**动脑性能优化**

**课后资料**

# 概要

本文的主要读者是软件开发人员，有一定的编程基础，主要使用的环境是SSM框架、Spring boot或其他Java框架。高级程序员当然也可以阅读本文，仅供一些参考。

并没有多少故事，也不会是一本书。其核心内容是将一些性能调优的关键点，个人经验阐述一下，介绍性能调优的思路及处理方式。

为什么要性能调优，明显是个不需要讨论的话题。但仍要提出来，是因为往往程序员自测的系统，和实际生产环境运行的效果有很大差异。经常性能调优的系统与未调优的系统，或许都可以在生产环境正常运行，但明显前者更有发展的潜力。

本文的重点也不在于使用最新的技术、框架进行分布式开发阐述或性能调优，一般仅用最常用最基础的方法进行处理。因为低成本及可持续推进优化才使我们更容易达到目标。

# 性能评估

首先要区分我们的开发环境与生产环境的不同，尤其生产环境的配置对性能调优起到至关重要的作用。在性能调优之前，我们需要对生产环境及系统程序进行一些基础的性能评估，以对接下来的操作有一定的了解及准备，并易于制定调优方案。

性能评估本节专指整体，而不是单个单元，如虚拟机或Tomcat或数据库环节。评估一般分为三个方面：

**吞吐量**：对于事务处理系统，吞吐量通常用每秒事务数 (TPS) 或每分钟事务数 (TPM) 来评估。

**响应时间**：响应时间评估单个事务或查询的性能。通常认为响应时间是从用户输入一个命令或激活一个函数开始，至应用程序指示该命令或函数已完成为止所消耗的时间。

**每个事务的成本**：每个事务的成本是财务上的量度，通常用于比较应用程序、数据库服务器或硬件平台之间的总体操作成本。

在以上三个方面整体考察性能评估结论。但并不总是可以同时优化的，往往需要在事务的完整性与呑吐量及响应时间中取得一个平衡，也就是证系统的正常运行的同时强调性能，否则性能也失去了意义。

需要了解的几个常用的指标：

硬件级：CPU的运行速度，内存大小，磁盘写入性能，磁盘读取性能；

操作系统级：线程数限制，文件限制，连接限制；

应用级：WEB应用的常规并发量，数据库的最高并发量；

后文将逐一对以上指标解释并在有关场景中通过配置优化进行调整。

# 调优策略

有战略胜于无战略，在战略的基础上考虑战术问题（也就是一些具体的指标配置）。

战略1：确保数据完整性；

战略2：以高可用为目标；

战略3：备用方案一定要有；

战略4：必须是可持续推进的思路。

下面是对于以上四条战略的详细解释：

战略1：确保数据完整性。无论性能要求如何迫切，也必须确保数据的完整性。此条看起来正常，操作起来难度大，尤其是特殊事件（停电，宕机），分布式集群环境（脑裂，双主）下的数据完整性，并不很容易达到。在实际的生产环境中，WEB服务有不可用的情况发生时，用户催，老板逼！此时要稳住，首先要考虑数据完整性，如不能保证，就让服务不可用好了。否则在服务启动后，所有数据丢失的责任你要承担，并善后。不如在一开始就堵住，不让它发生。

具体来说：数据库**主备**及定时冷备份一定要有，不强行杀死进程，使用服务的方式重启。阻塞时无法重启时，**断网**比杀死进程安全。切换从库为主时，要保证主库不启动，关闭从库的复制功能，断网并重启主库，并设为从库，再联网。在ZK环境下的分布式数据库暂不讨论。

战略2：以高可用为目标。单机环境下，也需要负载均衡，当然这是极端情况，普通系统不存在高可用目标，故障时直接重启的代价可以接受，不用考虑太多。在多用户高并发的需求中，需要多机分布式环境：至少三机，两台WEB，一台数据库。备库可在一台WEB机上部署。采用高效的WEB服务器如nginx，建立负载均衡、在局域网内部署。

战略3：备用方案一定要有。除基本优化（不可替代性，如服务器参数、虚拟机参数）外，数据库优化及程序优化必须有可用替代方案优化，不能优化走极端，一但思路出问题无法挽回。

战略4：必须是可持续推进的思路。很简单，当前的优化，在目前的承载力可用，随着系统及用户的发展以及营销方式的发展，会出现更复杂的情况。一切思路都要预留空间，比如SQL语句优化后，后续发展中可能要改写了，分表了，分库了，分布式了。避免单一的SQL优化过于极端，花费巨大的代价优化一个语句，达到次优解，并可用时就停止优化，这是其一。通过汇总表，拆表的方式的优化，需要避免在新的环境里重新设计结构才能可用，是其二。更多的就不提了，需在各自的使用过程中进一步体会。

步骤：

1. 基础环境优化优先

地基是一切的基础，硬件及操作系统的限制，使得其他手段必须建立其上。操作级的参数，提高连接数限制，打开文件数等是第一基础工作。

1. 使用先进的软件其次

好的服务软件，可充分利用硬件及操作系统级性能。nginx>apache>tomcat。数据库其实mysql在大数据量时表现不佳，不要仅比较写入与读取性能，统计及复杂查询也是必须的功能。虽然它有办法解决，但是工作量大了几倍（个人见解）。

1. 开启小型分布式环境及负载均衡

通过nginx的反向代理，建立内部负载均衡环境。单机多tomcat，多机多nginx负载，全面利用服务器能力。

1. 调优数据库

瓶颈在数据库上！

1. 调优系统程序

最后才需要反省你的代码好不好，你的代码会被编译器改好很多的，不用担心（太臭的不要提）。

# 服务器优化

## 硬件采购

一台好的服务器，必然给性能带来提升空间。自购与VPS各有优劣。自购的难于管理，安装操作系统，配置环境，硬件级维护，机房托管等事情繁杂。目前推荐VPS这种形式，应该会越来越普及。也许在未来某一阶段，大型的服务器采购活动只存在于VPS服务商之间。

VPS服务器配比：CPU与内存1：2或者1：4均可，也就是2核8G之类的

VPS数据库配比：CPU与内存1：2。因数据库繁忙时，CPU的压力也大，内存当然是越大越好。尤其innodb引擎对于内存的要求很大。

常规采购，VPS4核8G已能满足要求，可达到tomcat的承载力水平。更高当然更好。带宽5M为性价比之选。5M以上贵，不如多机负载。单独购买负载均衡也是对的，按流量计费。

## 服务器安装

建议采用Centos发行版，因其来源于RED HAT的长期支持。稳定性很好，安全性可选择启用SELINUX安全套件，但启用后会导致性能下降5%。其他发行版也很好，但不作为推荐之选。Windows不要考虑了，IIS什么的自己花钱买台个人电脑玩下算了。

升级软件，不升级核心，切记！！！

记得买份快照，省事省力。

要不要建swap交换分区（虚拟内存）。建的话，当Mysql内存不足时，可以使用，不会造成连管理控制台都登录不了恶果。不建的话，性能必然快些，因为没有数据可以进入交换分区了，也就是内存数据不会走磁盘！自己考虑。

