bilibili

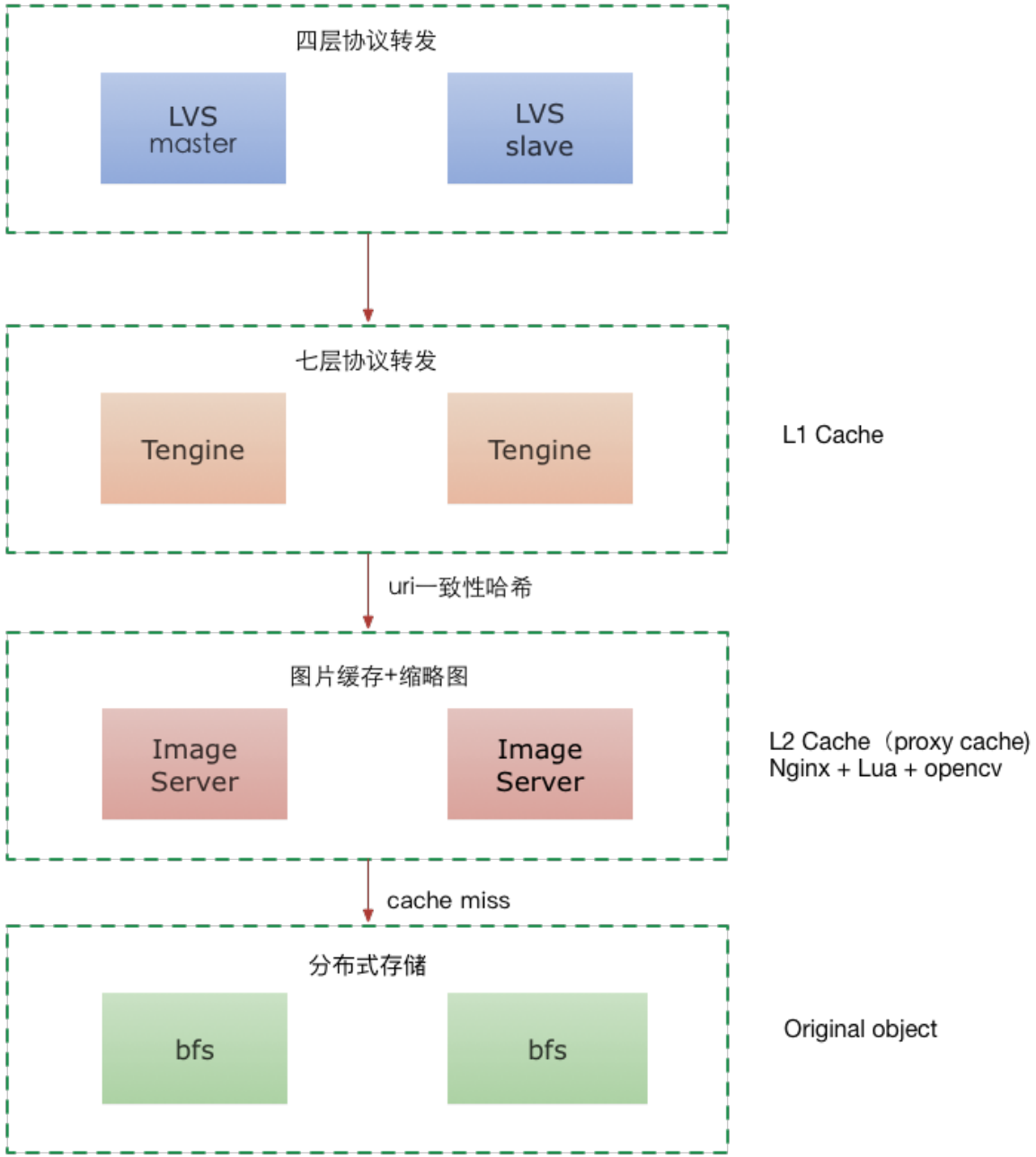
bilibili的存储实战（bfs）
—— 分布式小文件存储

目录
content

1 背景 & 整体架构

- B站文化社区的技术背景；
- MongoDB GridFS、FastDFS、TFS？
- 基于Facebook Haystack Paper；
- 我们需要什么样的存储系统（自研）？
  - 写非常多、读相对少（长尾效应）、从不修改、很少删除；
  - 非常稳定、高可用、可扩容、可扩展、方便运维；
  - 适合小文件（尤其是图片），各种扩展需求（高定制化）；

1 背景 & 整体架构



1 背景 & 整体架构

- OSPF + LVS：四层负载均衡；
- Tengine：七层负载均衡，L1 Cache proxy_cache；
- ImageServer：L2 Cache proxy_cache，处理实时缩略图（lua + opencv），当L1 Cache miss，使用uri的一致性hash命中；
- BFS：当L1/L2 Cache miss后，请求的终点，即对象存储，和ImageServer在同一个内网，所以访问速度也是很快的；
- More：源站的二级缓存节点，避免暴露核心机房IP给第三方CDN，二级缓存节点可以考虑大带宽的二线机房，流量比核心机房便宜，避免CDN故障时候直接打向核心机房；
- CDN -> L2 源站 -> L1 源站 -> 源站内部 L1、2 缓存 -> BFS；

