lightFS 系统设计说明书

易剑 2013/6/8

目录

修订记录

修改人 （格式：中文全名）	修改日期 （格式：2013/6/8）	修改记录 （格式：要求描述增修减了什么内容）
易剑	2013/6/8	创建
曹彪	2013/6/12	新增目录树设计

1 前言

1.1 什么是 lightFS？

lightFS 是一个设计理念完全不同于 GFS/HDFS 的轻量级分布式容灾文件系统。它完美的利用了 CAP 原理，通过牺牲一致性，实现了超高的可用性和可靠性。

1.2 文档目的

编写本文的目的，是为进一步将脑海中的思路沉淀下来，以便做进一步优化。同时，用以指导其他开发人员去实现 lightFS。

1.3 参考资料

《lightFS-牺牲强一致性获得超高可用性和可靠性的磁盘级轻量分布式文件系统.ppt》

1.4 技术交流

1.4.1 技术博客

http://aquester.cublog.cn

1.4.2 开源项目

http://code.google.com/p/mooon，lightFS 将作为 mooon 的一个子项目存在。

1.4.3 技术论坛

http://bbs.hadoopor.com

2 应用场景

- 无超大文件，主要几十兆或几百兆字节的文件，不合适存储 1M 以下的小文件，和大量超 2G 以上的文件；
- 不需要强一致性；
- 觉得 HDFS 等太重，部署复杂；
- 适合存储各种用于数据分析的日志文件。

3 设计目标

- 文件自动容灾，对外透明；
- 超高的可用性和可靠性，近完美的 A 和 P（相对 CAP 原理）；
- 最终的一致性；
- 可容忍一半的节点故障；
- 兼容 sshd，可通过 scp 上传和下载文件。

4 设计指标

4.1 集群大小

目标集群为 100~1000 个节点，最佳性能在 100 个节点左右。

4.2 存储容量

假设平均文件路径为 1K，则 1000 个文件为 1M，100 万文件为 1G，10 万文件为 100M，设计支持容量为 10 万个文件，文件最大为 4G。

4.3 设计约束

- 不支持文件分块；
- 单个文件最大不能超过 2G；
- 不提供强一致性，但保证最终一致性。

5 接口设计

采用和 POSIX 兼容接口设计：

long **open**(const char *pathname, int flags, mode_t mode);
long **open**(const char *pathname, int flags);
int **close**(long fd);
ssize_t **write**(long fd, const void *buf, size_t count);
ssize_t **read**(long fd, void *buf, size_t count);
int **remove**(const char *pathname);

