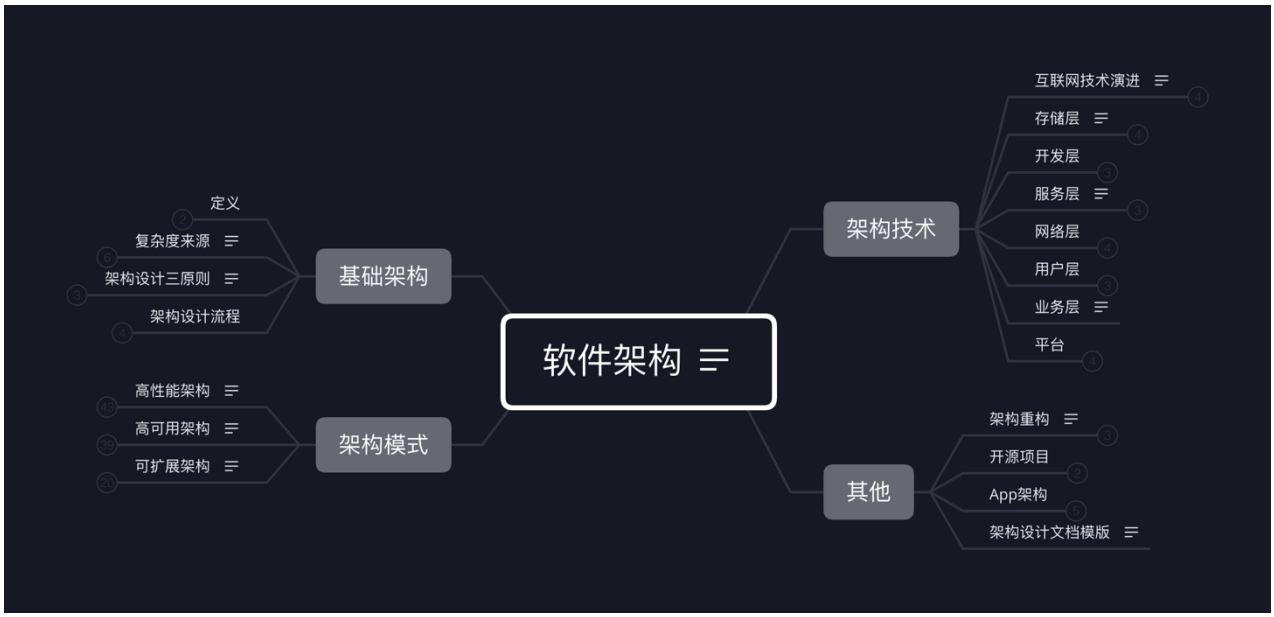
软件架构

软件架构指软件系统的基础结构： 架构需要明确系统包含哪些“个体”：“系统是一群关联个体组成”，这些“个体”可以是“子系统” “模块” “组件”等。 架构需要明确个体运作和协作的规则：系统中的个体需要“根据某种规则”运作。 解决的问题： 系统规模庞大，内部耦合严重，开发效率低； 系统耦合严重，牵一发动全身，后续修改和扩展困难； 系统逻辑复杂，容易出问题，出问题后很难排查和修复。



1. 架构技术

1.1. 互联网技术演进

互联网技术演进的方向： 业务影响技术的方向 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

1.1.1. 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。

1.1.2. 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。 如何做到“快”，一般会经历下面几个阶段： 堆功能期 优化期 架构期

1.1.3. 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。新业务的创新给技术带来的典型压力就是新的系统会更多，同时，原有的系统也会拆得越来越多。两者合力的一个典型后果就是系统数量在原来的基础上又增加了很多。架构拆分后带来的美好时光又开始慢慢消逝，技术工作又开始进入了“慢”的状态。系统数量越来越多，到了一个临界点后就产生了质变，即系统数量的量变带来了技术工作的质变。主要体现在下面几个方面： 重复造轮子 系统交互一团乱麻 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：解决“重复造轮子”的问题。 服务化：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。

1.1.4. 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……此时技术上其实也基本进入了成熟期，该拆的也拆了，该平台化的也平台化了，技术上能做的大动作其实也不多了，更多的是进行优化。但有时候也会为了满足某个优化，系统做很大的改变。例如，为了将用户响应时间从 200ms 降低到 50ms，可能就需要从很多方面进行优化：CDN、数据库、网络等。这个时候的技术优化没有固定的套路，只能按照竞争的要求，找出自己的弱项，然后逐项优化。在逐项优化时，可以采取之前各个时期采用的手段。

1.2. 存储层

很多人对于 BAT 的技术有一种莫名的崇拜感，觉得只有天才才能做出这样的系统，但经过前面对架构的本质、架构的设计原则、架构的设计模式、架构演进等多方位的探讨和阐述，你可以看到，其实并没有什么神秘的力量和魔力融合在技术里面，而是业务的不断发展推动了技术的发展，这样一步一个脚印，持续几年甚至十几年的发展，才能达到当前技术复杂度和先进性。抛开 BAT 各自差异很大的业务，站在技术的角度来看，其实 BAT 的技术架构基本是一样的。再将视角放大，你会发现整个互联网行业的技术发展，最后都是殊途同归。如果你正处于一个创业公司，或者正在为成为另一个 BAT 拼搏，那么深入理解这种技术模式（或者叫技术结构、技术架构），对于自己和公司的发展都大有裨益。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。从本期开始，我将逐层介绍每个技术点的产生背景、应用场景、关键技术，有的技术点可能已经在前面的架构模式部分有所涉及，因此就不再详细展开技术细节了，而是将关键技术点分门别类，进而形成一张架构大图，让架构师对一个公司的整体技术架构有一个完整的全貌认知。今天我们首先来聊聊互联网架构模板的“存储层”技术。SQLSQL 即我们通常所说的关系数据。前几年

NoSQL 火了一阵子，很多人都理解为 NoSQL 是完全抛弃关系数据，全部采用非关系型数据。但经过几年的试验后，大家发现关系数据不可能完全被抛弃，NoSQL 不是 No SQL，而是 Not Only SQL，即 NoSQL 是 SQL 的补充。所以互联网行业也必须依赖关系数据，考虑到 Oracle 太贵，还需要专人维护，一般情况下互联网行业都是用 MySQL、PostgreSQL 这类开源数据库。这类数据库的特点是开源免费，拿来就用；但缺点是性能相比商业数据库要差一些。随着互联网业务的发展，性能要求越来越高，必然要面对一个问题：将数据拆分到多个数据库实例才能满足业务的性能需求（其实 Oracle 也一样，只是时间早晚的问题）。数据库拆分满足了性能的要求，但带来了复杂度的问题：数据如何拆分、数据如何组合？这个复杂度的问题解决起来并不容易，如果每个业务都去实现一遍，重复造轮子将导致投入浪费、效率降低，业务开发想快都快不起来。所以互联网公司流行的做法是业务发

1.2.1. SQL

SQL 即我们通常所说的关系数据。前几年 NoSQL 火了一阵子，很多人都理解为 NoSQL 是完全抛弃关系数据，全部采用非关系型数据。但经过几年的试验后，大家发现关系数据不可能完全被抛弃，NoSQL 不是 No SQL，而是 Not Only SQL，即 NoSQL 是 SQL 的补充。所以互联网行业也必须依赖关系数据，考虑到 Oracle 太贵，还需要专人维护，一般情况下互联网行业都是用 MySQL、PostgreSQL 这类开源数据库。这类数据库的特点是开源免费，拿来就用；但缺点是性能相比商业数据库要差一些。随着互联网业务的发展，性能要求越来越高，必然要面对一个问题：将数据拆分到多个数据库实例才能满足业务的性能需求（其实 Oracle 也一样，只是时间早晚的问题）。数据库拆分满足了性能的要求，但带来了复杂度的问题：数据如何拆分、数据如何组合？这个复杂度的问题解决起来并不容易，如果每个业务都去实现一遍，重复造轮子将导致投入浪费、效率降低，业务开发想快都快不起来。所以互联网公司流行的做法是业务发展到一定阶段后，就会将这部分功能独立成中间件，例如百度的 DBProxy、淘宝的 TDDL。不过这部分的技术要求很高，将分库分表做到自动化和平台化，不是一件容易的事情，所以一般是规模很大的公司才会自己做。中小公司建议使用开源方案，例如 MySQL 官方推荐的 MySQL Router、360 开源的数据库中间件 Atlas。假如公司业务继续发展，规模继续扩大，SQL 服务器越来越多，如果每个业务都基于统一的数据库中间件独立部署自己的 SQL 集群，就会导致新的复杂度问题，具体表现在：数据库资源使用率不高，比较浪费。各 SQL 集群分开维护，投入的维护成本越来越高。因此，实力雄厚的大公司此时一般都会在 SQL 集群上构建 SQL 存储平台，以对业务透明的形式提供资源分配、数据备份、迁移、容灾、读写分离、分库分表等一系列服务，例如淘宝的 UMP（Unified MySQL Platform）系统。

1.2.2. NoSQL

首先 NoSQL 在数据结构上与传统的 SQL 的不同，例如典型的 Memcache 的 key-value 结构、Redis 的复杂数据结构、MongoDB 的文档数据结构；其次，NoSQL 无一例外地都会将性能作为自己的一大卖点。NoSQL 的这两个特点很好地弥补了关系数据库的不足，因此在互联网行业 NoSQL 的应用基本上是基础要求。由于 NoSQL 方案一般自己本身就提供集群的功能，例如 Memcache 的一致性 Hash 集群、Redis 3.0 的集群，因此 NoSQL 在刚开始应用时很方便，不像 SQL 分库分表那么复杂。一般公司也不会在开始时就考虑将 NoSQL 包装成存储平台，但如果公司发展很快，例如 Memcache 的节点有上千甚至几千时，NoSQL 存储平台就很有意义了。首先是存储平台通过集中管理能够大大提升运维效率；其次是存储平台可以大大提升资源利用效率，2000 台机器，如果利用率能提升 10%，就可以减少 200 台机器，一年几十万元就节省出来了。所以，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。当然要发展到这个阶段，一般也是大公司才会这么做，简单来说就是如果只有几十台 NoSQL 服务器，做存储平台收益不大；但如果有几千台 NoSQL 服务器，NoSQL 存储平台就能够产生很大的收益。

