[引擎开发] 深入C++内存管理

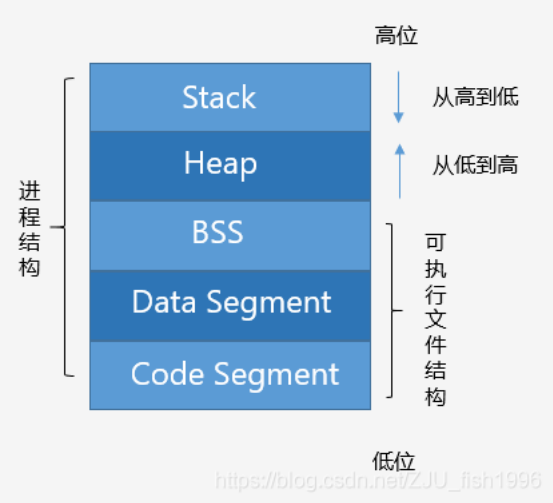
引入

说到C++的内存管理，我们可能会想到栈空间的本地变量、堆上通过new动态分配的变量以及全局命名空间的变量等，这些变量的分配位置都是由系统来控制管理的，而调用者只需要考虑**变量的生命周期**相关内容即可，而无需关心变量的**具体布局**。这对于普通软件的开发已经足够，但对于引擎开发而言，我们必须对内存有着更为精细的管理。

基础概念

在文章的开篇，先对一些基础概念进行简单的介绍，以便能够更好地理解后续的内容。

内存布局



内存分布（可执行映像）

如图，描述了C++程序的内存分布。

Code Segment（代码区）

也称Text Segment，存放可执行程序的机器码。

Data Segment (数据区)

存放已初始化的全局和静态变量， 常量数据（如字符串常量）。

BSS（Block started by symbol)

存放未初始化的全局和静态变量。（默认设为0）

Heap（堆）

从低地址向高地址增长。容量大于栈，程序中动态分配的内存在此区域。

Stack（栈）

从高地址向低地址增长。由编译器自动管理分配。程序中的局部变量、函数参数值、返回变量等存在此区域。

函数栈

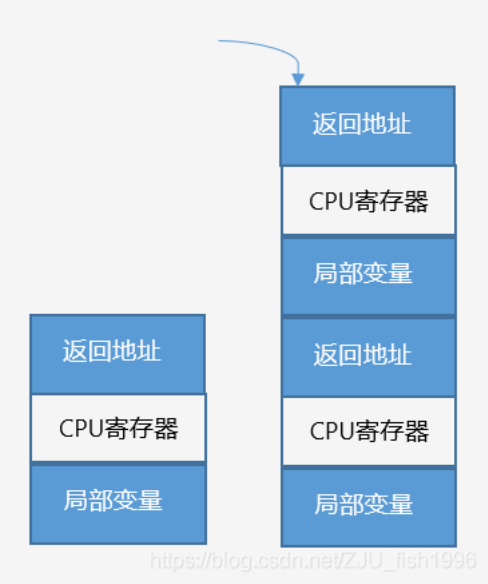
如上图所示，可执行程序的文件包含BSS，Data Segment和Code Segment，当可执行程序载入内存后，系统会保留一些空间，即堆区和栈区。堆区主要是动态分配的内存（默认情况下），而栈区主要是函数以及局部变量等（包括main函数）。一般而言，栈的空间小于堆的空间。

当调用函数时，一块连续内存(堆栈帧）压入栈；函数返回时，堆栈帧弹出。

堆栈帧包含如下数据:

① 函数返回地址

② 局部变量/CPU寄存器数据备份



函数压栈

全局变量

当全局/静态变量（如下代码中的x和y变量）未初始化的时候，它们记录在BSS段。

```
1. int x;
2. int z = 5;
3. void func()
4. {
5.      static int y;
6. }
7. int main()
8. {
9.     return 0;
10. }
```

处于BSS段的变量的值默认为0，考虑到这一点，BSS段内部无需存储大量的零值，而只需记录字节个数即可。

系统载入可执行程序后，将BSS段的数据载入数据段(Data Segment） ，并将内存初始化为0，再调用程序入口（main函数）。

而对于已经初始化了的全局/静态变量而言，如以上代码中的z变量，则一直存储于数据段(Data Segment)。

内存对齐

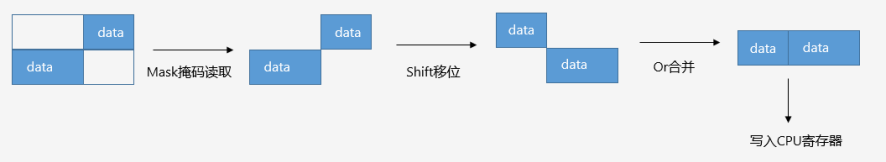
对于基础类型，如float, double, int, char等，它们的大小和内存占用是一致的。而对于结构体而言，如果我们取得其sizeof的结果，会发现这个值有可能会大于结构体内所有成员大小的总和，这是由于结构体内部成员进行了内存对齐。

为什么要进行内存对齐

① 内存对齐使数据读取更高效

在硬件设计上，数据读取的处理器只能从地址为k的倍数的内存处开始读取数据。这种读取方式相当于将内存分为了多个"块“，假设内存可以从任意位置开始存放的话，数据很可能会被分散到多个“块”中，处理分散在多个块中的数据需要移除首尾不需要的字节，再进行合并，非常耗时。

为了提高数据读取的效率，程序分配的内存并不是连续存储的，而是按首地址为k的倍数的方式存储；这样就可以一次性读取数据，而不需要额外的操作。



读取非对齐内存的过程示例

② 在某些平台下，不进行内存对齐会崩溃

内存对齐的规则

定义有效对齐值（alignment）为结构体中 最宽成员 和 编译器/用户指定对齐值 中较小的那个。

(1) 结构体起始地址为有效对齐值的整数倍

(2) 结构体总大小为有效对齐值的整数倍

(3) 结构体第一个成员偏移值为0，之后成员的偏移值为 min(有效对齐值, 自身大小) 的整数倍

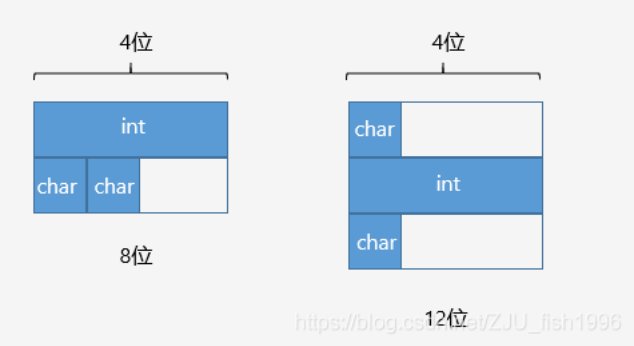
相当于每个成员要进行对齐，并且整个结构体也需要进行对齐。

示例

```
struct A
{
    int i;
    char c1;
    char c2;
};

struct B
{
    char c1;
    int i;
    char c2;
};

int main()
{
    cout << sizeof(A) << endl; // 有效对齐值为4, output : 8
    cout << sizeof(B) << endl; // 有效对齐值为4, output : 12
    return 0;
}
```



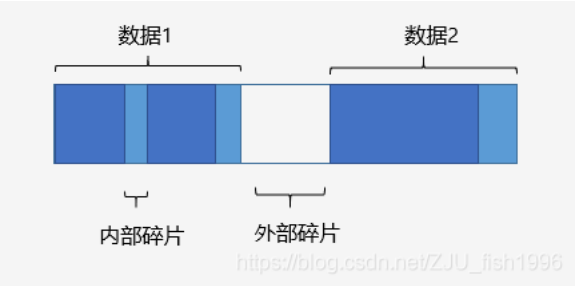
内存排布示例

内存碎片

程序的内存往往不是紧凑连续排布的，而是存在着许多碎片。我们根据碎片产生的原因把碎片分为内部碎片和外部碎片两种类型：

(1) 内部碎片：系统分配的内存大于实际所需的内存（由于对齐机制）；

(2) 外部碎片：不断分配回收不同大小的内存，由于内存分布散乱，较大内存无法分配；



内部碎片和外部碎片

为了提高内存的利用率，我们有必要减少内存碎片，具体的方案将在后文重点介绍。

继承类布局

继承

如果一个类继承自另一个类，那么它自身的数据位于父类之后。

含虚函数的类

如果当前类包含虚函数，则会在类的最前端占用4个字节，用于存储虚表指针（vpointer)，它指向一个虚函数表（vtable)。

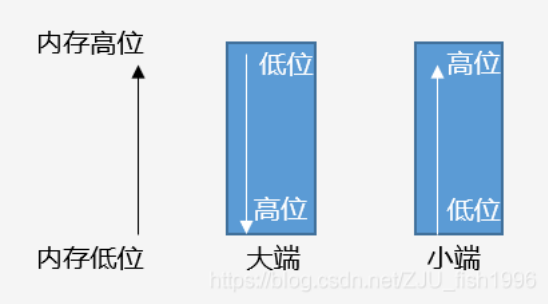
vtable中包含当前类的所有虚函数指针。

字节序（endianness)

