1、由于MyRocks具有高效的空间利用率，相比InnoDB，同样大小的内存可缓存更多的数据量；相比pika等Redis替代方案，具有成熟的故障恢复机制和主从复制架构；此外其更低的复制延迟有利进行读能力扩展。因此，MyRocks也是较合适的Redis缓存替代方案。

2、IOC组件，Autofac组件，Identity 组件，Redis组件，RabbMQ组件

3、低成本

.NoSQL（Memcache、Redis等）的出现是为了解决关系型数据库无法应对高并发访问带来的访问压力。

.全文搜索引擎（Sphinx、Elasticsearch、Solr）的出现是为了解决关系型数据库like搜索的低效的问题。

.Hadoop的出现是为了解决传统文件系统无法应对海量数据存储和计算的问题。

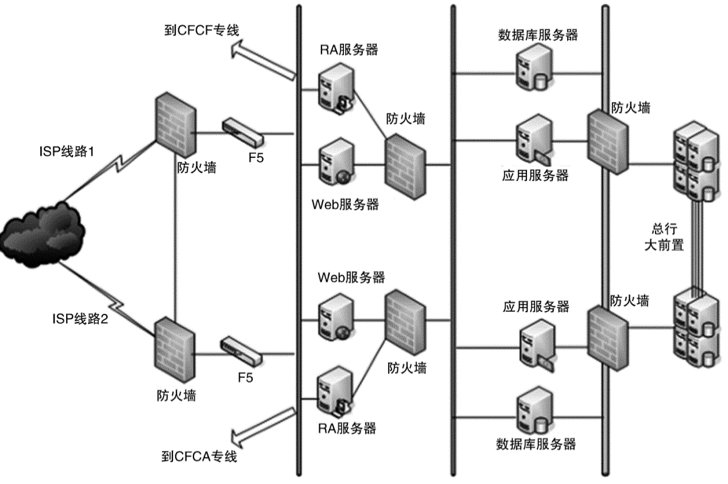
Facebook为了解决PHP的低效问题，刚开始的解决方案是HipHop PHP，可以将PHP语言翻译为C++语言执行，后来改为HHVM，将PHP翻译为字节码然后由虚拟机执行，和Java的JVM类似。

新浪微博将传统的Redis/MC + MySQL方式，扩展为Redis/MC + SSD Cache + MySQL方式，SSD Cache作为L2缓存使用，既解决了MC/Redis成本过高，容量小的问题，也解决了穿透DB带来的数据库访问压力（来源：http://www.infoq.com/cn/articles/weibo-platform-archieture ）。

Linkedin为了处理每天5千亿的事件，开发了高效的Kafka消息系统。

其他类似将Ruby on Rails改为Java、Lua + redis改为Go语言实现的例子还有很多。

3.1一个典型的银行系统的安全架构



3.2 MySQL单表的数据因不同的业务和应用场景会有不同的最优值，但不管怎样都肯定是有一定的限度的，一般推荐在5000万行左右。如果因为业务的发展，单表数据达到了10亿行，就会产生很多问题，例如：

添加索引会很慢，可能需要几个小时，这几个小时内数据库表是无法插入数据的，相当于业务停机了。

修改表结构和添加索引存在类似的问题，耗时可能会很长。

即使有索引，索引的性能也可能会很低，因为数据量太大。

数据库备份耗时很长。

4、架构三原则

**合适原则**：合适原则宣言：“合适优于业界领先”。

没那么多人，却想干那么多活，是失败的第一个主要原因。

没有那么多积累，却想一步登天，是失败的第二个主要原因。

没有那么卓越的业务场景，却幻想灵光一闪成为天才，是失败的第三个主要原因。

**简单原则**：简单原则宣言：“简单优于复杂”。包括结构简单和逻辑简单。

结构上的复杂性存在的第一个问题是，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。这个概率可以算出来，假设组件的故障率是10%（有10%的时间不可用），那么有3个组件的系统可用性是（1-10%）×（1-10%）×（1-10%）= 72.9%，有5个组件的系统可用性是（1-10%）×（1-10%）×（1-10%）×（1-10%）×（1-10%）=59%，两者的可用性相差13%。

**演化原则：**演化原则宣言：“演化优于一步到位”。 对于建筑来说，永恒是主题；而对于软件来说，变化才是主题。

软件架构描述的是一个软件系统的结构，包括各个模块，以及这些模块的关系；建筑架构描述的是一幢建筑的结构，包括各个部件，以及这些部件如何有机地组成成一幢完美的建筑。

**5、架构设计流程**

**5.1 识别复杂度**

QPS：Query Per Second，即每秒查询率，是对一个特定的查询服务器在规定时间内所处理流量多少的衡量标准。

TPS：Transactions Per Second，即每秒传输的事务处理个数，即服务器每秒处理的事物数。TPS包括一条消息入和一条消息出，加上一次用户数据库访问。（业务TPS = CAPS × 每个呼叫平均TPS）

TPS处理流程：

1、用户请求服务器

2、服务器自己的内部查询等处理

3、服务器返回给用户

这三个过程，每秒能够完成N个这三个过程，Tps也就是3；

**QPS 和 TPS 区别**

一次 TPS 表示一个事物的完整过程，这个过程中可能包含多个 QPS。例如：访问一次页面是一次 TPS，访问页面的时候页面有多个请求，每个请求算一次 QPS。

**系统吞吐量**

单位时间内系统处理请求的数量，体现系统的处理能力。

吞吐量 = （1000/响应时间 ms）x 并发数

系统吞吐量几个重要參数：QPS（TPS）、并发数、响应时间

QPS（TPS）：每秒钟 request/事务 数量

并发数： 系统同一时候处理的 request/事务数

响应时间： 一般取平均响应时间

**5.2设计备选方案**

常见的技术基础

a.NoSQL：Key-Value的存储和数据库的索引其实是类似的，Memcache只是把数据库的索引独立出来做成了一个缓存系统。

b.Hadoop大文件存储方案，基础其实是集群方案+ 数据复制方案。

c.Docker虚拟化，基础是LXC（Linux Containers）。

d.LevelDB的文件存储结构是Skip List。

e. 缓存方案中的Memcache和Redis为例，假如我们要挑选一个搭配MySQL使用的缓存，Memcache是纯内存缓存，支持基于一致性hash的集群；而Redis同时支持持久化、支持数据字典、支持主备、支持集群，看起来比Memcache好很多啊，所以就选Redis好了。

第一种常见的错误：设计最优秀的方案。

很多架构师在设计架构方案时，心里会默认有一种技术情结：我要设计一个优秀的架构，才能体现我的技术能力！例如，高可用的方案中，集群方案明显比主备方案要优秀和强大；高性能的方案中，淘宝的XX方案是业界领先的方案……

根据架构设计原则中“合适原则”和“简单原则“的要求，挑选合适自己业务、团队、技术能力的方案才是好方案；否则要么浪费大量资源开发了无用的系统（例如，之前提过的“亿级用户平台”的案例，设计了TPS 50000的系统，实际TPS只有500），要么根本无法实现（例如，10个人的团队要开发现在的整个淘宝系统）。

第二种常见的错误：只做一个方案。

很多架构师在做方案设计时，可能心里会简单地对几个方案进行初步的设想，再简单地判断哪个最好，然后就基于这个判断开始进行详细的架构设计了。

这样做有很多弊端：

A心里评估过于简单，可能没有想得全面，只是因为某一个缺点就把某个方案给否决了，而实际上没有哪个方案是完美的，某个地方有缺点的方案可能是综合来看最好的方案。

B架构师再怎么牛，经验知识和技能也有局限，有可能某个评估的标准或者经验是不正确的，或者是老的经验不适合新的情况，甚至有的评估标准是架构师自己原来就理解错了。

C单一方案设计会出现过度辩护的情况，即架构评审时，针对方案存在的问题和疑问，架构师会竭尽全力去为自己的设计进行辩护，经验不足的设计人员可能会强词夺理。

因此，架构师需要设计多个备选方案，但方案的数量可以说是无穷无尽的，架构师也不可能穷举所有方案，那合理的做法应该是什么样的呢？

A备选方案的数量以3 ~ 5个为最佳。少于3个方案可能是因为思维狭隘，考虑不周全；多于5个则需要耗费大量的精力和时间，并且方案之间的差别可能不明显。

B备选方案的差异要比较明显。例如，主备方案和集群方案差异就很明显，或者同样是主备方案，用ZooKeeper做主备决策和用Keepalived做主备决策的差异也很明显。但是都用ZooKeeper做主备决策，一个检测周期是1分钟，一个检测周期是5分钟，这就不是架构上的差异，而是细节上的差异了，不适合做成两个方案。

C备选方案的技术不要只局限于已经熟悉的技术。设计架构时，架构师需要将视野放宽，考虑更多可能性。很多架构师或者设计师积累了一些成功的经验，出于快速完成任务和降低风险的目的，可能自觉或者不自觉地倾向于使用自己已经熟悉的技术，对于新的技术有一种不放心的感觉。就像那句俗语说的：“如果你手里有一把锤子，所有的问题在你看来都是钉子”。例如，架构师对MySQL很熟悉，因此不管什么存储都基于MySQL去设计方案，系统性能不够了，首先考虑的就是MySQL分库分表，而事实上也许引入一个Memcache缓存就能够解决问题。

第三种常见的错误：备选方案过于详细。

有的架构师或者设计师在写备选方案时，错误地将备选方案等同于最终的方案，每个备选方案都写得很细。这样做的弊端显而易见：

A耗费了大量的时间和精力。

B将注意力集中到细节中，忽略了整体的技术设计，导致备选方案数量不够或者差异不大。

C评审的时候其他人会被很多细节给绕进去，评审效果很差。例如，评审的时候针对某个定时器应该是1分钟还是30秒，争论得不可开交。

正确的做法是备选阶段关注的是技术选型，而不是技术细节，技术选型的差异要比较明显。

**5.3评估和选择备选方案实战**

**1.备选方案1：**采用开源Kafka方案

a业务主管倾向于采用Kafka方案，因为Kafka已经比较成熟，各个业务团队或多或少都了解过Kafka。

b中间件团队部分研发人员也支持使用Kafka，因为使用Kafka能节省大量的开发投入；但部分人员认为Kafka可能并不适合我们的业务场景，因为Kafka的设计目的是为了支撑大容量的日志消息传输，而我们的消息队列是为了业务数据的可靠传输。

c运维代表提出了强烈的反对意见：首先，Kafka是Scala语言编写的，运维团队没有维护Scala语言开发的系统的经验，出问题后很难快速处理；其次，目前运维团队已经有一套成熟的运维体系，包括部署、监控、应急等，使用Kafka无法融入这套体系，需要单独投入运维人力。

d测试代表也倾向于引入Kafka，因为Kafka比较成熟，无须太多测试投入。

**2.备选方案2**：集群 + MySQL存储

a中间件团队的研发人员认为这个方案比较简单，但部分研发人员对于这个方案的性能持怀疑态度，毕竟使用MySQL来存储消息数据，性能肯定不如使用文件系统；并且有的研发人员担心做这样的方案是否会影响中间件团队的技术声誉，毕竟用MySQL来做消息队列，看起来比较“土”、比较另类。

b运维代表赞同这个方案，因为这个方案可以融入到现有的运维体系中，而且使用MySQL存储数据，可靠性有保证，运维团队也有丰富的MySQL运维经验；但运维团队认为这个方案的成本比较高，一个数据分组就需要4台机器（2台服务器 + 2台数据库）。

c测试代表认为这个方案测试人力投入较大，包括功能测试、性能测试、可靠性测试等都需要大量地投入人力。

d业务主管对这个方案既不肯定也不否定，因为反正都不是业务团队来投入人力来开发，系统维护也是中间件团队负责，对业务团队来说，只要保证消息队列系统稳定和可靠即可。

**3.备选方案**3：集群 + 自研存储系统

a中间件团队部分研发人员认为这是一个很好的方案，既能够展现中间件团队的技术实力，性能上相比MySQL也要高；但另外的研发人员认为这个方案复杂度太高，按照目前的团队人力和技术实力，要做到稳定可靠的存储系统，需要耗时较长的迭代，这个过程中消息队列系统可能因为存储出现严重问题，例如文件损坏导致丢失大量数据。

b运维代表不太赞成这个方案，因为运维之前遇到过几次类似的存储系统故障导致数据丢失的问题，损失惨重。例如，MongoDB丢数据、Tokyo Tyrant丢数据无法恢复等。运维团队并不相信目前的中间件团队的技术实力足以支撑自己研发一个存储系统（这让中间件团队的人员感觉有点不爽）。

c测试代表赞同运维代表的意见，并且自研存储系统的测试难度也很高，投入也很大。

d业务主管对自研存储系统也持保留意见，因为从历史经验来看，新系统上线肯定有bug，而存储系统出bug是最严重的，一旦出bug导致大量消息丢失，对系统的影响会严重。

*架构师经过思考后，给出了最终选择备选方案2，原因有：*

排除备选方案1的主要原因是可运维性，因为再成熟的系统，上线后都可能出问题，如果出问题无法快速解决，则无法满足业务的需求；并且Kafka的主要设计目标是高性能日志传输，而我们的消息队列设计的主要目标是业务消息的可靠传输。

排除备选方案3的主要原因是复杂度，目前团队技术实力和人员规模（总共6人，还有其他中间件系统需要开发和维护）无法支撑自研存储系统（参考架构设计原则2：简单原则）。

备选方案2的优点就是复杂度不高，也可以很好地融入现有运维体系，可靠性也有保障。

*针对备选方案2的缺点，架构师解释是：*

备选方案2的第一个缺点是性能，业务目前需要的性能并不是非常高，方案2能够满足，即使后面性能需求增加，方案2的数据分组方案也能够平行扩展进行支撑（参考架构设计原则3：演化原则）。

备选方案2的第二个缺点是成本，一个分组就需要4台机器，支撑目前的业务需求可能需要12台服务器，但实际上备机（包括服务器和数据库）主要用作备份，可以和其他系统并行部署在同一台机器上。

备选方案2的第三个缺点是技术上看起来并不很优越，但我们的设计目的不是为了证明自己（参考架构设计原则1：合适原则），而是更快更好地满足业务需求。

5.4 详细方案设计

详细设计方案阶段可能遇到的一种极端情况就是在详细设计阶段发现备选方案不可行，一般情况下主要的原因是备选方案设计时遗漏了某个关键技术点或者关键的质量属性。在备选方案阶段确定是可行的，但在详细方案设计阶段，发现由于细节点太多，方案非常庞大，整个项目可能要开发长达1年时间，最后只得废弃原来的备选方案，重新调整项目目标、计划和方案。这个项目的主要失误就是在备选方案评估时忽略了开发周期这个质量属性。

这种情况可以通过下面方式有效地避免：

\*\*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理解。\*\*例如，架构师选择了Elasticsearch作为全文搜索解决方案，前提必须是架构师自己对Elasticsearch的设计原理有深入的理解，比如索引、副本、集群等技术点；而不能道听途说Elasticsearch很牛，所以选择它，更不能成为把“细节我们不讨论”这句话挂在嘴边的“PPT架构师”。

\*\*通过分步骤、分阶段、分系统等方式，尽量降低方案复杂度，方案本身的复杂度越高，某个细节推翻整个方案的可能性就越高，适当降低复杂性，可以减少这种风险。

\*\*如果方案本身就很复杂，那就采取设计团队的方式来进行设计，博采众长，汇集大家的智慧和经验，防止只有1~2个架构师可能出现的思维盲点或者经验盲区。

**如上节所述选择备选方案2的设计：**

1.细化设计点1：数据库表如何设计？

数据库设计两类表，一类是日志表，用于消息写入时快速存储到MySQL中；另一类是消息表，每个消息队列一张表。

业务系统发布消息时，首先写入到日志表，日志表写入成功就代表消息写入成功；后台线程再从日志表中读取消息写入记录，将消息内容写入到消息表中。

业务系统读取消息时，从消息表中读取。

日志表表名为MQ\_LOG，包含的字段：日志ID、发布者信息、发布时间、队列名称、消息内容。

消息表表名就是队列名称，包含的字段：消息ID（递增生成）、消息内容、消息发布时间、消息发布者。

日志表需要及时清除已经写入消息表的日志数据，消息表最多保存30天的消息数据。

2.细化设计点2：数据如何复制？

直接采用MySQL主从复制即可，只复制消息存储表，不复制日志表。

3.细化设计点3：主备服务器如何倒换？

采用ZooKeeper来做主备决策，主备服务器都连接到ZooKeeper建立自己的节点，主服务器的路径规则为“/MQ/server/分区编号/master”，备机为“/MQ/server/分区编号/slave”，节点类型为EPHEMERAL。

