走向分布式

Release 1.0.1

ccshih

March 22, 2015

CONTENTS

1 Day 1: Scalability 2

2 Day 2: 分散式系统的面向 3

3 Day 3: Partition 5

4 Day 4: 为什麼有有些时候不要把 query 洒到所有机器上平行处理？ 7

5 Day 5: 资料切割的 metadata 管理 8

6 Day 6: Replication 9

7 Day 7: 无强一致性及无法决定执行顺序带来的问题 10

8 Day 8: 最终一致性 12

9 Day 9: CAP Theorem 13

10 Day 10: In-Memory data 15

11 Day 11: Zookeeper 16

12 Day 12: Zookeeper (续) 17

13 Day 13: Apache Kafka 18

14 Day 14: Apache Kafka (2) 19

15 Day 15: Apache Kafka (3) 21

16 Day 16: Apache Kafka (4) 23

17 Day 17: Apache kafka (5) 24

18 Day 18: Apache Kafka 与 Stream Computing 25

19 Day 19: 分散式资料系统 vs. 科层组织 26

20 Day 20: In-Memory 的技术议题? 27

21 Day 21: 分散式运算系统 28

22 Day 22: 分散式运算系统的沟通方式 29

23 Day 23: Stream Computing 的应用范围 30

24 Day 24: Stream Computing 特性 31

25 Day 25: 选择 Stream Computing 框架 32

26 Day 26: Stream Computing 框架的组成角色 33

27 Day 27: 如何追踪每一个 record 的处理进度 34

28 Day 28: 错误处理机制 35

29 Day 29: 从 Stream 到 Micro batch 36

30 Day 30: Stream States & Finale 37

Contents:

走向分布式

作者： ccshih

来源： http://ithelp.ithome.com.tw/profile/share?id=20060041&page=1

整理： 邓草原 [1]

[1] 整理本文并不意味本人同意文中所有观点

CHAPTER

ONE

DAY 1: SCALABILITY

一个系统走向分散式，一定有其不得不为的理由。Scalability 是最常见的理由之一。

我先简单的将 Scalabilty 的需求分成两种：

- Data Scalability: 单台机器的容量不足以 (经济的) 承载所有资料，所以需要分散。如：NoSQL
- Computing Scalability: 单台机器的运算能力不足以 (经济的) 及时完成运算，所以需要分散。如：科学运算。

在之後几天，我会试着就这两种需求来解析其中会遇到的问题与常见解法。

不管是哪一种需求，在决定采用分散式架构时，就几乎注定要接受一些牺牲：

- 牺牲效率：网路延迟与节点间的协调，都会降低执行效率。
- 牺牲 AP 弹性：有些在单机上能执行的运算，无法轻易在分散式环境中完成。
- 牺牲维护维运能力：分散式架构的问题常常很难重现，也很难追踪。

另外，跟单机系统一样，也有一些系统设计上的 tradeoffs

- CPU 使用效率优化或是 IO 效率优化
- 读取优化或是写入优化
- Throughput 优化或是 Latency 优化
- 资料一致性或是资料可得性

选择了不同的 tradeoff，就会有不同的系统架构。

下次我们就来谈一下，哪些面向的设计决策，会塑造出不同特质的分散式系统。

CHAPTER

TWO

DAY 2: 分散式系统的面向

昨天的重点归纳一句话就是：分散式系统都是特化的，而不是通用的。所以不同的设计决策就会衍生出不同用途的系统。

也如同昨天所说，我先大致将分散式系统分种两种：资料系统和运算系统

对於资料系统来说，主要的技术手段是 partition 和 replication，再搭配不同的读写方式就会有很多不同的变化。几个设计决策包括：

- 资料切割
- 读写分工
- 处理颗粒度
- 交易处理
- 资料复制
- 可用性保证
- 错误回复

对於纯运算系统来说，主要的技术手段是资料平行化和运算平行化。如果运算过程中会变更状态丶或是或参照易变的资料，状况就更加复杂。几个设计决策包括：

- 分工方式
- 讯息交换方式
- 支援的运算种类
- 交易处理
- 状态管理 & Rollback

- 可用性保证

接下来几天我就挑几个项目来开始讨论吧。

CHAPTER

THREE

DAY 3: PARTITION

分散式资料系统的两个问题根源：partition 和 replication。

先谈 partition。当资料放不进一台机器，或是对资料的运算太过耗时，单台机器无法负荷时，就是考虑 partition 的时候。

partition 就是把资料切割放到多台机器上，首先要考量的，就是要怎麼切资料。

资料有几种常见的切法：

- Round-Robin: 资料轮流进多台机器。好处是 load balance，坏处是不适合有 session 或资料相依性 (need join) 的应用。变型是可以用 thread pool，每个机器固定配几个 thread，这可以避免某个运算耗时过久，而档到後面运算的问题。
- Range: 事先定好每台机器的防守范围，如 key 在 1~1000 到 A 机器。优点是简单，只需要维护一些 metadata。问题是弹性较差，且会有 hotspot 的问题 (大量资料数值都集中在某个范围)。MongoDB 在早期版本只支援这种切割方式。
- Hash: 用 Hash 来决定资料要在哪台机器上。简单的 Hash 像是取馀数，但取馀数在新增机器时会有资料迁移的问题，所以现在大家比较常用 Consistent Hashing 来避免这个问题。Hash 可以很平均的分布，且只需要非常少的 metadata。但 Hash 规则不好掌握，比方说我们就很难透过自定 Hash 规则让某几笔资料一定要在一起。大部分的 Data Store 都是采用 Consistent Hashing。
- Manual: 手动建一个对照表，优点是想要怎麼分配都可以，缺点是要自己控制资料和负载的均衡，且会有大量 metadata 要维护。

除了切法之外，还要决定用哪个栏位当做切割的 key。

资料切割是非常应用导向的问题，因为有一好没两好，某个切法可能能让某种运算很有效率，但会害到其他种运算。

有一个 tradeoff 在读写之间，优化写可能会害到读。比方说 Round-Robin 可以平均写入负载，但 Range Query 就要到所有机器上查询。而如果用流水号的 Range 来切，有利於流水号的 Range Query，但写入会挤在同一台机器上。

另一个 tradeof f 在 application 之间，比方说一个用户表格资料如果用地区来切割，那就有利於常带地区条件的应用，因为这些应用可以只锁定几个 partition 进行查询，而不用把 query 洒到所有机器上。

