19271169-张东植-读后感

本部分为对Google MapReduce论文的读后感兼阅读笔记。

1. MapReduce

1.1 Introduction

> MapReduce：
>
> Google在2003年和2004年提出一篇论文,面向大数据的并行处理的框架模型--MapReduce.
>
> 面向大数据并行处理的计算模型、框架和平台，是一种编程模型，用于大规模数据集（大于1TB）的并行运算，我们只需要我们想要执行的运算即可，而那些并行计算、容错、数据分布、负载均衡等复杂的细节，这些问题都被封装在一个库中，我们能直接调用。
>
> 概念"Map"和"Reduce"，是它们的主要思想，都是从函数式编程语言里借来的，还有从矢量编程语言里借来的特性。它极大地方便了编程人员在不会分布式并行编程的情况下，将自己的程序运行在分布式系统上。 当前的软件实现是指定一个Map函数，用来把一组键值对映射成一组新的键值对，指定并发的Reduce函数，用来保证所有映射的键值对中的每一个共享相同的键组。
>
> 单个计算机无论再好的设备处理能力终究有限，而MapReduce通过简单的接口来实现自动的并行化和大规模的分布式计算，在大量普通的PC机上实现高性能运算。他可以很简单粗暴的通过不断增加PC机数目来提高性能.将此难题迎刃而解，同时促进了大数据时代的到来。MapReduce的主要思想概括说来就是四个字：化整为零，通过Map(映射)和Reduce(规约)来实现.

- **适用场景：**
  - 数据查找：分布式Grep
  - Web访问日志分析：词频统计、网站PV UV统计、 Top K问题
  - 倒排索引：建立搜索引擎索引（根据值找健）
  - 分布式排序
- **优缺点：**
  - **优点**：
    - 模型简单：Map + Reduce；
    - 高伸缩性：支持横向扩展；
    - 灵活：结构化和非结构化数据；
    - 速度快：高吞吐离线处理数据；
    - 并行处理：编程模型天然支持并行处理；
    - 容错能力强；
  - **缺点**：
    - 流式数据：MapReduce处理模型就决定了需要静态数据；
    - 实时计算：不适合低延迟数据处理，需要毫秒级别响应（MapReduce处理延迟较高）；
    - 复杂算法：例如SVM支持向量（机器学习算法）；
    - 迭代计算：例如斐波那契数列（后边的计算需要前面的计算结果）；

1.2 Programming Model

MapReduce库为用户提供了两个函数：

- **Map**
- **Reduce**

MapReduce 模型的原理是利用一个输 入K / V pair集合来产生一个输出的K / V pair集合。

论文中举了一个例子：就是在一个大的文档中计算每个单词出现的的次数，它首先用Map函数输出文档中的每个词、以及这个词的出现次数，Reduce函数把Map函数产生的每一个特定的词的计数累加起来。然后再用户的代码中使用一个可选的调节参数来完成一个符合MapReduce模型规范的对象，在使用中，直接调用MapReduce库，链接在一起，便可实现。

1.2.1 Principle

用户自定义的 Map 函数接受一个输入的 key/value pair 值，然后产生一个中间 key/value pair 值的集合。MapReduce 库把所有具有相同中间 key 值 I 的中间 value 值集合在一起后传递给 reduce 函数。用一个例子详细解释mapreduce思想。要求数出1000张纸牌中有多少张黑桃，100个人每人随机发放10张，每个人数出自己拿到的牌中有多少张黑桃，这是map函数实现的，reduce函数的作用是将100人数的张数加在一起输出。Key是黑桃，value是黑桃的张数。

1.3 Implementation

MapReduce模型可以有多种不同的实现方式:有的适用于小型的共享内存方式的机器，有的则适用于大型NUMA架构的多处理器的主机，而有的实现方式更适合人型的网络连接集群。