1.2.3. 小文件存储

除了关系型的业务数据，互联网行业还有很多用于展示的数据。例如，淘宝的商品图片、商品描述；Facebook 的用户图片；新浪微博的一条微博内容等。这些数据具有三个典型特征：一是数据小，一般在 1MB 以下；二是数量巨大，Facebook 在 2013 年每天上传的照片就达到了 3.5 亿张；三是访问量巨大，Facebook 每天的访问量超过 10 亿。由于互联网行业基本上每个业务都会有大量的小数据，如果每个业务都自己去考虑如何设计海量存储和海量访问，效率自然会低，重复造轮子也会投入浪费，所以自然而然就要将小文件存储做成统一的和业务无关的平台。和 SQL 和 NoSQL 不同的是，小文件存储不一定需要公司或者业务规模很大，基本上认为业务在起步阶段就可以考虑做小文件统一存储。得益于开源运动的发展和最近几年大数据的火爆，在开源方案的基础上封装一个小文件存储平台并不是太难的事情。例如，HBase、Hadoop、Hypertable、FastDFS 等都可以作为小文件存储的底层平台，只需要将这些开源方案再包装一下基本上就可以用了。典型的小文件存储有：淘宝的 TFS、京东 JFS、Facebook 的 Haystack。

1.2.4. 大文件存储

互联网行业的大文件主要分为两类：一类是业务上的大数据，例如 Youtube 的视频、电影网站的电影；另一类是海量的日志数据，例如各种访问日志、操作日志、用户轨迹日志等。和小文件的特点正好相反，大文件的数量没有小文件那么多，但每个文件都很大，几百 MB、几个 GB 都是常见的，几十 GB、几 TB 也是有可能的，因此在存储上和小文件有较大差别，不能直接将小文件存储系统拿来存储大文件。说到大文件，特别要提到 Google 和 Yahoo，Google 的 3 篇大数据论文（Bigtable/Map- Reduce/GFS）开启了一个大数据的时代，而 Yahoo 开源的 Hadoop 系列（HDFS、HBase 等），基本上垄断了开源界的大数据处理。当然，江山代有才人出，长江后浪推前浪，Hadoop 后又有更多优秀的开源方案被贡献出来，现在随便走到大街上拉住一个程序员，如果他不知道大数据，那基本上可以确定是“火星程序员”。对照 Google 的论文构建一套完整的大数据处理方案的难度和成本实在太高，而且开源方案现在也很成熟了，所以大数据存储和处理这块反而是最简单的，因为你没有太多选择，只能用这几个流行的开源方案，例如，Hadoop、HBase、Storm、Hive 等。实力雄厚一些的大公司会基于这些开源方案，结合自己的业务特点，封装成大数据平台，例如淘宝的云梯系统、腾讯的 TDW 系统。

1.3. 开发层

1.3.1. 开发框架

在专栏第 38、39 期中，我们深入分析了互联网业务发展的一个特点：复杂度越来越高。复杂度增加的典型现象就是系统越来越多，不同的系统由不同的小组开发。如果每个小组用不同的开发框架和技术，则会带来很多问题，典型的问题有：技术人员之间没有共同的技术语言，交流合作少。每类技术都需要投入大量的人力和资源并熟练精通。不同团队之间人员无法快速流动，人力资源不能高效的利用。所以，互联网公司都会指定一个大的技术方向，然后使用统一的开发框架。例如，Java 相关的开发框架 SSH、SpringMVC、Play，Ruby 的 Ruby on Rails，PHP 的 ThinkPHP，Python 的 Django 等。使用统一的开发框架能够解决上面提到的各种问题，大大提升组织和团队的开发效率。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！为什么呢？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。

1.3.2. 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。独立开发一个成熟的 Web 服务器，成本非常高，况且业界又有那么多成熟的开源 Web 服务器，所以互联网行业基本上都是“拿来主义”，挑选一个流行的开源服务器即可。大一点的公司，可能会在开源服务器的基础上，结合自己的业务特点做二次开发，例如淘宝的 Tengine，但一般公司基本上只需要将开源服务器摸透，优化一下参数，调整一下配置就差不多了。选择一个服务器主要和开发语言相关，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。你可能会担心 Apache 的性能之类的问题，其实不用过早担心这个，等到业务真的发展到 Apache 撑不住的时候再考虑切换也不迟，那时候你有的是钱，有的是人，有的是时间。

1.3.3. 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。例如，腾讯万台规模的 Docker 应用实践（http://www.infoq.com/cn/articles/tencent-millions-scale-docker-application-practice）、新浪微博红包的大规模 Docker 集群（http://www.infoq.com/cn/articles/large-scale-docker-cluster-practise-experience-share）等。传统的虚拟化技术是虚拟机，解决了跨平台的问题，但由于虚拟机太庞大，启动又慢，运行时太占资源，在互联网行业并没有大规模应用；而 Docker 的容器技术，虽然没有跨平台，但启动快，几乎不占资源，推出后立刻就火起来了，预计 Docker 类的容器技术将是技术发展的主流方向。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：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。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。例如，一个传统的网站包括登录注册、页面访问、搜索等功能，没有用容器的情况下，除非有特别大的访问量，否则这些功能开始时都是集成在一个系统里面的；有了容器技术后，一开始就可以将这些功能按照服务的方式设计，避免后续访问量增大时又要重构系统。

1.4. 服务层

互联网业务的不断发展带来了复杂度的不断提升，业务系统也越来越多，系统间相互依赖程度加深。比如说为了完成 A 业务系统，可能需要 B、C、D、E 等十几个其他系统进行合作。从数学的角度进行评估，可以发现系统间的依赖是呈指数级增长的：3 个系统相互关联的路径为 3 条，6 个系统相互关联的路径为 15 条。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。

1.4.1. 配置中心

故名思议，配置中心就是集中管理各个系统的配置。当系统数量不多的时候，一般是各系统自己管理自己的配置，但系统数量多了以后，这样的处理方式会有问题：某个功能上线时，需要多个系统配合一起上线，分散配置时，配置检查、沟通协调需要耗费较多时间。处理线上问题时，需要多个系统配合查询相关信息，分散配置时，操作效率很低，沟通协调也需要耗费较多时间。各系统自己管理配置时，一般是通过文本编辑的方式修改的，没有自动的校验机制，容易配置错误，而且很难发现。例如，我曾经遇到将 IP 地址的数字 0 误敲成了键盘的字母 O，肉眼非常难发现，但程序检查其实就很容易。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：集中配置多个系统，操作效率高。所有配置都在一个集中的地方，检查方便，协作效率高。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。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。整机磁盘坏掉、机器主板坏掉……遇到这些不可恢复的故障时，基本上只能重新搭建新的环境。程序包肯定是已经有的，加上配置中心的配置，能够很快搭建新的运行环境，恢复业务。否则几十个配置文件重新一个个去 Vim 中修改，耗时很长，还很容易出错。下面是配置中心简单的设计，其中通过“系统标识 + host + port”来标识唯一一个系统运行实例是常见的设计方法。

1.4.2. 服务中心

当系统数量不多的时候，系统间的调用一般都是直接通过配置文件记录在各系统内部的，但当系统数量多了以后，这种方式就存在问题了。比如说总共有 10 个系统依赖 A 系统的 X 接口，A 系统实现了一个新接口 Y，能够更好地提供原有 X 接口的功能，如果要让已有的 10 个系统都切换到 Y 接口，则这 10 个系统的几十上百台机器的配置都要修改，然后重启，可想而知这个效率是很低的。除此以外，如果 A 系统总共有 20 台机器，现在其中 5 台出故障了，其他系统如果是通过域名访问 A 系统，则域名缓存失效前，还是可能访问到这 5 台故障机器的；如果其他系统通过 IP 访问 A 系统，那么 A 系统每次增加或者删除机器，其他所有 10 个系统的几十上百台机器都要同步修改，这样的协调工作量也是非常大的。服务中心就是为了解决上面提到的跨系统依赖的“配置”和“调度”问题。服务中心的实现一般来说有两种方式：服务名字系统和服务总线系统。服务名字系统（Service Name System）看到这个翻译，相信你会立刻联想到 DNS，即 Domain Name System。没错，两者的性质是基本类似的。DNS 的作用将域名解析为 IP 地址，主要原因是我们记不住太多的数字 IP，域名就容易记住。服务名字系统是为了将 Service 名称解析为“host + port + 接口名称”，但是和 DNS 一样，真正发起请求的还是请求方。基本的设计如下：服务总线系统（Service Bus System）看到这个翻译，相信你可能立刻联想到计算机的总线。没错，两者的本质也是基本类似的。相比服务名字系统，服务总线系统更进一步了：由总线系统完成调用，服务请求方都不需要直接和服务提供方交互了。基本的设计如下：“服务名字系统”和“服务总线系统”简单对比如下表所示：

1.4.3. 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。例如，大 V 发布一条微博后，系统需要发消息给关注的用户，我们不可能等到所有消息都发送给关注用户后再告诉大 V 说微博发布成功了，只能先让大 V 发布微博，然后再发消息给关注用户。传统的异步通知方式是由消息生产者直接调用消息消费者提供的接口进行通知的，但当业务变得庞大，子系统数量增多时，这样做会导致系统间交互非常复杂和难以管理，因为系统间互相依赖和调用，整个系统的结构就像一张蜘蛛网，如下图所示：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。以微博为例，可以清晰地看到异步通知的实现和作用，如下图所示。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：整体结构从网状结构变为线性结构，结构清晰；消息生产和消息消费解耦，实现简单；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；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；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。开源的用起来方便，但要改就很麻烦了。由于其相对比较简单，很多公司也会花费人力和时间重复造一个轮子，这样也有好处，因为可以根据自己的业务特点做快速的适配开发。