注意和 POSIX 标准稍有不同：fd 的数据类型由 int 变成了 long（intptr_t），但 long 是兼容 int 的。

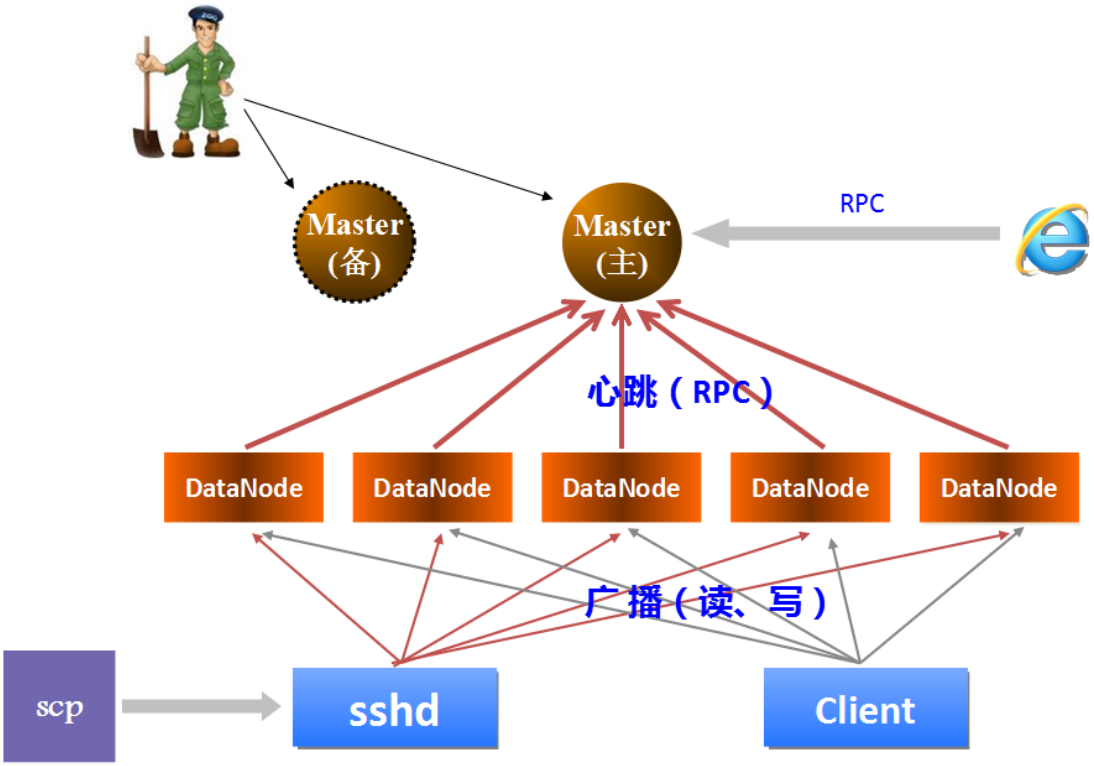
6 设计约束

1) 最多只支持 4 个副本。

7 命令行工具

命令行工具的存在，是为便于直接操作存在于 lightFS 上的整个文件或多个文件。

8 总体结构



备注：lightFS 采用的是弱主架构。

8.1 lightFS-Master

主控节点，主要行驶如下功能：

1) 监控 DataNode，异常时告警；
2) 汇总元数据，提供浏览接口；
3) 补齐副本数；
4) 提供查询缺少副本的接口；
5) 删除脏数据；
6) 提供整个集群运营统计接口；
7) 采用 RPC 和 DataNode 通讯。

8.2 lightFS-DataNode

数据节点，用来存放文件的节点，主要功能为：

1) 存储文件，并提供读写文件接口；
2) 维护自身的文件元数据；
3) 能够根据文件扫描自动重建元数据；
4) 元数据按磁盘分开管理；
5) 磁盘级容灾；
6) 接受 Master 指令，能够从另一 DataNode 拉取副本；

7) 支持 100~1000 的并发量。

8.3 lightFS-Client

客户端，可以为用户程序，也可以为 lightFS 提供的客户端工具：

1) 发起写文件和读文件广播
2) 多写方式实现多副本（缺点：效率低）

8.4 lightFS-Web

管理前台，提供如下功能：

1) 浏览 lightFS 运营状态；
2) 浏览目录树；
3) 执行对文件的删除、改名等操作；
4) 查询缺少副本。

8.4.1 运营指标

1) 每小时/天/周/月，新增/删除/修改的文件个数和大小；
2) 各 DataNode 节点的系统资源、负载情况。

8.5 网络通信

1) lightFS-Master 和 lightFS-Web 间采用 thrift RPC 通信；
2) lightFS-Master 和 lightFS-DataNode 间采用 thrift RPC 通信；
3) lightFS-Client 和 lightFS-DataNode 采用 socket 通信。

lightFS-Master 和 lightFS-DataNode 均为 thrift service，提供 RPC 接口。这里有个需要特别注意地方：在 lightFS-DataNode 端并不配置任何 lightFS-Master 的 IP，采用如下策略来确定 lightFS-Master 的 IP：

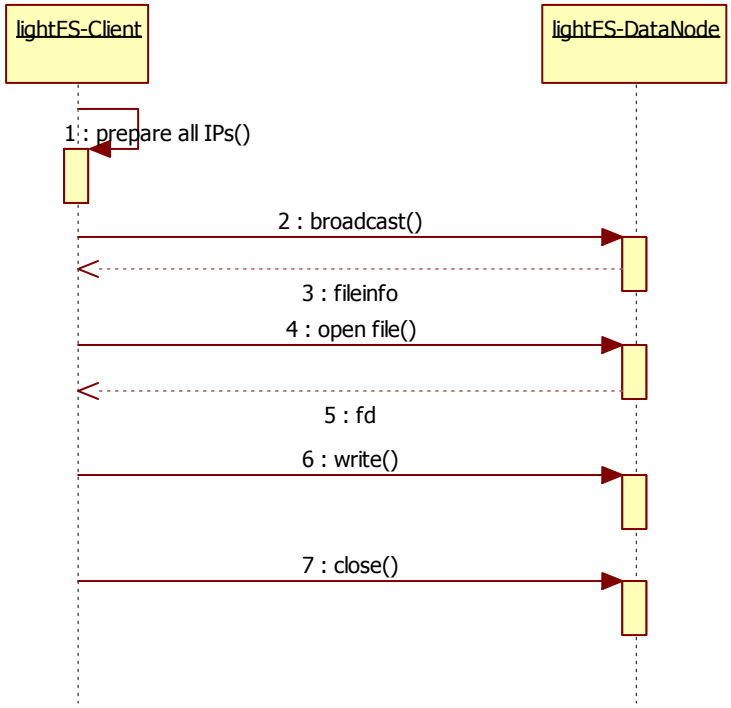
1) lightFS-Master 分主备两个，其中只有主 lightFS-Master 提供服务，而备 lightFS-Master 处于冷备状态；
2) 在 lightFS-Master 成为主时，会通知所有的 lightFS-DataNode，它就是主，这个时候 lightFS-DataNode 就有 lightFS-Master 的 IP 了；
3) 接下来，lightFS-DataNode 就定时向 lightFS-Master 发起心跳。

