阿里技术专家详解 DDD 系列- Domain Primitive

淘系技术 (/users/wmj4vfh4u33bc) 2019-09-02 17:59:37 浏览7684

淘宝技术 (/tags/type_blog-tagid_24470/)

> 作者|殷浩
> 出品|阿里巴巴新零售淘系技术部

导读：对于一个架构师来说，在软件开发中如何降低系统复杂度是一个永恒的挑战，无论是 94 年 GoF 的 Design Patterns， 99 年的 Martin Fowler 的 Refactoring， 02 年的 P of EAA，还是 03 年的 Enterprise Integration Patterns，都是通过一系列的设计模式或范例来降低一些常见的复杂度。但是问题在于，这些书的理念是通过技术手段解决技术问题，但并没有从根本上解决业务的问题。所以 03 年 Eric Evans 的 Domain Driven Design 一书，以及后续 Vaughn Vernon 的 Implementing DDD， Uncle Bob 的 Clean Architecture 等书，真正的从业务的角度出发，为全世界绝大部分做纯业务的开发提供了一整套的架构思路。

前言

由于 DDD 不是一套框架，而是一种架构思想，所以在代码层面缺乏了足够的约束，导致 DDD 在实际应用中上手门槛很高，甚至可以说绝大部分人都对 DDD 的理解有所偏差。举个例子， Martin Fowler 在他个人博客里描述的一个 Anti-pattern，Anemic Domain Model (https://yq.aliyun.com/go/articleRenderRedirect?url=https%3A%2F%2Fmartinfowler.com%2Fbliki%2FAnemicDomainModel.html)（贫血域模型）在实际应用当中层出不穷，而一些仍然火热的 ORM 工具比如 Hibernate，Entity Framework 实际上助长了贫血模型的扩散。同样的，传统的基于数据库技术以及 MVC 的四层应用架构（UI、Business、Data Access、Database），在一定程度上和 DDD 的一些概念混淆，导致绝大部分人在实际应用当中仅仅用到了 DDD 的建模的思想，而其对于整个架构体系的思想无法落地。

我第一次接触 DDD 应该是 2012 年，当时除了大型互联网公司，基本上商业应用都还处于单机的时代，服务化的架构还局限于单机 +LB 用 MVC 提供 Rest 接口供外部调用，或者用 SOAP 或 WebServices 做 RPC 调用，但其实更多局限于对外部依赖的协议。让我关注到 DDD 思想的是一个叫 Anti-Corruption Layer（防腐层）的概念，特别是其在解决外部依赖频繁变更的情况下，如何将核心业务逻辑和外部依赖隔离的机制。到了 2014 年， SOA 开始大行其道，微服务的概念开始冒头，而如何将一个 Monolith 应用合理的拆分为多个微服务成为了各大论坛的热门话题，而 DDD 里面的 Bounded

Context（限界上下文）的思想为微服务拆分提供了一套合理的框架。而在今天，在一个所有的东西都能被称之为"服务"的时代（XAAS）， DDD 的思想让我们能冷静下来，去思考到底哪些东西可以被服务化拆分，哪些逻辑需要聚合，才能带来最小的维护成本，而不是简单的去追求开发效率。

所以今天，我开始这个关于 DDD 的一系列文章，希望能继续在总结前人的基础上发扬光大 DDD 的思想，但是通过一套我认为合理的代码结构、框架和约束，来降低 DDD 的实践门槛，提升代码质量、可测试性、安全性、健壮性。

未来会覆盖的内容包括：

- 最佳架构实践：六边形应用架构 / Clean 架构的核心思想和落地方案
- 持续发现和交付：Event Storming > Context Map > Design Heuristics > Modelling
- 降低架构腐败速度：通过 Anti-Corruption Layer 集成第三方库的模块化方案
- 标准组件的规范和边界：Entity, Aggregate, Repository, Domain Service, Application Service, Event, DTO Assembler 等
- 基于 Use Case 重定义应用服务的边界
- 基于 DDD 的微服务化改造及颗粒度控制
- CQRS 架构的改造和挑战
- 基于事件驱动的架构的挑战
- 等等

