共享内存cache架构设想

需求：

总的来说：设计一种使用共享内存作为cache的系统，cache由key-> value 组成

细的来说：此cache，有如下特点：

**强容错机制**

1. 当某挂接进程abort掉后，只破坏当时操作的key对应value；
2. 如果某进程操作指针破坏了共享内存某些地方，影响应该减到最小；

**各value之间严格隔离；**

**Lru淘汰简单，且与具体业务，具体数据无关；**

**内存分配简单，gc简单；**

**适合大单位复杂数据value；**

**key 采用定长//简单；**

**Value采用自描述结构组织；**

设计：

**MemHeader**

Header header//4096字节

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| CreateTime | Revision | MiCkSize | ChunkNum | VirginCkIndex | FreeCkIndex0 |
| UsedCkNum | MaxKeyNum | KeyNum | MaxKeyLen | KeyType | KeyHbNum |
| GetCnt | GetHitCnt | AddCnt | WahCnt | AbortCnt | MKeySearchN |
| AKeySearchN | MaChunkE | MaCkSize | MidSize | FreeCkIndex1 | FreeCkIndex2 |
| KeyFreeList | VirginKyIndex | Verify |  |  |  |

注：avrage search = AllKeySearchN / GetHitCnt

ViriginckIndex 和 VirginKyIndex 都利用verifycode来校验;

KeyHashtable

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

内存布局

|  |  |  |
| --- | --- | --- |
| LruHead | HashBuckts | KeyNode |

Struct keyNode {

T \_key;

int \_ck\_index; char \_ck\_type;

};

MemHeader 放在一块共享内存里面；

MemData 放另外一块共享内存；

这样方便key或者data数目扩展；

**MemData**

一个value包括：value，struct；一个 mem mgr；

Mem分配原则：一次向mem data申请分配一个chunk，当chunk用完再向memdata申请；各chunk用链表串联；

|  |  |  |
| --- | --- | --- |
| Verify | FailedFlag (4) | Version(4) |
| FreeIndex | ChunkNum | UsedMem |
| Struct | ChunkVector | ChunkType |

Struct 包括：

String, vector, map, list,还有最重要的MemPtr;

Struct MemPtr {

u\_int:10 chunk\_uno; //index, max=1024

u\_int:11 offset;//按4对齐 max cksize=8k(0<offset<2048)

u\_int:8 padding0; u\_int:3 padding1;

};

Union String {

Struct {

MemPtr data\_ptr;

u\_int:11 calloc\_len; //按4对齐u\_int:13 str\_len; u\_int:8 padding0;

};

Char str[8];//support <=7 bytes string;

}

采用3段data共享内存，如第一段chunksize=1024;第二段4096；第三段8k

最大单位为：8M

Chunk分配流程：首先请求第一段，如果没有，则第二段，否则第三段；如果第一段有，则用之；继续用，如果分配用完，则类似calloc，如果申请到更大内存，直接memcpy即可…

Mem pool分配流程：检查某chunk剩余内存，如果大于需要，则分配，否则下一个chunk；

采用slab策略，可以让客户选择几种常见大小类型；

**内存gc**

Ck 内部gc，当扫描chunk 内部时候，只需要检查chunk的内存使用效率；如果小于某一阀值，则对此chunk开始gc；

Gc线程先检查需要何种gc

如果是压缩整个value，则直接复制，然后释放旧的即可；

否则进入较复杂的chunk gc；

chunk gc先将需要gc的chunk index取出来。然后递归扫描整个value（其实value data有点像用指针串起来的一颗树），如果有指针指向target gc chunk，则将值复制出来到新的压缩chunk，target gc chunk used mem 相应减法（注意有可能有多个指针指向同一个位置，所以一定要注意去重）；

最后释放整个gc chunks（释放过程汇需要验证used mem是否==0，否则就是有异常）

**内存保护**

1. MemPtr 的chunk\_no 和 offset 可以保护指针；
2. 采用 mprotect 函数将大部分内存置为可读，只有在需要写的时候，才打开value所对应的页；
3. Verify 用来验证 chunkVector等头部值；

**事务**

主要目的用于：

确保程序如果中途abort能被检测出，然后将未能预测的相关cache数据清除；如果正常退出，则做一些清理工作（比如关闭页表写权限）

设计:

|  |  |  |  |
| --- | --- | --- | --- |
| Key\_offset0 | Key\_offset1 | Key\_offset2 | Key\_offset3 |

采用文件锁，如果进程中途abort掉，则锁自动释放；

Cache独立维护进程：

内存cache由独立进程维护，包括：

1. 统计输出
2. 内存gc
3. 事务检测监控

**容错**

free\_list 错乱：将free list置空，然后遍历所有chunks，将free的节点挂到free list上去；

hash list 错乱：将hash list置空，然后遍历所有的key nodes，将keynodes挂到hash list里面去；

**持久化**

如果每天保存一次镜像，然后记录每次写操作涉及的key；那么就可以很牛逼的做成一个带持久化的cache系统；

**压缩，传输**

压缩可以节省内存，传输可以直接用socket传递内存数据结构，从而不再需要编解码过程；