1 FastDFS

1.1 Fastdfs 简介

— 国人在 mogileFS 基础上进行改进的 key-value 型文件系统，不支持 FUSE，提供比 mogileFS 更好的性能

— **轻量级**（移植性比较强，资源依赖性小？）的开源分布式文件系统

— **解决的问题**：1.大容量的文件存储 2.高并发的访问 3.文件存取时的负载均衡

— **特色**：实现了软件方式的 RAID;支持服务器在线扩充；支持相同的文件只存一份，节省了磁盘空间

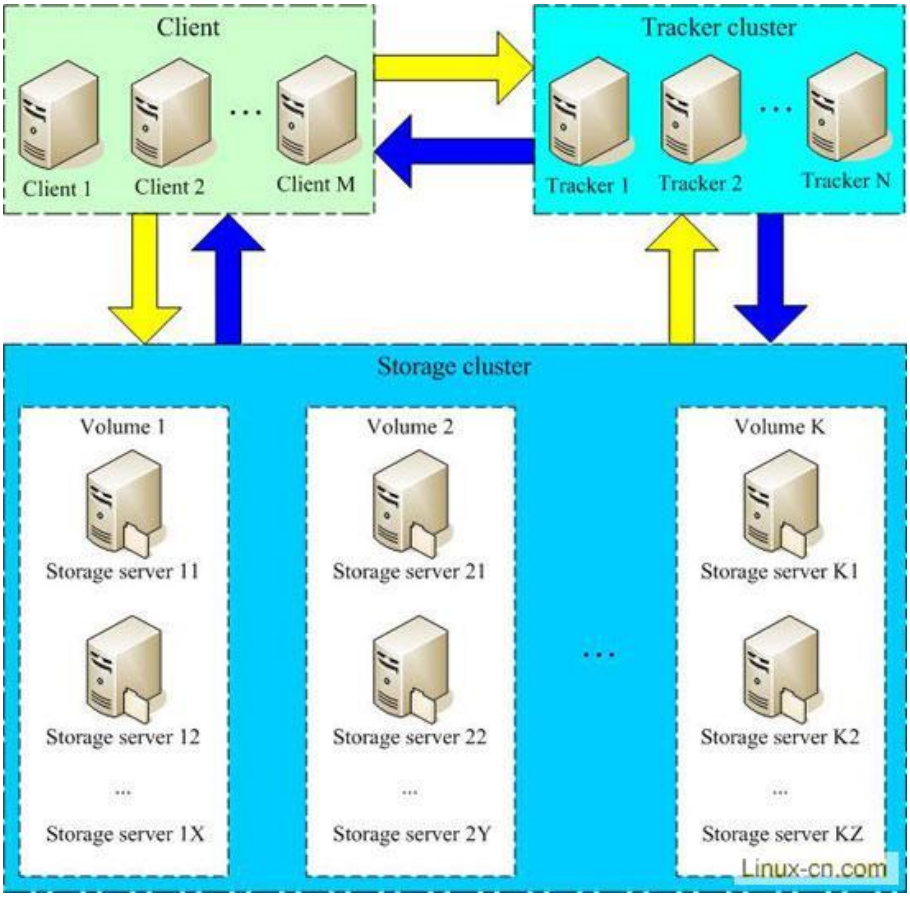
— 限制：只能通过 client api 方式访问，不支持 posix 方式访问

— 适合范围：大中型网站用来存储资源文件（如图片、文档、音频、视频、音频等），即以文件为载体的在线服务

— FastDFS 服务端有两个角色：跟踪器（）和存储节点（），跟踪器总要做调度工作，在访问上做负载均衡的作用，且跟踪器可用多台服务器进行均衡，这样可避免单点故障的发生。

— 通信协议：有专门协议，下载文件支持 HTTP

1.2 Fastdfs 系统结构图



1.3 FastDFS 和 mogileFS 的对比

1．FastDFS 完善程度较高，不需要二次开发即可直接使用；

2．和 MogileFs 相比，FastDFS 裁减了跟踪用的数据库，只有两个角色：tracker 和 storage。FastDFS 的架构既简　化了系统，同时也消除了性能瓶颈；

3．在系统中增加任何角色的服务器都很容易：增加 tracker 服务器时，只需要修改 storage 和 client 的配置文件（增加一行 tracker 配置）；增加 storage 服务器时，通常不需要修改任何配置文件，系统会自动将该卷中已有文件复制到该服务器；

4．FastDFS 比 MogileFS 更高效。表现在如下几个方面：

1）参见上面的第 2 点，FastDFS 和 MogileFS 相比，没有文件索引数据库，FastDFS 整体性能更高；

2）从采用的开发语言上看，FastDFS 比 MogileFS 更底层、更高

效。FastDFS 用 C 语言编写，代码量不到 2 万行，没有依赖其他开源软件或程序包，安装和部署特别简洁；而 MogileFS 用 perl 编写；

3）FastDFS 直接使用 socket 通信方式，相对于 MogileFS 的 HTTP 方式，效率更高。并且 FastDFS 使用 sendfile 传输文件，采用了内存零拷贝，系统开销更小，文件传输效率更高。

5. FastDFS 有着详细的设计和使用文档，而 MogileFS 的文档相对比较缺乏。

6. FastDFS 的日志记录非常详细，系统运行时发生的任何错误信息都会记录到日志文件中，当出现问题时方便管理员定位错误所在。

7. FastDFS 还对文件附加属性（即 meta data，如文件大小、图片宽度、高度等）进行存取，应用不需要使用数据库来存储这些信息。

8. FastDFS 从 V1.14 开始支持相同文件内容只保存一份，这样可以节省存储空间，提高文件访问性能。

2 MogileFS

2.1 Mogilefs 简介

— 一种分布式文件存储系统，可支持文件自动备份的功能，提供可用性和高可扩展性，用 Perl 语言编写，由于有依赖模块的问题，安装过程需要其他库和模块的支持，安装不算容易。

— key-value 型元文件系统，不支持 FUSE，应用程序访问它需要 API，主要在 web 领域处理海量小图片，效率高，

— **适用性**：不支持对一个文件的随机读写，只适合做一部分应用。比如图片服务，静态 html 服务，即文件写入后基本上那个不需要修改的应用。

2.2 Mogilefs 组成部分

2.2.1 数据库（MySQL）部分

mogdbsetup 程序可用来初始化数据库。数据库保存了 Mogilefs 的所有元数据，你可以单独拿数据库服务器来做，也可以跟其他程序跑在一起，数据库部分非常重要，类似邮件系统的认证中心那么重要，如果这儿挂了，那么整个 Mogilefs 将处于不可用状态。因此最好是 HA 结构。

2.2.2 存储节点

mogstored 程序的启动将使本机成为一个存储节点。启动时默认去读 /etc/mogilefs/mogstored.conf ，具体配置可以参考配置部分。mogstored 启动后，便可以通过 mogadm 增加这台机器到 cluster 中。一台机器可以只运行一个 mogstored 作为存储节点即可，也可以同时运行其他程序。

2.2.3 trackers（跟踪器）

mogilefsd 即 trackers 程序，类似 mogilefs 的 wiki 上介绍的，trackers 做了很多工作， Replication ， Deletion， Query， Reaper， Monitor 等等。mogadm,mogtool 的所有操作都要跟 trackers 打交道，Client 的一些操作也需要定义好 trackers，因此最好同时运行多个 trackers 来做负载均衡。trackers 也可以只运行在一台机器上，也可以跟其他程序运行在一起，只要你配置好他的配置文件即可，默认在/etc/mogilefs/mogilefsd.conf。

2.2.4 工具

主要就是 mogadm，mogtool 这两个工具了，用来在命令行下控制整个 mogilefs 系统以及查看状态等等。

2.2.5 Client

Client 实际上是一个 Perl 的 pm，可以写程序调用该 pm 来使用 mogilefs 系统，对整个系统进行读写操作。

2.3 Mogilefs 的特点

2.3.1 应用层——没有特殊的组件要求

2.3.2 无单点失败

MogileFS 启动的三个组件（存储节点、跟踪器、跟踪用的数据库），均可运行在多个 机器上，因此没有单点失败。（你也可以将跟踪器和存储节点运行在同一台机器上，这样你就没有必要用 4 台机器）推荐至少两台机器。

