版本号：v1.0 阶段：初稿

# C++面试宝典

MRL Liu

（未经许可，不得传播）

2022年02月02日

如今，计算机行业已经诞生了无数优秀的现代编程语言，但是现在C++却仍然具有强盛的生命力。显然，C++语言具备独特的优势：**和C兼容，可以和硬件设备密切交互，完成各种稀奇古怪的底层功能**。系统级软件是数字世界的基础设施，C++被誉为系统级编程皇冠上的明珠。现代C++版本一般指的是国际C++标准委员会发布的4个版本：C++11/14/17/20。

# 第1章 C++程序篇

## 一、C++的文件

C++大型程序由多个源代码文件（.cpp）组成，不同的文件可能共享一些数据，每个文件都可以独立修改后由C++编译器（g++）单独编译，最终链接成可执行程序文件（.exe）。

C/C++中的变量和函数等都必须先声明后使用，而且C++还有著名的单定义规则（One Definition Rule，ODR），即C++的变量或非内联函数只能有一次定义。C++提供了2种声明：定义声明（简称为定义）和引用声明（简称为声明）。定义声明一般要显示地进行初始化，它会在程序运行时给该变量分配存储空间，引用声明则不会在程序运行时分配存储空间，因为它引用已有的变量。

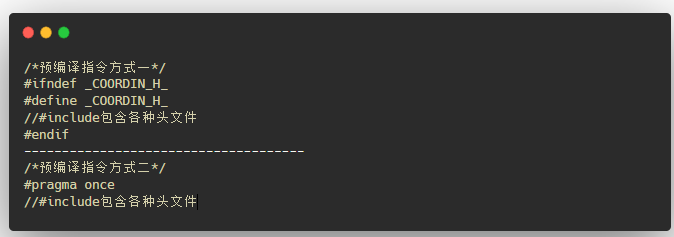
由于不同的源代码文件可能使用同一个声明，为了维护这种声明一致性，C++鼓励程序员将结构声明都放在头文件（.h）中，然后在每个源文件（.cpp）中包含该头文件，这样修改结构声明只需要在头文件修改一次即可。由此，整个C++程序也可以划分为三个部分：

|  |  |
| --- | --- |
| **三类文件** | **文件内容** |
| 头文件（.h） | 函数原型、结构声明、类声明、模板声明、内联函数、使用#define或const定义的符号常量 |
| 主程序文件（.cpp） | 包含main()的文件，程序入口，调用模块文件实现的方法。 |
| 模块文件（.cpp） | 放置一些函数定义，也称为功能模块。 |

### 1、头文件的预编译

头文件中包含的各种声明都不会让编译器分配内存，结构声明、类声明、模板声明、函数原型等不会创建变量，只是告诉编译器如何创建该变量；被声明为const的数据和内联函数有特殊的链接属性。

同一个源文件（.cpp）只能将同一个头文件（.h）包含一次，但是很可能在不知情的情况下将头文件包含多次。为了避免一个头文件在多个.cpp文件中被包含而被多次重复编译，现代C++提供了针对头文件的预编译机制，以尽可能减少反复编译同一个头文件。这种预编译机制如下：



这种预编译指令告知编译器只编译一次包含该头文件的指令，这样头文件就不会被多个源文件重复编译，也可以加快大型C++程序的编译速度。C++中的头文件会被预处理器进行预编译，包含在不同的源文件中。一个源文件（.cpp）和其对应的头文件（.h）会组成一个编译单元。

### 源文件的预处理和主函数

C++程序源文件的后缀是.cpp，一个最简单的.cpp文件如下：



这个最简单的.cpp文件中也包含了三部分：预编译指令、主函数main()。

#### 预处理指令

预处理指令经常用于C++程序文件的开头，用来指定C++预处理器的任务。预处理指令以#开头，常见的预处理指令如#include、#define、#if、#else、#line。这里介绍两种常见的预处理指令。

##### #include指令

#include指令有两种使用方法：

#include<>：一般用于包含系统头文件，预处理程序默认在系统默认目录查找文件。

#include””：一般用于包含用户自定义头文件，预处理程序会现在程序源文件中查找，找不到再找系统默认目录。

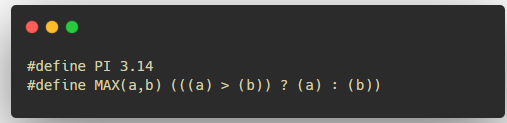
由于历史原因，系统头文件存在几种不同的写法，例如：

#include<math.h>是是C风格旧式写法

#include<iostream>是C++新式风格

##### #define指令

#define经常用于定义一个宏，其为一个标识符定义了宏名，源程序中宏名出现的地方都会用其定义的标识符进行替换，称为宏替换。宏名一般使用大写字母定义。宏经常用于定义一个常量或者简单函数，如下：

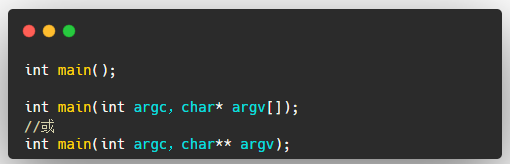


上述示范中用宏定义了一个常量PI，但现代C++中建议使用const进行常量定义，因为宏替换并不会进行类型匹配之类的安全性检查。

上述示范中也用宏定义了一个MAX函数，其好处是没有函数调用的额外开销，运行速度较快，但容易出错。

#### main函数

main函数是C++程序的入口函数，所有C++程序的执行从main函数开始，并由main的返回结束（Windows编程中的DLL除外）。main函数的返回值为一个整数，返回0表示程序正常退出，返回非0表示程序出现异常。main函数的原型如下：

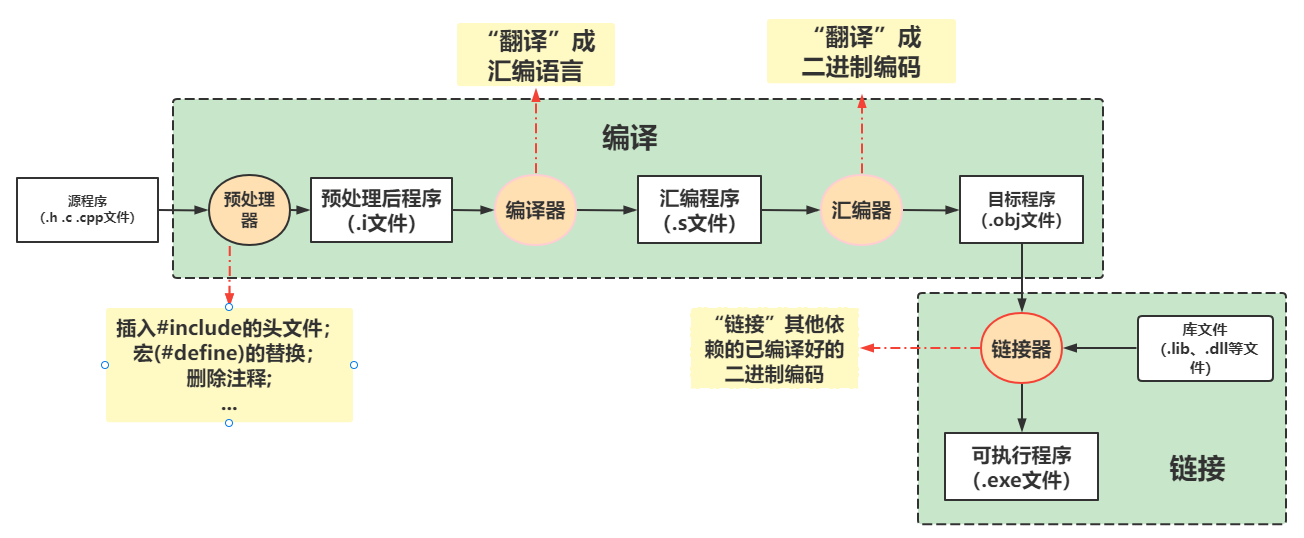


main()函数是C++程序经过特殊处理的函数，其他的返回值类型不是void的函数，如果没有使用return语句，编译器将报错，当main()函数内没有出现return语句时，同样可以通过编译并正常运行，这是因为编译器在main()函数的末尾自动添加了return 0;语句。

main()函数的带参原型可以接收用户向程序输入的参数，其中argc代表参数个数，argv数组中的每一个元素保存命令行参数内容的字符串。

## C++的编译流程

在开发过程中，程序员写的C++源程序文件（.cpp）都要被IDE中集成的C++编译器（g++）编译生成.exe文件，C++编译器就完成了C++的编译流程。C++的编译流程总体可以分为两个步骤：**编译**和**链接**，其过程如下：



#### 编译过程

C++的编译过程首先读取各类源程序文件，头文件（.h）会被预处理器包含进源文件中（.cpp），然后进行宏替换，删除注释等就得到了可以用于编译的预处理后程序文件。预处理后的程序文件会被依次翻译为二进制编码语言，从而得到目标程序文件，此时翻译过程便结束了。

#### 链接过程

编译过程产生的目标程序文件不能直接执行，因为每个目标程序文件都是由一个源文件（.cpp）单独编译过来的，不同的源文件可能相互引用，所以还需要进行C++的链接过程将各个目标程序文件相互链接最终生成可执行程序文件（.exe）。

链接过程有静态链接和动态链接2种方式。从图中也可以看出链接过程有.lib和.dll两种文件。

静态链接就是将各种目标程序文件和静态链接库（.lib文件）链接起来，实际静态链接库也是一组目标程序文件的集合，所有的目标程序文件会被以复制的形式拷贝到最终的可执行程序文件（.exe）中。

动态链接是指一些目标程序文件不会被复制到可执行程序文件（.exe）中，而是存放在动态链接库（.dll文件）中，作为一种可执行程序组件，其不可以单独执行，其必须依赖于对应的可执行程序（.exe）执行。

## C++的内存模型

C++的内存模型决定了C++程序在执行过程中各种数据所在的内存分区。C++的内存分区总体可以划分为栈区、堆区和固定存储区。

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **内存分区** | **内存特点** | **变量类型** | **声明方式** | **生命周期** | **作用域** | **链接性** |
| 栈 | 系统自动维护 | 局部变量  函数参数  局部常量 | 在代码块中 | 自动存储持续性 | 所在代码块 | 无链接性 |
| 寄存器 | 系统自动维护 | 寄存器变量 | 在代码块中，使用关键字register | 自动存储持续性 | 所在代码块 | 无链接性 |
| （静态存储区）  固定存储 | 系统自动维护 | 局部静态变量 | 在代码块中，使用static关键字 | 静态存储持续性 | 所在代码块 | 无链接性 |
| 全局静态变量 | 不在任何函数内，使用static关键字 | 所在文件 | 内部链接性 |
| 全局变量 | 不在任何函数内 | 所在文件 | 外部链接性 |
| （常量存储区）  固定存储 | 全局常量  函数指针 |  |  |  |  |
| 堆 | 程序员手动维护 | 动态变量（new分配的指针变量） | 使用C++运算符new声明 | 动态存储持续性 |  |  |
| 代码区 | 系统自动维护 | 函数体的二进制代码 |  | 静态存储持续性 |  | 外部链接性 |

不同内存分区的数据具有不同的生命周期、作用域和链接性，下面依次讲解。

### 1、数据的生命周期

数据的生命周期指的是数据在内存中的保留时间，即分配内存到回收内存的时间。数据生命周期的不同是因为其定义生命的起始时间可能不一样。

C语言和C++都不允许在一个函数中定义另一个函数，因此函数的生命周期都是静态存储持续期，即从程序开始执行到程序结束。

变量可以在各种代码块中定义，存在不同的生命周期，总体分为如下4种：

|  |  |  |
| --- | --- | --- |
| **生命周期** | **内存分配** | **相关变量** |
| 自动存储持续性 | 函数被执行时分配内存；函数执行完毕，内存释放 | 函数定义内部声明的变量和函数参数 |
| 静态存储持续性 | 程序被执行时分配内存；程序执行完毕，内存释放 | 函数定义外定义的变量和使用关键字static声明的变量 |
| 动态存储持续性 | new运算符被执行时分配内存；delete运算符被执行或程序结束内存释放 | 由new运算符分配的变量 |
| 线程存储持续性 | 生命周期和所属线程一样长（属于并行编程内容） | 使用关键字thread\_local声明的变量 |

### 2、名称的作用域

名称（函数或变量的名字）的作用域描述了名称在文件的多大范围内可见。C++函数的作用域可以是整个类或全局的，不能是局部的，因为函数不能在代码块内定义。

变量的作用域从大到小如下：

命名空间内声明的变量作用域为整个命名空间；源文件内代码块外声明的变量为全局作用域（文件作用域），作用于整个文件；类成员的作用域为整个类；局部变量有局部作用域，作用于所在代码块；函数原型作用域中使用的名称只在包含参数列表的括号中可用。

请注意，因为C++要求函数和变量必须先声明后使用，所以实际作用域都是默认从声明开始到其理论作用域结束。代码块内部的变量的作用域会隐藏代码块外部的相同名称的全局变量的作用域。

### 3、名称的链接性

名称（函数或变量的名字）的链接性描述了名称如何在不同源文件间共享。名称的链接性有三种：

|  |  |  |
| --- | --- | --- |
| **链接性** | **相关含义** | **相关代表** |
| 外部链接性 | 可以在其他源文件访问 | 全局变量（也叫外部变量）、非静态函数 |
| 内部链接性 | 只能在当前文件访问 | 全局静态变量、静态函数 |
| 无链接性 | 只能在当前函数或代码块中访问 | 局部静态变量、局部变量 |

全局变量（外部变量）和函数由于具有外部链接性，可以用来在多文件程序的不同部分之间共享数据。如果要在多个文件中使用全局变量/函数，只需要在一个源文件中包含该变量/函数的定义声明，在其他使用该变量/函数的源文件中使用extern引用声明它即可。

全局静态变量/全局静态函数由于具有内部链接性，可以用来在同一个文件中的多个函数之间共享数据。

局部静态变量由于具有无链接性，其只能用于所在代码块，但由于其静态的生命周期，所以其可以用于再生，即同一函数调用2次，该值第二次调用不会再次初始化而是使用上次计算的值进行计算。

