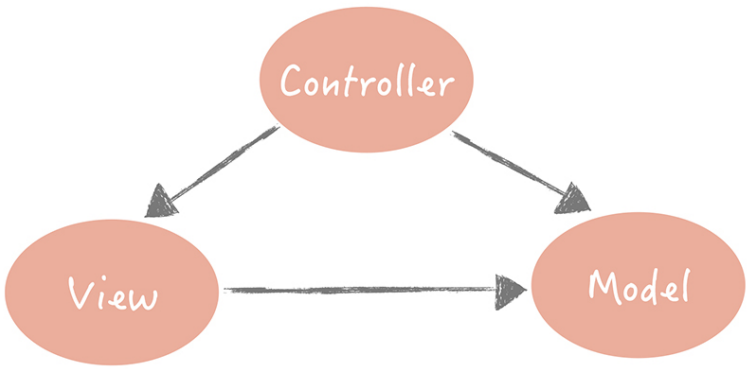
02 | 架构分层：我们为什么一定要这么做？

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。

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**：

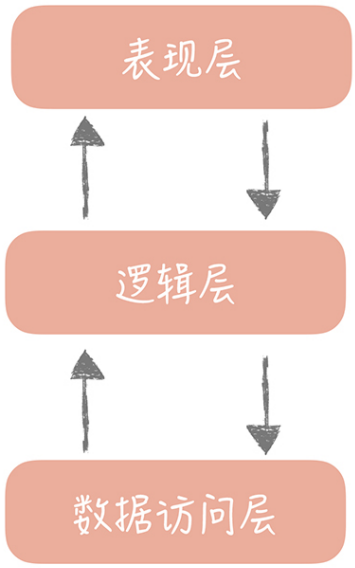
表现层，顾名思义嘛，就是展示数据结果和接受用户指令的，是最靠近用户的一层；

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

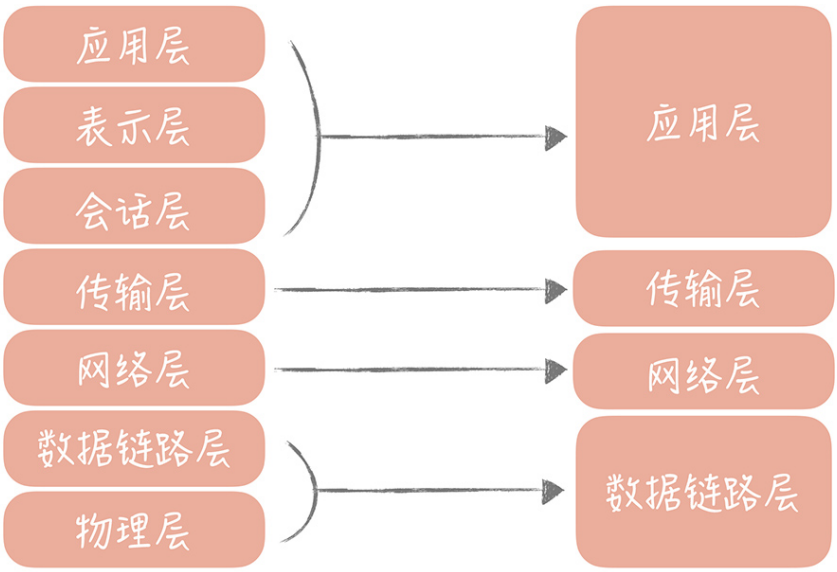
别对
应了表现层、逻辑层还有数据访问层。



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，它把整个网络分成了七层，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。

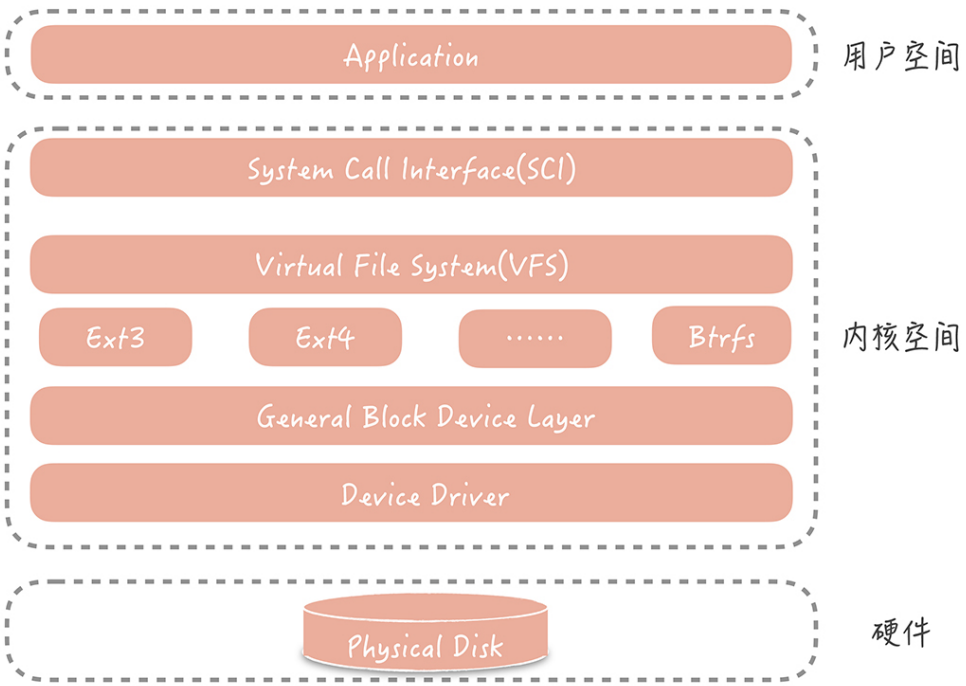
工作中经常能用到 TCP/IP 协议，它把网络简化成了四层，即链路层、网络层、传输层和应用层。每一层各司其职又互相帮助，网络层负责端到端的寻址和建立连接，传输层负责端到端的数据传输等，同时相邻两层还会有数据的交互。这样可以隔离关注点，让不同的层专注做不同的事情。



