调研报告

小组成员

- 柯景瀚
- 吴骏
- 丁程
- 余丰
- 王腾岳
- wkdwy(蜗壳动物园)荣誉出品

目录

项目概述

在2021年OSH项目x-DisGraFS 的基础上，根据文件自然属性的逻辑关联，完善分布式文件系统的缓存机制，同时在索引和写入读取上进行优化，并且保证部署的稳定性和便捷性，以期望提高分布式文件系统的性能。

项目背景

日志型文件系统

日志文件系统（英语：Journaling file system）是一种文件系统。在发生变化时，它先把相关的信息写入一个被称为**日志**的区域，然后再把变化写入主文件系统。在文件系统发生故障（如内核崩溃或突然停电）时，日志文件系统更容易保持一致性，并且可以较快恢复。

对文件系统进行修改时，需要进行很多操作。这些操作可能中途被打断，也就是说，这些操作不是“不可中断”(atomic)的。如果操作被打断，就可能造成文件系统出现不一致的状态。

例如：删除文件时，先要从目录树中移除文件的标示，然后收回文件占用的空间。如果在这两步之间操作被打断，文件占用的空间就无法收回。文件系统认为它是被占用的，但实际上目录树中已经找不到使用它的文件了。

在非日志文件系统中，要检查并修复类似的错误就必须对整个文件系统的数据结构进行检查。一般在挂载文件系统前，操作系统会检查它上次是否被成功卸载，如果没有，就会对其进行检查。如果文件系统很大或者**I/O**有限，这个操作可能会花费很长时间。

为了避免这样的问题，日志文件系统分配了一个称为日志（journal）的区域来提前记录要对文件系统做的更改。在崩溃后，只要读取日志重新执行未完成的操作，文件系统就可以恢复一致。这种恢复是原子的，因为只存在几种情况：

- 不需要重新执行：这个事务被标记为已经完成
- 成功重新执行：根据日志，这个事务被重新执行
- 无法重新执行：这个事务会被撤销，就如同这个事务从来没有发生过
- 日志本身不完整：事务还没有被完全写入日志，它会被简单忽略

APFS

apfs是apple公司推出的取代HFS+的文件系统，它的性能优秀，并且采用了一种日志的机制来增改文件，作为重要的Crash protection的手段，它为了避免由系统崩溃而引起的元数据的损坏，它并不是在覆盖现有元数据的情况下进行数据的删改，而是编写日志记录，然后将指针指向新记录，释放旧记录，这样就避免了由于更新过程中发生的崩溃而导致的包含部分旧数据和部分新数据的损坏记录。它的这种方法同时还避免了不得不两次写入变化，就像HFS+一样，变化首先被写入日志，然后再被写入目录文件。

Juicefs

JuiceFS 是一款面向云环境设计的高性能共享文件系统，在 AGPL v3.0 开源协议下发布。提供完备的 POSIX 兼容性，可将海量低价的云存储作为本地磁盘使用，亦可同时被多台主机同时挂载读写。使用 JuiceFS 存储数据，数据本身会被持久化在对象存储（例如，Amazon S3），而数据所对应的元数据可以根据场景需要被持久化在 Redis、MySQL、SQLite 等多种数据库中。但是对于这个文件系统来说，尽管JuiceFS拥有很多的优点，而且很努力的去逼近本地存储的性能，但网络延迟，IOPS以及对象存储自身的限制，在写方面，依然和本地存储(SSD)有接近百倍的性能差距，这对于很多OLTP数据库是有比较大的影响的，但对读的优化，使得其对于OLAP等以读多的引擎则帮助巨大。详细信息见juicsfs。

数据一致性问题

什么是数据一致性问题

对于这个话题，我们根据仓库中的已有报告来看看什么是数据一致性：

什么是强一致性问题?

[TOC]

什么是强一致性问题?

目录

#需要讨论的两个方面#

对于分布式系统而言，**CAP的C是指多副本、单操作的数据一致性**。

而在数据库领域，“一致性”与事务密切相关，又进一步细化到 ACID 四个方面。**ACID里的C是指单副本、多操作的事务一致性**。

因此，当我们谈论分布式数据库的一致性时，实质上是在谈论**数据一致性和事务一致性两个方面**。（这一点，可以从 Google Spanner 对其外部一致性（External Consistency）的论述中得到佐证）

#数据一致性#

；为什么要讨论这个

包括分布式数据库在内的分布式存储系统，为了避免设备与网络的不可靠带来的影响，通常会存储多个数据副本。一份数据同时存储在多个物理副本上，自然带来了数据一致性问题：若同时存在读操作和写操作，数据一致性如何保证?

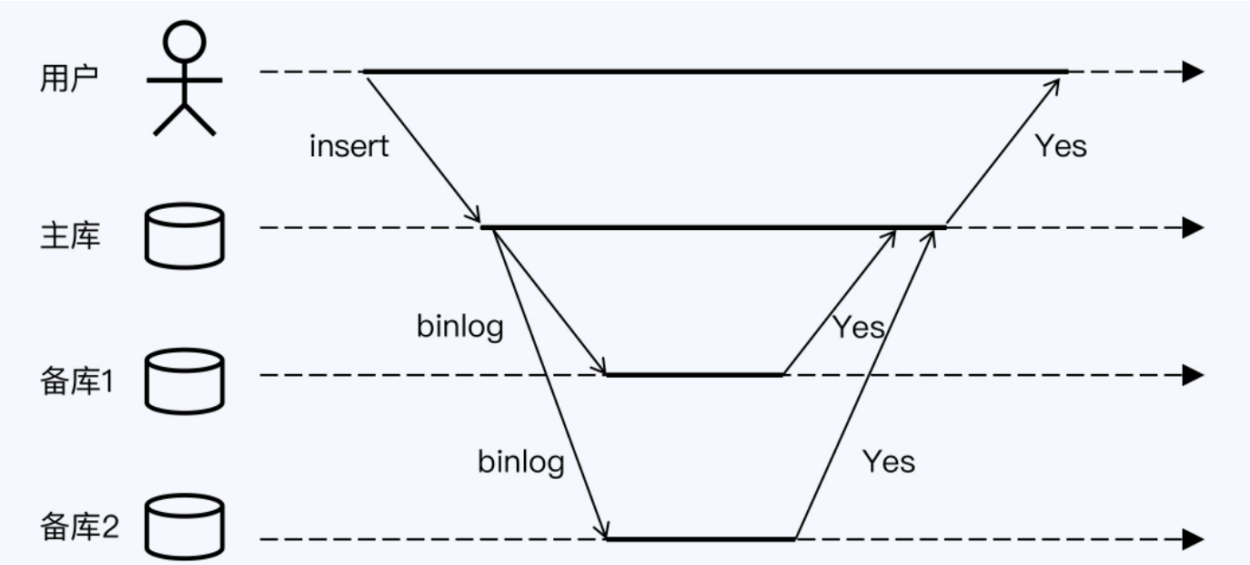
多副本数据上的一组读写策略，被称为“一致性模型”（Consistency Model）。

；分析框架

观察数据一致性的两个视角：状态和操作。（该说法来源于论文“The many faces of consistency”）

状态上：两种，操作后，所有副本一致（强一致）或不一致（弱一致）。

- 强一致如MySQL的全同步复制



  - 性能差：主库必须等到所有备库回应后才向用户反馈
  - 可用性问题：若单机可用性为P（P<1），集群串联的可用性为P的n次方，弱于单机
- 弱一致如NoSQL的最终一致性（BASE 理论中的 E：Eventually Consistency）：在主副本执行写操作并反馈成功时，不要求其他副本与主副本保持一致，但在**经过一段时间**后这些副本最终会追上主副本的进度，重新达到数据状态的一致。

