请求到来时，处理流程流经相应的能力节点时，将通过当前请求上下文中的参数自动识别业务身份并将请求路由到相应的能力实例上进行处理。



能力编排操作的是能力节点而不是领域能力实例，这样可以让能力实例更灵活的进行横向拓展（点击放大查看）

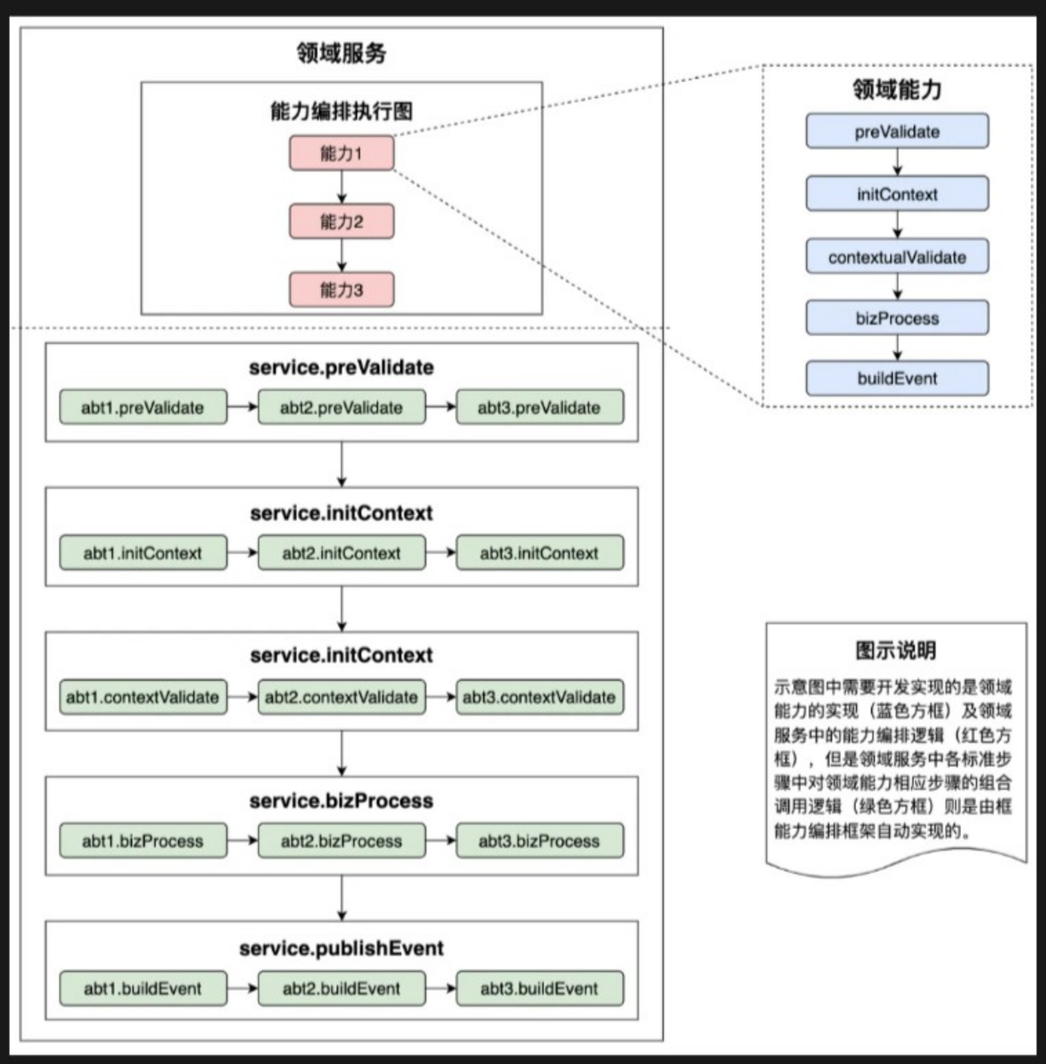
上文提到了能力编排和路由机制都已经在新工程中提供了框架化的实现，本文主要是为了分享我们架构设计的动机，所以不会介绍这些功能的实现原理和使用方法。

