高可用、高并发、负载均衡的架构设计

1. 高可用-互联网高可用架构设计
2. 什么是高可用

高可用HA通常指，通过设计减少系统不能提供服务的时间。

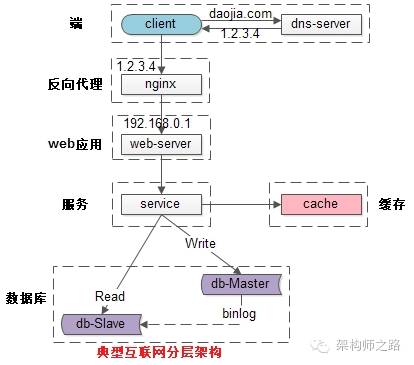
1. 如何保障高可用

单点是系统高可用的大敌，应该尽量在系统设计中避免单点。方法论上，高可用保证的原则是“集群化”，或者叫“冗余”：只有一个单点，挂了的话服务会受影响；如果有冗余备份，挂了还有其他backup能够顶上。

保证系统高可用，架构设计的核心准则是：冗余。有了“冗余”还需要在出现故障之后人工介入恢复，这也会增加系统的不可服务时间，所以往往通过“自动故障转移”来实现系统的高可用。

也就是说“冗余+自动故障转移”保证了系统的高可用特性。下面学习以下几个架构设计如何保证高可用

1. 互联网分层架构

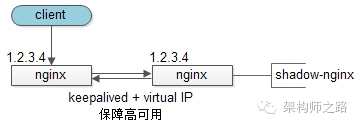


如上：

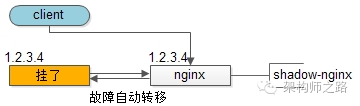
* 客户端层：典型调用方是浏览器browser或者手机应用APP
* 反向代理层：系统入口，反向代理
* 站点应用层：实现核心应用逻辑，返回html或者json
* 服务层：如果实现了服务化的话
* 数据-缓存层：缓存加速访问存储
* 数据-数据库层：数据库固化数据存储

其中每一层的“冗余+自动故障转移”综合实现，保证了系统的高可用性

* 客户端层->反向代理层的高可用

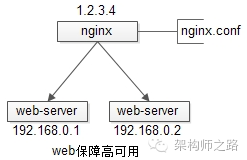


通过反向代理层的冗余来实现该一层的高可用。有两台nginx，其中一台用于对线上提供服务，另一台冗余以保证高可用，常见的时间是keepalived存活探测，相同virtual IP提供服务。

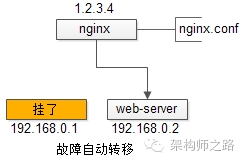


当nginx挂了的时候，keepalived能够探测到，会自动进行故障转难以，将流量自动迁移到shadow-nginx，由于使用的是相同的virtual IP，这个切换过程对调用方是透明的

* 反向代理层->站点层的高可用

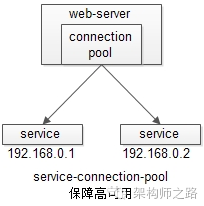


通过站点层的冗余来实现，假设反向代理层是nginx，nginx.conf里能够配置多个web后端，并且nginx能够探测到多个后端的存活性

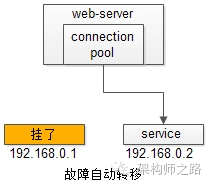


自动故障转移：当web-server挂了的时候，nginx能够探测到，会自动地进行故障转移，将流量自动迁移到其他的web-server，整个过程有nginx自动完成，对调用方透明。

