领域驱动架构（DDD）建模中的模型到底是什么？

<https://www.zhihu.com/question/25089273/answer/233316164>

**聚合根**

聚合根（Aggregate Root）是DDD中的一个概念，是一种更大范围的封装，会把一组有相同生命周期、在业务上不可分割的实体和值对象放在一起，只有根实体可以对外暴露引用，这也是一种内聚性的表现。

确定聚合边界要满足固定规则（Invariant），是指在数据变化时必须保持的一致性规则，具体规则如下。

1根实体具有全局标识，最终负责检查规定规则。

2聚合内的实体具有本地标识，这些标识在Aggregate内部才是唯一的。

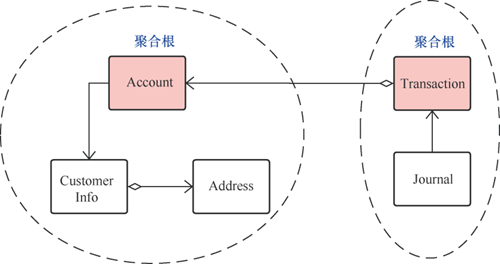
3外部对象不能引用除根Entity之外的任何内部对象。

4只有Aggregate的根Entity才能直接通过数据库查询获取，其他对象必须通过遍历关联来发现。

5Aggegate内部的对象可以保持对其他Aggregate根的引用。

6Aggregate边界内的任何对象在修改时，整个Aggregate的所有固定规则都必须满足。

仍以银行转账的例子来说明，如图5所示，账号（Account）是客户信息（CustomerInfo）Entity和值对象（Address）的聚合根，交易（Tansaction）是流水（Journal）的聚合根，流水是因为交易才产生的，具有相同的生命周期。

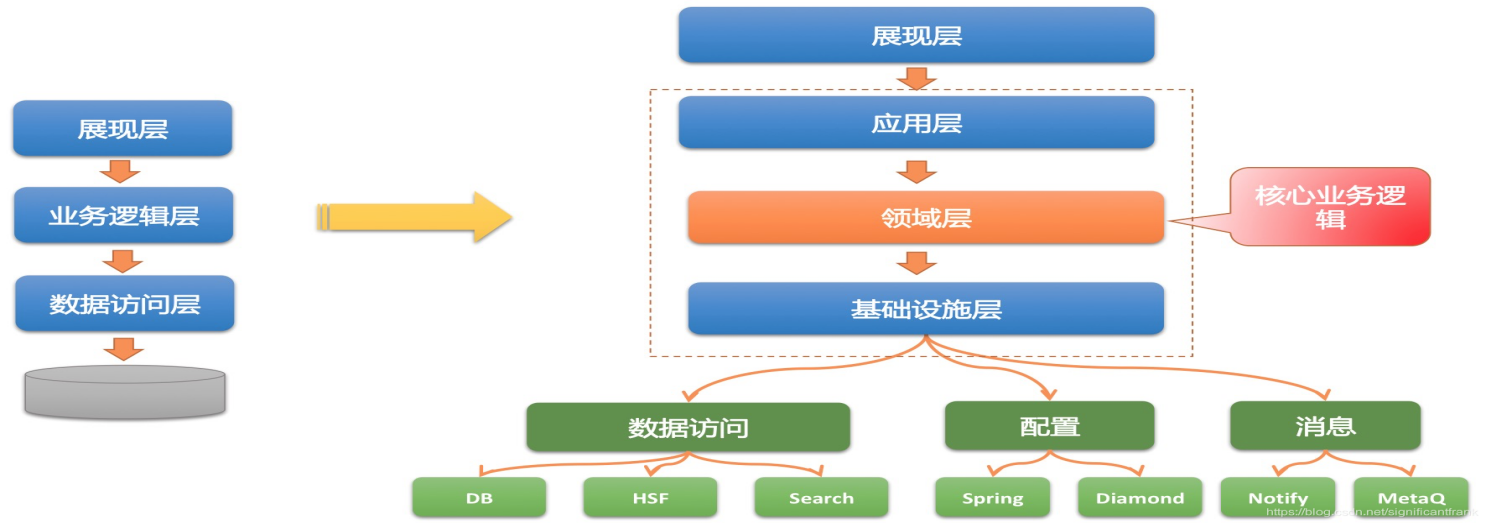


**COLA应用架构**

COLA架构是我团队自主研发的应用架构，目前已经开源（ https://github.com/alibaba/COLA ）。在COLA的设计中，我们充分汲取了经典架构的优秀思想。除此之外，我们补充了规范设计和扩展设计，并且使用Archetype的方式，将架构固化下来，以便可以快速的在开发中使用。

