lecture 3 Google File System

1. 概述

1. 分布式系统/分布式存储系统通常是为了获得强大的综合性能
2. **Sharding**：分片，将文件切分到各个服务器上
3. 分布式存储系统的折中：
    - 追求性能 → 分片
    - 分布式系统经常会有部分出错 → 容错
    - 容错 → 复制
    - 复制 → 不一致性
    - 一致性 → 降低性能

    人们往往会牺牲部分的一致性
4. 强一致性：可以认为与整个分布式系统通信等同于与一台计算机通信
5. 将GFS改造成强一致性
    - 检测重复操作（append失败，client retries时，GFS应该能知道之前有重复的操作）
    - 两次提交协议
        - 第一次primary向secondary询问是否能执行相关操作
        - 第二次primary才要求secondary正式执行相关操作
    - primary崩溃时，其中一个secondary接任primary。新的primary必须在开始时与secondary进行显式重新同步（因为旧primary的崩溃可能会导致有部分secondary没收到消息）

1.1. 术语

后文提到的块 == chunk

1.2. GFS创新与特色

1. 相比以往的分布式文件系统，GFS认为部分服务器的故障是一种常态，而不是一种特殊情况或例外。因此，GFS设计时，将**持续监测**、**错误检测**、**容错**、**自动恢复**纳入GFS必不可少的重要功能。

2. GFS中存储的文件具有一定的特点：

    - 数据集很大（TB级别），但数据集中对象很小（KB级别）且很多。

    - 文件的修改主要体现为增加数据而不是重写。

    - 数据一旦写入，很少更改，只会读取，并且为顺序读取。

3. 根据GFS存储文件的特点，对性能的评估和原子性的保证主要集中于追加数据操作，而不是缓存；GFS支持追加数据(append)的**原子性操作**。

4. GFS降低了数据一致性的要求，可以降低应用程序开发的复杂度。

1.2.1. GFS在设计之初提出的几点假设/目标

1. GFS建立在很多容易崩溃的机器上，因此需要容错和恢复机制

2. GFS需要存储大文件，文件的大小在$100MB$级别。也会有若干GB大小的文件。GFS也需要支持小文件，但不需要成为优化重点。

3. 读取工作负载由两方面组成

    - 流式读取（绝大多数情况下） 通常连续读取$1MB$左右的数据。同一客户机的多次读取通常会在同一文件的同一区域进行连续读取

    - 随机读取（少数情况下） 读取位置任意

4. 写入工作负载：

    - 写入数据大小与读取负载的数据大小差不多

    - 写入后一般很少修改

    - 写操作通常也是线性的

    - 需要支持随机写入，但不需要成为优化重点

5. 对于多部客户机同时进行追加操作，需要严格定义语义

6. 高持续(high sustained) 的带宽比延迟更重要。（大部分应用程序需要大批量高速率地处理数据，只有少量应用程序追求低延时）

2. 原理&基本组成

1. GFS由一个master和若干个chunkserver（块服务器）组成。客户端和chunkserver可以运行在同一台计算机上

2. chunk

    - 文件由若干个chunk组成。每个chunk有唯一的chunk handle，在chuck创建时由master分配
    - 每个chunk在文件系统上默认有3份chunk，用户可指定replication factor。
    - chunk本质是linux文件系统上的普通文件，其名字即为chunk句柄

3. 元数据

    - master中存储了整个文件系统的**元数据（metadata）**。包括文件和块的命名空间/存取控制信息/文件映射表（文件名 $\longrightarrow$ 块号数组）/副本映射表（chunk句柄 $\longrightarrow$ chunkserver数组、chunk version number数组、primary chunk、lease expiration time）。并且master控制**块出租管理（chunk lease management？？）**、**垃圾回收机制**、**块迁移机制**。元数据存储在内存中。
    - 元数据存储在内存中的缺点：chunk的数量会受到master内存大小的限制。
    - 文件映射表被登记在操作日志（**operation log**）中，存在master的本地磁盘上。并且操作日志在别的机器上存有副本。
    - master并不会永久保存映射chunk handle $\longrightarrow$ chunkserver数组）、primary chunk 、lease expiration time。master在初始化以及新chunkserver加入时，会向chunkserver询问chunk的位置信息。但是会保存映射chunk handle $\longrightarrow$ chunk version number数组
    - 每个chunk含有$64Byte$的元数据

