# day00架构设计

主要目的是为了**解决软件系统复杂度**带来的问题；

集群方案都会存在一个或多个任务分配器；

计算和存储是高性能和高可用的关键问题；

涉及到的开源技术：

淘宝的TDDL（存储高性能 -- 程序代码封装）

MySQL Router 和 Atlas（存储高性能 – 中间件封装）

# day01架构基础

## 知识点1 架构到底是什么

弄清架构到底是什么，关键在于梳理几个有关系而又相似的概念：系统与子系统、模块与组件、框架与架构；

### 知识点1.1 系统与子系统

子系统的定义和系统定义是一样的，只是观察的角度有差异，一个系统可能是另外一个更大系统的子系统。

#### 系统

**系统**泛指由一群有关联的个体组成，根据某种规则运作，能完成个别元件不能单独完成的工作的群体。它的意思是“总体” “整体” 或 “联盟”；

**Ps**:规则规定了系统内个体分工和协作的方式。

#### 子系统

**子系统**也是由一群有关联的个体所组成的系统，多半会是更大系统中的一部分；

#### 注意事项

一个系统的架构，只包括顶层这一个层级的架构，而不包括下属子系统层级的架构。（例如：微信架构，就是指微信系统这个层级的架构。当然，微信的子系统，比如支付系统，也有它自己的架构，同样只包括顶层）

### 知识点1.2 模块与组件

模块和组件都是系统的组成部分，只是从不同的角度拆分系统而已。

从业务逻辑的角度来拆分系统后，得到的单元就是“模块”；从物理部署的角度来拆分系统后，得到的单元就是“组件”。划分模块的主要目的是职责分离；划分组件的主要目的是单元复用。

**Ps：**“组件”的英文 Component 也可翻译成中文的“零件”；

#### 模块

**软件模块**（Module）是一套一致而互相有紧密关连的软件组织。它分别包含了**程序和数据结构**两部分。现代软件开发往往利用模块作为合成的单位。模块的接口表达了由该模块提供的功能和调用它时所需的元素。模块是可能分开被编写的单位。这使它们可再用和允许人员同时协作、编写及研究不同的模块。

#### 组件

**软件组件**定义为自包含的、可编程的、**可重用的**、**与语言无关**的软件单元，软件组件可以很容易被用于组装应用程序中。

#### 注意事项

**1.**学生信息管理系统，这个系统从逻辑的角度来拆分，可以分为“登录注册模块”“个人信息模块”和“个人成绩模块”；从物理的角度来拆分，可以拆分为 Nginx、Web 服务器和 MySQL；

**2.**首先需要思考怎么从**业务逻辑**的角度把系统拆分成一个个**模块角色**，其次需要思考怎么从**物理部署**的角度把系统拆分成**组件角色;（组件拆分的角度比模块拆分的角度高，所以模块往往是组件的一部分）**

### 知识点1.3 框架与架构

#### 框架

**软件框架**（Software framework）通常指的是为了实现某个业界标准或完成特定基本任务的软件组件规范，也指为了实现某个软件组件规范时，提供规范所要求之基础功能的软件产品。

**Ps:**

1. 框架是组件规范；（MVC最常见的开发规范，类似的还有MVP、MVVM、J2EE等）
2. 框架提供基础功能的产品；（Spring MVC :不仅满足MVC规范、还提供了很多基础功能帮助我们实现功能；）

#### 架构

**软件架构**指软件系统的“基础结构”，创造这些基础结构的准则，以及对这些结构的描述。

#### 注意事项

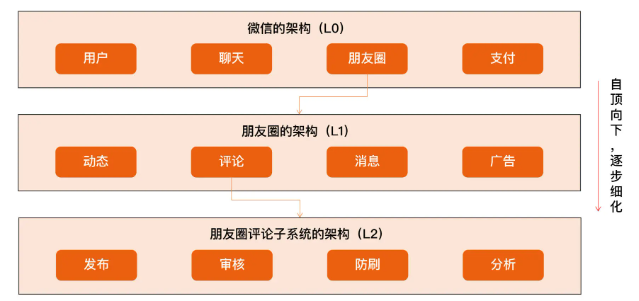
**1.框架关注的是“规范”，架构关注的是“结构”；**

**2.框架是一整套开发规范，架构是某一套开发规范下的具体落地方案，包括各个模块之间的组合关系以及它们协同起来完成功能的运作规则**

### 知识点1.4 重新定义架构（4R架构）

软件架构指软件系统的**顶层**（Rank）结构，它定义了系统由哪些**角色**（Role 个体）组成，角色之间的**关系**（Relation）和运作**规则**（Rule）

第一个R，Rank:是指软件架构是分层的，对应“系统”和“子系统”的分层关系，，避免将系统架构和子系统架构混淆导致架构层次混乱。通常情况下，我们只需要关注某一层的架构，最多展示相邻两层的架构，而不需要把每一层的架构全部糅杂在一起；



## 知识点2：架构设计的目的

### 知识点2.1 架构设计的误区

* 1. 架构很重要，所以要做架构设计
  2. 不是每个系统都要做架构设计吗
  3. 公司流程要求系统开发过程中必须有架构设计
  4. 为了高性能、高可用、可扩展，所以要做架构设计

