目录

[C++基础知识 2](#_Toc130143701)

[C++面向对象三大特性 2](#_Toc130143702)

[C++和C的区别 2](#_Toc130143703)

[联合体union 3](#_Toc130143704)

[结构体和联合体的区别 3](#_Toc130143705)

[struct和class的区别 3](#_Toc130143706)

[C++从代码到可执行二进制文件的过程 3](#_Toc130143707)

[静态链接和动态链接 4](#_Toc130143708)

[static关键字的作用 4](#_Toc130143709)

[构造函数是否可以为静态 5](#_Toc130143710)

[什么是野指针？怎么产生的，如何避免？ 5](#_Toc130143711)

[内联函数（inline）和宏函数（define）的区别 6](#_Toc130143712)

[const和define的区别 6](#_Toc130143713)

[new和malloc的区别 7](#_Toc130143714)

[malloc底层实现 7](#_Toc130143715)

[new底层实现 8](#_Toc130143716)

[C++有几种传参方式，之间的区别是什么？ 8](#_Toc130143717)

[多态 8](#_Toc130143718)

[Map 8](#_Toc130143719)

[虚函数的应用场景 8](#_Toc130143720)

[虚函数和纯虚函数的区别 9](#_Toc130143721)

[虚继承和多重继承 9](#_Toc130143722)

[链表排序通常使用哪些方式 9](#_Toc130143723)

[什么时候会实例化类模板 9](#_Toc130143724)

[数据结构 9](#_Toc130143725)

[Bfs及dfs及其实现 9](#_Toc130143726)

[循环和递归的优点和缺点 10](#_Toc130143727)

[单链表和循环链表的优缺点和区别 10](#_Toc130143728)

[对一个链表排序使用哪些方法 11](#_Toc130143729)

[求两个链表的交集 11](#_Toc130143730)

[操作体统&Linux 11](#_Toc130143731)

[处理死锁的策略、死锁的四个必要条件 11](#_Toc130143732)

[什么是同步（同步与一步，阻塞与非阻塞） 12](#_Toc130143733)

[进程和线程 12](#_Toc130143734)

[Linux基本指令 12](#_Toc130143735)

[软连接和硬链接的区别，建立方式 12](#_Toc130143736)

[Linux进程间的通信方式（pv+三种+socket） 12](#_Toc130143737)

[Makefile 13](#_Toc130143738)

[系统调用 13](#_Toc130143739)

[系统调用和库函数的区别 13](#_Toc130143740)

[文件描述符 14](#_Toc130143741)

[阻塞和非阻塞 14](#_Toc130143742)

[多线程&网络 15](#_Toc130143743)

[多线程 15](#_Toc130143744)

[OSI七层模型 15](#_Toc130143745)

[UDP优点 15](#_Toc130143746)

[TCP怎样实现可靠传输 15](#_Toc130143747)

[TCP和UDP的区别 15](#_Toc130143748)

[对于开发来说TCP的三次握手的过程 15](#_Toc130143749)

[数据库 15](#_Toc130143750)

[一个简单的sql查询 15](#_Toc130143751)

[视图和表的区别 16](#_Toc130143752)

# C++基础知识

## C++面向对象三大特性

面向对象：面向对象是一种编程思想，把一切对象看成是一个个对象，把这些对象拥有的属性变量和操作这些属性变量的函数打包成一个类来表示，将数据和函数绑到一起，进行封装，这样能够更**快速**的开发程序，**减少了重复代码**的重写过程

面向过程：根据业务逻辑从上到下编写程序

面向对象的三大特性是：封装、继承、多态

封装：将数据和操作数据的方法进行有机结合，隐藏对象的属性和实现细节，仅对外公开接口来和对象交互。开放一些成员函数对成员进行合理访问

继承：可以使用现有类的所有功能，并在无需重新编写原有类的情况下将这些功能拓展

## C++和C的区别

1.C++兼容C语言，又有许多新特性，如引用、智能指针、auto变量

2.C++面向对象，C语言面向过程

3.C语言有一些不安全的语言特性，如指针使用的潜在危险、内存泄漏、强制转换的不确定性等，C++增加了const常量、引用、cast转换、智能指针、try-catch等来改善。

