

# 影响一个系统性能的方方面面

一个 web 应用不是一个孤立的个体，它是一个系统的部分，系统中的每一部分都会影响整个系统的性能

## 常用的性能评价/测试指标

### 响应时间

提交请求和返回该请求的响应之间使用的时间，一般比较关注平均响应时间。

常用操作的响应时间列表：

操作	响应时间
打开一个站点	几秒
数据库查询一条记录（有索引）	十几毫秒
机械磁盘一次寻址定位	4 毫秒
从机械磁盘顺序读取 1M 数据	2 毫秒
从 SSD 磁盘顺序读取 1M 数据	0.3 毫秒
从远程分布式换成 Redis 读取一个数据	0.5 毫秒
从内存读取 1M 数据	十几微妙
Java 程序本地方法调用	几微妙
网络传输 2Kb 数据	1 微妙

### 并发数

同一时刻，对服务器有实际交互的请求数。

和网站在线用户数的关联：1000 个同时在线用户数，可以估计并发数在 5%到 15%之间，也就是同时并发数在 50~150 之间。

### 吞吐量

对单位时间内完成的工作量(请求)的量度

### 关系

系统吞吐量和系统并发数以及响应时间的关系：

理解为高速公路的通行状况：

吞吐量是每天通过收费站的车辆数目（可以换算成收费站收取的高速费），

并发数是高速公路上的正在行驶的车辆数目，

响应时间是车速。

车辆很少时，车速很快。但是收到的高速费也相应较少；随着高速公路上车辆数目的增多，车速略受影响，但是收到的高速费增加很快；

随着车辆的继续增加，车速变得越来越慢，高速公路越来越堵，收费不增反降；

如果车流量继续增加，超过某个极限后，任务偶然因素都会导致高速全部瘫痪，车走不动，当然后也收不着，而高速公路成了停车场（资源耗尽）。

# 常用的性能优化手段

## 避免过早优化

**不应该把大量的时间耗费在小的性能改进上，过早考虑优化是所有噩梦的根源。**

所以，我们应该编写清晰，直接，易读和易理解的代码，真正的优化应该留到以后，等到性能分析表明优化措施有巨大的收益时再进行。

但是过早优化，不表示我们应该编写已经知道的对性能不好的的代码结构。如《**4、编写高效优雅 Java 程序**》中的《**15、当心字符串连接的性能**》所说的部分。

## 进行系统性能测试

所有的性能调优，都应该建立在性能测试的基础上，直觉很重要，但是要用数据说话，可以推测，但是要通过测试求证。

## 寻找系统瓶颈，分而治之，逐步优化

性能测试后，对整个请求经历的各个环节进行分析，排查出现性能瓶颈的地方，定位问题，分析影响性能的主要因素是什么？内存、磁盘 IO、网络、CPU，还是代码问题？架构设计不足？或者确实是系统资源不足？

# 前端优化常用手段

## 浏览器/App

**减少请求数；**

合并 CSS，Js，图片

**使用客户端缓冲；**

静态资源文件缓存在浏览器中，有关的属性 **Cache-Control** 和 **Expires**

如果文件发生了变化，需要更新，则通过改变文件名来解决。

**启用压缩**

减少网络传输量，但会给浏览器和服务器带来性能的压力，需要权衡使用。

**资源文件加载顺序**

css 放在页面最上面，js 放在最下面

**减少 Cookie 传输**

cookie 包含在每次的请求和响应中，因此哪些数据写入 cookie 需要慎重考虑

**给用户一个提示**

有时候在前端给用户一个提示，就能收到良好的效果。毕竟用户需要的是不要不理他。

## CDN 加速

CDN，又称内容分发网络，本质仍然是一个缓存，而且是将数据缓存在用户最近的地方。

无法自行实现 CDN 的时候，可以考虑商用 CDN 服务。

## 反向代理缓存

将静态资源文件缓存在反向代理服务器上，一般是 Nginx。

## WEB 组件分离

将 **js**，**css** 和图片文件放在不同的域名下。可以提高浏览器在下载 **web** 组件的并发数。因为浏览器在下载同一个域名的数据存在并发数限制。

