介绍

dister(Distribution Cluster)是一款轻量级高性能的分布式集群管理软件，实现了分布式软件架构中的常用核心组件，包括：

1. 服务配置管理中心;
2. 服务注册与发现;
3. 服务健康检查;
4. 服务负载均衡;

dister的灵感来源于ZooKeeper、Consul、Etcd，它们都实现了类似的分布式组件，但是dister更加的轻量级、低成本、易维护、架构清晰、简单实用、性能高效，这也是dister设计的初衷。

特点

1. 开源、免费、跨平台；
2. 使用RAFT算法实现分布式一致性；
3. 使用通用的REST协议提供API操作；
4. 详尽的设计及使用说明文档，易于使用维护；
5. 超高读写性能，适合各种高并发的应用场景；
6. 使用分布式KV键值存储实现服务的配置管理；
7. 支持集群分组，不同的集群之间数据相互隔离；
8. 配置管理简单，简化的API接口以及终端管理命令；

安装

1. 建议下载预编译好的各平台版本使用，下载地址：https://gitee.com/johng/dists
2. (第三方依赖变化较大，目前暂无法编译，新版本开发完成后将会恢复)源码编译安装，需要gf框架的支持，框架地址：https://gitee.com/johng/gf

文档

官方网站：http://johng.cn/dister

相关文档：

1. dister的安装及使用
2. dister的使用示例
3. dister的性能测试

设计

1、网络拓扑

在去中心化、分布式集群的网络拓扑设计中，需要注意的是，同一集群以内的各节点之间需要保证连通性，
并且，尽可能地降低各节点之间的数据通信延迟(例如，尽可能保证集群各节点在同一内网环境中)，这是保证集群可靠性的前提。
这并不是dister的特点，所有去中心化、分布式集群网络拓扑设计都应当如此。

2、集群角色

dister集群有两种角色的概念，一种是**集群角色**，一种是**RAFT角色**。
集群角色主要用于区分集群节点的行使职能，RAFT角色主要用于RAFT算法处理数据一致性。
集群角色目前有两种：**client**和**server**。

1. **client**：可以看做一个独立进程的SDK，仅作API操作，不会参与RAFT的Leader选举、也不会存储任何的集群数据；
2. **server**：集群的关键节点，参与Leader选举，每个server节点保持数据同步与数据的强一致性，保证整个集群的高可用；

dister的SDK对于应用端是友好的、高性能的、非代码侵入式的，这使得dister对于业务系统的接入和使用变得更加的简便。
一个dister集群中至少应当有一个server节点，当集群中的leader节点出现故障后，其他的server就会按照既定的election算法快速执行选举，选举成功后新的leader将会接替故障的leader继续行使leader职能。

dister集群随时保持每个节点的数据同步，需要使用集群功能的应用端，本地需要运行一个client节点。

3、RAFT一致性

dister使用了流行的RAFT算法来保证一致性，并且对其中的某些部分进行了改进。

1、集群节点

1. RAFT需要至少3个及以上的节点构建集群，以保证稳定性；

2. dister支持1个及以上的节点构建集群，但是建议集群中包含2个及以上的server节点，以保证稳定性；

2、Leader选举

dister与RAFT一样，有三种角色：**follower**、**candidate**和**leader**。

1. 传统的RAFT有一套选举机制，在同一Term内一个节点只能投票给一个cadidate，并且保证n/2+1 个节点构成多数派选举出一个节点作为leader。缺点是，在实践中，这种选举过程其实非常缓慢，且集群的节点越多，耗费的时间也越多，极易出现split vote的情况。因此这种机制在灾难恢复的周期比较长，即在leader节点出现故障之后，重新完成选举并恢复集群正常运行相对来说耗费的时间会比较长，可靠性相对来说要差一些。

