极客时间训练营

系统架构设计师

第10章 软件架构的演化和维护

授课：王建平

目录

目录

软件架构的演化和定义的关系

◆软件架构的演化和维护的目的是为了使软件能够适应环境的变化而进行的纠错性修改和完善性修改。软件架构的演化和维护过程是一个不断迭代的过程，通过演化和维护，软件架构逐步得到完善，以满足用户需求。（★）

◆软件架构的演化就是软件整体结构的演化，演化过程涵盖软件架构的全生命周期，包括软件架构需求的获取、软件架构建模、软件架构文档、软件架构实现以及软件架构维护等阶段。人们通常说软件架构是演化来的，而不是设计来的。（★★）

软件架构定义是SA={组件，连接件，约束}（★）
1、组件是软件架构的基本要素和结构单元，表示系统中主要的计算元素、数据存储以及一些重要模块，当需要消除软件架构存在的缺陷、新增功能、适应新的环境时都涉及组件的演化。组件的演化体现在组件中模块的增加、删除或修改。
2、连接件是组件间的交互关系，多数情况下组件的演化牵涉到连接件的演化。连接件的演化体现在组件交互消息的增加、删除或改变。
3、约束是组件和连接件之间的拓扑关系和配置，它为组件和连接件提供额外数据支撑，可以是架构的约束数据，或架构的参数。

软件架构演化方式的分类

软件架构演化典型的分类方法:（★）

◆按照软件架构的实现方式和实施粒度分类：基于过程和函数的演化、面向对象的演化、基于组件的演化和基于架构的演化。

◆按照研究方法将软件架构演化方式分为4类:
- ✓ 第1类是对演化的支持，如代码模块化的准则、可维护性的支持(如内聚和耦合)、代码重构等;
- ✓ 第2类是版本和工程的管理工具，如CVS 和COCOMO;
- ✓ 第3类是架构变换的形式方法，包括系统结构和行为变换的模型，以及架构演化的重现风格等;
- ✓ 第4类是架构演化的成本收益分析决定如何增加系统的弹性。

◆针对软件架构的演化过程是否处于系统运行时期，将软架构演化分为静态演化和动态演化 。
- ✓ 静态演化发生在软件架构的设计、实现和维护过程中，软件系统还未运行或者处在运行停止状态。
- ✓ 动态演化发生在软件系统运行过程中。

软件架构演化时期

软件架构演化时期（★★）

◆设计时演化

设计时演化是指发生在体系结构模型和与之相关的代码编译之前的软件架构演化。

◆运行前演化

运行前演化是指发生在编译之后、执行之前的软件架构演化，这时由于应用程序并未执行，修改时不用考虑应用程序的状态，但需要考虑系统的体系结构，且系统需要具有添加和删除组件的机制。

◆有限制运行时演化

有限制运行时演化是指系统在设计时就规定了演化的具体条件，将系统置于“安全”模式下，演化只发生在某些特定约束满足时，可以进行一些规定好的演化操作。

◆运行时演化

运行时演化是指系统的体系结构在运行时不能满足要求时发生的软件架构演化，包括添加组件、删除组件、升级替换组件、改变体系结构的拓扑结构等。此时的演化是最难实现的。

软件架构静态演化

软件架构静态演化

◆静态演化需求

(1)设计时演化需求。在架构开发和实现过程中对原有架构进行调整，保证软件实现与架构的一致性以及软件开发过程的顺利进行。

(2)运行前演化需求。软件发布之后由于运行环境的变化，需要对软件进行修改升级，在此期间软件的架构同样要进行演化。

◆静态演化的一般过程

软件静态演化是系统停止运行期间的修改和更新，即一般意义上的软件修复和升级。与此时相对应的维护方法有三类：更正性维护、适应性维护、完善性维护。

软件的静态演化包括如下 5 个步骤：（★）

- ✓ 软件理解：查阅软件文档，分析软件架构，识别系统组成元素及其之间的相互关系，提取系统的抽象表示形式。
- ✓ 需求变更分析：静态演化往往是由于用户需求变化、系统运行出错和运行环境发生改变等原因所引起的，需要找出新的软件需求与原有的差异。
- ✓ 演化计划：分析原系统，确定演化范围和成本，选择合适的演化计划。
- ✓ 系统重构：根据演化计划对系统进行重构，使之适应当前的需求。
- ✓ 系统测试：对演化后的系统进行测试，查找其中的错误和不足。

软件架构动态演化

动态演化是在系统运行期间的演化，在不停止系统功能的情况下完成演化，较之静态演化更加困难。具体发生在有限制的运行时演化和运行时演化阶段。

◆动态演化的类型（★）

1)软件动态性的等级

软件的动态性分为3 个级别：

- ✓ 交互动态性，要求数据在固定的结构下动态交互。
- ✓ 结构动态性，允许对结构进行修改，通常形式是组件和连接件实例的添加和删除，这种动态性是研究和应用的主流。
- ✓ 架构动态性，允许软件架构的基本构造变动，即结构可以被重定义，如新的组件类型的定义。

2)动态演化的内容

根据所修改的内容不同，软件的动态演化包括以下 4 个方面：

- ✓ 属性改名：目前所有的ADL都支持对非功能属性的分析和规约，而在运行过程中，用户可能会对这些指标进行重新定义(如服务响应时间)。
- ✓ 行为变化：在运行过程中，用户需求变化或系统自身服务质量的调节都将引发软件行为的变化。如，为了提高安全级别而更换加密算法；将HTTP协议改为HTTPS协议。
- ✓ 拓扑结构改变：如增删组件，增删连接件，改变组件与连接件之间的关联关系等。
- ✓ 风格变化：一般软件演化后其架构风格应当保持不变，如果非要改变软件的架构风格也只能将架构风格变为衍生风格。如将两层C/S结构调整为三层C/S结构等。

典型真题

按照软件架构的实现方式和实施粒度分类：基于过程和函数的演化、面向对象的演化、（）的演化和基于架构的演化。
A.基于结构　B.基于数据流　　C.基于风格　　D.基于组件
参考答案：D

（）是指发生在编译之后、执行之前的软件架构演化，这时由于应用程序并未执行，修改时不用考虑应用程序的状态，但需要考虑系统的体系结构，且系统需要具有添加和删除组件的机制。
A.运行前演化　B.运行中演化　　C.运行后演化　　D.执行演化
参考答案：A

目录

架构演化原则

列举 18 种软件架构可持续演化原则，并针对每个原则设计了相应的度量方案。

1.演化成本控制原则（★★）
- ✓ 原则名称：演化成本控制 (ECC)原则。
- ✓ 原则解释：演化成本要控制在预期的范围之内，也就是演化成本要明显小于重新开发成本。
- ✓ 原则用途：用于判断架构演化的成本是否在可控范围内，以及用户是否可接受。
- ✓ 度量方案：CoE<<CoRD。
- ✓ 方案说明：CoE为演化成本，CoRD为重新开发成本，CoE远小于CoRD最佳。

2.进度可控原则（★★）
- ✓ 原则名称：进度可控原则。
- ✓ 原则解释：架构演化要在预期时间内完成，也就是时间可控。
- ✓ 原则用途：根据该原则可以规划每个演化过程的任务量；体现一种迭代、递增(持续演化)的演化思想。
- ✓ 度量方案:：ttask=|Ttask-T'task|。
- ✓ 方案说明：某个演化任务的实际完成时间 (Ttask) 和预期完成时间 (T'task)的时间差ttask越小越好。

架构演化原则

3.风险可控原则（★★）

- ✓ 原则名称：风险可控原则。
- ✓ 原则解释：架构演化过程中的经济风险、时间风险、人力风险、技术风险和环境风险等必须在可控范围内。
- ✓ 原则用途：用于判断架构演化过程中各种风险是否易于控制。
- ✓ 度量方案：分别检验。
- ✓ 方案说明：时间风险、经济风险、人力风险、技术风险都不存在。

4.主体维持原则（★★）

- ✓ 原则名称：主体维持原则
- ✓ 原则解释：对称稳定增长(AIG)原则所有其他因素须与软件演化协调，开发人员、销售人员、用户必须熟悉软件演化的内容，从而达到令人满意的演化。因此，软件演化的平均增量的增长须保持平稳，保证软件系统主体行为稳定。
- ✓ 原则用途：用于判断架构演化是否导致系统主体行为不稳定。
- ✓ 度量方案：计算AIG即可，AIG=主体规模的变更量/主体的规模。
- ✓ 方案说明：根据度量动态变更信息(类型、总量、范围) 来计算

架构演化原则

5.系统总体结构优化原则（★★）
✓ 原则名称：系统总体结构优化原则。
✓ 原则解释：架构演化要遵循系统总体结构优化原则，使得演化之后的软件系统整体结构(布局)更加合理。
✓ 原则用途：用于判断系统整体结构是否合理，是否最优。
✓ 度量方案：检查系统的整体可靠性和性能指标。
✓ 方案说明：判断整体结构优劣的主要指标是系统的可靠性和性能。

6.平滑演化原则（★★）
✓ 原则名称：平滑演化(IWR)原则
✓ 原则解释：在软件系统的生命周期里，软件的演化速率趋于稳定，如相邻版本的更新率相对固定。
✓ 原则用途：用于判断是否存在剧烈架构演化。
✓ 度量方案：计算IWR即可，IWR=变更总量/项目规模。
✓ 方案说明：根据度量动态变更信息 (类型、总量、范围等) 来计算。

架构演化原则

7.目标一致原则（★★）
- ✓ 原则名称：目标一致原则
- ✓ 原则解释：架构演化的阶段目标和最终目标要一致。
- ✓ 原则用途：用于判断每个演化过程是否达到阶段目标，所有演化过程结束是否能达到最终目标。
- ✓ 度量方案：otask=|Otask-O'task|。
- ✓ 方案说明：阶段目标的实际达成情况 (Otask) 和预期目标(O'task)的差otask越小越好。

8.模块独立演化原则（★★）
- ✓ 原则名称：模块独立演化原则，或修改局部化原则。
- ✓ 原则解释：软件中各模块(相同制品的模块，如Java的某个类或包)自身的演化最好相互独立，或者至少保证对其他模块的影响比较小或影响范围比较小。
- ✓ 原则用途：用于判断每个模块自身的演化是否相互独立。
- ✓ 度量方案：检查模块的修改是否是局部的。
- ✓ 方案说明：可以通过计算修改的影响范围来进行度量。

架构演化原则

9.影响可控原则（★★）
- ✓ 原则名称：影响可控原则。
- ✓ 原则解释：软件中一个模块如果发生变更，其给其他模块带来的影响要在可控范围内也就是影响范围可预测。
- ✓ 原则用途：用于判断是否存在对某个模块的修改导致大量其他修改的情况。
- ✓ 度量方案：检查影响的范围是否可控。
- ✓ 方案说明：通过计算修改的影响范围来进行度量。

10.复杂性可控原则（★★）
- ✓ 原则名称：复杂性可控(CC)原则。
- ✓ 原则解释：架构演化必须要控制架构的复杂性，从而进一步保障软件的复杂性在可控范围内。
- ✓ 原则用途：用于判断演化之后的架构是否易维护、易扩展、易分析、易测试等。
- ✓ 度量方案：CC小于某个阈值。
- ✓ 方案说明：CC增长可控。

架构演化原则

11.有利于重构原则（★★）
- ✓ 原则名称：有利于重构原则。
- ✓ 原则解释：架构演化要遵循有利于重构原则，使得演化之后的软件架构更方便重构。
- ✓ 原则用途：用于判断架构易重构性是否得到提高。
- ✓ 度量方案：检查系统的复杂度指标。
- ✓ 方案说明：系统越复杂越不容易重构。

12.有利于重用原则（★★）
- ✓ 原则名称：有利于重用原则。
- ✓ 原则解释：架构演化最好能维持，甚至提高整体架构的可重用性。
- ✓ 原则用途：用于判断整体架构可重用性是否遭到破坏。
- ✓ 度量方案：检查模块自身的内聚度、模块之间的耦合度。
- ✓ 方案说明：模块的内聚度越高，该模块与其他模块之间的耦合度越低，越容易重用。