**分层设计**

COLA的分层是一种改良了的三层架构。主要是将传统的业务逻辑层拆分成应用层、领域层和基础实施层。如下图所示，左边是传统的分层架构，右边是COLA的分层架构。



其每一层的作用范围和含义如下：

1）展现层（Presentation Layer）：负责以Rest的格式接受Web请求，然后将请求路由给Application层执行，并返回视图模型（View Model），其载体通常是DTO（Data Transfer Object）；

2）应用层（Application Layer）：主要负责获取输入，组装上下文，做输入校验，调用领域层做业务处理，如果需要的话，发送消息通知。当然，层次是开放的，若有需要，应用层也可以直接访问基础实施层；

3）领域层（Domain Layer）：主要是封装了核心业务逻辑，并通过领域服务（Domain Service）和领域对象（Entities）的函数对外部提供业务逻辑的计算和处理；

4）基础实施层（Infrastructure Layer）主要包含Tunnel（数据通道）、Config和Common。这里我们使用Tunnel这个概念来对所有的数据来源进行抽象，这些数据来源可以是数据库（MySQL，NoSql）、搜索引擎、文件系统、也可以是SOA服务等；Config负责应用的配置；Common是通用的工具类。

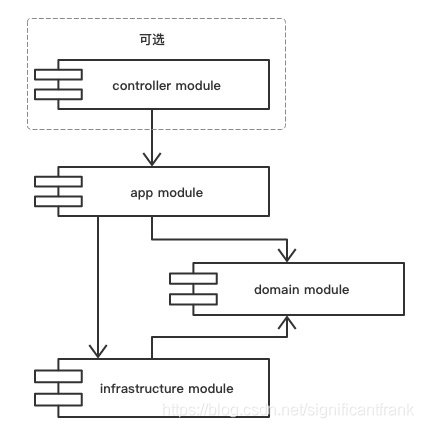
**扩展设计**

对于只有一个业务的简单场景，对扩展性的要求并不突出，这也是为什么扩展设计常被忽略的原因，因为我们大部分的系统都是从单一业务开始的。但是随着业务场景越来越复杂，代码里面开始出现大量的if-else逻辑。此时除了常规的策略模式以外，我们可以考虑在架构层面提供统一的扩展解决方案。

在扩展设计中，我们提炼出两个重要的概念，一个是业务身份，另一个是扩展点。

业务身份是指业务在系统唯一标识一个业务或者一个场景的标志。在具体实现中，我们使用BizCode来表示业务身份，其中BizCode采用类似Java包名命名空间的方式。例如，我们可以用“ali.tmall”表示阿里天猫业务，用“ali.tmall.car” 表示阿里天猫的汽车业务，而用"ali.tmall.car.aftermarket"代表这是阿里天猫的汽车业务的后市场场景。

每个业务或者场景都可以实现一个或多个扩展点（ExtensionPoint），也就是说一个业务身份加上一个扩展点，可以唯一地确定一个扩展实现（Extension）。而这个业务身份和扩展点的组合，我们将其称之为扩展坐标（ExtensionCoordinate），如下图所示。



