# SSD商用

## 存储

在一定条件下,RAM 访问仅需要大概 540 个 CPU 时钟周期,而磁盘访问则需要花费大概 20 000 000 个 CPU 时钟周期。存储系统需要考虑的问题：

1.磁盘空间

2.磁盘组性能。磁盘空间主要取决于磁盘阵列类型及磁盘个数。而磁盘性能指标主要就是：吞吐量（传输带宽）和磁盘IOPS。

* 大容量大吞吐，GFS，响应不要求高，顺序读，SATA (存储图片视频等文件)

磁盘阵列：

1. 直连式存储 (DAS：Direct Attached Storage)

是指将存储设备通过SAS/SATA接口或FC接口直接连接到一台计算机上。DAS不算是网络存储，因为只有它所挂载的主机才可访问它。

1. 网络存储设备 (NAS：Network Attached Storage)

是指将存储设备通过标准的网络拓扑结构（例如以太网），连接到一群计算机上。NAS有文件系统和IP地址，可以类似的理解为网上邻居的共享磁盘。在 NAS 存储结构中,存储系 统不再通过 I/O 总线附属于某个服务器或客户机,而是直接通过网络接口与网络直接相连, 由用户通过网络访问。

1. 存储网络 (SAN：Storage Area Network)

SAN是一种通过光纤交换机等连接设备将磁盘阵列、磁带等存储设备与相关服务器连接起来的高速专用子网。2Gbps 光纤通道,（250MB/s）, 4Gbps 光纤通道（500MB/S）

目前的SAN存储有2种：一是基于光纤通道的FC SAN；二是基于以太网的IP SCAN(也就常说的iSCSI)。iSCSI作为共享于以太网络上的存储则更类似于NAS。

FC SAN通过光纤交换机连接到主机(HBA卡)，也就是说可以连接到光纤交换机的主机都可以访问这个存储；服务器和客户机的数据通信是通过 SCSI 命令而非 TCP/IP 来实现的,数据处理是"块级 "(block level)。

* 数据库，小记录关系复杂，随机读写 SSD，sas辅助 （百度2011年全面SSD）

高并发，cache, 100G以内 MEMORY,ssd辅助

**IOPS** (Input/Output Operations Per Second)，即每秒进行读写（I/O）操作的次数，多用于数据库等场合，衡量随机访问的性能。

随机读写频繁的应用，如OLTP(Online Transaction Processing)，IOPS是关键衡量指标。

另一个重要指标是数据**吞吐量**(Throughput)，指单位时间内可以成功传输的数据数量。对于大量顺序读写的应用，如电视台的视频编辑，视频点播VOD(Video On Demand)，则更关注吞吐量指标。

存储的瓶颈有两个：一个是IOPS，另一个是吞吐量。

单块SAS盘的IOPS为150，如果每个IO最大为1MB，那么我们可以计算出磁盘的吞吐量瓶颈为150MB（实测吞吐量大致为170-250MB）。我们可以看到，IOPS是磁盘真正的瓶颈，随机IO对磁盘是致命的，而吞吐量对于磁盘通常不是瓶颈，所以磁盘更适合追求吞吐量的系统。

# 机械硬盘

传统磁盘完成一个I/O请求所花费的时间，它由寻道时间、旋转延迟和数据传输时间三部分构成。

寻道时间Tseek是指将读写磁头移动至正确的磁道上所需要的时间。寻道时间越短，I/O操作越快，目前磁盘的平均寻道时间一般在3－15ms。  
  
       旋转延迟Trotation是指盘片旋转将请求数据所在扇区移至读写磁头下方所需要的时间。旋转延迟取决于磁盘转速，通常使用磁盘旋转一周所需时间的1/2表示。比如，7200 rpm的磁盘平均旋转延迟大约为60\*1000/7200/2 = 4.17ms，而转速为15000 rpm的磁盘其平均旋转延迟约为2ms。 rpm: 每分钟转速  
  