* 站点层->服务层的高可用

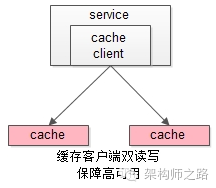


通过服务层的冗余来实现。“服务连接池”会建立与下游服务的多个连接，每次请求会“随机”选取连接来访问下游服务。

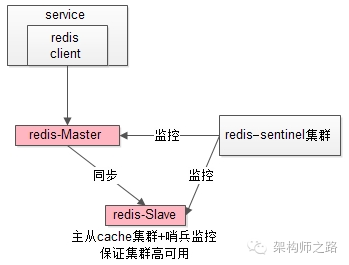


自动故障转移：当service挂了的时候，service-connection-pool能够探测到，会自动的进行故障转移，将流量自动迁移到其他的service，整个过程由连接池自动完成，对调用方是透明的(所以说RPC-client中的服务连接池是很重要的基础组件)

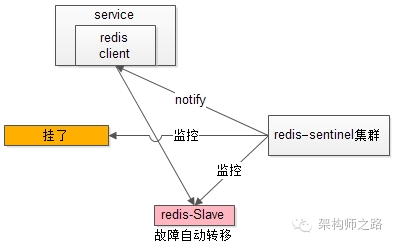
* 服务层-缓存层的高可用



* 通过缓存数据的冗余来实现。缓存层的数据冗余又有几种方式：第一种是利用客户端的封装，service对cache进行双读或双写。
* 缓存层还可以通过支持主从同步的缓存集群来解决缓存层的高可用问题。以redis为例，redis天然支持主从同步，redis官方也有sentine哨兵机制，来做redis存活性检测。

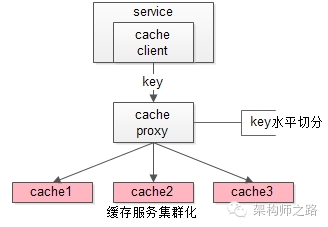


自动故障转移：当redis主挂了之后。Sentinel能够探测到，会通知调用方访问新的redis，整个过程由sentinel和redis集群配合完成，对调用方是透明的。

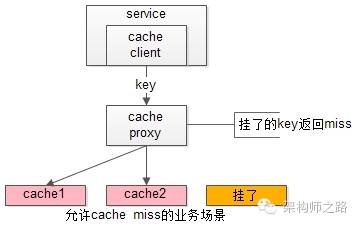


但是业务对缓存并一定有“高可用”的要求，更多的是对缓存的使用场景，是用来“加速数据访问”：把一部分数据放在缓存里，如果缓存挂了或者没有命中，是可以去后端的数据库再取数据的。

这类允许“cache-miss”的业务场景，缓存架构如下：

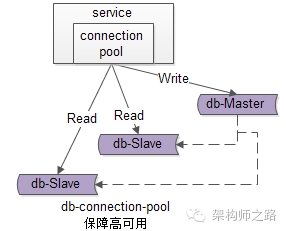


将kv缓存封装成服务集群，上游设置一个代理（代理可以用集群冗余的方式保证高可用），代理后端根据缓存访问的key水平切分成若干个实例，每个实例的访问并不做高可用。

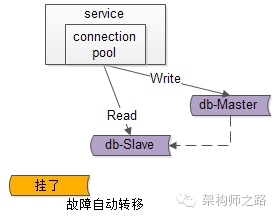


缓存实例挂了屏蔽：当有水平切分的实例挂掉时，代理层直接返回cache miss，此时缓存挂掉对调用方是透明的。Key水平切分实例减少，不建议做re-hash，这样容易引发缓存数据的不一致。

* 服务层->数据库层的高可用
* 读库高可用

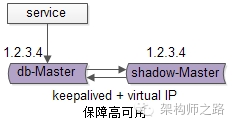


读库高可用通过读库的冗余实现。既然冗余了读库，一般来说就至少有2个从库，“数据库连接池”会建立与读库多个连接，每次请求会路由到这些读库

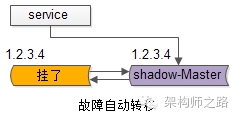


自动故障转移：当读库挂了的时候，db-connection-pool能够探测到，会自动的进行故障转移，将流量自动迁移到其他的读库，整个过程由连接池自动完成，对调用方透明（说明DAO中的数据库连接池是很重要的基础组建）。