4.C++的可复用性更高，引入了模板的概念，实现了方便开发的STL标准模板库。

## 联合体union

union 叫共用体，又叫联合、联合体。

“联合体”是一种特殊的类，也是一种构造类型的数据结构。

在一个“联合体”内能够定义多种不同的数据类型。

一个被说明为该“联合体”类型的变量中。同意装入该“联合体”所定义的不论什么一种数据。这些数据共享同一段内存，以达到**节省空间**的目的。

意思就是说联合体内部的结构体的首地址和无符号整形数据的首地址是相同的

## 结构体和联合体的区别

## struct和class的区别

1. 一般来说，struct是用于描述一个数据结构，class是对一个对象数据的封装
2. struct的默认访问权限是public，class的默认访问权限是private
3. struct默认是公有继承，class默认是私有继承
4. class关键字可以用于定义模板参数，struct不能用于定义模板参数

## C++从代码到可执行二进制文件的过程

有四个过程：预编译、编译、汇编、链接

1. 预编译：
   1. 展开宏定义
   2. 处理条件预编译指令，如#if、#ifdef
   3. 处理#include预编译指令，将被包含的文件插入到该预编译指令的位置
   4. 过滤注释
   5. 添加行号和文件名标识
2. 编译
   1. 词法分析：将源代码的字符序列分割成一系列的记号
   2. 语法分析：将记号进行语法分析，生成语法树
   3. 语义分析：判断表达式是否有意义
   4. 代码优化
   5. 生成目标代码：生成汇编代码
   6. 目标代码优化
3. 汇编：将汇编代码转变成及其可以执行的指令
4. 链接：将不同源文件生成的目标文件进行链接，从而形成可以执行的程序。链接可以分为静态链接和动态链接。

## 静态链接和动态链接

1. 静态链接：在链接的时候就已经把要调用的函数或者过程链接到了可执行文件中，就算再将静态库删除，也不会影响可执行程序的执行；生成的静态链接库，Windows下以.lib为后缀，Linux下以.a为后缀
2. 动态链接：在链接的时候，没有把调用的函数代码链接进去，而是在执行的过程中，再去找要链接的函数，生成的可执行文件中没有函数代码，只包含函数的重定位信息，所以当删除动态库后，可执行文件就不能运行。生成的动态链接库，Windows下以.dll为后缀，Linux下以.so为后缀。

* 静态链接：由链接器在链接时将库的内容加入到可执行程序中
  + 优点：
    - 对运行环境的依赖性较小，具有较好的兼容性。
  + 缺点：
    - 生成的程序比较大，需要更多的系统资源，在装入内存时需要耗费更多的时间。静态库在内存中存在多份拷贝，导致空间浪费
    - 库函数更新，必须重新编译应用程序
* 动态链接：连接器在链接时仅建立与所需库函数的链接关系，在程序运行时，才将所需要的资源调入
  + 优点：
    - 在需要的时候才会调入相应资源
    - 简化程序升级，有较小的程序体积
    - 实现进程之间的资源共享
  + 缺点：
    - 以来动态库，不能独立运行
    - 动态库依赖版本问题严重

## static关键字的作用

全局或静态变量在对象首次用到时进行构造

1. 修饰局部变量
   1. 在全局数据区分配内存（局部变量在栈区）
   2. 静态局部变量在程序执行到该变量的声明处，**首次初始化**，以后的函数调用不再进行初始化（局部变量每次函数调用都需要初始化）
   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.（局部变量不会被自动初始化）
   4. 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，不能在函数体外使用。（局部变量在栈区，函数结束后立即释放内存）
2. 修饰全局变量：
   1. 定义在函数体外，用于修饰全局变量，该变量只在本文件可见
3. 修饰函数：
   1. 静态函数不能被其他文件使用
   2. 其他文件中定义相同名字的函数不会发生冲突
4. 修饰类中数据成员：
   1. 存储在全局数据区
   2. 静态数据成员被当作是类的成员，由该类的所有对象共享访问，对该类的多个对象来说，静态数据成员只分配一次内存
   3. 静态数据成员定义时要分配内存空间，所以不能在类中定义
5. 修饰类中成员函数
   1. 静态成员函数只能访问静态成员变量和静态成员函数，不能访问非静态成员
   2. 非静态成员函数可以访问静态成员函数
   3. 当调用一个对象的非静态成员函数时，系统会把该对象的起始地址赋给成员函数的this指针。而静态成员函数不属于对象，因此C++规定静态成员函数没有this指针。既然他没有指向某一对象，也就无法对一个对象中的非静态成员进行访问

