1. static 修饰对象,如果类生命了构造函数,则会调用类的构造函数来初始化对象,如果没有的话,不会合成默认构造函数,但是依然会初始化,类的内置类型成员与普通内置类型初始化相同;

2. extern 关键字的作用

```
    extern声明变量在外部的定义?
        1、 全局变量如果不在文件开始定义,则其作用域有效区域为定义部分到文件结尾,之前的部分如果需要使用全局变量,需要使用extern 声明
        2、 如果多个文件公用一个全局变量,在非定义文件中需要使用extern声明
    extern 修饰函数?
        与修饰变量作用类似,如果在一个文件中定义了函数func1，在另一个文件中可以使用extern声明这个函数，然后直接调用;不需要包含头文件；一般使用在较小的文件中，较大的文件通过头文件来使用;
    extern C的作用
        为了能够正确实现C++代码调用其他C语言代码。加上extern "C"后,会指示编译器这部分代码按C语言的方式进行编译。由于C++支持函数重载,因此编译器编译函数的过程中会将函数的参数类型加到编译后的代码中,而不仅仅是函数名;而C语言不支持函数重载,因此编译C语言代码不会带上函数的参数类型,一般只包含函数名;
```

3. static 关键字的作用

```
1. 什么是static?
static 是 C/C++ 中很常用的修饰符，它被用来控制变量的存储方式和可见性。

1.1 static 的引入
我们知道在函数内部定义的变量，当程序执行到它的定义处时，编译器为它在栈上分配空间，函数在栈上分配的空间在此函数执行结束时会释放掉，这样就产生了一个问题：如果想将函数中此变量的值保存至下一次调用时，如何实现？ 最容易想到的方法是定义为全局的变量，但定义一个全局变量有许多缺点，最明显的缺点是破坏了此变量的访问范围（使得在此函数中定义的变量，不仅仅只受此函数控制）。static 关键字则可以很好的解决这个问题。

另外，在 C++ 中，需要一个数据对象为整个类而非某个对象服务,同时又力求不破坏类的封装性,即要求此成员隐藏在类的内部，对外不可见时，可将其定义为静态数据。

1.2 静态数据的存储
全局（静态）存储区：分为 DATA 段和 BSS 段。DATA 段（全局初始化区）存放初始化的全局变量和静态变量；BSS 段（全局未初始化区）存放未初始化的全局变量和静态变量。程序运行结束时自动释放。其中BBS段在程序执行之前会被系统自动清0，所以未初始化的全局变量和静态变量在程序执行之前已经为0。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。

在 C++ 中 static 的内部实现机制：静态数据成员要在程序一开始运行时就必须存在。因为函数在程序运行中被调用，所以静态数据成员不能在任何函数内分配空间和初始化。

这样，它的空间分配有三个可能的地方，一是作为类的外部接口的头文件，那里有类声明；二是类定义的内部实现，那里有类的成员函数定义；三是应用程序的 main() 函数前的全局数据声明和定义处。

静态数据成员要实际地分配空间，故不能在类的声明中定义（只能声明数据成员）。类声明只声明一个类的"尺寸和规格"，并不进行实际的内存分配，所以在类声明中写成定义是错误的。它也不能在头文件中类声明的外部定义，因为那会造成在多个使用该类的源文件中，对其重复定义。
```

static 被引入以告知编译器，将变量存储在程序的静态存储区而非栈上空间，静态数据成员按定义出现的先后顺序依次初始化，注意静态成员嵌套时，要保证所嵌套的成员已经初始化了。消除时的顺序是初始化的反顺序。

优势：可以节省内存，因为它是所有对象所公有的，因此，对多个对象来说，静态数据成员只存储一处，供所有对象共用。静态数据成员的值对每个对象都是一样，但它的值是可以更新的。只要对静态数据成员的值更新一次，保证所有对象存取更新后的相同的值，这样可以提高时间效率。

2. 在 C/C++ 中static的作用

```
    static 修饰全局变量
        限定变量使用范围只能在本文件中使用,即使使用extern外部声明也不可以在其他文件内使用;
    static 修饰普通函数
        限定使用范围,这个函数只能在本文件内使用,不能被其他文件调用.
    static 修饰成员变量
        static成员变量属于类,不属于某个具体的对象,即使创建多个对象,所有对象使用的都是这份内存中的数据,当某个对象修改这个变量,其他对象也会影响其他对象
        static成员变量必须在类声明的外部初始化,具体形式为:  type class::name = value
        静态成员变量初始化时不能再加static 但必须要有数据类型
        static成员变量的内存既不是在声明类时分配,也不是在创建对象时分配,而是在初始化时分配;
        static既可以通过类直接访问 Class::static 也可以通过Object.static访问,也可以Point->static访问
        这三种方式等效
        static成员变量不占用对象的内存,而是在所有对象之外开辟内存,即使不创建对象也可以访问

    static 修饰成员函数
        1. static成员函数不包含this指针，所以static成员函数不能访问非static类成员,只能访问static修饰的类成员
        2.static成员函数不能定义为const的，static函数不能访问非静态成员变量(备注:编译错误 static member function  cannot have cv-qualifier)
引用:
    2.1 总的来说
     (1) 在修饰变量的时候，static 修饰的静态局部变量只执行初始化一次，而且延长了局部变量的生命周期，直到程序运行结束以后才释放。
     (2) static 修饰全局变量的时候，这个全局变量只能在本文件中访问，不能在其它文件中访问，即便是 extern 外部声明也不可以。
     (3) static 修饰一个函数，则这个函数的只能在本文件中调用，不能被其他文件调用。static 修饰的变量存放在全局数据区的静态变量区，包括全局静态变量和局部静态变量，都在全局数据区分配内存。初始化的时候自动初始化为 0。
     (4) 不想被释放的时候，可以使用static修饰。比如修饰函数中存放在栈空间的数组。如果不想让这个数组在函数调用结束释放可以使用 static 修饰。
     (5) 考虑到数据安全性（当程序想要使用全局变量的时候应该先考虑使用 static）。
    2.2 静态变量与普通变量
    静态全局变量有以下特点:

     (1) 静态变量都在全局数据区分配内存，包括后面将要提到的静态局部变量;
     (2) 未经初始化的静态全局变量会被程序自动初始化为0（在函数体内声明的自动变量的值是随机的，除非它被显式初始化，而在函数体外被声明的自动变量也会被初始化为 0）;
     (3) 静态全局变量在声明它的整个文件都是可见的，而在文件之外是不可见的。
```

