目录

- C/C++
- STL
- 数据结构
- 算法
- Problems
- 操作系统
- 计算机网络
- 网络编程
- 数据库
- 设计模式
- 链接装载库
- 海量数据处理
- 音视频
- 其他

C/C++

const

作用

- 修饰变量，说明该变量不可以被改变；
- 修饰指针，分为指向常量的指针和指针常量；
- 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；
- 修饰成员函数，说明该成员函数内不能修改成员变量。

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(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
int* const function7();     // 返回一个指向变量的常指针，使用：int* const p =
```

static

作用

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。
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，为了防止与他人命令函数重名，可以将函数定位为 static。
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。

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

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。它指向正在被该成员函数操作的那个对象。
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，由隐含使用 this 指针。
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。
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）；
this 并不是一个常规变量，而是个右值，所以不能取得 this 的地址（不能 &this ）。
在以下场景中，经常需要显式引用 this 指针：

为实现对象的链式引用；
为避免对同一对象进行赋值操作；
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；
不能包含循环、递归、switch 等复杂操作；
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

```
// 声明1（加 inline，建议使用）
inline int functionName(int first, int secend,...);

// 声明2（不加 inline）
int functionName(int first, int secend,...);

// 定义
inline int functionName(int first, int secend,...) {/****/};

// 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

将 inline 函数体复制到 inline 函数调用点处；
为所用 inline 函数中的局部变量分配内存空间；
将 inline 函数的的输入参数和返回值映射到调用方法的局部变量空间中；
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

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。
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。
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。
内联函数在运行时可调试，而宏定义不可以。

缺点

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。
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。
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"?
> 答案：http://www.cs.technion.ac.il/users/yechiel/c++-faq/inline-virtuals.html

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
    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
    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
    delete ptr;
    ptr = nullptr;

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

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。

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

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
```

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

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

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 语言方式编译和连接的

`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。

"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 Studen

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

C++ 中 struct 和 class

总的来说，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

默认的继承访问权限。struct 是 public 的，class 是 private 的。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

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：
> http://dongxicheng.org/cpp/ooc/

explicit（显式）构造函数

explicit 修饰的构造函数可用来防止隐式转换

explicit 使用

```
class Test1
{
public:
    Test1(int n)            // 普通构造函数
    {
        num=n;
    }
private:
    int num;
};

class Test2
{
public:
    explicit Test2(int n)   // explicit（显式）构造函数
```

```
    {
        num=n;
    }
private:
    int num;
};

int main()
{
    Test1 t1=12;            // 隐式调用其构造函数，成功
    Test2 t2=12;            // 编译错误，不能隐式调用其构造函数
    Test2 t2(12);           // 显式调用成功
    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【C++11】

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**只导入了制定的名称**。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

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
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
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

```
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 )
```

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)>::typ
{
    // 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到右值（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：
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
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
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【C++11】

用花括号初始化器列表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.

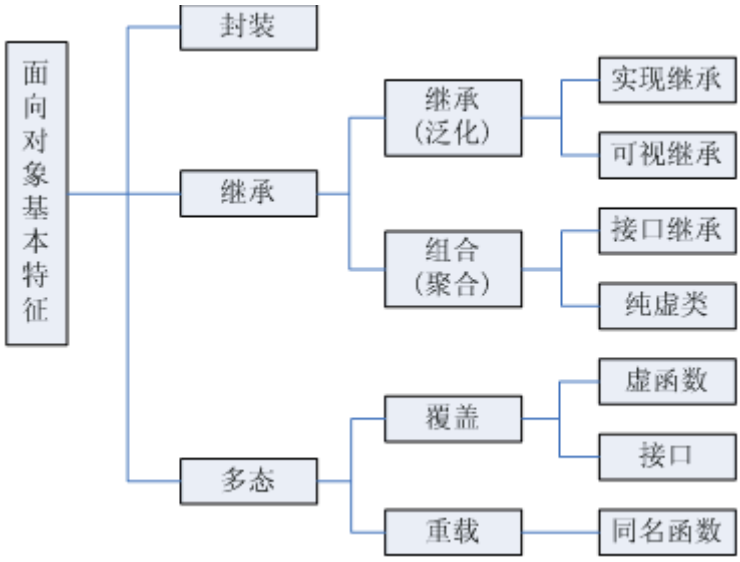
initializer_list 使用

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

```cpp
#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 {
        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, friendly, private。不写默认为 friendly。

关键字	当前类	包内	子孙类
public	√	√	√
protected	√	√	√
friendly	√	√	×
private	√	×	×

继承

基类（父类）——> 派生类（子类）

多态

多态，即多种状态，在面向对象语言中，接口的多种不同的实现方式即为多态。
C++ 多态有两种：静态多态（早绑定）、动态多态（晚绑定）。静态多态是通过函数重载实现的；动态多态是通过虚函数实现的。
多态是以封装和继承为基础的。

静态多态（早绑定）

函数重载

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

动态多态（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？：http://t.cn/E4WVXSP

动态多态使用