2.3.3 自动的文件复制

基于不同的文件“分类”，文件可以被自动的复制到多个有足够存储空间的存储节点上，这样可以满足这个“类别”的最少复制要求。比如你有一个图片网站，你可以设置原始的 JPEG 图片需要复制至少三份，但实际只有 1 or 2 分拷贝，如果丢失了数据，那么 Mogile 可以重新建立遗失的拷贝数。用这种办法，MogileFS (不做 RAID)可以节约磁盘，否则你将存储同样的拷贝多份，完全没有必要。

2.3.4 “比 RAID 好多了”

在一个非存储区域网络的 RAID（non-SAN RAID）的建立中，磁盘是冗余的，但主机不是，如果你整个机器坏了，那么文件也将不能访问。 MogileFS 在不同的机器之间进行文件复制，因此文件始终是可用的。

2.3.5 传输中立，无特殊协议

MogileFS 客户端可以通过 NFS 或 HTTP 来和 MogileFS 的存储节点来通信，但首先需要告知跟踪器一下。

2.3.6 简单的命名空间

文件通过一个给定的 key 来确定，是一个全局的命名空间。你可以自己生成多个命名空间，只要你愿意，但是这样可能在同一 MogileFS 中，会造成冲突 key。

2.3.7 不用共享任何东西

MogileFS 不需要依靠昂贵的 SAN 来共享磁盘，每个机器只用维护好自己

的磁盘。

2.3.8 不需要 RAID

在 MogileFS 中的磁盘可以是做了 RAID 的也可以是没有，如果是为了安全性着想的话 RAID 没有必要买了，因为 MogileFS 已经提供了。

2.3.9 不会碰到文件系统本身的不可知情况

在 MogileFS 中的存储节点的磁盘可以被格式化成多种格（ext3,reiserFS 等等）。MogilesFS 会做自己内部目录的哈希，所以它不会碰到文件系统本身的一些限制，比如一个目录中的最大文件数。你可以放心的使用。

3 HDFS

3.1 HDFS 简介

HDFS 全称是 Hadoop Distributed FileSystem。目前 HDFS 支持的使用接口除了 Java 的还有，Thrift、C、FUSE、WebDAV、HTTP 等。构成 HDFS 主要是 Namenode（master）和一系列的 Datanode（workers）。

3.2 特点和目标

3.2.1 硬件故障

硬件故障是计算机常见的问题。整个 HDFS 系统由数百或数千个存储着文件数据片断的服务器组成。实际上它里面有非常巨大的组成部分，每一个组成部分都会频繁地出现故障，这就意味着 HDFS 里的一些组成部分总是失效的，因此，故障的检测和自动快速恢复是 HDFS 一个核心的目标。

3.2.2 流式的数据访问

HDFS 使应用程序流式地访问它们的数据集。HDFS 是设计成适合批量处理的，而不是用户交互式的。所以其重视数据吞吐量，而不是数据访问的反应速度。*亦即 HDFS 是为以**流**的方式存取**大文件**而设计的。适用于几百 MB，*

*GB 以及 TB，并**写一次读多次**的场合。而对于低延时数据访问、大量小文件、同时写和任意的文件修改，则并不是十分适合。*

3.2.3 简单一致性模型

大部分的 HDFS 程序对文件操作需要的是一次写入，多次读取的。一个文件一旦创建、写入、关闭之后就不需要修改了。这个假定简化了数据一致的问题和高吞吐量的数据访问。

3.2.4 通信协议

所有的通信协议都是在 TCP/IP 协议之上的。一个客户端和明确的配置端口的名字节点建立连接之后，它和名字节点的协议是 ClientProtocal。数据节点和名字节点之间用 DatanodeProtocal。

3.3 基本概念

3.3.1 数据块(block)

- HDFS(Hadoop Distributed File System)默认的最基本的存储单位是 64M 的数据块。
- 和普通文件系统相同的是，HDFS 中的文件是被分成 64M 一块的数据块存储的。
- 不同于普通文件系统的是，HDFS 中，如果一个文件小于一个数据块的大小，并不占用

整个数据块存储空间。之所以将默认的 block 大小设置为 64MB 这么大，是因为 block-sized 对于文件定位很有帮助，同时大文件更使传输的时间远大于文件寻找的时间，这样可以最大化地减少文件定位的时间在整个文件获取总时间中的比例 。

3.3.2 元数据节点(Namenode)和数据节点(datanode)

3.3.2.1 这些结点的用途

- 元数据节点用来管理文件系统的命名空间

1)其将所有的文件和文件夹的元数据保存在一个文件系统树中。

2)这些信息也会在硬盘上保存成以下文件：命名空间镜像(namespace image)及修改日志(edit log)

3)其还保存了一个文件包括哪些数据块，分布在哪些数据节点上。然而这些信息并不存储在硬盘上，而是在系统启动的时候从数据节点收集而成的。

- 数据节点是文件系统中真正存储数据的地方。

1） 客户端(client)或者元数据信息(namenode)可以向数据节点请求写入或者读出数据块。

2） 其周期性的向元数据节点回报其存储的数据块信息。

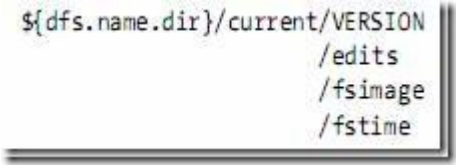
- 从元数据节点(secondary namenode)

1） 从元数据节点并不是元数据节点出现问题时候的备用节点，它和元数据节点负责不同的事情。

2） 其主要功能就是周期性将元数据节点的命名空间镜像文件和修改日志合并，以防日志文件过大。这点在下面会相信叙述。

3） 合并过后的命名空间镜像文件也在从元数据节点保存了一份，以防元数据节点失败的时候，可以恢复。

3.3.2.2 元数据节点文件夹结构



- VERSION 文件是 java properties 文件，保存了 HDFS 的版本号。
  - layoutVersion 是一个负整数，保存了 HDFS 的持续化在硬盘上的数据结构的格式版本号。

- namespaceID 是文件系统的唯一标识符，是在文件系统初次格式化时生成的。
- cTime 此处为 0
- storageType 表示此文件夹中保存的是元数据节点的数据结构。

```
namespaceID=1232737062
cTime=0
storageType=NAME_NODE
layoutVersion=-18
```

3.3.2.3 文件系统命名空间映像文件及修改日志

- 当文件系统客户端(client)进行写操作时，首先把它记录在修改日志中(edit log)
- 元数据节点在内存中保存了文件系统的元数据信息。在记录了修改日志后，元数据节点则修改内存中的数据结构。
- 每次的写操作成功之前，修改日志都会同步(sync)到文件系统。
- fsimage 文件，也即命名空间映像文件，是内存中的元数据在硬盘上的 checkpoint，它是一种序列化的格式，并不能够在硬盘上直接修改。
- 同数据的机制相似，当元数据节点失败时，则最新 checkpoint 的元数据信息从 fsimage 加载到内存中，然后逐一重新执行修改日志中的操作。
- 从元数据节点就是用来帮助元数据节点将内存中的元数据信息 checkpoint 到硬盘上的
- checkpoint 的过程如下：

