# 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。

## 并发与并行

### 并发VS并行

并发：一个处理器同时处理多个任务

并行：多个处理器或者是多核的处理器同时处理多个不同的任务

### 同步VS异步

**同步和异步的区别：**

同步：执行某个操作开始后就一直等着按部就班的直到操作结束

异步：执行某个操作后立即离开，后面有响应的话再来通知执行者

接着我们再了解一个重要的概念：

临界区：公共资源或者共享数据

由于共享数据的出现，必然会导致竞争，所以我们需要再了解一下：

阻塞：某个操作需要的共享资源被占用了，只能等待，称为阻塞

非阻塞：某个操作需要的共享资源被占用了，不等待立即返回，并携带错误信息回去，期待重试

如果两个操作都在等待某个共享资源而且都互不退让就会造成死锁：

死锁：参考著名的哲学家吃饭问题

饥饿：饥饿的哲学家等不齐筷子吃饭

活锁：相互谦让而导致阻塞无法进入下一步操作，跟死锁相反，死锁是相互竞争而导致的阻塞

### 并发级别

理想情况下我们希望所有线程都一起并行飞起来。但是CPU数量有限，线程源源不断，总得有个先来后到，不同场景需要的并发需求也不一样，比如秒杀系统我们需要很高的并发程度，但是对于一些下载服务，我们需要的是更快的响应，并发反而是其次的。所以我们也定义了并发的级别，来应对不同的需求场景。

阻塞：阻塞是指一个线程进入临界区后，其它线程就必须在临界区外等待，待进去的线程执行完任务离开临界区后，其它线程才能再进去。

无饥饿：线程排队先来后到，不管优先级大小，先来先执行，就不会产生饥饿等待资源，也即公平锁；相反非公平锁则是根据优先级来执行，有可能排在前面的低优先级线程被后面的高优先级线程插队，就形成饥饿

无障碍：共享资源不加锁，每个线程都可以自有读写，单监测到被其他线程修改过则回滚操作，重试直到单独操作成功；风险就是如果多个线程发现彼此修改了，所有线程都需要回滚，就会导致死循环的回滚中，造成死锁

无锁：无锁是无障碍的加强版，无锁级别保证至少有一个线程在有限操作步骤内成功退出，不管是否修改成功，这样保证了多个线程回滚不至于导致死循环

无等待：无等待是无锁的升级版，并发编程的最高境界，无锁只保证有线程能成功退出，但存在低级别的线程一直处于饥饿状态，无等待则要求所有线程必须在有限步骤内完成退出，让低级别的线程有机会执行，从而保证所有线程都能运行，提高并发度。

## 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

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

吞吐量：单位时间内处理的请求数量。TPS、QPS都是吞吐量的常用量化指标。

#### 系统吞吐量要素

一个系统的吞吐量（承压能力）与request（请求）对cpu的消耗，外部接口，IO等等紧密关联。

单个request 对cpu消耗越高，外部系统接口，IO影响速度越慢，系统吞吐能力越低，反之越高。

#### 重要参数

QPS(TPS),并发数，响应时间

QPS(TPS)：每秒钟request/事务 数量

并发数：系统同时处理的request/事务数

响应时间：一般取平均响应时间

**关系**

QPS(TPS)=并发数/平均响应时间

一个系统吞吐量通常有QPS(TPS),并发数两个因素决定，每套系统这个两个值都有一个相对极限值，在应用场景访问压力下，只要某一项达到系统最高值，系统吞吐量就上不去了，如果压力继续增大，系统的吞吐量反而会下降，原因是系统超负荷工作，上下文切换，内存等等其他消耗导致系统性能下降。

#### 系统吞吐量评估

我们在做系统设计的时候就需要考虑CPU运算，IO，外部系统响应因素造成的影响以及对系统性能的初步预估。

而通常情况下，我们面对需求，我们评估出来的出来QPS，并发数之外，还有另外一个维度：日PV。

通过观察系统的访问日志发现，在用户量很大的情况下，各个时间周期内的同一时间段的访问流量几乎一样。比如工作日的每天早上。只要能拿到日流量图和QPS我们就可以推算日流量。

通常的技术方法：

1、找出系统的最高TPS和日PV，这两个要素有相对比较稳定的关系（除了放假、季节性因素影响之外）

2、通过压力测试或者经验预估，得出最高TPS，然后跟进1的关系，计算出系统最高的日吞吐量。B2B中文和淘宝面对的客户群不一样，这两个客户群的网络行为不应用，他们之间的TPS和PV关系比例也不一样。

### QPS

QPS：每秒响应请求数，是一台服务器每秒能够相应的查询次数，是对一个特定的查询服务器在规定时间内所处理流量多少的衡量标准, 即每秒的响应请求数，也即是最大吞吐能力。