### 知识点2.2以史为鉴

* 1. 1940年之前，机器语言；
  2. 20世纪40年代（1940年~1949年），汇编语言；
  3. 20世纪50年代，高级语言（Fortran语言，LISP语言，Cobol语言）；
  4. 20世纪60年代~20世纪70年代，第一次软件危机与结构化程序设计（Pascal语言、C语言）；

结构化程序设计：抛弃goto,采取“自顶向下、逐步细化、模块化”；

软件危机与软件工程概念由此提出；

* 1. 20世纪80年代，第二次软件危机与面向对象（C++,JAVA,C#）；
  2. 20世纪90年代，软件架构

Ps:

结构化编程，创造了”模块”的概念；

面向对象编程，创造了”对象”的概念；

软件架构，创造了“组件”的概念；

### 知识点2.3架构设计的真正目的

整个软件技术发展的历史，其实就是一部与“复杂度”斗争的历史，架构的出现也不例外。简而言之，架构也是为了应对软件系统复杂度而提出的一个解决方案

**架构设计的准则：架构设计的主要目的是为了解决软件系统复杂度带来的问题；**

遵循这条准则能够让**“新手”**架构师：心中有数，而不是一头雾水。

遵循这条准则能够让**“老鸟”**架构师：有的放矢，而不是贪大求全。

## 知识点3：架构复杂度的来源

关键：了解什么会带来软件的 “复杂度”；

复杂度来源：

1.性能 2.可用性

3.可扩展 4.安全

5.成本 6.规模

### 知识点3.1 高性能

软件系统的高性能带来的复杂度主要体现在两方面：

1. 单台计算机内部为了高性能带来的复杂度；
2. 多台计算机集群为了高性能带来的复杂度；

#### 知识点3.1.1 单机复杂度

##### 复杂度识别

计算机内部最关键的复杂度：操作系统；

操作系统和**性能最相关**的两个关键点：进程和线程。

##### 操作系统发展史

1. 最开始的操作系统：最开始是没有操作系统的，只有输入、计算和输出功能；
2. 批处理操作系统：

优点：

为了解决手工操作带来的低效，先把要执行的指令预先写下来（写到纸带、磁带、磁盘等），形成一个指令清单（常说的“任务”）；

缺点：

计算机一次只能执行一个任务;

1. 进程：进一步提升性能；

优点：

用进程来对应一个任务，每个任务都有自己独立的内存空间，进程间互不相关，由操作系统来进行调度。此时的 CPU 还没有多核和多线程的概念，为了达到多进程并行运行的目的，采取了分时的方式，即把 CPU 的时间分成很多片段，每个片段只能执行某个进程中的指令。从CPU角度：串行处理，从用户角度：多进程在并行处理（CPU运算速度很快）；

为了解决进程之间通信的问题,设计了管道、消息队列、信号量、共享存储；

缺点：

多进程虽然可以让多任务能够并行处理任务，但本身还有缺点单个进程内部只能串行处理，而实际上很多进程内部的子任务并不要求是严格按照时间顺序来执行的，也需要并行处理。

1. 线程

优点：

解决单个进程内部只能串行处理的问题；线程就是进程内部的子任务；但子任务都共享同一份进程数据。为了保证数据的正确性，又发明了互斥锁机制。

有了多线程后，**操作系统调度的最小单位**就变成了线程，**操作系统分配资源（内存空间）的最小单位**变成了进程；

缺点：

多进程多线程虽然让多任务并行处理的性能大大提升，但本质上还是分时系统，并不能做到时间上真正的并行；

1. 多核处理器

多个 CPU 能够同时执行计算任务，从而实现真正意义上的多任务并行；

目前的解决方案有 3 种：

* + - 1. SMP（Symmetric Multi-Processor，对称多处理器结构）；
      2. NUMA（Non-Uniform Memory Access，非一致存储访问结构）；
      3. MPP（Massive Parallel Processing，海量并行处理结构）；

