14 错误侦测：如何保证分布式系统稳定？

经过上一讲的学习，相信你已经了解了分布式数据库领域中，分布式系统部分所重点解决的问题，即围绕失败模型来设计算法、解决各种稳定性问题。

解决问题的前提是发现问题，所以这一讲我们来说说如何发现系统内的错误，这是之后要介绍的算法们所依赖的前置条件。比如上一讲提到的共识算法，如果没有失败侦测手段，我们是无法解决拜占庭将军问题的，也就是会陷入 FLP 假说所描述的境地中，从而无法实现一个可用的共识算法。这里同时要指明，失败不仅仅是节点崩溃，而主要从其他节点看，该节点无法响应、延迟增大，从而降低系统整体的可用性。

这一讲，我将从影响侦测算法表现的几组特性出发，为评估这些算法给出可观标准；而后从你我耳熟能详的心跳算法开始介绍，逐步探讨几种其改良变种；最后介绍大型分布式数据库，特别是无主数据库常用的 Gossip 方案。

现在让我们从影响算法表现的因素开始说起。

影响算法的因素

失败可能发生在节点之间的连接，比如丢包或者延迟增大；也可能发生在节点进程本身，比如节点崩溃或者处理缓慢。我们其实很难区分节点到底是处理慢，还是完全无法处理请求。所以所有的侦测算法需要在这两个状态中平衡，比如发现节点无法响应后，一般会在特定的延迟时间后再去侦测，从而更准确地判断节点到底处于哪种状态。

基于以上原因，我们需要通过一系列的指标来衡量算法的特性。首先是**任何算法都需要遵守一组特性：活跃性与安全性，它们是算法的必要条件。**

- 活跃性指的是任何失败的消息都能被安全地处理，也就是如果一个节点失败了而无法响应正常的请求，它一定会被算法检测出来，而不会产生遗漏。
- 安全性则相反，算法不产生任何异常的消息，以至于使得正常的节点被判定为异常节点，从而将它标记为失败。也就是一个节点失败了，它是真正失败了，而不是如上文所述的只是暂时性的缓慢。

还有一个必要条件就是算法的完成性。完成性被表述为算法要在预计的时间内得到结果，也就是它最终会产生一个符合活跃性和安全性的检测结果，而不会无限制地停留在某个状态，从而得不到任何结果。这其实也是任何分布式算法需要实现的特性。

上面介绍的三个特性都是失败检测的必要条件。而下面我将介绍的这一对概念，可以根据使用场景的不同在它们之间进行取舍。

首先要介绍的就是算法执行效率，效率表现为算法能多快地获取失败检测的结果。其次就是准确性，它表示获取的结果的精确程度，这个精确程度就是上文所述的对于活跃性与安全性的实现程度。不精准的算法要么表现为不能将已经失败的节点检测出来，要么就是将并没有失败的节点标记为失败。

效率和准确被认为是不可兼得的，如果我们想提高算法的执行效率，那么必然会带来准确性的降低，反之亦然。故在设计失败侦测算法时，要对这两个特性进行权衡，针对不同的场景提出不同的取舍标准。

基于以上的标准，让我开始介绍最常用的失败检测算法——心跳检测法，及其多样的变种。

心跳检测法

心跳检测法使用非常广泛，最主要的原因是它非常简单且直观。我们可以直接将它理解为一个随身心率检测仪，一旦该仪器检测不到心跳，就会报警。

心跳检测有许多实现手段，这里我会介绍基于超时和不基于超时的检测法，以及为了提高检测精准度的间接检测法。

基于超时

基于超时的心跳检测法一般包括两种方法。

1. 发送一个 ping 包到远程节点，如果该节点可以在规定的时间内返回正确的响应，我们认为它就是在线节点；否则，就会将它标记为失败。
2. 一个节点向周围节点以一个固定的频率发送特定的数据包（称为心跳包），周围节点根据接收的频率判断该节点的健康状态。如果超出规定时间，未收到数据包，则认为该节点已经离线。

