C++基础知识汇总

目录

1. new、delete、malloc、free 关系

delete 会调用对象的析构函数,和 new 对应。free 只会释放内存，new 调用构造函数。malloc 与 free 是 C++/C 语言的标准库函数，**new/delete 是 C++的运算符**。它们都可用于申请动态内存和释放内存。

对于非内部数据类型的对象而言，光用 maloc/free 无法满足动态对象的要求。对象在**创建**的同时要自动执行**构造函数**，对象在**消亡之前要自动执行析构**函数。由于 malloc/free 是库函数而不是运算符，不在编译器控制权限之内，不能够把执行构造函数和析构函数的任务强加于 malloc/free。因此 C++语言需要一个能完成动态内存分配和初始化工作的运算符 new，以及一个能完成清理与释放内存工作的运算符 delete。注意 new/delete 不是库函数。

2. delete 与 delete []区别

delete 只会调用一次析构函数，而 delete[]会调用每一个成员的析构函数。在 More Effective C++中有更为详细的解释：“当 delete 操作符用于数组时，它为每个数组元素调用析构函数，然后调用 operator delete 来释放内存。”delete 与 new 配套，delete []与 new []配套

```
MemTest *mTest1=new MemTest[10];
MemTest *mTest2=new MemTest;
Int *pInt1=new int [10];
Int *pInt2=new int;
delete[]pInt1; //-1-
delete[]pInt2; //-2-
delete[]mTest1;//-3-
delete[]mTest2;//-4-
```

在-4-处报错。

这就说明：**对于内建简单数据类型，delete 和 delete[]功能是相同的**。对于自定义的**复杂数据类型，delete 和 delete[]不能互用**。delete[]删除一个数组，delete 删除一个指针。简单来说，用 new 分配的内存用 delete 删除；用 new[]分配的内存用 delete[]删除。**delete[]会调用数组元素的析构函数。内部数据类型没有析构函数，所以问题不大**。如果你在用 delete 时没用括号，delete 就会认为指向的是单个对象，否则，它就会认为指向的是一个数组。

3. C++有哪些性质（面向对象特点）

封装，继承和多态。

4. 子类析构时要调用父类的析构函数吗？

析构函数调用的次序是**先派生类**的析构**后基类**的析构，也就是说在基类的的析构调用的时候,派生类的信息已经全部销毁了。**定义一个对象时先调用基类的构造函数、然后调用派生类的构造函数**；析构的时候恰好相反：先调用派生类的析构函数、然后调用基类的析构函数。

5. 多态，虚函数，纯虚函数

多态：是对于不同对象接收**相同消息**时产生**不同的动作**。

C++的多态性具体体现在运行和编译两个方面：

在**程序运行时**的多态性通过**继承和虚函数**来体现；

在**程序编译时**多态性体现在**函数和运算符的重载**上；

虚函数：在基类中冠以关键字 **virtual** 的**成员函数**。 它提供了一种接口界面。允许在**派生类**中对**基类**的**虚函数重新定义**。

纯虚函数的作用：在基类中为其派生类保留一个函数的名字，以便派生类根据需要对它进行定义。作为**接口**而存在 纯虚函数**不具备函数的功能**，一般不能直接被调用。

从基类继承来的纯虚函数，在派生类中仍是虚函数。如果一个类中至少**有一个纯虚函数**，那么这个类被称为**抽象类**（abstract class）。

抽象类中不仅包括纯虚函数，也可包括虚函数。抽象类**必须用作**派生其他类的**基类，而不能用于直接创建对象实例**。但仍可使用**指向抽象类的指针**支持**运行时多态性**。

6. 求下面函数的返回值（微软）

```
int  func(x)
{
int  countx  =  0;
while(x)
{
```

```
    countx  ++;
    x  =  x&(x-1);
  }
    return  countx;
  }
```

假定 x = 9999。 答案：8

思路：将 x 转化为 2 进制，看含有的 1 的个数。

7. 什么是“引用”？申明和使用“引用”要注意哪些问题？

答：**引用**就是某个目标**变量的“别名”**(alias)，对引用的操作与对变量直接操作效果完全相同。申明一个**引用**的时候，**必须初始化**。引用声明完毕后，相当于目标变量名有两个名称，即该目标原名称和引用名，不能再把**该引用名作为其他变量名的别名**。声明一个引用，不是新定义了一个变量，它只表示该引用名是目标变量名的一个别名，它本身不是一种数据类型，因此引用**本身不占存储单元**，系统也不给引用分配存储单元。不能建立数组的引用。

8. 将“引用”作为函数参数有哪些特点？

（1）**传递引用给函数与传递指针的效果是一样的**。这时，被调函数的形参就成为原来主调函数中的实参变量或对象的一个别名来使用，所以在被调函数中对形参变量的操作就是对其相应的目标对象（在主调函数中）的操作。

（2）使用引用传递函数的参数，在内存中并**没有产生实参的副本**，它是直接对实参操作；而使用**一般变量传递函数的参数**，当发生函数调用时，需要给形参**分配存储单元**，形参变量是实参变量的副本；**如果传递的是对象，还将调用拷贝构造函数**。因此，当参数传递的数据较大时，用引用比用一般变量传递参数的效率和所占空间都好。

（3）使用**指针作为函数的参数**虽然也能**达到**与使用**引用的效果**，但是，在被调函数中同样要**给形参分配存储单元**，且需要重复使用"*指针变量名"的形式进行运算，这很**容易产生错误**且程序的阅读性较差；另一方面，在主调函数的调用点处，**必须用变量的地址作为实参**。而**引用更容易使用，更清晰**。

9. 在什么时候需要使用“常引用”？

如果既要**利用引用提高程序的效率**，又要**保护**传递给函数的**数据**不在函数中被改变，就应使用常引用。常引用声明方式：const 类型标识符 &引用名=目标变量名；

例 1

```
int  a  ;
const  int  &ra=a;
ra=1;  //错误 不能再被赋值
a=1;  //正确
```

例 2

```
string  foo( );
void  bar(string  &  s); //非 const
```

那么下面的表达式将是非法的：

```
bar(foo( ));
bar("hello  world");
```

原因在于 foo( )和"hello world"串都会产生一个临时对象，而在 C++中，这些**临时对象都是 const 类型**的。因此上面的表达式就是试图将一个 const 类型的对象转换为非 const 类型，这是非法的。引用型参数应该在**能被定义为 const 的情况下，尽量定义为 const** 。

10. 将“引用”作为函数返回值类型的格式、好处和需要遵守的规则?

格式：类型标识符 &函数名（形参列表及类型说明）{ //函数体 }

好处：在内存中不产生被返回值的副本；（注意：正是因为这点原因，所以返回一个**局部变量的引用是不可取**的。因为随着该局部变量生存期的结束，相应的引用也会失效，产生 runtime error!

注意事项：

（1）**不能返回局部变量的引用**。这条可以参照 Effective C++[1]的 Item 31。主要原因是局部变量会在函数返回后被销毁，因此被返回的引用就成为了"无所指"的引用，程序会进入未知状态。

（2）**不能返回函数内部 new 分配的内存的引用**。这条可以参照 Effective C++[1]的 Item 31。虽然不存在局部变量的被动销毁问题，可对于这种情况（返回函数内部 new 分配内存的引用），又面临其它

尴尬局面。例如，被函数返回的引用只是作为一个临时变量出现，而没有被赋予一个实际的变量，那么这个引用所指向的空间（由 new 分配）就**无法释放**，造成 memory leak。

（3）**可以返回类成员的引用，但最好是 const**。这条原则可以参照 Effective C++[1]的 Item 30。主要原因是**当对象的属性是与某种业务规则（business rule）相关联**的时候，其赋值常常与某些其它属性或者对象的状态有关，因此有必要将赋值操作封装在一个业务规则当中。如果其它对象可以获得该属性的非常量引用（或指针），那么对该属性的**单纯赋值**就会**破坏业务规则的完整性**。

（4）**流操作符重载返回值申明为“引用”的作用**：

流操作符<<和>>，这两个操作符常常希望被连续使用，例如：cout << "hello" << endl; 因此这两个操作符的返回值应该是一个仍然支持这两个操作符的流引用。

可选的其它方案包括：返回一个**流对象**和返回一个**流对象指针**。但是对于返回一个流对象，**程序必须重新（拷贝）构造一个新的流对象**，也就是说，连续的两个<<操作符实际上是针对不同对象的！这无法让人接受。对于**返回一个流指针则不能连续使用<<操作符**。因此，返回一个流对象引用是惟一选择。这个唯一选择很关键，它说明了引用的重要性以及无可替代性，也许这就是 C++语言中引入引用这个概念的原因吧。

赋值操作符=。这个操作符象流操作符一样，是**可以连续使用**的，例如：x = j = 10;或者(x=10)=100;**赋值操作符的返回值必须是一个左值**，以便可以被继续赋值。因此引用成了这个操作符的惟一返回值选择。

```
#include<iostream.h>
int &put(int n);
int vals[10];
int error=-1;
void main()
{
    put(0)=10; //以 put(0)函数值作为左值，等价于 vals[0]=10;
    put(9)=20; //以 put(9)函数值作为左值，等价于 vals[9]=20;
    cout<<vals[0];
    cout<<vals[9];
}
int &put(int n)
{
    if (n>=0 && n<=9 ) return vals[n];
    else { cout<<"subscript error"; return error; }
}
```

（5）在另外的一些操作符中，却**千万不能返回引用**：+-*/ **四则运算符**。它们不能返回引用，Effective C++[1]的 Item23 详细的讨论了这个问题。主要原因是这四个操作符没有 side effect，因此，它们必须构造一个对象作为返回值，可选的方案包括：返回一个**对象**、返回一个**局部变量的引用**，返回一个 **new 分配的对象的引用**、返回一个**静态对象引用**。根据前面提到的引用作为返回值的三个规则，2、3 两个方案都被否决了。静态对象的引用又因为((a+b) == (c+d))会永远为 true 而导致错误。所以可选的只剩下**返回一个对象了**。

11 结构与联合有和区别？

(1). 结构和联合都是由多个不同的数据类型成员组成,但某时刻, **联合中只存放了一个被选中的成员**（所有成员共用一块地址空间）， **而结构的所有成员都存在**（不同成员的存放地址不同）。

(2). 对于**联合的不同成员赋值,将会对其它成员覆盖写**,原来成员的值就不存在了， 而对于**结构的不同成员赋值是互不影响**。

12 试写出程序结果：

```
int  a=4;
int  &f(int  x)
{    a=a+x;
     return  a;
}
int main(void)
```

```
{
    int   t=5;
    cout<<f(t)<<endl; // a = 9
    f(t)=20;             // a = 20 赋值
    cout<<f(t)<<endl;   //  t = 5,a = 20  a = 25
    t=f(t);              //  a = 30 t = 30
    cout<<f(t)<<endl; //  t = 60
}
```

13. 重载（overload)和重写(overried，有的书也叫做“覆盖”）的区别？

常考的题目。从定义上来说：

重载：是指允许存在**多个同名函数**，而这些函数的**参数表不同**（或许**参数个数**不同，或许**参数类型**不同，或许两者都不同）。

重写：是指**子类重新定义父类虚函数**的方法。

从实现原理上来说：

重载：**编译器**根据函数不同的参数表，对**同名函数的名称做修饰**，然后这些同名函数就成了不同的函数（至少对于编译器来说是这样的）。如，有两个同名函数：function func(p:integer):integer;和 function func(p:string):integer;。那么编译器做过修饰后的函数名称可能是这样的：int_func、str_func。对于这**两个函数的调用，在编译器间就已经确定了，是静态的**。也就是说，它们的地址在编译期就绑定了（早绑定），因此，重载和多态无关!

重写：和多态真正相关。当子类重新定义了父类的虚函数后，**父类指针根据赋给它的不同的子类指针，动态的调用属于子类的该函数**，这样的函数调用在编译期间是无法确定的（调用的子类的虚函数的地址无法给出）。因此，这样的函数地址是在运行期绑定的（晚绑定）。

14. 有哪几种情况只能用 intialization list 而不能用 assignment?

答案：当类中含有 const、reference **成员变量**；**基类的构造函数都需要初始化表**。

15. C++是不是类型安全的？

答案：不是。两个**不同类型的指针之间可以强制转换**（用 reinterpret cast)。C#是类型安全的。

16. main 函数执行以前，还会执行什么代码？

答案：**全局对象的构造函数**会在 main 函数之前执行。

17. 描述内存分配方式以及它们的区别?

1） 从**静态存储区域**分配。内存在程序**编译的时候**就已经分配好，这块内存在程序的整个运行期间都存在。例如**全局变量**，**static 变量**。

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

3） 从**堆上分配**，亦称动态内存分配。程序在运行的时候用 malloc 或 new 申请任意多少的内存，程序员自己负责在何时用 free 或 delete 释放内存。**动态内存的生存期由程序员决定，使用非常灵活，但问题也最多**。

18. 分别写出 BOOL,int,float,指针类型的变量 a 与“零”比较语句。

```
BOOL : if ( !a ) or if(a)
int : if ( a == 0)
float : const float EXPRESSION EXP = 0.000001
if ( a < EXP && a >-EXP)
pointer : if ( a != NULL) or if(a == NULL)
```

19. 请说出 const 与#define 相比，有何优点？

const 作用：定义**常量**、修饰**函数参数**、修饰**函数返回值**三个作用。被 const 修饰的东西都受到**强制保护**，可以预防意外的变动，能提高程序的健壮性。

1）const 常量有**数据类型**，而宏常量没有数据类型。**编译器可以对前者进行类型安全检查**。而对后者只进行字符替换，没有类型安全检查，并且在**字符替换可能会产生意料不到的错误**。

2）有些集成化的调试工具可以对 const **常量进行调试**，但是不能对宏常量进行调试。

20. 简述数组与指针的区别？

数组要么在**静态存储区**被创建（如全局数组），要么在**栈上被创建**。指针可以随时指向任意类型的内存块。

(1)修改内容上的差别

```
char a[] = “hello”;
a[0] = ‘X’;
char *p = “world”; // 注意 p 指向常量字符串
p[0] = ‘X’; // 编译器不能发现该错误，运行时错误
```

(2) **用运算符 sizeof 可以计算出数组的容量**（字节数）。sizeof(p),p 为指针得到的是一个指针变量的字节数，而不是 p 所指的内存容量。

C++/C 语言**没有办法知道指针所指的内存容量，除非在申请内存时记住它**。注意当**数组作为函数的参数**进行传递时，**该数组自动退化为同类型的指针**。

```
char a[] = "hello world";
char *p = a;
cout<< sizeof(a) << endl; // 12 字节
cout<< sizeof(p) << endl; // 4 字节
```

计算数组和指针的内存容量

```
void Func(char a[100])
{
    cout<< sizeof(a) << endl; // 4 字节而不是 100 字节
}
```

21 int (*s[10])(int) 表示的是什么？

int (*s[10])(int) **函数指针数组**，每个指针指向一个 int func(int param)的函数。

22 栈内存与文字常量区

```
char str1[] = "abc";
char str2[] = "abc";
const char str3[] = "abc";
const char str4[] = "abc";
const char *str5 = "abc";
const char *str6 = "abc";
char *str7 = "abc";
char *str8 = "abc";
cout << ( str1 == str2 ) << endl;//0  分别指向各自的栈内存
cout << ( str3 == str4 ) << endl;//0  分别指向各自的栈内存
cout << ( str5 == str6 ) << endl;//1 指向文字常量区地址相同
cout << ( str7 == str8 ) << endl;//1 指向文字常量区地址相同
```

结果是：0 0 1 1

解答：**str1,str2,str3,str4 是数组变量，它们有各自的内存空间；而 str5,str6,str7,str8 是指针，它们指向相同的常量区域**。

23 将程序跳转到指定内存地址

要对绝对地址 0x100000 赋值，我们可以用

```
(unsigned int*)0x100000 = 1234;（值的类型 int）
```

那么要是想让**程序跳转到绝对地址是 0x100000 去执行**，应该怎么做？

```
*((void (*)( ))0x100000 ) ( );
```

首先要将 0x100000 强制转换成**函数指针**,即：

```
(void (*)())0x100000
```

然后再调用它：

```
*((void (*)())0x100000)();
```

用 typedef 可以看得更直观些：

```
typedef void(*) () voidFuncPtr;
*((voidFuncPtr)0x100000)();
```

24 题：int id[sizeof(unsigned long)];这个对吗？为什么？

答案:正确 这个 sizeof 是编译时运算符，编译时就确定了 ，可以看成和机器有关的常量。

25 题：引用与指针有什么区别？

【参考答案】

1）**引用必须被初始化**，指针不必。

2）**引用初始化以后不能被改变**，指针可以改变所指的对象。

3）**不存在指向空值的引用，但是存在指向空值的指针**。

26 题：const 与 #define 的比较 ，const 有什么优点？

同 17

27 题：复杂声明

```
void * ( * (*fp1)(int))[10];
float (*(* fp2)(int,int,int))(int);
int (* ( * fp3)())[10]();
```

分别表示什么意思？

【标准答案】

1.void * (* (*fp1)(int))[10]; fp1 是一个指针，指向一个函数，这个函数的参数为 int 型，函数的返回值是一个指针，这个指针指向一个数组，这个数组有 10 个元素，每个元素是一个 void*型指针。

2.float (*(* fp2)(int,int,int))(int); fp2 是一个指针，指向一个函数，这个函数的参数为 3 个 int 型，函数的返回值是一个指针，这个指针指向一个函数，这个函数的参数为 int 型，函数的返回值是 float 型。

3.int (* (* fp3)())[10](); fp3 是一个指针，指向一个函数，这个函数的参数为空，函数的返回值是一个指针，这个指针指向一个数组，这个数组有 10 个元素，每个元素是一个指针，指向一个函数，这个函数的参数为空，函数的返回值是 int 型。

29 基类的析构函数不是虚函数，会带来什么问题？

【参考答案】派生类的**析构函数用不上**，会造成资源的泄漏。

30 题：全局变量和局部变量有什么区别？是怎么实现的？操作系统和编译器是怎么知道的？

【参考答案】

生命周期不同：

全局变量随主程序创建时创建，随主程序销毁而销毁；**局部变量**在局部函数内部，甚至局部循环体等内部存在，退出就不存在；

使用方式不同：**通过声明后**全局变量程序的各个部分都可以用到；**局部变量只能在局部使用**；分配在栈区。

操作系统和编译器通过内存分配的位置来知道的，全局变量分配在全局数据段并且在程序开始运行的时候被加载。局部变量则分配在堆栈里面 。

C++ 知识要点

一.变量

1）全局变量与 static 变量？（作用域、生存周期）

2）static 函数与普通函数的区别？

3）两个文件中声明两个同名变量？（使用了与未使用 extern？）

4）全局数组和局部数组的初始化？

5）指针和引用的区别？（代表意义、内存占用、初始化、指向是否可改、能否为空）

6）C/C++中的强制转换

7）如何修改 const 变量、const 与 volatile

8）静态类型获取与动态类型获取（typeid、dynamic_cast:转换目标类型必须是引用类型）

9）如何比较浮点数大小？（直接使用==比较出现错误的例子）

二.函数

1）重载（参数必须不同(const 修饰形参)、重载与作用域、继承中的重载(using)、重载与 const 成员函数）

三.类

1）面向对象的三大特性（封装、继承、多态）
2）struct 和 class 的区别？
3）访问权限说明符？（目的是加强类的封装性）
4）类的静态成员（所属？静态成员函数不能声明成 const、类类型的成员、定义时不能重复使用 static、具有类内初始值的静态成员定义时不可再设初值）
5）构造函数相关

- 有哪些构造函数（默认、委托、拷贝、移动）
- 合成的默认拷贝构造函数（默认行为？什么情况下不会合成？怎么解决？如果成员包含类内初始值，合成默认构造函数会使用该成员的类内初始值初始化该成员）
- 拷贝构造函数（调用时机、合成版的行为、explict？、为何第一个参数必须是引用类型）
- 移动拷贝构造函数（非拷贝而是窃取资源、与 noexcept?、何时合成）
- 可否通过对象或对象的引用(指针或引用)调用

6）初始值列表（顺序、效率(内置类型不进行隐式初始化故无所谓,但..)、无默认构造函数的成员,const 成员,引用成员必须通过初始值列表初始化）
7）赋值运算符相关

- 拷贝赋值运算符（合成版的行为？、与 delete？、自定义时要注意自赋值，参数与返回类型、大部分组合了拷贝构造函数与析构函数的工作）
- 阻止拷贝（某些对象应该独一无二(比方说人)、C++11 前:private 并且不定义(试图拷贝会报链接错误)，C++11:=delete《Effective C++:条款 6》）
- 移动赋值运算符（与 noexcept？何时合成）
- 可以定义为成员或非成员函数，定义成成员函数时第一个操作数隐式绑定到 this 指针
- 不可重载的操作符有哪些？（?:，::）

8）析构函数相关

- 销毁过程的理解（delete 会执行哪些操作？逆序析构成员）
- 为什么析构函数中不能抛出异常？（不能是指“不应该”，C++本身并不禁止《Effective C++:条款 8》）
- 如果析构函数中包含可能抛出异常的代码怎么办？（Effective C++:条款 8》）
- 可否通过对象或对象的引用(指针或引用)调用
- 为什么将继承体系中基类的析构函数声明为虚函数？（《Effective C++:条款 7》）
- 不应该将非继承体系中的类的虚函数声明为虚函数（《Effective C++:条款 7》）
- 不应该继承析构函数非虚的类（《Effective C++:条款 7》，final 防止继承）
- 防止继承的方式

9）删除的合成函数（一般函数而言不想调用的话不定义就好）
10）继承相关

- 继承体系中的构造、拷贝、析构顺序？（派生类只负责自己成员的拷贝控制，可以(换而言之非必须，如果不显示调用，会调用父类合成的默认版本)在初始值列表或函数体中调用基类相应函数）
- 继承中的名字查找（作用域嵌套、从子类到父类查找；成员名字的处理）
- 成员函数体内、成员函数的参数列表的名字解析时机（因此，务必将“内嵌的类型声明”放在 class 起始处）
- 同名名字隐藏（如何解决？(域作用符，从指示的类开始查找)、不同作用域无法重载、using 的作用？除此之外呢？）
- 虚继承（解决什么问题？(多继承中的子对象冗余)）

11）多态的实现？
12）虚函数的实现原理？对类大小的影响？（vtbl 是一个由函数指针组成的数组，无论 pb 指向哪种类型的对象，只要能够确定被调函数在虚函数中的偏移值，待运行时，能够确定具体类型，并能找到相应 vptr，进一步能找出真正应该调用的函数）

13）为什么不要在构造、析构函数中调用虚函数？（子对象的 base class 构造期间，对象的类型是 base class 《Effective C++:条款 9》，设置虚函数指针的时机）
14）虚函数被覆盖？
15）virtual 函数动态绑定，缺省参数值静态绑定（《Effective C++:条款 37》）
16）纯虚函数与抽象基类（纯虚函数与虚函数、一般成员函数的选择）
17）静态类型与动态类型（引用是否可实现动态绑定）
18）浅拷贝与深拷贝（安全性、行为像值的类与行为像指针的类）
19）如何定义类内常量？（enum 而不是 static const 《Effective C++:条款 2》）
20）继承与组合(复合)之间如何选择？（《Effective C++:条款 38》）
21）private 继承？（《Effective C++:条款 39》）
22）如何定义一个只能在堆上（栈上）生成对象的类？
23）内联函数、构造函数、静态成员函数可以是虚函数吗？

四.内存管理

1）C++内存分区
2）new 和 malloc 的区别？（函数，运算符、类型安全、计算空间、步骤，operator new 的实现）
3）new[]与 delete[]？（步骤：如何分配内存，构建对象、如何析构与释放内存？构造与析构）
4）new 带括号和不带的区别？（无自定义构造函数时，不带括号的 new 只分配内存，带括号的 new 会初始化为 0）
5）new 时内存不足？（《Effective C++:条款 49》）(new-handler)
6）malloc、calloc、realloc、alloca，malloc 的实现？
7）调用 malloc 函数之后，OS 会马上分配内存空间吗？（不会，只会返回一个虚拟地址，待用户要使用内存时，OS 会发出一个缺页中断，此时，内存管理模块才会为程序分配真正内存）
8）delete（步骤、delete 与析构、可以 delete 空指针、可以 delete 动态 const 对象）
9）为什么要内存对齐？(性能原因、平台原因)
10）struct 内存对齐方式？
11）如何取消内存对其？（添加预处理指令`#pragma pack(1)`）
12）什么是内存泄露？如何检测与避免？（Mtrace，valgrind）
13）智能指针相关
- 种类、区别、原理、能否管理动态数组
- shared_ptr（使用、计数的变化，get()函数要注意什么）
- unique_ptr(如何转移控制权)
- weak_ptr(特点、用途：可以解决 shared_ptr 的循环引用问题)
- 手写实现智能指针

14）实现 memcpy
15）memcpy 与 memmove 的区别（前者不处理重叠，后者处理重叠）
16）能否使用 memcpy 比较两个结构体对象？
17）实现 strlen、strcmp、strcat、strcpy

五.STL

1）顺序容器与关联容器的比较？有哪些顺序容器与关联容器？
2）vector 底层的实现（迭代器类型为随机迭代器）？insert 具体做了哪些事？resize()调用的是什么？
3）vector 的 push_back 要注意什么（大量调用会伴随大量的拷贝构造与析构，内存分配与释放）
4）vector 的 resize()与 reserve()（测试程序）
5）如何释放 vector 的空间？（swap）、容器的元素类型为指针？（会有内存泄露，指针是 trivial_destructor；也可以使用智能指针来管理）
6）vector 的 clear 与 deque 的 clear（vector 的 erase 和 clear 只会析构不会释放内存，deque 的 erase 和 clear 不但会析构，还可能会释放缓冲区）
7）list 的底层实现（迭代器类型为双向迭代器）
8）deque 的底层实现（迭代器类型为随机迭代器）
9）vector 与 deque 的区别？（deque 能以常数时间在首尾插入元素；deque 没有 capacity 的概念）

10）map、set 的实现原理（红黑树、对于 set 来说 key 和 value 合一，value 就是 key，map 的元素是一个 pair，包括 key 和 value、set 不支持[]，map(不包括 multimap)支持[]）
11）set(map)和 multiset(multimap)的区别？（set 不允许 key 重复,其 insert 操作调用 rb_tree 的 insert_unique 函数，multiset 允许 key 重复,其 insert 操作调用 rb_tree 的 insert_equal 函数）
12）set(multiset)和 map(multimap)的迭代器（由于 set(multiset)key 和 value 合一，迭代器不允许修改 key、map(multimap)除了 key 有 data，迭代器允许修改 data 不允许修改 key）
13）map 与 unordered_map 的区别？（hash_map 需要 hash 函数及等于函数，map 只需小于函数）
14）set(multiset)和 map(multimap)的迭代器++操作、--操作的时间复杂度？
15）空间分配器 allocator

- 将 new 和 delete 的 2 阶段操作分离（construct 和 destroy 负责内存分配？allocate 和 deallocate 负责对象构造析构？）
- SGI 符合部分标准的空间分配器——std::allocator
- SGI 特殊的空间分配器——std::alloc（对象构造与析构、内存分配与释放——两级分配器）
  - 第一级分配器（如何仿真 new-handler 机制？不能直接用 C++ new-handler，因为没有使用::operator new）
  - 第二级分配器（为什么要二级分配器？内存池与 16 个 free-list？空间分配和释放的步骤？）

16）traits 与迭代器相应类型

六.对象内存模型

1. 数据成员

1. 成员变量在类对象中的布局规则
2. 通过指针和通过'.'进行 Data Member 存取的区别
3. 数据成员的布局——无继承
4. 数据成员的布局——不含多态的继承（C++标准并未强制指定派生类和基类成员的排列顺序；理论上编译器可以自由安排。在大部分编译器上，基类成员总是先出现，虚基类除外）
5. 数据成员的布局——含多态的继承（vptr 的位置也没有强制规定，放在不同位置分别有什么好处？）
6. 数据成员的布局——多重继承（基类子对象的排列顺序也没有硬性规定；指针的调整方式？）
7. 数据成员的布局——虚继承（虚基类子对象的偏移信息记录在虚函数表之前与使用一个额外指针来记录的对比？）
8. 指向数据成员的指针

2. 函数成员

1. nonstatic 成员函数的转换（目的是为了提供和一般非成员函数相同的效率）
2. 重载成员函数的名字处理
3. static 成员函数的转换
4. 编译器如何处理经由指针和经由'.'进行的调用
5. 指向函数成员的指针
6. 虚函数的调用——单继承
7. 虚函数的调用——多重继承（子类对象关联有多少个虚函数表？不同虚函数表的名称？执行期什么情况下如何调整 this 指针？）

七.关键字

1）extern？（extern "C"?、与 static？、有什么问题？、extern 的时候定义变量？）
2）const？（修饰变量、修饰指针与引用、修饰成员函数 《Effective C++:条款 3》）
3）mutable？
4）static？（修饰变量、类中使用）
5）define 与 const、enum、inline？（《Effective C++:条款 2》、C 中默认 const 是外部连接的，而 C++中默认 const 是内部连接的）

6）explict?（抑制隐式转换、可通过显示转换或直接初始化解决、类外定义时不应重复出现）
7）noexcept？（承诺不会抛出异常）
8）default、delete?（显示要求编译器合成、不能被调用）
9）using？（用于命名空间？、用于类中？）
10）final？（修饰类？、修饰成员函数？只有虚函数能使用 final）
11）auto(初始值为引用时类型为所引对象的类型、必须初始化、不能用于函数及模板)、decltype？
12）volatile?（对象的值可能在程序的控制外被改变时，应将变量申明为 volatile，告诉编译器不应对这样的对象进行优化，如果优化，从内存读取后 CPU 会优先访问数据在寄存器中的结果，但是内存中的数据可能在程序之外被改变、可以既是 const 又是 volatile，const 只是告诉程序不能试图去修改它.volatile 是告诉编译器不要优化，因为变量可能在程序外部被改变）

八.其它

1）调试程序的方法?（gdb）
2）遇到 coredump 要怎么调试？
3）模板的用法与适用场景
4）用过 C++11？新特性？（auto,decltype、explicit、lambda、final）
5）函数调用的压栈过程
6）sizeof 和 strlen 的区别？（运算符与函数、计算的对象、编译时运行时）
7）union？
8）覆盖、重载与隐藏（覆盖要求参数完全相同，用于继承体系的虚函数中，重载要求参数不同）
9）C++是不是类型安全的？（不是，两个不同类型指针可以强制转换）
10）gcc 和 g++的区别？（gcc 代表 GUN Compiler Collection，是一堆编译器的集合，包括 g++）
11）运行时类型识别实现对象比较函数
12）使用 C++实现线程安全的单例模式
13）什么是异常安全？

Linux 网络相关理论

一.理论

1.应用层

1）http 协议与 TCP 联系？
2）http/1.0 和 http/1.1 的区别（非持久连接与持久连接、范围请求(断点续传)、缓存处理、更多状态码）
3）http1.1 和 http2.0 的区别？（二进制分帧层、服务端推送、首部压缩）
4）http 的请求方法有哪些？GET 和 POST 的区别（获取资源与传输数据、额外参数的位置、支持的编码）
5）http 的状态码
6）http 和 https 的区别，由 http 升级为 https 需要做哪些操作？
7）https 的具体实现，怎么确保安全性？
8）cookie 和 session 的区别？
9）服务器攻击（DDos 攻击）
10）对称加密和非对称加密
11）数字证书的了解
12）RSA 加密算法，MD5 原理（MD5 不算加密算法）
13）在浏览器中输入 URL 后执行的全部过程（DHCP 获取主机和网关路由 IP，ARP 解析网关路由 MAC 地址,DNS 解析域名，TCP 连接，HTTP 请求响应）
14）URL 包括哪些部分？（协议、服务器名称、文件路径、还可能带有参数）
15）http 请求/响应报文构成
16）DNS？（查询过程？DNS 记录？）

2.运输层

1）一个机器能够使用的端口号上限是多少，为什么？可以改变吗？那如果想要用的端口超过这个限制怎么办？
2）TCP 和 UDP 的区别（什么时候用 TCP，什么时候用 UDP、首部？）
3）TCP 和 UDP 相关的协议与端口号
4）TCP 为什么需要三次握手和四次挥手？
5）TCP 三次握手和四次挥手的状态转换？（SYN 洪泛攻击？）
6）如果第二次握手丢包怎么办？第三次呢？
7）TIME_WAIT 状态时，可以接收到新的请求吗？意义、为什么要等待 2MSL
8）TCP 怎么保证可靠性（通过超时重传，应答保证数据不会丢失、通过校验和保证数据可靠，通过序号，ACK，滑动窗口保证数据顺序性和有效性，通过拥塞控制缓解网络压力，通过流量控制同步收发速率）？
9）流量控制的介绍，采用滑动窗口会有什么问题（死锁可能，糊涂窗口综合征）？
10）TCP 滑动窗口协议
11）拥塞控制和流量控制的区别（减轻网络传输数据的压力、减轻接收方接收数据的压力）
12）TCP 拥塞控制，算法名字？（慢启动、拥塞避免、快重传、快恢复）
13）TCP、UDP、IP 等首部的认识
14）超时重传机制（不太高频）
15）TCP 数据的正确性
16）TCP 数据流的理解以及粘包
17）客户端收到一个接收窗口为 0 的包？

3.网络层

1）路由协议所使用的算法
2）对路由协议的了解，内部网关协议 IGP 包括 RIP，OSPF，和外部网关协议 EGP 和 BGP
3）网络层分片的原因与具体实现（标识、标志、比特片偏移，分片、重组）
4）ICMP？
5）介绍一下 ping 的过程，分别用到了哪些协议（ping 域名、ping ip、DNS,ARP,ICMP 回显，ICMP 回显应答）
6）TraceRoute 实现原理
7）DHCP？
8）NAT 过程？
9）一个 ip 配置多个域名，靠什么识别？
10）为什么选择在网络层分片

4.链路层

1）多路访问协议
2）MAC 地址
3）ARP 地址解析过程
4）为什么链路层有 MTU？

5.其它

1）单条记录高并发访问的优化
2）网卡工作在哪一层（既工作在物理层，也工作在链路层的 MAC 子层）

二.Linux

1）bcopy 和 memcpy 的区别？
2）字节序？（小端：低序字节存储在低地址；大端：低序字节存储在高地址）
3）TCP 通信的各个系统调用
连接建立：socket、connect、bind、listen、accept、read、write（发送缓冲区）
连接关闭：close、shutdown（close 与 shutdown 的区别？）
4）Linux 服务器最大 TCP 连接数？（一个端口能接受 tcp 连接数量的理论上限？）
5）非阻塞 connect？
6）连接建立过程中每个 SYN 可以包含哪些 TCP 选项？（MSS 选项(TCP_MAXSEG)、窗口规模选项）作用是什么？

7）TCP 连接建立过程中的超时
8）UDP 通信的各个系统调用
socket、connect、bind、sendto（发送缓冲区）、recvfrom、close
连接 UDP 套接字与非连接 UDP 套接字的区别？性能？
9）数据发送：write、send、sendto、sendmsg
10）接收数据：read、recv、recvfrom、recvmsg
11）I/O 复用：select、poll、epoll？
select：优缺点、中描述符集中最大描述符的个数（由 FD_SETSIZE 决定、怎么更改）？描述符就绪的条件？
poll：优缺点？事件？
epoll：优缺点、工作模式
12）Linux 高性能服务器编程——进程池和线程池
13）100 万并发连接服务器性能调优
14）TIME_WAIT 和 CLOSE_WAIT 状态详解及性能调优
15）accept 与 epoll 惊群
16）使用同步 IO 模型实现的 Reactor 模式的工作流程

I/O 模型

1. 使用同步 IO 模型实现的 Reactor 模式的工作流程

以 epoll_wait 为例

1. 主线程往 epoll 内核事件表中注册 socket 上的读就绪事件
2. 主线程调用 epoll_wait 等待 socket 上有数据可读
3. 当 socket 上有数据可读时，epoll_wait 通知主线程。主线程将 socket 可读事件放入请求队列
4. 睡眠在请求队列上的某个工作线程被唤醒，它从 socket 读取数据，并处理客户请求，然后往 epoll 内核事件表中注册该 socket 上的写就绪事件
5. 主线程调用 epoll_wait 通知主线程。主线程将 socket 可写事件放入请求队列
6. 睡眠在请求队列上的某个工作线程被唤醒，它往 socket 上写入服务器处理客户请求的结果

