Tencent 腾讯引力计划前置化培养班

简单分布式缓存系统

程序开发报告

第 2 组

小组成员: 曹晨涛 王易航 王琛 庞家明 牛鹏程 宋耀辉 李明悦

2021 年 12 月

目 录

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：

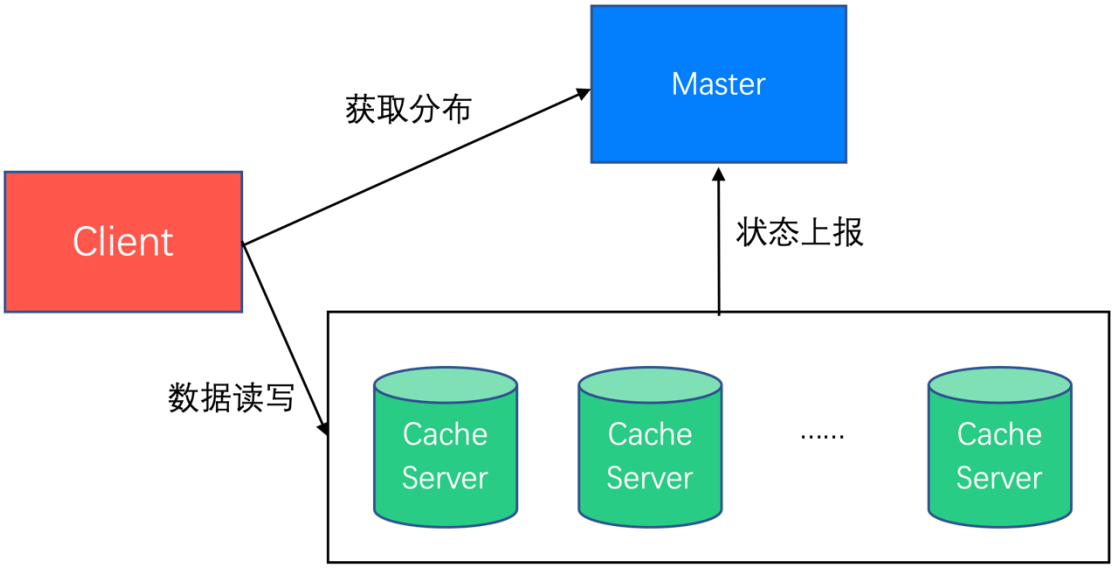


图 1-1 分布式缓存系统设计模式图

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

```
// tiner.h

void start();   // 开启
void stop();    // 停止
void reset();   // 重置
void pause();   // 暂停

void setInterval(int32_t timeout_ms);     // 设置定时间隔
void setPeriodic(bool isPeriodic);        // 单词定时 or 循环定时
void setCallback(std::function<void (void *)> TimerCallback);   // 设置回调函数
```

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

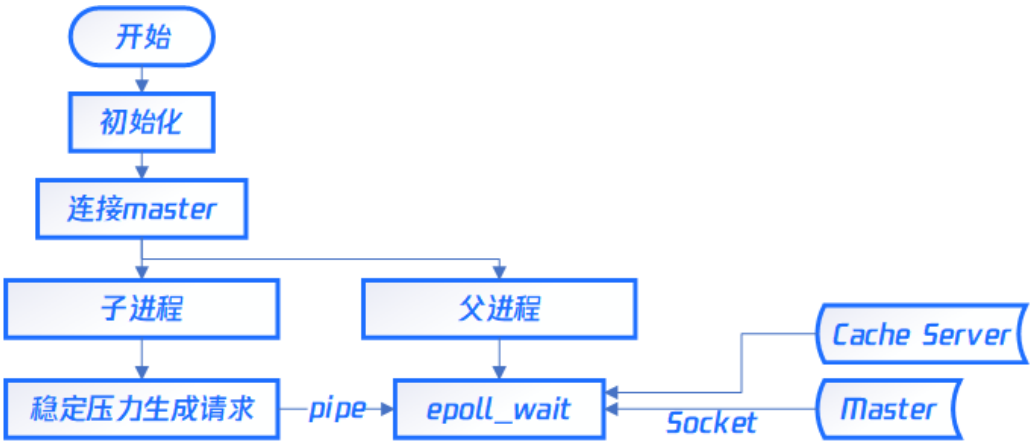


图 2-2 模块间的工作关系

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。

```
// 定义全局变量pTimers存储全部定时器
std::list<Timer *> Timer::pTimers;
```

为每个 Cache-Server 连接分配一个定时器。

```
struct ReSendMassage {
    int sock;
    std::string addr;                // Cache-Server地址
    std::string massage;             // 需要重传的包
    std::shared_ptr<Timer> timer;    // 定时器指针
};
// cache server重传定时器表
std::unordered_map<std::string, ReSendMassage> cache_server_timers;
```

（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。

```
// 管理向cache_server发送的请求,<ip, list<key>>
std::unordered_map<std::string, std::list<std::string>> request_map_;
```