今天先给大家带来一篇最基础，但极其有价值的Domain Primitive的概念。

Domain Primitive

就好像在学任何语言时首先需要了解的是基础数据类型一样，在全面了解 DDD 之前，首先给大家介绍一个最基础的概念: Domain Primitive（DP）。

Primitive 的定义是：

> 不从任何其他事物发展而来
> 初级的形成或生长的早期阶段

就好像 Integer、String 是所有编程语言的Primitive一样，在 DDD 里， DP 可以说是一切模型、方法、架构的基础，而就像 Integer、String 一样， DP 又是无所不在的。所以，第一讲会对 DP 做一个全面的介绍和分析，但我们先不去讲概念，而是从案例入手，看看为什么 DP 是一个强大的概念。

1、案例分析

我们先看一个简单的例子，这个 case 的业务逻辑如下：

> 一个新应用在全国通过 地推业务员 做推广，需要做一个用户注册系统，同时希望在用户注册后能够通过用户电话（先假设仅限座机）的地域（区号）对业务员发奖金。

先不要去纠结这个根据用户电话去发奖金的业务逻辑是否合理，也先不要去管用户是否应该在注册时和业务员做绑定，这里我们看的主要还是如何更加合理的去实现这个逻辑。一个简单的用户和用户注册的代码实现如下：

```
public class User {
    Long userId;
    String name;
    String phone;
    String address;
    Long repId;
}

public class RegistrationServiceImpl implements RegistrationService {

    private SalesRepRepository salesRepRepo;
    private UserRepository userRepo;

    public User register(String name, String phone, String address)
      throws ValidationException {
        // 校验逻辑
        if (name == null || name.length() == 0) {
            throw new ValidationException("name");
        }
        if (phone == null || !isValidPhoneNumber(phone)) {
            throw new ValidationException("phone");
        }
        // 此处省略address的校验逻辑

        // 取电话号里的区号，然后通过区号找到区域内的SalesRep
        String areaCode = null;
        String[] areas = new String[]{"0571", "021", "010"};
        for (int i = 0; i < phone.length(); i++) {
            String prefix = phone.substring(0, i);
            if (Arrays.asList(areas).contains(prefix)) {
                areaCode = prefix;
                break;
            }
        }
        SalesRep rep = salesRepRepo.findRep(areaCode);

        // 最后创建用户，落盘，然后返回
        User user = new User();
        user.name = name;
        user.phone = phone;
        user.address = address;
        if (rep != null) {
            user.repId = rep.repId;
        }

        return userRepo.save(user);
    }

    private boolean isValidPhoneNumber(String phone) {
        String pattern = "^0[1-9]{2,3}-?\\d{8}$";
        return phone.matches(pattern);
    }
}
```

我们日常绝大部分代码和模型其实都跟这个是类似的，乍一看貌似没啥问题，但我们再深入一步，从以下四个维度去分析一下：接口的清晰度（可阅读性）、数据验证和错误处理、业务逻辑代码的清晰度、和可测试性。

问题1 - 接口的清晰度

在Java代码中，对于一个方法来说所有的参数名在编译时丢失，留下的仅仅是一个参数类型的列表，所以我们重新看一下以上的接口定义，其实在运行时仅仅是：

```
User register(String, String, String);
```

所以以下的代码是一段编译器完全不会报错的，很难通过看代码就能发现的 bug：

```
service.register("殷浩", "浙江省杭州市余杭区文三西路969号", "0571-12345678");
```

当然，在真实代码中运行时会报错，但这种 bug 是在运行时被发现的，而不是在编译时。普通的 Code Review 也很难发现这种问题，很有可能是代码上线后才会被暴露出来。这里的思考是，有没有办法在编码时就避免这种可能会出现的问题？

另外一种常见的，特别是在查询服务中容易出现的例子如下：

```
User findByName(String name);
User findByPhone(String phone);
User findByNameAndPhone(String name, String phone);
```

在这个场景下，由于入参都是 String 类型，不得不在方法名上面加上 ByXXX 来区分，而 findByNameAndPhone 同样也会陷入前面的入参顺序错误的问题，而且和前面的入参不同，这里参数顺序如果输错了，方法不会报错只会返回 null，而这种 bug 更加难被发现。这里的思考是，**有没有办法让方法入参一目了然，避免入参错误导致的 bug？**

问题2 - 数据验证和错误处理

在前面这段数据校验代码：

```
if (phone == null || !isValidPhoneNumber(phone)) {
    throw new ValidationException("phone");
}
```

