DDD 战术设计(Kratos)

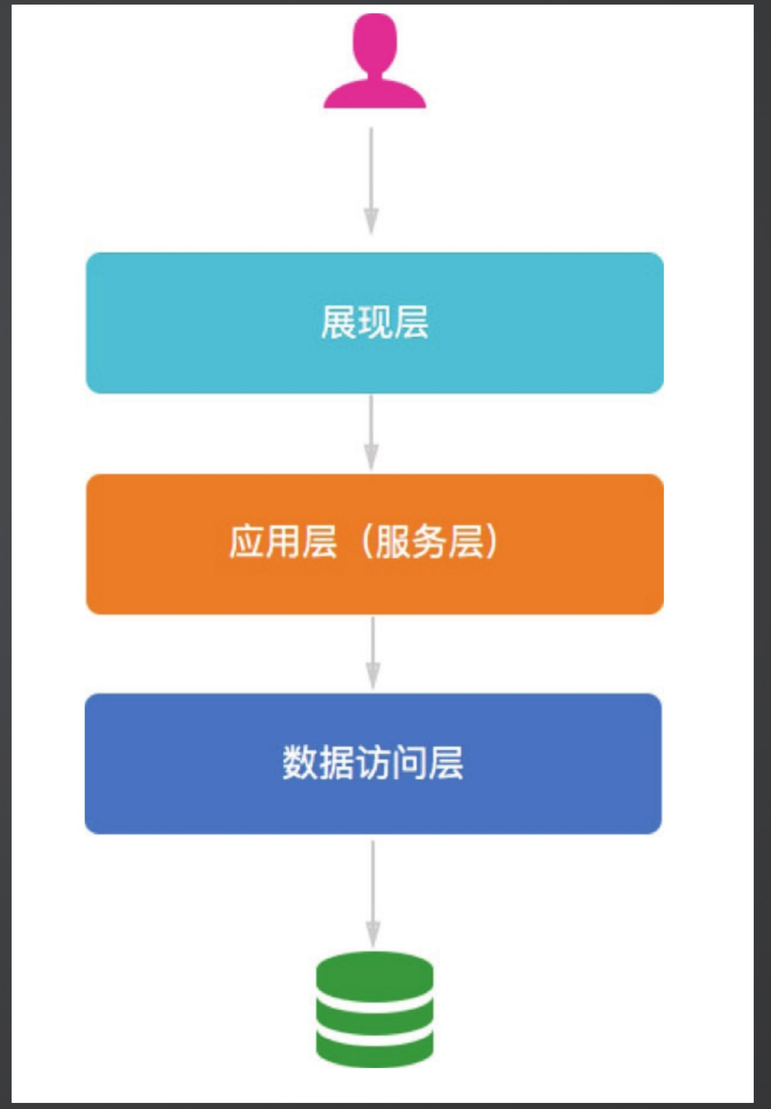
毛剑/bilibili 基础架构部 负责人

目录

贫血模型 vs 领域模型

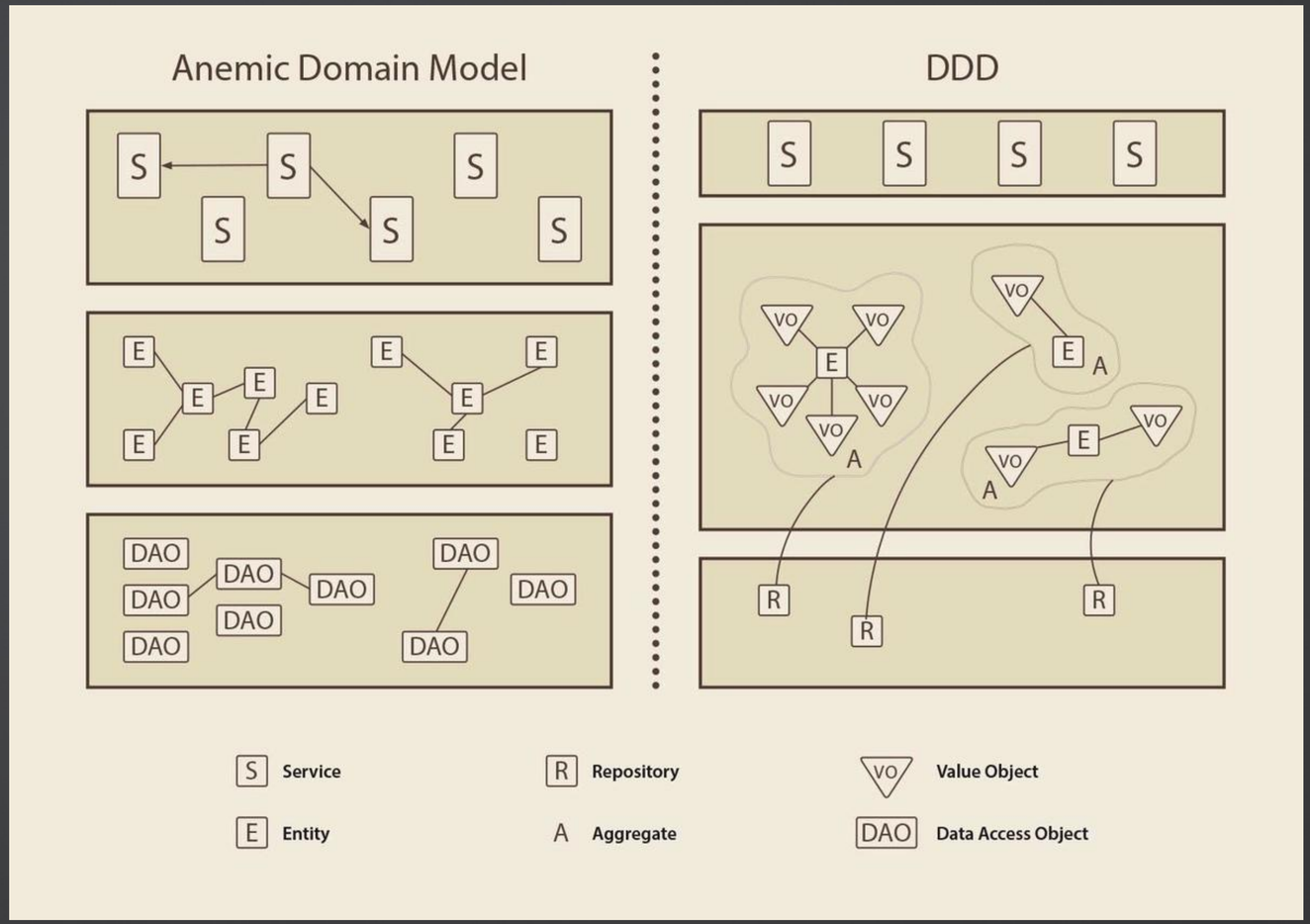
为了管理庞大的资源种类和繁复的引用关系，人们自然而然的将做同样事情的代码放在了统一的地方。将不同职责的事物分类，将复杂的、庞大的问题分解、降级成可以解决的问题，然后分而治之。