```
class Shape                     // 形状类
{
public:
    virtual double calcArea()
    {
        ...
    }
```

```
    virtual ~Shape();
};
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
    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

> CSDN . C++ 中的虚函数、纯虚函数区别和联系：http://t.cn/E4WVQBI

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。

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:http://t.cn/E4WVBeF），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。

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：
- 虚函数不占用存储空间
- 虚函数表存储的是虚函数地址
- 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
- 虚继承
- 虚函数

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
- 没有有定于任何构造函数
- 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

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。
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。
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。
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

> new / new[]：完成两件事，先底层调用 malloc 分了配内存，然后调用构造函数（创建对象）。
> 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。
> 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 传递额外的参数。

```
new (palce_address) type
new (palce_address) type (initializers)
new (palce_address) type [size]
new (palce_address) type [size] { braced initializer list }
```

> `palce_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?
> 答案：http://t.cn/E4Wfcfl

合法，但：

- 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
- 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
- 必须保证成员函数的 `delete this` 后面没有调用 this 了
- 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？
> 答案：http://t.cn/E4WfDhP

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

> shared_ptr
> unique_ptr
> weak_ptr
> 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：http://t.cn/E4WIt5W

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 B

    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++

视 C++ 为一个语言联邦（C、Object-Oriented C++、Template C++、STL）
宁可以编译器替换预处理器（尽量以 `const` 、 `enum` 、 `inline` 替换 `#define` ）
尽可能使用 const
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）
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）
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）
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）
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）

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）
令 `operator=` 返回一个 `reference to *this` （用于连锁赋值）
在 `operator=` 中处理“自我赋值”
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）
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））
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））
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）
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`）
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）
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）
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。
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）
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。）
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）
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）
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
考虑写一个不抛异常的 swap 函数
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）
尽量少做转型动作（旧式：`(T)expression`、`T(expression)`；新式：`const_cast(expression)`、`dynamic_cast(expression)`、`reinterpret_cast(expression)`、`static_cast(expression)`、；尽量避

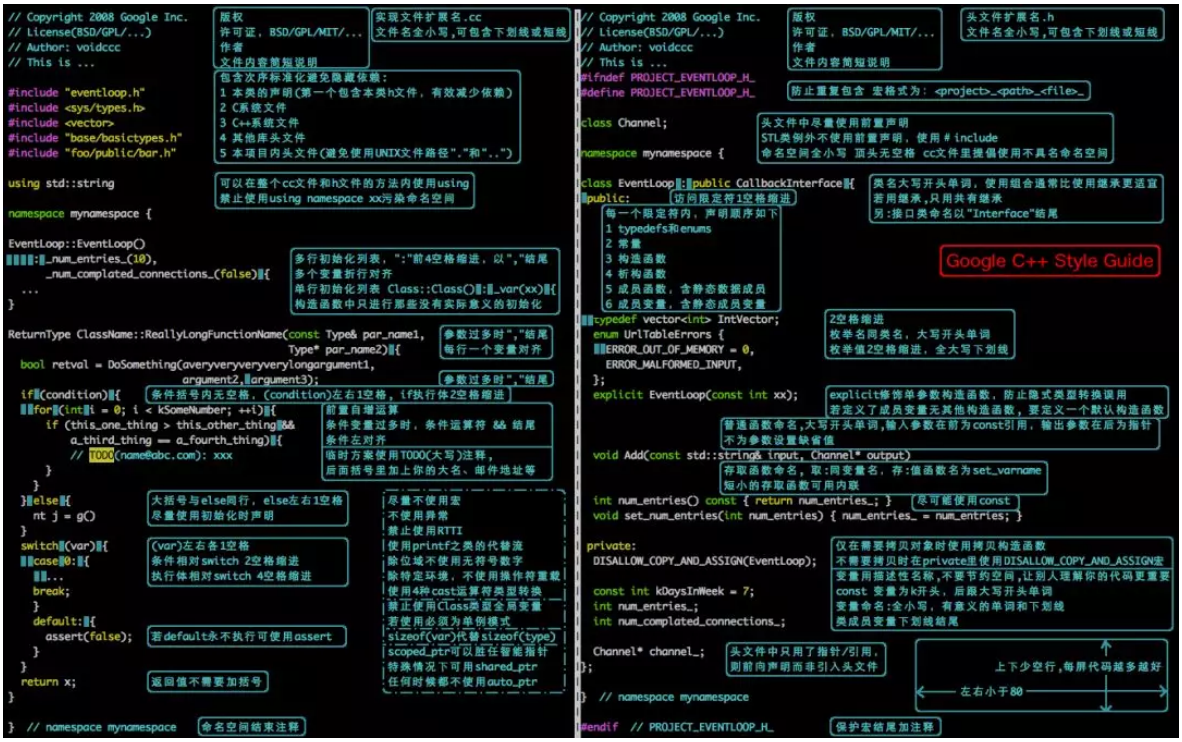
免转型、注重效率避免 dynamic_casts、尽量设计成无需转型、可把转型封装成函数、宁可用新式转型）
避免使用 handles（包括 引用、指针、迭代器）指向对象内部（以增加封装性、使 const 成员函数的行为更像 const、降低“虚吊号码牌”（dangling handles，如悬空指针等）的可能性）
为“异常安全”而努力是值得的（异常安全函数（Exception-safe functions）即使发生异常也不会泄露资源或允许任何数据结构败坏，分为三种可能的保证：基本型、强列型、不抛异常型）
透彻了解 inlining 的里里外外（inlining 在大多数 C++ 程序中是编译期的行为；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）
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）
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）
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）
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）
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）
绝不重新定义继承而来的 non-virtual 函数
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）
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））
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）

