分布式金融系统基础

任杰

2020 年 7 月 7 日

目录

序

夫以铜为镜，可以正衣冠；以史为镜，可以知兴替；以人为镜，可以明得失。

李世民
《旧唐书·魏徵传》

想写这本书已经由来已久了。金融是一个非常古老的行业，在欧美等国家已经有成百上千年的成熟发展。随着上世纪中叶计算机技术的崛起，金融行业的广度和深度都有了长足的发展。国内在上世纪末才与外界有了广泛的接触，逐步引入国外的业务和系统。然而知其然而不知其所以然，虽然软件系统可以引进，软件设计的核心原理及发展历史却无法引进。另外，金融行业的利润来自于信息不对称，计算机技术作为处理信息的最重要工具，在金融行业里是核心竞争力，不会有人会把最新的系统拱手送人。这个发展是比较合理的。因为在行业发展早期，市场会比效率更重要，系统只要能快速上线支撑业务就可以了，不需要以运维成本作为核心考核指标。由于没有历史遗留负担，业务可以快速弯道超车，但是买来的系统却不能快速迭代。这时候系统作为核心竞争力才能体现出应有的重要性。

不想写这本书也有很久时间了。虽然我在金融行业有一些浸润，毕竟时间有限，能认识的业务有限，所掌握的系统也有限。有一些内容是耳熟能详，另一些则是道听途说。如果给大家介绍的太少，行业面会偏窄，大家能学习、借鉴和运用的有限，担心会浪费大家的财力和时间。如果给大家介绍的太多，一定会有些内容深度不够，怕将大家引入歧途。既然说多说少都有错，最为稳妥的是不言而内省。

最终还是决定写一写。条条大路通罗马，金融系统理应有多种实现，偏重点不一。现在一家之言的态势愈发明显。因为担心劣币驱逐良币，特此毛遂自荐，梳理各方脉络，希望能正本清源，为正确行业的标准树立起到绵薄之力。

任杰
上海浦东
2020 年 5 月

引言

> A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable
>
> Leslie Lamport
> Email message sent to a DEC SRC bulletin board at 12:23:29 PDT on 28 May 87.

不同的软件有不同的复杂度。当人们提到一个软件要求非常高的时候，会说它会被用在金融系统中。比如现在做数据分析常用的 numpy，它在介绍自己数值计算部分的时候很自豪的提到自己已经在金融系统中有所使用。那么金融系统究竟和一般的软件系统的区别在哪里呢?

金融系统的特殊之处在于它处理钱。在日常生活中一块钱和一亿块钱有很大的区别。一块钱可以随随便便拿走，一亿块钱可能一般的家用型小汽车还装不下，处理起来比较麻烦。所以在现实世界里不同额度的钱处理的方式会不一样。但是在计算机世界里，不管是一块钱还是一亿块钱都只是一个数字而已。因为都只是一个数字，计算机在处理钱的时候要分外小心，如果出了点问题，不管金额大小都会被影响。

所以当我们提到金融系统要求非常高的时候，并不是对正常的情况有特别高的要求，比如说常见的系统延时和吞吐量等等。而是对不正常的情况有非常高的要求。系统尽量不要出任何问题。如果万一出问题了，需要有办法能清楚的知道究竟是在什么时候出了什么问题。出了问题并不可怕，怕的是不知道出了什么问题。

具体细分来说金融系统有如下几点要求：todo

1. 正确性。这是从系统实现人员角度来看的正确性。系统需要有合理的架构设计，有正确的实现，有相应的测试来保证代码正确率。
2. 可解释。这是从第三方审计的角度来看的正确性。产品和业务人员需要有能力独立验证关键业务逻辑的正确性。
3. 可靠性。系统在各种不同异常情况下对正确性保证的量化衡量。
4. 可维护。完整的软件系统并不是一蹴而就，而是逐步迭代和升级的。金融系统的设计需要让系统容易升级。
5. 性能。虽然一般来说性能不是金融系统追求的目标，但是需要保证业务所需要的合理的性能要求。

总之，金融系统要让人用的开心，用的放心。这更多的是对软件质量的要求。接下来我们从设计一个金融系统中最核心的支付系统入手，分以下章节逐步介绍金融系统的设计思路:

1. 常见的方案是怎么做的。这是一个普遍采用的方案，可以快速的解决很多行业的问题，不仅仅是金融行业。
2. 如何对金融行业做正确的建模。建模的过程是定义合理数据结构的过程。
3. 如何对金融业务的模型做合理的操作。
4. 如何消除数据单点，使其拥有可选的容灾性。

5. 如何消除服务单点。
6. 如何在消除系统单点的情况下保证数据的正确性。
7. 单点持久化存储不够的情况下的横向扩展。
8. 单点内存不够的情况下的业务处理。
9. 深入理解数据持久化。各种不同的方案对正确性都有哪些不同的保证。
10. 严格执行一次。
11. 数据处理。
12. 架构反思。

