

# 架构师笔记

---

## 1. 架构到底是指什么

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

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

模块和组件都是系统的组成部分，只是从不同的角度划分系统而已。从逻辑的角度划分系统后，得到的单元就是“模块”，从物理的角度来拆分系统后，得到的单元就是“组件”。划分模块是为了职责分离，划分组件是为了单元复用。

以一个简单的网站系统为例。假设我们要做一个学生信息管理系统。这个系统按逻辑的角度来拆分，可以分为“登录注册模块”“个人信息模块”“个人成绩模块”，从物理的角度来拆分，可以拆分为nginx， web服务器， Mysql。

框架与架构：

框架是一个组件规范，是提供基础功能的产品。

软件架构值软件系统的“基础结构”，创造这些基础结构的准则，以及对这些结构的描述。框架关心的是规范，架构关心的是结构。

升华——软件架构是指系统的顶层结构。

## 2. 架构设计的历史背景

### (1) 软件开发的历史：

机器语言（1940年之前），直接用0、1进行编码，在8086机器上完成“ $s=768+12288-1280$ ”的数学运算，机器码如下：

```
1011000000000000000000000011  
000001010000000000110000  
0010110100000000000000101
```

汇编语言（20世纪40年代）

高级语言（20世纪50年代）

## （2）软件危机

### 第一次软件危机与结构化程序设计（20世纪 60 年代~20世纪 70 年代）

有了高级语言以后，随着软件的规模和复杂度的大大增加，20世纪60年代中期开始爆发了第一次软件危机，典型的表现有软件质量低下、项目无法如期完成、项目严重超支等。例如：1963年美国（[http://en.wikipedia.org/wiki/Mariner\\_1](http://en.wikipedia.org/wiki/Mariner_1)）的水手一号火箭发射失败事故，就是因为一行 FORTRAN 代码错误导致的。

提出了“软件工程”和“结构化程序设计”来解决问题。

### 第二次软件危机与面向对象（20世纪 80 年代）

因为业务需求越来越复杂，以及编程应用领域越来越广泛。根本原因还在于软件生产力远远跟不上硬件和业务的发展。

第一次软件危机的根源在于软件的“逻辑”变得非常复杂，而第二次软件危机主要体现在软件的“扩展”变得非常复杂。

面向对象的思想并不是在第二次软件危机后才出现的，早在 1967 年的 Simula 语言中就开始提出来了，但第二次软件危机促进了面向对象的发展。面向对象真正开始流行是在 20 世纪 80 年代，主要得益于 C++ 的功劳，后来的 Java、C# 把面向对象推向了新的高峰。到现在为止，面向对象已经成为了主流的开发思想。

## 软件架构的历史背景

软件架构真正流行却是从 20 世纪 90 年代开始的，由于在 Rational 和 Microsoft 内部的相关活动，软件架构的概念开始越来越流行了。

与之前的各种新方法或者新理念不同的是，“软件架构”出现的背景并不是整个行业都面临类似相同的问题，“软件架构”也不是为了解决新的软件危机而产生的，这是怎么回事呢？

随着软件系统规模的增加，计算相关的算法和数据结构不再构成主要的设计问题；当系统由许多部分组成时，整个系统的组织，也就是所说的“软件架构”，导致了一系列新的设计问题。（《软件架构介绍》）

这段话很好地解释了“软件架构”为何先在 Rational 或者 Microsoft 这样的大公司开始逐步流行起来。因为只有大公司开发的软件系统才具备较大规模，而只有规模较大的软件系统才会面临软件架构相关的问题，例如：

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。

### 3. 架构设计的目的

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。

#### 几个问题

“这么多需求，从哪开始下手开始进行架构设计？”

re: 通过熟悉和理解需求，识别系统复杂性所在的地方，然后针对这些复杂点进行架构设计。

“架构设计要考虑高性能、高可用、高扩展……这么多高XX，全部设计完成估计要1个月，但是老大只给了一周时间？”

re: 架构设计并不是要面面俱到，不需要每个架构都具备高性能、高可用、高扩展等特点，而是要识别出复杂点然后针对性地解决问题。

“业界A公司的架构是X，B公司的方案是Y，两个差别比较大，该参考哪一个呢？”

re: 理解每个架构背后所要解决的复杂点，然后对比自己业务的复杂点，参考复杂点相似的方案。

#### 识别复杂性的几个方面

性能、可扩展性、高可用、安全性、成本

#### 一些有用的观点

我觉得做软件架构就是为了两件事服务的：业务架构和业务量级，这应该算是“软件复杂度所带来的问题”的具体化吧。

业务架构是对业务需求的提炼和抽象，开发软件必须要满足业务需求，否则就是空中楼阁。软件系统业务上的复杂问题，可以从业务架构的角度切分工作交界面来解决。设计软件架构，首先要保证能和业务架构对的上，这也是从业务逻辑转向代码逻辑的过程，所以软件架构的设计为开发指明了方向。另外架构设计也为接下来的开发工作分工奠定了基础。

业务量级表现在存储能力、吞吐能力和容错能力等，主要是软件运维期的复杂度。做软件架构设计，是要保证软件有能力托起它在业务量级上的要求的。如果软件到运行使用期废了，前面所有的工作都付诸东流了。不同的业务量级，对应的软件的架构复杂度是不同的，所以对于不同的项目，业务量及不同，架构设计也不同。

## 架构是为了解决软件复杂度而提出的一个解决方案

架构即（重要）决策，是在一个有约束的盒子里去求解或接近更合适的解。这个有约束的盒子是团队经验，成本，资源，进度，业务所处阶段等所编织、掺杂在一起的综合体（人、财、物、时间、事情等）。架构无优劣，但是存在恰当的架构用在合适的软件系统中，而这些就是决策的结果。

## 4. 复杂度来源-高性能

软件系统中高性能带来的复杂度主要体现在两方面，一方面是单台计算机内部为了高性能带来的复杂度；另一方面是多台计算机集群为了高性能带来的复杂度。

### （1）单机复杂度

计算机内部复杂度最关键的地方就是操作系统：操作系统是软件系统的运行环境，操作系统的复杂度直接决定了软件系统的复杂度。

而操作系统和性能最相关的就是进程和线程。在现代操作系统中，进程是一个操作系统分配资源的最小单位；线程是进程内部的子任务，但这些子任务都共享同一份进程数据，线程是操作系统调度的最小单位。

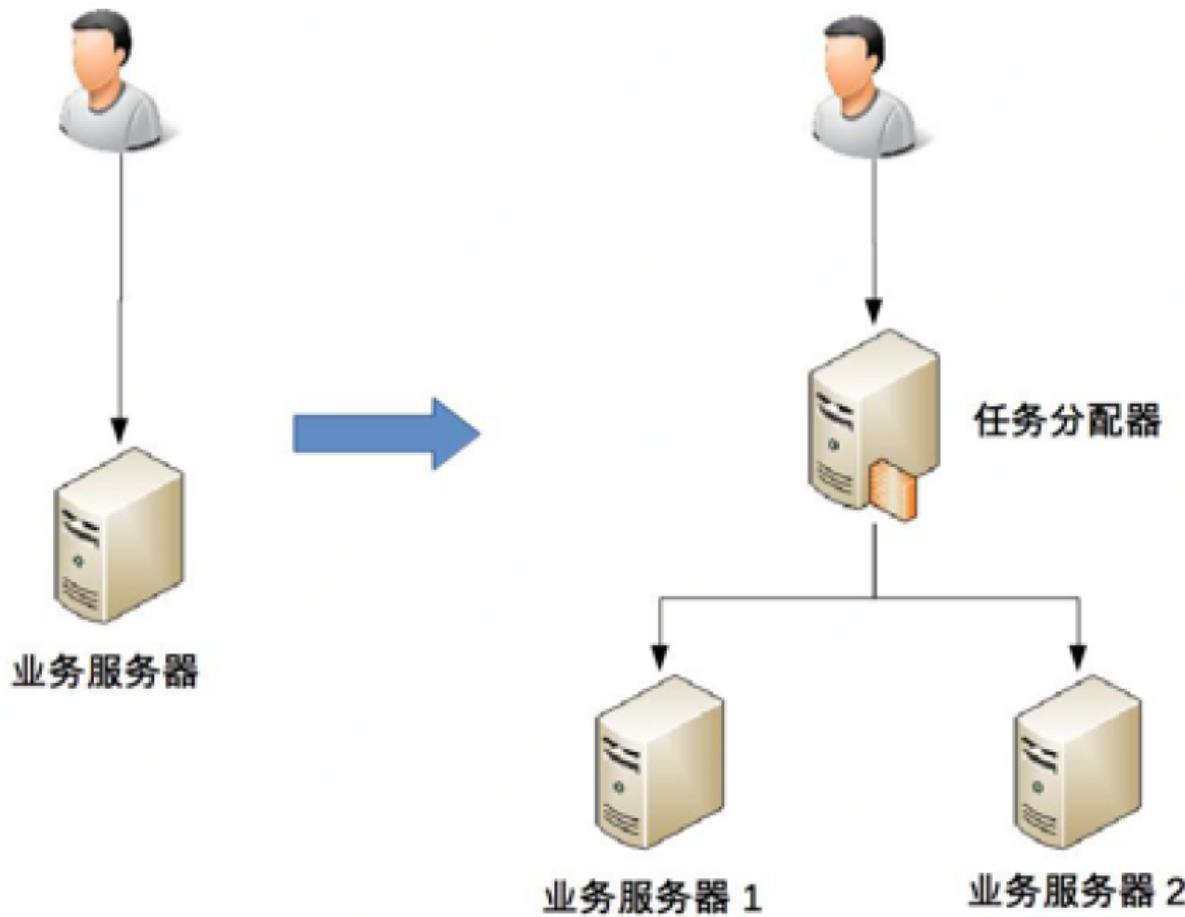
操作系统发展到现在，如果我们要完成一个高性能的软件系统，需要考虑诸如多线程、多进程、进程间通信、多线程并发等技术点，而且这些技术并不是最新的就是最好的，也不是非此即彼的选择。在做架构的时候，需要花费很大的精力来结合业务进行分析、判断、选择、组合，这个过程同样很复杂。

## (2) 集群的复杂度

通过大量机器来提升性能，并不仅仅是增加机器这么简单，让多台机器配合起来达到高性能的目的，是一个复杂的任务，有下面几种常见的方法。

### 任务分配

任务分配的意思是指每台机器都可以处理完整的业务任务，不同的任务分配到不同的机器上执行。

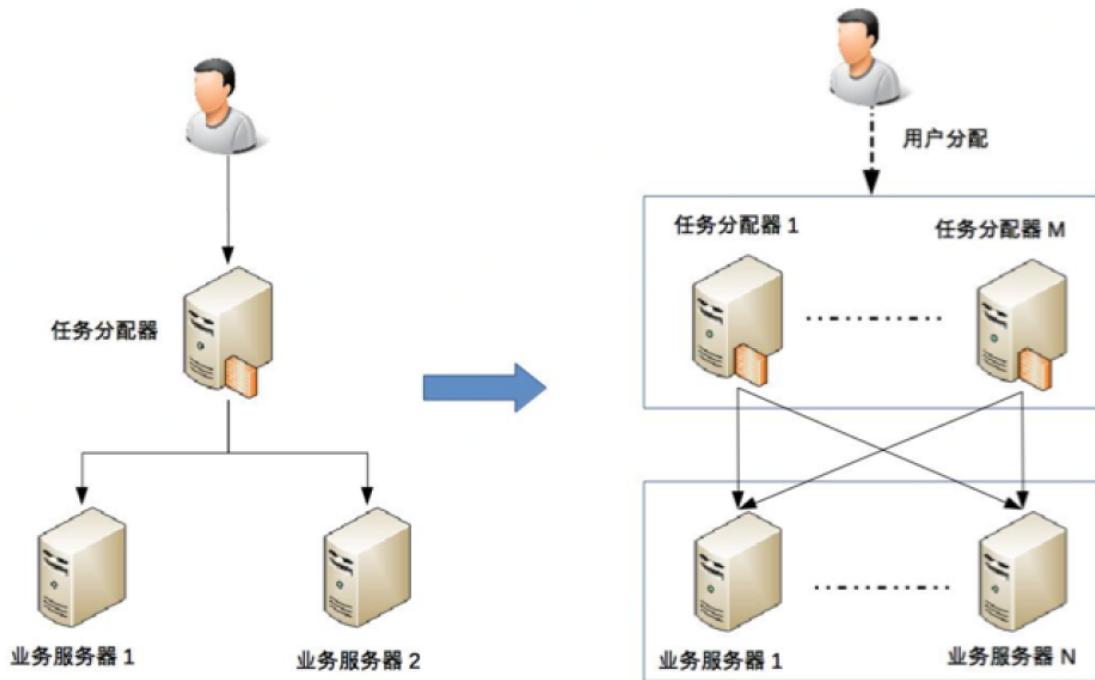


架构明显变复杂了，主要体现在：

- 需要增加一个任务分配器，这个任务分配器可能是硬件设备（例如，F5、交换机等），可能是软件网络设备（例如,LVS），也可能是负载均衡软件（例如，Nginx, HAProxy），还可能是自己开发的系统。选择合适的任务分配器也是一件复杂的事情，需要综合考虑性能、成本、可维护性、可用性等各方面的因素。
- 任务分配器和真正的业务系统之间有连接和交互，需要选择合适的连接方式，并对连接进行管理。例如连接建立、连接检测、连接中断后如何处理等。

- 任务分配器需要增加分配算法。例如，是采用轮询算法，还是按权重分配，又或者按照负载进行分配。如果按照服务器的负载进行分配，则业务服务器还要能够上报自己的状态给任务分配器。

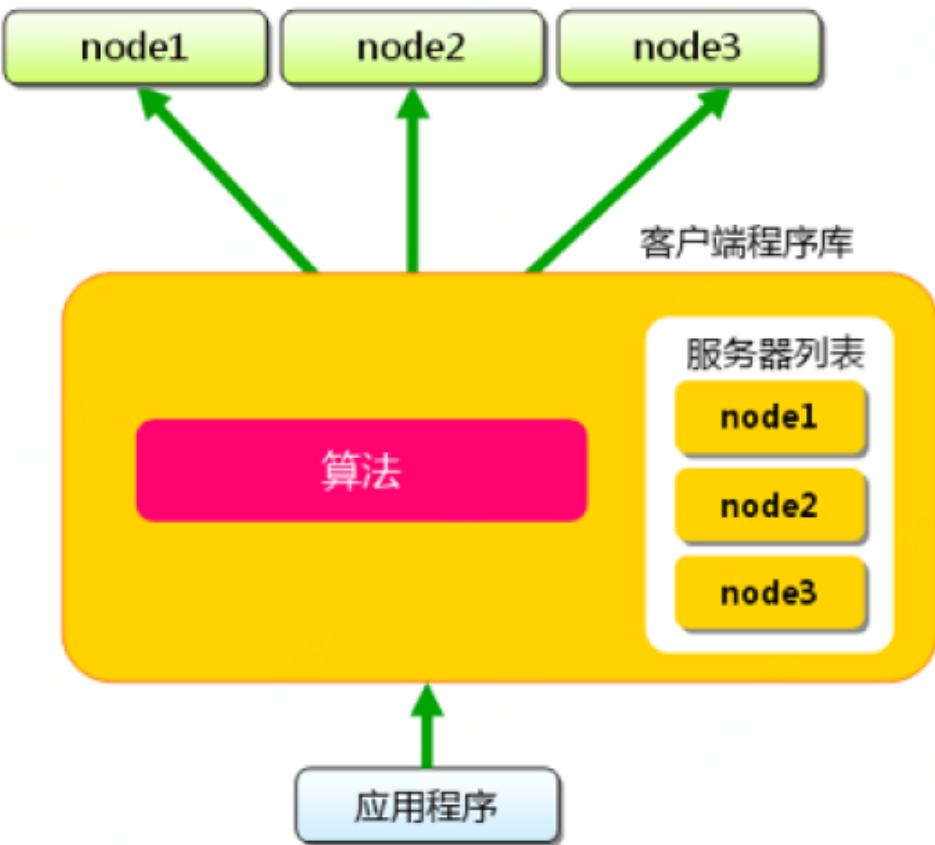
然而随着业务的增长，任务分配器本身又会成为瓶颈，这是任务分配器本身也需要扩展为多台机器。



这里的架构明显更加复杂：

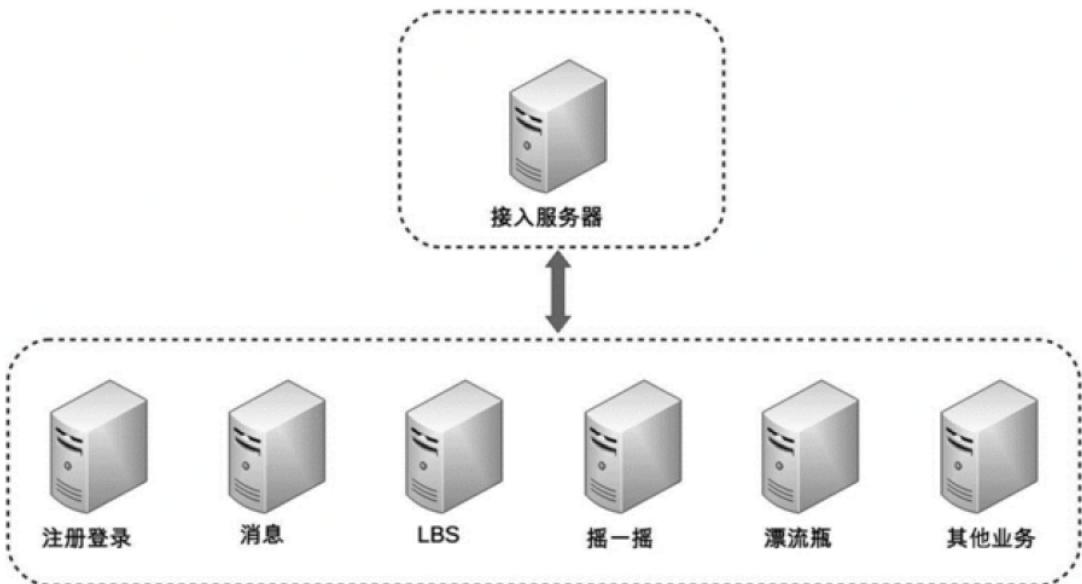
- 任务分配器从1台变成了多台，这个变化带来的复杂度就是需要将不同的用户分配到不同的任务分配器上，常见的方法包括DNS轮询、智能DNS、CDN(Content Delivery Network)、GSLB设备（Global Server Load Balance）等。
- 任务分配器和业务服务器的连接从简单的“1对多”变成了“多对多”的网状结构。
- 机器数量增加，状态管理、故障处理复杂度也大大增加。

“任务分配器”也并不一定只能是物理上存在的机器或者一个独立运行的程序，也可以是嵌入在其他程序中的算法；例如Memcache的集群架构。



## 任务分解

“业务服务器”如果越来越复杂，我们可以将其拆分为更多的组成部分。下图以微信为例。



从业务的角度来看，任务分解既不会减少功能，也不会减少代码量（事实上代码量可能还会增加，因为从代码内部调用改为通过服务之间的接口进行调用），那为何通过任务分解就能够提升性能呢？

主要有几方面的原因：

- 简单的系统更加容易做到高性能

复杂的系统难以找到关键性能点，就算话大力气找到了，优化起来也不容易（很可能在A点提高了性能，却在B点降低了性能）。

- 可以针对单个任务进行扩展

例如，如果用户增长过快，只需要优化注册登录子系统的性能，其他系统不需要修改。

然而任务并不是拆分的越小越好，因为任务在网络上的rt时间会随着任务数的增加而增加，最后反而会导致请求时间变长。同时任务数过多，也会为维护工作带来困难。

## 一些有用的观点

性能&伸缩性：性能更多的是衡量软件系统处理一个请求或者执行一个任务需要耗费的时间长短；而伸缩性则更加关注软件在不影响用户体验的前提下，能够随着请求数量或者执行任务数量的增加（减少）而相应地拥有相适应的处理能力。

## 5. 复杂度来源-高可用

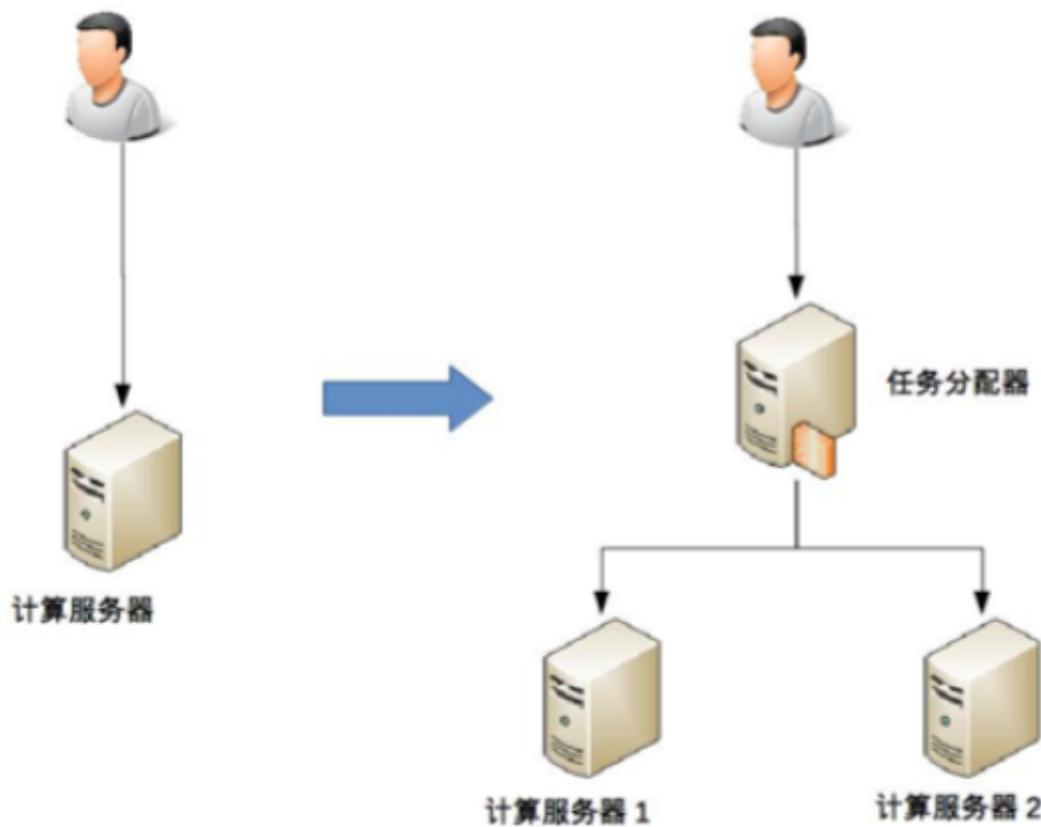
维基百科定义：系统无中断地执行其功能的能力，代表系统可用性程度，是进行系统设计时的准则之一。

可以明确，关键点和难点都在“无中断”上，然而单点（单个软件、单个硬件）很难保证“无中断”，所以系统的高可用方案五花八门，但万变不离其宗，本质上都是通过“冗余”来实现高可用。

