02-架构分层：我们为什么一定要这么做？

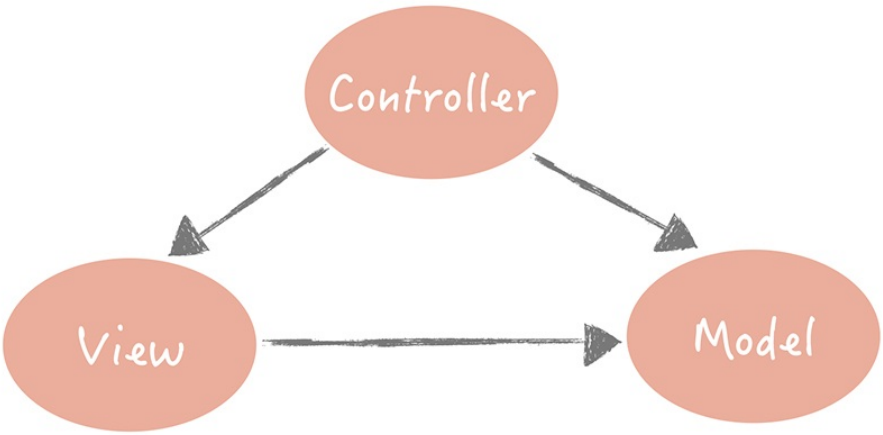
在系统从0到1的阶段，为了让系统快速上线，我们通常是不考虑分层的。但是随着业务越来越复杂，大量的代码纠缠在一起，会出现逻辑不清晰、各模块相互依赖、代码扩展性差、改动一处就牵一发而动全身等问题。

这时，对系统进行分层就会被提上日程，那么我们要如何对架构进行分层？架构分层和高并发架构设计又有什么关系呢？本节课，我将带你寻找答案。

什么是分层架构

软件架构分层在软件工程中是一种常见的设计方式，它是将整体系统拆分成N个层次，每个层次有独立的职责，多个层次协同提供完整的功能。

我们在刚刚成为程序员的时候，会被“教育”说系统的设计要是“MVC”（Model-View-Controller）架构。它将整体的系统分成了Model（模型），View（视图）和Controller（控制器）三个层次，也就是将用户视图和业务处理隔离开，并且通过控制器连接起来，很好地实现了表现和逻辑的解耦，是一种标准的软件分层架构。

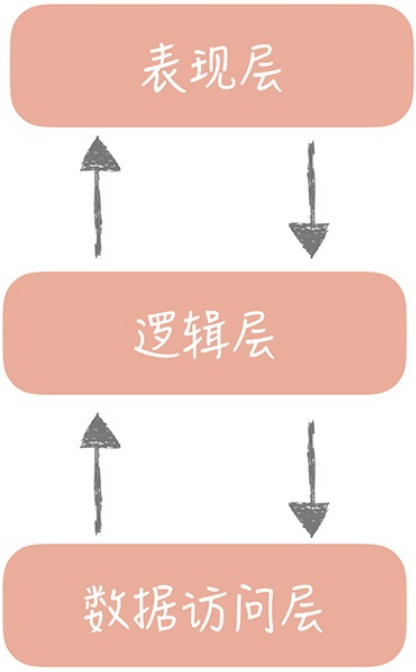


MVC架构图

另外一种常见的分层方式是将整体架构分为表现层、逻辑层和数据访问层：

- 表现层，顾名思义嘛，就是展示数据结果和接受用户指令的，是最靠近用户的一层；
- 逻辑层里面有复杂业务的具体实现；
- 数据访问层则是主要处理和存储之间的交互。

