目录

华南理工大学：欧建永　　2020/03/28

第一章． C++语法

一． 虚函数

1. C++的多态表现，动态多态。使用 Virtual 关键字，子类继承父类，并在子类中实现，当使用父类指针调用虚函数时，根据运行时的实际对象动态调用。
2. 有子类的话，虚构函数必须为虚函数，否则析构子类时有可能不会调用到。
3. 编译器根据虚函数表找到恰当的虚函数。对于一个父类的对象指针（或引用）类型变量，如果给他赋值父类对象的指针，那么他就调用父类中的函数，如果给他赋子类对象的指针，他就调用子类中的函数。函数执行之前通过查虚函数表来查找调用的函数。
4. 虚函数表是针对类的，一个类的所有对象的虚函数表都一样。
5. 虚函数一般会在末尾加上 override 关键字，因为如果该函数没有覆盖已有的虚函数，编译器会报错。
6. 虚函数后加上 final 关键字，则之后任何想覆盖该函数的行为会报错。
7. 纯虚函数的是在虚函数末尾加上“=0”，包含纯虚函数的类是抽象类，不能生成实例，只能被继承。纯虚函数只定义不实现。
8. 构造函数不能是虚函数，每个对象的虚函数表指针在构造函数中初始化，所以构造函数未执行，虚函数表不起作用。
9. 构造/析构函数中调用虚函数和调用普通函数一样。因为虚函数表可能未构造或者已经被析构。
10. 虚拟继承是用来解决多重继承中的三角继承关系的，避免派生类中产生多余的实例。
11. **静态多态：**函数重载；**动态多态**：虚函数机制。1. 多态有利于提升程序的扩

充性。2. 提升程序的可维护性。

12. **虚函数表**，只属于类，在编译期间产生，存在于只读数据段。**虚函数表指针**，属于对象，在编译期间产生，存放在堆或者栈上。

二． 构造函数/析构函数

1. 构造时：父类构造函数->子类构造函数
2. 析构时：子类析构函数->父类析构函数
3. 构造函数中的初始化列表初始化成员只和类中定义的变量的顺序相关，与初始化列表中变量排序的顺序无关。
4. 类中的 const 成员，reference 成员变量只能用初始化列表初始化,或者赋值一个默认参数。
5. 构造函数：

1> 当类中定义了其他构造函数之后，将不存在**默认构造函数**，需要自己合成，即再无参构造函数末尾加上“=default”

2> **使用 explicite** 修饰构造函数，使其无法进行隐式类型转换。

3> **拷贝构造函数**的写法型如：Foo(const Foo &)，一般发生在使用已有的对象初始化一个正在创建的对象：包含使用等号，传参，返回值<非引用>等

4> **移动构造函数**的写法型如：Foo(const Foo &&):右值引用，可以避免不必要的拷贝赋值，提升速度。

5> 如果我们想禁止拷贝，例如 iostream.我们应该将它们定义为删除的=delete

a) Matrix(const Matrix<_T>&) = delete;

6> 如果**要阻止类的继承**，则将该类的构造函数或者析构函数设置成 private

7> 浅拷贝出错怎么解决？： 1. 定义自己的拷贝构造函数；2. 禁止拷贝（将拷贝

构造函数与拷贝运算符设置成删除函数)。

8> 什么时候需要定义自己的拷贝构造函数？：1.当要处理类的静态变量的时候，例如共享指针中的引用计数。2. 需要进行深拷贝的时候。

三． C++的内存管理方式

1. 除了栈以外，堆，只读数据区，全局变量地址增长方式都是从低到高。

2. C++的内存划分:

栈区：编译器自动分配释放，存放函数参数值，局部变量值。

堆区：由程序员申请，程序员释放。程序员不释放，程序结束由 OS 回收。

全局区：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。例如 static 类型。

文字常量区：存储字符串常量等等，程序结束后系统释放。

程序代码区：存放函数体的二进制代码。

3. 一个类没有定义属性和函数（一个空类），其 sizeof 是多大?

一个空类的大小是 1 字节，这样保证每个类由独一无二的地址。

4. 内存对齐与虚函数指针:

一个类只要继承了一个带虚函数的类，就会产生一个虚函数指针。其次编译器为了加速数据存取一般会将类的数据大小调整至 4 字节的倍数。内存对齐策略有利于加速数据读取!

四． 引用，指针与数组

1. 引用必须初始化，引用不能为空，引用在赋值之后不能修改。

2. 引用是对象的一个别名，指针指向某个变量通过地址操作变量。

3. 数组要么在静态存储区被创建（全局数组），要么在栈上被创建。
4. 使用 sizeof（数组名）可以计算出数组的容量，使用 sizeof（指针）得到的是指针的内存大小。C++没法知道指针所值对象的内存大小，只能在申请的时候记住它。当数组作为函数参数进行传递时，退化为同类型的指针。
5. 对一个引用执行 sizeof 运算，得到的是变量的内存大小。
6. 常量引用
   a) 指向常量对象时，一定要使用“常量引用”，而不能是一般的引用。
   b) “常量引用”可以指向一个非常量对象，但不允许用过该引用修改非常量对象的值。
   c) 在函数参数中，使用常量引用非常重要。因为函数有可能接受临时对象，而且同时需要禁止对所引用对象的一切修改。

五． 常用关键字

1. **“Extern C”** 告诉编译器以 C 的方式编译，因为 C 语言没有重载，函数在底层的签名就是函数名，而 C++编译某个函数时，会包含参数类型作为函数名。
2. **Extern** 作用声明外部，因为现代编译器按文件的方式编译，因此全局变量的作用域仅在当前文件内（未 extern 声明的前提下）。
3. **struct 和 class** 的区别：主要是访问默认权限不同，struct 默认是 public，class 默认是 private。
4. **static 关键字**：

（1） 局部变量加 static:局部变量变成一个静态的局部变量，由原来的栈变为静态存储区及其生命周期直到程序结束，但是作用域未改变。

（2） 全局变量加 static:全局变量被定义成一个全局静态变量，存储位置未改变，

全局变量初值若未初始化则默认为 0，静态全局变量仅在当前文件可以被访问，其他文件定义同样名字的变量也不会冲突。

（3） 成员变量中使用 static:使该变量称为类的全局变量，为所有对象包括派生类的对象共享，所有的对象仅有一份实例。

（4） 成员函数加 static:这个类只存一份函数，所有对象共享该函数，不含 this 指针。不需要通过对象便可直接访问，静态成员函数只能访问类的 static 成员，不能访问非 static 成员。

（5） **普通函数**前加 static:静态函数的定义和声明默认都是 extern 的，但只在当前文件中可见，不可以被其他文件引用，且不会与其他文件中的同名函数冲突。不要在 cpp 内部声明非 static 的全局函数。

5. const 关键字：

(1) 修饰变量：限定变量不可被改变。

(2) 修饰成员函数：函数不可以修改变量。

(3) 修饰指针：具体如下所示<顶层 const 与底层 const>

1> const char *p | char const *p // 指向的内容不可以改变

2> char *const p // p 是常指针，指针指向不可改变，但是值的内容可以改变

3> const char * const p // 两者兼具

不可以同时用 const 与 static 修饰成员变量：C++编译器在实现 const 的成员函数的时候为了确保该函数不能修改类的实例的状态，会在函数中添加一个隐式的参数 const this*。但当一个成员为 static 的时候，该函数是没有 this 指针的。也就是说此时 const 的用法和 static 是冲突的。

(4) const 修饰类对象，定义常量对象:常量对象只能调用常量函数，别的成员函数都不能调用。

(5) 顶层 const:表示指针本身是常量。底层 const:表示指针所指对象是常量。

6. new/delete 关键字：

new/delete 和 malloc/free 的区别：前者为 C++的关键字，当其生成对象时会自动调用对象的构造与析构函数；后者只是一个 C 语言函数，其无法自动调用对象的构造与析构函数。

7. volatile 关键字：

(1) Volatile 关键词的第一个特性：易变性。所谓的易变性，在汇编层面反映出来，就是两条语句，下一条语句不会直接使用上一条语句对应的 volatile 变量的寄存器内容，而是重新从内存中读取。

(2) Volatile 关键词的第二个特性："不可优化"特性。volatile 告诉编译器，不要对我这个变量进行各种激进的优化，甚至将变量直接消除，保证程序员写在代码中的指令，一定会被执行。

(3) Volatile 关键词的第三个特性："顺序性"，能够保证 Volatile 变量间的顺序性，编译器不会进行乱序优化。

(4) C/C++ Volatile 变量，与非 Volatile 变量之间的操作，是可能被编译器交换顺序的。C/C++ Volatile 变量间的操作，是不会被编译器交换顺序的。哪怕将所有的变量全部都声明为 volatile，哪怕杜绝了编译器的乱序优化，但是针对生成的汇编代码，CPU 有可能仍旧会乱序执行指令，导致程序依赖的逻辑出错，volatile 对此无能为力.

