架构之路

罗马不是一天建成的，架构也不是一蹴而就的，需求-重构-上线不断的循环才有造就了架构之美或者架构之殇。

从事it开发工作已经8个年头了，参与10多个项目的开发，主导数个互联网项目的架构设计，主要是电商或者电商相关的项目，从开始的无从下手到现在的轻车熟路，过程磕磕绊绊，所幸都没有夭折，基本上顺利上线。读过一些架构相关的书籍，书中的架构的思路，实现过程和方式方法和自己架构设计的过程差别很大(我都在怀疑是否接触的层次太低了，哈哈),所以就想把自己的一些经验整理一下，整理总结思路，做为一种沉淀，另外一方面也可以和大家互相交流学习，知道自己的不足才能有更大的进步。

# 1、什么是架构设计[#](https://www.cnblogs.com/gdsblog/p/6637103.html#idx_0)

“架构设计是人们对一个结构内的元素及元素间关系的一种主观映射的产物。架构设计是一系列相关的抽象模式，用于指导大型软件系统各个方面的设计。” 来自百度百科。资料中的定义是准确、完备和书面化的，仍然很难理解架构设计的本质。通俗的描述，架构设计就像是小学考试中解答应用题的过程，但是解决的问题更复杂，构思设计的过程更庞大，解题的工作量更大。

# 2、项目的质量指标: 功能，性能和扩展性[#](https://www.cnblogs.com/gdsblog/p/6637103.html#idx_1)

软件开发的最终目标是使用代码去实现抽象的业务逻辑，可以从3个方面衡量：功能，性能和扩展性。

功能:功能目标是应用的基本要求，如果不能实现既定的功能逻辑，应用就失去了存在的意义，因此实现产品需求是应用的基本的目标。

性能:在基本的功能之上，会有一些性能的要求，但是很少有产品经理或者用户能提前提出这样的要求，因此架构师要有丰富的经验去发现和解决(或者为未来提升性能做准备)性能问题。性能的主要衡量有：单次请求的相应时间，单实例请求并发数，服务最大并发量等。

扩展性:目前互联网应用的开发模式:快速响应，迭代开发；提出需求，快速相应，尽快上线，是骡子是马拉出来溜溜。所以这就要求系统的架构设计要更好的响应新的需求和需求变更。

# 3、架构设计的主要过程[#](https://www.cnblogs.com/gdsblog/p/6637103.html#idx_2)

3年多来，数个项目的架构经验，我自己的架构设计过程是 : 确定问题域，数据建模，模块划分，关键流程描述，技术选型，代码实现，验收测试

## 3.1、确定问题域

记得小学考试后拿到老师改过的卷子，对着一个个的大红叉都会懊恼：”哎，又看错题目了“。错误的方向危害大于错误的方法，没有找对方向，项目就会南辕北辙，远远偏离目标。来自产品经理或者用户的需求描述就是我们的问题域，但是来自于产品经理的需求描述会比较全面，内容也很多，来自用户比较简单，相对比较模糊。比如电商项目的需求文档会非常大，拿到一个几十上百页的需求文档(有程序员拍砖说，我家的产品只一段话”像XXX网站的功能copy一下吧”，哈哈)时，往往不知道从何处下手，所以我们要从繁杂的问题域中找到关键问题。

a、用户可以在我们的网站上购买XXX商品。

从a条出发，又延伸出来几个问题:

b、用户访问:用户登录，注册

c、商品来源:商品的管理，增删改查等

d、交易的过程:订单的管理等

从d交易出发，又能延伸出来

e、用户付钱:支付

f、商家配送:收货地址，配送流程

不断的展开问题域，就可以把整个流程转起来。当然实际应用的时候我们不会把所有的问题域都总结出来，确定了关键问题就可以开始数据建模了。

上面我们分析的是功能问题域，这会也需要确定一下性能和扩展性的问题域。性能的问题域应该是针对关键路径确定的，比如商品浏览，订单创建，订单支付等。针对于这些关键路径问题，可以定义一些问题域，比如单实例支持1000pv/秒商品浏览，100单/秒的订单提交等。

