面试问题

1.抽象类和接口的区别

- 1.抽象类是一种只能定义类型，不能生成对象的类。
    - 2.c++中定义的纯虚函数的类为抽象类。
- 2.c++中接口是一种特殊的抽象类。
    - 1.类中没有定义任何成员变量。
    - 2.类中所有成员函数都是公有且都是纯虚函数。

2.c++构造单例模式

- 1.饿汉模式

```
class Singleton {
public:
  static Singleton* GetInstance() {
    return singleton_;
  }

  static void DestreyInstance() {
    if (singleton_ != NULL) {
      delete singleton_;
    }
  }

private:
  // 防止外部构造。
  Singleton() = default;

  // 防止拷贝和赋值。
  Singleton& operator=(const Singleton&) = delete;
  Singleton(const Singleton& singleton2) = delete;

private:
  static Singleton* singleton_;
};

Singleton* Singleton::singleton_ = new Singleton;
```

- 2.懒汉模式

```
#include <mutex>

class Singleton {
public:
  static Singleton* GetInstance() {
    if (instance_ == nullptr) {
      std::lock_guard<std::mutex> lock(mutex_);
      if (instance_ == nullptr) {
        instance_ = new Singleton;
      }
    }

    return instance_;
  }

  \~Singleton() = default;
  // 释放资源。
  void Destroy() {
    if (instance_ != nullptr) {
      delete instance_;
      instance_ = nullptr;
    }
  }

  void PrintAddress() const {
    std::cout << this << std::endl;
  }

private:
  Singleton() = default;

  Singleton(const Singleton&) = delete;
  Singleton& operator=(const Singleton&) = delete;

private:
  static Singleton* instance_;
  static std::mutex mutex_;
};

Singleton* Singleton::instance_ = nullptr;
std::mutex Singleton::mutex_;
```

c++堆操作:

- 1.建堆

```
vector<int> res;
make_heap(res.begin(),res.end(),less<int>()); //默认大顶堆，使用greater<int>
```

```
()可以得到小顶堆。
```

- 2.弹出元素

```
pop_heap(res.begin(),res.end());
res.pop_back();
```

- 3.元素入堆

```
res.push_back(tmp);
push_heap(res.begin(),res.end());
```

- 4.堆排序

```
sort_heap(res.begin(),res.end());
```

函数模板和类模板的区别

- 1.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；类模板的实例化必须由程序员在程序中显式地指定。

消除模板代码膨胀

- 把C++模板中与参数无关的代码分离出来。也就是让与参数无关的代码只有一份拷贝。
  - 1.模板生成多个类和多个函数，所以任何模板代码都不该与某个造成膨胀的模板参数产生相依关系。
  - 2.因非类型模板参数而造成的代码膨胀，往往可消除，做法是以函数或类成员变量替换template参数。
  - 3.因类型参数而造成的代码膨胀，往往可降低，做法是让带有完全相同的二进制表述的具现类型共享实现码。

栈展开

- 栈展开：抛出异常时，将暂停当前函数的执行，开始查找匹配的catch子句。首先检查throw本身是否在try块内部，如果是，检查与该try相关的catch子句，看是否可以处理该异常。如果不能处理，就退出当前函数，并且释放当前函数的内存并销毁局部对象，继续到上层的调用函数中查找，直到找到一个可以处理该异常的catch。

构造函数和析构函数为什么没有返回值？

- 如果它们有返回值，要么编译器必须知道如何处理返回值，要么就只能由客户程序员自己来显式的调用构造函数与析构函数，这样一来，安全性就被人破坏了。

析构函数为什么没有任何参数？

- 因为析构不需任何选项。

为什么拷贝构造函数的参数类型必须是引用？

- 如果采用传值的方式，而传值的方式会调用该类的拷贝构造函数(生成一个临时对象类)，会造成无穷递归的调用拷贝构造函数。传指针也是传值。只有传引用不是传值外，其他所有的传递方式都是传值。

为什么内联函数，构造函数，静态成员函数和友元函数不能为**virtual**函数？

- 1.内联函数
  - 内联函数是在编译时期展开,而虚函数的特性是运行时才动态联编,所以两者矛盾,不能定义内联函数为虚函数。
- 2.构造函数
  - 构造函数用来创建一个新的对象,而虚函数的运行是建立在对象的基础上,在构造函数执行时,对象尚未形成,所以不能将构造函数定义为虚函数。
- 3.静态成员函数
  - 静态成员函数属于一个类而非某一对象,没有this指针,它无法进行对象的判别。
- 4，友元函数
  - C++不支持友元函数的继承，对于没有继承性的函数没有虚函数。
- virtual意味着在执行时期进行绑定，所以在编译时刻需确定信息的不能为virtual。
- virtual意味着派生类可以改写其动作。

继承机制中对象之间是如何转换的？

- 在子类的实例对象所在的内存中，一般而言内存顶部是基类的内容。如果将一个子类对象赋予一个基类对象（非指针和引用），那么会发生自动截断，仅仅保留父类内容。子类对象值可以赋予基类对象，但是基类对象值赋予子类对象会产生问题。
- 子对象转换为父对象可能会造成切片（内存截断），子类对象就变成父类对象了。

继承机制中引用和指针之间如何转换？

- C++中的指针仅仅表示这个指针值所指向的那块内存地址的大小而已。对于基类指针来说，指向的内存块也仅仅是基类的范围，指不到子类的范围，但是子类的指针会比基类指针范围大，如果将基类对象赋予子类指针，会产生问题。
- 使用指针转换，基类对象指针转为子类对象指针，基类对象并没有变成子类对象，仅仅是基类指针所指的类型变了，指针所指的类型变了，那么用指针能引用的对象成员也就变了。指针所指的对象本质没变，因此指针间的转换不会造成切片。

如何定义一个只能在堆上（**new**）或者栈上实例化的类？

- 1.只能在堆上实例化的类
  - 只能在堆上实例化的类的意思就是不能用A a这种形式来进行类对象的实例化，只能用A* a = new A这种形式来进行实例化。
  - 将析构函数设为私有，类对象就无法建立在栈上。

- 因为将析构函数设为私有，类中必须提供一个destory函数，来进行内存空间的释放。类对象使用完成后，必须调用destory函数。
- 为解决继承问题，将析构函数设为protected。

```
class A{
public:
        void destroy(){
                delete this
        }
private:
        ~A(){}
}
```