优点：静态全局变量不能被其它文件所用；其它文件中可以定义相同名字的变量，不会发生冲突。

（1）全局变量和全局静态变量的区别
1）全局变量是不显式用 static 修饰的全局变量，全局变量默认是有外部链接性的，作用域是整个工程，在一个文件内定义的全局变量，在另一个文件中，通过 extern 全局变量名的声明，就可以使用全局变量。
2）全局静态变量是显式用 static 修饰的全局变量，作用域是声明此变量所在的文件，其他的文件即使用 extern 声明也不能使用。
2.3 静态局部变量有以下特点：
（1）该变量在全局数据区分配内存；
（2）静态局部变量在程序执行到该对象的声明处时被首次初始化，即以后的函数调用不再进行初始化；
（3）静态局部变量一般在声明处初始化，如果没有显式初始化，会被程序自动初始化为 0；
（4）它始终驻留在全局数据区，直到程序运行结束。但其作用域为局部作用域，当定义它的函数或语句块结束时，其作用域随之结束。
一般程序把新产生的动态数据存放在堆区，函数内部的自动变量存放在栈区。自动变量一般会随着函数的退出而释放空间，静态数据（即使是函数内部的静态局部变量）也存放在全局数据区。全局数据区的数据并不会因为函数的退出而释放空间。

static 修饰局部变量
存储区：由栈变为静态存储区,生存期为整个源程序,只能在定义该变量的函数内使用.退出该函数后,尽管变量还继续存在,但不能使用他;
作用域：作用域仍为局部作用域,当定义它的函数或者语句块结束的时候,作用域随之结束.

3. volatile的作用是什么

1．访问寄存器要比访问内存快,因此CPU会优先访问该数据在寄存器中的存储效果,但是内存中的数据可能已经发生改变,而寄存器中还保留着原来的结果.为了避免这种情况的发生,将该变量声明为volatile,告诉cpu每次都从内存去读取数据.
2．一个参数可以即是const又是volatile的吗？ 可以

4. const的作用:

1．const 修饰全局变量
2．const 修饰局部变量
3．const 修饰指针const int *;
5．const 修饰指针指向的对象 int * const
6．const 修饰成员变量,必须在构造函数列表中初始化
7．const 修饰成员函数,说明该函数不应该修改非静态成员,但是这并不十分可靠,指针所指的非静态成员可能会被改变

5. new与malloc的区别: 验证下 operator new delete 与 delete new的效果

1．new分配内存按照数据类型进行分配,malloc分配内存按照大下分配;
2．new不仅分配一段内存,而且会调用构造函数,但是malloc不会;

```
解析:
new实现的原理:
    调用operator new 函数申请空间
    在申请的空间上执行构造函数,完成对象的构造
delete的原理:
    在空间上执行析构函数,完成对象中资源的清理工作在空间上执行析构函数,完成对象中资源的清理工作
new T[N]的原理:
    调用operator new[]函数,在operator new[]中实际调用operator new函数完成N个对象的申请
    在申请的空间上执行N次构造函数
delete[]的原理;
    在是否的对象空间上执行N次析构函数,完成N个对象中的资源的清理
    调用operator delete[]释放空间,实际在operator delete[]中调用operator delete来释放空间

    3. new返回的是指定对象的指针,而malloc返回的是void*
    4. new是一个操作符可以重载,malloc是一个库函数
    5. new分配的内存要用delete销毁,malloc要用free来销毁;delete销毁的时候会调用对象的析构函数,而free不会
    6. malloc分配的内不够的时候,可以用realloc扩容(扩容的原理? ) new没有这样的操作
    7. new如果分配失败了会抛出bad_malloc的异常,而malloc失败了会返回NULL。因此对于new, 正确的字试是采用try...catch语法,而malloc则应该判断指针的返回值,为了兼容很多c语言程序员的习惯,c++也可以采用new nothrow的方法禁止抛出异常返回NULL;
    8. new和new[] 的区别,new[]一次分配所有内存,多次调用构造函数;分别搭配使用delete和delete[].
    9. 谈谈new和malloc的实现,空闲链表,分配方法(首次适配原则,最佳视频原则,最差适配原则,快速适配原则)。delete和free实现原理; free为什么知道销毁多大的空间;
```