2. 如何理解阻塞非阻塞与同步异步的区别

1. 同步和异步关注的是消息通讯机制
    1. 所谓同步，就是在发出一个调用时，在没有得到结果之前，该调用就不返回。但是一旦调用返回，就得到返回值了。换句话说，就是由调用者主动等待这个调用的结果
    2. 而异步则是相反，调用在发出之后，这个调用就直接返回了，所以没有返回结果。换句话说，当一个异步过程调用发出后，调用者不会立刻得到结果。而是在调用发出后，被调用者通过状态、通知来通知调用者，或通过回调函数处理这个调用
2. 阻塞和非阻塞关注的是程序在等待调用结果（消息，返回值）时的状态
    1. 阻塞调用是指调用结果返回之前，当前线程会被挂起。调用线程只有在得到结果之后才会返回
    2. 非阻塞调用指在不能立刻得到结果之前，该调用不会阻塞当前

C++ 基础知识小结

1. 为什么要内存对齐

1. **性能原因**：内存对齐可以**提高存取效率**（例如，有些平台每次读都是从偶地址开始，如果一个 int 型存放在偶地址开始的地方，那么一个读周期就可以读出这 32bit，而如果存放在奇地址开始的地方，就需要 2 个读周期，并且要对两次读出的结果的**高低字节进行拼凑**才能得到这 32bit 的数据）
2. **平台原因**：各个硬件平台对存储空间的处理有很大的不同，一些平台对某些特定类型的数据只能从某些特定地址开始存取，例如，有些架构的 CPU 在访问一个**没有对齐的变量时会发生错误**，那么这时候编程必须保证字节对齐

2. valgrind

2.1 框架

vagrind 包括下列一些工具：

- **Memcheck**：valgrind 应用最广泛的工具，一个重量级的**内存检查**器，能够发现开发中绝大多数内存错误使用情况，**我们主要使用即此工具，默认选项**。此工具检查下面的程序错误：
  - 使用未初始化的内存(Use of uninitialised memory)
  - 使用已经释放了的内存(Reading/writing memory after it has been free'd)
  - 使用超过 malloc 分配的内存空间(Reading/writing off the endof malloc'd blocks)
  - **对堆栈的非法访问**(Reading/writing inappropriate areas on the stack)
  - 内存泄露(Memory leaks – where pointers to malloc'd blocks are lost forever)
  - malloc/free/new/delete 申请和释放内存的匹配(Mismatched use of malloc/new/new [] vs free/delete/delete [])
  - src 和 dst 的重叠(Overlapping src and dst pointers inmemcpy() and related functions)
- **Callgrind**：主要用来检查程序中**函数调用过程**中出现的问题
- **Cachegrind**：它主要用来检查程序中**缓存使用**出现的问题
- **Helgrind**：它主要用来检查**多线程**程序中出现**的竞争**问题
- **Massif**：主要用来检查程序中**堆栈使用**中出现的问题
- **Extension**：可以利用 core 提供的功能，自己编写特定的内存调试工具

2.2 内存检测原理

Memcheck 能够检测出内存问题，关键在于其建立了两个全局表：

- **Valid-Value 表**：对于进程的整个地址空间中的每一个字节(byte)，都有与之对应的 **8 个 bits**；对于 CPU 的每个寄存器，也有一个与之对应的 bit 向量。这些 bits 负责记录该字节或者寄存器**值是否具有有效的、已初始化的值。**
- **Valid-Address 表**：对于进程整个地址空间中的每一个字节(byte)，还有与之对应的 **1 个 bit**，负责记录**该地址是否能够被读写。**

检测原理：

- 当要读写内存中某个字节时，首先检查这个字节对应的 A bit。如果该 A bit 显示该位置是无效位置，memcheck 则报告读写错误
- 内核（core）类似于一个**虚拟的 CPU 环境**，这样当内存中的某个字节被加载到真实的 CPU 中时，该字节对应的 V bit 也被加载到虚拟的 CPU 环境中。一旦寄存器中的值，被用来产生内存地址，或者该值能够影响程序输出，则 memcheck 会检查对应的 V bits，**如果该值尚未初始化，则会报告使用未初始化内存错误**

2.3 检测步骤与示例

步骤：

- **编译源文件获取可执行程序**：为了使 valgrind 发现的错误更精确，如能够定位到源代码行，建议在编译时加上**-g 参数** `gcc|g++ -g 源文件`
- **在 valgrind 下，运行可执行程序**：
  - Valgrind 的参数分为两类：
    - 一类是 core 的参数，它对所有的工具都适用
    - 另外一类就是具体某个工具如 memcheck 的参数。Valgrind 默认的工具就是 memcheck，也可以通过"–tool=tool name"指定其他的工具

```
valgrind [valgrind-options] program [program-options]
```

1）使用未初始化内存

```
#include <stdio.h>

int main()
{
    int s,i;
    printf("sum:%d\n",s);

    return 0;
}
```

程序：

错误信息：

2）内存越界访问

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int *arr = (int*)malloc(sizeof(int) * 4);
    arr[4] = 10;

    return 0;
}
```

程序：

错误信息：

3）内存覆盖

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h> //memcpy

int main()
{
    char buf[20];

    int i;
    for(i = 1;i <= 20;i++)
        buf[i - 1] = i;

    //(dst,src,size)
    memcpy(buf + 5,buf,10);
    memcpy(buf,buf + 5,10);

    return 0;
}
```

程序：

错误信息：

4）动态内存管理错误

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    char *buf = (char*)malloc(20);

    int i;
    for(i = 1;i <= 20;i++)
        buf[i - 1] = i;

    delete buf;     //1.申请与释放不匹配
```

```
    buf[1] = 'a';    //2.读写释放后的内存

    return 0;
}
```

程序：

错误信息：

5）内存泄露

```
#include <stdio.h>
#include <stdlib.h>

struct ListNode{
public:
    ListNode(int v,ListNode *n) : next(n),val(v) {}
private:
    ListNode *next;
    int val;
};

int main()
{
    //n2->n1->NULL
    ListNode *n1 = new ListNode(1,NULL);
    ListNode *n2 = new ListNode(2,n1);
    return 0;
}
```

程序：

错误信息：

- **确定的内存泄露**
  - **直接的内存泄露（definitely lost）**：直接是没有任何指针指向该内存
  - **间接的内存泄露（indirectly lost）**：间接是指向该内存的指针都位于内存泄露处，即由直接内存泄露引起的内存泄露
- **可能的内存泄露（possibly lost）**：指仍然存在某个指针能够访问某块内存，但该指针指向的已经不是该内存首地址

3. 访问控制说明符

1. **类的成员的访问控制说明符**用于控制**类的使用者**对类中成员的访问权限
2. **派生列表中的访问控制说明符**用于控制**派生类的使用者**对**派生类从基类继承的成员**的访问权限

对应上图，基类中成员的访问控制说明符就是控制派生类与基类的用户对基类的访问权限。而派生类定义时使用的**派生列表中的访问控制说明符与派生类对基类成员的访问没有任何关系**，它控制的是派生类的派生类和派生类用户(即派生类的使用者)对派生类从基类继承的成员的访问权限

1）类的成员访问控制说明符

成员	派生类的成员和(派生类的)友元	类的用户
public 成员	可访问	可访问
protected 成员	**只能访问派生类对象中的基类部分的 protected 成员**	不可访问
private 成员	不可访问	不可访问

如果派生类的成员和友元能直接访问基类对象的 protected 成员，那么类的用户就可以定义一个继承基类的类，然后通过这个类来获得基类 protected 成员的访问。从而简单地规避掉 protected 提供的访问保护，违背基类只希望和派生类分享 protected 成员，而不想被其他公共访问的初衷了。

```
struct Base {
public:
    string pub_string = "public string";
protected:
    string pro_string = "protected string";
private:
    string pri_string = "private string";//只有类自己能访问
};

struct Derived : public Base {
public:
    void access_parent_public(const Base &b){
        cout << b.pub_string << endl;
    }
    //不能直接访问基类对象的protected 成员
    void access_parent_protected(const Base &b){
        //cout << b.pro_string << endl;
    }
    //派生类只能访问派生类对象中基类部分的protected 成员
    void access_protected_in_derived(){
        cout << pro_string << endl;
    }
};
```

2）派生列表中的访问控制说明符

派生列表中的访问控制说明符控制**派生类的使用者(派生类的派生类和派生类的用户)**对**派生类继承自基类成员**的访问权限

- public 继承：**遵循其原有的访问权限**
- protected 继承：基类中所以 public 成员在派生类中”相当于“protected 的
- private 继承：所有成员都无法访问

对于派生类的派生类(包括其成员和友元)：

继承方式	public 成员	protected 成员	private 成员
public 继承	可访问	**只能访问继承到的受保护成员**	不可访问
protected 继承	**只能访问继承到的 public 成员**	不可访问?	不可访问
private 继承	不可访问	不可访问	不可访问

对于派生类的用户：

继承方式	public 成员	protected 成员	private 成员
public 继承	可访问	不可访问	不可访问
protected 继承	不可访问	不可访问	不可访问
private 继承	不可访问	不可访问	不可访问

3）(另)派生类向基类转换的可行性

受两个因素影响：

1. 使用该转换的代码
2. 派生类的派生访问说明符

假定 D 继承自 B（B<–D）：

- **对于 D 的成员及友元**：不论以什么方式继承，都能使用派生类向基类的转换
- **对于 D 的派生类的成员及友元**：只有当 D 以 public 或 protected 方式继承时能使用

- **对于用户代码**：只有当 D 以 public 继承 B 时，才能使用派生类向基类的转换

总的来说，对于代码中的某个给定节点来说，如果基类的公用成员是可访问的，则派生类向基类的类型转换也是可性的

4. static 函数与普通函数的区别

- static 函数与普通函数**作用域不同**,仅在本文件。**只在当前源文件**中使用的函数应该说明为内部函数(static 修饰的函数)，内部函数应该在当前源文件中说明和定义。**对于可在当前源文件以外使用的函数，应该在一个头文件中说明，要使用这些函数的源文件要包含这个头文件**
- **static 函数在内存中只有一份**，普通函数在每个被调用中维持一份拷贝（这里暂时不理解）

C++ Primer

1 变量

1. 类型

算术类型

整形

包括 char 和 bool 在内

浮点型

单精度

双精度

扩展精度

空类型（void）

使用建议：

- **使用 int 执行整数运算**，超过范围用 long long，因为 long 一般和 int 大小一样
- **浮点运算用 double**，float 通常精度不够而且双精度和单精度的计算代价相差无几。long double 提供的精度通常没必要，而且运算时的消耗也不容忽视。

2. 大小

- **字节**：**内存可寻址的最小块**，大多数计算机将内存中的每个字节与一个数字(地址)关联起来。C++中，一个字节要至少能容纳机器基本字符集中的字符；
- **字**：一般是 32 比特(4 字节)或 64 比特(8 字节)在不同机器上有所差别，对于 C++标准(P30)：
  - 一个 char 的大小和机器字节一样；
  - bool 大小未定义；
  - int 至少和 short 一样大；
  - long 至少和 int 一样大；
  - long long(C++11)至少和 long 一样大

3. signed 与 unsigned

除了 **bool** 和**扩展字符型**外，都可以分为 signed 和 unsigned；char 可以表现为 signed char 和 unsigned char，具体由编译器决定；

- **unsigned 减去一个数必须保证结果不能是一个负值，否则结果是取模后的值**（比如，很多字符串的长度为无符号型，**在 for 循环非常容易出现** `str.length() - i >= 0` 这种表达，如果 i 比字符串长度大，那么就会引发错误）
- **signed 会转化为 unsigned（切勿混用 signed 和 unsigned)**
- **溢出**
  1. **赋给 unsigned 超过范围的值**：结果是初始值对无符号类型表示值总数取模后的余数
  2. **赋给 signed 超过范围的值**：结果未定义，可能继续工作、崩溃、生成垃圾数据

4. 类型转换

4.1 隐式转换与显式转换

1 **隐式转换**

1.1 **整形的隐式转换**：多数表达式中，比 int 小的**整形首先提升**为较大整形

1.2 **数组转成指针**

1.3 **指针的转换**：0,nullptr 转成任意指针，任意指针转 void

1.4 **转换时机**：

- 拷贝初始化
- 算术或关系运算
- 函数调用时

2 **显示转换**

2.1 **命名强制类型转换** `cast-name<type>(expression)`

- **static_cast**：只要**不包含底层 const**，都可以使用。适合将**较大**算术类型**转换成较小**算术类型

- **const_cast**：只能改变底层 const，例如指向 const 的指针(指向的对象不一定是常量，但是无法通过指针修改)，如果指向的对象是常量，则这种转换在修改对象时，结果未定义。
- **reinterpret_cast**：通常为算术对象的位模式提供较低层次上的重新解释。如将 int*转换成 char*。很危险！
- **dynamic_cast**：一种动态类型识别。转换的目标类型，即 type，是指针或者左右值引用，主要用于**基类指针转换成派生类类型的指针**(或引用)，通常需要知道转换源和转换目标的类型。如果转换失败，返回 0（转换目标类型为指针类型时）或抛出 bad_cast 异常（转换目标类型为引用类型时）

2.2 旧式强制类型转换 type (expr) 或 (type) expr

旧式强制类型转换与 const_cast，static_cast，reinterpret_cast 拥有相似行为，如果换成 const_cast，static_cast 也合法，则其行为与对应命名转换一致。不合法，则执行与 reinterpret_cast 类似的行为

4. 2 算术转换

1. 既有浮点型也有整形时，整形将转换成相应浮点型
2. 整形提升：bool,char,signed char,unsigned char,short,unsigned short 所有可能值能存于 int 则提升为 int，否则提升为 unsigned int
3. signed 类型相同则转换成相同 signed 类型中的较大类型
4. unsigned 类型大于等于 signed 类型时，signed 转换成 unsigned
5. **unsigned 类型小于 signed 类型时**：
    1. 如果 unsigned 类型所有值**能存在 signed 类型中**，则转换成 signed 类型
    2. 如果**不能，则 signed 类型转换成 unsigned 类型**

5. 初始化与赋值

很多语言中二者的区别几乎可以忽略，即使在 C++中有时这种区别也无关紧要，所以特别容易把二者混为一谈

C++中初始化和赋值是 2 个完全不同的操作

1) **显示初始化**：创建变量时的赋值行为
    1. **拷贝初始化**：`int a = 0;`
    2. **直接初始化**：`int a(0);`
    3. **初始值列表**：`int a = {0};` 或 `int a{0};`
2) **默认初始化**（程序）
    1. 局部变量
        1. non-static：内置类型非静态局部变量**不会执行默认初始化**
        2. static：如果没有初始值则使用值初始化。
    2. 全局变量：内置类型全局变量初始化为 0。
3) **值初始化**
    1. 内置类型的**值初始化为 0。**
    2. `container<T> c(n)` 只指定了容器的大小，未指定初始值，此时容器内的元素进行值初始化
    3. 使用**初始值列表**时，未提供的元素会进行值初始化
    4. 静态**局部变量会使用值初始化**

6. 声明与定义

1) 声明：`extern 类型 变量名字;`
2) **声明+定义**：`类型 变量名字;`

`extern 类型 变量名字 = 值;`（**如果在函数内则会报错**）

声明不会分配存储空间，定义会分配存储空间

7. 作用域

访问被同名局部变量覆盖的全局变量：**::变量名**（不管有多少层覆盖，都是访问全局）

8. 复合类型

8. 1 引用

1) **本质**：引用并非对象，别名，未分配空间。

2) **形式**：`int &a = b;`

理解与使用：

1) **非常量引用不能绑定到字面值或表达式的**计算结果
2) 一般来说，**引用类型和绑定的对象类型需严格匹配**
3) 程序把**引用和其初始值绑定**到一起（对引用的操作都在其绑定的对象上进行）因此一旦初始化完成，无法另引用重新绑定到另外一个对象。因此必须初始化
4) **引用本身并非对象**，故不能定义引用的引用

8.2 指针

1) 指针不同与引用，**指针**本身就是一个**对象**
2) 因为引用不是对象，没有实际地址，所以不能定义指向引用的指针
3) 指针是一个对象，所以**存在对指针的引用**
4) 一般来说，**指针类型和指向的对象类型也需严格匹配**
5) 编译器并**不负责检查**试图拷贝或以其它方式**访问无效指针**
6) 和试图使用未经初始化的变量一样，使用未经无效指针的**后果无法估计**
7) **空指针**：不指向任何对象（不要混淆空指针和**空类型**(**void**)的指针）

```
1.  int *p1 = nullptr; (C++11)
2.  int *p2 = 0;
3.  int *p3 = NULL; //#include cstdlib
```

8) 把 int 变量**直接赋给指针是错误的，即使变量的值恰好等于 0**
9) **空类型**(**void**)指针用于存放**任意对象的地址**

8.3 复合类型的声明

1）非数组与复合类型的声明

从右到左分析

```
int *&r = p; //r 是一个引用，引用一个 int 指针 p
```

变量的定义包括一个基本数据类型和一组声明符。同一条语句中，虽然基本数据类型只有一个，但是声明的形式却可以不同：

```
int* p1, p2; //p1 是一个 int*，p2 是一个 int
```

2）数组与复合类型的复杂申明

从数组名字开始，由内到外分析（数组的**维度**紧跟着被声明的名字 ）

1) 数组与指针的复杂申明 c++ `int (*Parray)[10] = &arr;` //Parry 是一个指针，指向一个含有 10 个 int 的数组
2) 数组与引用的复杂申明 c++ `int (&arrRef)[10] = arr;` //arrRef 是一个引用，引用一个含有 10 个 int 的数组
3) 数组与指针及引用的混合复杂申明 c++ `int *(&arry)[10] = ptrs;` //arry 是一个引用，引用一个包含 10 个 int 指针的数组

9. const

1）const 对象

1) **const 对象必须初始化**，因为创建后 const 对象的值就不能再改变，初始值可以是任意复杂的表达式 c++

```
const int i = get_size(); //运行时初始化
const int j = 42;         //编译时初始化
```

2) 只能在 const 类型的对象上执行**不改变其内容的操作**
3) 当以**编译时初始化的方式**定义一个 const 对象时，编译器将在**编译过程中**把用到该对象的地方**替换成对应值**
4) **默认状态下，const 对象仅在文件内有效**。多个文件的同名 const 对象等同于在不同文件中定义了独立的变量
5) 要在**多个文件之间共享同一个 const 对象，需在定义和声明时都加上 extern**

2）const 的引用（常量引用）

6) **不能修改所绑定的对象**

7) 和非常量引用不同，**常量引用可以使用字面值或任意表达式作为初始值**（原因:绑定了一个**临时量常量**）

3）指针与 const

1) **所指地址内容不可修改的指针**(并不一定要指向常量，只是为了说明无法修改所指的对象) c++ `const int *a = &b;`

2) **const 指针**(**常量指针**)：不能修改指针，将一直指向一个地址，因此必须初始化。但是指向的对象不是常量的话，可以修改指向的对象

```
int *const a = &b;
const double *const pip = &pi; //pip 是一个常量指针，指向的对象是一个双精度浮点型常量
```

4）顶层 const 与底层 const

1) **顶层 const**：无法修改指针本身（顶层是一种直接的关系） c++

```
const int ci = 123;     int *const a = &b;
```

2) **底层 const**：无法修改所指的对象（底层是一种间接的关系）

   1. 用于**声明引用的 const 都是底层 const**

10. constexpr 与常量表达式

1) **常量表达式**：在“编译过程”就能确定结果的表达式。

   包括：

   字面值

   常量表达式初始化的 const 对象

   以下不是常量表达式

```
int s = 123;
const int sz = get_size();
```

2) **constexpr 变量(C++11)**：

   变量声明为 contexpr 类型，编译器会检查变量的值是否是个常量表达式

```
constexpr int mf = 20          //20 是常量表达式
constexpr int limit = mf + 1;  //mf + 1 是常量表达式
const int sz = size();         //只有当 size 是一个 constexpr 函数时，声明才正确
```

3) **constexpr 函数**：这种函数足够简单以使编译时就可以计算其结果。

4) **字面值类型**：能使用 constexpr 声明的类型应该足够简单，称为字面值类型

包括

- 算数类型
- 引用 & 指针
  - constexpr 的指针初始值必须是 nullptr，0 或存储于某个固定地址中的对象
  - 一般来说**全局变量**和**静态局部变量**的地址不变
  - constexpr 指针，constexpr 只对指针有效，与指针所指对象无关
  - `constexpr const int *p = &i //p 是常量指针，指向整形常量 i`

不包括

- 自定义类型
- I/O 库
- string 字符串

11. 类型别名

- **typedef**：`typedef double wages;`
- **using**（C++11）：`using SI = Sales_item;`

const 与指针的类型别名使用时，还原别名来理解 const 的限定是错误的

12. auto

1) 编译器根据**初始值判断变量**类型

2) **必须初始化**
3) 一条语句声明多个变量（**只能有一个基本类型**，const int 和 int 属于不同类型）`auto i = 0, *p = &i; //正确`
`auto sz = 0, pi = 3.14 //错误`
4) **初始值为引用时**，类型为所引对象的类型
5) auto 一般会**忽略掉顶层 const**，底层 const 会保留下来
6) 如果希望判断出的 auto 是一个顶层 const，使用 const auto。
7) 还可以将引用的类型设为 auto，此时原来的初始化规则仍然适用

13. decltype

希望**根据表达式判定变量类型**，但不用表达式的值初始化变量
`decltype(f()) sum = x;` **f()并不会被调用**，sum 为 f()的返回类型
引用从来都作为其所指对象的同义词出现，只有在 decltype 处是一个例外
1) 如果**表达式的结果**对象**能作为一条赋值语句的左值**，则表达式将向 decltype 返回一个**引用类型**
`decltype(*p) c; //错误，c 是 int &，必须初始化`
2) 变量加上括号后会被编译器视为一个表达式
`decltype((i)) d; //错误，d 是 int &，必须初始化`

2 模板与泛型编程

1. 模板函数

```
template <typename T>
int compare (const T &v1,const T &v2)
{
    if(v1 < v2) return -1;
    if(v2 < v1) return 1;
    return 0;
}
```

当调用一个函数模板时，编译器(通常)**用函数实参来为我们推断模板实参**。编译器用推断出的模板参数为我们**实例化一个特定版本的函数**，这些编译器生成的版本通常被称为模板的实例
上面的模板函数说明了编写泛型代码的两个重要原则：

- 模板中的**函数参数是 const 的引用**（保证了函数可以用于不能拷贝的类型。同时，如果 compare 用于处理大对象，这种设计策略还能使函数运行得更快）
- 函数体中的**条件判断仅使用<比较运算**（如果编写代码时只使用<运算符，就降低了 compare 函数对要处理的类型的要求。**这种类型必须支持<，但不必支持>**。实际上，**如果真的涉及类型无关和可移植性，应该用 less**，因为<无法比较指针，但是 less 可以）

函数模板可以声明为 inline 或 constexpr 的，如同非模板函数一样。inline 或 constexpr 说明符放在模板参数列表之后，返回类型之前。

1.1 模板参数

- 在模板定义中，**模板参数列表不能为空**
- 模板参数的名字没有什么内在含义，通常将类型参数命名为 T，但实际上可以使用任何名字
- 一个模板参数名的可用范围是在其声明之后，至模板声明或定义结束之前。模板参数会隐藏外层作用域中声明的相同名字，模板内不能重用模板参数名
- 与函数参数相同，声明中的模板参数的名字不必与定义中相同；
- **typename 和 class 并没有什么不同**，typename 可能更直观，因为 class 可能会让人觉得能使用的类型必须是类类型

1）模板类型参数

用来指定返回类型或函数类型，以及在函数体内用于变量声明或类型转换

```
//T 用作了返回类型、参数类型、变量类型
template <typename T> T foo (T* p)
{
    T tmp = *p;
```

```
    //...
    return tmp;
}
```

2）非类型模板参数

```
template<unsigned N,unsigned M>
int compare(const char (&p1)[N], const char (&p2) [M])
{
    return strcmp(p1,p2);
}
```

1) 第一个非类型模板参数表示第一个数组的长度
2) 第二个非类型模板参数表示第二个数组的长度

当调用这个模板时，compare("hi","mom");编译器会使用字面常量的大小来代替 N 和 M，从而实例化模板

非类型模板参数包括：

1) **整形**：绑定到非类型整形参数的**实参必须**是一个**常量表达式**
2) **指针或引用**：绑定到指针或引用非类型参数的**实参必须具有静态的生存期**，不能用一个普通局部变量或动态对象作为指针或引用非类型模板参数的实参。**指针也可以用 nullptr 或一个值为 0 的常量表达式来实例化**

1.2 函数形参

模板函数的**形参**中可以含有正常类型。即，不一定全必须是模板类型：

```
template <typename T> ostream &print(ostream &os,const T &obj)
{
    return os << obj;
}
```

1.3 成员模板

1）普通类的成员模板

```
class DebugDelete {
public:
    DebugDelete(std::ostream &s = std::cerr) : os(s) { }
    template <typename T> void operator( ) (T *p) const
    {os << "deleting unique_ptr" << std::endl;delete p;}
private:
    std::ostream &os;
};
```

2）类模板的成员模板

类和成员各自有自己的独立的模板参数

```
template <typename T> class Blob {
    template <typename It> Blob(It b,It e);
}
```

当在**类外定义成员模板时，必须同时为类模板和成员模板**提供模板参数：

```
template <typename T>
template <typename It>
Blob<T>::Blob(It b,It e) : data(...) {...}
```

实例化成员模板：

```
int ia[ ] = {0,1,2,3,4,5,6,7,8,9};
vector<long> vi = {0,1,2,3,...};
list<const char*> w = {"now","is","the"};
Blob<int> a1(begin(ia),end(ia));
Blob<int> a2(vi.begin( ),vi.end( ));
Blob<string> a3(w.begin( ),w.end( ));
```

2. 类模板

```
template <typename T> class Blob {
    //typename 告诉编译器 size_type 是一个类型而不是一个对象
    typedef typename std::vector<T>::size_type size_type
    //...
};
```

一个类模板的每个实例都形成一个独立的类：

```
Blob<string> names;
Blob<double> prices;
```

2.1 与模板函数的区别

1) **编译器不能为类模板推断模板参数类型**

2) **使用时必须在模板名后的尖括号中提供额外信息**

2.2 模板类名的使用

1）类内使用不需要指明

```
BlobPtr& operator++( );
```

当处于一个类模板的作用域中时，编译器处理模板自身引用时就好像我们已经提供了与模板参数匹配的实参一样

2）类外使用需要指明

```
template <typename T>
BlobPtr<T> BlobPtr<T>::operator++(int)
{
 //...
}
```

由于位于**类作用域外**，必须指出**返回类型是一个实例化的 BlobPtr**，它所用类型与类实例化所用类型一致

2.3 类模板的成员函数

类外定义成员函数时要加 template<typename T>。类模板的成员函数具有和模板相同的模板参数。因此，**定义在类模板之外的成员函数就必须以关键字 template 开始**，后接类模板参数列表：

```
template <typename T>
ret-type Blob<T>::member-name(parm-list)
```

对于一个实例化了的类模板，其成员函数只有当程序用到它时才进行实例化

```
//实例化 Blob<int>和接受 initializer_list<int>的构造函数
Blob<int> squares = {0,1,2,3,4,5,6,7,8,9};
```

如果一个成员函数没有被使用，则它不会被实例化，成员函数只有在被用到时才会进行实例化，这一特性使得即使某种类型不能完全符合模板操作的要求，我们仍然能用该类型实例化类

2.4 类型成员

假定 T 是一个模板类型参数，当编译器遇到类似 T::mem 这样的代码时，它不会知道 mem 是一个类型成员还是一个 static 数据成员，直至初始化时才会知道。但是，为了处理模板，编译器必须知道名字是否表示一个类型。例如，假定 T 是一个类型参数的名字，当编译器遇到如下形式的语句时：

```
T::size_type *p;
```

它需要知道我们是整在定义一个名为 p 的变量还是一个名为 size_type 的 static 数据成员与名为 p 的变量相乘

默认情况下，C++假定通过作用域运算符访问的名字不是类型。因此，如果我们希望使用一个模板类型参数的类型成员，就必须显示告诉编译器该名字是一个类型。通过关键字 typename 来实现这一点

2.5 类模板和友元

1）普通类中将另一模板类声明为友元

```
template <typename T> class Pal;

class C {
    //用类 C 实例化的 Pal 是 C 的一个友元
```

```
    friend class Pal<C>;
    //Pal2 的所有实例都是 C 的友元
    template <typename T> friend class Pal2;
};
```

2）模板类中将另一模板类声明为友元

```
template <typename T> class Pal;

template <typename T> class C2 {
    //C2 的每个实例将相同实例化的 Pal 声明为友元
    friend class Pal<T>;
    //Pal2 的所有实例都是 C2 的每个实例的友元
    template <typename X> friend class Pal2;
};
```

为了让所有实例成为友元，友元声明中必须使用与类模板本身不同的模板参数（上面的 X）

3）令模板自己的类型参数成为友元

```
template <typename T> class Bar{
    //将访问权限授予用来实例化 Bar 的类型
    friend T;
};
```

对于某个类型 Foo，Foo 将成为 Bar 的友元…

2.6 模板类型别名

类模板的一个实例化定义了一个类类型，可以定义一个 typedef 来引用实例化的类：

```
typedef Blob<string> StrBlob;
```

由于模板不是一个类型，所以不能定义一个 typedef 引用一个模板。即，无法定义一个 typedef 引用 Blob<T>

但是，**新标准**允许我们为类模板定义一个类型别名 **using**：

```
template <typename T> using twin = pair<T,T>;
twin<string> authors; //authors 是一个 pair<string,string>;
```

定义一个模板类型别名时，可以固定一个或多个模板参数：

```
template <typename T> using partNo = pair<T,unsigned>;
partNo<string> books; //pair<string,unsigned>;
```

2.7 类模板的 static 成员

3) **static 属于每个实例化的类类型，而不是类模板。即，每个实例化的类都有一个自己对应的 static 成员**

4) **模板类的每个 static 成员必须有且仅有一个定义。但是，类模板的每个实例都有一个独有的 static 对象**

```
template <typename T>
size_t Foo<T>::ctr = 0;
```

可通过类类型对象或作用域运算符访问：

```
Foo<int> f1;
auto ct = Foo<int>::count( );
ct = f1.count( );
```

只有使用时才会实例化

3. 模板编译

1) **遇到模板时不生成代码，实例化时生成代码**

2) 函数模板和类模板成员函数的定义通常放在头文件中

3) 实例化冗余：当模板被使用时才会进行实例化这一特性意味着，相同的实例可能出现在多个对象文件中。当两个或多个独立编译的源文件使用了相同的模板，并提供了相同的模板参数时，每个文件中就都会有该模板的一个实例

3.1 实例化声明

形式：extern template declaration

```
extern template class Blob<string>;
extern template int compare(const int&,const int&);
```

当遇到 extern 模板声明时，不会在本文件中生成实例化代码。将一个实例化声明为 extern 就表示承诺在程序其他位置有该实例化的一个定义。对于一个给定的实例化版本，可能有多个 extern 声明，但必须只有一个定义

4) 实例化声明可以有多个：即多个源文件可能含有相同声明

5) 实例化声明必须出现在任何使用此实例化版本的代码之前。因为编译器在使用一个模板时会自动对其实例化

3.2 实例化定义

```
template declaration
template int compare(const int &,const int&);
template class Blob<string>;
```

6) **类模板的实例化定义会实例化该模板的所有成员**

7) **所用类型必须能用于模板的所有成员**：与处理类模板的普通实例化不同，编译器会实例化该类的所有成员。即使我们不使用某个成员，它也会被实例化。因此，我们用来显式实例化一个类模板的类型，必须能用于模板的所有成员

4. 模板参数

4.1 默认模板实参

为模板提供默认类型

1）模板函数

```
template <typename T,typename F = less<T>>
int compare(const T &v1,const T &v2,F f = F( ))
{
    if(f(v1,v2)) return -1;
    if(f(v2,v1)) return 1;
    return 0;
}
```

和函数默认实参一样，所有提供了默认实参的形参右边的形参都需要提供默认实参

2）类模板

```
template <class T = int> class Numbers {
public:
    Numbers(T v = 0) : val(v) { }
    private:
    T val;
};

Numbers<long double> lots_of_precision;
Numbers<> average_precision;            //空<>表示希望使用默认类型;
```

4.2 模板实参推断

1）函数模板的参数转换

8) **模板类型参数的类型转换**：将实参传递给带模板类型的函数形参时，能够自动应用的类型转换只有 **const 转换**及**数组或函数到指针的转换**

1. **const 的转换**
    1. 可以将一个 const 对象传递给一个非 const 的非引用形参 c++

```
template <typename T> fobj(T,T);
string s1("a value");
const string s2("another value");
```

fobj(s1,s2); //正确； fobj 调用中，传递了一个 string 和一个 const string。虽然这些类型不严格匹配，但两个调用都是合法的，由于实参被拷贝，因此原对象是否是 const 没有关系；

2. 可以将一个非 const 对象的引用(或指针)传递给一个 const 的引用(或指针)形参 c++
```
template <typename T> fref(const T&,const T&);
string s1("a value");
const string s2("another value");
fref(s1,s2); //正确； 在 fref 调用中，参数类型是 const 的引用。对于一个引用参数来说，转换为 const 是允许的，因此合法；
```

2. **非引用类型形参可以对数组或函数指针应用正常的指针转换** c++ `template <typename T> fobj(T,T); template <typename T> fref(const T&,const T&); int a[10],b[42]; fobj(a,b); //调用 fobj(int*,int*) fref(a,b); //错误，数组类型不匹配；` 在 fobj 调用中，数组大小不同无关紧要，两个数组都被转换为指针。fobj 中的模板类型为 Int*；但是，fref 调用是不合法的，如果形参是一个引用，则数组不会转换为指针。a 和 b 的类型不匹配

9) **普通类型参数的类型转换**：模板函数可以有用普通类型定义的参数，即，不涉及模板类型参数的类型。这种函数实参不进行特殊处理，这些实参执行正常类型的转换

2）显示实参

为什么需要显示实参？编译器无法推断出模板实参的类型。假设定义如下模板:
```
template <typename T>
T sum(T,T);
```
则调用 sum 时，必须要求传入相同类型的参数，否则会报错。因此可以按这种方式定义模板：
```
template <typename T1,typename T2,typename T3>
T1 sum(T2,T3);
```
但是，这种情况下，无论传入什么函数实参，都无法推断 T1 的类型。因此，每次调用 sum 时，调用者必须为 T1 提供一个显示实参：
```
auto val3 = sum<long long>(i,lng);
```
这个调用显示指定了 T1 的类型，而 T2 和 T3 的类型则由编译器从 i 和 lng 的类型判断出来

显示实参配对顺序：由左至右。只有尾部参数的显示模板实参可以忽略，但必须能推断出来

因此，如果按找这种形式定义模板：
```
template <typename T1,typename T2,typename T3>
T3 sum(T2,T1);
```
则总是必须为所有三个形参指定参数。希望控制模板实例化

对于 sum 模板，如果保留原有的设计：`template T sum(T,T)` 则当函数调用传入不同类型的参数时，我们必须放弃参数类型推断，采取控制模板实例化的方式来调用：`sum<int>(long,1024);` 这种情况下，会实例化一个 `int sum(int,int)`的函数，传入的参数都会按照内置类型的转换规则转换为 int

3）尾置返回类型与 traits

当我们希望用户确定返回类型时，用显示模板实参表示模板函数的返回类型是很有效的。在其他情况下，要求显示指定模板实参会给用户增添额外负担，而且不会带来什么好处：
```
template <typename It>
??? &fcn(It beg,It end)
{
    //处理序列
    return *beg;
}
```
在这个例子中，并不知道返回结果的准确类型，但知道所需类型是所处理的序列的元素类型;我们知道函数应该返回`*beg`，可以使用 `decltype(*beg)`来获取此表达式的类型。但是在编译器遇到函数的参数列表之前，beg 是不存在的。所以必须使用尾置类型：