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

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。

### TPS

Transactions Per Second，也就是事务数/秒。一个事务是指一个客户机向服务器发送请求然后服务器做出反应的过程。客户机在发送请求时开始计时，收到服务器响应后结束计时，以此来计算使用的时间和完成的事务个数。

**TPS与QPS：**

1、TPS即每秒处理事务数，包括了

用户请求服务器

服务器自己的内部处理

服务器返回给用户

这三个过程，每秒能够完成N个这三个过程，TPS也就是N；

1. QPS基本类似于TPS，但是不同的是，对于一个页面的一次访问，形成一个TPS；但一次页面请求，可能产生多次对服务器的请求，服务器对这些请求，就可计入“QPS”之中。

### PV

PV（Page View）：页面访问量，即页面浏览量或点击量，用户每次刷新即被计算一次。可以统计服务一天的访问日志得到。

### UV

UV（Unique Visitor）：独立访客，统计1天内访问某站点的用户数。可以统计服务一天的访问日志并根据用户的唯一标识去重得到。响应时间（RT）：响应时间是指系统对请求作出响应的时间，一般取平均响应时间。可以通过Nginx、Apache之类的Web Server得到。

### DAU

DAU(Daily Active User)，日活跃用户数量。常用于反映网站、互联网应用或网络游戏的运营情况。DAU通常统计一日（统计日）之内，登录或使用了某个产品的用户数（去除重复登录的用户），与UV概念相似。

### MAU

MAU(Month Active User)：月活跃用户数量，指网站、app等去重后的月活跃用户数量

### 关注性能指标

软件做性能测试时需要关注哪些性能呢？

首先，开发软件的目的是为了让用户使用，我们先站在用户的角度分析一下，用户需要关注哪些性能。

对于用户来说，当点击一个按钮、链接或发出一条指令开始，到系统把结果已用户感知的形式展现出来为止，这个过程所消耗的时间是用户对这个软件性能的直观印 象。也就是我们所说的响应时间，当相应时间较小时，用户体验是很好的，当然用户体验的响应时间包括个人主观因素和客观响应时间，在设计软件时，我们就需要 考虑到如何更好地结合这两部分达到用户最佳的体验。如：用户在大数据量查询时，我们可以将先提取出来的数据展示给用户，在用户看的过程中继续进行数据检 索，这时用户并不知道我们后台在做什么。

用户关注的是用户操作的相应时间。

其次，我们站在管理员的角度考虑需要关注的性能点。

1、 响应时间

2、 服务器资源使用情况是否合理

3、 应用服务器和数据库资源使用是否合理

4、 系统能否实现扩展

5、 系统最多支持多少用户访问、系统最大业务处理量是多少

6、 系统性能可能存在的瓶颈在哪里

7、 更换那些设备可以提高性能

8、 系统能否支持7×24小时的业务访问

再次，站在开发（设计）人员角度去考虑。

1、 架构设计是否合理

2、 数据库设计是否合理

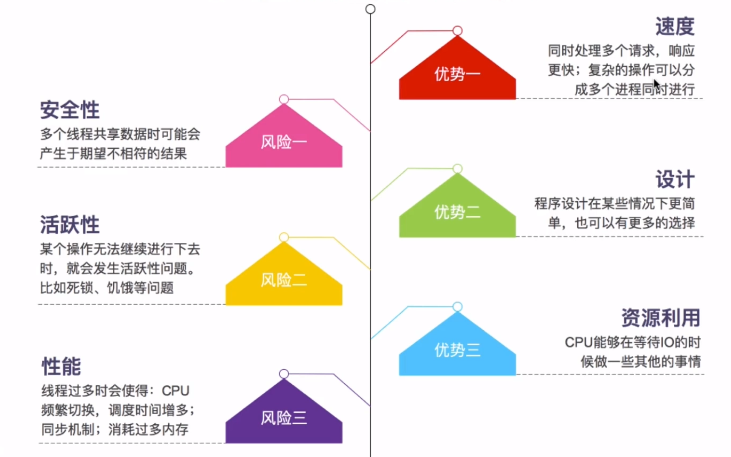
3、 代码是否存在性能方面的问题

4、 系统中是否有不合理的内存使用方式

5、 系统中是否存在不合理的线程同步方式

6、 系统中是否存在不合理的资源竞争

## 优缺点



## 内存模型

宏观上分布式系统需要解决的首要问题是数据一致性，同样，微观上并发编程要解决的首要问题也是数据一致性。貌似我们搞了这么多年的斗争都是在公关一致性这个世界性难题。既然并发编程要从微观开始，那么我们肯定要对CPU和内存的工作机理有所了解，尤其是数据在CPU和内存直接的传输机制。