可以看到这两种方法虽然实现细节不同，但都包含了一个所谓“规定时间”的概念，那就是超时机制。我们现在以第一种模式来详细介绍这种算法，请看下面这张图片。

图 1 模拟两个连续心跳访问

上面的图模拟了两个连续心跳访问，节点 1 发送 ping 包，在规定的时间内节点 2 返回了 pong 包。从而节点 1 判断节点 2 是存活的。但在现实场景中经常会发生图 2 所示的情况。

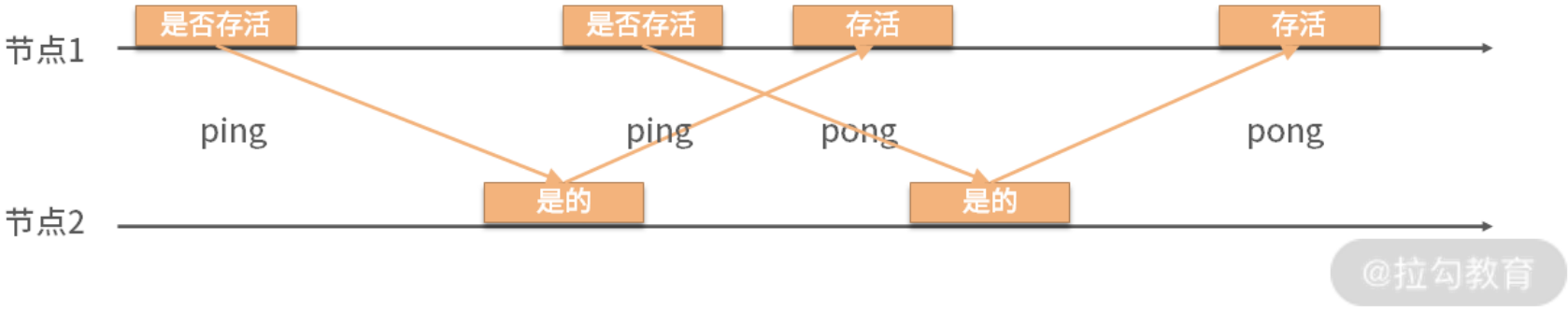

图 2 现实场景下的心跳访问

可以看到节点 1 发送 ping 后，节点没有在规定时间内返回 pong，此时节点 1 又发送了另外的 ping。此种情况表明，节点 2 存在延迟情况。偶尔的延迟在分布式场景中是极其常见的，故基于超时的心跳检测算法需要设置一个超时总数阈值。当超时次数超过该阈值后，才判断远程节点是离线状态，从而避免偶尔产生的延迟影响算法的准确性。

由上面的描述可知，基于超时的心跳检测法会为了调高算法的准确度，从而牺牲算法的效率。那有没有什么办法能改善算法的效率呢？下面我就要介绍一种不基于超时的心跳检测算法。

不基于超时

不基于超时的心跳检测算法是基于异步系统理论的。它保存一个全局节点的心跳列表，上面记录了每一个节点的心跳状态，从而可以直观地看到系统中节点的健康度。由此可知，该算法除了可以提高检测的效率外，还可以非常容易地获得所有节点的健康状态。那么这个全局列表是如何生成的呢？下图展示了该列表在节点之间的流转过程。

图 3 全局列表在节点之间的流转过程

由图可知，该算法需要生成一个节点间的主要路径，该路径就是数据流在节点间最常经过的一条路径，该路径同时要包含集群内的所有节点。如上图所示，这条路径就是从节点 1 经过节点 2，最后到达节点 3。

算法开始的时候，节点首先将自己记录到表格中，然后将表格发送给节点 2；节点 2 首先将表格中的节点 1 的计数器加 1，然后将自己记录在表格中，而后发送给节点 3；节点 3 如节点 2 一样，将其中的所有节点计数器加 1，再把自己记录进去。一旦节点 3 发现所有节点全部被记录了，就停止这个表格的传播。

