**fio详解**

编 写 人：孙自翔

统 稿 人：

编写时间：2013-2-20

部 门 名：

审 核 人：

审核时间：

修订记录

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 编号 | 修订内容简述 | 修订日期 | 版本 | 修订人 |
| 1 | 初稿 | 2013-2-20 | 1.0 | 孙自翔 |
| 2 |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

正文目录

[1 fio介绍 4](#_Toc350180754)

[1.1 fio的作用 4](#_Toc350180755)

[1.2 fio具有的功能及适用范围 4](#_Toc350180756)

[1.3 fio的工作方式 4](#_Toc350180757)

[2 fio使用详解 4](#_Toc350180758)

[2.1 job文件的书写规范 4](#_Toc350180759)

[2.2 job文件举例说明 5](#_Toc350180760)

[2.3 参数类型 5](#_Toc350180761)

[2.4 常用参数 5](#_Toc350180762)

[3 fio常用的基本选项 6](#_Toc350180763)

[4 fio数据统计的方式及常用的磁盘性能参数解释 9](#_Toc350180764)

[4.1 sys/block/<dev>/stat数据文件 9](#_Toc350180765)

[4.2 iostat及fio数据统计 10](#_Toc350180766)

[4.3 fio输出结果详解 11](#_Toc350180767)

[4.4 磁盘性能参数 12](#_Toc350180768)

[4.4.1 磁盘体系结构 12](#_Toc350180769)

[4.4.2 磁盘的IO单位及blocksize 13](#_Toc350180770)

[4.4.3 磁盘的IOPS及传输带宽 13](#_Toc350180771)

[4.4.4 磁盘的IO延迟和队列深度 14](#_Toc350180772)

[5 fio启动流程及线程间同步方式 14](#_Toc350180773)

[5.1 源码版本说明 14](#_Toc350180774)

[5.2 fio重要数据结构 15](#_Toc350180775)

[5.2.1 struct io\_u 15](#_Toc350180776)

[5.2.2 struct ioengine\_ops 15](#_Toc350180777)

[5.2.3 struct thread\_data 16](#_Toc350180778)

[5.3 fio启动流程 16](#_Toc350180779)

[5.3.1 fio启动流程 16](#_Toc350180780)

[5.3.2 fio线程/进程运行流程 16](#_Toc350180781)

[5.4 线程/进程间同步方式 17](#_Toc350180782)

[6 IO引擎发送并接受数据的流程 17](#_Toc350180783)

[6.1 随机发送数据块的方式 17](#_Toc350180784)

[6.1.1 数据结构及标记方式 17](#_Toc350180785)

[6.1.2 获取随机数据块(struct io\_u)流程 18](#_Toc350180786)

[6.2 skeleton\_external.c文件及IO请求处理的流程 18](#_Toc350180787)

[6.3 fio限制流量的方式 19](#_Toc350180788)

[7 总结 19](#_Toc350180789)

[7.1 fio的其他功能 19](#_Toc350180790)

[7.2 遗留待跟踪问题 20](#_Toc350180791)

# fio介绍

## fio的作用

fio主要用来测试磁盘I/O读写性能；根据测试用例，模拟给定的io负载，而不用重复写特定的测试程序。fio可以产生多个线程或进程，每一个线程或进程都用自己的方式产生io模拟测试用例，以保证足够的测试灵活性。

## fio具有的功能及适用范围

fio支持13中不同的I/O 引擎(sync, mmap, libaio, posixaio, SG v3, splice, null, network, syslet, guasi, solarisaio等等)，而且可以根据fio给定的框架生成动态链接库来加入自己的I/O 引擎。fio支持I/O优先级、I/O速率控制、多线程或进程并发、io回放、绘图等等。fio可以很灵活地对cpu、内存、进程/线程、文件或IO特性进行配置，可以工作于块设备或文件系统上，并支持多种操作系统包括Linux, FreeBSD, NetBSD, OS X, OpenSolaris, AIX, HP-UX,和Windows，本文档专注于linux系统。

## fio的工作方式

fio通过job文件来描述要仿真的io负载。一个job文件可以控制产生任意数目的线程和文件。典型的job文件有一个global段（定义共享参数），一个或多个job（描述具体要产生的job的参数）。

运行时，fio从job文件读取这些参数并处理，根据这些参数描述，启动仿真线程/进程。事实上job文件中的每个参数都等价于一个长选项命令行。比如job文件的中的参数iodepth=2等价于--iodepth 2 或者--iodepth=2

有了job文件，fio的运行方式如下

$ fio job\_file

运行fio不需要超级用户，除非所访问的文件或设备需要用到或者某些选项如锁定内存，修改io调度器或者降低io优先级需要用到

# fio使用详解

## job文件的书写规范

* job文件是典型的ini文件，方括号[]定义的一节定义一个job名字
* [global]分节有着特殊含义，这个分节仅仅定义其他分节的默认参数值，而不作为一个单独的任务来执行，其他分节使用(并且可以覆盖)这个分节定义的默认参数值。
* 以‘;’，’#’开头的都是job文件的注释

## job文件举例说明

* 下面的这个job文件代表的含义为

$ fio --name=global --rw=randread --size=128m --name=job1 --name=job2

; -- start job file --

[global]

rw=randread

size=128m

[job1]

[job2]

; -- end job file --

这个job文件的分节都共享着[global]分节的参数，而且没有指定filename=参数，fio为每个job构造合适的文件名.