## 构造函数是否可以为静态

1. 与静态函数相似，只能访问静态成员
2. 与构造函数相似，具有初始化作用，没有返回值；不同的是**只执行一次**。
3. 用于初始化静态成员，或者是用于执行只需要执行一次的操作；
4. 静态构造函数**没有修饰符修饰**(public,private),也没有参数。
5. 在程序中，用户无法控制何时执行静态构造函数，静态构造函数不是程序员调用的，是由.net 框架在合适的时机调用的。
6. 静态函数的调用时机，创建第一个实例对象或者引用任何静态变量之前。
7. 静态构造函数中不能实例化实例变量。（变量可以分为类级别和实例级别的变量，其中类级别的有static关键字修饰）。
8. 一个类中只能有一个静态构造函数。
9. 无参数的静态构造函数和无参数的构造函数是可以并存的。因为他们一个属于类级别，一个属于实例级别，并不冲突。
10. 就像如果没有在类中写构造函数，那么框架会生成一个构造函数，那么如果定义了静态成员变量，但是又没有定义静态构造函数，那么框架也会帮助生成一个静态构造函数来让框架自身来调用。
11. 线程安全

## 什么是野指针？怎么产生的，如何避免？

1. 概念：野指针就是指针指向的位置是不可知的
2. 产生原因：释放内存后，指针不及时置空，依然指向了该内存，那么可能出现非法访问的错误
3. 避免办法：
   1. 初始化指针置NULL
   2. 申请内存后判空
   3. 指针内存释放后置NULL
   4. 使用智能指针

## 内联函数（inline）和宏函数（define）的区别

1. 本质上：
   1. 宏定义不是函数，使用起来像函数。预处理器用复制宏代码的方式代替函数的调用，省去了函数压栈出栈的操作，提高了效率；
   2. 内联函数本质上是个函数，一般用于函数体代码比较简单的函数，不能包含复杂的控制语句。编译器会在每处调用内联函数的地方直接把内联函数的内容展开，这样可以省去函数调用的开销，提高效率
2. 特点：
   1. 宏函数在预编译的时候把所有的宏名用宏体替换；
   2. 内联函数是在编译阶段进行代码插入。一般用于比较小的、并且频繁调用的函数。如果函数的定义体较长或复杂，编译器会自动取消不符合要求的内联
3. 安全性：
   1. 宏定义没有类型检查，无论对错都是直接替换
   2. 内联函数在编译的时候会进行类型检查，内联函数满足函数的性质，有返回值和参数列表

文本, 信件

描述已自动生成

## const和define的区别

const用于定义常量；define用于定义宏，也可以定义常量。都用于定义常量时，它们的区别有：

1. const生效于编译阶段，define生效于预编译阶段
2. const定义的常量存储在内存中，需要额外的内存空间；define定义的常量，运行时是直接的操作数，并不会存放在内存中
3. const定义的常量带有类型，define定义的常量没有类型。因此define定义常量不利于类型检查。

## new和malloc的区别

1. new/delete是操作符，需要编译器支持。malloc/free是库函数，需要头文件支持。
2. 使用new操作符申请内存无需指定内存块大小，编译器会根据类型信息自行计算；malloc需要显式指出申请的内存块的大小
3. new操作符内存分配成功时，返回的是对象类型的指针，类型严格与对象匹配，不用类型转换，所以new是符合类型安全的操作符；malloc内存分配成功，返回值类型是void\*，需要强制类型转换；
4. new内存分配失败时，会抛出bac\_alloc异常。malloc分配内存失败时返回NULL。
5. new会先调用operator new函数，申请足够的内存（通常底层使用malloc实现）。然后调用类型的**构造函数**，初始化成员变量，最后返回自定义类型指针。delete先调用**析构函数**，然后调用operator delete函数释放内存（通常底层使用free实现）。malloc/free是库函数，只能动态的申请和释放内存，无法强制要求其做自定义类型对象构造和析构工作。
6. C++允许重载new/delete操作符，而malloc不允许重载。
7. new操作符从自由存储区（free store）上为对象动态分配内存空间，而malloc函数从堆上动态分配内存。自由存储区是C++基于new操作符的一个抽象概念，凡是通过new操作符进行内存申请，该内存即为自由存储区。而堆是操作系统中的术语，是操作系统所维护的一块特殊内存，用于程序的内存动态分配，C语言使用malloc从堆上分配内存，使用free释放已分配的对应内存。自由存储区不等于堆。

