1. 程序结构简介

缓存程序(下文中简称cache)，在整个缓存系统中承担着数据的最终存储的任务。在cache内部的缓存通过最近最少使用（LRU）算法进行管理。它能够接受来自客户端(client)的请求，读出键值或者改变键值，也能够向控制程序(master)通过传送心跳包的方法向其报告自身状况，并根据master的指令传递向其他cache传递扩/缩容信息。

另外，为了保证整个缓存系统的灵活性和可靠性，在cache程序内分别实现了系统的扩容和缩容功能，和缓存的备份和容灾功能。为了实现这些功能，cache程序由数个相互独立的模块构成，分别是LRU缓存模块、一致性哈希模块、cache-cache通信模块、cache-client通信模块和master-client通信模块。缓存程序的结构如图2-1所示。



图 2-1 缓存程序的结构示意图

2. 底层算法和数据结构

2.1 LRU缓存算法

程序底层的缓存算法的更新采用最近最少使用（Least Recently Used, LRU）算法进行维护。该算法的原理是，将一段时间内最近最少被使用的数据剔除出缓存中。程序中采用了一个双向链表和一个哈希表实现了该算法。

其中，双向链表按照被使用的顺序储存了这些键值对，靠近头部的键值对是最近使用的，靠近尾部的键值对则是最久未被使用的。哈希表采用了普通的哈希映射，通过缓存数据的键映射到其在双线链表中的位置。这样，就可以使用哈希表进行定位，找到缓存项在双向链表中的位置，随后将之移动到双向链表的头部。

LRU模块支持两个最基本的操作，分别是查询(get)操作和写入(put)操作。这两种操作的具体方法如下：

1）查询（get）操作，首先判key是否存在。如果key不存在，则返回空值；如果key存在，那么key对应的节点则是最近被使用的节点。通过哈希表定位到该节点在双向链表中的位置，并将其移动到双向链表的头部，最后返回该节点的值。

2）写入（put）操作，首先判断key是否存在。如果key不存在，则使用key和value创建一个新的节点，在双向链表的头部添加该节点，并将key和该节点添加到哈希表中。然后判断双向链表的节点数是否超出容量，如果超出容量，那么删除双向链表中的尾部节点，并删除哈希表中对应项目。

如果key存在，那么与get操作类似，先通过哈希表定位，再将对应的节点值更新为value，并将该节点移动到双向链表的头部。

2.2 一致性哈希算法

在Master收到Client端发送的读写请求之后，为了保证各个Cache的负载均衡以及保证缓存能够有效命中存有缓存数据的Cache，需要利用一致性哈希算法来计算Client端发送的Key和IP的映射关系。一致性哈希算法是通过将每一个IP地址通过哈希算法映射成均匀的哈希值之后，将其均匀的分布在哈希环上面。

当我们收到用户请求的Key之后，通过哈希算法将其映射到环上，然后顺时针找到距离其最近的服务节点作为该Key将要与其通信的节点。所以，利用一致性哈希算法，当某个服务节点挂机之后，仅仅影响到距离他顺时针的最近的节点的负载，而不至于全体的Cache为此付出代价,有效的防止了缓存雪崩的问题。但是当Cache的节点比较少的时候，由于用户的Key无法均匀地铺满整个哈希环，所以会出现数据倾斜地问题，也就是部分节点的负载压力较高。

