链滴

C++& 计算机基础知识

作者：GNX

原文链接：https://ld246.com/article/1683859424356

来源网站：链滴

1. 指针和引用的区别?

指针和引用在汇编层面是一样的，引用就是指针。引用是编译器对指针做出各种约束之后的产物，我常说“引用是变量的别名”，这是因为当我们使用引用时，**编译器会自动做解引用的操作**，因此我们作引用时，总是操作引用的对象。

因此指针和引用的区别就是语义语法上的区别，引用类似于指针常量，即一旦赋值，不可更改指向，如：

- 定义引用时必须指定绑定的对象，因此不存在指向空值的引用，更安全。
- 指针可以非常灵活的改变指向的对象，引用不可以
- sizeof 指针得到的是本指针的大小，sizeof 引用得到的是引用所指向变量的大小

2. 堆和栈的区别

申请方式不同。

- 栈由系统自动分配。
- 堆是自己申请和释放的。

申请大小限制不同。

linux 下栈空间默认大小为 8m

堆区大小和虚拟内存大小有关，

申请效率不同。

** 栈由系统分配，有专门的 stack 指针寄存器，速度快**，不会有碎片。

堆由程序员分配，速度慢，且会有碎片

作用不同

栈中主要存储各个函数调用的栈帧信息，包括局部变量等，一旦函数结束，相应的栈帧信息和局部变就自动销毁

堆中内存需要手动释放，但是现代 c++ 一般使用利用 RAII 惯用法的智能指针来管理堆区内存，防止存泄漏。

int *p = new float[2]; //编译错误 int *p = new float[2]; //编译错误

```
int p = (int)malloc(2 * sizeof(double));//编译无错误
```

3. 你觉得堆快一点还是栈快一点?

毫无疑问是栈快一点。

1. 因为操作系统会在底层对栈提供支持，会分配专门的寄存器存放栈的地址，栈的入栈出栈操作也十简

单，并且有专门的指令执行，所以栈的效率比较高也比较快。

2. 而堆的操作是由 C/C++ 函数库提供的，在分配堆内存的时候需要一定的算法寻找合适大小的内存。

4.new / delete 与 malloc / free 的异同

- **相同点**:

都可以从堆区动态分配内存以及释放。

- **不同点:**

1. 前者是 C++ 运算符，后者是 C/C++ 语言标准库函数
2. new 自动计算要分配的空间大小，malloc 需要手工计算
3. new 是类型安全的，malloc 不是。例如:

类型安全:

每个对象在定义时被分配一个类型。对于一个程序或者程序的一部分，如果使用的对象符合它们定的类型，那么它们是类型安全的。不幸的是，有很多执行的操作不是类型安全的。

```
int *p = new float[2];     //编译错误 int *p = new float[2]; //编译错误
int p = (int)malloc(2 * sizeof(double));    //编译无错误
```

new 调用名为 operator new 的标准库函数分配足够空间并调用相关对象的构造函数，delete 对指所指对象运行适当的析构函数；然后通过调用名为 operator delete 的标准库函数释放该对象所用内。后者均没有相关调用。

对操作符的重载分为全局和类对象相关的局部重载，可以为每个类设计自己的 operator new（）

4. malloc 和 free 返回的是 void 类型指针（必须进行类型转换），new 和 delete 返回的是具体型指针。

c++ 的类型安全

如果 C++ 使用得当，它将远比 C 更有类型安全性。相比于 C，C++ 提供了一些新的机制保障类型全:

主要就是两个方面:

- 通过模板来代替 void 类型变量，而模板是支持类型检查的;
- 通过 inline 和 const 代替 define，c++ 中的 const 和 c 中的是不同的，c++ 中 const 变量一旦义好，在编译期就会存储一份变量的值，运行期可以更改变量在内存的值，但是使用该变量的地方，经在编译器进行值替换了，所以这个改变是无意义的。

具体来说：

（1）操作符 new 返回的指针类型严格与对象匹配，而不是 void*；

（2）C 中很多以 void*为参数的函数可以改写为 C++ 模板函数，而模板是支持类型检查的；

（3）引入 const 关键字代替 define constants，它是有类型、有作用域的，define constants 只是单的文本替换；

（4）一些#define 宏可被改写为 inline 函数，结合函数的重载，可在类型安全的前提下支持多种类，当然改写为模板也能保证类型安全；

（5）C++ 提供了 dynamic_cast 关键字，使得转换过程更加安全，因为 dynamic_cast 比 static_cas 涉及更多具体的类型检查。

即便如此，C++ 也不是绝对类型安全的编程语言。如果使用不得当，同样无法保证类型安全。比如面两个例子：

```
int i=5;
void* pInt=&i;
double d=(*(double*)pInt);
cout<<d<<endl;
```

————————————————————————

原文链接：https://blog.csdn.net/ixsea/article/details/6693178

5. new 和 delete 是如何实现的?

- new 的实现过程是：
  - 首先调用名为 operator new 的标准库函数，分配足够大的原始为类型化的内存，以保存指定型的一个对象；
  - 接下来运行该类型的一个构造函数，用指定初始化构造对象；
  - 最后返回指向新分配并构造后的的对象的指针。

可以对 operator new()函数进行重载，包括全局和局部重载，每个类可以定义自己的 new 函数，简的来讲 operator new()内部可以使对 malloc 的简单封装

6.宏定义和内联函数，typedef 有何区别?

宏定义

宏在预处理阶段进行简单的文本替换，替换后的内容才会参与编译，因此宏没有返回值，不会进行类

typedef

在编译器起作用，用于定义类型别名，有类型检查，是真正的类型别名，而非宏那样简单的文本替换。

```
#define a_char char*
typedef char* b_char;
int main()
{    //两者的简单对比，可以看出宏定义就是简单的文本替换
    a_char p1,p2;  //p1 是个 char 指针，p2 是个 char 变量
    b_char p3,p4; //p3,p4 都是 char 指针
    return 0;
}
```

7.变量声明、定义区别?

1.什么是对象定义：

定义一个对象，就是编译器为此对象**分配内存的地方**。

对函数和函数模板而言，定义式**提供了代码本体**。

对类和类模板而言，定义式**列出了他们成员**(一般在类中声函数，在类外定义成员函数)。

2.什么是对象的声明：

声明告诉编译器某个东西的名称和类型，但略去细节，**对象可以多次声明**。

8.strlen 和 sizeof 区别?

sizeof 是运算符，并不是函数，结果在编译时得到而非运行中获得；

strlen 是字符处理的库函数。

sizeof 参数可以是任何数据的类型或者数据（sizeof 参数不退化）；

strlen 的参数只能是字符指针且结尾是'\0'的字符串。

因为 sizeof 值在编译时确定，所以不能用来得到动态分配（运行时分配）存储空间的大小

```
 sizeof(str); // 取的是指针 str 的长度，是 8
 strlen(str); // 取的是这个字符串的长度，不包含结尾的 \0。大小是 4
 return 0;
 }
```

9.常量指针和指针常量区别?

指针常量:

又叫**顶层 const**，英文为：Constant pointer to variable，直译为指针常量。const 修饰的 pointer

意思为这个指针是个常量，一旦初始化后就不能更改指针变量的指向。表示为:

```
int * const p = &a;   //指针常量，指针变量本身不可改变
```

常量指针:

又叫底层 const，英文是：Pointer to constant，直译为指向常量的指针，即常量指针。意思为指向个常量，即指针变量本身可以更改指向，但是指向的内容不可改变

```
const int * p = &a; //指针常量
```

指针常量和常量指针这两个名字主要是看 const 和的相对位置，const 在*前面就叫常量指针。

10.C++ 中 struct 和 class 的区别

两者中如果不对成员不指定公私有，struct 默认是公有的，class 则默认是私有的，其他没有区别

11.C++ 和 C 中的 struct 区别

- C 语言中，struct 是用户自定义数据类型（UDT）；
- C++ 中 struct 是抽象数据类型（ADT），支持成员函数的定义（C++ 中的 struct 能继承，能实多态）。
- C 中 struct 是没有权限的设置的，且 struct 中只能是一些变量的集合体，可以封装数据却不可以藏数据，而且成员不可以是函数。
- C++ 中，struct 增加了访问权限，且可以和类一样有成员函数，成员默认访问说明符为 public（了与 C 兼容）struct 作为类的一种特例是用来自定义数据结构的。一个结构标记声明后，在 C 中必在结构标记前加上 struct，才能做结构类型名（除：typedef struct class{};）;C++ 中结构体标记（构体名）可以直接作为结构体类型名使用，此外结构体 struct 在 C++ 中被当作类的一种特例

12.C++ 中的 static 修饰符

13.C++ 中的 const 修饰符

14.数组名和指针的区别

● 数组名这个符号和数组的首地址对应，它的类型是一个数组，因此通过 sizeof(数组名)能确定数组长度。

● 数组名记录了数组的首地址，因此可以赋值给一个指针

```
int* p = NULL;
 ;表示将立即数0赋值给arry表示的内存地址中存储的数，数的大小为四字节
 int arry[4] = {1,1,2,3};
00007FF727DB1953  mov         dword ptr [arry],1
00007FF727DB195A  mov         dword ptr [rbp+4Ch],1
00007FF727DB1961  mov         dword ptr [rbp+50h],2
00007FF727DB1968  mov         dword ptr [rbp+54h],3
p = arry;
 ；数组名赋值给指针变量时，主要是下面这个lea指令
 ；lea指令可以用来将一个内存地址直接赋给目的操作数。
 ；而不是把ebx+8处的内存地址里的数据赋给eax。
00007FF727DB199B  lea         rax,[arry]
00007FF727DB199F  mov         qword ptr [p],rax

    a = *p;
 //解引用有如下两行代码
00007FF727DB19A3  mov         rax,qword ptr [p]
00007FF727DB19A7  mov         eax,dword ptr [rax]
```

15.初始化和赋值的区别

● 初始化是在定义变量的时候对变量赋予储值的过程，即定义和初始化发生在一条语句中。

● 对于简单类型来说，初始化和赋值没什么区别。

● 对于类和复杂类型区别明显：

● 在默认构造函数的初始化列表中才叫初始化，**这段代码先于构造函数的函数体执行。**若在函数中赋予初值，实际上是赋值，因为若没有在初始化列表中指定初值，则所有成员变量会进行默认值初化。

```
class A{
public:
 int num1;
 int num2;
public:
 A(int a=0, int b=0):num1(a),num2(b){};
 A(const A& a){};
 //重载 = 号操作符函数
 A& operator=(const A& a){
     num1 = a.num1 + 1;
     num2 = a.num2 + 1;
     return *this;
 };
};
int main(){
```

```
    A a(1,1);
    A a1 = a; //拷贝初始化操作，调用拷贝构造函数
    A b;
    b = a;//赋值操作，对象a中，num1 = 1，num2 = 1；对象b中，num1 = 2，num2 = 2
    return 0;
}
```

16.野指针和悬空指针

野指针:

指的是没有被初始化过的指针

悬空指针:

指针最初指向的内存已经被释放了的一种指针。

17.C++ 的类型安全

所谓类型安全：

每个对象在定义时被分配一个类型。对于一个程序或者程序的一部分，如果使用的对象符合它们规定类型，那么它们是类型安全的。而类型检查，检查的正是所使用对象的类型和类型对应的相关操作是相符。

如果 C++ 使用得当，它将远比 C 更有类 型安全性。相比于 C 语言，C++ 提供了一些新的机制保障型安全：

1. 操作符 new 返回的指针类型严格与对象匹配，而不是 void*

2. C 中很多以 void*为参数的函数可以改写为 C++ 模板函数，而模板是支持类型检查的；

3. 引入 const 关键字代替 #define constants,它是有类型、有作用域的，而#define constants 只简单的文本替换

4. 一些#define 宏可被改写为 inline 函数，结合函数的重载，可在类型安全的前提下支持多种类型，然改写为模板也能保证类型安全

5. C++ 提供了 dynamic_cast 关键字，使得转换过程更安全，因为 dynamic_cast 比 static_cast 涉更多具体的类型检查。

18.C++ 中的重载、重写（覆盖）和隐藏/名称遮蔽的区别

重载(overload)

C++ 允许为在同一**作用域**中的某个**函数**和**运算符**指定**多个定义**，分别称为**函数重载**和**运算符重载。**

函数重载

语法要求：**函数名相同，参数列表不同**（包括参数个数，参数类型，形参类型顺序，三个条件任意不，均可构成重载）但是要注意**返回值类型**不做判断

运算符重载

c++ 中大部分运算符都可以重载，重载的运算符是**带有特殊名称的函数**，函数名是由关键字 operator 和其后要重载的运算符符号构成的。与其他函数一样，重载运算符有一个返回类型和一个参数列表。

重载运算符有两种方式：

1. **全局重载** 2. 类内重载**

全局重载时，参数必须为类类型，否则无法定义成全局函数。

当定义为类成员函数时，二元运算符只需指定一个参数，因为另一个参数就是调用者本身的 this 指针，

当定义成全局函数时就需要两个类类型参数。

语法格式：

```
Box operator+(const Box&);   //类内定义，只需要一个参数，默认将调用者对象作为左参数
Box operator+(const Box&, const Box&);   //全局函数
```

下面是不可重载的运算符列表：
.：成员访问运算符
.*, ->*：成员指针访问运算符
::：域运算符
sizeof：长度运算符
?:：条件运算符
#： 预处理符号

重写（覆盖）（override）

主要对于类中的虚函数而言。重写指的是在子类中覆盖父类中的同名同参函数虚函数，override 就是写函数体，要求基类函数**必须是虚函数**且：

1. 与基类的虚函数有相同的参数个数

2. 与基类的虚函数有相同的参数类型

3.与基类的虚函数有相同的参数顺序
4.与基类的虚函数有相同的返回值类

隐藏/屏蔽（hide）

在嵌套的作用域中，内层作用域和外层作用域有同名函数(**参数列表可以不同**)，在内层作用域的**同名数**会覆盖上一层的同名函数，此时在内层作用域无法使用被遮蔽的函数。比如说子类中有一个和父类名的函数（无论形参是否相同），使用子类对象调用这个函数时，即使传入的参数是和父类中被遮蔽函数一样，也不会调用那个函数，只会提示参数列表不符。

三者区别：

重载：重载是针对相同作用域的同名但是不同参的函数而言，重载能加函数的灵活性，它使一个函数以有不同的接口。

重写：重新主要针对具有继承关系的两个类中的虚函数而言，这提供了对多态的支持。

屏蔽：屏蔽和重写都是针对具有**层级关系**但是属于不同作用域的两个函数而言。但重新要求函数必须虚函数，且名称和参数列表都相同。但是屏蔽只要求名字相同。

19.C++ 有哪几种的构造函数

C++ 中的构造函数可以分为 6 类：

1. 默认构造函数：

主要指编译器自动合成的构造函数，或者不带参数的构造函数

2.有参初始化构造函数：

需要指定参数的构造函数

3.拷贝构造函数：

用另一个对象初始化当前类对象。默认是浅拷贝，如果有指针资源，一定要深拷贝，区别于移动构造数的特点。

4.移动构造函数（move 和右值引用）：

若判定传入的是右值对象，则调用移动构造而非拷贝构造。移动构造两个特点：

1. 对指针资源执行浅拷贝。
2. 将传入对象的指针置 NULL，即转移对方的资源到自己名下。

若定义了移动构造函数，而没有定义拷贝构造函数，c++默认不会是再生产默认的拷贝构造函数，即不可拷贝，只能移动，反过来不会。

5.委托构造函数：

实际就是在一个构造函数中的初始化列表中调用其他构造函数，完成代码的复用。使用上和子类调用类构造函数，完成对父类的初始化一样。

要注意避免递归调用，导致栈溢出。

6.转换构造函数：

转换构造函数(conversion constructor function) 的作用是将一个其他类型的数据转换成一个类的对
。
当一个构造函数**只有一个参数**，而且该参数又不是本类的 const 引用时，这种构造函数称为转换构造
数。

它其实是普通构造函数的一种。

比如：

```
//string接收了一个char*型数据，在内部转换为string对象
string a = "hello world";
```

```

#include <iostream>
using namespace std;
class Student{
public:
//默认构造函数，没有参数
 Student():age(20),num(1000),name("NULL"){};
//初始化构造函数，有参数和参数列表
 Student(int a, int n):age(a), num(n),name("NULL"){};
//委托构造函数,它的作用就是复用其他构造函数
Student(string name):Student(0,0),name(name){}
//拷贝构造函数，这里与编译器生成的一致
//当有指针资源时，要在拷贝构造函数中写深拷贝的代码。
 Student(const Student& s){
     this->age = s.age;
     this->num = s.num;
 };
//转换构造函数,形参是其他类型变量，且只有一个形参
 Student(int r){
     this->age = r;
     this->num = 1002;
 };
 ~Student(){}
public:
 int age;
 int num;
string name;
int* p = NULL;
};
```

19.浅拷贝和深拷贝的区别

浅拷贝

浅拷贝只是拷贝一个指针，但没有新开辟一块空间取赋值原指针指向的内容。拷贝的指针和原来的指指向同一块地址，如果原来的指针所指向的资源释放了，那么再释放浅拷贝的指针的资源就会出现错。

类的拷贝构造函数一定要深拷贝，但是移动构造函数一定是浅拷贝。

深拷贝

深拷贝不仅拷贝值，还开辟出一块新的空间用来存放新的值，即使原先的对象被析构掉，释放内存了不会影响到深拷贝得到的值。在自己实现拷贝赋值的时候，如果有指针变量的话是需要自己实现深拷的。

浅拷贝在对象的拷贝创建时存在风险，即被拷贝的对象析构释放资源之后，拷贝对象析构时会再次释一个已经释放的资源，深拷贝的结果是两个对象之间没有任何关系，各自成员地址不同。

20.内存对齐

什么是内存对齐

理论上计算机对于任何变量的访问都可以从任意位置开始，然而实际上系统会对这些变量的存放地址限制，通常将变量首地址设为某个数 N 的倍数，这就是内存对齐。

优点

● 这种对齐限制简化了形成处理器和内存系统之间接口的硬件设计。

● 兼容性。不同硬件平台不一定支持任何内存地址的存取，为了保证不同平台处理器正确存取数，需要进行内存对齐··

● 提高 CPU 内存访问速度，一般处理器的内存存取粒度都是 N 的整数倍，因此可以访问的地址顺序 0 - N-1再到N-2N 假如访问 N 大小的数据，没有进行内存对齐，有可能就需要两次访问才可以读取数据，而进行内存对齐可以一次性把数据全部读取出来，提高效率

无论数据是否对齐，x86-64 硬件都能正常工作，但是 Intel 还是建议对齐数据以提高内存系统的性能。

对齐原则：

1. 数据成员对齐规则：

struct 或者 union 的数据成员，第一个数据成员放在 offset 为 0 的地方，以后每个数据成员都按照#ragma

pack 数值和这个数据成员自身大小中更小的那个进行对齐。

2. 整体对齐规则：struct 或者 union 的首地址按照内部最大数据成员的大小和#pragma pack 数值小的那个 N 进行对齐，并且

结构体的总大小为 N 的整数倍，如有必要编译器也会在最后一个成员后面填充一些字节用于对齐

计算机存储结构体对象时，为结构体分配的首地址肯定是已经对齐过的，也就是可以直接访问到结构对象的首地址，所以结构体内的各种类型可以将结构体的首地址当做 0 来对齐，也就是偏移地址对齐。

原因

两方面考虑：

1.系统内存管理层面

内存的存取粒度一般为4k的页大小，整个内存空间都会以4K最最小值进行分片。

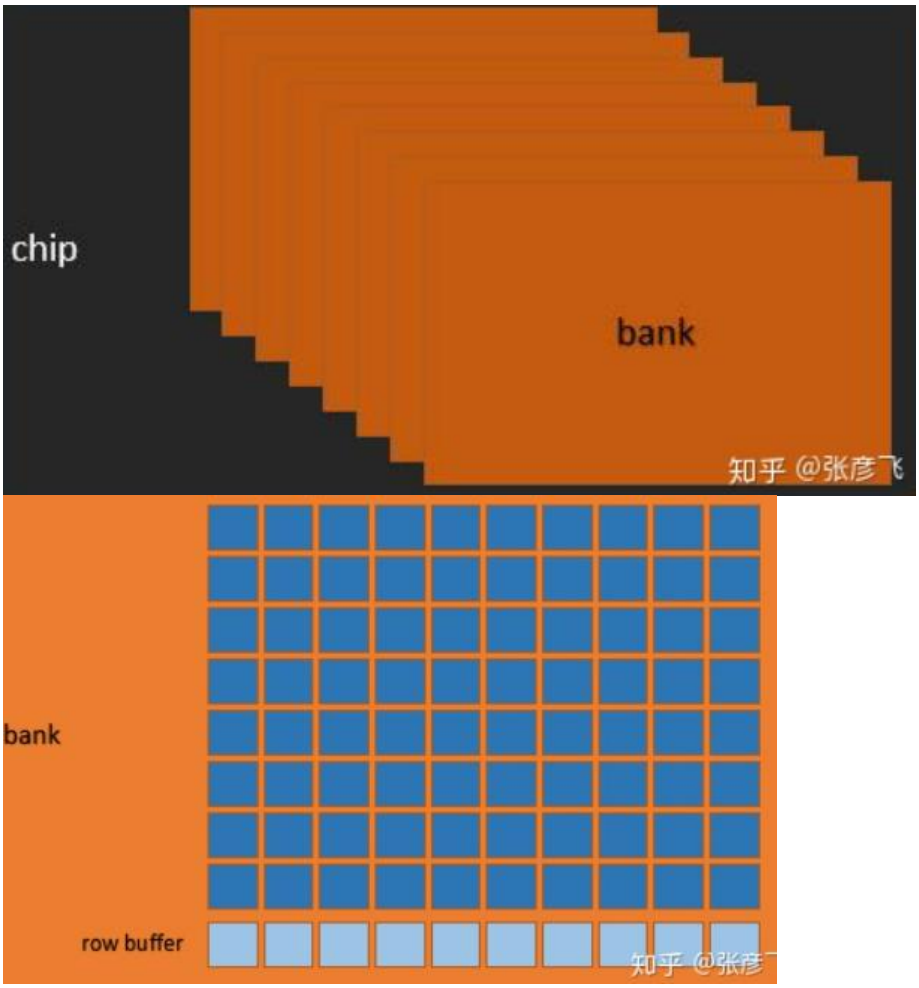
所以如果对象不做内存对齐，可能使得一个对象位于两个页中，降低访问幅度。

2.硬件底层原因

要想理解到底为什么要内存对齐，必须理解内存时如何获取的。

我们来了解一下内存的物理构造，一般内存的外形图片如下图：

一个内存是由若干个黑色的内存颗粒构成的。每一个内存颗粒叫做一个 **chip**。每个 chip 内部，是由 **8 个 bank** 组成的。其构造如下图：

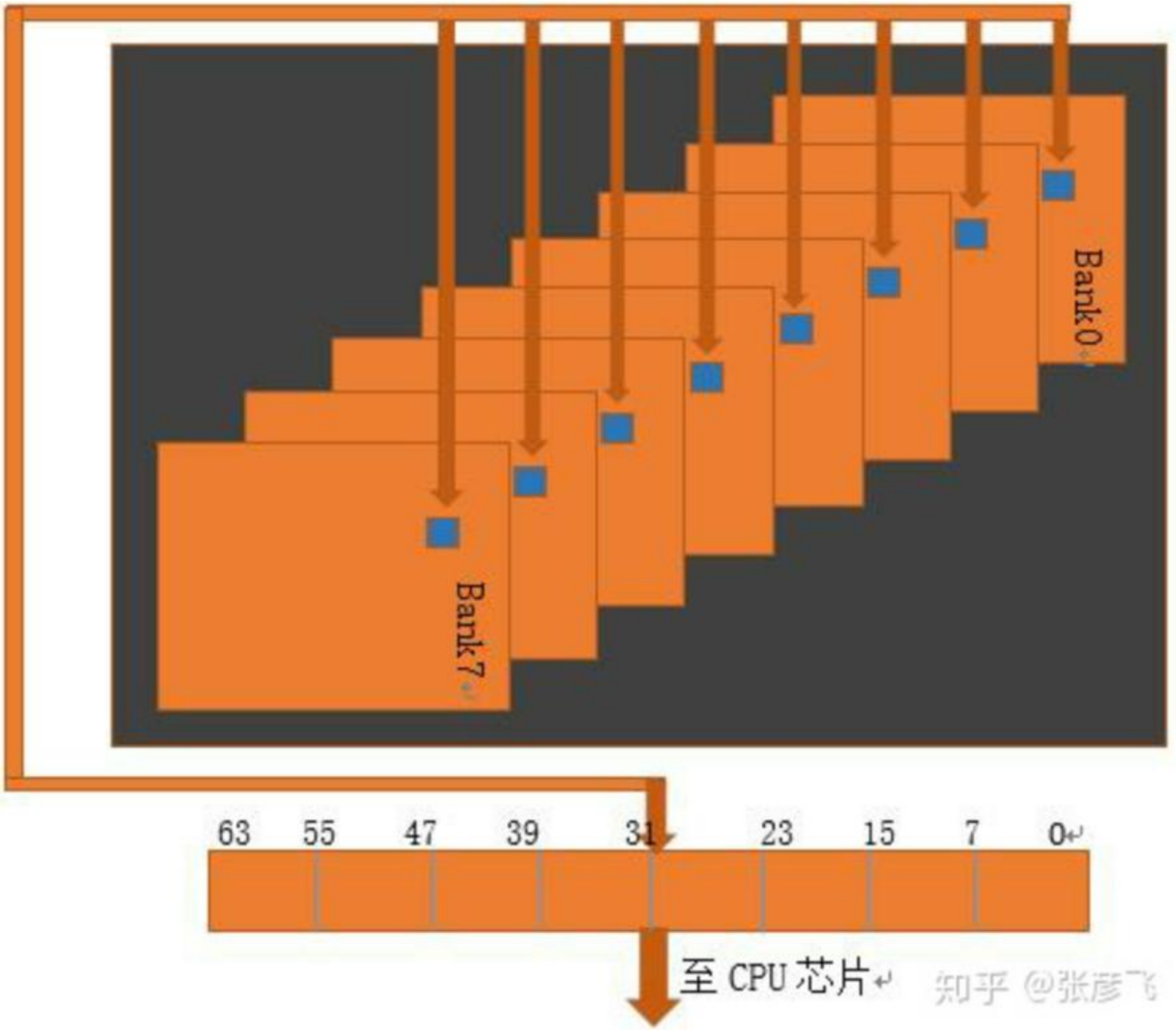


而每一个 bank 是一个二维平面上的矩阵，前说到过,矩阵中每一个元素中都是保存了 1 个字节，也就 8 个 bit。

内存编址方式：

那么对于我们在应用程序中内存中地址连续的 8 个字节,例如 0x0000-0x0007，是从位于 bank 上的？直观感觉，应该是在第一个 bank 上吗？ 其实不是的，程序员视角看起来连续的地址 0x0000-0x007，实际上位 8 个 bank 中的，每一个 bank 只保存了一个字节。**在物理上，他们并不连续。**下图很地阐述了实际情况。

图 4 连续 8 字节在内存中实际分布



你可能想知道这是为什么，原因是电路工作效率。内存中的 8 个 bank 是可以并行工作的。 如果你读取址 0x0000-0x0007，每个 bank 工作一次，拼起来就是你要的数据，IO 效率会比较高。但要存一个 bank 里，那这个 bank 只能自己干活。只能串行进行读取，需要读 8 次，这样速度会慢很多。

结论

所以，内存对齐最最底层的原因是内存的 IO 是以 8 个字节 64bit 为单位进行的。 对于 64 位数据宽的内存，假如 cpu 也是 64 位的 cpu（现在的计算机基本都是这样的），每次内存 IO 获取数据都是同行同列的 8 个 bank 中各自读取一个字节拼起来的。从内存的 0 地址开始，0-7 字节的数据可以一 IO 读取出来，8-15 字节的数据也可以一次读取出来。

换个例子，假如你指定要获取的是 0x0001-0x0008，也是 8 字节，但是地址不是 0 开头的，内存需怎么工作呢?

没有好办法，内存只好先工作一次把 0x0000-0x0007 取出来，然后再把 0x0008-0x0015 取出来，两次的结果都返回给你。 CPU 和内存 IO 的硬件限制导致没办法一次跨在两个数据宽度中间进行 IO 这样你的应用程序就会变慢，算是计算机因为你不懂内存对齐而给你的一点点惩罚。

扩展 1：事实上，编译和链接器会自动替开发者对齐内存的，尽量帮你保证一个变量不跨列寻址。但他不能做到十分完美。扩展 2：其实在内存硬件层上，还有操作系统层。操作系统还管理了 CPU 的

级、二级、三级缓存。不知道你有没有印象，我们前面的文章说过高速缓存里的 Cache Line 也是 64 字节，它是内存 IO 的整数倍，不会让内存 IO 浪费。

21.字节序的大端小端模式

对于跨越多字节的对象，必须建立两个规则：

- **这个对象的地址是什么?**
- **在内存中如何排列这些字节?**

1.由于多字节对象都被存储为连续的字节序列，因此对象的内存地址为所使用字节中最小的地址。
2.首先要明确三个概念：

- 字节序的最高有效字节（高位）
- 字节序最低有效字节（地位）
- 内存地址的增长方向

例子：整数 127 的十六进制表示 0x00 00 00 7F
最左边的 00 是最高有效字节，最右边的 7F 是最低有效位。
内存地址的增长方向是从对象的首地址开始顺序增长的。

小端模式：最低有效位在对象地址空间的最前面。

大端模式：最高有效位在对象地址空间的最前面。

再举个例子：

存储 0x1A2B3C4D，总共四个字节。
大端模式存储：
内存低地址 ----------> 内存高地址

0x1A | 0x2B | 0x3C | 0x4D
高位字节 <---------- 低位字节
小端模式存储：
内存低地址 ----------> 内存高地址
0x4D | 0x3C | 0x2B | 0x1A
低位字节 ----------> 高位字节

相关知识：

arm 处理器和 X86-64 的机器几乎都是小端存储模式，选择哪种字节序没有技术上的理由。

在 TCP/IP 传输中，为了确保接收方明确知道数据的二进制序列排序方式，因此网络发送的数据一律换为大端模式。

还要特别注意，数据在内存中是以补码形式存储的

为什么网络字节序选择大端模式?

1. 大端模式的数据比较方便阅读，可能利于抓包分析数据

2. 早年设备的缓存很小，先接收高字节能快速的判断报文信息：包长度（需要准备多大缓存）、地址围（IP 地址是从前到后匹配的）。在性能不是很好的设备上，高字节在先确实是会更快一些。

为什么主机字节序（多数情况下）是小端?

