03 | The Google File System （一）： Master的三个身份

2021-09-24 徐文浩

《大数据经典论文解读》 课程介绍 >

讲述：徐文浩
时长 20:27 大小 18.74M

你好，我是徐文浩。从今天开始，我们就正式地来一起解读和学习大数据领域中，一些经典的论文。这节课，我们就从“The Google File System”这篇论文开始。

这篇论文发表在 2003 年，现在来看，它算是一篇“老”论文了。然而在我第一次看到这篇论文的时候，它可代表着强大而神秘的黑科技。

在这篇论文发表之前，工业界的分布式系统最多也就是几十台服务器的 MPI 集群。而这篇 GFS 的论文一发表，一下子就拿出了一个运作在 1000 台服务器以上的分布式文件系统，并且这个文件系统，还会面临外部数百个并发访问的客户端，可以称得上是石破天惊。

当然，在 18 年后的今天，开源社区里的各种分布式系统，也都远比当初的 GFS 更加复杂、强大。回顾这篇 18 年前的论文，GFS 可以说是“**技术上辉煌而工程上保守**”。说

GFS 技术上辉煌，是因为 Google 通过廉价的 PC 级别的硬件，搭建出了可以处理整个互联网网页数据的系统。而说 GFS 工程上保守，则是因为 GFS 没有“发明”什么特别的黑科技，而是在工程上做了大量的取舍（trade-off）。

GFS 的设计决策

在我看来，GFS 定了三个非常重要的设计原则，这三个原则带来了很多和传统的分布式系统研究大相径庭的设计决策。但是这三个原则又带来了大量工程上的实用性，使得 GFS 的设计思路后续被 Hadoop 这样的系统快速借鉴并予以实现。

那这三个原则是什么呢？接下来我就给你介绍介绍。

第一个是以工程上“简单”作为设计原则。

GFS 直接使用了 Linux 服务上的普通文件作为基础存储层，并且选择了最简单的单 Master 设计。单 Master 让 GFS 的架构变得非常简单，避免了需要管理复杂的一致性问题。不过它也带来了很多限制，比如一旦 Master 出现故障，整个集群就无法写入数据，而恢复 Master 则需要运维人员手动操作，所以 GFS 其实算不上一个高可用的系统。

但另外一方面，GFS 还是采用了 Checkpoints、操作日志（Operation Logs）、影子 Master（Shadow Master）等一系列的工程手段，来尽可能地保障整个系统的“可恢复（Recoverable）”，以及读层面的“可用性（Availability）”。

可以说，GFS 是恪守“简单”这个原则而设计了第一版本的系统，并且在不破坏这个设计原则的情况下，通过上面一系列独立可插拔的工程策略，来进一步提高系统的可用性。

第二个是根据硬件特性来进行设计取舍。

2003 年，大家都还在用机械硬盘，随机读写的性能很差，所以在 GFS 的设计中，重视的是顺序读写的性能，对随机写入的一致性甚至没有任何保障。

而你要知道，2003 年的数据中心，各台机器的网卡带宽只有 100MB，网络带宽常常是系统瓶颈。所以 GFS 在写数据的时候，选择了流水线式的数据传输，而没有选择树形的数据传输方式。更进一步地，GFS 专门设计了一个 Snapshot 的文件复制操作，在文件复制的

