**Google三大论文读后感**

**——18301092 陈佳林**

Google的三驾马车：Google fs，Mapreduce 和 Bigtable, 这三个产品Google并没有公布相关源码，但是发布了这三个产品的详细设计论文。Yahoo资助的Hadoop也有按照这三篇论文的开源Java实现:Hadoop对应Mapreduce, Hadoop Distributed File System (HDFS)对应Google fs, Hbase对应Bigtable。不过在性能上Hadoop比Google要差很多。

**PART I Bigtable**

Bigtable是一个管理结构化数据的分布式存储系统，它被设计用来处理海量数据。Bigtable的设计目的是可靠地适应PB级别的数据和成千上万台机器。Bigtable已经实现了下面的几个目标：广泛的适用性、可扩展、高性能和高可用性。

在很多方面，Bigtable和数据库很类似：它使用了很多数据库的实现策略。并行数据库和内存数据库已经具备可扩展性和高性能，但是Bigtable提供了一个和这些系统完全不同的接口。

Bigtable不支持完整的关系数据模型；与之相反，Bigtable为客户提供了简单的数据模型，利用这个模型，客户可以动态控制数据的布局和格式，也就是对BigTable而言，数据是没有格式的，用数据库领域的术语说，就是数据没有Schema，用户自己去定义Schema，用户也可以自己推测在底层存储中展示的数据的位置属性，比如具有相同前缀的数据存放的位置接近，在读取的时候可以将这些数据一次性读出来。数据用行和列的名字进行索引，名字可以是任意的字符串。虽然客户程序通常会在把各种结构化或半结构化的数据串行化到字符串里，Bigtable同样将数据视为未经解析的字符串。通过仔细选择数据的模式，客户可以控制数据的位置。最后，可以通过BigTable的模式参数动态地控制数据读或写。

Bigtable是一个稀疏的、分布式的、持久化存储的多维度排序Map，Map由行关键字、列关键字以及时间戳索引；Map中的每个value都是一个未经解析的字节数组。



表中的行关键字是任意字符串，在单一行关键字下的每一个读或者写操作都是原子的（不管在这一行里被读或者写的不同列的数目），这个设计决策能够使用户很容易地推测（reason about）对同一个行进行并发更新操作时的系统行为。

Bigtable通过行关键字的字典顺序来维护数据。表中一定范围内的行被动态分区。每个分区叫做一个”Tablet”，Tablet是数据分布和负载均衡的单位。这样做的结果是，读取一定范围内的少数行很高效，并且往往只需要跟少数机器通信。用户可以通过选择他们的行关键字来开发这种特性，这样可以为他们的数据访问获得好的本地性。

列关键字组成的集合叫做“列族“，列族构成了访问控制的基本单位。存放在同一列族下的所有数据通常都属于同一个类型（我们把同一个列族下的数据压缩在一起）。列族必须先创建，然后才能在列族中任何的列关键字下存放数据；列族创建后，其中的任何一个列关键字下都可以存放数据。我们的意图是，一张表中不同列族的数目要小（最多几百个），并且列族在操作中很少改变。与此相反，一张表可以有无限多个列。

列关键字的命名语法如下：列族：限定词。访问控制、磁盘和内存的计数都是在列族层面进行的。上述的控制权限能帮助我们管理不同类型的应用：一些应用可以添加新的基本数据、一些可以读取基本数据并创建派生的列族、一些则只允许浏览现存数据（甚至可能因为隐私的原因不能浏览所有现存列族）。

在Bigtable中，每一个数据项都可以包含同一数据的不同版本；这些版本通过时间戳来索引。Bigtable时间戳是64位整型数。时间戳可由Bigtable指定，这种情况下时间戳代表精确到毫秒的“实时”时间，或者该值由库户程序明确指定。需要避免冲突的程序必须自己生成一个唯一的时间戳。数据项中不同版本按照时间戳倒序排列，所以最新的版本可以被先读到。

