12 | 分布式系统设计：数据一致性与故障容错的纠葛

2022-11-05 郑建勋 来自北京

《Go进阶 · 分布式爬虫实战》

课程介绍 >

讲述：郑建勋

时长 13:59　大小 12.77M

你好，我是郑建勋。

前面几节课，我们介绍了微服务为什么是一种自然演进的架构，也讨论了微服务架构伴随而来的问题。

微服务可以分散到多个机器中，它本身是分布式架构的一种特例，所以自然也面临着和分布式架构同样的问题。除了我们之前介绍的可观测性等问题之外，微服务还面临着分布式架构所面临的核心难题：数据一致性和可用性问题。

这节课，我们还是循序渐进地看看，随着系统的发展，为什么必然会面临数据一致性问题，又怎么在实践中解决这类问题。

数据一致性的诞生背景

在微服务架构中，服务一般被细粒度地拆分为无状态的服务。无状态服务（stateless service）指的是当前的请求不依赖其他请求，服务本身不存储任何信息，处理一次请求所需的全部信息要么都包含在这个请求里，要么可以从外部（例如缓存、数据库）获取。这样，每一个服务看起来都是完全相同的。这种设计能够在业务量上涨时快速实现服务水平的扩容，并且非常容易排查问题。然而我们也需要看到，这种无状态的设计其实依托了第三方服务，比较典型的就是数据库。

以关系型数据库 MySQL 为例，在实践中，随着我们业务量的上涨，一般会经历下面几个阶段。

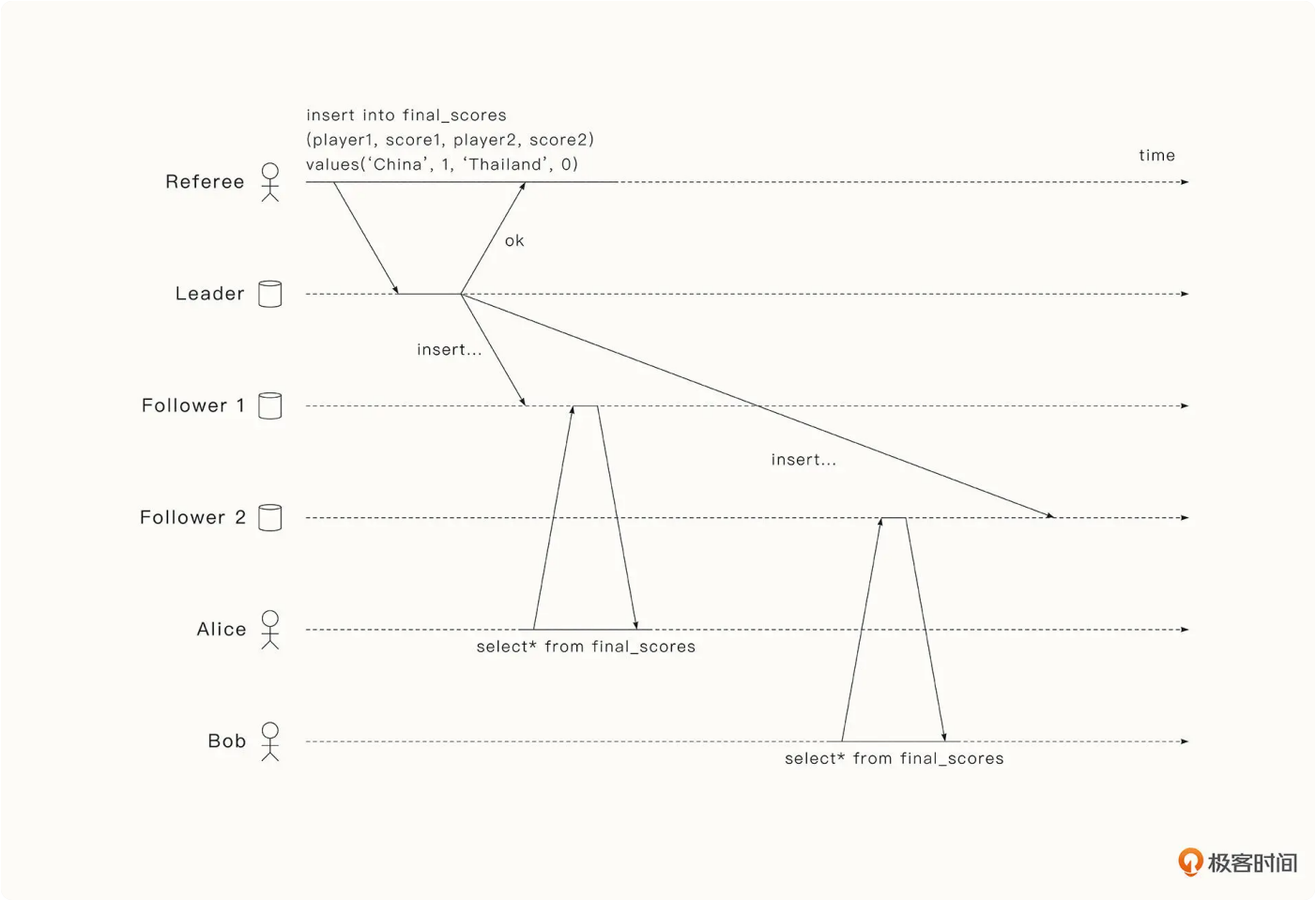
1. 硬件的提升：选择更强的 CPU、更大的内存、更快的存储设备。
2. 设计优化：通过增加缓存层减轻数据库的压力、利用合适的索引设计快速查找数据、使用监控慢查询日志优化不合理的业务 SQL 语句。
3. 服务拆分：拆分后，子系统配置单独的数据库服务器。
4. 分库分表：通过 ID 取余或者一致性哈希策略将请求分摊到不同的数据库和表中。
5. 数据备份：例如，将存储 1 年以上的数据转存到其他数据库中。
6. 主从复制与读写分离：将 Leader 节点数据同步到 Follower 节点中，一般只有一个 Leader 节点可以处理写请求，其余 Follower 节点处理读请求，这样可以提高数据库的并发访问。