大于一个字节的值被称为多字节量，多字节量存在高位有效字节和低位有效字节 (关于高位和低位，我们以十进制的数字来举例，对于数字482来说，4是高位，2是低位），微处理器有两种不同的顺序处理高位和低位字节的顺序：

- 小端（little_endian)：低位有效字节存储于较低的内存位置
- 大端（big_endian)：高位有效字节存储于较低的内存位置

我们使用的PC开发机默认是小端存储。



大小端排布

一般情况下，多字节量的排列顺序对编码没有影响。但如果要考虑跨平台的一些操作，就有必要考虑到大小端的问题。如下图，ue4引擎使用了PLATFORM_LITTLE_ENDIAN这一宏，在不同平台下对数据做特殊处理（内存排布交换，确保存储时的结果一致）。

```
81      // General byte swapping.
82      #if PLATFORM_LITTLE_ENDIAN
83          #define INTEL_ORDER16(x)    (x)
84          #define INTEL_ORDER32(x)    (x)
85          #define INTEL_ORDERF(x)     (x)
86          #define INTEL_ORDER64(x)    (x)
87          #define INTEL_ORDER_TCHARARRAY(x)
88          #define NETWORK_ORDER16(x)          BYTESWAP_ORDER16(x)
89          #define NETWORK_ORDER32(x)          BYTESWAP_ORDER32(x)
90          #define NETWORK_ORDERF(x)           BYTESWAP_ORDERF(x)
91          #define NETWORK_ORDER64(x)          BYTESWAP_ORDER64(x)
92          #define NETWORK_ORDER_TCHARARRAY(x) BYTESWAP_ORDER_TCHARARRAY(x)
93      #else
94          #define INTEL_ORDER16(x)            BYTESWAP_ORDER16(x)
95          #define INTEL_ORDER32(x)            BYTESWAP_ORDER32(x)
96          #define INTEL_ORDERF(x)             BYTESWAP_ORDERF(x)
97          #define INTEL_ORDER64(x)            BYTESWAP_ORDER64(x)
98          #define INTEL_ORDER_TCHARARRAY(x)   BYTESWAP_ORDER_TCHARARRAY(x)
99          #define NETWORK_ORDER16(x)    (x)
100         #define NETWORK_ORDER32(x)    (x)
101         #define NETWORK_ORDERF(x)     (x)
102         #define NETWORK_ORDER64(x)    (x)
103         #define NETWORK_ORDER_TCHARARRAY(x)
104     #endif                                  https://blog.csdn.net/ZJU_fish1996
```

ue4针对大小端对数据做特殊处理（ByteSwap.h)

POD类型

在C++，如果我们的类中包含指针或者容器，由于指针或容器的内存是单独管理的，因此内存一般都不是连续的。

```
1. class A
2. {
3.     std::vector<int> data;
4.     bool bValid;
5.     float* floatData;
6. };
7.
8. int main()
9. {
10.     A a[10];
11.     return 0;
12. }
```

如上，我们定义的数组a内存排布就是不连续的。

我们把那些类型平凡（trivial)，标准布局(standard-layout）的类型，称为POD类型。

类型平凡意味着在对象存储定义的时候就已经完成了对象的生存期，无需构造函数；标准布局意味着它的内存是有序排布的。

在编写程序时，一些频繁引用的仅用于记录信息的数据结构，如果能够设计为POD类型，能够提高程序的效率。

操作系统

对一些基础概念有所了解后，我们可以来关注操作系统底层的一些设计。在掌握了这些特性后，我们才能更好地针对性地编写高性能代码。

SIMD

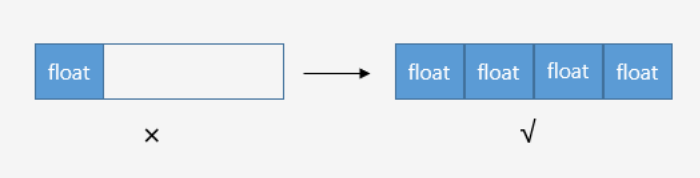
SIMD，即Single Instruction Multiple Data，用一个指令并行地对多个数据进行运算，是CPU基本指令集的扩展。

例一

处理器的寄存器通常是32位或者64位的，而图像的一个像素点可能只有8bit，如果一次只能处理一个数据比较浪费空间；此时可以将64位寄存器拆成8个8位寄存器，就可以并行完成8个操作，提升效率。

例二

SSE指令采用128位寄存器，我们通常将4个32位浮点值打包到128位寄存器中，单个指令可完成4对浮点数的计算，这对于矩阵/向量操作非常友好（除此之外，还有Neon/FPU等寄存器）

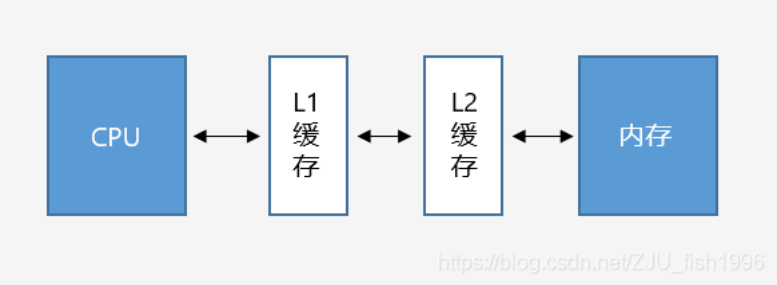


SIMD并行计算

高速缓存

一般来说CPU以超高速运行，而内存速度慢于CPU，硬盘速度慢于内存。

当我们把数据加载内存后，要对数据进行一定操作时，会将数据从内存载入CPU寄存器。考虑到CPU读/写主内存速度较慢，处理器使用了高速的缓存（Cache)，作为内存到CPU中间的媒介。



L1缓存和L2缓存

引入L1和L2缓存后，CPU和内存之间的将无法进行直接的数据交互，而是需要经过两级缓存（目前也已出现L3缓存）。

① CPU请求数据：如果数据已经在缓存中，则直接从缓存载入寄存器；如果数据不在缓存中（缓存命中失败），则需要从内存读取，并将内存载入缓存中。

② CPU写入数据：有两种方案，(1) 写入到缓存时同步写入内存（write through cache) (2) 仅写入到缓存中，有必要时再写入内存(write-back)

为了提高程序性能，则需要尽可能避免缓存命中失败。一般而言，遵循尽可能地集中连续访问内存，减少"跳变"访问的原则（locality of reference)。这里其实隐含了两个意思，一个是内存空间上要尽可能连续，另外一个是访问时序上要尽可能连续。像节点式的数据结构的遍历就会差于内存连续性的容器。

虚拟内存

虚拟内存，也就是把不连续的物理内存块映射到虚拟地址空间（virtual address space)。使内存页对于应用程序来说看起来是连续的。一般而言，出于程序安全性和物理内存可能不足的考虑，我们的程序都会运行在虚拟内存上。

这意味着，每个程序都有自己的地址空间，我们使用的内存存在一个虚拟地址和一个物理地址，两者之间需要进行地址翻译。

缺页

在虚拟内存中，每个程序的地址空间被划分为多个块，每个内存块被称作页，每个页的包含了连续的地址，并且被映射到物理内存。并非所有页都在物理内存中，当我们访问了不在物理内存中的页时，这一现象称为缺页，操作系统会从磁盘将对应内容装载到物理内存；当内存不足，部分页也会写回磁盘。