架构演化原则

13.设计原则遵从性原则（★★）

- ✓ 原则名称：设计原则遵从性原则。
- ✓ 原则解释：架构演化最好不能与架构设计原则冲突。
- ✓ 原则用途：用于判断架构设计原则是否遭到破坏(架构设计原则是好的设计经验总结，要保障其得到充分使用)。
- ✓ 度量方案：RCP=|CDP| / |DP|。
- ✓ 方案说明：冲突的设计原则集合 (CDP) 和总的设计原则集合(DP)的比较，RCP越小越好。

14.适应新技术原则（★★）

- ✓ 原则名称：适应新技术(TI) 原则。
- ✓ 原则解释：软件要独立于特定的技术手段，这样才能够让软件运行于不同平台。
- ✓ 原则用途：用于判断架构演化是否存在对某种技术依赖过强的情况。
- ✓ 度量方案：TI=1-DDT，DDT=|依赖的技术集合|/|用到的技术合集|。
- ✓ 方案说明：根据演化系统对关键技术的依赖程度进行度量。

架构演化原则

15.环境适应性原则（★★）
- ✓ 原则名称：环境适应性原则。
- ✓ 原则解释：架构演化后的软件版本能够比较容易适应新的硬件环境与软件环境。
- ✓ 原则用途：用于判断架构在不同环境下是否仍然可使用，或者容易进行环境配置。
- ✓ 度量方案：硬件/软件兼容性。
- ✓ 方案说明：结合软件质量中兼容性指标进行度量。

16.标准依从性原则（★★）
- ✓ 原则名称：标准依从性原则。
- ✓ 原则解释：架构演化不会违背相关质量标准(国际标准、国家标准、行业标准、企业标准等)。
- ✓ 原则用途：用于判断架构演化是否具有规范性，是否有章可循；而不是胡乱或随意地演化。
- ✓ 度量方案：需要人工判定。

架构演化原则

17.质量向好原则（★★）

- ✓ 原则名称：质量向好(QI)原则。
- ✓ 原则解释：通过演化使得所关注的某个质量指标或某些质量指标的综合效果变得更好或者更满意，例如可靠性提高了。
- ✓ 原则用途：用于判断架构演化是否导致某些质量指标变得很差。
- ✓ 度量方案：EQI>SQ。
- ✓ 方案说明：演化之后的质量(EQI) 比原来的质量(SQ) 要好。

18.适应新需求原则（★★）

- ✓ 原则名称：适应新需求原则。
- ✓ 原则解释：架构演化要容易适应新的需求变更；架构演化不能降低原有架构适应新需求的能力；架构演化最好可以提高适应新需求的能力。
- ✓ 原则用途：用于判断演化之后的架构是否降低了架构适应新需求的能力。
- ✓ 度量方案：RNR=|ANR|/|NR|。
- ✓ 方案说明：适应的新需求集合 (ANR) 和实际新需求集合 (NR)的比较，RNR越小越好。

典型真题

（）原则是架构演化后的软件版本能够比较容易适应新的硬件环境与软件环境。
A.环境适应性原则　B. 质量向好原则　　C.适应新技术原则　　D.适应新需求原则
参考答案：A

软件架构演化评估方法

根据演化过程是否已知可将评估过程分为：

✓ 演化过程已知的评估

✓ 演化过程未知的评估

一、演化过程已知的评估

演化过程已知的评估其目的在于通过对架构演化过程进行度量，比较架构内部结构上的差异以及由此导致的外部质量属性上的变化，对该演化过程中相关质量属性进行评估。涉及评估流程以及具体的相关指标的计算方法。

1.评估流程

架构演化评估的基本思路是将架构度量应用到演化过程中，通过对演化前后的不同版本的架构分别进行度量，得到度量结果的差值及其变化趋势，并计算架构间质量属性距离，进而对相关质量属性进行评估。

软件架构演化评估方法

2.架构演化中间版本度量（★★）
对于不同类型的质量属性，其度量方法不同，度量结果的类型也不同。本章主要度量的是架构的可维护性和可靠性。

- ✓ 对于可靠性，度量结果是一个实数值。
- ✓ 对于可维护性，它包含圈复杂度、扇入扇出度、模块间耦合度、模块的响应、紧内聚度、松内聚度等 6个子度量指标。通过比较原子演化前后架构质量属性 Qi-1和Qi间的变化，可以分析该类演化对评估系统的外部质量属性的影响，进而找出架构内部结构变化和外部质量属性变化间的关联。

二、演化过程未知的评估
演化过程未知时，我们无法像演化过程已知时那样追踪架构在演化过程中的每一步变化，只能根据架构演化前后的度量结果逆向推测出架构发生了哪些改变，并分析这些改变与架构相关质量属性的关联关系。

目录

案例引入

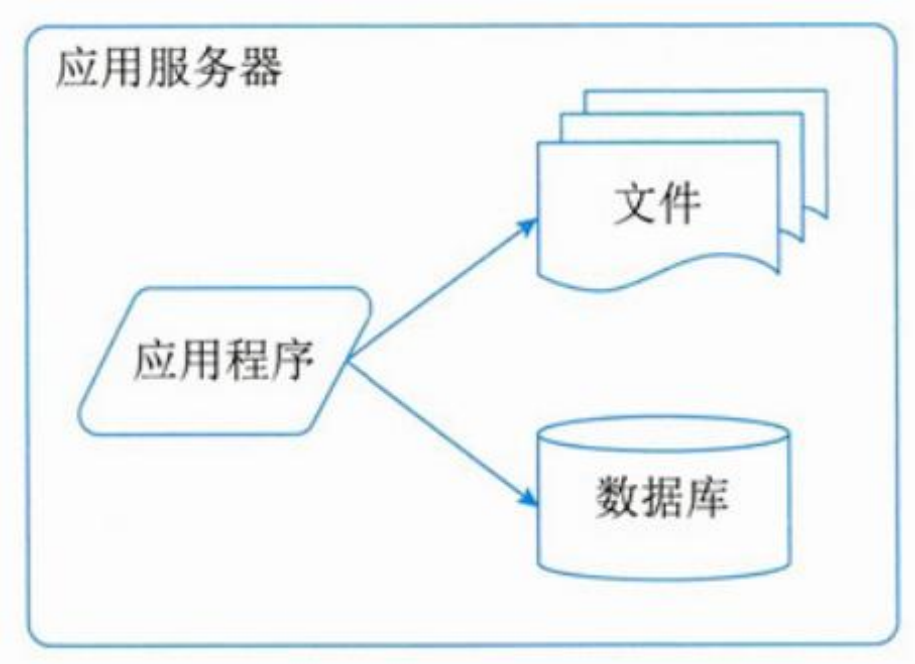
案例
热点新闻、热点话题、京东天猫双11大促、12306春运抢票

高并发的两种类型
- 读并发
- 写并发

大型网站系统架构演化实例

一、第一阶段：单体架构

小型网站最开始没有太多人访问，只需要一台服务器就绰绰有余，这时网站的应用程序、数据库、文件等所有资源都在一台服务器上。



大型网站系统架构演化实例

二、第二阶段: 垂直架构

将应用和数据分离，整个网站使用 3 台服务器的：应用服务器、文件服务器和数据库服务器。

✓ 应用服务器处理大量的业务逻辑，因此需要更快更强大的处理器速度。

✓ 数据库服务器需要快速磁盘检索和数据缓存，因此需要更快的磁盘和更大的内存。

✓ 文件服务器需要存储大量用户上传的文件，因此需要更大容量的硬盘。

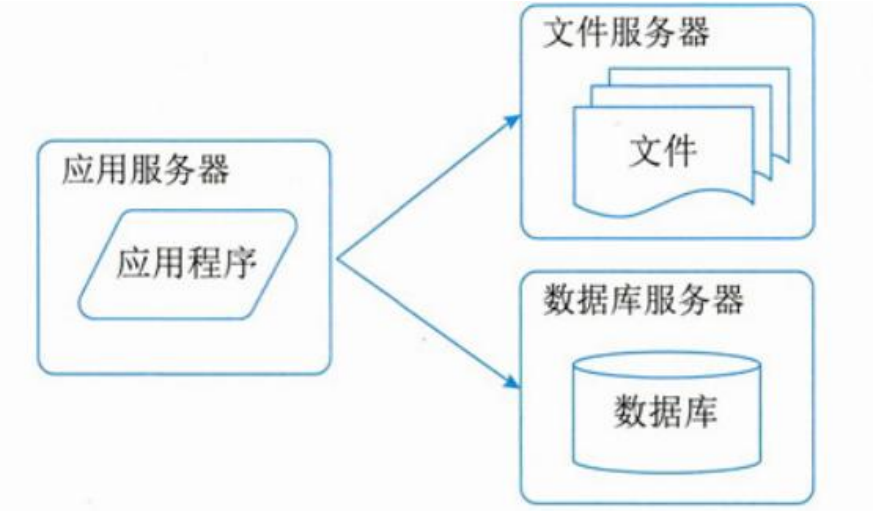


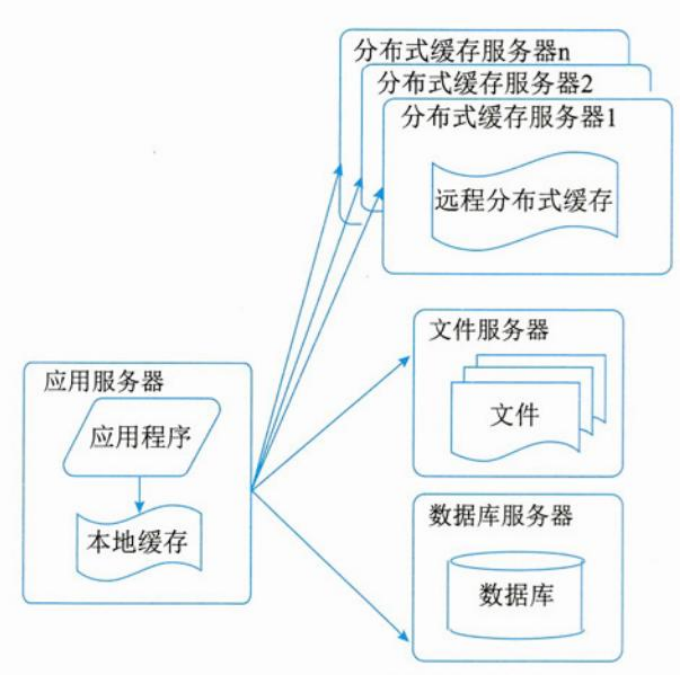
图 10-11　第二阶段网站架构

大型网站系统架构演化实例

三、第三阶段：使用缓存改善网站性能

把经常访问的小部分数据缓存在内存中，可以减少数据库的访问压力，提高整个网站的数据访问速度，改善数据库的写入性能。网站使用的缓存可分为两种：

- ✓ 缓存在应用服务器上的本地缓存
- ✓ 缓存在专门的分布式缓存服务器上的远程缓存使用缓存后，数据访问压力得到有效缓解，但是单一应用服务器能够处理的请求连接有限，在网站访问高峰期，应用服务器成为整个网站的瓶颈。



大型网站系统架构演化实例

◆缓存技术（★★）

- ✓ MemCache: Memcache是一个高性能的分布式的内存对象缓存系统，用于动态Web应用以减轻数据库负载。Memcache通过在内存里维护一个统一的巨大的hash表，它能够用来存储各种格式的数据，包括图像、视频、文件以及数据库检索的结果等。
- ✓ Redis: Redis是一个开源的使用ANSI C语言编写、支持网络、可基于内存亦可持久化的日志型、Key-Value数据库，并提供多种语言的API。
- ✓ Squid: Squid是一个高性能的代理缓存服务器，Squid支持FTP, gopher、HTTPS和HTTP协议。和一般的代理缓存软件不同，Squid用一个单独的、非模块化的、I/O驱动的进程来处理所有的客户端请求。