其中最流行的多核处理器就是SMP方案；

##### 高性能软件系统

考虑如多进程、多线程、进程间通信、多线程并发等技术点，而且这些技术并不是**最新的就是最好的，也不是非此即彼的选择；**

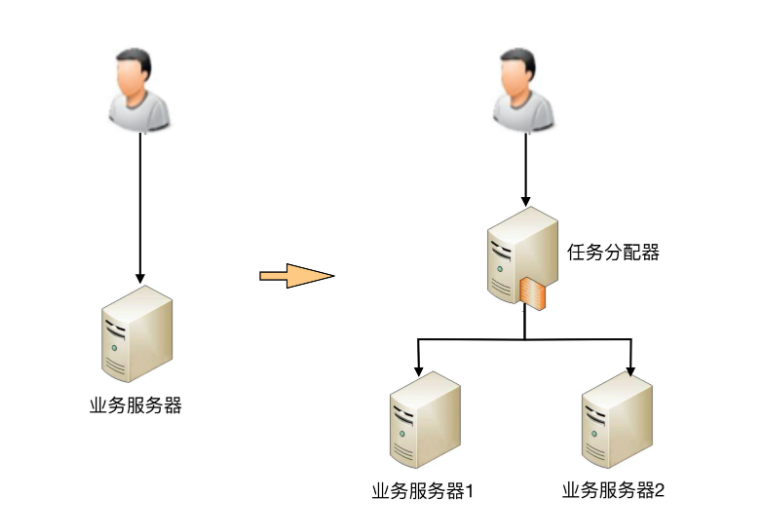
#### 知识点3.1.2多机（集群）复杂度

业务要求的性能，在高配的单机性能也不满足时，必须采用机器集群的方式来达到高性能；

如何处理多台机器配合起来达到高性能的目的：任务分配、任务分解；

##### 任务分配

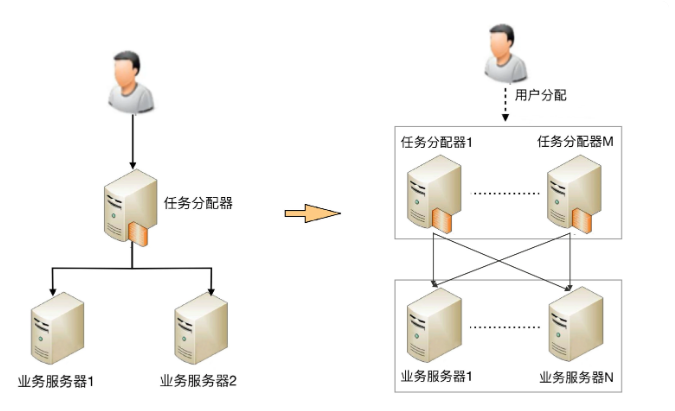
每台机器都可以处理完整的业务任务，不同的任务分配到不同的机器上执行；

****

从图中可以看到，1 台服务器演变为 2 台服务器后，架构上明显要复杂多了，主要体现在：

* 增加一个任务分配器：分配器可能是硬件网络设备（例如，F5、交换机等），软件网络设备（例如，LVS），负载均衡软件（例如，Nginx、HAProxy），自己开发的系统。选择合适的任务分配器也是一件复杂的事情，需要综合考虑性能、成本、可维护性、可用性等各方面的因素。
* 任务分配器和真正的业务服务器之间有连接和交互（即图中任务分配器到业务服务器的连接线），需要选择合适的连接方式，并且对连接进行管理。例如，连接建立、连接检测、连接中断后如何处理等。
* 增加分配算法。例如，是采用轮询算法，还是按权重分配，又或者按照负载进行分配。

当一台任务分配器不满足业务性能需求时，任务分配器本身也需要扩展为多台机器，这是架构复杂度更上一个层级：



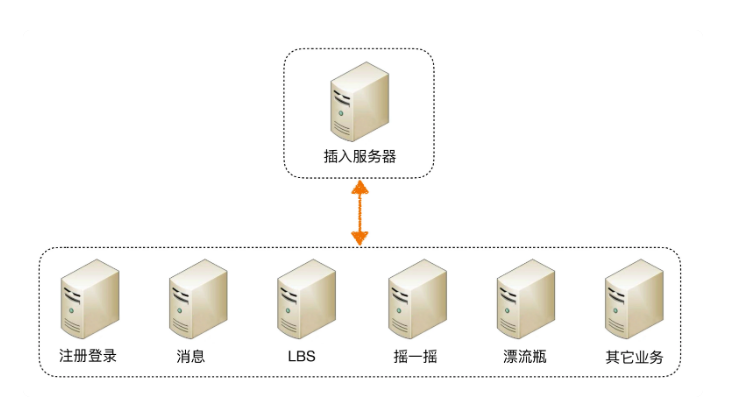
这个架构比 2 台业务服务器的架构要复杂，主要体现在：

* 任务分配器从 1 台变成了多台（对应图中的任务分配器 1 到任务分配器 M），这个变化带来的复杂度就是需要将不同的用户分配到不同的任务分配器上（即图中的虚线“用户分配”部分），常见的方法包括 **DNS 轮询、智能 DNS、CDN（Content Delivery Network，内容分发网络）、GSLB 设备（Global Server Load Balance，全局负载均衡）**等。
* 任务分配器和业务服务器的连接从简单的“1 对多”（1 台任务分配器连接多台业务服务器）变成了“多对多”（多台任务分配器连接多台业务服务器）的网状结构。
* 机器数量从 3 台扩展到 30 台（一般任务分配器数量比业务服务器要少，这里我们假设业务服务器为 25 台，任务分配器为 5 台），状态管理、故障处理复杂度也大大增加。

##### 任务分解（拆）

业务极其复杂时，单台机器处理该业务的性能越来越低；为了能够继续提升性能，此时就需要采取**任务分解**方式；注意：任务分解会增加任务量，从代码内部调用改为通过服务器之间的接口调用；

以微信后台架构为例：



**任务分解为何增加任务量，还能提高性能？**

* + - 简单的系统更加容易做到高性能；（关键点少）
    - 可以针对单个任务进行扩展；（性能瓶颈更容易发现）

##### 注意事项

1. **“任务分配器”**不一定是物理上存在的机器或者一个独立运行的程序，也可以是嵌入在其他程序中的算法;
2. **“任务分解”**并不是越细越好，对于架构设计来说，如何把握这个粒度就非常关键。