# 应用服务性能优化

## 缓存

网站性能优化第一定律：优先考虑使用缓存优化性能

Mark 老师的推论：缓存离用户越近越好

## 缓存的基本原理和本质

缓存是将数据存在访问速度较高的介质中。可以减少数据访问的时间，同时避免重复计算。

## 合理使用缓冲的准则

频繁修改的数据，尽量不要缓存，读写比 2:1 以上才有缓存的价值。

缓存一定是热点数据。

应用需要容忍一定时间的数据不一致。

缓存可用性问题，一般通过热备或者集群来解决。

缓存预热，新启动的缓存系统没有任何数据，可以考虑将一些热点数据提前加载到缓存系统。

解决缓存击穿：

1、布隆过滤器，或者 2、把不存在的数据也缓存起来，比如有请求总是访问 **key = 23** 的数据，但是这个 **key = 23** 的数据在系统中不存在，可以考虑在缓存中构建一个 (**key=23 value = null**) 的数据。

## 分布式缓存与一致性哈希

以集群的方式提供缓存服务，有两种实现：

- 1、需要更新同步的分布式缓存，所有的服务器保存相同的缓存数据，带来的问题就是，缓存的数据量受限制，其次，数据要在所有的机器上同步，代价很大。
- 2、每台机器只缓存一部分数据，然后通过一定的算法选择缓存服务器。常见的余数 **hash** 算法存在当有服务器上下线的时候，大量缓存数据重建的问题。所以提出了一致性哈希算法。

一致性哈希：

1. 首先求出服务器（节点）的哈希值，并将其配置到 0~232 的圆（**continuum**）上。
2. 然后采用同样的方法求出存储数据的键的哈希值，并映射到相同的圆上。
3. 然后从数据映射到的位置开始顺时针查找，将数据保存到找到的第一个服务器上。如果超过 232 仍然找不到服务器，就会保存到第一台服务器上。

一致性哈希算法对于节点的增减都只需重定位环空间中的一小部分数据，具有较好的容错性和可扩展性。

一致性哈希算法在服务节点太少时，容易因为节点分部不均匀而造成数据倾斜问题，此时必然造成大量数据集中到 **Node A** 上，而只有极少量会定位到 **Node B** 上。为了解决这种数据倾斜问题，一致性哈希算法引入了虚拟节点机制，即对每一个服务节点计算多个哈希，每个计算结果位置都放置一个此服务节点，称为虚拟节点。具体做法可以在服务器 **ip** 或主机名的后面增加编号来实现。例如，可以为每台服务器计算三个虚拟节点，于是可以

分别计算 “Node A#1”、“Node A#2”、“Node A#3”、“Node B#1”、“Node B#2”、“Node B#3”的哈希值，于是形成六个虚拟节点：同时数据定位算法不变，只是多了一步虚拟节点到实际节点的映射，例如定位到“Node A#1”、“Node A#2”、“Node A#3”三个虚拟节点的数据均定位到 Node A 上。这样就解决了服务节点少时数据倾斜的问题。在实际应用中，通常将虚拟节点数设置为 32 甚至更大，因此即使很少的服务节点也能做到相对均匀的数据分布。

## 异步

### 同步和异步，阻塞和非阻塞

同步和异步关注的是结果消息的通信机制

同步:同步的意思就是调用方需要主动等待结果的返回

异步:异步的意思就是不需要主动等待结果的返回，而是通过其他手段比如，状态通知，回调函数等。

阻塞和非阻塞主要关注的是等待结果返回调用方的状态

阻塞:是指结果返回之前，当前线程被挂起，不做任何事

非阻塞:是指结果在返回之前，线程可以做一些其他事，不会被挂起。

1.同步阻塞:同步阻塞基本也是编程中最常见的模型，打个比方你去商店买衣服，你去了之后发现衣服卖完了，那你就在店里面一直等，期间不做任何事(包括看手机)，等着商家进货，直到有货为止，这个效率很低。jdk 里的 BIO 就属于 同步阻塞

2.同步非阻塞:同步非阻塞在编程中可以抽象为一个轮询模式，你去了商店之后，发现衣服卖完了，这个时候不需要傻傻的等着，你可以去其他地方比如奶茶店，买杯水，但是你还是需要时不时的去商店问老板新衣服到了吗。jdk 里的 NIO 就属于 同步非阻塞

3.异步阻塞:异步阻塞这个编程里面用的较少，有点类似你写了个线程池,submit 然后马上 future.get(), 这样线程其实还是挂起的。有点像你去商店买衣服，这个时候发现衣服没有了，这个时候你就给老板留给电话，说衣服到了就给我打电话，然后你就守着这个电话，一直等着他响什么事也不做。这样感觉的确有点傻，所以这个模式用得比较少。

