# 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。

## 指标

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。

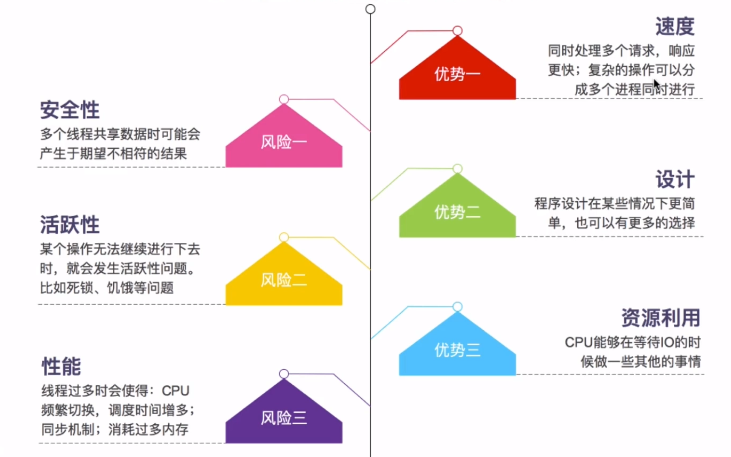
响应时间：系统对请求做出响应的时间。例如系统处理一个HTTP请求需要200ms，这个200ms就是系统的响应时间。

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。

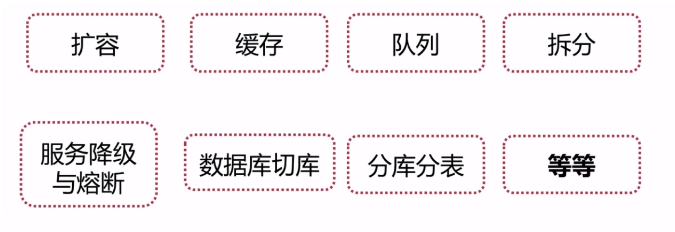
并发用户数：同时承载正常使用系统功能的用户数量。例如一个即时通讯系统，同时在线量一定程度上代表了系统的并发用户数。

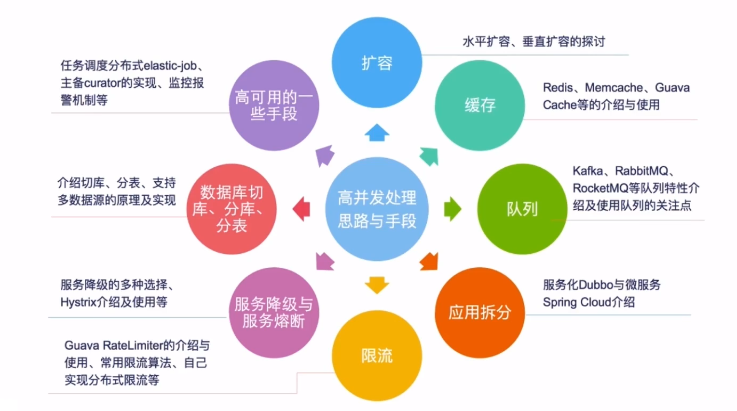
## 优缺点



## 提升并发

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。

# 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

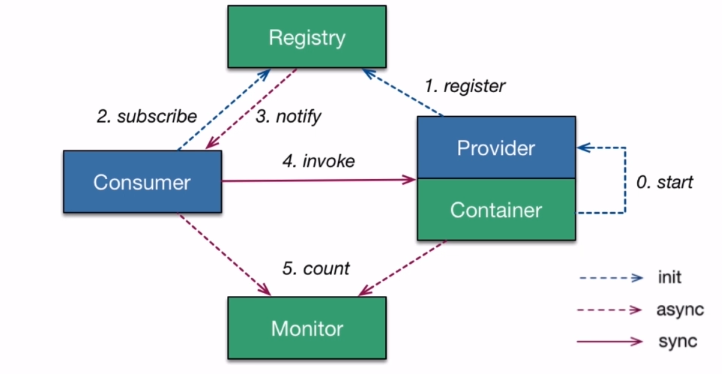
应用之间的通信：RPC（dubbo）、消息队列

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。