（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

3.1.1 底层算法和数据结构

（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同图 2-2，如下所示。

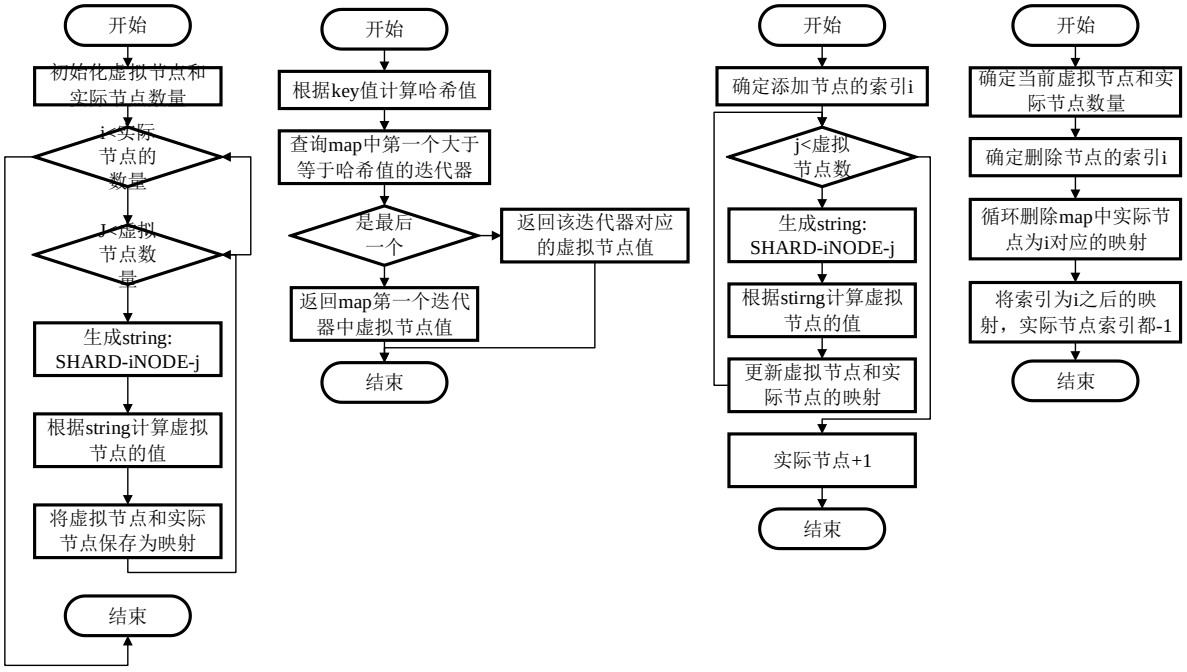


图 3-1 一致性哈希算法模式

（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：

```
1.  struct fdmap {
2.      char status;                    //  'P'/'R'，主备份状态
3.      int fd;                         //  socket 的文件描述符
4.      // int vec;                     //待使用的实际节点映射
5.      int pair_fd;                    //  pair fd
6.      std::string ip_port;        //  IP#PORT_for_client
7.      std::string ip_cache;       //  IP#PORT_for_cache
8.      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
9.  };
```

（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。。

3.2 程序功能介绍

3.2.1 client 查询 cache 地址功能

Master 收到 client 传输来的读写信息（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哈希环上虚拟节点对应的值，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 cache 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，通信格式为当配对不成功:“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：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

心跳功能通过存储接收心跳包的时间，并周期性检测所有 cache 的存活状态来确定 master 连接的 cache 是否掉线。

进行心跳时间戳更新：master 从 cache 处接收格式为“x#local_cache_IP#port_for_client#port_for_cache#P/R”的心跳包，对接收信息进行解析，对相应的 ip 和 port 进行存储。获取 cache 对应的 local_cache_IP 和 port_for_cache，并当前的时间戳，从而更新 cacheAddrMap 中该 cache 对应的时间戳。（cacheAddrMap< local_cache_IP#port_for_cache， 时间戳>）

检测 cache 存活状态：根据存储的 cache 时间戳和检测时刻的时间戳确定 cache 是否存活。设置心跳间隔最大容忍时间为 5s。首先，获取检测 cache 的 fd，从而根据 fd 获取该 cache 的 ip 和 port，通过 cacheAddrMap 获取该 cache 现存的时间戳值；之后，获取检测时刻的时间戳；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最大容忍时间时，则该 cache 存活，否则不存活。

周期性心跳检测：访问 cacheAddrMap 中所有的 cache，通过检测存活状态的函数确定 cache 是否掉线。当无 cache 掉线，则重复周期性检测；否则，进行相应的容灾处理。

3.2.4 cache 容灾功能

容灾功能实现当 cache 掉线后，根据掉线 cache 的属性，通过 master 中数据结构的删改、master-cache 之间的信息传输，实现信息更新和信息同步并减少 client 的信息访问错误。假设主 cache 为 cache_1，当 cache_1 存在备份 cache 时，备份 cache 为 cache_2。

（1）如果是主 cache_1 掉线

当主 cache_1 存在备份 cache2。首先更新 fd 列表（fd_node）中的值：将 fd 列表中主 cache_1 的 fd 改为备份 cache_2 的 fd。之后进行信息发送：master 以 C#origin_ip:origin_port#backup_ip:backup_cache 格式，通知所有 cache，将原本存储的 origin_ip:origin_port，修改为 backup_ip:backup_cache，从而保证信息同步。最后更新本地信息：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；由于此时 cache_2 无配对 cache，因此将 cache_2 设置为主 cache 中的待配对状态。关闭掉线 cache 的 socket。最后退出后删除 master 中 cache 列表的主 cache_1。

当主 cache_1 无备份 cache。首先删除 fd 列表（fd_node）中的值：将 fd 列表中主 cache_1 的 fd 删除。之后进行信息发送：master 以 D#delete_ip#delete_port 格式，通知所有 cache 将原本存 delete_ip#delete_port 的数据删除。关闭掉线 cache 的 socket。最后退出后删除 master 中 cache 列表的主 cache_1。

（2）如果是备份 cache_2 掉线

当备份 cache 掉线后，不需要进行 master 进行信息传输。之后进行 master 的信息更新。首先更新主 cache 的备份信息：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。之后更新 cache 列表：确定 cache 列表中要删除的 cache 信息。Master 将通过心跳检测后的信息回传通知主 cache，它没有备份 cache 了。关闭 master 与 cache_2 之间的 socket。删除 master 中 cache 列表的备份 cache_2。

3.2.5 扩缩容功能

缩容：缩容功能通过监听 master 的终端输入，当输入缩容标识时，更新本地信息并通知缩容设备进行信息迁移，最后关闭缩容 cache 及其备份 cache。

哈希更新：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。cache 通信实现信息同步：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。本地信息更新：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：当主 cache 上线时，执行扩容操作，更新本地信息，并对新上线的 cache 和已有 cache 进行信息共享。

哈希更新；当上线的 cache 为主 cache 时，通过一致性哈希中增加节点算法，设置哈希运算。cache 信息共享：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 的格式广播给所有的 cache。最后将所有的 cache 的 ip 和 port 以 ip1#port1#ip2#port2...的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。