- 2.只能在栈上实例化的类
  - 只有使用new运算符对象才会建立在堆上，所以将new运算符私有化之后就可以让类的对象无法实例化在堆上。

```
class A{
private:
        void * operator new(size_t){}
        void operator delete(void * ptr){}
public:
        A(){}
        ~A(){}
};
```

虚函数 vs 纯虚函数

- 虚函数是为了允许用基类的指针来调用子类的这个函数，不代表函数为不被实现的函数
- 纯虚函数代表函数没有被实现，定义纯虚函数是为了实现一个接口，起到一个规范的作用，规范继承这个类的程序员必须实现这个函数。

析构函数能抛出异常吗？

- 不能。
  - 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。

必须在构造函数初始化列表里进行初始化的数据成员

- 1.const成员
- 2.引用类型

- 3.没有默认构造函数

构造函数中赋值 **vs** 初始化列表赋值

- 1.构造函数中赋值，先调用默认构造函数，再调用拷贝构造函数。
- 2.初始化列表只调用一次拷贝构造函数赋值。

局部变量和全局变量是否可以同名**?**

- 能。局部会屏蔽全局。要用全局变量，需要使用 "::"(域运算符)。

Debug 版本和 **Release** 版本的区别**?**

- Debug 版本是调试版本，Release 版本是发布给用户的最终非调试的版本。

sizeof小结

- sizeof计算的是在栈中分配的内存大小。
    - sizeof不计算static变量占得内存。
    - 32位系统的指针的大小是4个字节，64位系统的指针是8字节，而不用管指针类型。
    - char型占1个字节，int占4个字节，short int占2个字节，long int占4个字节，float占4字节，double占8字节，string占4字节。
    - 一个空类占1个字节，单一继承的空类占1个字节，虚继承涉及到虚指针所以占4个字节。
    - 数组的长度
        - 若指定了数组长度，则不看元素个数，总字节数=数组长度*sizeof（元素类型）。
        - 若没有指定长度，则按实际元素个数类确定。
        - 若是字符数组，则应考虑末尾的空字符。
    - 结构体长度需要对齐。

sizeof与**strlen**的区别

- 1.sizeof的返回值类型为size_t（unsigned int），strlen返回类型也是size_t。
- 2.sizeof是运算符。而strlen是函数。
- 3.sizeof可以用类型做参数，其参数可以是任意类型的或者是变量、函数。而strlen只能用char*做参数，且必须是以'\0'结尾。
- 4.数组作sizeof的参数时不会退化为指针，而传递给strlen是就退化为指针。
- 5.sizeof是编译时的常量，而strlen要到运行时才会计算出来，且是字符串中字符的个数而不是内存大小。
- 6.sizeof返回静态数组的空间字节数。sizeof返回动态数组的元素指针字节数。
    - 静态数组是在栈中申请的。
    - new动态数组是在堆中的分配的。

孤儿进程和僵尸进程

- 1.孤儿进程：一个父进程退出，而它的一个或多个子进程还在运行，那么那些子进程将成为孤儿进程。孤儿进程将被init进程(进程号为1)所收养，并由init进程对它们完成状态收集工作。
- 2.僵尸进程：一个进程使用fork创建子进程，如果子进程退出，而父进程并没有调用wait或waitpid获取子进程的状态信息，那么子进程的进程描述符仍然保存在系统中。

- 孤儿进程并没有什么危害，init进程会托管孤儿进程。
- 如果大量的产生僵尸进程，将因为没有可用的进程号而导致系统不能产生新的进程。
    - 如果进程不调用wait/waitpid的话， 那么保留的那段信息就不会释放，其进程号就会一直被占用，但是系统所能使用的进程号是有限的。

数组与指针的区别

- 1.数组要么在静态存储区被创建（如全局数组），要么在栈上被创建。指针可以随时指向任意类型的内存块。
- 2.当数组作为函数的参数进行传递时，该数组自动退化为同类型的指针。

堆和栈的区别

- 1.空间分配区别
    - 1.栈：由操作系统自动分配释放 ，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。地址向低址增长。
    - 2.堆：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回收，分配方式倒是类似于链表。地址向高址增长。
- 2.缓存方式区别
    - 1.栈使用的是一级缓存， 他们通常都是被调用时处于存储空间中，调用完毕立即释放。
    - 2.堆是存放在二级缓存中，生命周期由虚拟机的垃圾回收算法来决定（并不是一旦成为孤儿对象就能被回收）。所以调用这些对象的速度要相对来得低一些。
- 3.大小
    - 1.栈的空间比较小，和操作系统有关。
    - 2.堆的空间大小，接近于虚拟内存的大小。

什么时候必须重写拷贝构造函数?

- 当构造函数涉及到动态存储分配空间时，要自己写拷贝构造函数，并且要深拷贝。

不允许重载的5个运算符

- 1.*（成员指针访问运算符号）
- 2.:: 域运算符
- 3.sizeof 长度运算符号
- 4.?: 条件运算符号
- 5. .（成员访问符）

流运算符为什么不能通过类的成员函数重载?一般怎么解决?

- 因为通过类的成员函数重载必须是运算符的第一个是自己，而对流运算的重载要求第一个参数是流对象.
- 一般通过友元来解决。

对象间是怎样实现数据的共享的?

- 通过类的静态成员变量来实现对象间的数据共享。静态成员变量占有自己独立的空间不为某个对象所私有。

对对象成员进行初始化的次序是什么?

- 它的次序完全不受它们在初始化表中次序的影响，只有成员对象在类中声明的次序来决定的。

如何定义和实现一个类的成员函数为回调函数?

- 定义一个类的成员函数时在该函数前加 CALLBACK 即将其定义为回调函数，函数的实现和普通成员函数没有区别。
- 所谓的回调函数，就是预先在系统的对函数进行注册，让系统知道这个函数的存在，以后，当某个事件发生时，再调用这个函数对事件进行响应。

构造函数和析构函数的调用顺序

- 1.构造函数的调用顺序：基类构造函数————对象成员构造函数————派生类构造函数。
- 2.析构函数的调用顺序：派生类析构函数————基类析构函数。

不能做switch()的参数类型

- 1.实型

C++ 是不是类型安全的？

- 1.不是。两个不同类型的指针之间可以强制转换。

class VS struct

- struct 的成员默认是公有的，而class的成员默认是私有的。
- 除此之外，再无区别。struct也可以继承，含成员函数，友元等。

c中struct与c++中struct的区别

- c++中Struct是抽象数据类型（ADT）（Abstract Data Type），支持成员函数的定义，是一种对象的实现体。
- c语言中Struct是用户自定义数据类型（UDT）（User Defined Type），是一种数据结构的实现体。成员不可以是函数。