4. master会定期给chunkserver发送**HeartBeat**，用于给chunkserver发送指令/获取其状态

5. 客户端与master只会进行元数据的通信，chunk的传输由客户端与chunkserver直接进行。client先与master通信，master根据映射表，找到chunk对应的位置（所有副本的位置），将该位置返回给client，client随后会缓存这些位置（即在哪一台chunkserver上），选择最近的chunkserver进行访问。流程如图

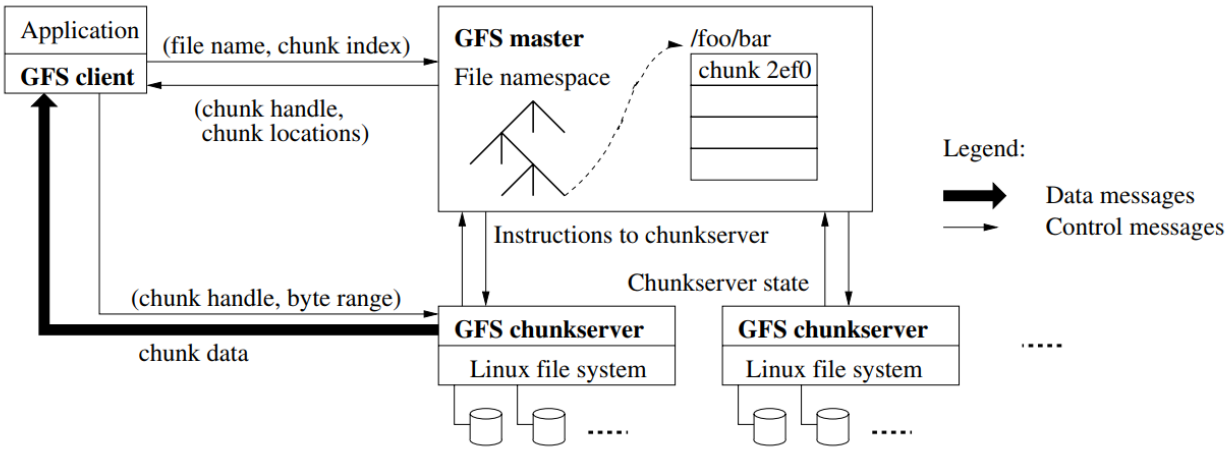

Figure 1: GFS Architecture

6. 当client第二次访问某文件时，不需要再与master通信，除非client缓存的信息丢失或者文件重新打开。

7. client与chunkserver均不缓存文件

   - 原因：对client而言，client读取的文件通常是流式的、线性的（可能无法利用局部性原理），并且client的文件通常很大，无法完整地缓存；对chunkserver而言，chunk存在本地，linux系统会自动进行缓存。

   - 好处：可以简化client和chunkserver，避免引入cache一致性问题

8. 通常client需要请求连续的几个块，因此master可以在一次请求中同时给出其后继的块，这样可以减少client与master的交互。

9. chunk大小

   - 通常为$64MB$，大chunk的优点：

     - 减少client与master的通信次数

     - 减少client与chunkserver的网络开销

     - 减小元数据大小，使得master中元数据可以存储在内存之中。

10. hot spot 问题：

    - 有些文件很小，可能只有一个块。当很多client同时访问存储这些文件的chunkserver时，这些块就构成了**hot spots**

    - 对于GFS设计的背景而言，hot spots并不重要，因为绝大多数文件访问都是顺序读取的

- 在GFS启动的初期，会有一些很小的可执行文件，chunkserver需要依赖这些文件来启动，此时会产生hot spots。解决方案：对这一些文件构造多份副本。