高可用和高性能都是增加机器，但是本质是有不同的，高性能增加机器的目的是为了“扩展”处理能力，高可用增加机器的目的在于“冗余”处理单元。

### 1. 计算高可用

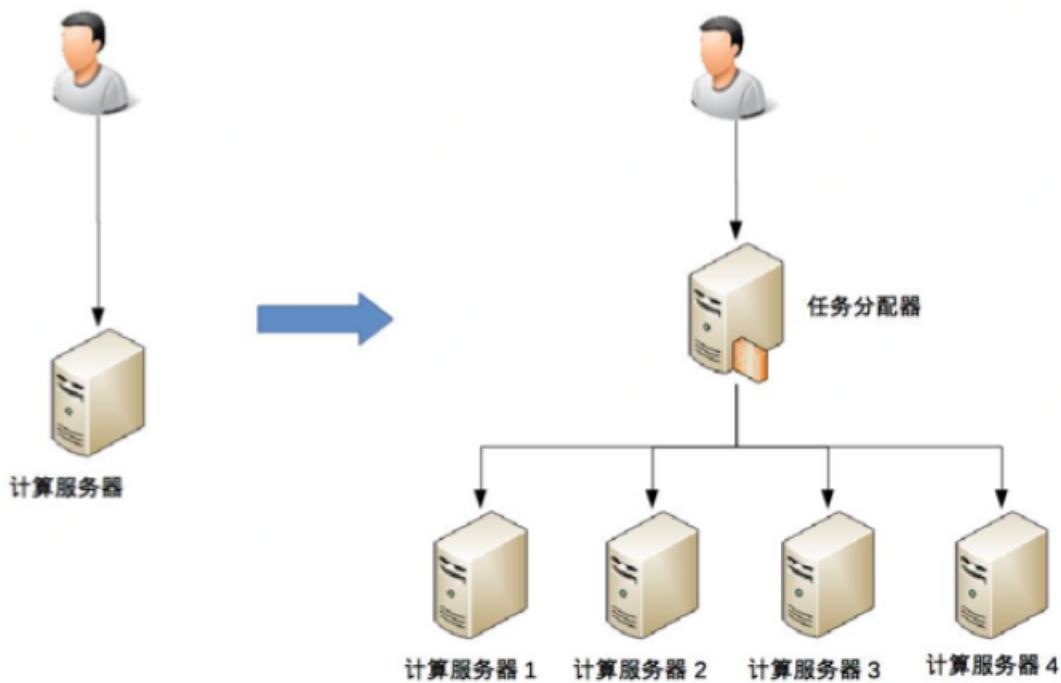
这里的“计算”指的是业务逻辑的处理。计算有一个特点就是无论在哪台机器上进行计算，同样的算法和输入数据，产出的结果都是一样的，所以将计算从一台机器迁移到另一台机器，对业务并没有什么影响。



这和“高性能”中讲到的双机架构是一样的，因此复杂度也是类似的。

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

一个更复杂的高可用架构：



这个高可用集群相比双机来说，分配算法更加复杂，可以是1主3备、2主2备、3主1备、4主0备，具体采用哪种方式，需要结合实际业务和需求来进行判断，并不存在某种算法就一定优于另外的算法。

## 2. 存储高可用

存储与计算相比，有一个本质的区别：将数据从一台机器搬到另一台机器，需要经过线路进行传输。线路传输的速度是毫秒级别，同一机房内部能够做到几毫秒，分布在不同地方的机房，传输耗时需要几十甚至上百毫秒。

这就意味着在某个时间点，整个系统中的数据肯定是不一致的。按照“数据”+“逻辑”=“业务”的公式来进行套用的话，如果数据不一致，就算是逻辑一致，也会导致业务不一致。

除了物理上的传输速度的限制，传输线路本身也存在可用性问题，传输线路可能中断、可能拥塞、可能异常（错报、丢包），而且这些异常情况一般持续时间是比较长的。

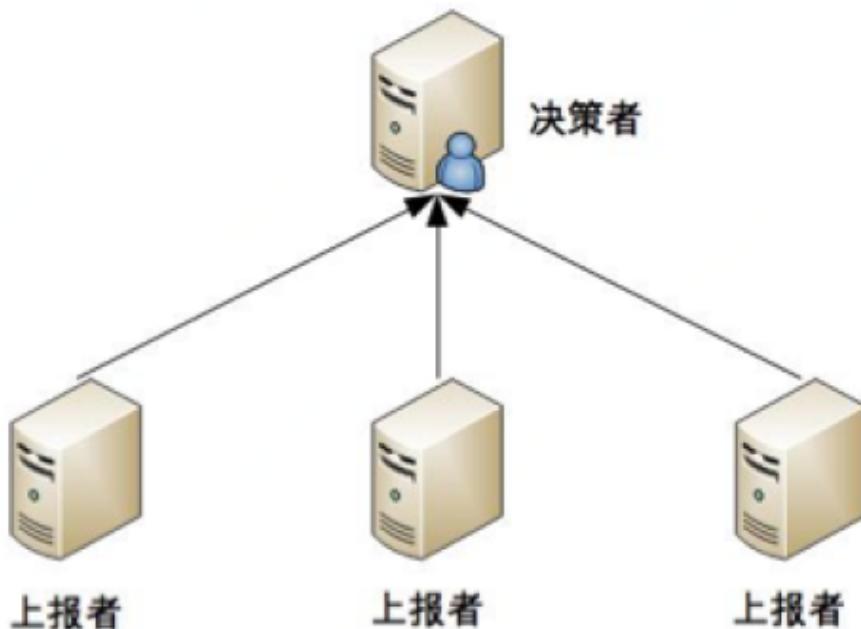
存储高可用的难点不在于如何备份数据，而在于如何减少或者规避数据不一致对业务造成的影响。

分布式领域的**CAP**理论就是从理论上论证了存储高可用的复杂性。

### 3. 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”；如果状态决策本身都是有错误或者有偏差的，那么后续的任何行动和处理无论多么完美也都没有意义和价值。但是在具体的实践过程中，恰好存在一个本质的矛盾：通过冗余来实现的高可用系统，状态决策本质上就不可能做到完全正确。

#### 独裁式



独裁的决策方式不会出现决策混乱的问题，因为只有一个决策者，但问题也恰好是因为只有一个决策者。当决策者故障时，整个系统就无法实现准确的状态决策。如果决策者本身又做一套状态决策，那就陷入了一个递归的死循环。

#### 协商式

协商式决策指的是两个独立的个体通过交流信息，然后根据规则进行决策，最常用的协商式决策就是主备决策。

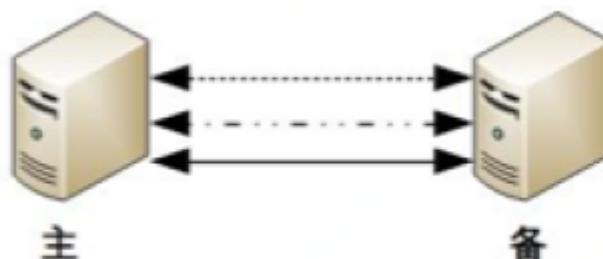


这个架构的基本协商规则可以设计成：

- 2台服务器启动时都是备机
- 2台服务器建立连接
- 2台服务器交换状态信息
- 某1台服务器根据规则胜出，成为主机；另一台服务器继续保持备机身份

协商式决策的架构不复杂，规则也复杂，其难点在于，如果两者的信息交换出现问题（比如主备连接中断），此时状态决策应该怎么做？

- 如果备机在主机断开连接的情况下认为主机故障，那么备机需要升级为主机，但实际上此时主机并没有故障，那么系统就会出现两个主机，这与1主1备的设计初衷是不符合的。
- 如果备机在主机断开连接的情况下不认为主机故障，则此时如果主机真的发生了故障，那么系统就没有主机了，这同样与1主1备的设计初衷是不符合的。
- 为了规避以上的问题，可以在主备之间建立多条连接。例如双连接、三连接。但这样有引入了这几条连接如何取舍的问题，即如果不同连接传递的信息不同，应该以哪个连接为准？实际上这也是一个无解的答案，无论以哪个连接为准，在特定的场景下都可能存在问题。



所以，总体看来，协商式决策在某些场景总是存在一些问题的。

## 民主式

民主式决策是指多个独立个体通过投票的方式来进行状态决策。

民主式决策和协商式决策类似，其基础都是独立个体之间的信息交换，每个个体都做出自己的决策，然后按照“多数取胜”的规则来确定最终的状态。不同点在于民主式决策比协商式决策要复杂的多。

除了算法复杂，民主式决策还有一个固有缺陷：脑裂。

综合分析，无论采用什么样的方案，状态决策都不可能在任何场景下都没有问题，但是完全不做高可用方案又会产生更大的问题，如何选取适合系统的高可用方案，也是一个复杂的分析、判断和选择的过程。

## 6. 复杂度来源：可扩展性

可扩展性指的是系统的一种应对将来需求变化而提供的一种扩展能力，当新需求出现时，系统不需要或者只需要少量的修改就可以支持，无须整个系统重构或重建。

一个具备良好可扩展性的架构或软件应当符合开闭原则：对扩展开放，对修改关闭。衡量一个软件的扩展性主要表现在但不限于：（1）软件自身内部方面。在当前软件系统中需要增加新功能时，对现有系统功能影响较少，即不需要修改原有功能或改动很小。（2）软件外部方面。软件系统本身与其他外部系统之间是松耦合关系，软件的变化对其他系统无影响。

设计具备良好扩展性的系统，有两个基本条件：正确预测变化（在业务方面要有深入的理解），完美封装变化（技术方面要有一定的技术能力）。

### 预测变化

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

准确的预测了变化，就能很容易的实现可扩展性了吗？也没有那么理想！因为预测变化时一回事，采取什么方案来应对变化又是一回事。也就是说，即使准确的预测了变化，但是应对变化的方案不合适，系统扩展一样很麻烦。

第一种应对变化的常见方案是将“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”。这就给系统带来了复杂性：

### (1) 系统需要拆分出变化层和稳定层

对于哪些属于变化层，哪些属于稳定层，很多时候并不明显，不同的人有不同的见解，在讨论的时候往往难以达成一致性。

### (2) 需要设计变化层与稳定层之间的接口

接口设计至关重要，对于稳定层来说接口肯定是越稳定越好；但对于变化层来说，在有差异的多个实现中找出共同点，并且还要保证当加入新的功能时原有的接口设计不需要进行太大修改，这是一件很复杂的事情。

第二种应对变化的方案是提炼出一个“抽象层”和一个“实现层”。抽象层是稳定的，实现层可以根据业务需要进行开发，当加入新的功能时，只需要增加新的实现，无须修改抽象层。这种方案的典型实践是设计模式和规则引擎。

## 7. 复杂度来源：低成本、安全、规模

### 低成本

低成本很多时候不是架构设计的首要目标，而是架构设计的附加约束。也就是说我们首先会设置一个成本目标，当我们根据高性能、高可用的要求设计出方案时，评估一下方案是否满足成本目标，如果不满足，就需要重新设计架构；如果无论如何也无法满足成本目标，那就只能找老板调整成本目标。

低成本给架构设计带来的复杂性在于，往往只有“创新”才能达到低成本目标。这里的“创新”，包括引入新技术和创造新技术。

### 安全

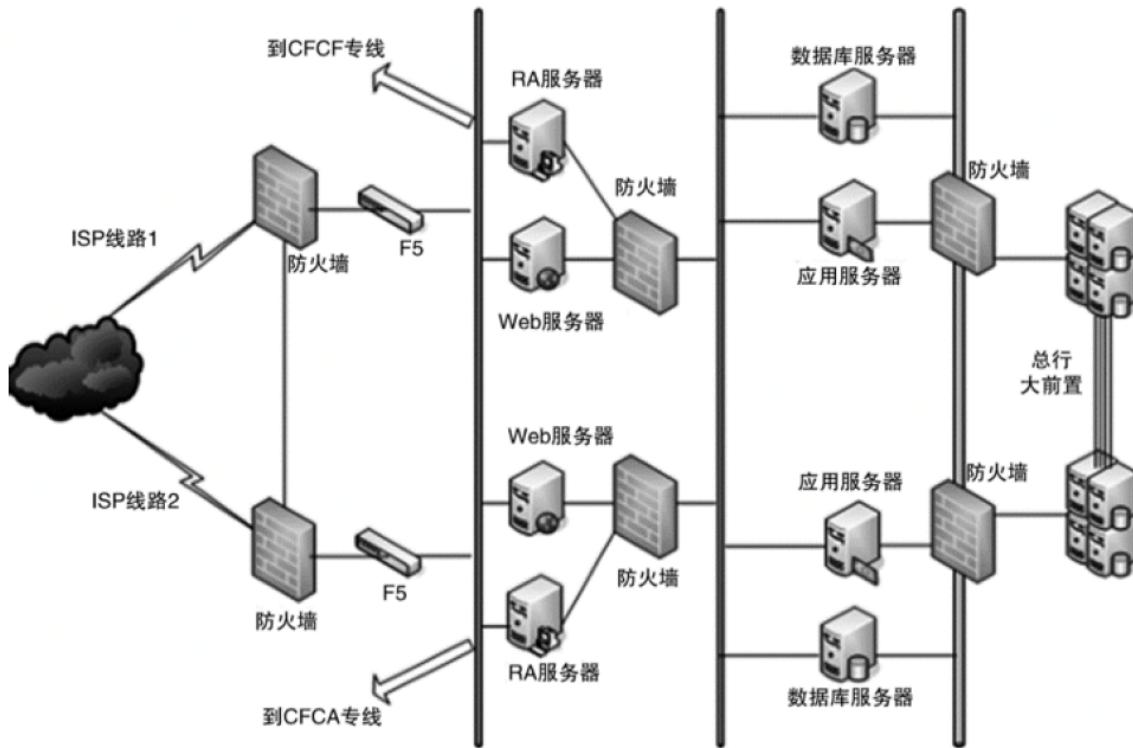
从技术角度上讲，安全可以分为两类：一类是功能上的安全，一类是架构上的安全。

### (1) 功能安全

功能安全就是防“小偷”，功能安全更多地是和具体的编码相关，与架构关系不大。

## (2) 架构安全

架构安全就是防“强盗”。传统的架构安全主要是依靠防火墙。



防火墙虽然功能强大，但性能一般，所以在传统的银行和企业应用较多。但在互联网企业应用并不多见，因为互联网行业具有海量用户访问和高并发的特性，防火墙的性能不足以支撑。就算是公司对钱不在乎，一般也不会堆防火墙来防 DDoS 攻击，因为 DDoS 攻击最大的影响是大量消耗机房的出口总带宽。不管防火墙处理能力有多强，当出口带宽被耗尽时，整个业务在用户看来就是不可用的，因为用户的正常请求已经无法到达系统了。防火墙能够保证内部系统不受冲击，但用户也是进不来的。对于用户来说，业务都已经受到影响了，至于是因为用户自己进不去，还是因为系统出故障，用户其实根本不会关心。所以互联网的架构安全并没有太好的设计手段来实现，更多的是依靠运营商和云服务商强大的带宽和流量清洗能力，较少自己设计和实现。

## 规模

规模带来复杂度的主要原因是“量变引起质变”，当数量超过一定的阀值后，复杂度会发生质的变化。常见的规模带来的复杂度有：

- (1) 功能越来越多，导致系统复杂度指数级上升。

计算公式：功能复杂度 = 功能数 + 功能之间的连接数

(2) 数据越来越多，系统复杂度发生质变。

## 8. 架构设计三原则

程序员和架构师之间的鸿沟——“不确定性”。在面临众多不变性的问题的时候，有几个共性原则可以帮助我们做出更好的选择：合适原则、简单原则、演化原则。

### 合适原则

合适原则宣言：“合适优于业界领先”。

(1) 将军难打无兵之仗

(2) 罗马不是一天建成的

(3) 冰山下面才是关键

没有那么卓越的业务场景，却想灵光一闪成为天才，当然是不靠谱的。

### 简单原则

简单原则宣言：“简单优于复杂”。

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

软件领域的复杂性体现在两个方面：

(1) 结构的复杂性

结构复杂的系统几乎毫无例外具备两个特点：

- 组成复杂系统的组件众多
- 这些组件之间的关系复杂

结构上复杂的第一个问题是，系统故障的概率就越大；假设组件故障的概率为10%，那么有三个组件的系统的可用性是

$(1-10\%) \times (1-10\%) \times (1-10\%) = 72.9\%$ ，有五个组件的系统的可用性是  $(1-10\%) \times (1-10\%) \times (1-10\%) \times (1-10\%) \times (1-10\%) = 59\%$ 。

结构上复杂的第二个问题是，修改一个组件，会影响关联的所有组件，这些被影响的组件同样会影响更多的组件。

结构上复杂的第三个问题是，定位一个复杂系统中的问题总是比简单系统更加困难。首先是组件多，每个组件都有嫌疑，一次要逐一排查；其次组件间的关系复杂，有可能表现故障的组件并不是真正原因。

## (2) 逻辑的复杂性

逻辑复杂的一个典型特征就是单个组件承担了太多的功能（也就是断绝了我们无底线的减少组件数量来解决结构的复杂性的幻想）；另外一个典型表现就是采用了复杂的算法。

## 演化原则

演化原则宣言：“演化优于一步到位”。

如果没有把握“软件架构需要根据业务发展不断变化”这个本质，在做架构设计的时候很容易陷入一个误区：试图一步到位设计一个架构，期望不论业务如何变化，架构都稳如磐石。

## 9. 架构设计原则案例

主要介绍了“淘宝”和“手机QQ”的架构发展历程；可以阅读的数据《淘宝技术这十年》

## 10. 架构设计流程：识别复杂度

### 架构设计第一步：识别复杂度

软件的复杂度主要来源于“高性能”、“高可用”、“可扩展”等几个方面，但架构师在具体判断复杂性的时候，不能生搬硬套，认为任何时候架构都必须同时满足这三方面的要求。实际上大部分场景下，复杂度只是其中的一个，少数情况下包含其中的两个，如果真的同时出现需要同时解决三个或者三个以上的复杂度，那么说明这个系统之前设计的有问题，要么可能是架构师的判断出现了失误，即使真的认为要同时满足这三方面的要求，也必须要进行优先级排序。

如果真的面对一个各种复杂度都存在问题的系统，那应该怎么办？答案是一个一个来解决，不要幻想一次架构重构解决所有问题。将主要的复杂度的问题列出来，然后根据业务、技术、团队等综合情况进行排序，优先解决当前面临的最主要的复杂度问题。

对于按照复杂度优先级解决的方式，存在一个普遍的担忧：如果按照优先级来解决复杂度，可能会解决了优先级排在前面的复杂度问题后，解决后续复杂度问题的方案需要将已经落地的方案推导重来。但真实的情况是总是可以挑出综合来看性价比比较高的方案。即使架构师决定推到重来，这个新的方案也必须能够同时解决已经被解决的复杂度问题(一般来说能够达到这种理想状态的方案基本都是依靠新技术的引入)。

## 一个实例

针对前浪微博的消息队列系统，采用“排查法”来分析复杂度，具体分析过程是：

### (1) 这个消息队列是否需要高性能

我们假设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，我们再假设平均一条消息有 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。1000 万和 1 亿看起来很吓人，但对于架构师来说，关注的不是一天的数据，而是 1 秒的数据，即 TPS 和 QPS。我们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，这个量级的数据意味着并不要求高性能。（nginx负载均衡性能是3万左右，mc的读取性能5万左右，kafka号称百万级，zookeeper写入读取2万以上，http请求访问大概在2万左右。）

虽然根据当前业务规模计算的性能要求并不高，但业务会增长，因此系统设计需要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。注意，这里的设计目标设定为峰值的 4 倍是根据业务发展速度来预估的，不是固定为 4 倍，不同的业务可以是 2 倍，也可以是 8 倍，但一般不要设定在 10 倍以上，更不要一上来就按照 100 倍预估。

### (2) 这个消息队列是否需要高可用性

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。

### (3) 这个消息队列是否需要高可扩展性

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复杂度关键。

## 11. 架构设计流程：设计备选方案

为什么要有备选方案？

(1) 心里评估过于简单，可能没有想的全面，只是因为某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看来最好的方案。

(2) 凭借个人的经验完全可能出现判断错误的情况，同时若果把这个方案进行上会，则可能会出现过度辩护的情况。

所以要设计备选方案，一般为3-5个。备选方案之间要有明显的差别；备选方案不要只限于自己所熟悉掌握或者成熟的结果，避免“手里有了锤子，所有的问题都变成了钉子”的问题出现。

备选方案不要设计的过于详细，架构师在设计备选方案的时候要避免将备选方案当作最终方案那样进行设计（每个备选方案都写的很细）：

(1) 写的太详细，就要花费大量的时间。

(2) 如果将注意力集中在了细节上，就会容易忽略整体的技术设计，导致方案数量不够或者差异不大。

(3) 评审的时候其他人容易被细节给绕进去，评审效果差。

所以在备选方案阶段，我们更要注重的是技术选型而不是技术细节，且技术选型的差异要比较明显。所有细节应该在最终过会确定了最终方案后，在最终方案中进行细化。

## 12. 架构设计流程：评估和选择备选方案

备选方案选择的一些指导思想

(1) 最简派

挑选一个看起来最简单的方案。

(2) 最牛派

挑选技术上看起来最牛的方案。

(3) 最熟派

设计师基于过往的经验，挑选自己最熟悉的方案。

#### (4) 领导派

列出备选方案，由领导来定夺。

这些思想都没有绝对的对错，但是前面都要加定语“有时候”；也就是这些思想各自有各自的适合场景，然而这个场景却很难确定。

所以给出“360环评”：列出我们需要关注的质量属性点，然后从这些质量属性的纬度去评估每个方案，再综合挑选合适当时情况的最优方案。

常见的方案质量属性点：性能、可用性、硬件成本、项目投入、复杂度、安全性、可扩展性等。在评估这些质量属性的时候，要遵循“架构设计三原则”。

完成360度环评后，我们可以基于评估结果整理出360度环评表，这样就能一目了然的看到各个方案的优劣点。但是360度环评表虽然帮助我们分析了各个备选方案，但是要选出具体方案还需要做一些工作。有几种错误的做法。

（1）数量对比法：简单的看哪个方案的优点多就选哪个。这种方式的主要错误就是把所有质量属性的重要性都一视同仁，没有各个质量属性的优先级。其次，有时候会出现两个方案的优点数一样，造成平局的情况。

（2）加权法：每个质量属性给一个权重。但是如何打分没有一定的标准，会造成打分比较随意。从而影响结果。

正确的做法是按优先级选择，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，将质量按照优先级排序，首先挑选满足第一优先的，如果方案都满足，那么就再看第二优先的.....依次类推。这样理论上是会出多个方案质量属性优缺点都一样的情况，但实际上是不可能的。因为我们在设计备选方案的时候就要求各个备选方案之间的差别要比较明显，而差别明显的系统的优缺点不可能都是一样的。

下面是一个具体的360环评表：

质量属性	引入Kafka	MySQL存储	自研存储
性能	高	中	高
复杂度	低，基本开箱即用	中，MySQL存储和复制，方案只需要开发服务器集群就可以	高，自研存储方案复杂度很高
硬件成本	低	高，一个分区就4台机器	低，和Kafka一样
可运维性	低，无法融入现有的运维体系，且运维团队无Scala经验	高，可以融入现有运维体系，MySQL运维很成熟	高，可以融入现有运维体系，并且只需要维护服务器即可，无须维护MySQL
可靠性	高，成熟开源方案	高，MySQL存储很成熟	低，自研存储系统可靠性在最初阶段难以保证
人力投入	低，开箱即用	中，只需要开发服务器集群	高，需要开发服务器集群和存储系统