6. C++多态性和虚函数表 ``` 多态分为静态多态和动态多态。静态多态是通过重载和模板技术实现,在编译的时候确定。动态多态通过虚函数和继承关系实现,执行动态绑定,在运行的时候确定。 动态多态实现有几个条件: 1. 虚函数 2. 一个基类的指针或引用指向派生类的对象

```
    基类指针在调用成员函数(虚函数)时,就会去查找该对象的虚函数表。虚函数表的地址在每个对象的首地址。
    查找该虚函数表中该函数的指针进行调用。每个对象中保存的只是一个虚函数表的指针,C++内部为每一个类维持一个虚函数表,
    该类的对象的都指向这同一个虚函数表。虚函数表中为什么就能准确查找相应的函数指针呢? 因为在类设计的时候,
    虚函数表直接从基类也继承过来, 如果覆盖了其中的某个虚函数, 那么虚函数表的指针就会被替换, 因此可以根据指针准确找到该调用哪个函数。
```

7. 虚函数的作用?

```
虚函数用于实现多态, 这点大家都能答上来
但是虚函数在设计上还具有封装和抽象的作用。比如抽象工厂模式。(实现下)
动态绑定是如何实现的?
第一个问题中基本回答了, 主要都是结合虚函数表来答就行。
```

静态多态和动态多态。静态多态是指通过模板技术或者函数重载技术实现的多态，其在编译器确定行为。动态多态是指通过虚函数技术实现在运行期动态绑定的技术。

虚函数表

虚函数表是针对类的还是针对对象的？同一个类的两个对象的虚函数表是怎么维护的？

编译器为每一个类维护一个虚函数表，每个对象的首地址保存着该虚函数表的指针，同一个类的不同对象实际上指向同一张虚函数表。

纯虚函数如何定义，为什么对于存在虚函数的类中析构函数要定义成虚函数

为了实现多态进行动态绑定，将派生类对象指针绑定到基类指针上，对象销毁时，如果析构函数没有定义为析构函数，

则会调用基类的析构函数，显然只能销毁部分数据。

如果要调用对象的析构函数，就需要将该对象的析构函数定义为虚函数，销毁时通过虚函数表找到对应的析构函数。

纯虚函数定义

```
virtual ~myClass() = 0;
```

8. 析构函数能抛出异常吗

答案肯定是不能。

C++标准指明析构函数不能、也不应该抛出异常。

C++异常处理模型最大的特点和优势就是对C++中的面向对象提供了最强大的无缝支持。

那么如果对象在运行期间出现了异常，C++异常处理模型有责任清除那些由于出现异常所导致的已经失效了的对象(也即对象超出了它原来的作用域)，

并释放对象原来所分配的资源， 这就是调用这些对象的析构函数来完成释放资源的任务，

所以从这个意义上说，析构函数已经变成了异常处理的一部分。

(1) 如果析构函数抛出异常，则异常点之后的程序不会执行，如果析构函数在异常点之后执行了某些必要的动作比如释放某些资源，

则这些动作不会执行，会造成诸如资源泄漏的问题。

(2) 通常异常发生时，c++的机制会调用已经构造对象的析构函数来释放资源，此时若析构函数本身也抛出异常，

则前一个异常尚未处理，又有新的异常，会造成程序崩溃的问题。

9. 构造函数和析构函数中调用虚函数吗?

构造函数不能(待验证)

10. 指针和引用的区别

指针保存的是所指对象的地址，引用是所指对象的别名，指针需要通过解引用间接访问，而引用是直接访问;

指针可以改变地址，从而改变所指的对象，而引用必须从一而终;

引用在定义的时候必须初始化，而指针则不需要;

指针有指向常量的指针和指针常量，而引用没有常量引用;

指针更灵活，用的好威力无比，用的不好处处是坑，而引用用起来则安全多了，但是比较死板。

11. 指针与数组千丝万缕的联系

```
一个一维int数组的数组名实际上是一个int* const 类型;
一个二维int数组的数组名实际上是一个int (*const p)[n];
数组名做参数会退化为指针，除了sizeof
```

12. 智能指针是怎么实现的？什么时候改变引用计数?

```
构造函数中计数初始化为1;
拷贝构造函数中计数值加1;
赋值运算符中，左边的对象引用计数减一，右边的对象引用计数加一;
析构函数中引用计数减一;
在赋值运算符和析构函数中，如果减一后为0，则调用delete释放对象。
share_prt<T>与weak_ptr<T>的区别?
//share_ptr可能出现循环引用，从而导致内存泄露
class A
{
    public:
    share_ptr<B> p;
};

class B
{
    public:
    share_ptr<A> p;
}
int main()
{
    while(true)
    {
        share_prt<A> pa(new A()); //pa的引用计数初始化为1
        share_prt<B> pb(new B()); //pb的引用计数初始化为1
        pa->p = pb; //pb的引用计数变为2
        pb->p = pa; //pa的引用计数变为2
    }
    //假设pa先离开，引用计数减一变为1，不为0因此不会调用class A的析构函数，因此其成员p也不会被析构，pb的引用计数仍然为2;
    //同理pb离开的时候，引用计数也不能减到0
    return 0;
}
/*
** weak_ptr是一种弱引用指针，其存在不会影响引用计数，从而解决循环引用的问题
*/
```