从以上的过程，可以看出，lightFS-DataNode 和 lightFS-Master 必须互提供 RPC 服务，这样做的好处是：

1) 区别于传统的做法，这里的 lightFS-DataNode 不需要做主备 lightFS-Master 的切换；
2) 同时保证了 lightFS-Master 的简单性，因为它仍是心跳的 server 端，lightFS-DataNode 仍扮演 agent 角色。

8.6 工作流

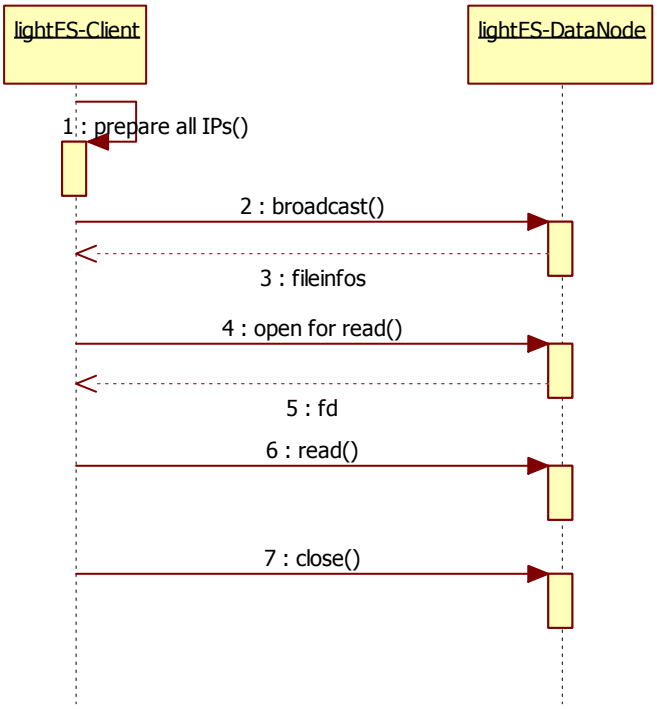
8.6.1 写文件



过程描述：

1) lightFS-Client 准备好所有的 IP（参数传入，或通过域名解析）；
2) lightFS-向所有的 lightFS-DataNode 发起广播，查询指定的文件信息；
3) lightFS-DataNode 返回文件信息（或不存在）；
4) 如果是追加写或覆盖写，而文件存在于某 lightFS-DataNode，则向该节点发起写操作；
   如果是完整的新文件，则随机选择 lightFS-DataNode；
5) 写完成后，lightFS-Client 主动关闭连接。

8.6.2 读文件



过程描述：

1) lightFS-Client 准备好所有的 IP（参数传入，或通过域名解析）；
2) lightFS-向所有的 lightFS-DataNode 发起广播，查询指定的文件信息；
3) lightFS-DataNode 返回文件信息（或不存在）；
4) lightFS-Client 选择一个存在文件的 lightFS-DataNode，向它发起读请求；
5) 读结束后，lightFS-Client 主动关闭连接。

