**简单分布式缓存系统**

**程序开发报告**

**第二组**

**2021. 12.17**

目 录

[**1. 概述** 5](#_Toc90614205)

[**2. 客户端（Client）程序实现** 5](#_Toc90614206)

[**2.1 定时器Timer类** 5](#_Toc90614207)

[**2.1.1 功能** 5](#_Toc90614208)

[**2.1.2 部分成员** 6](#_Toc90614210)

[**2.1.3 实现方法** 6](#_Toc90614213)

[**2.2 Client设计** 6](#_Toc90614215)

[**2.2.1 本地缓存** 6](#_Toc90614216)

[**2.2.2 工作流程** 6](#_Toc90614218)

[**2.2.3 异常处理** 7](#_Toc90614223)

[**3. 控制端（Master）程序实现** 8](#_Toc90614238)

[**3. 1程序结构简介** 9](#_Toc90614239)

[**3. 2 底层算法和数据结构** 9](#_Toc90614240)

[**3.2.1 一致性哈希算法** 9](#_Toc90614241)

[**3.2.2 cache映射** 9](#_Toc90614242)

[**3.2.3 存活检测** 10](#_Toc90614243)

[**3.3 程序功能介绍** 10](#_Toc90614244)

[**3.3.1 client查询cache地址功能** 10](#_Toc90614245)

[**3.3.2 主备份分配功能** 10](#_Toc90614246)

[**3.3.3 心跳功能** 11](#_Toc90614247)

[**3.3.4 cache容灾功能** 11](#_Toc90614248)

[**3.3.5 扩缩容功能** 12](#_Toc90614249)

[**3.4程序模块简介** 12](#_Toc90614250)

[**3.4.1 Master-client通信模块** 12](#_Toc90614251)

[**3.4.2 Master-cache通信模块** 12](#_Toc90614252)

[**3.4.3 周期性心跳检测模块** 13](#_Toc90614253)

[**3.4.4 缩容模块** 13](#_Toc90614254)

[**4. 缓存端（Cache）程序实现** 13](#_Toc90614255)

[**4.1. 程序结构简介** 13](#_Toc90614256)

[**4. 2. 底层算法和数据结构** 14](#_Toc90614257)

[**4.2.3 地址表** 16](#_Toc90614258)

[**4.3. 程序功能** 17](#_Toc90614259)

[**4.3.1 程序功能简介** 17](#_Toc90614260)

[**4.3.3 扩容和缩容功能** 18](#_Toc90614261)

[**4.3.4 心跳功能** 19](#_Toc90614262)

[**4.3.5 容灾功能** 21](#_Toc90614263)

[**4.4.1 线程池模块** 22](#_Toc90614264)

[**4.4.2 客户端通信模块** 23](#_Toc90614265)

[**4.4.3 服务端通信模块** 23](#_Toc90614266)

[**5. 系统通信格式** 23](#_Toc90614267)

[**5.1 Cache-Master通信格式** 23](#_Toc90614268)

[**5.1.1 Cache接收Master信息** 23](#_Toc90614269)

[**5.2.2 cache发送给master的信息** 25](#_Toc90614270)

[**5.3 Cache-Client通信格式** 25](#_Toc90614271)

[**5.3.1 Cache接收Client的信息** 25](#_Toc90614272)

[**5.3.2 Cache返回给Client的信息** 25](#_Toc90614273)

[**6. 系统测试报告** 25](#_Toc90614278)

[**7. 成员分工安排** 25](#_Toc90614279)

**1. 概述**

在云计算时代，全球的数据存储量以平均每年30%的速度不断增长。在这些数据中，用户访问的主要都是最近上传的热数据。作为存储系统应对热数据访问的有效补充，一个好的缓存系统就显得越来越重要。本文设计了一个简单的分布式缓存系统，并基于该系统实现了分布式缓存系统的管理、分布式访问、扩缩容、容灾等功能。

该分布式缓存系统的总体架构如下：

图形用户界面, 应用程序

描述已自动生成

图1-1 分布式缓存系统设计模式图

系统总体上由一个Master，多个Cache Server，以及若干client组成。Master负责管理数据分布信息，每个Cache Server以LRU的形式缓存了一部分Key-Value数据。Client在读写数据之前，先查看本地数据分布缓存，如果有效，则根据key去访问对应的Cache Server；如果无效则需要重新从Master拉取分部信息。Cache Server定期向Master汇报存活信息，如果Cache Server故障，则心跳失效，Master会重新划分数据分布，并通知仍然存活的Cache Server这一决定。Cache Server的扩容和缩容逻辑与上面的描述类似。