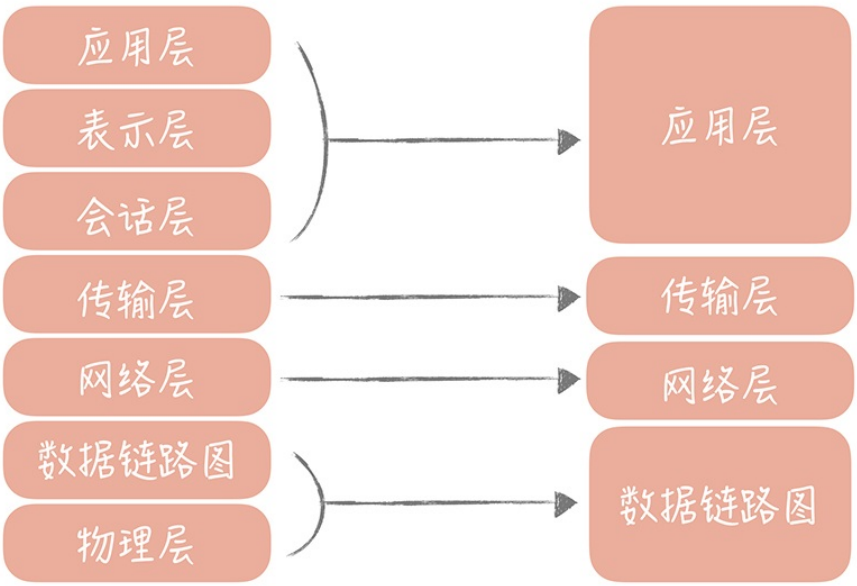
这是在架构上最简单的一种分层方式。其实，我们在不经意间已经按照三层架构来做系统分层设计了，比如在构建项目的时候，我们通常会建立三个目录：Web、Service和Dao，它们分别对应了表现层、逻辑层还有数据访问层。



三层架构示意图

除此之外，如果我们稍加留意，就可以发现很多的分层的例子。比如我们在大学中学到的OSI网络模型，它把整个网络分了七层，自下而上分别是物理层、数据链路层、网络层、传输层、会话层、表示层和应用层。

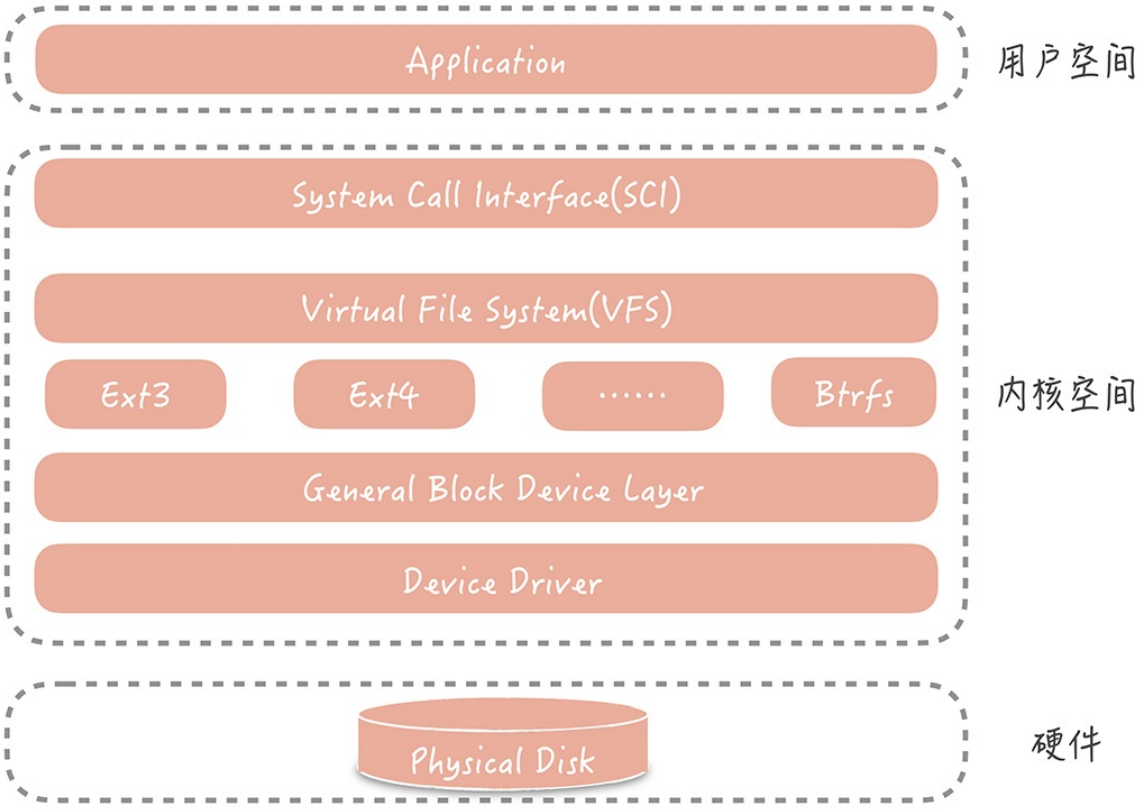
工作中经常能用到TCP/IP协议，它把网络简化成了四层，即链路层、网络层、传输层和应用层。每一层各司其职又互相帮助，网络层负责端到端的寻址和建立连接，传输层负责端到端的数据传输等，同时呢相邻两层还会有数据的交互。这样可以隔离关注点，让不同的层专注做不同的事情。



