Squid 中文权威指南

（第 8 章）

译者序：

本人在工作中维护着数台 Squid 服务器，多次参阅 Duane Wessels（他也是 Squid 的创始人）的这本书，原书名是"Squid: The Definitive Guide"，由 O'Reilly 出版。我在业余时间把它翻译成中文，希望对中文 Squid 用户有所帮助。对普通的单位上网用户，Squid 可充当代理服务器；而对 Sina,NetEase 这样的大型站点，Squid 又充当 WEB 加速器。这两个角色它都扮演得异常优秀。窗外繁星点点，开源的世界亦如这星空般美丽，而 Squid 是其中耀眼的一颗星。

对本译版有任何问题，请跟我联系，我的Email是：yonghua_peng@yahoo.com.cn

彭勇华

目 录

第 8 章 高级磁盘缓存主题

8.1 是否存在磁盘 I/O 瓶颈？

Web 缓存器例如 squid，通常在磁盘 I/O 变成瓶颈时，不会正确的体现和告知你。代替的是，随着负载的增加，响应时间和/或命中率会更低效。当然，响应时间和命中率可能因为其他原因而改变，例如网络延时和客户请求方式的改变。

也许探测 cache 性能瓶颈的最好方式是做压力测试，例如 Web Polygraph。压力测试的前提是你能完全控制环境，消除未知因素。你可以用不同的 cache 配置来重复相同的测试。不幸的是，压力测试通常需要大量的时间，并要求有空闲的系统（也许它们正在使用中）。

假如你有资源执行 squid 压力测试，请以标准的 cache 工作负载开始。当你增加负载时，在某些点上你能看到明显的响应延时和/或命中率下降。一旦你观察到这样的性能降低，就禁止掉磁盘缓存，再测试一次。你可以配置 squid 从来不缓存任何响应（使用 null 存储机制，见 8.7 章）。代替的，你能配置工作负载到 100%不可 cache 响应。假如不使用 cache 时，平均响应时间明显更好，那么可以确认磁盘 I/O 是该水平吞吐量的瓶颈。

假如你没有时间或没有资源来执行 squid 压力测试，那么可检查 squid 的运行时统计来查找磁盘 I/O 瓶颈。cache 管理器的 General Runtime Information 页面（见 14 章）会显示出 cache 命中和 cache 丢失的中值响应时间。

```
Median Service Times (seconds)  5 min     60 min:
        HTTP Requests (All):    0.39928   0.35832
        Cache Misses:           0.42149   0.39928
        Cache Hits:             0.12783   0.11465
        Near Hits:              0.37825   0.39928
        Not-Modified Replies:   0.07825   0.07409
```

对健壮的 squid 缓存来说，命中显然快于丢失。中值命中响应时间典型的少于 0.5 秒或更少。我强烈建议你使用 SNMP 或其他的网络监视工具来从 squid 缓存采集定期测量值。如果平均命中响应时间增加得太明显，意味着系统有磁盘 I/0 瓶颈。

假如你认为产品 cache 面临此类问题，可以用前面提到的同样的技术来验证你的推测。配置 squid 不 cache 任何响应，这样就避开了所有磁盘 I/O。然后仔细观察 cache 丢失响应时间。假如它降下去，那么你的推测该是正确的。

一旦你确认了磁盘吞吐能力是 squid 的性能瓶颈，那么可做许多事来改进它。其中一些方法要求重编译 squid，然而另一些相对较简单，只需调整 Unix 文件系统。

8.2 文件系统调整选项

首先，从来不在 squid 的缓存目录中使用 RAID。以我的经验看，RAID 总是降低 squid 使用的文件系统的性能。最好有许多独立的文件系统，每个文件系统使用单独的磁盘驱动器。

我发现 4 个简单的方法来改进 squid 的 UFS 性能。其中某些特指某种类型的操作系统例如 BSD 和 Linux，也许对你的平台不太合适：

1.某些 UFS 支持一个 noatime 的 mount 选项。使用 noatime 选项来 mount 的文件系统，不会在读取时，更新相应的 i 节点访问时间。使用该选项的最容易的方法是在/etc/fstab 里增加如下行：

```
# Device                Mountpoint      FStype  Options         Dump    Pass#
/dev/ad1s1c             /cache0         ufs     rw,noatime      0       0
```