为什麼我们不要把 query 洒到所有机器上，这就下次再讲好了。

CHAPTER

FOUR

DAY 4: 为什麼有有些时候不要把 QUERY 洒到所有机器上平行处理？

昨天讲到 partition，事实上 partition 比较常用在 write 需求高的应用 (平行写)，这是为什麼呢？

以前同事问过一个问题：既然有多台机器，那当然是把 query 分散到多台机器上啊。为什麼我们不想把 query 洒到所有机器上呢？

这问题的答案是：如果 query 很耗时，那分散的确会比较好；但如果 query 很快 (比方有用到 DB 的 index lookup)，那分散会增加效能降低的风险。

Hadoop 的 Map Reduce 就是透过分散提升效率，因为有很多资料要扫，所以分散是值得的。在这种状况下，效能的增加盖过效能降低的风险。

所谓效能降低的风险是指：因为要所有机器都回传资料後才能完成运算，所以运算时间是 Max(各台机器的处理时间)。当机器越多的时候，发生异常的机会越高，导致运算延迟。

一个现实的例子像是在桥开会时间，当与会人员越多时，就越有可能要花更久时间来协调，因为要等最慢回覆的那个人。

所以，在分散式资料系统中，如过查询费时，可以尽量分散；但如果查询很快，请尽量集中在少数机器处理。

不好意思，今天的文章有点水 ^^，请多包含 ~

CHAPTER

FIVE

DAY 5: 资料切割的 METADATA 管理

啊啊今天要谈什麼呢？来谈谈资料切割的 metadata 好了。

现在有好几台机器，都必须要 follow 同一套的资料切割方式，这个切割方式存在 metadata 中。这个 metadata 如果不见，那之後的资料就不知道该写入哪一台，且每次查询时都必须要广播找资料，这是很不方便的。所以要想办法保存好这些 metadata。

有些切割方式，像是 Hashing 的 metadata 量非常少，这是相对容易管的。但有些切割方式有很多 metadata，且有些方式在每次 insert 都要更新 metadata (bad practice~)，那这就麻烦了。

一个最简单的方式就是有一台机器专门管这些 metadata (meta server, config server...)，若需要 metadata 就来这边问。但明显的这会有单点问题。

现在常见的解法是用 Apache Zookeeper (ZK)，这是一个维持 cluster 中共同状态的分散式系统，透过 ZK 来维护这些 metadata 是许多分散式系统的普遍作法。ZK 有自己的 HA 和 consistency 机制可以保障资料，而且在 production 环境中一次要用 2n+1 (n>0) 个节点 (minumum = 3)，只要不要大於 n 个节点挂掉都可以正常服务。因为 ZK 里的资料

很重要!

很重要!

很重要!

因为很重要所以至少要保存三份!

ZK 为了保障资料的一致性，存取资料的手续有点麻烦。所以请不要因为 ZK 好用就让 ZK 太过操劳。

当然，如果 metadata 真的很少，又不大会更新的时候，连 ZK 都可以省掉。这就是完全 P2P 的系统了，像 Cassandra 就是这种。

CHAPTER

SIX

DAY 6: REPLICATION

今天来谈谈资料复制吧

资料复制是维持可用性的方法，因为资料复制好几份到不同机器，所以只要有一台机器还在，资料就拿的到。

但只要有资料复制，就一定会有延迟的状况，也就是在资料复制完成前，多台机器的资料是不一致的。

有的系统对於资料一致性读很要求，就会采同步复制，要复制完成後资料写入才会完成。但这样会很慢，尤其是副本越来越多的时候。

所以比较有效率的作法是非同步复制，但一定会有一段时间不一致。

那有没有折衷作法呢。有的，Quorum 就是一个折衷的作法，R+W>N，你可以控制要 write-efficient 还是 read-efficient，然後牺牲另一个 operation 的效率来换资料的一致性。

另外呢，每更新一笔资料就发一次复本更新是很没有效率的，通常要累积一些更新或隔一段时间才会 batch update。

常见的复制是有三个副本，除了原本的资料之外，同一个 rack 或 data center 一个副本，另一个 rack 或 data center 再一个。

那副本允不允许写入呢？多数资料系统是不允许的，也就是说，副本纯粹只是增加 read concurrency/efficiency/availabilty。同样是副本，同时只有一个 master 副本负责写入，其他的 slave 副本只负责 read request。

也有允许副本写入的系统，像是 cassandra。多个互相冲突的写入会以写入的 timestamp 订输赢 (所以 NTP 是必须的，但 NTP 也不能保证多台机器的时间完全一样)，Last write win。当然在还没协调前会存在有资料不一致的时间，那这就是应用必须要忍受的。

CHAPTER

SEVEN

DAY 7: 无强一致性及无法决定执行顺序带来的问题

昨天讲到多数系统不允许在副本写入，因为如果有好几个写入同时发生在不同的节点上，资料会不一致。就算能忍受资料不一致，也缺乏一个跨节点且精确同步的时钟来协调出这些写入的执行顺序，导致事後补救的困难。

假设有两个节点，两个不同机器的节点已经同步好帐户馀额 =15 元，有两个客户，一个要提款 10 元，一个要提 15 元，但是发到不同的机器上处理。两个节点都以为都会以为没问题 (馀额 >=0)，但实际上需要拒绝其中一个请求。

为了避免这种问题，许多资料系统都只允许在一个节点上写资料。如果有拆 partition，那每个 partition 内会只有一个 master 及零至多个 slave(replica)，只有 master 能写资料。再极端一点，有些资料系统为了避免读到不一致的资料，甚至会只允许在 master 上读资料，那这样 replica 就完全只是备援的角色，没有分散读取的能力。

上面那个例子，再退一步想。就算能允许用户馀额 <0，事後再补款。到最後若我们需要整理出执行顺序，也会遇到困难。因为在分散式系统里没有一个 global clock 能作为参照。