(一) 从元数据节点通知元数据节点生成新的日志文件，以后的日志都写到新的日志文件中。

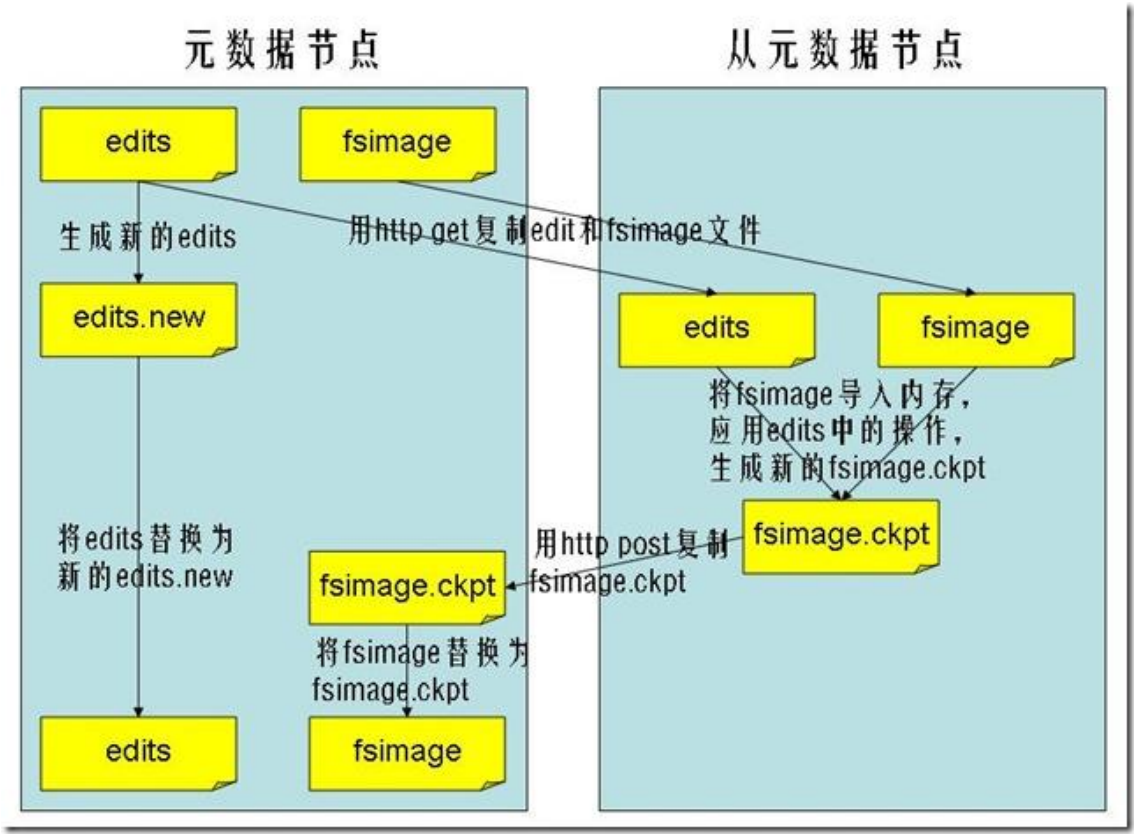
(二) 从元数据节点用 http get 从元数据节点获得 fsimage 文件及旧的日志文件。

(三) 从元数据节点将 fsimage 文件加载到内存中，并执行日志文件中的操作，然后生成新的 fsimage 文件。

(四) 从元数据节点奖新的 fsimage 文件用 http post 传回元数据节点

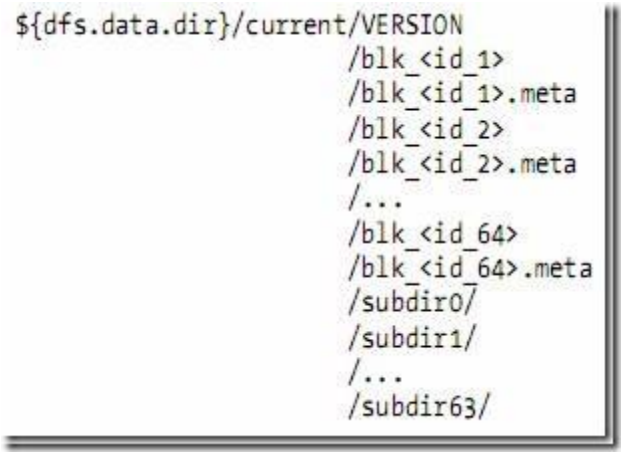
(五) 元数据节点可以将旧的 fsimage 文件及旧的日志文件，换为新的 fsimage 文件和新的日志文件(第一步生成的)，然后更新 fstime 文件，写入此次 checkpoint 的时间。

(六) 这样元数据节点中的 fsimage 文件保存了最新的 checkpoint 的元数据信息，日志文件也重新开始，不会变的很大了。



3.3.2.4 从元数据节点的目录结构

3.3.2.5 数据节点的目录结构



数据节点的 VERSION 文件格式如下：

```
namespaceID=1232737062
storageID=DS-1640411682-127.0.1.1-50010-1254997319480
cTime=0
storageType=DATA_NODE
layoutVersion=-18
```

blk_<id>保存的是 HDFS 的数据块，其中保存了具体的二进制数据。

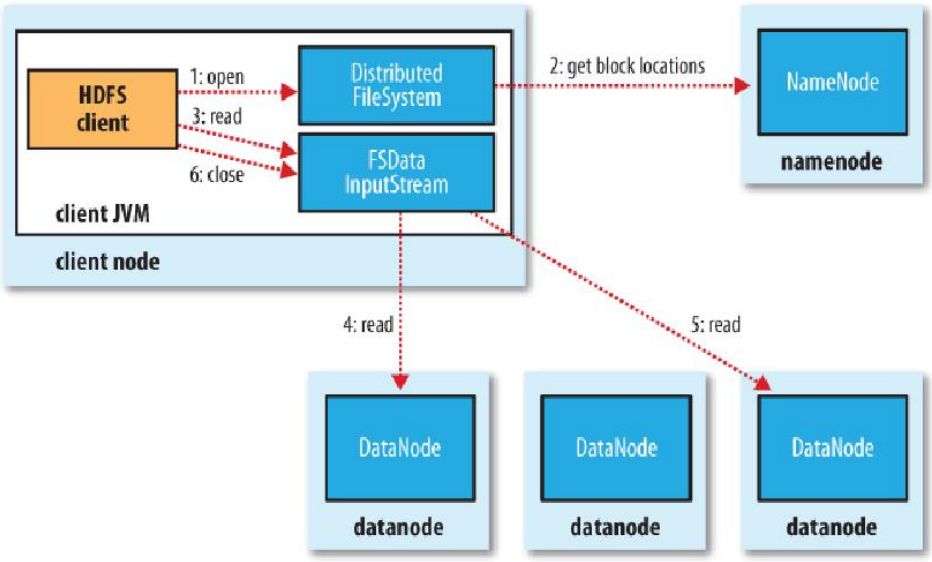
blk_<id>.meta 保存的是数据块的属性信息：版本信息，类型信息，和 checksum

当一个目录中的数据块到达一定数量的时候，则创建子文件夹来保存数据块及数据块属性

3.4 文件读写

3.4.1 读取文件

3.4.1.1 读取文件示意图



3.4.1.2 文件读取的过程

使用 HDFS 提供的客户端开发库，向远程的 Namenode 发起 RPC（Remote Procedure Call）请求；

Namenode 会视情况返回文件的部分或者全部 block 列表，对于每个 block，Namenode 都会返回有该 block 拷贝的 datanode 地址；

客户端开发库会选取离客户端最接近的 datanode 来读取 block；

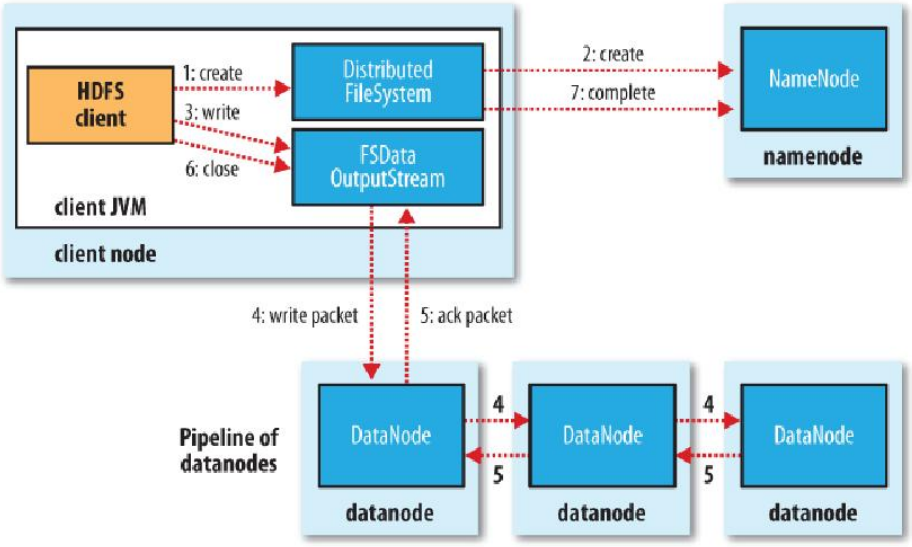
读取完当前 block 的数据后，关闭与当前的 datanode 连接，并为读取下一个 block 寻找最佳的 datanode；