海量数据

- 1.最大的k个数
- 2.出现频率最大的k个数
    - 1.映射切分(分治)。使用哈希把数据分散到n个文件中。
    - 2.对每个切分小文件进行统计。找到每个文件出现次数最多的数。
    - 3.排序。得到top n个频率数，使用最小堆排序，得到出现频率最大的k个数。

例题

- 1.给定100亿个整数，设计算法找到只出现一次的整数
    - 1.映射切分(分治)。使用哈希把数据分散到n个文件中。
    - 2.对每个文件用位图统计只出现一次的数。

    - 3.使用位图对第二步得到的数处理。
- 2.给两个文件，分别有100亿个整数，我们只有1G内存，如何找到两个文件交集
    - 1.把两个文件分别分成n份。
    - 2.每次使用两个位图统计数据是否出现，结果按位与得到最终结果。

.bss段和.data段

- 1.bss段是用来存放未初始化的全局变量和静态变量的一块内存区域。属于静态内存分配。
- 2.data段时用来存放已初始化的全局变量和静态变量的内存区域，常量也在这个data段。属于静态内存分配。
- 3.heap堆用来存放进程再运行中被动态分配的内存段。
- 4.stack栈用来存放局部遍历的内存段。
- 5.text段是程序代码段，存放程序代码，编译时确定，只读。

程序地址

- 从低地址到高地址依次是：text段-->data段-->bss段-->堆-->栈-->环境变量。
- 其中堆和栈之间有很大的地址空间空闲着，在需要分配空间的时候，堆向上增长，栈向下增长。这样设计可以使得堆和栈能够充分利用空闲的地址空间。

linux文件

- 1.linux系统把一切资源都当成是文件。
- 2.文件类型
    - 1.普通文件。
    - 2.目录文件。
    - 3.链接文件。用于不同目录下文件下文件的共享。
    - 4.设备文件。用来访问硬件设备。
    - 5.命令管道。用来进行进程之间的通信。

malloc和mmap的区别

- 1.malloc
    - 调用malloc()时，是在PCB表(进程表)结构中的堆重点内容中申请空间，若申请空间失败，即超过给定的堆最大空间时，将会调用brk()系统调用，将堆空间向未使用的区域扩展，brk()之后新增的堆空间不会自动清除，需使用相应的系统调用来清除。
- 2.mmap
    - mmap并不分配空间, 只是将文件映射到调用进程的地址空间里。
    - 调用mmap()系统调用使得进程之间通过映射同一个普通文件实现共享内存。普通文件被映射到进程地址空间后，进程可以像访问普通内存一样对文件进行访问，不必再调用read()，write()等操作。

linux的虚拟内存管理

- 1.每个进程都有独立的虚拟地址空间，进程访问的虚拟地址并不是真正的物理地址。
- 2.虚拟地址可通过每个进程上的页表(在每个进程的内核虚拟地址空间)与物理地址进行映射，获得真正物理地址。

- 3.如果虚拟地址对应物理地址不在物理内存中，则产生缺页中断，真正分配物理地址，同时更新进程的页表；如果此时物理内存已耗尽，则根据内存替换算法淘汰部分页面至物理磁盘中。

Linux的虚拟地址空间(从低地址到高地址)

- 1.只读段(text)：该部分空间只能读，不可写，(包括：代码段(放置二进制代码)、rodata 段(C常量字符串和#define定义的常量))。
- 2.数据段(data,bss)：保存全局变量、静态变量的空间。
- 3.堆(heap)：就是平时所说的动态内存， malloc/new 大部分都来源于此。其中堆顶的位置可通过函数 brk 和 sbrk 进行动态调整。
- 4.文件映射区域：如动态库、共享内存等映射物理空间的内存，一般是 mmap 函数所分配的虚拟地址空间。
- 5.栈(stack)：用于维护函数调用的上下文空间，一般为8M可通过 ulimit –s 查看。
- 6.内核虚拟空间：用户代码不可见的内存区域，由内核管理(页表就存放在内核虚拟空间)。

进程分配内存(虚拟内存)

- 1.由mmap系统调用完成
  - 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。
- 2.由brk系统调用完成
  - 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。

栈为什么比堆快

- 1.寄存器直接对栈进行访问（esp，ebp），而对堆访问，只能是间接寻址。
- 2.栈中数据cpu命中率更高，满足局部性原理。
- 3.栈是编译时系统自动分配空间，而堆是动态分配（运行时分配空间），所以栈的速度快。
- 4.栈是先进后出的队列结构，比堆结构相对简单，分配速度大于堆。

malloc的底层原理

- 1.malloc小于128k的内存，使用系统调用brk分配内存，将_edata(_edata指针（glibc里面定义）指向数据段的最高地址。)往高地址推(只分配虚拟空间，不对应物理内存(因此没有初始化)，第一次读/写数据时，引起内核缺页中断，内核才分配对应的物理内存，然后虚拟地址空间建立映射关系)。
- 2.malloc大于128k的内存，使用系统调用mmap分配内存，在堆和栈之间找一块空闲内存分配(对应独立内存，而且初始化为0)。
  - 因为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。

free的底层原理

- 1.当free由brk分配的内存且高地址还有未释放的内存时，_edata指针并不回退，只是系统这块内存可以被重用，虚拟内存和物理内存都没有被释放。
- 2.当free由brk分配的内存且高地址没有未释放的内存时，_edata指针回退，虚拟内存和物理内存都被释放了。
- 3.当free由mmap分配的内存，直接释放内存。虚拟内存和物理内存都被释放了。
- 4.当最高地址空间的空闲内存超过128K时，执行内存紧缩操作（trim）。

