目　录

致谢

当前文档 《后台开发核心知识》 由 进击的皇虫 使用 书栈(BookStack.CN) 进行构建，生成于 2018-04-10。

书栈(BookStack.CN) 仅提供文档编写、整理、归类等功能，以及对文档内容的生成和导出工具。

文档内容由网友们编写和整理，书栈(BookStack.CN) 难以确认文档内容知识点是否错漏。如果您在阅读文档获取知识的时候，发现文档内容有不恰当的地方，请向我们反馈，让我们共同携手，将知识准确、高效且有效地传递给每一个人。

同时，如果您在日常生活、工作和学习中遇到有价值有营养的知识文档，欢迎分享到 书栈(BookStack.CN) ，为知识的传承献上您的一份力量！

如果当前文档生成时间太久，请到 书栈(BookStack.CN) 获取最新的文档，以跟上知识更新换代的步伐。

文档地址：http://www.bookstack.cn/books/Skill-Tree

书栈官网：http://www.bookstack.cn

书栈开源：https://github.com/TruthHun

分享，让知识传承更久远！ 感谢知识的创造者，感谢知识的分享者，也感谢每一位阅读到此处的读者，因为我们都将成为知识的传承者。

后台开发核心知识

后台开发核心知识

> 准备秋招，欢迎来树上取果实。

看过很多书，但总是忘得很快。知识广度越大越容易接纳新东西，但从考察角度来说，自然是对某个方面了解越深越好。那些大而全的著作虽然每本都是经典中的经典，但实际工作中可能只用到其中的一小部分。之前实习经历使我对后台开发有了更深刻的认知和了解，现在距离秋招只有两个月了，这里将以最短的篇幅，最清晰的层级结构去总结那些对C++后台开发最为核心的内容。

我现在越发觉得少即是多，看再多东西没有理解透彻都是白搭，把最常用的每天过一遍才是最有效的。开发中我们经常用缓存来提高吞吐率，学习知识何不也给自己加个Cache呢？

最后，希望大家秋招都能找到满意的工作。

快速索引

上面我们提到了Cache来缩小知识范围，但是即使是被压缩过的知识依旧很多，我们怎么能够在脑海中快速检索它们呢？结合查找算法，Hash无疑是最快的，但又有多少人能够给一个"key"立马对应上"value"呢？所以，最适合人类认知的方式是通过索引 + 树状结构，在整理这份笔记时，我划分了很多级索引用来将各部分知识点划分到相应的模块中，检索任意一个知识点最多5级深度，不仅检索速度上去了还可以对整个知识体系有宏观认识。

独立专题

哲学中，整体与个体的关系是物质世界普遍存在的规律。上面各部分知识相对独立，既要有零又要有整才能收获更多，实战无疑是最好的。

- 纸上代码
- 练手项目

其他

> 路过的豪杰麻烦右上角点一个*star !*

- 学习计划
- gitlab或github下fork后如何同步源的新更新内容？

来源(书栈小编注)

> *https://github.com/linw7/Skill-Tree*

错题精解

- Q 36 :
- Q 37 :
- Q 38 :
- Q 39 :
- Q 40 :
- Q 41 :
- Q 42 :
- Q 43 :
- Q 44 :
- Q 45 :
- Q 46 :
- Q 47 :
- Q 48 :
- Q 49 :
- Q 50 :
- Q 51 :
- Q 52 :
- Q 53 :
- Q 54 :
- Q 55 :
- Q 56 :
- Q 57 :
- Q 58 :
- Q 59 :
- Q 60 :
- Q 61 :
- Q 62 :
- Q 63 :
- Q 64 :
- Q 65 :
- Q 66 :
- Q 67 :
- Q 68 :
- Q 69 :
- Q 70 :
- Q 71 :
- Q 72 :
- Q 73 :
- Q 74 :
- Q 75 :
- Q 76 :
- Q 77 :

错题精解

> 牛客网基础题总结。

目录

Chapter 1	Chapter 2
编程语言(C++)	其他

内容

编程语言(C++)" class="reference-link">

编程语言(C++)

Q 1 :

```
题目：
    以下代码：
    class ClassA{
    public:
        virtual ~ClassA(){};
        virtual void FunctionA(){};
    };
    class ClassB{
    public:
        virtual void FunctionB(){};
    };
    class ClassC:public ClassA, public ClassB{
        public:
    };
    ClassC Object;
    ClassA* pA = &Object;
    ClassB* pB = &Object;
    ClassC* pC = &Object;
    关于pA,pB,pC的取值,下面的描述中正确的是:

答案:
    pA和pB不相同

解答:
    考察多继承且有虚函数情况下C++存储对象模型。
    1. 多继承按继承顺序组织对象模型，有虚函数时低地址包含指向虚函数表的指针。
    2. 对象Object的存储模型：类A虚函数表指针(ptrA) | 类A数据 | 类B虚函数表指针(ptrB) | 类B数据 | 类C数据。
    3. 子类的虚函数被放到了第一个基类的虚函数表最后（ptrA指向的虚函数表结构：类A虚函数 | 类C虚函数）。
    4. 有虚函数的继承，对象地址为指向虚函数表的指针的地址，即pC = &Object = &ptrA。
    6. pC = pA = &ptrA = &Object, (pC = pA) < pB。
```

Q 2 :

```
题目：
    下列程序的输出结果：
    #include <iostream>
    using namespace std;
    class A{
    public:
        void print(){
            cout << "A:print()";
```

```
        }
    };
    class B:private A{
    public:
        void print(){
          cout << "B:print()";
        }
    };
    class C:public B{
    public:
       void print(){
            A::print();
        }
    };
    int main(){
        C b;
        b.print();
    }

答案:
    编译出错

解答:
    考察C++继承问题。
    1. 类B私有继承类A。
    2. 私有继承：类A的公有成员和保护成员都作为类B的私有成员，并且不能被类B的子类（如类C）所访问。
```

Q 3 :

```
题目：
    下面两个结构体：
    struct One{
        double d;
        char c;
        int i;
    }
    struct Two{
        char c;
        double d;
        int i;
    }
    在#pragma pack(4)和#pragma pack(8)的情况下，结构体的大小分别是：

答案:
    16, 16
    16, 24

```

```
解答：
    考察结构体对齐。
    1. 4字节对齐：Struct One[8 + (1 + 3(pading)) + 4], struct Two[(1 + 3(pading)) + 8 + 4]。
    2. 8字节对齐：Struct One[8 + (1 + 3(pading) + 4)], struct Two[(1 + 7(pading)) + 8 + (4 + 4(pading))]。
    3. 一句话总结：按序存储，装得下尽量装，装不下换一行。
```

Q 4 ：

```
题目：
    下列代码的输出为：
    #include "iostream"
    #include "vector"
    using namespace std;
    int main(void)
    {
        vector<int>array;
        array.push_back(100);
        array.push_back(300);
        array.push_back(300);
        array.push_back(500);
        vector<int>::iterator itor;
        for(itor = array.begin(); itor != array.end(); itor++){
            if(*itor == 300){
                itor = array.erase(itor);
            }
        }
        for(itor = array.begin(); itor != array.end(); itor++){
            cout << *itor << " ";
        }
        return 0;
    }

答案：
    100
    300
    500

解答：
    考察STL中erase和迭代器问题。
    1. erase返回值是一个迭代器，指向删除元素下一个元素。
    2. 删除第一个300时返回指向下一个300的迭代器，在循环体又被再加了一次，跳过了第二个300。
```

Q 5 ：

```
题目：
```

```
    下面程序的输出是什么？
    int main(void)
    {
        int a[5] = {1, 2, 3, 4, 5};
        int *ptr = (int *)(&a + 1);
        printf("%d,%d", *(a + 1), *(ptr - 1));
        return 0;
    }

答案：
    2
    5

解答：
    1. a表示数组首元素的地址，对a的所有操作均是以一个元素为单位的。
    2. &a表示整个数组的地址，对&a的所有操作均是以一个数组为单位的。
    3. ptr类型为int *，所有对ptr的所有操作均是以int大小为单位进行的。
    4. (int *)(&a + 1)表示指向a数组最后一个元素后一个字节的int类型指针，*(ptr - 1)表示向前移动一个int类型的数据的位置。
    5. 所有指针类型操作先看右侧是以什么为单位，之后再转换为左侧定义的单位。
```

Q 6* ：

```
题目：
    32位机上根据下面的代码，问哪些说法是正确的？
    signed char a = 0xe0;
    unsigned int b = a;
    unsigned char c = a;

答案：
    b的十六进制表示是：0xfffffffe0

解答：
    考察有符号数和无符号数之间的转换。
    1. a ： 1110 0000。
    2. 扩展问题：
        长 -> 短：低位对齐，按位复制。
        短 -> 长：符号位扩展。
    3. 精度提升：
        两个变量运算，表示范围小的变量精度达的变量提升（signed -> unsigned）。
```

Q 7 ：

```
题目：
    下列代码的输出为：
    int* pint = 0;
```

```
    pint += 6;
    cout << pint << endl;

答案:
    24

解答:
    考察指针运算。
    1. 变量pint为指向int类型的指针，这里“+1”表示地址加4（pint值加4）。
    2. 变量pint初值为0，pint + 6后pint的值变为24。
```

Q 8 ：

```
题目：
    如果两段内存重叠，用memcpy函数可能会导致行为未定义。而memmove函数能够避免这种问题，下面是一种实现方式，请补充代码。
    #include <iostream>
    using namespace std;
    void* memmove(void* str1, const void* str2, size_t n)
    {
        char* pStr1 = (char*) str1;
        const char* pStr2 = (const char*)str2;
        if( ){
            for(size_t i = 0;i != n; ++i){
                *(pStr1++) = *(pStr2++);
            }
        }
        else{
            pStr1 += n - 1;
            pStr2 += n - 1;
            for(size_t i = 0; i != n; ++i){
                *(pStr1--) = *(pStr2--);
            }
        }
        return ( );
    }

答案:
    pStr1 < pStr2
    str1

解答:
    1. 逐字符自动不存在内存覆盖问题。
```

Q 9 ：

```
题目：
    设x、y、t均为int型变量，则执行语句：t = 3; x = y = 2; t = x++ || ++y; 后，变量t和y的值分别为：

答案:
    t = 1
    y = 2

解答:
    考察逻辑短路和运算符优先级。
    1*. =的优先级最低，t = (x++ || ++y) = 1。
    2. x++和++y为或关系，因为x++的值非0，所以++y不执行，y不变。
```

Q 10 :

```
题目：
    指出下面程序哪里可能有问题？
    class CBuffer
    {
        char * m_pBuffer;
        int m_size;
    public:
        CBuffer(){
            m_pBuffer = NULL;
        }
        ~CBuffer(){
            Free();
        }
        void Allocte(int size) (1) {
            m_size = size;
            m_pBuffer = new char[size];
        }
    private:
        void Free(){
            if(m_pBuffer! = NULL) (2){
                delete[] m_pBuffer;
                m_pBuffer = NULL;
            }
        }
    public:
        void SaveString(const char* pText) const (3){
            strcpy(m_pBuffer, pText); (4)
        }
        char* GetBuffer() const{
            return m_pBuffer;
        }
    };
```

```
    void main (int argc, char* argv[])
    {
        CBuffer buffer1;
        buffer1.SaveString("Microsoft");
        printf(buffer1.GetBuffer());
    }

答案:
    1
    3
    4

解答:
    考察动态分配空间等周边细节处理。
    1. 分配内存时，未检测m_pBuffer是否为空，容易造成内存泄露。
    2. 常成员函数不应该对数据成员做出修改，虽然可以修改指针数据成员指向的数据，但原则上不应该这么做。
    3*. 字符串拷贝时，未检测是否有足够空间，可能造成程序崩溃。
```

Q 11 :

```
题目：
    某32位系统下，C++程序，请计算sizeof 的值：
    char str[] = "http://www.xxxxx.com";
    char *p = str;
    int n = 10;
    sizeof(str) = (1);
    sizeof(p) = (2;
    sizeof(n) = (3);
    void Foo(char str[100]){
        sizeof(str) = (4);
    }
    void *p = malloc(100);
    sizeof(p) = (5);

答案:
    21
    4
    4
    4
    4

解答:
    考察sizeof返回值。
    1. 具体类型，返回该类型所占的空间大小。
    2. 对象，返回对象的实际占用空间大小。
    3. 数组，返回编译时分配的数组空间大小（数组名 ≠ 指针）。作为参数时数组退化为指针。
    4. 指针，返回存储该指针所用的空间大小。
```

```
    5. 函数，返回函数的返回类型所占的空间大小。函数的返回类型不能是void。
    6. 上题中(2)(4)(5)均为指针。
```

Q 12* :

```
题目：
    在C++中，
    const int i = 0;
    int *j = (int *) &i;
    *j = 1;
    printf("%d, %d", i, *j);
    输出是多少？

答案：
    0
    1

解答：
    考察C++常量折叠。
    1. const变量放在编译器的符号表中，计算时编译器直接从表中取值，省去了访问内存的时间，从而达到了优化。
    2. 结论，const变量通过取地址方式可以修改该地址存储的数据值，但不能修改常量的值。
```

Q 13 :

```
题目：
    下列代码的输出为：
    class parent{
    public:
        virtual void output();
    };
    void parent::output(){
        printf("parent!");
    }
    class son : public parent{
    public:
        virtual void output();
    };
    void son::output(){
        printf("son!");
    }
    son s;
    memset(&s, 0, sizeof(s));
    parent& p = s;
    p.output();

答案：
```

```
    没有输出结果，程序运行出错。

解答:
    考察memset和虚函数指针。
    1. 虚函数表地址被清空。
```

Q 14 :

```
题目：
    有哪几种情况只能用intialization list而不能用assignment？

答案*:
    当类中含有const成员变量；基类无默认构造函数时，有参的构造函数都需要初始化表；当类中含有reference成员变量。

解答:
    1. 见答案。
```

Q 15 :

```
题目：
    对以下数据结构中data的处理方式描述正确的是：
    struct Node{
        int size;
        char data[0];
    };

答案:
    编译器会认为这就是一个长度为0的数组,而且会支持对于数组data的越界访问。

解答:
    考察柔性数组。
    1*. 柔性数组，作为占位符放在结构体末尾，使得结构体的大小动态可变，在声明结构体变量的时候可根据需要动态分配内存。
    2. 长度为0的数组并不占用空间，因为数组名本身不占空间，它只是一个偏移量， 数组名这个符号本身代表了一个不可修改的地址常量。
    3. 常用于网络通信中构造不定长数据包，不会浪费空间浪费网络流量。
```

Q 16 :

```
题目：
    给定3个int类型的正整数x, y, z, 对如下4组表达式判断正确的选项：
    int a1 = x + y - z; int a2 = x - z + y;
    int b1 = x * y / z; int b2 = x / z * y;
    int c1 = x << y >> z; int c2 = x >> z << y ;
    int d1 = x & y | z; int d2 = x | z & y;
```

```
答案：
    a1一定等于a2

解答：
    考察对变量运算原理的了解。
    1. 加减操作虽然可能出现溢出，但相同操作数的不同顺序只是中间结果不同，最终结果相同。
    2. int类型做除法可能会造成截断，比如3/2 = 1。
    3. 移位运算可能会丢弃超出的位数。有符号数二进制数1111 1111，先左移2位再右移三位为1111 1111，反之1111 1100。
```

Q 17 ：

```
题目：
    若有以下定义和语句：
    char s1[] = "12345", *s2 = "1234";
    printf("%d\n", strlen(strcpy(s1, s2)));
    则输出结果是：

答案：
    4

解答：
    考察strcpy和strlen。
    1. 首先strlen得到的是'\0'之前的字符长度。
    2. strcpy将s2指向的字符串'1234\0'全部拷贝到s1指向位置并覆盖其'12345'部分。
```

Q 18 ：

```
题目：
    以下函数用法正确的个数是：
    void test1(){
        unsigned char array[MAX_CHAR + 1], i;
        for(i = 0;i <= MAX_CHAR; i++){
            array[i] = i;
        }
    }

    char *test2(){
        char p[] = "hello world";
        return p;
    }
    char *p = test2();

    void test3(){
        char str[10];
```

```
        str++;
        *str = '0';
    }

答案:
    0

解答:
    考察数组名和指针区别。
    1. i的范围有可能超过unsigned char范围。
    2. 这里char p[] = "hello world"是数组，该数组是临时变量，函数结束后不能继续使用。
    3. 如果为char *p = "hello world"，这里p是指针并指向常量区字串，虽然p会被销毁，但字符串仍然在，就不会出问题。
    4. 这里str是数组名，数组名是常量，不可以自增，正确的操作是char *p = str; p++; *p = '0'。
```

Q 19 :

```
题目：
    假设在一个32位little endian的机器上运行下面的程序，结果是多少？
    #include <stdio.h>
    int main(){
        long long a = 1, b = 2, c = 3;
        printf("%d %d %d\n", a, b, c);
        return 0;
    }

答案:
    1
    0
    2

解答:
    考察小端法及printf输出控制符。
    1. long long占8字节。
    2. 小端表示，低字节在低位，最低4字节为1，接下来四字节为高位部分的0，再接下来4字节为第二个数低位的2。
    3. printf的控制符相当于分配好待打印容器大小，这里"%d %d %d"就分配了12字节，分别装入三个8字节元素，只装入一半。
```

Q 20 :

```
题目：
    请选择下列程序的运行结果:
    #include<iostream>
    using namespace std;
    class B0{
    public:
```

```
        virtual void display(){
            cout << "B0::display0" << endl;
        }
    };
    class B1:public B0{
    public:
        void display(){
            cout << "B1::display0" << endl;
        }
    };
    class D1: public B1{
    public:
        void display(){
            cout << "D1::display0" << endl;
        }
    };
    void fun(B0 ptr){
        ptr.display();
    }
    int main(){
        B0 b0;
        B1 b1;
        D1 d1;
        fun(b0);
        fun(b1);
        fun(d1);
    }

答案:
    B0::display0
    B0::display0
    B0::display0
解答:
    1. 这里传递的是对象本身而非指针，对象被直接转为基类对象，调用基类的函数。
    2. 如果要实现虚函数动态绑定需要将B0 ptr改为B0* ptr，ptr->display()。
    3. 对象的形参传递需要先使用拷贝构造函数（默认）生成B0类型的临时变量，只拷贝基类部分数据（只有指向基类虚函数表的虚函数指针）。
```

Q 21 :

```
题目：
    i的初始值为0，i++在两个线程里面分别执行100次，能得到最大值是()，最小值是()。

答案:
    200
    2

```

```
解答：
    考察多线程操作同一未上锁变量。
    1. 每次都准确加1，结果为最大，200。
    2. 结果为2时步骤：
        a取内存0到寄存器，b取内存0到寄存器；
        a执行99次并写入内存，内存值为99；
        b执行1次并写入内存，内存值被覆盖为1；
        a取内存1到寄存器，b取内存1到寄存器；
        b执行99次并写入内存，内存值为100；
        a执行1次，写入内存，覆盖之前的100，值为2。
    3. 每次计算过程必须是先从内存取数然后计算，之后再重新写入内存。但对各个线程而言，取数和计算中间可以被另一个线程打断。
```

Q 22 ：

```
题目：
    char fun(char x, char y){
        if(x)
            return(y);
    }
    int main(){
        int a = '0', b = '1', c = '2';
        printf("%c\n", fun(fun(a, b), fun(b, c)));
    }

答案：
    2

解答：
    1. 均为字符，非布尔值的0，所以每次返回后者。
```

Q 23 ：

```
题目：
    当一个类A中没有声明任何成员变量与成员函数,这时sizeof(A)的值是多少？

答案：
    1

解答：
    1. 一个空类对象的大小是1byte。这是被编译器安插进去的一个字节，这样就使得这个空类的两个实例得以在内存中配置独一无二的地址。
```

Q 24 ：

```
题目：
    有以下程序：
    #include<stdio.h>
    #include<stdlib.h>
    void fun(int *pl, int *p2, int *s){
        s = (int*)calloc(1, sizeof(int));
        *s = *pl + *p2;
        free(s);
    }
    int main(){
        int a[2] = {1, 2}, b[2] = {40, 50}, *q = a;
        fun(a, b, q);
        printf("%d\n", *q);
    }

答案：
    1

解答：
    考察形参不改变变量值问题。
    1. p是指针变量，但是是值传递，其值(指向数组a首元素的地址)并没有改变。
    2. 通过解引用*p才是数组a的地址，才能改变数组a的值。
```

Q 25* ：

```
题目：
    在32位操作系统gcc编译器环境下，下面程序的运行结果为：
    #include <iostream>
    using namespace std;
    class A{
    public:
        int b;
        char c;
        virtual void print(){
            cout << "this is father’s fuction! " << endl;
        }
    };
    class B: A{
    public:
        virtual void print(){
            cout << "this is children’s fuction! " << endl;
        }
    };
    int main(int argc, char * argv[]){
        cout << sizeof(A) << " " << sizeof(B) << endl;
        return 0;
```

```
    }

答案:
    12
    12

解答:
    考察结构体对齐及虚继承和虚函数继承的区别。
    1. A的大小包括本身的虚函数指针及定义的变量。
    2. B的大小包括本身的虚函数指针和继承自A的变量b和c。
    3. 如果是虚继承，则B的大小会增加4字节，增加的内容为指向虚继承的指针。
```

Q 26 :

```
题目：
    有如下语句序列：
    char str[10];
    cin >> str;
    当从键盘输入"I love this game"时，str中的字符串是:

答案:
    I
解答:
    1*. cin遇空格，结束输入。
```

Q 27 :

```
题目：
    阅读下面代码，程序会打印出来的值是：
    #include <stdio.h>
    void f(char** p){
        *p += 2;
    }
    int main(){
        char *a[] = {"123", "abc", "456"}, **p;
        p = a;
        f(p);
        printf("%s\r\n", *p);
    }

答案:
    3

解答:
    1. p的类型为char **，(*P)的类型为char *。
    2. p原本指向字符串"123"。
```

```
    3. *p是char *类型的，*p + 2表示指向第一个字符串第三个字符。
    4. p是char **类型的，p + 2表示只想第三个字符串，*(p + 2) = "456"。
    5. p的值是*p的地址，虽然p是形参本身值未变，但*p的值在调用函数中被改变。
```

Q 28 ：

```
题目：
    下列对函数double add(int a, int b)进行重载，正确的是：

答案：
    int add(int a, int b, int c)
    int add(double a, double b)
    double add(double a, double b)

解答：
    考察重载概念。
    1. 在使用重载时只能通过相同的方法名，不同的参数形式实现。
    2. 不同参数形式包括：
        参数类型不同（至少有一个）
        参数个数不同
        *如果同时在类中，对于函数名相同的const函数和非const函数能够构成重载
    3. 编译器区分重载函数是通过“返回类型 + 函数名 + 参数列表”重新改写函数名还区分重载函数的，但返回值类型在C++中并不作为重载标记。
```

Q 29 ：

```
题目：
    在linux gcc下，关于以下代码，正确的是：
    std::string& test_str(){
        std::string str = "test";
        return str;
    }
    int main(){
        std::string& str_ref = test_str();
        std::cout << str_ref << std::endl;
        return 0;
    }

答案：
    编译警告
    返回局部变量的引用,运行时出现未知错误
    把代码里的&都去掉之后,程序可以正常运行

解答：
    考察调用函数返回值和变量生命周期问题。
    1. 返回值为局部变量时可以正确运行。
```

```
    2. 返回值为指针时，看指针指向的变量实体定义的位置，如果是定义在栈上的变量则会出错，指向静态区则不会有问题。
    3. 引用返回的是局部变量本身，而不是复制一份再返回，所以结果难以预料。
    4. 如果去掉&，string类会调用复制构造函数，形同局部变量返回，可以正常运行。
```

Q 30 :

```
题目：
    下面有关继承、多态、组合的描述，说法错误的是：

答案:
    继承可以使用现有类的所有功能，并在无需重新编写原来的类的情况下对这些功能进行扩展
    覆盖是指不同的函数使用相同的函数名，但是函数的参数个数或类型不同

解答:
    考察继承、多态概念。
    1. 父类只有非private的部分才能被子类继承访问。
    2. 重载（overload）：函数名相同 、函数参数不同、 必须位于同一个域（类）中。
    3. 覆盖（override）：函数名相同 、函数参数相同、 分别位于派生类和基类中（虚函数）。
```