4.异步非阻塞:好比你去商店买衣服，衣服没了，你只需要给老板说这是我的电话，衣服到了就打。然后你就随心所欲的去玩，也不用操心衣服什么时候到，衣服一到，电话一响就可以去买衣服了。jdk 里的 AIO 就属于异步

### 常见异步的手段

#### Servlet 异步

servlet3 中才有，支持的 web 容器在 tomcat7 和 jetty8 以后。

#### 多线程

#### 消息队列

## 集群

可以很好的将用户的请求分配到多个机器处理，对总体性能有很大的提升

## 程序

### 代码级别

一个应用的性能归根结底取决于代码是如何编写的。

## 选择合适的数据结构

```
public static void main(String[] args) {  
  
    List<Object> list = new LinkedList<>();  
    int i=0;  
    while(true) {  
        i++;  
        if(i%10000==0) System.out.println("i="+i);  
        list.add(new Object());  
    }  
}
```

选择 ArrayList 和 LinkedList 对我们的程序性能影响很大，为什么？因为 ArrayList 内部是数组实现，存在着不停的扩容和数据复制。

## 选择更优的算法

举个例子，最大子列和问题：

给定一个整数序列， $a_0, a_1, a_2, \dots, a_n$ （项可以为负数），求其中最大的子序列和。

如果所有整数都是负数，那么最大子序列和为 0；

例如  $(a[1], a[2], a[3], a[4], a[5], a[6]) = (-2, 11, -4, 13, -5, -2)$  时，

最大子段和为 20，子段为  $a[2], a[3], a[4]$ 。

最坏的算法：穷举法，所需要的计算时间是  $O(n^3)$ 。

一般的算法：分治法的计算时间复杂度为  $O(n \log n)$ 。

最好的算法：最大子段和的动态规划算法，计算时间复杂度为  $O(n)$

$n$  越大，时间就相差越大，比如 10000 个元素，最坏的算法和最好的算法之间的差距绝非多线程或者集群化能轻松解决的。

## 编写更少的代码

同样正确的程序，小程序比大程序要快，这点无关乎编程语言。

# 并发编程

充分利用 CPU 多核，

实现线程安全的类，避免线程安全问题

同步下减少锁的竞争

# 资源的复用

目的是减少开销很大的系统资源的创建和销毁，比如数据库连接，网络通信连接，线程资源等等。

单例模式

池化技术

# JVM

## 与 JIT 编译器相关的优化

对 JVM 性能影响最大的是编译器。选择编译器是运行 java 程序首先要做的选择之一

## 热点编译的概念

对于程序来说，通常只有一部分代码被经常执行，这些关键代码被称为应用的热点，执行的越多就认为是越热。将这些代码编译为本地机器特定的二进制码，可以有效提高应用性能。

## 选择编译器类型

-server, 更晚编译, 但是编译后的优化更多, 性能更高

-client, 很早就开始编译

-XX:+TieredCompilation, 开启分层编译, 可以让 jvm 在启动时启用 client 编译, 随着代码变热后再转为 server 编译。

缺省编译器取决于机器位数、操作系统和 CPU 数目。32 位的机器上, 一般默认都是 client 编译, 64 位机器上一般都是 server 编译, 多核机器一般是 server 编译。

```
D:\XiangXue\vipLesson\JVM\code\src\vip-jvm\bin\com\xiangxue\test>java -version
java version "1.8.0_101"
Java(TM) SE Runtime Environment (build 1.8.0_101-b13)
Java HotSpot(TM) 64-Bit Server VM (build 25.101-b13, mixed mode)
```

中的 mix mode 一般指编译时机:

-Xint 表示禁用 JIT, 所有字节码都被解释执行, 这个模式的速度最慢的。

-Xcomp 表示所有字节码都首先被编译成本地代码, 然后再执行。

-Xmixed, 默认模式, 让 JIT 根据程序运行的情况, 有选择地将某些代码编译成本地代码。

-Xcomp 和 -Xmixed 到底谁的速度快, 针对不同的程序可能有不同的结果, 基本还是推荐用默认模式。

## 代码缓存相关

在编译后, 会有一个代码缓存保存编译后的代码, 一旦这个缓存满了, jvm 将无法继续编译代码。

当 jvm 提示: CodeCache is full, 就表示需要增加代码缓存大小。

-XX:ReservedCodeCacheSize=N 可以用来调整这个大小。

表4-6: 各种平台上代码缓存的默认大小

JVM类型	代码缓存的默认大小
32 位 client, Java 8	32 MB
32 位 server, 分层编译, Java 8	240 MB
64 位 server, 分层编译, Java 8	240 MB
32 位 client, Java 7	32 MB
32 位 server, Java 7	32 MB
64 位 server, Java 7	48 MB
64 位 server, 分层编译, Java 7	96 MB

## 编译阈值

代码是否进行编译, 取决于代码执行的频度, 是否到达编译阈值。

计数器有两种: 方法调用计数器和方法里的循环回边计数器

一个方法是否达到编译阈值取决于方法中的两种计数器之和。编译阈值调整的参数为: -XX:CompileThreshold=N

方法调用计数器统计的并不是方法被调用的绝对次数, 而是一个相对的执行频率, 即一段时间之内方法被调用的次数。当超过一定的时间限度, 如果方法的调用次数仍然不足以让它提交给即时编译器编译, 那这个方法的调用计数器就会被减少一半, 这个过程称为方法调用计数器热度的衰减 (Counter Decay), 而这段时间就称为此方法统计的半衰周期 (Counter Half Life Time)。进行热度衰减的动作是在虚拟机进行垃圾收集时顺便进行的, 可以使用虚拟机参数-XX: -UseCounterDecay 来关闭热度衰减, 让方法计数器统计方

法调用的绝对次数，这样，只要系统运行时间足够长，绝大部分方法都会被编译成本地代码。另外，可以使用 **-XX: CounterHalfLifeTime** 参数设置半衰周期的时间，单位是秒。与方法计数器不同，回边计数器没有计数热度衰减的过程，因此这个计数器统计的就是该方法循环执行的绝对次数。

## 编译线程

进行代码编译的时候，是采用多线程进行编译的。

## 方法内联

内联默认开启，**-XX:-Inline**，可以关闭，但是不要关闭，一旦关闭对性能有巨大影响。

方法是否内联取决于方法有多热和方法的大小，

很热的方法如果方法字节码小于 325 字节才会内联，这个大小由参数 **-**

**XX:MaxFreqInlinesSzie=N** 调整，但是这个很热与热点编译不同，没有任何参数可以调整热度。

方法小于 35 个字节码，一定会内联，这个大小可以通过参数 **-XX:MaxInlinesSzie=N** 调整。

## 逃逸分析

是 JVM 所做的最激进的优化，最好不要调整相关的参数。

# GC 调优

## 目的

GC 的时间够小

GC 的次数够少

发生 Full GC 的周期足够的长，时间合理，最好是不发生。

## 调优的原则和步骤

1. 大多数的 java 应用不需要 GC 调优
2. 大部分需要 GC 调优的，不是参数问题，是代码问题
3. 在实际使用中，分析 GC 情况优化代码比优化 GC 参数要多得多；
4. GC 调优是最后的手段

GC 调优的最重要的三个选项：

第一位：选择合适的 GC 回收器

第二位：选择合适的堆大小

第三位：选择年轻代在堆中的比重

## 步骤

### 1，监控 GC 的状态

使用各种 JVM 工具，查看当前日志，分析当前 JVM 参数设置，并且分析当前堆内存快照和 gc 日志，根据实际的各区域内存划分和 GC 执行时间，觉得是否进行优化；

### 2，分析结果，判断是否需要优化

如果各项参数设置合理，系统没有超时日志出现，GC 频率不高，GC 耗时不高，那么没有必要进行 GC 优化；如果 GC 时间超过 1-3 秒，或者频繁 GC，则必须优化；

注：如果满足下面的指标，则一般不需要进行 GC：

Minor GC 执行时间不到 50ms；

Minor GC 执行不频繁，约 10 秒一次；

**Full GC 执行时间不到 1s；**

Full GC 执行频率不算频繁，不低于 10 分钟 1 次；

### 3, 调整 GC 类型和内存分配

如果内存分配过大或过小, 或者采用的 GC 收集器比较慢, 则应该优先调整这些参数, 并且先找 1 台或几台机器进行 beta, 然后比较优化过的机器和没有优化的机器的性能对比, 并有针对性的做出最后选择;