**2. 客户端（Client）程序实现**

**2.1 定时器Timer类**

**2.1.1 功能**

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。

**2.1.2 部分成员**

文本

描述已自动生成

图2-1 timer类的部分成员

**2.1.3 实现方法**

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。

**2.2 Client设计**

**2.2.1 本地缓存**

为了减轻Master-Server的压力，增加一层本地缓存存放Key-Addr数据，防止每次都需要拉取分布导致Master-Server压力过大。缓存方案选择了Key-Addr，在实现上采用LRU作为缓存淘汰方案。

**2.2.2 工作流程**

图示

描述已自动生成

图2-2 模块间的工作关系

图片包含 图示

描述已自动生成

图2-3 Client的工作流程

**2.2.3 异常处理**

1）应用层超时重传

TCP虽然可以保证数据可靠传输，但传输成功不代表被对方逻辑层成功处理。基于内存容量方面的考虑，服务端可能使用了限长的消息队列，如果收到的瞬时消息过多，超过了消息队列的可处理个数，所有超出的消息会被它丢弃。因此，应用层超时重传可以解决没有被逻辑层成功处理的情况。

图形用户界面, 文本, 应用程序

描述已自动生成

为每个Cache-Server连接分配一个定时器。

文本

描述已自动生成2）收发包管理

当Cache-Server缩容或宕机时，无论通过本地缓存的addr还是通过Master-Server再获取的addr，都已经失效，此时读写请求将会失败。为了保证Client的每一个请求都能够得到正确响应，在本地为每个Cache-Server维护一个Key-List用于进行收发包管理。Client向Cache-Server发送一次请求时，将Key存入本地的某个对应此Cache-Server的Key-List中。当Client收到此Cache-Server的回复后，消除掉此Cache-Server的Key-List中对应的请求key。如果某个Key-List的size超过一定的大小，则判定该Key-List所属的Cache-Server出现了缩容或宕机。在主动发现缩容或宕机后，重新连接Master并获取没有被响应的Key的最新分布，然后重新向Cache-Server发送请求。

手机屏幕截图

描述已自动生成

3）日志

使用easylogging++作为日志库。Easylogging的特点是只需一个头文件，所有功能都是内部实现，无需依靠其他第三方库，使用方便。

4） 需要改进之处

1. 定时器定时时间只能是遍历间隔的整数倍，可以使用基于升序链表的定时器容器、时间轮、时间堆来处理多个定时事件

2. 某个Cache-Server宕机后，Master-Server无法立刻发现其宕机，此时若Client访问宕机的Cache-Server，将无法连接，那么Client会去Master-Server拉取最的分布，但此时Master-Server并不知道Cache-Server宕机，因此其内部的分布信息依然是旧的。

3. 在Client上线初期，将多次访问Master-Server以建立本地缓存，若很多Client仅仅只查询少量几次数据，则Client本地的Key-Address缓存可能无法被使用到，即在此种情况下Key-Address缓存失效，造成Master-Server压力增大。

4. 无法实现透明传输，数据包采用#为控制符，Key或Value中出现'#'会产生错误

**3. 控制端（Master）程序实现**

# **3. 1程序结构简介**

# **3.2 底层算法和数据结构**

## **3.2.1 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。

## **3.2.2 cache映射**

为了保存每个上线的cache的相关信息，将socket连接建立时分配的文件描述符fd与fdmap结构体指针映射起来，数据结构为std::unordered\_map<int, struct fdmap \*> caches\_list;。用fd作为键值是考虑到当有cache发送信息时，其fd是可以直接拿到，通过访问其映射的结构体内容，就能直接得到cache的主备份信息，端口信息等，便于进行数据处理。

fdmap的数据结构：

struct fdmap {

char status; // 'P'/'R'，主备份状态

int fd; // socket的文件描述符

// int vec; //待使用的实际节点映射

int pair\_fd; // pair fd

std::string ip\_port; // IP#PORT\_for\_client

std::string ip\_cache; // IP#PORT\_for\_cache

fdmap(int fd) :status('n'), fd(fd), pair\_fd(-1), ip\_port("0"), ip\_cache("0"){} //无参数的构造函数数组初始化时调用

};

## **3.2.3 存活检测**

cache server会周期性向Master发送心跳包，Master接收到cache的心跳包就会更新该cache对应的时间戳，数据结构为：unordered\_map<string, time\_t> cacheAddrMap；。