Q 31 :

```
题目：
    分析一下这段程序的输出：
    #include<iostream>
    using namespace std;
    class B{
    public:
        B(){
            cout << "default constructor" << " ";
        }
        ~B(){
            cout << "destructed" << " ";
        }
        B(int i):data(i){
            cout << "constructed by parameter" << data << " ";
        }
    private:
        int data;
    };
    B Play(B b){
        return b;
    }
    int main(int argc, char *argv[]){
        B temp = Play(5);
        return 0;
    }
```

```
26.
27. 答案:
28.     constructed by parameter5
29.     destructed
30.     destructed
31.
32. 解答:
33.     考察赋值运算顺序以及拷贝构造函数。
34.     1. B temp = Play(5)从右向左执行。
35.     2. 先将"5"转为形式参数B的类型，之后调用B(int i)打印"constructed by parameter"。
36.     3. B temp调用B的默认浅拷贝构造函数，完成赋值，由于拷贝构造函数没有输出，所以没有打印东西。
37.     4. Play()生命周期结束后，b析构打印"destructed"。
38.     5. main()生命周期结束后，temp析构打印"destructed"。
```

Q 32 :

```
1. 题目：
2.     int i=10, j=10, k=3;
3.     k*=i+j;
4.     k最后的值是？
5.
6. 答案:
7.     60
8.
9. 解答:
10.     考察运算符优先级。
11.     1. +优先级高于*=，等价于k = k * (i +j)。
```

Q 33 :

```
1. 题目：
2.     #include命令的功能是：
3.
4. 答案:
5.     在命令处插入一个文本文件
6.
7. 解答:
8.     1. "#include"在命令处插入，插入文本过程为预处理过程。
```

Q 34 :

```
1. 题目：
2.     有一个类A，其数据成员如下：
3.     class A {
```

```
    private:
        int a;
    public:
        const int b;
        float* &c;
        static const char* d;
        static double* e;
    };
    则构造函数中，成员变量一定要通过初始化列表来初始化的是：

答案:
    b
    c

解答:
    考察构造函数初始化列表的使用。
    1. 构造函数中，成员变量一定要通过初始化列表来初始化的有以下几种情况：
        const常量成员：因为常量只能在初始化，不能赋值，所以必须放在初始化列表中。
        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中。
        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。
```

Q 35 ：

```
题目：
    在一个64位的操作系统中定义如下结构体：
    struct st_task{
        uint16_t id;
        uint32_t value;
        uint64_t timestamp;
    };
    同时定义fool函数如下：
    void fool(){
        st_task task = {};
        uint64_t a = 0x00010001;
        memcpy(&task, &a, sizeof(uint64_t));
        printf("%11u, %11u, %11u", task.id, task.value, task.timestamp);
    }
    上述fool()程序的执行结果为：

答案:
    1
    0
    0

解答:
    考察结构体对齐。
    1. 假设低地址在低位，最低16 bits被赋给低16位的id变量。
```

```
    2. 接下来16 bits的0x0001部分被赋值给了pading部分，没有被使用。
    3. value和timestamp均未被赋值。
```

Q 36 ：

```
题目：
    在32位系统中：
    char arr[] = {4, 3, 9, 9, 2, 0, 1, 5};
    char *str = arr;
    sizeof(arr) = (1);
    sizeof(str) = (2);
    strlen(str) = (3);
答案：
    8；4；5

解答：
    考察指针和数组名使用sizeof时区别及转义字符。
    1. 数字0对应'\0'。
    2. strlen求字符串长到'\0'前。
```

Q 37 ：

```
题目：
    下面代码输出什么：
    #include<stdio.h>
    int main( ){
        unsigned int a = 6;
        int b = -20;
        (a + b > 6) ? printf(">6") : printf("<=6");
        return 0;
    }

答案：
    >6

解答：
    考察强制类型转换。
    1. 必须先明确：int与unsigned相加，int -> unsigned int。
    2. int b = -20，首位位"1"，用无符号型表示是非常大的正整数。
```

Q 38 ：

```
题目：
    对于下面的C语言声明描述正确的一项是：
```

```
    char (*p)[16]

答案:
    p是指向长度为16的字符数组的指针

解答:
    考察运算符优先级。
    1. p先和那个运算符结合就是什么。
    2. char *p[16]：p是一个包含16个元素的Char型指针数组，[]优于*，p[]先结合，是数组。
    3. char (*p)[16]：p是一个指针，指向一个包含16个元素的char数组，由于()出现，先和*结合，是指针。
```

Q 39 ：

```
题目：
    下面程序输出结果是什么：
    #include<iostream>
    using namespace std;
    class A{
    public:
        A(char *s){
            cout << s << endl;
        }
        ~A(){}
    };
    class B:virtual public A{
    public:
        B(char *s1, char*s2):A(s1){
            cout << s2 << endl;
        }
    };
    class C:virtual public A{
    public:
        C(char *s1, char*s2):A(s1){
            cout << s2 << endl;
        }
    };
    class D:public B, public C{
    public:
        D(char *s1, char *s2, char *s3, char *s4):B(s1, s2), C(s1, s3), A(s1){
            cout << s4 << endl;
        }
    };
    int main() {
        D *p = new D("class A", "class B", "class C", "class D");
        delete p;
        return 0;
    }
```

```
35.
36. 答案:
37.     class A
38.     class B
39.     class C
40.     class D
41.
42. 解答:
43.     考察虚继承的继承顺序。
44.     1. class B, class C为虚继承。
45.     2. 虚继承（class 派生类:virtual 继承方式 基类名）：从不同的路径继承过来的同名数据成员在内存中就只有一个拷贝。
46.     3. 继承顺序：
47.         执行基类构造函数，多个基类的构造函数按照被继承的顺序构造。
48.         执行成员对象的构造函数，多个成员对象的构造函数按照声明的顺序构造。
49.         执行派生类自己的构造函数。
50.     4. 要执行D构造函数必须先执行参数列表，欲构造B，C必须先构造A，其B，C虚继承A，所以只要执行一次构造函数。
```

Q 40 :

```
1. 题目：
2.     如下程序段：
3.     char a[] = "xyz", b[] = {'x', 'y', 'z'};
4.     if(strlen(a) > strlen(b))
5.         printf("a > b\n");
6.     else
7.         printf("a <= b\n");
8.     则程序输出：
9.
10. 答案:
11.     a<=b
12.
13. 解答:
14.     考察对strlen实现的理解。
15.     1. strlen函数判断字符串长仅仅是通过字符串末的'\0'（字符0）来确定。
16.     2. 数组b为标识具体'\0'位置，所以使用strlen结果至少大于等于3。
```

Q 41 :

```
1. 题目：
2.     执行以下语句，输出结果为：
3.     char *p1 = "hello";
4.     char *p2 = "world";
5.     char *p3 = "a piece of cake";
6.     char *str[] = {p1, p2, p3};
7.     printf("%c", *(str[0] + 1));
```

```
8.
9.  答案：
10.     e
11.
12. 解答：
13.     考察指针、指针函数及其操作。
14.     1. str是指针数组，每个元素都是指针。str[0]代表的是char *类型指针p1。
15.     2. p1是char *类型，只想字符串"hello"，所以这里“+1”代表以字符为单位，结果为'e'。
```

Q 42 ：

```
1.  题目：
2.      以下表达式那些会被编译器禁止：
3.      int a = 248, b = 4;
4.      int const c = 21;
5.      const int *d = &a;
6.      int *const e = &b;
7.      int const * const f = &a;
8.
9.  答案：
10.     *c = 32
11.     *d = 43
12.     e = &a
13.     f = 0x321f
14.
15. 解答：
16.     考察指针常量、常量指针。
17.     1. 区分是指针的值不会变还是指针指向的变量值不会变。
18.     2. 方法：
19.         如果 const 位于 * 的左侧，则 const 就是用来修饰指针所指向的变量，即指针指向为常量。
20.         如果 const 位于 * 的右侧， const 就是修饰指针本身，即指针本身是常量。
21.     3. 方法使用：
22.         int const c -> 变量c的值不可改变。
23.         const int *d -> const在*左（离指针远），修饰指向的变量 -> 指针d指向的变量不可变。
24.         int *const e-> const在*右（例指针近），修饰指针 -> 指针e的值不可变。
25.         int const * const f -> 有左有右 -> 值和指针均不可变。
```

Q 43 ：

```
1.  题目：
2.      以下描述正确的是：
3.
4.  答案：
5.      虚函数不能是内联函数
6.      父类的析构函数是非虚的，但是子类的析构函数是虚的，delete子类对象指针会调用父类的析构函数
7.
```

```
解答：
    1. 虚函数不能是内联函数（编译时展开，必须有实体），不能是静态函数（属于自身类，不属于对象，而虚函数要求有实体），不能是构造函数（尚未建立虚函数表）。
    2. delete子类对象是一定会调用父类的析构函数的先调用子类的析构函数然后调用父类的析构函数。
```

Q 44 ：

```
题目：
    class Base{
    public:
        Base(){
            Init();
        }
        virtual void Init(){
            printf("Base Init\n");
        }
        void func(){
            printf("Base func\n");
        }
    };
    class Derived: public Base{
    public:
        virtual void Init(){
            printf("Derived Init\n");
        }
        void func(){
            printf("Derived func\n");
        }
    };
    int main(){
        Derived d;
        ((Base *)&d)->func();
        return 0;
    }

答案：
    Base Init
    Base func

解答：
    考察虚函数。
    1. 类Derived继承自Base，先调用基类构造函数Base()，再调用基类的init()，输出Base Init。
        构造子类对象，基类中不会调用子类的虚函数：
            基类构造函数 -> 子类构造函数
            子类还没有构造，还没有初始化，属于未初始化对象
            基类不会去调用子类虚函数（哪怕子类中确实声明为虚函数）
    2. 虽然 ((Base *)&d)->func()是虚函数调用的样子，但func()跟本没有被定义为虚函数，基类指针访问基类的
```

```
    func()。
```

Q 45 ：

```
1. 题目：
2.     采用多路复用I/O监听3个套接字的数据时，如果套接字描述符分别是：5,17,19,则：
3.     select(int maxfd, struct fd_set* rdset, NULL, NULL)
4.     中的maxfd应取为：
5.
6. 答案:
7.     20
8.
9. 解答:
10.     1. maxfd是三个套接字描述符中最大数字加上1。
```

Q 46 ：

```
1. 题目：
2.
3.     下面的说法那个正确：
4.     #define NUMA 10000000
5.     #define NUMB 1000
6.     int a[NUMA], b[NUMB];
7.     void pa(){
8.         int i, j;
9.         for(i = 0; i < NUMB; ++i)
10.             for(j = 0; j < NUMA; ++j)
11.                 ++a[j];
12.     }
13.     void pb(){
14.         int i, j;
15.         for(i = 0; i < NUMA; ++i)
16.             for(j = 0; j < NUMB; ++j)
17.                 ++b[j];
18.     }
19.
20. 答案:
21.     pb比pa快
22.
23. 解答:
24.     1. 二维数组操作时，因为缓存的原因，外层放小循环，内层放大循环效率高。
25.     2. 个人觉得这题是因为在给大数组赋值时会发生缺页，而小数组赋值不会，所以pb比较快。
```

Q 47 ：

```
题目：
    看以下代码：
    class A{
    public:
        ~A();
    };
    A::~A(){
        printf("delete A");
    }
    class B : public A{
    public:
        ~B();
    };
    B::~B(){
        printf("delete B");
    }

    请问执行以下代码的输出是：
    A *pa = new B();
    delete pa;

答案:
    delete A

解答:
    考察虚析构函数。
    1. 若B *pb = new B(), 则本题会同时输出delete A和delete B。
    2. 但如果delete的是一个指向派生类的基类指针，则需要虚构造函数。
    3. 这里基类没有定义虚析构函数，属未定义行为。
```

Q 48 ：

```
题目：
    下面程序输出结果为？
    #include<iostream.h>
    #define SUB(X,Y) (X)*Y
    int main(){
        int a = 3, b = 4;
        cout << SUB (a++, ++b);
        return 0;
    }

答案：
    15

解答：
    考察宏定义命令和自加运算。
```

```
    1. SUB(3++, ++4) = (3++) * ++4 = 3 * 5 =15
    2. 3++是否有括号并不影响，后置自增变量值在本条语句结束前（分号之前）均不改变。
    3. 前置自增值在本条语句内立即改变且自增优先级高于*。
```

Q 49 :

```
题目：
    以下代码输出结果为：
    int main(){
        int a[2][5] = {{1, 2, 3, 4, 5}, {6, 7, 8, 9, 10}};
        int *ptr = (int *)(&a + 1);
        printf("%d\n", *(ptr - 3));
    }

答案：
    8

解答：
    考察指针运算。
    1. 首先需要明确a为二维数组，其类型是int **。
    2. 此时对a进行加1操作，操作的单位的每个一维数组。
    3. &a的单位可以理解为int ***，对其操作单位是整个二维数组。
    4. ptr = (int *)(&a + 1)指向的是二维数组后一个单元。
    5. 进行输出操作时，&a + 1被强制转换为int *类型，按四字节读取。
    6. ptr - 3从数组末尾回退3个int大小，指向数字8。
```

Q 50 :

```
题目：
    在一台主流配置的PC机上，调用f(35)所需的时间大概是：
    int f(int x){
        int s = 0;
        while(x-- > 0)
            s += f(x);
        return max(s, 1);
    }

答案：
    几分钟

解答：
    考察递归时间复杂度。
    1.   O(f(n)) = O(f(n - 1)) + O(f(n - 2)) + ... + O(f(0))
       = 2*O(f(n - 2)) + 2*(O(f(n - 3)) + ... + 2*O(f(0))
       = 2^35*O(f(0))
```

Q 51 :

```
题目：
    以下代码执行后，val的值是：
    unsigned long val = 0;
    char a = 0x48;
    char b = 0x52;
    val = b << 8 | a;

答案：
    21064

解答：
    考察默认类型转换和移位运算及优先级。
    1. 移位运算>>优先级高于|，val = (b << 8) | a;
    2. 类型转换：
        运算前转换：
            char/short -> int
            float -> double
            这里的b << 8计算时先需要把char转为int（b << 8 = 0x00005200，a = 0x00000048）
        运算中转换：
            int -> long -> unsigned -> double
    3. val = 0x00005248 =  21064
```

Q 52 :

```
题目：
    下列说法错误的有：

答案：
    在类方法中可用this来调用本类的类方法。
    在类方法中只能调用本类中的类方法。
    在类方法中绝对不能调用实例方法。

解答：
    1. 成员方法又称为实例方法，静态方法又称为类方法。
    2. 类方法（静态方法）不属于特定的类，没有this指针。
    3. 可以通过类名作用域的方式调用ClassName::fun()。
    4. 类中申请一个类对象或者参数传递一个对象或者指针都可以调用实例方法。
```

Q 53 :

```
题目：
    typedef struct{
        char flag[3];
```

```
        short value;
    }sampleStruct;
    union{
        char flag[3];
        short value;
    }sampleUnion;
    假设sizeof(char)=1, sizeof(short)=2
    那么sizeof(sampleStruct) = ()
        sizeof(sampleUnion) = ()

答案：
    6
    4

解答：
    考察结构体、联合体大小及对齐问题。
    1. union：sizeof的取值不仅考虑sizeof最大的成员，还要考虑对齐字节。
    2. 注：如果结构体内类型的最大字节小于系统位数对应的字节，那么按类型的最大字节对齐。所以本题结构体按2字节对齐。
```

Q 54 ：

```
题目：
    以下程序的输出结果为：
    #include "stdio.h"
    int func(int x, int y){
        return (x + y);
    }
    int main(){
        int a = 1, b = 2, c = 3, d = 4, e = 5;
        printf(" %d\n", func((a + b, b + c, c + a), (d, e)));
        return 0;
    }

答案：
    9

解答：
    考察逗号运算符。
    1. 逗号表达式的结果是其最右边表达式的值。
    2. (a + b, b + c, c + a)取最右边的值c + a = 4, (d, e)取最右边的值e = 5。
```

Q 55 ：

```
题目：
    当参数*x=1, *y=1, *z=1时，下列不可能是函数add的返回值的：
    int add(int *x, int *y, int *z){
```

```
        *x += *x;
        *y += *x;
        *z += *y;
        return *z;
    }

答案：
    7

解答：
    1. 此题考虑x, y, z是否可能指向同一个变量。
    2. 可能情况：
        x, y, z指向同一区域：8
        x, y指向同一区域：5
        x, z指向同一区域：5
        y, z指向同一区域：6
        x, y, z指向不同区域：4
```

Q 56 ：

```
题目：
    以下涉及到内存管理的代码段中，有错误的是：

答案：
    1. int *a = new int(12);
       free(a);
    2. int *ip = static_cast<int*>(malloc(sizeof(int)));
       *ip = 10;
       delete ip;
    3. int *ip = new int(12);
       for(int i = 0; i < 12; ++i)
           ip[i] = i;
       delete []ip;

解答：
    考察动态内存分配与释放。
    1. malloc和free, new和delete配套使用。
    2. int *ip = new int(12)：动态分配一个int类型变量并赋值为12, ip指向这个变量。
    3. int *ip = new int[12], 表示分配大小为12的int类型数组, ip指向这个数组。
```

Q 57 ：

```
题目：
    关于内联函数正确的是：

答案：
```

```
    在所有类说明中内部定义的成员函数都是内联函数

解答：
    考察内联函数。
    1. 见答案。
    2. 使用内联函数的地方会在编译阶段用内联函数体替换掉。
```

Q 58 ：

```
题目：
    以下函数中，和其他函数不属于一类的是：

答案：
    pwrite

解答：
    考察系统调用和库函数。
    1. 常见文件系统的系统函数：
        fcntl 文件控制
        open 打开文件
        creat 创建新文件
        close 关闭文件描述字
        read 读文件
        write 写文件
        read 从文件读入数据到缓冲数组中
        write 将缓冲数组里的数据写入文件
        pread 对文件随机读
        pwrite 对文件随机写
```

Q 59 ：

```
题目：
    std::vector::iterator重载了下面哪些运算符：

答案：
    ++
    ==
    *

解答：
    考察迭代器基本概念。
    1. ++和--用于迭代器以后移动。
    2. ==用于判断迭代器是否相等。
    3. *用于对迭代器指向的变量的引用。
```

Q 60 :

```
题目：
    请问func(2012,2102)的结果是：
    int fuc(int m,int n){
        if(m%n == 0)
            return n;
        else
            return fuc(n,   m%n);
    }

答案：
    2

解答：
    考察辗转相除法。
    1. 辗转相除法求两个数的最大公约数。
```

Q 61 :

```
题目：
    class Eye{
        public:
        void Look(void);
    };
    现在希望定义一个Head类，也想实现Look的功能，应该使用()方法，实现代码重用。

答案：
    组合

解答：
    考察组合和继承的使用场景。
    1. 继承是细化的继承公共的，被继承的基类是抽象出的公共部分。
    2. 组合就是在定义类时直接在新类中以原有类的对象作为数据成员。
    3. 继承是派生类对基类的扩展和包含，组合是原有类被包含，这里Eye应该被包含在Head内。
    4. 优先使用对象组合，而不是继承。
```

Q 62 :

```
题目：
    设m和n都是int类型，那么以下for循环语句：
    for(m = 0, n = -1; n = 0; m++, n++)
        n++;

答案：
```

```
    循环体一次也不执行

解答：
    1. 见答案。
```

Q 63 ：

```
题目：
    #pragma pack(2)
    class BU{
        int number;
        union UBffer{
            char buffer[13];
            int number;
        }ubuf;
        void foo(){}
        typedef char*(*f)(void*);
        enum{hdd,ssd,blueray}disk;
    }bu;

答案：
    22

解答：
    考察结构体、联合体、枚举大小综合题。
    1. pack(2)，所以int num和union大小分别为4和14没有疑问。
    2. 其他部分：
        void foo(){}：0。
        typedef char*(*f)(void*)：0。
        enum{hdd,ssd,blueray}disk：4。
        无虚函数，不存在虚函数指针的4字节。
    3. 枚举类型的sizeof值都是4。
```

Q 64 ：

```
题目：
    设变量已正确定义，以下不能统计出一行中输入字符个数（不包含回车符）的程序段是：

答案：
    int n = 0;
    for(ch = getchar(); ch != '\n'; n++);

解答：
    1. 对于for循环，其初始条件只执行一次，因此ch只从输入流中取一个字符，之后就再不会取字符，因此会死循环。
    2. int n = 0; while(getchar() != '\n') n++;和int n = 0; while(ch = getchar() != '\n') n++;均可。
```

Q 65 ：

```
题目：
    假设下面的函数foo会被多线程调用，那么让i、j、k三个变量哪些因为线程间共享访问需要加锁保护：
    int i = 0;
    void foo(){
        static int j = 0;
        int k = 0;
        i++;
        j++;
        k++;
    }

答案：
    i和j

解答：
    考察多线程情况下数据加锁问题。
    1. 多线程调用时要进行保护时，主要是针对全局变量和静态变量（无论局部或全局）的，函数内的局部变量不会受到影响。
    2. i是全局变量，j是静态局部变量。
```

Q 66 ：

```
题目：
    在C++面向对象编程语言中，以下阐述不正确的是：

答案：
    接口中可以用虚方法
    接口中可以包含已经实现的方法

解答：
    考察C++抽象类。
    1. 接口是一个概念，它在C++中用抽象类来实现。
    2. 抽象类必须是纯虚函数。
```

Q 67 ：

```
题目：
    #include<iostream>
    using namespace std;
    class MyClass{
    public:
        MyClass(int i = 0){
            cout << i;
```

```
        }
        MyClass(const MyClass &x){
            cout << 2;
        }
        MyClass &operator=(const MyClass &x){
            cout << 3;
            return *this;
        }
        ~MyClass(){
            cout << 4;
        }
    };
    int main(){
        MyClass obj1(1), obj2(2);
        MyClass obj3 = obj1;
        return 0;
    }
    运行时的输出结果是：

答案：
    122444

解答：
    考察拷贝构造函数和赋值运算符的区别。
    1. 前两个1和2没有疑问，最后三个析构输出4也没有疑问。
    2. C MyClass obj3 = obj1;
        若obj3还不存在，调用拷贝构造函数输出2。
        若obj3存在，obj3 = obj，则调用赋值运算符重载函数。
```

Q 68 ：

```
题目：
    有以下程序：
    #include < stdio.h >
    int main(){
        char a[5][10] ={"one", "two", "three", "four", "five"};
        int i, j;
        char t;
        for (i = 0; i < 2; i++){
            for(j = i + 1; j < 5; j++ ){
                if(a[i][0] > a[j][0]){
                    t = a[i][0];
                    a[i][0] = a[j][0];
                    a[j][0] = t;
                }
            }
        }
```

```
        puts(a[1]);
    }

答案：
    fwo

解答：
    1. 对五个字符串首字母进行冒泡排序。
```

Q 69 ：

```
题目：
    两个等价线程并发的执行下列程序，a为全局变量，初始为0，假设printf、++、--操作都是原子性的，则输出肯定不是哪个：

答案：
    0
    1

解答：
    考察多线程并发结果。
```

Q 70 ：

```
题目：
    在32位机器上，设有以下说明和定义：
    typedef union{
        long i;
        int k[5];
        char c;
    }DATE;
    struct data{
        int cat;
        DATE cow;
        double dog;
    }too;
    DATE max;
    则sizeof(struct data) + sizeof(max)的执行结果是：

答案：
    52

解答：
    1. 联合体大小为20。
    2. 如果最大的基本元素小于等于机器位宽，按照最大基本元素大小对齐，否则按照机器字长对齐，此处对齐单位为4字节。
```

Q 71 :

```
题目：
    若PAT是一个类，则程序运行时，语句“PAT(*ad)[3];”调用PAT的构造函数的次数是：

答案：
    0

解答：
    考察数组指针。
    1. PAT *at[3]表示指针数组，本质是数组，数组元素是指向PAT的指针，数组大小为3。
    2. PAT(*ad)[3]表示数组指针，本质是指针，但该指针指向的是一个数组且数组大小为3。
    3. 同理，int *fun()是指针函数，int (*fun)()是函数指针。
    4. 这里并未构造PAT对象。
```