从上面的优化中我们可以看到，拆分是解决大规模数据量的利器。但是，当数据分散到更多的机器，或者当我们希望通过主从复制实现可用性和读写分离时，我们也面临着新的问题：数据一致性。

这里我举一个 MySQL 常见的主从复制导致数据不一致的例子。假设 Alice 与 Bob 都在查看一场中国队对战泰国队的足球赛，中国队在结束时打入点球，系统管理员将最新的数据比分更新到数据库中。但是 MySQL 数据库采用了主从复制的架构，而且一般是一个 Leader 节点处理写操作。所以当 Leader 把数据异步同步给其余 Follower 的时候，Follower 收到数据的时间差可能会导致出现如下几种情况。

- 第一种情况：Alice 与 Bob 访问数据时，数据还未同步到 Follower A 和 Follower B，因此 Alice 与 Bob 得到的是过时的比分数据。
- 第二种情况：Alice 与 Bob 访问数据时，数据已经同步到 Follower A 和 Follower B，因此 Alice 和 Bob 可以同时看到最新的比分数据。

- 第三种情况：如下图所示，Alice 访问 Follower A 时，数据已经同步到 Follower A，因此 Alice 看到的是最新的比分数据。但是 Bob 访问访问 Follower B 时，数据还未同步到 Follower B，所以 Bob 得到的是过时的数据。更糟糕的是，由于网络中断等原因，数据同步的时间是不可控的，也就是说，Bob 什么时候能看到数据是不确定的。此外，如果 Alice 再次访问，也可能会访问到 Follower B，再次得到过时的数据。



上面第三种情况就是数据不一致的具体体现，而且在实际生产环境中其实是经常发生的。这种情况在有些场景下是让人难以接受的，想想在银行转账和查看银行余额时遇到这类问题，你会有什么反应？

数据一致性问题，本质上是分布式架构相比单个程序而言有着巨大的不确定性。在分布式系统中会遇到下面这些问题。

- 网络延迟：消息到达的时候有时延，而且不确定。
- 网络分区：网络可能被分割为多个互不连通的区域。
- 系统故障：硬件问题、断电、内核崩溃导致部分机器故障，当机器数量越来越多时，机器故障也变成了大概率事件。