这样，通过业务身份+扩展点，我们就可以从框架层面实现对不同租户，不同业务，不同场景的扩展定制了。整个阿里业务中台正是基于这个思想，实现的多业务支撑的。

**规范设计**

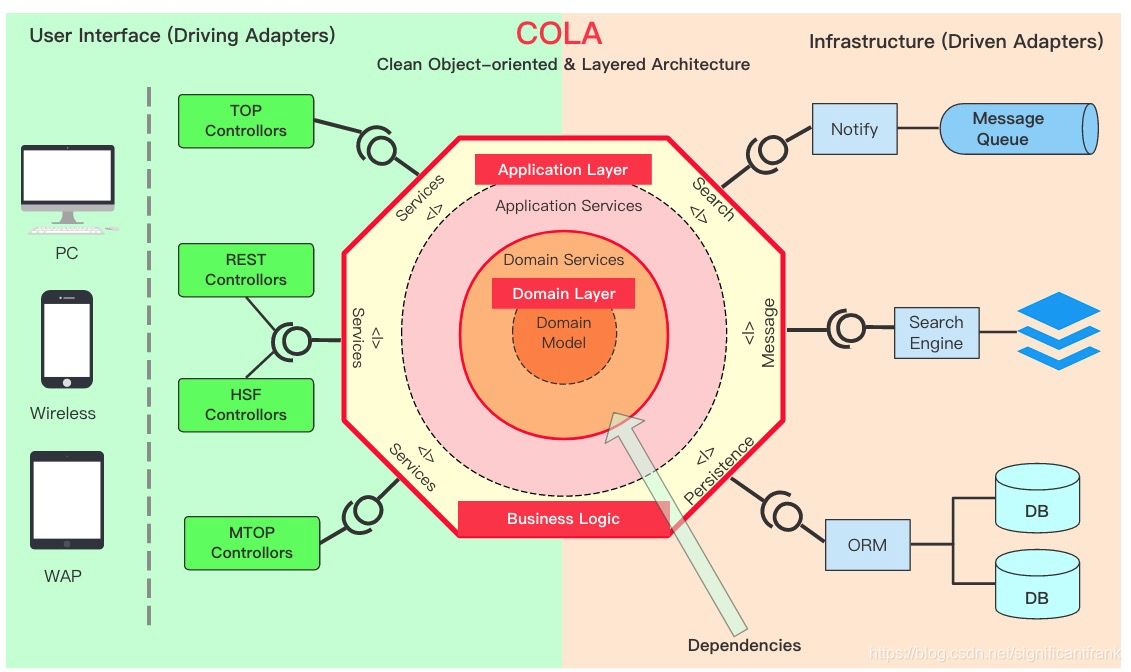
任何事物都是规则性和随机性的组合。规范的意义就在于我们可以将规则性的东西固化下来，尽量减少随心所欲带来的复杂度，一致性可以降低系统复杂度。从命名到架构皆是如此，而架构本身就是一种规范和约束，破坏这个约束，也就破坏了架构。