需要提醒大家的是金融业务非常复杂，对应的软件系统也各有特点，不能一概而论，需要具体问题具体分析。在这里只是给大家介绍最基础的一些内容和思路，希望能起到一点抛砖引玉的作用。

第一章　架构设计小论

1.1　架构师的定义

架构师没有一个公认的清晰准确的定义。架构师会做一些开发人员的工作。一般来说高级或者资深程序员的下一步发展之路是成为一位架构师，所以偶尔重操旧业写写代码是常见的事情。架构师也可能会做一些产品经理的工作，比如在设计系统架构的时候考虑对用户交互体验的影响。架构师也有可能做一些项目管理工作，追踪项目进度，收集和分配项目需要的人力和物力资源。架构师也有可能做一些销售工作，和项目拓展人员一起列席项目推介会，从系统的角度向潜在客户营销产品优势。架构师日常工作之一是画各种架构图，需要熟练使用各种绘图工具和非线性编辑工具，而这些一般都是设计师的专长。不同的公司可能都有架构师这个岗位，但是对这个岗位的定位很有可能各不相同。

但是既然都叫架构师，那么一定是有一些共性的。架构师是什么虽然不太容易定义，但是我们可以从架构师解决的问题出发尝试寻找一些共性。

首先架构师解决的是偏抽象的问题。具体的问题，比如说程序代码每一行应该怎么写，代码测试应该怎么寻找边界条件，代码的发布和上线应该遵循什么样的流程，这些都是程序员在早期需要熟练掌握的基本技能，是成为架构师的基础。只有在这些具体软件问题都能游刃有余的解决的情况下，一个程序员才能思考更深的软件开发问题，向架构师方向发展。这也是常提的道与术的区别。

其次架构师解决的问题时间跨度长。如果一个需求不需要存在很久，就不需要考虑合理的架构，也就不需要架构师。比如现在互联网创业者都在积极的寻找下一个风口，对于每一个可能的增长点都小规模的试一试，希望通过撞概率的方式来选择正确的赛道。既然是小规模的试一试，就知道项目可能不能存活很久，不需要有长远的架构设计。如果万幸之中撞到了下一个风口，成为了独角兽企业，这时候企业就需要有长期发展的计划。成熟的企业创始人会在引入投资之后马上招聘架构师进行架构升级，为接下来的用户快速增长提前打好基础。

再者架构师的解决方案一旦实施便难以更改。复杂的软件是一个围绕核心组件的生态系统。核心组件用软件的形式解决核心业务需求，周边还需要有各种辅助系统来保证业务的完整度。核心组件由于牵涉过多，其升级会牵一发而动全身。因此为了保证复杂系统的演进，架构师需要在早期全盘思考未来的各种可能发展，从中选择一条可行的道路。

最后架构师需要将系统设计的容易改变。乍一看这是一个悖论，如果架构师的长期架构设计一旦实施就难以改变，那怎么又能容易改变了呢？这是一个哲学问题，需要从哲学的角度来看。这和唯一不变的是变化是同一个道理，也是区分优秀架构师和平庸架构师的标志。系统的主体架构奠定了系统的基础，和房子的承重墙一样，起到了中流砥柱[1]的作用。承重墙不能敲，系统的主体架构也不能轻易改变。但是在主体架构之上可以有很多衍生的系统功能，就和房子可以有不同装修一样，对不同的需求可以有不同的实现[2]。

所以总结一下。架构师解决的问题是长期抽象的问题，需要能将难以改变的系统变得容易改变。

[1] 《晏子春秋·内篇谏下》:“吾尝从君济于河，黿衔左骖，以入砥柱之中流。”

[2] 忒修斯之船（The Ship of Theseus）悖论：如果忒修斯的船上的木头被逐渐替换，直到所有的木头都不是原来的木头，那这艘船还是原来的那艘船吗？

1.2 架构设计六小原则

计算机软件是一个复杂的系统。复杂的系统总是似而不同，在细节的地方各不一样。因为细节决定成败，复杂的系统很难有屡试不爽[3]的解法。尽管如此，解决的多了也会遗留一些经验。这些经验再加以重复验证就形成了原则。这些原则有其特殊的历史背景，不一定放之四海而皆准。但是不听老人言，吃亏在眼前，在能自行顿悟之前建议能熟诵于胸，以防不测。

jià gòu liù zé
架构六则
zhuàng tài duō jù xiàng dú xiě běn bù tóng
状态多具象，读写本不同。
zhuàng tài chángqiān yí yī zhì yǒu chéng běn
状态常迁移，一致有成本。
yōu huà shǒu běn zhì biàn huà xiāng duì lùn
优化守本质，变化相对论。
shú sòng liù xiǎo jì jìn náng suí shēnxíng
熟诵六小计，锦囊随身行。