既然堆内内存brk和sbrk不能直接释放，为什么不全部使用 mmap 来分配，munmap直接释放呢？

- 1.mmap 申请的内存被 munmap 后，重新申请会产生更多的缺页中断。缺页中断是内核行为，会导致内核态 CPU 消耗较大。
- 2.如果使用 mmap 分配小内存，会导致地址空间的分片更多，内核的管理负担更大。

fork和exec区别

- 1.调用fork()时，产生的子进程是复制的进程表（PCB表），但是PID（进程号：区别进程的根本标志）不同。
- 2.exec()则是改变进程的进程地址空间（虚拟的），它的PID保持原来的PID，即exec()前后是同一个进程（PID未改变），可以说，exec()实现了让一个进程执行多个可执行文件。

linux的32位和64位有什么区别

- 1.32位系统的虚拟地址空间为4GB(2^32bytes),其中0x08048000~0xbfffffff 是用户空间，0xc0000000~0xffffffff 是内核空间。
- 2.64位系统的虚拟地址空间位256TB(2^48bytes)，64 位 Linux 一般使用 48 位来表示虚拟地址空间，40 位表示物理地址。
    - 0x0000000000000000~0x00007fffffffffff表示用户空间。
    - 0xFFFF800000000000~ 0xFFFFFFFFFFFFFFFF表示内核空间。

alloc,calloc,malloc,free,realloc,sbrk

- 1.alloc。alloc是向栈申请内存,因此无需释放。
- 2.calloc。calloc则将初始化的malloc分配的内存,设置为0。
- 3.malloc。malloc分配的内存是位于堆中的,并且没有初始化内存的内容,因此基本上malloc之后,调用函数memset来初始化这部分的内存空间。
- 4.realloc。而realloc则对malloc申请的内存进行大小的调整。
- 5.sbrk。brk是增加数据段的大小。

C++函数参数入栈顺序

- 从右向左
    - 为了支持可变参数，所以函数参数入栈顺序是从右向左。
        - 栈底是高址，栈顶是低址，从高地址向低地址增长。
        - 最右边的参数在栈底，最左边的参数在栈顶。除非知道参数个数，否则是无法通过栈指针的相对位移求得最右边的参数。右边的参数个数的不确定，可以支持可变参数。

fork一个子进程，和父进程共享什么

- fork子进程会拷贝父进程的所有资源，变量。两者的虚拟空间不同，每个进程都拥有自己的独立地址空间。
- 共享的空间只有代码段。
- 写时复制
    - 只有进程空间的各段的内容要发生变化时，才会将父进程的内容复制一份给子进程。否则子进程就指向父进程的物理空间。

静态链接和动态链接

- 1.静态链接是在形成可执行程序前。
    - 每个源文件都是独立编译的，形成独立的.o文件，静态链接将这些.o文件进行链接，形成可执行文件。
    - 链接器在链接静态链接库的时候是以目标文件为单位的。(导入整个文件的代码，很可能很多没用的函数都被一起链接进了输出结果中)
        - 浪费空间。因为每个可执行程序中对所有需要的目标文件都要有一份副本，所以同一个目标文件都在内存存在多个副本。
        - 更新困难。每当库函数的代码修改了，这个时候就需要重新进行编译链接形成可执行程序。
        - 静态链接在执行程序中已经具备了所有执行程序所需要的任何东西，在执行的时候运行速度快。
    - 静态重定位(程序和数据装入内存时需对目标程序中的地址进行修改。这种把逻辑地址转变为内存的物理地址的过程)
        - 1.静态重定位是在目标程序装入内存时，由装入程序对目标程序中的指令和数据的地址进行修改，即把程序的逻辑地址都改成实际的地址。对每个程序来说，这种地址变换只是在装入时一次完成，在程序运行期间不再进行重定位。
            - 无需增加硬件地址转换机构，便于实现程序的静态连接。
            - 程序的存储空间只能是连续的一片区域，而且在重定位之后就不能再移动。这不利于内存空间的有效使用。
            - 各个用户进程很难共享内存中的同一程序的副本。
- 2.动态链接的进行则是在程序执行时。
    - 1.动态链接的基本思想是把程序按照模块拆分成各个相对独立部分，在程序运行时才将它们链接在一起形成一个完整的程序，而不是像静态链接一样把所有程序模块都链接成一个单独的可执行文件。
        - 1.装载时动态链接。
            - 提前就已经知道程序要调用哪些模块，在编译时就在相应地方存放链接这些模块的信息，运行时对应去调用。
        - 2.运行时动态链接。
            - 提前无法预知，完全等运行时决定调用哪个函数。
    - 节约空间。同一个目标文件在内存中只有一个副本。
    - 更新方便。更新时只需要替换原来的目标文件，而无需将所有的程序再重新链接一遍。
    - 动态链接因为在执行期间需要动态链接，所以运行速度比较慢。
    - 动态重定位
        - 1.动态重定位是在程序执行期间每次访问内存之前进行重定位。
        - 2.使用一个重定位寄存器，放有当前正在执行的程序在内存空间中的起始地址，而地址空间中的代码在装入过程中不发生变化。
        - 程序占用的内存空间动态可变，不必连续存放在一处。
        - 比较容易实现几个进程对同一程序副本的共享使用。
        - 需要附加的硬件支持，增加了机器成本，而且实现存储管理的软件算法比较复杂。

分区的存储保护

- 存储保护是为了防止一个作业有意或无意地破坏操作系统或其他作业。
    - 1.上、下界寄存器方法。

        - 采用上、下界寄存器分别存放作业的结束地址和开始地址。
    - 2.基址、限长寄存器方法。
        - 采用基址和限长寄存器分别存放作业的起始地址及作业的地址空间长度。
    - 存储保护键方法是给每个存储块分配一个单独的保护键，它相当于一把锁。进入系统的每个作业也被赋予一个保护键，它相当于一把钥匙。当作业运行时，检查钥匙和锁是否一致，如果二者不匹配，则系统发出保护性中断信号，并停止作业的运行。

数据库的隔离级别

- 1.Read uncommitted(读取未提交内容)
    - 所有事务都可以看到其他未提交事务的执行结果。
    - 会出现脏读，不可重复读，幻读。
- 2.Read committed(读取提交内容)
    - 一个事务只能看见已经提交事务所做的改变。
    - 会出现不可重复读，幻读。
- 3.Repeatable read(可重复读)
    - mysql默认事务隔离级别。
    - 确保同一事务的多个实例在并发读取数据时，会看到同样的数据行。
    - 可能出现幻读。
- 4.Serializable(可串行化)
    - 通过强制事务排序，使之不可能相互冲突，从而解决幻读问题。
        - 它是在每个读的数据行上加上共享锁。
    - 不会出现脏读，不可重复读和幻读。

漏桶算法和令牌桶算法(高并发系统限流)