在一个真实的环境中，节点不是如例子中那样是线性排布的，而很可能是一个节点会与许多节点连接。这个算法的一个优点是，即使两个节点连接偶尔不通，只要这个远程节点可以至少被一个节点访问，它就有机会被记录在列表中。

这个算法是不基于超时设计的，故可以很快获取集群内的失败节点。并可以知道节点的健康度是由哪些节点给出的判断。但是它同时存在需要压制异常计算节点的问题，这些异常记录的计数器会将一个正常的节点标记为异常，从而使算法的精准度下降。

那么有没有方法能提高对于单一节点的判断呢？现在我就来介绍一种间接的检测方法。

间接检测

间接检测法可以有效提高算法的稳定性。它是将整个网络进行分组，我们不需要知道网络中所有节点的健康度，而只需要在子网中选取部分节点，它们会告知其相邻节点的健康状态。

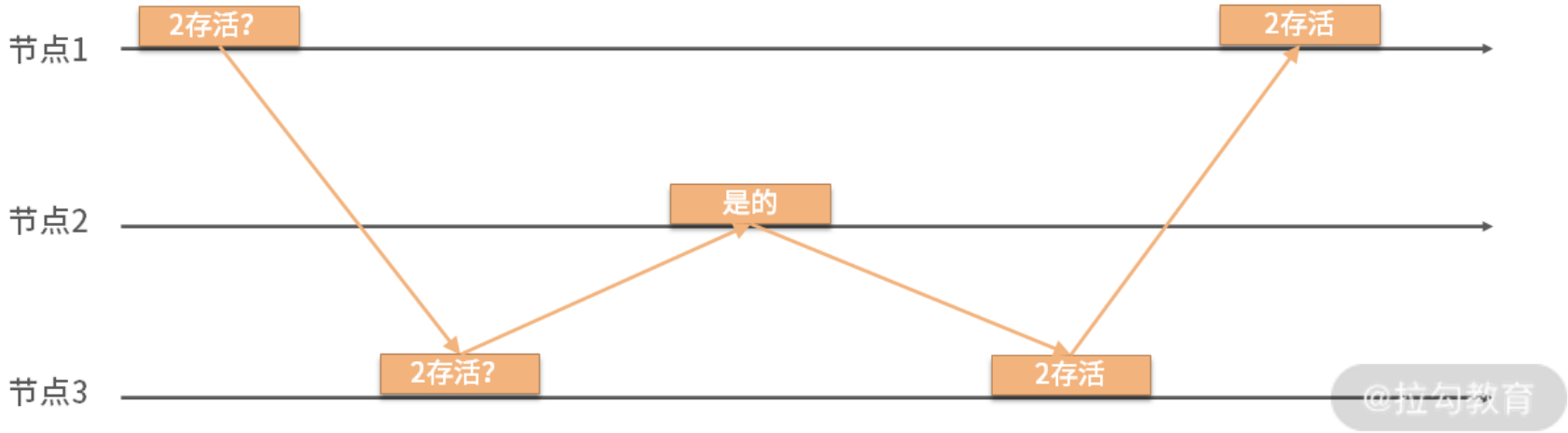

图 4 间接检测法

如图所示，节点 1 无法直接去判断节点 2 是否存活，这个时候它转而询问其相邻节点 3。由节点 3 去询问节点 2 的健康情况，最后将此信息由节点 3 返回给节点 1。

这种算法的好处是不需要将心跳检测进行广播，而是通过有限的网络连接，就可以检测到集群中各个分组内的健康情况，从而得知整个集群的健康情况。此种方法由于使用了组内的多个节点进行检测，其算法的准确度相比于一个节点去检测提高了很多。同时我们可以并行进行检测，算法的收敛速度也是很快的。因此可以说，**间接检测法在准确度和效率上取得了比较好的平衡。**

但是在大规模分布式数据库中，心跳检测法会面临效率上的挑战，那么何种算法比较好处理这种挑战呢？下面我要为你介绍 Gossip 协议检测法。

Gossip 协议检测

除了心跳检测外，在大型分布式数据库中一个比较常用的检测方案就是 Gossip 协议检测法。Gossip 的原理是每个节点都检测与它相邻的节点，从而可以非常迅速地发现系统内的异常节点。