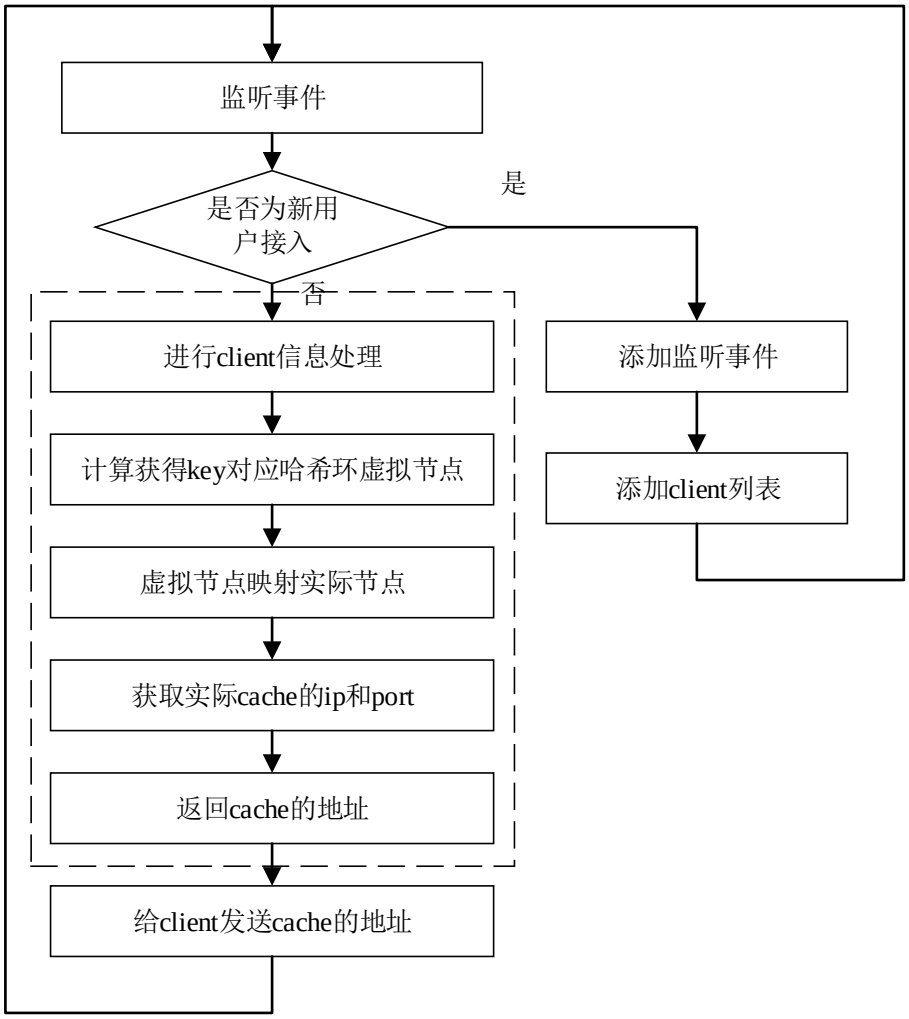


图 3.1 master-client 通信流程图

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。

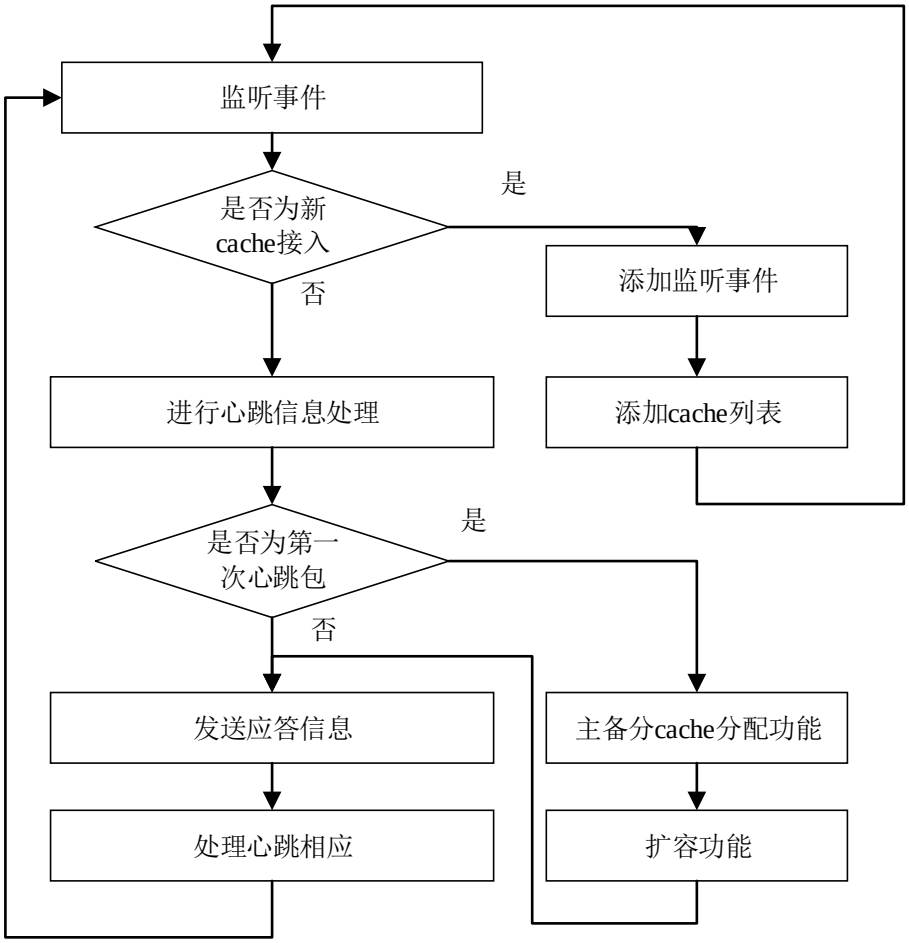


图 3.2 master-cache 通信流程图

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。

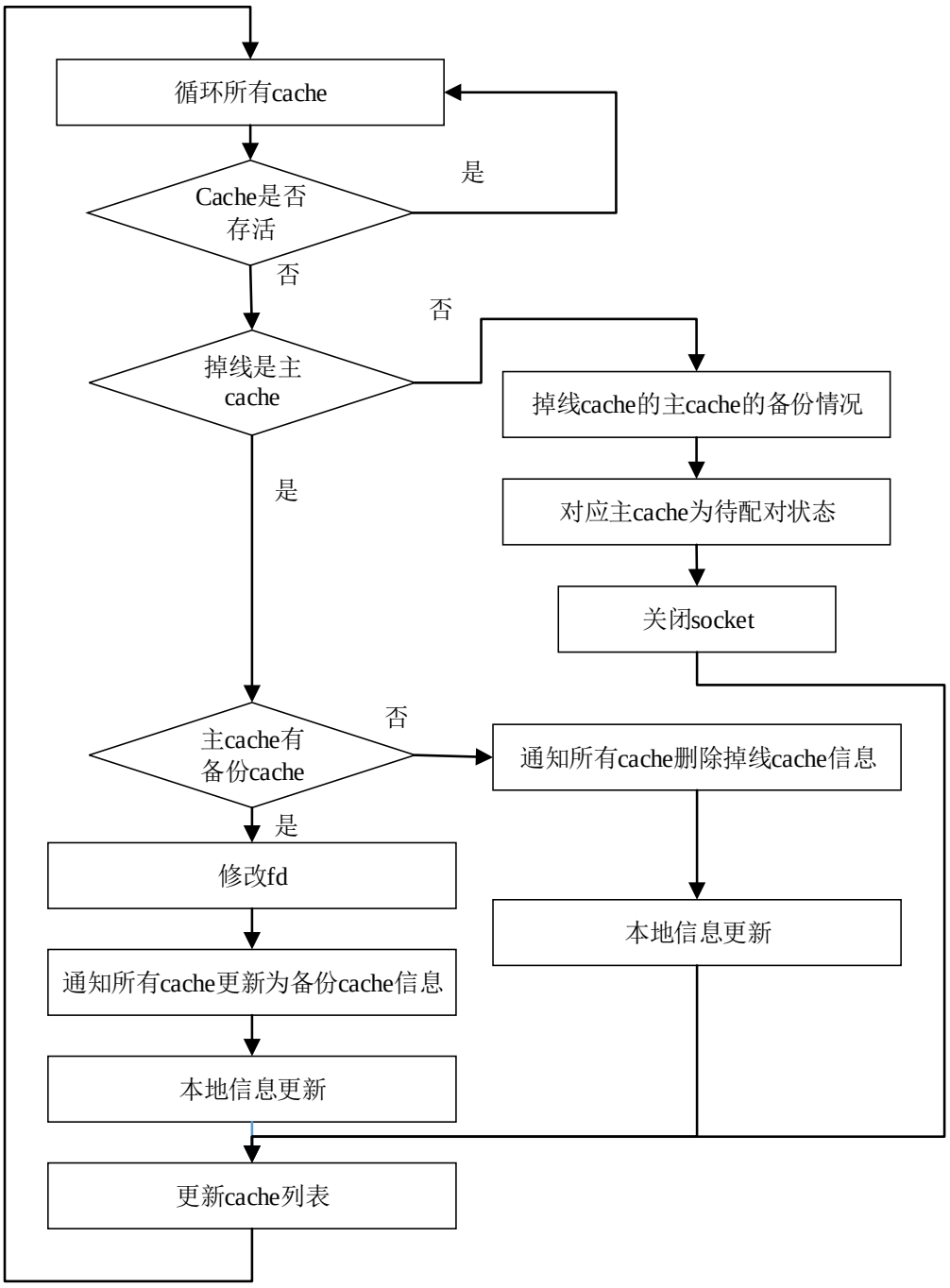


图 3.3 周期性心跳检测通信流程图

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。

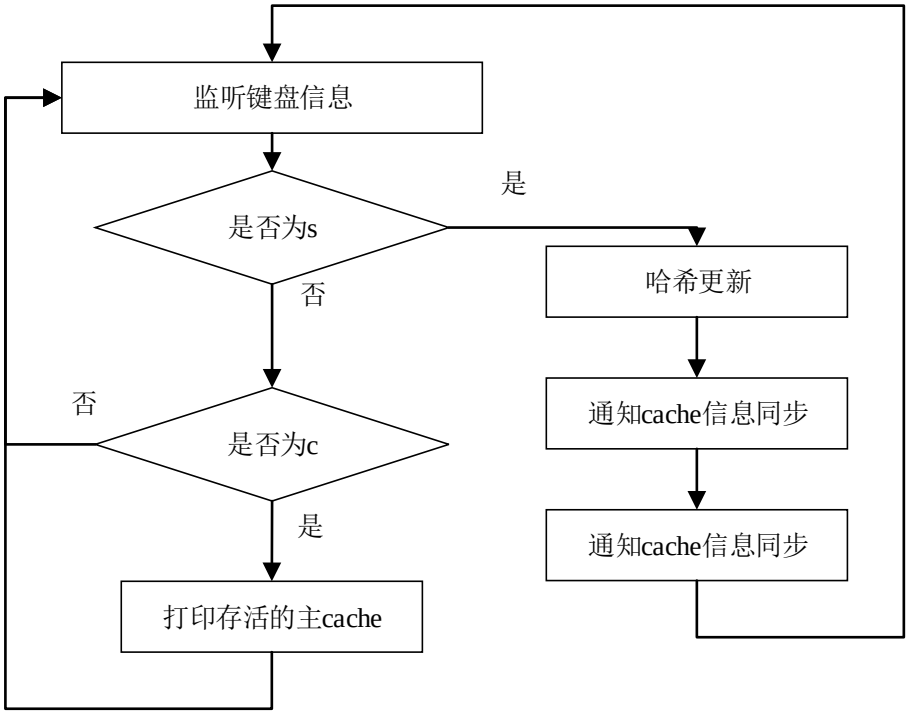

监听键盘信息
是否为s
是
否
哈希更新
通知cache信息同步
否
是否为c
是
