C++ 基础知识总结

目录

C++ 面试基础知识总结 1
C/C++ 1
const 1
static 2
this 指针 2
inline 内联函数 2
volatile 4
assert() 4
sizeof() 4
#pragma pack(n) 4
位域 5
extern “C” 5
struct 和 typedef struct 5
C++ 中 struct 和 class 6
union 联合 6
C 实现 C++ 类 7
explicit（显式）关键字 7
friend 友元类和友元函数 8
using 8
:: 范围解析运算符 9
enum 枚举类型 9
decltype 9
引用 9
宏 10
成员初始化列表 10
initializer_list 列表初始化 10
面向对象 11
封装 11
继承 11
多态 12
虚析构函数 13
纯虚函数 13
虚函数、纯虚函数 13
虚函数指针、虚函数表 13
虚继承 14
虚继承、虚函数 14
模板类、成员模板、虚函数 14
抽象类、接口类、聚合类 14
内存分配和管理 14
delete this 合法吗？ 15
如何定义一个只能在堆上（栈上）生成对象的类？ 15
智能指针 15

强制类型转换运算符 16
运行时类型信息 (RTTI) 17
Effective C++ 18
Google C++ Style Guide 21
其他 21
 STL 21
STL 容器 21
STL 索引 22
array 22
vector 23
deque 24
forward_list 24
list 24
stack 24
queue 24
priority_queue 24
set 24
multiset 24
map 24
multimap 25
unordered_set 25
unordered_multiset 25
unordered_map 25
unordered_multimap 25
tuple 25
pair 25
组成 30
容器（containers） 30
array 30
vector 39
deque 56
forward_list 60
list 63
stack 63
queue 63
priority_queue 63
set 63
multiset 63
map 63
multimap 70
无序容器（Unordered Container）：unordered_set、unordered_multiset、unordered_map、unordered_multimap 70
unordered_set 71
unordered_multiset 71
unordered_map 71
unordered_multimap 71

tuple 71
pair 73
数据结构 74
顺序结构 74
链式结构 77
哈希表 78
递归 79
二叉树 81
其他树及森林 82
算法 84
排序 84
查找 85
图搜索算法 85
其他算法 85
Problems 85
Single Problem 85
Leetcode Problems 85
剑指 Offer 85
Cracking the Coding Interview 程序员面试金典 85
牛客网 86
操作系统 86
进程与线程 86
Linux 内核的同步方式 88
死锁 88
文件系统 88
主机字节序与网络字节序 88
页面置换算法 90
计算机网络 90
各层作用及协议 91
物理层 91
数据链路层 91
网络层 92
运输层 94
应用层 102
网络编程 103
Socket 103
数据库 105
基本概念 105
常用数据模型 106
常用 SQL 操作 106
关系型数据库 107
数据库完整性 107
关系数据理论 107

数据库恢复 107
并发控制 107
设计模式 108
单例模式 108
抽象工厂模式 108
适配器模式 108
桥接模式 108
观察者模式 108
设计模式的六大原则 108
链接装载库 108
内存、栈、堆 108
编译链接 109
Linux 的共享库（Shared Library） 110
Windows 应用程序入口函数 111
Windows 的动态链接库（Dynamic-Link Library） 112
运行库（Runtime Library） 117
书籍 118
语言 118
算法 118
系统 118
网络 118
其他 118
C/C++ 发展方向 118
后台/服务器 118
桌面客户端 118
图形学/游戏/VR/AR 118
测试开发 119
网络安全/逆向 119
嵌入式/物联网 119
音视频/流媒体/SDK 119
计算机视觉/机器学习 119

C/C++

const

作用

1. 修饰变量，说明该变量不可以被改变；
2. 修饰指针，分为指向常量的指针和指针常量；
3. 常量引用，经常用于形参类型，即避免了拷贝，又避免了函数对值的修改；
4. 修饰成员函数，说明该成员函数内不能修改成员变量。

使用

const 使用

```
// 类
class A
{
private:
    const int a;                // 常对象成员，只能在初始化列表赋值
public:
    // 构造函数
    A() : a(0) { };          // 初始化列表
    A(int x) : a(x) { };        // 初始化列表
    // const 可用于对重载函数的区分
    int getValue();             // 普通成员函数
    int getValue() const;       // 常成员函数，不得修改类中的任何数据成员的值
};

void function()
{
    // 对象
    A b;                        // 普通对象，可以调用全部成员函数
    const A a;                  // 常对象，只能调用常成员函数、更新常成员变量
    const A *p = &a;            // 常指针
    const A &q = a;             // 常引用
    // 指针
    char greeting[] = "Hello";
    char* p1 = greeting;                // 指针变量，指向字符数组变量
    const char* p2 = greeting;          // 指针变量，指向字符数组常量
    char* const p3 = greeting;          // 常指针，指向字符数组变量
    const char* const p4 = greeting;    // 常指针，指向字符数组常量
}
// 函数
void function1(const int Var);           // 传递过来的参数在函数内不可变
void function2(const char* Var);         // 参数指针所指内容为常量
void function3(char* const Var);         // 参数指针为常指针
void function4(const int& Var);          // 引用参数在函数内为常量
// 函数返回值
const int function5();      // 返回一个常数
const int* function6();     // 返回一个指向常量的指针变量，使用：const int *p = function6();
int* const function7();     // 返回一个指向变量的常指针，使用：int* const p = function7();
```

static

作用

1. 修饰普通变量，修改变量的存储区域和生命周期，使变量存储在静态区，在 main 函数运行前就分配了空间，如果有初始值就用初始值初始化它，如果没有初始值系统用默认值初始化它。
2. 修饰普通函数，表明函数的作用范围，仅在定义该函数的文件内才能使用。在多人开发项目时，为了防止与他人命名空间里的函数重名，可以将函数定位为 static。
3. 修饰成员变量，修饰成员变量使所有的对象只保存一个该变量，而且不需要生成对象就可以访问该成员。
4. 修饰成员函数，修饰成员函数使得不需要生成对象就可以访问该函数，但是在 static 函数内不能访问非静态成员。

this 指针

1. this 指针是一个隐含于每一个非静态成员函数中的特殊指针。它指向调用该成员函数的那个对象。
2. 当对一个对象调用成员函数时，编译程序先将对象的地址赋给 this 指针，然后调用成员函数，每次成员函数存取数据成员时，都隐式使用 this 指针。
3. 当一个成员函数被调用时，自动向它传递一个隐含的参数，该参数是一个指向这个成员函数所在的对象的指针。
4. this 指针被隐含地声明为: ClassName *const this，这意味着不能给 this 指针赋值；在 ClassName 类的 const 成员函数中，this 指针的类型为：const ClassName* const，这说明不能对 this 指针所指向的这种对象是不可修改的（即不能对这种对象的数据成员进行赋值操作）；
5. this 并不是一个常规变量，而是个右值（不能取地址），所以不能取得 this 的地址（不能 &this）。
6. 在以下场景中，经常需要显式引用 this 指针：
    1. 为实现对象的链式引用；
    2. 为避免对同一对象进行赋值操作；
    3. 在实现一些数据结构时，如 list。

inline 内联函数

特征

- 相当于把内联函数里面的内容写在调用内联函数处；
- 相当于不用执行进入函数的步骤，直接执行函数体；
- 相当于宏，却比宏多了类型检查，真正具有函数特性；
- 编译器一般不内联包含循环、递归、switch 等复杂操作的内联函数；
- 在类声明中定义的函数，除了虚函数的其他函数都会自动隐式地当成内联函数。

使用

inline 使用

```
// 声明1（加 inline，建议使用）
inline int functionName(int first, int second,...);
// 声明2（不加 inline）
int functionName(int first, int second,...);
// 定义
inline int functionName(int first, int second,...) {/****/};
// 类内定义，隐式内联
class A {
    int doA() { return 0; }         // 隐式内联
}

// 类外定义，需要显式内联
class A {
    int doA();
```

```
}
inline int A::doA() { return 0; }   // 需要显式内联
```

编译器对 inline 函数的处理步骤

1. 将 inline 函数体复制到 inline 函数调用点处；
2. 为所用 inline 函数中的局部变量分配内存空间；
3. 将 inline 函数的输入参数和返回值映射到调用方法的局部变量空间中；
4. 如果 inline 函数有多个返回点，将其转变为 inline 函数代码块末尾的分支（使用 GOTO）

优缺点

优点

1. 内联函数同宏函数一样将在被调用处进行代码展开，省去了参数压栈、栈帧开辟与回收，结果返回等，从而提高程序运行速度。
2. 内联函数相比宏函数来说，在代码展开时，会做安全检查或自动类型转换（同普通函数），而宏定义则不会。
3. 在类中声明同时定义的成员函数，自动转化为内联函数，因此内联函数可以访问类的成员变量，宏定义则不能。
4. 内联函数在运行时可调试，而宏定义不可以。

缺点

1. 代码膨胀。内联是以代码膨胀（复制）为代价，消除函数调用带来的开销。如果执行函数体内代码的时间，相比于函数调用的开销较大，那么效率的收获会很少。另一方面，每一处内联函数的调用都要复制代码，将使程序的总代码量增大，消耗更多的内存空间。
2. inline 函数无法随着函数库升级而升级。inline 函数的改变需要重新编译，不像 non-inline 可以直接链接。
3. 是否内联，程序员不可控。内联函数只是对编译器的建议，是否对函数内联，决定权在于编译器。

虚函数（virtual）可以是内联函数（inline）吗？

Are "inline virtual" member functions ever actually "inlined"?

- 虚函数可以是内联函数，内联是可以修饰虚函数的，但是当虚函数表现多态性的时候不能内联。
- 内联是在编译器建议编译器内联，而虚函数的多态性在运行期，编译器无法知道运行期调用哪个代码，因此虚函数表现为多态性时（运行期）不可以内联。
- `inline virtual` 唯一可以内联的时候是：编译器知道所调用的对象是哪个类（如 `Base::who()`），这只有在编译器具有实际对象，而不是对象的指针或引用时才会发生。

虚函数内联使用

```
#include <iostream>
using namespace std;
class Base
{
public:
    inline virtual void who()
    {
        cout << "I am Base\n";
    }
    virtual ~Base() {}
};
class Derived : public Base
{
public:
    inline void who()  // 不写inline 时隐式内联
    {
        cout << "I am Derived\n";
    }
};
```

```
int main()
{
    // 此处的虚函数 who()，是通过类（Base）的具体对象（b）来调用的，编译期间就能确定了，所以它可以是内联的，但最终是否内联取决于编译器。
    Base b;
    b.who();
   //此处的虚函数是通过指针调用的，呈现多态性，需要在运行时期间才能确定，所以不能为内联。
    Base *ptr = new Derived();
    ptr->who();
     // 因为Base 有虚析构函数（virtual ~Base() {}），所以 delete 时，会先调用派生类（Derived）析构函数，再调用基类（Base）析构函数，防止内存泄漏。
    delete ptr;
    ptr = nullptr;

    system("pause");
    return 0;
}
```

volatile

```
volatile int i = 10;
```

- volatile 关键字是一种类型修饰符，用它声明的类型变量表示可以被某些编译器未知的因素（操作系统、硬件、其它线程等）更改。所以使用 volatile 告诉编译器不应对这样的对象进行优化。
- volatile 关键字声明的变量，每次访问时都必须从内存中取出值（没有被 volatile 修饰的变量，可能由于编译器的优化，从 CPU 寄存器中取值）
- const 可以是 volatile （如只读的状态寄存器）
- 指针可以是 volatile

assert()

断言，是宏，而非函数。assert 宏的原型定义在 <assert.h>（C）、<cassert>（C++）中，其作用是如果它的条件返回错误，则终止程序执行。可以通过定义 NDEBUG 来关闭 assert，但是需要在源代码的开头，include <assert.h> 之前。

assert() 使用

```
#define NDEBUG          // 加上这行，则 assert 不可用
#include <assert.h>

assert( p != NULL );    // assert 不可用
```

sizeof()

- sizeof 对数组，得到整个数组所占空间大小。
- sizeof 对指针，得到指针本身所占空间大小。

#pragma pack(n)

设定结构体、联合以及类成员变量以 n 字节方式对齐

#pragma pack(n) 使用

```
#pragma pack(push)  // 保存对齐状态
#pragma pack(4)     // 设定为 4 字节对齐
struct test
{
    char m1;
    double m4;
    int m3;
};
#pragma pack(pop)   // 恢复对齐状态
typedef struct {
    unsigned char prior;  /* 优先级 */
}__attribute__((packed))  USER_SETTING;
```

位域

```
Bit mode: 2;    // mode 占 2 位
```

类可以将其（非静态）数据成员定义为位域（bit-field），在一个位域中含有一定数量的二进制位。当一个程序需要向其他程序或硬件设备传递二进制数据时，通常会用到位域。

- 位域在内存中的布局是与机器有关的
- 位域的类型必须是整型或枚举类型，带符号类型中的位域的行为将因具体实现而定
- 取地址运算符（&）不能作用于位域，任何指针都无法指向类的位域

extern “C”

- 被 extern 限定的函数或变量是 extern 类型的
- 被 `extern "C"` 修饰的变量和函数是按照 C 语言方式编译和链接的

`extern "C"` 的作用是让 C++ 编译器将 `extern "C"` 声明的代码当作 C 语言代码处理，可以避免 C++ 因符号修饰导致代码不能和 C 语言库中的符号进行链接的问题。

extern “C” 使用

```
#ifdef __cplusplus
extern "C" {
#endif
void *memset(void *, int, size_t);

#ifdef __cplusplus
}
#endif
```

struct 和 typedef struct

C 中

```
// c
typedef struct Student {
    int age;
} S;
```

等价于

```
// c
struct Student {
    int age;
};

typedef struct Student S;
```

此时 S 等价于 `struct Student`，但两个标识符名称空间不相同。

另外还可以定义与 `struct Student` 不冲突的 `void Student() {}`。

C++ 中

由于编译器定位符号的规则（搜索规则）改变，导致不同于 C 语言。

一、如果在类标识符空间定义了 `struct Student {...};`，使用 `Student me;` 时，编译器将搜索全局标识符表，`Student` 未找到，则在类标识符内搜索。

即表现为可以使用 `Student` 也可以使用 `struct Student`，如下：

```
// cpp
struct Student {
    int age;
};

void f( Student me );        // 正确，"struct" 关键字可省略
```

二、若定义了与 `Student` 同名函数之后，则 `Student` 只代表函数，不代表结构体，如下：

```
typedef struct Student {
    int age;
} S;
```

```
void Student() {}           // 正确，定义后 "Student" 只代表此函数

//void S() {}   // 错误，符号 "S" 已经被定义为一个 "struct Student" 的别名

int main() {
    Student();
    struct Student me;      // 或者 "S me";
    return 0;
}
```

C++ 中 struct 和 class

认为，struct 更适合看成是一个数据结构的实现体，class 更适合看成是一个对象的实现体。

区别

- 最本质的一个区别就是默认的访问控制
    1. 默认的继承访问权限。struct 是 public 的，class 是 private 的。
    2. struct 作为数据结构的实现体，它默认的数据访问控制是 public 的，而 class 作为对象的实现体，它默认的成员变量访问控制是 private 的。

union 联合

联合（union）是一种节省空间的特殊的类，一个 union 可以有多个数据成员，但是在任意时刻只有一个数据成员可以有值。当某个成员被赋值后其他成员变为未定义状态。联合有如下特点：

- 默认访问控制符为 public
- 可以含有构造函数、析构函数
- 不能含有引用类型的成员
- 不能继承自其他类，不能作为基类
- 不能含有虚函数
- 匿名 union 在定义所在作用域可直接访问 union 成员
- 匿名 union 不能包含 protected 成员或 private 成员
- 全局匿名联合必须是静态（static）的

union 使用

```
#include<iostream>

union UnionTest {
    UnionTest() : i(10) {};
    int i;
    double d;
};

static union {
    int i;
    double d;
};

int main() {
    UnionTest u;

    union {
        int i;
        double d;
    };
    std::cout << u.i << std::endl;  // 输出 UnionTest 联合的 10

    ::i = 20;
    std::cout << ::i << std::endl;  // 输出全局静态匿名联合的 20
```

```
    i = 30;
    std::cout << i << std::endl;    // 输出局部匿名联合的 30

    return 0;
}
```

C 实现 C++类

C 语言实现封装、继承和多态

explicit（显式）关键字

- explicit 修饰构造函数时，可以防止隐式转换和复制初始化
- explicit 修饰转换函数时，可以防止隐式转换，但 按语境转换 除外

explicit 使用

```
struct A
{
    A(int) { }
    operator bool() const { return true; }
};

struct B
{
    explicit B(int) {}
    explicit operator bool() const { return true; }
};

void doA(A a) {}

void doB(B b) {}

int main()
{
    A a1(1);        // OK：直接初始化
    A a2 = 1;       // OK：复制初始化
    A a3{ 1 };      // OK：直接列表初始化
    A a4 = { 1 };   // OK：复制列表初始化
    A a5 = (A)1;    // OK：允许 static_cast 的显式转换
    doA(1);     // OK：允许从 int 到 A 的隐式转换
    if (a1);    // OK：使用转换函数 A::operator bool() 的从 A 到 bool 的隐式转换
    bool a6（a1）; // OK：使用转换函数 A::operator bool() 的从 A 到 bool 的隐式转换
    bool a7 = a1; // OK：使用转换函数 A::operator bool() 的从 A 到 bool 的隐式转换
    bool a8 = static_cast<bool>(a1);  // OK ：static_cast 进行直接初始化

    B b1(1);        // OK：直接初始化
    B b2 = 1;       // 错误：被 explicit 修饰构造函数的对象不可以复制初始化
    B b3{ 1 };      // OK：直接列表初始化
    B b4 = { 1 };       // 错误：被 explicit 修饰构造函数的对象不可以复制列表初始化
    B b5 = (B)1;        // OK：允许 static_cast 的显式转换
    doB(1);    // 错误：被 explicit 修饰构造函数的对象不可以从 int 到 B 的隐式转换
    if (b1);   // OK：被 explicit 修饰转换函数 B::operator bool() 的对象可以从 B 到 bool
的按语境转换
    bool b6(b1);        // OK：被 explicit 修饰转换函数 B::operator bool() 的对象可以从
B 到 bool 的按语境转换
    bool b7 = b1;       // 错误：被 explicit 修饰转换函数 B::operator bool() 的对象不可
```

```
以隐式转换
    bool b8 = static_cast<bool>(b1);  // OK：static_cast 进行直接初始化

    return 0;
}
```

friend 友元类和友元函数

- 能访问私有成员
- 破坏封装性
- 友元关系不可传递
- 友元关系的单向性
- 友元声明的形式及数量不受限制

using

using 声明

一条 using 声明 语句一次只引入命名空间的一个成员。它使得我们可以清楚知道程序中所引用的到底是哪个名字。如：

```
using namespace_name::name;
```

构造函数的 using 声明

在 C++11 中，派生类能够重用其*直接基类定义*的构造函数。

```
class Derived : Base {
public:
    using Base::Base;
    /* ... */
};
```

如上 using 声明，对于基类的每个构造函数，编译器都生成一个与之对应（形参列表完全相同）的派生类构造函数。生成如下类型构造函数：

```
derived(parms) : base(args) { }
```

using 指示

using 指示 使得某个特定命名空间中所有名字都可见，这样我们就无需再为它们添加任何前缀限定符了。如：

```
using namespace_name name;
```

尽量少使用 using 指示 污染命名空间

一般说来，使用 using 命令比使用 using 编译命令更安全，这是由于它**只导入了指定的名称**。如果该名称与局部名称发生冲突，编译器将发出指示。using 编译命令导入所有的名称，包括可能并不需要的名称。如果与局部名称发生冲突，则局部名称将覆盖名称空间版本，而编译器并不会发出警告。另外，名称空间的开放性意味着名称空间的名称可能分散在多个地方，这使得难以准确知道添加了哪些名称。

using 使用

尽量少使用 using 指示

```
using namespace std;
```

应该多使用 using 声明

```
int x;
std::cin >> x ;
std::cout << x << std::endl;
```

或者

```
using std::cin;
using std::cout;
using std::endl;
int x;
cin >> x;
cout << x << endl;
```

:: 范围解析运算符

分类

1. 全局作用域符（::name）：用于类型名称（类、类成员、成员函数、变量等）前，表示作用域为全局命名空间
2. 类作用域符（class::name）：用于表示指定类型的作用域范围是具体某个类的
3. 命名空间作用域符（namespace::name）:用于表示指定类型的作用域范围是具体某个命名空间的

:: 使用

```
int count = 0;          // 全局（::）的 count

class A {
public:
    static int count; // 类 A 的 count（A::count）
};

int main() {
    ::count = 1;        // 设置全局的 count 的值为 1

    A::count = 2;       // 设置类 A 的 count 为 2

    int count = 0;      // 局部的 count
    count = 3;          // 设置局部的 count 的值为 3

    return 0;
}
```

enum 枚举类型

限定作用域的枚举类型（class）

```
enum class open_modes { input, output, append };
```

不限定作用域的枚举类型

```
enum color { red, yellow, green };
enum { floatPrec = 6, doublePrec = 10 };
```

decltype

decltype 关键字用于检查实体的声明类型或表达式的类型及值分类。语法：

```
decltype ( expression )
```

decltype 使用

```
// 尾置返回允许我们在参数列表之后声明返回类型
template <typename It>
auto fcn(It beg, It end) -> decltype(*beg)
{
    // 处理序列
    return *beg;    // 返回序列中一个元素的引用
}
// 为了使用模板参数成员，必须用 typename
template <typename It>
auto fcn2(It beg, It end) -> typename remove_reference<decltype(*beg)>::type
{
    // 处理序列
    return *beg;    // 返回序列中一个元素的拷贝
}
```

引用

左值引用

常规引用，一般表示对象的身份。

右值引用

右值引用就是必须绑定右值（一个临时对象、将要销毁的对象）的引用，一般表示对象的值。

右值引用可实现转移语义（Move Sementics）和精确传递（Perfect Forwarding），它的主要目的有两个方面：

- 消除两个对象交互时不必要的对象拷贝，节省运算存储资源，提高效率。
- 能够更简洁、明确地定义泛型函数。

引用折叠

- X& &、X& &&、X&& & 可折叠成 X&
- X&& && 可折叠成 X&&

宏

- 宏定义可以实现类似于函数的功能，但是它终归不是函数，而宏定义中括弧中的“参数”也不是真的参数，在宏展开的时候对 “参数” 进行的是一对一的替换。

成员初始化列表

好处

- 更高效：少了一次调用默认构造函数的过程。
- 有些场合必须要用初始化列表：
  1. 常量成员，因为常量只能初始化不能赋值，所以必须放在初始化列表里面
  2. 引用类型，引用必须在定义的时候初始化，并且不能重新赋值，所以也要写在初始化列表里面
  3. 没有默认构造函数的类类型，因为使用初始化列表可以不必调用默认构造函数来初始化，而是直接调用拷贝构造函数初始化。

initializer_list 列表初始化

用花括号初始化器列表初始化一个对象，其中对应构造函数接受一个 std::initializer_list 参数.

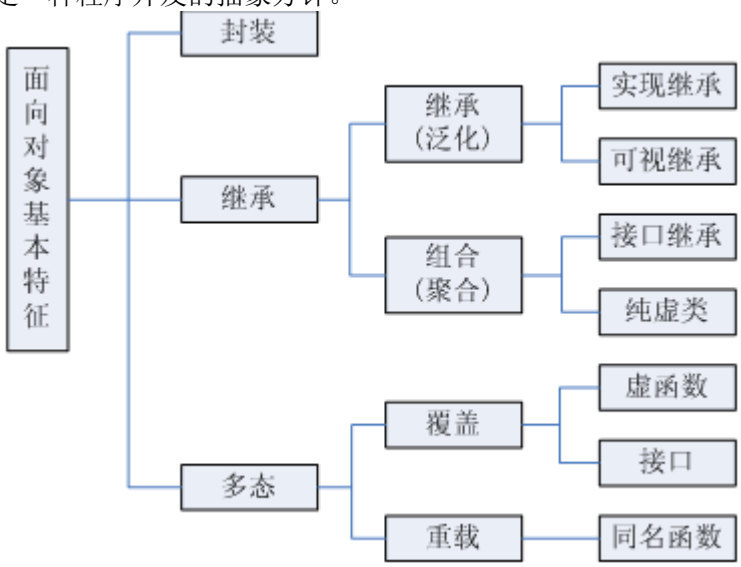
initializer_list 使用

```
#include <iostream>
#include <vector>
#include <initializer_list>

template <class T>
struct S {
    std::vector<T> v;
    S(std::initializer_list<T> l) : v(l) {
         std::cout << "constructed with a " << l.size() << "-element list\n";
    }
    void append(std::initializer_list<T> l) {
        v.insert(v.end(), l.begin(), l.end());
    }
    std::pair<const T*, std::size_t> c_arr() const {// std::pair 合并为一个结构体
        return {&v[0], v.size()};  // 在 return 语句中复制列表初始化
                                   // 这不使用 std::initializer_list
    }
};

template <typename T>
void templated_fn(T) {}

int main()
{
    S<int> s = {1, 2, 3, 4, 5}; // 复制初始化
    s.append({6, 7, 8});      // 函数调用中的列表初始化
```

```
    std::cout << "The vector size is now " << s.c_arr().second << " ints:\n";

    for (auto n : s.v)
        std::cout << n << ' ';
    std::cout << '\n';

    std::cout << "Range-for over brace-init-list: \n";

    for (int x : {-1, -2, -3}) // auto 的规则令此带范围 for 工作
        std::cout << x << ' ';
    std::cout << '\n';

    auto al = {10, 11, 12};   // auto 的特殊规则

    std::cout << "The list bound to auto has size() = " << al.size() << '\n';

//    templated_fn({1, 2, 3}); // 编译错误！“ {1, 2, 3} ”不是表达式，
                               // 它无类型，故 T 无法推导
    templated_fn<std::initializer_list<int>>({1, 2, 3}); // OK
    templated_fn<std::vector<int>>({1, 2, 3});           // 也 OK
}
```

面向对象

面向对象程序设计（Object-oriented programming，OOP）是种具有对象概念的程序编程典范，同时也是一种程序开发的抽象方针。



面向对象特征

面向对象三大特征 —— 封装、继承、多态

封装

把客观事物封装成抽象的类，并且类可以把自己的数据和方法只让可信的类或者对象操作，对不可信的进行信息隐藏。关键字：public, protected, private。不写默认为 private。

- `public` 成员：可以被任意实体访问
- `protected` 成员：只允许被子类及本类的成员函数访问
- `private` 成员：只允许被本类的成员函数访问

继承

- 基类（父类）——> 派生类（子类）

多态

- 多态，即多种状态（形态）。简单来说，我们将多态定义为消息以多种形式显示的能力。
- 多态是以封装和继承为基础的。
- C++ 多态分类及实现：
    1. 重载多态（Ad-hoc Polymorphism，编译期）：函数重载、运算符重载
    2. 子类型多态（Subtype Polymorphism，运行期）：虚函数
    3. 参数多态性（Parametric Polymorphism，编译期）：类模板、函数模板
    4. 强制多态（Coercion Polymorphism，编译期/运行期）：基本类型转换、自定义类型转换

The Four Polymorphisms in C++

静态多态（编译期/早绑定）

函数重载

```
class A
{
public:
    void do(int a);
    void do(int a, int b);
};
```

动态多态（运行期期/晚绑定）

- 虚函数：用 virtual 修饰成员函数，使其成为虚函数

注意：

- 普通函数（非类成员函数）不能是虚函数
- 静态函数（static）不能是虚函数
- 构造函数不能是虚函数（因为在调用构造函数时，虚表指针并没有在对象的内存空间中，必须要构造函数调用完成后才会形成虚表指针）
- 内联函数不能是表现多态性时的虚函数，解释见：虚函数（virtual）可以是内联函数（inline）吗？

动态多态使用

```
class Shape                     // 形状类
{
public:
    virtual double calcArea()
    {
        ...
    }
    virtual ~Shape();                   //虚析构
};
class Circle : public Shape     // 圆形类
{
public:
    virtual double calcArea();
    ...
};
class Rect : public Shape       // 矩形类
{
public:
    virtual double calcArea();
    ...
};
int main()
{
    Shape * shape1 = new Circle(4.0);
    Shape * shape2 = new Rect(5.0, 6.0);
```

```
    shape1->calcArea();          // 调用圆形类里面的方法
    shape2->calcArea();          // 调用矩形类里面的方法
    delete shape1;
    shape1 = nullptr;
    delete shape2;
    shape2 = nullptr;
    return 0;
}
```

虚析构函数

虚析构函数是为了解决基类的指针指向派生类对象，并用基类的指针删除派生类对象。

虚析构函数使用

```
class Shape
{
public:
    Shape();                     // 构造函数不能是虚函数
    virtual double calcArea();
    virtual ~Shape();            // 虚析构函数
};
class Circle : public Shape      // 圆形类
{
public:
    virtual double calcArea();
    ...
};
int main()
{
    Shape * shape1 = new Circle(4.0);
    shape1->calcArea();
    delete shape1;  // 因为Shape 有虚析构函数，所以 delete 释放内存时，先调用子类析构函数，再调用基类析构函数，防止内存泄漏。（多态表现）
    shape1 = NULL;
    return 0;
}
```

纯虚函数

纯虚函数是一种特殊的虚函数，在基类中不能对虚函数给出有意义的实现，而把它声明为纯虚函数，它的实现留给该基类的派生类去做。

```
virtual int A() = 0;
```

虚函数、纯虚函数

- 类里如果声明了虚函数，这个函数是实现的，哪怕是空实现，它的作用就是为了能让这个函数在它的子类里面可以被覆盖，这样的话，这样编译器就可以使用后期绑定来达到多态了。纯虚函数只是一个接口，是个函数的声明而已，它要留到子类里去实现。
- 虚函数在子类里面也可以不重载的；但纯虚函数必须在子类去实现。
- 虚函数的类用于“实作继承”，继承接口的同时也继承了父类的实现。当然大家也可以完成自己的实现。纯虚函数关注的是接口的统一性，实现由子类完成。
- 带纯虚函数的类叫抽象类，这种类不能直接生成对象，而只有被继承，并重写其虚函数后，才能使用。抽象类和大家口头常说的虚基类还是有区别的，在 C# 中用 abstract 定义抽象类，而在 C++ 中有抽象类的概念，但是没有这个关键字。抽象类被继承后，子类可以继续是抽象类，也可以是普通类，而虚基类，是含有纯虚函数的类，它如果被继承，那么子类就必须实现虚基类里面的所有纯虚函数，其子类不能是抽象类。

CSDN . C++ 中的虚函数、纯虚函数区别和联系

虚函数指针、虚函数表

- 虚函数指针：在含有虚函数类的对象中，指向虚函数表，在运行时确定。

- 虚函数表：在程序只读数据段（`.rodata section`，见：目标文件存储结构），存放虚函数指针，如果派生类实现了基类的某个虚函数，则在虚表中覆盖原本基类的那个虚函数指针，在编译时根据类的声明创建。

C++中的虚函数(表)实现机制以及用 C 语言对其进行的模拟实现

虚继承

虚继承用于解决多继承条件下的菱形继承问题（浪费存储空间、存在二义性）。

底层实现原理与编译器相关，一般通过**虚基类指针**和**虚基类表**实现，每个虚继承的子类都有一个虚基类指针（占用一个指针的存储空间，4 字节）和虚基类表（不占用类对象的存储空间）（需要强调的是，虚基类依旧会在子类里面存在拷贝，只是仅仅最多存在一份而已，并不是不在子类里面了）；当虚继承的子类被当做父类继承时，虚基类指针也会被继承。

实际上，vbptr 指的是虚基类表指针（virtual base table pointer），该指针指向了一个虚基类表（virtual table），虚表中记录了虚基类与本类的偏移地址；通过偏移地址，这样就找到了虚基类成员，而虚继承也不用像普通多继承那样维持着公共基类（虚基类）的两份同样的拷贝，节省了存储空间。

虚继承、虚函数

- 相同之处：都利用了虚指针（均占用类的存储空间）和虚表（均不占用类的存储空间）
- 不同之处：
  - 虚继承
    - 虚基类依旧存在继承类中，只占用存储空间
    - 虚基类表存储的是虚基类相对直接继承类的偏移
  - 虚函数
    - 虚函数不占用存储空间
    - 虚函数表存储的是虚函数地址

模板类、成员模板、虚函数

- 模板类中可以使用虚函数
- 一个类（无论是普通类还是类模板）的成员模板（本身是模板的成员函数）不能是虚函数

抽象类、接口类、聚合类

- 抽象类：含有纯虚函数的类
- 接口类：仅含有纯虚函数的抽象类
- 聚合类：用户可以直接访问其成员，并且具有特殊的初始化语法形式。满足如下特点：
  - 所有成员都是 public
  - 没有定义任何构造函数
  - 没有类内初始化
  - 没有基类，也没有 virtual 函数

内存分配和管理

malloc、calloc、realloc、alloca

1. malloc：申请指定字节数的内存。申请到的内存中的初始值不确定。
2. calloc：为指定长度的对象，分配能容纳其指定个数的内存。申请到的内存的每一位（bit）都初始化为 0。
3. realloc：更改以前分配的内存长度（增加或减少）。当增加长度时，可能需将以前分配区的内容移到另一个足够大的区域，而新增区域内的初始值则不确定。
4. alloca：在栈上申请内存。程序在出栈的时候，会自动释放内存。但是需要注意的是，alloca 不具可移植性, 而且在没有传统堆栈的机器上很难实现。alloca 不宜使用在必须广泛移植的程序中。C99 中支持变长数组 (VLA)，可以用来替代 alloca。

malloc、free

用于分配、释放内存

malloc、free 使用

申请内存，确认是否申请成功

```
char *str = (char*) malloc(100);
assert(str != nullptr);
```
释放内存后指针置空
```
free(p);
p = nullptr;
```
new、delete
1. new / new[]：完成两件事，先底层调用 malloc 分配了内存，然后调用构造函数（创建对象）。
2. delete/delete[]：也完成两件事，先调用析构函数（清理资源），然后底层调用 free 释放空间。
3. new 在申请内存时会自动计算所需字节数，而 malloc 则需我们自己输入申请内存空间的字节数。

new、delete 使用

申请内存，确认是否申请成功
```
int main()
{
    T* t = new T();      // 先内存分配 ， 再构造函数
    delete t;            // 先析构函数，再内存释放
    return 0;
}
```
定位 new
定位 new（placement new）允许我们向 new 传递额外的地址参数，从而在预先指定的内存区域创建对象。
```
new (place_address) type
new (place_address) type (initializers)
new (place_address) type [size]
new (place_address) type [size] { braced initializer list }
```
- `place_address` 是个指针
- `initializers` 提供一个（可能为空的）以逗号分隔的初始值列表

delete this 合法吗？
Is it legal (and moral) for a member function to say delete this?