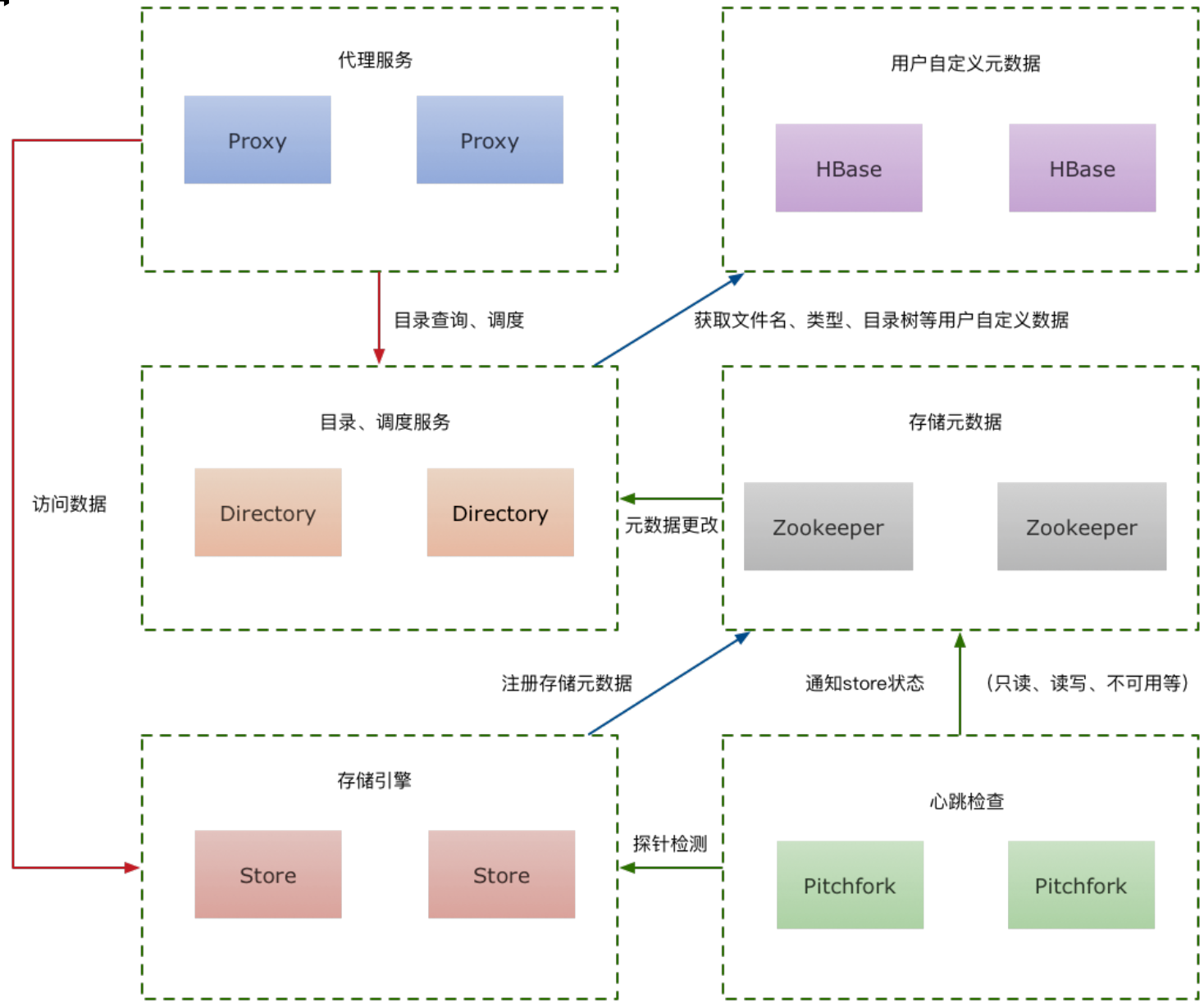
2 bfs 模块介绍

- 核心模块： proxy、 directory、 pitchfork， store；
- 运维模块： ops；
- 依赖： Zookeeper（元数据信息）， HBase（用户数据信息）；
- 流量走向：

  proxy -> directory

  -> store
- 总体来讲架构的特点核心：简单！

2 bfs 模块介绍



2 Store：负责所有图片数据存储

- 用户的图片分布在任意一个store节点；
- store存储是最核心的，架构需要简单！因此只提供增、删、读、修复操作；而类似存储镜像副本复制、心跳逻辑等一概没有；
- 唯一需要依赖的Zookeeper，是为了告诉集群有本节点存在。启动节点时候会把空闲volume和已分volume进行本地到ZK的双向同步；
- Volume：表示可以读写操作的具体卷，类似盘符（C：\），后面会详细介绍；

2 Pitchfork：心跳检测模块

- 监控所有的store节点，定期做probe探针检查，当出现异常时候会通知对应路径的zk node，设置具体状态；
- 上报store的统计数据如iops，free space等，方便directory调度模块使用；
- 单个pitchfork太累了，store节点越来越多的情况下，需要对任务进行分组，每个人各负责一部分；
- 把心跳任务隔离出单独进程，简化store模块本身的复杂度；

2 Directory：目录和存储调度服务

- 目录：按照bucket隔离存储不同业务的文件，从HBase中根据bucket和文件名作为Rowkey查找获取具体的store节点信息；
- 调度：根据pitchfork上报的store节点统计信息，进行打分计算，按照free space、iops、delay等综合条件进行调度，选出最合适的store节点集合；
- 内部信息都是根据Zookeeper在内存中构建的，因此本身无状态，非常容易扩展；

2 Proxy：代理模块，屏蔽内部业务复杂性

- 屏蔽内部先访问directory进行调度获取store节点，然后再根据store节点进行读取、写入行为等；
- 提供bucket验证和授权，因为bucket属于存储附加的特性，我们放在最外层进行最初的认证门槛；
- 面向资源的RESTful API设计；
- 可以提供更多的任务，比如读文件失败，可发送修复任务给队列；比如写文件，可发送文件信息给队列，消费者可以进行异构备份或者跨机房同步等；
- proxy 结构会让系统内部多一次HTTP 请求，但是整体逻辑可维护性变高了，也方便之后在proxy 中完成更多逻辑；

3 存储模块：store 背景

- 目录文件多了访问很慢，常用做法根据文件名HASH创建00-FF开头的目录来打散文件；
- 单个文件inode占用几百字节，浪费空间（海量小文件）；
- 打开文件调用open，进行一次IO查找操作，缓存所有的fd提前打开，但是海量文件依然不太可能全部保存到内存中，而每次打开又用IO成本；

3 存储模块：store 方案

- 文件多，我们就合并！这里引入一个概念Superblock（超级大块），把海量小文件合并成一个大文件；
- 机械硬盘优化写入，尽可能的顺序写，减少寻道和旋转延迟；
- 组成Superblock的一个个小文件称之为Needle（类似探针的意思）；
- 解决了两个大难题：inode少了，优化了IO（写入）；

