缓存服务器设计文档

一: 需求分析

1. 缓存服务器支持对数据缓存功能, 支持与数据库系统进行交互。
2. 提供对缓存数据的SET 与GET网络命令接口。
3. 服务器多线程环境下保护数据一致性。
4. 实现哈希表存储及哈希算法设计。
5. 保证缓存与数据库同步一致性。
6. 实现缓存与服务器进程分离，保证服务器突然挂掉，缓存数据能够保留。
7. 多并发。

PS： 鉴于开发机上位装数据库，估计先用文件代替

二: 模块划分

目标：实现一个基于 Key-Value键值对存储且支持一定并发量的缓存服务器。

主要模块：

1. 内存管理模块
2. 哈希表存储及查找模块
3. LRU双链表
4. 多线程模块

三：模块设计分析

1. 哈希表模块设计

哈希表冲突采用链地址法 (数组+ 链表)

哈希节点设计：

Template<class Key, class Value>

Struct SphHashNode {

Key m\_key;

Value m\_value;

SphHashNode\* m\_prev;

SphHashNode\* m\_next;

};

哈希函数设计：

Struct Hash

{

int operator()(const std::string & lhs) {

int b = 378551;

int a = 63689;

int h = 0;

for (int index = 0; index != lhs.size(); ++index) {

h = h \*a + lhs[index];

a \*= b;

}

return (hash & 0x7FFFFFFF);

}

1. LRU 双链表设计

节点设计：

Template<class Key, class Value>

Struct SphListNode {

Key m\_key;

Value m\_value;

bool m\_tag; // 更新标志位

SphListNode\* m\_prev;

SphListNode\* m\_next;

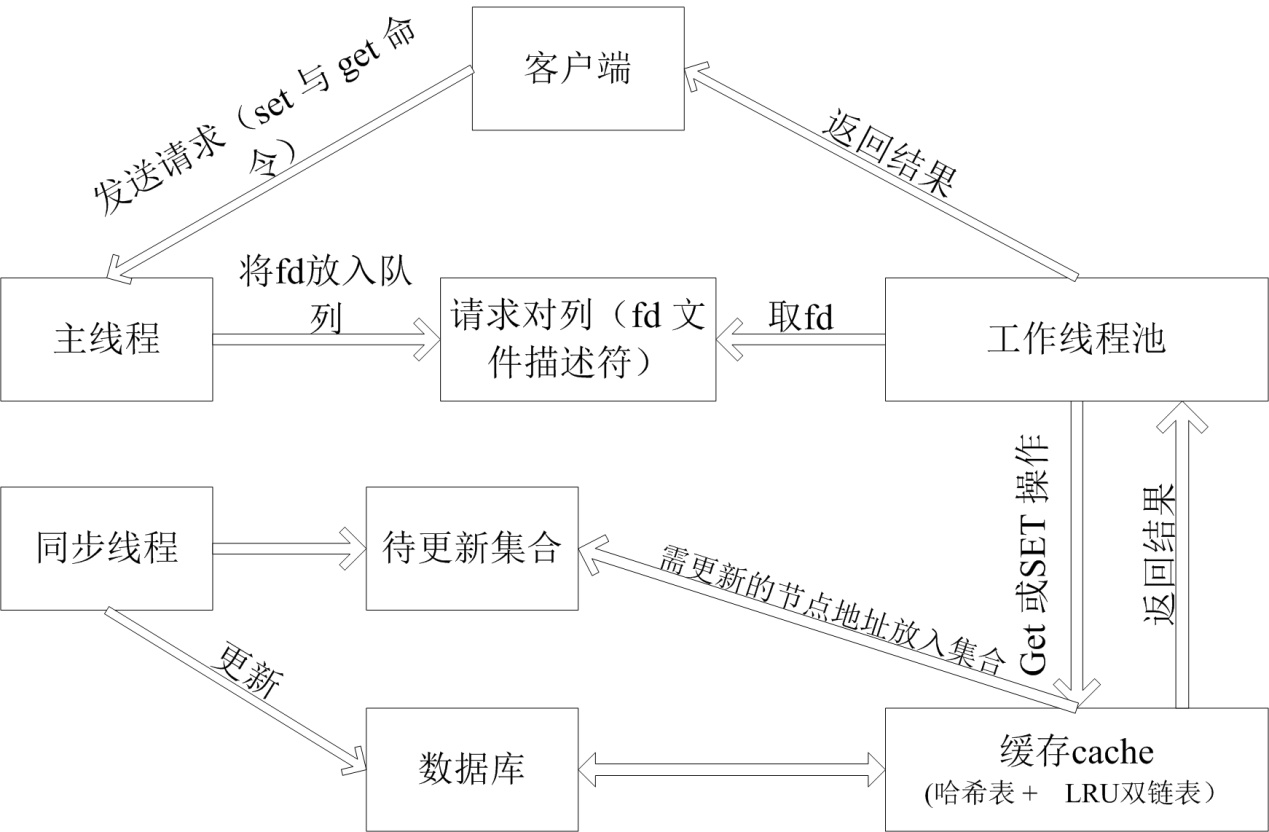
};