针对数据倾斜问题，可以采用虚拟节点到真实节点映射的方法将虚拟节点均匀地布满哈希环，这样用户传来的请求通过虚拟节点映射到真实节点之后，可以有效地将负载均衡。

一致性哈希算法的模式图如图2-2所示。

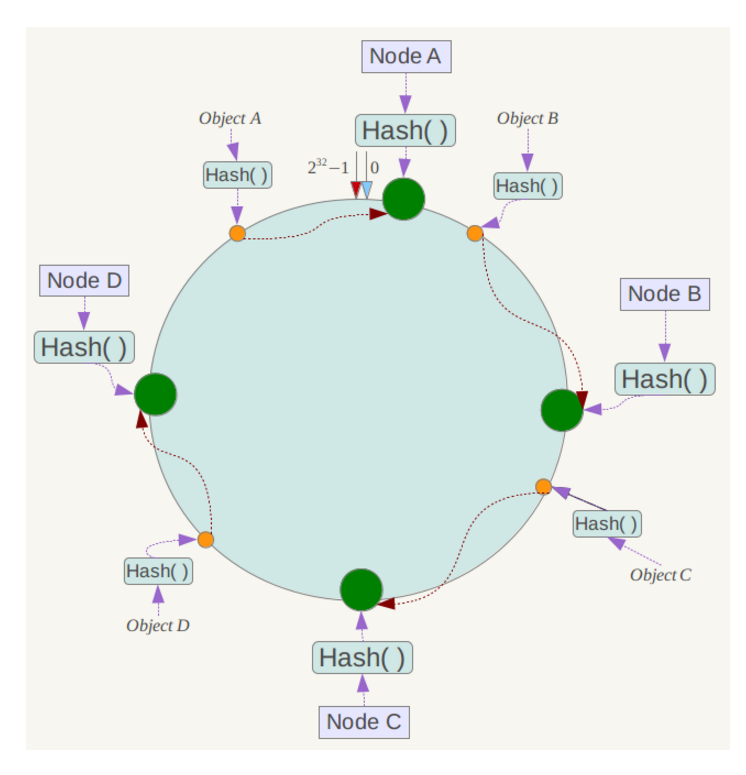


图2-2 一致性哈希算法示意图

2.3 地址表

在每Cache成相互中，为了能够在整个系统发生扩缩容行为时，被关停的缓存节点能够及时地根据一致性哈希算法计算每个键值对应当传向的节点地址，在每个缓存程序内设置了一个地址表。

该地址表的底层是一个vector<pair<string,string>> ,并支持查找、修改、删除、新增指定地址等方法。在每一次整个缓存系统中的缓存发生变动时，例如系统由控制端下达扩缩容命令，或是缓存节点发生意外故障时，每个缓存节点都将会根据控制端广播的指令更新地址表。具体的地址表更新规则如下：

1）系统扩容时，master广播新增节点的地址，所有缓存节点在地址表中新增该地址；

2）系统缩容时，master广播关停节点的地址，所有缓存节点在地址表中删除该地址；

3）缓存节点意外故障，master判断该节点意外关停时，广播故障节点及其备份节点的地址。所有缓存节点在地址表中将故障节点的地址替换为备份节点。如果该故障节点没有备份节点，则其他所有缓存节点在地址表中将删除该节点的地址。

4）对于新加入的缓存节点，master将发送现有所有缓存节点的地址，该节点将会将这些地址写入自身内部的顶底指标，让缓存节点自身完成初始化。

3. 程序功能

3.1 程序功能简介

为了实现缓存系统的键值管理、键值查询功能，并且为了保证缓存系统的可靠性和灵活性，在缓存程序中实现了键值查询功能、扩容和缩容功能、容灾功能、心跳包发送功能。其中，键值查询功能用来处理来自客户端的请求，并根据客户端的请求更改缓存或查询缓存中的值。扩容和缩容功能用来根据控制端的请求，进行缓存的扩容和缩容中涉及到的数据传输工作。容灾功能实现了主缓存模块和备份模块之间的同步和通信功能。一旦主缓存意外失效，备份缓存将第一时间顶替主缓存的位置。

3.2 键值查询功能

程序的键值查询功能是由LRU模块提供的。对于LRU模块而言，它提供了两个方法，分别是put(key, val),get(key),分别对应着更改值和查询值的功能。当客户端发出请求后，会通过报文的形式通知给缓存端，缓存端则通过IO复用的方法将这些不同的客户端所发出的请求打包成一个一个任务，并传到线程池中。

Client在master上请求key并获取其对应cache的地址成功，便会以读/写模式请求访问目标cache，cache会解析并识别其请求内容，并处理client需求。cache如何与多个client建立并保持通信等主要涉及通信部分请详见客户端通信模块。处理client需求则为本小节内容。

程序的键值查询功能是由LRU算法提供的。对于LRU算法而言，它提供了两个方法，分别是put(key, val),get(key),分别对应着更改值和查询值的功能。当客户端发出请求后，会通过报文的形式通知给缓存端，缓存端则通过IO复用的方法将这些不同的客户端所发出的请求打包成一个一个任务，并传到线程池中。

具体实现则是“LRU\_handle\_task”函数先通过分割协议报文判断是读还是写，若带有'#'则为写，若无则为读，后通过put(key, val),get(key)两种方法实现读和写，若键值已存在缓存链表中，则为成功读写；若键值不存在且为写入情况，则将键值写入链表中，否则返回FAILED，表示读取失败，返回格式为“SUCCESS/FAILED#key#ip:port”，如下表所示：

|  |  |  |
| --- | --- | --- |
| SUCCESS/FAILED | key | ip:port |
| 请求状态，成功为SUCCESS，失败为FAILED | 请求键 | 请求Cache的ip与端口 |

表2-1 cache-client键值请求回复格式

3.3 扩容和缩容功能

缓存中的扩容和缩容功能主要是依靠缓存间的通信模块和一致性哈希模块实现的。当控制端发出扩容命令时，只需要将新增的缓存节点加入自身的地址表中即可，反之，控制端发出缩容命令时，则需要指定一个节点进行数据转移和关停。此时，则需要分配该节点所掌握的所有键值对。