六． 内联

1. 内联函数使用关键字 inline，内联函数从源代码层看，有函数的结构，编译之后则不具备，其直接将源代码嵌入每一个调用处。但是具体是否执行内联操作，得根据实际情况。
2. 内联函数在编译期展开，宏在预编译阶段展开（#define）.

七． 智能指针

(1) shared_ptr 指针： 所有指针共享一片内存，直到最后一个指针对象释放则会自动释放内存。

(2) unique_ptr 指针：同时只允许一个指针指向某对象。

(3) weak_ptr 指针：允许指向 share_ptr，但是不改变对象的计数值。

(4) shared_ptr 的循环引用问题

八． C++中的类型转换

(1) 因为 C++可以强制类型转换，因此不是类型安全的语言。

(2) 强制类型转换：static_cast，dynamic_cast，const_cast，reinterpret_cast

1> static_cast:可以更改一切非常量性质的，具有明确定义的类型转换。

2> const_cast:改变对象的底层 const 属性，去掉 const 性质，但是对象本身如果就是一个常量，执行类型转换后的写操作是未定义的。

3> reinterpret_cast:较为底层的转换，可将 int * -> char *

4> dynamic_cast:转换包含虚函数的基类派生类间的相互转换：将基类的指针或者引用安全地转化成派生类的指针和引用。

(3) RTTI 的相关内容：

1> Typeid 运算符：可以用于比较两个对象是否是同样的类型，特别是继承派生的使用中有一些需求。

2> dynamic_cast 运算符：可以安全的将基类指针或引用转化成派生类的指针或引用；当基类指针想执行派生类的函数时，当该函数不是虚函数时，可以通过以上方式实现。

九． 左值/右值的概念

(1) 对象移动：当将一个复制完之后就消失的变量，如果还对其实施拷贝，那是一件耗费资源的事，因此移动可以解决这一问题。

(2) 右值引用&&，左值引用&

十． C/C++编译过程

(1) 基本流程

1> 预处理：将所有的#include 以及宏替换成真正的内容(主要是#开头的内容)

2> 编译：将预处理后的文件转换成汇编代码（词法分析，语法分析，语法优化）。

3> 汇编：将汇编代码转换成机器码（将汇编代码生成可重定位(.o)的机器码）。

4> 链接：将多个目标文件以及需要的库文件链接成最终的可执行文件(与库文件链接)。**静态链接**（把库中的代码复制一份，更新困难但是运行速度快），**动态链接**（运行时加载同一份共享库，这样更新方便但有性能损耗）在 windows 与 Linux 系统下的库文件如下所示：

5> Windows 下：

1. **.dll:动态链接库，**作为共享函数库的可执行文件。
2. **.obj:对象文件，**相当于源代码中的二进制文件，未经重定位。
3. **.lib:**相当于多个.obj 文件的集合，本质与.obj 相同。

6> Linux 下：

4. **.so:**动态链接库，跟 Windows 平台类似。

5. **.o:**对象文件，相当于源代码对应的二进制。

6. **.a:**与.o 类似，多个.o 文件的集合。

7> #program once 与#ifndef/#define/#endif 的区别：

#program once 是有编译器来保证的,对于不同文件名同样的内容的代码无法识别。#ifndef/#define/#endif 由名字来保证，根据不同的命名来区分，因此移植性来说后者更好。

8> #define 中的#与##的作用：#是转化为字符串，##是连接两个参数。

第二章． 操作系统

一． 进程与线程

(1) 进程与线程的定义：

进程是对运行时程序的封装，是系统进行资源调度和分配的基本单元，实现了操作系统的并发。**线程**是进程的子任务，是 CPU 调度和分派的基本单位，实现进程内部的并发。

(2) 进程与线程的区别：

1> 每个进程都独自占用一个虚拟处理器，独自的寄存器组，指令计数器和处理器状态；每个线程完成不同的任务，但是共享同一地址空间，打开的文件队列和其他内核资源。

2> 一个线程只能属于一个进程，而一个进程可以有多个线程，但至少有一个线程，线程依赖进程而存在。

3> 进程是 CPU 资源分配的最小单元，线程是 CPU 调度的最小单元。

4> **系统开销：**进程在创建或撤销时系统都要为之分配或回收资源（内存空间，I/O 设备），进程的切换涉及到当前进程 CPU 环境的保存以及新的调

度运行 CPU 环境设置；而线程切换只需要保存和设置少量寄存器的内容，并不涉及寄存器管理方面的操作。因此进程切换的开销的也远大于线程切换的开销。

5> 进程编程调试简单可靠性高，但是创建开销大；线程相反，开销小，切换速度快，但是编程调试相对复杂。

6> 进程间不会相互影响，一个线程挂掉将导致整个进程挂掉。

7> 进程适应于多核，多机分布；线程适应于多核。

8> **通信：**进程的多个线程因为拥有共同的地址空间，因此通信比较单，但是进程间通信只能使用 IPC。

(3) 为啥有了进程还要有线程：

主要因为进程存在以下缺点：

1> 进程在同一时间只能干一件事，如果过程中发生阻塞，整个进程就会挂起

2> 引入线程这种粒度更小的调度单元，可以提高并发性。根据统计一个进程的开销大约是一个线程开销的 30 倍。

3> 多线程对于多任务使 CPU 更加有效，操作系统可以保证当线程不大于 CPU 数目时，不同线程运行在不同 CPU 上。

4> 可以改善程序结构，将进程拆分成几个独立的部分，有利于理解和修改。

(4) 线程切换中的寄存器

1> 线程切换的过程需要保存当前线程 ID,线程状态，堆栈，寄存器状态等信息。其中寄存器主要包括 SP PC EAX 等寄存器，其主要功能如下：

2> **SP:堆栈指针**，指向当前栈的栈顶地址。

3> **PC:程序计数器**，存储下一条要执行的指令

4> **EAX:累加寄存器**，用于加法乘法的缺省寄存器。

(5) 进程状态转换

1> 进程的五种基本状态：创建进程（进程正在被创建）；**就绪状态**（进程被加入到就绪队列中等待 CPU 调度运行）；**执行状态**（进程正在被运行）；**等待阻塞状态**（进程因为某种原因，比如等待 I/O,等待设备，而暂时不能运行）；**终止状态**（进程运行完毕）。

2> **交换技术：**多个进程竞争内存资源时，造成内存紧张。提出两种解决方法 1.交换技术（换出一部分进程到外存，腾出内存空间）2.虚拟存储技术：每个级才能哼只能装入一部分程序和数据。

3> **活动阻塞：**进程在内存，但是由于某种原因被阻塞。**静止阻塞**：进程在外存，同时被某种原因阻塞。**活动就绪**：进程在内存，处于就绪状态，只要 CPU 调度就可以直接运行。**静止就绪**：进程在外存，处于就绪状态，只要调度到内存，给 CPU 调度就可以运行。

(6) 常见的线程模型

1> Future 模型：

2> Fork&join 模型：

3> Actor 模型：

4> 生产者消费者模型：

5> Master-worker 模型：

二． 进程间通信

进程不共享地址空间，但是如果进程需要协作，可以通过什么方式通信呢？主要包含以下方式：管道（普通管道，命名管道 FIFO）消息队列，信号

量，信号，共享内存以及套接字。

(1) 管道

1> **管道主要分为无名管道和命名管道**：管道可用于具有亲缘关系的父子进程间的通信，有名管道除了具有管道所具有的功能外，还允许无亲缘关系的进程间的通信。

2> **普通管道 PIPE**，数据只能在一个方向上流动，具有固定的读端和写端。他只能用于具有亲缘关系的进程之间通信。它同样可以使用 read/write 等函数，但他不是文件，并不属于任何文件系统，只存在与内存中

3> **有名管道 FIFO** 可以在无关的进程之间交换数据。FIFO 有路径名与之相关联，它一一种特殊设备文件形式存在于文件系统中。

(2) 系统 IPC

1> **消息队列**，是消息的链接表，存放在内核中，一个消息队列由一个标识符（队列 ID）来标记。(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特点)。具有写权限的进程可以按照一定规则向消息队列中添加新信息，对消息队列有读权限的进程，则可以从消息队列中读取信息.