3 存储模块：store

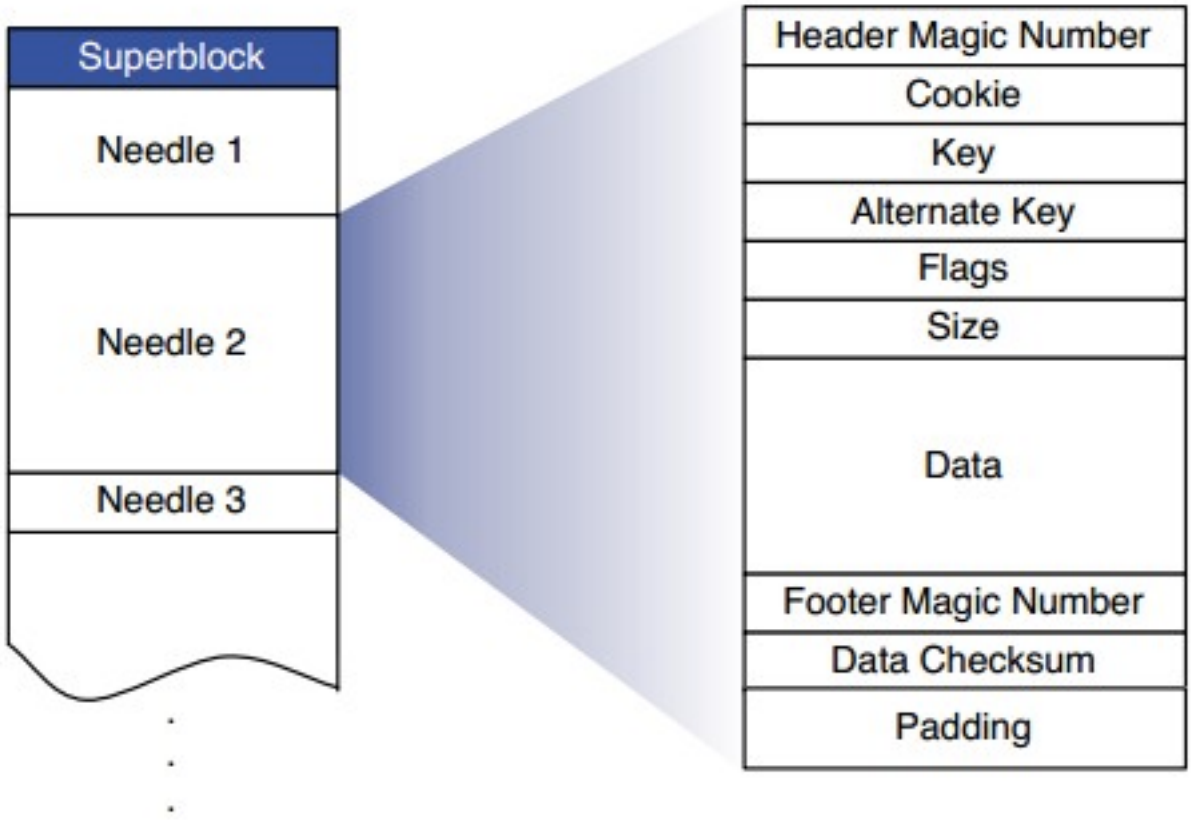


Figure 5: Layout of Haystack Store file

3 存储模块：store

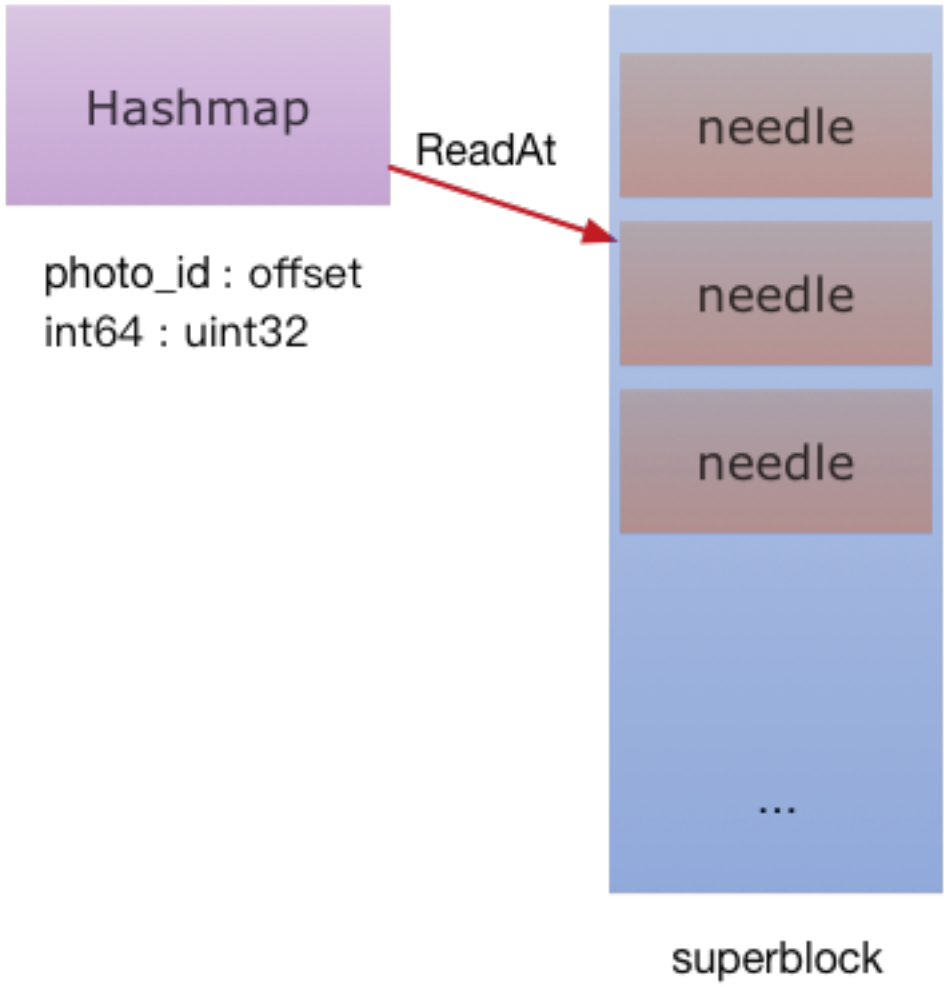
Field	Explanation
Header	Magic number used for recovery
Cookie	Random number to mitigate brute force lookups
Key	64-bit photo id
Alternate key	32-bit supplemental id
Flags	Signifies deleted status
Size	Data size
Data	The actual photo data
Footer	Magic number for recovery
Data Checksum	Used to check integrity
Padding	Total needle size is aligned to 8 bytes

