目录

C++

1. 引用和指针的区别?

1) 指针是一个实体，需要分配内存空间。引用只是变量的别名，不需要分配内存空间。

2) 引用在定义的时候必须进行初始化，并且不能够改变。指针在定义的时候不一定要初始化，并且指向的空间可变。（注：不能有引用的值不能为 NULL）

3) 有多级指针，但是没有多级引用，只能有一级引用。

4) 指针和引用的自增运算结果不一样。（指针是指向下一个空间，引用时引用的变量值加 1）

5) sizeof 引用得到的是所指向的变量（对象）的大小，而 sizeof 指针得到的是指针本身的大小。

6) 引用访问一个变量是直接访问，而指针访问一个变量是间接访问。

7) 使用指针前最好做类型检查，防止野指针的出现；

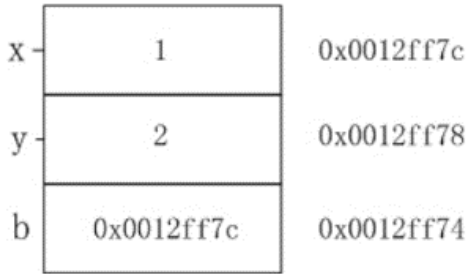
8) 引用底层是通过指针实现的；

9) 作为参数时也不同，传指针的实质是传值，传递的值是指针的地址；传引用的实质是传地址，传递的是变量的地址。

2. 从汇编层去解释一下引用

```
9:          int x = 1;
00401048    mov         dword ptr [ebp-4],1
10:         int &b = x;
0040104F    lea         eax,[ebp-4]
00401052    mov         dword ptr [ebp-8],eax
```

x 的地址为 ebp-4，b 的地址为 ebp-8，因为栈内的变量内存是从高往低进行分配的。所以 b 的地址比 x 的低。lea eax,[ebp-4] 这条语句将 x 的地址 ebp-4 放入 eax 寄存器 mov dword ptr [ebp-8],eax 这条语句将 eax 的值放入 b 的地址 ebp-8 中上面两条汇编的作用即：将 x 的地址存入变量 b 中，这不和将某个变量的地址存入指针变量是一样的吗？所以从汇编层次来看，的确引用是通过指针来实现的。



3. C++中的指针参数传递和引用参数传递

1）指针参数传递本质上是值传递，它所传递的是一个地址值。值传递过程中，被调函数的形式参数作为被调函数的局部变量处理，会在栈中开辟内存空间以存放由主调函数传递进来的实参值，从而形成了实参的一个副本（替身）。值传递的特点是，被调函数对形式参数的任何操作都是作为局部变量进行的，不会影响主调函数的实参变量的值（形参指针变了，实参指针不会变）。

2）引用参数传递过程中，被调函数的形式参数也作为局部变量在栈中开辟了内存空间，但是这时存放的是由主调函数放进来的实参变量的地址。被调函数对形参（本体）的任何操作都被处理成间接寻址，即通过栈中存放的地址访问主调函数中的实参变量（根据别名找到主调函数中的本体）。因此，被调函数对形参的任何操作都会影响主调函数中的实参变量。

3）引用传递和指针传递是不同的，虽然他们都是在被调函数栈空间上的一个局部变量，但是任何对于引用参数的处理都会通过一个间接寻址的方式操作到主调函数中的相关变量。而对于指针传递的参数，如果改变被调函数中的指针地址，它将应用不到主调函数的相关变量。如果想通过指针参数传递来改变主调函数中的相关变量（地址），那就得使用指向指针的指针或者指针引用。

4）从编译的角度来讲，程序在编译时分别将指针和引用添加到符号表上，符号表中记录的是变量名及变量所对应地址。指针变量在符号表上对应的地址值为指针变量的地址值，而引用在符号表上对应的地址值为引用对象的地址值（与实参名字不同，地址相同）。符号表生成之后就不会再改，因此指针可以改变其指向的对象（指针变量中的值可以改），而引用对象则不能修改。

4. 形参与实参的区别?

1) 形参变量只有在被调用时才分配内存单元，在调用结束时， 即刻释放所分配的内存单元。因此，形参只有在函数内部有效。 函数调用结束返回主调函数后则不能再使用该形参变量。

2) 实参可以是常量、变量、表达式、函数等， 无论实参是何种类型的量，在进行函数调用时，它们都必须具有确定的值， 以便把这些值传送给形参。 因此应预先用赋值，输入等办法使实参获得确定值，会产生一个临时变量。

3) 实参和形参在数量上，类型上，顺序上应严格一致， 否则会发生“类型不匹配”的错误。

4) 函数调用中发生的数据传送是单向的。 即只能把实参的值传送给形参，而不能把形参的值反向地传送给实参。 因此在函数调用过程中，形参的值发生改变，而实参中的值不会变化。

5) 当形参和实参不是指针类型时，在该函数运行时，形参和实参是不同的变量，他们在内存中位于不同的位置，形参将实参的内容复制一份，在该函数运行结束的时候形参被释放，而实参内容不会改变。

1) 值传递：有一个形参向函数所属的栈拷贝数据的过程，如果值传递的对象是类对象或是大的结构体对象，将耗费一定的时间和空间。（传值）

2) 指针传递：同样有一个形参向函数所属的栈拷贝数据的过程，但拷贝的数据是一个固定为 4 字节的地址。（传值，传递的是地址值）

3) 引用传递：同样有上述的数据拷贝过程，但其是针对地址的，相当于为该数据所在的地址起了一个别名。（传地址）

4) 效率上讲，指针传递和引用传递比值传递效率高。一般主张使用引用传递，代码逻辑上更加紧凑、清晰。

5. static 的用法和作用?

1. 先来介绍它的第一条也是最重要的一条：隐藏。（static 函数，static 变量均可）

当同时编译多个文件时，所有未加 static 前缀的全局变量和函数都具有全局可见性。

2. static 的第二个作用是保持变量内容的持久。（static 变量中的记忆功能和全局生存期）存储在静态数据区的变量会在程序刚开始运行时就完成初始化，也是唯一的一次初始化。共有两种变量存储在静态存储区：全局变量和 static 变量，只不过和全局变量比起来，static 可以控制变量的可见范围，说到底 static 还是用来隐藏的。
3. static 的第三个作用是默认初始化为 0（static 变量）

其实全局变量也具备这一属性，因为全局变量也存储在静态数据区。在静态数据区，内存中所有的字节默认值都是 0x00，某些时候这一特点可以减少程序员的工作量。

4. static 的第四个作用：C++中的类成员声明 static

1) 函数体内 static 变量的作用范围为该函数体，不同于 auto 变量，该变量的内存只被分配一次，因此其值在下次调用时仍维持上次的值；

2) 在模块内的 static 全局变量可以被模块内所用函数访问，但不能被模块外其它函数访问；

3) 在模块内的 static 函数只可被这一模块内的其它函数调用，这个函数的使用范围被限制在声明它的模块内；

4) 在类中的 static 成员变量属于整个类所拥有，对类的所有对象只有一份拷贝；

5) 在类中的 static 成员函数属于整个类所拥有，这个函数不接收 this 指针，因而只能访问类的 static 成员变量。

类内：

6) static 类对象必须要在类外进行初始化，static 修饰的变量先于对象存在，所以 static 修饰的变量要在类外初始化；

7) 由于 static 修饰的类成员属于类，不属于对象，因此 static 类成员函数是没有 this 指针的，this 指针是指向本对象的指针。正因为没有 this 指针，所以 static 类成员函数不能访问非 static 的类成员，只能访问 static 修饰的类成员；

8) static 成员函数不能被 virtual 修饰，static 成员不属于任何对象或实例，所以加上 virtual 没有任何实际意义；静态成员函数没有 this 指针，虚函数的实现是为每一个对象分配一个 vptr 指针，而 vptr 是通过 this 指针调用的，所以不能为 virtual；虚函数的调用关系，this->vptr->ctable->virtual function

6. 静态变量什么时候初始化

1) 初始化只有一次，但是可以多次赋值，在主程序之前，编译器已经为其分配好了内存。

2) 静态局部变量和全局变量一样，数据都存放在全局区域，所以在主程序之前，编译器已经为其分配好了内存，但在 C 和 C++中静态局部变量的初始化节点又有点不太一样。在 C 中，初始化发生在代码执行之前，编译阶段分配好内存之后，就会进行初始化，所以我们看到在 C 语言中无法使用变量对静态局部变量进行初始化，在程序运行结束，变量所处的全局内存会被全部回收。

3) 而在 C++中，初始化时在执行相关代码时才会进行初始化，主要是由于 C++引入对象后，要进行初始化必须执行相应构造函数和析构函数，在构造函数或析构函数中经常会需要进行某些程序中需要进行的特定操作，并非简单地分配内存。所以 C++标准定为全局或静态对象是有首次用到时才会进行构造，并通过 atexit()来管理。在程序结束，按照构造顺序反方向进行逐个析构。所以在 C++中是可以使用变量对静态局部变量进行初始化的。

7. const?

1) 阻止一个变量被改变，可以使用 const 关键字。在定义该 const 变量时，通常需要对它进行初始化，因为以后就没有机会再去改变它了；
2) 对指针来说，可以指定指针本身为 const，也可以指定指针所指的数据为 const，或二者同时指定为 const；
3) 在一个函数声明中，const 可以修饰形参，表明它是一个输入参数，在函数内部不能改变其值；
4) 对于类的成员函数，若指定其为 const 类型，则表明其是一个常函数，不能修改类的成员变量，类的常对象只能访问类的常成员函数；
5) 对于类的成员函数，有时候必须指定其返回值为 const 类型，以使得其返回值不为“左值”。
6) const 成员函数可以访问非 const 对象的非 const 数据成员、const 数据成员，也可以访问 const 对象内的所有数据成员；
7) 非 const 成员函数可以访问非 const 对象的非 const 数据成员、const 数据成员，但不可以访问 const 对象的任意数据成员；
8) 一个没有明确声明为 const 的成员函数被看作是将要修改对象中数据成员的函数，而且编译器不允许它为一个 const 对象所调用。因此 const 对象只能调用 const 成员函数。
9) const 类型变量可以通过类型转换符 const_cast 将 const 类型转换为非 const 类型；
10) const 类型变量必须定义的时候进行初始化，因此也导致如果类的成员变量有 const 类型的变量，那么该变量必须在类的初始化列表中进行初始化；
11) 对于函数值传递的情况，因为参数传递是通过复制实参创建一个临时变量传递进函数的，函数内只能改变临时变量，但无法改变实参。则这个时候无论加不加 const 对实参不会产生任何影响。但是在引用或指针传递函数调用中，因为传进去的是一个引用或指针，这样函数内部可以改变引用或指针所指向的变量，这时 const 才是实实在在地保护了实参所指向的变量。因为在编译阶段编译器对调用函数的选择是

根据实参进行的，所以，只有引用传递和指针传递可以用是否加 const 来重载。一个拥有顶层 const 的形参无法和另一个没有顶层 const 的形参区分开来。

8. const 成员函数的理解和应用?

① const Stock & Stock::topval (②const Stock & s) ③const

①处 const：确保返回的 Stock 对象在以后的使用中不能被修改

②处 const：确保此方法不修改传递的参数 S

③处 const：保证此方法不修改调用它的对象，const 对象只能调用 const 成员函数,不能调用非 const 函数

9. 指针和 const 的用法

1) 当 const 修饰指针时，由于 const 的位置不同，它的修饰对象会有所不同。
2) int *const p2 中 const 修饰 p2 的值,所以理解为 p2 的值不可以改变，即 p2 只能指向固定的一个变量地址，但可以通过*p2 读写这个变量的值。顶层指针表示指针本身是一个常量
3) int const *p1 或者 const int *p1 两种情况中 const 修饰*p1，所以理解为*p1 的值不可以改变，即不可以给*p1 赋值改变 p1 指向变量的值，但可以通过给 p 赋值不同的地址改变这个指针指向。底层指针表示指针所指向的变量是一个常量。
4) int const *const p;

10. mutable

1) 如果需要在 const 成员方法中修改一个成员变量的值，那么需要将这个成员变量修饰为 mutable。即用 mutable 修饰的成员变量不受 const 成员方法的限制;
2) 可以认为 mutable 的变量是类的辅助状态，但是只是起到类的一些方面表述的功能，修改他的内容我们可以认为对象的状态本身并没有改变的。实际上由于 const_cast 的存在，这个概念很多时候用处不是很到了。

11. extern 用法?

1) extern 修饰变量的声明
   如果文件 a.c 需要引用 b.c 中变量 int v，就可以在 a.c 中声明 extern int v，然后就可以引用变量 v。
2) extern 修饰函数的声明

如果文件 a.c 需要引用 b.c 中的函数，比如在 b.c 中原型是 int fun(int mu)，那么就可以在 a.c 中声明 extern int fun（int mu），然后就能使用 fun 来做任何事情。就像变量的声明一样，extern int fun（int mu）可以放在 a.c 中任何地方，而不一定非要放在 a.c 的文件作用域的范围中。

3) extern 修饰符可用于指示 C 或者 C++函数的调用规范。

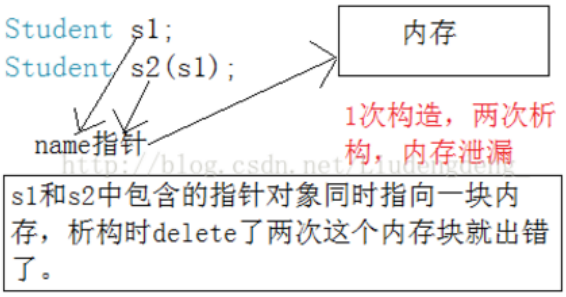
比如在 C++中调用 C 库函数，就需要在 C++程序中用 extern “C”声明要引用的函数。这是给链接器用的，告诉链接器在链接的时候用 C 函数规范来链接。主要原因是 C++和 C 程序编译完成后在目标代码中命名规则不同。

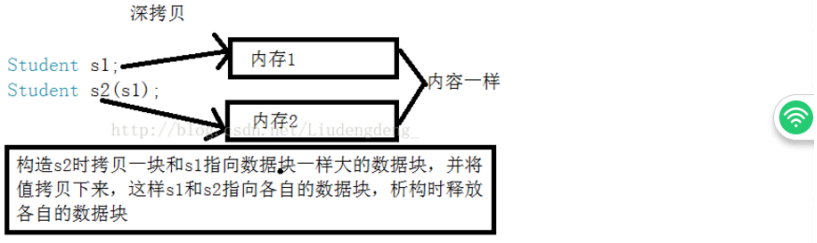
12. int 转字符串字符串转 intstrcat,strcpy,strncpy,memset,memcpy 的内部实现？

c++11 标准增加了全局函数 std::to_string
可以使用 std::stoi/stol/stoll 等等函数
strcpy 拥有返回值，有时候函数原本不需要返回值，但为了增加灵活性如支持链式表达，

13. 深拷贝与浅拷贝？

1) 浅复制 ——只是拷贝了基本类型的数据，而引用类型数据，复制后也是会发生引用，我们把这种拷贝叫做“（浅复制）浅拷贝”，换句话说，浅复制仅仅是指向被复制的内存地址，如果原地址中对象被改变了，那么浅复制出来的对象也会相应改变。
深复制 ——在计算机中开辟了一块新的内存地址用于存放复制的对象。





综上所述，浅拷贝是只对指针进行拷贝，两个指针指向同一个内存块，深拷贝是对指针和指针指向的内容都进行拷贝，拷贝后的指针是指向不同内的指针。

2) 在某些状况下，类内成员变量需要动态开辟堆内存，如果实行位拷贝，也就是把对象里的值完全复制给另一个对象，如 A=B。这时，如果 B 中有一个成员变量指针已经申请了内存，那 A 中的那个成员变量也指向同一块内存。这就出现了问题：当 B 把内存释放了（如：析构），这时 A 内的指针就是野指针了，出现运行错误。

14. C++模板是什么，底层怎么实现的?

1) 编译器并不是把函数模板处理成能够处理任意类的函数；编译器从函数模板通过具体类型产生不同的函数；编译器会对函数模板进行两次编译：在声明的地方对模板代码本身进行编译，在调用的地方对参数替换后的代码进行编译。
2) 这是因为函数模板要被实例化后才能成为真正的函数，在使用函数模板的源文件中包含函数模板的头文件，如果该头文件中只有声明，没有定义，那编译器无法实例化该模板，最终导致链接错误。

15. C 语言 struct 和 C++struct 区别

1) C 语言中：struct 是用户自定义数据类型（UDT）；C++中 struct 是抽象数据类型（ADT），支持成员函数的定义，（C++中的 struct 能继承，能实现多态）。
2) C 中 struct 是没有权限的设置的，且 struct 中只能是一些变量的集合体，可以封装数据却不可以隐藏数据，而且成员不可以是函数。
3) C++中，struct 的成员默认访问说明符为 public（为了与 C 兼容），class 中的默认访问限定符为 private，struct 增加了访问权限，且可以和类一样有成员函数。
4) struct 作为类的一种特例是用来自定义数据结构的。一个结构标记声明后，在 C 中必须在结构标记前加上 struct，才能做结构类型名

16. 虚函数可以声明为 inline 吗?

1) 虚函数用于实现运行时的多态，或者称为晚绑定或动态绑定。而内联函数用于提高效率。内联函数的原理是，在编译期间，对调用内联函数的地方的代码替换成函数代码。内联函数对于程序中需要频繁使用和调用的小函数非常有用。
2) 虚函数要求在运行时进行类型确定，而内敛函数要求在编译期完成相关的函数替换；

17. 类成员初始化方式？构造函数的执行顺序 ？为什么用成员初始化列表会快一些?

1) 赋值初始化，通过在函数体内进行赋值初始化；列表初始化，在冒号后使用初始化列表进行初始化。

   这两种方式的主要区别在于：

对于在函数体中初始化，是在所有的数据成员被分配内存空间后才进行的。

列表初始化是给数据成员分配内存空间时就进行初始化，就是说分配一个数据成员只要冒号后有此数据成员的赋值表达式(此表达式必须是括号赋值表达式)，那么分配了内存空间后在进入函数体之前给数据成员赋值，就是说初始化这个数据成员此时函数体还未执行。

2) 一个派生类构造函数的执行顺序如下：

① 虚拟基类的构造函数（多个虚拟基类则按照继承的顺序执行构造函数）。

② 基类的构造函数（多个普通基类也按照继承的顺序执行构造函数）。

③ 类类型的成员对象的构造函数（按照初始化顺序）

④ 派生类自己的构造函数。

3) 方法一是在构造函数当中做赋值的操作，而方法二是做纯粹的初始化操作。我们都知道，C++的赋值操作是会产生临时对象的。临时对象的出现会降低程序的效率。

18. 成员列表初始化?

1) 必须使用成员初始化的四种情况

① 当初始化一个引用成员时；

② 当初始化一个常量成员时；

③ 当调用一个基类的构造函数，而它拥有一组参数时；

④ 当调用一个成员类的构造函数，而它拥有一组参数时；

2) 成员初始化列表做了什么

① 编译器会一一操作初始化列表，以适当的顺序在构造函数之内安插初始化操作，并且在任何显示用户代码之前；

② list 中的项目顺序是由类中的成员声明顺序决定的，不是由初始化列表的顺序决定的；

19. 构造函数为什么不能为虚函数？析构函数为什么要虚函数？

1. 从存储空间角度，虚函数相应一个指向 vtable 虚函数表的指针，这大家都知道，但是这个指向 vtable 的指针事实上是存储在对象的内存空间的。问题出来了，假设构造函数是虚的，就须要通过 vtable 来调用，但是对象还没有实例化，也就是内存空间还没有，怎么找 vtable 呢？所以构造函数不能是虚函数。
2. 从使用角度，虚函数主要用于在信息不全的情况下，能使重载的函数得到相应的调用。构造函数本身就是要初始化实例，那使用虚函数也没有实际意义呀。所以构造函数没有必要是虚函数。虚函数的作用在于通过父类的指针或者引用来调用它的时候可以变成调用子类的那个成员函数。而构造函数是在创建对象时自己主动调用的，不可能通过父类的指针或者引用去调用，因此也就规定构造函数不能是虚函数。
3. 构造函数不须要是虚函数，也不同意是虚函数，由于创建一个对象时我们总是要明白指定对象的类型，虽然我们可能通过实验室的基类的指针或引用去訪问它但析构却不一定，我们往往通过基类的指针来销毁对象。这时候假设析构函数不是虚函数，就不能正确识别对象类型从而不能正确调用析构函数。
4. 从实现上看，vbtl 在构造函数调用后才建立，因而构造函数不可能成为虚函数从实际含义上看，在调用构造函数时还不能确定对象的真实类型（由于子类会调父类的构造函数）；并且构造函数的作用是提供初始化，在对象生命期仅仅运行一次，不是对象的动态行为，也没有必要成为虚函数。
5. 当一个构造函数被调用时，它做的首要的事情之中的一个是初始化它的 VPTR。因此，它仅仅能知道它是“当前”类的，而全然忽视这个对象后面是否还有继承者。当编译器为这个构造函数产生代码时，它是为这个类的构造函数产生代码——既不是为基类，也不是为它的派生类（由于类不知道谁继承它）。所以它使用的 VPTR 必须是对于这个类的 VTABLE。并且，仅仅要它是最后的构造函数调用，那么在这个对象的生命期内，VPTR 将保持被初始化为指向这个 VTABLE，但假设接着另一个更晚派生的构造函数被调用，这个构造函数又将设置 VPTR 指向它的 VTABLE，等.直到最后的构造函数结束。VPTR 的状态是由被最后调用的构造函数确定的。这就是为什么构造函数调用是从基类到更加派生类顺序的还有一个理由。可是，当这一系列构造函数调用正发生时，每一个构造函数都已经设置 VPTR 指向它自己的 VTABLE。假设函数调用使用虚机制，它将仅仅产生通过它自己的 VTABLE 的调用，而不是最后的 VTABLE（全部构造函数被调用后才会有最后的 VTABLE）。

因为构造函数本来就是为了明确初始化对象成员才产生的，然而 virtual function 主要是为了再不完全了解细节的情况下也能正确处理对象。另外，virtual 函数是在不同类型的对象产生不同的动作，现在对象还没有产生，如何使用 virtual 函数来完成你想完成的动作。

直接的讲，C++中基类采用 virtual 虚析构函数是为了防止内存泄漏。具体地说，如果派生类中申请了内存空间，并在其析构函数中对这些内存空间进行释放。假设基类中采用的是非虚析构函数，当删除基类指针指向的派生类对象时就不会触发动态绑定，因而只会调用基类的析构函数，而不会调用派生类的析构函数。那么在这种情况下，派生类中申请的空间就得不到释放从而产生内存泄漏。所以，为了防止这种情况的发生，C++中基类的析构函数应采用 virtual 虚析构函数。

20. 析构函数的作用，如何起作用？

1) 构造函数只是起初始化值的作用，但实例化一个对象的时候，可以通过实例去传递参数，从主函数传递到其他的函数里面，这样就使其他的函数里面有值了。规则，只要你一实例化对象，系统自动回调用一个构造函数，就是你不写，编译器也自动调用一次。
2) 析构函数与构造函数的作用相反，用于撤销对象的一些特殊任务处理，可以是释放对象分配的内存空间；特点：析构函数与构造函数同名，但该函数前面加~。 析构函数没有参数，也没有返回值，而且不能重载，在一个类中只能有一个析构函数。 当撤销对象时，编译器也会自动调用析构函数。 每一个类必须有一个析构函数，用户可以自定义析构函数，也可以是编译器自动生成默认的析构函数。一般析构函数定义为类的公有成员。

21. 构造函数和析构函数可以调用虚函数吗，为什么

1) 在 C++中，提倡不在构造函数和析构函数中调用虚函数；
2) 构造函数和析构函数调用虚函数时都不使用动态联编，如果在构造函数或析构函数中调用虚函数，则运行的是为构造函数或析构函数自身类型定义的版本；
3) 因为父类对象会在子类之前进行构造，此时子类部分的数据成员还未初始化，因此调用子类的虚函数时不安全的，故而 C++不会进行动态联编；
4) 析构函数是用来销毁一个对象的，在销毁一个对象时，先调用子类的析构函数，然后再调用基类的析构函数。所以在调用基类的析构函数时，派生类对象的数据成员已经销毁，这个时候再调用子类的虚函数没有任何意义。

22. 构造函数的执行顺序？析构函数的执行顺序？构造函数内部干了啥？拷贝构造干了啥？

1) 构造函数顺序
   ① 基类构造函数。如果有多个基类，则构造函数的调用顺序是某类在类派生表中出现的顺序，而不是它们在成员初始化表中的顺序。
   ② 成员类对象构造函数。如果有多个成员类对象则构造函数的调用顺序是对象在类中被声明的顺序，而不是它们出现在成员初始化表中的顺序。
   ③ 派生类构造函数。
2) 析构函数顺序
   ① 调用派生类的析构函数；
   ② 调用成员类对象的析构函数；

③ 调用基类的析构函数。

23. 虚析构函数的作用，父类的析构函数是否要设置为虚函数？

1) C++中基类采用 virtual 虚析构函数是为了防止内存泄漏。具体地说，如果派生类中申请了内存空间，并在其析构函数中对这些内存空间进行释放。假设基类中采用的是非虚析构函数，当删除基类指针指向的派生类对象时就不会触发动态绑定，因而只会调用基类的析构函数，而不会调用派生类的析构函数。那么在这种情况下，派生类中申请的空间就得不到释放从而产生内存泄漏。所以，为了防止这种情况的发生，C++中基类的析构函数应采用 virtual 虚析构函数。
2) 纯虚析构函数一定得定义，因为每一个派生类析构函数会被编译器加以扩张，以静态调用的方式调用其每一个虚基类以及上一层基类的析构函数。因此，缺乏任何一个基类析构函数的定义，就会导致链接失败。因此，最好不要把虚析构函数定义为纯虚析构函数。

24. 构造函数析构函数可以调用虚函数吗？

1) 在构造函数和析构函数中最好不要调用虚函数；
2) 构造函数或者析构函数调用虚函数并不会发挥虚函数动态绑定的特性，跟普通函数没区别；
3) 即使构造函数或者析构函数如果能成功调用虚函数， 程序的运行结果也是不可控的。

25. 构造函数析构函数可否抛出异常

1) C++只会析构已经完成的对象，对象只有在其构造函数执行完毕才算是完全构造妥当。在构造函数中发生异常，控制权转出构造函数之外。因此，在对象 b 的构造函数中发生异常，对象 b 的析构函数不会被调用。因此会造成内存泄漏。
2) 用 auto_ptr 对象来取代指针类成员，便对构造函数做了强化，免除了抛出异常时发生资源泄漏的危机，不再需要在析构函数中手动释放资源；
3) 如果控制权基于异常的因素离开析构函数，而此时正有另一个异常处于作用状态，C++会调用 terminate 函数让程序结束；
4) 如果异常从析构函数抛出，而且没有在当地进行捕捉，那个析构函数便是执行不全的。如果析构函数执行不全，就是没有完成他应该执行的每一件事情。

26. 类如何实现只能静态分配和只能动态分配

1) 前者是把 new、delete 运算符重载为 private 属性。后者是把构造、析构函数设为 protected 属性，再用子类来动态创建
2) 建立类的对象有两种方式：

① 静态建立，静态建立一个类对象，就是由编译器为对象在栈空间中分配内存；

② 动态建立，A *p = new A();动态建立一个类对象，就是使用 new 运算符为对象在堆空间中分配内存。这个过程分为两步，第一步执行 operator new()函数，在堆中搜索一块内存并进行分配；第二步调用类构造函数构造对象；

3) 只有使用 new 运算符，对象才会被建立在堆上，因此只要限制 new 运算符就可以实现类对象只能建立在栈上。可以将 new 运算符设为私有。

27. 如果想将某个类用作基类，为什么该类必须定义而非声明?

1) 派生类中包含并且可以使用它从基类继承而来的成员，为了使用这些成员，派生类必须知道他们是什么。

28. 什么情况会自动生成默认构造函数?

1) 带有默认构造函数的类成员对象，如果一个类没有任何构造函数，但它含有一个成员对象，而后者有默认构造函数，那么编译器就为该类合成出一个默认构造函数。不过这个合成操作只有在构造函数真正被需要的时候才会发生；如果一个类 A 含有多个成员类对象的话，那么类 A 的每一个构造函数必须调用每一个成员对象的默认构造函数而且必须按照类对象在类 A 中的声明顺序进行；
2) 带有默认构造函数的基类，如果一个没有任务构造函数的派生类派生自一个带有默认构造函数基类，那么该派生类会合成一个构造函数调用上一层基类的默认构造函数；
3) 带有一个虚函数的类
4) 带有一个虚基类的类
5) 合成的默认构造函数中，只有基类子对象和成员类对象会被初始化。所有其他的非静态数据成员都不会被初始化。

29. 什么是类的继承?

1) 类与类之间的关系
   has-A 包含关系，用以描述一个类由多个部件类构成，实现 has-A 关系用类的成员属性表示，即一个类的成员属性是另一个已经定义好的类；
   use-A，一个类使用另一个类，通过类之间的成员函数相互联系，定义友元或者通过传递参数的方式来实现；
   is-A，继承关系，关系具有传递性；
2) 继承的相关概念
   所谓的继承就是一个类继承了另一个类的属性和方法，这个新的类包含了上一个类的属性和方法，被称为子类或者派生类，被继承的类称为父类或者基类；
3) 继承的特点
   子类拥有父类的所有属性和方法，子类可以拥有父类没有的属性和方法，子类对象可以当做父类对象使用；

4）继承中的访问控制

public、protected、private

5）继承中的构造和析构函数

6）继承中的兼容性原则

30. 什么是组合?

1） 一个类里面的数据成员是另一个类的对象，即内嵌其他类的对象作为自己的成员；创建组合类的对象：首先创建各个内嵌对象，难点在于构造函数的设计。创建对象时既要对基本类型的成员进行初始化，又要对内嵌对象进行初始化。

2） 创建组合类对象，构造函数的执行顺序：先调用内嵌对象的构造函数，然后按照内嵌对象成员在组合类中的定义顺序，与组合类构造函数的初始化列表顺序无关。然后执行组合类构造函数的函数体，析构函数调用顺序相反。

31. 抽象基类为什么不能创建对象?

抽象类是一种特殊的类，它是为了抽象和设计的目的为建立的，它处于继承层次结构的较上层。

（1）抽象类的定义：

称带有纯虚函数的类为抽象类。

（2）抽象类的作用：

抽象类的主要作用是将有关的操作作为结果接口组织在一个继承层次结构中，由它来为派生类提供一个公共的根，派生类将具体实现在其基类中作为接口的操作。所以派生类实际上刻画了一组子类的操作接口的通用语义，这些语义也传给子类，子类可以具体实现这些语义，也可以再将这些语义传给自己的子类。

（3）使用抽象类时注意：

抽象类只能作为基类来使用，其纯虚函数的实现由派生类给出。如果派生类中没有重新定义纯虚函数，而只是继承基类的纯虚函数，则这个派生类仍然还是一个抽象类。如果派生类中给出了基类纯虚函数的实现，则该派生类就不再是抽象类了，它是一个可以建立对象的具体的类。

抽象类是不能定义对象的。一个纯虚函数不需要（但是可以）被定义。

一、纯虚函数定义

纯虚函数是一种特殊的虚函数，它的一般格式如下：

```
class <类名>
{
virtual <类型><函数名>(<参数表>)=0;
…
};
```

在许多情况下，在基类中不能对虚函数给出有意义的实现，而把它声明为纯虚函数，它的实现留给该基类的派生类去做。这就是纯虚函数的作用。

纯虚函数可以让类先具有一个操作名称，而没有操作内容，让派生类在继承时再去具体地给出定义。凡是含有纯虚函数的类叫做抽象类。这种类不能声明对象，只是作为基类为派生类服务。除非在派生类中完全实现基类中所有的的纯虚函数，否则，派生类

也变成了抽象类，不能实例化对象。

二、纯虚函数引入原因

1、为了方便使用多态特性，我们常常需要在基类中定义虚拟函数。

2、在很多情况下，基类本身生成对象是不合情理的。例如，动物作为一个基类可以派生出老虎、孔 雀等子类，但动物本身生成对象明显不合常理。

为了解决上述问题，引入了纯虚函数的概念，将函数定义为纯虚函数（方法：virtual ReturnType Function()= 0;）。若要使派生类为非抽象类，则编译器要求在派生类中，必须对纯虚函数予以重载以实现多态性。同时含有纯虚函数的类称为抽象类，它不能生成对象。这样就很好地解决了上述两个问题。

例如，绘画程序中，shape 作为一个基类可以派生出圆形、矩形、正方形、梯形等， 如果我要求面积总和的话，那么会可以使用一个 shape * 的数组，只要依次调用派生类的 area()函数了。如果不用接口就没法定义成数组，因为既可以是 circle ，也可以是 square ，而且以后还可能加上 rectangle，等等.

三、相似概念

1、多态性

指相同对象收到不同消息或不同对象收到相同消息时产生不同的实现动作。C++支持两种多态性：编译时多态性，运行时多态性。

a. 编译时多态性：通过重载函数实现

b. 运行时多态性：通过虚函数实现。

2、虚函数

虚函数是在基类中被声明为 virtual，并在派生类中重新定义的成员函数，可实现成员函数的动态重载。

3、抽象类

包含纯虚函数的类称为抽象类。由于抽象类包含了没有定义的纯虚函数，所以不能定义抽象类的对象。

32. 类什么时候会析构?

1) 对象生命周期结束，被销毁时；
2) delete 指向对象的指针时，或 delete 指向对象的基类类型指针，而其基类虚构函数是虚函数时；
3) 对象 i 是对象 o 的成员，o 的析构函数被调用时，对象 i 的析构函数也被调用。

33. 为什么友元函数必须在类内部声明?

1) 因为编译器必须能够读取这个结构的声明以理解这个数据类型的大、行为等方面的所有规则。有一条规则在任何关系中都很重要，那就是谁可以访问我的私有部分。

34. 介绍一下 C++里面的多态?