合法，但：
1. 必须保证 this 对象是通过 `new`（不是 `new[]`、不是 placement new、不是栈上、不是全局、不是其他对象成员）分配的
2. 必须保证调用 `delete this` 的成员函数是最后一个调用 this 的成员函数
3. 必须保证成员函数的 `delete this` 后面没有调用 this 了
4. 必须保证 `delete this` 后没有人使用了

如何定义一个只能在堆上（栈上）生成对象的类？
如何定义一个只能在堆上（栈上）生成对象的类?

只能在堆上
方法：将析构函数设置为私有

原因：C++ 是静态绑定语言，编译器管理栈上对象的生命周期，编译器在为类对象分配栈空间时，会先检查类的析构函数的访问性。若析构函数不可访问，则不能在栈上创建对象。

只能在栈上
方法：将 new 和 delete 重载为私有

原因：在堆上生成对象，使用 new 关键词操作，其过程分为两阶段：第一阶段，使用 new 在堆上寻找可用内存，分配给对象；第二阶段，调用构造函数生成对象。将 new 操作设置为私有，那么第一阶段就无法完成，就不能够在堆上生成对象。

智能指针
C++ 标准库（STL）中
头文件：`#include <memory>`
C++ 98
```
std::auto_ptr<std::string> ps (new std::string(str));
```

C++ 11

1. shared_ptr
2. unique_ptr
3. weak_ptr
4. auto_ptr（被 C++11 弃用）

- Class shared_ptr 实现共享式拥有（shared ownership）概念。多个智能指针指向相同对象，该对象和其相关资源会在 “最后一个 reference 被销毁” 时被释放。为了在结构较复杂的情景中执行上述工作，标准库提供 weak_ptr、bad_weak_ptr 和 enable_shared_from_this 等辅助类。
- Class unique_ptr 实现独占式拥有（exclusive ownership）或严格拥有（strict ownership）概念，保证同一时间内只有一个智能指针可以指向该对象。你可以移交拥有权。它对于避免内存泄漏（resource leak）——如 new 后忘记 delete ——特别有用。

shared_ptr

多个智能指针可以共享同一个对象，对象的最末一个拥有着有责任销毁对象，并清理与该对象相关的所有资源。

- 支持定制型删除器（custom deleter），可防范 Cross-DLL 问题（对象在动态链接库（DLL）中被 new 创建，却在另一个 DLL 内被 delete 销毁）、自动解除互斥锁

weak_ptr

weak_ptr 允许你共享但不拥有某对象，一旦最末一个拥有该对象的智能指针失去了所有权，任何 weak_ptr 都会自动成空（empty）。因此，在 default 和 copy 构造函数之外，weak_ptr 只提供 “接受一个 shared_ptr” 的构造函数。

- 可打破环状引用（cycles of references，两个其实已经没有被使用的对象彼此互指，使之看似还在 “被使用” 的状态）的问题

unique_ptr

unique_ptr 是 C++11 才开始提供的类型，是一种在异常时可以帮助避免资源泄漏的智能指针。采用独占式拥有，意味着可以确保一个对象和其相应的资源同一时间只被一个 pointer 拥有。一旦拥有着被销毁或编程 empty，或开始拥有另一个对象，先前拥有的那个对象就会被销毁，其任何相应资源亦会被释放。

- unique_ptr 用于取代 auto_ptr

auto_ptr

被 c++11 弃用，原因是缺乏语言特性如 “针对构造和赋值” 的 `std::move` 语义，以及其他瑕疵。

auto_ptr 与 unique_ptr 比较

- auto_ptr 可以赋值拷贝，复制拷贝后所有权转移；unqiue_ptr 无拷贝赋值语义，但实现了 `move` 语义；
- auto_ptr 对象不能管理数组（析构调用 `delete`），unique_ptr 可以管理数组（析构调用 `delete[]`）；

强制类型转换运算符

MSDN . 强制转换运算符

static_cast

- 用于非多态类型的转换
- 不执行运行时类型检查（转换安全性不如 dynamic_cast）
- 通常用于转换数值数据类型（如 float -> int）
- 可以在整个类层次结构中移动指针，子类转化为父类安全（向上转换），父类转化为子类不安全（因为子类可能有不在父类的字段或方法）

向上转换是一种隐式转换。

dynamic_cast

- 用于多态类型的转换
- 执行行运行时类型检查
- 只适用于指针或引用
- 对不明确的指针的转换将失败（返回 nullptr），但不引发异常

- 可以在整个类层次结构中移动指针，包括向上转换、向下转换

const_cast

- 用于删除 const、volatile 和 __unaligned 特性（如将 const int 类型转换为 int 类型 ）

reinterpret_cast

- 用于位的简单重新解释
- 滥用 reinterpret_cast 运算符可能很容易带来风险。 除非所需转换本身是低级别的，否则应使用其他强制转换运算符之一。
- 允许将任何指针转换为任何其他指针类型（如 `char*` 到 `int*` 或 `One_class*` 到 `Unrelated_class*` 之类的转换，但其本身并不安全）
- 也允许将任何整数类型转换为任何指针类型以及反向转换。
- reinterpret_cast 运算符不能丢掉 const、volatile 或 __unaligned 特性。
- reinterpret_cast 的一个实际用途是在哈希函数中，即，通过让两个不同的值几乎不以相同的索引结尾的方式将值映射到索引。

bad_cast

- 由于强制转换为引用类型失败，dynamic_cast 运算符引发 bad_cast 异常。

bad_cast 使用

```
try {
    Circle& ref_circle = dynamic_cast<Circle&>(ref_shape);
}
catch (bad_cast b) {
    cout << "Caught: " << b.what();
}
```

运行时类型信息 (RTTI)

dynamic_cast

- 用于多态类型的转换

typeid

- typeid 运算符允许在运行时确定对象的类型
- type_id 返回一个 type_info 对象的引用
- 如果想通过基类的指针获得派生类的数据类型，基类必须带有虚函数
- 只能获取对象的实际类型

type_info

- type_info 类描述编译器在程序中生成的类型信息。 此类的对象可以有效存储指向类型的名称的指针。 type_info 类还可存储适合比较两个类型是否相等或比较其排列顺序的编码值。 类型的编码规则和排列顺序是未指定的，并且可能因程序而异。
- 头文件：`typeinfo`

typeid、type_info 使用

```
class Flyable                       // 能飞的
{
public:
    virtual void takeoff() = 0;     // 起飞
    virtual void land() = 0;        // 降落
};
class Bird : public Flyable         // 鸟
{
public:
    void foraging() {...}           // 觅食
    virtual void takeoff() {...}
    virtual void land() {...}
};
class Plane : public Flyable        // 飞机
{
```

```
public:
    void carry() {...}              // 运输
    virtual void take off() {...}
    virtual void land() {...}
};

class type_info
{
public:
    const char* name() const;
    bool operator == (const type_info & rhs) const;
    bool operator != (const type_info & rhs) const;
    int before(const type_info & rhs) const;
    virtual ~type_info();
private:
    ...
};

class doSomething(Flyable *obj)                 // 做些事情
{
    obj->takeoff();

    cout << typeid(*obj).name() << endl;        // 输出传入对象类型（"class Bird" or
"class Plane"）

    if(typeid(*obj) == typeid(Bird))            // 判断对象类型
    {
        Bird *bird = dynamic_cast<Bird *>(obj); // 对象转化
        bird->foraging();
    }

    obj->land();
};
```

Effective C++

1. 视 C++ 为一个语言联邦（C、Object-Oriented C++、Template C++、STL）
2. 宁可以编译器替换预处理器（尽量以 `const`、`enum`、`inline` 替换 `#define`）
3. 尽可能使用 const
4. 确定对象被使用前已先被初始化（构造时赋值（copy 构造函数）比 default 构造后赋值（copy assignment）效率高）
5. 了解 C++ 默默编写并调用哪些函数（编译器暗自为 class 创建 default 构造函数、copy 构造函数、copy assignment 操作符、析构函数）
6. 若不想使用编译器自动生成的函数，就应该明确拒绝（将不想使用的成员函数声明为 private，并且不予实现）
7. 为多态基类声明 virtual 析构函数（如果 class 带有任何 virtual 函数，它就应该拥有一个 virtual 析构函数）
8. 别让异常逃离析构函数（析构函数应该吞下不传播异常，或者结束程序，而不是吐出异常；如果要处理异常应该在非析构的普通函数处理）
9. 绝不在构造和析构过程中调用 virtual 函数（因为这类调用从不下降至 derived class）
10. 令 `operator=` 返回一个 `reference to *this`（用于连锁赋值）
11. 在 `operator=` 中处理“自我赋值”
12. 赋值对象时应确保复制“对象内的所有成员变量”及“所有 base class 成分”（调用基类复制构造函数）

13. 以对象管理资源（资源在构造函数获得，在析构函数释放，建议使用智能指针，资源取得时机便是初始化时机（Resource Acquisition Is Initialization，RAII））
14. 在资源管理类中小心 copying 行为（普遍的 RAII class copying 行为是：抑制 copying、引用计数、深度拷贝、转移底部资源拥有权（类似 auto_ptr））
15. 在资源管理类中提供对原始资源（raw resources）的访问（对原始资源的访问可能经过显式转换或隐式转换，一般而言显示转换比较安全，隐式转换对客户比较方便）
16. 成对使用 new 和 delete 时要采取相同形式（`new` 中使用 `[]` 则 `delete []`，`new` 中不使用 `[]` 则 `delete`）
17. 以独立语句将 newed 对象存储于（置入）智能指针（如果不这样做，可能会因为编译器优化，导致难以察觉的资源泄漏）
18. 让接口容易被正确使用，不易被误用（促进正常使用的办法：接口的一致性、内置类型的行为兼容；阻止误用的办法：建立新类型，限制类型上的操作，约束对象值、消除客户的资源管理责任）
19. 设计 class 犹如设计 type，需要考虑对象创建、销毁、初始化、赋值、值传递、合法值、继承关系、转换、一般化等等。
20. 宁以 pass-by-reference-to-const 替换 pass-by-value （前者通常更高效、避免切割问题（slicing problem），但不适用于内置类型、STL 迭代器、函数对象）
21. 必须返回对象时，别妄想返回其 reference（绝不返回 pointer 或 reference 指向一个 local stack 对象，或返回 reference 指向一个 heap-allocated 对象，或返回 pointer 或 reference 指向一个 local static 对象而有可能同时需要多个这样的对象。）
22. 将成员变量声明为 private（为了封装、一致性、对其读写精确控制等）
23. 宁以 non-member、non-friend 替换 member 函数（可增加封装性、包裹弹性（packaging flexibility）、机能扩充性）
24. 若所有参数（包括被 this 指针所指的那个隐喻参数）皆须要类型转换，请为此采用 non-member 函数
25. 考虑写一个不抛异常的 swap 函数
26. 尽可能延后变量定义式的出现时间（可增加程序清晰度并改善程序效率）
27. 尽量少做转型动作（旧式：`(T)expression`、`T(expression)`；新式：`const_cast<T>(expression)`、`dynamic_cast<T>(expression)`、`reinterpret_cast<T>(expression)`、`static_cast<T>(expression)`、；尽量避免转型、注重效率避免 dynamic_casts、尽量设计成无需转型、可把转型封装成函数、宁可用新式转型）
28. 避免使用 handles（包括 引用、指针、迭代器）指向对象内部（以增加封装性、使 const 成员函数的行为更像 const、降低 “虚吊号码牌”（dangling handles，如悬空指针等）的可能性）
29. 为 “异常安全” 而努力是值得的（异常安全函数（Exception-safe functions）即使发生异常也不会泄露资源或允许任何数据结构败坏，分为三种可能的保证：基本型、强列型、不抛异常型）
30. 透彻了解 inlining 的里里外外（inlining 在大多数 C++ 程序中是编译期的行为；inline 函数是否真正 inline，取决于编译器；大部分编译器拒绝太过复杂（如带有循环或递归）的函数 inlining，而所有对 virtual 函数的调用（除非是最平淡无奇的）也都会使 inlining 落空；inline 造成的代码膨胀可能带来效率损失；inline 函数无法随着程序库的升级而升级）
31. 将文件间的编译依存关系降至最低（如果使用 object references 或 object pointers 可以完成任务，就不要使用 objects；如果能过够，尽量以 class 声明式替换 class 定义式；为声明式和定义式提供不同的头文件）
32. 确定你的 public 继承塑模出 is-a（是一种）关系（适用于 base classes 身上的每一件事情一定适用于 derived classes 身上，因为每一个 derived class 对象也都是一个 base class 对象）
33. 避免遮掩继承而来的名字（可使用 using 声明式或转交函数（forwarding functions）来让被遮掩的名字再见天日）

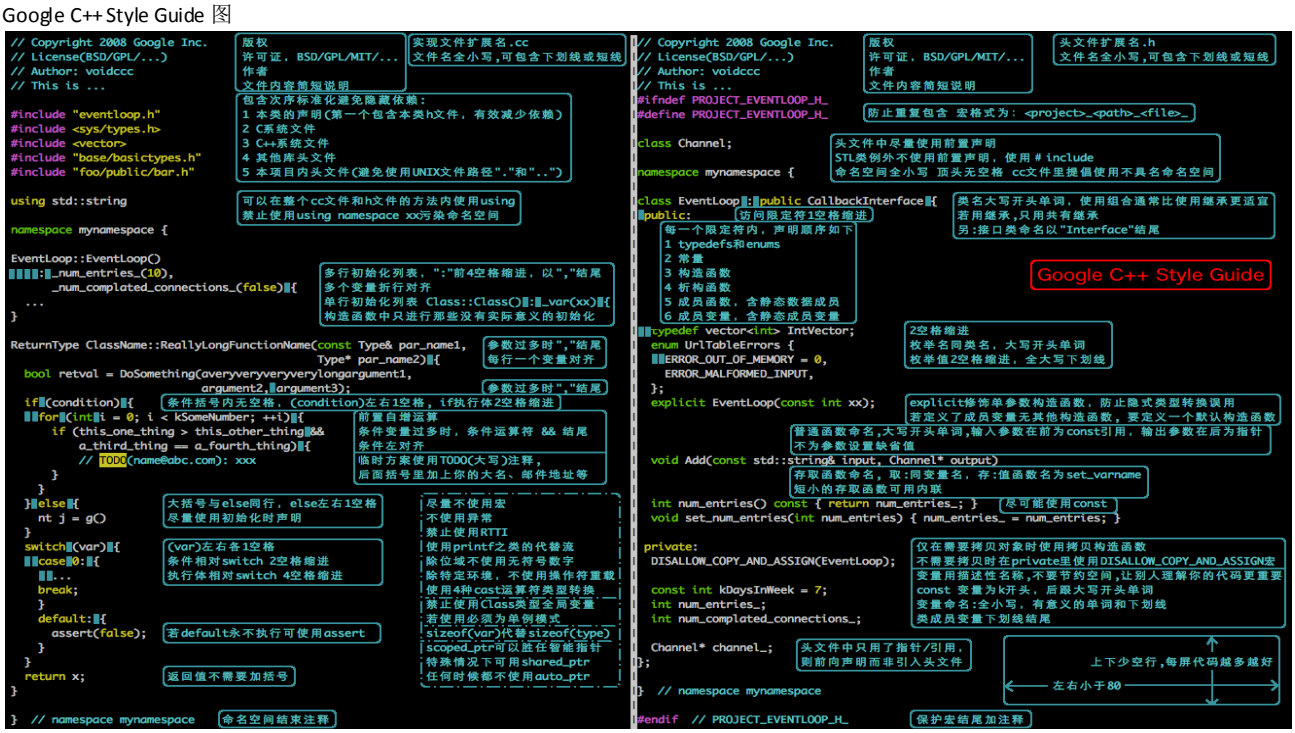
34. 区分接口继承和实现继承（在 public 继承之下，derived classes 总是继承 base class 的接口；pure virtual 函数只具体指定接口继承；非纯 impure virtual 函数具体指定接口继承及缺省实现继承；non-virtual 函数具体指定接口继承以及强制性实现继承）
35. 考虑 virtual 函数以外的其他选择（如 Template Method 设计模式的 non-virtual interface（NVI）手法，将 virtual 函数替换为“函数指针成员变量”，以 `tr1::function` 成员变量替换 virtual 函数，将继承体系内的 virtual 函数替换为另一个继承体系内的 virtual 函数）
36. 绝不重新定义继承而来的 non-virtual 函数
37. 绝不重新定义继承而来的缺省参数值，因为缺省参数值是静态绑定（statically bound），而 virtual 函数却是动态绑定（dynamically bound）
38. 通过复合塑模 has-a（有一个）或“根据某物实现出”（在应用域（application domain），复合意味 has-a（有一个）；在实现域（implementation domain），复合意味着 is-implemented-in-terms-of（根据某物实现出））
39. 明智而审慎地使用 private 继承（private 继承意味着 is-implemented-in-terms-of（根据某物实现出），尽可能使用复合，当 derived class 需要访问 protected base class 的成员，或需要重新定义继承而来的时候 virtual 函数，或需要 empty base 最优化时，才使用 private 继承）
40. 明智而审慎地使用多重继承（多继承比单一继承复杂，可能导致新的歧义性，以及对 virtual 继承的需要，但确有正当用途，如“public 继承某个 interface class”和“private 继承某个协助实现的 class”；virtual 继承可解决多继承下菱形继承的二义性问题，但会增加大小、速度、初始化及赋值的复杂度等等成本）
41. 了解隐式接口和编译期多态（class 和 templates 都支持接口（interfaces）和多态（polymorphism）；class 的接口是以签名为中心的显式的（explicit），多态则是通过 virtual 函数发生于运行期；template 的接口是奠基于有效表达式的隐式的（implicit），多态则是通过 template 具现化和函数重载解析（function overloading resolution）发生于编译期）
42. 了解 typename 的双重意义（声明 template 类型参数是，前缀关键字 class 和 typename 的意义完全相同；请使用关键字 typename 标识嵌套从属类型名称，但不得在基类列（base class lists）或成员初值列（member initialization list）内以它作为 base class 修饰符）
43. 学习处理模板化基类内的名称（可在 derived class templates 内通过 `this->` 指涉 base class templates 内的成员名称，或藉由一个明白写出的“base class 资格修饰符”完成）
44. 将与参数无关的代码抽离 templates（因类型模板参数（non-type template parameters）而造成代码膨胀往往可以通过函数参数或 class 成员变量替换 template 参数来消除；因类型参数（type parameters）而造成的代码膨胀往往可以通过让带有完全相同二进制表述（binary representations）的实现类型（instantiation types）共享实现码）
45. 运用成员函数模板接受所有兼容类型（请使用成员函数模板（member function templates）生成“可接受所有兼容类型”的函数；声明 member templates 用于“泛化 copy 构造”或“泛化 assignment 操作”时还需要声明正常的 copy 构造函数和 copy assignment 操作符）
46. 需要类型转换时请为模板定义非成员函数（当我们编写一个 class template，而它所提供之“与此 template 相关的”函数支持“所有参数之隐式类型转换”时，请将那些函数定义为“class template 内部的 friend 函数”）
47. 请使用 traits classes 表现类型信息（traits classes 通过 templates 和“templates 特化”使得“类型相关信息”在编译期可用，通过重载技术（overloading）实现在编译期对类型执行 if...else 测试）
48. 认识 template 元编程（模板元编程（TMP，template metaprogramming）可将工作由运行期移往编译期，因此得以实现早期错误侦测和更高的执行效率；TMP 可被用来生成“给予政策选择组合”（based on combinations of policy choices）的客户定制代码，也可用来避免生成对某些特殊类型并不适合的代码）
49. 了解 new-handler 的行为（set_new_handler 允许客户指定一个在内存分配无法获得满足时被调用的函数；nothrow new 是一个颇具局限的工具，因为它只适用于内存分配（operator new），后继的构造函数调用还是可能抛出异常）

50. 了解 new 和 delete 的合理替换时机（为了检测运用错误、收集动态分配内存之使用统计信息、增加分配和归还速度、降低缺省内存管理器带来的空间额外开销、弥补缺省分配器中的非最佳齐位、将相关对象成簇集中、获得非传统的行为）
51. 编写 new 和 delete 时需固守常规（operator new 应该内涵一个无穷循环，并在其中尝试分配内存，如果它无法满足内存需求，就应该调用 new-handler，它也应该有能力处理 0 bytes 申请，class 专属版本则还应该处理“比正确大小更大的（错误）申请”；operator delete 应该在收到 null 指针时不做任何事，class 专属版本则还应该处理“比正确大小更大的（错误）申请”）
52. 写了 placement new 也要写 placement delete（当你写一个 placement operator new，请确定也写出了对应的 placement operator delete，否则可能会发生隐微而时断时续的内存泄漏；当你声明 placement new 和 placement delete，请确定不要无意识（非故意）地遮掩了它们地正常版本）
53. 不要轻忽编译器的警告
54. 让自己熟悉包括 TR1 在内的标准程序库（TR1，C++ Technical Report 1，C++11 标准的草稿文件）
55. 让自己熟悉 Boost（准标准库）

Google C++ Style Guide

英文：Google C++ Style Guide
中文：C++ 风格指南

Google C++ Style Guide 图



Google C++ Style Guide

图片来源于：CSDN . 一张图总结 Google C++编程规范(Google C++ Style Guide)

其他

- Bjarne Stroustrup 的常见问题
- Bjarne Stroustrup 的 C++ 风格和技巧常见问题

STL

STL 容器

容器	底层数据结构	时间复杂度	有无序	可不可重复	其他
array	数组	随机读改 O(1)	无序	可重复	支持快速随机访问
vector	数组	随机读改、尾部插入、尾部删除 O(1)头部插入、头部删	无序	可重复	支持快速随机访问

		除 O(n)			
list	双向链表	插入、删除 O(1)随机读改 O(n)	无序	可重复	支持快速增删
deque	双端队列	头尾插入、头尾删除 O(1)	无序	可重复	一个中央控制器 + 多个缓冲区，支持首尾快速增删，支持随机访问
stack	deque / list	顶部插入、顶部删除 O(1)	无序	可重复	deque 或 list 封闭头端开口，不用 vector 的原因应该是容量大小有限制，扩容耗时
queue	deque / list	尾部插入、头部删除 O(1)	无序	可重复	deque 或 list 封闭头端开口，不用 vector 的原因应该是容量大小有限制，扩容耗时
priority_queue	vector + max-heap	插入、删除 O(log2n)	有序	可重复	vector 容器+heap 处理规则
set	红黑树	插入、删除、查找 O(log2n)	有序	不可重复	
multiset	红黑树	插入、删除、查找 O(log2n)	有序	可重复	
map	红黑树	插入、删除、查找 O(log2n)	有序	不可重复	
multimap	红黑树	插入、删除、查找 O(log2n)	有序	可重复	
hash_set	哈希表	插入、删除、查找 O(1) 最差 O(n)	无序	不可重复	
hash_multiset	哈希表	插入、删除、查找 O(1) 最差 O(n)	无序	可重复	
hash_map	哈希表	插入、删除、查找 O(1) 最差 O(n)	无序	不可重复	
hash_multimap	哈希表	插入、删除、查找 O(1) 最差 O(n)	无序	可重复	

STL 算法

算法	底层算法	时间复杂度	可不可重复
find	顺序查找	O(n)	可重复
sort	内省排序	O(n*log2n)	可重复

STL 索引

array

array 是固定大小的顺序容器，它们保存了一个以严格的线性顺序排列的特定数量的元素。

方法	含义
begin	返回指向数组容器中第一个元素的迭代器
end	返回指向数组容器中最后一个元素之后的理论元素的迭代器
rbegin	返回指向数组容器中最后一个元素的反向迭代器
rend	返回一个反向迭代器，指向数组中第一个元素之前的理论元素
cbegin	返回指向数组容器中第一个元素的常量迭代器（const_iterator）
cend	返回指向数组容器中最后一个元素之后的理论元素的常量迭代器（const_iterator）

crbegin	返回指向数组容器中最后一个元素的常量反向迭代器（const*reverse*iterator）
crend	返回指向数组中第一个元素之前的理论元素的常量反向迭代器（const*reverse*iterator）
size	返回数组容器中元素的数量
max_size	返回数组容器可容纳的最大元素数
empty	返回一个布尔值，指示数组容器是否为空
operator[]	返回容器中第 n（参数）个位置的元素的引用
at	返回容器中第 n（参数）个位置的元素的引用
front	返回对容器中第一个元素的引用
back	返回对容器中最后一个元素的引用
data	返回指向容器中第一个元素的指针
fill	用 val（参数）填充数组所有元素
swap	通过 x（参数）的内容交换数组的内容
get（array）	形如 `std::get<0>(myarray)`；传入一个数组容器，返回指定位置元素的引用
relational operators (array)	形如 `arrayA > arrayB`；依此比较数组每个元素的大小关系

vector

vector 是表示可以改变大小的数组的序列容器。

方法	含义
vector	构造函数
~vector	析构函数，销毁容器对象
operator=	将新内容分配给容器，替换其当前内容，并相应地修改其大小
begin	返回指向容器中第一个元素的迭代器
end	返回指向容器中最后一个元素之后的理论元素的迭代器
rbegin	返回指向容器中最后一个元素的反向迭代器
rend	返回一个反向迭代器，指向中第一个元素之前的理论元素
cbegin	返回指向容器中第一个元素的常量迭代器（const_iterator）
cend	返回指向容器中最后一个元素之后的理论元素的常量迭代器（const_iterator）
crbegin	返回指向容器中最后一个元素的常量反向迭代器（const*reverse*iterator）
crend	返回指向容器中第一个元素之前的理论元素的常量反向迭代器（const*reverse*iterator）
size	返回容器中元素的数量
max_size	返回容器可容纳的最大元素数
resize	调整容器的大小，使其包含 n（参数）个元素
capacity	返回当前为 vector 分配的存储空间（容量）的大小
empty	返回 vector 是否为空
reserve	请求 vector 容量至少足以包含 n（参数）个元素
shrink*to*fit	要求容器减小其 capacity（容量）以适应其 size（元素数量）
operator[]	返回容器中第 n（参数）个位置的元素的引用
at	返回容器中第 n（参数）个位置的元素的引用
front	返回对容器中第一个元素的引用
back	返回对容器中最后一个元素的引用
data	返回指向容器中第一个元素的指针
assign	将新内容分配给 vector，替换其当前内容，并相应地修改其 size
push_back	在容器的最后一个元素之后添加一个新元素
pop_back	删除容器中的最后一个元素，有效地将容器 size 减少一个
insert	通过在指定位置的元素之前插入新元素来扩展该容器，通过插入元素的数量有效地增加容器大小
erase	从 vector 中删除单个元素（`position`）或一系列元素（`[first，last)`），这有效地

	减少了被去除的元素的数量，从而破坏了容器的大小
swap	通过 x（参数）的内容交换容器的内容，x 是另一个类型相同、size 可能不同的 vector 对象
clear	从 vector 中删除所有的元素（被销毁），留下 size 为 0 的容器
emplace	通过在 position（参数）位置处插入新元素 args（参数）来扩展容器
emplace_back	在 vector 的末尾插入一个新的元素，紧跟在当前的最后一个元素之后
get_allocator	返回与 vector 关联的构造器对象的副本
swap(vector)	容器 x（参数）的内容与容器 y（参数）的内容交换。两个容器对象都必须是相同的类型（相同的模板参数），尽管大小可能不同
relational operators (vector)	形如 `vectorA > vectorB`；依此比较每个元素的大小关系

deque

deque（['dek]）（双端队列）是 double-ended queue 的一个不规则缩写。deque 是具有动态大小的序列容器，可以在两端（前端或后端）扩展或收缩。

方法	含义
deque	构造函数
push_back	在当前的最后一个元素之后，在 deque 容器的末尾添加一个新元素
push_front	在 deque 容器的开始位置插入一个新的元素，位于当前的第一个元素之前
pop_back	删除 deque 容器中的最后一个元素，有效地将容器大小减少一个
pop_front	删除 deque 容器中的第一个元素，有效地减小其大小
emplace_front	在 deque 的开头插入一个新的元素，就在其当前的第一个元素之前
emplace_back	在 deque 的末尾插入一个新的元素，紧跟在当前的最后一个元素之后

forward_list

forward_list（单向链表）是序列容器，允许在序列中的任何地方进行恒定的时间插入和擦除操作。

方法	含义
forward_list	返回指向容器中第一个元素之前的位置的迭代器
cbefore_begin	返回指向容器中第一个元素之前的位置的 const_iterator

list

list，双向链表，是序列容器，允许在序列中的任何地方进行常数时间插入和擦除操作，并在两个方向上进行迭代。

stack

stack 是一种容器适配器，用于 LIFO（后进先出）的操作，其中元素仅从容器的一端插入和提取。

queue

queue 是一种容器适配器，用于在 FIFO（先入先出）的操作，其中元素插入到容器的一端并从另一端提取。

priority_queue

set

set 是按照特定顺序存储唯一元素的容器。

multiset

map

map 是关联容器，按照特定顺序存储由 key value (键值) mapped value (映射值) 组合形成的元素。

方法	含义
map	构造函数
begin	返回引用容器中第一个元素的迭代器
key_comp	返回容器用于比较键的比较对象的副本
value_comp	返回可用于比较两个元素的比较对象，以获取第一个元素的键是否在第二个元素之前
find	在容器中搜索具有等于 k（参数）的键的元素，如果找到则返回迭代器，否则返回 map::end 的迭代器
count	在容器中搜索具有等于 k（参数）的键的元素，并返回匹配的数量

lower_bound	返回一个非递减序列 [first, last)（参数）中的第一个大于等于值 val（参数）的位置的迭代器
upper_bound	返回一个非递减序列 [first, last)（参数）中第一个大于 val（参数）的位置的迭代器
equal_range	获取相同元素的范围，返回包含容器中所有具有与 k（参数）等价的键的元素的范围边界（pair< map<char,int>::iterator, map<char,int>::iterator >）