Q 72 :

```
题目：
    下面有关volatile说法正确的有：

答案：
    当读取一个变量时，为提高存取速度，编译器优化时有时会先把变量读取到一个寄存器中，以后再取变量值时，就直接从寄存器中取值
    优化器在用到volatile变量时必须每次都小心地重新读取这个变量的值，而不是使用保存在寄存器里的备份
    volatile适用于多线程应用中被几个任务共享的变量

解答：
    考察volatile关键字修饰的变量。
    1. volatile作用是避免编译器优化，它是随时会变的。和const不矛盾，被const修饰的变量只是在当前作用范围无法修改，但是可能被其它程序修改。
    2. const volatile int i = 0; 表示：任何对i的直接修改都是错误的，但是i可能被意外情况修改掉。
```

Q 73 :

```
题目：
    C++中32位单精度浮点数能表示的十进制有效数字是多少位：

答案：
    7

解答：
    考察浮点数表示。
    1. float浮点数含有1bit符号位，8bit阶码，23bit位尾数，加上隐藏位的1，实际可直接表示的数在2^24以内。
    2. float可以表示的十进制有效数字7位，double为16位。
```

Q 74 ：

```
题目：
    下列关于赋值运算符“=”重载的叙述中，正确的是：

答案：
    赋值运算符只能作为类的成员函数重载

解答：
    1. 不能被重载的运算符：
        ::  ,  *  .  ?  :
    2. 必须作为成员函数重载的运算符：
        =  []  ()  ->
```

Q 75 ：

```
题目：
    函数fun的声明为int fun(int *p[4]),以下哪个变量可以作为fun的合法参数：

答案：
    int **a

解答：
    1. 函数参数为指针数组，数组大小为4。
    2.
```

Q 76 ：

```
题目：
    char* getmemory(void){
        char p[] = "hello world";
        return p;
    }
    void test(void){
        char *str = NULL;
        str = getmemory();
        printf(str);
    }
    请问运行Test函数会有什么样的结果

答案：
    输出乱码

解答：
    1. p是数组，是局部变量。
```

```
    2. 数组p的生命周期仅存在于getmemory函数中。
    3. 返回的指针指向的数据已经在调用结束后被销毁，输出乱码。
    4. 可以通过编译，只不过结果非预期。
```

Q 77 ：

```
题目：
    关于浅复制和深复制的说法，下列说法正确的是：

答案：
    浅层复制：只复制指向对象的指针，而不复制引用对象本身。
    深层复制：复制引用对象本身。
    如果是深拷贝，修改一个对象不会影响到另外一个对象。

解答：
    考察深拷贝和浅拷贝。
    1. 对象里有指针时，浅拷贝只拷贝指针字面值，并不拷贝指针指向的内容。
    2. 深拷贝会重新分配一块空间，并把被拷贝对象中指向的数据逐一复制过去。
    3. 对象中有指针时，使用深拷贝。
    4. 因为拷贝者和被拷贝里的指针指向同一区域，所以任意一个对数据的修改都会影响到另一个。
```

Q 78 ：

```
题目：
    STL中的一级容器有：

答案：
    vector, deque, list

解答：
    考察STL容器概念。
    1. STL中一级容器是容器元素本身是基本类型，非组合类型。
```

Q 79 ：

```
题目：
    程序出错在什么阶段：
    int main(void){
        http://www.taobao.com
        cout << "welcome to taobao" << endl;
    }

答案：
    正常运行
```

```
10.
11. 解答：
12.     这题有毒。
13.     1. //后面被当做注释了。
14.     2. http本身是label。
```

Q 80 ：

```
1. 题目：
2.     有如下程序段：
3.     class A{
4.         int _a;
5.     public:
6.         A(int a): _a(a){}
7.         friend int f1(A &);
8.         friend int f2(const A &);
9.         friend int f3(A);
10.        friend int f4(const A);
11.    };
12.
13. 答案：
14.    f1(0)
15.
16. 解答：
```

Q 81 ：

```
1. 题目：
2.     下面叙述错误的是：
3.      char acX[] = "abc";
4.      char acY[] = {'a', 'b', 'c'};
5.      char *szX = "abc";
6.      char *szY = "abc";
7.
8. 答案：
9.     szX的内容修改后，szY的内容也会被更改
10.
11. 解答：
12.    考察常量区字符串。
13.    1. szX和szY的值（指向的位置）相同。
14.    2. szX和szY指向的"abc"定义在常量区，不可以修改。
```

Q 82 ：

```
题目：
    以下程序输出结果是：
    class A{
        public:
        virtual void func(int val = 1){
            std::cout << "A->" << val << std::endl;
        }
        virtual void test(){
            func();
        }
    };
    class B : public A{
        public:
            void func(int val = 0){
                std::cout << "B->" << val << std::endl;
            }
    };
    int main(int argc ,char* argv[]){
        B*p = new B;
        p->test();
        return 0;
    }

答案：
    B->1

解答：
    1. 由于B类中没有覆盖（重写）基类中的虚函数test，指向派生类B的指针p调用继承自基类的函数test。
    2. test函数中继续调用虚函数func，因为虚函数执行动态绑定，p此时的动态类型为B*，因此执行的是B类中的func，输出"B->"。
    3. 缺省参数值是静态绑定，即此时val的值使用的是基类A中的缺省参数值，其值在编译阶段已经绑定，值为1。
    4. 结论：
        virtual函数是动态绑定，而缺省参数值却是静态绑定。
        绝不重新定义继承而来的缺省参数值。
```

Q 82 ：

```
题目：
    以下程序输出结果是：
    class A{
        public:
        virtual void func(int val = 1){
            std::cout << "A->" << val << std::endl;
        }
        virtual void test(){
            func();
        }
```

```
    };
    class B : public A{
        public:
            void func(int val = 0){
                std::cout << "B->" << val << std::endl;
            }
    };
    int main(int argc ,char* argv[]){
        B*p = new B;
        p->test();
        return 0;
    }

答案：
    B->1

解答：
    1. 由于B类中没有覆盖（重写）基类中的虚函数test，指向派生类B的指针p调用继承自基类的函数test。
    2. test函数中继续调用虚函数func，因为虚函数执行动态绑定，p此时的动态类型为B*，因此执行的是B类中的func，输出"B->"。
    3. 缺省参数值是静态绑定，即此时val的值使用的是基类A中的缺省参数值，其值在编译阶段已经绑定，值为1。
    4. 结论：
        virtual函数是动态绑定，而缺省参数值却是静态绑定。
        绝不重新定义继承而来的缺省参数值。
```

其他" class="reference-link">其他

数据结构与算法

计算机网络

操作系统

Q 1 ：

```
题目：
    一次I/O操作的结束，有可能导致：

答案：
    一个进程由睡眠变就绪

解答：
```

1. 独占设备：
   进程间互斥的访问这类设备，设备一旦被分配给某个进程，便由该进程独占。I/O操作后自然只有这个进程由等待进入就绪。
2. 共享设备：
   一段时间内允许多个进程同时访问的设备。对I/O设备的访问是并发，而不是并行。一次I/O操作的结束，只是其对应的进程I/O操作的结束，只会唤醒这一个进程。

Q 2 ：

题目：

在多道程序系统中，系统的现有空闲可用资源能否满足后备作业J的资源要求，是选择作业J进入内存的必要条件。

答案：

错

解答：

进入内存不一定有全部的资源。

Q 3 ：

题目：

对进程和线程的描述，以下错误的是：

答案：

父进程里的所有线程共享相同的地址空间，父进程的所有子进程共享相同的地址空间

改变进程里面主线程的状态会影响其他线程的行为，改变父进程的状态不会影响其他子进程

多线程会引起死锁，而多进程则不会

解答：

1. 子进程拥有独立的地址空间。
2. 多进程也会死锁。

Q 4 ：

题目：

在下列说法中，哪个是错误的：

答案：

若进程A和进程B在临界段上互斥，那么当进程A处于该临界段时，它不能被进程B中断

虚拟存储管理中的抖动(thrashing)现象是指页面置换(page replacement)时用于换页的时间远多于执行程序的时间

解答：

1. A进程是可以被B进程中断的，只是B不能进入临界区。

```
    2. 页面抖动现象是由于分配给进程的内存空间过小 + 不合理的置换算法导致的。
```

Q 5 ：

```
题目：
    关于读写锁的描述，以下正确的是：

答案：
    读写锁在读加锁的状态下，可用进行读共享

解答：
    1. 写加锁状态时，其他进行写操作线程会阻塞。
    2. 写锁就是防止其他进程读或写．读锁就是防止在读的时候有写进程进入。
```

Q 6 ：

```
题目：
    采用可重定位分区分配方式：

答案：
    解决了碎片问题

解答：
    1. 通过移动内存中作业的位置，把原来多个分散的小分区拼接成一个大分区的方法称为拼接或紧凑。
```

Q 7 ：

```
题目：
    若一个用户进程通过read 系统调用读取一个磁盘文件中的数据，则下列关于此过程的叙述中，正确的是：
    Ⅰ. 若该文件的数据不在内存中，则该进程进入睡眠等待状态
    Ⅱ. 请求read系统调用会导致CPU从用户态切换到核心态
    Ⅲ. read系统调用的参数应包含文件的名称

答案：
    Ⅰ & Ⅱ

解答：
    1. 通过移动内存中作业的位置，把原来多个分散的小分区拼接成一个大分区的方法称为拼接或紧凑。
    2. open系统调用应该包含文件的名称，read只是包含输入流。
```

Q 8 ：

```
题目：
```

```
    在Bash中，以下哪些说法是正确的：

答案：
    $?表示前一个命令的返回值
    $#表示参数的数量

解答：
    1. $# 是传给脚本的参数个数
    2. $0 是脚本本身的名字
    3. $1 是传递给该shell脚本的第一个参数
    4. $2 是传递给该shell脚本的第二个参数$@ 是传给脚本的所有参数的列表
    5. $* 是以一个单字符串显示所有向脚本传递的参数，与位置变量不同，参数可超过9个
    6. $$ 是脚本运行的当前进程ID号$? 是显示最后命令的退出状态，0表示没有错误，其他表示有错误
```

Q 9 ：

```
题目：
    下列关于线程调度的叙述中，错误的是：

答案：
    调用线程的yeild()方法，只会使与当前线程相同优先级的线程获得运行机会
    具有相同优先级的多个线程的调度一定是分时的

解答：

    1. yeild()使当前线程进入就绪队列，给相同优先级或者高优先级线程机会。
    2. slssp()方法会给其他任何线程提供运行的机会，不论优先级高低均可以。
```

Q 10 ：

```
题目：
    在存储管理中，采用覆盖与交换技术的目的是：

答案：
    减少程序占用的主存空间

解答：
    1. 覆盖技术的实现是把程序划分为若干个功能上相对独立的程序段，按照其自身的逻辑结构使那些不会同时运行的程序段共享同一块内存区域。程序段先保存在磁盘上，当有关程序的前一部分执行结束后，把后续程序段调入内存，覆盖前面的程序段。
    2. 在分时系统中，用户的进程比内存能容纳的数量更多，系统将那些不再运行的进程或某一部分调出内存，暂时放在外存上的一个后备存储区，通常称为交换区，当需要运行这些进程时，再将它们装入内存。
```

Q 11 ：

```
题目：
```

```
    下面有关线程的说法错误的是：

答案：
    每个线程有自己独立的地址空间
    线程包含CPU现场，可以独立执行程序

解答：
    1. 在多线程中，多个线程共享一个进程中的地址空间。
    2. 线程是CPU调度的最小单位，但不能独立执行程序。
```

Q 12 ：

```
题目：
    关于子进程和父进程的说法，下面哪一个是正确的：

答案：
    一个进程可以没有父进程或子进程

解答：
    1. init进程就没有父进程。
```

Q 13 ：

```
题目：
    下面关于软连接的描述，正确的是：

答案：
    软链接也叫符号链接
    如果原始文件被删除，所有指向它的软链接也都被破坏
    软链接指明了原始文件的位置，用户需要对原始文件的位置有访问权限才可以使用

解答：
    1. 软链接克服了硬链接的不足，没有任何文件系统的限制，任何用户可以创建指向目录的符号链接。因而现在更为广泛使用，它具有更大的灵活性，甚至可以跨越不同机器、不同网络对文件进行链接。
```

真题摘录

目录

Tencent" class="reference-link">Tencent

模拟1

Q1 ：

```
题目：
    随着IP网络的发展，为了节省可分配的注册IP地址，有一些地址被拿出来用于私有IP地址，以下不属于私有IP地址范围的是：
    A. 10.6.207.84
    B. 172.23.30.28
    C. 172.32.50.80
    D. 192.168.1.100

答案：
    C

解答：
```

```
    1. 私有IP地址共有三个范围段：
        A: 10.0.0.0~10.255.255.255, 即10.0.0.0/8。
        B: 172.16.0.0~172.31.255.255, 即172.16.0.0/12。
        C: 192.168.0.0~192.168.255.255, 即192.168.0.0/16。
    2. 私有IP在公网上不能使用，但在内网内可以通过NAT技术分配给具体设备，节省IP地址。
```

Q2 ：

```
1. 题目：
2.     下列关于一个类的静态成员的描述中，不正确的是：
3.     A. 该类的对象共享其静态成员变量的值
4.     B. 静态成员变量可被该类的所有方法访问
5.     C. 该类的静态方法能访问该类的静态成员变量
6.     D. 该类的静态数据成员变量的值不可修改
7.
8. 答案：
9.     D
10.
11. 解答：
12.     1. 类的静态成员和对象无关，和类相关，一个类的所有实例共享同一个静态成员。
13.     2. 静态成员函数不能调用非静态成员。
14.     3. 非静态成员函数可以调用静态成员。
15.     4. 静态成员变量必须初始化，且可以修改。
```

Q3 ：

```
1. 题目：
2.     C++将父类的析构函数定义为虚函数，下列正确的是哪个：
3.     A. 释放父类指针时能正确释放子类对象
4.     B. 释放子类指针时能正确释放父类对象
5.     C. 这样做是错误的
6.     D. 以上全错
7.
8. 答案：
9.     A
10.
11. 解答：
12.     1. 基类通常应定义一个虚析构函数，以确保能正确执行析构函数。
13.     2. 基类指针指向派生类对象，若基类析构函数未声明为虚函数，则只会调用基类析构函数。
14.     3. 基类声明为虚函数，释放指向派生类对象的基类指针时会先调用派生类析构函数，之后调用基类析构函数。
```

Q4 ：

```
1. 题目：
2.     下列哪一个不属于关系数据库的特点：
3.     A. 数据冗余度小
```

```
    B. 数据独立性高
    C. 数据共享性好
    D. 多用户访问

答案：
    D

解答：
    1. 数据库存在的一个目的就是统一管理数据，减少数据冗余度。
    2. 数据独立性，指数据和其管理软件独立，以及数据及其结构的独立。
    3. 数据库就是为了方便用户之间共享数据。
    4. 数据库中存在锁机制，如果多用户访问可能导致数据不一致等。
```

Q5 ：

```
题目：
    typedef char *String_t和#define String_d char *这两句在使用上有什么区别？

答案：
    1. typedef char *String_t定义了一个新的类型别名，有类型检查，更安全。发生在编译阶段。
    2. #define String_d char *仅仅是做字符串替换，无类型检查。发生在预编译阶段。
    3. 用法区别：String_t a, b;
                 String_d c, d; -> char *c, d;
                 a, b ,c是char*类型，而d为char类型。
```

Q6 ：

```
题目：
    void Func(char str_arg[2]){
        int m = sizeof(str_arg);
        int n = strlen(str_arg);
        printf("%d\n", m);
        printf("%d\n", n);
    }
    int main(void){
        char str[]="Hello";
        Func(str);
    }
    输出结果为：

答案：
    4, 5
    1. str为定义在main函数中的数组。
    2. 数组作为参数传递给函数会退化为指针。
    3. sizeof(指针变量) = 指针变量大小，strlen(指针变量) = 指针所指向的字符串长（遇'\0'停止）。
```

Q7 ：

```
题目：
    给定一个字符串,求出其最长的重复子串。

答案：
```

模拟2

Q1 ：

```
题目：
    Internet物理地址和IP地址转换采用什么协议？

答案：
    1. MAC地址 -> IP地址：ARP协议。
    2. IP地址 -> MAC地址：RARP协议。
```

Q2 ：

```
题目：
    static有什么用途？

答案：
    1. 修饰变量：
        静态局部变量：只定义一次，程序运行期间一直存在，作用于局限于定义的函数内。多线程中需要加锁保护。
        静态全局变量：程序运行期间一直存在，作用域为定义它的源文件。
    2. 修饰函数：
        一个被声明为静态的函数只可被这一模块内的其它函数调用。
```

Q3 ：

```
题目：
    引用与指针有什么区别？

答案：
    1. 指针是个实体，指针的内容是变量地址。引用只是变量别名。
    2. 指针可以指向新的变量地址。引用只能在定义时被初始化一次，之后不可变。
    3. 指针可以为空。引用不能为空。
    4. 指针可以用const修饰，引用不能用const修饰。
    5. 获取变量值指针需要解引用。引用不需要解引用。
    6. 指针变量需要分配实际内存空间。引用不需要分配内存空间，本身不是变量。
    7. 指针的sizeof得到的是指针变量的大小。引用得到的是实际变量的大小。
```

```
    8. 指针变量++是地址值的增加。引用的++是实际变量值得增加。
```

Q4 ：

```
题目：
    全局变量和局部变量在内存中是否有区别？如果有，是什么区别？

答案：
    1. 作用域：
        全局变量：具有全局作用域，只需要定义在一个源文件中就可以在所有源文件中使用。不包含变量定义的文件引用时要用extern声明。
        局部变量：具有局部作用域，只在函数运行期间存在，函数结束后就被销毁。
    2. 生存周期：
        全局变量：定义在静态区，与静态变量存储在一起，伴随程序整个生命周期。
        局部变量：定义在栈上，函数结束后释放。
```

Q5 ：

```
题目：
    什么是平衡二叉树？

答案：
    1. 空树或者左右两棵子树高度差绝对值小于1，且子树递归满足此定义。
    2. 最小平衡二叉树节点公式：F(n) = F(n - 1) + F(n - 2) + 1。
```

Q6 ：

```
题目：
    堆栈溢出一般是由什么原因导致的？

答案：
    1. 循环的递归调用（每次递归都需要压栈）。
    2. 大数据结构的局部变量。
```

Q7 ：

```
题目：
    什么函数不能声明为虚函数？

答案：
    1. 构造函数。虚函数主要针对对象而言，而构造函数是在对象创建之前。
    2. 内联函数。不能再运行中动态确定其位置。
    3. 静态成员函数。全局通用，不受限于具体对象。
```

Q8 ：

```
1. 题目：
2.     写出floatx与“零值”比较的if语句。
3.
4. 答案：
5.     1. if (fabs(x) < 0.00001f)
```

Q9 ：

```
1. 题目：
2.     不能做switch()的参数类型是？
3.
4. 答案：
5.     1. 只能是char, int, enum。
6.     2. 不能是bool, long, string, float, double。
```

Q10 ：

```
1. 题目：
2.     用户输入M、N值，从1至N开始顺序循环数数，每数到M输出该数值，直至全部输出。写出C程序。
3.
4. 答案：
```

模拟3

Q1 ：

```
1. 题目：
2.     写出下列代码的输出内容：
3.     int inc(int a){
4.         return(++a);
5.     }
6.     int multi(int*a, int*b, int*c){
7.         return(*c = *a**b);
8.     }
9.     typedef int(FUNC1)(int in);
10.    typedef int(FUNC2)(int*, int*, int*);
11.    void show(FUNC2 fun, int arg1, int*arg2){
12.        FUNC1 *p = &inc;
13.        int temp = p(arg1);
14.        fun(&temp, &arg1, arg2);
15.        printf("%d\n", *arg2);
```

```
    }
    int main(){
        int a;
        show(multi, 10, &a);
        return 0;
    }

答案：
    110
```

Q2 ：

```
题目：
    如何引用一个已经定义过的全局变量?

答案：
    1. 用extern重新声明已经在别的模块中定义的全局变量，如果写错变量名将会在链接阶段报错。
    2. 引用定义了该全局变量的头文件，如果拼写错误会在编译阶段报错。
```

Q3 ：

```
题目：
    语句for(; 1; )有什么问题？它是什么意思？

答案：
    1. 一直循环执行。
    2. 此处如果中间是0，则一次不执行。
```

Q4 ：

```
题目：
    static全局变量与普通的全局变量有什么区别？static局部变量和普通局部变量有什么区别？static函数与普通函数有什么区别？

答案：
    1. static全局变量仅能在定义的源文件中使用，全局变量可以在所有源文件中使用。
    2. static局部变量定义于静态区，生命周期为程序整个运行阶段，多次调用函数只定义一次。局部变量定义于栈，调用函数退出即销毁，多次调用多次分配。
    3. static函数只能在定义的源文件中使用。普通函数可以在头文件中声明，包含该头文件的源文件均可调用该函数。
```

Q5 ：

```
题目：
    请找出下面代码中的所有错误：
    #include<string.h>
```

```
    int main(){
        char*src = "hello,world";
        char* dest = NULL;
        int len = strlen(src);
        dest = (char*)malloc(len); (1)
        char* d = dest;
        char* s = src[len]; (2)
        while(len-- != 0) (3)
            d++ = s--; (4)
        printf("%s", dest);
        return 0;
    }

答案：
    1. 分配的空间要为len + 1，用于存放'\0'。
    2. s = &src[len]这里是取地址。
    3. 改为while(len-- >= 0)。
    4. 改为*d++ = *s--。
```

Q6 ：

```
题目：
    搜索引擎的日志要记录所有查询串，有一千万条查询，不重复的不超过三百万，要统计最热门的10条查询。
    条件：串内存<1G，字符串长0-255。
    给出主要解决思路，算法及其复杂度分析。

答案：
    1. 面对的问题有：
        (1)1G内存不够一次性装入所有数据？
        (2)如何去统计每个记录出现次数？
        (3)如何快速得到前十的记录？
    2. 解决方式：
        (1)255约2^8，一百万约2^20，即一百万记录约256MB，一千万约2.6GB。分多次处理。
        (2)利用hash统计，定义map，key为string类型日志，value为日志出现次数。
        (3)利用大根堆，取top 10，复杂度O(nlogn)。
    3. 新问题：
        如何划分数据？如何归并结果？
    4. 方案：
        (1)哈希表常驻内存，大小(255 +4) * 3 * 1000000，约800MB。
        (2)分13（200MB * 13 = 2.6GB）次调入日志数据，每次取200MB数据进行hash。
```

模拟4

Q1 ：

```
题目：
    考虑函数原型void hello(int a, int b = 7, char* pszC = "*"), 下面的函数调用中，属于不合法调用的是：
    A. hello(5);
    B. hello(5, 8);
    C. hello(6, "#");
    D. hello(0, 0, "#");

答案：
    C

解答：
    1. 参数从左往右依次赋值。
    2. 有默认值时，调用函数参数缺失时使用默认值。
    3. 参数中字符串会转为指向字符串的指针。
```

Q2 ：

```
题目：
    下列程序的运行结果为：
    #include<iostream>
    using namespace std;
    void main(){
        int a = 2;
        int b = ++a;
        cout << a / 6 << endl;
    }
    A. 0.5
    B. 0
    C. 0.7
    D. 0.666666

答案：
    B

解答：
    1. 这里的6仅仅是整型数，所以和a进行操作时不存在精度提升。
    2. a的值进过++a之后变为3，3 / 6 = 0。
```

Q3 ：

```
题目：
    #define ADD(x, y) x + y
    int m = 3;
    m += m * ADD(m, m);
    m的值为多少：
    A. 15
```

```
    B. 12
    C. 18
    D. 58

答案：
    A

解答：
    1. 原式 = m + [m * m + m] = 3 + [3 * 3 + 3] = 15
    2. 有+=时先算右边部分。
    3. 除非出现++m，否则在同一条语句内，变量值不会改变。
```

Q4 ：

```
题目：
    下面哪种情况下，B不能隐式转换为A?
    A. class B:public A{}
    B. class A:public B{}
    C. class B{operator A();}
    D. class A{A(const B&);}

答案：
    B

解答：
    1. 派生类 -> 基类，向上级转换是隐式的，只需要丢弃多余的部分即可，反之基类没有多余的空间存放B独有的变量。
    2. C是隐式类型转换操作符。
    3. D是拷贝构造函数进行隐式转化。
```