1.5. 网络层

1.5.1. 负载均衡

顾名思议，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。1.DNSDNS 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，南方的用户访问广州的机房。一般不会使用 DNS 来做机器级别的负载均衡，因为太耗费 IP 资源了。例如，百度搜索可能要 10000 台以上机器，不可能将这么多机器全部配置公网 IP，然后用 DNS 来做负载均衡。有兴趣的读者可以在 Linux 用“dig baidu.com”命令看看实际上用了几个 IP 地址。DNS 负载均衡的优点是通用（全球通用）、成本低（申请域名，注册 DNS 即可），但缺点也比较明显，主要体现在：DNS 缓存的时间比较长，即使将某台业务机器从 DNS 服务器上删除，由于缓存的原因，还是有很多用户会继续访问已经被删除的机器。DNS 不够灵活。DNS 不能感知后端服务器的状态，只能根据配置策略进行负载均衡，无法做到更加灵活的负载均衡策略。比如说某台机器的配置比其他机器要好很多，理论上来说应该多分配一些请求给它，但 DNS 无法做到这一点。所以对于时延和故障敏感的业务，有实力的公司可能会尝试实现 HTTP-DNS 的功能，即使用 HTTP 协议实现一个私有的 DNS 系统。HTTP-DNS 主要应用在通过 App 提供服务的业务上，因为在 App 端可以实现灵活的服务器访问策略，如果是 Web 业务，实现起来就比较麻烦一些，因为 URL 的解析是由浏览器来完成的，只有 Javascript 的访问可以像 App 那样实现比较灵活的控制。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

1.5.2. CDN

CDN 是为了解决用户网络访问时的“最后一公里”效应，本质上是一种“以空间换时间”的加速策略，即将内容缓存在离用户最近的地方，用户访问的是缓存的内容，而不是站点实时的内容。下面是简单的 CDN 请求流程示意图：图片来自网络 CDN 经过多年的发展，已经变成了一个很庞大的体系：分布式存储、全局负载均衡、网络重定向、流量控制等都属于 CDN 的范畴，尤其是在视频、直播等领域，如果没有 CDN，用户是不可能实现流畅观看内容的。幸运的是，大部分程序员和架构师都不太需要深入理解 CDN 的细节，因为 CDN 作为网络的基础服务，独立搭建的成本巨大，很少有公司自己设计和搭建 CDN 系统，从 CDN 服务商购买 CDN 服务即可，目前有专门的 CDN 服务商，例如网宿和蓝汛；也有云计算厂家提供 CDN 服务，例如阿里云和腾讯云都提供 CDN 的服务。

1.5.3. 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。例如，停电、机房网络中断、地震、水灾等都有可能导致一个机房完全瘫痪。多机房设计最核心的因素就是如何处理时延带来的影响，常见的策略有：1. 同城多机房同一个城市多个机房，距离不会太远，可以投入重金，搭建私有的高速网络，基本上能够做到和同机房一样的效果。这种方式对业务影响很小，但投入较大，如果不是大公司，一般是承受不起的；而且遇到极端的地震、水灾等自然灾害，同城多机房也是有很大风险的。2. 跨城多机房在不同的城市搭建多个机房，机房间通过网络进行数据复制（例如，MySQL 主备复制），但由于跨城网络时延的问题，业务上需要做一定的妥协和兼容，比如不需要数据的实时强一致性，只是保证最终一致性。例如，微博类产品，B 用户关注了 A 用户，A 用户在北京机房发布了一条微博，B 在广州机房不需要立刻看到 A 用户发的微博，等 10 分钟看到也可以。这种方式实现简单，但和业务有很强的相关性，微博可以这样做，支付宝的转账业务就不能这样做，因为用户余额是强一致性的。3. 跨国多机房和跨城多机房类似，只是地理上分布更远，时延更大。由于时延太大和用户跨国访问实在太慢，跨国多机房一般仅用于备份和服务本国用户。

1.5.4. 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。简单来说，多机房的主要目标是灾备，当机房故障时，可以比较快速地将业务切换到另外一个机房，这种切换操作允许一定时间的中断（例如，10 分钟、1 个小时），而且业务也可能有损失（例如，某些未同步的数据不能马上恢复，或者要等几天才恢复，甚至永远都不能恢复了）。因此相比多机房来说，多中心的要求就高多了，要求每个中心都同时对外提供服务，且业务能够自动在多中心之间切换，故障后不需人工干预或者很少的人工干预就能自动恢复。多中心设计的关键就在于“数据一致性”和“数据事务性”如何保证，这两个难点都和业务紧密相关，目前没有很成熟的且通用的解决方案，需要基于业务的特性进行详细的分析和设计。以淘宝为例，淘宝对外宣称自己是多中心的，但是在实际设计过程中，商品浏览的多中心方案、订单的多中心方案、支付的多中心方案都需要独立设计和实现。正因

为多中心设计的复杂性，不一定所有业务都能实现多中心，目前国内的银行、支付宝这类系统就没有完全实现多中心，不然也不会出现挖掘机一铲子下去，支付宝中断 4 小时的故障。

1.6. 用户层

1.6.1. 用户管理

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，目前最成熟的开源单点登录方案当属 CAS，其架构如下（https://apereo.github.io/cas/4.2.x/planning/Architecture.html ）：除此之外，当业务做大成为了平台后，开放成为了促进业务进一步发展的手段，需要允许第三方应用接入，由此引申出用户管理的第二个目标：授权登录。现在最流行的授权登录就是 OAuth 2.0 协议，基本上已经成为了事实上的标准，如果要做开放平台，则最好用这个协议，私有协议漏洞多，第三方接入也麻烦。用户管理系统面临的主要问题是用户数巨大，一般至少千万级，QQ、微信、支付宝这种巨无霸应用都是亿级用户。不过也不要被这个数据给吓倒了，用户管理虽然数据量巨大，但实现起来并不难，原因是什么呢？ 因为用户数据量虽然大，但是不同用户之间没有太强的业务关联，A 用户登录和 B 用户登录基本没有关系。因此虽然数据量巨大，但我们用一个简单的负载均衡架构就能轻松应对。

1.6.2. 消息推送

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。App 目前主要分为 iOS 和 Android 推送，iOS 系统比较规范和封闭，基本上只能使用苹果的 APNS；但 Android 就不一样了，在国外，用 GCM 和 APNS 差别不大；但是在国内，情况就复杂多了：首先是 GCM 不能用；其次是各个手机厂商都有自己的定制的 Android，消息推送实现也不完全一样。因此 Android 的消息推送就五花八门了，大部分有实力的大厂，都会自己实现一套消息推送机制，例如阿里云移动推送、腾讯信鸽推送、百度云推送；也有第三方公司提供商业推送服务，例如友盟推送、极光推送等。通常情况下，对于中小公司，如果不涉及敏感数据，Android 系统上推荐使用第三方推送服务，因为毕竟是专业做推送服务的，消息到达率是有一定保证的。如果涉及敏感数据，需要自己实现消息推送，这时就有一定的技术挑战了。消息推送主要包含 3 个功能：设备管理（唯一标识、注册、注销）、连接管理和消息管理，技术上面临的主要挑战有：海量设备和用户管理消息推送的设备数量众多，存储和管理这些设备是比较复杂的；同时，为了针对不同用户进行不同的业务推广，还需要收集用户的一些信息，简单来说就是将用户和设备关联起来，需要提取用户特征对用户进行分类或者打标签等。连接保活要想推送消息必须有连接通道，但是应用又不可能一直在前台运行，大部分设备为了省电省流量等原因都会限制应用后台运行，限制应用后台运行后连接通道可能就被中断了，导致消息无法及时的送达。连接保活是整个消息推送设计中细节和黑科技最多的地方，例如应用互相拉起、找手机厂商开白名单等。消息管理实际业务运营过程中，并不是每个消息都需要发送给每个用户，而是可能根据用户的特征，选择一些用户进行消息推送。由于用户特征

变化很大，各种排列组合都有可能，将消息推送给哪些用户这部分的逻辑要设计得非常灵活，才能支撑花样繁多的业务需求，具体的设计方案可以采取规则引擎之类的微内核架构技术。

1.6.3. 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：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（http://mi.techweb.com.cn/tmt/2016-05-25/2338330.shtml）。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。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。为了满足用户的文件上传和存储需求，需要对用户提供文件存储和访问功能，这里就需要用到前面我在专栏第 40 期介绍“存储层”技术时提到的“小文件存储”技术。简单来说，存储云和图片云通常的实现都是“CDN + 小文件存储”，现在有了“云”之后，除非 BAT 级别，一般不建议自己再重复造轮子了，直接买云服务可能是最快也是最经济的方式。既然存储云和图片云都是基于“CDN + 小文件存储”的技术，为何不统一一套系统，而将其拆分为两个系统呢？这是因为“图片”业务的复杂性导致的，普通的文件基本上提供存储和访问就够了，而图片涉及的业务会更多，包括裁剪、压缩、美化、审核、水印等处理，因此通常情况下图片云会拆分为独立的系统对用户提供服务。