Bigtable提供了建立和删除表以及列族的API函数。Bigtable还提供了修改集群、表和列族的元数据的API，比如修改访问权限。

Bigtable是建立在一些其他Google基础架构之上的。BigTable使用Google分布式文件系统(GFS)存储日志和数据文件。BigTable集群往往运行在一个共享的机器池中，池中的机器还会运行其它各种各样的分布式应用程序，BigTable的进程经常要和其它应用的进程共享机器。BigTable依赖集群管理系统在共享机器上调度作业、管理资源、处理机器的故障、以及监视机器的状态。

BigTable数据在内部使用Google SSTable文件格式存储。SSTable提供一个从键（key）到值（value）的持久化的、已排序、不可更改的映射（Map），这里的key和value 的都是任意的字节（Byte）串。对SSTable提供了如下操作：查询与一个指定key值相关的value，或者遍历指定key值范围内的所有键值对。从内部看，SSTable是一连串的数据块（通常每个块的大小是64KB，但是这个大小是可以配置的）。SSTable使用块索引（通常存储在SSTable 的最后）来定位数据块；在打开SSTable的时候，索引被加载到内存。一次查找可以通过一次磁盘搜索完成：首先执行二分查找在内存索引里找到合适数据块的位置，然后在从硬盘中读取合适的数据块。也可以选择把整个SSTable都映射到内存中，这样就可以在不用访问硬盘的情况下执行查询搜索了。

BigTable还依赖一个高可用的、持久化的分布式锁服务组件，叫做Chubby。一个Chubby服务包括了5 个活动的副本，其中一个副本被选为Master，并且积极处理请求。只有在大多数副本正常运行，并且彼此之间能够互相通信的情况下，Chubby服务才是可用的。当有副本失效的时候，出现故障时Chubby使用Paxos算法保证副本的一致性。Chubby提供了一个名字空间，里面包括了目录和小文件。每个目录或者文件可以当成一个锁使用，对文件的读写操作都是原子的。Chubby客户程序库提供对Chubby文件的一致性缓存。每个Chubby客户程序都维护一个与Chubby服务的会话。如果客户程序不能在租约到期的时间内重新签订会话租约，这个会话就过期失效了。当一个客户会话失效时，它拥有的锁和打开的文件句柄都失效了。Chubby客户程序可以在Chubby文件和目录上注册回调函数，当文件或目录改变、或者会话过期时，回调函数会通知客户程序。

Bigtable使用Chubby完成以下各种任务：保证在任意时间最多只有一个活动的Master；存储BigTable数据的引导程序的位置；发现tablet服务器，以及在Tablet服务器失效时进行善后；存储BigTable的模式信息（每张表的列族信息）；以及存储访问控制列表。如果Chubby长时间无法访问，BigTable就会失效。

Bigtable的实现有三个主要的组件：链接到每个客户程序的库、一个Master服务器和多个tablet服务器。在一个集群中可以动态地添加（或者删除）一个tablet服务器来适应工作负载的变化。

Master主要负责以下工作：为tablet服务器分配tablets，检测新加入的或者过期失效的tablet服务器、平衡tablet 服务器的负载、以及对GFS中的文件进行垃圾收集。除此之外，它还处理模式修改操作，例如建立表和列族。

每个Tablet服务器都管理一组tablet（通常每个tablet服务器有大约数十个至上千个tablet）。tablet服务器处理它所加载的tablet的读写操作，以及分割增长的过大的tablet。

和很多单主节点（Single-Master）类型的分布式存储系统类似，客户数据都不经过master服务器：客户程序直接和tablet服务器通信来进行读写操作。由于BigTable的客户程序不依赖master服务器来获取tablet的位置信息，大多数客户程序甚至完全不和master通信。因此，在实际应用中master的负载是很轻的。