在文中，作者主要描述了一个适用 Gougle内部广泛使用的运算环境的实现:用以太网交换机连接、由普通PC机组成的大型网络连接集群。

文中作者强调环境包括：

1. x86 架构、运行Linux操作系统、双处理器、2-4GB内存的机器。
2. 普通的网络硬件设备，每个机器的带宽为百兆或者千兆，但是远小于网络的平均带宽的一半。
3. 集群中包含成百 上千的机器，因此，机器故障是常态。
4. 存储为廉价的内置 IDE硬盘。
5. 用户提交工作(job)给调度系统。

1.3.1 Execution

1. 用户程序首先调用的 MapReduce 库将输入文件分成 M 个数据片度，每个数据片段的大小一般从 16MB 到 64MB(可以通过可选的参数来控制每个数据片段的大小)。然后用户程序在机群中创建大量的程序副本。
2. 这些程序副本中的有一个特殊的程序–master。副本中其它的程序都是 worker 程序，由 master 分配任务。有 M 个 Map 任务和 R 个 Reduce 任务将被分配，master 将一个 Map 任务或 Reduce 任务分配给一个空闲的 worker。
3. 被分配了 map 任务的 worker 程序读取相关的输入数据片段，从输入的数据片段中解析出 key/valuepair，然后把 key/value pair 传递给用户自定义的 Map 函数，由 Map 函数生成并输出的中间 key/valuepair，并缓存在内存中。
4. 缓存中的 key/value pair 通过分区函数分成 R 个区域，之后周期性的写入到本地磁盘上。缓存的 key/value pair 在本地磁盘上的存储位置将被回传给 master，由 master 负责把这些存储位置再传送给Reduce worker。
5. 当 Reduce worker 程序接收到 master 程序发来的数据存储位置信息后，使用 RPC 从 Map worker 所在主机的磁盘上读取这些缓存数据。当 Reduce worker 读取了所有的中间数据后，通过对 key 进行排序后使得具有相同 key 值的数据聚合在一起。由于许多不同的 key 值会映射到相同的 Reduce 任务上，因此必须进行排序。如果中间数据太大无法在内存中完成排序，那么就要在外部进行排序。
6. Reduce worker 程序遍历排序后的中间数据，对于每一个唯一的中间 key 值，Reduce worker 程序将这个 key 值和它相关的中间 value 值的集合传递给用户自定义的 Reduce 函数。Reduce 函数的输出被追加到所属分区的输出文件。

7. 当所有的 Map 和 Reduce 任务都完成之后，master 唤醒用户程序。在这个时候，在用户程序里的对 MapReduce 调用才返回。

1.3.2 Fault Tolerance

Master会周期性的ping每个worker，如果在一个周期中没有收到worker的消息就标记为失效，并把他的任务分配给其他的worker。而对于master失效的方法MapReduce没有太好的解决方法，作者是周期性的将信息写入磁盘，如果失效了就让用户重新启动MapReduce。

- **Worker失效**：

  MapReduce的Master节点会周期地向每一个Worker 发送 Ping信号（为什么不设计成Worker汇报机制？）。如果某个Worker一段时间内没有响应，Master 就会认为Worker已经不可用。

  对于失效Worker，任何分配给该 Worker 的 Map 任务，无论是正在运行还是已经完成，都需要由 Master 重新分配给其他 Worker，因为该 Worker 不可用也意味着存储在该 Worker 本地磁盘上的中间结果亦无法访问。同时，Master 也会将这次重试通知给所有 Reducer，Reducer 会开始从新的 Mapper 上获取数据。

  同时，Master 则会将Worker上未完成的Reduce任务分配给其他 Worker。鉴于 Google MapReduce 的结果是存储在 Google File System 上的，GFS保证了已完成任务的数据的可用性。

- **Master 失效**：

  整个 MapReduce 集群中只有一个 Master 节点，MapReduce 使用Chubby来提高服务的可用性。

  Master在运行时会周期性地将集群的当前状态作为保存点（Checkpoint）写入到磁盘（GFS？）中。Master 退出后，重新启动的 Master 进程即可利用存储在磁盘中的数据恢复到上一次保存点的状态。