1.7. 业务层

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”。复杂度越来越高的一个主要原因就是系统越来越庞大，业务越来越多。幸运的是，面对业务层的技术挑战，我们有一把“屠龙宝刀”，不管什么业务难题，用上“屠龙宝刀”问题都能迎刃而解。这把“屠龙宝刀”就是“拆”，化整为零、分而治之，将整体复杂性分散到多个子业务或者子系统里面去。具体拆的方式你可以查看专栏前面可扩展架构模式部分的分层架构、微服务、微内核等。我以一个简单的电商系统为例，如下图所示：我这个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。上面只是个样例，实际上随着业务的发展，子系统会越来越多，据说淘宝内部大大小小的已经有成百上千的子系统了。随着子系统数量越来越多，如果达到几百上千，另外一个复杂度问题又会凸显出来：子系统数量太多，已经没有人能够说清楚业务的调用流程了，出了问题排查也会特别复杂。此时应该怎么处理呢，总不可能又将子系统合成大系统吧？最终答案还是“合”，正所谓“合久必分、分久必合”，但合的方式不一样，此时采取的“合”的方式是按照“高内聚、低耦合”的原则，将职责关联比较强的子系统合成一个虚拟业务域，然后通过网关对外统一呈现，类似于设计模式中的 Facade 模式。同样以电商为样例，采用虚拟业务域后，

1.8. 平台

1.8.1. 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。1. 标准化需要制定运维标准，规范配置管理、部署流程、监控指标、应急能力等，各系统按照运维标准来实现，避免不同的系统不同的处理方式。标准化是运维平台的基础，没有标准化就没有运维平台。如果某个系统就是无法改造自己来满足运维标准，那该怎么办呢？常见的做法是不改造系统，由中间方来完成规范适配。例如，某个系统对外提供了 RESTful 接口的方式来查询当前的性能指标，而运维标准是性能数据通过日志定时上报，那么就可以写一个定时程序访问 RESTful 接口获取性能数据，然后转换为日志上报到运维平台。2. 平台化传统的手工运维方式需要投入大量人力，效率低，容易出错，因此需要在运维标准化的基础上，将运维的相关操作都集成到运维平台中，通过运维平台来完成运维工作。运维平台的好处有：可以将运维标准固化到平台中，无须运维人员死记硬背运维标准。运维平台提供简单方便的操作，相比之下人工操作低效且容易出错。运维平台是可复用的，一套运维平台可以支撑几百上千个业务系统。3. 自动化传统手工运维方式效率低下的一个主要原因就是要执行大量重复的操作，运维平台可以将这些重复操作固化下来，由系统自动完成。例如，一次手工部署需要登录机器、上传包、解压包、备份旧系统、覆盖旧系统、启动新系统，这个过程中需要执行大量的重复或者类似的操作。有了运维平台后，平台需要提供自动化的能力，完成上述操作，部署人员只需要在最开始单击“开始部署”按钮，系统部署完成后通知部署人员即可。类似的还有监控，有了运维平台后，运维平台可以实时收集数据并进行初步分析，当发现数据异常时自动发出告警，无须运维人员盯着数据看，或者写一大堆“grep + awk + sed”来分析日志才能发现问题。4. 可视化运维平台有非常多的数据，如果全部通过人工去查询数据再来判断，则效率很低。尤其是在故障应急时，

1.8.2. 测试平台

测试平台核心的职责当然就是测试了，包括单元测试、集成测试、接口测试、性能测试等，都可以在测试平台来完成。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如下图所示：1. 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。为了能够重复执行这些测试用例，测试平台需要将用例管理起来，管理的维度包括业务、系统、测试类型、用例代码。例如，网购业务的订单系统的接口测试用例。2. 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）。除了性能测试，一般的自动化测试对性能要求不高，所以为了提升资源利用率，大部分的测试平台都会使用虚拟技术来充分利用硬件资源，如虚拟机、Docker 等技术。3. 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。4. 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：展现当前用例的执行情况。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%。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？

1.8.3. 数据平台

数据平台的核心职责主要包括三部分：数据管理、数据分析和数据应用。每一部分又包含更多的细分领域，详细的数据平台架构如下图所示：1. 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。2. 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。3. 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。数据应用能够发挥价值的前提是需要有“大数据”，只有当数据的规模达到一定程度，基于数据的分析、挖掘才能发现有价值的规律、现象、问题等。如果数据没有达到一定规模，通常情况下做好数据统计就足够了，尤其是很多初创企业，无须一开始就参考 BAT 来构建自己的数据平台。

1.8.4. 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。权限管理主要分为两部分：身份认证、权限控制，其基本架构如下图所示。1. 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。2. 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。

2. 其他

2.1. 架构重构

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： 业务已经上线，不能停下来 关联方众多，牵一发动全身 旧架构的约束 架构重构对架构师的综合能力要求非常高：

业务上要求架构师能够说服产品经理暂缓甚至暂停业务来进行架构重构； 团队上需要架构师能够与其他团队达成一致的架构重构计划和步骤；技术上需要架构师给出让技术团队认可的架构重构方案。

2.1.1. 有的放矢

架构师的首要任务是从一大堆纷繁复杂的问题中识别出真正要通过架构重构来解决的问题，集中力量快速解决，而不是想着通过架构重构来解决所有的问题

2.1.2. 合纵连横

所以在沟通协调时，将技术语言转换为通俗语言，以事实说话，以数据说话，是沟通的关键！ 推动重构和沟通的有效策略是“换位思考、合作双赢、关注长期

2.1.3. 运筹帷幄

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

2.2. 开源项目

2.2.1. 选择

满足业务：聚焦于是否满足业务，而不需要过于关注开源项目是否优秀。 技术成熟：尽量选择成熟的开源项目，降低风险 运维能力

2.2.2. 二次开发

保持纯洁，加以包装：不要改动原系统，而是要开发辅助系统：监控、报警、负载均衡、管理等 发明你要的轮子

2.3. App 架构

2.3.1. Web App

2.3.2. 原生 App

2.3.3. Hybrid App

2.3.4. 组件化 & 容器化

2.3.5. 跨平台 App

2.4. 架构设计文档模版

备选方案模板 1. 需求介绍[需求介绍主要描述需求的背景、目标、范围等]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[需求分析主要全方位地描述需求相关的信息]5W[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。如果是复杂的业务系统，这部分也可以独立成“用例文档”]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[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

3. 架构模式

3.1. 高性能架构

站在架构师的角度，当然需要特别关注高性能架构的设计。高性能架构设计主要集中在两方面： 尽量提升单服务器的性能，将单服务器的性能发挥到极致。 如果单服务器无法支撑性能，设计服务器集群方案。

3.1.1. 储存架构

- 关系数据库 集群

读写分离分散了数据库读写的压力 分库分表分散了数据库存储的压力

- 读写分离

读写分离：将数据库读写操作分散到不同的节点上。 读写分离的基本实现是：搭建：主从集群，一主一从、一主多从都可以。 分工：数据库主机负责读写操作，从机只负责读操作。 同步：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：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。

- 主从复制延迟

主从复制延迟会带来一个问题：如果业务服务器将数据写入到数据库主服务器后立刻（1 秒内）进行读取，此时读操作访问的是从机，主机还没有将数据复制过来，到从机读取数据是读不到最新数据的，业务上就可能出现问题。 解决主从复制延迟有几种常见的方法： 写操作后的读操作指定发给数据库主服务器 二次读取：读从机失败后再读一次主机 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

- 分配机制

将读写操作区分开来，然后访问不同的数据库服务器，一般有两种方式：程序代码封装和中间件封装。

- 程序代码封装（中间层封装）

程序代码封装（中间层封装）指在代码中抽象一个数据访问层，实现读写操作分离和数据库服务器连接的管理。程序代码封装的方式具备几个特点： 实现简单，而且可以根据业务做较多定制化的功能。 每个编程语言都需要自己实现一次，无法通用，如果一个业务包含多个编程语言写的多个子系统，则重复开发的工作量比较大。 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，实现读写操作分离和数据库服务器连接的管理。数据库中间件的方式具备的特点是： 能够支持多种编程语言，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的协议（例如，MySQL 客户端和服务器的连接协议），实现比较复杂，细节特别多，很容易出现 bug，需要较长的时间才能稳定。 数据库中间件自己不执行真正的读写操作，但所有的数据库操作请求都要经过中间件，中间件的性能要求也很高。 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。例如，向某个测试表写入一条数据，成功的就是主机，失败的就是从机。

- 分库分表

当数据量达到千万甚至上亿条的时候，单台数据库服务器的存储能力会成为系统的瓶颈，主要体现在这几个方面： 读写性能：数据量太大，读写的性能会下降，即使有索引，索引也会变得很大，性能同样会下降。 备份和恢复：数据文件会变得很大，数据库备份和恢复需要耗费很长时间。 风险：数据文件越大，极端情况下丢失数据的风险越高（例如，机房火灾导致数据库主备机都发生故障）。

- 业务分库

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。 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： 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。 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。 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。

- 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主要体现在表操作的数量要增加

- 水平分表

水平分表适合表行数特别大的表。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：水平分表后，某条数据具体属于哪个切分后的子表，需要增加路由算法进行计算，这个算法会引入一定的复杂性。（范围路由，配置路由，Hash 路由）

- 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 关系数据库在大数据场景下 I/O 较高 关系数据库的全文搜索功能比较弱 针对上述问题，分别诞生了不同的 NoSQL 解决方案，这些方案与关系数据库相比，在某些应用场景下表现更好。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，因此应该将 NoSQL 作为 SQL 的一个有力补充，NoSQL != No SQL，而是 NoSQL = Not Only SQL。

- 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 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。

- 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。

- 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。

3.1.2. 缓存架构

在某些复杂的业务场景下，单纯依靠存储系统的性能提升不够的，典型的场景有： 需要经过复杂运算后得出的数据，存储系统无能为力 读多写少的数据，存储系统有心无力

- 缓存穿透

缓存穿透：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。通常情况下有两种情况： 存储数据不存在 缓存数据生成耗费大量时间或者资源

- 缓存缺失值

缓存穿透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，这样第二次读取缓存时就会获取到默认值，而不会继续访问存储系统。

