# 高并发

楼仔 著

目录

# 😊 前言

大家好，我是楼仔！

为了方便大家学习，我会把所有的系列文章整理成 PDF 手册。

高并发系列文章，是我今年 3 月初制定的计划，原计划 4 月底完成该系列文章的撰写，因为这段时间太忙，就一直拖到 5 月底。

我之前一直做电商领域，主实战，掌握的理论非常零碎，所以想通过这个系列，将自己的知识进行系统总结，于是就有了下面这 7 篇文章。

**文章内容肝、配图美、排版好看、可读性高**，每个细节我尽量做到精益求精，绝对经典。

# 📚 第 1 章：高并发分布式架构演进

之前给自己定了一个学习计划，今年上半年需要完成“高并发”系列文章，这个是该系列的第一篇。

在写“高并发”系列文章之前，我觉得有必要**让大家对高并发分布式架构有一个整体的认识，给大家搭建一套高并发的知识体系。**

**这套体系涉及的技术点非常多**，不限于熔断、流控、异步、池化、背压、负载均衡、主从分离、读写分离、缓存雪崩、缓存穿透、数据一致性、注册中心、配置中心、微服务等。

后面的系列文章，会根据这套体系涉及的知识点去讲解，当整个系列全部完成后，我会出一套高并发手册。

> 这篇文章是转载过来的，“高并发”的第一篇本来打算自己写，后来发现这篇写的非常不错，所以就没有单独写的必要，支持原创：
>
> - 原文作者：huashiou
> - 原文链接：https://segmentfault.com/a/1190000018626163

# 概述

**本文以淘宝作为例子，介绍从一百个到千万级并发情况下服务端的架构的演进过程**，同时列举出每个演进阶段会遇到的相关技术，让大家对架构的演进有一个整体的认知，文章最后汇总了一些架构设计的原则，架构演进过程如 下：

- 单机架构
- 第一次演进：Tomcat与数据库分开部署第
- 二次演进：引入本地缓存和分布式缓存第
- 三次演进：引入反向代理实现负载均衡第
- 四次演进：数据库读写分离
- 第五次演进：数据库按业务分库
- 第六次演进：把大表拆分为小表
- 第七次演进：使用LVS或F5来使多个Nginx负载均衡
- 第八次演进：通过DNS轮询实现机房间的负载均衡
- 第九次演进：引入NoSQL数据库和搜索引擎等技术
- 第十次演进：大应用拆分为小应用

- 第十一次演进：复用的功能抽离成微服务
- 第十二次演进：引入企业服务总线ESB屏蔽服务接口的访问差异 第
- 十三次演进：引入容器化技术实现运行环境隔离与动态服务管理第十
- 四次演进：以云平台承载系统

> 特别说明：本文以淘宝为例仅仅是为了便于说明演进过程可能遇到的问题，并非是淘宝真正的技术演进路径。

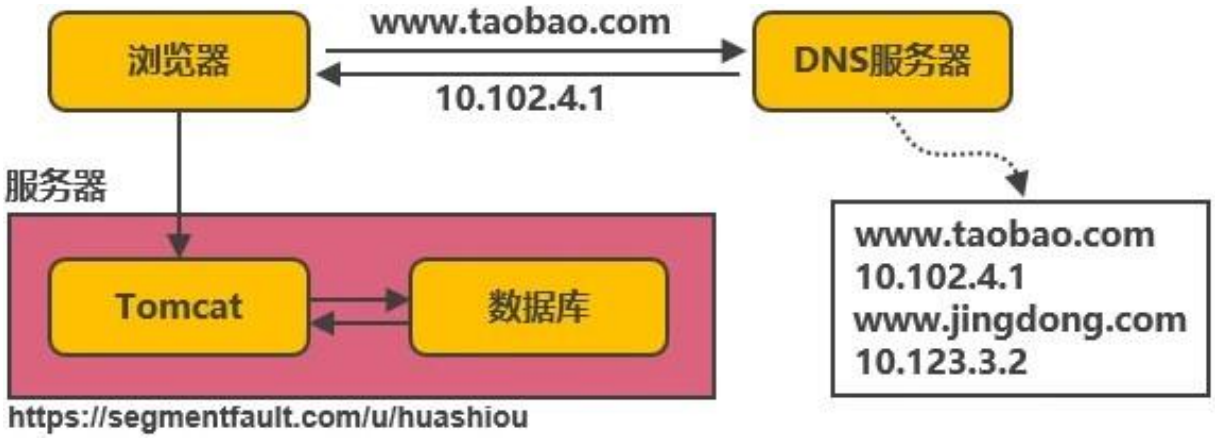
# 基本概念

在介绍架构之前，为了避免部分读者对架构设计中的一些概念不了解，下面对几个最基础的概念进行介绍：

- 分布式：系统中的多个模块在不同服务器上部署，即可称为分布式系统，如Tomcat和数据库分别部署在不同的服务器上，或两个相同功能的Tomcat分别部署在不同服务器上
- 高可用：系统中部分节点失效时，其他节点能够接替它继续提供服务，则可认为系统具有高可用性
- 集群：一个特定领域的软件部署在多台服务器上并作为一个整体提供一类服务，这个整体称为集群。如Zookeeper中的Master和Slave分别部署在多台服务器上，共同组成一个整体提供集中配置服务。在常见的集群中，客户端往往能够连接任意一个节点获得服务，并且当集群中一个节点掉线时，其他节点往往能够自动的接替它继续提供服务，这时候说明集群具有高可用性
- 负载均衡：请求发送到系统时，通过某些方式把请求均匀分发到多个节点上，使系统中每个节点能够均匀的处理请求负载，则可认为系统是负载均衡的
- 正向代理和反向代理：系统内部要访问外部网络时，统一通过一个代理服务器把请求转发出去，在外部网络看来就是代理服务器发起的访问，此时代理服务器实现的是正向代理；当外部请求进入系统时，代理服务器把该请求转发到系统中的某台服务器上，对外部请求来说，与之交互的只有代理服务器，此时代理服务器实现的是反向代理。简单来说，正向代理是代理服务器代替系统内部来访问外部网络的过程，反向代理是外部请求访问系统时通过代理服务器转发到内部服务器的过程。

# 架构演进

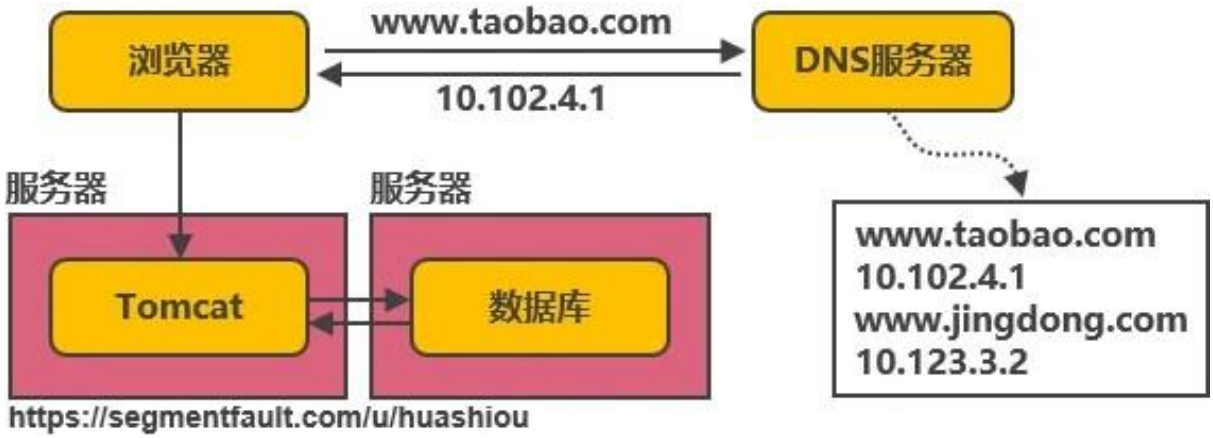
## 单机架构




以淘宝作为例子。在网站最初时，应用数量与用户数都较少，可以把Tomcat和数据库部署在同一台服务器上。浏览器往www.taobao.com发起请求时，首先经过DNS服务器（域名系统）把域名转换为实际IP地址10.102.4.1，浏览器转而访问该IP对应的Tomcat。

> 随着用户数的增长，Tomcat和数据库之间竞争资源，单机性能不足以支撑业务。

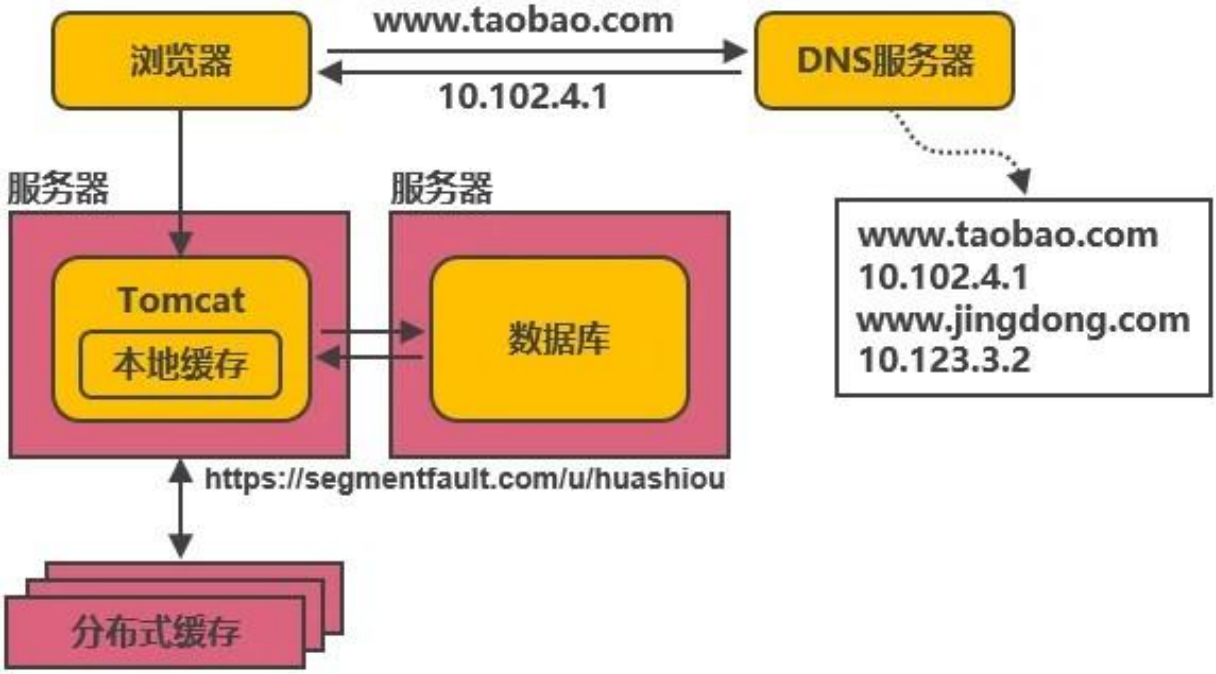
# 第一次演进：Tomcat与数据库分开部署




Tomcat和数据库分别独占服务器资源，显著提高两者各自性能。随

> 着用户数的增长，并发读写数据库成为瓶颈。

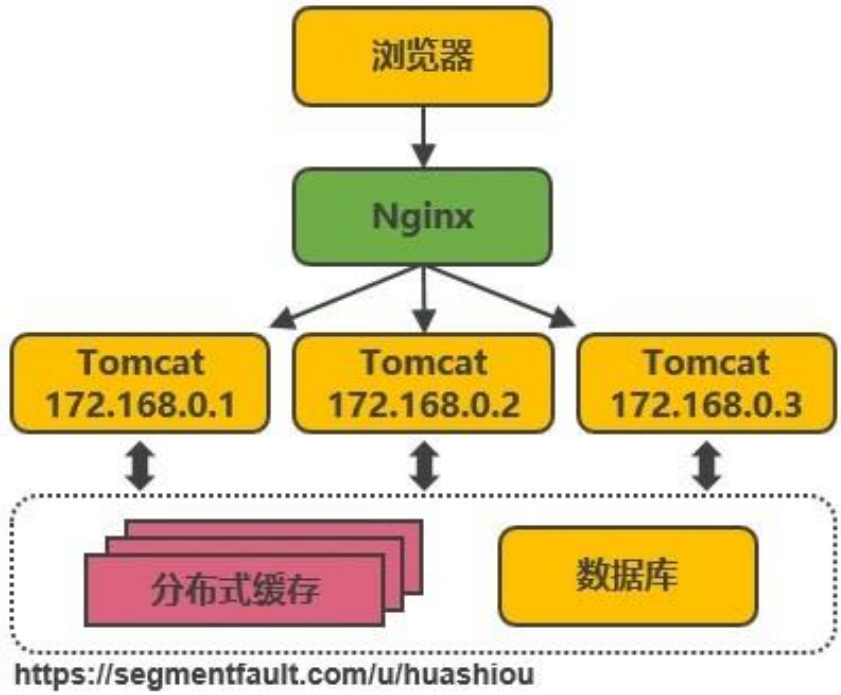
# 第二次演进：引入本地缓存和分布式缓存




在Tomcat同服务器上或同JVM中增加本地缓存，并在外部增加分布式缓存，缓存热门商品信息或热门商品的html页面等。通过缓存能把绝大多数请求在读写数据库前拦截掉，大大降低数据库压力。其中涉及的技术包括：使用memcached作为本地缓存，使用Redis作为分布式缓存，还会涉及缓存一致性、缓存穿透/击穿、缓存雪崩、热点数据集中失效等问题。

> 缓存抗住了大部分的访问请求，随着用户数的增长，并发压力主要落在单机的Tomcat上，响应逐渐变慢

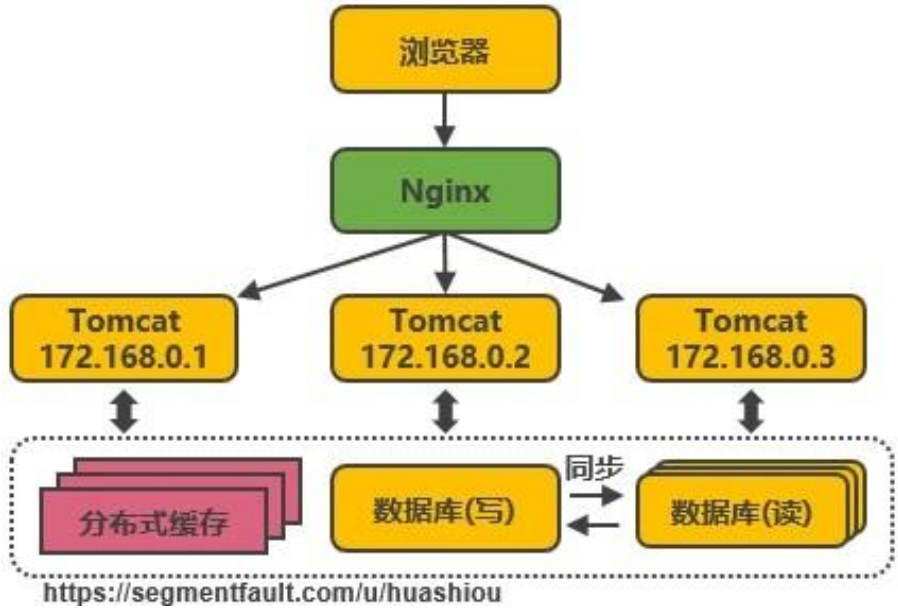
# 第三次演进：引入反向代理实现负载均衡




在多台服务器上分别部署Tomcat，使用反向代理软件（Nginx）把请求均匀分发到每个Tomcat中。此处假设 Tomcat 最多支持100个并发，Nginx最多支持50000个并发，那么理论上Nginx把请求分发到500个Tomcat上，就能抗住50000个并发。其中涉及的技术包括：Nginx、HAProxy，两者都是工作在网络第七层的反向代理软件，主要支持http协议，还会涉及session共享、文件上传下载的问题。

> 反向代理使应用服务器可支持的并发量大大增加，但并发量的增长也意味着更多请求穿透到数据库，单机的数据库最终成为瓶颈。

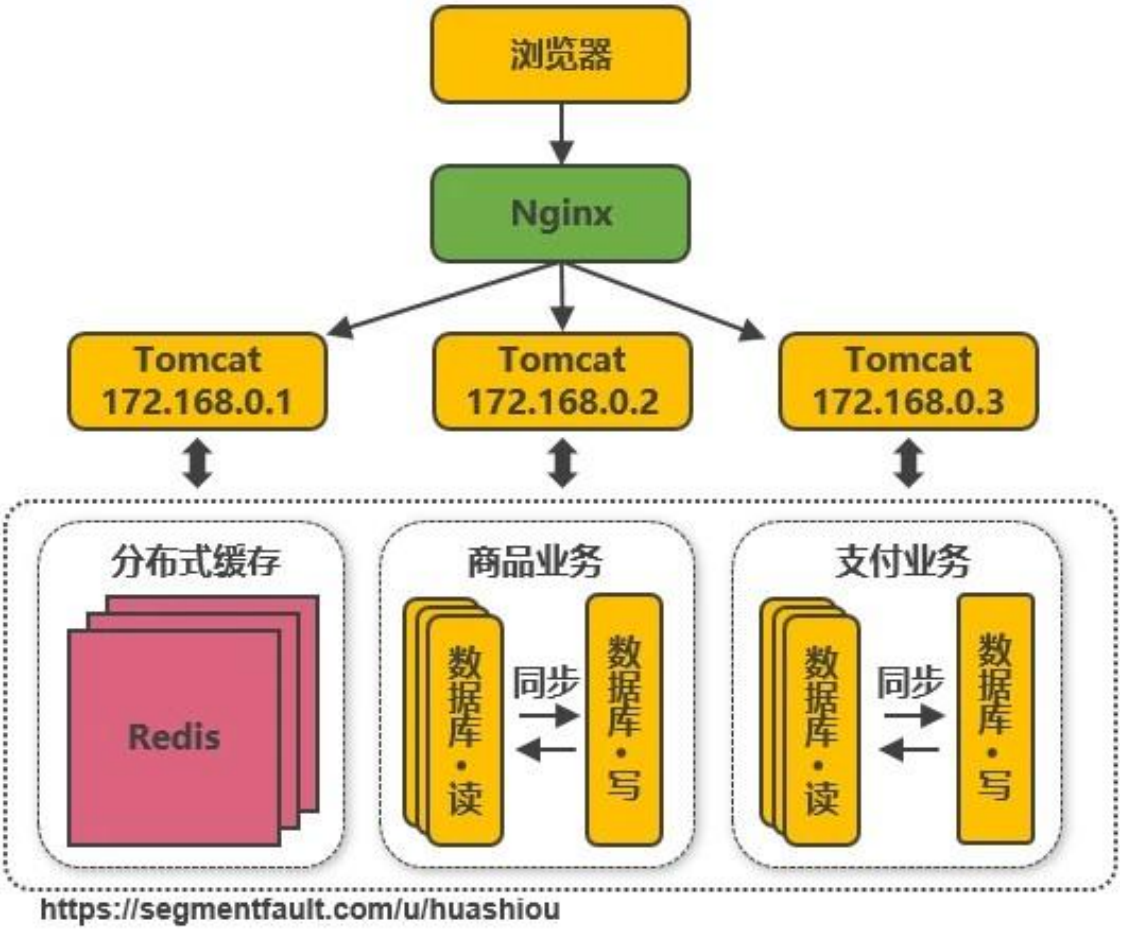
# 第四次演进：数据库读写分离




把数据库划分为读库和写库，读库可以有多个，通过同步机制把写库的数据同步到读库，对于需要查询最新写入数据场景，可通过在缓存中多写一份，通过缓存获得最新数据。其中涉及的技术包括：Mycat，它是数据库中间件，可通过它来组织数据库的分离读写和分库分表，客户端通过它来访问下层数据库，还会涉及数据同步，数据一致性的问题。

> 业务逐渐变多，不同业务之间的访问量差距较大，不同业务直接竞争数据库，相互影响性能。

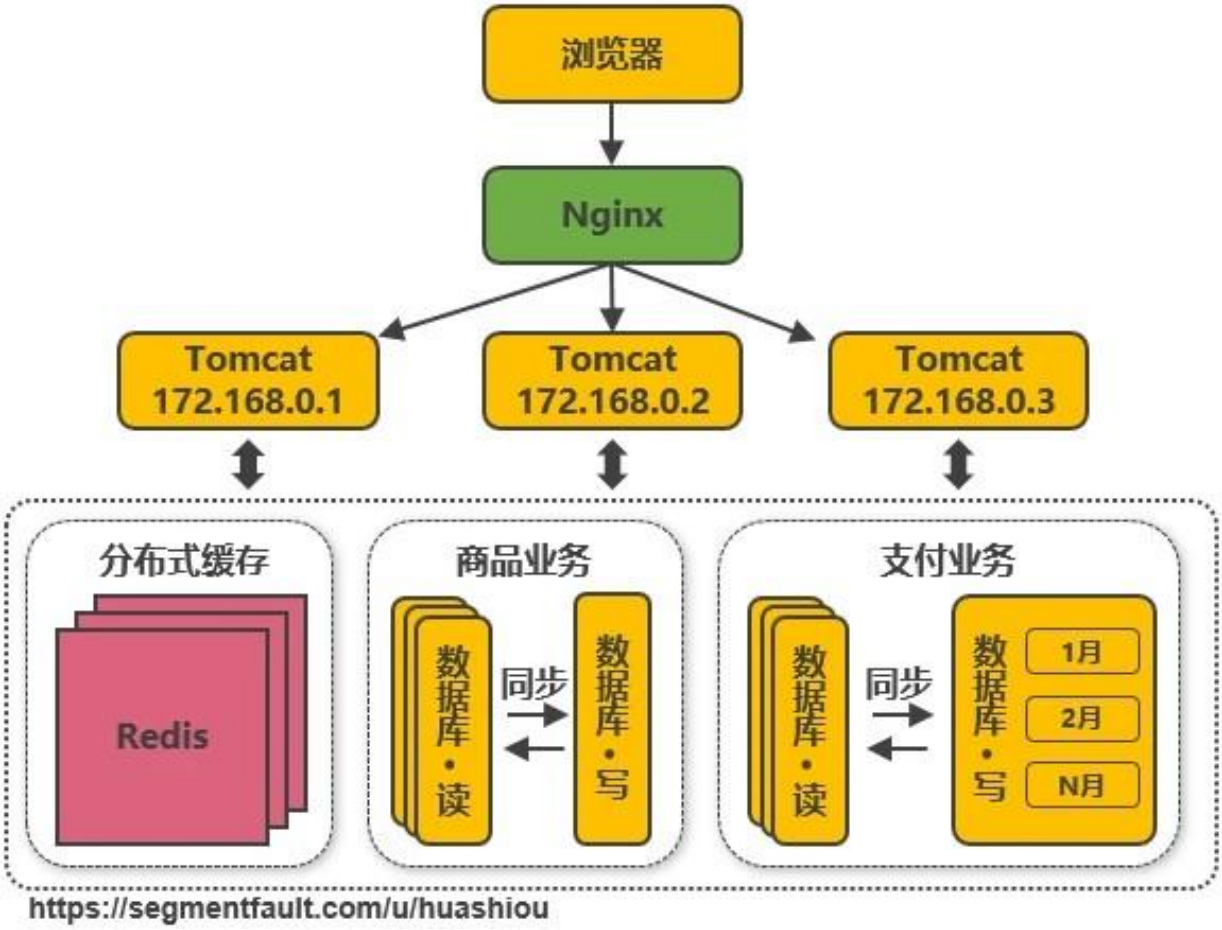
# 第五次演进：数据库按业务分库




把不同业务的数据保存到不同的数据库中，使业务之间的资源竞争降低，对于访问量大的业务，可以部署更多的服务器来支撑。这样同时导致跨业务的表无法直接做关联分析，需要通过其他途径来解决，但这不是本文讨论的重 点，有兴趣的可以自行搜索解决方案。