       数据传输时间Ttransfer是指完成传输所请求的数据所需要的时间，它取决于数据传输率，其值等于数据大小除以数据传输率。目前IDE/ATA能达到133MB/s，SATA II可达到300MB/s的接口数据传输率，数据传输时间通常远小于前两部分时间。因此，理论上可以计算出磁盘的最大IOPS，即IOPS = 1000 ms/ (Tseek + Troatation)，忽略数据传输时间。假设磁盘平均物理寻道时间为3ms, 磁盘转速为7200,10K,15K rpm，则磁盘IOPS理论最大值分别为，

IOPS = 1000 / (3 + 60000/7200/2) = 140  
IOPS = 1000 / (3 + 60000/10000/2) = 167  
IOPS = 1000 / (3 + 60000/15000/2) = 200

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Device** | **Type** | **IOPS** | **Interface** | **Notes** |
| 7,200 [rpm](http://en.wikipedia.org/wiki/Revolutions_per_minute) SATA drives | [HDD](http://en.wikipedia.org/wiki/Hard_disk_drive) | ~75-100 IOPS | SATA 3 Gbit/s |  |
| 10,000 rpm SATA drives | HDD | ~125-150 IOPS | SATA 3 Gbit/s |  |
| 10,000 rpm SAS drives | HDD | ~140 IOPS | SAS |  |
| 15,000 rpm SAS drives | HDD | ~175-210 IOPS | SAS |  |

测试工具：

wget <http://fossies.org/linux/privat/fio-2.1.6.1.tar.gz>

bs=16k           单次io的块文件大小为16k

rw=randwrite        测试随机写的I/O  
rw=randrw          测试随机写和读的I/O

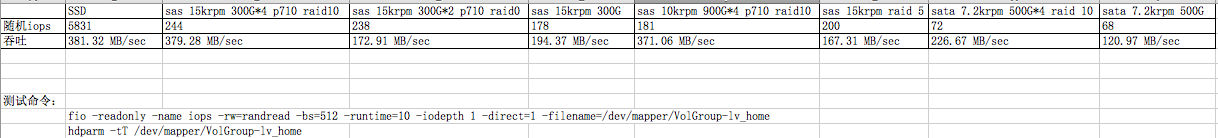
numjobs=30         本次的测试线程为30.

runtime=1000        测试时间为1000秒

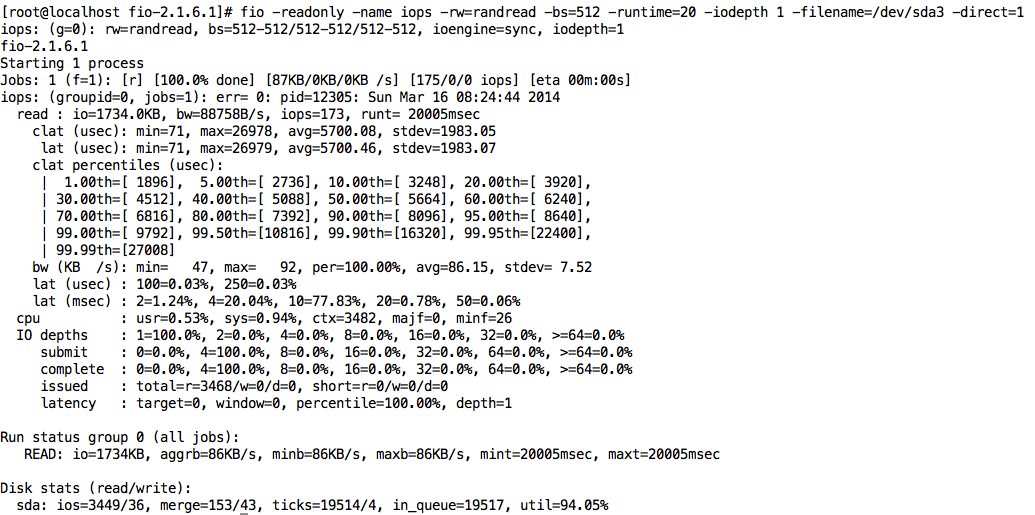
fio -readonly -name iops -rw=randread -bs=512 -runtime=10 -iodepth 1 -direct=1 -filename=/dev/mapper/VolGroup-lv\_home

fio -readonly -name iops -rw=read -bs=512 -runtime=20 -iodepth 1 -filename=/dev/sda1 -direct=1