操作上：如何讨论“经过一段时间”——在副本不一致的情况下，进行操作层面的封装来对外表现数据的状态。

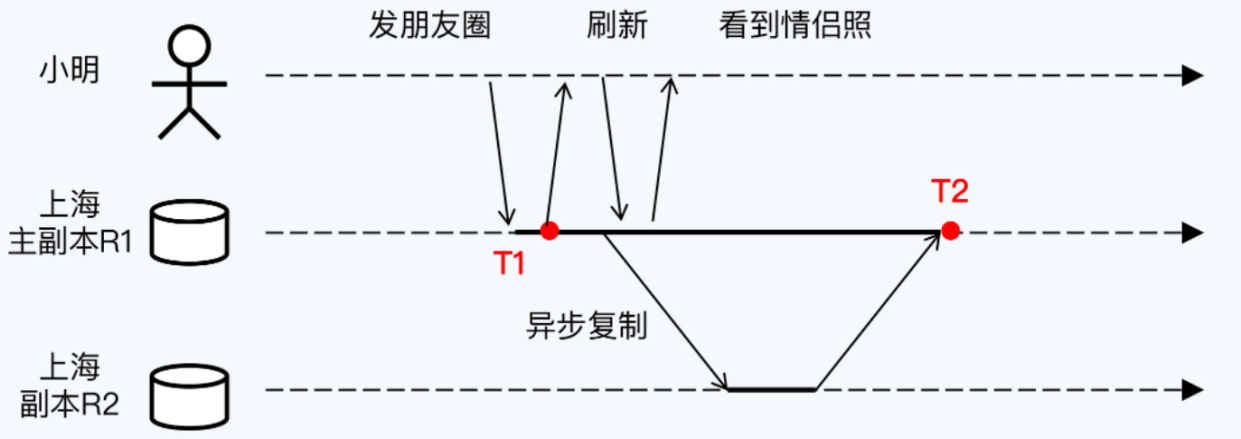
；模型分类

线性一致性 > 顺序一致性 > 因果一致性 > 最终一致性{ 写后读一致性，单调一致性，前缀一致性 .....}

&写后读一致性

自己写入成功的任何数据，下一刻一定能读取到，其内容保证与自己最后一次写入完全一致，这就是“读自己写一致性”名字的由来。当然，从旁观者角度看，可以称为“读你写一致性”（Read Your Writes Consistency），有些论文确实采用了这个名称。

1. 小明很喜欢在朋友圈分享自己的生活。这天是小明和女友小红的相识纪念日，小明特意在朋友圈分享了一张两人的情侣照。小明知道小红会很在意，特意又刷新了一下朋友圈，确认照片分享成功。

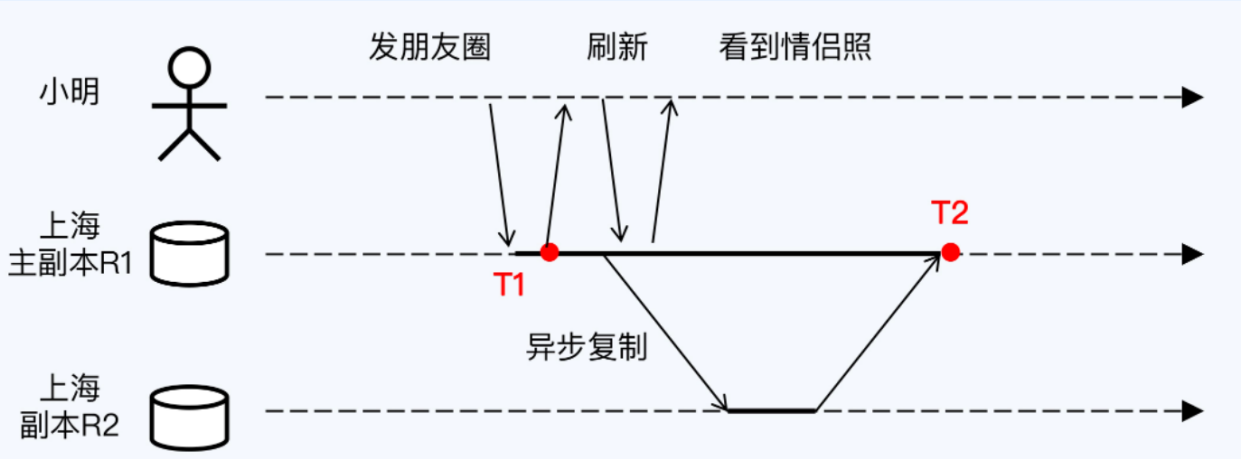


2. 在完成异步复制之前，小明只会访问主副本R1。

&单调一致性

一个用户一旦读到某个值，不会读到比这个值更旧的值。

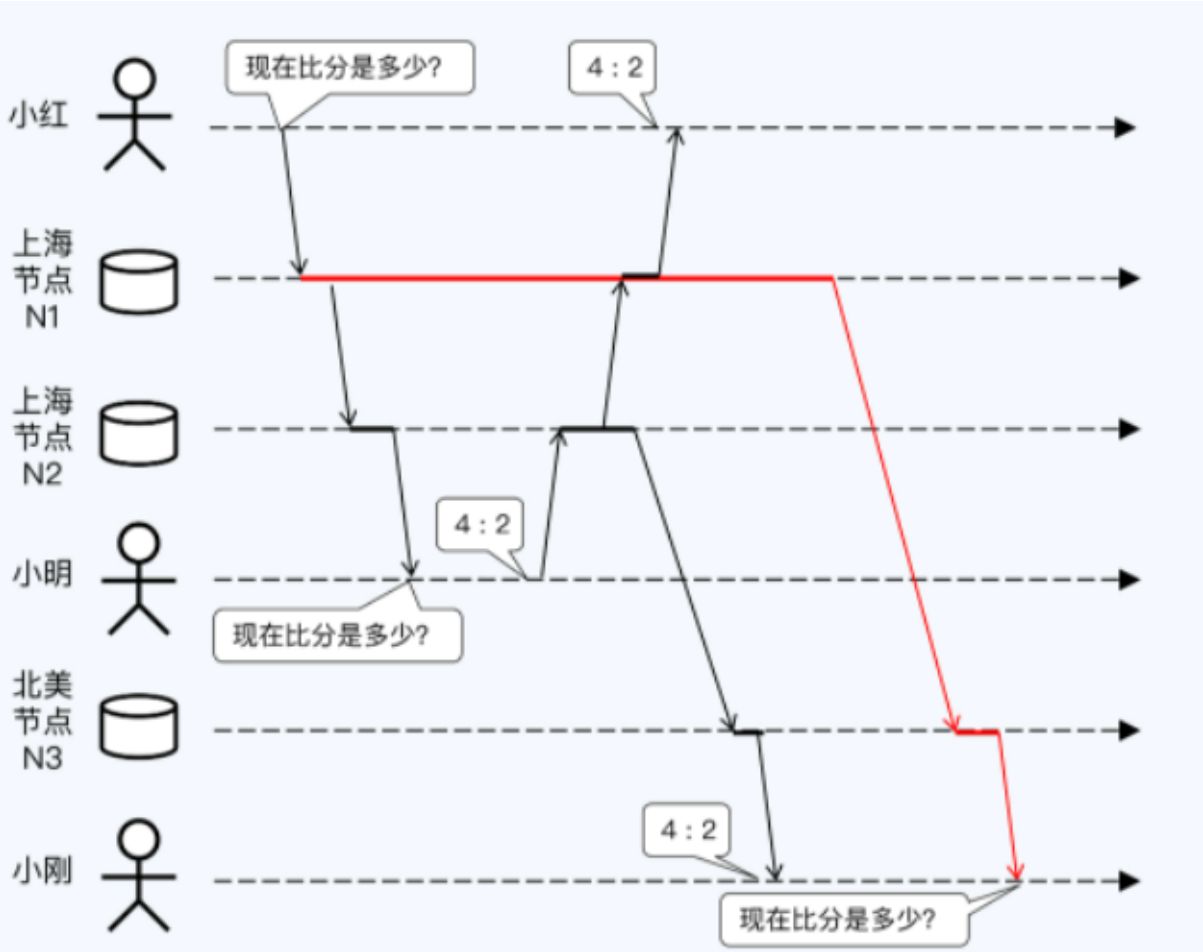
1. 此时，小红也在刷朋友圈，看到了小明刚刚分享的照片，非常开心。然后，小红收到一条信息，简单回复了一下，又回到朋友圈再次刷新，发现照片竟然不见了！小红很生气，打电话质问小明，为什么这么快就把照片删掉？小明听了一脸蒙，心想我没有删除呀。