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）
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）
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 basee class 修饰符）
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）
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）
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）
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”）
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）
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）
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）

Google C++ Style Guide

> 英文：Google C++ Style Guide ：http://t.cn/RqhluJP
>
> 中文：C++ 风格指南：http://t.cn/ELDTnur
>
> Google C++ Style Guide 图



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

STL

STL 索引

STL 方法含义索引：http://t.cn/E4WMXXs

STL 容器

容器的详细说明：http://t.cn/E4WMXXs

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
vector	数组	随机读改、尾部插入、尾部删除 O(1) 头部插入、头部删除 O(n)	无序	可重复	支持快速
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
deque	双端队列	头尾插入、头尾删除 O(1)	无序	可重复	一个中央 缓冲区， 增删，支
stack	deque / list	顶部插入、顶部删除 O(1)	无序	可重复	deque 或 开口，不 因应该是 制，扩容
queue	deque / list	尾部插入、头部删除 O(1)	无序	可重复	deque 或 开口，不 因应该是 制，扩容
priority_queue	vector + max-heap	插入、删除 O(log2n)	有序	可重复	vector容器 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	

容器	底层数据结构	时间复杂度	有无序	可不可重复	其他
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

http://t.cn/aEv0DV

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

数据结构

顺序结构

顺序栈（Sequence Stack）

SqStack.cpp：http://t.cn/E4WxO0b

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

非空二叉树第 i 层最多 $2^{(i-1)}$ 个结点 （i >= 1）

深度为 k 的二叉树最多 2^k - 1 个结点 （k >= 1）

度为 0 的结点数为 n_0，度为 2 的结点数为 n_2，则 $n_0 = n_2 + 1$

有 n 个结点的完全二叉树深度 $k = \lfloor \log_2(n) \rfloor + 1$

对于含 n 个结点的完全二叉树中编号为 i （1 <= i <= n） 的结点

- 若 i = 1，为根，否则双亲为 ⌊ i / 2 ⌋
- 若 2i > n，则 i 结点没有左孩子，否则孩子编号为 2i
- 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？

> 节点是红色或黑色。
> 根是黑色。
> 所有叶子都是黑色（叶子是 NIL 节点）。
> 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）
> 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

> 变色
> 左旋

右旋

应用

关联数组：如 STL 中的 map、set

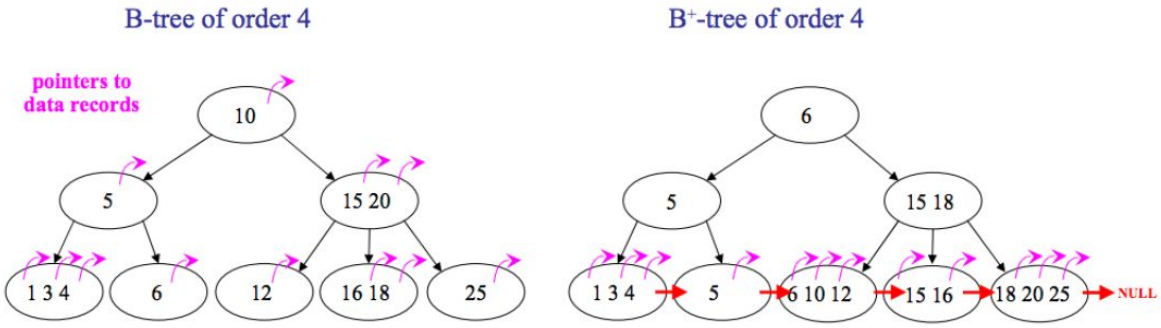
红黑树、B 树、B+ 树的区别?