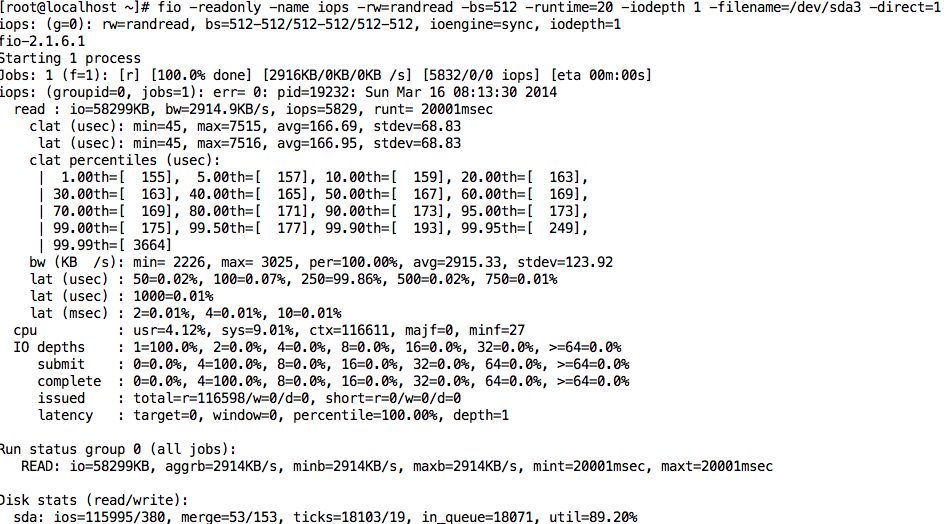
fio -name iops -rw=write -bs=4k -runtime=20 -iodepth 1 -filename /dev/sda1 -direct=1



sas 10k RPM raid 10 centos 6.4 64bit p710



ssd SLC no raid cent 6.4 64bit



# SSD

由主控芯片与NAND闪存芯片组成，NAND闪存负责重要的存储任务，NAND存储单元又分为两类：SLC（SINGLE LAYER CELL 单层单元）和MLC（MULTI-LEVEL CELL多层单元）。SLC的特点是成本高、容量小、但是速度快，而MLC的特点是容量大成本低，但是速度慢。SLC即Single-Level Cell，单层式存储，每个存储单元仅能储存1bit数据，MLC(Multi-Level Cell)可储存2bit数据，TLC(Trinary-Level)可储存3bit数据。我们知道，随着存储单元的位数越多，向每个单元存储中加入更多的数据会使得状态难以辨别，并且可靠性、耐用性和性能都会降低。

**SLC是单极单元，MLC是多级单元，两者的差异在于每单元存储的数据量（密度），SLC每单元只存储一位，只包含0和1两个电压符，MLC每单元可以存储两位，包含四个电压符（00,01,10,11）。显然，MLC的存储容量比SLC大，但是SLC更简单可靠，SLC读取和写入的速度都比MLC更快，而且SLC比MLC更耐用，MLC每单元可擦除1w次，而SLC可擦除10w次。**

MLC的每个单元是2BIT的，相对SLC来说整整多了一倍。由于每个MLC存储单元中存放的资料较多，结构相对复杂，出错的几率会增加，必须进行错误修正，这个动作导致其性能大幅落后于结构简单的SLC闪存。此外，SLC闪存的优点是复写次数高达10W次，比MLC闪存1W次高10倍。此外，为了保证MLC的寿命，控制芯片都校验和智能磨损平衡技术算法，使得每个存储单元的写入次数可以平均分摊，达到100万小时故障间隔时间(MTBF)，达到20年以上的使用年限。