时候避免了数据在网络上传输。这些设计都是为了减少数据在网络上的传输，避免我们有限的网络带宽成为瓶颈。

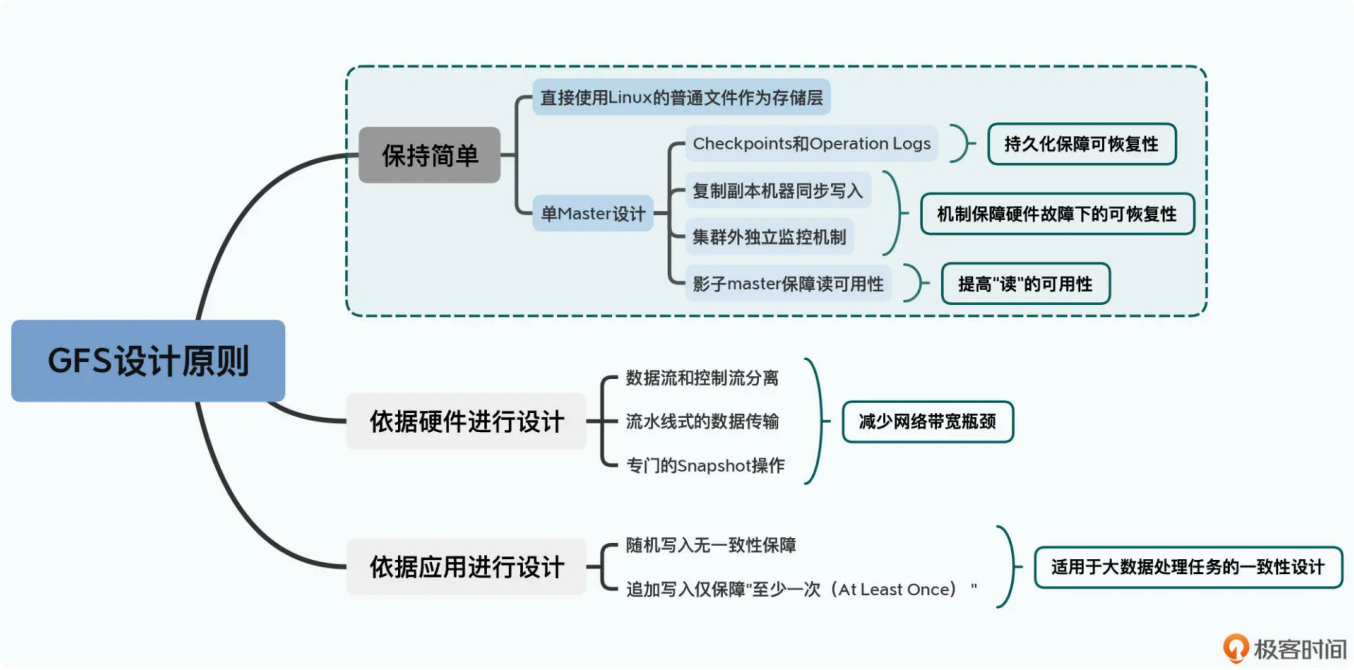
第三个是根据实际应用的特性，放宽了数据一致性（consistency）的选择。

最后，论文里也提到，GFS 是为了在廉价硬件上进行大规模数据处理而设计的。所以 GFS 的一致性相当宽松。GFS 本身对于随机写入的一致性没有任何保障，而是把这个任务交给了客户端。对于追加写入（Append），GFS 也只是作出了“至少一次（At Least Once）”这样宽松的保障。

可以说，GFS 是一个基本没有什么一致性保障的文件系统。但即使是这样，通过在客户端库里面加上校验、去重这样的处理机制，GFS 在大规模数据处理上已经算是足够好用了。

OK，那到这里，你应该就大致理解了 GFS 的设计决策。不过你也要清楚的是，这三个设计原则可远远没有以上我介绍的这么简单。所以接下来，我会通过三节课的时间，来带你分别剖析这三个重要的设计选择。

那么今天这节课，我们就先一起来看看第一个选择，也就是“简单”这个设计原则。



在这个设计原则下，我们会看到 GFS 是一个非常简单的单 Master 架构，但是这个 Master 其实有三种不同的身份，分别是：

相对于存储数据的 Chunkserver，Master 是一个目录服务；

相对于为了灾难恢复的 Backup Master，它是一个同步复制的主从架构下的主节点；

相对于为了保障读数据的可用性而设立的 Shadow Master，它是一个异步复制的主从架构下的主节点。

并且，这三种身份是依靠不同的独立模块完成的，互相之间并不干扰。所以，学完这一讲，你对常见的主从架构（Master-Slave）下的 Master 的职责，以及数据复制的模式也会有一个清晰的认识。

Master 的第一个身份：一个目录服务

作为一个分布式文件系统，一个有几千台服务器跑在线上的真实系统，GFS 的设计可以说是非常简单。

我给你举个例子。要把一个文件存在 GFS 上，其实和在 Linux 操作系统上很像，GFS 一样会通过“命名空间 + 文件名”来定义一个文件。比如，我们可以把这一讲的录音文件存储在 /data/geektime/bigdata/gfs01 这样一个路径下。这样，所有 GFS 的客户端，都可以通过这个 /data/geektime/bigdata 命名空间加上 gfs01 这个文件名，去读或者写这个文件。