- 1.漏桶算法
    - 请求先进入到漏桶里，漏桶以一定的速度出水，当水请求过大会直接溢出。漏桶算能强行限制数据的传输速率。
- 2.令牌桶算法
    - 大小固定的令牌桶可自行以恒定的速率源源不断地产生令牌。
    - 每个请求消耗一个令牌。
    - 如果令牌不被消耗，或者被消耗的速度小于产生的速度，令牌就会不断地增多，直到把桶填满。后面再产生的令牌就会从桶中溢出。最后桶中可以保存的最大令牌数永远不会超过桶的大小。
- 区别
    - “漏桶算法”能够强行限制数据的传输速率。
    - 令牌桶算法”在能够限制数据的平均传输速率外，还允许某种程度的突发传输。
        - 只要令牌桶中存在令牌，那么就允许突发地传输数据直到达到用户配置的门限，所以它适合于具有突发特性的流量。

数据库连接池的好处

- 频繁的建立、关闭连接，会极大的减低系统的性能，因为对于连接的使用成了系统性能的瓶颈。
- 数据库连接池的基本原理是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获取和返回方法。
- 数据库连接池技术带来的优势：
    - 资源重用。

        - 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。
    - 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中备用。此时连接的初始化工作均已完成。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。
    - 新的资源分配手段。
    - 统一的连接管理，避免数据库连接泄漏。

nosql vs sql

- 1.SQL的表 vs NoSQL的文档
    - 1.SQL表创建的是严格的数据模板。
    - 2.NoSQL数据库存储类JSON的键值对文档。
- 2.SQL架构 vs NoSQL去架构
    - 1.在一个SQL数据库中，在你确定表和字段类型这些架构之前是无法添加数据的。
    - 2.在NoSQL数据库中，数据可以被非常灵活的添加。并不需要事先进行字段设计和表的设计。
- 3.SQL 中心化 vs NoSQL去中心化
- 4.SQL关系型的JOIN vs NoSQL
    - 1.SQL查询提供了强劲的JOIN语法。我们可以使用一条SQL语句在多个数据表中获取关系数据库。
    - 2.NoSQL没有相对应的JOIN，必须手动的通过程序逻辑来把两者联系起来。
- 5.SQL vs NoSQL 数据完整性
    - 1.大部分的SQL数据库允许你通过外键限制的方式来强制的保证数据完整性。
    - 2.在NoSQL中没有类似的数据完整性保证。
- 6.SQL vs NoSQL 事务
    - 1.在SQL数据库中，两条或多条更新语句能够在一个事务中被同时执行。
    - 2.在NoSQL数据库中，对单个文档的修改是原子的。但是对于多条文档的更新而言却没有事务。
- 7.SQL vs NoSQL CRUD 语法
    - 1.SQL是轻量级的解释性语言。语法强大，并且已经成为了国际标准
    - 2.NoSQL数据库使用带json参数的类javascript语言一样的查询。基本的操作比较简单，但是对于更复杂的查询来说，嵌套的JSON会非常的繁复。
- 8.SQL vs NoSQL 性能表现
    - 1.NoSQL比SQL块
        - NoSQL更简单的去中心化存储允许你在单词请求中获取一个条目的所有信息。因此并不需要相关的JOIN或复杂的SQL查询。

epoll原理

- 操作系统如何知道网络数据对应于哪个socket？
    - 一个socket对应着一个端口号，而网络数据包中包含了ip和端口的信息，内核可以通过端口号找到对应的socket。了提高处理速度，操作系统会维护端口号到socket的索引结构，以快速读取。
- select
    - 每次调用select都需要将进程加入到所有监视socket的等待队列，每次唤醒都需要从每个队列中移除。
        - 1.每次都要将整个fds列表传递给内核，内核遍历一次socket。
        - 2.进程被唤醒后，程序并不知道哪些socket收到数据，还需要遍历一次。
- epoll的设计思路
    - 1.功能分离

        - select低效的原因之一是将“维护等待队列”和“阻塞进程”两个步骤合二为一。
        - epoll将这两个操作分开，先用epoll_ctl维护等待队列，再调用epoll_wait阻塞进程。
    - 2.就绪列表
        - select低效的另一个原因在于程序不知道哪些socket收到数据，只能一个个遍历。
        - epoll中内核维护一个“就绪列表”，引用收到数据的socket，就能避免遍历。
- epoll的原理和流程
    - 1.创建epoll对象
        - 当某个进程调用epoll_create方法时，内核会创建一个eventpoll对象。
        - 创建一个代表该epoll的eventpoll对象是必须的，因为内核要维护“就绪列表”等数据，“就绪列表”可以作为eventpoll的成员。
    - 2.维护监视列表
        - 创建epoll对象后，可以用epoll_ctl添加或删除所要监听的socket。
        - 当socket收到数据后，中断程序会操作eventpoll对象，而不是直接操作进程。
    - 3.接收数据
        - 当socket收到数据后，中断程序会给eventpoll的“就绪列表”添加socket引用。
            - socket的数据接收并不直接影响进程，而是通过改变eventpoll的就绪列表来改变进程状态。
        - 当程序执行到epoll_wait时，如果rdlist已经引用了socket，那么epoll_wait直接返回，如果rdlist为空，阻塞进程。
    - 4.阻塞和唤醒进程
        - 进程运行到了epoll_wait语句时，内核会将进程放入eventpoll的等待队列中，阻塞进程。
        - 当socket接收到数据，中断程序一方面修改rdlist，另一方面唤醒eventpoll等待队列中的进程，进程再次进入运行状态。
- epoll的实现细节
    - eventpoll包含了lock、mtx、wq（等待队列）、rdlist等成员。
    - 就绪列表的数据结构
        - epoll使用双向链表来实现就绪队列。
            - 程序可能随时调用epoll_ctl添加监视socket，也可能随时删除。
    - epoll使用了红黑树作为索引结构。
        - 因为epoll将“维护监视队列”和“进程阻塞”分离，也意味着需要有个数据结构来保存监视的socket。需要方便的添加和删除，还要便于搜索，以避免重复添加。
- poll机制
    - poll的机制与select类似，管理多个描述符也是进行轮询，但是poll没有最大文件描述符数量的限制。
    - poll改变了文件描述符集合的描述方式，使用了pollfd结构而不是select的fd_set结构，使得poll支持的文件描述符集合限制远大于select的1024。

伙伴算法

- 类似STL中二级空间分配器
- 伙伴算法（Buddy system）把所有的空闲页框分为11个块链表，每块链表中分布包含特定的连续页框地址空间。
    - 第0个块链表包含大小为2^0个连续的页框，第10个块链表包含大小为2^10个连续页框。
    - 伙伴算法每次只能分配2的幂次页的空间。
- 伙伴位图：用一位描述伙伴块的状态位码，称之为伙伴位码。
- 申请