## malloc底层实现

从操作系统角度看，进程分配内存有两种方式，分别由两个系统调用完成：brk和mmap

* brk是将数据段的最高地址指针往高地址推
* mmap是在进程的虚拟地址空间中，找一块空闲的虚拟内存块

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。

* 当开辟的空间小于128k时，调用brk（）函数
* 当开辟的空间大于128k时，调用mmap（）

brk分配的内存需要等到高地址内存释放以后才能释放（例如，在B释放之前，A是不可能释放的，这就是内存碎片产生的原因，什么时候紧缩看下面），而mmap分配的内存可以单独释放。

malloc采用的是内存池的管理方式，以减少内存碎片。先申请大块内存作为堆区，然后将堆区分为多个内存块。当用户申请内存时，直接从堆区分配一块合适的空闲块。

## new底层实现

1. 创建一个新兑现
2. 将构造函数的作用域赋值给新对象（因此this指向了这个对象）
3. 执行构造函数的代码（为这个新对象添加属性）
4. 返回新对象指针

## C++有几种传参方式，之间的区别是什么？

1. 值传递
2. 引用传递
3. 指针传递

值传递用于对象时，整个对象会拷贝一个副本，这样效率低；而引用传递 用于对象时，不发生拷贝行为，只是绑定对象，更高效；指针传递同理，但不如引用传递安全。不存在空引用，并且引用一旦被初始化为指向一个对象，它就不能被改变为另一个对象的引用，显得很安全。

## 多态

## Map

map是STL的一个关联容器，它提供一对一的hash。

第一个可以称为关键字(key)，每个关键字只能在map中出现一次；

第二个可能称为该关键字的值(value)；

map以模板(泛型)方式实现，可以存储任意类型的数据，包括使用者自定义的数据类型。Map主要用于资料一对一映射(one-to-one)的情況，map內部的实现自建一颗红黑树，这颗树具有对数据自动排序的功能。在map内部所有的数据都是有序的，

## 虚函数的应用场景

## 虚函数和纯虚函数的区别

## 虚继承和多重继承

一个类有多个基类,这样的继承关系称为多继承;

派生列表一个基类只能出现一次，但有些情况会间接继承一个类多次，比如通过两个直接基类分别继承同一个间接基类。默认情况下派生类含有继承链上每个类对应的子部分，所以如果某个类在派生过程中出现了多次，则派生类将包含该类的多个子对象

上述重复继承有时候会出问题，C++通过虚继承解决该问题，共享的基类子对象成为虚基类。虚继承让基类不管出现几次，都只包含唯一一个共享的虚基类子对象

提供虚继承机制，防止[类继承](https://so.csdn.net/so/search?q=%E7%B1%BB%E7%BB%A7%E6%89%BF&spm=1001.2101.3001.7020)关系中成员访问的二义性。

## 链表排序通常使用哪些方式

## 什么时候会实例化类模板

有没有发生实例化，可以通过看是不是需要提供存储空间信息。比如引用不分配内存，而指针分配的内存是固定的，声明也不需要分配内存，它们都不会发生实例化。而定义了一个对象，或者是函数值传递时，很明显需要分配内存空间，必须得知道类的实现方式，所以需要实例化。