### 4、C++的命名空间

C++11提供了命名空间（namespace），在命名空间中声明的函数和变量都默认具有外部链接性，所以它允许程序的其他部分使用该命名空间中声明的东西，同时，一个命名空间中的名称不会与另一个命名空间的相同名称发生冲突。使用命名空间内部的东西需要添加作用域解析运算符::或者using声明。

命名空间是一个逻辑分组，其用来作为附加信息区分不同库中的相同名称的函数、类和变量等。命名空间的定义需要使用namespace关键字，C++中的关键字都在命名空间std中。

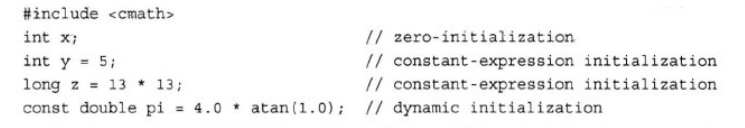
使用其他命名空间的对象时需要在前面加上命名空间的名称，例如使用std::cout。但是为了避免多次添加命名空间前缀的繁琐，C++提供了using namespace指令来告诉编译器，后续代码的关键字将使用指定的命名空间中的名称。

### 5、变量的初始化方式

变量的初始化方式分为静态初始化和动态初始化，其中静态初始化又分为零初始化和常量初始化。静态初始化是值变量在编译时进行初始化，动态初始化是指变量在程序运行时进行初始化。

普通变量必须由程序员显示地初始化，否则会编译报错，即必须进行静态初始化。

静态变量不必须由程序员显示地初始化，所有静态变量在编译时都会自动进行零初始化，如果静态变量还有显示初始化，则看编译是是否可以计算出来，可以则继续进行常量初始化，否则由程序执行时动态初始化。



## C++的说明符和限定符

### 1、常见说明符

关键字auto在C++11之前用于将变量声明为自动变量，在C++11之后用于自动类型推断。

关键字register用于在声明中知名函数用寄存器存储，提高访问速度，在C++11之后只是显示地指出该变量是自动的。

关键字static用于代码块外部声明时表示该变量具有内部链接性，用于代码块内部声明时表示该局部变量的存储持续性为静态的。

关键字extern表明该声明是引用声明，即引用其他源文件中定义的外部变量。

关键字thread\_local用于表明该变量的生命周期与所属线程相同，可以与static或extern结合使用。

### 2、常见限定符

关键字const用于表明变量的内存空间被初始化后程序不能再次修改它，如果const修饰全局变量，则将会将全局变量的链接性转换为内部，表明这是该文件独享的一组常量，但是其他文件可以继续使用extern来覆盖它的内部链接性。



关键字volatile用于表明硬件可以修改该变量的内存空间，即使程序不修改它。例如程序访问同一个变量2次时编译器一般会提前检查其值是否被程序更改过，如果程序没有更改它而自己发生了变化则认为需要优化，而该关键字避免编译器做这样的优化。

关键字mutable用来表明即使结构或类变量被整体声明为const，其某个使用了mutable的变量成员可以不受const限制继续修改。

## C++的常见面试题型

### 简述为什么要使用头文件、头文件的预编译机制。

### 简述#include<>和#include””的区别。

### 简述C++的编译流程和内存模型。

### 简述下全局变量、外部变量、静态变量的区别。

### 简述下常见说明符的作用或对比相近的说明符。

# 第2章 C++语法篇

## 一、变量

C++中有常量、数组、字符串、指针、引用、类、结构体、联合体、枚举等各种概念，但是它们**本质上都是变量**，是的，你没听错，**编程语言中的常量也是一种特殊的变量（使用const关键字）**。

变量是什么？**变量本质是程序可操作的内存空间的名称，每个变量概念都由type和value组成**，其中type就是其**数据类型**，其决定了数据在内存中的存储方式和运算方式，**常量本质是定义后不可更改value的变量**。所以准确理解了变量概念，就相当于拿到了C++中的各种变量类型的总把手。

变量的value没什么好说的，但是变量的type可是五花八门，不同的type在内存中的大小也不一样。**C++中的变量必须先声明后使用**。

这里先简单说一些基本变量类型、指针和引用。

### 基础变量

基础变量就是变量的type为编程语言内置数据类型的变量，这些类型很常见，但是我们要**记住每种变量所占的内存字节数**。可以使用一个或多个类型修饰符进行修饰：signed、unsigned、short和long。

#### （1）整型变量

布尔型（bool）、字符型（char）和整数型（int）一起合称为整型，他们可以方便地进行转换。

bool变量与整数变量可以进行隐式转换：true具有值1，false具有值0；非零整数转换为true，而0转换为false。

char变量占据一个字节数，因此它可以保存2^8=256种不同的值，其默认和ASCII字符集进行对应，因此int（c）可以将一个char变量c显示转换为ASCII值。char变量默认是无符号（unsigned），unsigned char表示的值是0~255，signed char表示的值是-128~127。wchar\_t用来保存更大的字符集里的字符。

int默认是有符号的（signed），这样默认的int变量就可以表示正数和负数，unsigned int是无符号整数，其只能用来表示整数，一般不常用。int一共有3种大小的用法，short int（简写为short）；int；long int（简写为long）。

#### （2）浮点型变量

浮点数也有3种精度大小的：float（单精度）、double（双精度）和long double（扩展精度）。

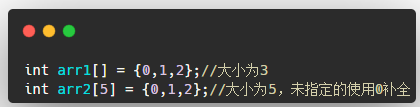
#### 不同变量的大小

C++对象的大小是用char大小的倍数表示的，所以char的大小一般为1，一个C++对象的字节数可以根据运算符sizeof得到。每一种数据类型实际的内存大小与其系统位数有关，下表列出的是目前为主的64位系统，一字节为8位。

|  |  |  |
| --- | --- | --- |
| **关键字** | **数据类型** | **内存大小** |
| void | 无类型 | —（没有void对象，只有void指针） |
| bool | 布尔型 | 1个字节 |
| char | 字符型 | 1个字节 |
| wchar\_t | 宽字符型 | 2或4个字节 |
| int | 整型 | 4个字节 |
| float | 浮点型 | 4个字节 |
| double | 双浮点型 | 8个字节 |

### 数组

C++中的数组必须在定义时就指定大小，C++中的数组初始化方式如下：



### 字符串

这里着重介绍下C++中的字符串，首先介绍下**字符串常量**。在C++中一个**单引号**用于单个**字符常量**（‘a’），一个**双引号**用于单个**字符串常量**（“A”）。

字符常量本质上就是一个固定value大小的字符变量（‘a’的类型是const char[1]），字符串常量本质上是末尾加了一个空字符’\0’的cost字符数组（“A”的类型是const char[2]）。特别地，一个字符串常量可以给一个char指针赋值：



C++中有两种表示字符串的形式：C风格的字符数组和C++引入的string类。string类本质上也是封装了C风格的字符数组，但是其也集成了许多字符串的常见操作，是C++推荐的表示字符串的方式。

数组类型

### 3、枚举类型

枚举类型可以保存一组定义的整数常量，其用法一般如下：



数据结构和类

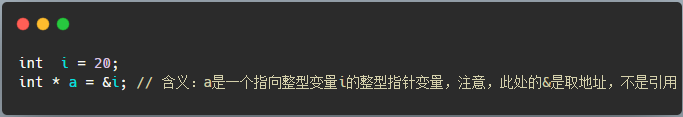
### 指针和引用变量

接下来我们介绍下**指针**和**引用**，这是面试的高频知识，这里我们只强调一些关键信息。

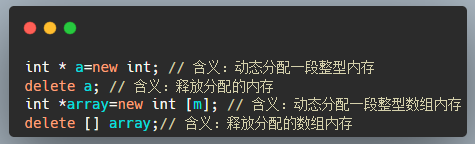
#### （1）指针

**指针本质上是一种特殊的变量，特殊在其value是另一个变量的地址，所以无论何种类型的指针，其内存大小都是固定的**（例如在X86中为4个字节，在X64中为8个字节）；**指针可以在任何时候初始化或者更改；存在空的指针**。

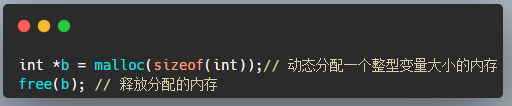
指针的定义如下：



但是实际编程中程序员不会这么无聊地用，基本都是new一个类对象，然后访问其属性和方法。new是一种操作符，和delete结合使用：



当然除了new和delete，也可以使用标准库函数malloc()和free()：



new和malloc()的主要区别：

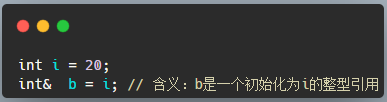
|  |  |  |
| --- | --- | --- |
|  | **new/delete** | **malloc()/free()** |
| **本质** | 运算符 | 标准库函数 |
| **分配大小** | 自动计算 | 作为参数手动输入 |
| **主要操作** | 分配/释放内存时调用对象的构造和析构函数 | 分配/释放内存时不调用对象的构造和析构函数 |
| **安全性** | **内存分配失败**时抛出bad\_alloc异常 | **内存分配失败**时返回null |
| **进行类型识别检测**，返回定义时具体类型指针，如果为int指针分配float变量时报错 | **不进行类型识别检测**，返回void指针，需要手动进行类型转换，如果为int指针分配float大小的字节数不报错。 |

总结，new封装了malloc()方法，new一般用来返回一个类对象，malloc(）一般用在基本变量（int、float），前者更加常用。

#### （2）引用

**引用**可能初学者不经常使用，但是没关系，**引用本质就是一个已存在的变量的别名，所以引用的内存分配区域和大小由其指向的变量类型决定**。需要注意的是**引用必须在创建时初始化，之后不可更改或再次初始化；不存在空的引用；这两点与指针很不同**。

引用的定义：



引用一般用在函数的参数和返回值上：



## **常量**

前文我们已经知道，常量本质上就是使用了关键字const的变量，使用了const的变量在其作用域内都不能更改其value，const的最常见用途是作为数组的最大界和分情况标号。

## 二、输入输出

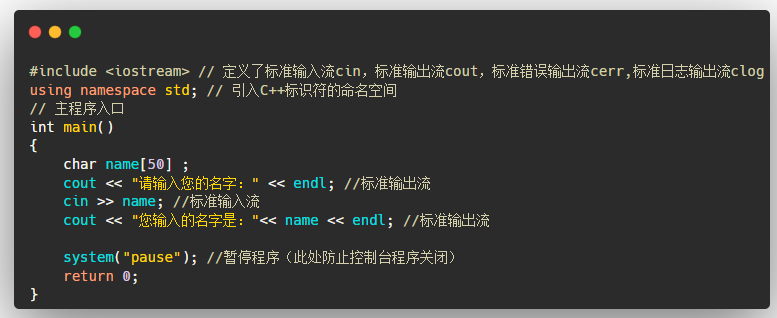
C++标准库提供了丰富的**I/O**（输入/输出）功能。首先看下C++编程中最基本和最常见的**I/O操作**。C++的I/O发生在**流**中，**流**是**字节序列**。

**输入操作**被C++看做**字节流**从**设备**（如键盘、磁盘驱动器、网络连接等）流向**内存**；

**输出操作**被C++看做**字节流**是从**内存**流向**设备**（如显示屏、打印机、磁盘驱动器、网络连接等）。

### 标准输入输出流

下面给出标准输入输出流的演示代码：



首先要用#include语句包含头文件<iostream>；using namespace std语句负责引入C++标识符的命名空间，没有该语句的话，则以下cout、cin和endl都得换成std::cout、std::cin和std::endl。cout可以直接输出字符串或字符数组，只在输出endl时换行；cin以阻塞的方式从控制台读取数据到字符数组中。

# 第3章 C++抽象篇

C++语言是面向对象语言，面向对象程序设计的四大特性：抽象、封装、继承和多态。

### 一、类的成员

C++语言通过类的概念来实现抽象和封装。一个类（class）中包含了成员变量和成员函数，一般C++的结构体（struct）中只有成员变量，不包含大量成员函数。

#### （1）类的访问权限

class通过private、public、protect三个关键字来实现类成员的访问控制，class的成员默认都是private类型，struct的成员默认都是public类型，这也是两者最大的不同。

class的private成员只准类本身内部的成员函数访问；

class的public成员可供类本身、派生类及外部访问；

class的protect成员可以被类本身以及派生类访问，外部不可访问；

#### （2）类的构造函数和析构函数

类的构造函数和析构函数一直被其开发者认为是C++的核心。

C++中类的构造函数具有与类相同的名字，输入参数可以自定义也可以不定义，其由C++程序自动调用，不需要程序员显示调用。一个类通常具有多个构造函数，从而为用户提供多种不同的初始化方式。在所有构造函数中，有一类特殊的复制构造函数，其特别之处在于传入的参数是一个自身类型的参数。包含const成员的类不能自动调用默认构造函数。

析构函数

#### （3）类的静态成员

一个类可以实例化出多个对象，不同的对象占据不同的内存。在类成员的声明中使用关键字static就可以定义类的静态成员，不同对象的所有static成员占据同一份内存。

#### （3）类的常量成员函数

在类的成员函数声明的参数表后面使用关键字const就变成了常量成员函数，常量成员函数不允许其修改类本身的变量。

成员函数

### 二、类的继承

继承是面向对象程序设计中最重要的一个概念，代表了is a关系。从一个类中派生出另一个类中，原始类称为基类（或父类），继承类称为派生类（或子类）。

子类可以访问父类中所有的非私有成员，并继承所有的父类方法，以下情况除外：

1. 父类的构造函数、析构函数和复制构造函数。
2. 父类的重载运算符。
3. 父类的友元函数。

类的继承有三种方式（在大部分情况下几乎都不使用protected和private）：

1. 公有继承（public）。父类的公有成员成为子类的公有成员，父类的保护成员成为子类的保护成员，子类无法直接访问父类的私有成员，只能通过父类的公有和保护成员函数来间接访问。
2. 保护继承（protected）。父类的公有成员和保护成员都会成为子类的保护成员。
3. 私有继承（private）。父类的公有成员和保护成员都将成为子类的私有成员。