那么，当我们的客户端，实际上要去读或者写这个 gfs01 文件的时候，这个文件是实际存放在哪个物理服务器上呢？以及我们的客户端，具体是怎么读到这个文件的呢？

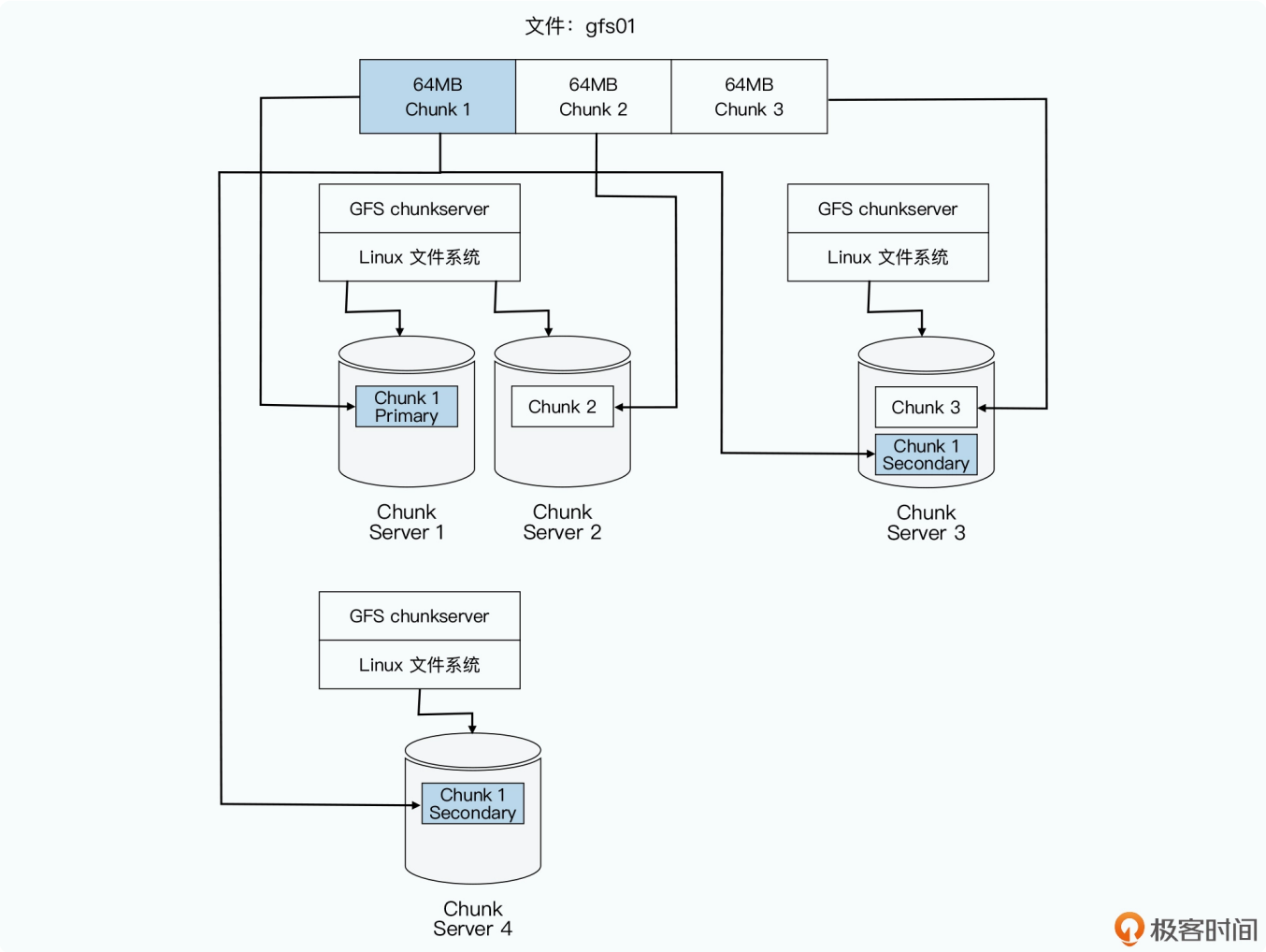
首先你要知道，在整个 GFS 中，有两种服务器，一种是 **master**，也就是整个 GFS 中有且仅有一个的主控节点；第二种是 **chunkserver**，也就是实际存储数据的节点。

而既然 GFS 是叫做分布式文件系统，那么这个文件，其实就可以不存储在同一个服务器上。

因此，在 GFS 里面，会把每一个文件按照 64MB 一块的大小，切分成一个个 chunk。每个 chunk 都会有一个在 GFS 上的唯一的 handle，这个 handle 其实就是一个编号，能够唯一标识出具体的 chunk。然后每一个 chunk，都会以一个文件的形式，放在 chunkserver 上。

而 chunkserver，其实就是一台普通的 Linux 服务器，上面跑了一个用户态的 GFS 的 chunkserver 程序。这个程序，会负责和 master 以及 GFS 的客户端进行 RPC 通信，完成实际的数据读写操作。

当然，为了确保数据不会因为某一个 chunkserver 坏了就丢失了，每个 chunk 都会存上整整三份副本（replica）。其中一份是主数据（primary），两份是副数据（secondary），当三份数据出现不一致的时候，就以主数据为准。有了三个副本，不仅可以防止因为各种原因丢数据，还可以在有很多并发读取的时候，分摊系统读取的压力。