2> **消息队列的特点：**面向记录，其中消息具有特定的格式以及特定的优先级。消息队列独立于发送与接受进程，进程终止时，消息队列及其内容不会被删除。消息队列可以实现消息的随机查询，消息不一定要以先进先出的次序读取，也可以按消息的类型读取。

3> **信号量是一个计数器**，可以用来控制多个进程对共享资源的访问。信号量用于实现进程间的互斥与同步，而不是存储进程间通信的数据。

4> **信号量的特点：**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。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。每次对信号量的 PV 操作不仅限于信号量值加 1 或减 1，而且可以加减任意正整数。支持信号量组。

5> **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。

6> **共享内存**使得进程可以访问同一块内存，不同进程可以及时看到对方进程中对数据的更新，这种方式需要依靠互斥锁和信号量等同步操作。

7> **共享内存的特点：**共享内存是最快的一种 IPC，因为进程是直接对内存进行存取。因为多个进程可以操作，需要进行同步。信号量+共享内存通常结合在一起使用，信号量用来同步对共享内存的访问。

(3) 套接字 SocKet

套接字不仅可以用于本机间进程，也可以用于不同主机间的通信，这部分也是网络编程的重点。

(4) 线程间同步

线程间通信通过全局（公有）变量进行通信即可，所以只要在多个线程间采取同步策略即可，主要有如下策略。

1> 临界区：通过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。

2> 互斥量：<实现互斥>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，所以可以保证公共资源不会被多个线程同时访问。

3> 信号量：<实现同步，基于互斥，同步即访问顺序控制>为控制具有有限

数量的用户资源而设计，它允许多个线程在同一时刻访问一个资源，但需要限制同一时刻访问此资源的最大线程数。

4> 事件（信号）：通过通知操作的方式来保持多线程同步，还可以方便实现多线程优先级的比较操作。

5> 锁机制

1. **互斥锁/mutex:**用于保证任何时候，都只有一个线程访问该对象。当获取锁操作失败时，线程会进入睡眠，等待锁释放时被唤醒。
2. **读写锁：**rwlock:分为读锁和写锁。处于读操作时，可以允许多个线程同时获得读操作。但是同一时刻只有一个线程可以获得写锁。其他获取写锁失败的线程都会进入睡眠状态，直到写锁释放被唤醒。此外，写锁会阻塞所有的读写锁。
3. **自旋锁：**在任何时刻只能有一个线程访问对象，但是当获取锁操作失败时，不会进入睡眠，而是在原地自旋，直到锁被释放。这样节省了线程从睡眠状态到被唤醒期间的消耗。在加锁时间短暂的环境下会极大的提高效率。但是如果加锁时间过长，则会非常浪费 CPU 资源。
4. **RCU:**即 read-copy-update,在修改数据时，首先需要读取数据，然后生成一个副本，对副本进行修改。修改完成后，再将老数据 update 成新的数据。使用 RCU 时，读者几乎不需要同步开销，既不需要获得锁，也不使用原子指令，不会导致竞争，因此就不用考虑死锁问题。而对写者的同步开销比较大，它需要复制被修改的数据，还必须使用锁机制同步并行其他写者修改操作。少量写的情况下效率较高。

三． 内存管理机制

操作系统的基本功能包括系统调度，内存管理，文件系统，I/O 交互，死锁等内容，进程线程与进程间通信属于系统调度的主体内容，一下了解一下内存管理的知识。

(1) 操作系统中的程序的内存结构<由地址到高地址>：

1> **代码段：**存放程序和只读变量。

2> **数据段：**也称静态区，分为 BSS(未初始化)和已初始化数据。

3> **堆：**动态数据区，大小不定。

4> **栈：**动态数据区，大小限定 2M，但可以调整。

5> **命令行参数和环境变量。**

(2) 操作系统中结构体字节对齐

1> 原因：1.平台原因：不是所有硬件平台都能访问任意地址上的任意数据，某些硬件平台只能在某些地址处取特定的类型的数据，否则抛出硬件异常。**2.性能原因**：数据结构应该尽可能地在自然边界上对齐。原因在于，为了访问未对齐内存，处理器需要两次内存访问，而对齐内存只要做一次。

2> 规则：按照#pragma pack(n)对齐。

(3) Linux 虚拟地址空间

1> **目的**：为了防止不同进程同一时刻在物理内存中运行而对物理内存的争夺和践踏，采用了虚拟内存。

2> **虚拟内存技术**使得不同进程在运行过程中，它所看到的是自己独自占有了当前系统的 4G 内存。所有进程共享同一物理内存，每个进程只把自己目前需要的虚拟内存空间映射并存储到物理内存上。事实上，

每个进程创建加载时，内核只是为进程“创建”了虚拟内存布局，具体就是初始化进程控制表中的相关的链表，实际上并不立即就把虚拟内存对应位置的程序数据和代码拷贝到物理内存中，只是建立好虚拟内存和磁盘文件的映射就好，等到运行到对应的程序时，才会通过缺页异常来拷贝数据。还有进程运行过程中，要动态分配内存，比如 malloc 时，也只是分配了虚拟内存，即为这块虚拟内存对应的页表项做相应设置，当进程真正访问到此数据时，才引发缺页异常。

3> **虚拟内存的好处**：**一是**可以扩大地址空间。**二是**起到内存保护的作用，每个进程运行在各自的虚拟内存地址空间，互相不干扰对方，虚存还对特定的内存地址提供写保护，可以防止代码或数据被篡改。**三是**公平内存分配，采用了虚存之后，每个进程都相当有相同大小的虚存空间。**四是**进程通信时，可采用虚存共享的方式实现。**五是**当不同的进程使用同样的代码时，比如库文件中的代码，物理内存可以只存储一份这样的代码，不同的进程只需要把自己的虚拟内存映射过去就可以了，节省内存。**六是**虚拟内存很适合在多道程序设计系统中使用，许多程序的片段同时保存在内存中。当一个程序等待他的一份读入内存时，可以把 CPU 交给另一进程使用，在内存中可以保留多个进程，系统并发度提高。**七是**在程序需要分配连续的内存空间的时候，只需要在虚拟内存分配连续空间，而不需要实际物理内存的连续空间，可以利用碎片。

4> **虚拟内存的代价：一是虚拟内存**需要建立很多数据结构，这些数据结构需要占用额外的内存。**二是**虚拟地址到物理地址的转换，增加了指

令的执行时间。**三是**页面的换入换出需要磁盘 I/O，这是很耗时的。**四是**一页中只有一部分数据，会浪费内存。

(4) 虚拟内存和物理内存对应的方式

1> **物理内存**是对内存芯片级的寻址，**虚拟内存**是对整个内存的抽象描述。

2> step1:CPU 段式管理中---逻辑地址转线性地址

3> step2:页式管理---线性地址转物理地址

(5) 操作系统的缺页中断

1> **定义：**malloc 和 mmap 等内存分配函数，在分配时只是建立了进程虚拟地址空间，并没有分配虚拟内存对应的物理内存。当进程访问这些没有建立映射关系的虚拟内存时，处理器自动触发一个缺页异常。

2> **在请求分页系统中，**可以通过查询页表中的状态来确定所要访问的页面是否存在与内存中。每当所要访问的页面不在内存中，会产生一次缺页中断，此时操作系统会根据页表中的外存地址在外存中找到所缺的一页，将其调入内存。

3> **缺页本身是一种中断：需**要经历 1.保护 CPU 现场 2.分析中断原因 3.转入缺页中断处理程序进行处理 4.恢复 CPU 现场，继续执行。

4> **缺页中断是由于访问的页面不存在于内存时，由硬件所产生的一种特殊中断**，与一般中断存在区别：1.在执行期间产生和处理缺页中断信号。2.一条指令在执行期间，可能产生多次缺页中断。3.缺页中断返回时，只想产生中断的一条指令，而一般中断返回是只想下一条指令。

