***面经整理***

***--xieyongqi***

***C++基础***

***Cpp特性***

封装：指将数据和函数封装成一个类，对外提供函数接口，类的使用者无需关心类内部的实现细节，使类具有安全性、隐蔽性；

继承：指子类能继承父类的数据和函数，同时也具有自己的数据和函数（自我特性）；

多态：指一个接口，多个实现;分为静态多态（重载）和动态多态（虚函数）；静态多态通过重载类内的同名函数达到通过一个同名函数调用得到不同结果；动态多态具体表现为子类通过重写父类的虚函数，达到通过指向子类的父类指针调用该虚函数能准确地调用到子类的该虚函数，隐藏父类的效果；

***变量作用域***

全局变量/全局的静态变量作用域一般是作用于整个程序的，但是当遇到同名的局部变量时，全局变量的这部分作用域将被局部同名变量作用，即全局变量的这部分作用域被局部同名变量使用占用了，被隐藏了。

1 int a;

2 static int b;

3 extern int c;

4 void internal(void \*input){

5 int a;

6 Static int b;

7 }

8 int main (char \*argv, char \*\*argc) {

9 int c;

10 int d;

11 }

全局变量abc作用域如下：

a 1-4 8-11

b 2-5 8-11

c 3-8

**多态类型**

静态多态：通过函数重载和模板来实现，体现在重载的函数在一个类里面，在编译期确定实现；

动态多态：通过虚函数和继承关系实现，体现在父子类之间，在运行（调用）的时候确定；

***虚函数***

虚函数的具体实现是通过虚函数表指针和虚函数表实现的，每个类拥有一张虚函数表，每个类对象拥有一个虚函数指针，当类对象调用虚函数时是通过虚函数表指针找到对应虚函数在虚函数表里面的地址来达到调用的效果；

***多继承***

有几个父类有虚函数表指针，该多继承的类就有几个虚函数表指针；

class a {

Public:

Virtual Void f1(){}

};

Class b {

Public:

Virtual void f2(){}

};

Class c : public a, public b {

Public:

Virtual void f3(){}

};

64位系统下，类a的对象大小为8个字节（一个虚函数表指针），类b对象大小为8个字节（一个虚函数表指针），类c对象大小为16个字节（两个虚函数表指针）；

***父子虚函数调用相关***

<https://blog.csdn.net/u012273328/article/details/51173111>

1**.当在构造函数中调用虚函数时**，虚函数表现为该类中虚函数的行为，即父类构造函数调用虚函数，则虚函数为父类中的虚函数；子类构造函数中调用虚函数，则调用的是子类中的虚函数；

**2.如果不是在构造函数中调用虚函数**，则会首先查看虚函数表，如果有实例实现，则调用实例。比如：父类中有虚函数watchTv，则调用父类中watchTv的，如果子类实现了watchTv，则调用子类的watchTv。

***析构函数能抛出异常吗？***

不能；c++标准指明析构函数不能也不应该抛出异常，c++异常处理模型有责任清除那些由于出现异常所导致的已经失效的对象，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。所以从这个意义来说，析构函数已经成为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。

***初始化列表***

*必须在初始化列表初始化的：*

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

***类型转换***

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。

***内联函数与宏区别***

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))

***数据类型大小***

32位 64位

Unsigned char 1 1

Char 1 1

Unsigned Short2 2

Short 2 2

Unsigned int 4 4

Int 4 4

**Long 4 8**

Float 4 4

Double 8 8

**Void\* 4 8**

Bool 1 1

只有long和void\*变化，其他无变化；

***比较Float数据***

Float无法通过==来判断数据是否对等，只能通过大小、大于等于、小于等于符号来预估该数据的大小值，即有一个精确度；f - 9.9 <= 0.1 即判断一个float类型值f等于10，误差在0.1以内；

***Volatile***

声明变量时，表示该变量的值可能被意想不到的改变，编译器对该变量的代码不再进行优化。每次读取变量的值时，都直接从内存中读取，即不再从寄存器读取；

***Const***

1、修饰数据，表示常量，常量的值不能被改变，与define相比，const常量数据有类型，能进行类型检查，安全；define只是进行简单的替换，无类型检查，不安全；