1.2.1 状态多具象

计算机数据分为正源和衍生两种。正源数据[4]指的是官方钦定的最原始的数据，所以有时候也叫原始数据。衍生数据是基于正源数据派生而来，对正源数据进行二次加工，是对正源数据的一个特殊的视角[5]。常见的衍生数据生成方式有筛选、聚合、统计等等。有一些数据属于衍生数据，但是并不那么直观。一个例子是数据库索引。数据库索引是为了加速数据查询而生成的特殊数据结构，是由数据库表衍生出来的一份数据。另一个例子是网页前端界面显示。一般的网页前端界面显示的流程是这样的：

1. 后端服务器生成业务数据。
2. 业务数据变成与前端通讯的格式，比如 json、xml 等，通过网络协议发送至网页浏览器。
3. 网页代码将数据变成特定的浏览器 Javascript 对象。
4. 浏览器将 Javascript 对象变为计算机显卡能识别的数据。
5. 显卡在显示器上展示数据。

以上每一步流程都是数据的处理和加工过程。从每个加工组件的角度来看，输入是正源数据，输出是衍生数据。从上到下完整的来看，后端服务器上的是正源数据，其余数据都是衍生而来。

判断数据是正源还是衍生的办法很简单：如果数据不见了，是否能够基于其它数据重新生成出来。比如前面提到的数据库索引数据。数据库有索引的删除和表的索引重建操作，表删除了就没有了，所以索引是衍生数据，数据库表是正源数据。对于网页显示的例子，浏览器刷新之后，网页显示会先消失然后重新展示出来，但是如果后端服务器宕机就没办法再展示网页了。所以前端显示是衍生数据，后端数据是正源数据。

在架构设计中，因为衍生数据可以基于正源数据重复产生，衍生数据的容灾度较高，恢复的方法较多。相对而言正源数据只有一份，损坏之后无法轻易恢复，因此需要针对业务对容灾的需求进行精心的设计。

1.2.2 读写本不同

对于每个人来说，读是一个内容输入的过程，写是一个内容输出的过程。一般会认为这是两种不同的个人能力，不一定同时都很优秀或者都不优秀，比如有人可能阅读能力强但是写作能力弱，也有可能写作能力强但是阅读能力弱。同理，计算机的读写也是两个不同的行为。计算机的读和写的对象都是数据，但是读操作是数据的输入过程，写操作是数据的输出过程。这里提到的读写本不同的第一个不同点在于操作的不同。数据的输入和输出可能会面临不同的吞吐量、延时、容灾等需求，需要分别对待。

[3] 清·蒲松龄《聊斋志异·冷生》：“言未已，驴已蹶然伏道上，屡试不爽”。爽，差错、失败的意思。
[4] 正源数据即 data from golden source。
[5] 视角即 view。

读写本不同的第二个不同点非常难以发觉，那就是数据的不同。在1.2.1节状态多具象里我们介绍了数据分为正源数据和衍生数据两大类。之所以会有衍生数据的一个原因是数据的使用，即数据的读取，可能需要有特殊的优化方式。为了读性能的原因需要对正源数据做一些预处理，比如建索引。所以对于数据库的例子来说写的数据是正源数据，读的数据是衍生数据，所以读写的操作是基于两份不同的数据进行的。

总结一下。读写的不同在于读写的数据不同以及数据的流向不同。在架构设计时需要将读写分开考虑，分开优化。

1.2.3 状态常迁移

1.2.4 一致有成本

1.2.5 优化守本质

1.2.6 变化相对论

第二章　抛砖引玉-直观方案

2.1　多服务 + 单 DB

核心思想：微服务。SOA。无状态服务，数据库维护状态。数据库有两张表：余额表和流水表。同时有主备两个数据库。服务器做 CRUD。通过事务访问数据库。

问题：

1. 数据库宕机了怎么办。如果是异步备份，主数据库 down 了，切换到备数据库会掉数据。如果是同步备份，备份数据库宕机了也会影响主服务器的服务。
2. 事务究竟能保证什么。有一些问题加锁也不一定能解决。比如要求用户多个账户合起来金额不能低于 100 块。两个程序分别查了之后发现余额是够的，分别扣不同账号的款之后发现总额不够。如果都加锁会很慢。

2.2　多服务 + 多 DB

业务继续增大的情况下，单个数据库不能支持数据量。这时候需要把余额数据和流水数据做近一步切分。但是分库策略不一样。流水是双边账，会影响两个用户，和借贷方任何一个放在一起都不合适。

这时候需要分布式事务。但是分布式事务也有自身的细节。比如有个节点 gc，返回延时了怎么办。

这些问题都是有解决方案的。但是大的核心思想不变，在此基础上小修小补。有没有更直击本质的解决方案。

第三章　业务建模-领域模型

money，currency，asset，cashflow，etc entity，object，repository，etc 定义合理的数据结构，解决 CRUD 不太合适的复杂业务逻辑。