* 这个job文件代表的含义为：

$ fio --name=random-writers --ioengine=libaio --iodepth=4 --rw=randwrite --bs=32k --direct=0 --size=64m --numjobs=4

; -- start job file --

[random-writers]

ioengine=libaio

iodepth=4

rw=randwrite

bs=32k

direct=0

size=64m

numjobs=4

; -- end job file --

这个job文件没有global分节、使用异步IO、深度为4,、块大小为32k。nmjobs=4表示程序运行时会复制4四个参数完全一样的job，即四个进程同时工作。

## 参数类型

参数的类型有如下几种：

str 字符串。

time 带时间后缀的整型，默认是秒。后缀s/m/h分别代表秒/分/时。

int 整型.后缀表示数字的基。k/m/g/t/p分别代表 kilo/mega/giga/ tera/peta. 后缀大小写不敏感.后缀还可以添加字母'b' (例如 'kb'等价于'k')，kib/mib/gib/tib/pib是以1000为基数的。

如果选项接受区间值，用':' 或'-‘分开。还可以使用前缀，0x 表示16进制。

bool 布尔值.通常解析成整型，但只有1和0两个值

irange 带后缀的区间值。比如1024-4096、1k:4k。如果选项允许两个区间以',' 或 '/'分开，如1k-4k/8k-32k

float\_list 浮点数列表以':' 分开。

## 常用参数

一个典型的job文件包含如下几个基本参数

1. IO type 定义发送文件的模式，如顺序读文件，随机写文件，顺序或随机混合读写文件
2. [Block size](#_磁盘的IO单位及blocksize) 以多大的块来分发io，可以是一个单独的值，或者是一个区间值。
3. IO size 将要读取/写入的数据大小
4. [IO engine](#_struct_ioengine_ops) 定义如何分发io，可以对文件内存映射读写，规则的read/write读写，异步io（async io）读写。
5. [IO depth](#_磁盘的IO延迟和队列深度) 如果IO engine 是异步的，定义要维持的[队列深度](#_磁盘的IO延迟和队列深度)。
6. IO type 定义进行buffered io还是direct/raw io?
7. Num files 工作负载分布在多少个文件上?
8. Num threads 开辟多少个线程/进程来工作。

# fio常用的基本选项

name=str

1. 作用

表示job的ASCII名字

1. 说明

在job文件中可以覆盖fio给的名字；在命令行中表示一个新的job的开始

readwrite=str，rw=str

1. 作用

表示io读写的模式，包括如下几种

read 顺序读

write 顺序写

randwrite 随机写

randread 随机读

rw,readwrite 顺序混合读写

randrw 随机混合读写

1. 说明

对于混合读写缺省值是以50/50分割.字符串的后面可以跟数字':<nr>'

比如对于顺序写，rw=write:4k将生成一个有4k空洞的文件(xxd查看).如果是

[blocksize](#_磁盘的IO单位及blocksize)=int

bs=int

1. 作用

fio所发送的块大小,默认为4k。

1. 说明

fio发送的[块大小](#_磁盘的IO单位及blocksize)是应用层上的块大小，比如bs=4k,8k表示4k的读块大小，8k的写块大小。bs=,8k表示读块大小使用默认大小，写块大小为8k。

nrfiles=int

1. 作用

fio所使用的文件的个数。

1. 说明

比如如果是写文件，制定这个参数 nrfiles=num，将生成num个文件。

[ioengine](#_struct_ioengine_ops) =str

1. 作用

fio提交IO所使用的引擎。

1. 说明

常用的有:

sync,使用read/write/lseek系统调用。

psync,使用pread/pwrite系统调用

vsync,使用 readv/writev系统调用。

libaio,liunx native异步IO,必须设置direct=1 or buffered=0

posixaio glibc posix异步io.

mmap,文件内存映射IO

null，不传输数据，只是用来测试fio本身。

runtime=time

1. 作用

fio所运行的时间,

1. 说明

指定每个job所运行时间，很难限制总时间。

numjobs=int

1. 作用

创建一个job的多个副本,

1. 说明

可以指定线程/进程,每个线程单独打印统计数据，如果想作为一个整体统计数据，需要使用group\_reporting选项。

group\_reporting

1. 作用

打印一组job的总的统计信息,而不是对每个单独但多job进行统计，参看‘numjobs’想想.