执行顺序这件事，对於某些资料操作的组合是重要的，对於某些资料操作的组合是不重要的。

之前举的那个是不重要的例子，因为先扣 10 元丶再扣 15 元；抑或是先扣 15 元丶再扣 10 元，最後的结果 (-10 元) 都是一样的。

但如果在中间加上一个计利息的操作，不同的顺序就会有不同的结果。假设利息是 10%，不同顺序的结果：

- 计利息 -> 扣 10 元 -> 扣 15 元：-8
- [*].5 元

- 计利息 -> 扣 15 元 -> 扣 10 元：-8
- [*].5 元

- 扣 15 元 -> 计利息 -> 扣 10 元：-10 元
- 扣 15 元 -> 扣 10 元 -> 计利息：-10 元
- 扣 10 元 -> 扣 15 元 -> 计利息：-10 元
- 扣 10 元 -> 计利息 -> 扣 15 元：-9
- .5 元

没有 global clock 的状况下，我们不能决定上面哪一个是正确的执行顺序。

CHAPTER

EIGHT

DAY 8: 最终一致性

昨天讲到执行时序的问题。当需要解决资料时序的问题时，表示已经放弃强一致性 (Strong Consistency) 了，转而只追求最终一致性 (Eventually Consistency)

最终一致性是说，只要资料不再更新，终有个时刻，所有节点会协调出一个一致的状态。这听起来相当的不可靠啊 XD

这的确是很不可靠。之前提到的执行顺序的问题，许多系统是利用 Vector clock，透过讯息传递来归纳出执行的时序。但由於 Vector clock 需要透过讯息交换 logical timestamp，才能整理出时序。所以如果有节点很孤僻不常跟其他人讲话，那推敲出来的时序就不精准。实际上，Vector clock 不保证能推测出完整时序 (total order)，只能推测出部分时序 (partial order)，也就是可能只能推敲出类似这样的结果：A < D, B < D, C < D，那你说 A, B, C 的顺序呢？很抱歉，因为只有部分时序，所以不知道。

所以真的是很不可靠啊，那为什麼还要用这种不可靠的东西呢？还是取舍的问题，因为想舍弃 Strong Consistency 来换其他的特性。

下次就来讲 CAP Theorem 罗，CAP 只能最多三选二 ~

CHAPTER

NINE

DAY 9: CAP THEOREM

CAP Theorem 的 CAP 分别是指：

- C (Strong Consistency): 在任何时候，从丛集中的任两个节点得到的状态都是一样的。
- A (Availability): 若一个节点没有坏掉，那它就必须要能正常服务。
- P (Partition Tolerance): 若一个丛集因网路问题或节点故障问题，被切割成两个 (或以上) 不完整的 sub cluster 时，系统整体还能正常运作。

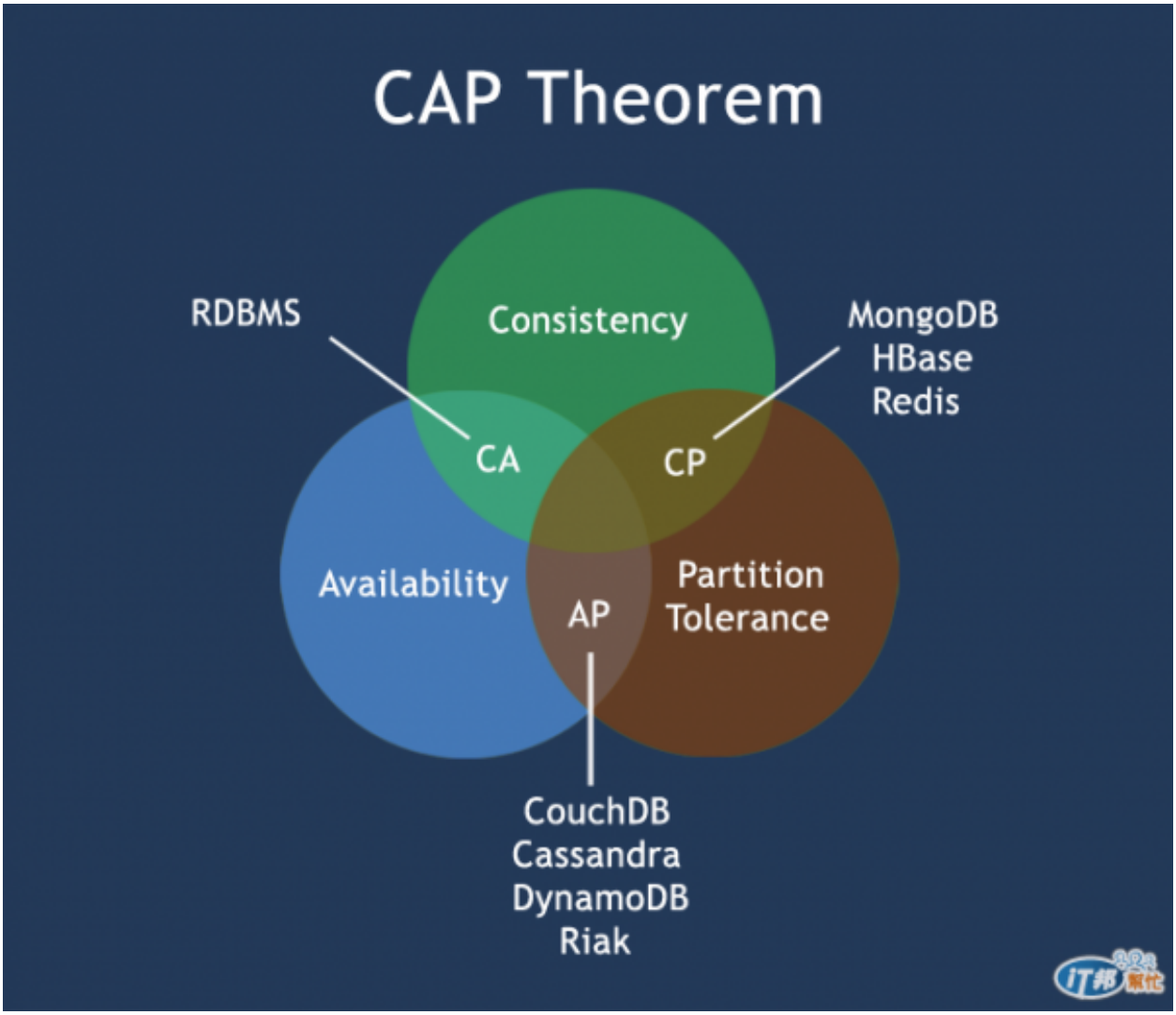
分散式系统中，这三个特性至多只有两个能同时存在。

假设有两台节点在不同机器上，如果存入资料的方式是 Two-phase commit，亦即所有节点同意後才能存入资料，那麼只要 partition 发生丶或任何一个节点故障，就不能运作，因此就只有 C 和 A。但这样的系统实在是太脆弱了 (记得 day 4 把查询洒到所有机器执行的风险吗?)，所以一般的分散式系统都会要求要有 P。