```
template <typename It>
auto fcn(It beg,It end) -> decltype(*beg)
{
    //处理序列
    return *beg;//返回序列中一个元素的引用
}
```

也可以使用标准库的类型转换模板。可以使用 remove_reference 来获得元素类型。这个模板有一个模板类型参数和一个名为 type 的成员。如果用一个引用类型实例化这个模板，则 type 将表示被引用的类型。如果实例化 remove_reference<int&>，则 type 成员将是 int。因此，可以通过下列模板满足需求：

```
template <typename It>
auto fcn2(It beg,It end) ->
typename remove_reference<decltype(*beg)>::type
{
     //处理序列
    return *beg;
}
```

4）函数指针和实参推断

用一个函数模板初始化一个函数指针或为一个函数指针赋值时，编译器使用指针的类型来推断模板实参

```
template <typename T> int compare(const T&,const T&);
int (*pf1)(const int&,const int&) = compare;
```

pf1 中参数的类型决定了 T 的模板实参的类型。如果不能从函数指针类型确定模板实参，则产生错误：

```
void func(int(*)(const string&,const string&));
void func(int(*)(const int&,const int&));
func(compare); //错误，使用那个实例?
```

对于这种情况，可以使用显示模板实参：

```
func(compare<int>);
```

5）引用与实参推断

非常重要的是记住两个规则：

- 编译器会应用正常的引用绑定规则；
- const 是底层的，不是顶层的；

当一个函数的参数是模板类型参数的一个普通(左值)引用时，绑定规则告诉我们，只能传递给它一个左值：

```
template <typename T> void f1(T&);
f1(i);  //i 是 int，T 推断为 int；
f1(ci); //ci 是 const int，T 推断为 const int；
f1(5);  //错误
```

如果参数类型是 const T&，正常的绑定规则告诉我们可以传递给它任何类型的实参：一个对象，临时对象或字面值常量：

```
template <typename T> void f2(const T&);
f2(i); //i 是 int，T 推断为 int；
f2(ci); //ci 是 const int，但 T 推断为 int；
f2(5); //T 推断为 int；
```

当参数是一个右值引用时，正常绑定规则告诉我们可以传递给它一个右值：

```
template <typename T> void f3(T&&);
f3(42); //实参是 int 型的右值，T 推断为 int；
```

引用折叠：

1. 如果将一个左值传递给函数的右值引用参数，且此右值引用指向模板类型参数(如:T&&)时，编译器推断模板的类型参数为左值引用类型 2. 如果因为 1.间接的创建了一个引用的引用，则引用形参了“折叠”、则： * 右值引用的右值引用会被折叠成右值引用 * 其它情况下都折叠成左值引用

因此，对于前面的 f3：

```
f3(i); //i 是左值，T 推断为 int&，T&&被折叠成 int &;
f3(ci); //ci 是左值，T 是 const int&;
```

因此，如果模板参数类型为右值引用，可以传递给它任意类型的实参

右值引用的问题：因为可以传递任意实参，引用折叠会导致 T 被推断为引用或非引用类型，所以函数内使用这个类型在传入不同参数时可能产生不同结果，此时，编写正确的代码就变得异常困难；

右值引用的使用场景：因为上述问题，所以右值引用主要应用于两个场景

- **模板转发其实参**：当使用右值引用作为模板参数时，如果 T 被推断成普通类型(即非引用)，可以通过 std::forward 保持其右值属性，会返回一个 T&&。如果被推断成一个(左值)引用，通过引用折叠，最终也还是会返回 T&；因此，当用于一个指向模板参数类型的右值引用函数参数(T&&)时，forward 会保持实参类型的所有细节
- **模板被重载**

5. 重载与模板

包含模板的函数匹配规则：

- **候选函数包括所有模板实参推断成功的函数模板实例** c++
- template <typename T> string debug_rep(const T &t) {...}
- template <typename T> string debug_rep(T *p) {...}
- string s("hi"); //第二个模板实参推断失败，所以调用第一个模板；
- cout << debug_rep(s) << endl;
- **可行函数按类型转换来排序**
- **如果恰好有一个比其他提供更好的匹配则使用该函数** c++
- template <typename T> string debug_rep(const T &t) {...}
- template <typename T> string debug_rep(T *p) {...}
- string s("hi"); //两个模板都能匹配：
- //第一个模板实例化 debug_rep(const string*&)，T 被绑定到 string*；
- //第二个模板实例化 debug_rep(string*)，T 被绑定到 string；
- //但由于第一个实例化版本需要进行普通指针到 const 指针的转换，所以第二个更匹配；
- cout << debug_rep(&s) << endl;
- **如果有多个函数提供"同样好的"匹配**

  同样好的函数中只有一个是非模板函数，则选择此函数 c++

```
template <typename T> string debug_rep(const T &t) {...}
template <typename T> string debug_rep(T *p) {...}
string debug_rep(const string &s) {...}
string s("hi");
//以下调用有两个同样好的可行函数：
//第一个模板实例化 debug_rep<string>(const string &)，T 被绑定到 string；
 //非模板版本 debug_rep(const string &s)；
//编译器会选择非模板版本，因为最特例化；
cout << debug_rep(s) << endl;
```

  同样好的函数中全是模板函数，选择更"特例化的模板" c++

```
template <typename T> string debug_rep(const T &t) {...}
 template <typename T> string debug_rep(T *p) {...}
 string s("hi");
 const string *sp = &s;    //以下调用两个模板实例化的版本都能精确匹配：
 //第一个模板实例化 debug_rep(const string *&)，T 被绑定到 string*；
//第二个模板实例化 debug_rep(const string *)，T 被绑定到 const string；
//我们可能觉得这个调用是有歧义的。但是，根据重载函数模板的特殊规则，调用被解析为
debug_rep(T*)，即更特例化的版本；
```

//如果不这样设计，将无法对一个 const 的指针调用指针版本的 debug_rep。
//问题在于模板 debug_rep(const T&)本质上可以用于任何类型，包括指针类型。此模板比 debug_rep(T*)更通用，后者只能用于指针类型；
cout << debug_rep(sp) << endl;
否则，调用有歧义

6. 可变参数模板

参数包：

10) **模板参数包**： template<typename T,typename... Args> Args 为模板参数包，class...或 typename...指出接下来的参数表示零个或多个类型的列表，一个类型名后面跟一个省略号表示零个或多个给定类型的非类型参数的列表；
11) **函数参数包** c++
```
template <typename T,typename... Args>
void foo(const T &t,const Args& ... rest);
```
rest 为函数参数包

使用参数包：

12) sizeof... 获取参数包大小。可以使用 sizeof...运算符获取包中元素的数目
13) **扩展包**：扩展一个包就是将包分解为构成的元素，对每个元素应用模式，获得扩展后的列表，通过在模式右边放一个省略号来触发扩展操作： c++
```
template <typename T,typename... Args>
ostream& print(ostream &os,const T &t,const Args&... rest) //扩展 Args    {
os << t << ", ";        return print(os,rest...); //扩展 rest    }
```
扩展中的模式会独立地应用于包中的每个元素： c++ debug_res(rest)... 是对包 rest 的每一个元素调用 debug_res；debug_res(rest...) 是调用一个参数数目和类型与 rest 中元素匹配的 debug_rest；

转发包参数：

新标准下，可以组合使用可变参数模板与 forward 机制来编写函数，实现将其参数不变地传递给其他函数：

```
template <typename... Args>
void fun(Args&&... args) //将 Args 扩展为一个右值引用的列表
{
    //work 的实参既扩展 Args 又扩展 args
    work(std::forward<Args>(args)...);
}
```

7. 模板特例化

编写单一模板，使之对任何可能的模板实参都是最合适的，都能实例化，这并不总是能办到。当我们不能（或不希望）使用模板版本时，可以定义类或函数模板的一个特例化版本
一个特例化版本本质上是一个实例，而非函数名的一个重载版本。因此，特例化不影响函数匹配；

14) **函数模板特例化** c++
```
template <typename T> int compare(const T&,const T&);
 //compare 函数模板的通用定义不适合字符指针的情况，
//我们希望 compare 通过 strcmp 比较两个字符指针而非比较指针值；
template <>   int compare(const char* const &p1,const char* const &p2)
{       return strcmp(p1,p2);   }
```
当定义一个特例化版本时，函数参数类型必须与一个先前声明的模板中对应的类型匹配。这个特例化版本中，T 为 const char*，先前声明的模板要求一个指向此类型 const 版本的引用。一个指针类型的 const 版本是一个常量指针而不是指向 const 类型的指针。需要在特例化版本中使用的类型是 const char* const &，即一个指向 const char 的 const 指针的引用；
15) **类模板特例化** c++ template <> struct 模板类名<Sales_data> { ... } 定义了某个模板能处理 Sales_data 的特例化版本

16) **类模板（偏特化）部分特例化**：与函数模板不同，类模板的特例化不必为所有模板参数提供实参。可以只提供一部分而非所有模板参数，或是参数的一部分而非全部特性。部分特例化本身是一个模板，**部分特例化版本的模板参数列表是原始模板的参数列表的一个子集或者是一个特例化版本** c++

```
//原始的，最通用的版本
template <class T> struct remove_reference    { typedef T type; };
//部分特例化版本，将用于左值引用和右值引用
template <class T> struct remove_reference<T&>   { typedef T type; };
template <class T> struct remove_reference<T&&>   { typedef T type; };  //用例
int i;   remove_reference<decltype(42)>::type a;
//decltype(42)为 int，使用原始模板；
remove_reference<decltype(i)>::type b;
//decltype(i)为 int&，使用第一个部分特例化版本；
 remove_reference<decltype(std::move(i))>::type c;
 //decltype(std::move(i))为 int&&，使用第二个部分特例化版本；
```

17) **特例化成员而非类** c++

```
 template <>   void Foo<int>::Bar( )   {       //进行应用于 int 的特例化处理；    }
Foo<string> fs; //实例化 Foo<string>::Foo( );
fs.Bar( );      //实例化 Foo<string>::Bar( );
Foo<int> fi;    //实例化 Foo<int>::Foo( );
fi.Bar( );      //使用特例化版本的 Foo<int>::Bar( );
```

3 内存管理

1. new 和 delete

1.1 new

1）动态分配单个对象

初始化：

```
int *pi1 = new int;          //默认初始化
int *pi2 = new int();        //值初始化
int *pi2 = new int(1024);    //直接初始化
string *ps = new string(10,'9');
//若 obj 是一个 int，则 p1 是 int*；
//不能用{...}代替(obj)包含多个对象；
auto p1 = new auto(obj);
```

动态分配 **const 对象**：

1) 必须进行**初始化**
2) **不能修改指向的对象**，但是能 delete(销毁)这个动态分配的 const 对象

```
const int *pci = new const int(1024);
const string *pcs = new const string;    //隐式初始化
```

内存耗尽：

1) 内存不足时，new 会失败，抛出类型为 **bad_alloc 的异常**
2) new (nothrow) T 可以**阻止抛出异常**（定位 new）

2）动态分配多个对象

使用注意：

1) 大多数应用**应该使用标准库容器**而不是动态分配的数组
2) 动态分配数组的**类必须定义**自己版本的**拷贝，复制，销毁**对象的操作

理解：

1) 通常**称 new T[]**分配的内存为“动态数组”某种程度上有些误导
   返回的并不是一个“数组类型”的对象，而是**一个”数组元素类型“的指针**
2) 即使使用类型别名也不会分配一个数组类型的对象

不能创建大小为 0 的动态数组，但**当[n]中 n 为 0 时，是合法的**。此时 new 返回一个合法的非空指针，**此指针保证与 new 返回的其它任何指针都不同**，就像尾后指针一样，**可以进行比较操作，加 0，减 0，不能解引用**

初始化：

```
int *pia = new int[get_size()];  //维度不必是常量，但是必须是整形
int *p1 = new int[42];               //未初始化
//以下为上一行的等价调用
typedef int arrT[42];
int *p = new arrT;
int *p2 = new int[42]();             //值初始化

//初始值列表中没有给定初始值的元素进行”值初始化“，如果初始值列表中元素超出，new 会失败
int *p3 = new int[5]{1,2,3,4,5};
```

1.2 delete

1) delete 单个对象：delete p;
2) delete 动态数组：delete [] pa;

- 不管分配时有没有用类型别名，**delete 时都要加上[]**
- **逆序销毁**
- []指示编译器指针指向的是一个**数组的首元素**

注意：

1) 不要 delete 非 new 分配的对象
2) 不要重复 delete

3) **可以 delete 空指针**
4) 可以 delete 动态分配的 const 对象

通常情况下，编译器不能分辨一个指针指向的是静态还是动态分配的对象。类似的，编译器也不能分辨一个指针所指向的内存是否已经被释放了。对于这些 delete 表达式，大多数编译器能通过，尽管它们是错误的。这些错误 delete 的结果是未定义的

空悬指针：指向原本存在数据现在已经**无效的内存**的指针

1) 当 delete 一个动态分配的对象后，原本指向这个对象的指针就变成了空悬指针
2) **防止使用空悬指针**（只能保证这个指针不会再访问无效内存，但是可能也还有其它指针也指向这块动态分配的内存，它们在 delete 后也可能会访问）
    1. 在**即将离开指针作用域时 delete**：这样之后，当离开作用域后这个指针就销毁了，而在 delete 前，指针指向的内存是有效的
    2. **delete 后赋值为空指针 nullptr**

2. 智能指针

2.1 通用操作

以下操作支持 shared_ptr 和 unique_ptr

1) **创建** `//默认初始化，保存一个空指针`
```
shared_ptr<T> sp;
unique_prt<T> up;
```
2) 作为条件：p
3) **访问指向的对象**：*p
4) **获取保存指针**：p.get()
    1. **不要 delete get()返回的指针**，假设 delete 没问题，**在引用计数为 0 时，智能指针会重复 delete**
    2. 如果 p 是 shared_ptr，**不要用 get()返回的指针初始化另一个 shared_ptr，这样不会递增引用计数**，当新建智能指针销毁后，这个动态对象就被释放了
5) 交换 c++ `swap(p,q);` `p.swap(q);`

2.2 shared_ptr

1）创建：

1) 调用函数 make_shared
    1. `make_shared<T>(args)`：推荐使用这种方式。
    args 用于初始化指向的对象，不传参数时”值初始化“

```
shared_ptr<int> p1 = make_shared<int>(42);
//动态对象初始化为 42
shared_ptr<string> p2 = make_shared<string>(10,'9');
//动态对象初始化为"9999999999"
shared_ptr<int> p3 = make_shared<int>();
//动态对象值初始化，0
```

2) 使用构造函数
    1. `shared_ptr<T> p(q)`
        1. **q 为 shared_ptr 时，会递增 q 的引用计数**
        2. **构造函数为 explicit，**如果 q **不是一个智能指针**，必须**直接初始化**，此时 q 必须能转换为 T*，如
        `shared_ptr<int> p(new int(1024))`
        3. **如果 q 不是一个指向动态内存的指针，须自定义释放操作。**shared_ptr 默认使用 delete 释放所指动态对象，如果指针不指向动态内存，不能 delete）
        4. **q 不是智能指针时，这种方式构建临时 shared_ptr 很危险**（比如一个函数参数为 shared_ptr，**由于 explicit，因此不能隐式转换**。如果 q 是 new int 创建的内置类型指针，则可能通过这个构造函数创建一个临时 shared_ptr 来满足调用要求，这样的话当函数返回时，两个 shared_ptr(形参与实参)都被销毁，所以函数外部原本指针指向的动态对象会被释放掉，在函数调用之后再使用就是空悬指针，**因此，最好使用 make_shared 来创建智能指针**）
    2. `shared_ptr<T> p(q,d)`：d 是可调用对象，用于代替 delete 执行释放操作，在这里 q 可以不指向动态内存

3. shared_ptr<T> p(p2,d)：p 是 shared_ptr p2 的拷贝，但是使用可调用对象 d 代替 delete 执行释放操作
4. shared_ptr<T> p(u)：从 unique_ptr u 那里接管了对象的所以权，将 u 置为空

2）赋值

```
p = q;  //递增 q 引用计数，递减 p 引用计数
```

3）重置

```
// 1）若 p 是唯一指向其对象的 shared_ptr，则释放对象；
// 2）将 p 置为空；
p.reset();

// 1）若 p 是唯一指向其对象的 shared_ptr，则释放对象；
// 2）p = q；
p.reset(q);

// 1）若 p 是唯一指向其对象的 shared_ptr，则释放对象；
// 2）p = q；
// 3）d 代替 delete 执行释放操作；
p.reset(q,d);
```

4）状态

```
//返回与 p 共享对象的智能指针数量；可能很慢，主要用于调试
p.use_count();

//若 use_count()为 1 则返回 true，否则返回 false
p.unique();
```

2.3 unique_ptr

1）初始化

```
unique_ptr<T> u1;            //创建一个空的 unique_ptr
unique_ptr<T D> u2;          //D 为自定义释放操作的类型
//D 为自定义释放操作的类型，d 为自定义释放操作的指针。这里没有传入指针参数，是一个空 unique_ptr
unique_ptr<T,D> u(d);
unique_ptr<T,D> u(T*,d);
```

2）赋值与拷贝

只有在 unique_ptr 即将销毁时才能赋值或拷贝。如：当函数返回一个局部 unique_ptr 时

3）交出控制权

```
//返回指针，放弃对指针的控制权，并将 u 置为空
//不会释放，主要目的在于切断与原来管理对象的联系，将其交由其它 unique_ptr 来管理
u.release()
p.release()             //内存泄露
auto pp = p.release()  //要记得 delete pp
```

4）释放

```
u = nullptr;         //释放 u 指向的对象，将 u 置为空；
u.reset();           //释放 u 指向的对象，并将 u 置为空；
u.reset(q);          //释放 u 指向的对象，转为控制指针 p 指向的对象
u.reset(nullptr);    //释放 u 指向的对象，并将 u 置为空；
```

5）管理动态数组

shared_ptr 不直接管理动态数组，如果要用 shared_ptr 来管，须提供自定义的删除操作，因为默认情况下 shared_ptr 使用 delete 销毁所指对象。但即使如此，也不能用下标访问每个元素，需要用 get()函数。

unique_ptr 可以用下标访问.

```
unique_ptr<int[]> up(new int[10]); //创建
up.release();     //放弃对指针的控制权，并将 u 置为空（不会释放。测试如此，和书本不同）
up[i];            //返回位置 i 处的对象，左值；
```

2.4 weak_ptr 解决循环引用

1）初始化

```
//空weak_ptr，可以指向类型为T的对象
weak_ptr<T> w;

//与shared_ptr sp指向相同对象的weak_ptr，T必须能转换为sp指向的类型
weak_ptr<T> w(sp);
```

2）赋值

```
w = p;  //p是shared_ptr或weak_ptr，赋值后w与p共享对象
```

3）重置

```
w.reset();      //将w置为空（不会释放对象）
```

4）状态

```
//返回与w共享对象的“shared_ptr”的数量
w.use_count();

//如果共享对象的"shared_ptr"为0(没有共享对象的shared_ptr)，则返回true，否则返回false
w.expired();
```

5）访问

```
//获取shared_ptr
//   如果没有共享对象的shared_ptr，则返回一个空的shared_ptr；
//   否则返回一个指向共享对象的shared_ptr；
//这种访问方式提供了对动态对象的安全访问；
w.lock();
```

智能指针

1. 智能指针背后的设计思想

我们先来看一个简单的例子：

```
void remodel(std::string & str)
{
    std::string * ps = new std::string(str);
    ...
    if (weird_thing())
        throw exception();
    str = *ps;
    delete ps;
    return;
}
```

当出现异常时（weird_thing()返回 true），delete 将不被执行，因此将导致内存泄露。 如何避免这种问题？有人会说，这还不简单，直接在 throw exception();之前加上 delete ps;不就行了。是的，你本应如此，问题是很多人都会忘记在适当的地方加上 delete 语句（连上述代码中最后的那句 delete 语句也会有很多人忘记吧），如果你要对一个庞大的工程进行 review，看是否有这种潜在的内存泄露问题，那就是一场灾难！ 这时我们会想：当 remodel 这样的函数终止（不管是正常终止，还是由于出现了异常而终止），本地变量都将自动从栈内存中删除—因此指针 ps 占据的内存将被释放，**如果 ps 指向的内存也被自动释放，那该有多好啊。** 我们知道析构函数有这个功能。如果 ps 有一个析构函数，该析构函数将在 ps 过期时自动释放它指向的内存。但 ps 的问题在于，它只是一个常规指针，不是有析构函数的类对象指针。如果它指向的是对象，则可以在对象过期时，让它的析构函数删除指向的内存。

这正是 auto_ptr、unique_ptr 和 shared_ptr 这几个智能指针背后的设计思想。我简单的总结下就是：**将基本类型指针封装为类对象指针**（这个类肯定是个模板，以适应不同基本类型的需求），并在析构函数里编写 delete 语句删除指针指向的内存空间。

因此，要转换 remodel()函数，应按下面 3 个步骤进行： - 包含头文件 memory（智能指针所在的头文件）； - 将指向 string 的指针替换为指向 string 的智能指针对象； - 删除 delete 语句。

下面是使用 auto_ptr 修改该函数的结果：

```
# include <memory>
void remodel (std::string & str)
{
    std::auto_ptr<std::string> ps (new std::string(str));
    ...
    if (weird_thing ())
        throw exception();
    str = *ps;
    // delete ps;  NO LONGER NEEDED
    return;
}
```

2. C++智能指针简单介绍

STL 一共给我们提供了四种智能指针：auto_ptr、unique_ptr、shared_ptr 和 weak_ptr（本文章暂不讨论）。 模板 auto_ptr 是 C++98 提供的解决方案，C+11 已将将其摒弃，并提供了另外两种解决方案。然而，虽然 auto_ptr 被摒弃，但它已使用了好多年：同时，如果您的编译器不支持其他两种解决力案，auto_ptr 将是唯一的选择。

使用注意点 - 所有的智能指针类都有一个 explicit 构造函数，以指针作为参数。比如 auto_ptr 的类模板原型为：

```
templet<class T>
class auto_ptr {
    explicit auto_ptr(X* p = 0) ;
    ...
};
```

因此不能自动将指针转换为智能指针对象，必须显式调用：

```
shared_ptr<double> pd;
double *p_reg = new double;
pd = p_reg;                                  // not allowed (implicit conversion)
pd = shared_ptr<double>(p_reg);              // allowed (explicit conversion)
shared_ptr<double> pshared = p_reg;          // not allowed (implicit conversion)
shared_ptr<double> pshared(p_reg);           // allowed (explicit conversion)
```

对全部三种智能指针都应避免的一点：

```
string vacation("I wandered lonely as a cloud.");
shared_ptr<string> pvac(&vacation);   // No
```

智能指针 pvac 过期时，程序将把 delete 运算符用于**非堆内存**，这是**错误**的。

使用举例

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

class report
{
private:
    std::string str;
public:
 report(const std::string s) : str(s) {
  std::cout << "Object created.\n";
 }
 ~report() {
  std::cout << "Object deleted.\n";
 }
 void comment() const {
  std::cout << str << "\n";
 }
};
```

```
int main() {
 {
  std::auto_ptr<report> ps(new report("using auto ptr"));
  ps->comment();
 }

 {
  std::shared_ptr<report> ps(new report("using shared ptr"));
  ps->comment();
 }

 {
  std::unique_ptr<report> ps(new report("using unique ptr"));
  ps->comment();
 }
 return 0;
}
```

3. 为什么摒弃 auto_ptr？

先来看下面的赋值语句:

```
auto_ptr< string> ps (new string ("I reigned lonely as a cloud."));
auto_ptr<string> vocation;
vocaticn = ps;
```

上述赋值语句将完成什么工作呢？如果 ps 和 vocation 是常规指针，则两个指针将指向同一个 string 对象。这是不能接受的，**因为程序将试图删除同一个对象两次——一次是 ps 过期时，另一次是 vocation 过期时**。要避免这种问题，方法有多种： - 定义赋值运算符，使之执行深复制。这样两个指针将指向不同的对象，其中的一个对象是另一个对象的副本，缺点是浪费空间，所以智能指针都未采用此方案。 - 建立所有权（ownership）概念。对于特定的对象，只能有一个智能指针可拥有，这样只有拥有对象的智能指针的构造函数会删除该对象。然后让赋值操作转让所有权。这就是用于 auto_ptr 和 uniqiie_ptr 的策略，但 unique_ptr 的策略更严格。 **- 创建智能更高的指针，跟踪引用特定对象的智能指针数。这称为引用计数。例如，赋值时，计数将加 1，而指针过期时，计数将减 1,。当减为 0 时才调用 delete。这是 shared_ptr 采用的策略**。

当然，同样的策略也适用于复制构造函数。 每种方法都有其用途，但为何说要摒弃 auto_ptr 呢？ 下面举个例子来说明。

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

int main() {
  auto_ptr<string> films[5] =
 {
  auto_ptr<string> (new string("Fowl Balls")),
  auto_ptr<string> (new string("Duck Walks")),
  auto_ptr<string> (new string("Chicken Runs")),
  auto_ptr<string> (new string("Turkey Errors")),
  auto_ptr<string> (new string("Goose Eggs"))
 };
 auto_ptr<string> pwin;
 pwin = films[2]; // films[2] loses ownership. 将所有权从films[2]转让给pwin，此时
films[2]不再引用该字符串从而变成空指针

 cout << "The nominees for best avian baseballl film are\n";
 for(int i = 0; i < 5; ++i)
  cout << *films[i] << endl;
```

```
 cout << "The winner is " << *pwin << endl;
 cin.get();

 return 0;
}
```

运行下发现程序崩溃了，原因在上面注释已经说的很清楚，films[2]已经是空指针了，下面输出访问空指针当然会崩溃了。但这里如果把 auto_ptr 换成 shared_ptr 或 unique_ptr 后，程序就不会崩溃，原因如下：- 使用 shared_ptr 时运行正常，因为 shared_ptr 采用引用计数，pwin 和 films[2]都指向同一块内存，在释放空间时因为事先要判断引用计数值的大小因此不会出现多次删除一个对象的错误。- **使用 unique_ptr 时编译出错**，与 auto_ptr 一样，**unique_ptr 也采用所有权模型**，但在使用 unique_ptr 时，程序不会等到运行阶段崩溃，而在编译器因下述代码行出现错误：

```
unique_ptr<string> pwin;
pwin = films[2]; // films[2] loses ownership.
```

指导你发现潜在的内存错误。

这就是为何要摒弃 auto_ptr 的原因，一句话总结就是：**避免潜在的内存崩溃问题。**

4. unique_ptr 为何优于 auto_ptr？

可能大家认为前面的例子已经说明了 unique_ptr 为何优于 auto_ptr，也就是安全问题，下面再叙述的清晰一点。 请看下面的语句:

```
auto_ptr<string> p1(new string ("auto") ;   //#1
auto_ptr<string> p2;                        //#2
p2 = p1;                                    //#3
```

在语句#3 中，p2 接管 string 对象的所有权后，p1 的所有权将被剥夺。前面说过，这是好事，可防止 p1 和 p2 的析构函数试图删同一个对象； 但如果程序随后试图使用 p1，这将是件坏事，因为 p1 不再指向有效的数据。

下面来看使用 unique_ptr 的情况：

```
unique_ptr<string> p3 (new string ("auto");   //#4
unique_ptr<string> p4;                        //#5
p4 = p3;                                      //#6
```

编译器认为语句#6 非法，避免了 p3 不再指向有效数据的问题。因此，unique_ptr 比 auto_ptr 更安全。

但 unique_ptr 还有更聪明的地方。 有时候，会将一个智能指针赋给另一个并不会留下危险的悬挂指针。

假设有如下函数定义：

```
unique_ptr<string> demo(const char * s)
{
    unique_ptr<string> temp (new string (s));
    return temp;
}
```

并假设编写了如下代码：

```
unique_ptr<string> ps;
ps = demo('Uniquely special");
```

demo()返回一个临时 unique_ptr，然后 ps 接管了原本归返回的 unique_ptr 所有的对象，而返回时临时的 unique_ptr 被销毁，也就是说没有机会使用 unique_ptr 来访问无效的数据，换句话来说，这种赋值是不会出现任何问题的，即没有理由禁止这种赋值。实际上，编译器确实允许这种赋值，这正是 unique_ptr 更聪明的地方。

总之，当程序试图将一个 **unique_ptr 赋值给另一个时**，如果源 **unique_ptr 是个临时右值，编译器允许这么做**；如果源 unique_ptr 将存在一段时间，编译器将禁止这么做，比如：

```
unique_ptr<string> pu1(new string ("hello world"));
unique_ptr<string> pu2;
pu2 = pu1;                                    // #1 not allowed
unique_ptr<string> pu3;
pu3 = unique_ptr<string>(new string ("You"));   // #2 allowed
```

其中#1 留下悬挂的 unique_ptr(pu1)，这可能导致危害。而#2 不会留下悬挂的 unique_ptr，因为它调用 unique_ptr 的构造函数，该构造函数创建的临时对象在其所有权让给 pu3 后就会被销毁。**这种随情况而已的行为表明，unique_ptr 优于允许两种赋值的 auto_ptr 。**

当然，您可能确实想执行类似于#1 的操作，仅当以非智能的方式使用摒弃的智能指针时（如解除引用时），这种赋值才不安全。要安全的重用这种指针，可给它赋新值。**C++有一个标准库函数 std::move()，让你能够将一个 unique_ptr 赋给另一个**。下面是一个使用前述 demo()函数的例子，该函数返回一个 unique_ptr 对象：**使用 move 后，原来的指针仍转让所有权变成空指针，可以对其重新赋值**。

```
unique_ptr<string> ps1, ps2;
ps1 = demo("hello");
ps2 = move(ps1);
ps1 = demo("alexia");
cout << *ps2 << *ps1 << endl;
```

5. 如何选择智能指针？

在掌握了这几种智能指针后，大家可能会想另一个问题：在实际应用中，应使用哪种智能指针呢？ 下面给出几个使用指南。

（1）如果程序要使用**多个指向同一个对象的指针，应选择 shared_ptr**。这样的情况包括：

- 有一个**指针数组**，并使用一些**辅助指针来标示特定的元素**，如最大的元素和最小的元素；
- **两个对象包含都指向第三个对象的指针**；
- **STL 容器包含指针**。很多 STL 算法都支持复制和赋值操作，这些操作可用于 shared_ptr，但不能用于 unique_ptr（编译器发出 warning）和 auto_ptr（行为不确定）。如果你的编译器没有提供 shared_ptr，可使用 **Boost 库提供的 shared_ptr。**

（2）如果程序**不需要多个指向同一个对象的指针，则可使用 unique_ptr**。如果**函数使用 new 分配内存，并返还指向该内存的指针，将其返回类型声明为 unique_ptr 是不错的选择**。这样，所有权转让给接受返回值的 unique_ptr，而该智能指针将负责调用 delete。**可将 unique_ptr 存储到 STL 容器在那个，只要不调用将一个 unique_ptr 复制或赋给另一个算法（如 sort()）**。例如，可在程序中使用类似于下面的代码段。

```
unique_ptr<int> make_int(int n)
{
    return unique_ptr<int>(new int(n));
}
void show(unique_ptr<int> &p1)
{
    cout << *a << ' ';
}
int main()
{
    ...
    vector<unique_ptr<int> > vp(size);
    for(int i = 0; i < vp.size(); i++)
        vp[i] = make_int(rand() % 1000);              // copy temporary unique_ptr
    vp.push_back(make_int(rand() % 1000));     // ok because arg is temporary
    for_each(vp.begin(), vp.end(), show);           // use for_each()
    ...
}
```

其中 **push_back 调用没有问题**，因为它返回一个**临时 unique_ptr**，该 unique_ptr 被赋给 vp 中的一个 unique_ptr。另外，**如果按值而不是按引用给 show()传递对象，for_each()将非法**，因为这将导致使用一个来自 vp 的**非临时 unique_ptr 初始化 p1**，而这是不允许的。

前面说过，编译器将发现错误使用 unique_ptr 的企图。 在 unique_ptr 为右值时，可将其赋给 shared_ptr，这与将一个 unique_ptr 赋给一个需要满足的条件相同。与前面一样，在下面的代码中，make_int()的返回类型为 unique_ptr：

```
unique_ptr<int> pup(make_int(rand() % 1000));   // ok
shared_ptr<int> spp(pup);                       // not allowed, pup as lvalue
shared_ptr<int> spr(make_int(rand() % 1000));   // ok
```

模板 **shared_ptr 包含一个显式构造函数，可用于将右值 unique_ptr 转换为 shared_ptr**。shared_ptr 将接管原来归 unique_ptr 所有的对象。 在满足 unique_ptr 要求的条件时，也可使用 auto_ptr，但 unique_ptr 是更

好的选择。如果你的**编译器没有 unique_ptr，可考虑使用 Boost 库提供的 scoped_ptr**，它与 unique_ptr 类似。

C++-Primer-Plus 18 个重点笔记

1. C++的 const 比 C 语言#define 更好的原因？

首先，它能够明确指定类型，有**类型检查功能**。 其次，可以使用 C++的**作用域规则将定义限制**在特定的函数或文件中。 第三，可以将 const 用于更**复杂的类型，比如数组和结构**。

C 语言中也有 const，其与 C++中 const 的区别是：一是作用域规则不同；另一个是，**在 C++中可以用 const 值来声明数组长度**。

2. 不能简单地将整数赋给指针

如下所示：

```
int *ptr;
ptr = 0xB8000000;  // type mismatch 类型匹配错误
```

在这里，左边是指向 int 的指针，因此可以把它赋给地址，但右边是一个整数。您可能知道，0xB8000000 是老式计算机系统中视频内存的组合段偏移地址，但这条语句并没有告诉程序，这个数字就是一个地址。**在 C99 标准发布之前，C 语言允许这样赋值**。但 C++在类型一致方面的要求更严格，编译器将显示一条错误消息，通告类型不匹配。要将数字值作为地址来使用，应通过**强制类型转换将数字转换为适当的地址类型**：