(6) OS 的缺页置换算法

1> 定义：当访问一个内存中不存在的页，并且内存已满，则需要从内存

调出一个页或将数据送至磁盘对换区，替换一个页，这种现象叫做缺页置换，当前操作系统最常采用的缺页置换算法如下。

2> **先进先出（FIFO）算法：**置换最先调入内存的页面，即置换在内存中驻留时间最久二点页面。按照进入内存的先后次序排成队列，从队尾进入，从队首删除。

3> **最近最少使用（LRU）算法**:置换最近一段时间以来长时间未访问的页面根据程序局部性原理，刚被访问的页面，马上又要被访问，则较长时间内没有被访问的页面，可能最近不会被访问。

4> **最不经常访问淘汰算法（LFU）：**如果数据过去被访问多次，那么将来访问的频率也高。

5> **最近未使用 K 次淘汰算法（LRU-K）：**

(7) 操作系统的页表寻址

1> **定义：**页式内存管理，内存分成固定长度的一个个页片。操作系统为每一个进程维护了一个从虚拟地址到物理地址映射关系的数据结构，叫页表，页表的内容就是该进程的虚拟地址到物理地址的一个映射。页式内存管理的优点就是比较灵活，内存管理以较小的页为单位，方便内存换入换出和扩充地址空间。

2> **Linux 最初的二级页表机制：**两级分页机制将 32 位虚拟空间分成 3 段，低十二位表示业内偏移，高二十位分成两段分别表示两级页表的偏移。PGD（Page Global Directory）:最高 10 位，全局页目录表索引；PTE（Page Table Entry）:中间 10 位，页表入口索引。

3> **Linux 的三级页表机制：**X86 引入地址扩展后，可以支持大于 4G 的

物理内存（36 位），但虚拟地址依然是 32 位。Linux 新增了一个层级 PMD。cr3,PGD,PMD,PTE.

4> **Linux 的四级页表机制：**最大寻址增加到 512G.PGD-PUD-PMD-PTE. 或 PML4-PGD-PMD-PTE.

四． 文件系统

(1) 软链接与硬链接：

为了解决文件共享问题，Linux 引入软链接和硬链接。除了为 Linux 解决文件共享使用，还带来了隐藏文件路径，增加权限安全及节省存储等好处。若 1 个 inode 号对应多个文件名，则为硬链接，即硬链接就是同一文件使用了不同的别名，使用 ln 创建。若文件用户数据块中存放的内容是另一个文件的路径名指向，则该文件是软链接，软连接是一个普通文件，有自己独立的 inode,但是其数据块内容比较特殊。

五． 输入输出-I/O 设备

六． 死锁

(1) 死锁产生的必要条件：

1> **互斥条件**：一个资源每次只能被一个进程使用。

2> **请求与保持条件：**一个进程因请求资源而阻塞时，对已获得的资源不放。

3> **不剥夺条件：**进程已获得的资源，在未使用完之前，不能强行剥夺。

4> **循环等待条件：**若干进程之间形成一种头尾相接的循环等待资源关系。

第三章． 计算机网络

一． 网络层次模型

(1) TCP/IP 4 层模型：

1> 网络接口层：MAC（mac 地址） VLAN（虚拟局域网）

2> 网络层：IP ARP（地址解析协议） ICMP（控制报文协议）

3> 传输层：TCP UDP

4> 应用层：HTTP（超文本传输协议） DNS（域名系统协议） SMTP（电子邮件协议） FTP（文件传输协议）

(2) OSI 7 层模型：

物理层->数据链路层->网络层->传输层->会话层->表示层->应用层

(3) MAC 协议与 IP 地址

MAC 地址是一个硬件地址，用来定义网络设备的位置，主要由数据链路层负责。而 **IP 地址**是 IP 协议提供的一种统一的地址格式，为互联网上的每一个网络和每一台主机分配一个逻辑地址，来屏蔽物理地址的差异。

二． TCP 协议

(1) TCP 协议的三次握手：

1> 建立 TCP 连接时服务器与客户端会发生 3 次数据交互。

2> Client 发送连接请求，将标志位 SYN 置为 1，随机产生一个值 seq=J，并将该数据包发送给 Server，Client 进入 SYN_SENT 状态，等待 Server 确认

3> Server 收到数据包后由标志位 SYN=1 直到 Client 请求建立连接，Server 将标志位 SYN 和 ACK 都置为 1，ack=J+1，随机产生一个值 seq=K，并将该数据包发送给 Client 以确认连接请求，Server 进入 SYN_RCVD 状态

4> Client 收到确认后，检查 ack 是否为 J+1，ACK 是否为 1，如果正确则将标志位 ACK 置为 1，ack=K+1,并将该数据包发送给 Server，Server 检查 ack 是否为 K+1，ACK 是否为 1，如果正确则连接建立成功，Client 和 Server 进入 ESTABLISHED 状态，完成三次握手，随后 Client 与 Server 之间可以开始传输数据了。

5> TCP 为啥要 3 次握手?

三次握手可以防止已经失效的连接请问报文突然又传输到服务器端导致服务器资源的浪费。TCP 握手不可以 2 次，4 次的原因：2 次的话无法确认双方的发送与接受都可以，握手中服务端确认收到与恢复可以合并为 1 次。

(2) TCP 的四次挥手：

1> TCP 在请求断开连接时会发送 4 次数据交互。<由于 TCP 连接是全双工的，因此每个方向都必须要单独进行关闭，首先关闭的一方执行主动关闭，另一方执行被动关闭>

2> **数据传输结束后，客户端的应用进程发出连接释放报文段**，并停止发送数

据，客户端进入 FIN_WAIT_1 状态，此时客户端依然可以接收服务器发送来的数据。

3> 服务器收到 FIN 后，发送一个 ACK 给客户端，确认序号为收到的序号+1，服务器进入 CLOSE_WAIT 状态。客户端收到后进入 FIN_WAIT_2 状态。

4> 当服务器没有数据发送时，服务器发送一个 FIN 报文，此时服务器进入 LAST_ACK 状态，等待客户端确认。

5> **客户端收到服务器的 FIN 报文后，**给服务器发送一个 ACK 报文，确认序列号为收到的序号+1。此时客户端进入 TIME_WAIT 状态，等待 2MSL,然后关闭连接。

6> 为什么 TCP 的终止要 4 次挥手：

1. **当客户端确认发送完数据且知道服务器已经接受完了，**想要关闭发送数据口，就会发送 FIN 给服务器。
2. **服务器收到客户端发送的 FIN，表示收到了，就会发送 ACK 回复。**
3. **但是这个时候服务器还在发送数据，没有想要关闭数据口的意思，所以 FIN 与 ACK 不是同时发送的，**而是等待服务器数据发送完了，才会发送 FIN 给客户端。
4. **客户端收到服务器发来的 FIN，**知道服务器数据也发送完了，回复 ACK,客户端等待 2MSL 以后，没有收到服务器传来的任何消息，知道服务器已经收到自己的 ACK,客户端就关闭连接，服务器也关闭连接。
5. **2MSL 的意义：**1.保证最后一次握手报文未到 S,能进行超时重传。2. 2SML 时间内，这次连接的所有报文都会消失，不会影响下一次连接。

(3) TCP 可靠传输的原因：

1> 序列号，确认应答，超时重传

数据到达接收方，接收方需发出一个确认应答，表示已经收到该数据段并且确认序号会说明它下一次需要接受的数据序列号。如果发送方迟迟未收到确认应答，那么可能是发送的数据丢失，也可能是确认应答丢失，这时发送方在等待一定时间后会进行重传。这个时间一般是 2*RTT+一个偏差值。

2> 窗口控制与快速重传

TCP 会利用窗口控制来提高传输速度，意思是在一个窗口大小内，不用一定要等到应答才能发送下一段数据，窗口大小就是无需等待确认可以继续发送数据的最大值，如果不使用窗口控制，每一个没收到确认应答的数据都要重发。

使用窗口控制，如果数据段 1001-2000 丢失，后面数据每次传输，确认应答都会不停的发送序列号为 1001 的应答，表示我要接受 1001 开始的数据，发送端如果收到 3 次相同应答，就会立刻进行重发，但是还有种情况是有可能是数据都受到了，但是有的应答丢失了，这种情况不会进行重发，因为发送端知道，如果是数据段丢失，接收端不会放过他，会不断向他提醒。