通知cache信息同步
打印存活的主cache

图 3.4 缩容流程图

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。

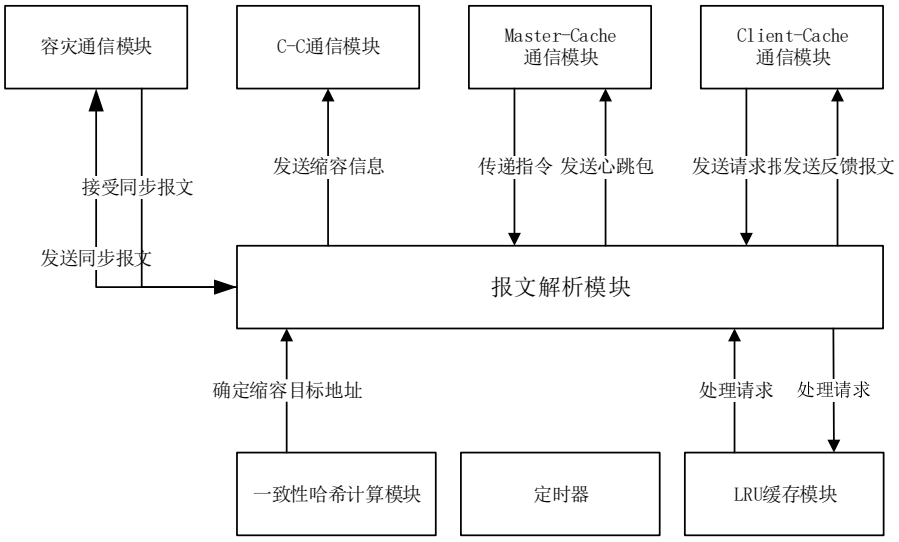


图 4-1 缓存程序的结构示意图

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。

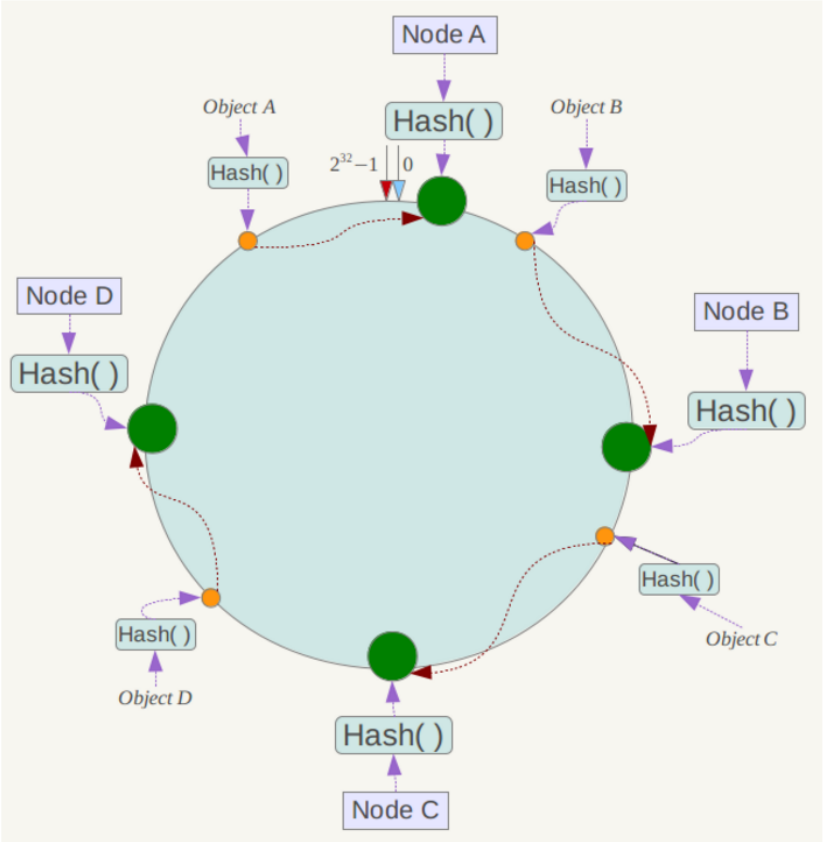


图 4-2 一致性哈希算法示意图

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

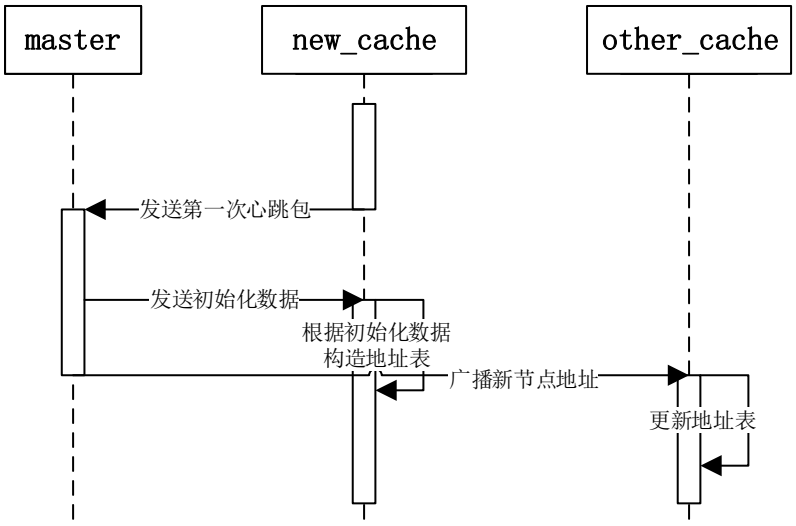
4.3.3 扩容和缩容功能

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。

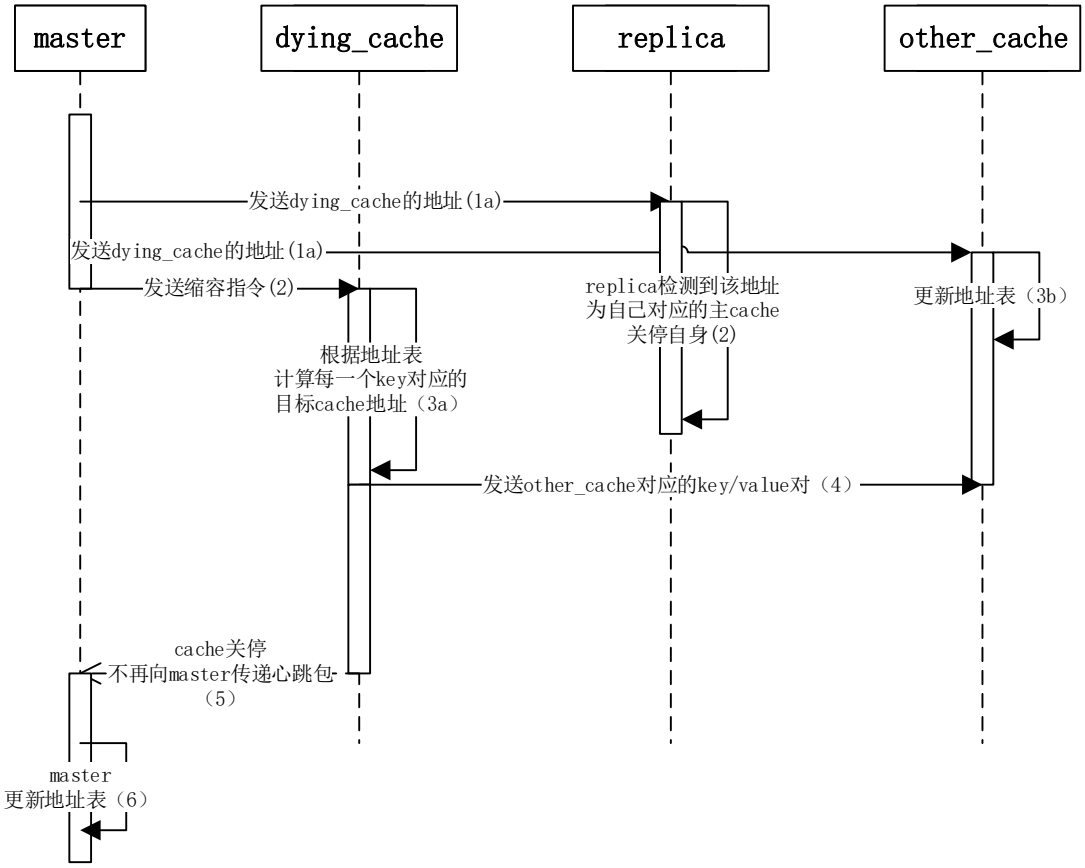
当控制端发出扩容命令时，只需要将新增的缓存节点加入自身的地址表中即可，反之，控制端发出缩容命令时，则需要指定一个节点进行数据转移和关停。此时，则需要分配该节点所掌握的所有键值对。

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：

```
// tiner.h

void start();   // 开启
void stop();    // 停止
void reset();   // 重置
void pause();   // 暂停

void setInterval(int32_t timeout_ms);     // 设置定时间隔
void setPeriodic(bool isPeriodic);        // 单词定时 or 循环定时
void setCallback(std::function<void (void *)> TimerCallback);   // 设置回调函数
```

