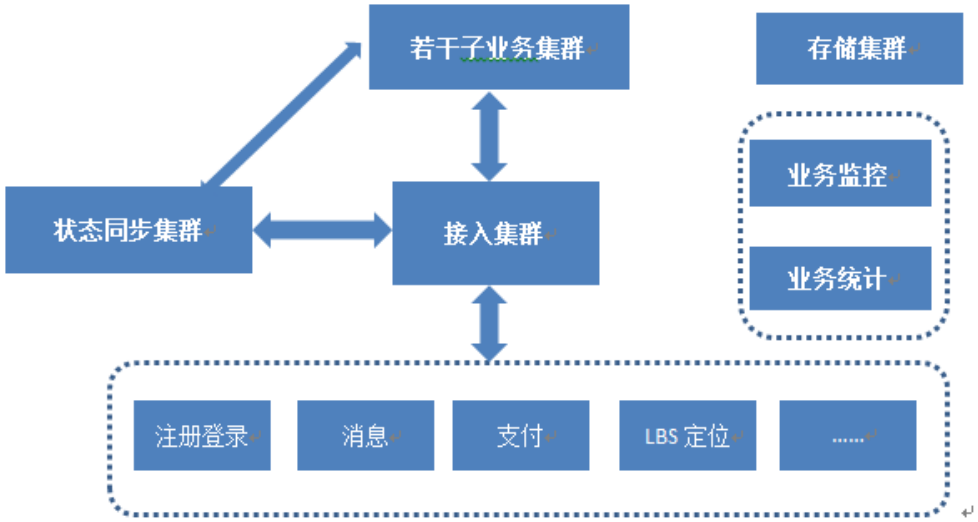
软件复用讨论课 2

1352921 廖山河

1. 分布式

分布式系统是为了解决单个计算机性能不足的问题而将软件分为不同的子系统并部署到多个计算机上的一种架构方式。说到分布式，现在我知道相关的如 Hadoop 用于进行分布式存储及计算，如阿里的 DUBBO 分布式服务框架可以提供高性能 RPC 服务，不过我想就软件复用的项目出发进行讨论，所以主要参考了微信等聊天软件的实现。

微信的系统架构图如下：



应用层提供各种服务业务，由接入集群接入服务器集群，中间有一个状态同步集群用于同步不同的服务集群和接入集群，使它们能够达到一致性的存储要求。另外还有诸如存储集群用来存储用户信息和应用信息，业务监控和业务统计等辅助功能模块。这种设计可以在保证服务质量的同时控制成本。

微信使用的 NoSQL 存储名称为 Quorum_KV，是基于 LSMTree 研发的强一致性、持久化 KV 分布式存储，支持 key-table 和 string-value 两种存储模型。Quorum_KV 通过 Quorum 协议实现双向可写功能，一个最小的存储单元由二台存储机和一台仲裁机组成，写存储机时经过仲裁决定被写的机器。写数据时直接写内存的 Memtable 表。Memtable 写满后转换成 Immutable。Immutable 定期 Dump 到本地磁盘变成数据文件，数据文件不断递增形成不同 level 级别数据文件，不同 level 级别的数据文件会定期合并。内存表使用 Skiplist 来做内存的 Key 索引。读数据时先读内存表，如果找不到记录再寻找 Immutable，然后是 level 0 磁盘文件、level 1……直至寻找到记录返回结果。所以可以看到，Quorum_KV 的设计是面向写量大，读量小的业务场景，很好地适应了微信消息写量大的特点。