> 随着用户数的增长，单机的写库会逐渐会达到性能瓶颈。

# 第六次演进：把大表拆分为小表



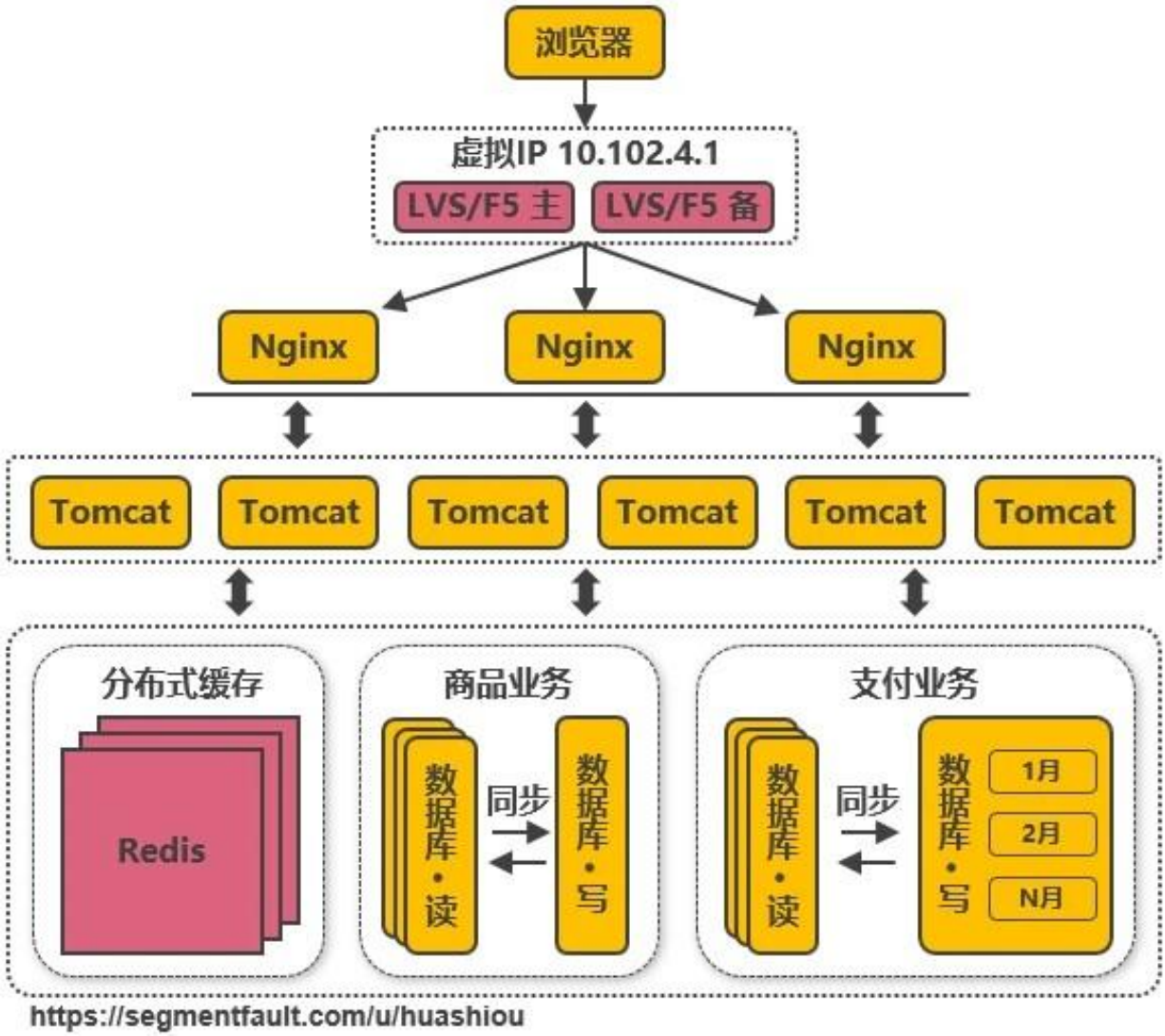

比如针对评论数据，可按照商品ID进行hash，路由到对应的表中存储；针对支付记录，可按照小时创建表，每个小时表继续拆分为小表，使用用户ID或记录编号来路由数据。只要实时操作的表数据量足够小，请求能够足够均匀的分发到多台服务器上的小表，那数据库就能通过水平扩展的方式来提高性能。其中前面提到的Mycat也支持在大表拆分为小表情况下的访问控制。

这种做法显著的增加了数据库运维的难度，对DBA的要求较高。数据库设计到这种结构时，已经可以称为分布式数据库，但是这只是一个逻辑的数据库整体，数据库里不同的组成部分是由不同的组件单独来实现的，如分库分表的管理和请求分发，由Mycat实现，SQL的解析由单机的数据库实现，读写分离可能由网关和消息队列来实现，查询结果的汇总可能由数据库接口层来实现等等，这种架构其实是MPP（大规模并行处理）架构的一类实现。

目前开源和商用都已经有不少MPP数据库，开源中比较流行的有Greenplum、TiDB、Postgresql XC、HAWQ等，商用的如南大通用的GBase、睿帆科技的雪球DB、华为的LibrA等等，不同的MPP数据库的侧重点也不一样，如 TiDB更侧重于分布式OLTP场景，Greenplum更侧重于分布式OLAP场景，这些MPP数据库基本都提供了类似 Postgresql、Oracle、MySQL那样的SQL标准支持能力，能把一个查询解析为分布式的执行计划分发到每台机器上并行执行，最终由数据库本身汇总数据进行返回，也提供了诸如权限管理、分库分表、事务、数据副本等能力，并且大多能够支持100个节点以上的集群，大大降低了数据库运维的成本，并且使数据库也能够实现水平扩展。

> 数据库和Tomcat都能够水平扩展，可支撑的并发大幅提高，随着用户数的增长，最终单机的Nginx会成为瓶颈。

## 第七次演进：使用LVS或F5来使多个Nginx负载均衡

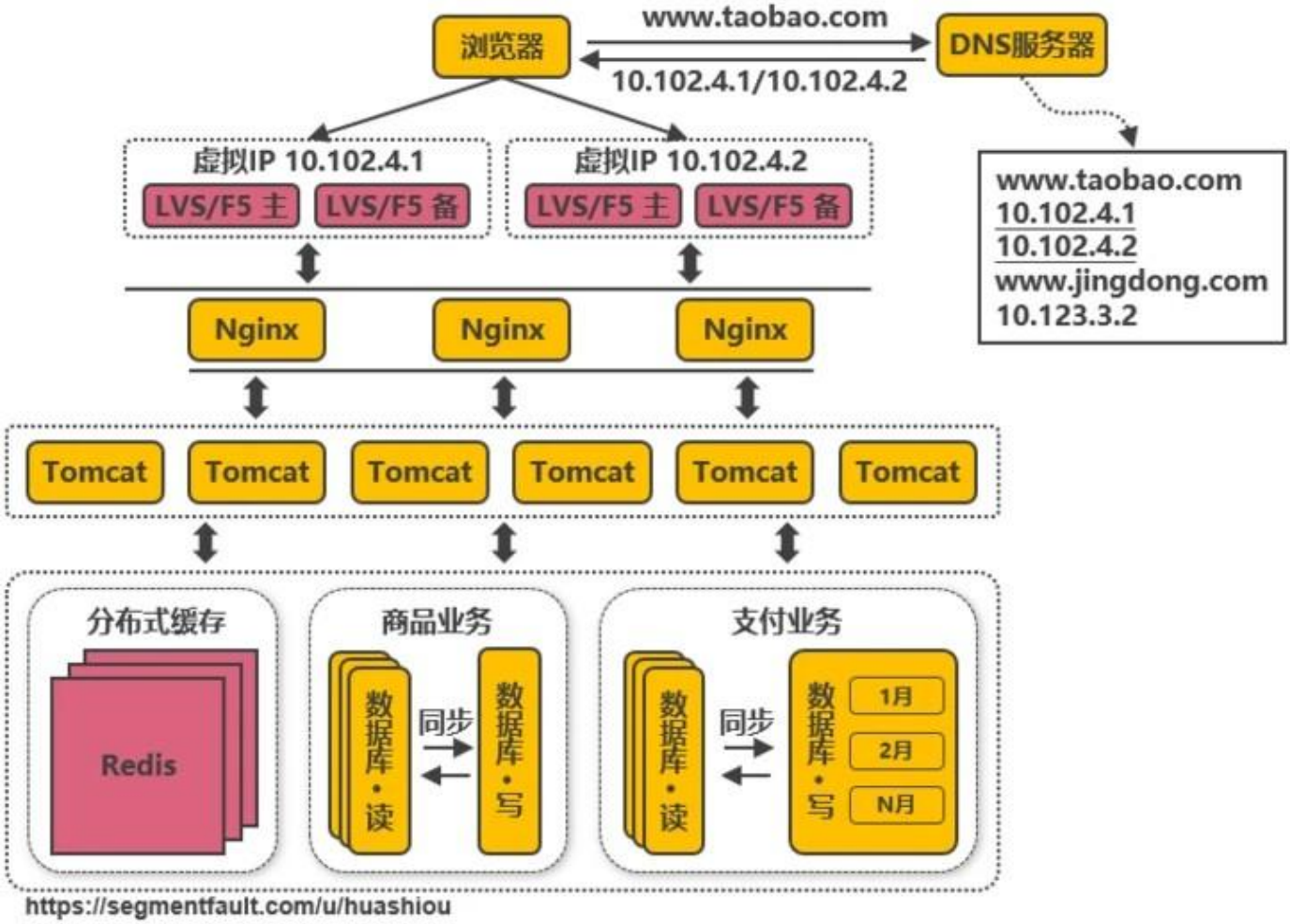



由于瓶颈在Nginx，因此无法通过两层的Nginx来实现多个Nginx的负载均衡。图中的LVS和F5是工作在网络第四层的负载均衡解决方案，其中LVS是软件，运行在操作系统内核态，可对TCP请求或更高层级的网络协议进行转发，因此支持的协议更丰富，并且性能也远高于Nginx，可假设单机的LVS可支持几十万个并发的请求转发；F5是一种负载均衡硬件，与LVS提供的能力类似，性能比LVS更高，但价格昂贵。由于LVS是单机版的软件，若LVS所在服务器宕机则会导致整个后端系统都无法访问，因此需要有备用节点。可使用keepalived软件模拟出虚拟IP，然后把虚拟IP绑定到多台LVS服务器上，浏览器访问虚拟IP时，会被路由器重定向到真实的LVS服务器，当主LVS服务器宕机时，keepalived软件会自动更新路由器中的路由表，把虚拟IP重定向到另外一台正常的LVS服务器，从而达到LVS服务器高可用的效果。

此处需要注意的是，上图中从Nginx层到Tomcat层这样画并不代表全部Nginx都转发请求到全部的Tomcat，在实际使用时，可能会是几个Nginx下面接一部分的Tomcat，这些Nginx之间通过keepalived实现高可用，其他的 Nginx接另外的Tomcat，这样可接入的Tomcat数量就能成倍的增加。

> 由于LVS也是单机的，随着并发数增长到几十万时，LVS服务器最终会达到瓶颈，此时用户数达到千万甚至上亿级别，用户分布在不同的地区，与服务器机房距离不同，导致了访问的延迟会明显不同。

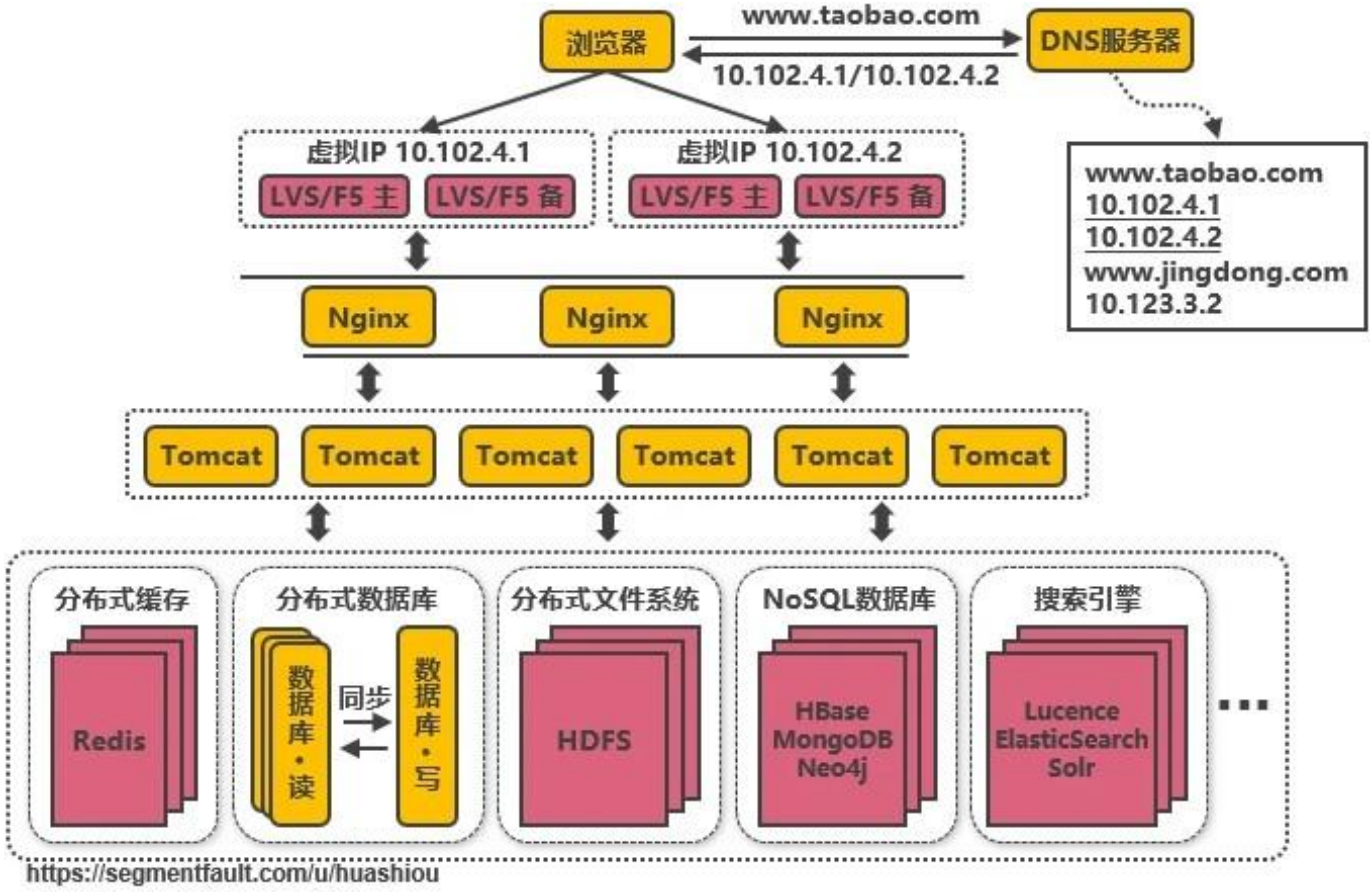
# 第八次演进：通过DNS轮询实现机房间的负载均衡




在DNS服务器中可配置一个域名对应多个IP地址，每个IP地址对应到不同的机房里的虚拟IP。当用户访问www.taobao.com时，DNS服务器会使用轮询策略或其他策略，来选择某个IP供用户访问。此方式能实现机房间的负载均 衡，至此，系统可做到机房级别的水平扩展，千万级到亿级的并发量都可通过增加机房来解决，系统入口处的请求并发量不再是问题。

> 随着数据的丰富程度和业务的发展，检索、分析等需求越来越丰富，单单依靠数据库无法解决如此丰富的需求。

# 第九次演进：引入NoSQL数据库和搜索引擎等技术

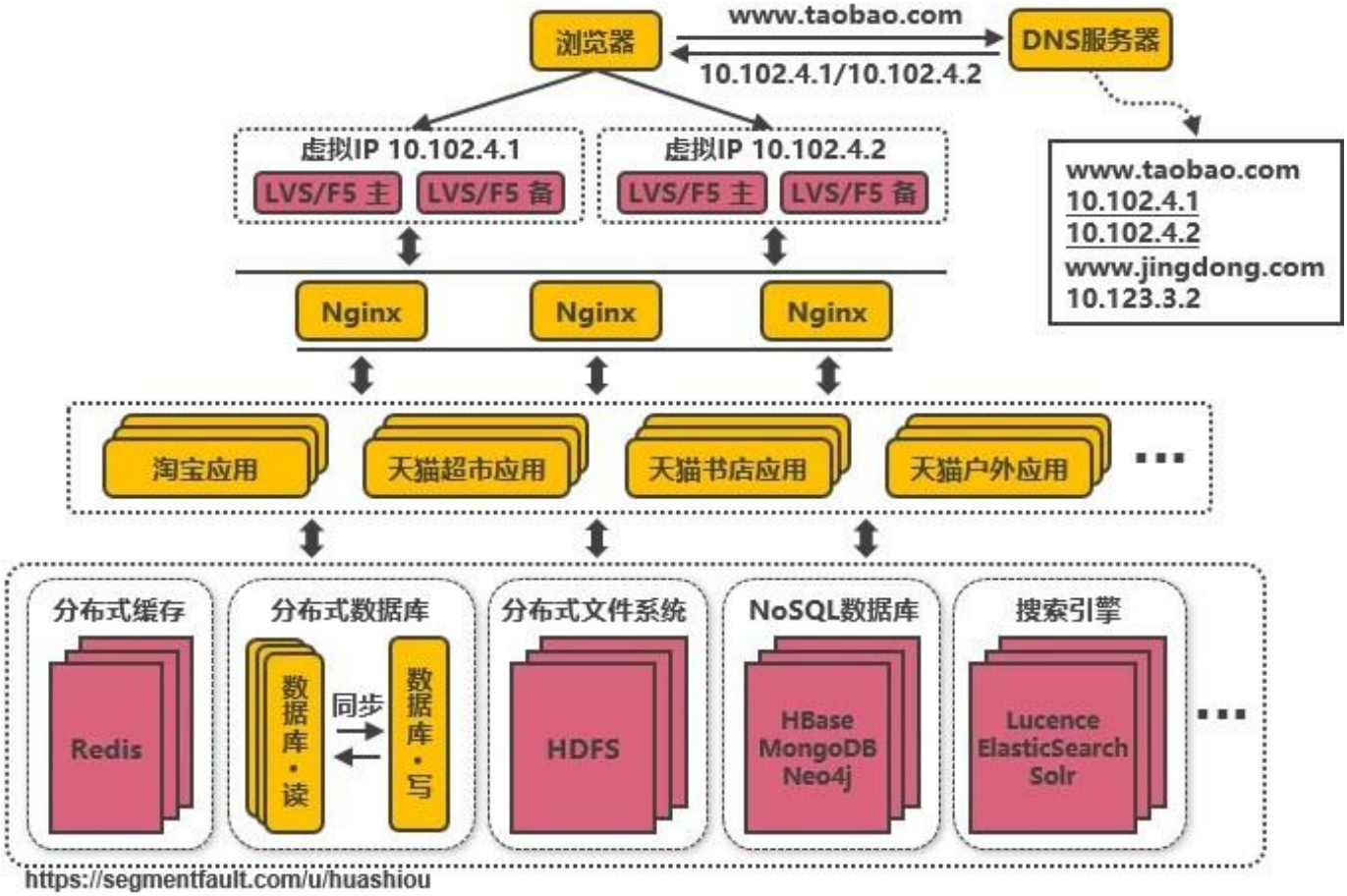



当数据库中的数据多到一定规模时，数据库就不适用于复杂的查询了，往往只能满足普通查询的场景。对于统计报表场景，在数据量大时不一定能跑出结果，而且在跑复杂查询时会导致其他查询变慢，对于全文检索、可变数据结构等场景，数据库天生不适用。因此需要针对特定的场景，引入合适的解决方案。如对于海量文件存储，可通过分布式文件系统HDFS解决，对于key value类型的数据，可通过HBase和Redis等方案解决，对于全文检索场景，可通过搜索引擎如ElasticSearch解决，对于多维分析场景，可通过Kylin或Druid等方案解决。

当然，引入更多组件同时会提高系统的复杂度，不同的组件保存的数据需要同步，需要考虑一致性的问题，需要有更多的运维手段来管理这些组件等。

> 引入更多组件解决了丰富的需求，业务维度能够极大扩充，随之而来的是一个应用中包含了太多的业务代码，业务的升级迭代变得困难。

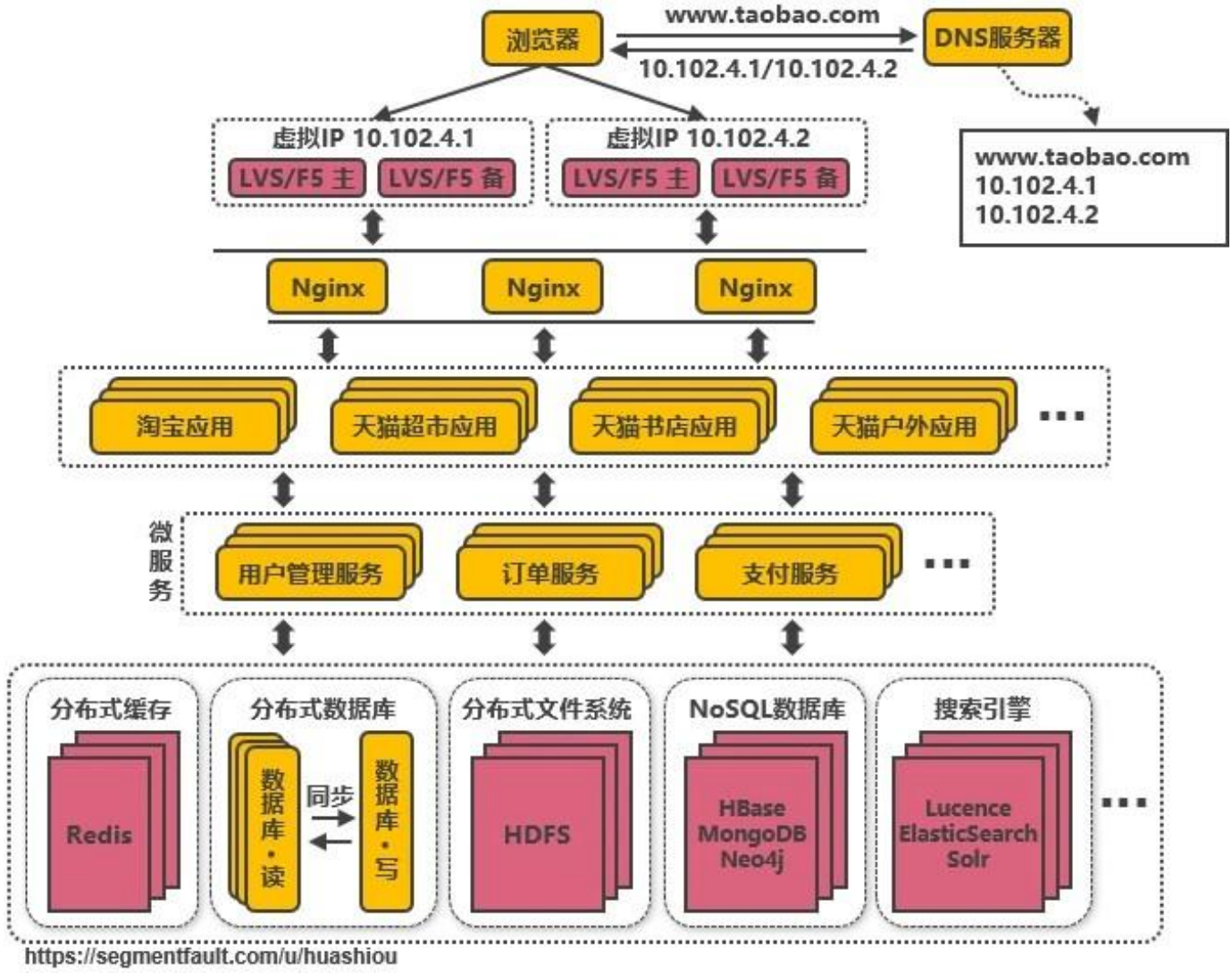
# 第十次演进：大应用拆分为小应用




按照业务板块来划分应用代码，使单个应用的职责更清晰，相互之间可以做到独立升级迭代。这时候应用之间可能会涉及到一些公共配置，可以通过分布式配置中心Zookeeper来解决。

> 不同应用之间存在共用的模块，由应用单独管理会导致相同代码存在多份，导致公共功能升级时全部应用代码都要跟着升级。

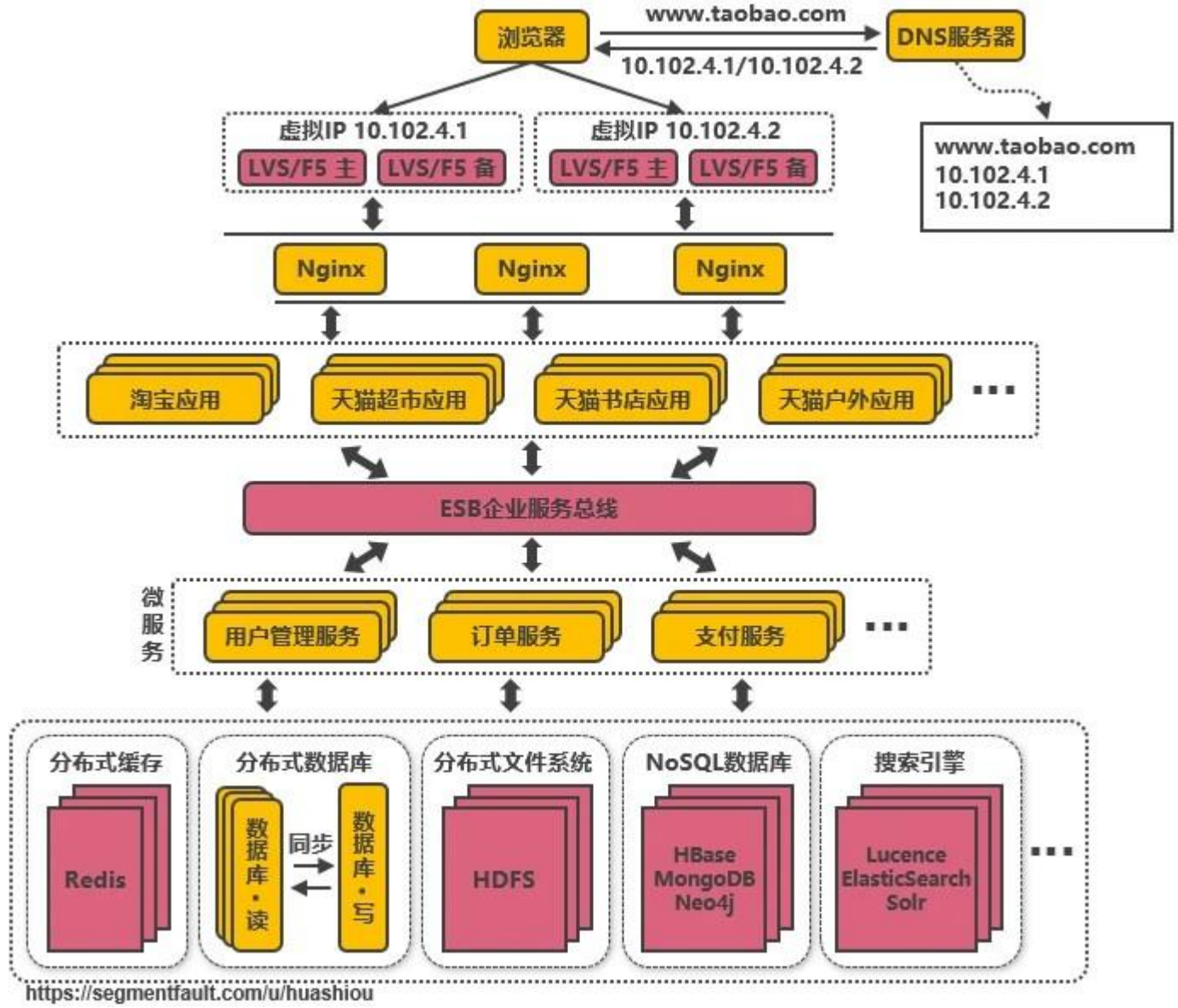
# 第十一次演进：复用的功能抽离成微服务




如用户管理、订单、支付、鉴权等功能在多个应用中都存在，那么可以把这些功能的代码单独抽取出来形成一个单独的服务来管理，这样的服务就是所谓的微服务，应用和服务之间通过HTTP、TCP或RPC请求等多种方式来访问公共服务，每个单独的服务都可以由单独的团队来管理。此外，可以通过Dubbo、SpringCloud等框架实现服务治 理、限流、熔断、降级等功能，提高服务的稳定性和可用性。