13. C++四种类型转换：static_cast, dynamic_cast, const_cast, reinterpret_cast

```
const_cast用于将const变量转为非const
static_cast用的最多，对于各种隐式转换，非const转const，void*转指针等，static_cast能用于多态想上转化，如果向下转能成功但是不安全，结果未知;
dynamic_cast用于动态类型转换。只能用于含有虚函数的类，用于类层次间的向上和向下转化。
```

只能转指针或引用。向下转化时，如果是非法的对于指针返回NULL，对于引用抛异常。要深入了解内部转换的原理。
reinterpret_cast几乎什么都可以转，比如将int转指针，可能会出问题，尽量少用；
为什么不使用C的强制转换？C的强制转换表面上看起来功能强大什么都能转，但是转化不够明确，不能进行错误检查，容易出错。

14. 内存对齐的原则

从0位置开始存储；
变量存储的起始位置是该变量大小的整数倍；
结构体总的大小是其最大元素的整数倍，不足的后面要补齐；
结构体中包含结构体，从结构体中最大元素的整数倍开始存；
如果加入pragma pack(n) ，取n和变量自身大小较小的一个。

15. 内联函数有什么优点？内联函数与宏定义的区别？

宏定义在预编译的时候就会进行宏替换；
内联函数在编译阶段，在调用内联函数的地方进行替换，减少了函数的调用过程，但是使得编译文件变大。因此，内联函数适合简单函数，对于复杂函数，即使定义了内联编译器可能也不会按照内联的方式进行编译。
内联函数相比宏定义更安全，内联函数可以检查参数，而宏定义只是简单的文本替换。因此推荐使用内联函数，而不是宏定义。
使用宏定义函数要特别注意给所有单元都加上括号，#define MUL(a, b) a b，这很危险，正确写法：#define MUL(a, b) ((a) (b))

16. C++内存管理

C++内存分为那几块？
（堆区，栈区，常量区，静态和全局区）
每块存储哪些变量？
学会迁移，可以说到malloc，从malloc说到操作系统的内存管理，
说道内核态和用户态，然后就什么高端内存，slab层，伙伴算法，VMA可以巴拉巴拉了，接着可以迁移到fork()。

17. STL里的内存池实现

STL内存分配分为一级分配器和二级分配器，一级分配器就是采用malloc分配内存，二级分配器采用内存池。
二级分配器设计的非常巧妙，分别给8k，16k,...，128k等比较小的内存片都维持一个空闲链表，每个链表的头节点由一个数组来维护。
需要分配内存时从合适大小的链表中取一块下来。假设需要分配一块10K的内存，那么就找到最小的大于等于10k的块，也就是16K，
从16K的空闲链表里取出一个用于分配。释放该块内存时，将内存节点归还给链表。
如果要分配的内存大于128K则直接调用一级分配器。为了节省维持链表的开销，采用了一个union结构体，

分配器使用union里的next指针来指向下一个节点，而用户则使用union的空指针来表示该节点的地址。

18. STL里set和map是基于什么实现的。红黑树的特点?

set和map都是基于红黑树实现的。
红黑树是一种平衡二叉查找树，与AVL树的区别是什么？AVL树是完全平衡的，红黑树基本上是平衡的。
为什么选用红黑数呢？因为红黑数是平衡二叉树，其插入和删除的效率都是N(logN)，与AVL相比红黑数插入和删除最多只需要3次旋转，
而AVL树为了维持其完全平衡性，在坏的情况下要旋转的次数太多。
红黑树的定义：
(1) 节点是红色或者黑色；
(2) 父节点是红色的话，子节点就不能为红色；
(3) 从根节点到每个页子节点路径上黑色节点的数量相同；
(4) 根是黑色的，NULL节点被认为是黑色的。

STL里的其他数据结构和算法实现也要清楚
这个问题，把STL源码剖析好好看看，不仅面试不慌，自己对STL的使用也会上升一个层次。

19. 必须在构造函数初始化式里进行初始化的数据成员有哪些

(1) 常量成员，因为常量只能初始化不能赋值，所以必须放在初始化列表里面
(2) 引用类型，引用必须在定义的时候初始化，并且不能重新赋值，所以也要写在初始化列表里面
(3) 没有默认构造函数的类类型，因为使用初始化列表可以不必调用默认构造函数来初始化，而是直接调用拷贝构造函数初始化

20. 模板特化

(1) 模板特化分为全特化和偏特化，模板特化的目的就是对于某一种变量类型具有不同的实现，因此需要特化版本。
例如，在STL里迭代器为了适应原生指针就将原生指针进行特化。

21. 定位内存泄露

(1)在windows平台下通过CRT中的库函数进行检测;
(2)在可能泄漏的调用前后生成块的快照，比较前后的状态，定位泄漏的位置
(3)Linux下通过工具valgrind检测

22. 手写strcpy

```
    char strcpy(char dst, const char src)
    {
    assert(dst);
    assert(src);
    char ret = dst;
    while((dst++ = src++) != '\0');
    return ret;
    }
    //该函数是没有考虑重叠的
    char strcpy(char dst, const char src)
    {
    assert((dst != NULL) && (src != NULL));
    char ret = dst;
    int size = strlen(src) + 1;
    if(dst > src || dst < src + len)
    {
    dst = dst + size - 1;
    src = src + size - 1;
    while(size--)
    {
    dst-- = src--;
    }
    }
    else
    {
    while(size--)
    {
    dst++ = src++;
    }
    }
    return ret;
    }
```

