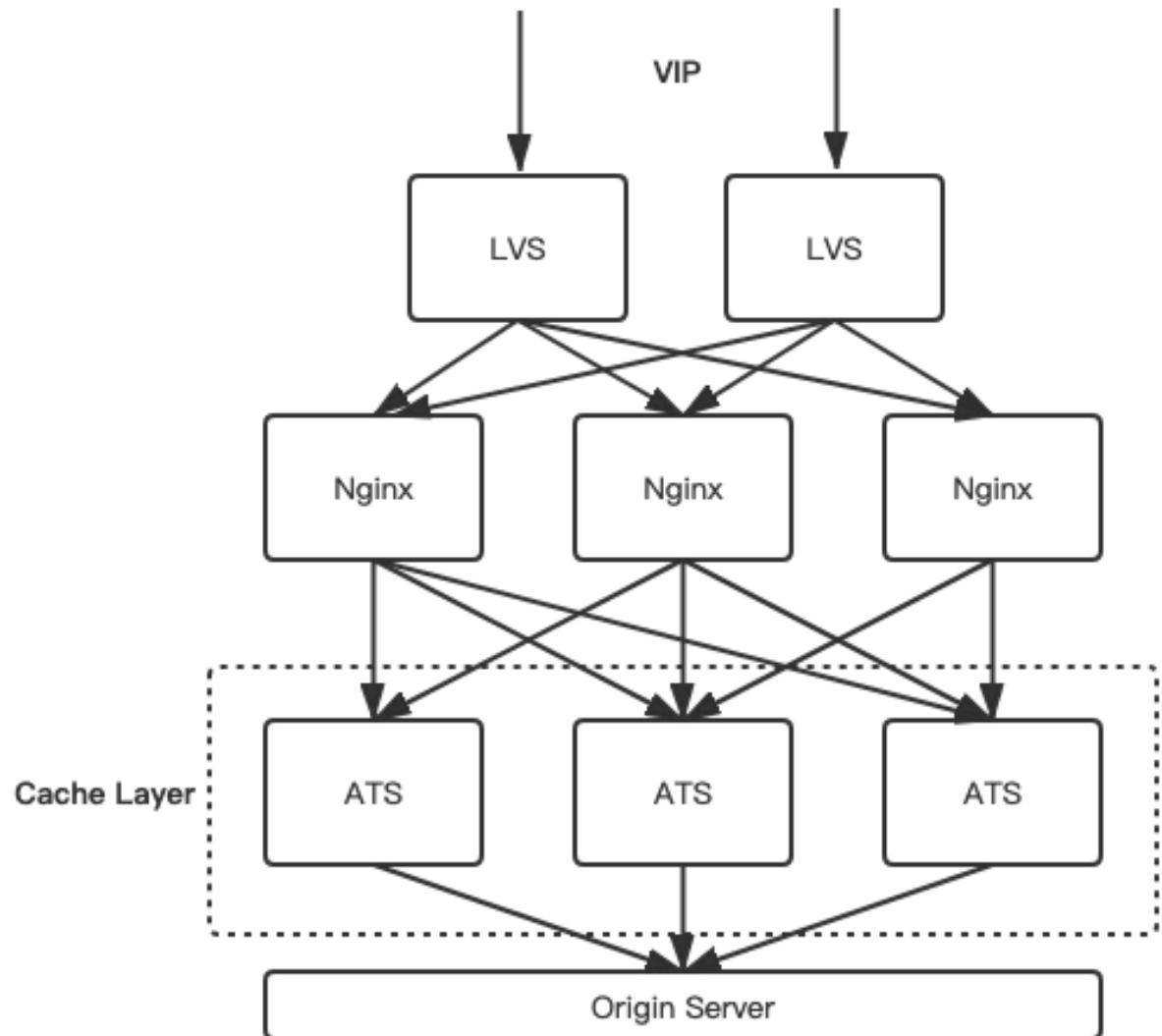


CDN缓存介绍

Apache Traffic Server简称ATS或TS，是一个高性能的、模块化的HTTP代理和缓存服务器。是一个开源项目，开发语言为C++。

ATS缓存功能：改进响应时间的同时降低了服务器负载与对带宽的需求，通过缓存并且重用经常请求的网页、图片和Web Service调用实现的。



缓存架构: 分片的概念-1

文件(File)分片

客户源站需要缓存的文件非常大时，从应用的角度，需要分割成同等大小的分片进行缓存

- 大文件作为一个对象缓存，失效或过期会引起大量回源流量
- 网民请求的仅仅是部分内容，如视频拖动，不需要一次回源整个文件
- 客户源站要求减少回源量，仅请求必要的内容，降低压力

文件分片行为在节点中由Nginx完成的，通过在http请求头中使用Content-Range，指定获取文件的部分内容在ATS上形成分片缓存。需要注意这种文件分片的概念对ATS是透明的，譬如客户源站上一个8M的文件，Nginx切分为4M大小分片，则Nginx会分别请求0-4M，4-8M两部分内容分别缓存到ATS上，Key假设如