Q5 ：

```
题目：
    假设你在编写一个使用多线程技术的程序，当程序中止运行时，需要怎样一个机制来安全有效的中止所有的线程?

答案：
    1. 主线程检查是否有子线程在运行。
    2. 若有则发起线程退出操作(quit)。
    3. wait线程完全停止，delete线程对象。
    4. 等待所有线程结束(发出finish信号)，才退出程序。
```

Q6 ：

```
题目：
    从程序健壮性进行分析，下面的FillUserInfo函数和main函数分别存在什么问题?
    #define MAX_NAME_LEN 20
    struct USERINFO{
```

```
        int nAge;
        char szName[MAX_NAME_LEN];
    };
    void FillUserInfo(USERINFO *parUserInfo){
        stu::cout << "请输入用户的个数:";
        int nCount = 0;
        std::cin >> nCount;
        for (int i = 0; i < nCount; i++){
            std::cout << "请输入年龄:";
            std::cin >> parUserInfo[i]->nAge;
            std::string strName;
            std::cout << "请输入姓名:";
            std::cin >> strName;
            strcpy(parUserInfo[i].szName, strName.c_str());
        }
    }
    int main(int argc, char *argv[]){
        USERINFO arUserInfos[100] = {0};
        FillUserInfo(arUserInfos);
        printf("The first name is:");
        printf(arUserInfos[0].szName);
        printf("\n");
        return 0;
    }

答案：
```

模拟5

Q1 ：

```
题目：
    设某种二叉树有如下特点：每个结点要么是叶子结点，要么有2棵子树。假如一棵这样的二叉树中有m（m > 0）个叶子结点，那么该二叉树上的结点总数为：
    A. 2m + 1
    B. 2m - 1
    C. 2(m - 1)
    D. 2m

答案：
    B

解答：
    1. 关键考点：
        (1)叶子节点数 = 度为2的节点数 + 1
        (2)树的度 = 所有节点度的和
```

```
        (3)树的节点数 = 树的度 + 1
    2. 计算步骤：
        (1)度为2的节点数 = m - 1
        (2)树的度 = m * 0 + 0 * 0 + (m - 1) * 2 = 2m - 2
        (3)树的节点数 = (2m - 2) + 1 = 2m - 1
```

Q2 ：

```
题目：
    中断响应时间是指：
    A. 从中断处理开始到中断处理结束所用的时间
    B. 从发出中断请求到中断处理结束所用的时间
    C. 从发出中断请求到进入中断处理所用的时间
    D. 从中断处理结束到再次中断请求的时间

答案：
    C

解答：
    见答案。
```

Q3 ：

```
题目：
    试写出“背包题目”的非递归解法。

答案：
```

模拟6

Q1 ：

```
题目：
    下推自动识别机的语言是：
    A. 0型语言
    B. 1型语言
    C. 2型语言
    D. 3型语言

答案：
    C

解答：
    1. 0型文法产生的语言称为0型语言。
```

```
    2. 1型文法产生的语言称为1型语言，也称作上下文有关语言。
    3. 2型文法产生的语言称为2型语言，也称作上下文无关语言。
    4. 3型文法产生的语言称为3型语言，也称作正规语言。
```

Q2 ：

```
题目：
    浏览器访问某页面，HTTP协议返回状态码为403时表示：
    A. 找不到该页面
    B. 禁止访问
    C. 内部服务器访问
    D. 服务器繁忙

答案：
    B

解答：
    1. 100-199，指定客服端相应的某些动作
    2. 200-299，表示请求成功
    3. 300-399，用于已经移动的文件并且包含在定位头信息中指定
    4. 400-499，客服端错误
    5. 500-599，服务端错误
```

Q3 ：

```
题目：
    递归函数最终会结束，那么这个函数一定：
    A. 使用了局部变量
    B. 有一个分支不调用自身
    C. 使用了全局变量或者使用了一个或多个参数
    D. 没有循环调用

答案：
    B

解答：
    1. 分支不调用自身即函数出口。
```

Q4 ：

```
题目：
    编译过程中，语法分析器的任务是：
    A. 分析单词是怎样构成的
    B. 分析单词串是如何构成语言和说明的
    C. 分析语句和说明是如何构成程序的
    D. 分析程序的结构
```

```
7.
8. 答案：
9.     B, C, D
10.
11. 解答：
12.     1. 词法分析：词法分析是编译过程的第一个阶段。这个阶段的任务是从左到右的读取每个字符，然后根据构词规则识别单词。词法分析可以用lex等工具自动生成。
13.     2. 语法分析：语法分析是编译过程的一个逻辑阶段。语法分析在词法分析的基础上，将单词序列组合成各类语法短语，如“程序”，“语句”，“表达式”等等。语法分析程序判断程序在结构上是否正确。
14.     3. 语义分析：属于逻辑阶段。对源程序进行上下文有关性质的审查，类型检查。如赋值语句左右端类型匹配问题。
```

Q5 ：

```
1. 题目：
2.     进程进入等待状态有哪几种方式:
3.     A. CPU调度给优先级更高的线程
4.     B. 阻塞的线程获得资源或者信号
5.     C. 在时间片轮转的情况下，如果时间片到了
6.     D. 获得spinlock未果
7.
8. 答案：
9.     D
10.
11. 解答：
12.     1. A和C均是由从运行态转为就绪状态。
13.     2. B是由阻塞状态转为就绪状态。
14.     3. 自旋锁（spinlock）是一种保护临界区最常见的技术。在同一时刻只能有一个进程获得自旋锁，其他企图获得自旋锁的任何进程将一直进行尝试。
```

Q6 ：

```
1. 题目：
2.     同一进程下的线程可以共享以下：
3.     A. stack
4.     B. data section
5.     C. register set
6.     D. file fd
7.
8. 答案：
9.     B, D
10.
11. 解答：
12.     1. 线程共享的内容包括：
13.         进程代码段
14.         进程的公有数据
15.         进程打开的文件描述符
16.         信号的处理器
```

```
        进程的当前目录
        进程用户ID与进程组ID
    2. 线程独有的内容包括：
        线程ID
        寄存器组的值
        线程的堆栈
        错误返回码
        线程的信号屏蔽码
```

Q7 ：

```
题目：
    设计模式中，属于结构型模式的有哪些：
    A. 状态模式
    B. 装饰模式
    C. 代理模式
    D. 观察者模式

答案：
    B, C

解答：
    1. 创建型模式：
        单例模式
        抽象工厂模式
        建造者模式
        工厂模式
        原型模式
    2. 结构型模式：
        适配器模式
        桥接模式
        装饰模式
        组合模式
        外观模式
        享元模式
        代理模式
    3. 行为型模式：
        模版方法模式
        命令模式
        迭代器模式
        观察者模式
        中介者模式
        备忘录模式
        解释器模式
        状态模式
        策略模式
        职责链模式
```

```
37.         访问者模式
```

Q8 ：

```
1. 题目：
2.     Unix系统中，哪些可以用于进程间的通信：
3.     A. Socket
4.     B. 共享内存
5.     C. 消息队列
6.     D. 信号量
7.
8. 答案：
9.     A, B, C, D
10.
11. 解答：
12.     1. Linux进程间通信：管道、信号、消息队列、共享内存、信号量、套接字。
13.     2. Linux线程间通信：互斥量、信号量、条件变量。
14.     3. Windows进程间通信：管道、消息队列、共享内存、信号量、套接字。
15.     3. Windows线程间通信：互斥量、信号量、临界区、事件。
```

Q9 ：

```
1. 题目：
2.     设t是给定的一棵二叉树，下面的递归程序count(t)用于求得：
3.     typedef struct node{
4.         int data;
5.         struct node *lchild, *rchild;
6.     }node;
7.     int N2, NL, NR, N0;
8.     void count(node *t){
9.         if (t->lchild != NULL)
10.             if (t->rchild != NULL) N2++;
11.             else NL++;
12.         else if(t->rchild != NULL)
13.             NR++;
14.         else N0++;
15.         if(t->lchild != NULL)
16.             count(t->lchild);
17.         if(t->rchild != NULL)
18.             count(t->rchild);
19.     }
20. 答案：
```

Q10 ：

```
1. 题目：
```

```
    请设计一个排队系统，能够让每个进入队伍的用户都能看到自己在队列中所处的位置和变化，队伍可能随时有人加入和退出；当有人退出影响到用户的位置排名时需要及时反馈到用户。

答案：
```

Q11 ：

```
题目：
    A、B两个整数集合，设计一个算法求他们的交集，尽可能的高效。

答案：
```

模拟6

Q1 ：

```
题目：
    如何减少换页错误：
    A. 进程倾向于占用CPU
    B. 访问局部性（locality of reference）满足进程要求
    C. 进程倾向于占用I/O
    D. 使用基于最短剩余时间（shortest remaining time）的调度机制

答案：
    B

解答：
    1. 换页错误又称缺页错误，当一个程序试图访问没有映射到物理内存的地方时，就会出现缺页错误。
    2. 减少缺页发生的方法：
        增加作业分配的内存块数。
        增加页面大小。
        页面替换算法。
        程序满足局部性原理。
```

Q2 ：

```
题目：
    有1000亿条记录，每条记录由url,ip,时间组成，设计一个系统能够快速查询以下内容。
    1. 给定url和时间段（精确到分钟）统计url的访问次数。
    2. 给定ip和时间段（精确到分钟）统计ip的访问次数。

答案：
```

Q3 ：

```
题目：

    给定一个包含了用户query的日志文件，对于输入的任意一个字符串s，输出以s为前缀的在日志中出现频率最高的前10条query。
    至少有26台机器，每个机器存储以26个字母开头的query日志文件（机器1以a字母开头的，机器2以b字母开头……）。
    各机器维护一张哈希表，每条query在哈希表中存放其地址（哈希地址为链式的），并对其进行排序，按频率由高到低进行排序。
    当用户进行搜索时，可以很快定位到某台机器，并根据哈希表，返回出现频率最高的前10条query。

    提示：
    1. 可以预处理日志。
    2. 假设query超过10亿条，每个query不超过50字节。
    3. 考虑在大查询量的情况下如何实现分布式服务。

答案：
```

模拟7

Q1 ：

```
题目：
    下列哪些http方法对于服务端和用户端一定是安全的？
    A. GET
    B. HEAD
    C. TRACE
    D. OPTION
    E. POST

答案：
    C

解答：
```

Q2 ：

```
题目：
    一个系统，提供多个http协议的接口，返回的结果Y有json格式和jsonp格式。Json的格式为{"code":100,"msg":"aaa"}，为了保证该协议变更之后更好的应用到多个接口，为了保证修改协议不影响到原先逻辑的代码，以下哪些设计模式是需要的?协议的变更指的是日后可能返回xml格式，或者是根据需求统一对返回的消息进行过滤。
    A. Aadapter
    B. factory method
    C. proxy
    D. decorator
    E. composite

```

```
答案：
    A, B, D

解答：
```

模拟8

Q1 ：

```
题目：
    在数据库系统中，产生不一致的根本原因是：
    A. 数据存储量太大
    B. 没有严格保护数据
    C. 未对数据进行完整性控制
    D. .数据冗余

答案：


解答：
```

Q2 ：

```
题目：
    请问下面的程序一共输出多少个“-”？
    int main(void){
        int i;
        for(i = 0; i < 2; i++){
            fork();
            printf("-");
        }
        return 0;
    }
    A. 2
    B. 4
    C. 6
    D. 8

答案：


解答：
```

Q3 ：

```
题目：
    请问下面的程序一共输出多少个“-”？为什么？
    #include <stdio.h>
    #include <sys/types.h>
    #include <unistd.h>
    int main(void){
        int i;
        for (i = 0; i < 2; i++){
            fork();
            printf("-\n");
        }
        return 0;
    }
    A. 4
    B. 5
    C. 6
    D. 8

答案：


解答：
```

Q4 ：

```
题目：

    A.
    B.
    C.
    D.

答案：


解答：
```

Q5 ：

```
题目：

    A.
    B.
    C.
    D.

```

```
8. 答案：
9.
10.
11. 解答：
```

NetEase" class="reference-link">NetEase

360" class="reference-link">360

Linux工具

Linux工具

> *Linux*下还是有很多超棒的开发工具的。

在Linux日常使用中，最常用的命令自然是sudo, ls, cp, mv, cat等，但作为后台开发者，上述命令远远不够。从我的理解来看，合格的C/C++开发者至少需要从开发及调试工具、文件处理、性能分析、网络工具四个方面针对性使用一些开发工具。这里我罗列了一些，大部分都是开发中经常需要使用的命令，有些功能比较简单的命令我会给出一些基本用法，有些本身自带体系（比如vim, gdb等）的命令只能附上链接了。

开发及调试工具介绍了从“编辑 -> 编译 -> 分析目标文件 -> 追踪调用过程”的全套命令，文件处理部分介绍了查找、统计、替换等基本文本操作命令，性能分析介绍了查看进程信息、CPU负载、I/O负载、内存使用情况等基本命令，网络工具介绍了可以查看“链路层 -> 网络层 -> 传输层 -> 应用层”信息的工具。除此以外，其他命令中也列出了开发者经常会用到的一些命令，基本可以满足日常开发需要。

- 开发及调试
  - 编辑器: vim
  - 编译器: gcc/g++
  - 调试工具: gdb
  - 查看依赖库: ldd

    - 二进制文件分析：objdump
    - ELF文件格式分析：readelf
    - 跟踪进程中系统调用：strace
    - 跟踪进程栈：pstack
    - 进程内存映射：pmap

- 文件处理

    - 文件查找：find
    - 文本搜索：grep
    - 排序：sort
    - 转换：tr
    - 按列切分文本：cut
    - 按列拼接文本：paste
    - 统计行和字符：wc
    - 文本替换：sed
    - 数据流处理：awk

- 性能分析

    - 进程查询：ps
    - 进程监控：top
    - 打开文件查询：lsof
    - 内存使用量：free
    - 监控性能指标：sar

- 网络工具

    - 网卡配置：ifconfig
    - 查看当前网络连接：netstat
    - 查看路由表：route
    - 检查网络连通性：ping
    - 转发路径：traceroute
    - 网络Debug分析：nc
    - 命令行抓包：tcpdump
    - 域名解析工具：dig
    - 网络请求：curl

- 其他

    - 终止进程：kill
    - 修改文件权限：chmod
    - 创建链接：ln
    - 显示文件尾：tail

- 版本控制：git
- 设置别名：alias

内容

开发及调试

> 开发工具大部分都提供了完善的功能，所以这里不一一列举用法。从技术层面来说，调试工具比开发工具更考验一个人的工程能力。

1. 编辑器：vim
    - 服务器端开发必知必会，功能强大，这里不一一列举，但基本的打开文件、保存退出要会。
    - 详见
2. 编译器：gcc/g++
    - C/C++编译器，必知必会，除此以外需要了解预处理-> 编译 -> 汇编 -> 链接等一系列流程。
    - 详见
3. 调试工具：gdb
    - 服务器端调试必备。
    - 详见
4. 查看依赖库：ldd
    - 程序依赖库查询

```
# ldd后接可执行文件
# 第一列为程序依赖什么库，第二列为系统提供的与程序需要的库所对应的库，第三列为库加载的开始地址
# 前两列可以判断系统提供的库和需要的库是否匹配，第三列可以知道当前库在进程地址空间中对应的开始位置

ldd a.out
```

5. 二进制文件分析：objdump
    - 反汇编，需要理解汇编语言
    - 详见
6. ELF文件格式分析：readelf
    - 可以得到ELF文件各段内容，分析链接、符号表等需要用到
    - 详见

7. 跟踪进程中系统调用：strace
    - 详见
8. 跟踪进程栈：pstack
    - 详见
9. 进程内存映射：pmap
    - 显示进程内存映射
      ```shell
      # -x显示扩展信息，后接进程pid
      # Address： 内存开始地址
      # 显示信息：
      Kbytes： 占用内存的字节数
      RSS： 保留内存的字节数
      Dirty： 脏页的字节数（包括共享和私有的）
      Mode： 内存的权限：read、write、execute、shared、private
      Mapping： 占用内存的文件、或[anon]（分配的内存）、或[stack]（堆栈）
      Device： 设备名 (major:minor)

      pmap -x 12345
      ```

文件处理

> *Everything is file.* 在*Linux*环境下，对文本处理相当频繁，所以有些命令的参数还是需要记忆的。另外其他很多命令的输出信息都需要通过文件处理命令来筛选有用信息。

1. 文件查找：find

   按名查找：

    - 查找具体文件（一般方式）

      ```
      1. find . -name *.cpp
      ```

- 查找具体文件（正则方式）

```
# -regex为正则查找，-iregex为忽略大小写的正则查找

find -regex ".*.cpp$"
```

定制查找：

- 按类型查找

```
# f(file)为文件，d(dictionary)为目录，l(link)为链接

find . -type f
```

- 按时间查找

```
# atime为访问时间，x天内加参数"-atime -x"，超过x天加"-atime -x"
# mtime为修改时间

find . -type f -atime -7
```

- 按大小查找

```
# -size后接文件大小，单位可以为k(kb)，m(MB)，g(GB)

find . -type f -size -1k
```

- 按权限查询

```
# -perm后接权限

find . -type -perm 644
```

2. 文本搜索：grep

- 模式匹配

```
# 匹配test.cpp文件中含有"iostream"串的内容

grep "iostream" test.cpp
```

- 多个模式匹配

```
# 匹配test.cpp文件中含有"iostream"和"using"串的内容

grep -e "using" -e "iostream" test.cpp
```

- 输出信息

```
# -n为打印匹配的行号；-i搜索时忽略大小写；-c统计包含文本次数

grep -n "iostream" test.cpp
```

3. 排序：sort

- 文件内容行排序

```
# 排序在内存进行，不改变文件
# -n(number)表示按数字排序，-d(dictionary)表示按字典序
# -k N表示按各行第N列进行排序
# -r(reverse)为逆序排序

sort -n -k 1 test
```

4. 转换：tr

- 字符替换

```
# 转换在内存进行，不改变文件
# 将打开文件中所有目标字符替换

cat test | tr '1' '2'
```

- 字符删除

```
# 转换在内存进行，不改变文件
# -d删除(delete)

cat test | tr -d '1'
```

- 字符压缩

```
# 转换在内存进行，不改变文件
# -s位于后部

cat test | tr ' ' -s
```

5. 按列切分文本：cut

- 截取特定列

```
# 截取的内存进行，不改变文件
# -b(byte)以字节为单位，-c(character)以字符为单位，-f以字段为单位
# 数字为具体列范围

cut -f 1,2 test
```

- 指定界定符

```
# 截取的内存进行，不改变文件
# -d后接界定符

cut -f 2 -d ',' new
```

6. 按列拼接文本：paste

- 按列拼接

```
# 在内存中拼接，不改变文件
# 将两个文件按对应列拼接
# 最后加上-d "x"会将x作为指定分隔符（paste test1 test2 -d ","）
# 两文件列数可以不同

paste test1 test2
```

- 指定界定符拼接

```
# 在内存中拼接，不改变文件
# 按照-d之后给出的界定符拼接

paste test1 test2 -d ","
```

7. 统计行和字符：wc

- 基本统计

```
# -l统计行数(line)，-w统计单词数(word)，-c统计字符数(character)

wc -l test
```

8. 文本替换：sed

- 区别于上面的命令，sed是可以直接改变被编辑文件内容的。
- 详见

9. 数据流处理：awk

- 区别于上面的命令，awk是可以直接改变被编辑文件内容的。
- 详见

系统信息

> 性能监视工具对于程序员的作用就像是听诊器对于医生的作用一样。系统信息主要针对于服务器性能较低时的排查工作，主要包括*CPU*信息，文件*I/O*和内存使用情况，通过进程为纽带得到系统运行的瓶颈。

1. 进程查询：ps

- 查看正在运行进程

```
# 常结合grep筛选信息(e.g, ps -ef | grep xxx)

ps -ef
```

- 以完整格式显示所有进程

```
# 常结合grep筛选信息

ps -ajx
```

2. 进程监控：top

- 显示实时进程信息

```
# 这是个大招，都不带参数的，具体信息通过grep筛选
# 交互模式下键入M进程列表按内存使用大小降序排列，键入P进程列表按CPU使用大小降序排列
# %id表示CPU空闲率，过低表示可能存在CPU存在瓶颈
# %wa表示等待I/O的CPU时间百分比，过高则I/O存在瓶颈 > 用iostat进一步分析

top
```

3. 打开文件查询：lsof

- 查看占用某端口的进程

```
# 最常见的就是mysql端口被占用使用(lsof i:3307)
# 周知端口(ftp:20/21, ssh:22, telnet:23, smtp:25, dns:53, http:80, pop3:110, https:443)
```

```
3.
4. lsof -i:53
```

- 查看某用户打开的文件
  ```shell

# -u(user)为用户，后接用户名

lsof -u inx

```
1.
2. - 查看指定进程打开的文件
3. ```shell
4. # -p(process)为进程，后接进程PID
5.
6. lsof -p 12345
```

- 查看指定目录下被进程打开的文件
  ```shell

# 这里是”+d”，需要注意，使用”+D”递归目录

lsof +d /test
```
```

4. 内存使用量：free

   - 内存使用量

```
1. # 可获得内存及交换区的总量，已使用量，空闲量等信息
2.
3. free
```

5. 监控性能指标：sar

   监控CPU

   - 监控CPU负载

```
1. # 加上-q可以查看运行队列中进程数，系统上进程大小，平均负载等
2. # 这里"1"表示采样时间间隔是1秒，这里"2"表示采样次数为2
```

```
3.
4. sar -q 1 2
```

- 监控CPU使用率

```
1. # 可以显示CPU使用情况
2. # 参数意义同上
3.
4. sar -u 1 2
```

监控内存

- 查询内存

```
1. # 可以显示内存使用情况
2. # 参数意义同上
3.
4. sar -r 1 2
```

- 页面交换查询

```
1. # 可以查看是否发生大量页面交换，吞吐率大幅下降时可用
2. # 参数意义同上
3.
4. sar -W 1 2
```

网络工具

> 网络工具部分只介绍基本功能，参数部分一笔带过。这部分重点不在于工具的使用而是对反馈的数据进行解读，并且这部分命令功能的重合度还是比较高的。

1. 网卡配置（链路层）：ifconfig

- 显示设备信息
  ```shell

# 可以显示已激活的网络设备信息

ifconfig
```

```
1. - 启动关闭指定网卡
2. ```shell
```

```
# 前一个参数为具体网卡，后一个为开关信息
# up为打开，down为关闭

ifconfig eth0 up
```

- 配置IP地址
  ```shell

# 前一个参数为具体网卡，后一个为配置的IP地址

ifconfig eth0 192.168.1.1
```

```
- 设置最大传输单元
```shell
前一个参数为具体网卡，后面为MTU的大小
# 设置链路层MTU值，通常为1500

ifconfig eth0 mtu 1500
```

- 启用和关闭ARP协议
  ```

开启arp如下，若关闭则-arp

ifconfig eth0 arp
```

1. 查看当前网络连接（链路层/网络层/传输层）：netstat

   - 网络接口信息

     ```
     # 显示网卡信息，可结合ifconfig学习

     netstat -i
     ```

   - 列出端口

     ```
     # -a(all)表示所有端口，-t(tcp)表示所有使用中的TCP端口
     # -l(listening)表示正在监听的端口

     ```
     ```

```
4. netstat -at
```

- 显示端口统计信息

```
1. # -s(status)显示各协议信息
2. # -加上-t(tcp)显示tcp协议信息，加上-u(udp)显示udp协议信息
3.
4. netstat -s
```

- 显示使用某协议的应用名

```
1. # -p(progress)表示程序，可以显示使用tcp/udp协议的应用的名称
2.
3. netstat -pt
```

- 查找指定进程、端口

```
1. # 互逆操作第一个显示某程序使用的端口号，第二个显示某端口号的使用进程
2. # 第二个操作可以用lsof替代
3.
4. netstat -ap | grep ssh
5. netstat -an | grep ':80'
```

2. 查看路由表（网络层IP协议）：route

- 查看路由信息

```
1. # 得到路由表信息，具体分析路由表工作需要网络知识
2. # 可以通过netstat -r(route)得到同样的路由表
3.
4. route
```

3. 检查网络连通性（网络层ICMP协议）：ping

- 检查是否连通