Master为了确定cache server 的存活，会周期性扫描这个字典，确保接收到最后一次心跳的时间与扫描时刻的差值不超过一个门限。若差值超过门限，说明该cache可能意外关闭，此时Master需要检查该掉线的cache的主备份状态，如果是备份cache，将其对应的主cache中配对cache删除；如果是主cache，检查是否有备份cache，若有，将其转为主cache，若没有，直接删除节点，然后更新cache节点信息。（流程图见图3-1，如果有的话？？？？？？？）。

# **3.2 程序功能介绍**

## **3.2.1 client查询cache地址功能**

Master收到client传输来的key后，计算key存储的cache地址。

首先根据key的具体内容通过一致性哈希计算得到虚拟节点对应的值，将虚拟节点映射到实际节点的索引，并根据通过fdmap索引查询到实际节点对应cache的ip和port。最后将ip和port的格式转化为“ip:port”，将该信息返回给client。

## **3.2.2 主备份分配功能**

1、当检测到cache上线，添加给它分配的fd到fd\_node列表，并初始化该fd在cache\_list 中的映射值，即结构体struct fdmap，用于保存该cache的相关信息。

2、当收到心跳包后，若cache相关信息不完整，说明是第一次发送信息，根据心跳包内容补充该cache的相关信息，并尝试分配主备份，具体如下。

如果上线的是主cache\_1，判断是否有未配对的备份cache\_2（未配对的备份cache保存在堆栈中，检查堆栈是否非空即可）。如果有备份cache\_2，则设置cache\_1的对应cache为cache\_2, cache\_2的对应cache为cache\_1，弹出备份cache栈（rcache）中的cache\_2。如果没有备份cache，则将主cache\_1放到主cache栈（pcache）中。

在分配完主备份后，master向cache进行回复，通信格式为：配对不成功：“P#None”；配对成功：“P#rcache\_IP#port\_for\_cache”

如果上线的是备份cache\_2，判断是否有未配对的主cache\_1（未配对的主cache与未配对的备份cache处理类似）。如果有主cache\_1，则设置cache\_2的对应cache为cache\_1, cache\_1的对应cache为cache\_2，弹出主cache栈（pcache）中的cache\_1。如果没有备份cache，则将主cache\_2放到备份cache栈（rcache）中。

在分配完主备份后，master向cache进行回复，通信格式为：配对不成功：“R#None”；配对成功：“R#pcache\_IP#port\_for\_cache”

## **3.2.3 心跳功能**

进行心跳时间戳更新：master从cache处接收格式为【x#ip#port】的心跳包，对接收信息进行解析，获取该cache对应的ip和port，获取当前的时间戳，更新cacheAddrMap中该cache对应的时间戳。（cacheAddrMap<cache的ip和port， 时间戳>）

检测cache是否存活：对所有给该部分传输cache对应的fd， 根据fd获取该cache的ip和port，通过cacheAddrMap获取该cache现存的时间戳值；获取当前的时间戳；对当前的时间戳和存储的时间戳做差值，当差值不超过设置的心跳间隔时间，则该cache存活，否则不存活。

## **3.2.4 cache容灾功能**

1）如果是主cache\_1掉线

将备份cache\_2中的配对cachefd设置为-1，并将状态设置为P（主cache），从而完成备份cache切换为主cache；将cache\_2设置为待配对状态；将fd列表（fd\_node）中主cache\_1的ip和port改为备份cache\_2的ip和port；最后进行信息通知，具体通知为： 如果cache\_1有备份cache\_2，则master通知备份cache\_2现在是主cache，通知所有cache，将原本存ip\_port的数据里，cache\_1的位置更新为cache\_2的位置；如果cache\_1没有备份cache，则master通知所有cache，将原本存ip\_port的数据里,cache\_1的数据删除。

2）如果是备份cache\_2掉线

Master设置本地cache列表（cache\_list），删除备份cache\_2对应的主cache\_1的备份，并将cache\_1设置为待配对状态；删除master中cache列表的备份cache\_2；master通知主cache，它没有备份cache了。