3> 拥塞控制

如果把窗口定的很大，发送端连续发送大量的数据，可能会造成网络的拥堵，甚至造成网络的瘫痪。所以 TCP 在为了防止这种情况进行了拥塞控制。

慢启动：定义拥塞窗口，一开始该将窗口大小设为 1，之后每次收到确认应答（经过一个 rtt）将窗口大小*2。

拥塞避免：设置慢启动阈值，一般开始都设为 65536。拥塞避免是当拥塞窗口大小达到这个阈值，拥塞窗口的值不再指数上升，而是加法上升，一

次避免拥塞。

将报文段的超时重传看作拥塞：一旦发生超时重传，我们需要先将阈值设为当前窗口大小的一般，并且将窗口大小设为初值 1，然后重新进入慢启动过程。

快速重传：在遇到 3 次重复确认应答（高速重发控制）时，代表收到了 3 个报文段，但这之前的一个段丢失了，便对它进行立即重传。然后先将阈值设置为当前窗口大小的一半，然后将拥塞窗口大小设为慢启动阈值+3 的大小。

这样可以达到：在 TCP 通信时，网络通信吞吐量呈现逐渐上升，并且随着拥堵来降低吞吐量，再进入慢慢上升的过程，网络不会轻易的发生瘫痪。

三． TCP 与 UDP 的差异

(1) **连接：**TCP 是面向连接的传输协议，即传输数据之前必须先建立好连接。UDP 无连接。

(2) **服务对象：**TCP 是点对点的两点间服务，即一条 TCP 连接只能有两个端点，UDP 支持一对一，一对多，多对一，多对多的交互通信。

(3) **可靠性：TCP 是可靠交付：无差错，不丢失，不重复，按序到达。UDP 是尽最大努力交付，不保证可靠交付。**

(4) **拥塞控制，流量控制：**TCP 有拥塞控制和流量控制保证数据传输的安全性。UDP 没有拥塞控制，网络拥塞不会影响主机的发送效率。

(5) **报文长度：**TCP 是动态报文长度，即 TCP 报文长度根据接收方的窗口大小和当前网络拥塞情况决定的。UDP 面向报文，不合并，不拆分，保留上面传下来报文的边界。

(6) **首部开销：**TCP 首部开销大，首部 20 个字节。UDP 首部开销小，8 字节。

(7) **适用场景：**TCP:文件传输，重要状态更新。UDP:视频传输，实时通信。

四． HTTP 协议

(1) HTTP 协议简介

1> 全称：**Hyper Text Transfer Protocol 超文本传输协议**。是用于从万维网服务器传输超文本到本地浏览器的传输协议。

2> HTTP 协议是一个**基于 TCP/IP 通信协议**来传输数据的（HTML 文件，图片文件，查询结果等）。

3> HTTP 是一个属于应用层的面向对象的协议，由于其简捷，快速的方式，适用于分布式超媒体信息。

4> **HTTP 协议工作于客户端-服务端构架上**。浏览器作为 HTTP 客户端通过 URL 向 HTTP 服务端（WEB 服务器）发送所有请求。WEB 服务器根据接受到的请求后，向客户端发送响应信息。

(2) HTTP 协议的特点

1> **简答快捷：**客户端向服务器请求服务时，只需传送请求方法和路径。请求的方法常有 GET,HEAD,POST.。每种方法规定了客户与服务器联系的类型不同。由于 HTTP 协议简单，使得 HTTP 服务器的程序规模小，因而通信速度很快。

2> **灵活：**HTTP 允许传输任意类型的数据对象。正在传输的类型由 Content-Type 加以标记。

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。

4> **无状态：**无状态是指协议对于事务处理没有记忆能力。缺少状态意味着如果后续处理需要前面的信息，则它必须重传，这样可能导致每次连接传送数据量增大。另一方面，在服务器不需要先前信息时它的应答就较快。

5> 支持 B/S 或 C/S 模式，默认端口 80，基于 TCP 协议。

(3) HTTP 工作过程概述

1> **简述：**HTTP 协议定义 Web 客户端如何从 Web 服务器请求 Web 页面，以及服务器如何把 Web 页面传给服务端。HTTP 协议采用了请求/响应模型。客户端向服务器发送一个请求报文，请求报文包含请求的方法，URL，协议版本，请求头部和请求数据。服务器以一个状态行作为响应，响应内容包含协议的版本，成功或者错误代码，服务器信息，响应头部和相应数据。

2> **客户端连接到 Web 服务器：**一个 HTTP 客户端，通常是浏览器，与 Web 服务器的 HTTP 端口（默认为 80）建立一个 TCP 套接字连接，例如，http:www.baidu.com.

3> **发送 HTTP 请求：**通过 TCP 套接字，客户端向 Web 服务器发送一个文本的请求报文，一个报文包含请求行，请求头部，空行和请求数据 4 部分。

4> **服务器接受请求并返回 HTTP 响应：**Web 服务器解析请求，定位请求资源。服务器将资源复本写到 TCP 套接字，由客户端读取。一个响应由状态行，响应头部，空行和响应数据 4 部分组成。

5> **释放 TCP 连接：**若 Connection 模式为 close，则服务器主动关闭 TCP 连接，客户端被动关闭连接，释放 TCP 连接；若 Connection 模式为 keepalive，

则该连接会保持一段时间，在该时间内可以继续接受请求。

6> **客户端浏览器解析 HTML 内容：**客户端浏览器首先解析状态行，查看表明是否成功的状态代码。然后解析每一个响应头，响应头告知以下为若干字节的 HTML 文档和文档的字符集。客户端浏览器读取响应数据 HTML，根据 HTML 的语法对其进行格式化，并在浏览器窗口显示。

(4) HTTP 和 HTTPS 的区别

1> HTTP 协议是以明文的形式在网络中传播数据，而 HTTPS 协议传输的数据则是经过 TLS 加密后的，HTTPS 具有更高的安全性。

2> HTTPS 在 TCP 三次握手之后，还需要进行 SSL 的 handshake，协商加密使用的是对称加密密钥。

3> HTTPS 协议需要服务端申请证书，浏览器端安装对应的根证书。

4> HTTPS 协议端口是 80，HTTPS 协议端口是 443。

5> **HTTPS 优点：**HTTPS 传输数据过程中使用密钥进行加密，所以安全性更高。HTTPS 协议可以认证用户和服务器，确保数据发送到正确的用户和服务器。

6> **HTTPS 缺点：**HTTPS 握手阶段延时较高，由于在进行 HTTP 会话之前还需要进行 SSL 握手，因此 HTTPS 协议握手阶段延时增加。HTTPS 部署成本高，一方面 HTTPS 协议需要使用证书来验证自身的安全性，所以需要购买 CA 证书；另一方面由于采用 HTTPS 协议需要进行加解密的计算占用的 CPU 资源较多，需要的服务器配置或数目高。

(5) HTTP 状态码简介：

HTTP 协议的响应报文由状态行，响应头部和响应包体组成，其响应状态码

总体描述如下：

1> **1xx：**指示信息—表示请求已接收，继续处理。

2> **2xx：**成功—表示请求已成功，理解，接受。

3> **3xx：**重定向—要完成必须进行进一步的操作。

4> **4xx：**客户端错误—请求有语法错误或请求无法实现。

5> **5xx：**服务器端错误—服务器端未能实现合法的请求。

6> 常见的状态代码，状态描述的详细说明如下。

200 OK:客户端请求成功。　　206：服务器已经正确处理部分 GET 请求

300：可选重定向　　301：永久重定向　　302：临时重定向　　304：

403：服务器收到请求，但是拒绝服务

五．　协议应用举例

(1) HTTP 响应流程（在浏览器地址键入 URL,按下回车之后经历以下流程）

1> 浏览器向 DNS 服务器请求解析该 URL 中域名锁对应的 IP 地址。

2> 解析 IP 地址后，根据该 IP 地址和默认端口 80，和服务器建立 TCP 连接。

3> 浏览器发出读取文件的 HTTP 请求（URL 中域名后面部分对应的文件），该请求报文作为 TCP 三次握手的第三个报文的错误数据发送给服务器。

4> 服务器堆浏览器做出响应，并把对应的 HTML 文本发送给浏览器。

5> 释放 TCP 连接。

6> 浏览器将该 html 文本并显示内容。

(2) HTTP 响应中协议分析：<以使用百度搜索为例>

1> 浏览器中输入 URL：浏览器要将 URL 解析为 IP 地址，解析域名要用到**DNS 协议**，主机首先会查询缓存，如果没有就给本地 DNS 发送查询，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…。DNS 服务器基于 UDP，因此用到 **UDP 协议。**