2.检查 mount(8)的 manpage 里的 async 选项。设置了该选项，特定的 I/O 操作（例如更新目录）会异步执行。某些系统的文档会标明这是个危险的标签。某天你的系统崩溃，你也许会丢失整个文件系统。对许多 squid 安装来说，执行性能的提高值得冒此风险。假如你不介意丢失整个 cache 内容，那么可以使用该选项。假如 cache 数据非常有价值，async 选项也许不适合你。

3.BSD 有一个功能叫做软更新。软更新是 BSD 用于 Journaling 文件系统的代替品。在 FreeBSD 上，你可以在没有 mount 的文件系统中，使用 tunefs 命令来激活该选项：

```
# umount /cache0
# tunefs -n enable /cache0
# mount /cache0
```

4.你对每个文件系统运行一次 tunefs 命令就可以了。在系统重启时，软更新自动在文件系统中激活了。

在 OpenBSD 和 NetBSD 中，可使用 softdep mount 选项：

```
# Device                Mountpoint      FStype  Options         Dump    Pass#
/dev/sd0f               /usr            ffs     rw,softdep      1       2
```

假如你象我一样，你可能想知道在 async 选项和软更新选项之间有何不同。一个重要的区别是，软更新代码被设计成在系统崩溃事件中，保持文件系统的一致性，而 async 选项不是这样的。这也许让你推断 async 执行性能好于软更新。然而，如我在附录 D 中指出的，事实相反。

以前我提到过，UFS 性能特别是写性能，依赖于空闲磁盘的数量。对空文件系统的磁盘写操作，要比满文件系统快得多。这是 UFS 的最小自由空间参数，和空间/时间优化权衡参数背后的理由之一。假如 cache 磁盘满了，squid 执行性能看起来很糟，那么试着减少 cache_dir 的容量值，以便更多的自由空间可用。当然，减少 cache 大小也会降低命中率，但响应时间的改进也许值得这么做。假如你给 squid 缓存配置新的设备，请考虑使用超过你需要的更大磁盘，并且仅仅使用空间的一半。

8.3 可选择的文件系统

某些操作系统支持不同于 UFS（或 ext2fs）的文件系统。Journaling 文件系统是较普遍的选择。在 UFS 和 Journaling 文件系统之间的主要不同在于它们处理更新的方式。在 UFS 下，更新是实时的。例如，当你改变了某个文件并且将它存储到磁盘，新数据就替换了旧数据。当你删除文件时，UFS 直接更新了目录。

Journaling 文件系统与之相反，它将更新写往独立的记帐系统，或日志文件。典型的你能选择是否记录文件改变或元数据改变，或两者兼备。某个后台进程在空闲时刻读取记帐，并且执行实际的改变操作。Journaling 文件系统典型的在系统崩溃后比 UFS 恢复更快。在系

统崩溃后，Journaling 文件系统简单的读取记帐，并且提交所有显著的改变。

Journaling 文件系统的主要弊端在于它们需要额外的磁盘写操作。改变首先写往日志文件，然后才写往实际的文件或目录。这对 web 缓存影响尤其明显，因为首先 web 缓存倾向于更多的磁盘写操作。

Journaling 文件系统对许多操作系统可用。在 Linux 上，你能选择 ext3fs,reiserfs, XFS, 和其他的。XFS 也可用在 SGI/IRIX，它原始是在这里开发的。Solaris 用户能使用 Veritas 文件系统产品。TRU64（以前的 Digital Unix）高级文件系统（advfs）支持 Journaling。

你可以不改变 squid 的任何配置而使用 Journaling 文件系统。简单的创建和挂载在操作系统文档里描述的文件系统，而不必改变 squid.cf 配置文件里的 cache_dir 行。