备机监听主机的节点消息，当发现主服务器节点断连后，备服务器修改自己的状态，对外提供消息读取服务。

4.细化设计点4：业务服务器如何写入消息？

消息队列系统设计两个角色：生产者和消费者，每个角色都有唯一的名称。

消息队列系统提供SDK供各业务系统调用，SDK从配置中读取所有消息队列系统的服务器信息，SDK采取轮询算法发起消息写入请求给主服务器。如果某个主服务器无响应或者返回错误，SDK将发起请求发送到下一台服务器。

5.细化设计点5：业务服务器如何读取消息？

消息队列系统提供SDK供各业务系统调用，SDK从配置中读取所有消息队列系统的服务器信息，轮流向所有服务器发起消息读取请求。

消息队列服务器需要记录每个消费者的消费状态，即当前消费者已经读取到了哪条消息，当收到消息读取请求时，返回下一条未被读取的消息给消费者。

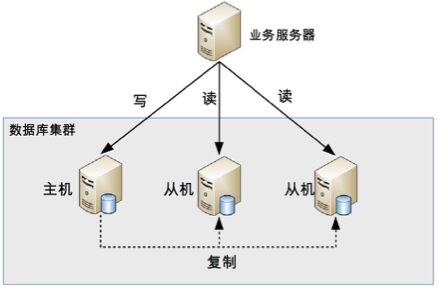
6.细化设计点6：业务服务器和消息队列服务器之间的通信协议如何设计？

考虑到消息队列系统后续可能会对接多种不同编程语言编写的系统，为了提升兼容性，传输协议用TCP，数据格式为ProtocolBuffer。

6、高性能数据库

6.1读写分离

读写分离的基本原理是将数据库读写操作分散到不同的节点上，下面是其基本架构图。



读写分离的基本实现是：

\*数据库服务器搭建主从集群，一主一从、一主多从都可以。

\*数据库主机负责读写操作，从机只负责读操作。

\*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。

\*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。

需要注意的是，这里用的是“主从集群”，而不是“主备集群”。“从机”的“从”可以理解为“仆从”，仆从是要帮主人干活的，“从机”是需要提供读数据的功能的；而“备机”一般被认为仅仅提供备份功能，不提供访问功能。所以使用“主从”还是“主备”，是要看场景的，这两个词并不是完全等同的。

读写分离的实现逻辑并不复杂，但有两个细节点将引入设计复杂度：主从复制延迟和分配机制。

解决主从复制延迟有几种常见的方法：

1.写操作后的读操作指定发给数据库主服务器

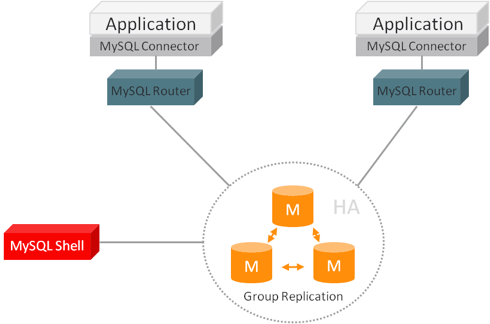
例如，注册账号完成后，登录时读取账号的读操作也发给数据库主服务器。这种方式和业务强绑定，对业务的侵入和影响较大，如果哪个新来的程序员不知道这样写代码，就会导致一个bug。

2.读从机失败后再读一次主机

这就是通常所说的“二次读取”，二次读取和业务无绑定，只需要对底层数据库访问的API进行封装即可，实现代价较小，不足之处在于如果有很多二次读取，将大大增加主机的读操作压力。例如，黑客暴力破解账号，会导致大量的二次读取操作，主机可能顶不住读操作的压力从而崩溃。

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

目前的开源数据库中间件方案中，MySQL官方先是提供了MySQL Proxy，但MySQL Proxy一直没有正式GA，现在MySQL官方推荐MySQL Router。MySQL Router的主要功能有读写分离、故障自动切换、负载均衡、连接池等，其基本架构如下：



6.2分库分表

业务分库指的是按照业务模块将数据分散到不同的数据库服务器。\*\*例如，一个简单的电商网站，包括用户、商品、订单三个业务模块，我们可以将用户数据、商品数据、订单数据分开放到三台不同的数据库服务器上，而不是将所有数据都放在一台数据库服务器上。

虽然业务分库能够分散存储和访问压力，但同时也带来了新的问题：

\*.join操作问题

业务分库后，原本在同一个数据库中的表分散到不同数据库中，导致无法使用SQL的join查询。数据分散在两个不同的数据库中，无法做join查询，只能采取先从订单数据库中查询购买了化妆品的用户ID列表，然后再到用户数据库中查询这批用户ID中的女性用户列表，这样实现就比简单的join查询要复杂一些。

\*.事务问题

原本在同一个数据库中不同的表可以在同一个事务中修改，业务分库后，表分散到不同的数据库中，无法通过事务统一修改。虽然数据库厂商提供了一些分布式事务的解决方案（例如，MySQL的XA），但性能实在太低，与高性能存储的目标是相违背的。

\*.成本问题

业务分库同时也带来了成本的代价，本来1台服务器搞定的事情，现在要3台，如果考虑备份，那就是2台变成了6台。

分表能够有效地分散存储压力和带来性能提升，但和分库一样，也会引入各种复杂性。

A.垂直分表

不同字段分配到不同表中，常用的一组，不常用的一组等方式去分

B. 水平分表

水平分表适合表行数特别大的表，有的公司要求单表行数超过5000万就必须进行分表，这个数字可以作为参考，但并不是绝对标准，关键还是要看表的访问性能。对于一些比较复杂的表，可能超过1000万就要分表了；而对于一些简单的表，即使存储数据超过1亿行，也可以不分表。但不管怎样，当看到表的数据量达到千万级别时，作为架构师就要警觉起来，因为这很可能是架构的性能瓶颈或者隐患。

水平分表相比垂直分表，会引入更多的复杂性，主要表现在下面几个方面：

**路由：包括范围路由；Hash路由；配置路由**

\*\*范围路由：\*\*选取有序的数据列（例如，整形、时间戳等）作为路由的条件，不同分段分散到不同的数据库表中。以最常见的用户ID为例，路由算法可以按照1000000的范围大小进行分段，1 ~ 999999放到数据库1的表中，1000000 ~ 1999999放到数据库2的表中，以此类推。

\*\*Hash路由：\*\*选取某个列（或者某几个列组合也可以）的值进行Hash运算，然后根据Hash结果分散到不同的数据库表中。

\*\*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。

**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，需要在业务代码或者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，虽然物理上数据分散到多个表中，但某些业务逻辑上还是会将这些表当作一个表来处理。例如，获取记录总数用于分页或者展示，水平分表前用一个count()就能完成的操作，在分表后就没那么简单了。

\*\*记录数表：\*\*具体做法是新建一张表，假如表名为“记录数表”，包含table\\_name、row\\_count两个字段，每次插入或者删除子表数据成功后，都更新“记录数表”。

7、NoSQL

关系数据库存在如下缺点:

\*\*关系数据库存储的是行记录，无法存储数据结构.

\*\*关系数据库的schema扩展很不方便

关系数据库的表结构schema是强约束，操作不存在的列会报错，业务变化时扩充列也比较麻烦，需要执行DDL（data definition language，如CREATE、ALTER、DROP等）语句修改，而且修改时可能会长时间锁表（例如，MySQL可能将表锁住1个小时）

\*\*关系数据库在大数据场景下I/O较高

如果对一些大量数据的表进行统计之类的运算，关系数据库的I/O会很高，因为即使只针对其中某一列进行运算，关系数据库也会将整行数据从存储设备读入内存。

\*\*关系数据库的全文搜索功能比较弱

关系数据库的全文搜索只能使用like进行整表扫描匹配，性能非常低，在互联网这种搜索复杂的场景下无法满足业务要求。

**NoSQL方案带来的优势**，本质上是牺牲ACID中的某个或者某几个特性，因此我们不能盲目地迷信NoSQL是银弹，而应该将NoSQL作为SQL的一个有力补充，NoSQL != No SQL，而是NoSQL = Not Only SQL。

常见的NoSQL方案分为4类:

\*\*K-V存储：解决关系数据库无法存储数据结构的问题，以Redis为代表。

Redis是K-V存储的典型代表，它是一款开源（基于BSD许可）的高性能K-V缓存和存储系统。Redis的缺点主要体现在并不支持完整的ACID事务，Redis虽然提供事务功能，但Redis的事务和关系数据库的事务不可同日而语，Redis的事务只能保证隔离性和一致性（I和C），无法保证原子性和持久性（A和D）。

\*\*文档数据库：解决关系数据库强schema约束的问题，以MongoDB为代表。

绝大部分文档数据库存储的数据格式是JSON（或者BSON），因为JSON数据是自描述的，无须在使用前定义字段，读取一个JSON中不存在的字段也不会导致SQL那样的语法错误。

文档数据库no-schema的特性带来的这些优势也是有代价的，最主要的代价就是不支持事务。文档数据库另外一个缺点就是无法实现关系数据库的join操作。

\*\*列式数据库：解决关系数据库大数据场景下的I/O问题，以HBase为代表。

行式存储的优势是在特定的业务场景下才能体现，如果不存在这样的业务场景，那么行式存储的优势也将不复存在，甚至成为劣势，典型的场景就是海量数据进行统计。计算某个城市体重超重的人员数据，实际上只需要读取每个人的体重这一列并进行统计即可，而行式存储即使最终只使用一列，也会将所有行数据都读取出来。如果单行用户信息有1KB，其中体重只有4个字节，行式存储还是会将整行1KB数据全部读取到内存中，这是明显的浪费。而如果采用列式存储，每个用户只需要读取4字节的体重数据即可，I/O将大大减少。

需要频繁地更新多个列。因为列式存储将不同列存储在磁盘上不连续的空间，导致更新多个列时磁盘是随机写操作；而行式存储时同一行多个列都存储在连续的空间，一次磁盘写操作就可以完成，列式存储的随机写效率要远远低于行式存储的写效率。此外，列式存储高压缩率在更新场景下也会成为劣势，因为更新时需要将存储数据解压后更新，然后再压缩，最后写入磁盘。

\*\*全文搜索引擎：解决关系数据库的全文搜索性能问题，以Elasticsearch为代表。

搜索条件是无法列举完全的，各种排列组合非常多，通过这个简单的样例我们就可以看出关系数据库在支撑全文搜索时的不足。Elastcisearch是分布式的文档存储方式。它能存储和检索复杂的数据结构——序列化成为JSON文档——以实时的方式。在Elasticsearch中，每个字段的所有数据都是默认被索引的。即每个字段都有为了快速检索设置的专用倒排索引。而且，不像其他多数的数据库，它能在相同的查询中使用所有倒排索引，并以惊人的速度返回结果。

8、高性能缓存架构

**缓存穿透**是指缓存没有发挥作用，业务系统虽然去缓存查询数据，但缓存中没有数据，业务系统需要再次去存储系统查询数据。通常有以下二种情况：

A.存储数据不存在，这种情况的解决办法比较简单，如果查询存储系统的数据没有找到，则直接设置一个默认值（可以是空值，也可以是具体的值）存到缓存中。

B. 缓存数据生成耗费大量时间或者资源，此情况是存储系统中存在数据，但生成缓存数据需要耗费较长时间或者耗费大量资源。如果刚好在业务访问的时候缓存失效了，那么也会出现缓存没有发挥作用，访问压力全部集中在存储系统上的情况。

**缓存雪崩**是指当缓存失效（过期）后引起系统性能急剧下降的情况。当缓存过期被清除后，业务系统需要重新生成缓存，因此需要再次访问存储系统，再次进行运算，这个处理步骤耗时几十毫秒甚至上百毫秒。而对于一个高并发的业务系统来说，几百毫秒内可能会接到几百上千个请求。由于旧的缓存已经被清除，新的缓存还未生成，并且处理这些请求的线程都不知道另外有一个线程正在生成缓存，因此所有的请求都会去重新生成缓存，都会去访问存储系统，从而对存储系统造成巨大的性能压力。这些压力又会拖慢整个系统，严重的会造成数据库宕机，从而形成一系列连锁反应，造成整个系统崩溃。

缓存雪崩的常见解决方法有两种：更新锁机制和后台更新机制：

A.更新锁，对缓存更新操作进行加锁保护，保证只有一个线程能够进行缓存更新，未能获取更新锁的线程要么等待锁释放后重新读取缓存，要么就返回空值或者默认值。

B. 后台更新，由后台线程来更新缓存，而不是由业务线程来更新缓存，缓存本身的有效期设置为永久，后台线程定时更新缓存。

后台更新机制还适合业务刚上线的时候进行缓存预热。缓存预热指系统上线后，将相关的缓存数据直接加载到缓存系统，而不是等待用户访问才来触发缓存加载。

**缓存热点**，虽然缓存系统本身的性能比较高，但对于一些特别热点的数据，如果大部分甚至所有的业务请求都命中同一份缓存数据，则这份数据所在的缓存服务器的压力也很大。

缓存热点的解决方案就是复制多份缓存副本，将请求分散到多个缓存服务器上，减轻缓存热点导致的单台缓存服务器压力。以微博为例，对于粉丝数超过100万的明星，每条微博都可以生成100份缓存，缓存的数据是一样的，通过在缓存的key里面加上编号进行区分，每次读缓存时都随机读取其中某份缓存。

缓存副本设计有一个细节需要注意，就是不同的缓存副本不要设置统一的过期时间，否则就会出现所有缓存副本同时生成同时失效的情况，从而引发缓存雪崩效应。正确的做法是设定一个过期时间范围，不同的缓存副本的过期时间是指定范围内的随机值。

9、单服务器高性能模式

单服务器高性能的关键之一就是服务器采取的并发模型，并发模型有如下两个关键设计点：

\*服务器如何管理连接。

\*服务器如何处理请求。

以上两个设计点最终都和操作系统的I/O模型及进程模型相关。

\*I/O模型：阻塞、非阻塞、同步、异步。

\*进程模型：单进程、多进程、多线程。

**PPC模式中**，当连接进来时才fork新进程来处理连接请求，由于fork进程代价高，用户访问时可能感觉比较慢，prefork模式的出现就是为了解决这个问题。

prefork模式和PPC一样，还是存在父子进程通信复杂、支持的并发连接数量有限的问题，因此目前实际应用也不多。Apache服务器提供了MPM prefork模式，推荐在需要可靠性或者与旧软件兼容的站点时采用这种模式，默认情况下最大支持256个并发连接。

**TPC**是Thread Per Connection的缩写，其含义是指每次有新的连接就新建一个线程去专门处理这个连接的请求。与进程相比，线程更轻量级，创建线程的消耗比进程要少得多；同时多线程是共享进程内存空间的，线程通信相比进程通信更简单。因此，TPC实际上是解决或者弱化了PPC fork代价高的问题和父子进程通信复杂的问题。

TPC虽然解决了fork代价高和进程通信复杂的问题，但是也引入了新的问题，具体表现在：

\*创建线程虽然比创建进程代价低，但并不是没有代价，高并发时（例如每秒上万连接）还是有性能问题。

\*无须进程间通信，但是线程间的互斥和共享又引入了复杂度，可能一不小心就导致了死锁问题。

\*多线程会出现互相影响的情况，某个线程出现异常时，可能导致整个进程退出（例如内存越界）。

TPC还是存在CPU线程调度和切换代价的问题。因此，TPC方案本质上和PPC方案基本类似，在并发几百连接的场景下，反而更多地是采用PPC的方案，因为PPC方案不会有死锁的风险，也不会多进程互相影响，稳定性更高。

prethread

TPC模式中，当连接进来时才创建新的线程来处理连接请求，虽然创建线程比创建进程要更加轻量级，但还是有一定的代价，而prethread模式就是为了解决这个问题。

和prefork类似，prethread模式会预先创建线程，然后才开始接受用户的请求，当有新的连接进来的时候，就可以省去创建线程的操作，让用户感觉更快、体验更好。