# 数据结构

## Bfs及dfs及其实现

bfs 是 Breadth-First Search 的缩写，称为广度优先搜索，或宽度优先搜索。

步骤 1：从源点出发，访问源点的邻居结点，将邻居节点依次放入队列中，并标记为已访问；

步骤 2：取出队列中的邻居结点，依次访问每个节点未被访问的邻居节点；

步骤 3：将邻居节点依次放入队列中，并标记为已访问；

步骤 4：重复步骤 2~3 直到访问到目标节点或所有节点都标记为已访问

dfs 是 Depth-First-Search 的缩写，称为深度优先搜索。

步骤 1：从源点出发，访问源点的某个邻居节点，将其放入栈中，并标记为已访问；

步骤 2：从栈中取出一个节点，访问该节点的未被访问的邻居节点；

步骤 3：将邻居节点放入栈中，并标记为已访问；

步骤 4：重复步骤 2 ~ 3，直到访问到目标节点或所有节点都标记为已访问；

## 循环和递归的优点和缺点

递归

优点：代码更简洁清晰，可读性更好

递归的话函数调用是有开销的，而且递归的次数受堆栈大小的限制。

缺点：

时间和空间消耗比较大。每一次函数调用都需要在内存栈中分配空间以保存参数，返回地址以及临时变量，而且往栈里面压入数据和弹出都需要时间。

另外递归会有重复的计算。递归本质是把一个问题分解为多个问题，如果这多个问题存在重复计算，有时候会随着n成指数增长。斐波那契的递归就是一个例子。

递归还有栈溢出的问题，每个进程的栈容量是有限的。由于递归需要系统堆栈，所以空间消耗要比非递归代码要大很多。而且，如果递归深度太大，可能系统撑不住。

循环

缺点：

不容易理解，编写复杂问题时困难。代码可读性不如递归.

优点：

优点是效率高。运行时间只因循环次数增加而增加，没什么额外开销。空间上没有什么增加。

## 单链表和循环链表的优缺点和区别

在一个循环链表中, 首节点和末节点被连接在一起。这种方式在单向和双向链表中皆可实现。要转换一个循环链表，你开始于任意一个节点然后沿着列表的任一方向直到返回开始的节点。循环链表可以被视为"无头无尾"。

循环链表中第一个节点之前就是最后一个节点，反之亦然。循环链表的无边界使得在这样的链表上设计算法会比普通链表更加容易。对于新加入的节点应该是在第一个节点之前还是最后一个节点之后可以根据实际要求灵活处理，区别不大。

单向链表，它包含两个域，一个信息域和一个指针域。这个链接指向表中的下一个节点，而最后一个节点则指向一个空值NULL。

单向链表只可向一个方向遍历。

查找一个节点的时候需要从第一个节点开始每次访问下一个节点，一直访问到需要的位置

## 对一个链表排序使用哪些方法

* 插入排序的思路主要是，将链表分为两条，一条主链表等待被插入，一条从链表逐个插入到主链表中。
* 具体操作需要对两条链表都进行遍历，如果从链表中的某个节点没有找到位置可以插入，需要补在主链表的末尾。
* 所以我们需要两次嵌套的for循环进行遍历，需要一个flag记录节点是否成功插入

## 求两个链表的交集

# 操作体统&Linux

## 处理死锁的策略、死锁的四个必要条件

**死锁（Deadlock）：**是指两个或两个以上的进程在执行过程中，因争夺资源而造成的一种互相等待的现象，若无外力作用，它们都将无法推进下去。称此时系统处于死锁状态或系统产生了死锁。  
称这些永远在互相等待的进程为死锁进程。  
所占用的资源或者需要它们进行某种合作的其它进程就会相继陷入死锁，最终可能导致整个系统处于瘫痪状态。

文本

中度可信度描述已自动生成

**死锁的四个必要条件**

**1）互斥条件：**指进程对所分配到的资源进行排它性使用，即在一段时间内某资源只由一个进程占用。如果此时还有其它进程请求资源，则请求者只能等待，直至占有资源的进程用毕释放。

**2）请求和保持条件：**指进程已经保持至少一个资源，但又提出了新的资源请求，而该资源已被其它进程占有，此时请求进程阻塞，但又对自己已获得的其它资源保持不放。

**3）不剥夺条件：**指进程已获得的资源，在未使用完之前，不能被剥夺，只能在使用完时由自己释放。

**4）环路等待条件：**指在发生死锁时，必然存在一个进程——资源的环形链，即进程集合{P0，P1，P2，···，Pn}中的P0正在等待一个P1占用的资源；P1正在等待P2占用的资源，……，Pn正在等待已被P0占用的资源。

## Linux基本指令

## 软连接和硬链接的区别，建立方式

[root@localhost ~]# ln [选项] 源文件 目标文件

* -s：建立软链接文件。如果不加 "-s" 选项，则建立硬链接文件；
* -f：强制。如果目标文件已经存在，则删除目标文件后再建立链接文件；

## Linux进程间的通信方式（pv+三种+socket）

## Makefile

一个工程中的源文件很多，其按类型、功能、模块分别放在若干个目录里，makefile定义了一系列的规则来指定，那些文件需要先编译，那些文件需要后编译，甚至进行更复杂的功能操作，因为mekefile就像shell脚本一样，其中也可以执行操作系统的命令