用类似如下命令在 Linux 中制作 reiserfs 文件系统：

/sbin/mkreiserfs /dev/sda2

对 XFS，使用：

mkfs -t xfs -f /dev/sda2

注意 ext3fs 其实简单的就是激活了记帐的 ext2fs。当创建该文件系统时，对 mke2fs 使用-j 选项：

/sbin/mke2fs -j /dev/sda2

请参考其他操作系统的相关文档。

8.4 aufs 存储机制

aufs 存储机制已经发展到超出了改进 squid 磁盘 I/O 响应时间的最初尝试。"a"代表着异步 I/O。默认的 ufs 和 aufs 之间的唯一区别，在于 I/O 是否被 squid 主进程执行。数据格式都是一样的，所以你能在两者之间轻松选择，而不用丢失任何 cache 数据。

aufs 使用大量线程进行磁盘 I/O 操作。每次 squid 需要读写，打开关闭，或删除 cache 文件时，I/O 请求被分派到这些线程之一。当线程完成了 I/O 后，它给 squid 主进程发送信号，并且返回一个状态码。实际上在 squid2.5 中，某些文件操作默认不是异步执行的。最明显的，磁盘写总是同步执行。你可以修改 src/fs/aufs/store_asyncufs.h 文件，将 ASYNC_WRITE 设为 1，并且重编译 squid。

aufs 代码需要 pthreads 库。这是 POSIX 定义的标准线程接口。尽管许多 Unix 系统支持 pthreads 库，但我经常遇到兼容性问题。aufs 存储系统看起来仅仅在 Linux 和 Solaris 上运行良好。在其他操作系统上，尽管代码能编译，但也许会面临严重的问题。

为了使用 aufs，可以在./configure 时增加一个选项：

% ./configure --enable-storeio=aufs,ufs

严格讲，你不必在 storeio 模块列表中指定 ufs。然而，假如你以后不喜欢 aufs，那么就需要指定 ufs，以便能重新使用稳定的 ufs 存储机制。

假如愿意，你也能使用—with-aio-threads=N 选项。假如你忽略它，squid 基于 aufs cache_dir 的数量，自动计算可使用的线程数量。表 8-1 显示了 1-6 个 cache 目录的默认线程数量。

Table 8-1. Default number of threads for up to six cache directories

cache_dirs	Threads
1	16
2	26

Table 8-1. Default number of threads for up to six cache directories	
cache_dirs	Threads
3	32
4	36
5	40
6	44

将 aufs 支持编译进 squid 后，你能在 squid.conf 文件里的 cache_dir 行后指定它：
cache_dir aufs /cache0 4096 16 256
在激活了 aufs 并启动 squid 后，请确认每件事仍能工作正常。可以运行 tail -f store.log 一会儿，以确认缓存目标被交换到磁盘。也可以运行 tail -f cache.log 并且观察任何新的错误或警告。

8.4.1 aufs 如何工作

Squid 通过调用 pthread_create()来创建大量的线程。所有线程在任何磁盘活动之上创建。这样，即使 squid 空闲，你也能见到所有的线程。
无论何时，squid 想执行某些磁盘 I/O 操作（例如打开文件读），它分配一对数据结构，并将 I/O 请求放进队列中。线程循环读取队列，取得 I/O 请求并执行它们。因为请求队列共享给所有线程，squid 使用独享锁来保证仅仅一个线程能在给定时间内更新队列。
I/O 操作阻塞线程直到它们被完成。然后，将操作状态放进一个完成队列里。作为完整的操作，squid 主进程周期性的检查完成队列。请求磁盘 I/O 的模块被通知操作已完成，并获取结果。
你可能已猜想到，aufs 在多 CPU 系统上优势更明显。唯一的锁操作发生在请求和结果队列。然而，所有其他的函数执行都是独立的。当主进程在一个 CPU 上执行时，其他的 CPU 处理实际的 I/O 系统调用。

