# 高并发

## 概述

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

## 度量指标

并发的指标一般有QPS、TPS、IOPS，这几个指标都是可归为系统吞吐率，QPS越高系统能hold住的请求数越多，但光关注这几个指标不够，我们还需要关注RT，即响应时间，也就是从发出request到收到response的时延，这个指标跟吞吐往往是此消彼长的，我们追求的是一定时延下的高吞吐。比如有100万次请求，99万次请求都在10毫秒内响应，其他次数10秒才响应，平均时延不高，但时延高的用户受不了，所以，就有了TP90/TP99指标，这个指标不是求平均，而是把时延从小到大排序，取排名90%/99%的时延，这个指标越大，对慢请求越敏感。除此之外，有时候，我们也会关注可用性指标，这可归到稳定性。一般而言，用户感知友好的高并发系统，时延应该控制在250毫秒以内。

通常，数据库单机每秒也就能抗住几千这个量级，而做逻辑处理的服务单台每秒抗几万、甚至几十万都有可能，而消息队列等中间件单机每秒处理个几万没问题，所以我们经常听到每秒处理数百万、数千万的消息中间件集群，而像阿里的API网关，每日百亿请求也有可能。

高并发相关常用的一些指标有响应时间（Response Time），吞吐量（Throughput），每秒查询率QPS（Query Per Second），并发用户数等。

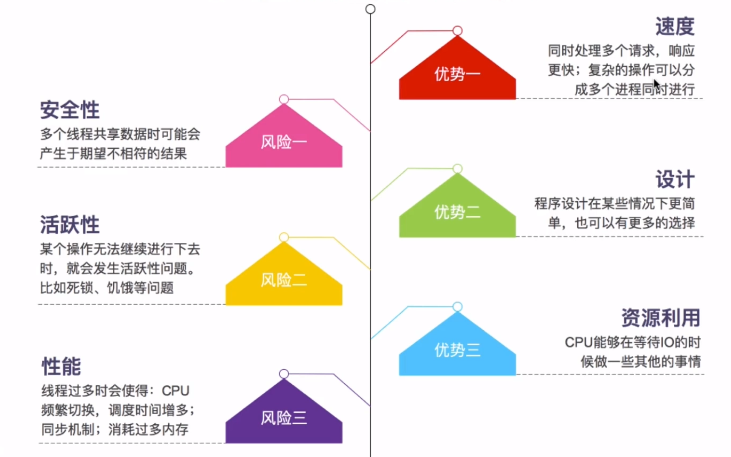
响应时间：系统对请求做出响应的时间。例如系统处理一个HTTP请求需要200ms，这个200ms就是系统的响应时间。