multimap

unordered_set

unordered_multiset

unordered_map

unordered_multimap

tuple

元组是一个能够容纳元素集合的对象。每个元素可以是不同的类型。

pair

这个类把一对值（values）结合在一起，这些值可能是不同的类型（T1 和 T2）。每个值可以被公有的成员变量 first、second 访问。

算　法

```
// 简单查找算法，要求输入迭代器（input iterator）
find(beg, end, val); //返回一个迭代器指向输入序列中第一个等于 val 的元素，未找到返回 end
find_if(beg, end, unaryPred); //返回迭代器，指向第一个满足 unaryPred 的元素，未找到返回 end
find_if_not(beg, end, unaryPred); // 返回迭代器，指向第一个令 unaryPred 为 false 的元素，未找到返回 end
count(beg, end, val); // 返回一个计数器，指出 val 出现了多少次
count_if(beg, end, unaryPred); // 统计有多少个元素满足 unaryPred
all_of(beg, end, unaryPred); // 返回一个 bool 值，判断是否所有元素都满足 unaryPred
any_of(beg, end, unaryPred); // 返回一个 bool 值，判断是否任意（存在）一个元素满足 unaryPred
none_of(beg, end, unaryPred); // 返回一个 bool 值，判断是否所有元素都不满足 unaryPred
// 查找重复值的算法，传入向前迭代器（forward iterator）
adjacent_find(beg, end); // 返回指向第一对相邻重复元素的迭代器，无相邻元素则返回 end
adjacent_find(beg, end, binaryPred); // 返回指向第一对相邻重复元素的迭代器，无相邻元素则返回 end
search_n(beg, end, count, val); // 返回迭代器，从此位置开始有 count 个相等元素，不存在则返回 end
search_n(beg, end, count, val, binaryPred); // 返回迭代器，从此位置开始有 count 个相等元素，不存在则返回 end
// 查找子序列算法，除 find_first_of（前两个输入迭代器，后两个前向迭代器） 外，都要求两个前向迭代器
search(beg1, end1, beg2, end2); // 返回第二个输入范围（子序列）在爹一个输入范围中第一次出现的位置，未找到则返回 end1
search(beg1, end1, beg2, end2, binaryPred); // 返回第二个输入范围（子序列）在爹一个输入范围中第一次出现的位置，未找到则返回 end1
find_first_of(beg1, end1, beg2, end2); // 返回迭代器，指向第二个输入范围中任意元素在第一个范围中首次出现的位置，未找到则返回 end1
find_first_of(beg1, end1, beg2, end2, binaryPred); // 返回迭代器，指向第二个输入范围中任意元素在第一个范围中首次出现的位置，未找到则返回 end1
find_end(beg1, end1, beg2, end2); // 类似 search，但返回的最后一次出现的位置。如果第二个输入范围为空，或者在第一个输入范围为空，或者在第一个输入范围中未找到它，则返回 end1
find_end(beg1, end1, beg2, end2, binaryPred); // 类似 search，但返回的最后一次出现的位置。如果第二个输入范围为空，或者在第一个输入范围为空，或者在第一个输入范围中未找到它，则返回 end1
// 其他只读算法，传入输入迭代器
for_each(beg, end, unaryOp); // 对输入序列中的每个元素应用可调用对象 unaryOp，unaryOp 的返回值被忽略
mismatch(beg1, end1, beg2); // 比较两个序列中的元素。返回一个迭代器的 pair，表示两个序列中第一个不匹配的元素
mismatch(beg1, end1, beg2, binaryPred); // 比较两个序列中的元素。返回一个迭代器的 pair，表示两个序列中第一个不匹配的元素
equal(beg1, end1, beg2); // 比较每个元素，确定两个序列是否相等。
equal(beg1, end1, beg2, binaryPred); // 比较每个元素，确定两个序列是否相等。
// 二分搜索算法，传入前向迭代器或随机访问迭代器（random-access iterator），要求序列中的元素已经是有序的。通过小于运算符（<）或 comp 比较操作实现比较。
lower_bound(beg, end, val); // 返回一个非递减序列 [beg, end) 中的第一个大于等于值 val 的位置的迭代器，不存在则返回 end
```

```
lower_bound(beg, end, val, comp); // 返回一个非递减序列 [beg, end) 中的第一个大于等于值 val 的位置的迭代器，不存在则返回 end
upper_bound(beg, end, val); // 返回一个非递减序列 [beg, end) 中第一个大于 val 的位置的迭代器，不存在则返回 end
upper_bound(beg, end, val, comp); // 返回一个非递减序列 [beg, end) 中第一个大于 val 的位置的迭代器，不存在则返回 end
equal_range(beg, end, val); // 返回一个 pair，其 first 成员是 lower_bound 返回的迭代器，其 second 成员是 upper_bound 返回的迭代器
binary_search(beg, end, val); // 返回一个 bool 值，指出序列中是否包含等于 val 的元素。对于两个值 x 和 y，当 x 不小于 y 且 y 也不小于 x 时，认为它们相等。
// 只写不读算法，要求输出迭代器（output iterator）
fill(beg, end, val); // 将 val 赋予每个元素，返回 void
fill_n(beg, cnt, val); // 将 val 赋予 cnt 个元素，返回指向写入到输出序列最有一个元素之后位置的迭代器
genetate(beg, end, Gen); // 每次调用 Gen() 生成不同的值赋予每个序列，返回 void
genetate_n(beg, cnt, Gen); // 每次调用 Gen() 生成不同的值赋予 cnt 个序列，返回指向写入到输出序列最有一个元素之后位置的迭代器
// 使用输入迭代器的写算法，读取一个输入序列，将值写入到一个输出序列（dest）中
copy(beg, end, dest); // 从输入范围将元素拷贝所有元素到 dest 指定定的目的序列
copy_if(beg, end, dest, unaryPred); // 从输入范围将元素拷贝满足 unaryPred 的元素到 dest 指定定的目的序列
copy_n(beg, n, dest); // 从输入范围将元素拷贝前 n 个元素到 dest 指定定的目的序列
move(beg, end, dest); // 对输入序列中的每个元素调用 std::move，将其移动到迭代器 dest 开始始的序列中
transform(beg, end, dest, unaryOp); // 调用给定操作（一元操作），并将结果写到dest 中
transform(beg, end, beg2, dest, binaryOp); // 调用给定操作（二元操作），并将结果写到dest 中
replace_copy(beg, end, dest, old_val, new_val); // 将每个元素拷贝到 dest，将等于 old_val 的的元素替换为 new_val
replace_copy_if(beg, end, dest, unaryPred, new_val); // 将每个元素拷贝到 dest，将满足 unaryPred 的的元素替换为 new_val
merge(beg1, end1, beg2, end2, dest); // 两个输入序列必须都是有序的，用 < 运算符将合并后的序列写入到 dest 中
merge(beg1, end1, beg2, end2, dest, comp); // 两个输入序列必须都是有序的，使用给定的比较操作（comp）将合并后的序列写入到 dest 中
// 使用前向迭代器的写算法，要求前向迭代器
iter_swap(iter1, iter2); // 交换 iter1 和 iter2 所表示的元素，返回 void
swap_ranges(beg1, end1, beg2); // 将输入范围中所有元素与 beg2 开始的第二个序列中所有元素进行交换。返回递增后的的 beg2，指向最后一个交换元素之后的位置。
replace(beg, end, old_val, new_val); // 用 new_val 替换等于 old_val 的每个匹配元素
replace_if(beg, end, unaryPred, new_val); // 用 new_val 替换满足 unaryPred 的每个匹配元素
// 使用双向迭代器的写算法，要求双向选代器（bidirectional iterator）
copy_backward(beg, end, dest); // 从输入范围中拷贝元素到指定目的位置。如果范围为空,则返回值为 dest；否则，返回值表示从 *beg 中拷贝或移动的元素。
move_backward(beg, end, dest);  //从输入范围中移动元素到指定目的位置。如果范围为空,则返回值为 dest；否则,返回值表示从 *beg 中拷贝或移动的元素。
inplace_merge(beg, mid, end); // 将同一个序列中的两个有序子序列合并为单一的有序序列。beg 到 mid 间的子序列和 mid 到 end 间的子序列被合并，并被写入到原序列中。使用 < 比较元素。
inplace_merge(beg, mid, end, comp); // 将同一个序列中的两个有序子序列合并为单一的有序序列。beg 到 mid 间的子序列和 mid 到 end 间的子序列被合并，并被写入到原序列中。使用给定的 comp
```

```
操作。
// 划分算法，要求双向迭代器（bidirectional iterator）
is_partitioned(beg, end, unaryPred); //如果所有满足谓词unaryPred 的元素都在不满足 unarypred 的元素之前，则返回 true。若序列为空，也返回 true
partition_copy(beg, end, dest1, dest2, unaryPred); // 将满足 unaryPred 的元素拷贝到到 dest1，并将不满足 unaryPred 的元素拷贝到到 dest2。返回一个迭代器 pair，其 first 成员表示拷贝到 dest1 的的元素的末尾，second 表示拷贝到 dest2 的元素的末尾。
partitioned_point(beg, end, unaryPred); // 输入序列必须是已经用 unaryPred 划分过的。返回满足 unaryPred 的范围的尾后迭代器。如果返回的迭代器不是 end，则它指向的元素及其后的元素必须都不满足 unaryPred
stable_partition(beg, end, unaryPred); // 使用 unaryPred 划分输入序列。满足 unaryPred 的元素放置在序列开始，不满足的元素放在序列尾部。返回迭代器，指向最后一个满足 unaryPred 的元素之后的位置如果所有元素都不满足 unaryPred，则返回 beg
partition(beg, end, unaryPred); // 使用 unaryPred 划分输入序列。满足 unaryPred 的元素放置在序列开始，不满足的元素放在序列尾部。返回迭代器，指向最后一个满足 unaryPred 的元素之后的位置如果所有元素都不满足 unaryPred，则返回 beg
// 排序算法，要求随机访问迭代器（random-access iterator）
sort(beg, end); // 排序整个范围
stable_sort(beg, end); // 排序整个范围（稳定排序）
sort(beg, end, comp); // 排序整个范围
stable_sort(beg, end, comp); // 排序整个范围（稳定排序）
is_sorted(beg, end); // 返回一个 bool 值，指出整个输入序列是否有序
is_sorted(beg, end, comp); // 返回一个 bool 值，指出整个输入序列是否有序
is_sorted_until(beg, end); // 在输入序列中查找最长初始有序子序列，并返回子序列的尾后迭代器
is_sorted_until(beg, end, comp); // 在输入序列中查找最长初始有序子序列，并返回子序列的尾后迭代器
partial_sort(beg, mid, end); // 排序 mid-beg 个元素。即，如果 mid-beg 等于 42，则此函数将值最小的 42 个元素有序放在序列前 42 个位置
partial_sort(beg, mid, end, comp); // 排序 mid-beg 个元素。即，如果 mid-beg 等于 42，则此函数将值最小的 42 个元素有序放在序列前 42 个位置
partial_sort_copy(beg, end, destBeg, destEnd); // 排序输入范围中的元素，并将足够多的已排序元素放到 destBeg 和 destEnd 所指示的序列中
partial_sort_copy(beg, end, destBeg, destEnd, comp); // 排序输入范围中的元素，并将足够多的已排序元素放到 destBeg 和 destEnd 所指示的序列中
nth_element(beg, nth, end); // nth 是一个迭代器，指向输入序列中第 n 大的元素。nth 之前的元素都小于等于它，而之后的元素都大于等于它
nth_element(beg, nth, end, comp); // nth 是一个迭代器，指向输入序列中第 n 大的元素。nth 之前的元素都小于等于它，而之后的元素都大于等于它
// 使用前向迭代器的重排算法。普通版本在输入序列自身内部重拍元素，_copy 版本完成重拍后写入到指定目的序列中，而不改变输入序列
remove(beg, end, val); // 通过用保留的元素覆盖要删除的元素实现删除 ==val 的元素，返回一个指向最后一个删除元素的尾后位置的迭代器
remove_if(beg, end, unaryPred);//通过用保留的元素覆盖要删除的元素实现删除满足 unaryPred 的元素，返回一个指向最后一个删除元素的尾后位置的迭代器
remove_copy(beg, end, dest, val); // 通过用保留的元素覆盖要删除的元素实现删除 ==val 的元素，返回一个指向最后一个删除元素的尾后位置的迭代器
remove_copy_if(beg, end, dest, unaryPred); // 通过用保留的元素覆盖要删除的元素实现删除满足 unaryPred 的元素，返回一个指向最后一个删除元素的尾后位置的迭代器
unique(beg, end); // 通过对覆盖相邻的重复元素（用 == 确定是否相同）实现重排序列。返回迭代器，指向不重复元素的尾后位置
unique (beg, end, binaryPred); // 通过对覆盖相邻的重复元素（用 binaryPred 确定是否相同）
```

```
实现重排序列。返回迭代器，指向不重复元素的尾后位置
unique_copy(beg, end, dest); // 通过对覆盖相邻的重复元素（用 == 确定是否相同）实现重排序列。返回迭代器，指向不重复元素的尾后位置
unique_copy_if(beg, end, dest, binaryPred); // 通过对覆盖相邻的重复元素（用 binaryPred 确定是否相同）实现重排序列。返回迭代器，指向不重复元素的尾后位置
rotate(beg, mid, end); // 围绕 mid 指向的元素进行元素转动。元素 mid 成为为首元素，随后是 mid+1 到到 end 之前的元素，再接着是 beg 到 mid 之前的元素。返回迭代器，指向原来在 beg 位置的元素
rotate_copy(beg, mid, end, dest); // 围绕 mid 指向的元素进行元素转动。元素 mid 成为为首元素，随后是 mid+1 到到 end 之前的元素，再接着是 beg 到 mid 之前的元素。返回迭代器，指向原来在 beg 位置的元素
// 使用双向迭代器的重排算法
reverse(beg, end); // 翻转序列中的元素，返回 void
reverse_copy(beg, end, dest);; // 翻转序列中的元素，返回迭代器，指向拷贝到目的序列的元素的尾后位置
// 使用随机访问迭代器的重排算法
random_shuffle(beg, end); // 混洗输入序列中的元素，返回 void
random_shuffle(beg, end, rand); // 混洗输入序列中的元素，rand 接受一个正整数的随机对象，返回 void
shuffle(beg, end, Uniform_rand); // 混洗输入序列中的元素，Uniform_rand 必须满足均匀分布随机数生成器的要求，返回 void
// 最小值和最大值，使用 < 运算符或给定的比较操作 comp 进行比较
min(val1, va12); // 返回 val1 和 val2 中的最小值，两个实参的类型必须完全一致。参数和返回类型都是 const 的引引用，意味着对象不会被拷贝。下略
min(val1, val2, comp);
min(init_list);
min(init_list, comp);
max(val1, val2);
max(val1, val2, comp);
max(init_list);
max(init_list, comp);
minmax(val1, val2); // 返回一个 pair，其 first 成员为提供的值中的较小者，second 成员为较大者。下略
minmax(vall, val2, comp);
minmax(init_list);
minmax(init_list, comp);
min_element(beg, end); // 返回指向输入序列中最小元素的迭代器
min_element(beg, end, comp); // 返回指向输入序列中最小元素的迭代器
max_element(beg, end); // 返回指向输入序列中最大元素的迭代器
max_element(beg, end, comp); // 返回指向输入序列中最大元素的迭代器
minmax_element(beg, end); // 返回一个 pair，其中 first 成员为最小元素，second 成员为最大元素
minmax_element(beg, end, comp); // 返回一个 pair，其中 first 成员为最小元素，second 成员为最大元素
// 字典序比较，根据第一对不相等的元素的相对大小来返回结果。如果第一个序列在字典序中小于第二个序列，则返回 true。否则，返回 fa1se。如果个序列比另一个短，且所有元素都与较长序列的对应元素相等，则较短序列在字典序中更小。如果序列长度相等，且对应元素都相等，则在字典序中任何一个都不大于另外一个。
lexicographical_compare(beg1, end1, beg2, end2);
lexicographical_compare(beg1, end1, beg2, end2, comp);
```

组成

- 容器（containers）
- 算法（algorithms）
- 迭代器（iterators）
- 仿函数（functors）
- 配接器（adapters）
- 空间配置器（allocator）

容器（containers）

- 序列式容器（sequence containers）：元素都是可序（ordered），但未必是有序（sorted）
- 关联式容器（associattive containers

array

array 是固定大小的顺序容器，它们保存了一个以严格的线性顺序排列的特定数量的元素。
在内部，一个数组除了它所包含的元素（甚至不是它的大小，它是一个模板参数，在编译时是固定的）以外不保存任何数据。存储大小与用语言括号语法（[]）声明的普通数组一样高效。这个类只是增加了一层成员函数和全局函数，所以数组可以作为标准容器使用。
与其他标准容器不同，数组具有固定的大小，并且不通过分配器管理其元素的分配：它们是封装固定大小数组元素的聚合类型。因此，他们不能动态地扩大或缩小。
零大小的数组是有效的，但是它们不应该被解除引用（成员的前面，后面和数据）。
与标准库中的其他容器不同，交换两个数组容器是一种线性操作，它涉及单独交换范围内的所有元素，这通常是相当低效的操作。另一方面，这允许迭代器在两个容器中的元素保持其原始容器关联。
数组容器的另一个独特特性是它们可以被当作元组对象来处理：array 头部重载 get 函数来访问数组元素，就像它是一个元组，以及专门的 tuple*size* *和* *tuple*element 类型。

```
template < class T, size_t N > class array;
```

array::begin

返回指向数组容器中第一个元素的迭代器。

```
      iterator begin() noexcept;
const_iterator begin() const noexcept;
```

```
#include <iostream>
#include <array>
int main()
{
    std::array<int, 5> myarray = {2, 16, 77,34, 50};
    std::cout << "myarray contains:";
    for(auto it = myarray.begin(); it != myarray.end(); ++i)
        std::cout << ' ' << *it;
    std::cout << '\n';

    return 0;
}
```

Output

```
myarray contains: 2 16 77 34 50
```

array::end

返回指向数组容器中最后一个元素之后的理论元素的迭代器。

```
      iterator end() noexcept;
const_iterator end() const noexcept;
```

Example

```
#include <iostream>
#include <array>
int main ()
{
    std::array<int,5> myarray = { 5, 19, 77, 34, 99 };

    std::cout << "myarray contains:";
    for ( auto it = myarray.begin(); it != myarray.end(); ++it )
        std::cout << ' ' << *it;

    std::cout << '\n';

    return 0;
}
```

Output

```
myarray contains: 5 19 77 34 99
```

array::rbegin

返回指向数组容器中最后一个元素的反向迭代器。

```
      reverse_iterator rbegin（）noexcept;
const_reverse_iterator rbegin（）const noexcept;
```

Example

```
#include <iostream>
#include <array>
int main ()
{
    std::array<int,4> myarray = {4, 26, 80, 14} ;
    for(auto rit = myarray.rbegin(); rit < myarray.rend(); ++rit)
        std::cout << ' ' << *rit;

    std::cout << '\n';

    return 0;
}
```

Output

```
myarray contains: 14 80 26 4
```

array::rend

返回一个反向迭代器，指向数组中第一个元素之前的理论元素（这被认为是它的反向结束）。

```
      reverse_iterator rend() noexcept;
const_reverse_iterator rend() const noexcept;
```

Example

```
#include <iostream>
#include <array>
int main ()
{
    std::array<int,4> myarray = {4, 26, 80, 14};
    std::cout << "myarray contains";
    for(auto rit = myarray.rbegin(); rit < myarray.rend(); ++rit)
        std::cout << ' ' << *rit;
```

```
    std::cout << '\n';
    return 0;
}
```

Output

```
myarray contains: 14 80 26 4
```

array::cbegin

返回指向数组容器中第一个元素的常量迭代器（const_iterator）；这个迭代器可以增加和减少，但是不能用来修改它指向的内容。

```
const_iterator cbegin（）const noexcept;
```

Example

```
#include <iostream>
#include <array>
int main ()
{
    std::array<int,5> myarray = {2, 16, 77, 34, 50};

    std::cout << "myarray contains:";

    for ( auto it = myarray.cbegin(); it != myarray.cend(); ++it )
        std::cout << ' ' << *it;   // cannot modify *it

    std::cout << '\n';

    return 0;
}
```

Output

```
myarray contains: 2 16 77 34 50
```

array::cend

返回指向数组容器中最后一个元素之后的理论元素的常量迭代器（const_iterator）。这个迭代器可以增加和减少，但是不能用来修改它指向的内容。

```
const_iterator cend() const noexcept;
```

Example

```
#include <iostream>
#include <array>
int main ()
{
    std::array<int,5> myarray = { 15, 720, 801, 1002, 3502 };

    std::cout << "myarray contains:";
    for ( auto it = myarray.cbegin(); it != myarray.cend(); ++it )
        std::cout << ' ' << *it;   // cannot modify *it

    std::cout << '\n';

    return 0;
}
```

Output

```
myarray contains: 2 16 77 34 50
```

array::crbegin

返回指向数组容器中最后一个元素的常量反向迭代器（const*reverse*iterator）

```
const_reverse_iterator crbegin () const noexcept;
```

Example

```
#include <iostream>
#include <array>
int main ()
{
    std::array<int,6> myarray = {10, 20, 30, 40, 50, 60} ;

    std::cout << "myarray backwards:";
    for ( auto rit=myarray.crbegin() ; rit < myarray.crend(); ++rit )
        std::cout << ' ' << *rit;   // cannot modify *rit
    std::cout << '\n';
    return 0;
}
```

Output

```
myarray backwards: 60 50 40 30 20 10
```

array::crend

返回指向数组中第一个元素之前的理论元素的常量反向迭代器（const*reverse*iterator），它被认为是其反向结束。

```
const_reverse_iterator crend() const noexcept;
```

```
#include <iostream>
#include <array>
int main ()
{
    std::array<int,6> myarray = {10, 20, 30, 40, 50, 60} ;

    std::cout << "myarray backwards:";
    for ( auto rit=myarray.crbegin() ; rit < myarray.crend(); ++rit )
        std::cout << ' ' << *rit;   // cannot modify *rit

    std::cout << '\n';

    return 0;
}
```

Output

```
myarray backwards: 60 50 40 30 20 10
```

array::size

返回数组容器中元素的数量。

```
constexpr size_type size () noexcept;
```

```
#include <iostream>
#include <array>
int main ()
{
    std::array<int,5> myints;
    std::cout << "size of myints:" << myints.size() << std::endl;
    std::cout << "sizeof(myints):" << sizeof(myints) << std::endl;

    return 0;
}
```

Possible Output

```
size of myints: 5
sizeof(myints): 20
```

array::max_size

返回数组容器可容纳的最大元素数。数组对象的 max_size 与其 size 一样，始终等于用于实例化数组模板类的第二个模板参数。

```
constexpr size_type max_size() noexcept;
```

```
#include <iostream>
#include <array>
int main ()
{
    std::array<int,10> myints;
    std::cout << "size of myints: " << myints.size() << '\n';
    std::cout << "max_size of myints: " << myints.max_size() << '\n';

    return 0;
}
```

Output

```
size of myints: 10
max_size of myints: 10
```

array::empty

返回一个布尔值，指示数组容器是否为空，即它的 size()是否为 0。

```
constexpr bool empty() noexcept;
```

```
#include <iostream>
#include <array>
int main ()
{
  std::array<int,0> first;
  std::array<int,5> second;
  std::cout << "first " << (first.empty() ? "is empty" : "is not empty") <<
'\n';
  std::cout << "second " << (second.empty() ? "is empty" : "is not empty") <<
'\n';
  return 0;
}
```

Output:

```
first is empty
second is not empt
```

array::operator[]

返回数组中第 n 个位置的元素的引用。与 array::at 相似，但 array::at 会检查数组边界并通过抛出一个 out*of*range 异常来判断 n 是否超出范围，而 array::operator[]不检查边界。

```
      reference operator[] (size_type n);
const_reference operator[] (size_type n) const;
```

Example

```
#include <iostream>
#include <array>

int main ()
{
    std::array<int,10> myarray;
```

```
    unsigned int i;
    // assign some values:
    for(i=0; i<10; i++)
        myarray[i] = i;
    // print content
    std::cout << "myarray contains:";
    for(i=0; i<10; i++)
        std::cout << ' ' << myarray[i];
    std::cout << '\n';

    return 0;
}
```

Output

```
myarray contains: 0 1 2 3 4 5 6 7 8 9
```

array::at

返回数组中第 n 个位置的元素的引用。与 array::operator[]相似，但 array::at 会检查数组边界并通过抛出一个 out*of*range 异常来判断 n 是否超出范围，而 array::operator[]不检查边界。

```
      reference at ( size_type n );
const_reference at ( size_type n ) const;
#include <iostream>
#include <array>
int main()
{
      std::array<int, 10> myarray;
      unsigned int i;

      // assign some values:
      for (i = 0; i<10; i++)
            myarray[i] = i;

      // print content
      std::cout << "myarray contains:";
      for (i = 0; i<10; i++)
            std::cout << ' ' << myarray.at(i);
      std::cout << '\n';

      return 0;
}
```

Output

```
myarray contains: 0 1 2 3 4 5 6 7 8 9
```

array::front

返回对数组容器中第一个元素的引用。array::begin 返回的是迭代器，array::front 返回的是直接引用。

在空容器上调用此函数会导致未定义的行为。

```
      reference front();
const_reference front() const;
```

Example

```
#include <iostream>
#include <array>
int main ()
{
```

```
  std::array<int,3> myarray = {2, 16, 77};

  std::cout << "front is: " << myarray.front() << std::endl;   // 2
  std::cout << "back is: " << myarray.back() << std::endl;     // 77
  myarray.front() = 100;
  std::cout << "myarray now contains:";
  for ( int& x : myarray ) std::cout << ' ' << x;
  std::cout << '\n';

  return 0;
}
```

Output

```
front is: 2
back is: 77
myarray now contains: 100 16 77
```

array::back

返回对数组容器中最后一个元素的引用。array::end 返回的是迭代器，array::back 返回的是直接引用。

在空容器上调用此函数会导致未定义的行为。

```
      reference back();
const_reference back() const;
```

Example

```
#include <iostream>
#include <array>

int main ()
{
  std::array<int,3> myarray = {5, 19, 77};

  std::cout << "front is: " << myarray.front() << std::endl;   // 5
  std::cout << "back is: " << myarray.back() << std::endl;     // 77
  myarray.back() = 50;
  std::cout << "myarray now contains:";
  for ( int& x : myarray ) std::cout << ' ' << x;
  std::cout << '\n';

  return 0;
}
```

Output

```
front is: 5
back is: 77
myarray now contains: 5 19 50
```

array::data

返回指向数组对象中第一个元素的指针。

由于数组中的元素存储在连续的存储位置，所以检索到的指针可以偏移以访问数组中的任何元素。

```
      value_type* data() noexcept;
const value_type* data() const noexcept;
```

Example

```
#include <iostream>
#include <cstring>
#include <array>

int main ()
{
  const char* cstr = "Test string";
  std::array<char,12> charray;
  std::memcpy (charray.data(),cstr,12);
  std::cout << charray.data() << '\n';

  return 0;
}
```

Output

```
Test string
```

array::fill

用 val 填充数组所有元素，将 val 设置为数组对象中所有元素的值。

```
void fill (const value_type& val);
```

Example

```
#include <iostream>
#include <array>

int main () {
  std::array<int,6> myarray;

  myarray.fill(5);

  std::cout << "myarray contains:";
  for ( int& x : myarray) { std::cout << ' ' << x; }

  std::cout << '\n';

  return 0;
}
```

Output

```
myarray contains: 5 5 5 5 5 5
```

array::swap

通过 x 的内容交换数组的内容，这是另一个相同类型的数组对象（包括相同的大小）。

与其他容器的交换成员函数不同，此成员函数通过在各个元素之间执行与其大小相同的单独交换操作，以线性时间运行。

```
void swap (array& x)
noexcept(noexcept(swap(declval<value_type&>(),declval<value_type&>())));
```

Example

```
#include <iostream>
#include <array>

int main ()
{
  std::array<int,5> first = {10, 20, 30, 40, 50};
  std::array<int,5> second = {11, 22, 33, 44, 55};
```

```
  first.swap (second);

  std::cout << "first:";
  for (int& x : first) std::cout << ' ' << x;
  std::cout << '\n';

  std::cout << "second:";
  for (int& x : second) std::cout << ' ' << x;
  std::cout << '\n';

  return 0;
}
```

Output

```
first: 11 22 33 44 55
second: 10 20 30 40 50
```

get（array）

形如：std::get<0>(myarray)；传入一个数组容器，返回指定位置元素的引用。

```
template <size_t I, class T, size_t N> T&get（array <T, N>&arr）noexcept;
template <size_t I, class T, size_t N> T && get（array <T, N> && arr）noexcept;
template <size_t I, class T, size_t N> const T&get（const array <T, N>&arr）
noexcept;
```

Example

```
#include <iostream>
#include <array>
#include <tuple>

int main ()
{
  std::array<int,3> myarray = {10, 20, 30};
  std::tuple<int,int,int> mytuple (10, 20, 30);

  std::tuple_element<0,decltype(myarray)>::type myelement;  // int myelement

  myelement = std::get<2>(myarray);
  std::get<2>(myarray) = std::get<0>(myarray);
  std::get<0>(myarray) = myelement;

  std::cout << "first element in myarray: " << std::get<0>(myarray) << "\n";
  std::cout << "first element in mytuple: " << std::get<0>(mytuple) << "\n";

  return 0;
}
```

Output

```
first element in myarray: 30
first element in mytuple: 10
```

relational operators (array)

形如：arrayA != arrayB、arrayA > arrayB；依此比较数组每个元素的大小关系。

（1）

```
template <class T, size_T N>
```

```
  bool operator == (const array <T, N>&lhs, const array <T, N>&rhs);
(2)
template <class T, size_T N>
  bool operator != (const array <T, N>&lhs, const array <T, N>&rhs);
(3)
template <class T, size_T N>
  bool operator < (const array <T, N>&lhs, const array <T, N>&rhs);
(4)
template <class T, size_T N>
  bool operator <= (const array <T, N>&lhs, const array <T, N>&rhs);
(5)
template <class T, size_T N>
  bool operator> (const array <T, N>&lhs, const array <T, N>&rhs);
(6)
template <class T, size_T N>
  bool operator>= (const array <T, N>&lhs, const array <T, N>&rhs);
```

Example

```
#include <iostream>
#include <array>

int main ()
{
  std::array<int,5> a = {10, 20, 30, 40, 50};
  std::array<int,5> b = {10, 20, 30, 40, 50};
  std::array<int,5> c = {50, 40, 30, 20, 10};

  if (a==b) std::cout << "a and b are equal\n";
  if (b!=c) std::cout << "b and c are not equal\n";
  if (b<c) std::cout << "b is less than c\n";
  if (c>b) std::cout << "c is greater than b\n";
  if (a<=b) std::cout << "a is less than or equal to b\n";
  if (a>=b) std::cout << "a is greater than or equal to b\n";

  return 0;
}
```

Output

```
a and b are equal
b and c are not equal
b is less than c
c is greater than b
a is less than or equal to b
a is greater than or equal to b
```

vector

vector 是表示可以改变大小的数组的序列容器。

就像数组一样，vector 为它们的元素使用连续的存储位置，这意味着它们的元素也可以使用到其元素的常规指针上的偏移来访问，而且和数组一样高效。但是与数组不同的是，它们的大小可以动态地改变，它们的存储由容器自动处理。

在内部，vector 使用一个动态分配的数组来存储它们的元素。这个数组可能需要重新分配，以便在插入新元素时增加大小，这意味着分配一个新数组并将所有元素移动到其中。就处理时间而言，这是一个相对昂贵的任务，因此每次将元素添加到容器时矢量都不会重新分配。
相反，vector 容器可以分配一些额外的存储以适应可能的增长，并且因此容器可以具有比严格需要包含其元素（即，其大小）的存储更大的实际容量。库可以实现不同的策略的增长到内存使用和重新分配之间的平衡，但在任何情况下，再分配应仅在对数生长的间隔发生尺寸，使得在所述载体的末端各个元件的插入可以与提供分期常量时间复杂性。
因此，与数组相比，载体消耗更多的内存来交换管理存储和以有效方式动态增长的能力。
与其他动态序列容器（deques，lists 和 forward_lists ）相比，vector 非常有效地访问其元素（就像数组一样），并相对有效地从元素末尾添加或移除元素。对于涉及插入或移除除了结尾之外的位置的元素的操作，它们执行比其他位置更差的操作，并且具有比列表和 forward_lists 更不一致的迭代器和引用。
针对 vector 的各种常见操作的复杂度（效率）如下：

- 随机访问 - 常数 O(1)
- 在尾部增删元素 - 平摊（amortized）常数 O(1)}}
- 增删元素 - 至 vector 尾部的线性距离 O(n)}}

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

vector::vector

（1）empty 容器构造函数（默认构造函数） 构造一个空的容器，没有元素。 （2）fill 构造函数 用 n 个元素构造一个容器。每个元素都是 val 的副本（如果提供）。 （3）范围（range）构造器 使用与[range，first，last]范围内的元素相同的顺序构造一个容器，其中的每个元素都是 emplace -从该范围内相应的元素构造而成。 （4）复制（copy）构造函数（并用分配器复制） 按照相同的顺序构造一个包含 x 中每个元素的副本的容器。 （5）移动（move）构造函数（和分配器移动） 构造一个获取 x 元素的容器。 如果指定了 alloc 并且与 x 的分配器不同，那么元素将被移动。否则，没有构建元素（他们的所有权直接转移）。 x 保持未指定但有效的状态。 （6）初始化列表构造函数 构造一个容器中的每个元件中的一个拷贝的 IL，以相同的顺序。

```
default (1)
explicit vector (const allocator_type& alloc = allocator_type());
fill (2)
explicit vector (size_type n);
         vector (size_type n, const value_type& val,
                 const allocator_type& alloc = allocator_type());
range (3)
template <class InputIterator>
  vector (InputIterator first, InputIterator last,
          const allocator_type& alloc = allocator_type());
copy (4)
vector (const vector& x);
vector (const vector& x, const allocator_type& alloc);
move (5)
vector (vector&& x);
vector (vector&& x, const allocator_type& alloc);
initializer list (6)
vector (initializer_list<value_type> il,
        const allocator_type& alloc = allocator_type());
```

Example

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

int main ()
{
    // constructors used in the same order as described above:
    std::vector<int> first;                // empty vector of ints
    std::vector<int> second(4, 100);       // four ints with value 100
    std::vector<int> third(second.begin(), second.end());// iterating through
second
    std::vector<int> fourth(third);       // a copy of third

    // the iterator constructor can also be used to construct from arrays:
    int myints[] = {16,2,77,29};
    std::vector<int> fifth(myints, myints + sizeof(myints) / sizeof(int));

    std::cout << "The contents of fifth are:";
    for(std::vector<int>::iterator it = fifth.begin(); it != fifth.end(); ++it)
        std::cout << ' ' << *it;
    std::cout << '\n';

    return 0;
}
```

Output

```
The contents of fifth are: 16 2 77 29
```

vector::~vector

销毁容器对象。这将在每个包含的元素上调用 allocator_traits::destroy，并使用其分配器释放由矢量分配的所有存储容量。

```
~vector();
```

vector::operator=

将新内容分配给容器，替换其当前内容，并相应地修改其大小。

```
copy (1)
vector& operator= (const vector& x);
move (2)
vector& operator= (vector&& x);
initializer list (3)
vector& operator= (initializer_list<value_type> il);
```

Example

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

int main ()
{
  std::vector<int> foo (3,0);
  std::vector<int> bar (5,0);

  bar = foo;
  foo = std::vector<int>();

  std::cout << "Size of foo: " << int(foo.size()) << '\n';
  std::cout << "Size of bar: " << int(bar.size()) << '\n';
```

```
  return 0;
}
```

Output

```
Size of foo: 0
Size of bar: 3
```

vector::begin
vector::end
vector::rbegin
vector::rend
vector::cbegin
vector::cend
vector::rcbegin
vector::rcend
vector::size

返回 vector 中元素的数量。

这是 vector 中保存的实际对象的数量，不一定等于其存储容量。

```
size_type size() const noexcept;
```

Example

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

int main ()
{
  std::vector<int> myints;
  std::cout << "0. size: " << myints.size() << '\n';

  for (int i=0; i<10; i++) myints.push_back(i);
  std::cout << "1. size: " << myints.size() << '\n';

  myints.insert (myints.end(),10,100);
  std::cout << "2. size: " << myints.size() << '\n';

  myints.pop_back();
  std::cout << "3. size: " << myints.size() << '\n';

  return 0;
}
```

Output

```
0. size: 0
1. size: 10
2. size: 20
3. size: 19
```

vector::max_size

返回该 vector 可容纳的元素的最大数量。由于已知的系统或库实现限制，

这是容器可以达到的最大潜在大小，但容器无法保证能够达到该大小：在达到该大小之前的任何时间，仍然无法分配存储。

```
size_type max_size() const noexcept;
```

Example

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

```
int main ()
{
  std::vector<int> myvector;

  // set some content in the vector:
  for (int i=0; i<100; i++) myvector.push_back(i);

  std::cout << "size: " << myvector.size() << "\n";
  std::cout << "capacity: " << myvector.capacity() << "\n";
  std::cout << "max_size: " << myvector.max_size() << "\n";
  return 0;
}
```

A possible output for this program could be:

```
size: 100
capacity: 128
max_size: 1073741823
```

vector::resize

调整容器的大小，使其包含 n 个元素。

如果 n 小于当前的容器 size，内容将被缩小到前 n 个元素，将其删除（并销毁它们）。
如果 n 大于当前容器 size，则通过在末尾插入尽可能多的元素以达到大小 n 来扩展内容。如果指定了 val，则新元素将初始化为 val 的副本，否则将进行值初始化。
如果 n 也大于当前的容器的 capacity（容量），分配的存储空间将自动重新分配。
注意这个函数通过插入或者删除元素的内容来改变容器的实际内容。

```
void resize (size_type n);
void resize (size_type n, const value_type& val);
```

Example

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

int main ()
{
  std::vector<int> myvector;

  // set some initial content:
  for (int i=1;i<10;i++) myvector.push_back(i);

  myvector.resize(5);
  myvector.resize(8,100);
  myvector.resize(12);

  std::cout << "myvector contains:";
  for (int i=0;i<myvector.size();i++)
    std::cout << ' ' << myvector[i];
  std::cout << '\n';

  return 0;
}
```

Output

```
myvector contains: 1 2 3 4 5 100 100 100 0 0 0 0
```

vector::capacity

返回当前为 vector 分配的存储空间的大小，用元素表示。这个 capacity(容量)不一定等于 vector 的 size。它可以相等或更大，额外的空间允许适应增长，而不需要重新分配每个插入。

```
size_type capacity() const noexcept;
```

Example

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

int main ()
{
  std::vector<int> myvector;

  // set some content in the vector:
  for (int i=0; i<100; i++) myvector.push_back(i);

  std::cout << "size: " << (int) myvector.size() << '\n';
  std::cout << "capacity: " << (int) myvector.capacity() << '\n';
  std::cout << "max_size: " << (int) myvector.max_size() << '\n';
  return 0;
}
```

A possible output for this program could be:

```
size: 100
capacity: 128
max_size: 1073741823
```

vector::empty

返回 vector 是否为空（即，它的 size 是否为 0）

```
bool empty() const noexcept;
```

Example

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

int main ()
{
  std::vector<int> myvector;
  int sum (0);

  for (int i=1;i<=10;i++) myvector.push_back(i);

  while (!myvector.empty())
  {
     sum += myvector.back();
     myvector.pop_back();
  }

  std::cout << "total: " << sum << '\n';

  return 0;
}
```

Output

```
total: 55
```

vector::reserve

请求 vector 容量至少足以包含 n 个元素。

如果 n 大于当前 vector 容量，则该函数使容器重新分配其存储容量，从而将其容量增加到 n（或更大）。

在所有其他情况下，函数调用不会导致重新分配，并且 vector 容量不受影响。

这个函数对 vector 大小没有影响，也不能改变它的元素。

```
void reserve (size_type n);
```

Example

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

int main ()
{
  std::vector<int>::size_type sz;

  std::vector<int> foo;
  sz = foo.capacity();
  std::cout << "making foo grow:\n";
  for (int i=0; i<100; ++i) {
    foo.push_back(i);
    if (sz!=foo.capacity()) {
      sz = foo.capacity();
      std::cout << "capacity changed: " << sz << '\n';
    }
  }

  std::vector<int> bar;
  sz = bar.capacity();
  bar.reserve(100);   // this is the only difference with foo above
  std::cout << "making bar grow:\n";
  for (int i=0; i<100; ++i) {
    bar.push_back(i);
    if (sz!=bar.capacity()) {
      sz = bar.capacity();
      std::cout << "capacity changed: " << sz << '\n';
    }
  }
  return 0;
}
```

Possible output