由于多线程之间数据共享和通信比较方便，因此实际上prethread的实现方式相比prefork要灵活一些，常见的实现方式有下面几种：

\*主进程accept，然后将连接交给某个线程处理。

\*子线程都尝试去accept，最终只有一个线程accept成功。

I/O多路复用技术归纳起来有两个关键实现点：

\*当多条连接共用一个阻塞对象后，进程只需要在一个阻塞对象上等待，而无须再轮询所有连接，常见的实现方式有select、epoll、kqueue等。

\*当某条连接有新的数据可以处理时，操作系统会通知进程，进程从阻塞状态返回，开始进行业务处理。

**Reactor模式**的核心组成部分包括Reactor和处理资源池（进程池或线程池），其中Reactor负责监听和分配事件，处理资源池负责处理事件。

Reactor模式也叫Dispatcher模式（在很多开源的系统里面会看到这个名称的类，其实就是实现Reactor模式的），更加贴近模式本身的含义，即I/O多路复用统一监听事件，收到事件后分配（Dispatch）给某个进程。

最终Reactor模式有这三种典型的实现方案：

\***单Reactor单进程/线程**

单Reactor单进程的模式优点就是很简单，没有进程间通信，没有进程竞争，全部都在同一个进程内完成。但其缺点也是非常明显，具体表现有：

a.只有一个进程，无法发挥多核CPU的性能；只能采取部署多个系统来利用多核CPU，但这样会带来运维复杂度，本来只要维护一个系统，用这种方式需要在一台机器上维护多套系统。

b.Handler在处理某个连接上的业务时，整个进程无法处理其他连接的事件，很容易导致性能瓶颈。

因此，单Reactor单进程的方案在实践中应用场景不多，只适用于业务处理非常快速的场景，目前比较著名的开源软件中使用单Reactor单进程的是Redis。

\*单Reactor多线程方案：

a. 主线程中，Reactor对象通过select监控连接事件，收到事件后通过dispatch进行分发。

b. 如果是连接建立的事件，则由Acceptor处理，Acceptor通过accept接受连接，并创建一个Handler来处理连接后续的各种事件。

c. 如果不是连接建立事件，则Reactor会调用连接对应的Handler（第2步中创建的Handler）来进行响应。

d. Handler只负责响应事件，不进行业务处理；Handler通过read读取到数据后，会发给Processor进行业务处理。

e. Processor会在独立的子线程中完成真正的业务处理，然后将响应结果发给主进程的Handler处理；Handler收到响应后通过send将响应结果返回给client。

**单Reator多线程***方案能够充分利用多核多CPU的处理能力，但同时也存在下面的问题*：

\*\*多线程数据共享和访问比较复杂。例如，子线程完成业务处理后，要把结果传递给主线程的Reactor进行发送，这里涉及共享数据的互斥和保护机制。以Java的NIO为例，Selector是线程安全的，但是通过Selector.selectKeys()返回的键的集合是非线程安全的，对selected keys的处理必须单线程处理或者采取同步措施进行保护。

\*\*Reactor承担所有事件的监听和响应，只在主线程中运行，瞬间高并发时会成为性能瓶颈。

\***多Reactor多进程/线程**。

方案详细说明如下：

\*\*父进程中mainReactor对象通过select监控连接建立事件，收到事件后通过Acceptor接收，将新的连接分配给某个子进程。

\*\*子进程的subReactor将mainReactor分配的连接加入连接队列进行监听，并创建一个Handler用于处理连接的各种事件。

\*\*当有新的事件发生时，subReactor会调用连接对应的Handler（即第2步中创建的Handler）来进行响应。

\*\*Handler完成read→业务处理→send的完整业务流程。

多Reactor多进程/线程的方案看起来比单Reactor多线程要复杂，但实际实现时反而更加简单，主要原因是：

\*\*父进程和子进程的职责非常明确，父进程只负责接收新连接，子进程负责完成后续的业务处理。

\*\*父进程和子进程的交互很简单，父进程只需要把新连接传给子进程，子进程无须返回数据。

\*\*子进程之间是互相独立的，无须同步共享之类的处理（这里仅限于网络模型相关的select、read、send等无须同步共享，“业务处理”还是有可能需要同步共享的）。

目前著名的开源系统Nginx采用的是多Reactor多进程，采用多Reactor多线程的实现有Memcache和Netty。Nginx采用的是多Reactor多进程的模式，但方案与标准的多Reactor多进程有差异。具体差异表现为主进程中仅仅创建了监听端口，并没有创建mainReactor来“accept”连接，而是由子进程的Reactor来“accept”连接，通过锁来控制一次只有一个子进程进行“accept”，子进程“accept”新连接后就放到自己的Reactor进行处理，不会再分配给其他子进程。

**Proactor** Proactor中文翻译为“前摄器”比较难理解，与其类似的单词是proactive，含义为“主动的”，因此我们照猫画虎翻译为“主动器”反而更好理解。

Reactor是非阻塞同步网络模型，因为真正的read和send操作都需要用户进程同步操作。这里的“同步”指用户进程在执行read和send这类I/O操作的时候是同步的，如果把I/O操作改为异步就能够进一步提升性能，这就是异步网络模型Proactor。

Proactor方案：

\*\*Proactor Initiator负责创建Proactor和Handler，并将Proactor和Handler都通过Asynchronous Operation Processor注册到内核。

\*\*Asynchronous Operation Processor负责处理注册请求，并完成I/O操作。

\*\*Asynchronous Operation Processor完成I/O操作后通知Proactor。

\*\*Proactor根据不同的事件类型回调不同的Handler进行业务处理。

\*\*Handler完成业务处理，Handler也可以注册新的Handler到内核进程。

Windows下通过IOCP实现了真正的异步I/O，而在Linux系统下的AIO并不完善，因此在Linux下实现高并发网络编程时都是以Reactor模式为主。所以即使Boost.Asio号称实现了Proactor模型，其实它在Windows下采用IOCP，而在Linux下是用Reactor模式（采用epoll）模拟出来的异步模型。

10、负载均衡

高性能集群的复杂性主要体现在需要增加一个任务分配器，以及为任务选择一个合适的任务分配算法。对于任务分配器，现在更流行的通用叫法是“负载均衡器”。但这个名称有一定的误导性，会让人潜意识里认为任务分配的目的是要保持各个计算单元的负载达到均衡状态。而实际上任务分配并不只是考虑计算单元的负载均衡，不同的任务分配算法目标是不一样的，有的基于负载考虑，有的基于性能（吞吐量、响应时间）考虑，有的基于业务考虑。考虑到“负载均衡”已经成为了事实上的标准术语，这里我也用“负载均衡”来代替“任务分配”，但请你时刻记住，负载均衡不只是为了计算单元的负载达到均衡状态。

常见的负载均衡系统包括3种：DNS负载均衡、硬件负载均衡和软件负载均衡。

常见的负载均衡系统包括3种：DNS负载均衡、硬件负载均衡和软件负载均衡。

**\*DNS负载均衡**

DNS是最简单也是最常见的负载均衡方式，一般用来实现地理级别的均衡。例如，北方的用户访问北京的机房，南方的用户访问深圳的机房。

**\*硬件负载均衡**

硬件负载均衡是通过单独的硬件设备来实现负载均衡功能，这类设备和路由器、交换机类似，可以理解为一个用于负载均衡的基础网络设备。目前业界典型的硬件负载均衡设备有两款：F5和A10。这类设备性能强劲、功能强大，但价格都不便宜，一般只有“土豪”公司才会考虑使用此类设备。普通业务量级的公司一是负担不起，二是业务量没那么大，用这些设备也是浪费。

**\*软件负载均衡**

软件负载均衡通过负载均衡软件来实现负载均衡功能，常见的有Nginx和LVS，其中Nginx是软件的7层负载均衡，LVS是Linux内核的4层负载均衡。4层和7层的区别就在于协议和灵活性，Nginx支持HTTP、E-mail协议；而LVS是4层负载均衡，和协议无关，几乎所有应用都可以做，例如，聊天、数据库等。

软件负载均衡的优点：

\*\*简单：无论是部署还是维护都比较简单。

\*\*便宜：只要买个Linux服务器，装上软件即可。

\*\*灵活：4层和7层负载均衡可以根据业务进行选择；也可以根据业务进行比较方便的扩展，例如，可以通过Nginx的插件来实现业务的定制化功能。

其实下面的缺点都是和硬件负载均衡相比的，并不是说软件负载均衡没法用。

\*\*性能一般：一个Nginx大约能支撑5万并发。

\*\*功能没有硬件负载均衡那么强大。

\*\*一般不具备防火墙和防DDoS攻击等安全功能。

**负载均衡算法**

\*轮询

负载均衡系统收到请求后，按照顺序轮流分配到服务器上。“简单”是轮询算法的优点，也是它的缺点。

\*加权轮询

负载均衡系统根据服务器权重进行任务分配，这里的权重一般是根据硬件配置进行静态配置的，采用动态的方式计算会更加契合业务，但复杂度也会更高。

加权轮询解决了轮询算法中无法根据服务器的配置差异进行任务分配的问题，但同样存在无法根据服务器的状态差异进行任务分配的问题。

\*负载最低优先

负载最低优先的算法解决了轮询算法中无法感知服务器状态的问题，由此带来的代价是复杂度要增加很多。例如：

\*\*最少连接数优先的算法要求负载均衡系统统计每个服务器当前建立的连接，其应用场景仅限于负载均衡接收的任何连接请求都会转发给服务器进行处理，否则如果负载均衡系统和服务器之间是固定的连接池方式，就不适合采取这种算法。例如，LVS可以采取这种算法进行负载均衡，而一个通过连接池的方式连接MySQL集群的负载均衡系统就不适合采取这种算法进行负载均衡。

\*\*CPU负载最低优先的算法要求负载均衡系统以某种方式收集每个服务器的CPU负载，而且要确定是以1分钟的负载为标准，还是以15分钟的负载为标准，不存在1分钟肯定比15分钟要好或者差。不同业务最优的时间间隔是不一样的，时间间隔太短容易造成频繁波动，时间间隔太长又可能造成峰值来临时响应缓慢。

\*性能最优类

负载最低优先类算法是站在服务器的角度来进行分配的，而性能最优优先类算法则是站在客户端的角度来进行分配的，优先将任务分配给处理速度最快的服务器，通过这种方式达到最快响应客户端的目的。

\*Hash类

负载均衡系统根据任务中的某些关键信息进行Hash运算，将相同Hash值的请求分配到同一台服务器上，这样做的目的主要是为了满足特定的业务需求。

11、FMEA方法

FMEA（Failure mode and effects analysis，故障模式与影响分析）又称为失效模式与后果分析、失效模式与效应分析、故障模式与后果分析等，专栏采用“故障模式与影响分析”，因为这个中文翻译更加符合可用性的语境。FMEA是一种在各行各业都有广泛应用的可用性分析方法，通过对系统范围内潜在的故障模式加以分析，并按照严重程度进行分类，以确定失效对于系统的最终影响。

FMEA的具体分析方法是：

\*给出初始的架构设计图。

\*假设架构中某个部件发生故障。

\*分析此故障对系统功能造成的影响。

\*根据分析结果，判断架构是否需要进行优化。

常见的FMEA分析表格包含下面部分。

\*1.功能点

当前的FMEA分析涉及的功能点，注意这里的“功能点”指的是从用户角度来看的，而不是从系统各个模块功能点划分来看的。例如，对于一个用户管理系统，使用FMEA分析时 “登录”“注册”才是功能点，而用户管理系统中的数据库存储功能、Redis缓存功能不能作为FMEA分析的功能点。

\*2.故障模式

故障模式指的是系统会出现什么样的故障，包括故障点和故障形式。需要特别注意的是，这里的故障模式并不需要给出真正的故障原因，我们只需要假设出现某种故障现象即可，例如MySQL响应时间达到3秒。造成MySQL响应时间达到3秒可能的原因很多：磁盘坏道、慢查询、服务器到MySQL的连接网络故障、MySQL bug等，我们并不需要在故障模式中一一列出来，而是在后面的“故障原因”一节中列出来。因为在实际应用过程中，不管哪种原因，只要现象是一样的，对业务的影响就是一样的。

此外，故障模式的描述要尽量精确，多使用量化描述，避免使用泛化的描述。例如，推荐使用“MySQL响应时间达到3秒”，而不是“MySQL响应慢”。

\*3.故障影响

当发生故障模式中描述的故障时，功能点具体会受到什么影响。常见的影响有：功能点偶尔不可用、功能点完全不可用、部分用户功能点不可用、功能点响应缓慢、功能点出错等。

故障影响也需要尽量准确描述。例如，推荐使用“20%的用户无法登录”，而不是“大部分用户无法登录”。要注意这里的数字不需要完全精确，比如21.25%这样的数据其实是没有必要的，我们只需要预估影响是20%还是40%。

\*4.严重程度

严重程度指站在业务的角度故障的影响程度，一般分为“致命/高/中/低/无”五个档次。严重程度按照这个公式进行评估：严重程度 = 功能点重要程度 × 故障影响范围 × 功能点受损程度。同样以用户管理系统为例：登录功能比修改用户资料要重要得多，80%的用户比20%的用户范围更大，完全无法登录比登录缓慢要更严重。因此我们可以得出如下故障模式的严重程度。

\*\*致命：超过70%用户无法登录。

\*\*高：超过30%的用户无法登录。

\*\*中：所有用户登录时间超过5秒。

\*\*低：10%的用户登录时间超过5秒。

\*\*中：所有用户都无法修改资料。

\*\*低：20%的用户无法修改头像。

\*5.故障原因

“故障模式”中只描述了故障的现象，并没有单独列出故障原因。主要原因在于不管什么故障原因，故障现象相同，对功能点的影响就相同。

\*6.故障概率

这里的概率就是指某个具体故障原因发生的概率。

\*7.风险程度

风险程度就是综合严重程度和故障概率来一起判断某个故障的最终等级，风险程度 = 严重程度 × 故障概率。因此可能出现某个故障影响非常严重，但其概率很低，最终来看风险程度就低。

\*8.已有措施

针对具体的故障原因，系统现在是否提供了某些措施来应对，包括：检测告警、容错、自恢复等。

\*9.规避措施

规避措施指为了降低故障发生概率而做的一些事情，可以是技术手段，也可以是管理手段。例如：

\*\*技术手段：为了避免新引入的MongoDB丢失数据，在MySQL中冗余一份。

\*\*管理手段：为了降低磁盘坏道的概率，强制统一更换服务时间超过2年的磁盘。

\*10.解决措施

解决措施指为了能够解决问题而做的一些事情，一般都是技术手段。

\*11.后续规划

综合前面的分析，就可以看出哪些故障我们目前还缺乏对应的措施，哪些已有措施还不够，针对这些不足的地方，再结合风险程度进行排序，给出后续的改进规划。这些规划既可以是技术手段，也可以是管理手段；可以是规避措施，也可以是解决措施。同时需要考虑资源的投入情况，优先将风险程度高的系统隐患解决。

12、高可用存储架构

存储高可用方案的本质都是通过将数据复制到多个存储设备，通过数据冗余的方式来实现高可用，其复杂性主要体现在如何应对复制延迟和中断导致的数据不一致问题。

常见的高可用存储架构有主备、主从、主主、集群、分区，每一种又可以根据业务的需求进行一些特殊的定制化功能，由此衍生出更多的变种。

**\*主备复制**

主备复制是最常见也是最简单的一种存储高可用方案，几乎所有的存储系统都提供了主备复制的功能，例如MySQL、Redis、MongoDB等。

主备复制架构的缺点主要有：

\*\*备机仅仅只为备份，并没有提供读写操作，硬件成本上有浪费。

\*\*故障后需要人工干预，无法自动恢复。人工处理的效率是很低的，可能打电话找到能够操作的人就耗费了10分钟，甚至如果是深更半夜，出了故障都没人知道。人工在执行恢复操作的过程中也容易出错，因为这类操作并不常见，可能1年就2、3次，实际操作的时候很可能遇到各种意想不到的问题。

**\*主从复制**

主从复制和主备复制只有一字之差，“从”意思是“随从、仆从”，“备”的意思是备份。我们可以理解为仆从是要帮主人干活的，这里的干活就是承担“读”的操作。也就是说，主机负责读写操作，从机只负责读操作，不负责写操作。