第四章　业务实现-事件溯源设计模式

4.1　核心模型

event，command，state。command 到 event 可以有随机性。NOP event

apply state：无 side effect，immutable。举例：发现 CPU core 寄存器出问题。

举支付的例子。event 是 +100 元，不是 100 元。

4.2　数据持久化

event 记录到 log。state 不需要记。command 也不需要记。可以选择记录 command 和 state 前后变化。举支付例子。

log 实现细节：

1. log 写了一半怎么办。需要加验证。
2. 写 append only log，记录所有状态变化的原因。user intension。又叫 WAL
3. log 是数据结构，可以写文件，或者写数据库。重点是一个只增的线性列表。
4. 单线程执行。原因：内存够大，oltp 访问很少的数据点。
5. 不可篡改。行签名，再加密。跨行 id，跨用户 id。和区块链类似的地方和不同的地方
6. log 需要有 retention。金融一般有长达 10 年的保存要求，因此是转移而不是删除。可以定时，定长。
7. 需要能定位到 log 的任何位置。因此需要两个文件，一个 index，一个 payload。其实需要 3 个。还有一个是 index 的 index，记录 log retention 问题。

4.3　CQRS

原则：一个服务只做一类事情。要么改变状态，要么查询状态。

举例：提供用户所有账号的余额查询。查不到怎么办？再刷新一下就好了。

读写分离。reader model。event sourcing：写。CQRS：读。读写有延时。需要最终一致性。

针对不同查询可以有不同的优化和物理存储形式。不一定需要是 RDBMS，但是一般这是首选。这时候数据库是一个 view，不是元数据。元数据是 log。

查询有可选的延时。不想延时的话需要 event 和 query 数据放在同一个事务中。

会有一定的延时。因此需要做业务补偿。举例：查询用户余额后再扣钱，可能这时候没钱了。没事刷新一下就好。频率低，影响小。

读的时候如何让用户知道是数据没到还是没有。如果是看服务器时间的话，读的服务器时间和写的服务器时间不一致了怎么办。

4.4 同步调用

之前的都是用消息的方式异步调用，调用者发了 command 之后不知道处理情况，只能通过 CQRS 查询处理情况。

加回调组件。可以回调给同一个人，或者制定的其他人。

4.5 满足的功能

1. 可回滚，reprocess，time machine
2. 可解释
3. future compatibility：reader model。消费 event，产生新的 view。

4.6 优化

1. 顺序写很快
2. 加速恢复靠 snapshot。snapshot 里记了最后一个 offset。可以有 delta snapshot，记两个 offset。一段 log 对应一个 delta offset。和 memtable 结构类似。
3. 避免多次写硬盘。log 一次，b+tree 两次
4. exact once：看最后一章。
5. 单线程执行。不要在核心线程上放太多内容。IO 并发，加解密并发。

4.7 数学视角

$$
\begin{aligned}
S_n &= f(S_{n-1}, e_n) \\
S_n &= S_{n-1} + f(e_n) \\
S_n &= \sum_{I=1}^{n} f(e_i)
\end{aligned}
\tag{4.1}
$$

反向演进：$S_{n-1} = S_n - f(e_n)$

假设：存在-，是 + 的反函数。如果 + 是加钱，− 是减钱。举个实际例子。

4.8 维护

1. 系统升级：event version，code version
2. 回归测试

4.9 其他案例

1. 迷宫
2. 区块链

第五章　消除数据单点

5.1　为什么要做数据复制

备份和容灾

5.2　常见考虑点

系统容灾有什么要求。需要多少 reliability？不一定高。
传统是用硬件来做，比如 raid 和 NAS。现在是用软件来提高系统容错率。
多少种写的方式。单 leader，多 leader，无 leader。
分布式读不一致都有哪些分类。
数据问题：有冲突，有丢失，有不一致
怎么解决冲突
怎么解决丢失
怎么解决不一致
摊销冲突解决的成本：读的人解决冲突还是服务器自己定时解决。
不同 copy 可以有不同的索引。

5.3　单 leader 解法

复制方式：async，sync。多个备份机一起提供服务。
从主读和从副节点读的问题。读是从 read model 读，和主业务系统的读不一样。
CAP 定义。为什么 CAP 理论不太对。假设 P，但是不一定有 P，比如 google。
async 复制的问题：

1. old leader rejoin
2. new leader discard uncommitted result
3. two leaders
4. how to detect leader is dead

eventual consistency 问题
机器挂了怎么办
如果想 CAP 都满足，看 consensus 章。

5.4　不合适的其它选择

5.4.1　多 leader 解法

多个 leader 都可以写同一份数据。

怎么发现冲突。

怎么解决冲突：

1. version vector，dotted version vector。举个购物车例子。金融行业不建议使用。
2. last write win LWW
3. 谁来 resolve：server，client，read，write

CRDT：转账是一类这种数据。

5.4.2 无 leader 解法

quorum：多写，多读

sync write 是一个特例。

填丢失数据：client，server 都可以

不能回滚。因此不知道哪些数据是脏数据。金融行业不建议使用

5.5 Kafka 举例

5.5.1 Kafka 实现原理

5.5.2 用 kafka 实现

第六章　消除服务单点

6.1　解决问题

多个服务同时存在的情况下如何保证只有一个提供服务，其它的都是备份。涉及到的是分布式系统中如何达到共识。

6.2　问题复杂度

简单的实现都有哪些，都在什么情况下会出现问题。

6.3　假设有正确的服务发现

通过服务发现选 leader，拿到 lease。比如 zookeeper 或者 etcd。新老两个服务有不同的 lease，都可以往下游写，下游怎么判断选择哪一个?