### 整体原则

探究内存模型之前我们要抛出三个概念:

**原子性**

在32位的系统中，对于4个字节32位的Integer的操作对应的JVM指令集映射到汇编指令为一个原子操作，所以对Integer类型的数据操作是原子性，但是Long类型为8个字节64位，32位系统要分为两条指令来操作，所以不是原子操作。

对于32位操作系统来说，单次次操作能处理的最长长度为32bit，而long类型8字节64bit，所以对long的读写都要两条指令才能完成（即每次读写64bit中的32bit）

**可见性**

线程修改变量对其他线程即时可见

**有序性**

串行指令顺序唯一，并行线程直接指令可能出现不一致，也即是指令被重排了

而指令重排也是有一定原则(摘自《深入理解Java虚拟机第12章》)：

程序次序规则：一个线程内，按照代码顺序，书写在前面的操作先行发生于书写在后面的操作；

锁定规则：一个unLock操作先行发生于后面对同一个锁额lock操作；

volatile变量规则：对一个变量的写操作先行发生于后面对这个变量的读操作；

传递规则：如果操作A先行发生于操作B，而操作B又先行发生于操作C，则可以得出操作A先行发生于操作C；

线程启动规则：Thread对象的start()方法先行发生于此线程的每个一个动作；

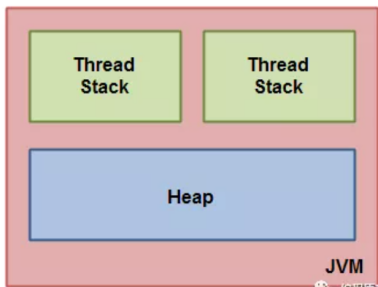
线程中断规则：对线程interrupt()方法的调用先行发生于被中断线程的代码检测到中断事件的发生；

线程终结规则：线程中所有的操作都先行发生于线程的终止检测，我们可以通过Thread.join()方法结束、Thread.isAlive()的返回值手段检测到线程已经终止执行；

对象终结规则：一个对象的初始化完成先行发生于他的finalize()方法的开始。

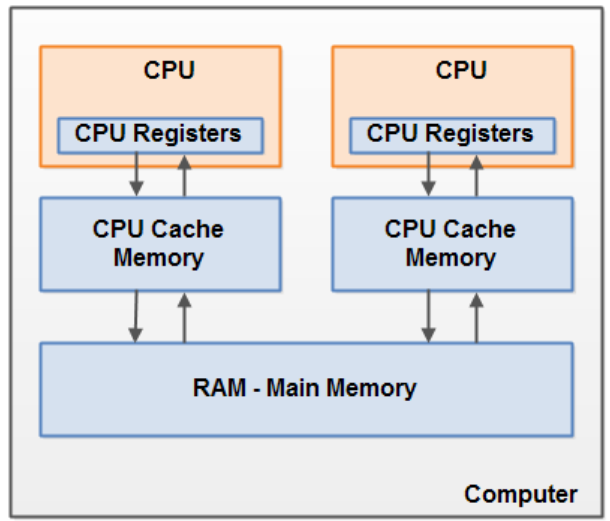
### 逻辑内存

我们谈的逻辑内存也即是JVM的内存格局。JVM将操作系统提供的物理内存和CPU缓存在逻辑分为堆，栈，方法区，和程序计数器。并发编程我们主要关注的是堆栈的分配，因为线程都是寄生在栈里面的内存段，把栈里面的方法逻辑读取到CPU进行运算。

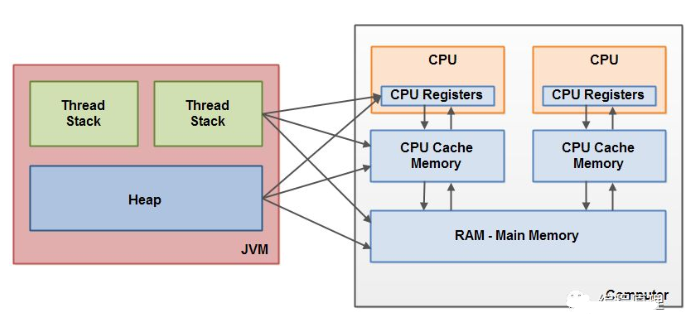


### 物理内存

而实际的物理内存包含了主存和CPU的各级缓存还有寄存器，而为了计算效率，CPU往往回就近从缓存里面读取数据。在并发的情况下就会造成多个线程之间对共享数据的错误使用。