### 三、类的重载

C++允许同一个作用域中的某个函数或运算符具有多个定义，这也叫做函数重载和运算法重载。重载声明是指该作用域内声明的函数或方法具有相同名称，但是参数列表和实现不相同。C++编译器会自行选用最合适的定义，即重载决策。

### 三、类的多态

类的多态指的是调用类的成员函数时会根据调用函数的对象的类型来执行不同的函数。虚函数是在父类中使用关键字virtual声明的函数。假设父类有一个方法area()，子类也有一个相同的方法area()。

假设父类的方法area()没有声明为虚函数，则调用子类的area()时会默认执行父类的area()，这就是所谓的静态多态（或静态链接），函数调用在程序执行前就准备好了。有时候这也被称为早绑定，因为area()函数在程序编译期间就已经设置好了。

假设父类的方法area()声明为虚函数，则调用子类的area()时会默认执行子类的area()，这就是所谓的动态链接，或后期绑定。虚函数的声明会告诉编译器不要静态链接到该函数。

虚函数必须进行实现，且子类不一定需要覆盖该函数；纯虚函数（也叫抽象函数）则不需要实现，且子类一定需要覆盖该函数。

在上述举例说明中，子类通过将父类方法声明为虚函数就可以在子类覆盖其实现，总结起来，子类方法想要覆盖父类方法需要满足如下要求：

一虚：父类中被覆盖的成员函数必须声明为虚函数。

二容：父类和子类的成员函数的返回类型和异常规格必须兼容

四同：父类和子类的成员函数的名程、形参类型、常量属性和引用限定符必须完全相同。

override是C++11中添加的安全声明的关键字，子类要覆盖的方法可以添加override关键字来声明，编译器将会对该函数和父类的对应函数进行一虚二容四同的比较，错误的地方会报错。注意，不添加override关键字也是可以的，但是出现问题时编译器可能不会报错。

在子类中调用父类的方法：父类名::方法名

# 数据结构篇

## 数组

数组很可能是最简单也最常见的数据结构之一，数组是**顺序存储结构**的代表，其是静态分配一段规定内存，数组初始化时必须指定其长度，在**查找**和**修改**时效率高。C++中存在字符和字符串的区别，字符串被看做是字符数组。

本节介绍数组中常见的LeetCode类型，部分题型可能会涉及到字符串的处理。

### STL中数组的用法

本小节主要介绍C++中动态数组vector的使用，其在LeetCode题目中的使用也非常高，请务必掌握。其中包含了vector对象的5种初始化方式、三种遍历写法、插入、删除、获取、排序、反转和分割的操作。

|  |  |
| --- | --- |
| 初始化 |  |
| 遍历方式 |  |
| 插入方式 |  |
| 删除方式 |  |
| 获取方式 |  |
| 排序方式 |  |
| 反转方式 |  |
| 分割数组 |  |

### 2、进制转换

本节介绍LeetCode数组相关题目中的进制转换类型的题目，其基础知识是**10进制转换为k进制**和**k进制转换为10进制**2种类型。在介绍进制转换前，先简单介绍下C++提供的字符串类string，可以将其看做是一个存储成员类型为char的vector，除了string.length()和vector.size()不同外，其他API的使用基本一致。

由于C++中默认的整数都是10进制，所以LeetCode题目中一般用vector或string存储转换后的k进制的数。所以其中可能会用到整数、字符、字符串的转换知识，如下：

|  |  |
| --- | --- |
| **整数和字符的相互转换** | **整数和字符串的相互转换** |
|  |  |

整数和字符可以通过加减进行相互转换，因为字符也属于整型，其参与四则运算时使用ASCII值，a’的ASCII值是97，‘A’的ASCII值是65，‘0’的ASCII值是48。

#### （1）进制转换的模板

|  |  |
| --- | --- |
| **理论依据** | **进制转换模板** |
|  |  |
|  |  |

#### （2）二进制转换的题目

|  |  |  |
| --- | --- | --- |
| **类型** | **解题思路** | **LeetCode题目** |
| **2进制转10进制** | 将链表添加到数组，遍历2进制数组转换为10进制 | **1290. 二进制链表转整数（简单难度）** |
| **10进制转2进制** | 10进制转换为2进制，然后遍历字符串统计字符1的个数 | 1. **位1的个数（简单难度）**   **剑指 Offer 15. 二进制中1的个数（简单难度）** |
| **10进制转2进制** | 遍历将n个数依次从10进制转换为2进制，然后遍历字符串统计字符1的个数 | 1. **比特位计数（简单难度）**   **剑指 Offer II 003. 前 n 个数字二进制中 1 的个数（简单难度）** |
| **10进制转2进制** | 10进制转换为2进制，然后遍历字符串查看相邻元素是否相同 | **693. 交替位二进制数（简单难度）** |
| **10进制转2进制2进制转10进制** | 10进制转换为2进制，然后遍历字符串将0和1字符替换，再从2进制转换为10进制。 | 1. **数字的补数（简单难度）**   **1009. 十进制整数的反码（简单难度）** |

#### （3）非二进制转换的题目

|  |  |  |
| --- | --- | --- |
| **类型** | **解题思路** | **LeetCode题目** |
| **10进制转7进制** | 套用模板，注意负数要先转换为正数，转换后加负号 | **504. 七进制数（简单难度）** |
| **10进制转16进制** | 套用模板，注意16进制中大于9的数要使用小写字母 | **405. 数字转换为十六进制数（简单难度）** |
| **26进制转10进制** | 套用模板，注意大写字母和数字转换 | **171. Excel 表列序号（简单难度）** |
| **10进制转26进制** | 套用模板，注意该题整数是1-26，每一位处理前需要-1 | **168. Excel表列名称（简单难度）** |

### 3、数位分离

数位分离就是将一个整数的各个位分离出来进行相关操作，常见的方法有两种，一种是直接将整数转换为string，然后对string进行操作；一种是通过10进制转（k=10）进制的模板拿到各个位。**判断一个数有多少位，只需要和10、100等比较即可（编程技巧）**。

|  |  |  |
| --- | --- | --- |
| **类型** | **解题思路** | **LeetCode题目** |
| **数位分离后累加** | 不断数位分离后累加直至一位数 | **1281. 整数的各位积和之差（简单难度）** |
| **数位分离后累加至1位数** | 不断数位分离后累加直至一位数（判断1位数可以通过和10比较） | **258. 各位相加（简单难度）** |
|  |  | **1945. 字符串转化后的各位数字之和（简单难度）** |
|  |  | **[1837. K 进制表示下的各位数字总和（简单难度）](https://leetcode-cn.com/problems/sum-of-digits-in-base-k/)** |
|  |  | **[9. 回文数（简单难度）](https://leetcode-cn.com/problems/palindrome-number/)** |

## 链表

链表可能是面试中被提及的最频繁的数据结构之一，其实现的代码量较少，而且需要对指针进行操作，可以很好地考察应聘者的编程功底。

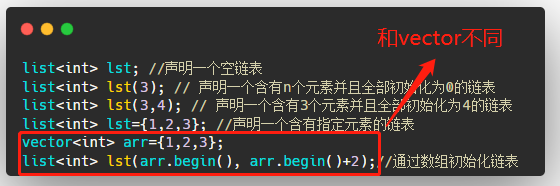
【**链表的特点**】链表是一种**链式存储结构**，在**插入**和**删除**效率上比顺序存储结构要高得多，但在**查找**和**修改**效率上不如**顺序存储结构**。链表创建时无需知道链表的长度，可以在每次插入新节点动态分配内存。

【**链表的种类**】链表大致可以分为单链表和双链表2种，单链表即单向链表，双链表即双向链表。在C++的STL中默认提供的是**双链表**<list>，LeetCode题目中考察最多的是**单链表**（因为单链表操作不方便，更容易考察编程功底）。

### 1、STL中链表的用法

首先介绍下C++中STL的链表使用方法，C++通过<list>文件来提供链表的数据结构操作，<list>是一个双向链表，其标准的用法如下：

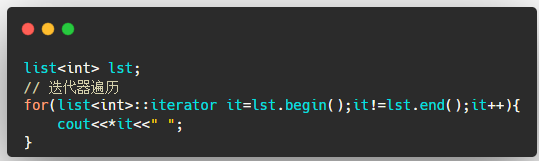
#### list的5种初始化方式



以上5种初始化方式中，第5中通过迭代器iterator的初始化方式和vector不同，其只能使用vector的迭代器进行初始化，请注意，list.begin()+2或者list.end()-1是违法的，会报错。（list.begin()指向）

#### list的遍历方式

list几乎只能通过迭代器遍历，以下是其遍历方式，注意练习：



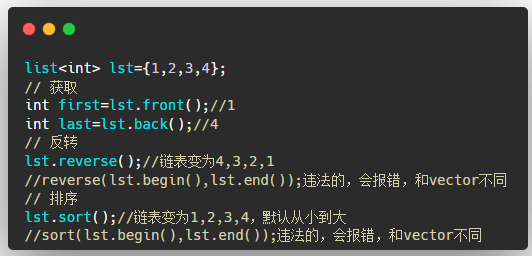
其中lst.begin()指向链表的第一个元素，lst.end()指向链表的最后一个元素的下一个，为空。

#### list的3种插入和删除方式

|  |
| --- |
|  |
|  |

请注意迭代器it的操作，只能it++,不能it+5，这点和vector的迭代器很不一样。

#### list的获取、反转、排序方式



### 2、单链表和双链表的实现

本小节介绍下单链表和双链表的实现方法，以表格的形式进行对比。在面试中很可能让手撕单链表或双链表的实现，**所以请读者一定在业余时间勤加练习**。

|  |  |  |
| --- | --- | --- |
|  | 单链表 | 双向链表 |
| 节点类型 |  |  |
| 插入操作 |  |  |
| 删除操作 |  |  |
| 获取操作 |  |  |

在上述插入、删除和获取的一般操作中，最关键的代码其实是**如何遍历找到第index个元素，这个编程技巧非常重要**，上述红框中的代码其实就是**计数法**，其在LeetCode题目中也会经常用到。**在链表中如何找到第index个元素，**有2种方法：**计数法**和**双指针法**：

计数法适用于index容易求得的情形，例如正数/倒数第k个，中间节点等。

双指针法使用于index不容易求解的情形，可以通过设置快慢指针的不同速率来达到找指定区间、判断环等复杂问题。以下将会在leetcode题目中进入深刻体会。

### 删除链表中的值

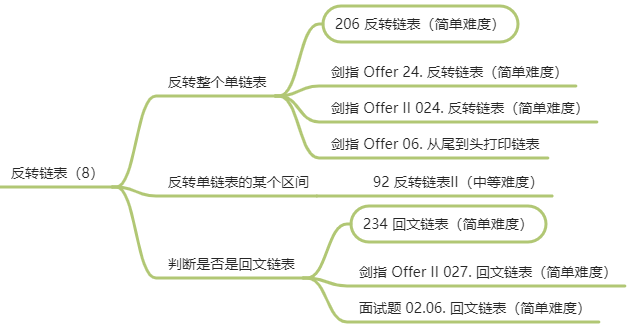
LeetCode中删除链表的题目一般是操作单链表，要删除某个元素就必须定位到待删除元素的上一个节点，修改上一个节点的next指针。



|  |  |  |
| --- | --- | --- |
| **类型** | **解题模板** | **相关LeetCode题目** |
| **删除单链表中的某个特定值元素/相同元素（双指针）** |  | **203 移除链表元素（简单难度）** |
| **剑指 Offer 18. 删除链表的节点（简单难度）** |
| [83. 删除排序链表中的重复元素](https://leetcode-cn.com/problems/remove-duplicates-from-sorted-list/)（简单难度） |
| **删除链表中的一段特殊序列（动态指针区间）** |  | **82. 删除排序链表中的重复元素II（中等难度）** |
|  |  | **1171. 从链表中删去总和值为零的连续节点（中等难度）** |
| **删除单链表中的倒数第N个节点/正中间节点（遍历到指定位置）** |  | **剑指 Offer 22. 链表中倒数第k个节点（简单难度）** |
| **19 删除链表的倒数第N个节点（中等难度）** |
| **剑指 Offer II 021. 删除链表的倒数第 n 个结点（中等难度）** |
| **2095. 删除链表的中间节点（中等难度）** |
| **只给出待删除元素，实现删除效果（类似于脑筋急转弯）** |  | **237 删除链表中的节点（简单难度）** |
| **面试题 02.03. 删除中间节点（简单难度）** |

### 4、反转链表

反转链表是LeetCode题目中的高频知识点，请务必掌握，由于LeetCode题目中的链表题目节点是自定义的，**无法直接调用反转函数，常常需要自己手写**。

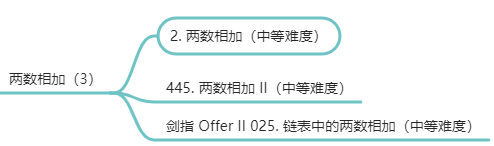


相关LeetCode题目有：

|  |  |  |
| --- | --- | --- |
| **类型** | **解题模板** | **LeetCode题目** |
| **反转整个单链表** |  | **206 反转链表（简单难度）** |
| **剑指 Offer 24. 反转链表（简单难度）** |
| **剑指 Offer II 024. 反转链表（简单难度）** |
| **剑指 Offer 06. 从尾到头打印链表** |
| **反转单链表的某个区间[left,right]** |  | **92 反转链表II（中等难度）** |
| **判断是否是回文链表（反转后半部分，比较前后两半部分）** |  | **234 回文链表（简单难度）**  **剑指 Offer II 027. 回文链表（简单难度）**  **面试题 02.06. 回文链表（简单难度）** |