```
making foo grow:
capacity changed: 1
capacity changed: 2
capacity changed: 4
capacity changed: 8
capacity changed: 16
capacity changed: 32
capacity changed: 64
capacity changed: 128
making bar grow:
capacity changed: 100
```

vector::shrink*to*fit

要求容器减小其 capacity(容量)以适应其尺寸。

该请求是非绑定的，并且容器实现可以自由地进行优化，并且保持 capacity 大于其 size 的 vector。这可能导致重新分配，但对矢量大小没有影响，并且不能改变其元素。

```
void shrink_to_fit();
```

Example

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

int main ()
{
  std::vector<int> myvector (100);
  std::cout << "1. capacity of myvector: " << myvector.capacity() << '\n';

  myvector.resize(10);
  std::cout << "2. capacity of myvector: " << myvector.capacity() << '\n';

  myvector.shrink_to_fit();
  std::cout << "3. capacity of myvector: " << myvector.capacity() << '\n';

  return 0;
}
```

Possible output

```
1. capacity of myvector: 100
2. capacity of myvector: 100
3. capacity of myvector: 10
```

vector::operator[]

vector::at

vector::front

vector::back

vector::data

vector::assign

将新内容分配给 vector，替换其当前内容，并相应地修改其大小。

在范围版本（1）中，新内容是从第一个和最后一个范围内的每个元素按相同顺序构造的元素。

在填充版本（2）中，新内容是 n 个元素，每个元素都被初始化为一个 val 的副本。

在初始化列表版本（3）中，新内容是以相同顺序作为初始化列表传递的值的副本。

所述内部分配器被用于（通过其性状），以分配和解除分配存储器如果重新分配发生。它也习惯于摧毁所有现有的元素，并构建新的元素。

```
range (1)
template <class InputIterator>
  void assign (InputIterator first, InputIterator last);
fill (2)
void assign (size_type n, const value_type& val);
initializer list (3)
void assign (initializer_list<value_type> il);
```

Example

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

int main ()
```

```
{
  std::vector<int> first;
  std::vector<int> second;
  std::vector<int> third;

  first.assign (7,100);             // 7 ints with a value of 100

  std::vector<int>::iterator it;
  it=first.begin()+1;

  second.assign (it,first.end()-1); // the 5 central values of first

  int myints[] = {1776,7,4};
  third.assign (myints,myints+3);   // assigning from array.

  std::cout << "Size of first: " << int (first.size()) << '\n';
  std::cout << "Size of second: " << int (second.size()) << '\n';
  std::cout << "Size of third: " << int (third.size()) << '\n';
  return 0;
}
```

Output

```
Size of first: 7
Size of second: 5
Size of third: 3
```

补充：vector::assign 与 vector::operator= 的区别：

vector::assign 实现源码

```
void assign(size_type __n, const _Tp& __val) { _M_fill_assign(__n, __val); }

template <class _Tp, class _Alloc>
void vector<_Tp, _Alloc>::_M_fill_assign(size_t __n, const value_type& __val)
{
  if (__n > capacity()) {
    vector<_Tp, _Alloc> __tmp(__n, __val, get_allocator());
    __tmp.swap(*this);
  }
  else if (__n > size()) {
    fill(begin(), end(), __val);
    _M_finish = uninitialized_fill_n(_M_finish, __n - size(), __val);
  }
  else
    erase(fill_n(begin(), __n, __val), end());
}
```

vector::operator= 实现源码

```
template <class _Tp, class _Alloc>
vector<_Tp,_Alloc>&
vector<_Tp,_Alloc>::operator=(const vector<_Tp, _Alloc>& __x)
{
  if (&__x != this) {
    const size_type __xlen = __x.size();
    if (__xlen > capacity()) {
      iterator __tmp = _M_allocate_and_copy(__xlen, __x.begin(), __x.end());
      destroy(_M_start, _M_finish);
```

```
      _M_deallocate(_M_start, _M_end_of_storage - _M_start);
      _M_start = __tmp;
      _M_end_of_storage = _M_start + __xlen;
    }
    else if (size() >= __xlen) {
      iterator __i = copy(__x.begin(), __x.end(), begin());
      destroy(__i, _M_finish);
    }
    else {
      copy(__x.begin(), __x.begin() + size(), _M_start);
      uninitialized_copy(__x.begin() + size(), __x.end(), _M_finish);
    }
    _M_finish = _M_start + __xlen;
  }
  return *this;
}
```

vector::push_back

在 vector 的最后一个元素之后添加一个新元素。val 的内容被复制（或移动）到新的元素。

这有效地将容器 size 增加了一个，如果新的矢量 size 超过了当前 vector 的 capacity，则导致所分配的存储空间自动重新分配。

```
void push_back (const value_type& val);
void push_back (value_type&& val);
```

Example

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

int main ()
{
  std::vector<int> myvector;
  int myint;

  std::cout << "Please enter some integers (enter 0 to end):\n";

  do {
    std::cin >> myint;
    myvector.push_back (myint);
  } while (myint);

  std::cout << "myvector stores " << int(myvector.size()) << " numbers.\n";

  return 0;
}
```

vector::pop_back

删除 vector 中的最后一个元素，有效地将容器 size 减少一个。

这破坏了被删除的元素。

```
void pop_back();
```

Example

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

int main ()
```

```
{
  std::vector<int> myvector;
  int sum (0);
  myvector.push_back (100);
  myvector.push_back (200);
  myvector.push_back (300);

  while (!myvector.empty())
  {
    sum+=myvector.back();
    myvector.pop_back();
  }

  std::cout << "The elements of myvector add up to " << sum << '\n';

  return 0;
}
```

Output

```
The elements of myvector add up to 600
```

vector::insert

通过在指定位置的元素之前插入新元素来扩展该 vector，通过插入元素的数量有效地增加容器大小。 这会导致分配的存储空间自动重新分配，只有在新的 vector 的 size 超过当前的 vector 的 capacity 的情况下。

由于 vector 使用数组作为其基础存储，因此除了将元素插入到 vector 末尾之后，或 vector 的 begin 之前，其他位置会导致容器重新定位位置之后的所有元素到他们的新位置。与其他种类的序列容器（例如 list 或 forward_list）执行相同操作的操作相比，这通常是低效的操作。

```
single element (1)
iterator insert (const_iterator position, const value_type& val);
fill (2)
iterator insert (const_iterator position, size_type n, const value_type& val);
range (3)
template <class InputIterator>
iterator insert (const_iterator position, InputIterator first, InputIterator
last);
move (4)
iterator insert (const_iterator position, value_type&& val);
initializer list (5)
iterator insert (const_iterator position, initializer_list<value_type> il);
```

Example

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

int main ()
{
  std::vector<int> myvector (3,100);
  std::vector<int>::iterator it;

  it = myvector.begin();
  it = myvector.insert ( it , 200 );

  myvector.insert (it,2,300);
```

```
  // "it" no longer valid, get a new one:
  it = myvector.begin();

  std::vector<int> anothervector (2,400);
  myvector.insert (it+2,anothervector.begin(),anothervector.end());

  int myarray [] = { 501,502,503 };
  myvector.insert (myvector.begin(), myarray, myarray+3);

  std::cout << "myvector contains:";
  for (it=myvector.begin(); it<myvector.end(); it++)
    std::cout << ' ' << *it;
  std::cout << '\n';

  return 0;
}
```

Output

```
myvector contains: 501 502 503 300 300 400 400 200 100 100 100
```

补充：insert 迭代器野指针错误：

```
int main()
{
	std::vector<int> v(5, 0);
	std::vector<int>::iterator vi;

  // 获取 vector 第一个元素的迭代器
	vi = v.begin();

  // push_back 插入元素之后可能会因为 push_back 的骚操作（创建一个新 vector 把旧 vector 的值复制到新 vector），导致 vector 迭代器 iterator 的指针变成野指针，而导致 insert 出错
	v.push_back(10);

	v.insert(vi, 2, 300);

	return 0;
}
```

改正：应该把 `vi = v.begin();` 放到 `v.push_back(10);` 后面

vector::erase

从 vector 中删除单个元素（position）或一系列元素（[first，last））。

这有效地减少了被去除的元素的数量，从而破坏了容器的大小。

由于 vector 使用一个数组作为其底层存储，所以删除除 vector 结束位置之后，或 vector 的 begin 之前的元素外，将导致容器将段被擦除后的所有元素重新定位到新的位置。与其他种类的序列容器（例如 list 或 forward_list）执行相同操作的操作相比，这通常是低效的操作。

```
iterator erase (const_iterator position);
iterator erase (const_iterator first, const_iterator last);
```

Example

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

```
int main ()
{
  std::vector<int> myvector;

  // set some values (from 1 to 10)
  for (int i=1; i<=10; i++) myvector.push_back(i);

  // erase the 6th element
  myvector.erase (myvector.begin()+5);

  // erase the first 3 elements:
  myvector.erase (myvector.begin(),myvector.begin()+3);

  std::cout << "myvector contains:";
  for (unsigned i=0; i<myvector.size(); ++i)
    std::cout << ' ' << myvector[i];
  std::cout << '\n';

  return 0;
}
```

Output

```
myvector contains: 4 5 7 8 9 10
```

vector::swap

通过 x 的内容交换容器的内容，x 是另一个相同类型的 vector 对象。尺寸可能不同。

在调用这个成员函数之后，这个容器中的元素是那些在调用之前在 x 中的元素，而 x 的元素是在这个元素中的元素。所有迭代器，引用和指针对交换对象保持有效。

请注意，非成员函数存在具有相同名称的交换，并使用与此成员函数相似的优化来重载该算法。

```
void swap (vector& x);
```

Example

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

int main ()
{
  std::vector<int> foo (3,100);   // three ints with a value of 100
  std::vector<int> bar (5,200);   // five ints with a value of 200

  foo.swap(bar);

  std::cout << "foo contains:";
  for (unsigned i=0; i<foo.size(); i++)
    std::cout << ' ' << foo[i];
  std::cout << '\n';

  std::cout << "bar contains:";
  for (unsigned i=0; i<bar.size(); i++)
    std::cout << ' ' << bar[i];
  std::cout << '\n';
```

```
  return 0;
}
```

Output

```
foo contains: 200 200 200 200 200
bar contains: 100 100 100
```

vector::clear

从 vector 中删除所有的元素（被销毁），留下 size 为 0 的容器。

不保证重新分配，并且由于调用此函数， vector 的 capacity 不保证发生变化。强制重新分配的典型替代方法是使用 swap：`vector<T>().swap(x);  // clear x reallocating`

```
void clear() noexcept;
```

Example

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

void printVector(const std::vector<int> &v)
{
	for (auto it = v.begin(); it != v.end(); ++it)
	{
		std::cout << *it << ' ';
	}
	std::cout << std::endl;
}

int main()
{
	std::vector<int> v1(5, 50);

	printVector(v1);
	std::cout << "v1 size  = " << v1.size() << std::endl;
	std::cout << "v1 capacity  = " << v1.capacity() << std::endl;

	v1.clear();

	printVector(v1);
	std::cout << "v1 size  = " << v1.size() << std::endl;
	std::cout << "v1 capacity  = " << v1.capacity() << std::endl;

	v1.push_back(11);
	v1.push_back(22);

	printVector(v1);
	std::cout << "v1 size  = " << v1.size() << std::endl;
	std::cout << "v1 capacity  = " << v1.capacity() << std::endl;

	return 0;
}
```

Output

```
50 50 50 50 50
v1 size  = 5
v1 capacity  = 5
```

```
v1 size  = 0
v1 capacity  = 5
11 22
v1 size  = 2
v1 capacity  = 5
```

vector::emplace

通过在 position 位置处插入新元素 args 来扩展容器。这个新元素是用 args 作为构建的参数来构建的。

这有效地增加了一个容器的大小。

分配存储空间的自动重新分配发生在新的 vector 的 size 超过当前向量容量的情况下。

由于 vector 使用数组作为其基础存储，因此除了将元素插入到 vector 末尾之后，或 vector 的 begin 之前，其他位置会导致容器重新定位位置之后的所有元素到他们的新位置。与其他种类的序列容器（例如 list 或 forward_list）执行相同操作的操作相比，这通常是低效的操作。

该元素是通过调用 allocator_traits::construct 来转换 args 来创建的。插入一个类似的成员函数，将现有对象复制或移动到容器中。

```
template <class... Args>
iterator emplace (const_iterator position, Args&&... args);
```

Example

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

int main ()
{
  std::vector<int> myvector = {10,20,30};

  auto it = myvector.emplace ( myvector.begin()+1, 100 );
  myvector.emplace ( it, 200 );
  myvector.emplace ( myvector.end(), 300 );

  std::cout << "myvector contains:";
  for (auto& x: myvector)
    std::cout << ' ' << x;
  std::cout << '\n';

  return 0;
}
```

Output

```
myvector contains: 10 200 100 20 30 300
```

vector::emplace_back

在 vector 的末尾插入一个新的元素，紧跟在当前的最后一个元素之后。这个新元素是用 args 作为构造函数的参数来构造的。

这有效地将容器大小增加了一个，如果新的矢量大小超过了当前的 vector 容量，则导致所分配的存储空间自动重新分配。

该元素是通过调用 allocator_traits :: construct 来转换 args 来创建的。

与 push_back 相比，emplace_back 可以避免额外的复制和移动操作。

```
template <class... Args>
  void emplace_back (Args&&... args);
```

Example

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

struct President
{
    std::string name;
    std::string country;
    int year;

    President(std::string p_name, std::string p_country, int p_year)
        : name(std::move(p_name)), country(std::move(p_country)), year(p_year)
    {
        std::cout << "I am being constructed.\n";
    }
    President(President&& other)
        : name(std::move(other.name)), country(std::move(other.country)),
year(other.year)
    {
        std::cout << "I am being moved.\n";
    }
    President& operator=(const President& other) = default;
};

int main()
{
    std::vector<President> elections;
    std::cout << "emplace_back:\n";
    elections.emplace_back("Nelson Mandela", "South Africa", 1994);

    std::vector<President> reElections;
    std::cout << "\npush_back:\n";
    reElections.push_back(President("Franklin Delano Roosevelt", "the USA",
1936));

    std::cout << "\nContents:\n";
    for (President const& president: elections) {
        std::cout << president.name << " was elected president of "
                  << president.country << " in " << president.year << ".\n";
    }
    for (President const& president: reElections) {
        std::cout << president.name << " was re-elected president of "
                  << president.country << " in " << president.year << ".\n";
    }
}
```

Output

```
emplace_back:
I am being constructed.

push_back:
I am being constructed.
I am being moved.
```

```
Contents:
Nelson Mandela was elected president of South Africa in 1994.
Franklin Delano Roosevelt was re-elected president of the USA in 1936.
```

vector::get_allocator

返回与 vector 关联的构造器对象的副本。

```
allocator_type get_allocator() const noexcept;
```

Example

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

int main ()
{
  std::vector<int> myvector;
  int * p;
  unsigned int i;

  // allocate an array with space for 5 elements using vector's allocator:
  p = myvector.get_allocator().allocate(5);

  // construct values in-place on the array:
  for (i=0; i<5; i++) myvector.get_allocator().construct(&p[i],i);

  std::cout << "The allocated array contains:";
  for (i=0; i<5; i++) std::cout << ' ' << p[i];
  std::cout << '\n';

  // destroy and deallocate:
  for (i=0; i<5; i++) myvector.get_allocator().destroy(&p[i]);
  myvector.get_allocator().deallocate(p,5);

  return 0;
}
```

Output

```
The allocated array contains: 0 1 2 3 4
```

注意：deallocate 和 destory 的关系：

deallocate 实现的源码：

```
template <class T>
inline void _deallocate(T* buffer)
{
      ::operator delete(buffer);     //为什么不用 delete [] ?  ,operator delete 区别于 delete
                                     //operator delete   是一个底层操作符
}
```

destory：

```
template <class T>
inline void _destory(T *ptr)
{
      ptr->~T();
}
```

destory 负责调用类型的析构函数，销毁相应内存上的内容（但销毁后内存地址仍保留）

deallocate 负责释放内存（此时相应内存中的值在此之前应调用 destory 销毁，将内存地址返回给系统，代表这部分地址使用引用-1）

relational operators (vector)
swap (vector)
vector
deque

deque（['dek]）（双端队列）是 double-ended queue 的一个不规则缩写。deque 是具有动态大小的序列容器，可以在两端（前端或后端）扩展或收缩。

特定的库可以以不同的方式实现 deques，通常作为某种形式的动态数组。但是在任何情况下，它们都允许通过随机访问迭代器直接访问各个元素，通过根据需要扩展和收缩容器来自动处理存储。

因此，它们提供了类似于 vector 的功能，但是在序列的开始部分也可以高效地插入和删除元素，而不仅仅是在结尾。但是，与 vector 不同，deques 并不保证将其所有元素存储在连续的存储位置：deque 通过偏移指向另一个元素的指针访问元素会导致未定义的行为。

两个 vector 和 deques 提供了一个非常相似的接口，可以用于类似的目的，但内部工作方式完全不同：虽然 vector 使用单个数组需要偶尔重新分配以增长，但是 deque 的元素可以分散在不同的块的容器，容器在内部保存必要的信息以提供对其任何元素的持续时间和统一的顺序接口（通过迭代器）的直接访问。因此，deques 在内部比 vector 更复杂一点，但是这使得他们在某些情况下更有效地增长，尤其是在重新分配变得更加昂贵的很长序列的情况下。

对于频繁插入或删除开始或结束位置以外的元素的操作，deques 表现得更差，并且与列表和转发列表相比，迭代器和引用的一致性更低。

deque 上常见操作的复杂性（效率）如下：

- 随机访问 - 常数 O(1)
- 在结尾或开头插入或移除元素 - 摊销不变 O(1)
- 插入或移除元素 - 线性 O(n)

```
template < class T, class Alloc = allocator<T> > class deque;
```

deque::deque

构造一个 deque 容器对象，根据所使用的构造函数版本初始化它的内容：

Example

```
#include <iostream>
#include <deque>

int main ()
{
  unsigned int i;

  // constructors used in the same order as described above:
  std::deque<int> first;                                // empty deque of ints
  std::deque<int> second (4,100);                       // four ints with value
100
  std::deque<int> third (second.begin(),second.end());  // iterating through
second
  std::deque<int> fourth (third);                       // a copy of third

  // the iterator constructor can be used to copy arrays:
  int myints[] = {16,2,77,29};
```

```
  std::deque<int> fifth (myints, myints + sizeof(myints) / sizeof(int) );

  std::cout << "The contents of fifth are:";
  for (std::deque<int>::iterator it = fifth.begin(); it!=fifth.end(); ++it)
    std::cout << ' ' << *it;

  std::cout << '\n';

  return 0;
}
```

Output

```
The contents of fifth are: 16 2 77 29
```

deque::push_back

在当前的最后一个元素之后，在 deque 容器的末尾添加一个新元素。val 的内容被复制（或移动）到新的元素。

这有效地增加了一个容器的大小。

```
void push_back (const value_type& val);
void push_back (value_type&& val);
```

Example

```
#include <iostream>
#include <deque>

int main ()
{
  std::deque<int> mydeque;
  int myint;

  std::cout << "Please enter some integers (enter 0 to end):\n";

  do {
    std::cin >> myint;
    mydeque.push_back (myint);
  } while (myint);

  std::cout << "mydeque stores " << (int) mydeque.size() << " numbers.\n";

  return 0;
}
```

deque::push_front

在 deque 容器的开始位置插入一个新的元素，位于当前的第一个元素之前。val 的内容被复制（或移动）到插入的元素。

这有效地增加了一个容器的大小。

```
void push_front (const value_type& val);
void push_front (value_type&& val);
```

Example

```
#include <iostream>
#include <deque>

int main ()
{
```

```
  std::deque<int> mydeque (2,100);     // two ints with a value of 100
  mydeque.push_front (200);
  mydeque.push_front (300);

  std::cout << "mydeque contains:";
  for (std::deque<int>::iterator it = mydeque.begin(); it != mydeque.end();
++it)
    std::cout << ' ' << *it;
  std::cout << '\n';

  return 0;
}
```

Output

```
300 200 100 100
```

deque::pop_back

删除 deque 容器中的最后一个元素，有效地将容器大小减少一个。

这破坏了被删除的元素。

```
void pop_back();
```

Example

```
#include <iostream>
#include <deque>

int main ()
{
  std::deque<int> mydeque;
  int sum (0);
  mydeque.push_back (10);
  mydeque.push_back (20);
  mydeque.push_back (30);

  while (!mydeque.empty())
  {
    sum+=mydeque.back();
    mydeque.pop_back();
  }

  std::cout << "The elements of mydeque add up to " << sum << '\n';

  return 0;
}
```

Output

```
The elements of mydeque add up to 60
```

deque::pop_front

删除 deque 容器中的第一个元素，有效地减小其大小。

这破坏了被删除的元素。

```
void pop_front();
```

Example

```
#include <iostream>
#include <deque>
```

```
int main ()
{
  std::deque<int> mydeque;

  mydeque.push_back (100);
  mydeque.push_back (200);
  mydeque.push_back (300);

  std::cout << "Popping out the elements in mydeque:";
  while (!mydeque.empty())
  {
    std::cout << ' ' << mydeque.front();
    mydeque.pop_front();
  }

  std::cout << "\nThe final size of mydeque is " << int(mydeque.size()) << '\n';

  return 0;
}
```

Output

```
Popping out the elements in mydeque: 100 200 300
The final size of mydeque is 0
```

deque::emplace_front

在 deque 的开头插入一个新的元素，就在其当前的第一个元素之前。这个新的元素是用 args 作为构建的参数来构建的。

这有效地增加了一个容器的大小。

该元素是通过调用 allocator_traits::construct 来转换 args 来创建的。

存在一个类似的成员函数 push_front，它可以将现有对象复制或移动到容器中。

```
template <class... Args>
  void emplace_front (Args&&... args);
```

Example

```
#include <iostream>
#include <deque>

int main ()
{
  std::deque<int> mydeque = {10,20,30};

  mydeque.emplace_front (111);
  mydeque.emplace_front (222);

  std::cout << "mydeque contains:";
  for (auto& x: mydeque)
    std::cout << ' ' << x;
  std::cout << '\n';

  return 0;
}
```

Output

```
mydeque contains: 222 111 10 20 30
```

deque::emplace_back

在 deque 的末尾插入一个新的元素，紧跟在当前的最后一个元素之后。这个新的元素是用 args 作为构建的参数来构建的。

这有效地增加了一个容器的大小。

该元素是通过调用 allocator_traits::construct 来转换 args 来创建的。

存在一个类似的成员函数 push_back，它可以将现有对象复制或移动到容器中

```
template <class... Args>
  void emplace_back (Args&&... args);
```

Example

```
#include <iostream>
#include <deque>

int main ()
{
  std::deque<int> mydeque = {10,20,30};

  mydeque.emplace_back (100);
  mydeque.emplace_back (200);

  std::cout << "mydeque contains:";
  for (auto& x: mydeque)
    std::cout << ' ' << x;
  std::cout << '\n';

  return 0;
}
```

Output

```
mydeque contains: 10 20 30 100 200
```

forward_list

forward_list（单向链表）是序列容器，允许在序列中的任何地方进行恒定的时间插入和擦除操作。

forward_list（单向链表）被实现为单链表; 单链表可以将它们包含的每个元素存储在不同和不相关的存储位置中。通过关联到序列中下一个元素的链接的每个元素来保留排序。forward_list 容器和列表

之间的主要设计区别容器是第一个内部只保留一个到下一个元素的链接，而后者每个元素保留两个链接：一个指向下一个元素，一个指向前一个元素，允许在两个方向上有效的迭代，但是每个元素消耗额外的存储空间并且插入和移除元件的时间开销略高。因此，forward_list 对象比列表对象更有效率，尽管它们只能向前迭代。

与其他基本的标准序列容器（array，vector 和 deque），forward_list 通常在插入，提取和移动容器内任何位置的元素方面效果更好，因此也适用于密集使用这些元素的算法，如排序算法。

的主要缺点修饰符 Modifiers S 和列表相比这些其它序列容器 s 是说，他们缺乏可以通过位置的元素的直接访问; 例如，要访问 forward_list 中的第六个元素，必须从开始位置迭代到该位置，这需要在这些位置之间的线性时间。它们还消耗一些额外的内存来保持与每个元素相关联的链接信息（这可能是大型小元素列表的重要因素）。

该修饰符 Modifiersclass 模板的设计考虑到效率：按照设计，它与简单的手写 C 型单链表一样高效，实际上是唯一的标准容器，为了效率的考虑故意缺少尺寸成员函数：由于其性质作为一个链表，具有一个需要一定时间的大小的成员将需要它保持一个内部计数器的大小（如列表所示

）。这会消耗一些额外的存储空间，并使插入和删除操作效率稍低。要获取 forward_list 对象的大小，可以使用距离算法的开始和结束，这是一个需要线性时间的操作。

forward_list::forward_list

```
default (1)
explicit forward_list (const allocator_type& alloc = allocator_type());
fill (2)
explicit forward_list (size_type n);
explicit forward_list (size_type n, const value_type& val,
                       const allocator_type& alloc = allocator_type());
range (3)
template <class InputIterator>
  forward_list (InputIterator first, InputIterator last,
                const allocator_type& alloc = allocator_type());
copy (4)
forward_list (const forward_list& fwdlst);
forward_list (const forward_list& fwdlst, const allocator_type& alloc);
move (5)
forward_list (forward_list&& fwdlst);
forward_list (forward_list&& fwdlst, const allocator_type& alloc);
initializer list (6)
forward_list (initializer_list<value_type> il,
              const allocator_type& alloc = allocator_type());
```

Example

```
#include <iostream>
#include <forward_list>

int main ()
{
  // constructors used in the same order as described above:

  std::forward_list<int> first;                          // default: empty
  std::forward_list<int> second (3,77);                  // fill: 3 seventy-sevens
  std::forward_list<int> third (second.begin(), second.end()); // range
initialization
  std::forward_list<int> fourth (third);               // copy constructor
  std::forward_list<int> fifth (std::move(fourth));  // move ctor. (fourth
wasted)
  std::forward_list<int> sixth = {3, 52, 25, 90};    // initializer_list
constructor

  std::cout << "first:" ; for (int& x: first)  std::cout << " " << x; std::cout
<< '\n';
  std::cout << "second:"; for (int& x: second) std::cout << " " << x; std::cout
<< '\n';
  std::cout << "third:";  for (int& x: third)  std::cout << " " << x; std::cout
<< '\n';
  std::cout << "fourth:"; for (int& x: fourth) std::cout << " " << x; std::cout
<< '\n';
  std::cout << "fifth:";  for (int& x: fifth)  std::cout << " " << x; std::cout
<< '\n';
```

```
  std::cout << "sixth:";  for (int& x: sixth)  std::cout << " " << x; std::cout
<< '\n';

  return 0;
}
```

Possible output

```
forward_list constructor examples:
first:
second: 77 77 77
third: 77 77 77
fourth:
fifth: 77 77 77
sixth: 3 52 25 90
```

forward_list::~forward_list

forward_list::before_begin

返回指向容器中第一个元素之前的位置的迭代器。

返回的迭代器不应被解除引用：它是为了用作成员函数的参数 emplace_after，insert_after，erase_after 或 splice_after，指定序列，其中执行该动作的位置的开始位置。

```
      iterator before_begin() noexcept;
const_iterator before_begin() const noexcept;
```

Example

```
#include <iostream>
#include <forward_list>

int main ()
{
  std::forward_list<int> mylist = {20, 30, 40, 50};

  mylist.insert_after ( mylist.before_begin(), 11 );

  std::cout << "mylist contains:";
  for ( int& x: mylist ) std::cout << ' ' << x;
  std::cout << '\n';

  return 0;
}
```

Output

```
mylist contains: 11 20 30 40 50
```

forward_list::cbefore_begin

返回指向容器中第一个元素之前的位置的 const_iterator。

一个常量性是指向常量内容的迭代器。这个迭代器可以增加和减少（除非它本身也是 const），就像 forward_list::before_begin 返回的迭代器一样，但不能用来修改它指向的内容。

返回的价值不得解除引用。

```
const_iterator cbefore_begin() const noexcept;
```

Example

```
#include <iostream>
#include <forward_list>

int main ()
{
```

```
  std::forward_list<int> mylist = {77, 2, 16};

  mylist.insert_after ( mylist.cbefore_begin(), 19 );

  std::cout << "mylist contains:";
  for ( int& x: mylist ) std::cout << ' ' << x;
  std::cout << '\n';

  return 0;
}
```

Output

```
mylist contains: 19 77 2 16
```

list

stack

queue

priority_queue

set

multiset

map

map 是关联容器，按照特定顺序存储由 key value (键值) 和 mapped value (映射值) 组合形成的元素。

在映射中，键值通常用于对元素进行排序和唯一标识，而映射的值存储与此键关联的内容。该类型的键和映射的值可能不同，并且在部件类型被分组在一起 VALUE_TYPE，这是一种对类型结合两种：

```
typedef pair<const Key, T> value_type;
```

在内部，映射中的元素总是按照由其内部比较对象（比较类型）指示的特定的严格弱排序标准按键排序。映射容器通常比 unordered_map 容器慢，以通过它们的键来访问各个元素，但是它们允许基于它们的顺序对子集进行直接迭代。 在该映射值地图可以直接通过使用其相应的键来访问括号运算符（（操作符[] ））。 映射通常如实施

```
template < class Key,                                     // map::key_type
           class T,                                       // map::mapped_type
           class Compare = less<Key>,                     // map::key_compare
           class Alloc = allocator<pair<const Key,T> >    // map::allocator_type
           > class map;
```

map::map

构造一个映射容器对象，根据所使用的构造器版本初始化其内容：

（1）空容器构造函数（默认构造函数）

构造一个空的容器，没有元素。

（2）范围构造函数

构造具有一样多的元素的范围内的容器[第一，最后一个），其中每个元件布设构造的从在该范围内它的相应的元件。

（3）复制构造函数（并用分配器复制）

使用 x 中的每个元素的副本构造一个容器。

（4）移动构造函数（并与分配器一起移动）

构造一个获取 x 元素的容器。 如果指定了 alloc 并且与 x 的分配器不同，那么元素将被移动。否则，没有构建元素（他们的所有权直接转移）。 x 保持未指定但有效的状态。

（5）初始化列表构造函数

用 il 中的每个元素的副本构造一个容器。

```
empty (1)
explicit map (const key_compare& comp = key_compare(),
              const allocator_type& alloc = allocator_type());
explicit map (const allocator_type& alloc);
range (2)
template <class InputIterator>
  map (InputIterator first, InputIterator last,
       const key_compare& comp = key_compare(),
       const allocator_type& = allocator_type());
copy (3)
map (const map& x);
map (const map& x, const allocator_type& alloc);
move (4)
map (map&& x);
map (map&& x, const allocator_type& alloc);
initializer list (5)
map (initializer_list<value_type> il,
     const key_compare& comp = key_compare(),
     const allocator_type& alloc = allocator_type());
```

Example

```
#include <iostream>
#include <map>

bool fncomp (char lhs, char rhs) {return lhs<rhs;}

struct classcomp {
  bool operator() (const char& lhs, const char& rhs) const
  {return lhs<rhs;}
};

int main ()
{
  std::map<char,int> first;

  first['a']=10;
  first['b']=30;
  first['c']=50;
  first['d']=70;

  std::map<char,int> second (first.begin(),first.end());

  std::map<char,int> third (second);

  std::map<char,int,classcomp> fourth;                 // class as Compare

  bool(*fn_pt)(char,char) = fncomp;
  std::map<char,int,bool(*)(char,char)> fifth (fn_pt); // function pointer as
Compare

  return 0;
}
```

map::begin

返回引用 map 容器中第一个元素的迭代器。

由于 map 容器始终保持其元素的顺序，所以开始指向遵循容器排序标准的元素。

如果容器是空的，则返回的迭代器值不应被解除引用。

```
      iterator begin() noexcept;
const_iterator begin() const noexcept;
```

Example

```
#include <iostream>
#include <map>

int main ()
{
  std::map<char,int> mymap;

  mymap['b'] = 100;
  mymap['a'] = 200;
  mymap['c'] = 300;

  // show content:
  for (std::map<char,int>::iterator it=mymap.begin(); it!=mymap.end(); ++it)
    std::cout << it->first << " => " << it->second << '\n';

  return 0;
}
```

Output

```
a => 200
b => 100
c => 300
```

map::key_comp

返回容器用于比较键的比较对象的副本。

```
key_compare key_comp() const;
```

Example

```
#include <iostream>
#include <map>

int main ()
{
  std::map<char,int> mymap;

  std::map<char,int>::key_compare mycomp = mymap.key_comp();

  mymap['a']=100;
  mymap['b']=200;
  mymap['c']=300;

  std::cout << "mymap contains:\n";

  char highest = mymap.rbegin()->first;     // key value of last element

  std::map<char,int>::iterator it = mymap.begin();
  do {
```

```
    std::cout << it->first << " => " << it->second << '\n';
  } while ( mycomp((*it++).first, highest) );

  std::cout << '\n';

  return 0;
}
```

Output

```
mymap contains:
a => 100
b => 200
c => 300
```

map::value_comp

返回可用于比较两个元素的比较对象，以获取第一个元素的键是否在第二个元素之前。

```
value_compare value_comp() const;
```

Example

```
#include <iostream>
#include <map>

int main ()
{
  std::map<char,int> mymap;

  mymap['x']=1001;
  mymap['y']=2002;
  mymap['z']=3003;

  std::cout << "mymap contains:\n";

  std::pair<char,int> highest = *mymap.rbegin();          // last element

  std::map<char,int>::iterator it = mymap.begin();
  do {
    std::cout << it->first << " => " << it->second << '\n';
  } while ( mymap.value_comp()(*it++, highest) );

  return 0;
}
```

Output

```
mymap contains:
x => 1001
y => 2002
z => 3003
```

map::find

在容器中搜索具有等于 k 的键的元素，如果找到则返回一个迭代器，否则返回 map::end 的迭代器。

如果容器的比较对象自反地返回假（即，不管元素作为参数传递的顺序），则两个 key 被认为是等同的。

另一个成员函数 map::count 可以用来检查一个特定的键是否存在。

```
      iterator find (const key_type& k);
const_iterator find (const key_type& k) const;
```

Example

```
#include <iostream>
#include <map>

int main ()
{
  std::map<char,int> mymap;
  std::map<char,int>::iterator it;

  mymap['a']=50;
  mymap['b']=100;
  mymap['c']=150;
  mymap['d']=200;

  it = mymap.find('b');
  if (it != mymap.end())
    mymap.erase (it);