一个BigTable集群存储了很多表，每个表包含了一组tablet，而每个tablet包含了某个范围内的行的所有相关数据。初始状态下，每个表只有一个tablet组成。随着表中数据的增长，它被自动分割成多个tablet，默认情况下每个tablet的大小大约是100MB到200MB。

我们使用一个三层的、类似于Ｂ+树的结构存储tablet的位置信息。

第一层是一个存储在Chubby中的文件，它包含了root tablet的位置信息。root tablet在一个特殊的元数据（METADATA）表包含了里所有的tablet的位置信息。每一个元数据tablet包含了一组用户tablet的位置信息。root tablet实际上只是元数据表的第一个tablet，只不过对它的处理比较特殊—root tablet永远不会被分割—这就保证了tablet的位置层次不会超过三层。

元数据表将每个tablet的位置信息存储在一个行关键字下，而这个行关键字是由tablet所在的表的标识符和tablet的最后一行编码而成的。每一个元数据行在内存中大约存储了1KB数据。在一个大小适中的、大小限制为128MB的元数据 tablet中，我们的三层结构位置信息模式足够寻址234（三层27+10\*27+10）个tablet（或者说在128M的元数据中可以存储261个字节）。

客户程序库会缓存tablet的位置信息。如果客户程序不知道一个tablet的位置信息，或者发现它缓存的地址信息不正确，那么客户程序就递归移动到tablet位置层次；如果客户端缓存是空的，那么寻址算法需要通过三次网络来回通信寻址，这其中包括了一次Chubby读操作。如果客户端缓存的地址信息过期了，那么寻址算法可能进行多达6次（其中的三次通信发现缓存过期，另外三次更新缓存数据）网络来回通信，因为过期缓存条目只有在没有查到数据（upon misses）的时候才能发现 (假设元数据tablet没有被频繁的移动）。尽管tablet的位置信息是存放在内存里的，所以不需访问GFS，但是，通常我们会通过预取tablet地址来进一步的减少访问开销：无论何时读取元数据表，都会为不止一个tablet读取元数据。

在元数据表中还存储了次级信息(secondary information)，包括与tablet有关的所有事件日志（例如，什么时候一个服务器开始为该tablet提供服务）。这些信息有助于排除故障和性能分析。

每个tablet一次分配给一个tablet服务器。master服务器记录活跃的tablet服务器、当前tablet到tablet服务器的分配、包括哪些tablet还没有被分配。当一个tablet还没有被分配、并且有一个tablet服务器有足够的空闲空间来装载该tablet并且可用，master通过给这个tablet服务器发送一个tablet装载请求分配该tablet。

BigTable使用Chubby跟踪记录tablet服务器。当一个tablet服务器启动时，它在一个指定的Chubby目录下建立一个有唯一名字的文件，并且获取该文件的独占锁。master监控着这个目录（服务器目录）。如果tablet服务器失去了Chubby上的独占锁—比如由于网络断开导致tablet服务器丢失Chubby会话—它就停止对tablet提供服务。（Chubby提供了一种高效的机制，利用这种机制，tablet服务器能够在不招致网络拥堵的情况下检查其是否还持有该锁）。只要该文件还存在，tablet服务器就会试图重新获得对该独占锁；如果文件不存在了，那么tablet服务器就永远不能再提供服务了，它会自行退出。只要tablet 服务器终止（比如，集群的管理系统将该tablet服务器的主机从集群中移除），它会尝试释放它持有的锁，以便master尽快重新分配它的tablet。