### 内存映射

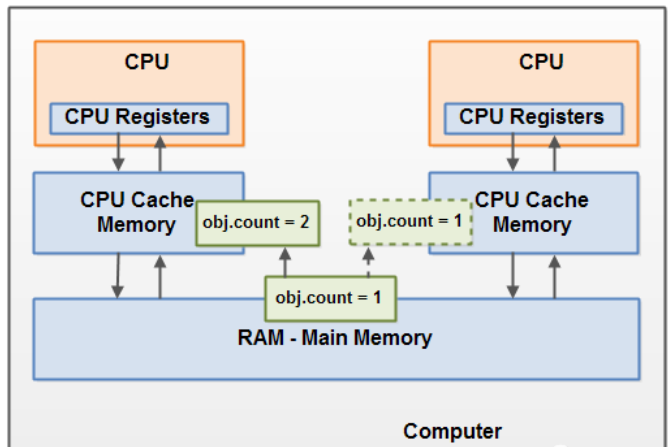


由于可能发生对象的变量同时出现在主存和CPU缓存中，就可能导致了如下问题：

线程修改的变量对外可见

读写共享变量时出现竞争资源

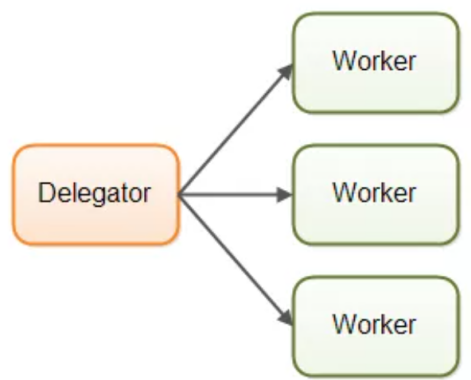
由于线程内的变量对栈外是不可见的，但是成员变量等共享资源是竞争条件，所有线程可见，就会出现如下当一个线程从主存拿了一个变量1修改后变成2存放在CPU缓存，还没来得及同步回主存时，另外一个线程又直接从主存读取变量为1，这样就出现了脏读。



现在我们弄清楚了线程同步过程数据不一致的原因，接下来要解决的目标就是如何避免这种情况的发生，经过大量的探索和实践，我们从概念上不断的革新比如并发模型的流水线化和无状态函数式化，而且也提供了大量的实用工具。接下来我们从无到有，先了解最简单的单个线程的一些特点，弄清楚一个线程有多少能耐后，才能深刻认识多个线程一起打交道会出现什么幺蛾子。

## 并发模型

### 多线程模式

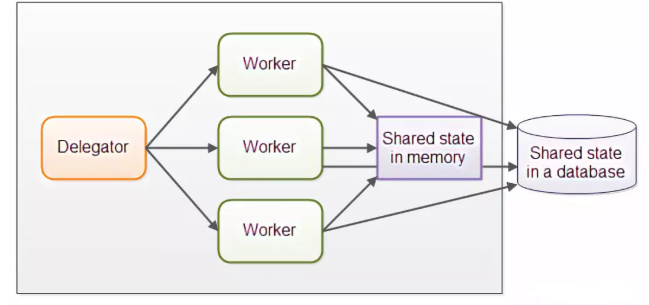


单位线程完成完整的任务，也即是一条龙服务线程。

优势：

映射现实单一任务，便于理解和编码

劣势：



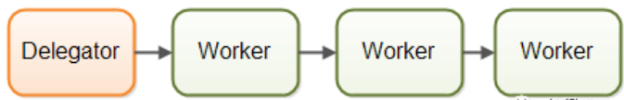
1、有状态多线程共享资源，导致资源竞争，死锁问题，线程等待阻塞，失去并发意义

2、有状态多线程非阻塞算法，有利减少竞争，提升性能，但难以实现

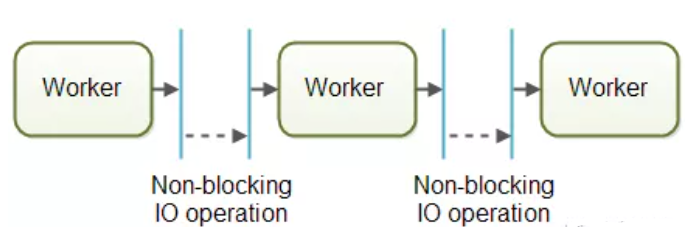
3、多线程执行顺序无法预知

### 流水线模型

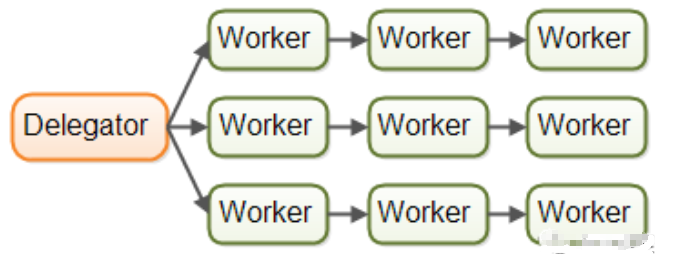
传统的多线程工作模式，理解起来很直观，接下来我们要介绍的另外一种并发模式看起来就不那么直观了。