Table 1: Explanation of fields in a needle

3 存储模块：读取文件

- Needle中包含一个key（图片、文件ID，int64），因此我们可以在内存中保存一个hashmap[key]offset，来进行查找block的偏移；
- 以前需要打开很多FD保存，现在只需要保存block的一个fd，然后查找hashmap内存数据，这样就少了一次元数据查找的IO；
- 通过posix fadvise告诉内核关闭预读，一旦读取一个图片我们上层L1/L2甚至CDN都会立马缓存，减少没必要的内存开销；

3 存储模块：读取文件



3 存储模块：写入文件

- append-only特性，直接追加到block尾部，然后插入或者更新hashmap的键值对；同一个key替换内存数据，会导致物理文件残留“孤儿Needle”；
- 优化内存占用，偏移从int64变成了uint32；那uint32最多寻址4GB，但Needle按照默认8字节对齐，寻址范围就变成了4GB * 8 = 32GB;
- 我们使用bufferIO，提供类似sync_binlog一样的参数控制多久flush磁盘，当flush后，按照策略清理内核的page cache，因为上层已经缓存过了（fadvise，sync_file_range);
- 使用posix falloc预分配空间，提前占用以及让extent连续，减少碎片；

3 存储模块：删除文件

- 使用逻辑删除，标识文件被删除的状态；内存和Needle中使用Flag来进行区分；
- 先根据原hashmap中的offset定位文件，然后使用WriteAt更新block；
- 修改hashmap中的offset为0，即0表示已删除的意思；

3 存储模块：索引文件

- store重启要重新构造hashmap，可根据Superblock扫描来还原，但IO成本比较高，建立hashmap数据的索引文件来我完成fast recovery！

Field	Explanation
Key	64-bit key
Alternate key	32-bit alternate key
Flags	Currently unused
Offset	Needle offset in the Haystack Store
Size	Needle data size

Table 2: Explanation of fields in index file.

3 存储模块：索引文件

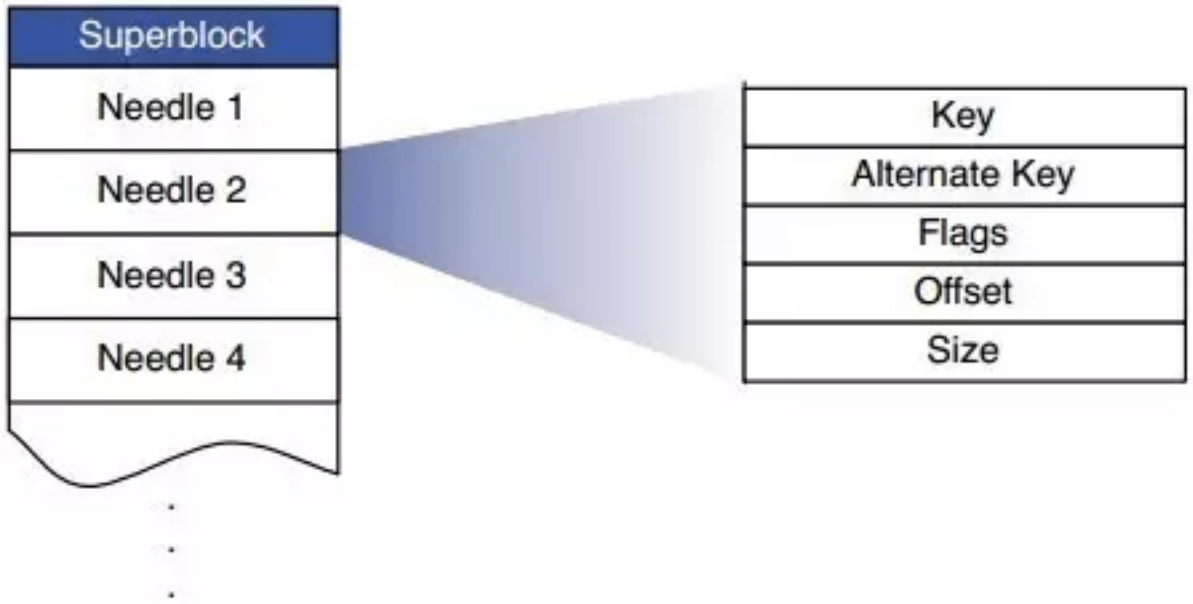
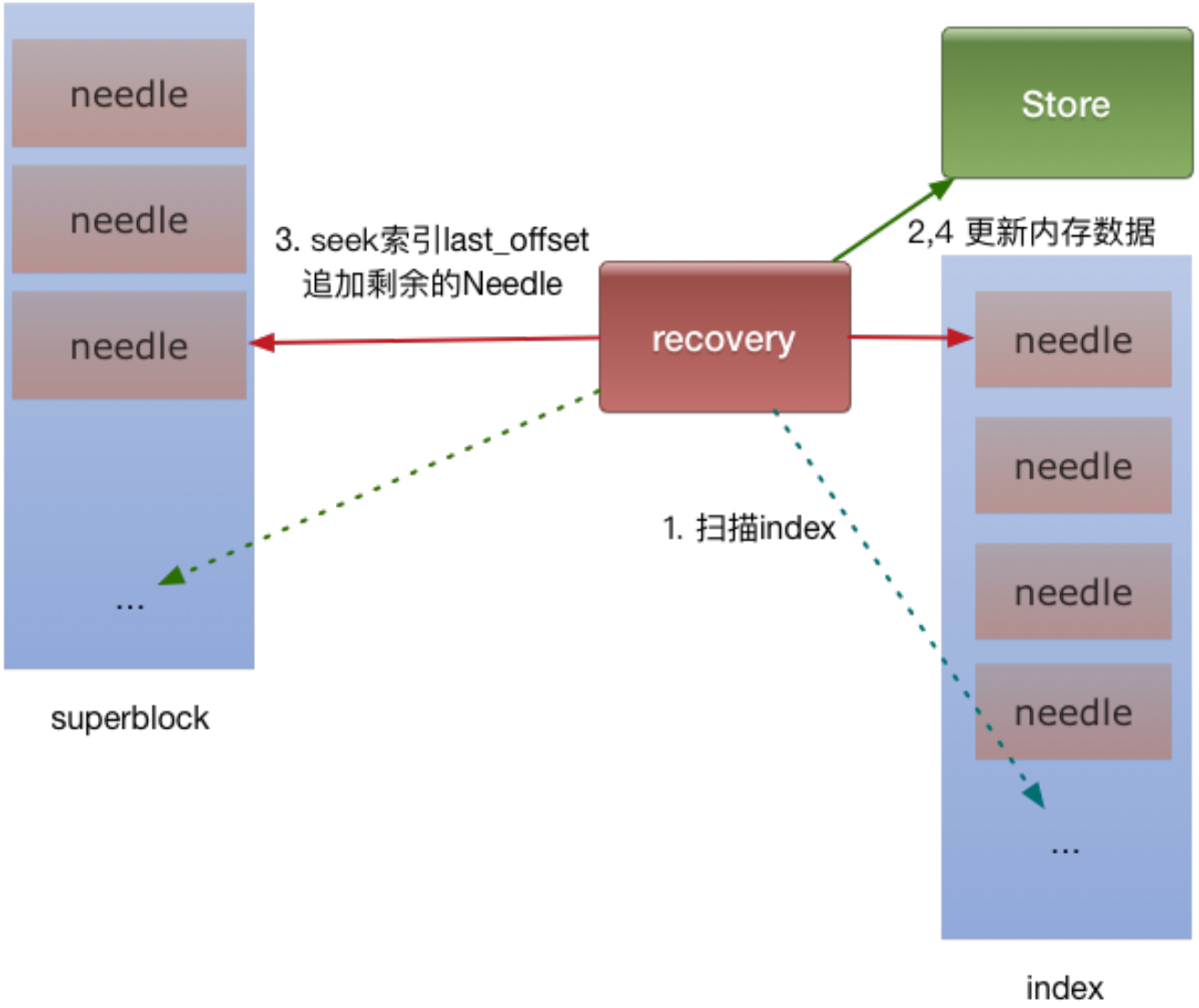