Master负责探测一个tablet服务器何时不再为它的tablet提供服务，并且尽快重新分配那些tablet。master通过轮询tablet服务器锁的状态来探测tablet服务器何时不再为tablet提供服务。如果一个tablet服务器报告它丢失了锁，或者master最近几次尝试都无法和该服务器通信，master就会尝试获取该tablet服务器文件的独占锁；如果master能够获取独占锁，那么就说明Chubby是正常运行的，而tablet 服务器要么是宕机了、要么是不能和Chubby通信了，因此，为了保证该tablet服务器不能再提供服，master就删除该tablet服务器在Chubby上的服务器文件。一旦服务器文件被删除了，master就把之前分配给该服务器的所有的tablet放入未分配的tablet集合中。为了确保Bigtable集群面对master和Chubby之间网络问题不那么脆弱，master在它的Chubby会话过期时会主动退出。但是不管怎样，如上所述，master的故障不会改变现有tablet到tablet服务器的分配。

当集群管理系统启动了一个master之后，master首先要了解当前tablet的分配状态，之后才能够修改它们。master在启动的时候执行以下步骤：（1）master在Chubby中获取一个唯一的master锁，用来阻止并发的master实例；（2）master扫描Chubby的服务器目录，获取寻找正在运行的服务器；（3）master和每一个正在运行的tablet服务器通信，搜寻哪些tablet已经分配到了tablet服务器中；（4）master服务器扫描元数据表获取tablet的集合。只要扫描发现了一个还没有分配的tablet，master就将这个tablet加入未分配的tablet 集合，该集合使该tablet有机会参与tablet分配。

有一种复杂情况是：元数据tablet还没有被分配之前是不能够扫描它的。因此，在开始扫描之前，如果在第三步中没有发现对root tablet的分配，master就把root tablet加入到未分配的tablet集合中。这个附加操作确保了root tablet会被分配。由于root tablet包括了所有元数据tablet的名字，master在扫描完root tablet以后才了解所有元数据信息。

现存tablet的集合只有在以下事件发生时才会改变：建立了一个新表或者删除了一个旧表、两个现存tablet合并组成一个大的tablet、或者一个现存tablet被分割成两个小的tablet。master可以跟踪这些改变，因为除了最后一个事件外的两个事件都是由它初始化的。tablet分割事件需要特殊处理，因为它是由tablet服务器初始化的。tablet服务器通过在元数据表中为新的tablet记录信息的方式提交分割操作。在分割操作提交之后tablet服务器会通知master。假如分割操作通知丢失（tablet服务器或者master宕机），master在请求tablet服务器装载已经被分割的tablet的时候会探测到一个新的tablet。由于在元数据tablet中发现的tablet条目只是列举了master请求加载的tablet的一部分，tablet服务器会通知master分割信息。

tablet的持久化状态信息保存在GFS上。更新操作提交到存储撤销(REDO)记录的提交日志中。在这些更新操作中，最近提交的那些存放在一个叫做memtable的排序的缓冲区中；较早的更新存放在一系列SSTable中。为了恢复一个tablet，tablet服务器在元数据表中读取它的元数据。这些元数据包含组成一个tablet的SSTable列表和一组还原点（redo points），这些点是指向包含tablet数据的任一提交日志的指针。tablet服务器把SSTable的索引读进内存，之后通过应用还原点之后提交的所有更新来重构memtable。

当写操作到达tablet服务器时，tablet服务器首先要检查这个操作格式是否正确、发送者是否有执行这个改变的权限。权限验证是通过从一个Chubby文件里读取具有写权限的操作者列表来进行的（这个文件几乎总会在Chubby客户缓存里命中）。有效的修改操作会记录在提交日志里。可以采用组提交方式来提高大量小的修改操作的吞吐量。当一个写操作提交后，它的内容被插入到memtable里面。

当读操作到达tablet服务器时，它同样会检查良构性和适当的权限。一个有效的读操作在一个由一系列SSTable和memtable合并的视图里执行。由于SSTable和memtable是按字典排序的数据结构，因此可以高效生成合并视图。