扩展性是最难把握的，因为每个人经历不同，针对同样的项目会对未来需求有不同的预期，因此怎么把握当前的功能和未来的变化，如何平衡性能和扩展的关系，是架构师设计的关键。以我的的经验来看，扩展把我关键问题，优先满足关键问题的性能，确定最小功能集。确定最小功能集的优势可以快速实现，快速验证需求的准确性，每次需求开发都完成最小和最关键的需求。设计的时候要满足一些思想和原则，OOP(面向对象设计)原则:1、单一职责原则；2、开放闭合原则；3、里氏替换原则；4、依赖倒置原则；5、接口隔离原则;数据库设计三范式等等。扩展的问题域也可以参考友商或者与有经验的产品运营沟通，大致了解存在的扩展性。电商项目可能会有:抢购，预定，团购等业务都是电商的一些扩展需求。

## 3.2、数据建模

确定了问题域就可以开始答题了，确定数据模型。大部分的应用基本使用的仍然是关系型数据库，所以我们针对问题域先创建数据表，当然也存在一些项目使用NoSQL存储或者不持久化数据，这里确定的就是问题域的实体类。3.1中描述的问题域每个名称都是一个数据表(或者实体对象)，用户，商品，订单，支付流水，收货地址，配送单。

我比较喜欢使用powerdesigner做数据库模型，可以直观的看到表结构，方便修改，可以生成大部分DB的DDL SQL。为上面找出的名词(实体结构)创建表结构，然后根据产品需求文档一条一条的阅读判断，当前表结构是否可以满足需求，如果不能满足，在表中添加列或者添加新的表来满足此需求，不断的去丰富表结构直到完全满足需求。当然在建模的过程中也会调整原来的表结构，毕竟不断的增加需求，会引起数据模型的变化，所以最初建立的肯定不完整，不断调整直到满足所有需求。

数据建模时的几个心得:

a、数据表包含自增id,创建时间createTime,更新时间updateTime和版本号version

自增的id:主键，根据id查询或者更新时，速度毕竟快。

createTime和updateTime记录创建时间和最后的更新时间，排查问题的关键点

version：编辑时version++，是一个很方便的乐观锁，能比较大的提升数据库的性能

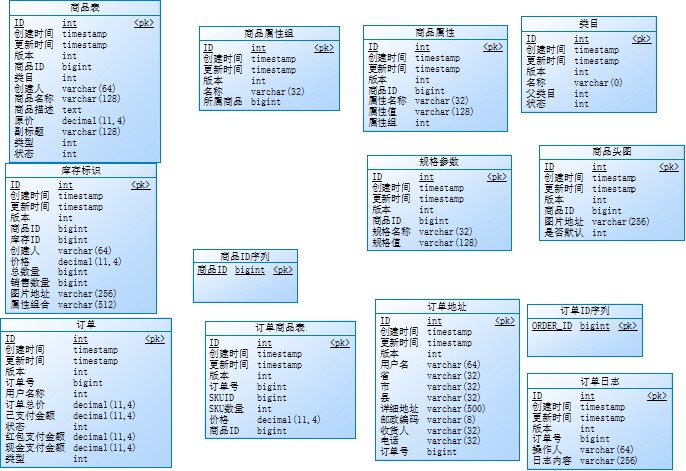
b、不使用外键，这点有一些和数据库设计的规范相悖，但是这是来自真实经验总结，外键约束带来的数据完整性的优势远远小于更新逻辑实现的难度。从性能和扩展性来看不使用外键也是利大于弊，大数据高并发大流量的互联网应用提供性能的常用方法是：提高数据库的访问速度，缓存数据，数据库分库分表支撑高并发等，外键是对这些方法的一个制约。

c、不使用id作为表关联，虽然我们不创建外键约束，但是不代表表之间没有关联关系。所以表之间仍然会有外键，但是没有外键约束，设计这个外键的时候要考虑数据的增长型，数据没有确定的规模，那么参考增长的速度，我们可以设定一个未来3-5年的数据规模，如果单表不能满足，则数据存在分表的可能性，那么表间的关联使用全局的唯一id的方式一个更好的选择。全局唯一id的方式有很多算法，使用数据库(oracle的sequence和mysql的自增id,这里是为生成id的特殊表的自增)生成是一个比较好的方式，当然也可以使用组合方式添加数据类型，时间，地域等方式，也有使用uuid算法计算的方式，只要可以满足不重复的特点，选取那种方式可以参考一下产品的意见，因为这个字段用户可能感知。

d、表有没有多少列的标准？记得刚开始做设计的时候，经常怀疑自己是不是分的表太多或者太少，太大了是不是会影响性能，太少了是不是有点画蛇添足。应用最初的设计最符合设计原则和设计思想的，没有收到工期，团队分割，实现难度等非设计因素的影响，所以我们应该尽量的坚持最初的设计，克服其他因素的影响。