* 写库高可用



写库的高可用，是通过写库的冗余来实现的。以mysql为例，可以设置两个mysql双主同步，一台对线上提供服务，另一台冗余以保证高可用，常见的实践是keepalived存活探测，相同的virtual IP提供服务



自动故障转移：当写库挂了的时候，keepalived能够探测到，会自动的进行故障转移，将流量自动迁移到shadow-db-master，由于使用的是相同的virtual IP，这个切换过程对调用方式透明的。

1. 高并发-互联网高并发架构设
2. 什么是高并发

高并发HC是互联网分布式系统架构设计中必须考虑的因素之一，他通常是指通过设计保证系统能够同时并行处理很多请求。

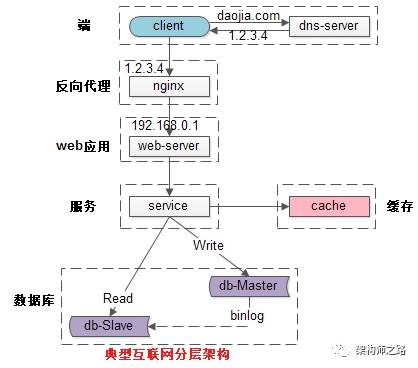
1. 高并发常用的指标

* 响应时间：系统对请求做出相应的时间
* 吞吐量：单位时间内处理的请求数量
* QPS：每秒相应请求书
* 并发用户数：同时承载正常使用系统功能的用户数量

1. 如何提升系统高并发能力

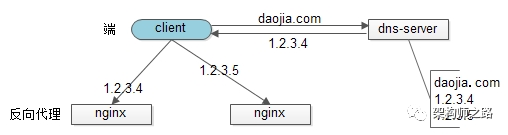
* 垂直扩展：提升单机处理能力
* 增强单机硬件性能
* 提升单机架构性能
* 水平扩展：增加服务器数量，线性扩充系统性能

1. 常见的互联网分层架构



1. 分层水平扩展架构实践

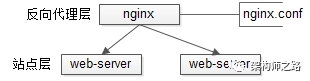
* 反向代理层的水平扩展



反向代理层的水平扩展，是通过“DNS轮询”实现：dns-server对于一个域名配置了多个解析ip，每次DNS解析请求来访问dns-server，会轮询返回这些ip。

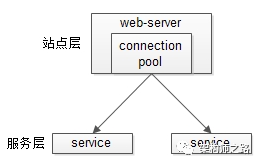
当nginx成为瓶颈，只要增加服务器数量，新增nginx服务的部署，增加了一个外网ip，就能扩展反向代理层的性能，做到理论上无限高并发。

* 站点层的水平扩展



站点层的水平扩展，是通过“nginx”来实现的。通过修改nginx.conf，可以设置多个web后端。当web后端成为瓶颈的时候，只要增加服务器数量，新增web服务的部署，在nginx配置中配置上新的web后端，就能扩展站点层的性能，做到理论上的无限高并发

* 服务层的水平扩展



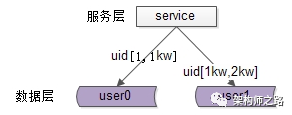
服务层的水平扩展，是通过“服务连接池”实现的。站点层通过RPC-client调用下游的服务层RPC-server时，RPC-client中的连接池会建立与下游服务多个连接，当服务成为瓶颈时，只要增加服务器数量，新增服务部署，在RPC-client处建立新的下游服务连接，就能扩展服务层性能，做到理论上的高并发。

