Nginx内存池刨析(高并发实现高效内存管理)

高并发下传统方式的弊端

```
void *malloc(size_t size);
void *calloc(size_t nmemb, size_t size);
void *realloc(void *ptr, size_t size);
void free(void *ptr);
```

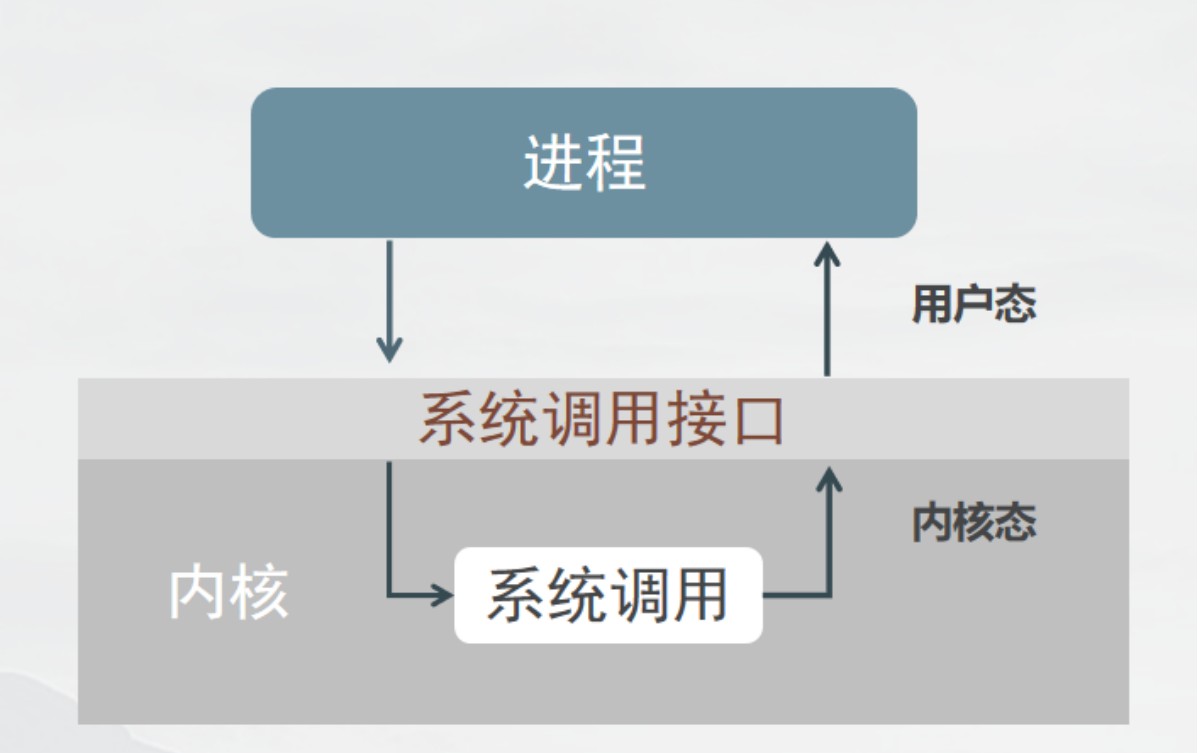
> malloc 在动态的内存储存区中分配一块长度为size字节的连续区域，返回该区域的首地址

> calloc 与malloc相似，参数size为申请地址的单位元素长度，nmemb为元素个数，即在内存中申请nmemb*size字节大小的连续地址空间

> realloc 给一个已经分配了地址的指针重新分配空间,参数ptr为原有的空间地址,newsize是重新申请的地址长度.ptr 若为NULL,它就等同于 malloc.