9 lightFS-Master 结构

9.1 模块结构

Metadata	Monitor	thrift service
mooon	thrift	boost

9.1.1 thrift service

对外提供基于 thrift 的 RPC 服务，服务对象为 lightFS-DataNode 和 lightFS-Web 等。

9.1.2 Metadata

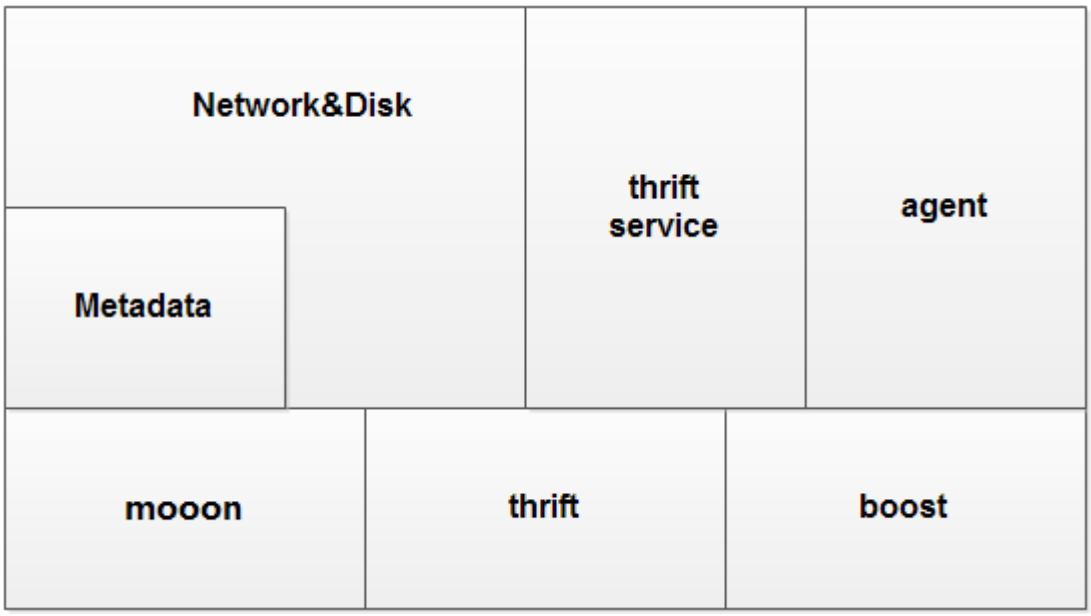
元数据管理。分两种情况：
1) 按节点的元数据管理；
2) 全局元数据管理。

9.1.3 Monitor

监控 lightFS-DataNode。

10 lightFS-DataNode 结构

10.1模块结构



10.1.1 thrift service

对外提供基于 thrift 的 RPC 服务，如向 ligthFS-Master 提供设置lightFS-Master 的 IP 接口。

10.1.2 Metadata

元数据管理。分两种情况：
1) 磁盘侧的元数据管理，即为每个磁盘维护一个独立的元数据；
2) 全局的元数据管理，不持久化，通过磁盘侧的元数据生成。

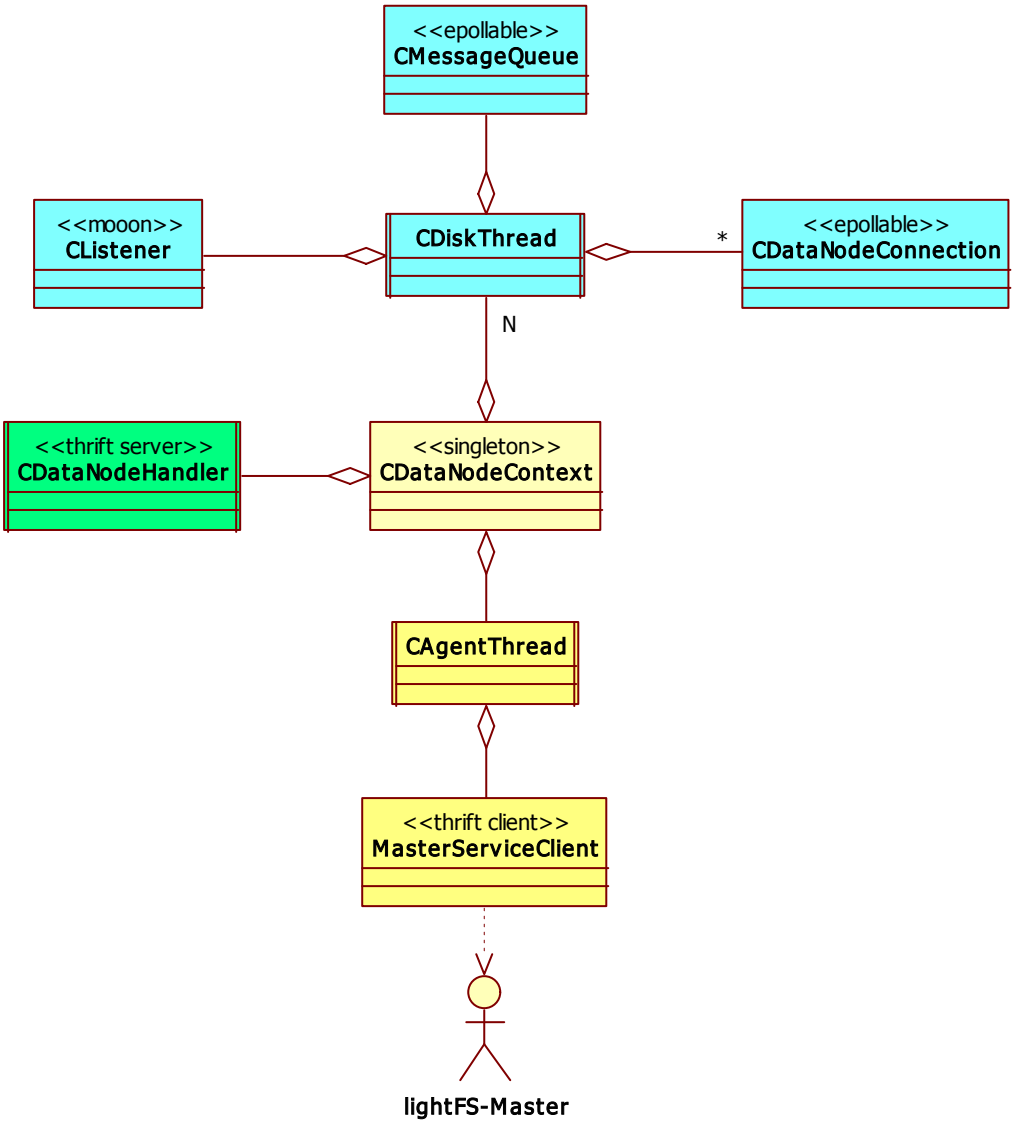
10.1.3 Network&Disk

10.1.3.1 磁盘级容灾

为了实现磁盘级的容灾，将网络数据的发收和磁盘操作耦合在同一个模块中。以磁盘为主角，每个磁盘都有自己专有的线程，这些专有的线程不会跨磁盘操作，提供以下服务：

1) 打开、关闭文件
2) 读写文件
3) 删除文件
4) 文件改名
5) 其它文件操作
6) 收发网络数据
7) 记录本线程的运行日志

10.2类图



10.2.1 CDataNodeContext

lightFS-DataNode 的 Context 上下文类。

10.2.2 thrift service

10.2.2.1 CDataNodeHandler

lightFS-DataNode 的 thrift service 实现。

10.2.3 agent

10.2.3.1 CAgentThread

代理线程，专门用于和 lightFS-Master 通信。

10.2.3.2 MasterServiceClient

和 lightFS-Master 通信的 thrift 客户端。