在这里，我们将CPU，高速缓存和主存视为一个整体，统称为DRAM。由于DRAM与磁盘之间的读写也比较耗时，为了提高程序性能，我们依然需要确保自己的程序具有良好的"局部性"——在任意时刻都在一个较小的活动页面上工作。

分页

当使用虚拟内存时，会通过MMU将虚拟地址映射到物理内存，虚拟内存的内存块称为页，而物理内存中的内存块称为页框，两者大小一致，DRAM和磁盘之间以页为单位进行交换。

简单来说，如果想要从虚拟内存翻译到物理地址，首先会从一个TLB（Translation Lookaside Buffer)的设备中查找，如果找不到，在虚拟地址中也记录了虚拟页号和偏移量，可以先通过虚拟页号找到页框号，再通过偏移量在对应页

框进行偏移，得到物理地址。为了加速这个翻译过程，有时候还会使用多级页表，倒排页表等结构。

置换算法

到目前为止，我们已经接触了不少和“置换”有关的内容：例如寄存器和高速缓存之间，DRAM和磁盘之间，以及TLB的缓存等。这个问题的本质是，我们在有限的空间内存储了一些快速查询的结构，但是我们无法存储所有的数据，所以当查询未命中时，就需要花更大的代价，而所谓置换，也就是我们的快速查询结构是在不断更新的，会随着我们的操作，使得一部分数据被装在到快速查询结构中，又有另一部分数据被卸载，相当于完成了数据的置换。

常见的置换有如下几种：

- 最近未使用置换（NRU）

出现未命中现象时，置换最近一个周期未使用的数据。

- 先入先出置换（FIFO)

出现未命中现象时，置换最早进入的数据。

- 最近最少使用置换（LRU)

出现未命中现象时，置换未使用时间最长的数据。

C++语法

位域（Bit Fields）

表示结构体位域的定义，指定变量所占位数。它通常位于成员变量后，用 声明符：常量表达式 表示。（参考资料）

声明符是可选的，匿名字段可用于填充。

以下是ue4中Float16的定义：

```
1. struct
2. {
3. #if PLATFORM_LITTLE_ENDIAN
4.     uint16      Mantissa : 10;
5.     uint16      Exponent : 5;
6.     uint16      Sign : 1;
7. #else
8.     uint16      Sign : 1;
9.     uint16      Exponent : 5;
10.    uint16      Mantissa : 10;
11. #endif
12. } Components;
```

new和placement new

new是C++中用于动态内存分配的运算符，它主要完成了以下两个操作：

① 调用operator new()函数，动态分配内存。

② 在分配的动态内存块上调用构造函数，以初始化相应类型的对象，并返回首地址。

当我们调用new时，会在堆中查找一个足够大的剩余空间，分配并返回；当我们调用delete时，则会将该内存标记为不再使用，而指针仍然执行原来的内存。

new的语法

```
::(optional) new (placement_params)(optional) ( type ) initializer(optional)
```

- 一般表达式

```
p_var = new type(initializer); // p_var = new type{initializer};
```

● 对象数组表达式

```
1. p_var = new type[size]; // 分配
2. delete[] p_var; // 释放
```

● 二维数组表达式

```
1. auto p = new double[2][2];
2. auto p = new double[2][2]{ {1.0,2.0},{3.0,4.0} };
```

● 不抛出异常的表达式

```
new (nothrow) Type (optional-initializer-expression-list)
```

默认情况下，如果内存分配失败，new运算符会选择抛出std::bad_alloc异常，如果加入nothrow，则不抛出异常，而是返回nullptr。

● 占位符类型

我们可以使用placeholder type（如auto/decltype）指定类型：

```
auto p = new auto('c');
```

● 带位置的表达式（placement new）

可以指定在哪块内存上构造类型。

它的意义在于我们可以利用placement new将内存**分配**和**构造**这两个模块分离（后续的allocator更好地践行了这一概念），这对于编写内存管理的代码非常重要，比如当我们想要编写内存池的代码时，可以预申请一块内存，然后通过placement new申请对象，一方面可以避免频繁调用系统new/delete带来的开销，另一方面可以自己控制内存的分配和释放。

预先分配的缓冲区可以是堆或者栈上的，一般按字节(char)类型来分配，这主要考虑了以下两个原因：

① 方便控制分配的内存大小（通过sizeof计算即可）

② 如果使用自定义类型，则会调用对应的构造函数。但是既然要做分配和构造的分离，我们实际上是不期望它做任何构造操作的，而且对于没有默认构造函数的自定义类型，我们是无法预分配缓冲区的。

以下是一个使用的例子：

```
1. class A
2. {
3. private:
4.         int data;
5. public:
6.         A(int indata)
7.                 : data(indata) { }
8.         void print()
9.         {
10.                cout << data << endl;
11.        }
12. };
13. int main()
14. {
```

```
15.         const int size = 10;
16.         char buf[size * sizeof(A)]; // 内存分配
17.         for (size_t i = 0; i < size; i++)
18.         {
19.                 new (buf + i * sizeof(A)) A(i); // 对象构造
20.         }
21.         A* arr = (A*)buf;
22.         for (size_t i = 0; i < size; i++)
23.         {
24.                 arr[i].print();
25.                 arr[i].~A(); // 对象析构
26.         }
27.         // 栈上预分配的内存自动释放
28.         return 0;
29. }
```

和数组越界访问不一定崩溃类似，这里如果在未分配的内存上执行placement new，可能也不会崩溃。

- 自定义参数的表达式

当我们调用new时，实际上执行了operator new运算符表达式，和其它函数一样，operator new有多种重载，如上文中的placement new，就是operator new以下形式的一个重载：

```
inline void* __CRTDECL operator new(size_t _Size, _Writable_bytes_(_Size) void* _Where) noexcept
{
    (void)_Size;
    return _Where;
}
```

placement new的定义

新语法（C++17）还支持带对齐的operator new：

```
_Ret_notnull_ _Post_writable_byte_size_(_Size)
_VCRT_ALLOCATOR void* __CRTDECL operator new(
    size_t          _Size,
    std::align_val_t _Al
    );

_Ret_maybenull_ _Success_(return != NULL) _Post_writable_byte_size_(_Size)
_VCRT_ALLOCATOR void* __CRTDECL operator new(
    size_t              _Size,
    std::align_val_t    _Al,
    std::nothrow_t const&
    ) noexcept;


_Ret_notnull_ _Post_writable_byte_size_(_Size)
_VCRT_ALLOCATOR void* __CRTDECL operator new[](
    size_t          _Size,
    std::align_val_t _Al
    );

_Ret_maybenull_ _Success_(return != NULL) _Post_writable_byte_size_(_Size)
_VCRT_ALLOCATOR void* __CRTDECL operator new[](
    size_t              _Size,
    std::align_val_t    _Al,
    std::nothrow_t const&
    ) noexcept;
```


aligned new的声明

调用示例：

```
auto p = new(std::align_val_t{ 32 }) A;
```

new的重载

在C++中，我们一般说new和delete动态分配和释放的对象位于自由存储区(free store)，这是一个抽象概念。默认情况下，C++编译器会使用堆实现自由存储。

前文已经提及了new的几种重载，包括数组，placement，align等。

如果我们想要实现自己的内存分配自定义操作，我们可以有如下两个方式：

① 编写重载的operator new，这意味着我们的参数需要和全局operator new有差异。

② 重定义operator new，根据名字查找规则，会优先在申请内存的数据内部/数据定义处查找new运算符，未找到才会调用全局::operator new()。

需要注意的是，如果该全局operator new已经实现为inline函数，则我们不能重定义相关函数，否则无法通过编译，如下：

```
1. // Default placement versions of operator new.
2. inline void* operator new(std::size_t, void* __p) throw() { return __p; }
3. inline void* operator new[](std::size_t, void* __p) throw() { return __p; }
4.
5. // Default placement versions of operator delete.
6. inline void  operator delete  (void*, void*) throw() { }
7. inline void  operator delete[](void*, void*) throw() { }
```

但是，我们可以重写如下nothrow的operator new：

```
1. void* operator new(std::size_t, const std::nothrow_t&) throw();
2. void* operator new[](std::size_t, const std::nothrow_t&) throw();
3. void operator delete(void*, const std::nothrow_t&) throw();
4. void operator delete[](void*, const std::nothrow_t&) throw();
```