```
int *ptr;
ptr = (int *) 0xB8000000;  // type now match
```

这样，赋值语句的两边都是整数的地址，因此这样赋值有效。 注意，pt 是 int 值的地址并不意味着 pt 本身的类型是 int。例如，**在有些平台中，int 类型是个 2 字节值，而地址是个 4 字节值**。

3. 为什么说前缀++/－比后缀++/－的效率高？

>对于内置类型和当代的编译器而言，这看似不是什么问题。然而，**C++允许您针对类定义这些运算符**，在这种情况下，用户这样定义前缀函数：将值加 1，然后返回结果；但后缀版本首先复制一个副本，将其加 1，然后将复制的副本返回。因此，对于类而言，前缀版本的效率比后缀版本高。 **总之，对于内置类型，采用哪种格式不会有差别，但对于用户定义的类型，如果有用户定义的递增和递减运算符，则前缀格式的效率更高。**

4. 逗号运算符

到目前为止，**逗号运算符**最常见的用途是将两个或**更多的表达式放到一个 for 循环表达式**中。逗号运算符的特性有下面几个：

- 它确保**先计算第一个表达式**，然后**计算第二个表达式**； i = 20, j = 2 * i; // i set to 20, then j set to 40
- **逗号表达式的值是第二部分的值**。例如，上面表达式的值为 40。
- **在所有运算符中，逗号运算符的优先级是最低的**。例如： cats = 17, 240; 被解释我： (cats = 17), 240; 也就是说，将 cats 设置为 17，后面的 240 不起作用。如果是 cats = (17, 240);那么 cats 就是 240 了。

5. 有用的字符函数库

cctype 从 C 语言继承而来，老式格式是 ctype.h，常用的有：

单字节	宽字节	描述
isalnum	iswalnum	是否为字母数字
isalpha	iswalpha	是否为字母
islower	iswlower	受否为小写字母
isupper	iswupper	是否为大写字母
isdigit	iswdigit	是否为数字
isxdigit	iswxdigit	是否为16进制数字
iscntrl	iswcntrl	是否为控制字符
isgraph	iswgraph	是否为图形字符（例如，空格、控制字符都不是）
isspace	iswspace	是否为空格字符（包括制表符、回车符、换行符等）
isblank	iswblank	是否为空白字符（C99/C++11新增）（包括水平制表符）
isprint	iswprint	是否为可打印字符
ispunct	iswpunct	是否为标点
tolower	towlower	转换为小写
toupper	towupper	转换为大写
不适用	iswctype	检查一个 wchar_t 是否是属于指定的分类
不适用	towctrans	使用指定的变换映射来转换一个 wchar_t （实际上是大小写的转换）
不适用	wctype	返回一个宽字符的类别，用于 iswctype 函数
不适用	wctrans	返回一个变换映射，用于 towctrans

6. 快排中中值的选取：

将元素每 5 个一组，分别取中值。在 n/5 个中值里面找到中值，作为 partition 的 pivot。 为什么*不每 3 个一组？保证 pivot 左边右边至少 3n/10 个元素，这样最差 O(n)。

7. C++存储方案

C++三种，C++11 四种 这些方案的区别就在于数据保留在内存中的时间。

自动存储持续性：在函数定义中声明的变量（包括函数参数）的存储持续性为自动的。它们在程序开始执行其所属的函数或代码块时被创建，在执行完函数或代码块时，它们使用的内存被释放。C++有两种存储持续性为自动的变量。

静态存储持续性：在函数定义外定义的变量和使用**关键字 static** 定义的变量的存储持续性都为静态。它们在程序整个运行过程中都存在。C++有 3 种存储持续性为静态的变量。

线程存储持续性（C++11）：当前，多核处理器很常见，这些 CPU 可同时处理多个执行任务。这让程序能够将计算放在可并行处理的不同线程中。**如果变量是使用关键字 thread_local 声明的，则其生命周期与所属的线程一样长**。本书不探讨并行编程。

动态存储持续性：用 new 运算符分配的内存将一直存在，直到使用 delete 运算符将其释放或程序结束为止。这种内存的存储持续性为动态，有时被称为自由存储（free store）或堆（heap）。

8. 自己写 string 类注意事项：

- 关于记录已有**对象数 object_count 不要在类声明（即头文件）中初始化静态成员变量**，这是因为**声明描述了如何分配内存，但并不分配内存**。对于**静态类成员**，可以在**类声明之外使用单独的语句来进行初始化**，这是因为静态类成员是**单独存储的**，而不是对象组成部分。请注意，初始化语句指出了**类型 int**（不可缺少），并使用了**作用域运算符**，但~~没有使用关键字 static~~。 初始化是在方法文件中，而不是在类声明文件中进行的，这是因为类声明位于头文件中，可能被包含多次，这样若在头文件中进行初始化静态成员，将出现多个初始化语句副本，从而引发错误。 对于不能在类声明中初始化静态

成员的一种例外情况是：**静态数据成员为整型或枚举型 const。即如果静态数据成员是整型或枚举型，则可以在类声明中初始化**。

- 注意重写拷贝构造函数和赋值运算符，其中赋值运算符的原型为： **Class_name & Class_name::operator=(const Class_name &);** 它接受并返回一个指向类对象的引用，目的应该是方便**串联使用**。

9. 何时调用拷贝（复制）构造函数：

```
StringBad ditto (motto);
StringBad metoo = motto;
StringBad also = StringBad(motto);
StringBad * pStringBad = new StringBad (motto);
```

以上 4 中方式都将调用：StringBad(const StringBad &) - 其中中间两种声明可能会使用复制构造函数直接创建 metoo 和 also 对象，也可能使用复制构造函数生成一个临时对象，然后将临时对象的内容赋给 metoo 和 also，这取决于具体的实现。最后一种声明使用 motto 初始化一个匿名对象，并将新对象的地址赋给 pStringBad 指针。 - 每当程序生成了对象副本时，编译器都将使用复制构造函数。**具体的说，当函数按值传递对象或函数返回对象时，都将使用复制构造函数**。

记住，**按值传递意味着创建原始变量的一个副本**。 - 编译器生成**临时对象**时，也将使用**复制构造函数**。例如，将 3 个 Vector 对象相加时，编译器可能生成临时的 Vector 对象来保存中间的结果。 - 另外，String sailor = sports;等价于 String sailor = (String)sports;因此调用的是**拷贝构造函数。**

10. 何时调用赋值运算符：

- 将**已有的对象赋给另一个对象时**，将调用重载的赋值运算符。
- **初始化对象时，并不一定**会使用赋值操作符：

```
StringBad metoo=knot;    // use copy constructor, possibly assignment, too
```

这里，metoo 是一个新创建的对象，被初始化为 knot 的值，因此使用赋值构造函数。不过，正如前面指出的，实现时也可能分两步来处理这条语句：使用**复制构造函数创建一个临时对象**，然后通过**赋值操作符将临时对象的值复制到新对象中**。这就是说，**初始化总是会调用复制构造函数**，而使用**=操作符时也可能调用赋值构造函数**。

与复制构造函数相似，**赋值运算符的隐式实现也对成员进行逐个复制**。如果**成员本身就是类对象**，则程序将使用为这个类定义的**赋值运算符**来复制该成员，**但静态数据成员不受影响**。

11. 赋值运算符和拷贝构造函数在实现上的区别：

- 由于目标对象可能引用了以前分配的数据，**所以函数应使用 delete[]来释放**这些数据。
- **函数应当避免将对象赋给自身**；否则给对象重新赋值前，**释放内存操作可能删除对象的内容**。
- **函数返回一个指向调用对象的引用**（方便串联使用），而**拷贝构造函数没有返回值**。

下面的代码说明了如何为 StringBad 类编写赋值操作符：

```
StringBad & StringBad::operator=(const StringBad & st) {
    if(this == & st)
        return * this;
    delete [] str;
    len = st.len;
    str = new char [len + 1];
    strcpy(str,st.str);
    return *this;
}
```

代码首先检查自我复制，这是通过查看赋值操作符右边的地址(&s)是否与接收对象(this)的地址相同来完成的，如果相同，程序将返回*this,然后结束。 如果不同，释放 str 指向的内存，这是因为稍后将把一个新字符串的地址赋给 str。如果不首先使用 delete 操作符，则上述字符串将保留在内存中。由于程序程序不再包含指向字符串的指针，一次这些内存被浪费掉。 接下来的操作与复制构造函数相似，即为新字符串分配足够的内存空间，然后复制字符串。赋值操作并不创建新的对象，因此不需要调整静态数据成员 num_strings 的值。

12. 重载运算符最好声明为友元

比如将**比较函数作为友元，有助于将 String 对象与常规的 C 字符串进行比较**。例如，假设 answer 是 String 对象，则下面的代码： if("love" == answer) 将被转换为： if(operator == ("love", answer)) 然

后，编译器将使用某个构造函数将代码转换为： if(operator == (String("love"), answer)) 这与原型是相匹配的。

13. 在重写 string 类时使用中括号访问字符时注意：

（1）为什么重载的[]返回值是个 char &而不是 char？

（2）为什么有两个重载[]的版本，另一个是 const 版本？

解答（1）： 将返回类制**声明为 char &,便可以给特定元素赋值**。例如，可以编写这样的代码： String means ("might"); means [9] = ' r'; 第二条语句将被转换为一个重载运算符函数调用： means.operator[][0] = 'r'; 这里将 r 赋给方法的返回值，而函数返回的是指向 means.str[0]的引用，因此上述代码等同于下面的代码：means.str[0] = 'r'； 代码的最后一行访问的是私有数据，但由于 operator 是类的一个方法，因此能够修改数组的内容。 最终的结果是"might"被改为"right"。

解答（2）： 假设有下面的常量对象： const String answer("futile"); 如果只有上述 operator 定义，则下面的代码将出错： cout << answer[1]; // compile-time error 原因是 answer 是常量，而上述方法无法确保不修改数据（实际上，有时该方法的工作就是修改数据， 因此无法确保不修改数据)。 **但在重载时，C++将区分常量和非常量函数的特征标**，因此可以提供另一个仅供 const String 对象使用 的 operator 版本： // for use with const String objects const char & string::operator const { return str[i]; } 有了上述定义后，就可以读/写常规 String 对象了 ：**而对于 const Siring 对象，则只能读取其数据**。

14. 静态成员函数

- 在类声明外定义实现时不能再加 static 关键字，与静态成员变量一样。

15. 实现 has-a 关系的两种方法：

- **组合（或包含）方式**。这是我们通常采用的方法。
- c++还有另一种实现 has-a 关系的途径—**私有继承**。使用私有继承，基类的公有成员和保护成员都将称为派生类的私有成员。这意味着基类方法将不会称为派生对象公有接口的一部分，但可以派生类的成员函数中使用它们。而使用公有继承，基类的公有方法将称为派生类的公有方法。简言之，派生类将继承基类的接口：这是 is-a 关系的一部分。使用私有继承，基类的公有方法将称为派生类的私有方法，即派生类不继承基类的接口。正如从被包含对象中看到的，这种不完全继承是 has-a 关系的一部分。 使用私有继承，类将继承实现。例如，如果从 String 类派生出 Student 类，后者将有一个 String 类组件，可用于保存字符串。另外，Student 方法可以使用 String 方法类访问 String 组件。

  包含将对象作为一个命名的成员对象添加到类中，而私有继承将对象作为一个未被命名的继承对象添加到类中。我们使用术语子对象来表示同继承或包含添加的对象。 因此，**私有继承提供的特性与包含相同：获得实现，但不获得接口。所以，私有继承也可以用来实现 has-a 关系**。
- 使用包含还是使用私有继承？ 由于既可以使用包含，也可以使用私有继承来建立 has-a 关系，那么应使用何种方式呢？**大多数 C++程序员倾向于使用包含。**
  - 首先，**它易于理解**。类声明中包含表示被包含类的显式命名对象，代码可以通过**名称引用这些对象**，而使用继承将使关系更抽象。
  - 其次，继承会引起很多问题，尤其从多个基类继承时，可能必须处理很多问题，如包含同名方法的独立的基类或共同祖先的独立基类。 总之，使用包含不太可能遇到这样的麻烦。
  - 另外，**包含能够包括多个同类的子对象**。如果某个类需要 3 个 string 对象，可以使用包含声明 3 个独立的 string 成员。而继承则只能使用一个这样的对象（当对象都没有名称时，将难以区分）。

  然而，**私有继承所提供的特性确实比包含多**。例如，假设**类包含保护成员**（可以是数据成员，也可以是成员函数），**则这样的成员在派生类中是可用的，但在继承层次结构外是不可用的**。如果使用组合将这样的类包含在另一个类中，则后者将不是派生类，而是位于继承层次结构之外，因此不能访问保护成员。但通过继承得到的将是派生类，因此它能够访问保护成员。 另一种需要使用私有继承的情况是需要重新定义虚

函数。派生类可以重新定义虚函数，但包含类不能。使用私有继承，重新定义的函数将只能在类中使用，而不是公有的。

- **通常，应使用包含来建立 has-a 关系；如果新类需要访问原有类的保护成员，或需要重新定义虚函数，则应使用私有继承。**

16. 关于保护继承 保护继承是私有继承的变体，保护继承在列出基类时使用关键字 protected;

```
class Student : protected std::string,
                protected std::valarray<double>
{
...
}
```

使用保护继承时，基类的公有成员和保护成员都将成为派生类的保护成员，和私有继承一样，基类的接口在派生类中也是可用的，但在继承层次结构之外是不可用的。当从派生类派生出另一个类的时，私有继承和保护继承 之间的主要区别便呈现出来了。**使用私有继承时，第三代将不能使用基类的接口**，这是因为基类的共有方法在**派生类中将变成私有方法**；使用**保护继承时**，基类的公有方法在**第二代中将受保护的**，因此**第三代派生类可以使用它们**。

下表总结了公有、私有和保护继承。隐式向上转换意味着无需进行显式类型转换，就可以将基类指针或引用指向派生类对象。

表 14.1 各种继承方式

特征	公有继承	保护继承	私有继承
公有成员变成	派生类的公有成员	派生类的保护成员	派生类的私有成员
保护成员变成	派生类的保护成员	派生类的保护成员	派生类的私有成员
私有成员变成	只能通过基类接口访问	只能通过基类接口访问	只能通过基类接口访问
能否隐式向上转换	是	是（但只能在派生类中）	否

特征	公有继承	保护继承	私有继承
公有继承变成	公有成员	派生类的保护成员	派生类的私有成员
公有继承变成	派生类的保护成员	派生类的保护成员	派生类的私有成员
公有继承变成	只能基类接口访问	只能基类接口访问	只能基类接口访问
能否隐式向上转换	是	是（只在派生类中）	否

17. 智能指针相关 请参考：C++智能指针简单剖析，推荐必看。
18. C++中的容器种类：

- 序列容器（7 个）
  - **vector**：提供了自动内存管理功能（采用了 STL 普遍的内存管理器 allocator），可以动态**改变对象长度**，提供**随机访问**。在**尾部添加和删除元素的时间是常数的**，但在头部或中间就是线性时间。
  - **deque**：双端队列（double-ended queue），支持随机访问，与 vector 类似，主要区别在于，**从 deque 对象的开始位置插入和删除元素的时间也是常数的，所以若多数操作发生在序列的起始和结尾处，则应考虑使用 deque 数据结构。**为实现在 deque 两端执行插入和删除操作的时间为常数时间这一目的，deque 对象的设计比 vector 更为复杂，因此，尽管二者都提供对元素的**随机访问**和在序列**中部执行线性时间的插入和删除操作**，但 **vector 容器更快些**。
  - **list**：双向链表（是循环的）。目的是实现快速插入和删除。
  - **forward_list(C++11)**：实现了单链表，不可反转。相比于 list，forward_list 更简单，更紧凑，但功能也更少。
  - **queue**：是一个适配器类。queue 模板让底层类（默认是 deque）展示典型的队列接口。queue 模板的限制比 deque 更多，它不仅不允许随机访问队列元素，甚至不允许遍历队列。

与队列相同，只能将元素**添加到队尾**、从**队首删除**元素、**查看队首和队尾的值**、检查**元素数目**和**测试队列是否为空**。

- **priority_queue**：是另一个适配器类，支持的操作与 queue 相同。 priority_queue 模板类是另一个适配器类，它支持的操作与 queue 相同。两者之间的主要区别在于，**priority_queue 中，最大元素在队首**。内部区别在于，默认的**底层类是 vector**。可以修改用于确定哪个元素放到队首的**比较方式**，方法是提供一个可选的构造函数参数：

```
priority_queue<int> pq1;                    // default version
priority_queue<int> pg2(greater<int>);      // use greater<int> to order
```

greater<>函数是一个预定义的函数对象。

- **stack**：与 queue 相似，stack 也是一个适配器类，它给底层类（默认情况下为 vector）提供了典型的栈接口。

- 关联容器
  - 4 种**有序**关联容器：set、multiset、map 和 multimap，底层基于**树结构**
  - C++11 又增加了 4 种**无序**关联容器：unordered_set、unordered_multiset、unordered_map 和 unordered_multimap，**底层基于 hash**。

Git 教程

一.git 配置

优先级：--local > --global > --system

用了--global 这个参数，表示你这台机器上所有的 Git 仓库都会使用这个配置

1.配置 git 用户名和邮箱

```
git config --global user.name          # 查看
git config --global user.name 用户名    # 修改
git config --global user.email         # 查看
git config --global user.email 邮箱     # 修改
```

二.仓库

1. 创建 git 仓库

```
git init 仓库名   #创建一个git 仓库
git init         #将一个项目转化为使用git 管理（创建.git 目录）
```

示例：

目录结构：

```
project
   |------.git
          |--------branches
          |--------config          #仓库的配置文件
          |--------description
          |--------HEAD
          |--------hooks
          |--------info
          |--------objects
          |--------refs
```

隐藏目录.git 不算工作区，而是 Git 的版本库

2. 查看仓库状态

```
git status
```

3. 远程仓库

最早，肯定只有一台机器有一个原始版本库，此后，别的机器可以“克隆”这个原始版本库，而且每台机器的版本库其实都是一样的，并没有主次之分

实际情况往往是这样，找一台电脑充当服务器的角色，每天 24 小时开机，其他每个人都从这个“服务器”仓库克隆一份到自己的电脑上，并且各自把各自的提交推送到服务器仓库里，也从服务器仓库中拉取别人的提交
GitHub 就是提供 Git 仓库托管服务的，所以，只要注册一个 GitHub 账号，就可以免费获得 Git 远程仓库，即 Github 为我们的 git 仓库提供了一个远程仓库，有了这个远程仓库，妈妈再也不用担心我的硬盘了

1）为本地与 GitHub 的通信配置 ssh

本地 git 仓库和 GitHub 上的远程仓库之间的传输是通过 SSH 加密的，所以，需要一点设置：

- **创建 ssh key：** `bash  ssh-keygen -t rsa -C "youremail@example.com"`
- **登录你的 GitHub 帐号，**`Settings -> SSH and GPG keys -> new SSH key`**，将 id_rsa.pub 的内容复制进去**

为什么 GitHub 需要 SSH Key 呢？因为 GitHub 需要识别出你推送的提交确实是你推送的，而不是别人冒充的，而 Git 支持 SSH 协议，所以，GitHub 只要知道了你的公钥，就可以确认只有你自己才能推送

2）让本地 git 仓库和远程仓库同步

在有了本地 git 仓库后，还需创建对应的远程仓库

- **在 GitHub 上创建远程仓库**（如果已有则省略）
- **为本地仓库设置远程仓库信息**（如果同时需要为本地仓库添加多个远程仓库（如果 github+码云），则可以将 `origin` 分别换成 `github` 和 `gitee`，推送操作时也要修改 `origin`。添加后，远程库的名字就是 `origin`，这是 Git 默认的叫法，也可以改成别的，但是 `origin` 这个名字一看就知道是远程库）`bash  git remote add origin https://github.com/用户名/仓库名`
  5. **删除本地仓库的远程仓库信息：**`git remote remove origin`
  6. **修改远端地址：**`git remote set-url` 新地址
  7. **查看远程仓库信息：**`git remote -v`
- **将本地 git 仓库 push 到远程仓库** `bash  # 由于远程库是空的，我们第一次推送 master 分支时，加上了-u 参数,Git 不但会把本地的  # master 分支内容推送的远程新的 master 分支，还会把本地的 master 分支和远程的 master  # 分支关联起来，在以后的推送或者拉取时就可以简化命令 git push [-u] origin 分支名`

并不是一定要把本地分支往远程推送。哪些分支需要推送、哪些不需要呢？

3) `master`：主分支，要时刻与远程同步
4) `dev`：开发分支，团队所有成员都需要在上面工作，所有也需要与远程同步
5) `bug`：只用于在本地修复 bug，就没必要推送到远程了，除非老板要看看你每周修复了几个 bug

4. 协同工作

拉取分支：

```
git pull
```

`git clone` 时，默认情况下只能看到本地的 `master` 分支。如果要在 `dev` 分支上开发，就必须创建远程 `origin` 的 `dev` 分支到本地，可以使用如下命令创建本地 `dev` 分支：

```
git checkout -b dev
```

将本地 `dev` 分支与远程 `origin/dev` 分支关联起来：

```
git branch --set-upstream dev origin/dev
```

5. 使用 GitHub

Bootstrap 的官方仓库 twbs/bootstrap、你在 GitHub 上克隆的仓库 my/bootstrap，以及你自己克隆到本地电脑的仓库，他们的关系就像下图显示的那样：

如果你想修复 bootstrap 的一个 bug，或者新增一个功能，立刻就可以开始干活，干完后，往自己的仓库推送

如果你希望 bootstrap 的官方库能接受你的修改，你就可以在 GitHub 上发起一个 pull request。当然，对方是否接受你的 pull request 就不一定了

三.版本控制

隐藏目录`.git` 不算工作区，而是 Git 的版本库。版本库里存了很多东西，其中最重要的就是**称为 stage（或者叫 index）的暂存区**。还有 Git 为我们自动创建的第一个分支 `master`，以及指向 `master` 的一个指针叫 HEAD

1. 添加或删除修改

将修改添加到暂存区：

```
git add 文件/目录
```
从暂存区删除修改：
```
git rm --cached 文件/目录
```
以下命令可以将暂存区的修改重置，暂存区的改变会被移除到工作区：
```
git reset HEAD [文件名]
```
以下命令可以丢弃工作区的修改：
```
git checkout -- [文件名]
```
如果刚对一个文件进行了编辑，可以撤销文件的改变，回到编辑开始。命令其实起到“一键恢复”的作用，还可用于“误删恢复”。可以在 `git reset HEAD [文件名]` 后使用

2. 提交版本

如果修改了 readme.txt，添加了文件 LICENSE，并将 2 者添加到暂存区后，暂存区的状态就变成这样：

使用 commit 提交修改，实际上就是把暂存区的所有内容提交到当前分支：
```
git commit -m '信息'
```

commit 相当于游戏里面一次存档。对应一个版本

3. 文件删除

`rm` 做出的删除不会被暂存，`git rm` 做出的改变会被暂存。如果使用 `rm` 删除掉，能使用 `git rm` 来暂存。`git rm` 不在意文件已经不存在了

6) 删除(暂存)单个文件
```
git rm
```
7) 删除(暂存)多个文件（一般情况下，更可能是对大量文件进行管理。可能同时会删除很多文件，不可能使用 `git rm` 一个个删除）
```
# 它会变量当前目录，将所有删除暂存
git add -u .
```
如果有文件被误删，可以使用 `git checkout --` 文件名恢复

4. 工作现场保存与恢复

有时候在修复 bug 或某项任务还未完成，但是需要紧急处理另外一个问题。此时可以先保存工作现场，当问题处理完成后，再恢复 bug 或任务的进度

8) 保存工作现场：`git stash`
9) 查看保存的工作现场：`git stash list`
10) 恢复工作现场：`git stash apply`
11) 删除 stash 内容：`git stash drop`
12) 恢复工作现场并删除 stash 内容（相当于上面 2 步合并）：`git stash pop`

5. 改动查询

```
git diff [选项]             # 查看工作区中的修改
git diff [选项] --staged    # 查看已添加到暂存区的修改
git diff [选项] HEAD        # 查看当前所有未提交的修改
```

选项：
```
--color-words:  颜色
--stat:         不显示具体修改，只显示修改了的文件
```

6. 版本回退

```
git reset --hard 版本 ID/HEAD 形式的版本
```

```
git reset --hard HEAD       # 当前版本
git reset --hard HEAD^      # 上一个版本
git reset --hard HEAD^^     # 上上个版本
git reset --hard HEAD~n     # 前 n 个版本
```
如果回到过去的版本，想要回到原来新的版本：

13) 如果终端未关，可以找到新版本的 id，通过上述命令回去新版本

14) 如果终端已关，git reflog 查看版本，再通过上述命令回去新版本

7. 查看历史提交

```
git log [选项]
```

选项：

--online：只显示提交提示信息
--stat：添加每次提交包含的文件信息
--path：查看每次提交改变的内容
--graph

加文件名可以显示具体文件相关的所有提交信息

四.分支管理

1. 创建与合并分支

每次 commit 相当于一次存档，对应一个版本。Git 都把它们串成一条时间线，这条时间线就是一个分支。master 就是主分支。HEAD 指向当前分支，而 master 指向主分支的最近提交。每次提交，master 分支都会向前移动一步

当创建一个分支时，如 dev，Git 创建一个指针 dev，指向 master 相同的提交，再把 HEAD 指向 dev，就表示当前分支在 dev 上：

从现在开始，对工作区的修改和提交就是针对 dev 分支了，比如新提交一次后，dev 指针往前移动一步，而 master 指针不变：

假如我们在 dev 上的工作完成了，就可以把 dev 合并到 master 上。最简单的方法，就是直接把 master 指向 dev 的当前提交，就完成了合并：

合并完分支后，甚至可以删除 dev 分支。删除 dev 分支就是把 dev 指针给删掉，删掉后，我们就剩下了一条 master 分支：

上面的合并使用的是 **Fast forward**。这种模式下，删除分支后，会丢掉分支信息。如果要强制禁用 **Fast forward** 模式，Git 就会在 merge 时生成一个新的提交，这样，从分支历史上就可以看出分支信息。通过在 git merge 命令中使用--no-ff 选项禁用 **Fast forward** 模式。比如在合并 dev 时：

```
git merge --no-ff -m "merge with no-ff" dev
```

由于会生成一个新的提交，所以需要使用-m 指明新提交的信息。此时分支情况如下：

相关命令如下：

15) (创建分支并)切换到新分支：git checkout -b 新分支
16) 创建分支：git branch 新分支
17) 切换分支：git checkout 欲切换到的分支
18) 查看当前分支：git branch
19) 合并某分支到当前分支：git merge 欲合并到当前分支的分支
20) 查看历史分支情况：git log --graph --pretty=oneline --abbrev-commit
21) 删除未合并的分支：git branch -D 分支

2. 分支合并冲突

如果两个分支修改了同一文件，合并时会发生冲突。比如 master 分支和 feature1 分支都修改了 readme.txt 文件，各自都有新的提交：

这种情况下，Git 无法执行“快速合并”，只能试图把各自的修改合并起来，但这种合并就可能会有冲突。此时 readme.txt 文件会变成如下形式：

```
Git is a distributed version control system.
Git is free software distributed under the GPL.
Git has a mutable index called stage.
Git tracks changes of files.
<<<<<<< HEAD
Creating a new branch is quick & simple.
=======
```

```
Creating a new branch is quick AND simple.
>>>>>>> feature1
```

Git 用<<<<<<<，=======，>>>>>>>标记出不同分支的内容，此时需要手动修改后保存。然后再使用 git commit 进行一次提交。分支会变成如下：

3. 分支管理策略

在实际开发中，我们应该按照几个基本原则进行分支管理

首先，master 分支应该是非常稳定的，也就是仅用来发布新版本，平时不能在上面干活

干活都在 dev 分支上，也就是说，dev 分支是不稳定的，到某个时候，比如 1.0 版本发布时，再把 dev 分支合并到 master 上，在 master 分支发布 1.0 版本

你和你的小伙伴们每个人都在 dev 分支上干活，每个人都有自己的分支，时不时地往 dev 分支上合并就可以了

所以，团队合作的分支看起来就像这样：

当你从远程仓库克隆时，实际上 Git 自动把本地的 master 分支和远程的 master 分支对应起来了，并且，远程仓库的默认名称是 origin

要查看远程库的信息，用 git remote：

```
$ git remote
origin
```

或者，用 git remote -v 显示更详细的信息：

```
$ git remote -v
origin  git@github.com:michaelliao/learngit.git (fetch)
origin  git@github.com:michaelliao/learngit.git (push)
```

上面显示了可以抓取和推送的 origin 的地址。如果没有推送权限，就看不到 push 的地址

推送分支

git push origin 欲推送的分支

22) master 分支是主分支，因此要时刻与远程同步
23) dev 分支是开发分支，团队所有成员都需要在上面工作，所以也需要与远程同步
24) bug 分支只用于在本地修复 bug，就没必要推到远程了，除非老板要看看你每周到底修复了几个 bug
25) feature 分支是否推到远程，取决于你是否和你的小伙伴合作在上面开发

Linux 常见命令

一.文件管理

1. 文件查找：find

使用方法

find [查找目录] [查找条件]

查找目录：

.：在当前目录及子目录下查找（默认）
A：在目录 A 及 A 的子目录下查找

查找条件：

-name：根据文件名查找
-regex：使用正则表达式匹配
-type：按类型查找（f:文件，d:目录，l:链接...）
-atime：按访问时间查找（n:n 天前的一天内，+n:n 天前(不含 n 天)，-n:n 天内(不含 n 天)）
-mtime：按修改时间查找（n:n 天前的一天内，+n:n 天前(不含 n 天)，-n:n 天内(不含 n 天)）
-size：按大小查找（单位 k，+nk:"比 nk 更大"，-nk:"比 nk 更小"）
-perm：按权限查找（644：权限等于 644 的文件）
-user/-nouser：用户名等于/用户名不等于
-group/-nogroup：组名等于/组名不等于

示例

```
#1.在当前目录及子目录下查找后缀为cpp 的文件
find . -name *.cpp

#2.使用正则表达式查找
find -regex ".*.cpp$"
```

2. 文件拷贝：cp

使用方法

```
cp [选项] 源路径 目的路径
```

选项：

- -a：将所有属性一起复制（包括拥有者、时间等信息）
- -i：目标文件存在时，进行询问
- -r：递归复制

3. 打包解包：tar

使用方法

```
tar [-j|-z] [cv] [-f 压缩包名] 目录
tar [-j|-z] [xv] [-f 解压包名] [-C 解压路径]
```

选项：

- -c/-x：打包/解包
- -j/-z：bzip2 格式/gzip 格式
- -v：显示过程

二.文本处理

1.（显示行号）查看文件：nl

行号计算不包括空行

2. 文本查找：grep

使用方法

```
grep [选项] 模式串 文件
输出 | grep [选项] 模式串
```

选项

- -e：使用多个模式串
- -i：忽略大小写
- -n：打印行号
- -c：统计次数（一行算一次）

示例

```
#1.在 test.c 中搜索包含字符串”printf“或”count“的行
grep -e "printf" -e "count" test.c
```

3. 排序：sort

使用方法

```
sort [选项] 文件
输出 | sort [选项]
```

选项

- -d：按字典序排序（默认）
- -n：按数字排序
- -k："-k n"表示按各行第 n 列进行排序
- -r：反序

4. 转换：tr

使用方法

```
#set1、set2 为字符集，可以是单个字符，也可以是字符串
输出 | tr [选项] set1 set2
```

选项：

```
    -d：删除字符
    -s：字符压缩
```

示例

```
#1.删除字符':'
cat /etc/passwd | tr -d ':'

#2.将小写字母替换成大写字母
last | tr '[a-z]' 'A-Z'

#3.将'a'、'b'、'c'替换成'z'
cat test | tr “abc” 'z'

#4.将连续的'a'压缩成'b'（单个或连续出现的多个‘a’会压缩成一个‘b’）
cat test | tr -s 'a' 'b'
```

5. 切分文本：cut

使用方法

```
cut [选项] 文件
输出 | cut [选项]
```

选项：

```
    -d：分隔符（-d ':' 以’:‘为分隔符）
    -f：选择域（-f 1,2 输出分隔后第 1 列和第 2 列）
    -c：字符范围（-c n-m 输出第 n 到 m 个字符。如果没有 m，输出到末尾）
```

示例

```
#1.按’:‘分隔$PATH，输出第 3 个和第 5 个
echo $PATH | cut -d ':' -f 3,5

#2.输出 export 运行结果每行的第 12-20 个字符
export | cut -c 12-20
```

6. 拼接文本：paste

使用方法

```
paste [选项] file1 file2
```

选项：

```
    -d：指定拼接时使用的分隔符（默认使用 tab 作为分隔符）
```

7. 统计：wc

使用方法

```
wc [选项] 文件
输出 | wc [选项]
```

选项：

```
    -c：统计字符数
    -w：统计单词数
    -l：统计行数
```

8. 数据处理：sed

sed 常用于一整行的处理。如果有一个 100 万行的文件，要在第 100 行加某些文字，此时由于文件太大，不适合用 vim 处理。因此使用 sed 是个很好的选择

使用方法

```
sed [选项] '[动作]' 文件
输入 | sed [选项] '[动作]'
```

选项：

- `-n`：安静模式，只输出 sed 处理过的行（否则未处理行也会输出）
- `-i`：结果直接作用到文件（没指定时不会修改文件）
- `-e`：在命令行模式上输入动作
- `-f`：从文件中读取动作

动作：`[n1[,n2]] function`

function:

- `a/i`：在后插入/在前插入
- `d`：删除
- `p`：打印
- `s`：替换

示例

```
#1.插入
nl /etc/passwd | sed '2a drink tea' #在第2行后插入一行："drink tea"
nl /etc/passwd | sed '2a aaa \
> bbb' #在第2行后插入两行："aaa"和"bbb"

#2.删除
nl /etc/passwd | sed '2,5d' #删除2~5行
sed '/^$/d' ip #将ip文件中的空行删除

#3.打印2~5行（安静模式，不使用安静模式2~5行会打印2次）
nl /etc/passwd | sed -n '2,5p'

#4.替换
nl /etc/passwd | sed '2s/daemon/root/g' #将第二行的daemon替换成root
ifconfig | grep 'inet addr' | sed 's/^.*addr://g' #将所有开头的“inet addr:”删除
```

9. 数据处理：awk

相比于 sed 常用于一整行的处理，awk 则比较倾向于将一行分成数个“字段”来处理。因此，相当适合小型的数据处理

awk 处理步骤：

- 读入第一行，并将第一行的数据填入$0,$1,$2 等变量当中
- 依据条件类型的限制，判断是否需要进行后面的动作
- 做完所有的动作与条件类型
- 若还有后续的“行”的数据，则重复 1~3 步，直到所有的数据都读完为止

使用方法

```
awk '条件类型1{动作1} 条件类型2{动作2} ...' filename
输出 | awk '条件类型1{动作1} 条件类型2{动作2} ...'
```

变量：

- `$0`：整行
- `$1`：按分隔符分隔后的第 1 列
- `$2`：按分隔符分隔后的第 2 列
- `$k`：按分隔符分隔后的第 k 列
- `NF`：每一行拥有的字段数

NR：目前所处理的行数
FS：目前的分隔字符（默认是空格或 tab）
条件判断：>、<、>=、<=、==、!=
命令分隔：使用';'或 Enter
示例

```
#1.打印 last -n 5 结果中每行经过分隔符(默认情况下为空格或 tab)分隔后的第 1 列和第 3 列
last -n 5 | awk '{print $1 "\t" $3}'

#2.以':'作为分隔符，打印第 3 列小于 10 的所有行的第 1 列和第 3 列
cat /etc/passwd | awk '{FS=":"} $3<10{print $1 "\t" $3}'       #（第一行不会处理）
cat /etc/passwd | awk 'BEGIN{FS=":"} $3<10{print $1 "\t" $3}' #（第一行会处理）

#3.假设 test 文件由 3 列数字组成，以空格分隔。该命令会计算每行的和然后打印
awk '{total=$1+$2+$3;printf "%10d %10d %10d %10.2f\n",$1,$2,$3,total}' test
```

注意上面的示例 2，awk 首先是读取一行，分隔后的数据填入$0,$1,$2 等变量中才开始进行条件判断和执行动作。因此第一条命令在按空格或 tab 分隔后才将分隔符换成':'，所以第一行显示结果不对

三.性能分析

1. 进程查询：ps

man ps 手册非常庞大，不是很好查阅，因此主要记住几个命令
示例

```
#1.列出仅与自身环境有关的进程，最上层的父进程是允许该 ps 命令的 bash 而没有扩展到 init 进程中去
ps -l

#2.列出系统所有进程的信息
ps aux
ps -ef    #aux 会截断 COMMAND 列，-ef 不会。aux 是 BSD 风格，-ef 是 System V 风格
ps axjf   #以"进程树"的方式显示所有进程
ps -lA    #输出格式同 ps -l
```

26) **F**：进程标志，说明进程的权限
    1. 4：root 权限
    2. 1：仅能 fork 而不能 exec
    3. 0：既非 4 也非 1
27) **S**：进程的状态
    1. R(running)：正在运行
    2. S(Sleep)：可被唤醒的睡眠
    3. D：不可被唤醒的睡眠（通常可能在等待 I/O）
    4. T：停止，可能是在后台暂停
    5. Z(Zombie)：僵尸进程
28) **C**：CPU 使用率
29) **PRI/NI**：Priority/Nice 的缩写，CPU 优先级(越小越高)
30) **ADDR/SZ/WCHAN**：内存相关，ADDR 指出进程在内存的哪个部分，running 进程一般显示'-'。SZ 为进程使用的内存。WCHAN 表示进程当前是否运行中'-'，当进程睡眠时，指出进程等待的事件
31) **TTY**：进程运行的终端机
32) **TIME**：进程用掉的 CPU 时间

33) **USER**：进程所属用户
34) **%CPU/%MEM**：进程消耗的 CPU 百分比和内存百分比
35) **VSZ**：进程用掉的虚拟内存(KB)
36) **RSS**：进程占用的固定内存(KB)

37) **TTY**：进程运行的终端机，与终端机无关则显示'?'。tty1~tty6 是本机的登陆者程序，pts/0 等表示由网络连接进主机的进程
38) **STAT**：进程目前的状态，与 ps -l 结果中的 **S** 等同
39) **START**：进程启动的时间
40) **TIME**：进程实际使用的 CPU 运行时间

2. 进程监控：top

使用方法