吞吐量：单位时间内处理的请求数量。

QPS：每秒响应请求数。在互联网领域，这个指标和吞吐量区分的没有这么明显。

并发用户数：同时承载正常使用系统功能的用户数量。例如一个即时通讯系统，同时在线量一定程度上代表了系统的并发用户数。

## 优缺点



## 提升并发

### 设计思路

高并发的设计思路有两个方向：

1. 垂直方向扩展，也叫竖向扩展
2. 水平方向扩展，也叫横向扩展

**垂直方向：提升单机能力**

提升单机处理能力又可分为硬件和软件两个方面：

硬件方向，很好理解，花钱升级机器，更多核更高主频更大存储空间更多带宽

软件方向，包括用各快的数据结构，改进架构，应用多线程、协程，以及上性能优化各种手段，但这玩意儿天花板低，就像提升个人产出一样，996、007、最多24 X 7。

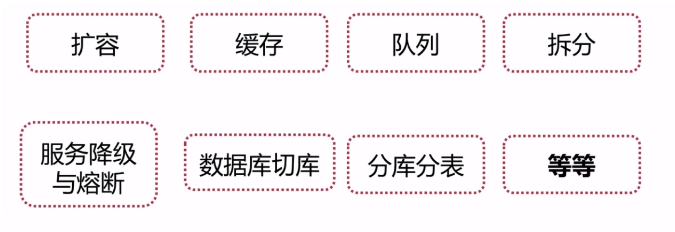
**水平方向：分布式集群**

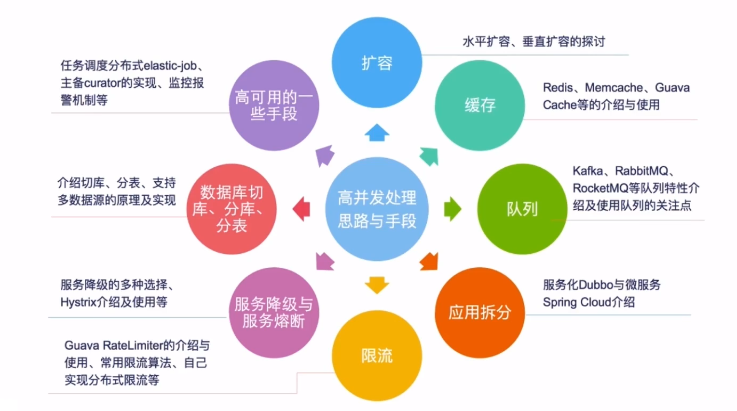
为了解决分布式系统的复杂性问题，一般会用到架构分层和服务拆分，通过分层做隔离，通过微服务解耦。

这个理论上没有上限，只要做好层次和服务划分，加机器扩容就能满足需求，但实际上并非如此，一方面分布式会增加系统复杂性，另一方面集群规模上去之后，也会引入一堆AIOps、服务发现、服务治理的新问题。

因为垂直向的限制，所以，我们通常更关注水平扩展，高并发系统的实施也主要围绕水平方向展开。

另外一种表述，高并发解决思路包括：





互联网分布式架构设计，提高系统并发能力的方式，方法论上主要有两种：垂直扩展（Scale Up）与水平扩展（Scale Out）。

垂直扩展：提升单机处理能力。垂直扩展的方式又有两种：

（1）增强单机硬件性能，例如：增加CPU核数如32核，升级更好的网卡如万兆，升级更好的硬盘如SSD，扩充硬盘容量如2T，扩充系统内存如128G；

（2）提升单机架构性能，例如：使用Cache来减少IO次数，使用异步来增加单服务吞吐量，使用无锁数据结构来减少响应时间；

在互联网业务发展非常迅猛的早期，如果预算不是问题，强烈建议使用“增强单机硬件性能”的方式提升系统并发能力，因为这个阶段，公司的战略往往是发展业务抢时间，而“增强单机硬件性能”往往是最快的方法。

不管是提升单机硬件性能，还是提升单机架构性能，都有一个致命的不足：单机性能总是有极限的。所以互联网分布式架构设计高并发终极解决方案还是水平扩展。

水平扩展：只要增加服务器数量，就能线性扩充系统性能。水平扩展对系统架构设计是有要求的，如何在架构各层进行可水平扩展的设计，以及互联网公司架构各层常见的水平扩展实践。

### 关键技术

#### 集群化

负载均衡负载均衡就是把负载（request）均衡分配到不同的服务实例，利用集群的能力去对抗高并发，负载均衡是服务集群化的实施要素，它分3种：

1. DNS负载均衡，客户端通过URL发起网络服务请求的时候，会去DNS服务器做域名解释，DNS会按一定的策略（比如就近策略）把URL转换成IP地址，同一个URL会被解释成不同的IP地址，这便是DNS负载均衡，它是一种粗粒度的负载均衡，它只用URL前半部分，因为DNS负载均衡一般采用就近原则，所以通常能降低时延，但DNS有cache，所以也会更新不及时的问题。
2. 硬件负载均衡，通过布置特殊的负载均衡设备到机房做负载均衡，比如F5，这种设备贵，性能高，可以支撑每秒百万并发，还能做一些安全防护，比如防火墙。
3. 软件负载均衡，根据工作在ISO 7层网络模型的层次，可分为四层负载均衡（比如章文嵩博士的LVS）和七层负载均衡（NGINX），软件负载均衡配置灵活，扩展性强，阿某云的SLB作为服务对外售卖，Nginx可以对URL的后半部做解释承担API网关的职责。