标准的业务执行模版

在第二、三代架构中，系统处理请求时会先执行全部参数的校验，校验通过后再将单元新建处理所需的全部数据对象查询出来。在这个过程中可以充分利用批量查询接口提升系统性能，同时也会对查询出来的数据对象进行校验，如果存在不合法的数据则终止处理流程，如果数据对象查询一切正常，则执行后续的数据组装和处理逻辑，最后批量执行数据的持久化。尽管会存在上文分析的一些问题，但是这种模式所带来的收益依然具备十分重要的意义。

然而在新架构中我们将原先连贯的业务逻辑打散，按照逻辑的内聚性将他们重组到一个能力实例中，然后在领域服务中通过能力编排将这些能力实例组装成一个完整的业务流程。这虽然贯彻了组合复用的原则，但是如果我们只是简单地通过顺序执行多个能力实例来组装领域服务，那么由于每个能力内部又依次执行与一小撮业务属性相关的参数校验、依赖数据查询、逻辑处理乃至数据持久化操作，从代码逻辑的执行流程上看我们又回退到了“数据访问与逻辑处理互相交织”的第一代架构上。除此之外，虽然服务之间逻辑上互相独立，但是他们可能会依赖相同的数据对象，比如人群包的绑定与预算调整两个能力都会依赖AdGroup对象，如果框架只是简单地串联执行这两个能力，那么必然会造成数据的重复查询。

为了解决上述问题，我们引入了标准的业务流程Executor模板，它把业务业务流程抽象为：参数校验、上下文初始化、上下文校验、业务逻辑处理、数据持久化、发布事件几个标准步骤，不论是领域能力的封装还是领域服务的实现都必须继承该模板。标准业务执行模板的引入一方面能够规范开发者的设计和实现，另一方面也将代码逻辑的串联执行权从开发者手中转移到了能力编排框架中，让框架能够实现逻辑的自动重组和执行，而开发者专注于业务逻辑并进行填空式开发。而框架在获取到了代码逻辑的串联执行权之后就可以在领域服务的每个标准步骤中按照能力编排执行图组装调用的各个能力实例中相应标准步骤，从而将打散到不同能力实例中的业务逻辑次按照标准步骤的类别还原回连贯完整的业务逻辑，如下图所示：