8.4.2 aufs 发行

线程的有趣特性是所有线程共享相同的资源，包括内存和文件描述符。例如，某个线程打开一个文件，文件描述符为 27，所有其他线程能以相同的文件描述符来访问该文件。可能你已知道，在初次管理 squid 时，文件描述符短缺是较普遍问题。Unix 内核典型的有两种文件描述符限制：
进程级的限制和系统级的限制。你也许认为每个进程拥有 256 个文件描述符足够了（因为使用线程），然而并非如此。在这样的情况下，所有线程共享少量的文件描述符。请确认增加系统的进程文件描述符限制到 4096 或更高，特别在使用 aufs 时。
调整线程数量有点棘手。在某些情况下，可在 cache.log 里见到如下警告：
2003/09/29 13:42:47| squidaio_queue_request: WARNING - Disk I/O overloading

这意味着 squid 有大量的 I/O 操作请求充满队列，等待着可用的线程。你首先会想到增加线

程数量，然而我建议，你该减少线程数量。
增加线程数量也会增加队列的大小。超过一定数量，它不会改进 aufs 的负载能力。它仅仅意味着更多的操作变成队列。太长的队列导致响应时间变长，这绝不是你想要的。
减少线程数量和队列大小，意味着 squid 检测负载条件更快。当某个 cache_dir 超载，它会从选择算法里移除掉（见 7.4 章）。然后，squid 选择其他的 cache_dir 或简单的不存储响应到磁盘。这可能是较好的解决方法。尽管命中率下降，响应时间却保持相对较低。

8.4.3 监视 aufs 操作

Cache 管理器菜单里的 Async IO Counters 选项，可以显示涉及到 aufs 的统计信息。它显示打开，关闭，读写，stat，和删除接受到的请求的数量。例如：

```
% squidclient mgr:squidaio_counts
...
ASYNC IO Counters:
Operation            # Requests
open                  15318822
close                 15318813
cancel                15318813
write                        0
read                  19237139
stat                         0
unlink                2484325
check_callback      311678364
queue                        0
```

取消(cancel)计数器正常情况下等同于关闭(close)计数器。这是因为 close 函数总是调用 cancel 函数，以确认任何未决的 I/O 操作被忽略。
写(write)计数器为 0，因为该版本的 squid 执行同步写操作，即使是 aufs。
check_callbak 计数器显示 squid 主进程对完成队列检查了多少次。
queue 值显示当前请求队列的长度。正常情况下，队列长度少于线程数量的 5 倍。假如你持续观察到队列长度大于这个值，说明 squid 配得有问题。增加更多的线程也许有帮助，但仅仅在特定范围内。

8.5 diskd 存储机制

diskd（disk 守护进程的短称）类似于 aufs，磁盘 I/O 被外部进程来执行。不同于 aufs 的是，diskd 不使用线程。代替的，它通过消息队列和共享内存来实现内部进程间通信。
消息队列是现代 Unix 操作系统的标准功能。许多年以前在 AT&T 的 Unix System V 的版本 1 上实现了它们。进程间的队列消息以较少的字节传递：32-40 字节。每个 diskd 进程使用一个队列来接受来自 squid 的请求，并使用另一个队列来传回请求。

8.5.1 diskd 如何工作

Squid 对每个 cache_dir 创建一个 diskd 进程。这不同于 aufs，aufs 对所有的 cache_dir 使用一个大的线程池。对每个 I/O 操作，squid 发送消息到相应的 diskd 进程。当该操作完成后，diskd 进程返回一个状态消息给 squid。squid 和 diskd 进程维护队列里的消息的顺序。这样，不必担心 I/O 会无序执行。

对读和写操作，squid 和 diskd 进程使用共享内存区域。两个进程能对同一内存区域进行读和写。例如，当 squid 产生读请求时，它告诉 diskd 进程在内存中何处放置数据。diskd 将内存位置传递给 read()系统调用，并且通过发送队列消息，通知 squid 该过程完成了。然后 squid 从共享内存区域访问最近的可读数据。

diskd 与 aufs 本质上都支持 squid 的无阻塞磁盘 I/O。当 diskd 进程在 I/O 操作上阻塞时，squid 有空去处理其他任务。在 diskd 进程能跟上负载情况下，这点确实工作良好。因为 squid 主进程现在能够去做更多工作，当然它有可能会加大 diskd 的负载。diskd 有两个功能来帮助解决这个问题。