Figure 6: Layout of Haystack Index file

3 存储模块：索引文件

- Volume：一个Superblock + 一个 Index；
- 为了加速写入文件的过程，我们使用golang channel来异步化，会在内存中先聚合一部分数据，一次写入；
- 删除操作不会写入索引，使用“Lazy Read”，当读取到Needle 的时候，发现flag为删除，这时候再来更新hashmap；
- 异步会导致和hashmap数据不一致，需要进行补起操作进行修复；

3 存储模块：索引文件



3 存储模块：压缩

- 文件替换会导致“孤儿Needle”，删除文件是逻辑删除，因此需要定期压缩Block来重复利用占用的空间；
- 核心思路是COW（Copy-On-Write），对原始文件进行一次复制，当到文件尾部的时候，所有修改操作阻塞（加锁），然后原子替换变量来实现；
- 在复制过程中也有删除行为，因此一旦复制开始，需要全量记录，然后在新的Block进行重放Redo；

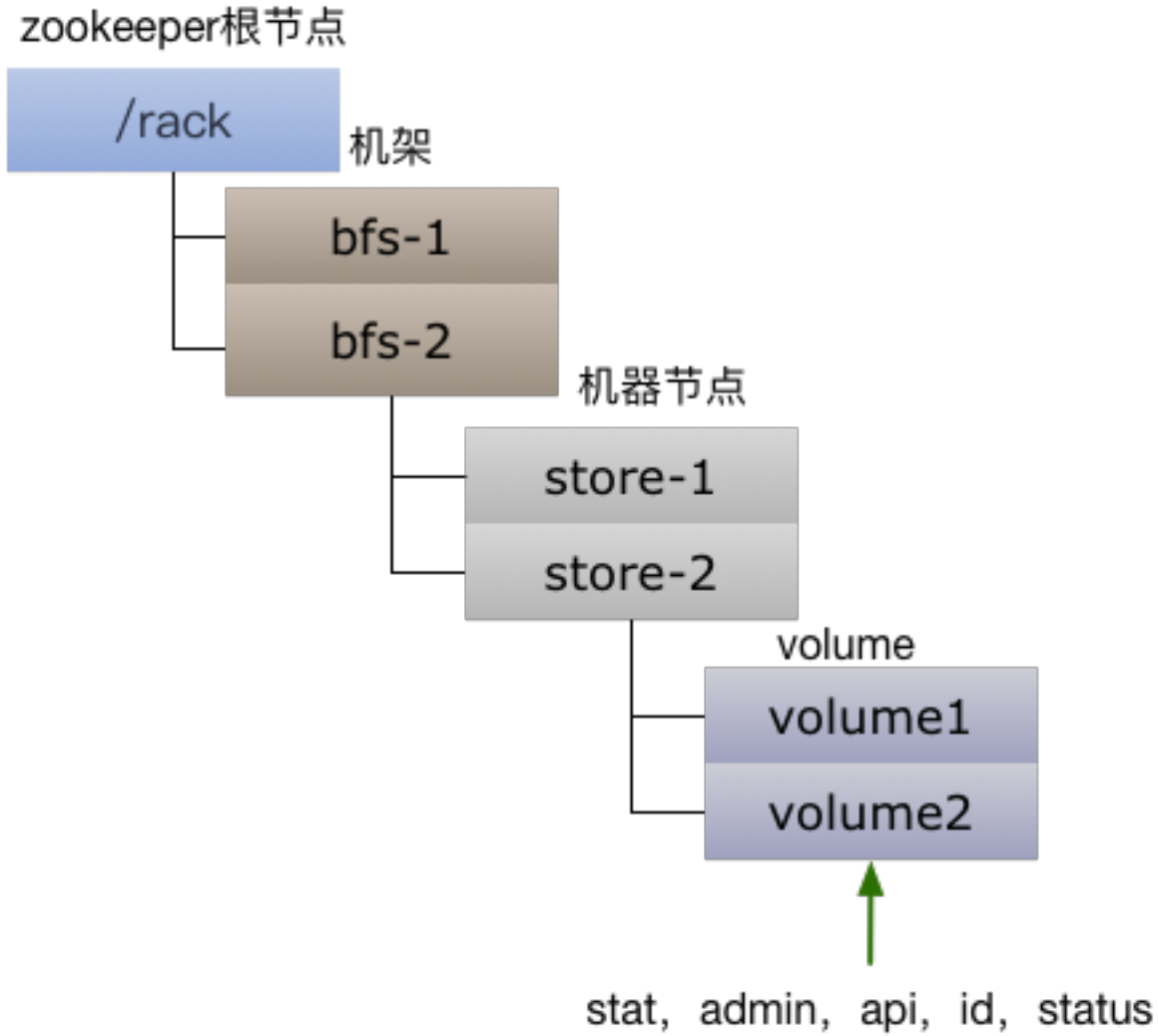
3 存储模块：故障恢复

- 任何故障，都会被pitchfork检查出来，这时候会设置整个store不可用（为了运维方便，我们粒度是按照store节点来做的）；
- 一旦确认有问题，更新Zookeeper的状态，设置成只读或者不可用，如果是单个Needle checksum失败，可以进行异步任务的修复或者手动强制修复，再重新设置状态可用；
- 有时候更严厉的情况下，只能从其他store节点进行bulk sync逻辑，这会消费大量内网带宽，目前还没有更好的方案；