弊端一

高并发时较小内存块使用导致系统调用频繁，降低了系统的执行效率



弊端二

频繁使用时增加了系统内存的碎片，降低内存使用效率。

内部碎片 - 已经被分配出去（能明确指出属于哪个进程）却不能被利用的内存空间；

产生根源：

1. 内存分配必须起始于可被 4、8 或 16 整除（视处理器体系结构而定）的地址
2. MMU的分页机制的限制

处理器	页大小	分页的级别	虚拟地址分级
x86	4KB	2	10+10+12
x86(extended)	4KB	1	10+22
x86(PAE)	4KB	3	2+9+9+12
x86-64	4KB	4	9+9+9+9+12

弊端三

没有垃圾回收机制，容易造成内存泄漏，导致内存枯竭

demo1

```
void log_error(char *reason) {
    char *p1;
    p1 = malloc(100);
    sprintf(p1,"The f1 error occurred because of '%s'.", reason);
    log(p1);
}
```

demo2

```
int getkey(char *filename) {
    FILE *fp;
    int key;
    fp = fopen(filename, "r");
    fscanf(fp, "%d", &key);
    //fclose(fp);
    return key;
}
```

弊端四

内存分配与释放内存的逻辑在程序中相隔较远时，降低程序的稳定性

A

```
 ret get_stu_info(  Student  * _stu  ) {
    char  * name= NULL;
    name = getName(_stu->no);
    //处理逻辑
    if(name) {
        free(name);
        name = NULL;
    }
}
```

B

```
char stu_name[MAX];

char * getName(int stu_no){

    //查找相应的学号并赋值给 stu_name
     snprintf(stu_name,MAX,“%s”,name);
     return stu_name;
}
```

弊端解决之道

内存管理维度分析

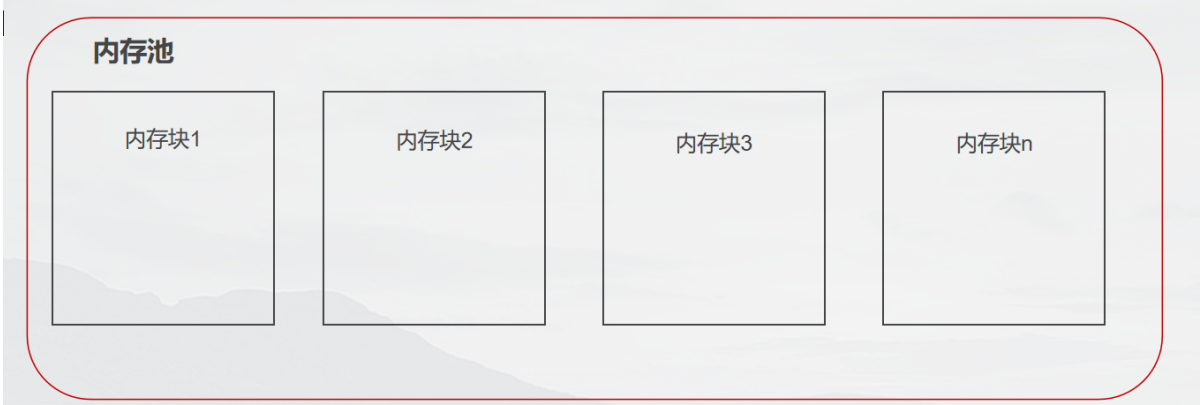
	PtMalloc(glibc 自带)	TcMalloc	JeMalloc
概念	Glibc 自带	Google 开源	Jason Evans（FreeBSD 著名开发人员）
性能(一次 malloc/free 操作)	300ns	50ns	<=50ns
弊端	锁机制降低性能，容易导致内存碎片	1%左右的额外内存开销	2%左右的额外内存开销
优点	传统，稳定	线程本地缓存，多线程分配效率高	线程本地缓存，多核多线程分配效率相当高
使用方式	Glibc 编译	动态链接库	动态链接库
谁在用	较普遍	safari、chrome等	facebook、firefox 等

	PtMalloc(glibc 自带)	TcMalloc	JeMalloc
适用场景	除特别追求高效内存分配以外的	多线程下高效内存分配	多线程下高效内存分配

高并发内存管理最佳实践

什么是内存池技术

在真正使用内存之前，先申请分配一定数量的、大小相等(一般情况下)的内存块留作备用。当有新的内存需求时，就从内存池中分出一部分内存块，若内存块不够再继续申请新的内存，统一对程序所使用的内存进行统一的分配和回收。这样做的一个显著优点是，使得内存分配效率得到很大的提升。