首先，squid 等待 diskd 进程捕获是否队列超出了某种极限。默认值是 64 个排队消息。假如 diskd 进程获取的数值远大于此，squid 会休眠片刻，并等待 diskd 完成一些未决操作。这本质上让 squid 进入阻塞 I/O 模式。它也让更多的 CPU 时间对 diskd 进程可用。通过指定 cache_dir 行的 Q2 参数的值，你可以配置这个极限值：

```
cache_dir diskd /cache0 7000 16 256 Q2=50
```

第二，假如排队操作的数量抵达了另一个极限，squid 会停止要求 diskd 进程打开文件。这里的默认值是 72 个消息。假如 squid 想打开一个磁盘文件读或写，但选中的 cache_dir 有太多的未完成操作，那么打开请求会失败。当打开文件读时，会导致 cache 丢失。当打开文件写时，会阻碍 squid 存储 cache 响应。这两种情况下用户仍能接受到有效响应。唯一实际的影响是 squid 的命中率下降。这个极限用 Q1 参数来配置：

```
cache_dir diskd /cache0 7000 16 256 Q1=60 Q2=50
```

注意在某些版本的 squid 中，Q1 和 Q2 参数混杂在默认的配置文件里。最佳选择是，Q1 应该大于 Q2。

8.5.2 编译和配置 diskd

为了使用 diskd，你必须在运行./configure 时，在--enable-storeio 列表后增加一项：

```
% ./configure --enable-storeio=ufs,diskd
```

diskd 看起来是可移植的，既然共享内存和消息队列在现代 Unix 系统上被广泛支持。然而，你可能需要调整与这两者相关的内核限制。内核典型的有如下可用参数：

MSGMNB

每个消息队列的最大字节限制。对 diskd 的实际限制是每个队列大约 100 个排队消息。squid 传送的消息是 32－40 字节，依赖于你的 CPU 体系。这样，MSGMNB 应该是 4000 或更多。为安全起见，我推荐设置到 8192。

MSGMNI
整个系统的最大数量的消息队列。squid 对每个 cache_dir 使用两个队列。假如你有 10 个磁盘，那就有 20 个队列。你也许该增加更多，因为其他应用程序也要使用消息队列。我推荐的值是 40。

MSGGSZ
消息片断的大小（字节）。大于该值的消息被分割成多个片断。我通常将这个值设为 64，以使 diskd 消息不被分割成多个片断。

MSGSEG
在单个队列里能存在的最大数量的消息片断。squid 正常情况下，限制队列的长度为 100 个排队消息。记住，在 64 位系统中，假如你没有增加 MSGSSZ 的值到 64，那么每个消息就会被分割成不止 1 个片断。为了安全起见，我推荐设置该值到 512。

MSGTQL
整个系统的最大数量的消息。至少是 cache_dir 数量的 100 倍。在 10 个 cache 目录情况下，我推荐设置到 2048。

MSGMAX
单个消息的最大 size。对 Squid 来说，64 字节足够了。然而，你系统中的其他应用程序可能要用到更大的消息。在某些操作系统例如 BSD 中，你不必设置这个。BSD 自动设置它为 MSGSSZ * MSGSEG。其他操作系统中，你也许需要改变这个参数的默认值，你可以设置它与 MSGMNB 相同。

SHMSEG
每个进程的最大数量的共享内存片断。squid 对每个 cache_dir 使用 1 个共享内存标签。我推荐设置到 16 或更高。

SHMMNI
共享内存片断数量的系统级的限制。大多数情况下，值为 40 足够了。

SHMMAX
单个共享内存片断的最大 size。默认的，squid 对每个片断使用大约 409600 字节。
为安全起见，我推荐设置到 2MB，或 2097152。