比如在实践中，展现部分的代码只负责将数据渲染出来，应用部分的代码只负责序列化 / 反序列化、组织并协调对业务服务的调用，数据访问层则负责屏蔽底层关系型数据库的差异，为上层提供数据。这就是层级架构的由来：上层的代码直接依赖于临近的下层，一般不对间接的下层产生依赖，层次之间通过精心设计的 API 来通信（依赖通常也是单向的）。



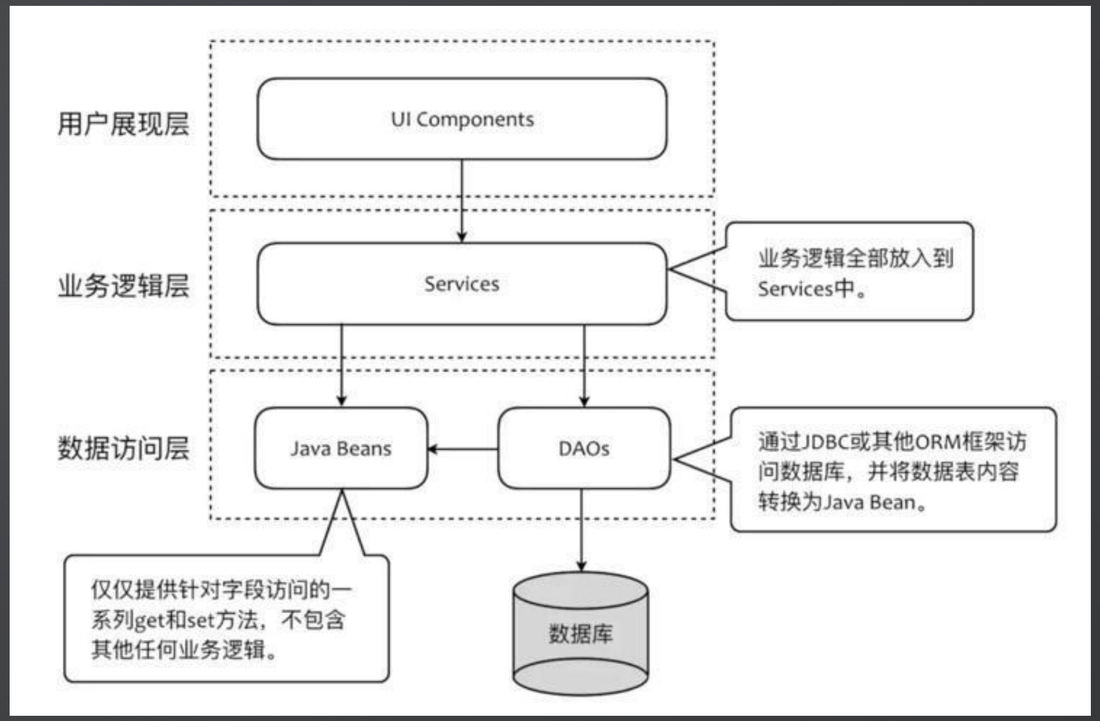
贫血模型 vs 领域模型

贫血模型是指，其领域对象贫血的模型。在贫血模型中，领域对象仅用作数据载体，没有行为和业务逻辑，业务逻辑通常放在服务、utils 和 helper 程序等中。被持久化的类，我们称之为实体。它们在上图的图表中以字母 E 表示。这些实体实际上是数据库表的 Object-Oriented Representation。我们没有在它们内部实现任何业务逻辑。它们唯一的作用是被一些 ORM 映射到它们的数据库等价物。