Makefile带来的好处就是————“自动化编译“，一旦写好，只需要一个make指令，整个工程完全自动编译，极大的提高了软件开发的效率。make是一个命令工具，是一个解释makefile中命令的工具。

make主要解决两个问题：

1. 大量代码的关系维护：

把代码维护命令以及编译命令写在makefile文件中，然后用make工具解析此文件自动执行相应命令，可实现代码的合理编译

1. 减少重复编译的时间

在改动其中一个文件的时候，能判断哪些文件被修改过，可以只对该文件进行重新编译，然后重新链接所有的目标文件，节省编译时间

## 系统调用

概念：系统调用，顾名思义，说的是操作系统提供给用户程序调用的一组“特殊”接口。用户程序可以通过这组“特殊”接口来获得操作系统内核提供的服务。

操作系统一般通过软件中断从用户态切换到内核态。

## 系统调用和库函数的区别

Linux对文件操作有两种方式：系统调用和库函数调用

库函数由两类函数组成：

1. 不需要调用系统调用：

不需要切换到内核空间即可完成函数全部功能，并且将结果反馈给应用程序，如strcpy、bzero等字符串操作

1. 需要调用系统调用

需要切换到内核空间，这类函数通过封装系统调用去实现功能，如printf、fread等。

程序中频繁的使用系统调用会降低程序的运行效率。当运行内核代码时，CPU工作在内核态，在系统调用发生前需要保存用户态的栈和内存环境，然后转入内核态工作。系统调用结束后，又要切换会用户态。这种环境的切换会消耗掉很多时间。

## 文件描述符

打开现存文件或者创建文件时，系统（内核）会返回一个文件描述符，文件描述符用来指定已经打开的文件。这个文件描述符相当于这个已打开文件的标号，文件描述符时**非负整数**，是文件的标识，操作这个文件描述符相当于操作这个描述符指定的文件。

程序运行起来后（每个进程）都有一张文件描述符的表，标准输入、标准输出、标准错误输出设备文件被打开，对应的文件描述符0、1、2记录在表中。程序运行起来后这三个文件描述符时默认打开的。

在程序运行起来后打开其他文件时，系统会返回文件描述符表中**最小**可用的文件描述符，并将文件描述符记录在表中。

## 阻塞和非阻塞

* 以读取文件的模型举例。

在发起读取文件的请求时，应用层会调用系统内核的I/O接口。

如果应用层调用的是阻塞型I/O，那么在调用之后，应用层即刻被挂起，一直处于等待数据返回的状态，知道系统内核从磁盘读取完数据并返回给应用层，应用层才用获得的数据进行接下来的操作。

如果应用层调用的是非阻塞型的I/O，那么调用后，系统内核会立即返回（虽然还没有文件内容的数据），应用层不会被挂起等待，它可以做其他任意操作。

总结来说，是阻塞还是非阻塞，关键是接口调用（发出请求）后等待数据返回时的状态。被挂起无法执行其他操作的时阻塞型的，可以立即去完成其他任务的则是非阻塞型的。

* 读常规文件是不会阻塞的，不管读多少字节，read一定会在有限的时间内返回。

从终端设备或网络读则不一定，如果从终端输入的数据没有换行符，调用read读终端设备就会阻塞，如果网络上没有接收到数据包，调用read从网络读就会阻塞，至于会阻塞多长时间是不确定的，如果一直没有数据到达就一直阻塞。

同样，写常规文件是不会阻塞的，而向终端设备或网络写则不一定。

* 阻塞与非阻塞是对于文件而言的，不是read、write等函数的属性

## 进程和程序

我们写的代码，通过编译器编译，会成为一个可执行程序，当这个可执行程序运行起来后（没有结束之前），它就成为了一个进程。

程序是存放在存储介质上的一个可执行文件，而进程是程序执行的过程。进程的状态是变化的，其包括进程的创建、调度和消亡。程序是静态的，进程是动态的。

进程是CPU分配资源的最小单位，在Linux系统中，操作系统通过进程去完成一个个任务，进程是管理事务的基本单元。

进程有自己独立的**处理环境**和**系统资源**。

## 并行和并发

* **并行**：同一时刻，有多条指令在多个处理器上同时运行
* **并发**：在同一时刻只有一条指令执行，多个进程指令快速轮换执行，使得在宏观上有多个进程同时执行的效果。在微观上不是同时执行的，只是把时间分成时间片，多个进程快速交替的执行。