10.2.4 Network&Disk

10.2.4.1 CDiskThread

和磁盘绑定的线程，完成各种磁盘操作，如读写文件，以及网络数据的收发。

10.2.4.2 CDataNodeConnection

lightFS-Client 和 lightFS-DataNode 间的 TCP/socket 连接。

10.2.4.3 CMessageQueue

消息队列，该消息队列要求是可以 epoll 的，和 CDataNodeConnection 共同由 epoll 监控。

10.2.4.4 CListener

TCP/socket 监听者。

11lightFS-Client 结构

11.1模块结构

lightFS-Client 使用 mooon 作为基础类库和组件：

11.2广播模块

负责向指定范围内的 IP 发起广播，查询指定文件的信息。

11.3删改模块

用于删除文件和对文件改名。

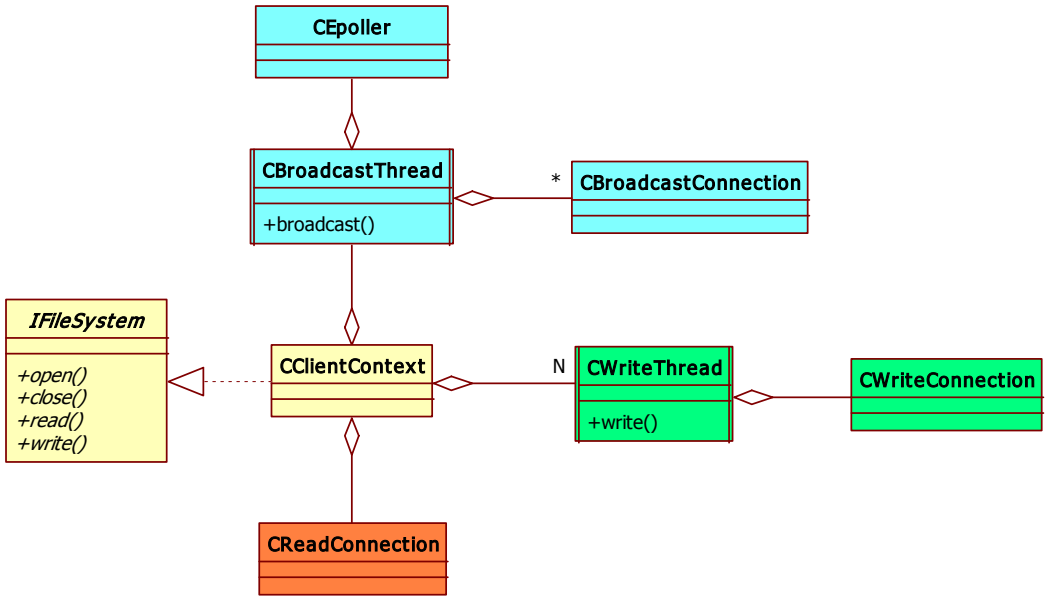
11.4读模块

从指定 IP 上读取文件数据。

11.5写模块

往指定的 IP 上写文件数据。

11.6类图



11.6.1 IFileSystem

lightFS 对外的抽象接口，提供对文件的各种操作。

11.6.2 CClientContext

lightFS-Client 的上下文类，一切能力以该类对外提供。

11.6.3 广播模块

11.6.3.1 CEpoller

来自 mooon::net 的类，提供 epoll 能力。

11.6.3.2 CBroadcastThread

广播线程，只需要一个。

11.6.3.3 CBroadcastConnection

广播用 TCP/socket 连接，采用异步的方式和 lightFS-DataNode 通信。

11.6.4 写模块

11.6.4.1 CWriteThread

向 lightFS-DataNode 写数据的线程，有多少个副本就有多少个 CWriteThread 线程。

11.6.4.2 CWriteConnection

往 lightFS-DataNode 写数据的 TCP/socket 连接，一个副本对应一个连接，由 CWriteThread 同步调度，为防止长时间无响应，需要指定一个超时时间。