（1）静态多态（重载，模板）

是在编译的时候，就确定调用函数的类型。
（2）动态多态（覆盖，虚函数实现）
在运行的时候，才确定调用的是哪个函数，动态绑定。运行基类指针指向派生类的对象，并调用派生类的函数。

虚函数实现原理：虚函数表和虚函数指针。

纯虚函数： virtual int fun() = 0;

函数的运行版本由实参决定，在运行时选择函数的版本，所以动态绑定又称为运行时绑定。
当编译器遇到一个模板定义时，它并不生成代码。只有当实例化出模板的一个特定版本时，编译器才会生成代码。

35. 用 C 语言实现 C++的继承

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

//C++中的继承与多态
struct A
{
    virtual void fun()    //C++中的多态:通过虚函数实现
    {
        cout<<"A:fun()"<<endl;
    }

    int a;
};
struct B:public A         //C++中的继承:B 类公有继承 A 类
{
    virtual void fun()    //C++中的多态:通过虚函数实现（子类的关键字 virtual 可加可不加）
    {
        cout<<"B:fun()"<<endl;
    }

    int b;
};

//C 语言模拟 C++的继承与多态

typedef void (*FUN)();      //定义一个函数指针来实现对成员函数的继承

struct _A       //父类
{
```

```
    FUN _fun;   //由于 C 语言中结构体不能包含函数，故只能用函数指针在外面实现

    int _a;
};

struct _B         //子类
{
    _A _a_;     //在子类中定义一个基类的对象即可实现对父类的继承
    int _b;
};

void _fA()        //父类的同名函数
{
    printf("_A:_fun()\n");
}
void _fB()        //子类的同名函数
{
    printf("_B:_fun()\n");
}


void Test()
{
    //测试 C++中的继承与多态
    A a;    //定义一个父类对象 a
    B b;    //定义一个子类对象 b

    A* p1 = &a;   //定义一个父类指针指向父类的对象
    p1->fun();    //调用父类的同名函数
    p1 = &b;      //让父类指针指向子类的对象
    p1->fun();    //调用子类的同名函数


    //C 语言模拟继承与多态的测试
    _A _a;    //定义一个父类对象_a
    _B _b;    //定义一个子类对象_b
    _a._fun = _fA;        //父类的对象调用父类的同名函数
    _b._a_._fun = _fB;    //子类的对象调用子类的同名函数

    _A* p2 = &_a;   //定义一个父类指针指向父类的对象
    p2->_fun();     //调用父类的同名函数
    p2 = (_A*)&_b;  //让父类指针指向子类的对象,由于类型不匹配所以要进行强转
    p2->_fun();     //调用子类的同名函数
}
```

36. 继承机制中对象之间如何转换？指针和引用之间如何转换？

1) 向上类型转换
   将派生类指针或引用转换为基类的指针或引用被称为向上类型转换，向上类型转换会自动进行，而且向上类型转换是安全的。
2) 向下类型转换
   将基类指针或引用转换为派生类指针或引用被称为向下类型转换，向下类型转换不会自动进行，因为一个基类对应几个派生类，所以向下类型转换时不知道对应哪个派生类，所以在向下类型转换时必须加动态类型识别技术。RTTI 技术，用 dynamic_cast 进行向下类型转换。

37. 组合与继承优缺点？

一：继承

继承是 Is a 的关系，比如说 Student 继承 Person,则说明 Student is a Person。继承的优点是子类可以重写父类的方法来方便地实现对父类的扩展。

继承的缺点有以下几点：

①：父类的内部细节对子类是可见的。

②：子类从父类继承的方法在编译时就确定下来了，所以无法在运行期间改变从父类继承的方法的行为。

③：如果对父类的方法做了修改的话（比如增加了一个参数），则子类的方法必须做出相应的修改。所以说子类与父类是一种高耦合，违背了面向对象思想。

二：组合

组合也就是设计类的时候把要组合的类的对象加入到该类中作为自己的成员变量。

组合的优点：

①：当前对象只能通过所包含的那个对象去调用其方法，所以所包含的对象的内部细节对当前对象时不可见的。

②：当前对象与包含的对象是一个低耦合关系，如果修改包含对象的类中代码不需要修改当前对象类的代码。

③：当前对象可以在运行时动态的绑定所包含的对象。可以通过 set 方法给所包含对象赋值。

组合的缺点：①：容易产生过多的对象。②：为了能组合多个对象，必须仔细对接口进行定义。

38. 左值右值

1) 在 C++11 中所有的值必属于左值、右值两者之一，右值又可以细分为纯右值、将亡值。在 C++11 中可以取地址的、有名字的就是左值，反之，不能取地址的、没有名字的就是右值（将亡值或纯右值）。举个例子，int a = b+c, a 就是左值，其有变量名为 a，通过&a 可以获取该变量的地址；表达式 b+c、函数 int func()的返回值是右值，在其被赋值给某一变量前，我们不能通过变量名找到它，&(b+c)这样的操作则不会通过编译。

2) C++11 对 C++98 中的右值进行了扩充。在 C++11 中右值又分为纯右值（prvalue，Pure Rvalue）和将亡值（xvalue，eXpiring Value）。其中纯右值的概念等同于我们在 C++98 标准中右值的概念，指的是临时变量和不跟对象关联的字面量值；将亡值则是 C++11 新增的跟右值引用相关的表达式，这样表达式通常是将要被移动的对象（移为他用），比如返回右值引用 T&&的函数返回值、std::move 的返回值，或者转换为 T&&的类型转换函数的返回值。将亡值可以理解为通过“盗取”其他变量内存空间的方式获取到的值。在确保其他变量不再被使用、或即将被销毁时，通过“盗取”的方式可以避免内存空间的释放和分配，能够延长变量值的生命期。

3) 左值引用就是对一个左值进行引用的类型。右值引用就是对一个右值进行引用的类型，事实上，由于右值通常不具有名字，我们也只能通过引用的方式找到它的存在。右值引用和左值引用都是属于引用类型。无论是声明一个左值引用还是右值引用，都必须立即进行初始化。而其原因可以理解为是引用类型本身自己并不拥有所绑定对象的内存，只是该对象的一个别名。左值引用是具名变量值的别名，而右值引用则是不具名（匿名）变量的别名。左值引用通常也不能绑定到右值，但常量左值引用是个“万能”的引用类型。它可以接受非常量左值、常量左值、右值对其进行初始化。不过常量左值所引用的右值在它的“余生”中只能是只读的。相对地，非常量左值只能接受非常量左值对其进行初始化。

4) 右值值引用通常不能绑定到任何的左值，要想绑定一个左值到右值引用，通常需要 std::move()将左值强制转换为右值。

39. 移动构造函数

1) 我们用对象 a 初始化对象 b，后对象 a 我们就不在使用了，但是对象 a 的空间还在呀（在析构之前），既然拷贝构造函数，实际上就是把 a 对象的内容复制一份到 b 中，那么为什么我们不能直接使用 a 的空间呢？这样就避免了新的空间的分配，大大降低了构造的成本。这就是移动构造函数设计的初衷；
2) 拷贝构造函数中，对于指针，我们一定要采用深层复制，而移动构造函数中，对于指针，我们采用浅层复制。浅层复制之所以危险，是因为两个指针共同指向一片内存空间，若

第一个指针将其释放，另一个指针的指向就不合法了。所以我们只要避免第一个指针释放空间就可以了。避免的方法就是将第一个指针（比如 a->value）置为 NULL，这样在调用析构函数的时候，由于有判断是否为 NULL 的语句，所以析构 a 的时候并不会回收 a->value 指向的空间；

3) 移动构造函数的参数和拷贝构造函数不同，拷贝构造函数的参数是一个左值引用，但是移动构造函数的初值是一个右值引用。意味着，移动构造函数的参数是一个右值或者将亡值的引用。也就是说，只用用一个右值，或者将亡值初始化另一个对象的时候，才会调用移动构造函数。而那个 move 语句，就是将一个左值变成一个将亡值。

40. C 语言的编译链接过程?

源代码——>预处理——>编译——>优化——>汇编——>链接-->可执行文件

1) 预处理

读取 c 源程序，对其中的伪指令（以#开头的指令）和特殊符号进行处理。包括宏定义替换、条件编译指令、头文件包含指令、特殊符号。 预编译程序所完成的基本上是对源程序的“替代”工作。经过此种替代，生成一个没有宏定义、没有条件编译指令、没有特殊符号的输出文件。.i 预处理后的 c 文件，.ii 预处理后的 C++ 文件。

2) 编译阶段

编译程序所要作得工作就是通过词法分析和语法分析，在确认所有的指令都符合语法规则之后，将其翻译成等价的中间代码表示或汇编代码。.s 文件

3) 汇编过程

汇编过程实际上指把汇编语言代码翻译成目标机器指令的过程。对于被翻译系统处理的每一个 C 语言源程序，都将最终经过这一处理而得到相应的目标文件。目标文件中所存放的也就是与源程序等效的目标的机器语言代码。.o 目标文件

4) 链接阶段

链接程序的主要工作就是将有关的目标文件彼此相连接，也即将在一个文件中引用的符号同该符号在另外一个文件中的定义连接起来，使得所有的这些目标文件成为一个能够诶操作系统装入执行的统一整体。

41. vector 与 list 的区别与应用？怎么找某 vector 或者 list 的倒数第二个元素

1) vector 数据结构

vector 和数组类似，拥有一段连续的内存空间，并且起始地址不变。因此能高效的进行随机存取，时间复杂度为 o(1);但因为内存空间是连续的，所以在进行插入和删除操作时，会造成内存块的拷贝，时间复杂度为 o(n)。另外，当数组中内存空间不够时，会重新申请一块内存空间并进行内存拷贝。连续存储结构：vector 是可以实现动态增长的对象数组，支持对数组高效率的访问和在数组尾端的删除和插入操作，在中间和头部删除和插入相对不易，需要挪动大量的数据。它与数组最大的区别就是 vector 不需程序员自己去考虑容量问题，库里面本身已经实现了容量的动态增长，而数组需要程序员手动写入扩容函数进形扩容。

2) list 数据结构

list 是由双向链表实现的，因此内存空间是不连续的。只能通过指针访问数据，所以 list 的随机存取非常没有效率，时间复杂度为 o(n);但由于链表的特点，能高效地进行插入和删除。非连续存储结构：list 是一个双链表结构，支持对链表的双向遍历。每个节点包括三个信息：元素本身，指向前一个元素的节点（prev）和指向下一个元素的节点（next）。因此 list 可以高效率的对数据元素任意位置进行访问和插入删除等操作。由于涉及对额外指针的维护，所以开销比较大。

区别：

vector 的随机访问效率高，但在插入和删除时（不包括尾部）需要挪动数据，不易操作。list 的访问要遍历整个链表，它的随机访问效率低。但对数据的插入和删除操作等都比较方便，改变指针的指向即可。list 是单向的，vector 是双向的。vector 中的迭代器在使用后就失效了，而 list 的迭代器在使用之后还可以继续使用。

3)

```
int  mySize = vec.size();vec.at(mySize -2);
```

list 不提供随机访问，所以不能用下标直接访问到某个位置的元素，要访问 list 里的元素只能遍历，不过你要是只需要访问 list 的最后 N 个元素的话，可以用反向迭代器来遍历：

42. STL vector 的实现，删除其中的元素，迭代器如何变化？为什么是两倍扩容？释放空间？

size()函数返回的是已用空间大小，capacity()返回的是总空间大小，capacity()-size()则是剩余的可用空间大小。当 size()和 capacity()相等，说明 vector 目前的空间已被用完，如果再添加新元素，则会引起 vector 空间的动态增长。

由于动态增长会引起重新分配内存空间、拷贝原空间、释放原空间，这些过程会降低程序效率。因此，可以使用 reserve(n)预先分配一块较大的指定大小的内存空间，这样当指定大小的内存空间未使用完时，是不会重新分配内存空间的，这样便提升了效率。只有当 n>capacity()时，调用 reserve(n)才会改变 vector 容量。

resize()成员函数只改变元素的数目，不改变 vector 的容量。

1. 空的 vector 对象，size()和 capacity()都为 0
2. 当空间大小不足时，新分配的空间大小为原空间大小的 2 倍。
3. 使用 reserve()预先分配一块内存后，在空间未满的情况下，不会引起重新分配，从而提升了效率。
4. 当 reserve()分配的空间比原空间小时，是不会引起重新分配的。
5. resize()函数只改变容器的元素数目，未改变容器大小。
6. 用 reserve(size_type)只是扩大 capacity 值，这些内存空间可能还是“野”的，如果此时使用“[ ]”来访问，则可能会越界。而 resize(size_type new_size)会真正使容器具有 new_size 个对象。

1. 不同的编译器，vector 有不同的扩容大小。在 vs 下是 1.5 倍，在 GCC 下是 2 倍；

2. 空间和时间的权衡。简单来说， 空间分配的多，平摊时间复杂度低，但浪费空间也多。
3. 使用 k=2 增长因子的问题在于，每次扩展的新尺寸必然刚好大于之前分配的总和，也就是说，之前分配的内存空间不可能被使用。这样对内存不友好。最好把增长因子设为(1,2)

比较内存分配的情况：

```
k = 2, c = 4
0123
    01234567
            012345789ABCDEF
                        0123456789ABCDEF0123456789ABCDEF
                                                        012345...

k = 1.5, c = 4
0123
    012345
          012345678
                  0123456789ABCD
                              0123456789ABCDEF0123
0123456789ABCDEF0123456789ABCD
                            0123456789ABCDEF0123456789ABCDEF...
```

4. 对比可以发现采用采用成倍方式扩容，可以保证常数的时间复杂度，而增加指定大小的容量只能达到 O(n)的时间复杂度，因此，使用成倍的方式扩容。

如何释放空间：

由于 vector 的内存占用空间只增不减，比如你首先分配了 10,000 个字节，然后 erase 掉后面 9,999 个，留下一个有效元素，但是内存占用仍为 10,000 个。所有内存空间是在 vector 析构时候才能被系统回收。empty()用来检测容器是否为空的，clear()可以清空所有元素。但是即使 clear()，vector 所占用的内存空间依然如故，无法保证内存的回收。

如果需要空间动态缩小，可以考虑使用 deque。如果 vector，可以用 swap()来帮助你释放内存。

vector(Vec).swap(Vec);
将 Vec 的内存空洞清除;
vector().swap(Vec);
清空 Vec 的内存;

43. 容器内部删除一个元素

1) 顺序容器
erase 迭代器不仅使所指向被删除的迭代器失效，而且使被删元素之后的所有迭代器失效(list 除外)，所以不能使用 erase(it++)的方式，但是 erase 的返回值是下一个有效迭代器；
```
It = c.erase(it);
```
2) 关联容器
erase 迭代器只是被删除元素的迭代器失效，但是返回值是 void，所以要采用 erase(it++)的方式删除迭代器；
```
c.erase(it++)
```

44. STL 迭代器如何实现

1. 迭代器是一种抽象的设计理念，通过迭代器可以在不了解容器内部原理的情况下遍历容器，除此之外，STL 中迭代器一个最重要的作用就是作为容器与 STL 算法的粘合剂。
2. 迭代器的作用就是提供一个遍历容器内部所有元素的接口，因此迭代器内部必须保存一个与容器相关联的指针，然后重载各种运算操作来遍历，其中最重要的是*运算符与->运算符，以及++、--等可能需要重载的运算符重载。这和 C++中的智能指针很像，智能指针也是将一个指针封装，然后通过引用计数或是其他方法完成自动释放内存的功能。
3. 最常用的迭代器的相应型别有五种：value type、difference type、pointer、reference、iterator catagoly;

45. set 与 hash_set 的区别

1. set 底层是以 RB-Tree 实现，hash_set 底层是以 hash_table 实现的；
2. RB-Tree 有自动排序功能，而 hash_table 不具有自动排序功能；
3. set 和 hash_set 元素的键值就是实值；
4. hash_table 有一些无法处理的型别；

46. hashmap 与 map 的区别

1. 底层实现不同；
2. map 具有自动排序的功能，hash_map 不具有自动排序的功能；
3. hashtable 有一些无法处理的型别；

47. map、set 是怎么实现的，红黑树是怎么能够同时实现这两种容器？为什么使用红黑树？

1) 他们的底层都是以红黑树的结构实现，因此插入删除等操作都在 O(logn)时间内完成，因此可以完成高效的插入删除；
2) 在这里我们定义了一个模版参数，如果它是 key 那么它就是 set，如果它是 map，那么它就是 map；底层是红黑树，实现 map 的红黑树的节点数据类型是 key+value，而实现 set 的节点数据类型是 value
3) 因为 map 和 set 要求是自动排序的，红黑树能够实现这一功能，而且时间复杂度比较低。

48. 如何在共享内存上使用 stl 标准库？

1) 想像一下把 STL 容器，例如 map， vector， list 等等，放入共享内存中，IPC 一旦

有了这些强大的通用数据结构做辅助，无疑进程间通信的能力一下子强大了很多。我们没必要再为共享内存设计其他额外的数据结构，另外，STL 的高度可扩展性将为 IPC 所驱使。STL 容器被良好的封装，默认情况下有它们自己的内存管理方案。当一个元素被插入到一个 STL 列表(list)中时，列表容器自动为其分配内存，保存数据。考虑到要将 STL 容器放到共享内存中，而容器却自己在堆上分配内存。一个最笨拙的办法是在堆上构造 STL 容器，然后把容器复制到共享内存，并且确保所有容器的内部分配的内存指向共享内存中的相应区域，这基本是个不可能完成的任务。

2) 假设进程 A 在共享内存中放入了数个容器，进程 B 如何找到这些容器呢？一个方法就是进程 A 把容器放在共享内存中的确定地址上（fixed offsets），则进程 B 可以从该已知地址上获取容器。另外一个改进点的办法是，进程 A 先在共享内存某块确定地址上放置一个 map 容器，然后进程 A 再创建其他容器，然后给其取个名字和地址一并保存到这个 map 容器里。进程 B 知道如何获取该保存了地址映射的 map 容器，然后同样再根据名字取得其他容器的地址。

49. map 插入方式有几种?

1) 用 insert 函数插入 pair 数据，

```
mapStudent.insert(pair<int, string>(1, "student_one"));
```

2) 用 insert 函数插入 value_type 数据

```
mapStudent.insert(map<int, string>::value_type (1, "student_one"));
```

3) 在 insert 函数中使用 make_pair()函数

```
mapStudent.insert(make_pair(1, "student_one"));
```

4) 用数组方式插入数据

```
mapStudent[1] = "student_one";
```

50. STL 中 unordered_map(hash_map)和 map 的区别，hash_map 如何解决冲突以及扩容

1) unordered_map 和 map 类似，都是存储的 key-value 的值，可以通过 key 快速索引到 value。不同的是 unordered_map 不会根据 key 的大小进行排序，

2) 存储时是根据 key 的 hash 值判断元素是否相同，即 unordered_map 内部元素是无序的，而 map 中的元素是按照二叉搜索树存储，进行中序遍历会得到有序遍历。

3) 所以使用时 map 的 key 需要定义 operator<。而 unordered_map 需要定义 hash_value 函数并且重载 operator==。但是很多系统内置的数据类型都自带这些，

4) 那么如果是自定义类型，那么就需要自己重载 operator<或者 hash_value()了。

5) 如果需要内部元素自动排序，使用 map，不需要排序使用 unordered_map

6) unordered_map 的底层实现是 hash_table;

7) hash_map 底层使用的是 hash_table，而 hash_table 使用的开链法进行冲突避免，所有 hash_map 采用开链法进行冲突解决。

8) **什么时候扩容：**当向容器添加元素的时候，会判断当前容器的元素个数，如果大于等于阈值---即当前数组的长度乘以加载因子的值的时候，就要自动扩容啦。

9) **扩容(resize)**就是重新计算容量，向 HashMap 对象里不停的添加元素，而 HashMap 对象内部的数组无法装载更多的元素时，对象就需要扩大数组的长度，以便能装入更多的元素。

51. vector 越界访问下标，map 越界访问下标？vector 删除元素时会不会释放空间?

1) 通过下标访问 vector 中的元素时不会做边界检查，即便下标越界。也就是说，下标与 first 迭代器相加的结果超过了 finish 迭代器的位置，程序也不会报错，而是返回这个地址中存储的值。如果想在访问 vector 中的元素时首先进行边界检查，可以使用 vector 中的 at 函数。通过使用 at 函数不但可以通过下标访问 vector 中的元素，而且在 at 函数内部会对下标进行边界检查。

2) map 的下标运算符[]的作用是：将 key 作为下标去执行查找，并返回相应的值；如果不存在这个 key，就将一个具有该 key 和 value 的某人值插入这个 map。

3) erase()函数，只能删除内容，不能改变容量大小； erase 成员函数，它删除了 itVect 迭代器指向的元素，并且返回要被删除的 itVect 之后的迭代器，迭代器相当于一个智能指针;clear()函数，只能清空内容，不能改变容量大小;如果要想在删除内容的同时释放内存，那么你可以选择 deque 容器。

52. map[]与 find 的区别?

1) map 的下标运算符[]的作用是：将关键码作为下标去执行查找，并返回对应的值；如果不存在这个关键码，就将一个具有该关键码和值类型的默认值的项插入这个 map。
2) map 的 find 函数：用关键码执行查找，找到了返回该位置的迭代器；如果不存在这个关键码，就返回尾迭代器。

53. STL 中 list 与 queue 之间的区别

1) list 不再能够像 vector 一样以普通指针作为迭代器，因为其节点不保证在存储空间中连续存在；
2) list 插入操作和结合才做都不会造成原有的 list 迭代器失效；
3) list 不仅是一个双向链表，而且还是一个环状双向链表，所以它只需要一个指针；
4) list 不像 vector 那样有可能在空间不足时做重新配置、数据移动的操作，所以插入前的所有迭代器在插入操作之后都仍然有效；
5) deque 是一种双向开口的连续线性空间，所谓双向开口，意思是可以在头尾两端分别做

元素的插入和删除操作；可以在头尾两端分别做元素的插入和删除操作；

6) deque 和 vector 最大的差异，一在于 deque 允许常数时间内对起头端进行元素的插入或移除操作，二在于 deque 没有所谓容量概念，因为它是动态地以分段连续空间组合而成，随时可以增加一段新的空间并链接起来，deque 没有所谓的空间保留功能。

54. STL 中的 allocator,deallocator

1) 第一级配置器直接使用 malloc()、free()和 relloc()，第二级配置器视情况采用不同的策略：当配置区块超过 128bytes 时，视之为足够大，便调用第一级配置器；当配置器区块小于 128bytes 时，为了降低额外负担，使用复杂的内存池整理方式，而不再用一级配置器；
2) 第二级配置器主动将任何小额区块的内存需求量上调至 8 的倍数，并维护 16 个 free-list，各自管理大小为 8~128bytes 的小额区块；
3) 空间配置函数 allocate()，首先判断区块大小，大于 128 就直接调用第一级配置器，小于 128 时就检查对应的 free-list。如果 free-list 之内有可用区块，就直接拿来用，如果没有可用区块，就将区块大小调整至 8 的倍数，然后调用 refill()，为 free-list 重新分配空间；
4) 空间释放函数 deallocate()，该函数首先判断区块大小，大于 128bytes 时，直接调用一级配置器，小于 128bytes 就找到对应的 free-list 然后释放内存。

55. STL 中 hash_map 扩容发生什么?

1) hash table 表格内的元素称为桶（bucket），而由桶所链接的元素称为节点（node），其中存入桶元素的容器为 stl 本身很重要的一种序列式容器——vector 容器。之所以选择 vector 为存放桶元素的基础容器，主要是因为 vector 容器本身具有动态扩容能力，无需人工干预。
2) 向前操作：首先尝试从目前所指的节点出发，前进一个位置（节点），由于节点被安置于 list 内，所以利用节点的 next 指针即可轻易完成前进操作，如果目前正巧是 list 的尾端，就跳至下一个 bucket 身上，那正是指向下一个 list 的头部节点。

56. map 如何创建?

1.vector 底层数据结构为数组 ，支持快速随机访问

2.list 底层数据结构为双向链表，支持快速增删

3.deque 底层数据结构为一个中央控制器和多个缓冲区，详细见 STL 源码剖析 P146，支持首尾（中间不能）快速增删，也支持随机访问

deque 是一个双端队列(double-ended queue)，也是在堆中保存内容的.它的保存形式如下:

[堆 1] --> [堆 2] -->[堆 3] --> ...

每个堆保存好几个元素,然后堆和堆之间有指针指向,看起来像是 list 和 vector 的结合品.

4.stack 底层一般用 list 或 deque 实现，封闭头部即可，不用 vector 的原因应该是容量大小有限制，扩容耗时

5.queue 底层一般用 list 或 deque 实现，封闭头部即可，不用 vector 的原因应该是容量大小有限制，扩容耗时（stack 和 queue 其实是适配器,而不叫容器，因为是对容器的再封装）

6.priority_queue 的底层数据结构一般为 vector 为底层容器，堆 heap 为处理规则来管理底层容器实现

7.set 底层数据结构为红黑树，有序，不重复

8.multiset 底层数据结构为红黑树，有序，可重复

9.map 底层数据结构为红黑树，有序，不重复

10.multimap 底层数据结构为红黑树，有序，可重复

11.hash_set 底层数据结构为 hash 表，无序，不重复

12.hash_multiset 底层数据结构为 hash 表，无序，可重复
13.hash_map 底层数据结构为 hash 表，无序，不重复
14.hash_multimap 底层数据结构为 hash 表，无序，可重复

57. vector 的增加删除都是怎么做的？为什么是 1.5 倍?

1) 新增元素：vector 通过一个连续的数组存放元素，如果集合已满，在新增数据的时候，就要分配一块更大的内存，将原来的数据复制过来，释放之前的内存，在插入新增的元素；
2) 对 vector 的任何操作，一旦引起空间重新配置，指向原 vector 的所有迭代器就都失效了 ；
3) 初始时刻 vector 的 capacity 为 0，塞入第一个元素后 capacity 增加为 1；
4) 不同的编译器实现的扩容方式不一样，VS2015 中以 1.5 倍扩容，GCC 以 2 倍扩容。

对比可以发现采用采用成倍方式扩容，可以保证常数的时间复杂度，而增加指定大小的容量只能达到 O(n)的时间复杂度，因此，使用成倍的方式扩容。

1) 考虑可能产生的堆空间浪费，成倍增长倍数不能太大，使用较为广泛的扩容方式有两种，以 2 二倍的方式扩容，或者以 1.5 倍的方式扩容。

2) 以 2 倍的方式扩容，导致下一次申请的内存必然大于之前分配内存的总和，导致之前分配的内存不能再被使用，所以最好倍增长因子设置为(1,2)之间：

3) 向量容器 vector 的成员函数 pop_back()可以删除最后一个元素.

4) 而函数 erase()可以删除由一个 iterator 指出的元素，也可以删除一个指定范围的元素。

5) 还可以采用通用算法 remove()来删除 vector 容器中的元素.

6) 不同的是：采用 remove 一般情况下不会改变容器的大小，而 pop_back()与 erase()等成员函数会改变容器的大小。

58. 函数指针?

1) 什么是函数指针?

函数指针指向的是特殊的数据类型，函数的类型是由其返回的数据类型和其参数列表共同决定的，而函数的名称则不是其类型的一部分。

一个具体函数的名字，如果后面不跟调用符号(即括号)，则该名字就是该函数的指针(注意：大部分情况下，可以这么认为，但这种说法并不很严格)。

2) 函数指针的声明方法

```
int  (*pf)(const  int&,  const  int&); (1)
```

上面的 pf 就是一个函数指针，指向所有返回类型为 int，并带有两个 const int&参数的函数。注意*pf 两边的括号是必须的，否则上面的定义就变成了：

```
int  *pf(const  int&,  const  int&); (2)
```

而这声明了一个函数 pf，其返回类型为 int *， 带有两个 const int&参数。

3) 为什么有函数指针

函数与数据项相似，函数也有地址。我们希望在同一个函数中通过使用相同的形参在不同的时间使用产生不同的效果。

4) 一个函数名就是一个指针，它指向函数的代码。一个函数地址是该函数的进入点，也就是调用函数的地址。函数的调用可以通过函数名，也可以通过指向函数的指针来调用。函数指针还允许将函数作为变元传递给其他函数；

5) 两种方法赋值：

指针名 = 函数名； 指针名 = &函数名

59. 说说你对 c 和 c++的看法，c 和 c++的区别?

1) 第一点就应该想到 C 是面向过程的语言，而 C++是面向对象的语言，一般简历上第一条都是熟悉 C/C++基本语法，了解 C++面向对象思想，那么，请问什么是面向对象？

2) C 和 C++动态管理内存的方法不一样，C 是使用 malloc/free 函数，而 C++除此之外还有 new/delete 关键字；（关于 malooc/free 与 new/delete 的不同又可以说一大堆，最后的扩展_1 部分列出十大区别）；

3) 接下来就不得不谈到 C 中的 struct 和 C++的类，C++的类是 C 所没有的，但是 C 中的 struct 是可以在 C++中正常使用的，并且 C++对 struct 进行了进一步的扩展，使 struct 在 C++中可以和 class 一样当做类使用，而唯一和 class 不同的地方在于 struct 的成员默认访问修饰符是 public,而 class 默认的是 private;

4) C++支持函数重载，而 C 不支持函数重载，而 C++支持重载的依仗就在于 C++的名字修饰与 C 不同，例如在 C++中函数 int fun(int ,int)经过名字修饰之后变为 _fun_int_int ,而 C 是
_fun，一般是这样的，所以 C++才会支持不同的参数调用不同的函数；

5) C++中有引用，而 C 没有；这样就不得不提一下引用和指针的区别（文后扩展_2）；

6) 当然还有 C++全部变量的默认链接属性是外链接，而 C 是内连接；

7) C 中用 const 修饰的变量不可以用在定义数组时的大小，但是 C++用 const 修饰的变量可以（如果不进行&,解引用的操作的话，是存放在符号表的，不开辟内存）；

8) 当然还有局部变量的声明规则不同，多态，C++特有输入输出流之类的，很多，下面就不再列出来了； “`

60. c/c++的内存分配，详细说一下栈、堆、静态存储区?

1、栈区（stack）— 由编译器自动分配释放，存放函数的参数值，局部变量的值等其操作方式类似于数据结构中的栈。
2、堆区（heap） — 一般由程序员分配释放，若程序员不释放，程序结束时可能由OS（操作系统）回收。注意它与数据结构中的堆是两回事，分配方式倒是类似于链表。

3、全局区（静态区）（static）—，全局变量和静态变量的存储是放在一块的，初始化的全局变量和静态变量在一块区域，未初始化的全局变量和未初始化的静态变量在相邻的另一块区域。程序结束后由系统释放。

4、文字常量区　　—常量字符串就是放在这里的。程序结束后由系统释放。

5、程序代码区　　—存放函数体的二进制代码。

61. 堆与栈的区别?

1) *管理方式*：对于栈来讲，是由编译器自动管理，无需我们手工控制；对于堆来说，释放工作由程序员控制，容易产生 memory leak。
2) *空间大小*：一般来讲在 32 位系统下，堆内存可以达到 4G 的空间，从这个角度来看堆内存几乎是没有什么限制的。但是对于栈来讲，一般都是有一定的空间大小的，例如，在 VC6 下面，默认的栈空间大小是 1M（好像是，记不清楚了）。当然，我们可以修改： 打开工程，依次操作菜单如下：Project->Setting->Link，在 Category 中选中 Output，然后在 Reserve 中设定堆栈的最大值和 commit。 注意：reserve 最小值为 4Byte；commit 是保留在虚拟内存的页文件里面，它设置的较大会使栈开辟较大的值，可能增加内存的开销和启动时间。
3) *碎片问题*：对于堆来讲，频繁的 new/delete 势必会造成内存空间的不连续，从而造成大量的碎片，使程序效率降低。对于栈来讲，则不会存在这个问题，因为栈是先进后出的队列，他们是如此的一一对应，以至于永远都不可能有一个内存块从栈中间弹出，在他弹出之前，在他上面的后进的栈内容已经被弹出，详细的可以参考数据结构，这里我们就不再一一讨论了。
4) *生长方向*：对于堆来讲，生长方向是向上的，也就是向着内存地址增加的方向；对于栈来讲，它的生长方向是向下的，是向着内存地址减小的方向增长。
5) *分配方式*：堆都是动态分配的，没有静态分配的堆。栈有 2 种分配方式：静态分配和动态分配。静态分配是编译器完成的，比如局部变量的分配。动态分配由 alloca 函数进行分配，但是栈的动态分配和堆是不同的，它的动态分配是由编译器进行释放，无需我们手工实现。
6) *分配效率*：栈是机器系统提供的数据结构，计算机会在底层对栈提供支持：分配专门的寄存器存放栈的地址，压栈出栈都有专门的指令执行，这就决定了栈的效率比较高。堆则是 C/C++函数库提供的，它的机制是很复杂的，例如为了分配一块内存，库函数会按照一定的算法（具体的算法可以参考数据结构/操作系统）在堆内存中搜索可用的足够大小的空间，如果没有足够大小的空间（可能是由于内存碎片太多），就有可能调用系统功能去增加程序数据段的内存空间，这样就有机会分到足够大小的内存，然后进行返回。显然，堆的效率比栈要低得多。

62. 野指针是什么？如何检测内存泄漏?

1) 野指针：指向内存被释放的内存或者没有访问权限的内存的指针。

2) “野指针”的成因主要有 3 种：

① 指针变量没有被初始化。任何指针变量刚被创建时不会自动成为 NULL 指针，它的缺省值是随机的，它会乱指一气。所以，指针变量在创建的同时应当被初始化，要么将指针设置为 NULL，要么让它指向合法的内存。例如

```
char *p = NULL;
char *str = new char(100);
```

② 指针 p 被 free 或者 delete 之后，没有置为 NULL；

③ 指针操作超越了变量的作用范围。

3) 如何避免野指针:

① 对指针进行初始化

①将指针初始化为 NULL。

```
char *    p    = NULL;
```

②用 malloc 分配内存

```
char * p = (char * )malloc(sizeof(char));
```

③用已有合法的可访问的内存地址对指针初始化

```
char num[ 30] = {0};
char *p = num;
```

② 指针用完后释放内存，将指针赋 NULL。

```
delete(p);
p = NULL;
```

63. 悬空指针和野指针有什么区别?

1) 野指针：野指针指，访问一个已删除或访问受限的内存区域的指针，野指针不能判断是否为 NULL 来避免。指针没有初始化，释放后没有置空，越界

2) 悬空指针：一个指针的指向对象已被删除，那么就成了悬空指针。野指针是那些未初始化的指针。

64. 内存泄漏

3) 内存泄漏
内存泄漏是指由于疏忽或错误造成了程序未能释放掉不再使用的内存的情况。内存泄漏并非指内存在物理上消失，而是应用程序分配某段内存后，由于设计错误，失去了对该段内存的控制；

4) 后果
只发生一次小的内存泄漏可能不被注意，但泄漏大量内存的程序将会出现各种证照：性能下降到内存逐渐用完，导致另一个程序失败；

5) 如何排除
使用工具软件 BoundsChecker，BoundsChecker 是一个运行时错误检测工具，它主要定位程序运行时期发生的各种错误；
调试运行 DEBUG 版程序，运用以下技术：CRT(C run-time libraries)、运行时函数调用堆栈、内存泄漏时提示的内存分配序号(集成开发环境 OUTPUT 窗口)，综合分析内存泄漏的原因，排除内存泄漏。

6) 解决方法
智能指针。

7) 检查、定位内存泄漏
检查方法：在 main 函数最后面一行，加上一句_CrtDumpMemoryLeaks()。调试程序，自然关闭程序让其退出，查看输出：
输出这样的格式{453}normal block at 0x02432CA8,868 bytes long
被{}包围的 453 就是我们需要的内存泄漏定位值，868 bytes long 就是说这个地方有 868 比特内存没有释放。
定位代码位置
在 main 函数第一行加上_CrtSetBreakAlloc(453);意思就是在申请 453 这块内存的位置中断。然后调试程序，程序中断了，查看调用堆栈。加上头文件#include <crtdbg.h>

65. new 和 malloc 的区别?

1、new/delete 是 C++关键字，需要编译器支持。malloc/free 是库函数，需要头文件支持；
2、使用 new 操作符申请内存分配时无须指定内存块的大小，编译器会根据类型信息自行计算。而 malloc 则需要显式地指出所需内存的尺寸。
3、new 操作符内存分配成功时，返回的是对象类型的指针，类型严格与对象匹配，无须进行类型转换，故 new 是符合类型安全性的操作符。而 malloc 内存分配成功则是返回 void * ，需要通过强制类型转换将 void*指针转换成我们需要的类型。
4、new 内存分配失败时，会抛出 bac_alloc 异常。malloc 分配内存失败时返回 NULL。
5、new 会先调用 operator new 函数，申请足够的内存（通常底层使用 malloc 实现）。

然后调用类型的构造函数，初始化成员变量，最后返回自定义类型指针。delete 先调用析构函数，然后调用 operator delete 函数释放内存（通常底层使用 free 实现）。malloc/free 是库函数，只能动态的申请和释放内存，无法强制要求其做自定义类型对象构造和析构工作。

66. delete p;与 delete[]p， allocator

1、 动态数组管理 new 一个数组时，[]中必须是一个整数，但是不一定是常量整数，普通数组必须是一个常量整数；
2、 new 动态数组返回的并不是数组类型，而是一个元素类型的指针；
3、 delete[]时，数组中的元素按逆序的顺序进行销毁；
4、 new 在内存分配上面有一些局限性，new 的机制是将内存分配和对象构造组合在一起，同样的，delete 也是将对象析构和内存释放组合在一起的。allocator 将这两部分分开进行，allocator 申请一部分内存，不进行初始化对象，只有当需要的时候才进行初始化操作。

67. new 和 delete 的实现原理， delete 是如何知道释放内存的大小的额?

1、 new 简单类型直接调用 operator new 分配内存；而对于复杂结构，先调用 operator new 分配内存，然后在分配的内存上调用构造函数；对于简单类型，new[]计算好大小后调用 operator new；对于复杂数据结构，new[]先调用 operator new[]分配内存，然后在 p 的前四个字节写入数组大小 n，然后调用 n 次构造函数，针对复杂类型，new[]会额外存储数组大小；

   ① new 表达式调用一个名为 operator new(operator new[])函数，分配一块足够大的、原始的、未命名的内存空间；

   ② 编译器运行相应的构造函数以构造这些对象，并为其传入初始值；

   ③ 对象被分配了空间并构造完成，返回一个指向该对象的指针。

2、 delete 简单数据类型默认只是调用 free 函数；复杂数据类型先调用析构函数再调用 operator delete；针对简单类型，delete 和 delete[]等同。假设指针 p 指向 new[]分配的内存。因为要 4 字节存储数组大小，实际分配的内存地址为[p-4]，系统记录的也是这个地址。delete[]实际释放的就是 p-4 指向的内存。而 delete 会直接释放 p 指向的内存，这个内存根本没有被系统记录，所以会崩溃。
3、 需要在 new [] 一个对象数组时，需要保存数组的维度，C++ 的做法是在分配数组空间时多分配了 4 个字节的大小，专门保存数组的大小，在 delete [] 时就可以取出这个保存的数，就知道了需要调用析构函数多少次了。

68. malloc 申请的存储空间能用 delete 释放吗

不能，malloc /free 主要为了兼容 C，new 和 delete 完全可以取代 malloc /free 的。malloc /free 的操作对象都是必须明确大小的。而且不能用在动态类上。new 和 delete 会自动进行类型检查和大小，malloc/free 不能执行构造函数与析构函数，所以动态对象它是不行的。当然从理论上说使用 malloc 申请的内存是可以通过 delete 释放的。不过一般不这样写的。而且也不能保证每个 C++的运行时都能正常。

69. malloc 与 free 的实现原理?

1、 在标准 C 库中，提供了 malloc/free 函数分配释放内存，这两个函数底层是由 brk、mmap、，munmap 这些系统调用实现的;

2、 brk 是将数据段(.data)的最高地址指针_edata 往高地址推,mmap 是在进程的虚拟地址空间中（堆和栈中间，称为文件映射区域的地方）找一块空闲的虚拟内存。这两种方式分配的都是虚拟内存，没有分配物理内存。在第一次访问已分配的虚拟地址空间的时候，发生缺页中断，操作系统负责分配物理内存，然后建立虚拟内存和物理内存之间的映射关系;

3、 malloc 小于 128k 的内存，使用 brk 分配内存，将_edata 往高地址推；malloc 大于 128k 的内存，使用 mmap 分配内存，在堆和栈之间找一块空闲内存分配；brk 分配的内存需要等到高地址内存释放以后才能释放，而 mmap 分配的内存可以单独释放。当最高地址空间的空闲内存超过 128K（可由 M_TRIM_THRESHOLD 选项调节）时，执行内存紧缩操作（trim）。在上一个步骤 free 的时候，发现最高地址空闲内存超过 128K，于是内存紧缩。

4、 malloc 是从堆里面申请内存，也就是说函数返回的指针是指向堆里面的一块内存。操作系统中有一个记录空闲内存地址的链表。当操作系统收到程序的申请时，就会遍历该链表，然后就寻找第一个空间大于所申请空间的堆结点，然后就将该结点从空闲结点链表中删除，并将该结点的空间分配给程序。

70. malloc、realloc、calloc 的区别

1) malloc 函数

```
void* malloc(unsigned int num_size);
int *p = malloc(20*sizeof(int));申请 20 个 int 类型的空间;
```

2) calloc 函数