如果下游是个数据系统，数据系统无法添加相应逻辑，需要所有数据使用方都知道怎么选择正确的数据源。

如果需要通过写服务来写数据系统，服务可以通过维护当前最新的 lease 数字来判断。但是这个服务如果在收到老请求后 gc 了一分钟，起来后继续处理怎么办? 这时候不会去拿最新的 lease。

6.4　总结

下游可以发现上游出现了脑裂。但是下游无法保证自己没有脑裂，是一个循环依赖的问题。想要解决的话需要有一个不会脑裂的服务集群。

第七章　不再多龙治水-在分布式系统中达成共识

7.1　共识的定义

3 个点。

7.2　线性一致性的定义

一堆 paper。

7.3　复制状态机

复制状态机和公式算法的等价性

7.4　consensus 历史

3 个算法

7.5　raft 协议介绍

1. last index，commit index，apply index
2. 用户交互协议
3. 选主。fsync。kafka 没有做 fsync
4. 复制

7.6　和事件溯源结合

单独拿出来做一个 log 复制系统有什么问题
结合在一起有什么要注意的地方。换主了怎么办。怎么分层和优化。
业务主和同步主不在一起有什么问题。在一起有什么难点。

7.7　deploy 方式

1. 2-3
2. 3-5
3. 3-9

复制效率分析：正常情况无损失 google 的需要 2 个 rpc。

7.8 研发成本

特别重要的系统需要这么复杂的设计。

第八章　横向扩展

需要分库。分库和复制是在一起的，要先解决复制的问题。

8.1　分库的策略

分的原则：mece：mutually exclusive collectively exhaustive 分的方式：by region，by hash。举例。by hash 的问题在于不可以做 range 查询。

8.2　基于消息的事务补偿

不用事务的解决方案。需要和业务一起优化。

8.3　分布式事务和 tcc

TCC 具体实现细节。双插解决了什么问题。TX manager 可以是单独模块，或者是第一个节点。

8.4　层级嵌套的分布式事务

8.5　分布式查询需要次级索引

举例：查 USD 相关的交易
局部和全局索引
更新方式和索引

8.6　动态分库与合并

需要有一个配置中心保存当前分库情况。可以有延时。动态分库的步骤动态合并的步骤问题：分割会让已经很忙的节点更忙。分割中的服务发现：谁发现，怎么发现

8.7　分布式快照

8.8　逻辑时间

8.8.1　逻辑时间的作用

物理时间有非常良好的性质，但是现实世界有种种因素导致我们无法完全利用这些性质。那么让我们看看究竟哪些性质是重要的、必不可缺的，哪些性质是可以放弃的。

时间的一个性质是两个时间点之间可以比大小。这是一个非常有用的信息。一旦我们可以比较两个时间的大小，我们就可以判断其发生的顺序。比如在处理数据库的读写冲突时，我们就可以知道是读操作先发生，还是写操作先发生。这个先后顺序能决定数据库应该用哪种方式来解决冲突。所以时间可以比大小这个性质需要保留。

时间的另一个性质是时间间隔有比例关系。这个貌似是不太常见的性质。比如我们知道一周的时间间隔是一天的 7 倍，而且无论 0 点如何选取，这个比例关系永远是 7 倍。可是这个倍数有什么用呢？所以这个比例关系是一种屠龙之技[1]，比较鸡肋[2]，可以放弃。

在放弃了时间段的比例关系后，我们需要进一步思考还有哪些性质是可以放弃的。在第8.9.4节我们提到过物理时间是无限精度而计算机时间是有限精度。一旦时间的衡量精度下降，如果两个事件发生的足够近，那么我们将无法判断这两个事件的发生先后顺序。既然我们做不到无限精度，有限精度又存在各种问题，还不如寻求另一种时间的表示方式来满足两个时间之间可以比大小这个功能。这便是逻辑时间。

8.8.2 逻辑时间的定义

逻辑时间是 Lamport 在 1978 年第一次完整提出的 [Lam78]。在正式定义逻辑时间前我们需要对系统做一些简单的假设。

我们假设在分布式系统中有多个节点。这些节点之间通过点对点发消息的方式来互相沟通。消息的传播需要一定的时间，而且无法预期这个时间需要多久。另外如果一个节点发了多条消息给另一个节点，这些消息可能会乱序到达，而不是保证先发先到。