小端的加法器比较好做，如果做一个 8 位*4 的加法器，只需要一个 8 位加法器，然后依次从低到高环加上所有字节就可以了，进位的电路非常简单，而如果是大端，则需要一次加载 32 位，不然的话位的设计比较困难

21.浮点数的表示

我们已经知道，浮点数是采用科学计数法来表示一个数字的，它的格式可以写成这样：

V = (-1)^S^ MR^E^

其中各个变量的含义如下：

- S：符号位，取值 0 或 1，决定一个数字的符号，0 表示正，1 表示负
- M：尾数，表示数字的有效数字，例如 8.345 * 10^0^, 8.345 就是尾数
- R：基数，表示十进制数 R 就是 10，表示二进制数 R 就是 2
- E：指数，用整数表示，例如前面看到的 10-1，-1 即是指数

单精度的尾数占 23 位，加上隐藏位共 24 位，能表示 2^24^=16,777,216 共 8 位，也就是能完整表示 7 位十进制数。

双精度时，2^53^= 9,007,199,254,740,992 共 16 位，能完整表示 15 位 10 进制数

超过这些位后，是要损失精度的

如果我们要在计算机中，用浮点数表示一个数字，只需要确认这几个变量即可。

假设现在我们用 32 bit 表示一个浮点数，把以上变量按照一定规则，填充到这些 bit 上就可以了：

我们可以看到，**指数**和**尾数**分配的位数不同，会产生以下情况：

- **指数位越多，尾数位则越少，其表示的范围越大，但精度就会变差，反之，指数位越少，尾数位则多，表示的范围越小，但精度就会变好**
- **一个数字的浮点数格式，会因为定义的规则不同，得到的结果也不同，表示的范围和精度也有差异**

直到 1985 年，IEEE 组织推出了浮点数标准，就是我们经常听到的 IEEE754 浮点数标准，这个标准一了浮点数的表示形式，并提供了 2 种浮点格式：

- 单精度浮点数 （signal float）：32 位，符号位 S 占 1 bit，指数 E 占 8 bit， **尾数 M 占 23 bit**
- 双精度浮点数（double float）：64 位，符号位 S 占 1 bit，指数 E 占 11 bit， **尾数 M 占 52 bit**

为了使其表示的**数字范围、精度最大化**，浮点数标准还对指数和尾数进行了规定：

- **尾数 M 的第一位总是 1（因为 1 <= M lt; 2），因此这个 1 可以省略不写，它是个隐藏位，这样单精度 23 位尾数可以表示了 24 位有效数，双精度 52 位尾数可以表示 53 位有效数字**
- 指数 E 是个无符号整数，表示 float 时，一共占 8 bit，所以它的取值范围为 0 ~ 255。但因为指数以是负的，所以规定在存入 E 时**在它原本的值加上一个中间数 127**，这样 E 的取值范围为 -127 ~ 12。表示 double 时，一共占 11 bit，存入 E 时加上中间数 1023，这样取值范围为 -1023 ~ 1024。

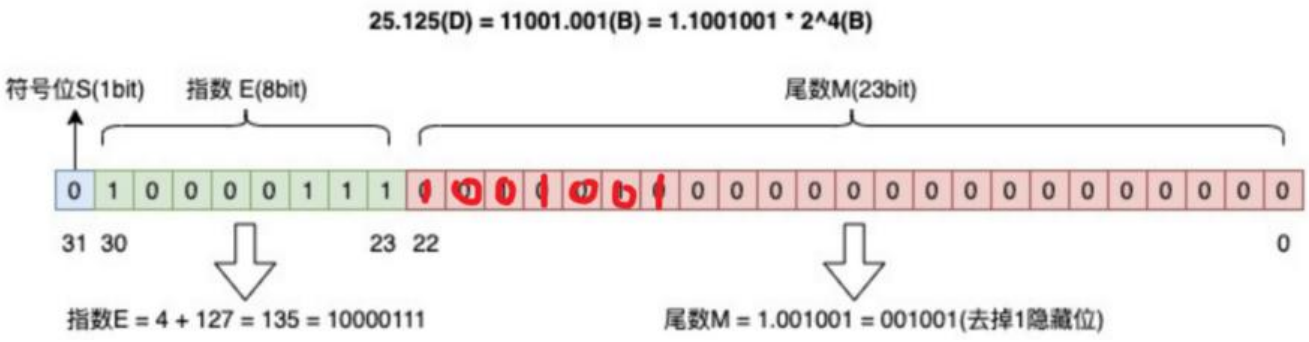
注意！！

指数位是有符号的，但是不同通常那样设置 1 位作为专门的符号位，而是加入中间数 127 的方法，即 +127，这里 127 表示 0，所以 0-126 对应-127 到（-1），128-255 对应 1-128

有了这个统一的浮点数标准，我们再把 25.125 转换为标准的 float 浮点数：

- 整数部分：25(D) = 11001(B)
- 小数部分：0.125(D) = 0.001(B)
- 用二进制科学计数法表示：25.125(D) = 11001.001(B) = 1.1001001 * 2^4(B)

所以 S = 0，尾数 M = 1.1001001 = 1001001(去掉 1，隐藏位)，指数 E = 4 + 127(中间数) = 135(D = 10000111(B)。填充到 32 bit 中，如下：



原文链接：C++& 计算机基础知识

这就是标准 32 位浮点数的结果。

如果用 double 表示，和这个规则类似，指数位 E 用 11 bit 填充，尾数位 M 用 52 bit 填充即可。

例子 2：

以 78.375 为例，它的整数和小数部分可以表示为： 因此二进制的科学计数法为：

$$(78)_{10} = (1001110)_2$$

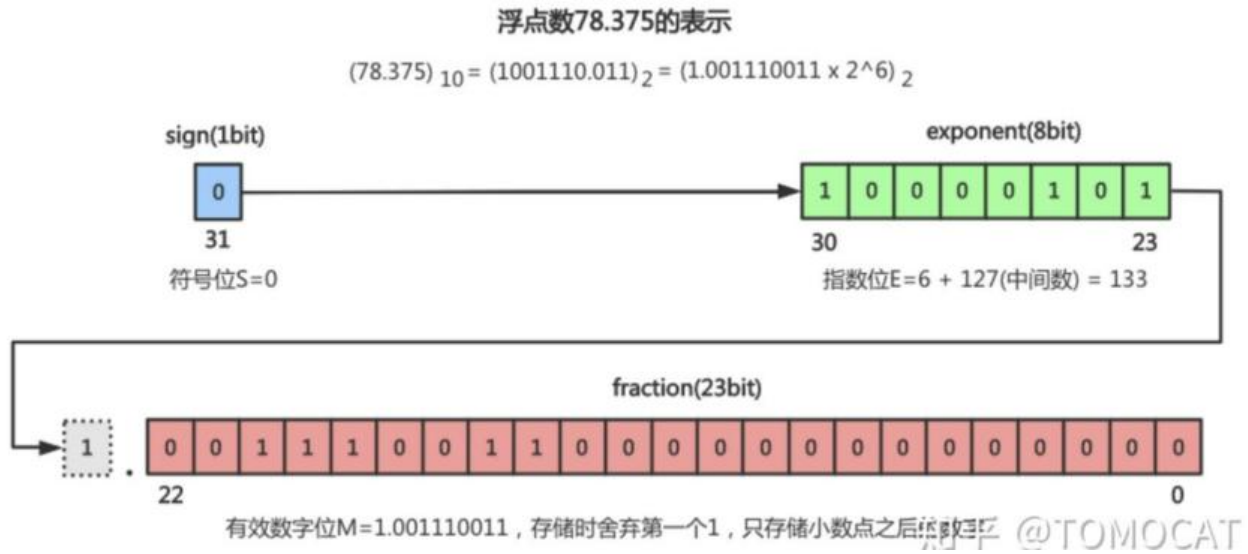
$$(0.375)_{10} = \frac{3}{8} = \frac{1}{4} + \frac{1}{8} = 2^{-2} + 2^{-3} = (0.01)_2 + (0.001)_2 = (0.011)_2$$

$$(78.375)_{10} = (1001110.011)_2 = 1.001110011 \times 2^6$$

一般而言转换过程包括如下几步：

- 改写整数部分：将整数部分的十进制改写成二进制
- 改写小数部分：拆成 到 的和
- 规格化：保证小数点前只有一位是 1，改写成二进制的科学计数法表示
- 填充：指数部分加上 127（中间数）填充 E；有效数字部分去掉 1 后填充 M

按照前面 IEEE 754 的要求，它的底层存储为：



2.2.浮点数为什么有精度损失?

我们再来看一下，平时经常听到的浮点数会有精度损失的情况是怎么回事?

如果我们现在想用浮点数表示 0.2，它的结果会是多少呢?

0.2 转换为二进制数的过程为，不断乘以 2，直到不存在小数为止，在这个计算过程中，得到的整数分从上到下排列就是二进制的结果。

0.2 * 2 = 0.4 -> 0
0.4 * 2 = 0.8 -> 0
0.8 * 2 = 1.6 -> 1

0.6 * 2 = 1.2 -> 1
0.2 * 2 = 0.4 -> 0（发生循环）
...

所以 0.2(D) = 0.00110...(B)。

因为十进制的 0.2 无法精确转换成二进制小数，而计算机在表示一个数字时，宽度是有限的，无限循的小数存储在计算机时，只能被截断，所以就会导致小数精度发生损失的情况。

2.3.浮点数的范围和精度有多大?

单精度浮点数 float 占四个字节，表示范围-3.40E+38 ~ +3.40E+38

双精度浮点数 double 占八个字节，表示范围-1.7E+308 ~ +1.79E+308

以 float 为例，能表示的最大二进制数据为 （小数点后为 23 个 1），而二进制下 ，因此能表示的大十进制数据是：

$$(+1.11111111111111111111111 \times 2^{127})_2 \approx (2^{128})_2 \approx (3.4 \times 10^{38})_{10}$$
[公式]

$$1.111111... \approx 2$$
[公式]

$$+1.11111111111111111111111 \times 2^{127}$$
[公式]

float 能表示的最小二进制数为 0.0000....1（小数点后 22 个 0，1 个 1），用十进制数表示就是 1/22。

用同样的方法可以算出，double 能表示的最大二进制数为 +1.111...111（小数点后 52 个 1） * 2103 ≈ 2^1024 = 1.79 * 10^308，所以 double 能表示范围为：-1.79 * 10^308^ ~ +1.79 * 10^308。

double 的最小精度为：0.0000...1(51 个 0，1 个 1)，用十进制表示就是 1/2^52。

从这里可以看出，虽然浮点数的范围和精度也有限，但其范围和精度都已非常之大，所以在计算机中对于小数的表示我们通常会使用浮点数来存储。

计算机将信息编码为位（比特），通常组织成字节序列。有不同的编码方式用来表示整数、实数和字串。不同的计算机模型在编码数字和多字节数据中的字节顺序时使用不同的约定

22.unicode 和 UTF-8/UTF-16/UTF-32

简单来说：

- Unicode 是「 **字符集**」
- UTF8/UTF16/UTF32 是「 **编码规则**」

其中：

字符集：某些字符的集合，并为每一个「字符」分配一个唯一的 ID（学名为码位 / 码点 / Code Point）

编码规则：将「码位」转换为方便计算机解释和传输的字节序列的规则（编码/解码 可以理解为密/解密 的过程）

Q:既然字符集中每个字符都有唯一编号，是否可以直接用这个编号存储在计算机中来表示这个字符呢?

A:这是可以的,UTF32 编码方式就直接使用固定的 4 字节长度和 UCS-4 编码一一对应，字符在 java 内部处理一律用的是 Unicode，这样处理的速度很快，但是很占存储空间。只有传输和储存的时候才用 gb2312，gbk，utf-8 来处理。

总的来说，用 UCS-4 来直接编码字符序列的问题就是太占空间，优势就是处理的快，为了解决太占间的问题，引出了 UTF8 编码方式。

广义的 Unicode 是一个标准，定义了一个字符集以及一系列的编码规则，即 Unicode 字符集和 UTF-、UTF-16、UTF-32 等等编码……

unicode 的全程为 Universal Character Set coded，简称 **UCS**，**一般用两个字节表示一个字符，简 UCS-2，但是两个字节能表示的字符太有限，所以有增加到 4 字节表示一个字符，简称 UCS-4。**

要注意，UCS-2 和 UCS-4 只规定了代码点和文字之间的对应关系，并没有规定代码点在计算机中如存储。**规定存储方式的称为 UTF**（Unicode Transformation Format）

1.UTF-8

UTF-8 顾名思义，是一套以 8 位为一个编码单位的可变长编码。会将一个码位编码为 1 到 4 个字节：

● 单字节的字符，字节的第一位设为 0，对于英语文本，UTF-8 码只占用一个字节，和 ASCII 码完全同；

● n 个字节的字符(n>1)，第一个字节的前 n 位设为 1，第 n+1 位设为 0，后面字节的前两位都设为 0，这 n 个字节的其余空位填充该字符 unicode 码，高位用 0 补足。

U+ 0080 ~ U+ 07FF: 110XXXXX 10XXXXXX
U+ 0800 ~ U+ FFFF: 1110XXXX 10XXXXXX 10XXXXXX
U+10000 ~ U+10FFFF: 11110XXX 10XXXXXX 10XXXXXX 10XXXXXX

根据上表中的编码规则，之前的「知」字的码位 U+77E5 属于第三行的范围：

7 7 E 5
0111 0111 1110 0101 二进制的 77E5

0111 011111 100101 二进制的 77E5
1110XXXX 10XXXXXX 10XXXXXX 模版（上表第三行）
11100111 10011111 10100101 代入模版
E 7 9 F A 5

这就是将 U+77E5 按照 UTF-8 编码为字节序列 E79FA5 的过程。反之亦然。

只有 UTF8 才会兼容 ASCII 码。

2.UTF-16

它使用 2 个或者 4 个字节来存储。

对于 Unicode 编号范围在 0 ~ FFFF 之间的字符，UTF-16 使用两个字节存储，并且直接存储 Unicod 编号，不用进行编码转换，这跟 UTF-32 非常类似。

对于 Unicode 编号范围在 10000-10FFFF 之间的字符，UTF-16 使用四个字节存储，具体来说就是将字符编号的所有比特位分成两部分，较高的一些比特位用一个值介于 D800DBFF 之间的双字节存，较低的一些比特位（剩下的比特位）用一个值介于 DC00~DFFF 之间的双字节存储。

Unicode 编号范围（十六进制）	具体的 Unicode 编号（二进制）	UTF-16 编码	编码后的字节数
0000 0000 ~ 0000 FFFF	xxxxxxxx xxxxxxxx	xxxxxxxx xxxxxxxx	2
0001 0000---0010 FFFF	yyyy yyyy yyxx xxxx xxxx	110110yy yyyyyyyy 110111xx xxxxxxxx	4

3.UTF-32

UTF-32 是固定长度的编码，始终占用 4 个字节，足以容纳所有的 Unicode 字符，所以直接存储 Unicode 编号即可，不需要任何编码转换。**浪费了空间，提高了效率。**

4.GB2312、GBK 等国家（地区）字符集怎么编码

GB2312、GBK、Shift-JIS 等特定国家的字符集都是在 ASCII 的基础上发展起来的，它们都兼容 ASCI，所以只能采用变长的编码方案：用一个字节存储 ASCII 字符，用多个字节存储本国字符。

以 GB2312 为例，该字符集收录的字符较少，所以使用 1~2 个字节编码。

- 对于 ASCII 字符，使用一个字节存储，并且该字节的最高位是 0；
- 对于中国的字符，使用两个字节存储，并且规定每个字节的最高位都是 1。

由于单字节和双字节的最高位不一样，所以很容易区分一个字符到底用了几个字节。

GBK 字符编码支持简体中文和繁体中文！

GBK 即汉字内码扩展规范，K 为汉语拼音 Kuo Zhan（扩展）中“扩”字的声母。英文全称 Chinese nternal Code Specification。

GB2312 只支持简体中文

GB2312 编码的范围：

高字节范围是 0XA1-0XFF，低字节范围是 0XA1-0XFF。GB2312 原始编码 (encoding) 是对所收录每个字符都用两个字节 (byte) 表示。第一字节为 “高字节”，由字符的区号值加上 32 而形成；第字节为 “低字节”，由字符的位号值加上 32 而形成。在区位号值上加 32 的原因大慨是为了避开低字节区间。由于 GB2312 原始编码与 ASCII 编码的字节有重叠，现在通行的 GB2312 编码是在原始码的两个字节上各加 128 修改而形成，如果不另加说明，GB2312 常指这种修改过的编码。

例如：汉字 “啊” ，编号为 16 区 01 位。

原始编码：高位字节为 16+32=48，16 进制为 0X30；低位字节为 1+32=33，16 进制为 0X21。所它的原始编码为 0x3021。

通行编码：高位字节为 16+32+128=176，16 进制为 0XB0；低位字节为 1+32+128=161，16 进为 0XA1。所以它的通行编码为 0xB0A1。

2.5.1.3 GB2312 字符集的区位分布表

GBK/GB2312 还存在的原因就是他们存储汉子时只需要两个字符，而 utf-8 一般需要 3 个字符，所更节省空间，但是缺点就是通用性更低，

GBK、GB2312 等与 UTF8 之间都必须通过 Unicode 编码才能相互转换：

GBK、GB2312 - - Unicode - - UTF8

UTF8 - - Unicode - - GBK、GB2312

5.宽字符和窄字符（多字节字符）

有的编码方式采用 1~n 个字节存储，是变长的，例如 UTF-8、GB2312、GBK 等；如果一个字符使了这种编码方式，我们就将它称为多字节字符，或者窄字符。

有的编码方式是固定长度的，不管字符编号大小，始终采用 n 个字节存储，例如 UTF-32、UTF-16；如果一个字符使用了这种编码方式，我们就将它称为宽字符。

Unicode 字符集可以使用窄字符的方式存储，也可以使用宽字符的方式存储；GB2312、GBK、Shift-JS 等国家编码一般都使用窄字符的方式存储；ASCII 只有一个字节，无所谓窄字符和宽字符。

□

23.字和字节

字： 计算机一次处理数据的最大单位，字长也是 cpu 通用寄存器的长度。常说的 32 位机器或者 64 机器，这里的 32 与 64 指的也是 cpu 中寄存器的长度。

扩展：一个 X86-64 的 cpu 包含一组 16 个存储 64 位值得通用目的寄存器，用来存放整数数据和指针

在汇编中，由于是从 16 位体系结构扩展成 32 位、64 位的，因此 Inter 用术语“字（word）”表示 6 进制数据类型，因此 32 位称为双字，64 位是四字

字节：字节是寻址的最小单位，计算机中以字节为单位来存储和解释信息，一个字节占用 8bit。

计算机将信息编码为位（比特），通常组织成字节序列。有不同的编码方式用来表示整数、实数和字串。不同的计算机模型在编码数字和多字节数据中的字节顺序时使用不同的约定

□

24.指针和引用的区别?

引用实际上就是基于指针加了一些语法糖，比如必须初始化、不能改变引用的指向等等。

但是引用是别名这是 c++ 语法规定的语义，那么到底引用在汇编层面和指针有什么区别呢?

没区别。 对，引用会被 c++ 编译器当做 const 指针来进行操作。

语义上来说对引用的任何操作都相当于对变量的直接操作，也就是在编译器内部会自动进行（p）操，所以对引用取地址，就是对引用的变量取地址

```
*int* p = &a;
00007FF7742D7908 lea rax,[rbp+24h]
00007FF7742D790C mov qword ptr [rbp+48h],rax
int& q = a;
00007FF7742D7910 lea rax,[rbp+24h]
00007FF7742D7914 mov qword ptr [rbp+68h],rax
int& Q = a;
00007FF7742D7918 lea rax,[rbp+24h]
00007FF7742D791C mov qword ptr [rbp+0000000000000088h],rax
int c = q;
00007FF7742D7923 mov rax,qword ptr [rbp+68h]
00007FF7742D7927 mov eax,dword ptr [rax]
00007FF7742D7929 mov dword ptr [rbp+00000000000000A4h],eax
*p = 9;
00007FF7742D792F mov rax,qword ptr [rbp+48h]
00007FF7742D7933 mov dword ptr [rax],9
```

25.存储器层次结构

CPU 寄存器-->Cach 高速缓存（SRAM）（包括 L1/L2/L3 多级缓存）--> 主存（DRAM）--> 磁盘固态硬盘--> 远程存储系统。

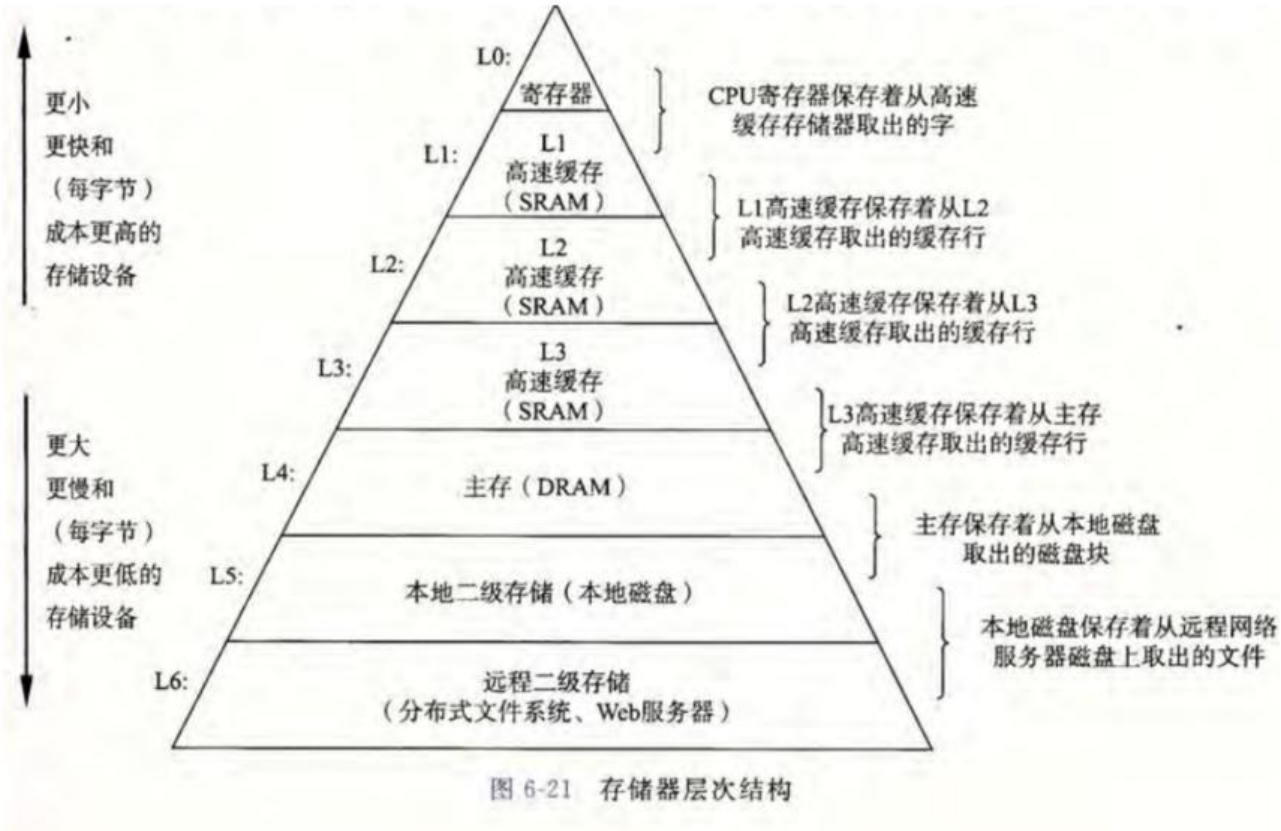


图 6-21 存储器层次结构

26.存储介质分类

RAM:

- SRAM:主要用做 cache，速度比 DRAM 快数十倍，成本更高
- DRAM:主要用作主存，也就是内存。

ROM:

- EPPROM:全称是“电可擦除可编程只读存储器”，即 Electrically Erasable Programmable Read-nly Memory。特点是可以随机访问和修改任何一个字节，具有较高的可靠性，比 Flash 寿命高，但电路复杂/成本也高。
- Flash:
    - **nor flash**:数据线和地址线分开，可以实现 ram 一样的随机寻址功能，可以读取任何一个字节但是擦除仍要按块来擦。因此在单片机中可以用 flash 存放程序以及常量，而不用加载到内存中，这点区别于 Nand Flash
    - **nand flash**:同样是按块擦除，但是数据线和地址线复用，不能利用地址线随机寻址。读取只能页来读取。（nandflash 按块来擦除，按页来读，norflash 没有页），由于 nandflash 引脚上复用因此读取速度比 nor flash 慢一点，但是擦除和写入速度比 nor flash 快很多。nand flash 内部电路简单，因此数据密度大，体积小，成本也低。因此大容量的 flash 都是 nand 型的。
    - **SSD**:固态硬盘是大部分是基于 NAND FLASH 的，包括硬盘控制器 +FLASH

27.c/c++ 的源文件生成可执行文件的过程

生成可执行文件的基本过程

预处理

预处理器主要处理源文件中以“#“开头的预编译指令，并执行文本替换等操作，最终将 ASCII 源码件翻译成一个 ACII 码的中间文件 main.i

编译

编译器将预处理完的 SCII 文件进行一系列的词法分析、语法分析、语义分析以及优化，并产生对象的 ASCII 码汇编语言文件 main.s，这一步是整个编译过程最复杂最核心的部分。

- 词法分析：源码文件被输入到扫描器，它运用一种类似于有限状态机的算法，将源代码的字符序列

割成一系列的记号（Token），并把这些记号代表的标识符存放到符号表（后续汇编器会利用这些符生成最终供链接器使用的符号表），将数字、字符串常量等放到文字表等，以备后续步骤使用，这些号一般被分为以下几类：

- 关键字
- 标识符
- 字面量（包括数字、字符串等）
- 特殊符号（如加好、等号）

● 语法分析：语法分析器会将扫描器产生的记号进行语法分析，从而产生语法树，它是由表达式为节的树结构。

● 语义分析：主要完成诸如类型的匹配，类型的转换等工作，比如赋值时的强制类型转换，给浮点数值一个指针的报错的。前一步的语法分析只是完成了对表达式的语法层面的分析，但它并不了解这个句是否真正有意义，比如 C 语言的两个指针做乘法是没有实际意义的，但是却符合语法要求。编译器分析的语义是静态语义，也就是编译器可以确定的语义，相对的动态语言只能在运行期才能确定。

● 源码优化：比如有些表达式的求值结果在编译期就能确定，这段代码就会被优化掉。还会优化诸如址方式，删除多余指令等操作。

汇编

汇编器的主要工作是将汇编代码转变成机器可以执行的指令，将 ASCII 码汇编文件转换成二进制文件每一个汇编语句几乎都对应一条机器指令，它会根据汇编指令和机器指令的对照表一一翻译。
当汇编器生成一个目标文件（目标模块）时，它并不知道数据和代码最终将放在内存中的什么位置，也不知道这个模块引用的任何外部定义的函数或者全局变量的位置，所以无论何时汇编器遇到对最终置位置的目　标引用，他就会生成一个**重定位条目**，告诉链接器在将目标文件合并成可执行文件如何修改这个引用。同时汇编器会利用编译器输出到汇编语言.s 文件中的符号，**构造符号表**。

链接

链接是将各种代码和数据片段收集并组合成为一个单一文件的过程，这个文件可被加载到内存并执行。

链接可以在不同的过程中执行：

- 执行于编译时，也就是在源代码被翻译成机器代码时
- 执行于加载时，也就是程序被加载器加载到内存并执行时
- 执行于运行时，也就是由应用程序来执行

链接器的存在使得源代码的分离编译成为可能，而不必将一个大型的应用程序组织为一个巨大的源文，而是可以把它分解为更小、更好管理的模块，并可以独立修改和编译这些模块，当更改某个模块（文件）中的部分代码时，只需要重新编译这个模块并重新链接即可，而不必重新编译其他文件模块。

链接器主要完成两个任务：

- **空间与地址分配**
  - 扫描所以输入的目标文件，获取每个文件各个段的长度、属性和位置，　**并将所有相同段合并**
  - 将所有目标文件中所有的　**符号定义**和**符号引用**收集起来，**统一放到全局符号表**
- **符号解析与重定位**

● **符号解析：**所有的函数名、变量等符号在编译阶段被解析成符号，并存放于每个文件的符号表，**符号解析的目的就是将每个符号引用正好和一个符号定义关联起来。**

● **重定位：**编译器和汇编器生成从地址 0 开始的代码和数据，其中每个目标文件都是这样。**一步会为每个符号分配运行时地址，链接器通过把每个符号定义与一个内存位置关联起来，然后找到有引用这个符号的位置，并使他们指向这个内存位置，这个过程就是重定位。**

static 修饰的变量符号在链接时，无法被不同的目标文件解析，这也就是 static 修饰的变量只能在本件中使用的原因

重定位时，将合并输入模块，并为每个符号分配运行时地址，重定位分两步组成：

● **重定位节和符号定义**：**由于待链接的目标文件总是有多个，而每个目标文件的文件结构都是几乎同的，所以这一步会把每个文件中相同的段进行合并，在最终的输出文件中生成一个合并完的新的聚段，比如所有目标文件都有.data 段，那就把这些.data 聚合在一块。然后链接器将运行时内存地址给新的聚合段，赋值给输入模块定义的每个段，以及赋给输入模块定义的每个符号，注意，这一步值定义处的符号，因此程序中的**每条指令和全局变量**都有唯一的运行时内存地址了。

● **重定位段中的符号引用：链接器修改代码段和数据段中对每个符号的引用，使得他们指向正确的运时地址。而这一步依赖于目标文件中汇编器产生的叫做重定位条目的数据结构**

重定位条目的详细指令，以及最终的符号表都是在编译和汇编阶段产生的，链接器不加甄别地执行这的重定位

所以说其实大部分的工作都是在编译和汇编阶段完成的。

对目标文件的理解

目标文件有三种形式：

● 可重定位的目标文件：由编译器和汇编器生成，包含二进制代码和数据，在链接时可与其他可重定目标文件合并起来，创建一个可执行文件

● 可执行目标文件：由链接器产生，包含二进制代码和数据，它可被直接加载到内存并执行。

● 共享目标文件：一种特殊类型的可重定位目标文件，可以再加载或者运行时被动态地加载进内存并接。

可重定位的目标文件纯粹是字节块的集合，这些块中，有些包含程序代码，有些包含程序数据，而其的则包含引导链接器和加载器的数据结构。链接器按照目标文件中的这些引导信息将这些块连接起来确定被连接块的运行时位置，并修改代码和数据块中的各种位置。

目标文件是按照特定的目标文件格式组织起来的，Linux 下使用可执行可链接格式 ELF

28.可执行文件如何被夹在到内存中执行

执行一个可执行程序化，该程序会被送给常驻在内存中的加载器，任何程序都可以调用 execve 函数调用加载器。加载一个程序的过程也就是创建一个进程的过程：

● 为进程分配一个 **独立的虚拟地址空间**，实际是创建下一步映射函数所需要的数据结构。

● 加载器读取可执行文件的 **文件头部信息**，建立虚拟地址空间与可执行文件的**映射关系**（指将一个拟内存区域与一个磁盘上的对象关联起来，也就是建立页表目录，此时页映射的地址是虚拟地址和磁

上的地址。比如第一次访问某个变量，会触发缺页中断，系统分配真实的物理页，并将当前映射条目的磁盘处的内容拷贝到物理页中）。虚拟空间中的堆区和栈区被初始化为 0，数据区和代码区被初始为可执行文件中的对应内容。