    - 申请k个页内存空间，系统首先从free_area[log(k)]查看是否有空闲块，有就分配；没有的话，就从他的上一级free_area[log(k)+1]中查找空闲块，有就分配一半给程序，另一半加入到free_area[log(k)]中；如果没有，就继续向上申请，如果free_area[10]都没有空闲空间，则报错。
- 释放
    - 现在对应的free_area链表中查找是否有伙伴存在，如果没有伙伴，则直将释放的内存插入链表头中，如果存在伙伴，则将伙伴从free_area摘下，与释放的内存合并成一个大块，继续向上查找，直至不能合并或者已经合并至最大块2^10为止。

slab机制

- 伙伴系统算法采用页作为基本内存区，这适合于大块内存的请求；对于小内存区的申请，比如说几十或几百个字节，使用slab机制。

- slab分配机制

    - slab分配器是基于对象(内核中的数据结构)进行管理的。相同类型的对象归为一类。
        - 申请一个对象时，slab分配器从slab列表中分配一个这样大小一个单元出去。
        - 释放一个对象时，将其重新保存到slab列表中，不会直接返回给伙伴系统。
        - slab分配器并不丢弃已经分配的对象，而是释放并把它们保存在内存中。
        - slab分配对象时，会使用最近释放的对象的内存块，因此其驻留在cpu高速缓存中的概率会大大提高。

- 内核中slab的主要数据结构

    - slab是slab分配器的最小单位，由一个或多个连续的物理页组成。
    - 首先寻找slabs_partial，如果非空，则返回一个指向已分配但是未使用的slab的指针。如果该slab满了，则从slabs_partial移除，并插入到slabs_full中。如果slabs_partial为空，则到slabs_empty寻找。如果slabs_empty也为空，则新创建一个slab。
        - slabs_full(完全分配的slab)
        - slabs_partial(部分分配的slab)
        - slabs_empty(空slab,或者没有对象被分配)

- slab着色

    - slab中倾向于把大小相同的对象放在同一个硬件cache line中，方便对齐和寻址。

协程

- Coroutine（协程）是一种用户态的轻量级线程。
    - C++通过Boost.Coroutine实现对协程的支持。
    - Python通过yield关键字实现协程，Python3.5开始使用async def对原生协程的支持。
- 1.协程(Coroutine)编译器级的，进程(Process)和线程(Thread)操作系统级的。
- 2.进程(Process)和线程(Thread)是os通过调度算法。
- 协程更加轻量，创建成本更小，降低了内存消耗。
    - 协程本身可以做在用户态，每个协程的体积比线程要小得多，因此一个进程可以容纳数量相当可观的协程。
- 协程优点
    - 协作式的用户态调度器，减少了 CPU 上下文切换的开销，提高了 CPU 缓存命中率。
        - 协作式调度相比抢占式调度的优势在于上下文切换开销更少、更容易把缓存跑热。

        - 进程 / 线程的切换需要在内核完成，而协程不需要，协程通过用户态栈实现，更加轻量，速度更快。
    - 减少同步加锁，整体上提高了性能。
        - 协程方案基于事件循环方案，减少了同步加锁的频率。
        - 若存在竞争，并不能保证临界区，因此该上锁的地方仍需要加上协程锁。
    - 可以按照同步思维写异步代码，即用同步的逻辑，写由协程调度的回调。
        - 协程的确可以减少 callback 的使用但是不能完全替换 callback。
        - 基于事件驱动的编程里面反而不能发挥协程的作用而用 callback 更适合。
- 协程缺点
    - 在协程执行中不能有阻塞操作，否则整个线程被阻塞。
        - 协程是语言级别的，线程，进程属于操作系统级别 。
    - 需要特别关注全局变量、对象引用的使用。
    - 协程可以处理 IO 密集型程序的效率问题，但是处理 CPU 密集型不是它的长处。

malloc底层原理

- Linux维护一个break指针，这个指针指向堆空间的某个地址。从堆起始地址到break之间的地址空间为映射好的，可以供进程访问；而从break往上，是未映射的地址空间，如果访问这段空间则程序会报错。
    - malloc就是从break上进行内存分配的。
- malloc实质
    - malloc 函数的实质是它有一个将可用的内存块连接为一个长长的列表的所谓空闲链表。
    - 调用 malloc（）函数时，它沿着连接表寻找一个大到足以满足用户请求所需要的内存块。 然后，将该内存块一分为二。将分配给用户的那块内存存储区域传给用户，并将剩下的那块（如果有的话）返回到连接表上。
        - 一块的大小与用户申请的大小相等，另一块的大小就是剩下来的字节。
    - 调用 free 函数时，它将用户释放的内存块连接到空闲链表上。
    - 空闲链会被切成很多的小内存片段，如果这时用户申请一个大的内存片段， 那么空闲链表上可能没有可以满足用户要求的片段了。malloc（）函数请求延时，并开始在空闲链表上检查各内存片段，对它们进行内存整理，将相邻的小空闲块合并成较大的内存块。
- malloc流程
    - 1.malloc分配内存前的初始化。
        - malloc_init 是初始化内存分配程序的函数。
            - 1.将分配程序标识为已经初始化。
            - 2.找到操作系统中最后一个有效的内存地址(中断点或者当前中断点)。
                - sbrk 函数根据参数中给出的字节数移动当前系统中断点，然后返回新的系统中断点。sbrk(0)返回当前中断点(break指向的地址)。
                    - brk将break指针直接设置为某个地址，而sbrk将break从当前位置移动increment所指定的增量。
            - 3.建立起指向需要管理的内存的指针。
    - 2.内存块的获取
        - 所要申请的内存是由多个内存块构成的链表。
        - 内存块数据结构
            - meta区。记录数据块信息（数据区大小、空闲标志位、指针等等)。

        - 数据区。真实分配的内存区域，并且数据区的第一个字节地址即为malloc返回的地址。

```
typedef struct s_block *t_block;
    struct s_block {
    size_t size; /* 数据区大小 */
    t_block next; /* 指向下个块的指针 */
    int free; /* 是否是空闲块 */
    int padding; /* 填充4字节，保证meta块长度为8的倍数 */
    char data[1] /* 这是一个虚拟字段，表示数据块的第一个字节，长度不应计入meta */
};
```