在日常编码中经常会出现，一般来说这种代码需要出现在方法的最前端，确保能够 fail-fast 。但是假设你有多个类似的接口和类似的入参，在每个方法里这段逻辑会被重复。而更严重的是如果未来我们要拓展电话号去包含手机时，很可能需要加入以下代码：

```
if (phone == null || !isValidPhoneNumber(phone) || !isValidCellNumber(phone)) {
    throw new ValidationException("phone");
}
```

如果你有很多个地方用到了 phone 这个入参，但是有个地方忘记修改了，会造成 bug 。这是一个 DRY 原则被违背时经常会发生的问题。

如果有个新的需求，需要把入参错误的原因返回，那么这段代码就变得更加复杂：

```
if (phone == null) {
    throw new ValidationException("phone不能为空");
} else if (!isValidPhoneNumber(phone)) {
    throw new ValidationException("phone格式错误");
}
```

可以想像得到，代码里充斥着大量的类似代码块时，维护成本要有多高。

最后，在这个业务方法里，会（隐性或显性的）抛 ValidationException，所以需要外部调用方去try/catch，而业务逻辑异常和数据校验异常被混在了一起，是否是合理的？

在传统Java架构里有几个办法能够去解决一部分问题，常见的如BeanValidation注解或ValidationUtils类，比如：

```
// Use Bean Validation
User registerWithBeanValidation(
  @NotNull @NotBlank String name,
  @NotNull @Pattern(regexp = "^0?[1-9]{2,3}-?\\d{8}$") String phone,
  @NotNull String address
);

// Use ValidationUtils:
public User registerWithUtils(String name, String phone, String address) {
    ValidationUtils.validateName(name); // throws ValidationException
    ValidationUtils.validatePhone(phone);
    ValidationUtils.validateAddress(address);
    ...
}
```

但这几个传统的方法同样有问题，

BeanValidation：

- 通常只能解决简单的校验逻辑，复杂的校验逻辑一样要写代码实现定制校验器
- 在添加了新校验逻辑时，同样会出现在某些地方忘记添加一个注解的情况，DRY原则还是会被违背

ValidationUtils类：

- 当大量的校验逻辑集中在一个类里之后，违背了Single Responsibility单一性原则，导致代码混乱和不可维护
- 业务异常和校验异常还是会混杂

所以，**有没有一种方法，能够一劳永逸的解决所有校验的问题以及降低后续的维护成本和异常处理成本呢?
**

问题3 - 业务代码的清晰度

在这段代码里：

```
String areaCode = null;
String[] areas = new String[]{"0571", "021", "010"};
for (int i = 0; i < phone.length(); i++) {
    String prefix = phone.substring(0, i);
    if (Arrays.asList(areas).contains(prefix)) {
        areaCode = prefix;
        break;
    }
}
SalesRep rep = salesRepRepo.findRep(areaCode);
```

实际上出现了另外一种常见的情况，那就是从一些入参里抽取一部分数据，然后调用一个外部依赖获取更多的数据，然后通常从新的数据中再抽取部分数据用作其他的作用。这种代码通常被称作"胶水代码"，其本质是由于外部依赖的服务的入参并不符合我们原始的入参导致的。比如，如果SalesRepRepository包含一个findRepByPhone的方法，则上面大部分的代码都不必要了。

所以，一个常见的办法是将这段代码抽离出来，变成独立的一个或多个方法：

```
private static String findAreaCode(String phone) {
    for (int i = 0; i < phone.length(); i++) {
        String prefix = phone.substring(0, i);
        if (isAreaCode(prefix)) {
            return prefix;
        }
    }
    return null;
}

private static boolean isAreaCode(String prefix) {
    String[] areas = new String[]{"0571", "021"};
    return Arrays.asList(areas).contains(prefix);
}
```

然后原始代码变为：

```
String areaCode = findAreaCode(phone);
SalesRep rep = salesRepRepo.findRep(areaCode);
```

而为了复用以上的方法，可能会抽离出一个静态工具类 PhoneUtils 。但是这里要思考的是，静态工具类是否是最好的实现方式呢？当你的项目里充斥着大量的静态工具类，业务代码散在多个文件当中时，你是否还能找到核心的业务逻辑呢？