读取速度快（MLC）读写都快（SLC）

wear leveling算法写入流程：

基本原理是在SSD中设置了两个block pool，一个是free block pool（空闲池），一个是数据池（data block pool），当需要改写某个page时（如果写入原有位置，必须先擦除整个block，然后才能写入数据），并不写入原有位置（不需要擦除的动作），而是从空闲池中取出新的block，将现有的数据和需要改写的数据合并为新的block，一起写入新的空白block，原有的block被标识为invalid状态（等待被擦除回收），新的block则进入数据池。后台任务会定时从data block中取出无效数据的block，擦除后回收到空闲池中。这样做的好处在于，一是不会反复擦写同一个block，二是写入的速度会比较快(省略了擦除的动作)。

因为SSD的erase-before-write的特性，所以就出现了一个写入放大的概念，比如你想改写4K的数据，必须首先将整个擦除块（512KB）中的数据读出到缓存中，改写后，将整个块一起写入，这时你实际写入了512KB的数据，写入放大系数是128。写入放大最好的情况是1，就是不存在放大的情况。

通过为SSD预留更多空间，可以显著缓解写入放大导致的性能问题。根据我们的测试结果，MLC SSD在长时间的随机写入后，性能下降很明显（随机写IOPS甚至降低到300）。如果为wear leveling预留更多空间，就可以显著改善MLC SSD在长时间写操作之后的性能下降问题，而且保留的空间越多，性能提升就越明显。

**接口**

目前SATA可以提供150MB/s的高峰传输速率。今后将达到300 MB/s和600 MB/s。Dell的3Gbps也就是300MB/S

SAS-3可以做到12Gbps

* fusion-io卡 （2倍）750G 价格 5-6W (支付宝系统方案)

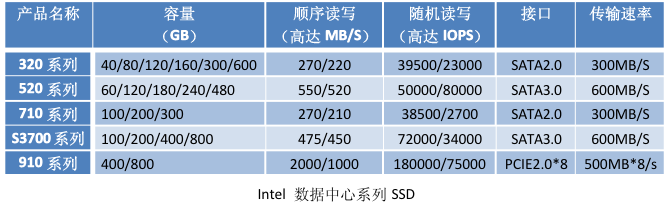
Fusion-io所产的ioDrive2 1.2TB，采用PCI-E 2.0 x4接口，连续读写速度可以达到1GB/s，1.5GB/s

* Pci-e ssd卡 100G+200G 2w （2倍速）
* 混搭 SAS+ssd

DELL不建议3Gps/s MLS和10k rpm hdd的性能类似。

CacheCade SSD DELL反馈实际生成效果不是很好

SSD硬盘的IOPS优势明显，如下位intel的ssd系列产品的测试数据：



从测试数据分析来看，SSD写入40T数据，磨损消耗仅为1%，而且基本没有坏块。这是在系统没有做任何优化的情况下，即将所有数据文件都放在SSD上的结果，这证明SSD的损耗均衡算法是很靠谱的。如果磨损是线性的，根据数据可以计算得出，单块SSD(64G)可以写入4PB的数据，与厂商提供的数据相符。而且，40T的数据大部分是系统测试时写入的，真实系统上线后写入的数据量很小，根据计算，我们的SSD如果磨损耗尽，可能还需要几十年的时间。

SSD应用场景：

数据库

在互联网的细分行业中，搜索和CDN对SSD的需求最多，这些行业有个普遍的特点，对于硬盘随机读写的性能要求很高，相对于机械硬盘400 IOPS SSD随机读写达到了几万甚至十万以上的IOPS，可见拥有大量客户访问量的互联网行业应用SSD，会大大改善用户的体验。

# Fusion-IO

官方数据：



Raid卡

开启BBWC

BBU

**文件系统与IO策略**

* ext4 (or xfs), mount with noatime nodiratime
* Scheduler – use deadline or noop

data=writeback

文件系统日志有三种日志模式：data=journal、data=ordered和data=writeback。默认设置data=ordered提供性能和防护之间的最佳平衡。但是如果你的服务器需要写入大量数据，它可以长期冻结你的服务器。如果是这种情况，使用iotop等工具，你会看到kjournald程序的高负载。如果你的服务器遇到了这种行为，使用data=writeback选项来获得更好的写入性能。