假设有两台节点在不同机器上，且必须要能容忍 Partition，那麼在 partition 发生时能怎麼做呢？

1. Keep Availability：两个节点虽然彼此之间不能沟通，但是既然都活着，就让他们都正常服务好了。这样其中一边的变化不能传递至另外一边，因此可能两边资料会不一致。更糟的状况是发生裂脑，就两边都以为自己是 master 可以写入资料，就会产生前两天提到的时序问题。(No consistency -- A 和 P)
2. Keep Consistency: 为了避免之前提到的不一致问题，因此必须停掉其中任一个节点 (No availability -- C 和 P)

网路上 CAP 的图很多，比方说这一张：



Source: http://www.w3resource.com/mongodb/nosql.php

没有意外的话，明天应该会讲一下 Zookeeper，Zookeeper 是偏向 CP 的设计，至於为什麼呢？等明天再说好了。

CHAPTER

TEN

DAY 10: IN-MEMORY DATA

本来今天应该要写 Zookeeper 的，不过看到这是第 10 天，想说来点特别的。所以临时插进来这个题目。

这个题目我不想讲太多，只是想丢个问题给大家来讨论一下。

In-Memory data 是想把硬碟当做是磁带的角色，把资料都放在 DRAM (or Flash, 现在暂时先不考虑这个) 里。单机的 DRAM 空间有限，所以要搭配一些技巧，像是：资料压缩丶分散式系统等等。

另外一个问题是资料保存的问题。DRAM 是 Volatile 的，也就是机器断电後资料就不见了，要怎麽让资料在断电後不会不见呢？有两个选项：

1. persistence: 在硬碟写 change log
2. replication: 在其他机器建立副本

大家觉得哪一个作法比较好呢？

CHAPTER

ELEVEN

DAY 11: ZOOKEEPER

现在很多分散式系统都会用 Zookeeper，在 Day 5 也有稍微提到一下 Zookeeper 可以用来维护 partition metadata。现在就来多介绍一些 ZK 的用处吧。

ZK 有几个常见的用法：

- 共享 Metadata
- 监控成员节点的状况 & 维护丛集的成员名单
- 协助选出丛集中的 leader(master)

ZK 的资料是以树状方式组织，树的节点叫做 znode，可以在 znode 里放资料。

由於 znode 里的资料是许多成员都关心的，所以 ZK 有一个 notification 机制，可以在 znode 里的资料更新时通知有事先对该 znode 注册 watcher 的 process。

有一种 znode 叫做 ephemeral node，用来监控成员的状况。这个 znode 跟建立 znode 的成员的 session 状况是连动的，若 session 一段时间没有回报 (heartbeat)，这个 znode 节点就会被删除。因此若有其他成员在此 znode 上设定 watcher，就会在此节点挂掉 (即 session 挂掉时) 收到通知。

所有成员都会对 master 对应的 ephemeral node 注册 watcher，所以在 master 失效後，会有成员侦测到而启动 leader election。

CHAPTER

TWELVE

DAY 12: ZOOKEEPER (续)

Zookeeper 能保证 global order，因为只有 leader 能处理写入要求。Zookeeper 在 partition 发生时仍能维持服务，因为采用了 Quorum。所以 Zookeeper 基本上是偏向 CA 的分散式系统。

所谓 Quorum 是指成员数达到最低投票门槛的成员集合。Zookeeper 成员有两者角色，Leader 或 Follower，一个 Quroum 里最多只能有一个 Leader，其他都是支持此 Leader 的 Followers。Leader Election 的目的就是要选出 Quorum 中的 Leader。

一个 Quorum 的成员数达到最低投票门槛，一般来说，是指成员数要大於 Zookeeper 节点的一半数量。比方说总共开了 5 个 Zookeeper 节点，Quorum 里最少要有三个成员 (3>2.5)。这样保证了，当丛集被 partition 成两半，其中一个 partition 的节点数是不足以形成 Quorum 的。若不足以形成 Quorum，就不允许对外服务。因此丛集中最多只有一个 Quorum 可以对外服务，就不会发生 inconsistency 的状况。

CHAPTER

THIRTEEN

DAY 13: APACHE KAFKA

Apache Kafka 是一个 Distributed Queue 的实现，很多 Stream Computing 平台都支援 Kafka 作为 data source。

Kafka 有几个特色：

- 分散式架构，所以天生就是容易扩充的。
- 基於磁碟空间，且避免随机存取。
- 因为储存空间大，因此 Queue 里的资料就算已消耗，也可以不用删掉。好处包括：其他新加入的 consumer 可以处理到过去的资料 (重要特色)。如果有 batch-oriented 的 consumer (如:Hadoop)，可以一次拉取足够大量的资料，以利 batch 的处理效率。
- 对资料的包装是轻量级的，且可压缩。避免掉不必要的物件包覆，可以直接以档案的型式来处理资料。
- 因为可以直接处理档案资料，直接用 OS 的 page cache，不需要额外 Applicaion Cache 来竞争珍贵的记忆体空间。