当读完列表的 block 后，且文件读取还没有结束，客户端开发库会继续向 Namenode 获取下一批的 block 列表。

读取完一个 block 都会进行 checksum 验证，如果读取 datanode 时出现错误，客户端会通知 Namenode，然后再从下一个拥有该 block 拷贝的 datanode 继续读。

3.4.2 写入文件

3.4.2.1 写入文件示意图



3.4.2.2 写入文件的过程

1）使用 HDFS 提供的客户端开发库，向远程的 Namenode 发起 RPC 请求；

2）Namenode 会检查要创建的文件是否已经存在，创建者是否有权限进行操作，成功则会为文件创建一个记录，否则会让客户端抛出异常；

3）当客户端开始写入文件的时候，开发库会将文件切分成多个 packets，并在内部以"data queue"的形式管理这些 packets，并向 Namenode 申请新的 blocks，获取用来存储 replicas 的合适的 datanodes 列表，列表的大小根据在 Namenode 中对 replication 的设置而定。

4）开始以 pipeline（管道）的形式将 packet 写入所有的 replicas 中。开发库把 packet 以流的方式写入第一个 datanode，该 datanode 把该 packet 存储之后，再将其传递给在此 pipeline 中的下一个 datanode，直到最后一个 datanode，这种写数据的方式呈流水线的形式。

5）最后一个 datanode 成功存储之后会返回一个 ack packet，在 pipeline 里传递至客户端，在客户端的开发库内部维护着"ack queue"，成功收到 datanode 返回的 ack packet 后会从"ack queue"移除相应的 packet。

如果传输过程中，有某个 datanode 出现了故障，那么当前的 pipeline 会被关闭，出现故障的 datanode 会从当前的 pipeline 中移除，剩余的 block 会继续剩下的 datanode 中继续以 pipeline 的形式传输，同时 Namenode 会分配一个新

的 datanode，保持 replicas 设定的数量。

3.5 HDFS 不能提供的特点

3.5.1 低延时访问

HDFS 不太适合于那些要求低延时（数十毫秒）访问的应用程序，因为 HDFS 是设计用于大吞吐量数据的，这是以一定延时为代价的。HDFS 是单 Master 的，所有的对文件的请求都要经过它，当请求多时，肯定会有延时。当前，对于那些有低延时要求的应用程序，HBase 是一个更好的选择。现在 HBase 的版本是 0.20，相对于以前的版本，在性能上有了很大的提升，它的口号就是 goes real time。

使用缓存或多 master 设计可以降低 client 的数据请求压力，以减少延时。还有就是对 HDFS 系统内部的修改，这就得权衡大吞吐量与低延时了，HDFS 不是万能的银弹。

3.5.2 大量小文件

因为 Namenode 把文件系统的元数据放置在内存中，所以文件系统所能容纳的文件数目是由 Namenode 的内存大小来决定。一般来说，每一个文件、文件夹和 Block 需要占据 150 字节左右的空间，所以，如果你有 100 万个文件，每一个占据一个 Block，你就至少需要 300MB 内存。当前来说，数百万的文件还是可行的，当扩展到数十亿时，对于当前的硬件水平来说就没法实现了。还有一个问题就是，因为 Map task 的数量是由 splits 来决定的，所以用 MR 处理大量的小文件时，就会产生过多的 Map task，线程管理开销将会增加作业时间。举个例子，处理 10000M 的文件，若每个 split 为 1M，那就会有 10000 个 Map tasks，会有很大的线程开销；若每个 split 为 100M，则只有 100 个 Map tasks，每个 Map task 将会有更多的事情做，而线程的管理开销也将减小很多。

要想让 HDFS 能处理好小文件，有不少方法：

1、利用 SequenceFile、MapFile、Har 等方式归档小文件，这个方法的原理就是把小文件归档起来管理，HBase 就是基于此的。对于这种方法，如果

想找回原来的小文件内容，那就必须得知道与归档文件的映射关系。

2、横向扩展，一个 Hadoop 集群能管理的小文件有限，那就把几个 Hadoop 集群拖在一个虚拟服务器后面，形成一个大的 Hadoop 集群。google 也是这么干过的。

3、多 Master 设计，这个作用显而易见了。正在研发中的 GFS II 也要改为分布式多 Master 设计，还支持 Master 的 Failover，而且 Block 大小改为 1M，有意要调优处理小文件啊。

附带个 Alibaba DFS 的设计，也是多 Master 设计，它把 Metadata 的映射存储和管理分开了，由多个 Metadata 存储节点和一个查询 Master 节点组成。

3.5.3 多用户写，任意文件修改

目前 Hadoop 只支持单用户写，不支持并发多用户写。可以使用 Append 操作在文件的末尾添加数据，但不支持在文件的任意位置进行修改。这些特性可能会在将来的版本中加入，但是这些特性的加入将会降低 Hadoop 的效率。利用 Chubby、ZooKeeper 之类的分布式协调服务来解决一致性问题。

4 TFS

4.1 TFS 简介

Taobao File System（TFS）作为淘宝内部使用的分布式文件系统，是一个高可扩展、高可用、高性能、面向互联网服务的分布式文件系统，其设计目标是“支持海量的非结构化数据”。针对**海量小文件的随机读写访问**性能做了特殊优化，承载着淘宝主站所有图片、商品描述等数据存储。

4.2 TFS 系统的基本情况

1.完全扁平化的数据组织结构，抛弃了传统文件系统的目录结构。

2.在块设备基础上建立自有的文件系统，减少 EXT3 等文件系统数据碎片带来的性能损耗。

3.单进程管理单块磁盘的方式，摒除 RAID5 机制。

4.带有 HA 机制的中央控制节点，在安全稳定和性能复杂度之间取得平衡。

5.尽量缩减元数据大小，将元数据全部加载入内存，提升访问速度。

6.跨机架和 IDC 的负载均衡和冗余安全策略。

7.完全平滑扩容。

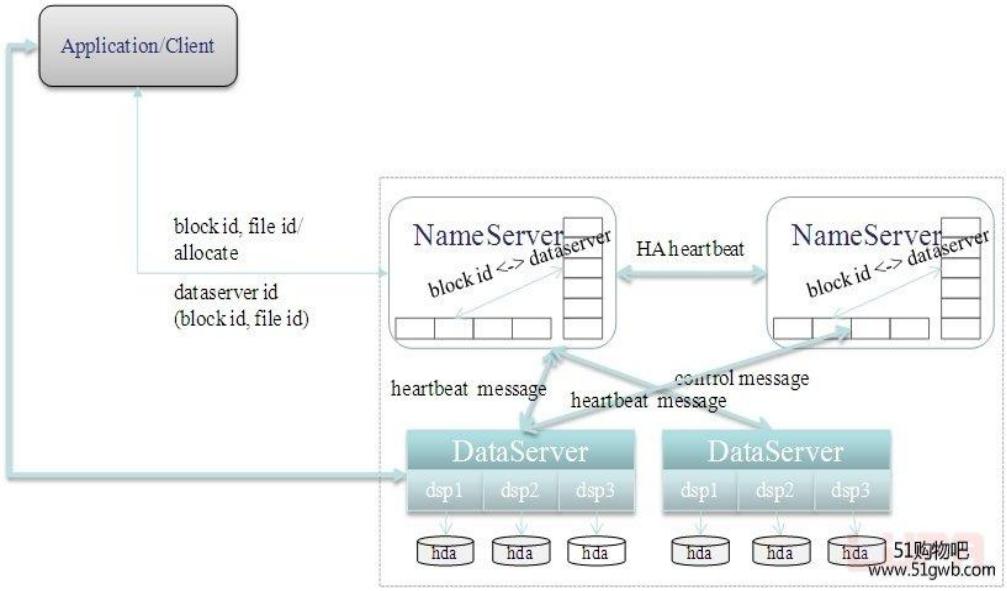
4.3 应用规模

达到“数百台 PCServer，PB 级数据量，百亿数据级别”

4.4 性能参数

TFS 在淘宝的部署环境中前端有两层缓冲，到达 TFS 系统的请求非常离散，所以 TFS 内部是没有任何数据的内存缓冲的，包括传统文件系统的内存缓冲也不存在.....基本上可以达到单块磁盘随机 IOPS（即 I/O per second）理论最大值的 60%左右，整机的输出随盘数增加而线性增加。