### 5、链表元素相加

**用链表元素相加不能将链表中的元素提取出来组成一个数再相加**，因为链表的元素数量可能是非常大的，转换成数字相加可能会超过int或long的大小限制。所以推荐的思路是：同时遍历2个链表来相加，这个模板也可以用来**解决大数相加问题**。



|  |  |  |
| --- | --- | --- |
| **类型** | **解题模板** | **LeetCode题目** |
| **2个链表的元素相加** |  | **2. 两数相加（中等难度）** |
| **445. 两数相加 II（中等难度）** |
| **剑指 Offer II 025. 链表中的两数相加（中等难度）** |

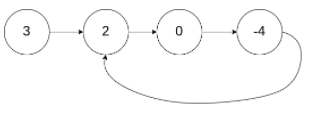
### 6、链表相交和环形链表

#### （1）找到2个单链表的的相交节点

|  |  |  |
| --- | --- | --- |
| **类型** | **解题模板** | **LeetCode题目** |
| **2个单链表的相交节点** | 如下图，找到A和B两个节点的第一个相交节点（c1）：    思路：分别求出lenA和lenB，如果lenA>lenB，则让遍历指针pA先走lenA-lenB个节点，然后和pB一起走，当pA==pB时则找到了相交节点。 | **[160. 相交链表（简单难度）](https://leetcode-cn.com/problems/intersection-of-two-linked-lists/)** |
| **[面试题 02.07. 链表相交（简单难度）](https://leetcode-cn.com/problems/intersection-of-two-linked-lists-lcci/)** |
| **[剑指 Offer 52. 两个链表的第一个公共节点（简单难度）](https://leetcode-cn.com/problems/liang-ge-lian-biao-de-di-yi-ge-gong-gong-jie-dian-lcof/)** |
| **[剑指 Offer II 023. 两个链表的第一个重合节点（简单难度）](https://leetcode-cn.com/problems/3u1WK4/)** |

#### （2）判断1个单链表的环和环的入口

如下图，找到单链表的环的入口（2）：



如果链表中有某个节点，可以通过连续跟踪 next 指针再次到达，则链表中存在环。使用快慢指针法，分别定义fast和slow指针，

（1）都从头结点出发，fast指针每次移动两个节点，slow指针每次移动一个节点，如果出现fast==slow，说明这个链表有环。

（2）当fast==slow，此时两个指针分别从head和相遇点以相同速度同时出发，再次相遇的节点即是环的入口。

|  |  |  |
| --- | --- | --- |
| **类型** | **解题模板** | **LeetCode题目** |
| **判断1个单链表的环和环的入口** |  | **[141. 环形链表（简单难度）](https://leetcode-cn.com/problems/linked-list-cycle/)** |
| **[142. 环形链表 II（中等难度）](https://leetcode-cn.com/problems/linked-list-cycle-ii/)** |
| **[剑指 Offer II 022. 链表中环的入口节点（中等难度）](https://leetcode-cn.com/problems/c32eOV/)** |

### 7、合并链表

|  |  |  |
| --- | --- | --- |
| **类型** | **解题模板** | **LeetCode题目** |
| **判断1个单链表的环和环的入口** |  | **[141. 环形链表（简单难度）](https://leetcode-cn.com/problems/linked-list-cycle/)** |
| **[142. 环形链表 II（中等难度）](https://leetcode-cn.com/problems/linked-list-cycle-ii/)** |
| **[剑指 Offer II 022. 链表中环的入口节点（中等难度）](https://leetcode-cn.com/problems/c32eOV/)** |

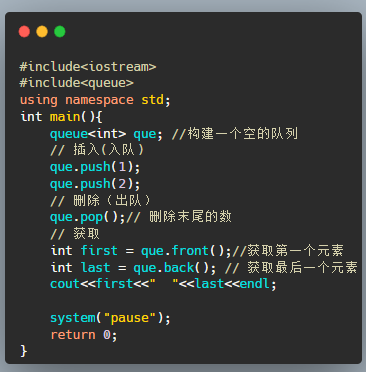
### 8、排序链表

用插入排序给链表排序

用归并排序给链表排序：

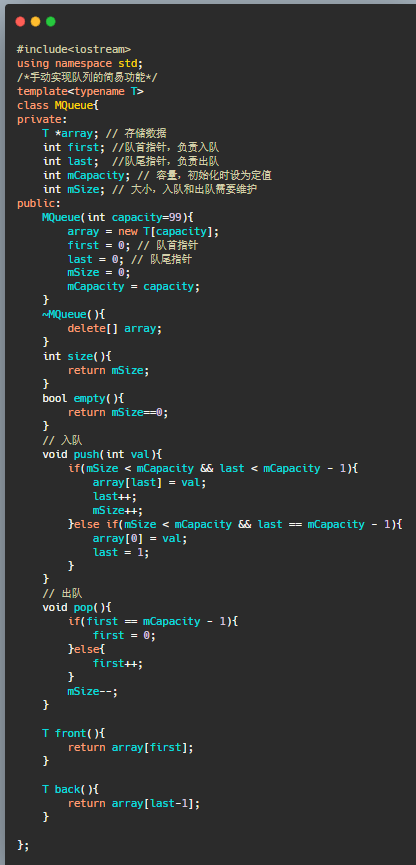
## 三、队列

本小节介绍首先介绍队列的使用，C++中可以直接使用已有的类：



### 用数组实现队列功能

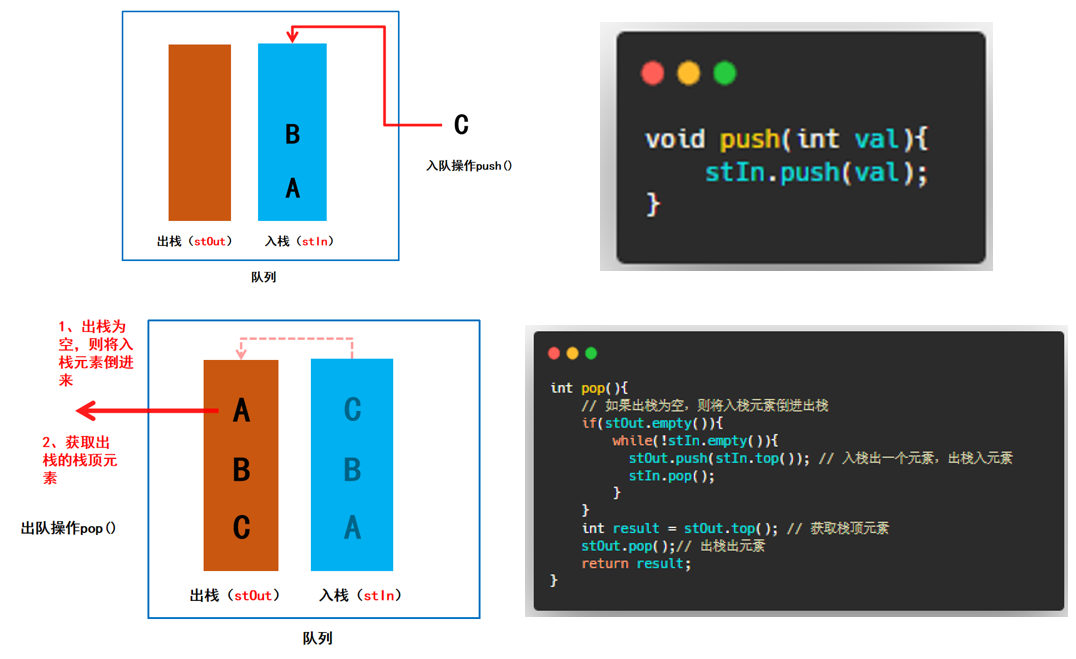
我们可以使用一个固定数组和指针来实现一个队列的入队和出队操作，其内部需要两个指针：队首指针和队尾指针，分别辅助实现出队和入队操作。



在实际做LeetCode题的时候，可以结合vector来快速实现队列的功能，但是结合vector等数组的时间复杂度较高，不适合频繁的插入操作，这个时候就需要结合链表来实现。

### 用栈实现队列功能

在leetCode中经常有用2个栈实现队列的题目，本小节介绍用栈模拟队列的思路。栈的“先进后出”的性质类似于一杯水，这里介绍用栈模拟的队列的入队、出队。假设队列内部有两个栈，分别叫做入栈stIn和出栈stOut。



全部的代码如下，注意获取队首元素和队尾元素的方法：



|  |  |  |
| --- | --- | --- |
| **LeetCode题目** | **相关链接** | **备注** |
| 232 用栈实现队列（简单难度） | [232. 用栈实现队列 - 力扣（LeetCode） (leetcode-cn.com)](https://leetcode-cn.com/problems/implement-queue-using-stacks/) |  |
| 面试题 03.04. 化栈为队（简单难度） | [面试题 03.04. 化栈为队 - 力扣（LeetCode） (leetcode-cn.com)](https://leetcode-cn.com/problems/implement-queue-using-stacks-lcci/) |  |
| 剑指 Offer 09. 用两个栈实现队列（简单难度） | [剑指 Offer 09. 用两个栈实现队列 - 力扣（LeetCode） (leetcode-cn.com)](https://leetcode-cn.com/problems/yong-liang-ge-zhan-shi-xian-dui-lie-lcof/submissions/) | 出队操作主要注意判断队列是否为空 |

### 实现双端队列功能

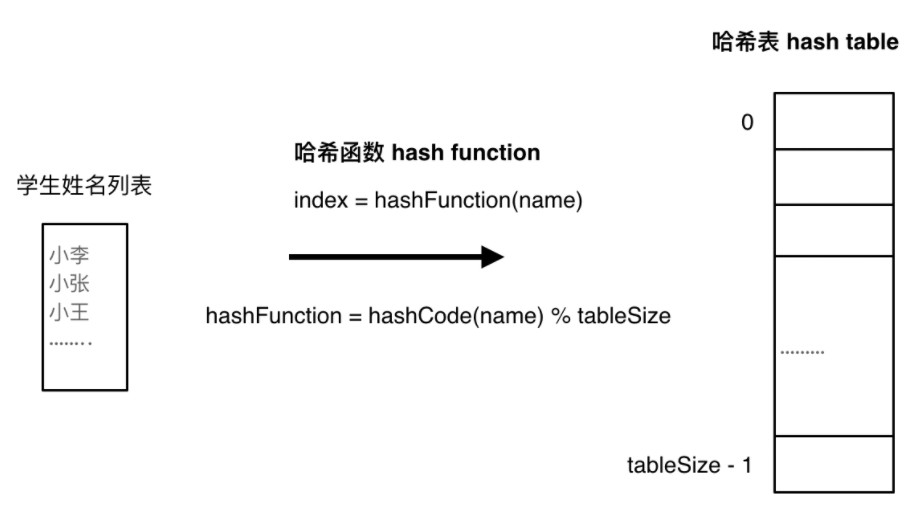
|  |  |  |
| --- | --- | --- |
| **LeetCode题目** | **相关链接** | **备注** |
| 232 用栈实现队列（简单难度） | [232. 用栈实现队列 - 力扣（LeetCode） (leetcode-cn.com)](https://leetcode-cn.com/problems/implement-queue-using-stacks/) |  |
| 面试题 03.04. 化栈为队（简单难度） | [面试题 03.04. 化栈为队 - 力扣（LeetCode） (leetcode-cn.com)](https://leetcode-cn.com/problems/implement-queue-using-stacks-lcci/) |  |
| 剑指 Offer 09. 用两个栈实现队列（简单难度） | [剑指 Offer 09. 用两个栈实现队列 - 力扣（LeetCode） (leetcode-cn.com)](https://leetcode-cn.com/problems/yong-liang-ge-zhan-shi-xian-dui-lie-lcof/submissions/) | 出队操作主要注意判断队列是否为空 |

## 四、栈

## 五、哈希表

哈希表（hash table）也叫散列表，是根据关键值而直接进行访问的数据结构，即key哈希映射value，这种映射关系通过hash function（哈希函数）来实现。

哈希函数的内部原理：假设要将若干名字存入一张哈希表中，输入的name是一个字符串，其通过hashCode（name）%tableSize被转换为数值，这样学生名字就映射为哈希表上的索引数字，%tableSize可以确保映射出来的索引数值都落在哈希表上。



如果学生的数量超过哈希表的大小就需要一些解决哈希冲突的机制，例如拉链法。

### unordered\_map的用法

# 排序算法篇

排序算法是最基本的算法之一。排序即对一系列对象根据某个关键字进行排序。首先介绍一下排序术语：

|  |
| --- |
| **稳定**：a=b，排序前a在b之前，排序后a仍然在b之前  **不稳定**：a=b，排序前a在b之前，排序后a可能在b之后  **内排序**：所有排序操作都可以在内存中完成  **外排序**：当数据太大时需要将数据放在磁盘中  **时间复杂度**：一个算法执行所耗费时间的估计值  **空间复杂度**：一个算法执行所耗费空间的估计值 |

排序算法分类如下：

根据排序方式，排序算法整体上可以划分为内部排序和外部排序。内部排序是指数据记录在内存中进行排序，不需要申请额外的存储空间。外部排序是指在排序过程中需要申请额外的内存空间。

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **排序方式** | **排序算法** | **最好情况** | **最坏情况** | **平均时间复杂度** | **空间复杂度** | **稳定性** |
| In-place | 选择排序 |  |  |  |  | 不稳定 |
| 冒泡排序 |  |  |  |  | 稳定 |
| 插入排序 |  |  |  |  | 稳定 |
| 快速排序 |  |  |  |  | 不稳定 |
| 希尔排序 |  |  |  |  | 不稳定 |
| 堆排序 |  |  |  |  | 不稳定 |
| Out-place | 归并排序 |  |  |  |  | 稳定 |
| 计数排序 |  |  |  |  | 稳定 |
| 桶排序 |  |  |  |  | 稳定 |
| 基数排序 |  |  |  |  | 稳定 |