> 不同服务的接口访问方式不同，应用代码需要适配多种访问方式才能使用服务，此外，应用访问服务，服务之间也可能相互访问，调用链将会变得非常复杂，逻辑变得混乱。

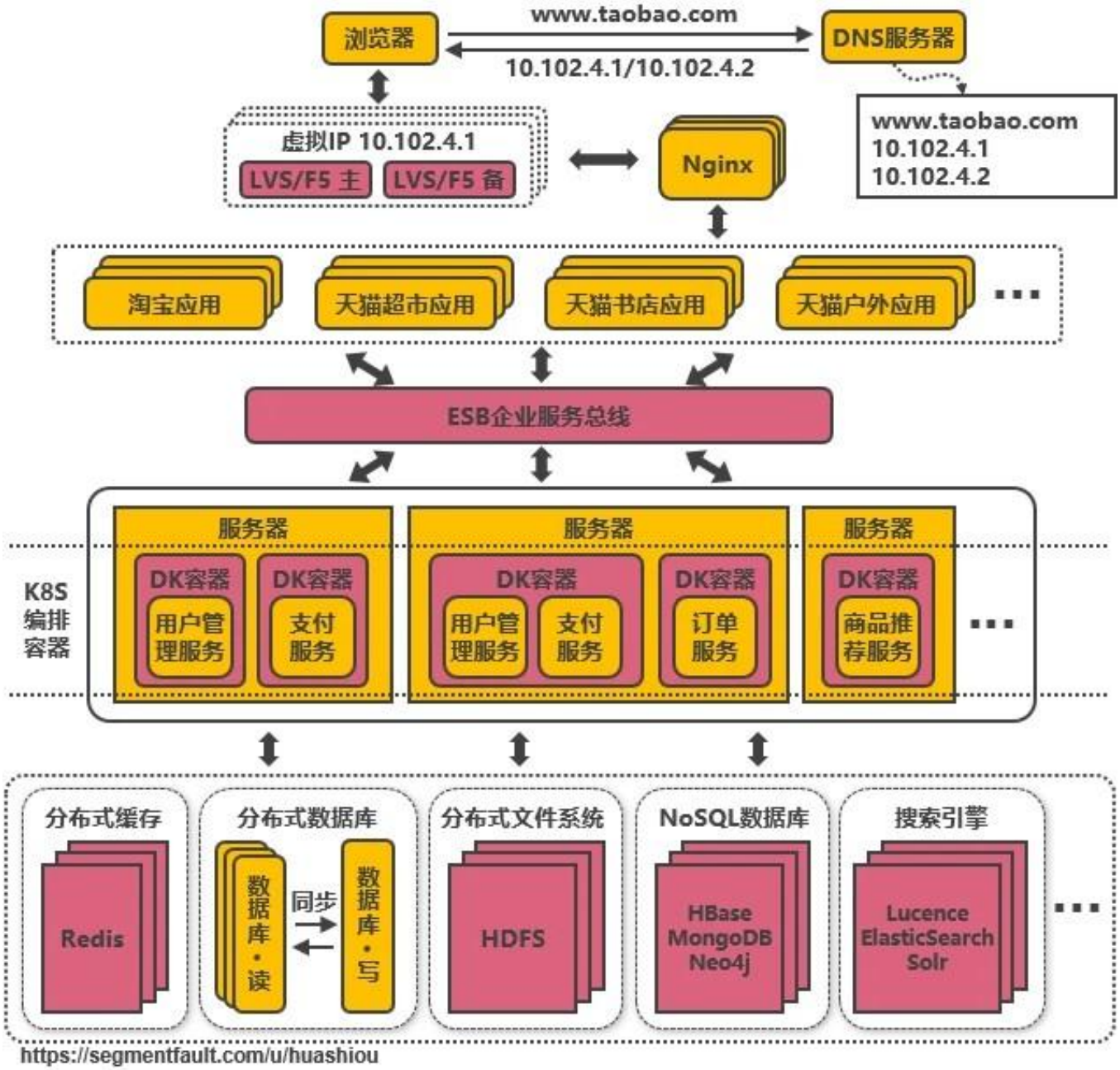
# 第十二次演进：引入企业服务总线ESB屏蔽服务接口的访问差异




通过ESB统一进行访问协议转换，应用统一通过ESB来访问后端服务，服务与服务之间也通过ESB来相互调用，以此降低系统的耦合程度。这种单个应用拆分为多个应用，公共服务单独抽取出来来管理，并使用企业消息总线来解除服务之间耦合问题的架构，就是所谓的SOA（面向服务）架构，这种架构与微服务架构容易混淆，因为表现形式十分相似。个人理解，微服务架构更多是指把系统里的公共服务抽取出来单独运维管理的思想，而SOA架构则是指一种拆分服务并使服务接口访问变得统一的架构思想，SOA架构中包含了微服务的思想。

> 业务不断发展，应用和服务都会不断变多，应用和服务的部署变得复杂，同一台服务器上部署多个服务还要解决运行环境冲突的问题，此外，对于如大促这类需要动态扩缩容的场景，需要水平扩展服务的性能，就需要在新增的服务上准备运行环境，部署服务等，运维将变得十分困难。

# 第十三次演进：引入容器化技术实现运行环境隔离与动态服务管理

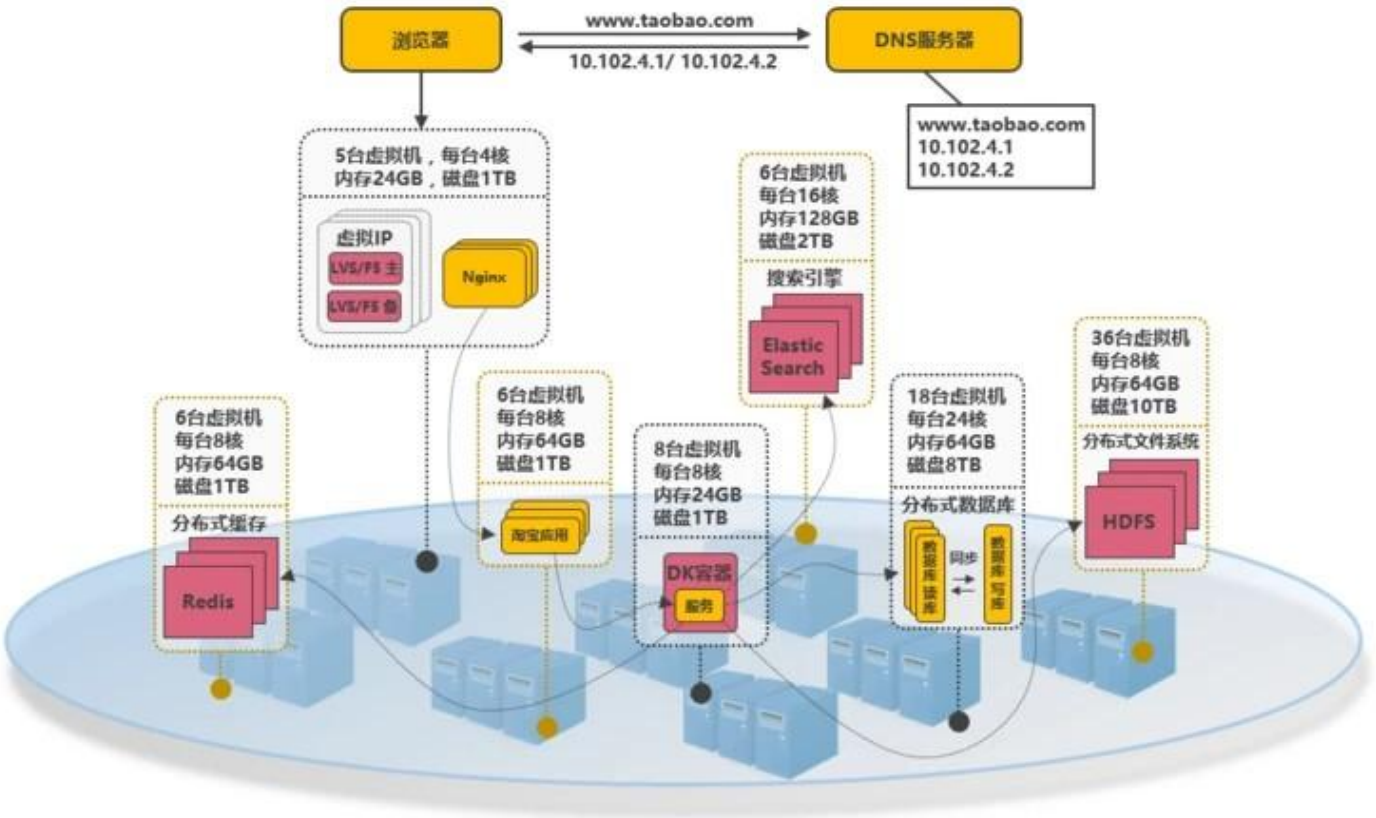



目前最流行的容器化技术是Docker，最流行的容器管理服务是Kubernetes(K8S)，应用/服务可以打包为Docker镜像，通过K8S来动态分发和部署镜像。Docker镜像可理解为一个能运行你的应用/服务的最小的操作系统，里面放着应用/服务的运行代码，运行环境根据实际的需要设置好。把整个“操作系统”打包为一个镜像后，就可以分发到需要部署相关服务的机器上，直接启动Docker镜像就可以把服务起起来，使服务的部署和运维变得简单。

在大促的之前，可以在现有的机器集群上划分出服务器来启动Docker镜像，增强服务的性能，大促过后就可以关闭镜像，对机器上的其他服务不造成影响（在3.14节之前，服务运行在新增机器上需要修改系统配置来适配服务，这会导致机器上其他服务需要的运行环境被破坏）。

> 使用容器化技术后服务动态扩缩容问题得以解决，但是机器还是需要公司自身来管理，在非大促的时候，还是需要闲置着大量的机器资源来应对大促，机器自身成本和运维成本都极高，资源利用率低。

# 第十四次演进：以云平台承载系统




系统可部署到公有云上，利用公有云的海量机器资源，解决动态硬件资源的问题，在大促的时间段里，在云平台中临时申请更多的资源，结合Docker和K8S来快速部署服务，在大促结束后释放资源，真正做到按需付费，资源利用率大大提高，同时大大降低了运维成本。

所谓的云平台，就是把海量机器资源，通过统一的资源管理，抽象为一个资源整体，在之上可按需动态申请硬件资源（如CPU、内存、网络等），并且之上提供通用的操作系统，提供常用的技术组件（如Hadoop技术栈，MPP数据库等）供用户使用，甚至提供开发好的应用，用户不需要关系应用内部使用了什么技术，就能够解决需求（如音视频转码服务、邮件服务、个人博客等）。在云平台中会涉及如下几个概念：

- IaaS：基础设施即服务。对应于上面所说的机器资源统一为资源整体，可动态申请硬件资源的层面；
- PaaS：平台即服务。对应于上面所说的提供常用的技术组件方便系统的开发和维护；
- SaaS：软件即服务。对应于上面所说的提供开发好的应用或服务，按功能或性能要求付费。

> 至此，以上所提到的从高并发访问问题，到服务的架构和系统实施的层面都有了各自的解决方案，但同时也应该意识到，在上面的介绍中，其实是有意忽略了诸如跨机房数据同步、分布式事务实现等等的实际问题，这些问题以后有机会再拿出来单独讨论。

# 架构设计总结

- **架构的调整是否必须按照上述演变路径进行？** 不是的，以上所说的架构演变顺序只是针对某个侧面进行单独的改进，在实际场景中，可能同一时间会有几个问题需要解决，或者可能先达到瓶颈的是另外的方面，这时候就应该按照实际问题实际解决。如在政府类的并发量可能不大，但业务可能很丰富的场景，高并发就不是重点解决的问题，此时优先需要的可能会是丰富需求的解决方案。
- **对于将要实施的系统，架构应该设计到什么程度？** 对于单次实施并且性能指标明确的系统，架构设计到能够支持系统的性能指标要求就足够了，但要留有扩展架构的接口以便不备之需。对于不断发展的系统，如电商平台，应设计到能满足下一阶段用户量和性能指标要求的程度，并根据业务的增长不断的迭代升级架构，以支持更高的并发和更丰富的业务。

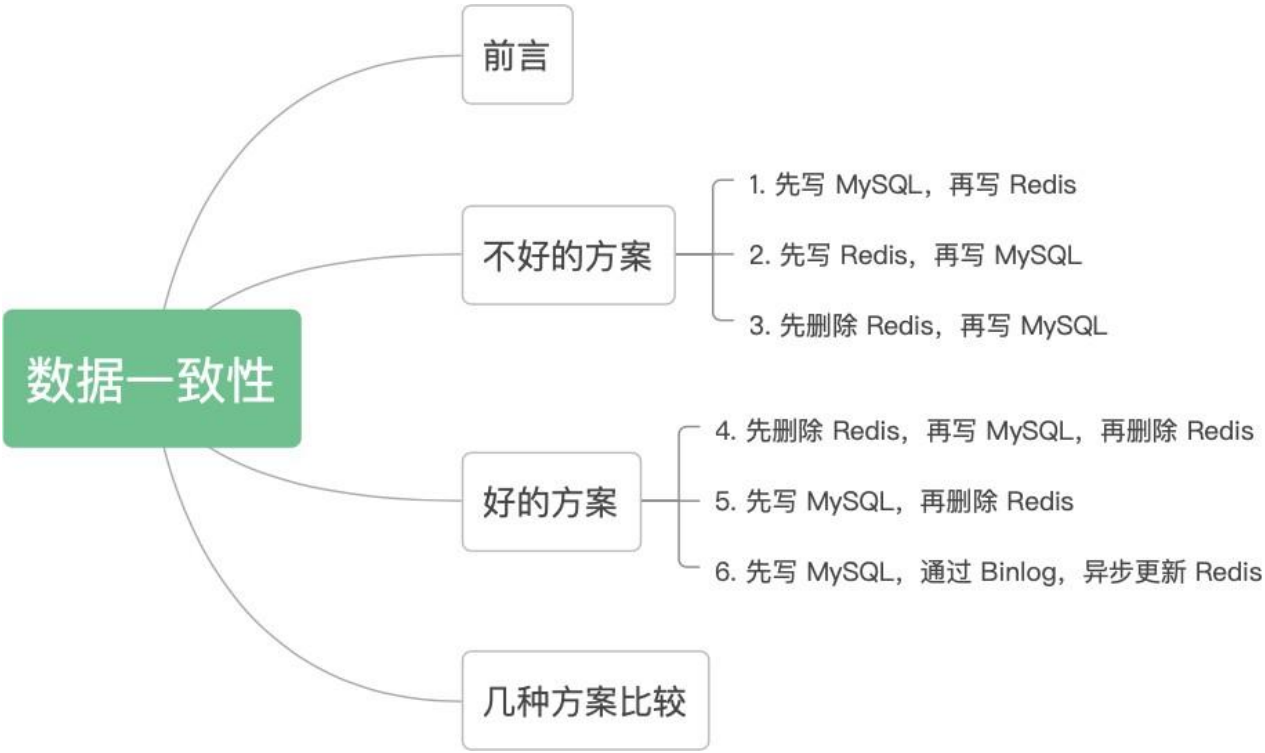
- **服务端架构和大数据架构有什么区别？** 所谓的“大数据”其实是海量数据采集清洗转换、数据存储、数据分析、数据服务等场景解决方案的一个统称，在每一个场景都包含了多种可选的技术，如数据采集有Flume、 Sqoop、Kettle等，数据存储有分布式文件系统HDFS、FastDFS，NoSQL数据库HBase、MongoDB等，数据分析有Spark技术栈、机器学习算法等。总的来说大数据架构就是根据业务的需求，整合各种大数据组件组合而成的架构，一般会提供分布式存储、分布式计算、多维分析、数据仓库、机器学习算法等能力。而服务端架构更多指的是应用组织层面的架构，底层能力往往是由大数据架构来提供。
- **有没有一些架构设计的原则？**
    - N+1设计。系统中的每个组件都应做到没有单点故障；
    - 回滚设计。确保系统可以向前兼容，在系统升级时应能有办法回滚版本；
    - 禁用设计。应该提供控制具体功能是否可用的配置，在系统出现故障时能够快速下线功能；
    - 监控设计。在设计阶段就要考虑监控的手段；
    - 多活数据中心设计。若系统需要极高的高可用，应考虑在多地实施数据中心进行多活，至少在一个机房断电的情况下系统依然可用；
    - 采用成熟的技术。刚开发的或开源的技术往往存在很多隐藏的bug，出了问题没有商业支持可能会是一个灾难；
    - 资源隔离设计。应避免单一业务占用全部资源；
    - 架构应能水平扩展。系统只有做到能水平扩展，才能有效避免瓶颈问题；
    - 非核心则购买。非核心功能若需要占用大量的研发资源才能解决，则考虑购买成熟的产品；
    - 使用商用硬件。商用硬件能有效降低硬件故障的机率；
    - 快速迭代。系统应该快速开发小功能模块，尽快上线进行验证，早日发现问题大大降低系统交付的风险；
    - 无状态设计。服务接口应该做成无状态的，当前接口的访问不依赖于接口上次访问的状态。

# 📚 第 2 章：MySQL 和 Redis 一致性

这个问题很早之前我就遇到过，但是一直没有仔细去研究，上个月看了极客的课程，有一篇文章专门有过讲解，刚好有粉丝也问我这个问题，所以感觉有必要单独出一篇。

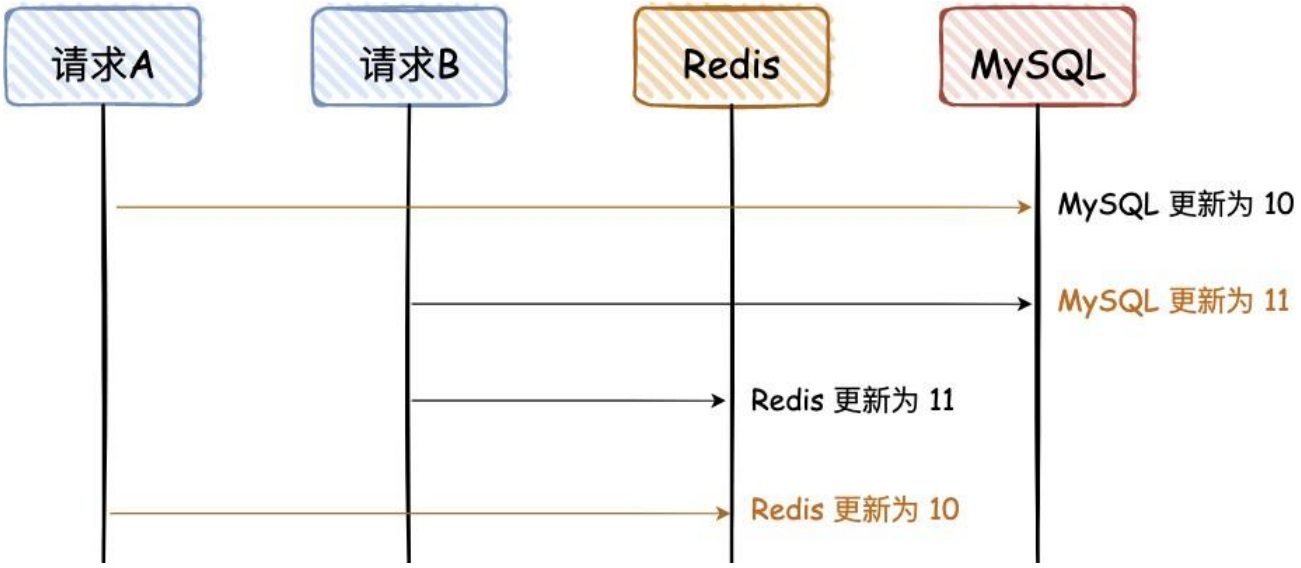
**之前也看了很多相关的文章，但是感觉讲的都不好**，很多文章都会去讲各种策略，比如（旁路缓存）策略、（读穿 / 写穿）策略和（写回）策略等，感觉意义真的不大，然后有的文章也只讲了部分情况，也没有告诉最优解。

我直接先抛一下结论：**在满足实时性的条件下，不存在两者完全保存一致的方案，只有最终一致性方案。** 根据网上的众多解决方案，总结出 6 种，直接看目录：




# 不好的方案

## 1. 先写 MySQL，再写 Redis

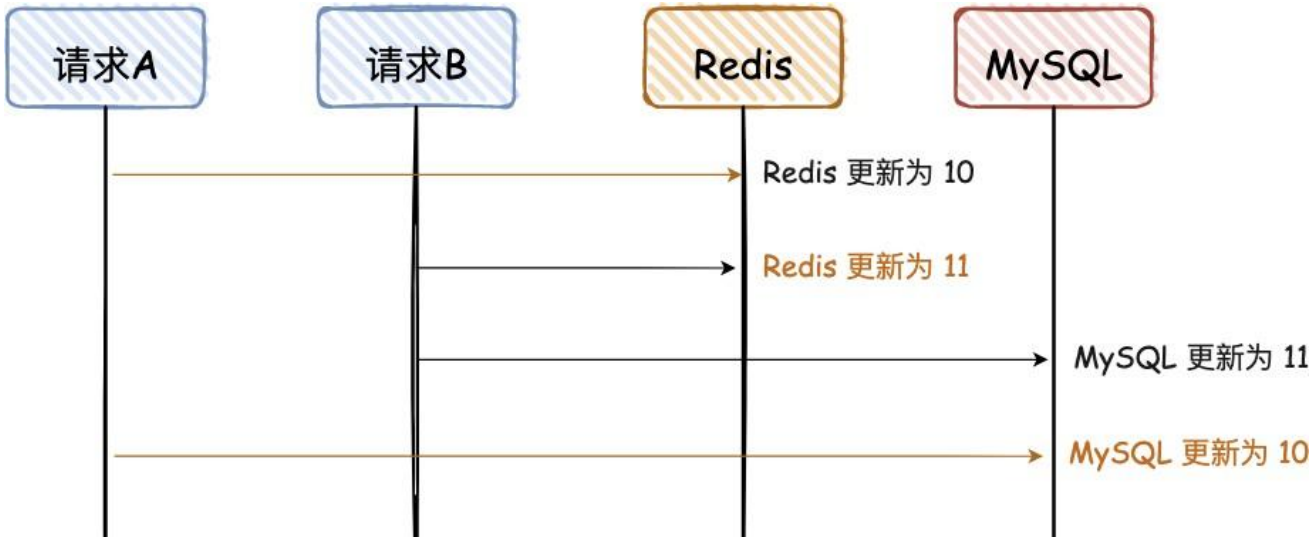



> 图解说明：
>
> - 这是一副时序图，描述请求的先后调用顺序；
> - 橘黄色的线是请求 A，黑色的线是请求 B；
> - 橘黄色的文字，是 MySQL 和 Redis 最终不一致的数据；数
> - 据是从 10 更新为 11；
> - 后面所有的图，都是这个含义，不再赘述。

请求 A、B 都是先写 MySQL，然后再写 Redis，在高并发情况下，如果请求 A 在写 Redis 时卡了一会，请求 B 已经依次完成数据的更新，就会出现图中的问题。

这个图已经画的很清晰了，我就不用再去啰嗦了吧，**不过这里有个前提，就是对于读请求，先去读** Redis，**如果没有，再去读** DB，**但是读请求不会再回写** Redis。 大白话说一下，就是读请求不会更新 Redis。

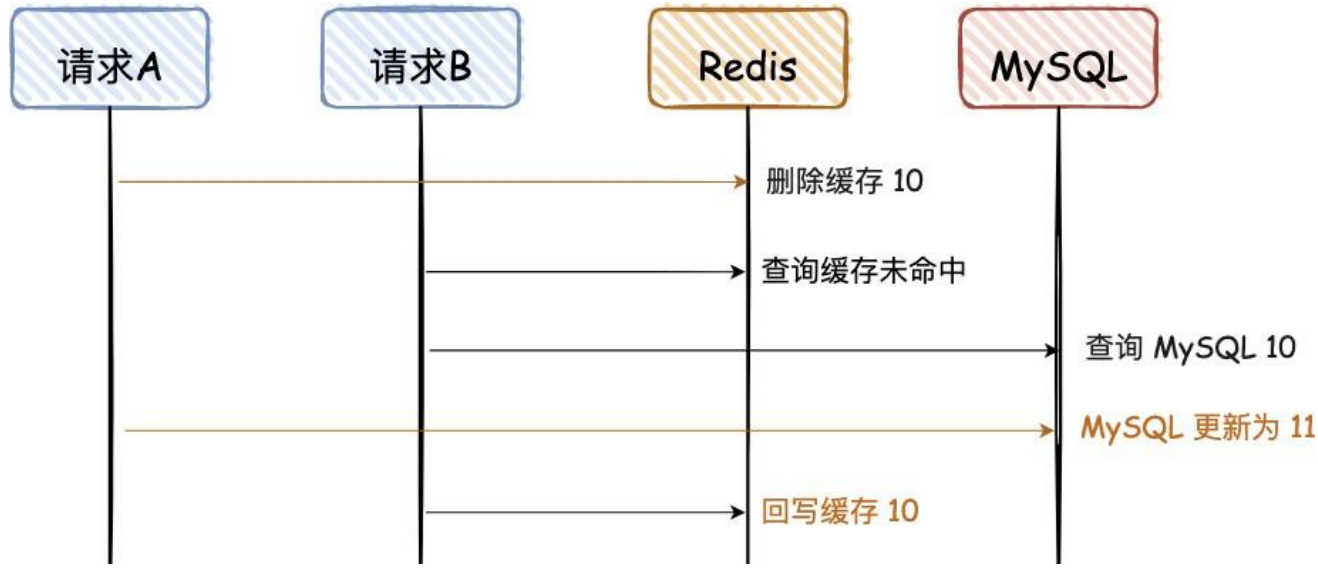
## 2. 先写 Redis，再写 MySQL




同“先写 MySQL，再写 Redis”，看图可秒懂。

## 3. 先删除 Redis，再写 MySQL

这幅图和上面有些不一样，前面的请求 A 和 B 都是更新请求，这里的请求 A 是更新请求，**但是请求 B 是读请求，且请求 B 的读请求会回写** Redis。