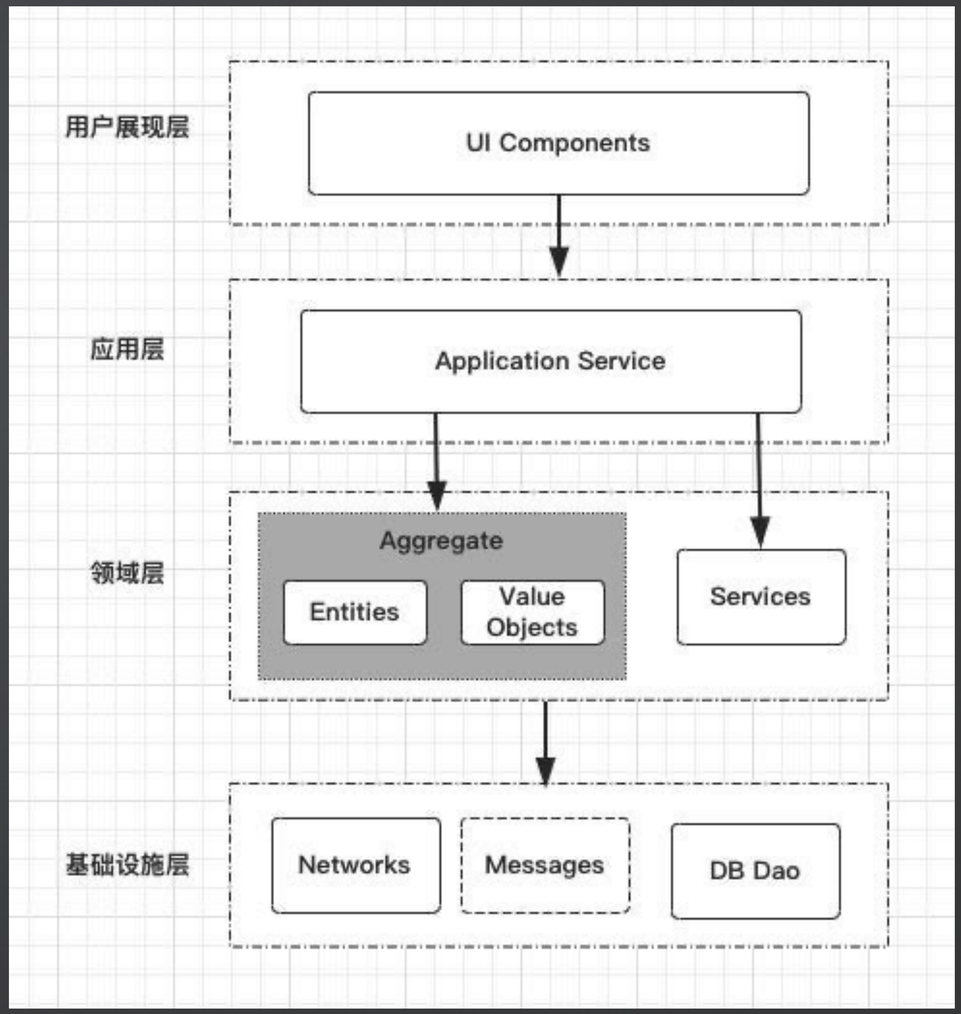
贫血模型 vs 领域模型

当我们的映射实体准备就绪时，下一步是创建用于读写它们的类。就是 DAO(数据访问对象)。通常，我们的每个实体都代表不同的业务用例，所以 DAO 类的数量与实体的数量相匹配。DAO 类只不过是用于检索和持久化实体的工具。在 DAO 之上的最后一层是我们实现的精髓 Service。典型的服务总是执行以下操作: 使用 DAO 加载实体，根据需求修改它们的状态并持久保存它们。Martin Fowler 将这种体系结构描述为一系列事务脚本。功能越复杂，加载和持久化之间的操作数量就越多。



贫血模型 vs 领域模型

上面提到，这种传统的三成模式，会导致贫血模型。要避免贫血模型，就需要合理地将操作数据的行为分配给领域模型对象（Domain Model），即战术设计中的 Entity 与 Value Object，而不是放到三层模型中的 Service 中。