- 不可靠的时钟：这意味着我们无法依靠绝对的时钟来确定操作的顺序。

前面我们举的那个主从复制的问题，本质上是网络延迟导致的。当网络恢复时，数据是能够完整同步到 Follower B 的，因此我们把这样的一致性称为最终一致性。**最终一致性指的是从长远来看，数据最终能够到达一致的状态，但过程中可能会读到过时的数据。**

也就是说，数据的一致性有多种维度去衡量，当我们在设计分布式架构时，我们的场景能够容忍哪一种类型的数据不一致，通常是决定我们架构设计和技术选型的重要因素。

比最终一致性更严格的一致性保证被称为线性一致性。在这里，我无意陷入到讨论学术概念的旋涡中，因为系统论述线性一致性是一个比较复杂的话题。这里我想说的是，线性一致性能够推导出我们更加常见的概念：**强一致性。即在更新完成之后，任何后续的访问都会返回更新的值。**如果我们上面案例的数据库遵循了线性一致性，那么就不会出现读出过时数据的情况。

那么问题来了，我们要怎么设计架构，才能让系统有更强的数据一致性保证呢？

在上面的主从复制架构中，**我们可以强制让读写都通过 Leader 节点实现，或者强制要求 Leader 节点复制到 Follower 节点之后，才能完成后续操作。**但我们很快又会发现新的难题：可用性问题。

例如，如果我们有一个 Follower 节点崩溃，那么系统是不是需要一直陷入等待，变得不可用了呢？很显然，分布式数据一致性其实是一种权衡。当我们希望保证更强一致性的时候，也必须要牺牲一些东西，这就是有名的 CAP 定理告诉我们的内容。

CAP 定理

CAP 定理的三个属性具体来说分别是：

- C（Consistency），线性一致性；
- A（Availability），可用性，意思是即便有失败节点不可用，其他节点仍然能正常工作，并对每一个接收到的请求给出响应；
- P （Partition tolerance），分区容忍度，指能够容忍任意数量的消息丢失。

CAP 理论证明，在异步网络中，这三个属性不能同时获得。这三种属性排列组合，可以得到 CP、AP、CA 三种类型系统。但由于分布式系统无法保证网络的可靠性，因此我们实际面临的是 CP 系统或者 AP 系统的选择，即在线性一致性与可用性之间进行权衡。不过其实在引入 CAP 定理时，我们就凭直觉点出了这一点。

CAP 理论论证过程中的条件是非常严格的：必须保证一致性是线性一致性；可用性指的是所有的请求都需要有回应；分区容忍只考虑了网络分区，没有考虑其他故障。另外还要强调的是，一个系统不可能同时拥有 CAP 三种属性，但这并不意味着放弃了其中一个属性，就一定会有另外两种属性。也难怪在《Designing Data-Intensive Applications》一书中也提到，尽管 CAP 在历史上具有影响力，但它对于设计系统的实用价值不大。

在分布式系统的设计中，线性一致性与可用性之间需要进行一些妥协。在实践中，很少有系统实现了真正的线性一致性，这是因为在可信的网络中，异常和网络延迟等情况其实是可控的。而要保证线性一致性，系统在正常情况下也要付出许多性能上的代价。

而对于可用性来讲，我们还需要考虑系统在异常情况下的故障容错性，保证服务正确且可用。虽然 CAP 理论中的 P 只考虑了网络分区的容错，但其实正如我们之前提到的，系统还可能遇到网络延迟、系统故障等问题。仍然以之前提到的主从复制案例为例，这个场景在正常情况下工作良好，也能够实现最终的数据一致性。但是如果 Leader 节点挂了怎么办？ 如果挂掉的节点没来得及将数据同步到 Follower 节点，当其中一个 Follower 节点提升为 Leader 节点时，这些没来得及同步的数据就会丢失。要解决这些问题，就需要依靠共识算法来保证了。