当进行tablet的合并和分割时，引入（incoming）的读写操作能够继续进行。

随着写操作的执行，memtable的大小不断增加。当memtable的尺寸到达一个临界值的时候，这个memtable就会被冻结，然后创建一个新的memtable；被冻结住memtable会被转换成SSTable并写入GFS（我们称这种Compaction行为为Minor Compaction）。Minor Compaction过程有两个目的：一是收缩tablet服务器内存使用，二是在服务器灾难恢复过程中，减少必须从提交日志里读取的数据量。在Compaction过程中，引入（incoming）的读写操作仍能继续。

**PART II 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GFS和早期文件系统有明显的不同。在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。

首先，组件失效被认为是常态事件，而不是意外事件。GFS包括几百甚至几千台普通的廉价设备组装的存储机器，同时被相当数量的客户机访问。GFS组件的数量和质量导致在事实上，任何给定时间内都有可能发生某些组件无法工作，某些组件无法从它们目前的失效状态中恢复。造成组件失效的原因包括应用程序bug、操作系统的bug、人为失误，甚至还有硬盘、内存、连接器、网络以及电源失效等。所以，持续的监控、错误侦测、灾难冗余以及自动恢复的机制必须集成在GFS中。

其次，以通常的标准衡量，我们的文件非常巨大。数GB的文件非常普遍。每个文件通常都包含许多应用程序对象，比如web文档。当我们经常需要处理快速增长的、并且由数亿个对象构成的、数以TB的数据集时，采用管理数亿个KB大小的小文件的方式是非常不明智的，尽管有些文件系统支持这样的管理方式。因此，设计的假设条件和参数，比如I/O操作和Block的尺寸都需要重新考虑。

第三，绝大部分文件的修改是采用在文件尾部追加数据，而不是覆盖原有数据的方式。对文件的随机写入操作在实际中几乎不存在。一旦写完之后，对文件的操作就只有读，而且通常是按顺序读。大量的数据符合这些特性，比如：数据分析程序扫描的超大的数据集；正在运行的应用程序生成的连续的数据流；存档的数据；由一台机器生成、另外一台机器处理的中间数据，这些中间数据的处理可能是同时进行的、也可能是后续才处理的。对于这种针对海量文件的访问模式，客户端对数据块缓存是没有意义的，数据的追加操作是性能优化和原子性保证的主要考量因素。

第四，应用程序和文件系统API的协同设计提高了整个系统的灵活性。比如，我们放松了对GFS一致性模型的要求，这样就减轻了文件系统对应用程序的苛刻要求，大大简化了GFS的设计。我们引入了原子性的记录追加操作，从而保证多个客户端能够同时进行追加操作，不需要额外的同步操作来保证数据的一致性。本文后面还有对这些问题的细节的详细讨论。

一个GFS集群包含一个单独的Master节点、多台Chunk服务器，并且同时被多个客户端访问。所有的这些机器通常都是普通的Linux机器，运行着用户级别(user-level)的服务进程。我们可以很容易的把Chunk服务器和客户端都放在同一台机器上，前提是机器资源允许，并且我们能够接受不可靠的应用程序代码带来的稳定性降低的风险。

GFS存储的文件都被分割成固定大小的Chunk。在Chunk创建的时候，Master服务器会给每个Chunk分配一个不变的、全球唯一的64位的Chunk标识。Chunk服务器把Chunk以linux文件的形式保存在本地硬盘上，并且根据指定的Chunk标识和字节范围来读写块数据。出于可靠性的考虑，每个块都会复制到多个块服务器上。缺省情况下，我们使用3个存储复制节点，不过用户可以为不同的文件命名空间设定不同的复制级别。

Master节点管理所有的文件系统元数据。这些元数据包括名字空间、访问控制信息、文件和Chunk的映射信息、以及当前Chunk的位置信息。Master节点还管理着系统范围内的活动，比如，Chunk租用管理、orphaned chunks的回收、以及Chunk在Chunk服务器之间的迁移。Master节点使用心跳信息周期地和每个Chunk服务器通讯，发送指令到各个Chunk服务器并接收Chunk服务器的状态信息。