◆数据库与缓存数据可能不一致问题。（★★★）

解决思路：

读思路：1、先根据key从缓存中读取 2、如果缓存没有，则根据key从数据库中查找。

3、读取到“值”之后，更新缓存。

写思路：如果是写操作：那么先根据key值写数据库，然后根据key更新缓存（或删除缓存）

大型网站系统架构演化实例

◆比较Memcache和Redis二者区别（★★★）

比较项	Memcache	Redis
数据类型	简单Key/Value结构	不仅仅支持简单的k/v类型的数据，同时还提供list，set，hash等数据结构的存储
持久性	不支持	支持（所以一旦服务器挂掉，Redis数据丢失还可以恢复）
分布式存储	Memcache服务器需要通过hash一致化来支撑主从结构（从的服务器内容各不相同）	多种方式，主从、sentinel、cluster等
多线程支持	支持	redis把任务封闭在一个线程中，不支持多线程（5.0版本后就支持了）
内存管理	使用stab Allcation进行内存管理，按照既定的内存，将内存切割成特定的长度来存储相应的数据	无
事务支持	弱支持，只能保证事务中的每个操作连续执行	有限支持
数据容灾	不支持，不能做数据恢复	支持，可以在灾难发生时，恢复数据。

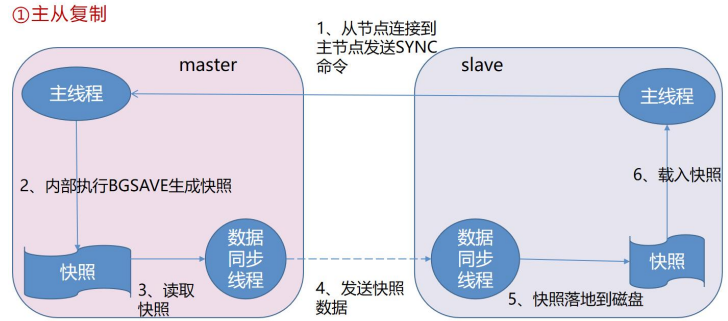
大型网站系统架构演化实例

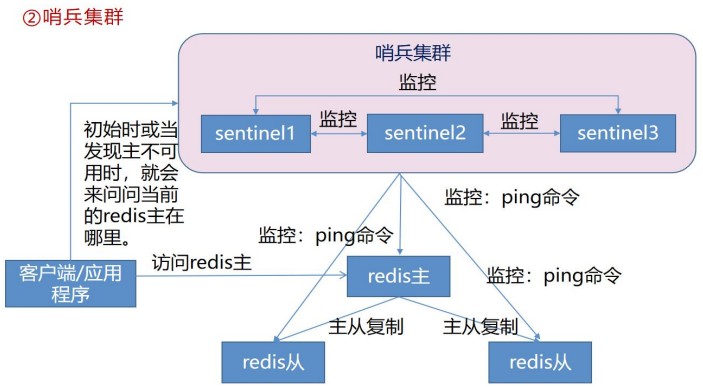
◆Redis内存淘汰机制（★★）

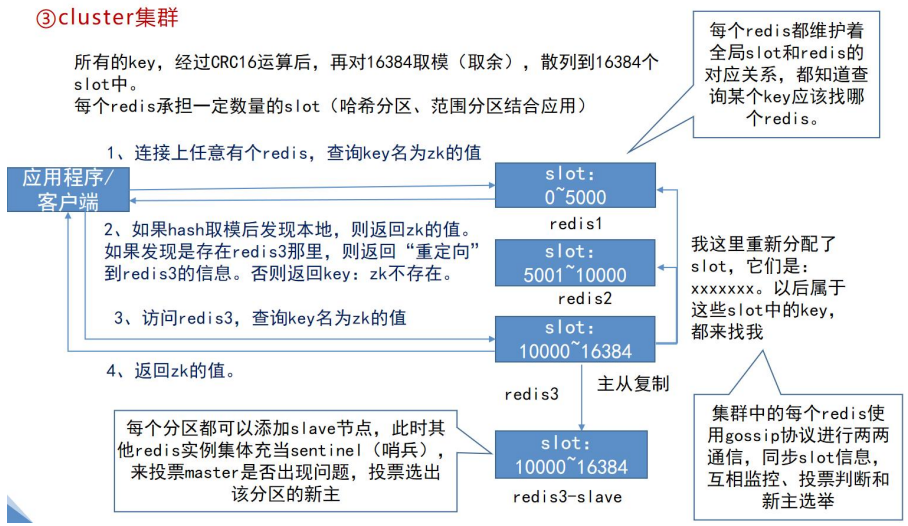
noeviction 不淘汰	禁止驱逐数据，内存不足报错。
volatile-lru 最近最少使用	从已设置过期时间的key中，移出最近最少使用的key进行淘汰
volatile-lfu 最不经常使用	从已设置过期时间的key中选择最不经常使用的进行淘汰。
volatile-random 随机淘汰算法	从已设置过期时间的key中随机选择key淘汰。
volatile-ttl 生存时间淘汰	从已设置过期时间的key中，移出将要过期的ttl值小的key。
全键空间allkeys-lru	从所有key中选择最近最少使用的进行淘汰
全键空间allkeys-lfu	从所有key中选择最不经常使用的进行淘汰。
全键空间 allkeys-random	从所有key中随机选择key进行淘汰

大型网站系统架构演化实例

Redis分布式存储方案	要点（★★）
主从模式	一主多从，故障时手动切换。
哨兵模式	有哨兵的一主多从，主节点故障自动选择新的主节点。
集群模式	分节点对等集群，分slots，不同slots的信息存储到不同节点







大型网站系统架构演化实例

Redis集群切片方式	特点（基于集群模式）（★★）
客户端分片	在客户端通过Key的hash值对应到不同的服务器。将分片工作放在业务程序端。不依赖于第三方分布式中间件，实现方法和代码可掌控，对开发人员要求高。
中间件分片	在应用软件和Redis中间由中间件实现服务到后台redis节点的路由分派。 将分片工作交给专门的代理程序来做，运维方便。代表：Twemproxy，Codis
客户端服务端协作分片	Redis Cluster将所有Key映射到16384个Slot中，集群中每个Redis实例负责一部分，业务程序通过集成的Redis Cluster客户端进行操作。 redisCluter模式，客户端采用一致性哈希，服务端提供错误节点的重定向服务slot上。不同的slot对应到不同服务器。

大型网站系统架构演化实例

Redis数据分片方案	说明（★★）
范围分片	按数据范围值（0-60，61-80，81-100）
哈希分片	通过key进行hash运算分片（类似于取余），余数相同放在一个实例。
一致性哈希分片	利用扩展结点，可以有效解决重新分配结点带来的无法命中问题。 哈希分片的改进（有效解决hash无法命中问题）

大型网站系统架构演化实例

◆Redis数据类型（★★）

Redis数据类型	特点	实例
String（字符串）	存放二进制如缓存	缓存、计数、共享session
Hash（字典）	无序字典，数组+链表，适合存储对象 key对应一个HashMap，针对一组数据。	存储、读取、修改用户属性
List（列表）	Linked List：双向链表，有序，增删快，查询慢。 Array List：数组方式，有序，增删慢，查询快。	消息队列，文章列表 记录前N个最新登录的用户ID列表
Set（集合）	键值对无序，唯一。支持并、交、差集操作	独立IP，共同爱好，标签。
Sorted Set（有序集合）	键值对，有序，自带按权重排序效果	排行榜

大型网站系统架构演化实例

◆Redis提供了两种持久化存储的机制，分别是RDB (Redis DataBase)持久化方式和AOF (Append Only File)持久化方式。一旦服务器宕机，内存中的数据将全部丢失。我们很容易想到的一个解决方案是，从后端数据库恢复这些数据，但这种方式存在两个问题：一是需要频繁访问数据库，会给数据库带来巨大的压力；二是这些数据是从慢速数据库中读取出来的，性能肯定比不上从 Redis 中读取，导致使用这些数据的应用程序响应变慢。所以，对 Redis 来说，实现数据的持久化，避免从后端数据库中进行恢复，是至关重要的。（★★）

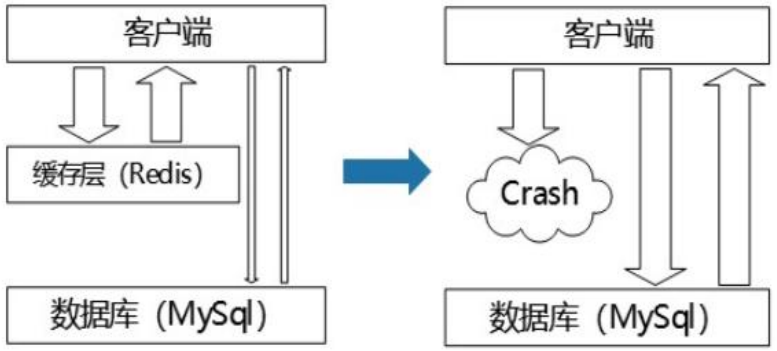
两种持久化机制	RDB内存快照 （RedisDataBase）	AOF日志 （Append Only Flie）
说明	把当前内存中的数据集快照写入磁盘（数据库中所有键值对数据）。恢复时是将快照文件直接读到内存里。	通过持续不断地保存Redis服务器所执行的更新命令来记录数据库状态，类似mysql的binlog。恢复数据时需要从头开始回放更新命令。
磁盘刷新频率	低	高
文件大小	小	大
数据恢复效率	高	低
数据安全	低	高

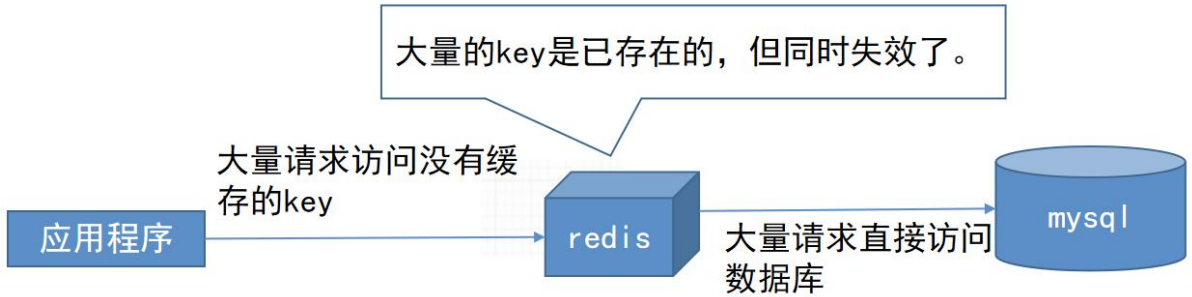
大型网站系统架构演化实例

◆Redis常见问题（★★）

1、缓存雪崩

问题：大部分缓存失效----数据库崩溃





解决方法：

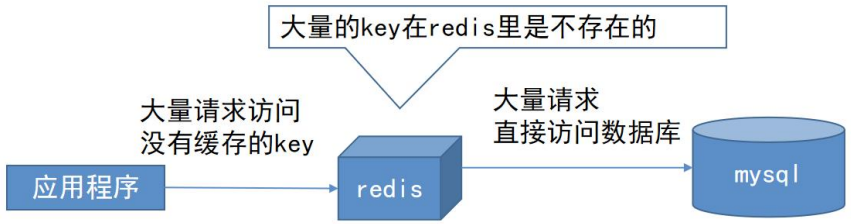
1、使用锁或者队列：保证不会有大量的线程对数据库一次性进行读写，从而避免失效时大量的并发请求落到最底层存储系统上。

2、为key设置不同的缓存失效时间：在固定的一个缓存时间的基础上加上随机一个时间作为缓存失效时间。

3、二级缓存：设置一个有时间限制的缓存和一个无时间限制的缓存。避免大规模访问数据库。

大型网站系统架构演化实例

2、缓存穿透



问题描述--查询无数据返回---直接查数据库。
解决方案：
1、如果查询结果为空，直接设置一个默认值存放到缓存，这样第二次到缓冲中获取就有值了。设置 一个不超过5分钟的过期时间，以便正常更新缓存。
2、设置布隆过滤器，将所有可能存在的数据哈希到一个足够大的bitmap中，一个一定不 存在 的数据会被这个bitmap拦截掉，从而避免了对底层存储系统的查询压力。

布隆过滤器：用于快速识别一个元素不在一个集合中 。通过 一个长二进制向量和一系列随机映射函数来记录与识别某个数据是否在一个集合中。
优点：（1）占有内存小（2）查询效率高（3）不需要存储元素本身，在某些对保密要求比较严格的场合有很大优势。
缺点：有一定的误判率 ，即存在假阳性，不能准确判断元素是否在集合中 。（2）一般情况下不能从布隆过滤器中删除元素。（3）不能获取元素本身。