问题4 - 可测试性

为了保证代码质量，每个方法里的每个入参的每个可能出现的条件都要有 TC 覆盖（假设我们先不去测试内部业务逻辑），所以在我们这个方法里需要以下的 TC：

条件 \ 入参	name	phone	address
入参为null	✓	✓	✓
入参为空	✓	✓	✓
入参不符合要求（可能多个）	✓	✓	✓

假如一个方法有 N 个参数，每个参数有 M 个校验逻辑，至少要有 N * M 个 TC 。

如果这时候在该方法中加入一个新的入参字段 fax ，即使 fax 和 phone 的校验逻辑完全一致，为了保证 TC 覆盖率，也一样需要 M 个新的 TC 。

而假设有 P 个方法中都用到了 phone 这个字段，这 P 个方法都需要对该字段进行测试，也就是说整体需要：

P N M

个测试用例才能完全覆盖所有数据验证的问题，在日常项目中，这个测试的成本非常之高，导致大量的代码没被覆盖到。而没被测试覆盖到的代码才是最有可能出现问题的地方。

在这个情况下，降低测试成本 == 提升代码质量，如何能够降低测试的成本呢？

2、解决方案

我们回头先重新看一下原始的 use case，并且标注其中可能重要的概念：

一个新应用在全国通过 地推业务员 做推广，需要做一个用户的注册系统，在用户注册后能够通过用户电话号的区号对业务员发奖金。

在分析了 use case 后，发现其中地推业务员、用户本身自带 ID 属性，属于 Entity（实体），而注册系统属于 Application Service（应用服务），这几个概念已经有存在。但是发现电话号这个概念却完全被隐藏到了代码之中。我们可以问一下自己，取电话号的区号的逻辑是否属于用户（用户的区号？）？是否属于注册服务（注册的区号？）？如果都不是很贴切，那就说明这个逻辑应该属于一个独立的概念。所以这里引入我们第一个原则：

Make Implicit Concepts Explicit

将隐性的概念显性化

在这里，我们可以看到，原来电话号仅仅是用户的一个参数，属于隐形概念，但实际上电话号的区号才是真正的业务逻辑，而我们需要将电话号的概念显性化，通过写一个Value Object：

```
public class PhoneNumber {

    private final String number;
    public String getNumber() {
        return number;
    }

    public PhoneNumber(String number) {
        if (number == null) {
            throw new ValidationException("number不能为空");
        } else if (isValid(number)) {
            throw new ValidationException("number格式错误");
        }
        this.number = number;
    }

    public String getAreaCode() {
        for (int i = 0; i < number.length(); i++) {
            String prefix = number.substring(0, i);
            if (isAreaCode(prefix)) {
                return prefix;
            }
        }
        return null;
    }

    private static boolean isAreaCode(String prefix) {
        String[] areas = new String[]{"0571", "021", "010"};
        return Arrays.asList(areas).contains(prefix);
    }

    public static boolean isValid(String number) {
        String pattern = "^0?[1-9]{2,3}-?\\d{8}$";
        return number.matches(pattern);
    }

}
```

这里面有几个很重要的元素：

通过 private final String number 确保 PhoneNumber 是一个（Immutable）Value Object。（一般来说 VO 都是 Immutable 的，这里只是重点强调一下）

校验逻辑都放在了 constructor 里面，确保只要 PhoneNumber 类被创建出来后，一定是校验通过的。

之前的 findAreaCode 方法变成了 PhoneNumber 类里的 getAreaCode，突出了 areaCode 是 PhoneNumber 的一个计算属性。

这样做完之后，我们发现把 PhoneNumber 显性化之后，其实是生成了一个 Type（数据类型）和一个 Class（类）：

- Type 指我们在今后的代码里可以通过 PhoneNumber 去显性的标识电话号这个概念
- Class 指我们可以把所有跟电话号相关的逻辑完整的收集到一个文件里

这两个概念加起来，构造成了本文标题的 Domain Primitive（DP）。

我们看一下全面使用了 DP 之后效果：

```
public class User {
    UserId userId;
    Name name;
    PhoneNumber phone;
    Address address;
    RepId repId;
}

public User register(
  @NotNull Name name,
  @NotNull PhoneNumber phone,
  @NotNull Address address
) {
    // 找到区域内的SalesRep
    SalesRep rep = salesRepRepo.findRep(phone.getAreaCode());

    // 最后创建用户，落盘，然后返回，这部分代码实际上也能用Builder解决
    User user = new User();
    user.name = name;
    user.phone = phone;
    user.address = address;
    if (rep != null) {
        user.repId = rep.repId;
    }

    return userRepo.saveUser(user);
}
```

