

# 《2024年春招八股文冲刺版》

Summarized By Laoqin

微信: Laoqin9011

## 1、知识储备

说明: 红色是最重要的,

绿色次重要

没有渲染颜色为次次重要

以语法为主, 网络其次, 操作系统主要关注进程和线程的相关原理

### 1、C++

#### 1、指针和引用的区别

\*\*\*\*\*

- 指针是一个变量, 存储的是一个地址, 引用跟原来的变量实质上是同一个东西, 是原变量的别名
- 指针可以有多级, 引用只有一级
- 指针可以为空, 引用不能为NULL且在定义时必须初始化
- 指针在初始化后可以改变指向, 而引用在初始化之后不可再改变
- 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
- 当把指针作为参数进行传递时, 也是将实参的一个拷贝传递给形参, 两者指向的地址相同, 但不是同一个变量, 在函数中改变这个变量的指向不影响实参, 而引用却可以。
- 引用本质是一个指针, 同样会占4字节内存; 指针是具体变量, 需要占用存储空间 (具体情况还要具体分析)。

```
#include <iostream>

void add(int* p, int& r, int num) {
    *p += num;
    r += num;
}

int main() {
    int x = 10;
    int* p = &x;
    int& r = x;
    std::cout << *p << " " << r << std::endl;
    add(p, r, 5);
    std::cout << *p << " " << r << std::endl;
    double nn = 0;
    double& inf = nn;
    double *ptr = &nn;
    std::cout << sizeof(nn) << " " << sizeof(inf) << " " << sizeof(ptr) << std::endl; // 8 8 4
    return 0;
}
```

/\*在这个示例中, x是一个整数, p是指向x的指针, r是对x的引用。add 函数接收一个整数指针和一个整数引用, 以及一个要增加的值 num。函数使用指针和引用更改x的值, 将其增加 num。

在主函数中, 我们首先输出了指针和引用所指向的值, 然后调用 add 函数, 并再次输出指针和引用所指向的

值。由于传入的是指针和引用的引用，所以函数可以在不返回任何值的情况下更改传入的参数的值，这使得在有些情况下使用指针和引用更为方便。

需要注意的是，指针和引用的主要区别在于指针可以指向空值，而引用不能，指针可以随意改变指向的对象，而引用不能，指向对象后就不再改变。另外，指针需要使用\*运算符来访问所指向的对象，而引用不需要。

\*/

## 2、请你描述一下堆和栈的区别

(1) 管理方式不同。栈由操作系统自动分配释放，无需我们手动控制；堆的申请和释放工作由程序员控制，容易产生内存泄漏；

(2) 空间大小不同。每个进程拥有的栈大小要远远小于堆大小。理论上，进程可申请的堆大小为虚拟内存大小，进程栈的大小 64bits 的 Windows 默认 1MB，64bits 的 Linux 默认 10MB；

(3) 生长方向不同。堆的生长方向向上，内存地址由低到高；栈的生长方向向下，内存地址由高到低。

(4) 分配方式不同。堆都是动态分配的，没有静态分配的堆。栈有 2 种分配方式：静态分配和动态分配。静态分配是由操作系统完成的，比如局部变量的分配。动态分配由 `alloca()` 函数分配，但是栈的动态分配和堆是不同的，它的动态分配是由操作系统进行释放，无需我们手工实现。

(5) 分配效率不同。栈由操作系统自动分配，会在硬件层级对栈提供支持：分配专门的寄存器存放栈的地址，压栈出栈都有专门的指令执行，这就决定了栈的效率比较高。堆则是由 C/C++ 提供的库函数或运算符来完成申请与管理，实现机制较为复杂，频繁的内存申请容易产生内存碎片。显然，堆的效率比栈要低得多。

	堆	栈
管理方式	堆中资源由程序员控制（容易产生memory leak）	栈资源由编译器自动管理，无需手工控制
内存管理机制	系统有一个记录空闲内存地址的链表，当系统收到程序申请时，遍历该链表，寻找第一个空间大于申请空间的堆结点，删除空闲结点链表中的该结点，并将该结点空间分配给程序（大多数系统会在这块内存空间首地址记录本次分配的大小，这样delete才能正确释放本内存空间，另外系统会将多余的部分重新放入空闲链表中）	只要栈的剩余空间大于所申请空间，系统为程序提供内存，否则报异常提示栈溢出。（这一块理解一下链表和队列的区别，不连续空间和连续空间的区分，应该就比较好理解这两种机制的区别了）
空间大小	堆是不连续的内存区域（因为系统是用链表来存储空闲内存地址，自然不是连续的），堆大小受限于计算机系统中有效的虚拟内存（32bit 系统理论上是4G），所以堆的空间比较灵活，比较大	栈是一块连续的内存区域，大小是操作系统预定好的，windows下栈大小是2M（也有是1M，在编译时确定，VC中可设置）
碎片问题	对于堆，频繁的new/delete会造成大量碎片，使程序效率降低	对于栈，它是有点类似于数据结构上的一个先进后出的栈，进出一一对应，不会产生碎片。（看到这里我突然明白了为什么面试官在问我堆和栈的区别之前先问了我栈和队列的区别）
生长方向	堆向上，向高地址方向增长。	栈向下，向低地址方向增长。

分配方式	堆都是动态分配（没有静态分配的堆）	栈有静态分配和动态分配，静态分配由编译器完成（如局部变量分配），动态分配由alloca函数分配，但栈的动态分配的资源由编译器进行释放，无需程序员实现。
分配效率	堆由C/C++函数库提供，机制很复杂。所以堆的效率比栈低很多。	栈是其系统提供的数据结构，计算机在底层对栈提供支持，分配专门 寄存器存放栈地址，栈操作有专门指令。

### 3.将“引用”作为函数参数有哪些特点？

- 1、使用引用传参，不会创建拷贝，可以提升效率并节省了空间，比如我们要传一个很大的结构体，用引用传参就省去了拷贝这个结构体的开销。
- 2、在函数中对该变量进行修改，则参数返回后修改依然存在，与值传递不同；
- 3、C++的标准不允许复制构造函数传值参数，最好是传引用，在下面代码中，复制构造函数A(A other)传入的参数是A的一个实例。由于是传值参数，我们把形参复制到实参会调用复制构造函数。因此如果允许复制构造函数传值，就会在复制构造函数内调用复制构造函数，就会形成永无休止的递归调用从而导致栈溢出。因此C++的标准不允许复制构造函数传值参数，在Visual Studio和GCC中，都将编译出错。要解决这个问题，我们可以把构造函数修改为A(const A& other),也就是把传值参数改成常量引用。

```
class A
{
private:
    int value;
public:
    A(int n) { value = n; }
    A(A& other) { value = other.value; }
    void Print()
    {
        std::cout << value << std::endl;
    }
};

int main()
{
    A a = 10;
    A b = a;
    b.Print();
    return 0;
}
```

## 4、基类的虚函数表存放在内存的什么区，虚表指针vptr的初始化时间

首先整理一下虚函数表的特征：

- 虚函数表是全局共享的元素，即全局仅有一个，在编译时就构造完成
- 虚函数表类似一个数组，类对象中存储vptr指针，指向虚函数表，即虚函数表不是函数，不是程序代码，不可能存储在代码段
- 虚函数表存储虚函数的地址，即虚函数表的元素是指向类成员函数的指针，而类中虚函数的个数在编译时期可以确定，即虚函数表的大小可以确定，即大小是在编译时期确定的，不必动态分配内存空间存储虚函数表，所以不在堆中

根据以上特征，虚函数表类似于类中静态成员变量。静态成员变量也是全局共享，大小确定，因此最有可能存在全局数据区，测试结果显示：

虚函数表vtable在Linux/Unix中存放在可执行文件的只读数据段中(rodara)，这与微软的编译器将虚函数表存放在常量段存在一些差别

由于虚表指针vptr跟虚函数密不可分，对于有虚函数或者继承于拥有虚函数的基类，对该类进行实例化时，在构造函数执行时会对虚表指针进行初始化，并且存在对象内存布局的最前面。一般分为五个区域：栈区、堆区、函数区（存放函数体等二进制代码）、全局静态区、常量区

C++中虚函数表位于只读数据段（.rodara），也就是C++内存模型中的常量区；而虚函数则位于代码段（.text），也就是C++内存模型中的代码区。

## 5、new / delete 与 malloc / free的异同

### 相同点

- 都可用于内存的动态申请和释放

### 不同点

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

```
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对指针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存。后者均没有相关调用
- 后者需要库文件支持，前者不用
- new是封装了malloc，直接free不会报错，但是这只是释放内存，而不会析构对象

## 6、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，以保存指定类型的一个对象；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对象的指针
- delete的实现过程：对指针指向的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存

## 7、以下四行代码中"123"是否可以修改？

```
const char* a = "123";char *b = "123";const char c[] = "123";char d[] = "123";
```

第1，2行，"123"位于常量区，加不加const效果一样，都无法修改。而第三行，"123"本来在栈上，但是由于const关键字编译器可能将其优化到常量区，第四行："123"位于栈区。只有第四行可以修改。

## 8、strlen和sizeof区别？

编译器在编译时就计算出了 sizeof 的结果。而 strlen 函数必须在运行时才能计算出来。

并且 sizeof 计算的是数据类型占内存的大小，而 strlen 计算的是字符串实际的长度。

数组做 sizeof 的参数不退化，传递给 strlen 就退化为指针了。

sizeof 是一个操作符，strlen 是库函数。

sizeof 的参数可以是数据的类型，也可以是变量，而 strlen 只能以结尾为 '\0' 的字符串作参数。

```
int main()
{
    const char* str = "name"; 4
    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、a和&a有什么区别？

假设数组

```
int a[10];    int (*p)[10] = &a;
```

- a是数组名，是数组首元素地址，+1表示地址值加上一个int类型的大小，如果a的值是0x00000001，加1操作后变为0x00000005。\*(a + 1) = a[1]。
- &a是数组的指针，其类型为int (\*)[10]（就是前面提到的数组指针），其加1时，系统会认为是数组首地址加上整个数组的偏移（10个int型变量）1，值为数组a尾元素后一个元素的地址。
- 若(int \*)p，此时输出 \*p时，其值为a[0]的值，因为被转为int \*类型，解引用时按照int类型大小来读取。

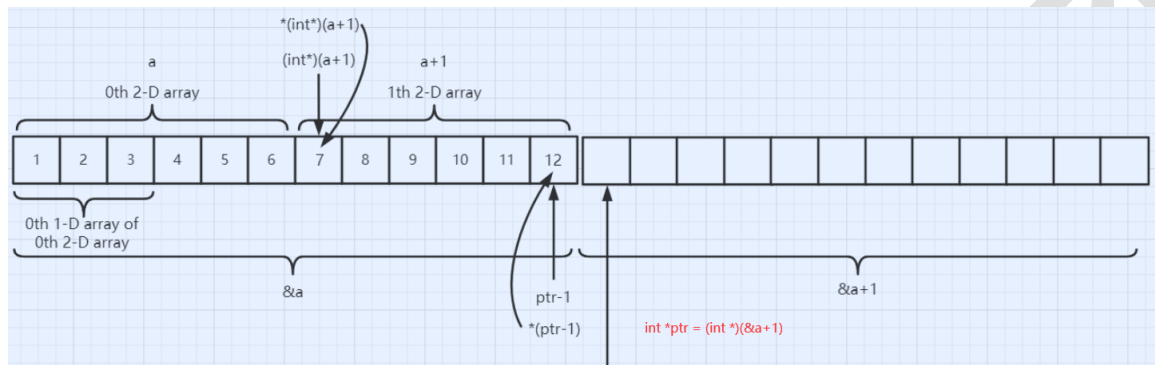
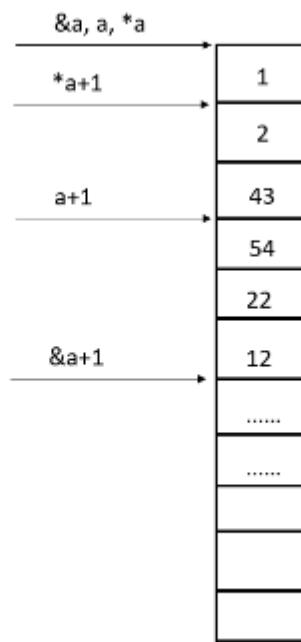
例子：求输出结果 ()

```
int main() {
    int a[2][2][3] = { {{1, 2, 3}, {4, 5, 6}}, {{7, 8, 9}, {10, 11, 12}} };
    int* ptr = (int*)(&a + 1);
    printf(" %d %d", *(int*)(a + 1), *(ptr - 1));
    printf(" %d %d", *(int*)(*a + 1), *(int*)(*a + 2));

    return 0;
}
```

A 7 12   B 1 6   C 1 3   D 7 9

Int a[2][3] = {{1,2,43},{54,22,12}}





## 10、数组名和指针（这里为指向数组首元素的指针）区别？

- 二者均可通过增减偏移量来访问数组中的元素。
- 数组名不是真正意义上的指针，可以理解为常指针，所以数组名没有自增、自减等操作。

当数组名当做形参传递给调用函数后，就失去了原有特性，退化成一般指针，多了自增、自减操作，但sizeof运算符不能再得到原数组的大小了。

求以下程序的输出

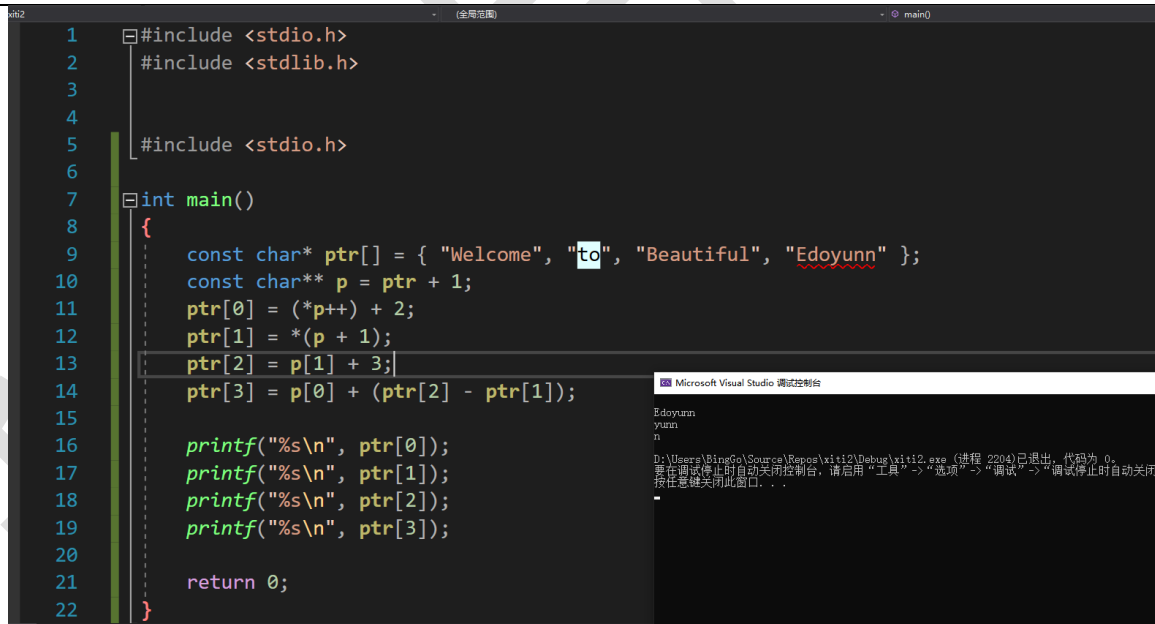
以下程序的输出是：

```
#include <stdio.h>

const char* c[] = { "Welcome", "to", "Beautiful", "Edoyun" };
const char** cp[] = { c + 3, c + 2, c + 1, c };
const char*** cpp = cp;

int main()
{
    printf("%s\n", **++cpp);
    printf("%s\n", *-- * ++cpp + 3);
    printf("%s\n", *cpp[-2] + 3);
    printf("%s\n", cpp[-1][-1] + 1);
    printf("\n");

    return 0;
}
```



```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4
5  #include <stdio.h>
6
7  int main()
8  {
9      const char* ptr[] = { "Welcome", "to", "Beautiful", "Edoyunn" };
10     const char** p = ptr + 1;
11     ptr[0] = (*p++) + 2;
12     ptr[1] = *(p + 1);
13     ptr[2] = p[1] + 3;
14     ptr[3] = p[0] + (ptr[2] - ptr[1]);
15
16     printf("%s\n", ptr[0]);
17     printf("%s\n", ptr[1]);
18     printf("%s\n", ptr[2]);
19     printf("%s\n", ptr[3]);
20
21     return 0;
22 }
```

Microsoft Visual Studio 调试控制台

Edoyun  
yun  
n

D:\Deez\BingGo\Source\Repos\xt12\Debug\xt12.exe (进程 2204) 已退出，代码为 0。  
要在调试停止时自动关闭控制台，请使用“工具”->“选项”->“调试”->“调试停止时自动关闭”  
放在任意键关闭此窗口。...

```

4
5     const char* c[] = { "Welcome", "to", "Beautiful", "Edoyun" };
6     const char** cp[] = { c + 3, c + 2, c + 1, c };
7     const char*** cpp = cp;
8
9     int main()
10    {
11        printf("%s\n", **++cpp); //cpp = cpp + 1 Beautiful
12        printf("%s\n ", *-- * ++cpp + 3); //come
13        printf("%s\n", *cpp[-2] + 3); //yun
14        printf("%s\n", cpp[-1][-1] + 1); //o
15        printf("\n");
16        system("pause");
17        return 0;
18    }
19

```

## 11、说说include头文件的顺序以及双引号""和尖括号<>的区别

1.区别:

(1)尖括号<>的头文件是系统文件，双引号""的头文件是自定义文件。

(2)编译器预处理阶段查找头文件的路径不--样。

2.查找路径:

(1)使用尖括号<>的头文件的查找路径:编译器设置的头文件路径-->系统变量。

(2)使用双引号""的头文件的查找路径:当前头文件目录-->编译器设置的头文件路径-->系统变量。

## 12、请你描述一下野指针和悬空指针的概念

野指针，未被初始化的指针

因此，为了防止出错，对于指针初始化时都是赋值为 `nullptr`，这样在使用时编译器就会直接报错，产生非法内存访问。

### • 悬空指针

悬空指针，指针最初指向的内存已经被释放了的一种指针。

```

int main(void) {
    int*P = nullptr;
    int* p2 = new int;
    P = p2;
    delete p2;
}

```

此时 `p`和`p2`就是悬空指针，指向的内存已经被释放。继续使用这两个指针，行为不可预料。需要设置为`p=p2=nullptr`。此时再使用，编译器会直接报错。

避免野指针比较简单，但悬空指针比较麻烦。c++引入了智能指针，C++智能指针的本质就是避免悬空指针的产生。

产生原因及解决办法:

野指针：指针变量未及时初始化 => 定义指针变量及时初始化，要么置空。

悬空指针：指针`free`或`delete`之后没有及时置空 => 释放操作后立即置空。

### 13、C++中struct和class的区别

#### 相同点

- 两者都拥有成员函数、公有和私有部分
- 任何可以使用class完成的工作，同样可以使用struct完成

#### 不同点

- 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
- class默认是private继承，而struct模式是public继承

#### 引申：C++和C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，能实现多态）
- C中struct是没有权限的设置，且struct中只能是一些变量的集合体，可以封装数据却不可以隐藏数据，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（为了与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中必须在结构标记前加上struct，才能做结构类型名（除：typedef struct class{};）；C++中结构体标记（结构体名）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

### 14、define宏定义和const的区别

#### 编译阶段

define是在编译的**预处理**阶段起作用，而const是在编译、运行的时候起作用

#### 安全性

•

- define只做替换，不做类型检查和计算，也不求解，容易产生错误，一般最好加上一个大括号包住全部的内容，要不然很容易出错
- const常量有数据类型，编译器可以对其进行类型安全检查

## 内存占用

- define只是将宏名称进行替换，在内存中会产生多份相同的备份。const在程序运行中只有一份备份，且可以执行常量折叠，能将复杂的表达式计算出结果放入常量表
- 宏替换发生在编译阶段之前，属于文本插入替换；const作用发生于编译过程中。
- 宏不检查类型；const会检查数据类型。
- 宏定义的数据没有分配内存空间，只是插入替换掉；const定义的变量只是值不能改变，但要分配内存空间。

## 15、C++中const和static的作用

### static

- 不考虑类的情况
  - 隐藏。所有不加static的全局变量和函数具有全局可见性，可以在其他文件中使用，加了之后只能在该文件所在的编译模块中使用
  - 默认初始化为0，包括未初始化的全局静态变量与局部静态变量，都存在全局未初始化区
  - 静态变量在函数内定义，始终存在，且只进行一次初始化，具有记忆性，其作用范围与局部变量相同，函数退出后仍然存在，但不能使用
- 考虑类的情况
  - static成员变量：只与类关联，不与类的对象关联。定义时要分配空间，不能在类声明中初始化，必须在类定义体外部初始化，初始化时不需要标示为static；可以被非static成员函数任意访问。
  - static成员函数：不具有this指针，无法访问类对象的非static成员变量和非static成员函数；**不能被声明为const、虚函数和volatile**；可以被非static成员函数任意访问

### const

- 不考虑类的情况
  - const常量在定义时必须初始化，之后无法更改
  - const形参可以接收const和非const类型的实参，例如

```
// i 可以是 int 型或者 const int 型
void fun(const int& i)
{
    //...
}
```

- 考虑类的情况
  - const成员变量：不能在类定义外部初始化，只能通过构造函数初始化列表进行初始化，并且必须有构造函数；不同类对其const数据成员的值可以不同，所以不能在类中声明时初始化
  - const成员函数：const对象不可以调用非const成员函数；非const对象都可以调用；不可以改变非mutable（用该关键字声明的变量可以在const成员函数中被修改）数据的值

## 16、类的对象存储空间？

- 非静态成员的数据类型大小之和。
- 编译器加入的额外成员变量（如指向虚函数表的指针）。为
- 了边缘对齐优化加入的padding。

空类(无非静态数据成员)的对象的size为1，当作为基类时，size为0.

## 17、初始化和赋值的区别

- 对于简单类型来说，初始化和赋值没什么区别
- 对于类和复杂数据类型来说，这两者的区别就大了，举例如下：

```
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;
}
```

## 18 不引入第三个变量的情况下，如何交换两个整数的值

```
//使用异或运算
#include <stdio.h>
int main()
{
    int x,y;
    printf("input two datas: \n");
    //输出提示信息
    scanf("%d %d", &x, &y);
    //将输入的数值分别存
    printf("x = %d, y = %d\n", x, y);
    x = x ^ y;
    //交换两个变量值的过程
    y = y ^ x;
    x = x ^ y;
    printf("x = %d, y = %d\n", x, y);
    return 0;
}
```

## 19 关于动态绑定和静态绑定：以下代码在VS2019上编译和运行结果是 ()

```
#include <stdio.h>
class A {
public:
    void test() { printf("test A"); }
};
int main() {
    A* pA = NULL;
    pA->test();
    return 0;
}
```

A 编译出错    B 程序运行崩溃    C 输出 "test A"    D 输出乱码

因为对于非虚成员函数，C++这门语言是静态绑定的。这也是C++语言和其它语言Java, Python的一个显著区别。以此下面的语句为例：

这语句的意图是：调用对象 pA 的 test 成员函数。如果这句话在Java或Python等动态绑定的语言之中，编译器生成的代码大概是：

找到 pA 的 test 成员函数，调用它。（注意，这里的找到是程序运行的时候才找的，这也是所谓动态绑定的含义：运行时才绑定这个函数名与其对应的实际代码。有些地方也称这种机制为迟绑定，晚绑定。）

但是对于C++。为了保证程序的运行时效率，C++的设计者认为凡是编译时能确定的事情，就不要拖到运行时再查找了。所以C++的编译器看到这句话会这么干：

- 1：查找 pA 的类型，发现它有一个非虚的成员函数叫 test。（编译器干的）
- 2：找到了，在这里生成一个函数调用，直接调 A::test ( pA )。

所以到了运行时，由于 test ()函数里面并没有任何需要解引用 pA 指针的代码，所以真实情况下也不会引发segment fault。这里对成员函数的解析，和查找其对应的代码的工作都是在编译阶段完成而非运行时完成的，这就是所谓的静态绑定，也叫早绑定。

正确理解C++的静态绑定可以理解一些特殊情况下C++的行为。

## 20 编程实现strcpy函数

已知strcpy函数的原型是：

```
char * strcpy(char * strDest, const char * strSrc);
```

(1) 不调用库函数，实现strcpy 函数。

(2) 解释为什么要返回char \*.

## 21、C和C++的类型安全

### 什么是类型安全？

类型安全很大程度上可以等价于内存安全，类型安全的代码不会试图访问自己没被授权的内存区域。“类型安全”常被用来形容编程语言，其根据在于该门编程语言是否提供保障类型安全的机制；有的时候也用“类型安全”形容某个程序，判别的标准在于该程序是否隐含类型错误。

类型安全的编程语言与类型安全的程序之间，没有必然联系。好的程序员可以使用类型不那么安全的语言写出类型相当安全的程序，相反的，差一点儿的程序员可能使用类型相当安全的语言写出类型不太安全的程序。绝对类型安全的编程语言暂时还没有。

#### (1) C的类型安全

C只在局部上下文中表现出类型安全，比如试图从一种结构体的指针转换成另一种结构体的指针时，编译器将会报告错误，除非使用显式类型转换。然而，C中相当多的操作是不安全的。以下是两个十分常见的例子：

- printf格式输出

```
1 #include <stdio.h>
2
3 int main()
4 {
5     printf("整型输出: %d\n", 10);
6     printf("浮点型输出: %f\n", 10);
7     return 0;
8 }
```

整型输出: 10  
浮点型输出: 0.000000

上述代码中，使用%d控制整型数字的输出，没有问题，但是改成%f时，明显输出错误，再改成%s时，运行直接报segmentation fault错误

- malloc函数的返回值

malloc是C中进行内存分配的函数，它的返回类型是void\*即空类型指针，常常有这样的用法char\* pStr=(char\*)malloc(100\*sizeof(char))，这里明显做了显式的类型转换。

类型匹配尚且没有问题，但是一旦出现int\* pInt=(int\*)malloc(100\*sizeof(char))就很可能带来一些问题，而这样的转换C并不会提示错误。

#### (2) 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语言，C++提供了一些新的机制保障类型安全：

- 操作符new返回的指针类型严格与对象匹配，而不是void\*
- 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；
- 引入const关键字代替#define constants，它是有类型、有作用域的，而#define constants只是简单的文本替换

- ◆ 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，当然改写为模板也能保证类型安全
- ◆ 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\_cast涉及更多具体的类型检查。

例1：使用void\*进行类型转换

```
1 #include <iostream>
2 using namespace std;
3
4 int main()
5 {
6     int i=5;
7     void* pInt=&i;
8     double d=*(double*)pInt;
9     cout<<"转换后输出："<< d <<endl;
10
11 }
```

转换后输出: 1.78416e-307

例2：不同类型指针之间转换

```
1 #include<iostream>
2 using namespace std;
3
4 class Parent{};
5 class Child1 : public Parent
6 {
7 public:
8     int i;
9     Child1(int e):i(e){}
10 };
11 class Child2 : public Parent
12 {
13 public:
14     double d;
15     Child2(double e):d(e){}
16 };
17 int main()
18 {
19     Child1 c1(5);
20     Child2 c2(4.1);
21     Parent* pp;
22     Child1* pc1;
23
24     pp=&c1;
25     pc1=(Child1*)pp; // 类型向下转换 强制转换，由于类型仍然为Child1*，不造成错误
26     cout<<pc1->i<<endl; //输出: 5
27
28     pp=&c2;
29     pc1=(Child1*)pp; //强制转换，且类型发生变化，将造成错误
30     cout<<pc1->i<<endl; // 输出: 1717986918
31     return 0;
32 }
33
34
35
36
```



上面两个例子之所以引起类型不安全的问题，是因为程序员使用不得当。第一个例子用到了空类型指针 `void*`，第二个例子则是在两个类型指针之间进行强制转换。因此，想保证程序的类型安全性，应尽量避免使用空类型指针 `void*`，尽量不对两种类型指针做强制转换。

## 22、为什么析构函数一般写成虚函数

由于类的多态性，基类指针可以指向派生类的对象，如果删除该基类的指针，就会调用该指针指向的派生类析构函数，而派生类的析构函数又自动调用基类的析构函数，这样整个派生类的对象完全被释放。

如果析构函数不被声明成虚函数，则编译器实施静态绑定，在删除基类指针时，只会调用基类的析构函数而不调用派生类析构函数，这样就会造成派生类对象析构不完全，造成内存泄漏。

所以将析构函数声明为虚函数是十分必要的。在实现多态时，当用基类操作派生类，在析构时防止只析构基类而不析构派生类的状况发生，要将基类的析构函数声明为虚函数。

```
1  #include <iostream>
2  using namespace std;
3
4  class Parent{
5  public:
6      Parent(){
7          cout << "Parent construct function" << endl;
8      };
9      ~Parent(){
10         cout << "Parent destructor function" << endl;
11     }
12 };
13
14 class Son : public Parent{
15 public:
16     Son(){
17         cout << "Son construct function" << endl;
18     };
19     ~Son(){
20         cout << "Son destructor function" << endl;
21     }
22 };
23
24 int main()
25 {
26     Parent* p = new Son();
27     delete p;
28     p = NULL;
29     return 0;
30 }
31 //运行结果:
32 //Parent construct function
33 //Son construct function
34 //Parent destructor function
35
36
37
```

将基类的析构函数声明为虚函数：

```

1  #include <iostream>
2  using namespace std;
3
4  class Parent{
5  public:
6      Parent(){
7          cout << "Parent construct function"      << endl;
8      };
9      virtual ~Parent(){
10         cout << "Parent destructor function" << endl;
11     }
12 };
13
14 class Son : public Parent{
15 public:
16     Son(){
17         cout << "Son construct function"      << endl;
18     };
19     ~Son(){
20         cout << "Son destructor function" << endl;
21     }
22 };
23
24 int main()
25 {
26     Parent* p = new Son();
27     delete p;
28     p = NULL;
29     return 0;
30 }
31 //运行结果:
32 //Parent construct function
33 //Son construct function
34 //Son destructor function
35 //Parent destructor function
36
37
38

```

但存在一种特例，在 CRTP 模板中，不应该将析构函数声明为虚函数，理论上所有的父类函数都不应该声明为虚函数，因为这种继承方式，不需要虚函数表。

## 23、构造函数能否声明为虚函数或者纯虚函数，析构函数呢？

析构函数：

- 析构函数可以为虚函数，并且一般情况下基类析构函数要定义为虚函数。
- 只有在基类析构函数定义为虚函数时，调用操作符delete销毁指向对象的基类指针时，才能准确调用派生类的析构函数（从该级向上按序调用虚函数），才能准确销毁数据。
- **析构函数可以是纯虚函数**，含有纯虚函数的类是抽象类，此时不能被实例化。但派生类中可以根据自身需求重新改写基类中的纯虚函数。

构造函数：

- ◆ 构造函数不能定义为虚函数。在构造函数中可以调用虚函数，不过此时调用的是正在构造的类中的虚函数，而不是子类的虚函数，因为此时子类尚未构造好。
- ◆ 虚函数对应一个vtable(虚函数表)，类中存储一个vptr指向这个vtable。如果构造函数是虚函数，就需要通过vtable调用，可是对象没有初始化就没有vptr，无法找到vtable，所以构造函数不能是虚函数。

## 24、C++中的重载、重写（覆盖）和隐藏的区别

### (1) 重载 (overload)

重载是指在同一范围定义中的同名成员函数才存在重载关系。主要特点是函数名相同，参数类型和数目有所不同，不能出现参数个数和类型均相同，仅仅依靠返回值不同来区分的函数。重载和函数成员是否是虚函数无关。举个例子：

```
1  class A{
2      ...
3      virtual int fun();
4      void fun(int);
5      void fun(double, double);
6      static int fun(char);
7      ...
8  }
9
10
11
```

### (2) 重写（覆盖） (override)

重写指的是在派生类中覆盖基类中的同名函数，**重写就是重写函数体，要求基类函数必须是虚函数且：**

- ◆ 与基类的虚函数有相同的参数个数
- ◆ 与基类的虚函数有相同的参数类型
- ◆ 与基类的虚函数有相同的返回值类型

举个例子：

```
1  //父类
2  class A{
3  public:
4      virtual int fun(int a){} 5
5      }
6  //子类
7  class B : public A{
8  public:
9      //重写,一般加override可以确保是重写父类的函数
10     virtual int fun(int a) override{} 11
11     }
12
13
14
```

重载与重写的区别：

- 重写是父类和子类之间的垂直关系，重载是不同函数之间的水平关系
- 重写要求参数列表相同，重载则要求参数列表不同，返回值不要求
- 重写关系中，调用方法根据对象类型决定，重载根据调用时实参与形参表的对应关系来选择函数体

### (3) 隐藏 (hide)

隐藏指的是某些情况下，派生类中的函数屏蔽了基类中的同名函数，包括以下情况：

- 两个函数参数相同，但是基类函数不是虚函数。**和重载的区别在于基类函数是否是虚函数。**举个例子：

```
1 //父类
2 class A{
3 public:
4     void fun(int a){
5         cout << "A中的fun函数" << endl; 6
6     }
7 };
8 //子类
9 class B : public A{
10 public:
11     //隐藏父类的fun函数
12     void fun(int a){
13         cout << "B中的fun函数" << endl; 14
14     }
15 };
16 int main(){
17     B b;
18     b.fun(2); //调用的是B中的fun函数
19     b.A::fun(2); //调用A中fun函数
20     return 0; 21
21 }
22
23
24
```

两个函数参数不同，无论基类函数是不是虚函数，都会被隐藏。和重载的区别在于两个函数不在同一个类中。举个例子：

```
//父类
class A {
public:
    virtual void fun(int a) {
        cout << "A中的fun函数" << endl;
    }
};
//子类
class B : public A {
public:
    //隐藏父类的fun函数
    virtual void fun(char* a) {
        cout << "A中的fun函数" << endl;
    }
};
int main() {
    B b;
    b.fun(2); //报错，调用的是B中的fun函数，参数类型不对
    b.A::fun(2); //调用A中fun函数
    return 0;
}
```

}

## 25、C++的多态如何实现

C++的多态性，一言以蔽之就是：

在基类的函数前加上**virtual**关键字，在派生类中重写该函数，运行时将会根据所指对象的实际类型来调用相应的函数，如果对象类型是派生类，就调用派生类的函数，如果对象类型是基类，就调用基类的函数。

举个例子：

```
1  #include <iostream>
2  using namespace std;
3
4  class Base{
5  public:
6      virtual void fun(){
7          cout << " Base::func()" <<endl; 8 }
9  };
10
11 class Son1 : public Base{
12 public:
13     virtual void fun() override{
14         cout << " Son1::func()" <<endl; 15
15     }
16 };
17
18 class Son2 : public
Base{ 19
19 };
20 };
21
22 int main()
23 {
24     Base* base = new Son1;
25     base->fun();
26     base = new Son2;
27     base->fun();
28     delete base;
29     base = NULL;
30     return 0; 31
31 }
32 // 运行结果
33 // Son1::func()
34 // Base::func()
35
36
```

例子中，Base为基类，其中的函数为虚函数。子类1继承并重写了基类的函数，子类2继承基类但没有重写基类的函数，从结果分析子类体现了多态性，那么为什么会出现多态性，其底层的原理是什么？这里需要引出虚表和虚基表指针的概念。

虚表：虚函数表的缩写，类中含有virtual关键字修饰的方法时，编译器会自动生成虚表

虚表指针：在含有虚函数的类实例化对象时，对象地址的前四个字节存储的指向虚表的指针



上图中展示了虚表和虚表指针在基类对象和派生类对象中的模型，下面阐述实现多态的过程：

- (1) 编译器在发现基类中有虚函数时，会自动为每个含有虚函数的类生成一份虚表，该表是一个一维数组，虚表里保存了虚函数的入口地址
- (2) 编译器会在每个对象的前四个字节中保存一个虚表指针，即**vptr**，指向对象所属类的虚表。在构造时，根据对象的类型去初始化虚指针vptr，从而让vptr指向正确的虚表，从而在调用虚函数时，能找到正确的函数
- (3) 所谓的合适时机，在派生类定义对象时，程序运行会自动调用构造函数，在构造函数中创建虚表并对虚表初始化。在构造子类对象时，会先调用父类的构造函数，此时，编译器只“看到了”父类，并为父类对象初始化虚表指针，令它指向父类的虚表；当调用子类的构造函数时，为子类对象初始化虚表指针，令它指向子类的虚表
- (4) 当派生类对基类的虚函数没有重写时，派生类的虚表指针指向的是基类的虚表；当派生类对基类的虚函数重写时，派生类的虚表指针指向的是自身的虚表；当派生类中有自己的虚函数时，在自己的虚表中将此虚函数地址添加在后面

这样指向派生类的基类指针在运行时，就可以根据派生类对虚函数重写情况动态的进行调用，从而实现多态性。

## 26、浅拷贝和深拷贝的区别

### 浅拷贝

浅拷贝只是拷贝一个指针，并没有新开辟一个地址，拷贝的指针和原来的指针指向同一块地址，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误。

## 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，释放内存了也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贝的。

```
#include <iostream>
#include <string.h>
using namespace std;
class Student
{
private:
    int num;
    char* name;
public:
    Student() {
        name = new char(20);
        cout << "Student" << endl;
    };
    ~Student() {
        cout << "~Student " << &name << endl;
        delete name;
        name = NULL;
    };
    Student(const Student& s) { //拷贝构造函数
        //浅拷贝，当对象的name和传入对象的name指向相同的地址
        name = s.name;
        //深拷贝
        //name = new char(20);
        //memcpy(name, s.name, strlen(s.name));
        cout << "copy Student" << endl;
    };
};

int main()
{
    // 花括号让s1和s2变成局部对象，方便测试
    Student s1;
    Student s2(s1); // 复制对象
}

system("pause");
return 0;
}

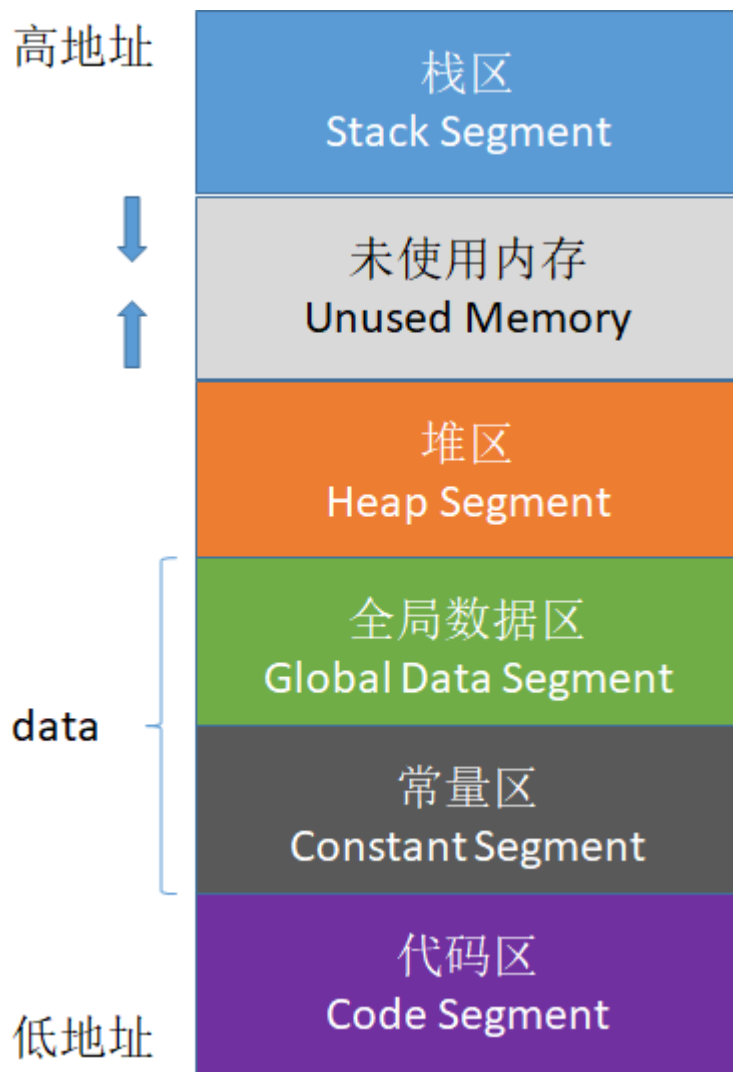
//浅拷贝执行结果:
//Student
//copy Student
//~Student 0x7fffed0c3ec0
//~Student 0x7fffed0c3ed0
//*** Error in `./tmp/815453382/a.out': double free or corruption (fasttop):
//0x0000000001c82c20 * **
//深拷贝执行结果:
//Student
//copy Student
//~Student 0x7fffebca9fb0
//~Student 0x7fffebca9fc0
```

从执行结果可以看出，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放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。

## 27、简要说明C++的内存分区

C++中的内存分区，分别是堆、栈、自由存储区、全局/静态存储区、常量存储区和代码区。如下图所示





**栈：**在执行函数时，函数内局部变量的存储单元都可以在栈上创建，函数执行结束时这些存储单元自动被释放。栈内存分配运算内置于处理器的指令集中，效率很高，但是分配的内存容量有限

**堆：**就是那些由 `new` 分配的内存块，他们的释放编译器不去管，由我们的应用程序去控制，一般一个 `new` 就要对应一个 `delete`。如果程序员没有释放掉，那么在程序结束后，操作系统会自动回收

**自由存储区：**就是那些由 `malloc` 等分配的内存块，它和堆是十分相似的，不过它是用 `free` 来结束自己的生命

**全局/静态存储区：**全局变量和静态变量被分配到同一块内存中，在以前的C语言中，全局变量和静态变量又分为初始化的和未初始化的，在C++里面没有这个区分了，它们共同占用同一块内存区，在该区定义的变量若没有初始化，则会被自动初始化，例如int型变量自动初始为0

**常量存储区：**这是一块比较特殊的存储区，这里面存放的是常量，不允许修改

**代码区：**存放函数体的二进制代码

《C/C++内存管理详解》：<https://chenqx.github.io/2014/09/25/Cpp-Memory-Management/>

## 28、C++的异常处理的方法

在程序执行过程中，由于程序员的疏忽或是系统资源紧张等因素都有可能导致异常，任何程序都无法保证绝对的稳定，常见的异常有：

- 数组下标越界

- 除法计算时除数为0
- 动态分配空间时空间不足
- ...