- 缓存雪崩

缓存雪崩：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。

- 更新锁机制

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。

- 后台更新机制

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。

- 缓存热点

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。

3.1.3. 单机高性能架构

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。

- 传统场景

  - Process Per Connection (PPC)

PPC: 每次有新的连接就新建一个进程去专门处理这个连接的请求，这是传统的 UNIX 网络服务器所采用的模型。基本的流程图是： 父进程接受连接（图中 accept）。 父进程 “fork” 子进程（图中 fork）。 子进程处理连接的读写请求（图中子进程 read、业务处理、write）。 子进程关闭连接（图中子进程中的 close）。互联网兴起后，服务器的并发和访问量从几十剧增到成千上万，这种模式的弊端就凸显出来了，主要体现在这几个方面： fork 代价高 父子进程通信复杂 支持的并发连接数量有限

  - prefork

prefork：提前创建进程（pre-fork）。系统在启动的时候就预先创建好进程，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 fork 进程的操作，让用户访问更快、体验更好。

  - Thread Per Connection (TPC)

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。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。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。 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）。

  - prethread

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。

- 高并发场景

  - Reactor (Dispatcher)

Reactor：根据事件类型来调用相应的代码进行处理，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，基于 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 模式有这三种典型的实现方案： 单 Reactor 单进程 / 线程。 单 Reactor 多线程。 多 Reactor 多进程 / 线程。

  - Proactor

Reactor 的非阻塞同步”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。

3.1.4. 集群高性能架构

- 负载均衡架构

  - DNS 负载均衡架构

DNS 解析同一个域名可以返回不同的 IP 地址。 优点有： 简单、成本低：负载均衡工作交给 DNS 服务器处理，无须自己开发或者维护负载均衡设备。 就近访问，提升访问速度：DNS 解析时可以根据请求来源 IP，解析成距离用户最近的服务器地址，可以加快访问速度，改善性能。 缺点有： 更新不及时：DNS 缓存的时间比较长，修改 DNS 配置后，由于缓存的原因，还是有很多用户会继续访问修改前的 IP，这样的访问会失败，达不到负载均衡的目的，并且也影响用户正常使用业务。 扩展性差：DNS 负载均衡的控制权在域名商那里，无法根据业务特点针对其做更多的定制化功能和扩展特性。 分配策略比较简单：DNS 负载均衡支持的算法少；不能区分服务器的差异（不能根据系统与服务的状态来判断负载）；也无法感知后端服务器的状态。

  - 硬件负载均衡架构

硬件负载均衡：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。 硬件负载均衡的优点

是： 功能强大：全面支持各层级的负载均衡，支持全面的负载均衡算法，支持全局负载均衡。 性能强大：对比一下，软件负载均衡支持到 10 万级并发已经很厉害了，硬件负载均衡可以支持 100 万以上的并发。 稳定性高：商用硬件负载均衡，经过了良好的严格测试，经过大规模使用，稳定性高。 支持安全防护：硬件均衡设备除具备负载均衡功能外，还具备防火墙、防 DDoS 攻击等安全功能。 硬件负载均衡的缺点是： 价格昂贵：最普通的一台 F5 就是一台“马 6”，好一点的就是“Q7”了。 扩展能力差：硬件设备，可以根据业务进行配置，但无法进行扩展和定制。

- 软件负载均衡架构

软件负载均衡：通过负载均衡软件来实现负载均衡功能。 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。

- 负载均衡典型架构

DNS 负载均衡用于实现地理级别的负载均衡； 硬件负载均衡用于实现集群级别的负载均衡； 软件负载均衡用于实现机器级别的负载均衡

- 负载均衡的分配算法

- 任务平分类

任务平分类：负载均衡系统将收到的任务平均分配给服务器进行处理，这里的“平均”可以是绝对数量的平均，也可以是比例或者权重上的平均。

- 负载均衡类

负载均衡类：负载均衡系统根据服务器的负载来进行分配，这里的负载并不一定是通常意义上我们说的“CPU 负载”，而是系统当前的压力，可以用 CPU 负载来衡量，也可以用连接数、I/O 使用率、网卡吞吐量等来衡量系统的压力。

- 性能最优类

性能最优类：负载均衡系统根据服务器的响应时间来进行任务分配，优先将新任务分配给响应最快的服务器。

- Hash 类

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。常见的有源地址 Hash、目标地址 Hash、session id hash、用户 ID Hash 等。 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。例如：源地址 Hash，ID Hash

3.2. 高可用架构

高可用存储方案的考虑因素： 数据如何复制？ 各个节点的职责是什么？ 如何应对复制延迟？ 如何应对复制中断？

3.2.1. 理论

- CAP

在一个分布式系统（指互相连接并共享数据的节点的集合）中，当涉及读写操作时，只能保证一致性（Consistence）、可用性（Availability）、分区容错性（Partition Tolerance）三者中的两个，另外一个必须被牺牲。 CAP 关注的是对数据的读写操作，而不是分布式系统的所有功能 CAP 是忽略网络延迟的。 正常运行情况下，不存在 CP 和 AP 的选择，可以同时满足 CA。 放弃并不等于什么都不做，需要为分区恢复后做准备。

- ACID

BASE 理论本质上是对 CAP 的延伸和补充，更具体地说，是对 CAP 中 AP 方案的一个补充： 基本可用（Basically Available）：分布式系统在出现故障时，允许损失部分可用性，即保证核心可用。 软状态（Soft State）：允许系统存在中间状态，而该中间状态不会影响系统整体可用性。这里的中间状态就是 CAP 理论中的数据不一致。 最终一致性（Eventual Consistency）：系统中的所有数据副本经过一定时间后，最终能够达到一致的状态。

- FMEA

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， FMEA 分析的方法其实很简单，就是一个 FMEA 分析表，常见的 FMEA 分析表格包含下面部分： 功能点 故障模式 故障影响（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） 严重程度（致命 / 高 / 中 / 低 / 无） 故障原因 故障概率（高 / 中 / 低） 风险程度 已有措施（检测告警、容错、自恢复等） 规避措施（技术手段-备份，管理手段-更换磁盘） 解决措施 后续规划（技术手段，管理手段，规避措施，解决措施）

3.2.2. 储存高可用架构

- 双机储存架构

主备复制和主从复制方案存在两个共性的问题： 主机故障后，无法进行写操作。如果主机无法恢复，需要人工指定新的主机角色。 双机切换就是为了解决这两个问题而产生的，包括主备切换和主从切换两种方案。

- 主备

整体架构比较简单，主备架构中的“备机”主要还是起到一个备份作用，并不承担实际的业务读写操作，如果要把备机改为主机，需要人工操作。综合主备复制架构的优缺点，内部的后台管理系统使用主备复制架构的情况会比较多，例如学生管理系统、员工管理系统、假期管理系统等，因为这类系统的数据变更频率低，即使在某些场景下丢失数据，也可以通过人工的方式补全。 主备复制架构的优点就是简单，表现有： 对于客户端来说，不需要感知备机的存在，即使灾难恢复后，原来的备机被人工修改为主机后，对于客户端来说，只是认为主机的地址换了而已，无须知道是原来的备机升级为主机。 对于主机和备机来说，双方只需要进行数据复制即可，无须进行状态判断和主备切换这类复杂的操作。 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

主机负责读写操作，从机只负责读操作，不负责写操作。综合主从复制的优缺点，一般情况下，写少读多的业务使用主从复制的存储架构比较多。例如，论坛、BBS、新闻网站这类业务 优缺点分析主从复制与主备复制相比，优点有： 主从复制在主机故障时，读操作相关的业务可以继续运行。 主从复制架构的从机提供读操作，发挥了硬件的性能。 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。

- 双机切换方法

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：状态传递的渠道、状态检测的内容 切换决策：切换时机、切换策略、自动程度 数据冲突解决

- 互连式

主备机直接建立状态传递的渠道

- 中介式

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

- 模拟式

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

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

- 集群储存架构

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。 数据集中集群架构中，客户端只能将数据写到主机；数据分散集群架构中，客户端可以向任意服务器中读写数据。正是因为这个关键的差异，决定了两种集群的应用场景不同。 数据集中集群适合数据量不大，集群机器数量不多的场景。例如，ZooKeeper 集群，一般推荐 5 台机器左右，数据量是单台服务器就能够支撑 而数据分散集群，由于其良好的可伸缩性，适合业务数据量巨大、集群机器数量庞大的业务场景。例如，Hadoop 集群、HBase 集群，大规模的集群可以达到上百台甚至上千台服务器。 数据分散集群和数据集中集群的不同点在于，数据分散集群中的每台服务器都可以处理读写请求，因此不存在数据集中集群中负责写的主机那样的角色。但在数据分散集群中，必须有一个角色来负责执行数据分配算法，这个角色可以是独立的一台服务器，也可以是集群自己选举出的一台服务器。如果是集群服务器选举出来一台机器承担数据分区分配的职责，则这台服务器一般也会叫作主机，但我们需要知道这里的“主机”和数据集中集群中的“主机”，其职责是有差异的。

- 数据集中集群

数据集中集群与主备、主从这类架构相似，我们也可以称数据集中集群为 1 主多备或者 1 主多从。无论是 1 主 1 从、1 主 1 备，还是 1 主多备、1 主多从，数据都只能往主机中写，而读操作可以参考主备、主从架构进行灵活多变。 由于集群里面的服务器数量更多，导致复杂度比双机架构更高： 主机如何将数据复制给备机：如何降低主机复制压力，或者降低主机复制给正常读写带来的压力 备机如何检测主机状态：在数据集中集群架构中，多台备机都需要对主机状态进行判断，而不同的备机判断的结果可能是不同的，如何处理不同备机对主机状态的不同判断，是一个复杂的问题 主机故障后，如何决定新的主机：而在数据集中集群架构中，有多台备