为什么说new是低效的

① 一般来说，操作越简单，意味着封装了更多的实现细节。new作为一个通用接口，需要处理任意时间、任意位置申请任意大小内存的请求，它在设计上就无法兼顾一些特殊场景的优化，在管理上也会带来一定开销。

② 系统调用带来的开销。多数操作系统上，申请内存会从用户模式切换到内核模式，当前线程会block住，上下文切换将会消耗一定时间。

③ 分配可能是带锁的。这意味着分配难以并行化。

alignas和alignof

不同的编译器一般都会有默认的对齐量，一般都为2的幂次。

在C中，我们可以通过预编译命令修改对齐量：

```
#pragma pack(n)
```

在内存对齐篇已经提及，我们最终的有效对齐量会取结构体最宽成员 和 编译器默认对齐量（或我们自己定义的对齐量）中较小的那个。

C++中也提供了类似的操作：

alignas

用于指定对齐量。

可以应用于类/结构体/union/枚举的声明/定义；非位域的成员变量的定义；变量的定义（除了函数参数或异常捕获的参数）；

alignas会对对齐量做检查，对齐量不能小于默认对齐，如下面的代码，struct U的对齐设置是错误的：

```
1. struct alignas(8) S
```

```
2. {
3.     // ...
4. };
5.
6. struct alignas(1) U
7. {
8.     S s;
9. };
```

以下对齐设置也是错误的：

```
1. struct alignas(2) S {
2.         int n;
3. };
```

此外，一些错误的格式也无法通过编译，如：

```
struct alignas(3) S { };
```

例子：

```
1. // every object of type sse_t will be aligned to 16-byte boundary
2. struct alignas(16) sse_t
3. {
4.     float sse_data[4];
5. };
6.
7. // the array "cacheline" will be aligned to 128-byte boundary
8. alignas(128)
9. char cacheline[128];
```

alignof operator

返回类型的std::size_t。如果是引用，则返回引用类型的对齐方式，如果是数组，则返回元素类型的对齐方式。

例子：

```
1. struct Foo {
2.     int i;
3.     float f;
4.     char c;
5. };
6.
7. struct Empty { };
8.
9. struct alignas(64) Empty64 { };
10.
```

```
11. int main()
12. {
13.     std::cout << "Alignment of" "\n"
14.                  "- char          :"    << alignof(char)    << "\n"   // 1
15.                  "- pointer       :"    << alignof(int*)    << "\n"   // 8
16.                  "- class Foo     :"    << alignof(Foo)     << "\n"   // 4
17.                  "- empty class   :"    << alignof(Empty)   << "\n"   // 1
18.                  "- alignas(64) Empty:" << alignof(Empty64) << "\n";  // 64
19. }
```


std::max_align_t

一般为16bytes，malloc返回的内存地址，对齐大小不能小于max_align_t。

allocator

当我们使用C++的容器时，我们往往需要提供两个参数，一个是容器的类型，另一个是容器的分配器。其中第二个参数有默认参数，即C++自带的分配器（allocator)：

```
template < class T, class Alloc = allocator<T> > class vector; // generic template
```

我们可以实现自己的allocator，只需实现分配、构造等相关的操作。在此之前，我们需要先对allocator的使用做一定的了解。

new操作将内存分配和对象构造组合在一起，而allocator的意义在于将内存分配和构造分离。这样就可以分配大块内存，而只在真正需要时才执行对象创建操作。

假设我们先申请n个对象，再根据情况逐一给对象赋值，如果内存分配和对象构造不分离可能带来的弊端如下：

① 我们可能会创建一些用不到的对象；

② 对象被赋值两次，一次是默认初始化时，一次是赋值时；

③ 没有默认构造函数的类甚至不能动态分配数组；

使用allocator之后，我们便可以解决上述问题。

分配

为n个string分配内存：

```
1. allocator<string> alloc; // 构造allocator对象
2. auto const p = alloc.allocate(n); // 分配n个未初始化的string
```

构造

在刚才分配的内存上构造两个string:

```
1. auto q = p;
2. alloc.construct(q++, "hello"); // 在分配的内存处创建对象
3. alloc.construct(q++, 10, 'c');
```

销毁

将已构造的string销毁：

```
1. while(q != p)
2.     alloc.destroy(--q);
```

释放

将分配的n个string内存空间释放：

```
alloc.deallocate(p, n);
```

注意：传递给deallocate的指针不能为空，且必须指向由allocate分配的内存，并保证大小参数一致。

拷贝和填充

```
uninitialized_copy(b, e, b2)
// 从迭代器b, e 中的元素拷贝到b2指定的未构造的原始内存中；

uninitialized_copy(b, n, b2)
// 从迭代器b指向的元素开始，拷贝n个元素到b2开始的内存中；

uninitialized_fill(b, e, t)
// 从迭代器b和e指定的原始内存范围中创建对象，对象的值均为t的拷贝；

uninitialized_fill_n(b, n, t)
// 从迭代器b指向的内存地址开始创建n个对象；
```

为什么stl的allocator并不好用

如果仔细观察，我们会发现很多商业引擎都没有使用stl中的容器和分配器，而是自己实现了相应的功能。这意味着allocator无法满足某些引擎开发一些定制化的需求：

① allocator内存对齐无法控制

② allocator难以应用内存池之类的优化机制

③ 绑定模板签名

shared_ptr, unique_ptr和weak_ptr

智能指针是针对裸指针可能出现的问题封装的指针类，它能够更安全、更方便地使用动态内存。

shared_ptr

shared_ptr的主要应用场景是当我们需要在多个类中共享指针时。

多个类共享指针存在这么一个问题：每个类都存储了指针地址的一个拷贝，如果其中一个类删除了这个指针，其它类并不知道这个指针已经失效，此时就会出现野指针的现象。为了解决这一问题，我们可以使用引用指针来计数，仅当检测到引用计数为0时，才主动删除这个数据，以上就是shared_ptr的工作原理。

shared_ptr的基本语法如下：

初始化

```
shared_ptr<int> p = make_shared<int>(42);
```

拷贝和赋值

```
auto p = make_shared<int>(42);
auto r = make_shared<int>(42);
r = q; // 递增q指向的对象，递减r指向的对象
```

只支持直接初始化

由于接受指针参数的构造函数是explicit的，因此不能将指针隐式转换为shared_ptr:

```

```

```
1. shared_ptr<int> p1 = new int(1024); // err
2. shared_ptr<int> p2(new int(1024)); // ok
```

不与普通指针混用

(1) 通过get()函数，我们可以获取原始指针，但我们不应该delete这一指针，也不应该用它赋值/初始化另一个智能指针；

(2) 当我们将原生指针传给shared_ptr后，就应该让shared_ptr接管这一指针，而不再直接操作原生指针。

重新赋值

```
p.reset(new int(1024));
```

共享自己

shared_ptr实际上是在裸指针的基础上加了一个封装，使其具备自我管理的能力，并表现得像一个指针。

我们可能会遇到如下需求：

```
1. class A
2. {
3. public:
4.     void run();
5. };
6.
7. void fun(shared_ptr<A> In)
8. {
9. }
10.
11. void A::run()
12. {
13.     fun(this); // ?
14. }
15.
16. int main()
17. {
18.     shared_ptr<A> data = make_shared<A>();
19.     data->run();
20.     return 0;
21. }
```

有一个函数fun接受类A的shared_ptr。

我们在外层将A封装为智能指针，并调用它的run函数。在run函数内部，我们的本意是将A继续作为智能指针调用fun函数，但在run成员函数内部，类A已经脱离了原有智能指针的管理，它无法获取到与智能指针有关的信息，无法知晓被谁托管。因此这个操作在C++中默认情况下是无效的。

为了能够实现这一点，我们需要添加一些辅助信息，让类A能够获取到它需要的数据：

```
1. class A : public enable_shared_from_this
2. {
```

```
3. public:
4.   void run() {
5.     shared_ptr<A> data = shared_from_this();
6.   }
7. };
```