往往要建swap分区时，是购买的VPS，没有多余的磁盘或分区，采用文件型swap也可行，操作如下：

1、dd if=/dev/zero of=/var/swap bs=1M count=2048

2、mkswap /var/swap

3、swapon /var/swap

4、在/etc/fstab中添加一行：/var/swap swap swap defaults 0 0

第1句，创建一个swap文件，块大小为1M，总量为2G（按内存的双倍，但你如果是8G内存，这个就不讲究了）。第2、3句启用它，第4句是保持开机启动。

自购服务器自己安装的话。。。。。。我们看下一节。

## 服务器参数调优

服务器默认参数对于连接数，文件打开数等有一定限制。我们需要从以下几个方面进行调整：

需处理的文件：/etc/sysctl.conf，用vi或vim打开它。

1. 连接数优化，优化三个方面，增加连接数，设置超时，使它可重用

net.ipv4.tcp\_syncookies = 1，防范少量SYN攻击

net.ipv4.tcp\_tw\_reuse = 1，开启重用，很重要！

net.ipv4.tcp\_tw\_recycle = 1，sockets的快速回收

net.ipv4.tcp\_fin\_timeout = 30，关闭前的等待时间

net.ipv4.tcp\_keepalive\_time = 1200，原为两小时

net.ipv4.tcp\_keepalive\_intvl = 30，robe 3次(每次30秒)不成功,内核才彻底放弃

net.ipv4.tcp\_keepalive\_probes = 3，原为9，作用见上条

net.ipv4.ip\_local\_port\_range = 1024   65000，扩大端口范围

net.ipv4.tcp\_max\_syn\_backlog = 8192，backlog都是指的后台等待队列

net.ipv4.netdev\_max\_backlog = 1000

et.core.tcp\_max\_tw\_buckets = 5000

net.ipv4.tcp\_mem[0]:低于此值,TCP没有内存压力.，建议786432

net.ipv4.tcp\_mem[1]:在此值下,进入内存压力阶段，建议1048576.

net.ipv4.tcp\_mem[2]:高于此值,TCP拒绝分配socket.，建议1572864

net.core.somaxconn = 32768，socket监听（listen）的backlog上限，原128

修改完了记得sysctl –p，重新加载内核参数

1. 文件打开数优化，Mysql打开表，就是打开了文件。Java运行也是读取class文件加载。。。

修改/etc/security/limits.conf

linux有文件句柄限制，默认不高，一般都是1024。

soft nofile 65535  
　 hard nofile 65535

或者用命令ulimit –HSn 65535

## 服务器软件安装

用最好的软件，才能有如飞的感觉。但是服务器上不宜安装过多软件。

精简如下：nginx，tomcat，mysql。如需监控，尽量安装轻量级的监控软件：vmstat，iostat，innotop等。

不需要提供下载链接，因yum安装方式足够便利。使用yum安装后，一般会在/etc目录生成配置文件，好找。另需服务的安装xxx-server，会自动安装依赖的版本及服务配置，也就是不需要自己配置/etc/init.d等目录及启动脚本了。

yum install nginx，这个不需要nginx-server，没有。

yum install tomcat，它会接着安装好openjdk（没有商业的jqs,jstack的附加软件）

yum install mariadb（版本老了，5.5的相当于mysql5.5）

因此需安装mariadb的最新版。先做如下工作：

在/etc/yum.repo.d目录新建一个MariaDB.repo，注意大小写，

命令为：touch /etc/yum.repo.d/MariaDB.repo

vim打开它，输入以下：

[mariadb]

name = MariaDB

baseurl = http://yum.mariadb.org/10.3.5/centos7-amd64/

gpgkey=https://yum.mariadb.org/RPM-GPG-KEY-MariaDB

gpgcheck=1

版本号自己填好，根据最新版来。vi系的编辑器，按i进入编辑模式，输入或粘贴以上文本，在xshell中可以粘贴。

然后就是yum install MariaDB-server。把服务一起装了。

问题是数据库安装好后，变更数据目录，为以后的磁盘迁移或扩容做准备。

方法是在/etc/my.cnf.d/server.cnf中配置安装目录，数据目录。后续再谈它的配置优化时贴出。以上软件安装好后，设置为服务自启动：

systemctl enable nginx

systemctl enable tomcat

systemctl enable mariadb

启动它们：

service nginx start

service tomcat start

service mariadb start

其中nginx和tomcat都需要再配置。tomcat最简单，只需在/etc/tomcat/tomcat.conf中配置一下JAVA\_OPTS（虚拟机参数放这），及server.conf中配置webapps的目录指向。

而nginx的配置较复杂，关键我们需要配上nginx的反向代理及server配置，看下节。

## 服务器软件配置

### nginx配置

启动nginx的是服务，或者说是root帐号。因此，nginx不能使用管理员权限去工作，工作线程需单独指定帐号。在安装好nginx后，就已经自动创建了nginx帐号，这是一个nginx.conf配置，放在/etc/nginx/目录中。

user nginx;

worker\_processes auto;

error\_log /var/log/nginx/error.log;

pid /run/nginx.pid;

include /usr/share/nginx/modules/\*.conf;

events {

use epoll;

worker\_connections 10240;

}

http {

log\_format main '$remote\_addr - $remote\_user [$time\_local] "$request" '

'$status $body\_bytes\_sent "$http\_referer" '

'"$http\_user\_agent" "$http\_x\_forwarded\_for"';

access\_log /var/log/nginx/access.log main;

server\_names\_hash\_bucket\_size 128;

client\_header\_buffer\_size 16k;

large\_client\_header\_buffers 4 16k;

client\_header\_timeout 10m;

client\_body\_timeout 10m;

client\_max\_body\_size 8m;

send\_timeout 10m;

connection\_pool\_size 256;

request\_pool\_size 4k;

fastcgi\_connect\_timeout 1800;

fastcgi\_send\_timeout 1800;

fastcgi\_read\_timeout 1800;

fastcgi\_buffer\_size 64k;

fastcgi\_buffers 4 64k;

fastcgi\_busy\_buffers\_size 128k;

fastcgi\_temp\_file\_write\_size 128k;

gzip on;

gzip\_static on;

#gzip\_min\_length 1k;

gzip\_buffers 4 16k;

gzip\_http\_version 1.1;

gzip\_comp\_level 6;

gzip\_proxied any;

gzip\_types text/xml text/css text/javascript text/plain application/json \

application/x-javascript application/xml application/xml+rss image/jpeg image/gif image/png;

gzip\_vary on;

output\_buffers 1 32k;

postpone\_output 1460;

sendfile on;

tcp\_nopush on;

tcp\_nodelay on;

keepalive\_timeout 75 20;

include /etc/nginx/mime.types;

default\_type application/octet-stream;

upstream java\_backend {

server 127.0.0.1:8080;

}

include /etc/nginx/conf.d/\*.conf;

server {

listen 80 default\_server;

listen [::]:80 default\_server;

server\_name \_;

root /usr/share/nginx/html;

# Load configuration files for the default server block.

include /etc/nginx/default.d/\*.conf;

location / {

}

error\_page 404 /404.html;

location = /40x.html {

}

error\_page 500 502 503 504 /50x.html;

location = /50x.html {

}

}

}

而这是nginx中一个server的配置，放在/etc/nginx/conf.d/。权限为root可以，因为是服务启动要用的，而不是工作有关。

server {

listen 80;

server\_name www.asitexxx.com;

charset utf-8;

access\_log /var/log/nginx/asite.access.log main;

location /nginxstatus {

stub\_status on;

access\_log on;