1. 说明

可以指定线程/进程,每个线程单独打印统计数据，如果想作为一个整体统计数据，需要使用group\_reporting选项。

rate=int

1. 作用

[限制fio所在job使用的带宽](#_fio限制流量的方式),

1. 说明

和rate\_iops=int选项互斥

rate\_iops=int

1. 作用

[限制fio所在job使用的IOPS](#_fio限制流量的方式),

1. 说明

和rate =int选项互斥,这里的[IOPS](#_磁盘的IOPS及传输带宽)是应用层上的。

ratemin=int

1. 作用

只是fio维持在这个带宽之上，如果不满足fio就退出。,

1. 说明

无

rate\_iops\_min=int

1. 作用

只是fio维持在这个iops之上，如果不满足fio就退出。,

1. 说明

无

[iodepth=int](#_磁盘的IO延迟和队列深度)

1. 作用

[io队列深度](#_磁盘的IO延迟和队列深度)。

1. 说明

默认为1。一般只对异步IO有作用。

direct=bool

1. 作用

添加读写标志O\_DIRECT，此时是非缓冲的。

1. 说明

和buffered=bool对立的。

buffered=bool

1. 作用

使用缓冲IO。

1. 说明

和direct=bool对立的。

offset=int

1. 作用

给定文件的偏移量，偏移量之前的将被忽略.

1. 说明

这样fio所操作的实际大小为real\_size – offset.

directory=str

1. 作用

目录的前缀名，可以替换当前目录

1. 说明

使用这个选项，可以单独mkdri一个目录，作为fio的工作目录，和job文件分开，比较清晰。

size=int

1. 作用

fio所操作的总文件io大小,

1. 说明

这个参数是必须的,除非runtime等参数的限制，fio将传输size大小的byte数，size可以使用百分比，size=20%表示读取文件或磁盘的20%

filesize=int

1. 作用

指定每个文件的大小

1. 说明

当有nrfiles=num参数的时候，会生成num个文件，每个文件大小为filesize指定大小.

blockalign=int

ba=int

1. 作用

[随机IO](#_获取随机数据块(struct_io_u)流程)偏移量对齐的边界大小。

1. 说明

，默认是和'blocksize'指定最小块大小相同，使用direct IO最小对齐的典型大小为512b。

bsrange=irange

1. 作用

指定blocksize大小的范围

1. 说明

发送的块大小总是最小值的倍数

bssplit=str

1. 作用

指定每个blocksize的百分比

1. 说明

形如bssplit=blocksize/percentage:blocksize/percentage.，可以有缺省值，但百分比加起来总和必须为一百,不然会报错。

thread

1. 作用

fio默认fork进程来运行job，使用这个选项讲调用pthread\_create创建线程。

1. 说明

无.

# fio数据统计的方式及常用的磁盘性能参数解释

由于fio的数据统计方式和iostat类似，针对数据统计方式，下面不区分iostat和fio。

## /sys/block/<dev>/stat数据文件

磁盘的stat文件详细记录着块设备<dev>的统计数据,这些数据都是针对块层(Block layer)而不是[应用上层](#_磁盘的IO单位及blocksize)。这些数据同样存在于文件/proc/diskstats(iostat以这个文件统计数据)中，和/sys/block/<dev>/stat中的数据不完全一致，但是增量是相同的.

stat文件包含以空格分开的11个分量，这些域的相信描述如下

Name units description

---- ----- -----------

read I/Os requests number of read I/Os processed

read merges requests number of read I/Os merged with in-queue I/O

read sectors sectors number of sectors read

read ticks milliseconds total wait time for read requests

write I/Os requests number of write I/Os processed

write merges requests number of write I/Os merged with in-queue I/O

write sectors sectors number of sectors written

write ticks milliseconds total wait time for write requests

in\_flight requests number of I/Os currently in flight

io\_ticks milliseconds total time this block device has been active

time\_in\_queue milliseconds total wait time for all requests

其中

1. read I/Os, write I/Os

当I/O请求完成时递增这些值.注意这些请求的是merge之后的请求。

* iostat的输出r/s、w/s(merge之后，块设备每秒钟完成的读写请求的次数),是根据这个来计算的。而且计算r\_await是也要用到。

r\_s=Δ(read\_ios)/interval

w\_s=Δ(write\_ios)/interval

其中interval为采样时间间隔(单位毫秒 ),下同.

1. read merges, write merges

当I/O请求被调度算法合并时，递增这些值.

* iostat的输出rrqm/s、wrqm/s（排队到块设备读/写请求被merge的次数） 是根据这个来计算的。

rrqm\_s=Δ(read\_merges)/ interval

wrqm\_s=Δ(write\_merges)/ interval

1. read sectors, write sectors

这些值记录从块设备读取或向块设备写入的扇区个数,[扇区大小](#_磁盘的IO单位及blocksize)为标准的UNIX 512-byte,当I/O完成时，递增这些值.