```
1. # 主要功能是检测网络连通性
2. # 可以额外得到网站的ip地址和连接最大/最小/平均耗时。
3.
4. ping baidu.com
```

4. 转发路径（网络层ICMP协议）：traceroute

- 文件包途径的IP

```
#
# 可以打印从沿途经过的路由器IP地址

traceroute baidu.com
```

5. 网络Debug分析（网络层/传输层）：nc

- 端口扫描

```
# 黑客很喜欢
# 扫描某服务器端口使用情况
# -v(view)显示指令执行过程，-w(wait)设置超时时长
# -z使用输入输出模式（只在端口扫描时使用）
# 数字为扫描的端口范围

nc -v -w 1 baidu.com  -z 75-1000
```

- 其他详见

6. 命令行抓包（网络层/传输层）：tcpdump

- 抓包利器，没有什么比数据更值得信赖。可以跟踪整个传输过程。
- 详见

7. 域名解析工具（应用层DNS协议）：dig

```
# 应用层，DNS
# 打印域名解析结果
# 打印域名解析过程中涉及的各级DNS服务器地址

dig baidu.com
```

8. 网络请求（应用层）：curl

- 详见

其他

> 这里都是日常开发中高频命令。

1. 终止进程：kill

- 杀死具体进程

```
# 加具体进程PID

kill 12345
```

- 杀死某进程相关进程

```
# 加上"-9"杀死某进程相关进程

kill -9 12345
```

2. 修改文件权限：chmod

- 更改文件权限

```
# 可以对三种使用者设置权限，u(user, owner)，g(group)，o(other)
# 文件可以有三种权限，r(read)，w(write)，x(execute)
# 这里u+r表示文件所有者在原有基础上增加文件读取权限
# 这里777分别对应，u=7，g=7，o=7，具体数字含义自行google

chmod u+r file
chmod 777 file
```

3. 创建链接：ln

- 创建硬链接

```
# 文件inode中链接数会增加，只有链接数减为0时文件才真正被删除

ln file1 file2
```

- 创建软（符号链接）链接

```
# -s(symbol)为符号链接，仅仅是引用路径
# 相比于硬链接最大特点是可以跨文件系统
# 类似于Windows创建快捷方式，实际文件删除则链接失效

ln -s file1 file2
```

4. 显示文件尾：tail

- 查看文件尾部

```
# -f参数可以不立即回传结束信号，当文件有新写入数据时会及时更新
# 查看日志时常用

tail -f test
```

5. 版本控制：git

   - 版本控制最好用的软件，没有之一。至少要知道”git init”，”git add”，”git commit”，”git pull”，”git push”几个命令。
   - 详见

6. 设置别名：alias

   - 常用命令添加别名

```
# ".bashrc"文件中配置常用命令别名，生效后在命令行只需要使用别名即可代替原先很长的命令

alias rm='rm -i'
```

实战

假设已经通过vim编辑，gcc编译得到可执行文件server，这时就可以使用一些开发者常用的工具来进行后期调试。这里都是给出最简单的用法，意在快速掌握一些基本开发工具。

先clone这个项目，然后使用src_code下代码编译通过后通过下面命令调试。代码

1. 单步调试：gdb

- 运行得不到正确结果可以通过gdb设置断点来查看每个中间变量值，以此来确定哪里出了问题。因为gdb调试内容较多，这里不详细说明。另外，gdb出了可以单步查看变量值，还可以分析coredump文件来排查错误。

1. 动态库依赖：ldd

- 命令：ldd ./server
- 可以查看可执行文件server所需的所有动态库，动态库所在目录及其被映射到的虚拟地址空间。

1. 性能分析：top

- top可以查看当前系统很多信息，比如1,5,15分钟内负载，运行、休眠、僵尸进程数，用户、内核程序占CPU百分比，存储信息等。top可以定位具体哪个进程CPU占用率高和内存使用率高。我们可以以此定位性能问题出在什么程序上（比如你后台执行TKeed server之后，可以看到CPU占用率为99%，这时候我们就需要从这个程序入手了）。

1. 系统调用：strace

- 命令：strace ./server
- 上面已经提到TKeed server的CPU占用率为99%，那么问题通常一定是出在了死循环上。我们接下来在代码中找到死循环位置。因为程序中epoll_wait需要阻塞进程，我们怀疑是不是这里没有阻塞，这时就可以通过上面的方式运行server程序。此时可以打印出没次系统调用及其参数等，我们也可以加-o filename将系统调用信息保存下来。

1. 打印进程：ps

- 命令：ps -ejH
- 我们在命令行下打开的程序的父进程是shell程序，之前用strace打开server程序，strace也是server的父进程。我们有时候需要知道进程间的层级关系就需要打印进程树，上面的ps命令可以做到。当出现僵尸进程时就可以通过进程树定位具体是哪个进程出了问题。另外当想要知道进程pid时，ps -el | grep XXX也是很常用的。

1. 打开文件：lsof

- lsof -i:3000
- 比如在运行server时发现端口被占用了，可以通过lsof -i:port来查看对应端口号正在被哪个进程所占用。端口占用是非常常见的问题，比如3306被占用我遇到过好几次，要么是某个程序正好占用了要么是之前没能结束进程，这些都可以借助lsof帮助查看端口。

1. 修改权限：chmod

- chmod 000 ./index.html
- 可以修改文件权限，这里设为000，这样任何人都无法访问，重新在浏览器请求127.0.0.1:3000/index.html就会因为文件权限不够而无法展示，服务器返回状态码为403，符合我们预期。修改权限后再请求一次可得到状态码200。

1. 网卡信息：ifconfig

- ifconfig
- 如果想看一下整个传输过程，可以使用tcpdump来抓包，但是抓包时参数需要加上网卡信息，这时候可以通过ifconfig来获得网卡信息。

1. 抓包分析：tcpdump

- tcpdump -i eth0 port 3000
- 可以用tcpdump来抓包分析三次握手及数据传输过程，-i之后加上上一步得到的网卡地址，port可以指定监听的端口号。

编程基础(C/C++)

编程语言(C/C++)

> 都是语言，为什么英语比C++难这么多呢？

目录

内容

编程基础" class="reference-link">编程基础

C/C++的内容又多又杂，常常看到有人罗列相关书单，觉得毫无意义，我不相信他们真的完全掌握了其中任何一本。学习任何东西，首先要掌握基本概念，基础不牢地动山摇，因为高级的内容都是通过低级的概念来描述的。当基本概念都没理解透，学习再多都是空中楼阁。这里罗列了一些听基本的问题，虽然看着不难，但是精确理解每句话中的每个词真的并不容易。

1. 变量声明和定义区别？
    - 声明仅仅是把变量的声明的位置及类型提供给编译器，并不分配内存空间；定义要在定义的地方为其分配存储空间。
    - 相同变量可以再多处声明（外部变量extern），但只能在一处定义。
2. “零值比较”？
    - bool类型: if(flag)

- int类型: if(flag == 0)
- 指针类型: if(flag == null)
- float类型: if((flag >= -0.000001) && (flag <= 0. 000001))

3. strlen和sizeof区别？
   - sizeof是运算符，并不是函数，结果在编译时得到而非运行中获得；strlen是字符处理的库函数。
   - sizeof参数可以是任何数据的类型或者数据（sizeof参数不退化）；strlen的参数只能是字符指针且结尾是'\0'的字符串。
   - 因为**sizeof**值在编译时确定，所以不能用来得到动态分配（运行时分配）存储空间的大小。
4. 同一不同对象可以互相赋值吗？
   - 可以，但含有指针成员时需要注意。
   - 对比类的对象赋值时深拷贝和浅拷贝。
5. 结构体内存对齐问题？
   - 结构体内成员按照声明顺序存储，第一个成员地址和整个结构体地址相同。
   - 未特殊说明时，按结构体中size最大的成员对齐（若有double成员），按8字节对齐。
6. static作用是什么？在C和C++中有何区别？
   - static可以修饰局部变量（静态局部变量）、全局变量（静态全局变量）和函数，被修饰的变量存储位置在静态区。对于静态局部变量，相对于一般局部变量其生命周期长，直到程序运行结束而非函数调用结束，且只在第一次被调用时定义；对于静态全局变量，相对于全局变量其可见范围被缩小，只能在本文件中可见；修饰函数时作用和修饰全局变量相同，都是为了限定访问域。
   - C++的static除了上述两种用途，还可以修饰类成员（静态成员变量和静态成员函数），静态成员变量和静态成员函数不属于任何一个对象，是所有类实例所共有。
   - static的数据记忆性可以满足函数在不同调用期的通信，也可以满足同一个类的多个实例间的通信。
   - 未初始化时，static变量默认值为0。
7. 结构体和类的区别？
   - 结构体的默认限定符是public；类是private。

- ~~结构体不可以继承，类可以。~~ C++中结构体也可以继承。

8. malloc和new的区别？

   - malloc和free是标准库函数，支持覆盖；new和delete是运算符，并且支持重载。
   - malloc仅仅分配内存空间，free仅仅回收空间，不具备调用构造函数和析构函数功能，用malloc分配空间存储类的对象存在风险；new和delete除了分配回收功能外，还会调用构造函数和析构函数。
   - malloc和free返回的是void类型指针（必须进行类型转换），new和delete返回的是具体类型指针。

9. 指针和引用区别？

   - 引用只是别名，不占用具体存储空间，只有声明没有定义；指针是具体变量，需要占用存储空间。
   - 引用在声明时必须初始化为另一变量，一旦出现必须为typename refname &varname形式；指针声明和定义可以分开，可以先只声明指针变量而不初始化，等用到时再指向具体变量。
   - 引用一旦初始化之后就不可以再改变（变量可以被引用为多次，但引用只能作为一个变量引用）；指针变量可以重新指向别的变量。
   - 不存在指向空值的引用，必须有具体实体；但是存在指向空值的指针。

0. 宏定义和函数有何区别？

   - 宏在编译时完成替换，之后被替换的文本参与编译，相当于直接插入了代码，运行时不存在函数调用，执行起来更快；函数调用在运行时需要跳转到具体调用函数。
   - 宏函数属于在结构中插入代码，没有返回值；函数调用具有返回值。
   - 宏函数参数没有类型，不进行类型检查；函数参数具有类型，需要检查类型。
   - 宏函数不要在最后加分号。

1. 宏定义和const区别？

   - 宏替换发生在编译阶段之前，属于文本插入替换；const作用发生于编译过程中。
   - 宏不检查类型；const会检查数据类型。
   - 宏定义的数据没有分配内存空间，只是插入替换掉；const定义的变量只是值不能改变，但要分配内存空间。

2. 宏定义和typedef区别？

- 宏主要用于定义常量及书写复杂的内容；typedef主要用于定义类型别名。
- 宏替换发生在编译阶段之前，属于文本插入替换；typedef是编译的一部分。
- 宏不检查类型；typedef会检查数据类型。
- 宏不是语句，不在在最后加分号；typedef是语句，要加分号标识结束。
- 注意对指针的操作，typedef char *p_char*和*#define p_char char* 区别巨大。

3. 宏定义和内联函数(inline)区别？
   - 在使用时，宏只做简单字符串替换（编译前）。而内联函数可以进行参数类型检查（编译时），且具有返回值。
   - 内联函数本身是函数，强调函数特性，具有重载等功能。
   - 内联函数可以作为某个类的成员函数，这样可以使用类的保护成员和私有成员。而当一个表达式涉及到类保护成员或私有成员时，宏就不能实现了。
4. 条件编译#ifdef, #else, #endif作用？
   - 可以通过加#define，并通过#ifdef来判断，将某些具体模块包括进要编译的内容。
   - 用于子程序前加#define DEBUG用于程序调试。
   - 应对硬件的设置（机器类型等）。
   - 条件编译功能if也可实现，但条件编译可以减少被编译语句，从而减少目标程序大小。
5. 区别以下几种变量？

```
1. const int a;
2. int const a;
3. const int *a;
4. int *const a;
```

   - int const a和const int a均表示定义常量类型a。
   - const int *a*，其中*a*为指向*int*型变量的指针，*const*在 左侧，表示a指向不可变常量。(看成const (*a)，对引用加const)
   - int *const a，依旧是指针类型，表示a为指向整型数据的常指针。(看成const(a)，对指针const)
6. volatile有什么作用？

- volatile定义变量的值是易变的，每次用到这个变量的值的时候都要去重新读取这个变量的值，而不是读寄存器内的备份。
- 多线程中被几个任务共享的变量需要定义为volatile类型。

7. 什么是常引用？

- 常引用可以理解为常量指针，形式为const typename & refname = varname。
- 常引用下，原变量值不会被别名所修改。
- 原变量的值可以通过原名修改。
- 常引用通常用作只读变量别名或是形参传递。

8. 区别以下指针类型？

```
int *p[10]
int (*p)[10]
int *p(int)
int (*p)(int)
```

- int *p[10]表示指针数组，强调数组概念，是一个数组变量，数组大小为10，数组内每个元素都是指向int类型的指针变量。
- int (*p)[10]表示数组指针，强调是指针，只有一个变量，是指针类型，不过指向的是一个int类型的数组，这个数组大小是10。
- int *p(int)*是函数声明，函数名是*p*，参数是*int*类型的，返回值是*int* 类型的。
- int (*p)()是函数指针，强调是指针，该指针指向的函数具有int类型参数，并且返回值是int类型的。

9. 常量指针和指针常量区别？

- 常量指针是一个指针，读成常量的指针，指向一个只读变量。如int const *p*或*const int* p。
- 指针常量是一个不能给改变指向的指针。如int *const p。

0. a和&a有什么区别？

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

- a是数组名，是数组首元素地址，+1表示地址值加上一个int类型的大小，如果a的值是

0x00000001，加1操作后变为0x00000005。*(a + 1) = a[1]。

- &a是数组的指针，其类型为int (*)[10]（就是前面提到的数组指针），其加1时，系统会认为是数组首地址加上整个数组的偏移（10个int型变量），值为数组a尾元素后一个元素的地址。
- 若(int)*p* ，此时输出 p时，其值为a[0]的值，因为被转为int *类型，解引用时按照int类型大小来读取。

1. 数组名和指针（这里为指向数组首元素的指针）区别？
   - 二者均可通过增减偏移量来访问数组中的元素。
   - 数组名不是真正意义上的指针，可以理解为常指针，所以数组名没有自增、自减等操作。
   - 当数组名当做形参传递给调用函数后，就失去了原有特性，退化成一般指针，多了自增、自减操作，但sizeof运算符不能再得到原数组的大小了。
2. 野指针是什么？
   - 也叫空悬指针，不是指向null的指针，是指向垃圾内存的指针。
   - 产生原因及解决办法：
     - 指针变量未及时初始化 => 定义指针变量及时初始化，要么置空。
     - 指针free或delete之后没有及时置空 => 释放操作后立即置空。
3. 堆和栈的区别？
   - 申请方式不同。
     - 栈由系统自动分配。
     - 堆由程序员手动分配。
   - 申请大小限制不同。
     - 栈顶和栈底是之前预设好的，大小固定，可以通过ulimit -a查看，由ulimit -s修改。
     - 堆向高地址扩展，是不连续的内存区域，大小可以灵活调整。
   - 申请效率不同。
     - 栈由系统分配，速度快，不会有碎片。
     - 堆由程序员分配，速度慢，且会有碎片。

4. delete和delete[]区别？
    - delete只会调用一次析构函数。
    - delete[]会调用数组中每个元素的析构函数。

面向对象基础" class="reference-link">面向对象基础

能够准确理解下面这些问题是从C程序员向C++程序员进阶的基础。当然了，这只是一部分。

1. 面向对象三大特性？
    - 封装性：数据和代码捆绑在一起，避免外界干扰和不确定性访问。
    - 继承性：让某种类型对象获得另一个类型对象的属性和方法。
    - 多态性：同一事物表现出不同事物的能力，即向不同对象发送同一消息，不同的对象在接收时会产生不同的行为（重载实现编译时多态，虚函数实现运行时多态）。
2. public/protected/private的区别？
    - public的变量和函数在类的内部外部都可以访问。
    - protected的变量和函数只能在类的内部和其派生类中访问。
    - private修饰的元素只能在类内访问。
3. 对象存储空间？
    - 非静态成员的数据类型大小之和。
    - 编译器加入的额外成员变量（如指向虚函数表的指针）。
    - 为了边缘对齐优化加入的panding。
4. C++空类有哪些成员函数?
    - 首先，空类大小为1字节。
    - 默认函数有：
        - 构造函数
        - 析构函数
        - 拷贝构造函数
        - 赋值运算符

5. 构造函数能否为虚函数，析构函数呢？

    - 析构函数：
        - 析构函数可以为虚函数，并且一般情况下基类析构函数要定义为虚函数。
        - 只有在基类析构函数定义为虚函数时，调用操作符delete销毁指向对象的基类指针时，才能准确调用派生类的析构函数（从该级向上按序调用虚函数），才能准确销毁数据。
        - 析构函数可以是纯虚函数，含有纯虚函数的类是抽象类，此时不能被实例化。但派生类中可以根据自身需求重新改写基类中的纯虚函数。
    - 构造函数：
        - 构造函数不能定义为虚函数，不仅如此，构造函数中还不能调用虚函数。因为那样实际执行的是父类对应的函数，因为自己还没有构造好（构造顺序先基类再派生类）。

6. 构造函数调用顺序，析构函数呢？

    - 基类的构造函数：如果有多个基类，先调用纵向上最上层基类构造函数，如果横向继承了多个类，调用顺序为派生表从左到右顺序。
    - 成员类对象的构造函数：如果类的变量中包含其他类（类的组合），需要在调用本类构造函数前先调用成员类对象的构造函数，调用顺序遵照在类中被声明的顺序。
    - 派生类的构造函数。
    - 析构函数与之相反。

7. 拷贝构造函数中深拷贝和浅拷贝区别？

    - 深拷贝时，当被拷贝对象存在动态分配的存储空间时，需要先动态申请一块存储空间，然后逐字节拷贝内容。
    - 浅拷贝仅仅是拷贝指针字面值。
    - 当使用浅拷贝时，如果原来的对象调用析构函数释放掉指针所指向的数据，则会产生空悬指针。因为所指向的内存空间已经被释放了。

8. 拷贝构造函数和赋值运算符重载的区别？

    - 拷贝构造函数是函数，赋值运算符是运算符重载。
    - 拷贝构造函数会生成新的类对象，赋值运算符不能。
    - 拷贝构造函数是直接构造一个新的类对象，所以在初始化对象前不需要检查源对象和新建对象是否相同；赋值运算符需要上述操作并提供两套不同的复制策略，另外赋值运算符中如果原来的对象有内存分配则需要先把内存释放掉。

- 形参传递是调用拷贝构造函数（调用的被赋值对象的拷贝构造函数），但并不是所有出现"="的地方都是使用赋值运算符，如下：

```
Student s;
Student s1 = 2;    // 调用拷贝构造函数
Student s2;
s2 = s;    // 赋值运算符操作
```

注：类中有指针变量时要重写析构函数、拷贝构造函数和赋值运算符

9. 虚函数和纯虚函数区别？

- 虚函数是为了实现动态编联产生的，目的是通过基类类型的指针指向不同对象时，自动调用相应的、和基类同名的函数（使用同一种调用形式，既能调用派生类又能调用基类的同名函数）。虚函数需要在基类中加上virtual修饰符修饰，因为virtual会被隐式继承，所以子类中相同函数都是虚函数。当一个成员函数被声明为虚函数之后，其派生类中同名函数自动成为虚函数，在派生类中重新定义此函数时要求函数名、返回值类型、参数个数和类型全部与基类函数相同。

- 纯虚函数只是相当于一个接口名，但含有纯虚函数的类不能够实例化。

0. 覆盖、重载和隐藏的区别？

- 覆盖是派生类中重新定义的函数，其函数名、参数列表（个数、类型和顺序）、返回值类型和父类完全相同，只有函数体有区别。派生类虽然继承了基类的同名函数，但用派生类对象调用该函数时会根据对象类型调用相应的函数。覆盖只能发生在类的成员函数中。

- 隐藏是指派生类函数屏蔽了与其同名的函数，这里仅要求基类和派生类函数同名即可。其他状态同覆盖。可以说隐藏比覆盖涵盖的范围更宽泛，毕竟参数不加限定。

- 重载是具有相同函数名但参数列表不同（个数、类型或顺序）的两个函数（不关心返回值），当调用函数时根据传递的参数列表来确定具体调用哪个函数。重载可以是同一个类的成员函数也可以是类外函数。

1. 在main执行之前执行的代码可能是什么？

- 全局对象的构造函数。

2. 哪几种情况必须用到初始化成员列表？

- 初始化一个const成员。

- 初始化一个reference成员。

- 调用一个基类的构造函数，而该函数有一组参数。

- 调用一个数据成员对象的构造函数，而该函数有一组参数。

3. 什么是虚指针？

- 虚指针或虚函数指针是虚函数的实现细节。
- 虚指针指向虚表结构。

4. 重载和函数模板的区别？

- 重载需要多个函数，这些函数彼此之间函数名相同，但参数列表中参数数量和类型不同。在区分各个重载函数时我们并不关心函数体。
- 模板函数是一个通用函数，函数的类型和形参不直接指定而用虚拟类型来代表。但只适用于参个数相同而类型不同的函数。

5. this指针是什么？

- this指针是类的指针，指向对象的首地址。
- this指针只能在成员函数中使用，在全局函数、静态成员函数中都不能用this。
- this指针只有在成员函数中才有定义，且存储位置会因编译器不同有不同存储位置。

6. 类模板是什么？

- 用于解决多个功能相同、数据类型不同的类需要重复定义的问题。
- 在建立类时候使用template及任意类型标识符T，之后在建立类对象时，会指定实际的类型，这样才会是一个实际的对象。
- 类模板是对一批仅数据成员类型不同的类的抽象，只要为这一批类创建一个类模板，即给出一套程序代码，就可以用来生成具体的类。

7. 构造函数和析构函数调用时机？

- 全局范围中的对象：构造函数在所有函数调用之前执行，在主函数执行完调用析构函数。
- 局部自动对象：建立对象时调用构造函数，函数结束时调用析构函数。
- 动态分配的对象：建立对象时调用构造函数，调用释放时调用析构函数。
- 静态局部变量对象：建立时调用一次构造函数，主函数结束时调用析构函数。

标准模板库" class="reference-link">标准模板库

STL内容虽然看起来很多，单独成书都不是问题（《STL源码剖析》），但从实际使用状况来看，我认

为只需要知道以下几点就可以了：

- 怎么用？

  各种STL基本的增删改查怎么使用。每种容器都提供了很多操作，但实际增删改查我们通常只需要掌握透彻一种方式即可。有些功能只是出于通用性考虑才存在的，但对于相应的STL这些操作完全可以忽略。所以我对STL使用的看法是，不需要花太多时间去了解所有功能，只要掌握最基本的即可，要把精力放在对需求的了解并选择适合的数据结构。

- 怎么实现？

  本身STL就是封装了我们常用的数据结构，所以最先需要了解每种数据结构的特性。而且了解实现方式对我们能够准确、高效使用STL打下了基础。

- 如何避免错误？

  在第二阶段了解了STL的实现之后，我们已经可以很清楚地知道他们底层使用的是什么数据结构以及该数据结构做什么操作比较高效。但还有一点需要注意的就是怎么才能用对他们，避免一些未知的错误，比如迭代器失效问题。

string

vector

用法：

```
1.    定义：
2.        vector<T> vec;
3.
4.    插入元素：
5.        vec.push_back(element);
6.        vec.insert(iterator, element);
7.
8.    删除元素：
9.        vec.pop_back();
10.       vec.erase(iterator);
11.
12.   修改元素：
13.       vec[position] = element;
14.
15.   遍历容器：
16.       for(auto it = vec.begin(); it != vec.end(); ++it) {......}
17.
18.   其他：
19.       vec.empty();    //判断是否空
20.       vec.size();    // 实际元素
21.       vec.capacity();    // 容器容量
```

```
22.         vec.begin();    // 获得首迭代器
23.         vec.end();    // 获得尾迭代器
24.         vec.clear();    // 清空
```

实现：

模拟Vector实现

- 线性表，数组实现。
    - 支持随机访问。
    - 插入删除操作需要大量移动数据。
- 需要连续的物理存储空间。
- 每当大小不够时，重新分配内存（*2），并复制原内容。

错误避免：

迭代器失效

- 插入元素
    - 尾后插入：size < capacity时，首迭代器不失效尾迭代实现（未重新分配空间），size == capacity时，所有迭代器均失效（需要重新分配空间）。
    - 中间插入：size < capacity时，首迭代器不失效但插入元素之后所有迭代器失效，size == capacity时，所有迭代器均失效。
- 删除元素
    - 尾后删除：只有尾迭代失效。
    - 中间删除：删除位置之后所有迭代失效。

map

用法：