2. 为了解决这种问题，就必须实现单调一致性。
3. 假如，变量 X 被赋值三次，依次是 10、20、30；之后读取变量 X，如果第一次读到了 20，那下一次只有读到 20 或 30 才是合理的。因为在第一次读到 20 的一刻，意味着 10 已经是过期数据，没有意义了。

&前缀一致性

1. 这天小明去看NBA总决赛，刚开球小明就拍了一张现场照片发到朋友圈，想要炫耀一下。小红也很喜欢篮球，但临时有事没有去现场，就在评论区问小明：“现在比分是多少？”小明回复：“4:2。”小明的同学，远在加拿大的小刚，却看到了一个奇怪的现象，评论区先出现了小明的回复“4:2。”，而后才刷到小红的评论“现在比分是多少？”。难道小明能够预知未来吗？



2. 显然，问题与答案之间是有因果关系的，但这种关系在复制的过程中被忽略了，于是出现了异常。保持这种因果关系的一致性，被称为前缀读或前缀一致性（Consistent Prefix）。要实现这种一致性，可以考虑在原有的评论数据上增加一种显式的因果关系，这样系统可以据此控制在其他进程的读取顺序。

&线性一致性Linearizability

但在现实中，多数场景的因果关系更加复杂，也不可能要求全部做显式声明。更可靠的方式是将自然语意的因果关系转变为**事件发生的先后顺序**。在线性一致性下，整个系统表现得好像只有一个副本，所有操作被记录在一条时间线上，并且被原子化，这样任意两个事件都可以比较先后顺序。这些事件一起构成的集合，在数学上称为具有“**全序关系**”的集合，而“全序”也称为“线性序”。但是，集群中的各个节点不能做到真正的时钟同步，这样节点有各自的时间线。那么，如何将操作记录在一条时间线上呢？这就需要一个绝对时间，也就是**全局时钟**。

1. &工业界状况：从产品层面看，主流分布式数据库大多以实现线性一致性为目标，在设计之初或演进过程中纷纷引入了全局时钟，比如 Spanner、TiDB、OceanBase、GoldenDB 和巨杉等等。工程实现上，多数产品采用单点授时（TSO），也就是从一台时间服务器获取时间，同时配有高可靠设计； 而 Spanner 以全球化部署为目标，因为 TSO 有部署范围上的限制，所以 Spanner 的实现方式是通过 GPS 和原子钟实现的全局时钟，也就是 TrueTime，它可以保证在全球范围内任意节点能同时获得的一个绝对时间，误差在 7 毫秒以内。
2. &反对者：“时间是相对的”

总结，线性一致性必须要有全局时钟，全局时钟可能来自授时服务器或者特殊物理设备（如原子钟），全局时钟的实现方式会影响到集群的部署范围。

&因果一致性Causal Consistency

不依赖绝对时间，基础是**偏序关系**——至少一个节点内部的事件是可以排序的，依靠节点的本地时钟就行了；节点间如果发生通讯，则参与通讯的两个事件也是可以排序的，接收方的事件一定晚于调用方的事件。

1. &**逻辑时钟**：基于这种偏序关系，Leslie Lamport 在论文“Time, Clocks, and the Ordering of Events in a Distributed System”中提出了逻辑时钟的概念。
2. &工业界状况：具体到分布式数据库领域，CockroachDB 和 YugabyteDB 都在设计中采用了**逻辑混合时钟**（Hybrid Logical Clocks），这个方案源自 Lamport 的逻辑时钟，也取得了不错的效果。因此，这两个产品都没有实现线性一致性，而是接近于因果一致性，其中 CockroachDB 将自己的一致性模型称为“No Stale Reads”。

总结：【线性一致性】必须要有全局时钟，全局时钟可能来自授时服务器或者特殊物理设备（如原子钟），全局时钟的实现方式会影响到集群的部署范围；【因果一致性】可以通过逻辑时钟实现，不依赖于硬件，不会限制集群的部署范围。

&顺序一致性Sequential consistency

较少在分布式数据库中使用。

#事务一致性#

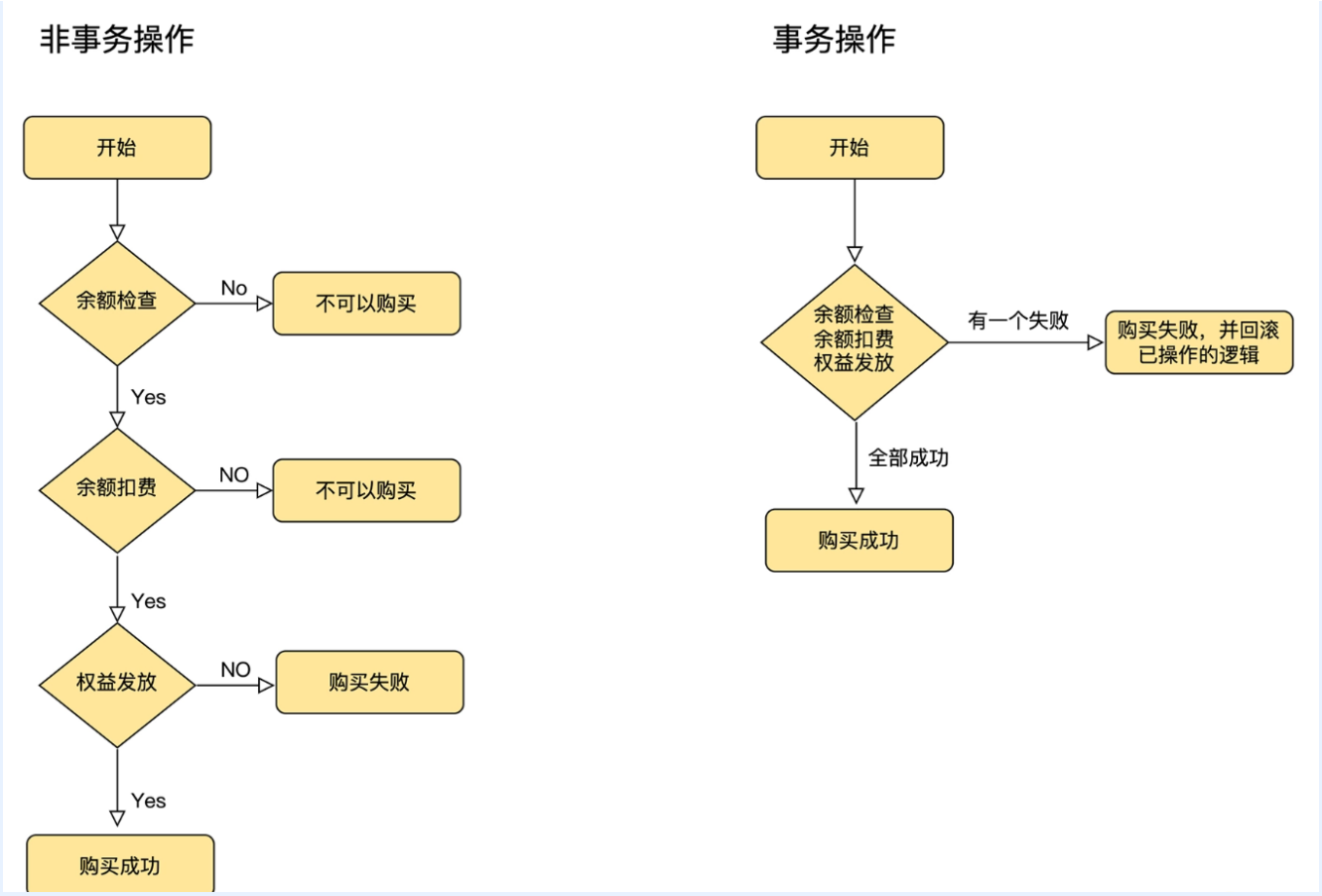
；什么是事务

在数据库中，“事务”是由多个操作构成的序列。

；为什么会有“事务”