11.6.5 读模块

11.6.5.1 CReadConnection

从 lightFS-DataNode 读数据的 TCP/socket 连接，在 lightFS-Client 的主线程中被调用，工作方式同 CWriteConnection，也是同步的，同样也有超时控制。

11.7 API

11.7.1 open

```
// 请注意 open 的返回类型是 long（intptr_t），而不是 int
long open(const char *pathname, int flags, mode_t mode)
{
    CClientRunner* runner = new CClientRunner;
    runner->open(pathname, flags, mode);

    return reinterpret_cast<long>(runner);
}
```

11.7.2 close

```
int close(long fd)
{
    CClientRunner* runner = reinterpret_cast<CClientRunner*>(fd);
    runner->close();
}
```

11.7.3 read

```
int read(long fd, char* buf, size_t bufsize)
{
    CClientRunner* runner = reinterpret_cast<CClientRunner*>(fd);
    return runner->read(buf, bufsize);
}
```

11.7.4 write

```
int write(long fd, const char* buf, size_t bufsize)
{
    CClientRunner* runner = reinterpret_cast<CClientRunner*>(fd);
    return runner->write(buf, bufsize);
}
```

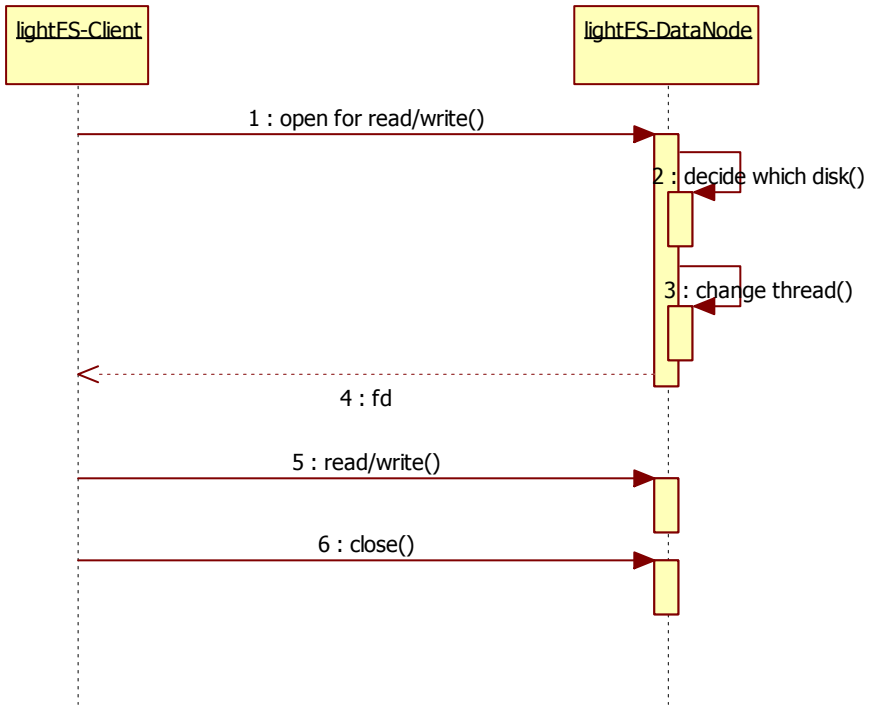
12 通讯协议

12.1广播协议

广播的目的是获知文件的分布存储情况，以便后续的读、写和删改等操作。

12.2读写协议

由于 lightFS 要实现磁盘级容灾，所以它必须有专有的线程。收发网络数据的线程并不一定是文件所在磁盘的专有线程，因此需要有一个切换机制，读写协议就是为解决这个问题而来（对于删改相同）：