所以，完整的负载均衡链路是client <-> DNS负载均衡-> F5 -> LVS/SLB -> NGINX

不管选择哪种LB策略，或者组合LB策略，逻辑上，我们都可以视为负载均衡层，通过添加负载均衡层，我们将负载均匀分散到了后面的服务集群，具备基础的高并发能力，但这只是万里长征第一步。

#### 数据库层面：分库分表+读写分离

前面通过负载均衡解决了无状态服务的水平扩展问题，但我们的系统不全是无状态的，后面通常还有有状态的数据库，所以解决了前面的问题，存储有可能成为系统的瓶颈，我们需要对有状态存储做分片路由。

数据库的单机QPS一般不高，也就几千，显然满足不了高并发的要求。所以，我们需要做分库分表 + 读写分离。

就是把一个库分成多个库，部署在多个数据库服务上，主库承载写请求，从库承载读请求。从库可以挂载多个，因为很多场景写的请求远少于读的请求，这样就把对单个库的压力降下来了。如果写的请求上升就继续分库分表，如果读的请求上升就挂更多的从库，但数据库天生不是很适合高并发，而且数据库对机器配置的要求一般很高，导致单位服务成本高，所以，这样加机器抗压力成本太高，还得另外想招。

#### 读多写少：缓存

缓存的理论依据是局部性原理。

一般系统的写入请求远少于读请求，针对写少读多的场景，很适合引入缓存集群。

在写数据库的时候同时写一份数据到缓存集群里，然后用缓存集群来承载大部分的读请求，因为缓存集群很容易做到高性能，所以，这样的话，通过缓存集群，就可以用更少的机器资源承载更高的并发。

缓存的命中率一般能做到很高，而且速度很快，处理能力也强（单机很容易做到几万并发），是理想的解决方案。

CDN本质上就是缓存，被用户大量访问的静态资源缓存在CDN中是目前的通用做法。

缓存也有很多需要谨慎处理的问题：

1. 一致性问题：

更新db成功+更新cache失败->不一致

更新db失败+更新cache成功->不一致

更新db成功+淘汰缓存失败->不一致

1. 缓存穿透：查询一定不存在的数据，会穿透缓存直接压到数据库，从而导致缓存失去作用，如果有人利用这个漏洞，大量查询一定不存在的数据，会对数据库造成压力，甚至打挂数据库。解决方案：布隆过滤器 或者 简单的方案，查询不存在的key，也把空结果写入缓存（设置较短的过期淘汰时间），从而降低命失
2. 缓存雪崩：如果大量缓存在一个时刻同时失效，则请求会转到DB，则对DB形成压迫，导致雪崩。简单的解决方案是为缓存失效时间添加随机值，降低同一时间点失效淘汰缓存数，避免集体失效事件发生。

但缓存是针对读，如果写的压力很大，怎么办？

#### 高写入：消息中间件

同理，通过跟主库加机器，耗费的机器资源是很大的，这个就是数据库系统的特点所决定的。

相同的资源下，数据库系统太重太复杂，所以并发承载能力就在几千/s的量级，所以此时你需要引入别的一些技术。

比如说消息中间件技术，也就是MQ集群，它是非常好的做写请求异步化处理，实现削峰填谷的效果。

消息队列能做解耦，在只需要最终一致性的场景下，很适合用来配合做流控。

假如说，每秒是1万次写请求，其中比如5千次请求是必须请求过来立马写入数据库中的，但是另外5千次写请求是可以允许异步化等待个几十秒，甚至几分钟后才落入数据库内的。

那么此时完全可以引入消息中间件集群，把允许异步化的每秒5千次请求写入MQ，然后基于MQ做一个削峰填谷。比如就以平稳的1000/s的速度消费出来然后落入数据库中即可，此时就会大幅度降低数据库的写入压力。

业界有很多著名的消息中间件，比如ZeroMQ，rabbitMQ，kafka等。

消息队列本身也跟缓存系统一样，可以用很少的资源支撑很高的并发请求，用它来支撑部分允许异步化的高并发写入是很合适的，比使用数据库直接支撑那部分高并发请求要减少很多的机器使用量。