共识算法

共识算法保证系统中的大部分节点能够就同一个意见达成一致，只有这样才能在小部分节点“失联”的时候，保证大部分节点可用，同时保证数据的正确性。

要考虑到各种可能的异常情况，还要兼顾并发的读写，达成共识并不是一件容易的事情。其中比较有名的共识算法是：Paxos、Raft、Zab。下节课，我们还会深入讲解介绍这些算法。

分布式协调服务

分布式容错和数据的一致性实现起来很困难，在实践中，我们也很少自己去实现分布式算法，因为即便是最简单的 Raft 算法，要保证其正确性，或者要排查问题都异常艰辛。

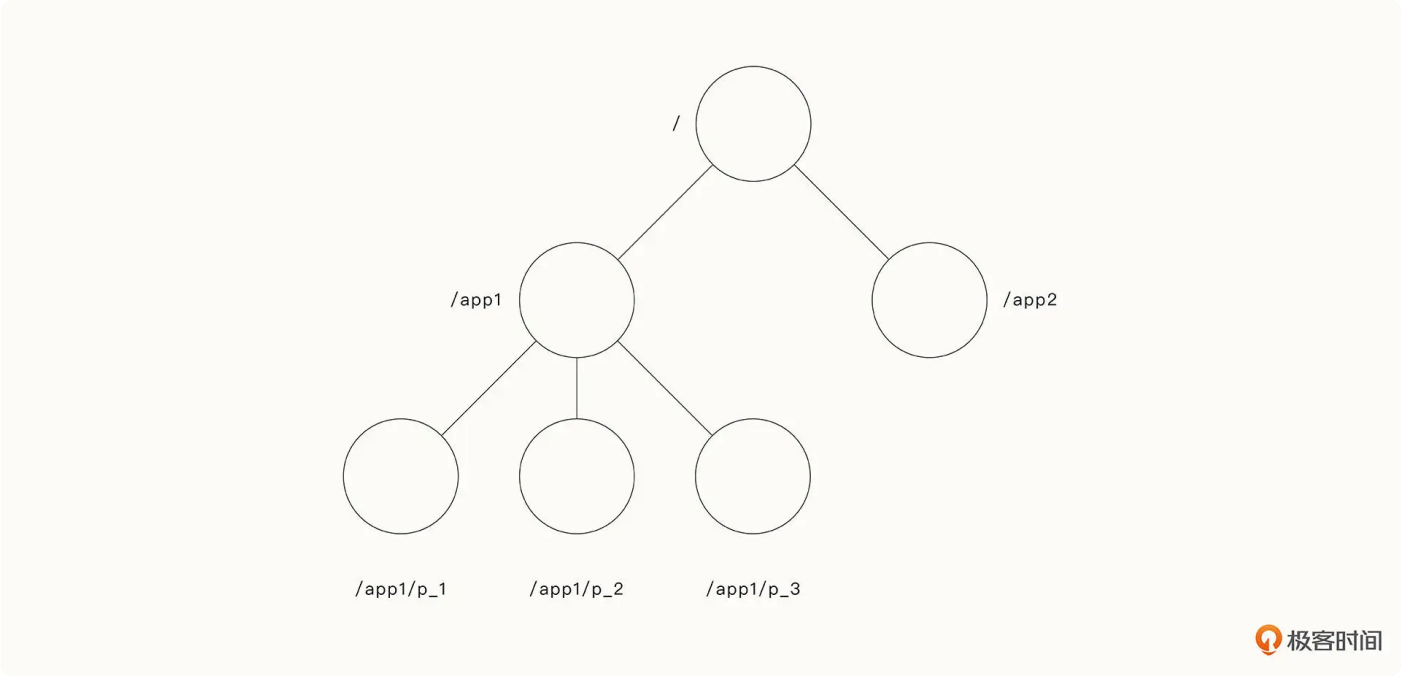
通常，我们会借助那些设计优秀、经过了检验的系统，帮助我们更容易地实现分布式服务之间的协调。这种系统被称作分布式协调服务，其中比较熟知的开源项目有 ZooKeeper、etcd。