Const int \*p 表示指针指向的数据不能被修改，指针可以被修改

Int const \*p 表示指针指向的数据不能被修改，指针可以被修改

Int \* const p 表示指针的值不能被修改，指针指向的数据可以被修改

Const int \* const p 表示指针不能被修改，数据也不能被修改

2、修饰类内的数据成员，表示该数据成员不能被修改，类内的const数据成员必须通过初始化列表进行初始化；

3、修饰类的成员函数，实际上是修饰该函数隐藏的this指针，表示该成员函数无法修改该类的数据成员；

4、修改函数的参数，表示该参数在函数体内无法被修改；

5、const对象不可以调用非const函数，只能调用const成员函数，因为非const函数可能修改对象成员变量值。

6、非const对象可以调用任何成员函数，只不过优先调用非const成员函数。

7、const成员函数不可以调用非const成员函数，只能调用const成员函数。因为非const成员函数需要this指针。

8、非const成员函数可以调用任何成员函数。

9、const常量存储在全局/静态/常量区

***Const/define区别联系***

联系：它们都是定义常量的一种方式；

区别：

1、define定义的常量没有类型，只是进行了简单的替换，可能会有多个拷贝，占用的内存空间大，const定义的常量是有类型的，存放在静态存储区，只有一个拷贝，占用的内存空间小。

2、define定义的常量是在预处理阶段进行替换，而const在编译阶段确定它的值。

3、define不会进行类型安全检查，而const会进行类型安全检查，安全性更高。

4、const可以定义函数而define不可以。

***Static***

1. 修饰数据，表示该数据是静态数据，类的静态数据成员为类所拥有，拥有该类的对象共享该静态数据，即只有一份数据的存储，在程序运行过程中一直存在。局部静态变量的作用域在函数体中，全局静态变量的作用域在这个文件里。
2. 修饰函数，表示该函数是静态函数，非类内的静态函数只能在本文件使用，无法通过extern关键字拓展该函数的作用域；类的静态成员函数为类所拥有，通过引用类名调用该函数；
3. 类的静态函数不能调用其他非静态成员函数和非静态数据成员，其他非静态成员函数可以调用该静态函数和静态数据成员；

注意和const的区别。const强调值不能被修改，而static强调唯一的拷贝，对所有类的对象都共用

***Struct/class区别***

1. struct数据默认访问权限为public，class为private
2. Struct默认继承权限为public，class为private
3. Class可以做模板，struct不能

***内存管理***

1. 栈区，存储局部变量、函数参数、函数地址，容量有限，大约为2M，地址增长由高到低，会导致栈溢出，即局部变量太大太多，函数入栈太多导致栈内存用光，运算速度较快（操作系统提供支持，有专门的寄存器）；
2. 堆区，动态内存分配区域，分配和释放需要程序员手动管理，即new/malloc，delete/free需要配对使用，容量较大，操作不当可能会导致内存泄露（分配的地址不需要使用时没有及时释放掉），运算速度较慢（无专门寄存器，库函数实现，速度稍慢）。由于频繁的地址分配和释放，容易产生内存碎片；

**栈的大小在WINDOWS是2M，在linux是8M（软限制），取决于系统内存；具体的硬限制是取决于用户空间的线性地址分配，linux中堆栈的内存起始地址是固定的，并向对方方向增长， 那么栈空间和堆空间最大值总和应该就是这两个固定地址直接的内存大小。(ulimit -a 可以查看到栈空间默认大小)**。

**栈的大小可以通过系统调用来拓展。**

1. 全局变量/静态变量区，存放全局变量，静态变量；初始化的全局变量和静态变量存储在这块区域；未初始化的全局变量和静态变量存储在bss区（block start by symbol）；
2. 代码区；
3. 文字常量区，存放常量字符串，不允许修改（非常规手段修改）；

//main.cpp int a = 0; 全局初始化区

char \*p1; 全局未初始化区

main() { int b; 栈

char s[] = "abc"; 栈

char \*p2; 栈

char \*p3 = "123456"; 123456\0在常量区，p3在栈上。

static int c =0； 全局（静态）初始化区

p1 = (char \*)malloc(10); p2 = (char \*)malloc(20); } 分配得来得10和20字节的区域就在堆区。

strcpy(p1, "123456"); 123456\0放在常量区，编译器可能会将它与p3所指向的"123456"优化成一个地方

*#include   <stdio.h>   
#include   <stdlib.h>   
#include   <sys/time.h>   
#include   <sys/resource.h>   
#include   <unistd.h>   
    
int   main()   
{   
        struct   rlimit   lmt;   
    
        getrlimit(RLIMIT\_STACK,&lmt);   
    
        printf( "rlim\_cur=%u,rlim\_max=%u\n ",lmt.rlim\_cur,lmt.rlim\_max);   
}   
  
在我的机器上（Linux   Kernel版本为2.4.18-3）输出结果为：   
rlim\_cur=8388608,rlim\_max=4294967295   
    