}

error\_page 500 502 503 504 /50x.html;

location = /50x.html {

root html;

}

location / {

proxy\_set\_header Host $host;

proxy\_set\_header X-Forwarded-For $http\_x\_forwarded\_for;

proxy\_set\_header X-Real-IP $remote\_addr;

proxy\_set\_header X-Forwarded-Proto http;

proxy\_pass http://java\_backend;

}

}

### tomcat配置

位置在/etc/tomcat/tomcat.conf。注意第一行，这行配置是指的多个tomcat配置时使用哪个目录，而不需要复制tomcat安装。除此外，最重要的就是JAVA\_OPTS，它就是虚拟机参数。

TOMCAT\_CFG\_LOADED="1"

TOMCATS\_BASE="/var/lib/tomcats/"

JAVA\_HOME="/usr/lib/jvm/jre"

CATALINA\_HOME="/usr/share/tomcat"

CATALINA\_TMPDIR="/var/cache/tomcat/temp"

JAVA\_OPTS="-server –Xms2048m -Xmx2048m -XX:MaxNewSize=256m -Djava.awt.headless=true"

SECURITY\_MANAGER="false"

# 虚拟机优化

虚拟机的原理在PPT中已讲得很清楚，这里复述的概念有：虚拟机组成，命令处理流程，及优化措施。

## 虚拟机组成

由三部分组成：编译器，堆与栈。

编译器分为即时编译器与解释器。即时编译器将代码编译成本地代码存于code区。因此它快，但它有内存限制！解释器逐行解释字节码，相当于脚本顺序执行，很慢，性能约为C语言的80%。优化的一部分是使代码尽早进入编译器。将部分代码内联（函数散开于代码中，与编译器无关）。

栈是JVM的函数栈。所有函数必分配于栈。栈中一个帧就是一个函数，因函数之间互相调用，栈帧中包含参数，返回地址，返回值等。第一个参数必然是this指针。递归函数会形成大量的栈帧，搞不好会溢出了。栈的大小可以配置，太大并不好。它是唯一可以配置的地方，其它就不可优化了。

堆是优化的重点，理解为所有对象在堆中。堆划分为年轻代，老年代，持久代（java8是metaspace-使用系统内存，而不是虚拟机堆内存，不再是perm，也就不是持久代了）。年轻代又分为幸存区两个或多个（其中一个必空）及eden（伊甸园区）。

初始的对象在伊甸园区，经过GC收集后，进入存活区，已被收集的对象当然哪也不去，被销毁了。存活区经过15次收集还存活，则进入老年代。存活区满了后交换存活区，并清空原存活区。

老年代最大，因它要放置大对象及长久不消失的对象。

好，看出来堆的优化，就是合理的设置堆内存空间的大小，使之少发生GC，不浪费，不拥挤，不抖动。

JVM的参数分为-X，-XX，-D三种类型。虚拟机自身划分为客户端与server模式。我们当然要用server模式。server模式中编译器默认为混合模式，也就是即时编译器JIT与解释器混合使用。不需要改它。

## 命令处理流程

1. 将java代码编译成字节码class文件
2. 虚拟机解释器执行
3. 或由编译器执行

关键是两个，一个是编译器在预热后，执行超过一定次数的函数将会以编译器模式执行，而另一个是小函数及final函数可能以内联方式执行。

## 优化措施

年轻代大小选择

响应时间优先的应用：尽可能设大，直到接近系统的最低响应时间限制 （根据实际情况选择）。在此种情况下，年轻代收集发生的频率也是最小的。同时，减少到达年老代的对象。

吞吐量优先的应用 ：尽可能的设置大，可能到达Gbit的程度。因为对响应时间没有要求，垃圾收集可以并行进行，一般适合8CPU以上的应用。

年老代大小选择

响应时间优先的应用 ：年老代使用并发收集器，所以其大小需要小心设置，一般要考虑并发会话率 和会话持续时间 等一些参数。如果堆设置小了，可以会造成内存碎片、高回收频率以及应用暂停而使用传统的标记清除方式；如果堆大了，则需要较长的收集时间。最优化的方案，一般需要参考以下数据获得：

并发垃圾收集信息

持久代并发收集次数

传统GC信息

花在年轻代和年老代回收上的时间比例

减少年轻代和年老代花费的时间，一般会提高应用的效率

吞吐量优先的应用 ：一般吞吐量优先的应用都有一个很大的年轻代和一个较小的年老代。原因是，这样可以尽可能回收掉大部分短期对象，减少中期的对象，而年老代尽存放长期存活对象。

较小堆引起的碎片问题

因为年老代的并发收集器使用标记、清除算法，所以不会对堆进行压缩。当收集器回收时，他会把相邻的空间进行合并，这样可以分配给较大的对象。但是，当堆空间较小时，运行一段时间以后，就会出现“碎片”，如果并发收集器找不到足够的空 间，那么并发收集器将会停止，然后使用传统的标记、清除方式进行回收。如果出现“碎片”，可能需要进行如下配置：

-XX:+UseCMSCompactAtFullCollection ：使用并发收集器时，开启对年老代的压缩。

-XX:CMSFullGCsBeforeCompaction=0 ：上面配置开启的情况下，这里设置多少次Full GC后，对年老代进行压缩

## 虚拟机参数表

-XX:+<option> 启用option，例如：-XX:+PrintGCDetails启动打印GC信息的选项，其中+号表示true，开启的意思

-XX:-<option> 不启用option，例如：-XX:-PrintGCDetails关闭启动打印GC信息的选项，其中-号表示false，关闭的意思

-XX:<option>=<number> 设定option的值为数字类型，可跟单位，例如 32k, 1024m, 2g。例如：-XX:MaxPermSize=64m

-XX:<option>=<string> 设定option的值为字符串，例如： -XX:HeapDumpPath="C:\Users\Daxin\Desktop\jvmgcin"

