高并发解决方案讲义(架构师)

北京动力节点教育科技有限公司

动力节点课程讲义

DONGLIJIEDIANKECHENGJIANGYI

www.bjpowernode.com

第 1 章　概述

高并发是互联网应用的一大特点，也是互联网应用不可避免的一个问题；比如 淘宝双 11 购物狂欢节，京东 618 购物促销节，12306 春节火车票，促销，秒杀等

解决高并发问题是一个系统工程，需要站在全局高度统筹谋划，从多个角度进行架构设计

解决高并发问题，不是一个或两个方案就能解决的，需要从各个维度综合施策才能完成

在实践中，我们总结和提炼出来了很多应对高并发的方案或者说手段，分别如下：

第 2 章　高并发解决方案

2.1　硬件

2.1.1 概述

系统访问用户增多，流量增大，导致服务器压力增大，出现性能瓶颈，我们可以采用一个简单粗暴的策略：提升服务器硬件配置，提升服务器硬件配置的策略，也称为：单体应用垂直扩容

比如：产品或者网站初期，通常功能较少，用户量也不多，一般就采用一个项目工程来完成，这就是单体应用，也叫集中式应用。

按照经典的 MVC 三层架构设计，部署单台服务器，使用单台数据库，应用系统和数据库部署在同一台服务器上，随着应用系统功能的增加，访问用户的增多，单台服务器已无法承受那么多的访问流量。

此时，我们可以直接采用简单粗暴的办法：提升硬件配置来解决

2.1.2 单体应用垂直扩容方案

- CPU 从 32 位提升为 64 位
- 内存从 64GB 提升为 256GB（比如缓存服务器）；
- 磁盘从 HDD(Hard Disk Drive)提升为 SSD(固态硬盘(Solid State Drives))，有大量读写的应用
- 磁盘扩容，1TB 扩展到 2TB，比如文件系统
- 千兆网卡提升为万兆网卡

但是不管怎么提升硬件性能，硬件性能的提升不可能永无止尽，所以最终还是要靠分布

式解决

2.2 缓存

2.2.1 概述

缓存可以说是解决大流量高并发，优化系统性能非常重要的一个策略
它是解决性能问题的利器，就像一把瑞士军刀，锋利强大
缓存在高并发系统中无处不在（到处都是）

2.2.2 http 缓存

（1） 浏览器缓存

浏览器缓存是指当我们使用浏览器访问一些网站页面或者 HTTP 服务时，根据服务器端返回的缓存设置响应头将响应内容缓存到浏览器，下次可以直接使用缓存内容或者仅需要去服务器端验证内容是否过期即可，这样可以减少浏览器和服务器之间来回传输的数据量，节省带宽，提升性能；

比如新浪：http://www.sina.com.cn/

第一次访问返回 200，第二次刷新访问，返回响应码为 304，表示页面内容没有修改过，浏览器缓存的内容还是最新的，不需要从服务器获取，直接读取浏览器缓存即可

我们也可以在 Java 代码中通过设置响应头，告诉前端浏览器进行缓存：

```
DateFormat format = new SimpleDateFormat("EEE,MMM yyyy HH: mm: ss 'GMT'", Locale. US);

//当前时间

long now = System.currentTimeMillis() * 1000 * 1000;

response.addHeader( "Date", format.format(new Date()));

//过期时间 http 1. 0 支持

response.addHeader("Expires", format.format (new Date(now+ 20 * 1000)));

//文档生存时间 http 1.1 支持

response.addHeader("Cache-Control", "max-age=20");
```