机都可以升级为主机，但实际上只能允许一台备机升级为主机，那么究竟选择哪一台备机作为新的主机，备机之间如何协调，这也是一个复杂的问题

- 数据分散集群

数据分散集群指多个服务器组成一个集群，每台服务器都会负责存储一部分数据；同时，为了提升硬件利用率，每台服务器又会备份一部分数据。 数据分散集群的复杂点在于如何将数据分配到不同的服务器上，算法需要考虑这些设计点： 均衡性：算法需要保证服务器上的数据分区基本是均衡的，不能存在某台服务器上的分区数量是另外一台服务器的几倍的情况。 容错性：当出现部分服务器故障时，算法需要将原来分配给故障服务器的数据分区分配给其他服务器。 可伸缩性：当集群容量不够，扩充新的服务器后，算法能够自动将部分数据分区迁移到新服务器，并保证扩容后所有服务器的均衡性。

- 分区方法

设计一个良好的数据分区架构，需要从多方面去考虑： 数据量：数据量越大，分区规则会越复杂，考虑的情况也越多 分区规则：地理位置有近有远，因此可以得到不同的分区规则，包括洲际分区、国家分区、城市分区。具体采取哪种或者哪几种规则，需要综合考虑业务范围、成本等因素。 复制规则：这部分数据如果损坏或者丢失，损失同样难以接受。因此即使是分区架构，同样需要考虑复制方案。

- 集中式

集中式备份指存在一个总的备份中心，所有的分区都将数据备份到备份中心，集中式备份架构的优缺点是： 设计简单，各分区之间并无直接联系，可以做到互不影响。 扩展容易，如果要增加第四个分区（例如，武汉分区），只需要将武汉分区的数据复制到西安备份中心即可，其他分区不受影响。 成本较高，需要建设一个独立的备份中心。

- 互备式

互备式备份指每个分区备份另外一个分区的数据，互备式备份架构的优缺点是： 设计比较复杂，各个分区除了要承担业务数据存储，还需要承担备份功能，相互之间互相关联和影响。 扩展麻烦，如果增加一个武汉分区，则需要修改广州分区的复制指向武汉分区，然后将武汉分区的复制指向北京分区。而原有北京分区已经备份了的广州分区的数据怎么处理也是个难题，不管是做数据迁移，还是广州分区历史数据保留在北京分区，新数据备份到武汉分区，无论哪种方式都很麻烦。 成本低，直接利用已有的设备。

- 独立式

独立式备份指每个分区自己有独立的备份中心，独立式备份架构的优缺点是： 设计简单，各分区互不影响。 扩展容易，新增加的分区只需要搭建自己的

备份中心即可。 成本高，每个分区需要独立的备份中心，备份中心的场地成本是主要成本，因此独立式比集中式成本要高很多。

3.2.3. 计算高可用架构

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。 计算高可用架构的设计复杂度主要体现在任务管理方面，即当任务在某台服务器上执行失败后，如何将任务重新分配到新的服务器进行执行。因此，计算高可用架构设计的关键点有下面两点：
哪些服务器可以执行任务 任务如何重新执行

- 双机储存架构

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。

- 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。主备方案的详细设计：主机执行所有计算任务。例如，读写数据、执行操作等。 当主机故障（例如，主机宕机）时，任务分配器不会自动将计算任务发送给备机，此时系统处于不可用状态。如果主机能够恢复（不管是人工恢复还是自动恢复），任务分配器继续将任务发送给主机。 如果主机不能够恢复（例如，机器硬盘损坏，短时间内无法恢复），则需要人工操作，将备机升为主机，然后让任务分配器将任务发送给新的主机（即原来的备机）；同时，为了继续保持主备架构，需要人工增加新的机器作为备机。 主备架构的优点就是 简单，主备机之间不需要进行交互，状态判断和切换操作由人工执行，系统实现很简单。 缺点： 人工操作，因为人工操作的时间不可控，可能系统已经发生问题了，但维护人员还没发现，等了 1 个小时才发现。发现后人工切换的操作效率也比较低，可能需要半个小时才完成切换操作，而且手工操作过程中容易出错。例如，修改配置文件改错了、启动了错误的程序等。 和存储高可用中的主备复制架构类似，计算高可用的主备架构也比较适合与内部管理系统、后台管理系统这类使用人数不多、使用频率不高的业务，不太适合在线的业务。

- 冷备架构

冷备：备机上的程序包和配置文件都准备好，但备机上的业务系统没有启动（注意：备机的服务器是启动的），主机故障后，需要人工手工将备机的业务系统启动，并将任务分配器的任务请求切换发送给备机。

- 温备架构

温备：备机上的业务系统已经启动，只是不对外提供服务，主机故障后，人工只需要将任务分配器的任务请求切换发送到备机即可。冷备可以节省一定的能源，但温备能够大大减少手工操作时间，因此一般情况下推荐用温备的方式。

- 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　主从方案详细设计： 正常情况下，主机执行部分计算任务（如图中的“计算任务 A”），备机执行部分计算任务（如图中的“计算任务 B”）。 当主机故障（例如，主机宕机）时，任务分配器不会自动将原本发送给主机的任务发送给从机，而是继续发送给主机，不管这些任务执行是否成功。 如果主机能够恢复（不管是人工恢复还是自动恢复），任务分配器继续按照原有的设计策略分配任务，即计算任务 A 发送给主机，计算任务 B 发送给从机。 如果主机不能够恢复（例如，机器硬盘损坏，短时间内无法恢复），则需要人工操作，将原来的从机升级为主机（一般只是修改配置即可），增加新的机器作为从机，新的从机准备就绪后，任务分配器继续按照原有的设计策略分配任务。　主从架构与主备架构相比，优点： 主从架构的从机也执行任务，发挥了从机的硬件性能。 缺点： 主从架构需要将任务分类，任务分配器会复杂一些。

- 集群储存架构

- 对称集群/负载均衡集群

对称集群、负载均衡集群：集群中每个服务器的角色都是一样的，都可以执行所有任务。　负载均衡集群详细设计： 正常情况下，任务分配器采取某种策略（随机、轮询等）将计算任务分配给集群中的不同服务器。 当集群中的某台服务器故障后，任务分配器不再将任务分配给它，而是将任务分配给其他服务器执行。 当故障的服务器恢复后，任务分配器重新将任务分配给它执行。　负载均衡集群的设计关键点在于两点： 任务分配器需要选取分配策略：轮询、随机 任务分配器需要检测服务器状态：检测服务器的状态、检测任务的执行状态。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

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。　非对称集群架构详细设计： 集群会通过某种方式来区分不同服务器的角色。例如，通过 ZAB 算法选举，或者简单地取当前存活服务器中节点 ID 最小的服务器作为 Master 服务器。 任务分配器将不同任务发送给不同服务器。例如，图中的计算任务 A 发送给 Master 服务器，计算任务 B 发送给 Slave 服务器。当指定类型的服务器故障时，需要重新分配角色。例如，Master 服务器故障后，需要将剩余的 Slave 服务器中的一个重新指定为 Master 服务器；如果是 Slave 服务器

故障，则并不需要重新分配角色，只需要将故障服务器从集群剔除即可。 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。

3.2.4. 业务高可用架构

- 异地多活架构

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。 实现异地多活架构的代价很高，具体表现为： 系统复杂度会发生质的变化，需要设计复杂的异地多活架构。 成本会上升，毕竟要多在一个或者多个机房搭建独立的一套业务系统。判别是否需要异地多活： 不需要：业务系统即使中断，对用户的影响并不会很大。如果无法承受异地多活带来的复杂度和成本，是可以不做异地多活的，只需要做异地备份即可。例如：新闻网站、企业内部的 IT 系统、游戏、博客站点等， 需要：业务系统中断后，对用户的影响很大。例如：共享单车、滴滴出行、支付宝、微信这类业务 异地多活设计的理念可以总结为一句话：采用多种手段，保证绝大部分用户的核心业务异地多活！跨城异地多活是架构设计复杂度最高的一种。跨城异地多活架构设计技巧和步骤： 保证核心业务的异地多活 保证核心数据最终一致性 采用多种手段同步数据 只保证绝大部分用户的异地多活 异地多活设计 4 步走： 业务分级（分级标准：访问量大的业务、核心业务、产生大量收入的业务） 数据分类（常见的数据特征分析维度：数据量、唯一性、实时性、可丢失性） 数据同步（常见的数据同步方案有：存储系统同步、消息队列同步、重复生成） 异常处理（常见的异常处理措施有这几类：多通道同步、同步和访问结合、日志记录、用户补偿）

- 同城异区

同城异区指的是将业务部署在同一个城市不同区的多个机房。例如，在北京部署两个机房，一个机房在海淀区，一个在通州区，然后将两个机房用专用的高速网络连接在一起。 同城异区是应对机房级别故障的最优架构

- 跨城异地

跨城异地指的是业务部署在不同城市的多个机房，而且距离最好要远一些。例如，将业务部署在北京和广州两个机房，而不是将业务部署在广州和深圳的两个机房。跨城异地多活的主要应用场景一般有这几种情况： 数据一致性要求不那么高，即延时影响不大 数据不怎么改变

- 跨国异地

跨国异地指的是业务部署在不同国家的多个机房。跨国异地多活的主要应用场景一般有这几种情况： 为不同地区用户提供服务 只读类业务做多活

3.2.5. 接口故障的解决方案

导致接口级故障的原因一般有下面几种： 内部原因：程序 bug 导致死循环，某个接口导致数据库慢查询，程序逻辑不完善导致耗尽内存等。 外部原因：黑客攻击、促销或者抢购引入了超出平时几倍甚至几十倍的用户，第三方系统大量请求，第三方系统响应缓慢等。 解决接口级故障的核心思想和异地多活基本类似： 优先保证核心业务 优先保证绝大部分用户