## MMU

MMU是memory management unit的缩写，中文名是**内存管理单元**，它是中央处理器（CPU）中用来

* 管理虚拟存储器、物理存储器的控制线路，
* 负责虚拟地址映射为物理地址
* 提供硬件机制的内存访问授权，多用户多进程操作系统

## 进程控制块PCB

进程运行时，内核为每个进程分配一个PCB，维护进程相关信息，Linux内核的进程控制块是task\_struct结构体。

其部分内部成员：

* 进程id。系统中每个进程有唯一的id，在C语言中用pid\_t类型表示，其实是一个非负整数
* 进程的状态，有就绪、运行、阻塞等状态
* 进程切换时需要保存和恢复的一些CPU寄存器
* 描述虚拟地址空间的信息
* 描述控制终端的信息
* 当前工作目录
* umask掩码
* 文件描述符表
* 和信号相关的信息
* 用户id和组id
* 会话（session）和进程组
* 进程可以使用的资源上线

## 进程的状态

进程状态反映进程执行过程的变化，这些状态随着进程的执行和外界条件的变化而转换

在三态模型中：运行态、就绪态、阻塞态

在五态模型中：新建态、终止态、运行态、就绪态、阻塞态

* 查看用户状态：ps

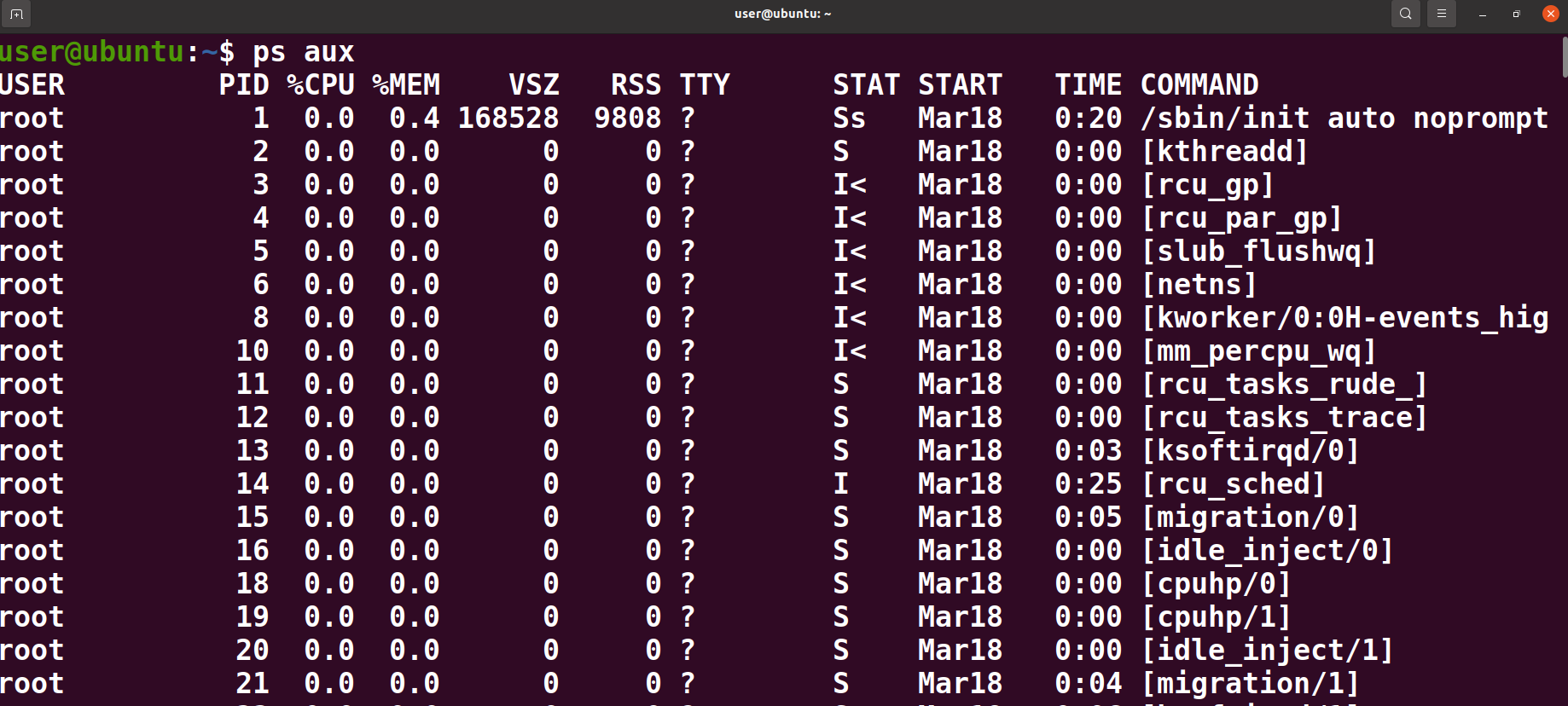
-a 显示终端上所有进程

-u 显示进程详细状态

-x 显示没有控制终端的进程

-w 显示加宽，以便于显示更多的信息

-r 只显示正在运行的进程



* 动态显示运行中的进程： top
* 杀死进程：kill pid

-9 强制结束

* 通过进程名字杀死进程 ：killall

## 进程号和相关函数

进程号PID：标识进程的一个非负整型数

父进程号PPID：任何进程都是由另一个进程创建，该进程称为被创建进程的父进程，对应的进程号称为父进程号

进程组号PGID：进程组是一个或多个进程的集合。它们之间相互关联，进程组可以接收同一终端的各种信号，关联的进程有一个进程组号。

* 获取当前进程的进程号：getpid
* 获取父进程号：getppid
* 获取进程组号：getpgid
* pid\_t fork（）； 用于从一个已存在的进程中创建一个新进程，新进程称为子进程，原进程称为父进程。

## 父子进程关系

使用fork（）函数得到的子进程是父进程的一个复制品，从父进程继承了整个进程的地址空间

子进程独有的只有它的进程号、计时器等。因此，使用fork函数的代价很大。