我们可以看到在使用了 DP 之后，所有的数据验证逻辑和非业务流程的逻辑都消失了，剩下都是核心业务逻辑，可以一目了然。我们重新用上面的四个维度评估一下：

评估1 - 接口的清晰度

重构后的方法签名变成了很清晰的：

```
public User register(Name, PhoneNumber, Address)
```

而之前容易出现的bug，如果按照现在的写法

```
service.register(new Name("殷浩"), new Address("浙江省杭州市余杭区文三西路969号"), new PhoneNu
```

让接口 API 变得很干净，易拓展。

评估2 - 数据验证和错误处理

```
public User register(
  @NotNull Name name,
  @NotNull PhoneNumber phone,
  @NotNull Address address
) // no throws
```

如前文代码展示的，重构后的方法里，完全没有了任何数据验证的逻辑，也不会抛 ValidationException 。原因是因为 DP 的特性，只要是能够带到入参里的一定是正确的或 null（Bean Validation 或 lombok 的注解能解决 null 的问题）。所以我们把数据验证的工作量前置到了调用方，而调用方本来就是应该提供合法数据的，所以更加合适。

再展开来看，使用DP的另一个好处就是代码遵循了 DRY 原则和单一性原则，如果未来需要修改 PhoneNumber 的校验逻辑，只需要在一个文件里修改即可，所有使用到了 PhoneNumber 的地方都会生效。

评估3 - 业务代码的清晰度

```
SalesRep rep = salesRepRepo.findRep(phone.getAreaCode());
User user = xxx;
return userRepo.save(user);
```

除了在业务方法里不需要校验数据之外，原来的一段胶水代码 findAreaCode 被改为了 PhoneNumber 类的一个计算属性 getAreaCode ，让代码清晰度大大提升。而且胶水代码通常都不可复用，但是使用了 DP 后，变成了可复用、可测试的代码。我们能看到，在刨除了数据验证代码、胶水代码之后，剩下的都是核心业务逻辑。（ Entity 相关的重构在后面文章会谈到，这次先忽略）

评估4 - 可测试性

条件 \ 入参	PhoneNumber	phone	fax
入参为null	✓	✓	✓
入参为空	✓		
入参不符合要求（可能多个）	✓		

当我们将 PhoneNumber 抽取出来之后，在来看测试的 TC：

- 首先 PhoneNumber 本身还是需要 M 个测试用例，但是由于我们只需要测试单一对象，每个用例的代码量会大大降低，维护成本降低。
- 每个方法里的每个参数，现在只需要覆盖为 null 的情况就可以了，其他的 case 不可能发生（因为只要不是 null 就一定是合法的）

所以，单个方法的 TC 从原来的 N * M 变成了今天的 N + M 。同样的，多个方法的 TC 数量变成了

N + M + P

这个数量一般来说要远低于原来的数量 N M P ，让测试成本极大的降低。

评估总结

3、进阶使用

在上文我介绍了 DP 的第一个原则：将隐性的概念显性化。在这里我将介绍 DP 的另外两个原则，用一个新的案例。

案例1 - 转账

假设现在要实现一个功能，让A用户可以支付 x 元给用户 B ，可能的实现如下：

```
public void pay(BigDecimal money, Long recipientId) {
    BankService.transfer(money, "CNY", recipientId);
}
```

如果这个是境内转账，并且境内的货币永远不变，该方法貌似没啥问题，但如果有一天货币变更了（比如欧元区曾经出现的问题），或者我们需要做跨境转账，该方法是明显的 bug ，因为 money 对应的货币不一定是 CNY 。

在这个 case 里，当我们说“支付 x 元”时，除了 x 本身的数字之外，实际上是有一个隐含的概念那就是货币“元”。但是在原始的入参里，之所以只用了 BigDecimal 的原因是我们认为 CNY 货币是默认的，是一个隐含的条件，但是在我们写代码时，需要把所有隐性的条件显性化，而这些条件整体组成当前的上下文。所以 DP 的第二个原则是：