<http://xxx.com/testfile?range=0-4M>

<http://xxx.com/testfile?range=4-8M>

对ATS来说就是两个对象。在这里我们称呼这种文件分片为对象(Object)。

缓存架构: 分片的概念-2

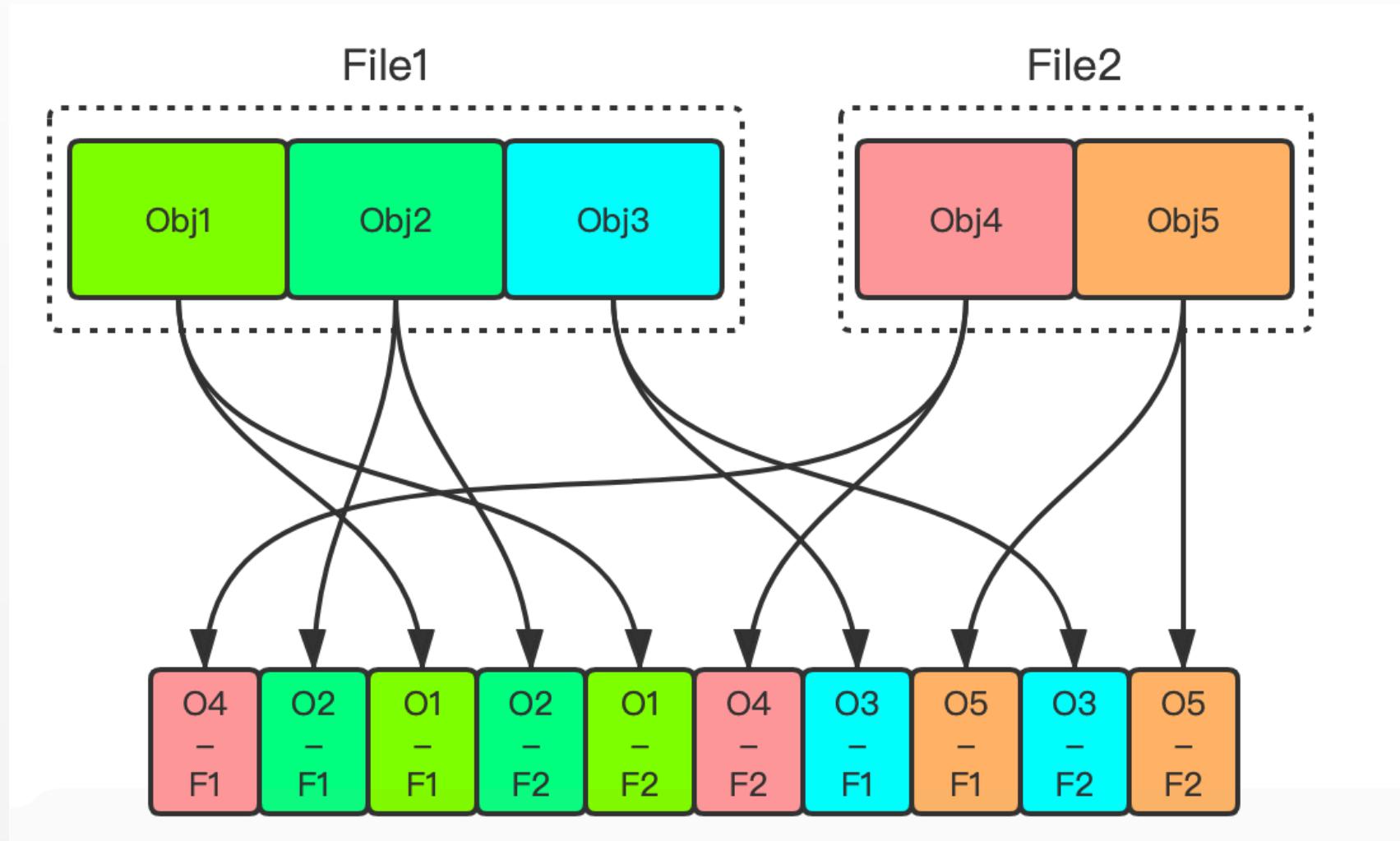
对象(Object)分片

对文件(File)分割成的多个对象(Object)，从存储的角度来看，对象(Object)还需要再次分片。

因为ATS对多个回源请求的并行性，多个回源是同时进行的，不可能一次存储太多的连续数据(一个完整的Object)，所以大对象必然要分片(否则并发的来很多大对象缓存请求将无法应对)。