但是有的参数不需要使用-XX，例如：-Xms， -Xmx ，-Xmn ，-Xss

|  |  |  |
| --- | --- | --- |
| **性能选项** | | |
| 选项与默认值 | 默认值与限制 | 描述 |
| -Xms | 初始堆大小 | 默认(MinHeapFreeRatio参数可以调整)空余堆内存小于40%时，JVM就会增大堆直到-Xmx的最大限制. |
| -Xmx | 最大堆大小 | 默认(MaxHeapFreeRatio参数可以调整)空余堆内存大于70%时，JVM会减少堆直到 -Xms的最小限制. 注意：此处的大小是（eden+ 2 survivor space).与jmap -heap中显示的New gen是不同的。 整个堆大小=年轻代大小 + 年老代大小 + 持久代大小. |
| -Xmn | 年轻代大小(1.4or lator) | 增大年轻代后,将会减小年老代大小.此值对系统性能影响较大,Sun官方推荐配置为整个堆的3/8 |
| -Xss | 每个线程的堆栈大小 | JDK5.0以后每个线程堆栈大小为1M,以前每个线程堆栈大小为256K.更具应用的线程所需内存大小进行 调整.在相同物理内存下,减小这个值能生成更多的线程.但是操作系统对一个进程内的线程数还是有限制的,不能无限生成,经验值在3000~5000左右 一般小的应用， 如果栈不是很深， 应该是128k够用的 大的应用建议使用256k。这个选项对性能影响比较大，需要严格的测试。（校长） 和threadstacksize选项解释很类似,官方文档似乎没有解释,在论坛中有这样一句话:"” -Xss is translated in a VM flag named ThreadStackSize” 一般设置这个值就可以了。 |
| -XX:MaxTenuringThreshold | 垃圾最大年龄 | 如果设置为0的话,则年轻代对象不经过Survivor区,直接进入年老代. 对于年老代比较多的应用,可以提高效率.如果将此值设置为一个较大值,则年轻代对象会在Survivor区进行多次复制,这样可以增加对象再年轻代的存活 时间,增加在年轻代即被回收的概率 该参数只有在串行GC时才有效. |
| -XX:ParallelGCThreads | 并行收集器的线程数 | 此值最好配置与处理器数目相等 同样适用于CMS |
| -XX:+AggressiveOpts | JDK 5 update 6后引入，但需要手动启用。 | 启用JVM开发团队最新的调优成果。例如编译优化，偏向锁，并行年老代收集等。 |
| JDK6默认启用。 |
| -XX:LargePageSizeInBytes=4m | 默认4m，amd64位：2m | 设置堆的内存页大小。 |
| -XX:MaxHeapFreeRatio=70 | 70 | GC后，如果发现空闲堆内存占到整个预估堆内存的70%，则收缩堆内存预估最大值。 |
| 什么是预估堆内存？ |
| 预估堆内存是堆大小动态调控的重要选项之一。堆内存预估最大值一定小于或等于固定最大值(-Xmx指定的数值)。前者会根据使用情况动态调大或缩小，以提高GC回收的效率。 |
| -XX:NewSize | 设置年轻代大小(for 1.3/1.4) |  |
| -XX:MaxNewSize=size | 1.3.1 Sparc: 32m | 新生代占整个堆内存的最大值。 |
| 1.3.1 x86: 2.5m |
| -XX:MaxPermSize=64m | 5.0以后: 64 bit VMs会增大预设值的30% | Perm占整个堆内存的最大值。 |
| 1.4 amd64: 96m |
| 1.3.1 -client: 32m |
| 其他默认 64m |
| -XX:MinHeapFreeRatio=40 | 40 | GC后，如果发现空闲堆内存占到整个预估堆内存的40%，则放大堆内存的预估最大值，但不超过固定最大值。 |
| 关联选项： |
| -XX:MaxHeapFreeRatio=70 |
| -XX:NewRatio=2 | Sparc -client: 8 | 新生代和年老代的堆内存占用比例。 这里的2表示，新生代占最大堆内存的1/2。也就是和年老代平分堆的占用。 |
| x86 -server: 8 |
| x86 -client: 12 |
| -client: 4 (1.3) |
| 8 (1.3.1+) |
| x86: 12 |
| 其他默认 2 |
| -XX:NewSize=2.125m | 5.0以后: 64 bit Vms会增大预设值的30% | 新生代预估堆内存占用的默认值。(什么是预估堆内存？见 -XX:MaxHeapFreeRatio 处的描述) |
| x86: 1m |
| x86, 5.0以后: 640k |
| 其他默认 2.125m |
| -XX:ReservedCodeCacheSize=32m | Solaris 64-bit, amd64, -server x86: 48m | 设置代码缓存的最大值，编译用。 |
| 1.5.0\_06之前, Solaris 64-bit amd64: 1024m |
| 其他默认 32m |
| -XX:SurvivorRatio=8 | Solaris amd64: 6 | Eden与Survivor的占用比例。这里的8表示，一个survivor区占用 1/8 的新生代内存，因为survivor有2个，所以是 2/8，那么Eden的占比为 6/8。 |
| Sparc in 1.3.1: 25 |
| Solaris platforms5.0以前: 32 |
| 其他默认 8 |
| -XX:TargetSurvivorRatio=50 | 50 | 实际使用的survivor空间大小占比。默认是50%，最高90%。 |
| -XX:ThreadStackSize=512 | Sparc: 512 | 线程堆栈大小 |
| Solaris x86: 320(5.0以前 256) |
| Sparc 64 bit: 1024 |
| Linux amd64: 1024 (5.0 以前 0) |
| 其他默认 512. |
| -XX:+UseBiasedLocking | JDK 5 update 6后引入，但需要手动启用。 | 启用偏向锁。 |
| JDK6默认启用。 |  |
| -XX:+UseFastAccessorMethods | 默认启用 | 启用原始类型的getter方法优化。 |
| -XX:-UseISM | 默认启用 | 启用solaris的ISM。 |
| 详见Intimate Shared Memory. |
| -XX:+UseLargePages | JDK 5 update 5后引入，但需要手动启用。 | 启用大内存分页。 |
| JDK6默认启用。 |  |
| -XX:+UseMPSS | 1.4.1 之前: 不启用 | 启用solaris的MPSS，不能与ISM同时使用。 |
| 其余版本默认启用 |
| -XX:+StringCache | 默认启用 | 启用字符串缓存。 |
| -XX:AllocatePrefetchLines=1 | 1 | 与机器码指令预读相关的一个选项，资料比较少，本文档不做解释。有兴趣的朋友请自行阅读官方doc。 |
| -XX:AllocatePrefetchStyle=1 | 1 | 与机器码指令预读相关的一个选项，资料比较少，本文档不做解释。有兴趣的朋友请自行阅读官方doc。 |
| -XX:-AllowUserSignalHandlers | 限于Linux和Solaris，默认不启用 | 允许为java进程安装信号处理器。 |
| -XX:-DisableExplicitGC | 默认不启用 | 禁止在运行期显式地调用 System.gc()。 开启该选项后，GC的触发时机将由Garbage Collector全权掌控。 需要注意的是，你程序里没调用System.gc()，你依赖的框架，工具可能在使用。例如RMI。请仔细权衡禁用带来的影响。 |
| -XX:-RelaxAccessControlCheck | 默认不启用 | 在Class校验器里，放松对访问控制的检查。作用与reflection里的setAccessible类似。 |
| -XX:-UseConcMarkSweepGC | 默认不启用 | 启用CMS低停顿垃圾收集器。 |
| -XX:-UseParallelGC | 默认不启用，-server时启用 | 策略为新生代使用并行清除，年老代使用单线程Mark-Sweep-Compact清除的垃圾收集器。 |
| -XX:-UseParallelOldGC | 默认不启用 | 策略为老年代和新生代都使用并行清除的垃圾收集器。 |
| -XX:-UseSerialGC | 默认不启用，-client时启用 | 使用串行垃圾收集器。 |
| -XX:+UseSplitVerifier | java5默认不启用，java6默认启用 | 使用新的Class类型校验器 。 |
| 什么是新Class类型校验器？ |
| 新Class类型校验器，将老的校验步骤拆分成两步： |
| 1，类型推断。2，类型校验。 |
| 新类型校验器通过在javac编译时嵌入类型信息到bytecode中，省略了类型推断这一步，从而提升了classloader的性能。 |
| Classload顺序：load -> verify -> prepare -> resove -> init |
| 关联选项： |
| -XX:+FailOverToOldVerifier |
| -XX:+FailOverToOldVerifier | Java6新引入选项，默认启用 | 如果新的Class校验器检查失败，则使用老的校验器。 |
| 关联选项： |
| -XX:+UseSplitVerifier |
| -XX:+HandlePromotionFailure | java5以前是默认不启用，java6默认启用 | 关闭新生代收集担保。 |
| 什么是新生代收集担保？ |
| 在一次理想化的minor gc中，活跃对象会从Eden和First Survivor中被复制到Second Survivor。然而，Second Survivor不一定能容纳所有的活跃对象。为了确保minor gc能够顺利完成，需要在年老代中保留一块足以容纳所有活跃对象的内存空间。这个预留的操作，被称之为新生代收集担保（New Generation Guarantee）。当预留操作无法完成时，就会触发major gc(full gc)。 |
| 为什么要关闭新生代收集担保？ |
| 因为在年老代中预留的空间大小，是无法精确计算的。为了确保极端情况的发生，GC参考了最坏情况下的新生代内存占用，即Eden+First Survivor。这种策略无疑是在浪费年老代内存，并从时序角度看，可能提前触发Full GC。为了避免如上情况的发生，JVM允许开发者关闭新生代收集担保。 |
| 在开启本选项后，minotr gc将不再提供新生代收集担保，而是在出现survior或年老代不够用时，抛出promotion failed异常。 |
| -XX:+UseSpinning | java1.4.2和1.5需要手动启用, java6默认已启用 | 启用多线程自旋锁优化。 |
| 自旋锁优化原理 |
| 大家知道，Java的多线程安全是基于Lock机制实现的，而Lock的性能往往不如人意。原因是，monitorenter与monitorexit这两个控制多线程同步的bytecode原语，是JVM依赖操作系统互斥(mutex)来实现的。互斥是一种会导致线程挂起，并在较短的时间内又需要重新调度回原线程的，较为消耗资源的操作。为了避免进入OS互斥，Java6的开发者们提出了自旋锁优化方法。自旋锁优化的原理是在线程进入OS互斥前，通过CAS自旋一定的次数来检测锁的释放。如果在自旋次数未达到预订值前，发现锁已被释放，则会立即持有该锁。 |
| CAS检测锁的原理详见: 关联选项： |
| -XX:PreBlockSpin=10 |
| -XX:PreBlockSpin=10 | -XX:+UseSpinning必须先启用，对于java6来说已经默认启用了，这里默认自旋10次 | 控制多线程自旋锁优化的自旋次数。(什么是自旋锁优化？见 -XX:+UseSpinning 处的描述) |
| 关联选项： |
| -XX:+UseSpinning |
| -XX:+ScavengeBeforeFullGC | 默认启用 | 在Full GC前触发一次Minor GC。 |
| -XX:+UseGCOverheadLimit | 默认启用 | 限制GC的运行时间。如果GC耗时过长，就抛OOM。 |
| -XX:+UseTLAB | 1.4.2以前和使用-client选项时，默认不启用，其余版本默认启用 | 启用线程本地缓存区（Thread Local）。 |
| -XX:+UseThreadPriorities | 默认启用 | 使用本地线程的优先级。 |
| -XX:+UseAltSigs | 限于Solaris，默认启用 | 为了防止与其他发送信号的应用程序冲突，允许使用候补信号替代 SIGUSR1和SIGUSR2。 |
| -XX:+UseBoundThreads | 限于Solaris, 默认启用 | 绑定所有的用户线程到内核线程。 |
| 减少线程进入饥饿状态（得不到任何cpu time）的次数。 |
| -XX:+UseLWPSynchronization | 限于solaris，默认启用 | 使用轻量级进程（内核线程）替换线程同步。 |
| -XX:+MaxFDLimit | 限于Solaris，默认启用 | 设置java进程可用文件描述符为操作系统允许的最大值。 |
| -XX:+UseVMInterruptibleIO | 限于solaris，默认启用 | 在solaris中，允许运行时中断线程 。 |
| -XX:CMSInitiatingOccupancyFraction=70 | 92 | 使用cms作为垃圾回收 使用70％后开始CMS收集 为了保证不出现promotion failed(见下面介绍)错误,该值的设置需要满足以下公式CMSInitiatingOccupancyFraction计算公式 |