### 知识点3.2 高可用

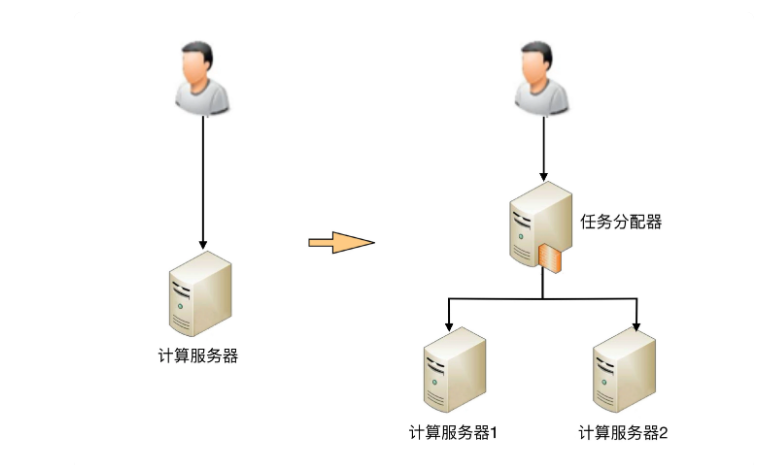
**高可用定义：**系统**无中断**地执行其功能的能力，代表系统的可用性程度，是进行系统设计时的准则之一。

系统高可用实现：本质上都是通过**“冗余”**来实现高可用的；

高可用的“冗余”方案，单从形式上来看，和之前讲的集群高性能是一样的，都是通过增加更多机器来达到目的；其本质上是有根本区别的：**高性能增加机器目的在于“扩展”处理性能；高可用增加机器目的在于“冗余”处理单元；（高性能扩展机器：执行不同的任务，高可用扩展机器：执行相同任务）；**

##### 计算高可用

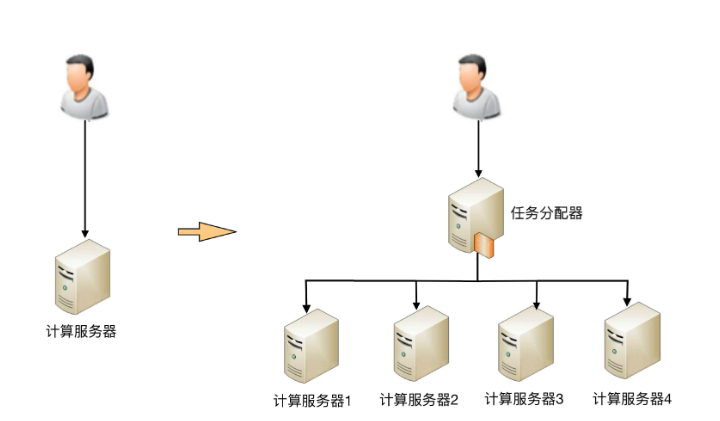
特点：无论在哪台机器上进行计算，同样的算法和输入数据，产出的结果都是一样的；



双机的架构图和“高性能”讲到的双机架构图是一样的，因此复杂度也是类似的，具体表现为：

* 需要增加一个任务分配器，选择合适的任务分配器也是一件复杂的事情，需要综合考虑性能、成本、可维护性、可用性等各方面因素。
* 任务分配器和真正的业务服务器之间有连接和交互，需要选择合适的连接方式，并且对连接进行管理。例如，连接建立、连接检测、连接中断后如何处理等。
* 任务分配器需要增加分配算法。例如，常见的双机算法有主备、主主，主备方案又可以细分为冷备、温备、热备。

复杂的高可用集群架构：



高可用集群相比双机来说，分配算法更加复杂，可以是 1 主 3 备、2 主 2 备、3 主 1 备、4 主 0 备，具体应该采用哪种方式，需要结合实际业务需求来分析和判断，并不存在某种算法就一定优于另外的算法。例如，ZooKeeper 采用的就是 1 主多备，而 Memcached 采用的就是全主 0 备

##### 存储高可用

存储与计算相比，有一个本质上的区别：将数据从一台机器搬到到另一台机器，需要经过线路进行传输；

存储高可用的**难点**不在于如何备份数据，而在于如何减少或者规避数据不一致对业务造成的影响；（**存储高可用还是需要任务分配器，分配读/写任务的**）

分布式领域里面有一个著名的 CAP 定理，从理论上论证了存储高可用的复杂度；存储高可用不可能同时满足“一致性、可用性、分区容错性”，最多满足其中两个，这就要求我们设计架构时结合业务进行取舍了；