4.5 TFS 的逻辑架构图



4.5.1 下载 (129.26 KB) 2010-9-29 22:39

4.5.2 结合架构图做了进一步说明

1.TFS 尚未对最终用户提供传统文件系统 API，需要通过 TFSClient 进行接口访问，现有 JAVA、JNI、C、PHP 的客户端

2.TFS 的 NameServer 作为中心控制节点，监控所有数据节点的运行状况，负责读写调度的负载均衡，同时管理一级元数据用来帮助客户端定位需要访问的数据节点

3.TFS 的 DataServer 作为数据节点，负责数据实际发生的负载均衡和数据冗余，同时管理二级元数据帮助客户端获取真实的业务数据。

4.6 TFS 的不足之处

TFS 与目前一些主流的开源分布式文件系统设计思想是相似的，如 HDFS, MFS, KFS, Sector。其高可扩展、高可用性是很好的，然而也***存在一定不足***，如通用性、用户接口、性能等方面。

4.6.1 通用性方面。

TFS 目前只支持小文件的应用，大文件应用是不支持的。对小图片、网页等几十 KB 内的数据存储非常适用，但对视频点播 VOD、文件下载等应用暂时无法适用。

4.6.2 性能方面。

Client 写文件是同步处理的，需要等所有 dataserver 写成功后才能返回，这很是影响性能。

4.6.3 用户接口。

TFS 没有提供 POSIX 接口，提供的 API 也与标准接口不一致。另外，TFS

有自己的文件命名规则，如果用户使用自定义的文件名，则需要自己维护文件名与 TFS 文件名之间的映射关系。

4.6.4 代码方面。

使用了 C++实现，感觉相对臃肿一点，如果用纯 C 实现应该会简洁不少。代码注释基本没有，代码质量也不是很好。

4.6.5 技术文档。

官方有一些文档，但显然非常不够深入和全面。

4.6.6 小文件优化。

官方称针对海量小文件的随机读写访问性能做了特殊优化，现在只看到把众多小文件存放与一个 Block 中，这与 Squid 中的 COSS 原理相似。其他特殊优化措施未知，LOFS(Lost of small files)是个难点问题。

5 MooseFS（简称 MFS）

5.1 MFS 简介

MFS 是一款网络分布式文件系统。它把数据分散在多台服务器上，但对于用户来讲，看到的只是一个源。MFS 也像其他类 unix 文件系统一样，包含了层级结构（目录树），存储着文 件属性（权限，最后访问和修改时间），可以创建特殊的文件（块设备，字符设备，管道，套接字），符号链接，硬链接。

5.2 MFS 的优点

1.免费开源

2.通用文件系统，不需要修改上层应用就可以使用

3.可以在线扩容，体系架构可伸缩性极强

4.部署简单

5.体系架构高可用，所有组件无单点故障

6.文件对象高可用，可任意设置文件的冗余程度，而绝对不会影响读或写的性能，只会加速。

7.提供 windows 回收站的功能

8.提供类似 java 语言的垃圾回收（GC）

9.提供 netapp,emc,ibm 等商业存储的 snapshot 特性。

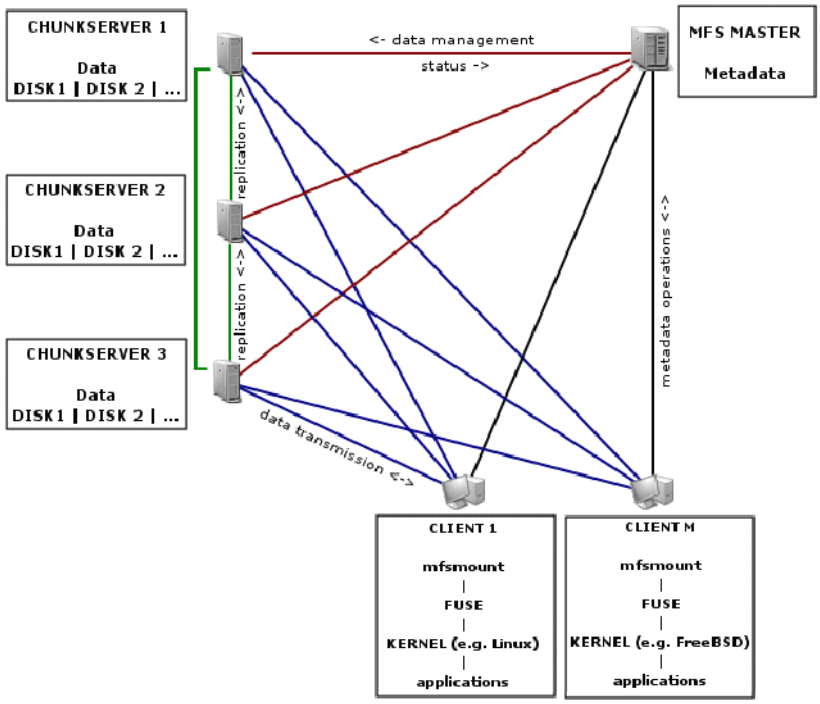
10.GFS 的一个 c 实现

11.提供 web gui 监控接口

12.提高随机读或写的效率

13.提高海量小文件的读写效率

5.3 网络示意图(如下)



5.4 MFS 文件系统结构

5.4.1 包含的 4 种角色

5.4.1.1 管理服务器 managing server (master)

负责各个数据存储服务器的管理, 文件读写调度, 文件空间回收以及恢复. 多

节点拷贝

5.4.1.2 元数据日志服务器 Metalogger serve（Metalogger）

负责备份 master 服务器的变化日志文件，文件类型为 changelog_ml.*.mfs，以便于在 master server 出问题的时候接替其进行工作。元数据存储在 Master 的内存中，同时会保存一份在硬盘上（作为临时更新的二进制文件和立即更新的增量日志方式）

5.4.1.3 数据存储服务器 data servers (chunkservers)

负责连接管理服务器，听从管理服务器调度，提供存储空间，并为客户提供数据传输．数据文件被分成 64Mb 大小的块，每个块被分散的存储在块服务器的硬盘上，同时块服务器上还会存储其他块服务器上块文件的副本。

5.4.1.4 客户端 client computers

通过 fuse 内核接口挂接远程管理服务器上所管理的数据存储服务器，.看起来共享的文件系统和本地 unix 文件系统使用一样的效果．客户端只需要 mount 上 MFS 就可像操作其他文件系统的文件一样操作 MFS 中的文件了。操作系统的内核把对文件的操作传递给 FUSE 模块，这个模块用来和 mfsmount 进程进行通信。mfsmount 进程后续通过网络和管理服务器和数据块服务器进行通信。整个过程对用户来讲是透明的。