* iostat的输出 rsec/s、wsec/s(设备每秒钟读写的扇区个数) 根据这个来计算.

rsec =Δ(read\_sectors)/interval

wsec=Δ(write sectors)/interval

* iostat的输出avgrq-sz(发送给设备的请求的平均大小，单位扇区)

avgrq-sz = (Δ(read\_sectors)+ Δ(write sectors))/( Δ(read\_ios)+ Δ(write\_ios))

1. read ticks, write ticks

这些值记录记录I/O请求在块设备上等待的毫秒数.如果60个读请求的平均等待时间为30ms，这个域将增加60\*30 = 1800.

iostat的输出r\_await、w\_await、await根据这个域来计算，计算公式为

r\_await=Δ(read\_ticks) /Δ(read\_ios)

w\_await=Δ(write\_ticks) /Δ(write\_ios)

await=Δ(read\_ticks+ write\_ticks) /Δ(read\_ios+ write\_ios)

从公式可以看出await不是r\_await和w\_await的简单叠加。

注：r\_await/w\_await含义为: 读/写请求分发到设备被服务的平均时间，包括读/写请求的排队时间和服务时间。

1. in\_flight

这个值表示已经发送到磁盘驱动器但是尚未完成的I/O请求的个数。不包含在排队尚未发送到磁盘驱动器的I/O请求

1. io\_ticks

这个值表示I/O 请求在块设备中排队的毫秒数。

iostat的输出util（磁盘利用率）根据这个域来计算

uitil=Δ(io\_ticks)\* /interval\*100

或者：

uitil=Δ(io\_ticks)\*HZ/interval

其中HZ为系统每秒的始终滴答数(clock ticks per second.)

interval为采样时间间隔，jiffies格式(time in jiffies)

1. time\_in\_queue

表示I/O请求在块设备中等待的毫秒数

iostat的输出avgqu-sz(磁盘请求的[平均队列长度](#_磁盘的IO延迟和队列深度))根据这个域来计算

avgqu-sz =Δ(time\_in\_queue)\*HZ/interval/1000.0

interval为采样时间间隔，jiffies格式(time in jiffies)

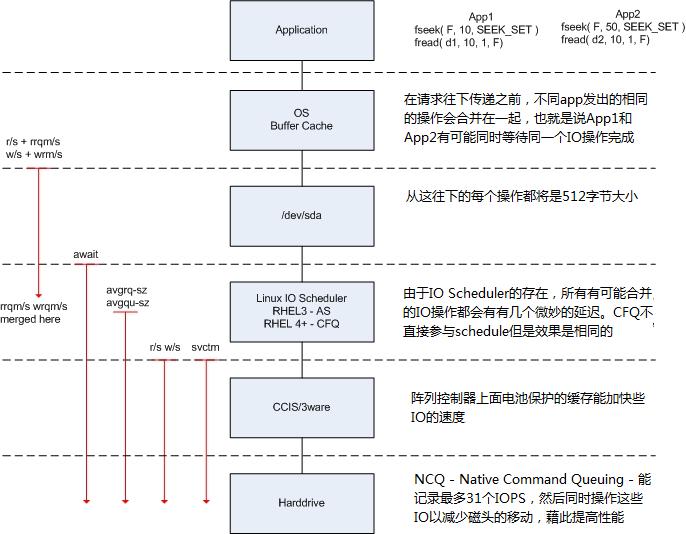
HZ为系统每秒的始终滴答数(clock ticks per second.)

## iostat及fio数据统计

fio专门起一个线程(create\_disk\_util\_thread())来处理来进行数据统计

fio的磁盘统计信息统计的是[stat数据文件](#_/sys/block/<dev>/stat数据文件（附加参考文献）)，fio的磁盘统计信息不太详细，在[5.1节](#_/sys/block/<dev>/stat数据文件（附加参考文献）)以iostat的输出为例，已经详细介绍了其计算方法.除磁盘统计信息之外的其他信息，都是在应用层上进行统计.

磁盘统计信息统计的时间点可以参考下图来理解：



## fio输出结果详解

* fio在一个运行时生命周期中，会打印不同的状态，类似

Threads: 1: [\_r] [24.8% done] [ 13509/ 8334 kb/s] [eta 00h:01m:31s]

Jobs: 1 (f=1), CR=2000/0 IOPS: [W] [60.6% done] [0K/16016K /s] [0 /1001 iops] [eta 00m:13s]

方括号中的字符指示每个线程的当前状态，在一个生命周期中可能的值为:

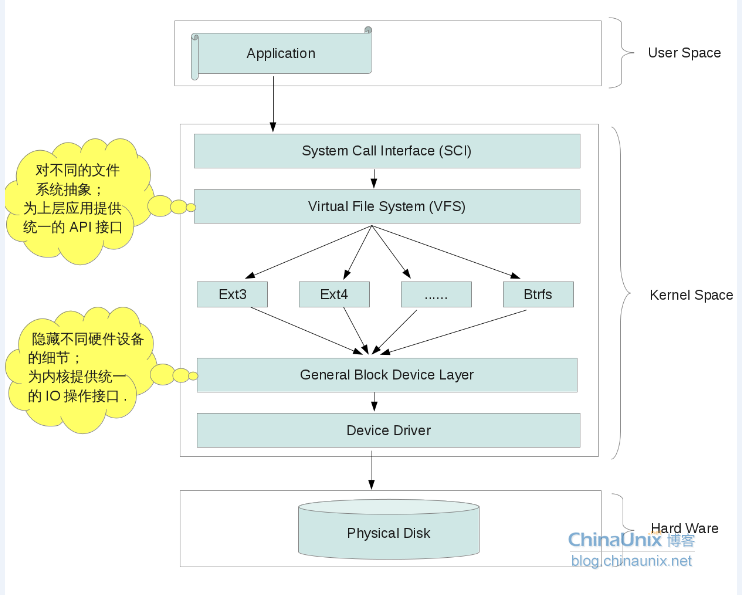
|  |  |  |
| --- | --- | --- |
| 线程状态 | | 含义 |
| Idle | run |
| P |  | 线程准备阶段，还未开始 |
| C |  | 线程被创建,但是还未启动（未进入线程） |
| I |  | 线程初始化(已经进入线程)，等待生成必要的数据(初始化日志文件，调整进程优先级，选择IO调度器等) |
|  | p | 线程运行、预读文件 |
|  | R | 运行中、顺序读 |
|  | r | 运行中、随机读 |
|  | W | 运行中、顺序写 |
|  | w | 运行中、随机写 |
|  | M | 运行中，混合顺序读写 |
|  | m | 运行中混合随机读写 |
|  | F | 运行中，等待同步 fsync() |
|  | V | 运行中，写数据验证. |
| E |  | 线程退出，但是还未被主线收割(reap) |
| \_ |  | 线程被收割 |
| X |  | 线程被收割，有错误的退出 |
| K |  | 线程被收割，由于信号（SIGINT、SIGTERM）被退出 |

* fio完成之后(或者可能被ctrl-c打断),每个线程，线程组及磁盘信息会显示出来，下表列出一些指标及含义。

|  |  |  |
| --- | --- | --- |
| 指标 | 含义 | 备注 |
| io | 总数据量 |  |
| bw | 平均带宽 |  |
| iops | 每秒钟IO次数 | 应用层上 |
| runt | 线程运行时间 |  |
| slat | 提交时延(Submission latency) | 只对异步IO有意义，同步IO的提交时延和完成时延一样 |
| clat | 完成时延(Completion latency) |  |
| IO depths | 队列深度的分布 | 都是2的倍数。16=表示深度在16-31之间 |
| IO submit | 在一次提交当中所提交的块的个数 |  |
| IO complete | 一次完成所返回的数据块的个数 |  |
| IO issued | 读写请求所发送的总的次数 |  |
| IO latencies | io请求完成时间的分布 | 20=12.8%表示12.8的IO请求花费大于10msecs但是小于20msecs. |

## 磁盘性能参数

### 磁盘体系结构



### 磁盘的IO单位及blocksize

磁盘读写的时候是以扇区为最小寻址单位的，也就是说不可能往某某扇区的前半部分写入某某数据.一个扇区的大小通常是512B，每次磁头连续读写的时候，只能以扇区为单位，即使一次只写入了一个字节的数据，那么下一次就不能再向这个扇区剩余的部分接着写入，而是要寻找一个空扇区来写.

“对于磁盘来说”，一次磁头的连续读或者写叫做一次IO。

IO的概念，可以理解成“输入输出”，在系统的上层到下层，存在很多接口，各个接口之间的“数据交互”都可以称为是一次IO，这是广义的IO。

* 卷管理程序对磁盘控制器驱动程序API所做的IO,一次这种IO可能要产生针对磁盘的N个IO，也就是说上层IO是稀疏的，简单的，越往下层越密集、复杂。
* 凌驾于卷管理之上的文件系统对卷的IO，就更加稀疏。
* 上层应用对文件系统API的IO更加简单，比如open()某个文件，seek()到某个位置,write()一段数据，close()这个文件，就是一次IO。而这一次IO可能对应文件系统到卷的M个IO，对应卷到控制器驱动的NxM个IO,对应最终磁盘的NxMxP个IO。
* 反之，当上层应用发送的IO块大小比较小时，根据磁盘调度策略,N个上层应用发送的IO可能合并为一个IO。

所以上层应用的IO块大小和磁盘每次实际读写的块大小不是一一对应的。fio的blocksize=int参数的大小对应于应用层上的blockszie,磁盘可能需要[N次IO](#_iostat及fio数据统计)写完这块数据。

### 磁盘的IOPS及传输带宽

1. IOPS

磁盘的IOPS，也就是每秒能进行多少次IO，每次IO根据写入数据的大小，这个值也不是固定的，而且IOPS随着上层应用的不同有很大变化。

* 如果在不频繁换道的情况下，每次IO都写入很大的一块连续数据，则此时每秒所做的IO次数时比较低的
* 如果磁头频繁换道，每次写入数据还比较大的话，此时的IOPS应该是这块硬盘的最低数值
* 如果在不频繁换道的条件下每次写入最小的数据块，比如512B，那么此时的iops将是最高值
* 如果IO的有效载荷长度为0，不包含开销，此时的iops是理论上的最大极限值.