一个是让A从enable_shared_from_this继承，第二个是调用shared_from_this获取自身的智能指针信息。

unique_ptr

有时候我们会在函数域内临时申请指针，或者在类中声明非共享的指针，但我们很有可能忘记删除这个指针，造成内存泄漏。此时我们可以考虑使用unique_ptr，由名字可见，某一时刻只有一个unique_ptr指向给定的对象，且它会在析构的时候自动释放对应指针的内存。

unique_ptr的基本语法如下：

初始化

```
unique_ptr<string> p = make_unique<string>("test");
```

不支持直接拷贝/赋值

为了确保某一时刻只有一个unique_ptr指向给定对象，unique_ptr不支持普通的拷贝或赋值。

```
1. unique_ptr<string> p1(new string("test"));
2. unique_ptr<string> p2(p1); // err
3. unique_ptr<string> p3;
4. p3 = p2; // err
```

所有权转移

可以通过调用release或reset将指针的所有权在unique_ptr之间转移：

```
1. unique_ptr<string> p2(p1.release());
2. unique_ptr<string> p3(new string("test"));
3. p2.reset(p3.release());
```

不能忽视release返回的结果

release返回的指针通常用来初始化/赋值另一个智能指针，如果我们只调用release，而没有删除其返回值，会造成内存泄漏：

```
1. p2.release(); // err
2. auto p = p2.release(); // ok, but remember to delete(p)
```

支持移动

```
1. unique_ptr<int> clone(int p) {
2.     return unique_ptr<int>(new int(p));
3. }
```

weak_ptr

weak_ptr不控制所指向对象的生存期，即不会影响引用计数。它指向一个shared_ptr管理的对象。通常而言，它的存在有如下两个作用：

(1) 解决循环引用的问题

(2) 作为一个“观察者”：

详细来说，和之前提到的多个类共享内存的例子一样，使用普通指针可能会导致一个类删除了数据后其它类无法同步这一信息，导致野指针；之前我们提出了shared_ptr，也就是每个类记录一个引用，释放时引用数减一，直到减为0才释放。

但在有些情况下，我们并不希望当前类影响到引用计数，而是希望实现这样的逻辑：假设有两个类引用一个数据，其中有一个类将主动控制类的释放，而无需等待另外一个类也释放才真正销毁指针所指对象。对于另一个类而言，它只需要知道这个指针已经失效即可，此时我们就可以使用weak_ptr。

我们可以像如下这样检测weak_ptr所有对象是否有效，并在有效的情况下做相关操作：

```
1. auto p = make_shared<int>(42);
2. weak_ptr<int> wp(p);
3.
4. if(shared_ptr<int> np = wp.lock())
5. {
6.    // ...
7. }
```

分配与管理机制

到目前为止，我们对内存的概念有了初步的了解，也掌握了一些基本的语法。接下来我们要讨论如何进行有效的内存管理。

设计高效的内存分配器通常会考虑到以下几点：

① 尽可能减少内存碎片，提高内存利用率

② 尽可能提高内存的访问局部性

③ 设计在不同场合上适用的内存分配器

④ 考虑到内存对齐

含freelist的分配器

我们首先来考虑一种能够处理任何请求的通用分配器。

一个非常朴素的想法是，对于释放的内存，通过链表将空闲内存链接起来，称为freelist。

分配内存时，先从freelist中查找是否存在满足要求的内存块，如果不存在，再从未分配内存中获取；当我们找到合适的内存块后，分配合适的内存，并将多余的部分放回freelist。

释放内存时，将内存插入到空闲链表，可能的话，合并前后内存块。

其中，有一些细节问题值得考虑：

① 空闲空间应该如何进行管理？

我们知道freelist是用于管理空闲内存的，但是freelist本身的存储也需要占用内存。我们可以按如下两种方式存储freelist:

● 隐式空闲链表

将空闲链表信息与内存块存储在一起。主要记录大小，已分配位等信息。

● 显式空闲链表

单独维护一块空间来记录所有空闲块信息。

● 分离适配（segregated-freelist）

将不同大小的内存块放在一起容易造成外部碎片，可以设置多个freelist，并让每个freelist存储不同大小的内存块，申请内存时选择满足条件的最小内存块。

● 位图

除了freelist之外，还可以考虑用0，1表示对应内存区域是否已分配，称为位图。

② 分配内存优先分配哪块内存？

一般而言，从策略不同来分，有以下几种常见的分配方式：

- 首次适应(first-fit）:找到的第一个满足大小要求的空闲区
- 最佳适应(best-fit) : 满足大小要求的最小空闲区
- 循环首次适应(next-fit) :在先前停止搜索的地方开始搜索找到的第一个满足大小要求的空闲区

③ 释放内存后如何放置到空闲链表中？

- 直接放回链表头部/尾部
- 按照地址顺序放回

这几种策略本质上都是取舍问题：分配/放回时间复杂度如果低，内存碎片就有可能更多，反之亦然。

buddy分配器

按照一分为二，二分为四的原则，直到分裂出一个满足大小的内存块；合并的时候看buddy是否空闲，如果是就合并。

可以通过位运算直接算出buddy，buddy的buddy，速度较快。但内存碎片较多。

含对齐的分配器

一般而言，对于通用分配器来说，都应当传回对齐的内存块，即根据对齐量，分配比请求多的对齐的内存。

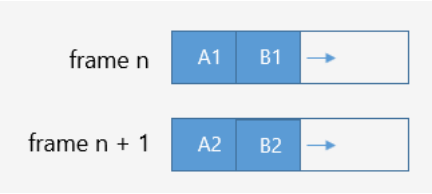
如下，是ue4中计算对齐的方式，它返回和对齐量向上对齐后的值，其中Alignment应为2的幂次。

```
1. template <typename T>
2. FORCEINLINE constexpr T Align(T Val, uint64 Alignment)
3. {
4.         static_assert(TIsIntegral<T>::Value || TIsPointer<T>::Value, "Align
   expects an integer or pointer type");
5.
6.         return (T)(((uint64)Val + Alignment - 1) & ~(Alignment - 1));
7. }
```

其中~（Alignment - 1) 代表的是高位掩码，类似于11110000的格式，它将剔除低位。在对Val进行掩码计算时，加上Alignment - 1的做法类似于（x + a) % a，避免Val值过小得到0的结果。

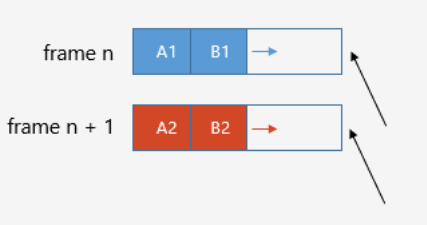
单帧分配器模型

用于分配一些临时的每帧生成的数据。分配的内存仅在当前帧适用，每帧开始时会将上一帧的缓冲数据清除，无需手动释放。



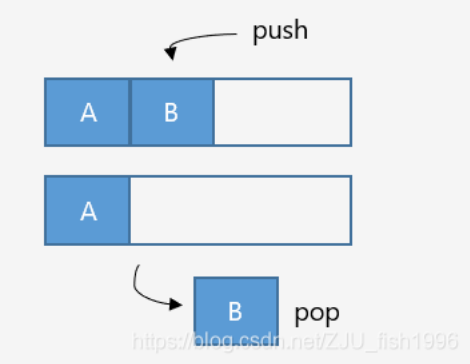
双帧分配器模型

它的基本特点和单帧分配器相近，区别在于第i+1帧适用第i帧分配的内存。它适用于处理非同步的一些数据，避免当前缓冲区被重写（同时读写）



堆栈分配器模型

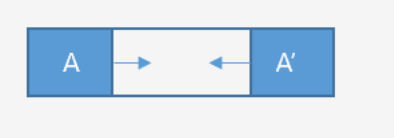
堆栈分配器，它的优点是实现简单，并且完全避免了内存碎片，如前文所述，函数栈的设计也使用了堆栈分配器的模型。