大型网站系统架构演化实例

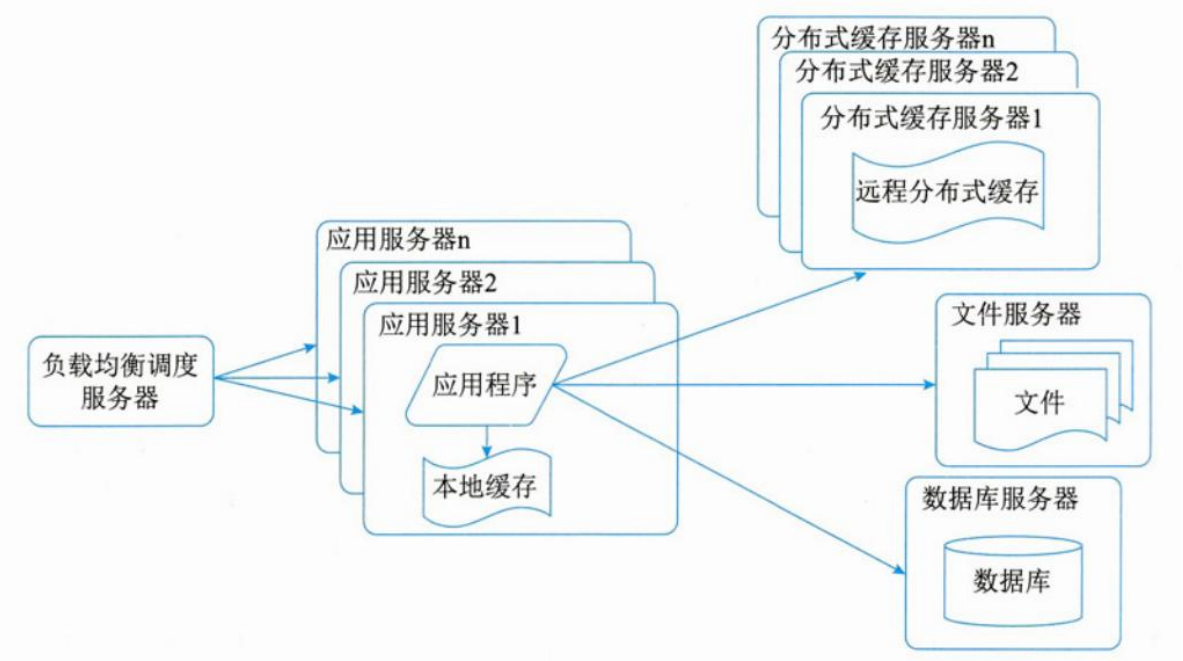
3、缓存预热
系统上线后，将相关需要缓存 数据直接加到缓存系统中。
解决方案：（1）直接写个缓存刷新页面，上线时手工操作 。（2）数据量不大时，可以在项目启动的时候自动进行加载。（3）定时刷新缓存 。

4、缓存更新
除Redis系统自带的缓存失效策略，常见使用以下两种：（1）定时清理过期的缓存。（2）当有用户请求过来时，再判断这个请求所用到的缓存是否过期，过期的话就去底层系统得到新数据并更新缓存 。

5、缓存降级
降级的目的是保证核心服务可用，即使是有损的，但是有些服务是无法降级的（如电商的购物流程等）；在进行降级之前要对系统进行梳理，从而梳理出哪些必须保护，哪些可降级。

大型网站系统架构演化实例

四、第四阶段：使用服务集群改善网站并发处理能力通过增加服务器的方式改善负载压力、提高系统性能，从而实现系统的可伸缩性。
应用服务器集群是网站可伸缩架构设计中较为简单成熟的一种。



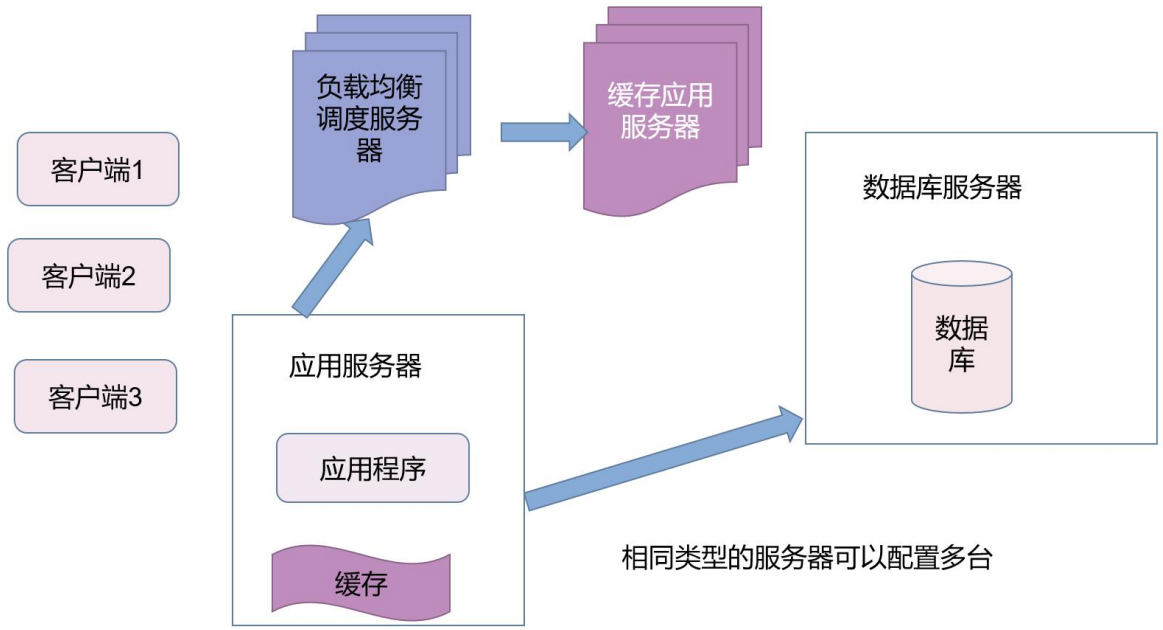
大型网站系统架构演化实例

系统演变到这个阶段会产生两个问题：
1、用户的请求由谁来转发到具体的应用服务器？应该发给哪个服务器进行响应请求？
2、用户如果每次访问的 服务器不一样，那么如何维护session的一致性？

所以带来两个问题：
1、负载均衡
2、有状态和无状态的问题

大型网站系统架构演化实例

负载均衡组件：keepalived。健康检查和失败切换是keepalived的两大核心功能。所谓的健康检查，就是采用tcp三次握手，icmp请求，http请求，udp echo请求等方式对负载均衡器后面的实际的服务器(通常是承载真实业务的服务器)进行主要是应用于配置了主备模式的负载均衡器，利用VRRP维持主备负载均衡器的心跳，当主负载均衡器出现问题时，由备负载均衡器承载对应的业务，从而在最大限度上减少流量损失，并提供服务的稳定性。Nginx、lvs、zookeeper等。

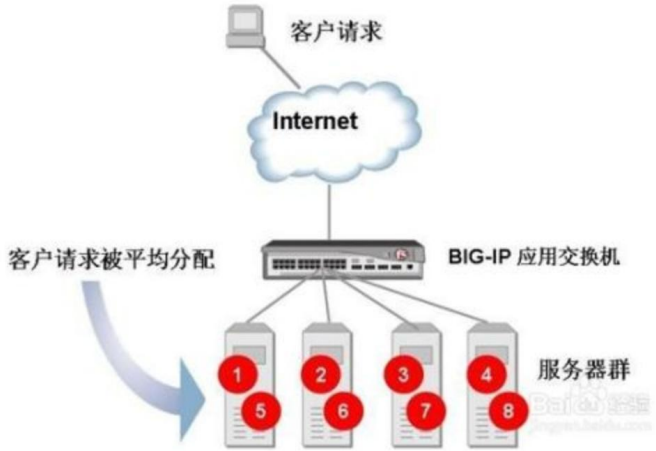


大型网站系统架构演化实例

负载均衡技术	原理（★★）	优点	缺点
基于http的 负载均衡	根据用户的http请求计算出一个真实的web服务器地址，并将该web服务器地址写入http重定向响应中返回给浏览器，由浏览器重新进行访问。	实现比较简单 应用层	浏览器需要两次请求服务器才能完成一次访问，性能较差。
反向代理 负载均衡	反向代理是作用在服务器端的，是一个虚拟ip(VIP)。对于用户的一个请求，会转发到多个后端处理器中的一台来处理该具体请求。在用户的请求到达反向代理服务器时（到网站机房），由反向代理服务器根据算法转发到具体的服务器，常用的是apache、nginx都可以充当反向代理服务器。	部署简单，处于http协议层面。 应用层	用了反向代理服务器后，web 服务器地址不能直接暴露在外，因此web服务器不需要使用外部IP地址，而反向代理服务作为沟通桥梁就需要配置双网卡、外部内部两套IP地址。
基于DNS的 负载均衡	在用户请求DNS服务器，获取域名对应的IP地址时，DNS服务器直接给出负载均衡后的服务器IP.将负载均衡的工作交给了DNS，省却了网站管理维护负载均衡服务器的麻烦，同时许多DNS还支持基于地理位置的域名解析，将域名解析成距离用户地理最近的一个服务器地址。	加快访问速度，改善性能 传输层	1、目前的DNS解析是多级解析，每一级DNS都可能化缓存记录，当某一服务器下线后，该服务器对应的DNS记录可能仍然存在，不能及时通知DNS导致分配到该服务器的用户访问失败。负载均衡效果并不是太好。 2、同时DNS服务器会成为瓶颈。
基于NAT（网络地址转换）的负载均衡	将一个外部IP地址映射为多个内部IP地址，对每次连接请求动态地转换为一个内部节点的地址，将外部连接请求引到转换得到地址的那个节点上，从而达到负载均衡的目的。	在响应请求时速度较反向服务器负载均衡要快 传输层	当请求数据较大（大型视频或文件）时，速度较慢。

大型网站系统架构演化实例

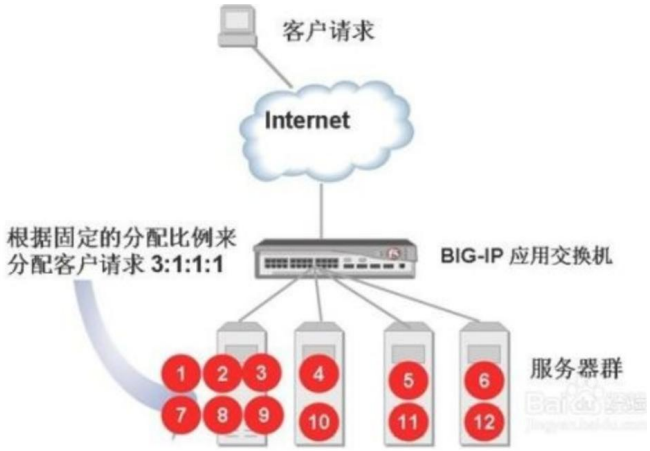
◆负载均衡的算法（★★★）分为按照是否考虑后端服务器负载可以分为静态负载均衡算法和动态负载均衡算法。
◆静态均衡算法：（不考虑动态负载 ）
①轮询法：
将请求按顺序轮流地分配到每个节点上，不关心每个节点实际的连接数和当前的系统负载。
优点：简单高效，易于水平扩展，每个节点满足字面意义上的均衡；
缺点：没有考虑机器的性能问题，根据木桶最短木板理论，集群性能瓶颈更多的会受性能差的服务器影响。
②随机法：
将请求随机分配到各个节点。由概率统计理论得知，随着客户端调用服务端的次数增多，其实际效果越来越接近于平均分配，也就是轮询的结果。优缺点和轮询相似。