##### 高可用状态决策

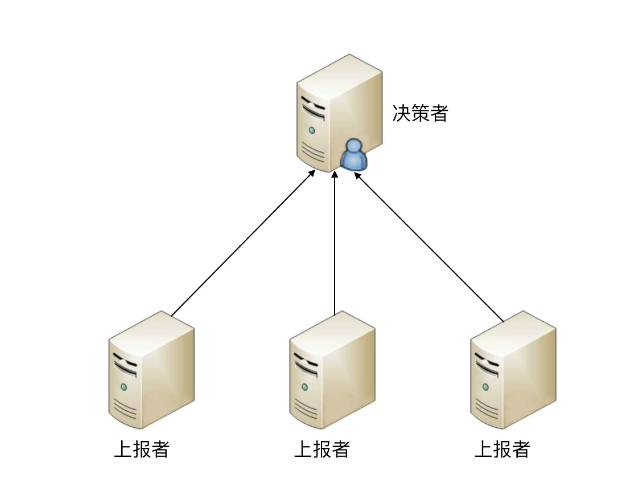
高可用的本质是“冗余”处理单元，如何判断当前处理单元的状态是正常还是异常，基础是”状态决策“；异常就要保证高可用了；如果状态决策本身都是有错误或者有偏差的，结果再完美也是毫无意义的；

这里就出现了一个本质上矛盾的地方：**通过冗余来实现的高可用系统，状态决策本质上就不可能做到完全正确；（只能尽量保证状态决策不出错，软件上不存在没有BUG的代码）**

高可用常见的几种方式：

* + 1. 独裁式

独裁式决策指的是存在一个独立的决策主体，我们姑且称它为“决策者”，负责收集信息然后进行决策；所有冗余的个体，我们姑且称它为“上报者”，都将状态信息发送给决策者；



优缺点：

优点：

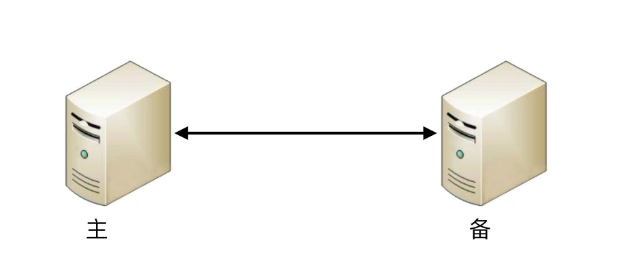
独裁式的决策方式不会出现决策混乱的问题，因为只有一个决策者

缺点：

当决策者本身故障时，整个系统就无法实现准确的状态决策。如果决策者本身又做一套状态决策，就陷入了递归是循环了；

* + 1. 协商式

协商式决策指的是**两个独立的个体通过交流信息**，**然后根据规则进行决策**；最常用的协商式决策就是主备决策；



该架构协商规则可以设计成：

* 2台服务器启动时都是备机。
* 2台服务器建立连接。
* 2台服务器交换状态信息。
* 某 1 台服务器做出决策，成为主机；另一台服务器继续保持备机身份

优缺点：

优点：

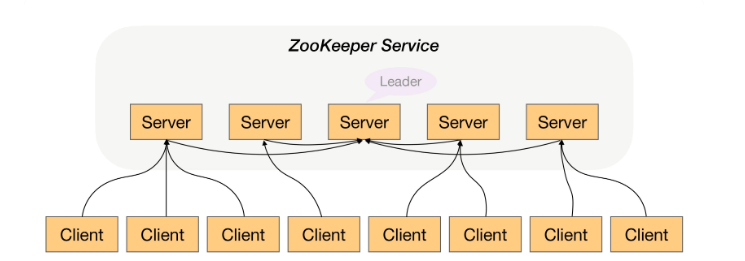
协商式决策的架构不复杂，规则也不复杂

缺点：

两者的信息交换出现问题（比如主备连接中断），此时状态决策应该怎么做。（三种分析”见书籍”，**最后的出的结论：协商式状态决策在某些场景总是存在一些问题的**）

* + 1. 民主式

民主式决策指的是**多个独立的个体通过投票的方式来**进行状态决策。以ZooKeeper选取Leader为典型;



优缺点：

优点：

民主式决策和协商式决策比较类似，其基础都是**独立的个体（Server）之间交换信息**，每个个体做出自己的决策，然后按照“多数取胜”的规则来确定最终的状态;

民主式决策比协商式决策要复杂得多，**ZooKeeper 的选举算法 ZAB;**

**缺点：**

算法复杂，民主式决策还有一个固有的缺陷：脑裂（见书籍）；

脑裂的根本原因：原来统一的集群因为连接中断，造成了两个独立分隔的子集群，每个子集群单独进行选举；

脑裂解决方案：一般都采用“投票节点数必须超过系统总节点数一半”规则来处理；这种方案降低了系统的可用性，即如果系统不是因为脑裂问题导致投票节点数过少，而真的是因为节点故障，导致了投票节点数没有超过总节点数的一半，此时也不会选出主节点，也就相当于“宕机”了；

### 知识点3.3 可扩展性

可扩展性，系统为了应对将来需求的变化而提供的一种扩展能力；

##### 预测变化

**一定要切记：”唯一不变的是变化“；**

预测变化的复杂度（靠架构师的经验和直觉）：

* 不能每个设计点都考虑可扩展性；
* 不能完全不考虑扩展性；
* 所有预测都存在出错的可能；

##### 应对变化

两种方式：

* + 1. 变化层与稳定层；

将“变化”的封装在一个“变化层”，将不变的封装在一个”稳定层“；

**两种方式：**

* + - * 1. **变化层依赖稳定层；**
        2. **稳定层依赖变化层；**

无论采取哪种方式，都会带来以下两个主要的复杂度问题:

* 1. 系统需要拆分出变化层与稳定层；
  2. 需要设计变化层与稳定层接口；
     1. 抽象层与实现层（见“设计模式”，“规则引擎”）

抽象层是稳定的，实现层根据业务需要定制开发；

### 知识点3.4 低成本

来控制项目投入的成本；（可以忽略）

### 知识点3.5 安全

安全本身就是一个庞大而复杂的技术领域；

安全分两类：

功能安全：在攻防大战中逐步完善（防小偷）

常见的有：XSS攻击、CSRF攻击 、SQL注入 、Windows漏洞 、 密码破解；

本质上都是系统实现有漏洞，和具体的编码有关；

架构安全：

防火墙（传统企业：银行等）：隔离网络；

**互联网行业：**具有海量用户、高并发特点，防火墙不足以支持；现在更多依靠的是运营商与云服务商强大的带宽和流量清洗功能

### 知识点3.6 规模

量变引起质变；

* 功能越来越多，导致系统的复杂度指数级上升；

系统的复杂度 = 功能数量 + 功能之间的链接数量；

* 数据越来越多，系统复杂度发生质变（大数据）

数据量太大，传统的数据收集、加工、存储、分析的手段和工具已无法适应；

# day02架构设计原则

架构设计遵循的是相对设计原则，多个方案不知如何选择时，我们就按照设计原则来筛选；

1. 合适的架构优于业界领先的架构；
2. 简单的架构优于复杂的架构；
3. 演化优于一步到位；

## 知识点1 合适原则

将军难打无兵之仗

没那么多人，却想干那么多活，是失败的第一个主要原因;

* 1. 罗马不是一天建成的

没那么多积累，却想一步登天，是失败的第二个主要原因;

* 1. 冰山下面才是关键

没那么多卓越的业务场景，却幻想灵光一闪成为天才，是失败的第三个主要原因;

**真正的优秀的架构，是在企业当前人力，条件，业务等各种约束下设计出来的;**

## 知识点2 简单原则

“复杂“:在制造领域代表先进，在建筑领域代表领先；但在软件领域，却恰恰相反，代表的是“问题”；

### 知识点2.1 结构的复杂性

结构复杂性特点：

1. 组成系统的组件数量更多；
2. 组件之间的关系更复杂；

结构复杂存在以下问题：

1. 组件越多，就越有可能其中某个组件出现故障，从而导致系统故障；
2. 某个组件改动，会影响关联的所有组件，这些被影响的组件同样会继续递归影响更多的组件；
3. 定义一个复杂系统中的问题总比是比简单系统更加困难；

### 知识点2.2 逻辑的复杂性

看到“结构复杂”后，第一反应降低组件数量，最简单的结构当然就是整个系统只有一个组件（即系统本身，所有功能和逻辑都在这一个组件中实现）；

逻辑复杂性特点：

单个组件承担了太多功能；

## 知识点3 演化原则

对于“建筑”来说，永恒才是主题；而对于软件来说，变化才是主题！

首先，设计出的架构要满足当时的业务需求；

其次，架构要不断地在实际应用过程中迭代，保留优秀设计，修复有缺陷的设计，改正错误的设计，去掉无用的设计，使得架构逐渐完善；

最后，当业务发生变化时，架构要扩展、重构、甚至重写；代码也许会重写，但有价值的经验、教训、逻辑、设计等却可以在新架构中延续；

架构师进行架构设计时，一定要谨记“演化原则”，时刻提醒自己不要贪大求全，盲目照搬大公司的做法，而应该认真分析当前业务特点，明确业务面临的主要问题，设计合理的架构，快速落地满足业务需要，然后在运行的过程中不断完善架构，不断随着业务演化架构；

# day03架构设计流程

## 知识点1：有的放失—识别复杂度

架构设计的本质目的是为了解决软件系统的复杂性；

架构的复杂度主要来源于“高性能”，“高可用”，“高扩展”等几个方面，但架构师不能认为任何时候从这三个方面进行复杂度分析就可以了，实际上绝大部分场景下，复杂度只是其中一个，少数情况包含其中两个，如果真出现需要解决三个或三个以上复杂度问题，不是这个系统做的太烂，就是架构师判断出现严重错误；

接手一个每个复杂度都存在问题的系统，那应该怎么办呢？

答：一个个来解决问题，不要幻想一次架构重构解决所有问题；将主要的复杂度问题列出来，然后根据业务、技术、团队等综合情况进行排序，优先解决当前面临的最主要的复杂度问题；

对于按照复杂度优先级解决的这种方式往往有一个普遍的担忧：如果按照优先级来解决复杂度，可能会出现解决了优先级排在前面的复杂度后，解决后续复杂度的方案需要将已经落地的方案推到重来？