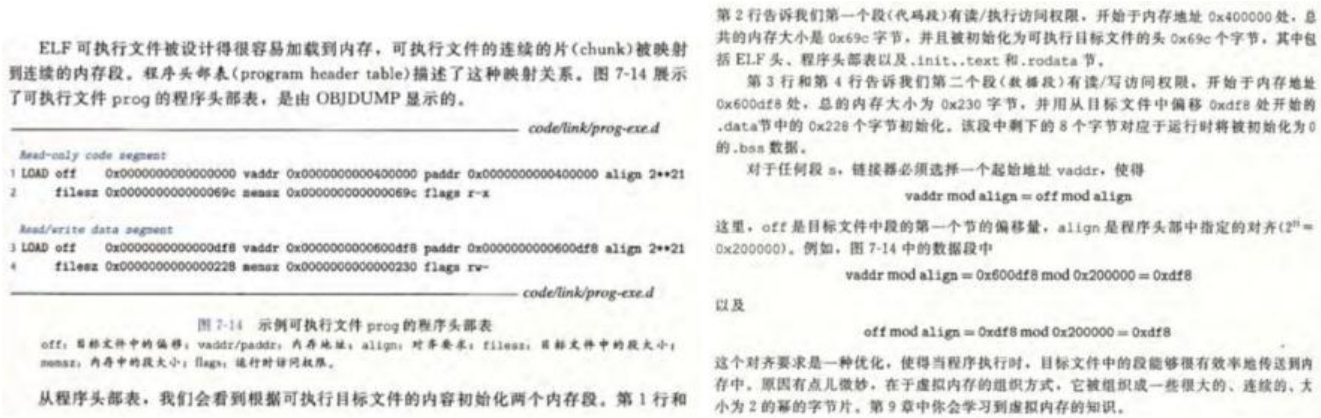
- 加载器跳转到程序的 **入口点**，根据引导执行启动函数，初始化执行环境，调用用户层的 main()函。

各个符号的虚拟地址在链接时已经确定好，第二部的建立映射关系的过程，实际就是建立页表的过程此时页表中的条目存储了分配好的虚拟内存和磁盘上的对象的映射。

注意：虽然链接时确定了符号的虚拟地址，但是这个地址是相对的，每次被加载到内存中时，这个实的绝对地址是会变的，这也是建立映射的一部分。这一过程叫做**地址空间布局随机化**，也就是每次加后，程序的各个段的绝对地址是不同的，目的是防止恶意的黑客共计行为。

注意：在这个过程中，**除了头部的引导信息，在加载过程中没有任何从磁盘到内存的数据复制。数据前都在磁盘上**。

直到 CPU 引用虚拟内存中被映射的虚拟页才会触发页错误，此时操作系统利用的它的页面调度机制动将页面从磁盘传送到内存



ELF 可执行文件被设计得很容易加载到内存，可执行文件的连续的片(chunk)被映射到连续的内存段。程序头部表(program header table)描述了这种映射关系。图 7-14 展示了可执行文件 prog 的程序头部表，是由 OBJDUMP 显示的。

code/link/prog-exe.d

```
Read-only code segment
1 LOAD off    0x0000000000000000 vaddr 0x0000000000400000 paddr 0x0000000000400000 align 2**21
2     filesz 0x000000000000069c memsz 0x000000000000069c flags r-x

Read/write data segment
3 LOAD off    0x0000000000000df8 vaddr 0x0000000000600df8 paddr 0x0000000000600df8 align 2**21
4     filesz 0x0000000000000228 memsz 0x0000000000000230 flags rw-
```

code/link/prog-exe.d

图 7-14 示例可执行文件 prog 的程序头部表

off：目标文件中的偏移；vaddr/paddr：内存地址；align：对齐要求；filesz：目标文件中的段大小；memsz：内存中的段大小；flags：运行时访问权限。

从程序头部表，我们会看到根据可执行目标文件的内容初始化两个内存段。第 1 行和第 2 行告诉我们第一个段(代码段)有读/执行访问权限，开始于内存地址 0x400000 处，总共的内存大小是 0x69c 字节，并且被初始化为可执行目标文件的头 0x69c 个字节，其中包括 ELF 头、程序头部表以及 .init、.text 和 .rodata 节。

第 3 行和第 4 行告诉我们第二个段(数据段)有读/写访问权限，开始于内存地址 0x600df8 处，总的内存大小为 0x230 字节，并用从目标文件中偏移 0xdf8 处开始的 .data节中的 0x228 个字节初始化。该段中剩下的 8 个字节对应于运行时将被初始化为 0 的 .bss 数据。

对于任何段 s，链接器必须选择一个起始地址 vaddr，使得

$$\text{vaddr} \bmod \text{align} = \text{off} \bmod \text{align}$$

这里，off 是目标文件中段的第一个节的偏移量，align 是程序头部中指定的对齐(2^{21} = 0x200000)。例如，图 7-14 中的数据段中

$$\text{vaddr} \bmod \text{align} = 0\text{x}600\text{df}8 \bmod 0\text{x}200000 = 0\text{xdf}8$$

以及

$$\text{off} \bmod \text{align} = 0\text{xdf}8 \bmod 0\text{x}200000 = 0\text{xdf}8$$

这个对齐要求是一种优化，使得当程序执行时，目标文件中的段能够很有效率地传送到内存中。原因有点儿微妙，在于虚拟内存的组织方式，它被组织成一些很大的、连续的、大小为 2 的幂的字节片。第 9 章中你会学习到虚拟内存的知识。

29.C/C++/Linux 内存模型

一个写好的程序最终会被链接成一个可执行文件，在文件中会记录这个程序应该如何进行内存映射，就是**将一个虚拟内存区域与一个磁盘上的对象关联起来**，每个变量和代码的具体的虚拟地址是多少，及其他信息。当这个可执行文件被加载到内存中后，它便成为了一个进程，会被操作系统分配各种资，其中就包括一个完整的、独立的虚拟内存空间，32 位系统一般为 4G，而 64 位系统最大支持 2^4 =248T 的内存空间。

内存模型指的就是这个虚拟内存空间如何分配，右图主要分为几个部分：

- 内核内存区：其中包括共享和私有部分，共享部分是所有进程都会使用到的内核代码和数据结构的源，他被映射到所有用户共享的物理内存页面，私有部分是维护当前进程的各种数据结构，包括内核上下文信息、进程控制块 PCB 数据结构、记录虚拟地址空间组织的数据结构等。
- 用户栈：用于维护函数调用的上下文，一般向下增长，在寄存器中拥有专门的寄存器，运行时创建保存嵌套的函数调用信息、以及属于函数的局部变量等数据。
- 共享库内存映射区：有些代码是所有进程都可能会使用的，比如各种库函数，因此没必要每个进程自保存这部分代码，所以可以通过共享库内存映射区，映射到所用用户共享的物理内存页。

- 堆区： 用于分配程序再运行过程中动态分配的内存。
- 数据段：包括初始化数据和未初始化的全局变量
- 代码段：具有只读属性，包括程序代码和常量数据。

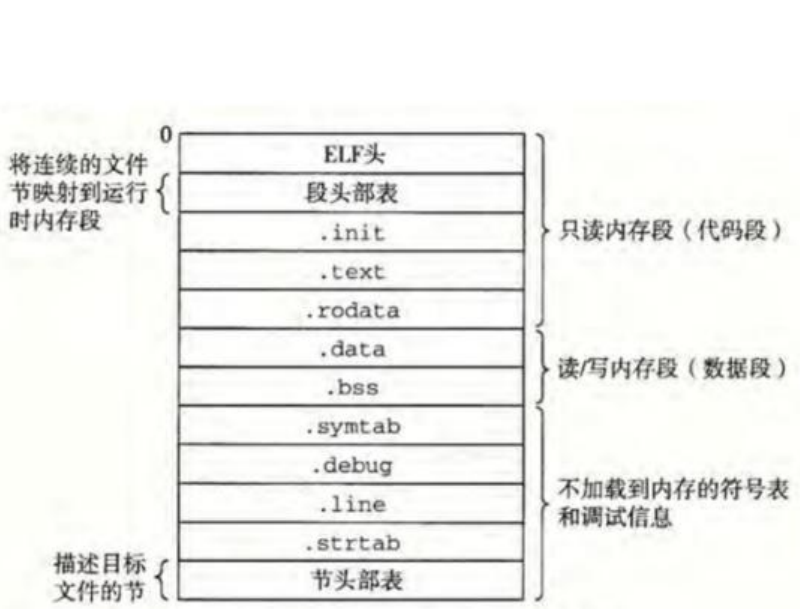


图 7-13 典型的 ELF 可执行目标文件

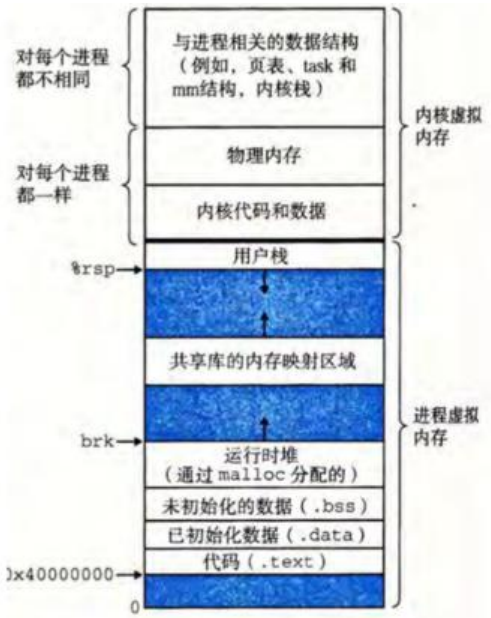


图 9-26 一个 Linux 进程的虚拟内存

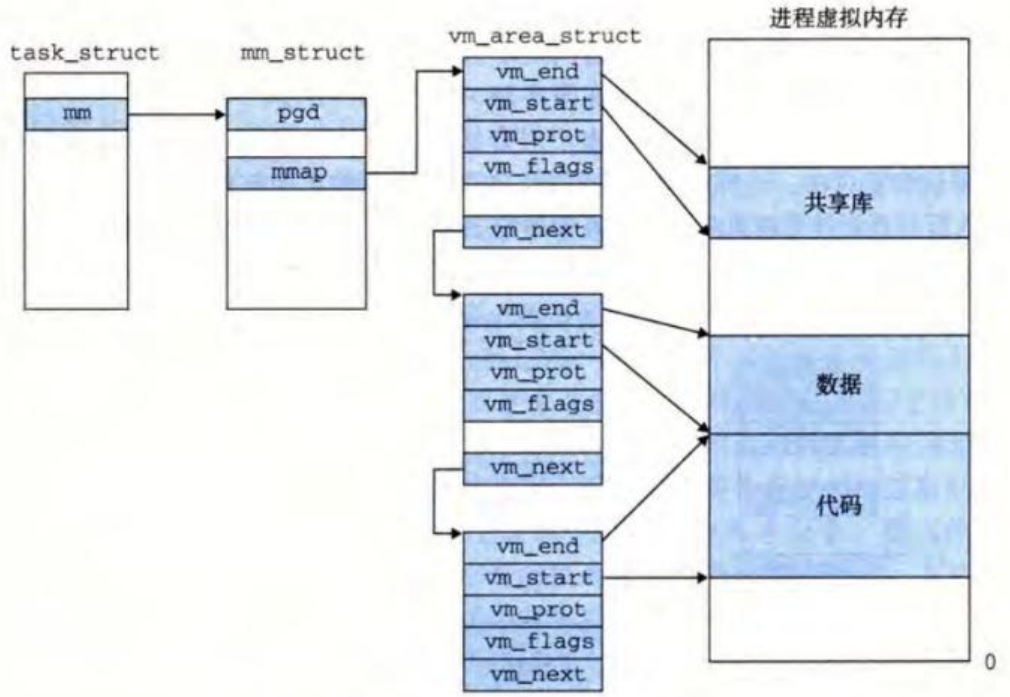


图 9-27 Linux 是如何组织虚拟内存的

内核为每个进程维护一个叫做任务结构的数据数据结构，它描述了虚拟内存的当前分布状态，也就是CB，进程控制块。

通过程序查看各个变量的存储位置

```
#include <stdio.h>
#include <string.h>
```

```
//初始化的全局变量
int  g_var1   = 10;
int g_var2 = 89;
//未初始化的全局变量
int  g_var3   ;
int g_var4;
//全局 const 修饰的常量
const int g_var5 = 3;
const int g_var6 =8;
//分别用普通全局变量和静态全局变量初始化的 const 变量
const int g_var7 =g_var2;
const int g_var8 =g_var6;
//初始化的 static 变量
static int g_static_var = 11;
//未初始化的全局 static 变量
static int g_static_var1 ;
//全局字符常量
const char *gp_str = "helloworld";
int main()
{
    //局部整形变量未初始化和初始化的
    int i;
    int i2;
    int i3 = 5;
    //局部静态变量
    static int l_static_var = 12;
    //局部字符常量
    const char *lp_str = "bye-bye";
    //局部 const 常量
    const int c_i = 2;
    const int *cp = &c_i;
    printf("g_var1       = %p\n", &g_var1);
    printf("g_var2       = %p\n", &g_var2);
    printf("g_var3       = %p\n", &g_var3);
    printf("g_var4       = %p\n", &g_var4);
    printf("g_var5       = %p\n", &g_var5);
    printf("g_var6       = %p\n", &g_var6);
    printf("g_var7       = %p\n", &g_var7);
    printf("g_var8       = %p\n", &g_var8);
    printf("g_static_var = %p\n", &g_static_var);
    printf("g_static_var1 = %p\n", &g_static_var1);
    printf("gp_str       = %p\n", gp_str);
    printf("g_static_var = %p\n", &g_static_var);
    printf("g_static_var1 = %p\n", &g_static_var1);
    printf("gp_str       = %p\n", gp_str);
    printf("-----------------------------\n");
    printf("i       = %p\n", &i);
    printf("i2       = %p\n", &i2);
    printf("i3       = %p\n", &i3);
    printf("l_static_var = %p\n", &l_static_var);
    printf("lp_str       = %p\n", lp_str);
    printf("c_i       = %p\n", &c_i);
    printf("cp       = %p\n", &cp);
    printf("&c = %p,c = %d,*p = %d,*p2 = %d",&c,c,*p,*p2);
```

```
    return 0;
}
```

运行结果：

```
GNX @ VM-24-14-centos in ~/workspace/APUE on git:master x [10:29:04]

$ ./memory<br />//初始化的全局变量<br />g_var1       = 0x601048
g_var2       = 0x60104c
//未初始化的全局变量
g_var3       = 0x601068
g_var4       = 0x60106c
//全局 const 修饰的常量
g_var5       = 0x400ae8
g_var6       = 0x400aec
//分别用普通全局变量和静态全局变量初始化的 const 变量
g_var7       = 0x601068
g_var8       = 0x400a20
//初始化的 static 变量
g_static_var = 0x601050
//未初始化的全局 static 变量
g_static_var1 = 0x601070
//全局字符常量
gp_str       = 0x400960
//局部整形变量未初始化和初始化的
i       = 0x7ffff9593564
i2       = 0x7ffff9593560
i3       = 0x7ffff959355c
//局部静态变量
l_static_var = 0x601060
//局部字符常量
lp_str       = 0x40096b
 //局部 const 常量
c_i       = 0x7ffff9593558
cp       = 0x7ffff9593550
```

另外要注意！！！！：

当用非常量值初始化 const 变量时，其属性会发生一些变换，g_var7 = 0x601068

g_var8 = 0x400a20，观察这两个 const 修饰的全局变量，g_var7 是用普通全局变量初始化的，则个 const 存储在了.data 段，g_var8 用另一个 const 初始化，则正常的存储在.rodata 段。说明 g_va 7 退化成了 c 语言中的 const。**这是因为由于 const 常量没有用一个常量值初始化，所以编译器无法定其值，所以等到程序最终运行后才能确定这个用来初始化 const 的初值是什么，此时由于没有在编期值替换，所以可以像 c 语言中那样通过指针间接修改。**

30.虚拟内存是什么，有什么好处?

什么是虚拟内存

所谓虚拟内存是指利用mmu，即**内存管理单元**来管理物理内存，并建立**物理内存和虚拟地址之间的射。**cpu直接访问的是虚拟内存，并由mmu来讲虚拟地址翻译成真实的物理地址，也就是在cpu和物内存间加了一个**中间层**，做了一种隔离。它解决了早期进程直接访问物理内存带来的诸多问题

虚拟内存的好处

1. 高效地使用主存，虚拟内存可以使得进程的 **运行内存超过物理内存大小**，因为程序运行符合局部原理，CPU 访问内存会有很明显的重复访问的倾向性，对于那些没有被经常使用到的内存，我们可以它换出到物理内存之外，比如硬盘上的 swap 区域

2. 它为每个进程提供了一致的的地址空间，从而简化了内存管理。

● **简化链接：**独立的地址空间允许每个进程的内存映像使用相同的基本格式，而不用管实际的码和数据放在物理内存的何处。体现为统一的 Linux 进程内存模型，内核区、栈区、共享库数据区、区、数据区、代码区这种相对位置不变的基本格式。这样的一致性极大地简化了链接器的设计和实现因为位每个符号重定位正是发生在链接的过程。

● **简化加载：**虚拟内存还使得容易向内存中加载可执行文件和共享对象文件。比如要把目标文中的.text 和.data 段加载到一个新创建的进程中，Linux 加载器为代码和数据段分配虚拟页，并标记未被缓存的，并将页表条目（PTE）指向目标文件中适当的位置。要注意，加载器从不从磁盘到内存制任何数据。复制的过程发生在 CPU 初次要访问某个虚拟地址时，虚拟内存系统会按照需要自动调数据页。

● **简化共享：**操作系统通过将不同进程中适当的虚拟页面映射到相同的物理页，从而使多个进共享这部分代码的一个副本。

● **简化内存分配：**当用户进程需要在堆区动态分配一块内存时，操作系统分配适当长度的连续虚拟内存页面，并将他们映射到物理内存中任意位置，只需要长度相同，不需要连续的物理内存。

3. **由于每个进程只能看到自己的虚拟地址空间，因此它保护了每个进程的地址空间不被其他进程破坏。**

● 每次 CPU 生成一个地址时，MMU 都会读一个 PTE，所有通过在 PTE 上添加额外的许可位来控对一个虚拟页面的内容访问十分简单。

● 一旦一条访问地址的指令违反了这些许可条件，CPU 就会触发一个一般保护故障，也就是段错
。

要充分理解虚拟内存作为工具的概念，他将主存变成了磁盘上某个地址空间的高速缓存。

虚拟地址被组织成一个由存放在磁盘上的 N 个连续的字节大小的单元组成的数组。每个字节都有一唯一的虚拟地址，作为到数组的索引。磁盘上数组的内容被缓存在主存中。

磁盘上的**数据被分割成页为单位的块**，每个页大小的典型值为 4KB，称为虚拟页（VP），对应主存的页就是物理页（PV）。每个虚拟页有三种状态：

● 未分配

● 缓存的：当前页已经缓存在物理内存中

● 未缓存的：已经分配了数据，但是没有缓存在物理内存中。

●

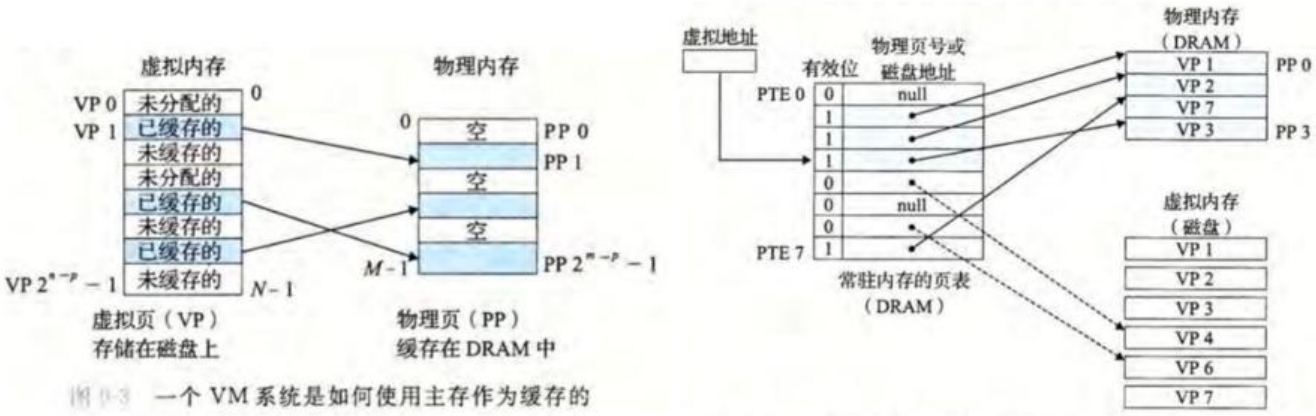


图 9-3　一个 VM 系统是如何使用主存作为缓存的

内核为每个进程维护一个**页表**，它是**页表条目（PTE）的数组。页表基址寄存器够找到每个进程的页表，再通过虚拟地址找到这个 PET，而每个 PTE 记录了该虚拟地址对应的物理位置**，或已经缓存在主存中，或只在磁盘中。

这个 CPU 拿着虚拟地址寻找物理地址的过程称为**地址翻译**，是在 CPU 的 **MMU** 中处理完成的，速非常快。

31.虚拟地址的寻址过程

单级页表情况下：

CPU 通过虚拟地址来访问主存，进程中的各个变量以及函数等符号的虚拟地址是在链接的过程中，链器通过汇编器生成的**重定位条目**的数据结构，为各个符号分配具体的虚拟地址，当 CPU 需要访问某数据时，他的主要流程为：

1. CPU 将要访问的虚拟地址送给 MMU，对应图 b 中的第一步。

2. MMU 解析虚拟地址：

- 通过 **页表基址寄存器（PTBR）找到页表基址（第一级页表），再解析出虚拟地址中的的拟页号（VPN）**，也就是所找页在页表中的偏移，虚拟页号有四部分组成，每部分分别表示了在某一页表中的偏移。注意：这个基址 + 偏移表示的是对应 PTE 是物理地址。
-

图 9-25 给出了 Core i7 MMU 如何使用四级的页表来将虚拟地址翻译成物理地址。36 位 VPN 被划分成四个 9 位的片，每个片被用作到一个页表的偏移量。CR3 寄存器包含 L1 页表的物理地址。VPN 1 提供到一个 L1 PET 的偏移量，这个 PTE 包含 L2 页表的基地址。VPN 2 提供到一个 L2 PTE 的偏移量，以此类推。

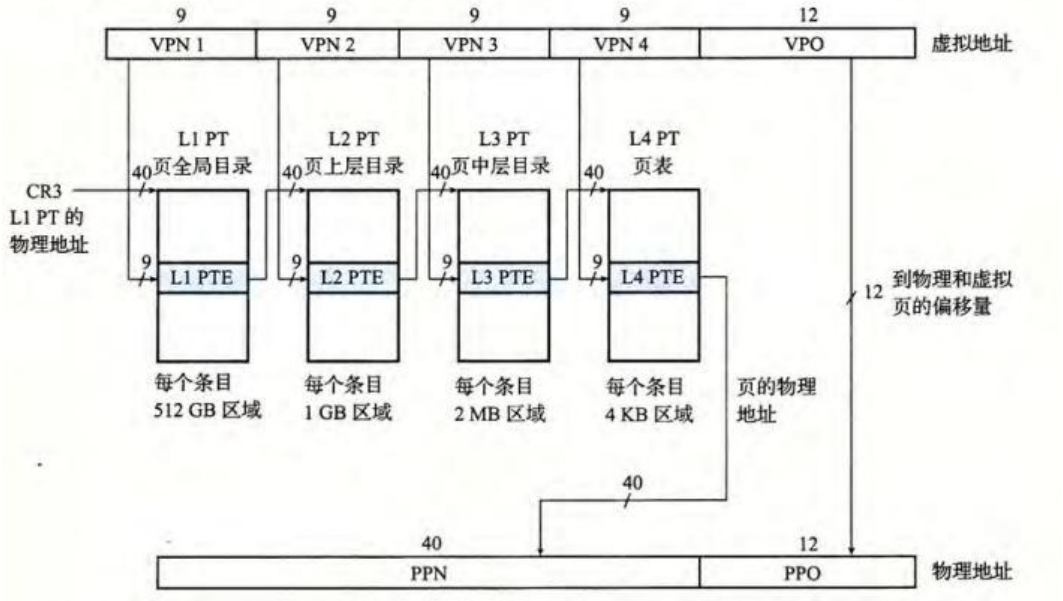


图 9-25 Core i7 页表翻译(PT：页表，PTE：页表条目，VPN：虚拟页号，VPO：虚拟页偏移，PPN：物理页号，PPO：物理页偏移量。图中还给出了这四级页表的 Linux 名字)

- MMU 通过虚拟页号，先到 **TLB(虚拟寻址缓存)**中找这个 PTE， **若命中直接返回，加速了 TE 的查找**，对应 b 中第二步。

若 TLB 未命中，**则 MMU 再向高速缓存/主存请求这个 PTE**，高速缓存/主存返回这个 PTE 到 TLB，更新 TLB，这可能会覆盖一个已经存在的 PTE。最后 MMU 拿到这个页表条目（PTE）。对应 b 中第三步和第四步

- **检查各个许可位**，若命中，且有权限，则根据这个页表条目（PTE）拿到真实的物理页地址

- 若未命中，则 MMU 触发 **缺页异常**，执行内核中的**缺页异常处理函数。**

- 缺页处理程序确定出物理内存中的牺牲页，如果这个页面被修改过，则把它 **换出**到内存，没修改过，则说明硬盘中的数据和他一致，所以无需移动操作，直接丢弃。

- 缺页处理程序 **换入**新的页面，并更**新内存中的 PTE。**

- 缺页处理程序返回到原来的进程， **再次执行导致缺页的指令，也就是从第一步开始**，即 CP 将要访问的虚拟地址送给 MMU

- MMU 再解析虚拟地址中的 **虚拟页面偏移（vpo）找到该数据的真实物地址。**

注意：虚拟页面偏移和物理页面偏移是一一对应的，地址相等。但是虚拟页地址和真实物理地址不同一个物理页底子可以对应多个虚拟页地址。另外页表是常驻在内存中的，也可能在高速缓存中，由于个进程都有一个页表，因此那个页表正被缓存在高速缓存中不一定。

3. MMU 解析出数据对应真实物理地址后，向高速缓存/主存请求数据，对应图 b 中的第五步。

4. 高速缓存/主存直接返回数据给 CPU，对应图 b 中的第六步。

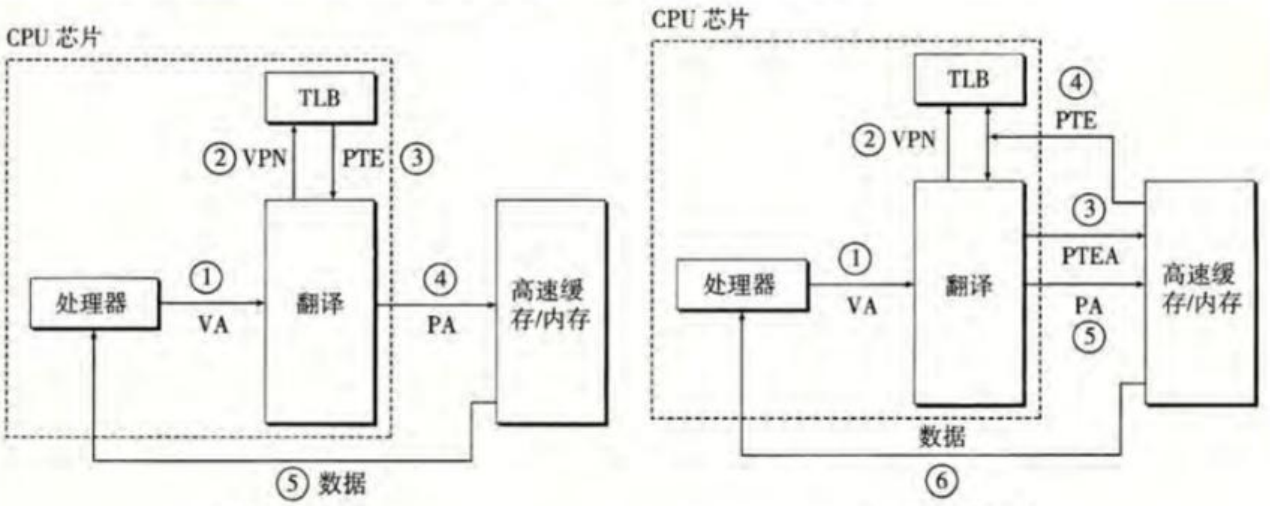


图 9-16　TLB 命中和不命中的操作图

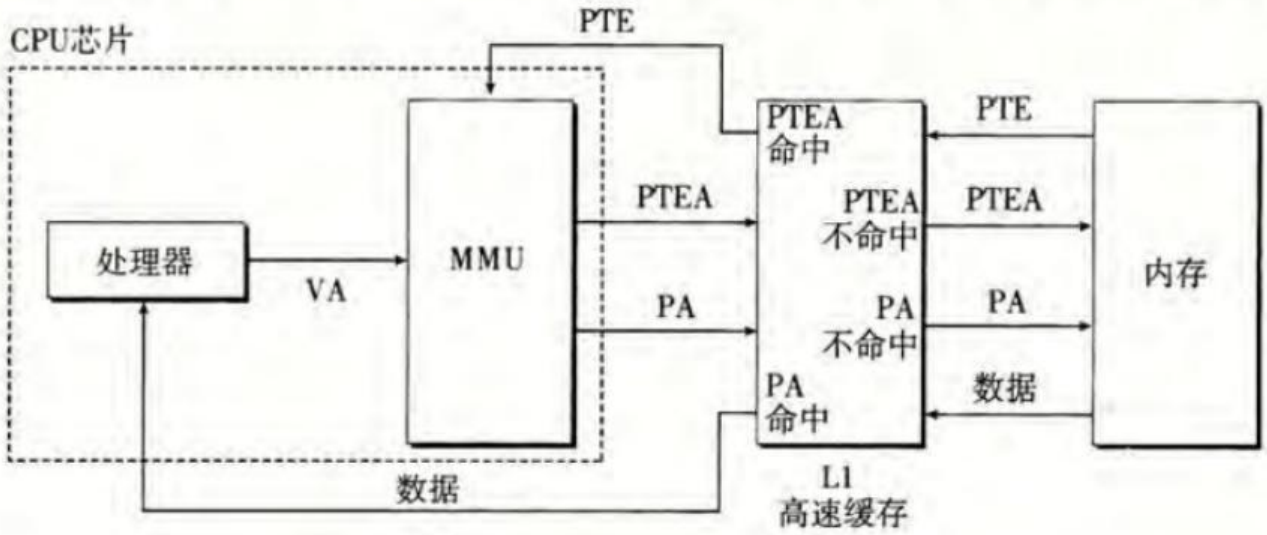


图 9-11　将 VM 与物理寻址的高速缓存结合起来(VA：虚拟地址。PTEA：页表条目地址。PTE：页表条目。PA：物理地址)