* 数据层的水平扩展

数据量很大的情况下，数据层(缓存，数据库)设计数据的水平扩展，将原本存储在一台服务器上的数据(缓存，数据库)水平拆分到不同服务器上去，以达到扩充系统性能的目的。

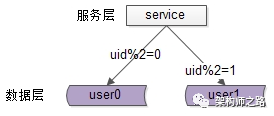
互联网数据层常见水平拆分方式有以下几种(以数据库为例)：

1. 按范围水平拆分



每一个数据服务，存储一定范围的数据，这样做的好处在于：1) 规则简单，service只要判断一下uid范围就能路由到对应的存储服务；2) 数据均衡性较好；3) 比较容易扩展。这样做的不足：请求的负载不一定均衡，一般来说，新注册的用户会比老用户更加活跃，大range的服务请求压力会更大。

1. 按照哈希水平拆分



每个数据库，存储某个key值hash后的部分数据，方案的好处在于：1）规则简单，service只需要对uid进行hash就能路由到对应的存储服务；2）数据均衡性较好；3）请求均匀性较好。不足在于：不容易扩展，hash方法改变的时候，可能需要数据迁移。

通过水平拆分扩充系统性能与主从同步读写分离来扩充数据库性能的方式有本质不同，在于前者：

1. 每个服务器上存储的数据量是总量的1/n，所以单机的性能也有提升
2. N个服务器的数据没有并集，n个服务器上数据的并集是数据的全集
3. 数据水平拆分到那个服务器，每个服务器存储1/n的数据，服务器的读性能和写性能提升n倍(实际上不止n倍，因为每个服务器的数据只有总数据量的1/n，单机性能也有提升)

而后者：

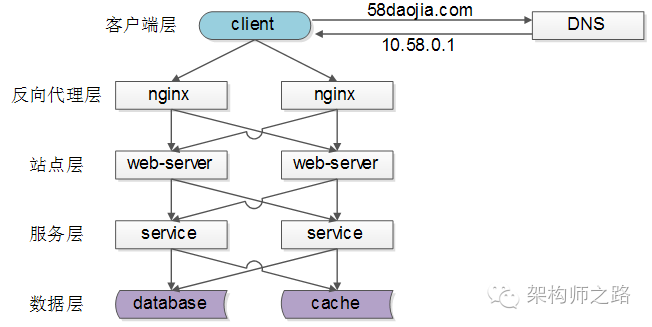
1. 每个服务器上都存储的数据量与总量相同
2. N个服务器上存储的数据都一样，都是全集
3. 理论上提升了读性能，但是写性能仍是单点

* 缓存层的水平拆分与数据库层的水平拆分类似，以范围拆分和哈希拆分为主。

1. 负载均衡-互联网负载均衡架构设计
2. 什么是负载均衡

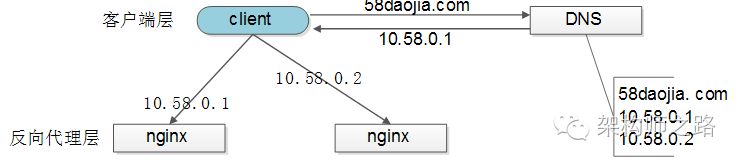
负载均衡(Load Balance)是分布式系统架构设计中必须考虑的因素之一，它通常是指将请求/数据均匀地分摊到多个操作单元上执行，关键在于“均衡”。

1. 常见的复杂均衡方案



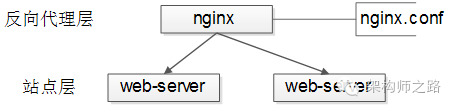
常见的互联网分布式架构如上，分为客户端层、反向代理nginx层，站点层，服务层，数据层。可以看到，每个下游都有多个上游调用，主要做到每一个上游都均匀地访问每一个下游，就能实现“将请求/数据均匀地分摊到多个操作单元上执行”。