11. **操作日志 （operation log）**
    - 记录了metadata
    - 具有时间线，可以追踪并发操作。
    - 文件、chunk以及文件与chunk的版本，都可以由时间线来区分（在创建时会产生时间线）
    - metadata在持久化结束之前对client不可见。metadata的修改会同时登记在master的本地磁盘以及远端的备份上（对远端备份的操作称为*flush*），这些持久化操作完成之后，master才会client的操作。
    - 当log增长超过一个阈值之后，会将log写入磁盘作为checkpoint。checkpoint使用B+树进行组织。
    - 为了方便checkpoint的创建，master的内部状态经过了特殊的组建，可以允许在创建新检查点时不用推迟新的master变化。切换新的log、checkpoint的创建都是在另一个线程中进行的。（个人理解：需要创建checkpoint时，新建或切换至另一个线程，由该线程负责创建checkpoint，而原线程仍然可以负责master的变化，但此时创建的checkpoint只包含在新建线程之前的master的状态，而不包含随后的master 的变化）
    - 从检查点恢复时，只需要最近的、最新的checkpoint。老的checkpoint原则上可以删除（可以保留下来，用于容灾）。当checkpoint创建过程中发生错误时，恢复程序会检测到这些错误，并且跳过创建时发生错误的checkpoint。

12. **一致性模型（consistency model）**
    - 文件命名空间的变化是原子性的，仅由master来进行相关操作，并且会被记录在operation log中
    - 每个file region在数据更新之后都会有一个状态，如下表所示：

	Write	Record Append
Serial success	*defined*	*defined* interspersed with *inconsistent*
Concurrent successes	*consistent* but *undefined*	
Failure	*inconsistent*	

一致性（consistency）：所有的clients能在所有的replica上看到相同的数据

defined状态：consistent + 数据的写入内容对client可见

undefined状态：上述两者至少有一个不满足

- 对于append操作，与传统的追加在文件末尾的方式不同，追加位置的偏移由master给出，并且会返回给client。master同时还会返回含有记录（record）的defined region的起始位置。
- master会在append的记录之间填充信息，这些信息占据的region为inconsistent，并且数量远远小于records
- 串行的变化/操作序列可以保证mutate之后的文件为defined的原因：
  - master按照同一顺序对某个块的副本（replica）进行操作。
  - master具有探测stale replica的功能。（这些replica可能因为chunkserver故障导致缺失了一些mutation）stale replica将不会参与mutation，它们的地址也不会被发给client。它们会被垃圾回收机制回收。
- 由于client会缓存chunk的位置，因此可能会读到stale replica。这与cache条目的超时以及下一次打开文件有关（？）。下一次打开文件时，这一文件的所有chunk信息将会被刷新。

- master会定期与chunkserver握手检查其状态。当故障发生时，master可以从其他有效的副本中恢复数据。因而仅当所有副本均失效时，数据才无法恢复。这种情况下，client也只会收到报错信息，而不是错误的数据。

2.1. 接口

- 文件系统使用多级目录，文件按名存取。
- 提供了基础文件系统应该提供的功能：*create*、*delete*、*open*、*close*、*read*、*write*等
- 额外提供*snapshot*（快照）*、**record append* 功能
  - snapshot 提供了某份文件/某个目录的备份
  - record append 保证了对某文件追加数据的原子性操作。

2.2. 租赁与变化队列（Leases and Mutation Order）

- **mutation**：可以更改某个块的数据或者元数据的操作。例如追加和随机写入。

- 所有的replica会遵循相同的mutation顺序：

    1. master根据lease grant order从某个chunk的所有replica中选择出一个授予lease，这个replica被称之为primary。lease会在$60s$之后过期。

    2. primary会从所有的mutation中排列一个执行顺序（串行的），所有的replica按照这个执行顺讯进行相关的操作。

- 只要还有replica在进行相关的mutation操作，primary就可以向master申请延期（extension）。延期的请求和授权信息通过master与chunkserver定期的HeartBeat来顺路传输。