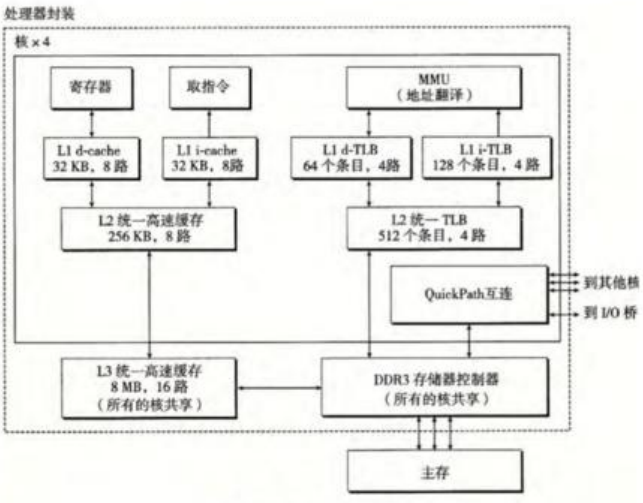


图 9-21　Core i7 的内存系统

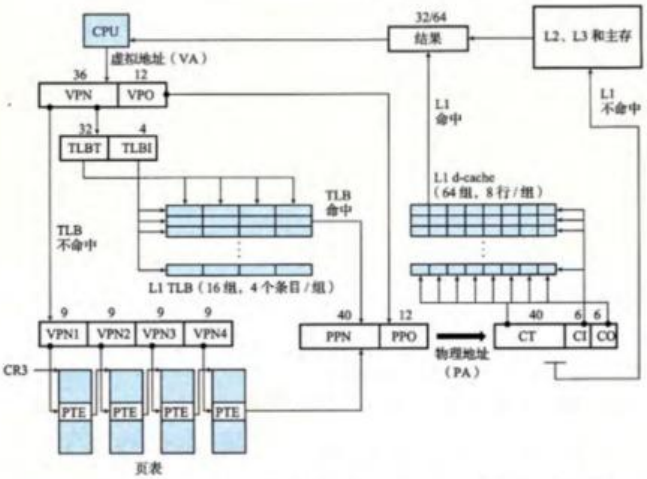


图 9-22　Core i7 地址翻译的概况。为了简化，没有显示 i-cache、i-TLB 和 L2 统一 TLB

32.多级页表存在的意义是什么？大概的工作流程是什么？

多级页表的目的就是为了**压缩页表所占空间的大小。**

对于 64 位的虚拟地址空间，加上页的大小为 4KB，页表中每个 PTE 位 8Byte，那么页表的大小为 **264/4KB*8byte=33553332GB。**这个大小是惊人的，因此引入多级页表来压缩大小是有必要的。

虚拟地址主要包含两部分：

- 虚拟页号
- 页内偏移

单级页表中，虚拟页号直接就是页表中的索引。但是多级页表中的虚拟页号会被多划分为 K 个部分（拟页号 0……虚拟页号 i，0<=i<=k）,虚拟页号 i 对应该该虚拟地址在第 i 级页表中的索引。**第一级表会常驻内存中**，但是若某级页表中的某个 PTE 为空时，后续的页表就不存在，因此不占用空间，这就减少了空间的占用。

33.c/c++ 中常见的与内存有关的错误

间接引用坏指针

在进程的虚拟地址空间中有较大的洞，没有映射到任何有意义的数据。如果我们试图间接引用一个指这些洞的指针，那么操作系统就会以段异常中止程序。而且，虚拟内存的某些区域是只读的。试图写些区域将会以保护异常中止这个程序。

读未初始化的内存

虽然 bss 内存位置（诸如未初始化的全局 C 变量）总是被加载器初始化为零，但是对于堆内存却并是这样的。一个常见的错误就是假设堆内存被初始化为零。正确的实现方式是显式地将变量设置为零或者使用 calloc

引用指针，而不是它所指向的对象

如果不太注意 C 操作符的优先级和结合性，我们就会错误地操作指针，而不是指针所指向的对象。

引用不存在的变量

没有太多经验的 c 程序员不理解栈的规则，有时会引用不再合法的本地变量。

引用空闲堆块中的数据

一个相似的错误是引用已经被释放了的堆块中的数据

引起内存泄漏

内存泄漏是缓慢、隐性的杀手，当程序员不小心忘记释放已分配块，而在堆里创建了垃圾时，会发生种问题。

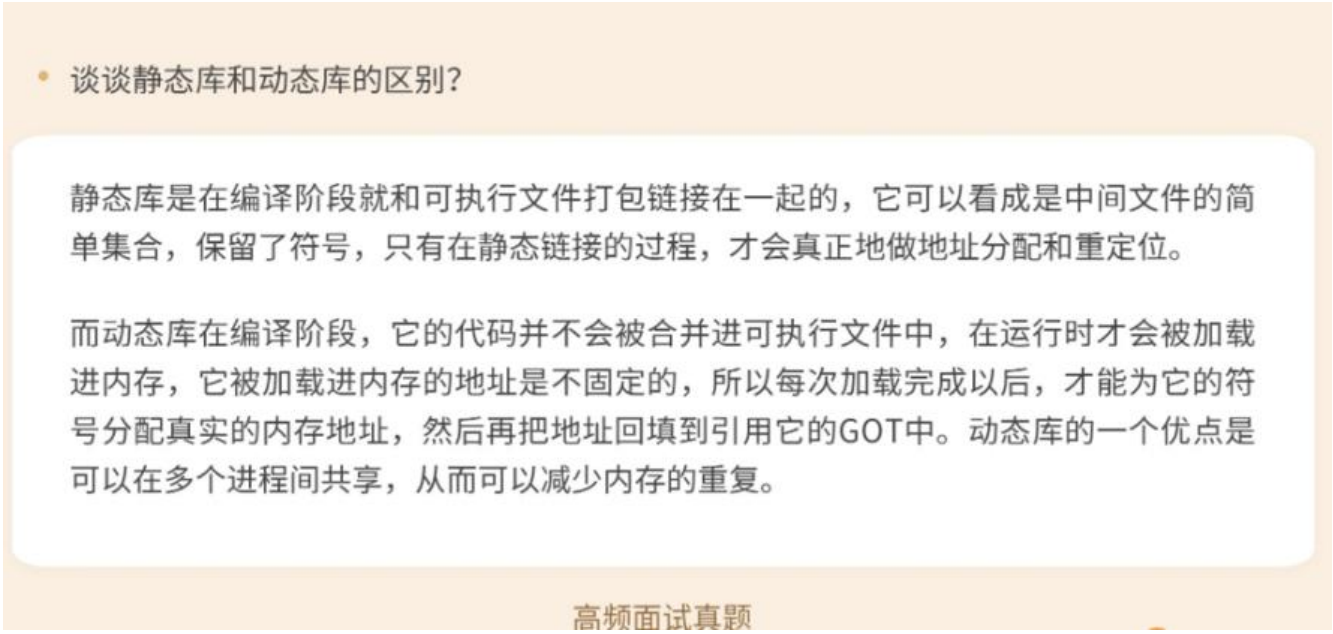
34.什么是内存映射、共享内存、写时拷贝技术?

内存映射：创建虚拟内存空间时，将一个虚拟内存区域与一个磁盘上的对象关联
来。

共享内存：允许同一个物理页在不同的应用程序间共享。也就是同一个物理页映射
不同进程的某个虚拟地址中。

写时拷贝技术：该技术允许两个进程以只读的方式共享同一段物理内存
前提是这段物理内存数据在各自页表中被设为**只读**，并且被操作系统**标记为写时拷贝**。一旦某个应用序对该内存区域**进行修改**，就会触发**缺页异常**，并调用对应的缺页异常处理函数。在该函数中，操作统发现当前的异常是因为进程向只读数据区写入数据，而且这段数据被标记为写时拷贝。于是，操作统会在物理内存中将缺页异常对应的物理页**重新拷贝一份**，并将新拷贝的物理页以**可读可写**的方式**重映射给触发异常的应用程序。**

35.静态库和动态库



动态链接中的GOT就是一个中间层，

里面存放跨模块的数据的地址，当然，可以在装载时动态填入。然后，共享对象指令中对跨模块数据访问，可以**通过GOT中的指针间接访问。**

这样的好处是，指令中的地址就从跨模块数据的地址，变成了got中指针的地址，而这个地址是相对码段确定的。

里面存放跨模块的数据的地址，当然，可以在装载时动态填入。然后，共享对象指令中对跨模块数据访问，可以通过GOT中的指针间接访问。

这样的好处是，指令中的地址就从跨模块数据的地址，变成了got中指针的地址，而这个地址是相对码段确定的。

动态链接的缺点

不过，细心的你也发现了，动态链接带来的**代价是性能的牺牲。**这里性能的牺牲主要来自于两个方面：

1. 每次对全局符号的访问都要转换为对 GOT 表的访问，然后进行 **间接寻址**，这必然要比直接的地访问速度慢很多；

2. 动态链接和静态链接的区别是将链接中 **重定位的过程推迟到程序加载时进行。**因此在程序启动的候，动态链接器需要对整个进程中依赖的 so 进行加载和链接，也就是对进程中所有 GOT 表中的符进行**解析重定位。**这样就导致了程序在**启动过程中速度的减慢。**

延迟绑定技术

为了避免在加载时就把 GOT 表中的符号全部解析并重定位，就需要采用计算机领域非常重要的一个想：Lazy。也就是说，把要做的事情推迟到必须做的时刻。对于我们当前的问题来说，将函数地址的定位工作一直推迟到第一次访问的时候再进行，这就是延迟绑定 (Lazy binding) 的技术。这样的话，于整个程序运行过程中没有访问到的全局函数，可以完全避免对这类符号的重定位工作，也就提高了序的性能。

只有用到的符号才会被重定位，这就是延迟绑定技术

36.c 和 c++ 中的 const 常量

const 修饰的局部变量在栈上分配空间

const 修饰的全局变量在只读存储区分配空间（修改将导致程序崩溃，报段错误，访问了虚拟内存的读区域）

c 语言中的 const 是只读变量，c++ 中的 const 才是真正的常量。

理解 const 的关键就是认识到什么是只读变量，什么事常量。

只读变量：仍然是个变量，只不过编译器约束其为只读，可通过指针间接修改。

常量：编译期间就可以确定这个符号的数值，则这个符号就表示一个常量，在编译期货预处理期值替换。

C 语言中的 const

const 修饰的变量是只读的，**本质还是变量**

因此 c 语言在的 const 变量，可以通过指针间接的修改其值，但 c++ 不可以

```
#include<stdio.h>
int main()
{
   const int c = 0;
   int* p = (int*)&c;
   *p = 5;
   printf("c = %d\n", c);
   return 0;
}**
```

*

程序中定义了局部 const 变量，存储在栈上，我们可以获取栈的地址来修改栈中的数据，C 语言中的 const 变量只是告诉编译器该变量不能出现在赋值符号的左边。

```
*$ gcc 2-1.c -o 2-1
$ ./2-1
c = 5*
```

可以看到，const 变量值依然可以改变。

C++ 中的 const

C++ 在 C 的基础上进行了优化

当碰见 const 声明时，若是用常量值初始化的，则在**符号表中放入常量，编译过程中使用常量直接以号表中的值替换。**若使用另一个变量初始化 const 常量，则编译期无法确定 const 常量值，推迟到运期确定，此时的 const 和 c 语言的 const 一样。

const 常量的判别规则：

- 只用字面量初始化的 const 常量才会进入符号表
- 使用其它变量初始化的 const 常量仍然是只读变量
- 被 volatile 修饰的 const 常量不会进入符号表
- 在编译期间不能直接确定初始值的 const 标识符，都被作为只读变量处理

编译过程中若发现下述情况则给对应的常量分配存储空间

- 对 const 常量使用了 extern，需要在其他文件中使用
- 对 const 常量使用了 & 操作符，要取地址

！！！注意：C++ 编译器虽然可能为 const 常量分配空间，但不会使用其存储空间中的值

如图所示，C++ 中用 const 定义常量时，放入符号表，使用时直接用符号表中的值替换

```
#include<stdio.h>
int main()
{
    const int c = 0;
```

```
    int* p = (int*)&c;
    *p = 5;
    printf("c = %d\n", c);
    printf("*p = %d\n", *p);
    return 0;
}
```

程序中使用了 &c，所以为 const 类型的变量 c 分配了空间，但是打印 c 变量时，使用的是符号表中变量 c，直接用 0 替换，改变了为 c 申请的内存中的值不改变符号表中的值。那分配的内存岂不是没用了嘛，其实这么做是为了兼容 C 语言的特性。

```
$ g++ 2-1.c -o 2-1
$ ./2-1
c = 0
*p = 5
```

C++ 中的 const 与宏

C++ 中的 const 常量类似于宏定义，都是替换，但又不相同

const 常量由**编译器处理**，编译器对 const 常量进行**类型检查**和作用域分析，编译器期间有优化代码步骤，这一步就将这些能在编译期确定的表达式直接用常量替换。

宏定义由预处理期处理，**单纯的文本替换**，没有作用域和类型的概念

```
#include<stdio.h>
void f()
{
    #define a 3
    const int b = 4;
}
void g()
{
    printf("a = %d\n", a);
    // printf("b = %d\n", b);
}
int main()
{
    const int A = 1;
    const int B = 2;
    int array[A + B] = {0};
    for (int i = 0; i < (A + B); i++)
    {
        printf("array[%d] = %d\n", i, array[i]);
    }
    f();
    g();
    return 0;
}
```

宏定义在预处理期替换，没有作用域检查，所以第 4 行定义的宏可以在第 9 行打印，const 常量由类检查和作用域检查，所以第 10 行打印会出错

第 16 行，C 语言中 const 局部变量存储在栈上，不是真正的常量，只有在运行期才知道数组大小，译器无法确定数组大小，所以会报错，C++ 中可以直接用符号表中的数值进行替换，可以确定数组小，不会报错。

先用 gcc 编译器编译，第 17 行数组没法初始化

```
$ gcc 2-2.c -o 2-2
2-2.c: In function ‘main’:
2-2.c:17:5: error: variable-sized object may not be initialized
    int array[A + B] = {0};
    ^~~
2-2.c:17:25: warning: excess elements in array initializer
    int array[A + B] = {0};
                       ^
2-2.c:17:25: note: (near initialization for ‘array’)
```

再用 g++ 编译器编译：

```
$ g++ 2-2.c -o 2-2
$ ./2-2
array[0] = 0
array[1] = 0
array[2] = 0
a = 3
直接用符号表中的数组替代 const 常量，可以在编译器确定数组大小
注意：
若 const 修饰的变量是用另一个变量赋予初值的，则 c++ 中的该 const 常量退化为 c 语言中的 const 常量。
这是因为由于 const 常量没有用一个常量值初始化，所以编译器无法确定其值，所以等到程序最终运后才能确定这个用来初始化 const 的初值是什么，此时由于没有在编译期值替换，所以可以像 c 语言那样通过指针间接修改。
int b = 3;
const int c = b;
int* p = (int*) & c;
*p = 9;
printf("c = %d,*p = %d",c,*p);

//运行结果：
c = 9,*p = 9
```

小结：

1. C 中的 const 是 **只读变量**，C++ 中的 const 才是真正意义上的**常量**
2. C++ 为了 **兼容** C 语言中的 const 特性，可能为 const 常量分配空间

常量是在预处理期间或者编译期间就可以确定的量，对引用常量的地方，可以直接进行值替换

32.C/c++ 的函数调用惯例

调用惯例是调用方和被调用方对于函数如何调用的一个明确的约定，只有双方都遵守同样的约定，函才能被正确地调用。如果不这样的话，函数将无法正确运行。假设有一个 foo 函数

```
int foo0(int n, float m)
{
    int a = 0, b = 0;
    ...
}
```

如果函数的调用方在传递参数时先压入参数 n，再压入参数 m，而 foo 函数却认为其调用方应该先压参数 m，后压入参数 n，那么不难想象 foo 内部的 m 和 n 的值将会被交换。再者如果函数的调用方定利用寄存器传递参数，而函数本身却仍然认为参数通过栈传递，那么显然函数无法获取正确的参数因此，毫无疑问函数的调用方和被调用方对于函数如何调用须要有一个明确的约定。

一个调用惯例一般会规定如下几个方面的内容：

- **函数参数的传递顺序和方式:**

函数参数的传递方式有很多种方式，最常见的一种是通过栈传递。函数的调用方将参数压入栈中，函自己再从栈中将参数取出。对于有多个参数的函数，调用惯例要规定函数调用方将参数压栈的顺序：从左至右，还是从右至左。有些调用惯例还允许使用寄存器传递参数，以提高性能。

- **栈的维护方式:**

在函数将参数压栈之后，函数体会被调用，此后需要将被压入栈中的参数全部弹出，以使得栈在函数用前后保持一致。这个弹出的工作可以由函数的调用方来完成，也可以由函数体本身来完成。

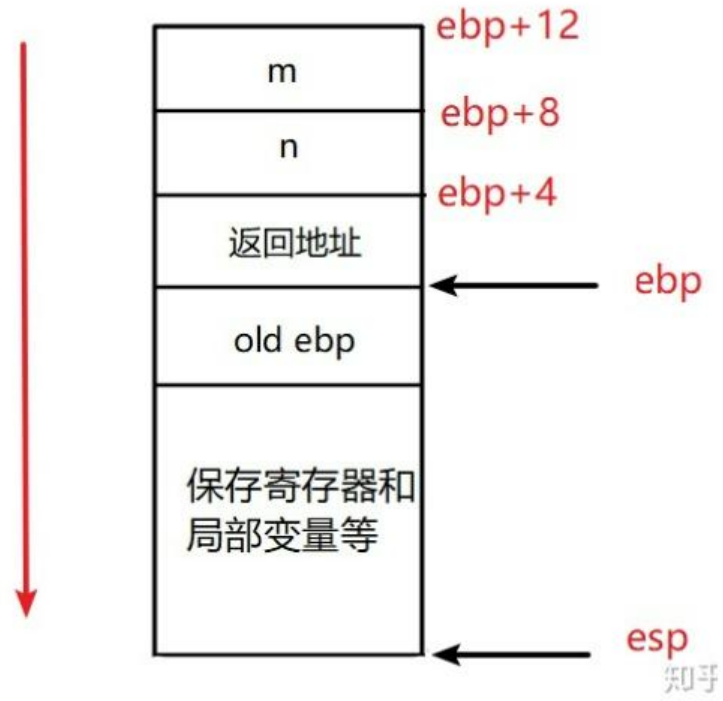
- **名字修饰的策略**

为了链接的时候对调用惯例进行区分，调用惯例要对函数本身的名字进行修饰。不同的调用惯例有不的名字修饰策略。

下表介绍了几项主要的调用惯例的内容。

调用惯例	出栈方	参数传递	名字修饰
cdecl	函数调用方	**从右至左**的顺序压参数入栈	下划线+函数名
stdcall	函数本身	**从右至左**的顺序压参数入栈	下划线+函数名+@+参数的字节数，如函数int func(int a,double b)的修饰名是_func@12
fastcall	函数本身	**头两个DWORD**（4字节）类型或者占更少字节的参数**被放入寄存器**，其他**剩下的**参数按**从右到左**的顺序压入栈	@+函数名+@+参数的字节数
pascal	函数本身	**从左至右**的顺序压参数入栈	较为复杂，参加pascal文档

C++ 自己还有一种特殊的调用惯例，称为 **thiscall**，专用于类成员函数的调用。其特点随编译器不同不同，在 VC 里是 this 指针存放于 ecx 寄存器，参数从右到左压栈，而对于 gcc、thiscall 和 cdecl 全一样，只是将 this 看作是函数的第一个参数。同时 thiscall 也是 C++ 成员函数缺省的调用约定



实例分析：

main()中调用出的代码，传入 4 个参数，其中三个普通变量，一个类对象.

通过汇编代码看出来，传入的参数，从右开始一次被压入栈中，因此通过栈来穿参数

```
    int a =3;
00263EA0  mov         dword ptr [a],3
    int b = 5;
00263EA7  mov         dword ptr [b],5
    int c;
    many d;
00263EAE  lea         ecx,[d]
00263EB1  call        many::many (0261244h)
    fun4(a,b,c,d);
00263EB6  lea         eax,[d]
00263EB9  push        eax
00263EBA  lea         ecx,[c]
00263EBD  push        ecx
00263EBE  mov         edx,dword ptr [b]
00263EC1  push        edx
00263EC2  mov         eax,dword ptr [a]
00263EC5  push        eax
00263EC6  call        fun4 (0261479h)
00263ECB  add         esp,10h
```

函数内的汇编：

首先将原来的 ebp，也就是栈基指针压栈，在设置新的 ebp，然后调整 esp 位置，为当前函数调用辟栈帧空间。

随后又将几个关键寄存器压栈，以备在当前函数使用者几个寄存器。

获取参数的方式就是通过 ebp 加偏移的方式。

```
//类的定义
class many {
public:
    int arry[10] = { 0 };
    char c[3];
};

//----------------------函数出的汇编-----------------------------
void fun4(int a, int b, int& c, many& d) {
//保存原来的栈基地址寄存区，方便函数退出时能返回到原来的函数中
00263820  push        ebp
00263821  mov         ebp,esp
//开辟栈帧空间
00263823  sub         esp,0C0h
//保存几个寄存器的值，以备在当前函数中重新使用跟他们
00263829  push        ebx
0026382A  push        esi
0026382B  push        edi
```

```
0026382C  mov         edi,ebp
0026382E  xor         ecx,ecx
00263830  mov         eax,0CCCCCCCCh
00263835  rep stos    dword ptr es:[edi]
00263837  mov         ecx,offset _77DAC90D_main@cpp (02710FCh)
0026383C  call        @__CheckForDebuggerJustMyCode@4 (0261500h)
//获取参数的方式就是通过ebp加偏移的方式
    a= 6;
00263841  mov         dword ptr [ebp+8],6
    d.arry[5] = 5;
00263848  mov         eax,4
0026384D  imul        ecx,eax,5
00263850  mov         edx,dword ptr [ebp+14h]
00263853  mov         dword ptr [edx+ecx],5
}
```

33.C++ 返回值优化技术之 RVO 和 NRVO

返回值优化的核心是:

直接将要返回的对象**创建在调用者的栈帧上**，避免了在被调函数内创建对象，返回时在把对象拷到调用者栈上，然后调用者在把自己栈上的临时对象拷贝到另一处，RVO 避免了这一系列多余的动。

- 如果返回值类型是基本数据类型，则直接通过存放在寄存器中进行返回的方式，
- 如果返回值类型的尺寸太大，C 语言在函数返回时会使用一个临时的栈上内存区域作为中转，结果是返回值对象会被拷贝两次。一次是拷贝到这个中转区域，一次是从中转区域拷贝个调用方的接收变。

RVO 针对返回一个匿名临时对象的优化:

```
A GetA()
{
    return A();
}
```

NRVO(named return value optimization、命名返回值优化)针对返回一个命名的局部变量对象:

```
A GetA()
{
    A o;
    return o;
}
```

RVO 优化：

```
#include <stdio.h>
class A
{
public:
    char cNum1;
    int    iNum2;
    int* pInt;
    A()
    {
        printf("%p construct\n", this);
    }
    A(const A& cp)
    {
        printf("%p copy construct\n", this);
    }
    ~A()
    {
        printf("%p destruct\n", this);
    }
};

A GetA()
{
    cout << "\n打印子函数第一个栈对象" << endl;
    A a;
    cout << "\n打印子函数第二个栈对象" << endl;
    A b;
    cout << "\n打印返回值对象" << endl;
    return A();
}
int main() {
    cout << sizeof(Test) << endl;
    cout << "\n打印main()函数第一个栈对象" << endl;
    A a;

    A b= GetA();

    cout << "\n打印main()函数第二个栈对象" << endl;
     A C;
    printf("退出主程序...\n");
    return 0;
}
```

在 g++ 和 vc6、vs 中，上述代码仅仅只会调用一次构造函数和析构函数，输出结果如下：

```
0x7ffe9d1edd0f construct
0x7ffe9d1edd0f destruct
```

在 g++ 中，加上 -fno-elide-constructors 选项关闭优化后，输出结果如下：

0x7ffc46947d4f construct // 在函数 GetA 中，调用无参构造函数 A()构造出一个临时变量 temp
0x7ffc46947d7f copy construct // 函数 GetA return 语句处，把临时变量 temp 做为参数传入并用拷贝构造函数 A(const A& cp)将返回值 ret 构造出来
0x7ffc46947d4f destruct // 函数 GetA 执行完 return 语句后，临时变量 temp 生命周期结束，调其析构函数A()
0x7ffc46947d7e copy construct // 函数 GetA 调用结束，返回上层 main 函数后，把返回值变量 ret 做为参数传入并调用拷贝构造函数 A(const A& cp)将变量 A a 构造出来
0x7ffc46947d7f destruct // A a = GetA()语句结束后，返回值 ret 生命周期结束，调用其析构函数A(

0x7ffc46947d7e destruct // A a 要离开作用域，生命周期结束，调用其析构函数~A()~

为了探究为什么 RVO 只调用一次构造函数，请看下面的输出：

NRVO 优化

g++ 编译器、vs2005 + Release(开启 O2 及以上优化开关)

修改上述代码，将 GetA 的实现修改成：

```
A GetA()
{
    A o;
    return o;
}
```

在 g++、vs2005 + Release 中，上述代码也仅仅只会调用一次构造函数和析构函数，输出结果如下

0x7ffe9d1edd0f construct
0x7ffe9d1edd0f destruct

g++ 加上 -fno-elide-constructors 选项关闭优化后，和上述结果一样：
0x7ffc46947d4f construct
0x7ffc46947d7f copy construct
0x7ffc46947d4f destruct
0x7ffc46947d7e copy construct
0x7ffc46947d7f destruct
0x7ffc46947d7e destruct

但在 vc6、vs2005 以下、vs2005 + Debug 中，没有进行 NRVO 优化，输出结果为：
18fec4 construct // 在函数 GetA 中，调用无参构造函数 A()构造出一个临时变量 o
18ff44 copy construct // 函数 GetA return 语句处，把临时变量 o 做为参数传入并调用拷贝构造数 A(const A& cp)将返回值 ret 构造出来
18fec4 destruct // 函数 GetA 执行完 return 语句后，临时变量 o 生命周期结束，调用其析构函数A(

18ff44 destruct // A a 要离开作用域，生命周期结束，调用其析构函数~A()

34.C++ 中 volatile、mutable 和 explicit 关键字的用法

volatile

定义变量的值是易变的，该关键字的作用是防止优化编译器把变量从内存装入 CPU 寄存器中。每次到这个变量的值的时候都要去重新读取这个变量的值，而不是读寄存器内的备份。多线程中被几个任共享的变量需要定义为 volatile 类型。

mutable

mutable 的中文意思是“可变的，易变的”，跟 constant（既 C++ 中的 const）是反义词。在 C++ 中，mutable 也是为了突破 const 的限制而设置的。被 mutable 修饰的变量，将永远处于可变的状，即使在一个 const 函数中。我们知道，如果类的成员函数不会改变对象的状态，那么这个成员函数般会声明成 const 的。但是，有些时候，我们需要在 const 函数里面修改一些跟类状态无关的数据成，那么这个函数就应该被

explicit

explicit 关键字用来修饰类的构造函数，被修饰的构造函数的类，不能发生相应的隐式类型转换，只以显示的方式进行类型转换，注意以下几点：

explicit 关键字只能用于类内部的构造函数声明上

explicit 关键字作用于单个参数，或者其他参数有默认值的构造函数

被 explicit 修饰的构造函数的类，不能发生相应的隐式类型转换

35.C++ 中 NULL 和 nullptr 区别

NULL 来自 C 语言，一般由宏定义实现，而 nullptr 则是 C++11 的新增关键字。在 C 语言中，NULL 被定义

为(void*)0,而在 C++ 语言中，NULL 则被定义为整数 0。编译器一般对其实际定义如下：

```
#ifdef __cplusplus
#define NULL 0
#else
#define NULL ((void *)0)
#endif
```

为什么 c 和 c++ 中的 NULL 定义不同呢?

因为在 c 中，void*类型的指针，可以赋值给其他类型的指针。但是在 c++ 中这是不合法的。

不过 void*类型的变量可以接收其他任何指针类型变量的赋值是合法的。

```
//下面的操作在c++中不合法
void* ptr = NULL;
int* ptr_2 = ptr;
```

所以 c++ 中的 NULL 实际是整数 0。

那为什么又要引入 nullptr 呢?

c++ 中有函数重载，而 c 中没有，因此在 c++ 用 NULL，即一个整数来表示“空指针”是有二义性！NULL 首先是个 int，但可以被隐士转换为指针。传入 NULL 会首先匹配到参数为 int 的函数重载这显然和我们的原意，即传入一个“指针”是违背的。

因此 C++11 中专门定关键字 nullptr 来表示空指针。nullptr 是个明确的指针类型，其次它可以被隐转换为 int。

另外若存在多个指针类型的重载，可以对 nullptr 强制转换成指定的指针类型，即 (char*)nullptr

```
void func(int x) {
    cout << "输出的是：func(int x)" << endl;
}
void func(void* y) {
    cout << "输出的是：func(void *y)" << endl;
}

int main() {

    func(NULL);
    func(nullptr);
    printf("退出主程序...\n");
    return 0;
}

输出：
输出的是：func(int x)
输出的是：func(void *y)
退出主程序...
```

C:\Users\GNX\source\repos\test\x64\Release\test.exe (进程 1664)已退出，代码为 0。
按任意键关闭此窗口. . .