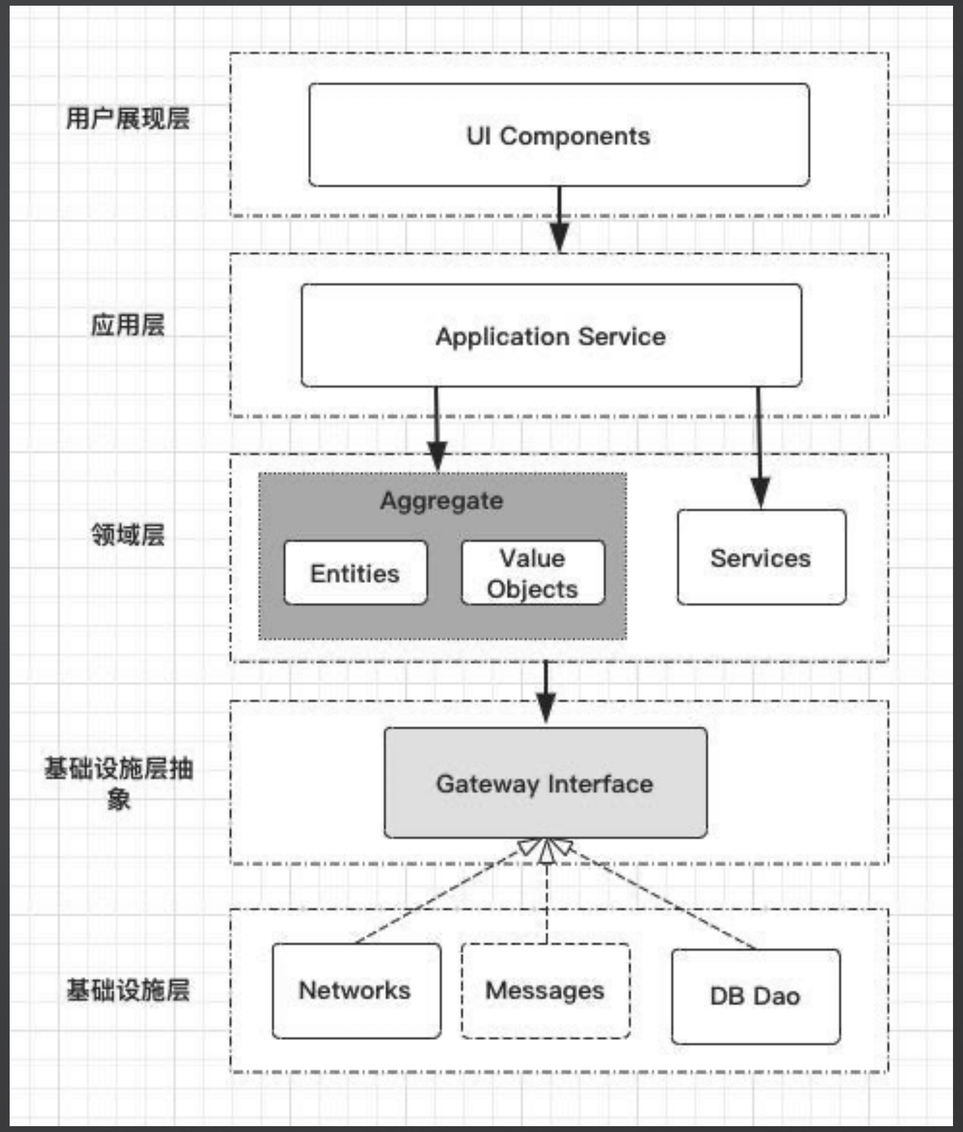


目录

领域驱动设计

将业务的行为合理的分配给了领域模型对象（Domain Model），这样可以避免贫血，同时又不会造成业务逻辑层太臃肿。

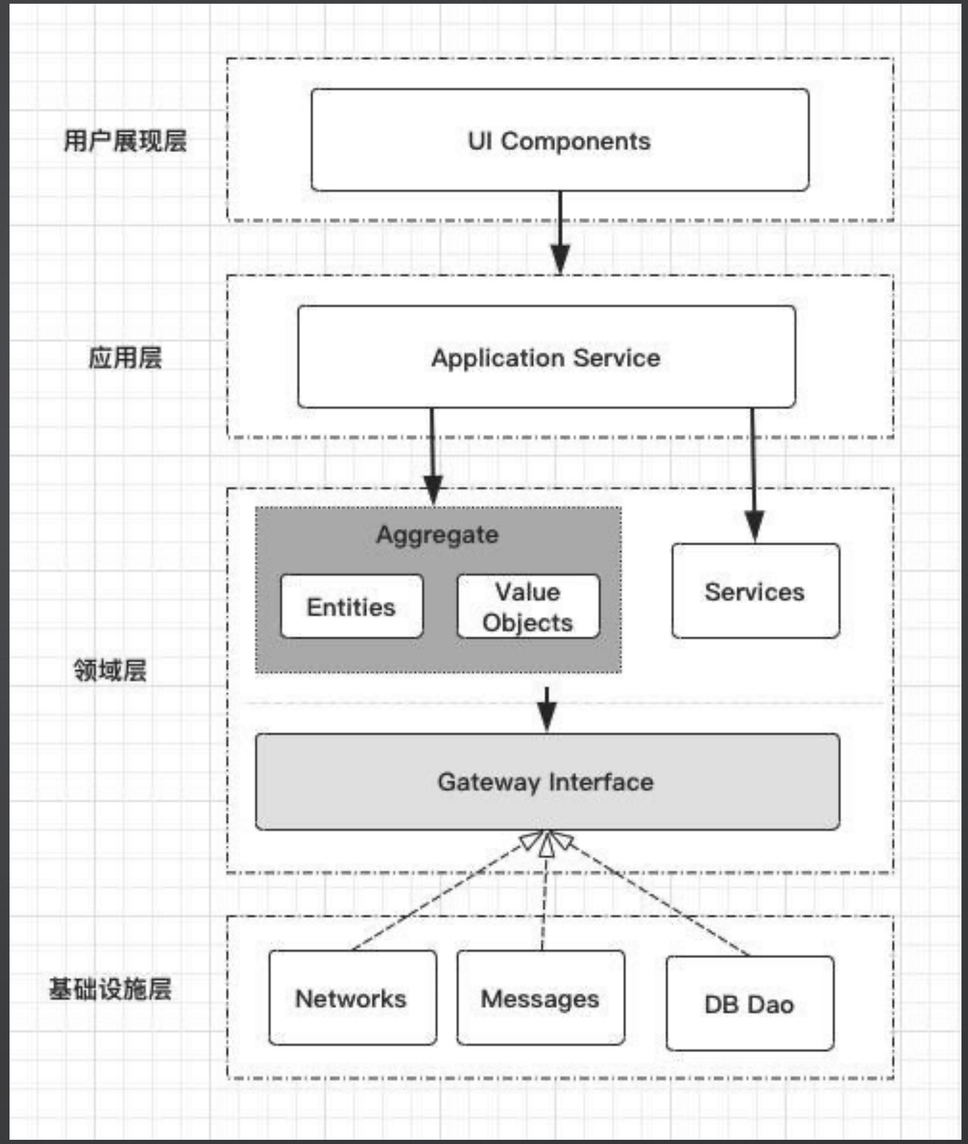
发完一个系统，不可能一成不变，都是会不断的更新迭代的，因此需求会不断的更新和变化。仔细观察，我们会发现变化总是有迹可循的。其一，用户体验、操作习惯的变化，往往会导致系统界面展示的多变；其二，部署平台，组件切换的变化，往往会导致系统底层存储的多变。但总体来说，系统的核心领域逻辑基本上不会大变。



领域驱动设计

访问这些领域对象属于业务要素，而如何访问这些领域对象（如通过外部资源），则属于具体实现的技术要素。从编码角度看，领域对象实例的容身之处不过就是一种数据结构而已，区别仅在于存储的位置。领域驱动设计将管理这些对象的数据结构抽象为资源 Repository。