堆栈分配器

双端堆栈分配器模型

可以从两端开始分配内存，分别用于处理不同的事务，能够更充分地利用内存。



双端堆栈分配器

池分配器模型

池分配器可以分配大量同尺寸的小块内存。它的空闲块也是由freelist管理的，但由于每个块的尺寸一致，它的操作复杂度更低，且也不存在内存碎片的问题。

tcmalloc的内存分配

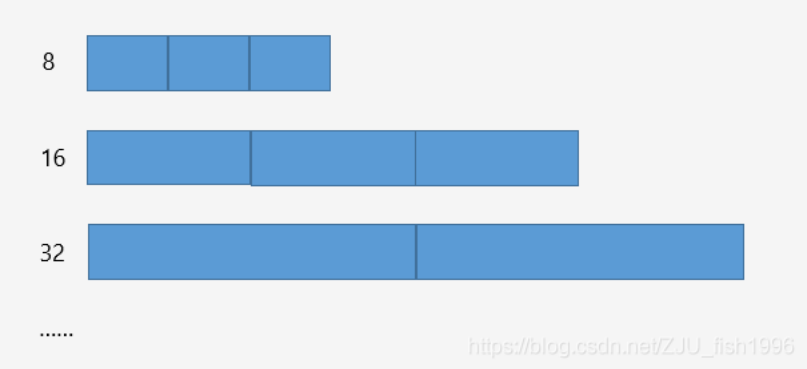
tcmalloc是一个应用比较广泛的内存分配第三方库。

对于大于页结构和小于页结构的内存块申请，tcmalloc分别做不同的处理。

小于页的内存块分配

使用多个内存块定长的freelist进行内存分配，如：8，16，32……，对实际申请的内存向上“取整”。

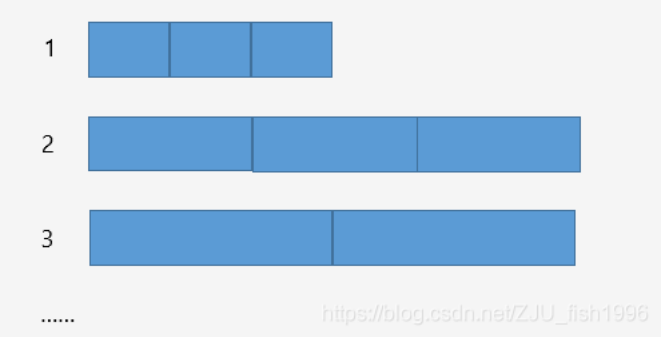
freelist采用隐式存储的方式。



多个定长的freelist

大于页的内存块分配

可以一次申请多个page，多个page构成一个span。同样的，我们使用多个定长的span链表来管理不同大小的span。



多个定长的spanlist

对于不同大小的对象，都有一个对应的内存分配器，称为CentralCache。具体的数据都存储在span内，每个CentralCache维护了对应的spanlist。如果一个span可以存储多个对象，spanlist内部还会维护对应的freelist。

容器的访问局部性

由于操作系统内部存在缓存命中的问题，所以我们需要考虑程序的访问局部性，这个访问局部性实际上有两层意思：

(1) 时间局部性：如果当前数据被访问，那么它将在不久后很可能在此被访问；

(2) 空间局部性：如果当前数据被访问，那么它相邻位置的数据很可能也被访问；

我们来认识一下常用的几种容器的内存布局：

数组/顺序容器：内存连续，访问局部性良好；

map：内部是树状结构，为节点存储，无法保证内存连续性，访问局部性较差（flat_map支持顺序存储）；

链表：初始状态下，如果我们连续顺序插入节点，此时我们认为内存连续，访问较快；但通过多次插入、删除、交换等操作，链表结构变得散乱，访问局部性较差；

碎片整理机制

内存碎片几乎是不可完全避免的，当一个程序运行一定时间后，将会出现越来越多的内存碎片。一个优化的思路就是在引擎底层支持定期地整理内存碎片。

简单来说，碎片整理通过不断的移动操作，使所有的内存块“贴合”在一起。为了处理指针可能失效的问题，可以考虑使用智能指针。

由于内存碎片整理会造成卡顿，我们可以考虑将整理操作分摊到多帧完成。

ue4内存管理

自定义内存管理

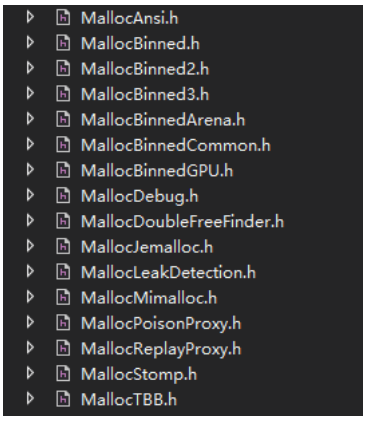
```
/** The global memory allocator. */
CORE_API extern class FMalloc* GMalloc;
CORE_API extern class FMalloc** GFixedMallocLocationPtr;
```

ue4的内存管理主要是通过FMalloc类型的GMalloc这一结构来完成特定的需求，这是一个虚基类，它定义了malloc，realloc，free等一系列常用的内存管理操作。其中，Malloc的两个参数分别是分配内存的大小和对应的对齐量，默认对齐量为0。

```
1. /** The global memory allocator's interface. */
2. class CORE_API FMalloc  :
3.         public FUseSystemMallocForNew,
4.         public FExec
5. {
6. public:
7.         virtual void* Malloc( SIZE_T Count, uint32 Alignment=DEFAULT_ALIGNMENT
   ) = 0;
```

```
8.          virtual void* TryMalloc( SIZE_T Count, uint32
    Alignment=DEFAULT_ALIGNMENT );
9.          virtual void* Realloc( void* Original, SIZE_T Count, uint32
    Alignment=DEFAULT_ALIGNMENT ) = 0;
10.         virtual void* TryRealloc(void* Original, SIZE_T Count, uint32
    Alignment=DEFAULT_ALIGNMENT);
11.         virtual void Free( void* Original ) = 0;
12.
13.         // ...
14. };
```

FMalloc有许多不同的实现，如FMallocBinned，FMallocBinned2等，可以在HAL文件夹下找到相关的头文件和定义，如下：



内部通过枚举量来确定对应使用的Allocator：

```
1.          /** Which allocator is being used */
2.          enum EMemoryAllocatorToUse
3.          {
4.                  Ansi, // Default C allocator
5.                  Stomp, // Allocator to check for memory stomping
6.                  TBB, // Thread Building Blocks malloc
7.                  Jemalloc, // Linux/FreeBSD malloc
8.                  Binned, // Older binned malloc
9.                  Binned2, // Newer binned malloc
10.                 Binned3, // Newer VM-based binned malloc, 64 bit only
11.                 Platform, // Custom platform specific allocator
12.                 Mimalloc, // mimalloc
13.         };
```

对于不同平台而言，都有自己对应的平台内存管理类，它们继承自FGenericPlatformMemory，封装了平台相关的内存操作。具体而言，包含FAndroidPlatformMemory，FApplePlatformMemory，FIOSPlatformMemory，FWindowsPlatformMemory等。

通过调用PlatformMemory的BaseAllocator函数，我们取得平台对应的FMalloc类型，基类默认返回默认的C allocator，而不同平台会有自己特殊的实现。

在PlatformMemory的基础上，为了方便调用，ue4又封装了FMemory类，定义通用内存操作，如在申请内存时，会调用FMemory::Malloc，FMemory内部又会继续调用GMalloc->Malloc。如下为节选代码：

```
struct CORE_API FMemory
{
        /** @name Memory functions (wrapper for FPlatformMemory) */

        static FORCEINLINE void* Memmove( void* Dest, const void* Src, SIZE_T Count )
        {
                return FPlatformMemory::Memmove( Dest, Src, Count );
        }

        static FORCEINLINE int32 Memcmp( const void* Buf1, const void* Buf2, SIZE_T Count )
        {
                return FPlatformMemory::Memcmp( Buf1, Buf2, Count );
        }

        // ...
        static void* Malloc(SIZE_T Count, uint32 Alignment = DEFAULT_ALIGNMENT);
        static void* Realloc(void* Original, SIZE_T Count, uint32 Alignment = DEFAULT_ALIGNMENT);
        static void Free(void* Original);
        static SIZE_T GetAllocSize(void* Original);

    // ...
};
```

为了在调用new/delete能够调用ue4的自定义函数，ue4内部替换了operator new。这一替换是通过IMPLEMENT_MODULE宏引入的：

```
IMPLEMENT_MODULE(FRendererModule, Renderer);
```