```
void* calloc(size_t n,size_t size);
int *p = calloc(20, sizeof(int));
```

省去了人为空间计算；malloc 申请的空间的值是随机初始化的，calloc 申请的空间的值

是初始化为 0 的；

3) realloc 函数

void realloc(void *p, size_t new_size);

给动态分配的空间分配额外的空间，用于扩充容量。

71. __stdcall 和__cdecl 的区别?

1) __stdcall

__stdcall 是函数恢复堆栈，只有在函数代码的结尾出现一次恢复堆栈的代码；在编译时就规定了参数个数，无法实现不定个数的参数调用；

2) __cdecl

__cdecl 是调用者恢复堆栈，假设有 100 个函数调用函数 a，那么内存中就有 100 端恢复堆栈的代码；可以不定参数个数；每一个调用它的函数都包含清空堆栈的代码，所以产生的可执行文件大小会比调用__stacall 函数大。

72. 使用智能指针管理内存资源，RAII

1) RAII 全称是“Resource Acquisition is Initialization”，直译过来是“资源获取即初始化”，也就是说在构造函数中申请分配资源，在析构函数中释放资源。因为 C++的语言机制保证了，当一个对象创建的时候，自动调用构造函数，当对象超出作用域的时候会自动调用析构函数。所以，在 RAII 的指导下，我们应该使用类来管理资源，将资源和对象的生命周期绑定。

2) 智能指针（std::shared_ptr 和 std::unique_ptr）即 RAII 最具代表的实现，使用智能指针，可以实现自动的内存管理，再也不需要担心忘记 delete 造成的内存泄漏。毫不夸张的来讲，有了智能指针，代码中几乎不需要再出现 delete 了。

73. 手写实现智能指针类

1) 智能指针是一个数据类型，一般用模板实现，模拟指针行为的同时还提供自动垃圾回收机制。它会自动记录 SmartPointer<T*>对象的引用计数，一旦 T 类型对象的引用计数为 0，就释放该对象。除了指针对象外，我们还需要一个引用计数的指针设定对象的值，并将引用计数计为 1，需要一个构造函数。新增对象还需要一个构造函数，析构函数负责引用计数减少和释放内存。通过覆写赋值运算符，才能将一个旧的智能指针赋值给另一个指针，同时旧的引用计数减 1，新的引用计数加 1

2) 一个构造函数、拷贝构造函数、复制构造函数、析构函数、移走函数；

74. 内存对齐？位域?

1、 分配内存的顺序是按照声明的顺序。

2、 每个变量相对于起始位置的偏移量必须是该变量类型大小的整数倍，不是整数倍空出内存，直到偏移量是整数倍为止。

3、 最后整个结构体的大小必须是里面变量类型最大值的整数倍。

添加了#pragma pack(n)后规则就变成了下面这样：

1、 偏移量要是 n 和当前变量大小中较小值的整数倍

2、 整体大小要是 n 和最大变量大小中较小值的整数倍

3、 n 值必须为 1,2,4,8…，为其他值时就按照默认的分配规则

75. 结构体变量比较是否相等

1) 重载了“==”操作符

```
struct foo {
    int a;
    int b;
    bool operator==(const foo& rhs) // 操作运算符重载
    {
        return( a == rhs.a) && (b == rhs.b);

    }
};
```

2) 元素的话，一个个比；

3) 指针直接比较，如果保存的是同一个实例地址，则(p1==p2)为真；

76. 位运算

若一个数 m 满足 m = 2^n;那么 k%m=k&(m-1)

77. 为什么内存对齐

1、平台原因(移植原因)

1) 不是所有的硬件平台都能访问任意地址上的任意数据的；

2) 某些硬件平台只能在某些地址处取某些特定类型的数据，否则抛出硬件异

2、性能原因：

1) 数据结构(尤其是栈)应该尽可能地在自然边界上对齐。

2) 原因在于，为了访问未对齐的内存，处理器需要作两次内存访问；而对齐的内存访问仅需要一次访问。

78. 函数调用过程栈的变化，返回值和参数变量哪个先入栈?

1、调用者函数把被调函数所需要的参数按照与被调函数的形参顺序相反的顺序压入栈中,即:从右向左依次把被调函数所需要的参数压入栈;

2、调用者函数使用 call 指令调用被调函数,并把 call 指令的下一条指令的地址当成返回地址压入栈中(这个压栈操作隐含在 call 指令中);

3、在被调函数中,被调函数会先保存调用者函数的栈底地址(push ebp),然后再保存调用者函数的栈顶地址,即:当前被调函数的栈底地址(mov ebp,esp);

4、在被调函数中,从 ebp 的位置处开始存放被调函数中的局部变量和临时变量,并且这些变量的地址按照定义时的顺序依次减小,即:这些变量的地址是按照栈的延伸方向排列的,先定义的变量先入栈,后定义的变量后入栈;

79. 怎样判断两个浮点数是否相等?

对两个浮点数判断大小和是否相等不能直接用==来判断，会出错！明明相等的两个数比较反而是不相等！对于两个浮点数比较只能通过相减并与预先设定的精度比较，记得要取绝对值！浮点数与 0 的比较也应该注意。与浮点数的表示方式有关。

80. 宏定义一个取两个数中较大值的功能

```
#define MAX（x,y）((x>y?)x:y)
```

81. define、const、typedef、inline 使用方法?

一、 const 与#define 的区别：

1) const 定义的常量是变量带类型，而#define 定义的只是个常数不带类型；
2) define 只在预处理阶段起作用，简单的文本替换，而 const 在编译、链接过程中起作用；
3) define 只是简单的字符串替换没有类型检查。而 const 是有数据类型的，是要进行判断的，可以避免一些低级错误；
4) define 预处理后，占用代码段空间，const 占用数据段空间；
5) const 不能重定义，而 define 可以通过#undef 取消某个符号的定义，进行重定义；
6) define 独特功能，比如可以用来防止文件重复引用。

二、 #define 和别名 typedef 的区别

1) 执行时间不同，typedef 在编译阶段有效，typedef 有类型检查的功能；#define 是宏定义，发生在预处理阶段，不进行类型检查；
2) 功能差异，typedef 用来定义类型的别名，定义与平台无关的数据类型，与 struct

的结合使用等。#define 不只是可以为类型取别名，还可以定义常量、变量、编译开关等。

3) 作用域不同，#define 没有作用域的限制，只要是之前预定义过的宏，在以后的程序中都可以使用。而 typedef 有自己的作用域。

三、 define 与 inline 的区别

1) #define 是关键字， inline 是函数；
2) 宏定义在预处理阶段进行文本替换，inline 函数在编译阶段进行替换；
3) inline 函数有类型检查，相比宏定义比较安全；

82. printf 实现原理?

在 C/C++中，对函数参数的扫描是从后向前的。C/C++的函数参数是通过压入堆栈的方式来给函数传参数的（堆栈是一种先进后出的数据结构），最先压入的参数最后出来，在计算机的内存中，数据有 2 块，一块是堆，一块是栈（函数参数及局部变量在这里），而栈是从内存的高地址向低地址生长的，控制生长的就是堆栈指针了，最先压入的参数是在最上面，就是说在所有参数的最后面，最后压入的参数在最下面，结构上看起来是第一个，所以最后压入的参数总是能够被函数找到，因为它就在堆栈指针的上方。printf 的第一个被找到的参数就是那个字符指针，就是被双引号括起来的那一部分，函数通过判断字符串里控制参数的个数来判断参数个数及数据类型，通过这些就可算出数据需要的堆栈指针的偏移量了，下面给出 printf("%d,%d",a,b);（其中 a、b 都是 int 型的）的汇编代码.

83. #include 的顺序以及尖叫括号和双引号的区别

表示编译器只在系统默认目录或尖括号内的工作目录下搜索头文件，并不去用户的工作目录下寻找，所以一般尖括号用于包含标准库文件；
表示编译器先在用户的工作目录下搜索头文件，如果搜索不到则到系统默认目录下去寻找，所以双引号一般用于包含用户自己编写的头文件。

84. lambda 函数

1) 利用 lambda 表达式可以编写内嵌的匿名函数，用以替换独立函数或者函数对象；
2) 每当你定义一个 lambda 表达式后，编译器会自动生成一个匿名类（这个类当然重载了()运算符），我们称为闭包类型（closure type）。那么在运行时，这个 lambda 表达式就会返回一个匿名的闭包实例，其实一个右值。所以，我们上面的 lambda 表达式的结果就是一个个闭包。闭包的一个强大之处是其可以通过传值或者引用的方式捕捉其封装作用域内的变量，前面的方括号就是用来定义捕捉模式以及变量，我们又将其称为 lambda 捕捉块。
3) lambda 表达式的语法定义如下：

```
[capture] (parameters) mutable ->return-type {statement};
```

4) lambda 必须使用尾置返回来指定返回类型，可以忽略参数列表和返回值，但必须永远包含捕获列表和函数体；

85. hello world 程序开始到打印到屏幕上的全过程?

1.用户告诉操作系统执行 HelloWorld 程序（通过键盘输入等）

2．操作系统：找到 helloworld 程序的相关信息，检查其类型是否是可执行文件；并通过程序首部信息，确定代码和数据在可执行文件中的位置并计算出对应的磁盘块地址。

3．操作系统：创建一个新进程，将 HelloWorld 可执行文件映射到该进程结构，表示由该进程执行 helloworld 程序。

4．操作系统：为 helloworld 程序设置 cpu 上下文环境，并跳到程序开始处。

5．执行 helloworld 程序的第一条指令，发生缺页异常

6．操作系统：分配一页物理内存，并将代码从磁盘读入内存，然后继续执行 helloworld 程序

7．helloword 程序执行 puts 函数（系统调用），在显示器上写一字符串

8．操作系统：找到要将字符串送往的显示设备，通常设备是由一个进程控制的，所以，操作系统将要写的字符串送给该进程

9．操作系统：控制设备的进程告诉设备的窗口系统，它要显示该字符串，窗口系统确定这是一个合法的操作，然后将字符串转换成像素，将像素写入设备的存储映像区

10．视频硬件将像素转换成显示器可接收和一组控制数据信号

11．显示器解释信号，激发液晶屏

12．OK，我们在屏幕上看到了 HelloWorld

86.模板类和模板函数的区别是什么?

函数模板的实例化是由编译程序在处理函数调用时自动完成的，而类模板的实例化必须由程序员在程序中显式地指定。即函数模板允许隐式调用和显式调用而类模板只能显示调用。在使用时类模板必须加<T>，而函数模板不必

87.为什么模板类一般都是放在一个 h 文件中

1） 模板定义很特殊。由 template<…>处理的任何东西都意味着编译器在当时不为它分配

存储空间，它一直处于等待状态直到被一个模板实例告知。在编译器和连接器的某一处，有一机制能去掉指定模板的多重定义。所以为了容易使用，几乎总是在头文件中放置全部的模板声明和定义。

2) 在分离式编译的环境下，编译器编译某一个.cpp 文件时并不知道另一个.cpp 文件的存在，也不会去查找（当遇到未决符号时它会寄希望于连接器）。这种模式在没有模板的情况下运行良好，但遇到模板时就傻眼了，因为模板仅在需要的时候才会实例化出来，所以，当编译器只看到模板的声明时，它不能实例化该模板，只能创建一个具有外部连接的符号并期待连接器能够将符号的地址决议出来。然而当实现该模板的.cpp 文件中没有用到模板的实例时，编译器懒得去实例化，所以，整个工程的.obj 中就找不到一行模板实例的二进制代码，于是连接器也黔驴技穷了。

88. C++中类成员的访问权限和继承权限问题。

1) 三种访问权限

① public:用该关键字修饰的成员表示公有成员，该成员不仅可以在类内可以被访问，在类外也是可以被访问的，是类对外提供的可访问接口；

② private:用该关键字修饰的成员表示私有成员，该成员仅在类内可以被访问，在类体外是隐藏状态；

③ protected:用该关键字修饰的成员表示保护成员，保护成员在类体外同样是隐藏状态，但是对于该类的派生类来说，相当于公有成员，在派生类中可以被访问。

2) 三种继承方式

① 若继承方式是 public，基类成员在派生类中的访问权限保持不变，也就是说，基类中的成员访问权限，在派生类中仍然保持原来的访问权限；

② 若继承方式是 private，基类所有成员在派生类中的访问权限都会变为私有(private)权限；

③ 若继承方式是 protected，基类的共有成员和保护成员在派生类中的访问权限都会变为保护(protected)权限，私有成员在派生类中的访问权限仍然是私有(private)权限。

89. cout 和 printf 有什么区别?

cout<<是一个函数，cout<<后可以跟不同的类型是因为 cout<<已存在针对各种类型数据的重载，所以会自动识别数据的类型。输出过程会首先将输出字符放入缓冲区，然后输出到屏幕。

cout 是有缓冲输出:

cout < < "abc " < <endl;

或 cout < < "abc\n ";cout < <flush; 这两个才是一样的.

endl 相当于输出回车后，再强迫缓冲输出。

flush 立即强迫缓冲输出。

printf 是无缓冲输出。有输出时立即输出

90. 重载运算符?

1、 我们只能重载已有的运算符，而无权发明新的运算符；对于一个重载的运算符，其优先级和结合律与内置类型一致才可以；不能改变运算符操作数个数；
2、 . :: ? : sizeof typeid **不能重载；
3、 两种重载方式，成员运算符和非成员运算符，成员运算符比非成员运算符少一个参数；下标运算符、箭头运算符必须是成员运算符；
4、 引入运算符重载，是为了实现类的多态性；
5、 当重载的运算符是成员函数时，this 绑定到左侧运算符对象。成员运算符函数的参数数量比运算符对象的数量少一个；至少含有一个类类型的参数；
6、 从参数的个数推断到底定义的是哪种运算符，当运算符既是一元运算符又是二元运算符（+，-，*，&）；
7、 下标运算符必须是成员函数，下标运算符通常以所访问元素的引用作为返回值，同时最好定义下标运算符的常量版本和非常量版本；
8、 箭头运算符必须是类的成员，解引用通常也是类的成员；重载的箭头运算符必须返回类的指针；

91. 函数重载函数匹配原则

1） 名字查找
2） 确定候选函数
3） 寻找最佳匹配

92. 定义和声明的区别

1. 如果是指变量的声明和定义
   从编译原理上来说，声明是仅仅告诉编译器，有个某类型的变量会被使用，但是编译器并不会为它分配任何内存。而定义就是分配了内存。
2. 如果是指函数的声明和定义

声明：一般在头文件里，对编译器说：这里我有一个函数叫 function() 让编译器知道这个函数的存在。
定义：一般在源文件里，具体就是函数的实现过程 写明函数体。

93. C++类型转换有四种

1) static_cast 能进行基础类型之间的转换，也是最常看到的类型转换。它主要有如下几种用法：
1．用于类层次结构中父类和子类之间指针或引用的转换。进行上行转换（把子类的指针或引用转换成父类表示）是安全的；
2．进行下行转换（把父类指针或引用转换成子类指针或引用）时，由于没有动态类型检查，所以是不安全的；
3．用于基本数据类型之间的转换，如把 int 转换成 char，把 int 转换成 enum。这种转换的安全性也要开发人员来保证。
4．把 void 指针转换成目标类型的指针（不安全!!）
5．把任何类型的表达式转换成 void 类型。
2) const_cast 运算符用来修改类型的 const 或 volatile 属性。除了去掉 const 或 volatile 修饰之外， type_id 和 expression 得到的类型是一样的。但需要特别注意的是 const_cast 不是用于去除变量的常量性，而是去除指向常数对象的指针或引用的常量性，其去除常量性的对象必须为指针或引用。
3) reinterpret_cast 它可以把一个指针转换成一个整数，也可以把一个整数转换成一个指针（先把一个指针转换成一个整数，在把该整数转换成原类型的指针，还可以得到原先的指针值）。
4) dynamic_cast 主要用在继承体系中的安全向下转型。它能安全地将指向基类的指针转型为指向子类的指针或引用，并获知转型动作成功是否。转型失败会返回 null（转型对象为指针时）或抛出异常 bad_cast（转型对象为引用时）。 dynamic_cast 会动用运行时信息（RTTI）来进行类型安全检查，因此 dynamic_cast 存在一定的效率损失。当使用 dynamic_cast 时，该类型必须含有虚函数，这是因为 dynamic_cast 使用了存储在 VTABLE 中的信息来判断实际的类型，RTTI 运行时类型识别用于判断类型。typeid 表达式的形式是 typeid(e)，typeid 操作的结果是一个常量对象的引用，该对象的类型是 type_info 或 type_info 的派生。

94. 全局变量和 static 变量的区别

1、全局变量（外部变量）的说明之前再冠以 static 就构成了静态的全局变量。全局变量本身就是静态存储方式，静态全局变量当然也是静态存储方式。

这两者在存储方式上并无不同。这两者的区别在于非静态全局变量的作用域是整个源程序，当一个源程序由多个原文件组成时，非静态的全局变量在各个源文件中都是有效的。而静态全局变量则限制了其作用域，即只在定义该变量的源文件内有效，在同一源程序的其它源文件中不能使用它。由于静态全局变量的作用域限于一个源文件内，只能为该源文件内的函数公用，因此可以避免在其他源文件中引起错误。static 全局变量

与普通的全局变量的区别是 static 全局变量只初始化一次，防止在其他文件单元被引用。

2. static 函数与普通函数有什么区别？

static 函数与普通的函数作用域不同。尽在本文件中。只在当前源文件中使用的函数应该说明为内部函数（static），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 函数与普通函数最主要区别是 static 函数在内存中只有一份，普通静态函数在每个被调用中维持一份拷贝程序的局部变量存在于（堆栈）中，全局变量存在于（静态区）中，动态申请数据存在于（堆）

95. 静态成员与普通成员的区别

1) 生命周期
   静态成员变量从类被加载开始到类被卸载，一直存在；
   普通成员变量只有在类创建对象后才开始存在，对象结束，它的生命期结束；
2) 共享方式
   静态成员变量是全类共享；普通成员变量是每个对象单独享用的；
3) 定义位置
   普通成员变量存储在栈或堆中，而静态成员变量存储在静态全局区；
4) 初始化位置
   普通成员变量在类中初始化；静态成员变量在类外初始化；
5) 默认实参
   可以使用静态成员变量作为默认实参，

96. 说一下理解 ifdef endif

1) 一般情况下，源程序中所有的行都参加编译。但是有时希望对其中一部分内容只在满足一定条件才进行编译，也就是对一部分内容指定编译的条件，这就是“条件编译”。有时，希望当满足某条件时对一组语句进行编译，而当条件不满足时则编译另一组语句。

2) 条件编译命令最常见的形式为：

```
#ifdef 标识符
程序段 1
#else
程序段 2
#endif
```

它的作用是：当标识符已经被定义过(一般是用#define 命令定义)，则对程序段 1 进行编译，否则编译程序段 2。

其中#else 部分也可以没有，即：

#ifdef

程序段 1

#denif

3) 在一个大的软件工程里面，可能会有多个文件同时包含一个头文件，当这些文件编译链接成一个可执行文件上时，就会出现大量“重定义”错误。在头文件中使用#define、#ifndef、#ifdef、#endif 能避免头文件重定义。

97. 隐式转换，如何消除隐式转换?

1. C++的基本类型中并非完全的对立，部分数据类型之间是可以进行隐式转换的。所谓隐式转换，是指不需要用户干预，编译器私下进行的类型转换行为。很多时候用户可能都不知道进行了哪些转换
2. C++面向对象的多态特性，就是通过父类的类型实现对子类的封装。通过隐式转换，你可以直接将一个子类的对象使用父类的类型进行返回。在比如，数值和布尔类型的转换，整数和浮点数的转换等。某些方面来说，隐式转换给 C++程序开发者带来了不小的便捷。C++是一门强类型语言，类型的检查是非常严格的。
3. 基本数据类型 基本数据类型的转换以取值范围的作为转换基础（保证精度不丢失）。隐式转换发生在从小->大的转换中。比如从 char 转换为 int。从 int->long。自定义对象 子类对象可以隐式的转换为父类对象。
4. C++中提供了 explicit 关键字，在构造函数声明的时候加上 explicit 关键字，能够禁止隐式转换。
5. 如果构造函数只接受一个参数，则它实际上定义了转换为此类类型的隐式转换机制。可以通过将构造函数声明为 explicit 加以制止隐式类型转换，关键字 explicit 只对一个实参的构造函数有效，需要多个实参的构造函数不能用于执行隐式转换，所以无需将这些构造函数指定为 explicit。

98. 虚函数的内存结构，那菱形继承的虚函数内存结构呢

99. 多继承的优缺点，作为一个开发者怎么看待多继承

1) C++允许为一个派生类指定多个基类，这样的继承结构被称做多重继承。
2) 多重继承的优点很明显，就是对象可以调用多个基类中的接口；
3) 如果派生类所继承的多个基类有相同的基类，而派生类对象需要调用这个祖先类的接口方法，就会容易出现二义性
4) 加上全局符确定调用哪一份拷贝。比如 pa.Author::eat()调用属于 Author 的拷贝。

5）使用虚拟继承，使得多重继承类 Programmer_Author 只拥有 Person 类的一份拷贝。

100. 迭代器++it,it++哪个好，为什么

1） 前置返回一个引用，后置返回一个对象

```
// ++i 实现代码为：
int& operator++()
{
    *this += 1;
    return *this;
}
```

2） 前置不会产生临时对象，后置必须产生临时对象，临时对象会导致效率降低

```
//i++实现代码为：
int operator++(int)
{
    int temp = *this;
    ++*this;
    return temp;
}
```

101. C++如何处理多个异常的?

1） **C++中的异常情况：**

语法错误（编译错误）：比如变量未定义、括号不匹配、关键字拼写错误等等编译器在编译时能发现的错误，这类错误可以及时被编译器发现，而且可以及时知道出错的位置及原因，方便改正。

运行时错误：比如数组下标越界、系统内存不足等等。这类错误不易被程序员发现，它能通过编译且能进入运行，但运行时会出错，导致程序崩溃。为了有效处理程序运行时错误，C++中引入异常处理机制来解决此问题。

2） **C++异常处理机制：**

异常处理基本思想：执行一个函数的过程中发现异常，可以不用在本函数内立即进行处理， 而是抛出该异常，让函数的调用者直接或间接处理这个问题。

C++异常处理机制由 3 个模块组成：try(检查)、throw(抛出)、catch(捕获)

抛出异常的语句格式为：throw 表达式；如果 try 块中程序段发现了异常则抛出异常。

```
try
{
可能抛出异常的语句;（检查）
}
catch（类型名[形参名]）//捕获特定类型的异常
```

```
{
//处理 1;
}
catch（类型名[形参名]）//捕获特定类型的异常
{
//处理 2;
}
catch（…）//捕获所有类型的异常
{
}
```

102. 模板和实现可不可以不写在一个文件里面？为什么？

因为在编译时模板并不能生成真正的二进制代码，而是在编译调用模板类或函数的 CPP 文件时才会去找对应的模板声明和实现，在这种情况下编译器是不知道实现模板类或函数的 CPP 文件的存在，所以它只能找到模板类或函数的声明而找不到实现，而只好创建一个符号寄希望于链接程序找地址。但模板类或函数的实现并不能被编译成二进制代码，结果链接程序找不到地址只好报错了。

《C++编程思想》第 15 章(第 300 页)说明了原因:模板定义很特殊。由 template<…> 处理的任何东西都意味着编译器在当时不为它分配存储空间，它一直处于等待状态直到被一个模板实例告知。在编译器和连接器的某一处，有一机制能去掉指定模板的多重定义。所以为了容易使用，几乎总是在头文件中放置全部的模板声明和定义。

103. 在成员函数中调用 delete this 会出现什么问题？对象还可以使用吗？

1. 在类对象的内存空间中，只有数据成员和虚函数表指针，并不包含代码内容，类的成员函数单独放在代码段中。在调用成员函数时，隐含传递一个 this 指针，让成员函数知道当前是哪个对象在调用它。当调用 delete this 时，类对象的内存空间被释放。在 delete this 之后进行的其他任何函数调用，只要不涉及到 this 指针的内容，都能够正常运行。一旦涉及到 this 指针，如操作数据成员，调用虚函数等，就会出现不可预期的问题。

2. 为什么是不可预期的问题？

   delete this 之后不是释放了类对象的内存空间了么，那么这段内存应该已经还给系统，不再属于这个进程。照这个逻辑来看，应该发生指针错误，无访问权限之类的令系统崩溃的问题才对啊？这个问题牵涉到操作系统的内存管理策略。delete this 释放了类对象的内存空间，但是内存空间却并不是马上被回收到系统中，可能是

缓冲或者其他什么原因，导致这段内存空间暂时并没有被系统收回。此时这段内存是可以访问的，你可以加上 100，加上 200，但是其中的值却是不确定的。当你获取数据成员，可能得到的是一串很长的未初始化的随机数；访问虚函数表，指针无效的可能性非常高，造成系统崩溃。

3. 如果在类的析构函数中调用 delete this，会发生什么？

会导致堆栈溢出。原因很简单，delete 的本质是“为将被释放的内存调用一个或多个析构函数，然后，释放内存”。显然，delete this 会去调用本对象的析构函数，而析构函数中又调用 delete this，形成无限递归，造成堆栈溢出，系统崩溃。

104. 智能指针的作用;

1) C++11 中引入了智能指针的概念，方便管理堆内存。使用普通指针，容易造成堆内存泄露（忘记释放），二次释放，程序发生异常时内存泄露等问题等，使用智能指针能更好的管理堆内存。
2) 智能指针在 C++11 版本之后提供，包含在头文件<memory>中，shared_ptr、unique_ptr、weak_ptr。shared_ptr 多个指针指向相同的对象。shared_ptr 使用引用计数，每一个 shared_ptr 的拷贝都指向相同的内存。每使用他一次，内部的引用计数加 1，每析构一次，内部的引用计数减 1，减为 0 时，自动删除所指向的堆内存。shared_ptr 内部的引用计数是线程安全的，但是对象的读取需要加锁。
3) 初始化。智能指针是个模板类，可以指定类型，传入指针通过构造函数初始化。也可以使用 make_shared 函数初始化。不能将指针直接赋值给一个智能指针，一个是类，一个是指针。例如 std::shared_ptr<int> p4 = new int(1);的写法是错误的拷贝和赋值。拷贝使得对象的引用计数增加 1，赋值使得原对象引用计数减 1，当计数为 0 时，自动释放内存。后来指向的对象引用计数加 1，指向后来的对象
4) unique_ptr“唯一”拥有其所指对象，同一时刻只能有一个 unique_ptr 指向给定对象（通过禁止拷贝语义、只有移动语义来实现）。相比与原始指针 unique_ptr 用于其 RAII 的特性，使得在出现异常的情况下，动态资源能得到释放。unique_ptr 指针本身的生命周期：从 unique_ptr 指针创建时开始，直到离开作用域。离开作用域时，若其指向对象，则将其所指对象销毁(默认使用 delete 操作符，用户可指定其他操作)。unique_ptr 指针与其所指对象的关系：在智能指针生命周期内，可以改变智能指针所指对象，如创建智能指针时通过构造函数指定、通过 reset 方法重新指定、通过 release 方法释放所有权、通过移动语义转移所有权。
5) 智能指针类将一个计数器与类指向的对象相关联，引用计数跟踪该类有多少个对象共享同一指针。每次创建类的新对象时，初始化指针并将引用计数置为 1；当对象作为另一对象的副本而创建时，拷贝构造函数拷贝指针并增加与之相应的引用计数；对一个对象进行赋值时，赋值操作符减少左操作数所指对象的引用计数（如果引用计数为减至 0，则删除对象），并增加右操作数所指对象的引用计数；调用析构函数时，构造函数减少引用计数（如果引用计数减至 0，则删除基础对象）。
6) weak_ptr 是一种不控制对象生命周期的智能指针，它指向一个 shared_ptr 管理的对象. 进行该对象的内存管理的是那个强引用的 shared_ptr. weak_ptr 只是提

供了对管理对象的一个访问手段。weak_ptr 设计的目的是为配合 shared_ptr 而引入的一种智能指针来协助 shared_ptr 工作，它只可以从一个 shared_ptr 或另一个 weak_ptr 对象构造，它的构造和析构不会引起引用记数的增加或减少.

105. auto_ptr 作用

1) auto_ptr 的出现，主要是为了解决“有异常抛出时发生内存泄漏”的问题；抛出异常，将导致指针 p 所指向的空间得不到释放而导致内存泄漏；
2) auto_ptr 构造时取得某个对象的控制权，在析构时释放该对象。我们实际上是创建一个 auto_ptr<Type>类型的局部对象，该局部对象析构时，会将自身所拥有的指针空间释放，所以不会有内存泄漏；
3) auto_ptr 的构造函数是 explicit，阻止了一般指针隐式转换为 auto_ptr 的构造，所以不能直接将一般类型的指针赋值给 auto_ptr 类型的对象，必须用 auto_ptr 的构造函数创建对象；
4) 由于 auto_ptr 对象析构时会删除它所拥有的指针，所以使用时避免多个 auto_ptr 对象管理同一个指针；
5) Auto_ptr 内部实现，析构函数中删除对象用的是 delete 而不是 delete[]，所以 auto_ptr 不能管理数组；
6) auto_ptr 支持所拥有的指针类型之间的隐式类型转换。
7) 可以通过*和->运算符对 auto_ptr 所有用的指针进行提领操作；
8) T* get(),获得 auto_ptr 所拥有的指针；T* release()，释放 auto_ptr 的所有权，并将所有用的指针返回。

106. class、union、struct 的区别

1) C 语言中，struct 只是一个聚合数据类型，没有权限设置，无法添加成员函数，无法实现面向对象编程，且如果没有 typedef 结构名，声明结构变量必须添加关键字 struct。
2) C++中，struct 功能大大扩展，可以有权限设置(默认权限为 public)，可以像 class 一样有成员函数，继承（默认 public 继承），可以实现面对对象编程，允许在声明结构变量时省略关键字 struct。
3) C 与 C++中的 union:一种数据格式，能够存储不同的数据类型，但只能同时存储其中的一种类型。C++ union 结构式一种特殊的类。它能够包含访问权限、成员变量、成员函数（可以包含构造函数和析构函数）。它不能包含虚函数和静态数据变量。它也不能被用作其他类的基类，它本身也不能有从某个基类派生而来。Union 中得默认访问权限是 public。union 类型是共享内存的，以 size 最大的结构作为自己的大小。每个数据成员在内存中的起始地址是相同的。
4) 在 C/C++程序的编写中，当多个基本数据类型或复合数据结构要占用同一片内存时，我们要使用联合体；当多种类型，多个对象，多个事物只取其一时（我们姑且通俗地称其为“n 选 1”），我们也可以使用联合体来发挥其长处。在某一时刻，一个 union 中只能有一个值是有效的。union 的一个用法就是可以用来测试 CPU 是大端模式还是小端模式：

107. 动态联编与静态联编

1) 在 C++中，联编是指一个计算机程序的不同部分彼此关联的过程。按照联编所进行的阶段不同，可以分为静态联编和动态联编；
2) 静态联编是指联编工作在编译阶段完成的，这种联编过程是在程序运行之前完成的，又称为早期联编。要实现静态联编，在编译阶段就必须确定程序中的操作调用（如函数调用）与执行该操作代码间的关系，确定这种关系称为束定，在编译时的束定称为静态束定。静态联编对函数的选择是基于指向对象的指针或者引用的类型。其优点是效率高，但灵活性差。
3) 动态联编是指联编在程序运行时动态地进行，根据当时的情况来确定调用哪个同名函数，实际上是在运行时虚函数的实现。这种联编又称为晚期联编，或动态束定。动态联编对成员函数的选择是基于对象的类型，针对不同的对象类型将做出不同的编译结果。C++中一般情况下的联编是静态联编，但是当涉及到多态性和虚函数时应该使用动态联编。动态联编的优点是灵活性强，但效率低。动态联编规定，只能通过指向基类的指针或基类对象的引用来调用虚函数，其格式为：指向基类的指针变量名->虚函数名（实参表）或基类对象的引用名.虚函数名（实参表）
4) 实现动态联编三个条件：
   必须把动态联编的行为定义为类的虚函数；
   类之间应满足子类型关系，通常表现为一个类从另一个类公有派生而来；
   必须先使用基类指针指向子类型的对象，然后直接或间接使用基类指针调用虚函数；

108. 动态编译与静态编译

1) 静态编译，编译器在编译可执行文件时，把需要用到的对应动态链接库中的部分提取出来，连接到可执行文件中去，使可执行文件在运行时不需要依赖于动态链接库；
2) 动态编译的可执行文件需要附带一个动态链接库，在执行时，需要调用其对应动态链接库的命令。所以其优点一方面是缩小了执行文件本身的体积，另一方面是加快了编译速度，节省了系统资源。缺点是哪怕是很简单的程序，只用到了链接库的一两条命令，也需要附带一个相对庞大的链接库；二是如果其他计算机上没有安装对应的运行库，则用动态编译的可执行文件就不能运行。

109. 动态链接和静态链接区别

1) 静态连接库就是把(lib)文件中用到的函数代码直接链接进目标程序，程序运行的时候不再需要其它的库文件；动态链接就是把调用的函数所在文件模块（DLL）和调用函数在文件中的位置等信息链接进目标程序，程序运行的时候再从 DLL 中寻找相应函数代码，因此需要相应 DLL 文件的支持。
2) 静态链接库与动态链接库都是共享代码的方式，如果采用静态链接库，则无论你愿不愿意，lib 中的指令都全部被直接包含在最终生成的 EXE 文件中了。但是若使用 DLL，该 DLL 不必被包含在最终 EXE 文件中，EXE 文件执行时可以“动态”地引用和卸载这个与 EXE 独立的 DLL 文件。静态链接库和动态链接库的另外一个区别在于静态链

接库中不能再包含其他的动态链接库或者静态库，而在动态链接库中还可以再包含其他的动态或静态链接库。

3) 动态库就是在需要调用其中的函数时，根据函数映射表找到该函数然后调入堆栈执行。如果在当前工程中有多处对 dll 文件中同一个函数的调用，那么执行时，这个函数只会留下一份拷贝。但是如果有多处对 lib 文件中同一个函数的调用，那么执行时，该函数将在当前程序的执行空间里留下多份拷贝，而且是一处调用就产生一份拷贝。

110. 在不使用额外空间的情况下，交换两个数？

1) 算术

```
x = x + y;
y = x - y;
x = x - y;
```

2) 异或

```
x = x^y;// 只能对 int,char..
y = x^y;
x = x^y;
x ^= y ^= x;
```

111. strcpy 和 memcpy 的区别

1、复制的内容不同。strcpy 只能复制字符串，而 memcpy 可以复制任意内容，例如字符数组、整型、结构体、类等。

2、复制的方法不同。strcpy 不需要指定长度，它遇到被复制字符的串结束符"\0"才结束，所以容易溢出。memcpy 则是根据其第 3 个参数决定复制的长度。

3、用途不同。通常在复制字符串时用 strcpy，而需要复制其他类型数据时则一般用 memcpy

112. 执行 int main(int argc, char *argv[])时的内存结构

参数的含义是程序在命令行下运行的时候，需要输入 argc 个参数，每个参数是以 char 类型输入的，依次存在数组里面，数组是 argv[]，所有的参数在指针 char * 指向的内存中，数组的中元素的个数为 argc 个，第一个参数为程序的名称。

113. volatile 关键字的作用？

volatile 关键字是一种类型修饰符，用它声明的类型变量表示可以被某些编译器

未知的因素更改，比如：操作系统、硬件或者其它线程等。遇到这个关键字声明的变量，编译器对访问该变量的代码就不再进行优化，从而可以提供对特殊地址的稳定访问。声明时语法：int volatile vInt; 当要求使用 volatile 声明的变量的值的时候，系统总是重新从它所在的内存读取数据，即使它前面的指令刚刚从该处读取过数据。而且读取的数据立刻被保存。

volatile 用在如下的几个地方：

1) 中断服务程序中修改的供其它程序检测的变量需要加 volatile；

2) 多任务环境下各任务间共享的标志应该加 volatile；

3) 存储器映射的硬件寄存器通常也要加 volatile 说明，因为每次对它的读写都可能由不同意义；

114. 讲讲大端小端，如何检测（三种方法）

大端模式：是指数据的高字节保存在内存的低地址中，而数据的低字节保存在内存的高地址端。

小端模式，是指数据的高字节保存在内存的高地址中，低位字节保存在在内存的低地址端。

1) 直接读取存放在内存中的十六进制数值，取低位进行值判断

```
int a = 0x12345678;
int *c = &a;
c[0] == 0x12    大端模式
c[0] == 0x78    小段模式
```

2) 用共同体来进行判断

union 共同体所有数据成员是共享一段内存的，后写入的成员数据将覆盖之前的成员数据，成员数据都有相同的首地址。Union 的大小为最大数据成员的大小。

union 的成员数据共用内存，并且首地址都是低地址首字节。Int i= 1 时：大端存储 1 放在最高位，小端存储 1 放在最低位。当读取 char ch 时，是最低地址首字节，大小端会显示不同的值。

```
union w
{
 int i;
 char ch;
};
```

```
w p;
p.i = 1;
if(ch == 1)
```

115. 查看内存的方法

1. 首先打开 vs 编译器，创建好项目，并且将代码写进去，这里就不贴代码了，你可以随便的写个做个测试；
2. 调试的时候做好相应的断点，然后点击**开始调试**；
3. 程序调试之后会在你设置断点的地方暂停，然后选择**调试->窗口->内存**，就打开了

内存数据查看的窗口了。

116. 空类会默认添加哪些东西？怎么写？

1) Empty(); // 缺省构造函数//
2) Empty(const Empty&); // 拷贝构造函数//
3) ~Empty(); // 析构函数//
4) Empty& operator=(const Empty&); // 赋值运算符//

117. 标准库是什么？

1) C++ 标准库可以分为两部分：

   标准函数库： 这个库是由通用的、独立的、不属于任何类的函数组成的。函数库继承自 C 语言。

   面向对象类库： 这个库是类及其相关函数的集合。

2) 输入/输出 I/O、字符串和字符处理、数学、时间、日期和本地化、动态分配、其他、宽字符函数

3) 标准的 C++ I/O 类、String 类、数值类、STL 容器类、STL 算法、STL 函数对象、STL 迭代器、STL 分配器、本地化库、异常处理类、杂项支持库

118. const char* 与 string 之间的关系，传递参数问题？

1) string 是 c++标准库里面其中一个，封装了对字符串的操作，实际操作过程我们可以用 const char*给 string 类初始化
2) 三者的转化关系如下所示：
   a) string 转 const char*