（注意：n指的是数据规模；k指的是“桶”的个数；In-place指的是占用常数内存，不占用额外内存；Out-place指的是占用额外内存）

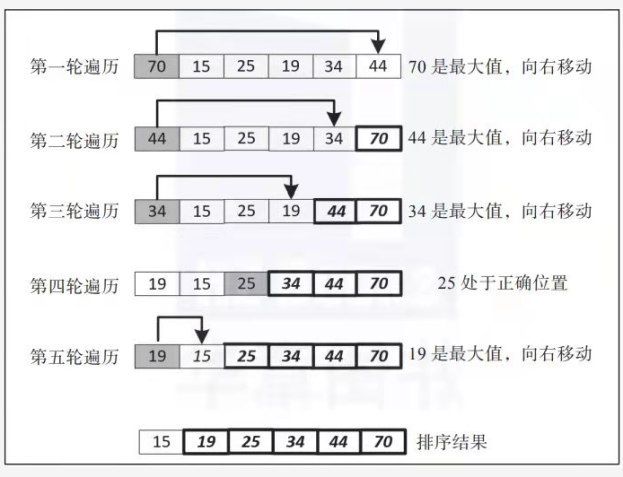
本章节接下来的测试代码可以在Leetcode 912题上编译运行，其中希尔排序（三分之一增量版本）、堆排序、归并排序不会超时。

## 一、内排序

### 1、选择排序

选择排序（selection-sort）是性能鲁棒性最好的排序算法之一，任何情况下耗费时间都为，而且不占用额外内存，适合于数据规模n较小的情景。

**【算法思路】**：从未排序序列中找到最小（大）元素，存放到排序序列的末尾位置，直至未排序序列为空。



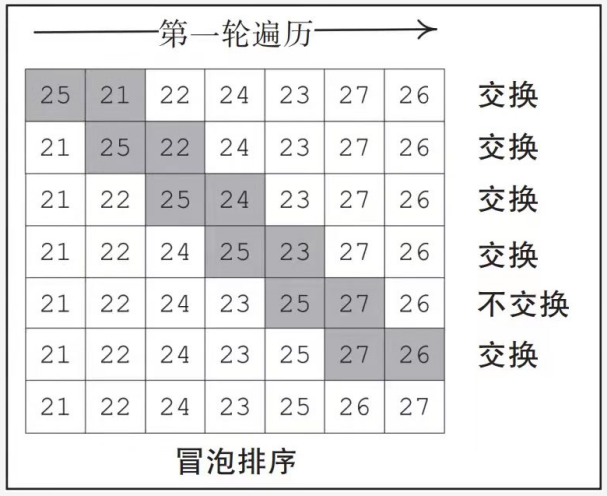
**【编码思路】：**双层循环，外层循环负责从左至右确定已排序序列，内层循环负责从左至右在未排序序列中查找最值，找到后交换两数即可。



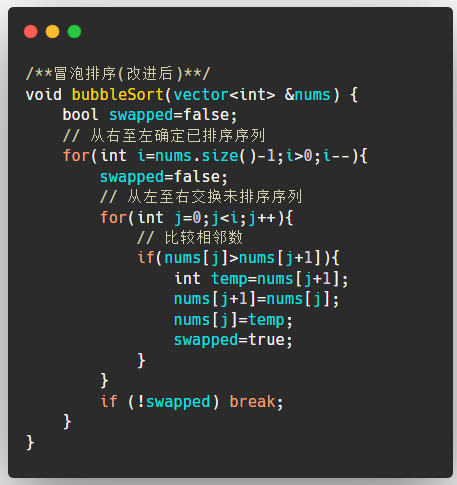
### 2、冒泡排序

冒泡排序（bubble-sort）比选择排序性能稍好一定，而且稳定，排序不影响相同数的位置，平均时间复杂度为，同样不占用额外内存，适合于数据规模n较小的情景。

**【算法思路】**：连续进行n轮遍历，每一轮遍历都让相邻数两两交换，确定一个已排序的数，n轮遍历后就会确定所有的数。



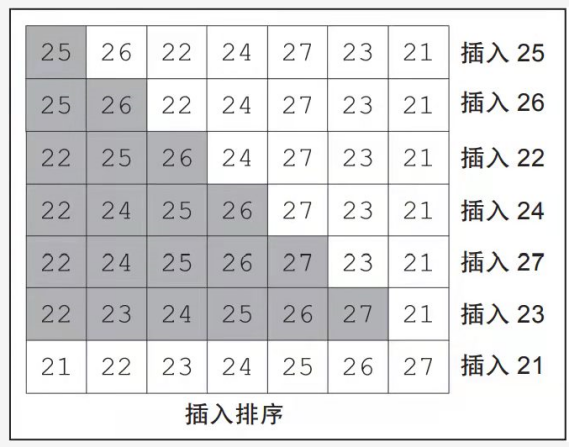
**【编码思路】：**双层循环，外层循环负责从右至左确定已排序序列，内存循环负责从左至右交换未排序序列中的相邻数。此处有个改进小技巧，内层循环维护一个bool值，如果当前一轮遍历没有交换过说明已完成排序，直接退出外层循环。



### 3、插入排序

插入排序（insert-sort）的性能和冒泡排序差不多，平均时间复杂度为，同样不占用额外内存，适合于数据规模n较小的情景。

**【算法思路】**：序列左侧视为已排序序列，序列右侧视为未排序序列，每次从未排序序列中移除一个元素到已排序序列，已排序序列每次加入新的元素就要重新交换一遍来确定顺序。



**【编码思路】：**双层循环，外层循环负责从左至右扩展已排序序列，内层循环负责从右至左在已排序序列中交换完成排序。



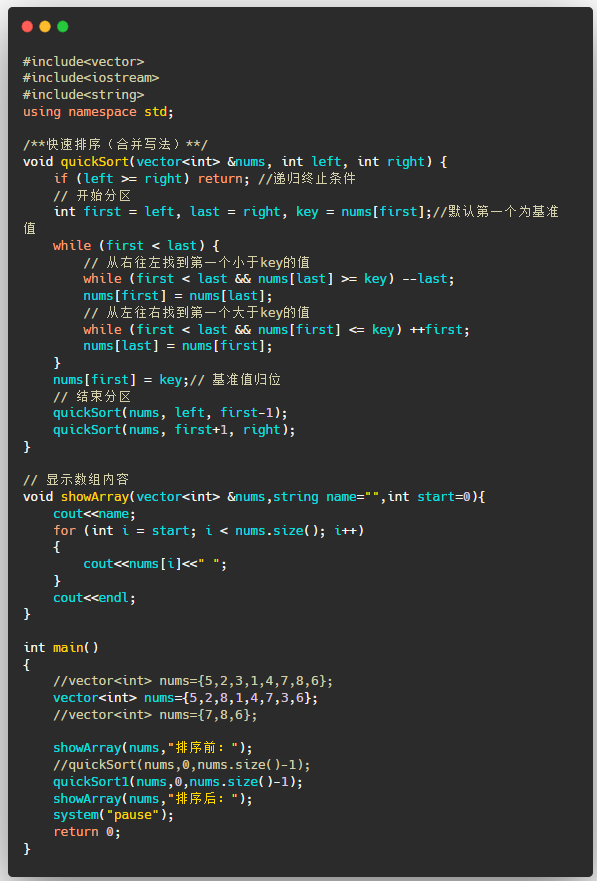
### 4、快速排序

快速排序（quick-sort）是性能鲁棒性最好的排序算法之一，任何情况下耗费时间都为，而且不占用额外内存，适合于数据规模n较小的情景。

**【算法思路】**：将一个数组分成两个子数组，将两部分独立地排序。切分地位置取决于数组的内容。先随意取第一个元素作为切分元素，从数组左端开始向右扫描直到找到一个大于等于它的元素，在从数组右端开始向左扫描找到一个小于等于它的元素，交换这两个元素的位置，直到两个指针相遇，最后和切分元素交换位置即可。

**【编码思路】：**双层循环，外层循环负责划分已排序序列和未排序序列，内存循环负责从未排序序列中查找最值，找到后交换两数即可。

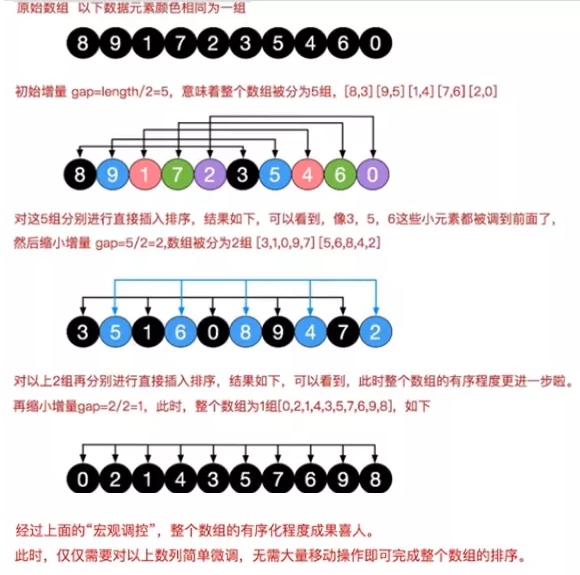




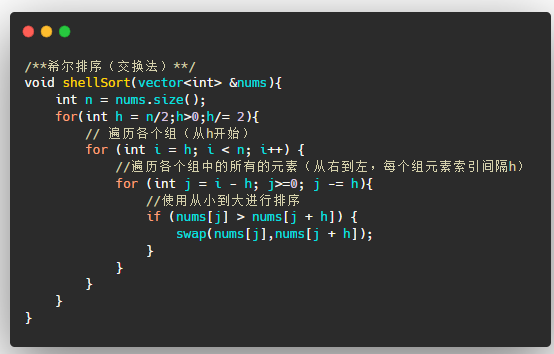
### 5、希尔排序

希尔排序（shell-sort）是1959年希尔（Donald Shell）提出的一种排序算法，可以看做是插入算法的改进版本，也成为了**缩小增量排序**。假设从小到大排序，简单插入算法插入一个较小的数时需要后移的次数明显增多，

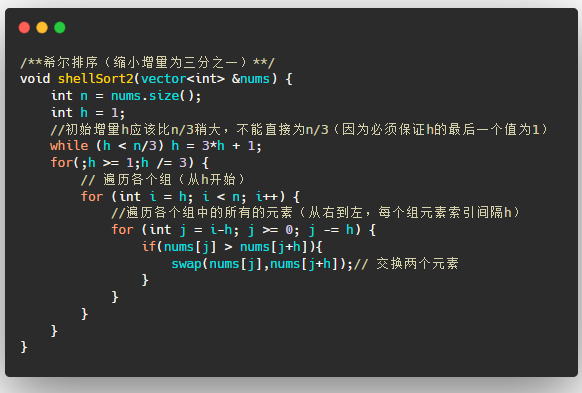
**【算法思路】**：整个数组按照增量h分成h个分组，各个分组独立进行插入排序；然后不断缩小增量h，当h等于1的时候整个数组就完成了排序。。增量h的取值也叫做Shell增量序列，如最常见的折半增量序列{n,n/2,...,1}。



**【编码思路】：三**层循环，第一层循环负责增量h的取值，第二层循环负责切换不同h分组最后一个元素索引的取值，第三层负责遍历当前h分组中所有的元素，把顺序不对的相邻元素交换过来。以折半增量序列为代表的交换式希尔算法代码实现如下：



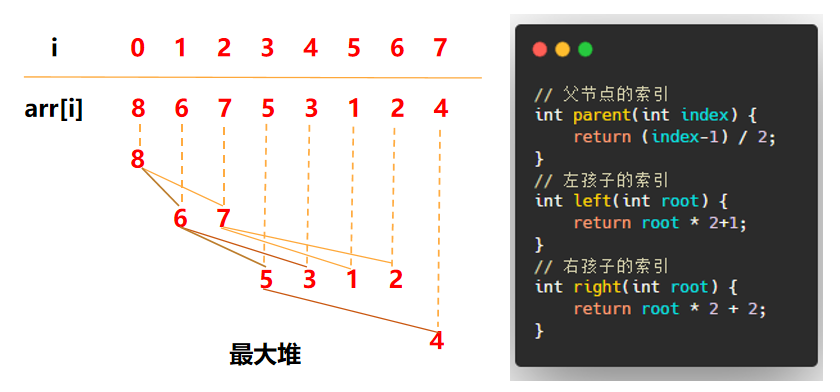
有时候我们简单换一下增量h的取值序列，就可以显著提升希尔排序的算法效率，如我们将h的取值切换为三分之一来看一下：



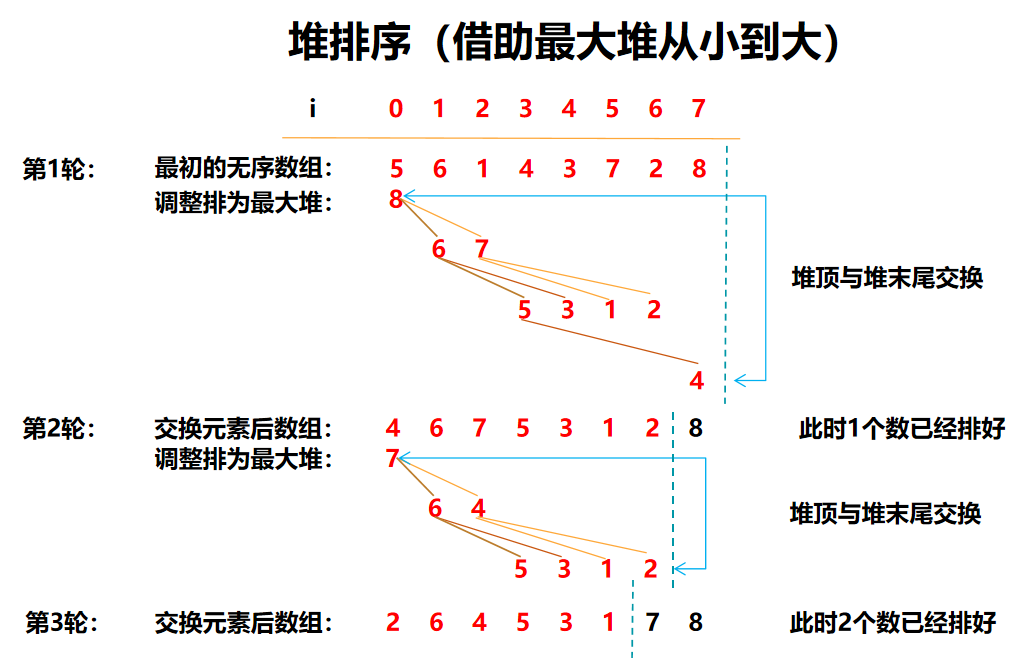
该h的取值比折半Shell增量的效率要高（力扣912题该h取值的希尔算法可以通过测试。）