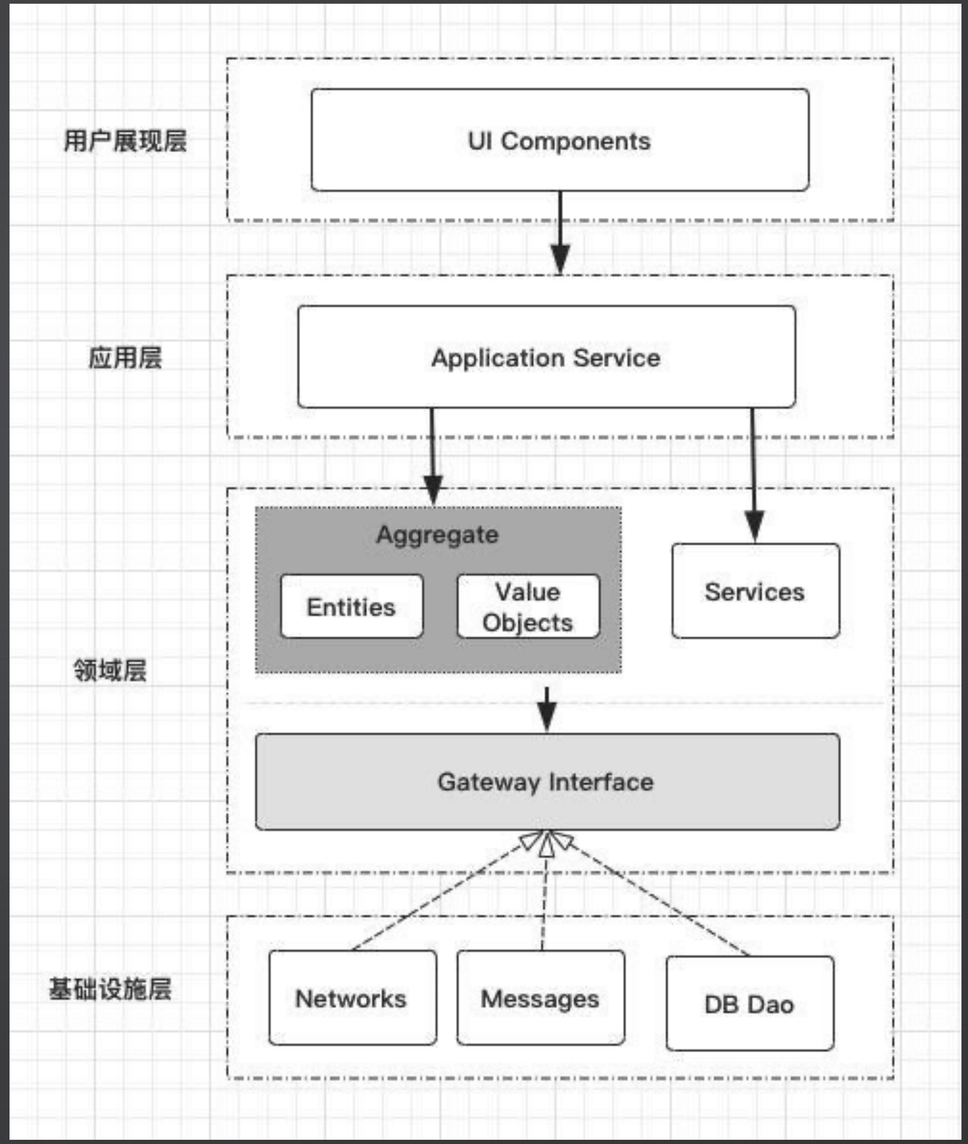
通过这个抽象的资源库访问领域对象，自然就应该看作是一种领域行为。倘若资源库的实现为数据库，并通过数据库持久化的机制来实现领域对象的生命周期管理，则这个持久化行为就是技术因素。



领域驱动设计

访问这些领域对象属于业务要素，而如何访问这些领域对象（如通过外部资源），则属于具体实现的技术要素。从编码角度看，领域对象实例的容身之处不过就是一种数据结构而已，区别仅在于存储的位置。领域驱动设计将管理这些对象的数据结构抽象为资源 Repository。

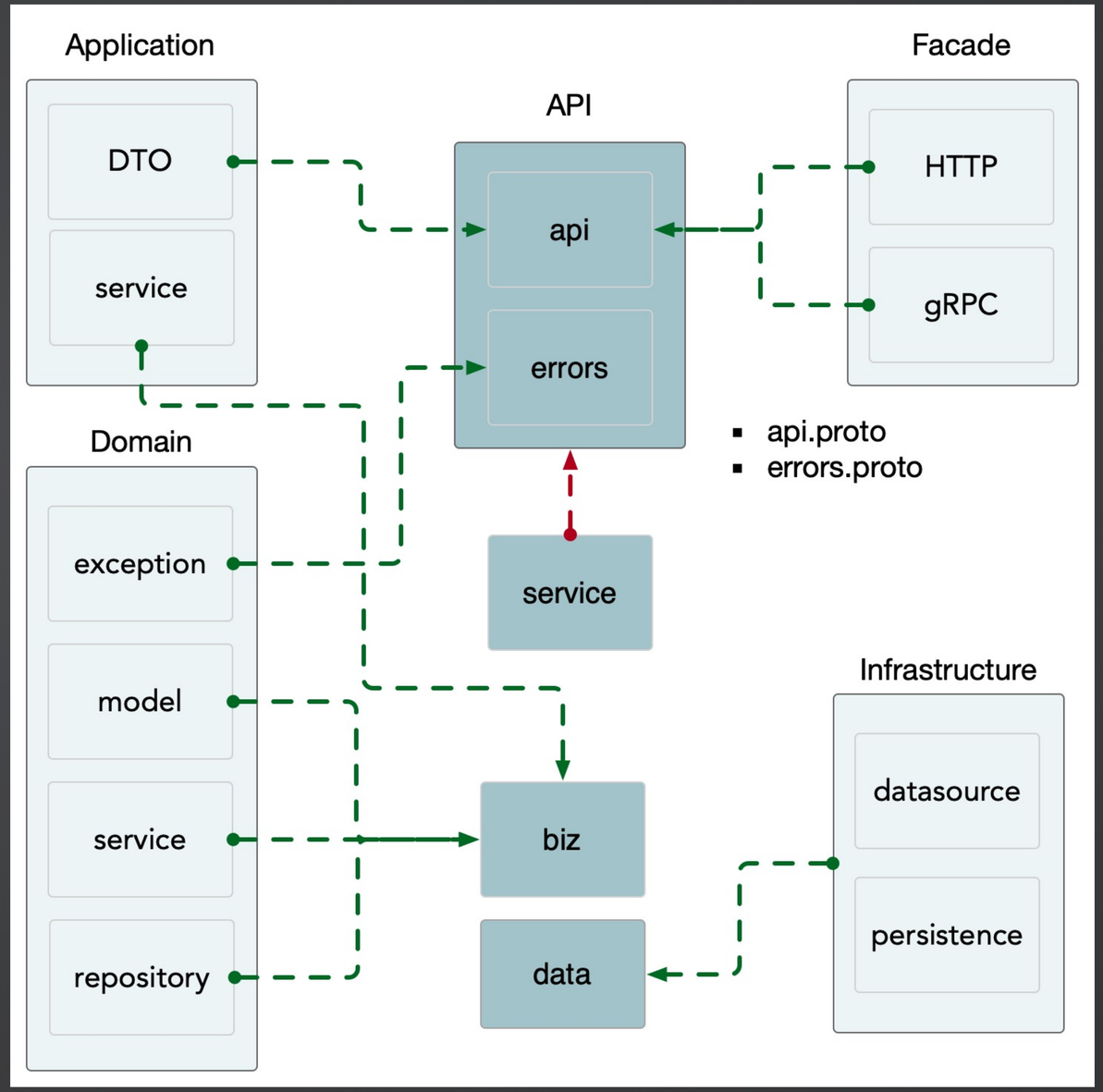
通过这个抽象的资源库访问领域对象，自然就应该看作是一种领域行为。倘若资源库的实现为数据库，并通过数据库持久化的机制来实现领域对象的生命周期管理，则这个持久化行为就是技术因素。