接下来的几天，我会再用之前讲过的维度，来介绍 Kafka 的分散式特徵。

CHAPTER

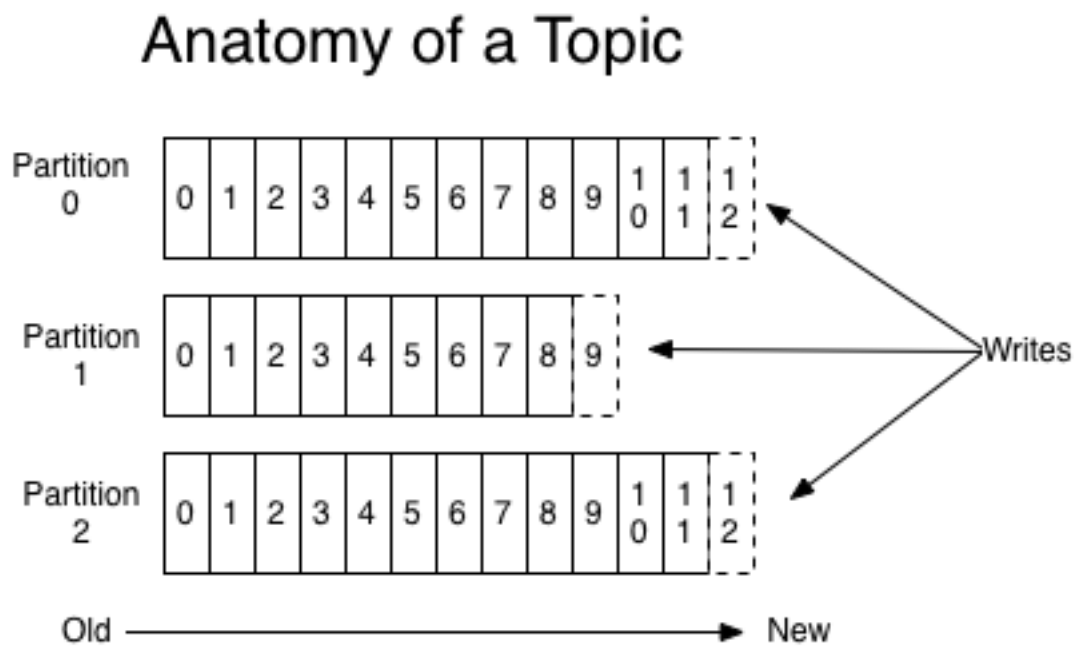
FOURTEEN

DAY 14: APACHE KAFKA (2)

先来介绍一下 Kafka 的基本架构吧 (以下图片都取自 Kafka documentation)。

基本上 Kafka 是一个 broker 的角色，仲介 producer 与 consumer。Kafka 一般是由多个节点所构成的 cluster。

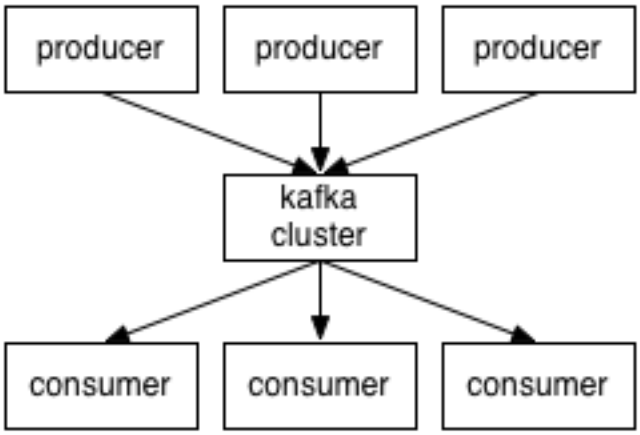
Kafka 有自己的 producer API 和 consumer API，produce/consumer client 必须要使用或依照 API spec 自行实作存取的方式。



一组资料流称为一个 Topic，为避免一个 Topic 的资料量过大，所以一个 Topic 可以分成好几个 Partition，每个 Partition 会在不同的节点上 (如果可以的话)。

Producer 必须要自己决定要将资料送到哪个 partition，在 producer API 里有一个参数可

以让使用者指定 partition key，然後 producer API 用 hash 方式决定 partition。



Kafka 可以弹性的支援 point-to-point 和 pub/sub 两种 Queue mode。主要是透过一个 Consumer Group 的抽象，每个 Consumer Group 当做是一个虚拟 consumer，但可以由多个实体的 consumer 组成。一组 Point-to-point 就是将所有 consumer 都划成同一个 Consumer Group；而 Pub/Sub 是将不同 Pub 的 Sub 分成不同的 Consumer Group

一个重点是，每一个 Consumer 只会同时 bind 一个 partition，也就是说，一个 Consumer 只会找一个 partition 来拉资料。[1]

这带来一些很重要的暗示，也是 Kafka 的限制所在，这些限制就下次再讲吧。

[1]准确地说，是一个 partition 只能同时被同一 consumer group 中的一个 consumer 消费，这样可以保证这个 partition 对于同一个 consumer group 来说不会被并发取。(by 邓草原)

CHAPTER

FIFTEEN

DAY 15: APACHE KAFKA (3)

以下是 Kafka 的设计所带来的限制：

1. Consumer Group 里的 consumer 数量不能小於 partition 数量。不然就会有 partition 里的资料对不到 consumer。[1]
2. 若各个 consumer 消耗资料速度不均，Partition 的消耗速度会失衡。也就是有的 partition 已经消耗到最近的资料，但有的 partition 还在消耗之前的资料。
3. 搭配上之前所说：「Producer 必须要自己决定要将资料送到哪个 partition」，所以失衡的状况无法透过 Producer 改善，又：「每一个 Consumer 只会同时 bind 一个 partition」，所以，失衡的状况无法藉由 produer/consumer 的 round-robin 来解决[2]。
4. 基於以上，每个 partition 实际上可以看成一个独立的 Queue。也就是说，虽然有 partition，但没有所谓跨 partition 的 total order，也就是只能保证各个 partition 自己的 local order。
5. 也就是说，若有一个 AP 会处理跨 partition 的资料，AP 不能假设会依 producer 产生的时序取得资料，而只能假设从每个 partition 里取得的资料是有按时序的 (Kafka 有保证，若 Producer 先送到 Partition 的资料，在 Partition 里也会排前面)。这就是在 Day 3 所讲到，partition 带来的 tradeoff。因此，若有 total order 需求的 AP 并不适合用 Kafka。

[1]这个说法不对。虽然一个 partition 应该只绑定到同一 consumer group 中的一个 consumer，但一个 consumer 是可以同时绑定到一个以上的 partition 的，只是，是否这么处理，取决于你是否认为消费端的并行能力能够适配多个 partitions。那么，partition 和 consumer 的数量究竟应该是怎么样一个关系呢？假设，partition 数量为 n, consumer group 中的 consumers 数量为 m，结合以上两点，我们可以得出，合理的配置是：m <= n，因为，每个 partition 只能绑定到 1 个 consumer，也即，最多只会有 n x 1 个 consumer 可以绑定到 partition，如果 m > n，就会有 m - n 个 consumer 是不会绑定到 partition 的，也即，不会接收到数据。(by 邓草原)

[2](Update 10/16): Kafka 有 rebalance 功能，若在 Consumer group 中新增 Consumer，会触动 rebalance, 重新分配 partition 与 consumer 之间的对应关系。这样有机会可以解决资料量失衡的问题。