答：这个担忧理论上是可能的，但现实中几乎不可能出现；原因在于软件系统的可塑性和易变性；对于同一个复杂度问题，软件系统的方案可以有多个，总是可以挑出综合来看性价比最高的方案；即使推到重来，新方案也必须能够同时解决已经被解决的复杂度问题，一般来说能够达到这种理想状态的方案，基本上都是靠引进新技术；

## 知识点2：按图索骥—设计备选方案

已经被各种场景验证过的成熟技术：高可用的主备方案、集群方案、高性能的负载均衡、多路复用、可扩展分层、插件化（微内核）等技术；绝大部分有了明确目标，按图索骥就能够找到可选的解决方案；

只有当这种方式完全无法满足需求时，才会考虑创新，而事实上方案的创新绝大部分情况下也都是基于已有的成熟技术。

技术的本质一书中，对技术的组合有清晰的阐述：

新技术都是在现有技术的基础上发展起来的，现有技术又来源于先前的技术。将技术进行功能性分组，可以大大简化设计过程，这是技术“模块化”的首要原因；技术的“组合”和“递归”特征，将彻底改变我们对技术本质的认识；

设计备选方案常见的错误：

* + 1. 设计最优秀的方案;
    2. 只做一个方案；
    3. 备选方案过于详细；

常见错误的解决方式：

a解决方案：遵循“简单原则”；

b解决方案：备选方案数量3~5个为最佳；备选方案差异比较明显；备选方案的技术不要只局限于已经熟悉的技术；

c解决方案：备选阶段关注的是技术选型，而不是技术细节，技术选型差异要比较明显。

## 知识点3：深思熟虑—评估和选择备选方案

挑选方案的指导思想：

* + 1. 最简派
    2. 最牛派
    3. 最熟派
    4. 领导派

如何确定选择那种派系，需要根据业务场景来确定；

解决方案：“360度环评”；

具体方式：

1. 列出我们需要关注的质量属性点；
2. 分别从这些“质量属性点”的维度去评估方案；
3. 挑选合适当时情况的最优方案；

常见的质量属性点有：性能，可用性，硬件成本，项目投入，复杂度，安全性，可扩展性；评估这些质量属性时，需要遵循架构设计原则：合适、简单、演化三大原则；如果某个质量属性点评估和业务发展有关系，需要根据未来业务发展规模时，一般是将当前业务规模\*2~4即可；基数低\*4，基数高\*2；

**360度环评具体做法：见书籍54页；**

## 知识点4：精雕细琢—详细设计方案

设计详细方案时，存在一种极端情况就是在详细设计阶段发现备选方案不可行；一般碰到该情况，主要原因就是备选方案设计时，遗漏了某个关键技术点或关键的质量属性。

如何避免上述问题？

解决方案：

* + - 1. 架构师不但要进行备选方案设计和选型，还需要对备选方案的关键细节有较深入的了解；不能成为“PPT架构师”
      2. 通过“分步骤”、“分阶段”、“分系统”等方式，尽量降低方案复杂度；
      3. 方案本身就很复杂，就采取设计团队的方式进行设计，博彩众长；

# 了解架构模式之前，需要了解一下概念：



# day04高性能架构模式

## 知识点1：存储高性能

### 知识点1.1关系数据库

关系数据库，由于其ACID的特性和功能强大的SQL查询，目前还是各种业务系统中关键和核心的存储系统；

**现如今，为了满足互联网业务的海量用户及海量数据的特点，单个数据库服务器已经难以满足业务需要，必须考虑数据库集群的方式来提升性能；**

两种方式:

* 读写分离：是将访问压力分散到集群中的多个节点，但是没有分散存储压力；
* 分库分表：既可以分散访问压力，又可以分散存储压力；

Ps:业务服务器操作数据库之前有一个 数据库访问层（任务分配器）

#### 读写分离

读写分离的基本原理是将数据库读写操作的压力分散到不同的节点上；

架构图：见书籍63页

读写分离的基本实现如下：

* 1. 数据库服务器搭建主从集群，一主一从、一主多从都可以；
  2. 数据库主机负责读写操作，从机只负责读操作；
  3. 数据库主机通过复制将数据同步到从机，每台数据库服务器都存储了所有的业务数据；
  4. 业务服务器将写操作发给数据库主机，将读操作发给数据库从机；

读写分离存在的问题：复制延迟带来的复杂性

**解决方案：**

* + 1. 写操作后的读操作指定发给读操作；
    2. 读从机失败后在读一次主机；（常说的：二次读取）
    3. 关键业务读写操作全部指向主机，非关键业务采用读写分离；（非关键业务：就是未影响到用户正常使用）

**不足之处：**

**a方案不足之处：**该方式和业务强绑定，对业务的侵入和影响较大，碰到新来的程序员，就会出现问题；

**b方案不足之处：**如果后很多二次读取，将大大增加主机读操作的压力，如 “黑客破解”

#### 分库分表

【分库】

业务分库按照业务模块将数据分散到不同的数据库服务器中；业务分库能够分散存储和访问压力，但同时也带来了新问题：

存在的问题：

* + - * + join操作问题：通过用户id分的多表，被分到不同数据库了，join用不了了；
        + 事务问题：原本同一个数据库中不同表可以在同一事务中修改；业务分库后，无法通过事务统一修改；
        + 成本问题：服务器数量增加；

