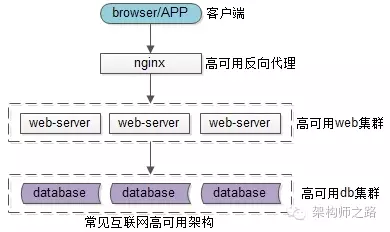
https://mp.weixin.qq.com/s/VCeR2TccicJ2OxREFl894w

# 互联网架构为什么要做服务化？

## ****一、互联网高可用架构，为什么要服务化？****

**【服务化之前高可用架构】**

在服务化之前，互联网的高可用架构大致是这样一个架构：

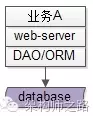


（1）用户端是浏览器browser，APP客户端

（2）后端入口是高可用的nginx集群，用于做反向代理

（3）中间核心是高可用的web-server集群，研发工程师主要编码工作就是在这一层

（4）后端存储是高可用的db集群，数据存储在这一层

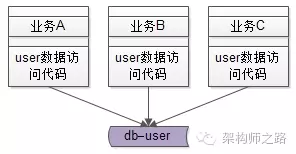


更典型的，web-server层是通过DAO/ORM等技术来访问数据库的。

可以看到，最初都是没有服务层的，此时架构会碰到一些什么痛点呢？

### 【架构痛点一：代码到处拷贝】

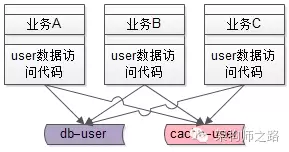
举一个最常见的业务的例子->用户数据的访问，绝大部分公司都有一个数据库存储用户数据，各个业务都有访问用户数据的需求：



在有用户服务之前，各个业务线都是自己通过DAO写SQL访问user库来存取用户数据，这无形中就导致了代码的拷贝。

### 【架构痛点二：复杂性扩散】

随着并发量的越来越高，用户数据的访问数据库成了瓶颈，需要加入缓存来降低数据库的读压力，于是架构中引入了缓存，由于没有统一的服务层，各个业务线都需要关注缓存的引入导致的复杂性：



对于用户数据的写请求，所有业务线都要升级代码：

（1）先淘汰cache

（2）再写数据

对于用户数据的读请求，所有业务线也都要升级代码：

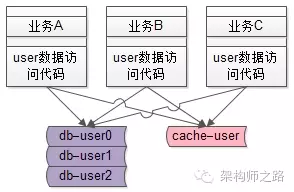
（1）先读cache，命中则返回

（2）没命中则读数据库

（3）再把数据放入cache

这个复杂性是典型的“业务无关”的复杂性，业务方需要被迫升级。

随着数据量的越来越大，数据库需要进行水平拆分，于是架构中又引入了分库分表，由于没有统一的服务层，各个业务线都需要关注分库分表的引入导致的复杂性：



这个复杂性也是典型的“业务无关”的复杂性，业务方需要被迫升级。

包括bug的修改，发现一个bug，多个地方都需要修改。

### 【架构痛点三：库的复用与耦合】

服务化并不是唯一的解决上述两痛点的方法，抽象出统一的“库”是最先容易想到的解决：

（1）代码拷贝

（2）复杂性扩散

的方法。抽象出一个user.so，负责整个用户数据的存取，从而避免代码的拷贝。至于复杂性，也只有user.so这一个地方需要关注了。

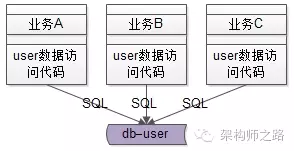
解决了旧的问题，会引入新的问题，库的版本维护与业务线之间代码的耦合：

业务线A将user.so由版本1升级至版本2，如果不兼容业务线B的代码，会导致B业务出现问题；

业务线A如果通知了业务线B升级，则是的业务线B会无故做一些“自身业务无关”的升级，非常郁闷。当然，如果各个业务线都是拷贝了一份代码则不存在这个问题。

### 【架构痛点四：SQL质量得不到保障，业务相互影响】

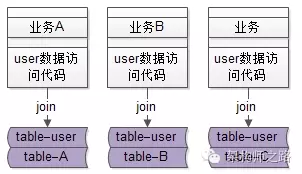
业务线通过DAO访问数据库：



本质上SQL语句还是各个业务线拼装的，资深的工程师写出高质量的SQL没啥问题，经验没有这么丰富的工程师可能会写出一些低效的SQL，假如业务线A写了一个全表扫描的SQL，导致数据库的CPU100%，影响的不只是一个业务线，而是所有的业务线都会受影响。