例如，写入100k个大小为1k的文件到硬盘上，耗费的时间要比写入1个100M大小的文件多得多(可以编写fio的job文件进行验证，用到nrfiles参数).

* 写入100k个1k文件时，根据文件分布情况和大小情况，可能需要好几万甚至十几万次IO才能完成，此时每秒需要的IO非常高，如果用具有较高iops的磁盘将会提速不少。
* 而写入一个100M的大文件，如果这个文件在磁盘上是连续存放的，那么只需要几百个IO就可以完成，此时就算用了较高iops的硬盘来做，也不会有提升,只有换用具有较大传输带宽的硬盘，才能体现其优势。

同一块磁盘在读写小块数据的时的IOPS较高，读写大块数据时候IOPS较低，因为读写大块数据花费的时间较长.

1. 传输带宽

传输带宽指的是硬盘或设备在传输数据的时候数据流的速度。

* 如果写入100k个1KB的文件需要10s，那么此时的传输带宽只能达到10MB/s,
* 如果写入一个100M的文件用了0.5s,那么此时的传输带宽可以达到200MB/s

可以看出，具有高带宽规格的硬盘在传输大块连续数据时具有优势，而具有高IOPS的硬盘在传输小块不连续数据时具有优势.

### 磁盘的IO延迟和队列深度

IO延迟是指控制器将IO指令发出之后，直到IO完成的过程中总共花费的时间。早前业界不成文的规定为，只要IO延迟在20ms内，IO性能对于应用程序来说都是可以接受的，但是如果大于20ms，应用程序的性能将会受到较大影响。

当接收到的IO很少的时候，IO延迟也会很小。比如1ms甚至小于1ms，。此时每个IO通道的IOPS=1000ms/1ms=1000.

控制器向存储设备发起的指令不是一条一条顺序发送的，而是一批一批的发送，存储目标设备批量执行IO，然后将数据和结果返回控制器。此时，只要存储设备度量和消化能力足够，在IO比较少时，处理一条指令和同时处理多条指令将会消耗几乎相同的时间。控制器所发出的批量指令的最大条数，由控制器上的Queue Depth（/sys/block/sda/queue/nr\_requests）决定

给出Queue Depth、IOPS、IO延迟三者中的任意两个，可以推算出第三者

IOPS=(Queue Depth)/(IO Latency)

实际上，

* 随着Queue Depth的增加，IO延迟也会增加。所以，随着IO数目的增多，将很快达到存储设备提供的最大IOPS处理能力，此时IO延迟将会陡峭升高，而IOPS则增加缓慢.
* 好的存储系统内部应该具有快速IO消化能力。对于消化不良的存储设备，其IO延迟将升高的很快，以至于在IOPS较低时，IO延迟已经达到了20ms的可接受值。
* 消化能力再高也有饱和的时候，下图为IO延迟和Queue Depth的示意图。



# fio启动流程及线程间同步方式

## 源码版本说明

高版本和低版本相比较

* 高版本有更多的参数，2.X版本增加了客户服务器模式.
* 高版本有更细致的代码控制，更加完善
* 有些功能用线程来代替信号(异步改为同步)，比如数据统计、更新全局时间struct timeval \*fio\_tv.
* 其主流程是相似的,所以数据结构的描述和fio运行的流程高版本和低版本都涉及。

## fio重要数据结构

### struct io\_u

struct io\_u {

union {

#ifdef FIO\_HAVE\_LIBAIO

struct iocb iocb;

#endif

#ifdef FIO\_HAVE\_POSIXAIO

struct aiocb aiocb;

#endif

};

struct timeval start\_time;

struct timeval issue\_time;

void \*buf;

unsigned int buflen;

unsigned long long offset;

unsigned int resid;

unsigned int error;

enum fio\_ddir ddir;

struct fio\_file \*file;

struct list\_head list;

};

这个结构主要存放着待处理(发送/接受)的数据块的信息，每个[struct thread\_data](#_struct_thread_data)中存放着struct io\_u的相关链表(正在使用的和空闲的io\_u链表，)

数据块缓冲区、长度、偏移量，IO方向（读写等），数据块所在的文件等.

### struct ioengine\_ops

结构体的定义大致如下

struct ioengine\_ops {

……

int (\*init)(struct thread\_data \*);

int (\*prep)(struct thread\_data \*, struct io\_u \*);

int (\*queue)(struct thread\_data \*, struct io\_u \*);

int (\*getevents)(struct thread\_data \*, int, int, struct timespec \*);

struct io\_u \*(\*event)(struct thread\_data \*, int);

int (\*cancel)(struct thread\_data \*, struct io\_u \*);

void (\*cleanup)(struct thread\_data \*);

……..