* 客户端层到反向代理层的负载均衡



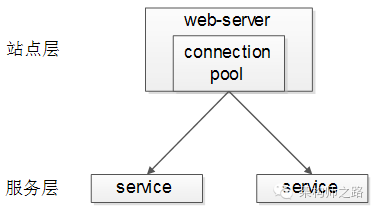
通过“DNS”轮询实现的：DNS-server对于一个域名配置了多个解析ip，没戏DNS解析请求来访问DNS-server，会轮询返回这些ip，保证每个ip的解析概率是相同的。这些ip就是nginx的外网ip，以做到每台nginx的请求也是均衡的。

* 反向代理层到站点层的负载均衡



通过“nginx”实现。通过修改nginx.conf，可以实现多种负载均衡策略：

* + 请求轮询：和DNS轮询类似，请求一次路由到各个web-server
  + 最少连接路由：哪个web-server连接少，路由到哪个web-server
  + Ip哈希：按照访问用户的ip哈希值来路由到web-server，只要用户的ip分布是均匀的，请求理论上也是均匀的，ip哈希均衡方法可以做到，同一个用户的请求固定落到同一台web-server上，此策略适合有状态服务，例如session(但是站点层无状态是分布式架构设计的基本远侧之一，session最好放到数据层存储)
* 站点层到服务层的负载均衡



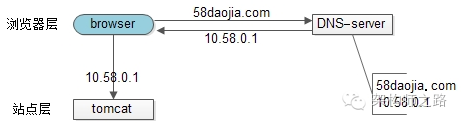
通过“服务连接池”实现，上游连接池会建立与下游服务的多个链接，每次请求会随机选取连接来访问下游服务。

* 数据层负载均衡

在数据量很大的时候，由于数据层(db,cache)涉及数据的谁聘切分，所以数据层的负载均衡更为复杂一些，它分为“数据的均衡”和“请求的均衡”。关于水平切分方式，不在赘述。

1. 接入层负载均衡、接入层技术演进

* 裸奔时代(0)\_单机架构



如上：

* + - 浏览器通过DNS-server，域名解析到ip
    - 浏览器通过ip访问web-server

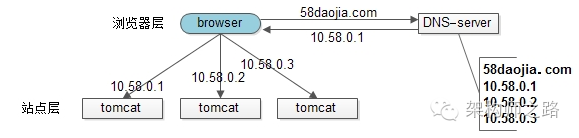
缺点：

* + - 非高可用，web-server挂了整个系统就挂掉了
    - 扩展性差，当吞吐量达到web-server上限，无法扩容

\*单机架构不涉及负载均衡问题

* 简易扩容方案(1)\_DNS轮询

假设tomcat的吞吐量是1000次每秒，当系统总的吞吐量达到3000时，如何扩容是首要解决的问题，由此到DNS轮询。



如上

* + - 多部署几份web-server，一个Tomcat分担1000就好了
    - 在DNS-server层面，域名每次解析到不同的ip

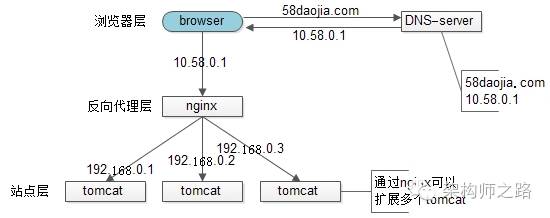
优点

* + - 零成本：在DNS-server上多分配几个ip即可，功能也不收费
    - 部署简单：多部署几个web-server即可，原系统架构不需要做任何改造
    - 负载均衡：变成了多机，但负载基本上是均衡的

缺点

* + - 非高可用：DNS-server只负责域名解析ip，这个ip对应的服务是否高可用，DNS-server是不保证的，假设有一个web-server挂了，部分服务会受到影响
    - 扩容非实时：DNS解析有一定的生效周期
    - 暴露了太多的外网ip
* 简易扩容方案(2)\_nginx