如果不及时对这些异常进行处理，程序多数情况下都会崩溃。

### (1) try、throw和catch关键字

C++中的异常处理机制主要使用**try**、**throw**和**catch**三个关键字，其在程序中的用法如下：

```

1  #include <iostream>
2  using namespace std;
3  int main()
4  {
5      double m = 1, n = 0;
6      try {
7          cout << "before dividing." << endl;
8          if (n == 0)
9              throw - 1; //抛出int型异常
10         else if (m == 0)
11             throw - 1.0; //抛出 double 型异常
12         else
13             cout << m / n << endl;
14         cout << "after dividing." << endl;
15     }
16     catch (double d) {
17         cout << "catch (double)" << d << endl;
18     }
19     catch (...) {
20         cout << "catch (...)" << endl;
21     }
22     cout << "finished" << endl;
23     return 0;
24 }
25 //运行结果
26 //before dividing.
27 //catch (...)
28 //finished
29
30
31

```

代码中，对两个数进行除法计算，其中除数为0。可以看到以上三个关键字，程序的执行流程是先执行try包裹的语句块，如果执行过程中没有异常发生，则不会进入任何catch包裹的语句块，如果发生异常，则使用throw进行异常抛出，再由catch进行捕获，throw可以抛出各种数据类型的信息，代码中使用的是数字，也可以自定义异常class。**catch根据throw抛出的数据类型进行精确捕获（不会出现类型转换），如果匹配不到就直接报错，可以使用catch(...)的方式捕获任何异常（不推荐）。**当然，如果catch了异常，当前函数如果不进行处理，或者已经处理了想通知上一层的调用者，可以在catch里面再throw异常。

### (2) 函数的异常声明列表

有时候，程序员在定义函数的时候知道函数可能发生的异常，可以在函数声明和定义时，指出所能抛出异常的列表，写法如下：

```

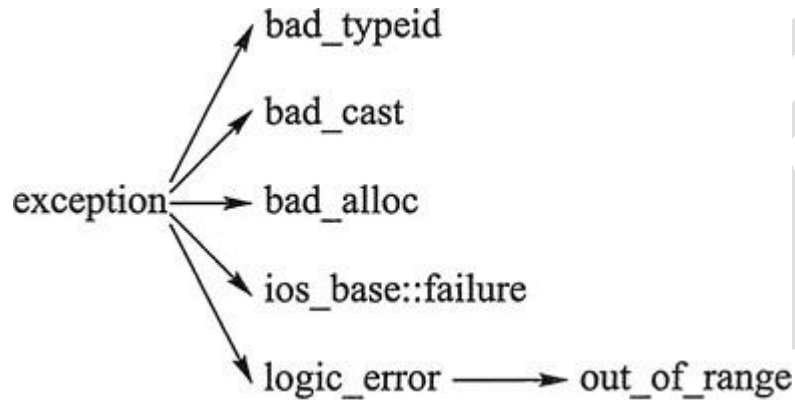
1  int fun() throw(int,double,A,B,C){...};
2
3
4

```

这种写法表明函数可能会抛出int,double型或者A、B、C三种类型的异常，如果throw中为空，表明不会抛出任何异常，如果没有throw则可能抛出任何异常

### (3) C++标准异常类 exception

C++ 标准库中有一些类代表异常，这些类都是从 exception 类派生而来的，如下图所示



- bad\_typeid: 使用typeid运算符，如果其操作数是一个多态类的指针，而该指针的值为 NULL，则会抛出此异常，例如：

```

1  #include <iostream>
2  #include <typeinfo>
3  using namespace std;
4
5  class A{
6  public:
7  virtual ~A();
8  };
9
10 using namespace std;
11 int main() {
12     A* a = NULL;
13     try {
14         cout << typeid(*a).name() << endl; // Error condition 15
15     }
16     catch (bad_typeid){
17         cout << "Object is NULL" << endl; 18
18     }
19     return 0;
20 }
21 //运行结果: bject is NULL
22
23
24
25

```

- bad\_cast: 在用 dynamic\_cast 进行从多态基类对象（或引用）到派生类的引用的强制类型转换时，如果转换是不安全的，则会抛出此异常
- bad\_alloc: 在用 new 运算符进行动态内存分配时，如果没有足够的内存，则会引发此异常

- out\_of\_range:用 vector 或 string的at 成员函数根据下标访问元素时, 如果下标越界, 则会抛出此异常

《C++异常处理 (try catch throw) 完全攻略》: <http://c.biancheng.net/view/422.html>

## 29、static的用法和作用?

1.先来介绍它的第一条也是最重要的一条:隐藏。(static函数, static变量均可) 当

同时编译多个文件时, 所有未加static前缀的全局变量和函数都具有全局可见性。

2.static的第二个作用是保持变量内容的持久。(static变量中的记忆功能和全局生存期) 存储在静态数据区的变量会在程序刚开始运行时就完成初始化, 也是唯一的一次初始化。共有两种变量存储在静态存储区: 全局变量和static变量, 只不过和全局变量比起来, static可以控制变量的可见范围, 说到底static还是用来隐藏的。

3.static的第三个作用是默认初始化为0 (static变量)

其实全局变量也具备这一属性, 因为全局变量也存储在静态数据区。在静态数据区, 内存中所有的字节默认值都是0x00, 某些时候这一特点可以减少程序员的工作量。

4.static的第四个作用: C++中的类成员声明static

1) 函数体内static变量的作用范围为该函数体, 不同于auto变量, 该变量的内存只被分配一次, 因此其值在下次调用时仍维持上次的值;

2) 在模块内的static全局变量可以被模块内所用函数访问, 但不能被模块外其它函数访问;

3) 在模块内的static函数只可被这一模块内的其它函数调用, 这个函数的使用范围被限制在声明它的模块内;

4) 在类中的static成员变量属于整个类所拥有, 对类的所有对象只有一份拷贝;

5) 在类中的static成员函数属于整个类所拥有, 这个函数不接收this指针, 因而只能访问类的static成员变量。

类内:

6) static类对象必须要在类外进行初始化, static修饰的变量先于对象存在, 所以static修饰的变量要在类外初始化;

7) 由于static修饰的类成员属于类, 不属于对象, 因此static类成员函数是没有this指针的, this指针是指向本对象的指针。正因为没有this指针, 所以static类成员函数不能访问非static的类成员, 只能访问static修饰的类成员;

8) static成员函数不能被virtual修饰, static成员不属于任何对象或实例, 所以加上virtual没有任何实际意义; 静态成员函数没有this指针, 虚函数的实现是为每一个对象分配一个vpPtr指针, 而vpPtr是通过this指针调用的, 所以不能为virtual; 虚函数的调用关系, this->vpPtr->ctable->virtual function

## 30、静态变量什么时候初始化

1) 初始化只有一次, 但是可以多次赋值, 在主程序之前, 编译器已经为其分配好了内存。

2) 静态局部变量和全局变量一样, 数据都存放在全局区域, 所以在主程序之前, 编译器已经为其分配好了内存, 但在C和C++中静态局部变量的初始化节点又有点不太一样。在C中, 初始化发生在代码执行之前, 编译阶段分配好内存之后, 就会进行初始化, 所以我们看到在C语言中无法使用变量对静态局部变量进行初始化, 在程序运行结束, 变量所处的全局内存会被全部回收。

3) 而在C++中，初始化时在执行相关代码时才会进行初始化，主要是由于C++引入对象后，要进行初始化必须执行相应构造函数和析构函数，在构造函数或析构函数中经常会需要进行某些程序中需要进行的特定操作，并非简单地分配内存。所以C++标准定为全局或静态对象是有首次用到时才会进行构造，并通过atexit()来管理。在程序结束，按照构造顺序反方向进行逐个析构。所以在C++中是可以使用变量对静态局部变量进行初始化的。

### 31、值传递、指针传递、引用传递的区别和效率

- 1) 值传递：有一个形参向函数所属的栈拷贝数据的过程，如果值传递的对象是类对象 或是大的结构体对象，将耗费一定的时间和空间。（传值）
- 2) 指针传递：同样有一个形参向函数所属的栈拷贝数据的过程，但拷贝的数据是一个固定为4字节的地址。（传值，传递的是地址值）
- 3) 引用传递：同样有上述的数据拷贝过程，但其是针对地址的，相当于为该数据所在的地址起了一个别名。（传地址）
- 4) 效率上讲，指针传递和引用传递比值传递效率高。一般主张使用引用传递，代码逻辑上更加紧凑、清晰。

### 32、类成员初始化方式？构造函数的执行顺序？为什么用成员初始化列表会快一些？

1) 赋值初始化，通过在函数体内进行赋值初始化；列表初始化，在冒号后使用初始化列表进行初始化。

这两种方式的主要区别在于：

对于在函数体中初始化,是在所有的数据成员被分配内存空间后才进行的。

列表初始化是给数据成员分配内存空间时就进行初始化,就是说分配一个数据成员只要冒号后有此数据成员的赋值表达式(此表达式必须是括号赋值表达式),那么分配了内存空间后在进入函数体之前给数据成员赋值,就是说初始化这个数据成员此时函数体还未执行。

2) 一个派生类构造函数的执行顺序如下：

- ① 虚拟基类的构造函数（多个虚拟基类则按照继承的顺序执行构造函数）。
- ② 基类的构造函数（多个普通基类也按照继承的顺序执行构造函数）。
- ③ 类类型的成员对象的构造函数（按照初始化顺序）
- ④ 派生类自己的构造函数。

3) 方法一是在构造函数当中做赋值的操作，而方法二是做纯粹的初始化操作。我们都知道，C++的赋值操作是会产生临时对象的。临时对象的出现会降低程序的效率。

### 33、什么是内存泄露，如何检测与避免

内存泄露

一般我们常说的内存泄漏是指**堆内存的泄漏**。堆内存是指程序从堆中分配的，大小任意的(内存块的大小可以在程序运行期决定)内存块，使用完后必须显式释放的内存。应用程序般使用malloc、realloc、new等函数从堆中分配到块内存，使用完后，程序必须负责相应的调用free或delete释放该内存块，否则，这块内存就不能被再次使用，我们就说这块内存泄漏了

## 避免内存泄露的几种方式

- 计数法：使用new或者malloc时，让该数+1，delete或free时，该数-1，程序执行完打印这个计数，如果不为0则表示存在内存泄露
- 一定要将基类的析构函数声明为**虚函数**
- 对象数组的释放一定要用**delete []**
- 有new就有delete，有malloc就有free，保证它们一定成对出现

## 检测工具

- Linux下可以使用**Valgrind工具**
- Windows下可以使用**CRT库**

## 34、C++中类的数据成员和成员函数内存分布情况

C++类是由结构体发展得来的，所以他们的成员变量(C语言的结构体只有成员变量)的内存分配机制是一样的。下面我们以类来说明问题，如果类的问题通了，结构体也也就没问题啦。类分为成员变量和成员函数，我们先来讨论成员变量。

一个类对象的地址就是类所包含的这片内存空间的首地址，这个首地址也就对应具体某一个成员变量的地址。（在定义类对象的同时这些成员变量也就被定义了），举个例子：

```
#include <iostream>
using namespace std;
class Person
{
public:
    Person()
    {
        this->age = 23;
    }
    void printAge()
    {
        cout << this->age << endl;
    }
    ~Person() {}
public:
    int age;
};
int main()
{
    Person p;
    cout << "对象地址: " << &p << endl;
    cout << "age地址: " << &(p.age) << endl;
    cout << "对象大小: " << sizeof(p) << endl;
    cout << "age大小: " << sizeof(p.age) << endl;
    return 0;
}
//输出结果
//对象地址: 0x7fffec0f15a8
//age地址: 0x7fffec0f15a8
//对象大小: 4
//age大小: 4
```

从代码运行结果来看，对象的大小和对象中数据成员的大小是一致的，也就是说，成员函数不占用对象的内存。这是因为所有的函数都是存放在代码区的，不管是全局函数，还是成员函数。要是成员函数占用类的对象空间，那么将是多么可怕的事情：定义一次类对象就有成员函数占用一段空间。我们再来补充一下静态成员函数的存放问题：**静态成员函数与一般成员函数的唯一区别就是没有this指针**，因此不能访问非静态数据成员，就像我前面提到的，**所有函数都存放在代码区，静态函数也不例外。所有人一看到 static 这个单词就主观的认为是存放在全局数据区，那是不对的。**

### 35、(超重要)构造函数为什么不能为虚函数？析构函数为什么要虚函数？

1、**从存储空间角度**，虚函数相应一个指向vtable虚函数表的指针，这大家都知道，但是这个指向vtable的指针事实上是存储在对象的内存空间的。



问题出来了，假设构造函数是虚的，就须要通过 vtable来调用，但是对象还没有实例化，也就是内存空间还没有，怎么找vtable呢？所以构造函数不能是虚函数。

**2、从使用角度**，虚函数主要用于在信息不全的情况下，能使重载的函数得到相应的调用。构造函数本身就是初始化实例，那使用虚函数也没有实际意义呀。

所以构造函数没有必要是虚函数。虚函数的作用在于通过父类的指针或者引用来调用它的时候可以变成调用子类的那个成员函数。而构造函数是在创建对象时自己主动调用的，不可能通过父类的指针或者引用来调用，因此也就规定构造函数不能是虚函数。

**3、构造函数不须要是虚函数，也不同意是虚函数**，由于创建一个对象时我们总是要明白指定对象的类型，虽然我们可能通过实验室的基类的指针或引用去访问它但析构却不一定，我们往往通过基类的指针来销毁对象。这时候假设析构函数不是虚函数，就不能正确识别对象类型从而不能正确调用析构函数。

**4、从实现上看**，vbt在构造函数调用后才建立，因而构造函数不可能成为虚函数从实际含义上看，在调用构造函数时还不能确定对象的真实类型（由于子类会调父类的构造函数）；并且构造函数的作用是提供初始化，在对象生命期仅仅运行一次，不是对象的动态行为，也没有必要成为虚函数。

**5、当一个构造函数被调用时，它做的首要的事情之中的一个是初始化它的VPTR。**

因此，它仅仅能知道它是“当前”类的，而全然忽视这个对象后面是否还有继承者。当编译器为这个构造函数产生代码时，它是为这个类的构造函数产生代码——既不是为基类，也不是为它的派生类（由于类不知道谁继承它）。所以它使用的VPTR必须是对于这个类的VTABLE。

并且，仅仅要它是最后的构造函数调用，那么在这个对象的生命期内，VPTR将保持被初始化为指向这个VTABLE，但假设接着另一个更晚派生的构造函数被调用，这个构造函数又将设置VPTR指向它的VTABLE，等直到最后的构造函数结束。

VPTR的状态是由被最后调用的构造函数确定的。这就是为什么构造函数调用是从基类到更加派生类顺序的还有一个理由。可是，当这一系列构造函数调用正发生时，每一个构造函数都已经设置VPTR指向它自己的VTABLE。假设函数调用使用虚机制，它将仅仅产生通过它自己的VTABLE的调用，而不是最后的VTABLE（全部构造函数被调用后才会有最后的VTABLE）。

因为构造函数本来就是为了明确初始化对象成员才产生的，然而virtual function主要是为了再不完全了解细节的情况下也能正确处理对象。另外，virtual函数是在不同类型的对象产生不同的动作，现在对象还没有产生，如何使用virtual函数来完成你想完成的动作。

直接的讲，C++中基类采用virtual虚析构函数是**为了防止内存泄漏**。

具体地说，如果派生类中申请了内存空间，并在其析构函数中对这些内存空间进行释放。假设基类中采用的是非虚析构函数，当删除基类指针指向的派生类对象时就不会触发动态绑定，因而只会调用基类的析构函数，而不会调用派生类的析构函数。那么在这种情况下，派生类中申请的空间就得不到释放从而产生内存泄漏。

所以，为了防止这种情况的发生，C++中基类的析构函数应采用virtual虚析构函数。

## **36、析构函数的作用，如何起作用？**

1) 构造函数只是起初始化值的作用，但实例化一个对象的时候，可以通过实例去传递参数，从主函数传递到其他的函数里面，这样就使其他的函数里面有值了。

规则，只要你一实例化对象，系统自动回调用一个构造函数就是你不写，编译器也自动调用一次。

2) 析构函数与构造函数的作用相反，用于撤销对象的一些特殊任务处理，可以是释放对象分配的内存空间；特点：析构函数与构造函数同名，但该函数前面加~。

析构函数没有参数，也没有返回值，而且不能重载，在一个类中只能有一个析构函数。当撤销对象时，编译器也会自动调用析构函数。



每一个类必须有一个析构函数，用户可以自定义析构函数，也可以是编译器自动生成默认的析构函数。一般析构函数定义为类的公有成员。

### 37、构造函数和析构函数可以调用虚函数吗，为什么

- 1) 在C++中，提倡不在构造函数和析构函数中调用虚函数；
- 2) 构造函数和析构函数调用虚函数时都不使用动态联编，如果在构造函数或析构函数中调用虚函数，则运行的是为构造函数或析构函数自身类型定义的版本；
- 3) 因为父类对象会在子类之前进行构造，此时子类部分的数据成员还未初始化，因此调用子类的虚函数时不安全的，故而C++不会进行动态联编；
- 4) 析构函数是用来销毁一个对象的，在销毁一个对象时，先调用子类的析构函数，然后再调用基类的析构函数。所以在调用基类的析构函数时，派生类对象的数据成员已经销毁，这个时候再调用子类的虚函数没有任何意义。

### 38、构造函数、析构函数的执行顺序？构造函数和拷贝构造的内部都干了啥？

#### 1) 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，而不是它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，而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。

#### 2) 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。

### 39、虚析构函数的作用，父类的析构函数是否要设置为虚函数？

- 1) C++中基类采用virtual虚析构函数是为了防止内存泄漏。

具体地说，如果派生类中申请了内存空间，并在其析构函数中对这些内存空间进行释放。

假设基类中采用的是非虚析构函数，当删除基类指针指向的派生类对象时就不会触发动态绑定，因而只会调用基类的析构函数，而不会调用派生类的析构函数。

那么在这种情况下，派生类中申请的空间就得不到释放从而产生内存泄漏。

所以，为了防止这种情况的发生，C++中基类的析构函数应采用virtual虚析构函数。

- 2) 纯虚析构函数一定得定义，因为每一个派生类析构函数会被编译器加以扩张，以静态调用的方式调用其每一个虚基类以及上一层基类的析构函数。

因此，缺乏任何一个基类析构函数的定义，就会导致链接失败，最好不要把虚析构函数定义为纯虚析构函数。

## 40、构造函数析构函数可否抛出异常

1) C++只会析构已经完成的对象，对象只有在其构造函数执行完毕才算是完全构造妥当。在构造函数中发生异常，控制权转出构造函数之外。

因此，在对象b的构造函数中发生异常，对象b的析构函数不会被调用。因此会造成内存泄漏。

2) 用auto\_ptr对象来取代指针类成员，便对构造函数做了强化，免除了抛出异常时发生资源泄漏的危机，不再需要在析构函数中手动释放资源；

3) 如果控制权基于异常的因素离开析构函数，而此时正有另一个异常处于作用状态，C++会调用terminate函数让程序结束；

4) 如果异常从析构函数抛出，而且没有在当地进行捕捉，那个析构函数便是执行不全的。如果析构函数执行不全，就是没有完成他应该执行的每一件事情。

## 41、智能指针的原理、常用的智能指针及实现

### 原理

智能指针是一个类，用来存储指向动态分配对象的指针，负责自动释放动态分配的对象，防止堆内存泄漏。动态分配的资源，交给一个类对象去管理，当类对象声明周期结束时，自动调用析构函数释放资源

### 常用的智能指针

#### (1) shared\_ptr

实现原理：采用引用计数器的方法，允许多个智能指针指向同一个对象，每当多一个指针指向该对象时，指向该对象的所有智能指针内部的引用计数加1，每当减少一个智能指针指向对象时，引用计数会减1，当计数为0的时候会自动的释放动态分配的资源。

- 智能指针将一个计数器与类指向的对象相关联，引用计数器跟踪共有多少个类对象共享同一指针
- 每次创建类的新对象时，初始化指针并将引用计数置为1
- 当对象作为另一对象的副本而创建时，拷贝构造函数拷贝指针并增加与之相应的引用计数
- 对一个对象进行赋值时，赋值操作符减少左操作数所指对象的引用计数（如果引用计数为减至0，则删除对象），并增加右操作数所指对象的引用计数
- 调用析构函数时，构造函数减少引用计数（如果引用计数减至0，则删除基础对象）

#### (2) unique\_ptr

unique\_ptr采用的是独享所有权语义，一个非空的unique\_ptr总是拥有它所指向的资源。转移一个unique\_ptr将会把所有权全部从源指针转移给目标指针，源指针被置空；所以unique\_ptr不支持普通的拷贝和赋值操作，不能用在STL标准容器中；局部变量的返回值除外（因为编译器知道要返回的对象将要被销毁）；如果你拷贝一个unique\_ptr，那么拷贝结束后，这两个unique\_ptr都会指向相同的资源，造成在结束时对同一内存指针多次释放而导致程序崩溃。

#### (3) weak\_ptr

weak\_ptr：弱引用。引用计数有一个问题就是互相引用形成环（环形引用），这样两个指针指向的内存都无法释放。需要使用weak\_ptr打破环形引用。weak\_ptr是一个弱引用，它是为了配合shared\_ptr而引入的一种智能指针，它指向一个由shared\_ptr管理的对象而不影响所指对象的生命周期，也就是说，它只引用，不计数。如果一块内存被shared\_ptr和weak\_ptr同时引用，当所有shared\_ptr析构了之后，不管还有没有weak\_ptr引用该内存，内存也会被释放。所以weak\_ptr不保证它指向的内存一定是有效的，在使用之前使用函数lock()检查weak\_ptr是否为空指针。

#### (4) auto\_ptr

主要是为了解决“有异常抛出时发生内存泄漏”的问题。因为发生异常而无法正常释放内存。

auto\_ptr有拷贝语义，拷贝后源对象变得无效，这可能引发很严重的问题；而unique\_ptr则无拷贝语义，但提供了移动语义，这样的错误不再可能发生，因为很明显必须使用std::move()进行转移。

auto\_ptr不支持拷贝和赋值操作，不能用在STL标准容器中。STL容器中的元素经常要支持拷贝、赋值操作，在这过程中auto\_ptr会传递所有权，所以不能在STL中使用。

**智能指针shared\_ptr代码实现：**

```

1  template<typename T>
2  class SharedPtr
3  {
4  public:
5      SharedPtr(T* ptr = NULL):_ptr(ptr), _pcount(new int(1))
6      {}
7
8      SharedPtr(const SharedPtr& s):_ptr(s._ptr),
9      _pcount(s._pcount){
10         *(_pcount)++;
11     }
12
13     SharedPtr<T>& operator=(const SharedPtr& s){
14         if (this != &s)
15         {
16             if (--(*this->_pcount) == 0)
17             {
18                 delete this->_ptr;
19                 delete this->_pcount;
20             }
21             _ptr = s._ptr;
22             _pcount = s._pcount;
23             *(_pcount)++;
24         }
25         return *this;
26     }
27     T& operator*()
28     {
29         return *(this->_ptr);
30     }
31     T* operator->()
32     {
33         return this->_ptr;
34     }
35     ~SharedPtr()
36     {
37         --(*this->_pcount);
38         if (this->_pcount == 0)
39         {
40             delete _ptr;
41             _ptr = NULL;
42             delete _pcount;
43             _pcount = NULL;
44         }
45     }
46 private:
47     T* _ptr;
48     int* _pcount; //指向引用计数的指针
49 };
50
51
52
53

```

## 42、C++函数调用的压栈过程

从代码入手，解释这个过程：

```
#include <iostream>
using namespace std;
int f(int n)
{
    cout << n << endl;
    return n;
}
void func(int param1, int param2)
{
    int var1 = param1;
    int var2 = param2;
    printf("var1=%d, var2=%d", f(var1), f(var2)); //如果将printf换为cout进行输出，输出结果则刚好相反
}
int main(int argc, char* argv[])
{
    func(1, 2);
    return 0;
}
//输出结果
//2
//1
//var1=1, var2=2
```

当函数从入口函数main函数开始执行时，编译器会将我们操作系统的运行状态，main函数的返回地址、main的参数、main函数中的变量、进行依次压栈；

当main函数开始调用func()函数时，编译器此时会将main函数的运行状态进行压栈，再将func()函数的返回地址、func()函数的参数从右到左、func()定义变量依次压栈；

当func()调用f()的时候，编译器此时会将func()函数的运行状态进行压栈，再将其返回地址、f()函数的参数从右到左、f()定义变量依次压栈

从代码的输出结果可以看出，函数f(var1)、f(var2)依次入栈，而后先执行f(var2)，再执行f(var1)，最后打印整个字符串，将栈中的变量依次弹出，最后主函数返回。

## 43、几个this指针的易混问题

### A. this指针是什么时候创建的？

this在成员函数的开始执行前构造，在成员的执行结束后清除。

但是如果class或者struct里面没有方法的话，它们是没有构造函数的，只能当做C的struct使用。采用TYPE xx的方式定义的话，在栈里分配内存，这时候this指针的值就是这块内存的地址。采用new的方式创建对象的话，在堆里分配内存，new操作符通过eax（累加寄存器）返回分配的地址，然后设置给指针变量。之后去调用构造函数（如果有构造函数的话），这时将这个内存块的地址传给ecx，之后构造函数里面怎么处理请看上面的回答

### B. this指针存放在何处？堆、栈、全局变量，还是其他？

this指针会因编译器不同而有不同的放置位置。可能是栈，也可能是寄存器，甚至全局变量。在汇编级别里面，一个值只会以3种形式出现：立即数、寄存器值和内存变量值。不是存放在寄存器就是存放在内存中，它们并不是和高级语言变量对应的。

### C. this指针是如何传递类中的函数的？绑定？还是在函数参数的首参数就是this指针？那么，this指针又是如何找到“类实例后函数的”？

大多数编译器通过ecx（寄数寄存器）寄存器传递this指针。事实上，这也是一个潜规则。一般来说，不同编译器都会遵从一致的传参规则，否则不同编译器产生的obj就无法匹配了。

在call之前，编译器会把对应的对象地址放到eax中。this是通过函数参数的首参来传递的。this指针在调用之前生成，至于“类实例后函数”，没有这个说法。类在实例化时，只分配类中的变量空间，并没有为函数分配空间。自从类的函数定义完成后，它就在那儿，不会跑的

### D. this指针是如何访问类中的变量的？

如果不是类，而是结构体的话，那么，如何通过结构指针来访问结构中的变量呢？如果你明白这一点的话，就很容易理解这个问题了。

在C++中，类和结构是只有一个区别的：类的成员默认是private，而结构是public。

this是类的指针，如果换成结构体，那this就是结构的指针了。

### E. 我们只有获得一个对象后，才能通过对象使用this指针。如果我们知道一个对象this指针的位置，可以直接使用吗？

**this指针只有在成员函数中才有定义。**因此，你获得一个对象后，也不能通过对象使用this指针。所以，我们无法知道一个对象的this指针的位置（只有在成员函数里才有this指针的位置）。当然，在成员函数里，你是可以知道this指针的位置的（可以通过&this获得），也可以直接使用它。

#### F. 每个类编译后，是否创建一个类中函数表保存函数指针，以便用来调用函数？

普通的类函数（不论是成员函数，还是静态函数）都不会创建一个函数表来保存函数指针。只有虚函数才会被放到函数表中。但是，即使是虚函数，如果编译期就能明确知道调用的是哪个函数，编译器就不会通过函数表中的指针来间接调用，而是会直接调用该函数。正是由于this指针的存在，用来指向不同的对象，从而确保不同对象之间调用相同的函数可以互不干扰

《C++中this指针的用法详解》<http://blog.chinaunix.net/uid-21411227-id-1826942.html>

## 44、智能指针的循环引用

循环引用是指使用多个智能指针share\_ptr时，出现了指针之间相互指向，从而形成环的情况，有点类似于死锁的情况，这种情况下，智能指针往往不能正常调用对象的析构函数，从而造成内存泄漏。举个例子：`#include <iostream>`

```
using namespace std;
template <typename T>
class Node
{
public:
    Node(const T& value)
        : _pPre(NULL)
        , _pNext(NULL)
        , _value(value)
    {
        cout << "Node()" << endl;
    }
    ~Node()
    {
        cout << "~Node()" << endl;
        cout << "this:" << this << endl;
    }
    shared_ptr<Node<T>> _pPre;
    shared_ptr<Node<T>> _pNext;
    T _value;
};

void Funtest()
{
    shared_ptr<Node<int>> sp1(new Node<int>(1));
    shared_ptr<Node<int>> sp2(new Node<int>(2));
    cout << "sp1.use_count:" << sp1.use_count() << endl;
    cout << "sp2.use_count:" << sp2.use_count() << endl;
    sp1->_pNext = sp2; //sp1的引用+1
    sp2->_pPre = sp1; //sp2的引用+1
    cout << "sp1.use_count:" << sp1.use_count() << endl;
    cout << "sp2.use_count:" << sp2.use_count() << endl;
}

int main()
{
    Funtest();
    system("pause");
    return 0;
}

//输出结果
//Node()
//Node()
//sp1.use_count:1
```

```
//sp2.use_count:1  
//sp1.use_count:2  
  
//sp2.use_count:2
```

程序员成长路线



从上面shared\_ptr的实现中我们知道了只有当引用计数减减之后等于0，析构时才会释放对象，而上述情况造成了一个僵局，那就是析构对象时先析构sp2,可是由于sp2的空间sp1还在使用中，所以sp2.use\_count减减之后为1，不释放，sp1也是相同的道理，由于sp1的空间sp2还在使用中，所以sp1.use\_count减减之后为1，也不释放。sp1等着sp2先释放，sp2等着sp1先释放,二者互不相让，导致最终都没能释放，内存泄漏。

在实际编程过程中，应该尽量避免出现智能指针之前相互指向的情况，如果不可避免，可以使用使用弱指针——weak\_ptr，它不增加引用计数，只要出了作用域就会自动析构。

《C++ 智能指针 (及循环引用问题) 》: [https://blog.csdn.net/m0\\_37968340/article/details/7\\_6737395](https://blog.csdn.net/m0_37968340/article/details/7_6737395)

## 45、C++中的指针参数传递和引用参数传递有什么区别？底层原理你知道吗？

1) 指针参数传递本质上是值传递，它所传递的是一个地址值。

值传递过程中，被调函数的形式参数作为被调函数的局部变量处理，会在栈中开辟内存空间以存放由主调函数传递进来的实参值，从而形成了实参的一个副本（替身）。

值传递的特点是，被调函数对形式参数的任何操作都是作为局部变量进行的，不会影响主调函数的实参变量的值（形参指针变了，实参指针不会变）。

2) 引用参数传递过程中，被调函数的形式参数也作为局部变量在栈中开辟了内存空间，但是这时存放的是由主调函数放进来的实参变量的地址。

被调函数对形参（本体）的任何操作都被处理成间接寻址，即通过栈中存放的地址访问主调函数中的实参变量（根据别名找到主调函数中的本体）。

因此，被调函数对形参的任何操作都会影响主调函数中的实参变量。

3) 引用传递和指针传递是不同的，虽然他们都是在被调函数栈空间上的一个局部变量，但是任何对于引用参数的处理都会通过一个间接寻址的方式操作到主调函数中的相关变量。

而对于指针传递的参数，如果改变被调函数中的指针地址，它将应用不到主调函数的相关变量。如果想通过指针参数传递来改变主调函数中的相关变量（地址），那就得使用指向指针的指针或者指针引用。

4) 从编译的角度来讲，程序在编译时分别将指针和引用添加到符号表上，符号表中记录的是变量名及变量所对应地址。

指针变量在符号表上对应的地址值为指针变量的地址值，而引用在符号表上对应的地址值为引用对象的地址值（与实参名字不同，地址相同）。

符号表生成之后就不会再改，因此指针可以改变其指向的对象（指针变量中的值可以改），而引用对象 则不能修改。

## 2、操作系统

### 1、进程、线程和协程的区别和联系

	进程	线程	协程
定义	资源分配和拥有的基本单位	程序执行的基本单位	用户态的轻量级线程，线程内部调度的基本单位

切换情况	进程CPU环境(栈、寄存器、页表和文件句柄等)的保存以及新调度的进程CPU环境的设置	保存和设置程序计数器、少量寄存器和栈的内容	先将寄存器上下文和栈保存，等切换回来的时候再进行恢复
切换者	操作系统	操作系统	用户
切换过程	用户态->内核态->用户态	用户态->内核态->用户态	用户态(没有陷入内核)
调用栈	内核栈	内核栈	用户栈
拥有资源	CPU资源、内存资源、文件资源和句柄等	程序计数器、寄存器、栈和状态字	拥有自己的寄存器上下文和栈
并发性	不同进程之间切换实现并发，各自占有CPU实现并行	一个进程内部的多个线程并发执行	同一时间只能执行一个协程，而其他协程处于休眠状态，适合对任务进行分时处理
系统开销	切换虚拟地址空间，切换内核栈和硬件上下文，CPU高速缓存失效、页表切换，开销很大	切换时只需保存和设置少量寄存器内容，因此开销很小	直接操作栈则基本没有内核切换的开销，可以不加锁的访问全局变量，所以上下文的切换非常快
通信方面	进程间通信需要借助操作系统	线程间可以直接读写进程数据段(如全局变量)来进行通信	共享内存、消息队列

1、进程是资源调度的基本单位，运行一个可执行程序会创建一个或多个进程，进程就是运行起来的可执行程序

2、线程是程序执行的基本单位，是轻量级的进程。每个进程中都有唯一的主线程，且只能有一个，主线程和进程是相互依存的关系，主线程结束进程也会结束。多提一句：协程是用户态的轻量级线程，线程内部调度的基本单位

## 2、线程与进程的比较

1、线程启动速度快，轻量级

2、线程的系统开销小

3、线程使用有一定难度，需要处理数据一致性问题

4、同一线程共享的有堆、全局变量、静态变量、指针，引用、文件等，而独自占有

### 3、一个进程可以创建多少线程，和什么有关？

理论上，一个进程可用虚拟空间是2G，默认情况下，线程的栈的大小是1MB，所以理论上最多只能创建2048个线程。如果要创建多于2048的话，必须修改编译器的设置。

因此，一个进程可以创建的线程数由可用虚拟空间和线程的栈的大小共同决定，只要虚拟空间足够，那么新线程的建立就会成功。如果需要创建超过2K以上的线程，减小你线程栈的大小就可以实现了，虽然在一般情况下，你不需要那么多的线程。过多的线程将会导致大量的时间浪费在线程切换上，给程序运行效率带来负面影响。

《一个进程到底能创建多少线程》：[https://www.cnblogs.com/Leo\\_wl/p/5969621.html](https://www.cnblogs.com/Leo_wl/p/5969621.html)

### 4、外中断和异常有什么区别？

外中断是指由 CPU 执行指令以外的事件引起，如 I/O 完成中断，表示设备输入/输出处理已经完成，处理器能够发送下一个输入/输出请求。此外还有时钟中断、控制台中断等。

而异常时由 CPU 执行指令的内部事件引起，如非法操作码、地址越界、算术溢出等。

### 5、进程线程模型你知道多少？

对于进程和线程的理解和把握可以说基本奠定了对系统的认知和把控能力。其核心意义绝不仅仅是“线程是调度的基本单位，进程是资源分配的基本单位”这么简单。

#### 多线程

我们这里讨论的是用户态的多线程模型，同一个进程内部有多个线程，所有的线程共享同一个进程的内存空间，进程中定义的全局变量会被所有的线程共享，比如有全局变量 `int i = 10`，这一进程中所有并发运行的线程都可以读取和修改这个 `i` 的值，而多个线程被 CPU 调度的顺序又是不可控的，所以对临界资源的访问尤其需要注意安全。

我们必须知道，**做一次简单的  $i=i+1$  在计算机中并不是原子操作，涉及内存取数，计算和写入内存几个环节**，而线程的切换有可能发生在上述任何一个环节中间，所以不同的操作顺序很有可能带来意想不到的结果。

但是，虽然线程在安全性方面会引入许多新挑战，但是线程带来的好处也是有目共睹的。首先，原先顺序执行的程序（暂时不考虑多进程）可以被拆分成几个独立的逻辑流，这些逻辑流可以独立完成一些任务（最好这些任务是不相关的）。

比如 QQ 可以一个线程处理聊天一个线程处理上传文件，两个线程互不干涉，在用户看来是同步在执行两个任务，试想如果线性完成这个任务的话，在数据传输完成之前用户聊天被一直阻塞会是多么尴尬的情况。

对于线程，我认为弄清以下两点非常重要：

- 线程之间有无先后访问顺序（线程依赖关系）多
- 个线程共享访问同一变量（同步互斥问题）

另外，我们通常只会去说同一进程的多个线程共享进程的资源，但是每个线程特有的部分却很少提及，除了标识线程的 `tid`，每个线程还有自己独立的栈空间，线程彼此之间是无法访问其他线程栈上内容的。

而作为处理机调度的最小单位，线程调度只需要保存线程栈、寄存器数据和PC即可，相比进程切换开销要小很多。

线程相关接口不少，主要需要了解各个参数意义和返回值意义。

## 1. 线程创建和结束

### ◦ 背景知识：

在一个文件内的多个函数通常都是按照main函数中出现的顺序来执行，但是在分时系统下，我们可以让每个函数都作为一个逻辑流并发执行，最简单的方式就是采用多线程策略。在main函数中调用多线程接口创建线程，每个线程对应特定的函数（操作），这样就可以不按main函数中各个函数出现的顺序来执行，避免了忙等的情况。线程基本操作的接口如下。

### ◦ 相关接口：

- 创建线程：int pthread\_create(pthread\_t \*pthread, const pthread\_attr\_t \*attr, void (start\_routine)(void \*), void \*arg);