```
top [选项]
```

选项：

-d：跟秒数指定更新间隔
-n：与-b 搭配，指定需要进行几次 top 输出，重定向时常用
-p：指定 PID，监控特定进程

top 模式下的命令：

41) **?**：显示可用的命令
42) **P**：以 CPU 使用情况排序
43) **M**：以内存使用情况排序
44) **N**：以 PID 排序
45) **q**：退出
46) **1**：多核情况下切换 CPU

%Cpu(s)后面的"wa"表示 I/O wait，过高说明长时间等待 I/O，I/O 存在瓶颈

3. 打开文件查询：lsof

使用方法

```
lsof [选项]
```

选项：

-i：-i:端口号查看端口被占用的情况
-u：后跟用户名查看具体用户打开的文件
-p：后跟 PID 查看指定进程打开的文件
+d：后跟目录查看指定目录下被进程打开的文件，"+D"递归

4. 内存使用量：free

使用方法

```
free [选项]
```

选项：

-b|-k|-m|-g：单位
-t：列出物理内存与 swap 的汇总情况

47) **buffers**：主要缓存 dentry 和 inode 等元数据
48) **cached**：主要缓存文件内容，即 page cache
49) **- buffers/cache**：实际使用的内存。used-buffers-cached
50) **+ buffers/cache**：可用内存。free+buffers+cached（在内存紧张时，buffers 和 cached 可以回收）

详细结果说明

5. shell 进程的资源限制：ulimit

使用方法

```
ulimit [选项]          #查看
ulimit [选项] 新值     #修改
```

选项：

-a：列出 shell 进程的所有资源限制情况（-a 命令会列出查看某一资源限制的选项参数）
...

使用 ulimit 修改资源限制只会对当前终端环境有效，如果想永久生效，可以修改文件 /etc/security/limits.conf，该文件的内容如下；

```
# /etc/security/limits.conf
#
#Each line describes a limit for a user in the form:
#
#<domain>        <type>  <item>  <value>
#
#Where:
#<domain> can be:
#        - a user name
#        - a group name, with @group syntax
#        - the wildcard *, for default entry
#        - the wildcard %, can be also used with %group syntax,
#                 for maxlogin limit
#        - NOTE: group and wildcard limits are not applied to root.
#          To apply a limit to the root user, <domain> must be
#          the literal username root.
#
#<type> can have the two values:
#        - "soft" for enforcing the soft limits
#        - "hard" for enforcing hard limits
#
#<item> can be one of the following:
#        - core - limits the core file size (KB)
#        - data - max data size (KB)
#        - fsize - maximum filesize (KB)
#        - memlock - max locked-in-memory address space (KB)
#        - nofile - max number of open files
#        - rss - max resident set size (KB)
#        - stack - max stack size (KB)
#        - cpu - max CPU time (MIN)
#        - nproc - max number of processes
#        - as - address space limit (KB)
#        - maxlogins - max number of logins for this user
#        - maxsyslogins - max number of logins on the system
#        - priority - the priority to run user process with
#        - locks - max number of file locks the user can hold
#        - sigpending - max number of pending signals
#        - msgqueue - max memory used by POSIX message queues (bytes)
#        - nice - max nice priority allowed to raise to values: [-20, 19]
#        - rtprio - max realtime priority
#        - chroot - change root to directory (Debian-specific)
#
#<domain>      <type>  <item>         <value>
#

#*               soft    core            0
#root            hard    core            100000
#*               hard    rss             10000
#@student        hard    nproc           20
#@faculty        soft    nproc           20
#@faculty        hard    nproc           50
#ftp             hard    nproc           0
#ftp             -       chroot          /ftp
#@student        -       maxlogins       4
```

```
# End of file
```

示例

```
root@068ca8da6d06:/# ulimit -a
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
scheduling priority             (-e) 0
file size               (blocks, -f) unlimited
pending signals                 (-i) 7863
max locked memory       (kbytes, -l) 82000
max memory size         (kbytes, -m) unlimited
open files                      (-n) 1048576
pipe size            (512 bytes, -p) 8
POSIX message queues     (bytes, -q) 819200
real-time priority              (-r) 0
stack size              (kbytes, -s) 8192
cpu time               (seconds, -t) unlimited
max user processes              (-u) unlimited
virtual memory          (kbytes, -v) unlimited
file locks                      (-x) unlimited
```

四.网络工具

1. 网卡配置：ifconfig

2. 查看当前网络连接：netstat

`netstat [选项]`

选项：

- `-a`：将所有的连接、监听、Socket 数据都列出来（如，默认情况下，不会列出监听状态的连接）
- `-t`：列出 tcp 连接
- `-u`：列出 udp 连接
- `-n`：将连接的进程服务名称以端口号显示（如下图中 Local Address 会换成 10.0.2.15:22）
- `-l`：列出处于监听状态的连接
- `-p`：添加一列，显示网络服务进程的 PID（需要 root 权限）
- `-i`：显示网络接口列表，可以配合 ifconfig 一起分析
- `-s`：打印网络统计数据，包括某个协议下的收发包数量

51) **Active Internet connections（w/o servers）：网络相关的连接**

1. **Recv-Q**：接收队列(已接收还未递交给应用)
2. **Send-Q**：发送队列(接收方未确认的数据)
3. **Local Address**：本地 IP(主机):端口(服务名)
4. **Foreign Address**：远端 IP:端口
5. Recv-Q 和 Send-Q 通常应该为 0，如果长时间不为 0 可能存在问题

52) **Active UNIX domain sockets（w/o servers）：本地相关的套接字**

1. **RefCnt**：连接到此 socket 的进程数
2. **Flags**：连接标识
3. **Type**：socket 访问的类型
4. **Path**：连接到此 socket 的相关程序的路径

netstat 的 10 个基本用法

3. 查看路由表：route
4. 检查网络连通性：ping
5. 转发路径：traceroute
6. 网络 Debug 分析：nc
7. 命令行抓包：tcpdump
使用方法
```
sudo tcpdump [选项] ...
```

选项：
- -D/-i：查看/指定网卡

示例
```
#抓取本地 9877 号端口的 TCP 数据包
sudo tcpdump -i lo tcp port 9877
```
下图为 tcp 回射服务器，客户端分别键入“hello”和“world”时，使用 tcpdump 抓取到的数据包

8. 域名解析工具：dig
9. 网络请求：curl
五.开发及调试
1. 编辑器：vim
2. 编译器：gcc 和 g++
C 程序的编译过程
使用方法
```
gcc/g++ [选项] 源文件
```

选项：
- -E：让编译器在预处理之后停止，不进行后续编译过程，得到.i 文件
- -S：让编译器在编译之后停止 ，不进行后续过程，得到.s 文件
- -c：生成机器码即二进制.o 文件
- -o：指定目标文件名
- -g：在编译的时候生成调试信息
- -Wall：生成所有警告信息
- -I 目录：指定头文件的查找目录
- 生成动态链接库：
  1. gcc/g++ -c -fPIC 源文件 -o 目标文件名
  2. gcc -shared 目标文件名 -o 动态链接库名.so
- 生成静态链接库：
  1. gcc/g++ -c 源文件 -o 目标文件名
  2. ar -crv 静态链接库名.a 目标文件名
- -l 库名 -L 目录：引入链接库，-L 指定查找该库的目录。如-lm 表示引入 libm.so

3. 调试工具：gdb
使用方法
```
#第一步：得到可执行文件
gcc/g++ -o 可执行文件 -g 源文件

#第二步：启动 gdb
gdb #启动 gdb

#第三步：执行 gdb 命令进行调试
(gdb) gdb 命令
```

gdb 命令：

- file 可执行文件：导入需要调试的文件
- r：运行程序
- q：退出 gdb
- b：设置断点
  - b 行号
  - b 函数名称
  - b *函数名
  - b *代码地址
  - b 编号
- c：继续执行，直到下一断点或程序结束
- s：执行一行代码，如果此行代码有函数调用则进入函数
- n：执行一行代码，如果此行代码有函数调用，不进入函数，直接执行函数
- i(info) 子命令：查看某些信息（只输入 info 或 i 可以查看有哪些子命令）
  - info thread：查看进程的所有线程，会显示每个线程的序号（1~n）
- thread 线程序号：切换到相应的线程（线程序号可以由 info thread 得到）
- f(frame) 函数栈帧号：切换到相应的函数栈帧（函数栈帧号可以由 where 等命令得到）
- list：查看源码
  - list 行号：查看指定行号附近的源码
  - list 函数：查看指定函数附近的源码
  - list 文件:行号：查看指定文件中指定行附近的代码
- where：查看当前位置
- p(print) /格式 表达式
  - 格式：
    - x：按十六进制格式显示变量
    - d：按十进制格式显示变量
    - u：按十六进制格式显示无符号整形
    - o：按八进制格式显示变量
    - t：按二进制格式显示变量
    - a：按十六进制格式显示变量
    - c：按字符格式显示变量
    - f：按浮点数格式显示变量
  - 表达式中可用的操作符：
    - @：一个和数组有关的操作符，左边是起始地址，右边是长度（p *arr@3）
    - ::：指定一个在文件或是函数中的变量（p 'f2.c'::x）
    - {<type>}<addr>：一个指向内存<addr>的类型为 type 的一个对象
- x(examine) <n/f/u> <addr>：查看内存
  - n：正整数，表示需要显示的内存单元个数
  - f：显示的格式（格式字母同上面的 print）
  - u：每个单元的字节数
    - b：1 字节
    - h：2 字节
    - w：4 字节（默认）
    - g：8 字节

4. 查看依赖库：ldd
5. 二进制文件分析：objdump
6. ELF 文件格式分析：readelf
7. 跟踪进程中系统调用：strace
8. 跟踪进程栈：pstack
9. 进程内存映射：pmap

六.其他

1. 终止进程：kill
2. 修改文件权限：chmod

53) w 权限不具有删除文件的能力
54) 目录的 x 权限表示能否进入目录

使用方法

```
chmod [选项] [u|g|o|a][+|-][r|w|x] 文件或目录
chmod [选项] 权限的数字表示 文件或目录
```

选项：

-R：递归式的修改

3. 创建链接：ln
4. 显示文件尾：tail
5. 版本控制：git
6. 设置别名：alias

Linux 内核

1. I/O 设备

设备类型大体上可以分为**块设备(block device)**和**字符设备(character device)**
块设备和字符设备关键的区别在于**数据访问的方式**：

1. **块设备**：数据访问的方式为**随机访问**，可以在块设备的不同位置进行跳转，随机访问数据，并不需要遵循一定的顺序。（常见块设备包括：hard disk(最普遍)、floppy drivers、Blu-ray reader、flash memory）
2. **字符设备**：数据访问的方式为**数据流**，拿键盘来说，如果键入“wolf”，驱动会严格按照字符串顺序返回字符流，如果乱序读取这个字符流，或者读取字符流中其它位置的字符，都会产生歧义。（常见字符设备包括：serial ports、keyboards）

因此，管理块设备通常会更复杂。因为字符设备只需要记录当前读取数据的位置就行了，而块设备在任何位置支持向前或者向后访问。加上块设备对性能十分敏感，因此内核为块设备单独提供了一个子系统**(块 I/O 层 block I/O layer)**进行管理

2. 扇区(sector)

块设备的最小可寻址单元。最常见大小是 **512B**(也有其它大小，如很多 CD-ROM discs 的扇区为 2KB)。尽管很多块设备能一次对多个扇区进行管理，但是无法对比块更小的单元进行寻找/操作

3. 块(block)

块是**文件系统层面的一个抽象**。尽管块设备以扇区为单元进行寻址，但是文件系统操作的数据以块为单位。内核要求块**不能小于扇区，不能大于页(page)**。一般是 2^k 个扇区大小，常见大小为 512B，1KB，4KB
扇区和块的关系下图：

4. buffer_head 结构

如果一个块被存进内存，那么就是说这个块存入了一个 buffer。**每一个 buffer 和一个具体的块对应**。可以说，一个 buffer 是一个块的内存表示
由于块不能比 page 大，因此在内存中，一个 page 能容纳一个或多个块。内核需要一些**控制信息(buffer 对应哪个设备的哪个块)来管理 buffer**，因此设计了名为 buffer_head 的描述符。定义在 <linux/buffer_head.h>中：

```
struct buffer_head {
    unsigned long b_state;          /* buffer 的状态位图 */
    struct buffer_head *b_this_page;    /* 该页 buffer 的循环链表 */
    struct page *b_page;            /* 该 buffer_head 映射到的 page */

    sector_t b_blocknr;         /* 起始块号 */
    size_t b_size;              /* size of mapping */
    char *b_data;               /* pointer to data within the page */

    struct block_device *b_bdev;
    bh_end_io_t *b_end_io;          /* I/O completion */
    void *b_private;            /* reserved for b_end_io */
    struct list_head b_assoc_buffers;   /* associated with another mapping */
    struct address_space *b_assoc_map;  /* mapping this buffer is
                        associated with */
    atomic_t b_count;           /* 引用计数 */
};
```

3. b_state：buffer 的状态。可能是下表中的值
4. b_count：buffer 的使用计数。这个值通过两个内联函数进行增加和减小，它们定义在 <linux/buffer_head.h>中 c static inline void get_bh(struct buffer_head *bh)

{ atomic_inc(&bh->b_count); } static inline void put_bh(struct buffer_head *bh) { atomic_dec(&bh->b_count); } 在操作一个 buffer 前，需要调用 get_bh()增加这个 buffer 的使用计数，确保 buffer 不会意外释放。当操作完成后，调用 put_bh()函数减小使用计数

5. b_bdev：buffer 对应的块设备
6. b_blocknr：buffer 对应块的逻辑块号
7. b_page：指向 buffer 对应的 page
8. b_data：直接指向具体的块（buffer 对应的块位于内存 b_data 到 b_data+b_size 范围内）
9. b_size：块的大小

标志值	描述
BH_Uptodate	Buffer contains valid data
BH_Dirty	Buffer is dirty
BH_Lock	Buffer is undergoing disk I/O and is locked to prevent concurrent access
BH_Req	Buffer is involved in an I/O request
BH_Mapped	Buffer is a valid buffer mapped to an on-disk block
BH_New	Buffer is newly mapped via get_block() and not yet accessed
BH_Async_Read	Buffer is undergoing asynchronous read I/O via end_buffer_async_read()
BH_Async_Write	Buffer is undergoing asynchronous write I/O via end_buffer_async_write()
BH_Delay	Buffer does not yet have an associated on-disk block (delayed allocation)
BH_Boundary	Buffer forms the boundary of contiguous blocks—the next block is discontinuous
BH_Write_EIO	Buffer incurred an I/O error on write
BH_Ordered	Ordered write
BH_Eopnotsupp	Buffer incurred a “not supported” error
BH_Unwritten	Space for the buffer has been allocated on disk but the actual data has not yet been written out
BH_Quiet	Suppress errors for this buffer

总的来说，buffer_head 包含内核操作 buffer 需要用到控制信息，它描绘了内存中 buffer 和磁盘块之间的映射关系

5. bio 结构

bio 结构是内核中块 I/O 的基本容器，定义在<linux/bio.h>中。它**代表一个活动的块 I/O 操作**这个块 I/O 操作的对象是以片段(segement)组织的链表。**片段(segement)是一个 buffer 在内存中连续的一块数据，用 bio_vec 结构表示**，因此，单个 buffer 不一定要在内存中连续（意思就是，可以由分散在内存中的多个连续的数据区——片段，组成一个 buffer）。即使单个 buffer 的数据可能分散在内存中的多个位置，bio 结构也提供了对这个 buffer 操作的能力(scatter-gather I/O)

```
struct bio {
    sector_t        bi_sector;  /* device address in 512 byte
                                   sectors */
    struct bio      *bi_next;   /* request queue link */
    struct block_device *bi_bdev;
    unsigned long       bi_flags;   /* status, command, etc */
    unsigned long       bi_rw;      /* bottom bits READ/WRITE,
                                     * top bits priority
                                     */

    unsigned short      bi_vcnt;    /* how many bio_vec's */
    unsigned short      bi_idx;     /* current index into bvl_vec */

    /* Number of segments in this BIO after
     * physical address coalescing is performed.
     */
    unsigned short      bi_phys_segments;

    /* Number of segments after physical and DMA remapping
```

```
     * hardware coalescing is performed.
     */
    unsigned short      bi_hw_segments;

    unsigned int        bi_size;    /* residual I/O count */

    /*
     * To keep track of the max hw size, we account for the
     * sizes of the first and last virtually mergeable segments
     * in this bio
     */
    unsigned int        bi_hw_front_size;
    unsigned int        bi_hw_back_size;

    unsigned int        bi_max_vecs;    /* max bvl_vecs we can hold */

    struct bio_vec      *bi_io_vec; /* the actual vec list */

    bio_end_io_t        *bi_end_io;
    atomic_t        bi_cnt;     /* pin count */

    void            *bi_private;

    bio_destructor_t    *bi_destructor; /* destructor */
};
```

10. bi_io_vec：指向片段组织的链表。每一个 bio_vec（）表示一个片段，被看作是一个<page,offset,len>形式的矢量(vector)。bio_vec 结构定义在<linux/bio.h>中： c struct bio_vec { struct page *bv_page; unsigned int bv_len; unsigned int bv_offset; };
11. bi_vcnt：链表的大小
12. bi_cnt：这个 bio 的使用计数。当 bi_cnt=0 时，这个 bio 结构会被销毁，释放。在对 bio 进行操作前，需要调用 bio_get 增加使用计数，操作完成后，需要调用 bio_put 减小使用计数。下面两个功能用来更新 bi_cnt 的值： c void bio_get(struct bio *bio) //增加使用计数 void bio_put(struct bio *bio) //减小使用计数
13. bi_private 域记录这个 bio 拥有者的私有信息，拥有者为分配这个 bio 结构的人

下图描绘了 bio、bio_vec、page 三者间的关系：

总之，每个块 I/O 请求用一个 bio 结构表示，每个请求由一个或多个块组成(对应一个或多个 buffer)，这些块通过 bi_io_vec 指向的 bio_vec 链表串连起来。随着块 I/O 层提交片段，bi_idx 域被更新指向当前段。

6. I/O 请求队列与 I/O 请求

块设备维护一个请求队列来存放待处理的块 I/O 请求。这个**请求队列由 request_queue 结构表示**，定义在<linux/blkdev.h>中。请求队列包含了：

14. 一个由请求组成的双链表
15. 还有一些相关的控制信息

请求在内核上层(如文件系统层)被加入到队列。只要请求队列非空，块设备驱动就从队列头部提取 I/O 请求，然后提交到对应的块设备

每个 I/O 请求用 request 结构表示。定义在<linux/blkdev.h>中。**每个 I/O 请求可以由不止一个 bio 结构组成，因为单个请求可以操作多个磁盘上连续的块**

7. 调度算法

调度算法就是控制 I/O 请求队列中 I/O 请求的合并方式与调用时机

如果只是简单的按 I/O 请求插入请求队列的顺序将 I/O 请求提交到对应的块设备，会导致非常差的性能。磁盘寻道是当今计算机最慢的操作之一。因此最小化寻道时间对于提升系统性能来说至关重要

为了最小化寻道时间，内核并不按 I/O 请求插入队列的顺序提交 I/O 请求。它们会对 I/O 请求进行**合并**和**排序**来提高系统的总体性能。**内核实现这两个操作的子系统就是调度算法**
通过合并和排序，I/O 调度器在待处理的 I/O 请求之间划分磁盘 I/O 资源

- **合并**：就是将多个请求合并成一个请求，举例来说，如果一个读取文件中一大块数据的请求被插入到请求队列，如果此时请求队列中已经存在一个读取磁盘中相邻扇区的请求。那么则两个请求能被合并成一个。这样的合并使得只需要提交一个命令到块设备就能完成多个请求的 I/O 操作，极大节省了寻道时间
- **排序**：即使没有找到合适的请求进行合并，也不会简单的将新请求插入到队尾。而是按访问扇区的位置插入新请求。这样可以在磁头从一端移动到另一端的过程中，处理这条路径上的所以请求(就像电梯)。这样设计的目的是为了最小化总寻道时间，并不是最小化每一个请求的寻道时间(想一下，对于电梯最高层的人来说，如果电梯上升过程中有另外的人要上电梯——即有新请求要插队，那么肯定会延长电梯到达最高层的时间)

1）Elevator（电梯）

内核 2.4 的默认调度算法；内核 2.6 中，被换成了 deadline 调度算法；但是由于这个算法更简单，并且在功能上有很多相似的地方，因此是个很好的入门
Linux Elevator 调度算法包括合并和排序操作
当有请求入队时：

- 检索是否有可以合并的候选者。包括**向前合并(front merging)**和**向后合并(back merging)**
    1. **如果新请求被合并到相邻请求前面，则是向前合并**（由于文件的布局方式(按扇区号增加的方向布局)以及一种典型负载的 I/O 操作特点(通常读文件从起始读到结尾，而不是反过来读)，向前合并相比于向后合并来说极少发生。但是尽管如此，Linux Elevator 还是会对两种合并进行检查）
    2. **如果新请求被合并到邻近请求的后面，则是向后合并**
- 如果新请求没有合并，则在队列中寻找一个合适的位置插入新请求，未找到则将新请求插入队尾
- 此外，如果在队列中存在驻留时间过长的请求(超过一个预定的阈值)，那么即使有适合新请求插入的位置，新请求也会被插入队尾。这是为了防止磁盘上邻近的大量请求饥饿其它位置的请求。但是这种时间检测并没有为驻留时间过长的请求提供服务，而是停止新请求的有序插入。虽然改善了延迟，但是仍然有可能导致饥饿。这是内核 2.4 的 I/O 调度算法必须修改的一点

总的来说，当一个被添加到队列时，可能会执行下列 4 个操作：

16. 如果有合适的请求合并，则执行请求合并操作；
17. 如果队列中存在驻留时间过长的请求，则将新请求插入队尾；
18. 如果在队列中找到合适的插入位置，则将新请求插入该位置(使得所有请求按磁盘中的物理位置排序)；
19. 如果没找到合适的位置插入，则将新请求插入队尾；

2）Deadline（截止日期）

这个调度算法尝试解决 Linus Elevator 中的饥饿问题。为了最小化寻道时间，相同位置的请求会插入请求队列，饥饿其它位置的请求
更坏的是，这个饥饿问题带来了一个特例：**写饥饿读**。写操作经常在内核空闲时被提交到磁盘，它和提交它(写操作)的程序完全异步执行。而读操作完全不同。一般情况下，当一个应用提交一个读请求后，它会被阻塞，直到读请求被满足。也就是说，读请求和提交它的程序同步执行。虽然写延迟和应用性能没有多大关系，但是对于读操作来说，应用必须等待读操作完成。因此，读延迟对系统性能来说十分重要
除此之外，读请求往往互相依赖。考虑读取一个大文件，如果前面一块数据没有读完，后面数据块的读取也无法执行。更糟的是，读和写操作都要求读取元素据(如 inode)。这使得 I/O 更加串行化。因此，如果读请求饥饿，整个串行化累积起来的延迟将会异常巨大
需要注意的是，**减少请求饥饿必须以全局吞吐量为代价**。deadline 调度算法非常努力地尝试在限制饥饿发生的同时，提供良好的全局吞吐量。但是不要搞错：要保证请求公平性的同时，最大化全局吞吐量仍然非常困难
在 deadline 调度算法中，每个请求有一个到期时间(expiration time)，默认为 500ms(对于读请求)和 5s(对于写请求)

20. 和 linux elevator 类似，它有一个名为 **sorted queue** 的有序队列。这个队列按请求数据的物理磁盘位置对请求排序。当请求被提交到 sorted queue 时，deadline 按照 linus elevator 的方式执行合并和插入操作
21. 此外，它还根据请求的类型将读请求和写请求分别插入到 **read FIFO queue** 和 **write FIFO queue**。虽然 sorted queue 是按照物理磁盘访问顺序对请求进行排序，但是 **read FIFO queue** 和 **write FIFO queue** 严格保持先进先出的顺序。正常情况下，deadline 从 sorted queue 头部取出一个请求，提交到分发队列(dispatch queue)，分发队列进一步将请求提交给磁盘。这保证了最小化寻道时间
22. 如果 **read FIFO queue** 或者 **write FIFO queue** 中的头部请求到期(也就是说，当前时间超过了请求的到期时间)，则 deadline 转为服务 FIFO 队列。这确保了不会有请求超过到期时间太多才完成

deadline 并不保证请求的**完成**时间，它只保证在到期时间来临之前或者在到期时间来临时**提交**请求。这能够防止饥饿发生。同时，由于读请求的到期时间远小于写请求的到期时间，它也能防止写饥饿读。对读请求的照顾确保了最小化的读延迟

3）Anticipatory(预测)

deadline 调度算法还是在全局吞吐量上做出了牺牲，考虑这种负载：一个应用在执行大量顺序写操作。在没有请求到期之前，按顺序满足这些顺序写请求。假设现在每隔一段时间有一个读请求到达，在读请求到期时间之前，会继续执行写请求。当读请求到期时，转而调度读请求。这里引入了磁盘寻道时间。在处理完读请求后，又回来出来顺序写，这里又引入了寻道开销。然后一段时间后下一个读请求到达，如此反复，每个读请求到来时，都会引入一定的寻道时间。导致全局吞吐量下降。因此，anticipatory 调度算法试图保持读请求低延迟的同时，优化全局吞吐量

首先，anticipatory 调度算法按照 deadline 的方式进行调度，**它也有 3 个队列，每个请求都有一个到期时间。主要改变是它引入了一个启发式预测**

继续用之前的例子说明。当一个读请求到达时，在到达日期来临前，会按照之前的方式进行处理。当这个读请求被提交后，anticipatory 不会马上转回去处理顺序写请求，而是会等待一小段时间(可以设置，默认是 6ms)。如果在这段时间内还有其它读请求到来，就可以较少来回寻道次数，提高吞吐量

当然，如果在等待时间内没有任何事发生，那么这段等待时间就浪费了。anticipatory 调度算法的性能取决于能否准确预测等待时间内系统的 I/O 活动。这通过一系列统计以及启发式来完成。anticipatory 跟踪并统计每个进程的块 I/O 操作行为

这个调度算法在大多数负载下都表现良好。对于服务器来说是个比较理想的算法，但是在一些不常见但是负载严格的场景中(包括 seek-happy databases)表现不好

4）CFQ(完全公平队列)

这个调度算法是为特定负载设计的，但是实际上在很多负载下它都能提供不错的性能。它和之前提到的算法有本质上的区别

cfq 为每个进程维护一个 I/O 队列，新的 I/O 请求会插入到发起这个请求的进程的 I/O 队列中。在每个队列中，请求进行合并以及排序操作

cfq 以轮询的方式服务每一个队列，每次从一个队列中选择一定数目的(默认 4，可以设置)请求服务。因此提供了进程之间的公平性。确保每个进程得到公平的带宽。这个算法主要是为多媒体负载设计，实际上能在很多场景下工作良好。它被推荐用于桌面负载

5）Noop

noop 调度算法仅仅只对新请求执行合并操作，除此之外它什么也不做。仅仅只是以 near-FIFO 的顺序维护一个请求队列

这种算法这样设计并不是没有原因的，它是考虑到如 ssd 这类在寻道上只有很小开销或者没有开销的设备，对于这些设备来说，不需要担心寻道延迟，因此也就不需要对请求进行排序。因此对于这类设备来说，noop 是个理想的候选者

6）查看与选择可用的调度算法

可用通过如下命令查看 linux 中支持和当前使用的调度算法：

```
cat /sys/block/设备名/queue/scheduler

noop [deadline] cfq #当前使用的调度算法为"deadline"
```

如果要切换调度算法，可用使用命令：

```
echo 算法名 >> /sys/block/设备名/queue/scheduler
```

8. Linux 通过什么方式实现系统调用

通过**软件中断**实现

首先，用户程序为系统调用设置参数，其中的一个参数是系统调用编号。参数设置完成后，程序执行“系统调用”指令。x86 系统上的软中断是由 int 产生。这个指令会导致一个异常：产生一个事件，这个事件会导致处理器切换到内核态并跳转到一个新的地址，并开始执行那里的异常处理程序，此时的异常处理程序就是系统调用程序

堆是一棵完全二叉树，使用数组实现堆，堆分为两种：

1. 最大堆：父节点大于任意子节点（因此堆顶为最大值）
2. 最小堆：父节点小于任意子节点（因此堆顶为最小值）

对于第 i 个节点（i 从 0 开始计数）：

1. 父节点：(i-1)/2
2. 左子节点：2i+1
3. 右子节点：2i+2

若包含 sz 个节点，则第一个非叶子节点的序号为(sz - 2) / 2

插入节点

插入节点时，进行下列操作：

- 将元素添加到数组末尾；（相当于叶节点接入堆中）
- 和父节点进行比较，如果大于父节点(以最大堆为例)，则与父节点交换，一直比较交换到根节点

```
/*****************************************
 * 向堆中插入元素
 *  hole：新元素所在的位置
 *****************************************/
template <class value>
void _push_heap(vector<value> &arr,int hole){
    value v = arr[hole]; //取出新元素，从而产生一个空洞
    int parent = (hole - 1) / 2;
    //建最大堆，如果建最小堆换成 arr[parent] > value
    while(hole > 0 && arr[parent] < v){
        arr[hole] = arr[parent];
        hole = parent;
        parent = (hole - 1) / 2;
    }
    arr[hole] = v;
}
```

删除堆顶

删除实际上是将堆顶元素移入数组末尾，并不是真的删除。删除节点时，进行下列操作：

- 保存数组末尾元素(存如临时变量 v)，将堆顶元素存入数组末尾
- 将原来堆顶元素的两个子节点中较大的一个移入堆顶(以最大堆为例)，填补空缺，此时产生新的空缺，继续此步骤，直到空缺为一个叶子节点
- 将 v 中存储的值移到空缺叶子节点的位置
- 对上一步中的新叶子节点完成向上比较交换操作

```
/*****************************************
 * 删除堆顶元素
 *****************************************/
template <class value>
void _pop_heap(vector<value> &arr,int sz)
```

```
{
    value v = arr[sz - 1];
    arr[sz - 1] = arr[0];
    --sz;
    int hole = 0;
    int child = 2 * (hole + 1); //右孩子
    while(child < sz){
        if(arr[child] < arr[child - 1])
            --child;
        arr[hole] = arr[child];
        hole = child;
        child = 2 * (hole + 1);
    }
    if(child == sz){
        arr[hole] = arr[child - 1];
        hole = child - 1;
    }
    arr[hole] = v;
    _push_heap(arr,hole);
}
```

建堆

1. **堆的大小固定(且所有元素已知)**：按“序号从大到小”的顺序遍历所有非叶子节点，将这些节点与左右子节点较大者(以最大堆为例)交换，执行 siftdown 一直到叶子节点，因此，每遍历到一个节点时，其左子树和右子树都已经是最大堆，只需对当前节点执行 siftdown 操作
2. **堆的大小未知(如数据流)**：可以通过插入操作来构建堆

```
/*********************************************
 * 建堆
 *  sz: 删除堆顶元素后的大小
 *  v:  被堆顶元素占据的位置原来的元素的值
 *********************************************/
template <class value>
void _make_heap(vector<value> &arr)
{
    int sz = arr.size();
    int parent = (sz - 2) / 2;
    while(parent >= 0){
        int hole = parent;
        int child = 2 * (hole + 1); //右孩子
        value v = arr[hole];
        while(child < sz){
            if(arr[child] < arr[child - 1])
                --child;
            arr[hole] = arr[child];
            hole = child;
            child = 2 * (hole + 1);
        }
        if(child == sz){
            arr[hole] = arr[child - 1];
            hole = child - 1;
        }
        arr[hole] = v;
        _push_heap(arr,hole);
        --parent;
    }
}
```

复杂度

3. **插入节点**：时间复杂度为 O(logn)
4. **删除堆顶**：时间复杂度为 O(logn)
5. **建堆**：
    1. **堆的大小固定(且所有元素已知)**：每个 siftdown 操作的最大代价是节点被向下移动到树底的层数。在任意一棵完全二叉树中，大约有一半的节点是叶节点，因此不需要向下移动。四分之一的节点在叶节点的上一层，这样的节点最多只需要移动一层。每向上一层，节点的数目就为前一层的一般，而子树高度加 1，因此移动层数加一。**时间复杂度为 O(n)**
    2. **堆的大小未知(如数据流)**：由于插入节点的时间代价为 O(logn)，对于 n 个元素，每个执行一次插入操作，所以**时间复杂度为 O(nlogn)**

二叉树

两种特殊二叉树

1. **满二叉树**（下图左）：除叶子节点外的所有分支节点都含有 2 个非空子节点的二叉树
2. **完全二叉树**（下图右）：除了最后一层，其余层都是“满”的，这样的二叉树是完全二叉树

二叉树定理

1）任意二叉树度数为 2 节点的个数等于叶节点个数减 1

当只有 1 个节点时，度为 0。每派生出 1 度，就会多出 1 个节点。派生出的度和派生出的节点数一定相等。那么就得出了总度数和节点总数的关系：

节点总数 = 总度数 + 1

设度数为 2 的节点数为 X2，度数为 1 的节点数为 X1，度数为 0 的节点数为 X0。可以得出如下关系式：

X2 + X1 + X0 = 2X2 + X1 + 1，推出 X2 = X0 - 1

因此，**度数为 2 的节点个数等于叶节点数减 1**

2）满二叉树定理：非空满二叉树的叶节点数等于其分支节点数加 1

如果已知前一个结论，那么这个定理显然成立。下面分析如果不知道前一个结论，怎么证明

对于只有 1 个节点的树，该定理成立。从这开始思考，每产生 1 个分支节点(度数为 2)。叶子节点数也会加 1。因为要产生一个分支节点，那么这个新的分支节点必然是原来的叶子节点，而新的分支节点又生成了 2 个新的叶子节点。因此叶子节点的总数先是减 1 然后加 2，因此总数加 1。因此，产生 n 个分支节点时，也产生了 n 个叶子节点，由于最初只有 1 个叶子节点，所以该定理成立

3）一颗非空二叉树空子树的数目等于其节点数目加 1

考虑只有 1 个根节点的二叉树：它有 2 个空子树，1 个节点，因此结论成立。从这里开始考虑，每产生 1 个节点。空子树便会先减 1 然后加 2。就和上面结论中每多出 1 个分支节点，叶子节点的变化一样。因此在原来结论的基础上，由于空子树和节点等量增长。所以结论成立

前中后序遍历

3. **前序**遍历：根->左->右
4. **中序**遍历：左->根->右
5. **后序**遍历：左->右->根

假设树节点的定义如下：

```
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode(int x) : val(x), left(NULL), right(NULL) {}
};
```

递归版

```
//前序遍历
void preorderTraversalRecursion(TreeNode *node)
{
    if(!node) return;
```

```
    cout << node->val << " ";//操作当前节点
    preorderTraversalRecursion(node->left);
    preorderTraversalRecursion(node->right);
}

//中序遍历
void inorderTraversalRecursion(TreeNode *node)
{
    if(!node) return;
    inorderTraversalRecursion(node->left);
    cout << node->val << " ";//操作当前节点
    inorderTraversalRecursion(node->right);
}

//后序遍历
void postorderTraversalRecursion(TreeNode *node)
{
    if(!node) return;
    postorderTraversalRecursion(node->left);
    postorderTraversalRecursion(node->right);
    cout << node->val << " ";//操作当前节点
}
```

迭代版

需要使用一个栈作为辅助空间

```
//前序遍历
void preorderTraversalIteration(TreeNode *root)
{
    stack<TreeNode*> st;
    if(root)
        st.push(root);

    while(!st.empty()){
        TreeNode *nd = st.top();
        st.pop();

        cout << nd->val << " ";//操作当前节点

        if(nd->right)
            st.push(nd->right);
        if(nd->left)
            st.push(nd->left);
    }
}

//中序遍历：
void inorderTraversalIteration(TreeNode *root)
{
    stack<TreeNode*> st;

    TreeNode *curr = root;

    while(curr || !st.empty()){
        if(curr){
            st.push(curr);
            curr = curr->left;
```