与消息相关的另一个定义是事件。事件是导致节点状态变化的因素。节点有 3 种不同事件：

1. 发送消息的事件。
2. 接受消息的事件。
3. 内部事件。

对于某个消息来说，其发送和接受都会影响节点的状态。同时节点内部也会自我发生一些状态变化，导致这些内部状态变化的是内部事件。

基本概念到这里已经定义的差不多了，接下来看看事件的先后关系定义：

Definition 1. 对于两个事件 e_i 和 e_j，它们之间的先后关系 $\rightarrow$ 定义为：

1. 如果 e_i 和 e_j 发生在同一个节点，且 $i < j$，那么 $e_i \rightarrow e_j$。
2. 如果 e_i 是一个消息的发送事件，e_j 是同一个消息的接受事件，那么 $e_i \rightarrow e_j$。
3. 对于第三个事件 e_k，如果 $e_i \rightarrow e_j$，而且 $e_j \rightarrow e_k$，那么 $e_i \rightarrow e_k$。

如果 e_i 和 e_j 之间既不满足 $e_i \rightarrow e_j$，也不满足 $e_j \rightarrow e_i$，那么 e_i 和 e_j 之间是并发关系，记为 $e_i \parallel e_j$。

$\rightarrow$ 和 $\parallel$ 的定义需要一些解释。在分布式系统中的各个节点只了解本地信息。所以如果单个节点内发生了两个事件 e_i 和 e_j，它是能判断这两个事件发生时打的标记 i 和 j 来判断事件的先后顺序。

但是这个节点对其它节点发生了什么变化并不能准确的了解。唯一了解途径是通过获取其它节点发送的消息。一旦一个节点收到了另一个节点的消息，这个节点可以一定程度猜测另一个节点发生了什么，从而能确定某些事件的先后顺序。最简单的先后顺序判定是收到消息的时间一定晚于发送消息事件，所以如果 e_i 是一个消息的发送事件，e_j 是同一个消息的接受事件，那么 e_i 一定早于 e_j 发生，因此 $e_i \rightarrow e_j$。更抽象一点来讲，每个节点像盲人摸象[3]一样在本地维护了对整个分布式系统的片面认

[1] 《庄子·列御寇》:“朱泙漫学屠龙于支离益，殚千金之家，三年技成，而无所用其巧。”.

[2] 《后汉书·杨修传》:“夫鸡肋，食之则无所得，弃之则如可惜。”。

[3] 《大般涅槃经》三二:“其触牙者即言象形如芦菔根，其触耳者言象如箕，其触头者言象如石，其触鼻者言象如杵，其触脚者言象如木臼，其触脊者言象如床，其触腹者言象如甕，其触尾者言象如绳。”。

识。一旦节点收到了其它节点的消息，这个节点便能更新它对整个分布式系统的认知，从而更近一步逼近最新的全局状态。虽然更新之后的认知依然是片面的，但比收到消息之前准确了一些。

先后顺序有一个良好的数学性质是它具有可传递性。所以一旦我们发现 e_i 在 e_j 之前发生，而且 e_j 在 e_k 之前发生，根据传递性规则，我们可以肯定 e_i 在 e_k 之前发生。因此如果 $e_i \to e_j$，而且 $e_j \to e_k$，那么 $e_i \to e_k$。

最后剩下一种情况是 e_i 和 e_j 之间没有任何可以推导出来的先后关系。这时候我们"定义"这两个事件是同时发生的。这里的同时指的是在逻辑时间的范畴内，而不是指真实的物理世界内同时发生。

说到这里我们终于可以看一下逻辑时间的定义：

Definition 2. 假设 $(S, <)$ 是一个偏序集合，$E : \{e_i\}$ 是所有事件的集合。逻辑时间$C : E \to S$ 是一个函数，满足以下性质：

$$e_i \to e_j \Rightarrow C(e_i) < C(e_j) \tag{8.1}$$

强一致性逻辑时间$C : E \to S$ 需要满足以下性质：

$$e_i \to e_j \Leftrightarrow C(e_i) < C(e_j) \tag{8.2}$$

我们简单解释一下这些深奥的定义。逻辑时间给每个事件赋予了一个数。这些数有一个特殊性质，那就是数之间是可以比大小的。这些数构成了一个集合叫偏序集合。偏序集合的定义是集合成员之间存在大小关系，而且这些大小关系是可以传递的，但是不保证任意两个成员之间都存在大小关系。比如 $\{1, 2, \ldots, 1000, \ldots\}$ 是一个偏序集合，这个偏序集合里任意两个数都可以比大小。所有虚数也是一个偏序集合，只不过只有实数之间可以比大小[4]，虚数之间以及虚数和实数之间不能比大小。

逻辑时间不是给每个事件随意的赋予一个数。这个赋予的过程需要保证大小顺序。如果两个事件之间存在先后顺序，那么它们被赋予的数之间也需要存在等价的大小顺序。强一致性逻辑时间是对这个赋予过程提的更高的要求。逻辑时间只是要求如果事件有先后，那么数值有对应的大小关系。强一致性逻辑时间要求反过来也成立，即如果两个事件被赋予的数值之间有大小关系，那么这两个事件之间存在对应的先后关系。第8.8.3会介绍逻辑时间把事件映射到标量时间，第8.8.4节会介绍强一致性逻辑时间把事件映射到矢量时间。