- **Worker落后**：

  如果某个 Worker 花了特别长的时间来完成最后的几个 Map 或 Reduce 任务，整个 MapReduce 计算任务的耗时就会因此被拉长，这样的 Worker 称为落后者（Straggler）。

  MapReduce 在整个计算完成到一定程度时（何时？）就会将剩余的任务进行冗余：即同时将其分配给其他空闲 Worker 来执行，并在其中一个 Worker 完成后将该任务视作已完成。

1.3.3 Locality

在 Google 内部所使用的计算环境中，机器间的网络带宽是稀缺的资源，设计时需要尽量减少在机器间过多地进行不必要的数据传输。

Google MapReduce 使用 Google File System 来存储输入源和输出结果。作为优化，Master 在分配 Map 任务时会感知 Google File System 中各个 Block 的位置信息，并尽量将对应的 Map 任务分配到持有或者接近该 Block数据的机器上。

1.3.4 Task Granularity

由于使用的机器是普通的计算机所以网络带宽是很缺乏的，MapReduce为了能处理这个问题使用了两办法，第一是每份要处理的数据被分的比较小只有64MB传输比较方便，并且MapReduce 的 master 在调度 Map 任务时会考虑输入文件的位置信息，尽量将一个 Map 任务调度在包含相关输入数据拷贝的机器上执行。这样大部分的数据都会从本地的机器读取，因此消耗非常少的网络带宽。

1.3.5 Backup Tasks

影响一个 MapReduce 的总执行时间最通常的因素是“落伍者”：在运算过程中，如果有一台机器花了很长的时间才完成最后几个 Map 或 Reduce 任务，导致 MapReduce 操作总的执行时间超过预期。而MapReduce使用一个在任务即将完成时启动备用的任务进程来执行剩下的，处于处理中的任务，而对于这些有多个进程的任务只要有一个完成了都会被标记为完成。

1.4 Refinement

- **Shuffle：**

  将maptask输出的处理结果数据，分发给reducetask，并在分发的过程中，对数据按key进行了分区和排序，这个过程称为shuffle。完整的shuffle是由Map端和Reduce端组成，Map端负责数据的溢写，Reduce负责将Map的数据拷贝到本地，并进行归并排序。

1.4.1 Partitioning Function

一个缺省的分区函数是使用 hash 方法(比如，hash(key) mod R)进行分区。hash 方法能产生非常平衡的分区。然而，有的时候，其它的一些分区函数对 key 值进行的分区将非常有用。比如，输出的 key 值是 URLs，希望每个主机的所有条目保持在同一个输出文件中。 为了支持类似的情况， MapReduce库的用户需要提供专门的分区函数。 例如， 使用 “hash(Hostname(urlkey)) mod R”作为分区函数就可以把所有来自同一个主机的 URLs 保存在同一个输出文件中。

1.4.2 Ordering Guarantee

MapReduce确保在给定的分区中，中间 key/value pair 数据的处理顺序是按照 key 值增量顺序处理的。这样的顺序保证对每个分成生成一个有序的输出文件，这对于需要对输出文件按 key 值随机存取的应用非常有意义，对在排序输出的数据集也很有帮助。

1.4.3 Combiner Funtion

Map端本地的Reduce，对Map的数据进行合并排序，减少HTTP进行文件的操作，用户可以自定义combiner。在WordCount开启combiner，在Map端处理数据后，对相同的Key先进行Reduce操作，先对Map中的数据进行求和，在发送到reduce端。Combiner适合满足结合律的数据（求和、求最大值），不适用对于求平均数。

在某些情形下，用户所定义的 Map 任务可能会产生大量重复的中间结果键，同时用户所定义的 Reduce 函数本身也是满足交换律和结合律的。

在这种情况下，Google MapReduce 系统允许用户声明在 Mapper 上执行的 Combiner 函数：Mapper 会使用由自己输出的 RR 个中间结果 Partition 调用 Combiner 函数以对中间结果进行局部合并，减少 Mapper 和 Reducer 间需要传输的数据量。