领域驱动设计

接口层：

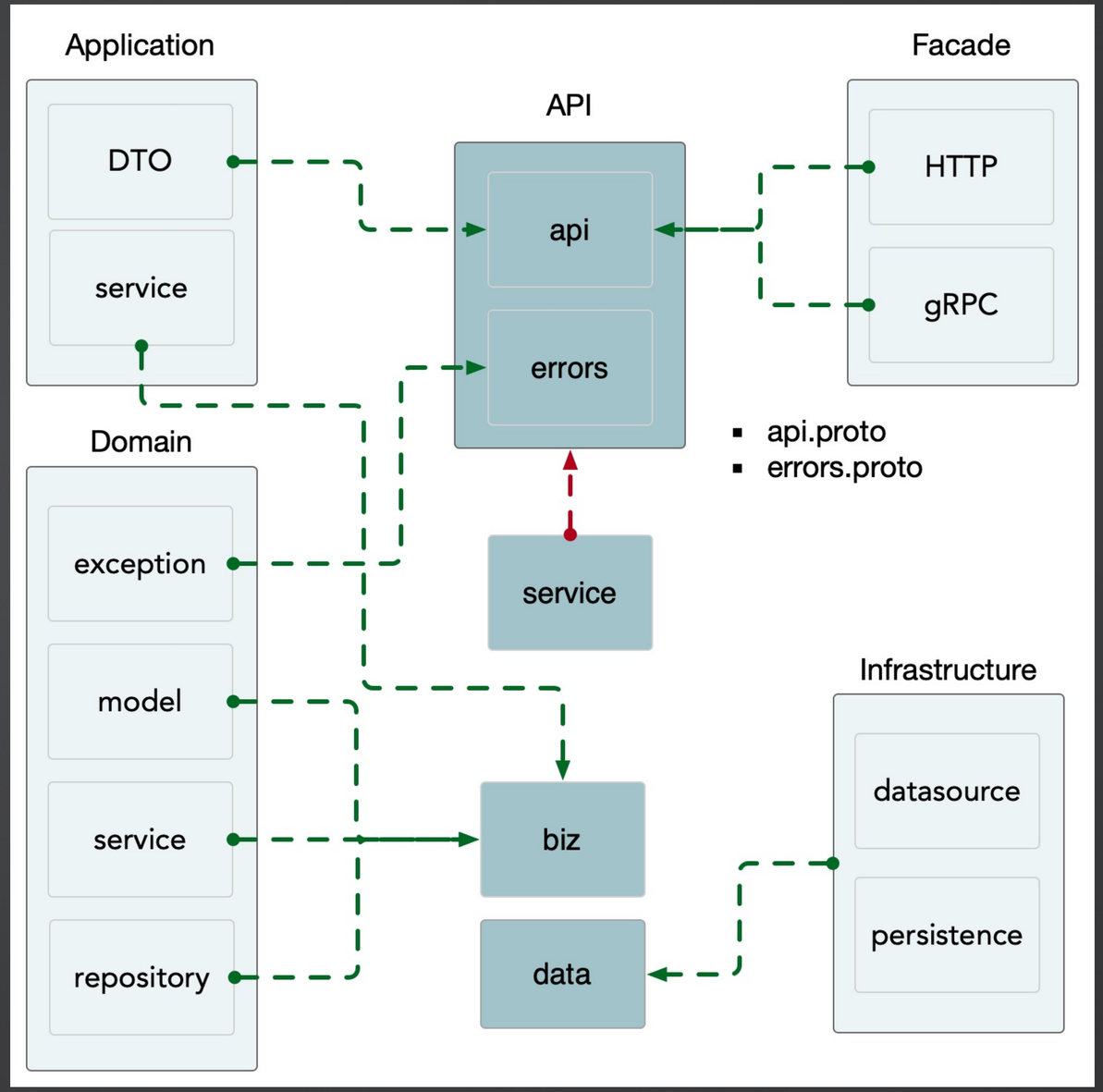
它负责向用户显示信息和解释用户命令，完成前端界面逻辑。这里的用户不一定是使用用户界面的人，也可以是另一个计算机系统（API），即主动适配器。



领域驱动设计

应用层：

它是很薄的一层，负责展现层与领域层之间的协调，也是与其它系统应用层进行交互的必要渠道。它主要负责服务的组合、编排和转发，负责处理业务用例的执行顺序以及结果的拼装，拼装完领域服务后以粗粒度的服务通过 API 对外暴露。应用层除了定义应用服务之外，在这层还可以进行安全认证，权限校验，持久化事务控制或向其他系统发送基于事件的消息通知。