说明当前的堆栈空间为8M，但作为扩展，最大可支持4G（理论值）*

***堆快？栈快？***

栈快一点。因为操作系统会在底层对栈提供支持，会分配专门的寄存器存放栈的地址，栈的入栈出栈操作也十分简单，并且有专门的指令执行，所以栈的效率比较高也比较快。而堆的操作是由C/C++函数库提供的，在分配堆内存的时候需要一定的算法寻找合适大小的内存。并且获取堆的内容需要两次访问，第一次访问指针，第二次根据指针保存的地址访问内存，因此堆比较慢。

***为什么要有new/delete***

malloc/free和new/delete都是用来申请内存和回收内存的。

在对非基本数据类型的对象使用的时候，对象创建的时候还需要执行构造函数，销毁的时候要执行析构函数。而malloc/free是库函数，是已经编译的代码，所以不能把构造函数和析构函数的功能强加给malloc/free。

***Delete/delete[]区别***

delete只会调用一次析构函数，而delete[]会调用每个成员的析构函数

用new分配的内存用delete释放，用new[]分配的内存用delete[]释放

***STL***

Vector：动态数组，地址连续，优点查询快（通过key可以准确查找），缺点插入删除慢（需要移动数据），适用于要求查询速度的场景；扩容机制为原大小的两倍，从原数组拷贝至新开辟的数组，具体为每插入一个新数据就顺带从原数组拷贝一个数据至新数组；

List：双向链表，地址随机，优点插入删除快（把节点从链表取下，释放掉），缺点查询慢（需要从头遍历），适用于插入删除频繁的场景；

Stack：栈，本质是链表，只能从一个方向插入删除的链表，后进先出特性；

Queue：队列，本质是链表，只能从特定方向插入，特定方向删除的链表，特性先进先出；

Deque：双向队列，本质是链表，能从两个方向插入删除；

Map：映射，本质是有序的红黑树，key-value，key唯一，value可以修改，查询速度快，查询时间复杂度O(lg(n))；

Multimap：多重映射，本质也是有序的红黑树，key-value，可以有多个key，value可以修改；

Set：集合，本质也是红黑树，有序，key（value），key就是value，value就是key，唯一，不能修改，只能插入删除；

Multiset，多重集合，key（value）可以修改；

Unordered\_map：无序映射，本质是hashtable哈希表，无序，查询速度快于红黑树，常数级时间复杂度；

Map的erase成员函数返回是剩余元素的个数，vector的erase函数返回是下一个元素的迭代器；

**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。

***哈希冲突处理方式***

1、线性探查。该元素的哈希值对应的桶不能存放元素时，循序往后一一查找，直到找到一个空桶为止，在查找时也一样，当哈希值对应位置上的元素与所要寻找的元素不同时，就往后一一查找，直到找到吻合的元素，或者空桶。