（2） Nginx 缓存

Nginx 提供了 expires 指令来实现缓存控制，比如：

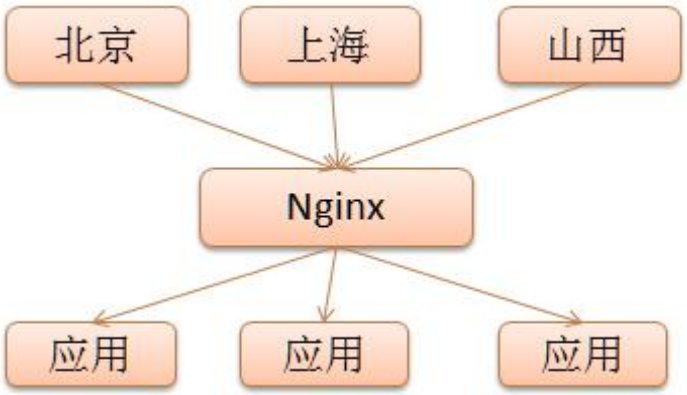
```
location /static {
    root /opt/static/;
    expires 1d;//全天
}
```

当用户访问时，Nginx 拦截到请求后先从 Nginx 本地缓存查询数据，如果有并且没有过期，则直接返回缓存内容

（3） CDN 缓存

CDN 的全称是 Content Delivery Network，即内容分发网络。CDN 是构建在网络之上的内容分发网络，依靠部署在各地的边缘服务器，通过中心平台的负载均衡、内容分发、调度等功能模块，使用户就近获取所需内容，降低网络拥塞，提高用户访问响应速度和命中率。CDN 的关键技术主要有内容存储和分发技术。

CDN 它本身也是一个缓存，它把后端应用的数据缓存起来，用户要访问的时候，直接从 CDN 上获取，不需要走后端的 Nginx，以及具体应用服务器 Tomcat，它的作用主要是加速数据的传输，也提高稳定性，如果从 CDN 上没有获取到数据，再走后端的 Nginx 缓存，Nginx 上也没有，则走后端的应用服务器，CDN 主要缓存静态资源



著名的厂商：（帝联科技）http://www.dnion.com/

2.2.3 应用缓存

（1） 内存缓存

在内存中缓存数据，效率高，速度快，应用重启缓存丢失

（2） 磁盘缓存

在磁盘缓存数据，读取效率较之内存缓存稍低，应用重启缓存不会丢失
代码组件：Guava、Ehcache
服务器：Redis、MemCache

2.2.4 多级缓存

在整个应用系统的不同层级进行数据的缓存，多层次缓存，来提升访问效率；
比如：浏览器 -> CDN -> Nginx -> Redis -> DB (磁盘、文件系统)

2.2.5 缓存的使用场景

- 经常需要读取的数据
- 频繁访问的数据
- 热点数据缓存
- IO 瓶颈数据
- 计算昂贵的数据
- 无需实时更新的数据
- 缓存的目的是减少对后端服务的访问，降低后端服务的压力

2.3 集群

有一个单体应用，当访问流量很大无法支撑，那么可以集群部署，也叫单体应用水平扩容，原来通过部署一台服务器提供服务，现在就多部署几台，那么服务的能力就会提升。

部署了多台服务器，但是用户访问入口只能是一个，比如 www.web.com，所以就需要负载均衡，负载均衡是应用集群扩容后的必须步骤，集群部署后，用户的会话 session 状态要保持的话，就需要实现 session 共享。

2.4 拆分

2.4.1 应用拆分

应用的拆分：分布式 （微服务）

单体应用，随着业务的发展，应用功能的增加，单体应用就逐步变得非常庞大，很多人维护这么一个系统，开发、测试、上线都会造成很大问题，比如代码冲突，代码重复，逻辑

错综混乱，代码逻辑复杂度增加，响应新需求的速度降低，隐藏的风险增大，所以需要按照业务维度进行应用拆分，采用分布式开发；

应用拆分之后，就将原来在同一进程里的调用变成了远程方法调用，此时就需要使用到一些远程调用技术：httpClient、hessian、dubbo、webservice 等；