3 存储模块：批量操作优化

- B站的视频缩略图，类似Qzone相册都有一个共同点，就是会批量写入，因此我们可以利用这个特点，提供批量操作接口，这样连续的大块的文件写入，是可以最大化利用机械磁盘顺序写的特点；

3 存储模块：meta信息



3 存储模块：meta信息

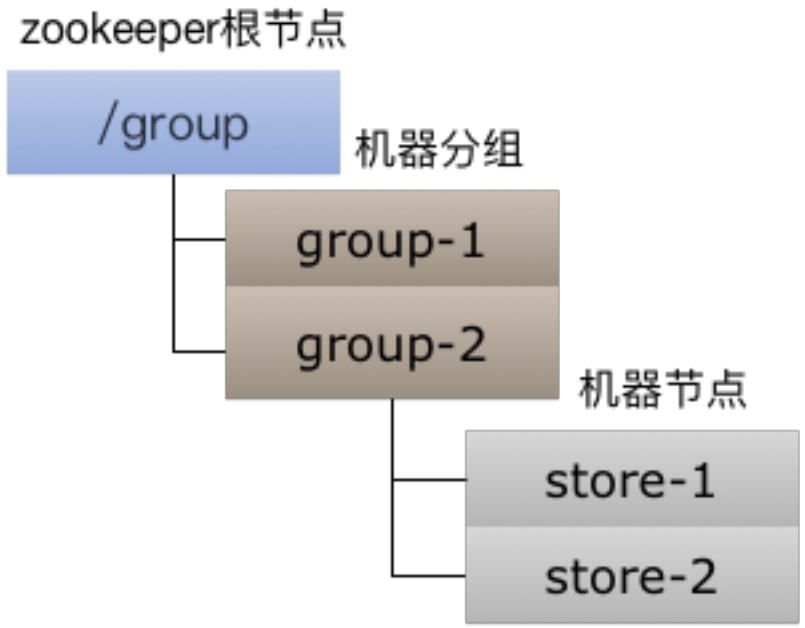
- 一个机架下有N个store节点；
- 一台物理机部署一个store节点；
- 一个store节点包含N个Volume；
- 一个Volume包含一个Superblock和一个索引文件；
- 后面还会提到Store节点还需要分组（Group）使用（比如跨机架容灾等）；

4 运维模块：ops

- 开源软件的经常忽视的一点就是运维，这会导致生产环境难以使用！因此bfs把ops单独拆分出来开发，目前公司前端同事正在使用VUE配合API来完成整个运维平台！

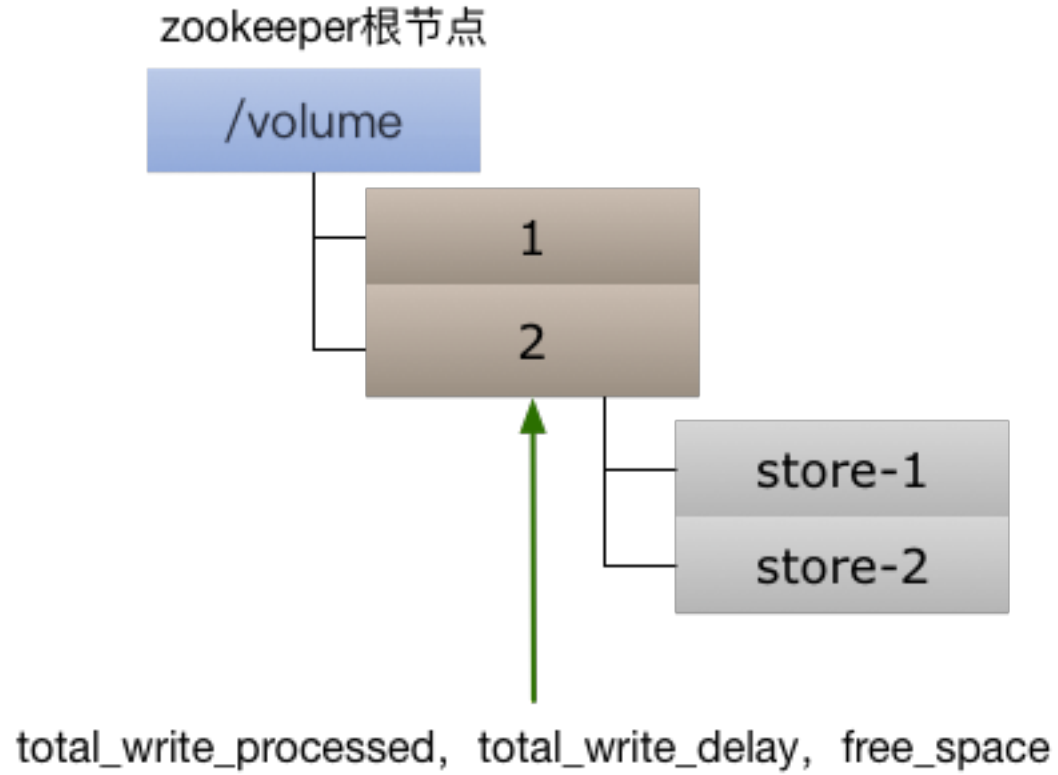
4 运维模块：分组Group

- bfs的副本策略是“镜像”（非EC，私有云有钱任性）；
- directory模块使用的时候，是按照分组获取store资源的，比如可以按照机架A、B上的两个Store组成一个分组，分组中的资源是镜像副本的；