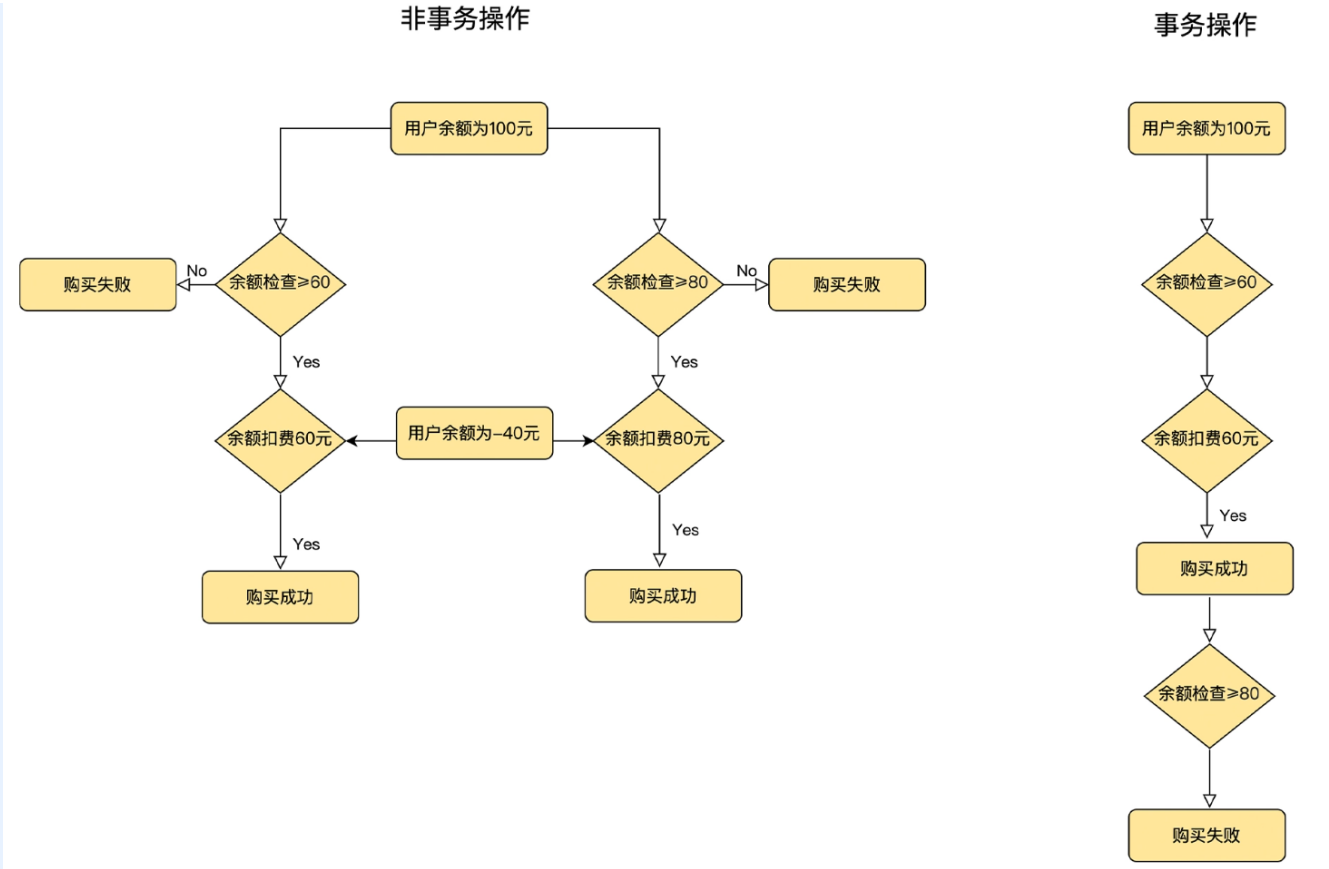
&例子一：用户购买在线付费资源->事务的原子性

在非事务操作中，如果在权益发放时资源下架或服务器崩溃，但用户已经被扣费了，必不会认同这种结果。



&例子二：用户购买操作并发执行->事务的隔离性

你看上了Steam上的两款游戏，一款60，另一款80，你只有100，但是你通过某些方式进行同时购买，如果是非事务操作：同时判断、同时扣费，这时会出现超额支出的情况。



&CPU原子操作与事务的原子性、隔离性

【原子操作】是在单核 CPU 时代定义的，由于原子操作是不可中断的，那么系统在执行原子操作的过程中，唯一的 CPU 就被占用了，这就确保了原子操作的临界区，不会出现竞争的情况。原子操作自带了线程安全的保证，即最严格的隔离级别的可串行化，所以我们在编程的时候，就不需要对原子操作加锁，来保护它的临界区了。即**原子操作同时包含了原子性和隔离性两个概念**。

而对分布式数据库，由于相对于内存来说非常慢的磁盘，而可串行化地去操作磁盘，在很多业务场景下的性能是难以接受的，于是将隔离性分离出来：隔离性为了在性能和正确性之间权衡，定义了多种隔离级别，我们可以依据自己的业务情况进行选择。

；事务的ACID特性

> **Atomicity:** Either all the changes from the transaction occur (writes, and messages sent), or none occur.
>
> > 原子性：一个事务所有的操作，要么全部执行，要么就一个都不执行，all-or-nothing。

A的实现依赖于隔离性的并发控制技术和持久性的日志技术。

> **Consistency:** The transaction preserves the integrity of stored information.
>
> > 一致性：一个事务能够正确地将数据从一个一致性的状态，变换到另一个一致性的状态。

C没有单独讨论的必要，它需要其他特性来协助完成，可以说是“事务”的整体目标。

> **Isolation:** Concurrently executing transactions see the stored information as if they were running serially (one after another).
>
> > 隔离性：多事务并行执行所得到的结果，与串行执行（一个接一个）完全相同。

> **Durability:** Once a transaction commits, the changes it made (writes and messages sent) survive any system failures.
>
> > 持久性：一旦事务提交，它对数据的改变将被永久保留，不应受到任何系统故障的影响。

D的目的就是应对系统故障，分两种情况讨论。

1. 【存储硬件无损、可恢复的故障】这种情况下，主要依托于预写日志（Write Ahead Log, WAL）保证第一时间存储数据。WAL 采用顺序写入的方式，可以保证数据库的低延时响应。WAL 是单体数据库的成熟技术，NoSQL 和分布式数据库都借鉴了过去。
2. 【存储硬件损坏、不可恢复的故障】这种情况下，需要用到日志复制技术，将本地日志及时同步到其他节点。实现方式大体有三种：
    a. 【单体数据库自带的同步或半同步的方式】，其中半同步方式具有一定的容错能力，实践中被更多采用；
    b. 【将日志存储到共享存储系统上】，后者会通过冗余存储保证日志的安全性，亚马逊的 Aurora 采用了这种方式，也被称为 Share Storage；
    c. 【基于 Paxos/Raft 的共识算法同步日志数据】，在分布式数据库中被广泛使用。无论采用哪种方式，目的都是保证在本地节点之外，至少有一份完整的日志可用于数据恢复。

; 隔离性

隔离性是事务的核心。降低隔离级别，其实就是在正确性上做妥协，将一些异常现象交给应用系统的开发人员去解决，从而获得更好的性能。所以，除“可串行化”以外的隔离级别，都有无法处理的异常现象。