单个分片在磁盘上是连续存储的。我们称呼这种对象分片为切片(Fragment)。

缓存架构: 分片的概念-3

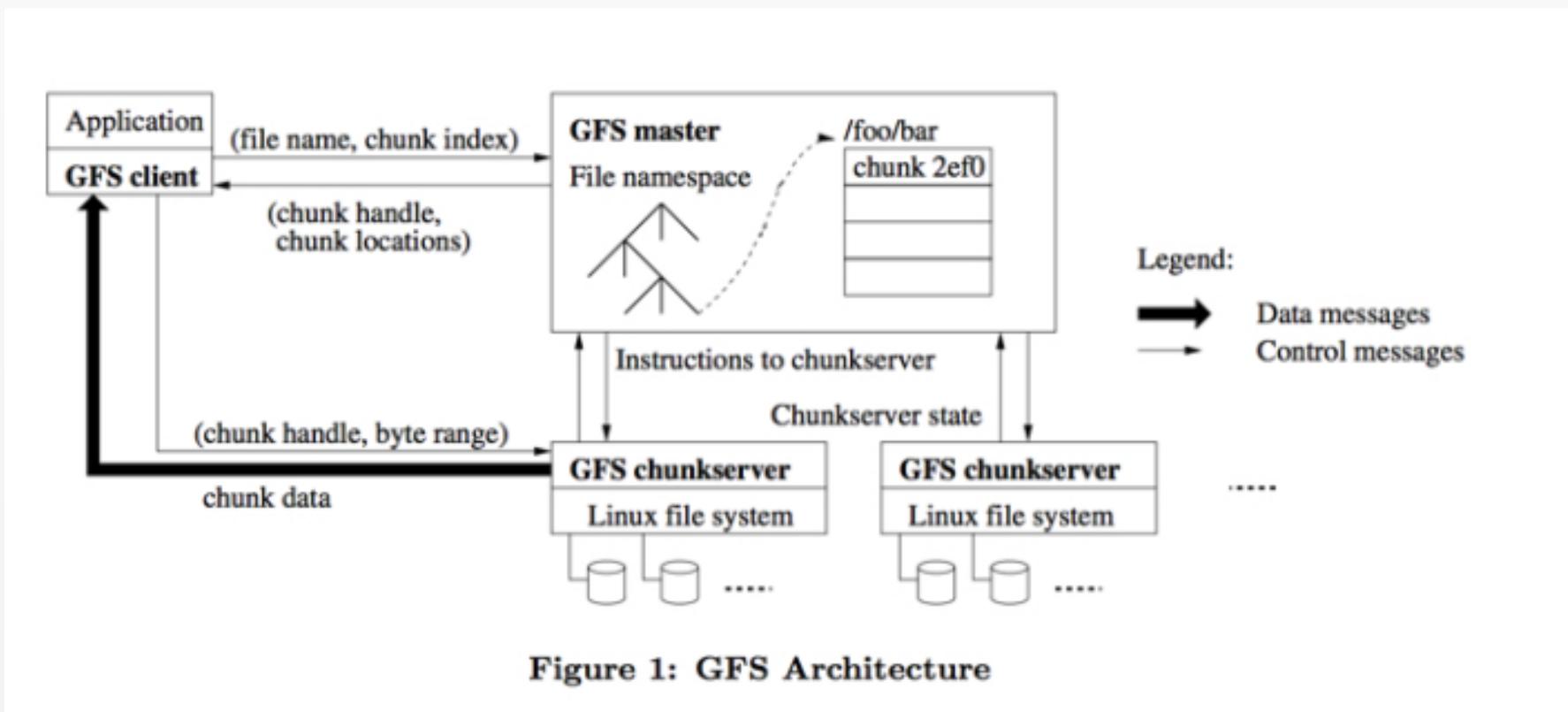


缓存架构: 架构对比

与分布式存储系统比较, 如GFS系统主要存储文件系统两类数据

- 文件元数据: 包括file system namespace(目录结构, 文件元数据)、文件存储位置信息等
- 文件数据: 文件在磁盘上的存储格式

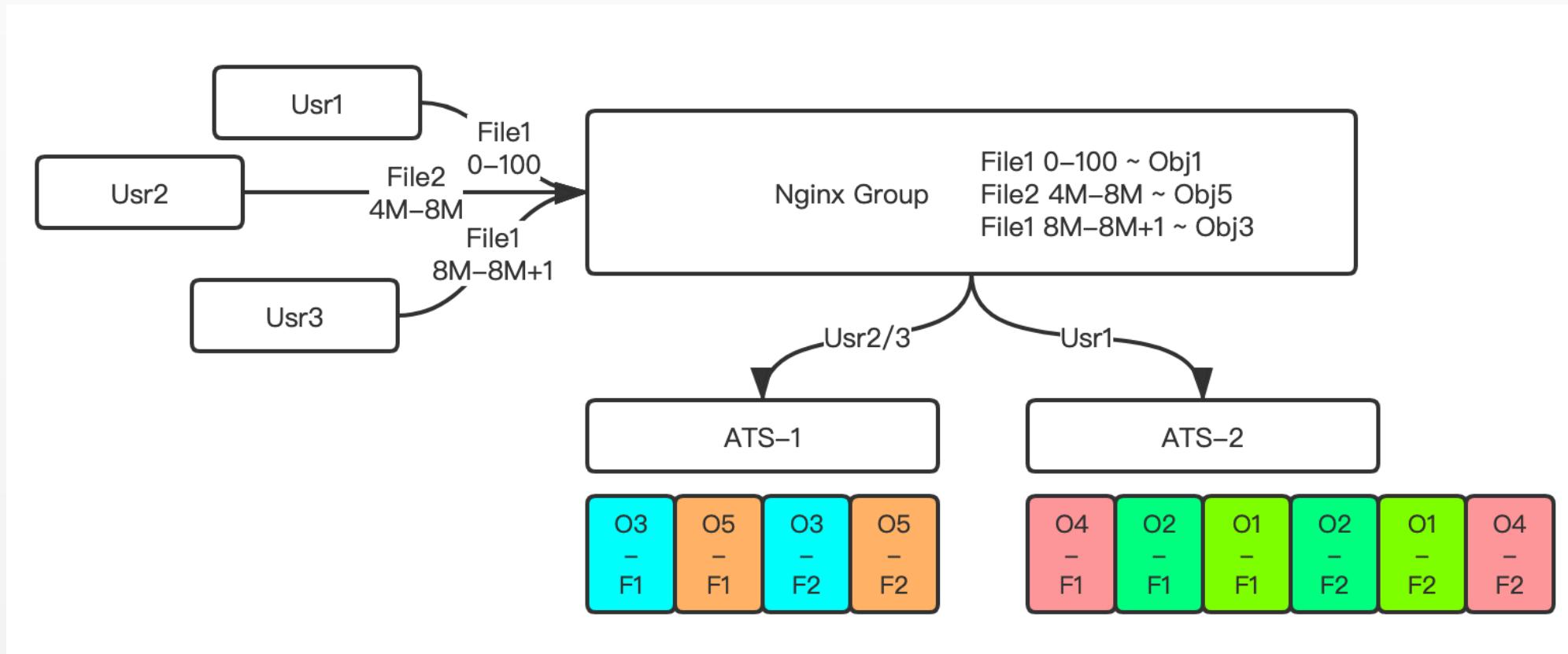
在GFS中, 文件被切割为固定大小的Chunk然后分散存储



缓存架构: 架构对比

存储异同

- 数据没有备份
- 文件存储位置信息不需要保存，通过一致性hash计算获得



单机架构: 数据结构

本质上存储的仍旧是KV数据，Key是上游(Nginx)访问的Url，Value是Url对应的用户源站文件的分片(Object)。因此使用什么样的数据结构和存储方式取决于数据特征，访问特征，存储能力等因素。

数据特征

- Key: 变长字符串，100字节~500字节不等Url
- Value: 文件分片(Object)，存储上分为多个Fragment(地址连续)，因此需要Key维护多个Fragment的索引关系

访问特征：连续时间访问的Url之间没有联系，无序和不可预测

- 随机读
- 随机写

存储需求

- 数据量巨大，更新快
- 允许存储失效，容忍可控的Cache Miss回源
- 需要考虑存储介质的成本

单机架构: 数据结构

最终选择的缓存结构: 以(多)磁盘缓存为主, 进程内内存缓存为辅的存储方式。

- 裸盘操作
- 快速索引: 索引和内容分开存储, 索引预读到索引内存缓存进行操作, 定期刷新回磁盘
- 磁盘I/O时间: 随机读通过内容内存缓存中获得内容; 随机写在程序内转换为追加写