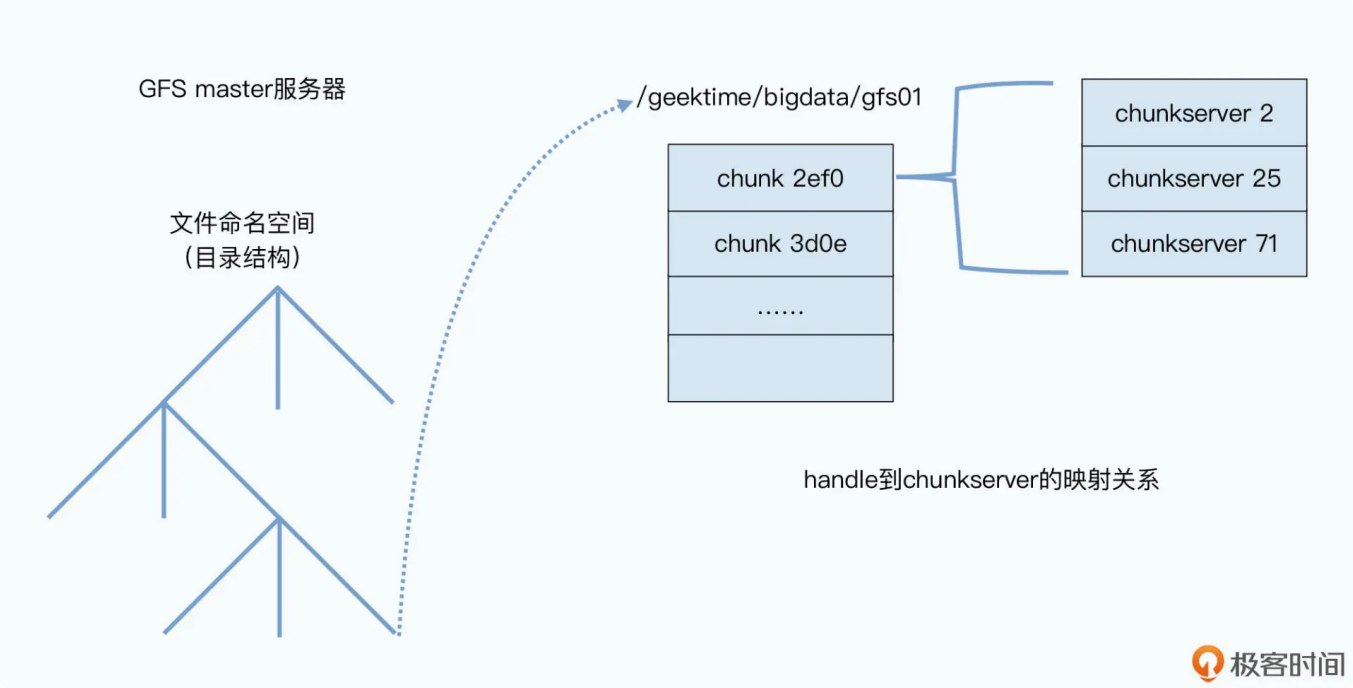
文件会拆分成chunk放在不同的chunkserver上

如此一来，文件就被拆分成了一个个的 chunk 存在了 chunkserver 上。那么你可能还要问：**GFS 的客户端，怎么知道该去哪个 chunkserver 找自己要的文件呢？**

答案当然是问 master 啦。

首先，master 里面会存放三种主要的元数据（metadata）：

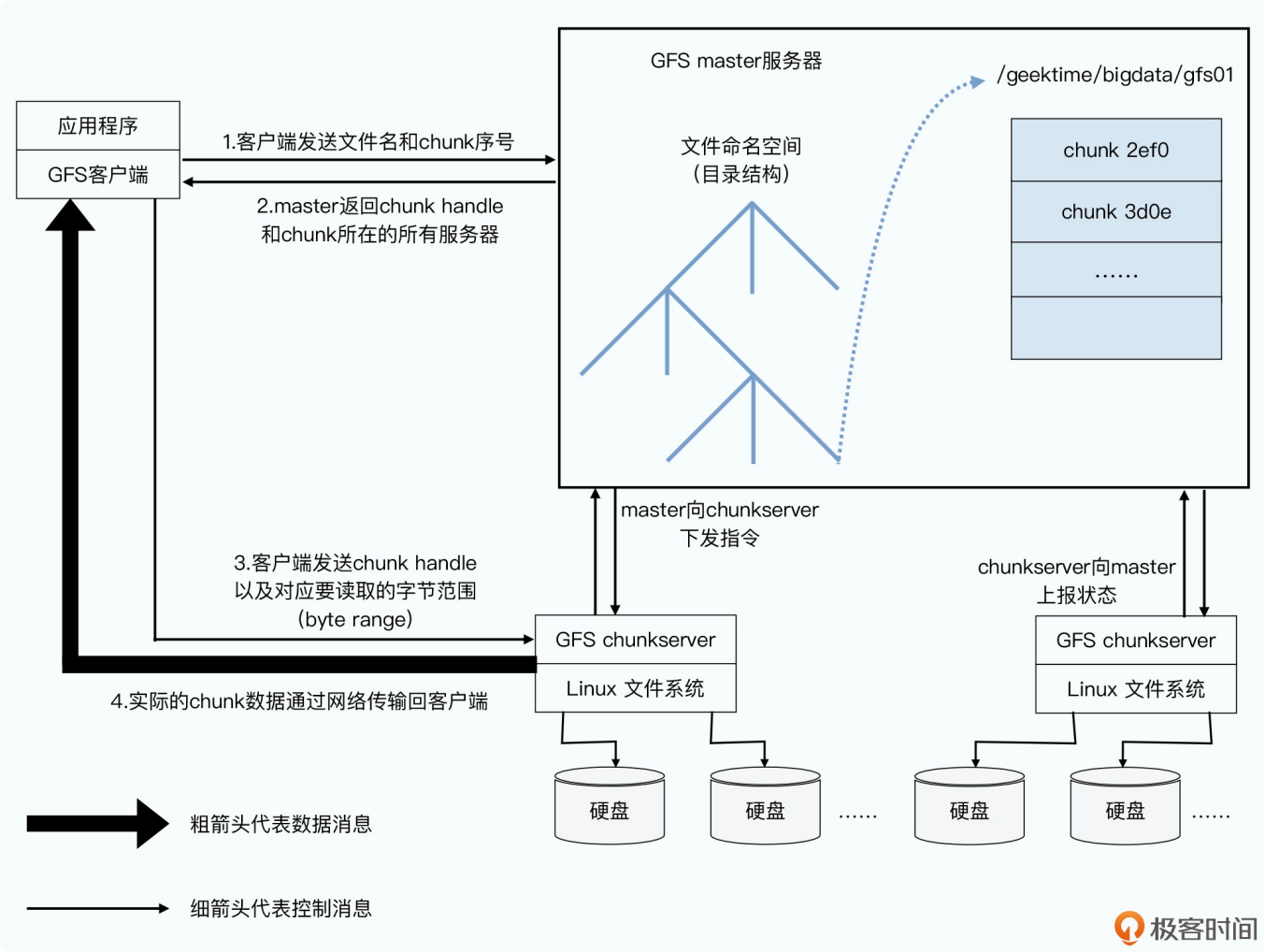
1. 文件和 chunk 的命名空间信息，也就是类似前面 /data/geektime/bigdata/gfs01 这样的路径和文件名；
2. 这些文件被拆分成了哪几个 chunk，也就是这个全路径文件名到多个 chunk handle 的映射关系；
3. 这些 chunk 实际被存储在了哪些 chunkserver 上，也就是 chunk handle 到 chunkserver 的映射关系。