## 13. 架构设计流程：详细方案设计

详细方案设计就是将方案涉及的关键技术细节给确定下来。

例如我们使用Elastic Search来做全文搜索，那么要确定索引是按照业务划分还是一个大索引就够了；副本数是2个、3个还是4个，集群节点数是3个还是6个等。

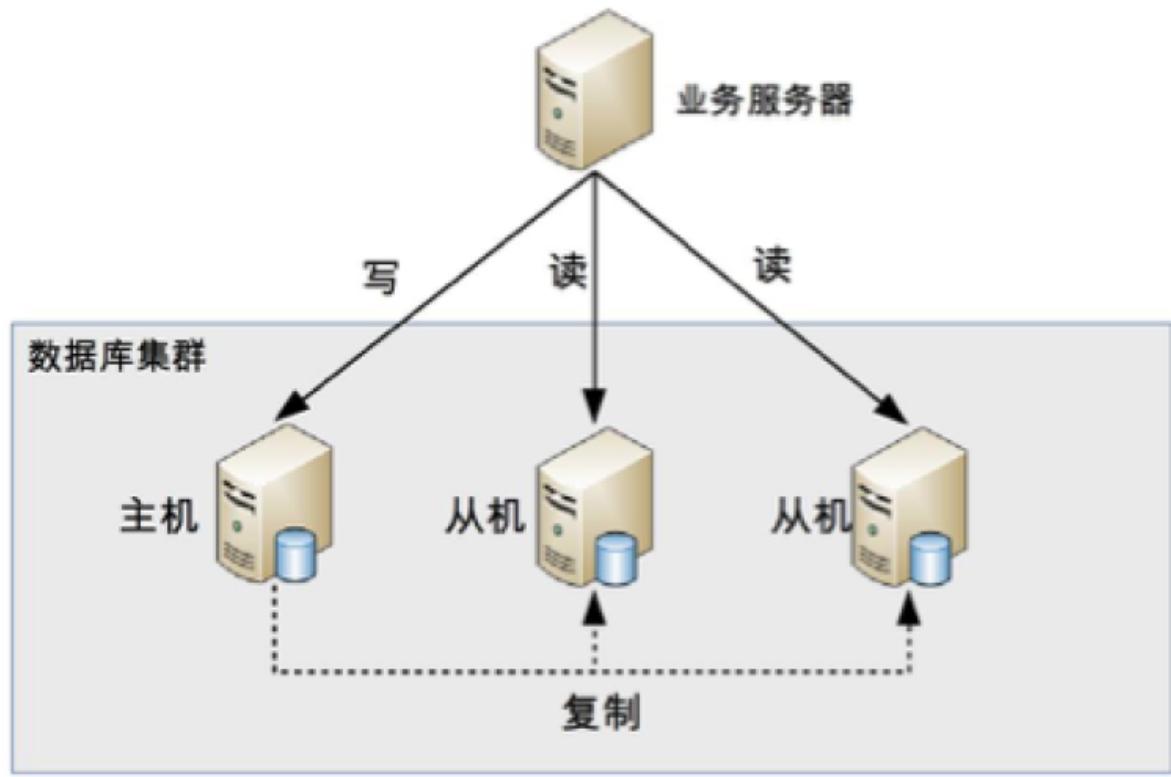
在详细方案设计阶段可能遇到的一种极端情况是发现所选的备选方案不可用，一般情况下主要是因为备选方案设计时遗漏了某个关键技术点或者关键质量属性。

可以通过一下方式来避免上面情况的出现：

- (1) 架构师不但要进行备选方案设计和选型，还需要对备选方案的关键细节有较深入的理解。
- (2) 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。
- (3) 如果方案本身很复杂，那么就采取团队设计的方式，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。

## 14. 高性能数据库集群：读写分离

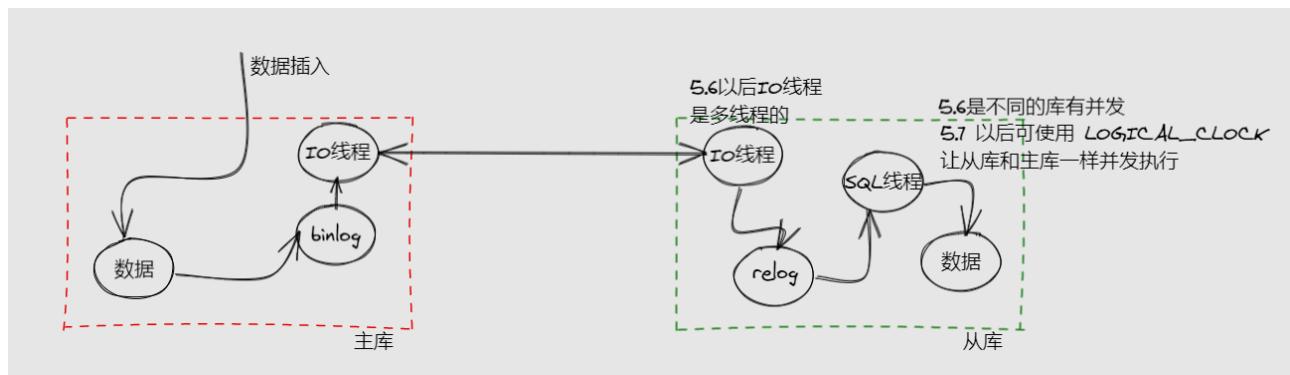
读写分离的基本原理是将数据读写操作分散到不同的节点上。



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

## 复制延迟

经验值：主库的写并发达到1000/s时，从库会落后几毫秒。2000/s时，从库会落后几十毫秒。4000/s时，就可能会落后几秒了。在一些极端的情况下，甚至会落后1分钟多。



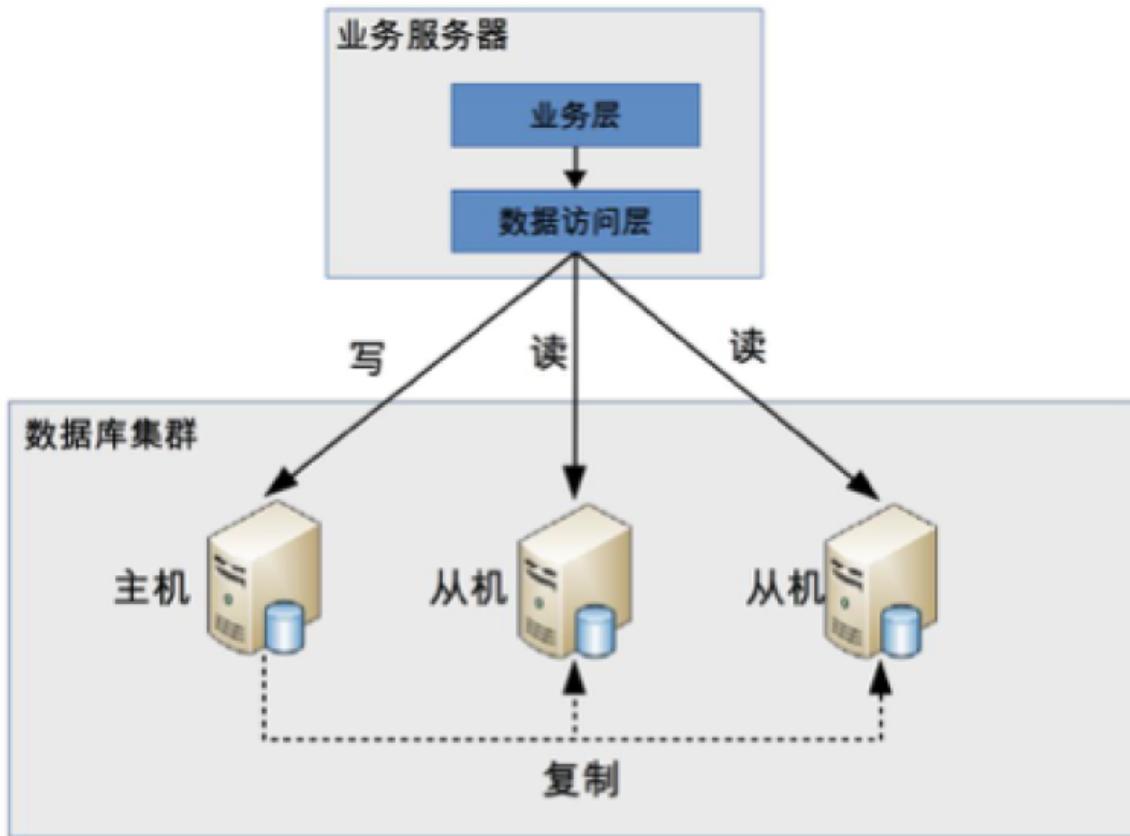
处理的办法：

- (1) 写操作后的读操作指定发给数据库主服务器
- (2) 读从机失败后再读一次主

(3) 关键业务读写全部指向主机，非关键业务用读写分离

## 分配机制

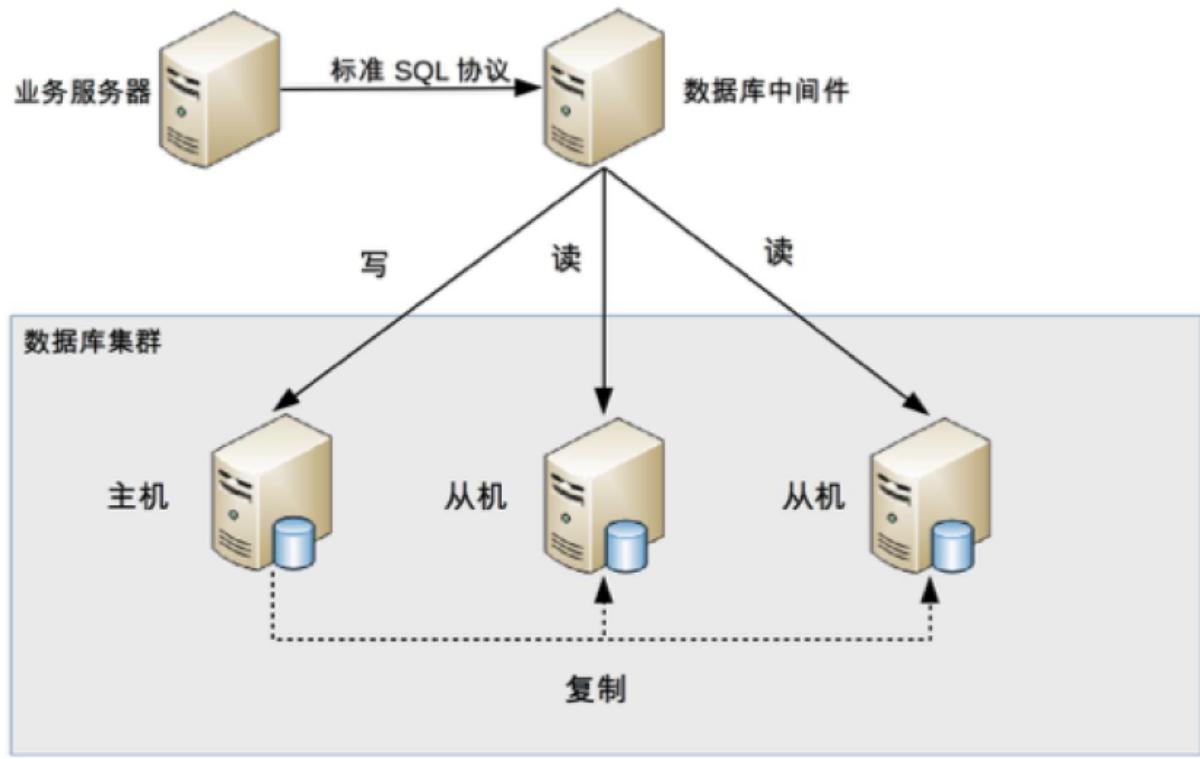
(1) 程序代码封装



特点：

- ① 实现简单，而且可以根据业务做较多的定制化功能。
- ② 每个编程语言都需要自己实现，无法通用。
- ③ 故障情况下，如果发生主从切换，则可能需要所有系统都修改配置并重启。

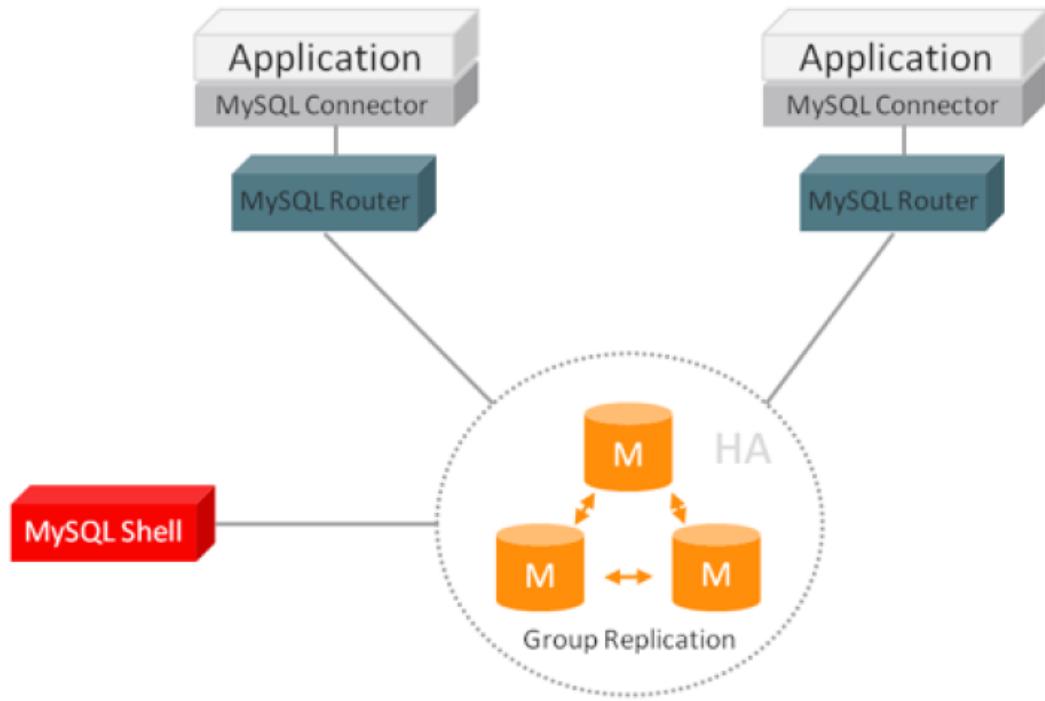
(2) 中间件封装



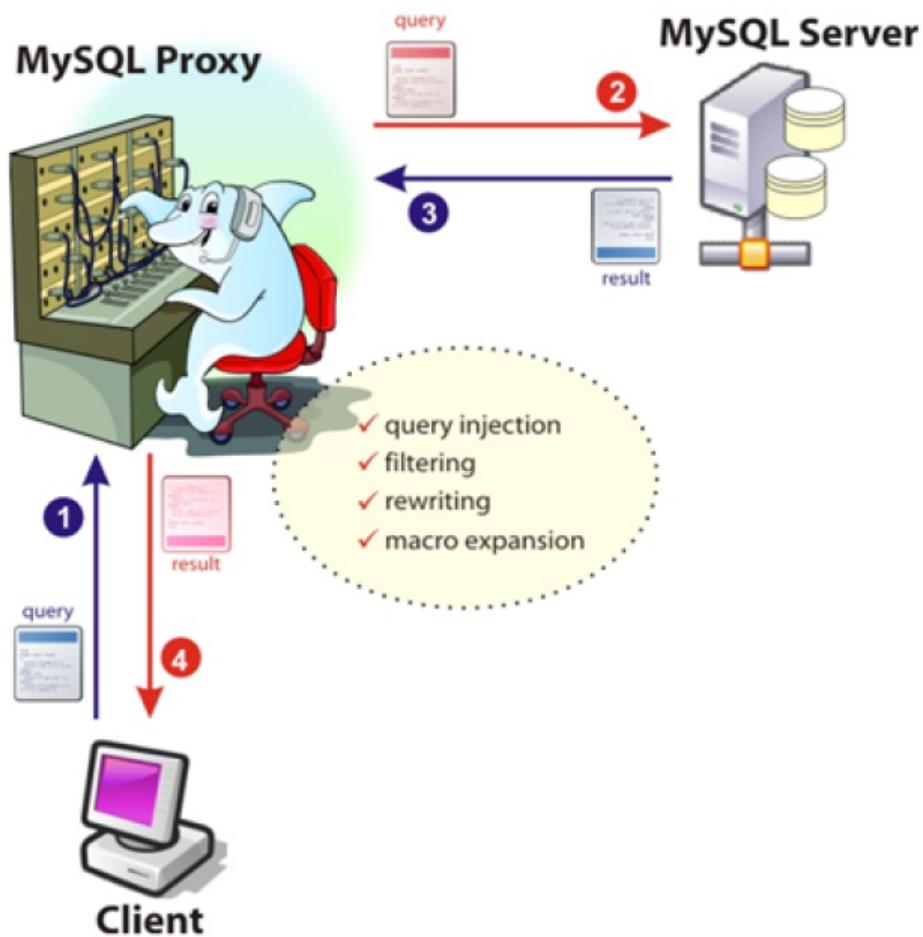
特点：

- ①能够支持多种编程语言，因为数据库中间件提供的是标准的SQL接口。
- ②数据库中间件要提供完整的SQL语法和数据库服务器的协议，时间比较复杂，细节特别多，很容易出现BUG，需要较长的时间才能稳定。
- ③数据库中间件不执行真正的读写操作，所有的数据库操作都要经过中间件，中间件本身也有高性能的要求。
- ④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主从的状态。

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：



奇虎 360 公司也开源了自己的数据库中间件 Atlas，Atlas 是基于 MySQL Proxy 实现的，基本架构如下：



开源的mycat

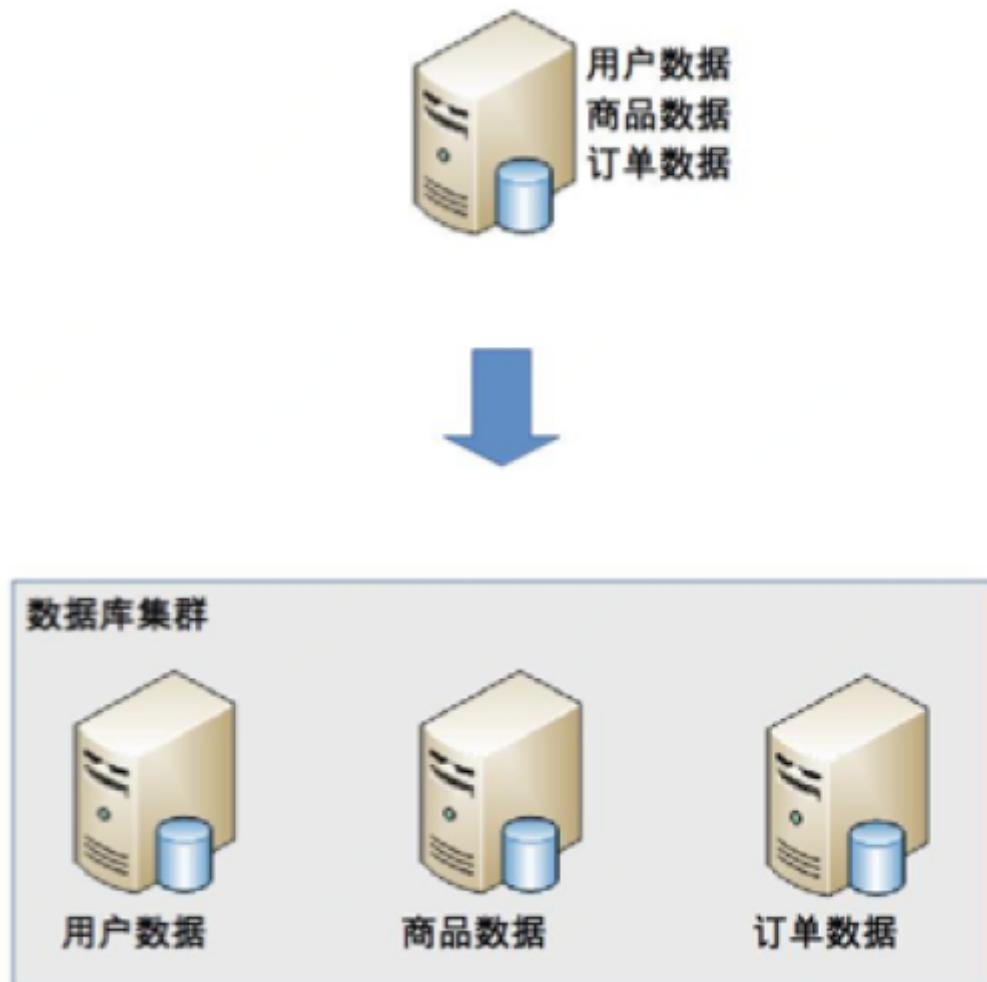
## 15. 高性能数据库集群：分库分表

分库分表主要用来处理数据库的存储压力，当数据量达到千万甚至上亿条的时候，单台数据库服务的存储能力会成为系统的瓶颈，主要体现在如下方面：

- (1) 数据量太大，读写的性能会下降，即使有索引，索引也会变的很大，性能同样会下降。
- (2) 数据文件会变的很大，数据库备份和恢复需要耗费很长的时间。
- (3) 数据文件越大，在极端情况下丢失数据的风险就越高。

### 业务分库

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。



这会带来一些问题：

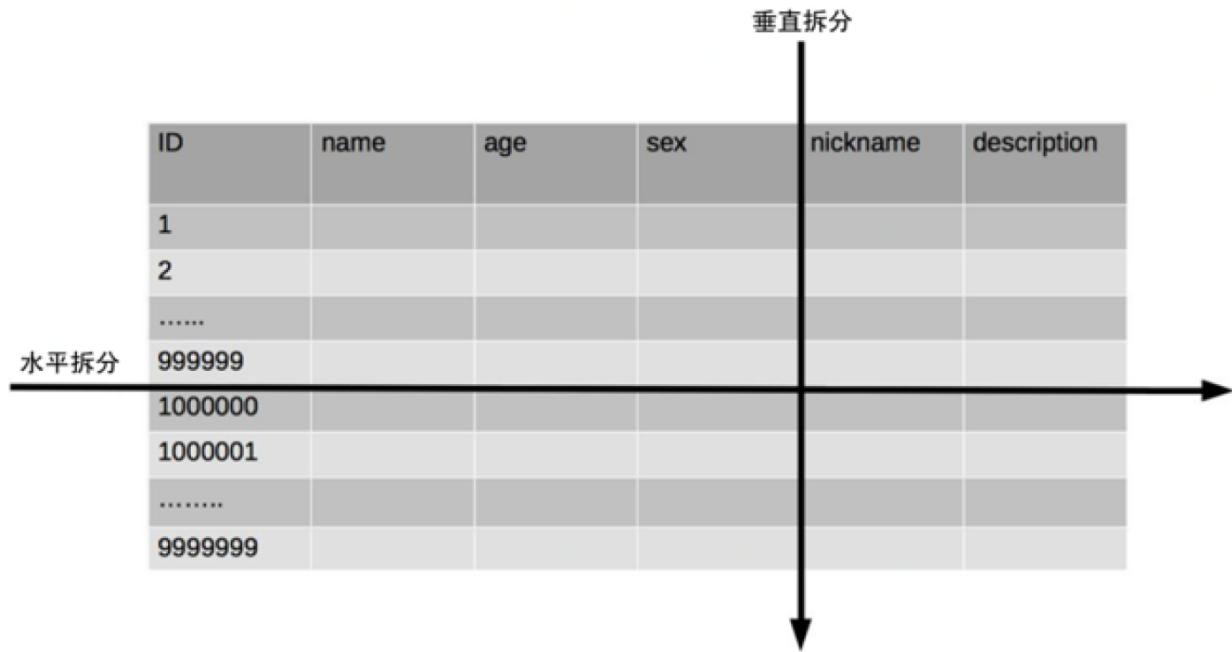
- (1) 无法进行join操作

## (2) 事务问题

## (3) 成本问题

## 分表

单表数据拆分有两种方式：垂直分表和水平分表。



### (1) 垂直分表引入的复杂性：

垂直分表的复杂性主要体现在要操作的表的数量增加，原来只需要一次就能够查询的数据，可能需要经过多次以后才能够获得。

### (2) 水平分表引入的复杂性：

**路由：**水平分表后某条数据具体属于哪个表，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。（常见的路由算法：范围路由（一般建议分段大小在100 w到1000 w）、Hash路由、配置路由）

**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操作，需要进行多次join，然后进行合并。

**Count操作：**水平分表后，虽然物理上数据分散在多个表上，但某些业务逻辑还是将这些表当作一个表来使用，水平分表前的一个count操作，在分表后，要么将每个表的数据进行count，然后把结果进行相加；这种方式实现简单，性能低。新建一张表，用来记录每张表的数据量。