COLA制定了一些列的规范：包括组件（Module）结构、包（Package）结构、命名等。

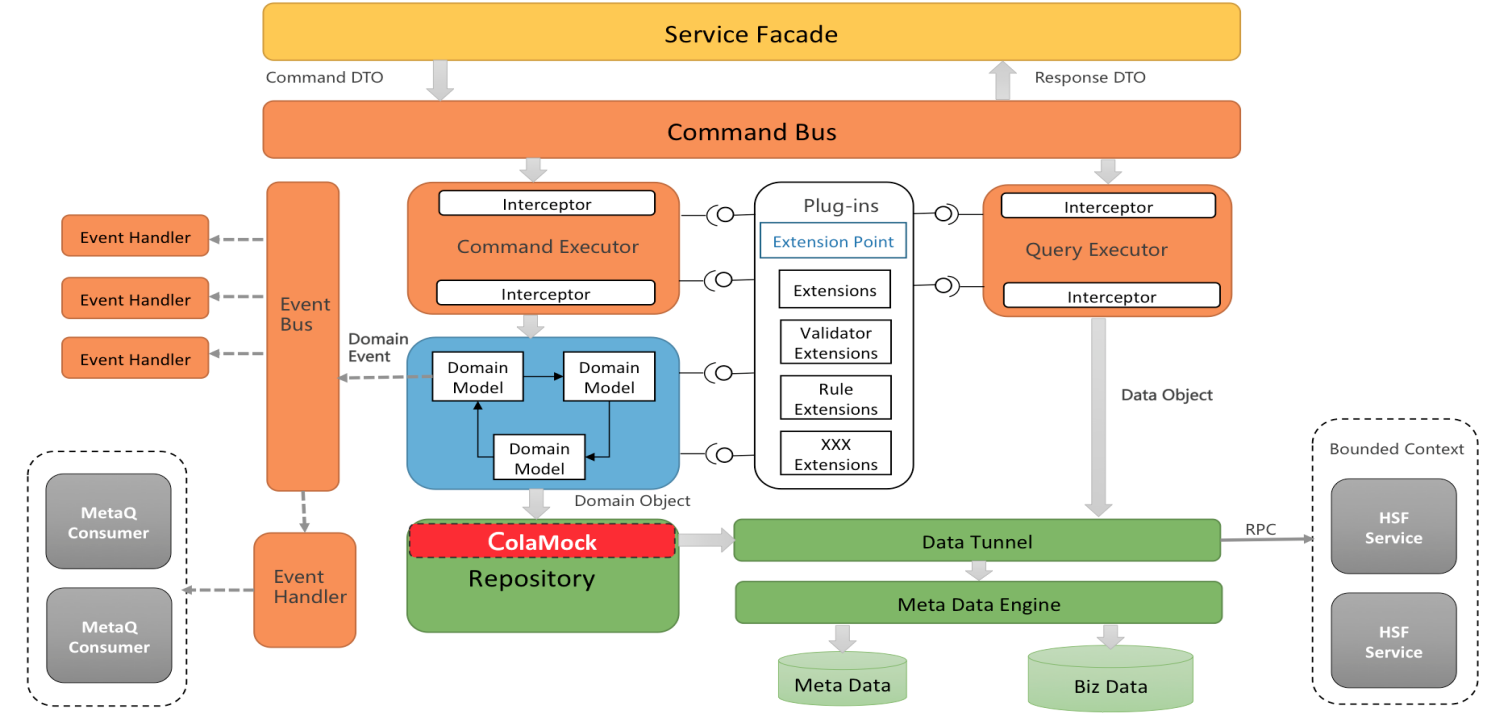
比如对于组件，我们要求使用COLA的应用都应该遵循如下图所示的组件划分：

**COLA架构总览**

在架构思想上，COLA主张像六边形架构那样，使用端口-适配器去解耦技术细节；主张像洋葱圈架构那样，以领域为核心，并通过依赖倒置反转领域层的依赖方向。最终形成如下图所示的组件关系。

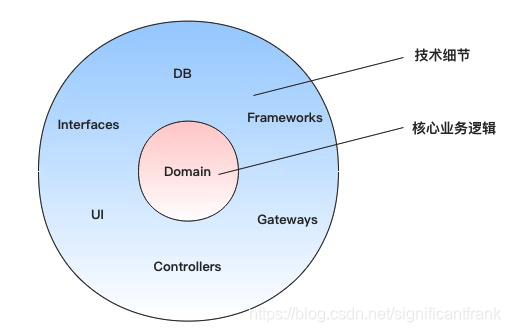


换一个视角，从COLA应用处理响应一个请求的过程来看。COLA使用了CQRS来分离命令和查询的职责，使用扩展点和元数据来提升应用的扩展性。整个处理流程如下图所示：



应用架构的核心

纵观上面介绍的所有应用架构，我们可以发现一个共同点，就是“核心业务逻辑和技术细节分离”。



是的，六边形架构、洋葱圈架构、以及COLA架构的核心职责就是要做核心业务逻辑和技术细节的分离和解耦。

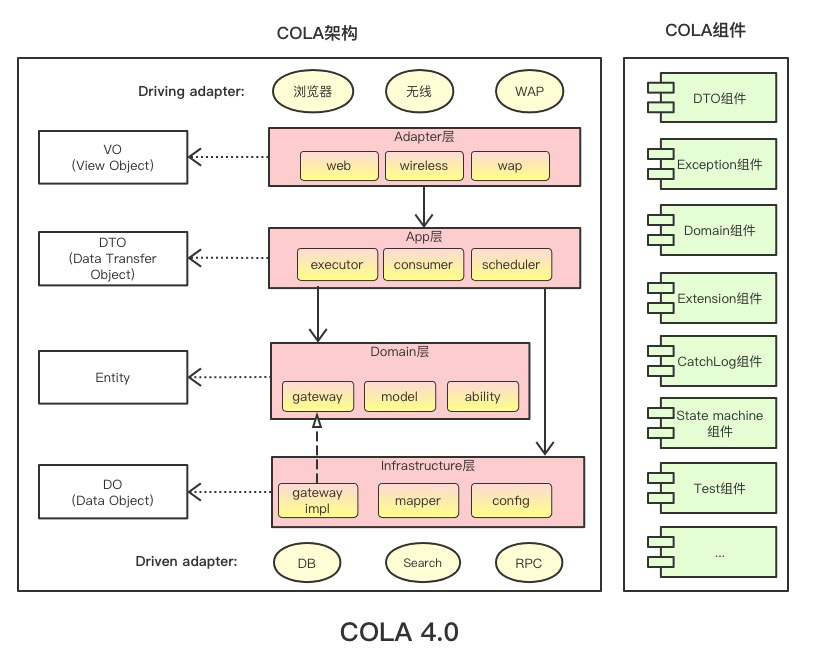
试想一下，业务逻辑和技术细节糅杂在一起的情况，所有的代码都写在ServiceImpl里面，前几行代码是做validation的事，接下来几行是做convert的事，然后是几行业务处理逻辑的代码，穿插着，我们需要通过RPC或者DAO获取更多的数据，拿到数据后，又是几行convert的代码，在接上一段业务逻辑代码，然后还要落库，发消息…等等。