在程序中，采用了一致性哈希算法实现了键值对的分配。在节点收到关停命令时，节点将会利用一致性哈希算法，根据自身的地址表，为每个键值对分配一个地址，并将该键值对发送到对应的缓存中去。此时，节点完成了数据的转移，并顺利关停。

对于该节点的备份节点而言，当控制端向主节点发送关停命令时，也会向所有缓存节点广播将亡节点的地址。此时，备份节点将会将该地址与自己的主节点的地址进行比较，如果二者一致，则意味着该备份节点也将会被关停。执行扩、缩容命令时的时序图如图2-4所示。



（a）执行扩容命令



（b）执行缩容命令

图2-3执行扩、缩容时的通信时序图

3.4 心跳功能

在长连接下，有可能很长一段时间都没有数据往来，在这个时候，就需要心跳包来维持长连接，保活。每个cache会隔一段时间发送一个心跳包给Master，Master收到后回复接受成功与否的信息，如果Master几分钟内没有收到cache信息则视cache断开。cache会定时向master上传心跳包，格式为

“x#local\_IP#local\_port\_for\_client#local\_port\_for\_cache#P/R”，用#分割信息，如下表所示

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| x | Local\_IP | local\_port\_for\_client | local\_port\_  for\_cache | P/R |
| 心跳包识别符 | 本地IP | 为client通信准备的端口 | 为其它cache通信准备的端口 | 主从关系 |

表2-2 cache-master心跳包通信格式

master收到会回复一个通信包。“P/R#None”表示配对失败，“P/R#r/pcache\_IP#port\_for\_cache”表示配对成功并通知其对应的主/备份IP与端口，以便主从cache间的备份。

对于心跳包的定时发送，采用了定时器Timer类，其功能主要为基于链表和信号实现了单进程下的多定时器，最小定时间隔1微秒，同时可指定单次定时或循环定时，通过void\*结构体为定时器回调函数传入参数。

部分成员如下：

文本

描述已自动生成

使用sigaction绑定SIGALRM信号处理函数，使用setitimer以一定的时间间隔发送SIGALRM信号，在信号处理函数中遍历定时器链表，更新并检查每个定时器的信息，判断是否达到定时时间并执行每个定时器单独的回调函数。

在Heartbeat()中定义静态定时器智能指针，指向一个定时器对象，设置定时器超时回调函数。该回调函数向Master上报心跳信息并接收Master返回。

3.5 容灾功能

为了提高缓存系统的可靠性，程序为每个主节点设置了一个备份节点。主节点和备份节点一同产生，且一同受控制端关停。当主节点接收到客户端的请求后，主节点也会将该请求转发给备份节点，使二者的缓存是同步的。当主节点意外关停，控制端无法接收到来自主节点的心跳时，控制端则会通知备份节点转为主节点，并将该节点的地址广播给其他的缓存，使得其他缓存更新自己的地址表。容灾功能的模式图如图所示：



（a） 正常情况下



（b） 主节点意外关停时

4. 程序模块简介

4.1 线程池模块

当client发出请求后，会通过报文的形式通知给cache，cache则通过IO复用的方法将这些不同的client所发出的请求打包成一个一个任务，并传到线程池中。

线程池的工作原理如图2-3所示。在线程池中，有数个事先启动并等待任务执行的工作线程。任务在被加入到线程池的任务队列后，会被一一派发给这些工作线程。在工作线程的启动和运行方面，采用了future-promise机制实现了线程的调用和同步。这些线程在完成任务后，会被统一将任务结果发送给future队列中，再由主线程统一应答。



图2-4线程池的原理示意图

4.2 客户端通信模块

服务端通信主要是通过client\_socket(std::string server\_IP, std::string server\_port)来实现，该函数将服务器地址和端口与sockfd绑定在一起，使之成为方便网络通信的文件描述符，后发送给服务端。Cache作为客户端运行一般出现在以下情况：作为客户端发送心跳包给master；扩缩容时作为键值发送方；容灾时作为键值发送方。

4.3 服务端通信模块

服务端通信主要是通过server\_socket(std::string server\_IP, std::string server\_port)来实现，原理与客户端一样，多考虑了监听连接的情况。

Cache作为服务端运行一般出现在以下情况：client请求读写时作为服务端；扩缩容时作为数据接受方；容灾时作为数据接受方。因为考虑到存在多个客户端与单个服务端通信的情况，引入线程池，则可以解决了高并发问题，服务端可以按照队列顺序处理任务。