IMPLEMENT_MODULE通过定义REPLACEMENT_OPERATOR_NEW_AND_DELETE宏实现替换，如下图所示，operator new/delete内实际调用被替换为FMemory的相关函数。

```
#if !FORCE_ANSI_ALLOCATOR
#define REPLACEMENT_OPERATOR_NEW_AND_DELETE \
    OPERATOR_NEW_MSVC_PRAGMA void* operator new   ( size_t Size                    ) OPERATOR_NEW_THROW_SPEC      { return FMemory::Malloc( Size ); } \
    OPERATOR_NEW_MSVC_PRAGMA void* operator new[] ( size_t Size                    ) OPERATOR_NEW_THROW_SPEC      { return FMemory::Malloc( Size ); } \
    OPERATOR_NEW_MSVC_PRAGMA void* operator new   ( size_t Size, const std::nothrow_t& ) OPERATOR_NEW_NOTHROW_SPEC  { return FMemory::Malloc( Size ); } \
    OPERATOR_NEW_MSVC_PRAGMA void* operator new[] ( size_t Size, const std::nothrow_t& ) OPERATOR_NEW_NOTHROW_SPEC  { return FMemory::Malloc( Size ); } \
    void operator delete   ( void* Ptr )                                           OPERATOR_DELETE_THROW_SPEC     { FMemory::Free( Ptr ); } \
    void operator delete[] ( void* Ptr )                                           OPERATOR_DELETE_THROW_SPEC     { FMemory::Free( Ptr ); } \
    void operator delete   ( void* Ptr, const std::nothrow_t& )                    OPERATOR_DELETE_NOTHROW_SPEC   { FMemory::Free( Ptr ); } \
    void operator delete[] ( void* Ptr, const std::nothrow_t& )                    OPERATOR_DELETE_NOTHROW_SPEC   { FMemory::Free( Ptr ); } \
    void operator delete   ( void* Ptr, size_t Size )                              OPERATOR_DELETE_THROW_SPEC     { FMemory::Free( Ptr ); } \
    void operator delete[] ( void* Ptr, size_t Size )                              OPERATOR_DELETE_THROW_SPEC     { FMemory::Free( Ptr ); } \
    void operator delete   ( void* Ptr, size_t Size, const std::nothrow_t& )       OPERATOR_DELETE_NOTHROW_SPEC   { FMemory::Free( Ptr ); } \
    void operator delete[] ( void* Ptr, size_t Size, const std::nothrow_t& )       OPERATOR_DELETE_NOTHROW_SPEC   { FMemory::Free( Ptr ); }
#else
    #define REPLACEMENT_OPERATOR_NEW_AND_DELETE
#endif
```


FMallocBinned

我们以FMallocBinned为例介绍ue4中通用内存的分配。

基本介绍

（1） 空闲内存如何管理？

FMallocBinned使用freelist机制管理空闲内存。每个空闲块的信息记录在FFreeMem结构中，显式存储。

（2）不同大小内存如何分配？

FMallocBinned使用内存池机制，内部包含POOL_COUNT(42)个内存池和2个扩展的页内存池；其中每个内存池的信息由FPoolInfo结构体维护，记录了当前FreeMem内存块指针等，而特定大小的所有内存池由FPoolTable维护；内存池内包含了内存块的双向链表。

（3）如何快速根据分配元素大小找到对应的内存池？

为了快速查询当前分配内存大小应该对应使用哪个内存池，有两种办法，一种是二分搜索O(logN)，另一种是打表(O1)，考虑到可分配内存数量并不大，MallocBinned选择了打表的方式，将信息记录在MemSizeToPoolTable。

（4）如何快速删除已分配内存？

为了能够在释放的时候以O(1)时间找到对应内存池，FMallocBinned维护了PoolHashBucket结构用于跟踪内存分配的记录。它组织为双向链表形式，存储了对应内存块和键值。

内存池

- 多个小对象内存池（内存池大小均为PageSize，但存储的数据量不一样）。数据块大小设定如下：

```
static const uint32 BlockSizes[POOL_COUNT] =
{
    8,      16,     32,     48,     64,     80,     96,     112,
    128,    160,    192,    224,    256,    288,    320,    384,
    448,    512,    576,    640,    704,    768,    896,    1024,
    1168,   1360,   1632,   2048,   2336,   2720,   3264,   4096,
    4672,   5456,   6544,   8192,   9360,   10912,  13104,  16384,
    21840,  32768
};
```

- 两个额外的页内存池，管理大于一个页的内存池，大小为3*PageSize和6*PageSize
- 操作系统的内存池

分配策略

分配内存的函数为void* FMallocBinned::Malloc(SIZE_T Size, uint32 Alignment)。

其中第一个参数为需要分配的内存的大小，第二个参数为对齐的内存数。

如果用户未指定对齐的内存大小，MallocBinned内部会默认对齐于16字节，如果指定了大于16字节的对齐内存大小，则对齐于用户指定的对齐大小。根据对齐量，计算出最终实际分配的内存大小。

MallocBinned内部对于不同的内存大小有三种不同的处理：

(1) 分配小块内存（0，PAGE_SIZE_LIMIT/2）

根据分配大小从MemSizeToPoolTable中获取对应内存池，并从内存池的当前空闲位置读取一块内存，并移动当前内存指针。如果移动后的内存指针指向的内存块已经使用，则将指针移动到FreeMem链表的下一个元素；如果当前内存池已满，将该内存池移除，并链接到耗尽的内存池。

如果当前内存池已经用尽，下次内存分配时，检测到内存池用尽，会从系统重新申请一块对应大小的内存池。

(2) 分配大块内存 [PAGE_SIZE_LIMIT/2, PAGE_SIZE_LIMIT*3/4]∪(PageSize，PageSize + PAGE_SIZE_LIMIT/2)

需要从额外的页内存池分配，分配方式和（1）一样。

(3) 分配超大内存

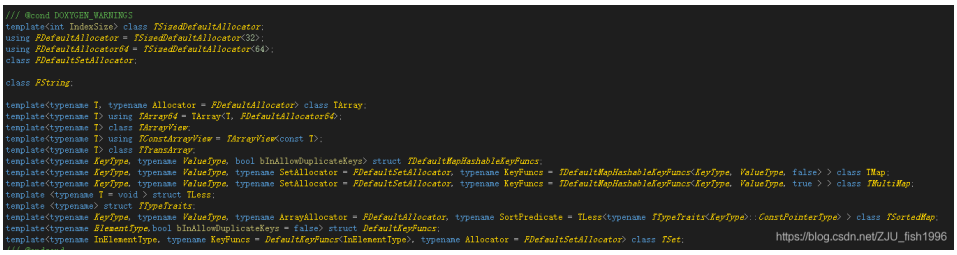
从系统内存池中分配。

Allocator

对于ue4中的容器而言，它的模板有两个参数，第一个是元素类型，第二个就是对应的分配器（Allocator）：

```
1. template<typename InElementType, typename InAllocator>
2. class TArray
3. {
4.     // ...
5. };
```

如下图，容器一般都指定了自己默认的分配器：



默认的堆分配器

```
1. template <int IndexSize>
2. class TSizedHeapAllocator { ... };
3.
4. // Default Allocator
5. using FHeapAllocator = TSizedHeapAllocator<32>;
```

默认情况下，如果我们不指定特定的Allocator，容器会使用大小类型为int32堆分配器，默认由FMemory控制分配（和new一致）

含对齐的分配器

```
1. template<uint32 Alignment = DEFAULT_ALIGNMENT>
2. class TAlignedHeapAllocator
3. {
4.     // ...
5. };
```

由FMemory控制分配，含对齐。

可扩展大小的分配器

```
1. template <uint32 NumInlineElements, typename SecondaryAllocator =
   FDefaultAllocator>
2. class TInlineAllocator
3. {
4.     //...
5. };
```

可扩展大小的分配器存储大小为NumInlineElements的定长数组，当实际存储的元素数量高于NumInlineElements时，会从SecondaryAllocator申请分配内存，默认情况下为堆分配器。

对齐量总为DEFAULT_ALIGNMENT。

不可重定位的可扩展大小的分配器

```
1. template <uint32 NumInlineElements>
2. class TNonRelocatableInlineAllocator
3. {
4.     // ...
5. };
```