请求 A 先删除缓存，可能因为卡顿，数据一直没有更新到 MySQL，导致两者数据不一致。

**这种情况出现的概率比较大，因为请求 A 更新** MySQL **可能耗时会比较长，而请求 B 的前两步都是查询，会非常快。**

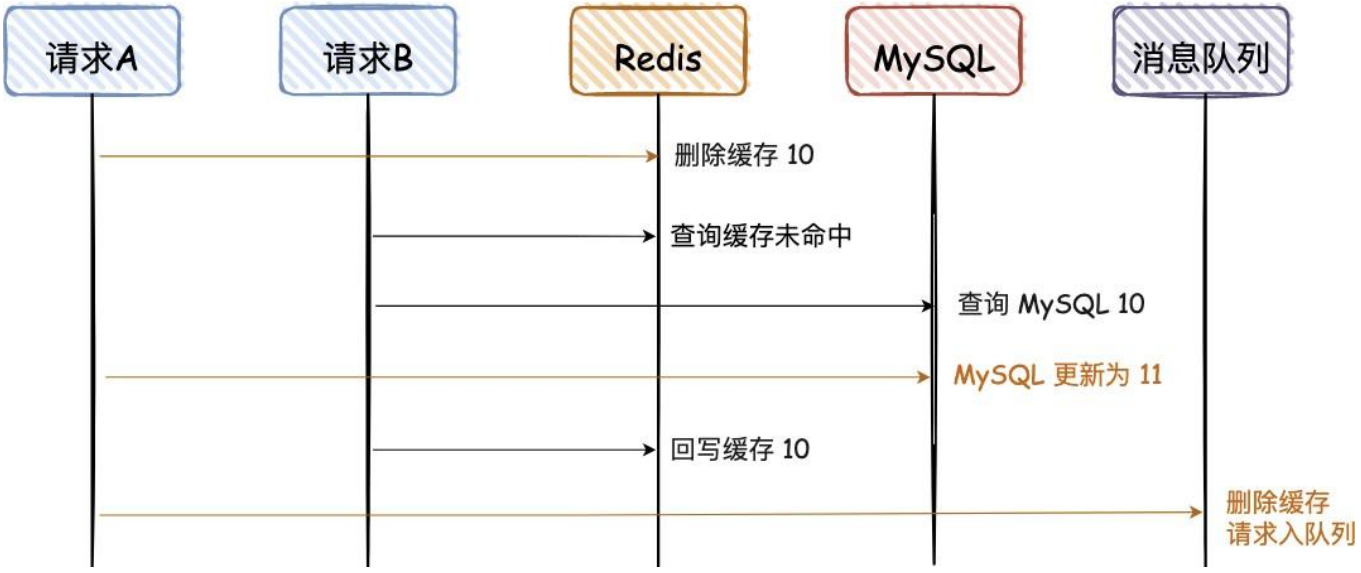
# 好的方案

## 4. 先删除 Redis，再写 MySQL，再删除 Redis

对于“先删除 Redis，再写 MySQL”，如果要解决最后的不一致问题，其实再对 Redis 重新删除即可，**这个也是大家常说的“缓存双删”。**


为了便于大家看图，对于蓝色的文字，“删除缓存 10”必须在“回写缓存10”后面，那如何才能保证一定是在后面呢？

**网上给出的第一个方案是，让请求 A 的最后一次删除，等待** 500ms。 对于这种方案，看看就行，反正我是不会用，太 Low 了，风险也不可控。**那有没有更好的方案呢，我建议异步串行化删除，即删除请求入队列**



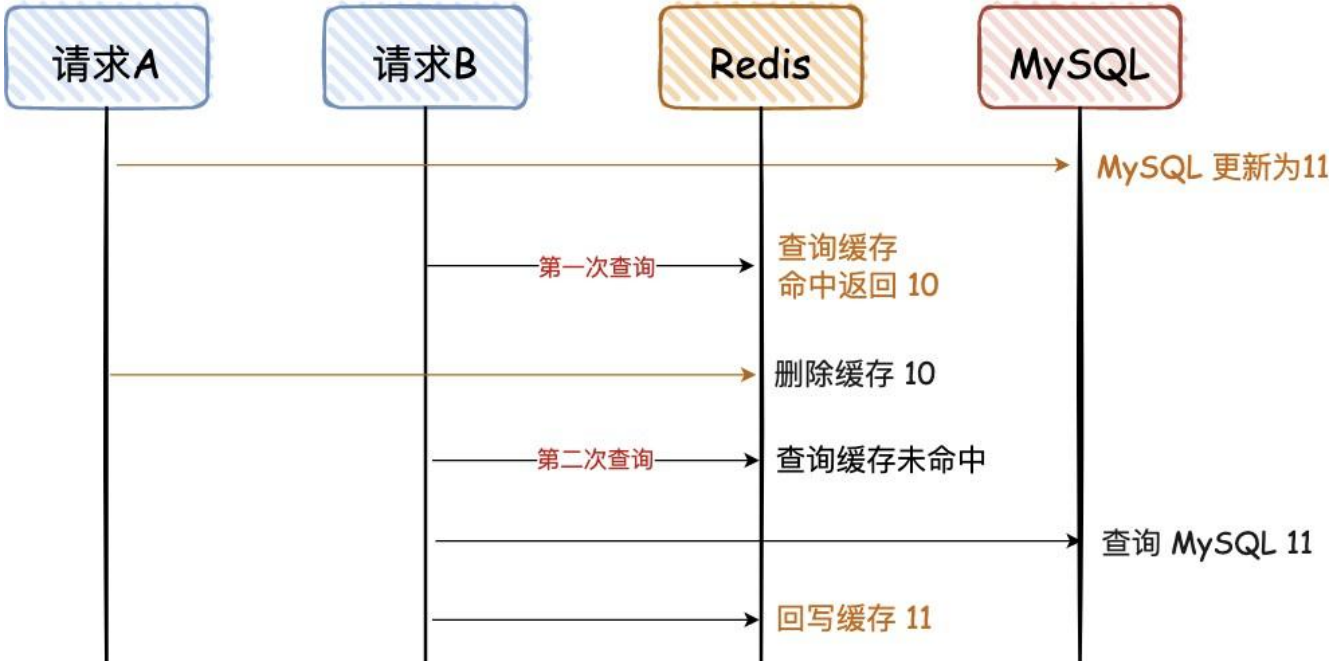

异步删除对线上业务无影响，串行化处理保障并发情况下正确删除。

如果双删失败怎么办，网上有给 Redis 加一个缓存过期时间的方案，这个不敢苟同。**个人建议整个重试机制，可以借助消息队列的重试机制，也可以自己整个表，记录重试次数**，方法很多。

> 简单小结一下：
>
> - “缓存双删”不要用无脑的 sleep 500 ms；

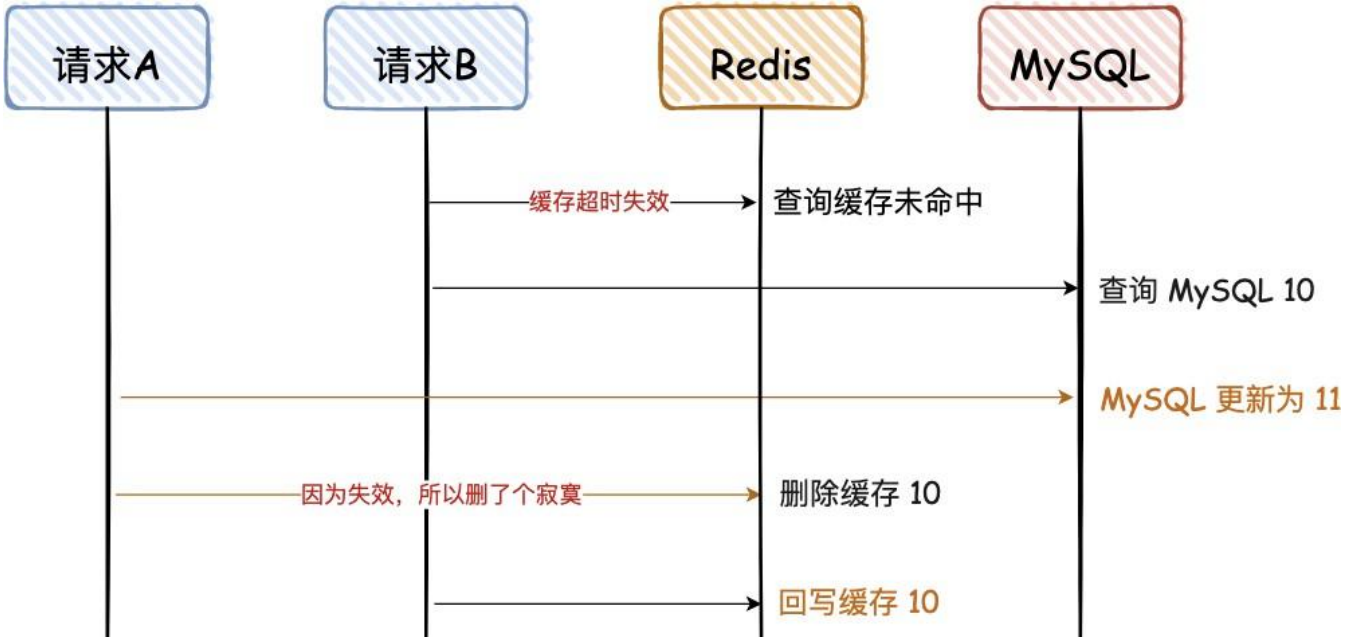
- 通过消息队列的异步&串行，实现最后一次缓存删除；
- 缓存删除失败，增加重试机制。

## 5. 先写 MySQL，再删除 Redis




对于上面这种情况，对于第一次查询，请求 B 查询的数据是 10，但是 MySQL 的数据是 11，**只存在这一次不一致的情况，对于不是强一致性要求的业务，可以容忍。**（那什么情况下不能容忍呢，比如秒杀业务、库存服务等。）

当请求 B 进行第二次查询时，因为没有命中 Redis，会重新查一次 DB，然后再回写到 Reids。




这里需要满足 2 个条件：

- 缓存刚好自动失效；
- 请求 B 从数据库查出 10，回写缓存的耗时，比请求 A 写数据库，并且删除缓存的还长。

对于第二个条件，我们都知道更新 DB 肯定比查询耗时要长，所以出现这个情况的概率很小，同时满足上述条件的情况更小。

## 6. 先写 MySQL，通过 Binlog，异步更新 Redis

这种方案，主要是监听 MySQL 的 Binlog，然后通过异步的方式，将数据更新到 Redis，这种方案有个前提，查询的请求，不会回写 Redis。


这个方案，会保证 MySQL 和 Redis 的最终一致性，但是如果中途请求 B 需要查询数据，如果缓存无数据，就直接查 DB；如果缓存有数据，查询的数据也会存在不一致的情况。

**所以这个方案，是实现最终一致性的终极解决方案，但是不能保证实时性。**

# 几种方案比较

我们对比上面讨论的 6 种方案：

1. 先写 Redis，再写 MySQL

- **这种方案，我肯定不会用**，万一 DB 挂了，你把数据写到缓存，DB 无数据，这个是灾难性的；
- 我之前也见同学这么用过，如果写 DB 失败，对 Redis 进行逆操作，那如果逆操作失败呢，是不是还要搞个重试？

2. 先写 MySQL，再写 Redis

- **对于并发量、一致性要求不高的项目，很多就是这么用的**，我之前也经常这么搞，但是不建议这么做；
- 当 Redis 瞬间不可用的情况，需要报警出来，然后线下处理。

3. 先删除 Redis，再写 MySQL

- 这种方式，我还真没用过，**直接忽略吧。**

4. 先删除 Redis，再写 MySQL，再删除 Redis

- 这种方式虽然可行，但是**感觉好复杂**，还要搞个消息队列去异步删除 Redis。

5. 先写 MySQL，再删除 Redis

- **比较推荐这种方式**，删除 Redis 如果失败，可以再多重试几次，否则报警出来；这个方案，是实时性中最好的方案，在一些高并发场景中，推荐这种。

6. 先写 MySQL，通过 Binlog，异步更新 Redis

- **对于异地容灾、数据汇总等，建议会用这种方式**，比如 binlog + kafka，数据的一致性也可以达到秒级；
- 纯粹的高并发场景，不建议用这种方案，比如抢购、秒杀等。

**个人结论：**

- **实时一致性方案**：采用“先写 MySQL，再删除 Redis”的策略，这种情况虽然也会存在两者不一致，但是需要满足的条件有点苛刻，**所以是满足实时性条件下，能尽量满足一致性的最优解。**
- **最终一致性方案**：采用“先写 MySQL，通过 Binlog，异步更新 Redis”，可以通过 Binlog，结合消息队列异步更新 Redis，**是最终一致性的最优解。**

# 📚 第 3 章：聊聊限流

在电商高并发场景下，我们经常会使用一些常用方法，去应对流量高峰，比如**限流、熔断、降级**，今天我们聊聊限流。

什么是限流呢？限流是限制到达系统的并发请求数量，保证系统能够正常响应部分用户请求，而对于超过限制的流量，则通过拒绝服务的方式保证整体系统的可用性。

根据限流作用范围，可以分为**单机限流和分布式限流**；根据限流方式，又分为**计数器、滑动窗口、漏桶限令牌桶限流**，下面我们对这块详细进行讲解。

# 常用限流方式

## 计数器

计数器是一种最简单限流算法，其原理就是：在一段时间间隔内，对请求进行计数，与阀值进行比较判断是否需要限流，一旦到了时间临界点，将计数器清零。

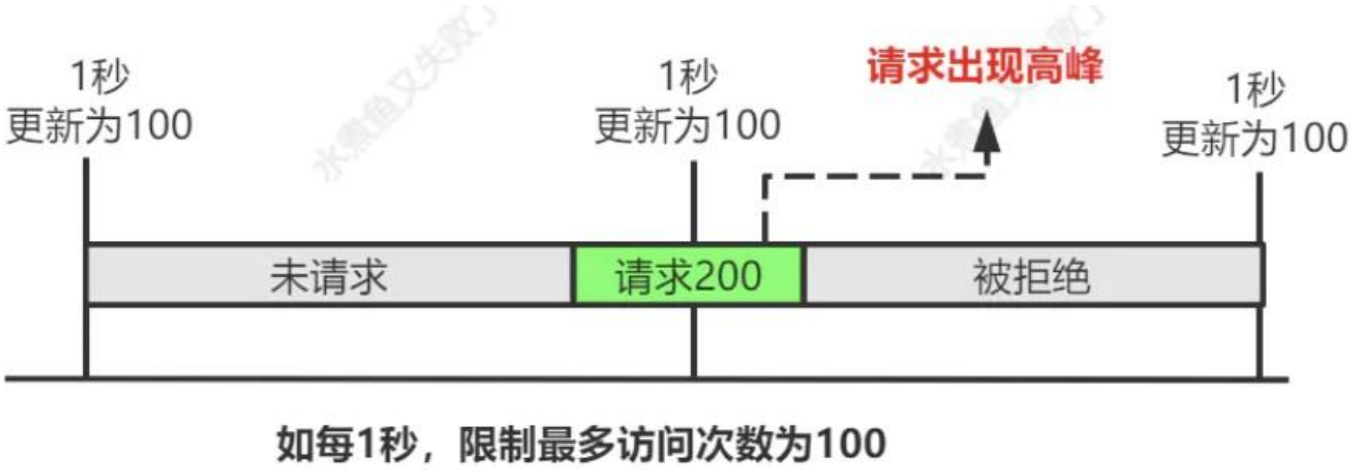
这个就像你去坐车一样，车厢规定了多少个位置，满了就不让上车了，不然就是超载了，被交警叔叔抓到了就要罚款的，如果我们的系统那就不是罚款的事情了，可能直接崩掉了。

程序执行逻辑：

- 可以在程序中设置一个变量 count，当过来一个请求我就将这个数 +1，同时记录请求时间。
- 当下一个请求来的时候判断 count 的计数值是否超过设定的频次，以及当前请求的时间和第一次请求时间是否在 1 分钟内。
- 如果在 1 分钟内并且超过设定的频次则证明请求过多，后面的请求就拒绝掉。
- 如果该请求与第一个请求的间隔时间大于计数周期，且 count 值还在限流范围内，就重置 count。

那么问题来了，如果有个需求对于某个接口 /query 每分钟最多允许访问 200 次，假设有个用户在第 59 秒的最后几毫秒瞬间发送 200 个请求，当 59 秒结束后 Counter 清零了，他在下一秒的时候又发送 200 个请求。

那么在 1 秒钟内这个用户发送了2 倍的请求，这个是符合我们的设计逻辑的，这也是计数器方法的设计缺陷，系统可能会承受恶意用户的大量请求，甚至击穿系统。**这种方法虽然简单，但也有个大问题就是没有很好的处理单位时间的边界。**

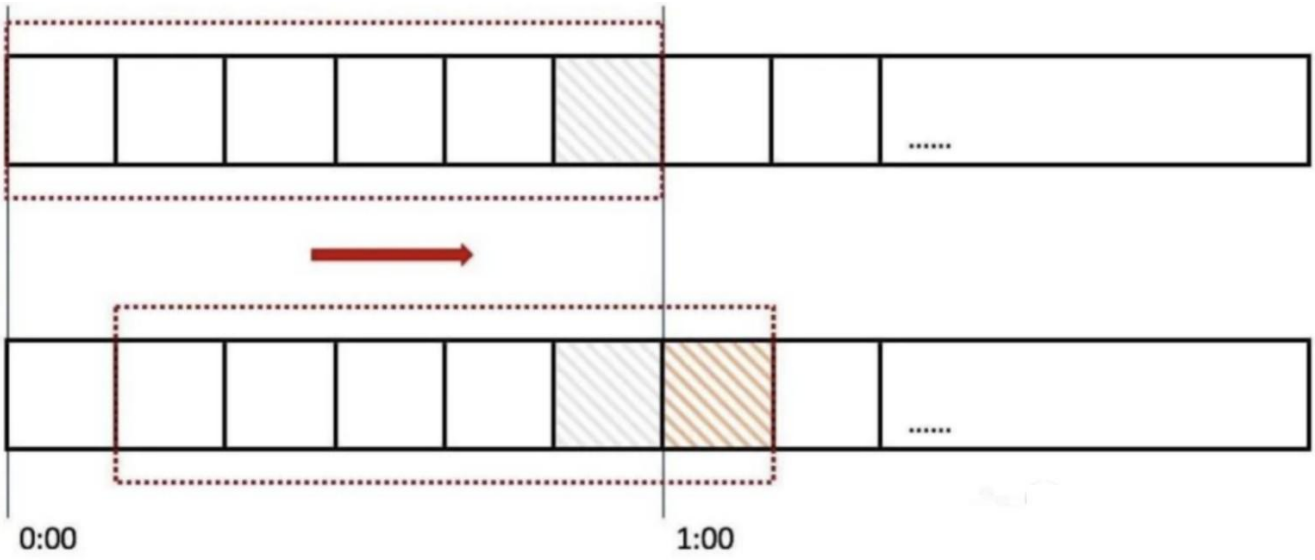



不过说实话，这个计数引用了锁，在高并发场景，这个方式可能不太实用，我建议将锁去掉，然后将 l.count++ 的逻辑通过原子计数处理，这样就可以保证 l.count 自增时不会被多个线程同时执行，**即通过原子计数的方式实现限流。**

> 为了不影响阅读，代码详见：https://github.com/lml200701158/go_demo/blob/master/current_limit/count.go

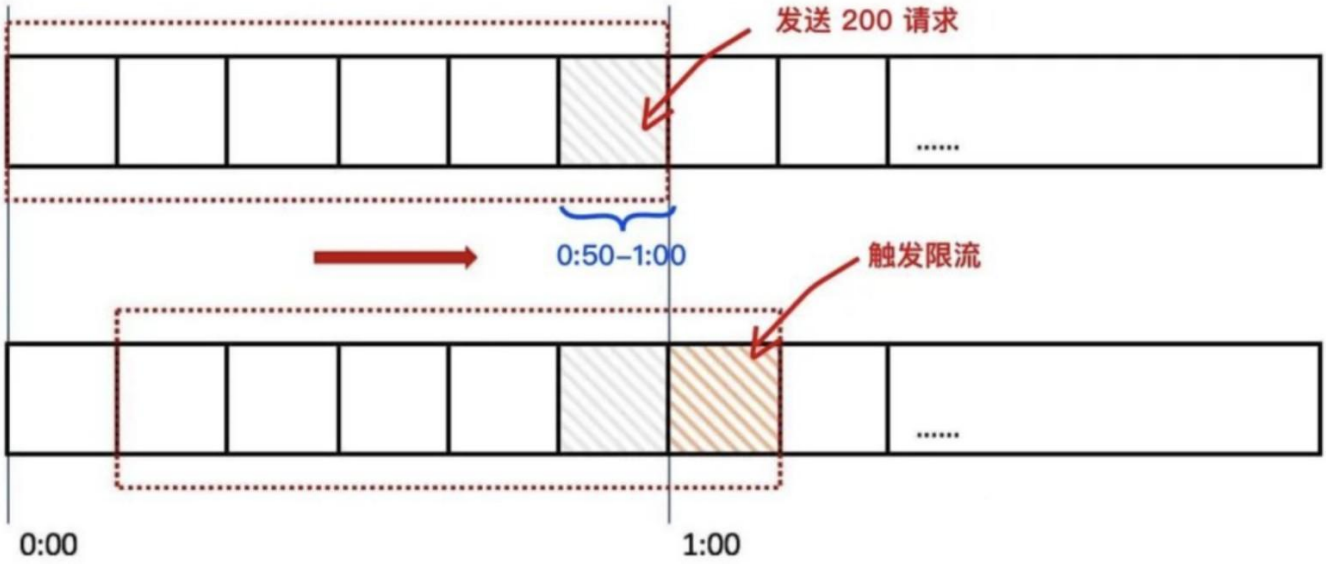
# 滑动窗口

滑动窗口是针对计数器存在的临界点缺陷，所谓滑动窗口（Sliding window）是一种流量控制技术，这个词出现在 TCP 协议中。滑动窗口把固定时间片进行划分，并且随着时间的流逝，进行移动，固定数量的可以移动的格子，进行计数并判断阀值。




上图中我们用红色的虚线代表一个时间窗口（一分钟），每个时间窗口有 6 个格子，每个格子是 10 秒钟。每过 10 秒钟时间窗口向右移动一格，可以看红色箭头的方向。我们为每个格子都设置一个独立的计数器Counter，假如一个请求在 0:45 访问了那么我们将第五个格子的计数器+1（也是就是 0:40~0:50），在判断限流的时候需要把所有格子的计数加起来和设定的频次进行比较即可。

那么滑动窗口如何解决我们上面遇到的问题呢？来看下面的图：




当用户在 0:59 秒钟发送了 200 个请求就会被第六个格子的计数器记录 +200，当下一秒的时候时间窗口向右移动了一个，此时计数器已经记录了该用户发送的 200 个请求，所以再发送的话就会触发限流，则拒绝新的请求。