创建一个新线程，pthread和start\_routine不可或缺，分别用于标识线程和执行体入口，其他可以填NULL。

- pthread：用来返回线程的tid，\*pthread值即为tid，类型pthread\_t == unsigned long
- int 。 attr：指向线程属性结构体的指针，用于改变所创线程的属性，填NULL使用默认值。
- start\_routine：线程执行函数的首地址，传入函数指针。
- arg：通过地址传递来传递函数参数，这里是无符号类型指针，可以传任意类型变量的地址，在被传入函数中先强制类型转换成所需类型即可。
- 获得线程ID：pthread\_t pthread\_self();  
调用时，会打印线程ID。
- 等待线程结束：int pthread\_join(pthread\_t tid, void\*\* retval);  
主线程调用，等待子线程退出并回收其资源，类似于进程中wait/waitpid回收僵尸进程，调用pthread\_join的线程会被阻塞。
  - tid：创建线程时通过指针得到tid值。
  - retval：指向返回值的指针。
- 结束线程：pthread\_exit(void \*retval);  
子线程执行，用来结束当前线程并通过retval传递返回值，该返回值可通过pthread\_join获得。
  - retval：同上。
- 分离线程：int pthread\_detach(pthread\_t tid);  
主线程、子线程均可调用。主线程中pthread\_detach(tid)，子线程中pthread\_detach(pthread\_self())，调用后和主线程分离，子线程结束时自己立即回收资源。
  - tid：同上。

## 2. 线程属性值修改

### ◦ 背景知识：

线程属性对象类型为pthread\_attr\_t，结构体定义如下：

```

1  typedef struct{ int
2      etachstate;          // 线程分离的状态
3      int schedpolicy;     // 线程调度策略
4      struct sched_param schedparam; // 线程的调度参数
5      int inheritsched;    // 线程的继承性
6      int scope;          // 线程的作用域
7      // 以下为线程栈的设置
8      size_t guardsize;    // 线程栈末尾警戒缓冲大小
9      int stackaddr_set;   // 线程的栈设置
10     void *    stackaddr;  // 线程栈的位置
11     size_t stacksize;    // 线程栈大小
12 }pthread_attr_t;
13

```

◦ 相关接口：

对上述结构体中各参数大多有：pthread\_attr\_get()和pthread\_attr\_set()系统调用函数来设置和获取。这里不一一罗列。

## 多进程

每一个进程是资源分配的基本单位。

进程结构由以下几个部分组成：代码段、堆栈段、数据段。代码段是静态的二进制代码，多个程序可以共享。

实际上在父进程创建子进程之后，父、子进程除了pid外，几乎所有的部分几乎一样。

父、子进程共享全部数据，但并不是说他们就是对同一块数据进行操作，子进程在读写数据时会通过写时复制机制将公共的数据重新拷贝一份，之后在拷贝出的数据上进行操作。

如果子进程想要运行自己的代码段，还可以通过调用execv()函数重新加载新的代码段，之后就与父进程独立开了。

我们在shell中执行程序就是通过shell进程先fork()一个子进程再通过execv()重新加载新的代码段的过程。

### 1. 进程创建与结束

◦ 背景知识：

进程有两种创建方式，一种是操作系统创建的一种是父进程创建的。从计算机启动到终端执行程序的过程为：0号进程 -> 1号内核进程 -> 1号用户进程(init进程) -> getty进程 -> shell进程 -> 命令行执行进程。所以我们在命令行中通过 ./program执行可执行文件时，所有创建的进程都是shell进程的子进程，这也就是为什么shell一关闭，在shell中执行的进程都自动被关闭的原因。从shell进程到创建其他子进程需要通过以下接口。

◦ 相关接口：

- 创建进程：pid\_t fork(void);  
返回值：出错返回-1；父进程中返回pid > 0；子进程中pid == 0
- 结束进程：void exit(int status);
  - status是退出状态，保存在全局变量中S?，通常0表示正常退出。
- 获得PID：pid\_t getpid(void);  
返回调用者pid。
- 获得父进程PID：pid\_t getppid(void);  
返回父进程pid。

- 其他补充：

- 正常退出方式：exit()、\_exit()、return（在main中）。

exit()和\_exit()区别：exit()是对\_exit()的封装，都会终止进程并做相关收尾工作，最主要的区别是\_exit()函数关闭全部描述符和清理函数后不会刷新流，但是exit()会在调用\_exit()函数前刷新数据流。

return和\_exit()区别：exit()是函数，但有参数，执行完之后控制权交给系统。return若是在调用函数中，执行完之后控制权交给调用进程，若是在main函数中，控制权交给系统。

- 异常退出方式：abort()、终止信号。

## 2. Linux进程控制

- 进程地址空间（地址空间）

虚拟存储器为每个进程提供了独占系统地址空间的假象。

尽管每个进程地址空间内容不尽相同，但是他们的都有相似的结构。X86 Linux进程的地址空间底部是保留给用户程序的，包括文本、数据、堆、栈等，其中文本区和数据区是通过存储器映射方式将磁盘中可执行文件的相应段映射至虚拟存储器地址空间中。

有一些"敏感"的地址需要注意下，对于32位进程来说，代码段从0x08048000开始。从0xC0000000开始到0xFFFFFFFF是内核地址空间，通常情况下代码运行在用户态（使用0x00000000~0xC0000000的用户地址空间），当发生系统调用、进程切换等操作时CPU控制寄存器设置模式位，进入内核模式，在该状态（超级用户模式）下进程可以访问全部存储器位置和 执行全部指令。

也就说32位进程的地址空间都是4G，但用户态下只能访问低3G的地址空间，若要访问3G ~ 4G的地址空间则只有进入内核态才行。

- 进程控制块（处理机）

进程的调度实际就是内核选择相应的进程控制块，被选择的进程控制块中包含了一个进程基本的信息。

- 上下文切换

内核管理所有进程控制块，而进程控制块记录了进程全部状态信息。每一次进程调度就是一次上下文切换，所谓的上下文本质上就是当前运行状态，主要包括通用寄存器、浮点寄存器、状态寄存器、程序计数器、用户栈和内核数据结构（页表、进程表、文件表）等。

进程执行时刻，内核可以决定抢占当前进程并开始新的进程，这个过程由内核调度器完成，当调度器选择了某个进程时称为该进程被调度，该过程通过上下文切换来改变当前状态。

一次完整的上下文切换通常是进程原先运行于用户态，之后因系统调用或时间片到切换到内核态执行内核指令，完成上下文切换后回到用户态，此时已经切换到进程B。

## 6、Linux下进程间通信方式？

- 管道：

- 无名管道（内存文件）：管道是一种半双工的通信方式，数据只能单向流动，而且只能在具有亲缘关系的进程之间使用。进程的亲缘关系通常是指父子进程关系。

- 有名管道（FIFO文件，借助文件系统）：有名管道也是半双工的通信方式，但是允许在没有亲缘关系的进程之间使用，管道是先进先出的通信方式。

- 共享内存：共享内存就是映射一段能被其他进程所访问的内存，这段共享内存由一个进程创建，但多个进程都可以访问。共享内存是最快的IPC方式，它是针对其他进程间通信方式运行效率低而专门设计的。它往往与信号量，配合使用来实现进程间的同步和通信。

- 消息队列：消息队列是有消息的链表，存放在内核中并由消息队列标识符标识。消息队列克服了信号传递信息少、管道只能承载无格式字节流以及缓冲区大小受限等缺点。

- 套接字：适用于不同机器间进程通信，在本地也可作为两个进程通信的方式。

-

信号：用于通知接收进程某个事件已经发生，比如按下ctrl + C就是信号。

- 信号量：信号量是一个计数器，可以用来控制多个进程对共享资源的访问。它常作为一种锁机制，实现进程、线程的对临界区的同步及互斥访问。

## 7、Linux下同步机制？

- POSIX信号量：可用于进程同步，也可用于线程同步。
- POSIX互斥锁 + 条件变量：只能用于线程同步。

### 1. 线程和进程的区别？

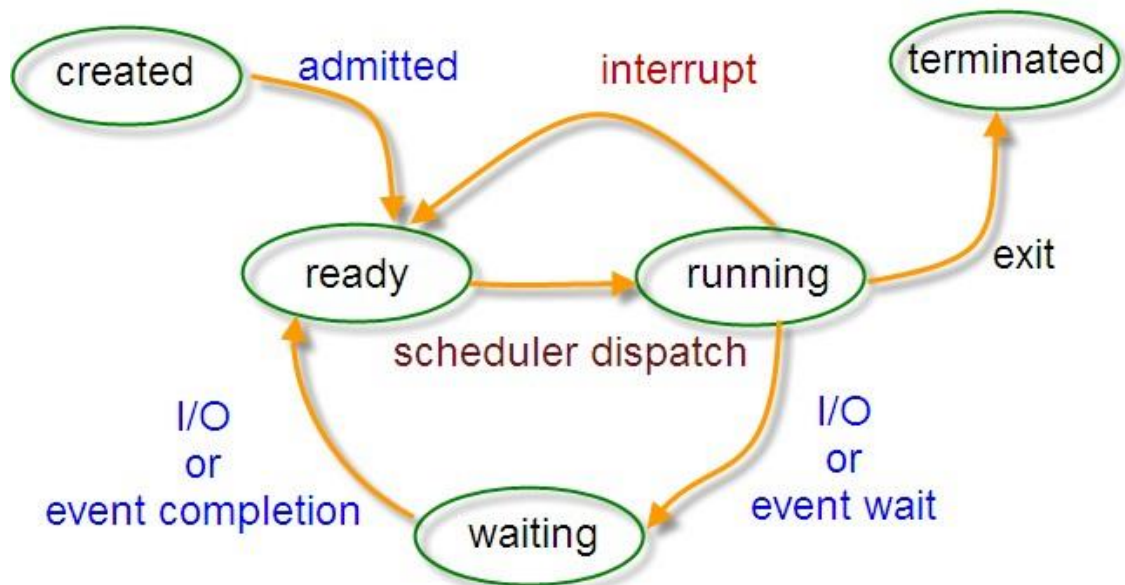
- 调度：线程是调度的基本单位（PC，状态码，通用寄存器，线程栈及栈指针）；进程是拥有资源的基本单位（打开文件，堆，静态区，代码段等）。
- 并发性：一个进程内多个线程可以并发（最好和CPU核数相等）；多个进程可以并发。
- 拥有资源：线程不拥有系统资源，但一个进程的多个线程可以共享隶属进程的资源；进程是拥有资源的独立单位。
- 系统开销：线程创建销毁只需要处理PC值，状态码，通用寄存器值，线程栈及栈指针即可；进程创建和销毁需要重新分配及销毁task\_struct结构。



## 8、进程状态的切换你知道多少？

程序员老蔡

# Process State



- 就绪状态 (ready)：等待被调度
- 运行状态 (running)
- 阻塞状态 (waiting)：等待资源

应该注意以下内容：

- 只有就绪态和运行态可以相互转换，其它的都是单向转换。就绪状态的进程通过调度算法从而获得 CPU 时间，转为运行状态；而运行状态的进程，在分配给它的 CPU 时间片用完之后就会转为就绪状态，等待下一次调度。
- 阻塞状态是缺少需要的资源从而由运行状态转换而来，但是该资源不包括 CPU 时间，缺少 CPU 时间会从运行态转换为就绪态。

## 9、一个程序从开始运行到结束的完整过程？

四个过程：

### (1) 预编译

主要处理源代码文件中的以“#”开头的预编译指令。处理规则见下

- 1、删除所有的#define，展开所有的宏定义。
- 2、处理所有的条件预编译指令，如“#if”、“#endif”、“#ifdef”、“#elif”和“#else”。
- 3、处理“#include”预编译指令，将文件内容替换到它的位置，这个过程是递归进行的，文件中包含其他文件。
- 4、删除所有的注释，“//”和“/\*\*/”。
- 5、保留所有的#pragma 编译器指令，编译器需要用到他们，如：#pragma once 是为了防止有文件被重复引用。
- 6、添加行号和文件标识，便于编译时编译器产生调试用的行号信息，和编译时产生编译错误或警告是能够显示行号。

### (2) 编译

把预编译之后生成的xxx.i或xxx.ii文件，进行一系列词法分析、语法分析、语义分析及优化后，生成相应的汇编代码文件。

- 1、词法分析：利用类似于“有限状态机”的算法，将源代码程序输入到扫描机中，将其中的字符序列分割成一系列的记号。
- 2、语法分析：语法分析器对由扫描器产生的记号，进行语法分析，产生语法树。由语法分析器输出的语法树是一种以表达式为节点的树。
- 3、语义分析：语法分析器只是完成了对表达式语法层面的分析，语义分析器则对表达式是否有意义进行判断，其分析的语义是静态语义——在编译期能分期的语义，相对应的动态语义是在运行期才能确定的语义。
- 4、优化：源代码级别的一个优化过程。
- 5、目标代码生成：由代码生成器将中间代码转换成目标机器代码，生成一系列的代码序列——汇编语言表示。
- 6、目标代码优化：目标代码优化器对上述的目标机器代码进行优化：寻找合适的寻址方式、使用位移来替代乘法运算、删除多余的指令等。

### (3) 汇编

将汇编代码转变成机器可以执行的指令(机器码文件)。汇编器的汇编过程相对于编译器来说更简单，没有复杂的语法，也没有语义，更不需要做指令优化，只是根据汇编指令和机器指令的对照表——翻译过来，汇编过程有汇编器完成。经汇编之后，产生目标文件(与可执行文件格式几乎一样)xxx.o(Linux下)、xxx.obj(Windows下)。

### (4) 链接

将不同的源文件产生的目标文件进行链接，从而形成一个可以执行的程序。链接分为静态链接和动态链接：

#### 1、静态链接：

函数和数据被编译进一个二进制文件。在使用静态库的情况下，在编译链接可执行文件时，链接器从库中复制这些函数和数据并把它们和应用程序的其它模块组合起来创建最终的可执行文件。

空间浪费：因为每个可执行程序中对所有需要的目标文件都要有一份副本，所以如果多个程序对同一个目标文件都有依赖，会出现同一个目标文件都在内存存在多个副本；

更新困难：每当库函数的代码修改了，这个时候就需要重新进行编译链接形成可执行程序。

运行速度快：但是静态链接的优点就是，在可执行程序中已经具备了所有执行程序所需要的任何东西，在执行的时候运行速度快。

#### 2、动态链接：

动态链接的基本思想是把程序按照模块拆分成各个相对独立部分，在程序运行时才将它们链接在一起形成一个完整的程序，而不是像静态链接一样把所有程序模块都链接成一个单独的可执行文件。

共享库：就是即使需要每个程序都依赖同一个库，但是该库不会像静态链接那样在内存中存在多份副本，而是这多个程序在执行时共享同一份副本；

更新方便：更新时只需要替换原来的目标文件，而无需将所有的程序再重新链接一遍。当程序下一次运行时，新版本的目标文件会被自动加载到内存并且链接起来，程序就完成了升级的目标。

性能损耗：因为把链接推迟到了程序运行时，所以每次执行程序都需要进行链接，所以性能会有一定损失。

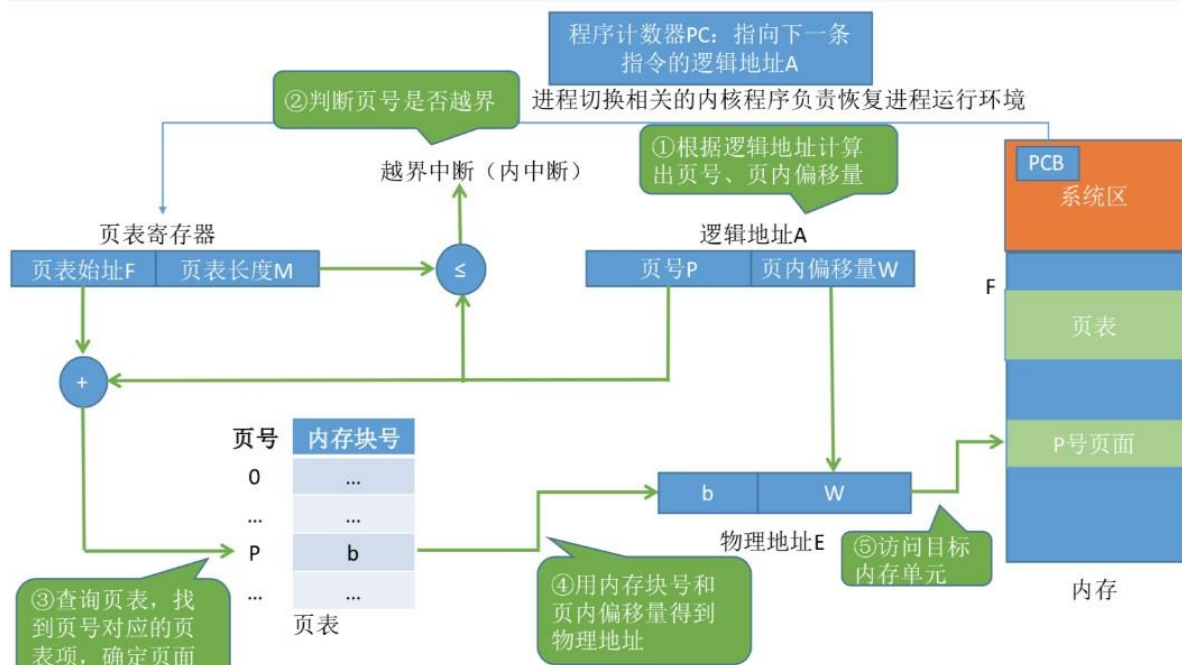
## 10、通过例子讲解逻辑地址转换为物理地址的基本过程

可以借助进程的页表将逻辑地址转换为物理地址。

通常会在系统中设置一个页表寄存器(PTR)，存放页表在内存中的起始地址F和页表长度M。进程未执行时，页表的始址和页表长度放在进程控制块(PCB)中，当进程被调度时，操作系统内核会把它放到页表寄存器中。

注意：页面大小是2的整数幂

设页面大小为L，逻辑地址A到物理地址E的变换过程如下：



- ①计算页号  $P$  和页内偏移量  $W$  (如果用十进制数手算, 则  $P=A/L$ ,  $W=A\%L$ ; 但是在计算机实际运行时, 逻辑地址结构是固定不变的, 因此计算机硬件可以更快地得到二进制表示的页号、页内偏移量)
- ②比较页号  $P$  和页表长度  $M$ , 若  $P \geq M$ , 则产生越界中断, 而页表长度至少是1, 因此  $P=M$  时也会产生越界中断。
- ③页表中页号  $P$  对应的页表项地址 = 页表起始地址 +  $P \times$  页表项长度, 即为内存块号。(注意区分页表项长度、页表项地址。页表项长度是指一个页表项占多大的存储空间; 页表项地址是指一个页表项在内存中的地址)
- ④计算  $E = b * L + W$ , 用得到的物理地址  $E$  去访问。(如果内存块号、页面偏移量是用二进制表示的, 那么把二者拼接起来就是最终的物理地址了)

例:若页面大小  $L$  为1K字节, 页号2对应的内存块号  $b=8$ , 将逻辑地址  $A=2500$  转换为物理地址  $E$ 。  
等价描述: 某系统按字节寻址, 逻辑地址结构中, 页内偏移量占10位(说明一个页面的大小为  $2^{10}B = 1KB$ ), 页号2对应的内存块号  $b=8$ , 将逻辑地址  $A=2500$  转换为物理地址  $E$ 。

#### ①计算页号、页内偏移量

页号  $P = A/L = 2500/1024 = 2$ ; 页内偏移量  $W = A\%L = 2500\%1024 = 452$

#### ②根据题中条件可知, 页号2没有越界, 其存放的内存块号 $b=8$

#### ③物理地址 $E = b * L + W = 8 * 1024 + 425 = 8644$

在分页存储管理(页式管理)的系统中, 只要确定了每个页面的大小, 逻辑地址结构就确定了。因此, 页式管理中地址是二维的。即, 只要给出一个逻辑地址, 系统就可以自动地算出页号、页内偏移量两个部分, 并不需要显式地告诉系统这个逻辑地址中, 页内偏移量占多少位。

## 11、进程同步的四种方法?

### 1. 临界区

对临界资源进行访问的那段代码称为临界区。

为了互斥访问临界资源, 每个进程在进入临界区之前, 需要先进行检查。

```

1 // entry section
2 // critical section;
3 // exit section
4
```

## 2. 同步与互斥

- 同步：多个进程因为合作产生的直接制约关系，使得进程有一定的先后执行关系。
- 互斥：多个进程在同一时刻只有一个进程能进入临界区。

## 3. 信号量

信号量(Semaphore)是一个整型变量，可以对其执行 down 和 up 操作，也就是常见的 P 和 V 操作。

- **down**：如果信号量大于 0，执行 -1 操作；如果信号量等于 0，进程睡眠，等待信号量大于 0；
- **up**：对信号量执行 +1 操作，唤醒睡眠的进程让其完成 down 操作。

down 和 up 操作需要被设计成原语，不可分割，通常的做法是在执行这些操作的时候屏蔽中断。

如果信号量的取值只能为 0 或者 1，那么就成为了 **互斥量 (Mutex)**，0 表示临界区已经加锁，1 表示临界区解锁。

```
1  typedef int semaphore;
2  semaphore mutex = 1;
3  void P1() {
4      down(&mutex);
5      // 临界区
6      up(&mutex); 7 }
8
9  void P2() {
10     down(&mutex);
11     // 临界区
12     up(&mutex); 13
14 }
```

## 12 使用信号量实现生产者-消费者问题

问题描述：使用一个缓冲区来保存物品，只有缓冲区没有满，生产者才可以放入物品；只有缓冲区不为空，消费者才可以拿走物品。

因为缓冲区属于临界资源，因此需要使用一个互斥量 mutex 来控制对缓冲区的互斥访问。

为了同步生产者和消费者的行为，需要记录缓冲区中物品的数量。数量可以使用信号量来进行统计，这里需要使用两个信号量：empty 记录空缓冲区的数量，full 记录满缓冲区的数量。

其中，empty 信号量是在生产者进程中使用，当 empty 不为 0 时，生产者才可以放入物品；full 信号量是在消费者进程中使用，当 full 信号量不为 0 时，消费者才可以取走物品。

**注意**，不能先对缓冲区进行加锁，再测试信号量。也就是说，不能先执行 down(mutex) 再执行 down(empty)。如果这么做了，那么可能会出现这种情况：生产者对缓冲区加锁后，执行 down(empty) 操作，发现 empty = 0，此时生产者睡眠。

消费者不能进入临界区，因为生产者对缓冲区加锁了，消费者就无法执行 up(empty) 操作，empty 永远都为 0，导致生产者永远等待下，不会释放锁，消费者因此也会永远等待下去。

```
#define N 100
typedef int semaphore;
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;
void producer() {
    while (TRUE) {
        int item = produce_item();
        down(&empty);
        down(&mutex);
```

```
        insert_item(item);
        up(&mutex);
        up(&full);
    }
}

void consumer() {
    while (TRUE) {
        down(&full);
        down(&mutex);
        int item = remove_item();
        consume_item(item);
        up(&mutex);
        up(&empty);
    }
}
```

#### 4. 管程

使用信号量机制实现的生产者消费者问题需要客户端代码做很多控制，而管程把控制的代码独立出来，不仅不容易出错，也使得客户端代码调用更容易。

c 语言不支持管程，下面的示例代码使用了类 Pascal 语言来描述管程。示例代码的管程提供了 insert()

和 remove() 方法，客户端代码通过调用这两个方法来解决生产者-消费者问题。

```
1  monitor ProducerConsumer
2      integer i;
3      condition c;
4
5      procedure insert();
6      begin
7          // ...
8      end;
9
10     procedure remove();
11     begin
12         // ...
13     end;
14 end monitor;
15
```

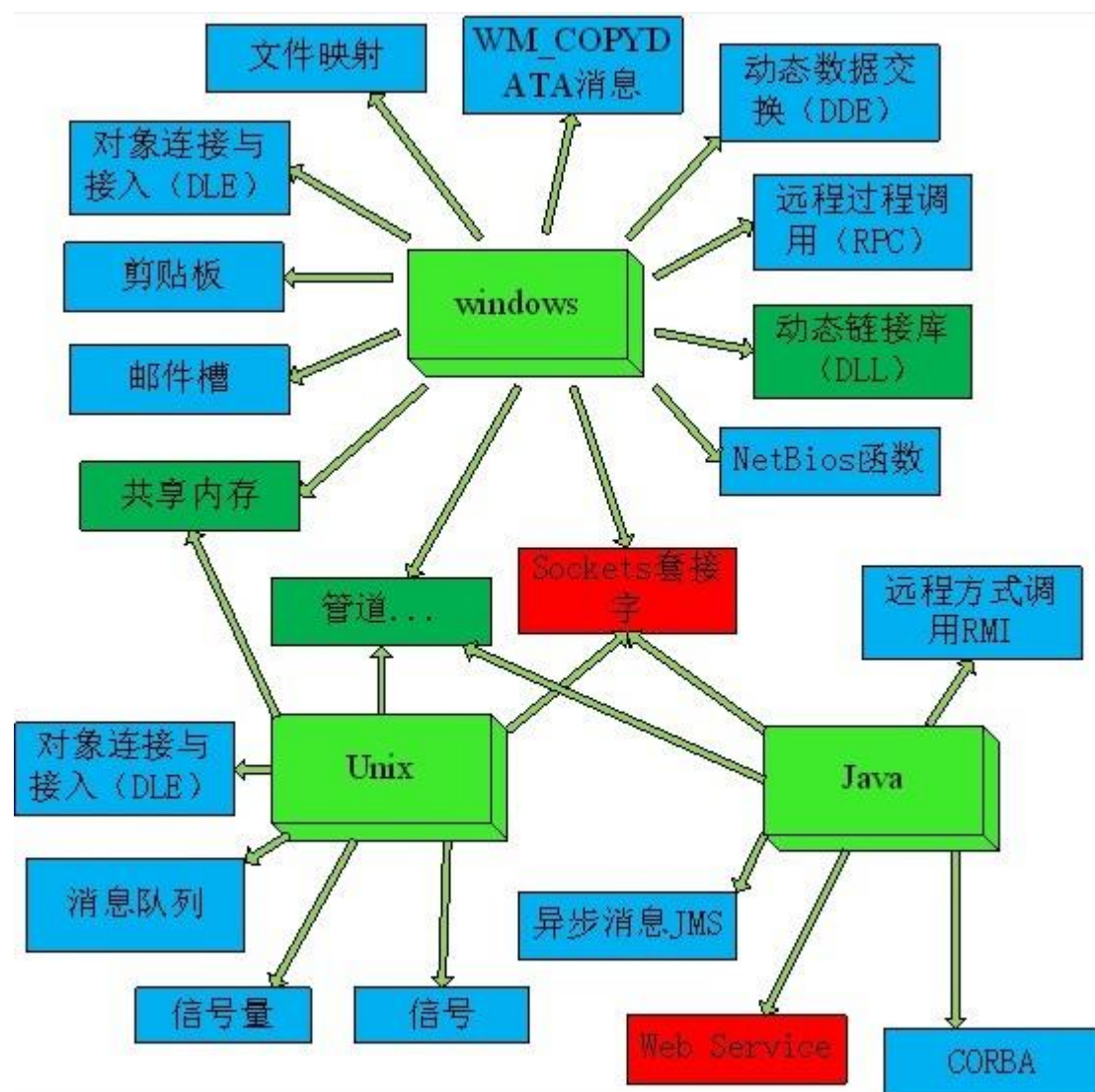
管程有一个重要特性：在一个时刻只能有一个进程使用管程。进程在无法继续执行的时候不能一直占用管程，否则其它进程永远不能使用管程。

管程引入了 **条件变量** 以及相关的操作：**wait()** 和 **signal()** 来实现同步操作。对条件变量执行 wait() 操作会导致调用进程阻塞，把管程让出来给另一个进程持有。signal() 操作用于唤醒被阻塞的进程。

### 13、进程通信方法（Linux和windows下），线程通信方法（Linux和windows下）

#### 进程通信方法





## 名称及方式

管道(pipe): 允许一个进程和另一个与它有共同祖先的进程之间进行通信

命名管道(FIFO): 类似于管道, 但是它可以用于任何两个进程之间的通信, 命名管道在文件系统中 有对应的文件名。命名管道通过命令mkfifo或系统调用mkfifo来创建

消息队列(MQ): 消息队列是消息的连接表, 包括POSIX消息对和System V消息队列。有足够权限的进程可以向队列中添加消息, 被赋予读权限的进程则可以读走队列中的消息。消息队列克服了信号承载信息量少, 管道只能承载无格式字节流以及缓冲区大小受限等缺点;

信号量(semaphore): 信号量主要作为进程间以及同进程不同线程之间的同步手段;

共享内存(shared memory): 它使得多个进程可以访问同一块内存空间, **是最快的可用IPC形式**。这是针对其他通信机制运行效率较低而设计的。它往往与其他通信机制, 如信号量结合使用, 以达到进程间的同步及互斥

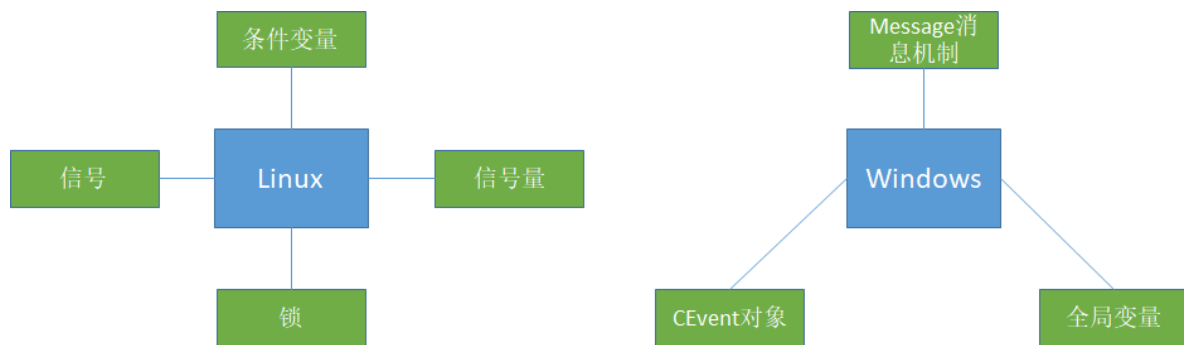
信号(signal): 信号是比较复杂的通信方式, 用于通知接收进程有某种事情发生, 除了用于进程间通信外, 进程还可以发送信号给进程本身

内存映射(mapped memory): 内存映射允许任何多个进程间通信, 每一个使用该机制的进程通过把一个共享的文件映射到自己的进程地址空间来实现它

Socket: 它是更为通用的进程间通信机制, 可用于不同机器之间的进程间通信

## 线程通信方法





名称及含义
<b>Linux:</b>
信号：类似进程间的信号处理
锁机制：互斥锁、读写锁和自旋锁
条件变量：使用通知的方式解锁，与互斥锁配合使用
信号量：包括无名线程信号量和命名线程信号量
<b>Windows:</b>
全局变量：需要多个线程来访问一个全局变量时，通常我们会在这个全局变量前加上volatile声明，以防编译器对此变量进行优化
Message消息机制：常用的Message通信的接口主要有两个：PostMessage和PostThreadMessage，PostMessage为线程向主窗口发送消息。而PostThreadMessage是任意两个线程之间的通信接口。
CEvent对象：CEvent为MFC中的一个对象，可以通过对CEvent的触发状态进行改变，从而实现线程间的通信和同步，这个主要是实现线程直接同步的一种方法。

## 14、进程间通信有哪几种方式？把你知道的都说出来

Linux几乎支持全部UNIX进程间通信方法，包括管道（有名管道和无名管道）、消息队列、共享内存、信号量和套接字。其中前四个属于同一台机器下进程间的通信，套接字则是用于网络通信。

### 管道

- 无名管道
  - 无名管道特点：
    - 无名管道是一种特殊的文件，这种文件只存在于内存中。
    - 无名管道只能用于父子进程或兄弟进程之间，必须用于具有亲缘关系的进程间的通信。
    - 无名管道只能由一端向另一端发送数据，是半双工方式，如果双方需要同时收发数据需要两个管道。
  - 相关接口：
    - `int pipe(int fd[2]);`
      - `fd[2]`：管道两端用`fd[0]`和`fd[1]`来描述，读的一端用`fd[0]`表示，写的一端用`fd[1]`表示。通信双方的进程中写数据的一方需要把`fd[0]`先close掉，读的一方需要先把`fd[1]`给close掉。
- 有名管道：

- 有名管道特点：
  - 有名管道是FIFO文件，存在于文件系统中，可以通过文件路径名来指出。
  - 有名管道可以在不具有亲缘关系的进程间进行通信。
- 相关接口：
  - `int mkfifo(const char *pathname, mode_t mode);`
    - `pathname`：即将创建的FIFO文件路径，如果文件存在需要先删除。
    - `mode`：和`open()`中的参数相同。

## 消息队列

相比于 FIFO，消息队列具有以下优点：

- 消息队列可以独立于读写进程存在，从而避免了 FIFO 中同步管道的打开和关闭时可能产生的困难；
- 避免了 FIFO 的同步阻塞问题，不需要进程自己提供同步方法；
- 读进程可以根据消息类型有选择地接收消息，而不像 FIFO 那样只能默认地接收。

## 共享内存

进程可以将同一段共享内存连接到它们自己的地址空间，所有进程都可以访问共享内存中的地址，如果某个进程向共享内存内写入数据，所做的改动将立即影响到可以访问该共享内存的其他所有进程。

- 相关接口
  - 创建共享内存：`int shmget(key_t key, int size, int flag);`  
成功时返回一个和key相关的共享内存标识符，失败返回-1。
    - `key`：为共享内存段命名，多个共享同一片内存的进程使用同一个key。
    - `size`：共享内存容量。
    - `flag`：权限标志位，和`open`的`mode`参数一样。
  - 连接到共享内存地址空间：`void *shmat(int shmid, void *addr, int flag);`  
返回值即共享内存实际地址。
    - `shmid`：`shmget()`返回的标识。
    - `addr`：决定以什么方式连接地址。
    - `flag`：访问模式。
  - 从共享内存分离：`int shmdt(const void *shmaddr);`  
调用成功返回0，失败返回-1。
    - `shmaddr`：是`shmat()`返回的地址指针。
- 其他补充

共享内存的方式像极了多线程中线程对全局变量的访问，大家都对等地有权去修改这块内存的值，这就导致在多线程并发下，最终结果是不可预期的。所以对这块临界区的访问需要通过信号量来进行进程同步。

但共享内存的优势也很明显，首先可以通过共享内存进行通信的进程不需要像无名管道一样需要通信的进程间有亲缘关系。其次内存共享的速度也比较快，不存在读取文件、消息传递等过程，只需要到相应映射到的内存地址直接读写数据即可。

## 信号量

在提到共享内存方式时也提到，进程共享内存和多线程共享全局变量非常相似。所以在使用内存共享的方式时也需要通过信号量来完成进程间同步。多线程同步的信号量是POSIX信号量，而在进程里使用SYSTEM V信号量。

- 相关接口

- 创建信号量: `int semget(key_t key, int nsems, int semflag);`

创建成功返回信号量标识符, 失败返回-1。

- `key`: 进程 `pid`。
- `nsems`: 创建信号量的个数。
- `semflag`: 指定信号量读写权限。

- 改变信号量值: `int semop(int semid, struct sembuf *sops, unsigned nsops);`

我们所需要做的主要工作就是串讲`sembuf`变量并设置其值, 然后调用`semop`, 把设置好的`sembuf`变量传递进去。

`struct sembuf`结构体定义如下:

```
1 struct sembuf{
2     short sem_num;
3     short sem_op;
4     short sem_flg;
5 }
```

成功返回信号量标识符, 失败返回-1。

- `semid`: 信号量集标识符, 由`semget()`函数返回。
- `sops`: 指向`struct sembuf`结构的指针, 先设置好`sembuf`值再通过指针传递。
- `nsops`: 进行操作信号量的个数, 即`sops`结构变量的个数, 需大于或等于1。最常见设置此值等于1, 只完成对一个信号量的操作。
- 直接控制信号量信息: `int semctl(int semid, int semnum, int cmd, union semun arg);`
  - `semid`: 信号量集标识符。
  - `semnum`: 信号量集数组上的下标, 表示某一个信号量。
  - `arg`: `union semun`类型。

## 辅助命令

`ipcs`命令用于报告共享内存、信号量和消息队列信息。

- `ipcs -a`: 列出共享内存、信号量和消息队列信息。
- `ipcs -l`: 列出系统限额。
- `ipcs -u`: 列出当前使用情况。

## 套接字

与其它通信机制不同的是, 它可用于不同机器间的进程通信。

## 15、虚拟内存的目的是什么?

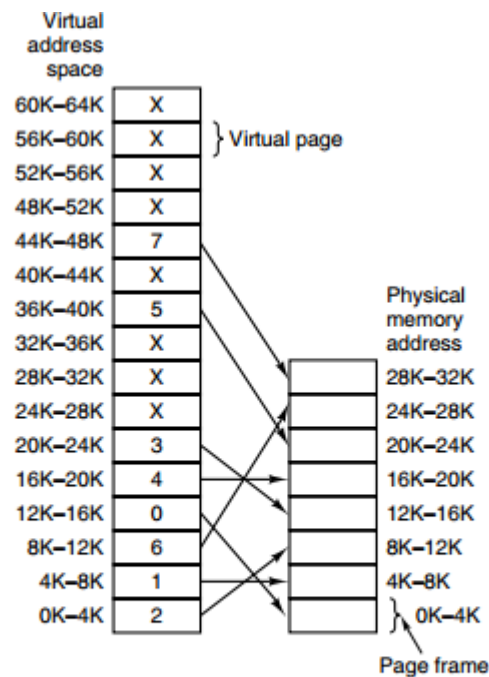
虚拟内存的目的是为了让物理内存扩充成更大的逻辑内存, 从而让程序获得更多的可用内存。

为了更好的管理内存, 操作系统将内存抽象成地址空间。每个程序拥有自己的地址空间, 这个地址空间被分割成多个块, 每一块称为一页。

这些页被映射到物理内存, 但不需要映射到连续的物理内存, 也不需要所有页都必须在物理内存中。当程序引用到不在物理内存中的页时, 由硬件执行必要的映射, 将缺失的部分装入物理内存并重新执行失败的指令。

从上面的描述中可以看出, 虚拟内存允许程序不用将地址空间中的每一页都映射到物理内存, 也就是说一个程序不需要全部调入内存就可以运行, 这使得有限的内存运行大程序成为可能。

例如有一台计算机可以产生 16 位地址，那么一个程序的地址空间范围是 0~64K。该计算机只有 32KB 的物理内存，虚拟内存技术允许该计算机运行一个 64K 大小的程序。



**Figure 3-9.** The relation between virtual addresses and physical memory addresses is given by the **page table**. Every page begins on a multiple of 4096 and ends 4095 addresses higher, so 4K-8K really means 4096-8191 and 8K to 12K means 8192-12287.