2> 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 协议报文格式上已经提到。http 生成一个 get 请求报文，将该报文传给 **TCP 层进行加密**。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 层数据包发送给 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。在一个网段内寻址是通过以太网协议实现，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。**

第四章． 数据库与设计模式的基本知识

一． 常用的设计模式

(1) 观察者（发布-订阅）模式：

1> 行为模式的一种，定义了一种一对多的依赖关系，让多个观察者同时监听某一主题对象。这个主题对象在状态变化时，会通知所有的观察者对象，使他们能够自动更新自己。

2> 用于代码建的解耦，当一个事件被多个（数量未知）可以通过增加或减少观察者的方式实现。

(2) Reactor（反应器）模式：

应用于同步非阻塞 io 操作，即 io 多路复用，即内核通知用户线程数据达到，此时用户线程将阻塞于数据读取。

(3) Preactor 模式：

运用于异步 io 操作，在异步 io 模型中，当用户线程收到通知时，数据已经被内核读取完毕，并放在用户线程指定的缓冲区。

(4) 单例模式（Singlenton）模式：

1> **作用：**确保全局唯一性，节约系统资源，避免对共享资源的多场占用。

2> **单例模式主要解决一个全局使用的类频繁的创建和销毁的问题**，单例模式下可以确保某一个类只有一个实例，而且自行实例化并向整个系统提供这个实例。单例模式有三个要素：一是某个类只能有一个实例，二是它必须自行创建这个实例，三是他必须自行向整个系统提供这个实例。

3> **实现方法：1.**将该类的构造方法定义为私有方法，这样其他处的代码无法通过调用该类的构造方法来实例化该类的对象，只有通过该类提供的静态方法来得到该类的唯一实例。**2.**在该类中提供一个静态方法，当我们调用这个方法时，如果类持有引用不为空就返回这个引用，如果类保持的引用为空就创建该类的实例并将实例的引用赋予该类保持的引用。

4> **单例模式的多线程安全问题**，有以下方式实现：

懒汉模式：在第一次调用时初始化，并使用双重锁防止多线程创建多个实例。

饿汉模式：静态的创建实例，一开始就初始化。

5> **单例模式代码示例**：

```
// 懒汉模式
class SingleTon
{
private:
  // 禁止外部构造
  SingleTon() {};
  //禁止拷贝
  SingleTon(const SingleTon&) {};
  //禁止赋值
  SingleTon& operator=(const SingleTon&) {};
  static SingleTon* instance;
  static std::mutex SingMutex;

public:
  // 获得单例的指针
  static SingleTon* GetInstance()
```

```
        {
            if (instance == NULL)
            {
                SingMutex.lock();
                if (instance == NULL)
                {
                    instance = new SingleTon();
                }
                SingMutex.unlock();
            }
            return  instance;
        }
    };
    // 懒汉模式：
    class Singlenton
    {
    private:
        Singlenton() {}
        ~Singlenton(){}
        static Singlenton* m_pInstance;

    public:
        static Singlenton* getInstance()
        {
            return m_pInstance;
        }
    };
    Singlenton* Singlenton::m_pInstance = new Singlenton();
```

(5) 工厂模式：

1> **作用：**工厂模式主要解决接口选择的问题，该模式下定义一个创建对象的接口，让其子类自己决定实例化哪一个工厂类，使其创建过程延迟到子类进行。

2> **简单工厂：**不同的运算符新建不同的类，在主类中根据输入进行选择，界面与运算分离。

3> **工厂方法：**简单工厂方法会根据客户端输入选择实例化不同的对象，与客户端交互的部分由一个工厂类使用 switch-case 的方式修改实例化的方式。而工厂方式则是让所有的类继承同一个基类，具体实例化对象让客户端进行选择。

(6) 装饰器模式：

1> **作用：**对已经存在的某些类就行装饰，以此来扩展一些功能，从而动态的为一个对象增加新的功能。装饰器模式是一种用于代替继承的技术，无需通过继承就能扩展对象的新功能。使用对象的关联关系代替继承关系（将一个类作为另一个类函数的入口参数），更加灵活，同时避免类型体系的快速膨胀。

二． 设计模式/面向对象（OOP）基本原则

(1) 单一职责原则：

一个是避免相同的职责分散到不同的类，**另一个**是避免一个类承担太多职责。减少类的耦合，提高类的复用性。

(2) 接口隔离原则：

每个接口服务于一个子模块，简单说就是多个专门的接口比使用单个接口好很多。1. 一个类对另一个类的依耐性应该是建立在最小接口上。2.客户端程序不应该依耐它不需要的接口方法。

(3) 开放封闭原则：

Open 意思是模块的行为必须是开放的，支持扩展的，而不是僵化的。Closed 意思是在对模块功能进行扩展时，不应该影响或大规模影响已有的程序模块。总结：**一个模块在扩展性方面应该是开放的而在更改性方面应该是封闭的。**

(4) 替换原则：

子类型必须能够替换掉他们的父类型，并且出现在父类能够出现的任何地方。

1.父类中的方法都要在子类中实现或者重写并且派生类只实现其抽象类中生命的方法，而不应该给出多余的方法定义或实现。**2.**在客户端程序中只应该

使用父类对象而不应当直接使用子类对象，这样可以实现运行期间绑定。

(5) 依赖倒置原则

上层模块不应该依赖于下层模块，他们共同依赖于一个抽象，即父类不能依赖子类，他们都要依赖抽象类。抽象不能依赖于具体，具体应该依赖于抽象。

二． 数据库基础知识

(1) 数据库的事物及其特性:

1> 定义：事务是由一系列对数据进行访问与更新操作所组成的一个程序执行逻辑单元。事务是 DBMS 中最基础的单位，事务不可分割。事务具有四个基本特征，分别是原子性，一致性，隔离性，持久性。

2> 原子性：指事务包含的操作要么全部成功，要么全部失败回滚，因此事务的操作如果成功就必须要完全应用到数据库，如果操作失败则不能对数据库有任何影响。

3> 一致性：指事务必须使数据库从一个一致性的状态变换到另一个一致性状态，也就是一个事务执行之前和执行之后都处于一致性状态。

4> 隔离性：指多个用户并发访问数据库时，比如操作同一张表时，数据库为每一个用户开启的事务，不能被其他事务的操作所干扰，多个并发事务之间要相互隔离。

5> 持久性：指一旦事务提交，其所做的修改将会永远保存到数据库中。

(2) 事务隔离性的几个级别：

1> Read Uncommitted（读取未提交内容）:最低的隔离级别，什么都不需要做，一个事务可以读到另一个事务未提交的结果。所有并发事务问题都会发生。

1> Read Committed（读取提交内容）：只有在事务提交后，其更新结果才会被其他事务看见。可以解决脏读问题。

2> Repeated Read（可重复读）:在一个事务中，对同一份数据的读取结果总是相同的，无论是否其他事务对这份数据进行操作，以及这个事务是否提交。可以解决脏读，不可重复度的问题。

3> Serialization（可串行化）：事务串行化执行，隔离级别最高，牺牲了系统的并发性，可以解决并发事务的所有问题。

(3) 数据库的索引：

1>定义：数据库索引是为了增加查询速度而对表字段附加的一种标识，是对数据库中的一列或多列的值进行排序的一种结构。DB 在执行一条 SQL 语句时，默认的方式是根据搜索条件进行全表扫描，遇到匹配条件就加入搜索结果集合。如果我们对某一字段增加索引，查询时就会先去索引列表中一次定位到特定值的列数，大大较少遍历匹配的行数，所以能明显增加查询的速度。

2>优点：1.通过创建唯一性索引可以保证数据库表中每一行数据的唯一性；2.可以大大加快数据的检索速度，这也是创建索引索引的最主要原因；3.可以加速表与表之间的连接，特别是在实现数据参考完整性方面特别有意义；4.在使用分组和排序子句进行数据检索时，同时可以显著减少查询中分组和排序的时间；5.通过使用索引，可以在查询的过程中，使用优化隐藏器，提高系统的性能。