随着业务复杂度增加，我们需要采用一些开源方案进行开发，提升开发和维护效率，比如 Dubbo、SpringCloud；

通过应用拆分之后，扩容就变得容易，如果此时系统处理能力跟不上，只需要增加服务器即可（把拆分后的每一个服务再多做几个集群）

2.4.2 数据库拆分

数据库拆分分为：垂直拆分和水平拆分 （分库分表）

按照业务维度把相同类型的表放在一个数据库，另一些表放在另一个数据库，这种方式的拆分叫垂直拆分，也就是在不同库建不同表，把表分散到各个数据库

比如产品、订单、用户三类数据以前在一个数据库中，现在可以用三个数据库，分别为产品数据库、订单数据库、用户数据库

这样可以将不同的数据库部署在不同的服务器上，提升单机容量和性能问题，也解决多个表之间的 IO 竞争问题

根据数据行的特点和规则，将表中的某些行切分到一个数据库，而另外的某些行又切分到另一个数据库，这种方式的拆分叫水平拆分

单库单表在数据量和流量增大的过程中，大表往往会成为性能瓶颈，所以数据库要进行水平拆分

数据库拆分，采用一些开源方案，降低开发难度，比如：MyCat、Sharding-Sphere

2.5 静态化

对于一些访问量大，更新频率较低的数据，可直接定时生成静态 html 页面，供前端访问，而不是访问 jsp

常用静态化的技术：freemaker、velocity

定时任务，每隔 2 分钟生成一次首页的静态化页面

页面静态化首先可以大大提升访问速度，不需要去访问数据库或者缓存来获取数据，浏览器直接加载 html 页即可

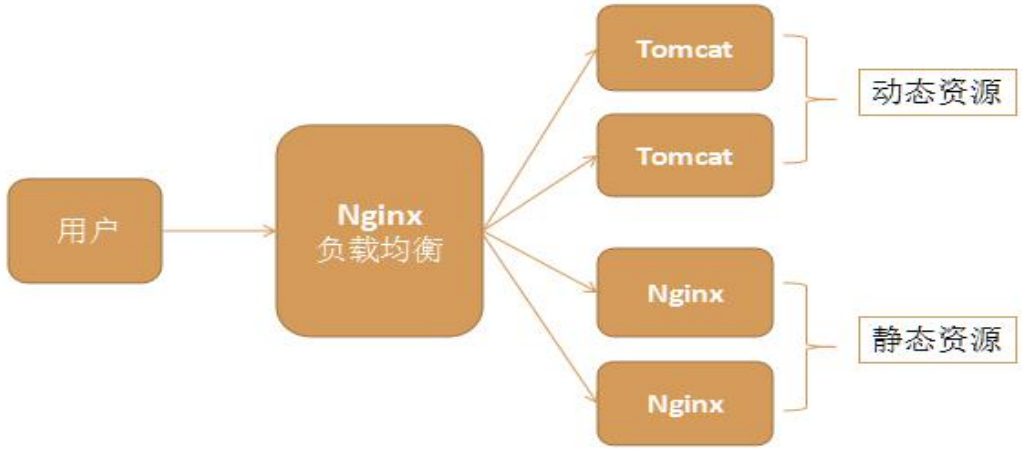
页面静态化可以提升网站稳定性，如果程序或数据库出了问题，静态页面依然可以正常访问

2.6 动静分离

采用比如 Nginx 实现动静分离，Nginx 负责代理静态资源，Tomcat 负责处理动态资源

Nginx 的效率极高，利用它处理静态资源，可以为后端服务器分担压力

动静分离架构示意图



redis 和 nginx 并发量 5w 左右,tomcat 和 mysql700 左右,当然可以通过一些方式调整

2.7 队列

- 采用队列是解决高并发大流量的利器
- 队列的作用就是：异步处理/流量削峰/系统解耦
- 异步处理是使用队列的一个主要原因，比如注册成功了，发优惠券/送积分/送红包/发短信/发邮件等操作都可以异步处理
- 使用队列流量削峰，比如并发下单、秒杀等，可以考虑使用队列将请求暂时入队，通过队列的方式将流量削平，变成平缓请求进行处理，避免应用系统因瞬间的巨大压力而压垮
- 使用队列实现系统解耦，比如支付成功了，发消息通知物流系统，发票系统，库存系统等，而无需直接调用这些系统；
- 队列应用场景