#### 避免挤兑：流控

再强大的系统，也怕流量短事件内集中爆发，就像银行怕挤兑一样，所以，高并发另一个必不可少的模块就是流控。流控的关键是流控算法，有4种常见的流控算法。

1. 计数器算法（固定窗口）：计数器算法是使用计数器在周期内累加访问次数，当达到设定的限流值时，触发限流策略，下一个周期开始时，进行清零，重新计数，实现简单。计数器算法方式限流对于周期比较长的限流，存在很大的弊端，有严重的临界问题。
2. 滑动窗口算法：将时间周期分为N个小周期，分别记录每个小周期内访问次数，并且根据时间滑动删除过期的小周期，当滑动窗口的格子划分的越多，那么滑动窗口的滚动就越平滑，限流的统计就会越精确。此算法可以很好的解决固定窗口算法的临界问题。
3. 漏桶算法：访问请求到达时直接放入漏桶，如当前容量已达到上限（限流值），则进行丢弃（触发限流策略）。漏桶以固定的速率进行释放访问请求（即请求通过），直到漏桶为空。分布式环境下实施难度高。
4. 令牌桶算法：程序以r（r=时间周期/限流值）的速度向令牌桶中增加令牌，直到令牌桶满，请求到达时向令牌桶请求令牌，如获取到令牌则通过请求，否则触发限流策略。分布式环境下实施难度高。

### 实践经验

接入-逻辑-存储是经典的互联网后端分层，但随着业务规模的提高，逻辑层的复杂度也上升了，所以，针对逻辑层的架构设计也出现很多新的技术和思路，常见的做法包括系统拆分，微服务。除此之外，也有很多业界的优秀实践，包括某信服务器通过协程（无侵入，已开源libco）改造，极大的提高了系统的并发度和稳定性，另外，缓存预热，预计算，批量读写（减少IO），池技术等也广泛应用在实践中，有效的提升了系统并发能力。为了提升并发能力，逻辑后端对请求的处理，一般会用到生产者-消费者多线程模型，即I/O线程负责网络IO，协议编解码，网络字节流被解码后产生的协议对象，会被包装成task投入到task queue，然后worker线程会从该队列取出task执行，有些系统会用多进程而非多线程，通过共享存储，维护2个方向的shm queue，一个input q，一个output q，为了提高并发度，有时候会引入协程，协程是用户线程态的多执行流，它的切换成本更低，通常有更好的调度效率。另外，构建漏斗型业务或者系统，从客户端请求到接入层，到逻辑层，到DB层，层层递减，过滤掉请求，Fail Fast（尽早发现尽早过滤），嘴大屁眼小，哈哈。漏斗型系统不仅仅是一个技术模型，它也可以是一个产品思维，配合产品的用户分流，逻辑分离，可以构建全方位的立体模型。

# 语言并发

## C++

### 多进程

### 多线程

### 互斥量

### unique\_lock

### 条件变量

#### condition\_variable

#### wait

#### notify\_one

#### notify\_all

#### std::async/std::future

#### std::packaged\_task

#### std::promise

#### std::atomic

## Java

### Synchronize

Synchronize是轻量级的锁，mutex属于重型锁（之所以叫重量级锁，因为需要用户态和内核态切换）。

在JDK1.4之前synchronize都是基于重量级锁mutex实现。后来使用自旋锁和mutex结合（先自旋几次，此时不会进入内核态，拿不到锁再使用mutex，加锁失败会挂起线程，让出CPU）。