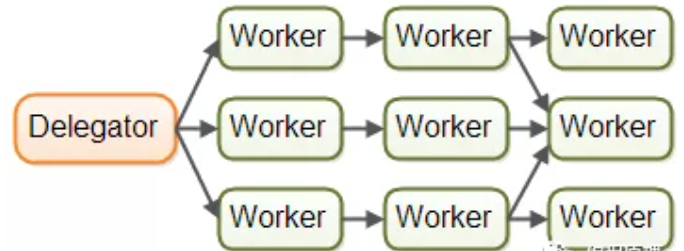
流水线模型，特点是无状态线程，无状态也意味着无需竞争共享资源，无需等待，也就是非阻塞模型。流水线模型顾名思义就是流水线上有多个环节，每个环节完成自己的工作后就交给下一个环节，无需等待上游，周而复始的完成自己岗位上的一亩三分地就行。各个环节之间交付无需等待，完成即可交付。



而工厂的流水线也不止一条，所以有多条流水线同时工作。



不同岗位的生产效率是不一样的，所以不同流水线之间也可以发生协同。

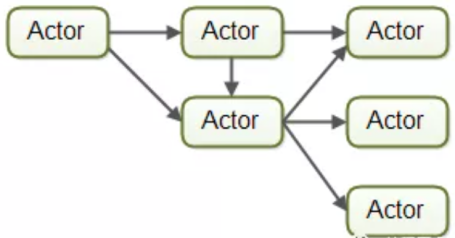


我们说流水线模型也称为响应式模型或者事件驱动模型，其实就是流水线上上游岗位完成生产就通知下游岗位，所以完成了一个事件的通知，每完成一次就通知一下，就是响应式的意思。

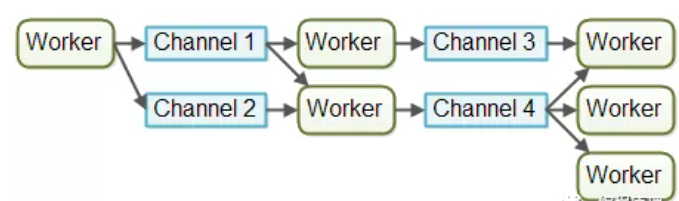
流水线模型总体的思想就是纵向切分任务，把任务里面耗时过久的环节单独隔离出来，避免完成一个任务需要耗费等待的时间。在实现上又分为Actors和Channels模型。

#### Actors模型

该模型跟我们讲述的流水线模型基本一致，可以理解为响应式模型：



#### Channels模型



由于各个环节直接不直接交互，所以上下游之间并不知道对方是谁，好比不同环节直接用的是几条公共的传送带来接收物品，各自只需要把完成后的半成品扔到传送带，即使后面流水线优化了，去掉中间的环节，对于个体岗位来说也是无感知的，它只是周而复始的从传送带拿物品来加工。

**流水线的优缺点：**

**优势：**

无共享状态：无需考虑资源抢占，死锁等问题

独享内存：worker可以持有内存，合并多次操作到内存后再持久化，提升效率

贴合底层：单线程模式贴合硬件运行流程，便于代码维护

任务顺序可预知

**劣势：**

不够直观：一个任务被拆分为流水线上多个环节，代码层面难以直观理解业务逻辑

由于流水线模式跟人类的顺序执行思维不一样，比较费解，那么有没有办法让我们编码的时候像写传统的多线程代码一样，而运行起来又是流水线模式呢？答案是肯定的，比如基于Java的Akka/Reator/Vert.x/Play/Qbit框架，或者golang就是为流水线模式而生的并发语言，还有nodeJS等等。

其实流水线模型背后用的也还是多线程来实现，只不过对于传统多线程模式下我们需要小心翼翼来处理跟踪资源共享问题，而流水线模式把以前一个线程做的事情拆成多个，每一个环节再用一条线程来完成，避免共享，线程直接通过管道传输消息。

这一块展开也是一个专题，主要设计NIO，Netty和Akka的编程实践。

### 函数式模型

函数式并行模型类似流水线模型，单一的函数是无状态的，所以避免了资源竞争的复杂度，同时每个函数类似流水线里面的单一环境，彼此直接通过函数调用传递参数副本，函数之外的数据不会被修改。函数式模式跟流水线模式相辅相成逐渐成为更为主流的并发架构。