**Order by**操作：水平分库后，数据分布在多个子表中，排序操作无法在数据库中完成，只能由业务代码或者数据库中间件分别查询每个子库中的数据，然后进行汇总排序。

一个可靠的执行顺序：硬件优化、数据库服务器参数调优、引入缓存技术、程序与数据库表优化重构、分库分表。

## 16. 高性能**NOSQL**

关系型数据库虽然提供了强大的SQL和ACID的能力，但是并不是完美的，存在的缺点如下所示：

(1) 关系数据库存储的是行记录，无法存储数据结构。例如，我关注的人是一个列表，这个列表在关系型数据库中必须存储为多行。

(2) 关系数据库的**schema**扩展很不方便。关系数据库的**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语句修改，而且修改期间会长时间锁表。

(3) 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，因为只针对其中某一列进行运算，关系型数据库也会将整行数据从硬盘读取到内存中。

(4) 关系数据库的全文搜索能力比较弱

针对这个问题出现了很多的NoSql解决方案，但是天下没有免费的午餐，NoSql解决方案都是在牺牲了ACID中的某些特性才换来了自己的优势。所以NoSql不是No Sql而是 Not Only Sql。

常见的NoSql方案分为四类：

- 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。（牺牲了原子性）
- 文档数据库：解决关系数据库强**schema**约束问题，以MongoDB为代表。（没有事务，不支持join操作）
- 列式数据库：解决关系数据库大I/O场景下的问题，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。

## 17. 高性能缓存架构

缓存适用的典型场景：

- (1) 需要经过复杂的运算过程后才能得到数据。
- (2) 读多写少的数据。

缓存架构的设计要点

缓存穿透

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中查询数据。通常有两种情况：

- 1、存储数据不存在
- 2、缓存数据生成耗费大量时间或者资源

典型的就是电商的商品分页，假设我们在某个电商平台上选择“手机”这个类别查看，由于数据巨大，不能把所有数据都缓存起来，只能按照分页来进行缓存，由于难以预测用户到底会访问哪些分页，因此业务上最简单的就是每次点击分页的时候按分页计算和生成缓存。通常情况下这样实现是基本满足要求的，但是如果被竞争对手用爬虫来遍历的时候，系统性能就可能出现问题。

缓存雪崩

缓存雪崩是当缓存失效（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。

常见的解决办法：

- (1) 加锁
- (2) 后台更新

(3) 双key策略：要缓存的key过期时间是t，key1没有过期时间。每次缓存读取不到key时就返回key1的内容，然后触发一个事件。这个事件会同时更新key和key1。

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。后台定时机制需要考虑一种特殊的场景，当缓存系统内存不够时，会“踢掉”一些缓存数据，从缓存被“踢掉”到下一次定时更新缓存的这段时间内，业务线程读取缓存返回空值，而业务线程本身又不会去更新缓存，因此业务上看到的现象就是数据丢了。

解决的方式有两种：

- 后台线程除了定时更新缓存，还要频繁地去读取缓存（例如，1秒或者100毫秒读取一次），如果发现缓存被“踢了”就立刻更新缓存，这种方式实现简单，但读取时间间隔不能设置太长，因为如果缓存被踢了，缓存读取间隔时间又太长，这段时间内业务访问都拿不到真正的数据而是一个空的缓存值，用户体验一般。
- 业务线程发现缓存失效后，通过消息队列发送一条消息通知后台线程更新缓存。可能会出现多个业务线程都发送了缓存更新消息，但其实对后台线程没有影响，后台线程收到消息后更新缓存前可以判断缓存是否存在，存在就不执行更新操作。这种方式实现依赖消息队列，复杂度会高一些，但缓存更新更及时，用户体验更好。

后台更新既适应单机多线程的场景，也适合分布式集群的场景，相比更新锁机制要简单一些。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

## 18. 单服务器高性能模式：PPC与TPC

架构设计决定了系统性能的上限，实现细节决定了系统性能的下限。

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 (Process Per Connection)，其含义是指每次有新的连接就专门新建一个进程去处理这个连接的请求。PPC的最大并发数量也就几百。

TPC (Thread Per Connection)，其含义是指每次有新的连接就新建一个线程去专门去处理这个链接的请求。与进程相比线程更加轻量级，创建线程的消耗也比进程要少得多。同时多线程的内存空间是共享的，线程通信比进程通信更简单。

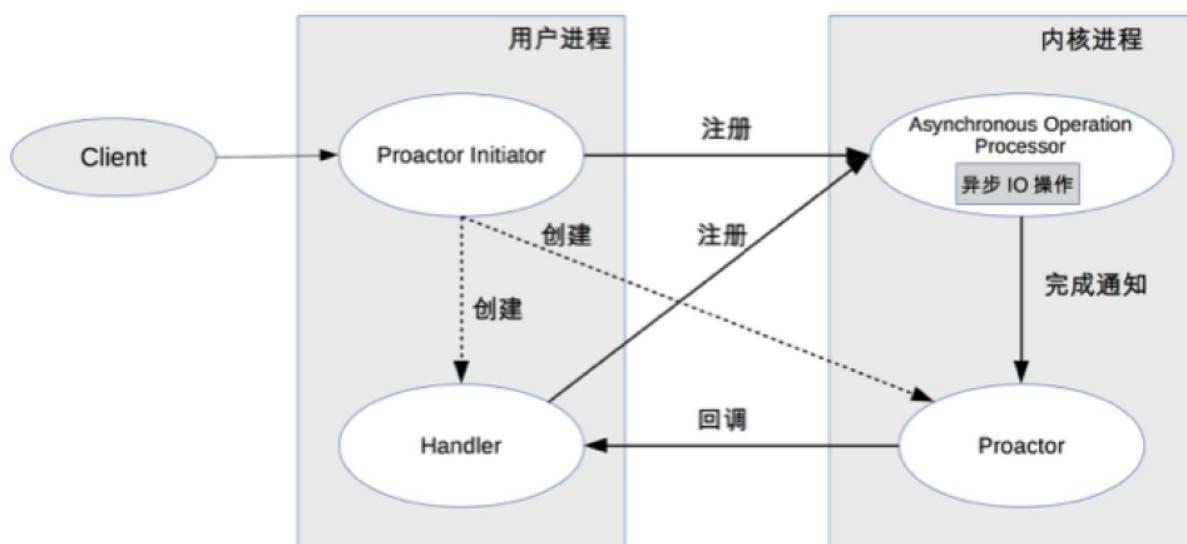
## 19. 单服务器高性能模式：Reactor与Proactor

I/O多路复用技术归纳起来有两个关键实现点：

(1) 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，而无须在轮询所有连接，常见的方法有select、epoll、kqueue等。

(2) 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 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 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进行处理。
- Handler完成业务处理，Handler也可以注册新的Handler到内核进行。

## 20. 高性能负载均衡：分类及架构

### 负载均衡分类

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。DNS负载均衡的本质是DNS解析同一个域名可以返回不同的IP地址。

DNS负载均衡实现简单、成本低，但也存在粒度太粗、负载均衡算法少等缺点。

优点：

- 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：

- 更新不及时：DNS缓存时间较长，修改DNS配置后，由于缓存的原因，还是有很多用户会访问原来的IP，这样的访问会失败，达不到负载均衡的目的，并且应用用户的正常业务。
- 扩展性差：DNS负载均衡的控制权在域名服务商那里，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）；也不能感知后端服务器的状态。

针对DNS负载均衡的一些缺点，对于延时和故障敏感的业务，也有一些公司自己实现了HTTP-DNS的功能，即使用HTTP协议实现一个私有的DNS系统。这样的方案和通用的DNS优缺点正好相反。

#### 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，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。普通业务量的公司一是负担不起，而是业务量没那么大，用这些设备也是浪费。

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：硬件负载均衡设备除具备负载均衡功能外，还具备防火墙、方DDoS攻击等安全功能。

硬件负载均衡的缺点是：

- 价格昂贵
- 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

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。

软件和硬件的主要区别就在于性能，硬件负载均衡性能远远高于软件负载均衡性能。Nginx的性能是万级，一般的Linux服务器上装一个Nginx大概能到5万/秒；LVS的性能是十万级，据说可达到80万/秒；而F5性能是百万级，从200万/秒到800万/秒都有。

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：只要买一个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插件来实现业务的定制化功能。

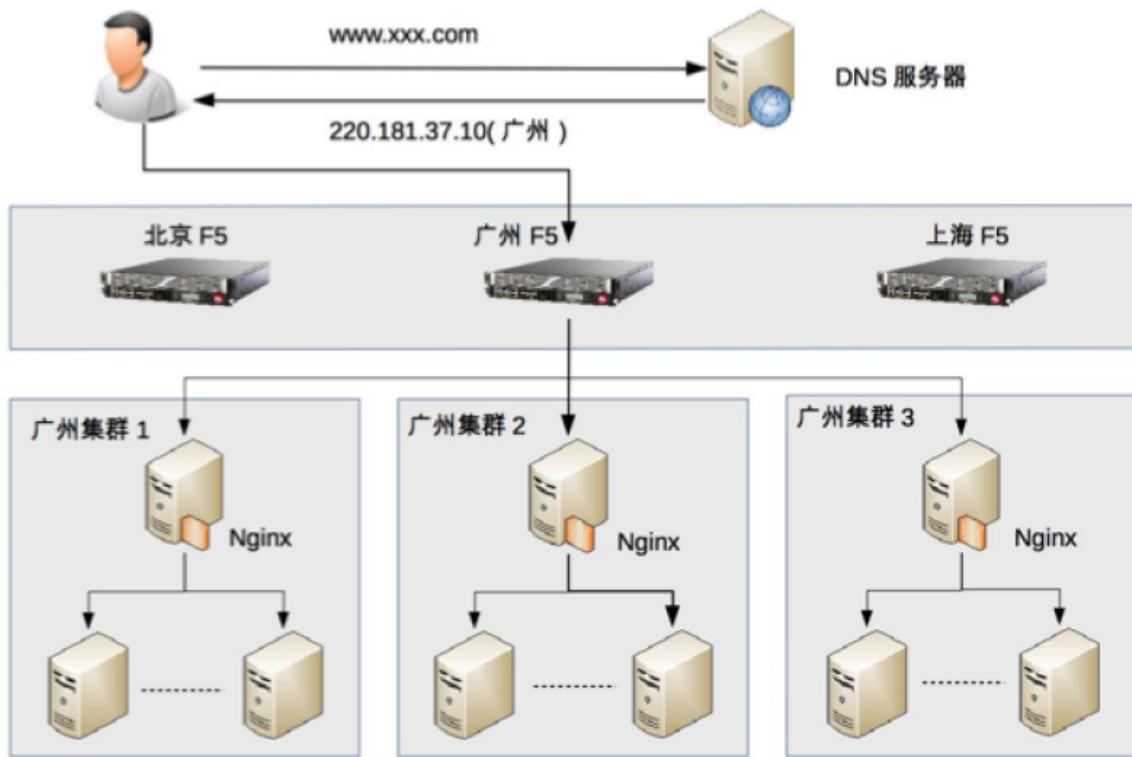
缺点（一下缺点都是和硬件负载均衡相比的，并不是说软件负载均衡没法用）

- 性能一般：一个Nginx大约能支持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。

## 负载均衡常见架构

在实际的使用中，会根据各种情况进行相应的组合，组合方式为：组合的基本原则为：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。



整个系统的负载均衡分为三层。

- 地理级别负载均衡： [www.xxx.com](http://www.xxx.com) 部署在北京、广州、上海三个机房，当用户访问时，DNS会根据用户的地理位置来决定返回哪个机房的IP，图中返回了广州机房的IP地址，这样用户就可以访问到广州机房了。
- 集群级别负载均衡：广州机房的负载均衡用的是F5设备，F5收到用户请求后，进行集群级别的负载均衡，将用户请求发给3个本地集群中的一个，我们假设F5将用户请求发给了“广州集群2”。
- 机器级别的负载均衡：广州集群2的负载均衡是Nginx，Nginx收到用户请求后，将用户请求发送给集群里面的某台服务器，服务器处理用户的业务请求返回业务响应。

## 一些有用的提议

日活千万的论坛，这个流量不低了。

1、首先，流量评估。

1000万DAU，换算成秒级，平均约等于116。

考虑每个用户操作次数，假定10，换算成平均QPS=1160。

考虑峰值是均值倍数，假定10，换算成峰值QPS=11600。

考虑静态资源、图片资源、服务拆分等，流量放大效应，假定10， $QPS \times 10 = 116000$ 。

2、其次，容量规划。

考虑高可用、异地多活， $QPS \times 2 = 232000$ 。

考虑未来半年增长， $QPS \times 1.5 = 348000$ 。

3、最后，方案设计。

三级导流。

第一级，DNS，确定机房，以目前量级，可以不考虑。

第二级，确定集群，扩展优先，则选HAProxy/LVS，稳定优先则选F5。

第三级，Nginx+KeepAlive，确定实例。

## 21. 高性能负载均衡：算法

负载均衡算法分类：

(1) 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。

(2) 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。

(3) 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。

(4) Hash类：负载均衡系统根据任务中的某些关键信息进行Hash计算，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。

各种负载均衡算法以及它们的优缺点：

(1) 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。

## (2) 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，采用动态的方式进行计算会更加契合业务，但是也会更加复杂。

加权轮询解决了轮询算法中无法根据服务器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。

## (3) 负载最低优先

负载均衡系统将任务分配给当前负载最低的服务器，这里的负载根据不同的任务类型和业务场景，可以使用不同的指标来进行衡量。

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，有此带来的代价是复杂度要增加很多。所以虽然看起来很美好，但是实现起来却很不容易，往往一个不好的实现会造成巨大的性能影响，所以使用范围没有轮询算法那么广泛。

## (4) 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相应客户端的目的。

## (4) 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，这样做的目的是为了满足特定的业务需求。

# 22. 想成为架构师，你必须知道**CAP**理论

## 定义

In a distributed system(a collection of interconnected nodes that share data),you can only have two out of the following three guarantees across a write/read pair: Consistency, Availability, and Partition Tolerance - one of them must be sacrificed.(<https://robertgreen.com/cap-theorem-revisited/>)

解释：

(1) CAP定理是针对特定的分布式系统定义的：强调了两点：interconnected 和 share data，为何要强调这两点呢？因为分布式系统不一定会互联和共享数据。最简单的例如 Memcache的集群，节点之间就没有相互连接和共享数据。而MySQL集群就是互联和进行数据复制的，因此是CAP理论探讨的对象。

(2) write/read pair , 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。例如，Zookeeper的选举机制就不是CAP所探讨的内容。

对C A P的定义的重新解释：

(1) 一致性 (Consistency)

A read is guaranteed to return the most recent write for a given client.

(2) 可用性 (Availability)

A non-failing node will return a reasonable response within a reasonable amount of time(no error or timeout).

(3) 分区容错性 (Partition Tolerance)

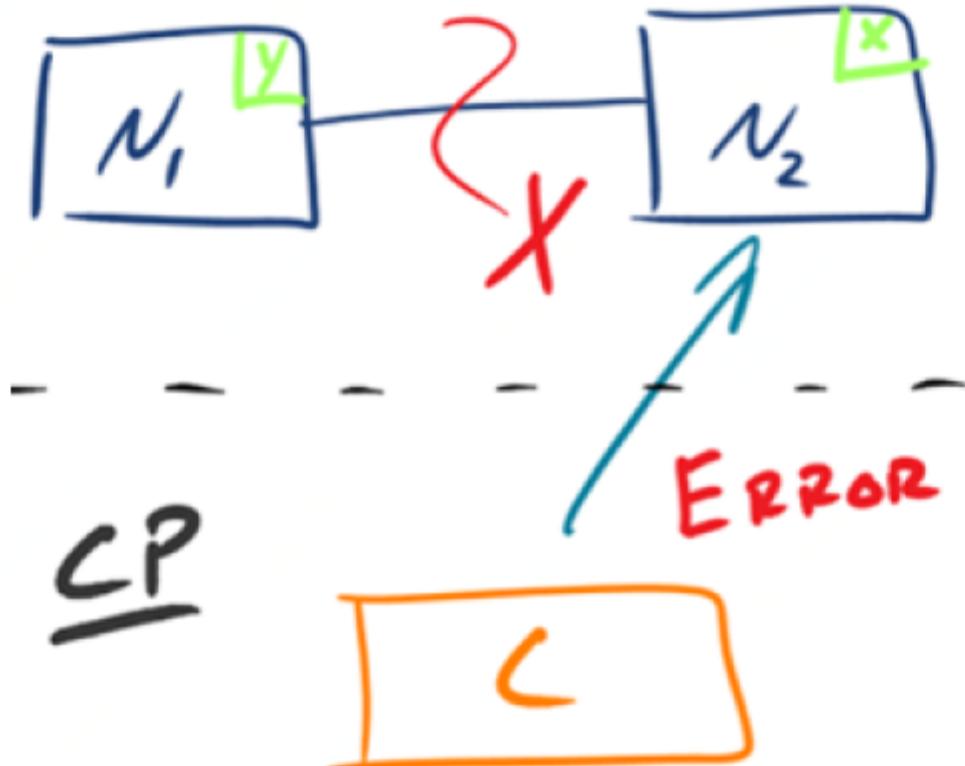
The system will continue to function when network partitions occur.

CAP应用

虽然CAP理论定义三个要素中只能取两个，但放到分布式环境下来思考，我们会发现必须选择P（分区容错）要素，因为网络本身无法做到100%可靠，有可能出现故障，所以分区是一个必然的现象。如果不选择P，那么系统只能变为了单机系统，这时也就没有什么CA的必要了。

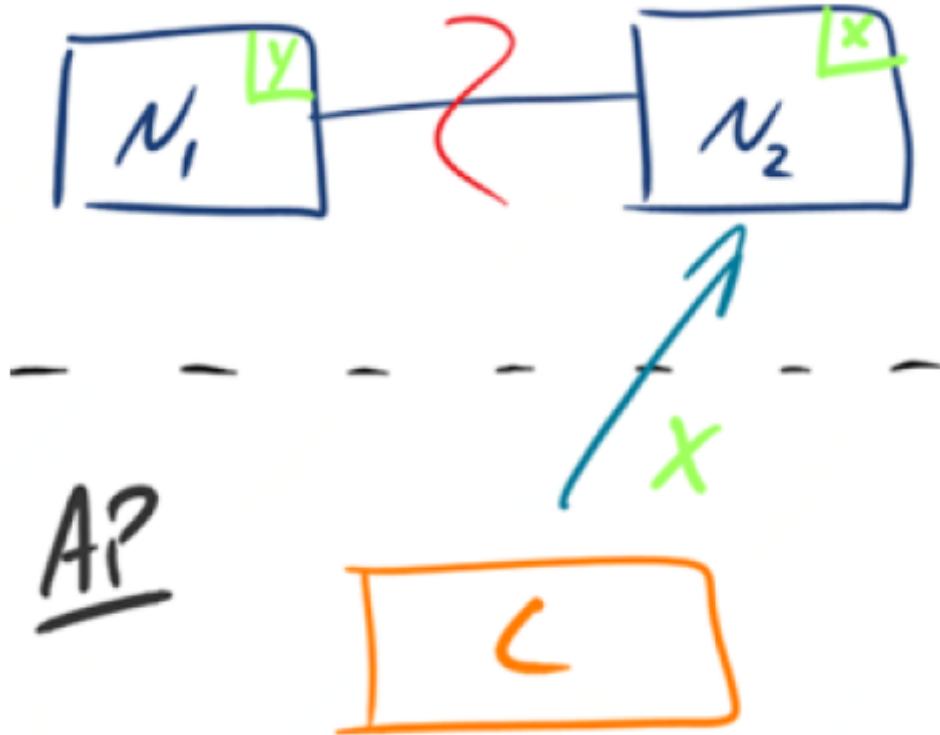
(1) CP - Consistency/Partition Tolerance

为了保证一致性，当发生分区现象后，N1 节点上的数据已经更新到 y，但由于N1 和 N2 之间的复制通道中断，数据 y 无法同步到 N2，N2 节点上的数据还是 x。这时客户端C 访问 N2 时，N2 需要返回 Error，提示客户端 C“系统现在发生了错误”，这种处理方式违背了可用性 (Availability) 的要求，因此 CAP 三者只能满足 CP。



## (2) AP - Availability/Partition Tolerance

为了保证可用性，当发生分区现象后，N1 节点上的数据已经更新到 y，但由于 N1 和 N2 之间的复制通道中断，数据 y 无法同步到 N2，N2 节点上的数据还是 x。这时客户端 C 访问 N2 时，N2 将当前自己拥有的数据 x 返回给客户端 C 了，而实际上当前最新的数据已经是 y 了，这就不满足一致性（Consistency）的要求了，因此 CAP 三者只能满足 AP。注意：这里 N2 节点返回 x，虽然不是一个“正确”的结果，但是一个“合理”的结果，因为 x 是旧的数据，并不是一个错乱的值，只是不是最新的数据而已。



## 23. 想成为架构师，你必须掌握的CAP细节

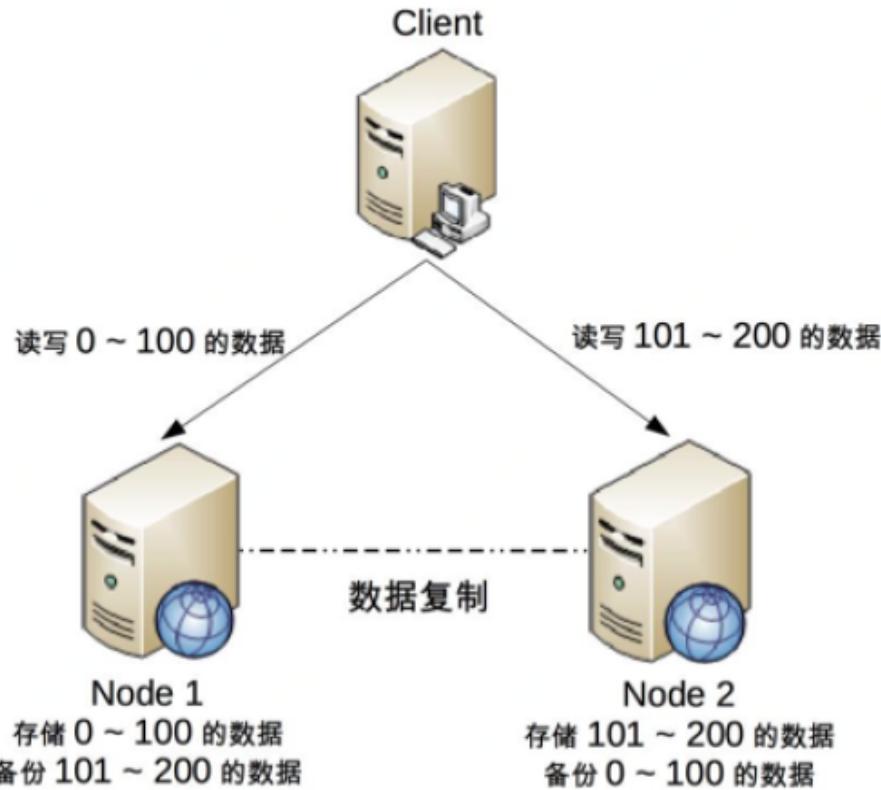
(1) CAP关注的粒度是数据，而不是整个系统。

C与A之间的取舍可以在同一系统内以非常小的粒度反复发生，而每一次的决策可能因为具体的操作，乃至因为牵涉到特定的数据或者用户而有所不同。所以在CAP理论落地实践时，我们需要将系统内的数据按照不同的应用场景和要求进行分类，每类数据选择不同的策略（CP还是AP），而不是直接限定整个系统所有数据都是同一策略。