缺点有：

\*\*主从复制架构中，客户端需要感知主从关系，并将不同的操作发给不同的机器进行处理，复杂度比主备复制要高。

\*\*主从复制架构中，从机提供读业务，如果主从复制延迟比较大，业务会因为数据不一致出现问题。

\*\*故障时需要人工干预。

**\*双机切换**

要实现一个完善的切换方案，必须考虑这几个关键的设计点：

\*\*主备间状态判断

主要包括两方面：状态传递的渠道，以及状态检测的内容。

状态传递的渠道：是相互间互相连接，还是第三方仲裁？

状态检测的内容：例如机器是否掉电、进程是否存在、响应是否缓慢等。

\*\*切换决策

主要包括几方面：切换时机、切换策略、自动程度。

切换时机：什么情况下备机应该升级为主机？是机器掉电后备机才升级，还是主机上的进程不存在就升级，还是主机响应时间超过2秒就升级，还是3分钟内主机连续重启3次就升级等。

切换策略：原来的主机故障恢复后，要再次切换，确保原来的主机继续做主机，还是原来的主机故障恢复后自动成为新的备机？

自动程度：切换是完全自动的，还是半自动的？例如，系统判断当前需要切换，但需要人工做最终的确认操作（例如，单击一下“切换”按钮）。

\*\*数据冲突解决

当原有故障的主机恢复后，新旧主机之间可能存在数据冲突。

**常见架构**

\*\*互连式，互连式就是指主备机直接建立状态传递的渠道。

\*\*中介式，中介式指的是在主备两者之外引入第三方中介，主备机之间不直接连接，而都去连接中介，并且通过中介来传递状态信息，开源方案已经有比较成熟的中介式解决方案，例如ZooKeeper和Keepalived。ZooKeeper本身已经实现了高可用集群架构，因此已经帮我们解决了中介本身的可靠性问题，在工程实践中推荐基于ZooKeeper搭建中介式切换架构。

\*模拟式，模拟式指主备机之间并不传递任何状态数据，而是备机模拟成一个客户端，向主机发起模拟的读写操作，根据读写操作的响应情况来判断主机的状态。

**\*主主复制**

主主复制指的是两台机器都是主机，互相将数据复制给对方，客户端可以任意挑选其中一台机器进行读写操作。

主主复制架构从总体上来看要简单很多，无须状态信息传递，也无须状态决策和状态切换。然而事实上主主复制架构也并不简单，而是有其独特的复杂性，具体表现在：如果采取主主复制架构，必须保证数据能够双向复制，而很多数据是不能双向复制的。

主主复制架构对数据的设计有严格的要求，一般适合于那些临时性、可丢失、可覆盖的数据场景。

**\*数据集群**

集群就是多台机器组合在一起形成一个统一的系统，这里的“多台”，数量上至少是3台；相比而言，主备、主从都是2台机器。根据集群中机器承担的不同角色来划分，集群可以分为两类：数据集中集群、数据分散集群。

13、计算高可用性

计算高可用的主要设计目标是当出现部分硬件损坏时，计算任务能够继续正常运行。因此计算高可用的本质是通过冗余来规避部分故障的风险，单台服务器是无论如何都达不到这个目标的。所以计算高可用的设计思想很简单：通过增加更多服务器来达到计算高可用。

常见的计算高可用架构：主备、主从和集群：

**主备**

主备架构是计算高可用最简单的架构，和存储高可用的主备复制架构类似，但是要更简单一些，因为计算高可用的主备架构无须数据复制。

**主从**

和存储高可用中的主从复制架构类似，计算高可用的主从架构中的从机也是要执行任务的。任务分配器需要将任务进行分类，确定哪些任务可以发送给主机执行，哪些任务可以发送给备机执行

**集群**

主备架构和主从架构通过冗余一台服务器来提升可用性，且需要人工来切换主备或者主从。这样的架构虽然简单，但存在一个主要的问题：人工操作效率低、容易出错、不能及时处理故障。因此在可用性要求更加严格的场景中，我们需要系统能够自动完成切换操作，这就是高可用集群方案。

*对称集群*，更通俗的叫法是负载均衡集群，任务分配策略比较简单，轮询和随机基本就够了。状态检测稍微复杂一些，既要检测服务器的状态，例如服务器是否宕机、网络是否正常等；同时还要检测任务的执行状态，例如任务是否卡死、是否执行时间过长等。常用的做法是任务分配器和服务器之间通过心跳来传递信息，包括服务器信息和任务信息，然后根据实际情况来确定状态判断条件。

*非对称集群*中不同服务器的角色是不同的，不同角色的服务器承担不同的职责。以Master-Slave为例，部分任务是Master服务器才能执行，部分任务是Slave服务器才能执行。

非对称集群相比负载均衡集群，设计复杂度主要体现在两个方面：

\*任务分配策略更加复杂：需要将任务划分为不同类型并分配给不同角色的集群节点。

\*角色分配策略实现比较复杂：例如，可能需要使用ZAB、Raft这类复杂的算法来实现Leader的选举。

以ZooKeeper为例：

\*任务分配器：ZooKeeper中不存在独立的任务分配器节点，每个Server都是任务分配器，Follower收到请求后会进行判断，如果是写请求就转发给Leader，如果是读请求就自己处理。

\*角色指定：ZooKeeper通过ZAB算法来选举Leader，当Leader故障后，所有的Follower节点会暂停读写操作，开始进行选举，直到新的Leader选举出来后才继续对Client提供服务。

14、异地多活设计

顾名思义，异地多活架构的关键点就是异地、多活，其中异地就是指地理位置上不同的地方，类似于“不要把鸡蛋都放在同一篮子里”；多活就是指不同地理位置上的系统都能够提供业务服务，这里的“活”是活动、活跃的意思。判断一个系统是否符合异地多活，需要满足两个标准：

\*\*正常情况下，用户无论访问哪一个地点的业务系统，都能够得到正确的业务服务。

\*\*某个地方业务异常的时候，用户访问其他地方正常的业务系统，能够得到正确的业务服务。

异地多活设计理念：

\*优先实现核心业务的异地多活架构

\*保证核心数据最终一致性

\*采用多种手段同步数据

**消息队列方式**，对于账号数据，由于账号只会创建，不会修改和删除（假设我们不提供删除功能），我们可以将账号数据通过消息队列同步到其他业务中心。

**二次读取方式，**某些情况下可能出现消息队列同步也延迟了，用户在A中心注册，然后访问B中心的业务，此时B中心本地拿不到用户的账号数据。为了解决这个问题，B中心在读取本地数据失败时，可以根据路由规则，再去A中心访问一次（这就是所谓的二次读取，第一次读取本地，本地失败后第二次读取对端），这样就能够解决异常情况下同步延迟的问题。

**存储系统同步方式**，对于密码数据，由于用户改密码频率较低，而且用户不可能在1秒内连续改多次密码，所以通过数据库的同步机制将数据复制到其他业务中心即可，用户信息数据和密码类似。

**回源读取方式，**对于登录的session数据，由于数据量很大，我们可以不同步数据；但当用户在A中心登录后，然后又在B中心登录，B中心拿到用户上传的session id后，根据路由判断session属于A中心，直接去A中心请求session数据即可；反之亦然，A中心也可以到B中心去获取session数据。

**重新生成数据方式，**对于“回源读取”场景，如果异常情况下，A中心宕机了，B中心请求session数据失败，此时就只能登录失败，让用户重新在B中心登录，生成新的session数据。

\*只保证绝大部分用户的异地多活

跨城异地多活架构设计的4个步骤：

第1步：业务分级

按照一定的标准将业务进行分级，挑选出核心的业务，只为核心业务设计异地多活，降低方案整体复杂度和实现成本。*常见的分级标准*：

\*访问量大的业务

以用户管理系统为例，业务包括登录、注册、用户信息管理，其中登录的访问量肯定是最大的。

\*核心业务

以QQ为例，QQ的主场景是聊天，QQ空间虽然也是重要业务，但和聊天相比，重要性就会低一些，如果要从聊天和QQ空间两个业务里面挑选一个做异地多活，那明显聊天要更重要（当然，此类公司如腾讯，应该是两个都实现了异地多活的）。

\*产生大量收入的业务

同样以QQ为例，聊天可能很难为腾讯带来收益，因为聊天没法插入广告；而QQ空间反而可能带来更多收益，因为QQ空间可以插入很多广告，因此如果从收入的角度来看，QQ空间做异地多活的优先级反而高于QQ聊天了。

第2步：数据分类

\*数据量

这里的数据量包括总的数据量和新增、修改、删除的量。对异地多活架构来说，新增、修改、删除的数据就是可能要同步的数据，数据量越大，同步延迟的几率越高，同步方案需要考虑相应的解决方案。

\*唯一性

唯一性指数据是否要求多个异地机房产生的同类数据必须保证唯一。例如用户ID，如果两个机房的两个不同用户注册后生成了一样的用户ID，这样业务上就出错了。

数据的唯一性影响业务的多活设计，如果数据不需要唯一，那就说明两个地方都产生同类数据是可能的；如果数据要求必须唯一，要么只能一个中心点产生数据，要么需要设计一个数据唯一生成的算法。

\*实时性

实时性指如果在A机房修改了数据，要求多长时间必须同步到B机房，实时性要求越高，对同步的要求越高，方案越复杂。

\*可丢失性

可丢失性指数据是否可以丢失。例如，写入A机房的数据还没有同步到B机房，此时A机房机器宕机会导致数据丢失，那这部分丢失的数据是否对业务会产生重大影响。

如，登录过程中产生的session数据就是可丢失的，因为用户只要重新登录就可以生成新的session；而用户ID数据是不可丢失的，丢失后用户就会失去所有和用户ID相关的数据，例如用户的好友、用户的钱等。

\*可恢复性

可恢复性指数据丢失后，是否可以通过某种手段进行恢复，如果数据可以恢复，至少说明对业务的影响不会那么大，这样可以相应地降低异地多活架构设计的复杂度。

第3步：数据同步

\*存储系统同步

这是最常用也是最简单的同步方式。例如，使用MySQL的数据主从数据同步、主主数据同步。

\*消息队列同步

采用独立消息队列进行数据同步，常见的消息队列有Kafka、ActiveMQ、RocketMQ等。

\*重复生成

数据不同步到异地机房，每个机房都可以生成数据，这个方案适合于可以重复生成的数据。

第4步：异常处理

无论数据同步方案如何设计，一旦出现极端异常的情况，总是会有部分数据出现异常的。例如，同步延迟、数据丢失、数据不一致等。异常处理就是假设在出现这些问题时，系统将采取什么措施来应对。

**处理接口级故障**

\*降级

降级指系统将某些业务或者接口的功能降低，可以是只提供部分功能，也可以是完全停掉所有功能。例如，论坛可以降级为只能看帖子，不能发帖子；也可以降级为只能看帖子和评论，不能发评论；而App的日志上传接口，可以完全停掉一段时间，这段时间内App都不能上传日志。降级的核心思想就是丢车保帅，优先保证核心业务。

\*熔断

熔断和降级是两个比较容易混淆的概念，因为单纯从名字上看好像都有禁止某个功能的意思，但其实内在含义是不同的，原因在于降级的目的是应对系统自身的故障，而熔断的目的是应对依赖的外部系统故障的情况。

熔断机制实现的关键是需要有一个统一的API调用层，由API调用层来进行采样或者统计，如果接口调用散落在代码各处就没法进行统一处理了。

\*限流

降级是从系统功能优先级的角度考虑如何应对故障，而限流则是从用户访问压力的角度来考虑如何应对故障。限流指只允许系统能够承受的访问量进来，超出系统访问能力的请求将被丢弃。

\*排队

排队实际上是限流的一个变种，限流是直接拒绝用户，排队是让用户等待一段时间，全世界最有名的排队当属12306网站排队了。排队虽然没有直接拒绝用户，但用户等了很长时间后进入系统，体验并不一定比限流好。

15、可扩展性设计

所有的可扩展性架构设计，背后的基本思想都可以总结为一个字：拆！拆，就是将原本大一统的系统拆分成多个规模小的部分，扩展时只修改其中一部分即可，无须整个系统到处都改，通过这种方式来减少改动范围，降低改动风险。

按照不同的思路来拆分软件系统，就会得到不同的架构。常见的拆分思路有如下三种。

\*面向流程拆分：将整个业务流程拆分为几个阶段，每个阶段作为一部分。

扩展时大部分情况只需要修改某一层，少部分情况可能修改关联的两层，不会出现所有层都同时要修改。例如学生信息管理系统，如果我们将存储层从MySQL扩展为同时支持MySQL和Oracle，那么只需要扩展存储层和数据层即可，展示层和业务层无须变动。

\*面向服务拆分：将系统提供的服务拆分，每个服务作为一部分。

对某个服务扩展，或者要增加新的服务时，只需要扩展相关服务即可，无须修改所有的服务。同样以学生管理系统为例，如果我们需要在注册服务中增加一种“学号注册”功能，则只需要修改“注册服务”和“登录服务”即可，“信息管理服务”和“安全设置”服务无须修改。

\*面向功能拆分：将系统提供的功能拆分，每个功能作为一部分。

对某个功能扩展，或者要增加新的功能时，只需要扩展相关功能即可，无须修改所有的服务。同样以学生管理系统为例，如果我们增加“学号注册”功能，则只需要在系统中增加一个新的功能模块，同时修改“登录功能”模块即可，其他功能都不受影响。

不同的拆分方式，将得到不同的系统架构，典型的可扩展系统架构有：

**\*\*面向流程拆分：分层架构。**

分层架构是很常见的架构模式，它也叫N层架构，通常情况下，N至少是2层。例如，C/S架构、B/S架构。常见的是3层架构（例如，MVC、MVP架构）、4层架构，5层架构的比较少见，一般是比较复杂的系统才会达到或者超过5层，比如操作系统内核架构。

分层包括C/S架构、B/S架构；MVC架构、MVP架构；逻辑分层架构

**\*\*面向服务拆分：SOA、微服务。**

SOA的全称是Service Oriented Architecture(“面向服务的架构”) SOA提出了3个关键概念:

服务

所有业务功能都是一项服务，服务就意味着要对外提供开放的能力，当其他系统需要使用这项功能时，无须定制化开发。

ESB

ESB的全称是Enterprise Service Bus，中文翻译为“企业服务总线”。从名字就可以看出，ESB参考了计算机总线的概念。计算机中的总线将各个不同的设备连接在一起，ESB将企业中各个不同的服务连接在一起。因为各个独立的服务是异构的，如果没有统一的标准，则各个异构系统对外提供的接口是各式各样的。SOA使用ESB来屏蔽异构系统对外提供各种不同的接口方式，以此来达到服务间高效的互联互通。

松耦合

松耦合的目的是减少各个服务间的依赖和互相影响。因为采用SOA架构后，各个服务是相互独立运行的，甚至都不清楚某个服务到底有多少对其他服务的依赖。如果做不到松耦合，某个服务一升级，依赖它的其他服务全部故障，这样肯定是无法满足业务需求的。



单纯从上面的对比来看，似乎微服务大大优于SOA，这也导致了很多团队在实践时不加思考地采用微服务——既不考虑团队的规模，也不考虑业务的发展，也没有考虑基础技术的支撑，只是觉得微服务很牛就赶紧来实施，以为实施了微服务后就什么问题都解决了，而一旦真正实施后才发现掉到微服务的坑里面去了。

A.服务划分过细，服务间关系复杂

服务划分过细，单个服务的复杂度确实下降了，但整个系统的复杂度却上升了，因为微服务将系统内的复杂度转移为系统间的复杂度了。

B. 服务数量太多，团队效率急剧下降

微服务的“微”字，本身就是一个陷阱，很多团队看到“微”字后，就想到必须将服务拆分得很细，有的团队人员规模是5 ~ 6个人，然而却拆分出30多个微服务，平均每个人要维护5个以上的微服务。

C. 调用链太长，性能下降

由于微服务之间都是通过HTTP或者RPC调用的，每次调用必须经过网络。一般线上的业务接口之间的调用，平均响应时间大约为50毫秒，如果用户的一起请求需要经过6次微服务调用，则性能消耗就是300毫秒，这在很多高性能业务场景下是难以满足需求的。为了支撑业务请求，可能需要大幅增加硬件，这就导致了硬件成本的大幅上升。

D. 调用链太长，问题定位困难

系统拆分为微服务后，一次用户请求需要多个微服务协同处理，任意微服务的故障都将导致整个业务失败。然而由于微服务数量较多，且故障存在扩散现象，快速定位到底是哪个微服务故障是一件复杂的事情。