  // print content:
  std::cout << "elements in mymap:" << '\n';
  std::cout << "a => " << mymap.find('a')->second << '\n';
  std::cout << "c => " << mymap.find('c')->second << '\n';
  std::cout << "d => " << mymap.find('d')->second << '\n';

  return 0;
}
```

Output

```
elements in mymap:
a => 50
c => 150
d => 200
```

map::count

在容器中搜索具有等于 k 的键的元素，并返回匹配的数量。

由于地图容器中的所有元素都是唯一的，因此该函数只能返回 1（如果找到该元素）或返回零（否则）。

如果容器的比较对象自反地返回错误（即，不管按键作为参数传递的顺序），则两个键被认为是等同的。

```
size_type count (const key_type& k) const;
```

Example

```
#include <iostream>
#include <map>

int main ()
{
  std::map<char,int> mymap;
  char c;

  mymap ['a']=101;
  mymap ['c']=202;
```

```
  mymap ['f']=303;

  for (c='a'; c<'h'; c++)
  {
    std::cout << c;
    if (mymap.count(c)>0)
      std::cout << " is an element of mymap.\n";
    else
      std::cout << " is not an element of mymap.\n";
  }

  return 0;
}
```

Output

```
a is an element of mymap.
b is not an element of mymap.
c is an element of mymap.
d is not an element of mymap.
e is not an element of mymap.
f is an element of mymap.
g is not an element of mymap.
```

map::lower_bound

将迭代器返回到下限

返回指向容器中第一个元素的迭代器，该元素的键不会在 k 之前出现（即，它是等价的或者在其后）。

该函数使用其内部比较对象（key_comp）来确定这一点，将迭代器返回到 key_comp（element_key，k）将返回 false 的第一个元素。

如果 map 类用默认的比较类型（less）实例化，则函数返回一个迭代器到第一个元素，其键不小于 k。

一个类似的成员函数 upper_bound 具有相同的行为 lower_bound，除非映射包含一个 key 值等于 k 的元素：在这种情况下，lower_bound 返回指向该元素的迭代器，而 upper_bound 返回指向下一个元素的迭代器。

```
      iterator lower_bound (const key_type& k);
const_iterator lower_bound (const key_type& k) const;
```

Example

```
#include <iostream>
#include <map>

int main ()
{
  std::map<char,int> mymap;
  std::map<char,int>::iterator itlow,itup;

  mymap['a']=20;
  mymap['b']=40;
  mymap['c']=60;
  mymap['d']=80;
  mymap['e']=100;

  itlow=mymap.lower_bound ('b');  // itlow points to b
  itup=mymap.upper_bound ('d');   // itup points to e (not d!)
```

```
  mymap.erase(itlow,itup);        // erases [itlow,itup)

  // print content:
  for (std::map<char,int>::iterator it=mymap.begin(); it!=mymap.end(); ++it)
    std::cout << it->first << " => " << it->second << '\n';

  return 0;
}
```

Output

```
a => 20
e => 100
```

map::upper_bound

将迭代器返回到上限

返回一个指向容器中第一个元素的迭代器，它的关键字被认为是在 k 之后。

该函数使用其内部比较对象（key_comp）来确定这一点，将迭代器返回到 key_comp（k，element_key）将返回 true 的第一个元素。

如果 map 类用默认的比较类型（less）实例化，则函数返回一个迭代器到第一个元素，其键大于 k。

类似的成员函数 lower_bound 具有与 upper_bound 相同的行为，除了 map 包含一个元素，其键值等于 k：在这种情况下，lower_bound 返回指向该元素的迭代器，而 upper_bound 返回指向下一个元素的迭代器。

```
      iterator upper_bound (const key_type& k);
const_iterator upper_bound (const key_type& k) const;
```

Example

```
#include <iostream>
#include <map>

int main ()
{
  std::map<char,int> mymap;
  std::map<char,int>::iterator itlow,itup;

  mymap['a']=20;
  mymap['b']=40;
  mymap['c']=60;
  mymap['d']=80;
  mymap['e']=100;

  itlow=mymap.lower_bound ('b');  // itlow points to b
  itup=mymap.upper_bound ('d');   // itup points to e (not d!)

  mymap.erase(itlow,itup);        // erases [itlow,itup)

  // print content:
  for (std::map<char,int>::iterator it=mymap.begin(); it!=mymap.end(); ++it)
    std::cout << it->first << " => " << it->second << '\n';

  return 0;
}
```

Output

```
a => 20
e => 100
```

map::equal_range

获取相同元素的范围

返回包含容器中所有具有与 k 等价的键的元素的范围边界。 由于地图容器中的元素具有唯一键，所以返回的范围最多只包含一个元素。

如果没有找到匹配，则返回的范围具有零的长度，与两个迭代器指向具有考虑去后一个密钥对所述第一元件 k, 根据容器的内部比较对象（key_comp）。如果容器的比较对象返回 false，则两个键被认为是等价的。

```
pair<const_iterator,const_iterator> equal_range (const key_type& k) const;
pair<iterator,iterator>             equal_range (const key_type& k);
```

Example

```
#include <iostream>
#include <map>

int main ()
{
  std::map<char,int> mymap;

  mymap['a']=10;
  mymap['b']=20;
  mymap['c']=30;

  std::pair<std::map<char,int>::iterator,std::map<char,int>::iterator> ret;
  ret = mymap.equal_range('b');

  std::cout << "lower bound points to: ";
  std::cout << ret.first->first << " => " << ret.first->second << '\n';

  std::cout << "upper bound points to: ";
  std::cout << ret.second->first << " => " << ret.second->second << '\n';

  return 0;
}
```

Output

```
lower bound points to: 'b' => 20
upper bound points to: 'c' => 30
```

multimap

无序容器（Unordered Container）：unordered_set、unordered_multiset、unordered_map、unordered_multimap

包括：

- unordered_set
- unordered_multiset
- unordered_map
- unordered_multimap

都是以哈希表实现的。

unordered_set、unodered_multiset 结构：

unordered_map、unodered_multimap 结构：

unordered_set
unordered_multiset
unordered_map
unordered_multimap
tuple

元组是一个能够容纳元素集合的对象。每个元素可以是不同的类型。

```
template <class... Types> class tuple;
```

Example

```
#include <iostream>     // std::cout
#include <tuple>        // std::tuple, std::get, std::tie, std::ignore

int main ()
{
  std::tuple<int,char> foo (10,'x');
  auto bar = std::make_tuple ("test", 3.1, 14, 'y');

  std::get<2>(bar) = 100;                                   // access element

  int myint; char mychar;

  std::tie (myint, mychar) = foo;                           // unpack elements
  std::tie (std::ignore, std::ignore, myint, mychar) = bar;  // unpack (with
ignore)

  mychar = std::get<3>(bar);

  std::get<0>(foo) = std::get<2>(bar);
  std::get<1>(foo) = mychar;

  std::cout << "foo contains: ";
  std::cout << std::get<0>(foo) << ' ';
  std::cout << std::get<1>(foo) << '\n';

  return 0;
}
```

Output

```
foo contains: 100 y
```

tuple::tuple

构建一个 tuple（元组）对象。

这涉及单独构建其元素，初始化取决于调用的构造函数形式：

（1）默认的构造函数

构建一个 元组对象的元素值初始化。

（2）复制/移动构造函数

该对象使用 tpl 的内容进行初始化 元组目的。tpl 的相应元素被传递给每个元素的构造函数。
（3）隐式转换构造函数
同上。tpl 中的 所有类型都可以隐含地转换为构造中它们各自元素的类型元组 目的。
（4）初始化构造函数 用 elems 中的相应元素初始化每个元素。elems 的相应元素被传递给每个元素的构造函数。
（5）对转换构造函数
该对象有两个对应于 pr.first 和的元素 pr.second。PR 中的所有类型都应该隐含地转换为其中各自元素的类型元组 目的。
（6）分配器版本
和上面的版本一样，除了每个元素都是使用 allocator alloc 构造的。

```
default (1)
constexpr tuple();
copy / move (2)
tuple (const tuple& tpl) = default;
tuple (tuple&& tpl) = default;
implicit conversion (3)
template <class... UTypes>
  tuple (const tuple<UTypes...>& tpl);
template <class... UTypes>
  tuple (tuple<UTypes...>&& tpl);
initialization (4)
explicit tuple (const Types&... elems);
template <class... UTypes>
  explicit tuple (UTypes&&... elems);
conversion from pair (5)
template <class U1, class U2>
  tuple (const pair<U1,U2>& pr);
template <class U1, class U2>
  tuple (pair<U1,U2>&& pr);
allocator (6)
template<class Alloc>
  tuple (allocator_arg_t aa, const Alloc& alloc);
template<class Alloc>
  tuple (allocator_arg_t aa, const Alloc& alloc, const tuple& tpl);
template<class Alloc>
  tuple (allocator_arg_t aa, const Alloc& alloc, tuple&& tpl);
template<class Alloc,class... UTypes>
  tuple (allocator_arg_t aa, const Alloc& alloc, const tuple<UTypes...>& tpl);
template<class Alloc, class... UTypes>
  tuple (allocator_arg_t aa, const Alloc& alloc, tuple<UTypes...>&& tpl);
template<class Alloc>
  tuple (allocator_arg_t aa, const Alloc& alloc, const Types&... elems);
template<class Alloc, class... UTypes>
  tuple (allocator_arg_t aa, const Alloc& alloc, UTypes&&... elems);
template<class Alloc, class U1, class U2>
  tuple (allocator_arg_t aa, const Alloc& alloc, const pair<U1,U2>& pr);
template<class Alloc, class U1, class U2>
  tuple (allocator_arg_t aa, const Alloc& alloc, pair<U1,U2>&& pr);
```

Example

```
#include <iostream>     // std::cout
#include <utility>      // std::make_pair
```

```cpp
#include <tuple>        // std::tuple, std::make_tuple, std::get

int main ()
{
  std::tuple<int,char> first;                              // default
  std::tuple<int,char> second (first);                     // copy
  std::tuple<int,char> third (std::make_tuple(20,'b'));    // move
  std::tuple<long,char> fourth (third);                    // implicit conversion
  std::tuple<int,char> fifth (10,'a');                     // initialization
  std::tuple<int,char> sixth (std::make_pair(30,'c'));     // from pair / move

  std::cout << "sixth contains: " << std::get<0>(sixth);
  std::cout << " and " << std::get<1>(sixth) << '\n';

  return 0;
}
```

Output

```
sixth contains: 30 and c
```

pair

这个类把一对值（values）结合在一起，这些值可能是不同的类型（T1 和 T2）。每个值可以被公有的成员变量 first、second 访问。

pair 是 tuple（元组）的一个特例。

pair 的实现是一个结构体，主要的两个成员变量是 first second 因为是使用 struct 不是 class，所以可以直接使用 pair 的成员变量。

应用：

- 可以将两个类型数据组合成一个如 map<key,value>
- 当某个函数需要两个返回值时

```cpp
template <class T1, class T2> struct pair;
```

pair::pair

构建一个 pair 对象。

这涉及到单独构建它的两个组件对象，初始化依赖于调用的构造器形式：

（1）默认的构造函数

构建一个 对对象的元素值初始化。

（2）复制/移动构造函数（和隐式转换）

该对象被初始化为 pr 的内容 对目的。pr 的相应成员被传递给每个成员的构造函数。

（3）初始化构造函数

会员 第一是由一个和成员构建的第二与 b。

（4）分段构造

构造成员 first 和 second 到位，传递元素 first_args 作为参数的构造函数 first，和元素 second_args 到的构造函数 second 。

```cpp
default (1)
constexpr pair();
copy / move (2)
template<class U, class V> pair (const pair<U,V>& pr);
template<class U, class V> pair (pair<U,V>&& pr);
pair (const pair& pr) = default;
pair (pair&& pr) = default;
initialization (3)
pair (const first_type& a, const second_type& b);
```

```
template<class U, class V> pair (U&& a, V&& b);
piecewise (4)
template <class... Args1, class... Args2>
  pair (piecewise_construct_t pwc, tuple<Args1...> first_args,
                                   tuple<Args2...> second_args);
```

Example

```
#include <utility>      // std::pair, std::make_pair
#include <string>       // std::string
#include <iostream>     // std::cout

int main () {
  std::pair <std::string,double> product1;                     // default
constructor
  std::pair <std::string,double> product2 ("tomatoes",2.30);   // value init
  std::pair <std::string,double> product3 (product2);          // copy
constructor
  product1 = std::make_pair(std::string("lightbulbs"),0.99);   // using
make_pair (move)
  product2.first = "shoes";                  // the type of first is string
  product2.second = 39.90;                   // the type of second is double

  std::cout << "The price of " << product1.first << " is $" << product1.second
<< '\n';
  std::cout << "The price of " << product2.first << " is $" << product2.second
<< '\n';
  std::cout << "The price of " << product3.first << " is $" << product3.second
<< '\n';
  return 0;
}
```

Output

```
The price of lightbulbs is $0.99
The price of shoes is $39.9
The price of tomatoes is $2.3
```

^乀□ 数据结构

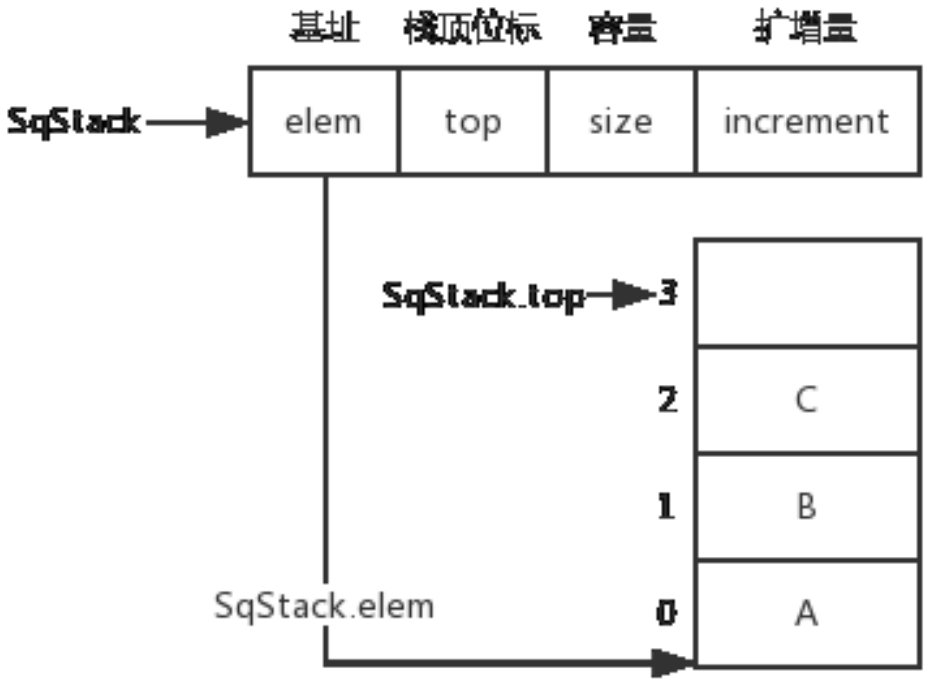
顺序结构

顺序栈（Sequence Stack）

SqStack.cpp

顺序栈数据结构和图片

```
typedef struct {
    ElemType *elem;
    int top;
    int size;
    int increment;
} SqSrack;
```



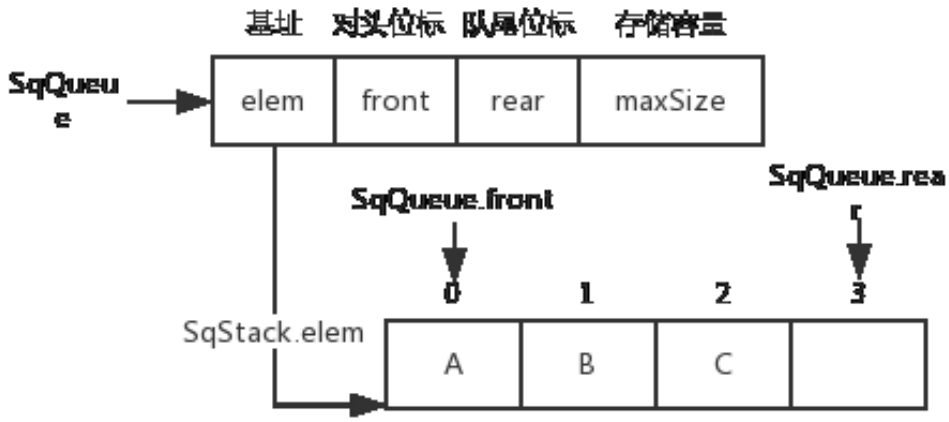
队列（Sequence Queue）

队列数据结构

```
typedef struct {
    ElemType * elem;
    int front;
    int rear;
    int maxSize;
}SqQueue;
```

非循环队列

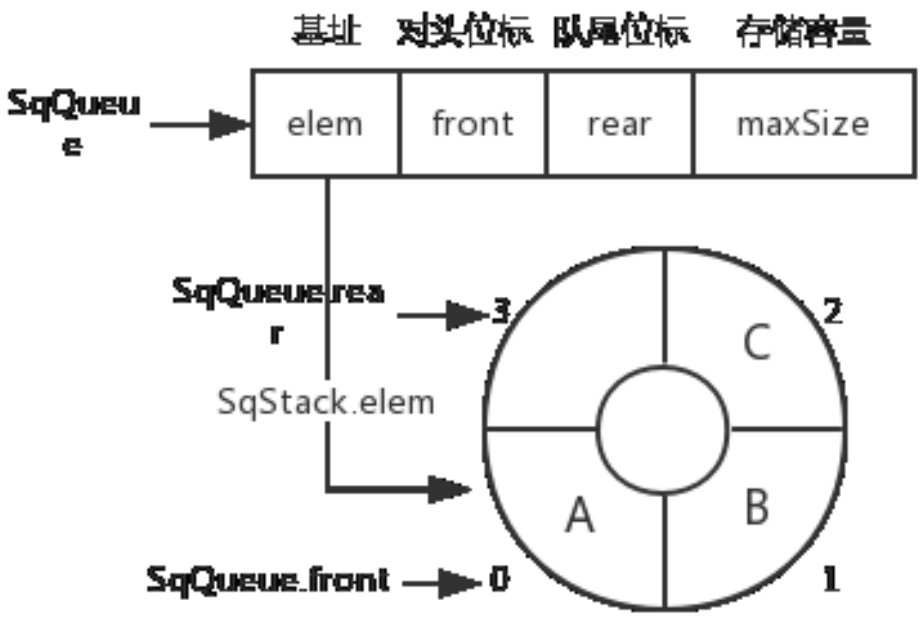
非循环队列图片



SqQueue.rear++

循环队列

循环队列图片



SqQueue.rear = (SqQueue.rear + 1) % SqQueue.maxSize

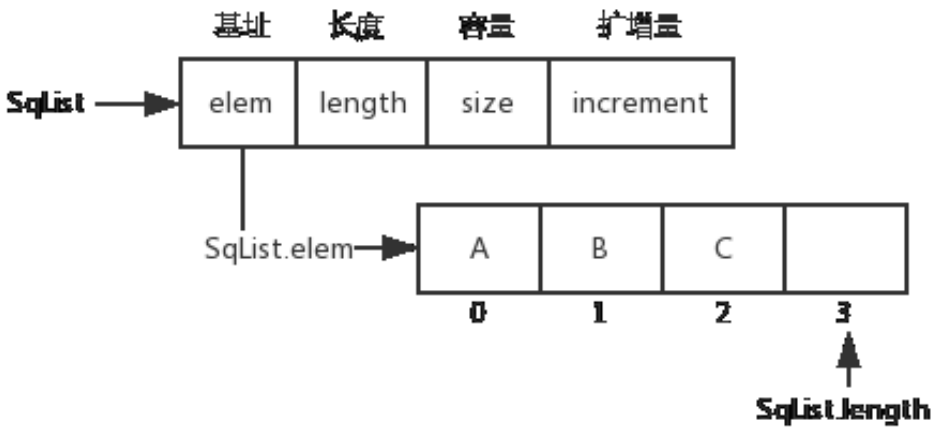
顺序表（Sequence List）

SqList.cpp

顺序表数据结构和图片

```
typedef struct {
    ElemType *elem;
    int length;
    int size;
```

```
    int increment;
} SqList;
```



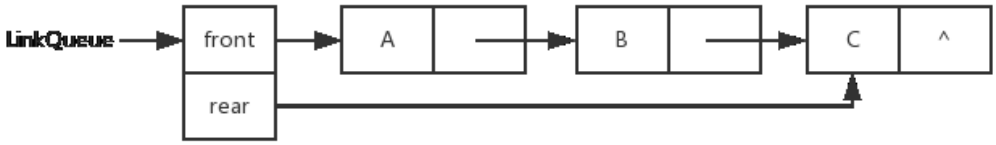
链式结构

LinkList.cpp
LinkList_with_head.cpp
链式数据结构

```
typedef struct LNode {
    ElemType data;
    struct LNode *next;
} LNode, *LinkList;
```

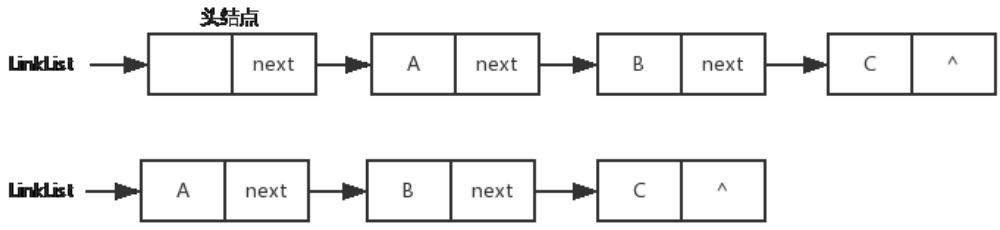
链队列（Link Queue）

链队列图片



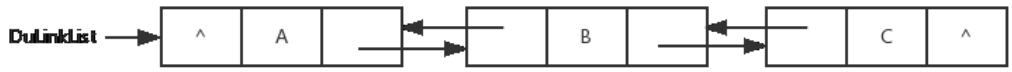
线性表的链式表示

单链表（Link List）
单链表图片



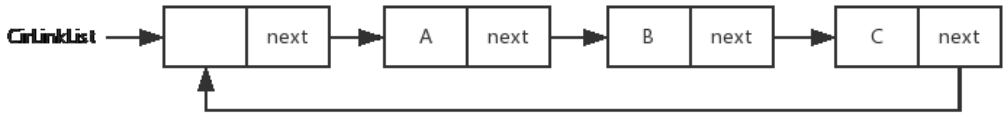
双向链表（Du-Link-List）

双向链表图片



循环链表（Cir-Link-List）

循环链表图片



哈希表

HashTable.cpp

概念

哈希函数：H(key): K -> D , key ∈K

构造方法

- 直接定址法
- 除留余数法
- 数字分析法
- 折叠法
- 平方取中法

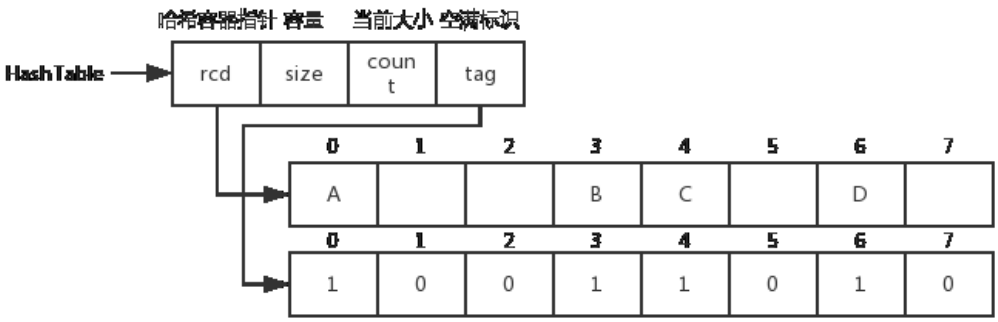
冲突处理方法

- 链地址法：key 相同的用单链表链接
- 开放定址法
  - 线性探测法：key 相同 -> 放到 key 的下一个位置，Hi = (H(key) + i) % m
  - 二次探测法：key 相同 -> 放到 Di = 1^2, -1^2, ..., ±（k)^2,(k<=m/2）
  - 随机探测法：H = (H(key) + 伪随机数) % m

线性探测的哈希表数据结构

线性探测的哈希表数据结构和图片

```
typedef char KeyType;

typedef struct {
    KeyType key;
}RcdType;

typedef struct {
    RcdType *rcd;
    int size;
    int count;
    bool *tag;
}HashTable;
```



递归

概念

函数直接或间接地调用自身

递归与分治

- 分治法
  - 问题的分解
  - 问题规模的分解
- 折半查找（递归）
- 归并查找（递归）
- 快速排序（递归）

递归与迭代

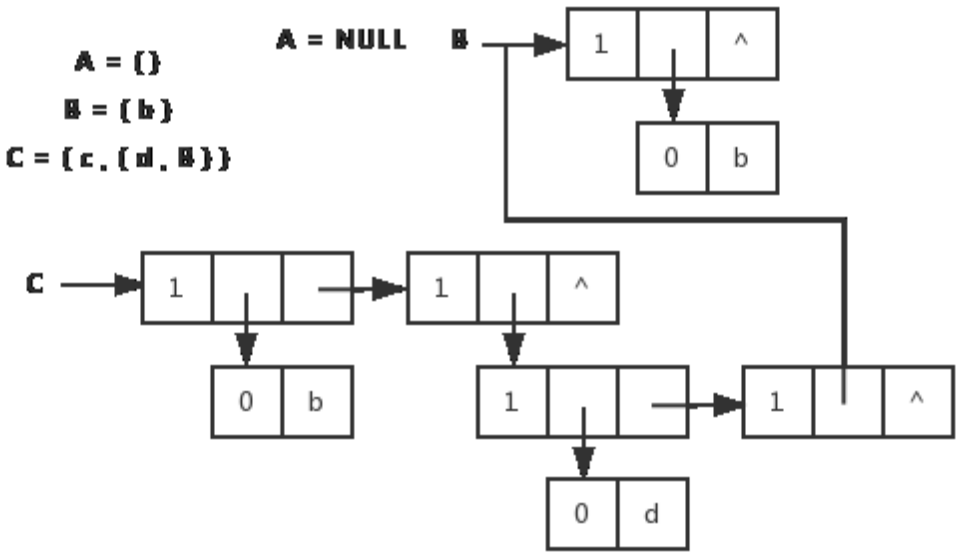
- 迭代：反复利用变量旧值推出新值
- 折半查找（迭代）
- 归并查找（迭代）

广义表

头尾链表存储表示

广义表的头尾链表存储表示和图片

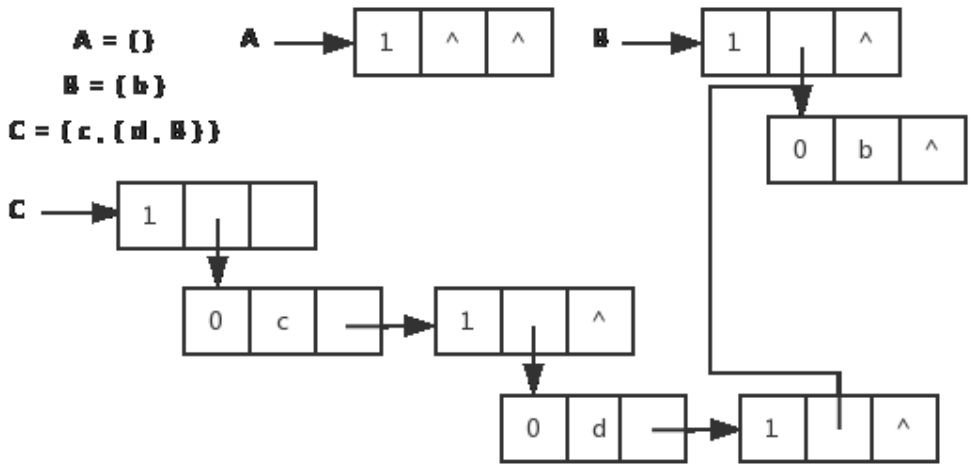
```
// 广义表的头尾链表存储表示
typedef enum {ATOM, LIST} ElemTag;
// ATOM==0：原子，LIST==1：子表
typedef struct GLNode {
    ElemTag tag;
    // 公共部分，用于区分原子结点和表结点
    union {
        // 原子结点和表结点的联合部分
        AtomType atom;
        // atom 是原子结点的值域，AtomType 由用户定义
        struct {
            struct GLNode *hp, *tp;
        } ptr;
        // ptr 是表结点的指针域，prt.hp 和 ptr.tp 分别指向表头和表尾
    } a;
} *GList, GLNode;
```



扩展线性链表存储表示

扩展线性链表存储表示和图片

```
// 广义表的扩展线性链表存储表示
typedef enum {ATOM, LIST} ElemTag;
// ATOM==0：原子，LIST==1：子表
typedef struct GLNode1 {
    ElemTag tag;
    // 公共部分，用于区分原子结点和表结点
    union {
        // 原子结点和表结点的联合部分
        AtomType atom; // 原子结点的值域
        struct GLNode1 *hp; // 表结点的表头指针
    } a;
    struct GLNode1 *tp;
    // 相当于线性链表的 next，指向下一个元素结点
} *GList1, GLNode1;
```



二叉树

BinaryTree.cpp

性质

1. 非空二叉树第 i 层最多 2(i-1) 个结点 （i >= 1）
2. 深度为 k 的二叉树最多 2k - 1 个结点 （k >= 1）
3. 度为 0 的结点数为 n0，度为 2 的结点数为 n2，则 n0 = n2 + 1
4. 有 n 个结点的完全二叉树深度 k = ⌊ log2(n) ⌋ + 1
5. 对于含 n 个结点的完全二叉树中编号为 i （1 <= i <= n） 的结点
    1. 若 i = 1，为根，否则双亲为 ⌊ i / 2 ⌋
    2. 若 2i > n，则 i 结点没有左孩子，否则孩子编号为 2i
    3. 若 2i + 1 > n，则 i 结点没有右孩子，否则孩子编号为 2i + 1

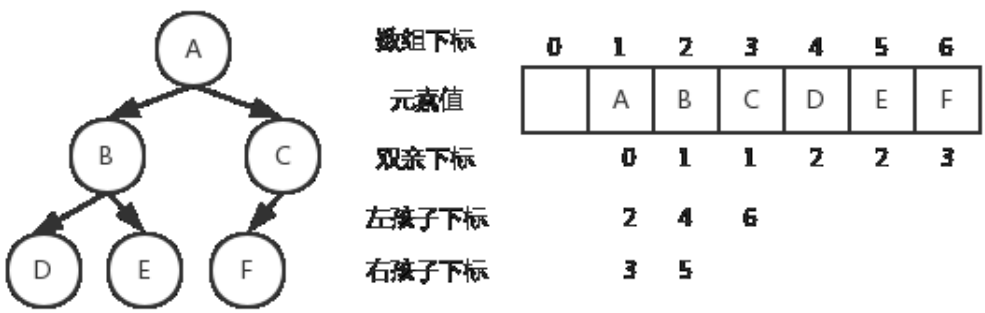
存储结构

二叉树数据结构