e、冗余字段是否有必要？我的做法是不使用，保持原有的设计，如果系统真的流量比较大，查询性能太低，可以通过把读服务从业务系统中分离(这里要注意不是数据库的读写分离)。从业务上把读写分开，做一些便于查询和提高性能的设计，通过一些数据抽取方式同步数据。这里会有人提出异议，这样做会导致用户的读延迟，其实展示性数据对数据的延迟是有很大的容忍度的，只有业务系统需要做到数据的一致，那么业务系统对数据的读取是针对性的，很少会出现需要很多关联数据的情况，所以最初设计系统时尽量少的使用冗余字段去提供查询的方便性和性能。

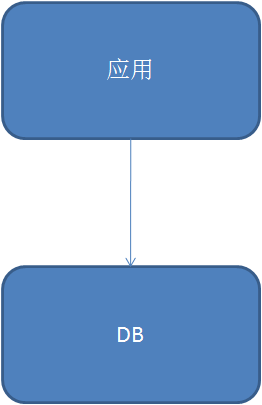
以下是电商应用部分表(商品和订单业务相关)设计，仅供参考:



## 3.3、模块划分

如果数据模型主要为满足功能目标的话，模块划分会比较多的兼顾性能和扩展性，常用的互联网应用的模块划分和部署结构有如下几种（这里讨论的划分和部署是互联网应用的服务器端），不考虑浏览器和APP等客户端，当然介绍的几种也是常用的结构。

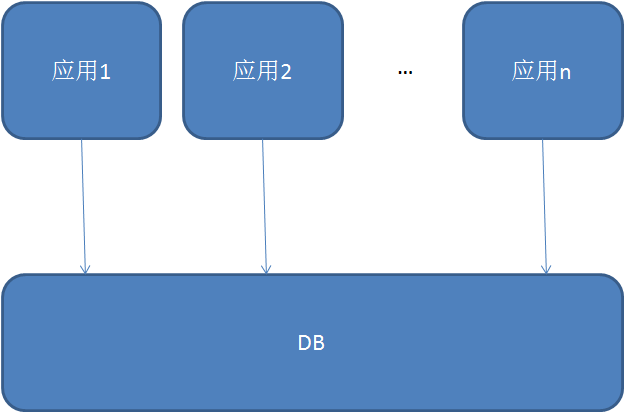
单实例结构:



优势: 结构简单，便于开发部署

劣势:可能存在性能瓶颈，扩展性差，系统耦合性高

集群结构：

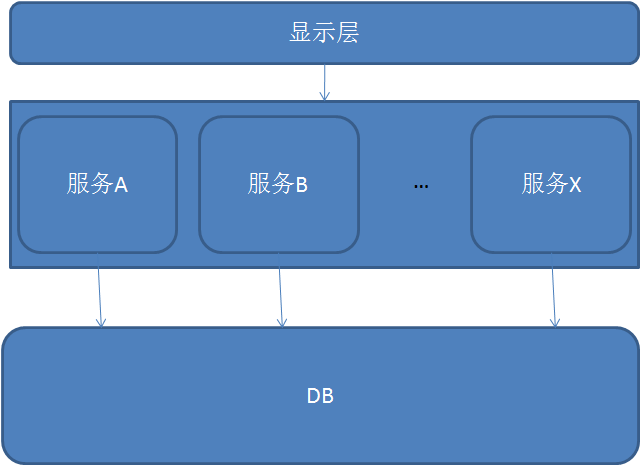


备注:db层作为整体描述，可能存在单DB，读写分离，分库分表或者数据cluster等技术

优势:性能大大提供(理论上可以无上限)

劣势:负载存在平衡的可能，仍然会存在性能瓶颈，扩展性差

分布式结构:

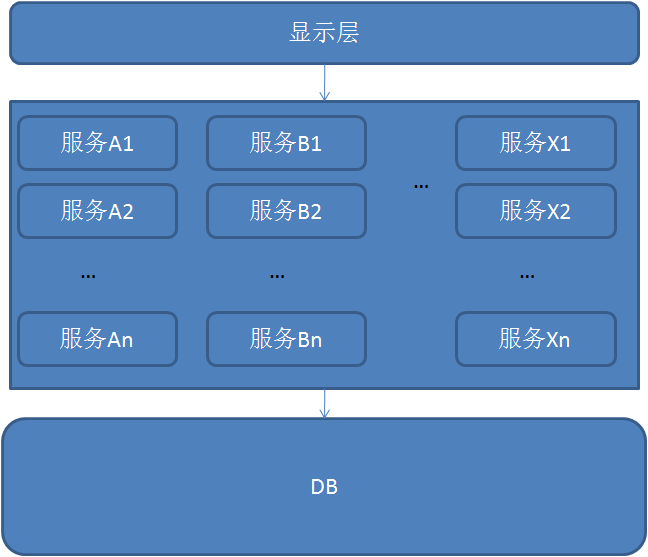


分布式系统是把不同的业务切分到不同的实例中，功能相关的聚合到同一个实例中，系统间使用网络协议通信的一种结构。

优势：扩展性强，高内聚，低耦合

劣势：结构复杂， 事务控制难度大，开发工作量大

混合结构：



混合结构是分布结构基础之上每个模块又实现集群结构，所以这个模式放大了分布式结构的优势和劣势

优势: 扩展性强，高内聚，低耦合,健壮性高

劣势: 结构复杂， 事务控制难度大，开发工作量大

大部分的互联网应用的结构都可以用上面的4个结构描述，当然这里只是简单的描述，一些应用为了提高数据库访问会加上DB缓存，为了提高页面的访问速度做页面静态化和CDN，为了应对大数据的存储和检索使用NoSQL数据库等，但是我们设计的结构是不受影响的。如何选择系统结构，可以从如下几方面考虑：

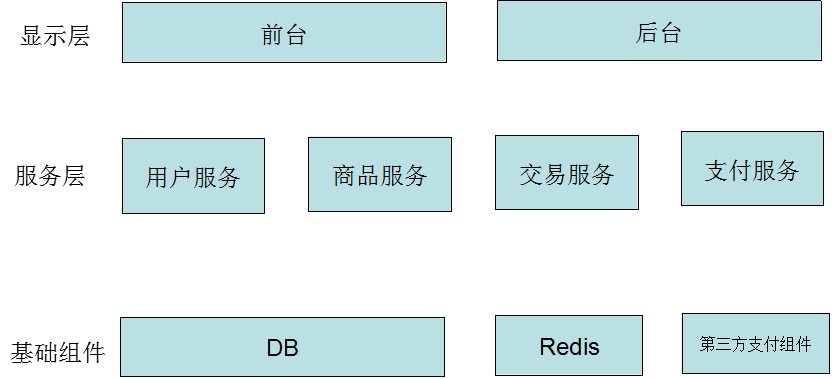
1、系统是否能满足未来2-3年的增长，如果采用混合结构的系统工作量要远远大于单例结构的系统，对初创企业来说，上线才是最大的需求，所以拼速度的时候就要放弃优雅。如果公司有一定的规模，开发的应用是核心业务或者未来的核心业务，采用扩展性强的混合结构应对未来的快速发展的业务需求是一个更好的选择。

2、人力因素，混合结构要比单实例的结构工作量增加很多，并且对团队整体的技术水平有较高的要求，所以要”量力而行”。

3、时间因素，很多互联网公司的工期不是技术评估的，是由”市场”确定的，所以火烧眉毛的时候就别讲究性能和扩展了，上线再说。

4、架构不是一成不变的，不断增加的访问和不断变化的需求改变着系统的架构。快速响应，不断迭代才是互联网应用的方式。所以最初的架构，尽量的做到高内聚低耦合，这样不断的提高系统的短板，逐渐完善系统结构。

分布式结构电商模块描述:



显示层:

前台：用户端界面显示层，依赖用户服务，商品服务，交易服务和支付服务

后台：运营端界面显示层，运营人员管理各种数据的界面。 依赖用户服务，商品服务，交易服务和支付服务；

服务层：为界面提供RPC服务

用户服务:注册，登录，用户管理等

商品服务:商品浏览，库存展示，商品管理，库存管理等

交易服务:购物车服务，订单计算， 订单提交，订单列表等

支付服务:生成支付链接，支付成功跳转，支付成功逻辑处理等

基础组件:

DB:数据存储

Redis:使用redis实现用户状态session机制，便于将来集群部署；实现购物车功能，用户购物车服务端持久化，便于用户跨浏览器购物车管理。

第三方支付组件:用于与第三方支付服务交互

3.4、关键流程描述

关键流程描述是检查系统架构是否满足需求和指导开发的必要条件。关键流程描述是使用流程图解决关键问题的过程，它的使用者是团队其他成员和自己，所以格式不重要，其他人能明白就好。