### 6、堆排序

堆排序（heap-sort）是利用一种叫做堆（heap）的数据结构所设计的排序算法。堆本质上就是用数组实现的二叉树，所以也叫做二叉堆。堆有一个性质，叫做堆有序。根据这种堆有序的性质，堆分为2种，最大堆（大顶堆）和最小堆（小顶堆）。在最大堆中，父节点的值都比每一个子节点的值要大；在最小堆中，父节点的值都比每一个子节点的值要小。（注意和二叉搜索树的区别，二叉搜索树是左子树的值比父节点的值小，右子树的值比父节点的值大）。下图是一个最大堆的内部数据的逻辑结构，根据这种特点，给定任意一个数所在的索引index，我们就可以知道它的父节点索引或者左右子节点的索引。

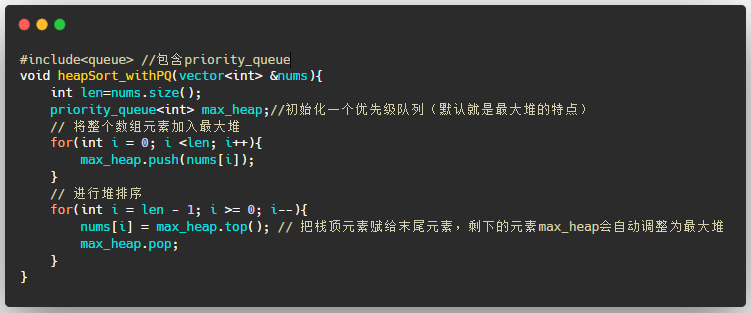


**【算法思路】**：假设从小到大排序，堆排序的基本思路是先将整个无序数组构造成一个最大堆，那么最大堆的根节点一定就是全局最大的数，将其取出来放入已排序序列；删除掉根节点的最大堆在重新构建一个最大堆，从而又找到了一个局部最大值，接着将其放入已排序序列，当无序序列中构建最大堆的数只剩下一个时，排序也就完成。过程如下：



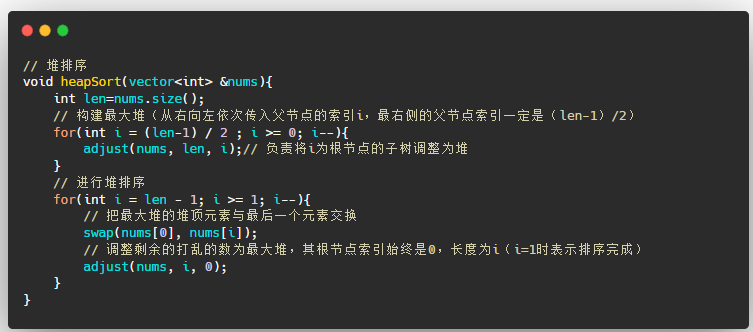
**【编码思路】：**在上述思路分析中，可知将一个无序序列转换为堆是最关键的步骤。最大堆是一个父节点都大于子节点的二叉树，在编码上可以采取递归的方式来实现。我们首先借助C++的优先级队列来实现排序。

在大多数编程语言中，优先级队列（Priority Queue）就可以看做是一个堆结构（其底层可能就是借助堆实现的），优先级队列的特点就是其队首元素始终是整个队列中最大或者最小的数，这和最大堆或最小堆的特点一样。所以，我们给出如下代码即可完成排序：

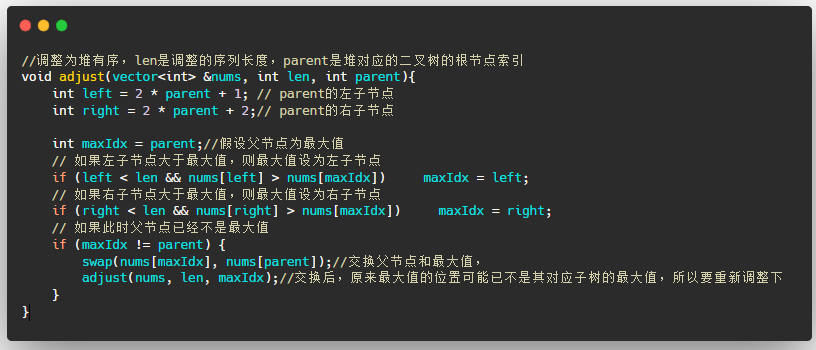


上述编码其实就是堆排序的思路，借助一个最大堆或最小堆，把无序序列始终调整为堆有序，其堆顶元素一定是全局最大或最小。现在假设面试官不让我们继续使用Priority Queue，让我们手写堆的调整结构，不要怕，难度也不会增加很多。

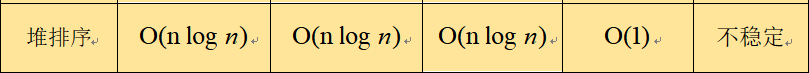
假设我们有一个函数adjust()，其功能就是负责将i为根节点的子树调整为堆有序，即满足最大堆或最小堆，那么我们借助这个adjust就可以写出如下堆排序的代码：



现在我们的任务就是写出adjust函数，其也利用了二叉树递归的思路，只要保证每个小子树中父节点比孩子节点都大或者都小即可，如下：



**【**性能分析**】**：



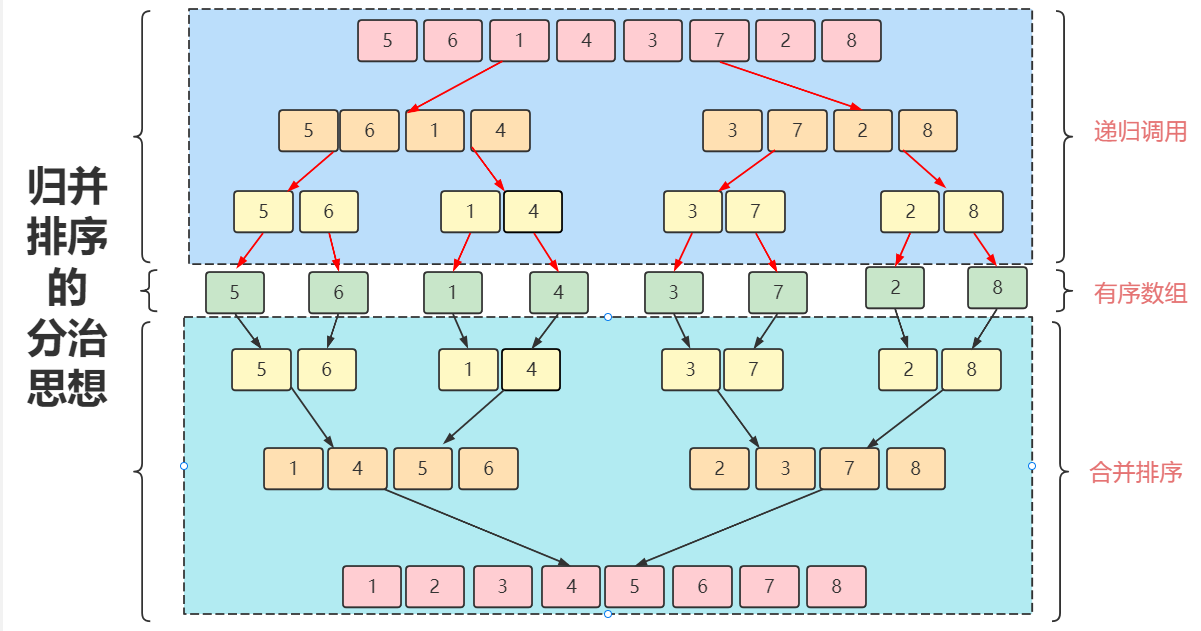
此时我们已经完成了整个堆排序的任务，最重要的是adjust的实现，读者要多默写几遍。学习堆排序后，我们实际也同时掌握了堆、优先级队列的核心知识，一石三鸟。

## 二、外排序

### 1、归并排序

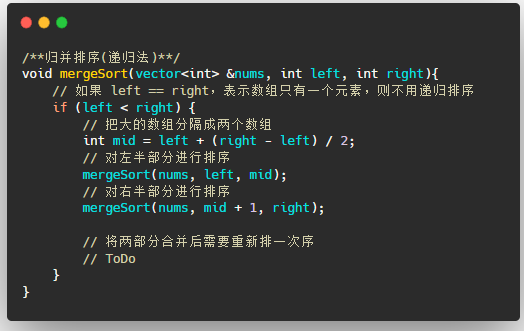
归并排序（merge-sort）采用了分治思想（即将一个大问题分解成若干个小问题来解决），分治思想在编码上一般容易使用递归法。

【算法思路】：归并排序认为如果要给一个数组排序，可以把整个大数组对等分成2个小数组，2个小数组继续分割，直到分出的每个小数组大小都是1，只有1个元素的数组一定是有序数组了，然后再把2个有序数组合并成新的更大数组，合并的时候需要重新排下序，合并后的数组就又有序了。

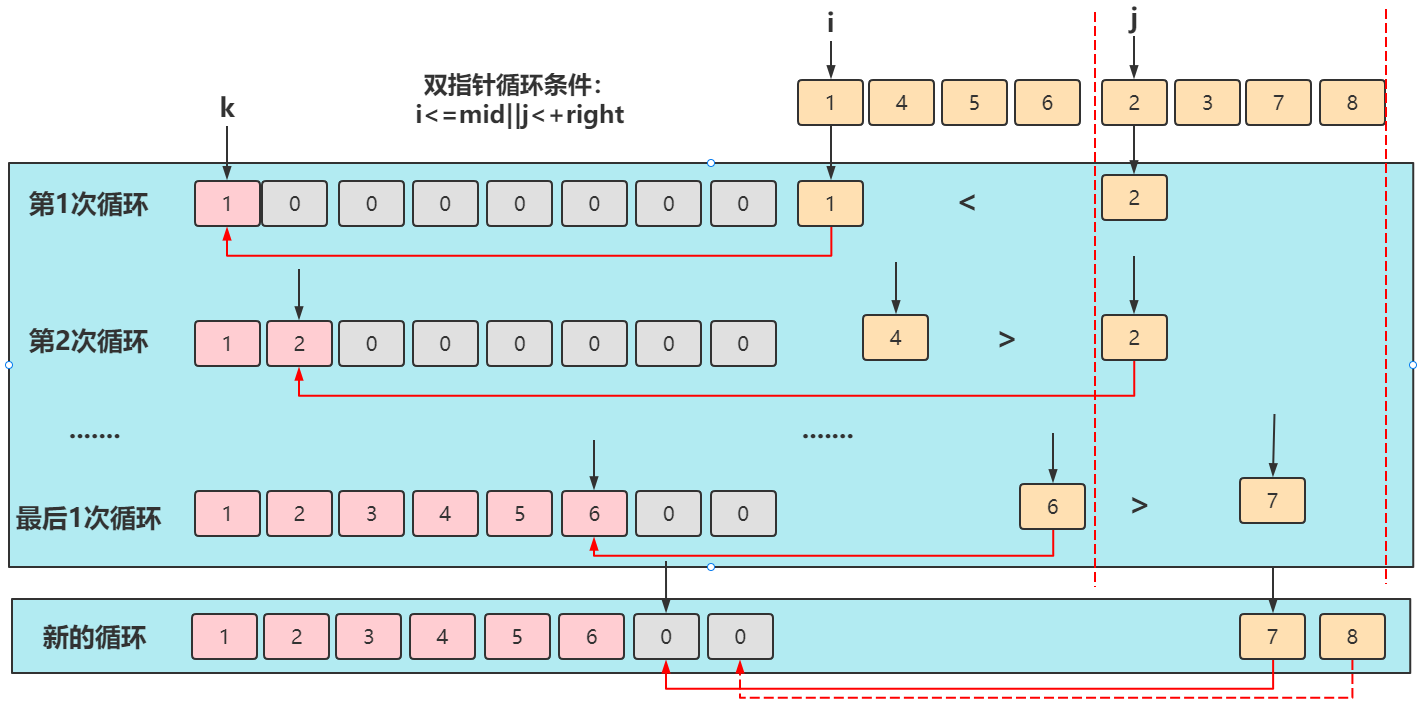


给2个小的有序数组（长度=N/2）重新排序通常比直接排序一个大的无序数组（长度=N）要快，这也是归并排序要先划分后合并的原因，先划分直至每个数组都是有序数组，然后合并排序成一个新的有序数组。

【编码思路】：在上述的思路分析中，首先递归调用来不断分割数组的编码很简单了，如下：



接下来就是考虑如何合并2个有序数组了，归并排序使用了申请新数组的双指针法来合并排序。首先直接申请一个足够容纳2个有序数组的临时数组，然后i指针指向数组的左半部分nums[left,mid]，j指针指向数组的右半部分nums[mid+1,right]。接下来依次比较2个数组中的每个元素，哪个小数组的数更小就放入临时数组中，如下：



最后双指针循环结束后，原数组中还可能会有剩余的没有复制，因为双指针的循环条件是有一个指针达到其终点即可，所以循环结束时至少有一个数组的元素全部被复制完（图中是i指针指向的左数组），另一个数组则需要一个新的循环来复制，最后直接上代码。



【性能分析】：先给出归并排序的性能分析结论：



归并排序分为两部分：递归分组和合并排序。通过递归方式把整个大数组一层一层的折半分组，则由完全二叉树的深度可知，整个排序过程需要进行logn（向上取整）次分组,最后合并排序的时候需要将每个有序小数组的数都扫描一次，整个排序过程一共扫描了整个大数组的数，所以每轮合并排序的时间复杂度为O(n)，则总的时间复杂度为 O(nlogn)。

归并排序的执行效率与要排序的原始数组的有序程度无关，所以在最好，最坏，平均情况下时间复杂度均为 O(nlogn) 。

归并排序所创建的临时数组都会在方法结束时释放，单次归并排序的最大空间是n,所以归并排序的空间复杂度为O(n).