SHMALL
可分配的共享内存数量的系统级限制。在某些系统上，SHMALL 可能表示成页数量，而不是字节数量。在 10 个 cache_dir 的系统上，设置该值到 16MB（4096 页）足够了，并有足够的保留给其他应用程序。

在 BSD 上配置消息队列，增加下列选项到内核配置文件里：

```
# System V message queues and tunable parameters
options         SYSVMSG         # include support for message queues
```

```
options         MSGMNB=8192         # max characters per message queue
options         MSGMNI=40           # max number of message queue identifiers
options         MSGSEG=512          # max number of message segments per queue
options         MSGSSZ=64           # size of a message segment MUST be power of 2
options         MSGTQL=2048         # max number of messages in the system
options         SYSVSHM
options         SHMSEG=16           # max shared mem segments per process
options         SHMMNI=32           # max shared mem segments in the system
options         SHMMAX=2097152      # max size of a shared mem segment
options         SHMALL=4096         # max size of all shared memory (pages)
```

在 Linux 上配置消息队列，增加下列行到/etc/sysctl.conf：

```
kernel.msgmnb=8192
kernel.msgmni=40
kernel.msgmax=8192
kernel.shmall=2097152
kernel.shmmni=32
kernel.shmmax=16777216
```

另外，假如你需要更多的控制，可以手工编辑内核资源文件中的 include/linux/msg.h 和 include/linux/shm.h。

在 Solaris 上，增加下列行到/etc/system，然后重启：

```
set msgsys:msginfo_msgmax=8192
set msgsys:msginfo_msgmnb=8192
set msgsys:msginfo_msgmni=40
set msgsys:msginfo_msgssz=64
set msgsys:msginfo_msgtql=2048
set shmsys:shminfo_shmmax=2097152
set shmsys:shminfo_shmmni=32
set shmsys:shminfo_shmseg=16
```

在 Digital Unix(TRU64)上，可以增加相应行到 BSD 风格的内核配置文件中，见前面所叙。另外，你可使用 sysconfig 命令。首先，创建如下的 ipc.stanza 文件：

```
ipc:
        msg-max = 2048
        msg-mni = 40
        msg-tql = 2048
        msg-mnb = 8192
        shm-seg = 16
        shm-mni = 32
```

shm-max = 2097152
shm-max = 4096

然后，运行这个命令并重启：
sysconfigdb -a -f ipc.stanza

一旦你在操作系统中配置了消息队列和共享内存，就可以在 squid.conf 里增加如下的 cache_dir 行：
cache_dir diskd /cache0 7000 16 256 Q1=72 Q2=64
cache_dir diskd /cache1 7000 16 256 Q1=72 Q2=64
...

8.5.3 监视 diskd

监视 diskd 运行的最好方法是使用 cache 管理器。请求 diskd 页面，例如：

% squidclient mgr:diskd
...
sent_count: 755627
recv_count: 755627
max_away: 14
max_shmuse: 14
open_fail_queue_len: 0
block_queue_len: 0

	OPS	SUCCESS	FAIL
open	51534	51530	4
create	67232	67232	0
close	118762	118762	0
unlink	56527	56526	1
read	98157	98153	0
write	363415	363415	0

请见 14.2.1.6 章关于该输出的详细描述。

8.6 coss 存储机制

循环目标存储机制（Cyclic Object Storage Scheme,coss）尝试为 squid 定制一个新的文件系统。在 ufs 基础的机制下，主要的性能瓶颈来自频繁的 open()和 unlink()系统调用。因为每个 cache 响应都存储在独立的磁盘文件里，squid 总是在打开，关闭，和删除文件。
与之相反的是，coss 使用 1 个大文件来存储所有响应。在这种情形下，它是特定供 squid 使用的，小的定制文件系统。coss 实现许多底层文件系统的正常功能，例如给新数据分配空间，记忆何处有自由空间等。