  不是所有的处理都必须要实时处理
  不是所有的请求都必须要实时告诉用户结果
  不是所有的请求都必须 100%一次性处理成功
  不知道哪个系统需要我的协助来实现它的业务处理，保证最终一致性，不需要强一致性

常见的消息队列产品：ActiveMQ/RabbitMQ/RocketMQ/kafka

- ActiveMQ 是 jms 规范下的一个老牌的成熟的消息中间件/消息服务器
- RabbitMQ/RocketMQ 数据可靠性极好，性能也非常优秀，在一些金融领域、电商领域使用很广泛；RocketMQ 是阿里巴巴的；
- kafka 主要运用在大数据领域，用于对数据的分析，日志的分析等处理，它有可能

产生消息的丢失问题，它追求性能，性能极好，不追求数据的可靠性

2.8 池化

在实际开发中，我们经常会采用一些池化技术，减少资源消耗，提升系统性能

2.8.1 对象池

通过复用对象，减少对象创建和垃圾收集器回收对象的资源开销
可以采用 commons-pool2 实现
实际项目采用对象池并不常见，主要在开发框架或组件的时候会采用

2.8.2 数据库连接池

Druid/DBCP/C3P0/BoneCP

2.8.3 Redis 连接池

JedisPool（内部基于 commons-pool2 实现）

2.8.4 HttpClient 连接池

核心实现类：PoolingClientConnectionManager
http://hc.apache.org/httpcomponents-client-ga/httpclient/examples/org/apache/http/examples/client/ClientMultiThreadedExecution.java

2.8.5 线程池

Java 提供 java.util.concurrent 包可以实现线程池
Executors.newFixedThreadPool(8);线程数量固定
Executors.newSingleThreadExecutor();只有一个线程，避免关闭情况
Executors.newCachedThreadPool();可以自动扩容
Executors.newScheduledThreadPool(10);每隔多久执行

2.9 优化

2.9.1 JVM 优化

设置 JVM 参数

```
-server -Xmx4g -Xms4g -Xmn256m
-XX:PermSize=128m
-Xss256k
-XX:+DisableExplicitGC
-XX:+UseConcMarkSweepGC
-XX:+CMSParallelRemarkEnabled
-XX:+UseCMSCompactAtFullCollection
-XX:LargePageSizeInBytes=128m
-XX:+UseFastAccessorMethods
-XX:+UseCMSInitiatingOccupancyOnly
-XX:CMSInitiatingOccupancyFraction=70
```

-server VM有两种运行模式Server与Client，两种模式的区别在于，Client模式启动速度较快，Server模式启动较慢；但是启动进入稳定期长期运行之后Server模式的程序运行速度比Client要快很多；
-Xmx2g 最大堆大小
-Xms2g 初始堆大小
-Xmn256m 堆中年轻代大小；
-XX:PermSize设置非堆内存初始值,默认是物理内存的1/64;由XX:MaxPermSize设置最大非堆内存的大小,默认是物理内存的1/4.
-Xss 每个线程的Stack大小
-XX:+DisableExplicitGC，这个参数作用是禁止代码中显示调用GC。代码如何显示调用GC呢，通过System.gc()函数调用。如果加上了这个JVM启动参数，那么代码中调用System.gc()没有任何效果，相当于是没有这行代码一样。
-XX:+UseConcMarkSweepGC 并发标记清除（CMS）收集器，CMS收集器也被称为短暂停顿并发收集器；
-XX:+CMSParallelRemarkEnabled 降低标记停顿；
-XX:+UseCMSCompactAtFullCollection:使用并发收集器时,开启对年老代的压缩.
-XX:LargePageSizeInBytes 指定 Java heap 的分页页面大小
-XX:+UseFastAccessorMethods 原始类型的快速优化
-XX:+UseCMSInitiatingOccupancyOnly 使用手动定义的初始化定义开始CMS收集
-XX:CMSInitiatingOccupancyFraction 使用cms作为垃圾回收使用70%后开始CMS收集；

参数指南：http://www.oracle.com/technetwork/java/javase/tech/vmoptions-jsp-140102.html