**锁膨胀问题：**

### Reentrantlock

# I/O多路复用

高并发服务器模型：

1. 多进程并发服务器
2. 多线程并发服务器
3. 多路IO复用服务器：select并发服务器、poll并发服务器、epoll并发服务器

## select

## poll

## epoll

# 扩容

# 内存/内存池

多个客户端同时访问服务器，后台服务器需要对每个客户端的进程分配内存空间，这里可以进行内存的优化，比如内存池、tcmalloc。

# 缓存

## CPU多级缓存

## 分类

本地缓存：编程实现（成员变量、局部变量、静态变量）、Guava Cache

分布式缓存：Redis、Memcache

## 特征

命中率：命中数/（命中数+没有命中数）

最大元素（空间）：缓存中可以存储的最大元素

缓存清空策略：FIFO，LFU（最少使用），LRU（最近最少使用，保证热点数据），过期时间，随机等

缓存更新（缓存同步）：缓存time out。如果缓存失效，重新去数据库查询，实时性比较差。一旦数据库中数据更新，立即通知前端的缓存更新，实时性比较高。

缓存换页：内存不够，将不活跃的数据换出内存。FIFO，LRU（最近最少使用），LFU（最不频繁使用。）

## 缓存命中率

缓存命中率影响因素：

1. 业务场景和业务需求：缓存适用于读多写少的场景；
2. 缓存的设计（粒度和策略）；
3. 缓存容量和基础设施；

注：一些异常场景，比如节点失效的场景也是需要考虑的，可以采用一致性哈希算法或者节点冗余的策略解决。

## 缓存一致性



## 缓存并发问题

### 缓存穿透问题

### 缓存雪崩现象

# 分布式缓存

将热点数据存储到缓存。

## Redis

## memcache

# 队列+连接池

## 背景

New thread的弊端：

1. 每次new Thread新建对象，性能差
2. 线程缺乏统一管理，可能无限制的新建线程，相互竞争，有可能占用过多系统资源导致死机或OOM；
3. 缺少更多功能，如更多执行、定期执行、线程中断

## 优点

线程池的好处：

1. 重用存在的线程，减少对象创建、消亡的开销，性能佳；
2. 可有效控制最大并发线程数，提高系统资源利用率，同时可以避免过多资源竞争，避免阻塞；
3. 提供定时执行、定期执行、单线程、并发数控制等功能

# 消息队列

## 背景

生产者和消费者速度或稳定性等因素的不一致。

## 特性

消息队列的特性：

1. 业务无关：只做消息分发
2. FIFO：先投递先到达
3. 容灾：节点的动态增删和消息的持久化
4. 性能：吞吐量提升，系统内部通信效率提高

优点：

1. 业务耦合；
2. 最终一致性（记录+补偿机制实现）；
3. 广播；
4. 错峰与流控。

# muduo

《Linux多线程服务器编程：使用muduo C++网络库》

博客：http://blog.csdn.net/solstice/

# Libevent

## 概述

Libevevt是一款事件驱动的网络开发包，由于使用C语言开发，体积小，跨平台，速度极快。大量开源项目使用了libevevt，比如谷歌浏览器和分布式的高速缓存系统Memcached。

Libenent支持kqueue、select、poll epoll、iocp等非阻塞的socket。内部事件机制完全独立于公开的事件API，libevent支持跨平台，可以在Linux、\*BSD、Mac OS X，Solaris、Windows等平台编译。