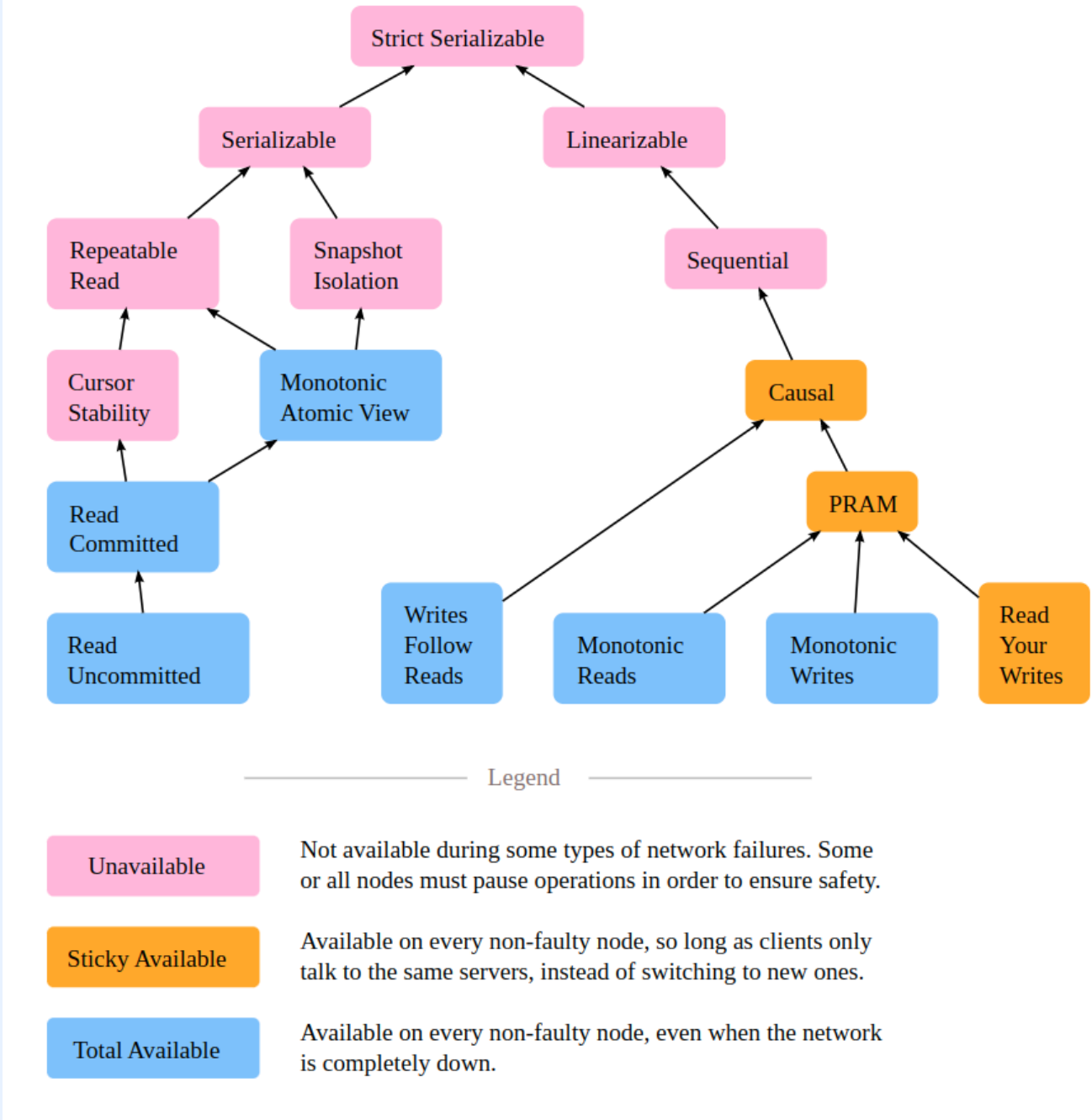
最早、最正式的对隔离级别的定义，是 ANSI SQL-92（简称 SQL-92），它定义的四种隔离级别和三种异常现象如下所示：

隔离级别	脏读 Dirty Read	不可重复读 Non-Repeatable	幻读 Phantom
未提交读 Read Uncommited	可能	可能	可能
已提交读 Read Committed	不可能	可能	可能
可重复读 Repeatable Read	不可能	不可能	可能
可串行化 Serializable	不可能	不可能	不可能

但SQL-92对异常现象的分析还是过于简单了，所以在不久之后的 1995 年，Jim Gray 等人发表了论文“A Critique of ANSI SQL Isolation Levels”（以下简称 Critique），对于事务隔离性进行了更加深入的分析。

#总结：强一致性#

在严格意义上，分布式数据库的“强一致性”意味着严格串行化（Strict Serializable），根据20年的文章，Spanner 达到了这个标准，其同时也带来了性能上的巨大开销。如果稍稍放松标准，那么“数据一致性”达到因果一致性（Causal）且“事务一致性”达到已提交读（Read Committed），即可认为是相对的“强一致性”（针对这点，不同人有不同见解，其本质原因是处理的工程问题不同）。当然，分布式数据一致性并不是越高越好，还要与可用性、性能指标结合，否则就成了形象工程。



图片原始出处是论文“Highly Available Transactions: Virtues and Limitations”，此处引用的是Jepsen 网站的简化版。

#参考#

https://time.geekbang.org/column/article/497528

https://time.geekbang.org/column/article/272999

http://kaiyuan.me/2018/04/21/consistency-concept

A Critique of ANSI SQL Isolation Levels

Jepsen : Consistency Models

Spanner: Google's Globally-Distributed Database

CRDT理论

CRDT，全称无冲突复制数据类型(Conflict-free Replicated Data Type)，在具体阐述之前，先简要介绍一下分布式系统的CAP理论。

CRDT理论主要包括以下几个方面：

强一致性(Consistency)： 即在分布式系统中的同一数据多副本情形下， 对千数据的更新操作体现出的效果与只有单份数据是一样的。

可用性(Availability)： 客户端在任何时刻对大规模数据系统的读 / 写操作都应该保证在限定延时内完成。

分区容忍性(Partition Tolerance)： 在大规模分布式数据系统中， 网络分区现象， 即分区间的机器无法进行网络通信的情况是必然会发生的， 所以系统应该能够在这种情况下仍然继续工作。

CAP 最初是由Eric Brewer于1999 年首先提出的， 他同时证明了：**对于一个大规模分布式数据系统来说， CAP 三要素不可兼得**， 同一个系统至多只能实现其中的两个， 而必须放宽第3 个要素来保证其他两个要素被满足。
关于CRDT的详细介绍见

CRDT调研报告

CRDT 调研报告

1.What is CRDT

CRDT，全称无冲突复制数据类型(Conflict-free Replicated Data Type)，在具体阐述之前，先简要介绍一下分布式系统的CAP理论。

1.1 CAP Theory

强一致性(Consistency)： 即在分布式系统中的同一数据多副本情形下， 对千数据的更新操作体现出的效果与只有单份数据是一样的。

可用性(Availability)： 客户端在任何时刻对大规模数据系统的读 / 写操作都应该保证在限定延时内完成。

分区容忍性(Partition Tolerance)： 在大规模分布式数据系统中， 网络分区现象， 即分区间的机器无法进行网络通信的情况是必然会发生的， 所以系统应该能够在这种情况下仍然继续工

作。

CAP 最初是由Eric Brewer于1999 年首先提出的， 他同时证明了：**对于一个大规模分布式数据系统来说， CAP 三要素不可兼得**， 同一个系统至多只能实现其中的两个， 而必须放宽第3个要素来保证其他两个要素被满足。

1.2 CRDT的提出

于2012年，CAP理论的创始者在*computer*杂志上刊载了一篇文章，指出了当时学界及业界对于CAP理论的误区：CAP理论并不是为了P（分区容忍性），要在A和C之间选择一个。事实上，分区很少出现，CAP在大多数时候允许完美的C和A。但当分区存在或可感知其影响的情况下，就要预备一种策略去探知分区并显式处理其影响。这样的策略应分为三个步骤：探知分区发生，进入显式的分区模式以限制某些操作，启动恢复过程以恢复数据一致性并补偿分区期间发生的错误。

CRDT应运而生，其最初提出的动机是因为最终一致性，即随时保持可用性，但是各个节点会存在不一致的时刻。论文 *Conflict-free replicated data types*提出了简单的、理论证明的方式来达到最终一致性。

CRDT 不提供「完美的一致性」，它提供了**强最终一致性(Strong Eventual Consistency)**，这代表进程A可能无法立即反映进程B上发生的状态变动，但是当A、B同步消息之后他们二者就可以恢复一致性，并且不需要解决潜在冲突（CRDT在数学上杜绝了冲突发生的可能性）。而「强最终一致性」是不与「可用性」和「分区容错性」冲突的，所以CRDT同时提供了这三者，达成了较好的CAP上的权衡。