然后，**当我们要通过一个客户端去读取 GFS 里面的数据的时候，需要怎么做呢？**GFS 会有以下三个步骤：

1. 客户端先去问 master，我们想要读取的数据在哪里。这里，客户端会发出两部分信息，一个是文件名，另一个则是要读取哪一段数据，也就是读取文件的 offset 及 length。因为所有文件都被切成 64MB 大小的一个 chunk 了，所以根据 offset 和 length，我们可以很容易地算出客户端要读取的数据在哪几个 chunk 里面。于是，客户端就会告诉 master，我要哪个文件的第几个 chunk。
2. master 拿到了这个请求之后，就会把这个 chunk 对应的所有副本所在的 chunkserver，告诉客户端。
3. 等客户端拿到 chunk 所在的 chunkserver 信息后，客户端就可以直接去找其中任意的一个 chunkserver 读取自己所要的数据。

你可以参考下图所展示的客户端读取数据的整个过程指令流向。



GFS里客户端读取数据的整个过程指令流向

这整个过程抽象一下，其实和 Linux 文件系统差不多。master节点和 chunkserver 这样两种节点的设计，其实和操作系统中的文件系统一脉相承。**master 就好像存储了所有 inode 信息的 super block，而 chunk 就是文件系统中的一个个 block。**只不过 chunk 比 block 的尺寸大了一些，并且放在了很多台不同的机器上而已。我们通过 master 找到 chunk 的位置来读取数据，就好像操作系统里通过 inode 到 block 的位置，再从 block 里面读取数据。

所以，这个时候的 master，其实就是一个“**目录服务**”，master 本身不存储数据，而是只是存储目录这样的元数据。这个和我们的单机系统的设计思想是一样的。其实在计算机这个行业中，所有的系统都是从最简单、最底层的系统演化而来的。而这个课程中你看到的大部分的设计，其实都有这个特质。

master 的快速恢复性和可用性保障

不过，简单不是没有代价的。

在这个设计下，你会发现 GFS 里面的 master 节点压力很大。在一个 1000 台服务器的集群里面，chunkserver 有上千个，但 master 只有一个。几百个客户端并发读取的数据，虽然可以分摊到那 1000 个 chunkserver 的节点上，但是找到要读的文件的数据存放在哪里，都要去 master 节点里面去找。