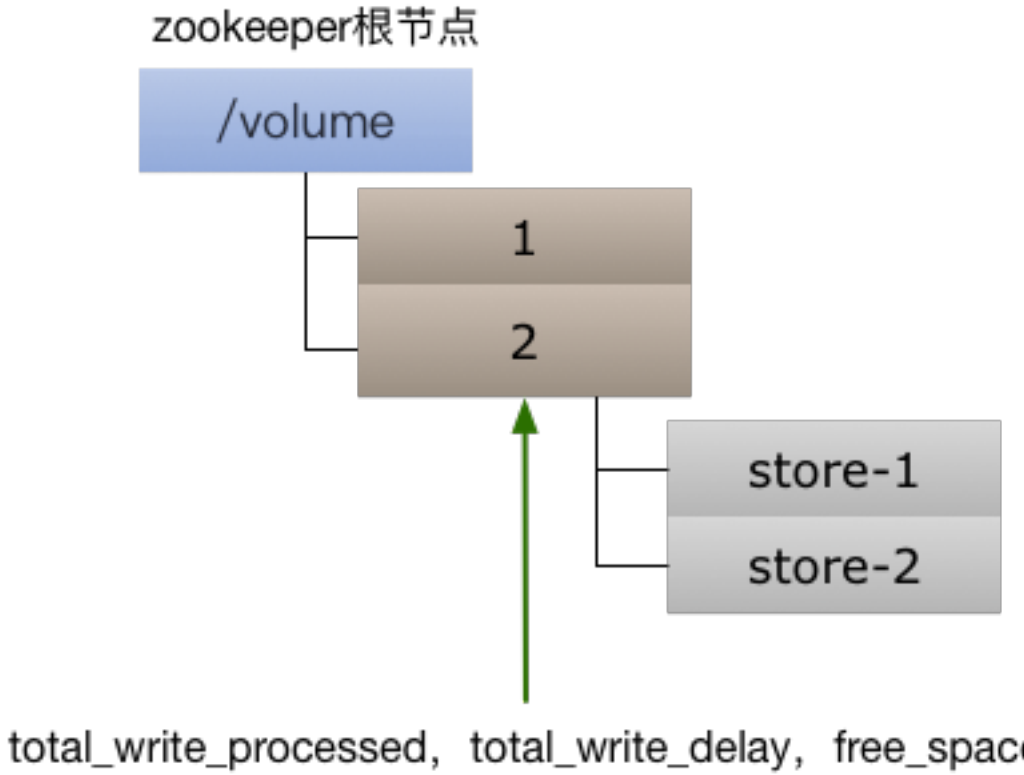
4 运维模块：卷轴Volume

- 创建好分组以后，去对应的Store分配Volume了，比如GroupA有StoreA和StoreB，那么新创建的Volume-1，需要去对应的两个Store节点分别创建；



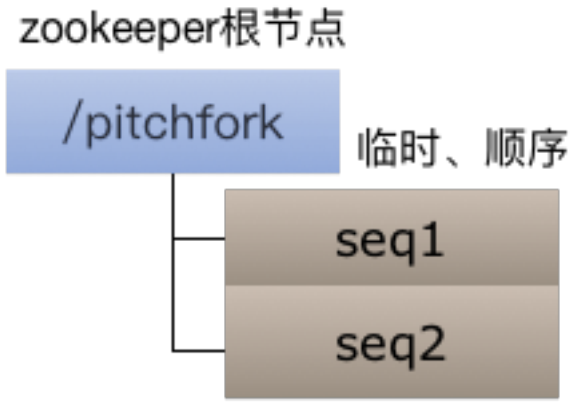
4 运维模块：卷轴Volume

- 这个也是上文提到的pitchfork需要上报统计信息的zk node；
- 这个关注的是Volume下有哪些store节点，而store模块提到的meta关注的是store下有哪些Volume，相辅相成；



5 心跳模块：pitchfork

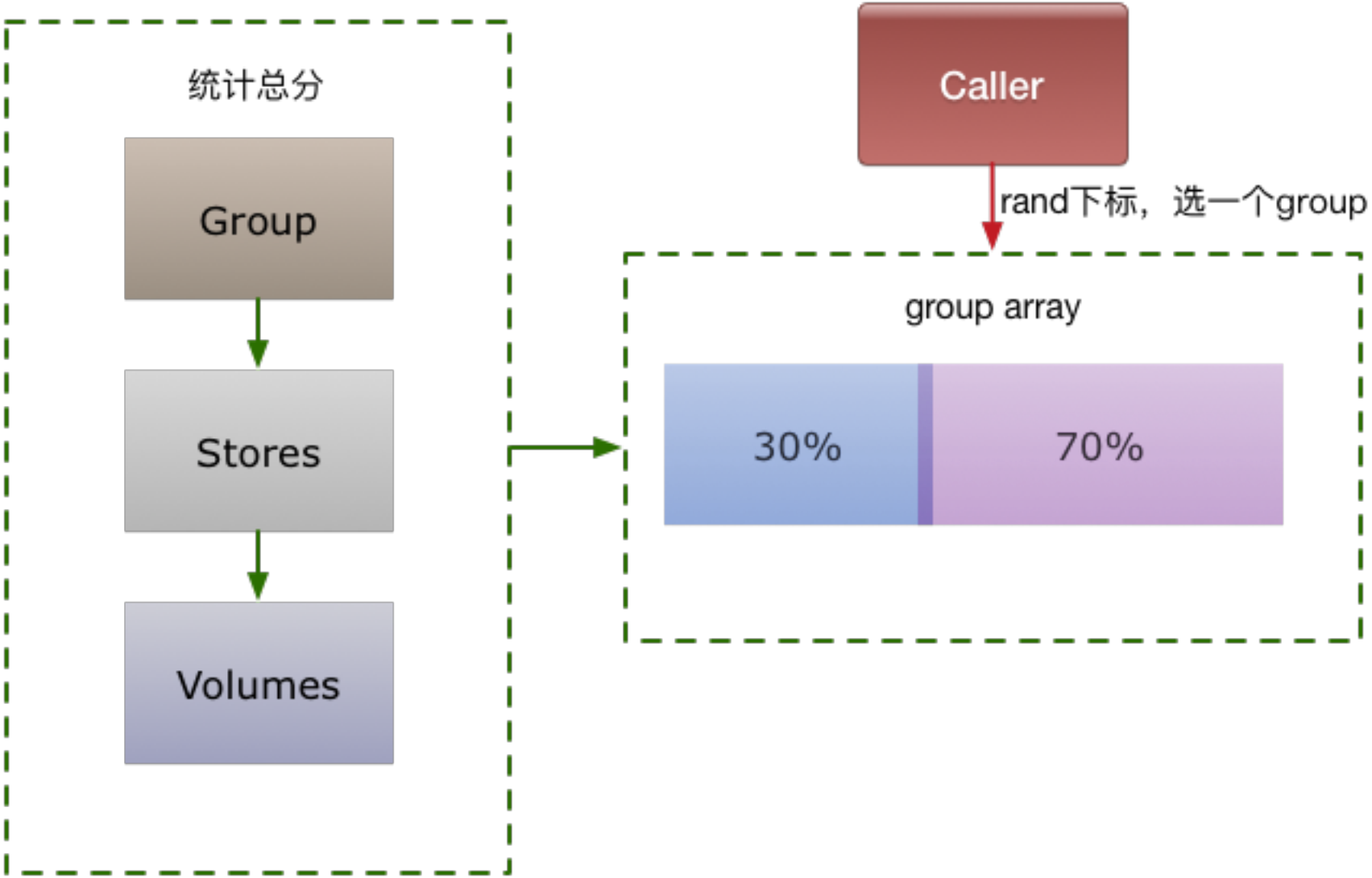
- 参考了Kafka的Consumer Group，对所有的Store节点进行拆分拆分；
- 使用zk的顺序和临时节点，启动时候创建进程退出时候销毁，pitchfork监听/pitchfork，一旦发生变化，重新进程任务分配。对store排序，再根据pitchfork节点总数分配的话，均分一下，每个节点就是一致的结果了；
- Golang map range具有随机性；