        - 寻找合适的block
            - First fit：从头开始，使用第一个数据区大小大于要求size的块所谓此次分配的块。(较好的运行效率)
            - Best fit：从头开始，遍历所有块，使用数据区大小大于size且差值最小的块作为此次分配的块。(较搞定的内存使用率)。
            - 在遍历时会更新一个叫last的指针，这个指针始终指向当前遍历的block。为找不到合适的block而开辟新block使用的。
        - 如果现有block都不能满足size的要求，则需要在链表最后开辟一个新的block。
    - 3.内存分配，下为内存分配代码。

url解析全过程

- 1.用户输入网址，浏览器发起DNS查询请求。
    - 域名映射到ip地址。
- 2.建立TCP连接。
    - 三次握手
- 3.浏览器发送HTTP请求。
- 4.服务器发送响应数据给客户端。
- 5.浏览器解析HTTP response。

DNS解析过程

- 1.浏览器向本地服务器进行递归查询。
- 2.本地域名服务器向根域名服务器进行迭代查询。
    - 1.根域名服务器告诉本地服务器下一次应查询的顶级域名服务器的IP地址。
    - 2.本地域名服务器向顶级域名服务器进行查询。顶级域名服务器返回权限服务器的IP地址。
    - 3.本地域名服务器向权限域名服务器进行查询。权限服务器返回所查询主机的IP地址。
- 3.本地域名服务器向浏览器返回查询结果。
- 域名服务器中泛使用了高速缓存，用来存放最近查询过的域名以及从何处获得域名映射信息的记录。
- DNS在进行区域传输的时候使用TCP，普通的查询使用UDP。
    - DNS的规范规定了2种类型的DNS服务器，一个叫主DNS服务器，一个叫辅助DNS服务器。

        - 主DNS服务器从自己本机的数据文件中读取该区的DNS数据信息。
        - 辅助DNS服务器则从区的主DNS服务器中读取该区的DNS数据信息。
    - 当一个辅助DNS服务器启动时，它需要与主DNS服务器通信，并加载数据信息，这就叫做区传送（zone transfer）。

DNS为什么在进行区域传输时使用TCP

- 因为DNS在需要跨越广域网或互联网，分组丢失率和往返时间的不确定性要更大些，这对于DNS客户端来说是个考验，好的重传和超时检测就显得更重要了。
- 因为数据同步传送的数据量比一个请求和应答的数据量要多得多。

DNS为什么域名解析时使用UDP协议

- 客户端向DNS服务器查询域名，一般返回的内容都不超过512字节，用UDP传输即可。

递归查询 vs 迭代查询

- 1.递归查询
    - 服务器帮你查。
- 2.迭代查询
    - 服务器告诉下一步应查询的地址，本地域名服务器自己去查。

重定义 vs 重写

- 1.重定义(隐藏)：基类函数无virtual修饰。
- 2.重写(覆盖)：基类函数有virtual修饰。

tcp会自动断开连接吗？

- 不会。
- 不管是客户端的还是服务器端的应用程序都没有应用程序级（application-level）的定时器来探测连接的不活动状态（inactivity），从而引起任何一个应用程序的终止。
- TCP的保活定时器能够保证TCP连接一直保持，但是TCP的保活定时器不是每个TCP/IP协议栈就实现了，因为RFC并不要求TCP保活定时器一定要实现。

get与post请求区别

- 1.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。
- 2.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。
- 3.get是不安全的，因为URL是可见的,post较get安全性较高,URL是不可见了。
- 4.get方式只能支持ASCII字符，向服务器传的中文字符可能会乱码；post支持标准字符集，可以正确传递中文字符。
- 5.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；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。

程序退出时还会有内存泄露吗？

- 不会。
    - 程序退出了，所有的资源都会释放的，被系统回收重新利用的。所以不会有内存泄漏这一说。

哈希表的桶个数为什么是质数，合数有何不妥？

- 质数比合数更容易避免冲撞，也就是说使用质数时，哈希效果更好，原始数据经哈希后分布更均匀。

进程分配内存的两种方式，分别由两种系统调用完成

- 1.brk。brk是将数据段(.data)的最高地址指针_edata往高地址推。
- 2.mmap。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。

缓存

- 1.缓存雪崩。
    - 由于原有缓存失效，新缓存未到期间，所有原本应该访问缓存的请求都去查询数据库了，而对数据库CPU和内存造成巨大压力。
    - 解决方法
        - 用加锁或者队列的方式保证来保证不会有大量的线程对数据库一次性进行读写，从而避免失效时大量的并发请求落到底层存储系统上。
        - 将缓存失效时间分散开，在原有的失效时间基础上增加一个随机值，，这样每一个缓存的过期时间的重复率就会降低，就很难引发集体失效的事件。
- 2.缓存穿透。
    - 缓存穿透是指用户查询数据，在数据库没有，自然在缓存中也不会有。
    - 进行了两次无用查询。
    - 解决方法
        - 布隆过滤器。
            - 将所有可能存在的数据哈希到一个足够大的bitmap中，一个一定不存在的数据会被这个bitmap拦截掉，从而避免了对底层存储系统的查询压力。
- 3.缓存预热。
    - 缓存预热就是系统上线后，将相关的缓存数据直接加载到缓存系统。这样就可以避免在用户请求的时候，先查询数据库，然后再将数据缓存的问题。用户直接查询事先被预热的缓存数据。
    - 解决方法
        - 1.直接写个缓存刷新页面，上线时手工操作下。
        - 2.数据量不大，可以在项目启动的时候自动进行加载。
        - 3.定时刷新缓存。
- 4.缓存更新。
    - 定时去清理过期的缓存。
    - 当有用户请求过来时，再判断这个请求所用到的缓存是否过期，过期的话就去底层系统得到新数据并更新缓存。
- 5.缓存降级。
    - 当访问量剧增、服务出现问题（如响应时间慢或不响应）或非核心服务影响到核心流程的性能时，仍然需要保证服务还是可用的，即使是有损服务。系统可以根据一些关键数据进行自动降级，也可以配置开关实现人工降级。

    - 降级的最终目的是保证核心服务可用，即使是有损的。而且有些服务是无法降级的（如加入购物车、结算）。
    - 方法
        - 一般：比如有些服务偶尔因为网络抖动或者服务正在上线而超时，可以自动降级。
        - 警告：有些服务在一段时间内成功率有波动（如在95~100%之间），可以自动降级或人工降级，并发送告警。
        - 错误：比如可用率低于90%，或者数据库连接池被打爆了，或者访问量突然猛增到系统能承受的最大阀值，此时可以根据情况自动降级或者人工降级。
        - 严重错误：比如因为特殊原因数据错误了，此时需要紧急人工降级。