解决方案：

join操作问题的解决方案：代码先从A数据库中查询购买化妆品的用户ID列表，然后到B数据库中查询这批用户id中的女性用户列表；

事务问题的解决方案：

1.分布式事务的解决方案（性能比较低）；

2.业务程序自己来模拟实现事务功能（例如：先扣减商品库存，扣减成功后生成订单，如果因为订单数据库异常导致生成订单失败，业务程序有需要将商品库存加上，但因为业务程序异常导致生成订单失败，则商品库存就无法恢复了，需要人工通过日志手动修复）；

【分表】

分表：垂直分表 、水平分表；

垂直分表：将二维表格从上往下切就是垂直分表；

垂直分表适合将表中某些不常用且占了大量空间的列拆分出去；

垂直分表引入的复杂性：体现在表操作的数量增加；

水平分表：将二位表格从左往右切就是水平分表；

水平分表适合表行数特别大的表，如果单表行数操作5000万行就必须进行分表（可以参考，并不是绝对标准）；

水平分表相比垂直分表引入了更多的复杂性；

**【路由】**

* + - * + 范围路由：选取有序的数据列，作为路由的条件，不同分段分散到不同的数据库表中；范围路由复杂点主要体现在分段大小的选取上；一般建议分段大小在100万~2000万之间；
        + Hash路由：选取某个列（或者某几个列组合）的值进行Hash运算，然后根据Hash结果分散到不同的数据库中；Hash路由复杂点主要体现在初始表数量的选取上；
        + 配置路由：配置路由就是路由表，用一张独立的表来记录路由信息；

优缺点：

* 范围路由：优点是可以随着数据的增加平滑地扩充新的表；缺点表的数据分布不均；
* Hash路由：优点是表的数据分布均匀，缺点是扩充新的表很麻烦，所有数据都要重新分布；
* 配置路由：优点设计简单，使用起来非常灵活，尤其在扩充表时，只需要迁移指定的数据，然后修改路由表就可以了；缺点是必须多查询几次，会影响整体性能；

【join操作】

水平分表后，数据分散在多个表中，如果需要与其他表进行join查询，则需要业务代码或数据库中间件中进行多次join查询，然后将结果合并；

【count()操作】

水平分表后，某些业务逻辑上需要将这些表当作一个表来处理。如记录总数分页展示；

* + - * + count()相加

在业务代码或数据库中间件中，对每个表进行count()操作，然后相加。

* + - * + 记录数表

新建一张表，表名为“记录数表”，包含table\_name,row\_count两个字段，每次插入或删除子表数据成功后，都更新“记录数表”；、

缺点是复杂度增加了不少，对子表的操作要同步“记录数表”，如果有个业务逻辑遗漏了，数据就会不一致；

解决方案：

对于一些不要求记录数实时保持精确的业务，可以通过后台定时更新记录表（实际上就是“count()相加” 和 “记录数表”）

【order by操作】

水平分表后，排序操作无法在数据库中完成，只能由业务代码或数据库中间件分别查询每个子表数据，然后汇总完成；

#### 实现方法

* 读写分离需要将读/写操作区分开，然后访问不同的数据库服务器；
* 分库分表需要根据不同的数据访问不同的数据库服务器；

两者都是一种分配机制，即将不同的SQL语句发送到不同的数据库服务器中；

常见的分配方式有两种：“程序代码封装”和“中间件封装”；

【程序代码封装】抽象一个数据库访问层来实现读写分离、分库分表。

特点：

* + - 实现简单，可以根据业务做较多的定制化的功能；
    - 每个编程语言都需要自己实现一次，无法通用；
    - 故障情况下，如果主从发生切换、则可能需要所有系统都修改配置并重启；

【中间件封装】中间件封装指的是独立一套系统出来，实现读写分离、分库分表。中间件对业务服务器提供SQL兼容的协议，在业务服务器看来，中间件就是一个数据库服务器；

特点：

* + - 支持多种编程语言，因为数据库中间件对业务服务器提供的是标准SQL接口；
    - 数据库中间件要支持完整的SQL语法和数据库服务器协议；
    - 数据库中间件自己不执行真正的读写操作，但所有的操作请求都要经过中间件，对中间件性能要求很高；
    - 数据库主从切换对业务服务器无感知，数据库中间件可以探测数据库服务器的主从状态；

## 知识点1.2 NoSQL

## 知识点1.3 缓存

## 知识点2：计算高性能

# day05高可用架构模式

## 知识点1：CAP

## 知识点2：FMEA

## 知识点3：存储高可用

## 知识点4：计算高可用

## 知识点5：业务高可用

# day06可扩展架构模式

## 知识点1：可扩展模式概念

## 知识点2：分层架构

## 知识点3：SOA架构

## 知识点4：微服务架构

## 知识点5：微内核架构（插件化架构）

# day07架构实战

## 知识点1：消息队列设计实战

## 知识点2：互联网架构演进

## 知识点3：互联网架构模板

## 知识点4：架构重构

## 知识点5：开源系统