## 16、说一下你理解中的内存？他有什么作用呢？

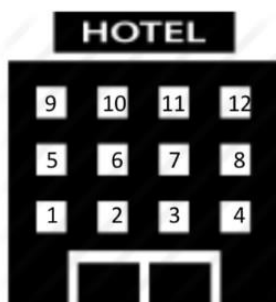
### 什么是内存？有何作用？

内存是用于存放数据的硬件。程序执行前需要放到内存中才能被CPU处理。



思考：在多道程序环境下，系统中会有多个程序并发执行，也就是说会有多个程序的数据需要同时放到内存中。那么，如何区分各个程序的数据是放在什么地方的呢？

方案：给内存的存储单元编地址



内存地址从0开始，每个地址对应一个存储单元

地址	内存
0	“小房间”
1	“小房间”
2	.....
3	
4	
5	
6	

内存中也有一个一个的“小房间”，每个小房间就是一个“存储单元”

如果计算机“按字节编址”，则每个存储单元大小为1字节，即1B，即8个二进制位

如果字长为16位的计算机“按字编址”，则每个存储单元大小为1个字；每个字的大小为16个二进制位

## 17、介绍一下几种典型的锁

读写锁

- 多个读者可以同时进行读
- 写者必须互斥（只允许一个写者写，也不能读者写者同时进行）
- 写者优先于读者（一旦有写者，则后续读者必须等待，唤醒时优先考虑写者）

## 互斥锁

一次只能一个线程拥有互斥锁，其他线程只有等待

互斥锁是在抢锁失败的情况下主动放弃CPU进入睡眠状态直到锁的状态改变时再唤醒，而操作系统负责线程调度，为了实现锁的状态发生改变时唤醒阻塞的线程或者进程，需要把锁交给操作系统管理，所以互斥锁在加锁操作时涉及上下文的切换。互斥锁实际的效率还是可以让人接受的，加锁的时间大概100ns左右，而实际上互斥锁的一种可能的实现是先自旋一段时间，当自旋的时间超过阈值之后再将线程投入睡眠中，因此在并发运算中使用互斥锁（每次占用锁的时间很短）的效果可能不亚于使用自旋锁

## 条件变量

互斥锁一个明显的缺点是他只有两种状态：锁定和非锁定。而条件变量通过允许线程阻塞和等待另一个线程发送信号的方法弥补了互斥锁的不足，他常和互斥锁一起使用，避免出现竞态条件。当条件不满足时，线程往往解开相应的互斥锁并阻塞线程然后等待条件发生变化。一旦其他的某个线程改变了条件变量，他将通知相应的条件变量唤醒一个或多个正被此条件变量阻塞的线程。总的来说**互斥锁是线程间互斥的机制，条件变量则是同步机制。**

## 自旋锁

如果进线程无法取得锁，进线程不会立刻放弃CPU时间片，而是一直循环尝试获取锁，直到获取为止。如果别的线程长时期占有锁，那么自旋就是在浪费CPU做无用功，但是自旋锁一般应用于加锁时间很短的场景，这个时候效率比较高。

### 1. 线程（POSIX）锁有哪些？

- 互斥锁（mutex）
  - 互斥锁属于sleep-waiting类型的锁。例如在一个双核的机器上有两个线程A和B，它们分别运行在core 0和core 1上。假设线程A想要通过pthread\_mutex\_lock操作去得到一个临界区的锁，而此时这个锁正被线程B所持有，那么线程A就会被阻塞，此时会通过上下文切换将线程A置于等待队列中，此时core0就可以运行其他的任务（如线程C）。
- 条件变量(cond)
- 自旋锁(spin)
  - 自旋锁属于busy-waiting类型的锁，如果线程A是使用pthread\_spin\_lock操作去请求锁，如果自旋锁已经被线程B所持有，那么线程A就会一直在core 0上进行忙等待并不停的进行锁请求，检查该自旋锁是否已经被线程B释放，直到得到这个锁为止。因为自旋锁不会引起调用者睡眠，所以自旋锁的效率远高于互斥锁。
  - 虽然它的效率比互斥锁高，但是它也有些不足之处：
    - 自旋锁一直占用CPU，在未获得锁的情况下，一直进行自旋，所以占用着CPU，如果不能在很短的时间内获得锁，无疑会使CPU效率降低。
    - 在用自旋锁时有可能造成死锁，当递归调用时有可能造成死锁。
  - 自旋锁只有在内核可抢占式或SMP的情况下才真正需要，在单CPU且不可抢占式的内核下，自旋锁的操作为空操作。自旋锁适用于锁使用者保持锁时间比较短的情况下。

## 18、逻辑地址VS物理地址

Eg:编译时只需确定变量x存放的相对地址是100（也就是说相对于进程在内存中的起始地址而言的地址）。CPU想要找到x在内存中的实际存放位置，只需要用进程的起始地址+100即可。

相对地址又称逻辑地址，绝对地址又称物理地址。

## 19、怎么回收线程？有哪几种方法？

- 等待线程结束：int pthread\_join(pthread\_t tid, void\*\* retval);  
主线程调用，等待子线程退出并回收其资源，类似于进程中wait/waitpid回收僵尸进程，调用pthread\_join的线程会被阻塞。
  - tid：创建线程时通过指针得到tid值。
  - retval：指向返回值的指针。
- 结束线程：pthread\_exit(void \*retval);  
子线程执行，用来结束当前线程并通过retval传递返回值，该返回值可通过pthread\_join获得。
  - retval：同上。
- 分离线程：int pthread\_detach(pthread\_t tid);  
主线程、子线程均可调用。主线程中pthread\_detach(tid)，子线程中pthread\_detach(pthread\_self())，调用后和主线程分离，子线程结束时自己立即回收资源。
  - tid：同上。

## 20、一个由C/C++编译的程序占用的内存分为哪几个部分？

- 1、栈区 (stack) — 地址向下增长，由编译器自动分配释放，存放函数的参数值，局部变量的值等。其操作方式类似于数据结构中的队列，先进后出。
- 2、堆区 (heap) — 地址向上增长，一般由程序员分配释放，若程序员不释放，程序结束时可能由OS 回收。注意它与数据结构中的堆是两回事，分配方式倒是类似于链表。
- 3、全局区 (静态区) (static) — 全局变量和静态变量的存储是放在一块的，初始化的全局变量和静态变量在一块区域，未初始化的全局变量和未初始化的静态变量在相邻的另一块区域。 - 程序结束后有系统释放

- 4、文字常量区 一常量字符串就是放在这里的。程序结束后由系统释放
- 5、程序代码区(text)—存放函数体的二进制代码。

## 21、从堆和栈上建立对象哪个快？（考察堆和栈的分配效率比较）

从两方面来考虑：

- 分配和释放，堆在分配和释放时都要调用函数(malloc,free)，比如分配时会到堆空间去寻找足够 大小的空间（因为多次分配释放后会造成内存碎片），这些都会花费一定的时间，具体可以看看malloc和free的源代码，函数做了很多额外的工作，而栈却不需要这些。
- 访问时间，访问堆的一个具体单元，需要两次访问内存，第一次得取得指针，第二次才是真正的数据，而栈只需访问一次。另外，堆的内容被操作系统交换到外存的概率比栈大，栈一般是不会被交换出去的。

## 22、常见内存分配方式有哪些？

### 内存分配方式

- (1) 从静态存储区域分配。内存存在程序编译的时候就已经分配好，这块内存存在程序的整个运行期间都存在。例如全局变量，static变量。
- (2) 在栈上创建。在执行函数时，函数内局部变量的存储单元都可以在栈上创建，函数执行结束时这些存储单元自动被释放。栈内存分配运算内置于处理器的指令集中，效率很高，但是分配的内存容量有限。
- (3) 从堆上分配，亦称动态内存分配。程序在运行的时候用malloc或new申请任意多少的内存，程序员自己负责在何时用free或delete释放内存。动态内存的生存期由我们决定，使用非常灵活，但问题也最多。

## 23、常见内存分配内存错误

- (1) 内存分配未成功，却使用了它。



编程新手常犯这种错误，因为他们没有意识到内存分配会不成功。常用解决办法是，在使用内存之前检查指针是否为NULL。如果指针p是函数的参数，那么在函数的入口处用assert(p!=NULL)进行检查。如果是用malloc或new来申请内存，应该用if(p==NULL) 或if(p!=NULL)进行防错处理。

(2) 内存分配虽然成功，但是尚未初始化就引用它。

犯这种错误主要有两个起因：一是没有初始化的观念；二是误以为内存的缺省初值全为零，导致引用初值错误（例如数组）。内存的缺省初值究竟是什么并没有统一的标准，尽管有些时候为零值，我们宁可 信其无不可信其有。所以无论用何种方式创建数组，都别忘了赋初值，即便是赋零值也不可省略，不要 嫌麻烦。

(3) 内存分配成功并且已经初始化，但操作越过了内存的边界。

例如在使用数组时经常发生下标“多1”或者“少1”的操作。特别是在for循环语句中，循环次数很容易搞错，导致数组操作越界。

(4) 忘记了释放内存，造成内存泄露。

含有这种错误的函数每被调用一次就丢失一块内存。刚开始时系统的内存充足，你看不到错误。终有一次程序突然挂掉，系统出现提示：内存耗尽。动态内存的申请与释放必须配对，程序中malloc与free的使用次数一定要相同，否则肯定有错误（new/delete同理）。

(5) 释放了内存却继续使用它。常见于以下有三种情况：

- 程序中的对象调用关系过于复杂，实在难以搞清楚某个对象究竟是否已经释放了内存，此时应该重新设计数据结构，从根本上解决对象管理的混乱局面。
- 函数的return语句写错了，注意不要返回指向“栈内存”的“指针”或者“引用”，因为该内存存在函数体结束时被自动销毁。
- 使用free或delete释放了内存后，没有将指针设置为NULL。导致产生“野指针”。

## 24、死锁相关问题大总结，！

**死锁是指两个（多个）线程相互等待对方数据的过程，死锁的产生会导致程序卡死，不解锁程序将永远 无法进行下去。**

### 1、死锁产生原因

举个例子：两个线程A和B，两个数据1和2。线程A在执行过程中，首先对资源1加锁，然后再去给资源2加锁，但是由于线程的切换，导致线程A没能给资源2加锁。线程切换到B后，线程B先对资源2加锁，然后再去给资源1加锁，由于资源1已经被线程A加锁，因此线程B无法加锁成功，当线程切换为A时，A也无法成功对资源2加锁，由此就造成了线程AB双方相互对一个已加锁资源的等待，死锁产生。



理论上认为死锁产生有以下四个必要条件，缺一不可：

1. **互斥条件**：进程对所需求的资源具有排他性，若有其他进程请求该资源，请求进程只能等待。
2. **不剥夺条件**：进程在所获得的资源未释放前，不能被其他进程强行夺走，只能自己释放。
3. **请求和保持条件**：进程当前所拥有的资源在进程请求其他新资源时，由该进程继续占有。
4. **循环等待条件**：存在一种进程资源循环等待链，链中每个进程已获得的资源同时被链中下一个进程所请求。

## 2、死锁演示

通过代码的形式进行演示，需要两个线程和两个互斥量。

```
#include <iostream>
#include <vector>
#include <list>
#include <thread>
#include <mutex>    //引入互斥量头文件
using namespace std;
class A {
public:
    //插入消息，模拟消息不断产生
    void insertMsg() {
        for (int i = 0; i < 100; i++) {
            cout << "插入一条消息:" << i << endl;
            my_mutex1.lock(); //语句1
            my_mutex2.lock(); //语句2
            Msg.push_back(i);
            my_mutex2.unlock();
            my_mutex1.unlock();
        }
    }
    //读取消息
    void readMsg() {
        int MsgCom;
        for (int i = 0; i < 100; i++) {
            MsgCom = MsgLULProc(i);
            if (MsgLULProc(MsgCom)) {
                //读出消息了
                cout << "消息已读出" << MsgCom << endl;
            }
            else {
                //消息暂时为空
                cout << "消息为空" << endl;
            }
        }
    }
    //加解锁代码
    bool MsgLULProc(int& command) {
        int curMsg;
        my_mutex2.lock();    //语句3
        my_mutex1.lock();    //语句4
        if (!Msg.empty()) {
            //读取消息，读完删除
            command = Msg.front();
            Msg.pop_front();
            my_mutex1.unlock();
            my_mutex2.unlock();
            return true;
        }
        my_mutex1.unlock();
        my_mutex2.unlock();
        return false;
    }
private:
    std::list<int> Msg;    //消息变量
```

```
std::mutex my_mutex1; //互斥量对象1
std::mutex my_mutex2; //互斥量对象2
};
int main() {
    A a;
    //创建一个插入消息线程
    std::thread insertTd(&A::insertMsg, &a); //这里要传入引用保证是同一个对象
    //创建一个读取消息线程
    std::thread readTd(&A::readMsg, &a); //这里要传入引用保证是同一个对象
    insertTd.join();
    readTd.join();
    return 0;
}
```

语句1和语句2表示线程A先锁资源1，再锁资源2，语句3和语句4表示线程B先锁资源2再锁资源1，具备死锁产生的条件。

### 3、死锁的解决方案

保证上锁的顺序一致。

### 4、死锁必要条件

- 互斥条件：进程对所需求的资源具有排他性，若有其他进程请求该资源，请求进程只能等待。不
- 剥夺条件：进程在所获得的资源未释放前，不能被其他进程强行夺走，只能自己释放
- 请求和保持条件：进程当前所拥有的资源在进程请求其他新资源时，由该进程继续占有。
- 循环等待条件：存在一种进程资源循环等待链，链中每个进程已获得的资源同时被链中下一个进程所请求。

### 5、处理方法

主要有以下四种方法：

- 鸵鸟策略
- 死锁检测与死锁恢复
- 死锁预防
- 死锁避免

#### 鸵鸟策略

把头埋在沙子里，假装根本没发生问题。

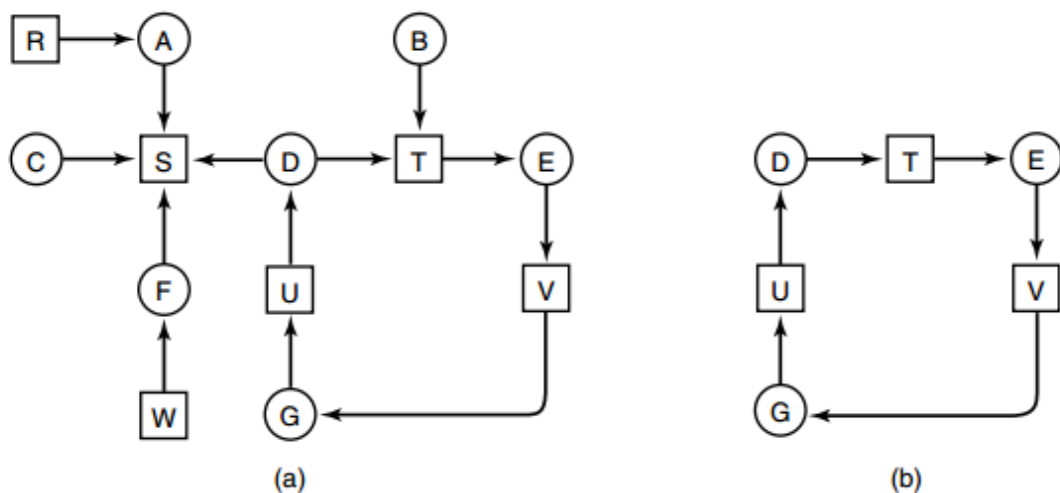
因为解决死锁问题的代价很高，因此鸵鸟策略这种不采取任何措施的方案会获得更高的性能。当发生死锁时不会对用户造成多大影响，或发生死锁的概率很低，可以采用鸵鸟策略。

大多数操作系统，包括 Unix, Linux 和 Windows, 处理死锁问题的办法仅仅是忽略它。

## 死锁检测与死锁恢复

不试图阻止死锁，而是当检测到死锁发生时，采取措施进行恢复。

### 1、每种类型一个资源的死锁检测



上图为资源分配图，其中方框表示资源，圆圈表示进程。资源指向进程表示该资源已经分配给该进程，进程指向资源表示进程请求获取该资源。

图 a 可以抽取出环，如图 b，它满足了环路等待条件，因此会发生死锁。

每种类型一个资源的死锁检测算法是通过检测有向图是否存在环来实现，从一个节点出发进行深度优先搜索，对访问过的节点进行标记，如果访问了已经标记的节点，就表示有向图存在环，也就是检测到死锁的发生。

### 2、每种类型多个资源的死锁检测

	Tape drives	Plotters	Scanners	Blu-rays
E =	( 4	2	3	1 )
A =	( 2	1	0	0 )
Current allocation matrix				
C =	$\begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{bmatrix}$			
Request matrix				
R =	$\begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{bmatrix}$			

上图中，有三个进程四个资源，每个数据代表的含义如下：

- E 向量：资源总量
- A 向量：资源剩余量
- C 矩阵：每个进程所拥有的资源数量，每一行都代表一个进程拥有资源的数量
- R 矩阵：每个进程请求的资源数量

进程  $P_1$  和  $P_2$  所请求的资源都得不到满足，只有进程  $P_3$  可以，让  $P_3$  执行，之后释放  $P_3$  拥有的资源，此时  $A = (2 \ 2 \ 2 \ 0)$ 。 $P_2$  可以执行，执行后释放  $P_2$  拥有的资源， $A = (4 \ 2 \ 2 \ 1)$ 。 $P_1$  也可以执行。所有进程都可以顺利执行，没有死锁。

算法总结如下：

每个进程最开始时都不被标记，执行过程有可能被标记。当算法结束时，任何没有被标记的进程都是死锁进程。

1. 寻找一个没有标记的进程  $P_i$ ，它所请求的资源小于等于  $A$ 。
2. 如果找到了这样一个进程，那么将  $C$  矩阵的第  $i$  行向量加到  $A$  中，标记该进程，并转回 1。
3. 如果没有这样一个进程，算法终止。

## 6、死锁恢复

- 利用抢占恢复
- 利用回滚恢复
- 通过杀死进程恢复

## 7、死锁预防

在程序运行之前预防发生死锁。

1. 破坏互斥条件

例如假脱机打印机技术允许若干个进程同时输出，唯一真正请求物理打印机的进程是打印机守护进程。

2. 破坏请求和保持条件

一种实现方式是规定所有进程在开始执行前请求所需要的全部资源。

3. 破坏不剥夺条件

允许抢占资源

4. 破坏循环请求等待

给资源统一编号，进程只能按编号顺序来请求资源。

## 8、死锁避免

在程序运行时避免发生死锁。

### 1. 安全状态

Has Max			Has Max			Has Max			Has Max			Has Max		
A	3	9	A	3	9	A	3	9	A	3	9	A	3	9
B	2	4	B	4	4	B	0	-	B	0	-	B	0	-
C	2	7	C	2	7	C	2	7	C	7	7	C	0	-
Free: 3			Free: 1			Free: 5			Free: 0			Free: 7		
(a)			(b)			(c)			(d)			(e)		

图 a 的第二列 Has 表示已拥有的资源数，第三列 Max 表示总共需要的资源数，Free 表示还有可以使用的资源数。从图 a 开始出发，先让 B 拥有所需的所有资源（图 b），运行结束后释放 B，此时 Free 变为 5（图 c）；接着以同样的方式运行 C 和 A，使得所有进程都能成功运行，因此可以称图 a 所示的状态时安全的。

定义：如果没有死锁发生，并且即使所有进程突然请求对资源的最大需求，也仍然存在某种调度次序能够使得每一个进程运行完毕，则称该状态是安全的。

安全状态的检测与死锁的检测类似，因为安全状态必须要求不能发生死锁。下面的银行家算法与死锁检测算法非常类似，可以结合着做参考对比。

### 2. 单个资源的银行家算法

一个小城镇的银行家，他向一群客户分别承诺了一定的贷款额度，算法要做的是判断对请求的满足是否会进入不安全状态，如果是，就拒绝请求；否则予以分配。

Has Max		
A	0	6
B	0	5
C	0	4
D	0	7
Free: 10		

(a)

Has Max		
A	1	6
B	1	5
C	2	4
D	4	7
Free: 2		

(b)

Has Max		
A	1	6
B	2	5
C	2	4
D	4	7
Free: 1		

(c)

上图 c 为不安全状态，因此算法会拒绝之前的请求，从而避免进入图 c 中的状态。

### 3. 多个资源的银行家算法

	Process	Tape drives	Plotters	Printers	Blu-rays
A	3	0	1	1	
B	0	1	0	0	
C	1	1	1	0	
D	1	1	0	1	
E	0	0	0	0	
Resources assigned					

	Process	Tape drives	Plotters	Printers	Blu-rays
A	1	1	0	0	
B	0	1	1	2	
C	3	1	0	0	
D	0	0	1	0	
E	2	1	1	0	
Resources still assigned					

$E = (6342)$   
 $P = (5322)$   
 $A = (1020)$

上图中有五个进程，四个资源。左边的图表示已经分配的资源，右边的图表示还需要分配的资源。最右边的 E、P 以及 A 分别表示：总资源、已分配资源以及可用资源，注意这三个为向量，而不是具体数值，例如  $A=(1020)$ ，表示 4 个资源分别还剩下 1/0/2/0。

#### 4. 检查一个状态是否安全的算法如下：

- 查找右边的矩阵是否存在一行小于等于向量 A。如果不存在这样的行，那么系统将会发生死锁，状态是不安全的。
- 假若找到这样一行，将该进程标记为终止，并将其已分配资源加到 A 中。
- 重复以上两步，直到所有进程都标记为终止，则状态是安全的。

如果一个状态不是安全的，需要拒绝进入这个状态。

## 25. 如何消除碎片文件

对于外部碎片，通过**紧凑技术**消除，就是操作系统不时地对进程进行移动和整理。但是这需要动态重定位寄存器地支持，且相对费时。紧凑地过程实际上类似于Windows系统中地磁盘整理程序，只不过后者是对外存空间地紧凑

解决外部内存碎片的问题就是**内存交换**。

可以把音乐程序占用的那 256MB 内存写到硬盘上，然后再从硬盘上读回到内存里。不过再读回的时候，我们不能装载回原来的位置，而是紧紧跟着那已经被占用了的 512MB 内存后面。这样就能空缺出

连续的 256MB 空间，于是新的 200MB 程序就可以装载进来。

回收内存时要尽可能地将相邻的空闲空间合并。

### 3、计算机网络

#### 1、OSI 的七层模型分别是？各自的功能是什么？

简要概括

- 物理层：底层数据传输，如网线；网卡标准。
- 数据链路层：定义数据的基本格式，如何传输，如何标识；如网卡MAC地址。
- 网络层：定义IP编址，定义路由功能；如不同设备的数据转发。
- 传输层：端到端传输数据的基本功能；如 TCP、UDP。
- 会话层：控制应用程序之间会话能力；如不同软件数据分发给不同软件。
- 表示层：数据格式标识，基本压缩加密功能。
- 应用层：各种应用软件，包括 Web 应用。

说明：

- 在四层，既传输层数据被称作**段**(Segments)； 三
- 层网络层数据被称做**包** (Packages) ；
- 二层数据链路层时数据被称为**帧** (Frames) ；
- 一层物理层时数据被称为**比特流** (Bits) 。

## 总结

- 网络七层模型是一个标准，而非实现。
- 网络四层模型是一个实现的应用模型。
- 网络四层模型由七层模型简化合并而来。

## 2、说一下一次完整的HTTP请求过程包括哪些内容？

### 第一种回答

- 建立起客户机和服务器连接。
- 建立连接后，客户机发送一个请求给服务器。
- 服务器收到请求给予响应信息。
- 客户端浏览器将返回的内容解析并呈现，断开连接。

### 第二种回答

域名解析 --> 发起TCP的3次握手 --> 建立TCP连接后发起http请求 --> 服务器响应http请求，浏览器得到html代码 --> 浏览器解析html代码，并请求html代码中的资源（如js、css、图片等） --> 浏览器对页面进行渲染呈现给用户。

## 3、HTTP长连接和短连接的区别

在HTTP/1.0中默认使用短连接。也就是说，客户端和服务端每进行一次HTTP操作，就建立一次连接，任务结束就中断连接。

而从HTTP/1.1起，默认使用长连接，用以保持连接特性。

## 4、什么是TCP粘包/拆包？发生的原因？

一个完整的业务可能会被TCP拆分成多个包进行发送，也有可能把多个小的包封装成一个大的数据包发送，这个就是TCP的拆包和粘包问题。

### 原因

- 1、应用程序写入数据的字节大小大于套接字发送缓冲区的大小。
- 2、进行MSS大小的TCP分段。（ $MSS = \text{TCP报文段长度} - \text{TCP首部长度}$ ）
- 3、以太网的payload大于MTU进行IP分片。（MTU指：一种通信协议的某一层上面所能通过的最大数据包大小。）

### 解决方案

- 1、消息定长。
- 2、在包尾部增加回车或者空格符等特殊字符进行分割
- 3、将消息分为消息头和消息尾。
- 4、使用其它复杂的协议，如RTMP协议等。



## 5、HTTP请求方法你知道多少？

客户端发送的 **请求报文** 第一行为请求行，包含了方法字段。

根据 HTTP 标准，HTTP 请求可以使用多种请求方法。

HTTP1.0 定义了三种请求方法：GET、POST 和 HEAD方法。

HTTP1.1 新增了六种请求方法：OPTIONS、PUT、PATCH、DELETE、TRACE 和 CONNECT 方法。

序号	方法	描述
1	GET	请求指定的页面信息，并返回实体主体。
2	HEAD	类似于 GET 请求，只不过返回的响应中没有具体的内容，用于获取报头
3	POST	向指定资源提交数据进行处理请求（例如提交表单或者上传文件）。数据被包含在请求体中。POST 请求可能会导致新的资源的建立和/或已有资源的修改。
4	PUT	从客户端向服务器传送的数据取代指定的文档的内容。
5	DELETE	请求服务器删除指定的页面。
6	CONNECT	HTTP/1.1 协议中预留给能够将连接改为管道方式的代理服务器。
7	OPTIONS	允许客户端查看服务器的性能。
8	TRACE	回显服务器收到的请求，主要用于测试或诊断。
9	PATCH	是对 PUT 方法的补充，用来对已知资源进行局部更新。

## 6、GET 和 POST 的区别，你知道哪些？

1. get是获取数据，post是修改数据
2. get把请求的数据放在url上，以?分割URL和传输数据，参数之间以&相连，所以get不太安全。而post把数据放在HTTP的包体内（request body）
3. get提交的数据最大是2k（限制实际上取决于浏览器），post理论上没有限制。

4. GET产生一个TCP数据包，浏览器会把http header和data一并发送出去，服务器响应200(返回数据); POST产生两个TCP数据包，浏览器先发送header，服务器响应100 continue，浏览器再发送data，服务器响应200 ok(返回数据)。
5. GET请求会被浏览器主动缓存，而POST不会，除非手动设置。
6. 本质区别：GET是幂等的，而POST不是幂等的

这里的幂等性：幂等性是指一次和多次请求某一个资源应该具有同样的副作用。简单来说意味着对同一URL的多个请求应该返回同样的结果。

正因为它们有这样的区别，所以不应该且**不能用get请求做数据的增删改这些有副作用的操作**。因为get请求是幂等的，**在网络不好的隧道中会尝试重试**。如果用get请求增数据，会有**重复操作**的风险，而这种重复操作可能会导致副作用（浏览器和操作系统并不知道你会用get请求去做增操作）。

## 7、一个 TCP 连接中 HTTP 请求发送可以一起发送么（比如一起发三个请求，再三个响应一起接收）？

HTTP/1.1 存在一个问题，单个 TCP 连接在同一时刻只能处理一个请求，意思是说：两个请求的生命周期不能重叠，任意两个 HTTP 请求从开始到结束的时间在同一个 TCP 连接里不能重叠。

在 HTTP/1.1 存在 Pipelining 技术可以完成这个多个请求同时发送，但是由于浏览器默认关闭，所以可以认为这是不可行的。在 HTTP2 中由于 Multiplexing 特点的存在，多个 HTTP 请求可以在同一个 TCP 连接中并行进行。

那么在 HTTP/1.1 时代，浏览器是如何提高页面加载效率的呢？主要有下面两点：

- 维持和服务器已经建立的 TCP 连接，在同一连接上顺序处理多个请求。
- 和服务器建立多个 TCP 连接。

## 8、在浏览器中输入url地址后显示主页的过程？

根据域名，进行DNS域名解析；

- 拿到解析的IP地址，建立TCP连接；
- 向IP地址，发送HTTP请求；
- 服务器处理请求；
- 返回响应结果；
- 关闭TCP连接；
- 浏览器解析HTML；
- 浏览器布局渲染；

## 9、在浏览器地址栏输入一个URL后回车，背后会进行哪些技术步骤？

### 第一种回答

- 1、查浏览器缓存，看看有没有已经缓存好的，如果没有
- 2、检查本机host文件，
- 3、调用API，Linux下Socket函数 `gethostbyname`
- 4、向DNS服务器发送DNS请求，查询本地DNS服务器，这其中用的是UDP的协议
- 6、如果在一个子网内采用ARP地址解析协议进行ARP查询如果不在一个子网那就需要对默认网关进行DNS查询，如果还找不到会一直向上找根DNS服务器，直到最终拿到IP地址(全球好像一共有13台根服务器)
- 7、这个时候我们就有了服务器的IP地址 以及默认的端口号了，http默认是80 https是 443 端口号，会，首先尝试http然后调用Socket建立TCP连接，
- 8、经过三次握手成功建立连接后，开始传送数据，如果正是http协议的话，就返回就完事了，
- 9、如果不是http协议，服务器会返回一个5开头的的重定向消息，告诉我们用的是https，那就是说IP没变，但是端口号从80变成443了，好了，再四次挥手，完事，
- 10、再来一遍，这次除了上述的端口号从80变成443之外，还会采用SSL的加密技术来保证传输数据的安全性，保证数据传输过程中不被修改或者替换之类的，
- 11、这次依然是三次握手，沟通好双方使用的认证算法，加密和检验算法，在此过程中也会检验对方的CA安全证书。
- 12、确认无误后，开始通信，然后服务器就会返回你所要访问的网址的一些数据，在此过程中会将界面进行渲染，牵涉到ajax技术之类的，直到最后我们看到色彩斑斓的网页

### 第二种回答

浏览器检查域名是否在缓存当中（要查看 Chrome 当中的缓存，打开 `chrome://net-internals/#dns`）。

如果缓存中没有，就去调用 `gethostbyname` 库函数（操作系统不同函数也不同）进行查询。

`gethostbyname` 函数在试图进行DNS解析之前首先检查域名是否在本地 `Hosts` 里，`Hosts` 的位置【不同的操作系统有所不同】

([https://en.wikipedia.org/wiki/Hosts\\_\(file\)#Location\\_in\\_the\\_file\\_system](https://en.wikipedia.org/wiki/Hosts_(file)#Location_in_the_file_system))

如果 `gethostbyname` 没有这个域名的缓存记录，也没有在 `hosts`` 里找到，它将会向 DNS 服务器发送一条 DNS 查询请求。DNS 服务器是由网络通信栈提供的，通常是本地路由器或者 ISP 的缓存 DNS 服务器。

查询本地 DNS 服务器

如果 DNS 服务器和我们的主机在同一个子网内，系统会按照下面的 ARP 过程对 DNS 服务器进行 ARP

查询

如果 DNS 服务器和我们的主机在不同的子网，系统会按照下面的 ARP 过程对默认网关进行查询

参考: <https://www.zhihu.com/question/34873227/answer/518086565>

## 10、HTTPS和HTTP的区别

- 1、HTTP协议传输的数据都是未加密的，也就是明文的，因此使用HTTP协议传输隐私信息非常不安全， HTTPS协议是由SSL+HTTP协议构建的可进行加密传输、身份认证的网络协议，要比http协议安全。
- 2、https协议需要到ca申请证书，一般免费证书较少，因而需要一定费用。
- 3、http和https使用的是完全不同的连接方式，用的端口也不一样，前者是80，后者是443。

## 11、HTTPS是如何保证数据传输的安全，整体的流程是什么？(SSL是怎么工作保证安全的)

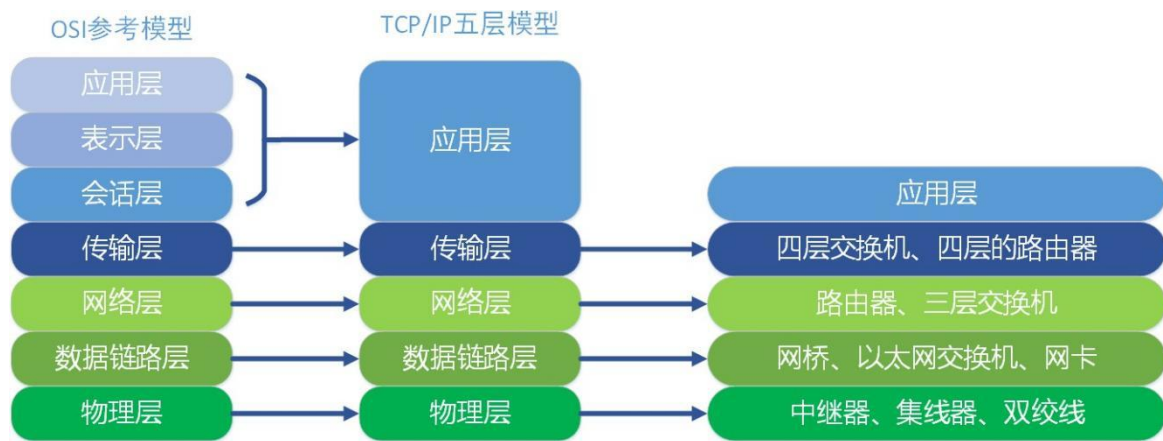
- (1) 客户端向服务器端发起SSL连接请求;
- (2) 服务器把公钥发送给客户端，并且服务器端保存着唯一的私钥
- (3) 客户端用公钥对双方通信的对称密钥进行加密，并发送给服务器端
- (4) 服务器利用自己唯一的私钥对客户端发来的对称密钥进行解密，
- (5) 进行数据传输，服务器和客户端双方用公有的相同的对称密钥对数据进行加密解密，可以保证在 数据收发过程中的安全，即是第三方获得数据包，也无法对其进行加密，解密和篡改。

因为数字签名、摘要证书防伪非常关键的武器。“摘要”就是对传输的内容，通过hash算法计算出一段固定长度的串。然后，在通过CA的私钥对这段摘要进行加密，加密后得到的结果就是“数字签名”

SSL/TLS协议的基本思路是采用公钥加密法，也就是说，客户端先向服务器端索要公钥，然后用公钥加密信息，服务器收到密文后，用自己的私钥解密。

## 12、网络的七层模型与各自的功能 (图片版)





### 13、端口有效范围是多少到多少？

0-1023为知名端口号，比如其中HTTP是80，FTP是20（数据端口）、21（控制端口）

UDP和TCP报头使用两个字节存放端口号，所以端口号的有效范围是从0到65535。动态端口的范围是从1024到65535

### 14、HTTP中有个缓存机制，但如何保证缓存是最新的呢？（缓存过期机制）

max-age 指令出现在请求报文，并且缓存资源的缓存时间小于该指令指定的时间，那么就能接受该缓存。

max-age 指令出现在响应报文，表示缓存资源在缓存服务器中保存的时间。

```
1 | Cache-Control: max-age=31536000
```

Expires 首部字段也可以用于告知缓存服务器该资源什么时候会过期。

```
1 | Expires: wed, 04 Jul 2012 08:26:05 GMT
```

- 在 HTTP/1.1 中，会优先处理 max-age 指令；
- 在 HTTP/1.0 中，max-age 指令会被忽略掉。

### 15、TCP头部中有哪些信息？

- 序号 (32bit) : 传输方向上字节流的字节编号。初始序号会被设置一个随机的初始值 (ISN) , 之后每次发送数据时, 序号值 = ISN + 数据在整个字节流中的偏移。假设A->B且ISN = 1024, 第一段数据512字节已经到B, 则第二段数据发送时序号为1024 + 512。用于解决网络包乱序问题。
- 确认号 (32bit) : 接收方对发送方TCP报文段的响应, 其值是收到的序号值 + 1。首
- 部长 (4bit) : 标识首部有多少个4字节 \* 首部长, 最大为15, 即60字节。
- 标志位 (6bit) :
  - URG: 标志紧急指针是否有效。
  - ACK: 标志确认号是否有效(确认报文段)。用于解决丢包问题。PSH:
  - 提示接收端立即从缓冲读走数据。
  - RST: 表示要求对方重新建立连接(复位报文段)。
  - SYN: 表示请求建立一个连接(连接报文段)。
  - FIN: 表示关闭连接(断开报文段)。
- 窗口 (16bit) : 接收窗口。用于告知对方(发送方)本方的缓冲还能接收多少字节数据。用于解决流控。
- 校验和 (16bit) : 接收端用CRC检验整个报文段有无损坏。

## 16、常见TCP的连接状态有哪些？

- CLOSED: 初始状态。
- LISTEN: 服务器处于监听状态。
- SYN\_SEND: 客户端socket执行CONNECT连接, 发送SYN包, 进入此状态。
- SYN\_RECV: 服务端收到SYN包并发送服务端SYN包, 进入此状态。
- ESTABLISH: 表示连接建立。客户端发送了最后一个ACK包后进入此状态, 服务端接收到ACK包后进入此状态。
- FIN\_WAIT\_1: 终止连接的一方(通常是客户机)发送了FIN报文后进入。等待对方FIN。
- CLOSE\_WAIT: (假设服务器)接收到客户机FIN包之后等待关闭的阶段。在接收到对方的FIN包之后, 自然是需要立即回复ACK包的, 表示已经知道断开请求。但是本方是否立即断开连接(发送 FIN包)取决于是否还有数据需要发送给客户端, 若有, 则在发送FIN包之前均为此状态。FIN\_WAIT\_2: 此时是半连接状态, 即有一方要求关闭连接, 等待另一方关闭。客户端接收到服务器的ACK包, 但并没有立即接收到服务端的FIN包, 进入FIN\_WAIT\_2状态。LAST\_ACK: 服务端发动最后的FIN包, 等待最后的客户端ACK响应, 进入此状态。TIME\_WAIT: 客户端收到服务端的FIN包, 并立即发出ACK包做最后的确认, 在此之后的2MSL时间称为TIME\_WAIT状态。

## 17、TCP头部报文字段介绍几个？各自的功能？

source port 和 destination port

两者分别为「源端口号」和「目的端口号」。源端口号就是指本地端口, 目的端口就是远程端口。

可以这么理解, 我们有很多软件, 每个软件都对应一个端口, 假如, 你想和我数据交互, 咱们得互相知道你我的端口号。

再来一个很官方的:

扩展: 应用程序的端口号和应用程序所在主机的 IP 地址统称为 socket (套接字), IP:端口号, 在互联网上 socket 唯一标识每一个应用程序, 源端口+源IP+目的端口+目的IP称为“套接字对”, 一对套接字就是一个连接, 一个客户端与服务器之间的连接。

Sequence Number

称为「序列号」。用于 TCP 通信过程中某一传输方向上字节流的每个字节的编号, 为了确保数据通信的有序性, 避免网络中乱序的问题。接收端根据这个编号进行确认, 保证分割的数据段在原



始数据包的位置。初始序列号由自己定，而后绪的序列号由对端的 ACK 决定： $SN_x = ACK_y(x \text{ 的序列号} = y \text{ 发给 } x \text{ 的 ACK})$ 。

说白了，类似于身份证一样，而且还得发送此时此刻的所在的位置，就相当于身份证上的地址一样。

#### Acknowledge Number

称为「确认序列号」。确认序列号是接收确认端所期望收到的下一序列号。确认序号应当是上次已成功收到数据字节序号加1，只有当标志位中的 ACK 标志为 1 时该确认序列号的字段才有效。主要用来解决不丢包的问题。

## TCP Flag

TCP 首部中有 6 个标志比特，它们中的多个可同时被设置为 1，主要是用于操控 TCP 的状态机的，依次为 URG, ACK, PSH, RST, SYN, FIN。

当然只介绍三个：

1. **ACK**：这个标识可以理解为发送端发送数据到接收端，发送的时候 ACK 为 0，标识接收端还未应答，一旦接收端接收数据之后，就将 ACK 置为 1，发送端接收到之后，就知道了接收端已经接收了数据。
2. **SYN**：表示「同步序列号」，是 TCP 握手的发送的第一个数据包。用来建立 TCP 的连接。SYN 标志位和 ACK 标志位搭配使用，当连接请求的时候，SYN=1，ACK=0 连接被响应的时候，SYN=1，ACK=1；这个标志的数据包经常被用来进行端口扫描。扫描者发送一个只有 SYN 的数据包，如果对方主机响应了一个数据包回来，就表明这台主机存在这个端口。
3. **FIN**：表示发送端已经达到数据末尾，也就是说双方的数据传送完成，没有数据可以传送了，发送 FIN 标志位的 TCP 数据包后，连接将被断开。这个标志的数据包也经常被用于进行端口扫描。发送端只剩最后的一段数据了，同时要告诉接收端后边没有数据可以接受了，所以用 FIN 标识一下，接收端看到这个 FIN 之后，哦！这是接受的最后的数据，接受完就关闭了；**TCP 四次分手必然问。**

## Window size

称为滑动窗口大小。所说的滑动窗口，用来进行流量控制。

## 18、应用层常见协议知道多少？了解几个？



协议	名称	默认端口	底层协议
HTTP	超文本传输协议	80	TCP
HTTPS	超文本传输安全协议	443	TCP
Telnet	远程登录服务的标准协议	23	TCP
FTP	文件传输协议	20传输和21连接	TCP
TFTP	简单文件传输协议	21	UDP
SMTP	简单邮件传输协议（发送用）	25	TCP
POP	邮局协议（接收用）	110	TCP
DNS	域名解析服务	53	服务器间进行域传输的时候用 TCP 客户端查询DNS服务器时用 UDP