```
string s = “abc”;
const char* c_s = s.c_str();
```

   b) const char* 转 string，直接赋值即可

```
const char* c_s = “abc”;
string s(c_s);
```

   c) string 转 char*

```
string s = “abc”;
char* c;
const int len = s.length();
c = new char[len+1];
strcpy(c,s.c_str());
```

   d) char* 转 string

```
char* c = “abc”;
string s(c);
```

e) const char* 转 char*

```
const char* cpc = “abc”;
char* pc = new char[strlen(cpc)+1];
strcpy(pc,cpc);
```

f) char* 转 const char*，直接赋值即可

```
char* pc = “abc”;
const char* cpc = pc;
```

119. new、delete、operator new、operator delete、placement new、placement delete

1) new operator

new operator 完成了两件事情：用于申请内存和初始化对象。

例如：string* ps = new string("abc");

2) operator new

operator new 类似于 C 语言中的 malloc，只是负责申请内存。

例如：void* buffer = operator new(sizeof(string)); 注意这里 new 前要有个 operator。

3) placement new
用于在给定的内存中初始化对象。

例如：void* buffer = operator new(sizeof(string));buffer = new(buffer) string("abc"); 调用了 placement new，在 buffer 所指向的内存中创建了一个 string 类型的对象并且初始值为“abc”。

4) 因此可以看出：new operator 可以分解 operator new 和 placement new 两个动作，是 operator new 和 placement new 的结合。与 new 对应的 delete 没有 placement delete 语法，它只有两种，分别是 delete operator 和 operator delete。delete operator 和 new operator 对应，完成析构对象和释放内存的操作。而 operator delete 只是用于内存的释放，与 C 语言中的 free 相似。

120. 为什么拷贝构造函数必须传引用不能传值？

1) 拷贝构造函数的作用就是用来复制对象的，在使用这个对象的实例来初始化这个对象的一个新的实例。
2) 参数传递过程到底发生了什么？

将地址传递和值传递统一起来，归根结底还是传递的是"值"(地址也是值，只不过通过它可以找到另一个值)！

i)值传递:

对于内置数据类型的传递时，直接赋值拷贝给形参(注意形参是函数内局部变量)；

对于类类型的传递时，需要首先调用该类的拷贝构造函数来初始化形参(局部对象)；如 void foo(class_type obj_local){}，如果调用 foo(obj)；首先 class_type obj_local(obj) ,这样就定义了局部变量 obj_local 供函数内部使用

ii)引用传递:

无论对内置类型还是类类型，传递引用或指针最终都是传递的地址值！而地址总是指针类型(属于简单类型)，显然参数传递时，按简单类型的赋值拷贝，而不会有拷贝构造函数的调用(对于类类型).

上述 1) 2)回答了为什么拷贝构造函数使用值传递会产生无限递归调用，内存溢出。

拷贝构造函数用来初始化一个非引用类类型对象，如果用传值的方式进行传参数，那么构造实参需要调用拷贝构造函数，而拷贝构造函数需要传递实参，所以会一直递归。

121. 空类的大小是多少？为什么？

1) C++空类的大小不为 0，不同编译器设置不一样，vs 设置为 1；
2) C++标准指出，不允许一个对象（当然包括类对象）的大小为 0，不同的对象不能具有相同的地址；
3) 带有虚函数的 C++类大小不为 1，因为每一个对象会有一个 vptr 指向虚函数表，具体大小根据指针大小确定；
4) C++中要求对于类的每个实例都必须有独一无二的地址,那么编译器自动为空类分配一个字节大小，这样便保证了每个实例均有独一无二的内存地址。

122. 你什么情况用指针当参数，什么时候用引用，为什么？

1) 使用引用参数的主要原因有两个：

   程序员能修改调用函数中的数据对象

   通过传递引用而不是整个数据 - 对象，可以提高程序的运行速度

2) 一般的原则：

   对于使用引用的值而不做修改的函数：

   如果数据对象很小，如内置数据类型或者小型结构，则按照值传递；

   如果数据对象是数组，则使用指针（唯一的选择），并且指针声明为指向 const 的指针；

   如果数据对象是较大的结构，则使用 const 指针或者引用，已提高程序的效率。这样可以节省结构所需的时间和空间；

如果数据对象是类对象，则使用 const 引用（传递类对象参数的标准方式是按照引用传递）；

3) 对于修改函数中数据的函数：

如果数据是内置数据类型，则使用指针

如果数据对象是数组，则只能使用指针

如果数据对象是结构，则使用引用或者指针

如果数据是类对象，则使用引用

123. 大内存申请时候选用哪种？C++变量存在哪？变量的大小存在哪？符号表存在哪？

1. 大内存申请时，采用堆申请空间，用 new 申请；
2. 不同的变量存储在不同的地方，局部变量、全局变量、静态变量；
3. C++对变量名不作存储，在汇编以后不会出现变量名，变量名作用只是用于方便编译成汇编代码，是给编译器看的，是方便人阅读的

124. 为什么会有大端小端，htol 这一类函数的作用

1) 这是因为在计算机系统中，我们是以字节为单位的，每个地址单元都对应着一个字节，一个字节为 8bit。但是在 C 语言中除了 8bit 的 char 之外，还有 16bit 的 short 型，32bit 的 long 型（要看具体的编译器），另外，对于位数大于 8 位的处理器，例如 16 位或者 32 位的处理器，由于寄存器宽度大于一个字节，那么必然存在着一个如何将多个字节安排的问题。因此就导致了大端存储模式和小端存储模式。例如一个 16bit 的 short 型 x，在内存中的地址为 0x0010，x 的值为 0x1122，那么 0x11 为高字节，0x22 为低字节。对于 大端模式，就将 0x11 放在低地址中，即 0x0010 中，0x22 放在高地址中，即 0x0011 中。小端模式，刚好相反。我们常用的 X86 结构是小端模式，而 KEIL C51 则为大端模式。很多的 ARM，DSP 都为小端模式。有些 ARM 处理器还可以由硬件来选择是大端模式还是小端模式。

125. 静态函数能定义为虚函数吗？常函数？

1、static 成员不属于任何类对象或类实例，所以即使给此函数加上 virutal 也是没有任何意义的。2. 静态与非静态成员函数之间有一个主要的区别。那就是静态成员函数没有 this 指针。虚函数依靠 vptr 和 vtable 来处理。vptr 是一个指针，在类的构造函数中创建生成，并且只能用 this 指针来访问它，因为它是类的一个成员，并且 vptr 指向保存虚函数地址的 vtable.对于静态成员函数，它没有 this 指针，所以无法访问 vptr. 这就是为何 static 函数不能为 virtual.虚函数的调用关系：this -> vptr -> vtable ->virtual function

126. this 指针调用成员变量时，堆栈会发生什么变化?

当在类的非静态成员函数访问类的非静态成员时，编译器会自动将对象的地址传给作为隐含参数传递给函数，这个隐含参数就是 this 指针。即使你并没有写 this 指针，编译器在链接时也会加上 this 的，对各成员的访问都是通过 this 的。例如你建立了类的多个对象时，在调用类的成员函数时，你并不知道具体是哪个对象在调用，此时你可以通过查看 this 指针来查看具体是哪个对象在调用。This 指针首先入栈，然后成员函数的参数从右向左进行入栈，最后函数返回地址入栈。

127. 静态绑定和动态绑定的介绍

1) 对象的静态类型：对象在声明时采用的类型。是在编译期确定的。
2) 对象的动态类型：目前所指对象的类型。是在运行期决定的。对象的动态类型可以更改，但是静态类型无法更改。
3) 静态绑定：绑定的是对象的静态类型，某特性（比如函数）依赖于对象的静态类型，发生在编译期。
4) 动态绑定：绑定的是对象的动态类型，某特性（比如函数）依赖于对象的动态类型，发生在运行期。

128. 设计一个类计算子类的个数

1. 为类设计一个 static 静态变量 count 作为计数器；
2. 类定义结束后初始化 count；
3. 在构造函数中对 count 进行+1；
4. 设计拷贝构造函数，在进行拷贝构造函数中进行 count +1，操作；
5. 设计复制构造函数，在进行复制函数中对 count+1 操作；
6. 在析构函数中对 count 进行-1；

129. 怎么快速定位错误出现的地方

1. 如果是简单的错误，可以直接双击错误列表里的错误项或者生成输出的错误信息中带行号的地方就可以让编辑窗口定位到错误的位置上。
2. 对于复杂的模板错误，最好使用生成输出窗口。多数情况下出发错误的位置是最靠后的引用位置。如果这样确定不了错误，就需要先把自己写的代码里的引用位置找出来，然后逐个分析了。

130. 虚函数的代价?

1) 带有虚函数的类，每一个类会产生一个虚函数表，用来存储指向虚成员函数的指针，增大类；
2) 带有虚函数的类的每一个对象，都会有有一个指向虚表的指针，会增加对象的空间

大小；

3) 不能再是内敛的函数，因为内敛函数在编译阶段进行替代，而虚函数表示等待，在运行阶段才能确定到低是采用哪种函数，虚函数不能是内敛函数。

131. 类对象的大小

1) 类的非静态成员变量大小，静态成员不占据类的空间，成员函数也不占据类的空间大小；
2) 内存对齐另外分配的空间大小，类内的数据也是需要进行内存对齐操作的；
3) 虚函数的话，会在类对象插入 vptr 指针，加上指针大小；
4) 当该该类是某类的派生类，那么派生类继承的基类部分的数据成员也会存在在派生类中的空间中，也会对派生类进行扩展。

132. 移动构造函数

1) 有时候我们会遇到这样一种情况，我们用对象 a 初始化对象 b 后对象 a 我们就不在使用了，但是对象 a 的空间还在呀（在析构之前），既然拷贝构造函数，实际上就是把 a 对象的内容复制一份到 b 中，那么为什么我们不能直接使用 a 的空间呢？这样就避免了新的空间的分配，大大降低了构造的成本。这就是移动构造函数设计的初衷；
2) 拷贝构造函数中，对于指针，我们一定要采用深层复制，而移动构造函数中，对于指针，我们采用浅层复制；
3) C++引入了移动构造函数，专门处理这种，用 a 初始化 b 后，就将 a 析构的情况；
4) 与拷贝类似，移动也使用一个对象的值设置另一个对象的值。但是，又与拷贝不同的是，移动实现的是对象值真实的转移（源对象到目的对象）：源对象将丢失其内容，其内容将被目的对象占有。移动操作的发生的时候，是当移动值的对象是未命名的对象的时候。这里未命名的对象就是那些临时变量，甚至都不会有名称。典型的未命名对象就是函数的返回值或者类型转换的对象。使用临时对象的值初始化另一个对象值，不会要求对对象的复制：因为临时对象不会有其它使用，因而，它的值可以被移动到目的对象。做到这些，就要使用移动构造函数和移动赋值：当使用一个临时变量对象进行构造初始化的时候，调用移动构造函数。类似的，使用未命名的变量的值赋给一个对象时，调用移动赋值操作；
5)

```
    Example6 (Example6&& x) : ptr(x.ptr)
    {
        x.ptr = nullptr;
    }
    // move assignment
    Example6& operator= (Example6&& x)
    {
        delete ptr;
        ptr = x.ptr;
        x.ptr=nullptr;
```

```
        return *this;
    }
```

133. 何时需要合成构造函数

1) 如果一个类没有任何构造函数，但他含有一个成员对象，该成员对象含有默认构造函数，那么编译器就为该类合成一个默认构造函数，因为不合成一个默认构造函数那么该成员对象的构造函数不能调用；
2) 没有任何构造函数的类派生自一个带有默认构造函数的基类，那么需要为该派生类合成一个构造函数，只有这样基类的构造函数才能被调用；
3) 带有虚函数的类，虚函数的引入需要进入虚表，指向虚表的指针，该指针是在构造函数中初始化的，所以没有构造函数的话该指针无法被初始化；
4) 带有一个虚基类的类

1) 并不是任何没有构造函数的类都会合成一个构造函数
2) 编译器合成出来的构造函数并不会显示设定类内的每一个成员变量

134. 何时需要合成复制构造函数

有三种情况会以一个对象的内容作为另一个对象的初值：
1) 对一个对象做显示的初始化操作，X xx = x;
2) 当对象被当做参数交给某个函数时；
3) 当函数传回一个类对象时；

1) 如果一个类没有拷贝构造函数，但是含有一个类类型的成员变量，该类型含有拷贝构造函数，此时编译器会为该类合成一个拷贝构造函数；
2) 如果一个类没有拷贝构造函数，但是该类继承自含有拷贝构造函数的基类，此时编译器会为该类合成一个拷贝构造函数；
3) 如果一个类没有拷贝构造函数，但是该类声明或继承了虚函数，此时编译器会为该类合成一个拷贝构造函数；
4) 如果一个类没有拷贝构造函数，但是该类含有虚基类，此时编译器会为该类合成一个拷贝构造函数；

135. 何时需要成员初始化列表？过程是什么？

1) 当初始化一个引用成员变量时；
2) 初始化一个 const 成员变量时；
3) 当调用一个基类的构造函数，而构造函数拥有一组参数时；
4) 当调用一个成员类的构造函数，而他拥有一组参数；
5) 编译器会一一操作初始化列表，以适当顺序在构造函数之内安插初始化操作，并且在任何显示用户代码前。list 中的项目顺序是由类中的成员声明顺序决定的，不是初始化列表中的排列顺序决定的。

136. 程序员定义的析构函数被扩展的过程?

1) 析构函数函数体被执行;
2) 如果 class 拥有成员类对象，而后者拥有析构函数，那么它们会以其声明顺序的相反顺序被调用;
3) 如果对象有一个 vptr，现在被重新定义
4) 如果有任何直接的上一层非虚基类拥有析构函数，则它们会以声明顺序被调用;
5) 如果任何虚基类拥有析构函数

137. 构造函数的执行算法?

1) 在派生类构造函数中，所有的虚基类及上一层基类的构造函数调用;
2) 对象的 vptr 被初始化;
3) 如果有成员初始化列表，将在构造函数体内扩展开来，这必须在 vptr 被设定之后才做;
4) 执行程序员所提供的代码;

138. 构造函数的扩展过程?

1) 记录在成员初始化列表中的数据成员初始化操作会被放在构造函数的函数体内，并与成员的声明顺序为顺序;
2) 如果一个成员并没有出现在成员初始化列表中，但它有一个默认构造函数，那么默认构造函数必须被调用;
3) 如果 class 有虚表，那么它必须被设定初值;
4) 所有上一层的基类构造函数必须被调用;
5) 所有虚基类的构造函数必须被调用。

139. 哪些函数不能是虚函数

1) 构造函数，构造函数初始化对象，派生类必须知道基类函数干了什么，才能进行构造；当有虚函数时，每一个类有一个虚表，每一个对象有一个虚表指针，虚表指针在构造函数中初始化;
2) 内联函数，内联函数表示在编译阶段进行函数体的替换操作，而虚函数意味着在运行期间进行类型确定，所以内联函数不能是虚函数;
3) 静态函数，静态函数不属于对象属于类，静态成员函数没有 this 指针，因此静态函数设置为虚函数没有任何意义。
4) 友元函数，友元函数不属于类的成员函数，不能被继承。对于没有继承特性的函数没有虚函数的说法。
5) 普通函数，普通函数不属于类的成员函数，不具有继承特性，因此普通函数没有虚函数。

140. sizeof 和 strlen 的区别

1) strlen 计算字符串的具体长度（只能是字符串），不包括字符串结束符。返回的是字符个数。
2) sizeof 计算声明后所占的内存数（字节大小），不是实际长度。
3) sizeof 是一个取字节运算符，而 strlen 是个函数。
4) sizeof 的返回值=字符个数*字符所占的字节数，字符实际长度小于定义的长度，此时字符个数就等于定义的长度。若未给出定义的大小，分类讨论，对于字符串数组，字符大 小等于实际的字符个数+1；对于整型数组，字符个数为实际的字符个数。字符串每个字符占 1 个字节，整型数据每个字符占的字节数需根据系统的位数类确定，32 位占 4 个字节。
5) sizeof 可以用类型做参数，strlen 只能用 char*做参数，且必须以‘\0’结尾，sizeof 还可以用函数做参数；
6) 数组做 sizeof 的参数不退化，传递给 strlen 就退化为指针；

141. 简述 strcpy、sprintf 与 memcpy 的区别

1) 操作对象不同

① strcpy 的两个操作对象均为字符串

② sprintf 的操作源对象可以是多种数据类型，目的操作对象是字符串

③ memcpy 的两个对象就是两个任意可操作的内存地址，并不限于何种数据类型。

2) 执行效率不同

memcpy 最高，strcpy 次之，sprintf 的效率最低。

3) 实现功能不同

① strcpy 主要实现字符串变量间的拷贝

② sprintf 主要实现其他数据类型格式到字符串的转化

③ memcpy 主要是内存块间的拷贝。

142. 编码实现某一变量某位清 0 或置 1

```
#define BIT3 (0x1 << 3 ) Satic int a;
```

```
//设置a 的bit 3:
void set_bit3( void )
{
      a |= BIT3; //将a 第3 位置1
}
//清a 的bit 3
void set_bit3( void )
{
    a &= ~BIT3; //将a 第3 位清零
}
```

143. 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。

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

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

无论是float还是double类型的变量，都有精度限制。所以一定要避免将浮点变量用“==”或“！=”与数字比较，应该设法转化成“>=”或“<=”形式。

145. 局部变量全局变量的问题?

1) 局部会屏蔽全局。要用全局变量，需要使用"::"局部变量可以与全局变量同名，在函数内引用这个变量时，会用到同名的局部变量，而不会用到全局变量。对于有些编译器而言，在同一个函数内可以定义多个同名的局部变量，比如在两个循

环体内都定义一个同名的局部变量，而那个局部变量的作用域就在那个循环体内。

2) 如何引用一个已经定义过的全局变量，可以用引用头文件的方式，也可以用 extern 关键字，如果用引用头文件方式来引用某个在头文件中声明的全局变理，假定你将那个变写错了，那么在编译期间会报错，如果你用 extern 方式引用时，假定你犯了同样的错误，那么在编译期间不会报错，而在连接期间报错。

3) 全局变量可不可以定义在可被多个.C 文件包含的头文件中，在不同的 C 文件中以 static 形式来声明同名全局变量。可以在不同的 C 文件中声明同名的全局变量，前提是其中只能有一个 C 文件中对此变量赋初值，此时连接不会出错

146. 数组和指针的区别?

1) 数组在内存中是连续存放的，开辟一块连续的内存空间；数组所占存储空间：sizeof（数组名）；数组大小：sizeof(数组名)/sizeof(数组元素数据类型)；

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。

3) 编译器为了简化对数组的支持，实际上是利用指针实现了对数组的支持。具体来说，就是将表达式中的数组元素引用转换为指针加偏移量的引用。

4) 在向函数传递参数的时候，如果实参是一个数组，那用于接受的形参为对应的指针。也就是传递过去是数组的首地址而不是整个数组，能够提高效率；

5) 在使用下标的时候，两者的用法相同，都是原地址加上下标值，不过数组的原地址就是数组首元素的地址是固定的，指针的原地址就不是固定的。

147. C++如何阻止一个类被实例化？一般在什么时候将构造函数声明为 private?

1) 将类定义为抽象基类或者将构造函数声明为 private；

2) 不允许类外部创建类对象，只能在类内部创建对象

148. 如何禁止自动生成拷贝构造函数?

1) 为了阻止编译器默认生成拷贝构造函数和拷贝赋值函数，我们需要手动去重写这两个函数，某些情况下，为了避免调用拷贝构造函数和拷贝赋值函数，我们需要将他们设置成 private，防止被调用。

2) 类的成员函数和 friend 函数还是可以调用 private 函数，如果这个 private 函数只声明不定义，则会产生一个连接错误；

3) 针对上述两种情况，我们可以定一个 base 类，在 base 类中将拷贝构造函数和拷贝赋值函数设置成 private,那么派生类中编译器将不会自动生成这两个函数，且由于 base 类中该函数是私有的，因此，派生类将阻止编译器执行相关的操作。

149. assert 与 NDEBUGE

1) assert 宏的原型定义在<assert.h>中，其作用是如果它的条件返回错误，则终止程序执行，原型定义：
   #include <assert.h>
   void assert(int expression);
   assert 的作用是现计算表达式 expression ，如果其值为假（即为 0），那么它先向 stderr 打印一条出错信息，然后通过调用 abort 来终止程序运行。如果表达式为真，assert 什么也不做。
2) NDEBUG 宏是 Standard C 中定义的宏，专门用来控制 assert()的行为。如果定义了这个宏，则 assert 不会起作用。定义 NDEBUG 能避免检查各种条件所需的运行时开销，当然此时根本就不会执行运行时检查。
3) C Standard 中规定了 assert 以宏来实现。<assert.h>被设计来可以被多次包含，其中一上来就 undef assert，然后由 NDEBUG 宏来决定其行为。

150. Denug 和 release 的区别

1) 调试版本，包含调试信息，所以容量比 Release 大很多，并且不进行任何优化（优化会使调试复杂化，因为源代码和生成的指令间关系会更复杂），便于程序员调试。Debug 模式下生成两个文件，除了.exe 或.dll 文件外，还有一个.pdb 文件，该文件记录了代码中断点等调试信息；
2) 发布版本，不对源代码进行调试，编译时对应用程序的速度进行优化，使得程序在代码大小和运行速度上都是最优的。（调试信息可在单独的 PDB 文件中生成）。Release 模式下生成一个文件.exe 或.dll 文件。
3) 实际上，Debug 和 Release 并没有本质的界限，他们只是一组编译选项的集合，编译器只是按照预定的选项行动。事实上，我们甚至可以修改这些选项，从而得到优化过的调试版本或是带跟踪语句的发布版本。

151. main 函数有没有返回值

1) 程序运行过程入口点 main 函数，main（）函数返回值类型必须是 int，这样返回值才能传递给程序激活者（如操作系统）表示程序正常退出。main（int args, char **argv） 参数的传递。参数的处理，一般会调用 getopt（）函数处理，但实践中，这仅仅是一部分，不会经常用到的技能点。

152. 写一个比较大小的模板函数

```
1. #include<iostream>
2. using namespace std;
3. template<typename type1,typename type2>//函数模板
4. type1 Max(type1 a,type2 b)
```

```
{
    return a > b ? a : b;
}
void main()
{
    cout<<"Max = "<<Max(5.5,'a')<<endl;
}
```

153. c++怎么实现一个函数先于 main 函数运行

1) 如果在 main 函数之前声明一个类的全局的对象。那么其执行顺序，根据全局对象的生存期和作用域，肯定先于 main 函数。

```
class simpleClass
{
public:
        simpleClass( )
        {
                cout << "simpleClass constructor.." << endl;
    }
};
simpleClass g_objectSimple;          //step1 全局对象
int _tmain(int argc, _TCHAR* argv[])  //step3
{
        return 0;
}
```

2) 定义在 main( )函数之前的全局对象、静态对象的构造函数在 main( )函数之前执行。

3) Main 函数执行之前，主要就是初始化系统相关资源；

① 设置栈指针

② 初始化 static 静态和 global 全局变量，即 data 段的内容

③ 将未初始化部分的全局变量赋初值：数值型 short，int，long 等为 0，bool 为 FALSE，指针为 NULL，等等，即.bss 段的内容

④ 全局对象初始化，在 main 之前调用构造函数

⑤ 将 main 函数的参数，argc，argv 等传递给 main 函数，然后才真正运行 main 函数

4) Main 函数执行之后

① 全局对象的析构函数会在 main 函数之后执行；

② 可以用_onexit 注册一个函数，它会在 main 之后执行;

154. 虚函数与纯虚函数的区别在于

1) 纯虚函数只有定义没有实现，虚函数既有定义又有实现；
2) 含有纯虚函数的类不能定义对象，含有虚函数的类能定义对象；

155. 智能指针怎么用？智能指针出现循环引用怎么解决？

1) shared_ptr
调用一个名为 make_shared 的标准库函数，shared_ptr<int> p = make_shared<int>(42);通常用 auto 更方便，auto p = …;shared_ptr<int> p2(new int(2));
每个 shared_ptr 都有一个关联的计数器，通常称为引用计数，一旦一个 shared_ptr 的计数器变为 0，它就会自动释放自己所管理的对象；shared_ptr 的析构函数就会递减它所指的对象的引用计数。如果引用计数变为 0，shared_ptr 的析构函数就会销毁对象，并释放它占用的内存。
2) unique_ptr
一个 unique_ptr 拥有它所指向的对象。某个时刻只能有一个 unique_ptr 指向一个给定对象。当 unique_ptr 被销毁时，它所指向的对象也被销毁。
3) weak_ptr
weak_ptr 是一种不控制所指向对象生存期的智能指针，它指向由一个 shared_ptr 管理的对象，将一个 weak_ptr 绑定到一个 shared_ptr 不会改变引用计数，一旦最后一个指向对象的 shared_ptr 被销毁，对象就会被释放，即使有 weak_ptr 指向对象，对象还是会被释放。
4) 弱指针用于专门解决 shared_ptr 循环引用的问题，weak_ptr 不会修改引用计数，即其存在与否并不影响对象的引用计数器。循环引用就是：两个对象互相使用一个 shared_ptr 成员变量指向对方。弱引用并不对对象的内存进行管理，在功能上类似于普通指针，然而一个比较大的区别是，弱引用能检测到所管理的对象是否已经被释放，从而避免访问非法内存。

156. strcpy 函数和 strncpy 函数的区别？哪个函数更安全？

1) 函数原型
char* strcpy(char* strDest, const char* strSrc)
char* strncpy(char* strDest, const char* strSrc, int pos)
2) strcpy 函数：如果参数 dest 所指的内存空间不够大，可能会造成缓冲溢出(buffer Overflow)的错误情况，在编写程序时请特别留意，或者用 strncpy()来取代。
strncpy 函数：用来复制源字符串的前 n 个字符，src 和 dest 所指的内存区域不

能重叠，且 dest 必须有足够的空间放置 n 个字符。

3) 如果目标长>指定长>源长，则将源长全部拷贝到目标长，自动加上’\0’
如果指定长<源长，则将源长中按指定长度拷贝到目标字符串，不包括’\0’
如果指定长>目标长，运行时错误 ；

157. 为什么要用 static_cast 转换而不用 c 语言中的转换?

1) 更加安全；
2) 更直接明显，能够一眼看出是什么类型转换为什么类型，容易找出程序中的错误；可清楚地辨别代码中每个显式的强制转；可读性更好，能体现程序员的意图

158. 成员函数里 memset(this,0,sizeof(*this))会发生什么

1) 有时候类里面定义了很多 int,char,struct 等 c 语言里的那些类型的变量，我习惯在构造函数中将它们初始化为 0，但是一句句的写太麻烦，所以直接就 memset(this, 0, sizeof *this);将整个对象的内存全部置为 0。对于这种情形可以很好的工作，但是下面几种情形是不可以这么使用的；
2) 类含有虚函数表：这么做会破坏虚函数表，后续对虚函数的调用都将出现异常；
3) 类中含有 C++类型的对象：例如，类中定义了一个 list 的对象，由于在构造函数体的代码执行之前就对 list 对象完成了初始化，假设 list 在它的构造函数里分配了内存，那么我们这么一做就破坏了 list 对象的内存。

159. 方法调用的原理（栈，汇编）

1) 机器用栈来传递过程参数、存储返回信息、保存寄存器用于以后恢复，以及本地存储。而为单个过程分配的那部分栈称为帧栈；帧栈可以认为是程序栈的一段，它有两个端点，一个标识起始地址，一个标识着结束地址，两个指针结束地址指针 esp，开始地址指针 ebp；
2) 由一系列栈帧构成，这些栈帧对应一个过程，而且每一个栈指针+4 的位置存储函数返回地址；每一个栈帧都建立在调用者的下方，当被调用者执行完毕时，这一段栈帧会被释放。由于栈帧是向地址递减的方向延伸，因此如果我们将栈指针减去一定的值，就相当于给栈帧分配了一定空间的内存。如果将栈指针加上一定的值，也就是向上移动，那么就相当于压缩了栈帧的长度，也就是说内存被释放了。
3) 过程实现

① 备份原来的帧指针，调整当前的栈帧指针到栈指针位置；

② 建立起来的栈帧就是为被调用者准备的，当被调用者使用栈帧时，需要给临时变量分配预留内存；

③ 使用建立好的栈帧，比如读取和写入，一般使用 mov，push 以及 pop 指令等等。

④ 恢复被调用者寄存器当中的值，这一过程其实是从栈帧中将备份的值再恢复到寄存

器，不过此时这些值可能已经不在栈顶了

⑤ 恢复被调用者寄存器当中的值，这一过程其实是从栈帧中将备份的值再恢复到寄存器，不过此时这些值可能已经不在栈顶了。

⑥ 释放被调用者的栈帧，释放就意味着将栈指针加大，而具体的做法一般是直接将栈指针指向帧指针，因此会采用类似下面的汇编代码处理。

⑦ 恢复调用者的栈帧，恢复其实就是调整栈帧两端，使得当前栈帧的区域又回到了原始的位置。

⑧ 弹出返回地址，跳出当前过程，继续执行调用者的代码。

4) 过程调用和返回指令

① call 指令

② leave 指令

③ ret 指令

160. MFC 消息处理如何封装的?

161. 回调函数的作用

1) 当发生某种事件时，系统或其他函数将会自动调用你定义的一段函数；
2) 回调函数就相当于一个中断处理函数，由系统在符合你设定的条件时自动调用。为此，你需要做三件事：1，声明；2，定义；3，设置触发条件，就是在你的函数中把你的回调函数名称转化为地址作为一个参数，以便于系统调用；
3) 回调函数就是一个通过函数指针调用的函数。如果你把函数的指针（地址）作为参数传递给另一个函数，当这个指针被用为调用它所指向的函数时，我们就说这是回调函数；
4) 因为可以把调用者与被调用者分开。调用者不关心谁是被调用者，所有它需知道的，只是存在一个具有某种特定原型、某些限制条件（如返回值为 int）的被调用函数。

162. 随机数的生成

1) #include<time.h> srand((unsigned)time(NULL)); cout<<(rand()%(b-a))+a;
2) 由于 rand()的内部实现是用线性同余法做的，所以生成的并不是真正的随机数，而是在一定范围内可看为随机的伪随机数。
3) 种子写为 srand(time(0))代表着获取系统时间，电脑右下角的时间，每一秒后系统时间的改变，数字序列的改变得到的数字不同，这才得带不同的数字，形成了真随机数，即使是真随机数，也是有规律可循。

操作系统

1. 操作系统特点

并发性、共享性、虚拟性、不确定性。

2. 什么是进程

1) 进程是指在系统中正在运行的一个应用程序，程序一旦运行就是进程；
2) 进程可以认为是程序执行的一个实例，进程是系统进行资源分配的最小单位，且每个进程拥有独立的地址空间；
3) 一个进程无法直接访问另一个进程的变量和数据结构，如果希望一个进程去访问另一个进程的资源，需要使用进程间的通信，比如：管道、消息队列等
4) 线程是进程的一个实体，是进程的一条执行路径；比进程更小的独立运行的基本单位，线程也被称为轻量级进程，一个程序至少有一个进程，一个进程至少有一个线程；

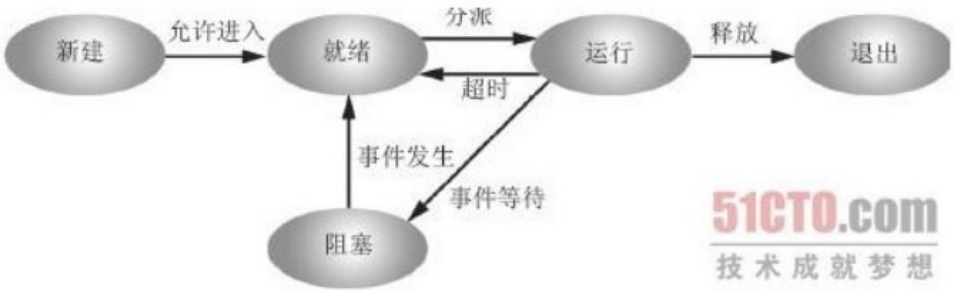
3. 进程

进程是程序的一次执行，该程序可以与其他程序并发执行；
进程有运行、阻塞、就绪三个基本状态；
进程调度算法：先来先服务调度算法、短作业优先调度算法、非抢占式优先级调度算法、抢占式优先级调度算法、高响应比优先调度算法、时间片轮转法调度算法；

4. 进程与线程的区别

1) 同一进程的线程共享本进程的地址空间，而进程之间则是独立的地址空间；
2) 同一进程内的线程共享本进程的资源，但是进程之间的资源是独立的；
3) 一个进程崩溃后，在保护模式下不会对其他进程产生影响，但是一个线程崩溃整个进程崩溃，所以多进程比多线程健壮；
4) 进程切换，消耗的资源大。所以涉及到频繁的切换，使用线程要好于进程；
5) 两者均可并发执行；
6) 每个独立的进程有一个程序的入口、程序出口。但是线程不能独立执行，必须依存在应用程序中，由应用程序提供多个线程执行控制。

5. 进程状态转换图



1) 新状态：进程已经创建
2) 就绪态：进程做好了准备，准备执行，等待分配处理机
3) 执行态：该进程正在执行；
4) 阻塞态：等待某事件发生才能执行，如等待 I/O 完成；
5) 终止状态

6. 进程的创建过程？需要哪些函数？需要哪些数据结构?

1) fork 函数创造的子进程是父进程的完整副本，复制了父亲进程的资源，包括内存的内容 task_struct 内容；
2) vfork 创建的子进程与父进程共享数据段，而且由 vfork 创建的子进程将先于父进程运行；
3) linux 上创建线程一般使用的是 pthread 库，实际上 linux 也给我们提供了创建线程的系统调用，就是 clone；

7. 进程创建子进程,fork 详解

1) 函数原型
   pid_t fork(void); //void 代表没有任何形式参数
2) 除了 0 号进程（系统创建的）之外，linux 系统中都是由其他进程创建的。创建新进程的进程，即调用 fork 函数的进程为父进程，新建的进程为子进程。
3) fork 函数不需要任何参数，对于返回值有三种情况：

   ① 对于父进程，fork 函数返回新建子进程的 pid；

   ② 对于子进程，fork 函数返回 0；

   ③ 如果出错， fork 函数返回 -1。