## **3.2.5 扩缩容功能**

缩容：首先根据fd列表（fd\_node）获取要删除的cache索引（默认删除最后一个cache），通过一致性哈希中删除节点算法，设置哈希运算。之后根据cache索引获取到缩容cache的ip（killed\_ip）和port(killed\_port)；将killed\_ip和killed\_port以【K#killed\_ip#killed\_port】格式广播给所有的cache。最后减少cachefd列表（fd\_node）中缩容cache的fd；删除cache列表（cache\_list）中缩容cache的信息；关闭缩容cache通信；如果存在备份cache，删除cache列表（cache\_list）中缩容cache的备份cache的信息；关闭备份cache通信。

扩容：当上线的cache为主cache时，进行扩容操作。首先通过一致性哈希中增加节点算法，设置哈希运算。之后根据新上线cache的fd值获取该fd对应的cache的ip(new\_ip)和port(new\_port)，将ip和port以【N#new\_ip#new\_port】格式广播给所有的cache。最后将所有的cache的ip和port以【ip1#port1#ip2#port2...】格式发送给新上线的cache。

# **3.3程序模块简介**

## **3.3.1 Master-client通信模块**

Master对Client的通信在start\_client()中实现，利用IO复用技术，实现socket监听以及与客户端的socket通信。为了方便通信读写，每个cache的socket连接都会分配一个文件描述符fd。

具体处理的工作为：接收Client发送的读/写key 请求，返回该key对应的cache服务器地址。

## **3.3.2 Master-cache通信模块**

Master对Cache的通信在start\_cache()中实现，基本原理和Master对Client的通信类似。由于需要向完成1）Client分配地址，2）进行一致性哈希运算，3）容灾几项工作，Master需要记录各个Cache的基本信息，包括1）对cache的地址，2）对client对地址，3）主备份状态，4）对应的主备份cache的fd。

具体处理的工作有：

1. 接收Cache的心跳包，回复为其分配的主/备份cache的地址；发送的读/写key 请求，返回该key对应读cache服务器地址。
2. 扩容时，向新上线的主cache发送已有的cache节点信息，并向所有cache节点发送新上线节点的消息
3. 缩容时，向所有cache节点发送删除的cache节点的消息
4. 容灾时，在检测到有主cache失效时，广播发送cache节点的更新消息。

## **3.3.3 周期性心跳检测模块**

Master通过心跳检测功能周期性检测所有cache的存活状态，存在不存活的cache时，执行相应的容灾功能。

## **3.3.4 缩容模块**

Master循环监听键盘的输入，当输入的字符为’s’ 时，进行缩容操作。

**4. 缓存端（Cache）程序实现**

**4.1. 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 4-1 缓存程序的结构示意图

**4. 2. 底层算法和数据结构**

**4.2.1 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。

**4.2.2 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4-2所示。

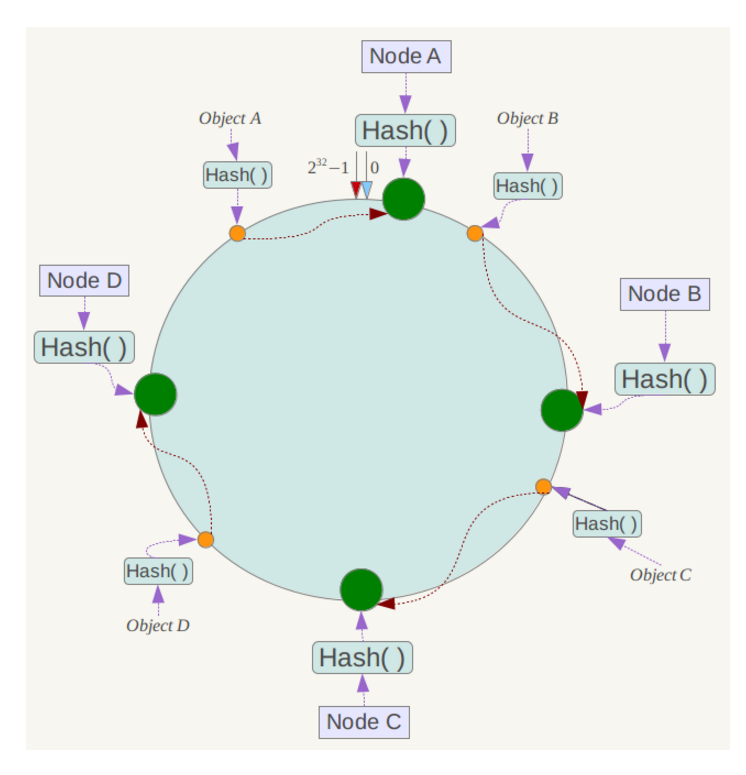


图4-2 一致性哈希算法示意图

**4.2.3 地址表**

在每Cache程序中，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。

**4.3. 程序功能**

**4.3.1 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。

**4.3.2 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：

表4-1 cache-client键值请求回复格式

|  |  |  |
| --- | --- | --- |
| 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 |