简单来说，Kafka 假设，AP 是不需要 total order 的；抑或是 AP 只需要 by-partition 的 local order，只要适当的做好 partition，那麼就可以维持好时序的资料消耗。

Kafka 实际上也倚赖 Zookeeper 来储存 Partition 的 metadata，如同 Day 5 我们提到的常见作法。

以上，这是从 Partition 的角度来看 Kafka。明天就再从 Replication 观点来看 Kafka 吧。

CHAPTER

SIXTEEN

DAY 16: APACHE KAFKA (4)

今天来讲一下 Kafka 的 replication 机制

Kafka 的 replication 是以 partition 做单位，方法也很简单，就是让 replica 去订阅要追踪的 partition。因为这是 Queue，所以直接用订阅的机制就可以处理掉 replication。

在每个 replica set 里，只会有一个 master，这个 master 负责所有的读写工作，其他的 slave 都只是作备援，所以不会有 Day 7 讲到的不一致问题。

每个 replica set 会维持一个 ISR (In-Sync Replicas) 名单，名单内的 replica 是与 master 较为同步。若 master 挂掉，会从这些 ISR 中多数决挑选出新的 master。这个 ISR 的名单是会变动的。

在写入时虽然是以 master 作为窗口，但预设要等到 master 的资料同步到所有的 ISR，该笔资料才算 committed，也才能被 consumer 所看到。

CHAPTER

SEVENTEEN

DAY 17: APACHE KAFKA (5)

啊哈，没想到 Kafka 可以写到第五篇啊...

今天要讲的是 ack，ack 问题在 stream computing 里也会遇到，这边就来先提一下。

stream computing 的 ack 还要解决多阶段丶数量大的 ack 管理，而 Kafka 的 ack 只要确保 message deliver semantics。

分散式系统因为有透过网路，所以 message deliver 都是很难保证的。就像网路，有时候可以允许遗失封包 (UDP)，如果不能遗失封包 (TCP)，就要有重送和检查重复的机制。

对於 Kafka 的 consumer 来说，何时回送 ack，就决定了 message deliver semantics。

如果在资料收到丶但还没处理前就先送 ack，由於机器可能在处理前挂掉，所以不能保证资料一定被处掉到，所以是 at-most once semantics。

如果在资料收到且处理完後才送 ack，由於机器可能在处理後丶且送 ack 前挂掉，这样就会重送已处理过的资料，所以是 at-least once semantics。

而 Apache Kafka 现在还没有内建支援 exactly-once semantics。

CHAPTER

EIGHTEEN

DAY 18: APACHE KAFKA 与 STREAM COMPUTING

至於为什麼 Kafka 适合搭配 Stream Computing 呢？因为 Stream Computing 本质上也是一种可平行处理、易扩充的分散式处理架构，所以也要搭配同样可平行读写、易扩充的 data source/data sink 才能发挥最大的效能。且 Stream Computing 的处理过程要求 low latency，这是类似於 Queue 的需求，而非 Log aggregation Tool (如: Flume) 的需求。

Kafka 目前是实现 Lambda architecture 的要角。因为只有 Kafka 能同时满足 real-time processing 与 batch processing 对於 data source 的需求。

对於 Stream Computing 来说，Kafka 可以不只作为 data source/data sink，也可以作为 state commit log 的 sync 工具。

总之，对於 Stream Computing 来说，搭配 Apache Kafka 能带来许多好处。最後我用一段影片来终结对於 Apache Kafka 的介绍。这段影片是 Amazon Kinesis 的介绍影片，Kinesis 是 Amazon 上类似 Kafka 的服务，来看看 Amazon 是怎麼介绍 Kinesis 的吧。

CHAPTER

NINETEEN

DAY 19: 分散式资料系统 VS. 科层组织

让我们用科层组织来类比分散式资料系统，作为分散式资料系统的小节吧。

我们从 partition 和 replication 谈起，partition 就像科层组织，为了避免规模过大的管理困难，所以切割成多个可相对独立运作的单位来分而治之。但问题就是协调困难，尤其是在一个沟通管道不稳定的环境中。replication 像是职位的代理人，有些代理人可以很快上手，还可以帮忙分摊一些负担；不过有些代理人完全就是备援角色，甚至还有些代理人完全没进入状况。怎麼把这样一个障碍重重又各自为政的组织联合起来，让它能如臂使指，那就是分散式资料系统想做的事。

我们还介绍了一些重要工具，Zookeeper 像是个专人管理的中央布告栏，帮助组织间沟通协调，确保大家的认知是相同的。Zookeeper 非常尽责，你可以先跟他说你关心哪些公告，那些公告有更新的话还会主动通知你 ~

Kafka 像是个高效的公文传送工具，让单位之间的资料能顺畅流通。但 Kafka 可不会主动通知你有公文，你要自己去检查你的公文箱。不过 Kafka 会帮忙保留一些历史公文，所以只要不要拖太久才去检查公文箱，基本上公文都还找的回来。

基本上就是这样啦 ~

接下来要开始介绍分散式运算系统罗。

CHAPTER

TWENTY

DAY 20: IN-MEMORY 的技术议题?

这篇跟 Day 10 是同系列的，同样是关於 In-Memory 的问题。

受惠於 Memory 的性价比越来越高，越来越多厂商推出 In-Memory Computing 的 Solution，不过，所谓 In-Memory Computing 跟目前已知的运算方式有什麽差别？目前所有的运算也是将资料放到 Memory hierarchy 里才能运算啊。

In-Memory Computing 号称把 Memory 当做 Disk，而 Disk 当做磁带，可以省去将资料从 memory 持久化到 disk 的过程。除非是要 recovery，也无须从 disk 读取资料。因为 disk 和 memory 的速度是多个数量级的差距，不碰 disk 自然有加速效果，但同时也会受限於 Memory 大小，必须思考解决之道。

所以 In-Memory computing solution 多半会强调资料压缩能力，能把更多资料存进 Memory。有的 solution 还强调他们能更有效运用 CPU，如：降低 cache miss 、减少 lock/latch contention 等，因为 bottleneck 已经不再在 I/O 了，所以需要让 CPU 能更高效运作。