所以，master 节点的所有数据，都是**保存在内存**里的。这样，master 的性能才能跟得上几百个客户端的并发访问。

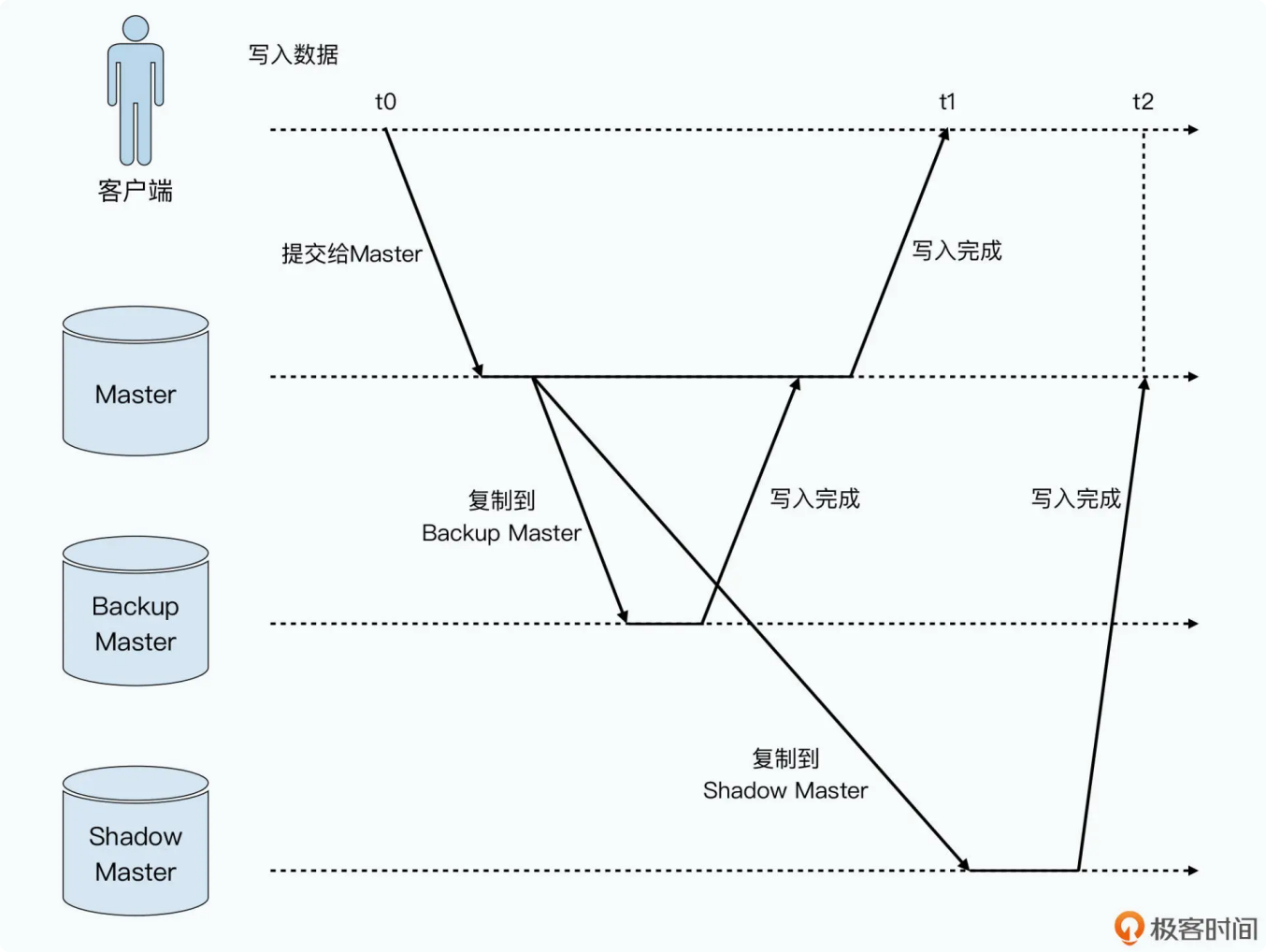
但是数据放在内存里带来的问题，就是一旦 master 挂掉，数据就会都丢了。所以，master 会通过记录操作日志和定期生成对应的 Checkpoints 进行持久化，也就是写到硬盘上。

这是为了确保在 master 里的这些数据，不会因为一次机器故障就丢失掉。当 master 节点重启的时候，就会先读取最新的 Checkpoints，然后重放（replay）Checkpoints 之后的操作日志，把 master 节点的状态恢复到之前最新的状态。这是最常见的存储系统会用到的**可恢复机制**。

当然，光有这些还不够，如果只是 master 节点重新启动一下，从 Checkpoints 和日志中恢复到最新状态自然是很快的。**可要是 master 节点的硬件彻底故障了呢？**

你要知道，去数据中心重新更换硬件可不是几分钟的事情，所以 GFS 还为 master 准备好了几个“备胎”，也就是另外几台 Backup Master。所有针对 master 的数据操作，都需要同样写到另外准备的这几台服务器上。只有当数据在 master 上操作成功，对应的操作记录刷新到硬盘上，并且这几个 Backup Master 的数据也写入成功，并把操作记录刷新到硬盘上，整个操作才会被视为操作成功。

这种方式，叫做数据的“**同步复制**”，是分布式数据系统里的一种典型模式。假如你需要一个高可用的 MySQL 集群，一样也可以采用同步复制的方式，在主从服务器之间同步数据。



在t1时间点上，客户端得到写入成功的返回，此时Master和Backup Master的数据是一致的，但是Shadow Master的数据，要到t2时间点才会追上来

而在同步复制这个机制之外，在集群外部还有监控 master 的服务在运行。如果只是 master 的进程挂掉了，那么这个监控程序会立刻重启 master 进程。而如果 master 所在的硬件或者硬盘出现损坏，那么这个监控程序就会在前面说的 Backup Master 里面找一个出来，启动对应的 master 进程，让它“备胎转正”，变成新的 master。

而这个里面的数据，和原来的 master 其实一模一样。

不过，为了让集群中的其他 chunkserver 以及客户端不用感知这个变化，GFS 通过一个规范名称（Canonical Name）来指定 master，而不是通过 IP 地址或者 Mac 地址。这样，一旦要切换 master，这个监控程序只需要修改 DNS 的别名，就能达到目的。有了这个机制，GFS 的 master 就从之前的可恢复（Recoverable），进化成了能够**快速恢复（Fast Recovery）**。