大型网站系统架构演化实例

③源地址（目标地址）哈希法：
源地址哈希的思想是根据客户端的IP地址，通过哈希函数计算得到一个数值，用该数值对服务器节点数进行取模，得到的结果便是要访问节点序号。采用源地址哈希法进行负载均衡，同一IP地址的客户端，当后端服务器列表不变时，它每次都会落到到同一台服务器进行访问。目标地址哈希算法：根据请求目标IP做散列找到对应结点。
④加权轮询法：
不同的后端服务器可能机器的配置和当前系统的负载并不相同，因此它们的抗压能力也不相同。给配置高、负载低的机器配置更高的权重，让其处理更多的请；而配置低、负载高的机器，给其分配较低的权重，降低其系统负载，加权轮询能很好地处理这一问题，并将请求顺序且按照权重分配到后端。
加权轮询算法要生成一个服务器序列，该序列中包含n个服务器。n是所有服务器的权重之和。在该序列中，每个服务器的出现的次数，等于其权重值。并且，生成的序列中，服务器的分布应该尽可能的均匀。比如序列{a, a, a, a, a, b, c}中，前五个请求都会分配给服务器a，这就是一种不均匀的分配方法，更好的序列应该是：{a, a, b, a, c, a, a}。

大型网站系统架构演化实例



⑤加权随机法：
与加权轮询法一样，加权随机法也根据后端机器的配置，系统的负载分配不同的权重。不同的是，它是按照权重随机请求后端服务器，而非顺序。
⑥键值范围法：
根据键的范围进行负债，比如0到10万的用户请求走第一个节点服务器，10万到20万的用户请求走第二个节点服务器……以此类推。
优点：容易水平扩展，随着用户量增加，可以增加节点而不影响旧数据；
缺点：容易负债不均衡，比如新注册的用户活跃度高，旧用户活跃度低，那么压力就全在新增的服务节点上，旧服务节点性能浪费。而且也容易单点故障，无法满足高可用。

大型网站系统架构演化实例

◆动态负载均衡算法：（考虑动态负载）

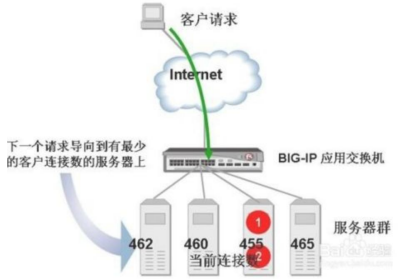
（1）最小连接数法：

根据每个节点当前的连接情况，动态地选取其中当前积压连接数最少的一个节点处理当前请求，尽可能地提高后端服务的利用效率，将请求合理地分流到每一台服务器。俗称闲的人不能闲着，大家一起动起来。

优点：动态，根据节点状况实时变化；

缺点：提高了复杂度，每次连接断开需要进行计数；

（2）加权最小连接算法：考虑结点处理能力不同，按最小连接数分配。



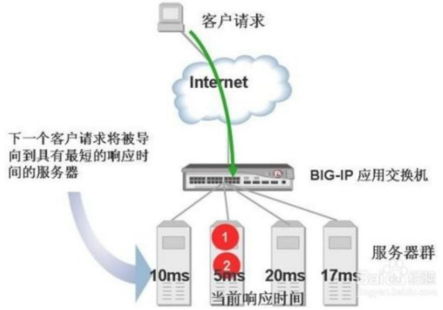
（3）最快响应速度法：

根据请求的响应时间，来动态调整每个节点的权重，将响应速度快的服务节点分配更多的请求，响应速度慢的服务节点分配更少的请求，俗称能者多劳，扶贫救弱。

优点：动态，实时变化，控制的粒度更细，跟灵敏；

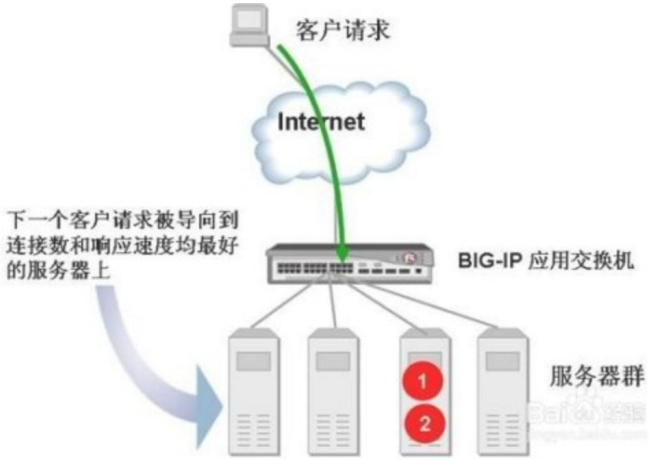
缺点：复杂度更高，每次需要计算请求的响应速度；

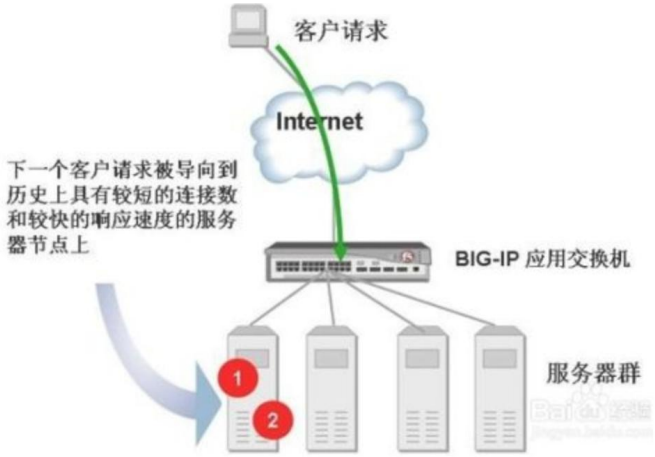
实现：可以根据响应时间进行打分，计算权重。



大型网站系统架构演化实例

（4）观察模式法：
观察者模式是综合了最小连接数和最快响应度，同时考量这两个指标数，进行一个权重的分配。
（5）加权百分比法：考虑了节点的利用率、硬盘速率、进程个数等，使用利用率来表现剩余处理能力。





大型网站系统架构演化实例

◆无服务状态：对单次请求的处理，不依赖其他请求，也就是说，处理一次请求所需的全部信息，要么都包含在这个请求里，要么可以从外部获取到，服务器不存储任何信息。
◆有状态：它会自身保存一些数据，先后的请求是有关联的。
（a）Identification Bean(身份认证构件）
（b）ResPublish Bean(资源发布构件)
（c）ResRetrieval Bean(资源检索构件)
（d）OnlineEdit Bean(在线编辑构件)
（e）Statistics Bean(统计分析构件)

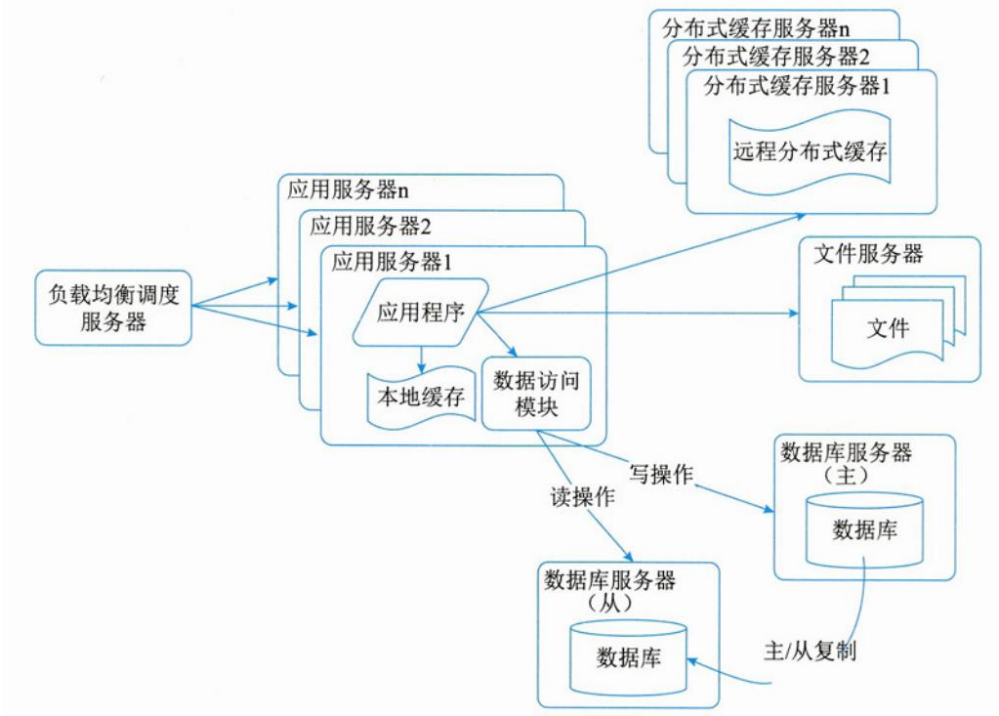
有状态构件包含：(a)、(b) 、(d)
无状态构件包含：(c)、(e)

◆集群的情况下，用户的请求由谁去分发到具体的应用服务器？当每次访问的时候如果维护session的一致性？
解决方法：
1、客户端：cookie携带session
2、集群服务器之间同步session
3、将session存入redis。

大型网站系统架构演化实例

五、第五阶段: 数据库读写分离

在网站的用户达到一定规模后，数据库因为负载压力过高而成为网站的瓶颈。目前的主流数据库提供主从热备功能，通过配置两台数据库主从关系，可以将一台数据库服务器的数据更新同步另一台服务器上。网站利用数据库的这一功能，实现数据库读写分离，从而改善数据库负载压力。应用服务器在写数据的时候，访问主数据库，主数据库通过主从复制(同步)机制将数据更新同步到从数据库，这样当应用服务器读数据的时候，就可以通过从数据库获得数据。



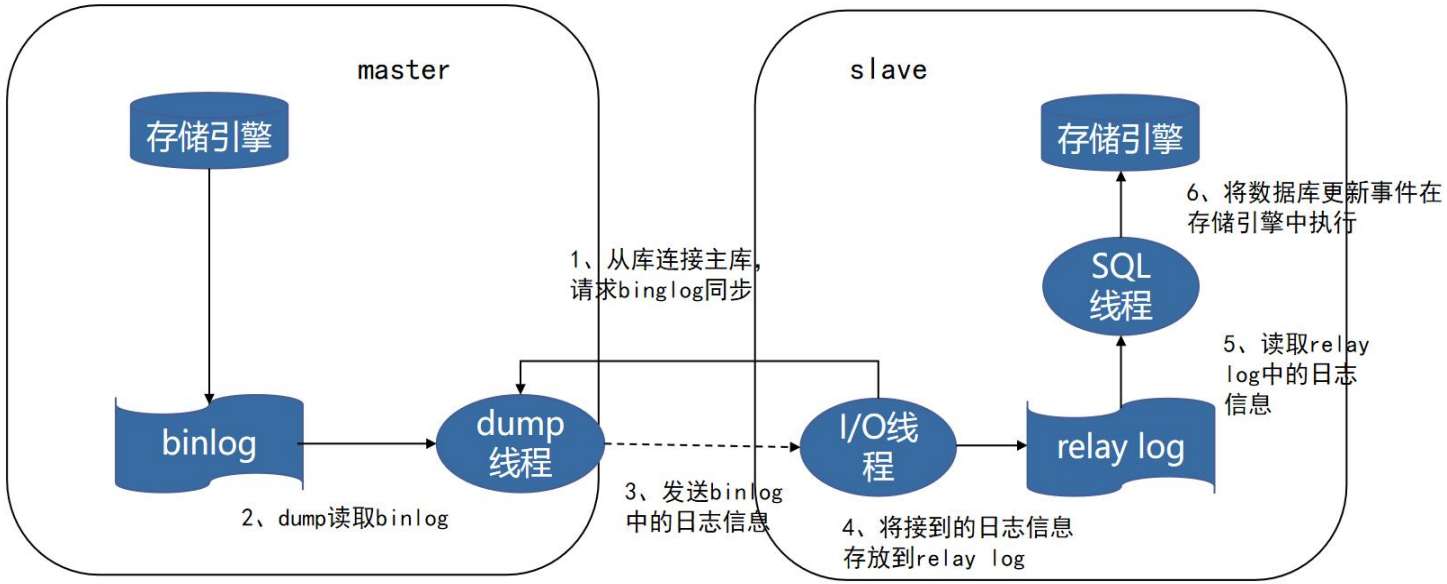
大型网站系统架构演化实例

◆主从复制：设置物理上不同的主/从服务器，一般而言一主多从，也可以多主多从。引入主从复制机制所带来的好处：

① 避免数据库单点故障、提高可用性：主服务器实时、异步复制数据到从服务器，当主数据库宕机时，可在从数据库中选择一个升级为主服务器，从而防止数据库单点故障。

② 提高查询效率：根据系统数据库访问特点，可以使用主数据库进行数据的插入、删除及更新等写操作，而从数据库则专门用来进行数据查询操作，从而将查询操作分担到不同的从服务器以提高数据库访问效率。