在可用性方面，可参考下图：

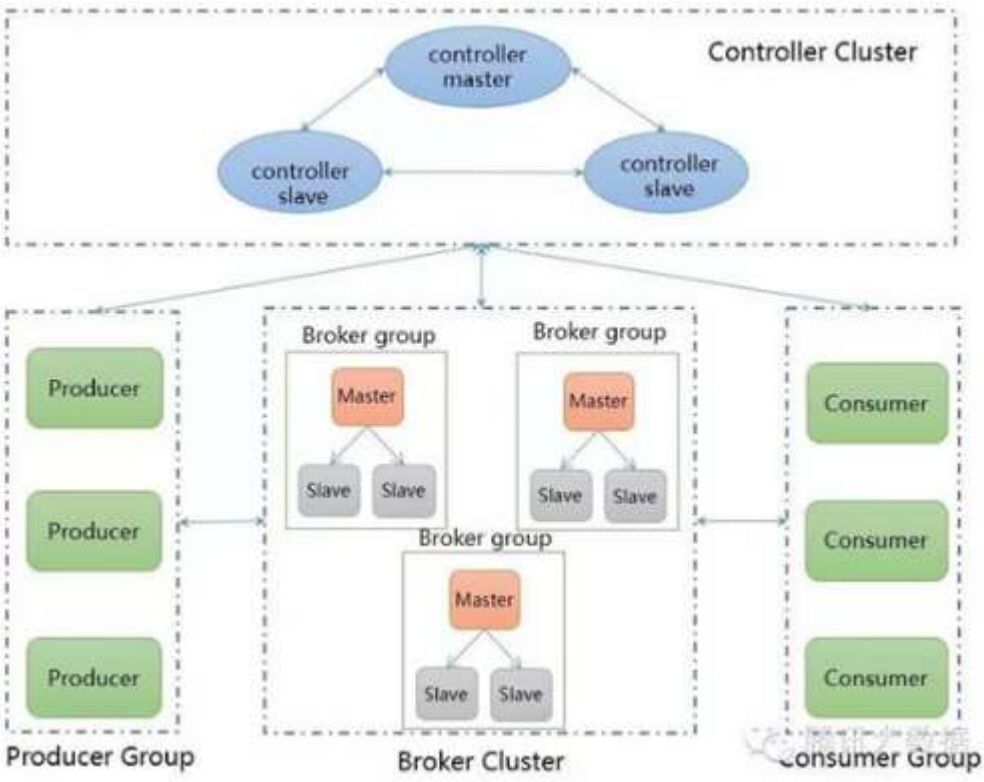


图3 系统逻辑交互图

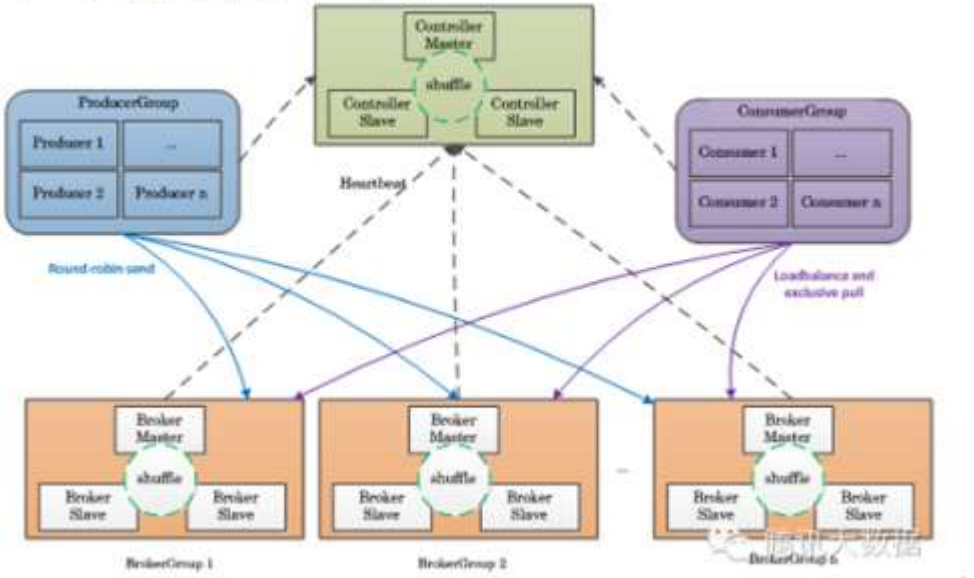


图4 系统交互图

对于 broker 组只要每个组内依旧有超过半数的节点存活，那么这个 broker 组便能继续对外提供服务。如果 SlaveBroker 宕掉不会对生产消费端有任何影响，如果是 MasterBroker 宕机那么会自动的从两个 slave 中选出一个新的 master。对于宕掉的机器通过监控手段发现后人工重启便会自动的同步宕机过程中滞后于同组节点的数据，直到追上最新数据为止。

对于 controller 与 broker 的情况类似，但是即使 controller 整个组都同时宕机也不会对当前的系统造成不可用的情况，当前系统将会继续维持目前的平衡状态，任何的

broker 组变更，consumer 变更都不会再次触发系统的均衡，直到 controller 恢复为止。

2. 数据一致性

微信设计了新一代消息系统 Hippo 以满足具有高可靠高可用应用场景的业务需求。

Hippo 系统[5]存在四种角色，分别为生产者（producer）、消费者（consumer）、存储层（broker）、中心控制节点（controller）

Controller:以组的形势存在，三台 controller 一主两备组成一个组（主备 controller 存在心跳检测以便在主故障的时候能够自动 failover）承担着整个系统节点数据的收集、状态的共享及事件的分发角色。提供控制台界面，根据当前收集到的正常运行的 broker 节点信息，可以指定给某个特定的 broker 组下发 topic 及 queue 添加事件。

Broker：以组的形势存在，三台 broker 一主两备组成一个组，由主 broker 向 controller 定期汇报心跳以告知 controller 当前组的存活状态，心跳携带当前组所管理的 topic 及 queue 信息。数据在 broker 以多副本的方式存储，Masterbroker 为数据写入入口，并把数据实时同步给同组的两台 Slavebroker，主备 broker 之间存在心跳检测功能，一旦 Slavebroker 发现 Masterbroker 故障或者收不到 Masterbroker 的心跳那么两台 Slavebroker 之间会从新发起一次选举以产生新的 Masterbroker，这个过程完全不用人工介入系统自动切换。因此在 broker 端不存在单点情况，数据冗余存储在不同的物理机器中，即使存在机器宕机或磁盘损坏的情况也不影响系统可靠对外提供服务。