1.3 CRDT原理

CRDT有两种类型：Op-based CRDT和State-based CRDT。下面分别介绍两种CRDT的设计思路。

1.3.1 Op-based CRDT

顾名思义，Op-based CRDT是基于用户的操作序列的。如果两个用户的操作序列完全一致，则最终文档的状态也一定是一致的。所以这种方法让各个用户保存对数据的所有操作，用户之间通过同步Operations来达到最终一致状态。但如何保证Operation的顺序是一致的呢？如果有并行的修改操作应该如何分辨先后？为了解决这种问题，Op-based CRDT要求所有可能并行的操作都是可交换的，从数据类型和Operation的层面杜绝了因操作先后顺序而导致的不一致性。

1.3.2 State-based CRDT

当可能并行的操作不满足可交换时，则可以考虑同步副本数据，同时附带额外的元信息协同副本的合并。让元信息满足条件的方式是让其更新保持单调，这个关系一般被称为偏序关系。例

如，让每个更新操作都带上当时的时间戳，在合并时对比本地副本时间戳及同步副本时间戳，取更新的结果，这样总能保证结果最新且最终一致。

1.4 A simple example

G计数器是CRDT的一个简单的例子，其中的元素满足 $a + b = b + a$ 和 $ a+(b+c) = (a+b)+c $.副本仅彼此交换更新。CRDT将通过合并更新来merge。

Time	Instance A	Instance B
t1	INCRBY key1 10	INCRBY key1 50
t2	-- Sync --	
t3	GET key1 => 60	GET key1 => 60
t4	DECRBY key1 60	INCRBY key1 60
t5	-- Sync --	
t6	GET key1 => 60	GET key1 => 60

2.Related Work

介绍基于CRDT构建的一些框架及一些相关文献，可供参考。

- Yjs （基于JavaScript）
- y-crdt （Yjs的rust实现）
- Automerge
- Delta-CRDT
- Diamond-type

项目 [https://github.com/dmonad/crdt-benchmarks] 中列出了前三者的benchmark，大致结果为Yjs性能最佳。

- [Conflict-free replicated data types](#) CRDT原论文
- [Near Real-Time Peer-to-Peer Shared Editing on Extensible Data Types](#) Yjs论文
- [A Conflict-Free Replicated JSON Datatype](#) Automerge论文
- [CRDTs The Hard Parts](#) Automerge作者的另一篇文章
- [5000x faster CRDTs: An Adventure in Optimization](#) Diamond-Type作者的文章

3.24更新 CRDT原论文摘录

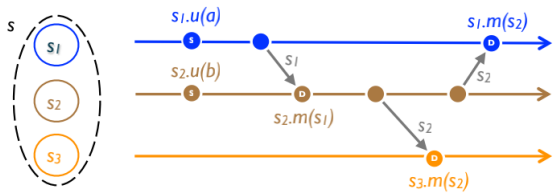


Figure 1: State-based replication

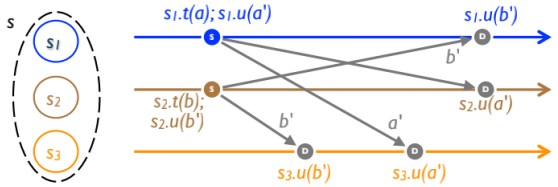


Figure 2: Operation-based replication

如图1所示，state-based replication在执行更新在执行会修改单个复制副本的状态。每个副本会定时将其本地状态发送给其他副本，后者会合并该状态。这样，每次更新直接或间接地最终都会到达每个副本。

Op-based replication种没有merge操作，并且将update操作分成了t和u两种操作。t是pre-update，u是effect-update。pre-update在产生更新的单个副本上进行，同时，在该副本（源副本）上，effect-update会在pre-update执行完之后立即执行。effect-update在所有副本（下游副本）上执行。源副本使用指定的通信协议将effect-update传递给下游副本。

在论文中，作者还提出了如何构建一个op-based Directed-Graph-CRDT。而DisGraFS正是基于图数据库做成的。或许我们可以运用图结构的CRDT去解决其强一致性问题。

payload set V, A -- *sets of pairs* { (element e, unique-tag w), ... }
  initial $\varnothing, \varnothing$ -- *V: vertices; A: arcs*
query *lookup* (vertex v) : boolean b
  let $b = (\exists w : (v, w) \in V)$
query *lookup* (arc (v', v'')) : boolean b
  let $b = (lookup(v') \wedge lookup(v'') \wedge (\exists w : ((v', v''), w) \in A)$
update *addVertex* (vertex v)
  prepare $(v) : w$
    let $w = unique()$ -- *$unique()$ returns a unique value*
  effect (v, w)
    $V := V \cup \{(v, w)\}$ -- *v + unique tag*
update *removeVertex* (vertex v)
  prepare $(v) : R$
    pre $lookup(v)$ -- *precondition*
    pre $\nexists v' : lookup((v, v'))$ -- *v is not the head of an existing arc*
    let $R = \{(v, w) | \exists w : (v, w) \in V\}$ -- *Collect all unique pairs in V containing v*
  effect (R)
    $V := V \setminus R$
update *addArc* (vertex v', vertex v'')
  prepare $(v', v'') : w$
    pre $lookup(v')$ -- *head node must exist*
    let $w = unique()$ -- *$unique()$ returns a unique value*
  effect (v', v'', w)
    $A := A \cup \{((v', v''), w)\}$ -- *(v', v'') + unique tag*
update *removeArc* (vertex v', vertex v'')
  prepare $(v', v'') : R$
    pre $lookup((v', v''))$ -- *$arc(v', v'')$ exists*
    let $R = \{((v', v''), w) | \exists w : ((v', v''), w) \in A\}$
  effect (R) -- *Collect all unique pairs in A containing arc (v', v'')*
    $A := A \setminus R$

Figure 3: Directed Graph Specification (op-based)

Paxos算法

Paxos算法是Lamport提出的一种基于消息传递的分布式一致性算法，使其获得2013年图灵奖。Paxos由Lamport于1998年在《The Part-Time Parliament》论文中首次公开，最初的描述使用希腊的一个小岛Paxos作为比喻，描述了Paxos小岛中通过决议的流程，并以此命名这个算法，但是这个描述理解起来比较有挑战性。后来在2001年，Lamport觉得同行不能理解他的幽默感，于是重新发表了朴实的算法描述版本《Paxos Made Simple》。自Paxos问世以来就持续垄断了分布式一致性算法，Paxos这个名词几乎等同于分布式一致性。Google的很多大型分布式系统都采用了Paxos算法来解决分布式一致性问题，如Chubby、Megastore以及Spanner等。开源的ZooKeeper，以及MySQL 5.7推出的用来取代传统的主从复制的MySQL Group Replication等纷纷采用Paxos算法解决分布式一致性问题。

算法具体描述参见Lamport的论文《Paxos Made Simple》。其中一个比较完善的实现是腾讯的一个工程：[phxpaxos]

PhxPaxos是腾讯公司微信后台团队自主研发的一套基于Paxos协议的多机状态拷贝类库。它以库函数的方式嵌入到开发者的代码当中， 使得一些单机状态服务可以扩展到多机器，从而获得强一

致性的多副本以及自动容灾的特性。 这个类库在微信服务里面经过一系列的工程验证，并且腾讯方面对它进行过大量的恶劣环境下的测试，使其在一致性的保证上更为健壮。关于该部分的详细介绍见仓库内报告：分布式文件系统的数据一致性。

分布式文件系统中的高性能问题

对一个分布式文件系统而言，性能是非常关键的衡量标准，有一些特性是一个分布式文件系统必须要满足的，否则就无法有竞争力。主要如下：

- 应该符合 POSIX 的文件接口标准，使该系统易于使用，同时对于用户的遗留系统也无需改造；
- 对用户透明，能够像使用本地文件系统那样直接使用；
- 持久化，保证数据不会丢失；
- 具有伸缩性，当数据压力逐渐增长时能顺利扩容；
- 具有可靠的安全机制，保证数据安全；
- 数据一致性，只要文件内容不发生变化，什么时候去读，得到的内容应该都是一样的。