日志写磁盘过程

常用的文件系统使用日志功能来保证文件系统的完整性。该功能背后的思路很简单：在写入新的数据块到磁盘之前，会先将元数据写入日志。预先将元数据写入日志可以保证在写入真实数据前后一旦发生错误，日志功能能很容易地回滚到更改之前的状态。这个方法确保了不会发生文件系统崩溃的情况。

磁盘上配有内部缓存，以便重新调整批量数据的写操作顺序，优化写入性能，因此文件系统必须在日志数据写入磁盘之后才能写commit记录，若commit 记录写入在先，而日志有可能损坏，那么就会影响数据完整性。Ext4默认启用barrier，只有当barrier之前的数据全部写入磁盘，才能写 barrier之后的数据。

Trim功能： discard (先不擦除，只做标记)

http://www.oenhan.com/linux-ssd-optimization

# vi /etc/fstab

ext4 defaults,async,noatime,nodiratime,data=writeback,barrier=0

/dev/sdf /tmp1 ext4 defaults,async,noatime,nodiratime,discard,data=writeback,barrier=0 0 0

# mount -o remount /dev/sda1

# mount

<http://news.watchstor.com/industry-130156.htm>

<http://www.xzcblog.com/post-92.html>

可以通过dmesg查看文件系统参数

vm.swappiness=0

* 选择EXT4文件系统+TRIM模式（mount -o defaults,noatime,nodiratime,barrier=0,discard）
* I/O Schedulers调度算法，可以使用NOOP或者Deadline算法
* 内核参数调整，SSD所在硬盘，echo 0 > /sys/block/sda/queue/rotational

NOOP算法的全写为No Operation。该算法实现了最最简单的FIFO队列，所有IO请求大致按照先来后到的顺序进行操作。之所以说“大致”，原因是NOOP在FIFO的基础上还做了相邻IO请求的合并，并不是完完全全按照先进先出的规则满足IO请求。

DEADLINE在CFQ的基础上，解决了IO请求饿死的极端情况。除了CFQ本身具有的IO排序队列之外，DEADLINE额外分别为读IO和写IO提供了FIFO队列。读FIFO队列的最大等待时间为500ms，写FIFO队列的最大等待时间为5s。FIFO队列内的IO请求优先级要比CFQ队列中的高，，而读FIFO队列的优先级又比写FIFO队列的优先级高。优先级可以表示如下：   
FIFO(Read) > FIFO(Write) > CFQ 

ANTICIPATORY的在DEADLINE的基础上，为每个读IO都设置了6ms 的等待时间窗口。如果在这6ms内OS收到了相邻位置的读IO请求，就可以立即满足. Anticipatory 的中文含义是”预料的, 预想的”, 这个词的确揭示了这个算法的特点，简单的说，有个 IO 发生的时候，如果又有进程请求 IO 操作，则将产生一个默认的 6 毫秒猜测时间，猜测下一个 进程请求 IO 是要干什么的。

CFQ算法的全写为Completely Fair Queuing（完全公平队列）。该算法的特点是按照IO请求的地址进行排序，而不是按照先来后到的顺序来进行响应。   
   在传统的SAS盘上，磁盘寻道花去了绝大多数的IO响应时间。CFQ的出发点是对IO地址进行排序，以尽量少的磁盘旋转次数来满足尽可能多的IO请求。在CFQ算法下，SAS盘的吞吐量大大提高了。但是相比于NOOP的缺点是，先来的IO请求并不一定能被满足，可能会出现饿死的情况。

查看和修改IO调度器的算法非常简单。假设我们要对sda进行操作，如下所示：   
cat /sys/block/sda/queue/scheduler   
echo “cfq” > /sys/block/sda/queue/scheduler

在传统的SAS盘上，CFQ、DEADLINE、ANTICIPATORY都是不错的选择；对于专属的数据库服务器，DEADLINE的吞吐量和响应时间都表现良好。然而在新兴的固态硬盘比如SSD、Fusion IO上，最简单的NOOP反而可能是最好的算法，因为其他三个算法的优化是基于缩短寻道时间的，而固态硬盘没有所谓的寻道时间且IO响应时间非常短。