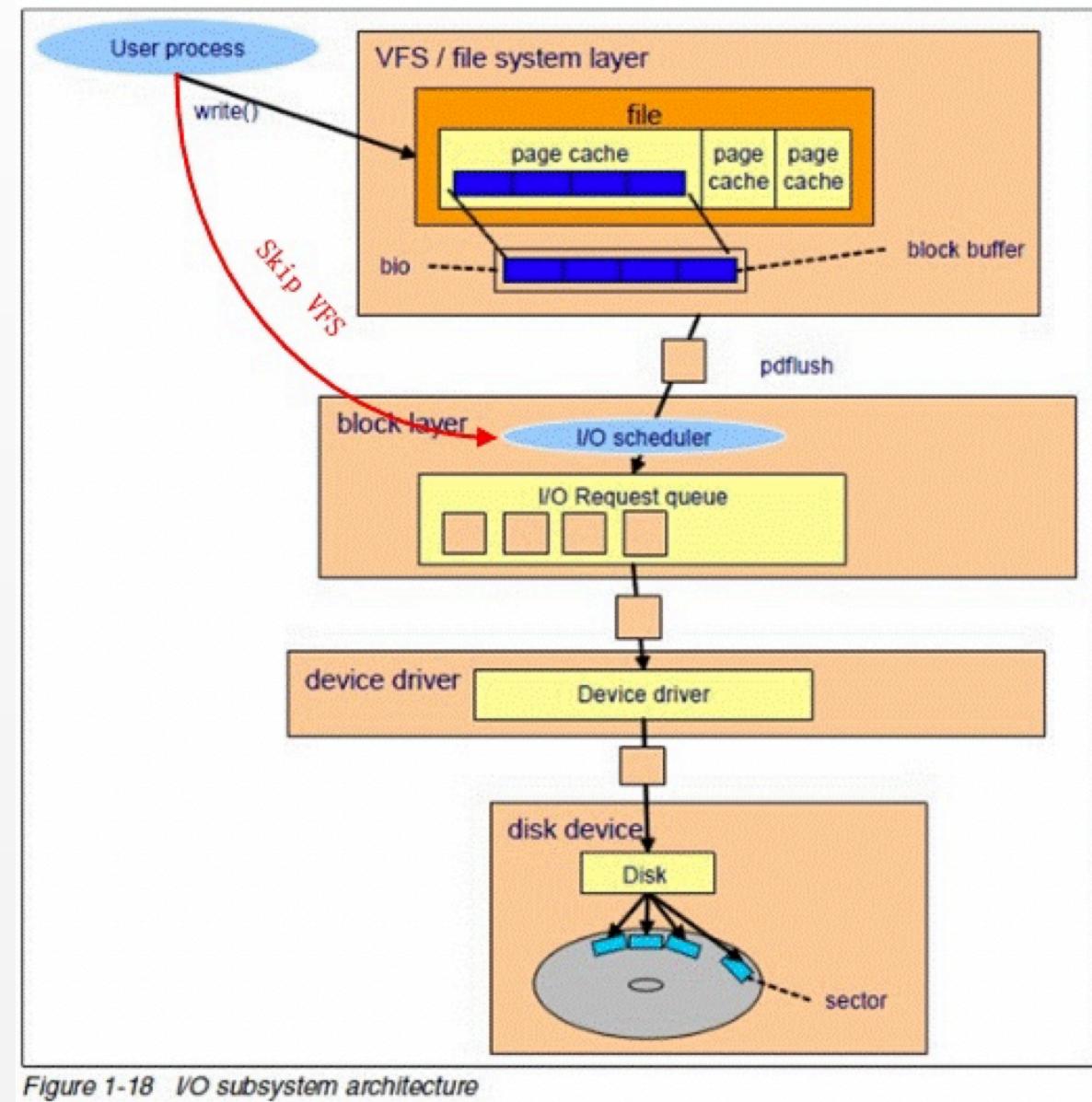
单机架构: 磁盘存储

为什么选择裸盘操作

第一，文件系统的元数据也会有IO的开销。而CDN的存储引擎自己进行缓存数据的管理，完全可以使用裸盘进行读写。消灭文件系统的开销。

第二，内存不足时，即使Page Cache中还有空闲内存，内核会使用Swap的内存，或者是回收内存带来的额外的CPU开销等问题；

对SATA机械盘以裸盘的方式操作，在内核中是直接走sd驱动，电梯层，到scsi到磁盘的，不需要走文件系统层和缓存层(O_DIRECT)。直接使用裸盘带来的另一个好处是可以使用内核提供的异步IO功能。异步IO可以解放CPU，进一步提高服务器的处理能力。



单机架构: Cache索引

对象(Object)的内容由若干个切片组成(Fragment)，通过索引来进行定位，索引中每一个元素我们称之为目录项(Dir)。每一个目录项代表了Cache中一段连续的存储空间，也就是一个切片(Fragment)。

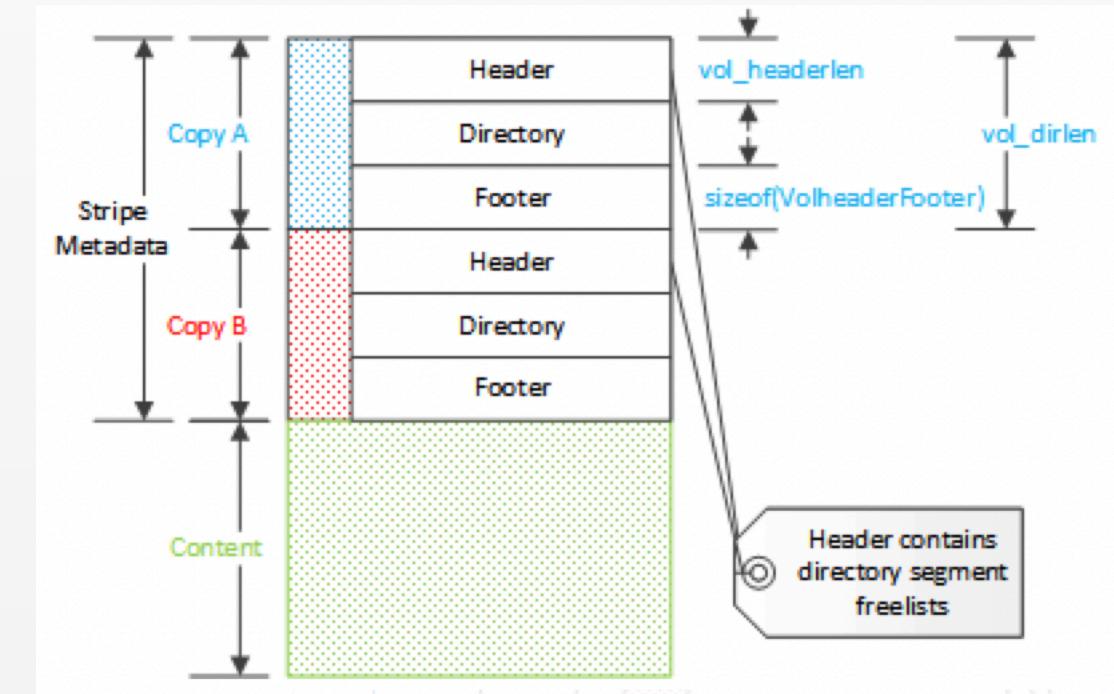
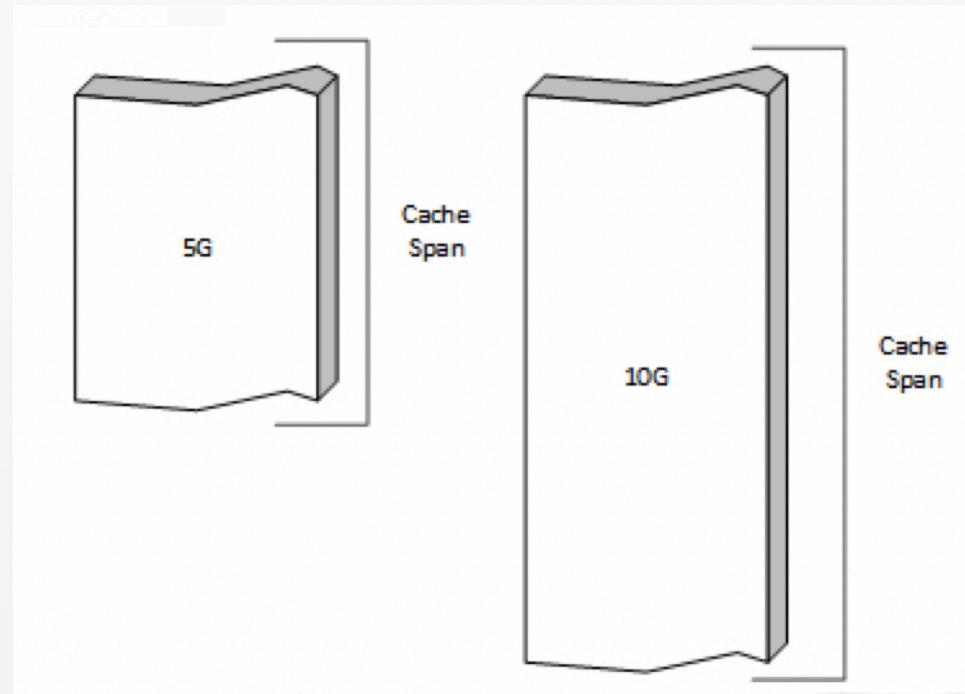
索引应该采用哪种数据结构保存：