在支持第二分配器的基础上，允许第二分配器存储指向内联元素的指针。这意味着Allocator不应做指针重定向的操作。但ue4的Allocator通常依赖于指针重定向，因此该分配器不应用于其它Allocator容器。

固定大小的分配器

```
template <uint32 NumInlineElements>
class TFixedAllocator
{
    // ...
};
```

类似于InlineAllocator，会分配固定大小内存，区别在于当内联存储耗尽后，不会提供额外的分配器。

稀疏数组分配器

```
template<typename InElementAllocator = FDefaultAllocator,typename
InBitArrayAllocator = FDefaultBitArrayAllocator>
class TSparseArrayAllocator
{
public:

        typedef InElementAllocator ElementAllocator;
        typedef InBitArrayAllocator BitArrayAllocator;
};
```

稀疏数组本身的定义比较简单，它主要用于稀疏数组（Sparse Array），相关的操作也在对应数组类中完成。稀疏数组支持不连续的下标索引，通过BitArrayAllocator来控制分配哪个位是可用的，能够以O(1)的时间删除元素。

默认使用堆分配。

哈希分配器

```
template<
        typename InSparseArrayAllocator               =
TSparseArrayAllocator<>,
        typename InHashAllocator                      =
TInlineAllocator<1,FDefaultAllocator>,
        uint32   AverageNumberOfElementsPerHashBucket =
DEFAULT_NUMBER_OF_ELEMENTS_PER_HASH_BUCKET,
        uint32   BaseNumberOfHashBuckets              =
DEFAULT_BASE_NUMBER_OF_HASH_BUCKETS,
        uint32   MinNumberOfHashedElements            =
DEFAULT_MIN_NUMBER_OF_HASHED_ELEMENTS
        >
class TSetAllocator
{
public:
        static FORCEINLINE uint32 GetNumberOfHashBuckets(uint32
NumHashedElements) { //... }

```

```
13.         typedef InSparseArrayAllocator SparseArrayAllocator;
14.         typedef InHashAllocator        HashAllocator;
15. };
```

用于TSet/TMap等结构的哈希分配器，同样的实现比较简单，具体的分配策略在TSet等结构中实现。其中SparseArrayAllocator用于管理Value，HashAllocator用于管理Key。Hash空间不足时，按照2的幂次进行扩展。

默认使用堆分配。

除了使用默认的堆分配器，稀疏数组分配器和哈希分配器都有对应的可扩展大小（InlineAllocator)/固定大小(FixedAllocator)分配版本。

动态内存管理

TSharedPtr

```
1. template< class ObjectType, ESPMode Mode >
2. class TSharedPtr
3. {
4.     // ...
5. private:
6.         ObjectType* Object;
7.         SharedPointerInternals::FSharedReferencer< Mode > SharedReferenceCount;
8. };
```

TSharedPtr是ue4提供的类似stl sharedptr的解决方案，但相比起stl，它可由第二个模板参数控制是否线程安全。

如上所示，它基于类内的引用计数实现(SharedReferenceCount)，为了确保多个TSharedPtr能够同步当前引用计数的信息，引用计数被设计为指针类型。在拷贝/构造/赋值等操作时，会增加或减少引用计数的值，当引用计数为0时将销毁指针所指对象。

TSharedRef

```
1. template< class ObjectType, ESPMode Mode >
2. class TSharedRef
3. {
4.     // ...
5. private:
6.         ObjectType* Object;
7.         SharedPointerInternals::FSharedReferencer< Mode > SharedReferenceCount;
8. };
```

和TSharedPtr类似，但存储的指针不可为空，创建时需同时初始化指针。类似于C++中的引用。

TRefCountPtr

```
1. template<typename ReferencedType>
2. class TRefCountPtr
3. {
4.     // ...
5. private:
```

```
6.         ReferencedType* Reference;
7. };
```

TRefCountPtr是基于引用计数的共享指针的另一种实现。和TSharedPtr的差异在于它的引用计数并非智能指针类内维护的，而是基于对象的，相当于TRefCountPtr内部只存储了对应的指针信息（ReferencedType* Reference)。

基于对象的引用计数，即引用计数存储在对象内部，这是通过从FRefCountBase继承引入的。这也就意味着TRefCountPtr引用的对象必须从FRefCountBase继承，它的使用是有局限性的。

但是在如统计资源引用而判断资源是否需要卸载的应用场景中，TRefCountPtr可手动添加/释放引用，使用上更友好。

```
1. class FRefCountBase
2. {
3. public:
4.     // ...
5. private:
6.         mutable int32 NumRefs = 0;
7. };
```

TWeakPtr

```
1. template< class ObjectType, ESPMode Mode >
2. class TWeakPtr
3. {
4. };
```

类似的，TWeakObjectPtr是ue4提供的类似stl weakptr的解决方案，它将不影响引用计数。

TWeakObjectPtr

```
1. template<class T, class TWeakObjectPtrBase>
2. struct TWeakObjectPtr : private TWeakObjectPtrBase
3. {
4.     // ...
5. };
6.
7. struct FWeakObjectPtr
8. {
9.     // ...
10.
11. private:
12.         int32           ObjectIndex;
13.         int32           ObjectSerialNumber;
14. };
```

特别的，由于UObject有对应的gc机制，TWeakObjectPtr为指向UObject的弱指针，用于查询对象是否有效（是否被回收）

垃圾回收

C++语言本身并没有垃圾回收机制，ue4基于内部的UObject，单独实现了一套GC机制，此处仅做简单介绍。

首先，对于UObject相关对象，为了维持引用（防止被回收），通常使用UProperty()宏，使用容器（如TArray存储），或调用AddToRoot的方法。

ue4的垃圾回收代码实现位于GarbageCollection.cpp中的CollectGarbage函数中。这一函数会在游戏线程中被反复调用，要么在一些情况下手动调用，要么在游戏循环Tick()中满足条件时自动调用。

GC过程中，首先会收集所有不可到达的对象（无引用）。

之后，根据当前情况，会在单帧（无时间限制）或多帧（有时间限制）的时间内，清理相关对象（IncrementalPurgeGarbage）

SIMD

合理的内存布局/对齐有利于SIMD的广泛应用，在编写定义基础类型/底层数学算法库时，我们通常有必要考虑到这一点。

我们可以参考ue4中封装的sse初始化、加法、减法、乘法等操作，其中，__m128类型的变量需程序确保为16字节对齐，它适用于浮点数存储，大部分情况下存储于内存中，计算时会在SSE寄存器中运用。

```
typedef __m128  VectorRegister;

FORCEINLINE VectorRegister VectorLoad( const void* Ptr )
{
        return _mm_loadu_ps((float*)(Ptr));
}

FORCEINLINE VectorRegister VectorAdd( const VectorRegister& Vec1, const VectorRegister& Vec2 )
{
        return _mm_add_ps(Vec1, Vec2);
}

FORCEINLINE VectorRegister VectorSubtract( const VectorRegister& Vec1, const VectorRegister& Vec2 )
{
        return _mm_sub_ps(Vec1, Vec2);
}

FORCEINLINE VectorRegister VectorMultiply( const VectorRegister& Vec1, const VectorRegister& Vec2 )
{
```

```
20.         return _mm_mul_ps(Vec1, Vec2);
21. }
```

除了SSE外，ue4还针对Neon/FPU等寄存器封装了统一的接口，这意味调用者可以无需考虑过多硬件的细节。

我们可以在多个数学运算库中看到相关的调用，如球谐向量的相加：

```
1.         /** Addition operator. */
2.         friend FORCEINLINE TSHVector operator+(const TSHVector& A,const
   TSHVector& B)
3.         {
4.                 TSHVector Result;
5.                 for(int32 BasisIndex = 0;BasisIndex <
   NumSIMDVectors;BasisIndex++)
6.                 {
7.                         VectorRegister AddResult = VectorAdd(
8.                                 VectorLoadAligned(&A.V[BasisIndex *
   NumComponentsPerSIMDVector]),
9.                                 VectorLoadAligned(&B.V[BasisIndex *
   NumComponentsPerSIMDVector])
10.                                );
11.
12.                        VectorStoreAligned(AddResult, &Result.V[BasisIndex *
   NumComponentsPerSIMDVector]);
13.                }
14.                return Result;
15.        }
```