Producer：轮询发送：向 controller 发布某个 topic 的信息，controller 返回相应 topic 所在的所有 broker 组对应的 IP 端口及 queue 信息。producer 轮询所获取的 broker 组信息列表发送消息并保持与 controller 的心跳，以便在 broker 组存在变更时，能够通过 controller 及时获取到最新的 broker 组信息。

Consumer：负载均衡：每个 consumer 都隶属于一个消费组，向 controller 订阅某个 topic 的消息，controller 除了返回相应 topic 对应的所有 broker 组信息列表之外还会返回与当前消费者处于同一个组的其它消费者信息列表，当前消费者获取到这两部分信息之后会进行排序然后按照固定的算法进行负载均衡以确定每个消费者具体消费哪个队列分区。同时每个 consumer 都会定期的向 controller 上报心跳，一旦消费组有节点数量的变更或 broker 组存在变更，controller 都会及时的通过心跳响应返回给当前组所有存活的 consumer 节点以进行新一轮的负载均衡。

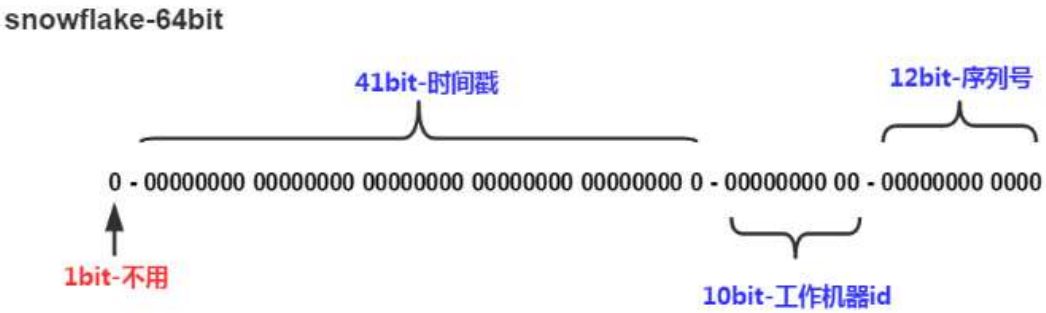
消费确认：consumer 进行消费的过程中对队列分区是以独占的形式存在的，即一个队列在一个消费组中只能被一个消费者占有并消费。为了保证消费的可靠对于每次拉取的数据，都需要 consumer 端在消费完成之后进行一次确认，否则下次拉取还是从原来的偏移量开始。

限时锁定：为了使某个 consumer 宕机其占有的队列分区能够顺利的释放并被其他 consumer 获取到，需要在每个消费者拉取数据与确认回调之间设置一个超时时间，一旦超过这个时间还没确认，那么队列自动解锁，解锁之后的队列最终能够被别的存活消费者占有并消费。

3. ID 分配

现在应用很多的一个算法：Twitter 的 Snowflake，64 位自增 ID 算法。其算法核心是把

时间戳，工作机器 id，序列号组合在一起。如下图所示：



除了最高位 bit 标记为不可用以外，其余三组 bit 占位均可浮动，看具体的业务需求而定。默认情况下 41bit 的时间戳可以支持该算法使用到 2082 年，10bit 的工作机器 id 可以支持 1023 台机器，序列号支持 1 毫秒产生 4095 个自增序列 id。具体的实现可见 https://github.com/twitter/snowflake/releases/tag/snowflake-2010。

4. 垃圾消息过滤

使用数据挖掘及机器学习来实现垃圾消息过滤。如可以通过贝叶斯算法来实现垃圾邮件的过滤。

预先提供两组已经识别好的邮件，一组是正常邮件，另一组是垃圾邮件。用这两组邮件，对过滤器进行"训练"。这两组邮件的规模越大，训练效果就越好。"训练"过程很简单。首先，解析所有邮件，提取每一个词。然后，计算每个词语在正常邮件和垃圾邮件中的出现频率。比如，我们假定"sex"这个词，在 4000 封垃圾邮件中，有 200 封包含这个词，那么它的出现频率就是 5%；而在 4000 封正常邮件中，只有 2 封包含这个词，那么出现频率就是 0.05%。（【注释】如果某个词只出现在垃圾邮件中，Paul Graham 就假定，它在正常邮件的出现频率是 1%，反之亦然。这样做是为了避免概率为 0。随着邮件数量的增加，计算结果会自动调整。）

有了这个初步的统计结果，过滤器就可以投入使用了。然后在使用中根据反馈对模型进行调整。

参考：

1. http://m.blog.csdn.net/article/details?id=51206100
2. http://www.lanindex.com/twitter-snowflake%EF%BC%8C64%E4%BD%8D%E8%87%AA%E5%A2%9Eid%E7%AE%97%E6%B3%95%E8%AF%A6%E8%A7%A3/
3. http://www.ruanyifeng.com/blog/2011/08/bayesian_inference_part_two.html