6 目录和调度模块：directory

- 获取所有Group以及所有归属的Store节点，每个Store下的所有Volume，统一进行一次打分计算，然后对Group进行权重分配，再选出Group进行写操作，完成存储调度均衡；
- 如果Group下的某个Store出现了故障（被Pitchfork发现），就会放弃整个Group，所以调度单元最终是Group；
- 选中Group以后，没做那么细的调度，直接随机选取一个Volume出来；

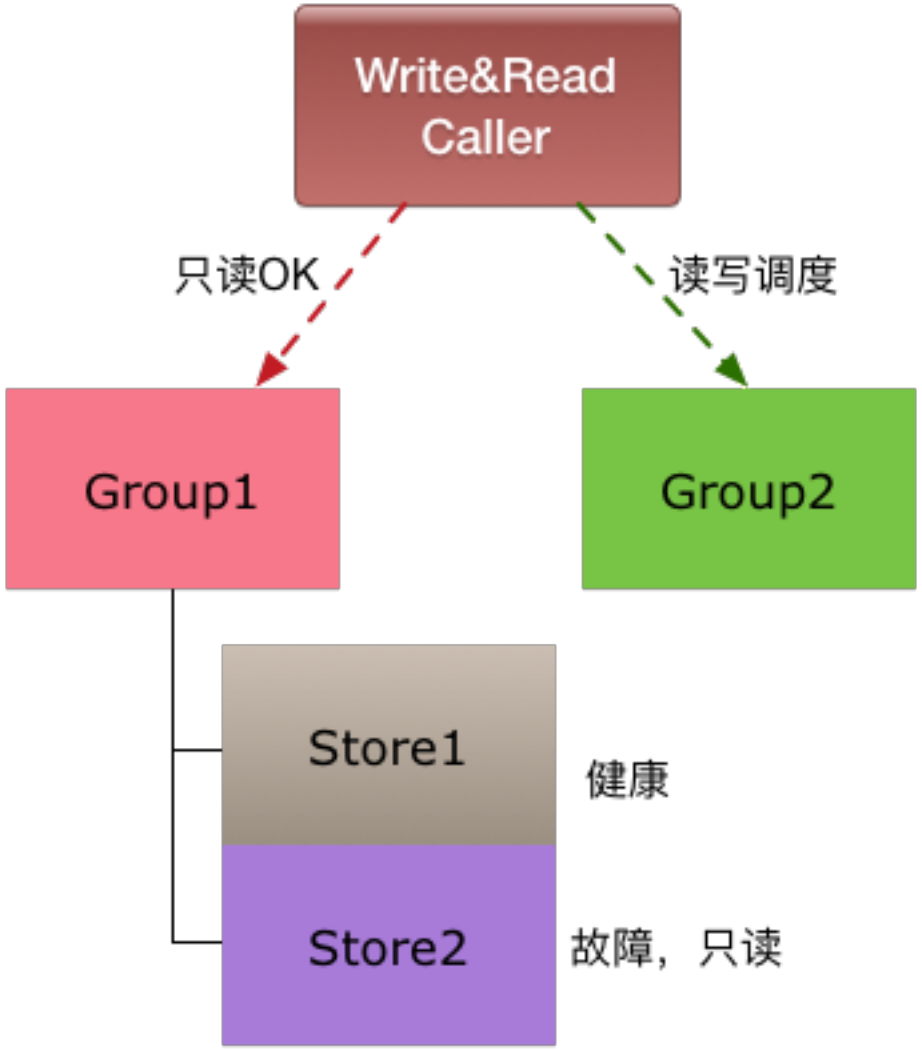
5 目录和调度模块：directory



6 目录和调度模块：高可用方案

- pitchfork来完成故障发现和统计上报的功能，最终提供给Directory使用，而Directory负责调度和故障转移，所以对Store几乎没有代码侵入，这是我的初衷；
- CAP理论的CP系统；
- CP还表示最大同性交友社区的隐射（bilibili干杯(゜-゜)つロ）；

6 目录和调度模块：高可用方案



7 总结 & QA

- bfs 从开发到上线大概 3-4 个月时间，是非常迅速的，我一直比较倾向简单的架构，牺牲一些比较不常用的功能，比如存储之间的 rebalance，更多的我们倾向把写比重不同来实现 rebalance 。

- 项目参考地址： https://github.com/Terry-Mao/bfs
- 开发人员：毛剑（review）、查普余。

- 也希望更多的同志给我们提供文档和建议，我们运维平台在内部测试通过以后会尽快同步到外网。

THANKS & QA

www.bilibili.com