22. 手写memcpy函数

```
void memcpy(void dst, const void src, size_t size)
{
if(dst == NULL || src == NULL)
{
return NULL;
}
void res = dst;
char pdst = (char)dst;
char psrc = (char)src;
if(pdst > psrc && pdst < psrc + size) //重叠

{

pdst = pdst + size - 1;

psrc = pdst + size - 1;
```

```
while(size--)
{
*pdst-- = *psrc--;
}
}
else //无重叠
{
while(size--)
{
*dst++ = *src++;
}
}
return ret;
}
```

23. 手写strcat函数

```
char strcat(char dst, const char src)
{
char ret = dst;

while(*dst != '\0')
++dst;

while((*dst++ = *src) != '\0');
return ret;
}
```

24. 手写strcmp函数

```
int strcmp(const char str1, const char str2)
{



while(*str1 == *str2 && *str1 != '\0')

{

++str1;

++str2;

}

return *str1 - *str2;

}
```

25. Hash表

- Hash表实现（拉链和分散地址）

- Hash策略常见的有哪些?

- STL中hash_map扩容发生什么?

(1) 创建一个新桶，该桶是原来桶两倍大最接近的质数(判断n是不是质数的方法：用n除2到$sqrt(n)$范围内的数）；

(2) 将原来桶里的数通过指针的转换，插入到新桶中(注意STL这里做的很精细，没有直接将数据从旧桶遍历拷贝数据插入到新桶，而是通过指针转换)

(3) 通过swap函数将新桶和旧桶交换，销毁新桶。

26.二叉树

- 二叉树结构，二叉查找树实现;

- 二叉树的六种遍历;

- 二叉树的按层遍历;

- 递归是解决二叉树相关问题的神级方法;

- 树的各种常见算法题
(http://blog.csdn.net/xiajun07061225/article/details/12760493);

27. Trie树(字典树)]

- 每个节点保存一个字符
- 根节点不保存字符
- 每个节点最多有n个子节点(n是所有可能出现字符的个数)
- 查询的复杂父为O(k)，k为查询字符串长度

28. 链表

- 链表和插入和删除，单向和双向链表都要会
- 链表的问题考虑多个指针和递归

(1) 反向打印链表(递归)

(2) 打印倒数第K个节点(前后指针)

(3) 链表是否有环(快慢指针)等等。b ggg

29. 栈和队列

- **队列和栈的区别**?（从实现，应用，自身特点多个方面来阐述，不要只说一个先入先出，先入后出，这个你会别人也会，要展现出你比别人掌握的更深)
- 典型的应用场景

30. 海量数据问题

- 十亿整数（随机生成，可重复）中前K最大的数

类似问题的解决方法思路：首先哈希将数据分成N个文件，然后对每个文件建立K个元素最小/大堆（根据要求来选择）。最后将文件中剩余的数插入堆中，并维持K个元素的堆。最后将N个堆中的元素合起来分析。可以采用归并的方式来合并。在归并的时候为了提高效率还需要建一个N个元素构成的最大堆，先用N个堆中的最大值填充这个堆，然后就是弹出最大值，指针后移的操作了。当然这种问题在现在的互联网技术中，一般就用map-reduce框架来做了。

大数据排序相同的思路：先哈希（哈希是好处是分布均匀，相同的数在同一个文件中），然后小文件装入内存快排，排序结果输出到文件。最后建堆归并。
十亿整数（随机生成，可重复）中出现频率最高的一千个

32. 排序算法

- 排序算法当然是基础内容了，必须至少能快速写出，快排，建堆，和归并
- 每种算法的时间空间复杂度，最好最差平均情况

33. 位运算

```
### 布隆过滤器

几十亿个数经常要查找某一个数在不在里面，使用布隆过滤器，布隆过滤器的原理。布隆过滤器可能出现误判，怎么保证无误差?
```

# 网络与TCP/IP

- TCP与UDP之间的区别

```
(1) IP首部，TCP首部，UDP首部

(2) TCP和UDP区别

(3) TCP和UDP应用场景

(4) 如何实现可靠的UDP
```

- TCP三次握手与四次挥手
- 详细说明TCP状态迁移过程

```
(1) 三次握手和四次挥手状态变化;

(2) 2MSL是什么状态? 作用是什么?
```

- TCP相关技术

    1. TCP重发机制，Nagle算法
    2. TCP的拥塞控制使用的算法和具体过程
    3. TCP的窗口滑动

- TCP客户与服务器模型，用到哪些函数

- UDP客户与服务器模型，用到哪些函数
- 域名解析过程，ARP的机制，RARP的实现

    1. RARP用于无盘服务器，开机后通过发送RARP包给RARP服务器，通过mac地址得到IP地址

36. Ping和TraceRoute实现原理

```
(1) Ping是通过发送ICMP报文回显请求实现。
(2) TraceRoute通过发送UDP报文，设置目的端口为一个不可能的值，将IP首部中的TTL分别设置从1到N，每次逐个增加，如果收到端口不可达，说明到达目的主机，如果是因为TTL跳数超过，路由器会发送主机不可达的ICMP报文。
```