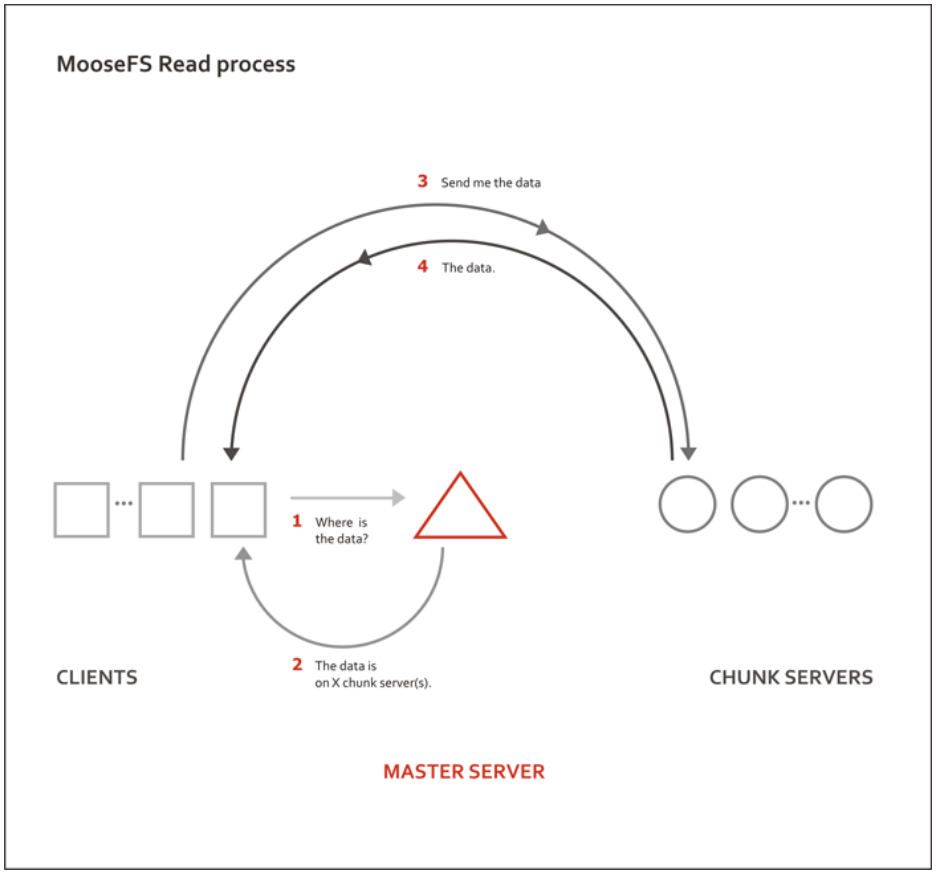
5.4.2 四种角色的协作过程

在对所有元数据文件（文件创建，文件删除，读文件夹，读取和更改属性，改变文件大小等等涉及到在 MFSMETA 上的特殊文件）进行操作的过程中，mfsmount 和管理服务器建立通信，然后开始读取和写入数据。数据发送到所有数据服务器中有相关文件块的一台上。在完成写操作之后，管理服务器收到文件长度和最后修改时间的更新信息。而且，数据服务器之间进行复制通信，保证每个块在不同的块服务器上都有拷贝。因为文件块存在多个拷贝，所以，任何一台数据服务器不可用都是不会影响到文件的正常访问的。整体来看 moosfs，他的

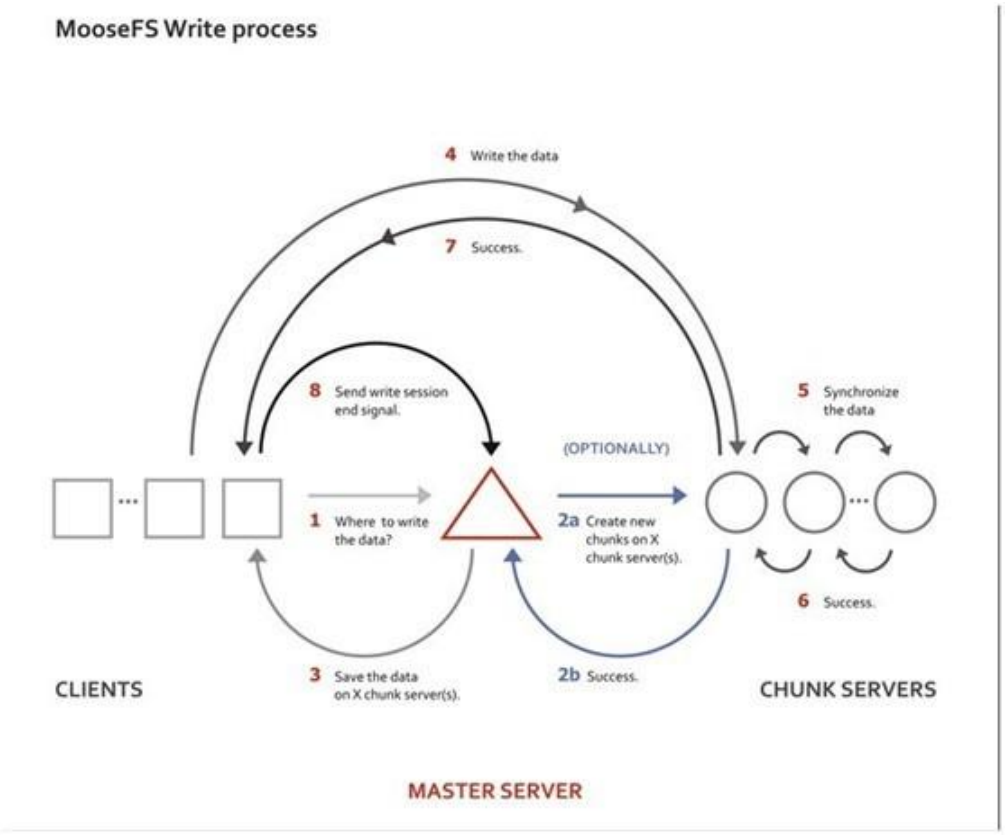
设计理念还是很符合 gfs 的，从架构图来看，整个系统实现起来还是很容易的。不过有一点值得注意的还是，对于 master 主机来说，这个是一个单点，会存在隐患，在正式环境应用的时候，如何解决这里，是个关键。

5.5 MFS 读写进程

5.5.1 MFS 读进程



5.5.2 MFS 写进程



PS: MFS 提供文件系统级回收站的配置，这个回收站在整个文件系统工作。那样如果用户删除一个文件，这个文件可以一直存在回收站中，只要管理员想留着它。回收站中的文件会在一段配置时间之后自动清理。

6 KFS

6.1 KFS 简介

KFS（KOSMOS DISTRIBUTED FILE SYSTEM），一个类似 GFS、Hadoop 中 HDFS 的一个开源的分布式文件系统。可以应用在诸如图片存储、搜索引擎、网格计算、数据挖掘这样需要处理大数据量的网络应用中。与 hadoop 集成得也比较好，这样可以充分利用了 hadoop 一些现成的功能，基于 C++。

6.2 KFS 的特性

6.2.1 1.自动存储扩充

添加新的 chunckserver，系统自动感知

6.2.2 2.有效性

复制机制保证文件有效性，一般文件会被以三种方式存储，当其中一个 chunkserver 出现错误的时候，不会影响数据的读取；

6.2.3 3.文件复制粒度

可以配置文件复制的粒度，最大可以被复制 64 份

6.2.4 4.还原复制

当其中一个 Chunckserver 出现故障的时候，Metaserver 会强制使用其他的 chunckserver

6.2.5 5.负载平衡

系统周期地检查 chunkservers 的磁盘利用，并重新平衡 chunkservers 的磁盘利用，HDFS 现在还没有支持

6.2.6 6.数据完整性

当要读取数据时检查数据的完整性，如果检验出错使用另外的备份覆盖当前的数据

6.2.7 7.文件写入

当一个应用程序创建了一个文件，这个文件名会被立刻写入文件系统，但为了性能，写入的数据会被缓存在 kfs 客户端. 并且周期性的从缓存中把数据更 新到 chunkserver 中。当然，应用程序也可以强制把数据更新到服务器上。一旦数据被更新到服务器，就可以被有效的读取了。（我怎么能知道这个文件什么时候可以读取了呢？）

6.2.8 8.契约

使用契约来保证 Client 缓存的数据和文件系统中的文件保持一致性

6.2.9 9.支持 FUSE

在 linux 系统下，可以通过 Fuse 映射一个文件夹，从而可以很方便的读取 kfs 的文件

6.2.10 支持 C++,Java,Python 方式的调用

6.2.11 提供了丰富的工具程序

如 kfsshell,cp2kfs 等

6.2.12 提供了启动和停止服务的脚本

6.3 KFS 高级特性

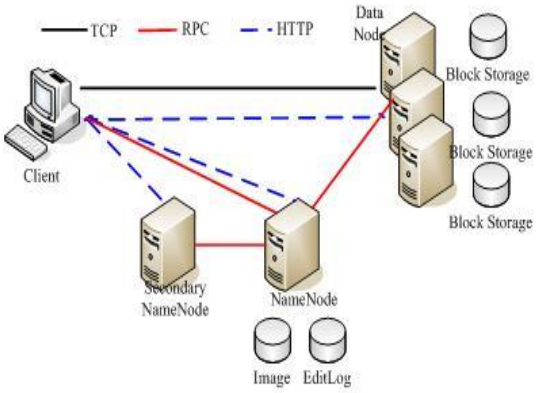
1.支持同一文件多次写入和 Append，不过不能在文件中间插入数据。（HDFS 支持一次写入多次读取，不支持 Append）

2.文件及时生效，当应用程序创建一个文件时，文件名在系统马上有效。（HDFS 文件只当输入流关闭时才在系统中有效，因此，如果应用程序在关闭前出现异常导致没有关闭输入流，数据将会丢失。)