E. 没有自动化支撑，无法快速交付

如果没有相应的自动化系统进行支撑，都是靠人工去操作，那么微服务不但达不到快速交付的目的，甚至还不如一个大而全的系统效率高。

F. 没有服务治理，微服务数量多了后管理混乱

信奉微服务理念的设计人员总是强调微服务的lightweight特性，并举出ESB的反例来证明微服务的优越之处。但具体实践后就会发现，随着微服务种类和数量越来越多，如果没有服务治理系统进行支撑，微服务提倡的lightweight就会变成问题。

\*\*面向功能拆分：微内核架构。

**服务粒度**

建议基于团队规模进行拆分，类似贝索斯在定义团队规模时提出的“两个披萨”理论（每个团队的人数不能多到两张披萨都不够吃的地步），分享一个我认为微服务拆分粒度的“三个火枪手”原则，即一个微服务三个人负责开发。当我们在实施微服务架构时，根据团队规模来划分微服务数量，如果业务规继续发展，团队规模扩大，我们再将已有的微服务进行拆分。

拆分方法

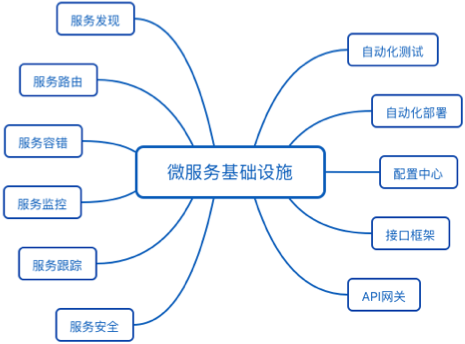
1. 基于业务逻辑拆分

2. 基于可扩展拆分

3. 基于可靠性拆分

4. 基于性能拆分

基础设施



\*\*自动化测试

微服务将原本大一统的系统拆分为多个独立运行的“微”服务，微服务之间的接口数量大大增加，并且微服务提倡快速交付，版本周期短，版本更新频繁。如果每次更新都靠人工回归整个系统，则工作量大，效率低下，达不到“快速交付”的目的，因此必须通过自动化测试系统来完成绝大部分测试回归的工作。

\*\*自动化部署

相比大一统的系统，微服务需要部署的节点增加了几倍甚至十几倍，微服务部署的频率也会大幅提升（例如，我们的业务系统70%的工作日都有部署操作），综合计算下来，微服务部署的次数是大一统系统部署次数的几十倍。这么大量的部署操作，如果继续采用人工手工处理，需要投入大量的人力，且容易出错，因此需要自动化部署的系统来完成部署操作。

\*\*配置中心

微服务的节点数量非常多，通过人工登录每台机器手工修改，效率低，容易出错。特别是在部署或者排障时，需要快速增删改查配置，人工操作的方式显然是不行的。除此以外，有的运行期配置需要动态修改并且所有节点即时生效，人工操作是无法做到的。综合上面的分析，微服务需要一个统一的配置中心来管理所有微服务节点的配置。

\*\*接口框架

微服务提倡轻量级的通信方式，一般采用HTTP/REST或者RPC方式统一接口协议。但在实践过程中，光统一接口协议还不够，还需要统一接口传递的数据格式。

\*\*API网关

系统拆分为微服务后，内部的微服务之间是互联互通的，相互之间的访问都是点对点的。如果外部系统想调用系统的某个功能，也采取点对点的方式，则外部系统会非常“头大”。因为在外部系统看来，它不需要也没办法理解这么多微服务的职责分工和边界，它只会关注它需要的能力，而不会关注这个能力应该由哪个微服务提供。

\*\*服务发现

微服务种类和数量很多，如果这些信息全部通过手工配置的方式写入各个微服务节点，首先配置工作量很大，配置文件可能要配几百上千行，几十个节点加起来后配置项就是几万几十万行了，人工维护这么大数量的配置项是一项灾难；其次是微服务节点经常变化，可能是由于扩容导致节点增加，也可能是故障处理时隔离掉一部分节点，还可能是采用灰度升级，先将一部分节点升级到新版本，然后让新老版本同时运行。不管哪种情况，我们都希望节点的变化能够及时同步到所有其他依赖的微服务。如果采用手工配置，是不可能做到实时更改生效的。因此，需要一套服务发现的系统来支撑微服务的自动注册和发现。

\*\*服务路由

有了服务发现后，微服务之间能够方便地获取相关配置信息，但具体进行某次调用请求时，我们还需要从所有符合条件的可用微服务节点中挑选出一个具体的节点发起请求，这就是服务路由需要完成的功能。

\*\*服务容错

系统拆分为微服务后，单个微服务故障的概率变小，故障影响范围也减少，但是微服务的节点数量大大增加。从整体上来看，系统中某个微服务出故障的概率会大大增加。

\*\*服务监控

系统拆分为微服务后，节点数量大大增加，导致需要监控的机器、网络、进程、接口调用数等监控对象的数量大大增加；同时，一旦发生故障，我们需要快速根据各类信息来定位故障。这两个目标如果靠人力去完成是不现实的。

\*\*服务跟踪

服务监控可以做到微服务节点级的监控和信息收集，但如果我们需要跟踪某一个请求在微服务中的完整路径，服务监控是难以实现的。因为如果每个服务的完整请求链信息都实时发送给服务监控系统，数据量会大到无法处理。

\*\*服务安全

系统拆分为微服务后，数据分散在各个微服务节点上。从系统连接的角度来说，任意微服务都可以访问所有其他微服务节点；但从业务的角度来说，部分敏感数据或者操作，只能部分微服务可以访问，而不是所有的微服务都可以访问，因此需要设计服务安全机制来保证业务和数据的安全性。

按照下面优先级来搭建基础设施：

1\*.服务发现、服务路由、服务容错：这是最基本的微服务基础设施。

2\*.接口框架、API网关：主要是为了提升开发效率，接口框架是提升内部服务的开发效率，API网关是为了提升与外部服务对接的效率。

3\*.自动化部署、自动化测试、配置中心：主要是为了提升测试和运维效率。

4\*.服务监控、服务跟踪、服务安全：主要是为了进一步提升运维效率。

16、微内核架构

设计关键点

微内核的核心系统设计的关键技术有：插件管理、插件连接和插件通信。

\*.插件管理

核心系统需要知道当前有哪些插件可用，如何加载这些插件，什么时候加载插件。常见的实现方法是插件注册表机制。

核心系统提供插件注册表（可以是配置文件，也可以是代码，还可以是数据库），插件注册表含有每个插件模块的信息，包括它的名字、位置、加载时机（启动就加载，还是按需加载）等。

\*.插件连接

插件连接指插件如何连接到核心系统。通常来说，核心系统必须制定插件和核心系统的连接规范，然后插件按照规范实现，核心系统按照规范加载即可。

常见的连接机制有OSGi（Eclipse使用）、消息模式、依赖注入（Spring使用），甚至使用分布式的协议都是可以的，比如RPC或者HTTP Web的方式。

\*.插件通信

插件通信指插件间的通信。虽然设计的时候插件间是完全解耦的，但实际业务运行过程中，必然会出现某个业务流程需要多个插件协作，这就要求两个插件间进行通信。由于插件之间没有直接联系，通信必须通过核心系统，因此核心系统需要提供插件通信机制。这种情况和计算机类似，计算机的CPU、硬盘、内存、网卡是独立设计的配件，但计算机运行过程中，CPU和内存、内存和硬盘肯定是有通信的，计算机通过主板上的总线提供了这些组件之间的通信功能。微内核的核心系统也必须提供类似的通信机制，各个插件之间才能进行正常的通信。

17、技术演进的方向

架构师应该如何判断技术演进的方向，关于这个问题的答案，基本上可以分为几个典型的派别：

\*.潮流派

潮流派的典型特征就是对于新技术特别热衷，紧跟技术潮流，当有新的技术出现时，迫切想将新的技术应用到自己的产品中。

例如：

NoSQL很火，咱们要大规模地切换为NoSQL。

大数据好牛呀，将我们的MySQL切换为Hadoop吧。

Node.js使得JavaScript统一前后端，这样非常有助于开展工作。

首先，新技术需要时间成熟，如果刚出来就用，此时新技术还不怎么成熟，实际应用中很可能遇到各种“坑”，自己成了实验小白鼠。

其次，新技术需要学习，需要花费一定的时间去掌握，这个也是较大的成本；如果等到掌握了技术后又发现不适用，则是一种较大的人力浪费。

\*.保守派

保守派的典型特征和潮流派正好相反，对于新技术抱有很强的戒备心，稳定压倒一切，已经掌握了某种技术，就一直用这种技术打天下。就像有句俗语说的，“如果你手里有一把锤子，那么所有的问题都变成了钉子”，保守派就是拿着一把锤子解决所有的问题。

例如：

MySQL咱们用了这么久了，很熟悉了，业务用MySQL，数据分析也用MySQL，报表还用MySQL吧。

Java语言我们都很熟，业务用Java，工具用Java，平台也用Java。

保守派的主要问题是不能享受新技术带来的收益，因为新技术很多都是为了解决以前技术存在的固有缺陷。就像汽车取代马车一样，不是量变而是质变，带来的收益不是线性变化的，而是爆发式变化的。如果无视技术的发展，形象一点说就是有了拖拉机，你还偏偏要用牛车。

\*.跟风派

跟风派与潮流派不同，这里的跟风派不是指跟着技术潮流，而是指跟着竞争对手的步子走。

简单来说，判断技术的发展就看竞争对手，竞争对手用了咱们就用，竞争对手没用咱们就等等看。

例如：

这项技术腾讯用了吗？腾讯用了我们就用。

阿里用了Hadoop，他们都在用，肯定是好东西，咱们也要尽快用起来，以提高咱们的竞争力。

Google都用了Docker，咱们也用吧。

可能很多人都会认为，跟风派与“潮流派”和“保守派”相比，是最有效的策略，既不会承担“潮流派”的风险，也不会遭受“保守派”的损失，花费的资源也少，简直就是一举多得。

**技术演进动力**

\*对于产品类业务，答案看起来很明显：技术创新推动业务发展！

\*对于“服务”类的业务，答案和产品类业务正好相反：业务发展推动技术的发展！

**技术演进的模式**

架构师具体应该按照什么标准来判断呢？答案就是基于业务发展阶段进行判断。

互联网业务发展一般分为几个时期：初创期、发展期、竞争期、成熟期。不同时期的差别主要体现在两个方面：复杂性、用户规模。

\*.初创期

互联网业务刚开始一般都是一个创新的业务点，这个业务点的重点不在于“完善”，而在于“创新”，只有创新才能吸引用户；而且因为其“新”的特点，其实一开始是不可能很完善的。只有随着越来越多的用户的使用，通过快速迭代试错、用户的反馈等手段，不断地在实践中去完善，才能继续创新。

初创期的业务对技术就一个要求：“快”，但这个时候却又是创业团队最弱小的时期，可能就几个技术人员，所以这个时候十八般武艺都需要用上：能买就买，有开源的就用开源的。

\*.发展期

当业务推出后经过市场验证如果是可行的，则吸引的用户就会越来越多，此时原来不完善的业务就进入了一个快速发展的时期。业务快速发展时期的主要目的是将原来不完善的业务逐渐完善，因此会有越来越多的新功能不断地加入到系统中。对于绝大部分技术团队来说，这个阶段技术的核心工作是快速地实现各种需求，只有这样才能满足业务发展的需要。

\*.竞争期

当业务继续发展，已经形成一定规模后，一定会有竞争对手开始加入行业来竞争，毕竟谁都想分一块蛋糕，甚至有可能一不小心还会成为下一个BAT。当竞争对手加入后，大家互相学习和模仿，业务更加完善，也不断有新的业务创新出来，而且由于竞争的压力，对技术的要求是更上一层楼了。

针对这个时期业务变化带来的问题，技术工作主要的解决手段有：

\*\*平台化

目的在于解决“重复造轮子”的问题。

存储平台化：淘宝的TFS、京东JFS。

数据库平台化：百度的DBProxy、淘宝TDDL。

缓存平台化：Twitter的Twemproxy，豆瓣的BeansDB、腾讯TTC。

\*\*服务化

目的在于解决“系统交互”的问题，常见的做法是通过消息队列来完成系统间的异步通知，通过服务框架来完成系统间的同步调用。

消息队列：淘宝的Notify、MetaQ，开源的Kafka、ActiveMQ等。

服务框架：Facebook的thrift、当当网的Dubbox、淘宝的HSF等。

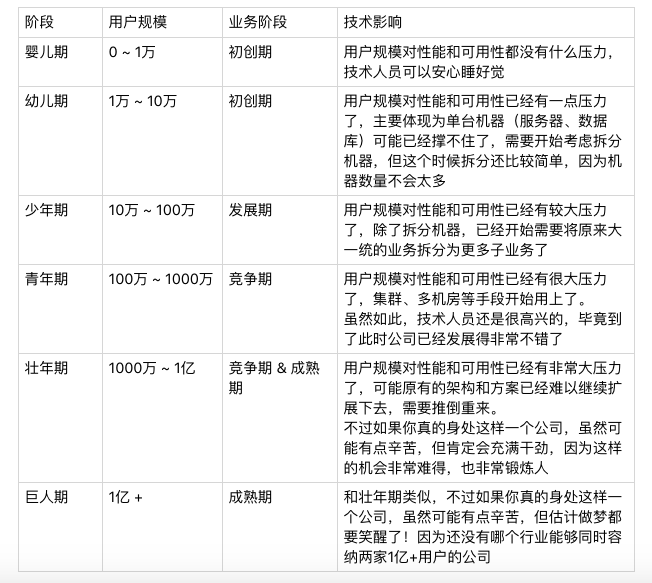
\*.成熟期

当企业熬过竞争期，成为了行业的领头羊，或者整个行业整体上已经处于比较成熟的阶段，市场地位已经比较牢固后，业务创新的机会已经不大，竞争压力也没有那么激烈，此时求快求新已经没有很大空间，业务上开始转向为“求精”：我们的响应时间是否比竞争对手快？我们的用户体验是否比竞争对手好？我们的成本是否比竞争对手低……

**用户规模**

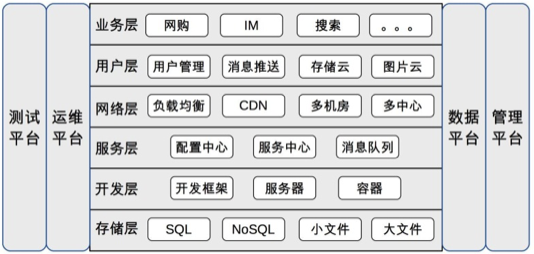
互联网业务驱动技术发展的两大主要因素是复杂性和用户规模，而这两个因素的本质其实都是“量变带来质变”。