---

# HTTP

**http/https 1.0、1.1、2.0**

1. http的主要特点: **简单快速：**当客户端向服务器端发送请求时，只是简单的填写请求路径和请求方法即可，然后就可以通过浏览器或其他方式将该请求发送就行了 **灵活：** HTTP 协议允许客户端和服务器端传输任意类型任意格式的数据对象 **无连接：**无连接的含义是限制每次连接只处理一个请求。服务器处理完客户的请求，并收到客户的应答后，即断开连接，采用这种方式可以节省传输时间。(当今多数服务器支持Keep-Alive功能，使用服务器支持长连接，解决无连接的问题) **无状态：**无状态是指协议对于事务处理没有记忆能力，服务器不知道客户端是什么状态。即客户端发送HTTP请求后，服务器根据请求，会给我们发送数据，发送完后，不会记录信息。(使用 cookie 机制可以保持 session，解决无状态的问题)

2. http1.1的特点 a、默认持久连接节省通信量，只要客户端服务端任意一端没有明确提出断开TCP连接，就一直保持连接，可以发送多次HTTP请求 b、管线化，客户端可以同时发出多个HTTP请求，而不用一个个等待响应 c、断点续传ftghh

3. http2.0的特点 a、HTTP/2采用二进制格式而非文本格式 b、HTTP/2是完全多路复用的，而非有序并阻塞的——只需一个HTTP连接就可以实现多个请求响应 c、使用报头压缩，HTTP/2降低了开销 d、HTTP/2让服务器可以将响应主动“推送”到客户端缓存中

**get/post 区别**

区别一： get重点在从服务器上获取资源，post重点在向服务器发送数据； 区别二： get传输数据是通过URL请求，以field（字段）= value的形式，置于URL后，并用"?"连接，多个请求数据间用"&"连接，如http://127.0.0.1/Test/login.action?name=admin&password=admin，这个过程用户是可见的； post传输数据通过Http的post机制，将字段与对应值封存在请求实体中发送给服务器，这个过程对用户是不可见的； 区别三：Get传输的数据量小，因为受URL长度限制，但效率较高； Post可以传输大量数据，所以上传文件时只能用Post方式； 区别四： get是不安全的，因为URL是可见的，可能会泄露私密信息，如密码等； post较get安全性较高；

1#### **返回状态码**

200：请求被正常处理 204：请求被受理但没有资源可以返回 206：客户端只是请求资源的一部分，服务器只对请求的部分资源执行GET方法，相应报文中通过Content-Range指定范围的资源。 301：永久性重定向 302：临时重定向 303：与302状态码有相似功能，只是它希望客户端在请求一个URI的时候，能通过GET方法重定向到另一个URI上 304：发送附带条件的请求时，条件不满足时返回，与重定向无关 307：临时重定向，与302类似，只是强制要求使用POST方法 400：请求报文语法有误，服务器无法识别 401：请求需要认证 403：请求的对应资源禁止被访问 404：服务器无法找到对应资源 500：服务器内部错误 503：服务器正忙

**http 协议头相关**

http数据由请求行，首部字段，空行，报文主体四个部分组成 首部字段分为：通用首部字段，请求首部字段，响应首部字段，实体首部字段

**https与http的区别？如何实现加密传输？**

- https就是在http与传输层之间加上了一个SSL
- 对称加密与非对称加密

**浏览器中输入一个URL发生什么，用到哪些协议？**

> 浏览器中输入URL，首先浏览器要将URL解析为IP地址，解析域名就要用到DNS协议，首先主机会查询DNS的缓存，如果没有就给本地DNS发送查询请求。
> DNS查询分为两种方式，一种是递归查询，一种是迭代查询。如果是迭代查询，本地的DNS服务器，向根域名服务器发送查询请求，
> 根域名服务器告知该域名的一级域名服务器，然后本地服务器给该一级域名服务器发送查询请求，然后依次类推直到查询到该域名的IP地址。
> DNS服务器是基于UDP的，因此会用到UDP协议。得到IP地址后，浏览器就要与服务器建立一个http连接。因此要用到http协议，http协议报文格式上面已经提到。
> http生成一个get请求报文，将该报文传给TCP层处理。如果采用https还会先对http数据进行加密。TCP层如果有需要先将HTTP数据包分片，分片依据路径MTU和MSS。TCP的数据包然后会发送给IP层，用到IP协议。IP层通过路由选路，一跳一跳发送到目的地址。当然在一个网段内的寻址是通过以太网协议实现
> (也可以是其他物理层协议，比如PPP，SLIP)，以太网协议需要直到目的IP地址的物理地址，有需要ARP协议。

---

**安全相关**

- SQL注入
- XSS
- RCFS
- APR欺骗

---

# 数据库

- SQL语言(内外连接，子查询，分组，聚集，嵌套，逻辑)

- MySQL索引方法？索引的优化？
- InnoDB与MyISAM区别？
- 事务的ACID
- 事务的四个隔离级别
- 查询优化(从索引上优化，从SQL语言上优化)
- B-与B+树区别？
- MySQL的联合索引(又称多列索引)是什么？生效的条件？
- 分库分表

---

# Linux

## 进程与线程

(1) 进程与线程区别？

(2) 线程比进程具有哪些优势？

(3) 什么时候用多进程？什么时候用多线程？

(4) LINUX中进程和线程使用的几个函数？

(5) 线程同步？

在Windows下线程同步的方式有：互斥量，信号量，事件，关键代码段

在Linux下线程同步的方式有：互斥锁，自旋锁，读写锁，屏障(并发完成同一项任务时，屏障的作用特别好使)

知道这些锁之间的区别，使用场景？

## 进程间通讯方式

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

**命名管道 (FIFO) **： 有名管道也是半双工的通信方式，但是它允许无亲缘关系进程间的通信。

> **信号量**：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，有XSI信号量和POSIX信号量，POSIX信号量更加完善。