1.4.4 Input & Output Type

MapReduce库支持几种不同的格式的输入数据。比如，文本模式的输入数据的每一行被视为是一个 key/value pair。key 是文件的偏移量，value 是那一行的内容。另外一种常见的格式是以 key 进行排序来存储的 key/value pair 的序列。每种输入类型的实现都必须能够把输入数据分割成数据片段，该数据片段能够由单独的 Map 任务来进行后续处理(例如，文本模式的范围分割必须确保仅仅在每行的边界进行范围分割)。虽然大多数 MapReduce 的使用者仅仅使用很少的预定义输入类型就满足要求了，但是使用者依然可以通过提供一个简单的 Reader 接口实现就能够支持一个新的输入类型。

Reader 并非一定要从文件中读取数据，比如，我们可以很容易的实现一个从数据库里读记录的 Reader，或者从内存中的数据结构读取数据的 Reader。

类似的，MapReduce提供了一些预定义的输出数据的类型，通过这些预定义类型能够产生不同格式的数据。用户采用类似添加新的输入数据类型的方式增加新的输出类型。

1.4.5 Side Effect

在某些情况下，MapReduce 的使用者发现，如果在 Map 和/或 Reduce 操作过程中增加辅助的输出文件会比较省事。我们依靠程序 writer 把这种“副作用”变成原子的和幂等的。通常应用程序首先把输出结果写到 一个临时文件中，在输出全部数据之后，在使用系统级的原子操作 rename 重新命名这个临时文件。

如果一个任务产生了多个输出文件，我们没有提供类似两阶段提交的原子操作支持这种情况。因此，对于会产生多个输出文件、并且对于跨文件有一致性要求的任务，都必须是确定性的任务。但是在实际应用过程中，这个限制还没有给我们带来过麻烦。

1.5 Performance

一个大型集群上运行的两个计算来衡量MapReduce的性能。一个计算在大约1TB的数据中进行特定的模式匹配，另一个计算对大约1TB的数据进行排序。这两个程序在大量的使用MapReduce的实际应用中是非常典型的——类是对数据格式进行转换，从一种表现形式转换为另外一种表现形式:另一类是从海量数据中抽取少部分的用户感兴趣的数据。

查询相关资料发现有人反映：“关闭了备用任务排序程序后程序的执行情况是在960秒之后只有5个Reduce任务没有完成但是这五个任务又执行了300秒才完成，导致整个计算消耗了1283秒，多了44%的执行时间。这是很可怕的，因为五个任务而浪费掉了300秒的时间，所以说备用任务的设置是很有必要的。”

1.6 Experience

1. 大规模机器学习问题。
2. Google News和Froogle产品的集群问题。
3. 从公众查询产品 (比如Google的Zeigeist)的报告中抽取数据。
4. 从大量的新应用 和新产品的网页中提取有用信息(比如，从大量的位置搜索网页中抽取地理位置信息)。
5. 大规模的图形计算。

1.7 Related Work

MapReduce编程模型在Google内部成功应用于多个领域。原因有这几个方面:首先，由于MapReduce封装了并行处理、容错处理、数据本地化优化、负载均衡等等技术难点的细节，这使得MapReduce库易于使用。即便对于完全没有并行或者分布式系统开发经验的程序员而言，其次，大量不同类型的问题都可以通过MapReduce简单的解决。比如，MapReduce用于生成Google的网络搜索服务所需要的数据、用来排序、用来数据挖掘、用于机器学习，以及很多其它的系统;第三，我们实现了一个在数千台计算机组成的大型集群，上灵活部署运行的MapReduce。这个实现使得有效利用这些丰富的计算资源变得非常简单，因此也适合用来解决Google遇到的其他很多需要大量计算的问题。