- Dir数目：Dir和切片一一对应，与磁盘的大小线性相关
$$\text{Dir数目} = \text{切片数目} = \text{磁盘大小}/\text{平均切片大小} = (6T * 12 * 1024 * 1024) / 1M \sim 70\text{M条目}$$
- 不能直接使用Url作为索引：Url的字符串长度不确定而且容易过长，直接作为索引则内存/磁盘占用空间太大，如平均一个Url为300字节的字符串
$$\text{磁盘/内存空间} = 70M * 300 \sim 22G$$
- 连续访问的Url无规律，采用有序的数据结构会有大量计算(B+tree, RBTree, SkipList等)

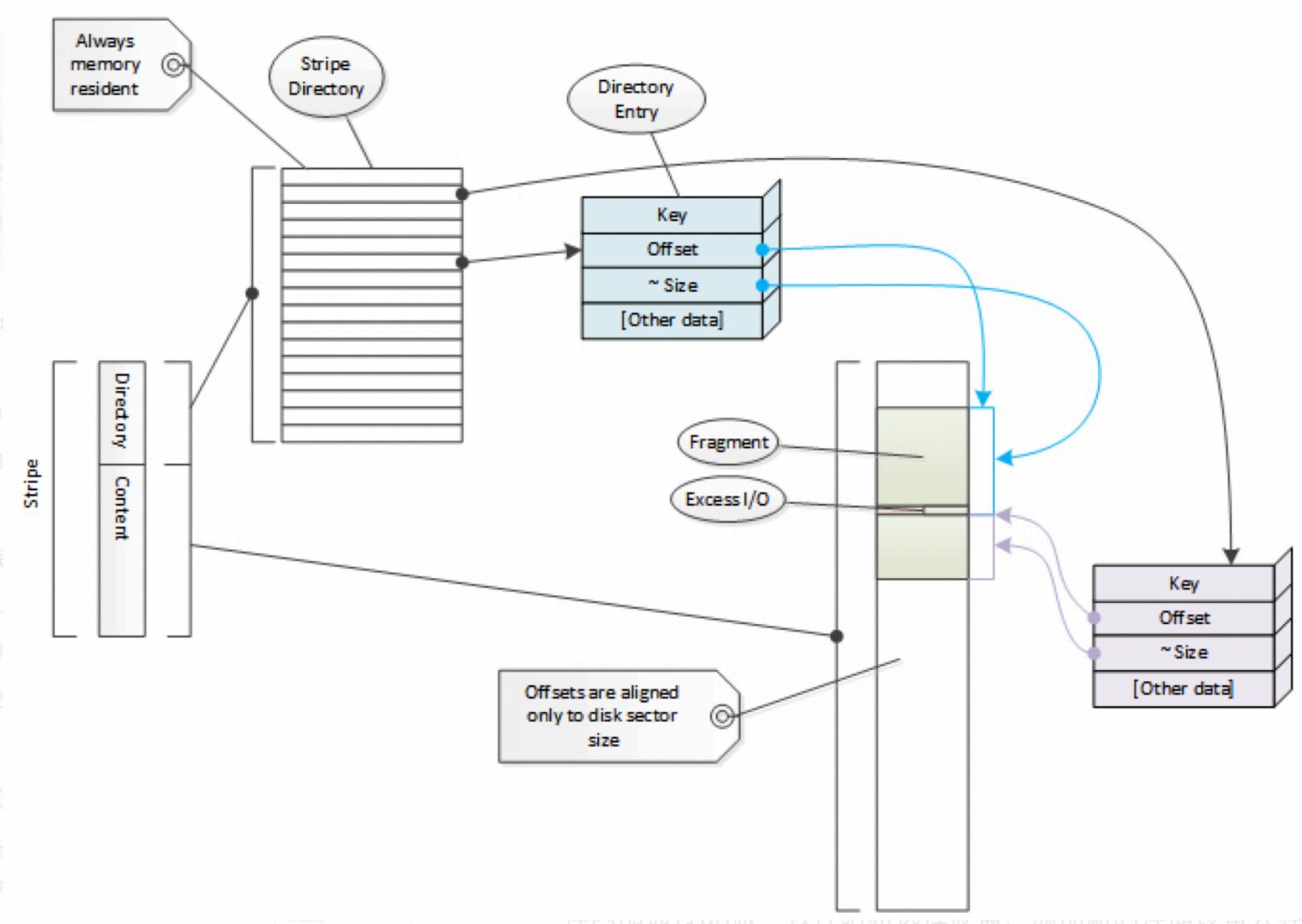
最终采用hashmap方式，极致精简索引大小，时间复杂度O(1)，空间复杂度O(10*n)，缺点是无序

单机架构: Cache索引

Meta元信息由三部分组成 - 头部、索引数据、尾部。元信息存储了两份，头部和尾部在代码中使用的是相同的数据结构VolHeaderFooter，这个数据结构的尾部包含一个变化长度的数组，这个数组用来保存每个段的空闲目录链表的表头，每一项包含对应段中空闲链表的第一项的索引，尾部其实是头部的拷贝，但不包含每个段的空闲链表数组。因此头部的大小会受目录项大小影响，但是尾部不会。



单机架构: Cache索引



单机架构: Cache索引

Cache Key : Url

Cache ID : hash(CacheKey)