究竟用户规模发展到什么阶段才会由量变带来质变，虽然不同的业务有所差别，但基本上可以按照下面这个模型去衡量。



18、互联网架构模板

互联网的标准技术架构如下图所示，这张图基本上涵盖了互联网技术公司的大部分技术点，不同的公司只是在具体的技术实现上稍有差异，但不会跳出这个框架的范畴。



\*数据库

NoSQL不是No SQL，而是Not Only SQL，即NoSQL是SQL的补充。

NoSQL发展到一定规模后，通常都会在NoSQL集群的基础之上再实现统一存储平台，统一存储平台主要实现这几个功能：

资源动态按需动态分配：例如同一台Memcache服务器，可以根据内存利用率，分配给多个业务使用。

资源自动化管理：例如新业务只需要申请多少Memcache缓存空间就可以了，无需关注具体是哪些Memcache服务器在为自己提供服务。

故障自动化处理：例如某台Memcache服务器挂掉后，有另外一台备份Memcache服务器能立刻接管缓存请求，不会导致丢失很多缓存数据。

\*开发层技术

\*\*开发框架

对于框架的选择，有一个总的原则：优选成熟的框架，避免盲目追逐新技术！

首先，成熟的框架资料文档齐备，各种坑基本上都有人踩过了，遇到问题很容易通过搜索来解决。

其次，成熟的框架受众更广，招聘时更加容易招到合适的人才。

第三，成熟的框架更加稳定，不会出现大的变动，适合长期发展。

\*\*Web服务器

开发框架只是负责完成业务功能的开发，真正能够运行起来给用户提供服务，还需要服务器配合。

例如，Java的有Tomcat、JBoss、Resin等，PHP/Python的用Nginx，当然最保险的就是用Apache了，什么语言都支持。

\*\*容器

容器是最近几年才开始火起来的，其中以Docker为代表，在BAT级别的公司已经有较多的应用。

千万不要以为Docker只是一个虚拟化或者容器技术，它将在很大程度上改变目前的技术形势：

A运维方式会发生革命性的变化：Docker启动快，几乎不占资源，随时启动和停止，基于Docker打造自动化运维、智能化运维将成为主流方式。

B设计模式会发生本质上的变化：启动一个新的容器实例代价如此低，将鼓励设计思路朝“微服务”的方向发展。

\*服务层技术

服务层的主要目标其实就是为了降低系统间相互关联的复杂度。

\*\*配置中心

实现配置中心主要就是为了解决上面这些问题，将配置中心做成通用的系统的好处有：

A集中配置多个系统，操作效率高。

B所有配置都在一个集中的地方，检查方便，协作效率高。

C配置中心可以实现程序化的规则检查，避免常见的错误。比如说检查最小值、最大值、是否IP地址、是否URL地址，都可以用正则表达式完成。

D配置中心相当于备份了系统的配置，当某些情况下需要搭建新的环境时，能够快速搭建环境和恢复业务。

\*\*服务中心

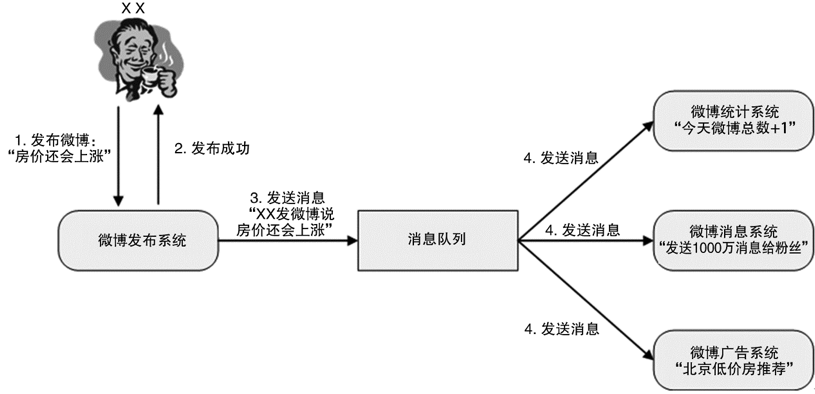
A服务名字系统（Service Name System），DNS，即Domain Name System。没错，两者的性质是基本类似的。

B服务总线系统（Service Bus System），相当于计算机的总线。没两者的本质也是基本类似的。

\*\*消息队列

互联网业务的一个特点是“快”，这就要求很多业务处理采用异步的方式。

消息队列就是为了实现这种跨系统异步通知的中间件系统。消息队列既可以“一对一”通知，也可以“一对多”广播。



对比前面的蜘蛛网架构，可以清晰地看出引入消息队列系统后的效果：

\*1.整体结构从网状结构变为线性结构，结构清晰。

\*2.消息生产和消息消费解耦，实现简单。

\*3.增加新的消息消费者，消息生产者完全不需要任何改动，扩展方便。

\*4.消息队列系统可以做高可用、高性能，避免各业务子系统各自独立做一套，减轻工作量。

\*5.业务子系统只需要聚焦业务即可，实现简单。

消息队列系统基本功能的实现比较简单，但要做到高性能、高可用、消息时序性、消息事务性则比较难。业界已经有很多成熟的开源实现方案，如果要求不高，基本上拿来用即可，例如，RocketMQ、Kafka、ActiveMQ等。但如果业务对消息的可靠性、时序、事务性要求较高时，则要深入研究这些开源方案，否则很容易踩坑。

\*负载均衡

负载均衡就是将请求均衡地分配到多个系统上。使用负载均衡的原因也很简单：每个系统的处理能力是有限的，为了应对大容量的访问，必须使用多个系统。例如，一台32核64GB内存的机器，性能测试数据显示每秒处理Hello World的HTTP请求不超过2万，实际业务机器处理HTTP请求每秒可能才几百QPS，而互联网业务并发超过1万是比较常见的，遇到双十一、过年发红包这些极端场景，每秒可以达到几十万的请求。

\*DNS负载均衡

HTTP-DNS的优缺点有：

灵活：HTTP-DNS可以根据业务需求灵活的设置各种策略。

可控：HTTP-DNS是自己开发的系统，IP更新、策略更新等无需依赖外部服务商。

及时：HTTP-DNS不受传统DNS缓存的影响，可以非常快地更新数据、隔离故障。

开发成本高：没有通用的解决方案，需要自己开发。

侵入性：需要App基于HTTP-DNS进行改造。

\*Nginx 、LVS 、F5

DNS用于实现地理级别的负载均衡，而Nginx、LVS、F5用于同一地点内机器级别的负载均衡。其中Nginx是软件的7层负载均衡，LVS是内核的4层负载均衡，F5是硬件的4层负载均衡。

\*多机房

从架构上来说，单机房就是一个全局的网络单点，在发生比较大的故障或者灾害时，单机房难以保证业务的高可用。

\*多中心

多中心必须以多机房为前提，但从设计的角度来看，多中心相比多机房是本质上的飞越，难度也高出一个等级。

\*用户层技术

互联网业务的一个典型特征就是通过互联网将众多分散的用户连接起来，因此用户管理是互联网业务必不可少的一部分。

稍微大一点的互联网业务，肯定会涉及多个子系统，这些子系统不可能每个都管理这么庞大的用户，由此引申出用户管理的第一个目标：

**单点登录（SSO）**，又叫统一登录。单点登录的技术实现手段较多，例如cookie、JSONP、token等。

**消息推送**根据不同的途径，分为短信、邮件、站内信、App推送。除了App，不同的途径基本上调用不同的API即可完成，技术上没有什么难度。例如，短信需要依赖运营商的短信接口，邮件需要依赖邮件服务商的邮件接口，站内信是系统提供的消息通知功能。

**存储云、图片云**

互联网业务场景中，用户会上传多种类型的文件数据，例如微信用户发朋友圈时上传图片，微博用户发微博时上传图片、视频，优酷用户上传视频，淘宝卖家上传商品图片等，这些文件具备几个典型特点：

A数据量大：用户基数大，用户上传行为频繁，例如2016年的时候微信朋友圈每天上传图片就达到了10亿张。

B文件体积小：大部分图片是几百KB到几MB，短视频播放时间也是在几分钟内。

C访问有时效性：大部分文件是刚上传的时候访问最多，随着时间的推移访问量越来越小。

\*业务层技术

互联网的业务千差万别，不同的业务分解下来有不同的系统，所以业务层没有办法提炼一些公共的系统或者组件。抛开业务上的差异，各个互联网业务发展最终面临的问题都是类似的：业务复杂度越来越高。也就是说，业务层面对的主要技术挑战是“复杂度”。

模拟的电商系统经历了3个发展阶段：

第一阶段：所有功能都在1个系统里面。

第二阶段：将商品和订单拆分到2个子系统里面。

第三阶段：商品子系统和订单子系统分别拆分成了更小的6个子系统。

\*运维平台

运维平台核心的职责分为四大块：配置、部署、监控、应急，每个职责对应系统生命周期的一个阶段，如下图所示。

\*\*配置：主要负责资源的管理。例如，机器管理、IP地址管理、虚拟机管理等。

\*\*部署：主要负责将系统发布到线上。例如，包管理、灰度发布管理、回滚等。

\*\*监控：主要负责收集系统上线运行后的相关数据并进行监控，以便及时发现问题。

\*\*应急：主要负责系统出故障后的处理。例如，停止程序、下线故障机器、切换IP等。

运维平台的核心设计要素是“四化”：标准化、平台化、自动化、可视化。

\*测试平台

测试平台的核心目的是提升测试效率，从而提升产品质量，其设计关键就是自动化。传统的测试方式是测试人员手工执行测试用例，测试效率低，重复的工作多。通过测试平台提供的自动化能力，测试用例能够重复执行，无须人工参与，大大提升了测试效率。

为了达到“自动化”的目标，测试平台的基本架构有以下几个方面：

\*\*用例管理：测试自动化的主要手段就是通过脚本或者代码来进行测试，例如单元测试用例是代码、接口测试用例可以用Python来写、可靠性测试用例可以用Shell来写。

\*\*资源管理：测试用例要放到具体的运行环境中才能真正执行，运行环境包括硬件（服务器、手机、平板电脑等）、软件（操作系统、数据库、Java虚拟机等）、业务系统（被测试的系统）。

\*\*任务管理：任务管理的主要职责是将测试用例分配到具体的资源上执行，跟踪任务的执行情况。任务管理是测试平台设计的核心，它将测试平台的各个部分串联起来从而完成自动化测试。

\*\*数据管理：测试任务执行完成后，需要记录各种相关的数据（例如，执行时间、执行结果、用例执行期间的CPU、内存占用情况等），这些数据具备下面这些作用：

A展现当前用例的执行情况。

B作为历史数据，方便后续的测试与历史数据进行对比，从而发现明显的变化趋势。例如，某个版本后单元测试覆盖率从90%下降到70%。

C作为大数据的一部分，可以基于测试的任务数据进行一些数据挖掘。例如，某个业务一年执行了10000个用例测试，另外一个业务只执行了1000个用例测试，两个业务规模和复杂度差不多，为何差异这么大？

\*数据平台

A数据管理

数据管理包含数据采集、数据存储、数据访问和数据安全四个核心职责，是数据平台的基础功能。

\*\*数据采集：从业务系统搜集各类数据。例如，日志、用户行为、业务数据等，将这些数据传送到数据平台。

\*\*数据存储：将从业务系统采集的数据存储到数据平台，用于后续数据分析。

\*\*数据访问：负责对外提供各种协议用于读写数据。例如，SQL、Hive、Key-Value等读写协议。

\*\*数据安全：通常情况下数据平台都是多个业务共享的，部分业务敏感数据需要加以保护，防止被其他业务读取甚至修改，因此需要设计数据安全策略来保护数据。

B数据分析

数据分析包括数据统计、数据挖掘、机器学习、深度学习等几个细分领域。

\*\*数据统计：根据原始数据统计出相关的总览数据。例如，PV、UV、交易额等。

\*\*数据挖掘：数据挖掘这个概念本身含义可以很广，为了与机器学习和深度学习区分开，这里的数据挖掘主要是指传统的数据挖掘方式。例如，有经验的数据分析人员基于数据仓库构建一系列规则来对数据进行分析从而发现一些隐含的规律、现象、问题等，经典的数据挖掘案例就是沃尔玛的啤酒与尿布的关联关系的发现。

\*\*机器学习、深度学习：机器学习和深度学习属于数据挖掘的一种具体实现方式，由于其实现方式与传统的数据挖掘方式差异较大，因此数据平台在实现机器学习和深度学习时，需要针对机器学习和深度学习独立进行设计。

C数据应用

数据应用很广泛，既包括在线业务，也包括离线业务。例如，推荐、广告等属于在线应用，报表、欺诈检测、异常检测等属于离线应用。

\*管理平台

管理平台的核心职责就是权限管理，无论是业务系统（例如，淘宝网）、中间件系统（例如，消息队列Kafka），还是平台系统（例如，运维平台），都需要进行管理。如果每个系统都自己来实现权限管理，效率太低，重复工作很多，因此需要统一的管理平台来管理所有的系统的权限。

A身份认证

确定当前的操作人员身份，防止非法人员进入系统。例如，不允许匿名用户进入系统。为了避免每个系统都自己来管理用户，通常情况下都会使用企业账号来做统一认证和登录。

B权限控制

根据操作人员的身份确定操作权限，防止未经授权的人员进行操作。例如，不允许研发人员进入财务系统查看别人的工资。

19、架构重构

第一式：有的放矢，需要架构师透过问题表象看到问题本质，找出真正需要通过架构重构解决的核心问题，而不是想着通过一次重构解决所有问题。

架构设计三原则中的演化原则中，系统架构是不断演化的，少部分架构演化可能需要推倒重来进行重写，但绝大部分的架构演化都是通过架构重构来实现的。相比全新的架构设计来说，架构重构对架构师的要求更高，主要体现在：

A．业务已经上线，不能停下来

架构重构时，业务已经上线运行了，重构既需要尽量保证业务继续往前发展，又要完成架构调整。

B．关联方众多，牵一发动全身

架构重构涉及的业务关联方很多，不同关联方的资源投入程度、业务发展速度、对架构痛点的敏感度等有很大差异，如何尽量减少对关联方的影响，或者协调关联方统一行动，是一项很大的挑战；

C．旧架构的约束

架构重构需要在旧的架构基础上进行，这是一个很强的约束，会限制架构师的技术选择范围；即使是我们决定推倒到重来，完全抛弃旧的架构而去设计新的架构，新架构也会受到旧架构的约束和影响，因为业务在旧架构上产生的数据是不能推倒重来的，新架构必须考虑如何将旧架构产生的数据转换过来。

第二式：合纵连横

架构重构是大动作，持续时间比较长，而且会占用一定的研发资源，包括开发和测试，因此不可避免地会影响业务功能的开发。因此，要想真正推动一个架构重构项目启动，需要花费大量的精力进行游说和沟通。

在沟通时还经常遇到的一个问题是凭感觉而不是凭数据说话。比如技术人员说“系统耦合导致我们的开发效率很低”，但是没有数据，也没有样例，单纯这样说，其他人员很难有直观的印象。

有的重构还需要和其他相关或者配合的系统的沟通协调。由于大家都是做技术的，有比较多的共同语言，所以这部分的沟通协调其实相对来说要容易一些，但也不是说想推动就能推动的，主要的阻力来自“这对我有什么好处”和“这部分我这边现在不急”。

如果真的出现了对公司或者部门有利，对某个小组不利的情况，那可能需要协调更高层级的管理者才能够推动，平级推动是比较难的。

如果对方真的是因为有其他更重要的业务，此时勉为其难也不好，还是那句话：换位思考！因为大部分重构的系统并不是到了火烧眉毛非常紧急的时候才开始启动的，而是有一定前瞻性的规划，如果对方真的有其他更加重要的事情，采取等待的策略也未尝不可，但要明确正式启动的时间。例如，3个月后开始、6月份开始，千万不能说“以后”“等不忙的时候”这种无法明确的时间点。

除了计划上灵活一点，方案上也可以灵活一点：我们可以先不做这个系统相关的重构，先把其他需要重构的做完。因为大部分需要重构的系统，需要做的事情很多，分阶段处理，在风险规避、计划安排等方面更加灵活可控。

第三式：运筹帷幄

从最初的100个问题列表，挑选出其中50个是需要在架构重构中解决的，其中一些是基础能力建设或者准备工作，而另外一些就是架构重构的核心工作。

\*\*第一个原因是没有区分问题的优先级，所有问题都一视同仁，没有集中有限资源去解决最重要或者最关键的问题，导致最后做了大半年，回头一看好像做了很多事情，但没取得什么阶段性的成果。

\*\*第二个原因是没有将问题分类，导致相似问题没有统筹考虑，方案可能出现反复，效率不高。

\*\*第三个原因是会迫于业务版本的压力，专门挑容易做的实施，到了稍微难一点的问题的时候，就因为复杂度和投入等原因被搁置，达不到重构的真正目的。

重构的做法，其实就是“分段实施”，将要解决的问题根据优先级、重要性、实施难度等划分为不同的阶段，每个阶段聚焦于一个整体的目标，集中精力和资源解决一类问题。这样做有几个好处：