|  |  |  |
| --- | --- | --- |
| **调试选项** | | |
| 选项与默认值 | 默认值与限制 | 描述 |
| -XX:-CITime | 1.4引入。 | 打印JIT编译器编译耗时。 |
| 默认启用 |
| -XX:ErrorFile=./hs\_err\_pid<pid>.log | Java 6引入。 | 如果JVM crash后，将错误日志输出到指定目录。 |
| -XX:-ExtendedDTraceProbes | Java6引入，限于solaris | 启用dtrace诊断。 |
| 默认不启用 |
| -XX:HeapDumpPath=./java\_pid<pid>.hprof | 默认是java进程启动位置，即user.dir | 堆内存快照的存储路径。 |
| 什么是堆内存快照？ |
| 当java进程因OOM或crash被强制退出后，生成hprof（Heap PROFling）格式的堆快照文件。用于出问题后调试，诊断。文件名一般为java\_<pid>\_<date>\_<time>\_heapDump.hprof.  解析快照文件，可以使用 jhat, eclipse MAT，gdb等工具。 |
| -XX:-HeapDumpOnOutOfMemoryError | 1.4.2 update12 和 5.0 update 7 引入。 | 在OOM时，输出一个dump.core文件，记录当时的堆内存快照（什么是堆内存快照? 见 -XX:HeapDumpPath 处的描述）。 |
| 默认不启用 |
| -XX:OnError="<cmd args>;<cmd args>" | 1.4.2 update 9引入 | 当java每抛出一个ERROR时，运行指定命令行指令集。指令集是与OS环境相关的，在linux下多数是bash脚本，windows下是某个dos批处理。 |
| -XX:OnOutOfMemoryError="<cmd args>;<cmd args>" | 1.4.2 update 12和java6时引入 | 当第一次OOM时，运行指定命令行指令集。指令集是与OS环境相关的，在linux下多数是bash脚本，windows下是某个dos批处理。 |
| -XX:-PrintClassHistogram | 默认不启用 | 打印class柱状图，图中除了class，还有该class的instance统计信息。 |
| Windows下, 按ctrl-break时。 |
| Linux下是执行kill -3，或发送SIGQUIT信号。 |
| Jmap –histo pid也实现了相同的功能。详见 http://java.sun.com/javase/6/docs/technotes/tools/share/jmap.html |
| -XX:-PrintConcurrentLocks | 默认不启用 | 在线程dump时，顺便打印java.util.concurrent锁状态。 |
| Jstack –l pid 也同样实现了相同的功能。详见 http://java.sun.com/javase/6/docs/technotes/tools/share/jstack.html |
| -XX:-PrintCommandLineFlags | 5.0 引入，默认不启用 | Java启动时，往stdout打印当前启用的非稳态jvm options。 |
| 例如： |
| -XX:+UseConcMarkSweepGC -XX:+HeapDumpOnOutOfMemoryError -XX:+DoEscapeAnalysis |
| -XX:-PrintCompilation | 默认不启用 | 打印方法被JIT编译时的信息到stdout。 |
| 例如： |
| java.lang.String::charAt (33 bytes) |
| -XX:-PrintGC | 默认不启用 | 开启GC日志打印。 |
| 例如： |
| [Full GC 131115K->7482K(1015808K), 0.1633180 secs] |
| 该选项可通过 com.sun.management.HotSpotDiagnosticMXBean API 和 Jconsole 动态启用。详见 http://java.sun.com/developer/technicalArticles/J2SE/monitoring/#Heap\_Dump |
| -XX:-PrintGCDetails | 1.4.0引入，默认不启用 | 打印GC回收的细节。 |
| 例如： |
| [Full GC (System) [Tenured: 0K->2394K(466048K), 0.0624140 secs] 30822K->2394K(518464K), [Perm : 10443K->10443K(16384K)], 0.0625410 secs] [Times: user=0.05 sys=0.01, real=0.06 secs] |
| 该选项可通过 com.sun.management.HotSpotDiagnosticMXBean API 和 Jconsole 动态启用。详见 http://java.sun.com/developer/technicalArticles/J2SE/monitoring/#Heap\_Dump |
| -XX:-PrintGCTimeStamps | 默认不启用 | 打印GC停顿耗时。 |
| 例如： |
| 2.744: [Full GC (System) 2.744: [Tenured: 0K->2441K(466048K), 0.0598400 secs] 31754K->2441K(518464K), [Perm : 10717K->10717K(16384K)], 0.0599570 secs] [Times: user=0.06 sys=0.00, real=0.06 |
| secs] |
| 该选项可通过 com.sun.management.HotSpotDiagnosticMXBean API 和 Jconsole 动态启用。详见 http://java.sun.com/developer/technicalArticles/J2SE/monitoring/#Heap\_Dump |
| -XX:-PrintTenuringDistribution | 默认不启用 | 打印对象的存活期限信息。 |
| 例如： |
| [GC |
| Desired survivor size 4653056 bytes, new threshold 32 (max 32) |
| - age 1: 2330640 bytes, 2330640 total |
| - age 2: 9520 bytes, 2340160 total |
| 204009K->21850K(515200K), 0.1563482 secs] |
| Age1 2表示在第1和2次GC后存活的对象大小。 |
| -XX:-TraceClassLoading | 默认不启用 | 打印class装载信息到stdout。记Loaded状态。 |
| 例如： |
| [Loaded java.lang.Object from /opt/taobao/install/jdk1.6.0\_07/jre/lib/rt.jar] |
| -XX:-TraceClassLoadingPreorder | 1.4.2引入，默认不启用 | 按class的引用/依赖顺序打印类装载信息到stdout。不同于 TraceClassLoading，本选项只记 Loading状态。 |
| 例如： |
| [Loading java.lang.Object from /home/confsrv/jdk1.6.0\_14/jre/lib/rt.jar] |
| -XX:-TraceClassResolution | 1.4.2引入，默认不启用 | 打印所有静态类，常量的代码引用位置。用于debug。 |
| 例如： |
| RESOLVE java.util.HashMap java.util.HashMap$Entry HashMap.java:209 |
| 说明HashMap类的209行引用了静态类 java.util.HashMap$Entry |
| -XX:-TraceClassUnloading | 默认不启用 | 打印class的卸载信息到stdout。记Unloaded状态。 |
| -XX:-TraceLoaderConstraints | Java6 引入，默认不启用 | 打印class的装载策略变化信息到stdout。 |
| 例如： |
| [Adding new constraint for name: java/lang/String, loader[0]: sun/misc/Launcher$ExtClassLoader, loader[1]: <bootloader> ] |
| [Setting class object in existing constraint for name: [Ljava/lang/Object; and loader sun/misc/Launcher$ExtClassLoader ] |
| [Updating constraint for name org/xml/sax/InputSource, loader <bootloader>, by setting class object ] |
| [Extending constraint for name java/lang/Object by adding loader[15]: sun/reflect/DelegatingClassLoader  ] |
| 装载策略变化是实现classloader隔离/名称空间一致性的关键技术。 |
| -XX:+PerfSaveDataToFile | 默认启用 | 当java进程因OOM或crash被强制退出后，生成一个堆快照文件（什么是堆内存快照? 见 -XX:HeapDumpPath 处的描述）。 |