1. 内存管理模块

采用队列对所有可用内存地址进托管，当需要分配新的内存地址空间，则从中弹出一个内存地址。当vector数组所有可用内存地址分配完后，采用LRU策略回收内存地址。

1. 多线程模块

采用 epoll 　+　主线程 +　工作线程池　＋　同步线程 架构模式， 多线程模块工作流程见如下：



同步线程 用于 同步缓存与数据库中的数据

链表节点：

Template<class Key, class Value>

Struct SphListNode {

Key m\_key;

Value m\_value;

bool m\_tag; // 更新标志位, 默认为0

// 写命中时，置为1, 表示此节点数据需要

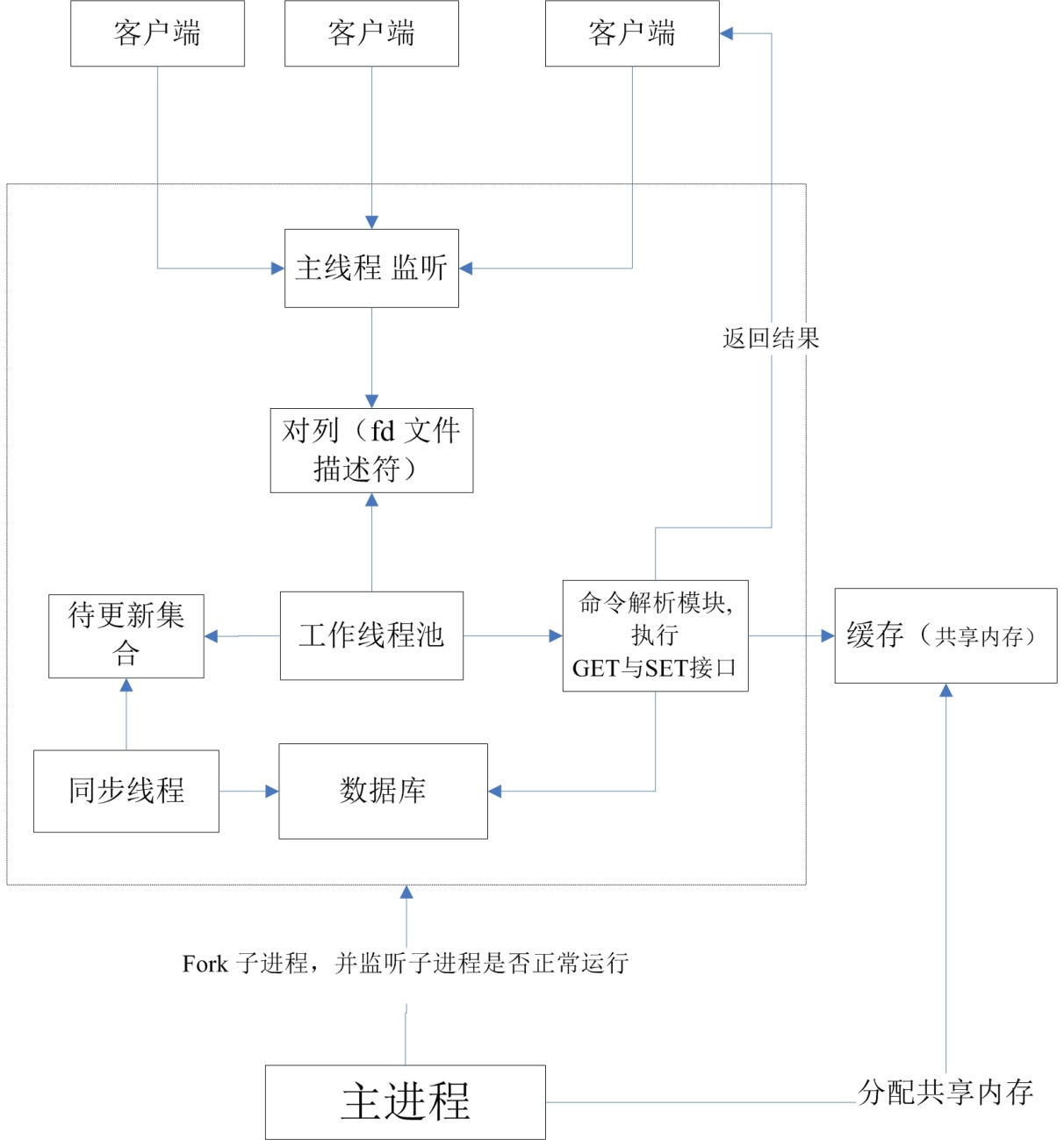
// 更新到数据库中

SphListNode\* m\_prev;

SphListNode\* m\_next;

};

四： 架构设计



子进程作为服务进程， 且缓存数据存储于共享内存， 使得服务进程与缓存数据分离。

当服务进程挂掉时，缓存数据能够保留。重新启动服务进程，即能够恢复正常工作。

五： 接口流程设计

Get 接口流程设计：