**4.3.3 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4-3所示。



（a）执行扩容命令



（b）执行缩容命令

图4-3执行扩、缩容时的通信时序图

**4.3.4 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

表4-2 cache-master心跳包通信格式

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 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 |

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

图4-4 定时器部分成员

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。

**4.3.5 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

图4-5 容灾功能逻辑图

**4.4. 程序模块简介**

**4.4.1 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4-6线程池的原理示意图

**4.4.2 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。

**4.4.3 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。

**5. 系统通信格式**

**5.1 Cache-Master通信格式**

**5.1.1 Cache接收Master信息**

**1）扩容**

1. 初始化cache列表：master向新上线的cache单独发送现有的所有主cache的IP和port\*\*（包括新上线的主cache自己）\*\*，通信格式：ip1#port1#ip2#port2#ip3#port3（这里的port指的是主cache开给别的cache的端口）
2. master向所有主cache广播新上线cache的IP和port，通信格式：N#ip#port

**2）缩容**

master向所有的cache发送K#killed\_ip#killed\_port，其中killed\_ip是被缩容的主cache IP，killed\_port是主cache开给别的cache的port

（运行过程说明：master在键盘输入缩容信号，随后master向所有cache广播K#killed\_ip#killed\_port，cache接收到信息后会下线被缩容的主cache和对应的备份cache）

**3）容灾**

所有命令可总结为：P/R#target\_ip#target\_port或P#None，以下详细说明：

a) 正常情况下：

1. master向主cache发送P#target\_ip#target\_port，要是主cache没有备份cache则发送P#None，target\_ip表示备份cache的ip，target\_port表示备份cache开给别的cache的port
2. master向备份cache发送R#target\_ip#target\_port，target\_ip表示主cache的ip，target\_port表示主cache开给别的cache的port

b) 非正常情况下，主cache宕机，容灾起效：

1. master向失效主cache的备份发送P#None，已转正的cache要是有备份cache上线，master向已转正的cache发送P#target\_ip#target\_port
2. 主cache有备份cache：master通知所有cache（包括备份）cache\_list更新为备份的cache，通信格式为：C#origin\_ip:origin\_port#backup\_ip:backup\_cache
3. 主cache没有备份cache：master通知所有cache（包括备份），将cache\_list中的主cache删除，通信格式：D#delete\_ip#delete\_port

（具体运行过程补充说明：先有一个主cache上线，此时尚未有备份cache上线，master给主cache发送P#None，待有新cache上线且新cache的输入参数为R时，master要将该备份cache分配给所有没有备份的主cache中的一个，给被分配的备份的主cache发送P#target\_ip#target\_port（target\_ip和target\_port为该备份cache的ip和port）；给该备份cache发送R#target\_ip#target\_port，target\_ip表示主cache的ip，target\_port表示主cache开给别的cache的port）

**5.2.2 cache发送给master的信息**

心跳包格式：x#local\_cache\_IP#port\_for\_client#port\_for\_cache#P/R

**5.3 Cache-Client通信格式**

**5.3.1 Cache接收Client的信息**

1. 读：key
2. 写：key#value

**5.3.2 Cache返回给Client的信息**

1. 读成功：SUCCESS#key#local\_cache\_IP:port\_for\_client\_ #value
2. 读失败：FAILED#key#local\_cache\_IP:port\_for\_client
3. 写成功：SUCCESS#key#local\_cache\_IP:port\_for\_client

**5.3 Master-Client通信格式**

1. master接收client的信息：key
2. master返回给client的信息：MASTER#key#ip:port

**6. 系统测试报告**

**7. 成员分工安排**

|  |  |  |
| --- | --- | --- |
| **成员姓名** | **负责模块** | **具体负责内容** |
| 曹晨涛 | Cache | 网络通信功能实现  程序测试安排 |
| 王亦航 | Cache | 容灾功能实现  答辩ppt撰写 |
| 王琛 | Cache | LRU、多线程功能实现  程序报告撰写 |
| 庞家明 | Client | Client功能实现  程序测试  程序报告撰写 |
| 牛鹏程 | Master |  |
| 宋耀辉 | Master |  |
| 李明悦 | Master |  |