不过，就算做到了快速恢复，我们还是不满足。毕竟，从监控程序发现 master 节点故障、启动备份节点上的 master 进程、读取 Checkpoints 和操作日志，仍然是一个几秒级

别乃至分钟级别的过程。在这个时间段里，我们可能仍然有几百个客户端程序“嗷嗷待哺”，希望能够在 GFS 上读写数据。虽然作为单个 master 的设计，这个时候的确是没有办法去写入数据的。

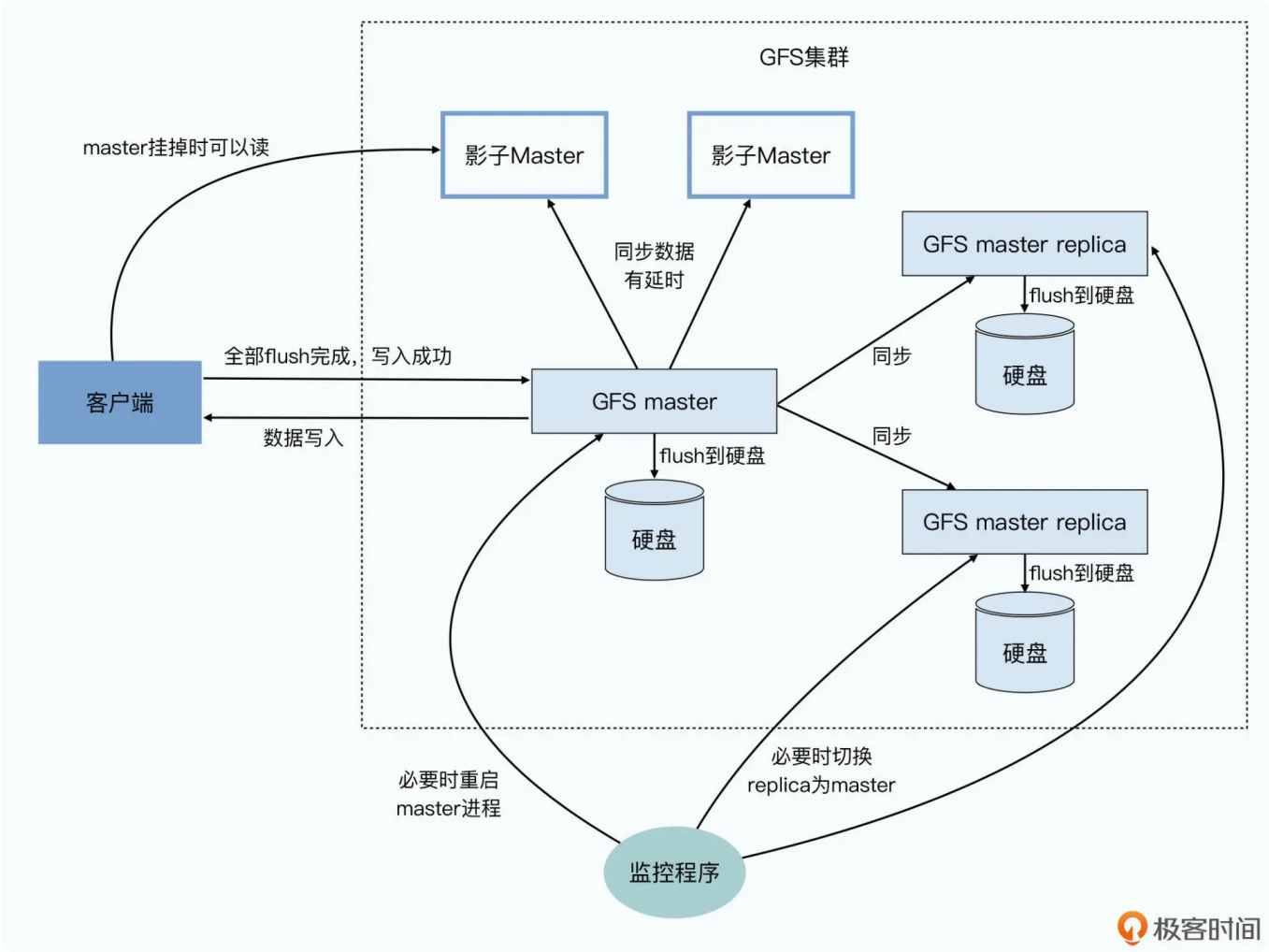
但是 Google 的工程师还是想了一个办法，让我们这个时候还能够从 GFS 上读取数据。

这个办法就是加入一系列**只读的“影子 Master”**，这些影子 Master 和前面的备胎不同，master 写入数据并不需要等到影子 Master 也写入完成才返回成功。而是影子 Master 不断同步 master 输入的写入，尽可能保持追上 master 的最新状态。

这种方式，叫做数据的“**异步复制**”，是分布式系统里另一种典型模式。异步复制下，影子 Master 并不是和 master 的数据完全同步的，而是可能会有一些小小的延时。