小结一下linux 2.6内核的四种IO调度算法

http://jackyrong.iteye.com/blog/898938

[innodb\_flush\_method][ innodb\_flush\_log\_at\_trx\_commit]

innodb\_flush\_log\_at\_trx\_commit参数确定日志文件**何时**write、flush。innodb\_flush\_method则确定日志及数据文件**如何**write、flush。

innodb\_flush\_log\_at\_trx\_commit参数对于InnoDB存储引擎写入操作的性能表现有重大影响。在MySQL默认的配 置中，此参数被设置为1，这是最安全的选项，但也是性能最差的选项。有很多同学抱怨MySQL每秒钟才能写入几十条数据，这就很可能是这个默认选项引起 的。

innodb\_flush\_log\_at\_trx\_commit参数的意思是：是否在transaction提交时对日志文件进行flush操作。有三种选择：

0，在transaction提交时不会把log buffer的数据写入到日志文件，也不对日志文件进行flush操作。这是很不安全的，当MySQL程序或者操作系统崩溃后，最后一秒钟的交易数据就会丢失。

1，在transaction提交时把log buffer的数据写入到日志文件，并对日志文件进行flush操作。只要磁盘不损坏，即使MySQL程序或者操作系统崩溃，都不会丢失任何交易数据。

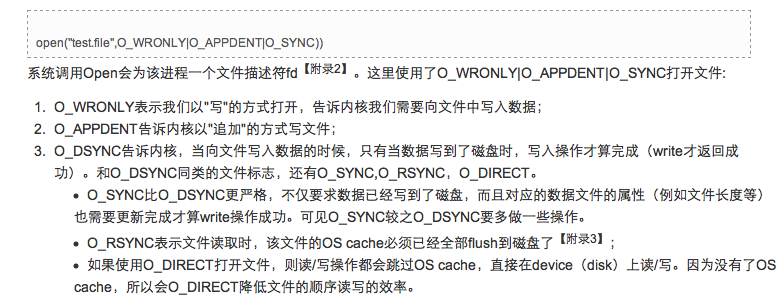
2，在transaction提交时，只是把log buffer的数据写入到日志文件，但并不对日志文件进行flush操作。在这种情况下，当MySQL程序崩溃，交易数据并不会丢失，但当操作系统崩溃时，就会丢失最后一秒钟的交易数据。

综上所述：

如果交易数据非常重要（例如银行交易），那么这个参数必须设置为1。

如果是一般的互联网应该（例如社区讨论），那么可以设置为2，可以获得比较好的写入性能。

在生产环境中，一般没有理由将此参数这是为0。



fsync()/sync()主动将pagecache中的dirty page写回磁盘。

**屏蔽文件cache**

可以通过open()的时候加入O\_DIRECT来绕过内核的cache，但是如果加入O\_DIRECT，则read()和write()的buffer大小一定是文件系统逻辑块大小的整数倍。

**innodb\_flush\_method**

参数Innodb\_flush\_method(Linux)可以设定为：Fdatasync、O\_DSYNC、O\_DIRECT。我们看看这个三个参数是如何影响程序MySQL对日志和数据文件的操作：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Open log | Flush log | Open datafile | Flush data |
| Fdatasync |  | **fsync()** |  | **fsync()** |
| O\_DSYNC | **O\_SYNC** |  |  | **fsync()** |
| O\_DIRECT |  | **fsync()** | **O\_DIRECT** | **跳过OS cache** |

fdatasync被认为是安全的，因为在MySQL总会调用fsync来flush数据。使用O\_DSYNC是有些风险的，有些OS会忽略该参数**O\_SYNC**。

我们看到O\_DIRECT和fdatasync和很类似，但是它会使用**O\_DIRECT**来打开数据文件。有数据表明，如果是大量随机写入操作，**O\_DIRECT**会提升效率。但是顺序写入和读取效率都会降低。所以使用O\_DIRECT需要谨慎。