2. dister按照 投票+比分 两个机制来进行leader选举，允许一个节点投票给多个candidate，并且保证至少 n/2+1 个节点票数作为选举前提，经过一轮节点之间的对比，谁获得的投票多，并且耗费的时间最短(通过请求投票的响应时间差计算比分，响应越快比分越高)，那么该节点即可被选举为leader，其他节点在对比中被淘汰为follower。这样的选举非常快，它只需要经过一轮各节点的对比便能选举出leader，不会出现spit vote的情况，并且选举出的leader往往是集群中网络吞吐比较快速的节点。集群故障恢复也会更加迅速，可靠性相对RAFT的机制要好。

当然，在dister选举中，影响选举的因素除了选票和比分外，还包括数据版本，即整个dister集群中数据集最新的节点优先成为leader。此外，dister的选举和RAFT的传统选举都有先发优势这一说，也就是首先发起选举的节点成为leader的概率越高。

3、数据同步

dister针对数据同步的思想与RAFT是一致的。

改进的地方主要在于dister的集群角色处理上（因为RAFT算法并未有集群角色这层概念）。

1. 传统的RAFT数据同步中，一条数据的请求将会阻塞地通过UncommittedLogEntry + AppendLogEntry两次请求同步到其他节点，并在保证 n/2+1 个节点返回成功之后，这条数据请求才会被判定为成功，否则即为失败，随后服务端才会返回结果给调用端。可见，在这种机制下，RAFT充分保证了数据的强一致性，但是集群的并发写入性能也受到了很大的影响，特别是随着集群节点的增多，RAFT集群的写入性能将会呈曲线下降。因此，dister对RAFT数据同步机制改进的目的，是对同步并发性能与数据强一致性之间取了一个折中的方案。

2. 在数据的请求处理中，dister将RAFT的UncommittedLogEntry + AppendLogEntry两次请求合并为一次请求，即直接将请求的数据并行发往其他的server节点执行写入(这个过程也是阻塞的)。dister保证leader和至少另外一个server节点处理成功，那么才能判断为成功，反之即为失败，失败情况下应用端可以选择重试，也可以选择放弃。dister的这种处理机制在保证数据一致性的同时也提供了良好的写入性能。并且，即使在处理过程中leader挂掉，只要有另外一个server节点有最新的请求数据，那么就算重新进行选举，也会选举到数据版本最新的server节点作为leader，这里的机制类似于主从备份的原理（当然不仅仅是备份概念，dister

保证了数据的强一致性）。此外，由于采用了异步并发请求的机制，如果集群存在多个其他server节点，出现仅有一个节点成功的概率很小，出现所有节点都失败的概率更小。

3. 有一点需要指出的是，在有且仅有一个节点的dister集群中(当然该节点为server节点，并且毫无疑问地被选举为leader，dister支持这种单server节点的集群，类似于zookeeper的standalone模式，但不推荐)，一个写入请求仅保证leader成功即为成功，因此，读写性能会更加高效，但是容易产生单点故障(这也是不推荐的原因)。

4. 在数据同步的比较方面，同一集群采用的是增量同步，如果出现两个不同集群(数据完全不一致)的节点组合成一个新的集群(例如，将另一个集群的server节点在未清除数据的情况下加入到另外一个集群)，那么dister不会合并，只会按照数据的新旧程度一个将另一个节点数据进行覆盖。

5. client节点不做数据同步，不存储任何的集群数据，任何的写入请求都是直接转发给leader，查询请求都是转发给server节点(负载均衡算法详见后续章节)，但是，对于查询请求，本地节点会做一定的缓存处理，降低对leader节点的请求压力；

4、分区及脑裂问题

1. **RAFT**：RAFT paper中对于分区问题的处理办法并未做特别详细的说明，分区造成的难点主要是分区后的leader选举问题以及分区恢复后的数据同步问题。例如，分区后的多数派按照RAFT协议仍然能够选举成功leader，假如分区前的leader在分区后位于少数派，这时在集群中会出现双leader情况，在分区后会引起脑裂问题，分区恢复后会引起数据同步问题。