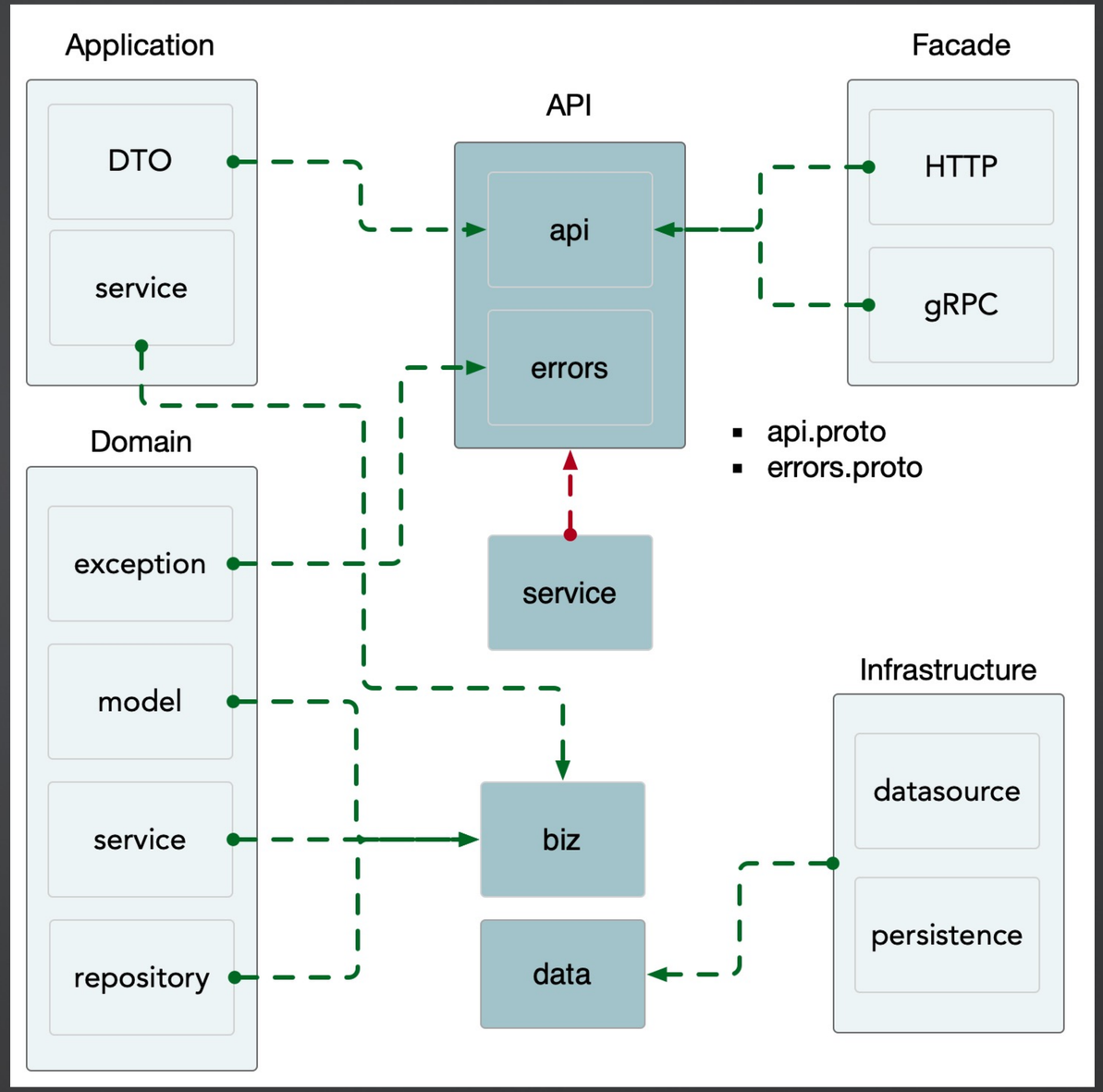


领域驱动设计

领域层：

它是业务软件的核心所在，包含了业务所涉及的领域对象（实体、值对象）、领域服务以及它们之间的关系，负责表达业务概念、业务状态信息以及业务规则，具体表现形式就是领域模型。

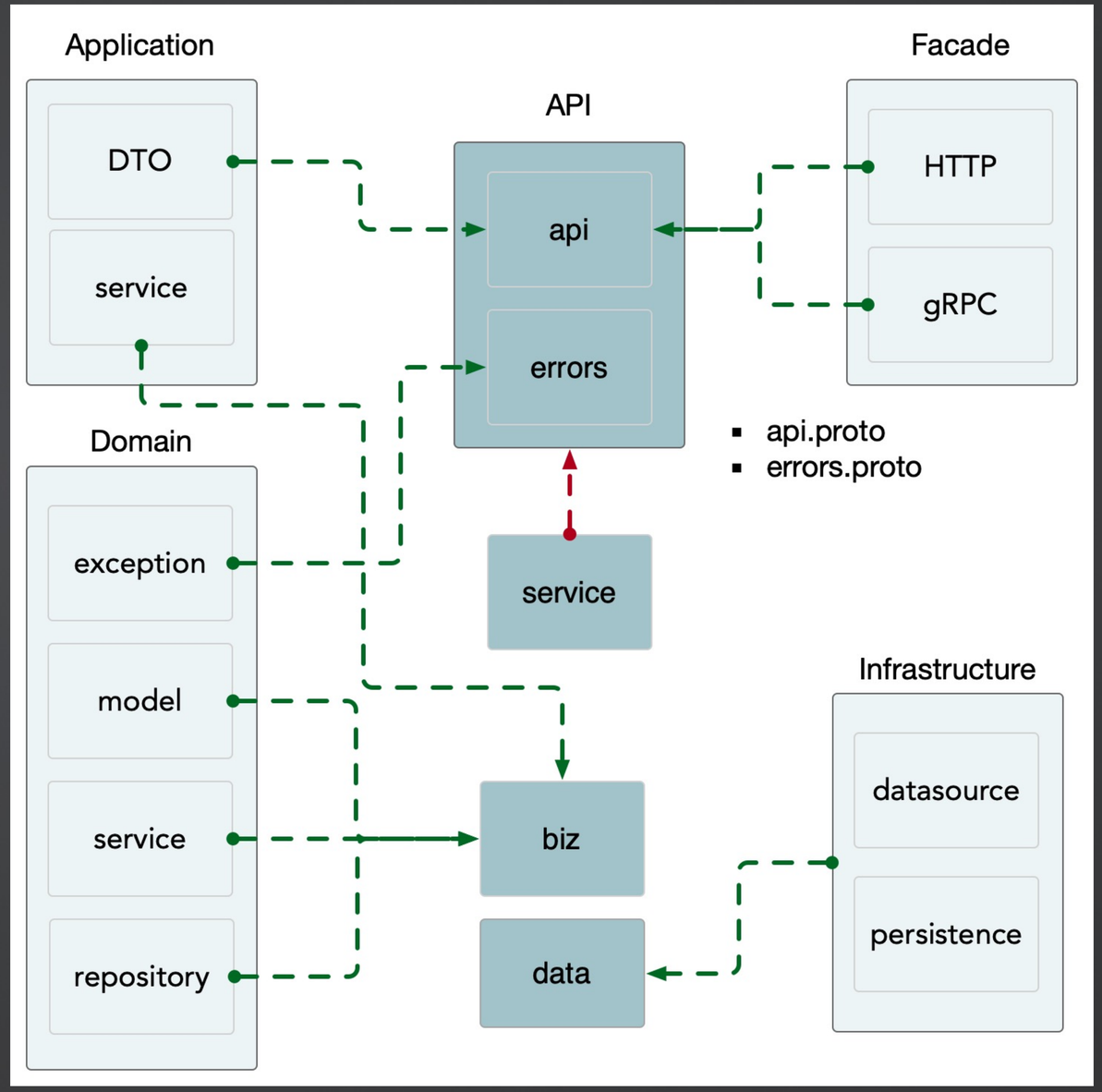
领域服务：当领域中，某个操作过程或转换过程不是实体或者值对象的指责时，我们便应该将操作放在一个单独的接口中，即领域服务。