2. The Google File System

本部分为对TheGoogleFileSystem论文的读后感兼阅读笔记。

2.1 Introduction

> GFS:
>
> GFS阐述了Google File System的设计原理，GFS是一个面向大规模数据:密集型应用的、可伸缩的分布式文件系统。GFS虽然运行在廉价的普遍硬件设备上，但是它依然了提供灾难冗余的能力，为大量客户机提供了高性能的服务。
>
> 虽然GFS的设计目标与许多传统的分布式文件系统有很多相同之处，但是，我们设计还是以我们对自己的应用的负载情况和技术环境的分析为基础的，不管现在还是将来，GFS和早期的分布式文件系统的设想都有明显的不同。所以我们重新审视了传统文件系统在设计上的折衷选择，衍生出了完全不同的设计思路。
>
> GFS完全满足了我们对存储的需求。GFS作为存储平台已经被广泛的部署在Google内部，存储我们的服务产生和处理的数据，同时还用于那些需要大规模数据集的研究和开发工作。目前为止，最大的一个集群利用数千台机器的数千个硬盘，提供了数百TB的存储空间，同时为数百个客户机服务。为了满足Google迅速增长的数据处理需求，我们设计并实现了Google文件系统(Google File System - GFS)。
>
> GFS与传统的分布式文件系统有着很多相同的设计目标，比如，性能、可伸缩性、可靠性以及可用性。但是，我们的设计还基于我们对我们自己的应用的负载情况和技术环境的观察的影响，不管现在还是将来，GFS和早期文件系统的假设都有明显的不同。所以我们重新审视了传统文件系统在设计上的折衷选择，衍生出了完全不同的设计思路。

2.2 Desgin Overview

2.2.1 Expectations

1. 这个系统由许多廉价易损的普通组件组成。它必须持续监视自己的状态，它必须在组件失效作为一种常态的情况下，迅速地侦测、承担并恢复那些组件失效。
2. 这个系统保存一定数量的大文件。我们预期有几百万文件，尺寸通常是100MB或者以上。数GB的文件也很寻常，而且被有效的管理。小文件必须支持，但是不需要去优化。
3. 负载中主要包含两种读操作：大规模的流式读取和小规模随机读取。大规模的流式读取通常一次操作就读取数百K数据，更常见的是一次性读取1MB甚至等多。同一个客户机的连续操作通常是对一个文件的某个区域进行连续读取。小规模的随机读取通常是在随机的位置读取几个KB。对性能有所要求的程序通常把小规模的读批量处理并且排序，这样就不需要对文件进行时前时后的读取，提高了对文件读取的顺序性。
4. 负载中还包括许多大规模的顺序的写操作，追加数据到文件尾部。一般来说这些写操作跟大规模读的尺寸类似。数据一旦被写入，文件就几乎不会被修改了。系统对文件的随机位置写入操作是支持的，但是不必有效率。
5. 系统必须高效的实现良好定义的多客户端并行追加到一个文件的语意。我们的文件经常用于"生产者-消费者"队列，或者多路文件合并。数百个生产者，一个机器一个，同时的对一个文件进行追加。用最小的同步开销实现追加的原子操作是非常重要的。文件可能稍后被读取，也可能一个消费者同步的读取文件。
6. 高度可用的带宽比低延迟更加重要。大多数我们的目标程序，在高传输速率下，大块地操作数据，因为大部分单独的读写操作没有严格的响应时间要求。

2.2.2 Systematic Architecture

GFS主要由以下三个系统模块组成：

- Master：管理元数据、整体协调系统活动
- ChunkServer：存储维护数据块（Chunk），读写文件数据
- Client：向Master请求元数据，并根据元数据访问对应ChunkServer的Chunk

2.3 System Interactions

系统交互的最重要原则就是减少Master节点的参与。

2.3.1 Lease and Mutation Order

变更是一个会改变Chunk内容或者元数据的操作，比如写入操作或者记录追加操作。变更操作会在Chunk的所有副本上执行。GFS使用租约（lease）机制来保持多个副本间变更顺序的一致性。Master收到变更操作请求后：