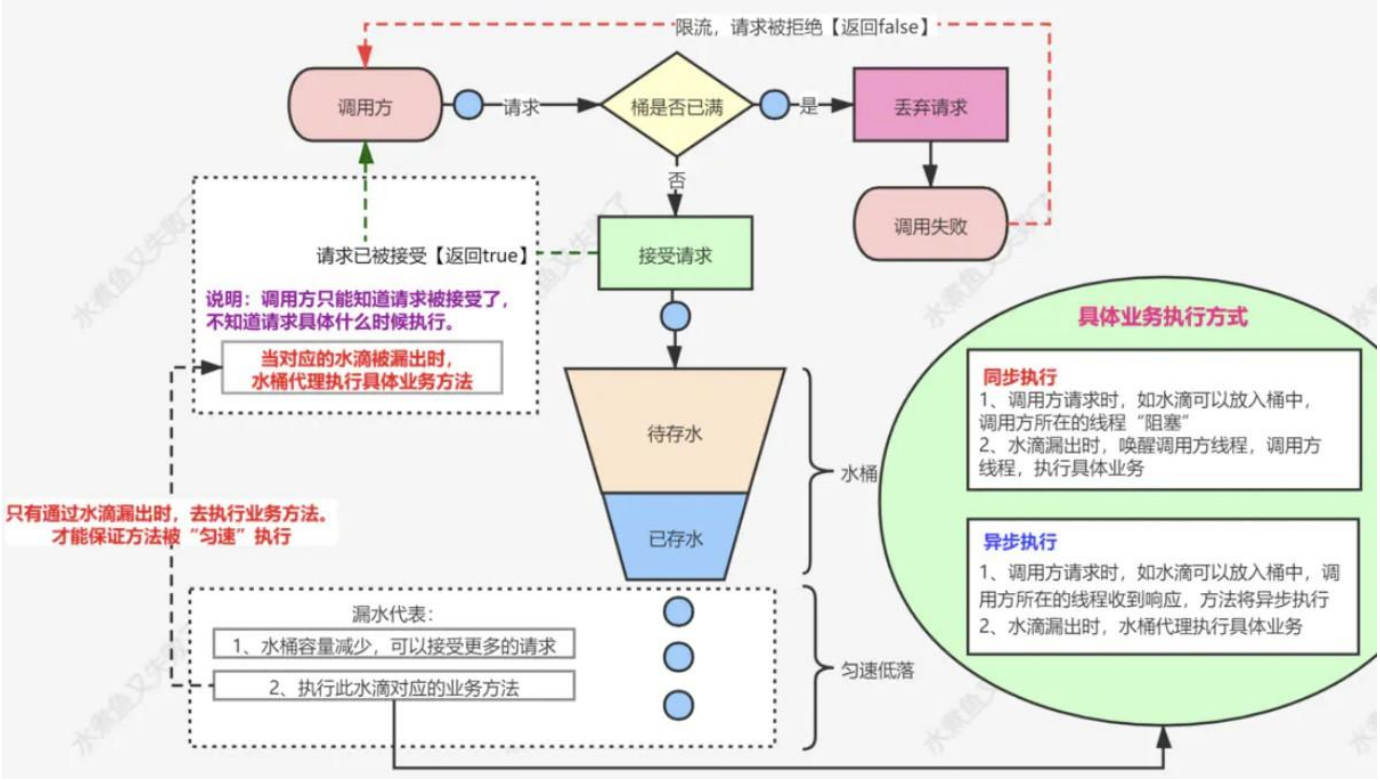
其实计数器就是滑动窗口啊，只不过只有一个格子而已，所以想让限流做的更精确只需要划分更多的格子就可以 了，为了更精确我们也不知道到底该设置多少个格子，**格子的数量影响着滑动窗口算法的精度，依然有时间片的概念，无法根本解决临界点问题。**

> 为了不影响阅读，代码详见：https://github.com/RussellLuo/slidingwindow

# 漏桶

漏桶算法（Leaky Bucket），原理就是一个固定容量的漏桶，按照固定速率流出水滴。

用过水龙头都知道，打开龙头开关水就会流下滴到水桶里，而漏桶指的是水桶下面有个漏洞可以出水,如果水龙头开的特别大那么水流速就会过大，这样就可能导致水桶的水满了然后溢出。




> 图片如果看不清，可单击图片并放大。

一个固定容量的桶，有水流进来，也有水流出去。对于流进来的水来说，我们无法预计一共有多少水会流进来，也无法预计水流的速度。但是对于流出去的水来说，这个桶可以固定水流出的速率（处理速度），从而达到流量整形和流量控制的效果。

漏桶算法有以下特点：

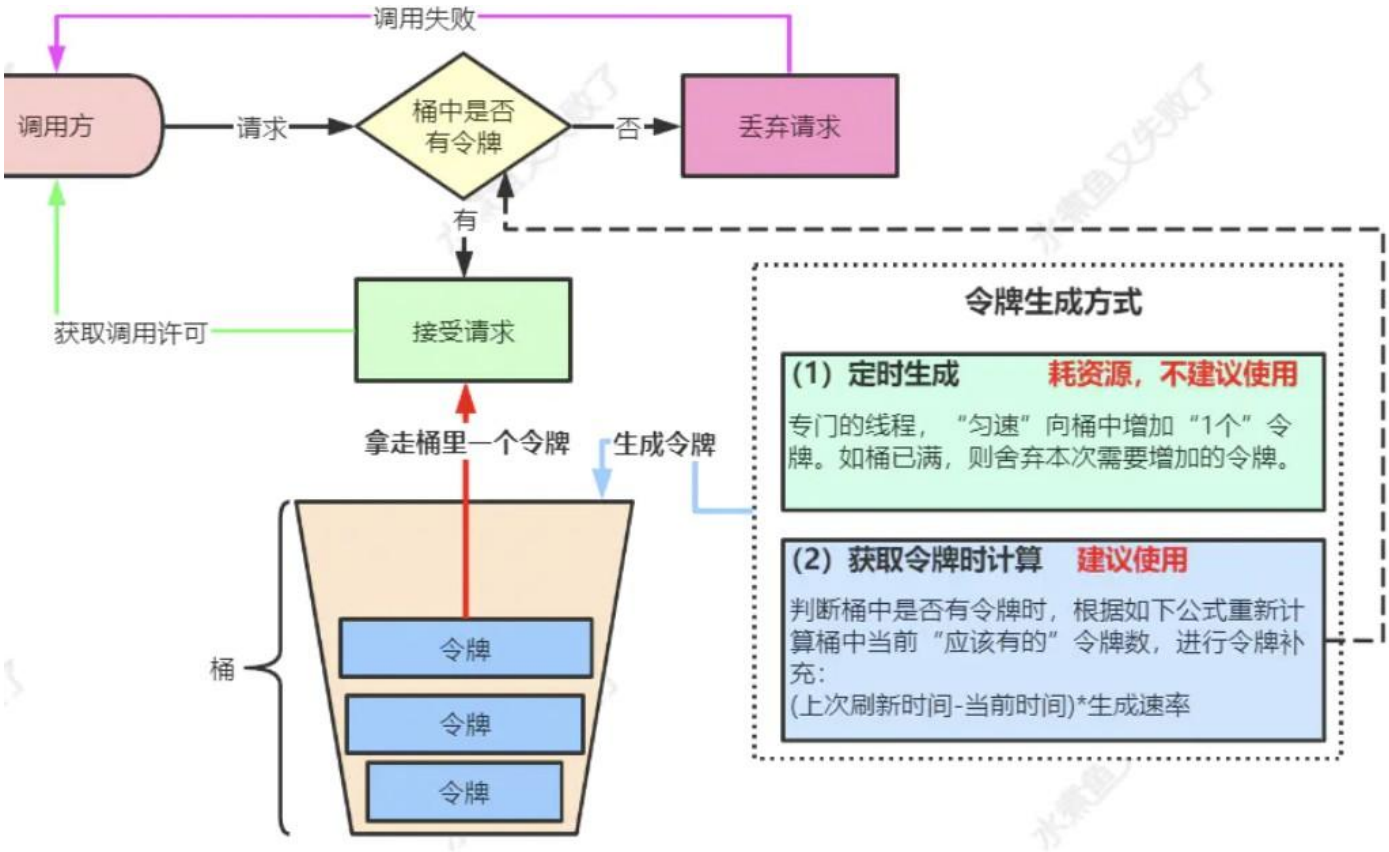
- 漏桶具有固定容量，出水速率是固定常量（流出请求）
- 如果桶是空的，则不需流出水滴
- 可以以任意速率流入水滴到漏桶（流入请求）
- 如果流入水滴超出了桶的容量，则流入的水滴溢出（新请求被拒绝）

漏桶限制的是常量流出速率（即流出速率是一个固定常量值），**所以最大的速率就是出水的速率，不能出现突发流量。**

> 为了不影响阅读，代码详见：https://github.com/lml200701158/go_demo/blob/master/current_limit/leaky_bucket.go

# 令牌桶

令牌桶算法（Token Bucket）是网络流量整形（Traffic Shaping）和速率限制（Rate Limiting）中最常使用的一种算法。典型情况下，令牌桶算法用来控制发送到网络上的数据的数目，并允许突发数据的发送。




> 图片如果看不清，可单击图片并放大。

我们有一个固定的桶，桶里存放着令牌（token）。一开始桶是空的，系统按固定的时间（rate）往桶里添加令 牌，直到桶里的令牌数满，多余的请求会被丢弃。当请求来的时候，从桶里移除一个令牌，如果桶是空的则拒绝请求或者阻塞。

令牌桶有以下特点：

- 令牌按固定的速率被放入令牌桶中
- 桶中最多存放 B 个令牌，当桶满时，新添加的令牌被丢弃或拒绝
- 如果桶中的令牌不足 N 个，则不会删除令牌，且请求将被限流（丢弃或阻塞等待）

**令牌桶限制的是平均流入速率**（允许突发请求，只要有令牌就可以处理，支持一次拿3个令牌，4个令牌...），**并允许一定程度突发流量，所以也是非常常用的限流算法。**

> 为了不影响阅读，代码详见：https://github.com/lml200701158/go_demo/blob/master/current_limit/token_bucket.go

## Redis + Lua 分布式限流

单机版限流仅能保护自身节点，但无法保护应用依赖的各种服务，并且在进行节点扩容、缩容时也无法准确控制整个服务的请求限制。

而分布式限流，以集群为维度，可以方便的控制这个集群的请求限制，从而保护下游依赖的各种服务资源。

**分布式限流最关键的是要将限流服务做成原子化**，我们可以借助 Redis 的计数器，Lua 执行的原子性，进行分布式限流，大致的 Lua 脚本代码如下：

```
local key = "rate.limit:" .. KEYS[1] --限流KEY local limit =
tonumber(ARGV[1])                                    --限流大小
local current = tonumber(redis.call('get', key) or "0") if current + 1 > limit then --如果
超出限流大小
    return 0
else --请求数+1，并设置1秒过期
    redis.call("INCRBY",key,"1")
      redis.call("expire",key,"1") return current + 1
end
```

限流逻辑（Java 语言）：

```
public static boolean accquire() throws IOException, URISyntaxException { Jedis jedis = new Jedis("127.0.0.1");
        File luaFile = new File(RedisLimitRateWithLUA.class.getResource("/").toURI().getPath() + "limit.lua");
        String luaScript = FileUtils.readFileToString(luaFile);

        String key = "ip:" + System.currentTimeMillis()/1000; // 当前秒
        String limit = "5"; // 最大限制
        List<String> keys = new ArrayList<String>(); keys.add(key);
        List<String> args = new ArrayList<String>(); args.add(limit);
        Long result = (Long)(jedis.eval(luaScript, keys, args)); // 执行lua脚本，传入参数
        return result == 1;
}
```

# 聊聊其它

上面的限流方式，主要是针对服务器进行限流，**我们也可以对容器进行限流，比如** Tomcat、Nginx **等限流手段。**

Tomcat 可以设置最大线程数（maxThreads），当并发超过最大线程数会排队等待执行；而 Nginx 提供了两种限流手段：一是控制速率，二是控制并发连接数。

对于 Java 语言，我们其实有相关的限流组件，比如**大家常用的** RateLimiter**，其实就是基于令牌桶算法**，大家知道为什么唯独选用令牌桶么？

对于 Go 语言，也有该语言特定的限流方式，比如可以通过 channel 实现并发控制限流，也支持第三方库 httpserver 实现限流，详见这篇 《Go 限流的常见方法》。

在实际的限流场景中，我们也可以控制单个 IP、城市、渠道、设备 id、用户 id 等在一定时间内发送的请求数；如果是开放平台，需要为每个 appkey 设置独立的访问速率规则。

# 限流对比

下面我们就对常用的线程策略，总结它们的优缺点，便于以后选型。

**计数器**

- 优点：固定时间段计数，实现简单，适用不太精准的场景；
- 缺点：对边界没有很好处理，导致限流不能精准控制。

**滑动窗口：**

- 优点：将固定时间段分块，时间比“计数器”复杂，适用于稍微精准的场景；
- 缺点：实现稍微复杂，还是不能彻底解决“计数器”存在的边界问题。

**漏桶：**

- 优点：可以很好的控制消费频率；
- 缺点：实现稍微复杂，单位时间内，不能多消费，感觉不太灵活。

**令牌桶：**

- 优点：可以解决“漏桶”不能灵活消费的问题，又能避免过渡消费，强烈推荐；
- 缺点：实现稍微复杂，其它缺点没有想到。

Redis + Lua **分布式限流：**

- 优点：支持分布式限流，有效保护下游依赖的服务资源；
- 缺点：依赖 Redis，对边界没有很好处理，导致限流不能精准控制。

# 📚 第 4 章：缓存雪崩、击穿、穿透

今天写的这个主题内容，其实非常基础，但是**作为高并发非常重要的几个场景，绝对绕不开**，估计大家面试时，也经常会遇到。

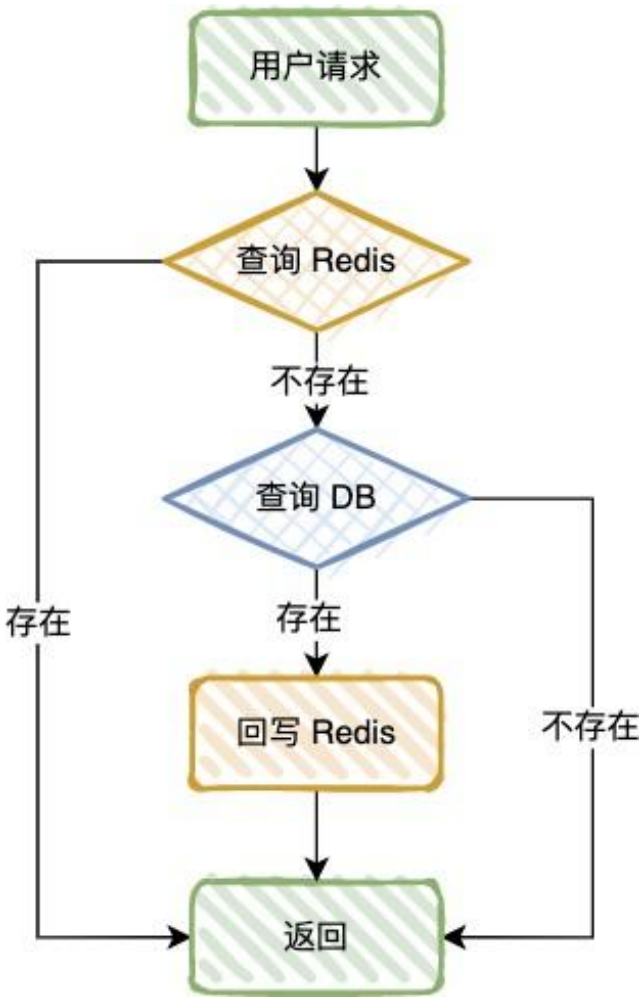
这个主题的文章，网上非常多，本来想直接转载一篇，但是感觉没有合适的，要么文章不够精炼，要么就是精简过头，所以还是自己写一篇吧。

内容虽然基础，但我还是秉承以往的写作风格，参考众多优秀的博客后，打算写一篇能通俗易懂，又不失全面的文章。

## 前言

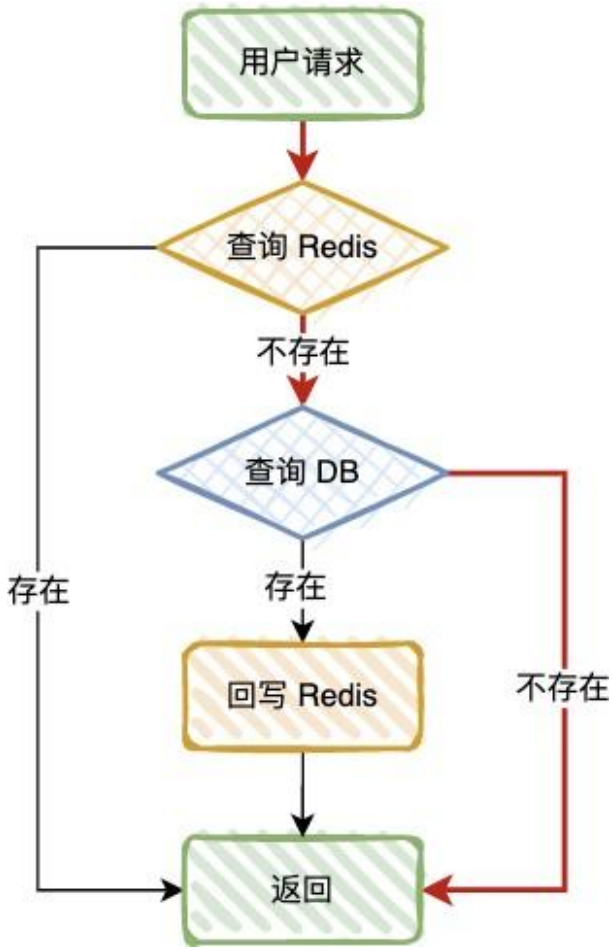
我们先看一下正常情况的查询过程：

- 先查询 Redis，如果查询成功，直接返回，查询不存在，去查询 DB；如
- 果 DB 查询成功，数据回写 Redis，查询不存在，直接返回。




# 缓存穿透

> 定义：当查询数据库和缓存都无数据时，因为数据库查询无数据，出于容错考虑，不会将结果保存到缓存中，因此**每次请求都会去查询数据库**，这种情况就叫做缓存穿透。



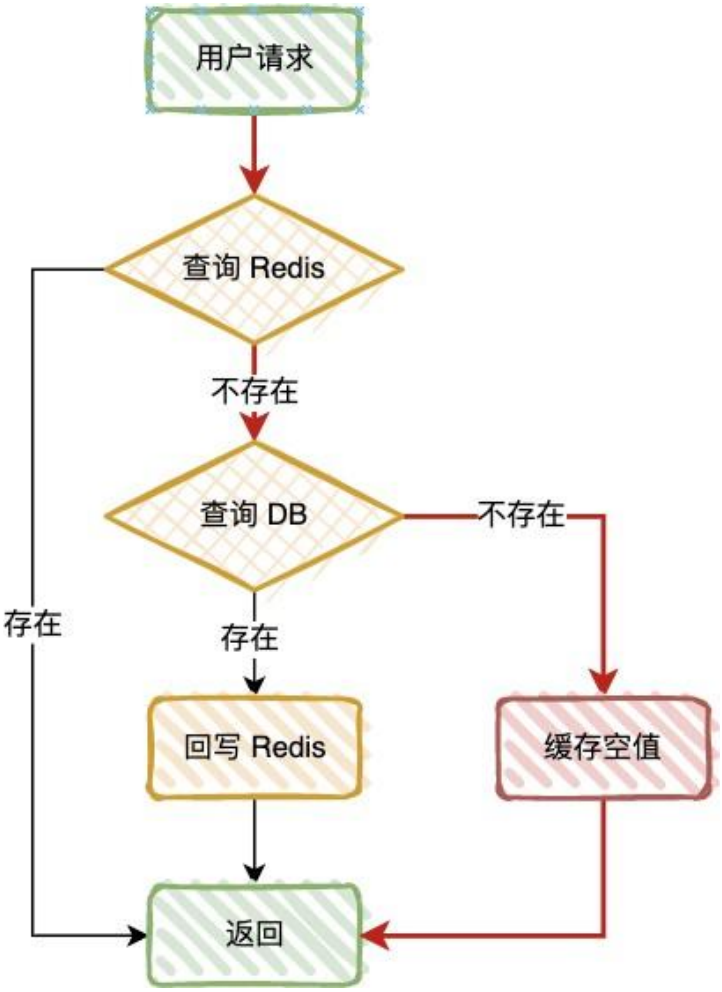

红色的线条，就是缓存穿透的场景，**当查询的 Key 在缓存和 DB 中都不存在时，就会出现这种情况。**

可以想象一下，比如有个接口需要查询商品信息，如果有恶意用户模拟不存在的商品 ID 发起请求，瞬间并发量很高，估计你的 DB 会直接挂掉。

可能大家第一反应就是对入参进行正则校验，过滤掉无效请求，对！这个没错，那有没有其它更好的方案呢？

## 缓存空值

当我们从数据库中查询到空值时，**我们可以向缓存中回种一个空值**，为了避免缓存被长时间占用，需要给这个空值加一个比较短的过期时间，例如 3~5 分钟。



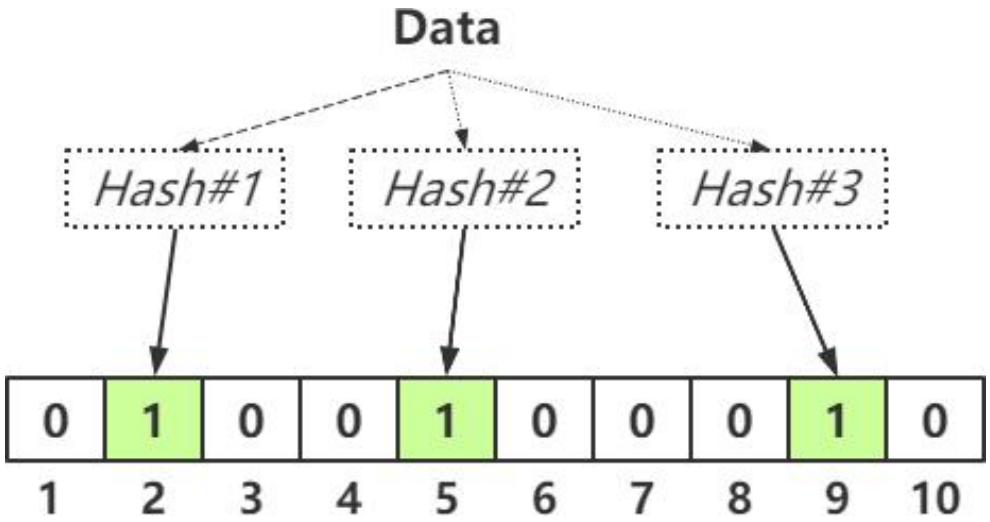

不过这个方案有个问题，当大量无效请求穿透过来时，缓存内就会有有大量的空值缓存，如果缓存空间被占满了，还会因剔除掉一些已经被缓存的用户信息，反而会造成缓存命中率的下降，**所以这个方案，需要评估缓存容量。**

如果缓存空值不可取，这时你可以考虑使用布隆过滤器。

## 布隆过滤器

布隆过滤器是由一个可变长度为 N 的二进制数组与一组数量可变 M 的哈希函数构成，说的简单粗暴一点，就是一个 Hash Map。

原理相当简单：比如元素 key=#3，假如通过 Hash 算法得到一个为 9 的值，就存在这个 Hash Map 的第 9 位元素中，通过标记 1 标识该位已经有数据，如下图所示，0 是无数据，1 是有数据。




所以通过该方法，会得到一个结论：**在 Hash Map 中，标记的数据，不一定存在，但是没有标记的数据，肯定不存在。**

为什么“标记的数据，不一定存在”呢？**因为 Hash 冲突！**

比如 Hash Map 的长度为 100，但是你有 101 个请求，假如你运气好到爆，这 100 个请求刚好均匀打在长度为 100 的 Hash Map 中，此时你的 Hash Map 已经全部标记为 1。

**当第 101 个请求过来时，就 100% 出现 Hash 冲突**，虽然我没有请求过，但是得到的标记却为 1，导致布隆过滤器没有拦截。

如果需要减少误判，可以增加 Hash Map 的长度，并选择却分度更高的 Hash 函数，比如多次对 key 进行hash。除了 Hash 冲突，**布隆过滤器其实会带来一个致命的问题：布隆过滤器更新失败。**

比如有一个商品 ID 第一次请求，当 DB 中存在时，需要在 Hash Map 中标记一下，但是由于网络原因，导致标记失败，那么下次这个商品 ID 重新发起请求时，请求会被布隆过滤器拦截，比如这个是双11的爆款商品库存，明明有 10W 件商品，你却提示库存不存在，领导可能会说“明天你可以不用来了”。

**所以如果使用布隆过滤器，在对 Hash Map 进行数据更新时，需要保证这个数据能 100% 更新成功**，可以通过异步、重试的方式，所以这个方案有一定的实现成本和风险。

# 缓存击穿

> 定义：**某个热点缓存在某一时刻恰好失效**，然后此时刚好有大量的并发请求，此时这些请求将会给数据库造成巨大的压力，这种情况就叫做缓存击穿。

这个其实和“缓存穿透”流程图一样，只是这个的出发点是“某个热点缓存在某一时刻恰好失效”，比如某个非常热门的爆款商品，缓存突然失效，流量直接全部打到 DB，**造成某一时刻数据库请求量过大，更强调瞬时性。**

解决问题的方法主要有 2 种：

1. **分布式锁**：只有拿到锁的第一个线程去请求数据库，然后插入缓存，当然每次拿到锁的时候都要去查询一下缓存有没有，这种在高并发场景下，个人不太建议用分布式锁，会影响查询效率；
2. **设置永不过期**：对于某些热点缓存，我们可以设置永不过期，这样就能保证缓存的稳定性，但需要注意在数据更改之后，要及时更新此热点缓存，不然就会造成查询结果的误差，比如热门商品，都先预热到数据库，后续

再下线掉。

网上还有“缓存续期”的方式，比如缓存 30 分钟失效，可以搞个定时任务，每 20 分钟跑一次，感觉这种方式不伦不类，仅供大家参考。

## 缓存雪崩

> 定义：**在短时间内有大量缓存同时过期，导致大量的请求直接查询数据库**，从而对数据库造成了巨大的压力，严重情况下可能会导致数据库宕机的情况叫做缓存雪崩。

如果说“缓存击穿”是单兵反抗，那“缓存雪崩”就是集体起义了，那什么情况会出现缓存雪崩呢？

1. 短时间内有大量缓存同时过期；
2. 缓存服务宕机，导致某一时刻发生大规模的缓存失效。

那么有哪些解决方案呢？

1. **缓存添加随机时间**：可在设置缓存时添加随机时间，比如 0~60s，这样就可以极大的避免大量的缓存同时失效；
2. **分布式锁**：加一个分布式锁，第一个请求将数据持久化到缓存后，其它的请求才能进入；
3. **限流和降级**：通过限流和降级策略，减少请求的流量；
4. **集群部署**：Redis 通过集群部署、主从策略，主节点宕机后，会切换到从节点，保证服务的可用性。

缓存添加随机时间示例：

```
// 缓存原本的失效时间
int exTime = 10 * 60;
// 随机数生成类
Random random = new Random();
// 缓存设置
jedis.setex(cacheKey, exTime + random.nextInt(1000) , value);
```

# 📚 第 5 章：Redis 高可用原理

Redis 的高可用，太重要啦！之前找工作面试，这个问题面试的频率都能排到前几，尤其是一些大厂，**先不要着急看文章，如果面试官给你抛这么个问题，你会怎么回答呢，可以先想 5 分钟。**

这里要等待 5 分钟 ...