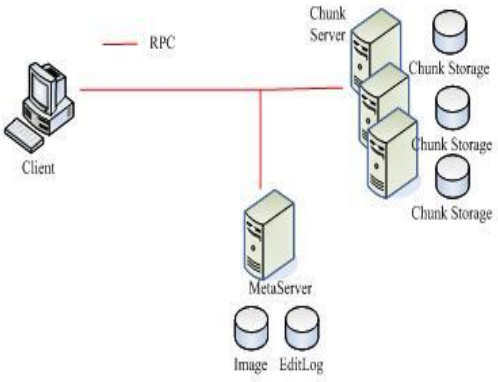
这一点好像也不是很好，如果输入流中断，在 kfs 里会留下一个错误的文件，当读取时会出现错误，好像也没有太大的意义。

6.4 KFS 与 HDFS 的比较

6.4.1 体系结构图的比较



HDFS 体系结构



KFS 体系结构

6.4.2 特点的比较

两者都是 GFS 的开源实现，而 HDFS 是 Hadoop 的子项目，用 Java 实现，为 Hadoop 上层应用提供高吞吐量的可扩展的大文件存储服务。KFS 是一个高性能的分布式文件系统，主要针对网络规模的应用，例如存储日志数据，Map/Reduce 数据等. 用 C++实现。它们的元数据管理采用集中式方式实现，数据实体先分片然后分布式存储。

7 Ceph

7.1 Ceph 的目标

1. 可轻松扩展到数 PB 容量

2. 对多种工作负载的高性能（每秒输入/输出操作[IOPS]和带宽）

3.高可靠性

不幸的是，这些目标之间会互相竞争（例如，可扩展性会降低或者抑制性能或者影响可靠性）。Ceph 开发了一些非常有趣的概念（例如，动态元数据分区，数据分布和复制），这些概念在本文中只进行简短地探讨。Ceph 的设计还包括保护单一点故障的容错功能，它假设大规模（PB 级存储）存储故障是常见现象而不是例外情况。最后，它的设计并没有假设某种特殊工作负载，但是包括适应变化的工作负载，提供最佳性能的能力。它利用 POSIX 的兼容性完成所有这些任务，允许它对当前依赖 POSIX 语义（通过以 Ceph 为目标的改进）的应用

进行透明的部署。最后，Ceph 是开源分布式存储，也是主线 Linux 内核（2.6.34）的一部分。

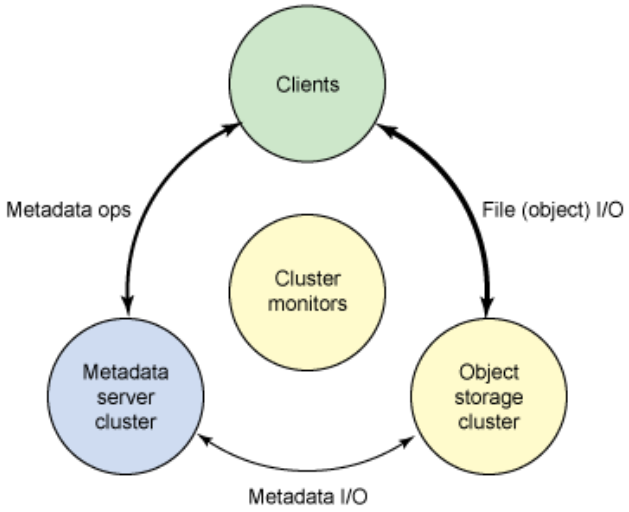
7.2 Ceph 生态系统

7.2.1 可以大致划分为四部分

1. 客户端（数据用户），
2. 元数据服务器（缓存和同步分布式元数据），
3. 一个对象存储集群（将数据和元数据作为对象存储，执行其他关键职能）
4. 集群监视器（执行监视功能）。

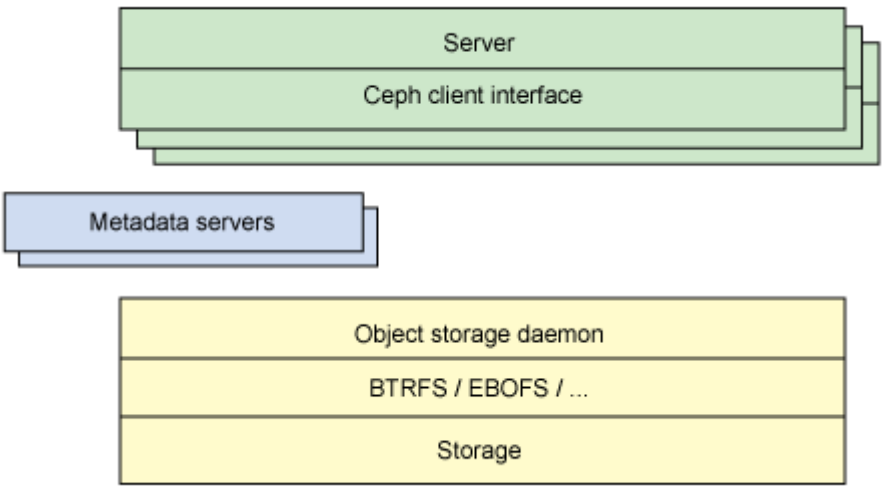
7.2.2 Ceph 生态系统的概念架构

7.2.2.1 架构视图 1



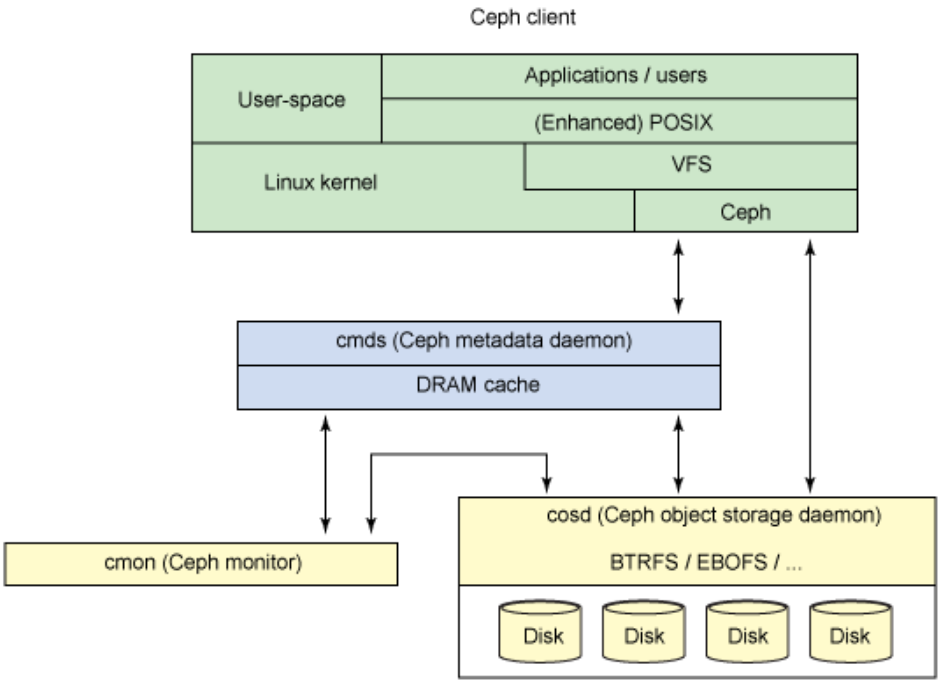
如上图 1 所示，客户使用元数据服务器，执行元数据操作（来确定数据位置）。元数据服务器管理数据位置，以及在何处存储新数据。值得注意的是，元数据存储在一个存储集群（标为 “元数据 I/O”）。实际的文件 I/O 发生在客户和对象存储集群之间。这样一来，更高层次的 POSIX 功能（例如，打开、关闭、重命名）就由元数据服务器管理，不过 POSIX 功能（例如读和写）则直接由对象存储集群管理。

7.2.2.2 架构视图 2



一系列服务器通过一个客户界面访问 Ceph 生态系统，这就明白了元数据服务器和对象级存储器之间的关系。分布式存储系统可以在一些层中查看，包括一个存储设备的格式（Extent and B-tree-based Object File System [EBOFS] 或者一个备选），还有一个设计用于管理数据复制，故障检测，恢复，以及随后的数据迁移的覆盖管理层，叫做 Reliable Autonomic Distributed Object Storage（RADOS）。最后，监视器用于识别组件故障，包括随后的通知。