这些服务通常具有友好的 API 设计，在这里我以 ZooKeeper 为例来说明分布式协调服务的使用场景。如下图，ZooKeeper 的数据模型类似于 Unix 文件系统，其中，Znode 是客户端通过 ZooKeeper API 处理的数据对象，Znode 以路径命名，通过分层的名称空间进行组织。

Znode 包含应用程序的元数据（配置信息、时间戳、版本号），它有两种类型：

- Regular（常规的），客户端通过显式创建和删除来操作常规 Znode；
- Ephemeral（临时的），此类 Znode 要么被显式删除，要么被自动删除（系统检测到会话中止时）。

为了引用给定的 Znode，我们使用标准的 Unix 符号表示文件系统路径。例如，我们使用 /A/B/C 表示 Znode C 的路径，其中 C 的父节点为 B，B 的父节点为 A，除 Ephemeral 节点外，所有节点都可以有子节点。Znode 命名规则为：name + 序列号。一个新 Znode 的序列号永远不会小于其父 Znode 之下的其他 Znode 的序列号。



ZooKeeper 提供了对用户友好的 API 用于操作 Znode，这些 API 包括了：

复制代码

```
create(path, data, flags)
delete(path, version)
```

```
exists(path, watch)
getData(path, watch)
setData(path, data, version)
getChildren(path, watch)
sync()
```


ZooKeeper 对数据的一致性有一些重要的保证：

- ZooKeeper 进行的所有写操作都是线性一致的，可以保证优先顺序；
- 每一个客户端的操作都是 FIFO 顺序执行的。

对 ZooKeeper 进行读操作时，因为可以直接在 Follower 中执行，所以确实有可能读到过时的数据。针对这个问题，ZooKeeper 提供了 sync() 方法来实现读的线性一致性。此外，通过允许读取操作返回过时数据，ZooKeeper 可以实现每秒数十万次操作，适用于多读而少写的场景。

基于分布式协调服务的特性，我们可以在应用服务中构建分布式锁，进行配置管理，并完成服务发现的工作。

分布式锁

基于 ZooKeeper 可以实现分布式锁，这是基于写操作的线性一致性保证。它的基本思想是每个客户端都创建一个 Znode，所有的 Znode 形成一个单调有序的队列，而排在队列前面的客户端能够优先获得锁，其余客户端陷入等待。

当锁释放时，下一个序号最低的 Znode 能够获得锁，这种机制还能够避免惊群效应，其伪代码如下所示：


```
acquire lock:
      n = create("app/lock/request-", "", empheral|sequential)
    retry:
      requests = getChildren(l, false)
      if n is lowest znode in requests:
        return
      p = "request-%d" % n - 1
      if exists(p, watch = True)
        goto retry

     watch_event:
```

```
12          goto retry
```


配置管理

分布式协调服务也可以实现分布式系统中的动态配置。当服务启动时，连接 ZooKeeper 获取配置信息，让 ZooKeeper 与服务保持连接。当配置发生变更时，通知所有连接的进程，获取最新的配置信息。

服务发现

在分布式系统中，服务可能随时扩容、重建或者销毁。因此，当服务启动时，需要自动注册自己的 IP 等信息到注册中心。这样客户端可以获取最新的服务信息，并采取负载均衡策略将请求均匀打到下游服务。

服务发现的另一个场景是监听服务的变化。例如调度器为了实现合理的调度，会监控 Worker 服务数量的变化，并及时调整任务的分配。这样，当一个 Worker 崩溃时，就能够及时将 Worker 上的任务转移到其他 Worker 中了。

无信网络中的共识问题

借助共识算法，我们可以实现服务的一致性以及故障时候的容错性。不过这里有一个大的前提你可能没有注意到，那就是，我们假设系统中的节点都是可信任的。然而，在一些网络中，节点并不一定是互信的，这就导致我们可能遇到**拜占庭将军问题。**它的意思是，在分布式系统中，当系统中的节点发送错误或欺骗性的信息时，节点之间无法达成一致。