## 虚拟机工具说明

1、命令行工具

Sun JDK监控和故障处理命令有jps，jstat，jmap，jhat，jstack，jinfo

jps，JVM Process Status Tool,显示指定系统内所有的HotSpot虚拟机进程。

jstat，JVM statistics Monitoring是用于监视虚拟机运行时状态信息的命令，它可以显示出虚拟机进程中的类装载、内存、垃圾收集、JIT编译等运行数据。

jmap，JVM Memory Map命令用于生成heap dump文件

jhat，JVM Heap Analysis Tool命令是与jmap搭配使用，用来分析jmap生成的dump，jhat内置了一个微型的HTTP/HTML服务器，生成dump的分析结果后，可以在浏览器中查看

jstack，用于生成java虚拟机当前时刻的线程快照。

jinfo，JVM Configuration info 这个命令作用是实时查看和调整虚拟机运行参数。

图形工具

常用调优工具分为两类,jdk自带监控工具：jconsole和jvisualvm，第三方有：MAT(Memory Analyzer Tool)、GChisto。

jconsole，Java Monitoring and Management Console是从java5开始，在JDK中自带的java监控和管理控制台，用于对JVM中内存，线程和类等的监控

jvisualvm，jdk自带全能工具，可以分析内存快照、线程快照；监控内存变化、GC变化等。

MAT，Memory Analyzer Tool，一个基于Eclipse的内存分析工具，是一个快速、功能丰富的Java heap分析工具，它可以帮助我们查找内存泄漏和减少内存消耗

GChisto，一款专业分析gc日志的工具

# Tomcat优化

## 虚拟机JVM的参数位置

默认在/etc/tomcat/tomcat.conf中有个JAVA\_OPTS参数，填写虚拟机优化参数即可。

## Tomcat自身连接、线程等的优化

server，service，engine，host，context一环套一环。中间vivle当桥梁。关键在于connectors。

而优化思路在于：网络优化，线程优化，并发优化

* **协议优化**

在connector的配置区，一定要加上这一句：

protocol="org.apache.coyote.http11.Http11AprProtocol"

虽然有四种连接方式，apr模式是操作系统级的，我们建议尽量换成apr。nio和nio2是非阻塞的也不错。bio模式就是默认的http1.1，该模式在tomcat8中已取消。除了修改为以上参数，yum install tomcat-native也是必要的，否则不能启动了。

* **启用压缩**

主流的web应用服务器都支持gzip，tomcat也不例外。虽然nginx也可启用，在单独访问tomcat资源或jsp时也需要启用。

compression="on"

disableUploadTimeout="true"

compressionMinSize="2048"

compressableMimeType="text/html,text/xml,text/plain,text/css,text/javascript,application/javascript"

URIEncoding="utf-8"

* **连接优化**

连接器及线程池的优化大于，能开多大开多大。问题是操作系统Linux单进程内不允许超过1000个线程，tomcat只是一个进程。线程池参数与connector参数一样。