归并排序的稳定性，取决于合并排序的时候如何处理相等的两个数。在归并排序的代码中，设置了nums[i]<=nums[j]，当两个元素相同时，先放nums[i] 的值到大数组中，所以两个相同元素的相对位置没有发生改变，此处归并排序是稳定的排序算法。

虽然归并排序时间复杂度很稳定，但是他的应用范围却不如快速排序广泛，这是因为归并排序不是原地排序算法，空间复杂度不为O(1)。

## 附加资料

本章节的主要参考资料：

[（四）排序【C++刷题】-Caoer199-博客园(cnblogs.com)](https://www.cnblogs.com/caoer/p/15722363.html)

[十大排序算法(背诵版+动图) - 力扣（LeetCode）](https://leetcode-cn.com/circle/article/0akb5U/)

借助**快速排序**的算法思想可以快速解决如下LeetCode题目：

[912. 排序数组 - 力扣（LeetCode）](https://leetcode-cn.com/problems/sort-an-array/)

[75. 颜色分类 - 力扣（LeetCode）](https://leetcode-cn.com/problems/sort-colors/submissions/)

借助**快速排序**的算法思想可以快速解决如下LeetCode题目：

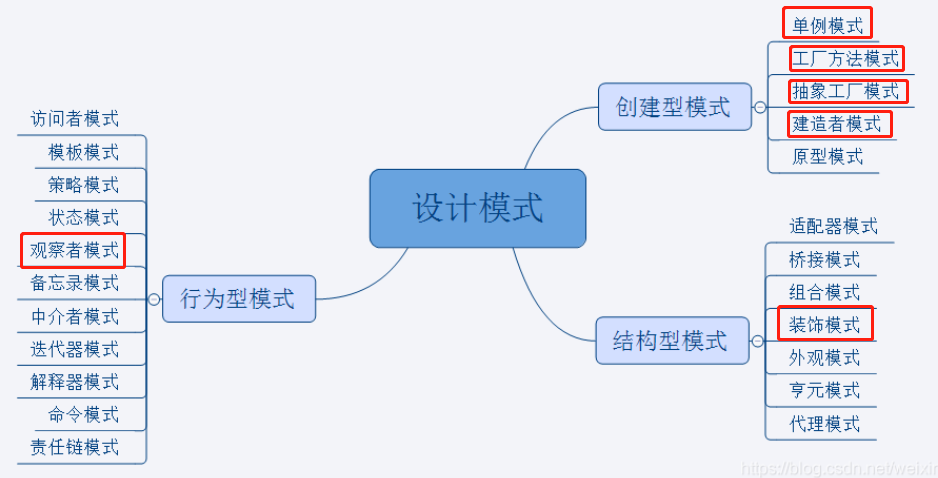
[215. 数组中的第K个最大元素 - 力扣（LeetCode）](https://leetcode-cn.com/problems/kth-largest-element-in-an-array/submissions/)

[剑指 Offer II 076.数组中的第k大的数字 - 力扣（LeetCode）](https://leetcode-cn.com/problems/xx4gT2/)



# 第6章 设计模式篇

**设计模式**是一套被**反复使用**、**多数人知晓**的、经过分类编目的**代码设计经验**的总结。使用设计模式是为了可重用代码、让代码更容易被他人理解、保证代码可靠性、程序的重用性。**设计模式**一共有23种，创建型模式，共五种：工厂方法模式、抽象工厂模式、单例模式、建造者模式、原型模式。结构型模式，共七种：适配器模式、装饰器模式、代理模式、外观模式、桥接模式、组合模式、享元模式。行为型模式，共十一种：策略模式、模板方法模式、观察者模式、迭代子模式、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。如下



## 一、设计模式的六大原则

### 1、开放封闭原则

思想：尽量通过扩展软件实体来解决需求变化，而不是通过修改已有的代码来完成变化。

描述：一个软件产品在生命周期内，都会发生变化，既然变化是一个既定的事实，我们就应该在设计的时候尽量适应这些变化，以提高项目的稳定性和灵活性。

优点：单一原则告诉我们，每个类都有自己负责的职责，里氏替换原则不能破坏继承关系的体系。

### 2、里氏代换原则

思想：使用的基类可以在任何地方使用继承的子类，完美的替换基类。

描述：子类可以扩展父类的功能，但不能改变父类原有的功能。子类可以实现父类的抽象方法，但不能覆盖父类的非抽象方法，子类中可以增加自己特有的方法。

优点：增加程序的健壮性，即使增加了子类，原有的子类还可以继续运行，互不影响。

### 3、依赖倒转原则

思想：面向接口编程，依赖于抽象而不依赖于具体

描述：它要求我们在程序代码中传递参数时或在关联关系中，尽量引用层次高的抽象层类，这个是开放封闭原则的基础。

### 4、接口隔离原则

描述：使用多个隔离的接口，比使用单个接口要好。

例如：支付类的接口和订单类的接口，需要把这俩个类别的接口变成俩个隔离的接口

优点：降低依赖，降低耦合。

### 5、迪米特法则（最少知道原则）

思想：一个对象应当对其他对象有尽可能少地了解，简称类间解耦

描述：一个类尽量减少自己对其他对象的依赖，原则是低耦合，高内聚，只有使各个模块之间的耦合尽量的低，才能提高代码的复用率。

优点：低耦合，高内聚。

### 6、单一职责原则

思想：一个方法只负责一件事情。

描述：单一职责原则很简单，一个方法 一个类只负责一个职责，各个职责的程序改动，不影响其它程序。 这是常识，几乎所有程序员都会遵循这个原则。

优点：降低类和类的耦合，提高可读性，增加可维护性和可拓展性，降低可变性的风险。

## 二、面试常见的设计模式

### 1、创建型模式

#### （1）单例模式

**单例模式**（Singleton）可能是最简单、最常见也最有效的一种设计模式。**单例模式用于创建整个程序全局唯一的一个对象**，例如整个Windows系统只能打开一个任务管理器。

单例模式的实现借助了类的static成员变量，其实现可以简单分为如下三点：

1. 将构造函数私有化，外部无法调用
2. 在类中添加一个static的类指针，用来指向构造的唯一一个对象。
3. 提供一个公有的创建对象方法，外部只能通过该方法创建其对象。

代码如下：



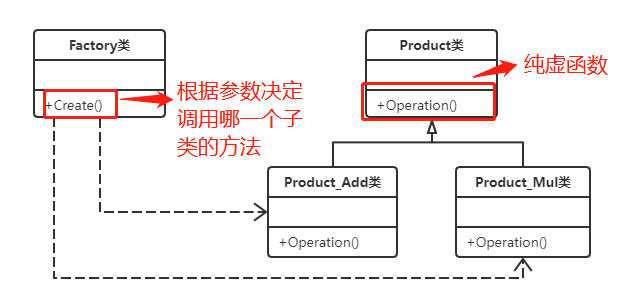
#### （2）工厂模式

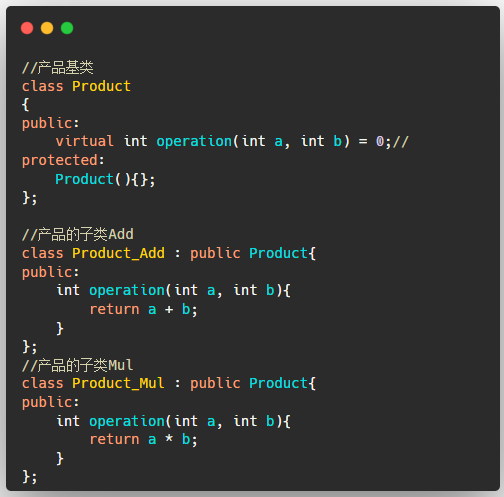
工厂模式（Factory）简单来讲就是让创建对象有了统一的接口，具体化类的工作延迟到了子类中，其实现用到了C++的继承和纯虚函数的知识。

工厂模式严格来讲有三种类型：简单工厂模式、工厂模式和抽象工厂模式。

##### 简单工厂模式

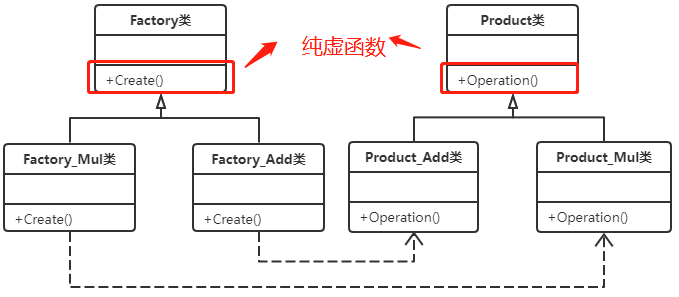
我们将被创建对象类命名为Product，创建对象的类命名为Factory。Factory调用Product的统一接口operation()，其具体实现由各自的子类决定。

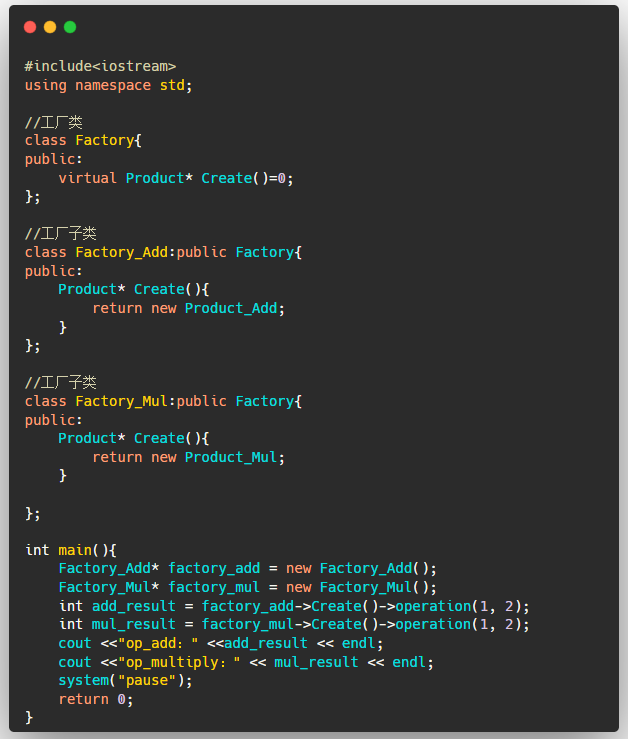
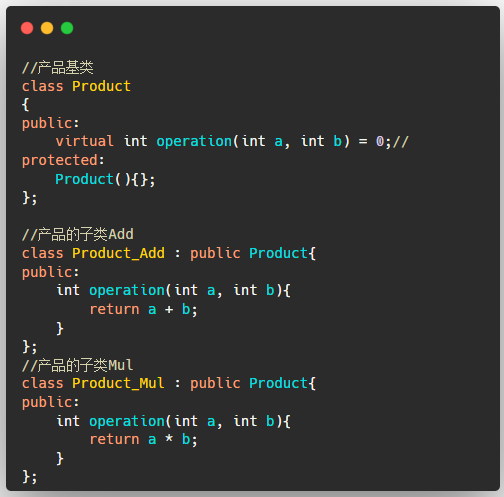




##### 工厂模式

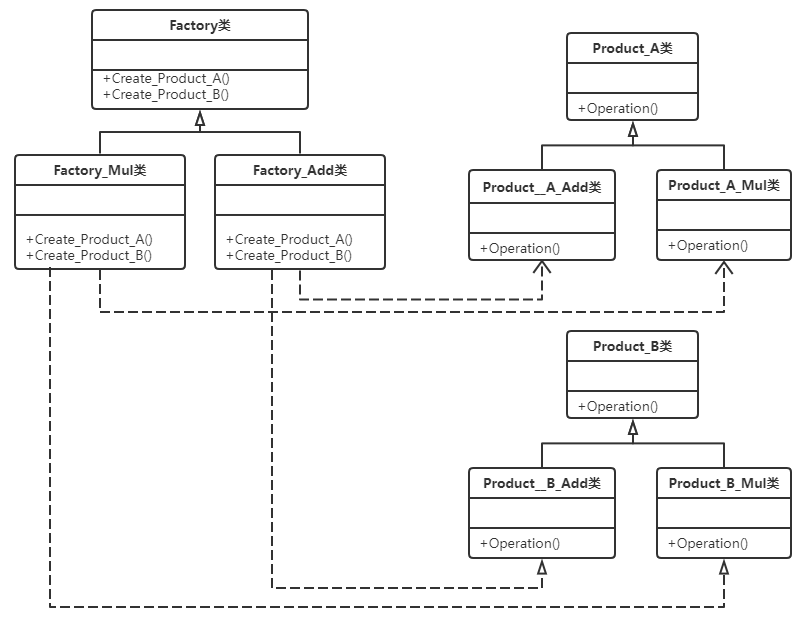
上述的简单工厂模式中还存在不足就是每次增加一个新的Product子类，都需要修改Factory的内部代码，这违背了开放-封闭原则（对扩展开放，对修改封闭）。工厂模式抽象出了Factory基类，不同的Factory子类负责创建一个对应的产品，所以可以修改如下。