- 有时master会提前收回lease授权

- 当primary断开连接时，master可以在其lease授权失效之后，给新的replica授权lease

- master会定期ping primary来检查primary是否仍然工作。

- **脑裂现象（split brain）：** 即存在两个primary同时工作的现象。

  这是一种网络错误，会导致数据不一致。因此即使master ping primary得不到回复，也不会立刻指派新的primary（很有可能只是master和primary之间的网络有问题，而primary仍然在继续工作），必须要等到租约到期之后才能指派新的primary

- 读流程：

    1. client向master发送文件名和偏移

    2. master给client发送chunk句柄以及存有该chunk 副本的所有chunkserver的地址。client会缓存这些地址

    3. client和其中之一的chunk server通信，发送chunk句柄和字节偏移

    4. chunk server返回给client数据

       注意，应用程序需要与GFS库进行连接，读操作会调用相关库的接口。若一次读取超过了一个chunk的范围，则GFS库会自动处理（拆分成多个读请求，而后由GFS库合并数据）

- 写流程：

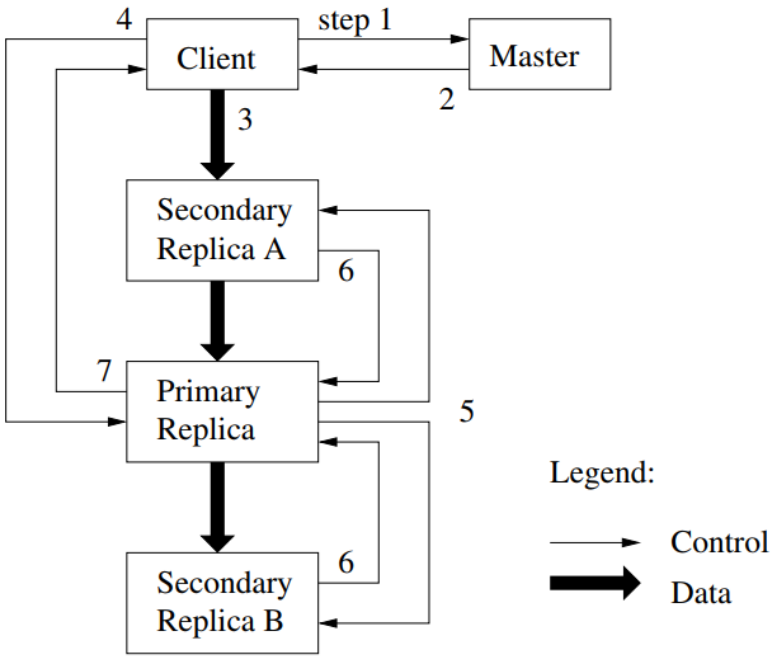

Figure 2: Write Control and Data Flow

1. client询问master哪个chunkserver有lease，向master发送文件名和偏移

2. client返回primary与secondary的数据

   - 若此时master已经指定了primary，则master将primary和secondary（**secondary：除了master之外的replica**）的地址一起返回

   - 若此时master还未指定primary，则master会先选择一个replica作为primary。master会根据自己的版本号，找到和自己版本号相同的replica，选择一个作为primary，而后master将自己对应的chunk的版本号递增，写入磁盘，并给所有的replica通信（内容包括当前replica是primary还是secondary，以及更新后的版本号）。版本号更新仅发生在租约授予阶段。

   仅当client无法与primary通信或者primary回复client它已经没有lease的时候，client才需要与master再次通信

3. client将数据推送到**所有的**replica，replicas将这些数据存在本地的缓存中。传送数据的过程类似一个管道，每个replica会优先寻找最近的chunkserver传送数据。（这条链路在整个数据中心上做了最小化跨越交换机的处理，对于底层网络传输很重要）

4. 所有的replica告知收到了数据后，client向primary发送写请求。primary为所有的并发操作编上一个串行的号码，用于指示执行顺序。而后primary按照这个顺序对primary所存储的数据副本进行相关的操作

5. primary把写请求发给所有的replica，这些replicas按照primary给出的串行顺序执行mutations

6. secondaries与primary通信告知是否已完成mutations

7. primary返回执行情况给client。(若报错，则primary必定没出错，否则会卡在编号这一环节无法推进。)出错时，client会多次试图重新执行3-7的操作