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

- A B⁺-tree can be viewed as a B-tree in which each node contains only keys (not pairs), and to which an additional level is added at the bottom with linked leaves



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。

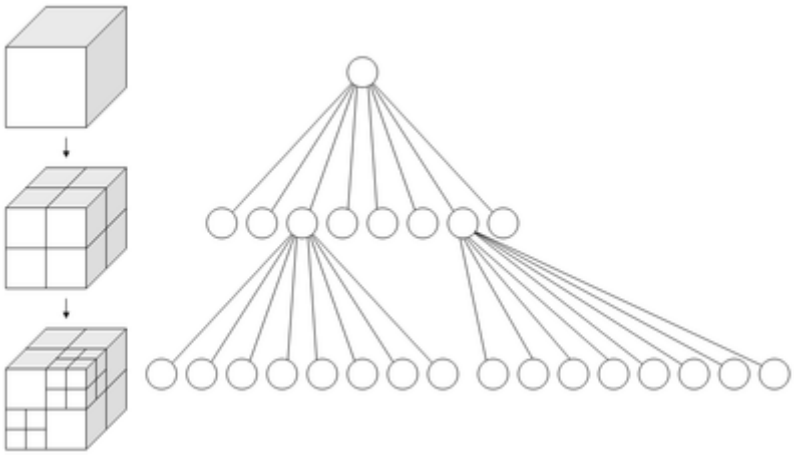
B+树的优点

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：
> http://t.cn/RrBAaZa
> http://t.cn/E4WJhmZ

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

http://t.cn/E4WJUGz

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 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
堆排序	O(n*log2n)	O(n*log2n)	O(1)	不稳
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
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
顺序查找	O(n)	O(1)	无序或
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
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
BFS广度优先搜索	邻接矩阵 邻接链表	O(|v|2) O(|v|+|E|)	O(|v O(|v
DFS深度优先搜索	邻接矩阵 邻接链表	O(|v|2) O(|v|+|E|)	O(|v O(|v

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	循环赛日程安排算法（快速排序）

算法	思想	应用
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
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 问题）、最小 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）
- 优点：简单方便
- 缺点:
- 优点：可以实现任意关系的进程间的通信
- 缺点:
- 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
- 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）
- 局限于单向通信
- 只能创建在它的进程以及其有亲缘关系的进程之间
- 缓冲区有限
- 长期存于系统中，使用不当容易出错
- 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：

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

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：

缺点：需对传输的数据进行解析，转化成应用级的数据。

传输数据为字节级，传输数据可自定义，数据量小效率高

传输数据时间短，性能高

适合于客户端和服务器端之间信息实时交互

可以加密,数据安全性强

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着是否已经释放锁。

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
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
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
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

对比维度	多进程	多线程	点
编程、调试	编程简单，调试简单	编程复杂，调试复杂	适
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	适
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	适

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 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

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	0x0
大端	12	34	56	78
小端	78	56	34	12

大端小端图片

大端序

小端序

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

计算机经网络体系结构：

计算机经网络体系结构

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 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、VLA 桥，交换机）

分层	作用	协议
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 X、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、R
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、SM 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
双向交替通行（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
B 类	16bit，前两位固定为 10	128.0 —— 191.255	16bit	128.0.0.0 —— 191.255.2
C 类	24bit，前三位固定为 110	192.0.0 —— 223.255.255	8bit	192.0.0.0 —— 223.255.2
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			

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?

网络 ID（Network ID, Network number）：就是目标地址的网络 ID。
子网掩码（subnet mask）：用来判断 IP 所属网络
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：

花费（Cost）：就是数据发送过程中通过路径所需要的花费。
路由的服务质量
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	SI
端口号	21	23	25	53	69	80	443	16

受限于公众号文章字数限制，后续部分请看**【今天的第二篇推文】**，

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 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

TCP 与 UDP 的区别

TCP 面向连接，UDP 是无连接的；
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
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
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
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）
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）
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

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 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】

客户端发送 SYN 给服务器，说明客户端请求建立连接；
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）；
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）；
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 释放连接全过程解释】

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）；
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）；
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）；
服务端继续发送之前没发完的数据给客户端；
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）；
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）；
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】