### 4, 不断的分析和调整

通过不断的试验和试错, 分析并找到最合适的参数

### 5, 全面应用参数

如果找到了最合适的参数, 则将这些参数应用到所有服务器, 并进行后续跟踪。

### 学会阅读 GC 日志

以参数-Xms5m -Xmx5m -XX:+PrintGCDetails -XX:+UseSerialGC 为例:

```
[DefNew: 1855K->1855K(1856K), 0.0000148 secs][Tenured: 2815K->4095K(4096K), 0.0134819 secs] 4671K
```

DefNew 指明了收集器类型, 而且说明了收集发生在新生代。

1855K->1855K(1856K)表示, 回收前 新生代占用 1855K, 回收后占用 1855K, 新生代大小 1856K。

0.0000148 secs 表明新生代回收耗时。

Tenured 表明收集发生在老年代

2815K->4095K(4096K), 0.0134819 secs: 含义同新生代

最后的 4671K 指明堆的大小。

收集器参数变为-XX:+UseParNewGC, 日志变为:

```
[ParNew: 1856K->1856K(1856K), 0.0000107 secs][Tenured: 2890K->4095K(4096K), 0.0121148 secs]
```

收集器参数变为-XX:+ UseParallelGC 或 UseParallelOldGC, 日志变为:

```
[PSYoungGen: 1024K->1022K(1536K)] [ParOldGen: 3783K->3782K(4096K)] 4807K->4804K(5632K),
```

CMS 收集器和 G1 收集器会有明显的相关字样

### 其他与 GC 相关的参数

调试跟踪之 打印简单的 GC 信息 参数: -verbose:gc, -XX:+PrintGC

打印详细的 GC 信息 -XX:+PrintGCDetails, +XX:+PrintGCTimeStamps

-Xlogger:logpath 设置 gc 的日志路, 如: -Xlogger:log/gc.log, 将 gc.log 的路径设置到当前目录的 log 目录下。

应用场景: 将 gc 的日志独立写入日志文件, 将 GC 日志与系统业务日志进行了分离, 方便开发人员进行追踪分析。

-XX:+PrintHeapAtGC, 打印堆信息

参数设置: -XX: +PrintHeapAtGC

应用场景: 获取 Heap 在每次垃圾回收前后的使用状况

-XX:+TraceClassLoading

参数方法: -XX:+TraceClassLoading

应用场景: 在系统控制台信息中看到 class 加载的过程和具体的 class 信息, 可用以分析类的加载顺序以及是否可进行精简操作。

-XX:+DisableExplicitGC 禁止在运行期显式地调用 System.gc()

-XX:-HeapDumpOnOutOfMemoryError 默认关闭, 建议开启, 在

java.lang.OutOfMemoryError 异常出现时, 输出一个 dump.core 文件, 记录当时的堆内存快照。



-XX:HeapDumpPath=./java\_pid<pid>.hprof 默认是 java 进程启动位置, 用来设置堆内存快照的存储文件路径。

## 推荐策略

### 1. 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.
- 避免设置过小.当新生代设置过小时会导致:1.YGC 次数更加频繁 2.可能导致 YGC 对象直接进入旧世代,如果此时旧世代满了,会触发 FGC.