- 降级

降级的目的是应对系统自身的故障。降级是从系统功能优先级的角度考虑如何应对故障 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。 常见的实现降级的方式有： 系统后门降级：系统后门降级的方式实现成本低，但主要缺点是如果服务器数量多，需要一台一台去操作，效率比较低，这在故障处理争分夺秒的场景下是比较浪费时间的。 独立降级系统：将降级操作独立到一个单独的系统中，可以实现复杂的权限管理、批量操作等功能。

- 熔断

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

限流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。 常见的限流方式可以分为两类： 基于请求限流：从外部访问的请求角度考虑限流，常见的方式有：限制总量、限制时间量。 基于资源限流：基于资源限流是从系统内部考虑的，找到系统内部影响性能的关键资源，对其使用上限进行限制。常见的内部资源有：连接数、文件句柄、线程数、请求队列等。

- 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。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。 由于排队需要临时缓存大量的业务请求，单个系统内部无法缓存这么多数据，一般情况下，排队需要用独立的系统去实现，例如使用 Kafka 这类消息队列来缓存用户请求。

3.3. 可扩展架构

幸运的是，可扩展性架构的设计方法很多，但万变不离其宗，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 面向服务拆分 面向功能拆分 这几个系统架构并不是非此即彼的，而是可以在系统架构设计中进行组合使用的

3.3.1. 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。

- 分层架构

  分层架构设计最核心的一点就是需要保证各层之间的差异足够清晰，边界足够明显，让人看到架构图后就能看懂整个架构。 分层结构的特点： 隔离关注点（separation of concerns），即每个层中的组件只会处理本层的逻辑。 层层传递，也就是说一旦分层确定，整个业务流程是按照层进行依次传递的，不能在层之间进行跳跃 按照分层架构进行设计时，根据不同的划分维度和对象，可以得到多种不同的分层架构。 C/S 架构、B/S 架构：划分的对象是整个业务系统，划分的维度是用户交互，即将和用户交互的部分独立为一层，支撑用户交互的后台作为另外一层。 MVC 架构、MVP 架构划分的对象是单个业务子系统，划分的维度是职责，将不同的职责划分到独立层，但各层的依赖关系比较灵活。 逻辑分层架构划分的对象可以是单个业务子系统，也可以是整个业务系统，划分的维度也是职责。虽然都是基于职责划分，但逻辑分层架构和 MVC 架构、MVP 架构的不同点在于，逻辑分层架构中的层是自顶向下依赖的。 分层结构的优缺点： 好处：于强制将分层依赖限定为两两依赖，降低了整体系统复杂度 缺点：分层结构的代价就是冗余，也就是说，不管这个业务有多么简单，每层都必须要参与处理，甚至可能每层都写了一个简单的包装函数。 缺点：性能，因为每一次业务请求都需要穿越所有的架构分层，有一些事情是多余的，多少都会有一些性能的浪费。当然，这里所谓的性能缺点只是理论上的分析，实际上分层带来的性能损失

3.3.2. 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。 微服务并没有减少复杂度，而只是将复杂度从 ESB 转移到了基础设施。应用场景： SOA ：更加适合于庞大、复杂、异构的企业级系统，这也是 SOA 诞生的背景。这类系统的典型特征就是很多系统已经发展多年，采用不同的企业级技术，有的是内部开发的，有的是外部购买的，无法完全推倒重来或者进行大规模的优化和重构。因为成本和影响太大，只能采用兼容的方式进行处理，而承担兼容任务的就是 ESB。 微服务：更加适合于快速、轻量级、基于 Web 的互联网系统，这类系统业务变化快，需要快速尝试、快速交付；同时基本都是基于 Web，虽然开发技术可能差异很大（例如，Java、C++、.NET 等），但对外接口基本都是提供 HTTP RESTful 风格的接口，无须考虑在接口层进行类似 SOA 的 ESB 那样的处理。

- Service Oriented Architecture (SOA)

SOA 架构是比较高层级的架构设计理念，一般情况下我们可以说某个企业采用了 SOA 的架构来构建 IT 系统，但不会说某个独立的系统采用了 SOA 架构： 服务粒度：SOA 的服务粒度要粗一些，例如“员工管理系统”就是一个 SOA 架构中的服务 服务通信：SOA 采用了 ESB 作为服务间通信的关键组件，负责服务定义、服务路由、消息转换、消息传递，总体上是重量级的实现。 服务交付：微服务的架构理念要求“快速交付”，相应地要求采取自动化测试、持续集成、自动化部署等敏捷开发相关的最佳实践 SOA 出现 的背景是企业内部的 IT 系统重复建设且效率低下，主要体现在： 企业各部门有独立的 IT 系统，比如人力资源系统、财务系统、销售系统，这些系统可能都涉及人员管理，各 IT 系统都需要重复开发人员管理的功能。例如，某个员工离职后，需要分别到上述三个系统中删除员工的权限。 各个独立的 IT 系统可能采购于不同的供应商，实现技术不同，企业自己也不太可能基于这些系统进行重构。 随着业务的发展，复杂度越来越高，更多的流程和业务需要多个 IT 系统合作完成。由于各个独立的 IT 系统没有标准的实现方式（例如，人力资源系统用 Java 开发，对外提供 RPC；而财务系统用 C# 开发，对外提供 SOAP 协议），每次开发新的流程和业务，都需要协调大量的 IT 系统，同时定制开发，效率很低。 为了应对传统 IT 系统存在的问题，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。 Enterprise Service Bus (ESB)，中文翻译为“企业服务总线”。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。 松耦合：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。但实际上真正做到松耦合并没有那么容易，要做到完全后向兼容，是一项复杂的任务。优缺点： SOA 解决了传统 IT 系统重复建设和扩展效率低的问题， 但其本身也引入了更多的复杂性。SOA 最广为人诟病的就是 ESB，ESB 需要实现与各种系统间的协议转换、数据转换、透明的动态路由等功能。

- Microservices Architecture

微服务的特质微 small、lightweight、automated。微服务是一种和 SOA 相似但本质上不同的架构理念： 服务粒度：微服务的服务粒度要细一些，例如“员工管理系统”会被拆分为更多的服务，比如“员工信息管理” “员工考勤管理” “员工假期管理”和“员工福利管理”等更多服务。 服务通信：微服务推荐使用统一的协议和格式，例如，RESTful 协议、RPC 协议，无须 ESB 这样的重量级实现。 服务交付：SOA 对服务的交付并没有特殊要求，因为 SOA 更多考虑的是兼容已有的系统 如果以下场景都依赖人工去管理，整个系统将陷入一片混乱，最终的解决方案必须依赖自动化的服务管理系统，这时就会发现，微服务所推崇的“lightweight”，最终也发展成和 ESB 几乎一样的复杂程度： 服务划分过细，服务间关系复杂 服务数量太多，团队效率急剧下降 调用链太长，性能下降，问题定位困难 没有自动化支撑，无法快速交付 没有服务治理，微服务数量多了后管理混乱 常见的拆分方式： 基于业务逻辑拆分：将系统中的业务模块按照职责范围识别出来，每个单独的业务模块拆分为一个独立的服务。 基于可扩展拆分：将系统中的业务模块按照稳定性排序，将已经成熟和改动不大的服务拆分为稳定服务，将经常变化和迭代的服务拆分为变动服务

基于可靠性拆分：将系统中的业务模块按照优先级排序，将可靠性要求高的核心服务和可靠性要求低的非核心服务拆分开来，然后重点保证核心服务的高可用。 拆分带来的好处： 避免非核心服务故障影响核心服务 核心服务高可用方案可以更简单 能够降低高可用成本 基于性能拆分 开源的微服务基础设施全家桶了，例如 Spring Cloud 项目 按照下面优先级来搭建基础设施： 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。 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。 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。 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。 简单介绍一下每个基础设施的主要作用，更多详细设计可以参考 Spring Cloud 的相关资：https://projects.spring.io/spring-cloud/）

- 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。自动化测试涵盖的范围包括代码级的单元测试、单个系统级的集成测试、系统间的接口测试，理想情况是每类测试都自动化。如果因为团队规模和人力的原因无法全面覆盖，至少要做到接口测试自动化。

- 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。自动化部署系统包括版本管理、资源管理（例如，机器管理、虚拟机管理）、部署操作、回退操作等功能。

- 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。配置中心包括配置版本管理（例如，同样的微服务，有 10 个节点是给移动用户服务的，有 20 个节点给联通用户服务的，配置项都一样，配置值不一样）、增删改查配置、节点管理、配置同步、配置推送等功能。

- 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。例

如，我们需要指定接口协议为 HTTP/REST，但这还不够，还需要指定 HTTP/REST 的数据格式采用 JSON，并且 JSON 的数据都遵循如下规范。如果我们只是简单指定了 HTTP/REST 协议，而不指定 JSON 和 JSON 的数据规范，那么就会出现这样混乱的情况：有的微服务采用 XML，有的采用 JSON，有的采用键值对；即使同样都是 JSON，JSON 数据格式也不一样。这样每个微服务都要适配几套甚至几十套接口协议，相当于把曾经由 ESB 做的事情转交给微服务自己做了，这样做的效率显然是无法接受的，因此需要统一接口框架。接口框架不是一个可运行的系统，一般以库或者包的形式提供给所有微服务调用。例如，针对上面的 JSON 样例，可以由某个基础技术团队提供多种不同语言的解析包（Java 包、Python 包、C 库等）。

- 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。除此以外，外部系统访问系统还涉及安全和权限相关的限制，如果外部系统直接访问某个微服务，则意味着每个微服务都要自己实现安全和权限的功能，这样做不但工作量大，而且都是重复工作。综合上面的分析，微服务需要一个统一的 API 网关，负责外部系统的访问操作。API 网关是外部系统访问的接口，所有的外部系统接入系统都需要通过 API 网关，主要包括接入鉴权（是否允许接入）、权限控制（可以访问哪些功能）、传输加密、请求路由、流量控制等功能。