(2) CAP是忽略网络延时的

这是一个非常隐含的假设，布鲁尔在定义一致性时，并没有将延迟考虑进去。也就是说，当事务提交时，数据能够瞬间复制到所有节点。但实际情况下，从节点A复制数据到节点B，总是需要花费一定时间的。如果是相同机房，耗费时间可能是几毫秒；如果是跨地域的机房，例如北京机房同步到广州机房，耗费的时间就可能是几十毫秒。这就意味着，CAP理论中的C在实践中是不可能完美实现的，在数据复制的过程中，节点A和节点B的数据并不一致。不要小看了这几毫秒或者几十毫秒的不一致，对于某些严苛的业务场景，例如和金钱相关的用户余额，或者和抢购相关的商品库存，技术上是无法做到分布式场景下完美的的一致性的。而业务上必须要求一致性，因此单个用户的余额、单个商品的库存，理论上要求选择CP而实际上CP都做不到，只能选择CA。也就是说，只能单点写入，其他节点做备份，无法做到分布式情况下多点写入。

需要注意的是，这并不意味着这类系统无法应用分布式架构，只是说“单个用户余额、单个商品库存”无法做分布式，但系统整体还是可以应用分布式架构的。例如，下面的架构图是常见的将用户分区的分布式架构。



(3) 正常运行情况下，不存在CP和AP的选择，可以同时满足CA。

架构设计的时候既要考虑分区发生时选择 CP 还是 AP，也要考虑分区没有发生时如何保证 CA。

(4) 放弃并不等于什么都不做，需要为分区恢复后做准备。

最典型的就是在分区期间记录一些日志，当分区故障解决后，系统根据日志进行数据恢复，使得重新达到 CA 状态。同样以用户管理系统为例，对于用户账号数据，假设我们选择了 CP，则分区发生后，节点 1 可以继续注册新用户，节点 2 无法注册新用户（这里就是不符合 A 的原因，因为节点 2 收到注册请求后会返回 error），此时节点 1 可以将新注册但未同步到节点 2 的用户记录到日志中。当分区恢复后，节点 1 读取日志中的记录，同步给节点 2，当同步完成后，节点 1 和节点 2 就达到了同时满足 CA 的状态。

而对于用户信息数据，假设我们选择了 AP，则分区发生后，节点 1 和节点 2 都可以修改用户信息，但两边可能修改不一样。例如，用户在节点 1 中将爱好改为“旅游、美食、跑步”，然后用户在节点 2 中将爱好改为“美食、游戏”，节点 1 和节点 2 都记录了未同步的爱好数据，当分区恢复后，系统按照某个规则来合并数据。例如，按照“最后修改优先规则”将用户爱好修改为“美食、游戏”，按照“字数最多优先规则”则将用户爱好修改为“旅游，美食、跑步”，也可以完全将数据冲突报告出来，由人工来选择具体应该采用哪一条。

## 24. FMEA方法，排除架构可用性隐患的利器

FMEA（Failure Mode and Effects Analysis，故障模式与影响分析）

### FMEA方法

在架构设计领域，FMEA的具体分析方法为：

- (1) 给出初始的架构设计图
- (2) 假设架构中某个部件发生故障
- (3) 分析此故障对系统功能造成的影响
- (4) 根据分析结果判断架构是否需要进行优化

FMEA分析的方法其实很简单，就是一个FMEA分析表，常见的FMEA分析表包含下面内容。

- (1) 功能点

当前的FMEA分析涉及的功能点，注意这里的“功能点”是从用户的角度来看的，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存储功能、Redis缓存功能是不能作为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。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、功能点相应缓慢、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。

#### (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。那为何还要列出故障原因呢？主要原因有几个：

- 不同的故障原因发生的概率不同
- 不同的故障原因检测手段不一样
- 不同的故障原因的处理措施不一样

#### (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。一般分为“高/中/低”三挡即可，具体评估的时候需要有以下几点需要重点关注。

- 硬件

硬件随着使用时间推移，故障概率会越来越高。例如，新的硬盘坏道几率很低，但使用了3年的硬盘，坏道几率就会高很多。

- 开源系统

成熟的开源系统 bug 率低，刚发布的开源系统 bug 率相比会高一些；自己已经有使用经验的开源系统 bug 率会低，刚开始尝试使用的开源系统 bug 率会高。

- 自研系统

和开源系统类似，成熟的自研系统故障概率会低，而新开发的系统故障概率会高。

#### (7) 风险程度

风险程度 = 严重程度 × 故障概率

#### (8) 已有措施

针对具体的故障原因，系统是否提供了某些措施来进行应对，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：为了降低磁盘坏道概率，强制统一更换服务时间超过两年的磁盘。

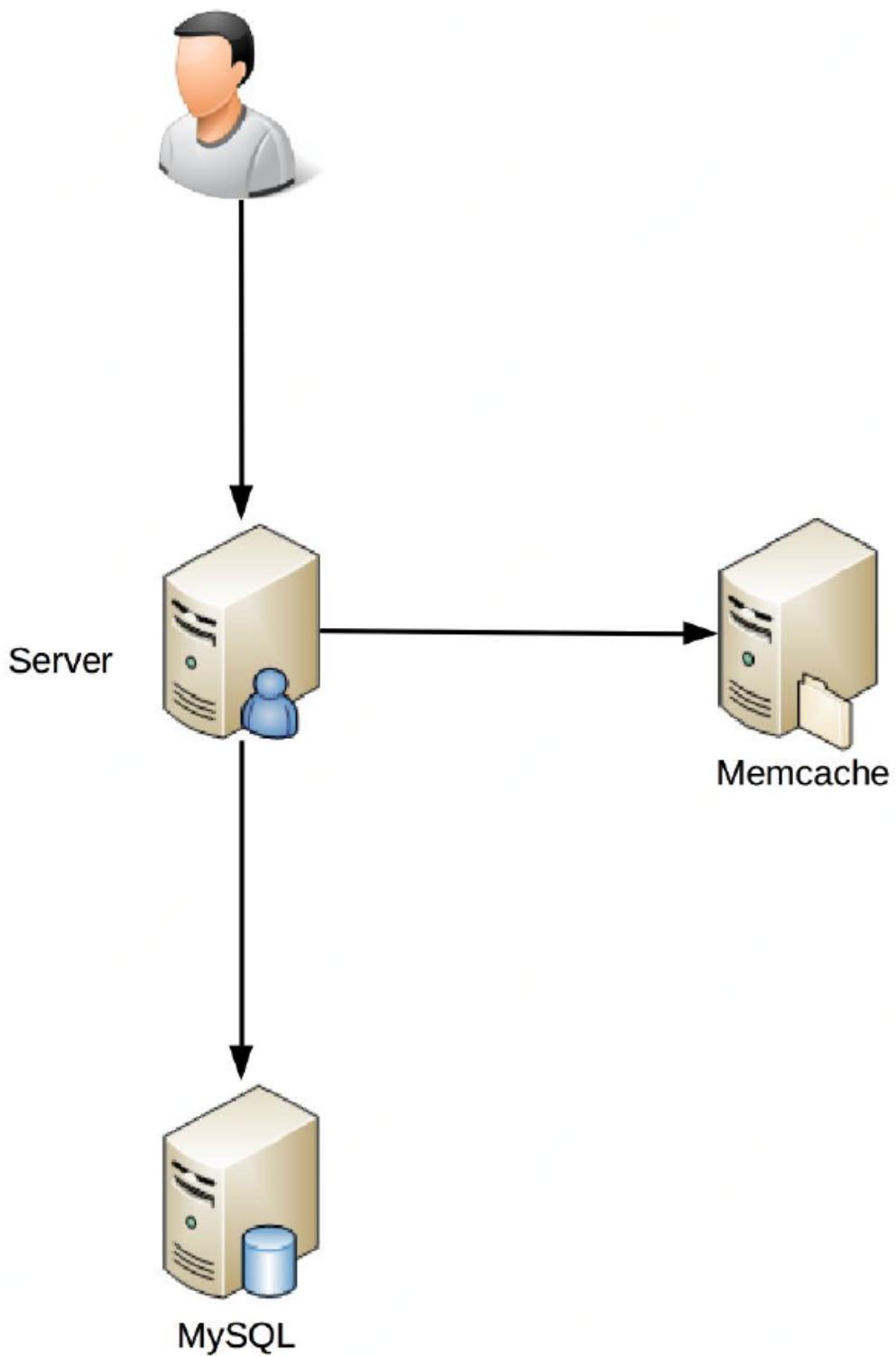
#### （10）解决措施

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。

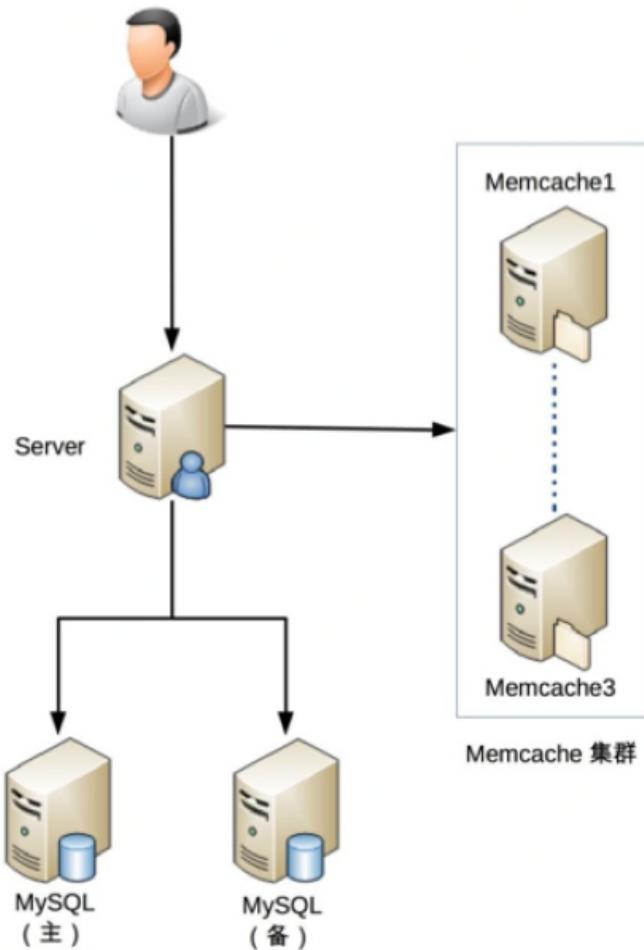
## FMEA实战



功能点	故障模式	故障影响	严重程度	故障原因	故障概率	风险程度	已有措施	规避措施	解决措施	后续规划
登录	MySQL无法访问	当MC中无缓存时，用户无法登录，预计有60%的用户	高	MySQL服务器断电	中	中	无	无	无	增加备份MySQL
登录	MySQL无法访问	同上	高	Server到MySQL的网络连接中断	中	中	无	无	无	MySQL双网卡连接
登录	MySQL响应时间超过5秒	60%的用户登录时间超过5秒	高	慢查询导致MySQL运行缓慢	高	高	慢查询检测	重启MySQL	无	不需要
登录	MC无法访问	所有用户都到MySQL查询信息，MySQL压力会增大，响应会变慢	低，虽然慢一些，但用户还是能够登录	MC服务器断电	中	低	无	无	无	MC集群
注册	MySQL无法	低，因为新用户无法注	注册的用户	MySQL服务	中	低	无	无	无	无，因为即使增加备份机器，也

	访问	册	每天大约只 有100个	器断电						无法作为主机写入
注册	MC无法访 问	无影响，用 户注册流程 不操作MC	无	MC服务器 断电	中	低	无	无	无	不需要

改进后的架构：



## 25. 高可用存储：双机架构

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。因此，对于任何一个高可用存储方案，我们需要从以下几个方面进行思考和分析：

- (1) 数据如何复制？
- (2) 各个节点的职责是什么？
- (3) 如何应对复制延迟？
- (4) 如何应对复制中断？

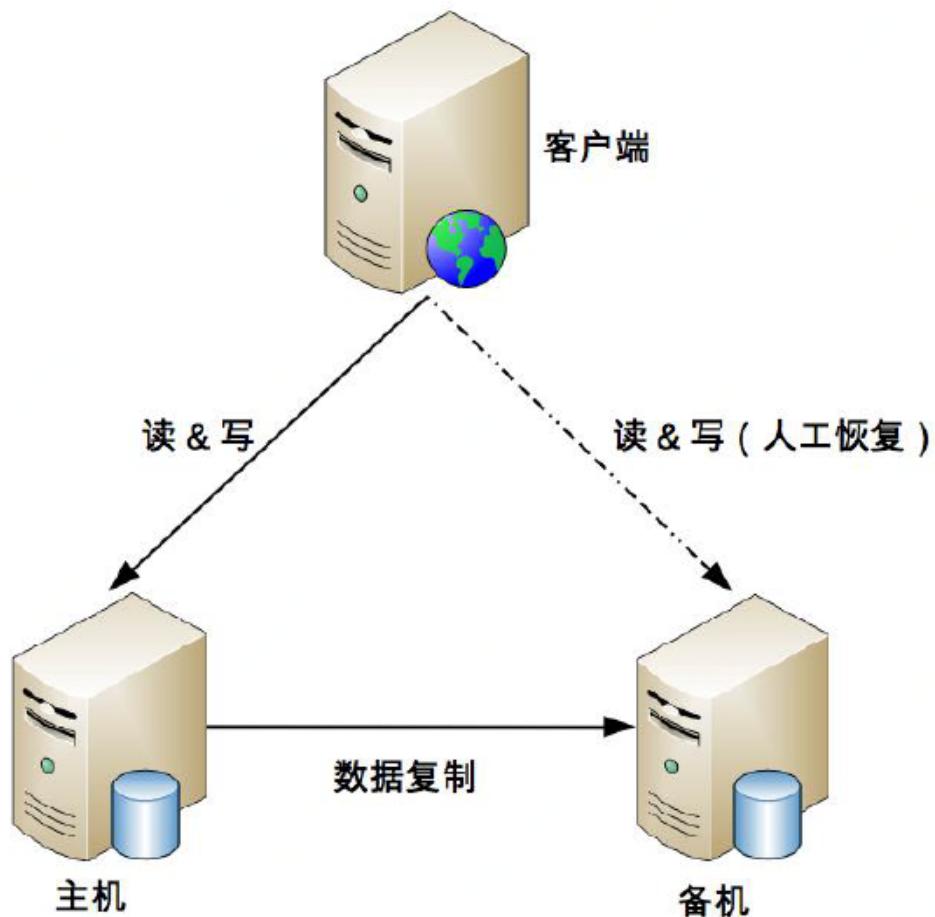
常见的高可用存储架构有主备、主从、主主、集群、分区，每一种又可以根据业务需求进行一些特殊的定制化功能，由此衍生出更多的变种。

## 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。

### 1、基本实现

下面是标准的主备方案结构图：



其整体架构比较简单，“备机”主要起到一个数据备份的作用，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。

### 2、优缺点分析

主备复制架构的优点就是简单，表现有：

(1) 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了主机。

(2) 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。

主备复制框架的缺点主要有：

(1) 备机仅仅只为备份，并没有提供读写操作，这是一种硬件资源上的浪费。

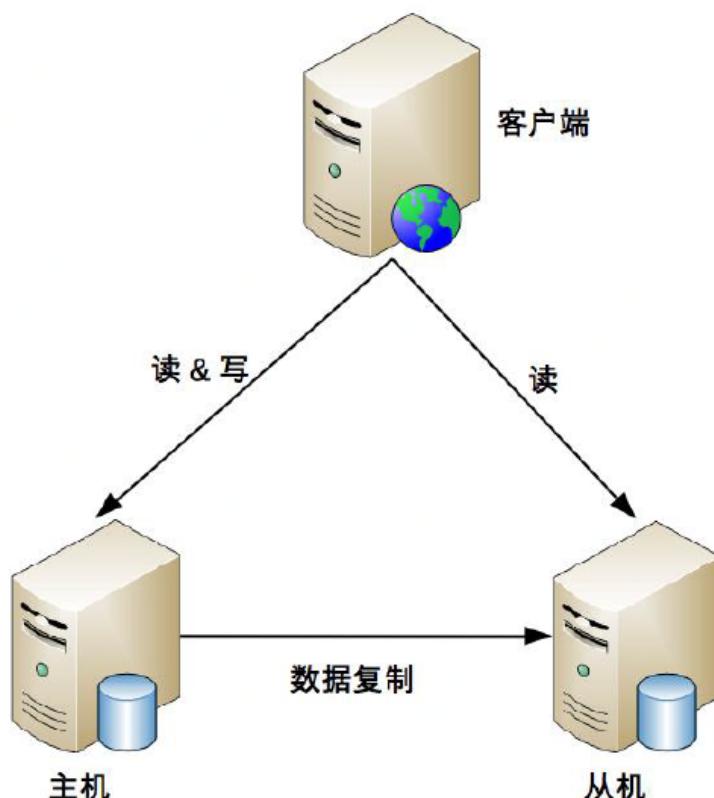
(2) 故障后需要人工干预，无法自动恢复。

综合主备复制的优缺点，内部的后台管理系统使用主备复制架构的情况会比较多。

## 主从复制

“从”是仆从的意思，那么就要帮主人干活。这里干的活就是承担“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。

### 1、基本实现



与主备复制框架比较类似，主要的差别点在于从机正常情况下也是要提供读操作的。

## 2、优缺点分析

与主备复制相比优点有：

(1) 主从复制在主机故障时，读操作相关的业务可以继续运行。

(2) 主从复制架构的从机提供读操作，能够利用硬件资源。

缺点：

(1) 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。

(2) 主从复制架构中，从机提供读业务，如果主从复制延迟较大，业务会因为数据不一致出现问题。

(3) 故障时需人工干预。

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。

## 双机切换

(1) 设计关键

主备复制和主从复制方案存在两个共性问题：

- 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这两种方案。简单来说，这两种方案就是在原来的基础上增加了“切换”功能，即系统自动决定主机角色，并完成角色切换。由于主备切换和主从切换在切换的设计上没有差别，接下来以主备切换为例，一起来看看双机切换架构是如何实现的。

要实现一个完善的切换方案，必须考虑这个几个关键的设计点：

- 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：是相互间相互连接，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、相应是否缓慢等。

- 切换策略

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：原来的主机回复故障后，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：切换完全是自动的，还是半自动？例如，系统最终切换前是否需要进行人工确认。

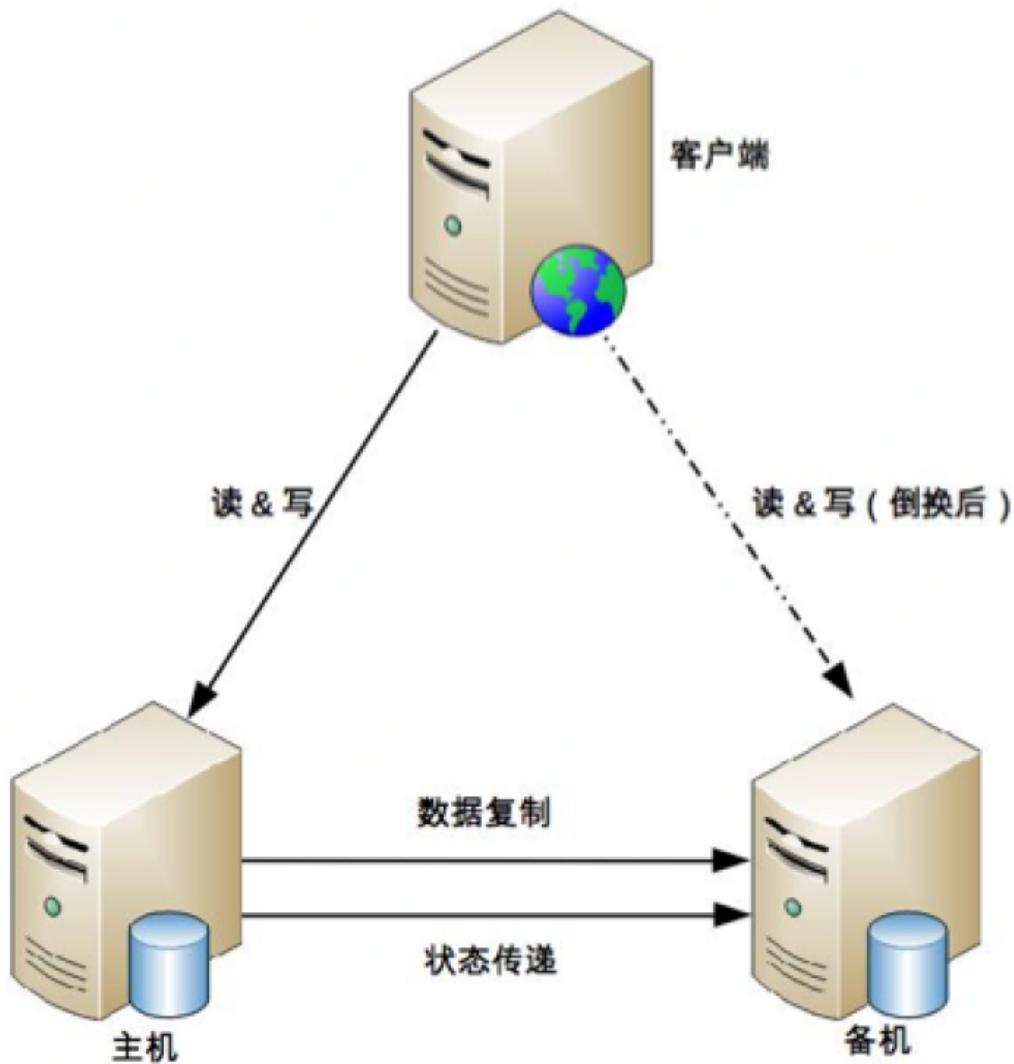
- 数据冲突解决

## (2) 常见框架

根据状态传递渠道的不同，常见的主备切换架构有三种形式：互联式、中介式和模拟式。

- 互联式

顾名思义，互联式就是指主备机直接建立状态传递的渠道，架构图请注意与主备复制架构对比。



在主备架构的基础上，主机和备机多了一个“状态传递”的通道，这个通道就是用来传递状态信息的。这个通道的具体实现可以有很多种方式。

- ① 可以是网络连接（例如，各开一个端口），也可以是非网络连接（用串口线连接）。
- ② 可以是主机发送状态信息给备机，也可以是备机到主机来获取状态信息。
- ③ 可以和数据复制公用一条通道，也可以独立使用一条通道。
- ④ 状态传递通道可以是一条，也可以是多条，还可以是不同类型的通道混合（例如，网络+串口）。