```
        }
        else{
            curr = st.top();
            st.pop();

            cout << curr->val << " ";//操作当前节点

            curr = curr->right;
        }
    }
}

//后序遍历
void postorderTraversalIteration(TreeNode *root)
{
    stack<TreeNode*> st;
    TreeNode *pre;

    if(root)
        st.push(root);

    while(!st.empty()){
        TreeNode *nd = st.top();
        /*
         * 出栈条件：
         * 对于叶子节点：直接弹出
         * 对于非叶子节点：如果已经遍历过其左子节点或右子节点，则弹出
         */
        if((!nd->left && !nd->right) || (pre && (nd->left == pre || nd->right ==
pre))){
            st.pop();
            cout << nd->val <<" ";//操作当前节点
            pre = nd;
        }
        else{//说明是一个非叶子节点，并且还未访问其左右孩子
            if(nd->right)
                st.push(nd->right);
            if(nd->left)
                st.push(nd->left);
        }
    }
}
```

对于后序遍历，由于其访问序列为：左->右->根。因此还有一种方法，可以按类似前序遍历的方式：根->右->左，然后对得到的结果反序

哈希表

槽总数的选择

1. 关键码范围较小
2. 关键码范围较大

简单的哈希函数

冲突解决策略

3. 开哈希法
4. 闭哈希法

1. **哈希**：把关键码值映射到表中的位置来访问记录的过程
2. **哈希函数**：将关键码值映射到位置的函数
3. **槽**：哈希表中的一个位置
4. **冲突**：不同的关键码经过哈希函数哈希后，映射到相同槽的情况
5. **探查序列**：冲突解决策略的闭哈希方法中，如果基位置冲突，需要根据探查函数查找下一个空槽，这个过程产生的序列加上基位置组成了某个关键码的探查序列
6. **基本聚集**：在探查函数的设计中，如果不同基位置关键码产生的探查序列发生重合，会导致对剩余空槽的选择概率不均等 。产生的后果是会导致很长的探查序列。这种现象就是基本聚集
7. **二次聚集**：基位置相同的关键码，产生的探查序列一样。如果哈希函数在某个基位置聚集，仍然会保持聚集

哈希方法不适用于下列场景：

8. 不适用于范围检索
9. 不能找到具有最大或最小关键码值的记录
10. 不能按关键码值的顺序访问记录

哈希方法既适合基于内存的检索，也适合基于磁盘的检索。是组织存储在磁盘上的大型数据库的主要方法之一（另一种是 B 树）

槽总数的选择

关键码范围较小

由于关键码范围比较小，可以使用一个槽总数大于关键码总数的表。直接使用槽的下标作为关键码值，此时，不需要将关键码值作为记录的一部分进行存储。哈希函数可以直接设计成 h(K)=K，但是这种情况比较少见

关键码范围较大

如果可能的关键码范围较大，而同一时间段内存储的记录总数较少时。如果槽数的设计和前者匹配通常意味着空间的浪费，而如果和后者匹配又容易导致冲突

除此之外，如果对关键码值的分布特性不了解，也会使得哈希函数的设计更为困难。如果了解关键码值的分布特性，应对使用一个依赖于分布的哈希函数，避免把一组相关的关键码值映射到表的同一个槽中（例如，如果对英文单词进行哈希，就不应当对第一个字符的值哈希，因为这样很可能使分布不均）

简单的哈希函数

关键码为数值的哈希函数的设计：

1. **取模**：哈希函数的返回值(槽的位置)只依赖于关键码的最低几位，由于这些位的分布可能很差，结果分布也就可能很差
2. **平方取中**：一个很好的用于数值的哈希函数。对于长度为 2^r 的表，取出平方后结果的中间 r 位作为槽的位置。由于关键码值的大多数位或者所有位都对结果有所贡献，所有效果很好

关键码为字符串的哈希函数的设计：

3. **所有字母 ASCII 值求和对 M 取模**

冲突解决策略

尽管哈希函数的目标是使冲突最少，但实际上冲突是无法避免的。**冲突解决技术可以分为两类**：

- **开哈希(单链表)法**
- **闭哈希(开放地址)法**

开哈希法

开哈希(单链表)法把冲突记录存储在表外，一种简单的形式是把哈希表中的每个槽定义为一个链表的表头，哈希到一个槽的所有记录都放到该槽的链表内，每个链表可以按如下方式组织记录：

- **按插入次序排序**：实现简单
- **按关键码值次序排序**：一旦到达比要检索的关键码大的节点，说明不存在，就可以停止检索
- **按访问频率次序排序**：访问较高的记录能快速检索到

在磁盘中用一种很有效的方式存储一个开哈希表是很困难的，因为一个链表中的多个元素能存储在不同的磁盘块中。这就会导致检索一个关键码值需要多次磁盘访问，从而抵消了哈希方法的好处

闭哈希法

闭哈希(开放地址)法把冲突记录存储在表中另一个槽内。每条记录 i 有一个**基位置**，即由哈希函数计算出的槽。如果要插入一条记录 R，而另一条记录已经占据了 R 的基位置，那么就把 R 存储在表中的其他槽

内，由冲突解决策略决定应该是哪个槽。自然，检索时也要像插入一样，遵循同样的策略，以便重复进行冲突解决过程，找出在某位置没有找到的记录

1）桶式哈希

1. **插入**：将 M 个槽分成 B 个桶，每个桶中包含 M/B 个槽。哈希函数把每条记录分配到某个桶的第一个槽中。如果该槽被占用，就顺序地沿着桶查找，直到找到一个空槽。如果一个桶全部被占满，那么就把这条记录存储在表后具有无限容量的**溢出桶**中，所有桶共享一个溢出桶
2. **检索**：确定桶，然后在桶中检索记录，如果没找到并且桶内有空槽，则检索结束。否则，检索溢出桶

桶式哈希适用于实现基于磁盘的哈希表，因为可以把桶的大小设置为磁盘块的大小。当检索时，就把整个桶读入内存。处理插入或检索操作只需进行一次磁盘访问，除非桶已经满了。如果桶满，需要从磁盘中检索溢出桶，自然应该使溢出很小，以最小化不必要的磁盘访问

2）线性探查

探查序列：通过哈希函数计算出关键码的基位置，如果基位置发生冲突，根据**探查函数**去寻找下一个槽，直到找到一个空槽，这个过程产生的一组槽序列，就是探查序列

线性探查就是探查函数线性递增的冲突解决策略，如果基位置为 30，整个探查序列会是 30,31,32,33…

线性探查的问题在于，会产生**基本聚集**，基本聚集是指不同基位置的关键码产生的探查序列会发生重合，导致对剩余空槽的选择概率不均等 。产生的后果是会导致很长的探查序列

在上图 a）中，如果使用线性探查，基位置为 0,1,2 的关键码在探查后都会选择序号为 2 的槽，同样，基位置为 7,8,9 的关键码在探查后都会选择序号为 9 的槽，这就使得剩余槽被选择的概率不相等。在图 b）中，这个问题会更明显，如果下一个记录插入了序号为 9 的槽，则序号为 2 的空槽被插入记录的概率将是 6/10

系数大于 1 的线性探查(对线性探查函数添加**常数 C** 跳过一些槽)，即：`(h(K) + iC) mod M`

比如当 C 为 2 时，基位置为 1 和 2 产生的探查序列为 1,3,5..和 2,4,6..这个方法对于不同关键码，将关键码值分成了几个集合，每个集合中的关键码只会探查所有槽中的一个部分。同时，相同集合中的关键码还是可能聚集

为了使探查序列走遍表中所有的槽，常数 C 必须与 M 互质（即 C 为质数或 M 为质数），如果 C 与 M 互质，那么任何关键码的探查序列都会走遍所有的槽

3）解决聚集的方法

1. **二次探查**：探查函数为 i 的平方，即基位置为 30 的关键码，产生的探查序列为 30,31,34,39..这种方法的缺陷在于并不是哈希表中所有的槽都在探查序列中

上述方法虽然能解决基本聚集，但是对于基位置相同的关键码，产生的探查序列还是一样。如果哈希函数在某个基位置聚集，那么上面的方法仍然会保持聚集。也就是所谓的**二次聚集**。解决二次聚集可以使用**双哈希**

2. **双哈希**：要使具有相同基位置的关键码产生不同的探查序列，那么探查函数也应该是基于关键码的函数。假设这个函数为 h2(K)，一种方式是根据这个函数产生线性探查序列，即 i*h2(K)。例如，如果 h2(K)=2，那么基位置为 30 的关键码产生的探查序列是：30，32，34…，由于 h2 是基于关键码值的函数，所以基位置相同的不同关键码会产生不同的探查序列，因此可以解决二次聚集

进程线程

一.进程创建

1.fork、vfork、clone

三个函数分别调用了 sys_fork、sys_vfork、sys_clone，最终都调用了 do_fork 函数

3. **fork()函数**
    1. 传统的 fork()系统调用在 Linux 中是用 clone()实现的，其中 clone()的 flags 参数指定为 SIGCHLD 信号及所有清 0 的 clone 标志
    2. 创建的子进程复制了父进程的资源，使用 copy-on-write 技术，子进程与父进程使用相同的物理页，只有子进程视图写一个物理页时，才 copy 这个物理页到一个新的物理页，子进程使用新的物理页，与父进程的内存地址分开，开始独立运行（fork）
4. **vfork()函数**
    1. vfork()系统调用在 Linux 中也是用 clone()实现的，其中 clone()的参数 flags 指定为 SIGCHLD 信号和 CLONE_VM 和 CLONE_VFORK 标志，clone()的参数 child_stack 等于父进程当前的栈指针

2. 创建的子进程运行在父进程的地址空间上，子进程对虚拟地址空间数据的修改对父进程都可见，这与 fork 完全不同，fork 是独立地址空间，而 vfork 是共享父进程空间，为了防止父进程重写子进程需要的数据，父进程被阻塞，直到子进程执行 exec()或 exit()。vfork 是一个过时的应用,vfork 最初是因为 fork 没有实现 COW 机制，而很多情况下 fork 之后会紧接着 exec，而 exec 的执行相当于之前 fork 复制的空间全部变成了无用功，所以设计了 vfork。现在 fork 使用了 COW 机制，唯一的代价仅仅是复制父进程页表的代价，所以 vfork 不应该出现在新的代码之中（vfork）
3. Linux 内核使用进程描述符 `task_struct` 记录进程信息，每个 Linux 线程都有唯一隶属于自己的 `task_struct` 结构，所以在内核中，它看起来就像一个普通进程（只是和其它一些进程共享某些资源，如地址空间）。vfork 创建的子进程作为父进程的一个单独的线程在它的地址空间运行。

5. **clone()函数**可以有选择性地继承父进程的资源，如果选择共享父进程空间，那么创建的是一个线程，也可以选择不与父进程共享同一空间
   1. `fn`：由新进程执行的函数
   2. `arg`：指向传递给 fn()函数的数据
   3. `flags`：各种各样的信息。低字节指定子进程结束时发送到父进程的信号代码，通常选择 SIGCHLD 信号。剩余的 3 个字节给一 clone 标志组用于编码
   4. `child_stack`：表示把用户态堆栈指针赋给子进程的 esp 寄存器。调用进程应该总是为子进程分配新的堆栈

二.进程切换

1. Linux 中的软中断和工作队列的作用

6. 软中断一般是“可延迟函数”的总称，它不能睡眠、不能阻塞，处于中断上下文，不能进行进程切换，软中断不能被自己打断，只能被硬件中断打断，可以并发运行在多个 cpu 上，所以软中断必须设计为可重入函数（允许多个 cpu 同时操作），因此也需要自旋锁来保护其数据结构
7. 工作队列中的函数处于进程上下文中，可以睡眠、阻塞。能够在不同进程间切换，以完成不同的工作

三.特殊进程

1. 如何实现守护进程

8. 守护进程独立于控制终端，并且周期性地执行某种任务或者等待处理某些发生的事件
9. 要实现守护进程，需要将它从启动它的父进程的运行环境中隔离开来，需要处理的内容大致包括会话、控制终端、进程组、文件描述符、文件权限掩码以及工作目录等
   1. 创建子进程，父进程退出（子进程继承了父进程的会话、进程组、控制终端等信息）
   2. 调用 setsid 函数，创建一个新会话，使当前进程脱离原会话的控制，使当前进程脱离原进程组的控制，使当前进程脱离原控制终端的控制
   3. 改变当前目录为根目录，直接调用 chdir 函数
   4. 重设文件权限掩码
   5. 关闭文件描述符

参考

四.进程与线程的限制

1. 进程与线程数量的限制

1）线程数量的限制

限制

10. 线程的数量取决于线程栈空间的大小（可以使用 `ulimit -s` 查看栈空间大小）
11. 32 位 Linux 下（**可以使用 `getconf LONG_BIT` 查看当前 CPU 运行在多少位的模式下**），用户空间是 3G，因此可创建的线程数量为 `3G/stack_size`，但是理论上除了栈空间每个线程还有线程控制块的开销，所以实际值会小一些

修改限制

- 使用 ulimit -s 新栈空间大小修改默认栈空间大小
  1. 可以在/etc/rc.local 内 加入 ulimit -s 新栈空间大小 则可以开机就设置栈空间大小
- 通过/etc/security/limits.conf 改变栈空间大小： #<domain> <type> <item> <value> #添加下列行 * soft stack 新栈空间大小 重新登录，执行 ulimit -s 即可看到改为新栈空间大小

ulimit 命令只对当前终端生效。如果需要永久生效：1）将命令写至 profile 和 bashrc 中，相当于在登录时自动动态修改限制；2）在/etc/security/limits.conf 中添加记录（需重启生效，并且在/etc/pam.d 中的 session 有使用到 limit 模块）

12. linux 下进程的最大线程数、进程最大数、进程打开的文件数
13. linux 查看修改线程默认栈空间大小

线程的限制

2）进程数量的限制

14. **最大理论数**
    1. 每个进程都要在**全局段描述表 GDT** 中占据两个表项
       1. 每个进程的**局部段描述表 LDT** 都作为一个独立的段而存在，在全局段描述表 GDT 中要有一个表项指向这个段的起始地址，并说明该段的长度以及其他一些参数
       2. 每个进程还有一个 **TSS 结构**（**任务状态段**）也是一样 所以，每个进程都要在全局段描述表 GDT 中占据两个表项
    2. GDT 容量？
       1. 段寄存器中用作 GDT 表下标的位段宽度是 13 位，所以 GDT 中可以有 2^13=8192 个描述项
       2. 除一些系统的开销(例如 GDT 中的第 2 项和第 3 项分别用于内核 的代码段和数据段，第 4 项和第 5 项永远用于当前进程的代码段和数据段，第 1 项永远是 0，等等)以外，尚有 8180 个表项可供使用。**所以理论上系统中最大的进程数量是** 8180/2=4090
15. **可创建的实际数**
    1. linux 内核通过**进程标识符 PID** 来标识进程，为了与老版本的 Unix 或者 Linux 兼容，PID 的最大值默认设置为 32768。可以通过 cat /proc/sys/kernel/pid_max 查看 PID 的最大值：
       bash chenximing@chenximing-MS-7823:~$ cat /proc/sys/kernel/pid_max
       32768
    2. 可以通过下面方式修改最大 PID
       1. 首先，ulimit -u 新值
       2. 然后，还需设置内核参数 kernel.pid_max：sysctl -w kernel.pid_max=新值

内存管理

一.内核内存分配

3. **vmalloc()**
   - 内核用于申请大块内存，特点是虚拟地址空间连续，物理地址不一定连续，不能直接用于 DMA
   - vmalloc()分配比 kmalloc()慢
   - 对应释放函数是 vfree()
4. **kmalloc()**
   - 内核用于申请小块内存，保证分配的内存在物理地址上是连续的
   - 它基于 slab 分配器
   - 总是调用_get_free_pages()来进行实际分配，最多只能分配 32 个 page 大小的内存（128K）
   - 分配的是常驻内存，不会被交换到文件中
   - 对应的释放函数是 kfree()

二.伙伴系统

1. 通过伙伴系统申请内核内存的函数有哪些

在物理页面管理上实现了基于区的伙伴系统（zone based buddy system），对不同区的内存使用单独的伙伴系统管理，而且独立地监控空闲页，相应接口：alloc_pages 等

2. 伙伴系统算法

在实际应用中，经常需要分配一组连续的页框，而频繁地申请和释放不同大小的连续页框，必然导致已分配的内存块中分散了很多小块的空闲页框

为了避免这种情况，Linux 内核中引入了伙伴算法（buddy system），把所有的空闲页框分组为 11 个块链表，每个块链表包含大小为 1，2，4，8，16，32，64，128，256，512 和 1024 个连续页框的页框块，最大可以申请 1024 个连续页框块，对应 4MB 大小的连续内存（从小往大查找大小最合适的链表）。每个页框块的第一个页框的物理地址是这块大小的整数倍

- 假设要申请 256 个页框的块，先从 256 个页框的链表中查找空闲块，如果没有，就去 512 个页框的链表中查找，找到了则将页框分为 2 个 256 个页框的块，一个分配给应用，一个移动到 256 个页框的链表中，如果 512 个页框的链表中没有空闲块，则继续从 1024 个页框的链表中查找
- 页框块在释放的时候，会主动将两个连续的页框块合并为一个较大的页框块

Linux 内存管理伙伴算法

三.slab 分配器

工作于伙伴系统算法之上，其基本思想是将内核中经常使用的对象放到高速缓存中，并且由系统保持为初始的可用状态

比如进程描述符，内核会频繁对此数据进行申请和释放，当一个新进程创建时，内核会直接从 slab 分配器的高速缓存中获取一个已经初始化了对象；当进程结束时，该结构所占的页框并不被释放，而是重新返回 slab 分配器中，如果没有基于对象的 slab 分配器，内核将花更多的时间去分配、初始化以及释放对象

每种对象一个高速缓存。这个缓存可以看做是同类型对象的一种储备，**每个高速缓存所占的内存区又被划分为多个 slab，每个 slab 是由一个或多个连续的页框组成**（缺省情况下，一个 slab 最多由 1024 个页框构成），**每个页框中包含若干个对象，既有已经分配的对象，也包含空闲的对象**（处于对齐等其他方面的要求，slab 中分配给对象的内存可能大于用户要求的对象实际大小，这会造成一定的内存浪费）

- slab 结构的最高层是 cache_chain，这是一个 slab 缓存的链接列表，可以用来最适合所需要的分配大小的缓存（伙伴系统算法）
- 每个缓存 kmem_cache 都包含了一个 slabs 列表，这是一段连续的内存块（通常是页面），存在 3 种 slab
  - **slabs_full**：完全分配的 slab
  - **slabs_partial**：部分分配的 slab
  - **slabs_empty**：空 slab，或者没有对象被分配（该列表中的 slab 是进行回收的主要备选对象）
- slab 列表中的每个 slab 都是一个连续的内存块（一个或多个连续页），被划分成一个个对象，这些对象都是从特定缓存中进行分配和释放的基本元素
- 由于对象是从 slab 中进行分配和释放的，因此单个 slab 可以在 slab 列表之间进行移动（例如：当一个 slab 中所有对象都被使用完之后，就从 slabs_partial 列表中移动到 slabs_full 列表中。当一个 slab 完全被分配并且有对象被释放后，就从 slab_full 列表中移动到 slab_partial 列表中）

1. 通过 slab 分配器申请内核内存的函数有哪些

kmem_cache_create/kmem_cache_alloc 是基于 slab 分配器的一种内存分配方式，适用于反复分配同一大小内存块的场合，首先用 kmem_cache_create 创建一个高速缓冲区，然后用 kmem_cache_alloc 从高速缓存区中获取新的内存块

Linux slab 分配器剖析

slub 分配器：简化 slab 分配器的设计理念，同时保留 slab 分配器的基本思想：每个缓冲区由多个 slab 组成，同时每个 slab 包含固定数目的对象。简化了 kmem_cache,slab 等相关的管理数据结构，摒弃了 slab 分配器中众多的队列概念，并针对多处理器、NUMA 系统进行了优化

四.Linux 内核空间与用户空间

1. Linux 内核空间与用户空间是如何划分的

- 内核将 4G 空间划分为 2 部分，最高的 1G 空间供内核使用，称为内核空间，较低的 3G 空间称为用户空间。每个进程通过系统调用进入内核，因此 Linux 内核由所有进程共享
- 从进程角度来看，每个进程可以拥有 4G 的虚拟空间
- Linux 内核和用户程序都运行在虚拟地址模式

16. 内排序
    1. 1.插入排序（稳定）
    2. 2.冒泡排序（稳定）
    3. 3.选择排序（不稳定）
    4. 4.shell 排序（不稳定）
    5. 5.快速排序（不稳定）
    6. 6.归并排序（稳定）
    7. 7.堆排序（不稳定）
17. 外排序
    1. 1.多路归并

稳定性：相同的元素在排序前和排序后的前后位置是否发生改变，没有改变则排序是稳定的，改变则排序是不稳定的 —— 八大排序算法的稳定性

1. 插入排序

逐个处理待排序的记录，每个记录与前面已排序的子序列进行比较，将它插入子序列中正确位置**插入排序会将之前的所有的比它大的元素进行两两交换（倒序）**

代码

```
template<class Elem>
void inssort(Elem A[],int n)
{
    for(int i = 1;i < n;i++)
        for(int j = i;j >= 1 && A[j] < A[j-1];j--)
            swap(A,j,j-1);
}
```

性能

- 最佳：升序。时间复杂度为 O(n)
- 最差：降序。时间复杂度为 O(n^2)
- 平均：对于每个元素，前面有一半元素比它大。时间复杂度为 O(n^2)

如果待排序数据已经“**基本有序”，使用插入排序可以获得接近 O(n)的性能**

优化

```
template<class Elem>
void inssort(Elem A[],int n)
{
    for(int i = 1;i < n;i++){
        int j = i;// i 个元素有序
        int tp = A[j];  //  临时变量 tp 存待插入 A[j]=a[i]
        for(;j >= 1 && tp < A[j-1];j--)// tp 倒序比较，j 在前移
            A[j] = A[j - 1];
        A[j] = tp;
    }
}
```

2. 冒泡排序

从数组的底部比较到顶部，**比较相邻元素**。如果下面的元素更小则交换，否则，上面的元素继续往上比较。这个过程每次使最小元素像个“气泡”似地被推到数组的顶部

代码

```
template<class Elem>
void bubsort(Elem A[],int n)
{
    for(int i = 0;i < n - 1;i++)
        for(int j = n - 1;j > i;j--)
            if(A[j] < A[j-1])
                swap(A,j,j-1);
}
```

性能

冒泡排序是一种相对较慢的排序，没有较好的最佳情况执行时间。通常情况下时间复杂度都是 O(n^2)

优化

增加一个变量 flag，用于记录一次循环是否发生了交换，如果没发生交换说明已经有序，可以提前结束

3. 选择排序

第 i 次“选择”数组中第 i 小的记录，并将该记录放到数组的第 i 个位置。换句话说，每次从未排序的序列中找到最小元素，放到未排序数组的最前面

代码

```
template<class Elem>
void selsort(Elem A[],int n)
{
    for(int i = 0;i < n - 1;i++){
        int lowindex = i;
        for(int j = i + 1;j < n;j++)
            if(A[j] < A[lowindex])
                lowindex = j;
        swap(A,i,lowindex);//n 次交换
    }
}
```

性能

不管数组是否有序，在从未排序的序列中查找最小元素时，都需要遍历完最小序列，所以时间复杂度为 O(n^2)

优化

每次内层除了找出一个最小值，同时找出一个最大值（初始为数组结尾）。将最小值与每次处理的初始位置的元素交换，将最大值与每次处理的末尾位置的元素交换。这样一次循环可以将数组规模减小 2，相比于原有的方案（减小 1）会更快

4. shell 排序

shell 排序在不相邻的元素之间比较和交换。利用了插入排序的最佳时间代价特性，它试图将待排序序列变成基本有序的，然后再用插入排序来完成排序工作

在执行每一次循环时，Shell 排序把序列分为互不相连的子序列，并使各个子序列中的元素在整个数组中的间距相同，每个子序列用**插入排序**进行排序。每次循环增量是前一次循环的 1/2，子序列元素是前一次循环的 2 倍

最后一轮将是一次“正常的”插入排序（即对包含所有元素的序列进行插入排序）

代码

```
const int INCRGAP = 3;

template<class Elem>
void shellsort(Elem A[],int n)
```

```
{
    for(int incr = n / INCRGAP;incr > 0;incr /= INCRGAP){//遍历所有增量大小
        for(int i = 0;i < incr;i++){
            /*对子序列进行插入排序，当增量为1 时，对所有元素进行最后一次插入排序*/
            for(int j = i + incr;j < n;j += incr){
                for(int k = j; k > i && A[k] < A[k - incr];k -= incr){
                    swap(A,k,k - incr);
                }
            }
        }
    }
}
```

性能

选择适当的增量序列可使 Shell 排序比其他排序法更有效，一般来说，增量每次除以 2 时并没有多大效果，而“增量每次除以 3”时效果更好

当选择“增量每次除以 3”递减时，Shell 排序的平均运行时间是 O(n^(1.5))

5. 快速排序

首先选择一个轴值，小于轴值的元素被放在数组中轴值左侧，大于轴值的元素被放在数组中轴值右侧，这称为数组的一个分割(partition)。快速排序再对轴值左右子数组分别进行类似的操作

选择轴值有多种方法。最简单的方法是使用首或尾元素。但是，如果输入的数组是正序或者逆序时，会将所有元素分到轴值的一边。较好的方法是随机选取轴值

代码

```
template <class Elem>
int partition(Elem A[],int i,int j)
{
    //这里选择尾元素作为轴值,轴值的选择可以设计为一个函数
    //如果选择的轴值不是尾元素，还需将轴值与尾元素交换
    int pivot = A[j];
    int l = i - 1;
    for(int r = i;r < j;r++)
        if(A[r] <= pivot)
            swap(A,++l,r);
    swap(A,++l,j);//将轴值从末尾与++l 位置的元素交换
    return l;
}

template <class Elem>
void qsort(Elem A[],int i,int j)
{
    if(j <= i)  return;
    int p = partition<Elem>(A,i,j);
    qsort<Elem>(A,i,p - 1);
    qsort<Elem>(A,p + 1,j);
}
```

性能

18. 最佳情况：O(nlogn)
19. 平均情况：O(nlogn)
20. 最差情况：每次处理将所有元素划分到轴值一侧，O(n^2)

快速排序平均情况下运行时间与其最佳情况下的运行时间很接近，而不是接近其最坏情况下的运行时间。**快速排序是所有内部排序算法中平均性能最优的排序算法**

优化

- 最明显的改进之处是轴值的选取，如果轴值选取合适，每次处理可以将元素较均匀的划分到轴值两侧：

三者取中法：三个随机值的中间一个。为了减少随机数生成器产生的延迟，可以选取首中尾三个元素作为随机值

- 当 n 很小时，快速排序会很慢。因此当子数组小于某个长度（经验值：9）时，什么也不要做。此时数组已经基本有序，最后调用一次插入排序完成最后处理

6. 归并排序

将一个序列分成两个长度相等的子序列，为每一个子序列排序，然后再将它们合并成一个序列。合并两个子序列的过程称为归并

代码

```
template<class Elem>
void mergesortcore(Elem A[],Elem temp[],int i,int j)
{
    if(i == j)  return;
    int mid = (i + j)/2;

    mergesortcore(A,temp,i,mid);
    mergesortcore(A,temp,mid + 1,j);

    /*归并*/
    int i1 = i,i2 = mid + 1,curr = i;
    while(i1 <= mid && i2 <= j){
        if(A[i1] < A[i2])
            temp[curr++] = A[i1++];
        else
            temp[curr++] = A[i2++];
    }
    while(i1 <= mid)
        temp[curr++] = A[i1++];
    while(i2 <= j)
        temp[curr++] = A[i2++];
    for(curr = i;curr <= j;curr++)
        A[curr] = temp[curr];
}

template<class Elem>
void mergesort(Elem A[],int sz)
{
    Elem *temp = new Elem[sz]();
    int i = 0,j = sz - 1;
    mergesortcore(A,temp,i,j);
    delete [] temp;
}
```

性能

logn 层递归中，每一层都需要 O(n)的时间代价，因此总的时间复杂度是 O(nlogn)，该时间复杂度不依赖于待排序数组中数值的相对顺序。因此，是最佳，平均和最差情况下的运行时间

由于需要一个和带排序数组大小相同的辅助数组，所以空间代价为 O(n)

优化

原地归并排序不需要辅助数组即可归并

```
void reverse(int *arr,int n)
{
    int i = 0,j = n - 1;
    while(i < j)
```

```
        swap(arr[i++],arr[j--]);
}

void exchange(int *arr,int sz,int left)
{
    reverse(arr,left);//翻转左边部分
    reverse(arr + left,sz - left);//翻转右边部分
    reverse(arr,sz);//翻转所有
}

void merge(int *arr,int begin,int mid,int end)
{
    int i = begin,j = mid,k = end;
    while(i < j && j <= k){
        int right = 0;
        while(i < j && arr[i] <= arr[j])
            ++i;
        while(j <= k && arr[j] <= arr[i]){
            ++j;
            ++right;
        }
        exchange(arr + i,j - i,j - i - right);
        i += right;
    }
}
```

7. 堆排序

堆排序首先根据数组构建最大堆，然后每次“删除”堆顶元素（将堆顶元素移至末尾）。最后得到的序列就是从小到大排序的序列

代码

这里直接使用 C++ STL 中堆的构建与删除函数

```
template <class Elem>
void heapsort(Elem A[],int n)
{
    Elem mval;
    int end = n;
    make_heap(A,A + end);
    for(int i = 0;i < n;i++){
        pop_heap(A,A + end);
        end--;
    }
}
```

如果不能使用现成的库函数：

```
/*******************************************
 * 向堆中插入元素
 *  hole：新元素所在的位置
 *******************************************/
template <class value>
void _push_heap(vector<value> &arr,int hole){
    value v = arr[hole];//取出新元素，从而产生一个空洞
    int parent = (hole - 1) / 2;
    //建最大堆，如果建最小堆换成 arr[parent] > value
    while(hole > 0 && arr[parent] < v){
        arr[hole] = arr[parent];
```

```
        hole = parent;
        parent = (hole - 1) / 2;
    }
    arr[hole] = v;
}

/*****************************************
 * 删除堆顶元素
 *****************************************/
template <class value>
void _pop_heap(vector<value> &arr,int sz)
{
    value v = arr[sz - 1];
    arr[sz - 1] = arr[0];
    --sz;
    int hole = 0;
    int child = 2 * (hole + 1); //右孩子
    while(child < sz){
        if(arr[child] < arr[child - 1])
            --child;
        arr[hole] = arr[child];
        hole = child;
        child = 2 * (hole + 1);
    }
    if(child == sz){
        arr[hole] = arr[child - 1];
        hole = child - 1;
    }
    arr[hole] = v;
    _push_heap(arr,hole);
}

/*****************************************
 * 建堆
 *  sz: 删除堆顶元素后的大小
 *  v:  被堆顶元素占据的位置原来的元素的值
 *****************************************/
template <class value>
void _make_heap(vector<value> &arr)
{
    int sz = arr.size();
    int parent = (sz - 2) / 2;
    while(parent >= 0){
        int hole = parent;
        int child = 2 * (hole + 1); //右孩子
        value v = arr[hole];
        while(child < sz){
            if(arr[child] < arr[child - 1])
                --child;
            arr[hole] = arr[child];
            hole = child;
            child = 2 * (hole + 1);
        }
        if(child == sz){
            arr[hole] = arr[child - 1];
            hole = child - 1;
```