```
int pid=fork();
if(pid < 0){
```

```
//失败，一般是该用户的进程数达到限制或者内存被用光了
........
}
else if(pid == 0){
//子进程执行的代码
......
}
else{
//父进程执行的代码
.........
}
```

8. 子进程和父进程怎么通信?

1) 在 Linux 系统中实现父子进程的通信可以采用 pipe()和 fork()函数进行实现;
2) 对于父子进程，在程序运行时首先进入的是父进程，其次是子进程，在此我个人认为，在创建父子进程的时候程序是先运行创建的程序，其次在复制父进程创建子进程。fork()函数主要是以父进程为蓝本复制一个进程，其 ID 号和父进程的 ID 号不同。对于结果 fork 出来的子进程的父进程 ID 号是执行 fork()函数的进程的 ID 号。
3) 管道：是指用于连接一个读进程和一个写进程，以实现它们之间通信的共享文件，又称 pipe 文件。
4) 写进程在管道的尾端写入数据，读进程在管道的首端读出数据。

9. 进程和作业的区别?

1) 进程是程序的一次动态执行，属于动态概念;
2) 一个进程可以执行一个或几个程序，同一个程序可由几个进程执行;
3) 程序可以作为一种软件资源长期保留，而进程是程序的一次执行;
4) 进程具有并发性，能与其他进程并发执行;
5) 进程是一个独立的运行单位;

10. 死锁是什么？必要条件？如何解决?

所谓死锁，是指多个进程循环等待它方占有的资源而无限期地僵持下去的局面。很显然，如果没有外力的作用，那麽死锁涉及到的各个进程都将永远处于封锁状态。当两个或两个以上的进程同时对多个互斥资源提出使用要求时，有可能导致死锁。

〈1〉 互斥条件。即某个资源在一段时间内只能由一个进程占有，不能同时被两个或两个以上的进程占有。这种独占资源如 CD-ROM 驱动器，打印机等等，必须在占有该资源的进程主动释放它之后，其它进程才能占有该资源。这是由资源本身的属性所决定的。如独木桥就是一种独占资源，两方的人不能同时过桥。

〈2〉 不可抢占条件。进程所获得的资源在未使用完毕之前，资源申请者不能强行地从资源占有者手中夺取资源，而只能由该资源的占有者进程自行释放。如过独木桥的人不能强迫对方后退，也不能非法地将对方推下桥，必须是桥上的人自己过桥后空出桥面（即主动释放占有资源），对方的人才能过桥。

〈3〉 占有且申请条件。进程至少已经占有一个资源，但又申请新的资源；由于该资源已被另外进程占有，此时该进程阻塞；但是，它在等待新资源之时，仍继续占用已占有的资源。还以过独木桥为例，甲乙两人在桥上相遇。甲走过一段桥面（即占有了一些资源），还需要走其余的桥面（申请新的资源），但那部分桥面被乙占有（乙走过一段桥面）。甲过不去，前进不能，又不后退；乙也处于同样的状况。

〈4〉 循环等待条件。存在一个进程等待序列{P1，P2，…，Pn}，其中 P1 等待 P2 所占有的某一资源，P2 等待 P3 所占有的某一源，……，而 Pn 等待 P1 所占有的的某一资源，形成一个进程循环等待环。就像前面的过独木桥问题，甲等待乙占有的桥面，而乙又等待甲占有的桥面，从而彼此循环等待。

死锁的预防是保证系统不进入死锁状态的一种策略。它的基本思想是要求进程申请资源时遵循某种协议，从而打破产生死锁的四个必要条件中的一个或几个，保证系统不会进入死锁状态。

<1>打破互斥条件。即允许进程同时访问某些资源。但是，有的资源是不允许被同时访问的，像打印机等等，这是由资源本身的属性所决定的。所以，这种办法并无实用价值。

<2>打破不可抢占条件。即允许进程强行从占有者那里夺取某些资源。就是说，当一个进程已占有了某些资源，它又申请新的资源，但不能立即被满足时，它必须释放所占有的全部资源，以后再重新申请。它所释放的资源可以分配给其它进程。这就相当于该进程占有的资源被隐蔽地强占了。这种预防死锁的方法实现起来困难，会降低系统性能。

<3>打破占有且申请条件。可以实行资源预先分配策略。即进程在运行前一次性地向系统申请它所需要的全部资源。如果某个进程所需的全部资源得不到满足，则不分配任何资源，此进程暂不运行。只有当系统能够满足当前进程的全部资源需求时，才一次性地将所申请的资源全部分配给该进程。由于运行的进程已占有了它所需的全部资源，所以不会发生占有资源又申请资源的现象，因此不会发生死锁。

<4>打破循环等待条件，实行资源有序分配策略。采用这种策略，即把资源事先分类编号，按号分配，使进程在申请，占用资源时不会形成环路。所有进程对资源的请求必须严格按资源序号递增的顺序提出。进程占用了小号资源，才能申请大号资源，就不会产生环路，从而预防了死锁

死锁避免：银行家算法

11. 鸵鸟策略

假设的前提是，这样的问题出现的概率很低。比如，在操作系统中，为应对死锁问题，可以采用这样的一种办法。当系统发生死锁时不会对用户造成多大影响，或系统很少发生死锁的场合采用允许死锁发生的鸵鸟算法，这样一来可能开销比不允许发生死锁及检测和解除死锁的小。如果死锁很长时间才发生一次，而系统每周都会因硬件故障、编译器错误或操作系统错误而崩溃一次，那么大多数工程师不会以性能损失或者易用性损失的代价来设计较为复杂的死锁解决策略，来消除死锁。鸵鸟策略的实质：出现死锁的概率很小，并且出现之后处理死锁会花费很大的代价，还不如不做处理，OS 中这种置之不理的策略称之为鸵鸟策略（也叫鸵鸟算法）。

12. 银行家算法

在避免死锁的方法中，所施加的限制条件较弱，有可能获得令人满意的系统性能。在该方法中把系统的状态分为安全状态和不安全状态，只要能使系统始终都处于安全状态，便可以避免发生死锁。

银行家算法的基本思想是分配资源之前，判断系统是否是安全的；若是，才分配。它是最具有代表性的避免死锁的算法。

设进程 cusneed 提出请求 REQUEST [i]，则银行家算法按如下规则进行判断。

(1)如果 REQUEST [cusneed] [i]<= NEED[cusneed][i]，则转（2)；否则，出错。

(2)如果 REQUEST [cusneed] [i]<= AVAILABLE[i]，则转（3)；否则，等待。

(3)系统试探分配资源，修改相关数据：

AVAILABLE[i]-=REQUEST[cusneed][i];

ALLOCATION[cusneed][i]+=REQUEST[cusneed][i];

NEED[cusneed][i]-=REQUEST[cusneed][i];

(4)系统执行安全性检查，如安全，则分配成立；否则试探险性分配作废，系统恢复原状，进程等待。

13. 进程间通信方式有几种，他们之间的区别是什么？

1) 管道

管道，通常指无名管道。

① 半双工的，具有固定的读端和写端；

② 只能用于具有亲属关系的进程之间的通信；

③ 可以看成是一种特殊的文件，对于它的读写也可以使用普通的 read、write 函数。但是它不是普通的文件，并不属于其他任何文件系统，只能用于内存中。

④ Int pipe(int fd[2]);当一个管道建立时，会创建两个文件文件描述符，要关闭管道只需将这两个文件描述符关闭即可。

2) FiFO（有名管道）

① FIFO 可以再无关的进程之间交换数据，与无名管道不同；

② FIFO 有路径名与之相关联，它以一种特殊设备文件形式存在于文件系统中；

③ Int mkfifo(const char* pathname,mode_t mode);

3) 消息队列

① 消息队列，是消息的连接表，存放在内核中。一个消息队列由一个标识符来标识；

② 消息队列是面向记录的，其中的消息具有特定的格式以及特定的优先级；

③ 消息队列独立于发送与接收进程。进程终止时，消息队列及其内容并不会被删除；

④ 消息队列可以实现消息的随机查询

4) 信号量

① 信号量是一个计数器，信号量用于实现进程间的互斥与同步，而不是用于存储进程间通信数据；

② 信号量用于进程间同步，若要在进程间传递数据需要结合共享内存；

③ 信号量基于操作系统的 PV 操作，程序对信号量的操作都是原子操作；

5) 共享内存

① 共享内存，指两个或多个进程共享一个给定的存储区；

② 共享内存是最快的一种进程通信方式，因为进程是直接对内存进行存取；

③ 因为多个进程可以同时操作，所以需要进行同步；

④ 信号量+共享内存通常结合在一起使用。

14. 线程同步的方式？怎么用？

1) 线程同步是指多线程通过特定的设置来控制线程之间的执行顺序，也可以说在线程之间通过同步建立起执行顺序的关系；
2) 主要四种方式，临界区、互斥对象、信号量、事件对象；其中临界区和互斥对象主要用于互斥控制，信号量和事件对象主要用于同步控制；
3) 临界区：通过对多线程的串行化来访问公共资源或一段代码，速度快、适合控制数据访问。在任意一个时刻只允许一个线程对共享资源进行访问，如果有多个线程试图访问公共资源，那么在有一个线程进入后，其他试图访问公共资源的线程将被挂起，并一直等到进入临界区的线程离开，临界区在被释放后，其他线程才可以抢占。
4) 互斥对象：互斥对象和临界区很像，采用互斥对象机制，只有拥有互斥对象的线程才有访问公共资源的权限。因为互斥对象只有一个，所以能保证公共资源不会同时被多个线程同时访问。当前拥有互斥对象的线程处理完任务后必须将线程交出，以便其他线程访问该资源。
5) 信号量：它允许多个线程在同一时刻访问同一资源，但是需要限制在同一时刻访问此资源的最大线程数目。在用 CreateSemaphore()创建信号量时即要同时指出允许的最大资源计数和当前可用资源计数。一般是将当前可用资源计数设置为最 大资源计数，每增加一个线程对共享资源的访问，当前可用资源计数就会减 1 ，只要当前可用资源计数是大于 0 的，就可以发出信号量信号。但是当前可用计数减小 到 0 时则说明当前占用资源的线程数已经达到了所允许的最大数目，不能在允许其他线程的进入，此时的信号量信号将无法发出。线程在处理完共享资源后，应在离 开的同时通过 ReleaseSemaphore （）函数将当前可用资源计数加 1 。在任何时候当前可用资源计数决不可能大于最大资源计数。
6) 事件对象：通过通知操作的方式来保持线程的同步，还可以方便实现对多个线程的优先级比较的操作。

15. 页和段的区别？

1) 页是信息的物理单位，分页是由于系统管理的需要。段是信息的逻辑单位，分段是为了满足用户的要求。

2) 页的大小固定且由系统决定，段的长度不固定，决定于用户所编写的程序，通常由编译程序在对源程序紧进行编译时，根据信息的性质来划分。

3) 分页的作业的地址空间是一维的，程序员只需要利用一个记忆符，即可表示一个地址。分段的作业地址空间则是二维的，程序员在标识一个地址时，既需要给出段名，又需要给出段的地址值。

16. 孤儿进程和僵尸进程的区别？怎么避免这两类进程？守护进程？

1、 一般情况下，子进程是由父进程创建，而子进程和父进程的退出是无顺序的，两者之间都不知道谁先退出。正常情况下父进程先结束会调用 wait 或者 waitpid 函数等待子进程完成再退出，而一旦父进程不等待直接退出，则剩下的子进程会被 init(pid=1)进程接收，成会孤儿进程。（进程树中除了 init 都会有父进程）。

2、 如果子进程先退出了，父进程还未结束并且没有调用 wait 或者 waitpid 函数获取子进程的状态信息，则子进程残留的状态信息（ task_struct 结构和少量资源信息）会变成僵尸进程。

子进程退出时向父进程发送 SIGCHILD 信号，父进程处理 SIGCHILD 信号。在信号处理函数中调用 wait 进行处理僵尸进程。

原理是将子进程成为孤儿进程，从而其的父进程变为 init 进程，通过 init 进程可以处理僵尸进程。

3、 守护进程（ daemon）是指在后台运行，没有控制终端与之相连的进程。它独立于控制终端，通常周期性地执行某种任务 。守护进程脱离于终端是为了避免进程在执行过程中的信息在任何终端上显示并且进程也不会被任何终端所产生的终端信息所打断。

17. 守护进程是什么？怎么实现？

1. 守护进程（Daemon）是运行在后台的一种特殊进程。它独立于控制终端并且周期性地执行某种任务或等待处理某些发生的事件。守护进程是一种很有用的进程。
2. 守护进程特点
   1) 守护进程最重要的特性是后台运行。
   2) 守护进程必须与其运行前的环境隔离开来。这些环境包括未关闭的文件描述符，控制终端，会话和进程组，工作目录以及文件创建掩模等。这些环境通常是守护进程从执行它的父进程（特别是 shell）中继承下来的。
   3) 守护进程的启动方式有其特殊之处。它可以在 Linux 系统启动时从启动脚本/etc/rc.d 中启动，可以由作业规划进程 crond 启动，还可以由用户终端（shell）执行。
3. 实现
   1) 在父进程中执行 fork 并 exit 推出；
   2) 在子进程中调用 setsid 函数创建新的会话；
   3) 在子进程中调用 chdir 函数，让根目录 ”/” 成为子进程的工作目录；
   4) 在子进程中调用 umask 函数，设置进程的 umask 为 0；
   5) 在子进程中关闭任何不需要的文件描述符

18. 线程和进程的区别？线程共享的资源是什么？

1) 一个程序至少有一个进程，一个进程至少有一个线程

2) 线程的划分尺度小于进程，使得多线程程序的并发性高

3) 进程在执行过程中拥有独立的内存单元，而多个线程共享内存，从而极大地提高了程序的运行效率

4) 每个独立的线程有一个程序运行的入口、顺序执行序列和程序的出口。但是线程不能够独立执行，必须依存在应用程序中，由应用程序提供多个线程执行控制

5) 多线程的意义在于一个应用程序中，有多个执行部分可以同时执行。但操作系统并没有将多个线程看做多个独立的应用，来实现进程的调度和管理以及资源分配

6) 一个进程中的所有线程共享该进程的地址空间，但它们有各自独立的（/私有的）栈(stack)，Windows 线程的缺省堆栈大小为 1M。堆(heap)的分配与栈有所不同，一般是一个进程有一个 C 运行时堆，这个堆为本进程中所有线程共享，windows 进程还有所谓进程默认堆，用户也可以创建自己的堆。

线程共享资源	线程独享资源
地址空间	程序计数器
全局变量	寄存器
打开的文件	栈
子进程	状态字
闹铃	
信号及信号服务程序	
记账信息	

线程私有：线程栈，寄存器，程序寄存器

共享：堆，地址空间，全局变量，静态变量

进程私有：地址空间，堆，全局变量，栈，寄存器

共享：代码段，公共数据，进程目录，进程 ID

19. 线程比进程具有哪些优势？

1) 线程在程序中是独立的，并发的执行流，但是，进程中的线程之间的隔离程度要小；
2) 线程比进程更具有更高的性能，这是由于同一个进程中的线程都有共性：多个线程将共享同一个进程虚拟空间；
3) 当操作系统创建一个进程时，必须为进程分配独立的内存空间，并分配大量相关资源；

20. 什么时候用多进程？什么时候用多线程？

1) 需要频繁创建销毁的优先用线程；
2) 需要进行大量计算的优先使用线程；
3) 强相关的处理用线程，弱相关的处理用进程；
4) 可能要扩展到多机分布的用进程，多核分布的用线程；

21. 协程是什么？

1）是一种比线程更加轻量级的存在。正如一个进程可以拥有多个线程一样，一个线程可以拥有多个协程；协程不是被操作系统内核管理，而完全是由程序所控制。
2）协程的开销远远小于线程；
3）协程拥有自己寄存器上下文和栈。协程调度切换时，将寄存器上下文和栈保存到其他地方，在切换回来的时候，恢复先前保存的寄存器上下文和栈。
4）每个协程表示一个执行单元，有自己的本地数据，与其他协程共享全局数据和其他资源。
5）跨平台、跨体系架构、无需线程上下文切换的开销、方便切换控制流，简化编程模型；
6）协程又称为微线程，协程的完成主要靠 yeild 关键字，协程执行过程中，在子程序内部可中断，然后转而执行别的子程序，在适当的时候再返回来接着执行；
7）协程极高的执行效率，和多线程相比，线程数量越多，协程的性能优势就越明显；
8）不需要多线程的锁机制；

22. 递归锁？

1） 线程同步能够保证多个线程安全访问竞争资源，最简单的同步机制是引入互斥锁。互斥锁为资源引入一个状态：锁定/非锁定。某个线程要更改共享数据时，先将其锁定，此时资源的状态为“锁定”，其他线程不能更改；直到该线程释放资源，将资源的状态变成“非锁定”，其他的线程才能再次锁定该资源。互斥锁保证了每次只有一个线程进行写入操作，从而保证了多线程情况下数据的正确性。

2） 读写锁从广义的逻辑上讲，也可以认为是一种共享版的互斥锁。如果对一个临界区大部分是读操作而只有少量的写操作，读写锁在一定程度上能够降低线程互斥产生的代价。

3） Mutex 可以分为递归锁(recursive mutex)和非递归锁(non-recursive mutex)。可递归锁也可称为可重入锁(reentrant mutex)，非递归锁又叫不可重入锁(non-reentrant mutex)。二者唯一的区别是，同一个线程可以多次获取同一个递归锁，不会产生死锁。而如果一个线程多次获取同一个非递归锁，则会产生死锁。

23. 用户态到内核态的转化原理？

1） 系统调用

这是用户态进程主动要求切换到内核态的一种方式，用户态进程通过系统调用申请使用操作系统提供的服务程序完成工作，比如前例中 fork()实际上就是执行了一个创建新进程的系统调用。而系统调用的机制其核心还是使用了操作系统为用户特别开放的一个中断来实现，例如 Linux 的 int 80h 中断。

2) 异常

当 CPU 在执行运行在用户态下的程序时，发生了某些事先不可知的异常，这时会触发由当前运行进程切换到处理此异常的内核相关程序中，也就转到了内核态，比如缺页异常。

3) 外围设备的中断

当外围设备完成用户请求的操作后，会向 CPU 发出相应的中断信号，这时 CPU 会暂停执行下一条即将要执行的指令转而去执行与中断信号对应的处理程序，如果先前执行的指令是用户态下的程序，那么这个转换的过程自然也就发生了由用户态到内核态的切换。比如硬盘读写操作完成，系统会切换到硬盘读写的中断处理程序中执行后续操作等。

24. 中断的实现与作用，中断的实现过程?

① 关中断，进入不可再次响应中断的状态，由硬件实现。

② 保存断点，为了在中断处理结束后能正确返回到中断点。由硬件实现。

③ 将中断服务程序入口地址送 PC，转向中断服务程序。可由硬件实现，也可由软件实现。

④ 保护现场、置屏蔽字、开中断，即保护 CPU 中某些寄存器的内容、设置中断处理次序、允许更高级的中断请求得到响应，实现中断嵌套。由软件实现。

⑤ 设备服务，实际上有效的中断处理工作是在此程序段中实现的。由软件程序实现

⑥ 退出中断。在退出时，又应进入不可中断状态，即关中断、恢复屏蔽字、恢复现场、开中断、中断返回。由软件实现。

25. 系统中断是什么，用户态和内核态的区别

1) 内核态与用户态是操作系统的两种运行级别，当程序运行在 3 级特权级上时，就可以称之为运行在用户态，因为这是最低特权级，是普通的用户进程运行的特权级，大部分用户直接面对的程序都是运行在用户态；反之，当程序运行在 0 级特权级上时，就

可以称之为运行在内核态。运行在用户态下的程序不能直接访问操作系统内核数据结构和程序。当我们在系统中执行一个程序时，大部分时间是运行在用户态下的，在其需要操作系统帮助完成某些它没有权力和能力完成的工作时就会切换到内核态。

2) 这两种状态的主要差别是： 处于用户态执行时，进程所能访问的内存空间和对象受到限制，其所处于占有的处理机是可被抢占的 ； 而处于核心态执行中的进程，则能访问所有的内存空间和对象，且所占有的处理机是不允许被抢占的。

26. CPU 中断

1) CPU 中断是什么

① 计算机处于执行期间；

② 系统内发生了非寻常或非预期的急需处理事件；

③ CPU 暂时中断当前正在执行的程序而转去执行相应的事件处理程序；

④ 处理完毕后返回原来被中断处继续执行；

2) CPU 中断的作用

① 可以使 CPU 和外设同时工作，使系统可以及时地响应外部事件；

② 可以允许多个外设同时工作，大大提高了 CPU 的利用率；

③ 可以使 CPU 及时处理各种软硬件故障。

27. 执行一个系统调用时，OS 发生的过程，越详细越好

1. 执行用户程序(如:fork)

2. 根据 glibc 中的函数实现，取得系统调用号并执行 int $0x80 产生中断。
3. 进行地址空间的转换和堆栈的切换，执行 SAVE_ALL。(进行内核模式)
4. 进行中断处理，根据系统调用表调用内核函数。
5. 执行内核函数。
6. 执行 RESTORE_ALL 并返回用户模式

28. 函数调用和系统调用的区别?

1) 系统调用

① 操作系统提供给用户程序调用的一组特殊的接口。用户程序可以通过这组特殊接

口来获得操作系统内核提供的服务；

② 系统调用可以用来控制硬件；设置系统状态或读取内核数据；进程管理，系统调用接口用来保证系统中进程能以多任务在虚拟环境下运行；

③ Linux 中实现系统调用利用了 0x86 体系结构中的软件中断；

2) 函数调用

① 函数调用运行在用户空间；

② 它主要是通过压栈操作来进行函数调用；

3) 区别

函数库调用	系统调用
在所有的ANSI C编译器版本中，C库函数是相同的	各个操作系统的系统调用是不同的
它调用函数库中的一段程序（或函数）	它调用系统内核的服务
与用户程序相联系	是操作系统的一个入口点
在用户地址空间执行	在内核地址空间执行
它的运行时间属于"用户时间"	它的运行时间属于"系统时间"
属于过程调用，调用开销较小	需要在用户空间和内核上下文环境间切换，开销较大
在C函数库libc中有大约300个函数	在UNIX中大约有90个系统调用
典型的C函数库调用：system fprintf malloc	典型的系统调用：chdir fork write brk；

29. 经典同步问题解法：生产者与消费者问题，哲学家进餐问题，读者写者问题。

30. 虚拟内存？使用虚拟内存的优点？什么是虚拟地址空间？

1) 虚拟内存，虚拟内存是一种内存管理技术，它会使程序自己认为自己拥有一块很大且连续的内存，然而，这个程序在内存中不是连续的，并且有些还会在磁盘上，在需要时进行数据交换；
2) 优点：可以弥补物理内存大小的不足；一定程度的提高反应速度；减少对物理内存的读取从而保护内存延长内存使用寿命；
3) 缺点：占用一定的物理硬盘空间；加大了对硬盘的读写；设置不得当会影响整机稳定性与速度。
4) 虚拟地址空间是对于一个单一进程的概念，这个进程看到的将是地址从 0000 开始的整个内存空间。虚拟存储器是一个抽象概念，它为每一个进程提供了一个假象，好像每一个进程都在独占的使用主存。每个进程看到的存储器都是一致的，称为虚拟地址空间。从最低的地址看起：程序代码和数据，堆，共享库，栈，内核虚拟存储器。大多数计算机的字长都是 32 位，这就限制了虚拟地址空间为 4GB。

31. 线程安全？如何实现？

1) 如果你的代码所在的进程中有多个线程在同时运行，而这些线程可能会同时运行这段代码。如果每次运行结果和单线程运行的结果是一样的，而且其他的变量的值也和预期的是一样的，就是线程安全的。
2) 线程安全问题都是由全局变量及静态变量引起的。
3) 若每个线程中对全局变量、静态变量只有读操作，而无写操作，一般来说，这个全局变量是线程安全的；若有多个线程同时执行写操作，一般都需要考虑线程同步，否则的话就可能影响线程安全。
4) 对于线程不安全的对象我们可以通过如下方法来实现线程安全：

① 加锁 利用 Synchronized 或者 ReenTrantLock 来对不安全对象进行加锁，来实现线程执行的串行化，从而保证多线程同时操作对象的安全性，一个是语法层面的互斥锁，一个是 API 层面的互斥锁.

② 非阻塞同步来实现线程安全。原理就是：通俗点讲，就是先进性操作，如果没有其他线程争用共享数据，那操作就成功了；如果共享数据有争用，产生冲突，那就再采取其他措施(最常见的措施就是不断地重试，知道成功为止)。这种方法需要硬件的支持，因为我们需要操作和冲突检测这两个步骤具备原子性。通常这种指令包括 CAS SC,FAI TAS 等。

③ 线程本地化，一种无同步的方案，就是利用 Threadlocal 来为每一个线程创造一个共享变量的副本来（副本之间是无关的）避免几个线程同时操作一个对象时发生线程安全问题。

32. linux 文件系统

1. 层次分析
   1) 用户层，日常使用的各种程序，需要的接口主要是文件的创建、删除、读、写、关闭等；
   2) VFS 层，文件相关的操作都有对应的 System Call 函数接口，接口调用 VFS 对应的函数；
   3) 文件系统层，用户的操作通过 VFS 转到各种文件系统。文件系统把文件读写命令转化为对磁盘 LBA 的操作，起了一个翻译和磁盘管理的工作；
   4) 缓存层；
   5) 块设备层，块设备接口 Block Device 是用来访问磁盘 LBA 的层级，读写命令组合之后插入到命令队列，磁盘的驱动从队列读命令执行；
   6) 磁盘驱动层；
   7) 磁盘物理层；

2. 读取文件过程

1) 根据文件所在目录的 inode 信息，找到目录文件对应数据块；
2) 根据文件名从数据块中找到对应的 inode 节点信息；
3) 从文件 inode 节点信息中找到文件内容所在数据块块号；
4) 读取数据块内容

33. 常见的 IO 模型，五种？异步 IO 应用场景？有什么缺点？

1) 同步

就是在发出一个功能调用时，在没有得到结果之前，该调用就不返回。**也就是必须一件一件事做**,等前一件做完了才能做下一件事。就是我调用一个功能，该功能没有结束前，我死等结果。

2) 异步

当一个异步过程调用发出后，调用者不能立刻得到结果。实际处理这个调用的部件在完成后，通过状态、通知和回调来通知调用者。就是我调用一个功能，不需要知道该功能结果，该功能有结果后通知我（回调通知）

3) 阻塞

阻塞调用是指调用结果返回之前，当前线程会被挂起（线程进入非可执行状态，在这个状态下，cpu 不会给线程分配时间片，即线程暂停运行）。函数只有在得到结果之后才会返回。对于同步调用来说，很多时候当前线程还是激活的，只是从逻辑上当前函数没有返回而已。 就是调用我（函数），我（函数）没有接收完数据或者没有得到结果之前，我不会返回。

4) 非阻塞

指在不能立刻得到结果之前，该函数不会阻塞当前线程，而会立刻返回。就是调用我（函数），我（函数）立即返回，通过 select 通知调用者。

1) 阻塞 I/O

应用程序调用一个 IO 函数，导致应用程序阻塞，等待数据准备好。 如果数据没有准备好，一直等待….数据准备好了，从内核拷贝到用户空间,IO 函数返回成功指示。

2) 非阻塞 I/O

我们把一个 SOCKET 接口设置为非阻塞就是告诉内核，当所请求的 I/O 操作无法完成时，不要将进程睡眠，而是返回一个错误。这样我们的 I/O 操作函数将不断的测试数据是否已经准备好，如果没有准备好，继续测试，直到数据准备好为止。在这个不断测试的过程中，会大量的占用 CPU 的时间。

3) I/O 复用

I/O 复用模型会用到 select、poll、epoll 函数，这几个函数也会使进程阻塞，但是和阻塞 I/O 所不同的的，这三个函数可以同时阻塞多个 I/O 操作。而且可以同时对多个读操作，多个写操作的 I/O 函数进行检测，直到有数据可读或可写时，才真正调用 I/O 操作函数。

4) 信号驱动 I/O

首先我们允许套接口进行信号驱动 I/O,并安装一个信号处理函数，进程继续运行并不阻塞。当数据准备好时，进程会收到一个 SIGIO 信号，可以在信号处理函数中调用 I/O 操作函数处理数据。

5) 异步 I/O

当一个异步过程调用发出后，调用者不能立刻得到结果。实际处理这个调用的部件在完成后，通过状态、通知和回调来通知调用者的输入输出操作。

34. IO 复用的原理？零拷贝？三个函数？epoll 的 LT 和 ET 模式的理解。

1) IO 复用是 Linux 中的 IO 模型之一，IO 复用就是进程预先告诉内核需要监视的 IO 条件，使得内核一旦发现进程指定的一个或多个 IO 条件就绪，就通过进程进程处理，从而不会在单个 IO 上阻塞了。Linux 中，提供了 select、poll、epoll 三种接口函数来实现 IO 复用。

2) Select

select 的缺点：

① 单个进程能够监视的文件描述符的数量存在最大限制，通常是 1024。由于 select 采用轮询的方式扫描文件描述符，文件描述符数量越多，性能越差；

② 内核/用户空间内存拷贝问题，select 需要大量句柄数据结构，产生巨大开销；

③ Select 返回的是含有整个句柄的数组，应用程序需要遍历整个数组才能发现哪些句柄发生事件；

④ Select 的触发方式是水平触发，应用程序如果没有完成对一个已经就绪的文件描述符进行 IO 操作，那么每次 select 调用还会将这些文件描述符通知进程。

3) Poll

与 select 相比，poll 使用链表保存文件描述符，一你才没有了监视文件数量的限制，但其他三个缺点依然存在

4) Epoll

上面所说的 select 缺点在 epoll 上不复存在，epoll 使用一个文件描述符管理多个描述符，将用户关系的文件描述符的事件存放到内核的一个事件表中，这样在用户空间和内核空间的 copy 只需一次。Epoll 是事件触发的，不是轮询查询的。没有最大的并发连接限制，内存拷贝，利用 mmap（）文件映射内存加速与内核空间的消息传递。

区别总结：

1) 支持一个进程所能打开的最大连接数

① Select 最大 1024 个连接，最大连接数有 FD_SETSIZE 宏定义，其大小是 32 位整数表示，可以改变宏定义进行修改，可以重新编译内核，性能可能会影响；

② Poll 没有最大连接限制，原因是它是基于链表来存储的；

③ 连接数限数有上限，但是很大；

2) FD 剧增后带来的 IO 效率问题

① 因为每次进行线性遍历，所以随着 FD 的增加会造成遍历速度下降，效率降低；

② Poll 同上；

③ 因为 epool 内核中实现是根据每个 fd 上的 callback 函数来实现的，只有活跃的 socket 才会主动调用 callback，所以在活跃 socket 较少的情况下，使用 epoll 没有前面两者的现象下降的性能问题。

3) 消息传递方式

① Select 内核需要将消息传递到用户空间，都需要内核拷贝；

② Poll 同上；

③ Epoll 通过内核和用户空间共享来实现的。

epoll 的 LT 和 ET 模式的理解:

epoll 对文件描述符的操作有两种模式：LT(level trigger)和 ET(edge trigger)，LT 是默认模式。

区别：

LT 模式：当 epoll_wait 检测到描述符事件发生并将此事件通知应用程序，应用程序可以不立即处理该事件。下次调用 epoll_wait 时，会再次响应应用程序并通知此事件。

ET 模式：当 epoll_wait 检测到描述符事件发生并将此事件通知应用程序，应用程序必须立即处理该事件。如果不处理，下次调用 epoll_wait 时，不会再次响应应用程序并通知此事件。

35. Linux 是如何避免内存碎片的

1) 在固定式分区分配中，为将一个用户作业装入内存，内存分配程序从系统分区表中找出一个能满足作业要求的空闲分区分配给作业，由于一个作业的大小并不一定与分区大小相等，因此，分区中有一部分存储空间浪费掉了．由此可知，固定式分区分配中存在**内碎片**.
2) 在可变式分区分配中，为把一个作业装入内存，应按照一定的分配算法从系统中找出一个能满足作业需求的空闲分区分配给作业，如果这个空闲分区的容量比作业申请的空间容量要大，则将该分区一分为二，一部分分配给作业，剩下的部分仍然留作系统的空闲分区。由此可知，可变式分区分配中存在**外碎片**.
3) 伙伴系统
4) 据可移动性组织页避免内存碎片

36. 递归的原理是啥？递归中遇到栈溢出怎么解决

1) 基本原理

第一：每一级的函数调用都有它自己的变量。

第二：每一次函数调用都会有一次返回，并且是某一级递归返回到调用它的那一级，而不是直接返回到 main()函数中的初始调用部分。

第三：递归函数中，位于递归调用前的语句和各级被调函数具有相同的执行顺序。例如在上面的程序中，打印语句#1 位于递归调用语句之前，它按照递归调用的顺序被执行了 4 次，即依次为第一级、第二级、第三级、第四级。

第四：递归函数中，位于递归调用后的语句的执行顺序和各个被调函数的顺序相反。例如上面程序中，打印语句#2 位于递归调用语句之后，其执行顺序依次是：第四级、第三级、第二级、第一级。（递归调用的这种特性在解决涉及到反向顺序的编程问题中很有用，下文会说到）

第五：虽然每一级递归都有自己的变量，但是函数代码不会复制。

第六：递归函数中必须包含终止递归的语句。通常递归函数会使用一个 if 条件语句或其他类似语句一边当函数参数达到某个特定值时结束递归调用，如上面程序的 if(n > 4)。

2) 用递归实现算法时，有两个因素是至关重要的：**递归式**和**递归边界**；
3) 函数调用时通过栈（Stack）来实现的，每当调用一个函数，栈就会加一层栈帧，函数返回就减一层栈帧。而栈资源有限，当递归深度达到一定程度后，就会出现意想不到的结果，比如堆栈溢出；
4) 利用循环函数或者栈加 while 循环来代替递归函数。

37. ++i 是否是原子操作

i++的操作分三步：

（1）栈中取出 i

（2）i 自增 1

（3）将 i 存到栈

所以 i++不是原子操作，上面的三个步骤中任何一个步骤同时操作，都可能导致 i 的值不正确自增

二.++i

在多核的机器上，cpu 在读取内存 i 时也会可能发生同时读取到同一值，这就导致两次自增，实际只增加了一次。

综上，我认为 i++和++i 都不是原子操作。

38. 缺页中断，页表寻址

1) 一个进程对应一个页表，分页存储机制，一个进程对应很多页，执行进程时并不是所有页装入内存中，部分装入内存，当需要的那页不存在内存中，将发生缺页中断，将需要的那页从外存中调入内存中；

2) 页表寻址，页分为页号（从 0 开始编号）与页内偏移地址，两个寄存器，页表基地址寄存器，页表长度寄存器，块表；页的大小相同，内存中的块与页大小相同，页大小相同，页在逻辑上连续在物理上不连续；

3) 调页算法：先进先出，最佳页面置换算法（OPT），最近最久未使用（NRU），最近最少使用置换算法（LRU），先进先出算法（FIFO）会导致 Baley 问题；抖动，页面在内存与外存中的频繁调页；

4) 程序局部性原理，时间局部性、空间局部性；

39. LRU 的实现

1) 用一个数组来存储数据，给每一个数据项标记一个访问时间戳，每次插入新数据项的时候，先把数组中存在的数据的时间戳自增，并将新数据时间戳置为 0 插入到数组中。每次访问数组中的数据项的时候，将被访问的数据项时间戳置为 0。当数组空间已经满时，将时间戳最大的数据项淘汰；

2) 利用一个链表来实现，每次新插入数据的时候将新数据插入到链表头部；每次缓存命中，则将数据移动到链表头部；那么当链表满时，就将链表尾部的数据丢弃；

3) 利用链表和 hashmap。当需要插入新的数据项 的时候，如果新数据命中，则把该节点放到链表头部，如果不存在，则将新数据放在链表头部。若缓存满了，则将链表尾部的节点删除。

40. 内存分区

1) 固态分区，分区大小固定，但并不一定相同；

2) 可变分区，分区大小动态变化，首先适配、最佳适配、最差适配、下一次适配；

41. 伙伴系统相关

1) 伙伴系统是一种经典的内存管理方法。Linux 伙伴系统的引入为内核提供了一种用于**分配一组连续的页**而建立的一种高效的分配策略，并有效的解决了外碎片问题。

2) Linux 中的内存管理的“页”大小为 4KB。把所有的空闲页分组为 11 个块链表，每个块链表分别包含大小为 1，2，4，8，16，32，64，128，256，512 和 1024 个连续页框的页块。最大可以申请 1024 个连续页，对应 4MB 大小的连续内存。每个页块的第一个页的物理地址是该块大小的整数倍。

3) 当向内核**请求分配(2^(i-1)，2^i]**数目的页块时，按照 2^i 页块请求处理。如果对应的块链表中没有空闲页块，则在更大的页块链表中找。当分配的页块中有多余的页时，伙伴系统根据多余的页框大小插入到对应的空闲页块链表中。

   当**释放单页**的内存时，内核将其置于 CPU 高速缓存中，对很可能出现在 cache 的页，则放到“快表”的列表中。在此过程中，内核先判断 CPU 高速缓存中的页数是否超过一定“阈值”，如果是，则将一批内存页还给伙伴系统，然后将该页添加到 CPU 高速缓存中。

   当**释放多页**的块时，内核首先计算出该内存块的伙伴的地址。**内核将满足以下条件的三个块称为伙伴**：(1)两个块具有相同的大小，记作 b。(2)它们的物理地址是**连续**的。(3)第一块的第一个页的**物理地址**是 2*(2^b)的倍数。如果找到了该内存块的伙伴，确保该伙伴的所有页都是空闲的，以便进行合并。内存继续检查合并后页块的“伙伴”并检查是否可以合并，依次类推。

4) 内核将已分配页分为以下三种不同的类型：

   不可移动页：这些页在内存中有固定的位置，不能够移动。

   可回收页：这些页不能移动，但可以删除。内核在回收页占据了太多的内存时或者内存短缺时进行页面回收。