- 选择一个Chunk副本发放定时租约作为主Chunk并返回给客户端；
- 客户端与主Chunk进行通信进行变更操作；
- 租约超时前，对该Chunk的变更操作都由主Chunk进行序列化控制。

2.3.2 Data Flow

为了提高网络效率，我们采取了把数据流和控制流分开的措施。在控制流从客户机到主Chunk、然后再到所有二级副本的同时，数据以管道的方式，顺序的沿着一个精心选择的Chunk服务器链推送。我们的目标是充分利用每台机器的带宽，避免网络瓶颈和高延时的连接，最小化推送所有数据的延时。 为了充分利用每台机器的带宽，数据沿着一个Chunk服务器链顺序的推送，而不是以其它拓扑形式分散推送（例如，树型拓扑结构）。线性推送模式下，每台机器所有的出口带宽都用于以最快的速度传输数据，而不是在多个接受者之间分配带宽。

为了尽可能的避免出现网络瓶颈和高延迟的链接（例如，inter-switch最有可能出现类似问题），每台机器都尽量的在网络拓扑中选择一台还没有接收到数据的、离自己最近的机器作为目标推送数据。假设客户机把数据从Chunk服务器S1推送到S4。它把数据推送到最近的Chunk服务器S1。S1把数据推送到S2，因为S2和S4中最接近的机器是S2。同样的，S2把数据传递给S3和S4之间更近的机器，依次类推推送下去。我们的网络拓扑非常简单，通过IP地址就可以计算出节点的“距离”。

最后，我们利用基于TCP连接的、管道式数据推送方式来最小化延迟。Chunk服务器接收到数据后，马上开始向前推送。管道方式的数据推送对我们帮助很大，因为我们采用全双工的交换网络。接收到数据后立刻向前推送不会降低接收的速度。在没有网络拥塞的情况下，传送B字节的数据到R个副本的理想时间是B/T+RL，T是网络的吞吐量，L是在两台机器数据传输的延迟。通常情况下，我们的网络连接速度是100Mbps（T），L将远小于1ms。因此，1MB的数据在理想情况下80ms左右就能分发出去。

2.3.3 Atomic Records Append

GFS提供了一种原子的数据追加操作–记录追加。传统方式的写入操作，客户程序会指定数据写入的偏移量。对同一个region的并行写入操作不是串行的：region尾部可能会包含多个不同客户机写入的数据片段。使用记录追加，客户机只需要指定要写入的数据。GFS保证至少有一次原子的写入操作成功执行（即写入一个顺序的byte流），写入的数据追加到GFS指定的偏移位置上，之后GFS返回这个偏移量给客户机。这类似于在Unix操作系统编程环境中，对以O_APPEND模式打开的文件，多个并发写操作在没有竞态条件时的行为。

2.3.4 Snapshot

GFS中生成快照的方式叫copy-on-write。也就是说，文件的备份在某些时候只是将快照文件指向原chunk，增加对chunk的引用计数而已，等到chunk上进行了写操作时，Chunk Server才会拷贝chunk块，后续的修改操作落到新生成的chunk上。快照实现的过程：

- 收回文件所有Chunk的租约；
- 操作元数据完成元数据拷贝；
- 客户端要写入该文件的Chunk时，Master通知该Chunk所在ChunkServer进行本地拷贝；

2.4 Master Operation

2.4.1 Namespace Managment and Locking

多操作并行，名称空间锁保证执行顺序，文件操作需获得父目录读锁和目标文件/目录写锁。

不同于许多传统文件系统，GFS没有针对每个目录实现能够列出目录下所有文件的数据结构。GFS也不支持文件或者目录的链接（即Unix术语中的硬链接或者符号链接）。在逻辑上，GFS的名称空间就是一个全路径和元数据映射关系的查找表。利用前缀压缩，这个表可以高效的存储在内存中。在存储名称空间的树型结构上，每个节点（绝对路径的文件名或绝对路径的目录名）都有一个关联的读写锁。