网络分层模型

Linux文件系统也是分层设计的，从下图你可以清晰地看出文件系统的层次。在文件系统的最上层是虚拟文件系统（VFS），用来屏蔽不同的文件系统之间的差异，提供统一的系统调用接口。虚拟文件系统的下层是Ext3、Ext4等各种文件系统，再向下是为了屏蔽不同硬件设备的实现细节，我们抽象出来的单独的一层——通用块设备层，然后就是不同类型的磁盘了。

我们可以看到，某些层次负责的是对下层不同实现的抽象，从而对上层屏蔽实现细节。比方说VFS对上层（系统调用层）来说提供了统一的调用接口，同时对下层中不同的文件系统规约了实现模型，当新增一种文件系统实现的时候，只需要按照这种模型来设计，就可以无缝插入到Linux文件系统中。



文件系统层次图

那么，为什么这么多系统一定要做分层的设计呢？答案是分层设计存在一定的优势。

分层有什么好处

分层的设计可以简化系统设计，让不同的人专注做某一层次的事情。想象一下，如果你要设计一款网络程序却没有分层，该是一件多么痛苦的事情。

因为你必须是一个通晓网络的全才，要知道各种网络设备的接口是什么样的，以便可以将数据包发送给它。你还要关注数据传输的细节，并且需要处理类似网络拥塞，数据超时重传这样的复杂问题。当然了，你更需要关注数据如何在网络上安全传输，不会被别人窥探和篡改。

而有了分层的设计，你只需要专注设计应用层的程序就可以了，其他的，都可以交给下面几层来完成。

再有，分层之后可以做到很高的复用。比如，我们在设计系统A的时候，发现某一层具有一定的通用性，那么我们可以把它抽取独立出来，在设计系统B的时候使用起来，这样可以减少研发周期，提升研发的效率。

最后一点，分层架构可以让我们更容易做横向扩展。如果系统没有分层，当流量增加时我们需要针对整体系统来做扩展。但是，如果我们按照上面提到的三层架构将系统分层后，那么我们就可以针对具体的问题来做细致的扩展。

比如说，业务逻辑里面包含有比较复杂的计算，导致CPU成为性能的瓶颈，那这样就可以把逻辑层单独抽取出来独立部署，然后只对逻辑层来做扩展，这相比于针对整体系统扩展所付出的代价就要小的多了。

这一点也可以解释我们课程开始时提出的问题：架构分层究竟和高并发设计的关系是怎样的？在“01 | 高并发系统：它的通用设计方法是什么？”中我们了解到，横向扩展是高并发系统设计的常用方法之一，既然分层的架构可以为横向扩展提供便捷， 那么支撑高并发的系统一定是分层的系统。