   可移动页：这些页可以任意移动，用户空间应用程序使用的页都属于该类别。它们是通过页表映射的。当它们移动到新的位置，页表项也会相应的更新。

42. I/O 控制方式

1) 直接 I/O（轮询）
   程序查询方式也称为程序轮询方式，该方式采用用户程序直接控制主机与外部设备之

间输入/输出操作。CPU 必须不停地循环测试 I/O 设备的状态端口，当发现设备处于准备好(Ready)状态时，CPU 就可以与 I/O 设备进行数据存取操作。这种方式下的 CPU 与 I/O 设备是串行工作的，输入/输出一般以字节或字为单位进行。这个方式频繁地测试 I/O 设备，I/O 设备的速度相对来说又很慢，极大地降低了 CPU 的处理效率，并且仅仅依靠测试设备状态位来进行数据传送，不能及时发现传输中的硬件错误。

2) 中断

当 I/O 设备结束(完成、特殊或异常)时，就会向 CPU 发出中断请求信号，CPU 收到信号就可以采取相应措施。当某个进程要启动某个设备时，CPU 就向相应的设备控制器发出一条设备 I/O 启动指令，然后 CPU 又返回做原来的工作。CPU 与 I/O 设备可以并行工作，与程序查询方式相比，大大提高了 CPU 的利用率。但是在中断方式下，同程序查询方式一样，也是以字节或字为单位进行。但是该方法大大降低了 CPU 的效率，因为当中断发生的非常频繁的时候，系统需要进行频繁的中断源识别、保护现场、中断处理、恢复现场。这种方法对于以“块”为存取单位的块设备，效率是低下的。

3) DMA

DMA 方式也称为直接主存存取方式，其思想是：允许主存储器和 I/O 设备之间通过“DMA 控制器(DMAC)”直接进行批量数据交换，除了在数据传输开始和结束时，整个过程无须 CPU 的干预。每传输一个“块”数据只需要占用一个主存周期。

4) 通道

通道(Channel)也称为外围设备处理器、输入输出处理机，是相对于 CPU 而言的。是一个处理器。也能执行指令和由指令的程序，只不过通道执行的指令是与外部设备相关的指令。是一种实现主存与 I/O 设备进行直接数据交换的控制方式，与 DMA 控制方式相比，通道所需要的 CPU 控制更少，一个通道可以控制多个设备，并且能够一次进行多个不连续的数据块的存取交换，从而大大提高了计算机系统效率。

43. Spooling 技术

1) 假脱机系统； 在联机的情况下实现的同时外围操作的技术称为 SPOOLing 技术，或称为假脱机技术。

2) 组成

1. **输入井和输出井**:输入井和输出井的存储区域是在**磁盘**上开辟出来的。输入输出井中的数据一般以文件的形式组织管理，这些文件称之为井文件。一个文件仅存放某一个进程的输入或输出数据，所有进程的数据输入或输出文件链接成为一个输入输出队列。
2. **输入缓冲区和输出缓冲区**:输入缓冲区和输出缓冲区的存储区域是在**内存**中开辟出来的。主要用于缓和 CPU 和磁盘之间速度不匹配的矛盾。输入缓冲区用于暂存有输入设备传送的数据，之后再传送到输入井；输出缓冲区 同理。
3. **输入进程和输出进程**:输入进程也称为预输入进程，用于模拟脱机输入时的外围控制机，将用户要求的数据从输入设备传送到输入缓冲区，再存放到输入井。当 CPU 需要的时候，直接从输入井将数据读入内存。反之，输出的同理。
4. **井管理程序**:用于控制作业与磁盘井之间信息的交换。

3) 特点

① **提高了 I/O 的速度:**,对数据执行的 I/O 操作，已从对低速 I/O 设备执行的 I/O 操作演变为对磁盘缓冲区中数据的存取，如同脱机输入输出一样，提高了 I/O 速度，缓和了 CPU 和低速的 I/Os 设备之间速度的不匹配的矛盾。

② **将独占设备改造成了共享设备**:因为在假脱机打印机系统中，实际上并没有为任何进程分配设备，而只是在磁盘缓冲区中为进程分配了一个空闲盘块和建立了一张 I/O 请求表。

③ **实现了虚拟设备功能:**宏观上，对于每一个进程而言，它们认为是自己独占了一个设备，即使实际上是多个进程在同时使用一台独占设备。也可以说，假脱机系统，实现了将独占设备变换为若干台对应的逻辑设备的功能。

44. 通道技术

1) 通道是独立于 CPU，专门用来负责数据输入/输出传输工作的处理机，对外部设备实现统一管理，代替 CPU 对输入/输出操作进行控制，从而使输入，输出操作可与 CPU 并行操作。
2) 引入通道的目的
   为了使 CPU 从 I/O 事务中解脱出来，同时为了提高 CPU 与设备，设备与设备之间的并行工作能力

45. 共享内存的实现

1) 两个不同进程 A、B 共享内存的意思是，同一块物理内存被映射到进程 A、B 各自的进程地址空间。进程 A 可以即时看到进程 B 对共享内存中数据的更新，反之亦然。由于多个进程共享同一块内存区域，必然需要某种同步机制，互斥锁和信号量都可以。
2) 共享内存是通过把同一块内存分别映射到不同的进程空间中实现进程间通信。而共享内存本身不带任何互斥与同步机制，但当多个进程同时对同一内存进行读写操作时会破坏该内存的内容，所以，在实际中，同步与互斥机制需要用户来完成。
3) （1）共享内存就是允许两个不想关的进程访问同一个内存
   （2）共享内存是两个正在运行的进程之间共享和传递数据的最有效的方式
   （3）不同进程之间共享的内存通常安排为同一段物理内存
   （4）共享内存不提供任何互斥和同步机制，一般用信号量对临界资源进行保护。
   （5）接口简单

46. 计一个线程池，内存池

1) 为什么需要线程池
   大多数的网络服务器，包括 Web 服务器都具有一个特点，就是单位时间内必须处理数目巨大的连接请求，但是处理时间却是比较短的。在传统的多线程服务器模型中是这样实现的：一旦有个请求到达，就创建一个新的线程，由该线程执行任务，任务执行

完毕之后，线程就退出。这就是”即时创建，即时销毁”的策略。尽管与创建进程相比，创建线程的时间已经大大的缩短，但是如果提交给线程的任务是执行时间较短，而且执行次数非常频繁，那么服务器就将处于一个不停的创建线程和销毁线程的状态。这笔开销是不可忽略的，尤其是线程执行的时间非常非常短的情况。

2) 线程池原理

在应用程序启动之后，就马上创建一定数量的线程，放入空闲的队列中。这些线程都是处于阻塞状态，这些线程只占一点内存，不占用 CPU。当任务到来后，线程池将选择一个空闲的线程，将任务传入此线程中运行。当所有的线程都处在处理任务的时候，线程池将自动创建一定的数量的新线程，用于处理更多的任务。执行任务完成之后线程并不退出，而是继续在线程池中等待下一次任务。当大部分线程处于阻塞状态时，线程池将自动销毁一部分的线程，回收系统资源。

3) 线程池的作用

需要大量的线程来完成任务，且完成任务的时间比较短；对性能要求苛刻的应用；对性能要求苛刻的应用

4) 内存池的原理

在软件开发中，有些对象使用非常频繁，那么我们可以预先在堆中实例化一些对象，我们把维护这些对象的结构叫“内存池”。在需要用的时候，直接从内存池中拿，而不用从新实例化，在要销毁的时候，不是直接 free/delete，而是返还给内存池。把那些常用的对象存在内存池中，就不用频繁的分配/回收内存，可以相对减少内存碎片，更重要的是实例化这样的对象更快，回收也更快。当内存池中的对象不够用的时候就扩容。

5) 内存池的优缺点

内存池对象不是线程安全的，在多线程编程中，创建一个对象时必须加锁。

Linux

1. Inode 节点

1) Linux 操作系统引进了一个非常重要的概念 inode，中文名为索引结点，引进索引接点是为了在物理内存上找到文件块，所以 inode 中包含文件的相关基本信息，比如文件位置、文件创建者、创建日期、文件大小等待，输入 stat 指令可以查看某个文件的 inode 信息；

2) 硬盘格式化的时候，操作系统自动将硬盘分成两个区域，一个是数据区，一个是 inode 区，存放 inode 所包含的信息，查看每个硬盘分区的 inode 总数和已经使用的数量，可以用 df 命令；

3) 在 linux 系统中，系统内部并不是采用文件名查找文件，而是使用 inode 编号来识别文件。查找文件分为三个过程：系统找到这个文件名对应的 inode 号码，通过 inode 号码获得 inode 信息，根据 inode 信息找到文件数据所在的 block 读取数据；

4) 除了文件名之外的所有文件信息，都存储在 inode 之中。

2. Linux 软连接、硬链接，删除了软连接的源文件软连接可用?

1) 软链接可以看作是 Windows 中的快捷方式，可以让你快速链接到目标档案或目录。硬链接则透过文件系统的 inode 来产生新档名，而不是产生新档案。

2) 软链接（符号链接） ln -s source target
硬链接 （实体链接）ln source target

3) 硬链接(hard link)：A 是 B 的硬链接（A 和 B 都是文件名），则 A 的目录项中的 inode 节点号与 B 的目录项中的 inode 节点号相同，即一个 inode 节点对应两个不同的文件名，两个文件名指向同一个文件，A 和 B 对文件系统来说是完全平等的。如果删除了其中一个，对另外一个没有影响。每增加一个文件名，inode 节点上的链接数增加一，每删除一个对应的文件名，inode 节点上的链接数减一，直到为 0，inode 节点和对应的数据块被回收。注：文件和文件名是不同的东西，rm A 删除的只是 A 这个文件名，而 A 对应的数据块（文件）只有在 inode 节点链接数减少为 0 的时候才会被系统回收。

4) 软链接(soft link)：A 是 B 的软链接（A 和 B 都是文件名），A 的目录项中的 inode 节点号与 B 的目录项中的 inode 节点号不相同，A 和 B 指向的是两个不同的 inode，继而指向两块不同的数据块。但是 A 的数据块中存放的只是 B 的路径名（可以根据这个找到 B 的目录项）。A 和 B 之间是“主从”关系，如果 B 被删除了，A 仍然存在（因为两个是不同的文件），但指向的是一个无效的链接。

5) 硬链接

不能对目录创建硬链接；不能对不同的文件系统创建硬链接；不能对不存在的文件创建硬链接；

6) 软连接

可以对目录创建软连接；可以跨文件系统；可以对不存在的文件创建软连接；

7) 因为链接文件包含有原文件的路径信息，所以当原文件从一个目录下移到其他目录中，再访问链接文件，系统就找不到了，而硬链接就没有这个缺陷，你想怎么移就怎么移；还有它要系统分配额外的空间用于建立新的索引节点和保存原文件的路径。

3. Linux 系统应用程序的内存空间是怎么分配的,用户空间多大，内核空间多大?

1) Linux 内核将这 4G 字节的空间分为两部分。将最高的 1G 字节（从虚拟地址 0xC0000000 到 0xFFFFFFFF），供内核使用，称为“内核空间”。而将较低的 3G 字节（从虚拟地址 0x00000000 到 0xBFFFFFFF），供各个进程使用，称为“用户空间“。因为每个进程可以通过系统调用进入内核，因此，Linux 内核由系统 内的所有进程共享。于是，从具体进程的角度来看，每个进程可以拥有 4G 字节的虚拟空间。

4. Linux 的共享内存如何实现

1) 管道只能在具有亲缘关系的进程间进行通信；通过文件共享，在处理效率上又差一些，而且访问文件描述符不如访问内存地址方便；

2) mmap 内存共享映射，mmap 本来是存储映射功能，它可以将一个文件映射到内存中，在程序里就可以直接使用内存地址对文件内容进行访问；Linux 的 mmap 实现了一种可以在父子进程之间共享内存地址的方式；

3) XSI 共享内存，XSI 是 X/Open 组织对 UNIX 定义的一套接口标准（X/Open System Interface）。XSI 共享内存在 Linux 底层的实现实际上跟 mmap 没有什么本质不同，只是在使用方法上有所区别。

4) POSIX 共享内存，Linux 提供的 POSIX 共享内存，实际上就是在/dev/shm 下创建一个文件，并将其 mmap 之后映射其内存地址即可。

5. 文件处理 grep,awk,sed 这三个命令必知必会

1) grep

grep (global search regular expression(RE) and print out the line,全面搜索正则表达式并把行打印出来)是一种强大的文本搜索工具，它能使用正则表达式搜索文本，并把匹配的行打印出来。常用来在结果中搜索特定的内容。

2) awk

awk 是一个强大的文本分析工具，相对于 grep 的查找，sed 的编辑，awk 在其对数据分析并生成报告时，显得尤为强大。简单来说 awk 就是把文件(或其他方式的输入流，如重定向输入)逐行的读入（看作一个记录集），把每一行看作一条记录，以空格(或\t,或用户自己指定的分隔符)为默认分隔符将每行切片（类似字段），切开的部分再进行各种分析处理。

3) sed

sed 更侧重对搜索文本的处理，如修改、删除、替换等等。sed 主要用来自动编辑一个或多个文件；简化对文件的反复操作；编写转换程序等。

6. 查询进程占用 CPU 的命令

1) top

top 命令可以实时动态地查看系统的整体运行情况，是一个综合了多方信息监测系统性能和运行信息的实用工具。

2) ps

ps 命令就是最基本进程查看命令。使用该命令可以确定有哪些进程正在运行和运行的状态、进程是否结束、进程有没有僵尸、哪些进程占用了过多的资源等等.总之大部分信息都是可以通过执行该命令得到。ps 是显示瞬间进程的状态，并不动态连续；如果想对进程进行实时监控应该用 top 命令。

7. 一个程序从开始运行到结束的完整过程

1) 预处理，主要处理源代码中的预处理指令，引入头文件，去除注释，处理所有的条件编译指令，宏替换，添加行号。经过预处理指令后生成一个.i 文件；

2) 编译，编译过程所进行的是对预处理后的文件进行语法分析、词法分析、符号汇总，然后生成汇编代码。生成.s 文件；

3) 汇编，将汇编文件转换成二进制文件，二进制文件就可以让机器来读取。生成.o 文件；

4) 链接，由汇编程序生成的目标文件并不能立即就被执行，其中可能还有许多没有解决的问题。

8. 一般情况下在 Linux/windows 平台下栈空间的大小

windows 是编译器决定栈的大小，记录在可执行文件中，默认是 1M。linux 是操作系统来决定的，在系统环境变量中设置， ulimit -s 字节数 命令查看修改，但是 linux 默认栈大小为 10M;vs 编译器设置：属性—>设置→链接→输出→栈分配→重新设置；

9. Linux 重定向

1 重定向符号

>	输出重定向到一个文件或设备 覆盖原来的文件
>!	输出重定向到一个文件或设备 强制覆盖原来的文件
>>	输出重定向到一个文件或设备 追加原来的文件
<	输入重定向到一个程序

2 标准错误重定向符号

2>	将一个标准错误输出重定向到一个文件或设备 覆盖原来的文件 b-shell
2>>	将一个标准错误输出重定向到一个文件或设备 追加到原来的文件
2>&1	将一个标准错误输出重定向到标准输出 注释:1 可能就是代表 标准输出
>&	将一个标准错误输出重定向到一个文件或设备 覆盖原来的文件 c-shell
|&	将一个标准错误 管道 输送 到另一个命令作为输入

3 命令重导向示例

在 bash 命令执行的过程中，主要有三种输出入的状况，分别是:

1. 标准输入；代码为 0 ；或称为 stdin ；使用的方式为 <
2. 标准输出：代码为 1 ；或称为 stdout；使用的方式为 1>
3. 错误输出：代码为 2 ；或称为 stderr；使用的方式为 2>

10. Linux 常用命令

1) ls 命令，不仅可以查看 linux 文件包含的文件，而且可以查看文件权限；
2) cd 命令，切换当前目录到 dirName
3) pwd 命令，查看当前工作目录路径；
4) mkdir 命令，创建文件夹
5) rm 命令，删除一个目录中的一个或多个文件或目录
6) rmdir 命令，从一个目录中删除一个或多个子目录项，
7) mv 命令，移动文件或修改文件名
8) cp 命令，将源文件复制至目标文件，或将多个源文件复制至目标目录
9) cat 命令，显示文件内容；
10) touch 命令，创建一个文件
11) vim 命令，
12) which 命令查看可执行文件的位置，whereis 查看文件的位置，find 实际搜寻硬盘查询文件名称；
13) chmod 命令，用于改变 linux 系统文件或目录的访问权限，421，ewr
14) tar 命令，用来压缩和解压文件。tar 本身不具有压缩功能，只具有打包功能，有关压缩及解压是调用其它的功能来完成。
15) chown 命令，将指定文件的拥有者改为指定的用户或组，用户可以是用户名或者用户 ID;
16) ln 命令；
17) grep 命令，强大的文本搜索命令，grep 全局正则表达式搜素；
18) ps 命令，用来查看当前运行的进程状态，一次性查看，如果需要动态连续结果使用 top;
19) top 命令，显示当前系统正在执行的进程的相关信息，包括进程 ID、内存占用率、CPU 占用率等；
20) kill 命令，发送指定的信号到相应进程。不指定型号将发送 SIGTERM（15）终止指定进程。

网络

一、 物理层

二、 数据链路层

三、 网络层

1. 路由器的功能?

路由选择与分组转发

2. ip 报文如何从下向上交付

3. ip 地址有什么用，ip 地址和 mac 地址，为什么需要 IP 地址

1) IP 地址是在网络上分配给每台计算机或网络设备的 32 位数字标识。在 Internet 上，每台计算机或网络设备的 IP 地址是全世界唯一的。IP 地址的格式是 xxx.xxx.xxx.xxx，其中 xxx 是 0 到 255 之间的任意整数。例如，每步站主机的 IP 地址是 219.134.132.131。
2) MAC 地址是数据链路层的地址，如果 mac 地址不可直达 ，直接丢弃，在 LAN 里面，一个网卡的 MAC 地址是唯一的。MAC 地址在 arp 协议里常常用到，mac 地址到 ip 地址的相互转化。Mac 地址是 48 位的地址。
3) IP 地址是网络层的地址，如果 ip 地址不可达，接着转发，在 WAN 里面，ip 地址不唯一，计算机的 ip 地址可以变动

4. ARP 协议的作用

ARP（地址解析）协议是一种解析协议，本来主机是完全不知道这个 IP 对应的是哪个主机的哪个接口，当主机要发送一个 IP 包的时候，会首先查一下自己的 ARP 高速缓存表（最近数据传递更新的 IP-MAC 地址对应表），如果查询的 IP－MAC 值对不存在，那么主机就向网络广播一个 ARP 请求包，这个包里面就有待查询的 IP 地址，而直接收到这份广播的包的所有主机都会查询自己的 IP 地址，如果收到广播包的某一个主机发现自己符合条件，那么就回应一个 ARP 应答包（将自己对应的 IP-MAC 对应地址发回主机），源主机拿到 ARP 应答包后会更新自己的 ARP 缓存表。源主机根据新的 ARP 缓存表准备好数据链路层的的数据包发送工作。

5. NAT 的原理,外网与内网或内网之间的通信中如何区分不同 IP 的数组包

1. **公有 IP 地址**：也叫全局地址，是指合法的 IP 地址，它是由 NIC（网络信息中心）或者 ISP(网络服务提供商)分配的地址，对外代表一个或多个内部局部地址，是全球统一的可寻 址的地址。
2. **私有 IP 地址**：也叫内部地址，属于非注册地址，专门为组织机构内部使用。因特网分配编号委员会（IANA）保留了 3 块 IP 地址做为私有 IP 地址；
3. NAT 英文全称是“Network Address Translation”，中文意思是“网络地址转换”，它是一个 IETF(Internet Engineering Task Force, Internet 工程任务组)标准，允许一个整体机构以一个公用 IP（Internet Protocol）地址出现在 Internet 上。顾名思义，它是一种把内部私有网络地址（IP 地址）翻译成合法网络 IP 地址的技术，如下图所示。因此我们可以认为，NAT 在一定程度上，能够有效的解决公网地址不足的问题。
4. NAT 就是在局域网内部网络中使用内部地址，而当内部节点要与外部网络进行通讯时，就在网关（可以理解为出口，打个比方就像院子的门一样）处，将内部地址替换成公用地址，从而在外部公网（internet）上正常使用，NAT 可以使多台计算机共享 Internet 连接，这一功能很好地解决了公共 IP 地址紧缺的问题。通过这种方法，可以只申请一个合法 IP 地址，就把整个局域网中的计算机接入 Internet 中。这时，NAT 屏蔽了内部网络，所有内部网计算机对于公共网络来说是不可见的，而内部网计算机用户通常不会意识到 NAT 的存在。

6. RIP 路由协议

1. 网络中的每一个路由器都要维护从它自己到其他每一个目标网络的距离记录；
2. 距离也称为跳数，规定从一路由器到直接连接的网络跳数为 1，而每经过一个路由器，则距离加 1；
3. RIP 认为好的路由就是它通过的路由器数量最少；
4. RIP 允许一条路径上最多有 15 个路由器，因为规定最大跳数为 16；
5. RIP 默认每 30 秒广播一次 RIP 路由更新信息。

每一个路由表项目包括三个内容：目的网络、距离、下一跳路由器

1、对地址为 X 的路由器发过来的路由表，先修改此路由表中的所有项目：把”下一跳”字段中的地址改为 X，并把所有”距离”字段都加 1。

2、对修改后的路由表中的每一个项目，进行以下步骤：

2.1、将 X 的路由表(修改过的)，与 S 的路由表的目的网络进行对比。

若在 X 中出现，在 S 中没出现，则将 X 路由表中的这一条项目添加到 S 的路由表中。

2.2、对于目的网络在 S 和 X 路由表中都有的项目进行下面步骤

2.2.1、在 S 的路由表中，若下一跳地址是 x

则直接用 X 路由表中这条项目替换 S 路由表中的项目。

2.2.2、在 S 的路由表中，若下一跳地址不是 x
若 X 路由表项目中的距离 d 小于 S 路由表中的距离，则进行更新。

3、若 3 分钟还没有收到相邻路由器的更新表，则把此相邻路由器记为不可到达路由器，即把距离设置为 16。

7. 为什么使用 IP 地址通信

1) 由于全世界存在着各式各样的网络，它们使用不同的硬件地址。要使这些异构网络能够互相通信就必须进行非常复杂的硬件地址转换工作，因此几乎是不可能的事。
2) 连接到因特网的主机都拥有统一的 IP 地址，它们之间的通信就像连接在同一个网络上那样简单方便，因为调用 ARP 来寻找某个路由器或主机的硬件地址都是由计算机软件自动进行的，对用户来说是看不见这种调用过程的。

8. 子网掩码有什么用?

子网掩码是一种用来指明一个 IP 地址所标示的主机处于哪个子网中。子网掩码不能单独存在，它必须结合 IP 地址一起使用。子网掩码只有一个作用，就是将某个 IP 地址划分成网络地址和主机地址两部分。

9. 子网划分的方法

1) 传统子网划分，ip 地址结构=网络号+主机号
2) 子网掩码
3) CIDR，减少了传统分法的 ip 浪费。

四、 运输层

1. TCP 协议有几大计时器?

1) 重传计时器
在一个 TCP 连接中，TCP 每发送一个报文段，就对此报文段设置一个超时重传计时器。若在收到了对此特定报文段的确认之前计时器截止期到，则重传此报文段，并将计时器复位。
2) 持续计时器
为了对付零窗口大小通知，TCP 需要另一个计时器。假定接收 TCP 宣布了窗口大小为零。发送 TCP 就停止传送报文段，直到接收 TCP 发送确认并宣布一个非零的窗口大小。但这个确认可能会丢失。我们知道在 TCP 中，对确认是不需要发送确认的。若确认丢失了，接收 TCP 并不知道，而是会认为它已经完成任务了，并等待着发送 TCP 接着会发送更多的报文段。但发送 TCP 由于没有收到确认，就等待对方发送确认来通知窗口的大小。双方的 TCP 都在永远地等待着对方。要打开这种死锁，TCP

为每一个连接使用一个坚持计时器。当发送 TCP 收到一个窗口大小为零的确认时，就启动坚持计时器。当坚持计时器期限到时，发送 TCP 就发送一个特殊的报文段，叫做探测报文段。这个报文段只有一个字节的数据。它有一个序号，但它的序号永远不需要确认；甚至在计算对其他部分的数据的确认时该序号也被忽略。探测报文段提醒对端：确认已丢失，必须重传。

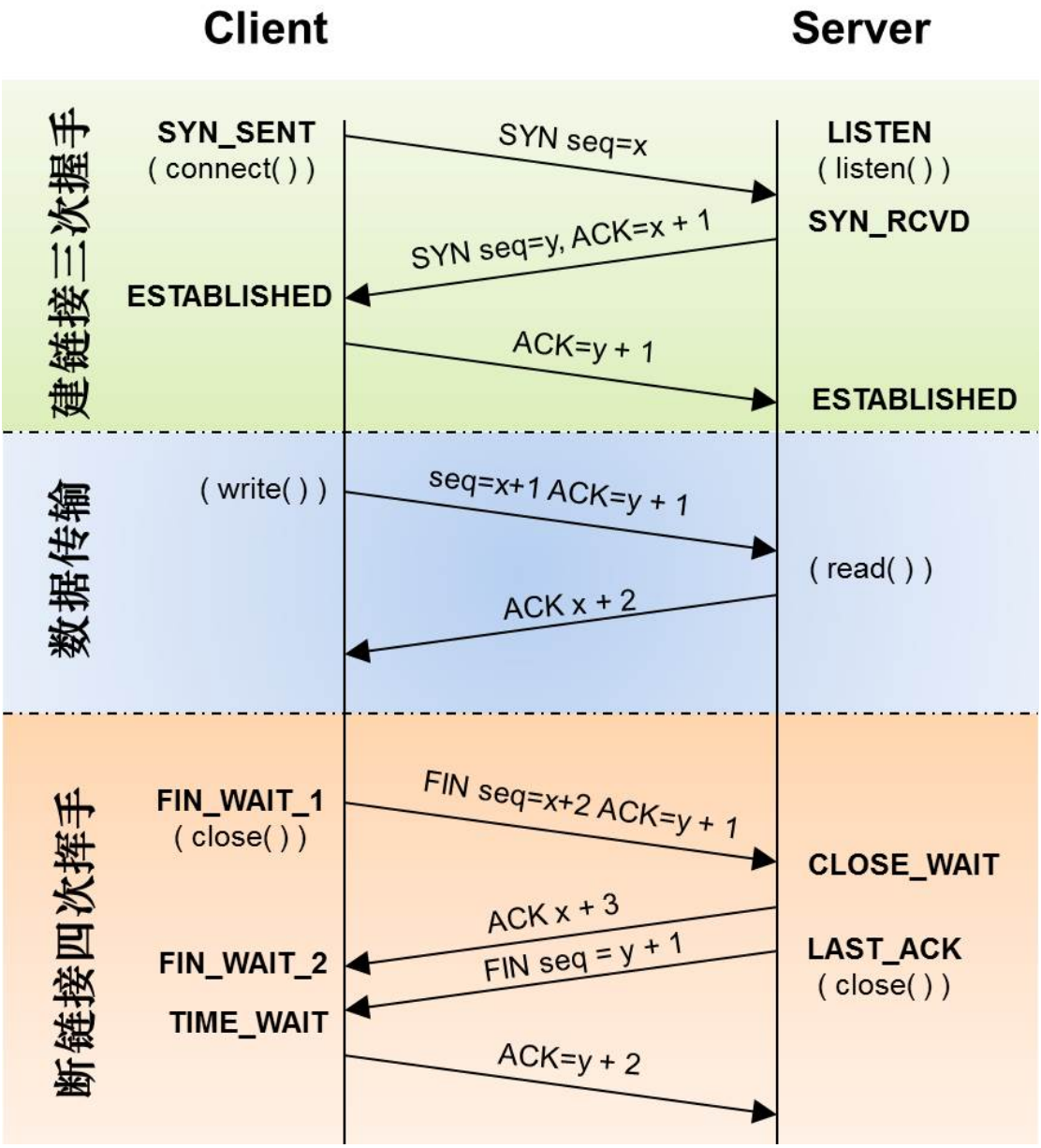
3) 保活计时器

保活计时器使用在某些实现中，用来防止在两个 TCP 之间的连接出现长时期的空闲。假定客户打开了到服务器的连接，传送了一些数据，然后就保持静默了。也许这个客户出故障了。在这种情况下，这个连接将永远地处理打开状态。

4) 时间等待计时器

时间等待计时器是在连接终止期间使用的。当 TCP 关闭一个连接时，它并不认为这个连接马上就真正地关闭了。在时间等待期间中，连接还处于一种中间过渡状态。这就可以使重复的 FIN 报文段（如果有的话）可以到达目的站因而可将其丢弃。这个计时器的值通常设置为一个报文段的寿命期待值的两倍。

2. 详细说一下 TCP 协议，三次握手传输的内容？13 种状态



1) 第一次握手：建立连接。客户端发送连接请求报文段，将 SYN 位置为 1，Sequence Number 为 x；然后，客户端进入 SYN_SEND 状态，等待服务器的确认；

2) 第二次握手：服务器收到 SYN 报文段。服务器收到客户端的 SYN 报文段，需要对这个 SYN 报文段进行确认，设置 Acknowledgment Number 为 x+1(Sequence Number+1)；同时，自己自己还要发送 SYN 请求信息，将 SYN 位置为 1，Sequence Number 为 y；服务器端将上述所有信息放到一个报文段（即 SYN+ACK 报文段）中，一并发送给客户端，此时服务器进入 SYN_RECV 状态；

3) 第三次握手：客户端收到服务器的 SYN+ACK 报文段。然后将 Acknowledgment Number 设置为 y+1，向服务器发送 ACK 报文段，这个报文段发送完毕以后，客户端和服务器

端都进入 ESTABLISHED 状态，完成 TCP 三次握手。

那四次分手呢？

当客户端和服务器通过三次握手建立了 TCP 连接以后，当数据传送完毕，肯定是要断开 TCP 连接的啊。那对于 TCP 的断开连接，这里就有了神秘的“四次分手”。

1) 第一次分手：主机 1（可以使客户端，也可以是服务器端），设置 Sequence Number 和 Acknowledgment Number，向主机 2 发送一个 FIN 报文段；此时，主机 1 进入 FIN_WAIT_1 状态；这表示主机 1 没有数据要发送给主机 2 了；

2) 第二次分手：主机 2 收到了主机 1 发送的 FIN 报文段，向主机 1 回一个 ACK 报文段，Acknowledgment Number 为 Sequence Number 加 1；主机 1 进入 FIN_WAIT_2 状态；主机 2 告诉主机 1，我“同意”你的关闭请求；

3) 第三次分手：主机 2 向主机 1 发送 FIN 报文段，请求关闭连接，同时主机 2 进入 LAST_ACK 状态；

4) 第四次分手：主机 1 收到主机 2 发送的 FIN 报文段，向主机 2 发送 ACK 报文段，然后主机 1 进入 TIME_WAIT 状态；主机 2 收到主机 1 的 ACK 报文段以后，就关闭连接；此时，主机 1 等待 2MSL 后依然没有收到回复，则证明 Server 端已正常关闭，那好，主机 1 也可以关闭连接了。

5) 六大标志位

SYN，同步标志位；ACK 确认标志位；PSH 传送标志位；FIN 结束标志位；RST 重置标志位；URG 紧急标志位；seq 序号；ack 确认号

3. TCP 为啥挥手要比握手多一次？

因为当处于 LISTEN 状态的服务器端收到来自客户端的 SYN 报文(客户端希望新建一个 TCP 连接)时，它可以把 ACK(确认应答)和 SYN(同步序号)放在同一个报文里来发送给客户端。但在关闭 TCP 连接时，当收到对方的 FIN 报文时，对方仅仅表示对方已经没有数据发送给你了，但是你自己可能还有数据需要发送给对方，则等你发送完剩余的数据给对方之后，再发送 FIN 报文给对方来表示你数据已经发送完毕，并请求关闭连接，所以通常情况下，这里的 ACK 报文和 FIN 报文都是分开发送的。

4. 为什么一定进行三次握手？

当客户端向服务器端发送一个连接请求时，由于某种原因长时间驻留在网络节点中，无法达到服务器端，由于 TCP 的超时重传机制，当客户端在特定的时间内没有收到服务器端的确认应答信息，则会重新向服务器端发送连接请求，且该连接请求得到服务器端的响应并正常建立连接，进而传输数据，当数据传输完毕，并释放了此次 TCP 连接。若此

时第一次发送的连接请求报文段延迟了一段时间后，到达了服务器端，本来这是一个早已失效的报文段，但是服务器端收到该连接请求后误以为客户端又发出了一次新的连接请求，于是服务器端向客户端发出确认应答报文段，并同意建立连接。如果没有采用三次握手建立连接，由于服务器端发送了确认应答信息，则表示新的连接已成功建立，但是客户端此时并没有向服务器端发出任何连接请求，因此客户端忽略服务器端的确认应答报文，更不会向服务器端传输数据。而服务器端却认为新的连接已经建立了，并在一直等待客户端发送数据，这样服务器端一直处于等待接收数据，直到超出计数器的设定值，则认为服务器端出现异常，并且关闭这个连接。在这个等待的过程中，浪费服务器的资源。如果采用三次握手，客户端就不会向服务器发出确认应答消息，服务器端由于没有收到客户端的确认应答信息，从而判定客户端并没有请求建立连接，从而不建立该连接。

5. TCP 与 UDP 的区别？应用场景都有哪些？

1) TCP 面向连接（如打电话要先拨号建立连接）;UDP 是无连接的，即发送数据之前不需要建立连接
2) TCP 提供可靠的服务。也就是说，通过 TCP 连接传送的数据，无差错，不丢失，不重复，且按序到达;UDP 尽最大努力交付，即不保证可靠交付 Tcp 通过校验和，重传控制，序号标识，滑动窗口、确认应答实现可靠传输。如丢包时的重发控制，还可以对次序乱掉的分包进行顺序控制。
3) UDP 具有较好的实时性，工作效率比 TCP 高，适用于对高速传输和实时性有较高的通信或广播通信。
4) 每一条 TCP 连接只能是点到点的;UDP 支持一对一，一对多，多对一和多对多的交互通信
5) TCP 对系统资源要求较多，UDP 对系统资源要求较少。
6) 若通信数据完整性需让位与通信实时性，则应该选用 TCP 协议（如文件传输、重要状态的更新等）；反之，则使用 UDP 协议（如视频传输、实时通信等）。
7) UDP:DNS SNMP
8) TCP 面向字节流，UTP 面向数据包；

6. 为什么 UDP 有时比 TCP 更有优势?

1) 网速的提升给 UDP 的稳定性提供可靠网络保障，丢包率很低，如果使用应用层重传，能够确保传输的可靠性。

2) TCP 为了实现网络通信的可靠性，使用了复杂的拥塞控制算法，建立了繁琐的握手过程，由于 TCP 内置的系统协议栈中，极难对其进行改进。

3) 采用 TCP，一旦发生丢包，TCP 会将后续的包缓存起来，等前面的包重传并接收到后再继续发送，延时会越来越大，基于 UDP 对实时性要求较为严格的情况下，采用

自定义重传机制，能够把丢包产生的延迟降到最低，尽量减少网络问题对游戏性造成影响。

7. UDP 中一个包的大小最大能多大

1) 以太网(Ethernet)数据帧的长度必须在 46-1500 字节之间,这是由以太网的物理特性决定的.这个 1500 字节被称为链路层的 MTU(最大传输单元).但这并不是指链路层的长度被限制在 1500 字节,其实这这个 MTU 指的是链路层的数据区.
2) 并不包括链路层的首部和尾部的 18 个字节.所以,事实上,这个 1500 字节就是网络层 IP 数据报的长度限制.因为 IP 数据报的首部为 20 字节,所以 IP 数据报的数据区长度最大为 1480 字节.
3) 而这个 1480 字节就是用来放 TCP 传来的 TCP 报文段或 UDP 传来的 UDP 数据报的.又因为 UDP 数据报的首部 8 字节,所以 UDP 数据报的数据区最大长度为 1472 字节.这个 1472 字节就是我们可以使用的字节数。

8. TCP 粘包

1) 在 socket 网络程序中，TCP 和 UDP 分别是面向连接和非面向连接的。因此 TCP 的 socket 编程，收发两端（客户端和服务器端）都要有成对的 socket，因此，发送端为了将多个发往接收端的包，更有效的发到对方，使用了优化方法（Nagle 算法），将多次间隔较小、数据量小的数据，合并成一个大的数据块，然后进行封包。这样，接收端，就难于分辨出来了，必须提供科学的拆包机制。

2) 对于 UDP，不会使用块的合并优化算法，这样，实际上目前认为，是由于 UDP 支持的是一对多的模式，所以接收端的 skbuff(套接字缓冲区）采用了链式结构来记录每一个到达的 UDP 包，在每个 UDP 包中就有了消息头（消息来源地址，端口等信息），这样，对于接收端来说，就容易进行区分处理了。所以 UDP 不会出现粘包问题

1) TCP 粘包是指发送方发送的若干包数据到接收方接收时粘成一包，从接收缓冲区看，后一包数据的头紧接着前一包数据的尾；
2) 发送方原因

   我们知道，TCP 默认会使用 Nagle 算法。而 Nagle 算法主要做两件事：1）只有上一个分组得到确认，才会发送下一个分组；2）收集多个小分组，在一个确认到来时一起发送。所以，正是 Nagle 算法造成了发送方有可能造成粘包现象。

3) 接收方原因

   TCP 接收到分组时，并不会立刻送至应用层处理，或者说，应用层并不一定会立即处理；实际上，TCP 将收到的分组保存至接收缓存里，然后应用程序主动从缓存里读收到的分组。这样一来，如果 TCP 接收分组的速度大于应用程序读分组的速度，多个包就会被存至缓存，应用程序读时，就会读到多个首尾相接粘到一起的包。

4) 解决方法

① **发送方**

对于发送方造成的粘包现象，我们可以通过关闭 Nagle 算法来解决，使用 TCP_NODELAY 选项来关闭 Nagle 算法。

② **接收方**

遗憾的是 TCP 并没有处理接收方粘包现象的机制，我们只能在应用层进行处理。

③ **应用层处理**

应用层的处理简单易行！并且不仅可以解决接收方造成的粘包问题，还能解决发送方造成的粘包问题。

9. 传输层功能

传输进程到进程的逻辑通信，即所说的端到端的通信，而网络层完成主机到主机之间的逻辑通信；

10. TCP 可靠性保证

1. 序号
   TCP 首部的序号字段用来保证数据能有序提交给应用层，TCP 把数据看成无结构的有序的字节流。数据流中的每一个字节都编上一个序号字段的值是指本报文段所发送的数据的第一个字节序号。
2. 确认
   TCP 首部的确认号是期望收到对方的下一个报文段的数据的第一个字节的序号；
3. 重传
   超时重传
   冗余 ACK 重传
4. 流量控制
   TCP 采用大小可变的滑动窗口进行流量控制，窗口大小的单位是字节。
   发送窗口在连接建立时由双方商定。但在通信的过程中，接收端可根据自己的资源情况，随时动态地调整对方的发送窗口上限值(可增大或减小)。
   1) 窗口
      接受窗口 rwnd，接收端缓冲区大小。接收端将此窗口值放在 TCP 报文的首部中的窗口字段，传送给发送端。
      拥塞窗口 cwnd，发送缓冲区大小。
      发送窗口 swnd，发送窗口的上限值 = Min [rwnd, cwnd]
5. 拥塞控制

6. 流量控制与拥塞控制的区别

所谓拥塞控制就是防止过多的数据注入到网络中，这样可以使网络中的路由器或链路不致过载。拥塞控制所要做的都有一个前提，就是网络能承受现有的网络负荷。流量控制往往指的是点对点通信量的控制，是个端到端的问题。流量控制所要做的就是控制发送端发送数据的速率，以便使接收端来得及接受。

11. 拥塞控制

1) 慢开始

发送方维持一个叫做**拥塞窗口** cwnd（congestion window）的状态变量。拥塞窗口的大小取决于网络的拥塞程度，并且动态地在变化。发送方让自己的发送窗口等于拥塞窗口，另外考虑到接受方的接收能力，发送窗口可能小于拥塞窗口。慢开始算法的思路就是，不要一开始就发送大量的数据，先探测一下网络的拥塞程度，也就是说由小到大逐渐增加拥塞窗口的大小。

当然收到单个确认但此确认多个数据报的时候就加相应的数值。所以一次传输轮次之后拥塞窗口就加倍。这就是乘法增长，和后面的拥塞避免算法的加法增长比较。

为了防止 cwnd 增长过大引起网络拥塞，还需设置一个慢开始门限 ssthresh 状态变量。ssthresh 的用法如下：

当 cwnd<ssthresh 时，使用慢开始算法。

当 cwnd>ssthresh 时，改用拥塞避免算法。

当 cwnd=ssthresh 时，慢开始与拥塞避免算法任意。

拥塞避免算法让拥塞窗口缓慢增长，即每经过一个往返时间 RTT 就把发送方的拥塞窗口 cwnd 加 1，而不是加倍。这样拥塞窗口按线性规律缓慢增长。