不幸的是，coss 仍没开发完善。coss 的开发在过去数年里进展缓慢。虽然如此，基于有人喜欢冒险的事实，我还是在这里描述它。

8.6.1 coss 如何工作

在磁盘上，每个 coss cache_dir 是一个大文件。文件大小一直增加，直到抵达它的大小上限。这样，squid 从文件的开头处开始，覆盖掉任何存储在这里的数据。然后，新的目标总是存储在该文件的末尾处。
squid 实际上并不立刻写新的目标数据到磁盘上。代替的，数据被拷贝进 1MB 的内存缓冲区，叫做 stripe。在 stripe 变满后，它被写往磁盘。coss 使用异步写操作，以便 squid 主进程不会在磁盘 I/O 上阻塞。
象其他文件系统一样，coss 也使用块大小概念。在 7.1.4 章里，我谈到了文件号码。每个 cache 目标有一个文件号码，以便 squid 用于定位磁盘中的数据。对 coss 来说，文件号码与块号码一样。例如，某个 cache 目标，其交换文件号码等于 112，那它在 coss 文件系统中就从第 112 块开始。因此 coss 不分配文件号码。某些文件号码不可用，因为 cache 目标通常在 coss 文件里占用了不止一个块。
coss 块大小在 cache_dir 选项中配置。因为 squid 的文件号码仅仅 24 位，块大小决定了 coss 缓存目录的最大 size：size = 块大小 x (2 的 24 次方)。例如，对 512 字节的块大小，你能在 coss cache_dir 中存储 8GB 数据。
coss 不执行任何 squid 正常的 cache 置换算法（见 7.5 章）。代替的，cache 命中被"移动"到循环文件的末尾。这本质上是 LRU 算法。不幸的是，它确实意味着 cache 命中导致磁盘写操作，虽然是间接的。
在 coss 中，没必要去删除 cache 目标。squid 简单的忘记无用目标所分配的空间。当循环文件的终点再次抵达该空间时，它就被重新利用。

8.6.2 编译和配置 coss

为了使用 coss，你必须在运行./configure 时，在--enable-storeio 列表里增加它：
% ./configure --enable-storeio=ufs,coss ...

coss 缓存目录要求 max-size 选项。它的值必须少于 stripe 大小（默认 1MB，但可以用 --enable-coss-membuf-size 选项来配置）。也请注意你必须忽略 L1 和 L2 的值，它们被 ufs 基础的文件系统使用。如下是示例：

```
cache_dir coss /cache0/coss 7000 max-size=1000000
cache_dir coss /cache1/coss 7000 max-size=1000000
cache_dir coss /cache2/coss 7000 max-size=1000000
cache_dir coss /cache3/coss 7000 max-size=1000000
cache_dir coss /cache4/coss 7000 max-size=1000000
```

甚至，你可以使用 block-size 选项来改变默认的 coss 块大小。
cache_dir coss /cache0/coss 30000 max-size=1000000 block-size=2048

关于 coss 的棘手的事情是，cache_dir 目录参数（例如/cache0/coss）实际上不是目录，它是 squid 打开或创建的常规文件。所以你可以用裸设备作为 coss 文件。假如你错误的创建 coss 文件作为目录，你可以在 squid 启动时见到如下错误：

2003/09/29 18:51:42| /usr/local/squid/var/cache: (21) Is a directory
FATAL: storeCossDirInit: Failed to open a coss file.

因为 cache_dir 参数不是目录，你必须使用 cache_swap_log 指令（见 13.6 章）。否则 squid 试图在 cache_dir 目录中创建 swap.state 文件。在该情形下，你可以见到这样的错误：

2003/09/29 18:53:38| /usr/local/squid/var/cache/coss/swap.state:
(2) No such file or directory
FATAL: storeCossDirOpenSwapLog: Failed to open swap log.

coss 使用异步 I/O 以实现更好的性能。实际上，它使用 aio_read()和 aio_write()系统调用。这点也许并非在所有操作系统中可用。当前它们可用在 FreeBSD,Solaris,和 Linux 中。假如 coss 代码看起来编译正常，但你得到"Function not implemented"错误消息，那就必须在内核里激活这些系统调用。在 FreeBSD 上，必须在内核配置文件中有如下选项：
options VFS_AIO