36.智能指针

智能指针的用处

智能指针用来代替裸指针，防止内存泄漏。它是一个通过 RAII 方式管理的类，用来存储指向动态分对象的指针，负责自动释放动态分配的对象，防止堆内存泄漏。当类对象声明周期结束时，自动调用构函数释放资源。

四种只能指针的特点

unique_ptr

std::unique_ptr<T> ： 独占资源所有权的指针

简单说，当我们独占资源的所有权的时候，可以使用 std::unique_ptr 对资源进行管理——离开 uniqe_ptr 对象的作用域时，会自动释放资源。这是很基本的 RAII 思想。

unique_ptr 虽然名字叫指针，用起来也很像，但它实际上并不是指针，而是一个**对象。**

所以，不要企图对它调用 delete，它会自动管理初始化时的指针，在离开作用域时析构释放内存。

另外，它也没有定义加减运算，不能随意移动指针地址，这就完全避免了指针越界等危险操作，可以代码更安全：

```
ptr1++; // 导致编译错误

ptr2 += 2; // 导致编译错误
```

std::unique_ptr 的使用比较简单，也是用得比较多的智能指针。这里直接看例子。

1. 使用裸指针时，要记得释放内存。

```
{
    int* p = new int(100);
    // ...
    delete p;  // 要记得释放内存
}
```

2. 使用 std::unique_ptr 自动管理内存。

```
{
//第一种方式
    std::unique_ptr<int> uptr = std::make_unique<int>(200);
//第二种方式
    std::unique_ptr<int> uptr(new int(200));
//第三种方式,有风险，若ptr被delete，则unique_ptr管理的指针就无效了
    int* ptr = new int(200);
    std::unique_ptr<int> uptr(ptr);
//第四种
   std::unique_ptr<int> uptr;
   uptr.reset(new int(200));

    //...
    // 离开 uptr 的作用域的时候自动释放内存
}
```

1. std::unique_ptr 是 move-only 的。

```
{
    std::unique_ptr<int> uptr = std::make_unique<int>(200);
    std::unique_ptr<int> uptr1 = uptr;  // 编译错误，std::unique_ptr<T> 是 move-only 的

    std::unique_ptr<int> uptr2 = std::move(uptr);
    assert(uptr == nullptr);
}
```

1. std::unique_ptr 可以指向一个数组。

```
{
    std::unique_ptr<int[]> uptr = std::make_unique<int[]>(10);
    for (int i = 0; i < 10; i++) {
        uptr[i] = i * i;
    }
    for (int i = 0; i < 10; i++) {
        std::cout << uptr[i] << std::endl;
    }
}
```

1. 自定义 deleter。

```
{
    struct FileCloser {
        void operator()(FILE* fp) const {
            if (fp != nullptr) {
                fclose(fp);
            }
        }
    };
    std::unique_ptr<FILE, FileCloser> uptr(fopen("test_file.txt", "w"));
}
```

1. 使用 Lambda 的 deleter。

```
{
    std::unique_ptr<FILE, std::function<void(FILE*)>> uptr(
        fopen("test_file.txt", "w"), [](FILE* fp) {
            fclose(fp);
        });
}
```

一个完整的 smart_ptr 代码列表

shared_ptr

std::shared_ptr<T> ： 共享资源所有权的指针。

注意：

1. 不要用裸指针初始化智能指针，而要使用new或者make_shared<>()来初始化智能指针，因为裸针存在于智能指针之外，可能被别的东西释放

2. 不要返回向外直接传递this指针，因为外部可能delete这个this指针

3. **不要将this指针以shared_ptr形式返回，比如shared_ptr<gt;(this)。这回导致this被释放两次，这是由于shared_ptr不会通过判断传入的指针是否相同来判断用计数，因此即使传入相同指针，也会认为是不同对象。引用计数还是1**

```
#include <iostream>
#include <boost/shared_ptr.hpp>
class Test
{
public:
    //析构函数
    ~Test() { std::cout << "Test Destructor." << std::endl; }
    //获取指向当前对象的指针
    boost::shared_ptr<Test> GetObject()
    {
        boost::shared_ptr<Test> pTest(this);
        return pTest;
    }
};
int main(int argc, char *argv[])
{
    {
        boost::shared_ptr<Test> p( new Test( ));
        std::cout << "p.use_count(): " << p.use_count() << std::endl;
        boost::shared_ptr<Test> q = p->GetObject();
    }
    return 0;
}
```

运行后，程序输出：

p.use_count(): 1

Test Destructor.

Test Destructor.

可以看到，对象只构造了一次，但却析构了两次。并且在增加一个指向的时候，shared_ptr的计数并有增加。也就是说，这个时候，p和q都认为自己是Test指针的唯一拥有者，这两个shared_ptr在计为0的时候，都会调用一次Test对象的析构函数，所以会出问题。

**那么为什么会这样呢？给一个shared_ptr<Test
gt;传递一个this指针难道不能引起shared_ptr<
est>的计数吗?**

答案是：对的，shared_ptr <Test>根本认不得你传进来的指针变量是不是之前已经传过。

看这样的代码：

```
int main()
{
    Test* test = new Test();
    shared_ptr<Test> p(test);
    shared_ptr<Test> q(test);
    std::cout << "p.use_count(): " << p.use_count() << std::endl;
    std::cout << "q.use_count(): " << q.use_count() << std::endl;
    return 0;
}
```

运行后，程序输出：

p.use_count(): 1

q.use_count(): 1

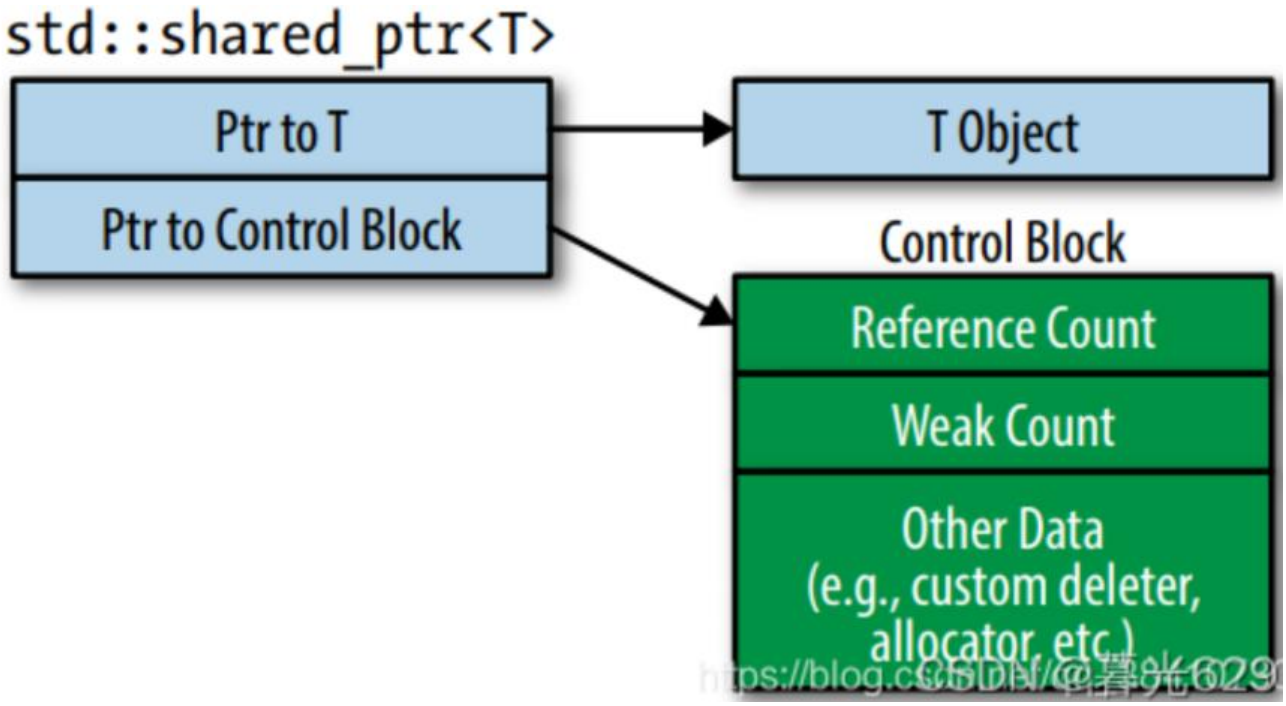
Test Destructor.

Test Destructor.

也证明了刚刚的论述：shared_ptr<Test>根本认不得你传进来的指针变量是不是之前已经传过。

事实上，类对象是由外部函数通过某种机制分配的，而且一经分配立即交给 shared_ptr管理，**而且以凡是需要共享使用类对象的地方，必须使用这个 shared_ptr当作右值来构造产生或者拷贝产生（shard_ptr类中定义了赋值运算符函数和拷贝构造函数）另一个shared_ptr ，从而达到共享使用的目的**

shared_ptr内存模型



由图可以看出，shared_ptr包含了 **一个指向对象的指针和一个指向控制块的指针** 。每一个由shared_ptr管理的对象都有一个控制块，它除了包含强引用计数、弱引用计数之外，还包含了自定义删除器的本和分配器的副本以及其他附加数据。

make_shared<>()的必要性

我们有下面的两个过程：

– new int申请内存，并把指针传给shared_ptr中的px

– 在shared_ptr中，会另外申请一块内存，初始化引用计数为1，并把指针赋值给pn

这样把创建一个智能指针需要分两步申请内存，会存在下面两个问题：

– 当 new int 申请内存成功，但引用计数内存申请失败时，很可能造成内存泄漏。

– 内存分配是一个消耗性能的过程，分两次分配内存，意味着性能会下降

为了解决直接使用shared_ptr创建智能指针带来的问题，C++11标准库引入了make_shared：

```
auto p = make_shared<int>(100);
```

make_shared只会申请一次内存，这块内存会大于int所占用的内存，多出的部分被用于智能指针引计数。这样就避免了直接使用shared_ptr带来的问题。

一种简单实现：

```
//一个完整的 smart_ptr 代码列表
#include <utility>  // std::swap

class shared_count {
public:
  shared_count() noexcept
```

```
    : count_(1) {}
  void add_count() noexcept
  {
    ++count_;
  }
  long reduce_count() noexcept
  {
    return --count_;
  }
  long get_count() const noexcept
  {
    return count_;
  }

private:
  long count_;
};

template <typename T>
class smart_ptr {
public:
  template <typename U>
  friend class smart_ptr;

  explicit smart_ptr(T* ptr = nullptr)
    : ptr_(ptr)
  {
    if (ptr) {
      shared_count_ =
        new shared_count();
    }
  }
  ~smart_ptr()
  {
    if (ptr_ &&
      !shared_count_
        ->reduce_count()) {
      delete ptr_;
      delete shared_count_;
    }
  }

  smart_ptr(const smart_ptr& other)
  {
    ptr_ = other.ptr_;
    if (ptr_) {
      other.shared_count_
        ->add_count();
      shared_count_ =
        other.shared_count_;
    }
  }
  template <typename U>
  smart_ptr(const smart_ptr<U>& other) noexcept
```

```
{
  ptr_ = other.ptr_;
  if (ptr_) {
    other.shared_count_->add_count();
    shared_count_ = other.shared_count_;
  }
}
template <typename U>
smart_ptr(smart_ptr<U>&& other) noexcept
{
  ptr_ = other.ptr_;
  if (ptr_) {
    shared_count_ =
      other.shared_count_;
    other.ptr_ = nullptr;
  }
}
template <typename U>
smart_ptr(const smart_ptr<U>& other,
          T* ptr) noexcept
{
  ptr_ = ptr;
  if (ptr_) {
    other.shared_count_
      ->add_count();
    shared_count_ =
      other.shared_count_;
  }
}
smart_ptr&
operator=(smart_ptr rhs) noexcept
{
  rhs.swap(*this);
  return *this;
}

T* get() const noexcept
{
  return ptr_;
}
long use_count() const noexcept
{
  if (ptr_) {
    return shared_count_
      ->get_count();
  } else {
    return 0;
  }
}
void swap(smart_ptr& rhs) noexcept
{
  using std::swap;
  swap(ptr_, rhs.ptr_);
  swap(shared_count_,
```

```
        rhs.shared_count_);
  }

  T& operator*() const noexcept
  {
    return *ptr_;
  }
  T* operator->() const noexcept
  {
    return ptr_;
  }
  operator bool() const noexcept
  {
    return ptr_;
  }

private:
  T* ptr_;
  shared_count* shared_count_;
};

template <typename T>
void swap(smart_ptr<T>& lhs,
          smart_ptr<T>& rhs) noexcept
{
  lhs.swap(rhs);
}

template <typename T, typename U>
smart_ptr<T> static_pointer_cast(
  const smart_ptr<U>& other) noexcept
{
  T* ptr = static_cast<T*>(other.get());
  return smart_ptr<T>(other, ptr);
}

template <typename T, typename U>
smart_ptr<T> reinterpret_pointer_cast(
  const smart_ptr<U>& other) noexcept
{
  T* ptr = reinterpret_cast<T*>(other.get());
  return smart_ptr<T>(other, ptr);
}

template <typename T, typename U>
smart_ptr<T> const_pointer_cast(
  const smart_ptr<U>& other) noexcept
{
  T* ptr = const_cast<T*>(other.get());
  return smart_ptr<T>(other, ptr);
}

template <typename T, typename U>
smart_ptr<T> dynamic_pointer_cast(
```

```
  const smart_ptr<U>& other) noexcept
{
  T* ptr = dynamic_cast<T*>(other.get());
  return smart_ptr<T>(other, ptr);
}
```

share_ptr 的循环引用

智能指针的开销即性能表现

智能指针在多线程下的安全问题

37.lambda 表达式

38.C++ 的自由存储区和堆区概念

堆，英文是 heap，在内存管理的语境下，指的是动态分配内存的区域。这个堆跟数据结构里的堆不一回事。这里的内存，被分配之后需要手工释放，否则，就会造成内存泄漏。

C++ 标准里一个相关概念是**自由存储区**，英文是 free store，特指使用 new 和 delete 来分配和释内存的区域。一般而言，这是堆的一个子集：

- new 和 delete 操作的区域是 free store
- malloc 和 free 操作的区域是 heap

但 new 和 delete 通常底层使用 malloc 和 free 来实现，所以 free store 也是 heap。鉴于对其区分实际意义并不大。

C++ 的 RAII 是什么

RAII，完整的英文是 Resource Acquisition Is Initialization，是 C++ 所特有的资源管理方式。有少其他语言，如 D、Ada 和 Rust 也采纳了 RAII，但主流的编程语言中， C++ 是唯一一个依赖 RAII 做资源管理的。

RAII 依托栈和析构/构造函数，来对所有的资源——包括堆内存在内——进行管理。对 RAII 的使

，使得 C++ 不需要类似于 Java 那样的垃圾收集方法，也能有效地对内存进行管理。RAII 的存在，是垃圾收集虽然理论上可以在 C++ 使用，但从来没有真正流行过的主要原因。

具体的管理流程：

我们可以应用代理模式，把裸指针包装起来，在构造函数里初始化，在析构函数里释放。在栈上创建象，当对应栈帧销毁时，对象的生命周期也就结束了，会自动调用析构函数（抛出异常也能完成析构，完成资源的释放管理。

设计模式

概述

一般而言，设计模式是针对面向对象思想而言的。

一个设计模式用来 描述几个模块或类对象之间关系、职责，以及他们之间如何进行分工与合作。一个者几个设计模式共同配合来解决软件设计中的问题。

大型项目中，设计模式保证锁设计的模块之间代码的灵活性和可复用性---都需要以增加代码的复杂性代价。

- 灵活性（可扩展行/低耦合性）
    - 修改现有的部分内容不会影响到其他部分的内容(影响面尽可能的窄，或者尽可能将需要修改的码集中在一起，不希望大范围修改代码)
    - 增加新内容的时候尽量少，甚至不需要改动系统现有的内容。
- 可复用性
    - 可复用：可以重复使用，可以到处用（可以被很多地方调用）
        - 面向对象三大特性：封装，继承，多态
        - 基于对象的编程模型融入继承和多态后就形成了面向对象。

设计模式中的抽象思维

- 耦合：两个模块相互依赖，修改其中一个模块，另一个也要跟着改。
- 解耦合：通过修改程序代码，切换两个模块之间的依赖关系，对任意一个模块的修改，不会影响到一个模块。

抽象思维的概念：能从事物中抽取或提炼一些本质的，共性的内容，把这些共性的内容组合到一起(封)

比如把猫和狗封装成动物类，他们都能吃，能喝，能叫。

抽象思维的目的：

- 设计原则：减少代码的重复性，提高代码的扩展性

● 做抽象的原则：把比较稳定的，不怎么变化的内容作为一个模块，单独定义出来。

抽象思维的检验，即如何检验某种抽象是否做的不错？

● 项目需求如果发生更改，是否可以不更改代码或者尽可能少更改现有代码，只通过增加新的代码应需求的变更。新需求就相当于一个新的模块，吧这个新模块加到原来的代码中，而不需要更改原来的西，这就是可扩展性。

● 类中的内容太多时，就要对类进行拆分，不要把相关度不高的内容写到一个类中。提高抽象度，降耦合度。

● 面向对象程序设计原则之一：单一职责原则---一个类只干好一件事，承担好一种责任，不然就会牵太多。

分类

常见设计模式有 23 种，通常分为 3 大类：

1. 创建型模式：关注如何创建对象。把对象的创建和使用分离（解耦）取代传统对象创建方式可能导的代码修改和维护上的问题。

2. 结构型模式：关注对象之间的关系。设计如何组合各种对象以便获得更佳灵活的结构，通过继承以更多的关系组合以获得更佳灵活的程序结构，达到简化设计的目的。

3. 行为型模式：关注对象行为或者交互方面的内容，主要设计算法和愁之间的职责分配。通过使用对组合，行为型模式可以描述一组对象如何协作来完成一个整体任务。

表 1.1　常用设计模式及分类一览表

设计模式名称	设计模式分类
模板方法（Template Method）模式	行为型模式
策略（Strategy）模式	
观察者（Observer）模式	
命令（Command）模式	
迭代器（Iterator）模式	
状态（State）模式	
中介者（Mediator）模式	
备忘录（Memento）模式	
职责链（Chain Of Responsibility）模式	
解释器（Interpreter）模式	
访问者（Visitor）模式	
工厂模式： ①简单工厂（Simple Factory）模式 ②工厂方法（Factory Method）模式 ③抽象工厂（Abstract Factory）模式	创建型模式
单件（Singleton）模式	
原型（Prototype）模式	
建造者（Builder）模式	
装饰（Decorator）模式	结构型模式
外观（facade）模式	
组合（Composite）模式	
享元（Flyweight）模式	
代理（Proxy）模式	
适配器（Adapter）模式	
桥接模式（Bridge）模式	

工厂类

简单工厂类

实现思路：使用工厂类，根据传入不同参数可以创建不同的子类对象，并通过返回一个父类指针的方返回创建的对象。

封装变化：把依赖范围尽可能缩小，把容易变化的代码段限制在一个小范围，就可以很大程度上提高码的可维护性和可扩展性。

通过增加新的 if else 分支来达到支持新子类对象增加的目的---违背了面向对象程序设计的原则：开

原则。

开闭原则：说的是代码扩展性问题---对扩展开放，对修改关闭（封闭）

详细解释：当增加新功能，不应该通过修改已经存在的代码来进行，而是应该通过扩展代码（比如增新类，增加新成员函数）来扩展功能。但是如果 if else 分支比较少可以不遵循开闭原则，

简单工厂类的缺点：

● 系统扩展难，一旦增加新产品，就需要修改工厂逻辑，不利于系统的扩展与维护， **违背了“开闭则”**；简单工厂模式中所有 **产品的创建都是由同一个工厂创建，工厂类职责较重，业务逻辑较为复杂具体产品与工厂类之间耦合度高，严重影响了系统的灵活性和扩展性** 。

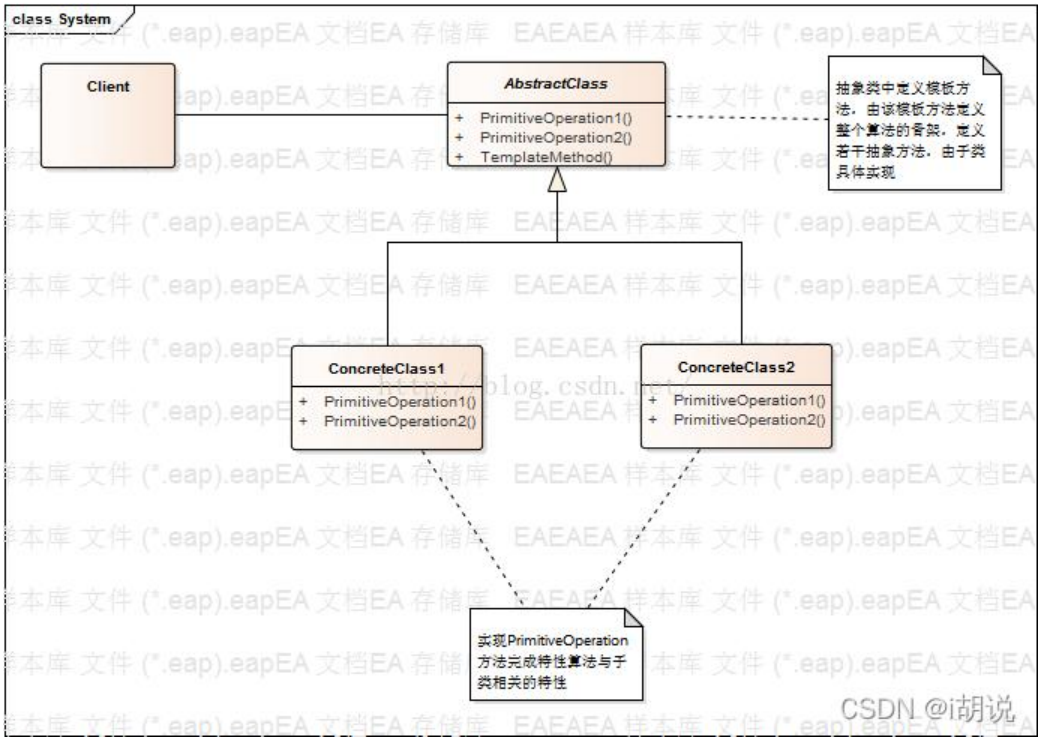
普通工厂模式

模板方法

该方法的作用在于最大化复用代码。它直接利用了继承机制，对于可以被子类们复用的代码直接放在类中，需要根据实际情况做相应特化的算法，则定义成虚函数，由子类去实现。

这里的关键在于将一个算法实现拆分成多个，这样细分一个算法的直接好处就是，可以挑出部分相同实现，把这部分放到父类中，由所有子类共享，充分利用继承机制，最大化复用代码。

模板方法定义一个操作中的算法骨架，而将一些步骤延迟到子类中实现 。模板方法使得子类可以再不变一个算法的结构即可重新定义该算法的某些特定步骤。通过把不变的行为搬移到基类中，去除了子的重复代码，提供了一个很好的代码复用解决方案。




```
// 面向对象
class Library {
public:
    // 稳定中包含变化
    void Run() {
        Step1();

        if (Step2()) {  // 支持变化 虚函数多态调用，这种判断也称为钩子函数，即在子类中定义的这个数能在这里决定模板的执行流程
            Step3();
        }

        for (...) {
            Step4();// 支持变化 虚函数多态调用
        }

        Step5();
    }

    virtual ~Library();
private:
    Step1();
    Step3();
    Step5();

    virtual Step2();
    virtual Step4();
};

class Application : public Library {
public:
```

```
    Step4();
    Step5();
};

int main() {
    Library* pLib = new Application;
    pLib->run();// run()并不是虚函数，此处调用基类的run()，但是在run()内部的Step2()、Step4()是虚函数，调用的时Application的Step2()、Step4()
    delete pLib;
    ...
}
```


策略模式

1. 介绍

目的: 定义一系列的算法(行为),把它们一个个封装成类，它们继承同一个基类，实现其中的纯虚函数。**用多态机制**，以父类对象接收某策略对象，实现动态绑定。

关键: 实现同一个接口.

优点:

- 算法之间可以互相替换;
- 可以避免多重条件判断;
- 扩展性良好.

缺点:

- 策略类会比较多;
- 所有策略都需要对外暴露.

使用场景:

1. 一个对象可能存在多种行为,需要使用多重条件判断时;
2. 系统需要在几种算法中选择一种.

```
#include <iostream>
#include <memory>

//接口类
```

```
class Strategy{
public:
//定义成纯虚函数
    virtual int doOperate(const int a, const int b) = 0;
};

//算法实现1
class StrategyAdd : public Strategy{
public:
//具体的策略要实现对应的算法
    int doOperate(const int a, const int b) override {
        return a+b;
    }
};

//算法实2
class StrategySub : public Strategy{
public:
//具体的策略要实现对应的算法
    int doOperate(const int a, const int b) override {
        return a-b;
    }
};

//条件选择
class Context {
 public:
    Context(std::shared_ptr<Strategy> Strategy) : Strategy_(Strategy){}

    std::shared_ptr<Strategy> GetStrategy() {
        return Strategy_;
    }

    void SetStrategy(std::shared_ptr<Strategy> Strategy) {
        Strategy_ = Strategy;
    }
    //根据传入的不同策略对象，运用多态机制，会最终调用不同的策略算法
    int doOperate(const int a, const int b) {
        return Strategy_->doOperate(a, b);
    }

 private:
    std::shared_ptr<Strategy> Strategy_;
};

int main() {
    //向对象传入不同的策略实例，运用多态机制，最终调用的就是对应策略的算法
    Context context(std::make_shared<StrategyAdd>());
    std::cout << context.doOperate(10, 100) << std::endl;

    context.SetStrategy(std::make_shared<StrategySub>());
    std::cout << context.doOperate(10, 100) << std::endl;
    return 0;
}
```

输出结果:

```
110
-90
```

其他

策略模式初看和工厂模式其实是有点像的,但是细看其实区别很大:

- 工厂模式是创建型模式,它的作用就是创建对象；策略模式是行为型模式,它的作用是让一个对象在多行为中选择一种行为;
- 工厂模式关注对象创建,实际创建对象是被封装的,用户不可见;而策略模式更多是关注行为的替换,创对象其实是对用户可见的.

当然也不要盲目追求设计模式,在行为类型并不多时,直接使用条件判断更加合理.

单例模式

懒汉模式

在懒汉式的单例类中，其实有两个状态，单例未初始化和单例已经初始化。假设单例还未初始化，有个线程同时调用GetInstance方法，这时执行 m_pInstance == NULL 肯定为真，然后两个线程都初化一个单例，最后得到的指针并不是指向同一个地方，不满足单例类的定义了，所以懒汉式的写法会现线程安全的问题！在多线程环境下，要对其进行修改。

第一个检查是防止创建后，每次在获取单例时重复加锁，降低了效率

```
class Singleton
{
private:
    static Singleton* m_instance;
    Singleton(){}
public:
    static Singleton* getInstance();
};

Singleton* Singleton::getInstance()
{
    if(NULL == m_instance)
    {
        Lock();//借用其它类来实现，如boost
        if(NULL == m_instance)
        {
            m_instance = new Singleton;
        }
        UnLock();
    }
    return m_instance;
}
```

文出处链接及本声明。
原文链接：https://blog.csdn.net/qq_43491149/article/details/121180390

静态对象版本

```
template<class T>
class Singleton {
public:
    static T* instance() {
        static T value;
        return &value;
    }
private:
Singleton() = delete;
~Singleton();
};
```

□

□

饿汉模式

```
class CSingleton
{
private:
    CSingleton() {}
public:
    static CSingleton * GetInstance()
    {
        static CSingleton instance;
        return &instance;
    }
};
```

————————————————

进程和线程

□

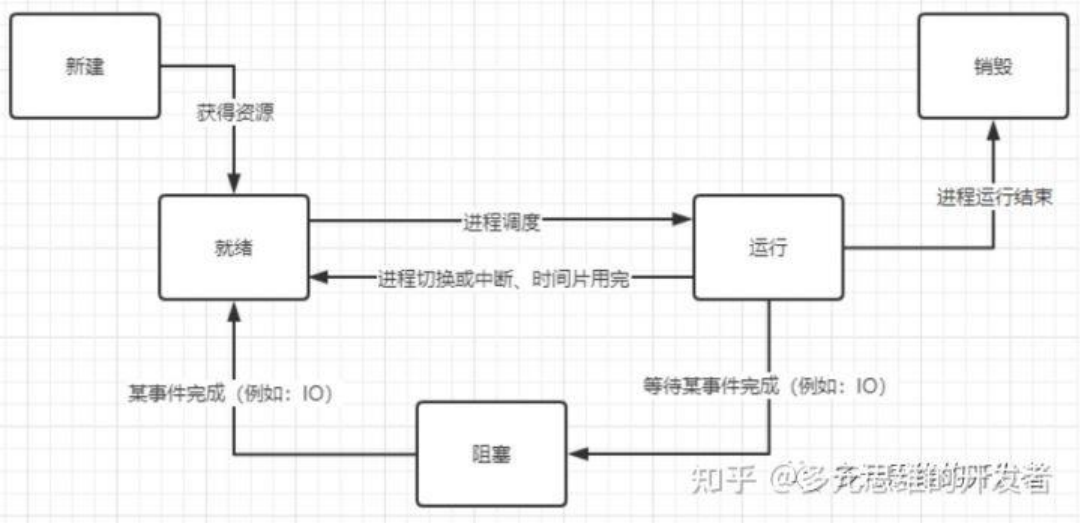
进程

从内核观点看，进程的目的就是担当分配系统资源（CPU 时间、内存等）的实体。

□

□

进程主要有五种状态，分别是新建、就绪、运行、阻塞、销毁。如下图



进程描述符 task_struct

线程

实现多线程应用程序的一个简单方式就是把轻量级进程与每个线程关联起来。这样，线程之间就可以过简单地共享同一内存地址空间、同一打开文件集等来访问相同的应用程序数据结构集;同时，每个线都可以由内核独立调度，以便一个睡眠的同时另一个仍然是可运行的。

POSIX 兼容的多线程应用程序由支持"线程组"的内核来处理最好不过。在 Linux 中，一个线程组基本就是实现了多线程应用的一组轻量级进程，对于像 getpid（），kill（），和 exit（）这样的一些系调用，它像一个组织，起整体的作用

因此在 Linux 下其实本并没有线程，只是搞了个轻量级进程出来就叫做了线程。**轻量级进程和进程一，都有自己独立的 task_struct 进程描述符，也都有自己独立的 pid。**从操作系统视角看，调度上和程没有什么区别，**都是在等待队列的双向链表里选择一个 task_struct 切到运行态而已。**只不过轻量进程和普通进程的区别是可以共享同一内存地址空间、代码段、全局变量、同一打开文件集合而已。

Linux 把不同的 PID 与系统中每个进程或轻量级进程相关联。这种方式能提供最大的灵活性，因为系中每个执行上下文都可以被唯一地识别。

另一方面，Unix 程序员希望同一组中的线程有共同的 PID。例如，把指定 PID 的信号发送给组中的有线程。**事实上，POSIX 1003.1c 标准规定一个多线程应用程序中的所有线程都必须有相同的 PID。**

遵照这个标准，Linux 引入线程组的表示(典型的先有实事，后有标准)。一个线程组中的所有线程使和该线程组的领头线程（thread group leader）相同的 PID，也就是该组中第一个轻量级进程的 PI，它被存入进程描述符的 tgid 字段中。getpid（）系统调用返回当前进程的 tgid 值而不是 pid 的值因此，一个多线程应用的所有线程共享相同的 PID。

简单概括：

- linux 开始只有进程，每个进程有唯一标识 pid，和描述其全部信息的 task_struct
- 后来有了线程，linux 为了在内核级实现线程这一概念，提出了轻量级进程（LWP）。每个 LWP 表一个线程，他有唯一的标识 PID，记录其全部信息并且方便调度的 taskt_struct。
- 在后来 posix 标准对线程做出了规定，Linux 为了遵循规定，比如同一进程的线程要有同一的 pid 所以新设了 tgid 标识第一个线程的内部 pid，对外的 pid 实际就是 tgid。而对外 LWP id 就是原来的 pid。

总的来说以 task_struct 中的 LWP 区分线程，tgid = pid 区分进程，pgid 区分进程组

pid: 进程 ID。getpid()获取的结果。

tgid：线程组 ID，线程组 leader 的 ID，也就是 pid，因为第一个线程的 pid = LWP

lwp: 线程 ID。每个线程都有自己的唯一 ID. #define gettid() syscall(__NR_gettid)获取的结果。

tid: 线程 ID，等于 lwp。tid 在系统提供的接口函数中更常用，比如 syscall(SYS_gettid)和 syscall(__R_gettid)。

pgid: 进程组 ID，也就是进程组 leader 的进程 ID。

pthread id: pthread 库提供的 ID，生效范围不在系统级别，可以忽略。pthread_self()