函数名后加const

- 表明该函数不会修改类对象。

c++对象之间的通信方式

- 1.全局单例对象作为交互对象的中介。
    - 全局单例对象记录交互对象的地址。
- 2.虚接口代理类。
- 3.基类方法函数指针
    - 在对象2中定义指向类1方法的指针。

迭代器失效

- 1.序列式容器
    - (vector，deque)删除当前的迭代器会使后面的所有元素的迭代器都失效。
        - vector，deque使用连续分配空间，删除元素会导致后续元素向前移动，会导致迭代器失效，之前获取的迭代器根据原有的信息就访问不到正确的数据。
    - vector插入元素时会导致后面所有元素的迭代器都失效。
        - vector插入元素导致重新分配空间，则所有的迭代器都会失效。
    - deque在头尾插入元素不会导致迭代器失效，在中间插入元素到导致后面所有元素的迭代器失效。
        - 采用iter=erase(iter)删除迭代器。返回一个指向下一个元素的迭代器。
    - list,slist删除(插入)当前元素的迭代器仅使得当前迭代器失效。
        - erase(iter)/erase(iter++)都可以删除迭代器。
    - 2.关联式容器
        - 底层为红黑树
            - 删除当前元素的迭代器仅会造成当前的迭代器失效。
                - 采用erase(iter++)的方式删除迭代器。
            - 插入元素不会造成任何迭代器失效。
        - 底层为哈希表
            - 删除元素时?不会造成任何迭代器失效。
            - 插入元素不会造成任何迭代器失效。

c++函数调用约定

- 函数调用约定使对函数调用的一种约束和规范，描述了函数参数使怎么传递和由谁清除堆栈。

    - 1.函数的压栈顺序。

    - 2.由调用者还是被调用者把参数弹出栈。
    - 3.产生函数修饰名的方法。

- 声明和定义处调用约定必须要相同。不能只在声明处有调用约定，而定义处没有或与声明不同。

- 函数调用过程

    - 1.先将调用者的堆栈的基址入栈，以保存之前任务的信息。
    - 2.再将调用者的栈顶指针的值赋给基址，作为新的基址。
    - 3.新的基址(被调用者的栈底)上开辟相应空间按用作被调用者的栈空间。
    - 4.被调用函数返回后，从当前栈的基址即恢复调用者的栈顶，使栈顶恢复被调用函数被调用前的位置。然后调用者从栈顶弹出之前的基址值。

- __cdecl(C Declaration)

    - 是 C 和 C++ 默认的函数调用约定。
        - 1.参数按从右至左的顺序压入栈。
        - 2.由调用者把参数弹出栈。
            - 对于传送参数的内存栈是由调用者来维护的。
        - 3.编译器对函数生成修饰名时，在函数名前加上一个下划线前缀(func(...)-->_func(...))。

- __stdcall(Standrad Call)

    - 是C++的标准调用方式。
        - 1.参数按从右至左的顺序压入栈。
        - 2.由被调用者把参数弹出栈。
            - 函数自己在退出时清空堆栈。
        - 3.编译器对函数生成修饰名时，在函数名前加上一个下划线前缀和@符号以及其参数的字节数。(func(int a)-->_@4func(int a))。

- __fastcall

    - 主要特点是快，使用寄存器来传送参数。
        - 1.前两个参数使用ECX和EXD寄存器传送(顺序从左向右)，剩下的参数仍自右向左压栈传送。
        - 2.由被调用者把参数弹出栈。
            - 函数自己在退出时清空堆栈。
        - 3.编译器对函数生成修饰名时，在函数名前加上@，函数名后加上@和其参数的字节数。(func(int a)-->@func@4(int a))。

- __thiscall

    - 是C++类成员函数缺省的调用约定，但它没有显示的声明形式。
        - 因为在C++类中，成员函数调用还有一个this指针参数，因此必须特殊处理。
        - 1.参数按从右至左的顺序压入栈。
        - 2.this指针入栈
            - 如果参数个数确定，this指针通过ecx传递给被调用者。
            - 如果参数个数不确定，this指针在所有参数压栈后被压入栈。
        - 3.栈恢复
            - 对于参数个数确定，被调用者函数退出时自己清理堆栈。
            - 对参数个数不定，由调用者清理栈。

C++中多重继承的二义性及解决办法

- 多重继承的二义性。
  - 多个父类拥有同样的函数，导致子类不知道该调用哪个父类函数。
- 解决方法
  - 类名限定。调用时指定调用的是哪个类的函数。
  - 同名覆盖。在子类中声明一个同名函数，该函数内部调用具体父类的函数。
  - 虚拟继承。解决菱形继承中的基类的二义性。

session,cookie和token

- session
  - session保存在服务器上。
- cookie
  - cookie保存在客户端浏览器上。
  - cookie由服务器生成，发送给浏览器。
- token
  - token(令牌)，是用户身份的验证方式。
  - token状态时存储在客户端上。

gcc vs g+++

- 1.对于.c文件，gcc认为是c语言程序，g++认为是c++程序。
- 2.对于.cpp文件，gcc和g++都认为是c++程序。
- 3.g++会自动链接STL标准库，gcc不会自动链接STL标准库。

编译器常用优化方法

- 1.常量传播
  - 将计算出结果的变量直接替换为常量。
- 2.常量折叠
  - 多个变量进行计算时，而且能够直接计算出结果，那么变量将有常量直接替换。
- 3.复写传播
  - 两个相同的变量可以用一个代替。
- 4.公共子表式消除
  - 如果一个表达式已经计算过了，并且从先前的计算到现在变量都没有发生变化，那么该表达式的此次出现就成为了公共子表达式。
- 5.无用代码消除
  - 永远不能被执行到的代码或者没有任何意义的代码会被清除掉。
- 6.数组范围检查消除
- 7.逃逸分析
- 8.返回值优化(RVO:Return Value Optimization)
  - 消除函数返回时创建的临时对象，达到减少拷贝构造函数调用的操作。
  - 将函数改成成传引用，在函数体内对该引用对象进行赋值，return空。

linux软链接和硬链接

- 1.软连接

    - 新建立一个文件专门指向其他文件。
- 2.硬链接
    - 在指向文件的node link count自增1，实际上是为文件建立一个别名，inode号是一样的，软连接建立一个指向文件的指针。

---

c++操作

- int 2 string

```
string num = to_string(a);
```

- string 2 int

```
int a =stoi(str);
```

- 循环输入

```
while(cin>>tmp){
  .......
  if(cin.get()=='\n'){
    break;
  }
}
```