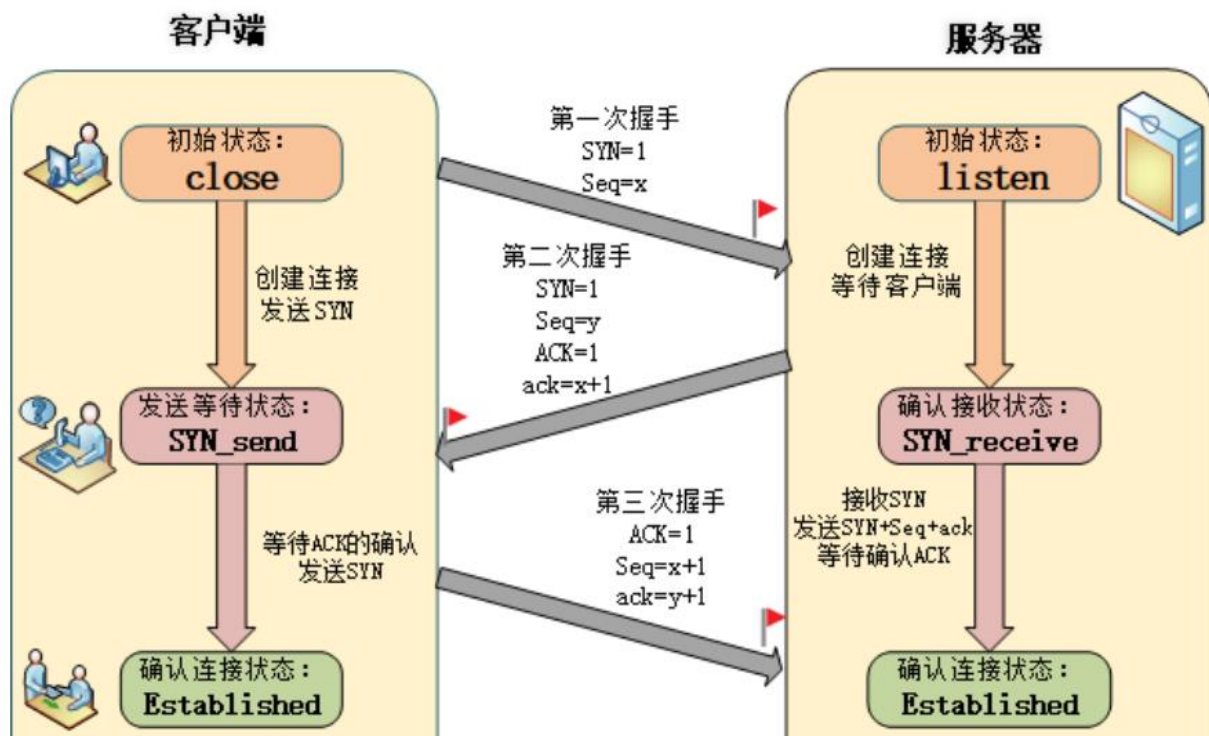
#### 54、浏览器在与服务器建立了一个 TCP 连接后是否会在一个 HTTP 请求完成后断开？什么情况下会断开？

在 HTTP/1.0 中，一个服务器在发送完一个 HTTP 响应后，会断开 TCP 链接。但是这样每次请求都会重新建立和断开 TCP 连接，代价过大。所以虽然标准中没有设定，**某些服务器对 Connection: keep-alive 的 Header 进行了支持**。意思是说，完成这个 HTTP 请求之后，不要断开 HTTP 请求使用的 TCP 连接。这样的好处是连接可以被重新使用，之后发送 HTTP 请求的时候不需要重新建立 TCP 连接，以及如果维持连接，那么 SSL 的开销也可以避免。

**持久连接：**既然维持 TCP 连接好处这么多，HTTP/1.1 就把 Connection 头写进标准，并且默认开启持久连接，除非请求中写明 Connection: close，那么浏览器和服务器之间是会维持一段时间的 TCP 连接，不会一个请求结束就断掉。

默认情况下建立 TCP 连接不会断开，只有在请求报头中声明 Connection: close 才会在请求完成后关闭连接。

#### 19、三次握手相关内容



三次握手 (Three-way Handshake) 其实就是指建立一个TCP连接时，需要客户端和服务端总共发送3个包。进行三次握手的主要作用就是为了确认双方的接收能力和发送能力是否正常、指定自己的初始化序列号为后面的可靠性传送做准备。实质上其实就是连接服务器指定端口，建立TCP连接，并同步连接双方的序列号和确认号，交换 TCP窗口大小 信息。

### 第一种回答

刚开始客户端处于 Closed 的状态，服务端处于 Listen 状态，进行三次握手：

- 第一次握手：客户端给服务端发一个 SYN 报文，并指明客户端的初始化序列号 ISN(c)。此时客户端处于 SYN\_SEND 状态。

首部的同步位SYN=1，初始序号seq=x，SYN=1的报文段不能携带数据，但要消耗掉一个序号。

- 第二次握手：服务器收到客户端的 SYN 报文之后，会以自己的 SYN 报文作为应答，并且也是指定了自己的初始化序列号 ISN(s)。同时会把客户端的 ISN+1 作为ACK 的值，表示自己已经收到了客户端的 SYN，此时服务器处于 SYN\_RCVD 的状态。

在确认报文段中SYN=1，ACK=1，确认号ack=x+1，初始序号seq=y。

- 第三次握手：客户端收到 SYN 报文之后，会发送一个 ACK 报文，当然，也是一样把服务器的 ISN+1 作为 ACK 的值，表示已经收到了服务端的 SYN 报文，此时客户端处于 ESTABLISHED 状态。服务器收到 ACK 报文之后，也处于 ESTABLISHED 状态，此时，双方已建立起了连接。

确认报文段ACK=1，确认号ack=y+1，序号seq=x+1（初始为seq=x，第二个报文段所以要+1），ACK报文段可以携带数据，不携带数据则不消耗序号。

发送第一个SYN的一端将执行主动打开 (active open)，接收这个SYN并发回下一个SYN的另一端执行被动打开 (passive open)。

在socket编程中，客户端执行connect()时，将触发三次握手。

### 第二种回答

- **初始状态**：客户端处于 `closed`(关闭) 状态，服务器处于 `listen`(监听) 状态。
- **第一次握手**：客户端发送请求报文将 `SYN=1` 同步序列号和初始化序列号 `seq=x` 发送给服务端，发送完之后客户端处于 `SYN_Send` 状态。（验证了客户端的发送能力和服务端的接收能力） **第二**
- **次握手**：服务端受到 `SYN` 请求报文之后，如果同意连接，会以自己的同步序列号 `SYN(服务端)=1`、初始化序列号 `seq=y` 和确认序列号（期望下次收到的数据包） `ack=x+1` 以及确认号 `ACK=1` 报文作为应答，服务器为 `SYN_Receive` 状态。（问题来了，两次握手之后，站在客户端角度上思考：我发送和接收都ok，服务端的发送和接收也都ok。但是站在服务端的角度思考：哎呀，我服务端接收ok，但是我不清楚我的发送ok不ok呀，而且我还不知道你接受能力如何呢？所以老哥，你需要给我三次握手来传个话告诉我一声。你要是不告诉我，万一我认为你跑了，然后我可能出于安全性的考虑继续给你发一次，看看你回不回我。）
- **第三次握手**：客户端接收到服务端的 `SYN+ACK` 之后，知道可以下次可以发送了下一序列的数据包了，然后发送同步序列号 `ack=y+1` 和数据包的序列号 `seq=x+1` 以及确认号 `ACK=1` 确认包作为应答，客户端转为 `established` 状态。（分别站在双方的角度上思考，各自ok）

## 20、为什么需要三次握手，两次不行吗？

弄清这个问题，我们需要先弄明白三次握手的目的是什么，能不能只用两次握手来达到同样的目的。

- 第一次握手：客户端发送网络包，服务端收到了。这样服务端就能得出结论：客户端的发送能力、服务端的接收能力是正常的。
- 第二次握手：服务端发包，客户端收到了。这样客户端就能得出结论：服务端的接收、发送能力，客户端的接收、发送能力是正常的。不过此时服务器并不能确认客户端的接收能力是否正常。
- 第三次握手：客户端发包，服务端收到了。这样服务端就能得出结论：客户端的接收、发送能力正常，服务器自己的发送、接收能力也正常。

因此，需要三次握手才能确认双方的接收与发送能力是否正常。

试想如果是用两次握手，则会出现下面这种情况：

如客户端发出连接请求，但因连接请求报文丢失而未收到确认，于是客户端再重传一次连接请求。后来收到了确认，建立了连接。数据传输完毕后，就释放了连接，客户端共发出了两个连接请求报文段，其中第一个丢失，第二个到达了服务端，但是第一个丢失的报文段只是在**某些网络结点长时间滞留了，延误到连接释放以后的某个时间才到达服务端**，此时服务端误认为客户端又发出一次新的连接请求，于是就向客户端发出确认报文段，同意建立连接，不采用三次握手，只要服务端发出确认，就建立新的连接了，此时客户端忽略服务端发来的确认，也不发送数据，则服务端一致等待客户端发送数据，浪费资源。

## 21、什么是半连接队列？

服务器第一次收到客户端的 `SYN` 之后，就会处于 `SYN_RCVD` 状态，此时双方还没有完全建立其连接，服务器会把此种状态下请求连接放在一个**队列**里，我们把这种队列称之为**半连接队列**。

当然还有一个**全连接队列**，就是已经完成三次握手，建立起连接的就会放在全连接队列中。如果队列满了就有可能会出现丢包现象。

这里在补充一点关于**SYN-ACK 重传次数**的问题：服务器发送完 `SYN-ACK` 包，如果未收到客户确认包，服务器进行首次重传，等待一段时间仍未收到客户确认包，进行第二次重传。如果重传次数超过系统规定的最大重传次数，系统将该连接信息从半连接队列中删除。注意，每次重传等待的时间不一定相同，一般会是指数增长，例如间隔时间为 1s, 2s, 4s, 8s.....

## 22、SYN攻击是什么？

服务器端的资源分配是在二次握手时分配的，而客户端的资源是在完成三次握手时分配的，所以服务器

容易受到SYN洪泛攻击。SYN攻击就是Client在短时间内伪造大量不存在的IP地址，并向Server不断地发送SYN包，Server则回复确认包，并等待Client确认，由于源地址不存在，因此Server需要不断重发直至超时，这些伪造的SYN包将长时间占用未连接队列，导致正常的SYN请求因为队列满而被丢弃，从而引起网络拥塞甚至系统瘫痪。SYN 攻击是一种典型的 DoS/DDoS 攻击。

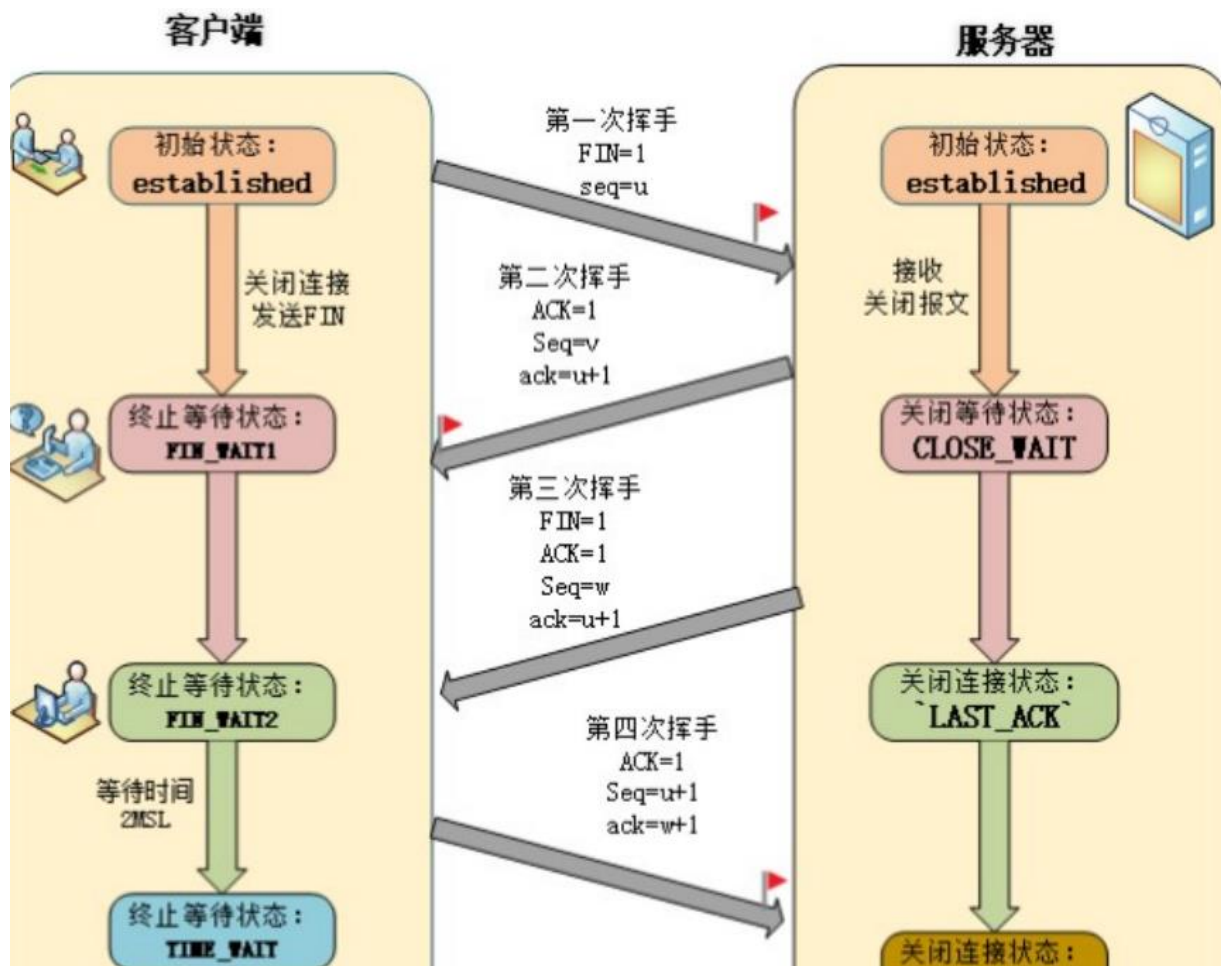
检测 SYN 攻击非常的方便，当你在服务器上看到大量的半连接状态时，特别是源IP地址是随机的，基本上可以断定这是一次SYN攻击。在 Linux/Unix 上可以使用系统自带的 netstats 命令来检测 SYN 攻击。

```
1 netstat -n -p TCP | grep SYN_RECV
2 复制代码
```

常见的防御 SYN 攻击的方法有如下几种：

- 缩短超时（SYN Timeout）时间
- 增加最大半连接数
- 过滤网关防护 SYN cookies 技术

## 22、四次挥手相关内容



建立一个连接需要三次握手，而终止一个连接要经过四次挥手（也有将四次挥手叫做四次握手的）。这由TCP的**半关闭**（half-close）造成的。所谓的半关闭，其实就是TCP提供了连接的一端在结束它的发送后还能接收来自另一端数据的能力。

TCP 的连接的拆除需要发送四个包，因此称为四次挥手(Four-way handshake)，客户端或服务器均可主动发起挥手动作。

### 第一种回答

刚开始双方都处于 ESTABLISHED 状态，假如是客户端先发起关闭请求。四次挥手的过程如下：

- 第一次挥手：客户端发送一个 FIN 报文，报文中会指定一个序列号。此时客户端处于 **FIN\_WAIT1** 状态。即发出**连接释放报文段**（FIN=1，序号seq=u），并停止再发送数据，主动关闭TCP连接，进入FIN\_WAIT1（终止等待1）状态，等待服务端的确认。
- 第二次挥手：服务端收到 FIN 之后，会发送 ACK 报文，且把客户端的序列号值 +1 作为 ACK 报文的序列号值，表明已经收到客户端的报文了，此时服务端处于 **CLOSE\_WAIT** 状态。即服务端收到连接释放报文段后即发出**确认报文段**（ACK=1，确认号ack=u+1，序号seq=v），服务端进入 CLOSE\_WAIT（关闭等待）状态，此时的TCP处于半关闭状态，客户端到服务端的连接释放。客户端收到服务端的确认后，进入FIN\_WAIT2(终止等待2)状态，等待服务端发出的连接释放报文段。
- 第三次挥手：如果服务端也想断开连接了，和客户端的第一次挥手一样，发给 FIN 报文，且指定一个序列号。此时服务端处于 **LAST\_ACK** 的状态。即服务端没有要向客户端发出的数据，服务端发出**连接释放报文段**（FIN=1，ACK=1，序号seq=w，确认号ack=u+1），服务端进入 LAST\_ACK（最后确认）状态，等待客户端的确认。
- 第四次挥手：客户端收到 FIN 之后，一样发送一个 ACK 报文作为应答，且把服务端的序列号值 +1 作为自己 ACK 报文的序列号值，此时客户端处于 **TIME\_WAIT** 状态。需要过一阵子以确保服务端



收到自己的 ACK 报文之后才会进入 CLOSED 状态，服务端收到 ACK 报文之后，就处于关闭连接了，处于 CLOSED 状态。即客户端收到服务端的连接释放报文段后，对此发出**确认报文段** (ACK=1, seq=u+1, ack=w+1)，客户端进入TIME\_WAIT (时间等待) 状态。此时TCP未释放掉，需要经过时间等待计时器设置的时间2MSL后，客户端才进入CLOSED状态。

收到一个FIN只意味着在这一方向上没有数据流动。**客户端执行主动关闭并进入TIME\_WAIT是正常的，服务端通常执行被动关闭，不会进入TIME\_WAIT状态。**

在socket编程中，任何一方执行close()操作即可产生挥手操作。

## 第二种回答

- **初始化状态**：客户端和服务端都在连接状态，接下来开始进行四次分手断开连接操作。
- **第一次分手**：第一次分手无论是客户端还是服务端都可以发起，因为 TCP 是全双工的。

假如客户端发送的数据已经发送完毕，发送FIN = 1 **告诉服务端，客户端所有数据已经全发完了，服务端你可以关闭接收了**，但是如果你们服务端有数据要发给客户端，客户端照样可以接收的。此时客户端处于FIN = 1等待服务端确认释放连接状态。

- **第二次分手**：服务端接收到客户端的释放请求连接之后，**知道客户端没有数据要发给自己了，然后服务端发送ACK = 1告诉客户端收到你发给我的信息**，此时服务端处于 CLOSE\_WAIT 等待关闭状态。(服务端先回应给客户端一声，我知道了，但服务端的发送数据能力即将等待关闭，于是接下来第三次就来了。)
- **第三次分手**：此时服务端向客户端把所有的数据发送完了，然后发送一个FIN = 1，**用于告诉客户端，服务端的所有数据发送完毕，客户端你也可以关闭接收数据连接了**。此时服务端状态处于 LAST\_ACK状态，来等待确认客户端是否收到了自己的请求。(服务端等客户端回复是否收到呢，不收到的话，服务端不知道客户端是不是挂掉了还是咋回事呢，所以服务端不敢关闭自己的接收能力，于是第四次就来了。)
- **第四次分手**：此时如果客户端收到了服务端发送完的信息之后，就发送ACK = 1，告诉服务端，客户端已经收到了你的信息。**有一个 2 MSL 的延迟等待。**

## 23、挥手为什么需要四次？

### 第一种回答

因为当服务端收到客户端的SYN连接请求报文后，可以直接发送SYN+ACK报文。其中**ACK报文是用来应答的，SYN报文是用来同步的**。但是关闭连接时，当服务端收到FIN报文时，很可能并不会立即关闭SOCKET，所以只能先回复一个ACK报文，告诉客户端，"你发的FIN报文我收到了"。只有等到我服务端所有的报文都发送完了，我才能发送FIN报文，因此不能一起发送。故需要四次挥手。

### 第二种回答

任何一方都可以在数据传送结束后发出连接释放的通知，待对方确认后进入半关闭状态。当另一方也没有数据再发送的时候，则发出连接释放通知，对方确认后就完全关闭了TCP连接。举个例子：A 和 B 打电话，通话即将结束后，A 说"我没啥要说的了"，B回答"我知道了"，但是 B 可能还会有要说的话，A 不能要求 B 跟着自己的节奏结束通话，于是 B 可能又巴拉巴拉说了一通，最后 B 说"我说完了"，A 回答

"知道了"，这样通话才算结束。

## 24、2MSL等待状态？

TIME\_WAIT状态也成为2MSL等待状态。每个具体TCP实现必须选择一个报文段最大生存时间MSL (Maximum Segment Lifetime)，它是任何报文段被丢弃前在网络内的最长时间。这个时间是有限的，因为TCP报文段以IP数据报在网络内传输，而IP数据报则有限制其生存时间的TTL字段。

对一个具体实现所给定的MSL值，处理的原则是：当TCP执行一个主动关闭，并发回最后一个ACK，该连接必须在TIME\_WAIT状态停留的时间为2倍的MSL。这样可让TCP再次发送最后的ACK以防这个ACK丢失（另一端超时并重发最后的FIN）。

这种2MSL等待的另一个结果是这个TCP连接在2MSL等待期间，定义这个连接的插口（客户的IP地址和端口号，服务器的IP地址和端口号）不能再被使用。这个连接只能在2MSL结束后才能再被使用。

## 25、四次挥手释放连接时，等待2MSL的意义？

MSL是Maximum Segment Lifetime的英文缩写，可译为“最长报文段寿命”，它是任何报文在网络中存在的最长时间，超过这个时间报文将被丢弃。

为了保证客户端发送的最后一个ACK报文段能够到达服务器。因为这个ACK有可能丢失，从而导致处在LAST-ACK状态的服务器收不到对FIN-ACK的确认报文。服务器会超时重传这个FIN-ACK，接着客户端再重传一次确认，重新启动时间等待计时器。最后客户端和服务器都能正常的关闭。假设客户端不等待2MSL，而是在发送完ACK之后直接释放关闭，一但这个ACK丢失的话，服务器就无法正常的进入关闭连接状态。

### 两个理由

1. 保证客户端发送的最后一个ACK报文段能够到达服务端。这个ACK报文段有可能丢失，使得处于LAST-ACK状态的B收不到对已发送的FIN+ACK报文段的确认，服务端超时重传FIN+ACK报文段，而客户端能在2MSL时间内收到这个重传的FIN+ACK报文段，接着客户端重传一次确认，重新启动2MSL计时器，最后客户端和服务端都进入到CLOSED状态，若客户端在TIME-WAIT状态不等待一段时间，而是发送完ACK报文段后立即释放连接，则无法收到服务端重传的FIN+ACK报文段，所以不会再发送一次确认报文段，则服务端无法正常进入到CLOSED状态。
2. 防止“已失效的连接请求报文段”出现在本连接中。客户端在发送完最后一个ACK报文段后，再经过2MSL，就可以使本连接持续的时间内所产生的所有报文段都从网络中消失，使下一个新的连接中不会出现这种旧的连接请求报文段。

## 26、为什么TIME\_WAIT状态需要经过2MSL才能返回到CLOSE状态？

### 第一种回答

理论上，四个报文都发送完毕，就可以直接进入CLOSE状态了，但是可能网络是不可靠的，有可能最后一个ACK丢失。所以TIME\_WAIT状态就是用来重发可能丢失的ACK报文。

### 第二种回答

对应这样一种情况，最后客户端发送的ACK = 1给服务端的过程中丢失了，服务端没收到，服务端怎么认为的？我已经发送完数据了，怎么客户端没回应我？是不是中途丢失了？然后服务端再次发起断开连接的请求，一个来回就是2MSL。

客户端给服务端发送的ACK = 1丢失，服务端等待 1MSL没收到，然后重新发送消息需要1MSL。如果再次接收到服务端的消息，则重启2MSL计时器，发送确认请求。客户端只需等待2MSL，如果没有再次收到服务端的消息，就说明服务端已经接收到自己确认消息；此时双方都关闭的连接，TCP 四次分手完毕

## 27、TCP粘包问题是什么？你会如何去解决它？

TCP粘包是指发送方发送的若干包数据到接收方接收时粘成一包，从接收缓冲区看，后一包数据的头紧接着前一包数据的尾。

- 由TCP连接复用造成的粘包问题。

- 因为TCP默认会使用**Nagle算法**，此算法会导致粘包问题。
  - 只有上一个分组得到确认，才会发送下一个分组；
  - 收集多个小分组，在一个确认到来时一起发送。
- **数据包过大**造成的粘包问题。
- 流量控制，**拥塞控制**也可能导致粘包。

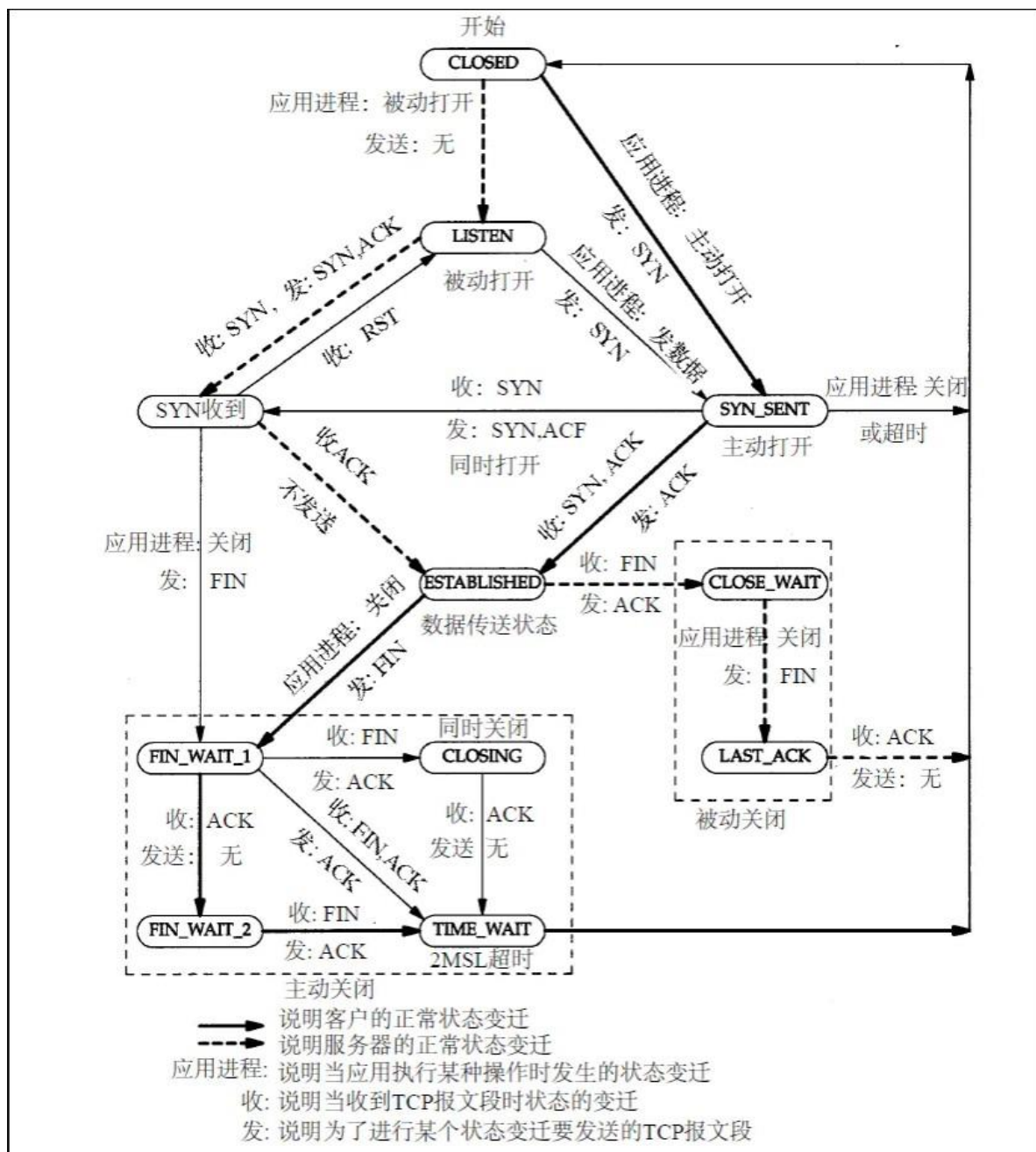
接收方不及时接收缓冲区的包，造成多个包接收解决：

1. **Nagle算法**问题导致的，需要结合应用场景适当关闭该算法
2. 尾部标记序列。通过特殊标识符表示数据包的边界，例如\n\r, \t, 或者一些隐藏字符。
3. 头部标记分步接收。在TCP报文的头部加上表示数据长度。
4. 应用层发送数据时**定长**发送。

## 28、三次握手四次挥手的变迁图

《TCP/IP详解 卷1:协议》有一张TCP状态变迁图，很具有代表性，有助于大家理解三次握手和四次挥手的状态变化。如下图所示，粗的实线箭头表示正常的客户端状态变迁，粗的虚线箭头表示正常的服务器状态变迁。





## 29、TCP四大拥塞控制算法总结？（极其重要）

### 四大算法

拥塞控制主要是四个算法：1) 慢启动，2) 拥塞避免，3) 拥塞发生，4) 快速恢复。这四个算法不是一天都搞出来的，这个四算法的发展经历了很多时间，到今天都还在优化中。

### 慢热启动算法 – Slow Start

所谓慢启动，也就是TCP连接刚建立，一点一点地提速，试探一下网络的承受能力，以免直接扰乱了网络通道的秩序。

慢启动算法：

- 1) 连接建好的开始先初始化拥塞窗口cwnd大小为1，表明可以传一个MSS大小的数据。
- 2) 每当收到一个ACK，cwnd大小加一，呈线性上升。
- 3) 每当过了一个往返延迟时间RTT(Round-Trip Time)，cwnd大小直接翻倍，乘以2，呈指数让升。
- 4) 还有一个sssthresh (slow start threshold)，是一个上限，当cwnd >= sssthresh时，就会进入“拥塞避免算法”（后面会说这个算法）

## 拥塞避免算法 - Congestion Avoidance

如同前边说的，当拥塞窗口大小cwnd大于等于慢启动阈值sssthresh后，就进入拥塞避免算法。算法如下：

- 1) 收到一个ACK，则 $cwnd = cwnd + 1 / cwnd$
- 2) 每当过了一个往返延迟时间RTT，cwnd大小加一。

过了慢启动阈值后，拥塞避免算法可以避免窗口增长过快导致窗口拥塞，而是缓慢的增加调整到网络的最佳值。

## 拥塞发生状态时的算法

一般来说，TCP拥塞控制默认认为网络丢包是由于网络拥塞导致的，所以一般的TCP拥塞控制算法以丢包为网络进入拥塞状态的信号。对于丢包有两种判定方式，一种是超时重传RTO[Retransmission Timeout]超时，另一个是收到三个重复确认ACK。

超时重传是TCP协议保证数据可靠性的一个重要机制，其原理是在发送一个数据以后就开启一个计时器，在一定时间内如果没有得到发送数据报的ACK报文，那么就重新发送数据，直到发送成功为止。

但是如果发送端接收到3个以上的重复ACK，TCP就意识到数据发生丢失，需要重传。这个机制不需要等到重传定时器超时，所以叫

做快速重传，而快速重传后没有使用慢启动算法，而是拥塞避免算法，所以这又叫做快速恢复算法。

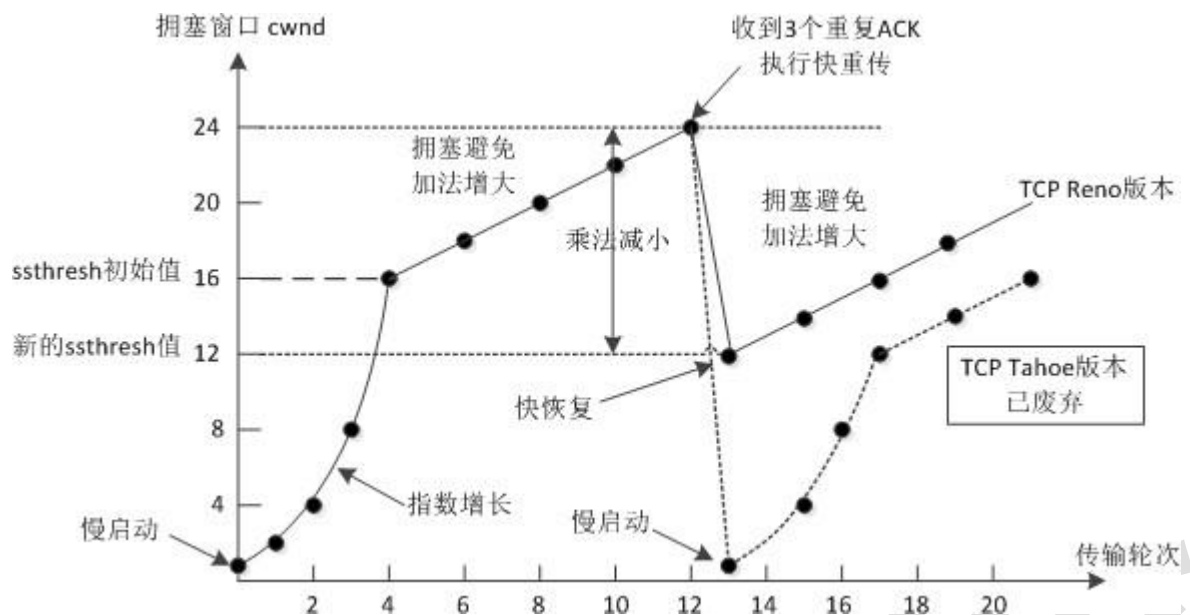
超时重传RTO[Retransmission Timeout]超时，TCP会重传数据包。TCP认为这种情况比较糟糕，反应也比较强烈：

- 由于发生丢包，将慢启动阈值sssthresh设置为当前cwnd的一半，即 $sssthresh = cwnd / 2$ 。
- cwnd重置为1
- 进入慢启动过程

最为早期的TCP Tahoe算法就只使用上述处理办法，但是由于一丢包就一切重来，导致cwnd又重置为1，十分不利于网络数据的稳定传递。

所以，TCP Reno算法进行了优化。当收到三个重复确认ACK时，TCP开启快速重传Fast Retransmit算法，而不用等到RTO超时再进行重传：

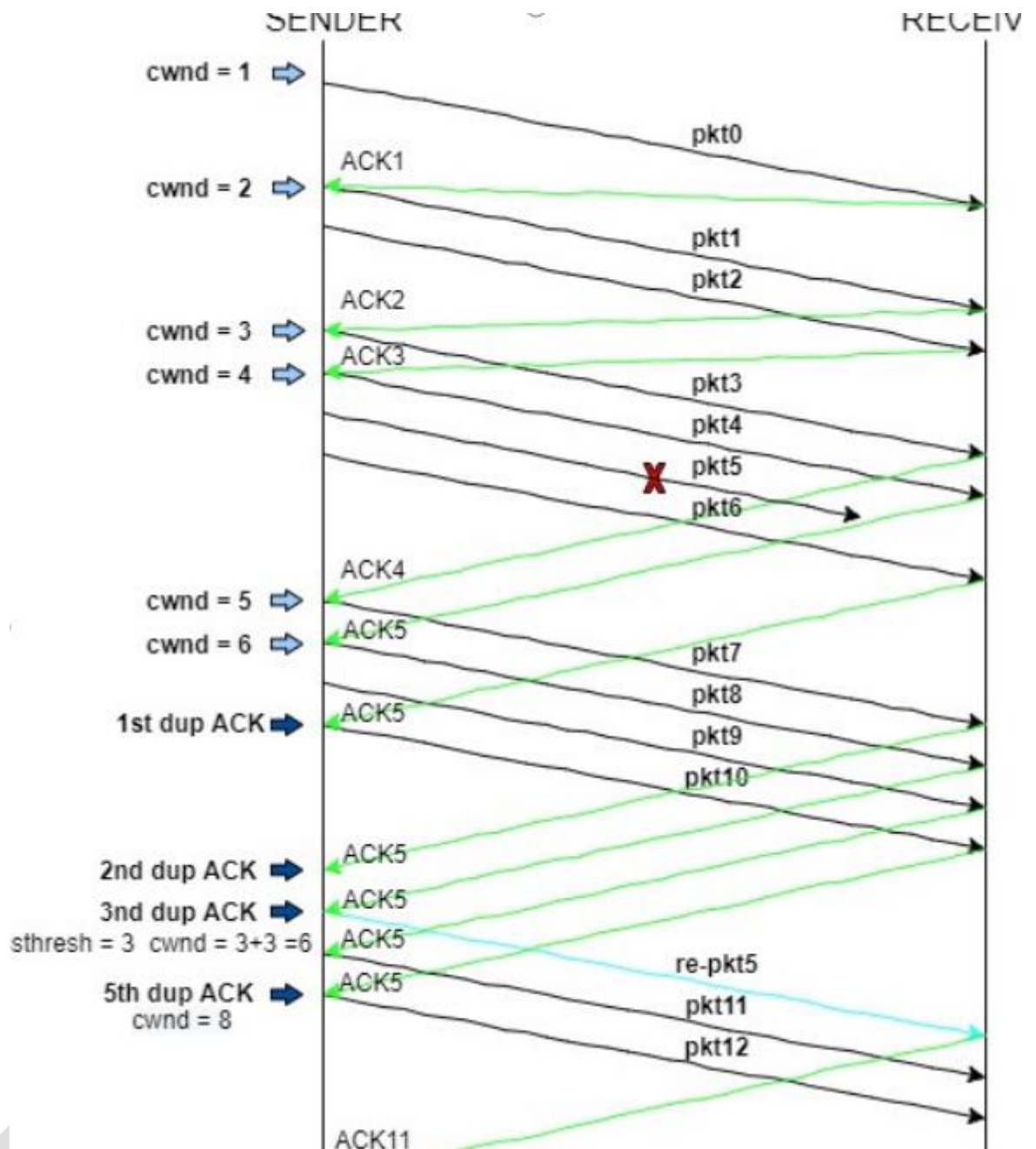
- cwnd大小缩小为当前的一半
- sssthresh设置为缩小后的cwnd大小
- 然后进入快速恢复算法Fast Recovery。



## 快速恢复算法 – Fast Recovery

TCP Tahoe是早期的算法，所以没有快速恢复算法，而Reno算法有。在进入快速恢复之前，cwnd和ssthresh已经被更改为原有cwnd的一半。快速恢复算法的逻辑如下：

- $cwnd = cwnd + 3 \text{ MSS}$ ，加3 MSS的原因是因为收到3个重复的ACK。
- 重传DACKs指定的数据包。
- 如果再收到DACKs，那么cwnd大小增加一。
- 如果收到新的ACK，表明重传的包成功了，那么退出快速恢复算法。将cwnd设置为ssthresh，然后进入拥塞避免算法。



如图所示，第五个包发生了丢失，所以导致接收方接收到三次重复ACK，也就是ACK5。所以将ssthresh设置当当时cwnd的一半，也就是 $6/2=3$ ，cwnd设置为 $3+3=6$ 。然后重传第五个包。当收到新的ACK时，也就是ACK11，则退出快速恢复阶段，将cwnd重新设置为当前的ssthresh，也就是3，然后进入拥塞避免算法阶段。

### 30、TCP和UDP的区别

1、TCP面向连接（如打电话要先拨号建立连接）；UDP是无连接的，即发送数据之前不需要建立连接

2、TCP提供可靠的服务。也就是说，通过TCP连接传送的数据，无差错，不丢失，不重复，且按序到达;UDP尽最大努力交付，即不保证可靠交付

3、TCP面向字节流，实际上是TCP把数据看成一连串无结构的字节流;UDP是面向报文的

UDP没有拥塞控制，因此网络出现拥塞不会使源主机的发送速率降低（对实时应用很有用，如IP电话，实时视频会议等）

4、每一条TCP连接只能是点到点的;UDP支持一对一，一对多，多对一和多对多的交互通信

5、TCP首部开销20字节;UDP的首部开销小，只有8个字节

6、TCP的逻辑通信信道是全双工的可靠信道，UDP则是不可靠信道

7、UDP是面向报文的，发送方的UDP对应用层交下来的报文，不合并，不拆分，只是在其上面加上首部后就交给了下面的网络层，论应用层交给UDP多长的报文，它统统发送，一次发送一个。而对接收方，接到后直接去除首部，交给上面的应用层就完成了任务了。因此，它需要应用层控制报文的大小

TCP是面向字节流的，它把上面应用层交下来的数据看成无结构的字节流会发送，可以想象成流水形式的，发送方TCP会将数据放入“蓄水池”（缓存区），等到可以发送的时候就发送，不能发送就等着TCP会根据当前网络的拥塞状态来确定每个报文段的大小。

《TCP数据段格式+UDP数据段格式详解》：<https://www.cnblogs.com/love-jelly-pig/p/8471181.html>

## 31、TCP对应的应用层协议

FTP：定义了文件传输协议，使用21端口。

Telnet：它是一种用于远程登陆的端口,23端口

SMTP：定义了简单邮件传送协议，服务器开放的是25号端口。

POP3：它是和SMTP对应，POP3用于接收邮件。

## 32、UDP对应的应用层协议

DNS：用于域名解析服务，用的是53号端口

SNMP：简单网络管理协议，使用161号端口

TFTP(Trivial File Transfer Protocol)：简单文件传输协议，69

## 33、数据链路层常见协议？可以说一下吗？

协议	名称	作用
ARP	地址解析协议	根据IP地址获取物理地址
RARP	反向地址转换协议	根据物理地址获取IP地址
PPP	点对点协议	主要是用来通过拨号或专线方式建立点对点连接发送数据，使其成为各种主机、网桥和路由器之间简单连接的一种共通的解决方案

## 34、Ping命令基于哪一层协议的原理是什么？

ping命令基于网络层的命令，是基于ICMP协议工作的。

## 35、在进行UDP编程的时候，一次发送多少bytes好？

当然,这个没有唯一答案，相对于不同的系统,不同的要求,其得到的答案是不一样的。

我这里仅对像ICQ一类的发送聊天消息的情况作分析，对于其他情况，你或许也能得到一点帮助:首先,我们知道,TCP/IP通常被认为是一个四层协议系统,包括链路层,网络层,运输层,应用层.UDP属于运输层,

下面我们由下至上一步一步来看:以太网(Ethernet)数据帧的长度必须在46-1500字节之间,这是由以太网的物理特性决定的.这个1500字节被称为链路层的MTU(最大传输单元).但这并不是指链路层的长度被限制在1500字节,其实这个MTU指的是链路层的数据区.并不包括链路层的首部和尾部的18个字节.