GFS客户端代码以库的形式被链接到客户程序里。客户端代码实现了GFS文件系统的API接口函数、应用程序与Master节点和Chunk服务器通讯、以及对数据进行读写操作。客户端和Master节点的通信只获取元数据，所有的数据操作都是由客户端直接和Chunk服务器进行交互的。我们不提供POSIX标准的API的功能，因此，GFS API调用不需要深入到Linux vnode级别。

无论是客户端还是Chunk服务器都不需要缓存文件数据。客户端缓存数据几乎没有什么用处，因为大部分程序要么以流的方式读取一个巨大文件，要么工作集太大根本无法被缓存。无需考虑缓存相关的问题也简化了客户端和整个系统的设计和实现。（不过，客户端会缓存元数据。）Chunk服务器不需要缓存文件数据的原因是，Chunk以本地文件的方式保存，Linux操作系统的文件系统缓存会把经常访问的数据缓存在内存中。

单一的Master节点的策略大大简化了我们的设计。单一的Master节点可以通过全局的信息精确定位Chunk的位置以及进行复制决策。另外，我们必须减少对Master节点的读写，避免Master节点成为系统的瓶颈。客户端并不通过Master节点读写文件数据。反之，客户端向Master节点询问它应该联系的Chunk服务器。客户端将这些元数据信息缓存一段时间，后续的操作将直接和Chunk服务器进行数据读写操作。

一次简单读取的流程如下。首先，客户端把文件名和程序指定的字节偏移，根据固定的Chunk大小，转换成文件的Chunk索引。然后，它把文件名和Chunk索引发送给Master节点。Master节点将相应的Chunk标识和副本的位置信息发还给客户端。客户端用文件名和Chunk索引作为key缓存这些信息。之后客户端发送请求到其中的一个副本处，一般会选择最近的。请求信息包含了Chunk的标识和字节范围。在对这个Chunk的后续读取操作中，客户端不必再和Master节点通讯了，除非缓存的元数据信息过期或者文件被重新打开。实际上，客户端通常会在一次请求中查询多个Chunk信息，Master节点的回应也可能包含了紧跟着这些被请求的Chunk后面的Chunk的信息。在实际应用中，这些额外的信息在没有任何代价的情况下，避免了客户端和Master节点未来可能会发生的几次通讯。

Chunk选择了64MB大小，这个尺寸远远大于一般文件系统的Block size。每个Chunk的副本都以普通Linux文件的形式保存在Chunk服务器上，只有在需要的时候才扩大。惰性空间分配策略避免了因内部碎片造成的空间浪费，内部碎片或许是对选择这么大的Chunk尺寸最具争议一点。

选择较大的Chunk尺寸有几个重要的优点。首先，它减少了客户端和Master节点通讯的需求，因为只需要一次和Mater节点的通信就可以获取Chunk的位置信息，之后就可以对同一个Chunk进行多次的读写操作。这种方式对降低我们的工作负载来说效果显著，因为我们的应用程序通常是连续读写大文件。即使是小规模的随机读取，采用较大的Chunk尺寸也带来明显的好处，客户端可以轻松的缓存一个数TB的工作数据集所有的Chunk位置信息。其次，采用较大的Chunk尺寸，客户端能够对一个块进行多次操作，这样就可以通过与Chunk服务器保持较长时间的TCP连接来减少网络负载。第三，选用较大的Chunk尺寸减少了Master节点需要保存的元数据的数量。这就允许我们把元数据全部放在内存中。

另一方面，即使配合惰性空间分配，采用较大的Chunk尺寸也有其缺陷。小文件包含较少的Chunk，甚至只有一个Chunk。当有许多的客户端对同一个小文件进行多次的访问时，存储这些Chunk的Chunk服务器就会变成热点。一个可能的长效解决方案是，在这种的情况下，允许客户端从其它客户端读取数据。