2.4.2 Replica Replacement

Chunk跨机架分布：

- 好处：
  - 防止整个机架破坏造成数据失效；
  - 综合利用多机架整合带宽（机架出入带宽可能小于机架上机器的总带宽，所以应最大化每台机架的带宽利用率）；
- 坏处：
  - 写操作需跨机架通信。

2.4.3 Chunk Management

Chunk的副本有三个用途：Chunk创建，重新复制和重新负载均衡。

1. Chunk的创建操作，主要考虑：
   - 平衡硬盘使用率；
   - 限制单ChunkServer短期创建次数（创建开销虽小，但每次创建往往意味着大量的后续写入）；
   - 跨机架分布。
2. 重复制，即有效副本不足时，通过复制增加副本数。优先考虑：
   - 副本数量和复制因数相差多的；
   - live（未被删除）文件的；
   - 阻塞客户机处理的

   Chunk进行重复制。策略与创建类似。为了防止克隆产生的网络流量大大超过客户机的流量，Master节点对整个集群和每个Chunk服务器上的同时进行的克隆操作的数量都进行了限制。另外，Chunk服务器通过调节它对源Chunk服务器读请求的频率来限制它用于克隆操作的带宽。
3. 重负载均衡，通过调整副本位置，平衡格机负载。策略与创建类似。新ChunkServer将被逐渐填满。

2.5 Fault Tolerance & Diagnosis

2.5.1 High Availability

主要的提高可用性的机制：

- 组件快速恢复；
- Chunk复制；
- Master服务器复制；
- Checkpoint和操作日志多机备份；
- Master进程失效重启，硬件失效则新机器重启Master进程；
- “影子”Master，通过操作日志“模仿”主Master操作，元数据版本稍慢。作用 - 分担一定负载、失效期暂时接管。

2.5.2 Data Integrity

ChunkServer独立维护CheckSum检验副本完整性。原因：

- 跨Chunk服务器比较副本开销大；
- 追加操作造成的的byte级别不一致，导致无法通过比较副本判断完整性。
- Chunk读取和Chunk服务器空闲时，进行CheckSum校验，发现损坏Chunk上报Master，进行重复制。
- Checksum校验的开销：
- Checksum读取开销；
- Checksum校验计算开销。

但整体开销可以接受，特别是对追加写操作。

覆盖写与追加写的Checksum计算开销比较。两者的关键区别在于不完整块的CheckSum计算：

- 追加写 - 对最后一个不完整块，在写入后进行增量的CheckSum计算。New CheckSum由Old CheckSum和新数据算出，未对原有数据重新计算，原有数据损坏，依然可以发现；
- 覆盖写 - 对第一个和最后一个不完整块，在写之前进行CheckSum校验。否则，覆盖写只能重新对整块计算CheckSum，若写操作未覆盖部分存在损坏数据，新CheckSum将从包含损坏数据的Chunk算出，之后再也无法校验出该损坏数据。

3. FastSGG

本部分为对TheGoogleFileSystem论文的读后感兼阅读笔记。

> FastSGG是一种高效且广泛应用的社交图生成器FastSGG。
>
> FastSGG根据用户定义的描述目标社交图特性的配置生成社交图，这是在各种应用程序中生成图的一种灵活方式。生成方法主要包括两个步骤:
>
> - 确定源顶点的出度
> - 确定目标顶点构造边；
>
> 为了加速图的生成过程，提出了一种度分布生成(d2g)模型。d2 G模型是在给定概率密度函数或概率质量函数的情况下，生成不同程度分布图的通用模型。大量的实验结果表明，FastSGG能够生成具有小世界属性、幂律度分布和社区结构的高质量社会图。此外，FastSGG生成图形的速度至少比最先进的图形生成器快4倍。此外，FastSGG的峰值内存使用量还不到最先进方法的七分之一。