./pthread 进程内额外创建了十个线程，加上主线程总共 11 个线程，正如 NLWP 所指。

```
# GNX @ VM-24-14-centos in ~/workspace on git:master x [19:30:20]
$ ps -eLf
UID        PID  PPID   LWP  C NLWP STIME TTY          TIME CMD
GNX       6653 32002  6653  0   11 19:29 pts/6    00:00:00 ./pthread
GNX       6653 32002  6654  2   11 19:29 pts/6    00:00:00 ./pthread
GNX       6653 32002  6655  2   11 19:29 pts/6    00:00:01 ./pthread
GNX       6653 32002  6656  2   11 19:29 pts/6    00:00:00 ./pthread
GNX       6653 32002  6657  2   11 19:29 pts/6    00:00:01 ./pthread
GNX       6653 32002  6658  2   11 19:29 pts/6    00:00:01 ./pthread
GNX       6653 32002  6659  2   11 19:29 pts/6    00:00:00 ./pthread
GNX       6653 32002  6660  2   11 19:29 pts/6    00:00:00 ./pthread
GNX       6653 32002  6661  2   11 19:29 pts/6    00:00:00 ./pthread
GNX       6653 32002  6662  2   11 19:29 pts/6    00:00:00 ./pthread
GNX       6653 32002  6663  2   11 19:29 pts/6    00:00:00 ./pthread
```

ps -eLf 各字段含义

UID：用户 ID

PID：process id 进程 id

PPID: parent process id 父进程 id

LWP：表示这是个线程；要么是主线程(进程)，要么是线程

NLWP: num of light weight process 轻量级进程数量，即线程数量

STIME: start time 启动时间

TIME: 占用的 CPU 总时间

TTY：该进程是在哪个终端运行的;pts/0255 代表虚拟终端，一般是远程连接的终端;tty1tty7 代表本控制台终端

CMD：进程的启动命令

```
#include <unistd.h>
#include <sys/syscall.h>
#include <stdio.h>
#include <pthread.h>

#define gettidv1() syscall(__NR_gettid) // new form
#define gettidv2() syscall(SYS_gettid)  // traditional form

void *ThreadFunc1()
{
    printf("the pthread_1 id is %ld\n", pthread_self());
    printf("the thread_1's Pid is %d\n", getpid());
    printf("The LWPID/tid of thread_1 is: %ld\n", (long int)gettidv1());
    pause();
    return 0;
}

void *ThreadFunc2()
{
    printf("the pthread_2 id is %ld\n", pthread_self());
    printf("the thread_2's Pid is %d\n", getpid());
    printf("The LWPID/tid of thread_2 is: %ld\n", (long int)gettidv1());
    pause();

    return 0;
}

int main(int argc, char *argv[])
{
    pid_t tid;
    pthread_t pthread_id;

    printf("the master thread's pthread id is %ld\n", pthread_self());
    printf("the master thread's Pid is %d\n", getpid());
    printf("The LWPID of master thread is: %ld\n", (long int)gettidv1());

    // 创建2个线程
    pthread_create(&pthread_id, NULL, ThreadFunc2, NULL);
    pthread_create(&pthread_id, NULL, ThreadFunc1, NULL);
    pause();

    return 0;
}
```

进程和线程的切换

无论是在多核还是单核系统中，一个 CPU 看上去都像是在并发的执行多个进程，这是通过处理器在程间切换来实现的。

- 操作系统实现这种交错执行的机制称为上下文切换。
- 操作系统保持跟踪 **进程运行所需的所有状态信息，这种状态，也就是上下文**，它包括许多信息例如 PC 和寄存器文件的当前值，以及主存的内容。

上下文切换

内核为每一个进程维持一个上下文。 **上下文就是内核重新启动一个被抢占的进程所需的状态。** 包括下内容：

- 通用目的寄存器
- 浮点寄存器
- 程序计数器
- 用户栈
- 状态寄存器
- 内核栈
- 各种内核数据结构：比如描绘地址空间的页表，包含有关当前进程信息的进程表，以及包含进程已开文件的信息的文件表。

进程上下文切换开销都有哪些

那么上下文切换的时候，CPU 的开销都具体有哪些呢？开销分成两种，**一种是直接开销、一种是间接销。**

直接开销就是在切换 task_struct 时，cpu 必须做的事情，包括：

- 1、切换页表全局目录，以安装一个新的地址空间。
- 2、切换内核态堆栈
- 3、切换硬件上下文，硬件上下文提供了内核执行新进程所需要的所有信息（进程恢复前，必须装寄存器的数据统称为硬件上下文）
    - ip(instruction pointer)：指向当前执行指令的下一条指令
    - bp(base pointer): 用于存放执行中的函数对应的栈帧的栈底地址
    - sp(stack poinger): 用于存放执行中的函数对应的栈帧的栈顶地址
    - cr3:页目录基址寄存器，保存页目录表的物理地址
    - ……
- 4、刷新 TLB
- 5、系统调度器的代码执行

间接开销主要指的是**缓存缺失**：

虽然切换到一个新进程后，由于各种缓存并不热，速度运行会慢一些。如果进程始终都在一个 CPU 调度还好一些，如果跨 CPU 的话，之前热起来的 TLB、L1、L2、L3 因为运行的进程已经变了，所以局部性原理 cache 起来的代码、数据也都没有用了，导致新进程穿透到内存的 IO 会变多。所以实际上下文切换开销可能比 3.5us 要大。

想了解更详细操作过程的同学请参考《深入理解 Linux 内核》中的第三章和第九章。

进程切换需要切换页表，导致刷新 TLB，来更换虚拟内存空间：

地址转换需要两个东西，一个是 CPU 上的内存管理单元 MMU，另一个是内存中的页表，页表中存虚拟地址到物理地址的映射

但是呢，每次访问内存，都需要进行虚拟地址到物理地址的转换，对吧，这样的话，页表就会被频繁访问，而页表又是存在于内存中的。所以说，访问页表（内存）次数太多导致其成为了操作系统地一性能瓶颈。

于是，引入了转换检测缓冲区 TLB，也就是快表，其实就是一个缓存，把经常访问到的内存地址映射在 TLB 中，因为 TLB 是在 CPU 的 MMU 中的嘛，所以访问起来非常快。

由于进程切换会涉及到虚拟地址空间的切换，这就导致内存中的页表也需要进行切换，一个进程对应个页表是不假，但是 CPU 中的 TLB 只有一个啊，这就尴尬了，页表切换后这个 TLB 就失效了。这样 TLB 在一段时间内肯定是无法被命中的，操作系统就必须去访问内存，那么虚拟地址转换为物理地址会变慢，表现出来的就是程序运行会变慢。

而线程切换呢，由于不涉及虚拟地址空间的切换，也就不存在这个问题了。

线程切换

线程是调度的基本单位，因为在 linux 中，进程被细化为轻量级进程（LWP）,每个 LPW 都有一个 struct_task 用于存储线程全部信息，方便调度切换不同线程。

从操作系统视角看，调度上和进程没有什么区别，**都是在等待队列的双向链表里选择一个 task_struct 切到运行态而已。**只不过轻量级进程和普通进程的区别是可以共享同一内存地址空间、代码段、全局量、同一打开文件集合而已。

当调度的两个线程属于不同进程，那就是进程切换，否则就是线程切换。

其实线程和进程在时间上的开销是差不多的，线程最主要的优势在于缓存仍然可用，仍然可以利用局性原理提高访问数据的速度。

切换的时间开销

进程上下文切换：数百纳秒到数微妙，一般 3-5us。

系统调用：系统调用的时候，最低值是 200ns。可见，上下文切换开销要比系统调用的开销要大。系调用只是在进程内将用户态切换到内核态，然后再切回来，而上下文切换可是直接从进程 A 切换到了程 B。显然这个上下文切换需要完成的工作量更大。

线程上下文切换：大约每次线程切换开销大约是 3.8us 左右。 **从上下文切换的耗时上来看，Linux 程（轻量级进程）其实和进程差别不太大** 。

那为什么还说切换线程比进程效率高呢?

答：首先要明确一点，我们说的线程切换效率高，是指同一进程中的线程切换。不同进程中的线程切，那就是进程的切换。

同一进程线程切换的间接开销要更小一些，也就是缓存中的内容大概率仍然可用，可以充分利用局部原来。包括 cpu cache 中的缓存，TLB 中的缓存。

协程

它主要有三个特点:

- 占用的资源更少；
- 所有的切换和调度都发生在用户态。
- 它的调度是协商式的，而不是抢占式的。

前两个特点容易理解，我来给你重点解释一下第三个特点。

目前主流语言基本上都选择了多线程作为并发设施，与线程相关的概念是抢占式多任务（Preemptive multitasking），而与协程相关的是协作式多任务。不管是进程还是线程，每次阻塞、切换都需要陷系统调用 (system call)，先让 CPU 执行操作系统的调度程序，然后再由调度程序决定该哪一个进程线程) 继续执行。由于抢占式调度执行顺序无法确定，我们使用线程时需要非常小心地处理同步问题而协程完全不存在这个问题。因为协作式的任务调度，是要用户自己来负责任务的让出的。如果一个务不主动让出，其他任务就不会得到调度。这是协程的一个弱点，但是如果使用得当，这其实是一个以变得很强大的优点。

平均每次协程切换的开销是（655035993-415197171)/2000000=120ns。相对于前面文章测得的程切换开销大约 3.5us，大约是其的三十分之一。比系统调用的造成的开销还要低。

协程由于是在用户态来完成上下文切换的，所以切换耗时只有区区 100ns 多一些，比进程切换要高 30 倍。单个协程需要的栈内存也足够小，只需要 2KB。所以，近几年来协程大火，在互联网后端的高并场景里大放光彩。

无论是空间还是时间性能都比进程（线程）好这么多，那么 Linus 为啥不把它在操作系统里实现了多?

实际上协程并不是一个新玩意，在上个世纪 60 年代的时候就已经有人提出了。操作系统的一个主要计目标是实时性，对优先级比较高的进程是会抢占当前占用 CPU 的进程。但是协程无法实现这一点还得依赖于使用 CPU 的一方主动释放，与操作系统的实现目的不相吻合。协程的高效是以牺牲了可占性为代价的。

协程代码实例：

```
#include <stdio.h>
```

```
#include <stdlib.h>

#define STACK_SIZE 1024

typedef void(*coro_start)();

class coroutine {
public:
    long* stack_pointer;
    char* stack;

    coroutine(coro_start entry) {
        if (entry == NULL) {
            stack = NULL;
            stack_pointer = NULL;
            return;
        }

        stack = (char*)malloc(STACK_SIZE);
        char* base = stack + STACK_SIZE;
        stack_pointer = (long*) base;
        stack_pointer -= 1;
        *stack_pointer = (long) entry;
        stack_pointer -= 1;
        *stack_pointer = (long) base;
    }

    ~coroutine() {
        if (!stack)
            return;
        free(stack);
        stack = NULL;
    }
};

coroutine* co_a, * co_b;

void yield_to(coroutine* old_co, coroutine* co) {
    __asm__ (
        "movq %%rsp, %0\n\t"
        "movq %%rax, %%rsp\n\t"
        :"=m"(old_co->stack_pointer):"a"(co->stack_pointer):);
}

void start_b() {
    printf("B");
    yield_to(co_b, co_a);
    printf("D");
    yield_to(co_b, co_a);
}

int main() {
    printf("A");
    co_b = new coroutine(start_b);
```

```
    co_a = new coroutine(NULL);
    yield_to(co_a, co_b);
    printf("C");
    yield_to(co_a, co_b);
    printf("E\n");
    delete co_a;
    delete co_b;
    return 0;
}
```

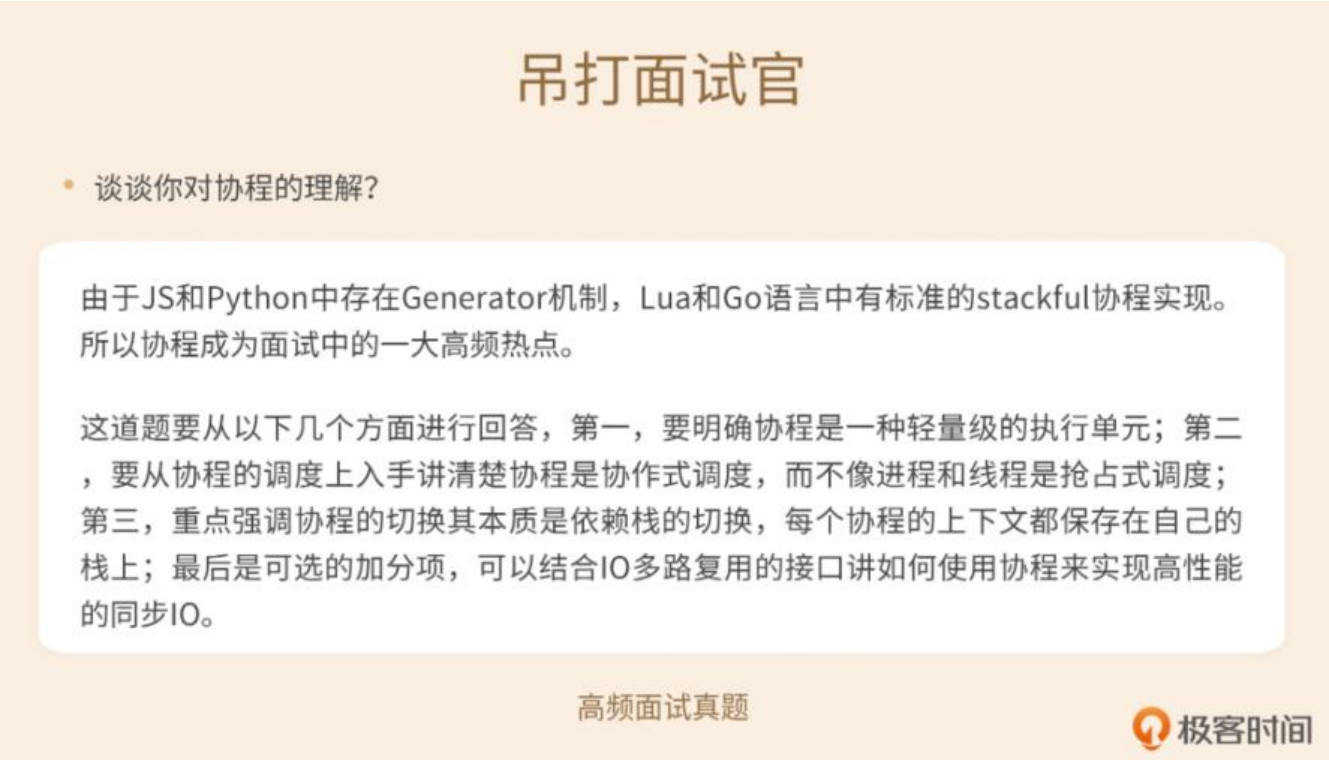
输出：

```
# g++ -g -o co -O0 coroutine.cpp
# ./co
ABCDE
```

协程能正常切换的关键在于这里的yield_to()函数。

每个协程都有一个入口函数，携程内在堆区创建的栈中会保存这个函数的地址。

调度某个协程时，实际就是将当前线程的栈sp指针设定为新的函数地址，然后将原来的sp值保存到对协程的栈中。这个过程都是在用户态完成的，不涉及系统调用



虚函数和多态

多态一般指动态多态，指类中通过虚函数和虚函数指针指向的虚函数表实现的动态多态。

具体表现形式为：**父类指针绑定不动的子类对象，从而可以调用同一个函数在不同子类中的具体实现。**

每个具有虚函数的类中，都会在类的内存空间最开头部分产生一个虚函数指针，指向一个虚函数表。类和父类中虚函数的函数指针都会存放在这个表中，**但是若子类中有和父类同名同参数的虚函数，则类的虚函数会在子类的虚函数表中覆盖父类的虚函数**，这是能调用到不同函数的关键。

那为什么一定是父类指针绑定不同的子类对象呢？

这是由类的继承模型决定的，继承的类要放在子类的内存最开始部分，因此父类存在于所有子类的开部分，而且虚函数指针就放在内存空间的最开头部分，**因此父类类型的指针有对这块内存空间的访问限。**

也因此父类指针绑定的是哪个子类对象，就能找到哪个子类的对应虚函数表，从而调用对应子类的虚数，由于虚函数覆盖的存在，所以可以调用子类的具体实现。

构造函数的执行顺序？析构函数的执行顺序？构造函数内部了啥？拷贝构造干了啥？

构造函数顺序：

1. 基类构造函数。如果有多个基类，则构造函数的调用顺序是某类在类派生表中出现的顺序，而不是们在成员初始化表中的顺序。
2. 成员类对象构造函数。如果有多个成员类对象则构造函数的调用顺序是对象在类中被声明的顺序，不是它们出现在成员初始化表中的顺序。
3. 派生类构造函数。

其实这就是有类的继承模型决定的，

析构函数顺序

1. 调用派生类的析构函数；
2. 调用成员类对象的析构函数；
3. 调用基类的析构函数。

构造函数做的工作：

构造函数默认的操作主要有

1. 调用父类构造函数，
2. 调用类类型的成员变量的构造函数
3. 以及将虚函数表的地址赋值给对象的虚函数指针等必要操作。
4. 有虚继承时会为虚继承表指针赋值
5. 然后给每个成员变量赋初值

因此

编译器生成默认构造函数的情况

在类没有显示声明构造函数的情况下，编译器并不总是为我们自动生成默认构造函数，以下4种情况编译器才会为我们自动生成默认构造函数：

1.类中有一个类成员含有默认构造函数的，编译器会为该类自动生成默认构造函数，自动插入代，调用该成员的构造函数；

2.类成员中含有默认构造函数，编译器会为该类自动生成默认构造函数，自动插入代码，调用基的构造函数；

3.类中含有类成员时，由于编译器要为该类生成虚函数表vtable，并为该类的对象生成指向该vtabe的vptr，所以需要为该类合成默认构造函数；

4.虚继承时；

除了这四种情况，编译器不会为我们自动生成默认构造函数，例如类中的整数、指针、数组等，为这些成员的初始化对编译器来说都不是必要的，所有都不会自动被初始化，这些成员的初始化需要序员自己显示编写代码实现；

C++中构造函数和析构函数可以抛出异常吗?

● 不建议在构造函数中抛出异常。当构造函数中抛出异常时，析构函数将不会被执行，需要手动释放存。

● 析构函数不应该抛出异常。当析构函数中有一些可能发生的异常时，这时候要把可能发生的异常完封装在析构函数内部，决不能让它抛出到函数之外

锁

死锁

死锁就是永远不会被解开的锁，死锁有四个条件：

1. 互斥访问
2. 持有并等待
3. 资源非抢占
4. 循环等待

当多线程中交互地使用多个锁时就很容易产生死锁。

另外，单线程，单锁情况下也可能产生死锁，比如在中断处理中加锁。一个中断在没释放锁前由进入断，这就产生了死锁，另外递归也容易产生死锁。但是他们都可以通过可重入锁来解决。

可重入锁

加锁时判断锁的持有者是否是线程自己，如果是自己则不会阻塞和等待，而是通过计数器来记录甲所次数。这个次数会在放锁时减少，当为0时才释放锁。

物理内存分配算法

内存分配最大的问题是内存碎片的问题，而内存碎片又分为内部碎片和外部碎片。

外部碎片：内存有很多小的内存块，导致想要申请一个大块，虽然总的小块合起来空间狗够，但是哪小块都不够，所以分配不了。

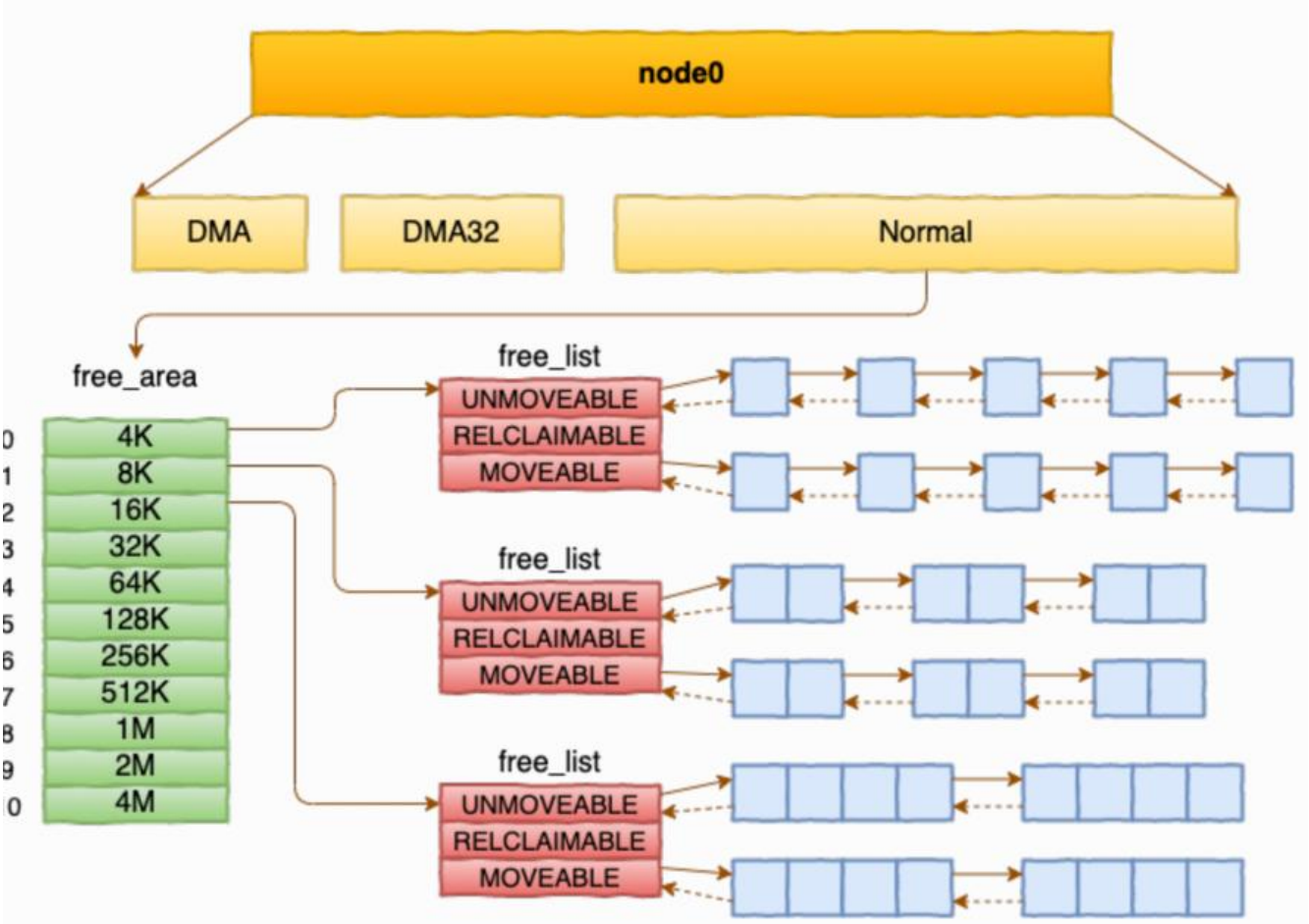
内部碎片：分配了比实际需要的更大的内存，导致分配过去的内存有一部分用不上导致浪费。

伙伴系统

每个 zone 下面都有如此之多的页面，Linux使用伙伴系统对这些页面进行高效的管理。 在内核中，示 zone 的数据结构是 struct zone 。 其下面的一个数组 free_area 管理了绝大部分可用的空闲页面这个数组就是伙伴系统实现的重要数据结构。

```
//file: include/linux/mmzone.h
#define MAX_ORDER 11
struct zone {
 free_area free_area[MAX_ORDER];
 ......
}
```

free_area是一个11个元素的数组，在每一个数组分别代表的是空闲可分配连续4K、8K、16K、.....、M内存链表。



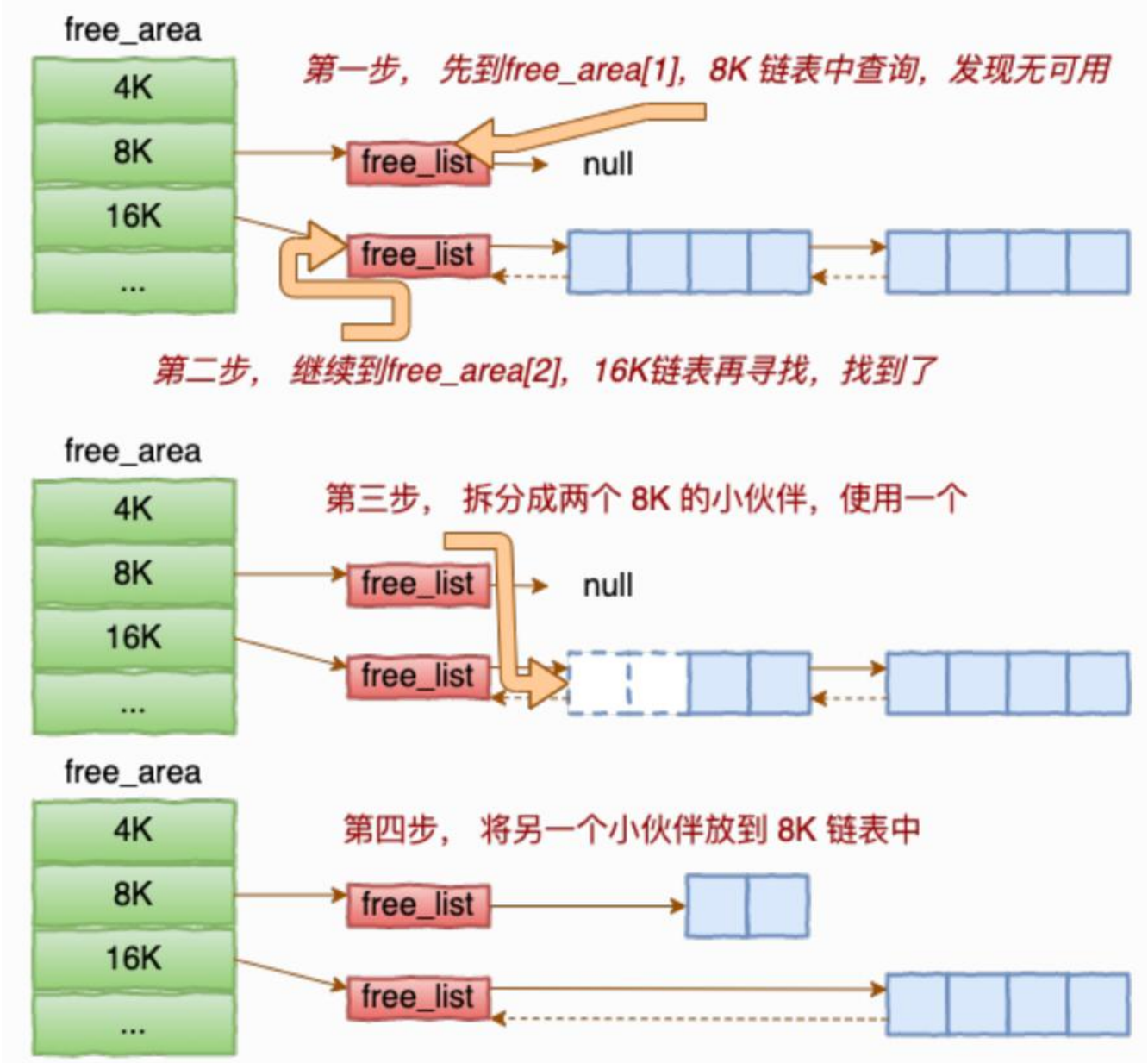
Free pages count per migrate type at order				0	1	2	3	4	5	6	7	8	9	10
Node	0, zone	DMA, type	Unmovable	1	0	1	0	2	1	1	0	1	0	0
Node	0, zone	DMA, type	Reclaimable	0	0	0	0	0	0	0	0	0	0	0
Node	0, zone	DMA, type	Movable	0	0	0	0	0	0	0	0	0	0	3
Node	0, zone	DMA, type	Reserve	0	0	0	0	0	0	0	0	0	1	0
Node	0, zone	DMA, type	CMA	0	0	0	0	0	0	0	0	0	0	0
Node	0, zone	DMA, type	Isolate	0	0	0	0	0	0	0	0	0	0	0
Node	0, zone	DMA32, type	Unmovable	153	126	44	11	15	18	15	18	13	10	64
Node	0, zone	DMA32, type	Reclaimable	1	0	48	61	67	43	30	10	8	4	1
Node	0, zone	DMA32, type	Movable	250	1478	617	174	100	48	14	3	1	1	280
Node	0, zone	DMA32, type	Reserve	0	0	0	0	0	0	0	0	0	0	1
Node	0, zone	DMA32, type	CMA	0	0	0	0	0	0	0	0	0	0	0
Node	0, zone	DMA32, type	Isolate	0	0	0	0	0	0	0	0	0	0	0
Node	0, zone	Normal, type	Unmovable	688	193	563	772	673	549	474	434	377	305	1459
Node	0, zone	Normal, type	Reclaimable	592	869	1499	926	981	674	455	281	165	80	20
Node	0, zone	Normal, type	Movable	0	24481	12880	3555	1378	950	385	131	76	51	12352
Node	0, zone	Normal, type	Reserve	0	0	0	0	0	0	0	0	0	0	1
Node	0, zone	Normal, type	CMA	0	0	0	0	0	0	0	0	0	0	0
Node	0, zone	Normal, type	Isolate	0	0	0	0	0	0	0	0	0	0	0

通过 cat /proc/pagetypeinfo , 你可以看到当前系统里伙伴系统里各个尺寸的可用连续内存块数量。

内核提供分配器函数 alloc_pages 到上面的多个链表中寻找可用连续页面。

struct page * alloc_pages(gfp_t gfp_mask, unsigned int order)

alloc_pages 是怎么工作的呢？我们举个简单的小例子。假如要申请8K-连续两个页框的内存。为了述方便，我们先暂时忽略UNMOVEABLE、RELCLAIMABLE等不同类型



伙伴系统中的伙伴指的是两个内存块，大小相同，地址连续，同属于一个大块区域。

基于伙伴系统的内存分配中，有可能需要将大块内存拆分成两个小伙伴。在释放中，可能会将两个小伴合并再次组成更大块的连续内存。

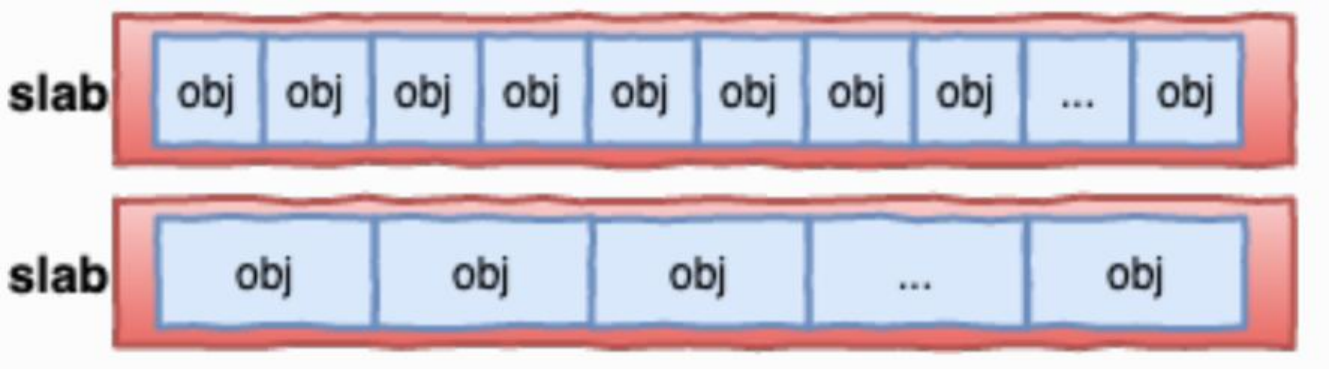
伙伴系统的一个优点是：通过称为合并的技术，可以将相邻伙伴快速组合以形成更大分段。例如，在图 1 中，当内核释放已被分配的 CL———————————— 时，系统可以将 CL———————————— 和 CR———— 合并成 64KB 的段。段 BL———————————— 继而可以与伙伴 BR———————————— 合并，以形成 128KB 段。最终，可以得到原来的 256KB 段。