因为元数据保存在内存中，所以Master服务器的操作速度非常快。并且，Master服务器可以在后台简单而高效的周期性扫描自己保存的全部状态信息。这种周期性的状态扫描也用于实现Chunk垃圾收集、在Chunk服务器失效的时重新复制数据、通过Chunk的迁移实现跨Chunk服务器的负载均衡以及磁盘使用状况统计等功能。

将元数据全部保存在内存中的方法有潜在问题：Chunk的数量以及整个系统的承载能力都受限于Master服务器所拥有的内存大小。但是在实际应用中，这并不是一个严重的问题。Master服务器只需要不到64个字节的元数据就能够管理一个64MB的Chunk。由于大多数文件都包含多个Chunk，因此绝大多数Chunk都是满的，除了文件的最后一个Chunk是部分填充的。同样的，每个文件的在命名空间中的数据大小通常在64字节以下，因为保存的文件名是用前缀压缩算法压缩过的。

Master服务器并不保存持久化保存哪个Chunk服务器存有指定Chunk的副本的信息。Master服务器只是在启动的时候轮询Chunk服务器以获取这些信息。Master服务器能够保证它持有的信息始终是最新的，因为它控制了所有的Chunk位置的分配，而且通过周期性的心跳信息监控Chunk服务器的状态。

操作日志包含了关键的元数据变更历史记录。这对GFS非常重要。这不仅仅是因为操作日志是元数据唯一的持久化存储记录，它也作为判断同步操作顺序的逻辑时间基线。文件和Chunk，连同它们的版本，都由它们创建的逻辑时间唯一的、永久的标识。

操作日志非常重要，我们必须确保日志文件的完整，确保只有在元数据的变化被持久化后，日志才对客户端是可见的。否则，即使Chunk本身没有出现任何问题，我们仍有可能丢失整个文件系统，或者丢失客户端最近的操作。所以，我们会把日志复制到多台远程机器，并且只有把相应的日志记录写入到本地以及远程机器的硬盘后，才会响应客户端的操作请求。Master服务器会收集多个日志记录后批量处理，以减少写入磁盘和复制对系统整体性能的影响。

Master服务器在灾难恢复时，通过重演操作日志把文件系统恢复到最近的状态。为了缩短Master启动的时间，我们必须使日志足够小。Master服务器在日志增长到一定量时对系统状态做一次Checkpoint，将所有的状态数据写入一个Checkpoint文件。在灾难恢复的时候，Master服务器就通过从磁盘上读取这个Checkpoint文件，以及重演Checkpoint之后的有限个日志文件就能够恢复系统。Checkpoint文件以压缩B-树形势的数据结构存储，可以直接映射到内存，在用于命名空间查询时无需额外的解析。这大大提高了恢复速度，增强了可用性。

由于创建一个Checkpoint文件需要一定的时间，所以Master服务器的内部状态被组织为一种格式，这种格式要确保在Checkpoint过程中不会阻塞正在进行的修改操作。Master服务器使用独立的线程切换到新的日志文件和创建新的Checkpoint文件。新的Checkpoint文件包括切换前所有的修改。对于一个包含数百万个文件的集群，创建一个Checkpoint文件需要1分钟左右的时间。创建完成后，Checkpoint文件会被写入在本地和远程的硬盘里。

Master服务器恢复只需要最新的Checkpoint文件和后续的日志文件。旧的Checkpoint文件和日志文件可以被删除，但是为了应对灾难性的故障，我们通常会多保存一些历史文件。Checkpoint失败不会对正确性产生任何影响，因为恢复功能的代码可以检测并跳过没有完成的Checkpoint文件。

GFS一致性保障机制。文件命名空间的修改（例如，文件创建）是原子性的。它们仅由Master节点的控制：命名空间锁提供了原子性和正确性的保障；Master节点的操作日志定义了这些操作在全局的顺序。