莱斯利·兰波特在它的论文中描述了这个问题，这里我引述一下维基百科中的描述：

> 一组拜占庭将军分别各率领一支军队共同围困一座城市。为了简化问题，将各支军队的行动策略限定为进攻或撤离两种。因为部分军队进攻部分军队撤离可能会造成灾难性后果，因此各位将军必须通过投票来达成一致策略，即所有军队一起进攻或所有军队一起撤离。因为各位将军分处城市不同方向，他们只能通过信使互相联系。在投票过程中，每位将军都将自己投票给进攻还是撤退的信息通过信使分别通知其他所有将军，这样一来每位将军根据自己的投票和其他所有将军送来的信息就可以知道共同的投票结果而决定行动策略。

系统的问题在于，如果将军中出现叛徒，他们不仅可能向较为糟糕的策略投票，还可能发送错误的投票信息。假设有 9 位将军投票，其中有 1 名叛徒。8 名忠诚的将军中 4 人投进攻，4 人

投撤离。这时候叛徒可能故意给 4 名投进攻的将领送信表示投票进攻，而给 4 名投撤离的将领送信表示投撤离。这样一来，在投进攻的将领们看来，投票结果是 5 人投进攻，会因此发起进攻；而在投撤离的将军们看来，则会发起撤离。这样各支军队的一致协同就遭到了破坏。

James A. Donald 在给中本聪的信件中，曾对拜占庭将军问题有更精彩的描述：

> 每一个人都知道 X 是不够的，还需要每一个人都知道每一个人都知道 X，但这还是不够的，还需要每个人都知道每个人都知道每个人都知道 X。

拜占庭问题是分布式系统中最难解决的问题之一。目前已经有不少理论用于解决拜占庭问题，而这其中让人最震撼、最跨时代的解决方案无疑是比特币带来的。

2008 年，正当金融危机席卷世界之际，人们开始反思当前经济社会、金融秩序所面临的问题。2008 年 11 月，一位化名为“中本聪” 的研究者在密码学邮件组中发表了比特币的奠基性白皮书《比特币：一种点对点式的电子现金系统》（Bitcoin: A Peer-to-Peer Electronic Cash System）。

在比特币白皮书中，中本聪阐述了一种他称之为“比特币”的系统及其实现方式。比特币系统被设计为分布式的网络，每个节点都有一份完全相同的账薄，任何人都可以直接验证区块链中的信息，不依靠任何第三方组织就能完成交易并保证交易的安全性。

比特币系统结合了现代密码学、应用数学和计算机科学的最新成果，解决了在陌生人社会（即便存在恶意的欺骗者）达成共识的难题。信任问题的解决极大地降低了社会的交易成本，会带来一场深刻的社会变革。

比特币中使用了 PoW（proof of work，工作量证明）来保证比特币网络分布式记账的一致性。早在 1993 年，美国计算机科学家、哈佛大学教授辛西娅·德沃克（Cynthia Dwork）首次提出了工作量证明思想，主要用来解决垃圾邮件的问题。该机制要求邮件发送者必须计算出某个数学难题的答案，以此证明他确实执行了一定程度的计算工作，借此提高垃圾邮件的发送成本。1999 年，马库斯·雅各布松（Markus Jakobsson）正式提出了“proof of work”概念。中本聪将 PoW 算法引入到区块链中，巧妙地解决了共识难题。

> PoW 通过分布式节点的算力竞争，保证了数据的一致性和共识的安全性。比特币系统的各节点（即矿工）通过各自的计算机算力相互竞争，共同解决一个求解复杂、但是验证容易的 SHA256 数学难题，最快解决该难题的节点将获得下一区块的记账权和区块的奖励。
>
> 中本聪在给 James A. Donald 的信中，曾经用拜占庭将军暴力破解密码国王的 Wi-Fi 这样的例子来说明 POW 算法如何解决拜占庭问题。其实，这背后的思想是如此简单：如果我们发现一群人解决了一个数学难题，而要解决这个数据难题必须绝大多数人齐心协力，那么我们就知道绝大多数人已经就这一问题达成了共识。