8.6.3 coss 发行

coss 还是实验性的功能。没有充分证实源代码在日常使用中的稳定性。假如你想试验一下，请做好存储在 coss cache_dir 中的资料丢失的准备。
从另一面说，coss 的初步性能测试表现非常优秀。示例请见附录 D。
coss 没有很好的支持从磁盘重建 cache 数据。当你重启 squid 时，你也许会发现从 swap.state 文件读取数据失败，这样就丢失了所有的缓存数据。甚至，squid 在重启后，不能记忆它在循环文件里的位置。它总是从头开始。
coss 对目标置换采用非标准的方式。相对其他存储机制来说，这可能导致命中率更低。
某些操作系统在单个文件大于 2GB 时，会有问题。假如这样的事发生，你可以创建更多小的 coss 区域。例如：

cache_dir coss /cache0/coss0 1900 max-size=1000000 block-size=128
cache_dir coss /cache0/coss1 1900 max-size=1000000 block-size=128
cache_dir coss /cache0/coss2 1900 max-size=1000000 block-size=128
cache_dir coss /cache0/coss3 1900 max-size=1000000 block-size=128

使用裸磁盘设备（例如/dev/da0s1c）也不会工作得很好。理由之一是磁盘设备通常要求 I/O 发生在 512 个字节的块边界（译者注：也就是块设备访问）。另外直接的磁盘访问绕过了系统高速缓存，可能会降低性能。然而，今天的许多磁盘驱动器，已经内建了高速缓存。

8.7 null 存储机制

Squid 有第 5 种存储机制叫做 null。就像名字暗示的一样，这是最不健壮的机制。写往 null cache_dir 的文件实际上不被写往磁盘。
大多数人没有任何理由要使用 null 存储系统。当你想完全禁止 squid 的磁盘缓存时，null 才有用。你不能简单的从 squid.conf 文件里删除所有 cache_dir 行，因为这样的话 squid 会增加默认的 ufs cache_dir。null 存储系统有些时候在测试 squid，和压力测试时有用。既然文件系统是典型的性能瓶颈，使用 null 存储机制能获取基于当前硬件的 squid 的性能上限。
为了使用该机制，在运行./configure 时，你必须首先在--enable-storeio 列表里指定它：
% ./configure --enable-storeio=ufs,null ...

然后在 squid.conf 里创建 cache_dir 类型为 null:
cache_dir /tmp null

也许看起来有点奇怪，你必须指定目录给 null 存储机制。squid 使用目录名字作为 cache_dir 标识符。例如，你能在 cache 管理器的输出里见到它。

8.8 哪种最适合我？

Squid 的存储机制选择看起来有点混乱和迷惑。aufs 比 diskd 更好？我的系统支持 aufs 或 coss 吗？假如我使用新的机制，会丢失数据吗？可否混合使用存储机制？
首先，假如 Squid 轻度使用（就是说每秒的请求数少于 5 个），默认的 ufs 存储机制足够了。在这样的低请求率中，使用其他存储机制，你不会观察到明显的性能改进。
假如你想决定何种机制值得一试，那你的操作系统可能是个决定因素。例如，aufs 在 Linux 和 Solaris 上运行良好，但看起来在其他系统中有问题。另外，coss 代码所用到的函数，当前不可用在某些操作系统中（例如 NetBSD）。
从我的观点看来，高性能的存储机制在系统崩溃事件中，更易受数据丢失的影响。这就是追求最好性能的权衡点。然而对大多数人来说，cache 数据相对价值较低。假如 squid 的缓存因为系统崩溃而破坏掉，你会发现这很容易，只需简单的 newfs 磁盘分区，让 cache 重新填满即可。如果你觉得替换 Squid 的缓存内容较困难或代价很大，你就应该使用低速的，但可信的文件系统和存储机制。
近期的 Squid 允许你对每个 cache_dir 使用不同的文件系统和存储机制。然而实际上，这种做法是少见的。假如所有的 cache_dir 使用相同的 size 和相同的存储机制，可能冲突更少。