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

**共享内存( shared memory ) **：共享内存就是映射一段能被其他进程所访问的内存，这段共享内存由一个进程创建，但多个进程都可以访问。共享内存是最快的 IPC 方式，它是针对其他进程间通信方式运行效率低而专门设计的。它往往与其他通信机制，如信号两，配合使用，来实现进程间的同步和通信。(原理一定要清楚，常考)

> **信号 ( sinal ) **： 信号是一种比较复杂的通信方式，用于通知接收进程某个事件已经发生，常见的信号。

**套接字( socket ) **： 套解口也是一种进程间通信机制，与其他通信机制不同的是，它可用于不同及其间的进程通信。

- **匿名管道与命名管道的区别**：匿名管道只能在具有公共祖先的两个进程间使用。
- **共享文件映射mmap**

mmap建立进程空间到文件的映射，在建立的时候并不直接将文件拷贝到物理内存，同样采用缺页终端。
mmap映射一个具体的文件可以实现任意进程间共享内存，映射一个匿名文件，可以实现父子进程间共享内存。

- **常见的信号有哪些?** ：SIGINT，SIGKILL(不能被捕获)，SIGTERM(可以被捕获)，SIGSEGV，SIGCHLD，SIGALRM

---

## 内存管理

1. 虚拟内存的作用?
2. 虚拟内存的实现?
3. 操作系统层面对内存的管理?
4. 内存池的作用？STL里内存池如何实现?
5. 进程空间和内核空间对内存的管理不同?
6. Linux的slab层，VAM?
7. 伙伴算法
8. 高端内存

---

## 进程调度

1. Linux进程分为两种，实时进程和非实时进程；
2. 优先级分为静态优先级和动态优先级，优先级的范围；
3. 调度策略，FIFO，LRU，时间片轮转
4. 交互进程通过平均睡眠时间而被奖励；

## 死锁

(1) 死锁产生的条件；

(2) 死锁的避免；

### 命令行

- Linux命令 在一个文件中，倒序打印第二行前100个大写字母 cat filename | head -n 2 | tail -n 1 | grep '[[:upper:]]' -o | tr -d '\n'| cut -c 1-100 | rev
- 与CPU，内存，磁盘相关的命令(top，free, df, fdisk)
- 网络相关的命令netstat，tcpdump等
- sed, awk, grep三个超强大的命名，分别用与格式化修改，统计，和正则查找
- ipcs和ipcrm命令
- 查找当前目录以及字母下以.c结尾的文件，且文件中包含"hello world"的文件的路径
- 创建定时任务

### IO模型

- **五种IO模型：**阻塞IO，非阻塞IO，IO复用，信号驱动式IO，异步IO
- **select，poll，epoll的区别**

**select：**是最初解决IO阻塞问题的方法。用结构体fd_set来告诉内核监听多个文件描述符，该结构体被称为描述符集。由数组来维持哪些描述符被置位了。对结构体的操作封装在三个宏定义中。通过轮寻来查找是否有描述符要被处理，如果没有返回**

存在的问题：

1. 内置数组的形式使得select的最大文件数受限与FD_SIZE；
2. 每次调用select前都要重新初始化描述符集，将fd从用户态拷贝到内核态，每次调用select后，都需要将fd从内核态拷贝到用户态；
3. 轮寻排查当文件描述符个数很多时，效率很低；

**poll：**通过一个可变长度的数组解决了select文件描述符受限的问题。数组中元素是结构体，该结构体保存描述符的信息，每增加一个文件描述符就向数组中加入一个结构体，结构体只需要拷贝一次到内核态。poll解决了select重复初始化的问题。轮寻排查的问题未解决。**

**epoll：**轮寻排查所有文件描述符的效率不高，使服务器并发能力受限。因此，epoll采用只返回状态发生变化的文件描述符，便解决了轮寻的瓶颈。

- 为什么使用IO多路复用，最主要的原因是什么?
- epoll有两种触发模式？这两种触发模式有什么区别？编程的时候有什么区别?
- 上一题中编程的时候有什么区别，是在边缘触发的时候要把套接字中的数据读干净，那么当有多个套接字时，在读的套接字一直不停的有数据到达，如何保证其他套接字不被饿死(面试网易游戏的时候问的一个问题，答不上来，印象贼深刻)。

1. select/poll/epoll区别
2. 几种网络服务器模型的介绍与比较
3. epoll为什么这么快(搞懂这篇文章，关于IO复用的问题就信手拈来了)

Linux的API

- **fork与vfork区别** fork和vfork都用于创建子进程。但是vfork创建子进程后，父进程阻塞，直到子进程调用exit()或者excle()。 对于内核中过程fork通过调用clone函数，然后clone函数调用do_fork()。do_fork()中调用copy_process()函数先复制task_struct结构体，然后复制其他关于内存，文件，寄存器等信息。fork采用写时拷贝技术，因此子进程和父进程的页表指向相同的页框。但是vfork不需要拷贝页表，因为父进程会一直阻塞，直接使用父进程页表。

- **exit()与_exit()区别** exit()清理后进入内核，_exit()直接陷入内核。

- 孤儿进程与僵死进程

1. 孤儿进程是怎么产生的?
2. 僵死进程是怎么产生的?
3. 僵死进程的危害?
4. 如何避免僵死进程的产生?