主从数据库之间通过 binary log（二进制日志）进行数据的同步。具体过程如下：



大型网站系统架构演化实例

◆binlog有三种模式：

①基于SQL语句的复制，每一条更新的语句（insert、update、delete）都会记录在binlog中，进而同步到从库的relaylog中，被从库的SQL线程取出来，回放执行。

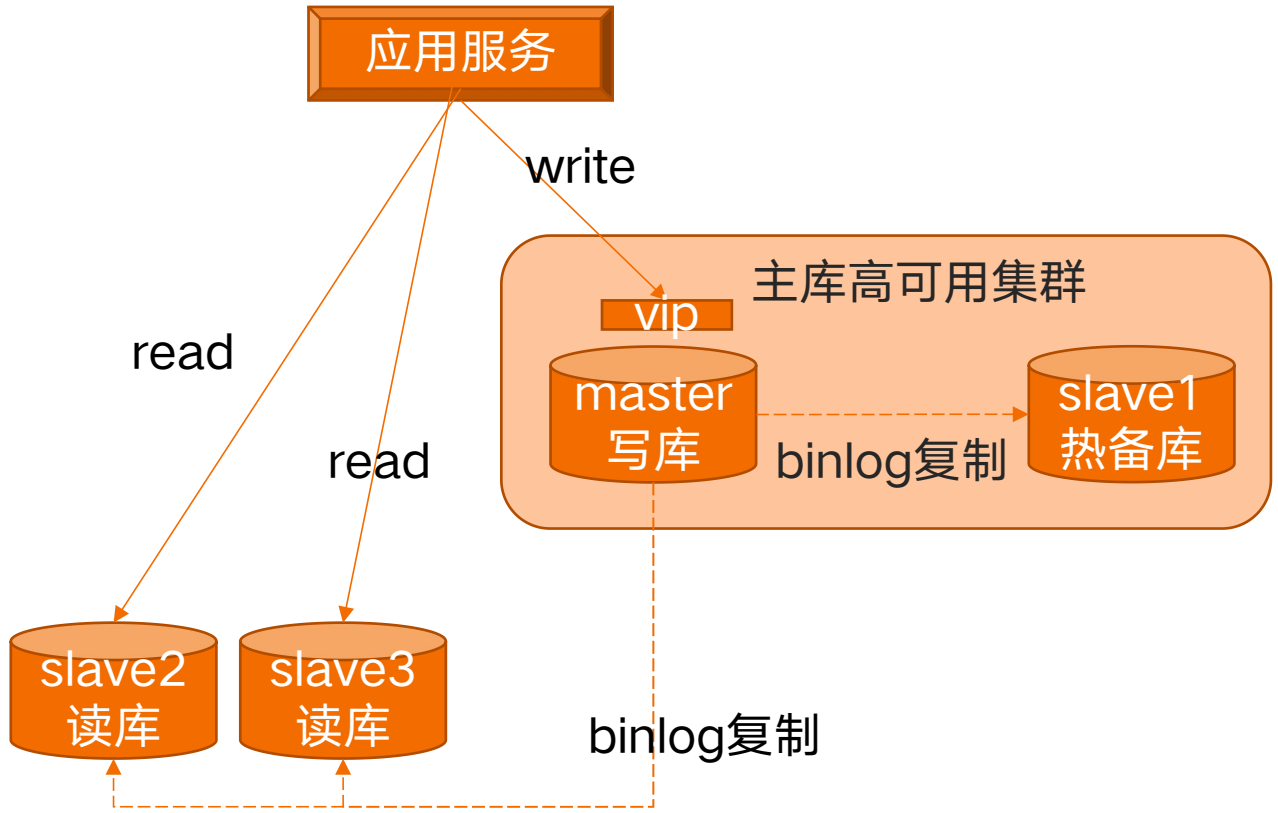
该模式的优点是binlog的日志量可能会比较少，比如一个涉及行数为1000行的update语句：同步这一个语句，就同步了1000行的数据。缺点是：同步的SQL语句里如果含有绑定本地变量的函数、关键字时，可能造成主从不一致的情况。比如SQL语句中有time函数，如果主从数据库的服务器时间不是精确相等，就会造成结果不一致。

②基于行的复制，不记录SQL语句，只记录了哪个记录更新前和更新后的数据，可以保证主从之间数据绝对相同。缺点是：1条SQL更新1000行的数据无法再偷懒，必须原原本本同步1000行的数据量。

③混合复制：以上两种模式的混合，选取两者的优点。对于有绑定本地特性、评估可能造成主从不一致的SQL语句，则自动选用ROW，其他的选择STATEMENT。

大型网站系统架构演化实例

◆读写分离：设置不同的主/从数据库分别负责不同的操作。让主数据库负责数据的写操作，从数据库负责数据的读操作。通过角色分担的策略，分别提升读写性能，有效减少数据并发操作的延迟。（★★★）



大型网站系统架构演化实例

◆mysql主从同步的同步方案（★★★）

数据同步方案有：1、触发器（当同步要求比较严格的时候）；2、一致性要求不严格（定时器）；3、数据库插件

◆mysql主从同步的同步模式：

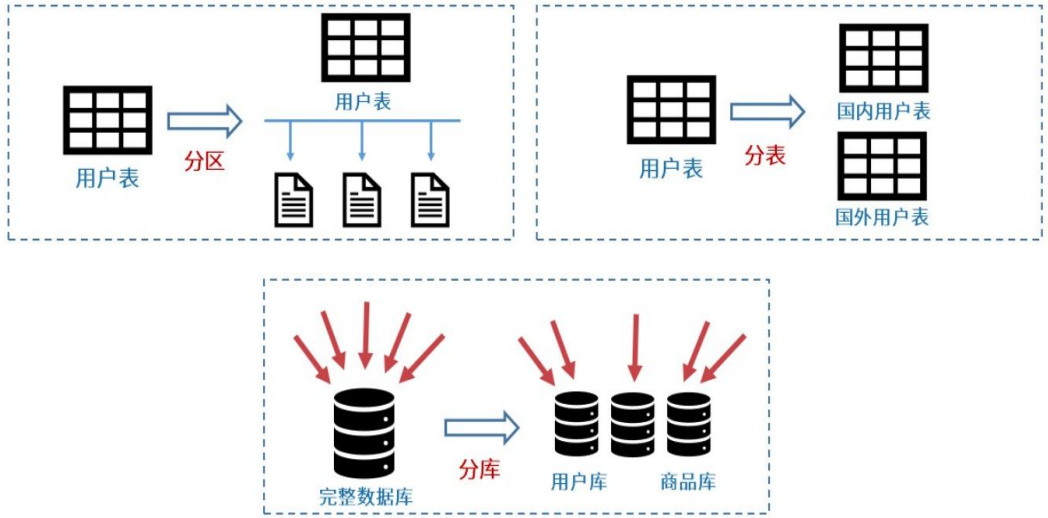
① 全同步：当主库执行完一个事务，所有的从库都执行了该事务才返回给客户端。优点是：可用性达到最大，只要不是所有数据库都发生异常，就能保证数据不丢失。缺点是：因为需要等待所有从库执行完该事务才能返回，所以全同步复制的性能必然会收到严重的影响。

② 半同步：主库只需要等待至少一个从库节点从 Binlog 中收到该请求，并保存到 Relay Log 文件即可，主库不需要等待所有从库给主库反馈。优点是：只收到的一个反馈，而不是全部从库完成提交的反馈，节省了很多时间，同时能保证主库发生故障后，至少能从余下的1个从库上找回全部数据。缺点是：降低了可用性，但权衡了可用性和性能，做出取舍。

③ 异步：主库在执行完客户端提交的事务后会立即将结果返给给客户端，并不关心从库是否已经接收并处理。缺点是：主库如果发生问题，此时主库已经提交的事务可能并没有同步到从库上。可能导致数据丢失或不完整。优点是：处理请求的延迟降到了最低。

大型网站系统架构演化实例

◆数据库分区分表分库（★★）



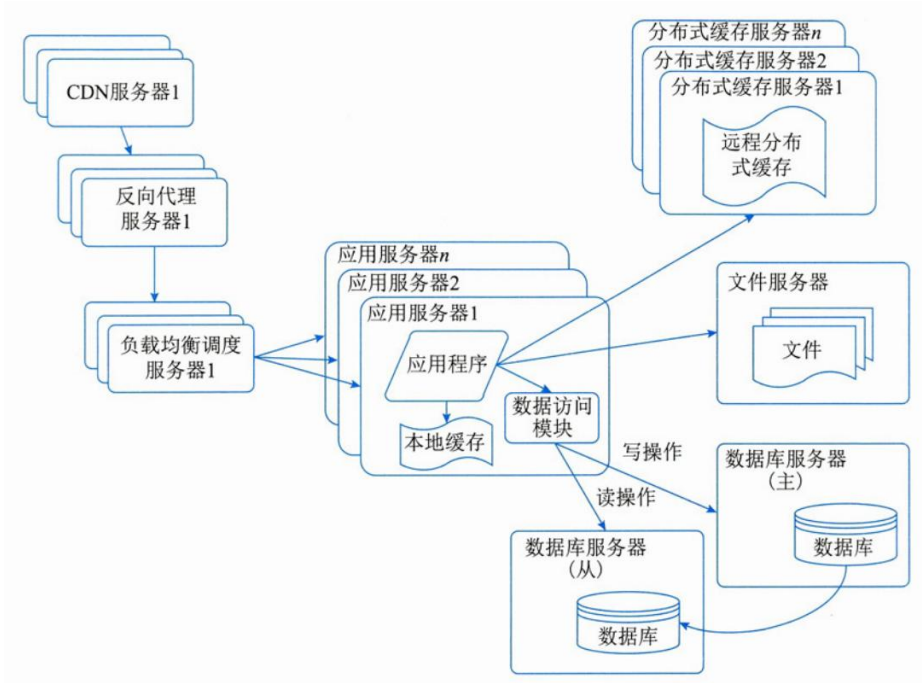
分区和分表的区别：
分区---逻辑上还是一张表。
分表：逻辑上已经是多张表。

分区策略	说明	举例
范围分区	范围值	0-60，61-80等
散列分区	key进行hash运算	余数相同一个分区
列表分区	根据某个具体字段值	如北京、上海、广州

大型网站系统架构演化实例

六、第六阶段: 使用反向代理和 CDN 加速网站响应随着网站业务不断发展，用户规模越来越大，由于区域的差别使得网络环境异常复杂，不同地区的用户访问网站时，速度差别也极大。为了更好的用户体验，网站需要加速网站访问速度。主要手段有使用CDN 和反向代理。CDN和反向代理的基本原理都是缓存。

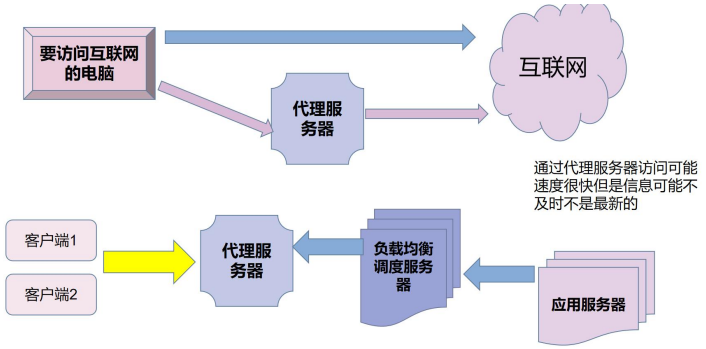
- CDN部署在网络提供商的机房，使用户在请求网站服务时，可以从距离自己最近的网络提供商机房获取数据。
- 反向代理部署在网站的中心机房，当用户请求到达中心机房后，首先访问的服务器是反向代理服务器，如果反向代理服务器中缓存着用户请求的资源，就将其直接返回给用户。