Make Implicit Context Explicit

将 隐性的 上下文 显性化

所以当我们做这个支付功能时，实际上需要的一个入参是支付金额 + 支付货币。我们可以把这两个概念组合成为一个独立的完整概念：Money。

```
@Value
public class Money {
    private BigDecimal amount;
    private Currency currency;
    public Money(BigDecimal amount, Currency currency) {
        this.amount = amount;
        this.currency = currency;
    }
}
```

而原有的代码则变为：

```
public void pay(Money money, Long recipientId) {
    BankService.transfer(money, recipientId);
}
```

通过将默认货币这个隐性的上下文概念显性化，并且和金额合并为 Money ，我们可以避免很多当前看不出来，但未来可能会暴雷的bug。

案例2 - 跨境转账

前面的案例升级一下，假设用户可能要做跨境转账从 CNY 到 USD ，并且货币汇率随时在波动：

```
public void pay(Money money, Currency targetCurrency, Long recipientId) {
    if (money.getCurrency().equals(targetCurrency)) {
        BankService.transfer(money, recipientId);
    } else {
        BigDecimal rate = ExchangeService.getRate(money.getCurrency(), targetCurrency);
        BigDecimal targetAmount = money.getAmount().multiply(new BigDecimal(rate));
        Money targetMoney = new Money(targetAmount, targetCurrency);
        BankService.transfer(targetMoney, recipientId);
    }
}
```

在这个case里，由于 targetCurrency 不一定和 money 的 Curreny 一致，需要调用一个服务去取汇率，然后做计算。最后用计算后的结果做转账。

这个case最大的问题在于，金额的计算被包含在了支付的服务中，涉及到的对象也有2个 Currency ，2 个 Money ， 1 个 BigDecimal ，总共 5 个对象。这种涉及到多个对象的业务逻辑，需要用 DP 包装掉，所以这里引出 DP 的第三个原则：

Encapsulate Multi-Object Behavior

封装 多对象 行为

在这个 case 里，可以将转换汇率的功能，封装到一个叫做 ExchangeRate 的 DP 里：

```
@Value
public class ExchangeRate {
    private BigDecimal rate;
    private Currency from;
    private Currency to;

    public ExchangeRate(BigDecimal rate, Currency from, Currency to) {
        this.rate = rate;
        this.from = from;
        this.to = to;
    }

    public Money exchange(Money fromMoney) {
        notNull(fromMoney);
        isTrue(this.from.equals(fromMoney.getCurrency()));
        BigDecimal targetAmount = fromMoney.getAmount().multiply(rate);
        return new Money(targetAmount, to);
    }
}
```

ExchangeRate 汇率对象，通过封装金额计算逻辑以及各种校验逻辑，让原始代码变得极其简单：

```
public void pay(Money money, Currency targetCurrency, Long recipientId) {
    ExchangeRate rate = ExchangeService.getRate(money.getCurrency(), targetCurrency);
    Money targetMoney = rate.exchange(money);
    BankService.transfer(targetMoney, recipientId);
}
```

4、讨论和总结

Domain Primitive 的定义

让我们重新来定义一下 Domain Primitive： Domain Primitive 是一个在特定领域里，拥有精准定义的、可自我验证的、拥有行为的 Value Object 。

- DP是一个传统意义上的Value Object，拥有Immutable的特性
- DP是一个完整的概念整体，拥有精准定义
- DP使用业务域中的原生语言
- DP可以是业务域的最小组成部分、也可以构建复杂组合

注：Domain Primitive的概念和命名来自于Dan Bergh Johnsson & Daniel Deogun的书 Secure by Design。

使用 Domain Primitive 的三原则

- 让隐性的概念显性化
- 让隐性的上下文显性化
- 封装多对象行为

Domain Primitive 和 DDD 里 Value Object 的区别

在 DDD 中， Value Object 这个概念其实已经存在：

- 在 Evans 的 DDD 蓝皮书中，Value Object 更多的是一个非 Entity 的值对象
- 在Vernon的IDDD红皮书中，作者更多的关注了Value Object的Immutability、Equals方法、Factory方法等

Domain Primitive 是 Value Object 的进阶版，在原始 VO 的基础上要求每个 DP 拥有概念的整体，而不仅仅是值对象。在 VO 的 Immutable 基础上增加了 Validity 和行为。当然同样的要求无副作用（side-effect free）。