A每个阶段都有明确目标，做完之后效果明显，团队信心足，后续推进更加容易。

B每个阶段的工作量不会太大，可以和业务并行。

C每个阶段的改动不会太大，降低了总体风险。

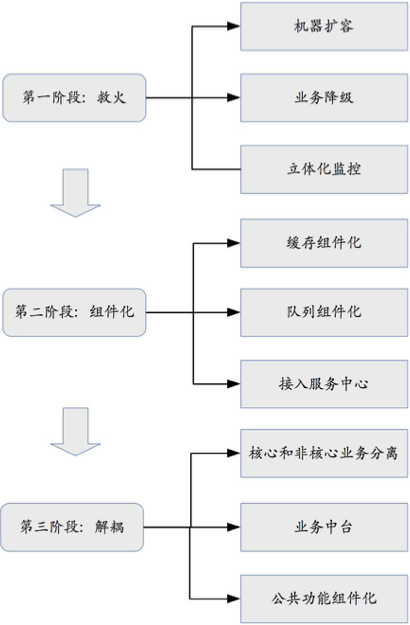
具体如何制定“分段实施”的策略：

\*\*优先级排序 ：将明显且又比较紧急的事项优先落地，解决目前遇到的主要问题。

\*\*问题分类：将问题按照性质分类，每个阶段集中解决一类问题。

\*\*先易后难：这点与很多人的直觉不太一样，有的人认为应该先攻克最难的问题，所谓“擒贼先擒王”，解决最难的问题后其他问题就不在话下。

\*\*循序渐进：按照前3个步骤划分了架构重构的实施阶段后，就需要评估每个阶段所需要耗费的时间，很可能会出现有的阶段耗时可能只要1个月，而有的却需要6个月，虽然这可能确实是客观事实，但通常情况下，按照固定的步骤和节奏，更有利于项目推进。我的经验是每个阶段最少1个月，最长不要超过3个月，如果评估超过3个月的，那就再拆分为更多阶段。就像X项目，我们先划分了阶段，每个阶段又分了任务子集，当任务子集比较小的时候，多个任务子集可以并行；当任务子集比较大的时候，就当成一个独立的里程碑推进。



20、如何选择开源项目

架构师在选择开源项目时，一个头疼的问题就是相似的开源项目较多，而且后面的总是要宣称比前面的更加优秀。有的架构师在选择时有点无所适从，总是会担心选择了A项目而错过了B项目。

\*聚焦是否满足业务

\*聚焦是否成熟

\*聚焦运维能力

**如何使用开源项目：**

\*.深入研究，仔细测试

MySQL的案例

很多团队最初使用MySQL时，也没有怎么研究过，经常有业务部门抱怨MySQL太慢了。但经过定位，发现最关键的几个参数（例如，innodb\\_buffer\\_pool\\_size、sync\\_binlog、innodb\\_log\\_file\\_size等）都没有配置或者配置错误，性能当然会慢。

\*.小心应用，灰度发布

\*.做好应急，以防万一

基于LevelDB的原理，自己做了一套缓存框架支持存储、备份、集群的功能，后来又在这个框架的基础上增加了跨机房同步的功能，很大程度上提升了业务的可用性水平。如果完全采用开源项目，等开源项目来实现，是不可能这么快速的，甚至开源项目完全就不支持我们的需求。

21、APP架构

**Web App**

最早的App有很多采用这种架构，大多数尝试性的业务，一开始也是这样的架构。Web App架构又叫包壳架构，简单来说就是在Web的业务上包装一个App的壳，业务逻辑完全还是Web实现，App壳完成安装的功能，让用户看起来像是在使用App，实际上和用浏览器访问PC网站没有太大差别。

**原生App**

Web App虽然解决了“快速开发”和“低成本”两个复杂度问题，但随着业务的发展，Web App的劣势逐渐成为了主要的复杂度问题，主要体现在：

\*移动设备的发展速度远远超过Web技术的发展速度，因此Web App的体验相比原生App的体验，差距越来越明显。

\*移动互联网飞速发展，趋势越来越明显，App承载的业务逻辑也越来越复杂，进一步加剧了Web App的体验问题。

\*移动设备在用户体验方面有很多优化和改进，而Web App无法利用这些技术优势，只有原生App才能够利用这些技术优势。

**组件化 & 容器化**

Hybrid App能够较好的平衡“用户体验”和“快速开发”两个复杂度问题（注意是“平衡”，不是“同时解决”），但对于一些超级App来说，随着业务规模越来越大、业务越来越复杂，虽然在用户看来可能是一个App，但事实上承载了几十上百个业务。

对于组件化和容器化并没有非常严格的定义，我理解两者在规范、拆分、团队协作方面都是一样的，区别在于发布方式，组件化采用的是静态发布，即所有的组件各自独自开发测试，然后跟随App的某个版本统一上线；容器化采用的是动态发布，即容器可以动态加载组件，组件准备好了直接发布，容器会动态更新组件，无需等待某个版本才能上线。

**跨平台App**

前面我介绍的各种App架构，除了Web App外，其他都面临着同一个问题：跨平台需要重复开发。同一个功能和业务，Android开发一遍，iOS也要开发一遍，这里其实存在人力投入的问题，违背了架构设计中的“简单原则”。站在企业的角度来讲，当然希望能够减少人力投入成本（虽然我站在程序员的角度来讲是不希望程序员被减少的），因此最近几年各种跨平台方案不断涌现，比较知名的有Facebook的React Native、阿里的Weex、Google的Flutter。

**22、架构设计模板**

备选方案模板

1.需求介绍

随着前浪微博业务的不断发展，业务上拆分的子系统越来越多，目前系统间的调用都是同步调用，由此带来几个明显的系统问题：

\*性能问题：当用户发布了一条微博后，微博发布子系统需要同步调用“统计子系统”“审核子系统”“奖励子系统”等共8个子系统，性能很低。

\*耦合问题：当新增一个子系统时，例如如果要增加“广告子系统”，那么广告子系统需要开发新的接口给微博发布子系统调用。

\*效率问题：每个子系统提供的接口参数和实现都有一些细微的差别，导致每次都需要重新设计接口和联调接口，开发团队和测试团队花费了许多重复工作量。

基于以上背景，我们需要引入消息队列进行系统解耦，将目前的同步调用改为异步通知。

2.需求分析

5W指Who、When、What、Why、Where

\*\*Who：需求利益干系人，包括开发者、使用者、购买者、决策者等。

\*\*When：需求使用时间，包括季节、时间、里程碑等。

\*\*What：需求的产出是什么，包括系统、数据、文件、开发库、平台等。

\*\*Where：需求的应用场景，包括国家、地点、环境等，例如测试平台只会在测试环境使用。

\*\*Why：需求需要解决的问题，通常和需求背景相关\]

消息队列的5W分析如下：

\*\*Who：消息队列系统主要是业务子系统来使用，子系统发送消息或者接收消息。

\*\*When：当子系统需要发送异步通知的时候，需要使用消息队列系统。

\*\*What：需要开发消息队列系统。

\*\*Where：开发环境、测试环境、生产环境都需要部署。

\*\*Why：消息队列系统将子系统解耦，将同步调用改为异步通知。

1H

这里的How不是设计方案也不是架构方案，而是关键业务流程。消息队列系统这部分内容很简单，但有的业务系统1H就是具体的用例了，有兴趣的同学可以尝试写写ATM机取款的业务流程。

消息队列有两大核心功能：

\*\*业务子系统发送消息给消息队列。

\*\*业务子系统从消息队列获取消息。

8C指的是8个约束和限制，即Constraints，包括性能Performance、成本Cost、时间Time、可靠性Reliability、安全性Security、合规性Compliance、技术性Technology、兼容性Compatibility

注：需求中涉及的性能、成本、可靠性等仅仅是利益关联方提出的诉求，不一定准确；如果经过分析有的约束没有必要，或成本太高、难度太大，这些约束是可以调整的。

\*\*性能：需要达到Kafka的性能水平。

\*\*成本：参考XX公司的设计方案，不超过10台服务器。

\*\*时间：期望3个月内上线第一个版本，在两个业务尝试使用。

\*\*可靠性：按照业务的要求，消息队列系统的可靠性需要达到99.99%。

\*\*安全性：消息队列系统仅在生产环境内网使用，无需考虑网络安全；如消息中有敏感信息，消息发送方需要自行进行加密，消息队列系统本身不考虑通用的加密。

\*\*合规性：消息队列系统需要按照公司目前的DevOps规范进行开发。

\*\*技术性：目前团队主要研发人员是Java，最好用Java开发。

\*\*兼容性：之前没有类似系统，无需考虑兼容性。

3.复杂度分析

分析需求的复杂度，复杂度常见的有高可用、高性能、可扩展等，具体分析方法请参考专栏前面的内容

**高可用**

对于微博子系统来说，如果消息丢了，导致没有审核，然后触犯了国家法律法规，则是非常严重的事情；对于等级子系统来说，如果用户达到相应等级后，系统没有给他奖品和专属服务，则VIP用户会很不满意，导致用户流失从而损失收入，虽然也比较关键，但没有审核子系统丢消息那么严重。

综合来看，消息队列需要高可用性，包括消息写入、消息存储、消息读取都需要保证高可用性。

**高性能**

前浪微博系统用户每天发送1000万条微博，那么微博子系统一天会产生1000万条消息，平均一条消息有10个子系统读取，那么其他子系统读取的消息大约是1亿次。将数据按照秒来计算，一天内平均每秒写入消息数为115条，每秒读取的消息数是1150条；再考虑系统的读写并不是完全平均的，设计的目标应该以峰值来计算。峰值一般取平均值的3倍，那么消息队列系统的TPS是345，QPS是3450，考虑一定的性能余量。由于现在的基数较低，为了预留一定的系统容量应对后续业务的发展，我们将设计目标设定为峰值的4倍，因此最终的性能要求是：TPS为1380，QPS为13800。TPS为1380并不高，但QPS为13800已经比较高了，因此高性能读取是复杂度之一。

**可扩展**

消息队列的功能很明确，基本无须扩展，因此可扩展性不是这个消息队列的关键复杂度。

4.备选方案

备选方案1：直接引入开源Kafka

备选方案2：集群 + MySQL存储

备选方案3：集群 + 自研存储

5.备选方案评估

**架构设计模板**

\*1.总体方案

总体方案需要从整体上描述方案的结构，其核心内容就是架构图，以及针对架构图的描述，包括模块或者子系统的职责描述、核心流程\

\*2.架构总览

架构总览给出架构图以及架构的描述

架构关键设计点：

\*\*采用数据分散集群的架构，集群中的服务器进行分组，每个分组存储一部分消息数据。

\*\*每个分组包含一台主MySQL和一台备MySQL，分组内主备数据复制，分组间数据不同步。

\*\*正常情况下，分组内的主服务器对外提供消息写入和消息读取服务，备服务器不对外提供服务；主服务器宕机的情况下，备服务器对外提供消息读取的服务。

\*\*客户端采取轮询的策略写入和读取消息。

\*3.核心流程

\*4.详细设计

**高可用设计**

\*\*消息发送可靠性

业务服务器中嵌入消息队列系统提供的SDK，SDK支持轮询发送消息，当某个分组的主服务器无法发送消息时，SDK挑选下一个分组主服务器重发消息，依次尝试所有主服务器直到发送成功；如果全部主服务器都无法发送，SDK可以缓存消息，也可以直接丢弃消息，具体策略可以在启动SDK的时候通过配置指定。

\*\*消息存储可靠性

消息存储在MySQL中，每个分组有一主一备两台MySQL服务器，MySQL服务器之间复制消息以保证消息存储高可用。

\*\*消息读取可靠性

每个分组有一主一备两台服务器，主服务器支持发送和读取消息，备服务器只支持读取消息，当主服务器正常的时候备服务器不对外提供服务，只有备服务器判断主服务器故障的时候才对外提供消息读取服务。

**高性能设计**

**可扩展设计**

**安全设计**

消息队列系统需要提供权限控制功能，权限控制包括两部分：身份识别和队列权限控制。

\*\*身份识别：消息队列系统给业务子系统分配身份标识和接入key，SDK首先需要建立连接并进行身份校验，消息队列服务器会中断校验不通过的连接。因此，任何业务子系统如果想接入消息队列系统，都必须首先申请身份标识和接入key，通过这种方式来防止恶意系统任意接入。

\*\*队列权限：某些队列信息可能比较敏感，只允许部分子系统发送或者读取，消息队列系统将队列权限保存在配置文件中，当收到发送或者读取消息的请求时，首先需要根据业务子系统的身份标识以及配置的权限信息来判断业务子系统是否有权限，如果没有权限则拒绝服务。

**其他设计**

\*\*消息队列系统需要接入公司已有的运维平台，通过运维平台发布和部署。

\*\*消息队列系统需要输出日志给公司已有的监控平台，通过监控平台监控消息队列系统的健康状态，包括发送消息的数量、发送消息的大小、积压消息的数量等，详细监控指标在后续设计方案中列出。

**部署方案：**部署方案主要包括硬件要求、服务器部署方式、组网方式等

消息队列系统的服务器和数据库服务器采取混布的方式部署，即：一台服务器上，部署同一分组的主服务器和主MySQL，或者备服务器和备MySQL。因为消息队列服务器主要是CPU密集型，而MySQL是磁盘密集型的，所以两者混布互相影响的几率不大。

\*5.架构演进规划

通常情况下，规划和设计的需求比较完善，但如果一次性全部做完，项目周期可能会很长，因此可以采取分阶段实施，即：第一期做什么、第二期做什么，以此类推

整个消息队列系统分三期实现：

第一期：实现消息发送、权限控制功能，预计时间3个月。

第二期：实现消息读取功能，预计时间1个月。

第三期：实现主备基于ZooKeeper切换的功能，预计时间2周。

附：

**一、如何高效学习开源项目**

对于开源项目，不能简单的采取“拿来主义”，而要比较深入的去学习开源项目，做到“知其然，知其所以然”，一方面是为了更好地应用这些开源项目，另一方面也是为了通过学习优秀的开源项目来提升自己的能力。

很多技术同学确实也想深入学习一些业界成熟和优秀的开源项目，例如Nginx、Redis、Netty等，但是在具体实践的时候，常常因为一些不正确的观点而误入歧途，例如：

\*只有开发这些开源项目的人才能真正理解，我没法参与这个项目开发，因此我很难深入理解。

\*我的项目没有用Redis，不用的话很难深入理解。

\*数据结构和算法很重要，所以我只要研究其数据结构和算法就够了，例如Nginx用的红黑树。

\*“Talk is cheap, show me the code”，一头扎进源码逐行阅读。

这些观点要么让自己望而生畏从而轻易放弃，要么让自己浪费大量时间而没有多大收获。

**首先**，需要树立正确的观念：不管你是什么身份，都可以从开源项目中学到很多东西。

要理解Redis的网络模型，我们不需要成为Redis的开发者，也不需要一定要用到Redis，只要具备一定的网络编程基础，再通过阅读Redis的源码，都可以学习Redis这种单进程的Reactor模型。

**其次**，不要只盯着数据结构和算法，事实上这两点在学习开源项目的时候并没有那么重要。

Nginx使用红黑树来管理定时器，对于绝大部分人来说，只要知道这点就够了，并不需要去研究Nginx实现红黑树的源码是如何写的，除非你需要修改这部分，但我认为极少人会有这个需求。

**第三**，采取“自顶向下”的学习方法，源码不是第一步，而是最后一步。

不要一上来就去看源码，而是要基本掌握了功能、原理、关键设计之后再去看源码，看源码的主要目的是为了学习其代码的写作方式，以及关键技术的实现。

\*原理研究

\*\*关键特性的基本实现原理

每个流行的开源项目之所以能够受到大众的欢迎，肯定是有一些卖点的，常见的有高性能、高可用、可扩展等特性，那到底这些项目是如何做到其所宣称的那么牛的呢？这些牛X的技术实现就是我们要学习的地方。

例如，Memcache的高性能具体是怎么做到的呢？首先是基于libevent实现了高性能的网络模型，其次是内存管理Slab Allocator机制。为了彻底理解Memcache的高性能网络模型，我们需要掌握很多知识：多路复用、Linux epoll、Reactor模型、多线程等，通过研究Memcache的高性能网络模型，我们能够学习一个具体的项目中如何将这些东西全部串起来实现了高性能。

再以React为例，Virtual DOM的实现原理是什么、为何要实现Virtual DOM、React是如何构建Virtual DOM树、Virtual DOM与DOM什么关系等，通过研究学习Virtual DOM，即使不使用React，我们也能够学习如何写出高性能的前端的代码。

\*\*优缺点对比分析

只有清楚掌握技术方案的优缺点后才算真正的掌握这门技术，也只有掌握了技术方案的优缺点后才能在架构设计的时候做出合理的选择。

**原理研究的手段，主要有三种**：

\*\*通读项目的设计文档：例如Kafka的设计文档，基本涵盖了消息队列设计的关键决策部分；Disruptor的设计白皮书，详细的阐述了Java单机高性能的设计技巧。