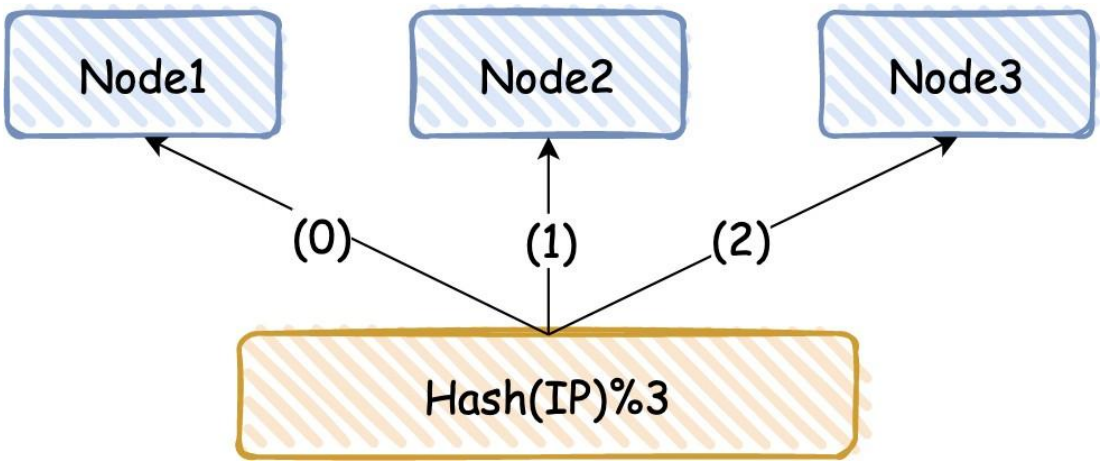
其实我也可以偷个懒，完全转载其它博客，但是没有找到我想要的，**为了不辜负广大粉丝，楼哥还是单独给大家写一篇**，主要根据这块知识，再结合之前的一些面试情况，给大家唠唠。

# 1. Redis 分片策略

## 1.1 Hash 分片

我们都知道，对于 Redis 集群，我们需要通过 hash 策略，将 key 打在 Redis 的不同分片上。假

如我们有 3 台机器，常见的分片方式为 hash(IP)%3，其中 3 是机器总数。



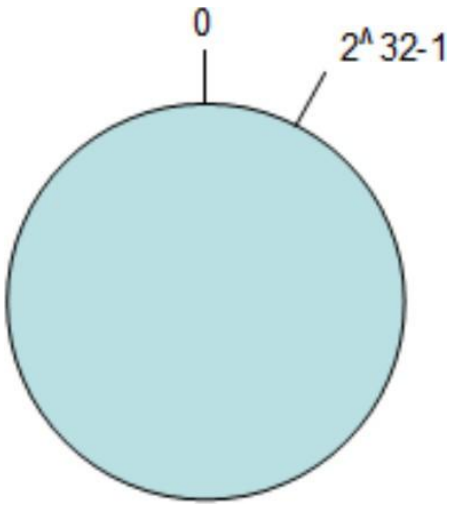

目前很多小公司都这么玩，上手快，简单粗暴，但是这种方式有一个致命的缺点：**当增加或者减少缓存节点时，总节点个数发生变化，导致分片值发生改变，需要对缓存数据做迁移。**

那如何解决该问题呢，答案是一致性 Hash。

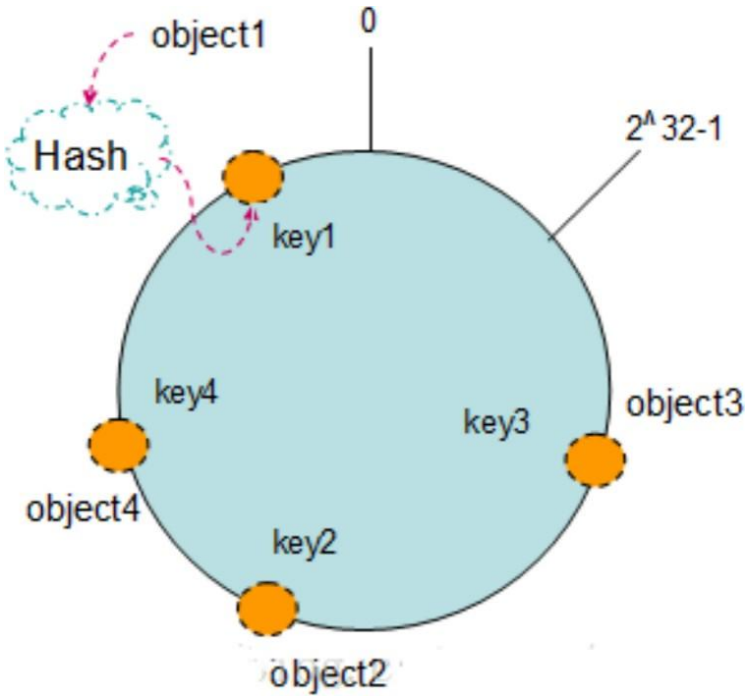
## 1.2 一致性 Hash

一致性哈希算法是 1997 年由麻省理工学院提出的一种分布式哈希实现算法。

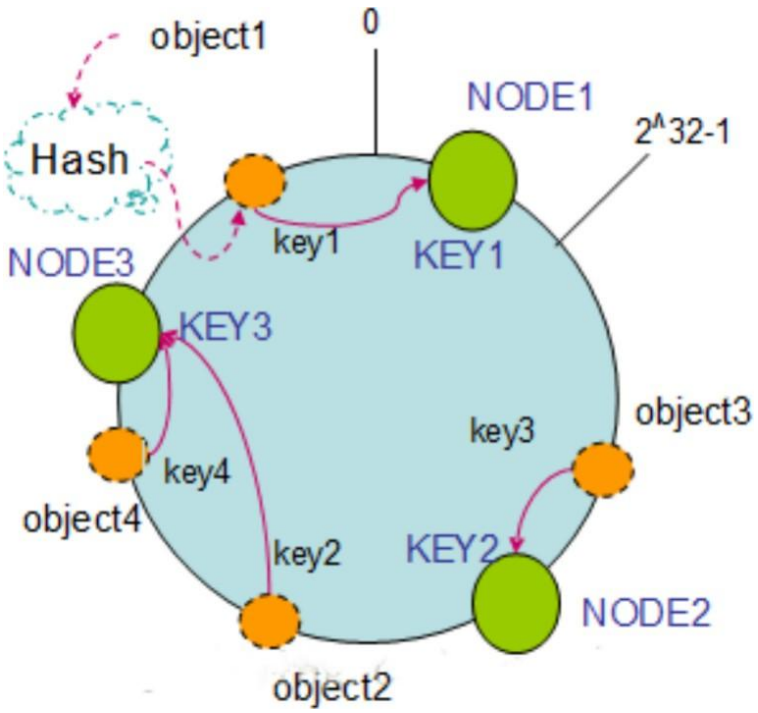
**环形空间：**按照常用的 hash 算法来将对应的 key 哈希到一个具有 2^32 次方个桶的空间中，即 0~(2^32)-1 的数字空间中，现在我们可以将这些数字头尾相连，想象成一个闭合的环形。



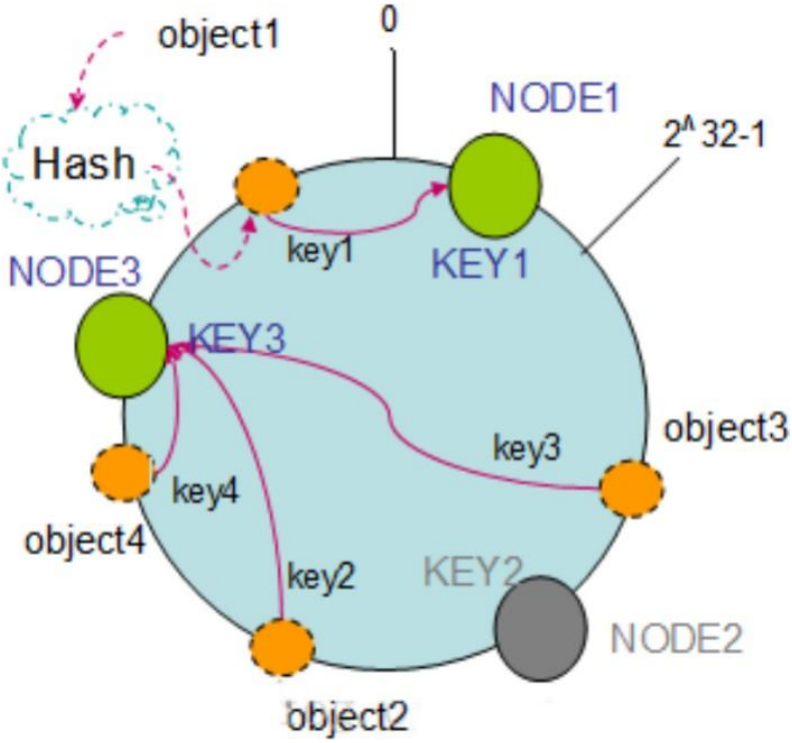

Key **散列** Hash **环**：现在我们将 object1、object2、object3、object4 四个对象通过特定的 Hash 函数计算出对应的 key 值，然后散列到 Hash 环上。



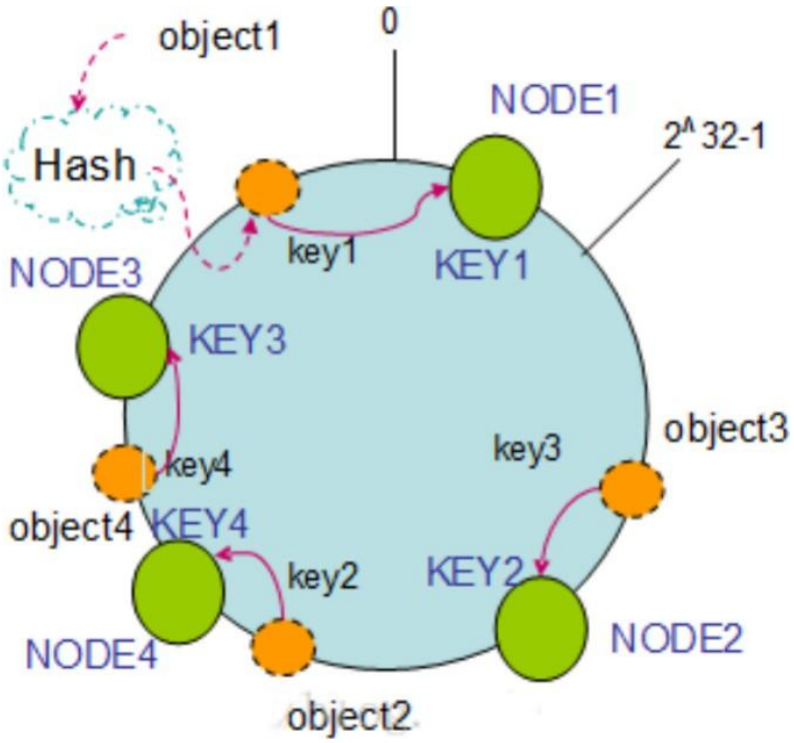

**机器散列** Hash **环**：假设现在有 NODE1、NODE2、NODE3 三台机器，以顺时针的方向计算，将所有对象存储到离自己最近的机器中，object1 存储到了NODE1，object3 存储到了NODE2，object2、object4 存储到了NODE3。




**节点删除**：如果 NODE2 出现故障被删除了，object3 将会被迁移到 NODE3 中，这样仅仅是 object3 的映射位置发生了变化，其它的对象没有任何的改动。




**添加节点**：如果往集群中添加一个新的节点 NODE4，object2 被迁移到了 NODE4 中，其它对象保持不变。




通过对节点的添加和删除的分析，**一致性哈希算法在保持了单调性的同时，还使数据的迁移达到了最小，这样的算法对分布式集群来说是非常合适的，避免了大量数据迁移，减小了服务器的的压力。**

> 如果机器个数太少，为了避免大量数据集中在几台机器，实现平衡性，可以建立虚拟节点（比如一台机器建立3-4 个虚拟节点），然后对虚拟节点进行Hash。

# 2. 高可用方案

很多时候，公司只给我们提供一套 Redis 集群，至于如何计算分片，我们一般有 2 套成熟的解决方案。

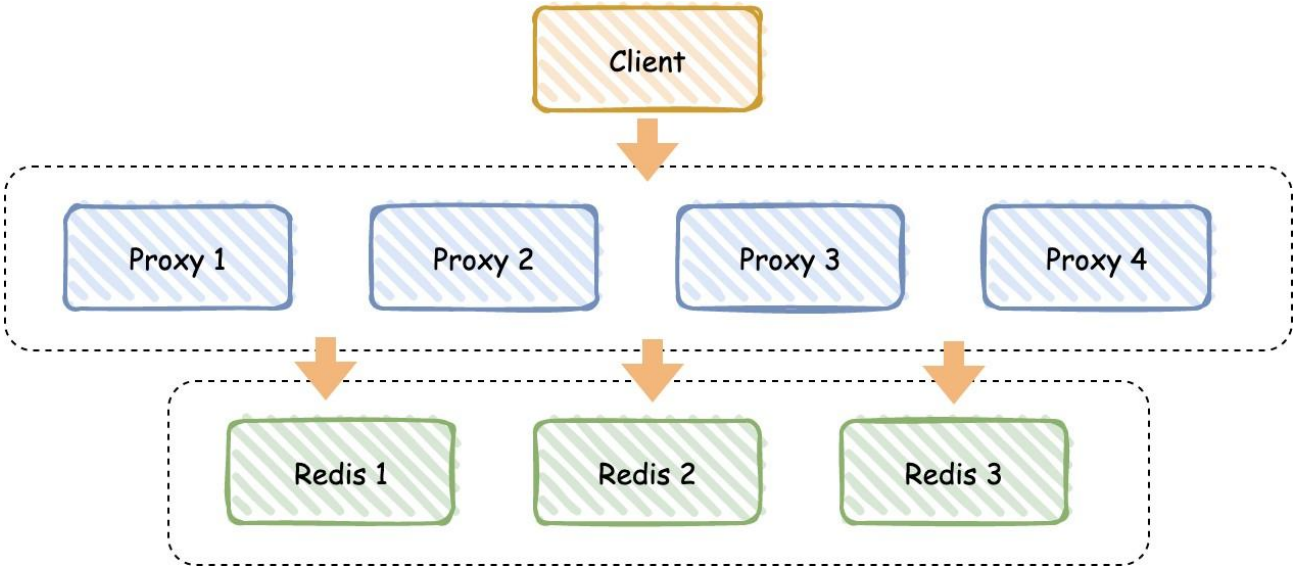
**客户端方案**：也就是客户端自己计算 Redis 分片，无论你使用Hash 分片，还是一致性 Hash，都是由客户端自己完成。

客户端方案简单粗暴，但是只能在单一语言系统之间复用，如果你使用的是 PHP 的系统，后来 Java 也需要使用，你需要用 Java 重新写一套分片逻辑。

**为了解决多语言、不同平台复用的问题**，就衍生出中间代理层方案。

**中间代理层方案**：将客户端解决方案的经验移植到代理层中，通过通用的协议（如 Redis 协议）来实现在其他语言中的复用，**用户无需关心缓存的高可用如何实现，只需要依赖你的代理层即可。**

代理层主要负责读写请求的路由功能，并且在其中内置了一些高可用的逻辑。

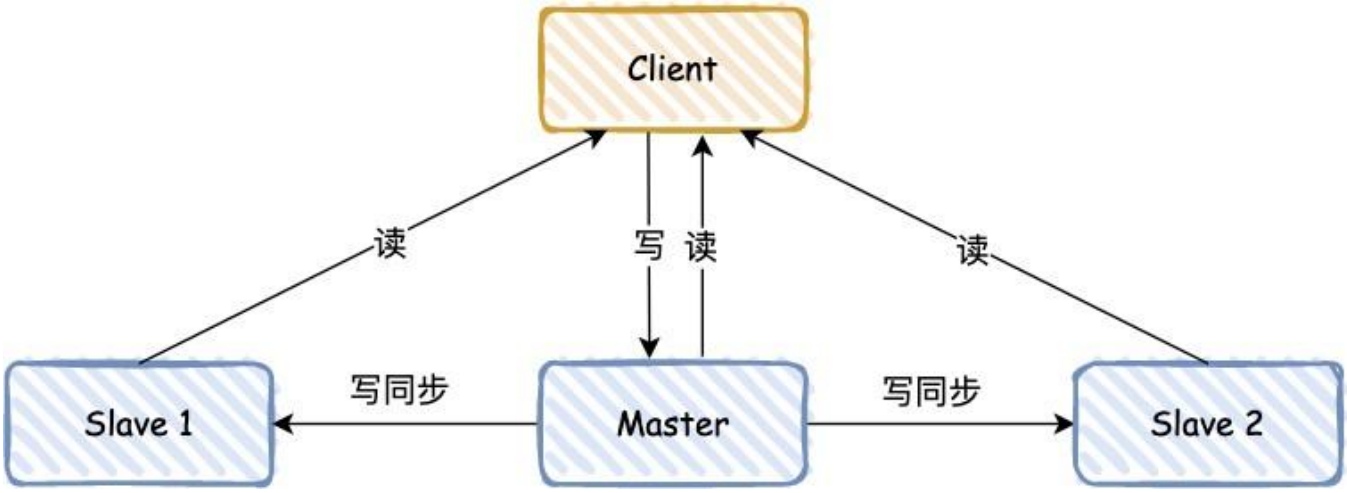



> 你可以看看，你们公司的 Redis 使用的是哪种方案呢？对于“客户端方案”，其实有的也不用自己去写，比如负责维护 Redis 的部门会提供不同语言的 SDK，你只需要去集成对应的 SDK 即可。

# 3. 高可用原理

## 3.1 Redis 主从

Redis 基本都通过“主 - 从”模式进行部署，**主从库之间采用的是读写分离的方式。**



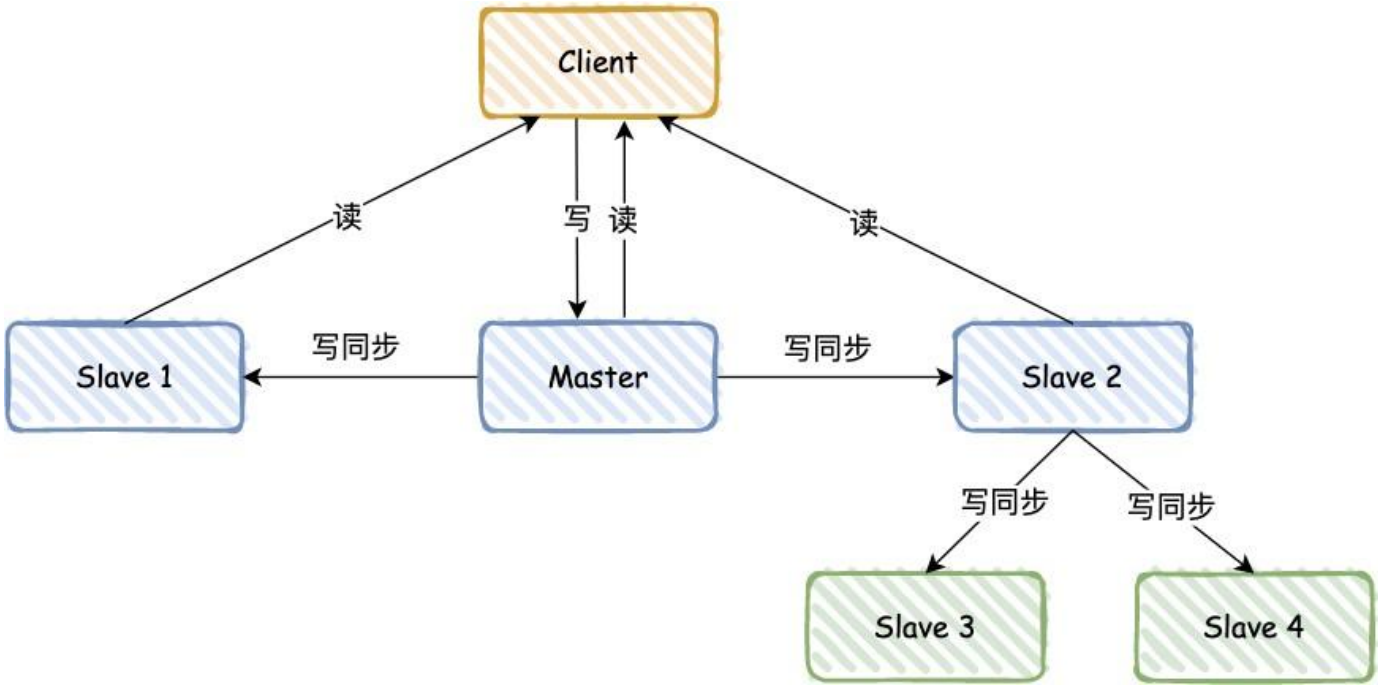

同 MySQL 类似，**主库支持写和读，从库只支持读，数据会先写到主库，然后定时同步给从库**，具体的同步规则，主要将 RDB 日志从主库同步给从库，然后从库读取 RDB 日志，这里比较复杂，其中还涉及到 replication buffer，就不再展开。

这里有个问题，一次同步过程中，主库需要完成 2 个耗时操作：生成 RDB 文件和传输 RDB 文件。

如果从库数量过多，主库忙于 fock 子进程生成 RDB 文件和数据同步，会阻塞主库正常请求。

这个如何解决呢？**答案是 “主 - 从 - 从” 模式。**

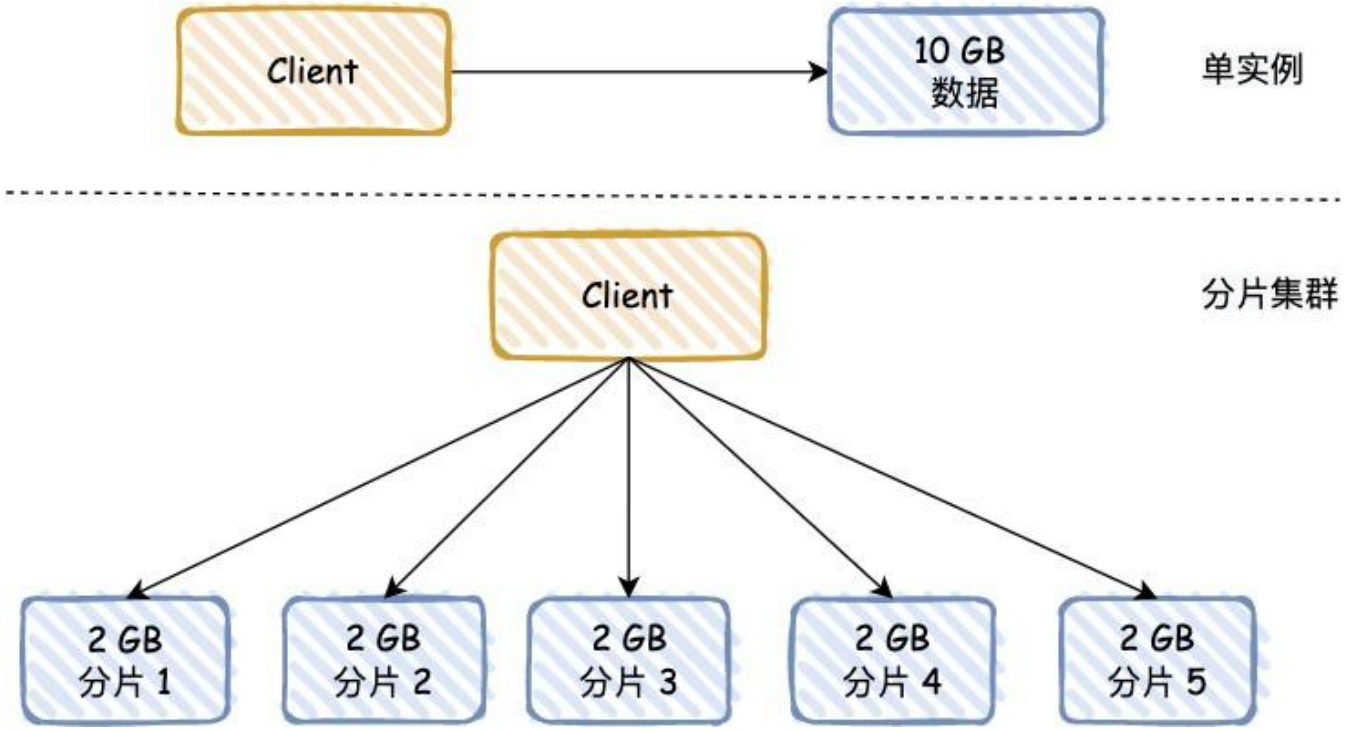
为了避免所有从库都从主库同步 RDB 日志，可以借助从库来完成同步：比如新增 3、4 两个 Slave，可以**等** Slave 2 **同步完后，再通过** Slave 2 **同步给** Slave 3 **和** Slave 4**。**