如何来做系统分层

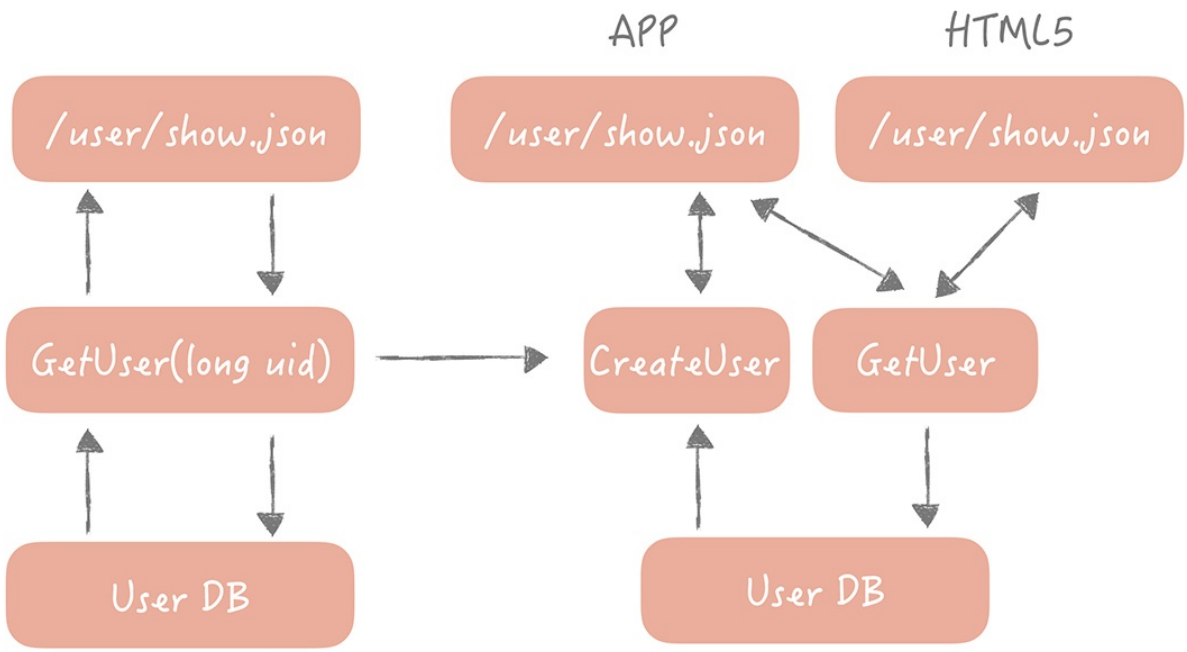
说了这么多分层的优点，那么当我们要做分层设计的时候，需要考虑哪些关键因素呢？

在我看来，最主要的一点就是你需要理清楚每个层次的边界是什么。你也许会问：“如果按照三层架构来分层的话，每一层的边界不是很容易就界定吗？”

没错，当业务逻辑简单时，层次之间的边界的确清晰，开发新的功能时也知道哪些代码要往哪儿写。但是当业务逻辑变得越来越复杂时，边界就会变得越来越模糊，给你举个例子。

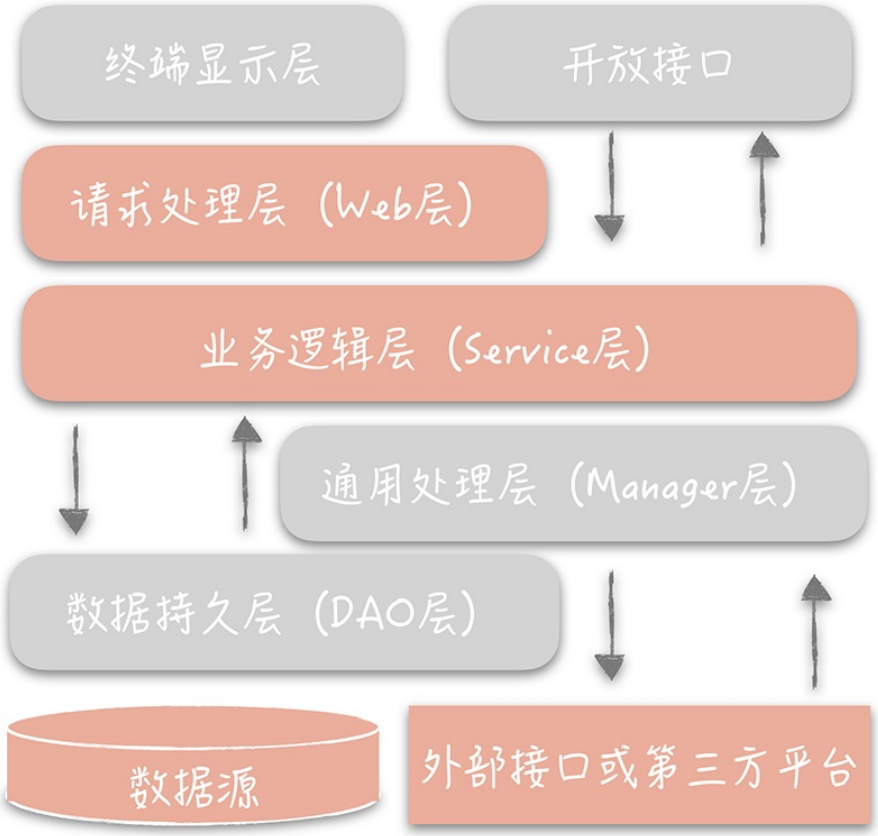
任何一个系统中都有用户系统，最基本的接口是返回用户信息的接口，它调用逻辑层的GetUser方法，GetUser方法又和User DB交互获取数据，就像下图左边展示的样子。