但以上这些议题，就算不是 In-Memory Computing 也是会遇到的。只是大资料的应用过去常卡在 I/O，所以还看不到这些议题罢了。

所以，In-Memory Computing 是一个真正的议题吗？还是只是 buzz word？大家来讨论一下吧。

CHAPTER

TWENTYONE

DAY 21: 分散式运算系统

谈到分散式运算系统，大家最熟悉的应该是 Hadoop。不过 Hadoop 是设计来处理 high throughput 的批次应用，相对来说不重视 latency。如果是要处理 low latency 应用的话就不适合用 Hadoop 了。

Hadoop 是典型的 Data parallelism，也就是将资料切成小块，每一块平行处理来增加处理时效。当然，这样的资料平行化精神，你也可以自己将资料洒在多机上，在多机透过 multi-thread / multi-process 来达成，如果真的还不能达到 latency 要求的话，就要考虑再加上 pipeline 处理。

用上 pipeline 的话，要把整个运算过程拆成好几个步骤，让上一步骤的单笔资料的产出尽速传送到下一步骤处理。这就像水在流动一样，资料循着 pipeline 往下流。

而 Stream Computing 结合了 Data parallelism 与 Pipeline，所以更加的复杂。

CHAPTER

TWENTYTWO

DAY 22: 分散式运算系统的沟通方式

作业系统有两种常用的 inter-process communication 方式：

1. Shared memory: 当做白板来交换资料，缺点是很多人用的话要排队 (lock)，效率不好。所以现在也演进出一些比较高效的共享方式，像是乐观锁、多版本控制等，但这些都有额外的 overhead
2. Message passing: 所有 process 间都透过讯息的方式来交换资料，缺点就是 Day 7 讲的，会缺乏 global order。

在分散式运算系统也是一样，如果一个运算的结果需要跟其他节点共享的话，那也需要透过每些沟通方式来达成。基本的思路也是上面这两种。

1. Shared data store: 找一个大家都能 access 到的 data store 来存资料，这个 data store 可能是某种分散式资料系统。
2. Peer Communication: 透过某些高效的通讯协定在各节点间交换讯息，通常是 Non-blocking 的通讯方式，而且还要用高效能的序列化框架。

CHAPTER

TWENTYTHREE

DAY 23: STREAM COMPUTING 的应用范围

Stream Computing 适用在有大量 event 涌进的应用，最常见的应用是 activity analysis，比方说即时分析用户在网站的浏览、点击行为，即时针对行为给予不同的个人化。

alert 也是很常见的 streaming 应用，常见的是分析系统的 log，在发现有问题时即时通知管理者。再更复杂一点，可以做 fraud 诈欺分析，比对多个来源资料来分析一笔交易是否为诈欺，尽早处理以避免损失扩大。

有个比较少见，但也很重要的应用是即时扣款，尤其用户规模大的时候。但扣款的公式可易可难，牵涉到的因素以及各种的副作用都会影响到实作的难度。

CHAPTER

TWENTYFOUR

DAY 24: STREAM COMPUTING 特性

突然发现我好像还没介绍过 Stream Computing :D

Stream Computing 是设计给需要 low-latency 的应用。batch processing 因为整批进整批出，有些已处理好的资料也需要等待其他同批资料都处理完才能一次送出，这样会导致一些不必要的 latency (类比: 跟团在跑景点时总会有些动作慢的拖到大家的集合时间)。为了要减少 latency，Stream Computing 将处理颗粒度变小到 record，且将处理过程切分成好几个阶段。透过 pipeline 的方式，只要前一个阶段处理完的 record，就可以马上进入下一个阶段，这样可以避免掉不必要的 latency。

但因为这种处理方式会增加资料传递量，因此 throughput 会比 batch processing 还低。为了解决这样的问题，Stream Computing 框架都会内建 Scalability。每一个阶段的处理程式都是可扩充的，也就是可以第一阶段用 10 个 thread 、第二阶段用 5 个 thread 之类的。这种用 efficiency 换 scalability ，再用 scalability 弥补 efficiecy 的方式，在分散式系统里相当常见。

如果是纯运算的应用，没有很多的 reference data 或其他的 side effect 的话，Stream Computing 有很好的 scalability。但有许多的应用并不是如此，也因此限制了 Stream Computing 的应用范围。目前 Stream Computing 最常被用来算即时的统计 (包含了 window count，如近五分钟的统计)，因为这些统计数字可以单纯从 input data 中算出，且计算结果有对分散式系统友善的单调渐增 (monotonic) 特性。

如果是其他的应用，牵涉到其他更复杂的参照资料或运算时，就需要再搭配 partition 或 replication 等分散式资料系统的处理方式。

CHAPTER

TWENTYFIVE

DAY 25: 选择 STREAM COMPUTING 框架

目前有几种开源软体可以选择：

1. Apache Storm
2. Apache Samza
3. Apache Spark Streaming

如果希望有 sub-second level latency，请选 1

如果不想踩太多雷，请选 1

如果处理过程要储存很多状态，请选 2

如果想要深入研究 Stream Computing，请选 2

如果已经在用 Spark，请选 3

如果希望 programmer-friendly，请选 3

以上是我的建议，给大家参考一下罗。

CHAPTER

TWENTYSIX

DAY 26: STREAM COMPUTING 框架的组成角色

虽然 Day 25 有提到好几种 Stream Computing 框架，但是这些这些框架都有一些共通的组成元素：

从角色来看，分成几组：

1. 处理 client 提出的运算要求，将运算工作拆分成小单位 (任务) 後，将运算工作分派到各个运算节点上
2. 管理运算节点，每台机器需要一个或多个这种角色
3. 运算节点，实际执行运算，并将运算结果透过高效率的方式送交到下一阶段的运算节点或中继站
4. 资料管道，担任运算节点之间的中继站，或是输入资料的中继站。若是 consumer 速度慢於 producer 速度时负责 buffer
5. 丛集协调中心，负责协调及交换整体丛集的状态。

其中 4 和 5 已经介绍过了，一般常用的是 Kafka 和 Zookeeper。

1~3 的话，每一种框架有不同的实作方式。

明天开始我会简单介绍一下 Day 25 提到的三种框架，然後就可以 happy ending 罗 ~

CHAPTER

TWENTYSEVEN

DAY 27: 如何追踪每一个 RECORD 的处理进度

在 Stream Computing，一笔 record 可能会需要同时进行好几种运算 (如: 更新各种 counter，计算统计值等等)。我们可以把一笔 record 从源头到完成所有计算看做是一个有向无循环图 (Directed Acyclic Graph，DAG) 。