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。

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。

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

方法	意义
OPTIONS	请求一些选项信息，允许客户端查看服务器的
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 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 上传文件）。数据被包含在请求体中。POST 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
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

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()

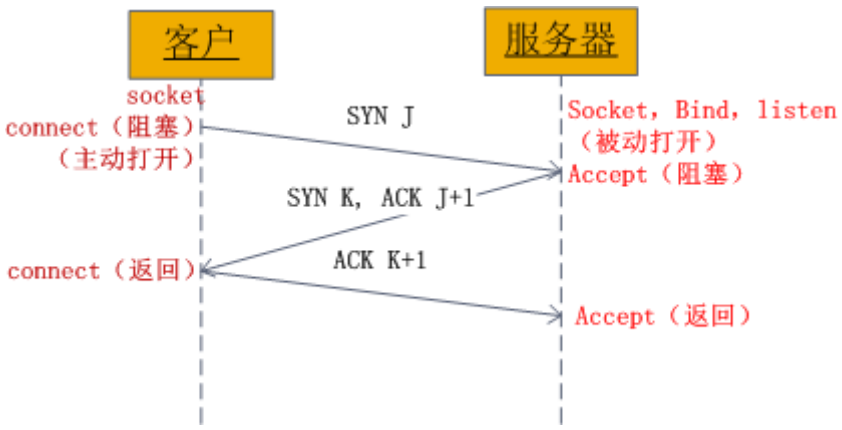
write 函数将 buf 中的 nbytes 字节内容写入文件描述符 fd。
成功时返回写的字节数。失败时返回 -1，并设置 errno 变量。
在网络程序中，当我们向套接字文件描述符写时有俩种可能。
（1）write 的返回值大于 0，表示写了部分或者是全部的数据。
（2）返回的值小于 0，此时出现了错误。
如果错误为 EINTR 表示在写的时候出现了中断错误；如果为 EPIPE 表示网络连接出现了问题（对方已经关闭了连接）。

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：

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



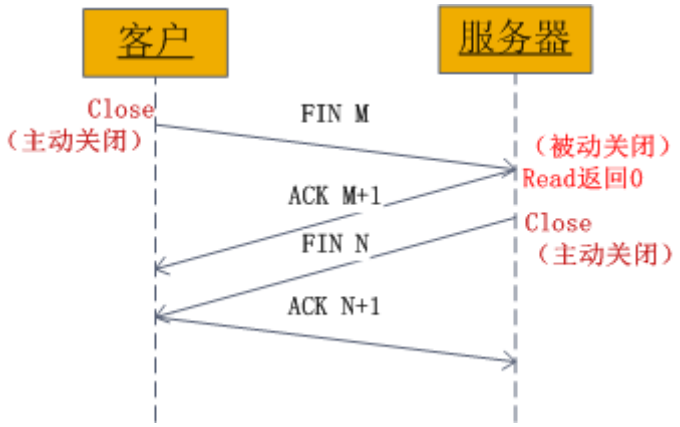
socket 中发送的 TCP 三次握手

从图中可以看出：

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；
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；
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；
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：

> 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；
> 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；
> 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；
> 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

> 数据库事务四大特性：原子性、一致性、分离性、持久性
> 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
> SQL 约束 (Constraints)

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）

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

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、frame

编译链接过程

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）
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）
汇编（汇编器把汇编码翻译成机器码，生成 `.o` 文件）
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（包括文件是静态链接或动态连接及入口地址、目标硬件统等）

段	功能
.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 和局部静态变量（因为默认值为 0，所以只 置，不占空间）
.rodata section	只读数据段，存放只读数据，一般是程序里 （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
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

```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 动函数
处理ANSI字符（串）的GUI应用程序	_tWinMain(WinMain)	WinMainCRTSa
处理Unicode字符（串）的GUI应用程序	_tWinMain(wWinMain)	wWinMainCRTS
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)	wmainCRTSart
动态链接库（Dynamic-Link Library）	DllMain	_DllMainCRTSt

Windows 的动态链接库（Dynamic-Link Library）

> 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

包含可执行文件的目录
Windows 的系统目录，可以通过 GetSystemDirectory 得到
16 位的系统目录，即 Windows 目录中的 System 子目录
Windows 目录，可以通过 GetWindowsDirectory 得到
进程的当前目录
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
        // 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
        // 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

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 LoadPackagedLil
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
```

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）

```c
// 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

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）
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）。

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。
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：

初始化和 OS 版本有关的全局变量。
初始化堆。
初始化 I/O。
获取命令行参数和环境变量。
初始化 C 库的一些数据。
调用 main 并记录返回值。
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）