- **Linux是如何避免内存碎片的**

1. 伙伴算法，用于管理物理内存，避免内存碎片;
2. 高速缓存Slab层用于管理内核分配内存，避免碎片。

- **共享内存的实现原理?**

共享内存实现分为两种方式一种是采用mmap，另一种是采用XSI机制中的共享内存方法。mmap是内存文件映射，将一个文件映射到进程的地址空间，用户进程的地址空间的管理是通过vm_area_struct结构体进行管理的。mmap通过映射一个相同的文件到两个不同的进程，就能实现这两个进程的通信，采用该方法可以实现任意进程之间的通信。mmap也可以采用匿名映射，不指定映射的文件，但是只能在父子进程间通信。XSI的内存共享实际上也是通过映射文件实现，只是其映射的是一种特殊文件系统下的文件，该文件是不能通过read和write访问的。

二者区别：

> 1、 系统V共享内存中的数据，从来不写入到实际磁盘文件中去；而通过mmap()映射普通文件实现的共享内存通信可以指定何时将数据写入磁盘文件中。注：前面讲到，系统V共享内存机制实际是通过映射特殊文件系统shm中的文件实现的，文件系统shm的安装点在交换分区上，系统重新引导后，所有的内容都丢失。

> 2、 系统V共享内存是随内核持续的，即使所有访问共享内存的进程都已经正常终止，共享内存区仍然存在（除非显式删除共享内存），在内核重新引导之前，对该共享内存区域的任何改写操作都将一直保留。

> 3、 通过调用mmap()映射普通文件进行进程间通信时，一定要注意考虑进程何时终止对通信的影响。而通过系统V共享内存实现通信的进程则不然。注：这里没有给出shmctl的使用范例，原理与消息队列大同小异。

- 系统调用与库函数(open, close, create, lseek, write, read)

- 同步方法有哪些?

1. 互斥锁，自旋锁，信号量，读写锁，屏障
2. 互斥锁与自旋锁的区别：互斥锁得不到资源的时候阻塞，不占用cpu资源。自旋锁得不到资源的时候，不停的查询，而然占用cpu资源。
3. 死锁

## 其他

- ++i是否是原子操作 明显不是，++i主要有三个步骤，把数据从内存放在寄存器上，在寄存器上进行自增，把数据从寄存器拷贝会内存，每个步骤都可能被中断。
- 判断大小端

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

void fun()
{
union un test;
test.i = 1;
if(ch == 1)
cout << "小端" << endl;
else
cout << "大端" << endl;
}
```

设计模式 单例模式线程安全的写法 STL里的迭代器使用了迭代器模式

- 在有继承关系的父子类中，构建和析构一个子类对象时，父子构造函数和析构函数的执行顺序分别是怎样的?

```
父构造函数  子构造函数  子析构函数  父析构函数
```

- 在有继承关系的类体系中，父类的构造函数和析构函数一定要申明为 virtual 吗? 如果不申明为 virtual 会怎样?

```
    构造函数不能够声明为虚函数:
        1. 构造一个对象需要先确定对象的类型,而虚函数是在运行时确定的，虚构造函数无法在构造对象的时候确定所构造的对象类型;
        2. 虚函数的执行需要依赖于虚函数表。而虚函数表在构造函数中进行初始化工作,即初始化vptr, 让它指向正确的虚函数表。而在构造
            对象期间,虚函数表还没有被初始化,将无法进行。
```

析构函数可以被声明为虚函数,且在某些情况下必须声明为虚函数:
1. 在类的继承中,如果有积累指针执行派生类,那么用基类指针delete是时,如果不定义为虚函数,派生类中派生的那部分无法析构;
构造函数析构函数是否能够调用虚函数:
１． 从语法上讲，调用完全没有问题。
２． 但是从效果上看，往往不能达到需要的目的。
Effective 的解释是:
派生类对象构造期间进入基类的构造函数时，对象类型变成了基类类型，而不是派生类类型。
同样，进入基类析构函数时，对象也是基类类型。
所以，虚函数始终仅仅调用基类的虚函数（如果是基类调用虚函数），不能达到多态的效果，所以放在构造函数中是没有意义的，而且往往不能达到本来想要的效果。

- 什么是虚表？虚表的内存结构布局如何？虚表的第一项（或第二项）是什么？

虚表： 虚函数的地址表,按照虚函数声明的顺序存储,如果子类覆盖父类函数,虚函数表中同样会覆盖;

实现一个 memmov 函数 这个题目考查点在于 memmov 函数与 memcpy 函数的区别，这两者对于源地址与目标地址内存有重叠的这一情况的处理方式是不一样的。 实现strcpy或strcpy函数 这个函数写出来没啥难度，但是除了边界条件需要检查以外，还有一个容易被忽视的地方即其返回值一定要是目标内存地址，以支持所谓的链式拷贝。即： strcpy(dest3, strcpy(dest2, strcpy(dest1, src1))); 实现atoi函数 这个函数的签名如下： int atoi(const char* p)

nagle算法； keepalive选项； Linger选项； 对于某一端出现大量CLOSE_WAIT 或者 TIME_WAIT如何解决； 通讯协议如何设计或如何解决数据包的粘包与分片问题； 心跳机制如何设计；（可能不会直接问问题本身，如问如何检查死链） 断线重连机制如何设计； 对 IO Multiplexing 技术的理解； 收发数据包正确的方式，收发缓冲区如何设计； 优雅关闭； 定时器如何设计； epoll 的实现原理。