2、二次探查。该元素的哈希值对应的桶不能存放元素时，就往后寻找1^2,2^2,3^2,4^2.....i^2个位置。

3、双散列函数法。当第一个散列函数发生冲突的时候，使用第二个散列函数进行哈希，作为步长。

4、开链法。在每一个桶中维护一个链表，由元素哈希值寻找到这个桶，然后将元素插入到对应的链表中，STL的hashtable就是采用这种实现方式。

5、建立公共溢出区。当发生冲突时，将所有冲突的数据放在公共溢出区。

***指针引用***

· 指针是一个新的变量，指向另一个变量的地址，我们可以通过访问这个地址来修改另一个变量；而引用是一个别名，对引用的操作就是对变量的本身进行操作

· 指针可以有多级，引用只有一级

· 传参的时候，使用指针的话需要解引用才能对参数进行修改，而使用引用可以直接对参数进行修改

· 指针的大小一般是4个字节，引用的大小取决于被引用对象的大小

· 指针可以为空，引用不可以。

***指针引用使用场景***

· 需要返回函数内局部变量的内存的时候用指针。使用指针传参需要开辟内存，用完要记得释放指针，不然会内存泄漏。而返回局部变量的引用是没有意义的

· 对栈空间大小比较敏感（比如递归）的时候使用引用。使用引用传递不需要创建临时变量，开销要更小

· 类对象作为参数传递的时候使用引用，这是C++类对象传递的标准方式

***空类***

Class A{

}；

Sizeof（A）=1

空类的大小是1， 在C++中空类会占一个字节，这是为了让对象的实例能够相互区别。具体来说，空类同样可以被实例化，并且每个实例在内存中都有独一无二的地址，因此，编译器会给空类隐含加上一个字节，这样空类实例化之后就会拥有独一无二的内存地址。当该空白类作为基类时，该类的大小就优化为0了，子类的大小就是子类本身的大小。这就是所谓的空白基类最优化。

空类的实例大小就是类的大小，所以sizeof(a)=1字节,如果a是指针，则sizeof(a)就是指针的大小，即4字节。

***重载重写区别***

1、重载（overload）是指函数名相同，参数列表不同的函数实现方法。它们的返回值可以不同，但返回值不可以作为区分不同重载函数的标志。

2、重写（overwide）是指函数名相同，参数列表相同，只有方法体不相同的实现方法。一般用于子类继承父类时对父类方法的重写。子类的同名方法屏蔽了父类方法的现象称为隐藏

***抽象类***

**含有纯虚函数的类叫抽象类；抽象类不能被实例化；**基类定义有纯虚函数即为抽象类，即派生类必须实现它的纯虚函数，如果不实现，该派生类也为抽象类，也不能被实例化，直至基类的纯虚函数被实现了的派生类才能被示例化；抽象类（基类）一般做接口类；

**类的析构函数可以定义为虚函数，但不能被定义为纯虚函数，因为纯虚函数的类不能实例化，而析构函数的类必须是已经实例化了的**；

**类的构造函数不能被定义为虚函数，因为类的构造函数是编译期确定实现，而虚函数则是运行时确定实现；**

基类的析构函数一般被定义为虚函数，便于派生类对象析构的时候能释放掉派生类部分的内存；如果基类没有被定义为虚函数，则派生类对象析构时调用的是基类的构造函数，只能释放掉基类部分的内存，导致派生类部分内存没有被释放，从而引起内存泄露；

***Windows程序消息机制***

系统将会维护一个或多个[消息队列](https://wenwen.sogou.com/s/?w=%E6%B6%88%E6%81%AF%E9%98%9F%E5%88%97&ch=ww.xqy.chain" \t "https://wenwen.sogou.com/z/_blank)，所有产生的消息都回被放入或是插入队列中。系统会在队列中取出每一条消息，根据消息的接收[句柄](https://wenwen.sogou.com/s/?w=%E5%8F%A5%E6%9F%84&ch=ww.xqy.chain" \t "https://wenwen.sogou.com/z/_blank)而将该消息发送给拥有该窗口的程序的消息循环。每一个运行的程序都有自己的消息循环，在循环中得到属于自己的消息并根据接收窗口的句柄调用相应的窗口过程。而在没有消息时消息循环就将控制权交给系统所以Windows可以同时进行多个任务。

Postmessage()异步执行，sendmessage()同步执行；

***智能指针***

智能指针的思想：资源获取释放，即对象构造析构；

智能指针的作用是管理一个指针，因为存在以下这种情况：申请的空间在函数结束时忘记释放，造成内存泄漏。使用智能指针可以很大程度上的避免这个问题，因为智能指针就是一个类，当超出了类的作用域是，类会自动调用析构函数，析构函数会自动释放资源。所以智能指针的作用原理就是在函数结束时自动释放内存空间，不需要手动释放内存空间。

1、auto\_ptr（c++98方案，c++11已舍弃），采用所有权模式。

auto\_ptr< string> p1 (new string ("I reigned lonely as a cloud.”));

auto\_ptr<string> p2;

p2 = p1; //auto\_ptr不会报错.