网络分层模型

nux 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，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得多了。

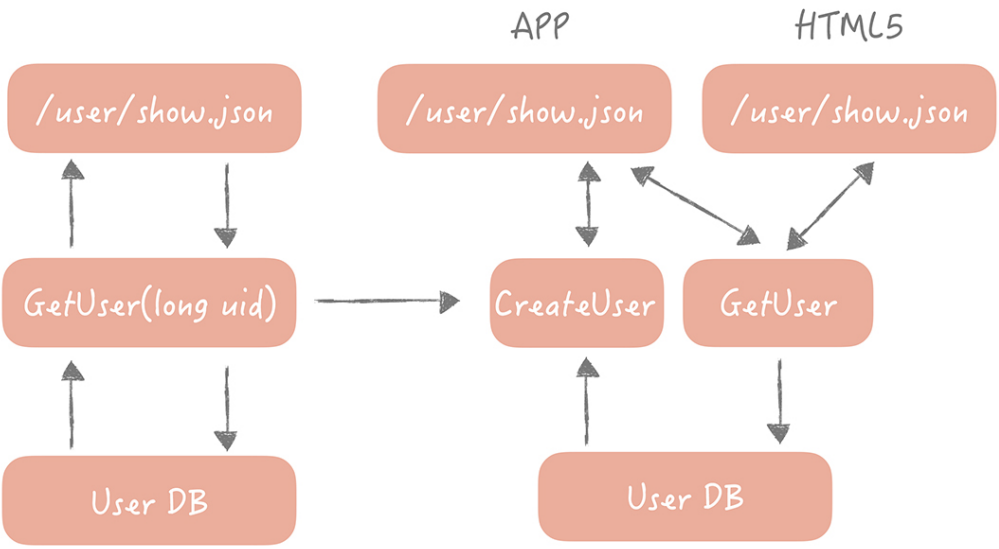
如何来做系统分层

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。

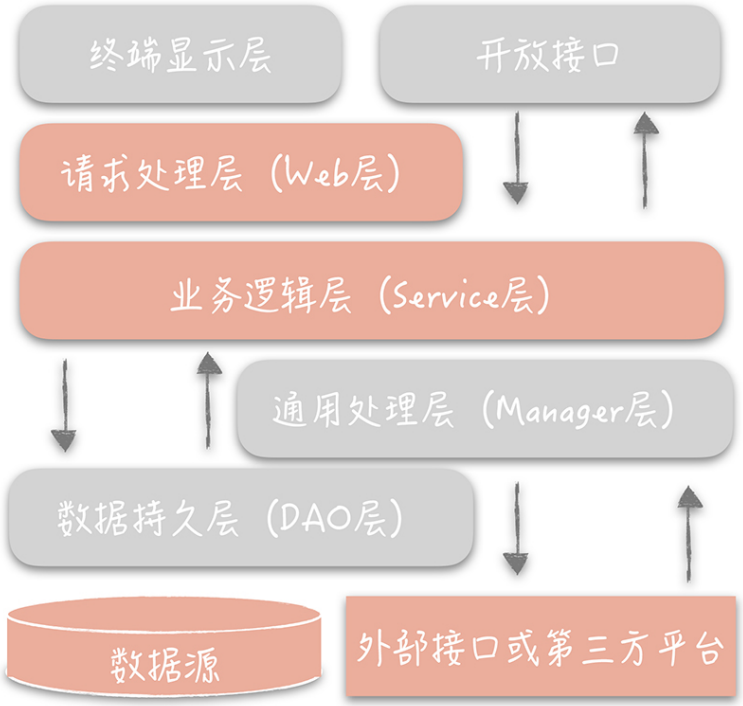
但是当业务逻辑变得越来越复杂时，边界就会变得越来越模糊，给你举个例子。任何一个系统中都有用户系统，最基本的接口是返回用户信息的接口，它调用逻辑层的 GetUser 方法，GetUser 方法又和 User DB 交互获取数据，就像下图左边展示的样子。这时，产品提出一个需求，在 APP 中展示用户信息的时候，如果用户不存在，那么要自动给用户创建一个用户。同时，要做一个 HTML5 的页面，HTML5 页面要保留之前的逻辑，也就是不需要创建用户。这时逻辑层的边界就变得不清晰，表现层也承担了一部分的业务逻辑（将获取用户和创建用户

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。

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。