```
typedef struct BiTNode
{
    TElemType data;
    struct BiTNode *lchild, *rchild;
}BiTNode, *BiTree;
```

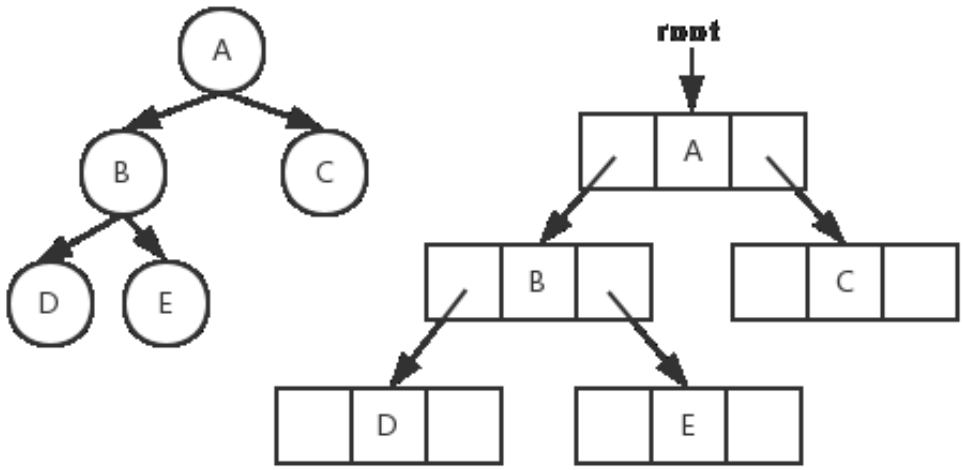
顺序存储

二叉树顺序存储图片



链式存储

二叉树链式存储图片



遍历方式

- 先序遍历
- 中序遍历
- 后续遍历
- 层次遍历

分类

- 满二叉树
- 完全二叉树（堆）
  - 大顶堆：根 >= 左 && 根 >= 右
  - 小顶堆：根 <= 左 && 根 <= 右
- 二叉查找树（二叉排序树）：左 < 根 < 右
- 平衡二叉树（AVL 树）：| 左子树树高 - 右子树树高 | <= 1
- 最小失衡树：平衡二叉树插入新结点导致失衡的子树：调整：
  - LL 型：根的左孩子右旋
  - RR 型：根的右孩子左旋
  - LR 型：根的左孩子左旋，再右旋
  - RL 型：右孩子的左子树，先右旋，再左旋

其他树及森林

树的存储结构

- 双亲表示法
- 双亲孩子表示法
- 孩子兄弟表示法

并查集

一种不相交的子集所构成的集合 S = {S1, S2, ..., Sn}

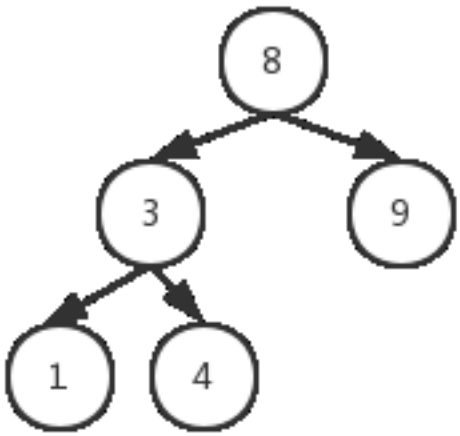
平衡二叉树（AVL 树）

性质

- 左子树树高 - 右子树树高 | <= 1
- 平衡二叉树必定是二叉搜索树，反之则不一定

- 最小二叉平衡树的节点的公式：F(n)=F(n-1)+F(n-2)+1（1 是根节点，F(n-1) 是左子树的节点数量，F(n-2) 是右子树的节点数量）

平衡二叉树图片



最小失衡树

平衡二叉树插入新结点导致失衡的子树

调整：

- LL 型：根的左孩子右旋
- RR 型：根的右孩子左旋
- LR 型：根的左孩子左旋，再右旋
- RL 型：右孩子的左子树，先右旋，再左旋

红黑树

红黑树的特征是什么？

1. 节点是红色或黑色。
2. 根是黑色。
3. 所有叶子都是黑色（叶子是 NIL 节点）。
4. 每个红色节点必须有两个黑色的子节点。（从每个叶子到根的所有路径上不能有两个连续的红色节点。）（新增节点的父节点必须相同）
5. 从任一节点到其每个叶子的所有简单路径都包含相同数目的黑色节点。（新增节点必须为红）

调整

1. 变色
2. 左旋
3. 右旋

应用

- 关联数组：如 STL 中的 map、set

红黑树、B 树、B+ 树的区别？

- 红黑树的深度比较大，而 B 树和 B+ 树的深度则相对要小一些
- B+ 树则将数据都保存在叶子节点，同时通过链表的形式将他们连接在一起。

B 树（B-tree）、B+ 树（B+-tree）

B 树、B+ 树图片

B 树（B-tree）、B+ 树（B+-tree）

B 树（B-tree）、B+ 树（B+-tree）

特点

- 一般化的二叉查找树（binary search tree）
- “矮胖”，内部（非叶子）节点可以拥有可变数量的子节点（数量范围预先定义好）

应用

- 大部分文件系统、数据库系统都采用 B 树、B+树作为索引结构

区别

- B+树中只有叶子节点会带有指向记录的指针（ROWID），而 B 树则所有节点都带有，在内部节点出现的索引项不会再出现在叶子节点中。
- B+树中所有叶子节点都是通过指针连接在一起，而 B 树不会。

B 树的优点

对于在内部节点的数据，可直接得到，不必根据叶子节点来定位。

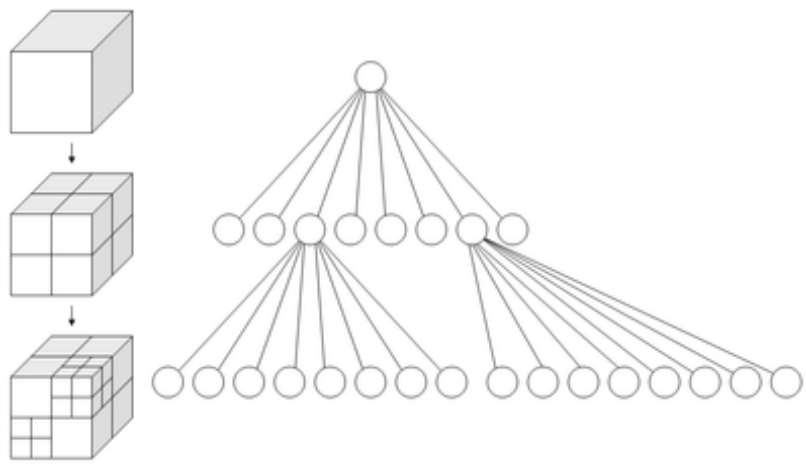
B+树的优点

- 非叶子节点不会带上 ROWID，这样，一个块中可以容纳更多的索引项，一是可以降低树的高度。二是一个内部节点可以定位更多的叶子节点。
- 叶子节点之间通过指针来连接，范围扫描将十分简单，而对于 B 树来说，则需要在叶子节点和内部节点不停的往返移动。

B 树、B+树区别来自：differences-between-b-trees-and-b-trees、B 树和 B+树的区别

八叉树

八叉树图片



八叉树（octree），或称八元树，是一种用于描述三维空间（划分空间）的树状数据结构。八叉树的每个节点表示一个正方体的体积元素，每个节点有八个子节点，这八个子节点所表示的体积元素加在一起就等于父节点的体积。一般中心点作为节点的分叉中心。

用途

- 三维计算机图形
- 最邻近搜索

□□ 算法

排序

排序算法	平均时间复杂度	最差时间复杂度	空间复杂度	数据对象稳定性
冒泡排序	O(n2)	O(n2)	O(1)	稳定
选择排序	O(n2)	O(n2)	O(1)	数组不稳定、链表稳定
插入排序	O(n2)	O(n2)	O(1)	稳定
快速排序	O(n*log2n)	O(n2)	O(log2n)	不稳定
堆排序	O(n*log2n)	O(n*log2n)	O(1)	不稳定
归并排序	O(n*log2n)	O(n*log2n)	O(n)	稳定
希尔排序	O(n*log2n)	O(n2)	O(1)	不稳定

计数排序	O(n+m)	O(n+m)	O(n+m)	稳定
桶排序	O(n)	O(n)	O(m)	稳定
基数排序	O(k*n)	O(n2)		稳定

- 均按从小到大排列
- k：代表数值中的“数位”个数
- n：代表数据规模
- m：代表数据的最大值减最小值
- 来自：wikipedia . 排序算法

查找

查找算法	平均时间复杂度	空间复杂度	查找条件
顺序查找	O(n)	O(1)	无序或有序
二分查找（折半查找）	O(log2n)	O(1)	有序
插值查找	O(log2(log2n))	O(1)	有序
斐波那契查找	O(log2n)	O(1)	有序
哈希查找	O(1)	O(n)	无序或有序
二叉查找树（二叉搜索树查找）	O(log2n)		
红黑树	O(log2n)		
2-3 树	O(log2n - log3n)		
B 树/B+树	O(log2n)		

图搜索算法

图搜索算法	数据结构	遍历时间复杂度	空间复杂度
BFS 广度优先搜索	邻接矩阵邻接链表	O(|v|2)O(|v|+|E|)	O(|v|2)O(|v|+|E|)
DFS 深度优先搜索	邻接矩阵邻接链表	O(|v|2)O(|v|+|E|)	O(|v|2)O(|v|+|E|)

其他算法

算法	思想	应用
分治法	把一个复杂的问题分成两个或更多的相同或相似的子问题，直到最后子问题可以简单的直接求解，原问题的解即子问题的解的合并	循环赛日程安排问题、排序算法（快速排序、归并排序）
动态规划	通过把原问题分解为相对简单的子问题的方式求解复杂问题的方法，适用于有重叠子问题和最优子结构性质的问题	背包问题、斐波那契数列
贪心法	一种在每一步选择中都采取在当前状态下最好或最优（即最有利）的选择，从而希望导致结果是最好或最优的算法	旅行推销员问题（最短路径问题）、最小生成树、哈夫曼编码

Problems

Single Problem

- Chessboard Coverage Problem（棋盘覆盖问题）
- Knapsack Problem（背包问题）
- Neumann Neighbor Problem（冯诺依曼邻居问题）
- Round Robin Problem（循环赛日程安排问题）
- Tubing Problem（输油管道问题）

Leetcode Problems

- Github . haoel/leetcode
- Github . pezy/LeetCode

剑指 Offer

- Github . zhedahht/CodingInterviewChinese2
- Github . gatieme/CodingInterviews

Cracking the Coding Interview 程序员面试金典

- Github . careercup/ctci
- 牛客网 . 程序员面试金典

牛客网

- 牛客网 . 在线编程专题

操作系统

进程与线程

对于有线程系统：*进程是资源分配的独立单位 *线程是资源调度的独立单位
对于无线程系统：*进程是资源调度、分配的独立单位

进程之间的通信方式以及优缺点

- 管道（PIPE）
  - 有名管道：一种半双工的通信方式，它允许无亲缘关系进程间的通信
    - 优点：可以实现任意关系的进程间的通信
    - 缺点：
      1. 长期存于系统中，使用不当容易出错
      2. 缓冲区有限
  - 无名管道：一种半双工的通信方式，只能在具有亲缘关系的进程间使用（父子进程）
    - 优点：简单方便
    - 缺点：
      1. 局限于单向通信
      2. 只能创建在它的进程以及其有亲缘关系的进程之间
      3. 缓冲区有限
- 信号量（Semaphore）：一个计数器，可以用来控制多个线程对共享资源的访问
  - 优点：可以同步进程
  - 缺点：信号量有限
- 信号（Signal）：一种比较复杂的通信方式，用于通知接收进程某个事件已经发生
- 消息队列（Message Queue）：是消息的链表，存放在内核中并由消息队列标识符标识
  - 优点：可以实现任意进程间的通信，并通过系统调用函数来实现消息发送和接收之间的同步，无需考虑同步问题，方便
  - 缺点：信息的复制需要额外消耗 CPU 的时间，不适宜于信息量大或操作频繁的场合
- 共享内存（Shared Memory）：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
  - 优点：无须复制，快捷，信息量大
  - 缺点：
    1. 通信是通过将共享空间缓冲区直接附加到进程的虚拟地址空间中来实现的，因此进程间的读写操作的同步问题
    2. 利用内存缓冲区直接交换信息，内存的实体存在于计算机中，只能同一个计算机系统中的诸多进程共享，不方便网络通信
- 套接字（Socket）：可用于不同及其间的进程通信
  - 优点：
    1. 传输数据为字节级，传输数据可自定义，数据量小效率高
    2. 传输数据时间短，性能高
    3. 适合于客户端和服务器端之间信息实时交互
    4. 可以加密,数据安全性强
  - 缺点：需对传输的数据进行解析，转化成应用级的数据。

线程之间的通信方式

- 锁机制：包括互斥锁/量（mutex）、读写锁（reader-writer lock）、自旋锁（spin lock）、条件变量（condition）
  - 互斥锁/量（mutex）：提供了以排他方式防止数据结构被并发修改的方法。
  - 读写锁（reader-writer lock）：允许多个线程同时读共享数据，而对写操作是互斥的。

  - 自旋锁（spin lock）与互斥锁类似，都是为了保护共享资源。互斥锁是当资源被占用，申请者进入睡眠状态；而自旋锁则循环检测保持者是否已经释放锁。
  - 条件变量（condition）：可以以原子的方式阻塞进程，直到某个特定条件为真为止。对条件的测试是在互斥锁的保护下进行的。条件变量始终与互斥锁一起使用。
- 信号量机制(Semaphore)
  - 无名线程信号量
  - 命名线程信号量
- 信号机制(Signal)：类似进程间的信号处理
- 屏障（barrier）：屏障允许每个线程等待，直到所有的合作线程都达到某一点，然后从该点继续执行。

线程间的通信目的主要是用于线程同步，所以线程没有像进程通信中的用于数据交换的通信机制
进程之间的通信方式以及优缺点来源于：进程线程面试题总结

进程之间私有和共享的资源

- 私有：地址空间、堆、全局变量、栈、寄存器
- 共享：代码段，公共数据，进程目录，进程 ID

线程之间私有和共享的资源

- 私有：线程栈，寄存器，程序寄存器
- 共享：堆，地址空间，全局变量，静态变量

多进程与多线程间的对比、优劣与选择

对比

对比维度	多进程	多线程	总结
数据共享、同步	数据共享复杂，需要用 IPC；数据是分开的，同步简单	因为共享进程数据，数据共享简单，但也是因为这个原因导致同步复杂	各有优势
内存、CPU	占用内存多，切换复杂，CPU 利用率低	占用内存少，切换简单，CPU 利用率高	线程占优
创建销毁、切换	创建销毁、切换复杂，速度慢	创建销毁、切换简单，速度很快	线程占优
编程、调试	编程简单，调试简单	编程复杂，调试复杂	进程占优
可靠性	进程间不会互相影响	一个线程挂掉将导致整个进程挂掉	进程占优
分布式	适应于多核、多机分布式；如果一台机器不够，扩展到多台机器比较简单	适应于多核分布式	进程占优

优劣

优劣	多进程	多线程
优点	编程、调试简单，可靠性较高	创建、销毁、切换速度快，内存、资源占用小
缺点	创建、销毁、切换速度慢，内存、资源占用大	编程、调试复杂，可靠性较差

选择

- 需要频繁创建销毁的优先用线程
- 需要进行大量计算的优先使用线程
- 强相关的处理用线程，弱相关的处理用进程
- 可能要扩展到多机分布的用进程，多核分布的用线程
- 都满足需求的情况下，用你最熟悉、最拿手的方式

多进程与多线程间的对比、优劣与选择来自：多线程还是多进程的选择及区别

Linux 内核的同步方式

原因

在现代操作系统里，同一时间可能有多个内核执行流在执行，因此内核其实象多进程多线程编程一样也需要一些同步机制来同步各执行单元对共享数据的访问。尤其是在多处理器系统上，更需要一些同步机制来同步不同处理器上的执行单元对共享的数据的访问。

同步方式

- 原子操作
- 信号量（semaphore）
- 读写信号量（rw_semaphore）
- 自旋锁（spinlock）
- 大内核锁（BKL，Big Kernel Lock）
- 读写锁（rwlock）
- 大读者锁（brlock-Big Reader Lock）
- 读-拷贝修改(RCU，Read-Copy Update)
- 顺序锁（seqlock）

来自：Linux 内核的同步机制，第 1 部分、Linux 内核的同步机制，第 2 部分

死锁

原因

- 系统资源不足
- 资源分配不当
- 进程运行推进顺序不合适

产生条件

- 互斥
- 请求和保持
- 不剥夺
- 环路

预防

- 打破互斥条件：改造独占性资源为虚拟资源，大部分资源已无法改造。
- 打破不可抢占条件：当一进程占有一独占性资源后又申请一独占性资源而无法满足，则退出原占有的资源。
- 打破占有且申请条件：采用资源预先分配策略，即进程运行前申请全部资源，满足则运行，不然就等待，这样就不会占有且申请。
- 打破循环等待条件：实现资源有序分配策略，对所有设备实现分类编号，所有进程只能采用按序号递增的形式申请资源。
- 有序资源分配法
- 银行家算法

文件系统

- Windows：FCB 表 + FAT + 位图
- Unix：inode + 混合索引 + 成组链接

主机字节序与网络字节序

主机字节序（CPU 字节序）

概念

主机字节序又叫 CPU 字节序，其不是由操作系统决定的，而是由 CPU 指令集架构决定的。主机字节序分为两种：

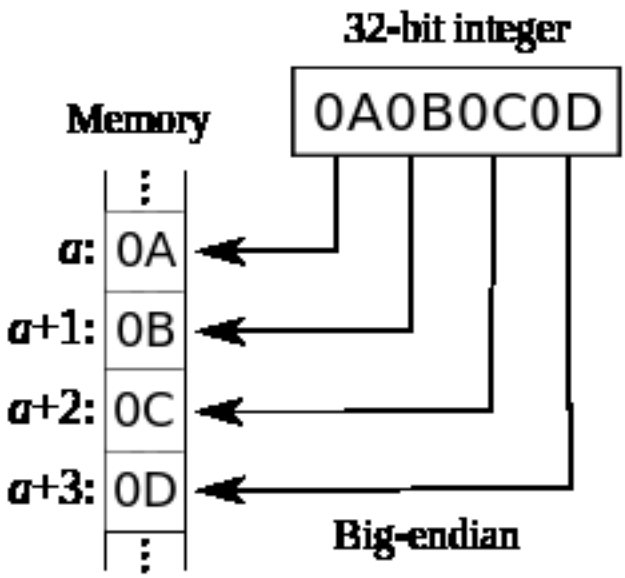
- 大端字节序（Big Endian）：高序字节存储在低位地址，低序字节存储在高位地址
- 小端字节序（Little Endian）：高序字节存储在高位地址，低序字节存储在低位地址

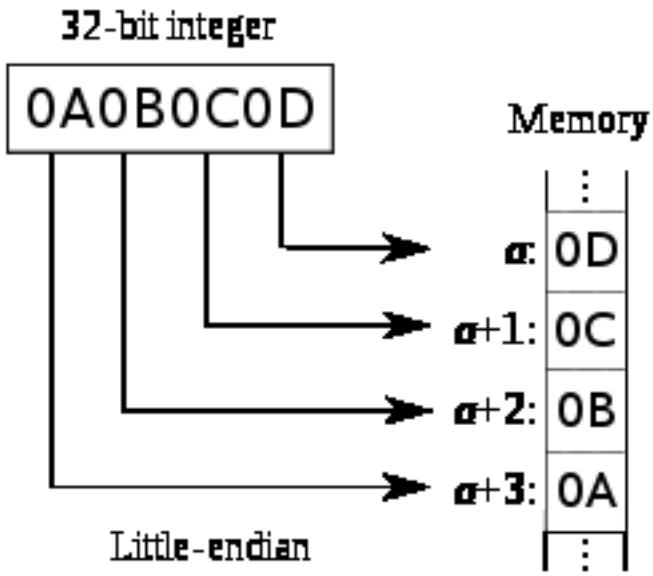
存储方式

32 位整数 `0x12345678` 是从起始位置为 `0x00` 的地址开始存放，则：

内存地址	0x00	0x01	0x02	0x03
大端	12	34	56	78
小端	78	56	34	12

大端小端图片





判断大端小端

判断大端小端

可以这样判断自己 CPU 字节序是大端还是小端：

```
#include <iostream>
using namespace std;

int main()
{
```

```
    int i = 0x12345678;

    if (*((char*)&i) == 0x12)
        cout << "大端" << endl;
    else
        cout << "小端" << endl;