此时不会报错，p2剥夺了p1的所有权，但是当程序运行时访问p1将会报错。所以auto\_ptr的缺点是：存在潜在的内存崩溃问题！

1. unique\_ptr，独占式拥有或严格拥有概念，保证同一时间内只有一个智能指针可以指向该对象。指针的所有权无法通过一般方法转移，可通过std::move()来实现赋值转移所有权；
2. shared\_ptr，实现共享式概念，多个智能指针可以指向同一个对象，该对象和其相关资源会在“最后一个引用被销毁”时候释放。从名字share就可以看出了资源可以被多个指针共享，它使用计数机制来表明资源被几个指针共享。可以通过成员函数use\_count()来查看资源的所有者个数。除了可以通过new来构造，还可以通过传入auto\_ptr, unique\_ptr,weak\_ptr来构造。当我们调用release()时，当前指针会释放资源所有权，计数减一。当计数等于0时，资源会被释放。

shared\_ptr 是为了解决 auto\_ptr 在对象所有权上的局限性(auto\_ptr 是独占的), 在使用引用计数的机制上提供了可以共享所有权的智能指针。

1. 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。weak\_ptr是用来解决shared\_ptr相互引用时的死锁问题,如果说两个shared\_ptr相互引用,那么这两个指针的引用计数永远不可能下降为0,资源永远不会释放。它是对对象的一种弱引用，不会增加对象

的引用计数，和shared\_ptr之间可以相互转化，shared\_ptr可以直接赋值给它，它可以通过调用lock函数来获得shared\_ptr。

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。

***字节对齐***

设对齐字节数为n(n = 4或8，区别于32位或者64位操作系统),每个成员内存长度为Li, Max(Li)为最大的成员内存长度,字节对齐规则是:  
 1.结构体对象的起始地址能够被Max(Li)所整除;(一般情况下这条规则满足)  
 2.结构体中每个成员相对于起始地址的偏移量,即对齐值应是min(n,Li)的倍数.若不满足对齐值的要求,编译器会在成员之间填充若干个字节;(这里总是容易出错)  
 3.结构体的总长度值应是min(n,Max(Li))的倍数,若不满足总长度值的要求,编译器在为最后一个成员分配空间后,会在其后填充若干个字节.

1. #pragma pack（2/4/8） 其他预计无效，最终是以4/8对齐（在linux上实践一波）

作用：减少cpu读取的次数，提高读取效率；

***主机网络序***

主机字节序，大端序小端序  
网络字节序：大端序  
htonl()--"Host to Network Long"  
      ntohl()--"Network to Host Long"  
      htons()--"Host to Network Short"  
      ntohs()--"Network to Host Short"

***Volatile***

访问寄存器要比访问内存要块，因此CPU会优先访问该数据在寄存器中的存储结果，但是内存中的数据可能已经发生了改变，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。

一个参数可以即是const又是volatile的吗？可以，一个例子是只读状态寄存器，是volatile是因为它可能被意想不到的被改变，是const告诉程序不应该试图去修改他。

***定位内存泄露***

Gdb调试，程序打印日志，coredump，工具valgrind；

***构造函数退化***

在C++中，如果确定了某一个构造函数的创建过程，在该构造函数中如果调用了其它重载的构造函数，它将不会执行其它构造函数的初始化列表部分代码，而是执行函数体代码，此时已经退化成普通函数了。例子说明如下：

class CBook {

public:

   double m\_price;

   CBook() {

       CBook(8.8);

    }

   CBook(double price) : m\_price(price) { }

};

int main() {

   CBook c;

   cout << c.m\_price << endl; // 此时并不会输出理想中的8.8

}

***C/C++结构体***

（1）C的结构体内不允许有函数存在，C++允许有内部成员函数，且允许该函数是虚函数。所以C的结构体是没有构造函数、析构函数、和this指针的。

（2）C的结构体对内部成员变量的访问权限只能是public，而C++允许public,protected,private三种。

（3）C语言的结构体是不可以继承的，C++的结构体是可以从其他的结构体或者类继承过来的。

以上都是表面的区别，实际区别就是面向过程和面向对象编程思路的区别：

C的结构体只是把数据变量给包裹起来了，并不涉及算法。

而C++是把数据变量及对这些数据变量的相关算法给封装起来，并且给对这些数据和类不同的访问权限。

C语言中是没有类的概念的，但是C语言可以通过结构体内创建函数指针实现面向对象思想。

***C++调用C***

#ifdef \_\_cplusplus

extern “C”{

}

#endif

指明以c obj文件的方式编译，链接时按照c的命名规则去查找链接；

***问题***

1. 线程等待条件变量时为什么要传入互斥锁？

先安全地访问互斥锁保护的全局变量，判断条件是否成立，如果不成立，则释放锁，线程挂起；等到一个使条件成立时，发出信号唤醒挂起的一个或者多个线程，在lock互斥锁，被唤醒的线程从pthread\_cond\_wait中返回；