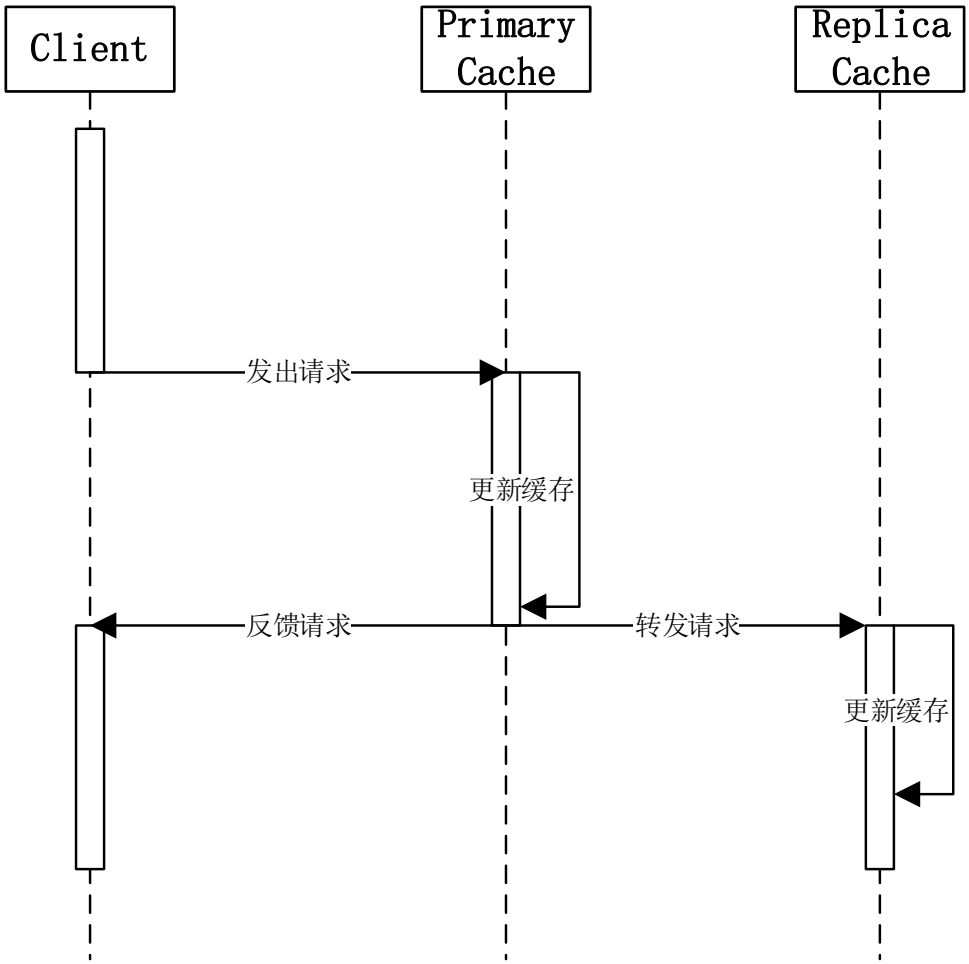
图 4-4 定时器部分成员

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。

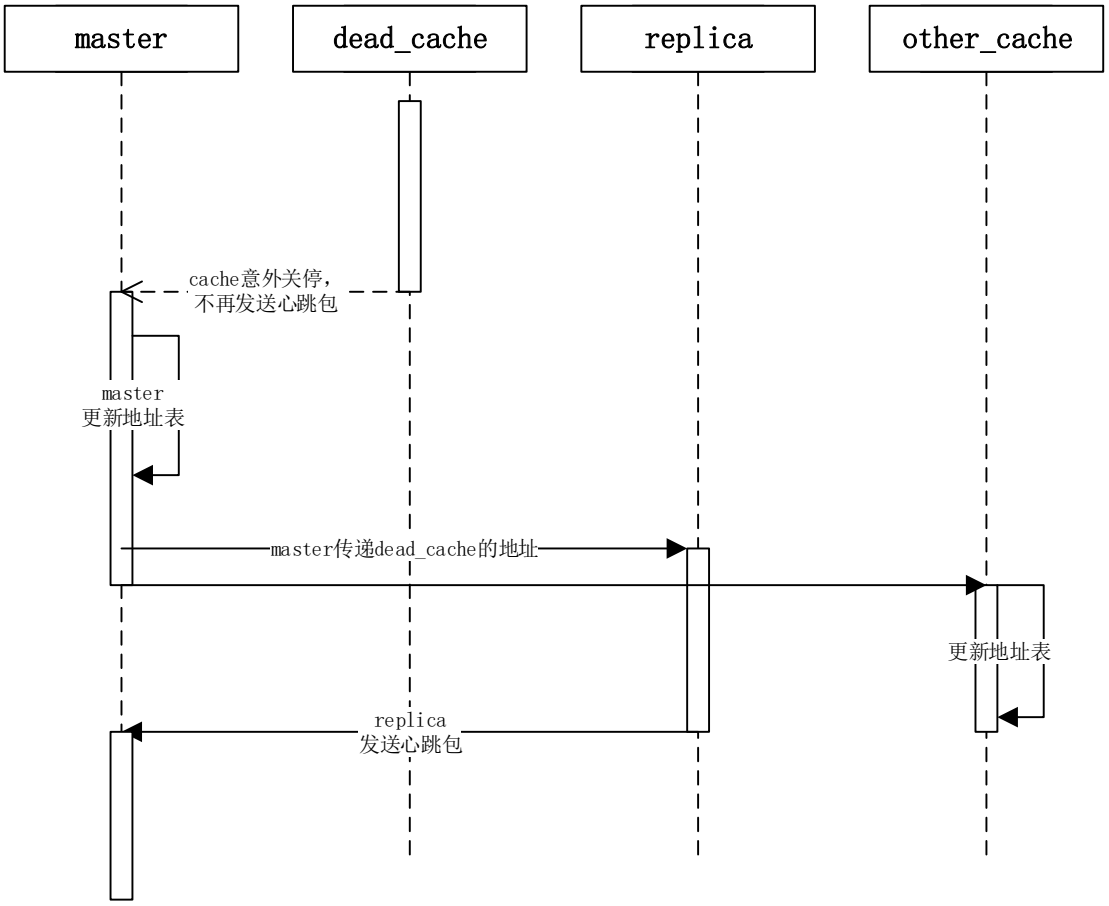
在 Heartbeat()中定义静态定时器智能指针，指向一个定时器对象，设置定时器超时回调函数。该回调函数向 Master 上报心跳信息并接收 Master 返回。

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：


Client
Primary Cache
Replica Cache
发出请求
更新缓存
反馈请求
转发请求
更新缓存

（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 4-6 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。