所以,事实上,这个1500字节就是网络层IP数据报的长度限制。因为IP数据报的首部为20字节,所以IP数据报的数据区长度最大为1480字节.而这个1480字节就是用来放TCP传来的TCP报文段或UDP传来的UDP数据报的.又因为UDP数据报的首部8字节,所以UDP数据报的数据区最大长度为1472字节.这个1472字节就是我们可以使用的字节数。

当我们发送的UDP数据大于1472的时候会怎样呢？

这也就是说IP数据报大于1500字节,大于MTU.这个时候发送方IP层就需要分片(fragmentation).

把数据报分成若干片,使每一片都小于MTU.而接收方IP层则需要进行数据报的重组.

这样就会多做许多事情,而更严重的是,由于UDP的特性,当某一片数据传送中丢失时,接收方便无法重组数据报.将导致丢弃整个UDP数据报。

因此,在普通的局域网环境下，我建议将UDP的数据控制在1472字节以下为好。

进行Internet编程时则不同,因为Internet上的路由器可能会将MTU设为不同的值.

如果我们假定MTU为1500来发送数据的,而途经的某个网络的MTU值小于1500字节,那么系统将会使用一系列的机制来调整MTU值,使数据报能够顺利到达目的地,这样就会做许多不必要的操作.

鉴于Internet上的标准MTU值为576字节,所以我建议在进行Internet的UDP编程时. 最好将UDP的数据长度控制在548字节(576-8-20)以内

## 4、设计模式

### 0 23种设计模式速记

设计模式的三大类

**创建型模式 (Creational Pattern)**：对类的实例化过程进行了抽象，能够将软件模块中**对象的创建**和对象的使用分离。

(5种) 工厂模式、抽象工厂模式、单例模式、建造者模式、原型模式

记忆口诀：创工原单建抽（创公园，但见愁）

**结构型模式 (Structural Pattern)**：关注于对象的组成以及对象之间的依赖关系，描述如何将类或者对象结合在一起形成更大的结构，就像**搭积木**，可以通过简单积木的组合形成复杂的、功能更为强大的结构。

(7种) 适配器模式、装饰者模式、代理模式、外观模式、桥接模式、组合模式、享元模式

记忆口诀：结享外组适代装桥（姐想外租，世代装桥）

**行为型模式 (Behavioral Pattern)**：关注于对象的行为问题，是对在不同的对象之间划分责任和算法的抽象化；不仅仅关注类和对象的结构，而且重点关注它们之间的**相互作用**。

(11种) 策略模式、模板方法模式、观察者模式、迭代器模式、责任链模式、命令模式、备忘录模式、状态模式、访问者模式、中介者模式、解释器模式

记忆口诀：行状责中模访解备观策命迭（模仿这种形状，观测戒备鸣笛儿）

### 23种设计模式的分类与特点概述

#### 1. 创建型设计模式 (Creational Patterns)：

这些模式关注对象的创建机制，旨在提供一种更灵活、更复杂的对象创建方式，而不是直接实例化对象。常见的创建型模式包括：

**单例模式 (Singleton Pattern)**：确保一个类只有一个实例，并提供全局访问点。



工厂方法模式 (Factory Method Pattern)： 定义一个创建对象的接口，但将对象的实际创建延迟到子类。

抽象工厂模式 (Abstract Factory Pattern)： 提供一个创建相关对象家族的接口，而无需指定具体类。

建造者模式 (Builder Pattern)： 将一个复杂对象的构建过程分解为多个步骤，以便更灵活地创建对象。

原型模式 (Prototype Pattern)： 通过复制现有对象来创建新对象，而不是从头开始构建。

## 2. 结构型设计模式 (Structural Patterns)：

这些模式关注如何组合类和对象以形成更大的结构，以解决系统中对象之间的关系。常见的结构型模式包括：

适配器模式 (Adapter Pattern)： 允许将一个接口转换为另一个客户端期望的接口。

装饰者模式 (Decorator Pattern)： 动态地为对象添加新的功能，而无需修改其源代码。

代理模式 (Proxy Pattern)： 提供一个代理对象以控制对其他对象的访问。

组合模式 (Composite Pattern)： 允许客户端以统一的方式处理单个对象和对象组合。

桥接模式 (Bridge Pattern)： 将抽象部分与其实现分离，以便它们可以独立变化。

外观设计模式 (Facade Design Pattern)： 外观模式提供了一个高级别的接口，将多个底层接口组合成一个更简单的接口，以供客户端使用。这有助于隔离客户端代码和复杂子系统之间的耦合关系。

享元模式 (Flyweight Pattern) 主要用于减少创建对象的数量，以减少内存占用和提高性能。这种类型的设计模式属于结构型模式，它提供了减少对象数量从而改善应用所需的对象结构的方式。

### 1 谈谈你对简单工厂设计模式的理解；

简单工厂设计模式 (Simple Factory Pattern) 是一种创建型设计模式，它提供了一个工厂类，用于根据客户端的请求创建不同类型的对象，而客户端无需直接与具体类相互耦合。这个工厂类拥有一个方法，该方法接受客户端的参数，并根据参数的不同来创建并返回相应的对象。

核心思想： 将对象的创建过程封装在一个独立的工厂类中，客户端只需要通过工厂类来创建所需的对象，而不需要直接实例化具体的类。

简单工厂的结构：

工厂类 (Factory)： 负责创建具体对象的类，通常包含一个或多个创建对象的方法。

产品类 (Product)： 具体对象的抽象，由工厂类来创建，客户端与产品类进行交互。

示例：

假设你正在开发一个图形绘制应用程序，需要根据用户的选择创建不同类型的图形对象，如圆形 (Circle) 和矩形 (Rectangle)。你可以使用简单工厂来实现这个场景。

```
// 产品类 - 图形
class Shape {
public:
    virtual void draw() = 0;
};

// 具体产品类 - 圆形
class Circle : public Shape {
public:
    void draw() override {
        std::cout << "绘制圆形" << std::endl;
    }
};
```



```

    }
};

// 具体产品类 - 矩形
class Rectangle : public Shape {
public:
    void draw() override {
        std::cout << "绘制矩形" << std::endl;
    }
};

// 工厂类
class ShapeFactory {
public:
    // 根据类型创建图形对象
    static Shape* createShape(std::string type) {
        if (type == "Circle") {
            return new Circle();
        }
        else if (type == "Rectangle") {
            return new Rectangle();
        }
        else {
            return nullptr; // 可以根据需要添加其他图形类型的创建逻辑
        }
    }
};

```

应用场景：

**图形绘制应用程序：** 如上面的示例，根据用户选择的不同图形类型创建相应的图形对象。

**日志记录器：** 根据不同的日志级别（信息、警告、错误）创建不同的日志记录器对象。

**数据库连接管理：** 根据数据库类型（MySQL、PostgreSQL、Oracle）创建数据库连接对象。

**UI库控件：** 根据用户需求创建不同类型的UI控件（按钮、文本框、下拉菜单等）。

简单工厂模式的优点在于**封装了对象的创建过程，客户端代码与具体类解耦，降低了代码的维护复杂度**。但缺点是**当需要添加新的产品类型时，需要修改工厂类的代码，违反了开闭原则**。因此，如果需要频繁添加新的产品类型，可能要考虑其他创建型设计模式，如工厂方法模式或抽象工厂模式。

## 2 单例设计模式的饿汉式和懒汉式的区别；

单例设计模式旨在确保一个类只有一个实例，并提供全局访问点以访问该实例。在单例模式中，有两种常见的实现方式，即饿汉式和懒汉式。它们的主要区别在于实例的创建时间和线程安全性。

```

class Singleton {
private:
    static Singleton* instance;

    Singleton() {} // 私有构造函数，禁止外部实例化

public:
    static Singleton* getInstance() {
        if (instance == nullptr) {
            instance = new Singleton();
        }
        return instance;
    }
};

```

在懒汉式中，实例在第一次被请求时创建。这意味着实例的创建是延迟的，直到需要使用它时才会创建。

## 1. 饿汉式单例模式：

在饿汉式中，实例在类加载时就被创建，不管是否需要使用它。这意味着无论何时引用该类，都会返回相同的实例。

### 主要特点：

**线程安全：**在多线程环境中，由于实例在类加载时创建，所以不会存在竞态条件，是线程安全的。

**简单：**实现简单，但可能会浪费内存，因为无论是否需要，实例都会创建。

## 2. 懒汉式单例模式：

### 主要特点：

**延迟加载：**实例的创建是延迟的，只有在需要时才会创建。

**非线程安全：**在多线程环境中，多个线程可能同时访问`getInstance()`方法，导致创建多个实例。需要额外的同步措施来保证线程安全。

### 注意事项：

在懒汉式中，需要考虑多线程环境下的线程安全问题。可以使用加锁或双重检查锁等方式来确保线程安全。饿汉式的主要优点是简单和线程安全，但可能浪费内存。懒汉式则具有延迟加载的特点，但需要处理线程安全性。选择哪种方式取决于具体需求和性能考虑。如果不需要延迟加载，并且希望简单的实现，可以选择饿汉式。如果需要延迟加载，并且能够处理线程安全问题，可以选择懒汉式。

## 3. 行为型设计模式 (Behavioral Patterns)：

这些模式关注对象之间的通信和职责分配，以帮助在系统中实现更松散耦合的对象之间的通信。常见的行为型模式包括：

**模板方法 (Template Method Pattern)：**定义一个算法的骨架，而将一些步骤的具体实现延迟到子类中。这种模式允许在不改变算法结构的情况下，通过子类重写或扩展某些步骤来定制算法的行为。

**观察者模式 (Observer Pattern)：**定义一种一对多的依赖关系，使得当一个对象状态发生变化时，所有依赖于它的对象都得到通知并自动更新。

**策略模式 (Strategy Pattern)：**定义一系列算法，将它们封装成对象，并使其可以互换使用，以便根据需要动态选择算法。

**命令模式 (Command Pattern)：**将请求封装成对象，以便可以参数化客户端对象，排队请求和记录日志，以及支持可撤销操作。

**状态模式 (State Pattern)：**允许对象在其内部状态发生变化时改变其行为。

**责任链模式 (Chain of Responsibility Pattern)：**将请求的发送者和接收者解耦，以便多个对象都有机会处理请求，将请求沿链传递，直到有一个对象处理它为止。

### 3 怎么实现懒汉式单例设计模式的线程安全？

双重检查锁定 (Double-Checked Locking) 是一种用于实现懒汉式单例模式的常见线程安全技术。它的目的是在多线程环境下延迟实例化对象，同时确保只有一个实例被创建。这种技术通常用于需要考虑性能的情况下，因为它在绝大多数情况下避免了不必要的锁定操作。

双重检查锁定的基本思想是：

首先检查实例是否已经创建，如果已经创建，则直接返回实例。

如果实例尚未创建，才进行锁定操作。

在锁定的情况下再次检查实例是否已经创建，以防止其他线程在等待锁定时已经创建了实例。

如果实例仍未创建，才创建并返回实例。

以下是一个使用双重检查锁定的懒汉式单例模式示例

```
class Singleton {
private:
    static Singleton* instance;
    static std::mutex mtx; // 用于线程安全的锁

    Singleton() {} // 私有构造函数，禁止外部实例化

public:
    static Singleton* getInstance() {
        // 第一次检查，如果实例已经存在，直接返回
        if (instance == nullptr) {
            std::lock_guard<std::mutex> lock(mtx); // 加锁
            // 第二次检查，防止在等待锁期间其他线程已经创建实例
            if (instance == nullptr) {
                instance = new Singleton(); // 创建实例
            }
        }
        return instance;
    }
};

// 静态成员初始化
Singleton* Singleton::instance = nullptr;
std::mutex Singleton::mtx;
```

双重检查锁定确保只有在实例未创建时才会加锁和创建实例，从而提高了性能。

需要注意的是，双重检查锁定需要在多线程环境下使用，因为它涉及到线程同步。另外，C++11之后，还可以使用更方便的局部静态变量来实现懒汉式单例模式，因为C++11引入了线程安全的局部静态变量初始化。

#### 4 聊聊你对抽象工厂设计模式的理解：

抽象工厂设计模式是一种创建型设计模式，它提供了一种创建一系列相关或相互依赖对象的接口，而无需指定其具体类。

抽象工厂设计模式通过定义一个抽象工厂接口，该接口声明了一组用于创建不同产品族的抽象方法。每个具体的工厂类实现这个接口，并负责创建特定的产品族。具体产品类实现了产品接口，每个工厂类创建的产品都属于同一个产品族，但是产品族之间有不同的实现。

抽象工厂模式的核心思想是通过创建一组相关的产品，而不是单个产品。它提供了一种封装对象创建的方式，使得客户端可以从具体实现中解耦。

下面是一个简单的例子来说明抽象工厂设计模式的实现：

```
#include <iostream>

// 抽象产品接口
class Button {
public:
    virtual void render() = 0;
};
```

```
// 具体产品类A
class WindowsButton : public Button {
public:
    void render() override {
        std::cout << "Rendering a Windows button." << std::endl;
    }
};

// 具体产品类B
class MacButton : public Button {
public:
    void render() override {
        std::cout << "Rendering a Mac button." << std::endl;
    }
};

// 抽象工厂接口
class GUIFactory {
public:
    virtual Button* createButton() = 0;
};

// 具体工厂类A
class WindowsFactory : public GUIFactory {
public:
    Button* createButton() override {
        return new WindowsButton();
    }
};

// 具体工厂类B
class MacFactory : public GUIFactory {
public:
    Button* createButton() override {
        return new MacButton();
    }
};

int main() {
    // 创建一个Windows风格的工厂
    GUIFactory* windowsFactory = new WindowsFactory();
    // 创建一个Windows风格的按钮
    Button* windowsButton = windowsFactory->createButton();
    // 渲染按钮
    windowsButton->render(); // Output: Rendering a Windows button.

    // 创建一个Mac风格的工厂
    GUIFactory* macFactory = new MacFactory();
    // 创建一个Mac风格的按钮
    Button* macButton = macFactory->createButton();
    // 渲染按钮
    macButton->render(); // Output: Rendering a Mac button.

    delete windowsFactory;
    delete windowsButton;
    delete macFactory;
    delete macButton;

    return 0;
}
```

在上述代码中定义了一个抽象产品接口 Button，它声明了一个抽象方法 render。然后我们创建了两个具体产品类 WindowsButton 和 MacButton，它们分别实现了 Button 接口。

接着，我们定义了一个抽象工厂接口 GUIFactory，它声明了一个抽象方法 createButton。然后我们创建了两个具体工厂类 WindowsFactory 和 MacFactory，它们分别实现了 GUIFactory 接口，并分别负责创建对应的产品。

在 main 函数中，我们首先创建了一个 WindowsFactory 工厂对象，然后通过该工厂对象创建了一个 WindowsButton 按钮对象，并调用其 render 方法来渲染按钮。接着，我们创建了一个 MacFactory 工厂对象，通过该工厂对象创建了一个 MacButton 按钮对象，并渲染按钮。

通过抽象工厂设计模式，我们可以通过工厂来创建一组相关的产品，而无需关心具体的产品类。这样可以实现一系列产品的一致性和互换性，同时也符合了开闭原则，使得系统更加灵活和可扩展。

场景举例：

**图形界面库**：图形界面库通常需要创建一组相关的界面元素，如按钮、文本框、标签等。通过抽象工厂模式，可以创建不同风格的界面元素，如Windows风格、Mac风格等，确保界面元素之间的风格一致性。

**数据库访问**：在数据库访问中，可能需要使用不同的数据库引擎，如MySQL、Oracle、SQL Server等。通过抽象工厂模式，可以创建不同数据库引擎的连接对象、命令对象等，以便在运行时根据需要切换使用不同的数据库引擎。

**操作系统适配**：在开发跨平台的软件时，可能需要根据不同的操作系统提供不同的实现。通过抽象工厂模式，可以根据操作系统的不同创建对应的工厂对象，从而提供与操作系统相关的实现。

## 5 你对原型设计模式的理解：

原型设计模式是一种对象创建型模式，它通过复制现有对象来创建新的对象，而无需显式地调用构造函数。

简单来说，原型设计模式就是通过克隆已有对象来创建新对象。

举一个通俗易懂的例子，假设我们有一个图形库，其中有一个基类 Shape 表示图形，它有一个纯虚函数 draw() 用于绘制图形。现在我希望能够复制已有的图形对象来创建新的图形对象，而无需重新构造图形对象。

下面是一个使用C++代码示例来说明原型设计模式的实现：

```
#include <iostream>
#include <string>

// 图形基类
class Shape {
public:
    virtual ~Shape() {}
    virtual void draw() = 0;
    virtual Shape* clone() = 0;
};

// 圆形类
class Circle : public Shape {
private:
    std::string color;
    int radius;

public:
    Circle(std::string color, int radius) : color(color), radius(radius) {}

    void draw() override {
        std::cout << "Drawing a circle with color " << color << " and radius " << radius << std::endl;
    }
}
```

```

    Shape* clone() override {
        return new Circle(*this);
    }
};

int main() {
    // 创建一个原型对象
    Shape* circlePrototype = new Circle("red", 5);

    // 克隆原型对象来创建新对象
    Shape* circle1 = circlePrototype->clone();
    Shape* circle2 = circlePrototype->clone();

    // 绘制图形
    circle1->draw(); // Output: Drawing a circle with color red and radius 5
    circle2->draw(); // Output: Drawing a circle with color red and radius 5

    delete circlePrototype;
    delete circle1;
    delete circle2;

    return 0;
}

```

在上述代码中，我们定义了一个图形基类 Shape，它声明了纯虚函数 draw 和 clone。然后我们创建了一个具体的图形类 Circle，它继承自 Shape，并实现了 draw 和 clone 函数。

在 main 函数中，我们首先创建了一个原型对象 circlePrototype，然后通过调用 clone 方法来创建了两个克隆对象 circle1 和 circle2。最后，我们分别调用了 draw 方法来绘制图形。

通过原型设计模式，我们可以通过复制已有的图形对象来创建新的图形对象，而无需重新构造图形对象。这样可以提高对象创建的效率，并且可以动态地添加新的图形对象。

原型设计模式的应用场景通常包括以下情况：

1. 创建对象的过程比较复杂，且需要频繁地创建相似对象。原型设计模式可以通过复制已有对象来创建新对象，避免了复杂的创建过程，提高了创建对象的效率。
2. 需要动态地添加新的对象，而且这些对象的类型在运行时才能确定。原型设计模式通过克隆已有对象来创建新对象，可以在运行时动态地添加新的对象类型，而无需显式地调用构造函数。

典型应用场景的举例：

1. 图形编辑器：在图形编辑器中，可能需要频繁地创建相似的图形对象，如矩形、圆形等。使用原型设计模式，可以先创建一个原型对象，并通过克隆原型对象来创建新的图形对象，而无需重新构造对象。
2. 缓存系统：在缓存系统中，可能需要频繁地复制已有的缓存对象来创建新的缓存对象。使用原型设计模式，可以通过克隆已有的缓存对象来创建新的缓存对象，而无需重新从数据库或其他地方获取数据。
3. 原始对象的状态保存：有时候需要保存对象的某个特定状态，以便在需要时恢复回来。原型设计模式可以通过克隆对象来保存和恢复对象的状态，而不需要手动记录和恢复每个属性的值。

原型设计模式适用于需要频繁地创建相似对象或动态地添加新对象的场景。它可以通过复制已有对象来创建新对象，避免了复杂的创建过程，并且可以在运行时动态地添加新的对象类型。

## 6 建造者设计模式的理解：

建造者设计模式是一种创建型设计模式，旨在将对象的构建过程与其表示分离。该模式允许逐步构建复杂的对象，同时保持构建过程的灵活性。

建造者设计模式的核心思想是将一个复杂对象的构建过程分解为多个简单的步骤，每个步骤由一个具体的建

造者类负责实现。这些步骤按照一定的顺序被调用，最终构建出一个完整的对象。

下面是一个简单的C++代码示例来说明建造者设计模式的实现：

```
#include <iostream>
#include <string>

// 产品类
class Pizza {
public:
    void setDough(const std::string& dough) {
        m_dough = dough;
    }

    void setSauce(const std::string& sauce) {
        m_sauce = sauce;
    }

    void setTopping(const std::string& topping) {
        m_topping = topping;
    }

    void showPizza() const {
        std::cout << "Pizza with " << m_dough << " dough, " << m_sauce << " sauce, and " << m_topping << " topping."
        << std::endl;
    }

private:
    std::string m_dough;
    std::string m_sauce;
    std::string m_topping;
};

// 抽象建造者类
class PizzaBuilder {
public:
    virtual ~PizzaBuilder() {}

    virtual void buildDough() = 0;
    virtual void buildSauce() = 0;
    virtual void buildTopping() = 0;
    virtual Pizza* getPizza() = 0;
};

// 具体建造者类A
class HawaiianPizzaBuilder : public PizzaBuilder {
public:
    void buildDough() override {
        m_pizza->setDough("cross");
    }

    void buildSauce() override {
        m_pizza->setSauce("mild");
    }

    void buildTopping() override {
        m_pizza->setTopping("ham and pineapple");
    }

    Pizza* getPizza() override {
        return m_pizza;
    }

private:
```



```

    Pizza* m_pizza = new Pizza();
};

// 具体建造者类B
class SpicyPizzaBuilder : public PizzaBuilder {
public:
    void buildDough() override {
        m_pizza->setDough("pan baked");
    }

    void buildSauce() override {
        m_pizza->setSauce("hot");
    }

    void buildTopping() override {
        m_pizza->setTopping("pepperoni and jalapeno");
    }

    Pizza* getPizza() override {
        return m_pizza;
    }

private:
    Pizza* m_pizza = new Pizza();
};

// 指导者类
class PizzaDirector {
public:
    void setPizzaBuilder(PizzaBuilder* builder) {
        m_builder = builder;
    }

    void constructPizza() {
        m_builder->buildDough();
        m_builder->buildSauce();
        m_builder->buildTopping();
    }

private:
    PizzaBuilder* m_builder;
};

int main() {
    PizzaDirector director;

    HawaiianPizzaBuilder hawaiianBuilder;
    director.setPizzaBuilder(&hawaiianBuilder);
    director.constructPizza();
    Pizza* hawaiianPizza = hawaiianBuilder.getPizza();
    hawaiianPizza->showPizza(); // .

    SpicyPizzaBuilder spicyBuilder;
    director.setPizzaBuilder(&spicyBuilder);
    director.constructPizza();
    Pizza* spicyPizza = spicyBuilder.getPizza();
    spicyPizza->showPizza(); //
    delete hawaiianPizza;
    delete spicyPizza;

    return 0;
}

```



在上述代码中，我们定义了一个产品类 Pizza，它具有三个属性：面团（dough）、酱料（sauce）和配料（topping）。然后我们定义了一个抽象建造者类 PizzaBuilder，它声明了一系列构建步骤的抽象方法，并提供了获取构建结果的方法 getPizza。

接下来，我们创建了两个具体建造者类 HawaiianPizzaBuilder 和 SpicyPizzaBuilder，它们分别实现了抽象建造者类中的构建步骤方法。这些方法负责设置不同的面团、酱料和配料。

然后我们定义了一个指导者类 PizzaDirector，它有一个成员变量指向具体的建造者对象。指导者类负责按照一定的顺序调用建造者对象的构建步骤方法来构建产品。

在 main 函数中，我们首先创建一个指导者对象，并将具体建造者对象 HawaiianPizzaBuilder 传递给指导者对象。然后通过指导者对象调用构建方法来构建产品。最后通过具体建造者对象的 getPizza 方法获取构建结果，并调用 showPizza 方法展示产品信息。

建造者设计模式的优点是可以将对象的构建过程与表示分离，使得构建过程更加灵活，同时可以避免构造函数的参数过多的问题。此外，通过使用建造者模式，可以更好地控制对象的创建过程，使得代码更加清晰易懂。

建造者设计模式的应用场景包括但不限于以下情况：

**创建复杂的对象：**当对象的构建过程比较复杂，涉及多个步骤、多个组件或者需要进行一些复杂的计算或处理时，可以使用建造者模式来将构建过程分解为多个简单的步骤，使得构建过程更加清晰和可控。

**避免构造函数的参数过多：**当一个对象的构造函数需要传递大量的参数时，使用构造函数来创建对象会导致代码难以理解和维护。而使用建造者模式，可以通过一系列方法来设置对象的属性，使得代码更加清晰易懂。

**创建一系列相似的对象：**当需要创建一系列相似但属性略有不同的对象时，使用建造者模式可以通过复用已有的建造者对象来构建这些对象，减少重复的代码。

**控制对象的创建过程：**建造者模式允许在构建过程中灵活地添加、修改或替换组件，从而更好地控制对象的创建过程。这样可以根据不同的需求，使用不同的建造者来构建具有不同特性的对象。

**应用场景：**

**创建游戏角色：**在游戏开发中，创建游戏角色通常涉及到多个属性的设置，例如角色的外观、能力、武器等。使用建造者模式可以将角色的构建过程分解为多个步骤，每个步骤由一个具体的建造者负责设置相应的属性。这样可以根据不同的建造者来构建不同类型的游戏角色。

**构建网页：**在网页设计中，创建网页通常需要设置不同的元素、样式和布局。使用建造者模式可以将网页的构建过程分解为多个步骤，例如设置标题、导航栏、内容区域等，每个步骤由一个具体的建造者负责设置相应的元素和样式。这样可以根据不同的建造者来构建不同风格和布局的网页。

**创建菜单：**在餐厅或者饮食应用中，创建菜单通常需要设置菜品的名称、描述、价格和配料等。使用建造者模式可以将菜单的构建过程分解为多个步骤，每个步骤由一个具体的建造者负责设置相应的属性。这样可以根据不同的建造者来构建不同类型和特色的菜单。

**生成报告：**在报告生成系统中，生成报告通常需要设置报告的标题、作者、日期、内容等。使用建造者模式可以将报告的构建过程分解为多个步骤，每个步骤由一个具体的建造者负责设置相应的属性。这样可以根据不同的建造者来构建不同类型和格式的报告。

## 7 享元设计模式的理解：

享元设计模式（Flyweight Design Pattern）是一种结构型设计模式，旨在通过共享对象来最小化内存使用和提高性能。该模式适用于需要创建大量细粒度对象的情况，通过共享相同的对象实例来减少内存占用。

在享元模式中，对象被分为两种类型：内部状态（Intrinsic State）和外部状态（Extrinsic State）。内部状态是对象共享的部分，不会随着外部环境的改变而改变；外部状态是对象特定的部分，会随着外部环境的改

变而改变。

下面是一个简单的C++代码示例来说明享元设计模式的实现：

```
#include <iostream>
#include <vector>
#include <unordered_map>

// 抽象享元类
class Flyweight {
public:
    virtual void operation(int extrinsicState) = 0;
};

// 具体享元类
class ConcreteFlyweight : public Flyweight {
public:
    void operation(int extrinsicState) override {
        std::cout << "具体享元对象，外部状态为: " << extrinsicState << std::endl;
    }
};

// 享元工厂类
class FlyweightFactory {
private:
    std::unordered_map<int, Flyweight*> flyweights;

public:
    Flyweight* getFlyweight(int key) {
        if (flyweights.find(key) == flyweights.end()) {
            flyweights[key] = new ConcreteFlyweight();
        }
        return flyweights[key];
    }
};

int main() {
    FlyweightFactory factory;

    Flyweight* flyweight = factory.getFlyweight(1);
    flyweight->operation(100);

    flyweight = factory.getFlyweight(2);
    flyweight->operation(200);

    return 0;
}
```

在上面的示例中，Flyweight 是抽象享元类，定义了享元对象的操作方法。ConcreteFlyweight 是具体享元类，实现了抽象享元类的方法。FlyweightFactory 是享元工厂类，用于创建和管理享元对象。

在 main 函数中，我们首先创建了一个享元工厂对象 factory。然后通过 factory 获取具体享元对象，调用其操作方法来展示对象的外部状态。

享元设计模式的应用场景：

**大量细粒度对象：**当需要创建大量细粒度的对象，并且这些对象之间有很多共享的部分时，可以使用享元模式来共享相同的对象实例，减少内存占用。

**缓存：**在需要频繁访问的数据缓存中，使用享元模式可以减少重复创建对象的开销，提高访问速度。

**文字处理器：**在文字处理器中，每个字符都可以看作一个享元对象，通过共享相同的字符对象实例，可以减少内存使用量。

**游戏中的角色和地图：**在游戏中，角色和地图等对象可能会有多个实例，使用享元模式可以共享相同的对象实例，提高游戏性能。

总之，享元设计模式适用于需要创建大量细粒度对象，并且这些对象之间有很多共享部分的场景。通过共享相同的对象实例，可以减少内存占用和提高性能。

## 8 外观者设计模式的理解：

外观设计模式（Facade Design Pattern）是一种结构型设计模式，旨在提供一个统一的接口，用于访问子系统中的一组接口，以简化客户端与子系统之间的交互。它允许客户端通过与外观对象交互，而不必直接与子系统的复杂性相互作用，从而降低了客户端代码的复杂性和依赖性。

核心思想： 外观模式提供了一个高级别的接口，将多个底层接口组合成一个更简单的接口，以供客户端使用。这有助于隔离客户端代码和复杂子系统之间的耦合关系。

主要组成部分：

外观类（Facade）： 提供一个简单的接口，封装了对一个或多个子系统的复杂操作。它知道如何与子系统协调工作以完成特定任务。

子系统类（Subsystems）： 这些类包含了系统的具体实现，但对客户端是不可见的。客户端通过外观类来与这些子系统进行交互。

特点：

简化接口： 外观模式提供了一个简化的接口，隐藏了子系统的复杂性，使客户端更容易使用。

降低耦合： 客户端代码与子系统之间的依赖性降低，因为客户端只需与外观对象交互，而不需要直接与多个子系统交互。

改进可维护性： 外观模式将系统的组件组织得更好，有助于维护和修改系统的不同部分，而不会影响客户端。

速记：

去医院看病，可能要去挂号、门诊、划价、取药，让患者或患者家属觉得很复杂，如果有提供接待人员，只让接待人员来处理，就很方便。

示例：

假设你正在开发一个多媒体播放器应用程序，其中包含**音频播放**、**视频播放**和**字幕显示**等多个子系统。你可以使用外观模式来创建一个多媒体播放器的外观类，将这些子系统组合起来，并提供一个简化的接口供客户端使用：

```
// 外观类 - 多媒体播放器
class MultimediaPlayer {
private:
    AudioPlayer audioPlayer;
    VideoPlayer videoPlayer;
    SubtitlePlayer subtitlePlayer;

public:
    MultimediaPlayer() {
        // 初始化子系统
    }

    // 提供一个播放多媒体文件的简化接口
    void playMedia(const std::string& mediaFilePath) {
        audioPlayer.playAudio(mediaFilePath);
        videoPlayer.playVideo(mediaFilePath);
    }
}
```

```

        subtitlePlayer.displaySubtitles(mediaFilePath);
    }
};

// 子系统类 - 音频播放器
class AudioPlayer {
public:
    void playAudio(const std::string& filePath) {
        std::cout << "播放音频: " << filePath << std::endl;
        // 实际音频播放逻辑
    }
};

// 子系统类 - 视频播放器
class VideoPlayer {
public:
    void playVideo(const std::string& filePath) {
        std::cout << "播放视频: " << filePath << std::endl;
        // 实际视频播放逻辑
    }
};

// 子系统类 - 字幕显示器
class SubtitlePlayer {
public:
    void displaySubtitles(const std::string& filePath) {
        std::cout << "显示字幕: " << filePath << std::endl;
        // 实际字幕显示逻辑
    }
};

```

在上面的示例中，MultimediaPlayer 类充当外观，隐藏了音频播放、视频播放和字幕显示等子系统的复杂性，提供了一个简化的接口供客户端使用。

应用场景：

外观模式通常在以下情况下使用：

当一个系统的多个复杂子系统需要与客户端交互，但客户端不需要了解这些子系统的具体工作方式时。

当你希望将系统的组件分层，使客户端可以轻松访问高级别的接口，而不必处理底层的复杂性。

当你希望隔离客户端代码和底层子系统，以提高代码的可维护性和可扩展性。外观模式可以帮助简化大型系统的设计和开发，降低了系统的复杂性。

## 9 组合设计模式的理解：

组合设计模式（Composite Design Pattern）是一种结构型设计模式，旨在将对象组织成树形结构，以表示“整体-部分”的层次结构。该模式使得客户端可以一致地处理单个对象和组合对象，从而简化了客户端的代码。

在组合模式中，有两种基本类型的对象：叶子对象（Leaf）和组合对象（Composite）。叶子对象表示树中的最细粒度的对象，没有子对象；组合对象是由一个或多个叶子对象或其他组合对象组成的对象，可以拥有子对象。

下面是一个简单的C++代码示例来说明组合设计模式的实现：

```

#include <iostream>
#include <vector>

```

```

// 抽象组件类
class Component {
public:
    virtual void operation() = 0;
};

// 叶子组件类
class Leaf : public Component {
public:
    void operation() override {
        std::cout << "叶子组件的操作" << std::endl;
    }
};

// 组合组件类
class Composite : public Component {
private:
    std::vector<Component*> components;

public:
    void addComponent(Component* component) {
        components.push_back(component);
    }

    void operation() override {
        std::cout << "组合组件的操作" << std::endl;
        for (Component* component : components) {
            component->operation();
        }
    }
};

int main() {
    Component* leaf = new Leaf();
    Component* composite = new Composite();
    composite->addComponent(leaf);

    composite->operation();

    delete composite;
    delete leaf;

    return 0;
}

```

在上面的示例中，Component 是抽象组件类，定义了组件对象的操作方法。Leaf 是叶子组件类，实现了抽象组件类的操作方法。Composite 是组合组件类，可持有多个叶子组件或其他组合组件，并实现了抽象组件类的操作方法。

在 main 函数中，我们首先创建了一个叶子组件对象 leaf 和一个组合组件对象 composite。然后通过 composite 的 addComponent 方法将叶子组件对象添加到组合组件对象中。最后调用组合组件对象的操作方法，会依次调用叶子组件对象的操作方法来展示整个组合对象的行为。

组合设计模式的应用场景：

**树形结构：**当需要表示对象的层次结构，且对象之间存在“整体-部分”的关系时，可以使用组合模式来构建树形结构，方便对整个结构进行统一操作。

**GUI界面中的布局：**在图形用户界面（GUI）的布局中，可以使用组合模式来构建复杂的布局结构，通过将容器组件和子组件组合起来，方便进行整体和部分的布局调整。

**文件系统的管理：**在文件系统中，文件和文件夹之间存在层次关系，可以使用组合模式来构建文件系统的层



次结构，方便对文件和文件夹进行统一的管理和操作。

**组织机构的管理：**在组织机构中，部门和员工之间存在层次关系，可以使用组合模式来构建组织机构的层次结构，方便对整个组织进行统一的管理和操作。

总之，组合设计模式适用于需要**构建树形结构、处理整体和部分的关系**，以及统一操作整个结构的场景。通过将对象组织成树形结构，可以简化客户端的代码，使得客户端可以一致地处理单个对象和组合对象。

## 10 你对适配器设计模式的理解：

适配器设计模式 (Adapter Design Pattern) 是一种结构型设计模式，用于将一个类的接口转换为另一个类的接口，以满足客户端的需求。适配器模式允许不兼容的类能够合作，通过适配器将一个类的接口转换为另一个类的接口，使得两者可以无缝协同工作。

适配器模式通常包含以下三个角色：目标接口 (Target)、适配器 (Adapter) 和被适配者 (Adaptee)。目标接口是客户端所期望的接口形式；适配器是将被适配者的接口转换为目标接口的类；被适配者是需要被适配的类。

下面是一个简单的C++代码示例来说明适配器设计模式的实现

```
#include <iostream>

// 目标接口
class Target {
public:
    virtual void request() = 0;
};

// 被适配者类
class Adaptee {
public:
    void specificRequest() {
        std::cout << "被适配者的特殊请求" << std::endl;
    }
};

// 类适配器
class Adapter : public Target, private Adaptee {
public:
    void request() override {
        specificRequest();
    }
};

int main() {
    Target* adapter = new Adapter();
    adapter->request();

    delete adapter;

    return 0;
}
```

在上面的示例中，Target 是目标接口，定义了客户端所期望的接口形式。Adaptee 是被适配者类，具有客户端无法直接使用的特殊接口。Adapter 是适配器类，通过继承被适配者类和实现目标接口的方法，将被适

配者类的接口转换为目标接口。

在 main 函数中，我们创建了一个适配器对象 adapter，然后调用它的 request 方法来展示适配器将被适配者的特殊请求转换为目标接口的能力。

适配器设计模式的应用场景包括但不限于以下情况：

**老旧接口的使用：**当需要使用一些老旧接口的功能，但又希望与现有的代码协同工作时，可以使用适配器模式来将老旧接口转换为现有代码所期望的接口形式。

**类库的兼容性：**当需要使用某个类库，但该类库的接口与当前项目的接口不兼容时，可以使用适配器模式来将类库的接口转换为项目所期望的接口形式。

**接口的统一：**当多个类具有不同的接口，但需要以统一的方式进行操作时，可以使用适配器模式来将它们的接口转换为统一的接口形式。

**与第三方组件的集成：**当需要将第三方组件集成到自己的系统中，但第三方组件的接口与自己的系统不兼容时，可以使用适配器模式来将第三方组件的接口转换为自己系统所期望的接口形式。

总之，适配器设计模式可以帮助不兼容的类能够协同工作，通过适配器将一个类的接口转换为另一个类的接口。适配器模式在实际开发中经常用于兼容性问题，以及将现有类库或组件集成到系统中。

## 11 代理设计模式的理解：

代理设计模式（Proxy Design Pattern）是一种结构型设计模式，用于在访问对象时提供一种代理，以控制对对象的访问。**代理模式通过引入一个代理对象来替代原始对象，从而可以在不改变原始对象的情况下增加额外的功能或控制访问。**

代理模式通常包含以下三个角色：抽象主题（Subject）、真实主题（Real Subject）和代理（Proxy）。抽象主题定义了真实主题和代理的共同接口；真实主题是实际执行业务逻辑的对象；代理是一个中间类，通过持有真实主题的引用并实现抽象主题的方法，将客户端的请求委派给真实主题。