总结

好了，总结一下。微服务系统作为分布式系统的特例，面临着一些核心困难的问题：即数据的一致性和可用性问题。随着业务量和数据量的膨胀，传统的关系型数据库的主从复制架构显示出解决这些问题的劣势，我们面临着网络分区、网络延迟、服务崩溃导致的数据丢失、数据过时等问题。

现代一些的新型分布式系统（例如 NoSQL 数据库 MongoDB）通常利用共识算法解决了上面的问题，具有更好的扩展能力和可用性。但 CAP 理论告诉我们，在分布式系统中通常需要在数据一致性与可用性之间做权衡。我们还知道了数据一致性有多种级别，最常见的为最终一致性、线性一致性、强一致性，其中线性一致性能够推导出强一致性，而强一致性能够保证不会读取到过时的数据。

有多种算法可以解决分布式系统可用性与数据一致性，其中比较有名的是 Paxos、Raft、Zab 算法。我们可以借助这些算法实现可靠的服务，并在此基础上实现诸多分布式协调的场景，例如选主、服务发现、分布式锁等。

最后我们还提到了无信网络中面临的拜占庭问题。由于攻击者的存在，达成共识异常困难。而比特币提供的 PoW 算法实在是划时代的解决方案。

课后题

最后，我也给你留一道思考题。

文中介绍的主从复制和 Raft 算法都有一个特殊的 Leader 节点，但现在有一些分布式系统的节点是完全平等的。以 Cassandra 为例，你知道它是如何保证一致性与可用性的吗？

欢迎你在留言区与我交流讨论，我们下节课再见！

上一篇 11丨微服务挑战：微服务治理体系与实践

下一篇 13丨智慧之火：详解分布式容错共识算法

精选留言 (3)

写留言

Realm 酥

2022-11-05 来自北京

Cassandra如何保证数据最终一致性：

1、逆熵机制(Anti-Entropy)

使用默克尔树(Merkle Tree)来确认多个副本数据一致，对于不一致数据，根据时间戳来获取最新数据。

2、读修复机制(Read Repair)

当Cassandra读数据时，需要根据读一致级别设定读取N个节点的副本数据，并按照时间戳返回最新数据给用户后，会对所有副本数据进行检测和修复，确保所有副本数据一致。

3、提示移交机制(Hinted Handoff)

当Cassandra写数据时，需要根据写一致性级别将数据写入到N个节点数据副本中，只有N个节点写入成功才会给用户返回操作成功，为防止要写入节点宕机导致操作失败，Cassandra采用提示移交机制将操作相关数据写入到随机节点，宕机节点恢复后可根据这些数据进行重放，最终获得数据一致性。

Gossip(闲话)协议会将宕机节点恢复的消息传递给其他节点，并及时进行数据修复。

提示移交机制产生的数据保存在系统表(system.hints)中，默认保存3小时。

4、分布式删除(Distributed Deletes)

由于Cassandra在多个节点上保存数据副本，如果直接对记录进行删除，在所有副本数据完全删除前，多个节点间数据不一致且无法按照时间戳判断该记录需要被修复还是被删除。Cassandra采用分布式删除机制，在删除记录时插入一条关于该记录的墓碑(tombstone),墓碑中包含接受客户端请求的存储节点执行请求的时间(Local delete time),通过墓碑来标识该记录已被删除。

Cassandra中压缩过程中实现垃圾回收机制，清理这些被墓碑标记的记录，以释放这些记录占用空间。

以上从网上查阅的资料，感觉对“时间戳”依赖很高，如何保障不同节点上事件的时间戳一定是准确的？

作者回复: 绝对的unix时间戳是无法保证准确的，但是逻辑时钟是可以保证准确的，再想一想呢

5

江楠大盗

2022-11-09 来自北京

老师说分区容忍度，指能够容忍任意数量的消息丢失。但是大部分说法是由于网络不稳定，可以容忍网络分区。这两种说法区别还是挺大的，希望老师能解惑一下，谢谢

2

那时刻

2022-11-18 来自北京

请问老师，文中提到，不可靠的时钟：这意味着我们无法依靠绝对的时钟来确定操作的顺序。如何来解决这个问题呢？使用相对时钟么？

1