## 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。

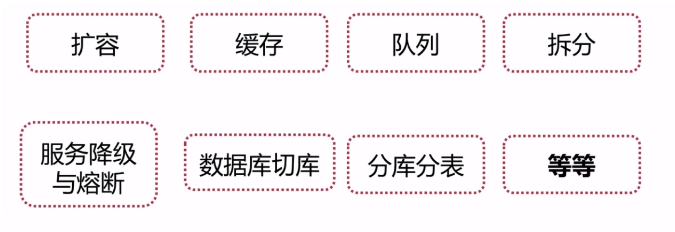
**水平方向：分布式集群**

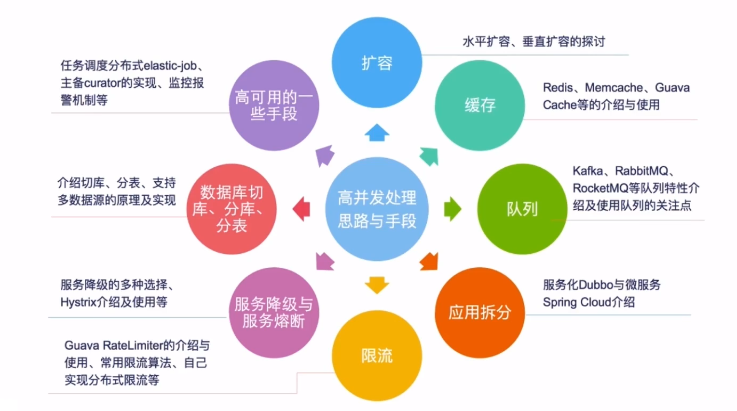
为了解决分布式系统的复杂性问题，一般会用到架构分层和服务拆分，通过分层做隔离，通过微服务解耦。

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

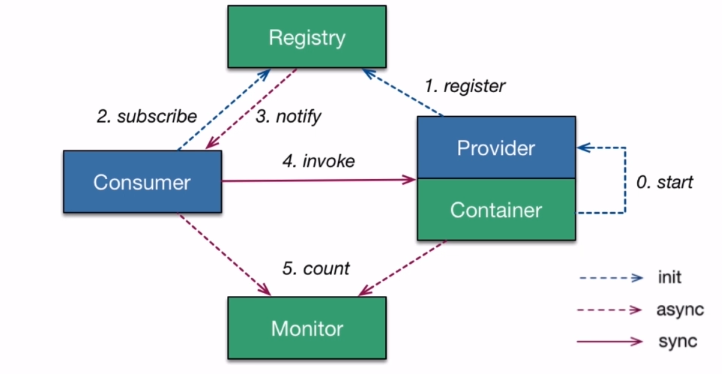
应用之间的通信：RPC（dubbo）、消息队列

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

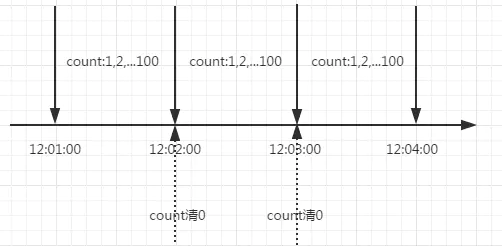
限流的常用处理手段有：计数器、滑动窗口、漏桶、令牌。

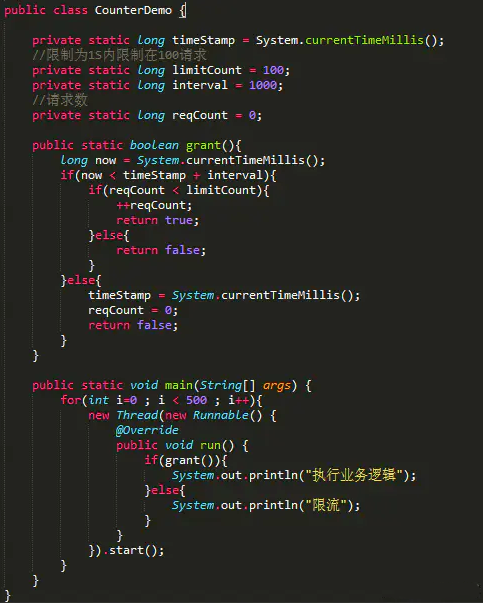
## 算法

### 计数器法

<https://blog.csdn.net/qq_35869079/article/details/88243754>

计数器是一种比较简单的限流算法，用途比较广泛，在接口层面，很多地方使用这种方式限流。在一段时间内，进行计数，与阀值进行比较，到了时间临界点，将计数器清0。





这里需要注意的是，存在一个时间临界点的问题。举个栗子，在12:01:00到12:01:58这段时间内没有用户请求，然后在12:01:59这一瞬时发出100个请求，OK，然后在12:02:00这一瞬时又发出了100个请求。这里你应该能感受到，在这个临界点可能会承受恶意用户的大量请求，甚至超出系统预期的承受。