除此之外，还有些特性是分布式加分项，具体如下：

- 支持的空间越大越好；
- 支持的并发访问请求越多越好；
- 性能越快越好；
- 硬件资源的利用率越高越合理越好。

使用缓存

为什么要解决缓存的问题

文件系统中可以使用缓存来实现性能的提升，对于一个由对象存储和数据库组合驱动的文件系统，缓存是本地客户端与远端服务之间高效交互的重要纽带。读写的数据可以提前或者异步载入缓存，再由客户端在后台与远端服务交互执行异步上传或预取数据。相比直接与远端服务交互，采用缓存技术可以大大降低存储操作的延时并提高数据吞吐量。

在这里Juicsfs实现了比较高效的缓存机制。详细内容见juicsfs cache_management。

数据追加型增删改

哈希存储引擎

哈希存储的基本思想是以关键字Key为自变量，通过一定的函数关系(散列函数或哈希函数），计算出对应函数值（哈希地址），以这个值作为数据元素的地址，并将数据元素存入到相应地址的存储单元中。查找时再根据要查找的关键字采用同样的函数计算出哈希地址，然后直接到相应的

存储单元中去取要找的数据元素。代表性的使用方包括Redis，Memcache，以及存储系统Bitcask等。

基于内存中的Hash,支持随机的增删改查，读写的时间复杂度O(1)。但无法支持顺序读写(指典型Hash，不包括如Redis的基于跳表的ZSet的其它功能),在不需要有序遍历时，性能最优。

基于哈希表结构的键值存储系统，仅支持追加写操作，即所有的写操作只追加而不修改老的数据，同一个时刻，只有一个活跃的新文件。

主要思想是：
1.内存中采用基于哈希表的索引结构，即hash表存放的是数据在磁盘上的位置索引，磁盘上存放的是主键和value的实际内容。

2.定期合并，定期将旧的数据或者删除操作进行合并，保留最新的数据。

3.掉电恢复，在磁盘上保留一份索引记录，在定期合并的时候产生这份索引记录，当磁盘掉电的时候直接通过这个索引记录到内存中重建即可。

存在的问题：索引的长度远小于数据的长度，这样内存存放的索引越多，磁盘存放的数据就越多。

常见的索引

首先，索引的出现就是为了提高数据查询的效率，像书的目录一样。常见的有三种：哈希表、有序数组和搜索树。

哈希表是一种以哈希函数完成键 - 值（key-value）存储数据的结构。
于是：

- 如何解决哈希冲突是一门艺术。
- 适用于等值查询，而不适用于范围查询。

有序数组，按照某一信息的有序性进行存储，范围查询的效率也很不错（比如可以使用二分法），仅看查询，它估计是最好的数据结构，但是更新数据很麻烦，所以只适用于静态存储引擎（只存储不再修改的数据）。

搜索树，典型如平衡二叉树，查询和更新的时间复杂度均为O(log(N))，效率在搜索树中是最高的，但是一般都不会使用，原因是：索引不止存在内存中，还要写到磁盘上。比如如果有100 万节点，树高20，一次查询可能需要访问 20 个数据块，而从磁盘随机读一个数据块需要 一定的寻址时间。所以一般都会使用N叉树（N取决于数据块大小），让查询过程中尽可能少地访问磁盘。典型如B+树、LSM树。

具体可参考redis中的hash，bitcask中的hash。

立项依据

如上所言，分布式文件系统在现在有着越来越多的应用范围，它的性能逐渐成为一个非常重要的指标，对于现有的分布式文件系统来说，如juicsfs等，其读写性能仍然未能达到我们所期望的高性能文件系统所应该有的标准。在这里我们决定从几个不同的方向优化打造一个高性能的文件系统——缓存服务器和地址的哈希映射等来优化2021年的OSH项目x-DisGraFS。利用它实现的一个对文件内容的打标搜索，提高原有文件系统的检索与读取性能，同时满足不同节点数据的一致性，打造一个高性能的分布式文件系统。

前瞻性/重要性分析

使用cache的必要性

用户cache

& 案例一，假设用户在使用关键词“雪”进行搜索后，界面将展示所有与“雪”有关的文件，用户打开其中的文件A，发现不是自己想用的，于是又打开文件B，也许还不是。打开文件A时，在DGFS中会进行：将消息传达服务器——JFS响应，在分布式存储集群中找到文件——请求节点得到文件，如果文件B不在用户JFS的cache里，那么打开文件B时也是同样的流程，这样的存取延迟在用户很少、文件很少时影响不大，但如果大量用户、文件接入，同时进行打开操作，不仅使服务器负载加大，更重要的是会带来大量的访存操作，况且网络读写的速度还是远慢于内存的读写，存取延迟也许让用户无法接受。

& 案例二，我们设想WowKiddy未来可以便利地参与构建用于AI模型的训练数据集，并与筛选预处理平台或训练平台进行对接。一个高精度AI模型离不开大量的优质数据集，这些数据集往往由标注结果文件和海量的图片组成。数据集都放在远端的对象存储集群中，当运行模型训练任务时就需要访问远程存储来获取数据集，带来较高的网络 I/O开销，也会造成数据集管理不便的问题。

在DGFS中，文件与文件的逻辑关系更清晰、紧密，若WowKiddy基于更强的逻辑关系布置cache，能有效提升访存命中率。这里的工作大致应该分为两个部分：

1.juicefs的默认缓存机制是顺序读取一个block作为用户的缓存，用以提高下次搜索的命中率，由于我们现在得到了文件的自然属性，那么我们可以将这个block通过某种方法替换成相同标签的block，这样对于cache的命中率会大幅度提高。

2.当同一个tag的文件内容过大时，我们不应该传入到本机的cache当中，这样会大大占用用户的网络带宽，这里我们可以将一个服务器节点抽象成一个虚拟的cache，不是作为本地的cache。比如我的存储集群在北京，而我在合肥，那么当我想去读取某个文件的时候，它可以实现将相同tag的文件发送到我的同城服务器中，这样当我再次去访问相同tag的文件的时候，就可以直接从同城服务器里拿到我想要的数据，这样的速度较于还是从北京拿数据要快得多。这样也没有占用用户本机的网络带宽，可以带给用户更好的体验效果。

另外，可以基于cache添加一个新功能——支持文件预览，比如将鼠标放在某jpg文件泡上，该jpg会自动展示给用户，比如将鼠标放在视频文件上，可以进行小窗口多倍数预览等。

索引服务器cache

&案例，在DGFS中，用户每次发送搜索请求时，索引服务器会根据关键字创建搜索语句，然后进入neo4j进行搜索，再将相关标签的所有文件通过图和列表方式返回网页端。第一，如果用户在第一次、第三次、第五次搜索的都是同一个tag，那么都进行这么一个过程，显然会造成不必要的计算开销和能源浪费；第二，每次那么如果大量用户接入，文件也很多时，如果大部分人常搜索部分tag，那么获取搜索结果信息也会有明显的延迟。
若WowKiddy增设索引服务器的cache呢?

相关工作

Juicefs

JuiceFS 是一款面向云环境设计的高性能共享文件系统，在 AGPL v3.0 开源协议下发布。提供完备的 POSIX 兼容性，可将海量低价的云存储作为本地磁盘使用，亦可同时被多台主机同时挂载读写。

