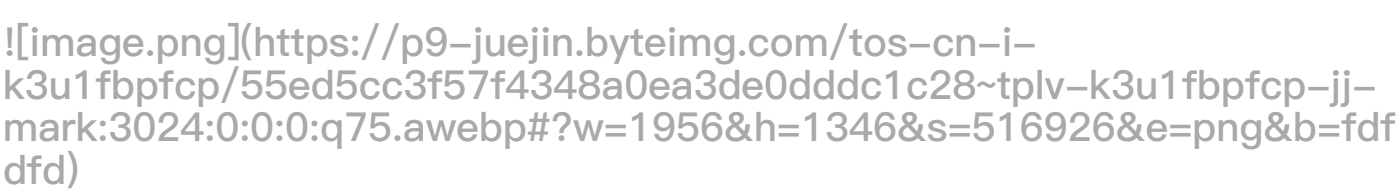
的诉求，同时被多个子域使用的通用功能子域，比如权限认证这种多个子域可以共用。
* 支撑域：必须的，但既不包含决定产品和公司核心竞争力的功能，也不包含通用功能的子域。比如代码类的数据字典。

贫血模型和充血模型



贫血模型

贫血模型具有一堆属性和set get方法，存在的问题就是通过pojo这个对象上看不出业务有哪些逻辑，一个pojo可能被多个模块调用，只能去上层各种各样的service来调用，这样以后当梳理这个实体有什么业务，只能一层一层去搜service，也就是贫血失忆症，不够面向对象。

在长期维护的 MVC 架构的项目中，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。

充血模型

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的若干子问题，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。

比如user用户有改密码，改手机号，修改登录失败次数等操作，都内聚在这个user实体中，每个实体的业务都是清晰的。

```
```

```
@NoArgsConstructor
@Getter
public class User extends Aggregate<Long, User> {

    /**
     * 用户名
     */
    private String userName;

    /**
     * 姓名
     */
    private String realName;

    /**
     * 手机号
     */
    private String phone;

    /**
     * 密码
     */
    private String password;

    /**
     * 锁定结束时间
     */
    private Date lockEndTime;

    /**
     * 登录失败次数
     */
    private Integer failNumber;

    /**
     * 用户角色
     */
    private List<Role> roles;

    /**
     * 部门
     */
    private Department department;

    /**
     *  有与之相关的业务功能，主要表达业务概念，业务状态信息以及业务规则
```

。

2. 真正的业务逻辑都在领域层编写，聚合根负责封装实现业务逻辑，对应用层

暴露领域级别的服务接口。
3. 聚合根不能直接操作其它聚合根，聚合根与聚合根之间只能通过聚合根ID引用；同限界上下文内的聚合之间的领域服务可直接调用；两个限界上下文的交互必须通过应用服务层抽离接口->适配层适配。
4. 跨实体的状态变化，使用领域服务，领域服务不能直接修改实体的状态，只能调用实体的业务方法

DDD 提倡富领域模型，尽量将业务逻辑归属到实体对象上，实在无法归属的部分则设计成领域服务。领域服务会对多个实体或实体方法进行组装和编排，实现跨多个实体的复杂核心业务逻辑。对于严格分层架构，如果单个实体的方法需要对应用层暴露，则需要通过领域服务封装后才能暴露给应用服务

基础层

也叫基础设施层，基础层是贯穿所有层的，它的作用就是为其它各层提供通用的技术和基础服务，包括第三方工具、驱动、消息中间件、网关、文件、缓存以及数据库等。比较常见的功能还是提供数据库持久化。

基础层的服务形态主要是仓储服务。仓储服务包括接口和实现两部分。仓储接口服务供应用层或者领域层服务调用，仓储实现服务，完成领域对象的持久化或数据初始化。

比如说，在传统架构设计中，由于上层应用对数据库的强耦合，很多公司在架构演进中最担忧的可能就是换数据库了，因为一旦更换数据库，就可能需要重写大部分的代码，这对应用来说是致命的。那采用依赖倒置的设计以后(说白了就是多套一层接口)，应用层就可以通过解耦来保持独立的核心业务

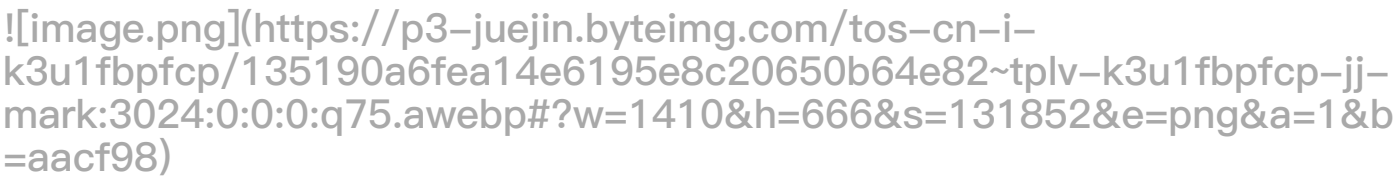
1. 为业务逻辑提供支撑能力，提供通用的技术能力，仓库写增删改查类似DAO。
2. 防腐层实现(封装变化)用于业务检查和隔离第三方服务，内部try catch

防腐层(ACL)

当某个功能模块需要依赖第三方系统提供的数据或者功能时，我们常用的策略就是直接使用外部系统的API、数据结构。这样存在的问题就是，因使用外部系统，而被外部系统的质量问题影响，从而“腐化”本身设计的问题。

因此我们的解决方案就是在两个系统之间加入一个中间层，隔离第三方系统的依赖，对第三方系统进行通讯转换和语义隔离，这个中间层，我们叫它防腐层

说白了就是，两个系统之间加了中间层，中间层类似适配器模式，解决接口差异的对接，接口转换是单向的（即从调用方向被调用方进行接口转换）；防腐层强调两个子系统语义解耦，接口转换是双向的。



如上图所示的引入了防腐层的架构中：

* 子系统A与防腐层之间的通讯，使用子系统A的数据模型和架构；
* 子系统B与防腐层之间的通讯，则使用子系统B的数据模型和架构；
* 防腐层实现了在两个系统之间进行通讯转换的全部逻辑（**双向转换**）。