<Connector executor="tomcatThreadPool"

connectionTimeout="20000"

minSpareThreads="100"

minProcessors="100“同时处理请求的最小数

maxProcessors=“1000”同时处理请求的最大数

maxConnections="1000" redirectPort="8443"

enableLookups="false" //禁用DNS查询 acceptCount="100" //繁忙时，可以放到处理队列中的请求数，超过这个数的请求将不予处理，默认设置 100 maxPostSize="10485760" // POST 提交方式，默认是 2097152(2兆)，配置为 10M。如果要禁用限制，则可以设置为 -1。 compression="on" disableUploadTimeout="true" compressionMinSize="2048" acceptorThreadCount="2" //用于接收连接的线程的数量，默认值是1。一般这个指需要改动的时候是因为该服务器是一个多核CPU，如果是多核 CPU 一般配置为 2. compressableMimeType="text/html,text/xml,text/plain,text/css,text/javascript,application/javascript" URIEncoding="utf-8" keepAliveTimeout="0"

/>

## 单机多个Tomcat提高并发

知道了单进程线程数不允许超1000，那么多进程就可以提高线程数，实际就是并发数提高了。线程太多引起操作系统切换也是耗时的，内存有限，多个tomcat因共享了java虚拟机参数，内存是等分的，需要权衡。

启用多个tomcat的操作非常简单，有必要知道。在yum 安装好tomcat后，服务就已经安装好了，不需要修改。在/lib/systemd/system目录下，有一个tomcat.service和tomcat@.service。这个@的就是多tomcat实例的服务，它是ok的，不要动。

复制一套配置到/var/lib/tomcats/1/conf目录下（就是从/etc/tomcat/），在conf目录同级，建work，logs等目录，权限设为tomcat属主。在/var/lib/tomcats/1/conf修改server.conf中webapps配置为你指向的第2个目录（记得端口换一个哦，8005是shutdown端口，宜关闭，ajp端口未使用时注释它）。虽然可以使用相同目录，但是可能application的一些单独配置需要修改，还是另外再指定一个目录好。然后：service tomcat@1 start就启动了另外一个实例！如果还要加，可以@2...@n这样下去，方便！

单机上多实例tomcat，在nginx的反向代理中添加一行不同的IP和端口，并可以设置权重，ip\_hash是指会话粘连，这样已经很好了，session共享不一定需要。

# 编程优化

## 编程原则

1、尽量指定类、方法的final修饰符，虚拟机会想办法内联所有的final方法

2、尽量重用对象，生存还是灭亡，这是个问题(哈姆雷特)

3、尽可能使用局部变量，一些局部变量是在栈中，免了垃圾回收过程

4、及时关闭流，不说了，资源浪费问题

5、尽量减少对变量的重复计算，list.size()在循环中的处理 for(int I = 0; I < list.size()…

6、尽量采用懒加载的策略，即在需要的时候才创建，还是创建对象问题，比如在if块外创建了对象 if(I < 1) { String nnn = “”

7、慎用异常，异常只能用于错误处理，不应该用来控制程序流程。

8、不要在循环中使用try…catch…，应该把其放在最外层

9、如果能估计到待添加的内容长度，为底层以数组方式实现的集合、工具类指定初始长度，

new HashMap(256)，StringBuilder(int size)

10、当复制大量数据时，使用System.arraycopy()命令，采用native

11、乘法和除法使用移位操作，乘法：<<，除法：>>

12、循环内不要不断创建对象引用，Object obj = null; for (int i = 0; i <= count; i++) { obj = new Object(); }

13、基于效率和类型检查的考虑，应该尽可能使用array，无法确定数组大小时才使用ArrayList

14、尽量使用HashMap、ArrayList、StringBuilder，除非线程安全需要，否则不推荐使用Hashtable、Vector、StringBuffer，

后三者由于使用同步机制而导致了性能开销

15、不要将数组声明为public static final，public不安全，而final没有用

16、尽量在合适的场合使用单例，节省加载开销

17、尽量避免随意使用静态变量，gc通常是不会回收， public class A{ private static B b = new B(); }，B会在A清除后才清除

18、及时清除不再需要的会话， HttpSession的invalidate()方法清除会话。

19、实现RandomAccess接口的集合比如ArrayList，应当使用最普通的for循环而不是foreach循环来遍历

假如是随机访问的，使用普通for循环效率将高于使用foreach循环；反过来，如果是顺序访问的，则使用Iterator会效率更高

20、使用同步代码块替代同步方法

21、将常量声明为static final，并以大写命名，常量池

22、不要创建一些不使用的对象，不要导入一些不使用的类，not use的警告，可以清除

23、程序运行过程中避免使用反射，尽量在启动时就反射完成，大家不关心启动花了多长时间

24、使用数据库连接池和线程池，这个都有了，可以使用的连接池：c3p0,dbcp…

25、使用带缓冲的输入输出流进行IO操作,即BufferedReader、BufferedWriter、BufferedInputStream、BufferedOutputStream，提升IO效率

26、顺序插入和随机访问比较多的场景使用ArrayList，元素删除和中间插入比较多的场景使用LinkedList

27、不要让public方法中有太多的形参，能用类就用类，不要直接使用属性

28、字符串变量和字符串常量equals的时候将字符串常量写在前面

29、请知道，在java中if (i == 1)和if (1 == i)是没有区别的，但从阅读习惯上讲，建议使用前者

30、不要对数组使用toString()方法，输出没有意义

31、不要对超出范围的基本数据类型做向下强制转型，长整型不适合强转为整型数

32、公用的集合类中不使用的数据一定要及时remove掉，因为内存泄漏问题

33、把一个基本数据类型转为字符串，基本数据类型.toString()是最快的方式、String.valueOf(数据)次之、数据+”"最慢

34、使用最有效率的方式去遍历Map， iterator接口

35、对资源的close()建议分开操作，分开关闭不同的资源，避免未释放 InputStream OutputStream os finally os.close, in.close,

36、JDK7中新引入的Objects工具类，对象比较

37、避免使用正则表达式,使用Apache Commons Lang作为代替

38、远离递归.递归会占用大量资源!

39、避免Random实例被多线程使用，虽然共享该实例是线程安全的，但会因竞争同一seed 导致的性能下降，

40、JDK7之后，可以使用ThreadLocalRandom来获取随机数

41、静态类、单例类、工厂类将它们的构造函数置为private

42、提前编译正则表达式

43、尽可能地缓存

## 针对性优化

内存缓存类的处理，一般使用静态的Map。将系统中一些字典数据，配置数据加载到Map中，避免访问数据。

本地缓存：guvua，ehcache等都可用，也方便。

共享缓存：redis或memcached，集群部署时才要用，一般没必要弄个麻烦。包括消息队列也是。

# 数据库优化

## 设计

数据库的设计肯定很重要，结构良好的关系表非常易于扩展，变动。有以下经验：

1. 主键是业务无关的（最重要），主键是bigint型最快，uuid我个人不常用
2. 功能表与业务区分开，附件、字典等
3. 层级关系建立冗余，parent\_ids字段很重要：0,1033,1057,3039,，记得以逗号结尾。在查询中直接追溯所有父节点（补0用in），所有子集的查询也仅需要like，这时有个逗号可以避免编号中继缺陷。
4. 先保持到第3范式，冗余在此基础上进行，否则就是乱来。
5. 文件存储在磁盘上，而不是数据库里
6. 命名全小写，下划线隔开单词，不要加通用前缀，但要加模块前缀。

## SQL优化

1. 子查询在查询列不要用，一律改为join，哪怕是left join也比它快
2. 分页机制要注意，mysql中有以下优化：Select \* from fentrust e

Inner join (select fid from fentrust limit 4100000, 10) a on a.fid = e.fid。这里小结果集先行，大大提高性能。

1. 多使用explain查看执行计划，参数解释如下：
   * Id，SQL执行的顺利的标识,SQL从大到小的执行.
   * select\_type，就是select类型,可以有以下几种
   * Table，显示这一行的数据是关于哪张表的.
   * Type，这列很重要,显示了连接使用了哪种类别,有无使用索引.  
     从最好到最差的连接类型为const、eq\_reg、ref、range、indexhe和ALL
   * possible\_keys，possible\_keys列指出MySQL能使用哪个索引在该表中找到行。
   * Key，key列显示MySQL实际决定使用的键（索引）。
   * key\_len，使用的索引的长度。在不损失精确性的情况下，长度越短越好
   * Ref，ref列显示使用哪个列或常数与key一起从表中选择行。
   * Rows，rows列显示MySQL认为它执行查询时必须检查的行数。
   * Extra，该列包含MySQL解决查询的详细信息,下面详细.