年老代大小选择

0. 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。

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

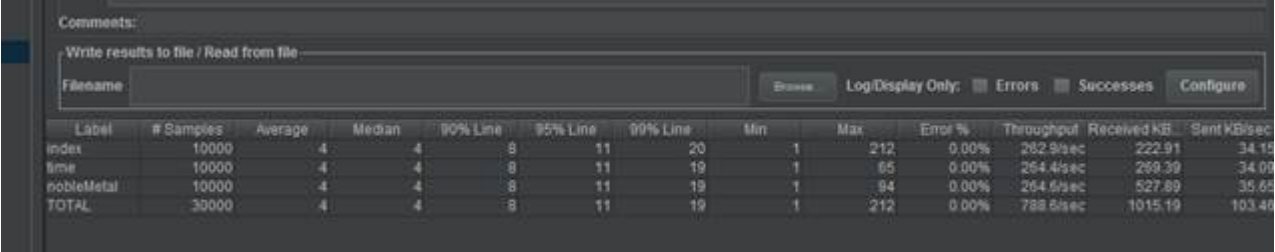
## 调优实战

### 不同的内存大小

参见视频。

### 不同的 GC 回收器

参见视频

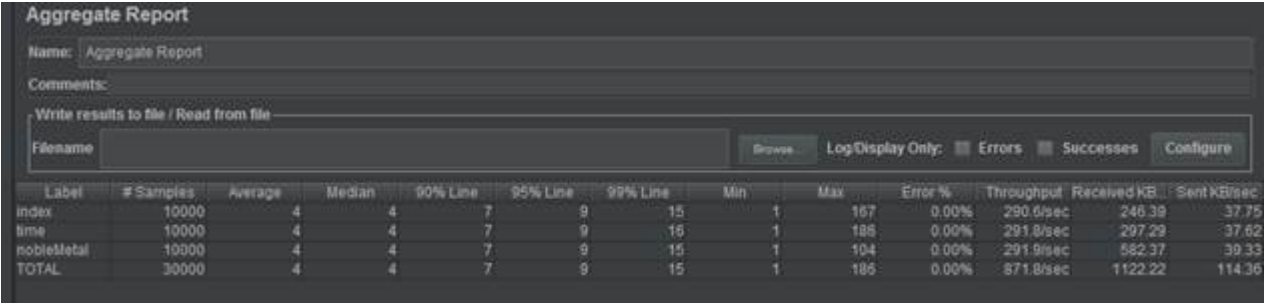


Comments:

Write results to file / Read from file

Filename:  Browse Log/Display Only: ☐ Errors ☐ Successes

Label	# Samples	Average	Median	90% Line	95% Line	99% Line	Min	Max	Error %	Throughput	Received KB	Sent KB/sec
index	10000	4	4	8	11	20	1	212	0.00%	282.9/sec	222.91	34.15
time	10000	4	4	8	11	19	1	65	0.00%	264.4/sec	269.39	34.09
nobleMetal	10000	4	4	8	11	19	1	94	0.00%	264.6/sec	527.69	35.65
TOTAL	30000	4	4	8	11	19	1	212	0.00%	788.5/sec	1015.19	103.46



Aggregate Report

Name:

Comments:

Write results to file / Read from file

Filename:  Browse Log/Display Only: ☐ Errors ☐ Successes

Label	# Samples	Average	Median	90% Line	95% Line	99% Line	Min	Max	Error %	Throughput	Received KB	Sent KB/sec
index	10000	4	4	7	9	15	1	167	0.00%	290.6/sec	246.39	37.75
time	10000	4	4	7	9	16	1	186	0.00%	291.8/sec	297.29	37.62
nobleMetal	10000	4	4	7	9	15	1	104	0.00%	291.9/sec	582.37	39.33
TOTAL	30000	4	4	7	9	15	1	186	0.00%	871.8/sec	1122.22	114.36

Name: Aggregate Report

Comments:

Write results to file / Read from file

Filename   Log/Display Only: ☐ Errors ☐ Successes

Label	# Samples	Average	Median	90% Line	95% Line	99% Line	Min	Max	Error %	Throughput	Received KB	Sent KB/sec
index	10000	4	4	7	9	14	0	139	0.00%	307.1/sec	260.34	39.88
time	10000	4	4	7	9	14	0	99	0.00%	308.4/sec	314.14	39.75
nobleMetal	10000	4	4	7	9	13	0	99	0.00%	308.5/sec	615.57	41.58
TOTAL	30000	4	4	7	9	14	0	139	0.00%	921.0/sec	1185.65	120.83

# 存储性能优化

尽量使用 **SSD**

定时清理数据或者按数据的性质分开存放

结果集处理

用 `setFetchSize` 控制 `jdbc` 每次从数据库中返回多少数据。

## 总结：

调优是个很复杂、很细致的过程，要根据实际情况调整，不同的机器、不同的应用、不同的性能要求调优的手段都是不同的。也没有一个放之四海而皆准的配置或者公式。**mark** 老师也无法告诉大家全部与性能相关的知识，即使是 `jvm` 参数也是如此，再比如说性能有关的操作系统工具，和操作系统本身相关的所谓大页机制，都需要大家平时去积累，去观察，去实践。

**mark** 老师在这个专题上告诉大家的除了各种 **java** 虚拟机基础知识、内部原理，也告诉大家一个性能优化的一个基本思路和着手的方向。