2. **dister**：dister对于以上问题的解决方案如下，

    - leader选举问题：集群一旦产生分区，那么少数派不会重新选举成功(无法满足 n/2+1 的条件)，也不会形成新的集群，即使分区前的leader在分区后位于少数派中，在分区后也会被迅速降级为follower。而分区后的多数派，则会成功选举形成为一个新的集群(如果分区后leader位于多数派，那么不会引起新的选举)。任何到达少数派的请求都会失败，到达多数派的会成功，只有分区恢复或者人为干预，集群状态才能够恢复。对于极端情况下，例如一个集群被划分为多个分区(架构师的脑壳抖了若干下)，每个分区都构不成多数派，那么到达该集群的所有请求都会失败。

    - 数据同步问题：dister对于leader选举问题进行完美解决后，便不会在一个集群中出现多个leader的情况(即使在分区前或者后)，那么也不会出现脑裂问题。但是仍然有可能会出现数据同步问题，例如分区前leader的数据未完全同步到其他节点，分区过后该leader节点位于少数派中，多数派会形成新的leader，分区恢复后，老的leader数据将会被新的leader数据**简单粗暴地覆盖掉**，造成这部分未同步数据的丢失。在dister集群中，不会出现follower向leader之间同步数据的可能，dister会认为，leader永远是正确的。

5、负载均衡设计

1. **数据负载均衡**：上面我们提到，在dister集群中，server节点负责数据的存储，client节点仅作为一个进程SDK使用。任何的数据写入/删除都是通过client写入到leader，并由leader按照RAFT算法保证集群的数据一致性。但是，数据在读取的时候便不能请求到leader(负载均衡考虑)，而是通过一致性哈希算法请求到其他的server节点，保证整个集群数据操作的负载均衡。当然，由于leader与其他节点之间的数据同步是有延迟的，因此，这里需要特别说明的是，数据在写入leader成功后，会在client节点进行数据缓存(在特定大小的内存区域，按照LRU算法进行数据缓存及淘汰)，保证对于当前应用端来讲，数据读写是即时的。

2. **服务负载均衡**：dister提供了独立的负载均衡接口来实现服务的负载均衡访问(负载均衡没有采用一致性哈希算法，而是通过服务的priority属性计算，灵活度更高)，由于应用端与dister的集群通信采用的是本地进程间通信实现，因此，这种机制会比远程RPC的执行效率更加高效。dister并没有提供服务节点DNS的负载均衡方式，使用DNS其实也是一种对于接口API的一层封装，对于使用者来说，他需要配置本地网络的DNS才能使用。然而这有点违背dister设计的初衷，就是dister希望尽可能的简单实用。因此，dister仅提供了通用的REST接口这么一种方式来访问所有的API。此外，使用者可以选择性地对负载均衡结果进行缓存，减少请求数。

6、REST接口设计

为保证接口的通用性，dister的接口采用了REST设计方式，这是和开发语言无关的，也无需提供SDK，只需要向本地的接口地址发送HTTP请求即可。dister的接口仅对本地开放，即接口地址都是监听的 127.0.0.1，因此，一个新加入的应用端想要使用dister集群功能，在应用端本地需要运行dister的节点。dister提供的接口模块目前有4种：集群节点管理、KV数据管理、服务管理、负载均衡管理。每个模块的API接口说明可查看dister的使用说明章节。

计划

v2.00

1. 重新梳理RAFT实现，查看有无进一步的改进空间；
2. 改进binlog设计，全新高可用的binlog文件结构及实现；
3. 使用KV数据存储设计，使用独立的嵌入式KV数据库进行存储；
4. 改进数据同步逻辑，保证节点在数据同步的高可用；
5. 重新梳理、简化、改进服务健康检查业务逻辑；
6. 严格、仔细的功能及性能测试；

v2.10

1. 增加服务的API控制功能*(服务注册与发现、服务健康检查)*；

v2.50

```
1. 增加Socket接口支持；
```

贡献

dister是开源的、免费的软件，这意味着任何人都可以为其开发和进步贡献力量。
dister的项目源代码目前同时托管在 Gitee 和 Github 平台上，您可以选择您喜欢的平台来 fork 项目和合并你的贡献，两个平台的仓库将会保持即时的同步。
我们非常欢迎有更多的朋友加入到dister的开发中来，您为dister所做出的任何贡献都将会被记录到dister的史册中。