8.8.3 标量时间

问题：相同时间值不表示同时发生时间大不表示发生在后面

8.8.4 矢量时间

矢量时间是强一致的，如果不具有可比性表示是同时发生

8.9 物理时间

8.9.1 时间的定义

时间的定义经过了漫长而且曲折的过程。最开始的时间是按照地球的自转速度来定义的。时间的单位是秒，一秒钟定义为 $\frac{1}{86400}$ 天。后来科学家觉得地球轨道是个椭圆，可能每天的长度不太一致，因此把一秒钟定义为地球按照太阳公转的 $\frac{1}{31556925.9747}$。这两种定义的都是天文秒。随着原子物理的发展，科学家们找到了更为可靠的周期性时间，因此有了原子秒，定义为铯-133 原子在其基态两个超精细能级间跃迁时辐射的 9192631770 个周期所持续的时间。

[4] 实数是虚数的子集。

虽然我们有非常正式的时间定义，但是在实际生活中大家不可能衡量每天的 $\frac{1}{86400}$ 到底有多长，也不会拿本子去数铯-133 原子是不是经过了 9192631770 个周期。因此我们需要其它的方法来提供一个相对准确的时间。电脑的时间是通过由主板 BIOS 的时钟电路来维护的。但是主板提供的这个时间不一定很准，温度的高低都有可能影响准确率。这时候就需要用一台我们认为准确的机器来校对时间不准的机器。这个校对通常是通过一个NTP服务[5]来提供的。

8.9.2 时间的作用

时间是个很常见的事物。大家的手机、电脑都会告诉你当前时间是多少。虽然时间很普遍，时间的标准却是国家级战略工程。中国科学院国家授时中心作为我们国家的唯一官方时间提供单位，其成立需要国务院和中央军委批准。原因很简单，火箭、导弹、卫星等都通过自己携带的电脑来做速度的计算。这些武器的运动速度非常快，一秒钟可以移动 7000 米。如果时间仅有 0.001 秒的误差，对应的则是 7 米的误差。

时间除了记录单个物体经历了多久以外，在分布式系统中时间更多是用来衡量不同节点的相对关系。在第8.8节 逻辑时间 中会有更多介绍。

8.9.3 时间的性质

在不考虑爱因斯坦的牛顿物理世界里，时间有个非常好的性质是它可以用连续的正实数在表示。因此时间的性质和正实数的性质一致。具体来说时间有如下特性：

1. 两个时间之间可以比大小。时间大表示后发生，时间小表示先发生。
2. 两个时间之间的时间间隔有比例关系。比如 3 小时时间间隔是 1 小时时间间隔的 3 倍。

虽然时间间隔之间是存在比例关系，两个时间点之间是不存在比例关系的，比如 2020 年不是 1010 年的两倍。存在这个区别的原因在于时间的值取决于参考点的设置，即 0 点是什么时候。如果是计算机，0 点是 1970 年 1 月 1 日 0 点。如果是公历，0 点是耶稣诞生的时候。

时间的另一个常见误解是人们通常想知道现在是什么时间。在日常生活中如果听见有人问现在几点了，大家会不假思索的看看表，报一个看到的时间。其实严格来讲，当你看到时间的一瞬间这个时间已经过去了，成为了历史时间。这也是人不能两次踏入同一条河流的原因[6]

8.9.4 时间的获取

时间是一个正实数，因此时间的精度是无限的[7]。由于计算机的数据表示形式是有限的，这就意味着我们需要对时间精度做一些取舍。一般来说精度越高，获取的难度越大。一般来说有以下几种获取方式：

1. 调用操作系统 API，获取本地机器当时的时间。
2. 通过 NTP 获取网络时间。NTP 是一个层级传播系统。最顶层通过高精度物理设备获取高精度时间。这个时间会逐层传播下去。用户通过 UDP 网络协议获取时间。
3. 通过 PTP[8]获取时间。PTP 精度在毫秒级以下，通常作为金融高频交易系统、电网、电信通讯等系统做时间同步用。PTP 假设时间的生产者通过广播的方式和消费者之间直接通讯，不通过路由器。

[5] 协议定义在互联网标准 RFC-1305。

[6] It is not possible to step twice into the same river according to Heraclitus, or to come into contact twice with a mortal being in the same state。

[7] 准确的说精度是可数的，即 countable。

[8] Precision Time Protocol。

4. 通过 GPS 获取时间。GPS 通常提供经纬度坐标和高度这 3 维数据。其实 GPS 还提供第 4 维数据时间。每个 GPS 内都有多个原子钟，这样就算某个原子中出了问题，备份系统也能提供准确时间。