    return 0;
}
```

各架构处理器的字节序

- x86（Intel、AMD）、MOS Technology 6502、Z80、VAX、PDP-11 等处理器为小端序；
- Motorola 6800、Motorola 68000、PowerPC 970、System/370、SPARC（除 V9 外）等处理器为大端序；
- ARM（默认小端序）、PowerPC（除 PowerPC 970 外）、DEC Alpha、SPARC V9、MIPS、PA-RISC 及 IA64 的字节序是可配置的。

网络字节序

网络字节顺序是 TCP/IP 中规定好的一种数据表示格式，它与具体的 CPU 类型、操作系统等无关，从而可以保重数据在不同主机之间传输时能够被正确解释。
网络字节顺序采用：大端（Big Endian）排列方式。

页面置换算法

在地址映射过程中，若在页面中发现所要访问的页面不在内存中，则产生缺页中断。当发生缺页中断时，如果操作系统内存中没有空闲页面，则操作系统必须在内存选择一个页面将其移出内存，以便为即将调入的页面让出空间。而用来选择淘汰哪一页的规则叫做页面置换算法。

分类

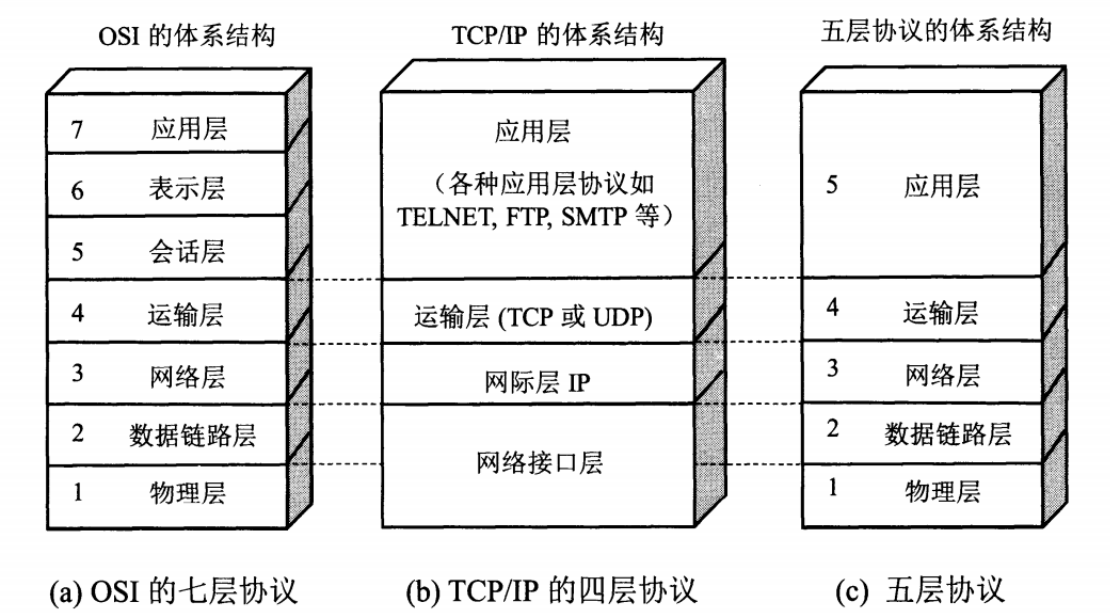
- 全局置换：在整个内存空间置换
- 局部置换：在本进程中进行置换

算法

全局：*工作集算法*缺页率置换算法
局部：*最佳置换算法（OPT）*先进先出置换算法（FIFO）*最近最久未使用（LRU）算法*时钟（Clock）置换算法

□□ 计算机网络

本节部分知识点来自《计算机网络（第 7 版）》
计算机网络体系结构：



计算机网络体系结构

各层作用及协议

分层	作用	协议
物理层	通过媒介传输比特，确定机械及电气规范（比特 Bit）	RJ45、CLOCK、IEEE802.3（中继器，集线器）
数据链路层	将比特组装成帧和点到点的传递（帧 Frame）	PPP、FR、HDLC、VLAN、MAC（网桥，交换机）
网络层	负责数据包从源到宿的传递和网际互连（包 Packet）	IP、ICMP、ARP、RARP、OSPF、IPX、RIP、IGRP（路由器）
运输层	提供端到端的可靠报文传递和错误恢复（段 Segment）	TCP、UDP、SPX
会话层	建立、管理和终止会话（会话协议数据单元 SPDU）	NFS、SQL、NETBIOS、RPC
表示层	对数据进行翻译、加密和压缩（表示协议数据单元 PPDU）	JPEG、MPEG、ASII
应用层	允许访问 OSI 环境的手段（应用协议数据单元 APDU）	FTP、DNS、Telnet、SMTP、HTTP、WWW、NFS

物理层

- 传输数据的单位：比特
- 数据传输系统：源系统（源点、发送器）–> 传输系统 –> 目的系统（接收器、终点）

通道：＊单向通道（单工通道）：只有一个方向通信，没有反方向交互，如广播＊双向交替通信（半双工通信）：通信双方都可发消息，但不能同时发送或接收＊双向同时通信（全双工通信）：通信双方可以同时发送和接收信息

通道复用技术：＊频分复用（FDM，Frequency Division Multiplexing）：不同用户在不同频带，所用用户在同样时间占用不同带宽资源＊时分复用（TDM，Time Division Multiplexing）：不同用户在同一时间段的不同时间片，所有用户在不同时间占用同样的频带宽度＊波分复用（WDM，Wavelength Division Multiplexing）：光的频分复用＊码分复用（CDM，Code Division Multiplexing）：不同用户使用不同的码，可以在同样时间使用同样频带通信

数据链路层

主要信道：＊点对点信道＊广播信道

点对点信道

- 数据单元：帧

三个基本问题：*封装成帧：把网络层的IP数据报封装成帧，SOH - 数据部分 - EOT *透明传输：不管数据部分什么字符，都能传输出去；可以通过字节填充方法解决（冲突字符前加转义字符）*差错检测：降低误码率（BER，Bit Error Rate），广泛使用循环冗余检测（CRC，Cyclic Redundancy Check）

点对点协议（Point-to-Point Protocol）：*点对点协议（Point-to-Point Protocol）：用户计算机和ISP通信时所使用的协议

广播信道

广播通信：*硬件地址（物理地址、MAC地址）*单播（unicast）帧（一对一）：收到的帧的MAC地址与本站的硬件地址相同*广播（broadcast）帧（一对全体）：发送给本局域网上所有站点的帧*多播（multicast）帧（一对多）：发送给本局域网上一部分站点的帧

网络层

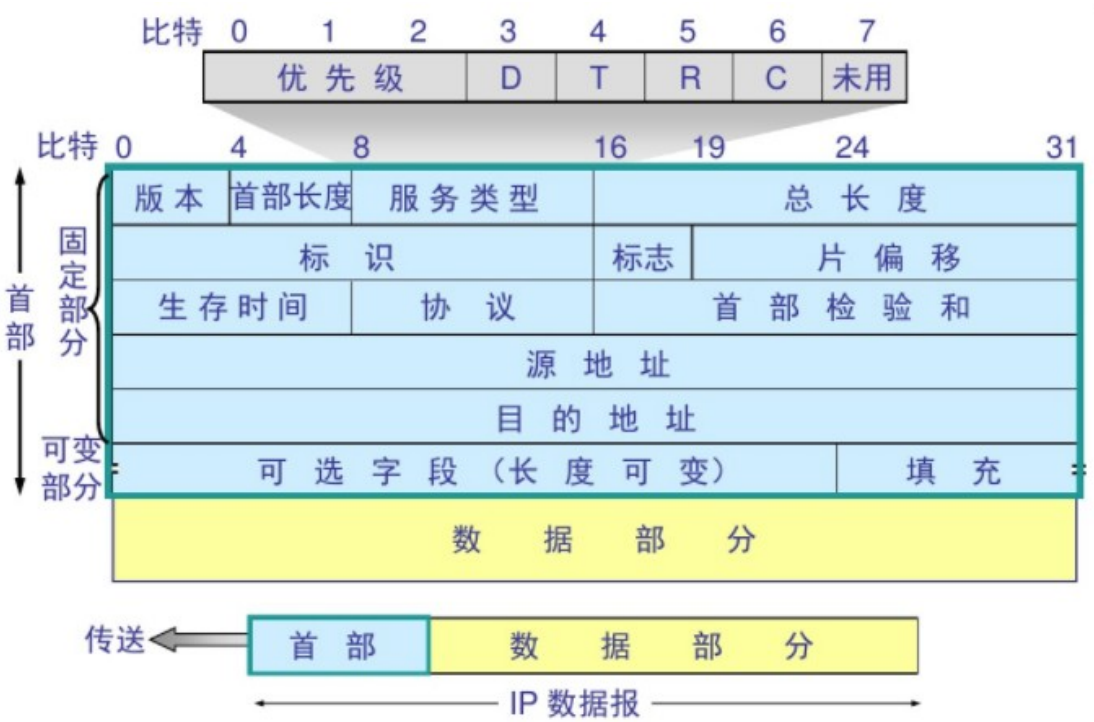
- IP（Internet Protocol，网际协议）是为计算机网络相互连接进行通信而设计的协议。
- ARP（Address Resolution Protocol，地址解析协议）
- ICMP（Internet Control Message Protocol，网际控制报文协议）
- IGMP（Internet Group Management Protocol，网际组管理协议）

IP网际协议

IP地址分类：*IP 地址 ::= {<网络号>,<主机号>}

IP地址类别	网络号	网络范围	主机号	IP地址范围
A类	8bit，第一位固定为0	0 —— 127	24bit	1.0.0.0 —— 127.255.255.255
B类	16bit，前两位固定为10	128.0 —— 191.255	16bit	128.0.0.0 —— 191.255.255.255
C类	24bit，前三位固定为110	192.0.0 —— 223.255.255	8bit	192.0.0.0 —— 223.255.255.255
D类	前四位固定为1110，后面为多播地址			
E类	前五位固定为11110，后面保留为今后所用			

IP数据报格式：



IP 数据报格式

ICMP 网际控制报文协议

ICMP 报文格式：

ICMP 报文格式

应用：* PING（Packet InterNet Groper，分组网间探测）测试两个主机之间的连通性 * TTL（Time To Live，生存时间）该字段指定 IP 包被路由器丢弃之前允许通过的最大网段数量

内部网关协议

- RIP（Routing Information Protocol，路由信息协议）
- OSPF（Open Sortest Path First，开放最短路径优先）

外部网关协议

- BGP（Border Gateway Protocol，边界网关协议）

IP 多播

- IGMP（Internet Group Management Protocol，网际组管理协议）
- 多播路由选择协议

VPN 和 NAT

- VPN（Virtual Private Network，虚拟专用网）
- NAT（Network Address Translation，网络地址转换）

路由表包含什么？

1. 网络 ID（Network ID, Network number）：就是目标地址的网络 ID。

2. 子网掩码（subnet mask）：用来判断 IP 所属网络
3. 下一跳地址/接口（Next hop / interface）：就是数据在发送到目标地址的旅途中下一站的地址。其中 interface 指向 next hop（即为下一个 route）。一个自治系统（AS, Autonomous system）中的 route 应该包含区域内所有的子网络，而默认网关（Network id: `0.0.0.0`, Netmask: `0.0.0.0`）指向自治系统的出口。

根据应用和执行的不同，路由表可能含有如下附加信息：

1. 花费（Cost）：就是数据发送过程中通过路径所需要的花费。
2. 路由的服务质量
3. 路由中需要过滤的出/入连接列表

运输层

协议：

- TCP（Transmission Control Protocol，传输控制协议）
- UDP（User Datagram Protocol，用户数据报协议）

端口：

应用程序	FTP	TELNET	SMTP	DNS	TFTP	HTTP	HTTPS	SNMP
端口号	21	23	25	53	69	80	443	161

TCP

- TCP（Transmission Control Protocol，传输控制协议）是一种面向连接的、可靠的、基于字节流的传输层通信协议，其传输的单位是报文段。

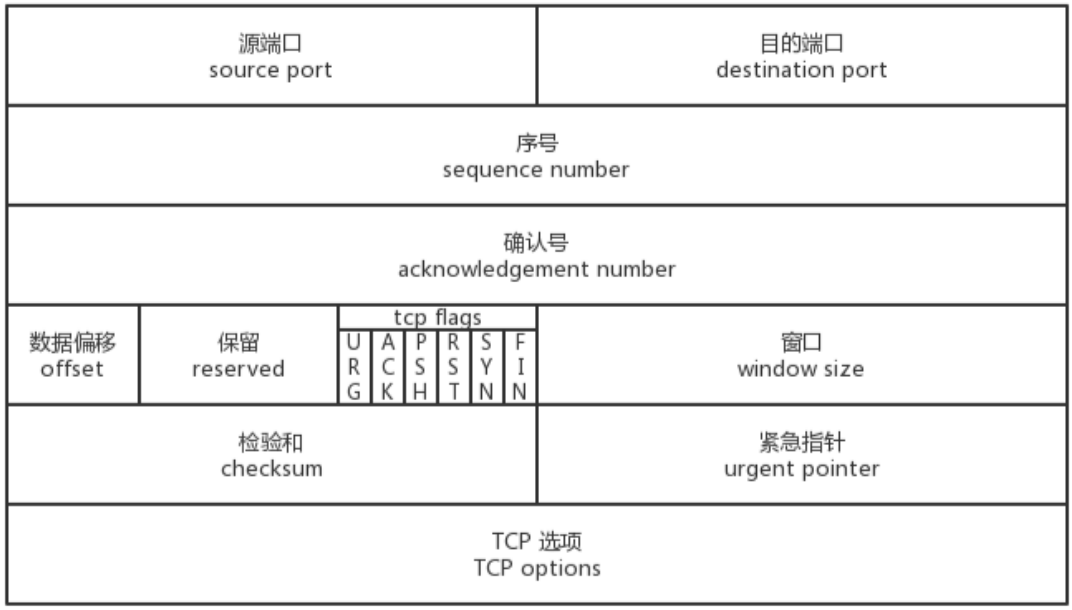
特征：* 面向连接 * 只能点对点（一对一）通信 * 可靠交互 * 全双工通信 * 面向字节流

TCP 如何保证可靠传输：* 确认和超时重传 * 数据合理分片和排序 * 流量控制 * 拥塞控制 * 数据校验

TCP 报文结构

TCP 报文

TCP 首部



TCP 首部

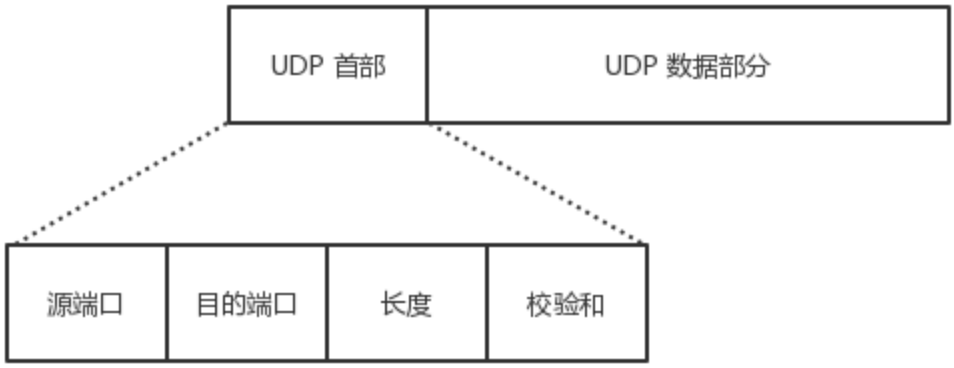
TCP：状态控制码（Code，Control Flag），占 6 比特，含义如下：* URG：紧急比特（urgent），当 URG＝1 时，表明紧急指针字段有效，代表该封包为紧急封包。它告诉系统此报文段中有紧急数据，应尽快传送(相当于高优先级的数据)， 且上图中的 Urgent Pointer 字段也会被启用。* ACK：确认比特（Acknowledge）。只有当 ACK＝1 时确认号字段才有效，代表这个封包为确认封包。当 ACK＝0 时，确认号无效。* PSH：（Push function）若为 1 时，代表要求对方立即传送缓冲区内的其他对应封包，而无需等缓冲满了才送。* RST：复位比特(Reset)，当 RST＝1 时，表明 TCP 连接中出现严重差错（如由于主机崩溃或其他原因），必须释放连接，然后再重新建立运输连接。* SYN：同步比特(Synchronous)，SYN 置为 1，就表示这是一个连接请求或连接接受报文，通常带有 SYN 标志的封包表示『主动』要连接到对方的意思。* FIN：终止比特(Final)，用来释放一个连接。当 FIN＝1 时，表明此报文段的发送端的数据已发送完毕，并要求释放运输连接。

UDP

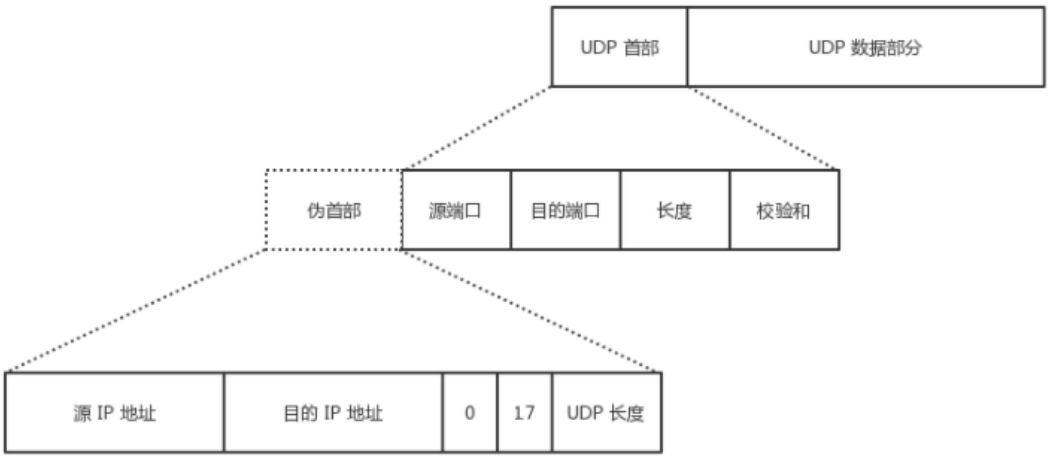
- UDP（User Datagram Protocol，用户数据报协议）是 OSI（Open System Interconnection 开放式系统互联） 参考模型中一种无连接的传输层协议，提供面向事务的简单不可靠信息传送服务，其传输的单位是用户数据报。

特征：* 无连接 * 尽最大努力交付 * 面向报文 * 没有拥塞控制 * 支持一对一、一对多、多对一、多对多的交互通信 * 首部开销小

UDP 报文结构



UDP 报文
UDP 首部



UDP 首部
TCP/UDP 图片来源于：https://github.com/JerryC8080/understand-tcp-udp

TCP 与 UDP 的区别

1. TCP 面向连接，UDP 是无连接的；
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 的逻辑通信信道是全双工的可靠信道；UDP 则是不可靠信道
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 面向字节流（可能出现黏包问题），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（不会出现黏包问题）
6. 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）
7. TCP 首部开销 20 字节；UDP 的首部开销小，只有 8 个字节

TCP 黏包问题

原因

TCP 是一个基于字节流的传输服务（UDP 基于报文的），“流” 意味着 TCP 所传输的数据是没有边界的。所以可能会出现两个数据包黏在一起的情况。

解决

- 发送定长包。如果每个消息的大小都是一样的，那么在接收对等方只要累计接收数据，直到数据等于一个定长的数值就将它作为一个消息。

- 包头加上包体长度。包头是定长的 4 个字节，说明了包体的长度。接收对等方先接收包头长度，依据包头长度来接收包体。
- 在数据包之间设置边界，如添加特殊符号 \r\n 标记。FTP 协议正是这么做的。但问题在于如果数据正文中也含有 \r\n，则会误判为消息的边界。
- 使用更加复杂的应用层协议。

TCP 流量控制

概念

流量控制（flow control）就是让发送方的发送速率不要太快，要让接收方来得及接收。

方法

利用可变窗口进行流量控制

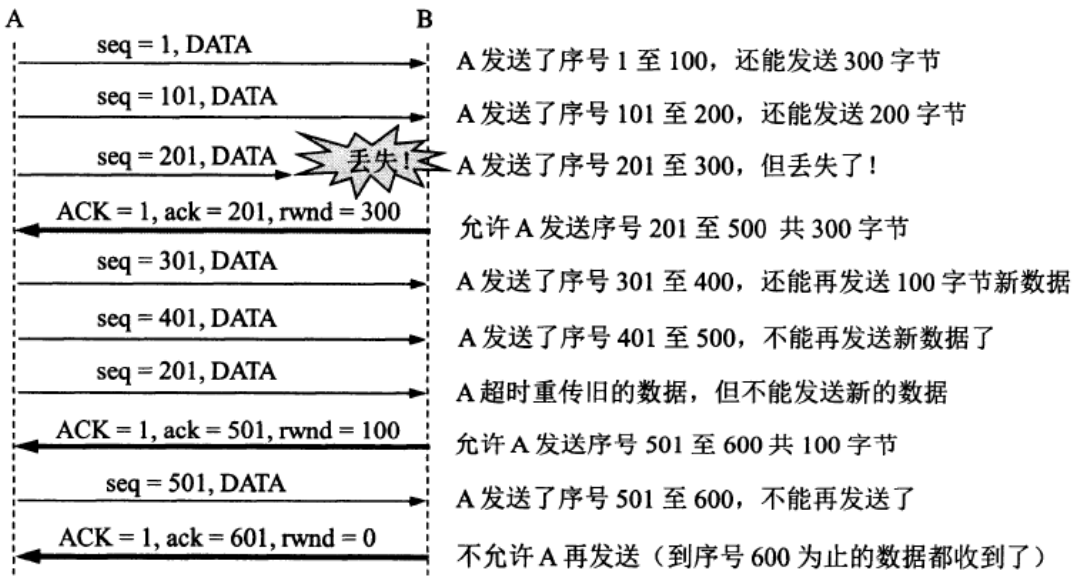


图 5-22　利用可变窗口进行流量控制举例

TCP 拥塞控制

概念

拥塞控制就是防止过多的数据注入到网络中，这样可以使网络中的路由器或链路不致过载。

方法

- 慢开始(slow-start)
- 拥塞避免(congestion avoidance)
- 快重传(fast retransmit)
- 快恢复(fast recovery)

TCP 的拥塞控制图

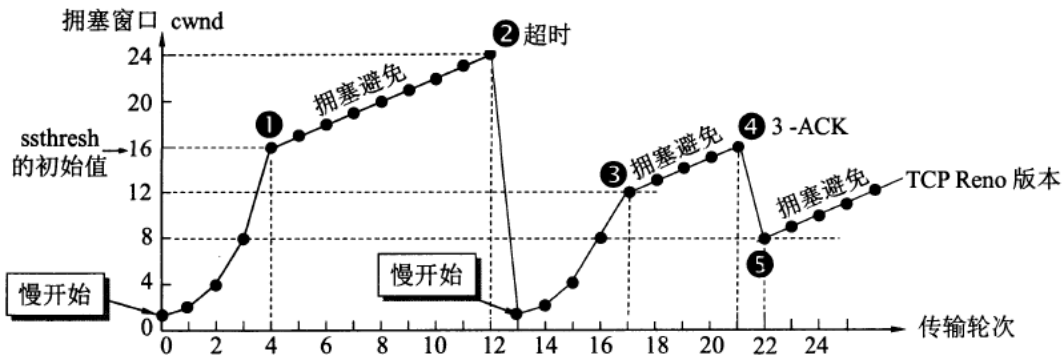


图 5-25　TCP 拥塞窗口 cwnd 在拥塞控制时的变化情况

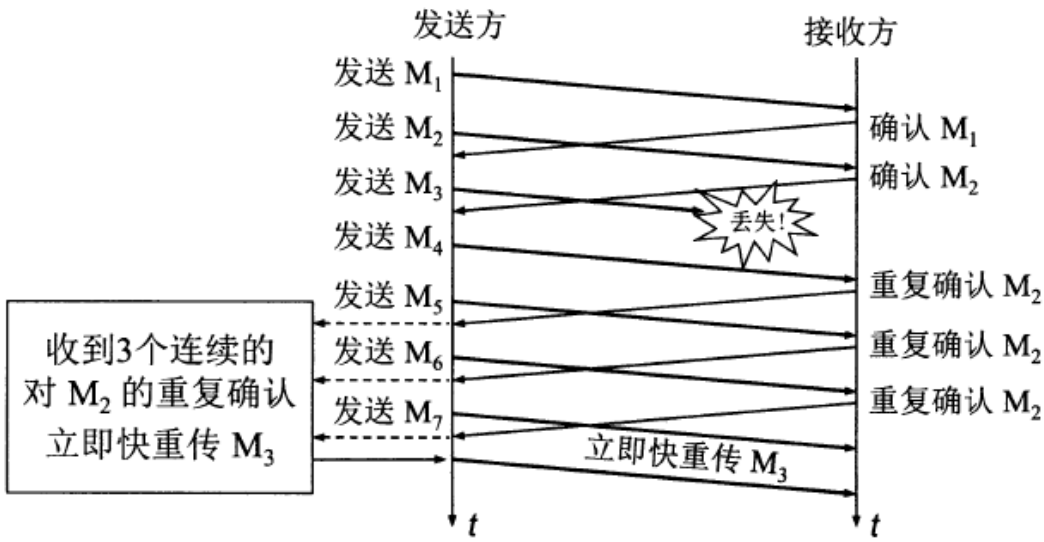


图 5-26　快重传的示意图

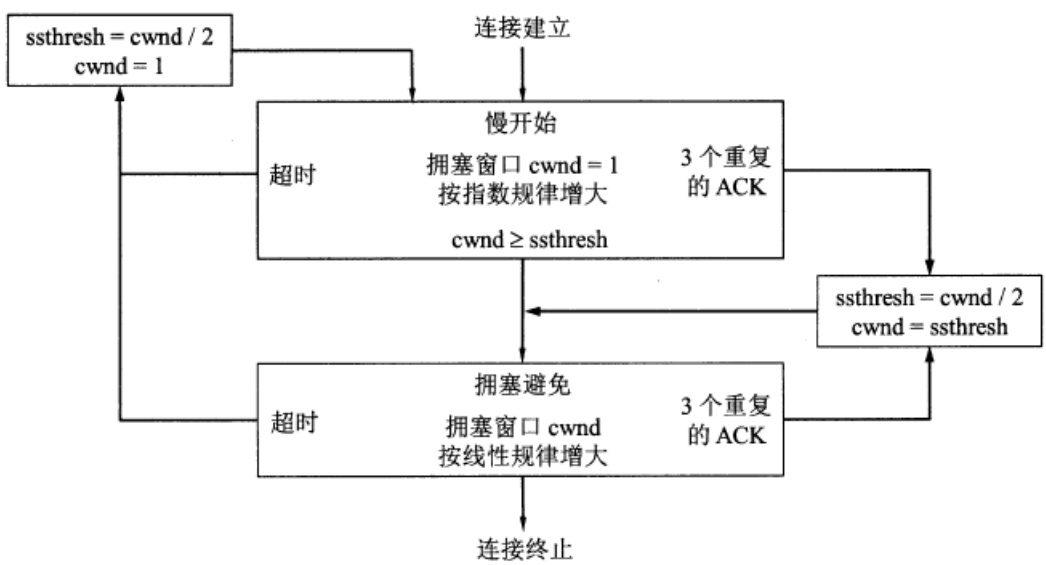


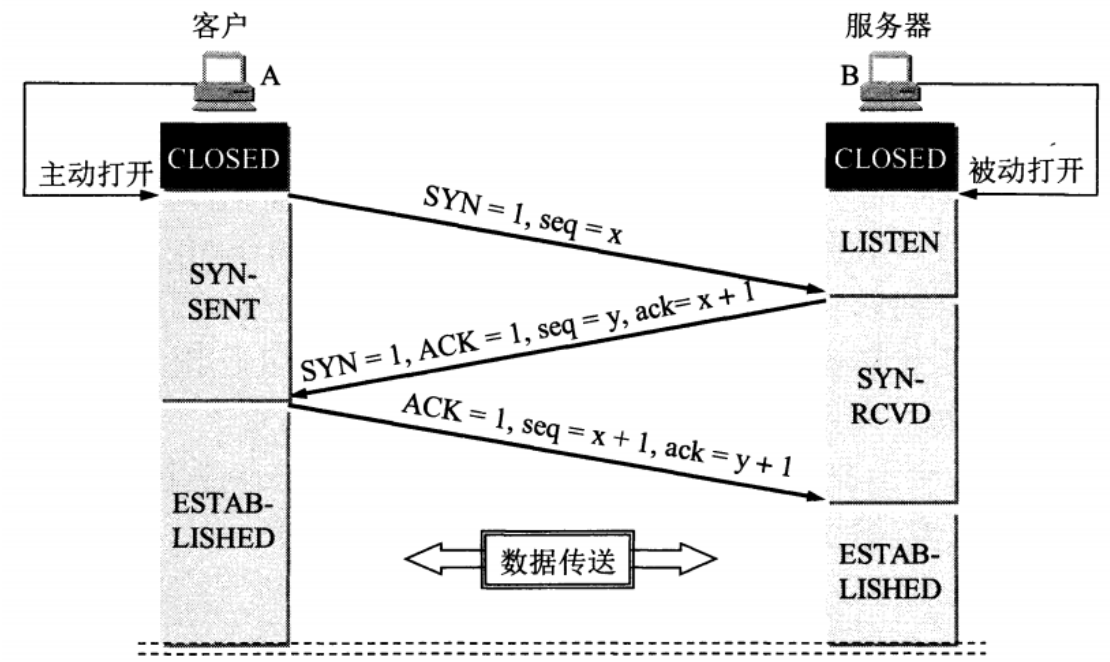
图 5-27　TCP 的拥塞控制的流程图

TCP 传输连接管理

因为 TCP 三次握手建立连接、四次挥手释放连接很重要，所以附上《计算机网络（第 7 版）-谢希仁》书中对此章的详细描述：

https://github.com/huihut/interview/blob/master/images/TCP-transport-connection-management.png

TCP 三次握手建立连接



UDP 报文

【TCP 建立连接全过程解释】

1. 客户端发送 SYN 给服务器，说明客户端请求建立连接；
2. 服务端收到客户端发的 SYN，并回复 SYN+ACK 给客户端（同意建立连接）；
3. 客户端收到服务端的 SYN+ACK 后，回复 ACK 给服务端（表示客户端收到了服务端发的同意报文）；
4. 服务端收到客户端的 ACK，连接已建立，可以数据传输。

TCP 为什么要进行三次握手？

【答案一】因为信道不可靠，而 TCP 想在不可靠信道上建立可靠地传输，那么三次通信是理论上的最小值。（而 UDP 则不需建立可靠传输，因此 UDP 不需要三次握手。）

Google Groups . TCP 建立连接为什么是三次握手？{技术}{网络通信}

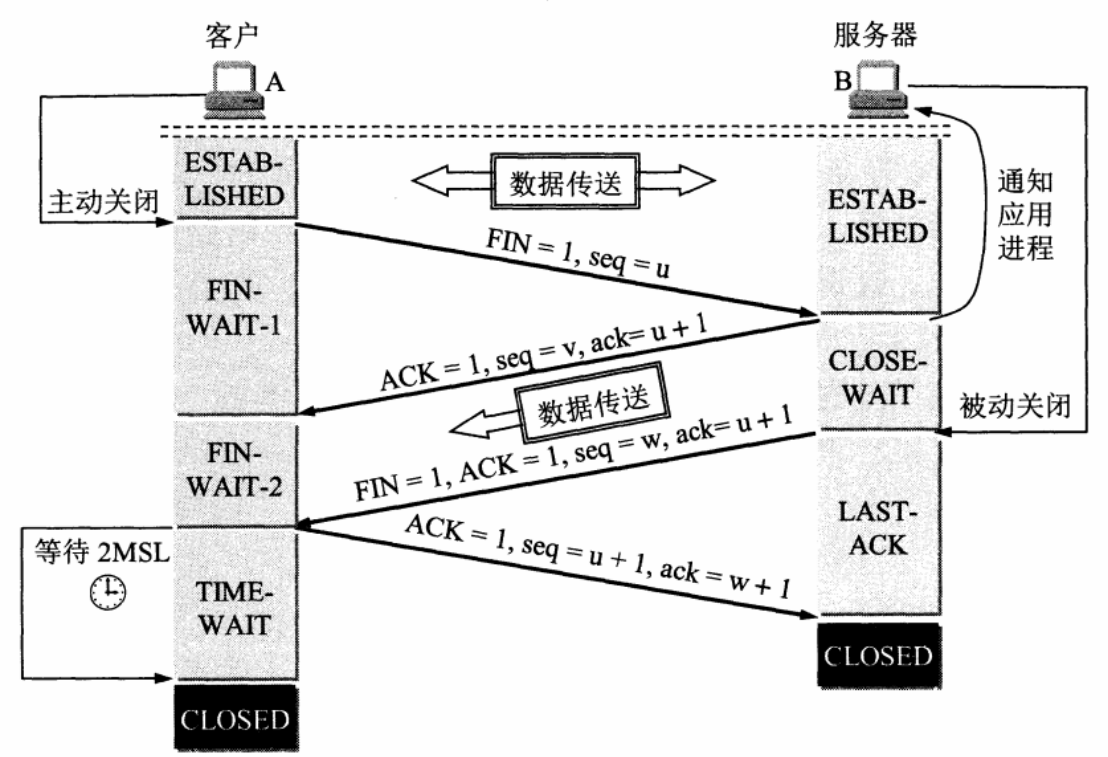
【答案二】因为双方都需要确认对方收到了自己发送的序列号，确认过程最少要进行三次通信。

知乎 . TCP 为什么是三次握手，而不是两次或四次？

【答案三】为了防止已失效的连接请求报文段突然又传送到了服务端，因而产生错误。

《计算机网络（第 7 版）-谢希仁》

TCP 四次挥手释放连接



UDP 报文

【TCP 释放连接全过程解释】

1. 客户端发送 FIN 给服务器，说明客户端不必发送数据给服务器了（请求释放从客户端到服务器的连接）；
2. 服务器接收到客户端发的 FIN，并回复 ACK 给客户端（同意释放从客户端到服务器的连接）；
3. 客户端收到服务端回复的 ACK，此时从客户端到服务器的连接已释放（但服务端到客户端的连接还未释放，并且客户端还可以接收数据）；
4. 服务端继续发送之前没发完的数据给客户端；
5. 服务端发送 FIN+ACK 给客户端，说明服务端发送完了数据（请求释放从服务端到客户端的连接，就算没收到客户端的回复，过段时间也会自动释放）；
6. 客户端收到服务端的 FIN+ACK，并回复 ACK 给客户端（同意释放从服务端到客户端的连接）；
7. 服务端收到客户端的 ACK 后，释放从服务端到客户端的连接。

TCP 为什么要进行四次挥手？

【问题一】TCP 为什么要进行四次挥手？ / 为什么 TCP 建立连接需要三次，而释放连接则需要四次？

【答案一】因为 TCP 是全双工模式，客户端请求关闭连接后，客户端向服务端的连接关闭（一二次挥手），服务端继续传输之前没传完的数据给客户端（数据传输），服务端向客户端的连接关闭（三四次挥手）。所以 TCP 释放连接时服务器的 ACK 和 FIN 是分开发送的（中间隔着数据传输），而 TCP 建立连接时服务器的 ACK 和 SYN 是一起发送的（第二次握手），所以 TCP 建立连接需要三次，而释放连接则需要四次。

【问题二】为什么 TCP 连接时可以 ACK 和 SYN 一起发送，而释放时则 ACK 和 FIN 分开发送呢？（ACK 和 FIN 分开是指第二次和第三次挥手）

【答案二】因为客户端请求释放时，服务器可能还有数据需要传输给客户端，因此服务端要先响应客户端 FIN 请求（服务端发送 ACK），然后数据传输，传输完成后，服务端再提出 FIN 请求（服务端发送 FIN）；而连接时则没有中间的数据传输，因此连接时可以 ACK 和 SYN 一起发送。

【问题三】为什么客户端释放最后需要 TIME-WAIT 等待 2MSL 呢？

【答案三】

1. 为了保证客户端发送的最后一个 ACK 报文能够到达服务端。若未成功到达，则服务端超时重传 FIN+ACK 报文段，客户端再重传 ACK，并重新计时。
2. 防止已失效的连接请求报文段出现在本连接中。TIME-WAIT 持续 2MSL 可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，这样可使下次连接中不会出现旧的连接报文段。

TCP 有限状态机

TCP 有限状态机图片

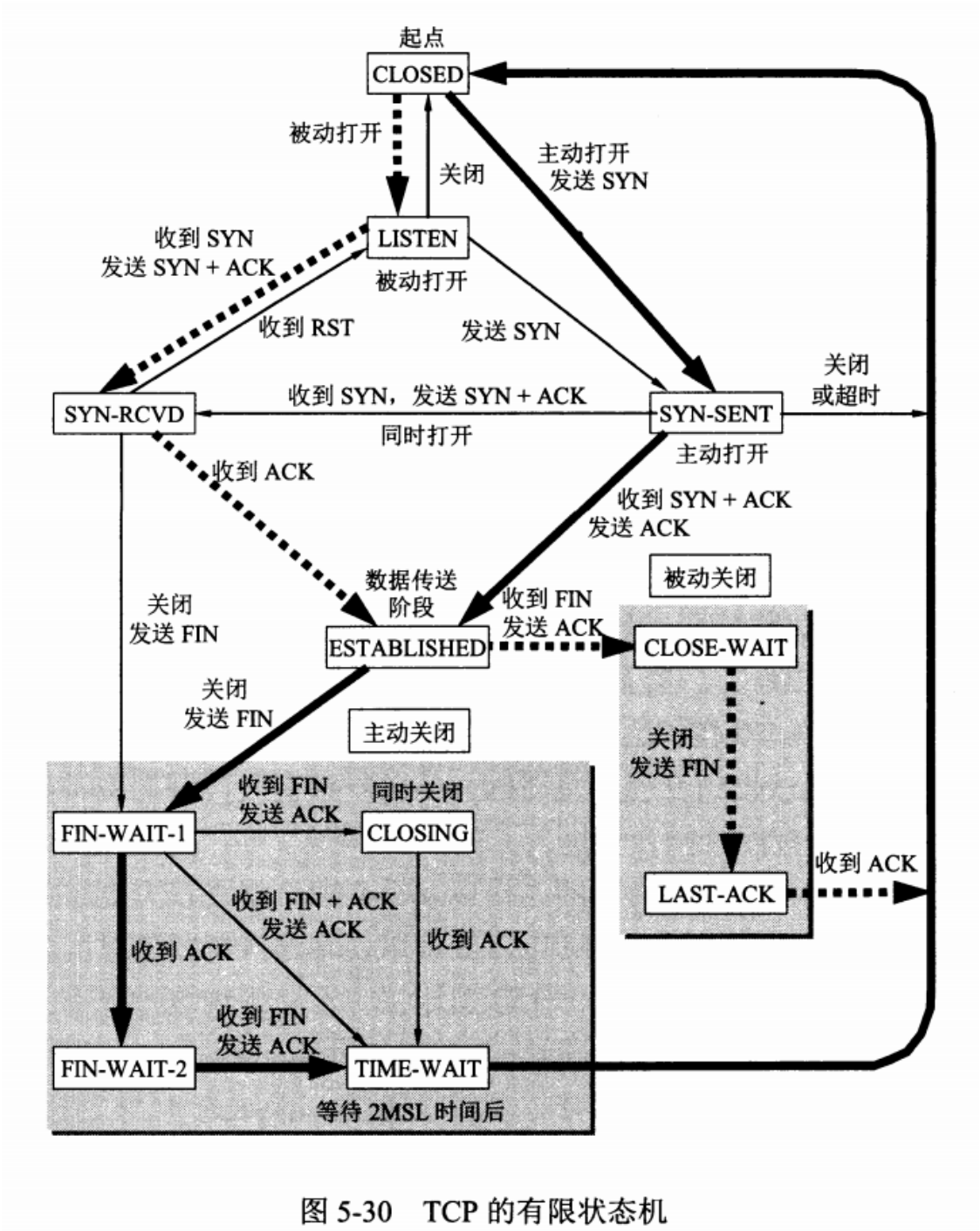


图 5-30　TCP 的有限状态机

TCP 的有限状态机

应用层

DNS

- DNS（Domain Name System，域名系统）是互联网的一项服务。它作为将域名和 IP 地址相互映射的一个分布式数据库，能够使人更方便地访问互联网。DNS 使用 TCP 和 UDP 端口 53。当前，对于每一级域名长度的限制是 63 个字符，域名总长度则不能超过 253 个字符。

域名：*域名 ::= {<三级域名>.<二级域名>.<顶级域名>}，如：blog.huihut.com

FTP

- FTP（File Transfer Protocol，文件传输协议）是用于在网络上进行文件传输的一套标准协议，使用客户/服务器模式，使用 TCP 数据报，提供交互式访问，双向传输。
- TFTP（Trivial File Transfer Protocol，简单文件传输协议）一个小且易实现的文件传输协议，也使用客户-服务器方式，使用 UDP 数据报，只支持文件传输而不支持交互，没有列目录，不能对用户进行身份鉴定

TELNET

- TELNET 协议是 TCP/IP 协议族中的一员，是 Internet 远程登陆服务的标准协议和主要方式。它为用户提供了在本地计算机上完成远程主机工作的能力。
- HTTP（HyperText Transfer Protocol，超文本传输协议）是用于从 WWW（World Wide Web，万维网）服务器传输超文本到本地浏览器的传送协议。
- SMTP（Simple Mail Transfer Protocol，简单邮件传输协议）是一组用于由源地址到目的地址传送邮件的规则，由它来控制信件的中转方式。SMTP 协议属于 TCP/IP 协议簇，它帮助每台计算机在发送或中转信件时找到下一个目的地。
- Socket 建立网络通信连接至少要一对端口号（Socket）。Socket 本质是编程接口（API），对 TCP/IP 的封装，TCP/IP 也要提供可供程序员做网络开发所用的接口，这就是 Socket 编程接口。

WWW

- WWW（World Wide Web，环球信息网，万维网）是一个由许多互相链接的超文本组成的系统，通过互联网访问

URL

- URL（Uniform Resource Locator，统一资源定位符）是因特网上标准的资源的地址（Address）

标准格式：

- 协议类型:[//服务器地址[:端口号]][/资源层级 UNIX 文件路径]文件名[?查询][#片段 ID]

完整格式：

- 协议类型:[//[访问资源需要的凭证信息@]服务器地址[:端口号]][/资源层级 UNIX 文件路径]文件名[?查询][#片段 ID]

其中【访问凭证信息@；:端口号；?查询；#片段 ID】都属于选填项

如：https://github.com/huihut/interview#cc

HTTP

HTTP（HyperText Transfer Protocol，超文本传输协议）是一种用于分布式、协作式和超媒体信息系统的应用层协议。HTTP 是万维网的数据通信的基础。

请求方法

方法	意义
OPTIONS	请求一些选项信息，允许客户端查看服务器的性能
GET	请求指定的页面信息，并返回实体主体
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
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
PUT	从客户端向服务器传送的数据取代指定的文档的内容
DELETE	请求服务器删除指定的页面
TRACE	回显服务器收到的请求，主要用于测试或诊断

状态码（Status-Code）

- 1xx：表示通知信息，如请求收到了或正在进行处理
  - 100 Continue：继续，客户端应继续其请求
  - 101 Switching Protocols 切换协议。服务器根据客户端的请求切换协议。只能切换到更高级的协议，例如，切换到 HTTP 的新版本协议
- 2xx：表示成功，如接收或知道了
  - 200 OK: 请求成功
- 3xx：表示重定向，如要完成请求还必须采取进一步的行动
  - 301 Moved Permanently: 永久移动。请求的资源已被永久的移动到新 URL，返回信息会包括新的 URL，浏览器会自动定向到新 URL。今后任何新的请求都应使用新的 URL 代替
- 4xx：表示客户的差错，如请求中有错误的语法或不能完成
  - 400 Bad Request: 客户端请求的语法错误，服务器无法理解
  - 401 Unauthorized: 请求要求用户的身份认证
  - 403 Forbidden: 服务器理解请求客户端的请求，但是拒绝执行此请求（权限不够）
  - 404 Not Found: 服务器无法根据客户端的请求找到资源（网页）。通过此代码，网站设计人员可设置“您所请求的资源无法找到”的个性页面
  - 408 Request Timeout: 服务器等待客户端发送的请求时间过长，超时
- 5xx：表示服务器的差错，如服务器失效无法完成请求
  - 500 Internal Server Error: 服务器内部错误，无法完成请求
  - 503 Service Unavailable: 由于超载或系统维护，服务器暂时的无法处理客户端的请求。延时的长度可包含在服务器的 Retry-After 头信息中
  - 504 Gateway Timeout: 充当网关或代理的服务器，未及时从远端服务器获取请求

更多状态码：菜鸟教程 . HTTP 状态码

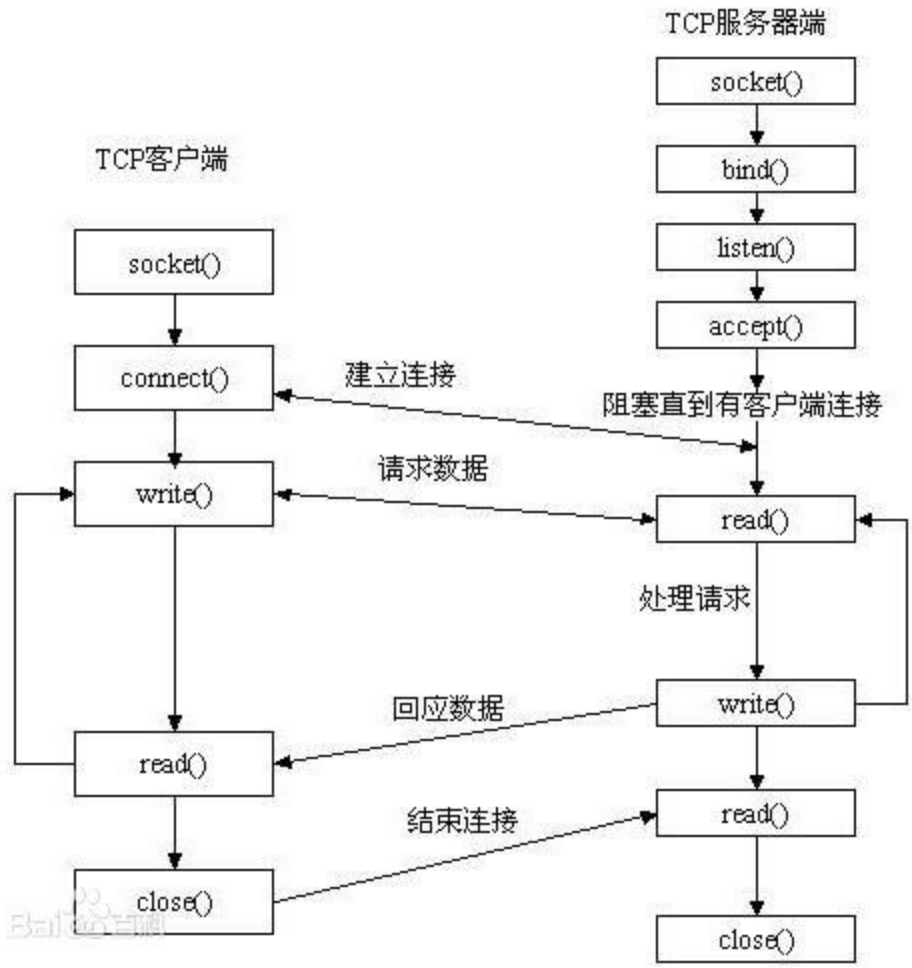
其他协议

- SMTP（Simple Main Transfer Protocol，简单邮件传输协议）是在 Internet 传输 Email 的标准，是一个相对简单的基于文本的协议。在其之上指定了一条消息的一个或多个接收者（在大多数情况下被确认是存在的），然后消息文本会被传输。可以很简单地通过 Telnet 程序来测试一个 SMTP 服务器。SMTP 使用 TCP 端口 25。
- DHCP（Dynamic Host Configuration Protocol，动态主机设置协议）是一个局域网的网络协议，使用 UDP 协议工作，主要有两个用途：
  - 用于内部网络或网络服务供应商自动分配 IP 地址给用户
  - 用于内部网络管理员作为对所有电脑作中央管理的手段
- SNMP（Simple Network Management Protocol，简单网络管理协议）构成了互联网工程工作小组（IETF，Internet Engineering Task Force）定义的 Internet 协议族的一部分。该协议能够支持网络管理系统，用以监测连接到网络上的设备是否有任何引起管理上关注的情况。

网络编程

Socket

Linux Socket 编程（不限 Linux）



Socket 客户端服务器通讯

Socket 中的 read()、write() 函数

