# 段式偏移数据架构

## 需求方案分析

**需求：**

1. 一种适合于各种介质的通用的数据结构，比如，自有内存，共享内存，硬盘，管道，socket 等等，即数据在内存和硬盘以及其他介质中的表示一样，这样就不需要任何编解码而保证数据在各类介质之中通畅流通；
2. 业务数据建立在1上面，且有自己完全独立的环境，包括：独有的内存管理，独有的内存页面（和其他业务数据的内存页面完全分开）；当然这些业务数据肯定都会稍微有点大，否则就有点牛刀来杀鸡了；

**需求分析：**

1. 为满足第一条需求，自然内存使用的指针一定要改成偏移值，即用偏移值来代替指针；这个其实在以前我写的共享内存数据结构中经常用，没什么好稀奇的；
2. 为满足第二条需求，就得费点工夫了，实际上就是要为每个业务数据保存一套完整的资源分配管理系统，每个数据都可以类比成操作系统的每个程序，都有自己的内存系统，都占有一定的系统内存页面，当需要内存的时候，向系统申请大块内存，回收的内存不会一般不会返回给系统，保留起来，下次有需要，可以先向已有空暇内存申请；

说的有点复杂，其实挺简单的；

**上述方案顺便带来的好处：**

1. 内存管理反倒简单了，不会有什么内存泄漏；回收某个业务数据，只是简单的将

其所占有的内存页面一一释放即可；就好比操作系统管理进程资源一样，即使进程内存泄漏，最终进程一结束，资源肯定全回收；

1. 另稍加一些代码，就可以带来非常强壮和健壮的共享内存设计；
2. 如果业务代码有bug，读写到了不属于自己的错误的内存，会导致共享内存烂掉；而程序内存出问题还是比较常见的，所以很难确保共享内存什么时候就被破坏掉了，当程序突然core掉gdb发现的堆栈已经乱七八糟了；这时候不得不清除整个共享内存；
3. 另外即使业务代码没有bug，也有各种可能的问题；比如进程被直接kill或者掉电之类的；都有可能中断内存操作，导致共享内存处于未定义的中间状态；