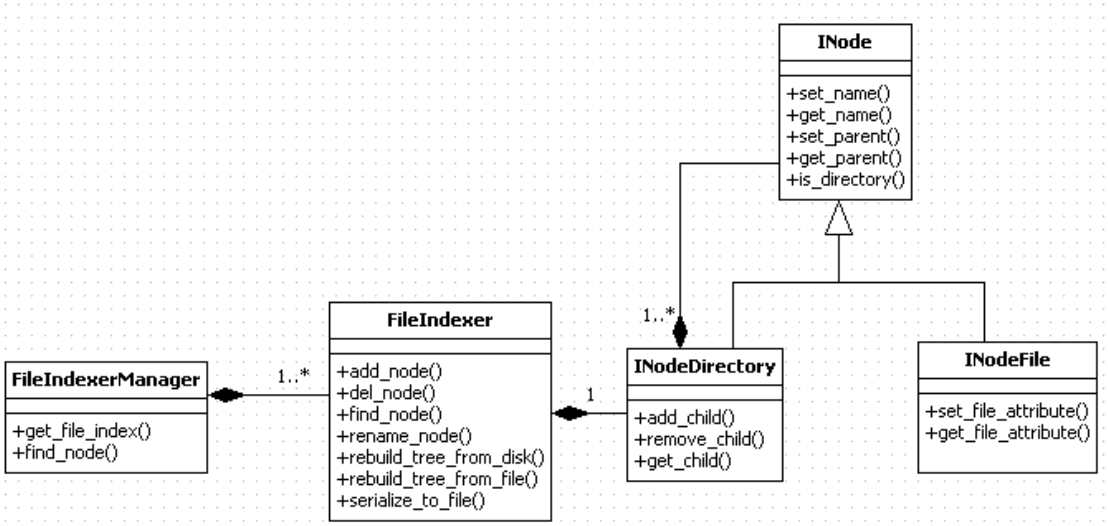


13 目录树设计

13.1设计要求

- 占用内存小，浪费控制在 10%以内；
- 提供一个基目录，可以自动扫描后生成；
- 能够快速打包成字节流，以方便从 DataNode 传到 Master；
- 能够合并，比如在 Master 需要将各节点的目录树合并成一颗大的。

13.2目录树结构



1) 目录树是一颗多叉树，对应类图，实际上是一个组合模式；
2) 每一个目录节点，存在一个孩子节点列表，按照节点名排序；
3) 由于孩子节点已排序，查找时，可以根据节点名二分查找来匹配孩子节点；
4) 每一个磁盘对应一颗目录树；
5) FileIndexerManager 可以根据磁盘 ID 找到对应的目录树；
6) FileIndexer 提供 serialize_to_file 接口,将内存目录树序列化到文件；
7) FileIndexer 提供 rebuild_tree_from_disk 接口，扫描指定目录，重建内存目录树；
8) FileIndexer 提供 rebuild_tree_from_file 接口，能根据目录树文件重建内存目录树；
9) 关于目录树的合并，FileIndexerManager 提供接口，支持从 N 个 CheckPoint 文件中重建成一颗大的内存目录树，master 将用这颗大目录树提给 OSS 提供目录查询服务。

13.3DataNode 的目录树如何实现故障恢复

当 DataNode 宕机时，DataNode 如何恢复内存目录树了？

通过 CheckPoint + 记录流水日志的方式来实现容灾，CheckPoint 即内存目录树的磁盘镜像文件。

1) 对于创建删除更新文件或目录的操作需要更新目录树结构,这些对目录树的更新操作需要先追加写到本地日志文件中，然后才对内存中的目录树进行更新；
2) DataNode 定期将流水和目前的 CheckPoint 文件 merge 成一个新的 CheckPoint 文件，具体做法如下：

A. 调用 FileIndexer 的 rebuild_tree_from_file 接口，根据 CheckPoint 文件建立一颗内存目录树；

B. 遍历流水日志，根据流水的操作类型，相应的调用 FileIndexer 的 add_node,del_node 等接口，这里要注意的是，要按照流水日志文件生成时间的先后顺序来处理，否则会导致数据错乱；

C. 全部处理完毕后，调用 FileIndexer 的 serialize_to_file 接口生成新的 CheckPoint 文件；

假设 DataNode 重启了，则 DataNode 先检测是否有 CheckPoint 文件，如果有，则先根据 CheckPoint 文件重建内存目录树，然后遍历所有有流水日志，调用目录树接口进行回放操作，回放完成后，内存目录树恢复完毕。

14 附 1：技术决策

14.1网络通讯

lightFS 计划基于 mooon 实现，同时采大量使用开源库，如 thrift RPC 来简化实现。

读写文件究竟是采用 RPC 还是直接的 socket 了？

RPC 直接上会简化逻辑，但如何处理网络事件了？比如 DataNode 数据收不过来，Client 什么时候才可以继续发送。

14.2是否支持文件分块

lightFS 的定位究竟是什么，根据它的定位来决定是否需要分块。基于 lightFS 的应用场景和设计目标，最终决定放弃对文件分块，以简化实现。

14.3多副本写入方式

写入多副本有两种常见选择：一是由 Client 直接写入所有副本；二是 Client 只直接写入一个副本，其它副本的写入交给 DataNode 完成：

	Client 直接写入所有副本	**Client 只直接写入一个副本**
优点	1) 简单明了； 2) DataNode 的逻辑相对简单。	1) Client 实现简单； 2) Client 和 DataNode 跨 IDC 时，写操作性能影响小； 3) Client 和 DataNode 可以分布不同的 IDC。
缺点	1) Client 写操作压力较大； 2) Client 和 DataNode 跨 IDC 时，写操作性能下降和副本数约成正比，有异常时更为严重； 3) 应用局限于 Client 和 DataNode 部署在同一个 IDC。	1) DataNode 的逻辑相对复杂。