注：libevent是封装的socket（分为阻塞和非阻塞/IO多路复用），是采用基于事件的非阻塞方式。

## 原理

### 接口

### 网络模型

## event事件

### 事件驱动

### 事件IO

### 事件处理流程和状态转换

## 使用

### bufferevent

### http接口

# 服务降级与熔断

## 服务降级

自动降级：超时、失败次数、故障、限流

人工降级：秒杀、双11大促等

服务降级要考虑的问题：

1. 核心服务、非核心服务
2. 是否支持降级，降级策略
3. 业务放通场景，策略

## 服务熔断

## 联系区别

共享：目的、最终表现、粒度、自治

区别：触发原因、管理目标层次、实现方式

# 应用拆分



## 原则

应用拆分原则：

1. 业务优先；
2. 循序渐进；
3. 兼顾技术：重构、分层；
4. 可靠性测试

## 设计

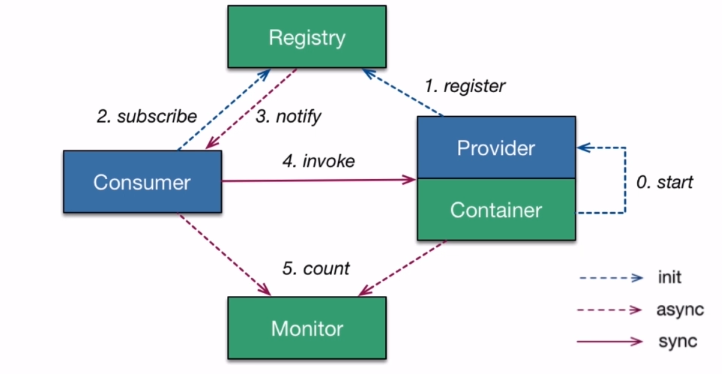
应用之间的通信：RPC（dubbo）、消息队列

应用之间数据库设计：每个应用都有独立的数据库

避免事务操作跨应用

## 应用框架

### Dubbo



### Spring cloud微服务



# 应用限流

## 算法

### 计数器法

### 滑动窗口

### 漏桶算法

### 令牌桶算法

## 选择

# 数据库切库分库分表

数据库瓶颈：

1. 单个库数据量太大（1T~2T）：多个库
2. 单个数据库服务器压力过大、读写瓶颈：多个库
3. 单个表数据量过大：分表

## 切库

切库的基础及实际应用：读写分离

## 分表

# 互联网分层



常见互联网分布式架构如上，分为：

（1）客户端层：典型调用方是浏览器browser或者手机应用APP

（2）反向代理层：系统入口，反向代理

（3）站点应用层：实现核心应用逻辑，返回html或者json

（4）服务层：如果实现了服务化，就有这一层

（5）数据-缓存层：缓存加速访问存储

（6）数据-数据库层：数据库固化数据存储

前端负载均衡（客户端层+反向代理层+站点应用层）：

1. DNS负载均衡

在DNS服务器中，可以为多个不同的地址配置同一个名字，对于不同的客户机访问同一个名字，得到不同的地址。

1. 反向代理

使用代理服务器将请求发给内部服务器，让代理服务器将请求均匀转发给多态内部Web服务器之一，从而达到负载均衡的目的。标准代理方式是客户使用代理访问多个外部Web服务器，而这种代理方式是多个客户使用它访问内部Web服务器，因此也被称为反向代理模式。

1. 基于NAT的负载均衡技术
2. LVS
3. F5硬件负载均衡

## 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”实现的：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成为瓶颈的时候，只要增加服务器数量，新增nginx服务的部署，增加一个外网ip，就能扩展反向代理层的性能，做到理论上的无限高并发。

## 站点层的水平扩展



站点层的水平扩展，是通过“nginx”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，当服务成为瓶颈的时候，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，做到理论上的无限高并发。如果需要优雅的进行服务层自动扩容，这里可能需要配置中心里服务自动发现功能的支持。

## 数据层的水平扩展

在数据量很大的情况下，数据层（缓存，数据库）涉及数据的水平扩展，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。

互联网数据层常见的水平拆分方式有这么几种，以数据库为例：

### 按照范围水平拆分



每一个数据服务，存储一定范围的数据，上图为例：

user0库，存储uid范围1-1kw

user1库，存储uid范围1kw-2kw

这个方案的好处：

（1）规则简单，service只需判断一下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，可以随时加一个uid[2kw,3kw]的数据服务；

不足：

（1）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活跃，大range的服务请求压力会更大；

### 按照哈希水平拆分



每一个数据库，存储某个key值hash后的部分数据，上图为例：

user0库，存储偶数uid数据

user1库，存储奇数uid数据

这个方案的好处：

（1）规则简单，service只需对uid进行hash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：

（1）不容易扩展，扩展一个数据服务，hash方法改变时候，可能需要进行数据迁移；

这里需要注意的是，通过水平拆分来扩充系统性能，与主从同步读写分离来扩充数据库性能的方式有本质的不同。

通过水平拆分扩展数据库性能：

（1）每个服务器上存储的数据量是总量的1/n，所以单机的性能也会有提升；

（2）n个服务器上的数据没有交集，那个服务器上数据的并集是数据的全集；

（3）数据水平拆分到了n个服务器上，理论上读性能扩充了n倍，写性能也扩充了n倍（其实远不止n倍，因为单机的数据量变为了原来的1/n）；

通过主从同步读写分离扩展数据库性能：

（1）每个服务器上存储的数据量是和总量相同；

（2）n个服务器上的数据都一样，都是全集；

（3）理论上读性能扩充了n倍，写仍然是单点，写性能不变；

缓存层的水平拆分和数据库层的水平拆分类似，也是以范围拆分和哈希拆分的方式居多。

## 总结

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

提高系统并发能力的方式，方法论上主要有两种：垂直扩展（Scale Up）与水平扩展（Scale Out）。前者垂直扩展可以通过提升单机硬件性能，或者提升单机架构性能，来提高并发性，但单机性能总是有极限的，互联网分布式架构设计高并发终极解决方案还是后者：水平扩展。