- 若一个写操作太大了，跨越了一个块的范围，则GFS client会将写请求划分成为若干个写请求。但此时如果存在并发的请求，会导致执行的结果consistent但是undefined。（个人理解：这个过大的写请求的执行原本应该是原子的，但是拆开之后可能会有别的并发操作夹在其中。）

2.3. 数据流

- 控制流和数据流解耦，可以充分提高网络传输效率。总体目的为避免网路成为瓶颈以及降低延迟
- 数据传输使用流水线式的方式，而不是使用树形结构进行传递。
- 每台chunkserver会选择距离最近的chunkserver进行传输。距离可以通过IP地址计算得出
- **elapsed time** = $\frac{B}{T}+R\cdot L$，其中B为数据大小（$Byte$），T为吞吐量（$Mbps$），R为replica的数目，L为两台计算机之间的传输延迟

2.4. Record Append 操作

- append允许追加的数据大小为块大小的$\frac{1}{4}$
- 数据流与控制流与前文所提到的类似。但primary会在收到写入请求之后，检查append操作是否会导致当前块超出一个块的大小。
    - 若超出，primary会将当前块填满，而后通知client在下一块上再次发起append请求。同时primary会通知其他所有的replicas进行相同的操作
    - 若不超出，则正常处理
- client只会给primary发送chunk handle以及待写入数据。会由primary选出一个偏移量（通常是primary对应的块的末尾）在此位置追加新纪录，并且会告诉所有的secondary进行同样的操作
- 若append操作失败，可能会导致数据不一致。（注意append操作的原子性只能保证append操作至少有一次是原子性操作）。在这种情况下只能将错就错，未完成执行情况下，offset仍然会增加，下一次追加操作会在当前未完成的append的offset的基础上再进行，并且GFS不

会撤销未完成的追加操作。若后续client再次尝试append，则各个replica中可能会有两份相同的数据（第一次失败的replica则只有一份）

- GFS不能保证数据每个字节都是一致的

2.4.1.

2.5. Snapshot（快照）

- 功能：
  - 快速地创建数据及副本
  - 检查当前状态（在进行可以轻易提交以及回滚的操作之前）
- 快照几乎是即时的（instantaneous）
- 当master收到对某个文件的快照请求时，会回收相关块的lease。如此，后续的写操作产生时，client就必定要与master通信，master此时可以创建相关块的副本。
- 所有的lease过期或回收之后，master把operation log写到磁盘中。随后master通过复制该文件的元数据或者将目录树将log应用于内存状态。
- 使用写时复制技术：在创建快照时，不会立刻复制，当对有快照的文件产生写请求时，才会进行复制
  - snapshot和源文件指向同一个块（会导致这一文件的所有的块引用数目大于等于1，master也就能通过这一信息知道这一个块属于有snapshot指向，但未复制的块）
  - **chunk的引用计数存在master的映射表（file $\longrightarrow chunk$）中**
  - 当对有快照的文件的某个块C第一次产生写请求时，master会要求所有有这个块C的replica的chunkserver创建一份副本C'，新的写请求在C'上进行。而后master才会在C'的所有replica中选择一个授予lease，将primary和secondary的地址返回，如前文所述。

2.6. Master

- 命名空间：
  - 与传统文件系统不同，不使用per-directory数据结构（即列出目录下的所有文件）
  - GFS命名空间的维护：使用一张映射表（从完整的文件路径映射到元数据）
    - 该映射表采用了前缀压缩的技术

        - 映射表存在master内存中
    - namespace tree上的每个结点（既包括文件名，也包括目录名），都有一个读写锁
        - 对于某个文件的所有操作，从根目录到该文件的父目录都需要加上读锁（用于保护目录不被删除、重命名或快照）
        - 对于该文件本身，可以根据操作来选择读锁还是写锁
        - 快照操作需要给文件本身和文件的快照都上写锁
    - 允许同一目录下的并发操作
    - 锁是延迟分配（实际需要之前，才会进行分配）的，并且一旦不被使用就会被删除