内存池解决之道

> ？高并发时系统调用频繁，降低了系统的执行效率
>
> - 内存池提前预先分配大块内存，统一释放，极大的减少了malloc 和 free 等函数的调用。

> ？频繁使用时增加了系统内存的碎片，降低内存使用效率
>
> - 内存池每次请求分配大小适度的内存块，避免了碎片的产生

> ？没有垃圾回收机制，容易造成内存泄漏
>
> - 在生命周期结束后统一释放内存，完全避免了内存泄露的产生

> ？内存分配与释放的逻辑在程序中相隔较远时，降低程序的稳定性
>
> - 在生命周期结束后统一释放内存，避免重复释放指针或释放空指针等情况

高并发时内存池如何实现

> 高并发（High Concurrency）是互联网分布式系统架构设计中必须考虑的因素之一，它通常是指，通过设计保证系统能够同时并行处理很多请求。

> 高并发特点?
>
> 响应时间、短吞吐量大、每秒响应请求数 QPS、并发用户数高：

内存池设计考虑

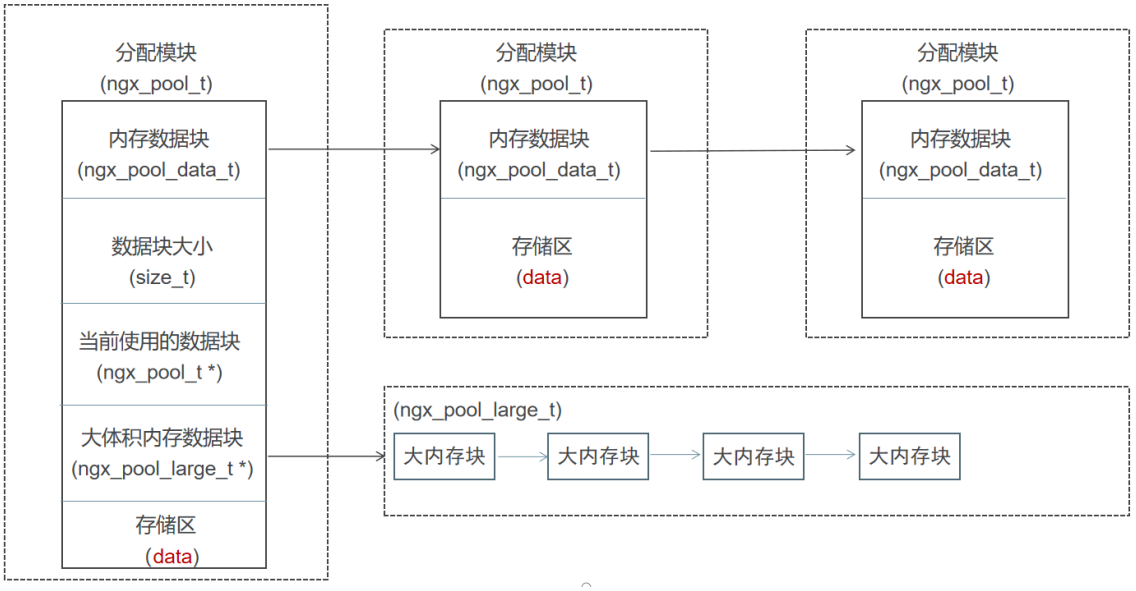
> - 设计逻辑应该尽量简单，避免不同请求之间互相影响，尽量降低不同模块之间的耦合
> - 内存池生存时间应该尽可能短，与请求或者连接具有相同的周期，减少碎片堆积和内存泄漏

高并发内存池涉及和实现

实现思想（分治）

> 对于每个请求或者连接都会建立相应的内存池，建立好内存池之后，我们可以直接从内存池中申请所需要的内存，不用去管内存的释放，当内存池使用完成之后一次性销毁内存池。

> 区分大小内存块的申请和释放，大于池尺寸的定义为大内存块，使用单独的大内存块链表保存，即时分配和释放；小于等于池尺寸的定义为小内存块，直接从预先分配的内存块中提取，不够就扩充池中的内存，在生命周期内对小块内存不做释放，直到最后统一销毁。