```
        }
        arr[hole] = v;
        _push_heap(arr,hole);
        --parent;
    }
}

template <class value>
void heap_sort(vector<value> &arr)
{
    _make_heap(arr);
    for(int sz = arr.size();sz > 1;sz--)
        _pop_heap(arr,sz);
}
```

性能

根据已有数组构建堆需要 O(n)的时间复杂度，每次删除堆顶元素需要 O(logn)的时间复杂度，所以总的时间开销为，O(n+nlogn)，平均时间复杂度为 O(nlogn)

注意根据已有元素建堆是很快的，如果希望找到数组中第 k 大的元素，可以用 O(n+klogn)的时间，如果 k 很小，时间开销接近 O(n)

8. 多路归并

多路归并是**外部排序最常用**的算法：**将原文件分解成多个能够一次性装入内存的部分，分别把每一部分调入内存完成排序。然后，对已经排序的子文件进行归并排序**

k 的选择

假设总共 m 个子文件，每次归并 k 个子文件，那么一共需要归并 $//log_k m$ 次（扫描磁盘），在 k 个元素中找出最小值（或最大值）需要比较 k-1 次。如果总记录数为 N，所以时间复杂度就是 $//(k-1)Nlog_k m = /frac(k-1)logkNlogm$，由于 $///frac(k-1)logk$ 随 k 的增大而增大，所以比较次数的增加会逐步抵消“低扫描次数”带来的性能增益，所以对于 k 值的选择，主要涉及两个问题：

- **每一轮归并会将结果写回到磁盘，那么 k 越小，磁盘与内存之间数据的传输就会越多，增大 k 可以较少扫描次数**
- **k 个元素中选取最小的元素需要比较 k-1 次，如果 k 越大，比较的次数就会越大**

优化

可以利用下列方法**减少比较次数**：

- **败者树**
- **建堆**：使用一个 k 个元素的数组，第一次将 k 个文件中最小的元素读入数组（并且记录每个元素来自哪个文件），然后建最小堆，将堆顶元素删除，并从堆顶元素的源文件中取出下一个数，插入堆中，调整后重复上述操作。虽然第一次需要遍历 k 个文件取出最小元素，加上建堆需要一定时间，但是后续操作可以很快完成

平衡查找树

一般的二插查找树如果节点有序插入，树的高度会是 n，因此无法实现 logn 的查找，平衡查找树保证树的高度平衡，因此不管节点插入顺序如何，都可以满足 logn 的查找

2-3 查找树

一棵 2-3 查找树或为一棵空树，或由以下节点组成：

2-节点：含有 1 个键（及其对应的值）和 2 条链接，左链接指向的 2-3 树中的键都小于该节点，右链接指向的 2-3 树中的键都大于该节点

3-节点：含有 2 个键（及其对应的值）和 3 条链接，左链接指向的 2-3 树中的键都小于该节点，中链接指向的 2-3 树中的键都位于该节点的两个键之间，右链接指向的 2-3 树中的键都大于该节点

1. 查找

先将查找的键与根节点中的键比较。如果和其中任意一个相等，则查找命中；否则根据比较的结果找到指向相应区间的链接，并在其指向的子树中递归地继续查找。如果这是个空链接，查找未命中

2. 插入

1）向 2-节点中插入新键

如果未命中的查找结束于一个 2-节点，只要把这个 2-节点替换为一个 3-节点，将要插入的键保存在其中即可。由于整棵树的节点并未发生变化，所以这个插入操作并没有引起高度变化，如果原本树是平衡的，那么插入后也能保持平衡

2）向 3-节点中插入新键

1. **整棵树只包含一个 3-节点（情况一）**
2. **整棵树包含多个节点**
    1. **3-节点的父节点为 2-节点（情况二）**
    2. **3-节点的父节点为 3-节点（情况三）**

情况一：先临时将新键存入该节点中，使之成为一个 4-节点。然后将这个 4-节点转换成一棵由 3 个 2-节点组成的 2-3 树：其中一个节点(根)含有中键，一个节点含有 3 个键中的最小者(左子节点)，一个节点含有 3 个键中的最大者(右子节点)。**很容易看出插入后的树依然是平衡的**

情况二：先构造一个临时的 4-节点并将其分解，但此时不会像情况一一样为中键创建一个新节点，而是将其移动至原来的父节点中。可以将这次转换看成将原 3-节点的一条链接替换为新父节点中的原中键左右两边的两条链接，并分别指向两个新的 2-节点。**插入后树仍是有序的，并且是完美平衡的**

情况三：和情况二一样，先构造一个临时的 4-节点并分解，将它的中键插入到父节点中。但父节点也是一个 3-节点，因此再用这个中键构造一个新的临时 4-节点，然后在这个节点上进行相同的变换（即分解这个父节点并将它的中键插入到父节点中）。推广到一般情况，这样一直向上不断分解临时的 4-节点并将中键插入更高层的父节点，直至遇到一个 2-节点并将它替换为一个不需要继续分解的 3-节点，或者是到达 3-节点的根（这种情况下，按情况一进行处理，将临时的 4-节点分解为 3 个 2-节点，使得树高加一，因为变换的是根节点，所以**仍然保持树的完美平衡性**）

3. 2–3 树构造实例

下面是根据字符序列[S,E,A,R,C,H,X,M,P,L]构造一棵 2-3 树的过程。左边是乱序插入，右边是顺序插入

可以发现，2-3 树在最坏情况下（顺序插入）仍然可以保证平衡

每个操作中处理每个节点的时间都不会超过一个很小的常数，并且只会访问一条路径上的节点，所以任何查找或插入的成本都肯定不会超过对数级别。含有 10 亿个节点的一颗 2-3 树的高度仅在 19 到 30 之间，最多只需要访问 30 个节点就能够在 10 亿个键中进行查找和插入操作

红黑树

尽管可以用不同的数据类型表示 2-节点和 3-节点并写出转换所需的代码，但用这种直白的表示方法实现大多数的操作并不方便，因为**需要处理的情况实在太多**：需要维护两种不同类型的节点，将被查找的键和节点中的每个键进行比较，将链接和其它信息从一种节点复制到另一种节点，将节点从一种数据类型转换到另一种数据类型，等等。**实现这些不仅需要大量代码，而且它们产生的额外开销可能会使算法比标准的二插查找树更慢**。平衡二叉树的初衷是为了消除最坏情况，但我们希望这种保障所需的代码能够越少越好，因此有了红黑树

红黑树背后的基本思想是：用标准的二插查找树（完全由 2-节点构成）和一些额外的信息（替换 3-节点）来表示 2-3 树。将红黑树中的链接分为 2 种类型：

- **红链接**：红链接将两个 2-节点连接起来构成一个 3-节点
- **黑链接**：2-3 树中的普通链接

将 2-3 树中的 3-节点表示为由一条**左斜的红链接相连的两个 2-节点**（优点是，无需修改就可以直接使用标准二插查找树的 get 方法）

红黑树是满足下列条件的二叉查找树：

- **红链接均为左链接**
- **没有任何一个节点同时和两个红链接相连**
- **该树是完美”黑色“平衡的，即任意空链接到根节点的路径上的黑链接数量相同**

如果将一棵红黑树中的红链接画平，那么所有的空链接到根节点的距离都将是相同的。如果将由红链接相连的节点合并，得到的就是一棵 2-3 树。相反，如果将一棵 2-3 树中的 3-节点画作由红色左链接相连的两个 2-节点，那么不会存在能够和两条红链接相连的节点，且树必然是完美黑色平衡的

1. 保存颜色信息

将链接的颜色保存在表示节点的 Node 数据类型的布尔变量 color 中。如果指向它的链接是红色的，那么该变量为 true，黑色则为 false

2. 旋转操作

某些操作中可能会出现红色右链接或者两条连续的红链接，但在操作完成前这些情况都会被小心地旋转并修复。旋转包括 2 种：

3. **左旋转**（下图左）
4. **右旋转**（下图右）

插入新键时，可以使用旋转操作保证红黑树的**有序性**和**完美平衡性**

3. 插入

1）向 2-节点中插入新键

5. **整棵树只有一个 2-节点**（下图左）：
    1. 插入根节点左侧时，直接插入并设置链接颜色即可，此时**父节点成为一个 3-节点**
    2. 插入根节点右侧时，需要进行左旋转操作，左旋转后的**父节点成为一个 3-节点**
6. **整棵树有多个节点**（下图右）：同上

2）向 3-节点中插入新建

分为下列 3 种情况，**每种情况都会产生一个同时连接到两条红链接的节点，需要使用旋转进行修正**：

7. **新键大于 3-节点的两个键**（下图左）：新键被连接到 3-节点的右链接。如果将两条链接的颜色都由红变黑，就得到了一棵由三个节点组成、高度为 2 的平衡树（相当于 2-3 树 4-节点的分解）
8. **新键小于 3-节点中的两个键**（下图中）：将上层的红链接右转得到情况 1，然后按情况 1 进行处理
9. **新键位于 3-节点两个键之间**（下图右）：将下层红链接左旋转得到上一种情况 2，然后按情况 2 进行处理

4. 颜色变换

向 3-节点插入新建的操作中，都涉及到将一个节点到两个子节点的红链接设置成黑链接的操作。在这个操作中，将两条红链接设置为黑链接使得左子树和右子树的高度都增加了 1，因此还需要将父节点的颜色由黑变红，这样以父节点为根节点的这颗树的高度又减了 1，从而保证整体平衡

如果父节点是整棵树的根节点，这个操作可能会将根节点设置为红，因此每次插入操作后会将根节点设为黑色，每当根节点由红变黑时树的黑链接高度就会加 1

红链接向上传递：每次必要的旋转之后，都会进行颜色转换，这使得中节点变红。在父节点看来，处理这样一个红色节点的方式和处理一个新插入的红色节点完全相同，即继续把红链接转移到中节点上去

下图展示了向一个 3-节点插入新键，红链接的向上传递过程：

5. 旋转与颜色变换过程总结

只要谨慎地使用左旋转、右旋转和颜色转换操作，就能够保证插入操作后红黑树和 2-3 树的一一对应关系。在沿着插入点到根节点的路径向上移动时，在所经过的每个节点中顺序完成以下操作，就能完成插入操作：

1. 如果右子节点是红色的而左子节点是黑色的，进行左旋转
2. 如果左子节点是红色的且它的左子节点也是红色的，进行右旋转
3. 如果左右子节点均为红色，进行颜色转换

6. 红黑树的性质

1）一棵大小为 N 的红黑树的高度不会超过 2lgN

1. **简略的证明**：红黑树的最坏情况是它所对应的 2-3 树中构成最左边的路径节点全部都是 3-节点而其余均为 2-节点，最左边的路径长度是只包含 2-节点的路径长度(~lgN)的两倍（要按照某种顺序构造一棵平均路径长度为 2lgN 的最差红黑树虽然可能，但并不容易）

2）一棵大小为 N 的红黑树中，根节点到任意节点的平均路径长度为~1.00lgN

无论键的插入顺序如何，红黑树都”几乎是“完美平衡的(基本平衡):

B 树与 B+树

B 树简介

B 树只是 2-3 树的一种推广，从另一方面来说，2-3 树是一个 3 阶 B 树

B 树涉及了在实现基于磁盘的检索树结构时遇到的所有问题：

- B 树总是树高平衡的，所有的叶节点都在同一层
- 更新和检索操作只影响一些磁盘块，因此性能很好
- B 树把相关的记录（即关键码有类似的值）放在同一磁盘块中，从而利用了访问局部性原理
- B 树保证了树中至少有一定比例的节点是满的。这样能改进空间效率，同时在检索和更新操作期间减少在一般情况下需要的磁盘读取数

一个 m 阶 B 树有以下特性（**m 阶中的 m 表示节点的最大孩子的节点个数，通常应使 B 树中节点的大小能够填满一个磁盘块**）：

2. 根或者是一个叶节点，或者至少有 2 个孩子节点
3. 除了根节点外，每个内部节点有 m/2(向上取整)到 m 个孩子节点
4. 所有叶节点在树结构的同一层，因此树总是高度平衡的

存储在树中的“指针”值实际上是包含孩子节点的块号，下图为一个 4 阶 B 树

B 树中检索关键码

- 在当前节点中对关键码进行二分查找
    1. 如果找到检索的关键码：返回这条记录
    2. 如果没找到，并且当节点是叶子节点：停止查找，报告检索失败
- 沿着正确的分支继续查找

B 树中插入关键码

B 树节点插入是 2-3 树的推广。第一步是找到应当包含插入关键码的叶节点，并检查是否有空间。如果有，直接插入。如果没有，就把这个节点分裂成 2 个节点，并且把中间的关键码提升到父节点。如果父节点也满了，继续分裂，并再次提升

B+树简介

B 树和 2-3 树实际上从来都没被实现过，最普遍使用的是 B 树的一个变体，称为 B+树。当需要更高的效率时，需要一个更为复杂的变体，称为 B*树

B+树最显著的差异在于：

5. **B+树只在叶节点存储记录(以及指向实际记录的指针)，内部节点存储关键码值，但是这些关键码值只是占据位置，用于引导索引**（意味着内部节点在结构上与叶节点有着显著的差异）
6. **B+树的叶节点一般链接起来，形成一个双链表**（这样，通过访问链表中的所有叶节点，就可以按排序的次序遍历全部记录）
7. **m 阶 B+树中的叶节点可能存储多于或少于 m 条记录**（最多 2m-3 个,因为此时插入一个关键码分裂后的 2 个内部节点都包含 m-1 个关键码，内部节点最多也只能包含这么多关键码），**但是至少要“半满”**
8. **除了有一个指向根节点的头指针，B+树还有一个指向最小关键码的头指针**

B+树特别适合范围查询，一旦找到了范围内的第一条记录，通过顺序处理节点中的其余记录，然后继续下去，尽可能深入叶节点链表，就可以找到范围内的其余记录

B+树中检索关键码

B+树的检索必须一直到达相应的叶节点，除此之外，它与正规 B 树的检索完全一样。即使在内部节点找到了检索关键码值，这个值也只是占据一个位置，并不提供对实际记录的访问

B+树中插入关键码

类似于 B 树插入过程：

9. 找到应当包含记录的**叶节点 L**
    1. 如果 L 没满：将新纪录添加进去
    2. 如果 L 已满：将其分裂成 2 个节点（在两个节点之间平均分配记录），然后在新形成的右边节点把最小关键码值的一份副本提升到父节点。这个提升过程可能会一直进行到根节点，引起根节点分裂，从而增加一层

下图展示了一个插入过程：

- (a)图：原始状态
- (b)图：插入 50
- (c)图：继续经过多次插入后
- (d)图：插入 30

B+树中删除关键码

删除之前，首先要找到包含要删除的记录 R 的**叶节点**：

- 如果叶节点超过半满：
    3. 只需清除记录 R，剩下的叶节点 L 仍然至少半满（**下图 1**）
- 如果叶节点没有半满，查看相邻的兄弟节点（节点下溢）：
    4. 兄弟节点的记录很多：从兄弟节点获取元素，使两个节点有同样多的记录（这样做是为了尽可能延迟由于删除而引起的节点再次下溢）（**下图 2**）

5. 如果没有一个兄弟节点可以把记录借出来，则将叶节点与一个兄弟节点合并（下图 3）。可能会依次使父节点下溢，如果根节点的最后两个子女合并到一起，那么树就会减少一层

B+树要求所有节点至少半满（除根节点外）。这样，存储利用率必须至少达到 50%。对于许多实现来说，已经可以满足要求。但如果能让节点更满的话，需要的空间就会更少（因为磁盘文件中未用的空间更少），而且处理效率也会更高（因为每个块中的信息量更大，平均来说，需要读入内存的块数也就更少）。由于 B+树已经广泛使用，许多人试图改进其性能。一种实现方法是使用 B+树的一个变体，**B*树**。除了分裂、合并节点的规则不同外，B*树与 B+树完全相同。B*树在节点上溢时不把它分裂成两半，而是把一些记录给它相邻的兄弟节点。如果兄弟节点也满了，那么就把 2 个节点分成 3 个。同一，当一个节点下溢时，它就与其两个兄弟结合合并，这三个节点减少为两个节点。这样，节点总是包含至少三分之二的记录

深入理解计算机系统重点笔记

1. 引言

深入理解计算机系统，对我来说是部大块头。说实话，我没有从头到尾完完整整的全部看完，而是选择性的看了一些我自认为重要的或感兴趣的章节，也从中获益良多，看清楚了计算机系统的一些本质东西或原理性的内容，这对每个想要深入学习编程的程序员来说都是至关重要的。只有很好的理解了系统到底是如何运行我们代码的，我们才能针对系统的特点写出高质量、高效率的代码来。这本书我以后还需要多研究几遍，今天就先总结下书中我已学到的几点知识。

---

2. 编写高效的程序需要下面几类活动：

- 选择一组合适的算法和数据结构。这是很重要的，好的数据结构有时能帮助更快的实现某些算法，这也要求编程人员能够**熟知各种常用的数据结构和算法。**
- 编写出使编译器能够有效优化以转换成高效可执行的源代码。因此，**理解编译器优化的能力和局限性是很重要的。**编写程序方式中看上去只是一点小小的变动，都会引起编译器优化方式很大的变化。有些编程语言比其他语言容易优化得多。C 语言的某些特性，例如执行指针运算和强制类型转换的能力，使得编译器很难对其进行优化。
- 并行技术，针对处理运算量特别大的计算，将一个任务分成多个部分，这些部分可以在多核和多处理器的某种组合上并行地计算。

3. 让编译器展开循环

说到程序优化，很多人都会提到循环展开技术。现在编译器可以很容易地执行循环展开，只要优化级别设置的足够高，许多编译器都能例行公事的做到这一点。**用命令行选项"-funroll-loops"调用 gcc，会执行循环展开。**

4. 性能提高技术：

- 高级设计，为手边的问题选择适当的算法和数据结构，要特别警觉，避免使用会渐进地产生糟糕性能的算法或编码技术。
- 基本编码原则。避免限制优化的因素，这样编译器就能产生高效代码。
  - 消除连续的函数调用。在可能时将计算移到循环外，考虑有选择的妥协程序的模块性以获得更大效率。
  - 消除不必要的存储器引用。引入临时变量来保存中间结果，只有在最后的值计算出来时，才能将结果放到数组或全局变量中。
- 低级优化。
  - 尝试各种与数组代码相对的指针形式。
  - 通过开展通过展开循环降低循环开销。
  - 通过诸如迭代分割之类的技术，找到使用流水线化的功能单元的方法。

说到性能提高，可能有人会有一些说法：>（1）不要过早优化，优化是万恶之源；（2）花费很多时间所作的优化可能效果不明显，不值得；（3）现在内存、CPU 价格都这么低了，性能的优化已经不是那么重要了。 ……

其实我的看法是：**我们也许不必特地把以前写过的程序拿出来优化下，花费 N 多时间只为提升那么几秒或几分钟的时间。但是，我们在重构别人的代码或自己最初开始构思代码时，就需要知道这些性能提高技术，一开始就遵守这些基本原则来写代码，写出的代码也就不需要让别人来重构以提高性能了。另外，有的很简单的技术，比如说将与循环无关的复杂计算或大内存操作的代码放到循环外，对于整个性能的提高真的是较明显的。**

5. 如何使用代码剖析程序（code profiler，即性能分析工具）来调优代码？

程序剖析（profiling）其实就是**在运行程序的一个版本中插入了工具代码，以确定程序的各个部分需要多少时间。** Unix 系统提供了一个 profiling 叫 GPROF，这个程序产生两类信息： > 首先，它确定程序中每个函数花费了多少 CPU 时间。 其次，它计算每个函数被调用的次数，以执行调用的函数来分类。还有每个函数被哪些函数调用，自身又调用了哪些函数。

使用 GPROF 进行剖析需要 3 个步骤，比如源程序为 prog.c。 1）编译： `gcc -O1 -pg prog.c -o prog` （只要加上-pg 参数即可） 2）运行： `./prog` 会生成一个 gmon.out 文件供 gprof 分析程序时候使用（运行比平时慢些）。 3）剖析：`gprof prog` 分析 gmon.out 中的数据，并显示出来。 **剖析报告的第一部分列出了执行各个函数花费的时间，按照降序排列。 剖析报告的第二部分是函数的调用历史。**具体例子可参考网上资料。

GPROF 有些属性值得注意： - 计时不是很准确。它的计时基于一个简单的间隔计数机制，编译过的程序为每个函数维护一个计数器，记录花费在执行该函数上的时间。对于运行时间较长的程序，相对准确。 - 调用信息相当可靠。 - 默认情况下，不显示库函数的调用。相反地，库函数的时间会被计算到调用它们的函数的时间中。

6. 静态链接和动态链接一个很重要的区别

是：动态链接时没有任何动态链接库的代码和数据节真正的被拷贝到可执行文件中，反之，链接器只需拷贝一些重定位和符号表信息，即可使得运行时可以解析对动态链接库中代码和数据的引用。

7. 存储器映射

指的是**将磁盘上的空间映射为虚拟存储器区域**。Unix 进程可以使用 **mmap 函数来创建新的虚拟存储器区域，并将对象映射到这些区域中**，这属于低级的分配方式。 一般 C 程序会使用 malloc 和 free 来动态分配存储器区域，这是利用堆的方式。

8. 造成堆利用率很低的主要原因是碎片

当虽然有未使用的存储器但不能用来满足分配请求时，就会发生这种现象。 有两种形式的碎片：内部碎片和外部碎片。**两者的区别如下：**

- 内部碎片是在一个已分配的块比有效载荷大时发生的。例如，有些分配器为了满足对其约束添加额外的 1 字的存储空间，这个 1 字的空间就是内部碎片。它就是已分配块大小和它们的有效载荷大小之差的和。
- 外部碎片是当空闲存储器合计起来足够满足一个分配请求，但是没有一个单独的空闲块足够大可以来处理这个请求时发生的。

9. 现代 OS 提供了三种方法实现并发编程：

- 进程。用这种方法，每个逻辑控制流都是一个进程，由内核来调度和维护。因为进程有独立的虚拟地址空间，想要和其他流通信，控制流必须使用进程间通信（IPC）。
- I/O 多路复用。这种形式的并发，应用程序在一个进程的上下文中显示地调度它们自己的逻辑流。逻辑流被模拟为“状态机”，数据到达文件描述符后，主程序显示地从一个状态转换到另一个状态。因为程序是一个单独的进程，所以所有的流都共享一个地址空间。
- 线程。线程是运行在一个单一进程上下文中的逻辑流，由内核进行调度。线程可以看做是进程和 I/O 多路复用的合体，像进程一样由内核调度，像 I/O 多路复用一样共享一个虚拟地址空间。

（1）基于进程的并发服务器 构造并发最简单的就是使用进程，像 fork 函数。例如，一个并发服务器，在父进程中接受客户端连接请求，然后创建一个新的子进程来为每个新客户端提供服务。为了了解这是如何工作的，假设我们有两个客户端和一个服务器，服务器正在监听一个监听描述符（比如描述符 3）上的连

接请求。下面显示了服务器是如何接受这两个客户端的请求的。

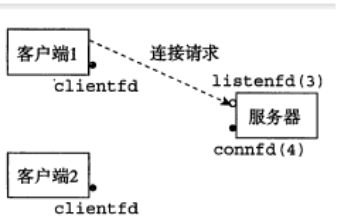


图 12-1 第一步：服务器接受客户端的连接请求

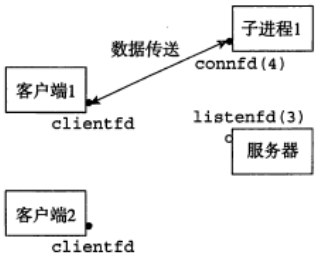


图 12-2 第二步：服务器派生一个子进程为这个客户端服务

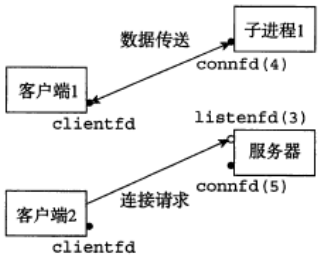


图 12-3 第三步：服务器接受另一个连接请求

关于进程的优劣，对于在父、子进程间共享状态信息，进程有一个非常清晰的模型：共享文件表，但是不共享用户地址空间。进程有独立的地址控件爱你既是优点又是缺点。由于独立的地址空间，所以进程不会覆盖另一个进程的虚拟存储器。但是另一方面进程间通信就比较麻烦，至少开销很高。

（2）基于 I/O 多路复用的并发编程 比如一个服务器，它有两个 I/O 事件：1）网络客户端发起连接请求，2）用户在键盘上键入命令行。我们先等待那个事件呢？没有那个选择是理想的。如果 accept 中等待连接，那么无法响应输入命令。如果在 read 中等待一个输入命令，我们就不能响应任何连接请求（这个前提是一个进程）。 针对这种困境的一个解决办法就是 I/O 多路复用技术。**基本思想是：使用 select 函数，要求内核挂起进程，只有在一个或者多个 I/O 事件发生后，才将控制返给应用程序。** I/O 多路复用的优劣：由于 I/O 多路复用是在单一进程的上下文中的，因此每个逻辑流程都能访问该进程的全部地址空间，所以开销比多进程低得多；缺点是编程复杂度高。

（3）基于线程的并发编程 每个线程都有自己的线程上下文，**包括一个线程 ID、栈、栈指针、程序计数器、通用目的寄存器和条件码。**所有的运行在一个进程里的线程共享该进程的整个虚拟地址空间。由于线程运行在单一进程中，因此共享这个进程虚拟地址空间的整个内容，**包括它的代码、数据、堆、共享库和打开的文件。**所以我认为不存在线程间通信，线程间只有锁的概念。

- 线程执行的模型。线程和进程的执行模型有些相似。每个进程的生明周期都是一个线程，我们称之为主线程。但是大家要有意识：线程是对等的，主线程跟其他线程的区别就是它先执行。 一般来说，线程的代码和本地数据被封装在一个线程例程中（就是一个函数）。该函数通常只有一个指针参数和一个指针返回值。 在 Unix 中线程可以是 joinable（可结合）或者 detached（分离）的。joinable 可以被其他线程杀死，detached 线程不能被杀死，它的存储器资源有系统自动释放。
  - 线程存储器模型，每个线程都有它自己的独立的线程上下文，包括线程 ID、栈、栈指针、程序计数器、条件码和通用目的寄存器。每个线程和其他线程共享剩下的部分，包括整个用户虚拟地址空间，它是由代码段、数据段、堆以及所有的共享库代码和数据区域组成。不同线程的栈是对其他线程不设防的，也就是说：如果一个线程以某种方式得到一个指向其他线程的指针，那么它可以读取这个线程栈的任何部分。

10. 什么样的变量多线程可以共享，什么样的不可以共享？

有三种变量：全局变量、本地自动变量（局部变量）和本地静态变量，其中本地自动变量每个线程的本地栈中都存有一份，不共享。而全局变量和静态变量可以共享。

世界是数学的

《世界是数字的》是世界顶尖计算机科学家 Brian W.Kernighan 写的一本计算机科普类读物，简明扼要但又深入全面地解释了计算机和通信系统背后的秘密，适合计算机初学者和非计算机专业的人读。这真的是一本好书，借 Google 常务董事长的话： >对计算机、互联网及其背后的奥秘充满好奇的人们，这绝对是一本不容错过的好书。

对于一个计算机已经学了 N 年的专业人士来说，这本书也许简单了点，不过我还是认真过了一遍，发现也有一定的收货，因为一个人很难掌握本领域里的所有知识，或多或少会有一些欠缺，总会有一些你以前不

知道的，或一直没理解清楚的但又很有必要知晓的知识，我在阅读此书过程中就有这种感觉，经常会有一种恍然大悟的感觉，比如理解了互联网上一些不为人知的跟踪原理（具体可以看我下面总结的第 12 点“**Cookie 如何暴露你在互联网上的行踪**”）。我是个喜欢记笔记和做总结的人，阅读完一本书，我经常会找个闲暇的时间总结下，主要是根据自己已有的知识储备体系总结一些对我有帮助的或有必要知道的知识点。

下面就简单总结下自己的所获和所感。**注意**：下面的知识都是科普知识，适合非计算机专业、计算机初学者及和像我一样计算机一开始就没学好的人看，那些牛 B 的大牛就不用来浪费时间来读你们已称之为“常识”的知识啦。

1. P 和 NP 问题

现在的程序员都很怕遇到 NP 问题，不仅算法复杂而且还保证不了每次都能找到解。那到底什么是 P 问题和 NP 问题呢？**作为一个程序员，你如果回答说“P 问题就是容易的问题，NP 问题就是复杂的难以解决的问题”那就太失败了**。P 即“Polynomial”（多项式），**P 问题是指具有“多项式”级复杂性的问题**。换句话说，解决这些问题的时间可以用 N^2 这样的多项式来表示，其中指数可以大于 2，但都是可能在多项式时间内被解决的，这些问题相对比较简单。 但是，现实中大量的问题或者说很多实际的问题似乎都需要指数级算法来解决，即我们还不知道对这类问题有没有多项式算法。这类问题被称为“NP（nondeterministic polynomial，非确定性多项式）”问题。**NP 问题的特点是，它可以快速验证某个解决方案是否正确，但要想迅速找到一个解方案却很难。**可以这么认为，这些问题可以用一个算法在多项式时间内靠猜测来解决，而且该算法必须每次都能猜中。在现实生活中，没有什么能幸运到始终都做出正确的选择，所以这只是理论上的一种设想而已。

可以举个简单的例子来说明 NP 问题，那就是著名的“**旅行推销员问题**”（Traveling Salesman Problem）。一个推销员必须从他居住的城市出发，到其他几个城市去推销，然后再回家。目标是每个城市只到一次（不能重复），而且走过的总距离最短。这个问题实际应用价值很大，其原理经常被应用于设计电路板上孔洞的位置，或者部署船只到墨西哥湾的特定地点采集水样。旅行推销员问题已经被仔细推敲了 50 多年，但还是解决不了 NP 难问题。

现在业界内也经常讨论一个问题：**P 是否等于 NP？即这些难题到底跟那些简单的问题是不是一类？** 尽管很多人都相信未来的某一天可以达到 P=NP，但我还是希望这一天不要太早到来，因为现在一些重要的应用，如加密软件，都是完全建立在某个特定的问题确实极难解决的基础之上的。设想一下，如果某天这些难问题都被攻破了，那我们的各个账号密码、网银岂不是要……当然，如果真有那么一天，也表明计算机领域又有了一个重大的突破，这是值得可贺的。

2. 没有删除只有覆盖

我们知道，磁盘没有真正的删除，**我们所谓的“delete”操作只是把文件占用的块回写到空闲块列表。**但是，这些文件的内容并没有被删除。换句话说，**原始文件占用的每个块中的所有字节都会原封不动地呆在原地。**除非相应的块从空闲块列表中被“除名”并奉送给某个应用程序，否则这些字节不会被新内容覆盖。这意味着什么呢？意味着你认为已经删除的信息实际上还保存在硬盘上。如果有人知道怎么读取它们，仍然可以把它们读出来。**任何可以不通过文件系统而能够逐块读取硬盘的程序，都可以看到那些被“删除”的内容。**

那么如何真正的彻底删除呢？Mac 中的“安全擦除”选项在释放磁盘块之前，会先用随机生成的比特重写其中的内容。但是即使用新信息重写了原有内容，一名训练有素的敌人仍旧可以凭借他掌握的大量资源发现蛛丝马迹。军事级的文件擦除会用随机的 1 和 0 对要释放的块进行多遍重写。更为保险的做法是把整块硬盘放到强磁场里进行消磁。而最保险的做法则是物理上销毁硬盘，这也是保证其中内容彻底销声 匿迹的唯一可靠方法。

也有一些彻底删除文件的软件，**比如我用过的 BCWipe（是看韩国黑客犯罪片“幽灵”时知道的，剧里经常用这个软件删除机密文件），**它提供 Delete with wiping、Wipe free disk space 两种方式来清除你的磁盘文件，还有其它选项，不过这款软件是收费软件，我只试用过一段时间，我本人没啥见不得人的文件，也不需要此类软件，只是当时看完电视好奇试玩了一把。

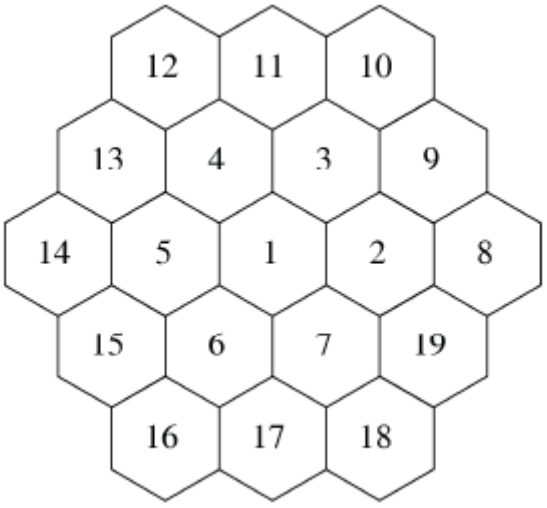
3. 无线网络上网原理

从技术角度讲，**无线网络利用电磁波传送信号。**电磁波是特定频率的电波，其振动频率以 Hz 来衡量（读者可能更熟悉广播电台常用的 MHz 或 GHz，比如北京交通广播电台的频率是 103.9 MHz）。在发送信号之前，首先要通过调制把数据信号附加到载波上。比如，调幅（AM）就是通过改变载波的振幅或强度来传达信息，而调频（FM）的原理则是围绕一个中心值来改变载波的频率。**接收器接收到信号的强度与发射器**

的功率成正比，与到发射器距离的平方成反比。由于存在这种二次方递减的关系，**距离发射器的距离增加一倍，接收器接收到的信号强度就只有原来的四分之一。**无线电波穿越各种物质时强度都会衰减，物质不同衰减程度也不同，比如说金属就会屏蔽任何电波（**突然想起《超验骇客》电影里卡斯特家花园里建的用来屏蔽信号的金属网**）。高频比低频更容易被吸收，二者在其他方面都一样。
无线联网对可以使用的频率范围—频段，以及使用多大的功率发送电波都有严格规定。**频段分配始终都是一个有争议的话题，因为各种需求总会发生冲突。**
无线以太网设备发射的电波频率为 2.4~2.5 GHz，某些 802.11 设备的频率会达到 5 GHz。所有无线设备的频率都局限于这一较窄的范围内，冲突的可能性大大增加。更糟的是，**有些无线电话、医疗设备，甚至微波炉也跟着凑热闹，同样使用这一频段**。有一次作者在使用厨房里那台旧笔记本时无线连接突然断了，后来才发现是用微波炉加热咖啡的缘故。30 秒钟的加热就足以让笔记本断开无线连接。
下面介绍**三种使用最广泛的无线联网技术**。>（1）首先就是蓝牙，蓝牙技术是为近距离临时性连接而发明的，使用与 802.11 相同的 2.4 GHz 频段。蓝牙连接的距离是 1 到 100 米，具体取决于功率大小，数据传输速度为 1～3 Mbit/s。**使用蓝牙技术的设备主要包括无线麦克风、耳机、键盘、鼠标、游戏手柄**，功率相对较低。（2）第二种技术是 RFID（radio-frequency identification），即无线射频识别，**主要用于电子门禁、各种商品的电子标签、自动收费系统、宠物植入芯片，以及护照等身份证明。** RFID 标签其实就是一个小型无线信号收发装置，对外广播身份信息。被动式标签不带电源，通过天线接收到的 RFID 读取器广播的信号来驱动。RFID 系统使用多种不同的频率，比较常见的是 13.56 MHz。**RFID 芯片让秘密监视物体和人的行踪成为可能。** 植入宠物体内的芯片就是一种常见的应用，已经有人建议也给人植入这种芯片了。至于动机嘛，就不好说了。（3）最后一种是 GPS（Global Positioning System，全球定位系统），它是一种重要的单向无线系统，常见于汽车和手机导航系统中。**GPS 卫星会广播精确的时间信息，而 GPS 接收器会根据它从三四颗卫星接收到信号的时间来计算自己在地面的位置**。然而，**GPS 只接收信号不发送信号**。以前曾有一个关于 GPS 的误解，认为它能悄悄地跟踪用户。给大家摘录一段《纽约时报》几年前闹的一个笑话吧：“有些（手机）依靠全球定位系统，也就是 GPS，通过向卫星发送信号来精确地定位用户。”这完全是误解。要 想利用 GPS 跟踪用户，必须得有地面系统（比如手机）转发位置信息。手机与基站之间保持密切通信，因而可以（而且确实会）不断地报告你的位置。只不过有了 GPS 接收器之后，它所报告的信息可以更加精确。

4. 手机为何又称作“蜂窝电话”？

何谓“蜂窝”？因为频段和无线电的覆盖范围都是有限的，因此就要把整个地区划分为蜂窝状的许多小区。可**以将每个这样的小区想象为六边形，然后中央有一个基站，相邻的小区之间通过基站相连。打电话的时候，手机会与最近的基站通信。**当用户移动到另一个小区时，进行中的通话就由原来的小区移交给新小区，但这个切换用户一般觉察不到。 **由于接收功率会随着距离的二次方衰减，所以位于既定频段中的频带在不相邻的小区内可以重用，而不会相互干扰。这就是可以高效利用有限频段的秘密所在。** 大家看下面这幅示意图：



1 号小区中的基站与 2 到 7 号小区中的基站不会使用相同的频率，但可以跟 8 到 19 号小区中的基站使用相同的频率，因为与它们之间的距离足以避免干扰了。“蜂窝”中小区的实际形状要取决很多因素，比如天线的辐射图形。这张图只是一种理想化的结果。 蜂窝手机是常规的电话网络的一部分，只不过连接这个网络不是靠电话线，而是**靠基站发射无线电波。**

手机使用的频段很窄，传输信息的能力有限。因为要使用电池，所以打电话时发射的都是低功率无线电波。而且根据法律规定，为了避免与其他无线设备发生干扰，它们的传输功率也受到限制。 手机在世界的不同地区会使用不同的频带，但一般都在 900 MHz 左右。每个频带被分成多个信道，每次通话时，收发信号各占用一个信道。发送呼叫信号的信道由小区中所有手机共享，在某些系统中这个信道也可以同时用于发送短信和数据。

拨打电话的原理：每个手机都有唯一的识别码（可不是说手机号啊），相当于以太网的地址。启动手机后，它就会广播自己的识别码。距离最近的基站接收到手机信号后，会通过后台系统验证该识别码。随着手机移动，基站实时更新其位置信息，并不断向后台系统报告。如果有人呼叫该手机，后台系统就能通过一直与它保持联系的基站找到它。

手机与基站通信时的信号强度很高。但手机会动态调整功率，在距离基站较近时降低功率。这样不仅可以省电，也可以减少干扰。待机时的耗电量远远比不上一次通话，而这也是为什么待机时间以天为单位，而通话时间以小时为单位的原因。**如果手机所在小区信号较弱或根本没有信号，那么它就会因为拼命查找基站而大量耗电。**

美国使用了**两种完全不同的手机通信技术**： >（1）AT&T 和 T-Mobile 使用 **GSM**（Global System for Mobile Communications，全球移动通信系统），这是一种在欧洲使用非常普遍的系统，它把频带分成很窄的信道，在每个信道内依次附加多路通话。GSM 是世界上应用范围最广的系统。 （2）Verizon 和 Sprint 使用 **CDMA** （Code Division Multiple Access，码分多址），这是一种“扩展频段”技术，它把信号扩展到频带之外，但对不同的通话采用不同的编码模式进行调制。这就意味着，虽然所有手机都使用相同的频带，但大多数情况下通话之间不会发生干扰。 GSM 和 CDMA 都会利用数据压缩来尽可能减少封装信号的比特量。对于通过嘈杂的无线电信道发送数据时无法避免的错误，再添加错误校验来解决问题。

手机带来了一系列难解的非技术问题： >（1）频段的分配。在美国，政府限 制每个频带最多只能有两家公司使用指定频率。**因此频段是非常稀缺的资源，也是无线联网系统的关键资源**。 （2）手机信号发射塔的位置。信号发射塔作为户外建筑算不上漂亮，很多地区为 此拒绝在自己的地界上搭设这种东西。

5. TCP/IP 协议作用

互联网有很多协议，其中最基础的有两个，一是互联网协议（Internet Procotol，IP），**定义了单个包的格式和传输方式**，二是传输控制协议（Transmission Control Protocol，TCP），**定义了 IP 包如何组合成数据流以及如何连接到服务。**两者合起来起就叫 TCP/IP。当然 TCP/IP 协议族不只是包含这两个协议，还包含其它许多的协议。

6. 数据压缩技术

数据压缩技术分为无损压缩和有损压缩。 - 无损压缩，即压缩过程中不丢失信息，解压后得到的数据和原始数据一模一样，**比如霍夫曼编码（Huffman coding），和广泛使用的 zip 程序或 bzip2 程序，都属于无损压缩。**只不过前者按单个字母来压缩，而后者按大块文字，比如 zip 就是根据原始文档的属性选择按单词或词组压缩。 - 有损压缩最常用于处理要给人看或听的内容。比如压缩数码相机拍出来 的照片。人眼分辨不出来非常相近的颜色，所以不必保留实际输入的那么多种颜色，颜色少一点没有任何问题，这样就可以减少编码所用的位数。与此类似，某些难以觉察的细节也可以丢弃，这样处理后的图像尽管没有原始画面那么精密，但眼睛看不出来。细微的亮度变化也是如此。**比如 JPEG 算法和用于压缩电影和电视节目的 MPEG 系列算法都是有损压缩。**

所有**压缩算法的思路**都是减少或去掉那些不能物尽其用的位串，采用的**主要方法包括把出现频率较高的元素编码成短位串、构造频率字典、用数字代替重复内容等**。无损压缩能够完美重现原始数据，有损压缩通过丢弃接收者不需要的信息，来达成数据质量和压缩率的折中。

####7. 如何根据银行卡号判断卡的真伪？ 在“错误检测和校正”小节看到了一个有意思的算法，是 IBM 公司的彼得·卢恩（Peter Luhn）于 1954 年设计的一个校验和（checksum）算法，来检测在实际操作中最常见的两种错误：单个数字错误、由于两个数字写错位置而引起的大多数换位错误。后来这个算法有了很多应用场景，**比如可以检测 16 位长的信用卡和储蓄卡的卡号是否是有效的卡号（这是美国的情况，中国的储蓄卡一般是 19 位，不过算法同样适用）**；10 位或 13 位的 ISBN 书号也采用了类似算法的校验和，用来对付同类错误。

这个算法很简单：**从最右一位数开始向左，把每个数字交替乘 1 或 2，如果结果大于 9 就减 9。如果把各位数的计算结果加起来，最后得到的总和能被 10 整除，那这个卡号就是有效卡号。**

你可以用这个方法测试一下信用卡，以“4417 1234 5678 9112”为例（此卡号取自某银行广告），这个卡号计算的结果是 69，所以不是真卡号；如果把它的最后一个数字换成 3，那就是有效卡号了。**我用该算法测试了自己的银行卡和信用卡，的确可以用来检测卡的真伪，这也算是个小知识吧。**

8. 你能用一句话解释 CGI 是干嘛的吗？

通用网关接口（Common Gateway Interface，CGI），**是 HTTP 协议里一个从客户端（你的浏览器）向服务器传递信息的机制，它能用来传递用户名和密码、查询条件、单选按钮和下拉菜单选项。** CGI 机制在 HTML 里用

…

标签来控制。你可以在

标签里放入文本输入区、按钮等常见界面元素。如果再加上一个“提交”按钮，按下去就会把表单里的数据发送到服务器，服务器用这些数据作为输入，来运行指定的程序。

9. 伟大的 Netscape 公司

Cookie 技术和 Javascript 脚本语言都是 Netscape 公司发明的，网景公司对互联网的贡献真是太大了（还记得网景浏览器吗？），不得不佩服。

10. 病毒和蠕虫的差别

病毒和蠕虫在技术上有个细微差别是：病毒的传播需要人工介入，也就是只有你的操作才能催生它的传播；而蠕虫的传播却不需要你的援手，完全自发进行。

11. 搜索引擎核心竞争力及主要收入来源

搜索引擎的核心竞争力在于**怎么才能迅速从抓取的页面中筛选出匹配度最高的 URL，比如最为匹配的十个页面。谁能把最佳匹配结果排在前头，谁的响应速度快，谁就能赢得用户。**

第一批搜索引擎只会显示一组包含搜索关键词的页面，而随着网页数量激增，搜索结果中就会混入大量无关页面。谷歌的 PageRank 算法**会给每个页面赋予一个权重，权重大小取决于是否有其他页面引用该页面，以及引用该页面的其他页面自身的权重。从理论上讲，权重越大的页面与查询的相关度就越高。**正如布林和佩奇所说：“凭直觉，那些经常被其他网页提及和引用的页面的价值一定更高一些。”当然，要产生高质量的搜索结果绝对不会只靠这一点。搜索引擎公司会不断采取措施来改进自己的结果质量，以期超越对手。

搜索引擎的**收入通常来自广告**。简单来说，搜索引擎的广告模式有两种：>（1）广告客户付钱在网页上显示广告，价格由多少人看过以及什么样的人看到该网页来决定。这种定价模式叫按页面浏览量收费，即按“展示”， 也就是按广告在页面上被展示的次数收费。 （2）另一种模式是按点击收费，即按浏览者点击 广告的次数收费。因此搜索引擎的广告模式。

说到底就是拍卖搜索关键词，且搜索引擎公司都有完备的手段避免虚假点击。

####12. Cookie 如何暴露你在互联网上的行踪？— Cookie 跟踪的原理 只要上网，我们的信息就会被收集，而如果没有我们留下的蛛丝马迹，几乎什么事儿也干不了。使用其他系统时的情况也一样，特别是使用手机的时候，手机网络随时都知道我们在哪里。如果是在户外，支持 GPS 的手机（现在的智能手机几乎都支持）定位用户的误差不超过 10 米，而且随时都会报告你的位置。有些数码相机也带 GPS，可以在照片中编入地理位置信息，这种做法被称为**打地理标签。**

把多个来源的跟踪信息汇总起来，就可以绘制一幅关于个人的活动、喜好、财务状况，以及其他很多方面的信息图。这些信息最起码可以**让广告客户更精准地定位我们，让我们看到乐意点击的广告**。不过，跟踪数据的应用可远不止于此。这些数据还可能被用在很多我们意想不到的地方。比如根据收入把人分成三六九等，在贷款时区别对待，或者更糟糕地，被人冒名顶替，被政府监控，被人图财，甚至害命。

怎么收集我们的浏览信息呢？有些信息会随着浏览器的每一次请求发送，包括你的 IP 地址、正在浏览的页面、浏览器的类型和版本、操作系统，还有语言偏好。 此外，如果服务器的域中有 cookie，那么这些“小甜饼”也会随浏览器请求一块发送。根据 cookie 的规范，只能把这些保存用户信息的小文件发给最初生成它们的域。**那还怎么利用 cookie 跟踪我对其他网站的访问呢？** 要知道答案，**就得明白链接的工作原理：** >每个网页都包含指向其他页面的链接（这正是“超链接”的本义）。我们都知道链接必须由我们主动点击，然后浏览器才会打开或转向新页面。**但图片不需要任何人点击，它会随着页面加载而自动下载。**网页中引用的图片可以来自任何域。于是，在浏览器取得图片时，提供该图片的域就知道我访问过哪个页面了。而且这个域也可以在我的计算机上存放 cookie，并且收到之前访问过的域所产生的 cookie。

以上就是实现跟踪的秘密所在，下面我们再通过例子来解释一下。假设我想买一辆新车，因此访问了toyota.com。我的浏览器因此会下载 60 KB 的 HTML 文件，还有一些 JavaScript，以及 40 张图片。其中一张图片的源代码如下：