Domain Primitive 和 Data Transfer Object (DTO) 的区别

在日常开发中经常会碰到的另一个数据结构是 DTO，比如方法的入参和出参。DP 和 DTO 的区别如下：

	DTO	DP
功能	数据传输 属于技术细节	代表业务域中的概念
数据的关联	只是一堆数据放在一起 不一定有关联度	数据之间的高相关性
行为	无行为	丰富的行为和业务逻辑

什么情况下应该用 Domain Primitive

常见的 DP 的使用场景包括：

- 有格式限制的 String：比如Name，PhoneNumber，OrderNumber，ZipCode，Address等
- 有限制的Integer：比如OrderId（>0），Percentage（0-100%），Quantity（>=0）等
- 可枚举的 int：比如 Status（一般不用Enum因为反序列化问题）
- Double 或 BigDecimal：一般用到的 Double 或 BigDecimal 都是有业务含义的，比如 Temperature、Money、Amount、ExchangeRate、Rating 等
- 复杂的数据结构：比如 Map> 等，尽量能把 Map 的所有操作包装掉，仅暴露必要行为

5、实战 - 老应用重构的流程

在新应用中使用 DP 是比较简单的，但在老应用中使用 DP 是可以遵循以下流程按部就班的升级。在此用本文的第一个 case 为例。

第一步 - 创建 Domain Primitive，收集所有 DP 行为

在前文中，我们发现取电话号的区号这个是一个可以独立出来的、可以放入 PhoneNumber 这个 Class 的逻辑。类似的，在真实的项目中，以前散落在各个服务或工具类里面的代码，可以都抽出来放在 DP 里，成为 DP 自己的行为或属性。这里面的原则是：所有抽离出来的方法要做到无状态，比如原来是 static 的方法。如果原来的方法有状态变更，需要将改变状态的部分和不改状态的部分分离，然后将无状态的部分融入 DP 。因为 DP 本身不能带状态，所以一切需要改变状态的代码都不属于 DP 的范畴。

(代码参考 PhoneNumber 的代码，这里不再重复)

第二步 - 替换数据校验和无状态逻辑

为了保障现有方法的兼容性，在第二步不会去修改接口的签名，而是通过代码替换原有的校验逻辑和根 DP 相关的业务逻辑。比如：

```
public User register(String name, String phone, String address)
        throws ValidationException {
    if (name == null || name.length() == 0) {
        throw new ValidationException("name");
    }
    if (phone == null || !isValidPhoneNumber(phone)) {
        throw new ValidationException("phone");
    }

    String areaCode = null;
    String[] areas = new String[]{"0571", "021", "010"};
    for (int i = 0; i < phone.length(); i++) {
        String prefix = phone.substring(0, i);
        if (Arrays.asList(areas).contains(prefix)) {
            areaCode = prefix;
            break;
        }
    }
    SalesRep rep = salesRepRepo.findRep(areaCode);
    // 其他代码...
}
```

通过 DP 替换代码后：

```
public User register(String name, String phone, String address)
        throws ValidationException {

    Name _name = new Name(name);
    PhoneNumber _phone = new PhoneNumber(phone);
    Address _address = new Address(address);

    SalesRep rep = salesRepRepo.findRep(_phone.getAreaCode());
    // 其他代码...
}
```

通过 new PhoneNumber(phone) 这种代码，替代了原有的校验代码。

通过 _phone.getAreaCode() 替换了原有的无状态的业务逻辑。

第三步 - 创建新接口

创建新接口，将DP的代码提升到接口参数层：

```
public User register(Name name, PhoneNumber phone, Address address) {
    SalesRep rep = salesRepRepo.findRep(phone.getAreaCode());
}
```

第四步 - 修改外部调用

外部调用方需要修改调用链路，比如：

```
service.register("殷浩", "0571-12345678", "浙江省杭州市余杭区文三西路969号");
```

改为：

```
service.register(new Name("殷浩"), new PhoneNumber("0571-12345678"), new Address("浙江省杭州
```

通过以上 4 步，就能让你的代码变得更加简洁、优雅、健壮、安全。你还在等什么？今天就去尝试吧！

更多技术干货，关注**「淘宝技术」**微信公众号~