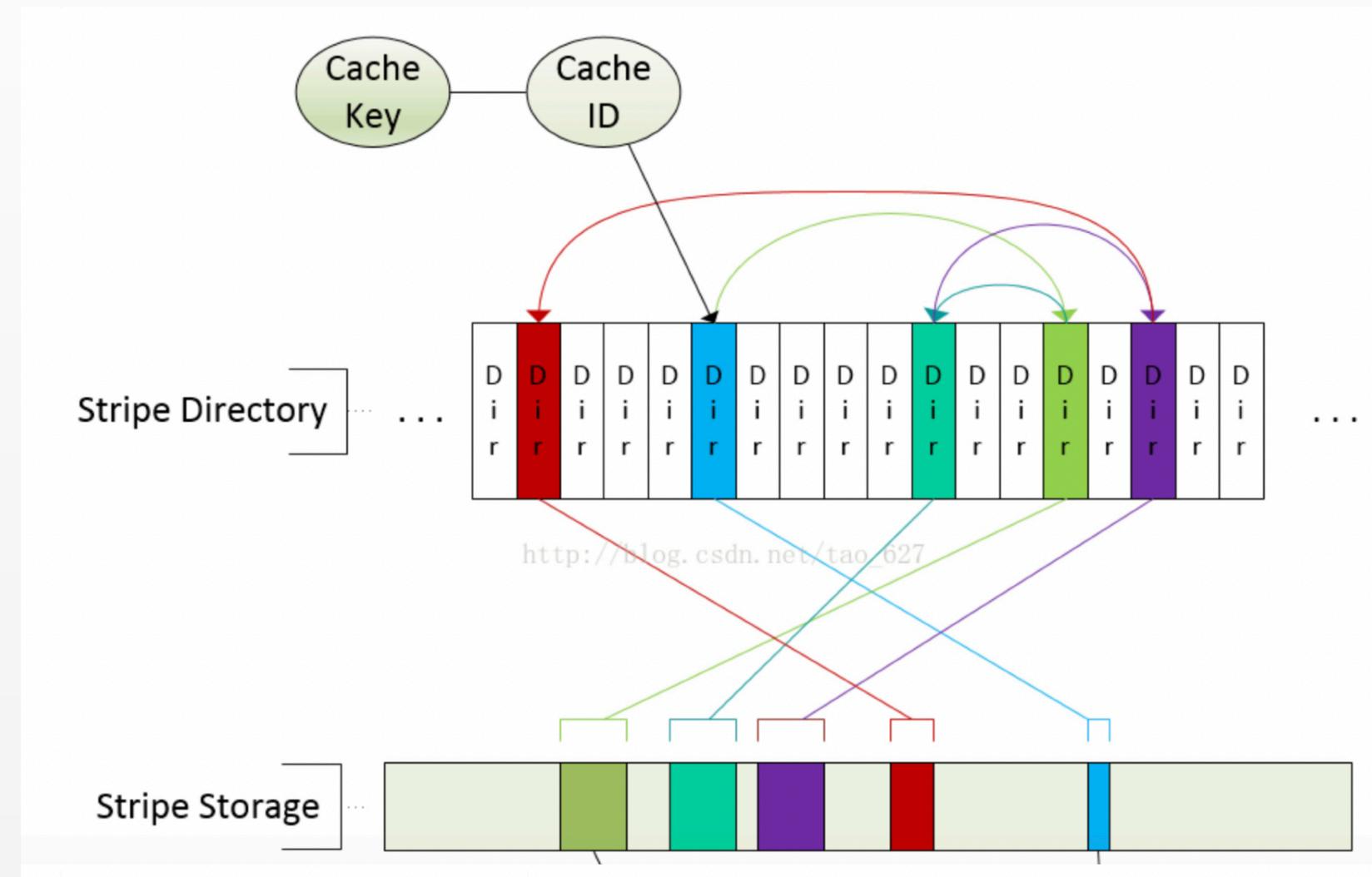
Dir : 共10Byte, 包括

分片大小

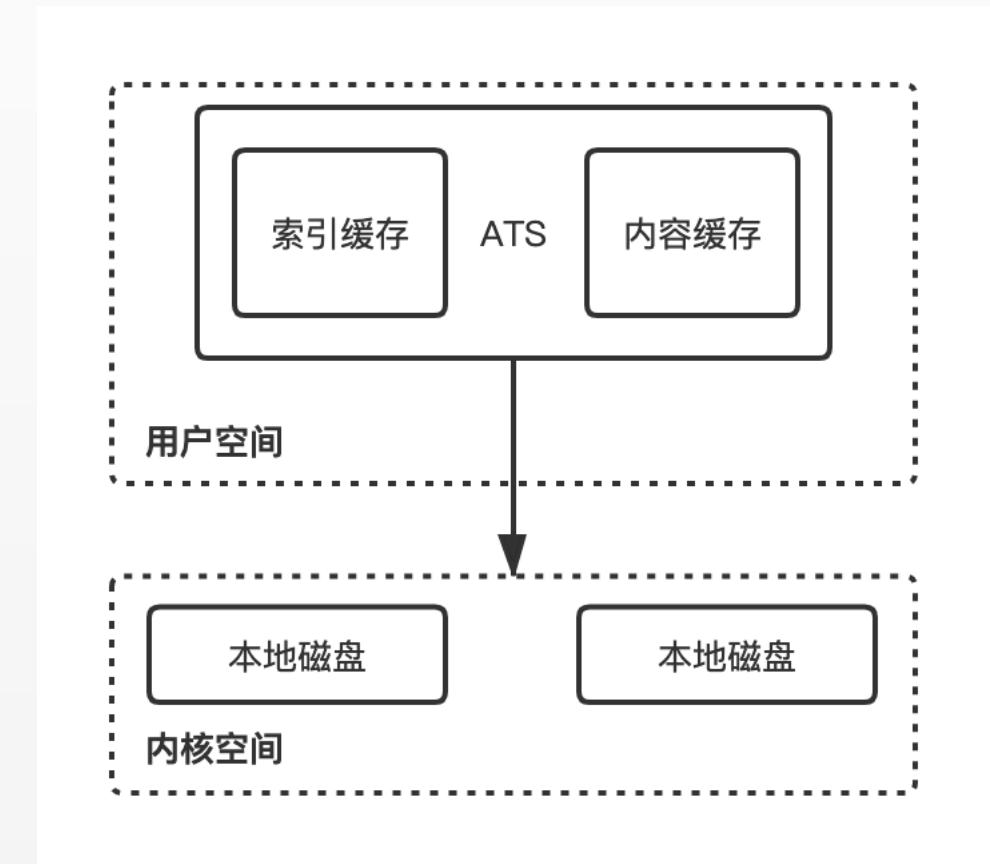
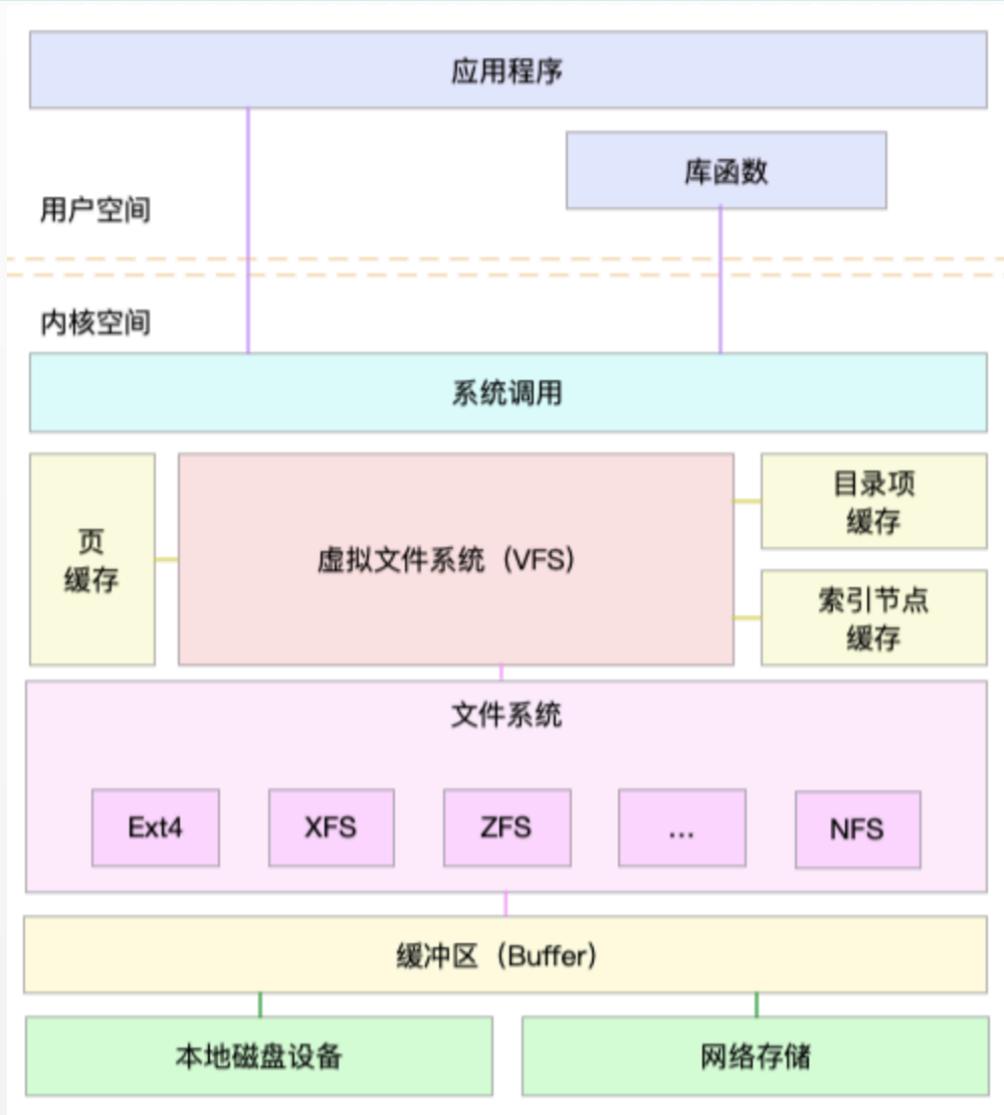
磁盘offset

hash tag

◦◦◦



单机架构: 内容存储



单机架构: 内容存储

磁盘存储

- 索引和Fragments在磁盘空间上分开，索引用于定位Fragment在磁盘上位置
- 追加写磁盘：内容区域会被当做一个环形的缓冲区，新对象会覆盖掉最早cache下来的对象。新的缓存对象在Cache中写到的位置被称为写光标，这意味着写光标到达的区域所保存的原来的对象将会被淘汰，即便这个对象还没有过期。当一个在磁盘上的对象被覆盖时，并不会立即的检测到因为索引并没有做更新，而是在后面读取对象分片的时候才会检测到失败。

内存存储

- 缓存索引：初始化时从磁盘加载到内存中，定期刷新到磁盘
- 缓存内容：预先配置一个总的大小，采用lru的方式淘汰

单机架构: 优缺点

优点:

- 索引效率高: 时间复杂度 $O(1)$, 空间复杂度 $O(10^n)$
- 磁盘I/O快: 随机读由RAM缓存提速; 随机写转化为追加写

缺点:

- 当一个磁盘初始化之后, 那么它对应的索引空间大小也就明确下来, 而且不会再改变。改变Fragment大小会导致整个索引失效引发大量回源, 严重时雪崩打死源站
- Nginx是通过对url做一致性哈希选择ATS, 节点内添加新的ATS服务器会导致部分已经缓存的内容出现Cache Miss回源