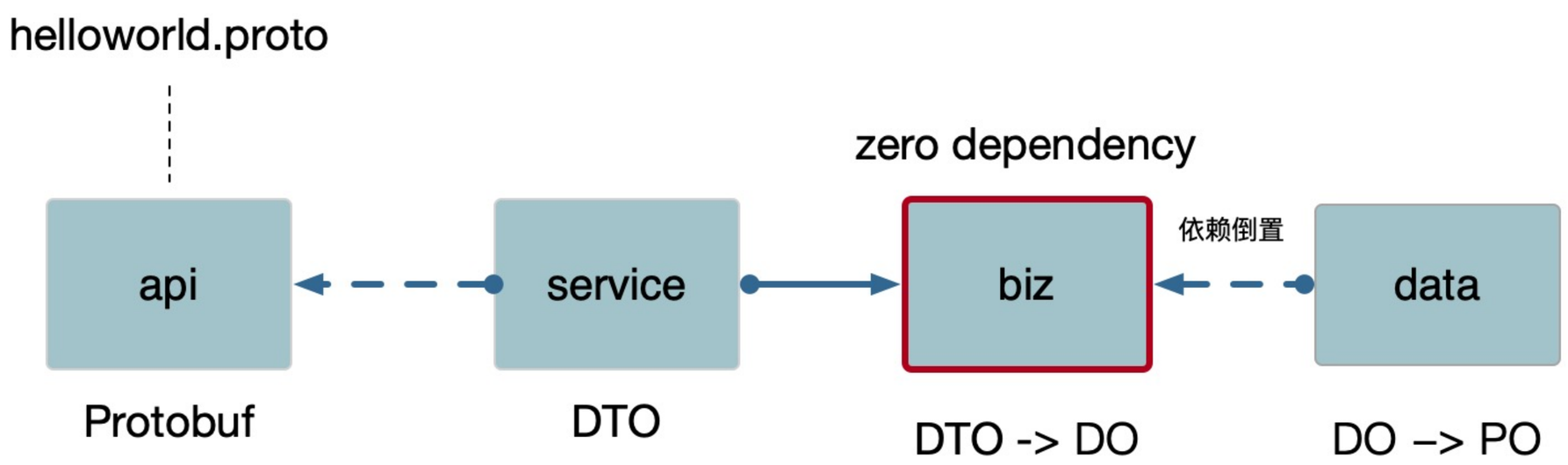
领域驱动设计

基础设施层：

一个系统的基础不仅仅限于对数据库的访问，还包括访问诸如网络、文件、消息队列或者其他硬件设施，因此本层更名为“基础设施层”是非常合理的。它向其他层提供通用的技术能力，为应用层传递消息，为领域层提供持久化机制（如数据库资源）等。即被动适配器。



领域驱动设计



api 为 gRPC/HTTP 的服务定义层。服务的 Server/Client interface，DTO 对象都会使用 protobuf 工具生成。

注: DTO 即数据传输对象(Data Transfer Object)，负责 gRPC/HTTP 传输的消息载体。

把 api.proto 生成的代码放入到该目录中，填写调用具体用例（Usecase）的代码。
该层把 DTO 对象传入到 Usecase 的方法中，即依赖了 biz 层的应用服务层，然后在 Usecase 的方法中来完成转换 DO 的逻辑。

注: DO 即领域对象(Domain Object)。负责业务领域实体逻辑的行为对象载体。

biz 为业务逻辑层，里面定义了 DO 对象，以及 DO 需要使用的 repository，repo 接口定义也在此，谁使用谁定义的原则。通常我们建议 DO 使用领域模型。
该层包含：Domain、Service、Usecase、Repository、Exception，如果微服务比较原子，可以不区分二级目录。如果有复杂的模块（限界上下文）可以按照模块划分。

data 为数据层，针对领域对象的持久化，以及装载出 DO 。为了隔离数据库的表定义和 DO 对象的耦合(实际上单表不完全映射为一个领域对象)。会使用一个 PO 对象来作为存储的载体。如果依赖 gRPC，也要在本层实现（通过 biz 定义的 ACL 接口）。

注: PO 即持久化对象，比如我们使用 facebook ent 的 ORM 库就会强制使用 PO 对象来进行操作。

目录

References

[1] https://www.jianshu.com/p/ae473acea7de

THANKS