- 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。服务发现主要有两种实现方式：自理式和代理式。自理式结构就是指每个微服务自己完成服务发现。例如，图中 SERVICE INSTANCE A 访问 SERVICE REGISTRY 获取服务注册信息，然后直接访问 SERVICE INSTANCE B。自理式服务发现实现比较简单，因为这部分的功能一般通过统一的程序库或者程序包提供给各个微服务调用，而不会每个微服务都自己来重复实现一遍；并且由于每个微服务都承担了服务发现的功能，访问压力分散到了各个微服务节点，性能和可用性上不存在明显的压力和风险。 代理式结构就是指微服务之间有一个负载均衡系统（图中的 LOAD BALANCER 节点），由负载均衡系统来完成微服务之间的服务发现。代理式的方式看起来更加清晰，微服务本身的实现也简单了很多，但实际上这个方案风险较大。第一个风险是可用性风险，一旦 LOAD BALANCER 系统故障，就会影响所有微服务之间的调用；第二个风险是性能风险，所有的微服务之间的调用流量都要经过 LOAD BALANCER 系统，性能压力会随着微服务数量和流量增加而不断增加，最

后成为性能瓶颈。因此 LOAD BALANCER 系统需要设计成集群的模式，但 LOAD BALANCER 集群的实现本身又增加了复杂性。不管是自理式还是代理式，服务发现的核心功能就是服务注册表，注册表记录了所有的服务节点的配置和状态，每个微服务启动后都需要将自己的信息注册到服务注册表，然后由微服务或者 LOAD BALANCER 系统到服务注册表查询可用服务。

- 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。服务路由和服务发现紧密相关，服务路由一般不会设计成一个独立运行的系统，通常情况下是和服务发现放在一起实现的。对于自理式服务发现，服务路由是微服务内部实现的；对于代理式服务发现，服务路由是由 LOAD BALANCER 系统实现的。无论放在哪里实现，服务路由核心的功能就是路由算法。常见的路由算法有：随机路由、轮询路由、最小压力路由、最小连接数路由等。

- 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。专栏第 34 期我在分析微服务陷阱时提到微服务具有故障扩散的特点，如果不及时处理故障，故障扩散开来就会导致看起来系统中很多服务节点都故障了，因此需要微服务能够自动应对这种出错场景，及时进行处理。否则，如果节点一故障就需要人工处理，投入人力大，处理速度慢；而一旦处理速度慢，则故障就很快扩散，所以我们需要服务容错的能力。常见的服务容错包括请求重试、流控和服务隔离。通常情况下，服务容错会集成在服务发现和服务路由系统中。

- 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。举个简单例子：我们收到用户投诉说业务有问题，如果此时采取人工的方式去搜集、分析信息，可能把几十个节点的日志打开一遍就需要十几分钟了，因此需要服务监控系统来完成微服务节点的监控。服务监控的主要作用有：实时搜集信息并进行分析，避免故障后再来分析，减少了处理时间。服务监控可以在实时分析的基础上进行预警，在问题萌芽的阶段发觉并预警，降低了问题影响的范围和时间。通常情况下，服务监控需要搜集并分析大量的数据，因此建议做成独立的系统，而不要集成到服务发现、API 网关等系统中。

- 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。服务监控和服务跟踪的区别可以简单概括为宏观和微观的区别。例如，A 服务通过 HTTP 协议请求 B 服务 10 次，B 通过 HTTP 返回 JSON 对象，服务监控会记录请求次数、响应时间平均值、响应时间最高值、错误码分布这些信息；而服务跟踪会记录其中某次请求的发起时间、响应时间、响应错误码、请求参数、返回的 JSON 对象等信息。目前无论是分布式跟踪还是微服务的服务跟踪，绝大部分请求跟踪的实现技术都基于 Google 的 Dapper 论文《Dapper, a Large-Scale Distributed Systems Tracing Infrastructure》。

- 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。服务安全主要分为三部分：接入安全、数据安全、传输安全。通常情况下，服务安全可以集成到配置中心系统中进行实现，即配置中心配置微服务的接入安全策略和数据安全策略，微服务节点从配置中心获取这些配置信息，然后在处理具体的微服务调用请求时根据安全策略进行处理。由于这些策略是通用的，一般会把策略封装成通用的库提供给各个微服务调用。

3.3.3. 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。

- Microkernel Architecture

微内核架构（Microkernel Architecture），也被称为插件化架构（Plug-in Architecture），是一种面向功能进行拆分的可扩展性架构，通常用于实现基于产品（原文为 product-based，指存在多个版本、需要下载安装才能使用，与 web-based 相对应）的应用。例如 Eclipse 这类 IDE 软件、UNIX 这类操作系统、淘宝 App 这类客户端软件等，也有一些企业将自己的业务系统设计成微内核的架构，例如保险公司的保险核算逻辑系统，不同的保险品种可以将逻辑封装成插件。 常见的两种微内核具体实现： OSGi 架构 规则引擎架构

- 核心系统（core system）

核心系统负责和具体业务功能无关的通用功能，例如模块加载、模块间通信等。微内核的核心系统设计的关键技术有： 插件管理：插件注册表机制 插件连接：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：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。

- 插件模块（plug-in modules）

插件模块负责实现具体的业务逻辑，例如专栏前面经常提到的“学生信息管理”系统中的“手机号注册”功能。

4. 基础架构

4.1. 定义

4.1.1. 模块与组件

从逻辑的角度来拆分系统后，得到的单元就是“Module 模块”；从物理的角度来拆分系统后，得到的单元就是“component 组件”。

4.1.2. 框架与架构

Framework 框架关注的是“规范”，Architecture 架构关注的是“结构”

4.2. 复杂度来源

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。

4.2.1. 高性能

高性能增加机器目的在于“扩展”处理性能。 单台计算机内部为了高性能带来的复杂度。 操作系统的历史演变： 手工操作 批处理 操作系统 串行的进程 操作系统 包含并行线程的进程 操作系统 多台计算机集群为了高性能带来的复杂度： 任务分配 任务分解：简单的系统更加容易做到高性能；可以针对单个任务进行扩展

4.2.2. 高可用

高可用增加机器目的在于“冗余”处理单元。 不同高可用的应用场景： 计算高可用：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 存储高可用：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。 高可用状态决策：无论是计算高可用还是存储高可用，其基础都是“状态决策”，即系统需要能够判断当前的状态是正常还是异常，如果出现了异常就要采取行动来保证高可用。常见的决策方式： 独裁式：1 决策+nshang bao 协商式：主机+备机 民主式

4.2.3. 可扩展性

预测变化的复杂性在于： 不能每个设计点都考虑可扩展性。 不能完全不考虑可扩展性。所有的预测都存在出错的可能性。 应对变化： 将“变化”封装在一个“变化层”，将不变的部分封装在一个独立的“稳定层”。 提炼出一个“抽象层”和一个“实现层”。抽象层是稳定的，实现层可以根据具体业务需要定制开发，当加入新的功能时，只需要增加新的实现，无须修改抽象层。 JAVA 的 23 种设计模式

4.2.4. 低成本

低成本本质上是与高性能和高可用冲突的，所以低成本很多时候不会是架构设计的首要目标，而是架构设计的附加约束。 引入新技术的主要复杂度：需要去熟悉新技术，并且将新技术与已有技术结合起来； 创造新技术的主要复杂度：需要自己去创造全新的理念和技术，并且新技术跟旧技术相比，需要有质的飞跃

4.2.5. 安全

从技术的角度来讲，安全可以分为两类： 功能上的安全：“防小偷”。本质上是因为系统实现有漏洞，黑客有了可乘之机。框架只能预防常见的安全漏洞和风险（常见的 XSS 攻击、CSRF 攻击、SQL 注入等），无法预知新的安全问题，而且框架本身很多时候也存在漏洞。 架构上的安全：“防强盗”。传统的架构安全主要依靠防火墙，防火墙最基本的功能就是隔离网络，通过将网络划分成不同的区域，制定出不同区域之间的访问控制策略来控制不同信任程度区域间传送的数据流。

4.2.6. 规模

规模带来复杂度的主要原因就是“量变引起质变”，当数量超过一定的阈值后，复杂度会发生质的变化。常见的规模带来的复杂度有： 功能越来越多，导致系统复杂度指数级上升 数据越来越多，系统复杂度发生质变

4.3. 架构设计三原则

即使是现在非常复杂、非常强大的架构，也并不是一开始就进行了复杂设计，而是 首先采取了简单的方式（简单原则）， 满足了当时的业务需要（合适原则）， 随着业务的发展逐步演化而来的（演化原则）。

4.3.1. 合适原则

合适原则宣言：“合适优于业界领先”。

4.3.2. 简单原则

简单原则宣言：“简单优于复杂”。

4.3.3. 演化原则

演化原则宣言：“演化优于一步到位”。

4.4. 架构设计流程

4.4.1. 识别复杂度

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。

4.4.2. 设计备选方案

备选方案的数量以 3 ~ 5 个为最佳。 备选方案的差异要比较明显。 备选方案的技术不要只局限于已经熟悉的技术

4.4.3. 评估和选择备选方案

按优先级选择：即架构师综合当前的业务发展情况、团队人员规模和技能、业务发展预测等因素，将质量属性按照优先级排序，首先挑选满足第一优先级的，如果方案都满足，那就再看第二优先级……以此类推 列出我们需要关注的质量属性点，然后分别从这些质量属性的维度去评估每个方案，再综合挑选适合当时情况的最优方案。 在评估这些质量属性时，需要遵循架构设计原则 1“合适原则”和原则 2“简单原则”，避免贪大求全，基本上某个质量属性能够满足一定时期内业务发展就可以了。 常见的方案质量属性点有：性能、可用性、硬件成本、项目投入、复杂度、安全性、可扩展性等。

4.4.4. 详细方案设计

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。通过分步骤、分阶段、分系统等方式，尽量降低方案复杂度， 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。