7.2.3 Ceph 组件



Ceph 和传统的文件系统之间的重要差异之一就是，它将智能都用在了生态环境而不是文件系统本身。

图 3 显示了一个简单的 Ceph 生态系统。Ceph Client 是 Ceph 文件系统的用户。Ceph Metadata Daemon 提供了元数据服务器，而 Ceph Object Storage Daemon 提供了实际存储（对数据和元数据两者）。最后，Ceph Monitor 提供了集群管理。要注意的是，Ceph 客户，对象存储端点，元数据服务器（根据文件系统的容量）可以有许多，而且至少有一对冗余的监视器。

7.2.3.1 Ceph 客户端

内核或用户空间

早期版本的 Ceph 利用在 User SpacE（FUSE）的 Filesystems，它把文件系统推入到用户空间，还可以很大程度上简化其开发。但是今天，Ceph 已经被集成到主线内核，使其更快速，因为用户空间上下文交换机对文件系统 I/O 已经不再需要。

因为 Linux 显示文件系统的一个公共界面（通过虚拟文件系统交换机[VFS]），Ceph 的用户透视图就是透明的。管理员的透视图肯定是不同的，考虑到很多服务器会包含存储系统这一潜在因素。从用户的角度看，他们访问大容量的存储系统，却不知道下面聚合成一个大容量的存储池的元数据服务器，监视器，还有独立的对象存储设备。用户只是简单地看到一个安装点，在这点上可以执行标准文件 I/O

Ceph 文件系统 — 或者至少是客户端接口 — 在 Linux 内核中实现。值得注意的是，在大多数文件系统中，所有的控制和智能在内核的文件系统源本身中执行。但是，在 Ceph 中，文件系统的智能分布在节点上，这简化了客户端接口，并为 Ceph 提供了大规模（甚至动态）扩展能力。

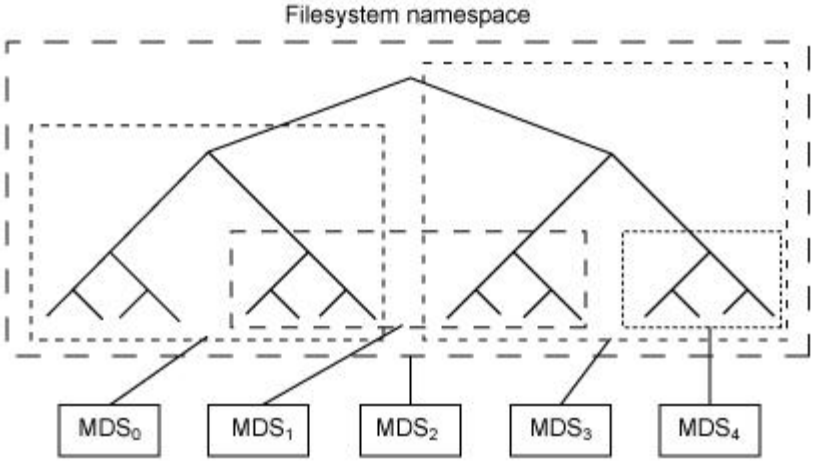
Ceph 使用一个有趣的备选，而不是依赖分配列表（将磁盘上的块映射到指定文件的元数据）。Linux 透视图中的一个文件会分配到一个来自元数据服务器的 inode number（INO），对于文件这是一个唯一的标识符。然后文件被推入一些对象中（根据文件的大小）。使用 INO 和 object number（ONO），每个对象都分配到一个对象 ID（OID）。在 OID 上使用一个简单的哈希，每个对象

都被分配到一个放置组。放置组（标识为 PGID）是一个对象的概念容器。最后，放置组到对象存储设备的映射是一个伪随机映射，使用一个叫做 Controlled Replication Under Scalable Hashing（CRUSH）的算法。这样一来，放置组（以及副本）到存储设备的映射就不用依赖任何元数据，而是依赖一个伪随机的映射函数。这种操作是理想的，因为它把存储的开销最小化，简化了分配和数据查询。

分配的最后组件是集群映射。集群映射 是设备的有效表示，显示了存储集群。有了 PGID 和集群映射，您就可以定位任何对象。

7.2.3.2 Ceph 元数据服务器

元数据服务器（cmds）的工作就是管理文件系统的名称空间。虽然元数据和数据两者都存储在对象存储集群，但两者分别管理，支持可扩展性。事实上，元数据在一个元数据服务器集群上被进一步拆分，元数据服务器能够自适应地复制和分配名称空间，避免出现热点。如图 4 所示，元数据服务器管理名称空间部分，可以（为冗余和性能）进行重叠。元数据服务器到名称空间的映射在 Ceph 中使用动态子树逻辑分区执行，它允许 Ceph 对变化的工作负载进行调整（在元数据服务器之间迁移名称空间）同时保留性能的位置。



但是因为每个元数据服务器只是简单地管理客户端人口的名称空间，它的主要应用就是一个智能元数据缓存（因为实际的元数据最终存储在对象存储集群中）。进行写操作的元数据被缓存在一个短期的日志中，它最终还是被推入物理存储器中。这个动作允许元数据服务器将最近的元数据回馈给客户（这在元数据操作中很常见）。这个日志对故障恢复也很有用：如果元数据服务器发生故障，

它的日志就会被重放，保证元数据安全存储在磁盘上。

元数据服务器管理 inode 空间，将文件名转变为元数据。元数据服务器将文件名转变为索引节点，文件大小，和 Ceph 客户端用于文件 I/O 的分段数据（布局）。

Ceph 监视器

Ceph 包含实施集群映射管理的监视器，但是故障管理的一些要素是在对象存储本身中执行的。当对象存储设备发生故障或者新设备添加时，监视器就检测和维护一个有效的集群映射。这个功能按一种分布的方式执行，这种方式中映射升级可以和当前的流量通信。Ceph 使用 Paxos，它是一系列分布式共识算法。

7.2.3.3 Ceph 对象存储

和传统的对象存储类似，Ceph 存储节点不仅包括存储，还包括智能。传统的驱动是只响应来自启动者的命令的简单目标。但是对象存储设备是智能设备，它能作为目标和启动者，支持与其他对象存储设备的通信和合作。

从存储角度来看，Ceph 对象存储设备执行从对象到块的映射（在客户端的文件系统层中常常执行的任务）。这个动作允许本地实体以最佳方式决定怎样存储一个对象。Ceph 的早期版本在一个名为 EBOFS 的本地存储器上实现一个自定义低级文件系统。这个系统实现一个到底层存储的非标准接口，这个底层存储已针对对象语义和其他特性（例如对磁盘提交的异步通知）调优。今天，B-tree 文件系统（BTRFS）可以被用于存储节点，它已经实现了部分必要功能（例如嵌入式完整性）。

因为 Ceph 客户实现 CRUSH，而且对磁盘上的文件映射块一无所知，下面的存储设备就能安全地管理对象到块的映射。这允许存储节点复制数据（当发现一个设备出现故障时）。分配故障恢复也允许存储系统扩展，因为故障检测和恢复跨生态系统分配。Ceph 称其为 RADOS（见 图 3）。

7.2.3.4 其他有趣功能

如果文件系统的动态和自适应特性不够，Ceph 还执行一些用户可视的有趣功能。用户可以创建快照，例如，在 Ceph 的任何子目录上（包括所有内容）。

文件和容量计算可以在子目录级别上执行，它报告一个给定子目录（以及其包含的内容）的存储大小和文件数量。

7.3 Ceph 的地位和未来

虽然 Ceph 现在被集成在主线 Linux 内核中，但只是标识为实验性的。在这种状态下的文件系统对测试是有用的，但是对生产环境没有做好准备。但是考虑到 Ceph 加入到 Linux 内核的行列，还有其创建人想继续研发的动机，不久之后它应该就能用于解决您的海量存储需要了。

7.4 其他分布式文件系统

Ceph 在分布式文件系统空间中并不是唯一的，但它在管理大容量存储生态环境的方法上是独一无二的。分布式文件系统的其他例子包括 Google File System（GFS），General Parallel File System（GPFS），还有 Lustre，这只提到了一部分。Ceph 背后的想法为分布式文件系统提供了一个有趣的未来，因为海量级别存储导致了海量存储问题的唯一挑战。

7.5 展望未来

Ceph 不只是一个文件系统，还是一个有企业级功能的对象存储生态环境。

补充：Ceph 的主要目标是设计成基于 POSIX 的没有单点故障的分布式文件系统，使数据能容错和无缝的复制。