\*\*阅读网上已有的分析文档：通常情况下比较热门的开源项目，都已经有非常多的分析文档了，我们可以站在前人的基础上，避免大量的重复投入。但需要注意的是，由于经验、水平、关注点等差异，不同的人分析的结论可能有差异，甚至有的是错误的，因此不能完全参照。一个比较好的方式就是多方对照，也就是说看很多篇分析文档，比较它们的内容共同点和差异点。

\*\*Demo验证：如果有些技术点难以查到资料，自己又不确定，则可以真正去写Demo进行验证，通过打印一些日志或者调试，能清晰的理解具体的细节。例如，写一个简单的分配内存程序，然后通过日志和命令行（jmap、jstat、jstack等）来查看Java虚拟机垃圾回收时的具体表现。

\*开源测试

通常情况下，如果是要在实践项目中使用，必须自己进行测试，因为网上搜的测试结果，不一定与自己的业务场景很契合，如果简单参考别人的测试结果，很可能会得出错误的结论。例如，开源系统的版本不同，测试结果可能差异较大。同样是K-V存储，别人测试的value是128字节，而你的场景value都达到了128k字节，两者的测试结果也差异很大，不能简单照搬。

曾经有团队安装完成MySQL 5.1后就进行性能测试，测试结果出来让人大跌眼镜，经过定位才发现innodb\\_buffer\\_pool\\_size使用的是默认值8M。

\*源码研究

源码研究的主要目的是学习原理背后的具体编码如何实现，通过学习这些技巧来提升我们自己的技术能力。例如Redis的RDB快照、Nginx的多Reactor模型、Disruptor如何使用volatile以及CAS来做无锁设计、Netty的Zero-Copy等，这些技巧都很精巧，掌握后能够大大提升自己的编码能力。

通常情况下，不建议通读所有源码，因为想掌握每行代码的含义和作用还是非常耗费时间的，尤其是MySQL、Nginx这种规模的项目，即使是他们的开发人员，都不一定每个人都掌握了所有代码。带着明确目的去研究源码，做到有的放矢，才能事半功倍，这也是源码研究要放在最后的原因。

\*时间分配

上5个步骤(安装，运行，原理研究，测试，源码研究)的前3个步骤，不管是已经成为架构师的技术人员，还是立志成为架构师的技术人员，在研究开源项目的时候都必不可少；第四步可以在准备采用开源项目的时候才实施，第五步可以根据你的时间来进行灵活安排。这里的“灵活安排”不是说省略不去做，而是在自己有一定时间和精力的时候做，因为只有这样才能真正理解和学到具体的技术。

二、照着做成为架构师

原本以为学习架构设计就像学习一门编程语言一样，先学习一下基本的语法，再研究一下细节和原理，然后实践一下就能够快速掌握。但真正实践后才发现，架构设计的难度和复杂度要高很多。

几个架构设计相关的特性：

\*1.架构设计的思维和程序设计的思维差异很大。

架构设计的关键思维是判断和取舍，程序设计的关键思维是逻辑和实现。很多程序员在转换为架构师后，很难一开始就意识到这个差异，还是按照写代码的方式去思考架构，会导致很多困惑。

\*2.架构设计没有体系化的培训和训练机制。

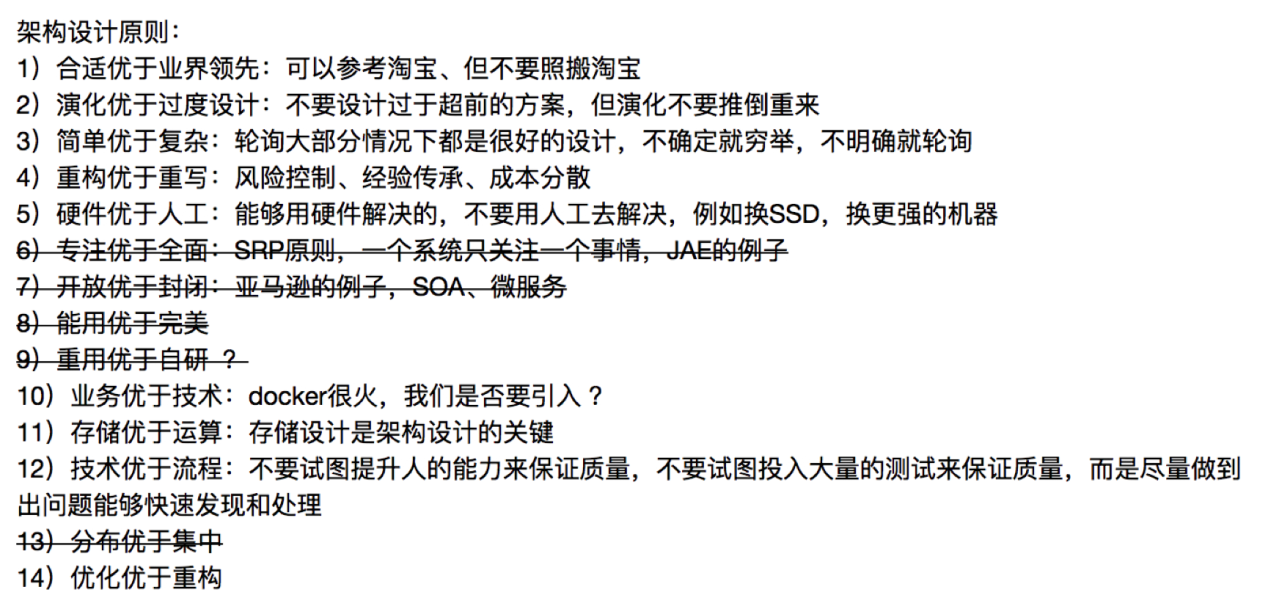
大学的课程几乎没有架构设计相关的课程，架构设计的书籍更多的也只是关注某个架构设计点，没有体系化的架构设计书籍，导致程序员在学习上没有明确指导，只能自己慢慢摸索，效率低，容易踩坑。

\*3.程序员对架构设计的理解存在很多误区。

例如：要成为架构师必须要有很强的技术天分；架构师必须有很强的创造力；架构设计必须要高大上才能体现架构师能力；架构一定要具备高可用、高性能……这些似是而非的误区让很多技术人员望而生畏，还没尝试就已经放弃了。

三、架构设计原则

手机App为例，首先，我们分析一下App的复杂度主要来源是什么？通常情况下，App的主要复杂度就是可扩展，因为要不断地开发新的需求；高性能和高可用也涉及，高性能主要和用户体验有关；高可用主要是减少崩溃。其次，再看App的架构需要遵循架构设计原则么？答案是肯定需要。刚开始为了业务快速开发，可能用“原生+H5”混合架构；后来业务发展，功能更复杂了，H5可能难以满足体验，架构又需要演进到“纯原生”；如果业务再发展，规模太庞大，则架构又可能需要演进到“组件化、容器化”。



3.1一个优秀的架构师，需要对细节有多少考虑：

以我学习Elasticsearch为例，具体的做法是：

\*搭建一个单机伪集群，搭建完成后看看安装路径下的文件和目录，看看配置文件有哪些配置项，不同的配置项会有什么样的影响。

\*执行常用的操作，例如创建索引，插入、删除、查询文档，查看一下各种输出。

\*研究其基本原理，例如索引、分片、副本等，研究的时候要多思考，例如索引应该如何建，分片数量和副本数量对系统有什么影响等。

\*和其他类似系统对比，例如Solr、Sphinx，研究其优点、缺点、适用场景。

\*模拟一个案例看看怎么应用。例如，假设我用Elasticsearch来存储淘宝的商品信息，我应该如何设计索引和分片。

\*查看业界使用的案例，思考一下别人为何这么用；看看别人测试的结果，大概了解性能范围。

\*如果某部分特别有兴趣或者很关键，可能去看源码，例如Elasticsearch的选举算法（我目前还没看^\_^）。

\*如果确定要引入，会进行性能和可用性测试。

3.2给企业做项目，甲方会不顾业务需要，只要是业界流行的技术就要求在项目中采用，这种情况下怎样才能符合“架构设计三原则”？

首先，业务第一，先把订单签下来，才有后面的架构设计，如果硬要说甲方的要求不合理，不满足“架构设计三原则”，结果订单都拿不到，那是没有意义的。

其次，这种情况我把它归为“架构约束”，即这不是架构师能够选择的，而是架构师必须遵守的，因此这里不需要使用“架构设计三原则”来判断。

第三，这种情况下，架构师还是可以应用“架构设计三原则”来指导架构设计，比如说客户要求采用Docker，Docker的网络模式有5种，host模式使用起来比bridge模式简单，那我们就用host模式；如果客户再要求需要对Docker进行统一管理，那我们是自己研发Docker管理平台，还是直接用Kubernetes呢？按照简单原则来说，肯定用Kubernetes了。

“**架构设计三原则**”主要是指架构师在选择和判断时采取的指导原则；但如果是架构的基本需求或者约束必须被满足时，架构师此时的选择是采取什么样的方案能够更好的满足这些需求和约束。

四、成长路线

架构师的内功主要包含三部分：判断力、执行力、创新力，简单解释如下：

\*判断力：能够准确判断系统的复杂度在哪里，就像武侠高手一样，能准确地看出对手的破绽和弱点。

\*执行力：能够使用合适的方案解决复杂度问题，就像武侠高手一样，能选择合适的招式或者方法打败对手。

\*创新力：能够创造新的解决方案解决复杂度问题，就像武侠世界里，小一些的创新是创新招式，而武学宗师能够创立新的武学或者心法，例如张三丰创立太极拳一样。

要成为一个优秀的架构师，就需要不断地提升自己这几方面的内功，而这三方面的能力主要来源于经验、视野、思考。

经验：设计过的系统越多、系统越复杂，架构师的内功也就越强，不管是成功的架构，还是失败的架构，不管是踩坑的经验，还是填坑的经验，都将成为架构师内功的一部分。

视野：掌握的知识和技能越多、越深，架构师的内功也就越强，他山之石可以攻玉，站在巨人的肩膀上会看的更高更远。

思考：经验和视野都是外部输入，类似于我们吃的食物，但光吃还不行，还要消化，将其变为我们自己的营养，这就是思考的作用。思考能够将经验和视野中的模式、判断、选择、技巧等提炼出来为我所用，思考也能促使我们产生新的创意和灵感。

**技术专家**

从高级工程师成长为技术专家，主要需要“拓展技术宽度”，因为一个“领域”必然会涉及众多的技术面。以Java后端开发为例，要成为一个Java开发专家，需要掌握Java多线程、JDBC、Java虚拟机、面向对象、设计模式、Netty、Elasticsearch、Memcache、Redis、MySQL等众多技术。常见的拓展技术宽度的方法有：

\*学习业界成熟的开源方案，例如，Java开发可以去学习Redis、Memcache、Netty等，Android开发可以去研究Retrofit、Fresco、OkHttp等。

\*研究业界的经验分享，例如BAT、FANG等大公司的经验，可以通过参加技术大会等方式去近距离了解。

**初级架构师**

初级架构师和技术专家的典型区别是：架构师是基于完善的架构设计方法论的指导来进行架构设计，而技术专家更多的是基于经验进行架构设计。简单来说，即使是同样一个方案，初级架构师能够清晰地阐述架构设计的理由和原因，而技术专家可能就是因为自己曾经这样做过，或者看到别人这样做过而选择设计方案。

但在实践工作中，技术专家和初级架构师的区别并不很明显，事实上很多技术专家其实就承担了初级架构师的角色，因为在系统复杂度相对不高的情况下，架构设计的难度不高，用不同的备选方案最终都能够较好地完成系统设计。

设计一个日PV 100万的网站：MySQL + Memcache + Spring Boot可以很好地完成，MongoDB + Redis + Nginx + php-fpm也可以很好地完成，备选方案设计和选择并不太难，更多的是看团队熟悉哪个技术。

从技术专家成长为初级架构师，最主要的是形成自己的“*架构设计方法论*”，我的架构设计专栏其实就是讲述完整的架构设计方法论，包括架构设计目的、架构设计原则、架构设计步骤、架构设计模式等，类似的架构设计方法论还有《恰如其分的软件架构：风险驱动的设计方法》和《领域驱动设计》等。

\*要形成自己的架构设计方法论，主要的手段有：

\*\*系统学习架构设计方法论，包括订阅专栏或者阅读书籍等。

\*\*深入研究成熟开源系统的架构设计，这个手段在技术专家阶段也会用到，但关注点不一样，同样是研究开源系统，技术专家阶段聚焦于如何更好地应用开源项目；初级架构师阶段聚焦于学习其架构设计原理和思想，例如Kafka的文档中就有关于消息队列架构设计的分析和取舍。

\*\*结合架构设计方法论，分析和总结自己团队甚至公司的各种系统的架构设计优缺点，尝试思考架构重构方案。如果在这个基础上真的能够推动架构重构，那就更好了，既能够实践自己的架构设计方法论，同时积累经验，又能够展现自己的技术实力，拿到结果。

**中级架构师**

典型特征是“能够完成复杂系统的架构设计”，包含高性能、高可用、可扩展、海量存储等复杂系统，例如设计一个和Kafka性能匹敌的消息队列系统、将业务改造为异地多活、设计一个总共100人参与开发的业务系统等。

从初级架构师成长为中级架构师，最关键的是“技术深度和技术理论的积累”，例如：

\*\*技术理论：CAP、BASE是异地多活的设计理论基础、Paxos是分布式一致性的基础算法、2PC、3PC是分布式事务的基础算法等。

\*\*技术深度：Kafka用磁盘存储还能做到高效是因为磁盘顺序写；Disruptor高性能是结合CPU预读取机制、缓存行、无锁设计等基础技术；Storm的高效异或确认机制；Flink的分布式快照算法等。

**高级架构师**

成长为高级架构师需要10年以上时间，其典型特征是“创造新的架构模式”，例如：

\*谷歌大数据论文，创造了分布式存储架构、分布式计算MapReduce架构、列式存储架构，开创了大数据时代。

\*在有MapReduce分布式计算架构的背景下，Storm又创造了流式计算架构。

\*在虚拟机很成熟的背景下，Docker创造了容器化的技术潮流。

总结几个可能诞生创造性架构的背景条件：

\*足够复杂的业务场景：例如谷歌的大数据、阿里的双十一、Facebook的海量用户等，业务场景越复杂，给技术带来的挑战更大，更有可能产生创造性的技术突破。

\*足够强大的技术团队：绝大部分创造性的架构都来源于大公司，或者知名的研究机构；没有技术实力支撑，想突破也是心有余而力不足。

\*不满足于现状的态度：例如虚拟机很成熟但是资源占用太多，所以发明Docker；MapReduce难以做到实时运算，所以创造Storm流式运算。

\*尊重技术价值的文化：创造性的东西往往需要投入大量的人力和时间，而且刚开始一般都不会很成熟，如果完全结果导向、KPI导向，创新技术很可能在萌芽阶段就被否定。

**第一个坚持：坚持梦想！**

几乎每个技术人员心中都有一个架构师的梦想，毕竟架构师代表了技术路线发展的巅峰。

所谓成长，其实就是不断学习、不断踩坑、不断填坑的过程。

**第二个坚持：坚持学习！**

学无止境。但我想，这就是技术的趣味所在，总是有更好的、更新的、更厉害的东西出来。

当年谷歌发布大数据论文的时候，我觉得MapReduce好牛，没想到后面Storm流式计算更厉害；当我觉得Storm好厉害的时候，Flink又创造了更厉害的流式计算架构。

**第三个坚持：坚持输出！**

输出就是把你所学到的东西，再传授给他人，包括培训、演讲、写博客、写书等，这是一个非常好的提升自己的手段。我相信很多人都有这个感觉，很多东西感觉自己学了也懂了，但一旦跟别人交流有些问题就可能回答不上来，或者一写博客就发现其实还有很多细节没有考虑，需要再重新去研究或者考证。

输出还能够锻炼自己的表达能力、临场反应能力，这些是大多数技术人员比较欠缺但又比较关键的能力。技术人员要想有更好的职业发展，不能只是埋头干活，也需要跟别人沟通交流，而输出是最方便的锻炼方法，不需要承担管理职责就可以锻炼，任何级别都可以通过输出来锻炼自己的能力，不一定需要长篇大论，也不一定需要篇篇10W+，只要你觉得某个点其他人也可能遇到，哪怕写100字的博客也是可以的，关键点在于持续地输出。