8.9.5 时间存在的问题

时间本身没有问题，问题在于我们获取的时间源不准：

1. 主板时间可能因为温度的变化导致时间忽快忽慢。
2. 操作系统可能获取时间后因为内核调度的原因无法及时返回给应用层。
3. 虚拟机作为操作系统之上的应用程序，可能因为中断的原因无法及时返回给虚拟机内的应用。
4. 获取时间的线程可能因为内核调度的原因休眠，无法及时处理获取到的时间。

另外一个极少有人知道的事情是时间并不一定是增加的，时间可能会不变。闰秒是指为保持协调世界时接近于世界时时刻，由国际计量局统一规定在年底或年中（也可能在季末）对协调世界时增加或减少 1 秒的调整。最近一次闰秒在北京时间 2017 年 1 月 1 日 7 时 59 分 59 秒。这时出现了 59 分 60 秒这个奇怪的时间。如果系统使用的是 CLOCK_REALTIME 而不是 CLOCK_MONOTONIC[9]，会发现这一秒时间没有变化。

因此结论是我们很少有正确的物理时间。那怎么办呢？lamport time，vector clock

8.10 分布式快照算法

[9]CLOCK_MONOTONIC 记录了系统在启动后经历了多久的时间，因此是一个相对时间，不会因为外界时间的定义发生改变。

第九章　内存外置-缓存设计

核心思路：层级缓存

9.1　纯 kv 设计

9.2　可持久化 kv 设计

9.2.1　优化写

假设内存不够，需要硬盘支持。
完整的 lsm 设计。bloomfilter

9.2.2　优化读

B+tree
WAL，snapshot。
skiplist

9.2.3　总结

最后发现和业务系统的实现完全一样。只是状态为 kv。rocksdb。

9.3　kv 分区

一致性哈希

9.4　基于缓存的实时查询

外置缓存会有延时，需要和内部缓存一起解决
对于不在的情况可以用 bloomfilter 加速。内部缓存不在，bloomfilter 不在，就不用查外部缓存了。
核心思路：再加一层缓存。

第十章　数据持久化

OLAP，不是 OLTP

10.1　object relation mismatch

10.2　数据库架构

10.3　事务的定义

事务简化了一些异常处理模式。没有事务也是可以的。需要有更复杂的业务处理模式。
把并发伪装成非并发。
不同一致性层级的定义和实现方法。

10.4　事务实现

日志回滚
mvcc

10.5　数据存储方式

b 树，列存储

10.6　分布式数据库

2PC 和 TCC。
证明 2PC 和 serializability 等价。
Spanner。
不再可能有 store procedure 了

10.7　常见问题

有了缓存和持久化存储之后，解决一个经典的先写数据库还是先写缓存的问题。
存在三个系统：缓存，业务系统，数据库。
解决问题：数据的准确查询。
假设：业务节点无状态，查询通过缓存而不是业务节点来做。这样先写缓存掉电之后数据丢失，先写数据库造成缓存更新不及时。

如果业务节点保存一部分内部最新的缓存数据，查询通过业务节点走，没有任何问题。这时候如果是多个业务节点，可能写是一个节点，查询是另一个业务节点，这时候还是会不一致。这时候可以通过查询节点的更新时间来判断是不是读到了最新的结果。这时候会碰到时间不一致的问题。通过逻辑时间可以解决时间不一致，但是需要知道当前最新的逻辑时间是多少。

第十一章　严格一次执行

11.1　至少一次

定义：上游确认下游至少收到了一次。

实现：上游一直重试直到收到下游的确认消息。

上游视角：

1. 记录要发送消息。
2. 一直重试
3. 收到确认后更新消息状态

下游视角：

1. 收到消息
2. 记录消息
3. 返回确认消息

异常处理：

1. 上游消失，再选主后重试
2. 下游消失，通过服务发现找到新节点，再重试

每个人都有上游和下游，因此在入口和出口先存再处理。

11.2　严格一次

定义：处理一次，不是存储一次。

因此需要解决多次处理的问题。

解决方法：

1. 强上游：有连续的 ID，通过 ID 是否连续来判断。减少或者有空洞都有问题
2. 弱上游：没有连续 ID，只有 ID。需要通过和所有历史数据比对来发现是否重复

弱上游可能造成历史数据过多，需要减少。要和业务一起，增加 command 的过期时间，通过内存，硬盘和数据库一起做 dedup。

11.3　通过消息系统

消息系统没有 dedup 能力

消息系统自己的 ID 虽然是连续的，不能用来作为下游 dedup。

第十二章　数据处理

12.1　流处理

event 是状态的梯度
各种流的 join

12.2　reactive graph

高级版本的 excel

12.3　批处理

12.4　lambda 架构

第十三章　混沌工程

hehe

第十四章　反思

微服务是把数据问题推到了数据解决方案，分而治之。因为网络存在不确定性，要求业务和数据之间有很正确的交互逻辑。事件溯源把业务和数据放在了一起，消除了网络的不确定性带来的影响。但是业务和数据是一一绑定的，限制于单机性能。因此只有非 CPU intensive 的核心系统可以这么做。

参考文献

[Lam78] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*, 21(7):558–565, 1978.

索引