Linux的fork（）使用是通过**写时拷贝**实现的。写时拷贝是一种可以推迟甚至避免拷贝数据的技术。内核此时并不复制整个进程的地址空间，而是让父子进程共享同一个地址空间。只用在需要写入的时候才会复制地址空间，从而使各个进程拥有各自的地址空间。也就是说，资源的复制是在需要写入的时候才会进行，其他时候以只读方式共享。

注意：fork之后父子进程共享文件，fork产生的子进程与父进程相同的文件描述符指向相同的文件表，引用计数增加，共享文件偏移指针。

## 区分父子进程

通过fork（）函数的返回值区分父子进程。

fork()函数在子进程中返回0，在父进程中返回子进程的pid。

图示

描述已自动生成

# 多线程&网络

## 多线程

## OSI七层模型

## UDP优点

## TCP怎样实现可靠传输

## TCP和UDP的区别

## 对于开发来说TCP的三次握手的过程

# 数据库

## 一个简单的sql查询

## 视图和表的区别

**视图是虚拟的表，不占用物理空间，只存储SQL逻辑，可以从多个表中提取数据，也可以保护表的结构和数据安全**[1](https://www.vsdiffer.com/table-vs-view.html),[2](https://www.php.cn/mysql-tutorials-475547.html),[3](https://zhidao.baidu.com/question/95004720.html),[4](https://blog.csdn.net/qq_38737586/article/details/100112132),[5](https://blog.csdn.net/a1097304791/article/details/115057518)。表是实际的表，占用物理空间，保存具体的数据，有有限的列和无限的行，需要多个表来存储关联的数据和记录[1](https://www.vsdiffer.com/table-vs-view.html),[2](https://www.php.cn/mysql-tutorials-475547.html),[4](https://blog.csdn.net/qq_38737586/article/details/100112132)。视图和表的操作方式相同，但视图只能用创建的语句来修改，而表可以及时对其进行修改[2](https://www.php.cn/mysql-tutorials-475547.html),[3](https://zhidao.baidu.com/question/95004720.html),[4](https://blog.csdn.net/qq_38737586/article/details/100112132)