### 【架构痛点五：疯狂的DB耦合】

业务线不仅需要访问user数据，还会结合自己的业务访问自己的数据：



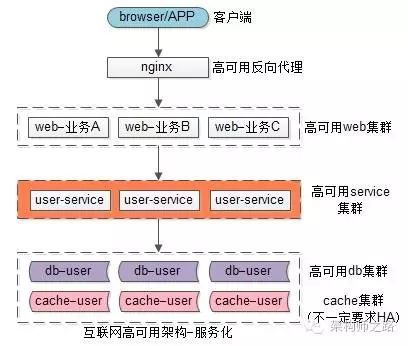
典型的，通过join数据表来实现各自业务线的一些业务逻辑。

这样的话，业务线A的table-user与table-A耦合在了一起，业务线B的table-user与table-B耦合在了一起，业务线C的table-user与table-C耦合在了一起，结果就是：table-user，table-A，table-B，table-C都耦合在了一起。

随着数据量的越来越大，业务线ABC的数据库是无法垂直拆分开的，必须使用一个大库（疯了，一个大库300多个业务表 =\_=）。

## 二、服务化解决什么问题？

为了解决上面的诸多问题，互联网高可用分层架构演进的过程中，引入了“服务层”。



以上文中的用户业务为例，引入了user-service，对业务线响应所用用户数据的存取。引入服务层有什么好处，解决什么问题呢？

### 【好处一：调用方爽】

有服务层之前：业务方访问用户数据，需要通过DAO拼装SQL访问

有服务层之后：业务方通过RPC访问用户数据，就像调用一个本地函数一样，非常之爽

User = UserService::GetUserById(uid);

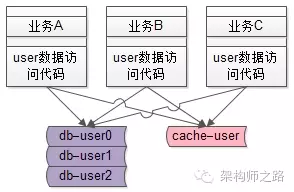
传入一个uid，得到一个User实体，就像调用本地函数一样，不需要关心序列化，网络传输，后端执行，网络传输，范序列化等复杂性。

### 【好处二：复用性，防止代码拷贝】

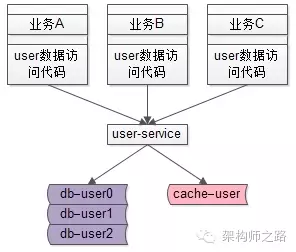
这个不展开叙述，所有user数据的存取，都通过user-service来进行，代码只此一份，不存在拷贝。

升级一处升级，bug修改一处修改。

### 【好处三：专注性，屏蔽底层复杂度】

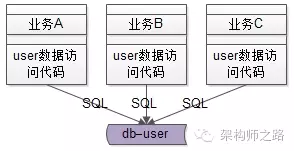


在没有服务层之前，所有业务线都需要关注缓存、分库分表这些细节。

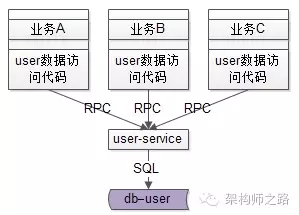


在有了服务层之后，只有服务层需要专注关注底层的复杂性了，向上游屏蔽了细节。

### 【好处四：SQL质量得到保障】

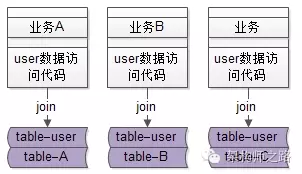


原来是业务向上游直接拼接SQL访问数据库。

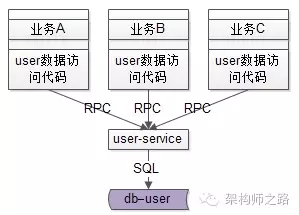


有了服务层之后，所有的SQL都是服务层提供的，业务线不能再为所欲为了。底层服务对于稳定性的要求更好的话，可以由更资深的工程师维护，而不是像原来SQL难以收口，难以控制。

### 【好处五：数据库解耦】



原来各个业务的数据库都混在一个大库里，相互join，难以拆分。



服务化之后，底层的数据库被隔离开了，可以很方便的拆分出来，进行扩容。

### 【好处六：提供有限接口，无限性能】

在服务化之前，各业务线上游想怎么操纵数据库都行，遇到了性能瓶颈，各业务线容易扯皮，相互推诿。

服务化之后，服务只提供有限的通用接口，理论上服务集群能够提供无限性能，性能出现瓶颈，服务层一处集中优化。