};

结构体中主要存放着一些函数指针，每个ioengine都会实现这些函数指针的全部或一部分，这些函数指针主要是用来准备及发送、接受数据块的，发送数据块的详细细节参看[6.2节](#_随机发送数据块的方式)，ioengine在程序运行之前加载。每个[struct thread\_data](#_struct_thread_data)中存放着一个struct ioengine\_ops结构的指针

### struct thread\_data

struct thread\_data {

struct thread\_options o;

pthread\_t thread;

int thread\_number;

struct thread\_stat ts;

struct io\_log \*slat\_log;

………

struct ioengine\_ops \*io\_ops;

………

struct fio\_mutex \*mutex;

}

这个数据结构是线程/进程[启动函数的最重要的参数](#_fio线程/进程运行流程)，每个线程/进程独占一个，并且数据是放在共享内存中的。

这些结构存放了线程运行、统计信息等的所有信息包括

* 解析的命令行选项和job文件参数、线程号、进程号、各种随机种子
* 线程的各种统计日志的信息
* fio所打开的文件信息，使用的ioengine，各种链表和互斥量、条件变量

## fio启动流程

### fio启动流程

* 在程序运行之前首先加载ioengine
* 解析命令行参数和job文件参数
* 启动一个线程用来统计磁盘利用率、实时打印运行状态。
* 做一些准备工作，如准备读写log文件(如果有),初始化互斥锁，设置起始时间(统计数据用)等
* 循环启动各个线程/进程，详细流程[参考下节](#_fio线程/进程运行流程)
* 显示统计数据,做一些数据清理工作，停止统计线程，释放互斥锁等

### fio线程/进程运行流程

#### 在循环启动每个线程/进程时

* 做一些准备工作，如启动一个线程专门用来更新当前时间,安装信号(SIGINT, SIGTERM, SIGBREAK, SIGUSR1)
* 根据参数numjobs来决定需要创建多少job的副本（启动多个线程/进程），
* 根据参数判断应该是启动一个线程还是fork一个进程
* 如果是启动一个线程则直接调用pthread\_create来启动线程，启动函数为[thread\_main](#_线程/进程调用函数*thread_main(void_*data)的流)，参数为struct thread\_data；如果是fork一个进程，则直接调用函数[thread\_main](#_线程/进程调用函数*thread_main(void_*data)的流)，参数也是struct thread\_data. struct thread\_data是存放在共享内存中的，线程/进程间的互斥参考[4.4节](#_线程/进程间同步方式)

#### 线程/进程调用函数\*thread\_main(void \*data)的流程如下：

* 做一些准备工作，设置struct thread\_data结构中的pid(进程或线程号)、初始化条件变量等
* 运行exec\_prerun参数所指定的命令, 根据参数pre\_read决定是否预读.
* 处理IO请求，流程参看[6.1.2节](#_发送及接受数据的流程)

## 线程/进程间同步方式

fio的线程/进程间同步比较直观,主要靠下面两点来实现

* 共享内存.线程函数的参数和进程运行函数的参数都是struct thread\_data，每个线程/进程都从共享内存中获取参数，而且每个线程/进程使用各自的参数。
* 通过函数pthread\_mutexattr\_setpshared(&attr, PTHREAD\_PROCESS\_SHARED)设置互斥锁变量的作用于是在进程间共享的。这样，线程/进程在访问共享内存时的互斥方式就统一起来了。
* 每个struct thread\_data结构体重都有一个互斥量struct fio\_mutex \*mutex。其定义如下

struct fio\_mutex {

pthread\_mutex\_t lock;

pthread\_cond\_t cond;

int value;

int waiters;

};

互斥量保证每个线程/进程都有各自的线程参数，条件变量用来和父进程通信。

# IO引擎发送并接受数据的流程

## 随机发送数据块的方式

### 数据结构及标记方式

struct fio\_file结构体中的unsigned long \*file\_map用来存放位图，数组的长度为num\_maps.

struct fio\_file {

……

unsigned long \*file\_map;

unsigned int num\_maps;

……

};

* file\_map数组中的每个bit位表示一个块,不考虑是否为大端或小端字节.

根据文件实际文件长度(文件总长度减去文件的偏移量)和最小块大小，可以得到文件所包含的最小块的个数。

file\_map中每个元素代表一个map,一个map可以存储64个块

#define BLOCKS\_PER\_MAP (8 \* sizeof(unsigned long))

* 根据所访问的块在文件长度中的位置，可以得到块所在的map(在file\_map中的哪个元素)

#define TO\_MAP\_BLOCK(f, b) (b)

#define RAND\_MAP\_IDX(f, b) (TO\_MAP\_BLOCK(f, b) / BLOCKS\_PER\_MAP)

其中b=(io\_u->offset - f->file\_offset) / (unsigned long long) min\_bs;

offset是要访问块的起点的偏移量, file\_offset为所读文件的偏移量

如idx = RAND\_MAP\_IDX(f, block);

* 进一步可以求出这个块所在map的位置

#define RAND\_MAP\_BIT(f, b) (TO\_MAP\_BLOCK(f, b) & (BLOCKS\_PER\_MAP - 1))

如bit = RAND\_MAP\_BIT(f, block);

其实质是截取块大小bit位的最后6位，取值范围为[0,63]，这样就可以一一映射为file\_map元素中的64个位置

* 标记块元素的方式，把序号为idx的元素的第bit个位置1.

f->file\_map[idx] |= (1UL << bit);

判断块元素是否被标记的方式，判断idx个元素的第bit个位是否为0

(f->file\_map[idx] & (1UL << bit)) == 0

举例如下，假如所访问的块的序号为1125，所访问的文件偏移量为0.