下面是一个简单的C++代码示例来说明代理设计模式的实现：

```
#include <iostream>

// 抽象主题
class Subject {
public:
    virtual void request() = 0;
};

// 真实主题
class RealSubject : public Subject {
public:
    void request() override {
        std::cout << "真实主题的请求" << std::endl;
    }
};

// 代理
```



```

class Proxy : public Subject {
private:
    RealSubject* realSubject;

public:
    Proxy() {
        realSubject = new RealSubject();
    }

    ~Proxy() {
        delete realSubject;
    }

    void request() override {
        // 在调用真实主题之前可以添加额外的逻辑
        std::cout << "代理的请求" << std::endl;

        // 委托给真实主题处理
        realSubject->request();
    }
};

int main() {
    Subject* proxy = new Proxy();
    proxy->request();

    delete proxy;

    return 0;
}

```

在上面的示例中，Subject 是抽象主题，定义了真实主题和代理的共同接口。RealSubject 是真实主题，实现了抽象主题的方法，用于执行实际的业务逻辑。Proxy 是代理类，持有真实主题的引用，并实现抽象主题的方法，通过委托真实主题来处理客户端的请求。

在 main 函数中，我们创建了一个代理对象 proxy，然后调用它的 request 方法来展示代理将请求委托给真实主题的能力。

代理设计模式的应用场景包括但不限于以下情况：

**远程代理：**当需要访问远程对象时，可以使用代理模式来隐藏底层的网络通信细节，客户端通过代理对象访问远程对象。

**虚拟代理：**当创建一个对象的成本很高时，可以使用代理模式来延迟对象的实际创建，只有在真正需要时才创建真实对象。

**安全代理：**当需要对对象的访问进行控制时，可以使用代理模式来添加额外的安全性检查，例如权限验证等。

**缓存代理：**当需要缓存对象的结果以提高性能时，可以使用代理模式来实现缓存功能，代理对象可以在调用真实对象之前检查缓存，并返回缓存结果。

**日志记录代理**：当需要记录对象的操作日志时，可以使用代理模式来在调用真实对象之前或之后添加日志记录的功能。

总之，代理设计模式提供了一种控制对对象访问的方式，通过引入一个代理对象来替代原始对象，可以增加额外的功能或控制访问。代理模式在实际开发中经常用于远程访问、延迟加载、安全性控制、性能优化等场景。

## 12 装饰者设计模式的理解：

装饰者设计模式 (Decorator Design Pattern) 是一种结构型设计模式，**用于动态地给对象添加额外的职责，同时又不改变其原始类的结构**。装饰者模式通过将对象包装在一个装饰者类中，然后逐层地添加装饰者，从而实现对对象的透明扩展。

装饰者模式通常包含以下几个角色：**抽象构件 (Component)**、**具体构件 (Concrete Component)**、**抽象装饰者 (Decorator)** 和 **具体装饰者 (Concrete Decorator)**。抽象构件定义了原始对象和装饰者共同的接口；具体构件是原始对象，实现了抽象构件的接口；抽象装饰者是装饰者的抽象类或接口，包含一个指向抽象构件的引用；具体装饰者是具体的装饰者类，继承自抽象装饰者，通过对抽象构件进行装饰。

下面是一个简单的C++代码示例来说明装饰者设计模式的实现：

```
#include <iostream>

// 抽象构件
class Component {
public:
    virtual void operation() = 0;
};

// 具体构件
class ConcreteComponent : public Component {
public:
    void operation() override {
        std::cout << "执行具体构件的操作" << std::endl;
    }
};

// 抽象装饰者
class Decorator : public Component {
protected:
    Component* component;
public:
    Decorator(Component* component) : component(component) {}

    void operation() override {
        if (component != nullptr) {
            component->operation();
        }
    }
};

// 具体装饰者A
class ConcreteDecoratorA : public Decorator {
public:
    ConcreteDecoratorA(Component* component) : Decorator(component) {}

    void operation() override {
        addedBehavior();
    }
};
```

```

        Decorator::operation();
    }

    void addedBehavior() {
        std::cout << "具体装饰者A的附加操作" << std::endl;
    }
};

// 具体装饰者B
class ConcreteDecoratorB : public Decorator {
public:
    ConcreteDecoratorB(Component* component) : Decorator(component) {}

    void operation() override {
        addedBehavior();
        Decorator::operation();
    }

    void addedBehavior() {
        std::cout << "具体装饰者B的附加操作" << std::endl;
    }
};

int main() {
    Component* component = new ConcreteComponent();

    Component* decoratorA = new ConcreteDecoratorA(component);
    decoratorA->operation();

    Component* decoratorB = new ConcreteDecoratorB(decoratorA);
    decoratorB->operation();

    delete decoratorB;
    delete decoratorA;
    delete component;

    return 0;
}

```

在上面的示例中，Component 是抽象构件，定义了原始对象和装饰者共同的接口。ConcreteComponent 是具体构件，实现了抽象构件的接口，表示原始对象。Decorator 是抽象装饰者，包含一个指向抽象构件的引用，并实现了抽象构件的接口。ConcreteDecoratorA 和 ConcreteDecoratorB 是具体装饰者，继承自抽象装饰者，通过对抽象构件进行装饰。

在 main 函数中，我们创建了一个具体构件 component，然后分别用具体装饰者 ConcreteDecoratorA 和 ConcreteDecoratorB 对其进行装饰。最终，通过调用装饰者的 operation 方法，实现了对原始对象的透明扩展。

装饰者设计模式的应用场景包括但不限于以下情况：

- 1 动态增加功能：当需要动态地给对象添加额外的职责时，可以使用装饰者模式。通过创建不同的装饰者类，可以在不改变原始对象的情况下，透明地为对象添加新的功能。
2. 多层次的装饰：当需要对对象进行多层次的装饰时，可以使用装饰者模式。每个具体装饰者可以在上一层装饰的基础上添加新的功能。
3. 组合功能：当需要将多个不同的功能组合在一起使用时，可以使用装饰者模式。通过创建不同的具体装饰

者类，可以按需组合功能。

4. 开放封闭原则：当希望在不修改现有代码的情况下扩展功能时，可以使用装饰者模式。通过创建新的具体装饰者类，可以在不修改原始对象的情况下添加新的功能。

5. 单一职责原则：当希望将不同的责任分离到不同的类中，并且可以动态地进行组合时，可以使用装饰者模式。

总之，装饰者设计模式提供了一种动态地给对象添加额外职责的方式，同时又不改变其原始类的结构。装饰者模式在实际开发中经常用于动态增加功能、多层次的装饰、组合功能等场景。

### 13 桥接设计模式的理解：

桥接设计模式 (Bridge Design Pattern) 是一种结构型设计模式，用于将抽象部分与其具体实现部分分离，使它们可以独立地变化。桥接模式通过将抽象和实现通过一个桥接口进行连接，使得它们可以独立地扩展和变化，而不会相互影响。

桥接模式通常包含以下几个角色：抽象部分 (Abstraction)、具体抽象部分 (Concrete Abstraction)、实现部分 (Implementor) 和具体实现部分 (Concrete Implementor)。抽象部分定义了抽象接口，并包含一个对实现部分的引用；具体抽象部分是抽象部分的具体实现；实现部分定义了实现接口，并提供基本的操作；具体实现部分是实现部分的具体实现。

下面是一个简单的C++代码示例来说明桥接设计模式的实现：

```
#include <iostream>

// 实现部分
class Implementor {
public:
    virtual void operationImpl() = 0;
};

// 具体实现部分A
class ConcreteImplementorA : public Implementor {
public:
    void operationImpl() override {
        std::cout << "具体实现部分A的操作" << std::endl;
    }
};

// 具体实现部分B
class ConcreteImplementorB : public Implementor {
public:
    void operationImpl() override {
        std::cout << "具体实现部分B的操作" << std::endl;
    }
};

// 抽象部分
class Abstraction {
protected:
    Implementor* implementor;

public:
    Abstraction(Implementor* implementor) : implementor(implementor) {}

    virtual void operation() = 0;
};
```

```

// 具体抽象部分A
class ConcreteAbstractionA : public Abstraction {
public:
    ConcreteAbstractionA(Implementor* implementor) : Abstraction(implementor) {}

    void operation() override {
        std::cout << "具体抽象部分A的操作" << std::endl;
        implementor->operationImpl();
    }
};

// 具体抽象部分B
class ConcreteAbstractionB : public Abstraction {
public:
    ConcreteAbstractionB(Implementor* implementor) : Abstraction(implementor) {}

    void operation() override {
        std::cout << "具体抽象部分B的操作" << std::endl;
        implementor->operationImpl();
    }
};

int main() {
    Implementor* implementorA = new ConcreteImplementorA();
    Abstraction* abstractionA = new ConcreteAbstractionA(implementorA);
    abstractionA->operation();

    Implementor* implementorB = new ConcreteImplementorB();
    Abstraction* abstractionB = new ConcreteAbstractionB(implementorB);
    abstractionB->operation();

    delete abstractionB;
    delete implementorB;
    delete abstractionA;
    delete implementorA;

    return 0;
}

```

在上面的示例中，Implementor 是实现部分，定义了实现的接口。ConcreteImplementorA 和 ConcreteImplementorB 是具体实现部分，分别实现了 Implementor 的接口。

Abstraction 是抽象部分，定义了抽象接口，并包含一个对实现部分的引用。ConcreteAbstractionA 和 ConcreteAbstractionB 是具体抽象部分，继承自 Abstraction，实现了抽象部分的接口。

在 main 函数中，我们创建了一个具体实现部分 ConcreteImplementorA，然后用具体抽象部分 ConcreteAbstractionA 对其进行桥接。最终，通过调用抽象部分的 operation 方法，实现了对实现部分的透明调用。

桥接设计模式的应用场景包括但不限于以下情况：

- 1 当需要在抽象部分和实现部分之间进行解耦，使它们可以独立地变化时，可以使用桥接模式。
2. 当需要在运行时切换不同的实现部分时，可以使用桥接模式。通过更换具体实现部分，可以动态改变系统的行为。
3. 当一个类存在多个变化维度时，可以使用桥接模式。通过将每个变化维度抽象成抽象部分和实现部分，可以灵活地组合不同的变化维度。
4. 当需要对抽象部分和实现部分进行独立扩展时，可以使用桥接模式。通过增加新的具体实现部分或具体抽象部分，可以扩展系统的功能。

总之，桥接设计模式通过将抽象部分和实现部分分离，使它们可以独立地变化。桥接模式在实际开发中经常用于解耦、运行时切换实现、处理多个变化维度等场景。

#### 14 模板方法设计模式的概念

模板方法设计模式 (Template Method Pattern) 是一种行为型设计模式，用于定义一个算法的骨架，而将一些步骤的具体实现延迟到子类中。这种模式允许在不改变算法结构的情况下，通过子类重写或扩展某些步骤来定制算法的行为。

核心思想： 模板方法模式将一个算法的步骤模板化，定义在一个抽象的基类中，其中一些步骤由抽象方法或默认方法实现，而其他步骤则由具体子类实现。

主要组成部分：

**模板类 (Abstract Class) :** 定义了算法的骨架，通常包含一个或多个抽象方法，代表算法中的不同步骤。同时，也可以包含具体的方法来处理一些通用的逻辑。

**具体子类 (Concrete Subclasses) :** 继承自模板类，实现模板中的抽象方法，以完成特定步骤的具体实现。每个具体子类可以定制算法的不同部分。

特点：

定义了一个算法的框架，具体步骤由子类实现。

遵循开闭原则，即允许在不修改模板代码的情况下扩展和变化算法的某些部分。

促使代码重用，将通用行为提取到模板方法中。

示例：

假设你正在开发一个烹饪应用程序，其中有不同的食谱，如制作咖啡和茶。你可以使用模板方法模式来定义一个通用的烹饪算法

```
// 模板类 - 食谱
class Recipe {
public:
    // 模板方法，定义烹饪算法的骨架
    void cook() {
        boilWater();
        brew();
        pourInCup();
        if (customerWantsCondiments()) {
            addCondiments();
        }
    }

    // 抽象方法，由子类实现
    virtual void brew() = 0;
    virtual void addCondiments() = 0;

    // 钩子方法，子类可以选择性地覆盖
    virtual bool customerWantsCondiments() {
        return true;
    }

    // 具体方法，通用逻辑
    void boilWater() {
        std::cout << "烧开水" << std::endl;
    }
}
```



```

    void pourInCup() {
        std::cout << "倒入杯中" << std::endl;
    }
};

// 具体子类 - 制作咖啡
class CoffeeRecipe : public Recipe {
public:
    void brew() override {
        std::cout << "冲泡咖啡" << std::endl;
    }

    void addCondiments() override {
        std::cout << "加入糖和牛奶" << std::endl;
    }
};

// 具体子类 - 制作茶
class TeaRecipe : public Recipe {
public:
    void brew() override {
        std::cout << "浸泡茶叶" << std::endl;
    }

    void addCondiments() override {
        std::cout << "加入柠檬" << std::endl;
    }
};

```

在上面的示例中，Recipe 类定义了一个通用的烹饪算法框架，而具体的烹饪步骤由子类（CoffeeRecipe 和 TeaRecipe）来实现。这使得你可以轻松添加新的食谱，而不必更改通用的烹饪算法。

### 应用场景：

模板方法模式通常在以下情况下使用：

当你有一个通用的算法骨架，但不同部分的具体实现可能会变化时。

当你想要确保一些步骤在整个算法中保持一致性，而其他步骤由子类提供不同实现时。

当你希望通过子类来定制算法的某些部分，同时保留算法的整体结构。

当你需要遵循开闭原则，以便在不修改现有代码的情况下添加新的算法变种。

### 15 访问者设计模式的理解：

访问者设计模式（Visitor Design Pattern）是一种行为型设计模式，用于将**数据结构与对数据的操作分离**。

访问者模式允许在不更改数据结构的情况下定义新的操作。

访问者模式通常包含以下几个角色：访问者（Visitor）、具体访问者（Concrete Visitor）、元素（Element）、具体元素（Concrete Element）和对象结构（Object Structure）。

访问者模式的核心思想是将**数据结构的遍历和操作分离开**。访问者可以根据需求定义不同的具体访问者，每个具体访问者实现不同的操作。元素是数据结构的抽象，定义了接受访问者的接口。具体元素是元素的具体实现。对象结构是数据结构的集合，负责遍历元素并将访问者应用于每个元素。

下面是一个简单的C++代码示例来说明访问者设计模式的实现：

```

#include <iostream>
#include <vector>

// 前向声明
class ConcreteElementA;

```



```
class ConcreteElementB;

// 访问者接口
class Visitor {
public:
    virtual void visit(ConcreteElementA* element) = 0;
    virtual void visit(ConcreteElementB* element) = 0;
};

// 具体访问者A
class ConcreteVisitorA : public Visitor {
public:
    void visit(ConcreteElementA* element) override {
        std::cout << "具体访问者A访问具体元素A，执行操作A" << std::endl;
    }

    void visit(ConcreteElementB* element) override {
        std::cout << "具体访问者A访问具体元素B，执行操作B" << std::endl;
    }
};

// 具体访问者B
class ConcreteVisitorB : public Visitor {
public:
    void visit(ConcreteElementA* element) override {
        std::cout << "具体访问者B访问具体元素A，执行操作C" << std::endl;
    }

    void visit(ConcreteElementB* element) override {
        std::cout << "具体访问者B访问具体元素B，执行操作D" << std::endl;
    }
};

// 元素接口
class Element {
public:
    virtual void accept(Visitor* visitor) = 0;
};

// 具体元素A
class ConcreteElementA : public Element {
public:
    void accept(Visitor* visitor) override {
        visitor->visit(this);
    }
};

// 具体元素B
class ConcreteElementB : public Element {
public:
    void accept(Visitor* visitor) override {
        visitor->visit(this);
    }
};

// 对象结构
class ObjectStructure {
private:
    std::vector<Element*> elements;

public:
    void addElement(Element* element) {
        elements.push_back(element);
    }
}
```

```

void removeElement(Element* element) {
    // 移除元素的代码
}

void accept(Visitor* visitor) {
    for (Element* element : elements) {
        element->accept(visitor);
    }
}

};

int main() {
    ObjectStructure objectStructure;
    objectStructure.addElement(new ConcreteElementA());
    objectStructure.addElement(new ConcreteElementB());

    Visitor* visitorA = new ConcreteVisitorA();
    Visitor* visitorB = new ConcreteVisitorB();

    objectStructure.accept(visitorA);
    objectStructure.accept(visitorB);

    delete visitorA;
    delete visitorB;

    return 0;
}

```

在上面的示例中，Visitor 是访问者接口，定义了对具体元素的访问方法。ConcreteVisitorA 和 ConcreteVisitorB 是具体访问者，分别实现了对具体元素的不同操作。

Element 是元素接口，定义了接受访问者的方法。ConcreteElementA 和 ConcreteElementB 是具体元素，实现了元素接口。

ObjectStructure 是对象结构，负责维护元素集合，并提供遍历元素的方法。

在 main 函数中，我们创建了一个对象结构 `objectStructure`，并向其中添加了具体元素 `ConcreteElementA` 和 `ConcreteElementB`。然后我们创建了具体访问者 `ConcreteVisitorA` 和 `ConcreteVisitorB`。通过调用 `objectStructure` 的 `accept` 方法，并传入具体访问者，实现了对元素的遍历和对应操作的执行。

访问者设计模式的应用场景包括但不限于以下情况：

1. 当存在一个复杂的对象结构，且需要对该结构进行不同的操作时，可以使用访问者模式。通过将操作封装在具体访问者中，可以实现对不同操作的灵活扩展。
2. 当不希望在元素类中添加新的操作，或者需要对元素进行操作的算法具有不同的变化时，可以使用访问者模式。通过将操作封装在具体访问者中，可以在不更改元素类的情况下定义新的操作。
3. 当需要对元素进行组合访问，且元素类的结构稳定，但操作算法的变化频繁时，可以使用访问者模式。通过访问者模式，可以将不同的操作逻辑抽离出来，避免对元素类的修改。

总之，访问者设计模式通过将数据结构与对数据的操作分离，实现了数据结构和操作的解耦。访问者模式在实际开发中常用于处理复杂的对象结构，实现对不同操作的灵活扩展。

## 16 责任链设计模式的理解：

责任链设计模式（Chain of Responsibility Design Pattern）是一种行为型设计模式，用于将请求的发送者

和接收者解耦，并使多个对象都有机会处理该请求。请求沿着对象链进行传递，直到有一个对象能够处理它为止。

责任链模式通常包含以下几个角色：抽象处理器（Handler）、具体处理器（Concrete Handler）和客户端（Client）。

责任链模式的核心思想是将请求发送者和接收者解耦，每个处理器都有一个指向下一个处理器的引用。当请求到达处理器时，它可以选择处理请求并结束，或者将请求传递给下一个处理器。这样，请求会依次在处理器链中传递，直到有一个处理器能够处理它。

下面是一个简单的C++代码示例来说明责任链设计模式的实现：

```
#include <iostream>

// 抽象处理器
class Handler {
protected:
    Handler* nextHandler;

public:
    Handler() : nextHandler(nullptr) {}

    void setNextHandler(Handler* handler) {
        nextHandler = handler;
    }

    virtual void handleRequest(int request) = 0;
};

// 具体处理器A
class ConcreteHandlerA : public Handler {
public:
    void handleRequest(int request) override {
        if (request >= 0 && request < 10) {
            std::cout << "具体处理器A处理请求: " << request << std::endl;
        }
        else if (nextHandler != nullptr) {
            nextHandler->handleRequest(request);
        }
    }
};

// 具体处理器B
class ConcreteHandlerB : public Handler {
public:
    void handleRequest(int request) override {
        if (request >= 10 && request < 20) {
            std::cout << "具体处理器B处理请求: " << request << std::endl;
        }
        else if (nextHandler != nullptr) {
            nextHandler->handleRequest(request);
        }
    }
};

// 具体处理器C
class ConcreteHandlerC : public Handler {
public:
    void handleRequest(int request) override {
        if (request >= 20 && request < 30) {
```

```

        std::cout << "具体处理器C处理请求：" << request << std::endl;
    }
    else if (nextHandler != nullptr) {
        nextHandler->handleRequest(request);
    }
}

};

int main() {
    Handler* handlerA = new ConcreteHandlerA();
    Handler* handlerB = new ConcreteHandlerB();
    Handler* handlerC = new ConcreteHandlerC();

    handlerA->setNextHandler(handlerB);
    handlerB->setNextHandler(handlerC);

    handlerA->handleRequest(5);
    handlerA->handleRequest(15);
    handlerA->handleRequest(25);

    delete handlerA;
    delete handlerB;
    delete handlerC;

    return 0;
}

```

在上面的示例中，Handler 是抽象处理器，定义了处理请求的接口和一个指向下一个处理器的引用。ConcreteHandlerA、ConcreteHandlerB 和 ConcreteHandlerC 是具体处理器，实现了处理请求的具体逻辑。

在 main 函数中，我们创建了具体处理器 handlerA、handlerB 和 handlerC，并通过调用 setNextHandler 方法将它们链接起来形成责任链。然后通过调用 handleRequest 方法来发送请求，并观察请求是如何在责任链中传递和处理的。

责任链设计模式的应用场景包括但不限于以下情况：

当需要将请求发送者和接收者解耦，并且有多个对象都有机会处理请求时，可以使用责任链模式。通过建立一个处理器链，每个处理器可以选择处理请求或将请求传递给下一个处理器。

当希望动态地指定处理请求的对象集合时，可以使用责任链模式。通过动态地调整处理器链，可以实现灵活的请求处理。

当需要按顺序处理请求，并且每个处理器都有不同的处理逻辑时，可以使用责任链模式。

通过责任链模式，可以将请求沿着处理器链进行传递，每个处理器都有机会处理请求或者将其传递给下一个处理器。这样可以实现请求的逐级处理，直到有一个处理器能够处理该请求为止。

例如，在一个电商网站中，可以使用责任链模式来处理用户的退款请求。假设有三个处理器，分别是财务部门、客服部门和仓储部门。当用户发起退款请求时，请求会依次经过这三个处理器。财务部门负责审核退款申请，客服部门负责与用户沟通并确认退款原因，仓储部门负责处理退货物品。如果某个处理器无法处理该请求，它会将请求传递给下一个处理器，直到有一个处理器能够处理该请求或处理器链结束。

责任链设计模式的优点包括：

1. 解耦发送者和接收者：责任链模式使得发送者不需要知道请求将由哪个处理器来处理，将请求和处理解耦。
2. 灵活性和可扩展性：可以动态地调整处理器链，增加或移除处理器，以满足不同的处理需求。
3. 可以避免请求的发送者与接收者之间的紧耦合关系：将请求发送给处理器链，每个处理器都有机会处理请求，避免了发送者和接收者之间的直接依赖关系。

总之，**责任链设计模式通过将请求发送者和接收者解耦**，实现了对请求的逐级处理。它在实际开发中常用于处理请求的多级处理场景，提高了代码的灵活性和可扩展性。

## 17 中介者设计模式的理解：

中介者设计模式 (Mediator Design Pattern) 是一种行为型设计模式，用于降低多个对象之间的耦合性。中介者模式通过引入一个中介者对象，将对象间的交互集中管理，从而避免了对象之间的直接通信。

中介者模式通常包含以下几个角色：中介者 (Mediator)、具体中介者 (Concrete Mediator) 和同事类 (Colleague)。

中介者模式的核心思想是将对象间的交互通过中介者来进行协调和控制。当一个对象需要与其他对象进行通信时，它不直接与其他对象进行交互，而是通过中介者来进行通信。中介者负责接收和分发消息，并协调各个对象之间的交互。

下面是一个简单的C++代码示例来说明中介者设计模式的实现：

```
#include <iostream>
#include <string>

class Colleague;

// 中介者抽象类
class Mediator {
public:
    virtual void sendMessage(const std::string& message, Colleague* colleague) = 0;
};

// 具体中介者类
class ConcreteMediator : public Mediator {
private:
    Colleague* colleague1;
    Colleague* colleague2;
public:
    void setColleague1(Colleague* colleague) {
        colleague1 = colleague;
    }

    void setColleague2(Colleague* colleague) {
        colleague2 = colleague;
    }

    void sendMessage(const std::string& message, Colleague* colleague) override {
        if (colleague == colleague1) {
            colleague2->receiveMessage(message);
        }
        else if (colleague == colleague2) {
            colleague1->receiveMessage(message);
        }
    }
};
```

```

// 同事类
class Colleague {
protected:
    Mediator* mediator;

public:
    void setMediator(Mediator* mediator) {
        this->mediator = mediator;
    }

    virtual void sendMessage(const std::string& message) = 0;
    virtual void receiveMessage(const std::string& message) = 0;
};

// 具体同事类A
class ConcreteColleagueA : public Colleague {
public:
    void sendMessage(const std::string& message) override {
        mediator->sendMessage(message, this);
    }

    void receiveMessage(const std::string& message) override {
        std::cout << "同事类A收到消息: " << message << std::endl;
    }
};

// 具体同事类B
class ConcreteColleagueB : public Colleague {
public:
    void sendMessage(const std::string& message) override {
        mediator->sendMessage(message, this);
    }

    void receiveMessage(const std::string& message) override {
        std::cout << "同事类B收到消息: " << message << std::endl;
    }
};

int main() {
    Mediator* mediator = new ConcreteMediator();

    Colleague* colleague1 = new ConcreteColleagueA();
    Colleague* colleague2 = new ConcreteColleagueB();

    mediator->setColleague1(colleague1);
    mediator->setColleague2(colleague2);

    colleague1->setMediator(mediator);
    colleague2->setMediator(mediator);

    colleague1->sendMessage("Hello, Colleague B!");
    colleague2->sendMessage("Hi, Colleague A!");

    delete mediator;
    delete colleague1;
    delete colleague2;

    return 0;
}

```

在上面的示例中, Mediator 是中介者抽象类, 定义了中介者的接口。ConcreteMediator 是具体中介者类,

实现了中介者的具体逻辑。Colleague 是同事类，定义了同事类的接口。ConcreteColleagueA 和 ConcreteColleagueB 是具体同事类，实现了同事类的具体逻辑。

在 main 函数中，我们创建了一个具体中介者对象 mediator，以及两个具体同事类对象 colleague1 和 colleague2。然后通过调用相应的方法进行对象之间的交互。

中介者设计模式的应用场景包括但不限于以下情况：

1. 当对象间的通信过程需要集中管理和协调时，可以使用中介者模式。中介者模式可以将对象间的交互逻辑集中在中介者中，简化对象之间的通信。
  2. 当对象间存在循环依赖关系时，可以使用中介者模式。中介者模式可以解决对象间的循环依赖问题，将复杂的关系转化为中介者与各个对象之间的简单关系。
  - 3 当希望通过一个共享的中介者来减少系统中对象的数量时，可以使用中介者模式。中介者模式将对象间的一对多关系转化为一对一关系，减少了对象之间的直接耦合。
- 总之，中介者设计模式通过引入一个中介者对象来降低对象间的耦合性，将对象间的交互集中管理。它在实际开发中常用于需要集中管理对象间交互的场景，例如聊天室、调停者模式等。通过使用中介者模式，可以简化对象之间的通信，提高系统的灵活性和可扩展性。

### 18 状态设计模式的理解：

状态设计模式（State Design Pattern）是一种行为型设计模式，用于根据对象的内部状态改变其行为。状态模式将对象的行为封装在不同的状态类中，对象在不同状态下具有不同的行为。

状态模式通常包含以下几个角色：上下文（Context）、状态（State）和具体状态（Concrete State）。

上下文是拥有状态的对象，它维护一个指向当前状态对象的引用，并将请求委托给当前状态处理。上下文可以根据内部状态的改变来改变行为。

状态是一个抽象类或接口，定义了一个或多个在特定状态下的方法。这些方法将在具体状态类中被实现。

具体状态是状态的具体实现类，实现了在特定状态下的行为。

下面是一个简单的C++代码示例来说明状态设计模式的实现：

```
#include <iostream>

class Context;

// 状态抽象类
class State {
public:
    virtual void handle(Context* context) = 0;
};

// 具体状态类A
class ConcreteStateA : public State {
public:
    void handle(Context* context) override;
};
```



```

// 具体状态类B
class ConcreteStateB : public State {
public:
    void handle(Context* context) override;
};

// 上下文类
class Context {
private:
    State* currentState;

public:
    Context(State* initialState) : currentState(initialState) {}

    void setState(State* state) {
        currentState = state;
    }

    void request() {
        currentState->handle(this);
    }
};

void ConcreteStateA::handle(Context* context) {
    std::cout << "当前状态是A，执行操作A，切换到状态B" << std::endl;
    context->setState(new ConcreteStateB());
}

void ConcreteStateB::handle(Context* context) {
    std::cout << "当前状态是B，执行操作B，切换到状态A" << std::endl;
    context->setState(new ConcreteStateA());
}

int main() {
    Context* context = new Context(new ConcreteStateA());

    context->request(); // 输出：当前状态是A，执行操作A，切换到状态B
    context->request(); // 输出：当前状态是B，执行操作B，切换到状态A

    delete context;

    return 0;
}

```

在上面的示例中，State 是状态抽象类，定义了状态的接口方法 handle。ConcreteStateA 和 ConcreteStateB 是具体状态类，分别实现了在不同状态下的行为。Context 是上下文类，拥有一个当前状态的引用，并将请求委托给当前状态处理。

在 main 函数中，我们创建了一个上下文对象 context，并将初始状态设置为 ConcreteStateA。然后通过调用 context 的 request 方法来触发状态的改变。

状态设计模式的应用场景包括但不限于以下情况：

- 1 当一个对象的行为取决于其内部状态，并且该对象在运行时需要根据状态改变行为时，可以使用状态模式。
- 2 当一个操作具有多个不同的实现方式，每个实现方式对应一个状态时，可以使用状态模式。状态模式可以

将不同的操作实现分离到不同的状态类中，避免了使用大量的条件语句。

3 当对象的行为在不同状态下会发生变化，并且状态转换的规则较为复杂时，可以使用状态模式。状态模式将状态转换的逻辑封装在状态类中，使得状态转换的逻辑更加清晰、可维护。

总之，状态设计模式通过将对象的行为封装在不同的状态类中，根据对象的内部状态来改变其行为。它在实际开发中常用于需要根据对象的状态来进行不同操作的场景，例如订单状态变化、游戏；

4. 当一个对象在不同状态下需要执行不同的操作，并且需要动态地切换状态时，可以使用状态模式。状态模式可以使对象的状态转换更加灵活，可以根据需要随时切换状态。

5. 当需要添加新的状态时，状态模式具有良好的扩展性。通过添加新的具体状态类，可以很容易地扩展系统的行为。

总之，状态设计模式通过将对象的行为封装在不同的状态类中，根据对象的内部状态来改变其行为。它在实际开发中常用于需要根据对象的状态来进行不同操作的场景，例如订单状态变化、游戏角色状态转换等。通过使用状态模式，可以使代码更加清晰、可维护，并提高系统的灵活性和可扩展性。

## 19 观察者设计模式的理解：

观察者设计模式（Observer Design Pattern）是一种行为型设计模式，用于建立对象之间的一对多依赖关系，当一个对象的状态发生变化时，所有依赖它的对象都会得到通知并自动更新。

观察者模式通常包含以下几个角色：主题(Subject)、观察者(Observer)和具体观察者(Concrete Observer)。

主题是被观察的对象，它维护一个观察者列表，并提供添加、删除和通知观察者的方法。

观察者是一个抽象类或接口，定义了一个或多个接收主题通知的方法。

具体观察者是观察者的具体实现类，实现了接收通知并做出相应处理的方法。

下面是一个简单的C++代码示例来说明观察者设计模式的实现：

```
#include <iostream>
#include <vector>

class Observer;

// 主题抽象类
class Subject {
public:
    virtual void attach(Observer* observer) = 0;
    virtual void detach(Observer* observer) = 0;
    virtual void notify() = 0;
};

// 观察者抽象类
class Observer {
public:
    virtual void update() = 0;
};

// 具体观察者类A
class ConcreteObserverA : public Observer {
public:
    void update() override {
```

```

        std::cout << "具体观察者A收到通知并作出响应" << std::endl;
    }
};

// 具体观察者类B
class ConcreteObserverB : public Observer {
public:
    void update() override {
        std::cout << "具体观察者B收到通知并作出响应" << std::endl;
    }
};

// 具体主题类
class ConcreteSubject : public Subject {
private:
    std::vector<Observer*> observers;

public:
    void attach(Observer* observer) override {
        observers.push_back(observer);
    }

    void detach(Observer* observer) override {
        for (auto it = observers.begin(); it != observers.end(); ++it) {
            if (*it == observer) {
                observers.erase(it);
                break;
            }
        }
    }

    void notify() override {
        for (auto observer : observers) {
            observer->update();
        }
    }
};

int main() {
    ConcreteSubject subject;

    Observer* observerA = new ConcreteObserverA();
    Observer* observerB = new ConcreteObserverB();

    subject.attach(observerA);
    subject.attach(observerB);

    subject.notify(); // 输出: 具体观察者A收到通知并作出响应, 具体观察者B收到通知并作出响应

    subject.detach(observerA);

    subject.notify(); // 输出: 具体观察者B收到通知并作出响应

    delete observerA;
    delete observerB;

    return 0;
}

```

在上面的示例中，Subject 是主题的抽象类，定义了添加、删除和通知观察者的方法。Observer 是观察者的抽象类，定义了接收通知的方法。ConcreteObserverA 和 ConcreteObserverB 是具体观察者类，分别实现了接收通知并作出响应的方法。ConcreteSubject 是具体主题类，维护了一个观察者列表，并在状态变

化时通知观察者。

在 main 函数中，我们创建了一个具体主题对象 subject，并创建了两个具体观察者对象 observerA 和 observerB。然后通过调用 subject 的 attach 方法将观察者添加到观察者列表中，再调用 subject 的 notify 方法发送通知。观察者收到通知后会作出相应的响应。

观察者设计模式的应用场景包括但不限于以下情况：

当一个对象的改变需要通知其他对象，并且不希望这些对象耦合在一起时，可以使用观察者模式。观察者模式可以将主题和观察者解耦，使它们可以独立地进行扩展和维护。

2. 当一个对象的状态改变需要影响其他多个对象，并且这些对象的数量和关系可能动态变化时，可以使用观察者模式。观察者模式可以方便地管理和维护多个观察者对象，实现对象之间的松耦合。

3. 当一个对象需要在特定情况下通知其他对象并触发一系列操作时，可以使用观察者模式。观察者模式可以将通知和操作封装在具体观察者中，实现更加灵活的处理。

4. 当一个对象需要与多个对象进行交互，并且对交互的顺序和方式有特定要求时，可以使用观察者模式。观察者模式可以通过定义不同的具体观察者，灵活地管理对象之间的交互。

总之，观察者设计模式通过建立对象之间的一对多依赖关系，实现了对象之间的松耦合。它在实际开发中常用于需要在对象状态变化时通知其他对象并作出相应处理的场景，例如事件驱动编程、GUI界面开发、消息传递等。通过使用观察者模式，可以提高代码的可维护性和扩展性，并实现对象之间的解耦。

## 20 策略设计模式的理解：

策略设计模式 (Strategy Design Pattern) 是一种行为型设计模式，它定义了一系列算法，并将每个算法封装到具体的策略类中，使得它们可以互相替换。通过使用策略模式，可以在运行时动态地选择算法，而不需要修改调用算法的代码。

策略模式通常包含以下几个角色：上下文 (Context)、策略 (Strategy) 和具体策略 (Concrete Strategy)。

上下文是使用策略的对象，它将具体的策略类传给客户端，并在需要时调用策略的方法。

策略是一个抽象类或接口，定义了算法的公共接口。

具体策略是策略的具体实现类，实现了算法的具体逻辑。

下面是一个简单的C++代码示例来说明策略设计模式的实现

```
#include <iostream>

// 策略抽象类
class Strategy {
public:
    virtual void execute() = 0;
};

// 具体策略类A
class ConcreteStrategyA : public Strategy {
public:
    void execute() override {
```

```

        std::cout << "使用策略A执行算法" << std::endl;
    }
};

// 具体策略类B
class ConcreteStrategyB : public Strategy {
public:
    void execute() override {
        std::cout << "使用策略B执行算法" << std::endl;
    }
};

// 上下文类
class Context {
private:
    Strategy* strategy;

public:
    void setStrategy(Strategy* strategy) {
        this->strategy = strategy;
    }

    void executeStrategy() {
        if (strategy) {
            strategy->execute();
        }
    }
};

int main() {
    Context context;

    // 使用策略A执行算法
    ConcreteStrategyA strategyA;
    context.setStrategy(&strategyA);
    context.executeStrategy();

    // 使用策略B执行算法
    ConcreteStrategyB strategyB;
    context.setStrategy(&strategyB);
    context.executeStrategy();

    return 0;
}

```

在上面的示例中，Strategy 是策略的抽象类，定义了算法的执行接口。ConcreteStrategyA 和 ConcreteStrategyB 是具体策略类，分别实现了不同的算法。Context 是上下文类，用于使用策略执行算法。

在 main 函数中，我们创建了一个上下文对象 context，并通过调用 setStrategy 方法将具体策略对象传给上下文。然后调用 executeStrategy 方法执行算法，实际执行的算法取决于具体策略对象。

策略设计模式的应用场景包括但不限于以下情况：

- 1 当一个系统需要在不同时间应用不同的算法，并且希望能够灵活地切换算法时，可以使用策略模式。策略模式允许在运行时动态地选择算法，而不需要修改调用算法的代码。

2 当一个系统需要根据不同的条件选择不同的算法，并且这些条件可能会发生变化时，可以使用策略模式。策略模式可以通过定义不同的具体策略类来适应不同的条件，实现算法的灵活选择。

3 当一个系统的算法逻辑复杂，需要将不同的算法进行解耦和封装时，可以使用策略模式。策略模式将算法封装在具体策略类中，使得每个策略类只需要关注自己的算法逻辑，提高了代码的可读性和可维护性。

总之，策略设计模式通过将算法封装到具体的策略类中，实现了算法的可替换和动态选择。它在实际开发中常用于需要根据不同条件选择不同算法的场景，例如排序算法、计算策略、日志记录等。通过使用策略模式，可以提高代码的灵活性、可扩展性和可维护性。

补充说明：

在上述示例中，我们可以在 `Context` 类中添加一个方法 `void changeStrategy(Strategy\* newStrategy)`，用于在运行时切换策略。这样，在程序运行过程中可以动态地改变算法的行为，而不需要修改 `Context` 类的代码。

另外，策略模式还可以与工厂模式结合使用，通过工厂类来创建具体策略对象，并将其传递给上下文类。这样可以进一步降低上下文类对具体策略类的依赖性，增加系统的灵活性和可拓展性。

总结一下，策略设计模式通过将算法封装到具体的策略类中，实现了算法的可替换和动态选择。它在实际开发中常用于需要根据不同条件选择不同算法的场景，通过使用策略模式，可以提高代码的灵活性、可扩展性和可维护性。

## 21 解释器设计模式的理解：

解释器设计模式（Interpreter Design Pattern）是一种行为型设计模式，它定义了一种语言和解释器的结构，用于解释和执行特定的语法规则。该模式将语言的表达式表示为一个抽象语法树，并定义了一组解释器来解释和执行这些语法规则。

解释器模式通常包含以下几个角色：抽象表达式（Abstract Expression）、终结符表达式（Terminal Expression）、非终结符表达式（Non-terminal Expression）和上下文（Context）。