3> 缺点：1.查询中很少使用或者参考的列不应该增加索引。这是因为，由于这些列取值很少，例如人事表的性别列，在查询的结果中，结果集的数据占了表数据行的很大比例，即需要在表中搜索的数据行比例很大。增加索引，并不能明显加快检索速度；2.定义为 text，image 和 bit 数据类型的不应该增加索引，这是因

为列的数据量要么相当大，要么取值很少；3.当修改性能远远大于检索性能时，不应该创建索引。这是因为，修改性能和索引性能是相互矛盾的。当增加索引时，会降低检索性能。当减少索引时会提高修改性能，降低检索性能。因此，当修改性能远远大于检索性能时，不应该创建索引。

(4) 索引底层原理：

1> B-树：B-树就是 B 树，即平衡多路查找树，m 阶的 B 树其基本定义如下：

1. 根节点不是叶节点，则其至少有两棵子树。**2.**每一个非根的分支节点都有 k-1 个元素和 k 个孩子，其中[m/2]<=k<=[m]。每一个叶子结点 n 都有 k-1 个元素。**3.** 所有叶子节点都位于同一层次。**4.** 所有分支包含下列信息数据（n,A0,K1,…,Kn,An）。

2>B+树：B+树是相对 B 树做的一种改进，因为 B 树中既包含索引右包含树据，因此树的深度会相对较高，在一次搜索中，磁盘的最大读取次数正比于树的深度，B+树相对 B 树做了如下改进：

1.B+树每个节点的指针上限是 2d 而不是 2d+1。**2.**内节点不存储 data，只存储 key。**3**.叶子节点不存储指针。**4**.其次同一层有带顺序访问的指针存在。

(5) 关系型数据库的三大范式：

1> 范式：是对数据库设计规范的一些标准，常见的标准有第一，二，三范式。

2> 第一范式：要求属性具有原子性，不可再分解。

3> 第二范式：要求记录具有唯一标识，即实体的唯一性，即不存在部分依耐。

4> 第三范式：要求任何字段不能由其他字段派生出来，他要求字段没有冗余，即不存在传递依赖。

5> 范式化设计的优点：1.可以尽量减少数据冗余。2.数据表更新快，体积小。

6> 范式化设计的缺点：1.对于查询需要对多个表进行关联，导致性能降低。2.更难进行搜索优化。

(6) 连接的形式：

1> Left join:

2>Right join:

4> Inner join:

(7) 数据库其他基础：

1> 主键：表中的一列（或几列）可以唯一标识自己。数据库应该总定义主键。

2> 外键：外键是用于和另外一张表关联列，外键一般在另外一张表是主键。

三． MySQL 入门：

(1) MySQL 对不同隔离级别的支持：

支持 read-uncommitted,read-committed,repeatable-read，serializable。

(2) MySQL 的 MVCC 机制（多版本并发控制）：

MVCC 是一种多版本并发控制机制，是 MySQL 的 InnoDB 存储引擎实现隔离级别的一种具体方式，用于实现提交读和可重复度这两种隔离级别。MVCC 的通过保存数据在某个时间点的快照来实现该机制，其在每行记录偶棉保存两个隐藏的列，分别保存这个行的创建版本号和删除版本号，然后 Innodb 的 MVCC 使用到的快照存储在 Undo 日志中，该日志通过回滚指针把一个数据行所有快照连接起来。

(3) SQL 查询优化的方式：

建立索引，避免全表扫描，优化查询。

(4) MySQL 引擎的区别：

1> MySQL 在 V5.1 以前默认的引擎是 **MyIsam**，之后默认的引擎是 **InnoDB**。

2>InnoDB 引擎的介绍：

1. 特点：支持 ACID 事务，可以从灾难中恢复（通过 bin-log 日志等）并实现了 4 种隔离级别，有行级锁定和外键约束，支持自动增加列属性 auto_incre。

2. 适用场景：经常更新的表，适合处理多重并发的更新请求。

3. 索引结构：B+Tree 索引结构，Innodb 的索引文件本身就是数据文件，成为**聚集索引，**这个索引的 key 就是数据表的主键。其辅助索引数据域也是响应记录主键的值而不是地址。

3>MyIsam 引擎的介绍：

1. 特点：不提供对事务的支持，也不支持行级锁和外键，因此 insert 或 update 数据时需要锁定整个表，效率会低一些。MyIsam 存储引擎独立于操作系统，可以在 Windows 或 Linux 系统上使用，但是不能再表损坏后主动恢复数据。

2.适用场景：MyIsam 强调快速读取操作，由于存储了表的行数，查询时不需要对全表进行扫描。

3. 索引结构：采用 B+ tree 来存储数据，他的指针指向的是键值的地址，地址存储的是数据，B+ Tree 的数据域存储的内容为实际数据的地址，它的索引和实际数据是分开的，称之为非聚集索引。

2>InnoDB 与 MyIsam 的区别：

1. 事务：InnoDB 支持事务与外键，MyIsam 不支持。

2. 性能：MyIsam 强调的是性能，其执行速度比 InnoDB 快。

3. 行数保存：InnoDB 中不保存表的具体函数，执行 select count () from table 要扫描一遍整个表，而 MyIsam 只需读出行数即可。但包含 where 条件的都一样。

4. 索引存储：对于 auto_increment 类型的字段，InnoDB 中必须包含该字段的索引，但是在 Myisam 中可以和其他字段一起建立联合索引。Myisam 支持全文索引，压缩索引，InnoDB 不支持。Myisam 的索引和数据是分开的，并且索引是有压缩的，内存使用率就对应提高了不少，能加载更多索引，而 InnoDB 是索引和数据紧密捆绑的，没有使用压缩，从而会造成 InnoDB 以 Myisam 大不少。

5. 服务器数据备份：InnoDB 必须导出 SQL 来备份。而 Myisam 可使用 LOAD TABLE FROM MASTER 备份。Myisam 对应错误编码导致的数据恢复速度快，他的数据是以文件形式存储的，所以在跨平台的数据转移会很方便。InnoDB 是拷贝数据文件，备份 binlog，或者用 mysqldump，在数据量几个 G 就比较麻烦。

6. 锁的支持：Myisam 只支持表索。InnoDB 支持表锁和行锁，可大大提高并发度，但是 InnoDB 的行锁只在 where 的主键有效时才有用，否则也会执行表锁。

四． Redis 入门：

(1) mongoDB 与 Redis 的区别：

1>内存管理机制上：Redis 数据全部存在内存，定期写入磁盘，当内存不够时，可以选择指定的 LRU 算法删除数据。MongoDB 数据存在内存，由 linux 系统 mmap 实现，当内存不够时，只将热点数据放入内存，其他数据存在磁盘。

2>支持的数据结构上：Redis 支持的数据结构丰富，包括 hash，set，list。MongoDB 数据结构比较单一，但是支持丰富的数据表达，索引，最类似关系型数据库，支持的查询语言非常丰富。

(2) memcache 与 Redis 的区别：

1>数据类型：Redis 数据类型丰富，支持 set liset 等类型；memcache 支持简单数据类型，需要客户端自己处理复杂对象。

2>持久性：Redis 支持数据落地持久化存储；memcache 不支持数据持久存储。

3>分布式存储：Redis 支持 master-slave 复制模式；memcache 可以使用一致性 hash 做分布式。

4>value 大小不同：memcache 是一个内存缓存，key 的长度小于 250 字符，单个 item 存储要小于 1M，不适合虚拟机使用。

5>数据一致性不同：redis 使用的是单线程模型，保证了数据按顺序提交；memcache 需要使用 cas 保证数据一致性，CAS 是一个确保并发一致性的机制，属于乐观锁范畴；原理很简单；拿版本号，操作，对比版本号，如果一致就操作，不一致就放弃任何操作。

6>CPU 利用：Redis 单线程模型只能使用一个 CPU，可以开启多个 Redis 进程。

(3) Redis 的数据类型与底层实现：

1>字符串：整数型，embstr 编码的简单动态字符串，简单动态字符串（SDS）

2>列表：压缩列表，双端链表

3>哈希：压缩列表，字典

4>集合：整数集合，字典

5>有序集合：压缩列表，跳跃表和字典

(4)Redis 是单线程的，但是为什么这么高效呢？

虽然 Redis 文件事情处理器以单线程方式运行，但是通过 I/O 多路复用程序来监听多个套接字，文件事件处理器既实现了高性能的网络通信模型，又可以很好地与 Redis 服务器中其他同样以单线程运行的模块进行对接，这还保持了 Redis 内部单线程设计的简单性。