如果我是面试官，我可能会继续问，如果数据同步了80%，网络突然中断，当网络后续又恢复后，Redis 会如何操作呢？

## 3.2 Redis 分片

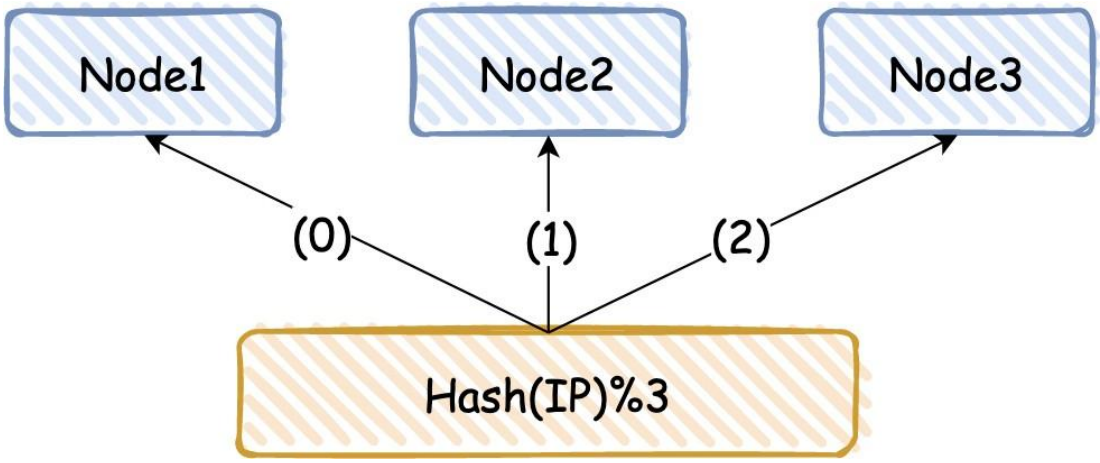
这个有点像 MySQL 分库分表，将数据存储到不同的地方，避免查询时全部集中到一个实例。




其实还有一个好处，就是数据进行主从同步时，如果 RDB 数据过大，会严重阻塞主线程，如果用分片的方式，可以将数据分摊，比如原来有 10 GB 的数据，分摊后，每个分片只有 2 GB。

可能有同学会问，Redis 分片，和“主 - 从”模式有啥关系呢？ 你可以理解，**图中的每个分片都是主库，每个分片都有自己的“主 - 从”模式结构。**

那么数据如何找到对应的分片呢，前面其实已经讲过，假如我们有 3 台机器，常见的分片方式为 hash(IP)%3，其中 3 是机器总数，hash 值为机器 IP，这样每台机器就有自己的分片号。

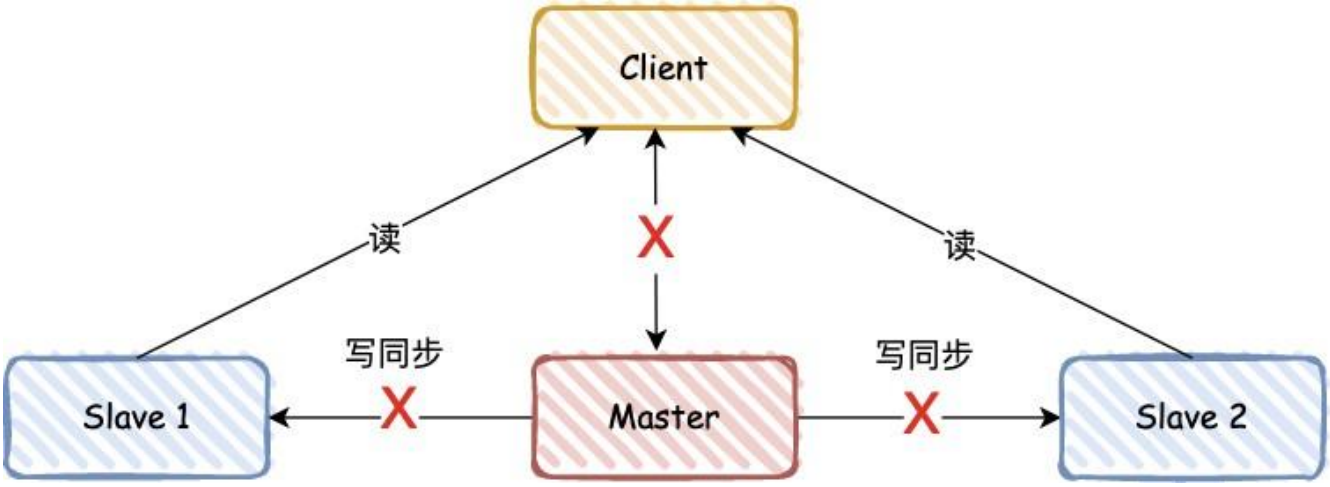



对于 key，也可以采用同样的方式，找到对应的机器分片号 hash(key)%3，hash 算法有很多，可以用 CRC16(key)，也可以直接取 key 中的字符，通过 ASCII 码转换成数字。

## 3.3 Redis 哨兵机制

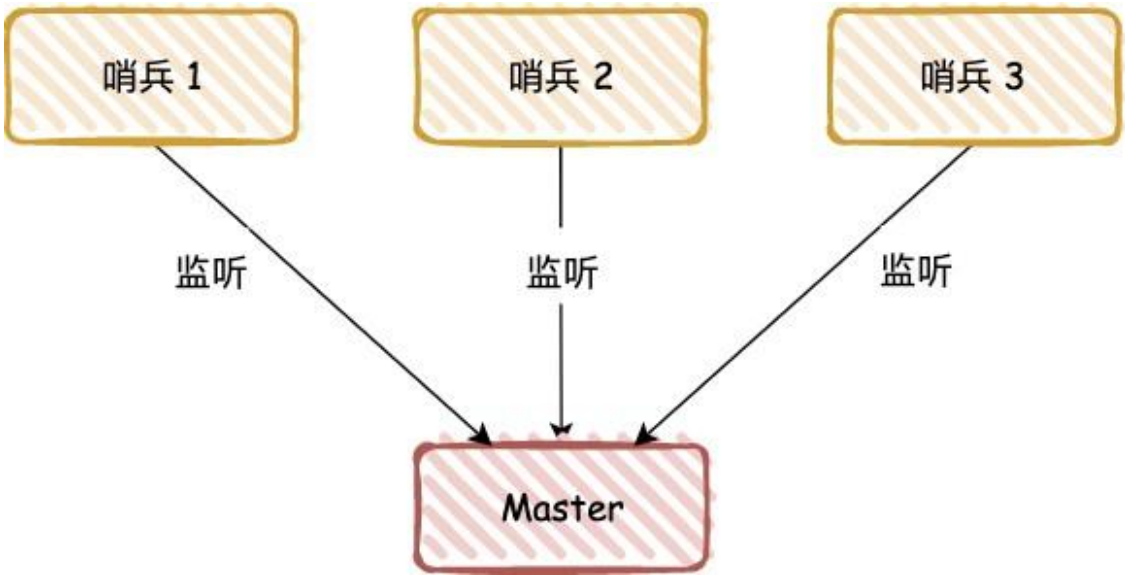
### 3.3.1 什么是哨兵机制 ？

在主从模式下，如果 master 宕机了，从库不能从主库同步数据，主库也不能提供读写功能。




**怎么办呢 ？这时就需要引入哨兵机制 ！**

**哨兵节点是特殊的 Redis 服务，不提供读写服务，主要用来监控 Redis 实例节点。**



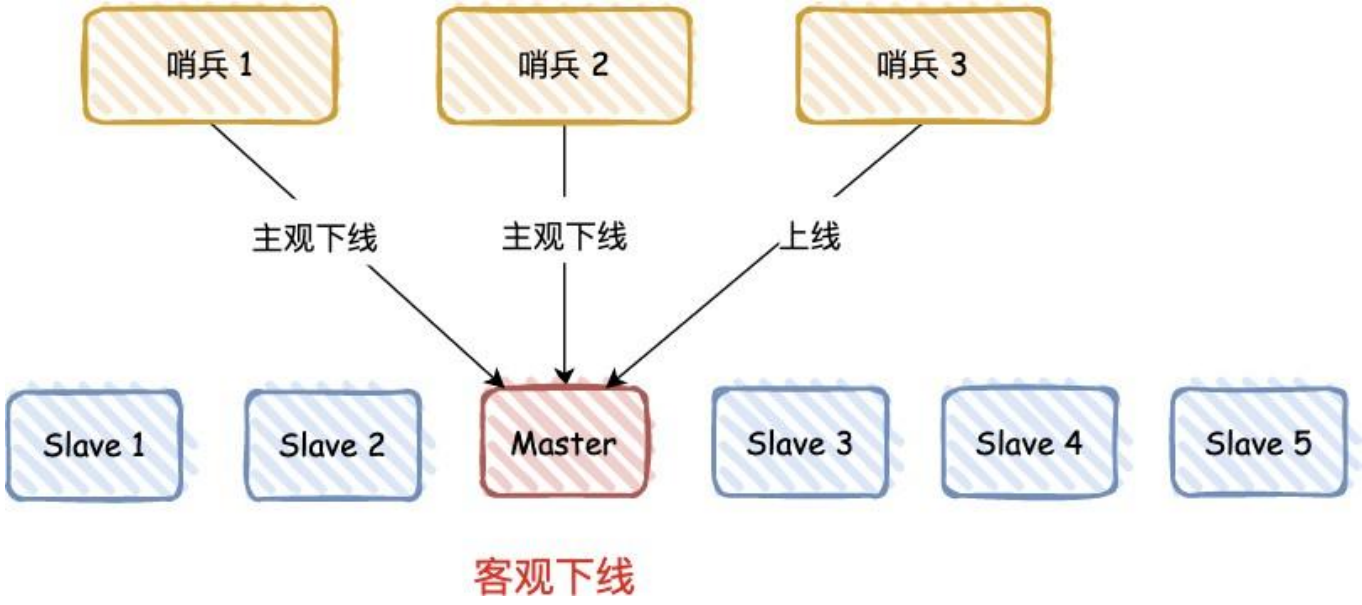

那么当 master 宕机，哨兵如何执行呢？

### 3.3.2 判断主机下线

哨兵进程会使用 PING 命令检测它自己和主、从库的网络连接情况，用来判断实例的状态，如果哨兵发现主库或从库对 PING 命令的响应超时了，哨兵就会先把它标记为“主观下线”。

那是否一个哨兵判断为“主观下线”，就直接下线 master 呢？

答案肯定是不行的，需要遵循 “少数服从多数” 原则：**有 N/2+1 个实例判断主库“主观下线”，才判定主库为“客观下线”。**




比如上图有 3 个哨兵，有 2 个判断 “主观下线”，那么就标记主库为 “客观下线”。

### 3.3.3 选取新主库

我们有 5 个从库，需要选取一个最优的从库作为主库，分 2 步：

- **筛选**：检查从库的当前在线状态和之前的网络连接状态，过滤不适合的从库；
- **打分**：根据从库优先级、和旧主库的数据同步接近度进行打分，选最高分作为主库。


**如果分数一致怎么办？** Redis 也有一个策略：ID 号最小的从库得分最高，会被选为新主库。

当 slave 3 选举为新主库后，**会通知其它从库和客户端，对外宣布自己是新主库**，大家都得听我的哈！今天就讲这么多，我们下期见，大家都学废了么？

# 第 6 章：分库分表

> 这篇文章是转载过来的
>
> 原文作者：捡田螺的小男孩

> 原文链接：https://mp.weixin.qq.com/s/JmgRkVM5J6Wj8lbqH6iX1g

今天跟大家聊聊分库分表。

1. 什么是分库分表
2. 为什么需要分库分表
3. 如何分库分表
4. 什么时候开始考虑分库分表
5. 分库分表会导致哪些问题
6. 分库分表中间件简介

# 1. 什么是分库分表

**分库**：就是一个数据库分成多个数据库，部署到不同机器。

**分表**：就是一个数据库表分成多个表。

# 2. 为什么需要分库分表

## 2.1 为什么需要分库呢？

如果业务量剧增，数据库可能会出现性能瓶颈，这时候我们就需要考虑拆分数据库。从这几方面来看：

- **磁盘存储**

业务量剧增，MySQL单机磁盘容量会撑爆，拆成多个数据库，磁盘使用率大大降低。

- **并发连接支撑**

我们知道数据库连接是有限的。在高并发的场景下，大量请求访问数据库，MySQL单机是扛不住的！当前非常火的**微服务架构**出现，就是为了应对高并发。它把**订单、用户、商品**等不同模块，拆分成多个应用，并且把单个数据库也拆分成多个不同功能模块的数据库（**订单库、用户库、商品库**），以分担读写压力。

## 2.2 为什么需要分表？

数据量太大的话，SQL的查询就会变慢。如果一个查询SQL**没命中索引**，千百万数据量级别的表可能会拖垮整个数据库。

即使SQL命中了索引，如果表的数据量**超过一千万**的话，**查询也是会明显变慢的**。这是因为索引一般是B+树结构，数据千万级别的话，B+树的高度会增高，查询就变慢啦。

**小伙伴们是否还记得，MySQL的B+树的高度怎么计算的呢？** 顺便复习一下吧

InnoDB存储引擎最小储存单元是页，一页大小就是16k。B+树叶子存的是数据，内部节点存的是键值+指针。索引组织表通过非叶子节点的二分查找法以及指针确定数据在哪个页中，进而再去数据页中找到需要的数据，B+树结构图如下：

假设B+树的高度为2的话，即有一个根结点和若干个叶子结点。这棵B+树的存放总记录数为=根结点指针数*单个叶子节点记录行数。

- 如果一行记录的数据大小为1k，那么单个叶子节点可以存的记录数 `=16k/1k =16.`
- 非叶子节点内存放多少指针呢？我们假设主键ID为bigint**类型，长度为**8**字节(面试官问你**int**类型，一个**int**就**

是32位，4字节)，而指针大小在InnoDB源码中设置为6字节，所以就是 8+6=14字节， 16k/14B =16*1024B/14B = 1170

因此，一棵高度为2的B+树，能存放 1170 * 16=18720条这样的数据记录。同理一棵高度为 3的B+树，能存放 1170 *1170 *16 =21902400，大概可以存放两千万左右的记录。B+树高度一般为1-3层，如果B+到了4层，查询的时候会**多查磁盘**的次数，SQL就会变慢。

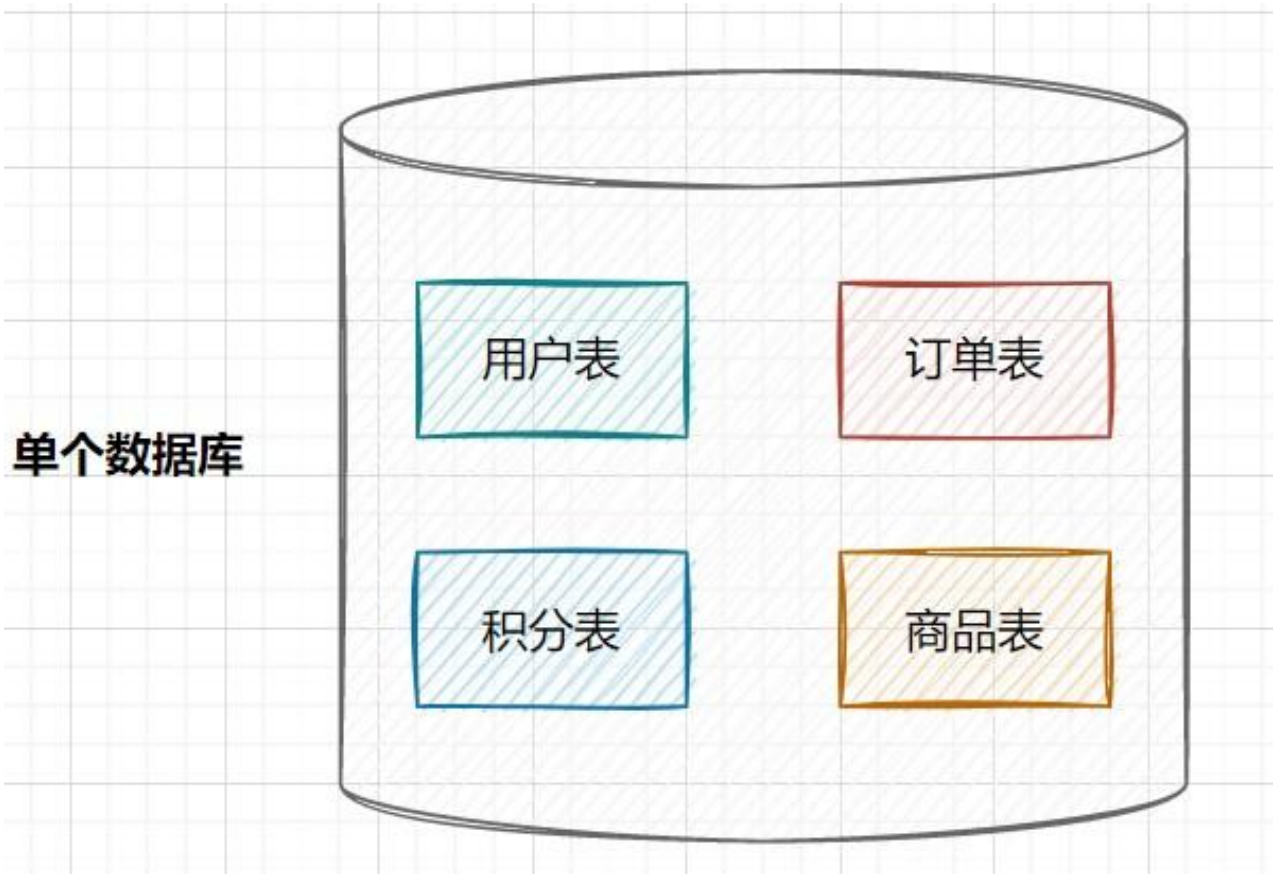
因此单表数据量太大，SQL查询会变慢，所以就需要考虑分表啦。

# 3. 如何分库分表

## 3.1 垂直拆分

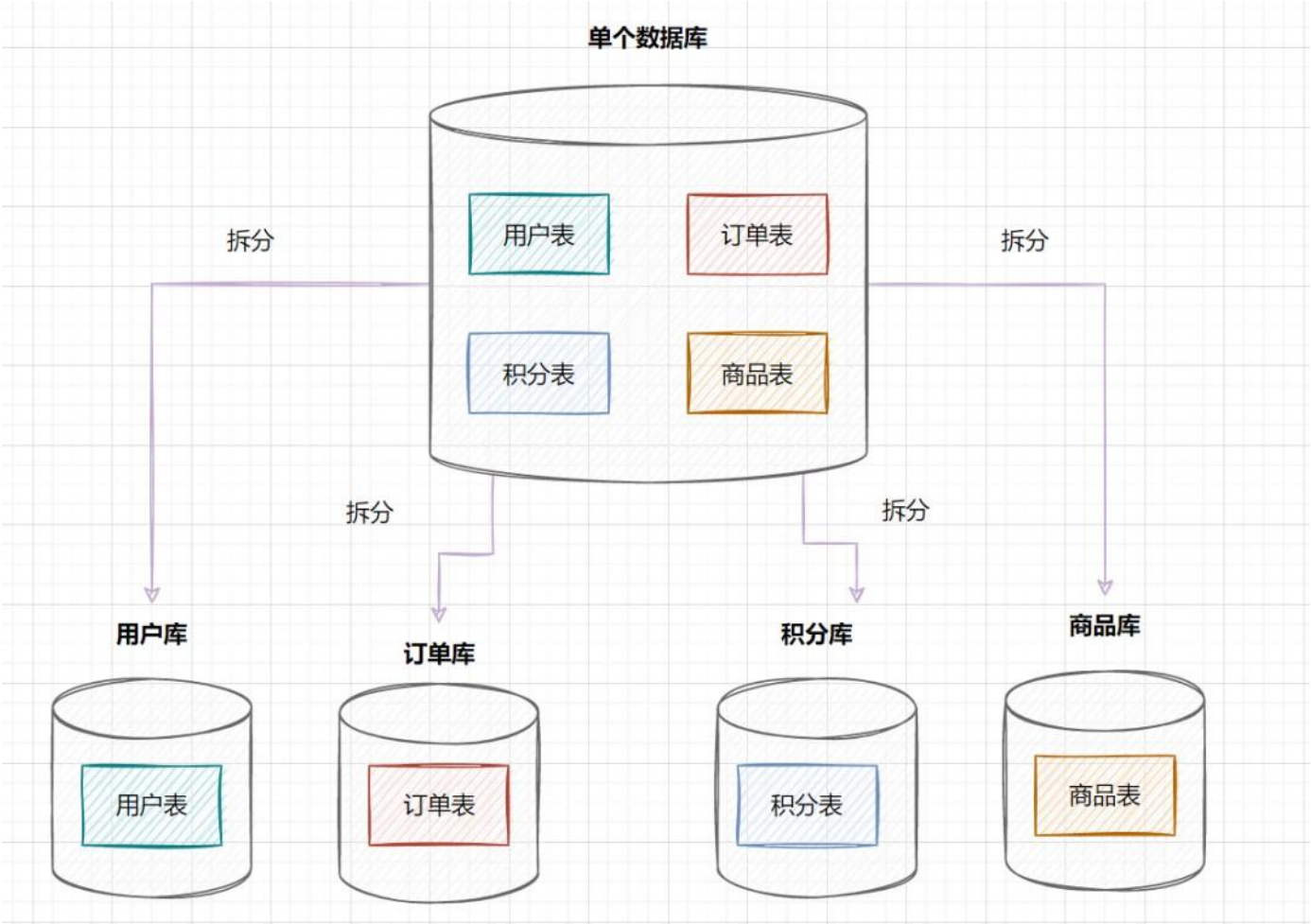
### 3.1.1 垂直分库

在业务发展初期，业务功能模块比较少，为了快速上线和迭代，往往采用单个数据库来保存数据。数据库架构如下：




但是随着业务蒸蒸日上，系统功能逐渐完善。这时候，可以按照系统中的不同业务进行拆分，比如拆分成**用户库、订单库、积分库、商品库**，把它们部署在不同的数据库服务器，这就是**垂直分库**。

垂直分库，将原来一个单数据库的压力分担到不同的数据库，可以很好应对高并发场景。数据库垂直拆分后的架构如下：




### 3.1.2 垂直分表

如果一个单表包含了几十列甚至上百列，管理起来很混乱，每次都select *的话，还占用IO资源。这时候，我们可以将一些**不常用的、数据较大或者长度较长的列**拆分到另外一张表。

比如一张用户表，它包含user_id、user_name、mobile_no、age、email、nickname、address、user_desc，如果email、address、user_desc等字段不常用，我们可以把它拆分到另外一张表，命名为用户详细信息表。这就是垂直分表

## 3.2 水平拆分

### 3.2.1 水平分库

水平分库是指，将表的数据量切分到不同的数据库服务器上，每个服务器具有相同的库和表，只是表中的数据集合不一样。它可以有效的缓解单机单库的性能瓶颈和压力。

用户库的水平拆分架构如下：

### 3.2.2 水平分表

如果一个表的数据量太大，可以按照某种规则（如hash取模、range），把数据切分到多张表去。一张

订单表，按时间下：

## 3.3. 水平分库分表策略

分库分表策略一般有几种，使用与不同的场景：

- range范围
- hash取模
- range+hash取模混合

### 3.3.1 range范围

range，即范围策略划分表。比如我们可以将表的主键，按照从 0~1000万 的划分为一个表， 1000~2000万 划分到另外一个表。如下图：

当然，有时候我们也可以按时间范围来划分，如不同年月的订单放到不同的表，它也是一种range的划分策略。

**这种方案的优点：**

- 这种方案有利于扩容，不需要数据迁移。假设数据量增加到5千万，我们只需要水平增加一张表就好啦，之前 0~4000万 的数据，不需要迁移。

**缺点：**

- 这种方案会有热点问题，因为订单id是一直在增大的，也就是说最近一段时间都是汇聚在一张表里面的。比如最近一个月的订单都在1000万~2000万之间，平时用户一般都查最近一个月的订单比较多，请求都打到 order_1 表啦，这就导致**数据热点**问题。

### 3.3.2 hash取模

hash取模策略：指定的路由key（一般是user_id、订单id作为key）对分表总数进行取模，把数据分散到各个表中。

比如原始订单表信息，我们把它分成4张分表：

- 比如id=1，对4取模，就会得到1，就把它放到t_order_1; id=3，
- 对4取模，就会得到3，就把它放到t_order_3;

**这种方案的优点：**

- hash取模的方式，不会存在明显的热点问题。

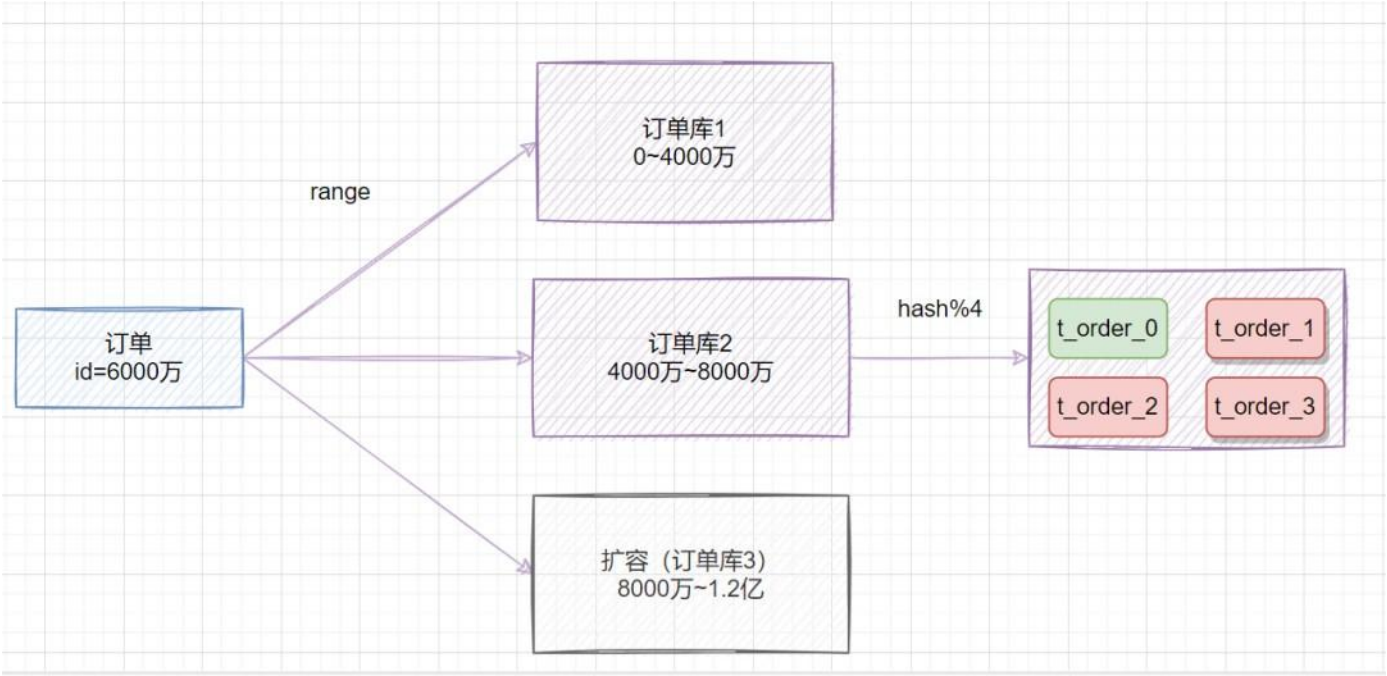
**缺点：**

- 如果一开始按照hash取模分成4个表了，未来某个时候，表数据量又到瓶颈了，需要扩容，这就比较棘手了。比如你从4张表，又扩容成 8张表，那之前id=5的数据是在（ 5%4=1，即t_order_1），现在应该放到（ 5%8=5，即t_order_5），也就是说**历史数据要做迁移了**。