Using filesort   
看到这个的时候，查询需要优化。MYSQL需要进行额外的步骤来发现如何对返回的行排序。它根据连接类型以及存储排序键值和匹配条件的全部行的行指针来排序全部行   
Using temporary

看到这个的时候，查询需要优化。这里，MYSQL需要创建一个临时表来存储结果，这通常发生在对不同的列集进行ORDER BY上，而不是GROUP BY上

以上两个扩展信息要引起高度注意，需要优化。

1. 索引更重要

覆盖索引：是指查询结果集从索引中就能取到，不必走二次磁盘扫描

三星索引：是指查询列，where条件，order by条件，group by全在索引中可找到结果

组合索引：强力工具，当经常用的条件，按区分度排列形成组合索引，查询性能大大提高

count(1)比count(\*)快，因为count(\*)会转化为count(1)

适当的时候，建立汇总表

建立流水表

最后才能考虑分表，哪怕是先考虑表分区也行。也就是尽量让你的代码有价值，而不是经常废弃掉它。

不宜建索引的情况：

1. 参与计算列
2. 小表
3. 频繁更新的表
4. 字段集中度高

附：区分度的确定

SELECT COUNT(DISTINCT 列\_xx)/COUNT(\*) FROM 表

## 配置优化

**查询缓存**不要开，它是缓存了SQL级的结果集。当涉及的表有更新，它会强制更新。如果频繁写入的数据库，缓存经常更新对性能没有好处。

query\_cache\_size

Query\_cache里的数据在更新时，该表相关的语句全部置为失效，然后在写入更新。那么如果Query\_cache非常大，该表的查询结构又比较多，查询语句失效也慢，一个更新或是Insert就会很慢，这样看到的就是Update或是Insert怎么这么慢了。所以在数据库写入量或是更新量也比较大的系统，该参数不适合分配过大。而且在高并发，写入量大的系统，建议把该功能禁掉。

query\_cache\_limit

指定单个查询能够使用的缓冲区大小，缺省为1M

query\_cache\_min\_res\_unit

默认是4KB，设置值大对大数据查询有好处，但如果你的查询都是小数据查询，就容易造成内存碎片和浪费

**innodb的缓存配置**

innodb\_buffer\_pool\_size = 2048M

只需要用Innodb的话则可以设置它高达 70-80% 的可用内存。一些应用于 key\_buffer 的规则有 — 如果你的数据量不大，并且不会暴增，那么无需把 innodb\_buffer\_pool\_size 设置的太大了。

innodb\_additional\_mem\_pool\_size = 16M

网络传输中一次消息传输量的最大值。系统默认值 为1MB，最大值是1GB，必须设置1024的倍数。

innodb\_log\_files\_in\_group = 3

循环方式将日志文件写到多个文件。推荐设置为3

innodb\_lock\_wait\_timeout = 120

InnoDB 有其内置的死锁检测机制，能导致未完成的事务回滚。innodb\_file\_per\_table = 0

**其他重要的缓存配置**

sort\_buffer\_size = 2M

connection级参数。太大将导致在连接数增高时，内存不足。

max\_allowed\_packet = 32M

网络传输中一次消息传输量的最大值。系统默认值 为1MB，最大值是1GB，必须设置1024的倍数。

join\_buffer\_size = 2M

和sort\_buffer\_size一样，该参数对应的分配内存也是每个连接独享

tmp\_table\_size = 256M

默认大小是 32M。GROUP BY 多不多的问题

max\_heap\_table\_size = 256M

key\_buffer\_size = 2048M

索引的缓冲区大小，对于内存在4GB左右的服务器来说，该参数可设置为256MB或384MB。

read\_buffer\_size = 1M

read\_rnd\_buffer\_size = 16M

进行排序查询时，MySql会首先扫描一遍该缓冲，以避免磁盘搜索

bulk\_insert\_buffer\_size = 64M

批量插入数据缓存大小，可以有效提高插入效率，默认为8M

**线程池默认不需要配置，了解就可。**

## 监控软件

**innotop，安装简单，使用也简单。yum**

**phpMyAdmin，可以尝试。**

**lepus，安装复杂，有兴趣的朋友可以尝试。http://www.lepus.cc/page/product**

phpMyAdmin的安装参考：

从https://www.phpmyadmin.net/下载最新版，上传到服务器比如/data/www目录，解压命令为：tar –xvf 文件名

进入phpMyAdmin目录，进入libraries，打开config.default.inc，配置两行：

$cfg['blowfish\_secret'] = 'F0FfVNCpCh2172.31。。。';

$cfg['Servers'][$i]['host'] = '127.0.0.1';避免mysql连接不上。

修改nginx/conf.d目录中添加文件，命名为如：8888\_phpMyAdmin.conf。配置如下：

server {

listen 8888;

server\_name ip;

root /data/www/phpMyAdmin;#目录要正确，权限均改为nginx

location = /favicon.ico {

log\_not\_found off;

access\_log off;

}

location = /robots.txt {

allow all;

log\_not\_found off;

access\_log off;

}

location ~ \..\*/.\*\.php$ {

return 403;

}

location ~ ^/sites/.\*/private/ {

return 403;

}

location ~(^|/)\. {

return 403;

}

location / {

try\_files $uri @rewrite;

}

location @rewrite {

rewrite ^ /index.php;

}

location ~ \.php$ {

fastcgi\_split\_path\_info ^(.+\.php)(/.+)$;

fastcgi\_param SCRIPT\_FILENAME $document\_root$fastcgi\_script\_name;

fastcgi\_split\_path\_info ^(.+\.php)(.\*)$;

fastcgi\_param PATH\_INFO $fastcgi\_path\_info;

include fastcgi\_params;

fastcgi\_intercept\_errors on;

fastcgi\_pass unix:/dev/shm/php-cgi.sock;

}

location ~ ^/sites/.\*/files/styles/ {

try\_files $uri @rewrite;

}

location ~\* \.(js|css|png|jpeg|jpg|gif|ico)$ {

expires max;

log\_not\_found off;

}

}

然后重启nginx，或Reload一下

# 结语

性能优化深入话题是永远讲不完，个人实力有限，也不能面面俱到。深造的朋友可以自行深入研究。本文主要介绍思路，常规配置，使生疏的程序员快速成长为熟悉的程序员，知道性能调优的各个方面，以及深入研究的方向。