(5) Redis 的定时机制怎么实现的？

Redis 服务器是一个事件驱动程序，服务器需要处理一下两类事件：文件事件（服务器对套接字操作的抽象）和时间事件（服务器对定时操作抽象）。Redis 的定时机制就是借助实践事件实现的。

一个时间事件主要由以下三个属性组成：id:时间事件标识号；when:记录时间事件的到达事件；timeproc:时间事件处理器，当时间事件到达时，服务器就会调用相应的处理器来处理时间，一个时间事件根据时间事件处理器的返回值来判断是定时事件还是周期性事件。

(6) Redis 的 rehash 与渐进 rehash 流程：

因为 Redis 是单线程，当 K 很多时，如果一次性将键值对全部 rehash，庞大的计算量会影响服务器性能，甚至可能会导致服务器在一段时间内停止服务。不可能一步完成整个 rehash 操作，所以 Redis 是分多次，渐进式的 rehash。渐进性哈希分为两种：

1> 操作 Redis 时，额外做一步 rehash:

对 Redis 做读取，插入，删除等操作时，会把位于 table[dict->rehashidx] 位置的链表移动到新的 dictht 中，然后把 rehashidx 做加一操作，移动到后面一个槽位。

2> 后台定时任务调用 rehash:

后台定时任务 rehash 调用链，同时可以通过 server.hz 控制 rehash 调用频率。

第五章． 操作系统网络编程实践（API）

一． 系统调度（进程线程）API

(1) fork 函数：

1> 函数形式: pid_t fork(void)

2> 功能:成功调用 fork()会创建一个新的进程,他几乎与调用 fork()的进程一模一样,这两个进程都会继续运行。在子进程中,成功的 fork()调用会返回 0.在父进程中 fork()返回子进程的 pid.如果出现错误,fork()返回一个负值.

3> 父进程与子进程之间的区别具体如下:

1. Fork()的返回值不同.
2. 进程 ID 不同.
3. 这两个进程的父进程不同.
4. 子进程的 tms_utime,tms_stime 等值设置为 0.

5. 子进程不继承父进程设置的文件锁.

6. 子进程的未处理闹钟被清楚,未处理的信号集设置为空集.

4> fork 失败的常见原因:

1. 系统中已经有太多的进程.

2. 该系统用户 ID 的进程总数超过了系统限制.

5> fork 常见的两种用法:

1. 一个父进程希望复制自己,使父进程和子进程同时执行不同的代码段,这个在网络编程中是较为常见的.

2. 一个进程要执行一个不同的程序,例如 shell,这种情况下使用 fork,待其返回后调用 exec.

(2) vfork 函数:

1> 函数形式: pid_t vfork(void)

2> vfork 除了子进程必须立刻执行一次对 exec 的系统调用,或者调用_exit()退出,其他的结果与 fork()产生的结果是一样.vfork()会挂起父进程直到子进程终止或者运行一个新的可执行文件的映像.按照这样的方式,vfork()避免了地址空间的按页复制.

(3) fork()与 vfork 的区别:

1> fork()的子进程拷贝父进程的数据段和代码段;vfork()的子进程与父进程共享数据段.

2> fork()的父进程的执行次序不确定;vfork()保证子进程先运行,在调用 exec 或 exit 之前与父进程数据是共享的,在它调用 exec 或者 exit 之后父进程才可能调度运行.

3> vfork()要保证子进程先运行,在它调用 exec 或者 exit 之后父进程才可能被调度运行.如果在调用这两个函数之前子进程依赖父进程进一步动作,则会导致死锁.

4> 当需要改变共享数据段中变量的值,则拷贝父进程.

(4) 进程终止相关的函数

1> 程序正常终止情况:使用 return;调用 exit(),_exit(),_Exit();最后一个线程从启动例程返回,该进程以终止状态 0 返回;进程的最后一个线程调用 pthread_exit(),进程终止状态返回 0.

2> 程序异常终止的情况:调用 abort(),产生 SIGABORT 信号等等

3> 子进程正常或者异常终止时会向父进程发送 SIGCHLD 信号.

4> wait()函数:等待子进程终止,会阻塞父进程

5> waitpid()函数:可等待一个特定的进程,可以设置为非阻塞.

(5) Exec 系列函数:

1> 作用:exec 函数用磁盘上一个新的函数替换了当前进程的正文段,数据段,堆段和栈段.

2> sysem 函数基于 fork,waitpid,exec 函数.

二． 进程/线程间通信（IPC）API

(1) 进程间通信

1> 管道 pipe:

使用函数:int pipe(int fd[2]);

fd[0]和 fd[1]以读/写的方式打开

2> 命名管道 FIFO:

使用函数:int mkfifo(const char *path, mode_t mode)或

int mkfifoat(int fd, const char *path, mode_t mode)

3> 消息队列:

msgget 用于创建或打开一个现有队列.

msgsnd 将新消息添加到队列尾端

4> 信号量<用于为多个进程提供对共享数据的访问>:

5> 共享内存:

int shmget(key_t key , size_t size, int flag).

(3) 线程间同步

1> 互斥量的基本函数:

1. Pthread_mutex_ini
2. Pthread_mutex_destroy
3. Pthread_mutex_lock
4. Pthread_mutex_trylock
5. Pthread_mutex_unlock

2> 读写锁的基本函数:

1. Pthread_rwlock_init
2. Pthread_rwlock_destroy
3. Pthread_rwlock_rdlock
4. Pthread_rwlock_wrlock
5. Pthread_rwlock_unlock

3> 条件变量,自旋锁

4> 屏蔽:pthread_join:允许一个线程等待,直到另一个线程退出.

二． 内存调度与文件系统 API

三． 输入输出（I/O）管理 API

(1) 阻塞 I/O 与非阻塞 I/O

1. 使用 open,read,write 等函数,通过修改标志确认其会不会阻塞.

(2) 信号驱动式 IO

(3) I/O 多路复用

1. 主要函数 select,pselect,poll,epoll:

2. 都可以实现一个进程阻塞在多个 IO 描述符上,当一个描述符可用时,即对该描述符进行处理,系统需要主动查询描述符.

3. pselect 相对 select 而言,其等待的时间精度提高到微秒,且其可以选择屏蔽某些信号中断.

4. **select 和 poll 的一些区别**：1. poll 的描述符是链式结构，select 的描述符是数组，故 poll 没有最大的描述符限制，而 select 最大限制为 1024。2.每次调用 select 都要把描述符从内核拷贝过来。3.select 和 poll 都是轮询。4. poll 有一个特点是水平触发，也就是通知程序 fd 就绪后，这次没有被处理，那么下次 poll 的时候会再次通知同个 fd 已经就绪

5. **epoll** 是一个 epoll 监听一个描述符，被动触发的方式产生，回调函数，epoll 也没有描述符限制。epoll 中的 **ET 模式（水平触发模式）：**内核告诉一个内核描述符是否就绪，如果未做任何操作，则会一直通知。**LT 模式（边缘触发模式）：**只在变化的那一刻触发。

(4) 异步 IO

1.同步 IO 是 CPU 会一直等待数据传输完成。**2.**异步 IO 是 CPU 不会等待数据的传输完成。

四. 网络编程（SocKet）API

(1) TCP 中常见形式：

客户端:socket()->connect()->write()->read()->close()

服务端:socket()->bind()->listen()->accept()->read()->write()->read()->close()

1> **socket():**用于新建一个套接字，并设置 IP 协议与传输层协议。

2> TCP 三次握手发生在客户端调用 **connect()**函数的过程中。

3> TCP 四次挥手起始于客户端调用 **close()**函数时，服务器最终有一个 **read()**调用获取客户端的 **Ack**。

(2) UDP 中的常见形式:

客户端: socket()->sendto()->recvfrom()->close()

服务端: socket()->bind()->recvfrom()->sendto()

1> UDP 是非连接的协议，因此客户端直接对服务器发送数据即可。

2> 因此一般说 UDP 服务器是迭代的，TCP 服务器是并发的。