**[以下部分可以跳过，mysql通常会跳过操作系统层的CACHED BUFFERS]**

pdflush dirty内存页面的回写  
太多dirty的页面意味着风险，比如故障时候的内容丢失，  
以及对突发的大量物理内存请求的响应(大量回写会导致糟糕的响应时间)

pdflush设置

|  |  |
| --- | --- |
| 变量(/proc/sys/vm) | 描述 |
| dirty\_background\_ratio  【下限】 | 指脏数据的空间占总内存的百分比，达到这个上限后会唤醒pdflush把这些脏数据刷新到磁盘，在把脏数据输入磁盘之前所有写IO会被阻塞。默认值10%，可以降低到5% |
| dirty\_ratio  【上限】 | 表示当脏数据占用总内存的百分比超过dirty\_ratio的时候，内核会把所有的写操作阻塞掉，**等待**pdflush把这些脏数据刷入到磁盘后才能恢复正常的IO写。要注意的是当这个事件发生时，会阻**塞掉所有写操作**。默认值40% |
| dirty\_writeback\_centisecs  【间隔】 | 该数值以百分之一秒为单位，它描述pdflush线程的运行频率，默认500。 不建议修改 |
| dirty\_expire\_centisecs  【驻留】 | 该数值以百分之一秒为单位，这个值表示page cache中的数据多久之后被标记为脏数据，只有标记为脏的数据在下一个周期到来时pdflush才会刷入到磁盘。默认3000，可以调整位10秒 |

减少预读：/sys/block/sdb/queue/read\_ahead\_kb，预读的大小，默认128，调整为16

这个参数对顺序读非常有用，意思是，一次提前读多少内容，无论实际需要多少。

增大队列：/sys/block/sdb/queue/nr\_requests，默认128，调整为512

会更占用内存，但能更加多的合并读写操作，速度变慢，但能读写更加多的量

cat /proc/meminfo 可以查看[dirty], cached,buffers,[swapCached]

linux IO 内核参数调优 之 原理和参数介绍

<http://backend.blog.163.com/blog/static/2022941262013111781643200/>

因为NUMA默认的内存分配策略是优先在进程所在CPU的本地内存中分配，会导致CPU节点之间内存分配不均衡，当某个CPU节点的内存不足时，会导致swap产生，而不是从远程节点分配内存。这就是所谓的swap insanity现象。  
MySQL采用了线程模式，对于NUMA特性的支持并不好，如果单机只运行一个MySQL实例，我们可以选择关闭NUMA，关闭的方法有三种：1.硬件层，在BIOS中设置关闭；2.OS内核，启动时设置numa=off；3.可以用numactl命令将内存分配策略修改为interleave（交叉），有些硬件可以在BIOS中设置。

相对于SMP(又称uniform. memory architecture),NUMA引入了node的概念，每个物理CPU都被视作一个node，而每个node都有一个local memory（访问速度更快），其他node的内存则为remote；

<http://blog.itpub.net/15480802/viewspace-759650/>

**数据库配置**

指标：

响应时间：平均响应时间，长耗时（100ms定义为长耗时）

吞吐：QPS， 并发量

测速工具：MYSQL Sysbench

参数：

在MySQL中一次刷新的脏页的数量有一个 innodb\_io\_capacity的参数进行控制，

innodb\_io\_capacity越大，一次刷新的脏页的数量也就越大，在SSD的场景下，由于IO能力大大增强，所以 innodb\_io\_capacity需要调高，可以配置到2000以上，但是对于普通机械磁盘，由于其随机IO的IOPS最多也就是300，所以 innodb\_io\_capacity开的过大，反而会造成磁盘IO不均匀。最后还需要再提一点，就是可以在SSD的场景下适当增大redo log的大小

innodb\_io\_capacity：根据IOPS能力设置，使用fuionio可以设置10000以上。

innodb\_write\_io\_threads = 16：用于写脏页的线程数,设置数量为16，提升写的能力。

innodb\_read\_io\_threads = 1：设置预读线程设置为1，因为线性预读的效果并不明显，所以无需设置更大。

innodb\_thread\_concurrency = 20 最高并发数

Linux环境设置为0 ，让系统自己控制即可，早期的版本在Linux环境下没有作用

innodb\_page\_size：如果使用fusionio，4K的性能最好；使用SAS磁盘，设置为8K。如果全表扫描很多，可以设置为16K。比较小的page size，可以提升cache的命中率。