抽象表达式是一个抽象类或接口，定义了解释器的抽象方法。

终结符表达式表示语言中的终结符，它实现了抽象表达式的解释方法。

非终结符表达式表示语言中的非终结符，它通常由多个终结符表达式组成，并实现了抽象表达式的解释方法。

上下文类保存了解释器的全局信息，并提供给解释器访问和操作的接口。

下面是一个简单的C++代码示例来说明解释器设计模式的实现：

```
#include <iostream>
#include <unordered_map>

// 上下文类
class Context {
private:
    std::unordered_map<std::string, bool> variables;

public:
    bool getVariable(const std::string& name) {
        return variables[name];
    }
};
```



```

    }

    void setVariable(const std::string& name, bool value) {
        variables[name] = value;
    }
};

// 抽象表达式类
class Expression {
public:
    virtual bool interpret(Context& context) = 0;
};

// 终结符表达式类
class TerminalExpression : public Expression {
private:
    std::string variable;

public:
    TerminalExpression(const std::string& variable) : variable(variable) {}

    bool interpret(Context& context) override {
        return context.getVariable(variable);
    }
};

// 非终结符表达式类
class AndExpression : public Expression {
private:
    Expression* expression1;
    Expression* expression2;

public:
    AndExpression(Expression* expression1, Expression* expression2)
        : expression1(expression1), expression2(expression2) {}

    bool interpret(Context& context) override {
        return expression1->interpret(context) && expression2->interpret(context);
    }
};

// 非终结符表达式类
class OrExpression : public Expression {
private:
    Expression* expression1;
    Expression* expression2;

public:
    OrExpression(Expression* expression1, Expression* expression2)
        : expression1(expression1), expression2(expression2) {}

    bool interpret(Context& context) override {
        return expression1->interpret(context) || expression2->interpret(context);
    }
};

int main() {
    Context context;
    context.setVariable("A", true);
    context.setVariable("B", false);

    Expression* expression1 = new TerminalExpression("A");
    Expression* expression2 = new TerminalExpression("B");

    // A AND B

```



```

Expression* andExpression = new AndExpression(expression1, expression2);
bool result = andExpression->interpret(context);
std::cout << "A AND B = " << result << std::endl;

// A OR B
Expression* orExpression = new OrExpression(expression1, expression2);
result = orExpression->interpret(context);
std::cout << "A OR B = " << result << std::endl;

delete expression1;
delete expression2;
delete andExpression;
delete orExpression;

return 0;
}

```

在上述示例中，我们创建了一个简单的布尔表达式语言，并实现了解释器模式来解释和执行这些表达式。

Context 类保存了表达式的上下文信息，包括变量和对应的值。

Expression 是抽象表达式类，定义了解释器的抽象方法 interpret。

TerminalExpression 是终结符表达式类，表示语言中的终结符，它根据上下文中的变量值来解释和执行。

AndExpression 和 OrExpression 是非终结符表达式类，表示语言中的非终结符，它们通过组合终结符表达式来实现复杂的表达式解释和执行。

在 `main` 函数中，我们创建了一个上下文对象 `context`，并设置了变量 A 和 B 的值。然后，我们创建了终结符表达式对象 `expression1` 和 `expression2`，分别表示变量 A 和 B。接着，我们创建了非终结符表达式对象 `andExpression` 和 `orExpression`，分别表示 A AND B 和 A OR B 的表达式。

最后，我们调用 `interpret` 方法来解释和执行这些表达式，并输出结果。

解释器设计模式的应用场景包括但不限于以下情况：

1. 当需要解释和执行一种特定语言或表达式时，可以使用解释器模式。例如，当需要解析和执行一种自定义查询语言、配置文件或规则引擎时，解释器模式可以帮助我们实现解释器，并执行相应的语法规则。
2. 当需要灵活地扩展语言的语法规则时，可以使用解释器模式。解释器模式通过定义抽象表达式、终结符表达式和非终结符表达式，使得我们可以轻松地添加新的语法规则，并解释和执行这些规则。
3. 当需要对复杂的逻辑进行解析和执行时，可以使用解释器模式。解释器模式可以将复杂的逻辑分解为简单的语法规则，并通过组合和嵌套表达式来实现逻辑的解释和执行。

总之，解释器设计模式通过定义一种语言和解释器的结构，实现了对特定语法规则的解释和执行。它在需要解释和执行特定语言或表达式、扩展语言的语法规则以及对复杂逻辑进行解析和执行的场景中有广泛的应用。

## 22 备忘录设计模式的理解：

备忘录设计模式 (Memento Design Pattern) 是一种行为型设计模式，用于捕获和保存对象的内部状态，以便在需要时能够恢复到之前的状态。它将对象的状态封装在备忘录对象中，同时提供了对备忘录的创建、恢复和管理。

备忘录设计模式通常包含以下几个角色：原发器 (Originator)、备忘录 (Memento)、负责人 (Caretaker)。

原发器是需要保存状态的对象，它可以创建备忘录对象，将当前状态保存到备忘录中，或从备忘录中恢复之前的状态。

备忘录是用于保存原发器状态的对象，它通常提供了访问原发器状态的接口，但不允许其他对象修改备忘录的状态。

负责人是用于管理备忘录对象的对象，它保存了多个备忘录对象，并提供了对备忘录的管理操作，例如保存备忘录、获取备忘录等。

下面是一个简单的C++代码示例来说明备忘录设计模式的实现：

```
#include <iostream>
#include <string>

// 备忘录类
class Memento {
private:
    std::string state;

public:
    Memento(const std::string& state) : state(state) {}

    std::string getState() const {
        return state;
    }
};

// 原发器类
class Originator {
private:
    std::string state;

public:
    void setState(const std::string& state) {
        this->state = state;
    }

    std::string getState() const {
        return state;
    }

    Memento* createMemento() const {
        return new Memento(state);
    }

    void restoreMemento(const Memento* memento) {
        state = memento->getState();
    }
};

// 负责人类
class Caretaker {
private:
    Memento* memento;

public:
    void saveMemento(Memento* memento) {
        this->memento = memento;
    }
};
```

```

Memento* getMemento() const {
    return memento;
}
};

int main() {
    Originator originator;
    Caretaker caretaker;

    // 设置初始状态
    originator.setState("State 1");
    std::cout << "Current state: " << originator.getState() << std::endl;

    // 保存备忘录
    caretaker.saveMemento(originator.createMemento());

    // 修改状态
    originator.setState("State 2");
    std::cout << "Current state: " << originator.getState() << std::endl;

    // 恢复到之前的状态
    originator.restoreMemento(caretaker.getMemento());
    std::cout << "Current state: " << originator.getState() << std::endl;

    return 0;
}

```

在上述示例中，我们创建了一个备忘录设计模式的简单实现。Originator 类表示原发器，它具有一个状态 state，并提供了设置状态、获取状态、创建备忘录和恢复备忘录的方法。

Memento 类表示备忘录，它保存了原发器的状态，并提供了获取状态的方法。

Caretaker 类表示负责人，它保存了备忘录对象，并提供了保存备忘录和获取备忘录的方法。

在 main 函数中，我们创建了原发器对象 originator 和负责人对象 caretaker。然后，我们设置原发器的初始状态，并输出当前状态。接着，我们保存备忘录，并修改原发器的状态，并输出当前状态。最后，我们恢复到之前的状态，并输出当前状态，验证备忘录的恢复功能。

备忘录设计模式的应用场景包括但不限于以下情况：

当需要保存和恢复对象的内部状态时，可以使用备忘录模式。例如，在文本编辑器中，我们可以使用备忘录模式来保存编辑器的撤销和恢复操作，以便用户可以回到之前的编辑状态。

2. 当需要实现快照功能，保存对象的某个状态，以便后续可以随时恢复到该状态时，可以使用备忘录模式。例如，在游戏中，我们可以使用备忘录模式来保存游戏的进度，以便玩家可以在需要时恢复到之前的进度。

3. 当需要实现事务的回滚功能，即将一系列操作封装成一个事务，并在失败或取消时能够回滚到事务开始前的状态时，可以使用备忘录模式。例如，在数据库管理系统中，我们可以使用备忘录模式来实现事务的回滚功能。

4. 当需要实现多级撤销和恢复操作时，可以使用备忘录模式。备忘录模式可以保存多个备忘录对象，以便实现多级撤销和恢复的功能。

总之，备忘录设计模式通过将对象的内部状态封装在备忘录对象中，提供了对状态的保存、恢复和管理。它适用于需要保存和恢复对象状态、实现快照功能、实现事务回滚、实现多级撤销和恢复等场景。

## 23 命令设计模式的理解：

命令设计模式（Command Design Pattern）是一种行为型设计模式，用于将请求封装成一个对象，从而可以在不同的上下文中使用、传递和操作请求。它将请求的发送者和接收者解耦，使得发送者只需知道如何发送请求，而不需要知道请求是如何被执行和处理的。

命令设计模式通常包含以下几个角色：命令接口（Command）、具体命令（ConcreteCommand）、请求者（Invoker）、接收者（Receiver）。

命令接口是一个抽象接口，用于声明执行命令的方法。

具体命令是命令接口的具体实现，它将一个接收者对象和一组操作绑定在一起，并实现了执行命令的方法。

请求者是负责发送命令的对象，它持有一个命令对象，并在需要时调用命令对象的执行方法。

接收者是执行命令的对象，它实现了具体的操作逻辑，并在接收到命令时执行相应的操作。

下面是一个简单的C++代码示例来说明命令设计模式的实现：

```
#include <iostream>
#include <string>

// 命令接口
class Command {
public:
    virtual ~Command() {}
    virtual void execute() = 0;
};

// 具体命令
class ConcreteCommand : public Command {
private:
    std::string message;

public:
    ConcreteCommand(const std::string& message) : message(message) {}

    void execute() override {
        std::cout << "Executing command: " << message << std::endl;
        // 执行命令的具体操作
    }
};

// 请求者
class Invoker {
private:
    Command* command;

public:
    void setCommand(Command* command) {
        this->command = command;
    }

    void executeCommand() {
        command->execute();
    }
};
```

```

// 接收者
class Receiver {
public:
    void performAction() {
        std::cout << "Performing action" << std::endl;
        // 执行接收者的操作
    }
};

int main() {
    Invoker invoker;
    Receiver receiver;
    Command* command = new ConcreteCommand("Do something");

    // 设置命令
    invoker.setCommand(command);

    // 执行命令
    invoker.executeCommand();

    // 执行接收者的操作
    receiver.performAction();

    delete command;

    return 0;
}

```

在上述示例中，我们创建了一个命令设计模式的简单实现。Command 类表示命令接口，它声明了执行命令的方法。

ConcreteCommand 类表示具体命令，它实现了命令接口，并将一个接收者对象和一组操作绑定在一起。

Invoker 类表示请求者，它持有一个命令对象，并在需要时调用命令对象的执行方法。

Receiver 类表示接收者，它实现了具体的操作逻辑，并在接收到命令时执行相应的操作。

在 main 函数中，我们创建了请求者对象 invoker 和接收者对象 receiver。然后，我们创建了具体命令对象，并将其设置到请求者对象中。接着，我们调用请求者对象的执行命令方法，触发命令的执行。最后，我们调用接收者对象的执行操作方法，执行接收者的操作逻辑。

命令设计模式的应用场景包括但不限于以下情况：

当需要将请求发送者和接收者解耦，使得它们可以独立变化时，可以使用命令模式。命令模式将请求封装成一个对象，使得请求的发送者只需要关注如何发送请求，而不需要知道请求是如何被执行和处理的。

当需要实现撤销、重做、事务等功能时，可以使用命令模式。命令模式可以将每个命令封装成一个对象，并保存在历史记录中，以便可以撤销和重做命令，或者将多个命令组合成一个事务进行执行。

3. 当需要实现日志记录、审计或回放功能时，可以使用命令模式。命令模式可以将执行的命令保存下来，以便后续可以进行日志记录、审计或回放操作。

4. 当需要实现任务调度和命令队列时，可以使用命令模式。命令模式可以将命令对象放入队列中，按照顺序

执行，从而实现任务调度和命令队列的功能。

总之，命令设计模式通过将请求封装成一个对象，解耦了请求的发送者和接收者，使得请求的发送者只需要关注如何发送请求，而不需要知道请求是如何被执行和处理的。它适用于需要解耦请求发送者和接收者、实现撤销、重做、事务等功能、实现日志记录、审计或回放、实现任务调度和命令队列等场景。

## 24 迭代器设计模式的理解：

迭代器设计模式（Iterator Design Pattern）是一种行为型设计模式，用于提供一种统一的方式来访问集合对象中的元素，而无需暴露集合的内部表示。

迭代器设计模式通常包含以下几个角色：迭代器接口（Iterator）、具体迭代器（ConcreteIterator）、聚合接口（Aggregate）和具体聚合（ConcreteAggregate）。

迭代器接口是一个抽象接口，定义了访问和遍历集合元素的方法。

具体迭代器是迭代器接口的具体实现，实现了迭代器接口中的方法，并维护了一个指向集合中当前元素的指针。

聚合接口是一个抽象接口，定义了获取迭代器的方法。

具体聚合是聚合接口的具体实现，实现了聚合接口中的方法，并返回一个具体迭代器的实例。

下面是一个简单的C++代码示例来说明迭代器设计模式的实现：

```
#include <iostream>
#include <vector>

// 迭代器接口
template<class T>
class Iterator {
public:
    virtual bool hasNext() = 0;
    virtual T next() = 0;
};

// 具体迭代器
template<class T>
class ConcreteIterator : public Iterator<T> {
private:
    std::vector<T>& collection;
    int index = 0;

public:
    ConcreteIterator(std::vector<T>& collection) : collection(collection) {}

    bool hasNext() override {
        return index < collection.size();
    }

    T next() override {
        return collection[index++];
    }
};

// 聚合接口
```



```

template<class T>
class Aggregate {
public:
    virtual Iterator<T>* createIterator() = 0;
};

// 具体聚合
template<class T>
class ConcreteAggregate : public Aggregate<T> {
private:
    std::vector<T> collection;

public:
    void add(T item) {
        collection.push_back(item);
    }

    Iterator<T>* createIterator() override {
        return new ConcreteIterator<T>(collection);
    }
};

int main() {
    ConcreteAggregate<int> aggregate;
    aggregate.add(1);
    aggregate.add(2);
    aggregate.add(3);

    Iterator<int>* iterator = aggregate.createIterator();

    while (iterator->hasNext()) {
        std::cout << iterator->next() << " ";
    }

    delete iterator;

    return 0;
}

```

在上述示例中，我们创建了一个迭代器设计模式的简单实现。Iterator 类表示迭代器接口，它定义了访问和遍历集合元素的方法。

ConcreteIterator 类表示具体迭代器，它实现了迭代器接口，并维护了一个指向集合中当前元素的指针。

Aggregate 类表示聚合接口，它定义了获取迭代器的方法。

ConcreteAggregate 类表示具体聚合，它实现了聚合接口，并返回一个具体迭代器的实例。

在 main 函数中，我们创建了具体聚合对象 aggregate 并添加了一些元素。然后，我们通过调用 createIterator 方法获取迭代器，并使用迭代器遍历集合中的元素并打印。

迭代器设计模式的应用场景包括但不限于以下情况：

1 当需要统一访问和遍历集合对象中的元素时，可以使用迭代器模式。迭代器模式提供了一种统一的方式来



访问集合对象中的元素，而无需关注集合的内部表示和实现方式。

2当需要隐藏集合对象的内部结构，并提供一种安全的方式来遍历集合中的元素时，可以使用迭代器模式。迭代器模式将集合的遍历操作封装在迭代器中，使得外部代码无法直接访问集合的内部结构。

3当需要实现不同类型的集合对象的统一遍历方式时，可以使用迭代器模式。迭代器模式可以为每种类型的集合对象提供一个具体的迭代器实现，使得不同类型的集合对象都可以通过相同的方式进行遍历。

4. 当需要支持多种遍历方式或遍历顺序时，可以使用迭代器模式。迭代器模式可以为同一个集合对象提供多个不同的迭代器实现，每个迭代器都可以以不同的方式或顺序遍历集合。

总之，迭代器设计模式通过提供一个统一的方式来访问和遍历集合对象中的元素，隐藏了集合的内部结构，提供了安全的遍历方式，并支持多种遍历方式或顺序。它适用于需要统一访问和遍历集合对象、隐藏集合的内部结构、支持多种遍历方式或遍历顺序的场景。

QQ: 1069619619

## 6 数据库

### 1 数据库事务有哪些？

原子性： 所有操作要么全部成功，要么全部失败

一致性： 例如转账，一个事务执行前和执行后必须一致

隔离性： 防止脏读， 重复读问题

持久性： 永久性提交数据库

### 2、什么是 Redis？简述它的优缺点？

Redis是一种内存数据结构存储系统，具有高效的键值存储能力和对多种数据结构的支持。它可以用于缓存、消息中间件、计数器、队列等多种应用场景。

Redis的优点包括：

1. 快速：Redis将数据存储在内存在中，因此查找和读取数据非常快速。
2. 多种数据结构支持：Redis支持多种数据结构，包括字符串、哈希表、列表、集合和有序集合。
3. 高可用性：Redis具有高可用性，支持主从复制和哨兵模式，可以确保数据的高可靠性和持久性。
4. 可扩展性：Redis的性能可以水平扩展，可以根据需要添加新的节点。
5. 简单易用：Redis API简单易用，上手容易，提供了大量的命令和操作。

Redis的缺点包括：

1. 数据持久化：Redis默认不支持持久化，需要手动配置。
2. 数据量限制：Redis在内存中存储数据，因此受到物理内存大小的限制。
3. 安全性：Redis缺少内置的权限和认证机制，需要自行配置。
4. 单线程：Redis是单线程的，不能利用多核CPU的优势。
5. 操作复杂度：Redis在处理一些高级操作时，可能复杂度较高，需要考虑操作的性能和可行性。

### 3、Redis 支持哪几种数据类型？

## 1. string 字符串

字符串类型是 Redis 最基础的数据结构，首先键是字符串类型，而且其他几种结构都是在字符串类型基础上构建的。字符串类型实际上可以是字符串：简单的字符串、XML、JSON；数字：整数、浮点数；二进制：图片、音频、视频。

使用场景：缓存、计数器、共享 Session、限速。

- ## 2. Hash (哈希)

在 Redis 中哈希类型是指键本身是一种键值对结构，如 `value={{field1,value1},.....{fieldN,valueN}}`

使用场景：哈希结构相对于字符串序列化缓存信息更加直观，并且在更新操作上更加便捷。所以常常用于用户信息管理等，但是哈希类型和关系型数据库有所不同，哈希类型是稀疏的，而关系型数据库是完全结构化的，关系型数据库可以做复杂的关系查询，而 Redis 去模拟关系型复杂查询开发困难且维护成本高。

- ## 3. List (列表)

列表类型是用来储存多个有序的字符串，列表中的每个字符串成为元素，一个列表最多可以储存  $2^{32} - 1$  个元素，在 Redis 中，可以队列表两端插入和弹出，还可以获取指定范围的元素列表、获取指定索引下的元素等，列表是一种比较灵活的数据结构，它可以充当栈和队列的角色。

使用场景：Redis 的 `lpush + brpop` 命令组合即可实现阻塞队列，生产者客户端是用 `lpush` 从列表左侧插入元素，多个消费者客户端使用 `brpop` 命令阻塞式的“抢”列表尾部的元素，多个客户端保证了消费的负载均衡和高可用性。



- **4. Set** (集合)

集合类型也是用来保存多个字符串的元素，但和列表不同的是集合中不允许有重复的元素，并且集合中的元素是无序的，不能通过索引下标获取元素，Redis 除了支持集合内的增删改查，同时还支持多个集合取交集、并集、差集。合理的使用好集合类型，能在实际开发中解决很多实际问题。

使用场景：如：一个用户对娱乐、体育比较感兴趣，另一个可能对新闻感兴趣，这些兴趣就是标签，有了这些数据就可以得到同一标签的人，以及用户的共同爱好的标签，这些数据对于用户体验以及增强用户粘度比较重要。

- **5. zset** (sorted set: \*\*有序集合) \*\*

有序集合和集合有着必然的联系，它保留了集合不能有重复成员的特性，但不同的是，有序集合中的元素是可以排序的，但是它和列表的使用索引下标作为排序依据不同的是：它给每个元素设置一个分数，作为排序的依据。

使用场景：排行榜是有序集合经典的使用场景。例如：视频网站需要对用户上传的文件做排行榜，榜单维护可能是多方面：按照时间、按照播放量、按照获得的赞数等。

## 4、Redis 一般都有哪些使用场景？

Redis 适合的场景

1. 缓存：减轻 MySQL 的查询压力，提升系统性能；
2. 排行榜：利用 Redis 的 SortSet (有序集合)实现；
3. 计数器/限速器：利用 Redis 中原子性的自增操作，我们可以统计类似用户点赞数、用户访问数等。这类操作如果用 MySQL，频繁的读写会带来相当大的压力；限速器比较典型的使用场景是限制某个用户访问某个 API 的频率，常用的有抢购时，防止用户疯狂点击带来不必要的压力；
4. 好友关系：利用集合的一些命令，比如求交集、并集、差集等。可以方便解决一些共同好友、共同爱好之类的功能；
5. 消息队列：除了 Redis 自身的发布/订阅模式，我们也可以利用 List 来实现一个队列机制，比如：到货

通知、邮件发送之类的需求，不需要高可靠，但是会带来非常大的 DB 压力，完全可以用 List 来完成异步解耦；

6. Session 共享：Session 是保存在服务器的文件中，如果是集群服务，同一个用户过来可能落在不同机器上，这就会导致用户频繁登陆；采用 Redis 保存 Session 后，无论用户落在那台机器上都能够获取到对应的 Session 信息。

## Redis 不适合的场景

数据量太大、数据访问频率非常低的业务都不适合使用 Redis，数据太大会增加成本，访问频率太低，保存在内存中纯属浪费资源。

## 5、Redis 有哪些常见的功能？

1. 数据缓存功能
2. 分布式锁的功能
3. 支持数据持久化
4. 支持事务
5. 支持消息队列

## 6、谈谈你对索引的理解？

索引的出现是为了提高数据的查询效率，就像书的目录一样。一本500页的书，如果你想快速找到其中的某一个知识点，在不借助目录的情况下，那我估计你可得找一会儿。同样，对于数据库的表而言，索引其实就是它的“目录”。

同样索引也会带来很多负面影响：创建索引和维护索引需要耗费时间，这个时间随着数据量的增加而增加；索引需要占用物理空间，不光是表需要占用数据空间，每个索引也需要占用物理空间；当对表进行增、删、改、的时候索引也要动态维护，这样就降低了数据的维护速度。

建立索引的原则：

1. 在最频繁使用的、用以缩小查询范围的字段上建立索引；

2. 在频繁使用的、需要排序的字段上建立索引。

不适合建立索引的情况：

1. 对于查询中很少涉及的列或者重复值比较多的列，不宜建立索引；
2. 对于一些特殊的数据类型，不宜建立索引，比如：文本字段(text)等。

## 7、索引的底层使用的是什么数据结构？

索引的数据结构和具体存储引擎的实现有关，在MySQL中使用较多的索引有 Hash 索引、B+树索引等。而我们经常使用的 InnoDB 存储引擎的默认索引实现为 B+ 树索引。

## 8、谈谈你对 B+ 树的理解？

1. B+ 树是基于 B 树和叶子节点顺序访问指针进行实现，它具有 B 树的平衡性，并且通过顺序访问指针来提高区间查询的性能。
2. 在 B+ 树中，一个节点中的 key 从左到右非递减排列，如果某个指针的左右相邻 key 分别是 key<sub>i</sub> 和 key<sub>i+1</sub>，且不为 null，则该指针指向节点的所有 key 大于等于 key<sub>i</sub> 且小于等于 key<sub>i+1</sub>。
3. 进行查找操作时，首先在根节点进行二分查找，找到一个 key 所在的指针，然后递归地在指针所指向的节点进行查找。直到查找到叶子节点，然后在叶子节点上进行二分查找，找出 key 所对应的 data。
4. 插入、删除操作会破坏平衡树的平衡性，因此在插入删除操作之后，需要对树进行一个分裂、合并、旋转等操作来维护平衡性。

## 9、为什么 InnoDB 存储引擎选用 B+ 树而不是 B 树呢？

用 B+ 树不用 B 树考虑的是 IO 对性能的影响，B 树的每个节点都存储数据，而 B+ 树只有叶子节点才存储数据，所以查找相同数据量的情况下，B 树的高度更高，IO 更频繁。数据库索引是存储在磁盘上的，当数据量大时，就不能把整个索引全部加载到内存了，只能逐一加载每一个磁盘页(对应索引树的节点)。

## 10、谈谈你对聚簇索引的理解？

聚簇索引是对磁盘上实际数据重新组织以按指定的一个或多个列的值排序的算法。特点是存储数据的顺序和索引顺序一致。一般情况下主键会默认创建聚簇索引，且一张表只允许存在一个聚簇索引。

聚簇索引和非聚簇索引的区别：

聚簇索引的叶子节点就是数据节点，而非聚簇索引的叶子节点仍然是索引节点，只不过有指向对应数据块的指针。

## 11、谈谈你对哈希索引的理解？

哈希索引能以 O(1) 时间进行查找，但是失去了有序性。无法用于排序与分组、只支持精确查找，无法用于部分查找和范围查找。

InnoDB 存储引擎有一个特殊的功能叫“自适应哈希索引”，当某个索引值被使用的非常频繁时，会在 B+ 树索引之上再创建一个哈希索引，这样就让 B+Tree 索引具有哈希索引的一些优点，比如：快速的哈希查找。

## 12、谈谈你对覆盖索引的认识？

如果一个索引包含了满足查询语句中字段与条件的数据就叫做覆盖索引。具有以下优点：

1. 索引通常远小于数据行的大小，只读取索引能大大减少数据访问量。
2. 一些存储引擎(例如：MyISAM)在内存中只缓存索引，而数据依赖于操作系统来缓存。因此，只访问索引可以不使用系统调用(通常比较费时)。
3. 对于 InnoDB 引擎，若辅助索引能够覆盖查询，则无需访问主索引。

## 13、索引的分类？

- 从数据结构角度

1. 树索引 ( $O(\log(n))$ )
2. Hash 索引

从物理存储角度

1. 聚集索引 (clustered index)
2. 非聚集索引 (non-clustered index)

从逻辑角度

1. 普通索引
2. 唯一索引
3. 主键索引
4. 联合索引
5. 全文索引

## 13、谈谈你对最左前缀原则的理解？

MySQL 使用联合索引时，需要满足最左前缀原则。下面举例对其进行说明：

1. 一个 2 列的索引 (name, age)，对 (name)、(name, age) 上建立了索引；
2. 一个 3 列的索引 (name, age, sex)，对 (name)、(name, age)、(name, age, sex) 上建立了索引。

1、B+ 树的数据项是复合的数据结构，比如：(name, age, sex) 的时候，B+ 树是按照从左到右的顺序来建立搜索树的，比如：当(小明, 22, 男)这样的数据来检索的时候，B+ 树会优先比较 name 来确定下一步的搜索方向，如果 name 相同再依次比较 age 和 sex，最后得到检索的数据。但当 (22, 男) 这样没有 name 的数据来的时候，B+ 树就不知道第一步该查哪个节点，因为建立搜索树的时候 name 就是第一个比较因子，必须要先根据 name 来搜索才能知道下一步去哪里查询。

2、当 (小明, 男) 这样的数据来检索时，B+ 树可以用 name 来指定搜索方向，但下一个字段 age 的缺失，所以只能把名字等于小明的数据都找到，然后再匹配性别是男的数据了，这个是非常重要的性质，即索引的最左匹配特性。

关于最左前缀的补充：

1. 最左前缀匹配原则会一直向右匹配直到遇到范围查询(>、<、between、like)就停止匹配，比如：  
a = 1 and b = 2 and c > 3 and d = 4 如果建立 (a, b, c, d) 顺序的索引，d 是用不到索引的。如果建立 (a, b, d, c) 的索引则都可以用到，a、b、d 的顺序可以任意调整。
2. = 和 in 可以乱序，比如：a = 1 and b = 2 and c = 3 建立 (a, b, c) 索引可以任意顺序，MySQL 的优化器会优化



成索引可以识别的形式。

## 14、怎么知道创建的索引有没有被使用到？或者说怎么才可以知道这条语句

### 运行很慢的原因？

使用 Explain 命令来查看语句的执行计划，MySQL 在执行某个语句之前，会将该语句过一遍查询优化器，之后会拿到对语句的分析，也就是执行计划，其中包含了许多信息。可以通过其中和索引有关的信息来分析是否命中了索引，例如：possible\_key、key、key\_len 等字段，分别说明了此语句可能会使用的索引、实际使用的索引以及使用的索引长度。

## 15、什么情况下索引会失效？即查询不走索引？

下面列举几种不走索引的 SQL 语句：

### 1、索引列参与表达式计算：

```
SELECT 'sname' FROM 'stu' WHERE 'age' + 10 = 30;
```

### 2、函数运算：

```
SELECT 'sname' FROM 'stu' WHERE LEFT('date',4) < 1990;
```

### 3、%词语%--模糊查询：

```
SELECT * FROM 'manong' WHERE `uname` LIKE '%码农%' -- 走索引
```

```
SELECT * FROM 'manong' WHERE `uname` LIKE "%码农%" -- 不走索引
```

### 4、字符串与数字比较不走索引：

```
CREATE TABLE 'a' ( 'a' char(10));
```

```
EXPLAIN SELECT * FROM 'a' WHERE 'a'="1" -- 走索引
```

```
EXPLAIN SELECT * FROM 'a' WHERE 'a'=1 -- 不走索引，同样也是使用了函数运算
```

### 5、查询条件中有 or，即使其中有条件带索引也不会使用。换言之，就是要求使用的所有字段，都必须建立索引：

```
select * from dept where dname= 'xxx' or loc= 'xx' or deptno = 45;
```

### 6、正则表达式不使用索引。

### 7、MySQL 内部优化器会对 SQL 语句进行优化，如果优化器估计使用全表扫描要比使用索引快，则不使用索引。

## 16、查询性能的优化方法？

减少请求的数据量

1. 只返回必要的列：最好不要使用 `SELECT *` 语句。
2. 只返回必要的行：使用 `LIMIT` 语句来限制返回的数据。
3. 缓存重复查询的数据：使用缓存可以避免在数据库中进行查询，特别在要查询的数据经常被重复查询时，缓存带来的查询性能提升将会是非常明显的。

减少服务器端扫描的行数

1. 最有效的方式是使用索引来覆盖查询。

## 17、InnoDB 和 MyISAM 的比较？

1. 事务：MyISAM不支持事务，InnoDB支持事务；
2. 全文索引：MyISAM 支持全文索引，InnoDB 5.6 之前不支持全文索引；
3. 关于 `count()`：MyISAM会直接存储总行数，InnoDB 则不会，需要按行扫描。意思就是对于 `select count() from table;` 如果数据量大，MyISAM 会瞬间返回，而 InnoDB 则会一行行扫描；
4. 外键：MyISAM 不支持外键，InnoDB 支持外键；
5. 锁：MyISAM 只支持表锁，InnoDB 可以支持行锁。

## 18、谈谈你对水平切分和垂直切分的理解？

- 水平切分

水平切分是将同一个表中的记录拆分到多个结构相同的表中。当一个表的数据不断增多时，水平切分是必然的选择，它可以将数据分布到集群的不同节点上，从而缓存单个数据库的压力。

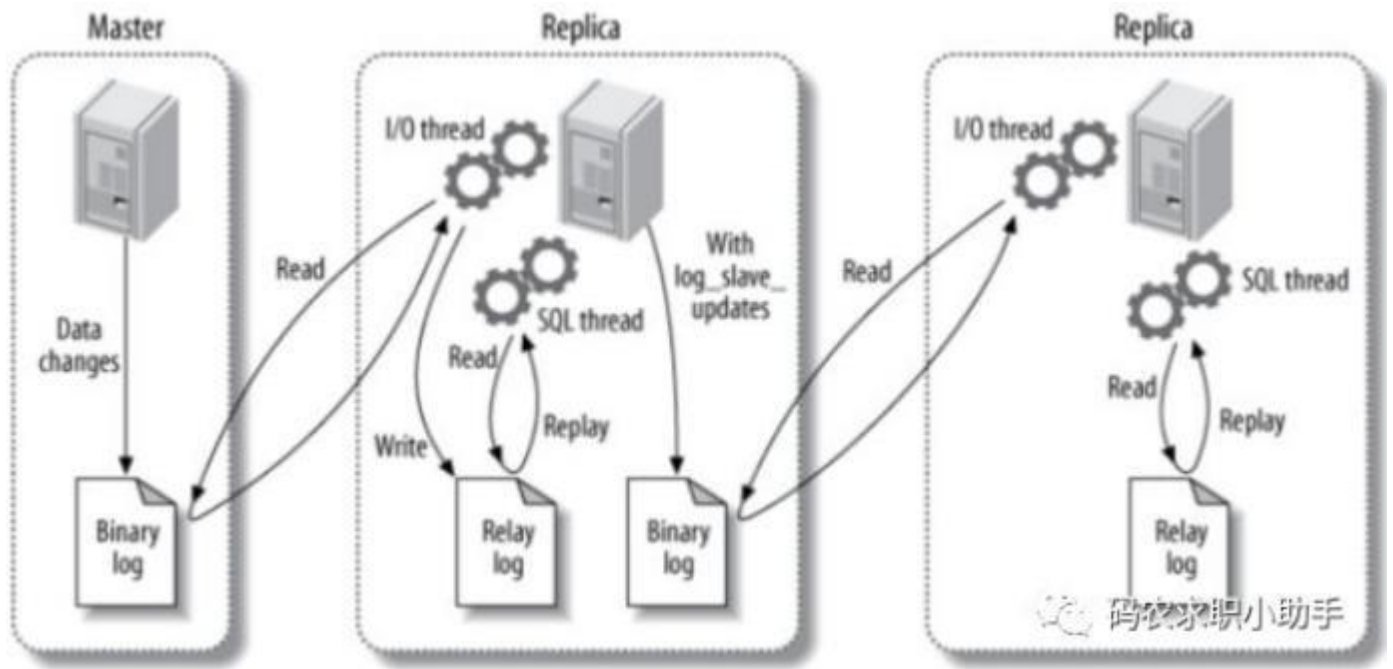
- 垂直切分

垂直切分是将一张表按列切分成多个表，通常是按照列的关系密集程度进行切分，也可以利用垂直切分将经常被使用的列和不经常被使用的列切分到不同的表中。例如：将原来的电商数据库垂直切分成商品数据库、用户数据库 等。

## 19、主从复制中涉及到哪三个线程？

主要涉及三个线程：binlog 线程、I/O 线程和 SQL 线程。

1. binlog 线程：负责将主服务器上的数据更改写入二进制日志(Binary log)中。
2. I/O 线程：负责从主服务器上读取二进制日志，并写入从服务器的重放日志(Relay log)中。
3. SQL 线程：负责读取重放日志并重放其中的 SQL 语句。



## 20、主从同步的延迟原因及解决办法？

主从同步的延迟的原因：

假如一个服务器开放 N 个连接给客户端，这样会有大并发的更新操作，但是从服务器的里面读取 binlog 的线程 仅有一个，当某个 SQL 在从服务器上执行的时间稍长或者由于某个 SQL 要进行锁表就会导致主服务器的 SQL 大量积压，未被同步到从服务器里。这就导致了主从不一致，也就是主从延迟。

主从同步延迟的解决办法：

实际上主从同步延迟根本没有什么一招制敌的办法，因为所有的 SQL 必须都要在从服务器里面执行一遍，但是主服务器如果不断的有更新操作源源不断的写入，那么一旦有延迟产生，那么延迟加重的可能性就会越来越大。当然 我们可以做一些缓解的措施。

1. 我们知道因为主服务器要负责更新操作，它对安全性的要求比从服务器高，所有有些设置可以修改，比如 `sync_binlog=1`，`innodb_flush_log_at_trx_commit = 1` 之类的设置，而 slave 则不需要这么高的数据安全，完全可以将 `sync_binlog` 设置为 0 或者关闭 binlog、`innodb_flushlog`、`innodb_flush_log_at_trx_commit` 也可以设置为 0 来提高 SQL 的执行效率。
2. 增加从服务器，这个目的还是分散读的压力，从而降低服务器负载。

## 21、谈谈你对数据库读写分离的理解？

读写分离常用代理方式来实现，代理服务器接收应用层传来的读写请求，然后决定转发到哪个服务器。主服务器处理写操作以及实时性要求比较高的读操作，而从服务器处理读操作。

读写分离能提高性能的原因在于：

1. 主从服务器负责各自的读和写，极大程度缓解了锁的争用；
2. 从服务器可以使用 MyISAM，提升查询性能以及节约系统开销；
3. 增加冗余，提高可用性。

## 22、请你描述下事务的特性？

1. 原子性：事务是最小的执行单位，不允许分割。事务的原子性确保动作要么全部完成，要么完全不起作用；
2. 一致性：执行事务前后，数据库从一个一致性状态转换到另一个一致性状态。
3. 隔离性：并发访问数据库时，一个用户的事物不被其他事务所干扰，各并发事务之间数据库是独立的；
4. 持久性：一个事务被提交之后。它对数据库中数据的改变是持久的，即使数据库发生故障也不应该对其有任何影响。

## 23、谈谈你对事务隔离级别的理解？

1. READ\_UNCOMMITTED（未提交读）：最低的隔离级别，允许读取尚未提交的数据变更，可能会导致脏读、幻读或不可重复读；
2. READ\_COMMITTED（提交读）：允许读取并发事务已经提交的数据，可以阻止脏读，但是幻读或不可重复读仍有可能发生；
3. REPEATABLE\_READ（可重复读）：对同一字段的多次读取结果都是一致的，除非数据是被本身事务自己所修改，可以阻止脏读和不可重复读，但幻读仍有可能发生；
4. SERIALIZABLE（串行化）：最高的隔离级别，完全服从 ACID 的隔离级别。所有的事务依次逐个执行，这样事务之间就完全不可能产生干扰，也就是说，该级别可以防止脏读、不可重复读以及幻读。但是这将严重影响程序的性能。通常情况下也不会用到该级别。

## 24、解释下什么叫脏读、不可重复读和幻读？

- 脏读：

表示一个事务能够读取另一个事务中还未提交的数据。比如：某个事务尝试插入记录 A，此时该事务还未提交，然后另一个事务尝试读取到了记录 A。

- 不可重复读：

是指在一个事务内，多次读同一数据。

- 幻读：

指同一个事务内多次查询返回的结果集不一样。比如同一个事务 A 第一次查询时候有 n 条记录，但是第二次同等条件下查询却有 n+1 条记录，这就好像产生了幻觉。发生幻读的原因也是另外一个事务新增或者删除或者修改了第一个事务结果集里面的数据，同一个记录的数据内容被修改了，所有数据行的记录就变多或者变少了。

## 25、MySQL 默认的隔离级别是什么？

MySQL默认采用的 REPEATABLE\_READ隔离级别。

Oracle 默认采用的 READ\_COMMITTED 隔离级别。

## 26、谈谈你对MVCC 的了解？

数据库并发场景：

1. 读-读：不存在任何问题，也不需要并发控制；
2. 读-写：有线程安全问题，可能会造成事务隔离性问题，可能遇到脏读，幻读，不可重复读；
3. 写-写：有线程安全问题，可能会存在更新丢失问题。

多版本并发控制(MVCC)是一种用来解决读-写冲突的无锁并发控制，也就是为事务分配单向增长的时间戳，为每个修改保存一个版本，版本与事务时间戳关联，读操作只读该事务开始前的数据库的快照。

**MVCC** 可以为数据库解决以下问题：

1. 在并发读写数据库时，可以做到在读操作时不用阻塞写操作，写操作也不用阻塞读操作，提高了数据库并发读写的性能；
2. 同时还可以解决脏读，幻读，不可重复读等事务隔离问题，但不能解决更新丢失问题。

## 27、说一下 MySQL 的行锁和表锁？

MyISAM 只支持表锁， InnoDB 支持表锁和行锁，默认为行锁。

表级锁：开销小，加锁快，不会出现死锁。锁定粒度大，发生锁冲突的概率最高，并发量最低。行级锁：开销大，加锁慢，会出现死锁。锁力度小，发生锁冲突的概率小，并发度最高。

## 28 、InnoDB 存储引擎的锁的算法有哪些？

1. Record lock：单个行记录上的锁；
2. Gap lock：间隙锁，锁定一个范围，不包括记录本身；
3. Next-key lock： record+gap 锁定一个范围，包含记录本身。

## 29 、MySQL 问题排查都有哪些手段？

1. 使用 show processlist 命令查看当前所有连接信息；
2. 使用 Explain 命令查询 SQL 语句执行计划；
3. 开启慢查询日志，查看慢查询的 SQL。

## 30 、MySQL 数据库 CPU 飙升到 500% 的话他怎么处理？

1. 列出所有进程 show processlist，观察所有进程，多秒没有状态变化的(干掉)；
2. 查看超时日志或者错误日志（一般会查询以及大批量的插入会导致 CPU与 I/O 上涨，当然不排除网络状态突然断了，导致一个请求服务器只接受到一半。

程序員老蔡