为了充分利用切换方案能够自动决定主机这个优势，客户端这里也会有相应的变化，常见的方法有：

- ① 为了切换后不影响客户端的访问，主机和备机之间共享一个客户端来说唯一的地址，例如，虚拟IP。

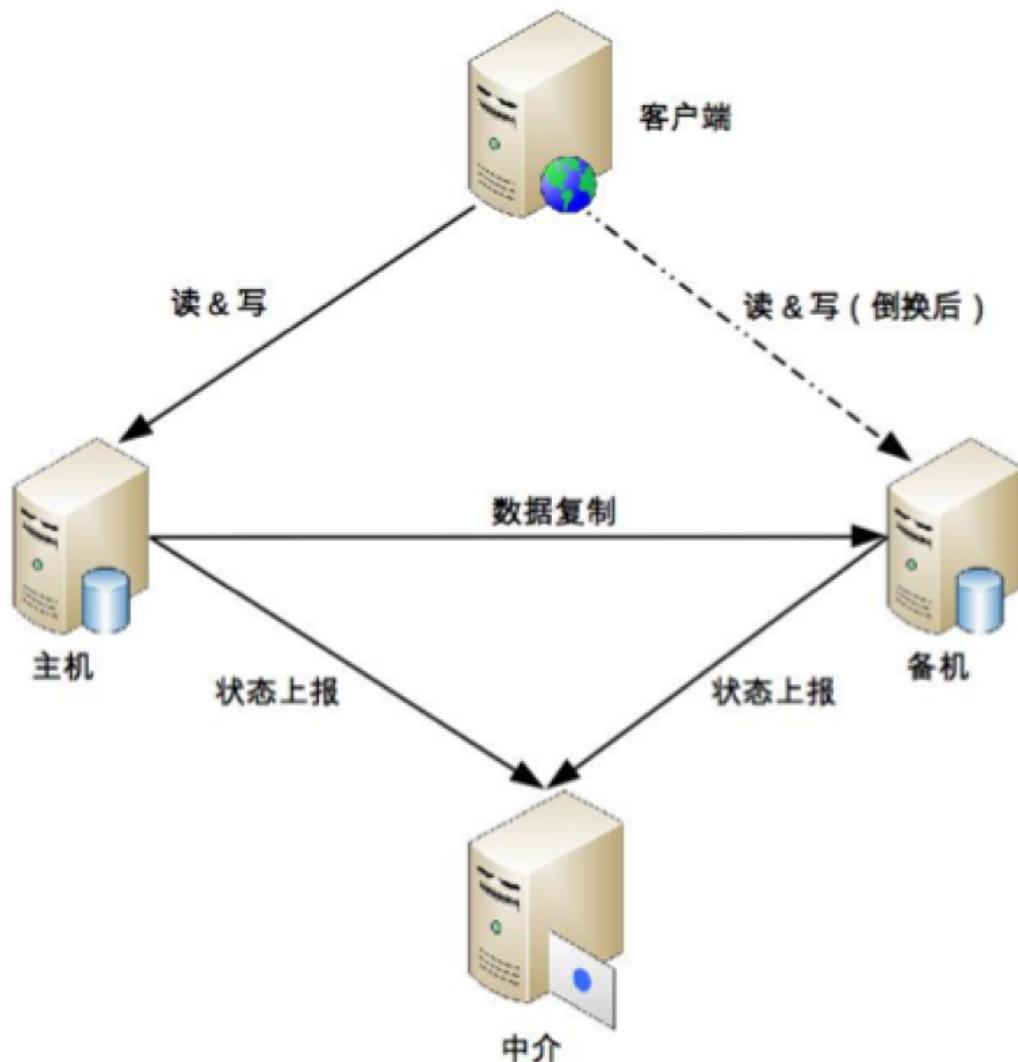
② 客户端同时记录主备机地址，哪个能访问就访问哪个；备机虽然能收到客户端的请求，但会直接拒绝，拒绝的原因就是“备机不对外提供服务”。

互联式主备切换的缺点在于：

- ① 如果状态转换的通道本身有故障（例如，网线被人不小心踢掉），那么备机也会认为主机故障而将自己升级为主机，而此时主机并没有故障，最终可能会出现两个主机。
- ② 虽然可以通过增加多个通道来增强状态转换的可靠性，但这样做只是降低了通道故障概率而已，不能从根本上解决这个缺点，而且通道越多，后续的状态决策就会越复杂，因为对于备机来说，可能从不同的通道收到了不同的信息。

- 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，而是去连接中介，并且通过中介来传递状态信息。



中介式架构在状态传递和决策上更加简单了，因为如下原因：

① 决策状态更简单：主机的状态决策简单了，无须考虑多种类型的连接通道获取的状态信息如何决策的问题，只需要按照下面简单的算法即可完成状态决策。

无论是主机还是备机，初始状态都是备机，并且只要与中介断开连接，就将自己降为备机，因此可能出现双机双备的情况。

主机与中介断连后，中介能够立刻告诉备机，备机将自己升级为主机。

如果是网络中断导致主机与中介断连，主机会将自己降为备机，与中介恢复连接后，发现已经有主机了，保持自己的备机状态不变。

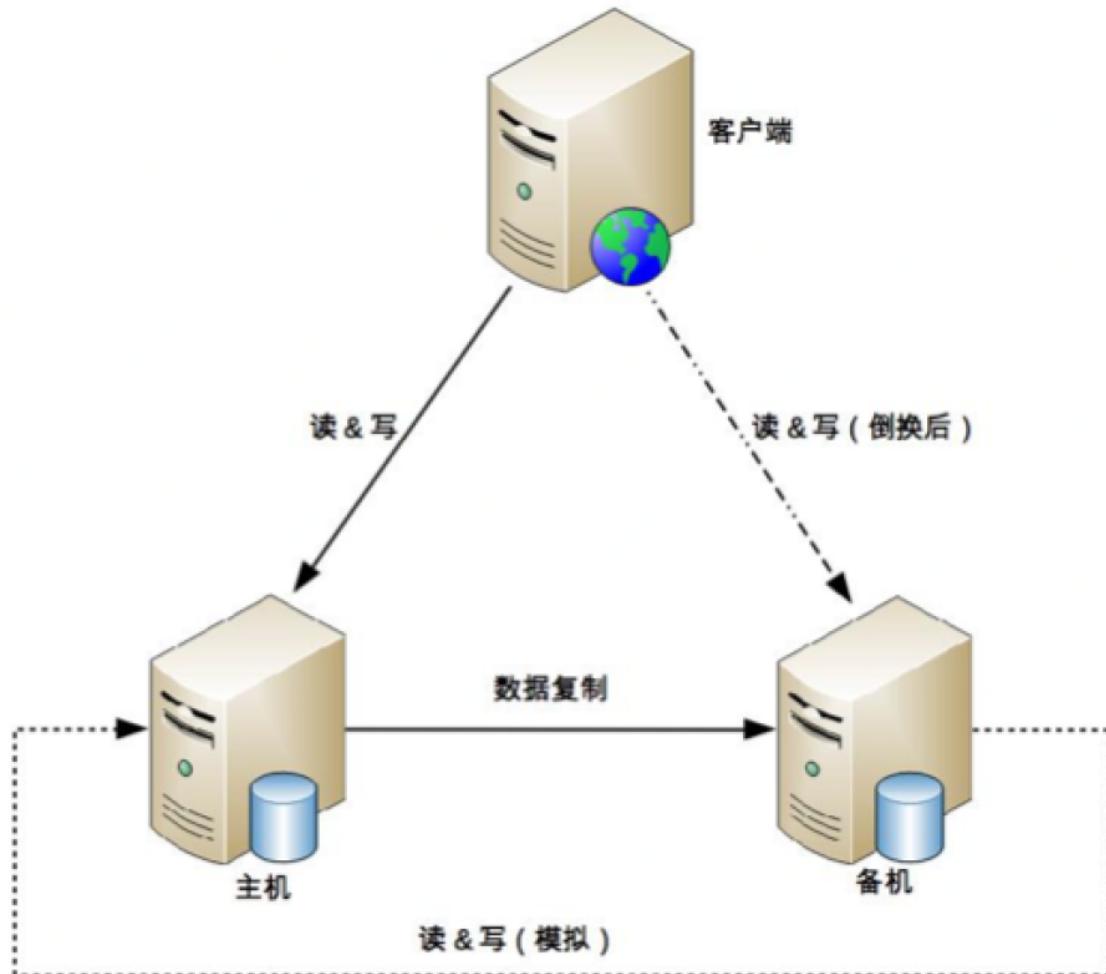
如果是掉电重启或者进程重启，旧的主机初始状态变为备机，与中介恢复连接后，发现已经有主机了，保持自己备机状态不变。

主备与中介连接都正常的情况下，按照实际的状态决定是否切换。例如，主机响应超过3秒就进行切换，主机降为备机，备机升级为主机即可。

虽然中介式架构在状态传递和状态决策上更加简单，但并不意味着这种优点是没有代价的，其关键代价就在于如何实现中介本身的高可用。如果中介自己宕机了，整个系统就进入了双备的状态，写操作相关的业务就不可用了。这就陷入了一个递归的陷阱：为了实现高可用，我们引入中介，但中介本身又要求高可用，于是又要设计中介的高可用方案……如此递归下去就无穷无尽了。

- 模拟式

模拟式指主备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，根据读写操作的相应情况来判断主机的状态。其基本架构如下：

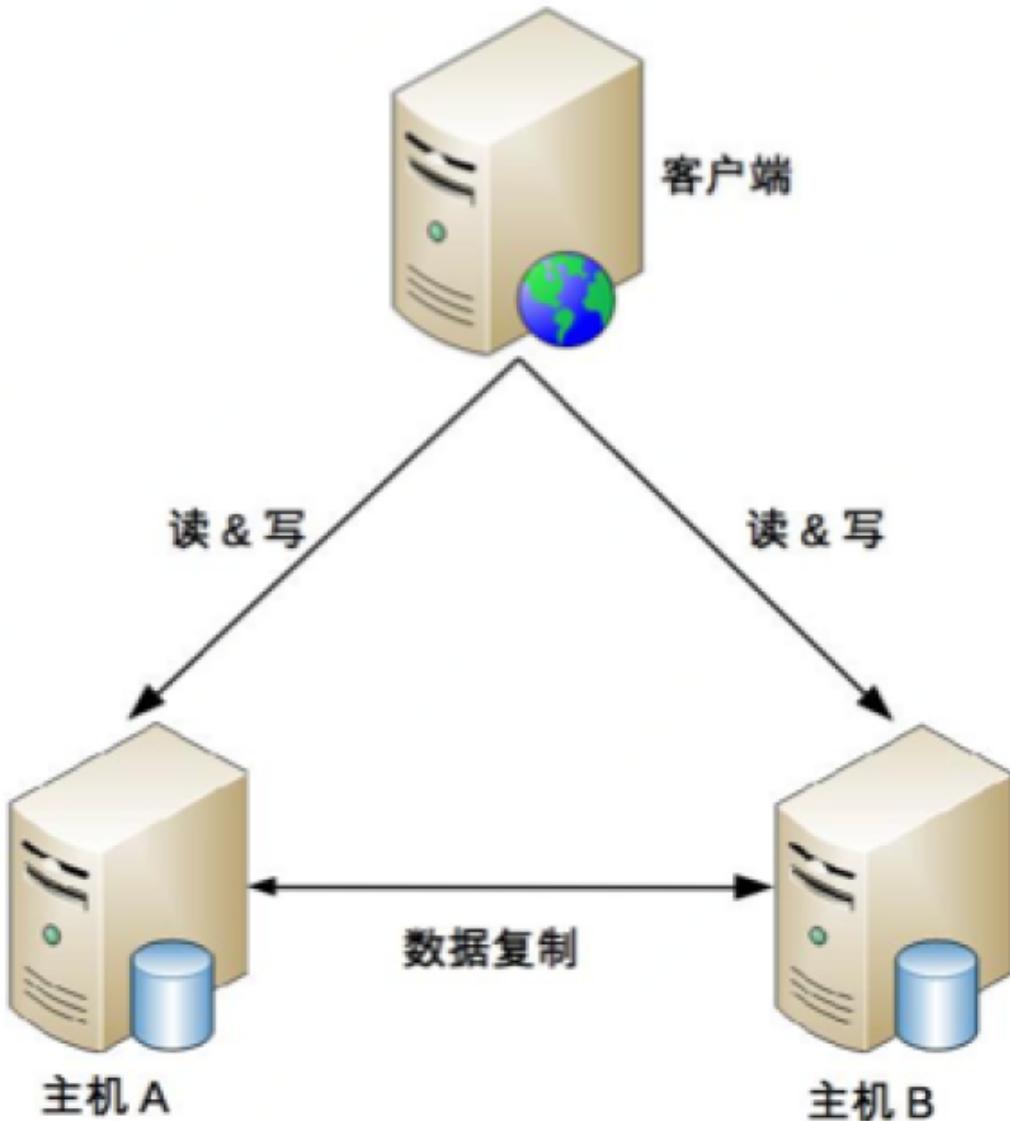


模拟式切换与互连式切换相比，优点是实现更加简单，因为省去了状态传递通道的建立和管理工作。

简单既是优点，同时也是缺点。因为模拟式读写操作获取的状态信息只有响应信息（例如，HTTP 404，超时、响应时间超过 3 秒等），没有互连式那样多样（除了响应信息，还可以包含CPU 负载、I/O 负载、吞吐量、响应时间等），基于有限的状态来做状态决策，可能出现偏差。

## 主主复制

主主复制是指两台服务器都是主机，互相将数据复制给对方，客户端可以任意挑选一台机器进行读写操作，下面是基本架构图：



相比于主备切换架构，主主复制架构有如下特点：

- ① 两台都是主机，不存在切换的概念
- ② 客户端无须区分不同角色的主机，随便将读写操作发送给哪台主机都可以

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。

因此，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。例如，用户登录产生的 session 数据（可以重新登录生成）、用户行为的日志数据（可以丢失）、论坛的草稿数据（可以丢失）等。

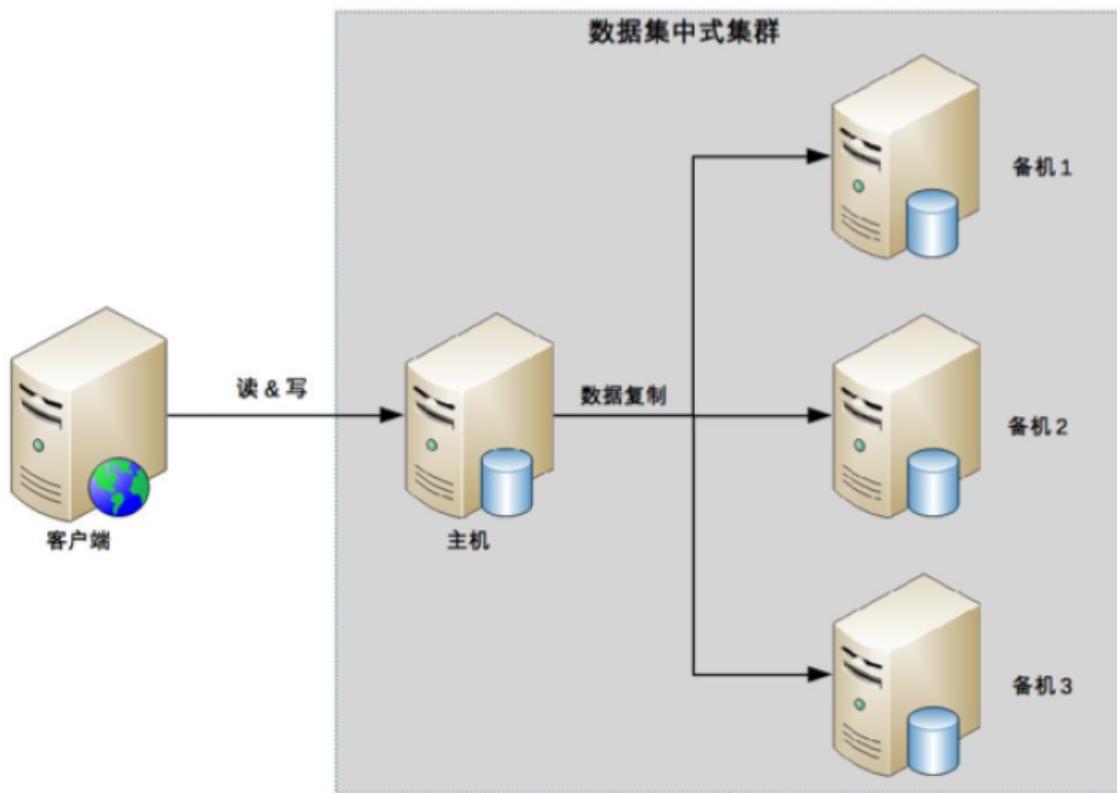
## 26. 高可用存储结构：集群和分区

### 数据集群

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。

#### 1、数据集中集群

数据集中集群与主备、主从这类架构类似，我们也可以称数据集中集群为1主多备或者1主多从。数据只能往主机中写，而读操作可以参考主备、主从架构进行灵活变化。



虽然架构上是类似的，但由于集群里的服务器数量更多，导致复杂度整体更高一些，具体体现在：

##### (1) 主机如何将数据复制给备机

主备和主从架构中，只有一条复制通道，而数据集中集群架构中，存在多条复制通道。多条复制通道首先肯定会增加主机复制的压力，某些场景下我们需要考虑如何降低主机的复制压力，或者降低主机复制给正常读写带来的压力。

其次，多条复制通道可能导致多个备机之间数据不一致，某些场景下我们需要对备机之间的数据一致性问题进行检查和修正。

### (2) 备机如何检测主机状态

主备架构中只有一台备机需要进行主机状态判断，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，而不同备机的判断状态有可能是不同的，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。

### (3) 主机故障后，如何确定新主机

究竟哪台备机升级为主机，备机之间如何协调，这也是一个复杂的问题。

目前开源的数据集群以zookeeper为典型，zookeeper通过ZAB协议来解决上述提到的几个问题，但ZAB算法的复杂度很高。

## 2、数据分散集群

数据分散集群指多个服务器组成一个集群，每个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，每天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散到不同的服务器上，算法需要考虑这些设计点：

### (1) 均衡性

算法需要保证服务器上的数据分区基本上是均衡的

### (2) 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配给其他服务器。

### (3) 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。

## 数据分区

数据分区指将数据按照一定的规则进行分区，不同分区分布在不同的地理位置上，每个分区存储一部分数据，通过这种方式来避免地理级别的故障造成的影响。采用了数据分区的架构后，即使某个地区发生严重的自然灾害或者事故，受影响的也只是一部分数据，而不是全部数据都不可用；当故障恢复后，其他地区备份的数据可以帮助故障地区快速恢复业务。

设计一个优良的数据分区架构，需要从多方面去考虑：

## 1、数据量

数据量的大小决定了分区的复杂度。数据量越大，需要的机器数量就越大。800台机器和4台机器根本无法同日而语。

- 800台机器中每周都可能有1、2台机器故障，从800台机器中定位两台并不是那么容易，运维复杂度很高。
- 新增加的服务器，分区相关的配置甚至规则需要修改，而每次修改理论上都有可能影响已有的800台服务器的运行，不小心改错配置的情况在实践中太常见了。
- 如此大量的数据，如果在地理位置上全部集中在某个城市，风险很大，遇到了水灾、大停电这种事故的时候，数据可能全部丢失，因此分区规则需要考虑地里容灾。

因此，数据量越大，分区规则会越复杂，考虑的情况也越多。

## 2、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。具体采用哪种或者哪几种规则，需要综合考虑业务范围、成本因素等。

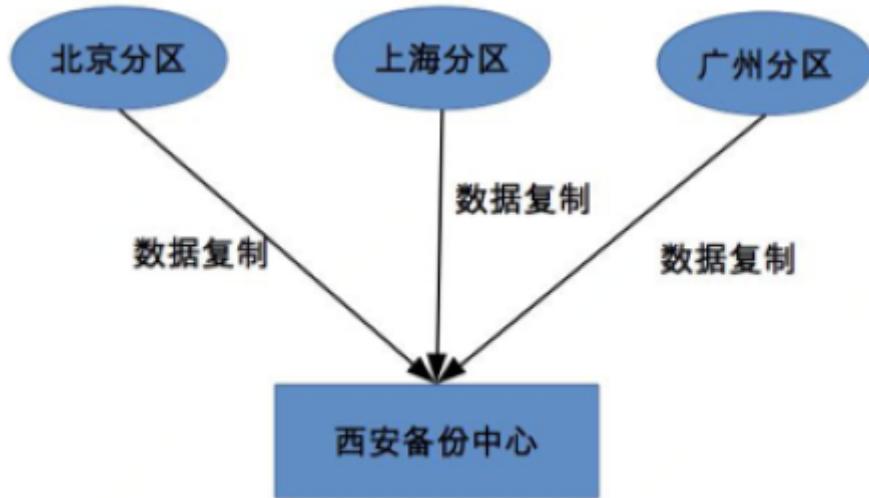
通常情况下，洲际分区主要用于面向不同大洲提供服务，由于跨洲通讯的网络延迟已经大到不适合提供在线服务了，因此洲际间的数据中心可以不互通或者仅仅作为备份；国家分区主要用于面向不同国家的用户提供服务，不同国家有不同语言、法律、业务等，国家间的分区一般也仅作为备份；城市分区由于都在同一个国家或者地区内，网络延迟较低，业务相似，分区同时对外提供服务，可以满足业务异地多活之类的需求。

## 3、复制规则

常见的分区复制规则有三种：集中式、互备式和独立式。

### (1) 集中式

集中式是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，其基本架构如下：

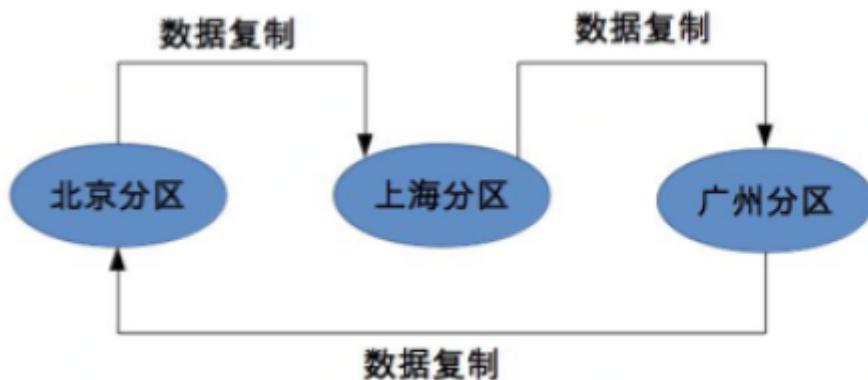


集中式备份的缺点为：

- 设计简单，各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，只需要将该分区的数据同样复制到“西安备份中心”即可。
- 成本较高，需要建立一个独立的备份中心。

## (2) 互备式

互备式备份指每个分区备份另外一个分区的数据，其基本架构如下：

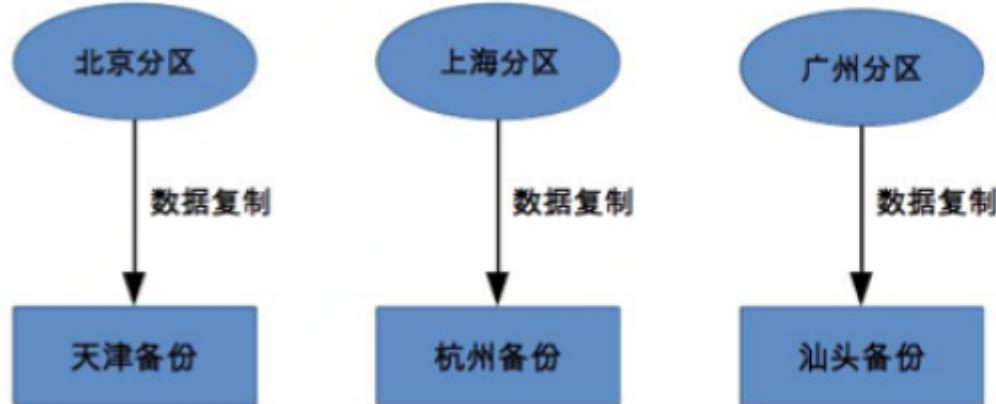


互备式架构的优缺点是：

- 设计比较复杂，各个分区除了承担业务数据存储，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，而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。

### (3) 独立式

独立式备份指每个分区自己有独立的备份中心，其基本架构如下：



有一个细节需要特别注意，各个分区的备份并不和原来的分区在一个地方。

独立式备份架构的优缺点：

- 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，因此独立式要比集中式成本要高的多。

## 27. 如何设计计算高可用

计算高可用：通过增加更多的服务器来达到计算高可用。

计算高可用架构的复杂度主要体现在任务管理方面，即当某个任务在某台机器上执行失败后，如何将任务重新分配到新的服务器进行执行。因此，计算高可用的设计关键点有如下两点：