- 缓存数据没有备份机制, 各种异常情况会导致Cache Miss回源
- hashmap索引不支持有序操作, 因此统计功能比较差, 譬如无法根据域名统计信息
- 索引/内容的内存缓存总是不够用

改进思路

问题1：hash索引不支持顺序查找和遍历，统计功能差。如何合理有效的使用索引信息，譬如能够快速查询某个域名的磁盘占用情况，方便统计用户信息。

问题2：当一个磁盘初始化之后，那么它对应的索引空间大小也就明确下来，而且不会再改变。

- Dir数目 = 磁盘大小/平均切片大小

动态改变平均Fragment大小会重新计算索引条目数(改变了hash基数)，导致整个索引失效。

如何合理的设置平均Fragment大小，设置过小会导致不合理的磁盘/内存占用；设置过大可能导致索引节点不够，譬如客户源站的文件大小普遍小于设置的平均Fragment大小，可能导致索引节点耗尽，但内容存储空间还有很多。

问题3：没有备份；RAM空间不足

改进思路：问题1

MQ/Redis

将log信息推送到其他服务器进行后期处理。

优点：

- 支持多种数据结构
- 快速分析

缺点：

- 增加了代码复杂度，需要处理一致性问题
- 需要更多的资源，如内网带宽消耗，机器等

LevelDB

单机数据存储引擎，可以在程序中以库的方式使用。内存和磁盘结合的存储方式，由于数据有序，随机写和顺序读的性能高，基于16字节Key的QPS。淘宝的Tair就选择了LevelDB来做持久化。LevelDB总体思想即LSM tree，已在主流NoSQL系统中广泛使用。