##### 抽象工厂模式

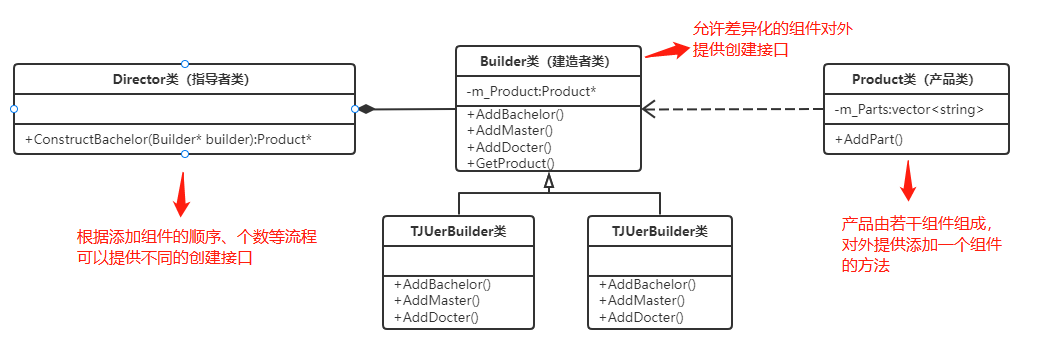
工厂模式中，一个工厂只生产一种产品，如果想要一个工厂生产多个产品，就需要抽象工厂模式，抽象工厂模式提供了创建一系列相互依赖对象的接口。抽象工厂模式本质上是简单工厂模式和工厂模式的组合。

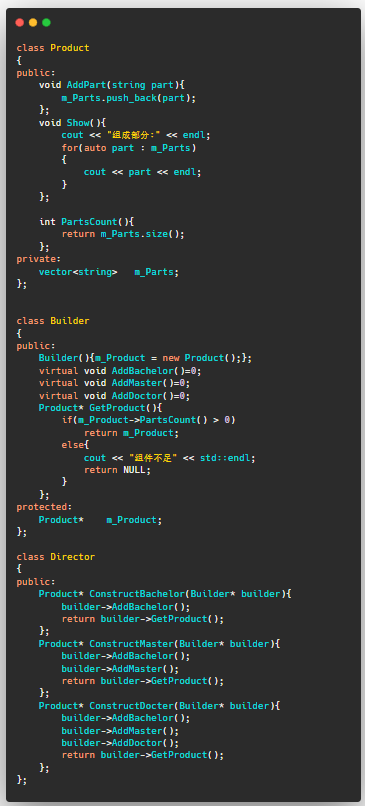




#### （3）建造者模式

建造者模式（Builder）适用于创建包含不同组件的对象，它分为Product、Builder和Director三部分，分别定义了产品组件的添加方式、差异组件的扩展方式、组件装配流程的对外接口，如下：



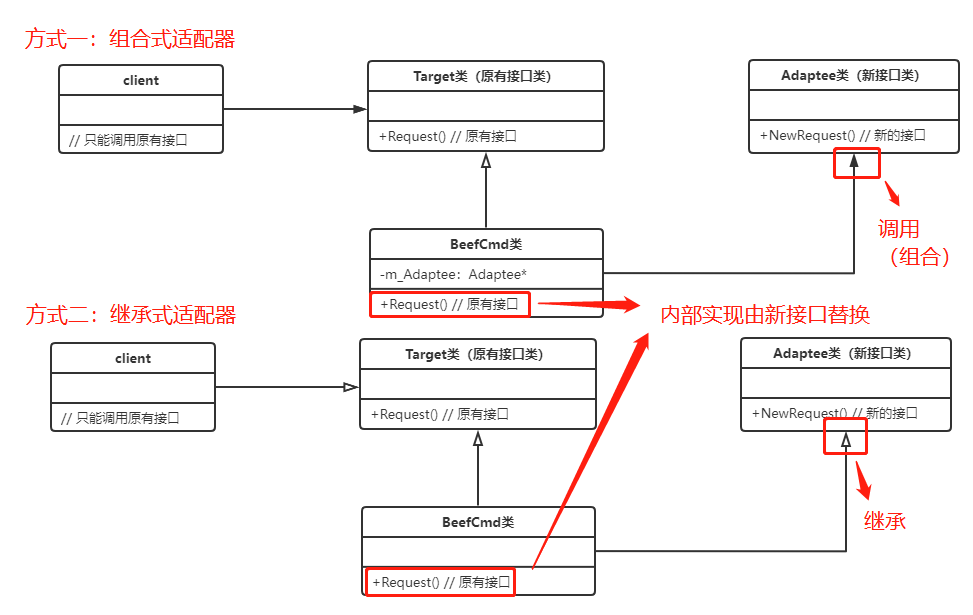


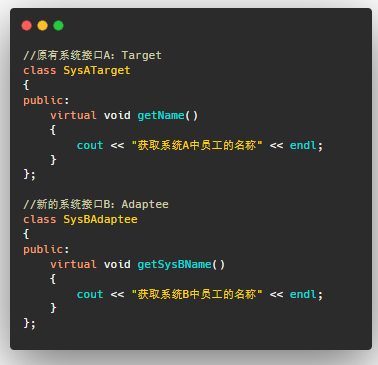
### 2、结构型模式

#### （1）适配器模式

适配器模式（Adapter）是用来在旧的系统接口中替换成新的系统接口的设计模式，简单说对于同一个功能存在一个老版本的类对象A，因为需求升级开发了新的类对象B，A和B实现相同的功能但是类的接口名称不一致，原有程序无法直接调用类B。而且由于系统生态的原因，大部分程序只能调用类A的接口，所以也不能舍弃A。

这种情况下就得使用适配器模式。假设存在一个原有接口Target和一个新的接口Adaptee，现在要使Target中的接口的实现被替换成Adaptee的实现。适配器模式的做法是首先从Target派生出一个子类Adapter，Adapter可以调用Adaptee的函数（组合方式）或继承Adaptee的函数来替换自身父类的实现。

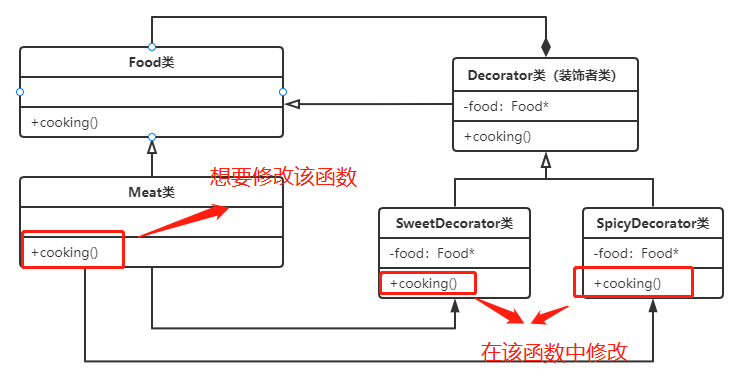




#### （2）装饰者模式

装饰者模式（Decorator）也叫包装模式。如果给一个已有的类添加新的实现方法，一种思路是继承该类，在子类中添加；另一种思路是组合。装饰者模式就是利用组合的方式在已由类添加新的功能。

假如现在有个Food类，其有一个公有接口Cooking()，Food类派生出了1个子类Meat，其Cooking()会输出肉。



这时候由于新的需求，需要修改Meat类的Cooking()，使其输出加了辣味或甜味的肉，如果不直接修改Meat类，一种方式是继承自Meat类，重写其Cooking()。装饰者模式的思路是不继承Meat()类，而是在Food派生出的新的子类Decorator，这个Decorator具有一个Food指针，从Decorator派生出更多子类，例如SweetDecorator。

由于Decorator和Meat的父类都是Food，所以SweetDecorator也可修饰Decorator类及其子类。如果想要一块添加了辣味和甜味的肉，不需要再新加装饰类，只需要用将SweetDecorator作为参数构造出一个SpicyDecorator类即可，可见代码。

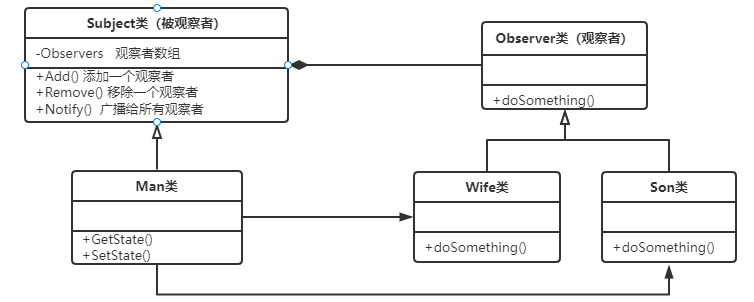


### 3、行为型模式

#### （1）观察者模式

观察者模式（Observer）很可能是应用最多、影响最广的设计模式之一，也叫发布-订阅模式。观察者模式的核心是一对多的通知，当被观察者状态发生变化时（发生某个事件）通知所有观察者做出响应。

经典应用场景：微信朋友圈动态通知、消息通知、邮件通知等。经典的MVC（Model/View/Control）结构就采用了其思路，其将业务逻辑和表示层进行接口。



代码实现思路：被观察者发生某个事件时所有观察者都要做出相应，这就要求被观察者要维护一个观察者的数组，然后当某个函数（暂定为notify）被调用时就遍历该数组调用观察者的函数，这就要求所有的观察者提供一个统一的接口（暂定为doSomething）。

观察者Observer和被观察者Subject的接口实现也非常简单，主要是被观察者要维护一个vector数组。



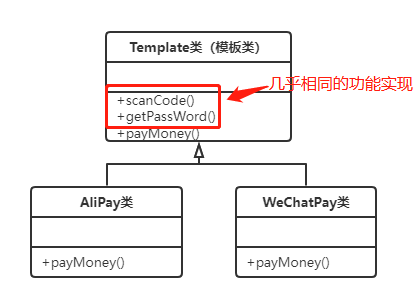
测试场景，丈夫回家发通知给妻子和孩子，丈夫是被观察者，妻子和孩子是观察者。如下：



#### （2）模板模式

模板模式（template）是一个非常自然也非常简单的设计模式，我们甚至不知不觉都在使用它。它解决的是如何组织业务（算法）逻辑相似的对象，例如微信支付、支付宝支付。模板模式采用继承的方式实现这一点，为算法逻辑相似的对象定义一个共同的抽象基类，子类负责实现具体的细节。

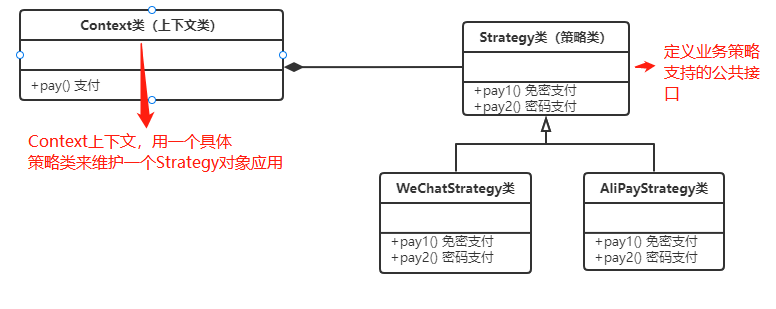
生活中常见的模板模式的例子：付款购物的流程：扫描获取二维码、获取支付密码、支付金额，其中支付金额可以由微信支付和支付宝支付两种接口来完成，其他的则可以使用同一套方法。





#### （3）策略模式

策略模式（Strategy）和模板模式解决的都是同一种问题，即如何组织业务（算法）逻辑相似的对象。例如开发一个支付模块，有微信支付和支付宝支付2种软件包的各种接口供调用，由于这两种支付是不同厂商开发的，接口名称可能都不统一。前面所述的模板模式直接为这2种支付定义一个共同基类；策略模式的方式是定义一个Strategy类，其中定义支付业务都会用到的公共接口，由此派生出两个具体的支付策略类。Context类负责根据Strategy类提供的公共接口来开发具体业务：例如支付时选用免密支付方式还是密码支付方式。



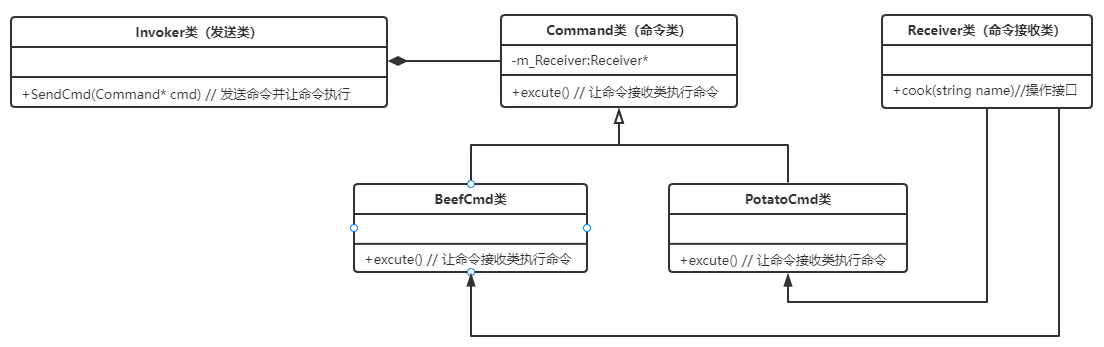
策略模式的优点是使用继承的方式已于修改和扩展；缺点是破坏了类的封装性，继承中父类的实现细节暴露给了子类，策略类会增多，所有的策略类都会对外暴露。



#### （4）命令模式

命令模式（Command）可以实现消息通信的发送者和接收者的完全解耦，发送者和接收者之间没有直接饮用官谢，发送者只需要知道如何发送请求。

例如服务员和厨师的沟通场景，厨师是命令接收者Receiver，服务员是命令发送者Invoker。普通写法下，Invoker根据不同的命令直接调用Receiver的不同操作函数，这样子开发者需要频繁修改Invoker类。命令模式下，Receiver和Invoker之间添加一个Command类，Invoker只调用Command的一个统一接口，不同的命令由Command不同的子类内部进行处理，新加一个命令时，开发者不需要改动Invoker，只需要继承一个Command子类，然后在其内部调用Receiver的合适操作接口即可。





命令模式的优点是将命令调用者和命令执行者的对象进行了解耦，添加新的命令只需要继承命令类即可。缺点是增加了类的数量，有几个具体命令就有多少具体类，可能需要大量具体命令类。

# 迭代日志表

|  |  |  |
| --- | --- | --- |
| **迭代版本** | **迭代工作** | **迭代日期** |
| V1.0 | 建立初稿文档，完成第1章和第2章的整理。 | 2022-02.02-02.05 |
| V1.1 |  |  |
| V1.2 |  |  |