```
1.     定义：
2.         mao<T_key, T_value> map;
3.
4.     插入元素：
5.         map.insert(pair<T_key, T_value>(key, value));    // 同key不插入
6.         map.insert(map<T_key, T_value>::value_type(key, value));    // 同key不插入
7.         map[key] = value;    // 同key覆盖
8.
```

```
    删除元素：
        map.erase(key);    // 按值删
        map.erase(iterator);    // 按迭代器删

    修改元素：
        map[key] = new_value;

    遍历容器：
          for(auto it = vec.begin(); it != vec.end(); ++it) {......}
```

实现：

RBTree实现

- 树状结构，RBTree实现。
    - 插入删除不需要数据复制。
    - 操作复杂度仅跟树高有关。
- RBTree本身也是二叉排序树的一种，key值有序，且唯一。
    - 必须保证key可排序。

基于红黑树实现的map结构（实际上是map， set， multimap，multiset底层均是红黑树），不仅增删数据时不需要移动数据，其所有操作都可以在O(logn)时间范围内完成。另外，基于红黑树的map在通过迭代器遍历时，得到的是key按序排列后的结果，这点特性在很多操作中非常方便。

面试时候现场写红黑树代码的概率几乎为0，但是红黑树一些基本概念还是需要掌握的。

1. 它是二叉排序树（继承二叉排序树特显）：
    - 若左子树不空，则左子树上所有结点的值均小于或等于它的根结点的值。
    - 若右子树不空，则右子树上所有结点的值均大于或等于它的根结点的值。
    - 左、右子树也分别为二叉排序树。
2. 它满足如下几点要求：
    - 树中所有节点非红即黑。
    - 根节点必为黑节点。
    - 红节点的子节点必为黑（黑节点子节点可为黑）。
    - 从根到NULL的任何路径上黑结点数相同。

3. 查找时间一定可以控制在O(logn)。

4. 红黑树的节点定义如下：

```
enum Color {
    RED = 0,
    BLACK = 1
};
struct RBTreeNode {
    struct RBTreeNode*left, *right, *parent;
    int key;
    int data;
    Color color;
};
```

所以对红黑树的操作需要满足两点：1.满足二叉排序树的要求；2.满足红黑树自身要求。通常在找到节点通过和根节点比较找到插入位置之后，还需要结合红黑树自身限制条件对子树进行左旋和右旋。

相比于AVL树，红黑树平衡性要稍微差一些，不过创建红黑树时所需的旋转操作也会少很多。相比于最简单的BST，BST最差情况下查找的时间复杂度会上升至O(n)，而红黑树最坏情况下查找效率依旧是O(logn)。所以说红黑树之所以能够在STL及Linux内核中被广泛应用就是因为其折中了两种方案，既减少了树高，又减少了建树时旋转的次数。

从红黑树的定义来看，红黑树从根到NULL的每条路径拥有相同的黑节点数（假设为n），所以最短的路径长度为n（全为黑节点情况）。因为红节点不能连续出现，所以路径最长的情况就是插入最多的红色节点，在黑节点数一致的情况下，最可观的情况就是黑红黑红排列……最长路径不会大于2n，这里路径长就是树高。

set

编译及调试" class="reference-link">编译及调试

编译

预处理

- 展开所有的宏定义，完成字符常量替换。
- 处理条件编译语句，通过是否具有某个宏来决定过滤掉哪些代码。
- 处理#include指令，将被包含的文件插入到该指令所在位置。
- 过滤掉所有注释语句。

- 添加行号和文件名标识。
- 保留所有#pragma编译器指令。

编译

- 词法分析。
- 语法分析。
- 语义分析。
- 中间语言生成。
- 目标代码生成与优化。

链接

各个源代码模块独立的被编译，然后将他们组装起来成为一个整体，组装的过程就是链接。被链接的各个部分本本身就是二进制文件，所以在被链接时需要将所有目标文件的代码段拼接在一起，然后将所有对符号地址的引用加以修正。

- 静态链接

  静态链接最简单的情况就是在编译时和静态库链接在一起成为完整的可执行程序。这里所说的静态库就是对多个目标文件（.o）文件的打包，通常静态链接的包名为lib**.a，静态链接所有被用到的目标文件都会复制到最终生成的可执行目标文件中。这种方式的好处是在运行时，可执行目标文件已经完全装载完毕，只要按指令序执行即可，速度比较快，但缺点也有很多，在讲动态链接时会比较一下。

  既然静态链接是对目标文件的打包，这里介绍些打包命令。

  ```
  1.  gcc -c test1.c    // 生成test1.o
  2.  gcc -c test2.c    // 生成test2.c
  3.  ar cr libtest.a test1.o test2.o
  ```

  首先编译得到test1.o和test2.o两个目标文件，之后通过ar命令将这两个文件打包为.a文件，文件名格式为lib + 静态库名 + .a后缀。在生成可执行文件需要使用到它的时候只需要在编译时加上即可。需要注意的是，使用静态库时加在最后的名字不是libtest.a，而是l + 静态库名。

  ```
  1.  gcc -o main main.c -ltest
  ```

- 动态链接

  静态链接发生于编译阶段，加载至内存前已经完整，但缺点是如果多个程序都需要使用某个静态

库，则该静态库会在每个程序中都拷贝一份，非常浪费内存资源，所以出现了动态链接的方式来解决这个问题。

动态链接在形式上倒是和静态链接非常相似，首先也是需要打包，打包成动态库，不过文件名格式为lib + 动态库名 + .so后缀。不过动态库的打包不需要使用ar命令，gcc就可以完成，但要注意在编译时要加上-fPIC选项，打包时加上-shared选项。

```
1.  gcc -fPIC -c test1.c
2.  gcc -fPIC -c test2.c
3.  gcc -shared test1.o test2.o -o libtest.so
```

使用动态链接的用法也和静态链接相同。

```
1.  gcc -o main main.c -ltest
```

如果仅仅像上面的步骤是没有办法正常使用库的，我们可以通过加-Lpath指定搜索库文件的目录（-L.表示当前目录），默认情况下会到环境变量LD_LIBRARY_PATH指定的目录下搜索库文件，默认情况是/usr/lib，我们可以将库文件拷贝到那个目录下再链接。

比较静态库和动态库我们可以得到二者的优缺点。

- 动态库运行时会先检查内存中是否已经有该库的拷贝，若有则共享拷贝，否则重新加载动态库（C语言的标准库就是动态库）。静态库则是每次在编译阶段都将静态库文件打包进去，当某个库被多次引用到时，内存中会有多份副本，浪费资源。
- 动态库另一个有点就是更新很容易，当库发生变化时，如果接口没变只需要用新的动态库替换掉就可以了。但是如果是静态库的话就需要重新被编译。
- 不过静态库也有优点，主要就是静态库一次性完成了所有内容的绑定，运行时就不必再去考虑链接的问题了，执行效率会稍微高一些。

makefile编写

对于大的工程通常涉及很多头文件和源文件，编译起来很很麻烦，makefile正是为了自动化编译产生的，makefile像是编译说明书，指示编译的步骤和条件，之后被make命令解释。

- 基本规则

```
1.  A:B
2.  (tab)<command>
```

其中A是语句最后生成的文件，B是生成A所依赖的文件，比如生成test.o依赖于test.c和test.h，则写成test.o:test.c test.h。接下来一行的开头必须是tab，再往下就是实际的命令了，比如gcc -c test.c -o test.o。

- 变量

  makefile的书写非常像shell脚本，可以在文件中定义”变量名 = 变量值”的形式，之后需要使用这个变量时只需要写一个$符号加上变量名即可，当然，和shell一样，最好用()包裹起语句来。

链接

符号解析

- 可重定位目标文件

  对于独立编译的可重定位目标文件，其ELF文件格式包括ELF头（指定文件大小及字节序）、.text（代码段）、.rodata（只读数据区）、.data（已初始化数据区）、.bss（未初始化全局变量）、.symtab（符号表）等，其中链接时最需要关注的就是符号表。每个可重定位目标文件都有一张符号表，它包含该模块定义和引用的符号的信息，简而言之就是我们在每个模块中定义和引用的全局变量（包括定义在本模块的全局变量、静态全局变量和引用自定义在其他模块的全局变量）需要通过一张表来记录，在链接时通过查表将各个独立的目标文件合并成一个完整的可执行文件。

- 解析符号表

  解析符号引用的目的是将每个引用与可重定位目标文件的符号表中的一个符号定义联系起来。

重定位

- 合并节

  多个可重定位目标文件中相同的节合并成一个完整的聚合节，比如多个目标文件的.data节合并成可执行文件的.data节。链接器将运行时存储地址赋予每个节，完成这步每条指令和全局变量都有运行时地址了。

- 重定位符号引用

  这步修改全部代码节和数据节对每个符号的符号引用，使其指向正确的运行时地址。局部变量可以通过进栈、出栈临时分配，但全局变量（”符号”）的位置则是在各个可重定位目标文件中预留好的。通过上一步合并节操作后，指令中所有涉及符号的引用都会通过一定的寻址方式来定位该符号，比如相对寻址、绝对寻址等。

可执行目标文件

- ELF头部

  描述文件总体格式，并且包括程序的入口点（entry point），也就是程序运行时执行的第一条指令地址。

- 段头部表

描述了可执行文件数据段、代码段等各段的大小、虚拟地址、段对齐、执行权限等。实际上通过段头部表描绘了虚拟存储器运行时存储映像，比如每个UNIX程序的代码段总是从虚拟地址Ox0804800开始的。

- 其他段

  和可重定位目标文件各段基本相同，但完成了多个节的合并和重定位工作。

加载

- 克隆

  新程序的执行首先需要通过父进程外壳通过fork得到一个子进程，该子进程除了pid等标识和父进程不同外其他基本均与父进程相同。

- 重新映射

  当子进程执行execve系统调用时会先清空子进程现有的虚拟存储器段（简而言之就是不再映射到父进程的各个段），之后重新创建子进程虚拟存储器各段和可执行目标文件各段的映射。这个阶段我们可以理解为对复制来的父进程页表进程重写，映射到外存中可执行文件的各个段。

- 虚页调入

  加载过程并没有实际将磁盘中可执行文件调入内存，所做的工作紧紧是复制父进程页表、清空旧页表、建立新页表映射工作。之后加载器跳转到入口地址_start开始执行程序，接下来的过程需要配合虚拟存储器来完成。CPU获得指令的虚拟地址后，若包含该指令或数据的页尚未调入内存则将其从外存中调入，调入内存后修改页表得到虚拟页号和物理页号的对应关系。之后重新取同一条指令或数据时因该页已经被调入内存，所以通过虚拟地址得到虚拟页号，虚拟页号通过查页表可以得到物理页号，通过物理页号 + 页内偏移得到具体的物理地址，此时可以通过物理地址取得想要的数据。

网络编程基础

网络编程基础

常见问题

Socket API

1. 网络编程一般步骤？
   - TCP:
     - 服务端: socket -> bind -> listen -> accept -> recv/send -> close。
     - 客户端: socket -> connect -> send/recv -> close。
   - UDP:
     - 服务端: socket -> bind -> recvfrom/sendto -> close。
     - 客户端: socket -> sendto/recvfrom -> close。
2. send、sendto区别，recv、recvfrom区别？

TCP/UDP

1. TCP和UDP区别？
   - TCP面向连接（三次握手），通信前需要先建立连接；UDP面向无连接，通信前不需要连接。

- TCP通过序号、重传、流量控制、拥塞控制实现可靠传输；UDP不保障可靠传输，尽最大努力交付。
- TCP面向字节流传输，因此可以被分割并在接收端重组；UDP面向数据报传输。

2. TCP为什么不是两次握手而是三次？

- 如果仅两次连接可能出现一种情况：客户端发送完连接报文（第一次握手）后由于网络不好，延时很久后报文到达服务端，服务端接收到报文后向客户端发起连接（第二次握手）。此时客户端会认定此报文为失效报文，但在两次握手情况下服务端会认为已经建立起了连接，服务端会一直等待客户端发送数据，但因为客户端会认为服务端第二次握手的回复是对失效请求的回复，不会去处理。这就造成了服务端一直等待客户端数据的情况，浪费资源。

3. TCP为什么挥手是四次而不是三次？

- TCP是全双工的，它允许两个方向的数据传输被独立关闭。当主动发起关闭的一方关闭连接之后，TCP进入半关闭状态，此时主动方可以只关闭输出流。
- 之所以不是三次而是四次主要是因为被动关闭方将”对主动关闭报文的确认”和”关闭连接”两个操作分两次进行。
- “对主动关闭报文的确认”是为了快速告知主动关闭方，此关闭连接报文已经收到。此时被动方不立即关闭连接是为了将缓冲中剩下的数据从输出流发回主动关闭方（主动方接收到数据后同样要进行确认），因此要把”确认关闭”和”关闭连接”分两次进行。
- **Linux**的**close**实际上是同时关闭输入流和输出流，并不是我们常说的四次握手。半关闭函数为**shutdown**，它可以用来断开某个具体描述符的**TCP**输入流或输出流。

4. 为什么要有TIME_WAIT状态，TIME_WAIT状态过多怎么解决？

- 主动关闭连接一方在发送对被动关闭方关闭连接的确认报文时，有可能因为网络状况不佳，被动关闭方超时未能收到此报文而重发断开连接（FIN）报文，此时如果主动方不等待而是直接进入CLOSED状态，则接收到被动关闭方重发的断开连接的报文会触发RST分组而非ACK分组，当被动关闭一方接收到RST后会认为出错了。所以说处于TIME_WAIT状态就是为了在重新收到断开连接分组情况下进行确认。
- 解决方法：
  - 可以通过修改sysctl中TIME_WAIT时间来减少此情况（HTTP 1.1也可以减少此状态）。
  - 利用SO_LINGER选项的强制关闭方式，发RST而不是FIN，来越过TIMEWAIT状态，直接进入CLOSED状态。

5. TCP建立连接及断开连接是状态转换？

- 客户端：SYN_SENT -> ESTABLISHED -> FIN_WAIT_1 -> FIN_WAIT_2 -> TIME_WAIT。
- 服务端：LISTEN -> SYN_RCVD -> ESTABLISHED -> CLOSE_WAIT -> LAST_ACK -> CLOSED。

6. TCP流量控制和拥塞控制的实现？

- 流量控制：TCP采用大小可变的滑动窗口进行流量控制。窗口大小的单位是字节，在TCP报文段首部的窗口字段写入的数值就是当前给对方设置的发送窗口数值的上限，发送窗口在连接建立时由双方商定。但在通信的过程中，接收端可根据自己的资源情况，随时动态地调整对方的发送窗口上限值。
- 拥塞控制：网络拥塞现象是指到达通信子网中某一部分的分组数量过多，使得该部分网络来不及处理，以致引起这部分乃至整个网络性能下降的现象。严重时甚至会导致网络通信业务陷入停顿，即出现死锁现象。拥塞控制是处理网络拥塞现象的一种机制。

7. TCP重传机制？

- 滑动窗口机制，确立收发的边界，能让发送方知道已经发送了多少、尚未确认的字节数、尚待发送的字节数；让接收方知道已经确认收到的字节数。
- 选择重传，用于对传输出错的序列进行重传。

8. 三次握手过程？

- 主动建立连接方A的TCP向主机B发出连接请求报文段，其首部中的SYN(同步)标志位应置为1，表示想与目标主机B进行通信，并发送一个同步序列号x进行同步，表明在后面传送数据时的第一个数据字节的序号是x + 1。SYN同步报文会指明客户端使用的端口以及TCP连接的初始序号。
- 接收连接方B的TCP收到连接请求报文段后，如同意则发回确认。在确认报中应将ACK位和SYN位置1，表示客户端的请求被接受。确认号应为x + 1，同时也为自己选择一个序号y。
- 主动方A的TCP收到目标主机B的确认后要向目标主机B给出确认，其ACK置1，确认号为y + 1，而自己的序号为x + 1。

9. 四次挥手过程？

- 主动关闭主机A的应用进程先向其TCP发出连接释放请求，并且不再发送数据。TCP通知对方要释放从A到B这个方向的连接，将发往主机B的TCP报文段首部的终止比特FIN置1，其序号x等于前面已传送过的数据的最后一个字节的序号加1。
- 被动关闭主机B的TCP收到释放连接通知后即发出确认，其序号为y，确认号为x + 1，同时通知高层应用进程，这样，从A到B的连接就释放了，连接处于半关闭状态。但若主机B还有一些数据要发送主机A，则可以继续发送。主机A只要正确收到数据，仍应向主机B发送确

认。

- 若主机B不再向主机A发送数据，其应用进程就通知TCP释放连接。主机B发出的连接释放报文段必须将终止比特FIN和确认比特ACK置1，并使其序号仍为y，但还必须重复上次已发送过的ACK = x + 1。
- 主机A必须对此发出确认，将ACK置1，ACK = y + 1，而自己的序号是x + 1。这样才把从B到A的反方向的连接释放掉。主机A的TCP再向其应用进程报告，整个连接已经全部释放。

I/O模型

1. 阻塞和非阻塞I/O区别？
   - 如果内核缓冲没有数据可读时，read()系统调用会一直等待有数据到来后才从阻塞态中返回，这就是阻塞I/O。
   - 非阻塞I/O在遇到上述情况时会立即返回给用户态进程一个返回值，并设置errno为EAGAIN。
   - 对于往缓冲区写的操作同理。
2. 同步和异步区别？
   - 同步I/O指处理I/O操作的进程和处理I/O操作的进程是同一个。
   - 异步I/O中I/O操作由操作系统完成，并不由产生I/O的用户进程执行。
3. Reactor和Proactor区别？
   - Reactor模式已经是同步I/O，处理I/O操作的依旧是产生I/O的程序；Proactor是异步I/O，产生I/O调用的用户进程不会等待I/O发生，具体I/O操作由操作系统完成。
   - 异步I/O需要操作系统支持，Linux异步I/O为AIO，Windows为IOCP。
4. epoll和select及poll区别？
   - 文件描述符数量限制：select文件描述符数量受到限制，最大为2048（FD_SETSIZE），可重编内核修改但治标不治本；poll没有最大文件描述符数量限制；epoll没有最大文件描述符数量限制。
   - 检查机制：select和poll会以遍历方式（轮询机制）检查每一个文件描述符以确定是否有I/O就绪，每次执行时间会随着连接数量的增加而线性增长；epoll则每次返回后只对活跃的文件描述符队列进行操作（每个描述符都通过回调函数实现，只有活跃的描述符会调用回调函数并添加至队列中）。当大量连接是非活跃连接时**epoll**相对于**select**和**poll**优势比较大，若大多为活跃连接则效率未必高（设计队列维护及红黑树创建）

- 数据传递方式：select和poll需要将FD_SET在内核空间和用户空间来回拷贝；epoll则避免了不必要的数据拷贝。

5. epoll中ET和LT模式的区别与实现原理？

   - LT：默认工作方式，同时支持阻塞I/O和非阻塞I/O，LT模式下，内核告知某一文件描述符读、写是否就绪了，然后你可以对这个就绪的文件描述符进行I/O操作。如果不作任何操作，内核还是会继续通知。这种模式编程出错误可能性较小但由于重复提醒，效率相对较低。传统的select、poll都是这种模型的代表。

   - ET：高速工作方式（因为减少了epoll_wait触发次数），适合高并发，只支持非阻塞I/O，ET模式下，内核告知某一文件描述符读、写是否就绪了，然后他假设已经知道该文件描述符是否已经就绪，内核不会再为这个文件描述符发更多的就绪通知（epoll_wait不会返回），直到某些操作导致文件描述符状态不再就绪。

6. ET模式下要注意什么（如何使用ET模式）？

   - 对于读操作，如果read没有一次读完buff数据，下一次将得不到就绪通知（ET特性），造成buff中数据无法读出，除非有新数据到达。

     - 解决方法：将套接字设置为非阻塞，用while循环包住read，只要buff中有数据，就一直读。一直读到产生EAGIN错误。

   - 对于写操作主要因为ET模式下非阻塞需要我们考虑如何将用户要求写的数据写完。

     - 解决方法：只要buff还有空间且用户请求写的数据还未写完，就一直写。

操作系统

1. Linux下进程间通信方式？

   - 管道：

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

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

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

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

- 套接字：适用于不同机器间进程通信，在本地也可作为两个进程通信的方式。
- 信号：用于通知接收进程某个事件已经发生，比如按下ctrl + C就是信号。
- 信号量：信号量是一个计数器，可以用来控制多个进程对共享资源的访问。它常作为一种锁机制，实现进程、线程的对临界区的同步及互斥访问。

2. Linux下同步机制？
   - POSIX信号量：可用于进程同步，也可用于线程同步。
   - POSIX互斥锁 + 条件变量：只能用于线程同步。
3. 线程和进程的区别？
   - 调度：线程是调度的基本单位（PC，状态码，通用寄存器，线程栈及栈指针）；进程是拥有资源的基本单位（打开文件，堆，静态区，代码段等）。
   - 并发性：一个进程内多个线程可以并发（最好和CPU核数相等）；多个进程可以并发。
   - 拥有资源：线程不拥有系统资源，但一个进程的多个线程可以共享隶属进程的资源；进程是拥有资源的独立单位。
   - 系统开销：线程创建销毁只需要处理PC值，状态码，通用寄存器值，线程栈及栈指针即可；进程创建和销毁需要重新分配及销毁task_struct结构。
4. 介绍虚拟内存？
5. 内存分配及碎片管理？
6. 有很多小的碎片文件怎么处理？

Linux

1. fork系统调用？
2. 什么场景用共享内存，什么场景用匿名管道？
3. 有没有用过开源的cgi框架？
4. epoll和select比有什么优势有什么劣势，epoll有什么局限性？
   - epoll优势：1. 没有描述符数量限制；2. 通过回调代替轮询；3. 内存映射代替数据在用户和内核空间来回拷贝。
   - epoll劣势（局限性）：select可以跨平台，epoll只能在Linux上使用。

5. 线程（POSIX）锁有哪些？

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

TKeed

1. 项目整体架构是什么？请求怎么进来？处理完怎么出去？

- 整体架构为：I/O多路复用 + 非阻塞I/O + 线程池，即Reactor反应堆模型。
- 处理流程：
  - 创建监听描述符并在epoll中注册。
  - 监听到新请求，epoll从阻塞中返回并建立新连接。
  - 将新建的连接描述符在epoll中注册。
  - 当某个连接接收到用户请求数据时，将任务投放到线程池任务队列中。

        - 工作线程被条件变量（任务队列不为空）唤醒，并互斥访问线程池。
        - 得到任务的线程完成解析及响应。
            - 工作线程执行函数为do_request，参数即为task结构。
                - 每个task结构在建立连接是被初始化，包含描述符、缓冲区等信息是，并在do_request执行时记录解析结果及状态。

2. 在做压测时，机器配置是什么样的？数据如何？
    - 本地测试。
        - 四核i5处理器 + 128G固态硬盘。
3. 为了QPS（Query per second, 1秒内完成的请求数量）更高可以做哪些改进？
    - 对请求结果做缓存。
    - 多次搜索请求采用异步I/O，改串行为并行。
    - 调整并发线程数量（通常和CPU核心数相同）。
4. 有没有注意到压测时内存，CPU，I/O指标？
    - 压测同时打开top -H -p pid查看CPU，I/O，内存信息。
5. 压测时有没有见过TIME_WAIT？怎么样会见到？怎么解决？
    - 当服务端关闭连接时会产生TIME_WAIT。
    - 解决方案：
        - HTTP 1.1在同一个TCP连接上尽量传输更多数据。
        - 通过修改sysctl配置减小TIME_WAIT时间。
6. 是会主动关闭还是会等待客户端关闭连接?
    - 服务端会在完成请求之后关闭连接。
7. 写一个Server需要注意哪些问题？
    - 只支持request/response，除此之外是否需要支持cgi。
    - 并发量，QPS，资源占用（内存，CPU，I/O，网络流量等）。
        - CPU占用是否过高。

        - 内存是否泄露。

8. 项目中遇到什么困难，你是如何解决的？
    - CPU占用过高。
    - 压测时，每次最后会挂掉。
9. 做这个项目的目的是什么？
0. 定时器是如何实现的？里面放了有多少个连接（怎么确定大小）？谁去取超时的连接？检查超时之后还会继续检查吗，还是检查完之后就断了？
1. 如果发生超时，在关闭连接时同时又收到了新的数据怎么办？
2. 用什么数据结构存放url，怎么解析的？
    - 使用tk_request_t结构中buff读取用户请求，buff为循环缓冲（8192 Bytes）。
    - 每次进入while循环时读取用户请求到buff中循环队列尾位置（plast），之后解析用户请求并响应。
    - 支持HTTP 1.1，只要有数据就读取 -> 解析 -> 响应。

实习经历

1. 介绍一下上网行为管理这个系统？
2. 介绍一下格林威治云平台做哪些任务？
3. 改变数据获取方式及校验数据一致性？
4. WTGGroup模块做什么用的？

数据结构

1. 层序遍历二叉树？
2. map和hashmap的区别是什么？
3. Hash发生冲突时怎么处理？
4. hashmap的时间复杂度是多少？map的时间复杂度？
5. 优先队列时间复杂度？

操作系统

操作系统

> 面向进程和线程学习操作系统。

目录

内容

进程线程模型" class="reference-link">进程线程模型

线程和进程的概念已经在操作系统书中被翻来覆去讲了很多遍。很多概念虽然都是套话，但没能理解透其中深意会导致很多内容理解不清晰。对于进程和线程的理解和把握可以说基本奠定了对系统的认知和把控能力。其核心意义绝不仅仅是“线程是调度的基本单位，进程是资源分配的基本单位”这么简单。

多线程

我们这里讨论的是用户态的多线程模型，同一个进程内部有多个线程，所有的线程共享同一个进程的内存空间，进程中定义的全局变量会被虽有的线程共享，比如有全局变量`int i = 10`，这一进程中所有并发运行的线程都可以读取和修改这个`i`的值，而多个线程被CPU调度的顺序又是不可控的，所以对临界资源的访问尤其需要注意安全。我们必须知道，做一次简单的`i = i + 1`在计算机中并不是原子操作，涉及内存取数，计算和写入内存几个环节，而线程的切换有可能发生在上述任何一个环节中间，所以不同的操作顺序很有可能带来意想不到的结果。

但是，虽然线程在安全性方面会引入许多新挑战，但是线程带来的好处也是有目共睹的。首先，原先

顺序执行的程序（暂时不考虑多进程）可以被拆分成几个独立的逻辑流，这些逻辑流可以独立完成一些任务（最好这些任务是不相关的）。比如QQ可以一个线程处理聊天一个线程处理上传文件，两个线程互不干涉，在用户看来是同步在执行两个任务，试想如果线性完成这个任务的话，在数据传输完成之前用户聊天被一直阻塞会是多么尴尬的情况。

对于线程，我认为弄清以下两点非常重要：

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

另外，我们通常只会去说同一进程的多个线程共享进程的资源，但是每个线程特有的部分却很少提及，除了标识线程的tid，每个线程还有自己独立的栈空间，线程彼此之间是无法访问其他线程栈上内容的。而作为处理机调度的最小单位，线程调度只需要保存线程栈、寄存器数据和PC即可，相比进程切换开销要小很多。

线程相关接口不少，主要需要了解各个参数意义和返回值意义。

1. 线程创建和结束

    - 背景知识：

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

    - 相关接口：

        - 创建线程：int pthread_create(pthread_t *pthread, const pthread_attr_t* attr, void *(start_routine)(void *), void* agr);

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

            - pthread：用来返回线程的tid，*pthread值即为tid，类型pthread_t == unsigned long int。
            - attr：指向线程属性结构体的指针，用于改变所创线程的属性，填NULL使用默认值。
            - start_routine：线程执行函数的首地址，传入函数指针。
            - arg：通过地址传递来传递函数参数，这里是无符号类型指针，可以传任意类型变量的地址，在被传入函数中先强制类型转换成所需类型即可。

        - 获得线程ID：pthread_t pthread_self();

调用时，会打印线程ID。

- 等待线程结束：int pthread_join(pthread_t tid, void** retval);

  主线程调用，等待子线程退出并回收其资源，类似于进程中wait/waitpid回收僵尸进程，调用pthread_join的线程会被阻塞。

  - tid：创建线程时通过指针得到tid值。
  - retval：指向返回值的指针。

- 结束线程：pthread_exit(void *retval);

  子线程执行，用来结束当前线程并通过retval传递返回值，该返回值可通过pthread_join获得。

  - retval：同上。

- 分离线程：int pthread_detach(pthread_t tid);

  主线程、子线程均可调用。主线程中pthread_detach(tid)，子线程中pthread_detach(pthread_self())，调用后和主线程分离，子线程结束时自己立即回收资源。

  - tid：同上。

2. 线程属性值修改

- 背景知识：

  线程属性对象类型为pthread_attr_t，结构体定义如下：