伙伴系统的明显缺点是：由于圆整到下一个 2 的幂，很可能造成分配段内的碎片。例如，33KB 的内请求只能使用 64KB 段来满足。事实上，我们不能保证因内部碎片而浪费的单元一定少于 50%。

slab分配

说到现在，不知道你注意到没有。目前我们介绍的内存分配都是以页面（4KB）为单位的。对于各个核运行中实际使用的对象来说，多大的对象都有。有的对象有1K多，但有的对象只有几百、甚至几十字节。如果都直接分配一个 4K的页面 来存储的话也太败家了，所以伙伴系统并不能直接使用。

在伙伴系统之上，内核又给自己搞了一个专用的内存分配器， 叫slab或slub。 这两个词老混用，为省事，接下来我们就统一叫 slab 吧。这个分配器最大的特点就是，一个slab内只分配特定大小、甚是特定的对象。这样当一个对象释放内存后，另一个同类对象可以直接使用这块内存。通过这种办法大地降低了碎片发生的几率。



但是slab分配器依然会存在内存碎片，只不过影响很小了，比如以内核TCP对象为例子：

```
# cat /proc/slabinfo | grep TCP
TCP 288 384 1984 16 8
```

可以看到 TCP cache下每个 slab 占用 **8 个 Page**，也就是 8 4096 = 32768KB *。

该对象的单个大小是 **1984 字节** ，每个slab内放了 **16 个对象。1984*16=31744**

这个时候再多放一个 TCP 对象又放不下，剩下的 1k内存就只好“浪费”掉了。 但是鉴于 slab 机制体提供的高性能、以及低碎片的效果，这一点点的额外开销还是很值得的。

总结：

进程间通信

管道

管道是单向的，基于字节流的，默认是阻塞的，也可以将其设置为非阻塞。对管道的操作其实就是对件描述符的操作，只不过这个文件描述符是内核提供的，读写只能单向。**所谓的管道，就是内核里面一串缓存**，最小为4K页大小 。从管道的一段写入的数据，实际上是缓存在内核中的，另一端读取，就是从内核中读取这段数据，由于是基于字节流的，如果如果涉及到多个进程间的通信，就会设计到包问题，还需要设计消息格式，所以提出了更佳方便的消息队列。

分为匿名和具名管 道，匿名管道适用于有血缘关系的两个进程，fork后子进程会继承父进程的管道描符。

具名管道是通过某个具体的**全局文件名**来指代具体管道的，即管道名，通过这种方式可以让两个没有缘关系的进程使用管道。

管道的通信效率比较低，因此管道不适合进程间频繁地交换数据。

它的优点就是比较简单。

消息队列

不同于管道，消息队列可以实现双向通信，双方都可以读写。而且写入消息后就会立即返回，不需要塞等待对端读取数据，对端可以随时读取。

消息队列是保存在内核中的消息链表 ，在发送数据时，会分成一个一个独立的数据单元，也就是消息（数据块），消息体是用户自定义的数据类型，消息的发送方和接收方要约定好消息体的数据类型，以每个消息体都是固定大小的存储块，不像管道是无格式的字节流数据。如果进程从消息队列中读取消息体，内核就会把这个消息体删除。

消息队列生命周期随内核，如果没有释放消息队列或者没有关闭操作系统，消息队列会一直存在，而面提到的匿名管道的生命周期，是随进程的创建而建立，随进程的结束而销毁。

它的缺点就是：

通信不及时

消息有大小限制，不适合传大文件

存在用户态与内核态之间的数据拷贝开销

共享内存

消息队列以及管道的读取和写入的过程，都会有发生用户态与内核态之间的消息拷贝过程。那**共享内存**方式，就很好的解决了这一问题。

共享内存的机制，就是拿出一块虚拟地址空间来，映射到相同的物理内存中 。这样这个进程写入的东，另外一个进程马上就能看到了，都不需要拷贝来拷贝去，传来传去，大大提高了进程间通信的速度。

它的缺点就是交互时可能需要同步。

信号量

信号量其实是一个整型的计数器，主要用于实现进程间的互斥与同步，而不是用于缓存进程间通信的据。

通过互斥量和条件变量可以实现信号量的功能。

信号

信号用于单向事件通知，使用信号，一个进程可以随时发送一个事件到特定的进程、线程。并且接收线程不需要阻塞等待该事件，这是一个异步的事件。内核通常在执行完异常、中断、系统调用等从内态返回到用户态的时候检查是否有信号产生，从而执行信号对于的处理动作。

信号有60多个，其中前31个属于标准信号，后面的属于实时信号。**标准信号通过位图方式实现**，若有个相同信号，则只会被处理一次，**而实时信号通过队列来管理信号**，所有到达的信号都会被放入队列等待处理，不会出现覆盖丢失的问题。

对信号的处理动作主要有三种：

- 忽略
- 捕获，执行对于的信号处理函数
- 执行默认操作，通常都是杀死进程

要注意！**SIGKILL和SIGSTOP这两种信号不能被自行处理，也不能被忽略**，对它们的操作只能是系统认操作。

根本原因是为了向系统超级用户提供使进程终止或停止的可靠方法。

socket

mmap内存映射

```
#include <unistd.h>
#include <sys/mman.h>
void* mmap(void* addr, size_t length, int prot, int flags, int fd, off_t offset);
```

- addr 代表该区域的起始地址；
- length 代表该区域长度；
- prot 描述了这块新的内存区域的访问权限；
- flags 描述了该区域的类型；
- fd 代表文件描述符；
- offset 代表文件内的偏移值。

mmap 的功能非常强大，根据参数的不同，它可以用于创建共享内存，也可以创建文件映射区域用于升 IO 效率，还可以用来申请堆内存。决定它的功能的，主要是 prot, flags 和 fd 这三个参数，

我们分别来看看。prot 的值可以是以下四个常量的组合：

- PROT_EXEC，表示这块内存区域有可执行权限，意味着这部分内存可以看成是代码段，它里面存的往往是 CPU 可以执行的机器码。
- PROT_READ，表示这块内存区域可读。PROT_WRITE，表示这块内存区域可写。
- PROT_NONE，表示这块内存区域的页面不能被访问。

而 flags 的值可取的常量比较多，你可以通过 man mmap 查看，这里我只列举一下最重要的四种可值常量：

- MAP_SHARED：创建一个 **共享映射的区域**，多个进程可以通过共享映射的方式，来共享同一个件。这样一来，**一个进程对该文件的修改，其他进程也可以观察到**，这就实现了数据的通讯。
- MAP_PRIVATE：创建一个 **私有的映射区域**，多个进程可以使用私有映射的方式，来映射同一个件。但是，**当一个进程对文件进行修改时，操作系统就会为它创建一个独立的副本，这样它对文件的改，其他进程就看不到了，从而达到映射区域私有的目的。**
- MAP_ANONYMOUS：创建一个 **匿名映射**，也就是没有关联文件。使用这个选项时，**fd 参数必为空。**
- MAP_FIXED：一般来说，addr 参数只是建议操作系统尽量以 addr 为起始地址进行内存映射，但果操作系统判断 addr 作为起始地址不能满足长度或者权限要求时，就会另外再找其他适合的区域进映射。如果 flags 的值取是 MAP_FIXED 的话，就不再把 addr 看成是建议了，而是将其视为强制要。如果不能成功映射，就会返回空指针。

根据flag的不同参数，可以有四种组合:

私有映射：映射时所有进程共享这一块区域，若某个进程修改了这块区域，就拷贝一份只属于自己的域备份，类似于**写时拷贝**技术，因此它合文件映射组合，**适合加载共享库**

共享映射：所有进程共享一块区域，任何进程都可修改，而且修改完其他进程立即可见，**适合进程间信**，但是要有同步手段，避免竞争。

匿名映射：如果mmap()的fd设置为null，则不与任何文件相关联，单纯分配一块堆区内存，因此和有映射配合，**适合从堆区动态分配内存。**和共享映射配合适合父子进程间通信。

文件映射：和具体文件相关联，**适合多进程间通信，也适合快速高效进行IO。**

	私有映射	共享映射
匿名映射	私有匿名映射，常用于分配内存	共享匿名映射，常用于父子进程间共享内存，因为只有父子进程之间才能对同一个mmap的返回值进行访问
文件映射	私有文件映射，常用于加载动态库。如果只是读和执行，那么物理内存中只有一份副本。如果动态库中存在全局变量，则数据段就会被复制一份，变成每个进程独有的，而这正是我们所期望的	共享文件映射，可以用于多个进程之间的共享内存。共享内存是通过文件名建立起联系的

共享匿名映射

应用父子进程间的通信

```
#include #include #include #include
int main() {
    pid_t pid;
   char* shm = (char*)mmap(0, 4096, PROT_READ | PROT_WRITE, MAP_SHARED | MAP_ANON
MOUS, -1, 0);
   if (!(pid = fork())){
       sleep(1);
       printf("child got a message: %s\n", shm);
       sprintf(shm, "%s", "hello, father.");
       exit(0);
   }
   sprintf(shm, "%s", "hello, my child");
   sleep(2);
   printf("parent got a message: %s\n", shm);
   return 0;
}
```

运行结果：

```
gcc -o mm mmap_shm.c
$ ./mm
child got a message: hello,my child
parent got a message: hello, father.
```

文件映射的应用

正常的read/write系统io调用读取磁盘文件需要经过**两次拷贝**：

- **将文件数据从磁盘读取到内核缓冲区**
- **在从内核缓冲区传送给用户缓存区**

mmap是将文件与进程虚拟空间进行了映射，从而只需要一次拷贝，避免了将文件先拷贝到内核缓冲的操作。

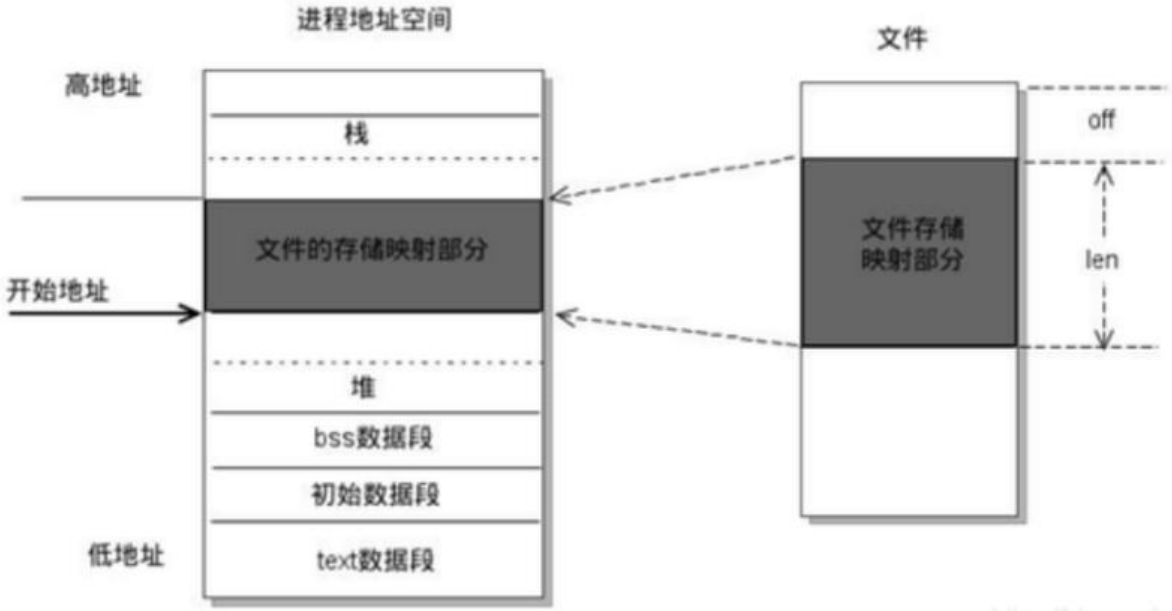
大概过程：

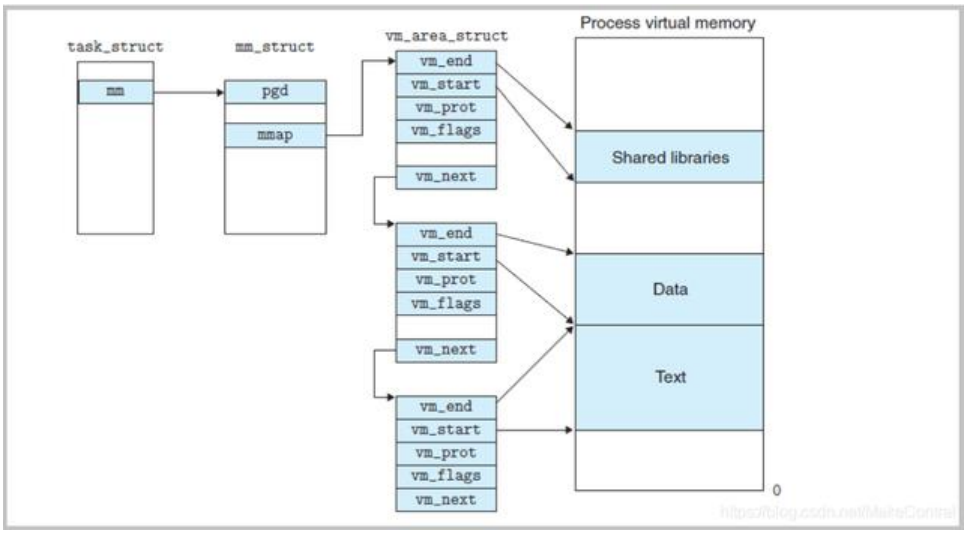
先分配出一片连续的虚拟地址空间，当访问该地址空间时，会由于没有建立页表而触发缺页异常，进，操作系统发现是文件映射所触发缺页，因此根据访问的偏移，从文件系统读取一定大小的页，并加pagecache，同时建立与你的mmap出来的虚拟地址的映射（页表）

所以，是操作系统帮忙建立了物理映射，并帮忙从文件系统中读出相应的内容。

当你写这片地址时，会触发权限异常，或者，cpu会有硬件管理脏页的机制，可以让操作系统知道你了数据进去，然后，在合适的时机，帮你写回文件系统

Linux通过下图的方式来组织虚拟内存。这里其他先不看，重点关注以下vm_area_struct。





在Linux内核，我们使用vm_area_struct结构来表示一个虚拟内存区域，一个具体的vm_area_struct含以下字段：

- **vm_start：指向这个区域的起始处。**
- **vm_end：指向这个区域的结束处。**
- **vm_port：描述这个区域包含的所有页的读写权限。**
- **vm_flags：描述这个区域是否是私有的还是共享的。**
- **vm_next：指向链表中下一个区域结构。**

使用mmap需要注意的一个关键点是，mmap映射区域大小必须是物理页大小(page_size)的整倍数（2位系统中通常是4k字节）。原因是，内存的最小粒度是页，而进程虚拟地址空间和内存的映射也是页为单位。为了匹配内存的操作，mmap从磁盘到虚拟地址空间的映射也必须是页。

既然内存映射可以提高文件的读取效率，为什么还要使用IO呢?

「内核缓冲区」实际上是磁盘高速缓存（PageCache）。

我们可以用 PageCache 来缓存最近被访问的数据，当空间不足时淘汰最久未被访问的缓存。

PageCache 的优点主要是两个：

- 缓存最近被访问的数据；
- 预读功能；

但是，在传输大文件（GB 级别的文件）的时候，PageCache 会不起作用，那就白白浪费 DMA 做的一次数据拷贝，造成性能的降低，即使使用了 PageCache 的零拷贝也会损失性能。

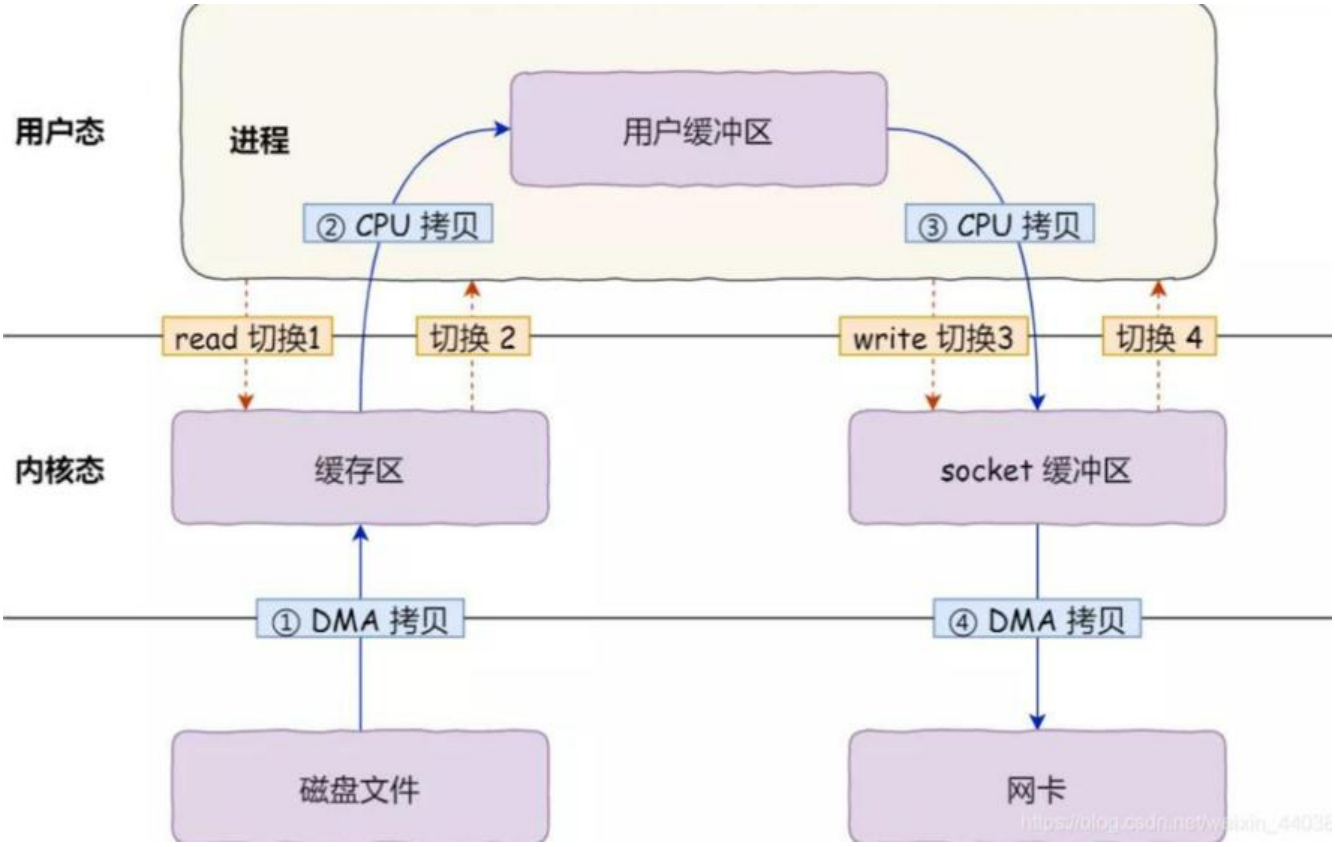
大文件采用异步和直接io方式传输。

零拷贝技术

传统拷贝方式

如果服务端要提供文件传输的功能，我们能想到的最简单的方式是：将磁盘上的文件读取出来，然后过网络协议发送给客户端。

传统 I/O 的工作方式是，数据读取和写入是从用户空间到内核空间来回复制，而内核空间的数据是通操作系统层面的 I/O 接口从磁盘读取或写入



期间共发生了 **4 次用户态与内核态的上下文切换**，因为发生了两次系统调用，一次是 read() ，一次是 write()，每次系统调用都得先从用户态切换到内核态，等内核完成任务后，再从内核态切换回用户态。

上下文切换到成本并不小，一次切换需要耗时几十纳秒到几微秒，虽然时间看上去很短，但是在

并发的场景下，这类时间容易被累积和放大，从而影响系统的性能。

其次，还发生了 **4 次数据拷贝**，其中两次是 DMA 的拷贝，另外两次则是通过 CPU 拷贝的，下说一下这个过程：

● 第一次拷贝，把磁盘上的数据拷贝到操作系统内核的缓冲区里，这个拷贝的过程是通过 DMA 搬运。

● 第二次拷贝，把内核缓冲区的数据拷贝到用户的缓冲区里，于是我们应用程序就可以使用这部分数了，这个拷贝到过程是由 CPU 完成的。

● 第三次拷贝，把刚才拷贝到用户的缓冲区里的数据，再拷贝到内核的 socket 的缓冲区里，这个过依然还是由 CPU 搬运的。

● 第四次拷贝，把内核的 socket 缓冲区里的数据，拷贝到网卡的缓冲区里，这个过程又是由 DMA 运的。

我们回过头看这个文件传输的过程，我们只是搬运一份数据，结果却搬运了 4 次，过多的数据拷无疑会消耗 CPU 资源，大大降低了系统性能。

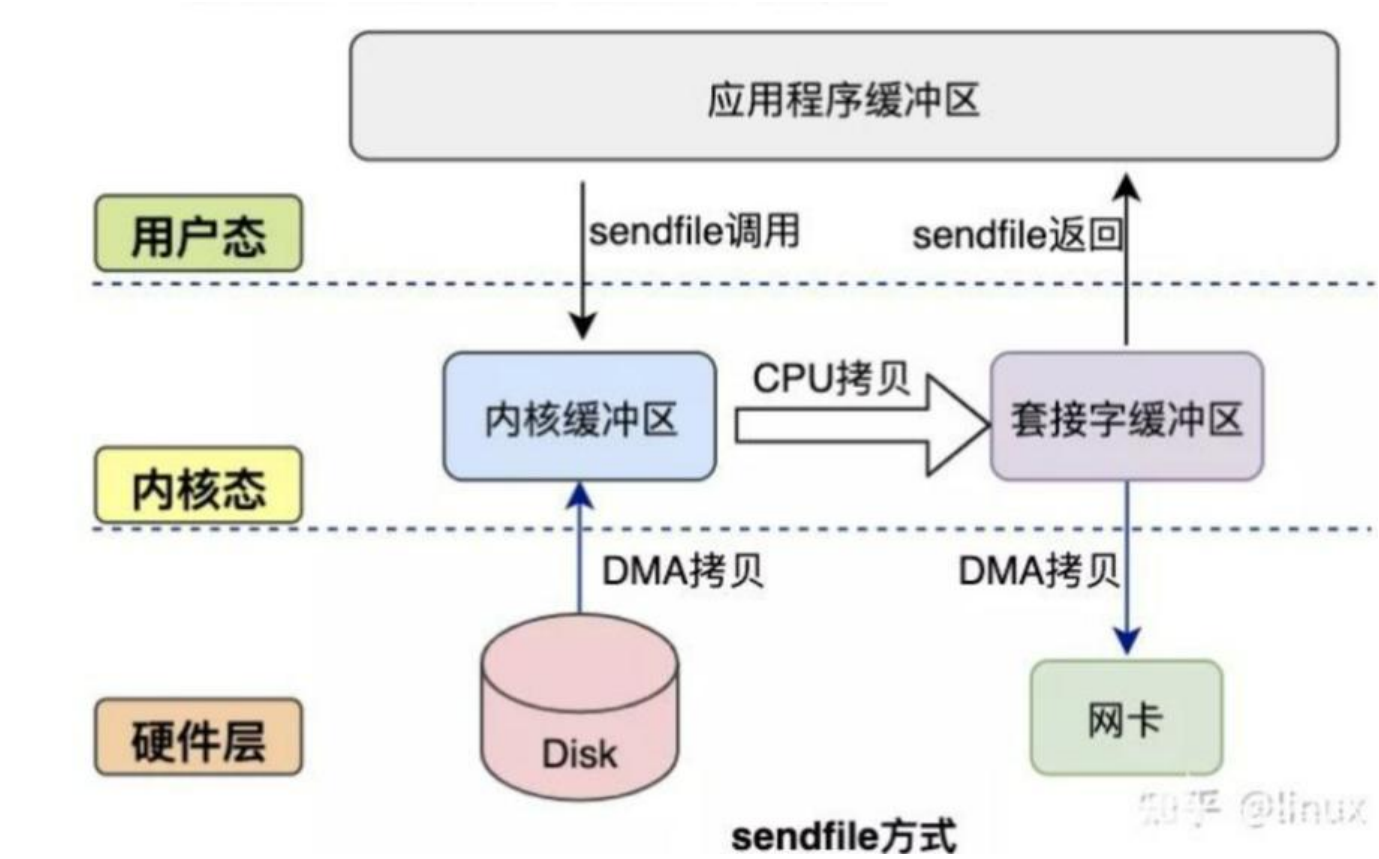
这种简单又传统的文件传输方式，存在冗余的上文切换和数据拷贝，在高并发系统里是非常糟糕，多了很多不必要的开销，会严重影响系统性能。

所以，要想提高文件传输的性能，就需要减少「用户态与内核态的上下文切换」和「内存拷贝」次数。

sendfile方式

sendfile系统调用是在 Linux 内核2.1版本中被引入，它建立了两个文件之间的传输通道。

sendfile方式只使用一个函数就可以完成之前的read+write 和 mmap+write的功能，这样就少了2次态切换，由于数据不经过用户缓冲区，因此该数据无法被修改。



原文链接：C++& 计算机基础知识

从图中可以看到，应用程序只需要调用sendfile函数即可完成，只有**2次状态切换**、**1次CPU拷贝**、**2 DMA拷贝**。

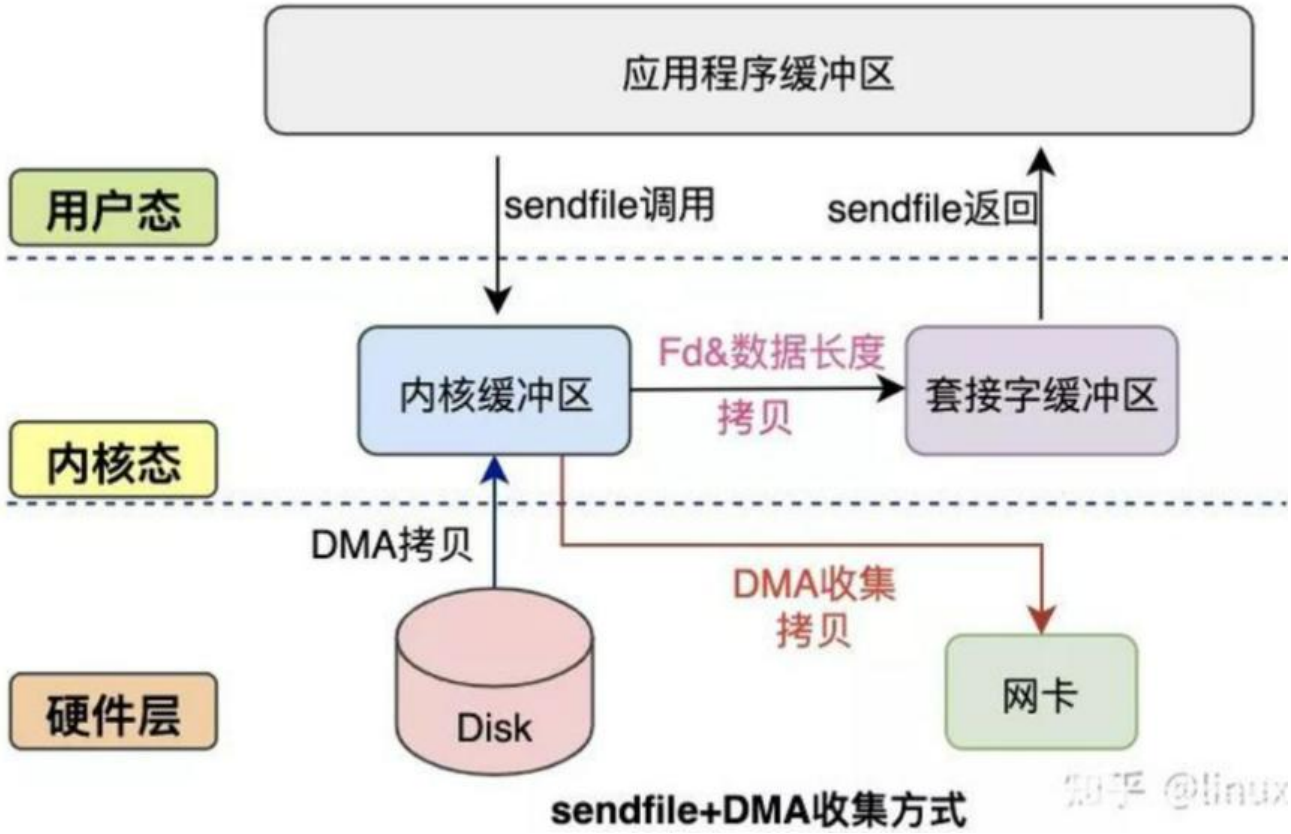
但是sendfile在内核缓冲区和socket缓冲区仍然存在一次CPU拷贝，或许这个还可以优化。

sendfile+DMA收集

Linux 2.4 内核对 sendfile 系统调用进行优化，但是**需要硬件DMA控制器的配合。**

升级后的sendfile将内核空间缓冲区中对应的数据描述信息（文件描述符、地址偏移量等信息）记录到ocket缓冲区中。

DMA控制器根据socket缓冲区中的地址和偏移量将数据从内核缓冲区拷贝到网卡中，从而省去了内空间中仅剩1次CPU拷贝。



这种方式有2次状态切换、**0次CPU拷贝**、2次DMA拷贝，但是仍然无法对数据进行修改，并且需要硬层面DMA的支持，并且sendfile只能将文件数据拷贝到socket描述符上，有一定的局限性。

C++模板编译模型，为什么模板的声明和定义要同时放在一个h文件中

使用C/C++进行编程时，一般会使用头文件以使定义和声明分离，并使得程序以模块方式组织。将函声明、类的定义放在头文件中，而将函数实现以及类成员函数的定义放在独立的文件中。

但是对于模板来说，这种方式是行不通的，具体的例子如下：

test.h文件：

```
template<class T>
class A
{
public:
void f();
}
```

test.cpp文件：

```
#include "test.h"
template<class T>
void A<T>::f()
{
.....
}
```

main.cpp文件：

```
#include "test.h"
int main()
{
A<int> a;
a.f();
return 0;
}
```

在编译时，会生成两个obj文件，在main.obj文件中找不到A<int>::f的实现，编译器认为这些函数的现是在其他源码文件中的，编译器不会报错，并将其看做外部链接类型，因为连接器会最终将所有的进制文件进行连接，从而完成符号查找，形成一个可执行文件。也就是需要在链接的时候从其他obj件中找到A<int>::f()定义处的二进制码，将其所在的地址给main.obj文件。这时问题就出现了，在链的时候，连接器在test.obj中找不到A<int>::f的实现（因为A没有被实例化），链接失败，发出了“法解析的外部命令”错误。换句话说，尽管编译器也编译了包含模板定义的源码文件temp.cpp，但该文件仅仅是模板的定义，而并没有真正的实例化出具体的函数来。因此在链接阶段，编译器进行符查找时，发现源码文件中的符号，在所有二进制文件中都找不到相关的定义，因此就报错了