1. idx = RAND\_MAP\_IDX(f, block) => idx=1125/64=17，即这个块在file\_map[17]中
2. bit = RAND\_MAP\_BIT(td, f, block)=>bit=1125&63 = 37,几这个块在file\_map[17]的第37个bit位上
3. 标记这个块元素为1，f->file\_map[17] |= (1UL << 37);

判断这个块元素是否被标记(f->file\_map[17] & (1UL << 37)) == 0

### 获取随机数据块(struct io\_u)流程

1. 初始化数据块信息。根据job参数初始化数据块(struct io\_u)相关信息

填充数据块的相关信息。如果是读操作可以从log中读取，填充ddir，file等

填充buflen.首先获取一个随机数r，

r = os\_random\_long(&td->bsrange\_state);

获取buflen介于(1, max\_bs[ddir])

buflen = (unsigned int) (1 + (double) (td->max\_bs[ddir] - 1) \* r / (RAND\_MAX + 1.0))

最后按最小块对齐(buflen的大小是最小块的倍数)，

buflen = (buflen + td->min\_bs[ddir] - 1) & ~(td->min\_bs[ddir] - 1);

可以看出由buflen可以计算所包含的最小块个数。

1. 寻找空闲块偏移量。不断循环，直到在file\_map随机寻找到一个空闲的块(偏移量)
2. 修正数据块长度。得到数据块长度(最小块个数)和偏移量之后,对要发送的数据从第一个块进行标记，一直标记到非空闲块。如果所标记的数据块长度小于buflen，就把这个长度赋值给buflen.

## skeleton\_external.c文件及IO请求处理的流程

skeleton\_external.c文件是编写外部ioengine的模板，同时也是fio内部的ioengine的模板.其主要定义了每个ioengine在[struct ioengine\_ops结构](#_fio重要数据结构)中的函数指针的具体实现.

fio准备、发送及接受数据的方式仿照异步io(libaio库)的方式进行 ,以libaio为例，这些函数的作用如下：

|  |  |  |
| --- | --- | --- |
| struct ioengine\_op中的函数指针 | 主要对应libaio的函数 | 在libaio中的作用 |
| init | int io\_setup(int maxevents, io\_context\_t \*ctxp); | 创建可以接收最多maxevents个请求的异步I/O环境变量, 即创建一个libaio的工作环境 |
| prep | io\_prep\_pwrite、io\_prep\_pread等一些宏定义 | 填充结构体struct iocb  ，指定读写类型、起始扇区、长度和设备标志符等 |
| queue | io\_submit | 提交IO读写请求（”提交时延”计算这个时间） |
| getevents | io\_getevents | 等待io结束信号,并返回已完成的事件的个数（”完成时延”计算这个时间） |
| event | 无 | 得到指定的数据块struct io\_u指针 |
| cancel | io\_cancel | 取消当前正在进行的由io\_submit 指定的I/O操作 |
| cleanup | int io\_destroy(io\_context\_t ctx); | 销毁环境变量，同时可以取消当前正在进行的I/O |

一个典型的(libaio)发送或接受数据流程基本上是依照上面表格的方式进行的：init->prep->queue->getevets.

## fio限制流量的方式

fio通过睡眠的方式在应用层上进行流量控制

1. fio的两个参数 rate=int及rate\_iops=int是用来进行流量控制的,而且这两个参数是互斥的,只需指定一个.
2. 根据上面两个参数中的一个来计算bps(byte per second),方式如下：

if (td->o.rate[ddir])

td->rate\_bps[ddir] = td->o.rate[ddir];

else

td->rate\_bps[ddir] = td->o.rate\_iops[ddir] \* bs;

1. 根据bps和当前要发送的字节数计算要睡眠的纳秒数，实际处理比这个复杂.

bytes = td->this\_io\_bytes[ddir];

bps = td->rate\_bps[ddir];

secs = bytes / bps;

1. 在每发送一块数据之后都睡眠一段时间

usec\_sleep(td, usec);

# 总结

文中写的还比较粗糙，后面可以做进一步补充及完善.

## fio的其他功能

fio有一些其他功能，文中并未涉及，比如对cpu的测试、cgroup的使用。

## 遗留待跟踪问题

* 参考文献有待补充
* 异步io(libaio、posixaio)的使用方式
* 对同步IO引擎(read/write、pread/pwrite、readv/writev)之间的相互对比
* 同步io引擎和异步io引擎的对比
* O\_DIRECT和非O\_DIRECT的对比.
* sysfs文件系统,磁盘调度算法