确保条件变量等待前锁被隐含地释放了。

pthread\_cond\_wait做的事：

1. Unlock
2. Waiting挂起
3. 条件满足Lock 返回
4. 自旋锁对比互斥锁，为什么要禁用本地中断和抢占？

防止中断上下文中请求自旋锁忙等待造成的死锁。

1. 链接主要的工作？

符号解析，重定位；

1. http如何确认http传输内容大小？
2. 通过http头content-length获取到内容大小
3. 通过chunked编码实时发送消息，chunked会带上每次传输内容的大小；

***计算机网络***

***HTTPS***

非对称加密（ssl握手过程）和对称加密（连接建立传输数据过程）。

***CA 证书内容***

Certificate authority

颁发者   
使用者   
版本   
签名算法   
签名哈希算法   
使用者   
公钥   
指纹   
指纹算法 ...

**颁发者**就是**CA机构**，下面会讲到。对于**签名算法、签名哈希算法**的理解，首先要先理解签名是什么东东？联系到实际情况，当我们向某机构提供报告时，往往在报告最后加上个人的名字，以表示该报告是我本人的。签名在网络通讯中的应用称为**数字签名，当服务器向客户端发送信息时，会将报文生成报文摘要，同时对报文摘要进行hash计算，得到hash值，然后对hash值进行加密，然后将加密的hash值放置在报文后面，这个加密后的hash值就称为签名**。服务器将报文、签名和数字证书一同发送给客户端。客户端收到这些信息后，会首先验证签名，利用**签名算法**对签名进行解密，得到报文摘要的hash值，然后将得到的报文生成报文摘要并利用**签名hash算法**生成新的hash值，通过对比这两个hash值是否一致，就能判断信息是否完整，是否是由真正的服务器发送的。可知**签名有两个作用确认消息发送方可靠，确认消息完整准确。**

**对称加密**就是发送双发使用相同的密钥对消息进行加解密，常见的对称加密为DES、3DES,AES等。**非对称加密**是发送双方各自拥有一对公钥私钥，其中公钥是公开的，私钥是保密的。当发送方向接收方发送消息时，发送方利用接收方的公钥对消息进行加密，接收方收到消息后，利用自己的私钥解密就能得到消息的明文。其中非对称加密方法有RSA、Elgamal、ECC等。

***https握手流程***：

客户端访问服务器（比如：https://www.12306.cn）,发送ssl版本、客户端支持的加密算法、随机数等消息。  
服务器向客户端发送ssl版本、随机数、加密算法、证书（证书出现了）等消息。  
客户端收到消息后，判断证书是否可信（如何判断可信，看下文介绍），若可信，则继续通信，发送消息包括：向服务器发送一个随机数，从证书中获取服务器端的公钥，对随机数加密；编码改变通知，表示随后信息都将使用双方协定的加密方法和密钥发送；客户端握手结束通知

***验证ca证书***

一般来说，现在公共网站数据传输都是使用SSL，即通过https协议访问。Https就是http加SSL。在上面SSL握手过程中，客户端是如何验证服务器发送的CA证书呢？我们以12306网站为例进行说明，此时，客户端就是浏览器，如果我们是第一次访问12306网站，使用https://www.12306.cn地址访问，返回如下结果：

提示此网站的安全证书有问题，说明证书不可信，如果我们忽略证书不可信，继续访问网站，打开12306网页，在首页提供一个根证书的下载，见下页面：

下载根证书，安装完毕，再次访问12036网站，就不会提示网站的安全证书有问题了。这是什么机制呢？

打开【工具】菜单（[360浏览器](https://www.baidu.com/s?wd=360%E6%B5%8F%E8%A7%88%E5%99%A8&tn=24004469_oem_dg&rsv_dl=gh_pc_csdn)为例），然后选择【内容】选项卡，点击【证书】按钮，打开证书窗口，选择【中级证书颁发机构】，会看到已经安装的12306的证书：

只要安装上12306证书，就说明证书是可信的。使用https协议访问时，服务器发送证书向浏览器时，首先查找该证书是否已在信任列表中，然后对证书进行校验，校验成功，那么就证明证书是可信的。另外，证书的认证是安装证书链执行的，证书链的意思是有一个证书机构A，A生成证书B，B也可以生成证书C，比如在证书窗口中有一个360证书，见下图：