这时，产品提出一个需求，在APP中展示用户信息的时候，如果用户不存在，那么要自动给用户创建一个用户。同时，要做一个HTML5的页面，HTML5页面要保留之前的逻辑，也就是不需要创建用户。这时逻辑层的边界就变得不清晰，表现层也承担了一部分的业务逻辑（将获取用户和创建用户接口编排起来）。



获取用户信息接口的进化

那我们要如何做呢？参照阿里发布的《阿里巴巴Java开发手册v1.4.0（详尽版）》，我们可以将原先的三层架构细化成下面的样子：



阿里系统分层的规约

我来解释一下这个分层架构中的每一层的作用。

- 终端显示层：各端模板渲染并执行显示的层。当前主要是 Velocity 渲染，JS 渲染， JSP 渲染，移动端展示等。
- 开放接口层：将Service层方法封装成开放接口，同时进行网关安全控制和流量控制等。
- Web层：主要是对访问控制进行转发，各类基本参数校验，或者不复用的业务简单处理等。
- Service层：业务逻辑层。
- Manager 层：通用业务处理层。这一层主要有两个作用，其一，你可以将原先Service层的一些通用能力下沉到这一层，比如与缓存和存储交互策略，中间件的接入；其二，你也可以在这一层封装对第三方接口的调用，比如调用支付服务，调用审核服务等。
- DAO层：数据访问层，与底层 MySQL、Oracle、Hbase 等进行数据交互。
- 外部接口或第三方平台：包括其它部门 RPC 开放接口，基础平台，其它公司的 HTTP 接口。

在这个分层架构中主要增加了Manager层，它与Service层的关系是：Manager层提供原子的服务接口，Service层负责依据业务逻辑来编排原子接口。

以上面的例子来说，Manager层提供创建用户和获取用户信息的接口，而Service层负责将这两个接口组装起来。这样就把原先散布在表现层的业务逻辑都统一到了Service层，每一层的边界就非常清晰了。

除此之外，分层架构需要考虑的另一个因素，是层次之间一定是相邻层互相依赖，数据的流转也只能在相邻的两层之间流转。

我们还是以三层架构为例，数据从表示层进入之后一定要流转到逻辑层，做业务逻辑处理，然后流转到数据访问层来和数据库交互。那么你可能会问：“如果业务逻辑很简单的话可不可以从表示层直接到数据访问层，甚至直接读数据库呢？”

其实从功能上是可以的，但是从长远的架构设计考虑，这样会造成层级调用的混乱，比方说如果表示层或者业务层可以直接操作数据库，那么一旦数据库地址发生变更，你就需要在多个层次做更改，这样就失去了分层的意义，并且对于后面的维护或者重构都会是灾难性的。

分层架构的不足

任何事物都不可能是尽善尽美的，分层架构虽有优势也会有缺陷，它最主要的一个缺陷就是增加了代码的复杂度。

这是显而易见的嘛，明明可以在接收到请求后就可以直接查询数据库获得结果，却偏偏要在中间插入多个层次，并且有可能每个层次只是简单地做数据的传递。有时增加一个小小的需求也需要更改所有层次上的代码，看起来增加了开发的成本，并且从调试上来看也增加了复杂度，原本如果直接访问数据库我只需要调试一个方法，现在我却要调试多个层次的多个方法。

另外一个可能的缺陷是，如果我们把每个层次独立部署，层次间通过网络来交互，那么多层的架构在性能上会有损耗。这也是为什么服务化架构性能要比单体架构略差的原因，也就是所谓的“多一跳”问题。

那我们是否要选择分层的架构呢？**答案当然是肯定的。**

你要知道，任何的方案架构都是有优势有缺陷的，天地尚且不全何况我们的架构呢？分层架构固然会增加系统复杂度，也可能会有性能的损耗，但是相比于它能带给我们的好处来说，这些都是可以接受的，或者可以通过其它的方案解决的。**我们在做决策的时候切不可以偏概全，因噎废食。**

课程小结

今天我带着你了解了分层架构的优势和不足，以及我们在实际工作中如何来对架构做分层。我想让你了解的是，分层架构是软件设计思想的外在体现，是一种实现方式。我们熟知的一些软件设计原则都在分层架构中有所体现。