**PART III MapReduce**

MapReduce是一种处理海量数据的并行编程模型，用于大规模数据集(通常大于1TB)的并行计算。以这种编程模型所编写的程序可以自动地在集群上并行执行，封装了并行计算、容错处理、数据存储、任务调度、任务间通信等细节，用户只需专心于并行程序的编写。MapReduce适用于复杂度不高的海量数据搜索、挖掘和分析。

MapReduce编程模型。MapReduce输入一个键值对，输出另一个键值对，用户则通过编写Map函数和Reduce函数来指定所要进行的计算。

Map：接受一个键值对(key-value pair)，产生一组中间键值对。MapReduce会将Map函数产生的中间键值对传递给一个Reduce函数。

Reduce：接受一个键以及相关的一组值，将这组值进行合并产生一组规模更小的值(通常只有一个或零个值)。

在实现的过程中，Reduce函数使用Iterator读取中间结果，为了防止值过多，而无法全部放到内存中。

MapReduce执行过程。

（1）用户程序将输入文件切分为M个16～64MB之间的split，并在集群中创建程序副本。

（2）程序副本分为1个Master和多个Worker。Master选取空闲的Worker，并为其分配Map任务或Reduce任务。Master存储Map任务和Reduce任务的状态(空闲，工作中，完成)和worker机器(非空闲任务的机器)的标识。

（3）Mapper读取对应的split并解析为键值对，调用用户定义的map函数进行处理，将输出的中间结果键值对存储到内存中。

（4）Mapper将缓存的键值对通过Partition函数(通常为hash(key) mod R)划分为R个部分，并周期性地写入本地磁盘，同时将本地磁盘上中间结果的位置信息发送给Master。

（5）Master将收到的中间结果的位置信息发送给Reducer，Reducer通过RPC读取存储在Mapper本地磁盘上的中间结果。在读取完毕后，Reducer会根据key对中间结果进行排序。

（6）对每一个key和其对应的一组值，调用用户定义的Reduce函数进行处理，输出结果存储在对应的Reduce Partition文件中。

（7）当所有的Map任务和Reduce任务完成后，Master通知用户程序。

MapReduce容错机制。由于MapReduce很大程度上利用了由Google File System提供的分布式原子文件读写操作，所以容错机制简洁很多，主要集中在任务意外中断的恢复上。worker失效情况，Master会周期性地ping每一个Worker，如果某个Worker在一段时间内没有响应，Master就会认为这个Worker已经不可用，对其上分配的任务重新分配给其他Worker。master失效情况下Master会周期性地将集群的当前状态作为checkpoint写入到磁盘中。Master发生故障后，重新启动Master即可利用存储在磁盘中的最近的checkpoint进行恢复。

数据存储机制。由于网络带宽是一种相当匮乏的计算资源，MapReduce将数据存储在本地磁盘，由GFS进行管理。GFS把每一份文件划分为多个64MB的Chunk，通常情况下，每个Chunk会在不同的机器上保存三份副本。Master在调度Map任务时会考虑输入数据的位置信息，采取就近原则，即尽量将Map任务调度到包含相关输入数据副本的机器上执行。因而大部分输入数据可以从本地读取，消耗非常少的网络带宽。

负载均衡机制。Master将Map任务和Reduce任务划分为M和R个片段，M和R的数量应远大于集群中Worker的数量。每个Worker执行大量不同的任务有助于提高集群的动态负载均衡能力，并且加快故障恢复的速度。

备份任务机制。如果集群中某个Worker花了很长时间才完成最后几个Map任务或Reduce任务，导致MapReduce总执行时间延长，这样的Worker被称为落后者(Straggler)。MapReduce提供了备份任务机制来缓解这种情况。当MapReduce快要完成时，Master为剩下的正在运行的任务启动备份任务，将其分配给其他的空闲Worker来执行，并在其中一个Worker完成后将该任务视作已完成。通过备份任务机制，大大减少了MapReduce的执行时间。