2.9.2 Tomcat 优化

- 设置 JVM 参数，可以参考 JVM 优化参数

  在tomcat的bin目录下的catalina.sh中设置jvm参数：
  JAVA_OPTS="-server -XX:+PrintGCDetails -Xmx4g -Xms4g -Xmn256m
  -XX:PermSize=128m
  -Xss256k
  -XX:+DisableExplicitGC
  -XX:+UseConcMarkSweepGC
  -XX:+CMSParallelRemarkEnabled
  -XX:+UseCMSCompactAtFullCollection
  -XX:LargePageSizeInBytes=128m
  -XX:+UseFastAccessorMethods
  -XX:+UseCMSInitiatingOccupancyOnly
  -XX:CMSInitiatingOccupancyFraction=70"
- 设置 tomcat 的线程池大小
- 设置 IO 模式
- 配置 APR

  可以参考文章：https://www.cnblogs.com/zhuawang/p/5213192.html

2.10 Java 程序优化

- 养成良好的编程习惯
- 不要重复创建太多对象
- 流/文件/连接 一定要记得在 finally 块中关闭
- 少用重量级同步锁 synchronized，采用 Lock
- 不要在循环体中使用 try/catch
- 多定义局部变量，少定义成员变量
- .....

2.11 数据库优化

2.11.1 数据库服务器优化

修改数据库服务器的配置文件的参数，偏 DBA （数据库管理员）

2.11.2 数据库架构优化

- 将数据库服务器和应用服务器分离
- 读写分离：通过数据库主从架构解决，写数据时操作主库，读数据时操作从库，分摊读写压力
- 分库分表：扩容数据库，解决数据量容量问题

2.11.3 数据库索引优化

- 建立合适的索引
- 建立索引的字段尽量的小，最好是数值
- 尽量在唯一性高的字段上创建索引，主键、序号等
- 不要在性别这种唯一性很低的字段上创建索引

2.11.4 SQL 优化

SQL 优化很多，可以总结出很多经验；
参考文章：https://blog.csdn.net/jie_liang/article/details/77340905

2.11.5 采用数据搜索引擎

solr / elasticsearch

2.12 Nginx 优化

2.12.1 调整配置文件参数

```
worker_processes 16;
gzip  on;  #开启 gzip 压缩输出
events {
    worker_connections 65535; #极限值 65535
    multi_accept on; #开启多路连接
    use epoll; #使用 epoll 模型
}
```

2.13 Linux 优化

优化 Linux 内核参数
修改/etc/sysctl.conf
http://blog.51cto.com/yangrong/1567427 偏运维的职责

2.14 网络优化

机房、带宽、路由器等方面优化
网络架构更合理
运维的职责

2.15 前端优化

2.15.1 js 优化

- 压缩变小
  压缩工具
- 多个 js 合并成一个 js 文件，直接手动拷贝到一个文件中去，页面只加载这一个文件或者利用程序,比如 controller，/aa/js?path=xxx.js,xxx.js

2.15.2 css 优化

- 压缩变小
- 多个 css 文件合并成一个 css 文件

2.15.3 html 页面优化

- 不要加载太多 js 和 css
- js 和 css 加载放在页面的尾部，从用户体验角度考虑的
- 页面上减少到服务的请求数

2.16 压测

- 压测就是压力测试

- 在系统上线前，需要对系统各个环节进行压力测试，发现系统的瓶颈点，然后对系统的瓶颈点，进行调优。调优完成后，还需要考虑另外一些风险因素，比如网络不稳定，机房故障等。所以我们需要提前有故障预备方案，比如多机房部署容灾、路由切换等。故障预备方案做好后，还需要提前进行演练，以确保预案的有效性
- 压力测试工具

  Apache JMeter / LoadRunner 等，偏测试的工作
- CTO、架构师，技术团队、测试团队、运维团队、DBA 等共同完成

2.17 总结

完成以上的工作，我们才能实现一个 高并发、高性能、高可用 的“三高”分布式系统