14.4DataNode 和 Master 通信方向

是 DataNode 主动连接 Master，还是 Master 主动连接 DataNode？

	DataNode 主动连接 Master	**Master 主动连接 DataNode**
优点	1) 实现简单，DataNode 只需要一个专用的 Agent，而 Master 为通用 Server	1) DataNode 上不用配置 Master 的 IP 2) 主备 Master 切换，DataNode 不用关心

缺点	1) 主备 Master 切换时，DataNode 连接的 Master 的 IP 需要变更	1) DataNode 需要为 Master 开端口 2) Master 需连接多个 Server

15 附 2：进一步思考

15.1如何支持文件分块

如果想支持超大（超过 4G）的文件，分块是必须的。

15.2如何支持快速检索

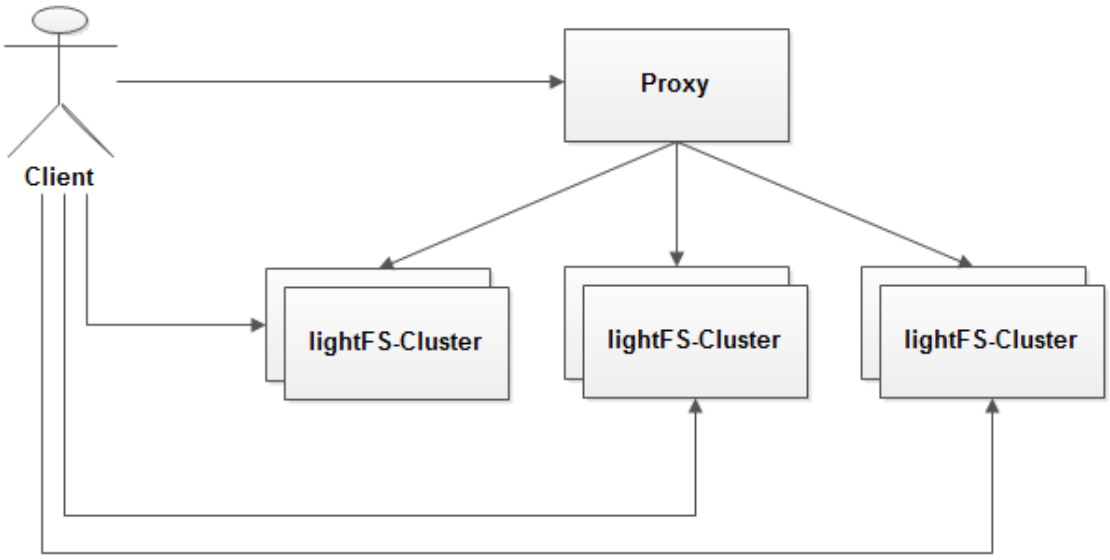
如果有检索的需求，只需要为相应的文件增加一个索引文件即可。

15.3如何支持写时压缩

写入磁盘可以改成抽象接口，通过对该接口的实现，支持不同的压缩策略。

15.4如何支持上万节点的集群

由于 lightFS 的实现，使用了全广播方式，因此限制了集群大小。为了线性扩展 lightFS 的容量，支持上万的节点，加入一个弱有状态的 Proxy：



对于每一个文件，通过它的 path 计算出一个 MD5 值，然后 Proxy 维护一个 MD5 值到 lightFS-Cluster 的映射表。

15.5DataNode 多进程化

将线程模式改成进程模式，即一个磁盘对应一个独立的进程。带来的好处是，磁盘异常时，可以 kill 掉对应的进程，在故障恢复后在重启它。

16 附 3：开发计划

预计召集 3+1 人，其中后台开发 3 人，前台 1 人，7 个工作日左右完成 demo 版本的开发，约 30 个工作日提供一个稳定版本（不包含 ssh 兼容特性）。

16.1人员名单

目前确定会参与开的人员名单如下：

姓名	前后台	负责模块	简介
易剑 eyjian@qq.com	后台	系统设计、项目协调和 lightFS-DataNode 模块的实现	2002 年本科毕业，擅长 C/C++程序开发，热衷于软件设计，对分布式系统和大规模数据系统具有较大兴趣
曹彪 caobiao@gmail.com	后台	lightFS-Client 模块和目录树的实现	熟悉 C/C++，对大数据处理有一定经验，对 MapReduce 等的实现原理有深刻理解
丁峰峰 haofefe@163.com	前台	lightFS-Web 的实现	擅长.net 和 js，自创了基于 HTML5 的画流程图组件 jmgraph
	后台	lightFS-Master 的实现	

17 附 3：授权协议

lightFS 将采用 Apache 协议，毫无保留地对外开放，供学习、研究和直接使用，不附带任何约束条件。