1. 访问量限流（通过控制单位时间段内调用量来限流）

原理：确定方法的最大访问量MAX，每次进入方法前计数器+1，将结果和最大并发量MAX比较，如果大于等于MAX，则直接返回；如果小于MAX，则继续执行。

优缺点：对于访问量限流这种限流方式，实现简单，适用于大多数场景，阈值可以通过服务端来动态设置，甚至可以当做业务开关来使用。

但是也有一定的局限性，因为我们的阈值是通过分析单位时间内调用量来设定的，如果它在单位时间段的几秒就被流量突刺消耗完了，将导致该时间内剩余的实践内该服务“拒绝服务”，可以将这种现象称为“突刺现象”。

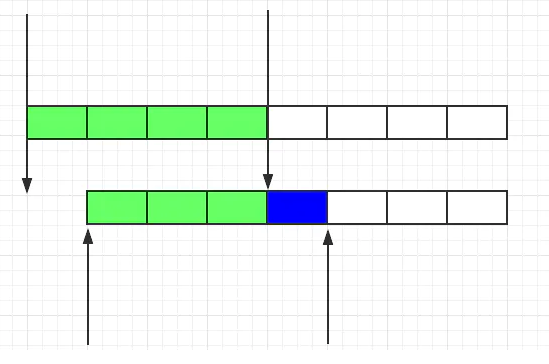
1. 并发量限流（通过限制系统的并发调用程度来限流）

原理：确定方法的最大并发量MAX，每次进入方法前计数器+1，将结果和最大并发量MAX比较，如果大于等于MAX，则直接返回；如果小于MAX，则继续执行；退出方法后计数器-1。  
比如限制服务的并发访问数是100，而服务处理的平均耗时是10毫秒，那么1分钟内，该服务平均能提供( 1000 / 10 ) \* 60 \* 100 = 6000 次

优缺点：  
 并发量限流一般用于对于服务资源有严格限制的场景，但是某个服务在业务高峰期和低峰期的并发量很难评估，这给并发阈值的设置带来了困难，但我们可以通过线上业务的监控数据来逐步对并发阈值进行调优，只要肯花时间，我们总能找到一个即能保证一定服务质量又能保证一定服务吞吐量的合理并发阈值，从表面上看并发量限流似乎很有用，但也不可否认，它仍然可以造成流量尖刺，即每台服务器上该服务的并发量从0上升到阈值是没有任何“阻力”的，这是因为并发量考虑的只是服务能力边界的问题。

### 滑动窗口

由于计数器存在临界点缺陷，后来出现了滑动窗口算法来解决。



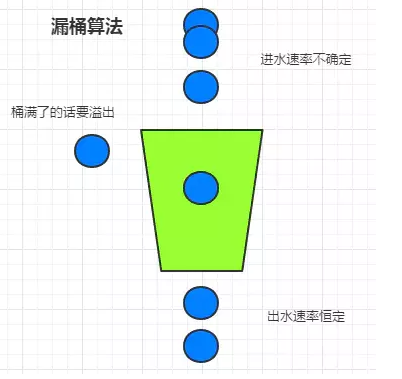
滑动窗口的意思是说把固定时间片，进行划分，并且随着时间的流逝，进行移动，这样就巧妙的避开了计数器的临界点问题。也就是说这些固定数量的可以移动的格子，将会进行计数判断阀值，因此格子的数量影响着滑动窗口算法的精度。

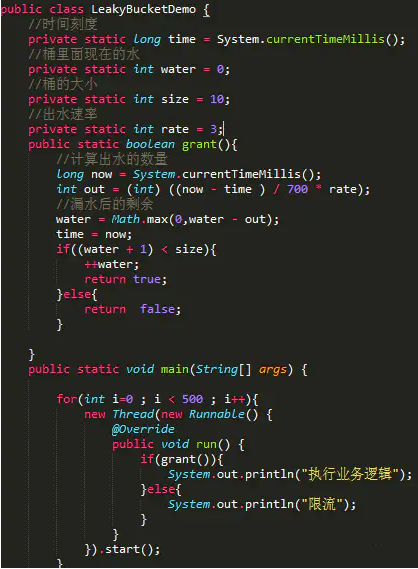
### 漏桶算法

为了消除"突刺现象"，可以采用漏桶算法实现限流，漏桶算法这个名字就很形象，算法内部有一个容器，类似生活用到的漏斗，  
 当请求进来时，相当于水倒入漏斗，然后从下端小口慢慢匀速的流出。不管上面流量多大，下面流出的速度始终保持不变。  
 不管服务调用方多么不稳定，通过漏桶算法进行限流，每10毫秒处理一次请求。因为处理的速度是固定的，请求进来的速度是未知的，可能突然进来很多请求，没来得及处理的请求就先放在桶里，既然是个桶，肯定是有容量上限，如果桶满了，那么新进来的请求就丢弃。