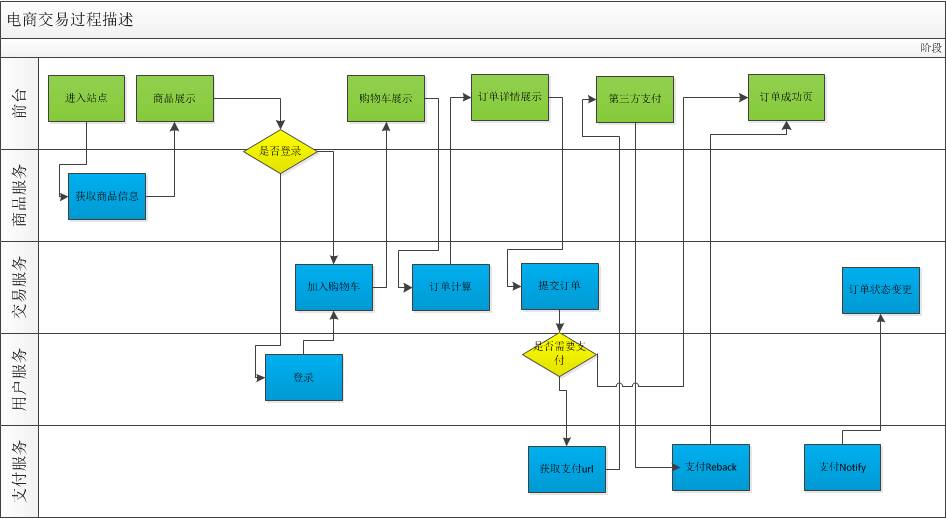
一些书写的经验如下:

1、有始有终，流程应该是从用户进入应用开始到离开应用的完整过程，比如交易的过程，应该从用户开始浏览商品到用户支付成功这一个过程。

2、流程图突出重点，比如上面举例的交易过程，应该突出交易相关的流程判断，不必描述用户注册，找回密码等过程。

3、简要说明，避免过于详细，比如交易的过程中需要更新库存，但是不需要描述更新库存前的库存校验这些是提交订单的内部实现。

示例如下:



3.5、技术选型

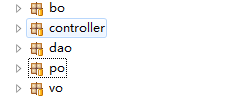
1、如果非必要请使用常用的技术，框架等，常用技术和框架使用者多，所以会比较少的遇到非业务问题。曾经参与的一个项目，其中一个模块由一个比较熟悉python的同学负责，系统刚刚上线，因为一些原因要离职，没有人可以接下来，只好找其他的语言重新开发了一遍。

2、熟悉的优于强大的，尽量采取团队比较熟悉的技术或者使用团队中有人可以指导的技术。记得5年前为甲方公司做一个需求和Bug跟踪的工作流的系统，轻率的决定使用JBPM，因为团队中没有人研究过，所以花了大量的时间使用这个框架，最后也没有很好的使用，导致项目步履蹒跚。

3.6、代码实现

代码首先是给人读的，其次才是给机器读，所以良好的代码结构是项目存活更长时间的良药。

1、代码分层：功能单一原则，mvc是互联网应用的一种基础模式，从功能层次上划分为，v显示层，c控制层，m业务层。以Java实现业务分层如下:



自上而下的层级,

controller:页面控制层，用于页面出参入参转换和页面跳转

vo:贫血实体对象，用于页面和业务层的数据传输

bo:业务实现层

dao:数据访问层，用于处理与数据库的交互

po:贫血实体对象,用于dao与数据库传输，和数据库表列意义对应

2、命名规范

代码中使用的类，方法，变量，参数等，采用统一风格命名，英文或者中文拼音，驼峰或者下划线分隔，尽量采用业界常用风格。类，变量和参数采用使用名词，方法使用动词等。

3、注释风格

采用统一的注释风格，方法内一般采用行注释，其他地方采用段注释。

3.7、验收测试

积极配合测试团队对项目的测试，他们是为项目健康上线保驾护航的人，不是挑刺的人。

## 4、总结[#](https://www.cnblogs.com/gdsblog/p/6637103.html#idx_3)

1、同样的题目有多种解法，我们做的只是其中的一种，所以要接受别人的质疑和建议，这样才能使系统完善。

2、没有银弹，没有解决一切问题的方法，那么也不可能使用一种方法解决所有问题， 所以要根据需求，团队，时间等选择合适的方式方法。

文章来源：<https://www.cnblogs.com/gdsblog/p/6637103.html>