再简单的业务，按照上面这种写代码的方式，都会变得复杂，难维护。

因此，我认为应用架构的核心使命就是要分离业务逻辑和技术细节。让核心业务逻辑可以反映领域模型和领域应用，可以复用，可以很容易被看懂。让技术细节在辅助实现业务功能的同时，可以被替换。

**COLA 4.0**

经过多次迭代，我们定义出了相对稳定、可靠的应用架构：COLA 4.0



**COLA Archetype**

好的应用架构，都遵循一些共同模式，不管是六边形架构、洋葱圈架构、整洁架构、还是COLA架构，都提倡以业务为核心，解耦外部依赖，分离业务复杂度和技术复杂度等。

COLA架构区别于这些架构的地方，在于除了思想之外，我们还提供了可落地的工具和实践指导。

为了能够快速创建满足COLA架构的应用，我们提供了两个archetype，位于cola-archetypes目录下：

cola-archetype-service：用来创建纯后端服务的archetype。

cola-archetype-web：用来创建adapter和后端服务一体的web应用archetype。

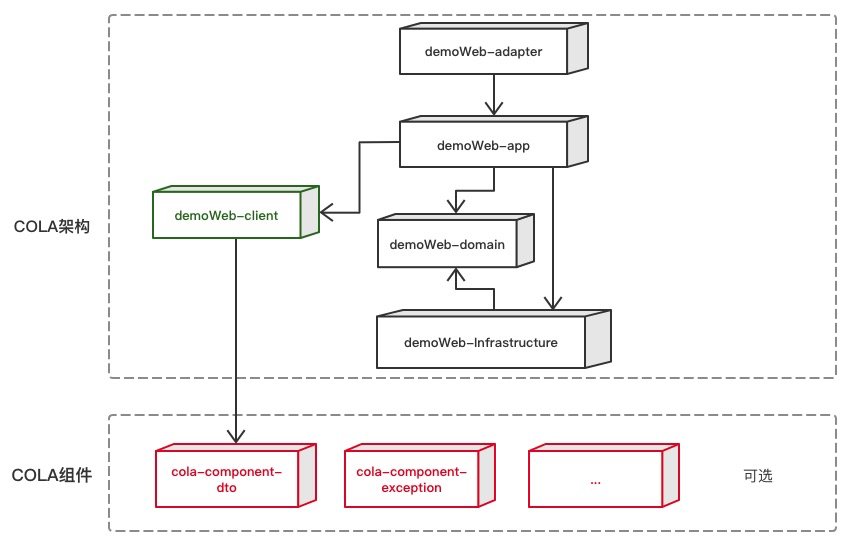
**COLA Components**

此外，我们还提供了一些非常有用的通用组件，这些组件可以帮助我们提升研发效率。

这些功能组件被收拢在[cola-components目录](https://gitee.com/mirrors/cola-framework/blob/master/cola-components)下面。到目前为止，我们已经沉淀了以下组件：

| 组件名称 | 功能 | 版本 | 依赖 |
| --- | --- | --- | --- |
| cola-component-dto | 定义了DTO格式，包括分页 | 1.0.0 | 无 |
| cola-component-exception | 定义了异常格式， 主要有BizException和SysException | 1.0.0 | 无 |
| cola-component-statemachine | 状态机组件 | 1.0.0 | 无 |
| cola-component-domain-starter | Spring托管的领域实体组件 | 1.0.0 | 无 |
| cola-component-catchlog-starter | 异常处理和日志组件 | 1.0.0 | exception 、dto组件 |
| cola-component-extension-starter | 扩展点组件 | 1.0.0 | 无 |
| cola-component-test-container | 测试容器组件 | 1.0.0 | 无 |

命令执行成功的话，会看到如下的应用代码结构：



注：也可以使用阿里云的应用生成器：<https://start.aliyun.com/bootstrap.html> 生成cola应用。