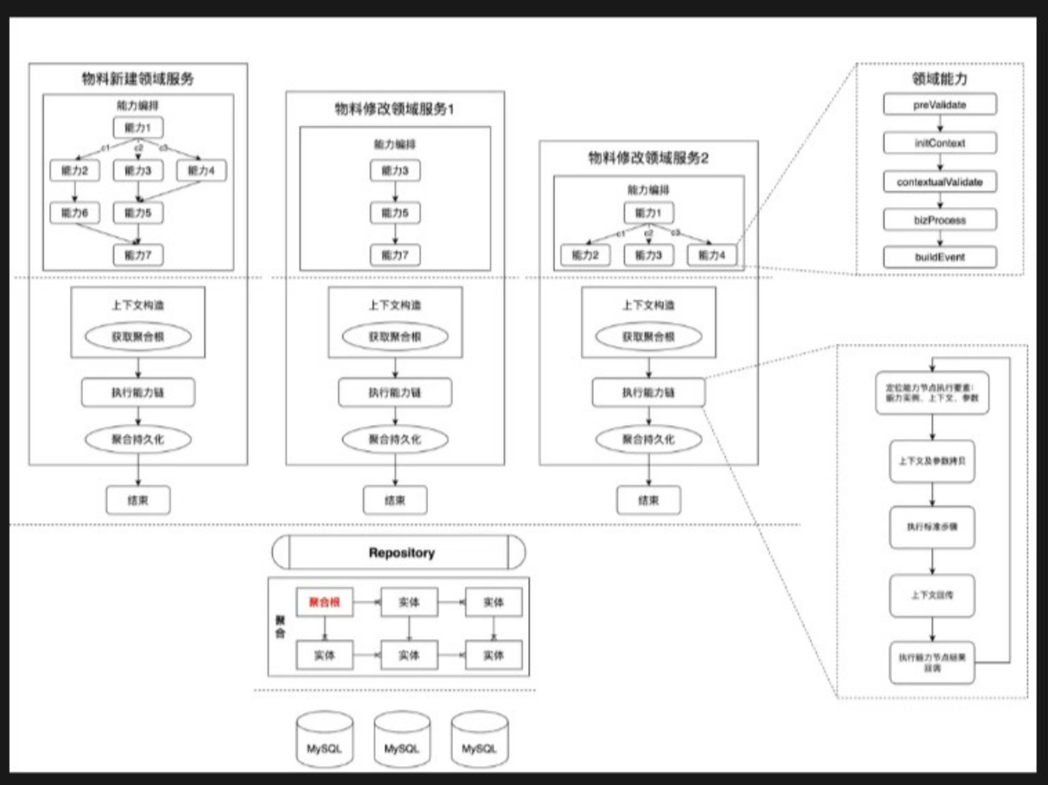
标准业务流程模版的引入让框架进行业务流程还原成为可能

除了实现业务逻辑按标准步骤自动还原之外，由于标准流程模板对每一个标准步骤方法的执行参数、依赖的上下文及返回值对象都进行了通用化的抽象，能力编排框架也得以在各个能力标准步骤调用之间插入参数及上下文的映射和传递逻辑，从而在不同能力之间以及能力与领域服务之间实现数据分发和共享。需要说明的是尽管这些流程都可以采用默认的自动处理规则，开发者也可以通过能力编排框架提供的DSL对默认的串联执行、数据传递、异常处理等规则进行修改。

在我们新架构中，我们通过领域能力拆分将复杂的问题域正交分解为多个互相独立的最小问题域，让设计者可以分而治之，逐个击破，降低了问题的复杂度和设计成本，同时单个能力节点下不同业务场景下的业务逻辑被分离到了不同的领域能力实例中，避免出现不同业务场景互相交织，便于快速梳理业务逻辑，定位改动点，这些都体现了“设计易拓展”的设计目标。

由于拆分出来的各个能力节点彼此正交，内部逻辑十分内聚，因此可以在各自的维度上进行迭代，比如同样是在单元维度下的出价设置和人群设置能力就分别在出价类型和人群类型这两个场景维度上各自进行路由，避免了不同场景互相交织带来的圈复杂度上升问题，也能够更加灵活在不同的业务场景中实现能力复用。同时由于我们的业务本质上就是对物料的创编，物料新建流程中的能力往往可以直接在物料修改流程中复用。还有一个特殊的场景就是批量物料操作类型的请求，借助能力编排框架提供的循环编排和数据共享机制，我们可以在领域服务的开始先批量完成所需数据的查询，然后通过循环编排机制循环复用单个请求处理能力中的纯内存调用的数据校验及数据处理逻辑，最后在批量操作领域服务中批量完成聚合根对象集合的写入，在实现逻辑复用的同时又能保证数据准确性及性能，以上特性都体现了“代码易复用”的设计目标。

领域能力的编排逻辑提供了一个业务流程的全景视图，当有新同学加入时，可以迅速通过阅读能力编排逻辑快速建立起对业务的宏观认知，再结合在第三代架构中引入的聚合机制，可以让新同学快速熟悉数据模型与业务流程。同时通过路由机制系统中全部的业务规则打包拆分成数量有限且边界清晰的能力节点，当需要快速梳理需求点对应业务规则时，可以由粗及细，先确定需求点归属的能力节点，然后根据场景定位到具体的能力实例，进而可以从代码中获取业务规则，这些特性都体现了“逻辑易传承”的设计目标。



基于能力拆分与编排的代码架构，最显著的收益就是同一个能力可以在不同的领域服务中直接复用

04 总结

以上便是我们为实现新架构所进行的种种尝试，这些设计是否正确我们也正在通过需求实战来进行验证，把他们发出来不是要说服大家认同，而是想通过对架构演进历程的推演帮助大家更好的理解我们新架构中各项功能的设计动机，从而更快的上手进行开发；另一方面也希望能够激发大家的思考和讨论，哪怕是对上述方案的质疑和批判，一个好的架构一定是在一次次批评声中改进出来的，我至今还在怀念当初摸索新架构时那些与永亮（我的良师益友，部门内探索中台化及领域驱动设计思想的第一人）争论到凌晨2、3点的日子。

- END -

关注京东零售技术微信公众号，长按下方二维码加入技术交流群！与京东零售技术专家们一起切磋！