无论是在**慢开始阶段**还是在**拥塞避免阶段**，只要发送方判断网络出现拥塞（其根据就是没有收到确认，虽然没有收到确认可能是其他原因的分组丢失，但是因为无法判定，所以都当做拥塞来处理），就把慢开始门限设置为出现拥塞时的发送窗口大小的一半。然后把拥塞窗口设置为 1，执行慢开始算法。如下图：

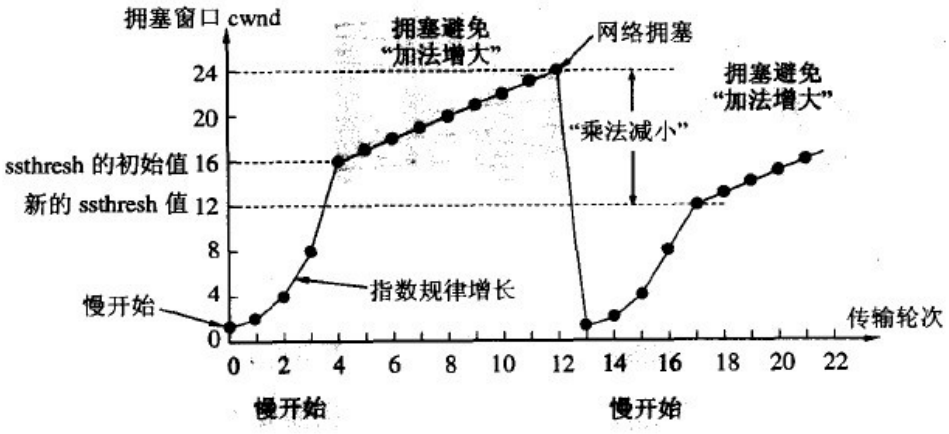


图 5-25　慢开始和拥塞避免算法的实现举例 net/sicofield

2)　快重传和快恢复

快重传要求接收方在收到一个失序的报文段后就立即发出重复确认（为的是使发送方及早知道有报文段没有到达对方）而不要等到自己发送数据时捎带确认。快重传算法规定，发送方只要一连收到三个重复确认就应当立即重传对方尚未收到的报文段，而不必继续等待设置的重传计时器时间到期。

快重传配合使用的还有快恢复算法，有以下两个要点：

①当发送方连续收到三个重复确认时，就执行“乘法减小”算法，把 ssthresh 门限减半。但是接下去并不执行慢开始算法。

②考虑到如果网络出现拥塞的话就不会收到好几个重复的确认，所以发送方现在认为网络可能没有出现拥塞。所以此时不执行慢开始算法，而是将 cwnd 设置为 ssthresh 的大小，然后执行拥塞避免算法。如下图：

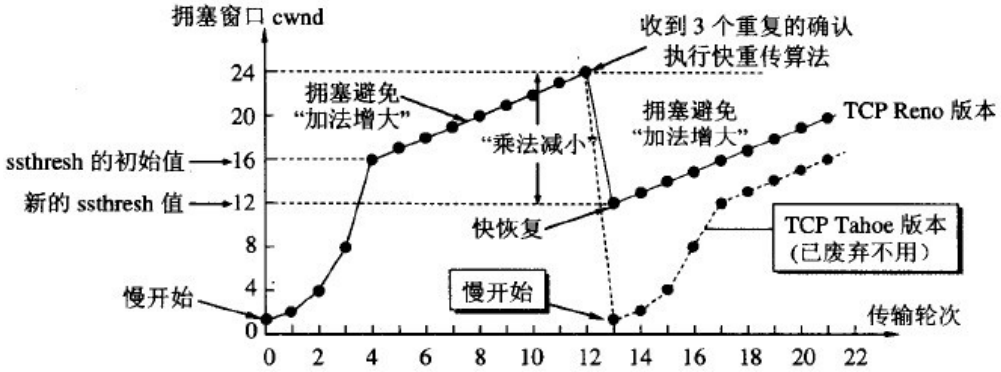


图 5-27　从连续收到三个重复的确认转入拥塞避免 net/sicofield

12. TCP 流量控制

1) 如果发送方把数据发送得过快，接收方可能会来不及接收，这就会造成数据的丢失。TCP 的流量控制是利用滑动窗口机制实现的，接收方在返回的 ACK 中会包含自己的接收窗口的大小，以控制发送方的数据发送。

2) 当某个 ACK 报文丢失了，就会出现 A 等待 B 确认，并且 B 等待 A 发送数据的死锁状态。为了解决这种问题，TCP 引入了持续计时器（Persistence timer），当 A 收到 rwnd=0 时，就启用该计时器，时间到了则发送一个 1 字节的探测报文，询问 B 是很忙还是上个 ACK 丢失了，然后 B 回应自身的接收窗口大小，返回仍为 0（A 重设持续计时器继续等待）或者会重发 rwnd=x。

13. 流量控制与拥塞控制的区别？

1) 拥塞控制就是防止过多的数据注入网络中，这样可以使网络中的路由器或链路不致过载。拥塞控制是一个全局性的过程，和流量控制不同，流量控制指点对点通信量的控制。
2) 所谓流量控制就是让发送发送速率不要过快，让接收方来得及接收。利用滑动窗口机制就可以实施流量控制。原理这就是运用 TCP 报文段中的窗口大小字段来控制，发送方的发送窗口不可以大于接收方发回的窗口大小。

14. time_wait 与 close_wait，time_wait 状态持续多长时间？为什么会有 time_wait 状态？

1) time_wait 另一边已经初始化一个释放，close_wait 连接一端被动关闭；
2) 首先调用 close()发起主动关闭的一方，在发送最后一个 ACK 之后会进入 time_wait 的状态，也就说该发送方会保持 2MSL 时间之后才会回到初始状态。MSL 指的是数据包在网络中的最大生存时间。产生这种结果使得这个 TCP 连接在 2MSL 连接等待期间，定义这个连接的四元组（客户端 IP 地址和端口，服务端 IP 地址和端口号）不能被使用。
3) 为什么存在 time_wait

   ① TCP 协议在关闭连接的四次握手过程中，最终的 ACK 是由主动关闭连接的一端（后面统称 A 端）发出的，如果这个 ACK 丢失，对方（后面统称 B 端）将重发出最终的 FIN，因此 A 端必须维护状态信息（TIME_WAIT）允许它重发最终的 ACK。如果 A 端不维持 TIME_WAIT 状态，而是处于 CLOSED 状态，那么 A 端将响应 RST 分节，B 端收到后将此分节解释成一个错误。因而，要实现 TCP 全双工连接的正常终止，必须处理终止过程中四个分节任何一个分节的丢失情况，主动关闭连接的 A 端必须维持 TIME_WAIT 状态 。

为实现 TCP 全双工连接的可靠释放

由 TCP 状态变迁图可知，假设发起主动关闭的一方（client）最后发送的 ACK 在网络中丢失，由于 TCP 协议的重传机制，执行被动关闭的一方（server）将会重发其 FIN，在该 FIN 到达 client 之前，client 必须维护这条连接状态，也就说这条 TCP 连接所对应的资源（client 方的 local_ip,local_port）不能被立即释放或重新分配，直到另一方重发的 FIN 达到之后，client 重发 ACK 后，经过 2MSL 时间周期没有再收到另一方的 FIN 之后，该 TCP 连接才能恢复初始的 CLOSED 状态。如果主动关闭一方不维护这样一个 TIME_WAIT 状态，那么当被动关闭一方重发的 FIN 到达时，主动关闭一方的 TCP 传输层会用 RST 包响应对方，这会被对方认为是有错误发生，然而这事实上只是正常的关闭连接过程，并非异常。

② TCP segment 可能由于路由器异常而“迷途”，在迷途期间，TCP 发送端可能因确认超时而重发这个 segment，迷途的 segment 在路由器修复后也会被送到最终目的地，这个迟到的迷途 segment 到达时可能会引起问题。在关闭“前一个连接”之后，马上又重新建立起一个相同的 IP 和端口之间的“新连接”，“前一个连接”的迷途重复分组在“前一个连接”终止后到达，而被“新连接”收到了。为了避免这个情况，TCP 协议不允许处于 TIME_WAIT 状态的连接启动一个新的可用连接，因为 TIME_WAIT 状态持续 2MSL，就可以保证当成功建立一个新 TCP 连接的时候，来自旧连接重复分组已经在网络中消逝。

为使旧的数据包在网络因过期而消失

为说明这个问题，我们先假设 TCP 协议中不存在 TIME_WAIT 状态的限制，再假设当前有一条 TCP 连接：(local_ip, local_port, remote_ip,remote_port)，因某些原因，我们先关闭，接着很快以相同的四元组建立一条新连接。本文前面介绍过，TCP 连接由四元组唯一标识，因此，在我们假设的情况中，TCP 协议栈是无法区分前后两条 TCP 连接的不同的，在它看来，这根本就是同一条连接，中间先释放再建立的过程对其来说是“感知”不到的。这样就可能发生这样的情况：前一条 TCP 连接由 local peer 发送的数据到达 remote peer 后，会被该 remot peer 的 TCP 传输层当做当前 TCP 连接的正常数据接收并向上传递至应用层（而事实上，在我们假设的场景下，这些旧数据到达 remote peer 前，旧连接已断开且一条由相同四元组构成的新 TCP 连接已建立，因此，这些旧数据是不应该被向上传递至应用层的），从而引起数据错乱进而导致各种无法预知的诡异现象。作为一种可靠的传输协议，TCP 必须在协议层面考虑并避免这种情况的发生，这正是 TIME_WAIT 状态存在的第 2 个原因。

4) 如果 time_wait 维持的时间过长，主动关闭连接端迟迟无法关闭连接，占用程序资源。

5) 如果服务器程序 TCP 连接一直保持在 CLOSE_WAIT 状态，那么只有一种情况，就是在对方关闭连接之后服务器程序自己没有进一步发出 ack 信号。换句话

说，就是在对方连接关闭之后，程序里没有检测到，或者程序压根就忘记了这个时候需要关闭连接，于是这个资源就一直被程序占着。

6) time_wait 状态如何避免

首先服务器可以设置 SO_REUSEADDR 套接字选项来通知内核，如果端口忙，但 TCP 连接位于 TIME_WAIT 状态时可以重用端口。在一个非常有用的场景就是，如果你的服务器程序停止后想立即重启，而新的套接字依旧希望使用同一端口，此时 SO_REUSEADDR 选项就可以避免 TIME_WAIT 状态。

Close_wait:

1) 产生原因

在被动关闭连接情况下，在已经接收到 FIN，但是还没有发送自己的 FIN 的时刻，连接处于 CLOSE_WAIT 状态。通常来讲，CLOSE_WAIT 状态的持续时间应该很短，正如 SYN_RCVD 状态。但是在一些特殊情况下，就会出现连接长时间处于 CLOSE_WAIT 状态的情况。出现大量 close_wait 的现象，主要原因是某种情况下对方关闭了 socket 链接，但是我方忙与读或者写，没有关闭连接。代码需要判断 socket，一旦读到 0，断开连接，read 返回负，检查一下 errno，如果不是 AGAIN，就断开连接。对方关闭连接之后服务器程序自己没有进一步发出 ack 信号。换句话说，就是在对方连接关闭之后，程序里没有检测到，或者程序压根就忘记了这个时候需要关闭连接，于是这个资源就一直被程序占着。

2) 解决方法

要检测出对方已经关闭的 socket，然后关闭它。

15. Time_wait 为什么是 2MSL 的时间长度

TIME_WAIT 的状态是为了等待连接上所有的分组的消失。单纯的想法，发送端只需要等待一个 MSL 就足够了。这是不够的，假设现在一个 MSL 的时候，接收端需要发送一个应答，这时候，我们也必须等待这个应答的消失，这个应答的消失也是需要一个 MSL，所以我们需要等待 2MSL。

16. 介绍一下 ping 的过程，分别用到了哪些协议

17. socket 编程

TCP 过程：

客户端：

1) 创建 socket
2) 绑定 ip、端口号到 socket 字
3) 连接服务器，connect()
4) 收发数据，send()、recv()
5) 关闭连接

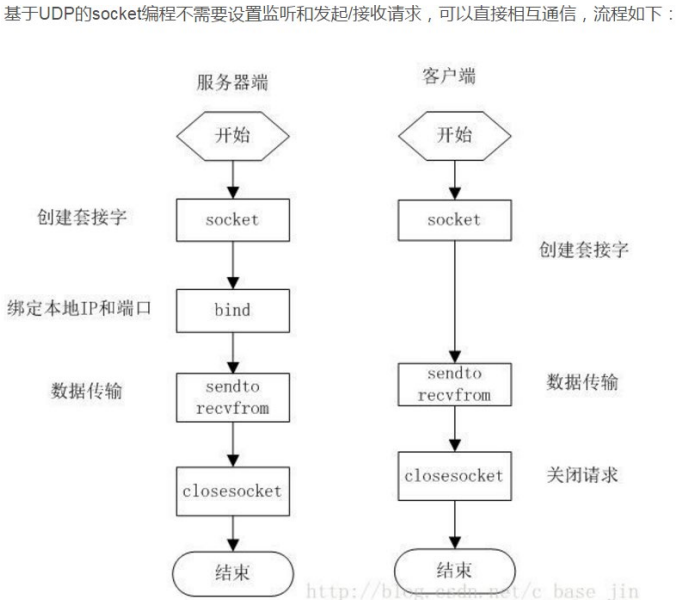
服务器端：

1) 创建 socket 字

2) 设置 socket 属性
3) 绑定 ip 与端口号
4) 开启监听，listen()
5) 接受发送端的连接 accept()
6) 收发数据 send()、recv()
7) 关闭网络连接
8) 关闭监听

对应关系：
客户端的 connect()指向服务器端的 accept()
客户端、服务器端的 send()/recv()是双向箭头的关系。

UDP 过程：



18. 客户端为什么不需要 bind

19. send 和 recv 的缺点

五、 应用层

1. 常用的网络协议?

1) DHCP

动态主机设置协议（Dynamic Host Configuration Protocol，DHCP）是一个局域网的网络协议，使用 UDP 协议工作，主要有两个用途：给内部网络或网络服务供应商自动分配 IP 地址给用户给内部网络管理员作为对所有计算机作中央管理的手段

2） ARP

将 32 位的 IP 地址转换为 48 位的物理地址。当路由器或主机选择了某条路由时，首先会查找 ARP 缓存，若缓存中有对应 IP 地址的物理地址，则以此封装以太帧，否则会广播（为二层广播）ARP 报文，每个主机接收到 ARP 请求报文后，会缓存发送源的 IP——MAC 对到 ARP 缓存中，目的主机会发送 ARP 回应（此时为单播），当发送源接收到回应时，会将目的方的 IP——MAC 对存放在 ARP 缓存中。在点到点的物理连接中，是不会用到 ARP 报文的，在启动时双方都会通告对方自己的 IP 地址，此时物理层的封装不需要 MAC 地址。windows 上可以使用 arp -a 查看本机的 ARP 缓存。ARP 缓存中的每个条目的最大存活时间为 20 分钟

3） ICMP

ICMP（Internet Control Message Protocol）因特网控制报文协议。它是 IPv4 协议族中的一个子协议，用于 IP 主机、路由器之间传递控制消息。控制消息是在网络通不通、主机是否可达、路由是否可用等网络本身的消息。这些控制消息虽然不传输用户数据，但是对于用户数据的传递起着重要的作用。
ICMP 协议与 ARP 协议不同，ICMP 靠 IP 协议来完成任务，所以 ICMP 报文中要封装 IP 头部。它与传输层协议（如 TCP 和 UDP）的目的不同，一般不用来在端系统之间传送数据，不被用户网络程序直接使用，除了想 Ping 和 Tracert 这样的诊断程序。

2. 网络协议各个层的网络设备？

一、集线器

集线器也称 HUB，工作在 OSI 七层结构的第一层物理层，属于共享型设备，接收数据广播发出，在局域网内一般都是星型连接拓扑结构，每台工作站都连接到集线器上。由于集线器的带宽共享特性导致网络利用效率极低，一般在大中型的网络中不会使用到集线器。现在的集线器基本都是全双工模式，市面上常见的集线器传输速率普遍都为 100Mbps。

二、中继器

中继器（Repeater）工作于 OSI 的第一层（物理层），中继器是最简单的网络互联设备，连接同一个网络的两个或多个网段，主要完成物理层的功能，负责在两个网络节点的物理层上按位传递信息，完成信号的复制、调整和放大功能，以此从而增加信号传输的距离，延长网络的长度和覆盖区域，支持远距离的通信。

一般来说，中继器两端的网络部分是网段，而不是子网。中继器只将任何电缆段上的数据发送到另一段电缆上，并不管数据中是否有错误数据或不适于网段的数据。大家最常接触的是网络中继器，在通讯上还有微波中继器、激光中继器、红外中继器等等，机理类似，触类旁通。

三、交换机

交换机顾名思义以交换为主要功能，工作在 OSI 第二层（数据链路层），根据 MAC 地址进行数据转发。交换机的每一个端口都属于一个冲突域，而集线器所有端口属于一个冲突域。交换机通过分析 Ethernet 包的包头信息（其中包含了源 MAC 地址、目标 MAC 地址、信息长度等），取得目标 MAC 地址后，查找交换机中存储的地址对照表（MAC 地址对应的端口），确认具有此 MAC 地址的网卡连接在哪个端口上，然后将信包送到对应端口，有效的抑制 IP 广播风暴。并且信息包处于并行状态，效率较高。

交换机的转发延迟非常小，主要的得益于其硬件设计机理非常高效，为了支持各端口的最大数据传输速率，交换机内部转发信包的背板带宽都必须远大于端口带宽，具有强大的整体吞吐率，才能为每台工作站提供更高的带宽和更高的网络利用率，可以满足大型网络环境大量数据并行处理 的要求。

四、网桥

网桥和交换机一样都是工作在 OSI 模型的第二层（数据链路层），可以看成是一个二层路由器（真正的路由器是工作在网络层，根据 IP 地址进行信包转发）。网桥可有效的将两个局域网（LAN）连起来，根据 MAC 地址（物理地址）来转发帧，使本地通信限制在本网段内，并转发相应的信号至另一网段，网桥通常用于联接数量不多的、同一类型的网段。

五、路由器

路由器跟集线器和交换机不同，是工作在 OSI 的第三层（网络层），根据 IP 进行寻址转发数据包。路由器是一种可以连接多个网络或网段的网络设备，能将不同网络

或网段之间（比如局域网——大网）的数据信息进行转换，并为信包传输分配最合适的路径，使它们之间能够进行数据传输，从而构成一个更大的网络。

路由器具有最主要的两个功能，即数据通道功能和控制功能。数据通道功能包括转发决定、背板转发以及输出链路调度等，一般由特定的硬件来完成；控制功能一般用软件来实现，包括与相邻路由器之间的信息交换、系统配置、系统管理等。

六、网关

网关(Gateway)又叫协议转换器，网关的概念实际上跟上面的设备型不是一类问题，但是为了方便参考还是放到这里一并介绍。

网关是一种复杂的网络连接设备，可以支持不同协议之间的转换，实现不同协议网络之间的互连。网关具有对不兼容的高层协议进行转换的能力，为了实现异构设备之间的通信，网关需要对不同的链路层、专用会话层、表示层和应用层协议进行翻译和转换。所以网关兼有路由器、网桥、中继器的特性。

若要使两个完全不同的网络（异构网）连接在一起，一般使用网关，在 Internet 中两个网络也要通过一台称为网关的计算机实现互联。这台计算机能根据用户通信目标计算机的 IP 地址，决定是否将用户发出的信息送出本地网络，同时，它还将外界发送给属于本地网络计算机的信息接收过来，它是一个网络与另一个网络相联的通道。为了使 TCP/IP 协议能够寻址，该通道被赋予一个 IP 地址，这个 IP 地址称为网关地址。

所以，网关的作用就是将两个使用不同协议的网络段连接在一起的设备，对两 个网络段中的使用不同传输协议的数据进行互相的翻译转换。在互连设备中，由于协议转换的复杂性，一般只能进行一对一的转换，或是少数几种特定应用协议的转换。

3. 讲讲浏览器输入地址后发生的全过程，以及对应的各个层次的过程

1、域名解析：浏览器获得 URL 地址，向操作系统请求该 URL 对应的 IP 地址，操作系统查询 DNS（首先查询本地 HOST 文件，没有则查询网络）获得对应的 IP 地址

解释：

把 URL 分割成几个部分：协议、网络地址、资源路径

协议：指从该计算机获取资源的方式，常见的是 HTTP、FTP

网络地址：可以是域名或者是 IP 地址，也可以包括端口号，如果不注明端口号，默认是 80 端口

如果地址不是一个 IP 地址，则需要通过 DNS（域名系统）将该地址解析成 IP 地址，IP 地址对应着网络上的一台计算机，DNS 服务器本身也有 IP，你的网络设置包含 DNS 服务器的 IP，例如，www.abc.com 不是一个 IP，则需要向 DNS 询问请求 www.abc.com 对应的 IP，获得 IP，在这个过程中，你的电脑直接询问 DNS 服务器可能没有发现 www.abc.com 对应的 IP，就会向它的上级服务器询问，这样依次一层层向上级找，最高可达根节点，直到找到或者全部找不到为止

端口号就相当于银行的窗口，不同的窗口负责不同的服务，如果输入 www.abc.com:8080/，则表示不使用默认的 80 端口，而使用指定的 8080 端口

2、确认好了 IP 和端口号，则可以向该 IP 地址对应的服务器的该端口号发起 TCP 连接请求

3、服务器接收到 TCP 连接请求后，回复可以连接请求，

4、浏览器收到回传的数据后，还会向服务器发送数据包，表示三次握手结束

5、三次握手成功后，开始通讯，根据 HTTP 协议的要求，组织一个请求的数据包，里面包含请求的资源路径、你的身份信息等，例如，www.abc.com/images/1/表示的资源路径是 images/1/，发送后，服务器响应请求，将数据返回给浏览器，数据可以是根据 HTML 协议组织的网页，里面包含页面的布局、文字等等，也可以是图片或者脚本程序等，如果资源路径指定的资源不存在，服务器就会返回 404 错误，如果返回的是一个页面，则根据页面里的一些外链 URL 地址，重复上述步骤，再次获取

6、渲染页面，并开始响应用户的操作

7、窗口关闭时，浏览器终止与服务器的连接

4. http 与 https 工作方式

1） http 包含如下动作：

① 浏览器打开一个 TCP 连接；

② 浏览器发送 HTTP 请求到服务器；

③ 服务器发送 HTTP 回应信息到服务器；

④ TCP 连接关闭；

2) SSL 包含如下动作：

① 验证服务器端；

② 允许客户端和服务器端选择加密算法和密码，确保双方都支持；

③ 验证客户端；

④ 使用公钥加密技术来生成共享加密数据；

⑤ 创建一个加密的 SSL 连接；

⑥ 基于该 SSL 连接传递 HTTP 请求；

5. http 协议，http 和 https 的区别

1) HTTP 和 HTTPS 的基本概念

HTTP：是互联网上应用最为广泛的一种网络协议，是一个客户端和服务器端请求和应答的标准（TCP），用于从 WWW 服务器传输超文本到本地浏览器的传输协议，它可以使浏览器更加高效，使网络传输减少。

HTTPS：是以安全为目标的 HTTP 通道，简单讲是 HTTP 的安全版，即 HTTP 下加入 SSL 层，HTTPS 的安全基础是 SSL，因此加密的详细内容就需要 SSL。

HTTPS 协议的主要作用可以分为两种：一种是建立一个信息安全通道，来保证数据传输的安全；另一种就是确认网站的真实性。

2) HTTPS 和 HTTP 的区别主要如下：

a) https 协议需要到 ca 申请证书，一般免费证书较少，因而需要一定费用。

b) http 是超文本传输协议，信息是明文传输，https 则是具有安全性的 ssl 加密传输协议。

c) http 和 https 使用的是完全不同的连接方式，用的端口也不一样，前者是 80，后者是 443。

d) http 的连接很简单，是无状态的；HTTPS 协议是由 SSL+HTTP 协议构建的可进行加密传输、身份认证的网络协议，比 http 协议安全。

e) 在 OSI 模型中，HTTP 工作于应用层，而 HTTPS 工作于传输层；

6. http 状态码

1) 1XX 信息码，服务器收到请求，需要请求者继续执行操作；
2) 2XX 成功码，操作被成功接收并处理；
3) 3XX 重定向，需要进一步的操作以完成请求；
4) 4XX 客户端错误，请求包含语法错误或无法完成请求；
5) 5XX 服务器错误，服务器在处理请求的过程中发生了错误

404 服务器无法根据客户端的请求找到资源（网页）。通过此代码，网站设计人员可设置"您所请求的资源无法找到"的个性页面

7. HTTP1.0 与 HTTP1.1 的区别?

1) HTTP1.0 需要使用 keep-alive 参数来告知服务器要建立一个长连接，而 HTTP1.1 默认支持长连接；
2) HTTP 1.1 支持只发送 header 信息(不带任何 body 信息)，如果服务器认为客户端有权限请求服务器，则返回 100，否则返回 401；
3) HTTP1.0 是没有 host 域的，HTTP1.1 才支持这个参数；
4) HTTP2.0 使用了多路复用的技术，做到同一个连接并发处理多个请求，而且并发请求的数量比 HTTP1.1 大了好几个数量级；
5) HTTP1.1 不支持 header 数据的压缩，HTTP2.0 使用 HPACK 算法对 header 的数据进行压缩，这样数据体积小了，在网络上传输就会更快；
6) 对支持 HTTP2.0 的 web server 请求数据的时候，服务器会顺便把一些客户端需要的资源一起推送到客户端，免得客户端再次创建连接发送请求到服务器端获取。这种方式非常合适加载静态资源

8. OSI 7 层网络模型中各层的名称及其作用？

OSI 层	功能	TCP/IP 协议
应用层(Application layer)	文件传输，电子邮件，文件服务，虚拟终端	TFTP，HTTP，SNMP，FTP，SMTP，DNS，Telnet
表示层(Presentation layer)	数据格式化，代码转换，数据加密	没有协议
会话层(Session layer)	解除或建立与其他接点的联系	没有协议
传输层(Transport layer)	提供端对端的接口	TCP，UDP
网络层(Network layer)	为数据包选择路由	IP，ICMP，RIP，OSPF，BGP，IGMP
数据链路层(Data link layer)	传输有地址的帧，错误检测功能	SLIP，CSLIP，PPP，ARP，RARP，MTU
物理层(Physical layer)	以二进制数据形式在物理媒体上传输数据	ISO2110，IEEE802，IEEE802.2

9. TCP/IP 4 层网络模型名称及其作用？

OSI七层网络模型		Linux TCP/IP概念层	对应网络协议	相应措施
应用层（Application）			TFTP、FTP、NFS、WAIS	
表示层(Presentation)		应用层	Telnet、rlogin、SNMP、Gopher	Linux应用命令测试
会话层(Session)			SMTP、DNS	
传输层(Transport)	⇔	传输层	TCP、UDP	TCP、UDP协议分析
网络层(Network)	⇔	网际层	IP、ICMP、ARP、RARP、AKP、UUCP	检查IP地址、路由设置
数据链路层(Data Link)	⇔	网络接口	FDDI、Ethernet、Arpanet、PDN、SLP、PPP	ARP地址检测、物理连接检测
物理层(Physical)			IEEE 802.1A、IEEE 802.2到IEEE 802.11	

10. OSI 7 层网络中各层的常见协议以及协议作用？设备

第一层：物理层(PhysicalLayer)

规定通信设备的机械的、电气的、功能的和过程的特性，用以建立、维护和拆除物理链路连接。具体地讲，机械 特性规定了网络连接时所需接插件的规格尺寸、引脚数量和排列情况等;电气特性规定了在物理连接上传输 bit 流时线路上信号电平的大小、阻抗匹配、传输速率 距离限制等;功能特性是指对各个信号先分配确切的信号含义，即定义了 DTE 和 DCE 之间各个线路的功能;规程特性定义了利用信号线进行 bit 流传输的一组 操作规程，是指在物理连接的建立、维护、交换信息是，DTE 和 DCE 双放在各电路上的动作系列。在这一层，数据的单位称为比特(bit)。属于物理层定义的典型规范代表包括：EIA/TIA RS-232、EIA/TIA RS-449、V.35、RJ-45 等。

第二层：数据链路层(DataLinkLayer)

在物理层提供比特流服务的基础上，建立相邻结点之间的数据链路，通过差错控制提供数据帧(Frame)在信道上无差错的传输，并进行各电路上的动作系列。数据链路层在不可靠的物理介质上提供可靠的传输。该层的作用包括：物理地址寻址、数据的成帧、流量控制、数据的检错、重发等。在这一层，数据的单位称为帧(frame)。数据链路层协议的代表包括：SDLC、HDLC、PPP、STP、帧中继等。

第三层是网络层

在 计算机网络中进行通信的两个计算机之间可能会经过很多个数据链路，也可能还要经过很多通信子网。网络层的任务就是选择合适的网间路由和交换结点， 确保数据及时传送。网络层将数据链路层提供的帧组成数据包，包中封装有网络层包头，其中含有逻辑地址信息- -源站点和目的站点地址的网络地址。如 果你在谈论一个 IP 地址，那么你是在处理第 3 层的问题，这是“数据包”问题，而不是第 2 层的“帧”。IP 是第 3 层问题的一部分，此外还有一些路由协议和地 址解析协议(ARP)。有关路由的一切事情都在这第 3 层处理。地址解析和路由是 3 层的重要目的。网络层还可以实现拥塞控制、网际互连等功能。在这一层，数据的单位称为数据包(packet)。网络层协议的代表包括：IP、IPX、RIP、OSPF 等。

第 四层是处理信息的传输层

第 4 层的数据单元也称作数据包(packets)。但是，当你谈论 TCP 等具体的协议时又有特殊的叫法，TCP 的数据单元称为段 (segments)而 UDP 协议的数据单元称为“数据报(datagrams)”。这个层负责获取全部信息，因此，它必须跟踪数据单元碎片、乱序到达的 数据包和其它在传输过程中可能发生的危险。第 4 层为上层提供端到端(最终用户到最终用户)的透明的、可靠的数据传输服务。所为透明的传输是指在通信过程中 传

输层对上层屏蔽了通信传输系统的具体细节。传输层协议的代表包括：TCP、UDP、SPX 等。

第五层是会话层

这一层也可以称为会晤层或对话层，在会话层及以上的高层次中，数据传送的单位不再另外命名，而是统称为报文。会话层不参与具体的传输，它提供包括访问验证和会话管理在内的建立和维护应用之间通信的机制。如服务器验证用户登录便是由会话层完成的。

第六层是表示层

这一层主要解决拥护信息的语法表示问题。它将欲交换的数据从适合于某一用户的抽象语法，转换为适合于 OSI 系统内部使用的传送语法。即提供格式化的表示和转换数据服务。数据的压缩和解压缩， 加密和解密等工作都由表示层负责。

第七层应用层

应用层为操作系统或网络应用程序提供访问网络服务的接口。应用层协议的代表包括：Telnet、FTP、HTTP、SNMP 等

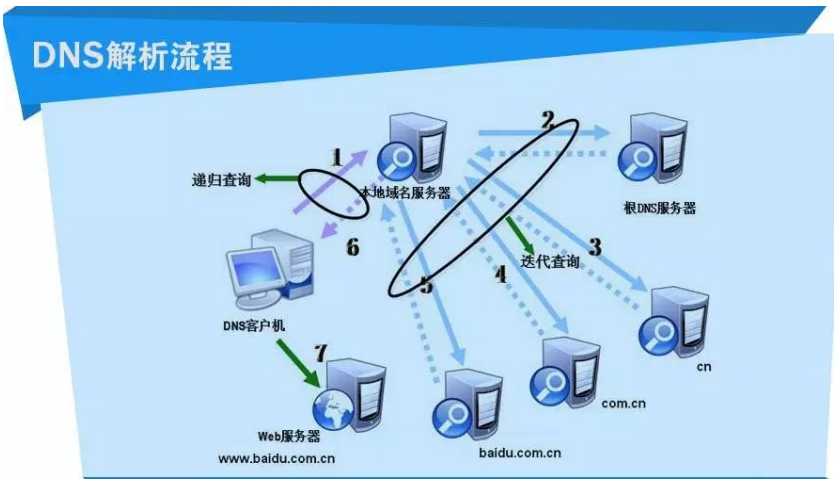
11. OSI 与 TCP 模型的区别?

1) TCP/IP 协议中的应用层处理开放式系统互联模型中的第五层、第六层和第七层的功能；

2) TCP/IP 协议中的传输层并不能总是保证在传输层可靠地传输数据包，而开放式系统互联模型可以做到。TCP/IP 协议还提供一项名为 UDP(用户数据报协议)的选择。UDP 不能保证可靠的数据包传输。

12. DNS 是干什么的？？

1) 主机解析域名的顺序
   找缓存、找本机的 hosts 文件、找 DNS 服务器
2) DNS 协议运行在 UDP 协议之上，使用端口号 53
3) 根服务器：ISP 的 DNS 服务器还找不到的话，它就会向根服务器发出请求，进行递归查询（DNS 服务器先问根域名服务器.com 域名服务器的 IP 地址，然后再问.com 域名服务器，依次类推）



十个过程：

① 浏览器先检查自身缓存中有没有被解析过这个域名对应的 ip 地址；

② 如果浏览器缓存没有命中，浏览器会检查操作系统缓存中有没有对应的已解析过的结果。在 windows 中可通过 c 盘里 hosts 文件来设置；

③ 还没命中，请求本地域名服务器来解析这个域名，一般都会在本地域名服务器找到；

④ 本地域名服务器没有命中，则去根域名服务器请求解析；

⑤ 根域名服务器返回给本地域名服务器一个所查询域的主域名服务器；

⑥ 本地域名服务器向主域名服务器发送请求；

⑦ 接受请求的主域名服务器查找并返回这个域名对应的域名服务器的地址；

⑧ 域名服务器根据映射关系找到 ip 地址，返回给本地域名服务器；

⑨ 本地域名服务器缓存这个结果；

⑩ 本地域名服务器将该结果返回给用户；

13. get/post 区别

1) 后退按钮或刷新，Get 无害，post 数据会被重新提交；
2) Get 所使用的 URL 可以被设置为书签，而 post 不可以；
3) Get 能够被缓存，而 post 不可以；
4) Get 参数保留在浏览器历史中，而 post 参数不会保留在浏览器历史中；
5) 当发生数据时，get 方法向 URL 添加数据，URL 的数据长度是受限的，而 post 没有数据长度限制；

6) Get 只允许 ASCII 编码，而 post 没有限制；
7) Get 安全性没有 post 安全性好；
8) Get 数据在 URL 中对所有人是可见的，而在 post 中数据不会显示在 URL 中。
9) Get 产生一个 TCP 数据包，post 产生两个 TCP 数据包；对于 get 方式的请求，浏览器会把 header 和 data 一并发送出去；对于 post，浏览器先发送 header 再发送 data；
10) GET 和 POST 本质上就是 TCP 链接，并无差别。但是由于 HTTP 的规定和浏览器/服务器的限制，导致他们在应用过程中体现出一些不同；

14.说一下网卡从接收到数据后发生了什么

15.send 函数什么情况下会阻塞

16.常用端口号

FTP 21;TELNET 23;SMTP 25;DNS 53;HTTP 80;HTTPS 443

17.https 的过程

数据结构

1. 常用查找算法？具体实现

2. 常用排序算法？具体实现，哪些是稳定的，时间复杂度、空间复杂度，快速排序非递归如何实现？快排的优势？

3. 图的常用算法？

1) 深度广度遍历；
2) 广度优先遍历；
3) 最短路径 Floyed 算法；
4) 最短路径 Dijktral 算法；
5) 最小生成树，Prime 算法；
6) 最小生成树 Kuraul 算法；

4. 哈夫曼编码?

1) 给定 n 个权值作为 n 个叶子结点，构造一棵二叉树，若带权路径长度达到最小，称这样的二叉树为最优二叉树，也称为哈夫曼树(Huffman Tree)。哈夫曼树是带权路径长度最短的树，权值较大的结点离根较近。
2) 哈夫曼树的构造
   将节点权重进行升序排序；
   选择权重最小的两个节点，将两个节点的和作为新节点，将新节点加入数组中；
   重复以上步骤，最后数组中剩下一个值，该值就是树的带权路径长度，即哈夫曼树；

5. ***AVL 树、B+树、红黑树、B 树 B+树区别，B+树应用在哪里?

1) 一个 m 阶的 B+树具有如下特征:
   ① 有 k 个子树的中间节点包含有 k 个元素（B 树中是 k-1 个元素），每个元素不保存数据，只用来索引，所有数据都保存在叶子节点。
   ② 所有的叶子结点中包含了全部元素的信息，及指向含这些元素记录的指针，且叶子结点本身依关键字的大小自小而大顺序链接。
   ③ 所有的中间节点元素都同时存在于子节点，在子节点元素中是最大（或最小）元素。
   ④ 在 B+树中，只有叶子节点带有数据，其余中间节点仅仅是索引，没有关联任何数据。
2) B+树的优势
   ① 单一节点存储更多的元素，使得查询的 IO 次数更少；
   ② 所有查询都要查询到叶子节点，查询性能稳定；
   ③ 所有叶子节点形成有序链表，便于范围查询；
3) B+树与 B-树的区别

6. 为什么使用红黑树，什么情况使用 AVL 树。红黑树比 AVL 树有什么优点。

1) 首先红黑树是不符合 AVL 树的平衡条件的，即每个节点的左子树和右子树的高度最多差 1 的二叉查找树。但是提出了为节点增加颜色，红黑是用非严格的平衡来换取增删

节点时候旋转次数的降低，任何不平衡都会在三次旋转之内解决，而 AVL 是严格平衡树，因此在增加或者删除节点的时候，根据不同情况，旋转的次数比红黑树要多。所以红黑树的插入效率更高!!! 红黑树的查询性能略微逊色于 AVL 树，因为他比 avl 树会稍微不平衡最多一层，也就是说红黑树的查询性能只比相同内容的 avl 树最多多一次比较，但是，红黑树在插入和删除上完爆 avl 树，avl 树每次插入删除会进行大量的平衡度计算，而红黑树为了维持红黑性质所做的红黑变换和旋转的开销，相较于 avl 树为了维持平衡的开销要小得多。

2) 如果你的应用中，搜索的次数远远大于插入和删除，那么选择 AVL，如果搜索，插入删除次数几乎差不多，应该选择 RB。

7. 单链表如何判断有环?

1) 在链表头部设置两个指针，一个每次向后移动两个位置，一个每次向后移动一个位置。两个指针遍历链表过程中，如果快指针到达链表尾还没有和慢指针相遇，说明链表无环，反之有环；
2) 环的长度，从两个指针的交点开始，移动指针，当指针再次指向两个指针的交点的时候，就可以求出环的长度；
3) 环的入口，一个指针从头开始，一个指针指向(1)中的环中快慢指针的交点，开始遍历，直到这两个指针相遇，两个指针相遇的点就是环的入口点。

8. 如何判断一个图是否连同?

可以用 DFS（O（v^2））和 BFS(O(v+e))的思想都能实现，只要从一个点出发，然后判断是否能遍历完所有的点。

9. hash 用在什么地方，解决 hash 冲突的几种方法?负载因子?

1) 如何构造哈希函数
   a) 数字分析法；
   b) 平方取中法；
   c) 除留余数法；
   d) 伪随机数法；
2) 处理冲突
   e) 线性探测；
   f) 二次探测；
   g) 伪随机数探测；
   h) 拉链探测。
3) 如果负载因子是默认的 0.75，HashMap(16)的时候，占 16 个内存空间，实际上只用到了 12 个，超过 12 个就扩容。
   如果负载因子是 1 的话，HashMap(16)的时候，占 16 个内存空间，实际上会填满 16 个以后才会扩容。增大负载因子可以减少 hash 表的内存，如果负载因子是 0.75，hashmap(16)最多可以存储 12 个元素，想存第 16 个就得扩容成 32。如果负载因子

是 1，hashmap(16)最多可以存储 16 个元素。同样存 16 个元素，一个占了 32 个空间，一个占了 16 个空间的内存。

10. n 个节点的二叉树的所有不同构的个数

11. 二叉树的公共祖先，排序二叉树的公共祖先

1) 搜索二叉树

从树的根节点开始和两个节点进行比较，如果根节点大于两个节点值，则去根节点的左孩子去进行查找；如果根节点小于两个节点值，则去根节点的右孩子去进行查找；当根节点大于其中一个节点，小于其中一个节点，则该节点是最近的祖先节点。

方法一首先给出 node1 的父节点 node1->_parent，然后将 node1 的所有父节点依次和 node2->parent 作比较，如果发现两个节点相等，则该节点就是最近公共祖先，直接将其返回。如果没找到相等节点，则将 node2 的所有父节点依次和 node1->_parent->_parent 作比较.....直到 node1->_parent==NULL。

方法二给定的两个节点都含有父节点，因此，可将这两个节点看做是两个链表的头结点，将求两个节点的最近公共祖先节点转化为求两链表的交点，这两个链表的尾节点都是根节点。

2) 一般二叉树

方法一，将从根节点到 node1 的路径保存在数组中；将从根节点到 node2 的路径保存在数组中；两个数组从头开始遍历，直到找到不相同的两个值，该值前一个就是最近祖先；

方法二，从根节点开始遍历，如果 node1 和 node2 中的任一个和 root 匹配，那么 root 就是最低公共祖先。 如果都不匹配，则分别递归左、右子树，如果有一个 节点出现在左子树，并且另一个节点出现在右子树，则 root 就是最低公共祖先. 如果两个节点都出现在左子树，则说明最低公共祖先在左子树中，否则在右子树。

12. 节点的最大距离

1) 如果具有最远距离的两个节点经过根节点，那么最远的距离就是左边最深的深度加上右边最深的深度之和；如果具有最远距离的两个节点之间的路径不经过根节点，那么最远的距离就在根节点的其中一个子树上的两个叶子节点。