算法的细节是每个节点都有一份全局节点列表，从中选择一些节点进行检测。如果成功就增加成功计数器，同时记录最近一次的检测时间；而后该节点把自己的检测列表的周期性同步给邻居节点，邻居节点获得这份列表后会与自己本地的列表进行合并；最终系统内所有节点都会知道整个系统的健康状态。

如果某些节点没有进行正确响应，那么它们就会被标记为失败，从而进行后续的处理。**这里注意，要设置合适的阈值来防止将正常的节点标记为错误。**Gossip 算法广泛应用在无主的分布式系统中，比较著名的 Cassandra 就是采用了这种检测手法。

我们会发现，这种检测方法吸收了上文提到的间接检测方法的一些优势。每个节点是否应该被认为失败，是由多个节点判断的结果推导出的，并不是由单一节点做出的判断，这大大提高了系统的稳定性。但是，此种检测方法会极大增加系统内消息数量，故选择合适的数据包成为优化该模式的关键。这个问题我会在“17 | 数据可靠传播：反熵理论如何帮助数据库可靠工作”中详细介绍 Gossip 协议时给出答案。

Cassandra 作为 Gossip 检测法的主要案例，它同时还使用了另外一种方式去评价节点是否失败，那就是 φ 值检测法。

φ 值检测

以上提到的大部分检测方法都是使用二元数值来表示检测的结果，也就是一个节点不是健康的就是失败了，非黑即白。而 φ 值检测法引入了一个变量，它是一个数值，用来评价节点失败的可能性。现在我们来看看这个数值是如何计算的。

首先，我们需要生成一个检测消息到达的时间窗口，这个窗口保存着最近到的检测消息的延迟情况。根据这个窗口内的数值，我们使用一定的算法来“预测”未来消息的延迟。当消息实际到达时，我们用真实值与预测值来计算这个 φ 值。

其次，给 φ 设置一个阈值，一旦它超过这个阈值，我们就可以将节点设置为失败。这种检测模式可以根据实际情况动态调整阈值，故可以动态优化检测方案。同时，如果配合 Gossip 检测法，可以保证窗口内的数据更加有代表性，而不会由于个别节点的异常而影响 φ 值的计算。故这种评估检测法与 Gossip 检测具有某种天然的联系。

从以上算法的细节出发，我们很容易设计出该算法所需的多个组件。

1. 延迟搜集器：搜集节点的延迟情况，用来构建延迟窗口。
2. 分析器：根据搜集数据计算 φ 值，并根据阈值判断节点是否失败。
3. 结果执行器：一旦节点被标记为失败，后续处理流程由结果执行器去触发。

你可以发现，这种检测模式将一个二元判断变为了一个连续值判断，也就是将一个开关变成了一个进度条。这种模式其实广泛应用在状态判断领域，比如 APM 领域中的 Apdex 指标，它也是将应用健康度抽象为一个评分，从而更细粒度地判断应用性能。我们看到，虽然这类算法有点复杂，但可以更加有效地判断系统的状态。

总结

这一讲内容比较简单、易理解，但是却非常重要且应用广泛。作为大部分分布式算法的基础，之后我要介绍的所有算法都包含今天所说的失败检测环节。

这一讲的算法都是在准确性与效率上直接进行平衡的。有些会使用点对点的心跳模式，有些会使用 Gossip 和消息广播模式，有些会使用单一的指标判断，而有些则使用估算的连续变换的数值……它们有各自的优缺点，但都是在以上两种特点之间去平衡的。当然简单性也被用作衡量算法实用程度的一个指标，这符合 UNIX 哲学，简单往往是应对复杂最佳的利器。

大部分分布式数据库都是主从模式，故一般由主节点进行失败检测，这样做的好处是能够有效控制集群内的消息数量，下一讲我会为你介绍如何在集群中选择领导节点。