### 3.3.3 range+hash取模混合

既然range存在热点数据问题，hash取模扩容迁移数据比较困难，我们可以综合两种方案一起嘛，取之之长，弃之之短。

比较简单的做法就是，在拆分库的时候，我们可以先用range**范围**方案，比如订单id在0~4000万的区间，划分为订单库1;id在4000万~8000万的数据，划分到订单库2,将来要扩容时，id在8000万~1.2亿的数据，划分到订单库3。然后订单库内，再用hash**取模**的策略，把不同订单划分到不同的表。




# 4. 什么时候才考虑分库分表呢？

## 4.1 什么时候分表？

如果你的系统处于快速发展时期，如果每天的订单流水都新增几十万，并且，订单表的查询效率明变慢时，就需要规划分库分表了。一般B+树索引高度是2~3层最佳，如果**数据量千万级别**，可能高度就变4层了，数据量就会明显变慢了。不过业界流传，一般500万数据就要**考虑分表**了。

## 4.2 什么时候分库

业务发展很快，还是多个服务共享一个单体数据库，数据库成为了性能瓶颈，就需要考虑分库了。比如订单、用户等，都可以抽取出来，新搞个应用（其实就是微服务思想），并且拆分数据库（订单库、用户库）。

# 5. 分库分表会导致哪些问题

分库分表之后，也会存在一些问题：

- 事务问题
- 跨库关联
- 排序问题
- 分页问题
- 分布式ID

## 5.1 事务问题

分库分表后，假设两个表在不同的数据库，那么本地事务已经无效啦，需要使用分布式事务了。

## 5.2 跨库关联

跨节点Join的问题：解决这一问题可以分两次查询实现

## 5.3 排序问题

跨节点的count,order by,group by以及聚合函数等问题：可以分别在各个节点上得到结果后在应用程序端进行合并。

## 5.4 分页问题

- 方案1：在个节点查到对应结果后，在代码端汇聚再分页。
- 方案2：把分页交给前端，前端传来pageSize和pageNo，在各个数据库节点都执行分页，然后汇聚总数量前端。这样缺点就是会造成空查，如果分页需要排序，也不好搞。

## 5.5 分布式ID

数据库被切分后，不能再依赖数据库自身的主键生成机制啦，最简单可以考虑UUID，或者使用雪花算法生成分布式ID。

# 6. 分库分表中间件

目前流行的分库分表中间件比较多：

- cobar
- Mycat
- Sharding-JDBC
- Atlas　TDDL
- （淘宝） vitess
-

# 📚 第 7 章：MySQL 主从

> 之前发的这篇文章，因为没有标注参考文章，所以已经删除，现在重新发出。

大家好，我是楼仔！

MySQL　主从一直是面试常客，里面的知识点虽然基础，但是能回答全的同学不多。

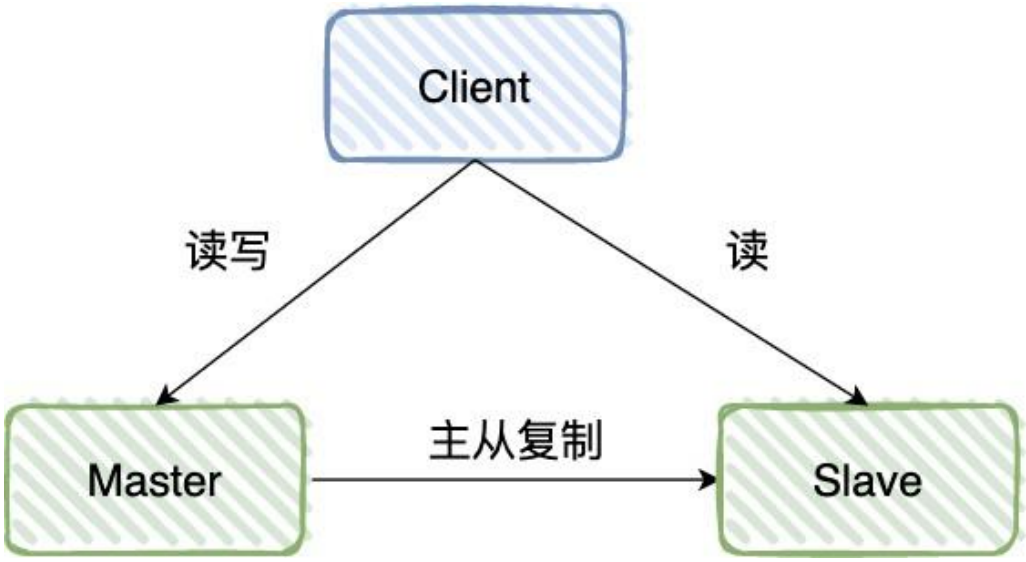
比如我之前面试小米，就被问到过主从复制的原理，以及主从延迟的解决方案，你之前面试，有遇到过哪些 MySQL 主从的问题呢？

文章很多内容参考田螺哥的文章，详见参考文章，特此说明！

# 1. MySQL 主从

## 1.1 什么是 MySQL 主从？

所谓 MySQL 主从，就是建立两个完全一样的数据库，一个是主库，一个是从库，**主库对外提供读写的操作，从库对外提供读的操作**。




## 1.2 为什么使用 MySQL 主从？

对于数据库单机部署，在 4 核 8G 的机器上运行 MySQL 5.7 时，大概可以支撑 500 的 TPS 和 10000 的 QPS，**当遇到一些活动时，查询流量骤然，就需要进行主从分离。**

大部分系统的访问模型是读多写少，读写请求量的差距可能达到几个数量级，所以我们可以通过一主多从的方式，**主库只负责写入和部分核心逻辑的查询，多个从库只负责查询，提升查询性能，降低主库压力。**

当主库宕机时，从库可以切成主库，保证服务的高可用，然后主库也可以做数据的容灾备份，整体场景总结如下：

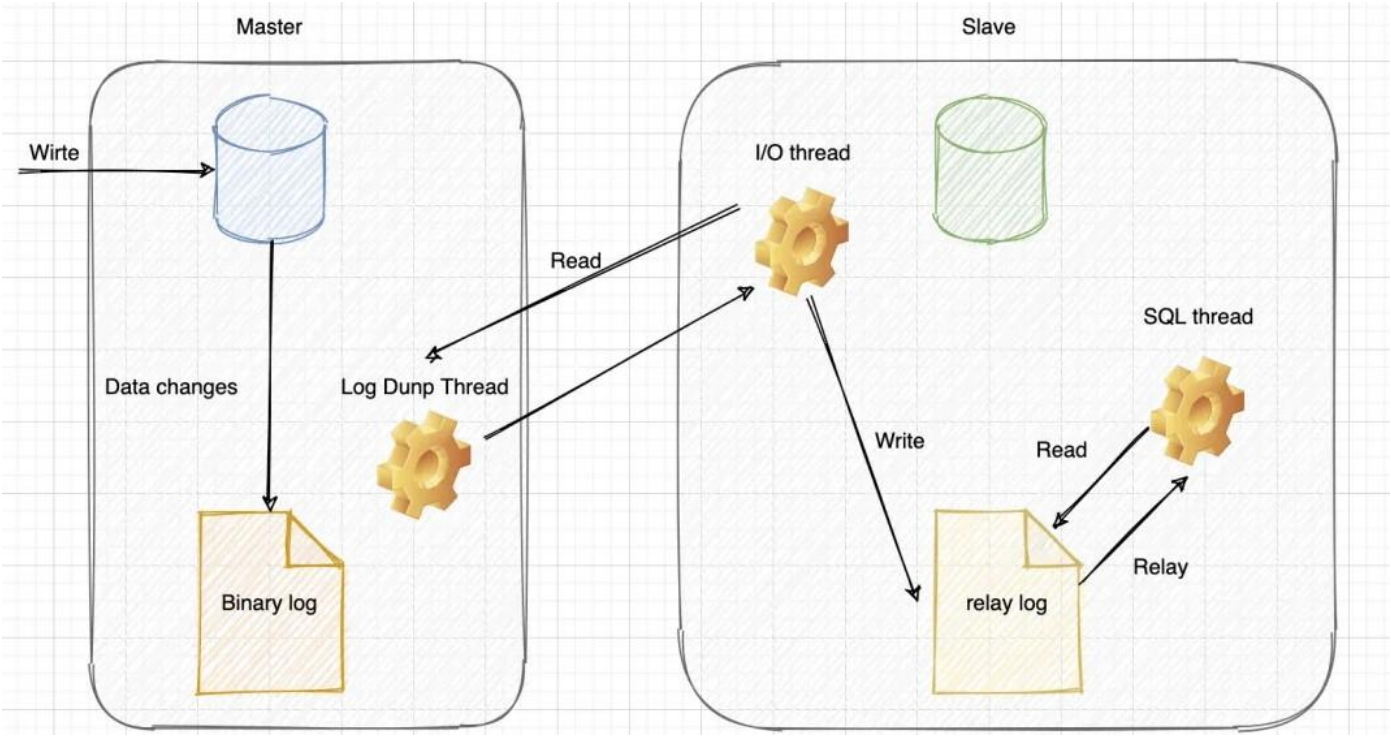
- **读写分离**：从库提供查询，减少主库压力，提升性能；
- **高可用**：故障时可切换从库，保证服务高可用；
- **数据备份**：数据备份到从库，防止服务器宕机导致数据丢失。

# 2. 主从复制

## 2.1 主从复制原理

MySQL 的主从复制是依赖于 binlog，也就是记录 MySQL 上的所有变化并以二进制形式保存在磁盘上二进制日志文件。

主从复制就是将 binlog 中的数据从主库传输到从库上，一般这个过程是异步的，即主库上的操作不会等待 binlog 同步地完成。




详细流程如下：

1. **主库写** binlog：主库的更新 SQL(update、insert、delete) 被写到 binlog；
2. **主库发送** binlog：主库创建一个 log dump 线程来发送 binlog 给从库；
3. **从库写** relay log：从库在连接到主节点时会创建一个 IO 线程，以请求主库更新的 binlog，并且把接收到的 binlog 信息写入一个叫做 relay log 的日志文件；
4. **从库回放**：从库还会创建一个 SQL 线程读取 relay log 中的内容，并且在从库中做回放，最终实现主从的一致性。

## 2.2 如何保证主从一致

当主库和从库数据同步时，突然中断怎么办？因为主库与从库之间维持了一个长链接，主库内部有一个线程，专门服务于从库的这个长链接。

对于下面的情况，假如主库执行如下 SQL，其中 a 和 create_time 都是索引：

```
delete from t where a > '666' and create_time<'2022-03-01' limit 1;
```

我们知道，数据选择了 a 索引和选择 create_time 索引，最后 limit 1 出来的数据一般是不一样的。

所以就会存在这种情况：在 binlog = statement 格式时，主库在执行这条 SQL 时，使用的是索引 a，而从库在执行这条 SQL 时，使用了索引 create_time，最后主从数据不一致了。

**那么我们该如何解决呢？**

可以把 binlog 格式修改为 row，row 格式的 binlog 日志记录的不是 SQL 原文，而是两个 event:Table_map 和 Delete_rows。

Table_map event 说明要操作的表，Delete_rows event用于定义要删除的行为，记录删除的具体行数。row **格式的** binlog **记录的就是要删除的主键** ID **信息，因此不会出现主从不一致的问题。**

但是如果 SQL 删除 10 万行数据，使用 row 格式就会很占空间，10 万条数据都在 binlog 里面，写 binlog 的时候也很耗 IO。但是 statement 格式的 binlog 可能会导致数据不一致。

设计 MySQL 的大叔想了一个折中的方案，mixed 格式的 binlog，其实就是 row 和 statement 格式混合使用，**当** MySQL **判断可能数据不一致时，就用** row **格式，否则使用就用** statement **格式。**

# 3. 主从延迟

有时候我们遇到从数据库中获取不到信息的诡异问题时，会纠结于代码中是否有一些逻辑会把之前写入的内容删除，但是你又会发现，过了一段时间再去查询时又可以读到数据了，这基本上就是主从延迟在作怪。

主从延迟，其实就是“从库回放” 完成的时间，与 “主库写 binlog” 完成时间的差值，**会导致从库查询的数据，和主库的不一致**。

## 3.1 主从延迟原理

谈到 MySQL 数据库主从同步延迟原理，得从 MySQL 的主从复制原理说起：

- MySQL 的主从复制都是单线程的操作，主库对所有 DDL 和 DML 产生 binlog，binlog 是顺序写，所以效率很高；
- Slave 的 Slave_IO_Running 线程会到主库取日志，放入 relay log，效率会比较高；
- Slave 的 Slave_SQL_Running 线程将主库的 DDL 和 DML 操作都在 Slave 实施，DML **和** DDL **的** IO **操作是随机的，不是顺序的，因此成本会很高**，还可能是 Slave 上的其他查询产生 lock 争用，由于 Slave_SQL_Running 也是单线程的，所以**一个** DDL **卡住了，需要执行** 10 **分钟，那么所有之后的** DDL **会等待这个** DDL **执行完才会继续执行，这就导致了延时。**

**总结一下主从延迟的主要原因**：主从延迟主要是出现在 “relay log 回放” 这一步，当主库的 TPS 并发较高，产生的 DDL 数量超过从库一个 SQL 线程所能承受的范围，那么延时就产生了，当然还有就是可能与从库的大型 query 语句产生了锁等待。

## 3.2 主从延迟情况

- **从库机器性能**：从库机器比主库的机器性能差，只需选择主从库一样规格的机器就好。
- **从库压力大**：可以搞了一主多从的架构，还可以把 binlog 接入到 Hadoop 这类系统，让它们提供查询的能力。
- **从库过多**：要避免复制的从节点数量过多，**从库数据一般以**3-5**个为宜。**
- **大事务**：如果一个事务执行就要 10 分钟，那么主库执行完后，给到从库执行，最后这个事务可能就会导致从库延迟 10 分钟啦。日常开发中，不要一次性 delete 太多 SQL，需要分批进行，另外大表的 DDL 语句，也会导致大事务。
- **网络延迟**：优化网络，比如带宽 20M 升级到 100M。
- MySQL **版本低**：低版本的 MySQL 只支持单线程复制，如果主库并发高，来不及传送到从库，就会导致延迟，可以换用更高版本的 MySQL，支持多线程复制。

## 3.3 主从延迟解决方案

我们一般会把从库落后的时间作为一个重点的数据库指标做监控和报警，正常的时间是在毫秒级别，一旦落后的时间达到了秒级别就需要告警了。

解决该问题的方法，除了缩短主从延迟的时间，还有一些其它的方法，基本原理都是尽量不查询从库，具体解决方案如下：

- **使用缓存**：我们在同步写数据库的同时，也把数据写到缓存，查询数据时，会先查询缓存，不过这种情况会带来 MySQL 和 Redis 数据一致性问题。
- **查询主库**：直接查询主库，这种情况会给主库太大压力，不建议这种方式。
- **数据冗余**：对于一些异步处理的场景，如果只扔数据 ID，消费数据时，需要查询从库，我们可以把数据全部都扔给消息队列，这样消费者就无需再查询从库。（这种情况应该不太能出现，数据转了一圈，MySQL 主从还没有同步好，直接去撕 DBA 吧）

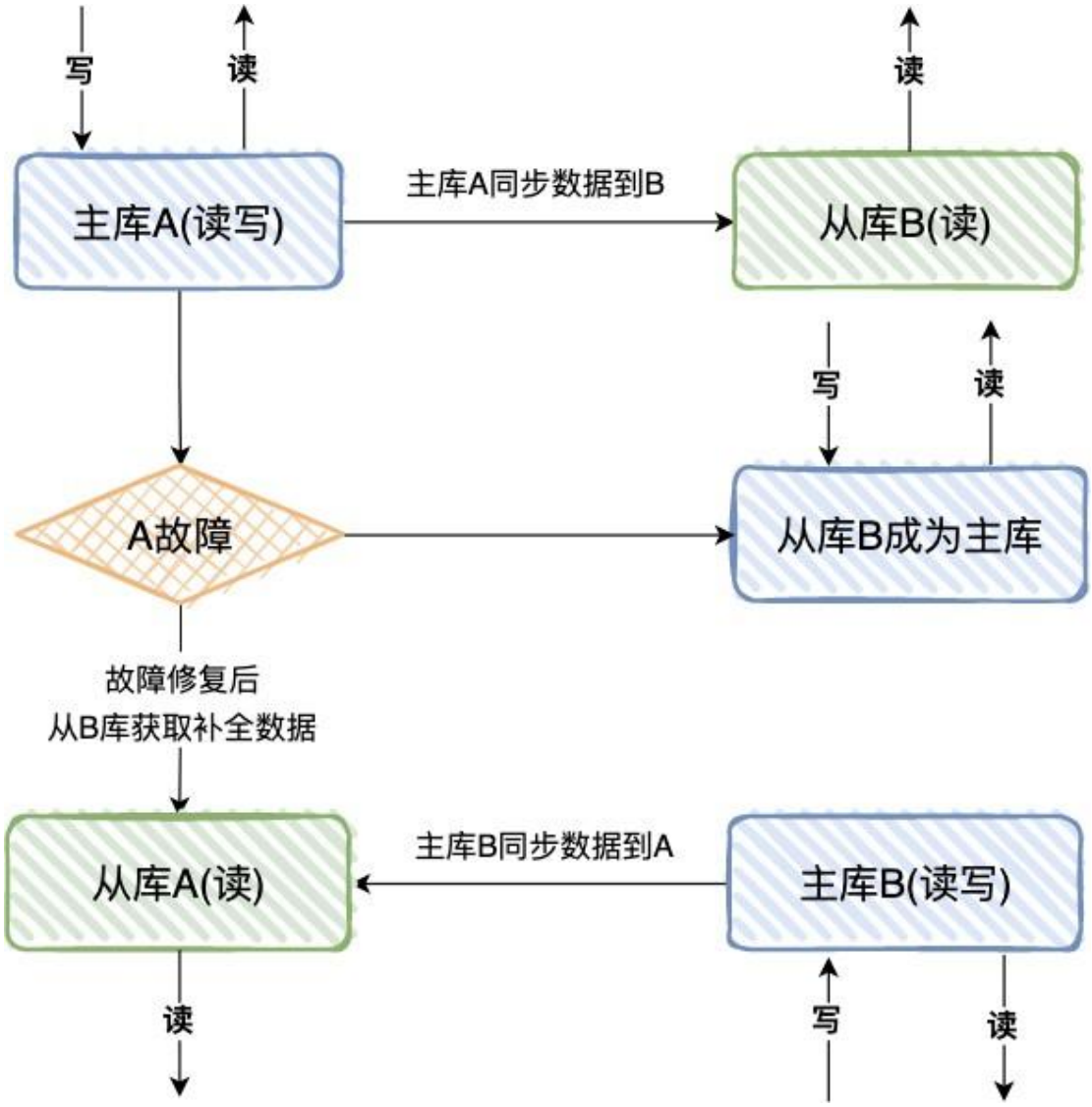
> 在实际应用场景中，对于一些非常核心的场景，比如库存，支付订单等，需要直接查询主库，其它非核心场景，就不要去查主库了。

# 4. 主从切换

## 4.1 一主一从

两台机器 A 和 B，A 为主库，负责读写，B 为从库，负责读数据。

如果 A 库发生故障，B 库成为主库负责读写，修复故障后，A 成为从库，主库 B 同步数据到从库 A。



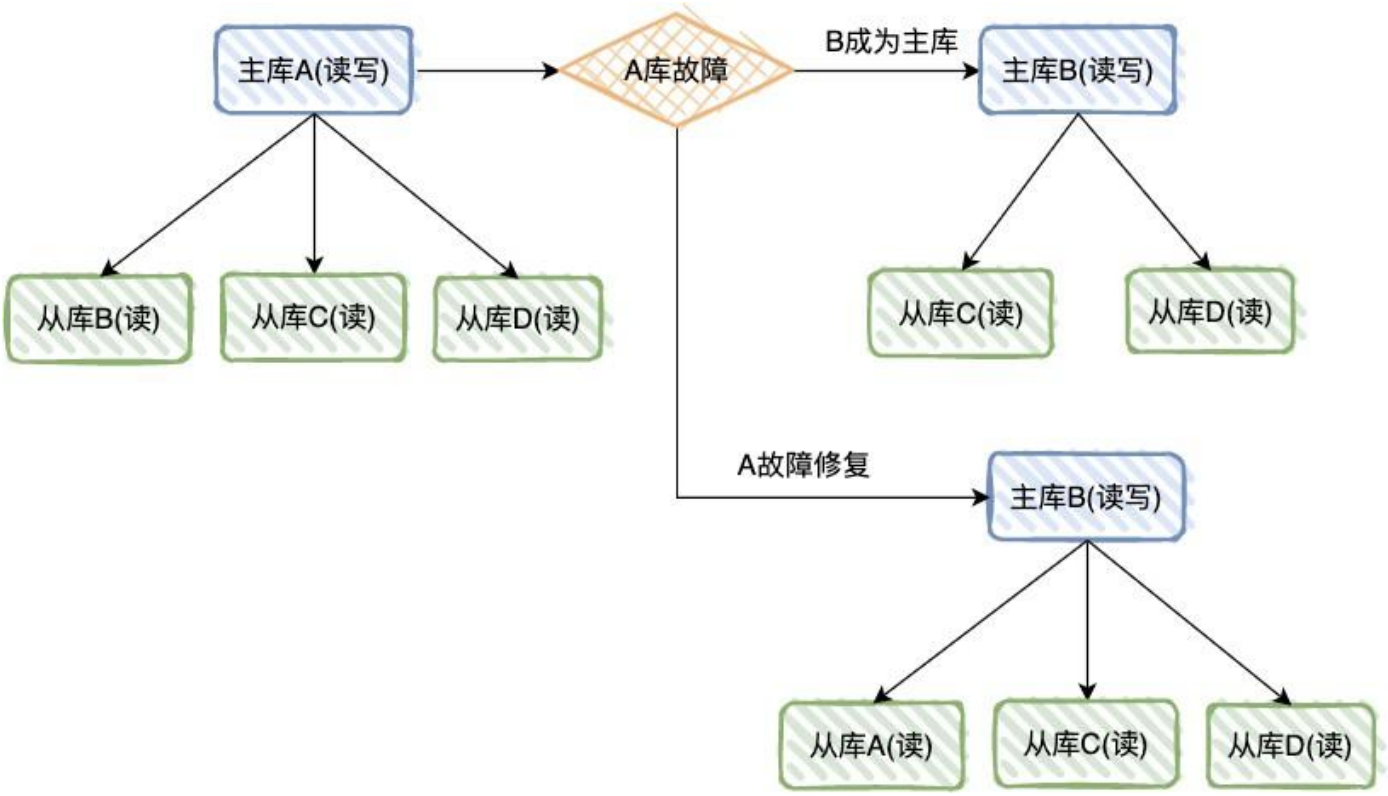

- **优点：** 从库支持读，分担了主库的压力，提升了并发度，且一个机器故障了可以自动切换，操作比较简单，公司从库还可以充当数据备份的角色；
- **缺点：** 一台从库，并发支持还是不够，并且一共两台机器，还是存在同时故障的机率，不够高可用。

> 对于一主一从的模式，一般小公司会这么用，不过该模式下，主从分离的意义其实并不大，因为小公司的流量不高，更多是为了数据库的可用性，以及数据备份。

## 4.2 一主多从

一台主库多台从库，A 为主库，负责读写，B、C、D为从库，负责读数据。

如果 A 库发生故障，B 库成为主库负责读写，C、D 负责读，修复故障后，A 也成为从库，主库 B 同步数据到从库 A。




- **优点：** 多个从库支持读，分担了主库的压力，明显提升了读的并发度。
- **缺点：** 只有一台主机写，因此写的并发度不高。

> 基本上大公司，比如百度、滴滴，都是这种一主多从的模式，因为查询流量太高，一定需要进行读写分离，同时也需要支持服务的高可用、数据容灾。
>
> 再结合上一篇文章讲的分库分表，基本就是大公司的标配了。

# 参考资料


- 捡田螺的小男孩 《我们为什么要分库分表？》
- 极客时间《高并发系统设计 40 问》


也可以扫下面的二维码，和楼仔一起交流学习哈 ~~