```
typedef struct{
    int etachstate;    // 线程分离的状态
    int schedpolicy;    // 线程调度策略
    struct sched_param schedparam;    // 线程的调度参数
    int inheritsched;    // 线程的继承性
    int scope;    // 线程的作用域
    // 以下为线程栈的设置
    size_t guardsize;    // 线程栈末尾警戒缓冲大小
    int stackaddr_set;    // 线程的栈设置
    void *    stackaddr;    // 线程栈的位置
    size_t stacksize;    // 线程栈大小
}pthread_arrt_t;
```

- 相关接口：

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

3. 线程同步

- 详见同步互斥专题

多进程

每一个进程是资源分配的基本单位。进程结构由以下几个部分组成：代码段、堆栈段、数据段。代码段是静态的二进制代码，多个程序可以共享。实际上在父进程创建子进程之后，父、子进程除了pid外，几乎所有的部分几乎一样，子进程创建时拷贝父进程PCB中大部分内容，而PCB的内容实际上是各种数据、代码的地址或索引表地址，所以复制了PCB中这些指针实际就等于获取了全部父进程可访问数据。所以简单来说，创建新进程需要复制整个PCB，之后操作系统将PCB添加到进程核心堆栈底部，这样就可以被操作系统感知和调度了。

父、子进程共享全部数据，但并不是说他们就是对同一块数据进行操作，子进程在读写数据时会通过写时复制机制将公共的数据重新拷贝一份，之后在拷贝出的数据上进行操作。如果子进程想要运行自己的代码段，还可以通过调用execv()函数重新加载新的代码段，之后就和父进程独立开了。我们在shell中执行程序就是通过shell进程先fork()一个子进程再通过execv()重新加载新的代码段的过程。

1. 进程创建与结束

- 背景知识：

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

- 相关接口：

  - 创建进程：pid_t fork(void);

    返回值：出错返回-1；父进程中返回pid > 0；子进程中pid == 0

  - 结束进程：void exit(int status);

    - status是退出状态，保存在全局变量中S?，通常0表示正常退出。

  - 获得PID：pid_t getpid(void);

    返回调用者pid。

  - 获得父进程PID：pid_t getppid(void);

返回父进程pid。

- 其他补充：

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

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

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

  - 异常退出方式：abort()、终止信号。

2. 僵尸进程、孤儿进程

- 背景知识：

  父进程在调用fork接口之后和子进程已经可以独立开，之后父进程和子进程就以未知的顺序向下执行（异步过程）。所以父进程和子进程都有可能先执行完。当父进程先结束，子进程此时就会变成孤儿进程，不过这种情况问题不大，孤儿进程会自动向上被init进程收养，init进程完成对状态收集工作。而且这种过继的方式也是守护进程能够实现的因素。如果子进程先结束，父进程并未调用wait或者waitpid获取进程状态信息，那么子进程描述符就会一直保存在系统中，这种进程称为僵尸进程。

- 相关接口：

  - 回收进程（1）：pid_t wait(int *status);

    一旦调用wait()，就会立即阻塞自己，wait()自动分析某个子进程是否已经退出，如果找到僵尸进程就会负责收集和销毁，如果没有找到就一直阻塞在这里。

    - status：指向子进程结束状态值。

    - 回收进程（2）：pid_t waitpid(pid_t pid, int *status, int options);

      返回值：返回pid：返回收集的子进程id。返回-1：出错。返回0：没有被手机的子进程。

      - pid：子进程识别码，控制等待哪些子进程。

        a. pid < -1，等待进程组识别码为pid绝对值的任何进程。

        b. pid = -1，等待任何子进程。

c. pid = 0，等待进程组识别码与目前进程相同的任何子进程。

d. pid > 0，等待任何子进程识别码为pid的子进程。

- status：指向返回码的指针。
- options：选项决定父进程调用waitpid后的状态。

a. options = WNOHANG，即使没有子进程退出也会立即返回。

b. options = WUNYRACED，子进程进入暂停马上返回，但结束状态不予理会。

3. 守护进程

- 背景知识：

  守护进程是脱离终端并在后台运行的进程，执行过程中信息不会显示在终端上并且也不会被终端发出的信号打断。

- 操作步骤：
  - 创建子进程，父进程退出：fork() + if(pid > 0){exit(0);}，使子进程称为孤儿进程被init进程收养。
  - 在子进程中创建新会话：setsid()。
  - 改变当前目录结构为根：chdir(“/“)。
  - 重设文件掩码：umask(0)。
  - 关闭文件描述符：for(int i = 0; i < 65535; ++i){close(i);}。

4. Linux进程控制

- 进程地址空间（地址空间）

  虚拟存储器为每个进程提供了独占系统地址空间的假象。尽管每个进程地址空间内容不尽相同，但是他们的都有相似的结构。X86 Linux进程的地址空间底部是保留给用户程序的，包括文本、数据、堆、栈等，其中文本区和数据区是通过存储器映射方式将磁盘中可执行文件的相应段映射至虚拟存储器地址空间中。有一些”敏感”的地址需要注意下，对于32位进程来说，代码段从0x08048000开始。从0xC0000000开始到0xFFFFFFFF是内核地址空间，通常情况下代码运行在用户态（使用0x00000000 ~ 0xC0000000的用户地址空间），当发生系统调用、进程切换等操作时CPU控制寄存器设置模式位，进入内和模式，在该状态（超级用户模式）下进程可以访问全部存储器位置和执行全部指令。也就说32位进程的地址空间都是4G，但用户态下只能访问低3G的地址空间，若要访问3G ~ 4G的地址空间则只有进入内核态才行。

- 进程控制块（处理机）

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

- 上下文切换

  内核管理所有进程控制块，而进程控制块记录了进程全部状态信息。每一次进程调度就是一次上下文切换，所谓的上下文本质上就是当前运行状态，主要包括通用寄存器、浮点寄存器、状态寄存器、程序计数器、用户栈和内核数据结构（页表、进程表、文件表）等。进程执行时刻，内核可以决定抢占当前进程并开始新的进程，这个过程由内核调度器完成，当调度器选择了某个进程时称为该进程被调度，该过程通过上下文切换来改变当前状态。一次完整的上下文切换通常是进程原先运行于用户态，之后因系统调用或时间片到切换到内核态执行内核指令，完成上下文切换后回到用户态，此时已经切换到进程B。

线程、进程比较

关于进程和线程的区别这里就不一一罗列了，主要对比下线程和进程操作中主要的接口。

- `fork()`和`pthread_create()`

  负责创建。调用`fork()`后返回两次，一次标识主进程一次标识子进程；调用`pthread_create()`后得到一个可以独立执行的线程。

- `wait()`和`pthread_join()`

  负责回收。调用`wait()`后父进程阻塞；调用`pthread_join()`后主线程阻塞。

- `exit()`和`pthread_exit()`

  负责退出。调用`exit()`后调用进程退出，控制权交给系统；调用`pthread_exit()`后线程退出，控制权交给主线程。

进程间通信" class="reference-link">进程间通信

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

管道

- 无名管道
  - 无名管道特点：
    - 无名管道是一种特殊的文件，这种文件只存在于内存中。

        - 无名管道只能用于父子进程或兄弟进程之间，必须用于具有亲缘关系的进程间的通信。
        - 无名管道只能由一端向另一端发送数据，是半双工方式，如果双方需要同时收发数据需要两个管道。
    - 相关接口：
        - `int pipe(int fd[2]);`
            - `fd[2]`：管道两端用`fd[0]`和`fd[1]`来描述，读的一端用`fd[0]`表示，写的一端用`fd[1]`表示。通信双方的进程中写数据的一方需要把`fd[0]`先`close`掉，读的一方需要先把`fd[1]`给`close`掉。
- 有名管道：
    - 有名管道特点：
        - 有名管道是`FIFO`文件，存在于文件系统中，可以通过文件路径名来指出。
        - 无名管道可以在不具有亲缘关系的进程间进行通信。
    - 相关接口：
        - `int mkfifo(const char *pathname, mode_t mode);`
            - `pathname`：即将创建的`FIFO`文件路径，如果文件存在需要先删除。
            - `mode`：和`open()`中的参数相同。

消息队列

共享内存

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

- 相关接口
    - 创建共享内存：`int shmget(key_t key, int size, int flag);`

      成功时返回一个和`key`相关的共享内存标识符，失败范湖范围`-1`。
        - `key`：为共享内存段命名，多个共享同一片内存的进程使用同一个`key`。
        - `size`：共享内存容量。
        - `flag`：权限标志位，和`open`的`mode`参数一样。

    - 连接到共享内存地址空间: void *shmat(int shmid, void* addr, int flag);

      返回值即共享内存实际地址。

      - shmid: shmget()返回的标识。
      - addr: 决定以什么方式连接地址。
      - flag: 访问模式。

    - 从共享内存分离: int shmdt(const void *shmaddr);

      调用成功返回0，失败返回-1。

      - shmaddr: 是shmat()返回的地址指针。

- 其他补充

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

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

信号量

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

- 相关接口

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

      创建成功返回信号量标识符，失败返回-1。

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

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

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

sembuf变量传递进去。

struct sembuf结构体定义如下：

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

成功返回信号量标识符，失败返回-1。

- semid：信号量集标识符，由semget()函数返回。
- sops：指向struct sembuf结构的指针，先设置好sembuf值再通过指针传递。
- nsops：进行操作信号量的个数，即sops结构变量的个数，需大于或等于1。最常见设置此值等于1，只完成对一个信号量的操作。

- 直接控制信号量信息：int semctl(int semid, int semnum, int cmd, union semun arg);
  - semid：信号量集标识符。
  - semnum：信号量集数组上的下标，表示某一个信号量。
  - arg：union semun类型。

辅助命令

ipcs命令用于报告共享内存、信号量和消息队列信息。

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

套接字

- 详见socket交互流程
- 详见网络I/O模型

同步互斥机制" class="reference-link">同步互斥机制

> 待补充

网络I/O模型" class="reference-link">网络I/O模型

在描述这块内容的诸多书籍中，很多都只说笼统的概念，我们将问题具体化，暂时只考虑服务器端的网络I/O情形。我们假定目前的情形是服务器已经在监听用户请求，建立连接后服务器调用read()函数等待读取用户发送过来的数据流，之后将接收到的数据打印出来。

所以服务器端简单是这样的流程：建立连接 -> 监听请求 -> 等待用户数据 -> 打印数据。我们总结网络通信中的等待：

- 建立连接时等待对方的ACK包（TCP）。
- 等待客户端请求（HTTP）。
- 输入等待：服务器用户数据到达内核缓冲区（read函数等待）。
- 输出等待：用户端等待缓冲区有足够空间可以输入（write函数等待）。

另外为了能够解释清楚网络I/O模型，还需要了解一些基础。对服务器而言，打印出用户输入的字符串（printf函数）和从网络中获取数据（read函数）需要单独来看。服务器首先accept用户连接请求后首先调用read函数等待数据，这里的read函数是系统调用，运行于内核态，使用的也是内核地址空间，并且从网络中取得的数据需要先写入到内核缓冲区。当read系统调用获取到数据后将这些数据再复制到用户地址空间的用户缓冲区中，之后返回到用户态执行printf函数打印字符串。我们需要明确两点：

- read执行在内核态且数据流先读入内核缓冲区；printf运行于用户态，打印的数据会先从内核缓冲区复制到进程的用户缓冲区，之后打印出来。
- printf函数一定是在read函数已经准备好数据之后才能执行，但read函数作为I/O操作通常需要等待而触发阻塞。调用read函数的是服务器进程，一旦被read调用阻塞，整个服务器在获取到用户数据前都不能接受任何其他用户的请求（单进程/线程）。

有了上面的基础，我们就可以介绍下面四种网路I/O模型。

阻塞式

- 阻塞表示一旦调用I/O函数必须等整个I/O完成才返回。正如上面提到的那种情形，当服务器调用了read函数之后，如果不是立即接收到数据，服务器进程会被阻塞，之后一直在等待用户数据到达，用户数据到达后首先会写进内核缓冲区，之后内核缓冲区数据复制到用户进程（服务器进程）缓冲区。完成了上述所有的工作后，才会把执行权限返回给用户（从内核态 -> 用户态）。
- 很显然，阻塞式I/O的效率实在太低，如果用户输入数据迟迟不到的话，整个服务器就会一直被阻塞（单进程/线程）。为了不影响服务器接收其他进程的连接，我们可以考虑多进程模型，这样当服务器建立连接后为连接的用户创建新线程，新线程即使是使用阻塞式I/O也仅仅是这一个线

程被阻塞，不会影响服务器等待接收新的连接。

- 多线程模型下，主线程等待用户请求，用户有请求到达时创建新线程。新线程负责具体的工作，即使是因为调用了read函数被阻塞也不会影响服务器。我们还可以进一步优化创建连接池和线程池以减小频繁调用I/O接口的开销。但新问题随之产生，每个新线程或者进程（加入使用对进程模型）都会占用大量系统资源，除此之外过多的线程和进程在调度方面开销也会大很对，所以这种模型并不适合大并发量。

非阻塞**I/O**

- 阻塞和非阻塞最大的区别在于调用I/O系统调用后，是等整个I/O过程完成再把操作权限返回给用户还是会立即返回。
- 可以使用以下语句将句柄fd设置为非阻塞I/O: `fcntl(fd, F_SETFL, O_NONBLOCK);`
- 非阻塞I/O在调用后会立即返回，用户进程对返回的返回值判断以区分是否完成了I/O。如果返回大于0表示完成了数据读取，返回值即读取的字节数；返回0表示连接已经正常断开；返回-1表示错误，接下来用户进程会不停地询问kernel是否准备完毕。
- 非阻塞I/O虽然不再会完全阻塞用户进程，但实际上由于用户进程需要不停地询问kernel是否准备完数据，所以整体效率依旧非常低，不适合做并发。

I/O多路复用（事件驱动模型）

前面已经论述了多进程、多进程模型会因为开销巨大和调度困难而导致并不能承受高并发量。但不适用这种模型的话，无论是阻塞还是非阻塞方式都会导致整个服务器停滞。

所以对于大并发量，我们需要一种代理模型可以帮助我们集中去管理所有的socket连接，一旦某个socket数据到达了就执行其对应的用户进程，I/O多路复用就是这么一种模型。Linux下I/O多路复用的系统调用有select，poll和epoll，但从本质上来讲他们都是同步I/O范畴。

1. select
    - 相关接口:

      ```
      int select (int maxfd, fd_set readfds, fd_set writefds, fd_set errorfds, struct timeval timeout);

      FD_ZERO(int fd, fd_set* fds) //清空集合

      FD_SET(int fd, fd_set* fds) //将给定的描述符加入集合

      FD_ISSET(int fd, fd_set* fds) //将给定的描述符从文件中删除

      FD_CLR(int fd, fd_set* fds) //判断指定描述符是否在集合中
      ```

    - 参数:

maxfd：当前最大文件描述符的值+1（≠ MAX_CONN）。

readfds：指向读文件队列集合（fd_set）的指针。

writefds：同上，指向读集合的指针。

writefds：同上，指向错误集合的指针。

timeout：指向timeval结构指针，用于设置超时。

- 其他：

判断和操作对象为set_fd集合，集合大小为单个进程可打开的最大文件数1024或2048（可重新编译内核修改但不建议）。

2. poll

- 相关接口：
  int poll(struct pollfd *fds, unsigned int nfds, int timeout);
- 结构体定义：
  struct pollfd{

```
int fd;    // 文件描述符
short events;    // 等到的事件
short revents;    // 实际发生的事件
```

  }
- 参数：
  fds：指向pollfd结构体数组的指针。

  nfds：pollfd数组当前已被使用的最大下标。

  timeout：等待毫秒数。
- 其他：

判断和操作对象是元素为pollfd类型的数组，数组大小自己设定，即为最大连接数。

3. epoll

- 相关接口：
  int epoll_create(int size); // 创建epoll句柄
  int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event*);
  // 事件注册函数
  int epoll_wait(int epfd, struct epoll_event events, int maxevents,

int timeout);

- 结构体定义：
struct epoll_event{