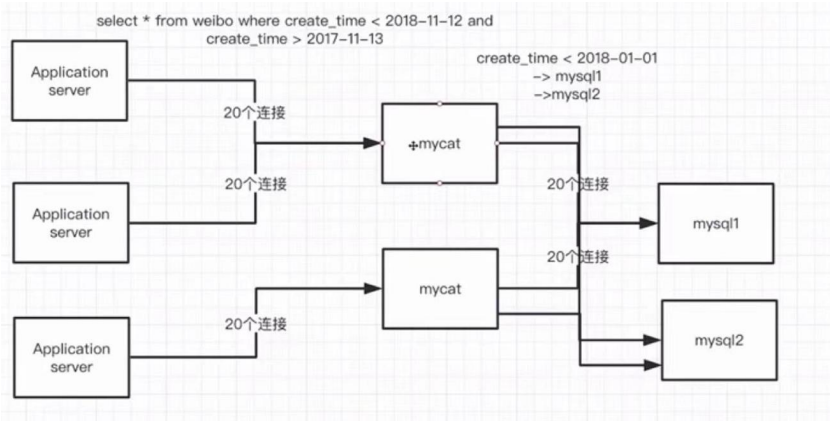


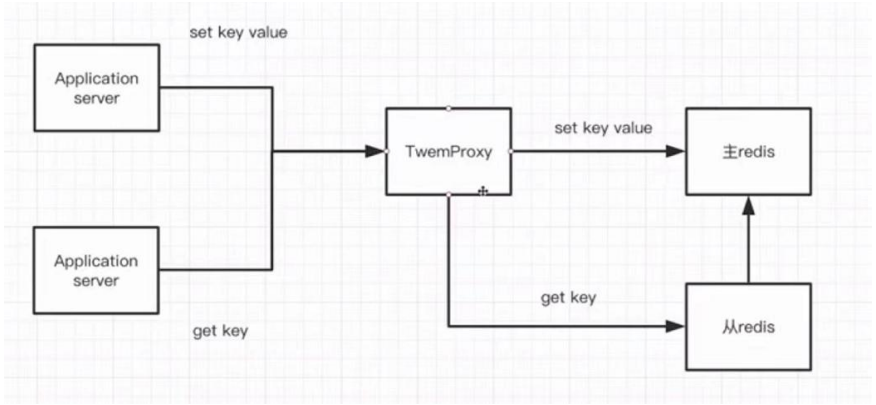
大型网站系统架构演化实例

代理一个或一群后端的被访问对象，使得调用端看似在直接访问后端对象一样，代理访问的代理服务器从而可以实现多种负载均衡、故障转移、缓存策略等个性化配置，同时又可以分散被代理的后端对象压力。

代理分为：后端应用服务器代理如Nginx反向代理
数据库服务器如Mycat代理
缓存服务器如Twemproxy代理



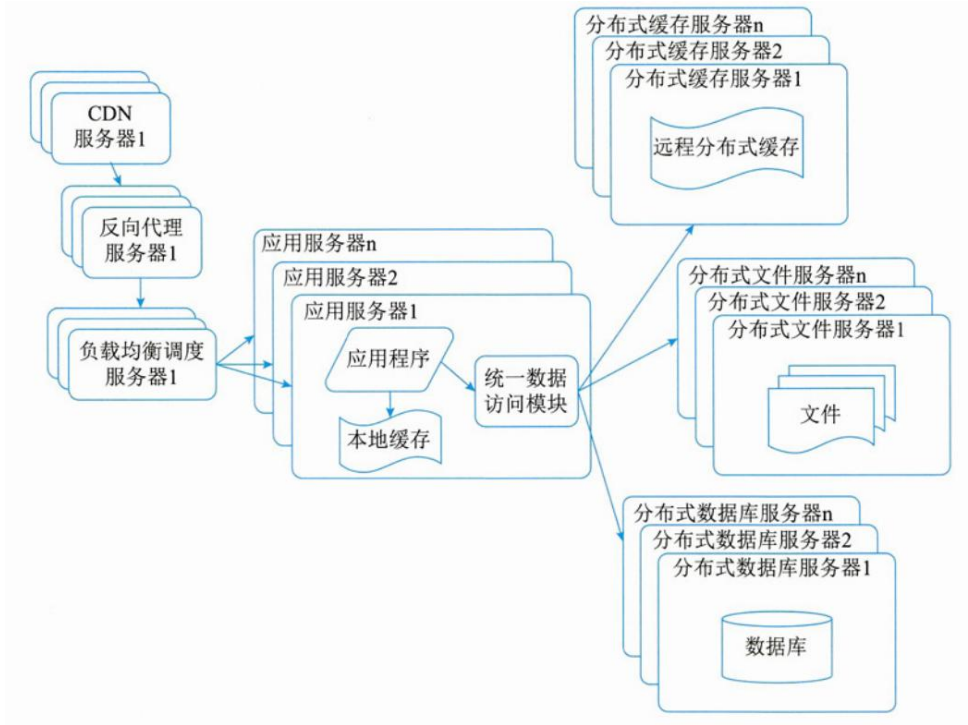




大型网站系统架构演化实例

七、第七阶段：使用分布式文件系统和分布式数据库系统

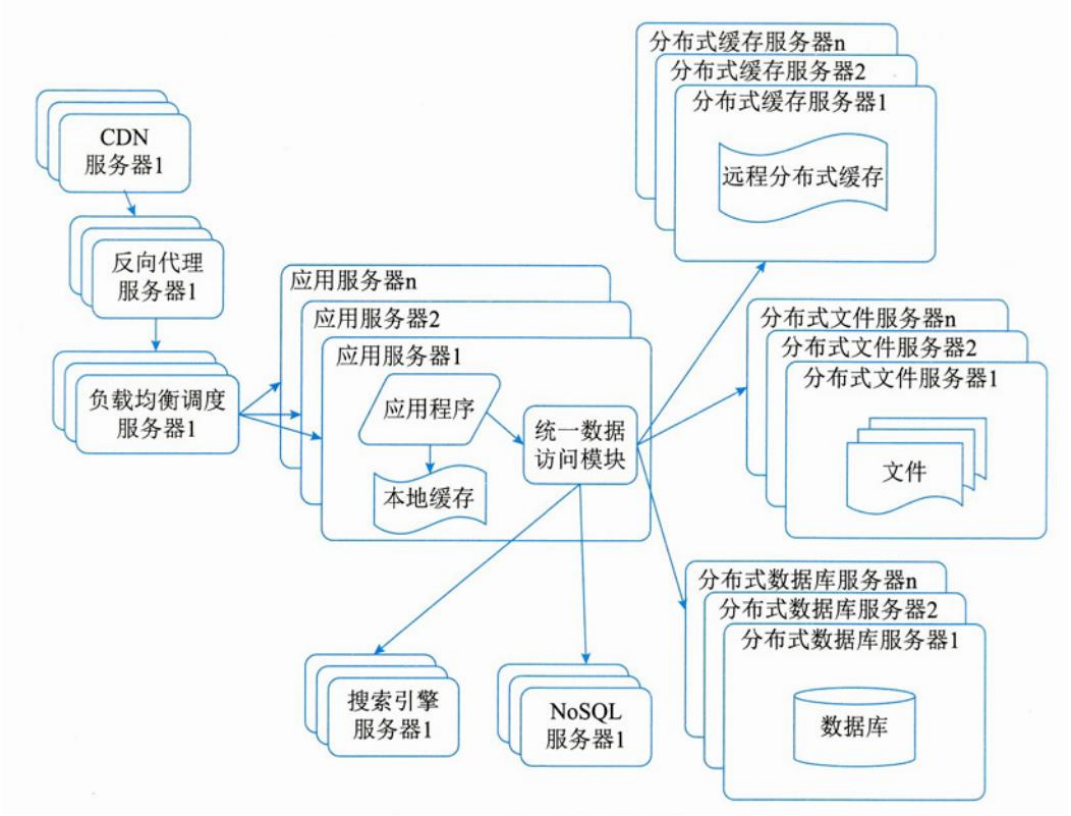
任何强大的单一服务器都满足不了大型网站持续增长的业务需求。数据库经过读写分离后，一台服务器拆分成两台服务器，但是随着网站业务的发展依然不能满足需求，这时需要使用分布式数据库。文件系统也一样，需要使用分布式文件系统。分布式数据库是网站数据库拆分最后手段，只有在单表数据规模非常庞大的时候才使用。不到不得已时，网站更常用的数据库拆分手段是业务分库，将不同业务的数据部署在不同的物理服务器上。



大型网站系统架构演化实例

八、第八阶段: 使用NoSQL和搜索引擎

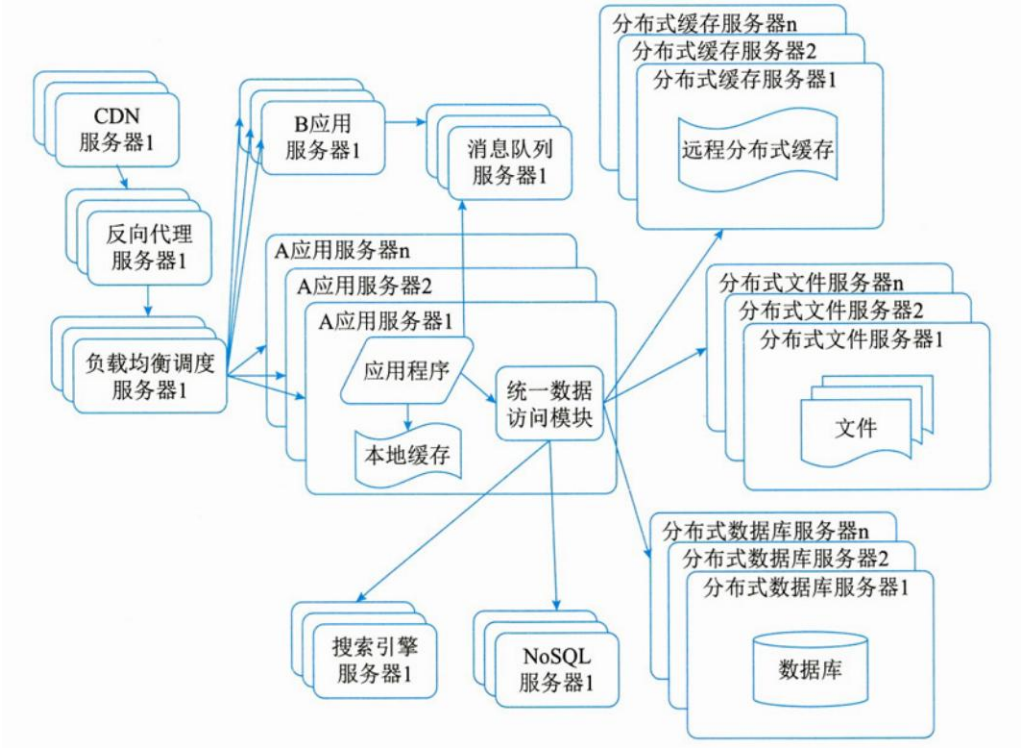
随着网站业务越来越复杂，对数据存储和检索的需求也越来越复杂，网站需要采用一些非关系数据库技术，如 NoSQL 和非数据库查询技术如搜索引擎。NoSQL 和搜索引擎都是源自互联网的技术手段，对可伸缩的分布式特性具有更好的支持。应用服务器则通过一个统一数据访问模块访问各种数据，减轻应用程序管理诸多数据源的麻烦。