2.7. 副本分布

- replica分布在不同的机架（rack）：避免同一机架上的机器全部不能访问
- 多级分发：很多chunkserver分布在不同的机架上，不同机架上的机器通信可能需要经过多个交换机
    - 进出机架的带宽可能小于机架上所有机器的总贷款
    - 为可拓展性（scalability）、可靠性（reliability）和可用性（availability）带来挑战
        - Availability：可用性是系统在某个时间点在正常情况下保持运行以服务于其预期目的**概率。**

          计算公式：$可用性 = \frac{正常运行时间}{正常运行时间+停机时间}$
        - reliability：系统在给定条件下、给定时间段内正确提供服务的能力。

          衡量指标：$MTBF（平均故障间隔时间） = \frac{运行时间}{故障次数}$
        - 可靠性和可用性的极端例子：高可靠、低可用：200年不故障，但是一旦故障要1000年修复；低可靠、高可用：1秒故障一次，但是每次故障修复只需要0.1ms

2.8. 创建（creation）、再复制（re-replication）、再平衡（rebalancing）

2.8.1. 创建

- chunk replica的目的：
    - 创建

    - 再复制
    - 再平衡
- 创建chunk replica的几个考量因素：
    1. 在磁盘利用率低于平均水平的chunkserver上创建replica。随着时间推移，可以使得所有机器的磁盘利用率很接近
    2. 限制每个chunkserver上最近创建的数量
    3. 在不同的机架上设置replica

2.8.2. **再复制**

- 一旦某个chunk可获得的replicas数目小于一定值（由用户指定），就会触发re-replication
- 再复制的原因：
    - 用户提高了可用replica的目标
    - replica无法访问
- 再复制的chunk拥有优先级，优先级可由下面的几点决定：
    - chunk丢失的replica的数目
    - 优先再复制活动文件的块
    - 丢失的replicas是否阻塞的client进程的运行
- 再复制时，会选择优先级最高的chunk进行。创建chunk的考量因素与前文相同。
- 为了保证再复制操作不会过度影响正常的client操作，GFS对整个集群以及每一部chunkserver都设置了活动再复制操作的数目；同时，也限制了每台chunkserver上用于再复制操作的带宽。

2.8.3. **再平衡**

master定期进行再平衡操作。选择那些可用剩余空间低于平均水平的chunkserver，将其中的一些chunks移动到别的chunkserver上。同时，master会优先填满新加入集群的chunkserver

2.8.4. **垃圾收集**

- 文件删除后不会立即删除，而是将其重命名为隐藏文件名+时间戳的格式。可以通过改名进行回复

- master会定期扫描file namespace，会识别超过设定时间（可由用户指定）的、处于删除状态的文件，而后将其元数据删除（从而断开了它与chunks的关联）
- master也会定期扫描chunk namespace，会识别**孤儿chunks（没有file与其关联的chunk）**，而后删除其元数据
- 在heartbeat中，chunkserver向master汇报自己有哪些chunks，master告诉chunkserver哪些chunks的元数据已经被删除。
- 优点：
    - 提高了可靠性和可用性：在创建chunk过程中，可能有些副本创建会失败。删除某文件的消息可能会丢失。这种机制提供了一个统一可靠的方式来回收那些主机不知道的chunks
    - 将垃圾回收融合到日常的heartbeat和namespace扫描中，降低成本。同时还能够使得master更集中于client的请求，而非垃圾收集相关工作
    - 避免了意外不可逆转的删除
- 缺点：由于延迟删除，在存储空间紧张时会捉襟见肘。解决方案为：当某文件第二次被删除时，加快垃圾回收；允许用户根据命名空间指定不同策略（例如在某些目录下，不需要replica等）