影子 Master 会不断同步 master 里的数据，不过当 master 出现问题的时候，客户端们就可以从这些影子 Master 里找到自己想要的信息。当然，因为小小的延时，客户端有很小的概率，会读到一些过时的 master 里面的信息，比如命名空间、文件名等这些元数据。但你也要知道，这种情况其实只会发生在以下三个条件都满足的情况下：

1. 第一个，是 master 挂掉了；
2. 第二个，是挂掉的 master 或者 Backup Master 上的 Checkpoints 和操作日志，还没有被影子 Master 同步完；
3. 第三个，则是我们要读取的内容，恰恰是在没有同步完的那部分操作上；



相比于这个小小的可能性，影子 Master 让整个 GFS 在 master 快速恢复的过程中，虽然不能写数据，但仍然是完全可读的。至少在集群的读取操作上，GFS 可以算得上是“高可用（High Availability）”的了。

小结

好了，相信到这里，你应该就对 GFS 的基本架构有一定的了解了。GFS 并不是一篇有着大量新的研究发现的理论性的论文。恰恰相反，整个 GFS 的架构，是通过非常重要的工程原则来设计的，也就是**尽量简单、需要考虑实际的硬件情况**，以及**针对实际的应用场景进行设计**。

今天，我们重点围绕 GFS 的“简单性”，了解了 GFS 的基础架构设计。GFS 采用了单 Master 架构，但是这个 Master 有着三重不同的含义。

首先，作为和 chunkserver 相对的 master，它是一个目录服务。所有的元数据都会保留在我们的 master 里，而所有的数据会保存在 chunkserver 里。master 存的就好比是 inode 信息，而 chunkserver 存的则是实际的 block。

其次，为了保障系统在故障下的快速恢复能力和读数据的高可用性，GFS 又对这个 master 添加了各种工程保障措施。这些措施，包括 Checkpoints 和操作日志、Backup Master、外部的监控程序，以及只读的影子 Master。在这个高可用场景下，Master 是唯一可以写入数据的节点。Backup Master 通过同步复制的方式从 master 同步数据，而影子 Master 则通过异步复制的方式来同步 master 的数据。

在这个场景下，Backup Master 和影子 Master 其实是 Master 的 Slave 节点。如果我们把 Master 当成是一个 MySQL 数据库的主节点，那么 Backup Master 就是配置了高可用的同步复制的 HA 节点，而影子 Master 就是只配置了异步复制分担数据读取压力的 readonly 节点。

我希望你在学完这篇论文之后，也能够学以致用，**当你日后在做系统设计的时候，就从“简单”开始，让系统更容易维护和迭代**。同样的，你也一定会面临，你的 master 需要承担哪些职责、采用哪些数据复制方式的问题。

下节课，我会带你来看看 GFS 另外一个非常重要的设计原则，那就是根据实际硬件情况进行系统设计。你会发现，软件系统的设计，不是抽象的画画图纸，而是和我们设计汽车、飞机一样，需要考虑真实的物理硬件的性能和限制的。甚至有些设计，就是专门针对硬件的物理特性的。

推荐阅读

最后，专栏是你学习的起点，我非常推荐你去读一读 GFS 论文的原文。如果你觉得自己英文不够好，作为 2003 年的一篇“老”论文，网上也有大量的中文解读笔记，你都可以去翻一翻，我把论文的链接放在了这里。当然，还有一个选择，就是去读一读借鉴了 GFS 的 Hadoop 上的 HDFS 的论文，我也把链接放在了这里。

除此之外，MIT 的开放课程，分布式系统里的第三讲，就是针对 GFS 的专门讲解，你也可以去看一看他们的视频，网上也能找到有中文字幕的版本。

思考题

在课程的最后，我来给你留两道思考题，你可以思考一下：

1. 如果你读了论文原文，你应该看到 GFS 的客户端会在读取数据的过程中，把一些数据缓存了下来，那么它究竟缓存了哪些数据来减少频繁的网络往来？在这个缓存机制上，客户端有可能会读到过时的数据吗？
2. 我前面说了，master 的数据都会通过操作日志和 Checkpoints 持久化在硬盘上。但这句话其实不完全正确，每个 chunk 存放在什么 chunkserver 上的这些元数据，master 并不会持久化。读完论文之后，你能说说当 master 重启的时候，怎么重新拿到这个数据吗？

你可以把你的答案留在评论区，和大家一起探讨、交流，看看自己的理解是不是正确。

上一篇 02 | 学习方法：建立你的大数据知识网络

精选留言 (2)

pedro

2021-09-24

mit6.824在讲GFS的时候说，GFS并没有什么理论创新，但是它一下子搞了1000多台机器的集群，对其它论文而言完全是降维打击，然后它就被会议接收了。


峰

2021-09-24

1. 为了减少master的压力，所以要缓存master上的元数据信息。应该会造成读过期数据，因为写入不可变，但支持追加写，对于追加写入的chunk的元数据，怎么同步到客户端缓存按照GFS简单性的原则怕是不想做了。

...

展开



1