```
__uint32_t events;
epoll_data_t data;
```

};
typedef union epoll_data{

```
void *ptr;
int fd;
__uint32_t u32;
__uint64_t u64;
```

}epoll_data_t;

- 参数：

size：用来告诉内核要监听的数目。

epfd：epoll函数的返回值。

op：表示动作（EPOLL_CTL_ADD/EPOLL_CTL_FD/EPOLL_CTL_DEL）。

fd：需要监听的fd。

events：指向epoll_event的指针，该结构记录监听的事件。

maxevents：告诉内核events的大小。

timeout：超时时间（ms为单位，0表示立即返回，-1将不确定）。

4. select、poll和epoll区别

- 操作方式及效率：

select是遍历，需要遍历fd_set每一个比特位（= MAX_CONN），O(n)；poll是遍历，但只遍历到pollfd数组当前已使用的最大下标（≠ MAX_CONN），O(n)；epoll是回调，O(1)。

- 最大连接数：

select为1024/2048（一个进程打开的文件数是有限制的）；poll无上限；epoll无上限。

- fd拷贝：

  select每次都需要把fd集合从用户态拷贝到内核态；poll每次都需要把fd集合从用户态拷贝到内核态；epoll调用epoll_ctl时拷贝进内核并放到事件表中，但用户进程和内核通过mmap映射共享同一块存储，避免了fd从内核赋值到用户空间。

- 其他：

  select每次内核仅仅是通知有消息到了需要处理，具体是哪一个需要遍历所有的描述符才能找到。epoll不仅通知有I/O到来还可通过callback函数具体定位到活跃的socket，实现伪AIO。

异步**I/O**模型

- 上面三种I/O方式均属于同步I/O。
- 从阻塞式I/O到非阻塞I/O，我们已经做到了调用I/O请求后立即返回，但不停轮询的操作效率又很低，如果能够既像非阻塞I/O能够立即返回又能不一直轮询的话会更符合我们的预期。
- 之所以用户进程会不停轮询就是因为在数据准备完毕后内核不会回调用户进程，只能通过用户进程一次又一次轮询来查询I/O结果。如果内核能够在完成I/O后通过消息告知用户进程来处理已经得到的数据自然是最好的，异步I/O就是这么回事。
- 异步I/O就是当用户进程发起I/O请求后立即返回，直到内核发送一个信号，告知进程I/O已完成，在整个过程中，都没有进程被阻塞。看上去异步I/O和非阻塞I/O的区别在于：判断数据是否准备完毕的任务从用户进程本身被委托给内核来完成。这里所谓的异步只是操作系统提供的一直机制罢了。

学习计划

学习计划

> *Learning by doing.*

攻坚目标

1. 知识点覆盖训练
    - 牛客网以往错题梳理（7.10前）
    - 牛客网新题专项训练（全程）
2. 零散知识点整合
    - 单项知识点总结提炼（全程）
3. 针对性训练
    - 针对性刷往年笔试题（7.5开始，每日3套）
4. 编程、项目能力
    - LeetCode、剑指offer（7.6开始，每日3题）
    - 暑期练手项目（7月15后）

海量数据处理

海量数据处理

TOP N问题

1. 如何在海量数据中找出重复最多一个。

- 通过hash映射为小文件
- 通过hash_map统计各个小文件重读最多的并记录次数
- 对每个小文件重复最多的进行建立大根堆

1. 上亿有重数据，统计最多前N个。

- 内存存不下
  - 通过hash映射为小文件
  - 通过hash_map统计各个小文件重读最多的并记录次数
  - 对每个小文件重复最多的进行建立大根堆并重复N次取走堆顶并重建堆操作
- 内存存得下
  - 直接内存通过hash_map统计并建大根堆
  - 重复N次取走堆顶并重建堆操作

1. 海量日志数据，提取出某日访问百度次数最多的那个IP（同1）。

- 将IP % 1000映射到1000个小文件中
  - 相同IP会被映射到同一个文件

    - 不会出现累加和更大情况
- 分1000次在内存处理小文件，得到频率最大IP（使用map统计）
- 对这1000个IP建立大根堆

1. 1000w查询串统计最热门10个（同2）。

- 同上

1. 1G的文件，里面1行1个不超过16字节的词。内存限制1M，返回频数最高前100（同2）。

- 将单词 % 5000存入5000小文件
    - 平均各文件约200K
    - 对超过1M的文件继续分割直到小于200K
- 使用map统计各个词出现的频率
- 对5000词使用堆排序或归并排序

分布式TOP N问题

1. 分布在100台电脑的海量数据，统计前十。

- 各数据只出现在一台机器中
    - 先在独立机器得到前十
        - 若可以放入内存直接堆排序
        - 若不可全放入内存：哈希分块 -> map统计 -> 归总堆排
    - 再将100台计算机的TOP10组合起来堆排序
- 同一元素可同时出现在不同机器中
    - 遍历所有数据，重新hash取模，使同一个元素只出现在单独的一台电脑中，然后采用上面方法先统计每台电脑TOP10再汇总起来

快速外排序问题

1. 有10个1G文件，每行都是一个可重复用户query，按query频度排序。

- 顺序读取十个文件并采取哈希，将query写入10个文件中

- 通过hash_map(query, count)统计每个query出现次数，至少2G内存
- 通过得到的hash_map中query和query_count，对query_count排序并将重新输出到文件中，得到已排序好的文件
- 对十个文件进行归并排序（外排序）

公共数据问题

1. A,B两个文件各存放50亿url，每个为64Byte，限制内存4G找出公共url。

- 对A和B两个大文件，先通过url % 1000将数据映射到1000个文件中，单个文件大小约320M（我们只需要检查对应小文件A1 V B1……，不对应小文件不会有相同url）
- 通过hash_set统计，把A1的url存储到hash_set中，再遍历对应的B1小文件，检查是否在hash_set中，若存在则写入外存。重复循环处理对应的1000个对。

1. 1000w有重字符串，对字符串去重。

- 先hash分为多个文件
- 逐个文件检查并插入set中
- 多个set取交集

内存内TOP N问题

1. 100w个数字找出最大100个。

- 堆排序法
  - 建大根堆，取走堆顶并重建堆，重复100次
- 快排法
  - 使用快速排序划分，若某次枢纽元在后10000时（具体情况具体分析），对后10000数据排序后取前100

位图法

1. 在2.5亿数字中找出不重复的整数。

- 使用2-Bit位图法，00表示不存在，01表示出现一次，10表示出现多次，11无意义。这样只需

要1G内存。

- 或者hash划分小文件，小文件使用hash_set检查各个元素，得到的。

1. 如何在40亿数字中快速判断是否有某个数？

- 位图法标记某个数字是否存在，check标记数组。

计算机网络

计算机网络

> 重点在*TCP/IP*协议和*HTTP*协议。

目录

内容

网络层(IP)" class="reference-link">网络层(IP)

> 待补充

传输层(TCP/UDP)" class="reference-link">传输层(TCP/UDP)

1. ISO七层模型中表示层和会话层功能是什么？
   - 表示层：图像、视频编码解，数据加密。
   - 会话层：建立会话，如session认证、断点续传。
2. 描述TCP头部？
   - 序号（32bit）：传输方向上字节流的字节编号。初始时序号会被设置一个随机的初始值（ISN），之后每次发送数据时，序号值 = ISN + 数据在整个字节流中的偏移。假设A ->

B且ISN = 1024，第一段数据512字节已经到B，则第二段数据发送时序号为1024 + 512。用于解决网络包乱序问题。

- 确认号（32bit）：接收方对发送方TCP报文段的响应，其值是收到的序号值 + 1。
- 首部长（4bit）：标识首部有多少个4字节 * 首部长，最大为15，即60字节。
- 标志位（6bit）：
  - URG：标志紧急指针是否有效。
  - ACK：标志确认号是否有效（确认报文段）。用于解决丢包问题。
  - PSH：提示接收端立即从缓冲读走数据。
  - RST：表示要求对方重新建立连接（复位报文段）。
  - SYN：表示请求建立一个连接（连接报文段）。
  - FIN：表示关闭连接（断开报文段）。
- 窗口（16bit）：接收窗口。用于告知对方（发送方）本方的缓冲还能接收多少字节数据。用于解决流控。
- 校验和（16bit）：接收端用CRC检验整个报文段有无损坏。

3. 三次握手过程？
   - 第一次：客户端发含SYN位，SEQ_NUM = S的包到服务器。（客 -> SYN_SEND）
   - 第二次：服务器发含ACK，SYN位且ACK_NUM = S + 1，SEQ_NUM = P的包到客户机。（服 -> SYN_RECV）
   - 第三次：客户机发送含ACK位，ACK_NUM = P + 1的包到服务器。（客 -> ESTABLISH，服 -> ESTABLISH）
4. 四次挥手过程？
   - 第一次：客户机发含FIN位，SEQ = Q的包到服务器。（客 -> FIN_WAIT_1）
   - 第二次：服务器发送含ACK且ACK_NUM = Q + 1的包到服务器。（服 -> CLOSE_WAIT，客 -> FIN_WAIT_2）
     - 此处有等待
   - 第三次：服务器发送含FIN且SEQ_NUM = R的包到客户机。（服 -> LAST_ACK，客 -> TIME_WAIT）

        - 此处有等待

    - 第四次：客户机发送最后一个含有ACK位且ACK_NUM = R + 1的包到客户机。（服 -> CLOSED）

5. 为什么握手是三次，挥手是四次？

    - 对于握手：握手只需要确认双方通信时的初始化序号，保证通信不会乱序。（第三次握手必要性：假设服务端的确认丢失，连接并未断开，客户机超时重发连接请求，这样服务器会对同一个客户机保持多个连接，造成资源浪费。）

    - 对于挥手：TCP是双工的，所以发送方和接收方都需要FIN和ACK。只不过有一方是被动的，所以看上去就成了4次挥手。

6. TCP连接状态？

    - CLOSED：初始状态。

    - LISTEN：服务器处于监听状态。

    - SYN_SEND：客户端socket执行CONNECT连接，发送SYN包，进入此状态。

    - SYN_RECV：服务端收到SYN包并发送服务端SYN包，进入此状态。

    - ESTABLISH：表示连接建立。客户端发送了最后一个ACK包后进入此状态，服务端接收到ACK包后进入此状态。

    - FIN_WAIT_1：终止连接的一方（通常是客户机）发送了FIN报文后进入。等待对方FIN。

    - CLOSE_WAIT：（假设服务器）接收到客户机FIN包之后等待关闭的阶段。在接收到对方的FIN包之后，自然是需要立即回复ACK包的，表示已经知道断开请求。但是本方是否立即断开连接（发送FIN包）取决于是否还有数据需要发送给客户端，若有，则在发送FIN包之前均为此状态。

    - FIN_WAIT_2：此时是半连接状态，即有一方要求关闭连接，等待另一方关闭。客户端接收到服务器的ACK包，但并没有立即接收到服务端的FIN包，进入FIN_WAIT_2状态。

    - LAST_ACK：服务端发动最后的FIN包，等待最后的客户端ACK响应，进入此状态。

    - TIME_WAIT：客户端收到服务端的FIN包，并立即发出ACK包做最后的确认，在此之后的2MSL时间称为TIME_WAIT状态。

7. 解释FIN_WAIT_2，CLOSE_WAIT状态和TIME_WAIT状态？

    - FIN_WAIT_2：

        - 半关闭状态。

        - 发送断开请求一方还有接收数据能力，但已经没有发送数据能力。
    - CLOSE_WAIT状态：
        - 被动关闭连接一方接收到FIN包会立即回应ACK包表示已接收到断开请求。
        - 被动关闭连接一方如果还有剩余数据要发送就会进入CLOSED_WAIT状态。
    - TIME_WAIT状态：
        - 又叫2MSL等待状态。
        - 如果客户端直接进入CLOSED状态，如果服务端没有接收到最后一次ACK包会在超时之后重新再发FIN包，此时因为客户端已经CLOSED，所以服务端就不会收到ACK而是收到RST。所以TIME_WAIT状态目的是防止最后一次握手数据没有到达对方而触发重传FIN准备的。
        - 在2MSL时间内，同一个socket不能再被使用，否则有可能会和旧连接数据混淆（如果新连接和旧连接的socket相同的话）。

8. 解释RTO，RTT和超时重传？
    - 超时重传：发送端发送报文后若长时间未收到确认的报文则需要重发该报文。可能有以下几种情况：
        - 发送的数据没能到达接收端，所以对方没有响应。
        - 接收端接收到数据，但是ACK报文在返回过程中丢失。
        - 接收端拒绝或丢弃数据。
    - RTO：从上一次发送数据，因为长期没有收到ACK响应，到下一次重发之间的时间。就是重传间隔。
        - 通常每次重传RTO是前一次重传间隔的两倍，计量单位通常是RTT。例：1RTT，2RTT，4RTT，8RTT……
        - 重传次数到达上限之后停止重传。
    - RTT：数据从发送到接收到对方响应之间的时间间隔，即数据报在网络中一个往返用时。大小不稳定。

9. 流量控制原理？
    - 目的是接收方通过TCP头窗口字段告知发送方本方可接收的最大数据量，用以解决发送速率过快导致接收方不能接收的问题。所以流量控制是点对点控制。
    - TCP是双工协议，双方可以同时通信，所以发送方接收方各自维护一个发送窗和接收窗。

        - 发送窗：用来限制发送方可以发送的数据大小，其中发送窗口的大小由接收端返回的TCP报文段中窗口字段来控制，接收方通过此字段告知发送方自己的缓冲（受系统、硬件等限制）大小。
        - 接收窗：用来标记可以接收的数据大小。
    - TCP是流数据，发送出去的数据流可以被分为以下四部分：已发送且被确认部分 | 已发送未被确认部分 | 未发送但可发送部分 | 不可发送部分，其中发送窗 = 已发送未确认部分 + 未发但可发送部分。接收到的数据流可分为：已接收 | 未接收但准备接收 | 未接收不准备接收。接收窗 = 未接收但准备接收部分。
    - 发送窗内数据只有当接收到接收端某段发送数据的ACK响应时才移动发送窗，左边缘紧贴刚被确认的数据。接收窗也只有接收到数据且最左侧连续时才移动接收窗口。

0. 拥塞控制原理？
    - 拥塞控制目的是防止数据被过多注网络中导致网络资源（路由器、交换机等）过载。因为拥塞控制涉及网络链路全局，所以属于全局控制。控制拥塞使用拥塞窗口。
    - TCP拥塞控制算法：
        - 慢开始 & 拥塞避免：先试探网络拥塞程度再逐渐增大拥塞窗口。每次收到确认后拥塞窗口翻倍，直到达到阀值ssthresh，这部分是慢开始过程。达到阀值后每次以一个MSS为单位增长拥塞窗口大小，当发生拥塞（超时未收到确认），将阀值减为原先一半，继续执行线性增加，这个过程为拥塞避免。
        - 快速重传 & 快速恢复：略。
        - 最终拥塞窗口会收敛于稳定值。

1. 如何区分流量控制和拥塞控制？
    - 流量控制属于通信双方协商；拥塞控制涉及通信链路全局。
    - 流量控制需要通信双方各维护一个发送窗、一个接收窗，对任意一方，接收窗大小由自身决定，发送窗大小由接收方响应的TCP报文段中窗口值确定；拥塞控制的拥塞窗口大小变化由试探性发送一定数据量数据探查网络状况后而自适应调整。
    - 实际最终发送窗口 = min{流控发送窗口，拥塞窗口}。

2. TCP如何提供可靠数据传输的？
    - 建立连接（标志位）：通信前确认通信实体存在。
    - 序号机制（序号、确认号）：确保了数据是按序、完整到达。

- 数据校验（校验和）：CRC校验全部数据。
- 超时重传（定时器）：保证因链路故障未能到达数据能够被多次重发。
- 窗口机制（窗口）：提供流量控制，避免过量发送。
- 拥塞控制：同上。

3. TCP soctet交互流程？
   - 服务器：
     - 创建socket -> int socket(int domain, int type, int protocol);
       - domain：协议域，决定了socket的地址类型，IPv4为AF_INET。
       - type：指定socket类型，SOCK_STREAM为TCP连接。
       - protocol：指定协议。IPPROTO_TCP表示TCP协议，为0时自动选择type默认协议。
     - 绑定socket和端口号 -> int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
       - sockfd：socket返回的套接字描述符，类似于文件描述符fd。
       - addr：有个sockaddr类型数据的指针，指向的是被绑定结构变量。

         ```
         // IPv4的sockaddr地址结构
         struct sockaddr_in {
             sa_family_t sin_family;    // 协议类型，AF_INET
             in_port_t sin_port;    // 端口号
             struct in_addr sin_addr;    // IP地址
         };
         struct in_addr {
             uint32_t s_addr;
         }
         ```

       - addrlen：地址长度。
     - 监听端口号 -> int listen(int sockfd, int backlog);
       - sockfd：要监听的sock描述字。
       - backlog：socket可以排队的最大连接数。
     - 接收用户请求 -> int accept(int sockfd, struct sockaddr *addr,*

socklen_t addrlen);

- sockfd: 服务器socket描述字。
- addr: 指向地址结构指针。
- addrlen: 协议地址长度。
- 注: 一旦accept某个客户机请求成功将返回一个全新的描述符用于标识具体客户的TCP连接。

- 从socket中读取字符 -> ssize_t read(int fd, void *buf, size_t count);
  - fd: 连接描述字。
  - buf: 缓冲区buf。
  - count: 缓冲区长度。
  - 注: 大于0表示读取的字节数，返回0表示文件读取结束，小于0表示发生错误。
- 关闭socket -> int close(int fd);
  - fd: accept返回的连接描述字，每个连接有一个，生命周期为连接周期。
  - 注: sockfd是监听描述字，一个服务器只有一个，用于监听是否有连接；fd是连接描述字，用于每个连接的操作。

- 客户机:
  - 创建socket -> int socket(int domain, int type, int protocol);
  - 连接指定计算机 -> int connect(int sockfd, struct sockaddr* addr, socklen_t addrlen);
    - sockfd客户端的sock描述字。
    - addr: 服务器的地址。
    - addrlen: socket地址长度。
  - 向socket写入信息 -> ssize_t write(int fd, const void *buf, size_t count);
    - fd、buf、count: 同read中意义。
    - 大于0表示写了部分或全部数据，小于0表示出错。

- 关闭oscket -> int close(int fd);
  - fd: 同服务器端fd。

应用层(HTTP)" class="reference-link">应用层(HTTP)

HTTP协议工作在应用层，端口号是80。HTTP协议被用于网络中两台计算机间的通信，相比于TCP/IP这些底层协议，HTTP协议更像是高层标记型语言，浏览器根据从服务器得到的HTTP响应体中分别得到报文头，响应头和信息体（HTML正文等），之后将HTML文件解析并呈现在浏览器上。同样，我们在浏览器地址栏输入网址之后，浏览器相当于用户代理帮助我们组织好报文头，请求头和信息体（可选），之后通过网络发送到服务器，，服务器根据请求的内容准备数据。所以如果想要完全弄明白HTTP协议，你需要写一个浏览器 + 一个Web服务器，一侧来生成请求信息，一侧生成响应信息。

从网络分层模型来看，HTTP工作在应用层，其在传输层由TCP协议为其提供服务。所以可以猜到，HTTP请求前，客户机和服务器之间一定已经通过三次握手建立起连接，其中套接字中服务器一侧的端口号为HTTP周知端口80。在请求和传输数据时也是有讲究的，通常一个页面上不只有文本数据，有时会内嵌很多图片，这时候有两种选择可以考虑。一种是对每一个文件都建立一个TCP连接，传送完数据后立马断开，通过多次这样的操作获取引用的所有数据，但是这样一个页面的打开需要建立多次连接，效率会低很多。另一种是对于有多个资源的页面，传送完一个数据后不立即断开连接，在同一次连接下多次传输数据直至传完，但这种情况有可能会长时间占用服务器资源，降低吞吐率。上述两种模式分别是HTTP 1.0和HTTP 1.1版本的默认方式，具体是什么含义会在后面详细解释。

HTTP工作流程

一次完整的HTTP请求事务包含以下四个环节：

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

HTTP协议结构

请求报文

对于HTTP请求报文我们可以通过以下两种方式比较直观的看到：一是在浏览器调试模式下（F12）看请求响应信息，二是通过wireshark或者tcpdump抓包实现。通过前者看到的数据更加清晰直观，通过后者抓到的数据更真实。但无论是用哪种方式查看，得到的请求报文主题体信息都是相同的，对于请求报文，主要包含以下四个部分，每一行数据必须通过”\r\n”分割，这里可以理解为行末标识符。

- 报文头（只有一行）

  结构：method uri version

  - method

    HTTP的请求方法，一共有9中，但GET和POST占了99%以上的使用频次。GET表示向特定资源发起请求，当然也能提交部分数据，不过提交的数据以明文方式出现在URL中。POST通常用于向指定资源提交数据进行处理，提交的数据被包含在请求体中，相对而言比较安全些。

  - uri

    用来指代请求的文件，≠URL。

  - version

    HTTP协议的版本，该字段有HTTP/1.0和HTTP/1.1两种。

- 请求头（多行）

  在HTTP/1.1中，请求头除了Host都是可选的。包含的头五花八门，这里只介绍部分。

  - Host：指定请求资源的主机和端口号。端口号默认80。
  - Connection：值为keep-alive和close。keep-alive使客户端到服务器的连接持续有效，不需要每次重连，此功能为HTTP/1.1预设功能。
  - Accept：浏览器可接收的MIME类型。假设为text/html表示接收服务器回发的数据类型为text/html，如果服务器无法返回这种类型，返回406错误。
  - Cache-control：缓存控制，Public内容可以被任何缓存所缓存，Private内容只能被缓存到私有缓存，non-cache指所有内容都不会被缓存。
  - Cookie：将存储在本地的Cookie值发送给服务器，实现无状态的HTTP协议的会话跟踪。
  - Content-Length：请求消息正文长度。

    另有User-Agent、Accept-Encoding、Accept-Language、Accept-Charset、Content-Type等请求头这里不一一罗列。由此可见，请求报文是告知服务器请求的内容，而请求头是为了提供服务器一些关于客户机浏览器的基本信息，包括编码、是否缓存等。

- 空行（一行）
- 可选消息体（多行）

响应报文

响应报文是服务器对请求资源的响应，通过上面提到的方式同样可以看到，同样地，数据也是

以”\r\n”来分割。

- 报文头（一行）

  结构：version status_code status_message

  - version

    描述所遵循的HTTP版本。

  - status_code

    状态码，指明对请求处理的状态，常见的如下。

    - 200：成功。
    - 301：内容已经移动。
    - 400：请求不能被服务器理解。
    - 403：无权访问该文件。
    - 404：不能找到请求文件。
    - 500：服务器内部错误。
    - 501：服务器不支持请求的方法。
    - 505：服务器不支持请求的版本。

  - status_message

    显示和状态码等价英文描述。

- 响应头（多行）

  这里只罗列部分。

  - Date：表示信息发送的时间。
  - Server：Web服务器用来处理请求的软件信息。
  - Content-Encoding：Web服务器表明了自己用什么压缩方法压缩对象。
  - Content-Length：服务器告知浏览器自己响应的对象长度。
  - Content-Type：告知浏览器响应对象类型。

- 空行（一行）

- 信息体（多行）

  实际有效数据，通常是HTML格式的文件，该文件被浏览器获取到之后解析呈现在浏览器中。

CGI与环境变量

- CGI程序

  服务器为客户端提供动态服务首先需要解决的是得到用户提供的参数再根据参数信息返回。为了和客户端进行交互，服务器需要先创建子进程，之后子进程执行相应的程序去为客户服务。CGI正是帮助我们解决参数获取、输出结果的。

  动态内容获取其实请求报文的头部和请求静态数据时完全相同，但请求的资源从静态的HTML文件变成了后台程序。服务器收到请求后fork()一个子进程，子进程执行请求的程序，这样的程序称为CGI程序（Python、Perl、C++等均可）。通常在服务器中我们会预留一个单独的目录（cgi-bin）用来存放所有的CGI程序，请求报文头部中请求资源的前缀都是/cgi-bin，之后加上所请求调用的CGI程序即可。

  所以上述流程就是：客户端请求程序 -> 服务器fork()子进程 -> 执行被请求程序。接下来需要解决的问题就是如何获取客户端发送过来的参数和输出信息怎么传递回客户端。

- 环境变量

  对CGI程序来说，CGI环境变量在创建时被初始化，结束时被销毁。当CGI程序被HTTP服务器调用时，因为是被服务器fork()出来的子进程，所以其继承了其父进程的环境变量，这些环境变量包含了很多基本信息，请求头中和响应头中列出的内容（比如用户Cookie、客户机主机名、客户机IP地址、浏览器信息等），CGI程序所需要的参数也在其中。

  - GET方法下参数获取

    服务器把接收到的参数数据编码到环境变量QUERY_STRING中，在请求时只需要直接把参数写到URL最后即可，比如”http:127.0.0.1:80/cgi-bin/test?a=1&b=2&c=3”，表示请求cgi-bin目录下test程序，’?’之后部分为参数，多个参数用’&’分割开。服务器接收到请求后环境变量QUERY_STRING的值即为a=1&b=2&c=3。

    在CGI程序中获取环境变量值的方法是：getenv()，比如我们需要得到上述QUERY_STRING的值，只需要下面这行语句就可以了。

    ```
    1.   char *value = getenv("QUERY_STRING");
    ```

    之后对获得的字符串处理一下提取出每个参数信息即可。

  - POST方法下参数获取

    POST方法下，CGI可以直接从服务器标准输入获取数据，不过要先从CONTENT_LENGTH这个

环境变量中得到POST参数长度，再获取对应长度内容。

会话机制

HTTP作为无状态协议，必然需要在某种方式保持连接状态。这里简要介绍一下Cookie和Session。

- Cookie

    Cookie是客户端保持状态的方法。

    Cookie简单的理解就是存储由服务器发至客户端并由客户端保存的一段字符串。为了保持会话，服务器可以在响应客户端请求时将Cookie字符串放在Set-Cookie下，客户机收到Cookie之后保存这段字符串，之后再请求时候带上Cookie就可以被识别。

    除了上面提到的这些，Cookie在客户端的保存形式可以有两种，一种是会话Cookie一种是持久Cookie，会话Cookie就是将服务器返回的Cookie字符串保持在内存中，关闭浏览器之后自动销毁，持久Cookie则是存储在客户端磁盘上，其有效时间在服务器响应头中被指定，在有效期内，客户端再次请求服务器时都可以直接从本地取出。需要说明的是，存储在磁盘中的Cookie是可以被多个浏览器代理所共享的。

- Session

    Session是服务器保持状态的方法。

    首先需要明确的是，Session保存在服务器上，可以保存在数据库、文件或内存中，每个用户有独立的Session用户在客户端上记录用户的操作。我们可以理解为每个用户有一个独一无二的Session ID作为Session文件的Hash键，通过这个值可以锁定具体的Session结构的数据，这个Session结构中存储了用户操作行为。

当服务器需要识别客户端时就需要结合Cookie了。每次HTTP请求的时候，客户端都会发送相应的Cookie信息到服务端。实际上大多数的应用都是用Cookie来实现Session跟踪的，第一次创建Session的时候，服务端会在HTTP协议中告诉客户端，需要在Cookie里面记录一个Session ID，以后每次请求把这个会话ID发送到服务器，我就知道你是谁了。如果客户端的浏览器禁用了Cookie，会使用一种叫做URL重写的技术来进行会话跟踪，即每次HTTP交互，URL后面都会被附加上一个诸如sid=xxxxx这样的参数，服务端据此来识别用户，这样就可以帮用户完成诸如用户名等信息自动填入的操作了。