innodb\_adaptive\_checkpoint：如果使用fusionio，设置为3，提高刷新频率到0.1秒；使用SAS磁盘，设置为2，采用estimate方式刷新脏页。

innodb\_flush\_neighbors= 0：针对fusionio或者SSD，因为随机IO足够好，所以关闭此功能。

innodb\_fast\_checksum = 1：开启Fast checksum特性。

MySQL数据库优化实践

http://www.hellodb.net/2011/07/mysql-linux-hardware-tuning.html

## 相关测试数据及参考：

Fusionio性能测试与瓶颈分析

http://backend.blog.163.com/blog/static/2022941262013102811320942/

SSD磨损数据的分析报告

http://www.hellodb.net/2011/04/ssd-media-wear.html

基于SSD的数据库性能优化

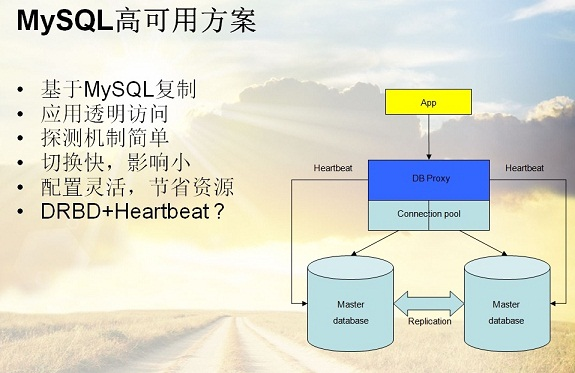
<http://www.hellodb.net/2010/10/ssd-database-2.html>

Linux performance tuning tips for MySQL

<http://www.mysqlperformanceblog.com/2013/12/07/linux-performance-tuning-tips-mysql/>

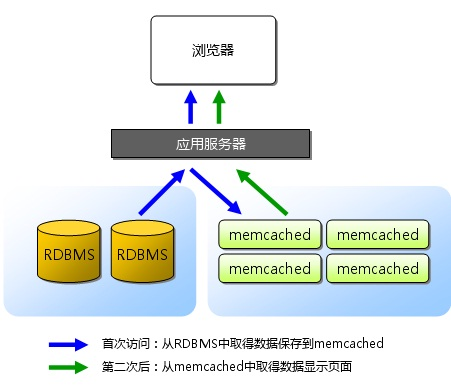
SAS vs SSD各种模式下MySQL TPCC OLTP对比测试结果

<http://outofmemory.cn/wr?u=http%3A%2F%2Fimysql.cn%2F2012%2F09%2F11%2Fsas-vs-ssd-mysql-tpcc-oltp-benchmark.html>



MySQL+Memcache架构思路

由数据库承载数据持久化存储的功能，而Memcached则用来承载 大量的并发访问。通常的做法是：应用的读请求会首先访问Memcached，如果命中则返回，如果没有命中，则会去数据库中读取，并将数据加载到 Memcached中。关于新增，修改和删除操作，一般采用lazy load的策略，即新增时只写入数据库，并不会马上更新Memcached，而是等到再次读取时才会加载到Memcached中，修改和删除操作也是更新 数据库，然后将Memcached中的数据标记为失效，等待下次读取时再加载。



读写分离架构利用了数据库的复制技术，将读和 写分布在不同的处理节点上，从而达到提高可用性和扩展性的目的。最通常的做法是利用MySQL Replication技术，Master DB承担写操作，将数据变化复制到多台Slave DB上，并承担读的操作。这种架构适合read-intensive类型的应用，通过增加Slave DB的数量，读的性能可以线性增长。为了避免Master DB的单点故障，集群一般都会采用两台Master DB做双机热备，所以整个集群的读和写的可用性都非常高。除了MySQL，Oracle从11g开始提供Active Standby的功能，也具备了实现读写分离架构的基础。读写分离架构的缺陷在于，不管是Master还是Slave，每个节点都必须保存完整的数据，如 果在数据量很大的情况下，集群的扩展能力还是受限于单个节点的存储能力，而且对于Write-intensive类型的应用，读写分离架构并不适合。