360证书的颁发者是certification authority of wosign，在【受信任的根证书颁发机构】中，我们会看到该证书：见下图，

也就是说，certification authority of wosign生成360证书，360可以生成其它的证书。A证书或者certification authority of wosign证书在整个证书链上就被称为根证书，证书验证的机制是只要根证书是受信任的，那么它的子证书都是可信的。比如说，我们使用https协议访问了需要360证书的网站，即使我们不安装360证书，那么网站也不会提示证书不安全，因为，生成360证书的根证书certification authority of wosign证书，在受信任的证书列表中。如果一个证书的根证书是不可信的，那么这个证书肯定也是不可信任的。

由以上可知，根证书在证书验证中极其重要，而且，根证书是无条件信任的，只要我们将根证书安装上，就说明我们对根证书是信任的。比如我们安装12306的根证书，是出于我们对国家的信任，对网站的信任，我们才放心安装这个根证书。对于一些不安全的网站的证书，一定要慎重安装。

另外需要知道的是，【受信任的根证书颁发机构】中的证书是windows预先安装的一些证书，都是国际上很有权威的证书机构，他们证书的生成都有很严格的流程，因此他们的证书被认为是安全，就像我们相信银行是安全，所以把钱存入到银行。

***CA证书链***

CA证书链是在验证对端证书时，用来寻找该证书所隶属的信任CA所需的一个系列CA证书的集合。也就是说，CA证书链用于追踪底层CA到实体所信任的根CA之间的证书路径，以便实体确认对端的证书是可信任的。

***HTTP***

***特点：***

1. 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   
    2、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   
    3、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)   
    4、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)

***http1.1特点***

1. 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（请求头range响应头content-range，请求部分，多次请求）

***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；

**操作系统**

***软链接硬链接***

1. 硬链接，类似于智能指针引用，指向同一个文件的区块，只有当该文件区块的引用计数为0，才会彻底删除该文件；

2、软链接，保存的是另一个文件的绝对路径，是另外一种文件类型；

数据库

***Redis***

***简介：***

redis是一个基于内存，单进程单线程架构的分布式key-value非关系数据库，被广泛应用于缓存方向，做分布式缓存（数据一致性）；

特点：

数据加载于内存，访问数据就是直接访问内存，不需要再从磁盘上读取；

支持数据持久化，默认支持RDB方式持久化（快照），也支持AOF（追加文件）；

数据类型丰富，string，hash，list，set，zset；

支持lua脚本，多种集群方案，支持事务；

***Memcache区别***

1、redis支持更丰富的数据类型，redis不仅支持简单的string，还支持list，hash，zset等数据结构的存储；memcache只支持string数据类型；

2、Redis支持数据的持久化，可以将内存汇总的数据保持在磁盘中，重启的时候可以再次加载进行使用；而memcache仅仅将数据存在于内存中；

3、集群模式，memcache没有原生的集群模式，需要依靠客户端来实现往集群中分片写入数据，但是目前redis是原生支持cluster模式的；

4、memcache是多线程的，非阻塞io复用的网络模型；redis是单线程的多路io复用模型；

***过期时间***

Redis可以给key设置过期时间，当key超过过期时间后，redis采用的定期删除+惰性删除；

**定期删除**：redis默认是每个100ms就随机抽取一些设置了过期时间的key，检查其是否过期，如果过期就删除；

**惰性删除**：靠定期删除没有被删除的过期key，还停留在内存中，除非你的系统去查一下那个key，才会被redis给删除掉，这就是所谓的惰性删除；

但是，key数据巨大，当很多key超时时并不能及时删除掉，久而久之，就会造成内存用完，数据无法再插入，怎么办？

Redis内存淘汰机制；

***内存淘汰机制***

可以通过配置文件配置，redis.conf

1. **valatile-lru**：从已设置过期时间的数据集中挑选最近最少使用的数据淘汰；
2. **volatile-ttl**：从已设置过期时间的数据集中挑选将要过期的数据淘汰；
3. **volatile-random**：从已设置过期时间的数据集中任意选择数据淘汰；
4. **Allkeys-lru**: 当内存不足以容纳写入数据时，在键控件中，移除最近最少使用的key
5. **allkeys-random**：从数据集中任意淘汰数据；
6. **no-eviction**：禁止驱逐数据，就是说当内存不足时，禁止写入数据；