另外，一笔 record 所有运算可能分散在多个点节上处理，不见得每一个 record 只会在一个节点上处理。

这有点像是，我们雇了 100 个人，每个人可以从好几个上游接工作，做一些处理後，再往好几个下游丢。这样的处理方式很难追踪是否每个工作都顺利完成。而这正是 Stream Computing 要面对的问题。

接下来我以 Storm 为例，说明如何在 DAG 里追踪每一个 record 的处理进度。

每一笔 record 在最源头会被指派一个 message ID，在每次处理完产生新资料的後续传送中，都会带这个 ID，以分辨资料的源头。有可能一笔资料有多个源头 (如: 经过 join 之後)。这个 ID 会被用来追踪，与该 record 相关的所有运算的完成状况。

在每个阶段处理完一笔 input data，产生 output data 往下游送时，会向一个特别的角色 (Ackor) 发出 ack 或 fail 的回报，并带有 input data 与 output data 的 64 bit ID。在正常处理的状况下，ackor 会收到两次同一个 id (一次是产生该资料丶一次是该资料处理完)，表示该笔资料已经被处理完。由於资料量很多 (每个 record 可以衍生出许多笔资料)，所以不可能为每笔资料都维护一个 counter。ackor 采用的方式是为每一个 record 维护一个初始值为 0 的 64 bit 的 value，每次收到一个 id 回报，就将 value <- value XOR id。如果每个 id 都出现两次，那麼 value 又会回到 0，就表示该 record 相关的处理都完成了。

CHAPTER

TWENTYEIGHT

DAY 28: 错误处理机制

昨天讲到，Storm 用 Ackor 将所有收到的 ID XOR 之後，来侦测一笔 record 是否已完全被处理。今天来讲一下，如果遇到问题的话会怎麼处理。

其中一个常见的问题是：运算节点死掉了。所以 Ackor 收不到某些 data id。这种状况是用一个 timeout 机制来解决，如果一段时间内没收到该资料的第二个 id 回传，就直接宣判该笔 record 处理失败，要求资料来源 (e.g. Kafka) 启动重传机制。也因为这样，Storm 不能保证 exactly-once semantics (ref. day 17)。可能有些资料处理到一半後才重传，那前半段就会被处理两次。如同 day 24 所说，这是 pure stream computing 的特性，会限制 stream computing 的应用范围。

那 Ackor 死掉要怎麼办？前面没讲到的是，资料来源也会设 timeout，如果 Ackor 挂了而无法向资料来源发出 message 的 ack，message(record) 也会被重送。当然这也是会有不能保证 exactly-once semantics 的状况。

那如果资料来源死掉呢？那就真的死掉了。所以像 Kafka 会需要用 replication 维持强大的 availability，且需要发展 stateless 的 consumer，以便於在错误後重启亦能轻松回覆运作。

CHAPTER

TWENTYNINE

DAY 29: 从 STREAM 到 MICRO BATCH

昨天讲到 pure stream computing 不能提供 exactly-once semantics，不过有许多应用都需要 exactly-once semantics。

为了解决这问题，开始有些框架不走 pure stream，而是走 micro batch。

Micro batch 也是集结一段时间的资料再批次处理，只是集结的时间很短，通常是几秒就集结一次，所以称为 Micro batch。

Micro batch 的好处是可以实现 exactly-once semantics。而 exactly-once semantics 主要牵涉到 state 的更新，我明天会提到这个问题。但 micro batch 的坏处在於 latency 变高了，但如果应用对於 latency 没那麼要求 (可容忍秒级的延迟)，用 Micro batch 其实很 OK。

Storm 有一个延伸框架 Trident，在 Storm 上多架一层 Coordination layer 来实现 Micro batch。但有点讨厌的是，有些 Storm 有的功能 (如: window count)，在 Trident 里还不支援；而且 Trident 和 Storm 的运算结果是不相通的 (不能混用)。

而 Spark Streaming 本身就是 Micro Batch，因为这是把 Spark 的批次维度缩小後的实作。

CHAPTER

THIRTY

DAY 30: STREAM STATES & FINALE

这是最後一篇文章罗，我要来介绍一下 stream computing 里状态的管理。

所谓状态是指处理过程中的副作用，比方说，要更新 counter。如果没有 exactly-once semantics，一笔资料可能被处理多次，那麼更新多次 counter 就会产出错误的结果。

在 Storm 里是在 micro-batch 里用一个 transaction id 表示 batch 的顺序，并且保证 state 的更新会依 id 顺序处理。在储存 state 时除了存最新状态值之外，还会存上一版的状态值。因此，如果遇到同一个 micro-batch 被重新处理，只要把上一版状态取出後重新更新即可。这样就保证了每一个 micro-batch 只会更新到一次状态。

好了，exactly-once semantics 问题解决了，但状态带来的另外一个问题是效率问题。虽然 Stream Computing 有效的分散运算，可是所有状态更新都集中在同一个地方，还是会变成处理瓶颈。这就是我在前半部谈到的分散式资料系统要解决的问题。不过，若运算系统和资料系统是独立运作的，更新时的 network latency 会对某些要求低延迟的应用造成伤害。在这方面，Samza 有个聪明的解决方法。大家可以去看看。

好了，来总结一下吧。前半部分，我先提出一些分散式系统的设计决策，这些设计决策也可以在各种分散式系统中 (e.g., Hadoop) 看到。然後开始介绍分散式资料系统，以 Zookeeper 和 Kafka 作为范例。而之所以会选这两个，是因为它们常是 Stream Computing 框架中的元件。後半部介绍分散式运算系统，主要是以 Stream Computing 为主。Stream Computing 因为要求低延时，所以除了一般分散式系统的问题之外，还要面对细处理颗粒所带来的问题。各种不同的框架 (Storm, Samza, Spark Streaming) 各有其优缺点。纯 Stream computing 因为不保证 exactly-once ，所以无法保证数值会绝对正确。替代方案是改用 Micro batch，但这会增加一些 latency。不过，为了增加 Stream Computing 的应用范围，许多框架都支援 (或是原生) micro batch。

未来的资料运算系统，都要结合分散式资料系统和分散式运算系统 (如: HDFS & MapReduce)。不像 Hadoop 已经有完整解决方案，Stream Computing 仍然在寻求资料运算系统

的最佳组合。

以上就是 30 天的分享内容，希望能对大家能有些帮助。