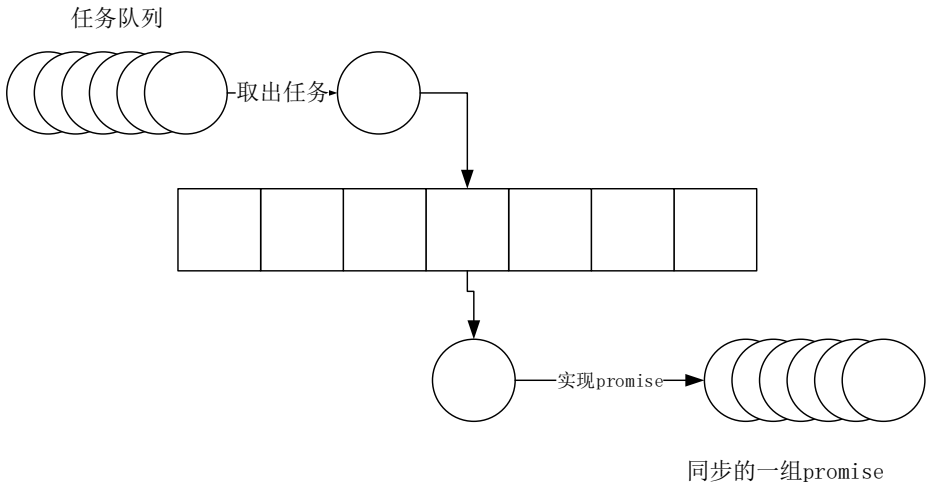


图 4-6 线程池的原理示意图

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

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）

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：

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

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：

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

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

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）

5.1.2 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

5.2 Cache-Client 通信格式

5.2.1 Cache 接收 Client 的信息

读：key

写：key#value

5.2.2 Cache 返回给 Client 的信息

读成功：SUCCESS#key#local_cache_IP:port_for_client_ #value

读失败：FAILED#key#local_cache_IP:port_for_client

写成功：SUCCESS#key#local_cache_IP:port_for_client

5.3 Master-Client 通信格式

master 接收 client 的信息：key

master 返回给 client 的信息：MASTER#key#ip:port

6 系统测试报告

6.1 功能实现

（1）LRU 置换算法

（2）一致性哈希负载均衡

（3）稳定压力分布式访问

（4）扩容

（5）缩容

（6）cache 容灾

6.2 测试环境

单核 CPU; 2GB 内存; CentOS; 所有角色运行在同一台服务器上

6.3 测试过程

（1）LRU 置换算法

执行 lru_test 单元测试文件。

```
// 简单的插入、读取操作
TEST(CacheTest, SimplePut) {
    cache::lru_cache<int, int> cache_lru(1);
    cache_lru.put(7, 777);
    cache_lru.put(8, 888);
    EXPECT_TRUE(cache_lru.exists(7));
    EXPECT_TRUE(cache_lru.exists(8));
    EXPECT_EQ(777, cache_lru.get(7));
    EXPECT_EQ(888, cache_lru.get(8));
    EXPECT_EQ(2, cache_lru.size());
}

TEST(CacheTest, MissingValue) {
    cache::lru_cache<int, int> cache_lru(1);
    EXPECT_THROW(cache_lru.get(7), std::range_error);
}
```

```
TEST(CacheTest1, KeepsAllValuesWithinCapacity) {
    cache::lru_cache<int, int> cache_lru(TEST_CACHE_CAPACITY);

    // 容量50，插入100个键值对，前50个被淘汰
    for (int i = 0; i < NUM_OF_TEST_RECORDS; ++i) {
        cache_lru.put(i, i);
    }
    for (int i = 0; i < NUM_OF_TEST_RECORDS - TEST_CACHE_CAPACITY; ++i) {
        EXPECT_FALSE(cache_lru.exists(i));
    }

    // 后插入的50个依然存在
    for (int i = NUM_OF_TEST_RECORDS - TEST_CACHE_CAPACITY; i < NUM_OF_TEST_RECORDS; ++i) {
        EXPECT_TRUE(cache_lru.exists(i));
        EXPECT_EQ(i, cache_lru.get(i));
    }

    size_t size = cache_lru.size();
    EXPECT_EQ(TEST_CACHE_CAPACITY, size);
}
```

（2）一致性哈希负载均衡 + 稳定压力分布式访问

上线 3 个 cache 服务，client 以 50ms 的周期随机生成请求，记录并打印向每个 cache 的请求次数。

```
Request to master, key: 4T4A3G5E5D
Num of request to 127.0.0.1:8884 is 2411
Request to cache server SUCCESS key: 4T4A3G5E5D cache addr: 127.0.0.
1:8884
Request to master, key: XKB3YVT09F
Num of request to 127.0.0.1:8886 is 2781
Request to cache server SUCCESS key: XKB3YVT09F cache addr: 127.0.0.
1:8886
Request to master, key: 41VIP1TY0A
Num of request to 127.0.0.1:8885 is 2803
Request to cache server SUCCESS key: 41VIP1TY0A cache addr: 127.0.0.