### 1、哪些服务器可以执行任务

第一种方式和计算高性能中的集群类似，每个服务器都可以执行任务。

第二种方式和存储高可用中的集群类似，只有特定的服务器可以执行任务。当执行任务的服务器故障后，系统需要挑选新的服务器来执行任务。

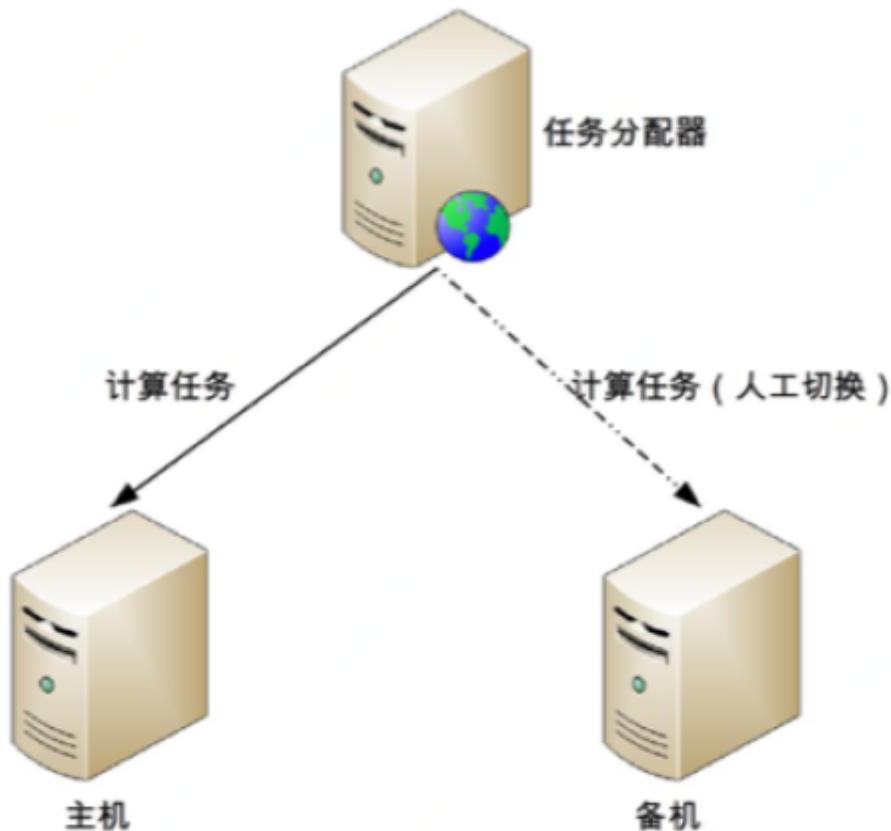
### 2、任务如何重新执行

第一种策略是对已经分配的任务即使处理失败也不做任何处理，系统只需要能够保证新的任务能够分配到其他非故障机器上，并能够成功执行。

第二种策略是设计一个任务管理器来管理需要执行的任务，服务器执行完任务后，需要向任务管理器反馈执行结果，任务管理器根据执行结果来决定是否需要将任务分配到另外的服务器上重新执行。

## 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：

- (1) 主机执行所有任务
- (2) 当主机故障时，任务分配器不会自动将任务发送给备机，此时系统处于不可用状态
- (3) 如果主机能够恢复，任务分配器将继续发送任务给主机
- (4) 如果主机不能恢复，则需要人工操作，将备机升级为主机，然后让任务分配器将任务分配给新的主机；同时为了保持主备架构，需要人工增加新的机器作为备机。

根据备机的不同状态，主备架构又可以细分为冷备架构和温备架构。

(1) 冷备：机器上的所有配置文件都和程序都准备好了，但备机业务系统没有启动，主机故障后需要人工手动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。

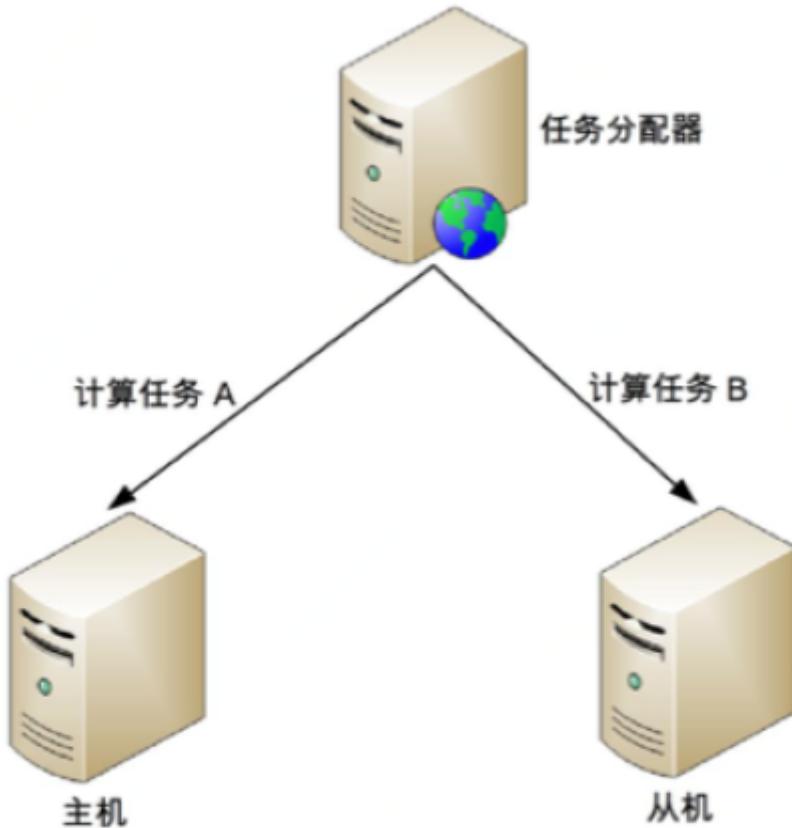
(2) 温备：备机上的业务系统已经启动，只是不对外提供服务，主机故障后，只需人工将任务分配的任务请求切换发送到备机即可。

主备架构的有点就是简单，主备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。而缺点正好也体现在“人工操作”这点上，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。

## 主从

和存储高可用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，其基本架构示意图如下：



主从方案详细设计：

- (1) 正常情况下，主机执行部分任务，备机执行部分任务。
- (2) 当主机故障后，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，不管这些任务是否成功。
- (3) 如果主机能够恢复，任务分配器会按照原有的这几策略继续分配任务。
- (4) 如果主机不能恢复，则需要人工操作，将原来的从机升级为主机，增加新的主机作为从机，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的优缺点：

- (1) 优点：主从架构的从机也执行任务，发挥了从机的硬件优势。
- (2) 缺点：主从架构需要将任务分类，任务分配器会复杂一些。

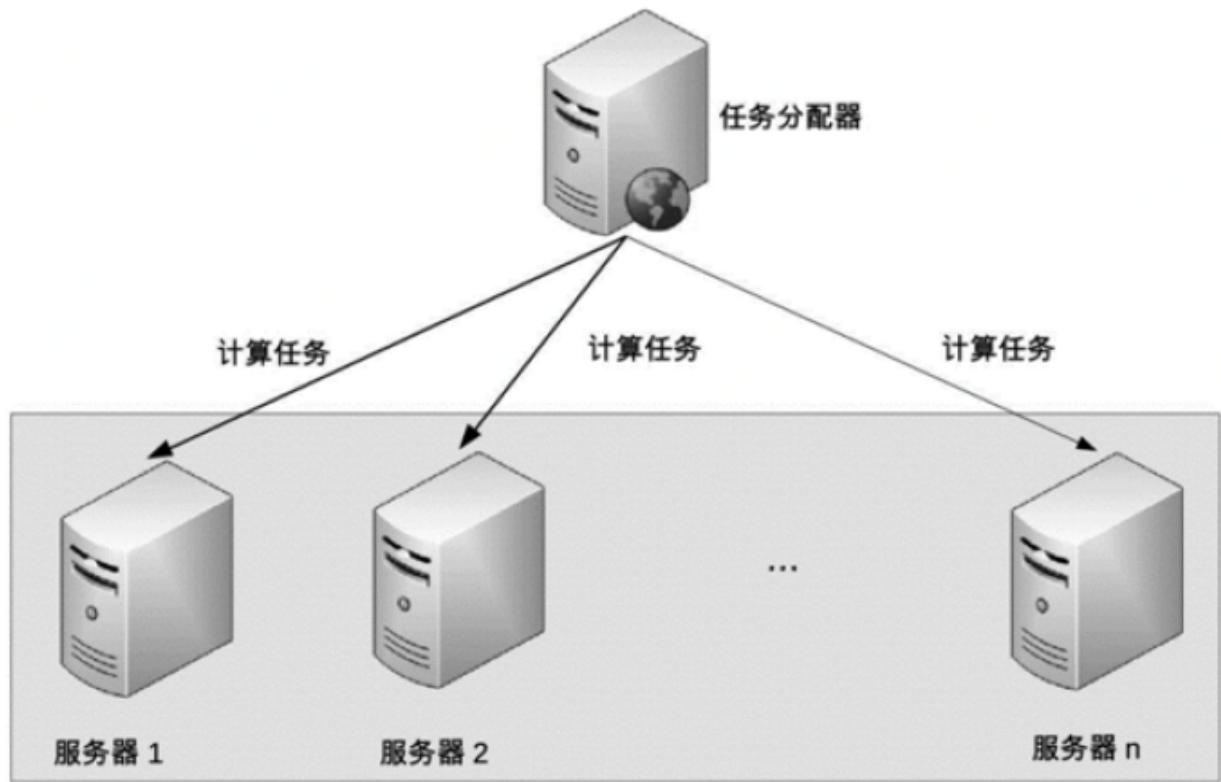
## 集群

高可用计算集群方案根据集群中服务器节点角色的不同，可以分为两类：

一类是对称集群，即集群中的每个服务器的角色都是一样的，都可以执行所有任务；另一类是非对称集群，集群中的服务器分为多个不同的角色，不同的角色执行不同的任务，例如最常见的Master-Slave角色。

- (1) 对称集群

对称集群更通俗的叫法是负载均衡集群，架构示意图如下：



负载均衡集群详细设计：

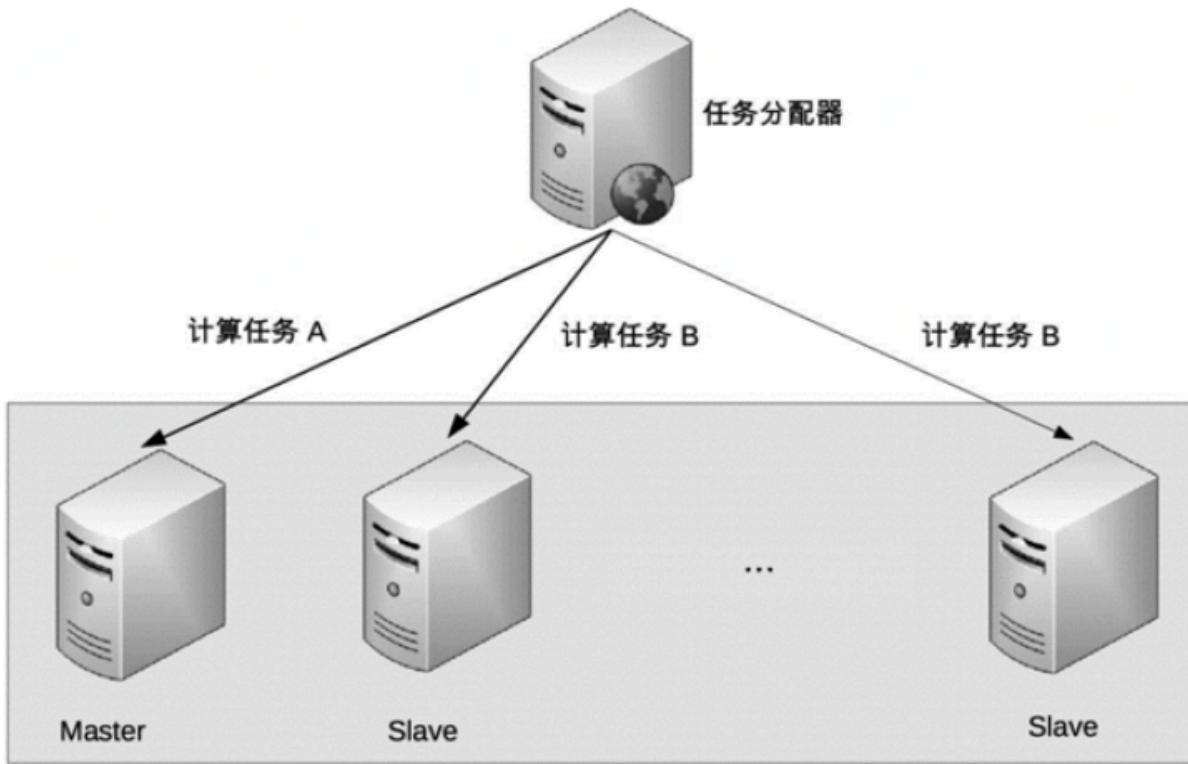
- 正常情况下，任务分配器采用某种策略（随机、轮询），将计算任务分配给集群中的不同服务器。
- 当集群中的某台服务器故障后，任务分配器不再将任务分配给它，而是将任务分配给其他服务器执行。
- 当故障服务器恢复后，任务分配器重新将分配给它执行。

负载均衡集群的设计关键在于两点：

- 任务分配器需要选取分配策略。
- 任务分配器需要检测服务器状态。

## (2) 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才能执行，部分任务是Slave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节点服务器，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服务器宕机后，需要从剩余的Slave服务器中重新指定一个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中剔除即可。

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的类型，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。

## 28. 业务高可用的保障：异地多活架构

判断一个系统是否满足异地多活，需要满足两个标准：

- (1) 正常情况下，无论用户访问哪一个地点的业务系统，都能够得到正确的业务服务。
- (2) 某个地方业务服务异常的时候，用户访问其他地方的正常的业务系统，能够得到正确的业务服务。

并不是任何业务系统都适合异地多活，因为实现异地多活架构的代价很高：

- (1) 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。
- (2) 成本会上升，毕竟要多在一个或者多个机房搭建独立一套业务系统。

## 架构模式

根据地理位置上的距离划分，异地多活架构可以分为同城异区、跨城异地、跨国异地。

### (1) 同城异区

同城的两个机房，距离上一般大约就是几十千米，通过搭建高速的网络，同城异区的两个机房能够实现和同一个机房内几乎一样的网络传输速度。这就意味着虽然是两个不同地理位置上的机房，但逻辑上我们可以将它们看作同一个机房，这样的设计大大降低了复杂度，减少了异地多活的设计和实现复杂度及成本。

### (2) 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。较远的距离，能够有效应对极端灾难事件。

跨城异地较远带来的网络传输延迟问题，给异地多活架构带来了复杂性，如果要做到真正意义上的多活，业务系统需要考虑部署在不同地点的两个机房，在数据短时间不一致的情况下，还能够正常提供业务。这就引入了一个看似矛盾的地方：数据不一致业务肯定不会正常，但跨城异地肯定会导致数据不一致。

如何解决这个问题呢？重点还是在“数据”上，即根据数据的特性来做不同的架构。

例如，用户登录（数据不一致时用户重新登录即可）、新闻类网站（一天内的新闻数据变化较少）、微博类网站（丢失用户发布的微博或者评论影响不大），这些业务采用跨城异地多活，能够很好地应对极端灾难的场景。

### (3) 跨国异地

跨国异地的“多活”，和跨城异地的“多活”，实际的含义并不完全一致。跨国异地多活的主要应用场景一般有这几种情况：

- (1) 为不同地区用户提供服务
- (2) 只读类型业务做多活

## 29. 异地多活设计4大技巧

跨城异地多活的一些设计技巧

技巧一：保证核心业务的异地多活

技巧二：保证核心数据最终一致

- (1) 尽量减少异地多活机房的距离，搭建高速网络
- (2) 尽量减少数据同步，只同步核心业务相关的数据
- (3) 保证最终一致性，不保证实时一致性

技巧三：采用多种手段同步数据

虽然绝大部分场景下，存储系统本身的同步功能基本上也就够用了，但在某些比较极端的情况下，存储系统本身的同步功能可能难以满足业务需求。

(1) 消息队列方式

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

(2) 二次读取方式

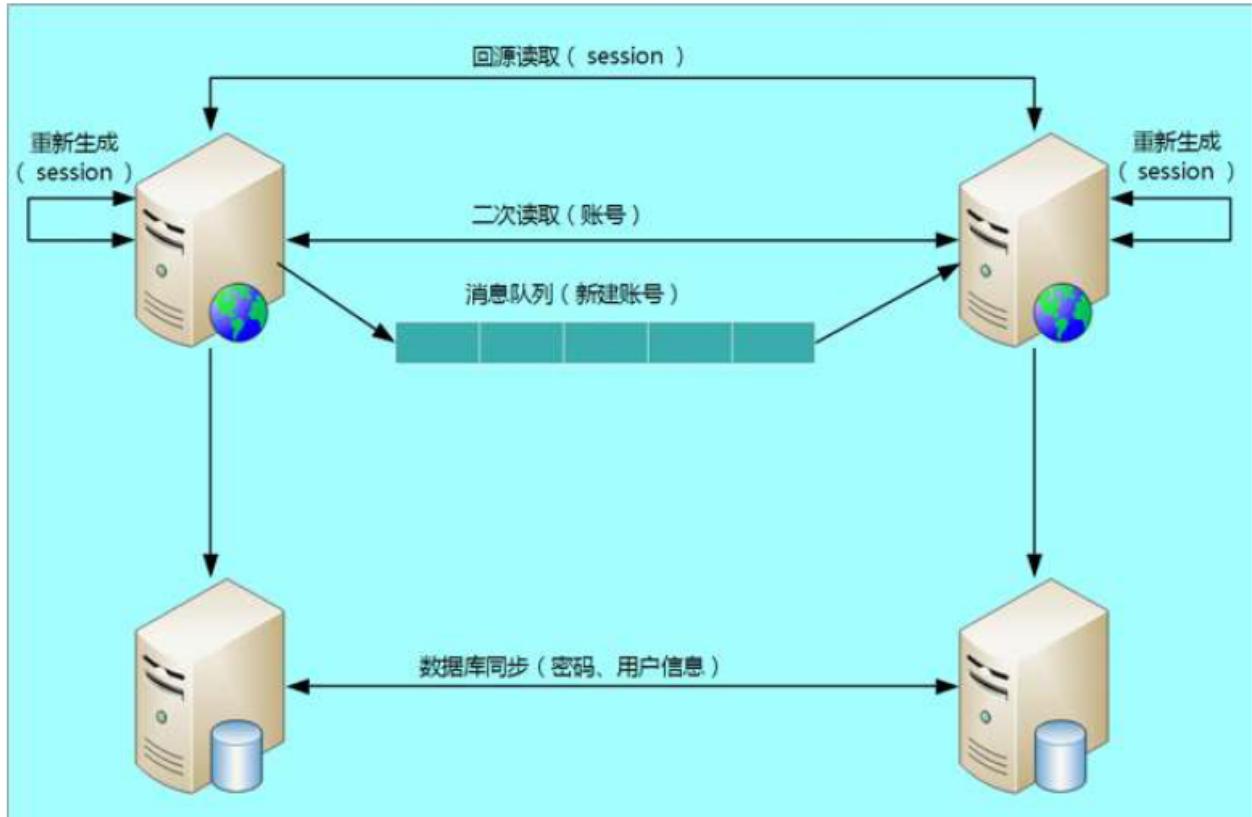
某些情况下可能出现消息队列同步也延迟了，用户在A中心注册，然后访问B中心的业务，此时B中心拿不到用户的信息。为了解决这个问题，B中心在读取本地失败后，可以根据路由规则，再去A中心访问一次，这样就能够解决异常情况下同步延迟的问题。

(3) 存储系统同步方式

对于密码数据，由于用户修改密码频率较低，而且用户不可能在1秒内连续多次修改密码，所以通过数据库的同步机制将数据同步到其他业务中心即可，用户信息数据和密码类似。

(4) 重新生成数据方式

也就是让用户重新操作，比如重新登录。



## 技巧四：只保证绝大多数用户的异地多活

放弃一部分用户，例如恰好密码没有同步造成的用户无法登录问题，用户信息不同步问题。毕竟只是少数用户。

对于这部分用户可以采取一些安抚和补偿：

(1) 挂公告

说明现在有问题和基本的问题原因，如果不明确原因或者不方便说出原因，可以发布“技术哥哥正在紧急处理”，这类比较轻松和有趣的公告。

(2) 事后对用户进行补偿

(3) 补充体验

对于为了做异地多活而带来的体验损失，可以想一些办法减少或者规避。以“转账申请”为例，为了让用户不用确认转账是否成功，我们可以在转账成功或者失败后直接给用户发个短信，告诉他转账结果，这样用户就不用时不时登录系统来确认转账是否成功了。

核心思想：采用多种手段，保证绝大部分用户的核心业务异地多活。

## 30. 异地多活设计4步走（跨城异地多活架构）

### 1、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，只为核心的业务设计异地多活，降低方案整体的复杂度和实现成本。

常见的分级标准有下面几种：

- (1) 访问量大的业务
- (2) 核心业务
- (3) 产生大量收入的业务

### 2、数据分类

挑选出核心业务后，需要对核心业务相关的数据进一步分析，目的在于识别所有的数据及数据特征，这些数据特征会影响后面的设计方案。

常见的数据特征分析纬度有：

- (1) 数据量
- (2) 唯一性
- (3) 实时性
- (4) 可丢失性
- (5) 可恢复性

我们同样以用户管理系统的登录业务为例，简单分析如下表所示。

数据	数据量	唯一性	实时性	可丢失性	可恢复性
用户ID	每天新增1万注册用户	全局唯一	5秒内同步	不可丢失	不可恢复
用户密码	每天1千用户修改密码	用户唯一	5秒内同步	可丢失	可重置密码恢复
登录session	每天1000万	全局唯一	无须同步	可丢失	可重复生成

### 3、数据同步

确定数据特点后，我们可以根据不同的数据设计不同的同步方案。

常见的同步方案有：

#### (1) 存储系统同步

这类数据同步的优点是使用简单，因为几乎主流的存储系统都会有自己的同步方案；缺点是这类同步方案都是通用的，无法针对业务数据的特点做定制化的控制。

#### (2) 消息队列同步

消息队列同步更适合无事务性或者无时序性要求的数据。

#### (3) 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。

我们同样以用户管理系统的登录业务为例，针对不同的数据特点设计不同的同步方案，如下表所示。

数据	数据量	唯一性	实时性	可丢失性	可恢复性	同步方案
用户ID	每天新增1万注册用户	全局唯一	5秒内同步	不可丢失	不可恢复	消息队列同步
用户密码	每天1千用户修改密码	用户唯一	5秒内同步	可丢失	可重置密码恢复	MySQL同步
登录session	每天1000万	全局唯一	无须同步	可丢失	可重复生成	重复生成