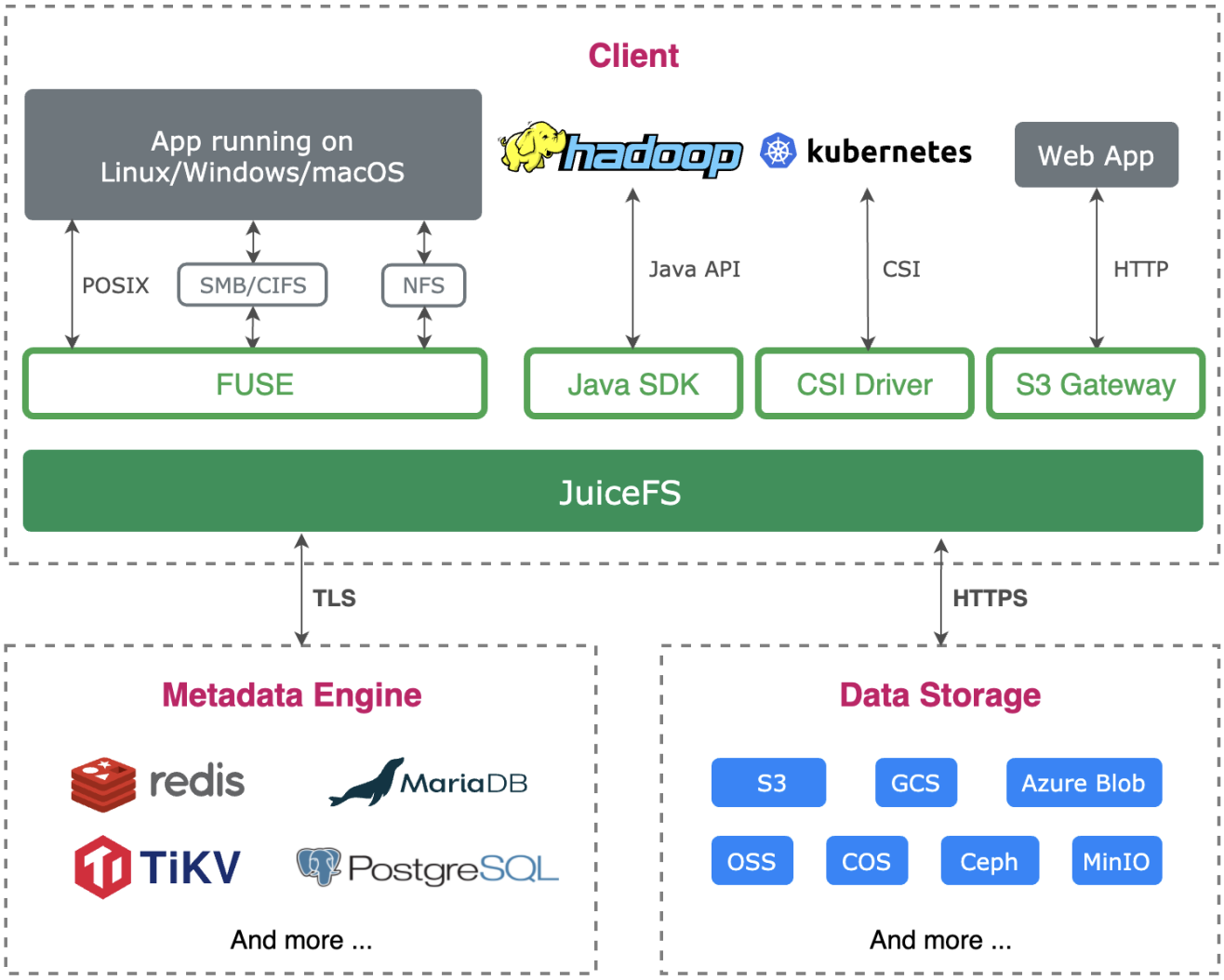
使用 JuiceFS 存储数据，数据本身会被持久化在对象存储（例如，Amazon S3），而数据所对应的元数据可以根据场景需要被持久化在 Redis、MySQL、SQLite 等多种数据库中。

JuiceFS 提供了丰富的 API，可以在不修改代码的前提下无缝对接已投入生产的大数据、机器学习、人工智能等应用平台，为其提供海量、弹性、低价的高性能存储。

核心特性

1. **POSIX 兼容**：像本地文件系统一样使用，无缝对接已有应用，无业务侵入性；
2. **HDFS 兼容**：完整兼容 HDFS API，是大数据集群实现存储计算分离架构的理想存储选择；
3. **S3 兼容**：提供 S3 Gateway 实现 S3 协议兼容的访问接口；
4. **云原生**：通过 Kubernetes CSI Driver 可以很便捷地在 Kubernetes 中使用 JuiceFS；
5. **多端共享**：同一文件系统可在上千台服务器同时挂载，高性能并发读写，共享数据；
6. **强一致性**：确认的修改会在所有挂载了同一文件系统的服务器上立即可见，保证强一致性；
7. **强悍性能**：毫秒级的延迟，近乎无限的吞吐量（取决于对象存储规模）；
8. **数据安全**：支持传输中加密（encryption in transit）以及静态加密（encryption at rest）；
9. **文件锁**：支持 BSD 锁（flock）及 POSIX 锁（fcntl）；
10. **数据压缩**：支持使用 LZ4 或 Zstandard 压缩数据，节省存储空间；

技术架构



计算和存储分离

在分布式架构发展过程中，计算和存储融合的架构缺点也在逐渐暴露：

- 机器的浪费：业务是计算先达到瓶颈的，还是存储先达到瓶颈的。这两种情况往往是不一样的，往往时间点也是不一样的。在架构里就存在一定的浪费。如果说计算不够，也是加一台机器；存储不够，还是加一台机器。所以这里就会存在很多浪费。
- 机器配比需要频繁更新：一般来说在一个公司内机器的配型比较固定比如提供好几种多少核，多少内存，多少存储空间等等。但是由于业务在不断的发展，那么我们的机器配型也需要不断的更新。
- 扩展不容易：如果我们存储不够了通常需要扩展，计算和存储耦合的模式下如果扩展就需要存在迁移大量数据。

x-DisGraFS

在分布式机群的规模上，图文件系统能够实现所管理的信息规模的扩大与各类资源的均衡分配，从而在超出人类记忆能力的信息规模上体现出图文件系统相对于传统树形结构的优越性。通过主机（Master）对从机（Slave）的存储空间以及算力的合理调度以及在主机的指导下用户与从机

之间的直接对接，x-DisGraFs的项目实现了一个高效的、用户友好的、高可扩展性的分布式图文件系统，以进一步拓展图文件系统在未来应用中的可能性。

部署问题与难点

dfs的代码中对于服务器等ip地址依赖较为严重，进行部署的过程中需要修改部分代码，对于用户的部署十分不友好（*对比Juicefs*）。同时他们的部署文档不够详细且不够用户友好，同时对于各种错误没有详细的说明与解决方案。对于一个成熟的开源项目来说，特别是文件系统这种更接近于大众的产品，一个更加友好的部署策略和一个多平台可扩展的代码实现时非常重要的。同时他们的代码存在一定的问题，他们的客户端Linux的release版本并没有完成，无法完成对应的操作，只有Windows的版本能够工作，在这里也对部署工作造成了较大的不便。于是在这里我们完成了一个较为详细的部署文档，同时萌生了自动化部署的想法。

Lua

Lua，葡萄牙语“月亮”，是一个简洁、轻量、可扩展的脚本语言。Lua有着相对简单的C API而很容易嵌入应用中。很多应用程序使用Lua作为自己的嵌入式脚本语言，以此来实现可配置性、可扩展性。

Lua是一种轻量语言，它的官方版本只包括一个精简的核心和最基本的库。这使得Lua体积小、启动速度快。它用ANSI C语言编写，并以源代码形式开放，编译后的完整参考解释器只有大约247kB，到5.4.3版本，该体积变成283kB（Linux,amd64），依然非常小巧，可以很方便的嵌入别的程序里。和许多“大而全”的语言不一样，网络通信、图形界面等都没有默认提供。但是Lua可以很容易地被扩展：由宿主语言（通常是C或C++）提供这些功能，Lua可以使用它们，就像是本来就内置的功能一样。事实上，现在已经有很多成熟的扩展模块可供选用。

Lua是一个动态类型语言，支持增量式垃圾收集策略。有内建的，与操作系统无关的协作式多线程支持。Lua原生支持的数据类型很少，只提供了数值（默认是双精度浮点数，可配置）、布尔量、字符串、表格、函数、线程以及用户自定义数据这几种。但是其处理表和字符串的效率非常之高，加上元表的支持，开发者可以高效的模拟出需要的复杂数据类型（比如集合、数组等）。

参考文献

日志文件系统wiki
redis中的hash
bitcask中的hash
juicsfs community
phxpaxos
Lua

注：不同md应用渲染效果有所不同，完整报告内容见仓库pdf。
仓库调研报告关系图谱：


README
research
分布式文件系统的数据一致性
什么是强一致性问题?
CRDT调研报告
README
README