相比于tomcat，nginx作为反向代理的性能要好的多。假设线上到1W，性能就比tomcat高了10倍，可以利用这个特性扩容



如上

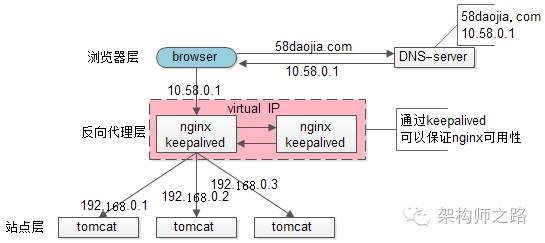
* + - 站点层与浏览器层之间加入一个反向代理，利用高性能的nginx来做反向代理
    - Nginx将http请求分发给后端多个web-server

优点

* + - DNS-server不需要动
    - 负载均衡：通过nginx保证
    - 只暴露一个外网ip，nginx之间使用内网访问
    - 扩容实时，nginx内部可控，随时增加web-server随时实时扩容
    - 能够保证站点层的可用性：任何一台tomcat挂了，nginx可以将流量迁移到其他tomcat

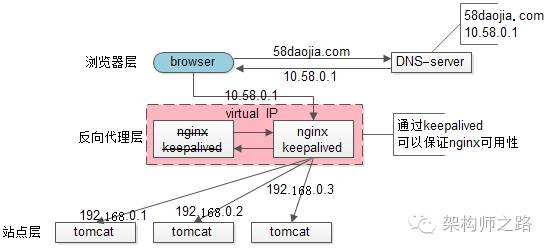
缺点

* + - 时延增加，架构复杂
    - 反向代理层变成了单点，非高可用
* 高可用方案(3)\_keepaliced



如上

* + - 两台nginx组成一个集群，分别部署上keepalived，设置成心相通的虚IP，保证nginx高可用
    - 当一台nginx挂掉的时候，keepalived能够探测到，自动将流量转移到另一台nginx上，整个过程对调用方透明



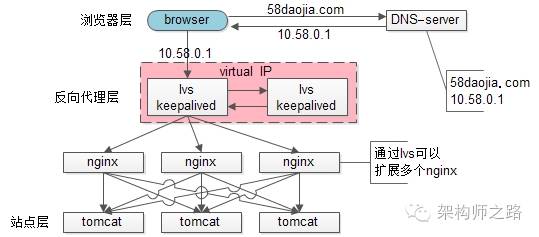
优点

* + - 解决了高可用问题

缺点

* + - 资源利用率只有50%
    - Nginx仍然是单点，接入的吞吐量超过nginx上限的情况仍然无法应对
* Scale up 扩容方案(4)\_lvs/f5

nginx毕竟是软件，就算性能比tomcat好，但是也存在上限，由此引出lvs和f5。lvs实施在操作系统层面，f5实施在硬件层面，性能递增：



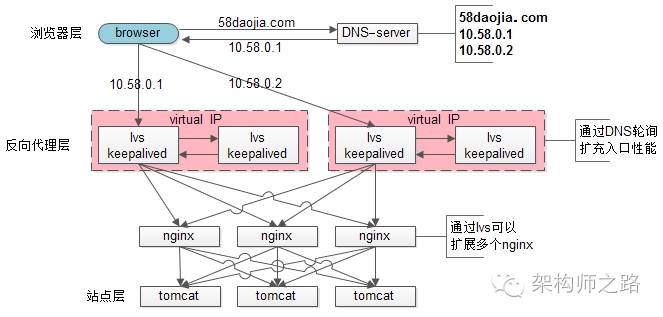
如上

* + - 可以通过lvs扩展多个nginx
    - 通过keepalived+VIP保证了高可用性

但是lvs+f5的性能还是存在上限的（已经很高了，80亿）

* Scale up 扩容方案(5)\_DNS轮询

如果超过了80亿，这是就又回到了DNS轮询的方案上



如上

* + - 通过DNS轮询来线性扩展入口lvs层的性能
    - 通过keepalived来保证高可用
    - 通过lvs来扩展多个nginx
    - 通过nginx来做负载均衡，业务七层路由