缺点：

- 需要一定量内存/磁盘占用
- key约为500字节的url，字符串的比较排序性能，性能严重下降
- 访问索引的过程中访问磁盘，延迟变得不可控(SST合并，读取等)

改进思路: 问题2

提前分析需要存储的对象信息，在同一ATS上尽量缓存相同大小的对象，目前线上缓存节点按大文件和小文件两种存储。

拓展解决思路：

1. 能否对小文件采取其他缓存方式，而不通过ATS
2. 哈希表(索引)动态扩展(Hashing Index for Persistent Memory)

来自华中科技大学，好像实现了零突破。传统 hash table 的 resizing hash table 性能很差，因为 resizing hash table，需要创建一个新的 2 倍大小的 table，然后在将旧的 hash table 的数据复制过去。level hashing 通过 level，使得在读写性能 $O(1)$ 的情况下，表现出了非常好的 resizing 性能，可以减少 $2/3$ 的数据移动。

改进思路: 索引优化

SlimTrie: 战胜Btree单机百亿文件的极致索引

白山云开源的低内存, 高性能索引架构, 基于Trie树的优化方案。预先对所有的key 进行扫描, 提取特征, 大大降低索引信息的量。以做到单机100TB 数据为例, 如果文件都是10KB 小文件, 那么就有100 亿个文件(文件名 $\leq 1KB$), 常规方式需要10TB内存空间, SlimTrie 算法最终只需10GB 内存空间。

优点:

- 低内存, 高性能
- 支持顺序查找和遍历

缺点:

- 静态数据索引, 不支持新KV添加

适合大量静态小文件存储。

	空间开销	查询时间
Hash map	$O(k * n)$	$O(k)$
Skiplist, btree	$O(k * n)$	$O(k * \log(n))$
Trie	$O(k * n)$	$O(k)$
SlimTrie	$O(n)$	$O(\log(n))$

改进思路: 磁盘操作优化

随机读/追加写

问题1：如何提高随机读在RAM缓存的命中率

问题2：什么情况会打断追加写：如果不在内存缓存中，则需要到磁盘中读取，改变磁头位置(读写分离?)

解决方案：

- 分配更多缓存空间
- 优化LRU，将高频内容尽量保存在RAM，甚至不替换出去(Spin)
- 按照域名/目录的分类进行后台统计，内存/磁盘存储按照域名进行优先排序

改进思路: 磁盘操作优化

NVMe/bCache

NVMe:

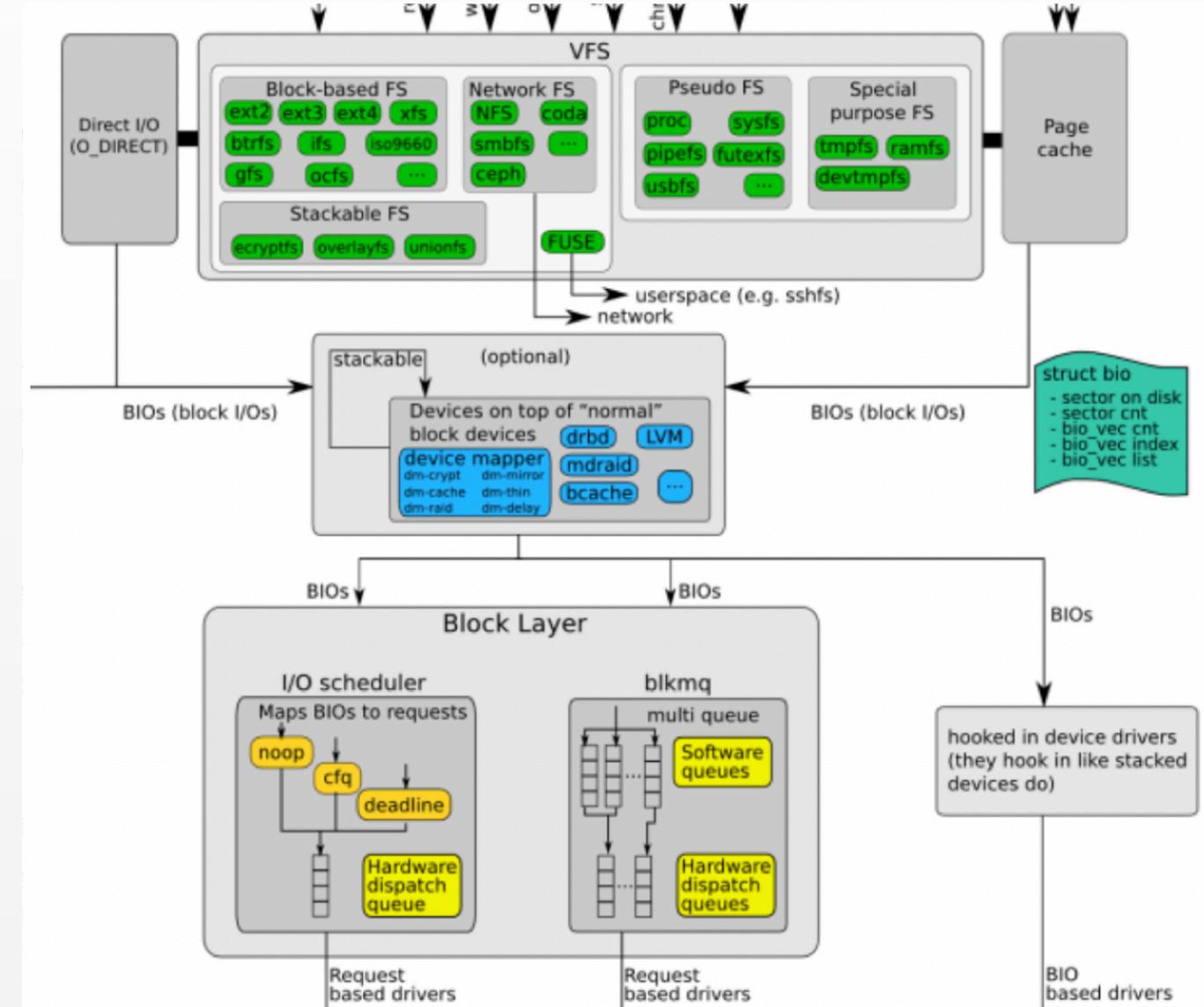
采用blk-mq (multi-queue) 的设备驱动方式, 一种新的方式引入了多级 queue。

bCache:

使用SSD在磁盘和内存之间增加一级缓存, 减少对磁盘IO操作, 从而提升整体性能。

SSD主要优势在于随机IO, bCache内部可检测是否为顺序IO并避开。

采用B+树和日志方式组织缓存数据。



改进思路: 磁盘操作优化

NVMe/bCache