***持久化机制***

**RDB：**快照，某一个时刻的数据快照文件，可能会造成卡顿，因为快照是周期性的操作，可能会丢失一个周期的数据，不具备实时性；

好处是：设备重启，数据恢复时速度较快；

**AOF：**追加文件，记录redis数据操作的每条指令，默认是每一秒写入一次，最多丢失一秒的数据，实时性较强；缺点是恢复数据时间较长，需要重新执行一遍数据操作指令；

***事务***

Redis通过multi，exec，watch命令来实现事务的功能；事务将多个命令打包，然后一次性按顺序地执行多个命令的机制，并且在事务执行期间，服务器不会中断事务去执行其他客户端其你去，会执行完事务的命令；

事务具有原子性，一致性，隔离性，持久性；

***常见异常***

1. **缓存穿透**：外部数据请求穿过缓存（redis）直接去请求数据库（缓存不存在），导致数据库压力飙升，甚至崩溃；

解决方案：将不存在的数据访问结果也存储到缓存中；

1. **缓存雪崩**：当缓存重启或者大量的缓存在某一时间段失效，这样就导致大批流量直接访问数据库，对DB造成压力，从而引起DB崩溃；

解决方案：根据数据热度设置不同过期时间，热点数据过期时间稍长，冷数据则过期时间短；缓存的过期时间设置一个区间，取随机；提前预估DB能力，如果缓存挂掉，数据库仍可以在一定程度上抗住流量压力；

1. **缓存预热**：系统上线后，将相关的缓存数据直接加载到缓存系统。这样就可以避免用户在请求的时候，先查询数据库；再讲数据缓存的问题；

解决方案：1.数据量不大的时候，工程启动的时候进行加载缓存动作；2、数据量大的时候，设置一个定时任务脚本，进行缓存的刷新；3、数据量太大的时候，要优先保证热点数据提前加载到缓存中；

1. **缓存降级**：缓存失效的时候或者缓存服务挂掉的时候；我们也不去访问数据库；我们直接访问内存部分数据缓存或者直接返回默认数据；

***集群模式***

**主从模式**：

一台主服务器，多台从服务器；主服务器提供读写功能，写数据后自动将数据同步至从服务器（非阻塞），从服务器提供读功能，可以分散主服务的压力，实现读写分离；从服务器之间可以彼此连接和同步请求，减少主服务同步的压力

**缺点**：

不具备容错和恢复能力，主服务存在单点风险；redis的主从赋值采用全量复制，需要服务器有足够的内存；主从模式难在线扩容；

**哨兵模式**：

redis提供sentinel机制，通过哨兵机制启动redis后，自动监控主从服务器的运行状态，基本原理是心跳机制+投票裁决；

**哨兵作用**：

**监控**，sentinel会定期检查主从服务器是否处于正常工作状态；

**提醒**：当被监控的某个redis服务器出现异常是，sentinel可以通过api向管理员或者其他应用程序发送通知；

**自动故障迁移**，当一个主服务器不能正常工作时，sentinel会开始一次自动故障迁移操作，它会将失效主服务器的其中一个从服务器升级为新的主服务器，集群也会想客户端返回新主服务器的地址，使得集群可以使用新主服务器代表失效服务器；

**优点**：

哨兵模式主从可以切换，具备基本的故障转移能力；哨兵模式具有主从模式的所有优点

**缺点**：

哨兵模式也难支持在线扩容，集群的配置信息管理比较复杂；

**Cluster集群模式**：

无中心结构，一个节点连接其他所有节点，保存数据和整个集群状态；

**优点**：

无中心架构，节点间数据共享，可动态调整数据分布；

节点可动态添加和删除，扩展性灵活

部分节点异常，不影响整体集群的可用性；

**缺点**：

集群实现比较复杂

批量操作支持有限；

事务操作支持有限；

***Key并发竞争***

所谓的redis的并发竞争key的问题也就是多个系统同时对一个key操作，但是最后执行的顺序和我们期待的顺序不一致，导致结果的不同。

解决方案：分布式锁（zookeeper和redis都可以实现分布式锁）；

***数据一致性***

如何保证缓存与数据库双写的数据一致性？

要求不严格，则将读请求和写请求串行化，串到一个内存队列里，串行化后会导致系统的吞吐量大幅度降低。

Mysql