互联网分层架构中，各层次水平扩展的实践又有所不同：

（1）反向代理层可以通过“DNS轮询”的方式来进行水平扩展；

（2）站点层可以通过nginx来进行水平扩展；

（3）服务层可以通过服务连接池来进行水平扩展；

（4）数据库可以按照数据范围，或者数据哈希的方式来进行水平扩展；

各层实施水平扩展后，能够通过增加服务器数量的方式来提升系统的性能，做到理论上的性能无限。

# 高并发方案

高并发解决思路与手段包括：扩容，缓存，消息队列，应用拆分，服务降级与熔断，分库分表，数据库切库。

## LVS

## keepalive

## Nginx

# 数据库高并发

## 数据库锁

## 写时复制

## MVCC

## 读写分离

## 负载均衡

# 分布式高并发多线程

当提起这三个词的时候，是不是很多人都认为分布式=高并发=多线程？

当面试官问到高并发系统可以采用哪些手段来解决，或者被问到分布式系统如何解决一致性的问题，是不是一脸懵逼？

确实，在一开始接触的时候，不少人都会将三者混淆，误以为所谓的分布式高并发的系统就是能同时供海量用户访问，而采用多线程手段不就是可以提供系统的并发能力吗？

实际上，他们三个总是相伴而生，但侧重点又有不同。

## 什么是分布式？

分布式更多的一个概念，是为了解决单个物理服务器容量和性能瓶颈问题而采用的优化手段。

该领域需要解决的问题极多，在不同的技术层面上，又包括：分布式文件系统、分布式缓存、分布式数据库、分布式计算等，一些名词如Hadoop、zookeeper、MQ等都跟分布式有关。

从理念上讲，分布式的实现有两种形式：

水平扩展：当一台机器扛不住流量时，就通过添加机器的方式，将流量平分到所有服务器上，所有机器都可以提供相当的服务；

垂直拆分：前端有多种查询需求时，一台机器扛不住，可以将不同的需求分发到不同的机器上，比如A机器处理余票查询的请求，B机器处理支付的请求。

## 什么是高并发？

相对于分布式来讲，高并发在解决的问题上会集中一些，其反应的是同时有多少量：比如在线直播服务，同时有上万人观看。

高并发可以通过分布式技术去解决，将并发流量分到不同的物理服务器上。

但除此之外，还可以有很多其他优化手段：比如使用缓存系统，将所有的，静态内容放到CDN等；还可以使用多线程技术将一台服务器的服务能力最大化。

## 什么是多线程？

多线程是指从软件或者硬件上实现多个线程并发执行的技术，它更多的是解决CPU调度多个进程的问题，从而让这些进程看上去是同时执行（实际是交替运行的）。

这几个概念中，多线程解决的问题是最明确的，手段也是比较单一的，基本上遇到的最大问题就是线程安全。

在JAVA语言中，需要对JVM内存模型、指令重排等深入了解，才能写出一份高质量的多线程代码。

## 总结

分布式是从物理资源的角度去将不同的机器组成一个整体对外服务，技术范围非常广且难度非常大，有了这个基础，高并发、高吞吐等系统很容易构建；

高并发是从业务角度去描述系统的能力，实现高并发的手段可以采用分布式，也可以采用诸如缓存、CDN等，当然也包括多线程；

多线程则聚焦于如何使用编程语言将CPU调度能力最大化。

分布式与高并发系统，涉及到大量的概念和知识点，如果没有系统的学习，很容易会杂糅概念而辨识不清，在面试与实际工作中都会遇到困难。