比方说，**单一职责原则**规定每个类只有单一的功能，在这里可以引申为每一层拥有单一职责，且层与层之间边界清晰；**迪米特法则**原意是一个对象应当对其它对象有尽可能少的了解，在分层架构的体现是数据的交互不能跨层，只能在相邻层之间进行；而**开闭原则**要求软件对扩展开放，对修改关闭。它的含义其实就是将抽象层和实现层分离，抽象层是对实现层共有特征的归纳总结，不可以修改，但是具体的实现是可以无限扩展，随意替换的。

掌握这些设计思想会自然而然地明白分层架构设计的妙处，同时也能帮助我们做出更好的设计方案。

思考时间

课程中我们提到了分层架构的多种模型，比如三层架构模型，阿里巴巴提出的分层架构模型，那么在你日常开发的过程中，会如何来做架构分层呢？你觉得如此分层的优势是什么呢？欢迎在留言区与我一同交流。

最后，感谢你的阅读，如果这篇文章让你有所收获，也欢迎你将它分享给更多的朋友。

精选留言：

- ly 2019-09-20 08:33:35
  目前有个业务（下单业务）**不知道这样的分层是否合理。
  OrderServiceFacade: 接受Controller的调用，具体是一些组织逻辑，组织数据等，然后调用OrderService的创建订单方法.
  OrderBillService.createBill(): 具有统一事务，具体把order入库，另外它依赖了库存服务(InventoryService),
  InventoryService.reduce(): 来减少库存

  (OrderBillService 依赖自己的OrderDao，但依赖了库存Service而不是库存DAO)

  // 订单门面
  public class OrderServiceFacade {
  @Autowrited
  private OrderBillService orderBillService;
  public boolean submitOrder(Dto dto) {
  // TODO 组织数据 一些特殊验证check
  // TODO 其他非事务性逻辑代码

  String billNo = orderBillService.createBill(dto);
  // TODO 后续工作
  return billNo != null;
  }
  }
  // 订单服务
  public class OrderBillService {
  @Autowrited
  private OrderDao orderDao;
  @Autowrited
  private InventoryService inventoryService;
  @Transaction

```
public String createBill(Dto dto) {
Bill bill = dto.convertToBill();
orderDao.insert(bill); // 保存订单
inventoryService.reduce(); // 扣减库存服务
return bill.getBillNo();
}
}
// 库存服务
public class InventoryService {
@Autowrited
private InventoryDao inventoryDao;
@Transaction
public void reduce() {
int updateCount = inventoryDao.decrease();
if( updateCount != 1 ) {
throw new BizException("扣减库存失败");
}
//TODO 记录库存日志
}
}
```

- 段启超 2019-09-20 08:28:51
  mvc这种结构让太多的人觉得项目工程结构理所应当就是这样的，然后呢，一大堆的业务逻辑就随意的堆砌在了service中，对象啥的，只是单纯的数据传输作用，出现了用面向对象的语言，写面向过程的程序的普遍现象。按照领域驱动设计的思路，最重要的还要有领域模型层。当然manage层这种方案也是一种思路，但是我觉得，这种方式，还不够，必须有清晰的业务模型和合理的分层结构配合，才能更好的提现分层的作用。

- _CountingStars 2019-09-20 07:46:30
  阿里这个分层和领域驱动设计里的分层有点类似

- AllenGFLiu 2019-09-20 07:23:15
  Python中web开发框架大而全的是Django，其遵循的也是MVC的变形体：MVT。

- 三年过后 2019-09-20 07:05:54
  阿里分层架构图中，上层依赖下层，箭头关系表示直接依赖。文中提到需求案例getUser细化，引申出阿里分层架构图，那么新增用户addUser业务逻辑是放在业务逻辑层（Service层），getUser就放在通用处理层（Manager层）吧？ 老师可以引入最近大家在讨论的中台，个人拙见。

  作者回复2019-09-20 07:26:41
  adduser和getUser都应该是在manager层，service来对这两个方法封装。中台其实是整合不同业务的通用逻辑的，让新业务接入时间更短，是更大范围的事情

- 高源 2019-09-20 06:08:11
  老师我问一下本节课提到Linux分层那块，用的是哪种设计模式或者借鉴的参考抽象出参考代码

  作者回复2019-09-20 07:27:12
  应该是策略模式

- 刘晖 2019-09-20 00:46:21

总觉得php框架大都是MVC。

实际工作中不同公司会加其他层。比如service。比如manager。

感觉php框架没有Java框架分层分的好。

作者回复2019-09-20 07:27:29
其实是不分语言的：）