### 4、异常处理

异常处理主要有以下几个目的：

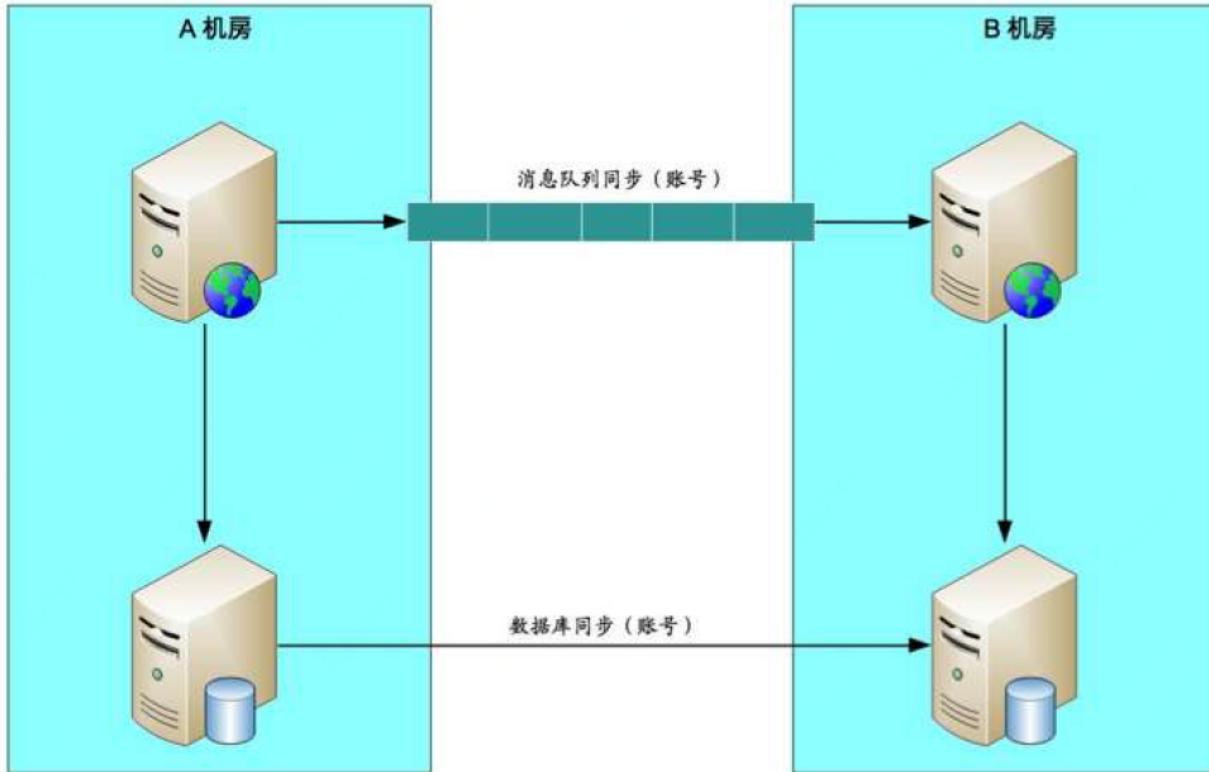
#### (1) 问题发生时，避免少量数据异常导致整体业务不可用。

(2) 问题恢复后，将异常的数据进行修正。

(3) 对用户进行安抚，弥补用户损失。

常见的异常处理措施有如下几类：

(1) 多通道同步

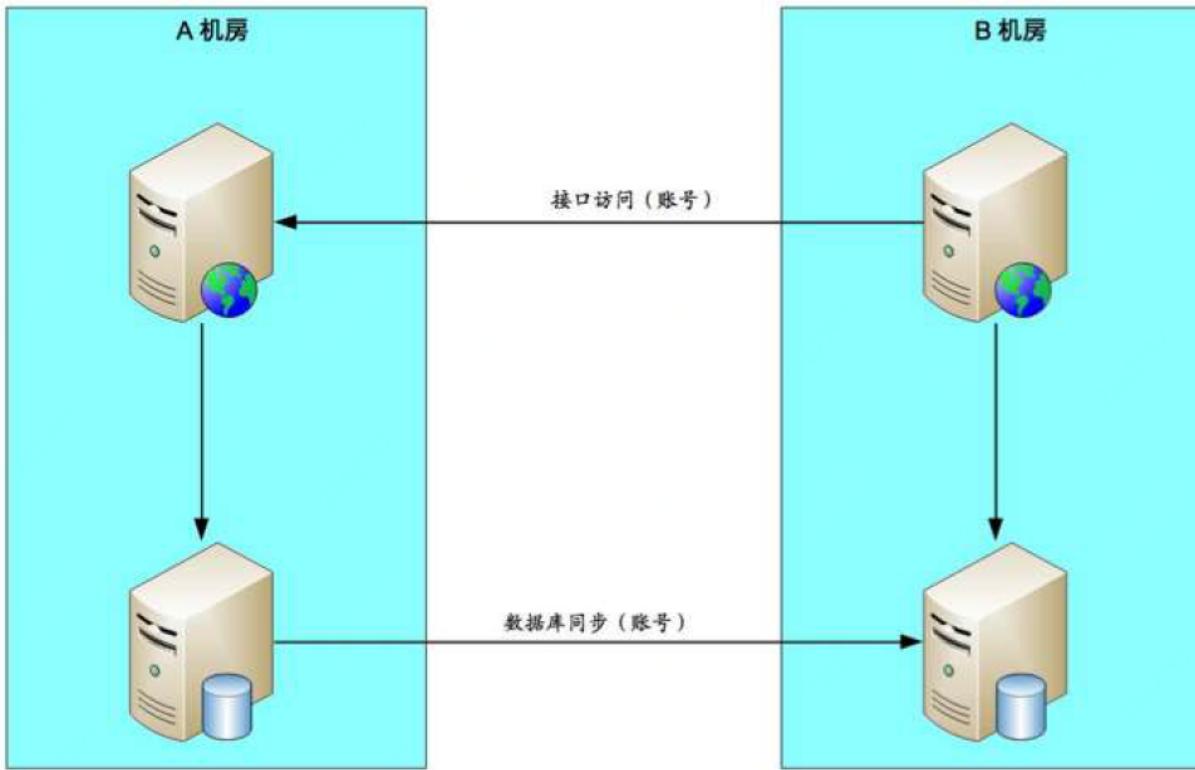


设计关键点：

- 一般情况下双通道即可，采取更多通道理论上能够降低风险，但付出的成本也会增加很多。
- 数据库同步通道和消息同步通道不能采用相同的网络连接，否则一旦网络故障，两个通道都同时故障。
- 需要数据是可以覆盖的，无论哪个通道的数据先到，最终结果是一样的。

(2) 同步和访问结合

这里的访问指异地机房通过系统的接口来进行数据访问。例如业务部署在异地两个机房A和B。B机房的业务系统通过A机房的接口来访问A机房的系统获取帐号信息。



设计关键点：

- 接口访问通道和数据库同步通道不能采用相同的网络连接。
- 数据有路由规则，可以根据数据来推断应该访问哪个机房的接口来读取数据。
- 由于有同步通道，优先读取本地数据，本都数据无法使用在通过接口去访问，这样可以大大降低跨机房的异地接口访问数量，适合实时性要求非常高的数据。

### (3) 日志记录

日志记录主要用于用户故障恢复后对数据进行恢复，其主要方式是每个关键操作前后都记录一条相关日志，然后将日志保存在一个独立的地方，当故障恢复后，通过日志对数据进行恢复。

为了应对不同级别的故障，日志保存的要求也不一样，常见的日志保存方式有：

- 服务器上保存日志，数据库中保存数据，这种方式可以应对单台数据库服务器故障或者宕机的情况。
- 本地独立系统保存日志，这种方式可以应对某业务服务器和数据库服务器同时宕机的情况。
- 日志异地保存，这种方式可以应对机房宕机的情况。

### (4) 用户补偿

## 31. 如何应对接口级的故障

导致接口级故障的原因一般有以下几个方面：

### (1) 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。

### (2) 外部原因

黑客攻击、促销或者抢购引入超出平时几倍甚至十几倍的用户，第三方系统大量请求，第三方系统相应慢等。

解决接口级故障的核心思想：优先保证核心业务和优先保证绝大部分用户。

## 1、降级

降级指系统将某些业务接口的功能降低，可以只提供部分功能，也可以完全停掉所有功能。

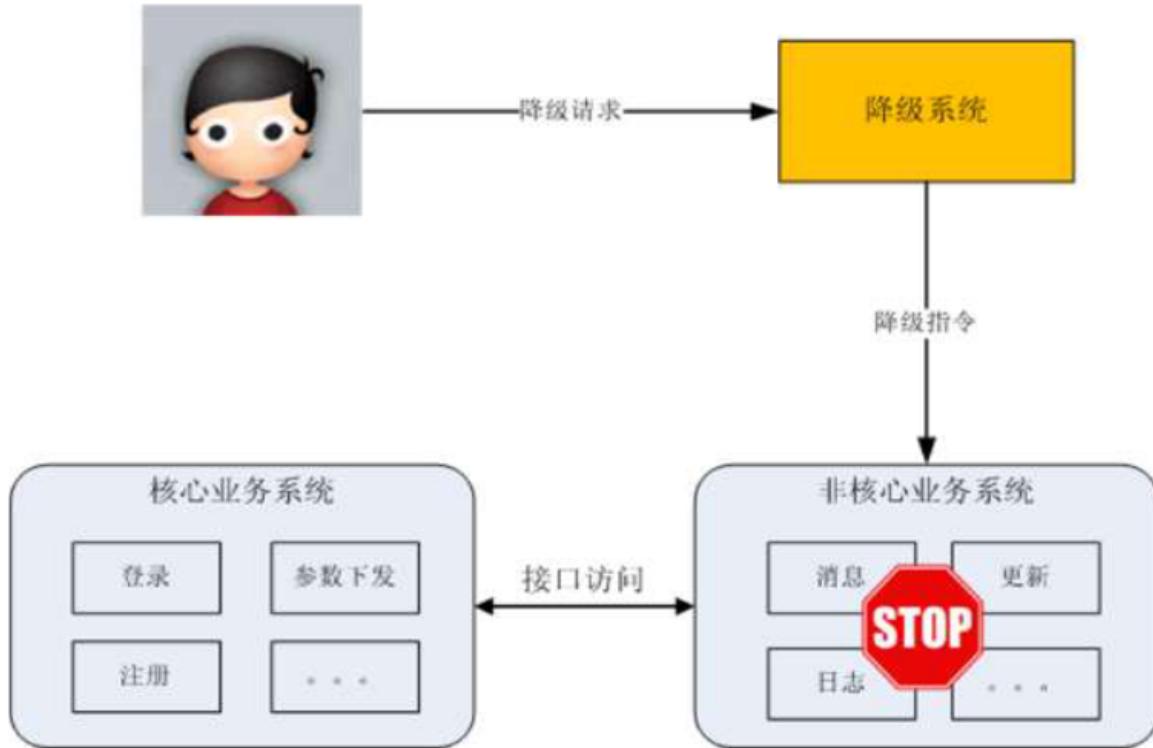
常见的降级方式有：

### (1) 系统后门降级

系统预留了后门用于降级操作。

### (2) 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审批，批量操作等功能。其基本架构如下：



## 2、熔断

降级的目的在于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。

## 3、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会被丢弃。虽然“丢弃”这个词听起来让人不舒服，但保证一部分请求能够正常响应，总比全部请求都不能响应要好的多。

限流方式可以分为两类：

### (1) 基于请求限流

基于请求的限流指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。

通常只能通过采用性能压测来确定阈值，但性能压测也存在覆盖场景有限的问题，可能出现某个性能压测没有覆盖的功能导致系统压力很大；另外一种方式是逐步优化，即：先设定一个阈值然后线上观察运行状况，发现不合理就调整阈值。

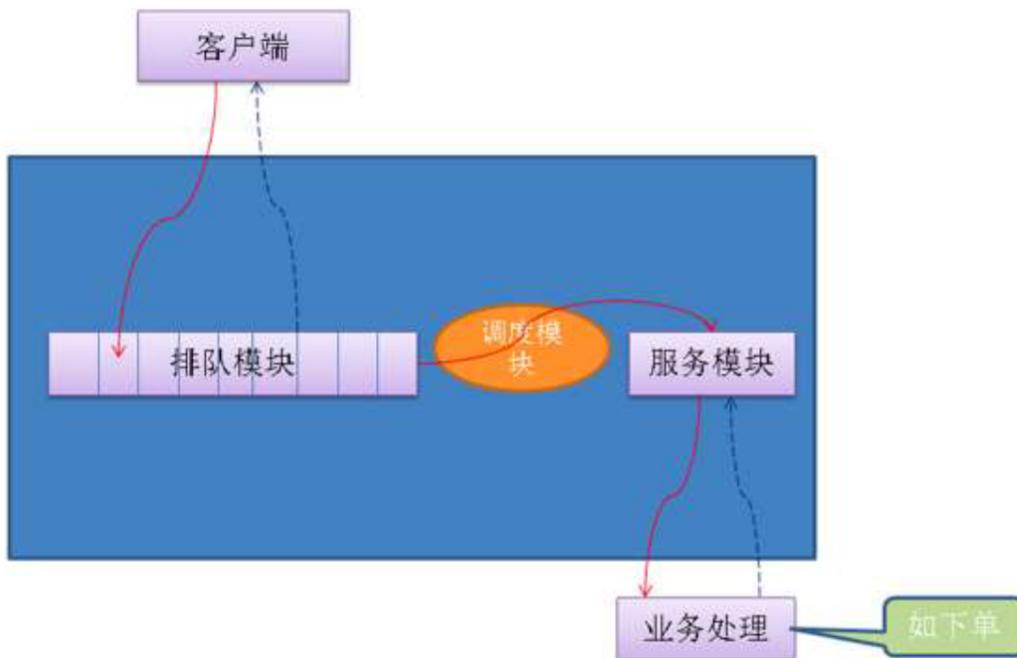
所以根据阈值来限制访问量的方式更多的适应于业务功能比较简单的系统。

### (2) 基于资源限流

基于资源限流相比基于请求限流能够更加有效地反应当前系统的压力，但实际设计中也面临两个主要的难点：如何确定关键资源，如何确定关键资源的阈值。通常情况下，这也是一个逐步调优的过程，即设计的时候先根据推断选择某个关键资源和阈值，然后测试验证，在上线观察，如果发现不合理再进行优化。

#### 4、排队

排队实际上是一个变种，限流是直接拒绝用户，排队是让用户等待一段时间。



## 32. 可扩展架构的基本思想和模式

可扩展架构的设计方法很多，但万变不离其宗，所有的可扩展架构设计，背后的基本思想都可以总结为一个字：拆！

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

- (1) 面向流程拆分：将整个业务流程拆分为几个阶段，每个阶段作为一个部分。
- (2) 面向服务拆分：将系统提供的服务拆分，每个服务作为一个部分。
- (3) 面向功能拆分：将系统提供的功能拆分，每个功能违一个部分。

不同的拆分方式，本质上决定了系统的扩展方式。

不同拆分方式应对扩展时的优势：

### 1、面向流程拆分

扩展时大部分情况下只需要修改某一层，少部分情况可能修改关联的两层，不会出现所有的层都同时要修改。

### 2、面向服务拆分

对某个服务扩展，或者要增加新服务时，只需要扩展相关服务即可，无须修改所有的服务。

### 3、面向功能拆分

对某个功能进行扩展，或者要增加新的功能时，只需要扩展相关功能即可，无须修改所有的功能。

不同的拆分方式将得到不同的系统架构，经典的可扩展系统架构有：

1、面向流程拆分：分层架构

2、面向服务拆分：SOA、微服务

3、面向功能拆分：微内核架构

当然，这几个系统架构并不是非此即彼的，而是可以在系统架构中进行组合使用的。

## 33. 传统的可扩展架构模式：分层架构和SOA

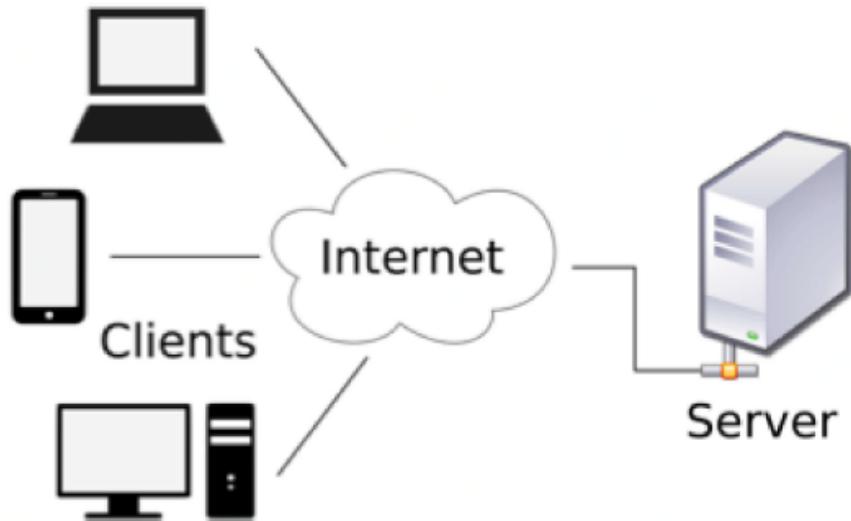
### 分层架构

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。

按照分层架构进行设计时，根据不同的划分纬度和对象，可以得到多种不同的分层架构。

#### 1、C/S架构、B/S架构

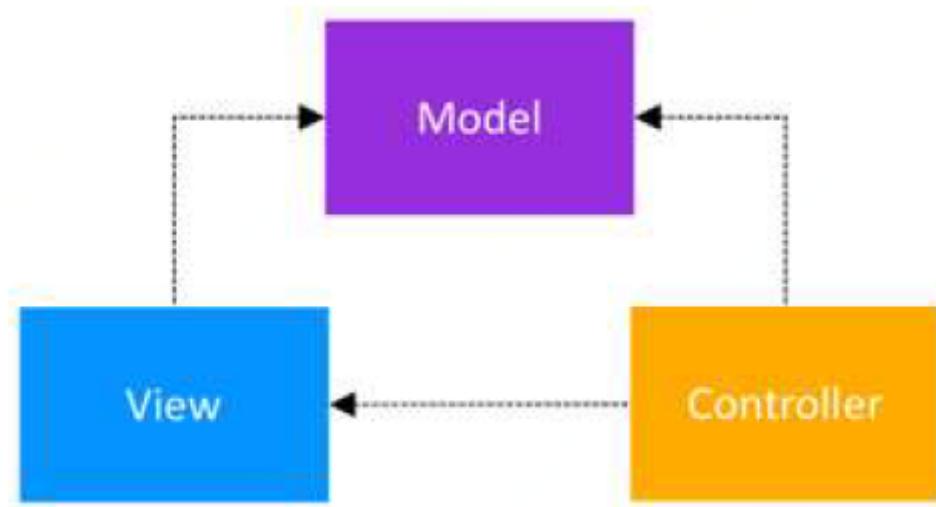
划分的对象是整个业务系统，划分的纬度是用户交互，即将和用户交互的部分独立为一层，支撑用户交互的后台作为另一层。下图是标准的C/S架构。



( <http://www.bkjia.com/uploads/allimg/150507/0424434N3-0.png> )

## 2、MVC架构、MVP架构

划分的对象是单个业务子系统，划分的纬度是职责，将不同的职责划分到独立层，但各层的依赖管理比较灵活。例如，MVC各层之间是两两交互的。

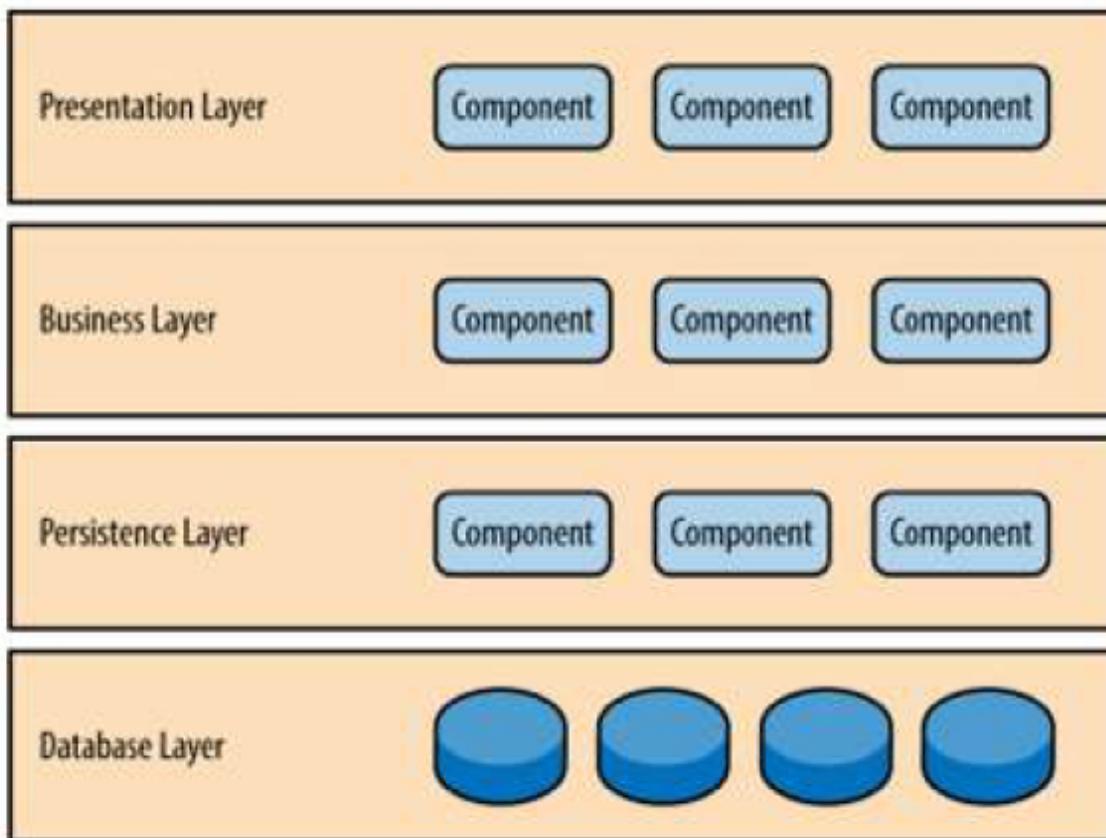


## 3、逻辑分层架构

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，划分的纬度也是职责。虽然都是基于职责的划分，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。典型的有操作系统内核架构、TCP/IP架构。例如，下面是Android操作系统的架构。



典型的J2EE系统架构也是逻辑分层架构，架构图如下：



分层架构之所以能够很好的支撑系统扩展，本质在于隔离关注点（separation of concerns），即每个层的组件只会处理本层的逻辑。

# SOA

SOA的全称是 Service Oriented Architecture，中文翻译为“面向服务的架构”，诞生于上世纪90年代。SOA出现的背景是企业内部的IT系统重复建设效率低下。

SOA提出了3个关键概念：

## 1、服务

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

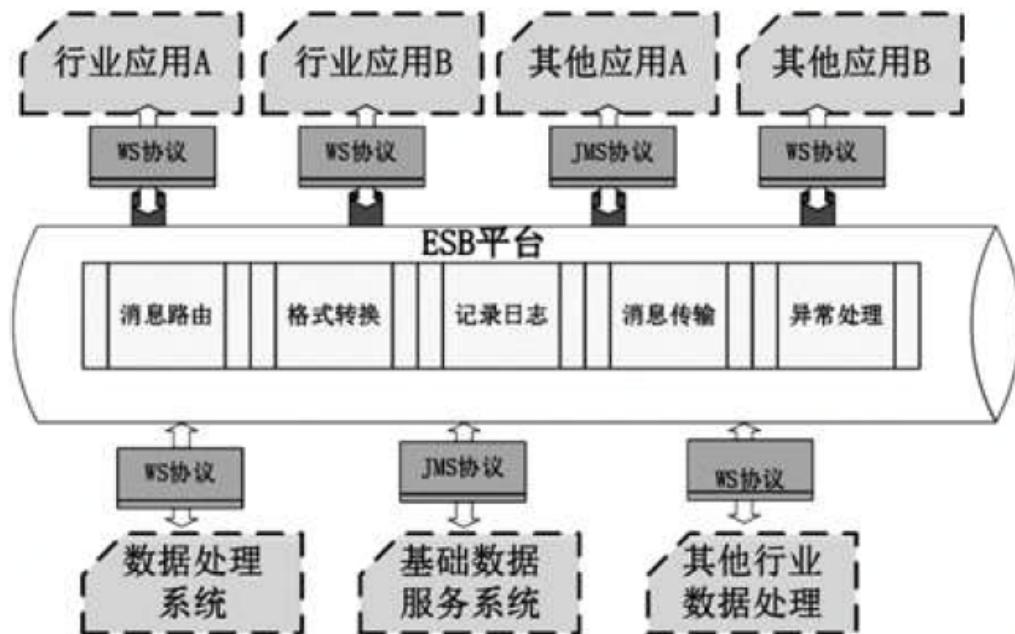
## 2、ESB

ESB将企业中各个不同的服务连接起来。因为各个独立的服务是异构的，如果没有统一的标准，则各个异构的系统对外提供的结构会是各式各样的。SOA使用ESB来屏蔽。

## 3、松耦合

松耦合是减少各个服务间的依赖和相互影响。

典型的 SOA 架构样例如下：



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架构，但不会说某个独立系统采用了SOA架构。