虽然滑动窗口有效避免了时间临界点的问题，但是依然有时间片的概念，而漏桶算法在这方面比滑动窗口而言，更加先进。

有一个固定的桶，进水的速率是不确定的，但是出水的速率是恒定的，当水满的时候是会溢出的。

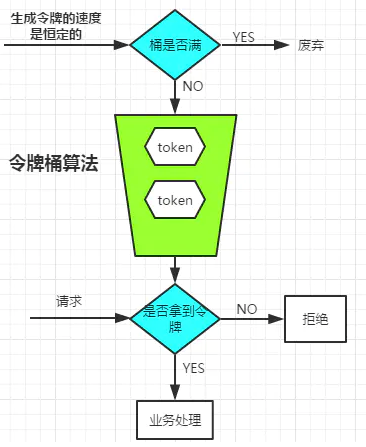




### 令牌桶算法

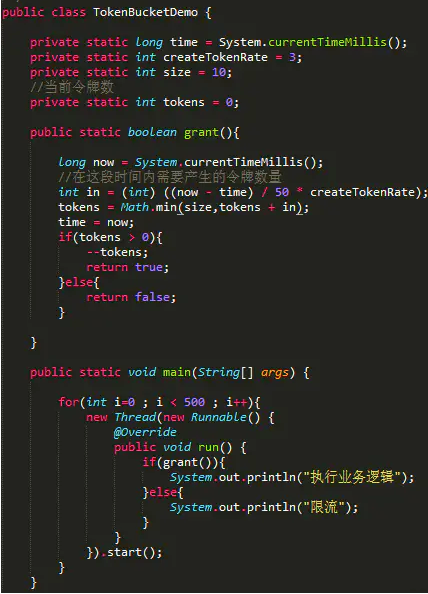
从某种意义上讲，令牌桶算法是对漏桶算法的一种改进，桶算法能够限制请求调用的速率，而令牌桶算法能够在限制调用的平均速率的同时还允许一定程度的突发调用。  
 在令牌桶算法中，存在一个桶，用来存放固定数量的令牌。算法中存在一种机制，以一定的速率往桶中放令牌。  
 每次请求调用需要先获取令牌，只有拿到令牌，才有机会继续执行，否则选择选择等待可用的令牌、或者直接拒绝。  
 放令牌这个动作是持续不断的进行，如果桶中令牌数达到上限，就丢弃令牌，所以就存在这种情况，桶中一直有大量的可用令牌，这时进来的请求就可以直接拿到令牌执行，比如设置qps为100/s，那么限流器初始化完成一秒后，桶中就已经有100个令牌了，这时服务还没完全启动好，等启动完成对外提供服务时，该限流器可以抵挡瞬时的100个请求。所以，只有桶中没有令牌时，请求才会进行等待，最后相当于以一定的速率执行。

注意到，漏桶的出水速度是恒定的，那么意味着如果瞬时大流量的话，将有大部分请求被丢弃掉（也就是所谓的溢出）。为了解决这个问题，令牌桶进行了算法改进。



生成令牌的速度是恒定的，而请求去拿令牌是没有速度限制的。这意味，面对瞬时大流量，该算法可以在短时间内请求拿到大量令牌，而且拿令牌的过程并不是消耗很大的事情。（有一点生产令牌，消费令牌的意味）

不论是对于令牌桶拿不到令牌被拒绝，还是漏桶的水满了溢出，都是为了保证大部分流量的正常使用，而牺牲掉了少部分流量，这是合理的，如果因为极少部分流量需要保证的话，那么就可能导致系统达到极限而挂掉，得不偿失。



## Guava RateLimiter

限流神器：Guava不仅仅在集合、缓存、异步回调等方面功能强大，而且还给我们封装好了限流的API！

Guava RateLimiter基于令牌桶算法，我们只需要告诉RateLimiter系统限制的QPS是多少，那么RateLimiter将以这个速度往桶里面放入令牌，然后请求的时候，通过tryAcquire()方法向RateLimiter获取许可（令牌）。

## 选择

上面所说的限流的一些方式，都是针对单机而言的，其实大部分的场景，单机的限流已经足够了。分布式下限流的手段常常需要多种技术相结合，比如Nginx+Lua，Redis+Lua等去做。

# 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

## HAProxy

## Twemproxy

## ats

## perlbal

## pound

# 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。

# 大型网站性能优化

参考：《大型网站性能优化实战》

## 概述

### 基于用户体验的性能优化要素

### 网站性能分析

### 大型网站性能监控体系

### 大型网站容量评估

## 优化

### 前端性能优化

### 服务端性能优化

### TCP优化

### DNS优化

### CDN优化

### 高性能系统架构模式

### 大促保障体系

### 数据分析驱动性能优化