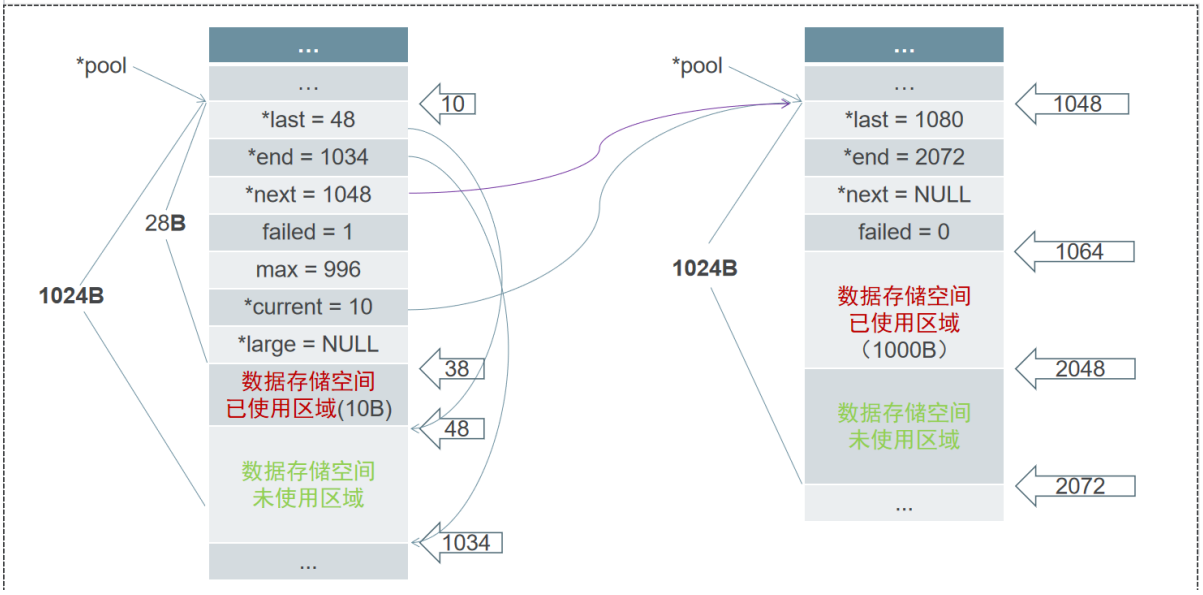


关键数据结构

```
typedef struct {
    u_char               *last;          // 保存当前数据块中内存分配指针的当前位置
    u_char               *end;           // 保存内存块的结束位置
    ngx_pool_t           *next;       // 内存池由多块内存块组成，指向下一个数据块的位置
    ngx_uint_t            failed;        // 当前数据块内存不足引起分配失败的次数
} ngx_pool_data_t;
```

```
struct ngx_pool_s {
    ngx_pool_data_t       d;         // 内存池当前的数据区指针的结构体
    size_t                max;       // 当前数据块最大可分配的内存大小（Bytes）
    ngx_pool_t           *current;   // 当前正在使用的数据块的指针
    ngx_pool_large_t     *large;     // pool 中指向大数据块的指针（大数据快是指 size >
max 的数据块）
};
```

ngx_pool_t 结构示意图 （大小为1024的池）



内存池基本API

内存池创建、销毁和重置：

操作	函数
创建内存池	ngx_pool_t * ngx_create_pool(size_t size);
销毁内存池	void ngx_destroy_pool(ngx_pool_t *pool);
重置内存池	void ngx_reset_pool(ngx_pool_t *pool);

内存池申请、释放和回收操作：

操作	函数
内存申请（对齐）	void * ngx_palloc(ngx_pool_t *pool, size_t size);
内存申请（不对齐）	void * ngx_pnalloc(ngx_pool_t *pool, size_t size);
内存申请（对齐并初始化）	void * ngx_pcalloc(ngx_pool_t *pool, size_t size);
内存清除	ngx_int_t ngx_pfree(ngx_pool_t *pool, void *p);

操作	函数
内存申请（对齐）	void * ngx_palloc(ngx_pool_t *pool, size_t size);
内存申请（不对齐）	void * ngx_pnalloc(ngx_pool_t *pool, size_t size);
内存申请（对齐并初始化）	void * ngx_pcalloc(ngx_pool_t *pool, size_t size);
内存清除	ngx_int_t ngx_pfree(ngx_pool_t *pool, void *p);

源码部分

不见 了