1:8885
```

（3）扩容

每当有新的 cache 上线时，master 与其它已经上线的 cache 都将打印新 cache 的上线信息。随后请求压力将被新的 cache 分散。

（4）缩容

Master 监听键盘信号，使用 s 模拟主动缩容操作，master 和其他 cache 都会打印被下线的 cache 信息。

```
Receive from master:K#127.0.0.1#9886
Shrink a cache, cache ip:127.0.0.1 cache port:9886
Number of online cache:2
Online cache list:
cache1 ip:127.0.0.1, cache1 port:9887
cache2 ip:127.0.0.1, cache2 port:9885
==================
```

（5）cache 容灾 + client 超时重传 + client 主动发现 cache 宕机

通过手动 kill 一个 cache 模拟宕机，

在从 cache 宕机到完成容灾处理之间，client 的请求可能会发送到原 cache 上，导致请求无法被正确响应。Client 的日志记录了超时重传及主动发现 cache 宕机并处理未被响应的包的过程

```
1:18,117 | INFO] | Client start
1:24,839 | ERROR] | Connect cache server failed, addr: 127.0.0.1:8885
1:25,239 | ERROR] | Connect cache server failed, addr: 127.0.0.1:8885
1:25,540 | ERROR] | Connect cache server failed, addr: 127.0.0.1:8885
1:25,660 | ERROR] | Connect cache server failed, addr: 127.0.0.1:8885
1:26,117 | INFO] | Cache server timeout retransmission, message = 3TU#R5D4KTJTR6TV
1:26,117 | ERROR] | Re-connect cache server failed, addr: 127.0.0.1:8885
1:26,683 | ERROR] | Connect cache server failed, addr: 127.0.0.1:8885
1:27,084 | ERROR] | Connect cache server failed, addr: 127.0.0.1:8885
1:27,184 | ERROR] | Connect cache server failed, addr: 127.0.0.1:8885
1:27,284 | INFO] | The cache server may be down: 127.0.0.1:8885
1:27,284 | ERROR] | Re-send to master, key: SMM
1:27,317 | ERROR] | Re-send to master, key: 51T
1:27,417 | ERROR] | Re-send to master, key: 4B0
1:27,517 | ERROR] | Re-send to master, key: 3TU
1:27,617 | INFO] | Cache server timeout retransmission, key = PX4#5T9397LE37R0KCE1
1:27,617 | ERROR] | Re-connect cache server failed, addr: 127.0.0.1:8885
1:27,617 | ERROR] | Re-send to master, key: 2LE
1:27,717 | ERROR] | Re-send to master, key: SMQ
1:27,817 | ERROR] | Re-send to master, key: PX4
```

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	1.独立构建了 Cache 端的网络通信模块(cache 和 master 间的通信、cache 和 client 间的通信、cache 和 cache 间的通信) 2.参与构建了程序的扩缩容功能、容灾功能 3.参与 cache 端与 master 端的联合调试测试 4.组织小组讨论，协调制定 master 与 cache 间的通信协议 5.对最终报告和 ppt 进行加工美化排版 6.现场参与项目最终答辩
王易航	Cache	1.调研 cache 端的扩缩容与容灾的方案 2.参与构建了 cache 的定时器功能 3.整体设计报告撰写，答辩 PPT 撰写
王琛	Cache	1.独立构建了 cache 端的 LRU 模块、线程池模块 2.参与构建了程序的扩缩容功能、容灾功能和报文解析模块 3.程序报告、ppt 撰写
庞家明	Client	1.负责 client 部分的代码，文档 2.LRU 单元测试 3.定时器类 4.整个系统的调试 5.现场负责项目最终演示
牛鹏程	Master	1.负责 master 部分的负载均衡模块 2.负责 master 部分负载均衡模块的报告
宋耀辉	Master	1.负责 master 端处理 cache 的扩容、容灾功能

		2.负责 master 端处理 client 端信息的功能，Master 端的心跳响应与心跳检测功能 3.参与 master 端对 cache 容灾的处理 4.程序报告、ppt 撰写 5.现场参与项目最终答辩
李明悦	Master	1.负责构建 master 部分的通信框架与数据结构定义 2.负责 master 端的通信(master 和 cache 间的通信，master 和 client 间的通信) 3.处理 master 端的心跳信息响应与主备份 cache 的分配 4.参与 master 端对 cache 容灾的处理 5.程序报告、ppt 撰写