大型网站系统架构演化实例

九、第九阶段: 业务拆分

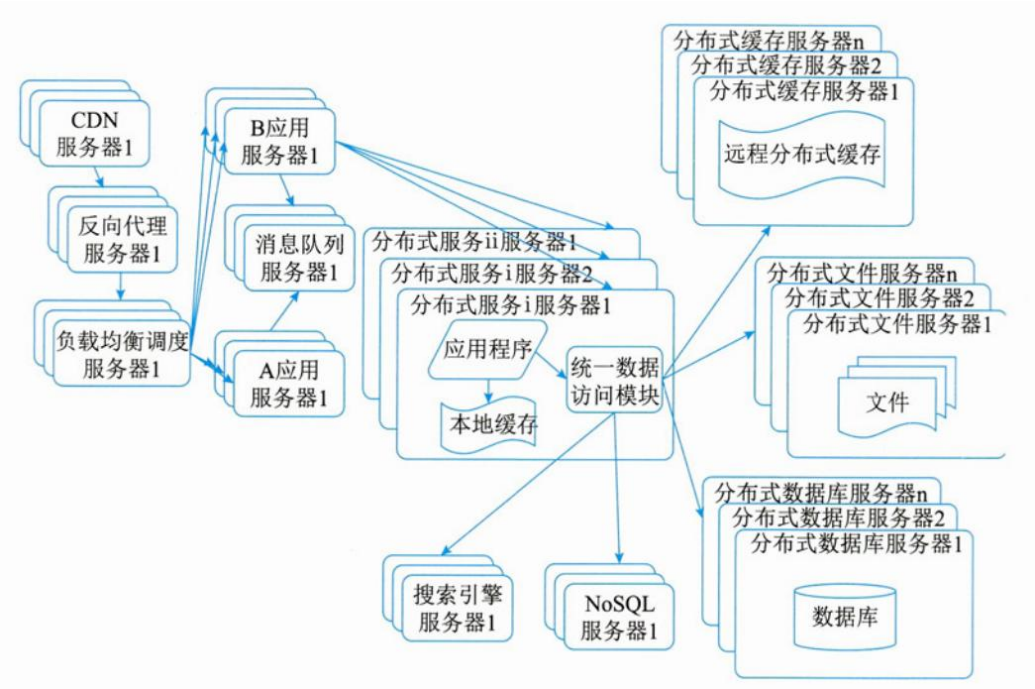
大型网站为了应对日益复杂的业务场景，通过使用分而治之的手段将整个网站业务分成不同的产品线。如电商网站都会将首页、商铺、订单、买家、卖家等拆分成不同的产品线，分归不同的业务团队负责。具体到技术上，也会根据产品线划分，将一个网站拆分成许多不同的应用，每个应用独立部署。应用之间可以通过一个超链接建立关系，也可以通过消息队列进行数据分发，当然最多的还是通过访问同一个数据存储系统来构成一个完整系统。



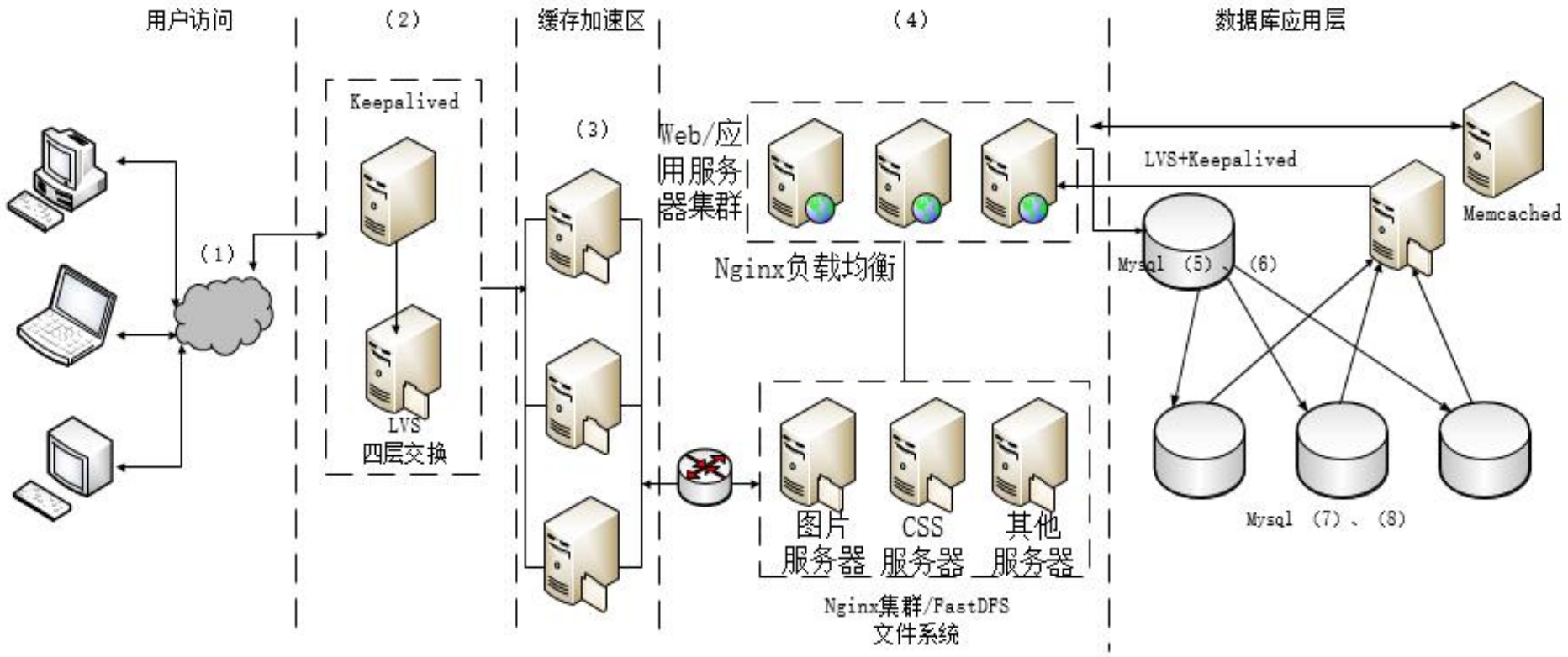
大型网站系统架构演化实例

十、第十阶段: 分布式服务

随着业务拆分越来越小，存储系统越来越庞大，应用系统的整体复杂度呈指数级增加，部署维护越来越困难。由于所有应用要和所有数据库系统连接，在数万台服务器规模的网站中，这些连接的数目是服务器规模的平方，导致数据库连接资源不足，拒绝服务。既然每一个应用系统都需要执行许多相同的业务操作，比如用户管理、商品管理等，那么可以将这些共用的业务提取出来，独立部署。由这些可复用的业务连接数据库，提供共用业务服务，而应用系统只需要管理用户界面，通过分布式服务调用共用业务服务完成具体业务操作。



大型网站系统架构演化实例



1、CDN内容分发　2、负载均衡层　3、缓存服务器集群　4、Web 应用层　56主数据库写操作　78从数据库读操作

目录

软件架构维护

软件架构是软件开发和维护过程中的一个重点制品，是软件需求和设计、实现之间的桥梁。
软件架构维护过程一般涉及架构知识管理、架构修改管理和架构版本管理等内容。（★★）

◆软件架构知识管理
软件架构知识管理是对架构设计中所隐含的决策来源进行文档化表示，进而在架构维护过程中帮助维护人员对架构的修改进行完善的考虑，并能够为其他软件架构的相关活动提供参考。
1.架构知识= 架构设计 + 架构设计决策。
2.架构知识管理侧重于软件开发和实现过程所涉及的架构静态演化，从架构文档等信息来源中捕捉架构知识，进而提供架构的质量属性及其设计依据以进行记录和评价。

◆软件架构修改管理
在软件架构修改管理中，一个主要的做法就是建立一个隔离区域，保障该区域中任何修改对其他部分的影响比较小，甚至没有影响。为此，需要明确修改规则、修改类型，以及可能的影响范围和副作用等。

◆软件架构版本管理
软件架构版本管理为软件架构演化的版本演化控制、使用和评价等提供了可靠的依据，并为架构演化量化度量奠定了基础。

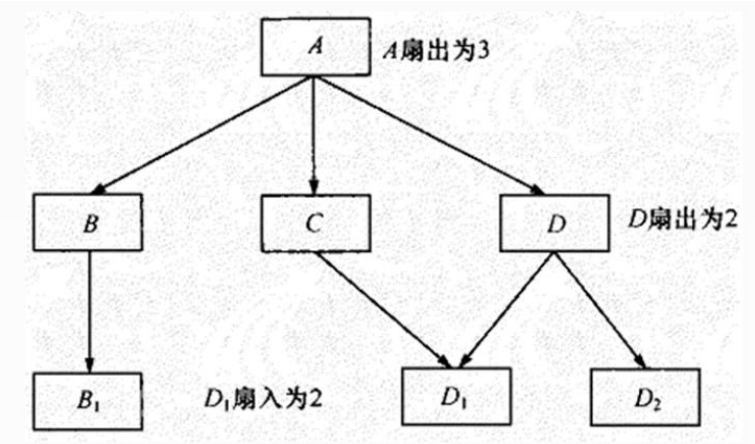
软件架构维护

◆软件架构可维护性度量实践：（★★★）
架构可维护性的六个子度量指标：
1.圈复杂度(CCN)
2.扇入扇出度(FFC)
3.模块间耦合度(CBO)
4.模块的响应(RFC)
5.紧内聚度TCC
6.松内聚度LCC

软件架构维护

1.圈复杂度(CCN)
由于在组件图中组件是独立的，每个组件代表一个系统或子系统中的封装单位，封装了完整的事务处理行为，组件图能够通过组件之间的控制依赖关系来体现整个系统的组成结构。
对架构的组件图进行圈复杂度的度量，可以对整个系统的复杂程度做出初步评估，在设计早期发现问题和做出调整，并预测待评估系统的测试复杂度，及早规避风险，提高软件质量。实践表明程序规模以CCN≤10为宜。（★★★）

2.扇入扇出度(FFC)
扇入是指直接调用该模块的上级模块的个数，扇出指该模块直接调用的下级模块的个数。
用扇入扇出度综合评估组件主动调用以及被调用的频率。扇入扇出度越大，表明该组件与其他组件间的接口关联或依赖关联越多。（★）

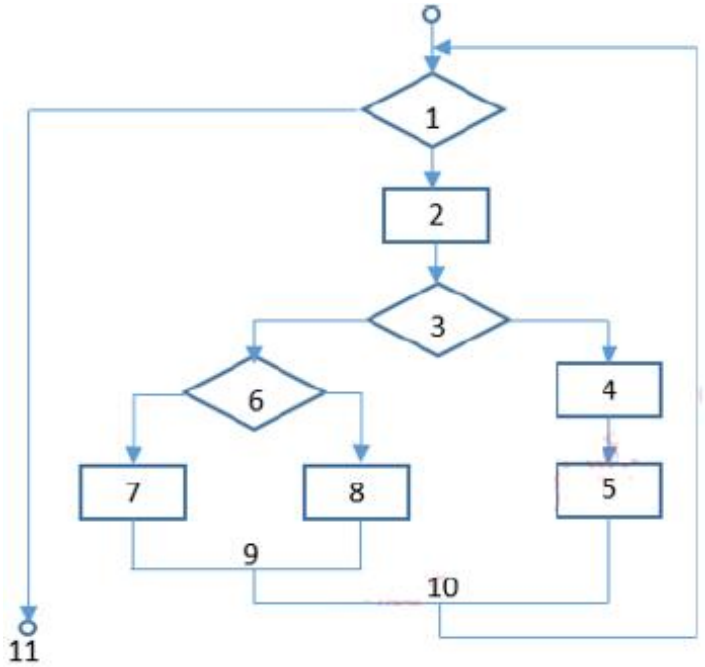


典型真题

使用 McCabe方法可以计算程序流程图的环形复杂度，下图的环形复杂度为( )。

A.3　　B.4　　C.5　　D.6

答案：B
解析：闭区间+1=3+1=4.



软件架构维护

3.模块间耦合度(CBO)（★）
模块间耦合度CBO度量模块与其他模块交互的频繁程度。
组件与其他组件的依赖关系及接口越多，该组件的耦合度越大。CBO 越大的模块，越容易受到其他模块修改和错误的影响，因而可维护性越差，风险越高。

4.模块的响应(RFC)
RFC度量组件执行所需的功能的数量，包括接口提供的功能、依赖的其他模块提供的功能以及子模块提供的功能。

5.模块间内聚度--紧内聚度TCC和松内聚度LCC
内聚度是模块内部各成份之间的联系紧密程度，联系越紧其内聚度越大。好的架构设计应该遵循"高内聚-低耦合"原则，提高模块的独立性，降低模块间接口调用的复杂性。

本章重点回顾

1、架构演化原则
2、软件演过过程实例（Redis、负载均衡技术和算法、数据库读写分离）
3、圈复杂度

THANKS