2.8.5. Stale Replica探测

- **stale replica**：
    1. chunkserver fails
    2. miss some mutations
- master为每个chunk维护了一个chunk version number用于区分stale chunk和up-to-date chunk，chunk version number同时存储在master和对应的replica上。存储磁盘上
- 每一次master给一个replica授予租约时，会将块所有replica的版本号的数目增加。这一过程发生在client可以写入chunk之前
- chunkserver在启动时，会向master报告自己所有的chunk句柄，以及这些chunk对应的版本号。
- 当master发现有个chunk版本号高于master中记录的版本号时，说明在上一次授予租约时，master failed。此时，master会将这个块在master中的版本号更新到最新的版本号。

- 由于master检测到stale chunk的时机仅为 chunkserver启动/定期垃圾回收机制，因此master可能会将stale chunk作为secondary replica返回给client，也可能会在cloning operation（再复制）时，使得新的块复制了stale chunk的数据。解决办法为在返回时，同时返回master中记录的版本号。则client以及cloning operation在找对应的replica获得数据时，会比较版本号。

3. 容错和诊断

3.1. 高可用性（High Availability）

- 快速恢复（fast recovery）
    - chunkserver和master都能在几秒钟之内恢复
    - client在未完成的请求超时时，会重启、重连、重试操作，会经历暂时性的小故障（打嗝：hiccup）
- master复制
    - master的operation log 和 checkpoint都会在多个机器上保存副本
    - 若运行master的机器上，master进程发生了故障，可以即可重启；若该机器的磁盘等硬件发生了损坏，**需要GFS之外的基础设施介入**，根据operation log选择一台机器作为master
    - client只使用主机的规范名称，这是一个DNS别名，随着master被重定位到别的机器上而改变
    - 一个mutation只有在本地和远端的operation log中都被记录了，才会被认为是commited
    - shadow master
        - 对GFS只读访问
        - 落后于文件系统几分之一秒
        - 为不是频繁改动的文件以及不介意有轻微stale的应用提供了更好的读可用性（由于应用直接从chunkserver获取数据，因此不会读到stale的实际文件内容。可能获取到的stale信息为文件的元数据或者是目录信息）
        - shadow master会在启动时不断轮训各个chunkserver获取其chunk以及replica 的信息，随后也会频繁地与chunkserver通信获取握手信息
        - 仅primary master才有资格决定创建/删除副本，从而导致副本位置变化

3.2. 数据完整性

- 集群中故障现象很常见，但每一台chunkserver又不能通过和其他的chunkserver对比chunk来确保数据完整性，因此只能通过维持和检查校验和（checksum）来实现
- 一个chunk$64MB$被分成若干$64KB$的block，一个block需要$32bit$的校验和。校验和被存放在内存中，并且和日志一起持续性存储。校验和与用户数据分离
- 在请求者（request或者需要进行cloning的chunkserver）向某个chunkserver请求时，chunkserver会检查请求者请求的那一部分数据的checksum，如果数据与checksum不一致，则向请求者报错，并且也向master报错。此时master会启动cloning，而后将这一个chunk replica（不完整的）进行删除
- checksum对**读性能**几乎没有影响：
  - 常见的读取请求基本只会读取几个block
  - 将读取数据与校验和的块边界对齐
  - 检验过程不需要IO操作
- checksum对**写性能**：
  - GFS大多数写操作都是append操作（append在chunk末尾）
  - 可以只在chunk的最后几个与append数据相关的block重新计算校验和
  - 当append时，chunk已经stale的时候，仍然会计算checksum，但是这个不完整的数据会在下一次被读取时检测出来
  - 当覆盖写入已经存有数据的block时，必须检验覆盖范围的第一个block和最后一个block的checksum，然后再执行写入（否则会导致写入的部分隐藏了未写入部分的错误）
  - 对于不那么活跃的chunkserver，chunkserver本身会定期地进行校验。这可以保证那些不活跃的chunk的完整性

4. 诊断工具

- 诊断工具：诊断日志
- 保存内容：chunkserver的启动和关闭、RPC的请求和回复等（但是不包括具体写入和读取的数据）
- 作用：问题隔离、调试、性能分析

- 对性能的影响很小。诊断日志会在磁盘空间允许的情况下尽量保存（删除后对整个系统完全无影响）。诊断日志中最新的数据会保存在内存中并且允许在线监控