```
int _Height(root, distance)
{
    if(root 为空)
        return 0;
    left = _Height(左子树);
    right = _Height(右子树);
    if(left+right>distance)
        distance = left+right;
```

return 左子树、右子树较大加 1;
}

13. 把一颗二叉树原地变成一个双向链表

递归中序遍历；

14. 二叉树的所有路径

```
vector<string> binaryTreePaths(TreeNode* root) {
        // Write your code here
        vector<string> res;
        if(root==NULL)  return res;
        binaryTreePathsCore(root,res,to_string(root->val));
        return res;
    }

    void binaryTreePathsCore(TreeNode* root,vector<string> &str,string strpath){

        if(root->left==NULL&&root->right==NULL){
            //叶子结点
            str.push_back(strpath);
            return;
        }
        if(root->left!=NULL){
            binaryTreePathsCore(root->left,str,strpath+"->"+to_string(root->left->val));
        }
        if(root->right!=NULL){
            binaryTreePathsCore(root->right,str,strpath+"->"+to_string(root->right->val));
        }
    }
```

15. 二叉树中寻找每一层中最大值?

1) 利用队列来分别将每层的树添加进队列进行分析，先记录下第一个值作为最大值并弹出，然后往后边比较边弹出，如果当前值比前面的最大值还要大则替换当前最大值。在弹出每次的节点时将孩子节点加进队列。直到整棵树被遍历完。
2) 伪代码
   if(根节点为空)
   返回
   申请队列 Q，将根节点入队列
   While(队列不空)

```
{
    s=队列长度
    for(i=0;i < s;i++)
    {
        比较队首元素，并将队首元素出栈
        队首左孩子入栈，右孩子入栈
    }
}
```

16. 最大深度、最小深度、会否是平衡树

1) 二叉树的深度等于二叉树的高度，也就等于根节点的高度。根节点的高度为左右子树的高度较大者+1。

```
int deepth(root)
{
    if(root 为空)
        return 0;
    else
    {
        left = deepth(左孩子);
        right = deepth(右孩子);
        return left>right? left+1:right+1;
    }
}
```

2) 求最大深度的时候，只需要比较左右子树的深度，取较大者+1 就行了；但是求最小深度的时候，需要区分双子树与单子树，双子树时，深度较小者+1，单子树时（即左右子树有一颗为空时）为深度较大者+1。

```
int deepth(root)
{
    if(root 为空)
        return 0;
    left = deepth(左孩子);
    right = deepth(右孩子);
    if(左孩子或有孩子为空)
        return 不为空的深度+1;
    return 左右较小者+1;
}
```

3) 判断平衡二叉树，只需要判断平衡因子小于 1 即可，递归判断左右节点是否是平衡的即可；

```
bool isBalance(root)
{
    left = 根节点左孩子高度;
    right = 根节点右孩子高度;
    if(right 与 left 不满足平衡因子)
```

```
            return false;
        return isBalance(左孩子)&&isBalance(右孩子);
    }
```

17. 二叉树中叶子节点的数量

1) 叶子节点就是左右孩子都是空的节点，在进行遍历的过程中，判断是否为叶子节点，如果是叶子节点，将计数器加 1；
2) 伪代码

```
void leafNodeNum(root,k)
{
    if(树为空)
        return;
    if(root 不为空)
    {
        if(叶子节点)
            k++;
        leafNodeNum(root->左孩子,k);
        leafNodeNum(root->右孩子,k);
    }
}
```

18. 交换左右孩子、二叉树镜像

1) 递归交换左右孩子，从跟节点开始交换，节点的左右孩子不都为空，则进行交换操作；否则返回；

```
void exchangeChild(root)
{
    if(root->left 空&&root->right 空)
        return;
    swap(root->left,root->right);
    exchangeChild(root->left);
    exchangeChild(root->right);
}
```

19. 两个二叉树是否相等

1) 判断两颗树的根节点是否相同，如果不相同返回 false，如果相同则递归判断根节点的左右子节点；如果两颗树中有一个树没有遍历完则说明不相等；两棵树都为空则两棵树相等；两棵树一颗为空一颗不为空则不相等；

```
bool treesEqual(root1,root2)
{
```

```
        if(root1 空  && root2 空)
            return true;
        if(root1 空  || root2 空)
            return false;
        if(root1->data == root2->data)
            return
treesEqual(root1->left,root2->left)&&treesEqual(root1->right,root2->right);
        else
            return false;
    }
```

20. 是否为完全二叉树

1) 如果一个结点有右孩子而没有左孩子，那么这棵树一定不是完全二叉树。
如果一个结点有左孩子，而没有右孩子，那么按照层序遍历的结果，这个结点之后的所有结点都是叶子结点这棵树才是完全二叉树。
如果一个结点是叶子结点，那么按照层序遍历的结果，这个结点之后的所有结点都必须是叶子结点这棵树才是完全二叉树。
用一个标记变量 leaf 标记，当一个节点有左孩子无右孩子，leaf=true。之后所有的节点必须为叶子节点。

21. 是否为对称二叉树

```
bool isSymmetrical(root1,root2)
{
    if(root1 空  && root2 空)
        return true;
    if(root1 空  || root2 空)
        return false;
    if(root1->data != root2->data)
        return false;
    else
        return
isSymmetrical(root1->left,root2->right)&&isSymmetrical(root1->right,root2->left);
}
```

22. 判断 B 是否为 A 的子树

1) 找值相同的根结点（遍历解决）
判断两结点是否包含（递归：值、左孩子、右孩子分别相同）

```
bool isPart(root1,root2)
{
```

```
        if(root1 空&&root2 空)
            return true;
        if(root1 空 || root2 空)
            return false;
        if(root1->data != root2->data)
            return false;
        else
            return isPart(root1->left,root2->left)&&isPart(root1->right,root2->right);
    }
    bool isPartTree(root1,root2)
    {
        if (root1 不空 && root2 不空)
        {
            if (root1->data == root2->data)
                result = IsPart(root1, root2);
            if (!result)
                result = IsPartTree(root1->left, root2);
            if (!result)
                result = IsPartTree(root1->right, root2);
        }
        return result;
    }
```

23. 构建哈夫曼树

24. 手写单链表反转？删除指定的单链表的一个节点

1. 单链表反转：尾插法转头查法；设置三个指针，当前节点指针，下一个节点指针，上一个节点指针，一开始上一个节点指针置为空；最后当下一个节点指针为空时说明到达最后节点，则返回该节点指针。
2. 删除指定节点：如果我们把要删除节点的下一个节点的内容复制到需要删除的节点上，然后把删除节点的下一个节点删除，就可以完成删除该节点，同时时间复杂度为 O(1)。如果是尾节点，只能遍历删除，如果只有一个节点，还要删除头节点。

25. 实现一个循环队列

循环中 front 与 rear 的求值。
队首指针进 1：front = (front+1)%MaxSize;
队尾指针进 1：rear = (rear+1)%MaxSize;
队空：rear == front;
队满：(rear+1)%MaxSize == front

26. Top K 问题

1) 如果要找前 K 个最大的数，我们用最小堆，每次用堆顶元素和遍历的数比，如果堆顶元素小，则让堆顶元素的值等于它，然后向下调整
2) 如果要找前 K 个最小的数，我们用最大堆，每次用堆顶元素和遍历的数比，如果堆顶元素大，则让堆顶元素的值等于它，然后向下调整
3) 用快速排序将数组进行排序，然后将数组输出
4) 两者的时间复杂度都是 O（$n\log_2 n$），快排的空间复杂度为 O($\log_2 n$)，堆排序的空间复杂度为 O(1)

27. 求一颗树的最大距离

对于二叉树，若要两个节点 U，V 相距最远，有两种情况：

1，从 U 节点到 V 节点之间的路径经过根节点

2，从 U 节点到 V 节点之间的路径不经过根节点，这种情况下，U，V 节点必定在根节点的左子树或者右子树上，这样就转化为求以根节点的孩子节点为根节点的二叉树中最远的两个节点间的距离

28. KMP

核心是 next（）函数的书写；

29. 数组和链表的区别?

1) 数组必须事先定义固定的长度（元素个数），不能适应数据动态地增减的情况，即数组的大小一旦定义就不能改变。当数据增加时，可能超出原先 定义的元素个数；当数据减少时，造成内存浪费；链表动态地进行存储分配，可以适应数据动态地增减的情况，且可以方便地插入、删除数据项。（数组中插入、删 除数据项时，需要移动其它数据项）。
2) （静态）数组从栈中分配空间（用 NEW 创建的在堆中）， 对于程序员方便快速，但是自由度小；链表从堆中分配空间， 自由度大但是申请管理比较麻烦.
3) 数组在内存中是连续存储的，因此，可以利用下标索引进行随机访问；链表是链式存储结构，在访问元素的时候只能通过线性的方式由前到后顺序访问，所以访问效率比数组要低。

30. 逆序对思路

31. 100 个有序数组合并

归并排序，两个数组合并生成 50 个数组，生成 25 个数组，13 个数组，6 个数组，3 个数组，2 个数组，1 个数组

32. 使用递归和非递归求二叉树的深度

33. 索引、链表的优缺点?

34. 找一个点为中心的圆里包含的所有的点。

35. 字典树的理解

1) 字典树，又称单词查找树，是一种树形结构，是一种哈希树的变种；
2) 根节点不包含字符，除根节点外的每一个子节点都包含一个字符；
   从根节点到叶子节点，路径上经过的字符链接起来，就是该节点对应的字符串；
   每个节点的所有子节点包含的字符都不同；
3) 典型应用是用于统计，排序和保存大量的字符串(不仅限于字符串)，经常被搜索引擎系统用于文本词频统计。

36. 快速排序的优化

1) 当我们每次划分的时候选择的基准数接近于整组数据的最大值或者最小值时，快速排序就会发生最坏的情况，但是每次选择的基准数都接近于最大数或者最小数的概率随着排序元素的增多就会越来越小，我们完全可以忽略这种情况。但是在数组有序的情况下，它也会发生最坏的情况，为了避免这种情况，我们在选择基准数的时候可以采用三数取中法来选择基准数。三数取中法：选择这组数据的第一个元素、中间的元素、最后一个元素，这三个元素里面值居中的元素作为基准数。
2) 当划分的子序列很小的时候(一般认为小于 13 个元素左右时)，我们在使用快速排序对这些小序列排序反而不如直接插入排序高效。因为快速排序对数组进行划分最后就像一颗二叉树一样，当序列小于 13 个元素时我们再使用快排的话就相当于增加了二叉树的最后几层的结点数目，增加了递归的次数。所以我们在当子序列小于 13 个元素的时候就改用直接插入排序来对这些子序列进行排序。

37. 海量数据的 bitmap 使用原理

1) BitMap 解决海量数据寻找重复、判断个别元素是否在海量数据当中等问题;
2) 40 亿个 int 占（40 亿*4）/1024/1024/1024 大概为 14.9G 左右，很明显内存只有 2G，放不下，因此不可能将这 40 亿数据放到内存中计算；40 亿个 int 需要的内存空间为 40 亿/8/1024/1024 大概为 476.83MB;

算法

1. 动态规划，最长公共子序列

2. 分治与递归

3. 贪心算法，背包问题

4. BFS,DFS,地杰斯特拉算法,佛洛依德算法

5. 动态规划的回文字符串？

6. 排序算法?时间复杂度？稳定性算法？

7. 查找算法

8. 字符串匹配？

9. 求一个数开根号（二分）

10. 万个数找到第 20 个大小？

11. 设计抢红包算法

12. 字符串中最长不重复子串

13. 动态规划与分支界限的差异，背包问题和分支界限的差异

14. 出 1-n 的子集。比如 123，有 1，2,3,12，13,123,23.

数据库

1. 事务是什么

1) 事务（txn）是一系列在共享数据库上执行的行为，以达到更高层次更复杂逻辑的功能。事务是 DBMS 中最基础的单位，事务不可分割。
2) ACID，是指在可靠数据库管理系统（DBMS）中，事务(transaction)所应该具有的四个特性：原子性（Atomicity）、一致性（Consistency）、隔离性（Isolation）、持久性（Durability）。
3) 原子性是指事务是一个不可再分割的工作单位，事务中的操作要么都发生，要么都不发生。事务在执行过程中发生错误，会被回滚（Rollback）到事务开始前的状态，就像这个事务从来没有执行过一样。
4) 一致性是指事务使得系统从一个一致的状态转换到另一个一致状态。这是说数据库事务不能破坏关系数据的完整性以及业务逻辑上的一致性。
5) 多个事务并发访问时，事务之间是隔离的，一个事务不应该影响其它事务运行效果。这指的是在并发环境中，当不同的事务同时操纵相同的数据时，每个事务都有各自的完整数据空间。由并发事务所做的修改必须与任何其他并发事务所做的修改隔离。
6) 持久性，意味着在事务完成以后，该事务所对数据库所作的更改便持久的保存在数据库之中，并不会被回滚。即使出现了任何事故比如断电等，事务一旦提交，则持久化保存在数据库中。

2. 分布式事务

1) 本地事务数据库断电的这种情况，它是怎么保证数据一致性的呢？我们使用 SQL Server 来举例，我们知道我们在使用 SQL Server 数据库是由两个文件组成的，一个数据库文件和一个日志文件，通常情况下，日志文件都要比数据库文件大很多。数据库进行任何写入操作的时候都是要先写日志的，同样的道理，我们在执行事务的时候数据库首先会记录下这个事务的 redo 操作日志，然后才开始真正操作数据库，在操作之前首先会把日志文件写入磁盘，那么当突然断电的时候，即使操作没有完成，在重新启动数据库时候，数据库会根据当前数据的情况进行 undo 回滚或者是 redo 前滚，这样就保证了数据的强一致性。
2)

3. 一二三范式

1) 第一范式，数据库表中的字段都是单一属性的，不可再分；每一个属性都是原子项，不可分割；如果实体中的某个属性有多个值时，必须拆分为不同的属性 通俗解释。1NF 是关系模式应具备的最起码的条件，如果数据库设计不能满足第一范式，就不称为关系型数据库。也就是说，只要是关系型数据库，就一定满足第一范式。
2) 第二范式，数据库表中不存在非关键字段对任一候选关键字段的部分函数依赖，即符合

第二范式；如果一个表中某一个字段 A 的值是由另外一个字段或一组字段 B 的值来确定的，就称为 A 函数依赖于 B；当某张表中的非主键信息不是由整个主键函数来决定时，即存在依赖于该表中不是主键的部分或者依赖于主键一部分的部分时，通常会违反 2NF。

3) 第三范式，在第二范式的基础上，数据表中如果不存在非关键字段对任一候选关键字段的传递函数依赖则符合 3NF；第三范式规则查找以消除没有直接依赖于第一范式和第二范式形成的表的主键的属性。我们为没有与表的主键关联的所有信息建立了一张新表。每张新表保存了来自源表的信息和它们所依赖的主键；如果某一属性依赖于其他非主键属性，而其他非主键属性又依赖于主键，那么这个属性就是间接依赖于主键，这被称作传递依赖于主属性。 通俗理解：一张表最多只存 2 层同类型信息 。

4. 数据库的索引类型，数据库索引的作用

1) 数据库索引好比是一本书前面的目录，能加快数据库的查询速度。索引是对数据库表中一个或多个列（例如，employee 表的姓氏 (lname) 列）的值进行排序的结构。如果想按特定职员的姓来查找他或她，则与在表中搜索所有的行相比，索引有助于更快地获取信息。

2) 优点

大大加快数据的检索速度； 创建唯一性索引，保证数据库表中每一行数据的唯一性；加速表和表之间的连接； 在使用分组和排序子句进行数据检索时，可以显著减少查询中分组和排序的时间。

3) 缺点

索引需要占用数据表以外的物理存储空间；创建索引和维护索引要花费一定的时间；当对表进行更新操作时，索引需要被重建，这样降低了数据的维护速度。

4) 类型

唯一索引——UNIQUE，例如：create unique index stusno on student（sno）；表明此索引的每一个索引值只对应唯一的数据记录，对于单列惟一性索引，这保证单列不包含重复的值。对于多列惟一性索引，保证多个值的组合不重复。

主键索引——primary key，数据库表经常有一列或列组合，其值唯一标识表中的每一行。该列称为表的主键。 在数据库关系图中为表定义主键将自动创建主键索引，主键索引是唯一索引的特定类型。该索引要求主键中的每个值都唯一。当在查询中使用主键索引时，它还允许对数据的快速访问。

聚集索引（也叫聚簇索引）——cluster，在聚集索引中，表中行的物理顺序与键值的逻辑（索引）顺序相同。一个表只能包含一个聚集索引，如果某索引不是聚集索引，则表中行的物理顺序与键值的逻辑顺序不匹配。与非聚集索引相比，聚集索引通常提供更快的数据访问速度。

5) 实现方式

B+树、散列索引、位图索引

5. 聚集索引和非聚集索引的区别

1) 聚集索引表示表中存储的数据按照索引的顺序存储，检索效率比非聚集索引高，但对数据更新影响较大。非聚集索引表示数据存储在一个地方，索引存储在另一个地方，索引带有指针指向数据的存储位置，非聚集索引检索效率比聚集索引低，但对数据更新影响

较小。

2) 聚集索引一个表只能有一个，而非聚集索引一个表可以存在多个。聚集索引存储记录是物理上连续存在，而非聚集索引是逻辑上的连续，物理存储并不连续

6. 唯一性索引和主码索引的区别

7. 数据库引擎，innodb 和 myisam 的特点与区别

1) Innodb 引擎提供了对数据库 ACID 事务的支持，并且实现了 SQL 标准的四种隔离级别，关于数据库事务与其隔离级别的内容请见数据库事务与其隔离级别这篇文章。该引擎还提供了行级锁和外键约束，它的设计目标是处理大容量数据库系统，它本身其实就是基于 MySQL 后台的完整数据库系统，MySQL 运行时 Innodb 会在内存中建立缓冲池，用于缓冲数据和索引。但是该引擎不支持 FULLTEXT 类型的索引，而且它没有保存表的行数，当 SELECT COUNT(*) FROM TABLE 时需要扫描全表。当需要使用数据库事务时，该引擎当然是首选。由于锁的粒度更小，写操作不会锁定全表，所以在并发较高时，使用 Innodb 引擎会提升效率。但是使用行级锁也不是绝对的，如果在执行一个 SQL 语句时 MySQL 不能确定要扫描的范围，InnoDB 表同样会锁全表。

2) MyIASM 是 MySQL 默认的引擎，但是它没有提供对数据库事务的支持，也不支持行级锁和外键，因此当 INSERT(插入)或 UPDATE(更新)数据时即写操作需要锁定整个表，效率便会低一些。不过和 Innodb 不同，MyIASM 中存储了表的行数，于是 SELECT COUNT(*) FROM TABLE 时只需要直接读取已经保存好的值而不需要进行全表扫描。如果表的读操作远远多于写操作且不需要数据库事务的支持，那么 MyIASM 也是很好的选择。

3) 大尺寸的数据集趋向于选择 InnoDB 引擎，因为它支持事务处理和故障恢复。数据库的大小决定了故障恢复的时间长短，InnoDB 可以利用事务日志进行数据恢复，这会比较快。主键查询在 InnoDB 引擎下也会相当快，不过需要注意的是如果主键太长也会导致性能问题，关于这个问题我会在下文中讲到。大批的 INSERT 语句(在每个 INSERT 语句中写入多行，批量插入)在 MyISAM 下会快一些，但是 UPDATE 语句在 InnoDB 下则会更快一些，尤其是在并发量大的时候。

8. 关系型和非关系型数据库的区别

数据库类型	特性	优点	缺点
关系型数据库 SQLite、Oracle、mysql	1、关系型数据库，是指采用了关系模型来组织数据的数据库； 2、关系型数据库的最大特点就是事务的一致性； 3、简单来说，关系模型指的就是二维表格模型，而一个关系型数据库就是由二维表及其之间的联系所组成的一个数据组织。	1、容易理解：二维表结构是非常贴近逻辑世界一个概念，关系模型相对网状、层次等其他模型来说更容易理解； 2、使用方便：通用的SQL语言使得操作关系型数据库非常方便； 3、易于维护：丰富的完整性(实体完整性、参照完整性和用户定义的完整性)大大减低了数据冗余和数据不一致的概率； 4、支持SQL，可用于复杂的查询。	1、为了维护一致性所付出的巨大代价就是其读写性能比较差； 2、固定的表结构； 3、高并发读写需求； 4、海量数据的高效率读写；
非关系型数据库 MongoDb、redis、HBase	1、使用键值对存储数据； 2、分布式； 3、一般不支持ACID特性； 4、非关系型数据库严格上不是一种数据库，应该是一种数据结构化存储方法的集合。	1、无需经过sql层的解析，读写性能很高； 2、基于键值对，数据没有耦合性，容易扩展； 3、存储数据的格式：nosql的存储格式是key,value形式、文档形式、图片形式等等，文档形式、图片形式等等，而关系型数据库则只支持基础类型。	1、不提供sql支持，学习和使用成本较高； 2、无事务处理，附加功能bi和报表等支持也不好；

9. 数据库的隔离级别

1) 隔离级别高的数据库的可靠性高，但并发量低，而隔离级别低的数据库可靠性低，但并发量高，系统开销小。
2) READ UNCIMMITTED（未提交读），事务中的修改，即使没有提交，其他事务也可以看得到，比如说上面的两步这种现象就叫做脏读，这种隔离级别会引起很多问题，如无必要，不要随便使用；这就是事务还没提交，而别的事务可以看到他其中修改的数据的后果，也就是脏读；
3) READ COMMITTED（提交读），大多数数据库系统的默认隔离级别是 READ COMMITTED，这种隔离级别就是一个事务的开始，只能看到已经完成的事务的结果，正在执行的，是无法被其他事务看到的。这种级别会出现读取旧数据的现象
4) REPEATABLE READ（可重复读），REPEATABLE READ 解决了脏读的问题，该级别保证了每行的记录的结果是一致的，也就是上面说的读了旧数据的问题，但是却无法解决另一个问题，幻行，顾名思义就是突然蹦出来的行数据。指的就是某个事务在读取某个范围的数据，但是另一个事务又向这个范围的数据去插入数据，导致多次读取的时候，数据的行数不一致。虽然读取同一条数据可以保证一致性，但是却不能保证没有插入新的数据。
5) SERIALIZABLE（可串行化），SERIALIZABLE 是最高的隔离级别，它通过强制事务串行执行（注意是串行），避免了前面的幻读情况，由于他大量加上锁，导致大量的请求超时，因此性能会比较底下，再特别需要数据一致性且并发量不需要那么大的时候才可能考虑这个隔离级别。

10. 数据库连接池的作用

1) 在内部对象池中，维护一定数量的数据库连接，并对外暴露数据库连接的获取和返回方法，如外部使用者可通过 getConnection 方法获取数据库连接，使用完毕后再通过 releaseConnection 方法将连接返回，注意此时的连接并没有关闭，而是由连接池管理器回收，并为下一次使用做好准备。

2) 资源重用，由于数据库连接得到重用，避免了频繁创建、释放连接引起的大量性能开销。在减少系统消耗的基础上，增进了系统环境的平稳性（减少内存碎片以级数据库临时进程、线程的数量）
3) **更快的系统响应速度**，数据库连接池在初始化过程中，往往已经创建了若干数据库连接置于池内备用。此时连接池的初始化操作均已完成。对于业务请求处理而言，直接利用现有可用连接，避免了数据库连接初始化和释放过程的时间开销，从而缩减了系统整体响应时间。
4) **新的资源分配手段**，对于多应用共享同一数据库的系统而言，可在应用层通过数据库连接的配置，实现数据库连接技术。
5) **统一的连接管理，避免数据库连接泄露**，较较为完备的数据库连接池实现中，可根据预先的连接占用超时设定，强制收回被占用的连接，从而避免了常规数据库连接操作中可能出现的资源泄露。

11. 数据的锁的种类，加锁的方式

1) **锁**是网络数据库中的一个非常重要的概念，当多个用户同时对数据库并发操作时，会带来数据不一致的问题，所以，锁主要用于多用户环境下保证数据库完整性和一致性。
2) 数据库锁出现的目的：处理并发问题；
3) **并发控制的主要采用的技术手段**：乐观锁、悲观锁和时间戳。
4) 从数据库系统角度分为三种：排他锁、共享锁、更新锁。从程序员角度分为两种：一种是悲观锁，一种乐观锁。

12. 数据库 union join 的区别

1) join 是两张表做交连后里面条件相同的部分记录产生一个记录集，union 是产生的两个记录集(字段要一样的)并在一起，成为一个新的记录集 。
2) union 在数据库运算中会过滤掉重复数据，并且合并之后的是根据行合并的，即：如果 a 表和 b 表中的数据各有五行，且有两行是重复数据，合并之后为 8 行。运用场景：适合于需要进行统计的运算
3) union all 是进行全部合并运算的，即：如果 a 表和 b 表中的数据各有五行，且有两行是重复数据，合并之后为 10 行。
4) join 是进行表关联运算的，两个表要有一定的关系。即：如果 a 表和 b 表中的数据各有五行，且有两行是重复数据，根据某一列值进行笛卡尔运算和条件过滤，假如 a 表有 2 列，b 表有 2 列，join 之后是 4 列。

13. Inner join,left outter join,right outter join 之间的区别

设计模式

1. 单例模式

1) 在它的核心结构中包含一个被称为单例的特殊类，一个类只有一个实例，即一个类只有一个对象实例；
2) 所有的单例模式都是使用静态方法进行创建的，所以单例对象在内存中静态共享区中存储；
3) 单例模式分为饿汉式和懒汉式，懒汉式单例模式在类加载时不初始化，饿汉式单例模式，在类加载时就完成初始化，所以类加载较慢，但获取对象速度快。

2. 手写线程安全的单例模式?

3. 工厂模式

1) 工厂模式，简单工厂模式是由一个工厂对象根据收到的消息决定要创建哪一个类的对象实例。需要 switch 或 if 进行类型选择；工厂类创建的对象比较少，客户只需要传入工厂类参数，对于如何创建对象不关心；
2) 工厂方法模式，定义一个创建对象的工厂接口，让子类决定实例化哪一个类，将实际创建工作推迟到子类当中。创建对象的接口，让子类决定具体实例化的对象，把简单的内部逻辑判断移动到客户端。
3) 抽象工厂模式，抽象工厂是围绕一个超级工厂创建其他工厂，该超级工厂又称为其他工厂的工厂。提供一个创建一系列相关或相互依赖对象的接口，而无需指定他们具体的类。当一个产品族中的多个对象被设计成一起工作时，它能保证客户端始终只使用同一个产品族中的对象。

4. 装饰器模式

1) 动态地给一个对象增加一些额外的职责。在为对象增加额外职责方面，装饰模式替代了继承，它比子类继承父类更为灵活，它用无需定义子类的方式来给对象动态的增加职责，使用对象之间的关联关系来取代继承，同时避免类型体系的快速膨胀。
2) 装饰器模式，顾名思义，就是对已经存在的某些类进行装饰，以此来扩展一些功能。
3) 装饰器的价值在于装饰，他并不影响被装饰类本身的核心功能。在一个继承的体系中，子类通常是互斥的。

5. 订阅/发布模式

1) 订阅发布模式定义了一种一对多的依赖关系，让多个订阅者对象同时监听某一个主题对象。这个主题对象在自身状态变化时，会通知所有的订阅者，使它们能够自动更新自己的状态；
2) 发布者向某个信道发布一条消息，订阅者绑定这个信道，当有消息发布至信道时就会接受到一个通知。

6. 观察者模式

1) 在对象之间定义一个一对多的依赖关系，这样一来，当一个对象改变状态，依赖他的对象会收到通知自动更新；
2) 抽象被观察对象，抽象观察对象，具体被观察者对象，具体观察者对象；
3) 微信公众号是一个典型的例子，有一个微信公众号服务，不定时发布一些消息，关注公众号就可以收到推送消息，取消关注就收不到推送消息；
4) 一个目标对象管理所有相依于它的观察者对象，并且在它本身的状态改变时主动发出通知。这通常透过呼叫各观察者所提供的方法来实现。此种模式通常被用来实时事件处理系统。

7. MVC 模式

1) MVC 的全名是 Model View Controller，是模型(model)－视图(view)－控制器(controller)的缩写，是一种软件设计典范。它是用一种业务逻辑、数据与界面显示分离的方法来组织代码，将众多的业务逻辑聚集到一个部件里面，在需要改进和个性化定制界面及用户交互的同时，不需要重新编写业务逻辑，达到减少编码的时间。
2) 使用的 MVC 的目的：在于将 M 和 V 的实现代码分离，从而使同一个程序可以使用不同的表现形式。
3) 用户首先在界面中进行人机交互，然后请求发送到控制器，控制器根据请求类型和请求的指令发送到相应的模型，模型可以与数据库进行交互，进行增删改查操作，完成之后，根据业务的逻辑选择相应的视图进行显示，此时用户获得此次交互的反馈信息，用户可以进行下一步交互，如此循环。

多线程编程

1.

HR 问题

1. 请描述一件小时候您印象最深刻的事情
2. **最大的挫折是什么，学到什么？**
   求职者大可不必为自己的缺点遮遮掩掩，因为 HR 问这个问题的真实目的并不是想知道你有什么缺点，而是想借此问题考察你对自己的缺点有没有改正的态度。就好像 HR 问：你最大的失败经历是什么？他并不是想知道你是怎么失败的，而是想看你怎么对待失败的这件事。高考的失败，一方面原因是当时太看重高考，导致紧张，另一方面，因为过度的自信，对自己做过的题太自信，导致检查的时候检查不出错误，走出考场后发现自己在什么地方错了；后面就是吸取教训，第一，不要把得失看得太重要，第二，保持自信，但是不能过分自信，多想想，多问问。
3. **请评价自身个性的长处及不足**

长处：自信、乐观、责任、吃苦耐劳；　信心、恒心、耐心、细心
比较善于交际，人缘好；比较有条理，我的工位上的书，笔，杯子等物品都是摆放整洁的；
缺点：我对我认为不对的人或事，容易提出不同意见，导致经常得罪人；我办事比较急，准确性有时不够；说缺点同时表明这个缺点正在改进中，最好的方式就是说缺点的同时能带出一个优点；

4. **自己 1-5 的职业规划**

自我的认知定位；对这个职位的认知；对这家公司的认知；求职态度；
这个行业是 XX，职位是 XX，我觉得我个人有哪几点匹配，而且过去 XX 原因让我有足够多的兴趣，所以未来我主要想在这个领域里深耕细作，认真成为行业专家——这个就是长远的规划和自我认知。

我在大学里做过几次 iOS 的相关项目，因此产生了浓厚的兴趣，我学过 XX 课程，XX 教材，参与参加过 XX 项目，个人经历非常适合这个职位，因此，长远的规划，我没想太具体，主要是想深耕这个领域，争取早日成为一位 iOS 的编程专家。

至于短期的规划，我因为刚毕业，项目经验还不是很充足，但是如果我能获得这次宝贵的机会，那么我会努力学习，多请教前辈同事，提高自己的项目，在公司的 iOS 项目上做出自己的贡献，比如现在公司做的几个 App，分别是 xx，xx，xx，我相信我的加盟，会让开发进度大大加快，未来如果有别的 iOS 项目，也能保质保量的完成。

首先做好本职工作，脚踏实地，满满积攒技术经验，走技术路线。不断学习，与时俱进，有机会的话可以做其他的技术路线。长远看来，希望通过自己的不断奋斗能够给公司带来一定的价值，同时实现自己的价值。

5. **请分别描述到目前为止让自己最伤感、最快乐和最感动的事**
6. **总结你的大学四年**
7. **除了课设之外参加哪些其他活动**

从事学生工作，本科期间一直担任班长，研究生期间担任实验室党支部书记一职；

8. 队合作时与别人意见不合怎么办，说一个发生在自己身上的真事；
9. **压力特别大的时候怎么释放压力，有坚持健身吗；**

看书，打羽毛球；做家务；

10. 跟你的导师有冲突的时候，怎么解决的？
11. **对加班的看法？**

如果是工作需要我会义不容辞加班，我现在单身，没有任何家庭负担，可以全身心的投入工作。但同时，我也会提高工作效率，减少不必要的加班。

12. 如果你已经在　CVTE　工作，是什么原因会让你离职
13. **你最佩服的人有哪些特征？**

我的导师，1、真正在搞学术；2、精神满满；3、不同流合污

14. **你最佩服的同学有哪些特点？**

同门，1、他的见识；2、他的个人能力；3、努力；

15. **最近在看什么技术书？学到什么？**

机器学习

16. 有意思的代码是什么？

17. 项目如何体现你的能力与思维方式？
18. 为什么成绩不好？
19. **你会推荐什么网站？**

    雷锋网，只是一个科技网站。该网站上会介绍很多新奇的技术的应用。
20. **你会推荐什么书，为什么？**

    诺贝尔文学奖获得者马尔克斯的小说《霍乱时期的爱情》，这本书讲述了一段跨越半个世纪的爱情史诗。穷尽了所有爱情的可能性，被誉为人类有史以来最伟大的爱情小说。
21. **公司为什么要聘用你？**

    我认为公司需要一个 xx 的人，我具备 xx 的能力，符合公司的岗位需求，我非常期待能够加入公司，通过自己的努力与公司一起成长。希望能够在公司做到技术专家。
22. **如何胜任你的工作？**

    首先承认我没有工作经验，我会毫无保留的去学习，争取在最快的速度由学生转换成一个工作者。我有扎实的理论基础，结合自己吃苦耐劳的精神，一直保持奋斗者的状态，我相信我会做到胜任工作，而且做得非常优秀。
23. **你还有什么想问的？**

    关于这个岗位，公司有培训吗；这个岗位的晋升空间是怎样的；这个职位所在部门在公司的地位；这个岗位具体负责的工作是哪些？