而如果采用现有方案加上一点点改动就可以完全解决此类问题：

对于第一种情况因为各业务数据的内存页面是完全独立，所以可以采用内存保护来

授权可以访问的内存，这时候错误的内存读写将导致程序core掉，既保证

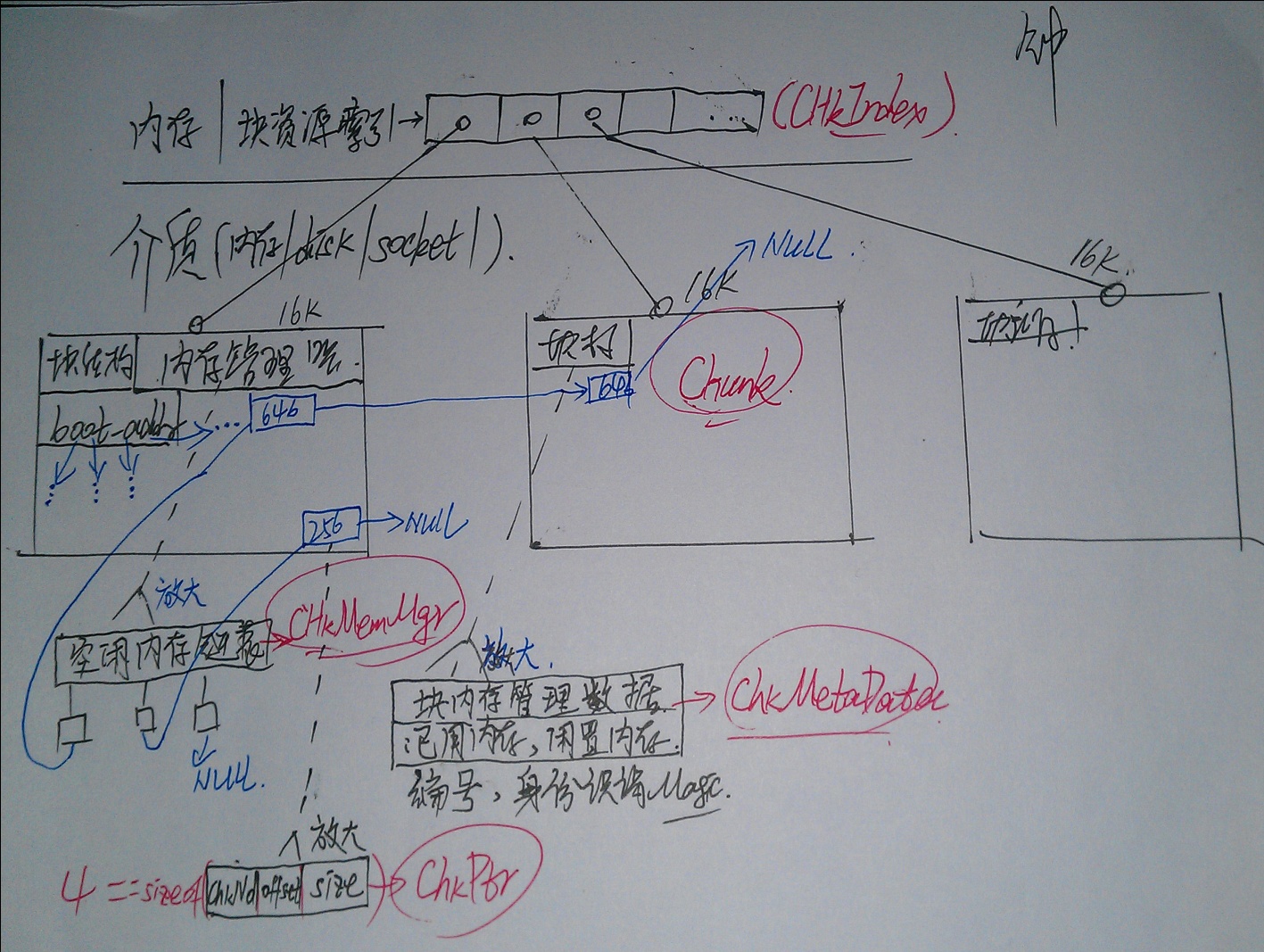
了bu的立即现行；也保证了共享内存的安全；

而对于第二种情况则完全可以通过一定的日志来发现此类操作，对于处于中间状态

未定义的业务数据，完全可以直接从内存中直接清除即可，而清除数据是非常

简单的；

## 架构设计图：

****

**注：**

1. 红色字体标志了关键的数据结构，下面会有详细说明
2. 蓝色代表指针指向
3. 虚线表示放大

## 块资源(内存)管理

**ChkIndex**

{

uint16\_t \_chk\_num; //总共多少个 chunk

uint16\_t \_mem\_alloc\_type:2; //暂时都用1

uint16\_t \_compressed:1; //是否压缩

uint16\_t \_padding1:13;

uint16\_t \_chk\_addr\_size; //已分配的chunk adrr 索引大小

ChkMetadata\*\* \_chk\_addr; //chunk addr 索引块

｝

**ChkMemData**

{

uint32\_t \_magic; //魔幻数,用于识别chunk块

uint32\_t \_capacity:5;//max 31 //此chunk容量 = 4096 \* (1<<\_capacity)

uint32\_t \_padding:3;

uint32\_t \_used:12; //此 chunk 已用内存大小

uint32\_t \_virgin:12; //此 chunk 未初始化内存偏移量

uint32\_t \_no:11; //此 chunk 编号，顺序号

uint32\_t \_alloc\_capacity:16; //用于压缩的内部数据

}

**ChkMemMgr**

{

uint32\_t \_total\_used; //总共使用内存大小

uint32\_t \_total\_free; //总共空闲内存大小

uint16\_t \_total\_capacity; //总共容量

uint16\_t \_free\_0:4;//max=(16\*8 = 128) //第一种空闲内存链表块大小

uint16\_t \_free\_1:5;//max=(32\*8 = 256) //..

uint16\_t \_free\_2:7;//max=(128\*8= 1024)

ChkSList \_free\_list[3]; //空闲内存链表，共3级

｝

注：

1. 在代码实现中，每块内存都是直接分配 16K
2. 每块内存都使用前固定数目（现在是16）字节来保存块自身的管理数据，包括：内存管理，大小管理，身份识别等等；
3. 第一块内存上面比较特殊，初始化就会建立ChkMemMgr 这一关键结构，另外也包括业务数据的boot 数据等等…
4. 资源分配有些特点：比如最小分配单位为8，且以8为倍数分配；按8对齐；
5. 块分配必须按页对齐，即分配出来的块起始地址必须为 4096 的倍数

## 数据结构

**MemPtr** //特殊指针

{

u\_int:11 chunk\_uno; //index, max=2048，为所指目标所在的块编号

u\_int:12 offset;//按4对齐 max cksize=16k(0<offset<4096)，为所指目标在其块内之偏移

uint32\_t size:9; //按 8 对齐，为所指目标的大小，最大为 4096

};

**MemPtr**特点：

1. 通过 chk\_no+offset完整的表达了所指向内存的具体位置，结合**ChkIndex**的**块索引**可以定位到目标；
2. 通过 size 表达了所指目标的完整大小；这个和linux的malloc有类似的功效，析构的时候需要这个大小；
3. 这个指针所能指向的目标范围是有限的，且目标的大小也是有限的，**ChkMetadata** 和 **ChkIndex** 的能力大小均受限于这个结构

**ChkStr** //字符串

{

Union {

Struct {

MemPtr data\_ptr;

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

} chk\_data;

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

}

}

**ChkStr**特点：

1. 大小为8个字节，包括了完整的分配内存，已用内存，字符串长度(MemPtr.size)
2. 最大长度为指针最大指向能力即 < 4096
3. 优化：当字符串内存小于8字节的时候，直接用本身的空间来存储字符串，而不需要分配额外的空间；

**ChkSlistNode** { //单向链表

ChkPtr next;

};

**ChklistNode** {//双向链表

ChkPtr next;

ChkPtr prev;

};

**ChkHashSList** // 单链表 hashList

{

union {

ChkSlistNode list;

ChkPtr buckets;

};

static const int MAX\_BUCKETS = 4096 / sizeof(ChkSlistNode);

}

**ChkHashList** 特点：

1. Hash 桶的最大个数为 4096/4 = 1024
2. 当hash通个数为1的时候，直接用本身来存储hash桶的结构，而不需要另外分配保存hash桶；即当hash桶为1个时候，直接退化成一个单链表；

**ChkVector** // 向量

TODO…

**压缩，传输**

1. 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；
2. 另外从现在已经实现的情况来看，一次分配 16K 的内存，对于偏小的业务数据是严重浪费内存的；所以现在已经实现了压缩，且在压缩的时候，会将未用的内存排除在外，所以对于小业务数据来说，压缩作用非常明显；同样，对于大业务数据来说，压缩效果也会比较明显，特别是如果 ChkStr 用的比较多的话;