```
<img src="http://ad.doubleclick.net/ad/
                    N2724.deduped_spotlight/B1009212;
                    sz=1x1;tag=total_traffic;ord=1?"
                    width=1 height=1 border=0>
```

这个<img>标签会让浏览器从 ad.doubleclick.net 下载一张图片。**这张图片只有 1 像素宽、1 像素高，没有边框，而且很可能是透明的，总之页面上看不见它（称之为网页信标）。**当然，这张图片根本就没想让人看到。当我的浏览器请求它时，DoubleClick 会知道我正在浏览丰田汽车公司网站的某个页面，而且（如果我允许）还会在我的计算机中保存一个 cookie 文件。要是我随后又访问了一个内置 DoubleClick 图片的网站，DoubleClick 就可以绘制一张我的“足迹图”。如果我的“足迹”大都留在汽车网站上，DoubleClick 会把这个信息透露给自己的广告客户。于是乎，我就能看到汽车经销商、购车贷款、修车服务、汽车配件等等各种广告。如果我的“足迹”更多与交通事故或止疼有关，那么就会看到律师和医生投放的广告。 DoubleClick（现为谷歌所有）在拿到用户访问过的站点信息后，会根据这些信息向丰田等广告客户推销广告位。丰田公司继而利用这些信息定向投放广告，而且（可能）会参考包括我的 IP 地址在内的其他信息。

（DoubleClick 不会把这些信息卖给任何人。）随着我访问的页面越来越多，DoubleClick 就可以绘制一幅关于我的更详细的图画，借以推断我的个性、爱好，甚至知道我已经 60 多岁了，是个男的，收入中上，住在新泽西中部，在普林斯顿大学上班。知道我的信息越多，DoubleClick 的广告客户投放的广告就越精准。到了某个时刻，DoubleClick 甚至可以确定那个人就是我，尽管大多数公司都声称不会针对具体的某个人。可是假如我的确在某些网页中填过自己的名字和电子邮件地址，那谁也不敢保证这些信息不会被传播。

这套互联网广告系统设计得极其精密。打开一个网页，这个网页的发布者会立即通知雅虎的 Right Media 或谷歌的 Ad Exchange，说这个网页上有一个空地儿正虚位以待，可以显示广告。同时发过去的还有浏览者的信息（例如，25 到 40 岁之间、单身、住在旧金山，是个技术宅，喜欢泡馆子）。于是，广告客户会为这个广告位而竞价，胜出者的广告将被插入到这个网页中。整个过程不过零点几秒而已。

如何防范？

只要上网，防范基本上是不可能的，不过可以选择性的关闭一些 cookie 跟踪。比如作者使用过 **Firefox 的一个扩展 TACO（Target Advertising Cookie Opt-out，定向广告 Cookie 自愿回避），这个扩展维护着一个 cookie 跟踪站点的列表（目前有大约 150 个名字），**在浏览器中保存着它们的自愿回避 cookie。而我呢，同时对大多数网站都选择关闭 cookie。

许多网站都含有多家公司的跟踪程序。**给大家推荐一个浏览器扩展 Ghostery，通过它可以禁用 JavaScript 跟踪代码，还能查看被阻止的跟踪器。装上它，你会惊讶于互联网上潜伏着多少“间谍”。**仅适用于 Firefox 的 Noscript 插件也有类似的功能。

总结：

一个像素大的图片或者叫网页信标（web beacon，一个很小而且通常是看不到的图片，用于记录某个网页是否已经被下载过了。）都可以用来跟踪你。 用于取得像素图片的 URL 可以包含一个标识码，表示你正在浏览什么网页，还可以包含一个标识符，表示特定的用户。这两个标志就足以跟踪你的浏览活动了。

12. 隐私失控问题

随着社交网站的流行，为了娱乐和与其他人联系，我们自愿放弃了很多个人隐私。**社交网站存在隐私问题是毫无疑义的，因为它们会收集注册用户的大量信息，而且是通过把这些信息卖给广告客户来赚钱。** 作为最大也最成功的社交网站，Facebook 的问题也最明显。Facebook 给第三方提供了 API，以方便编写 Facebook 用户可以使用的应用。但这些 API 有时候会违背公司隐私政策透露一些隐私信息。当然，并非只有 Facebook 一家如此。做地理定位服务的 Foursquare 会在手机上显示用户的位置，能够为找朋友和基于位置的游戏提供方便。在知道潜在用户位置的情况下，定向广告的效果特别好。如果你走到一家餐馆的门口，而手机上恰好是关于这家餐馆的报道，那你很可能就会推门进去体验一下。虽然让朋 友知道你在哪儿没什么问题，但把自己的位置昭告天下则非明智之举。比如，有人做了一个示范性的网站叫“来抢劫我吧”（Please Rob Me），该网站根据 Foursquare 用户在 Twitter 上发表的微博可以推断出他们什么时候不在家，这就为入室行窃提供了机会。 社交网站很容易根据自己的用户构建一个交往群体的“社交图谱”，其中包括被这些用户牵连进来但并未同意甚至毫不知情的人。

数据库

一.基本概念

1. 数据模型

数据库系统的核心和基础是数据模型。一般来说，数据模型是严格定义的一组概念的集合。这些概念精确地描述了系统的静态特征、动态特征和完整性约束条件。因此**数据模型一般由数据结构、数据操作和完整性约束三部分组成**

- **数据结构**：存储在数据库中对象类型的集合，作用是描述数据库组成对象以及对象之间的联系
- **数据操作**：指对数据库中各种对象实例允许执行的操作的集合，包括操作及其相关的操作规则
- **完整性约束**：指在给定的数据模型中，数据及其联系所遵守的一组通用的完整性规则，它能保证数据的正确性和一致性

根据模型应用目的的不同，数据模型分为 2 类：

10. 第一类
    1. 1）**概念模型**：也称为信息模型。它是按用户的观点来对数据和信息建模，主要用于数据库设计
11. 第二类
    1. 2）**逻辑模型**：主要包括层次模型、网状模型、**关系模型**、面向对象模型和对象关系模型等
    2. 3）**物理模型**：是对数据最底层的抽象，它描述数据在系统内部的表示方法和存取方法，在磁盘或磁带上的存储方式和存取方法，是面向计算机系统的

关系模型是目前最重要的一种数据类型。关系数据库系统采用关系模型作为数据的组织方式

12. 关系模型中数据的逻辑结构是一张*二维表*，或者说关系的数据结构就是一张表
13. 关系数据模型的数据操作主要包含*查询*、*插入*、*删除*和*更新*数据
14. 关系模型的完整性约束条件包含三大类：*实体完整性*、*参照完整性*和*用户自定义的完整性*
    1. 关系模型的*实体完整性规则*：若属性（指一个或一组属性）A 是基本关系 R 的主属性，则 A 不能取空值（由此规则可得一直接结论：主键不能为空）
    2. 关系模型的*参照完整性规则*：若属性（或属性组）F 是某基本关系 R 的外键，且它与基本关系 R1 的主键相对应，则对于 R 中，每个 F 上的值或为空值或者等于 R1 中的主键值

2. 主键与外键

15. **候选码**：关系（二维表）中能唯一标识一个元组的属性组
16. **主键**：如果一张表有多个候选码，则选定其中一个为主键
17. **外键**：如果关系模式 R 中的某属性集不是 R 的主键，而是另一个关系 R1 的主键，则该属性集是关系模式 R 的外键。外键表示了两个关系（表）之间的联系。以另一个关系的外键作主键的表被称为主表，具有此外键的表被称为主表的从表
18. **主属性与非主属性**：候选码的诸属性称为主属性。不包含在任何候选码中的属性称为非主属性

3. 事务

事务是指用户定义的一个数据库操作序列，这些操作要么全做要么全不做，是一个不可分割的工作单位

事务具有 4 个特性：**原子性**、**一致性**、**隔离性**、**持续性**。简称为 **ACID** 特性

4. 索引

索引是对数据库中一列或多列的值进行排序的一种结构，使用索引可快速访问数据库表中的特定信息

为表设置索引的好处与坏处：

19. 好处
    1. 通过创建唯一性索引，可以保证数据库表中每一行数据的唯一性
    2. 可以大大加快数据的检索速度（创建索引的主要原因）

3. 在使用分组（group by）和排序（order by）子句进行数据检索时，同样可以显著减少查询中分组和排序的时间
4. 可以加速表和表之间的连接，特别是在实现数据的参照完整性方面特别有意义

20. 坏处
    1. 一是增加了数据库的存储空间
    2. 二是插入和删除数据时要花费较多时间（因为索引也要随之变动）

索引是建立在数据库表中的某些列的上面。在创建索引时，应该考虑在哪些列上可以创建索引，在哪些列上不能创建索引：

21. 一般来说，应该在这些列上创建索引
    1. 1）在经常需要搜索的列上创建索引，可以加快搜索的速度
    2. 2）在作为主键的列上创建索引，强制该列的唯一性和组织表中数据的排列结构
    3. 3）在经常用在连接的列上创建索引，这些列主要是一些外键，可以加快连接的速度
    4. 4）在经常需要根据范围进行搜索的列上创建索引，因为索引已经排序，其指定的范围是连续的
    5. 5）在经常需要排序的列上创建索引，因为索引已经排序，这样查询可以利用索引的排序，加快排序查询时间
    6. 6）在经常使用在 WHERE 子句中的列上创建索引，加快条件的判断速度

22. 一般来说，不应该创建索引的这些列具有下列特点
    1. 1）那些在查询中很少使用的列不应该创建索引。很少使用故而即使创建索引也不会带来很大性能提升。索引又会带来空间和维护上的负担
    2. 2）只有很少数据值的列也不应该创建索引。如性别，结果集的数据行占了表中数据行的很大比例，即需要在表中搜索的数据行的比例很大。增加索引并不能明显加快检索速度
    3. 3）那些定义为 text 和 bit 等数据类型的列不应该创建索引。因为这些列的数据量要么相当大，要么取值很少，不利于使用索引
    4. 4）当修改操作远远大于检索操作时，不应该创建索引。因为修改性能和检索性能互相矛盾。当增加索引时，会提高检索性能，但是会降低修改性能。当减少索引时，会提高修改的性能，降低检索的性能

5. 视图

视图是从一个或几个基本表（或试图）导出的表。与基本表不同，它是一个虚表

数据库中只存放视图的定义，而不存放视图对应的数据，这些数据仍存放在原来的基本表中。所以基本表中的数据发生变化时，视图中查询出的数据也就随之改变了。从这个意义上讲，视图就像一个窗口，透过它可以看到数据库中自己感兴趣的数据及其变化

视图一经定义，就可以和基本表一样被查询、删除

二.SQL 语句

SQL 语句主要包括：

23. 数据定义：create、drop、alter
24. 数据查询：select
25. 数据操作：insert、update、delete
26. 数据控制：grant、revoke

1. 数据定义

1）CREATE TABLE

定义基本表

```
CREATE TABLE <表名> (<列名> <数据类型> [列级完整性约束条件]
                    [, <列名> <数据类型> [列级完整性约束条件]]
                    ...
                    [, <表级完整性约束条件>]);
```

27. primary key (A1,A2,A3,...)：指定主键属性集
28. foreign key (A1,A2,A3,...) references T2：声明表示关系中任意元组在属性(A1,A2,A3,...)上的取值必须对应于 T2 中某元组在主码属性上的取值

数据类型：

29. int：整形。等价于全称 integer

30. smallint：小整数型
31. real，double precision：浮点数与双精度浮点数（精度与机器相关）
32. float(n)：精度至少为 n 位的浮点数
33. char(n)：固定长度的字符串
34. varchar(n)：可变长度的字符串

例：建立一个“学生信息”表 Student：

```
CREATE TABLE Student
(Sno CHAR(9) PRIMARY KEY,
 Sname CHAR(20) UNIQUE,
 Ssex CHAR(2),
 Sage SMALLINT,
 Sdept CHAR(20)
);
```

2）ALTER TABLE

修改基本表

```
ALTER TABLE <表名>
[ADD <新列名> <数据类型> [完整性约束]]
[DROP <完整性约束>]
[MODIFY COLUMN <列名> <数据类型>];
```

35. ADD 子句：增加新列和新的完整性约束
36. DROP 子句：删除指定的完整性约束
37. MODIFY COLUMN 子句：修改原有列的定义，包括列名和数据类型

例子：

```
ALTER TABLE Student ADD S_entrance DATE;      //向 Student 表增加“入学时间”列，其数据类型为日期型
ALTER TABLE Student MODITY COLUMN Sage INT;   //将年龄的数据类型由字符型改为整数
ALTER TABLE Student ADD UNIQUE(Sname);        //增加 Student 表 Sname 必须取唯一值的约束条件
```

3）DROP TABLE

删除基本表

```
DROP TABLE <表名> [RESTRICT | CASCADE];
```

38. RESTRICT：删除是有限制条件的。欲删除的基本表不能被其他表的约束所引用（如：check、foreign key 等约束），不能有视图，不能有触发器，不能有存储过程或函数等。如果存在这些依赖该表的对象，则该表不能被删除
39. CASCADE：删除没有条件限制。在删除该表的同时，相关的依赖对象，例如视图，都将被一起删除

2. 数据查询

1）SELECT

```
SELECT [ALL | DISTINCT] <目标列表达式> [, <目标列表达式>]...
FROM <表名或视图名> [, <表名或视图名>]...
[WHERE <条件表达式>]
[GROUP BY <列名 1> [HAVING <条件表达式>]]
[ORDER BY <列名 2> [ASC | DESC]];
```

40. ALL：显示所有（不去重）
41. DISTINCT：去除重复

整个 SELECT 语句的含义是：根据 WHERE 子句的条件表达式，从 FROM 子句指定的基本表或视图中找出满足条件的元组，再按 SELECT 子句中的目标列表达式，选出元组中的属性值形成结果表

如果有 GROUP BY 子句，则将结果按<列名 1>的值进行分组，该属性列值相等的元组为一个组。通常会在每组中作用聚合函数。如果 GROUP BY 子句带 HAVING 子句，则只有满足指定条件的组才予以输出

如果有 ORDER BY 子句，则结果表还要按<列名 2>的值的升序或降序排列

2）WHERE

WHERE 子句可以使用下列一些条件表达式进行筛选：

42. =：指定属性的值为给定值的
43. IS：如 IS NULL。不能被=代替
44. like：字符串匹配
    1. %：匹配任意子串
    2. _：匹配任意一个字符
45. and、or、not
46. BETWEEN AND（NOT BETWEEN AND）：介于...之间的（不介于...之间的）
47. IN(...)：指定属性的值为 IN 中给出的某个值的

```
SELECT * from Student WHERE Sname='Bill Gates';                    //名字是 Bill Gates
的
SELECT * from Student WHERE Sname like '%Bill%';                   //名字中包含有 Bill
的
SELECT * from Student WHERE Sage BETWEEN 20 AND 23;                //年龄 20~23 的
SELECT Sname , Ssex from Student WHERE Sdept IN('CS','IS','MA');   //CS、IS 或 MA 系的
SELECT * FROM Student WHERE Sage IS NULL;                          //没有年龄信息的
```

3）ORDER BY

对查询结果按一个或多个属性列的升序(ACS)或降低(DESC)排序，默认为升序

例子：

```
SELECT * FROM Student ORDER BY Sage;
SELECT * FROM Student ORDER BY Sdept, Sage desc; //先按专业升序排序，然后同一专业按年龄
降序排序
```

4）LIMIT

可以用于强制 SELECT 返回指定的记录数

接受 1 个或 2 个数字参数。参数必须是一个整数常量：

48. 1 个参数：表示返回最前面的记录行数目
49. 2 个参数：第一个指定第一个返回记录行的偏移量（从 0 开始算），第二个参数指定返回记录行的最大数目

例子：

```
SELECT * FROM Student LIMIT 5, 10;  //返回记录行 6-15
SELECT * FROM Student LIMIT 5;      //返回前 5 个记录行
```

5）聚集函数

聚集函数有以下几种：count、sum、avg、max、min

50. **总数**：select count(*) as totalcount from table1;
51. **求和**：select sum(field1) as sumvalue from table1;
52. **平均**：select avg(field1) as avgvalue from table1;
53. **最大**：select max(field1) as maxvalue from table1;
54. **最小**：select min(field1) as minvalue from table1;

6）GROUP BY

根据一个或多个属性的值对元组分组，值相同的为一组

分组后聚集函数将作用于每一个组，即每一组都有一个函数值

如果分组后还要求按一定的条件对这些分组进行筛选，最终只输出满足指定条件的组，则使用 HAVING 短语指定筛选条件

例子：

```
//按年龄分组，统计每个年龄的人数，并输出(年龄,该年龄的人数)
select Sage, count(*) from Student group by Sage;
//按年龄分组，统计每个年龄的人数，选出人数大于 1 的分组，输出(年龄,该年龄的人数)
select Sage, count(*) from Student group by Sage having count(*) > 1;
```

7）连接查询

一个查询涉及多个表

假设有 2 个表——Student 表和 SC 表（选课表）：

55. **内连接（自然连接）**：当使用内连接时，如果 Student 中某些学生没有选课，则在 SC 中没有相应元组。最终查询结果舍弃了这些学生的信息

56. 外连接：如果想以 Student 表为主体列出每个学生的基本情况及其选课情况。即使某个学生没有选课，依然在查询结果中显示（SC 表的属性上填空值）。就需要使用外连接

例子：

```
//内连接：查询每个学生及其选修课程的情况（没选课的学生不会列出）
SELECT Student.*, SC.*
FROM Student , SC
WHERE Student.Sno=SC.Sno;

//外连接：查询每个学生及其选修课程的情况（没选课的学生也会列出）
SELECT Student.*, SC.*
FROM Student LEFT JOIN SC ON(Student.Sno=SC.Sno);
```

3. 数据操作

1）INSERT

插入元组

```
INSERT
INTO table1(field1,field2...)
VALUES(value1,value2...);
```

如果 INTO 语句没有指定任何属性列名，则新插入的元组必须在每个属性列上均有值

例子：

```
INSERT INTO Student(Sno, Sname, Ssex, Sdept, Sage)
VALUES('201009013', '王明', 'M', 'CS', 23);
```

2）UPDATE

修改(更新)数据

```
UPDATE table1
SET field1=value1, field2=value2
WHERE 范围;
```

功能是修改指定表中满足 WHERE 子句条件的元组。如果省略 WHERE 子句，则表示要修改表中的所有元组

例子：

```
UPDATE Student
SET Sage=22
WHERE Sno='201009013';
```

3）DELETE

删除元素

```
DELETE
FROM table1
WHERE 范围;
```

功能是删除指定表中满足 WHERE 子句条件的元组。如果省略 WHERE 子句，则表示删除表中的所有元组。但表仍存在

例子：

```
DELETE
FROM Student
where Sno='201009013';
```

三.例题

```
1) CREATE TABLE tableQQ (
    ID INTEGER NOT NULL,
    Nickname Varchar(30) NOT NULL
   );
2) select * from tableQQ where Nickname='QQ' order by ID desc;
3) delete from tableQQ where ID=1234;
4) insert into tableQQ values(5555,'1234');
5) drop table tableQQ;
```

```
1) SELECT sc.sno from sc , c
    where sc.cno=c.cno and c.cname='db';
2) SELECT sno, avg(grade) as g from sc
    group by sno order by g desc limit 1;
3) SELECT cno, count(sno) from sc
    where grade > 90 group by cno;
4) SELECT s.sno, s.sname from s, (select sc.sno FROM sc, c where sc.cno=c.cno
          and c.cname in ('math', 'english') group by sno having count
          (DISTINCT c.cno)=2)x
    where s.sno=x.sno;
5) SELECT s.sno, s.sname , avg(sc.grade) as avggrade from s, sc, (select sno
        FROM sc where grade<60 group by sno having count(DISTINCT cno)>=2)x
    where s.sno=x.sno and sc.sno=x.sno group by s.sno;
6) SELECT s.sname from s,
  (select sno, grade from sc where cno in (select cno from c where
       cname='math'))A,
  (select sno, grade from sc where cno in (select cno from c where
       cname='english'))B
   where s.sno=A.sno and s.sno=B.sno and A.grade>B.grade;
```

图可以用 G=(V,E)来表示，每个图都包括一个顶点集合 V 和一个边集合 E，顶点总数记为|V|，边总数记为|E|

稀疏图：边数较少的图
密集图：边数较多的图
完全图：包含所有可能边的图
带权图：边上标有权的图
邻接点：一条边所连的两个顶点
简单路径：路径上不包含重复顶点的图
回路：将某个顶点连接到本身，且长度大于等于 3 的路径
无环图：不带回路的图

图的表示

图有两种常用的表示方法：

1. **邻接矩阵**
2. **邻接表**

1. 邻接矩阵

使用一个二维矩阵来表示图:

1. (i,j)=1，表示顶点 i 到顶点 j 之间有一条边（**非带权图**）
2. (i,j)=n，表示顶点 i 到顶点 j 之间有一条权重为 n 的边（**带权图**）

使用邻接矩阵的空间代价总是 O(|V|^2)

2. 邻接表

邻接表使用一个顶点指针数组来表示：

1. 数组的元素 i 表示顶点 i 的指针，它是一个链表的头结点
2. 链表其余的顶点表示与顶点 i 之间存在边的顶点

邻接表的空间代价与图中边的数目和顶点的数目均有关系。每个顶点要占据一个数组元素的位置，且每条边必须出现在其中某个顶点的边链表中

图的遍历

DFS（深度优先遍历）

DFS 会递归地访问它的所有未被访问的相邻顶点：

- 先访问顶点 v，把所有与 v 相关联的边存入栈中；
- 弹出栈顶元素，栈顶元素代表的边所关联的另一个顶点就是要访问的下一个元素 k；
- 对 k 重复对 v 的操作；
- 重复，直至栈中所有元素都被处理完毕

DFS 的执行过程将产生一棵 **DFS(深度优先搜索)树**：

整个 DFS 的过程如下：

相关题目：* Leetcode：695.Max Area of Island

BFS（广度优先遍历）

使用一个队列。对于每个顶点，在访问其它顶点前，检查当前节点所有邻接点。和树的广度优先遍历类似

BFS 执行过程将产生一棵 **BFS(广度优先搜索)树**：

整个 BFS 的过程如下：

拓扑排序

(DAG)有向无环图可以描述这样一种场景：有一组任务，任务的执行顺序之间具有依赖性，一些任务必须在另一些任务完成之后才开始执行，如下图：

在这种场景下，任务之间的依赖关系不能出现环，否则任何一个都无法开始执行。**将一个(DAG)有向无环图中所有顶点在不违反先决条件规定的基础上排成线性序列的过程就是拓扑排序**

有 2 种方法实现拓扑排序：

1. **基于 DFS 的方法(递归)**：当访问某个顶点时，不对这个顶点进行任何处理。当递归返回到这个顶点时，打印这个顶点。这将产生一个逆序的拓扑排序。对其进行一次反序操作就可以得到一个拓扑排序的序列。序列从哪个顶点开始并不重要，只要所有顶点最终都能被访问到
2. **基于 BFS 的方法(迭代)**：首先访问所有的边，计算指向每个顶点的边数(即计算每个顶点的先决条件数目)。将所有没有先决条件的顶点放入队列，然后开始处理队列。当从顶点中删除一个顶点时，把它打印出来，同时将其所有相邻顶点的先决条件计数减 1。当某个相邻顶点的计数为 0 时，就将其放入队列。如果还有顶点未被打印，而队列已经为空，则图中必然包含回路

最小生成树

最小生成树(MST)是一个包括图 G 所有顶点及其部分边的图，包括的边是 G 的子集，满足下列条件：

1. 这个子集中所有边的权之和为所有子集中最小的
2. 子集中的边能保证图是连通的

下面是一个最小生成树的例子：

如果上图中使用边(D,F)代替(C,F)，可得到另一个最小生成树，即最小生成树可能有多个

最小生成树适合解决如下问题：怎样使连接电路板上一系列接头所需焊接的线路最短，或者怎样使得在几个城市之间建立电话网所需的线路最短

Prim 算法

原理：从图中任意一个节点 N 开始，初始化，MST 为 N。选出与 N 相关联的边中权最小的一条边，设其连接顶点 N 与另一个顶点 M。把顶点 M 和边(N,M)加入 MST 中。接下来，选出与顶点 N 或顶点 M 相关联的边中权最小的一条边，设其连接另一个新顶点，将这条边和新顶点添加到 MST 中。反复进行这样的处理，每一步都选出一条边来扩展 MST，这条边是连接当前已在 MST 中的某个顶点与一个不在 MST 中的顶点的所有边中代价最小的

证明 Prim 算法能生成 MST：反证法。设图 G=(V,E)不能通过 Prim 算法生成最小生成树。根据 Prim 算法中各顶点加入 MST 的顺序，依次定义图 G 中各顶点为 v0,v1,…,v(n-1)。令 ei 代表边(vx,vi)，其中 x=1，令 ej 为 Prim 算法添加的序号最小的那条(第一条)出现以下情况的边：加入 ej 后的边集不能被扩展而构成图 G 的一个 MST。换句话说，ej 是 Prim 算法发生错误的第一条边。设 T 为“真正的”MST。令 vp 为边 ej 所关联的顶点，即 ej=（vp,vj）

因为 T 是一个树结构，所以 T 中将存在一条连接 vp 和 vj 的路径，且此路径中一定存在某条边 e’连接 vu 和 vw，其中 u<j，w>=j。因为 ej 不是 T 的一部分，所以把 ej 加入到 T 会构成一个回路。又因为 Prim 算法不能生成一个 MST，所以 e’的权比 ej 的权更小。这种情况如下图。但是，Prim 算法应选择可能的最小权边，它一定会选择 e’，而不是 ej。这与 Prim 算法选错了边 ej 的假设相矛盾。因此，Prim 算法一定是正确的

信号

1. 信号

1）2 种信号处理方式

这种方式已被废弃

1. 主要原因是在 UNIX 实现中，收到信号之后，会重置回默认的信号处理行为
2. 同时，该行为是不跨平台的
3. 参数
    1. signo：信号
    2. func：信号处理函数（捕捉函数）
        1. SIG_IGN：忽略此信号
        2. SIG_DFL：默认处理动作
            1. 默认处理通常是收到信号后终止进程
            2. SIGCHLD 和 SIGURG(带外数据到达时发送)的默认处理是忽略信号

推荐使用 **sigaction 函数**

4. 参数
    1. signo：信号
    2. act：非空则表示修改信号的处理配置
    3. oact：非空则表示关心信号旧的处理配置

信号处理配置使用一个 sigaction 结构：

```
struct sigaction {
    void (*sa_handler)(int);
    sigset_t sa_mask;
    int sa_flags;
    void (*sa_sigaction)(int,siginfo_t *,void *);
}
```

5. sa_handler：信号处理函数
    1. SIG_IGN：忽略此信号
    2. SIG_DFL：默认处理动作
        1. 默认处理通常是收到信号后终止进程
        2. SIGCHLD 和 SIGURG(带外数据到达时发送)的默认处理是忽略信号
6. sa_mask：信号集，在调用信号处理函数之前，这一信号集要加到进程的信号屏蔽字中。仅当从信号处理函数返回时再将进程的信号屏蔽字恢复为原先值。这样，在调用信号处理函数时就能阻塞某些信号。在信号处理程序被调用时，操作系统建立的新信号屏蔽字包括正被传递的信号。因此保证了在处

理一个给定的信号时，如果这种信号再次发生，那么它会被阻塞到对前一信号的处理结束为止。如果一个信号在被阻塞期间产生了一次或多次，那么该信号被解阻塞之后通常只递交一次，也就是说 UNIX 信号默认是不排队的

7. `sa_flags`：信号标志（详见下图）
8. `sa_sigaction`：替代的信号处理函数。在当使用了 `SA_SIGINFO` 标志时，使用该信号处理函数

2）常见信号

信号	描述
SIGHUP	挂起进程，当父进程退出时，所有子进程都会收到 SIGHUP 信号
SIGINT	终止进程（CTRL + C）
SIGQUIT	终止前台进程组并产生一个 core 文件（CTRL + ）
SIGCHLD	子进程终止时给父进程发送的信号，如果父进程没有捕获代码，则这个信号被忽略
SIGTERM	关机时，init 进程向系统中进程发出的信号，提示进程即将关机，进程可以捕获然后执行一些清理
SIGKILL(不能捕获或忽略)	杀掉进程
SIGSTOP(不能捕获或忽略)	停止进程，但不终止进程
SIGTSTP	停止或暂停进程，但不终止进程（CTRL + Z）
SIGCONT	继续运行停止的进程

1. `Ctrl + C` 组合键会发送 `SIGINT` 信号，终止 shell 中当前运行的进程
2. `Ctrl + Z` 组合键会发送 `SIGTSTP` 信号，停止 shell 中运行的进程，停止的进程会继续保留在保存在内存中，并能够从上次停止的位置继续运行
    1. 使用 `jobs` 命令查看后台任务，通过 `fg %jobnumber` 将用 `CTRL + Z` 暂停的后台任务放到前台继续执行

2. 线程与信号

- 信号处理函数是进程层面的概念，或者说是线程组层面的概念，线程组内所有线程共享对信号的处理函数
- 对于发送给进程的信号，内核会任选一个线程来执行信号处理函数，执行完后，会将其从挂起信号队列中去除，其他线程不会对一个信号重复响应
- 可以针对进程中的某个线程发送信号，那么只有该线程能响应，执行相应的信号处理函数
- 信号掩码是线程层面的概念，信号处理函数在线程组内是统一的，但是信号掩码是各自独立可配置的，各个线程独立配置自己要阻止或放行的信号集合
- 挂起信号（内核已经收到，但尚未递送给线程处理的信号）既是针对进程的，又是针对线程的。内核维护两个挂起信号队列，一个是线程共享的挂起信号队列，一个是线程特有的挂起信号