首先要明白，C++中每一个对象所占的空间大小，对象的内存分布都是在编译时期就确定下来的。而于模板类来说，对象占空间的大小和内存分布是不知道的，依所套用的类型而定，比如A为模板类，A<int>类对象所占的空间大小和内存分布显然不同于A<double>。也就是说，

当编译器看到模板定义的时候，它不立即产生代码，他会**推迟实例化**。只有在看到用到模板时，如调了函数模板或调用了类模板的对象的时候，编译器才产生特定类型的模板实例。

一般而言，当调用函数的时候，编译器只需要看到函数的声明。类似地，定义类类型的对象时，定义必须可用，但成员函数的定义不是必须存在的。因此，应该将类定义和函数声明放在头文件中，普通函数和类成员函数的定义放在源文件中。

模板则不同：要进行实例化，编译器必须能够访问定义模板的源代码，也就是模板的定义和实例的代码(使用模板类时才会实例化)。当调用函数模板或类模板的成员函数的时候，编译器需要函数定，需要那些通常放在源文件中的代码。

因此，对于未实例化的模板类，编译器无法确定其大小，所以略过对模板类的编译，在编译时只检查些与模板无关的错误。而此时如果模板类的声明和定义中有错误的话，编译器就检查不到。

所以，结论就是，把模板的定义和实现都放到头文件中

关于模板实例化的理解

模板定义好后并不会立即被编译，也就是在符号表中不会有它相关的任何内容，直到在同一个文件中该模板的实例化代码，也就是指明T的具体类型后才会生成实例代码。

这是由于没确定T的具体类型，就无法分配内存空间，很多内容就无法确定。

c++11特性

类型方面

auto 和 decltype——类型推导，避免类型名称的不必要重复，方便使用

using--可以轻松的定义别名

long long——更长的整数类型

enum class——枚举值带有作用域的强类型枚举，而且可以指定底层类型

nullptr——给空指针一个名字（§4.2.6）

constexpr 函数——在编译期进行求值的函数

移动语义和右值引用——减少数据拷贝（§4.2.3）

万能引用、引用折叠、完美转发

类相关的

override 和 final、delete——用于管理大型类层次结构的明确语法

类内初始化——给数据成员一个默认值，这个默认值可以被构造函数中的初始化所取代

函数相关

noexcept——确保函数不会抛出异常的方法（§4.5.3）

lambda 表达式——匿名函数对象（§4.3.1）

- std::function--函数包装器
- 用std::bind-- 可以将可调用对象和参数一起绑定，绑定后的结果使用 std::function 进行保存，并迟调用到任何我们需要的时

候

模板相关

- 变参模板——可以处理任意个任意类型的参数的模板（§4.3.2）
- 模板别名——能够重命名模板并为新名称绑定一些模板参数（§4.3.3）
-

stl相关

新增内容

unique_ptr 和 shared_ptr——依赖 RAII（§2.2.1）的资源管理指针（§4.2.4）

以及对应的make_shared<>()和make_unique<>（c++14才引入）

- 正则表达式匹配（§4.6）
- 随机数——带有许多生成器（引擎）和多种分布（§4.6）
- 时间——time_point 和 duration（§4.6）

容器相关

范围 for——内部通过迭代器，对整个容器的简单顺序遍历（§4.2.2

- unordered_map 等——哈希表
- forward_list——单向链表
- array——具有固定常量大小的数组，并且会记住自己的大小
- emplace 运算——在容器内直接构建对象，避免拷贝
- exception_ptr——允许在线程之间传递异常

线程和同步变量相关

thread——包括thread_local局部变量，（§4.1.2）

mutex、condition_variable——为基本的系统层级的并发提供了线程安全、可移植的支持

std::unique_lock<std::mutex>

lock_guard和unique_lock的区别的，后者能够中途解锁，因此和条件变量配合使用

atomic 变量

malloc原理

malloc内部的具体实现，基于伙伴系统的链表数组方式

malloc 的实现，在历史上先后共有几十种策略，这些策略往往就是上述三种算法的组合。

malloc具体实现

具体到 glibc 中的 malloc 实现，它就采用了分桶与伙伴系统的策略，但是它的每个桶里的内存不是定大小的，而是采用了将 1 ~ 4 字节的块挂到第一个链表里，将 5 ~ 8 字节的块挂到第二个链表里，9-16 字节的块挂到第三个链表里，依次类推。

在单个链表内部则采用 naive 的分配方式，比如要分配 5 个字节的内存块，我们会先在 5 ~ 8 这个表里查找，如果查找到的内存大小是 8 字节的，那就会将这个区域分割成 5 字节和 3 字节两个部分其中 5 字节用于分配，剩余的 3 字节的空闲区域则会挂载到 1-4 这个链表里，其实也就是伙伴系统方式。

可见 malloc 的实现策略是比较灵活的，针对不同的场景，不同的分配策略的性能表现也是不一样的很多公司的基础平台都选择自己实现内存池来提供 malloc 接口，这样可以更好地服务本公司的业务。

最著名的例子就是 Google 公司实现的 Tcmalloc 库。Tcmalloc 相比起其他的 malloc 实现，最大的进是在多线程的情况下性能提升。我们知道，在多线程并发地分配内存时，每次分配都要对 free list 行加锁以避免并发程序带来的问题，这就容易形成性能瓶颈。

	简单算法	分桶	伙伴系统
碎片	会产生块间碎片，块内无碎片	块间无碎片，但会有块内碎片。块内浪费有可能很严重	可以动态割内存区域，块内无碎片，块间碎片不严重
分配	分为First Fit/Next Fit/Best Fit等查询方法，最差情况下是O(n)	效率高，时间复杂度是O(1)	最优情况下是O(1)，最差情况下是O(lg n)
释放	要在链表中查询它的前一个区域和后一个区域是否是空闲区域，如果是则合并成一个更大的区域	直接还到相应大小的链里，时间复杂度是O(1)	要判断伙伴是否空闲，如果空闲则合并。最好情况是O(1)，最差情况是O(lg n)

GNU2.9的标准分配器

一种定长的链表数组方式，它的问题是会产生内部随时。比如要分配30字节的数据，malloc找到3号置分配一个32字节的块，此时就产生了两字节的内部内存碎片。解决办法就是上述所说的方法。

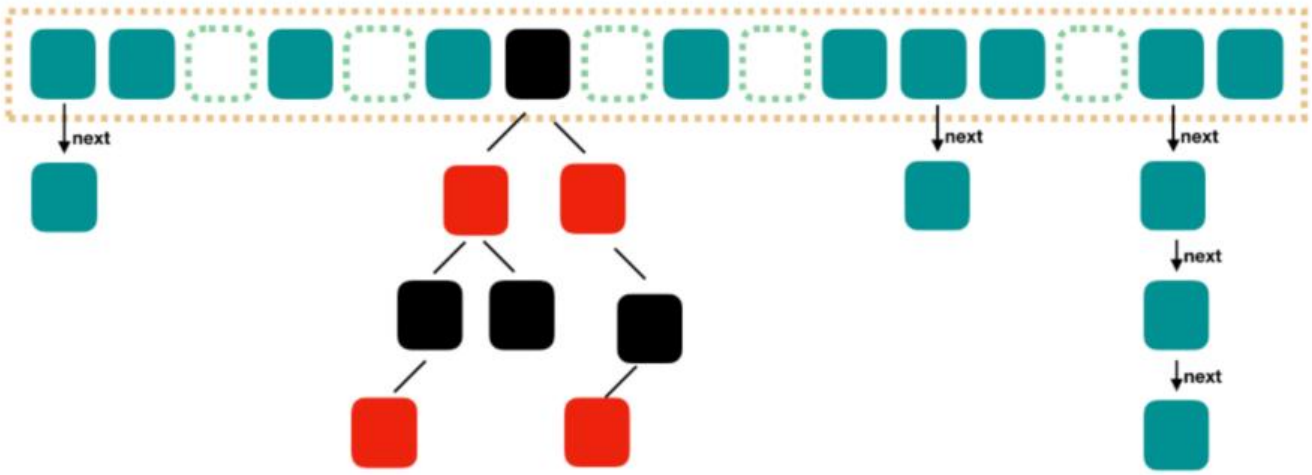
即让每个链表存储长度近似的内存块，然后在利用伙伴系统动态灵活地分裂与合并，最大限度地避免部和外部碎片。1

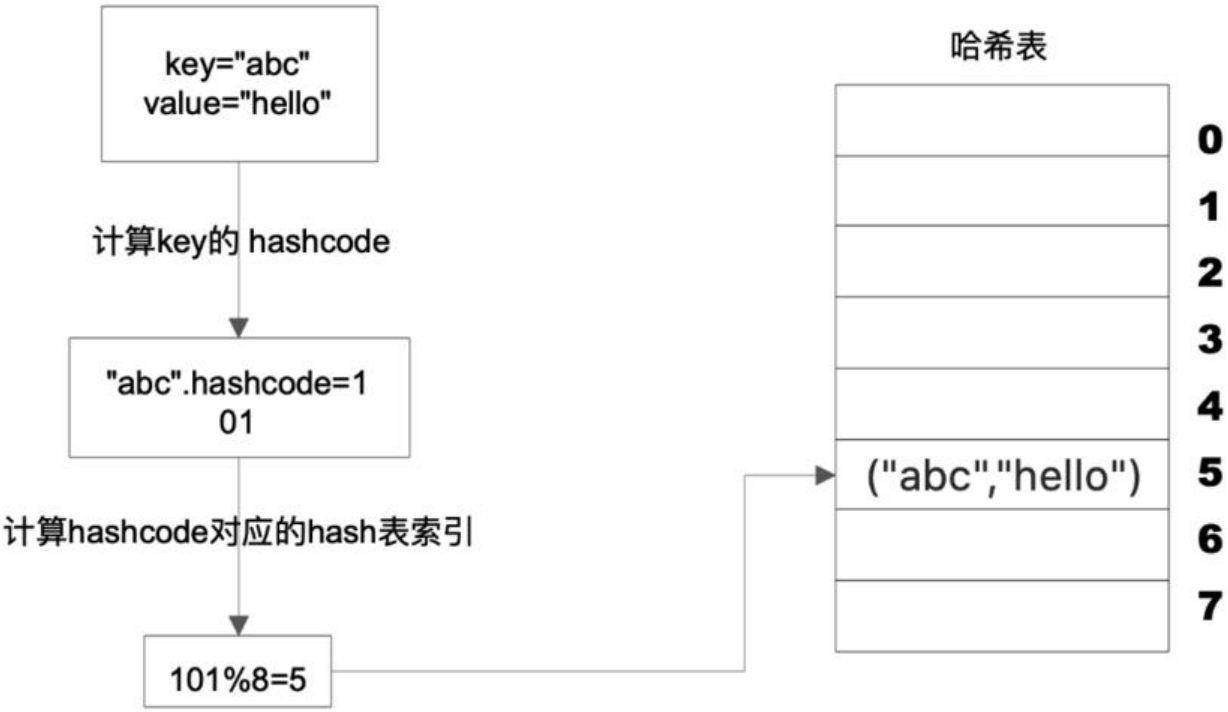
Hash 表

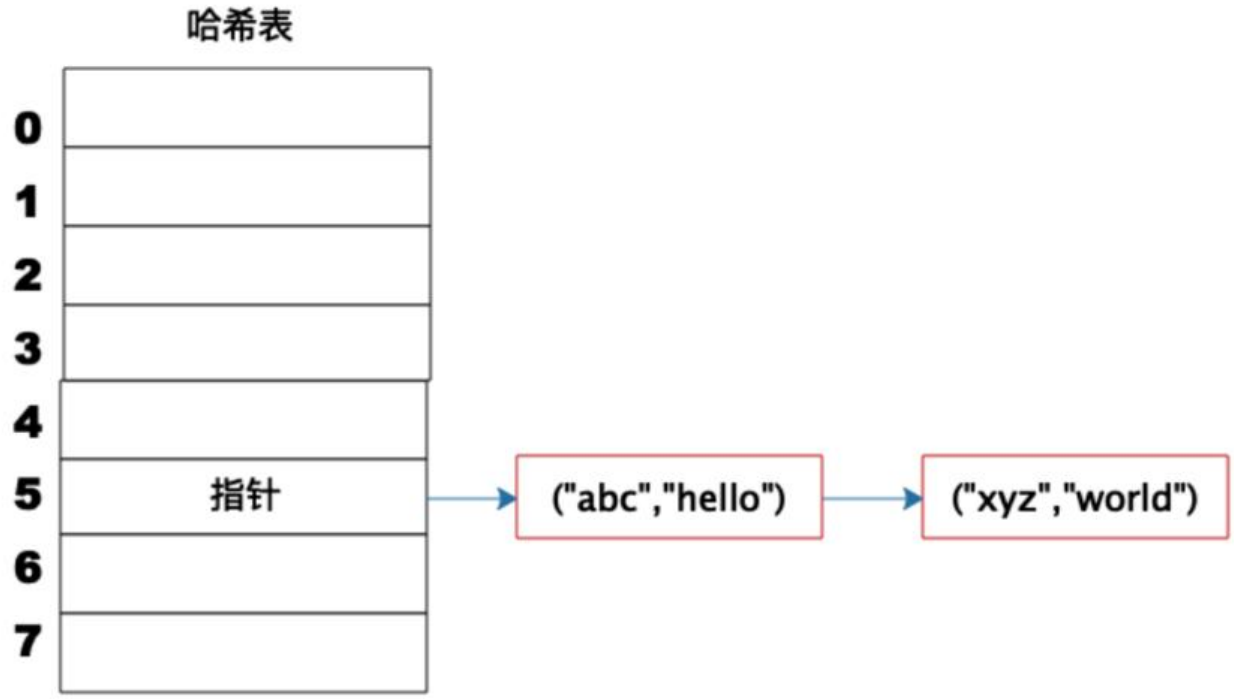
Hash 表中数据以 Key、Value 的方式存储

Hash 表实际上是一个数组+链表+红黑树，如果我们能够根据 Key 计算出数组下标，那么就可快速在数组中查找到需要的 Key 和 Value，最简单的方法就是余数法。**这样获取到的 index 查找数的时间复杂度就是 O(1)。 但是，这只是理想情况，**很多时候会出现 hash 冲突，就是两个不相同的 k y hash值却相同，这时候就要用到链表，链表查找的时间复杂度是 O(n)，链表长度达到一定值后会化为红黑树， 并且红黑树在满足在一定条件的时候会再次退回链表。**

Java8 HashMap 结构







上图这个例子中，Key 是字符串 abc，Value 是字符串 hello。

我们先计算 Key 的哈希值，得到 101 这样一个整型值。然后用 101 对 8 取模，这个 8 是哈希表数的长度。101 对 8 取模余 5，这个 5 就是数组的下标，这样就可以把（“abc”，“hello”）这样一个 Key、Value 值存储在下标为 5 的数组记录中。

当我们要读取数据的时候，只要给定 Key abc，还是用这样一个算法过程，先求取它的 HashCode 10，然后再对 8 取模，因为数组的长度不变，对 8 取模以后依然是余 5，那么我们到数组下标中去找 5 的这个位置，就可以找到前面存储进去的 abc 对应的 Value 值。

但是如果不同的 Key 计算出来的数组下标相同怎么办？7

HashCode101 对 8 取模余数是 5，HashCode109 对 8 取模余数还是 5，也就是说，不同的 Key 有能计算得到相同的数组下标，这就是所谓的 Hash 冲突，**解决 Hash 冲突常用的方法是链表法。**

事实上，（“abc”，“hello”）这样的 Key、Value 数据并不会直接存储在 Hash 表的数组中，因为组要求存储固定数据类型，主要目的是每个数组元素中要存放固定长度的数据。所以，数组中存储的是 Key、Value 数据元素的地址指针。一旦发生 Hash 冲突，只需要将相同下标，不同 Key 的数据元添加到这个链表就可以了。查找的时候再遍历这个链表，匹配正确的 Key。

因为有 Hash 冲突的存在，所以“**Hash 表的时间复杂度为什么是 O(1)?**”这句话并不严谨，极端情下，如果所有 Key 的数组下标都冲突，那么 Hash 表就退化为一条链表，查询的时间复杂度是 O(N) 但是作为一个面试题，“Hash 表的时间复杂度为什么是 O(1)”是没有问题的。

虚拟内存为什么总是从0x400000开始?

先来看看这个0X400000的地址在哪儿规定的。实际上在源码/usr/lib/ldscripts/elf_x86_64.x中可以

到如下定义

```
PROVIDE (__executable_start = SEGMENT_START("text-segment", 0x400000));
<br/>    . = SEGMENT_START("text-segment", 0x400000) + SIZEOF_HEADERS;
```

同时，对于X86-64位Linux系统而言，空出低端内存最重要的作用是为了**保证空指针可以触发访问缺页的异常SIGSEGV**，而不是访问了不该访问的资源从而导致了奇奇怪怪的问题。可能你回好奇，为什这里不直接从0x0000001开始做数据段呢？理论上说一个0就够用了呀？

这可以归结为以下几点原因：

- 考虑到多级页表的分配，省一点点不如多省一点，反正64位空间足够用，不再乎几页
- **默认页大小一般是4KB，但是实际会存在很多大页机制，而大页的大小默认是4MB**
- 再追问一个问题，是否能少分配一点呢？答案是当然可以，只要大于65536即可。65536的规定来于mmap_min_addr，可以通过/proc/sys/vm/mmap_min_addr查看。

————————————————

原文链接：https://blog.csdn.net/u013354486/article/details/111398333

几种压缩算法和其实现

压缩算法

deflate算法

- Lz4

LZ4是一种无损数据压缩算法，着重于压缩和解压缩速度。

- Lzo

LZO是致力于解压速度的一种数据压缩算法，LZO是Lempel-Ziv-Oberhumer的缩写，这个算法是无算法。

程序库

Zlib

Gzip

bzip2

Snappy

7z

zlib、gzip之间的关系

zlib是一种数据压缩程序库，它的设计目标是处理单纯的数据（而不管数据的来源是什么）。

gzip是一种文件压缩工具（或该压缩工具产生的压缩文件式），它的设计目标是处理单个的文件。

gzip在压缩文件中的数据时使用的就是zlib。为了保存与文件属性有关的信息，gzip需要在压缩文件（gz）中保存更多的头信息内容，而zlib不用考虑这一点。但gzip只适用于单个文件，所以我们在UNIX/Linux上经常看到的压缩包后缀都是.tar.gz或*.tgz，也就是先用tar把多个文件打包成单个文件，再用gzi压缩的结果。

压缩算法

deflate压缩算法

DEFLATE是同时使用了LZ77算法与哈夫曼编码（Huffman Coding）的一个无损数据压缩算法。DEFATE压缩与解代码可以在自由、通用的压缩库zlib上找到。常见的压缩算法如下：

zlib(RFC1950):一种格式，是对deflate进行了简单的封装，zlib = zlib头 + deflate编码的实际内容 + lib尾

gzip(RFC1952):一种格式，也是对deflate进行的封装，gzip = gzip头 + deflate编码的实际内容 + gzp尾

---

bzip2

实现了LZ77修改版以位（bit）而非字节（byte）为单元级别的操作，并通过马可夫链实现字典索引速率和压缩率优于bzip2，另有多线程优化的版本LZMA2涉及多种算法，主要流程包括先使用 Run-length Encoding 游程编码对原始数据进行处理，然后通过 Burrows-Wheeler Transform 转换（可逆处理一段输入数据使得相同字符连续出现的次数最大化），再用 Move-to-front transform 转换，后再次使用Run-length Encoding游程编码处理，接下来还会进行霍夫曼编码以及一系列相关处理，为复杂，速率劣于DEFLATE但压缩率更高

LZMA

实现了LZ77修改版以位（bit）而非字节（byte）为单元级别的操作，并通过马可夫链实现字典索引速率和压缩率优于bzip2，另有多线程优化的版本LZMA2

- Bzip2

bzip2是Julian Seward开发并按照自由软件 / 开源软件协议发布的数据压缩算法及程序。Seward在1996年7月第一次公开发布了bzip2 0.15版，在随后几年中这个压缩工具稳定性得到改善并且日渐流行，eward在2000年晚些时候发布了1.0版。bzip2比传统的gzip的压缩效率更高，但是它的压缩速度较慢。

- Deflater

DEFLATE是同时使用了LZ77算法与哈夫曼编码（Huffman Coding）的一个无损数据压缩算法，DEFATE压缩与解压的源代码可以在自由、通用的压缩库zlib上找到，zlib官网：http://www.zlib.net/ jd中对zlib压缩库提供了支持，压缩类Deflater和解压类Inflater，Deflater和Inflater都提供了native方。

- Gzip

gzip的实现算法还是deflate，只是在deflate格式上增加了文件头和文件尾，同样jdk也对gzip提供了持，分别是GZIPOutputStream和GZIPInputStream类，同样可以发现GZIPOutputStream是继承于eflaterOutputStream的，GZIPInputStream继承于InflaterInputStream，并且可以在源码中发现wrieHeader和writeTrailer方法。

- Lz4

LZ4是一种无损数据压缩算法，着重于压缩和解压缩速度。

- Lzo

LZO是致力于解压速度的一种数据压缩算法，LZO是Lempel-Ziv-Oberhumer的缩写，这个算法是无算法。

- Snappy

Snappy（以前称Zippy）是Google基于LZ77的思路用C++语言编写的快速数据压缩与解压程序库，在2011年开源。它的目标并非最大压缩率或与其他压缩程序库的兼容性，而是非常高的速度和合理的缩率。

相关常见名词说明

RAR： 商业软件WinRAR提供的压缩文件格式，压缩算法实现带专利（可能衍生自LZSS）

Zlib：zlib是一种数据压缩程序库，它的设计目标是处理单纯的数据（而不管数据的来源是什么）

Zip： 一种规范开放的压缩文件容器，被多种压缩软件实现，兼容多种压缩算法主要为DEFLATE

GZip： gnu/Linux下的文件压缩软件，提供gz压缩格式，压缩算法基于DEFLATE。它的设计目标是理单个的文件，gzip在压缩文件中的数据时使用的就是zlib。为了保存与文件属性有关的信息，gzip要在压缩文件（.gz）中保存更多的头信息内容，而zlib不用考虑这一点。但gzip只适用于单个文件，以我们在UNIX/Linux上经常看到的压缩包后缀都是.tar.gz或*.tgz，也就是先用tar把多个文件打包成个文件，再用gzip压缩的结果。

7-Zip： 开源跨平台压缩软件，提供7z压缩格式，压缩算法主要为Bzip2以及LZMA

Nginx服务器的反向代理

Nginx 是一个**高性能的HTTP和反向代理web服务器。**

正向代理

正向代理，其实是"代理服务器"，去和"客户端"进行交互。

举例：当我们的服务器直接访问外网(举例：美国服务器)的时候特别慢，使用vpn代理，通过vpn提供代理服务器(举例：香港服务器)。

我们请求代理服务器(香港服务器) --> 代理服务器请求外网(美国服务器)；然后外网响应给代理服务，代理服务器再把数据响应给我们的客户端服务器。

正向代理的用途

1. 突破访问限制

通过代理服务器，可以突破自身IP访问限制，访问国外网站，教育网等。

2. **提高访问速度**

通常代理服务器都设置一个较大的硬盘缓冲区，会将部分请求的响应保存到缓冲区中，当其他用户再问相同的信息时， 则直接由缓冲区中取出信息，传给用户，以提高访问速度

3. **隐藏客户端真实IP**

上网者也可以通过这种方法隐藏自己的IP，免受攻击。

反向代理

反向代理，其实是"代理服务器"，去和"目标服务器"进行交互。

是指以代理服务器来接受internet上的连接请求，然后将请求转发给内部网络上的服务器，并将从服务器上得到的结果返回给internet上请求连接的客户端，此时代理服务器对外就表现为一个反向代理服务器。

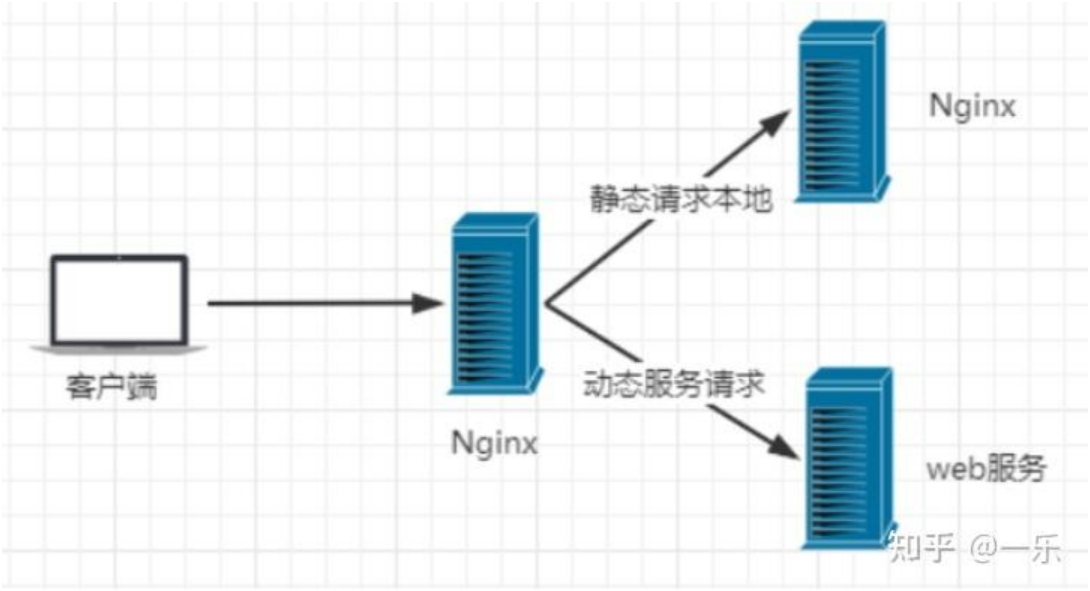
反向代理用途

1. **负载均衡**

反向代理服务器可以做[负载均衡]，根据所有真实服务器的负载情况，将客户端请求分发到不同的真服务器上。

2. **提高访问速度**

反向代理服务器可以对于**静态内容**及**短时间内有大量访问请求的动态内容**提供缓存服务，提高访问速
。



4

3. **提供安全保障**

反向代理服务器可以作为应用层防火墙，为网站提供对基于Web的攻击行为（例如DoS/DDoS）的防
，更容易排查恶意软件等。还可以为后端服务器统一提供加密和SSL加速（如SSL终端代理），提供HT
P访问认证等。

4. **隐藏服务器真实IP**

使用反向代理，可以对客户端隐藏服务器的IP地址。

负载均衡

Nginx提供的负载均衡策略有2种：**内置策略和扩展策略**

内置策略

1. 轮询

2. 加权轮询

3. Ip Hash

对客户端请求的ip进行hash操作，然后根据hash结果将同一个客户端ip的请求分发给同一台服务器进
处理，可以解决session不共享的问题。

针对Hash，产生了**一致性哈希**的问题

守护进程

基本概念

一种长期运行的进程：这种进程在后台运行，并且不跟任何的控制终端关联

实现步骤

1. fork（）子进程，关闭父进程
2. 调用setsid（）时子进程成为会话的领头进程
3. 关闭标准输入输出

```
void daemon_run()
{
    int pid;
    signal(SIGCHLD, SIG_IGN);
    //1）在父进程中，fork返回新创建子进程的进程ID；
    //2）在子进程中，fork返回0；
    //3）如果出现错误，fork返回一个负值；
    pid = fork();
    if (pid < 0)
    {
        std::cout << "fork error" << std::endl;
        exit(-1);
    }
    //第一步：父进程退出，子进程独立运行
    else if (pid > 0) {
        exit(0);
    }
    //之前parent和child运行在同一个session里,parent是会话（session）的领头进程,
    //parent进程作为会话的领头进程，如果exit结束执行的话，那么子进程会成为孤儿进程，并被ini
收养。
    //执行setsid()之后,child将重新获得一个新的会话(session)id。
    //这时parent退出之后,将不会影响到child了。

    //第二步
    setsid();
    int fd;
//第三步
    fd = open("/dev/null", O_RDWR, 0);
    if (fd != -1)
    {
        dup2(fd, STDIN_FILENO);
        dup2(fd, STDOUT_FILENO);
        dup2(fd, STDERR_FILENO);
    }
    if (fd > 2)
        close(fd);
}
```

setsid（）

那么，在创建守护进程时为什么要调用setsid函数呢？由于创建守护进程的第一步调用了fork函数来建子进程，再将 父进程退出。由于在调用了fork函数时，子进程全盘拷贝了父进程的会话期、进程组

控制终端等，虽然父进程退出了，但会话期、进程组、控制终端等并没有改变，因此，这还不是真正义上的独立开来，而setsid函数能够使进程完全独立出来，从而摆脱其他进程的控制。

会话和进程组

我们常见的Linux **session**一般是指shell session。Shell session 是终端中当前的状态，在**一个终端只能有一个 session。当我们打开一个新的终端时，总会创建一个新的 shell session。**这表明会话我们和shell交互的一个过程。

站在进程的角度看：会话可以看成是一个或多个进程组的集合，而进程组诗一个多多个进程的集合。个会话开始于用户登录，终止于用户退出。期间所有进程都属于这个会话期。

进程调用setsid函数建立一个新会话：

```
#include <unistd.h>

pid_t setsid(void);        //成功返回进程组ID，出错返回-1
```

如果调用此函数的进程不是一个进程组的组长进程，那么将会发生以下三件事：

1. **该进程成为新会话的首进程** （创建会话的进程）
2. **该进程成为一个新进程组的组长进程**
3. **新会话丢弃原有控制终端，该进程没有控制终端**

当然，如果该进程是组长进程，此函数会出错

僵尸进程

如何避免僵尸进程

1. 忽略SIGCHLD信号，由于子进程退出后会给父进程发SIGCHLD信号，此时如果忽略该信号，内核知道父进程对子进程的退出漠不关心，因此子进程直接退出
2. 父进程调用wait（）/watipid（）主动回收子进程
    1. 可以再SIGCHLD的信号处理函数中调用wait（）回收
    2. 也可以直接在父进程里回收
3. 两次fork的方式，使子进程由init进程负责回收

C/C++ include<>和include""的区别

首先明确，预处理器搜索的**顺序**永远都是：

（当前文件所在目录) --> 编译选项-I指定的目录 --> 默认的标准库目录

只是括号里的目录不一定会搜。

#include" "按照上面的顺序依次去三个地方搜索头文件，一但搜到就不继续往下搜了，意味着如果用目录下和标准库目录下有同名文件，会使用用户目录下的文件。

#include<>**不搜第一个部分**，即它不搜当前文件所在目录。后面俩个的搜索顺序是不变的。