```
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
```

read()

- read 函数是负责从 fd 中读取内容。
- 当读成功时，read 返回实际所读的字节数。
- 如果返回的值是 0 表示已经读到文件的结束了，小于 0 表示出现了错误。
- 如果错误为 EINTR 说明读是由中断引起的；如果是 ECONNREST 表示网络连接出了问题。

write()

- write 函数将 buf 中的 nbytes 字节内容写入文件描述符 fd。
- 成功时返回写的字节数。失败时返回 -1，并设置 errno 变量。
- 在网络程序中，当我们向套接字文件描述符写时有俩种可能。
- （1）write 的返回值大于 0，表示写了部分或者是全部的数据。
- （2）返回的值小于 0，此时出现了错误。
- 如果错误为 EINTR 表示在写的时候出现了中断错误；如果为 EPIPE 表示网络连接出现了问题（对方已经关闭了连接）。

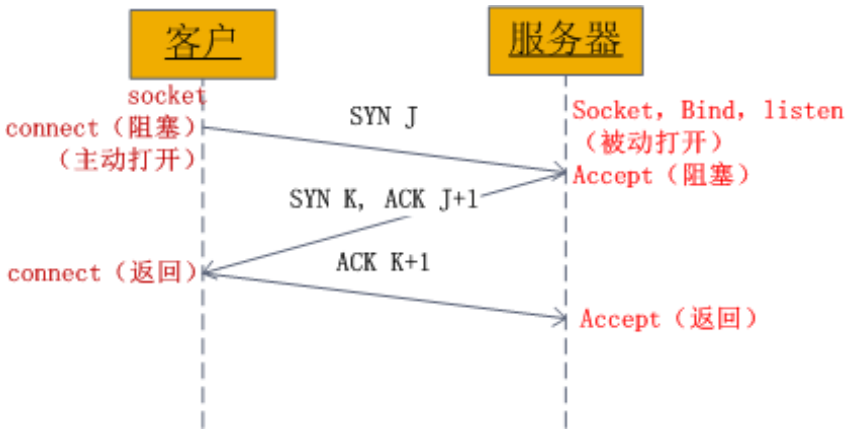
Socket 中 TCP 的三次握手建立连接

我们知道 TCP 建立连接要进行 “三次握手”，即交换三个分组。大致流程如下：

1. 客户端向服务器发送一个 SYN J
2. 服务器向客户端响应一个 SYN K，并对 SYN J 进行确认 ACK J+1

3. 客户端再想服务器发一个确认 ACK K+1

只有就完了三次握手，但是这个三次握手发生在 Socket 的那几个函数中呢？请看下图：



socket 中发送的 TCP 三次握手

从图中可以看出： 1. 当客户端调用 connect 时，触发了连接请求，向服务器发送了 SYN J 包，这时 connect 进入阻塞状态；

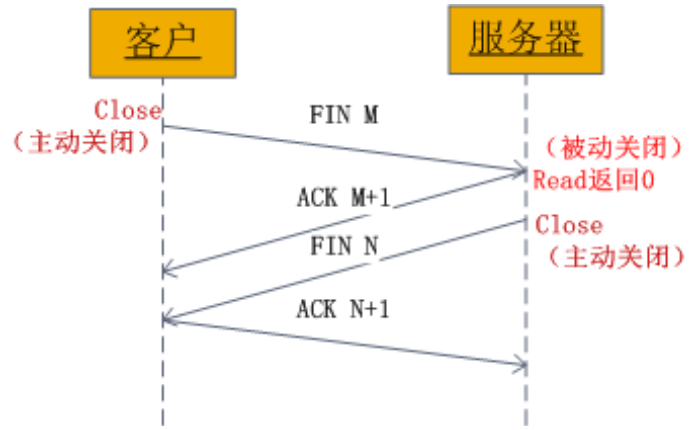
2. 服务器监听到连接请求，即收到 SYN J 包，调用 accept 函数接收请求向客户端发送 SYN K，ACK J+1，这时 accept 进入阻塞状态；

3. 客户端收到服务器的 SYN K，ACK J+1 之后，这时 connect 返回，并对 SYN K 进行确认；

4. 服务器收到 ACK K+1 时，accept 返回，至此三次握手完毕，连接建立。

Socket 中 TCP 的四次握手释放连接

上面介绍了 socket 中 TCP 的三次握手建立过程，及其涉及的 socket 函数。现在我们介绍 socket 中的四次握手释放连接的过程，请看下图：



socket 中发送的 TCP 四次握手

图示过程如下：

1. 某个应用进程首先调用 close 主动关闭连接，这时 TCP 发送一个 FIN M；
2. 另一端接收到 FIN M 之后，执行被动关闭，对这个 FIN 进行确认。它的接收也作为文件结束符传递给应用进程，因为 FIN 的接收意味着应用进程在相应的连接上再也接收不到额外数据；
3. 一段时间之后，接收到文件结束符的应用进程调用 close 关闭它的 socket。这导致它的 TCP 也发送一个 FIN N；
4. 接收到这个 FIN 的源发送端 TCP 对它进行确认。

这样每个方向上都有一个 FIN 和 ACK。

数据库

本节部分知识点来自《数据库系统概论（第 5 版）》

基本概念

- 数据（data）：描述事物的符号记录称为数据。

- 数据库（DataBase，DB）：是长期存储在计算机内、有组织的、可共享的大量数据的集合，具有永久存储、有组织、可共享三个基本特点。
- 数据库管理系统（DataBase Management System，DBMS）：是位于用户与操作系统之间的一层数据管理软件。
- 数据库系统（DataBase System，DBS）：是有数据库、数据库管理系统（及其应用开发工具）、应用程序和数据库管理员（DataBase Administrator DBA）组成的存储、管理、处理和维护数据的系统。
- 实体（entity）：客观存在并可相互区别的事物称为实体。
- 属性（attribute）：实体所具有的某一特性称为属性。
- 码（key）：唯一标识实体的属性集称为码。
- 实体型（entity type）：用实体名及其属性名集合来抽象和刻画同类实体，称为实体型。
- 实体集（entity set）：同一实体型的集合称为实体集。
- 联系（relationship）：实体之间的联系通常是指不同实体集之间的联系。
- 模式（schema）：模式也称逻辑模式，是数据库全体数据的逻辑结构和特征的描述，是所有用户的公共数据视图。
- 外模式（external schema）：外模式也称子模式（subschema）或用户模式，它是数据库用户（包括应用程序员和最终用户）能够看见和使用的局部数据的逻辑结构和特征的描述，是数据库用户的数据视图，是与某一应用有关的数据的逻辑表示。
- 内模式（internal schema）：内模式也称为存储模式（storage schema），一个数据库只有一个内模式。他是数据物理结构和存储方式的描述，是数据库在数据库内部的组织方式。

常用数据模型

- 层次模型（hierarchical model）
- 网状模型（network model）
- 关系模型（relational model）
  - 关系（relation）：一个关系对应通常说的一张表
  - 元组（tuple）：表中的一行即为一个元组
  - 属性（attribute）：表中的一列即为一个属性
  - 码（key）：表中可以唯一确定一个元组的某个属性组
  - 域（domain）：一组具有相同数据类型的值的集合
  - 分量：元组中的一个属性值
  - 关系模式：对关系的描述，一般表示为 关系名(属性 1，属性 2，...，属性 n)
- 面向对象数据模型（object oriented data model）
- 对象关系数据模型（object relational data model）
- 半结构化数据模型（semistructure data model）

常用 SQL 操作

对象类型
对象
操作类型
数据库模式
模式
CREATE SCHEMA
基本表
CREATE SCHEMA，ALTER TABLE
视图
CREATE VIEW
索引
CREATE INDEX
数据

基本表和视图
SELECT，INSERT，UPDATE，DELETE，REFERENCES，ALL PRIVILEGES
属性列
SELECT，INSERT，UPDATE，REFERENCES，ALL PRIVILEGES
SQL 语法教程：runoob . SQL 教程

关系型数据库

- 基本关系操作：查询（选择、投影、连接（等值连接、自然连接、外连接（左外连接、右外连接））、除、并、差、交、笛卡尔积等）、插入、删除、修改
- 关系模型中的三类完整性约束：实体完整性、参照完整性、用户定义的完整性

索引

- 数据库索引：顺序索引、B+ 树索引、hash 索引
- MySQL 索引背后的数据结构及算法原理

数据库完整性

- 数据库的完整性是指数据的正确性和相容性。
  - 完整性：为了防止数据库中存在不符合语义（不正确）的数据。
  - 安全性：为了保护数据库防止恶意破坏和非法存取。
- 触发器：是用户定义在关系表中的一类由事件驱动的特殊过程。

关系数据理论

- 数据依赖是一个关系内部属性与属性之间的一种约束关系，是通过属性间值的相等与否体现出来的数据间相关联系。
- 最重要的数据依赖：函数依赖、多值依赖。

范式

- 第一范式（1NF）：属性（字段）是最小单位不可再分。
- 第二范式（2NF）：满足 1NF，每个非主属性完全依赖于主键（消除 1NF 非主属性对码的部分函数依赖）。
- 第三范式（3NF）：满足 2NF，任何非主属性不依赖于其他非主属性（消除 2NF 主属性对码的传递函数依赖）。
- 鲍依斯-科得范式（BCNF）：满足 3NF，任何非主属性不能对主键子集依赖（消除 3NF 主属性对码的部分和传递函数依赖）。
- 第四范式（4NF）：满足 3NF，属性之间不能有非平凡且非函数依赖的多值依赖（消除 3NF 非平凡且非函数依赖的多值依赖）。

数据库恢复

- 事务：是用户定义的一个数据库操作序列，这些操作要么全做，要么全不做，是一个不可分割的工作单位。
- 事物的 ACID 特性：原子性、一致性、隔离性、持续性。
- 恢复的实现技术：建立冗余数据 -> 利用冗余数据实施数据库恢复。
- 建立冗余数据常用技术：数据转储（动态海量转储、动态增量转储、静态海量转储、静态增量转储）、登记日志文件。

并发控制

- 事务是并发控制的基本单位。
- 并发操作带来的数据不一致性包括：丢失修改、不可重复读、读“脏”数据。
- 并发控制主要技术：封锁、时间戳、乐观控制法、多版本并发控制等。
- 基本封锁类型：排他锁（X 锁 / 写锁）、共享锁（S 锁 / 读锁）。
- 活锁死锁：
  - 活锁：事务永远处于等待状态，可通过先来先服务的策略避免。
  - 死锁：事物永远不能结束
    - 预防：一次封锁法、顺序封锁法；
    - 诊断：超时法、等待图法；

- 解除：撤销处理死锁代价最小的事务，并释放此事务的所有的锁，使其他事务得以继续运行下去。

- 可串行化调度：多个事务的并发执行是正确的，当且仅当其结果与按某一次序串行地执行这些事务时的结果相同。可串行性时并发事务正确调度的准则。

设计模式

各大设计模式例子参考：CSDN 专栏 . C++ 设计模式 系列博文

设计模式工程目录

单例模式

单例模式例子

抽象工厂模式

抽象工厂模式例子

适配器模式

适配器模式例子

桥接模式

桥接模式例子

观察者模式

观察者模式例子

设计模式的六大原则

- 单一职责原则（SRP，Single Responsibility Principle）
- 里氏替换原则（LSP，Liskov Substitution Principle）
- 依赖倒置原则（DIP，Dependence Inversion Principle）
- 接口隔离原则（ISP，Interface Segregation Principle）
- 迪米特法则（LoD，Law of Demeter）
- 开放封闭原则（OCP，Open Close Principle）

链接装载库

本节部分知识点来自《程序员的自我修养——链接装载库》

内存、栈、堆

一般应用程序内存空间有如下区域：

- 栈：由操作系统自动分配释放，存放函数的参数值、局部变量等的值，用于维护函数调用的上下文
- 堆：一般由程序员分配释放，若程序员不释放，程序结束时可能由操作系统回收，用来容纳应用程序动态分配的内存区域
- 可执行文件映像：存储着可执行文件在内存中的映像，由装载器装载是将可执行文件的内存读取或映射到这里
- 保留区：保留区并不是一个单一的内存区域，而是对内存中受到保护而禁止访问的内存区域的总称，如通常 C 语言讲无效指针赋值为 0（NULL），因此 0 地址正常情况下不可能有效的访问数据

栈

栈保存了一个函数调用所需要的维护信息，常被称为堆栈帧（Stack Frame）或活动记录（Activate Record），一般包含以下几方面：

- 函数的返回地址和参数
- 临时变量：包括函数的非静态局部变量以及编译器自动生成的其他临时变量
- 保存上下文：包括函数调用前后需要保持不变的寄存器

堆

堆分配算法：

- 空闲链表（Free List）
- 位图（Bitmap）
- 对象池

“段错误（segment fault）”或“非法操作，该内存地址不能 read/write”

典型的非法指针解引用造成的错误。当指针指向一个不允许读写的内存地址，而程序却试图利用指针来读或写该地址时，会出现这个错误。

普遍原因：

- 将指针初始化为 NULL，之后没有给它一个合理的值就开始使用指针
- 没用初始化栈中的指针，指针的值一般会是随机数，之后就直接开始使用指针

编译链接

各平台文件格式

平台	可执行文件	目标文件	动态库/共享对象	静态库
Windows	exe	obj	dll	lib
Unix/Linux	ELF、out	o	so	a
Mac	Mach-O	o	dylib、tbd、framework	a、framework

编译链接过程

1. 预编译（预编译器处理如 `#include`、`#define` 等预编译指令，生成 `.i` 或 `.ii` 文件）
2. 编译（编译器进行词法分析、语法分析、语义分析、中间代码生成、目标代码生成、优化，生成 `.s` 文件）
3. 汇编（汇编器把汇编码翻译成机器码，生成 `.o` 文件）
4. 链接（连接器进行地址和空间分配、符号决议、重定位，生成 `.out` 文件）

现在版本 GCC 把预编译和编译合成一步，预编译编译程序 cc1、汇编器 as、连接器 ld

MSVC 编译环境，编译器 cl、连接器 link、可执行文件查看器 dumpbin

目标文件

编译器编译源代码后生成的文件叫做目标文件。目标文件从结构上讲，它是已经编译后的可执行文件格式，只是还没有经过链接的过程，其中可能有些符号或有些地址还没有被调整。

可执行文件（Windows 的 `.exe` 和 Linux 的 `ELF`）、动态链接库（Windows 的 `.dll` 和 Linux 的 `.so`）、静态链接库（Windows 的 `.lib` 和 Linux 的 `.a`）都是按照可执行文件格式存储（Windows 按照 PE-COFF，Linux 按照 ELF）

目标文件格式

- Windows 的 PE（Portable Executable），或称为 PE-COFF，`.obj` 格式
- Linux 的 ELF（Executable Linkable Format），`.o` 格式
- Intel/Microsoft 的 OMF（Object Module Format）
- Unix 的 `a.out` 格式
- MS-DOS 的 `.COM` 格式

PE 和 ELF 都是 COFF（Common File Format）的变种

目标文件存储结构

段	功能
File Header	文件头，描述整个文件的文件属性（包括文件是否可执行、是静态链接或动态连接及入口地址、目标硬件、目标操作系统等）
.text section	代码段，执行语句编译成的机器代码
.data section	数据段，已初始化的全局变量和局部静态变量
.bss section	BSS 段（Block Started by Symbol），未初始化的全局变量和局部静态变量（因为默认值为 0，所以只是在此预留位置，不占空间）
.rodata section	只读数据段，存放只读数据，一般是程序里面的只读变量（如 const 修饰的变量）和字符串常量
.comment section	注释信息段，存放编译器版本信息
.note.GNU-stack section	堆栈提示段

其他段略

链接的接口————符号

在链接中，目标文件之间相互拼合实际上是目标文件之间对地址的引用，即对函数和变量的地址的引用。我们将函数和变量统称为符号（Symbol），函数名或变量名就是符号名（Symbol Name）。

如下符号表（Symbol Table）：

Symbol（符号名）	Symbol Value （地址）
main	0x100
Add	0x123
...	...

Linux 的共享库（Shared Library）

Linux 下的共享库就是普通的 ELF 共享对象。

共享库版本更新应该保证二进制接口 ABI（Application Binary Interface）的兼容

命名

`libname.so.x.y.z`

- x：主版本号，不同主版本号的库之间不兼容，需要重新编译
- y：次版本号，高版本号向后兼容低版本号
- z：发布版本号，不对接口进行更改，完全兼容

路径

大部分包括 Linux 在内的开源系统遵循 FHS（File Hierarchy Standard）的标准，这标准规定了系统文件如何存放，包括各个目录结构、组织和作用。

- `/lib`：存放系统最关键和最基础的共享库，如动态链接器、C 语言运行库、数学库等
- `/usr/lib`：存放非系统运行时所需要的关键性的库，主要是开发库
- `/usr/local/lib`：存放跟操作系统本身并不十分相关的库，主要是一些第三方应用程序的库

动态链接器会在 `/lib`、`/usr/lib` 和由 `/etc/ld.so.conf` 配置文件指定的，目录中查找共享库

环境变量

- `LD_LIBRARY_PATH`：临时改变某个应用程序的共享库查找路径，而不会影响其他应用程序
- `LD_PRELOAD`：指定预先装载的一些共享库甚至是目标文件
- `LD_DEBUG`：打开动态链接器的调试功能

so 共享库的编写

使用 CLion 编写共享库

创建一个名为 MySharedLib 的共享库

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.10)
project(MySharedLib)

set(CMAKE_CXX_STANDARD 11)

add_library(MySharedLib SHARED library.cpp library.h)
```

library.h

```
#ifndef MYSHAREDLIB_LIBRARY_H
#define MYSHAREDLIB_LIBRARY_H

// 打印 Hello World!
void hello();

// 使用可变模版参数求和
template <typename T>
T sum(T t)
{
    return t;
}
template <typename T, typename ...Types>
T sum(T first, Types ... rest)
{
    return first + sum<T>(rest...);
}
```

```
#endif
```
library.cpp
```
#include <iostream>
#include "library.h"
void hello() {
    std::cout << "Hello, World!" << std::endl;
}
```
so 共享库的使用（被可执行项目调用）
使用 CLion 调用共享库
创建一个名为 TestSharedLib 的可执行项目
CMakeLists.txt
```
cmake_minimum_required(VERSION 3.10)
project(TestSharedLib)

# C++11 编译
set(CMAKE_CXX_STANDARD 11)

# 头文件路径
set(INC_DIR /home/xx/code/clion/MySharedLib)
# 库文件路径
set(LIB_DIR /home/xx/code/clion/MySharedLib/cmake-build-debug)

include_directories(${INC_DIR})
link_directories(${LIB_DIR})
link_libraries(MySharedLib)

add_executable(TestSharedLib main.cpp)

# 链接 MySharedLib 库
target_link_libraries(TestSharedLib MySharedLib)
```
main.cpp
```
#include <iostream>
#include "library.h"
using std::cout;
using std::endl;
int main() {

    hello();
    cout << "1 + 2 = " << sum(1,2) << endl;
    cout << "1 + 2 + 3 = " << sum(1,2,3) << endl;

    return 0;
}
```
执行结果
```
Hello, World!
1 + 2 = 3
1 + 2 + 3 = 6
```
Windows 应用程序入口函数
- GUI（Graphical User Interface）应用，链接器选项：/SUBSYSTEM:WINDOWS
- CUI（Console User Interface）应用，链接器选项：/SUBSYSTEM:CONSOLE

_tWinMain 与 _tmain 函数声明
```
Int WINAPI _tWinMain(
    HINSTANCE hInstanceExe,
```

```
    HINSTANCE,
    PTSTR pszCmdLine,
    int nCmdShow);

int _tmain(
    int argc,
    TCHAR *argv[],
    TCHAR *envp[]);
```

应用程序类型	入口点函数	嵌入可执行文件的启动函数
处理 ANSI 字符（串）的 GUI 应用程序	_tWinMain(WinMain)	WinMainCRTSartup
处理 Unicode 字符（串）的 GUI 应用程序	_tWinMain(wWinMain)	wWinMainCRTSartup
处理 ANSI 字符（串）的 CUI 应用程序	_tmain(Main)	mainCRTSartup
处理 Unicode 字符（串）的 CUI 应用程序	_tmain(wMain)	wmainCRTSartup
动态链接库（Dynamic-Link Library）	DllMain	_DllMainCRTStartup

Windows 的动态链接库（Dynamic-Link Library）

部分知识点来自《Windows 核心编程（第五版）》

用处

- 扩展了应用程序的特性
- 简化了项目管理
- 有助于节省内存
- 促进了资源的共享
- 促进了本地化
- 有助于解决平台间的差异
- 可以用于特殊目的

注意

- 创建 DLL，事实上是在创建可供一个可执行模块调用的函数
- 当一个模块提供一个内存分配函数（malloc、new）的时候，它必须同时提供另一个内存释放函数（free、delete）
- 在使用 C 和 C++ 混编的时候，要使用 extern “C” 修饰符
- 一个 DLL 可以导出函数、变量（避免导出）、C++ 类（导出导入需要同编译器，否则避免导出）
- DLL 模块：cpp 文件中的 _declspec(dllexport) 写在 include 头文件之前
- 调用 DLL 的可执行模块：cpp 文件的 _declspec(dllimport) 之前不应该定义 MYLIBAPI

加载 Windows 程序的搜索顺序

1. 包含可执行文件的目录
2. Windows 的系统目录，可以通过 GetSystemDirectory 得到
3. 16 位的系统目录，即 Windows 目录中的 System 子目录
4. Windows 目录，可以通过 GetWindowsDirectory 得到
5. 进程的当前目录
6. PATH 环境变量中所列出的目录

DLL 入口函数

DllMain 函数

```
BOOL WINAPI DllMain(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpvReserved)
{
    switch(fdwReason)
    {
    case DLL_PROCESS_ATTACH:
        // 第一次将一个DLL 映射到进程地址空间时调用
        // The DLL is being mapped into the process' address space.
        break;
    case DLL_THREAD_ATTACH:
```

```
        // 当进程创建一个线程的时候，用于告诉DLL 执行与线程相关的初始化（非主线程执行）
        // A thread is bing created.
        break;
    case DLL_THREAD_DETACH:
        // 系统调用 ExitThread 线程退出前，即将终止的线程通过告诉DLL 执行与线程相关的清理
        // A thread is exiting cleanly.
        break;
    case DLL_PROCESS_DETACH:
        // 将一个DLL 从进程的地址空间时调用
        // The DLL is being unmapped from the process' address space.
        break;
    }
    return (TRUE); // Used only for DLL_PROCESS_ATTACH
}
```

载入卸载库

LoadLibrary、LoadLibraryExA、LoadPackagedLibrary、FreeLibrary、FreeLibraryAndExitThread 函数声明

```
// 载入库
HMODULE WINAPI LoadLibrary(
  _In_ LPCTSTR lpFileName
);
HMODULE LoadLibraryExA(
  LPCSTR lpLibFileName,
  HANDLE hFile,
  DWORD  dwFlags
);
// 若要在通用 Windows 平台（UWP）应用中加载 Win32 DLL，需要调用 LoadPackagedLibrary，而不是 LoadLibrary 或 LoadLibraryEx
HMODULE LoadPackagedLibrary(
  LPCWSTR lpwLibFileName,
  DWORD   Reserved
);

// 卸载库
BOOL WINAPI FreeLibrary(
  _In_ HMODULE hModule
);
// 卸载库和退出线程
VOID WINAPI FreeLibraryAndExitThread(
  _In_ HMODULE hModule,
  _In_ DWORD   dwExitCode
);
```

显示地链接到导出符号

GetProcAddress 函数声明

```
FARPROC GetProcAddress(
  HMODULE hInstDll,
  PCSTR pszSymbolName  // 只能接受 ANSI 字符串，不能是 Unicode
);
```

DumpBin.exe 查看 DLL 信息

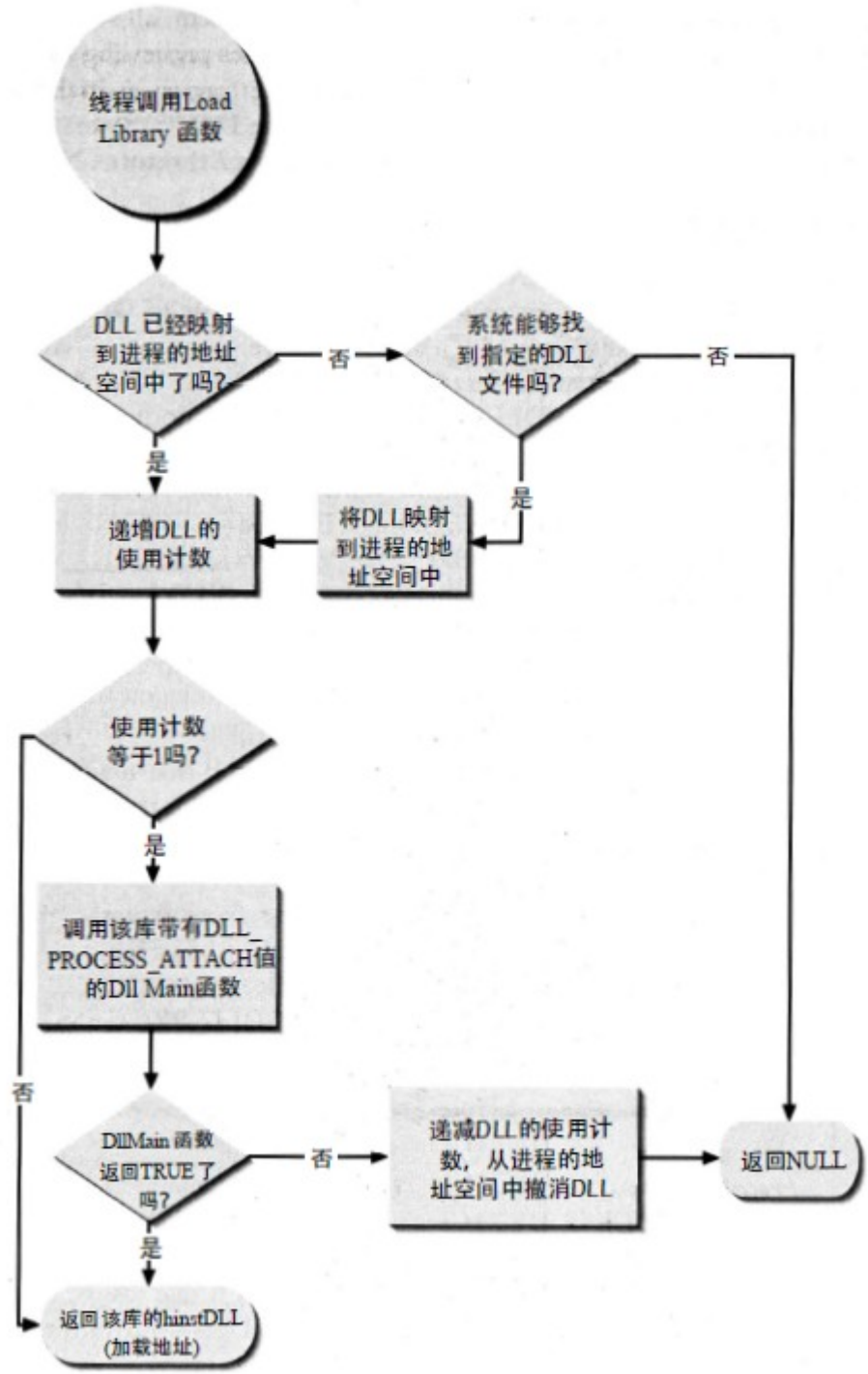
在 VS 的开发人员命令提示符 使用 DumpBin.exe 可查看 DLL 库的导出段（导出的变量、函数、类名的符号）、相对虚拟地址（RVA，relative virtual address）。如：

```
DUMPBIN -exports D:\mydll.dll
```

LoadLibrary 与 FreeLibrary 流程图

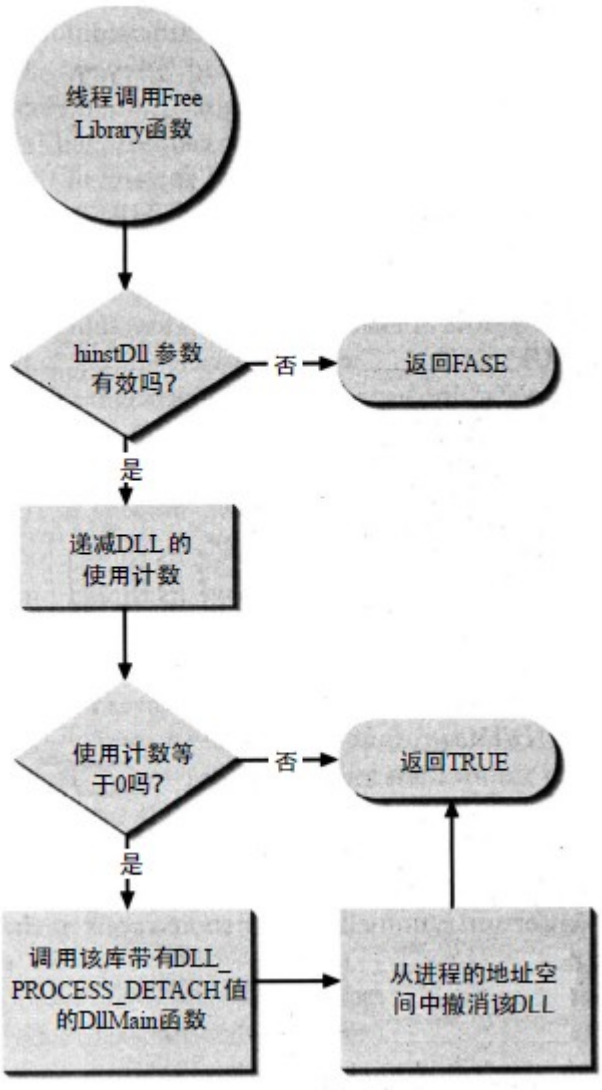
LoadLibrary 与 FreeLibrary 流程图

LoadLibrary



WindowsLoadLibrary

FreeLibrary



WindowsFreeLibrary

DLL 库的编写（导出一个 DLL 模块）

DLL 库的编写（导出一个 DLL 模块） DLL 头文件

```
// MyLib.h
#ifdef MYLIBAPI
// MYLIBAPI 应该在全部 DLL 源文件的 include "Mylib.h" 之前被定义
// 全部函数/变量正在被导出

#else

// 这个头文件被一个exe 源代码模块包含，意味着全部函数/变量被导入
#define MYLIBAPI extern "C" __declspec(dllimport)

#endif

// 这里定义任何的数据结构和符号
```

```
// 定义导出的变量（避免导出变量）
MYLIBAPI int g_nResult;

// 定义导出函数原型
MYLIBAPI int Add(int nLeft, int nRight);
```

DLL 源文件

```
// MyLibFile1.cpp

// 包含标准Windows 和C 运行时头文件
#include <windows.h>
// DLL 源码文件导出的函数和变量
#define MYLIBAPI extern "C" __declspec(dllexport)
// 包含导出的数据结构、符号、函数、变量
#include "MyLib.h"
// 将此DLL 源代码文件的代码放在此处
int g_nResult;

int Add(int nLeft, int nRight)
{
    g_nResult = nLeft + nRight;
    return g_nResult;
}
```

DLL 库的使用（运行时动态链接 DLL）

DLL 库的使用（运行时动态链接 DLL）

```
// A simple program that uses LoadLibrary and
// GetProcAddress to access myPuts from Myputs.dll.

#include <windows.h>
#include <stdio.h>

typedef int (__cdecl *MYPROC)(LPWSTR);

int main( void )
{
    HINSTANCE hinstLib;
    MYPROC ProcAdd;
    BOOL fFreeResult, fRunTimeLinkSuccess = FALSE;
    // Get a handle to the DLL module.
    hinstLib = LoadLibrary(TEXT("MyPuts.dll"));
    // If the handle is valid, try to get the function address.
    if (hinstLib != NULL)
    {
        ProcAdd = (MYPROC) GetProcAddress(hinstLib, "myPuts");

        // If the function address is valid, call the function.

        if (NULL != ProcAdd)
        {
            fRunTimeLinkSuccess = TRUE;
            (ProcAdd) (L"Message sent to the DLL function\n");
        }
        // Free the DLL module.
        fFreeResult = FreeLibrary(hinstLib);
    }
```

```
    // If unable to call the DLL function, use an alternative.
    if (! fRunTimeLinkSuccess)
        printf("Message printed from executable\n");
    return 0;
}
```

运行库（Runtime Library）

典型程序运行步骤

1. 操作系统创建进程，把控制权交给程序的入口（往往是运行库中的某个入口函数）
2. 入口函数对运行库和程序运行环境进行初始化（包括堆、I/O、线程、全局变量构造等等）。
3. 入口函数初始化后，调用 main 函数，正式开始执行程序主体部分。
4. main 函数执行完毕后，返回到入口函数进行清理工作（包括全局变量析构、堆销毁、关闭 I/O 等），然后进行系统调用结束进程。

一个程序的 I/O 指代程序与外界的交互，包括文件、管程、网络、命令行、信号等。更广义地讲，I/O 指代操作系统理解为“文件”的事物。

glibc 入口

```
_start -> __libc_start_main -> exit -> _exit
```

其中 `main(argc, argv, __environ)` 函数在 `__libc_start_main` 里执行。

MSVC CRT 入口

```
int mainCRTStartup(void)
```

执行如下操作：

1. 初始化和 OS 版本有关的全局变量。
2. 初始化堆。
3. 初始化 I/O。
4. 获取命令行参数和环境变量。
5. 初始化 C 库的一些数据。
6. 调用 main 并记录返回值。
7. 检查错误并将 main 的返回值返回。

C 语言运行库（CRT）

大致包含如下功能：

- 启动与退出：包括入口函数及入口函数所依赖的其他函数等。
- 标准函数：有 C 语言标准规定的 C 语言标准库所拥有的函数实现。
- I/O：I/O 功能的封装和实现。
- 堆：堆的封装和实现。
- 语言实现：语言中一些特殊功能的实现。
- 调试：实现调试功能的代码。

C 语言标准库（ANSI C）

包含：

- 标准输入输出（stdio.h）
- 文件操作（stdio.h）
- 字符操作（ctype.h）
- 字符串操作（string.h）
- 数学函数（math.h）
- 资源管理（stdlib.h）
- 格式转换（stdlib.h）
- 时间/日期（time.h）
- 断言（assert.h）
- 各种类型上的常数（limits.h & float.h）
- 变长参数（stdarg.h）
- 非局部跳转（setjmp.h）

书籍

huihut/CS-Books：Computer Science Books 计算机技术类书籍

语言

- 《C++ Primer》
- 《Effective C++》
- 《More Effective C++》
- 《深度探索 C++ 对象模型》
- 《深入理解 C++11》
- 《STL 源码剖析》

算法

- 《剑指 Offer》
- 《编程珠玑》
- 《程序员面试宝典》

系统

- 《深入理解计算机系统》
- 《Windows 核心编程》
- 《Unix 环境高级编程》

网络

- 《Unix 网络编程》
- 《TCP/IP 详解》

其他

- 《程序员的自我修养》

C/C++ 发展方向

C/C++ 发展方向甚广，包括不限于以下方向， 以下列举一些大厂校招岗位要求。

后台/服务器

【后台开发】

- 编程基本功扎实，掌握 C/C++/JAVA 等开发语言、常用算法和数据结构；
- 熟悉 TCP/UDP 网络协议及相关编程、进程间通讯编程；
- 了解 Python、Shell、Perl 等脚本语言；
- 了解 MYSQL 及 SQL 语言、编程，了解 NoSQL, key-value 存储原理；
- 全面、扎实的软件知识结构，掌握操作系统、软件工程、设计模式、数据结构、数据库系统、网络安全等专业知识；
- 了解分布式系统设计与开发、负载均衡技术，系统容灾设计，高可用系统等知识。

桌面客户端

【PC 客户端开发】

- 计算机软件相关专业本科或以上学历，热爱编程，基础扎实，理解算法和数据结构相关知识；
- 熟悉 windows 操作系统的内存管理、文件系统、进程线程调度；
- 熟悉 MFC/windows 界面实现机制，熟练使用 VC，精通 C/C++，熟练使用 STL，以及 Windows 下网络编程经验；
- 熟练掌握 Windows 客户端开发、调试，有 Windows 应用软件开发经验优先；
- 对于创新及解决具有挑战性的问题充满激情，具有良好的算法基础及系统分析能力。

图形学/游戏/VR/AR

【游戏客户端开发】

- 计算机科学/工程相关专业本科或以上学历，热爱编程，基础扎实，理解算法、数据结构、软件设计相关知识；
- 至少掌握一种游戏开发常用的编程语言，具 C++/C# 编程经验优先；
- 具游戏引擎（如 Unity、Unreal）使用经验者优先；
- 了解某方面的游戏客户端技术（如图形、音频、动画、物理、人工智能、网络同步）优先；

- 对于创新及解决具有挑战性的问题充满激情，有较强的学习能力、分析及解决问题能力，具备良好的团队合作意识；
- 具阅读英文技术文档能力；
- 热爱游戏。

测试开发

【测试开发】

- 计算机或相关专业本科及以上学历；
- 一至两年的 C/C++/Python 或其他计算机语言的编程经验；
- 具备撰写测试计划、测试用例、以及实现性能和安全等测试的能力；
- 具备实现自动化系统的能力；
- 具备定位调查产品缺陷能力、以及代码级别调试缺陷的能力；
- 工作主动积极，有责任心，具有良好的团队合作精神。

网络安全/逆向

【安全技术】

- 热爱互联网，对操作系统和网络安全有狂热的追求，专业不限；
- 熟悉漏洞挖掘、网络安全攻防技术，了解常见黑客攻击手法；
- 掌握基本开发能力，熟练使用 C/C++ 语言；
- 对数据库、操作系统、网络原理有较好掌握；
- 具有软件逆向，网络安全攻防或安全系统开发经验者优先。

嵌入式/物联网

【嵌入式应用开发】

- 有良好的编程基础，熟练掌握 C/C++ 语言；
- 掌握操作系统、数据结构等软件开发必备知识；
- 具备较强的沟通理解能力及良好的团队合作意识；
- 有 Linux/Android 系统平台的开发经验者优先。

音视频/流媒体/SDK

【音视频编解码】

- 硕士及以上学历，计算机、信号处理、数学、信息类及相关专业和方向；
- 视频编解码基础扎实，熟常用的 HEVC 或 H264，有较好的数字信号处理基础；
- 掌握 C/C++，代码能力强, 熟悉一种汇编语言尤佳；
- 较强的英文文献阅读能力；
- 学习能力强，具有团队协作精神，有较强的抗压能力。

计算机视觉/机器学习

【计算机视觉研究】

- 计算机、应用数学、模式识别、人工智能、自控、统计学、运筹学、生物信息、物理学/量子计算、神经科学、社会学/心理学等专业，图像处理、模式识别、机器学习相关研究方向，本科及以上，博士优先；
- 熟练掌握计算机视觉和图像处理相关的基本算法及应用；
- 较强的算法实现能力，熟练掌握 C/C++ 编程，熟悉 Shell/Python/Matlab 至少一种编程语言；
- 在计算机视觉、模式识别等学术会议或者期刊上发表论文、相关国际比赛获奖、及有相关专利者优先。