c++后端开发 资料整理学习

2021年春招 基础资料学习整理
主要内容及框架参考：超全面的后端开发C/C++面经整理分享含参考答案 简历分享
c/c++后端开发面经整理，包括C/C++语言基础，计网，数据库，linux，操作系统，场景题，智力题和hr常问题

一、c/c++语言基础

1、基础

1、指针和引用的区别

- 指针是一个新的变量，指向另一个变量的地址，我们可以通过这个地址来修改该另一个变量；
  引用是一个别名，对引用的操作就是对变量本身进行操作；
- 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做修改；
  而使用引用可以直接对参数进行修改
- 指针的大小一般是四个字节
  引用的大小取决于被引用对象的大小（指的是使用sizeof运算符得到的结果，引用本质上还是使用指针，因此所占内存和指针是一样的）
- 指针可以为空
  引用不行

2、在函数传递参数时，什么时候用指针，什么时候用引用?

- **需要返回函数内局部变量的地址的内存时使用指针。**使用指针传参需要开辟内存，用完要记得释放指针，不然会有内存泄漏。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（如递归）的时候使用引用。使用引用不需要创建临时变量，开销更小；
- 类对象作为参数传递时使用引用，这是C++类对象传递的标准方式。

3、堆和栈有什么区别

- 定义上：堆是由new和malloc开辟的一块内存，由程序员手动管理；
  栈是编译器自动管理的内存，存放函数的参数和局部变量。
- 堆空间因为会有频繁的分配释放操作，会产生内存碎片。
- 堆的生长空间向上，地址越来越大；
  栈的生长空间向下，地址越来越小。

4、堆快一些还是栈快一些?

- 栈快一些。

- 操作系统在底层会对栈提供支持，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，所以栈的效率比价快也比较高。
- 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的内存大小，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。

5、new和delete是如何实现的，new和malloc的异同

- 在new一个对象的时候，首先会调用malloc为对象分配内存空间，然后调用对象的构造函数。
- delete会调用对象的析构函数，然后调用free回收内存。
- new 和 malloc都会分配空间，但是new还会根据调用对象的构造函数进行初始化，malloc需要给定空间大小，而new只需要对象名。

5.1、linux下brk、mmap、malloc和new的区别

- brk是系统调用，主要工作是实现虚拟内存到内存的映射，可以让进程的堆指针增长一定的大小，逻辑上消耗掉一块虚拟地址空间，malloc向OS获取的内存大小比较小时，将直接通过brk调用获取虚拟地址。

- mmap是系统调用，也是实现虚拟内存到内存的映射，可以让进程的虚拟地址区间切分出一块指定大小的虚拟地址空间vma_struct，一个进程的所有动态库文件.so的加载，都需要通过mmap系统调用映射指定大小的虚拟地址区间，被mmap映射返回的虚拟地址，逻辑上被消耗了，直到用户进程调用unmap，会回收回来。malloc向系统获取比较大的内存时，会通过mmap直接映射一块虚拟地址区间。

- malloc是C语言标准库中的函数，主要用于申请动态内存的分配，其原理是当堆内存不够时，通过brk/mmap等系统调用向内核申请进程的虚拟地址区间，如果堆内部的内存能满足malloc调用，则直接从堆里获取地址块返回。

- new是C++内置操作符，用于申请动态内存的分配，并同时进行初始化操作。其实现会调用malloc，对于基本类型变量，它只是增加了一个cookie结构, 比如需要new的对象大小是object_size, 则事实上调用 malloc 的参数是 object_size + cookie， 这个cookie 结构存放的信息包括对象大小，对象前后会包含两个用于检测内存溢出的变量，所有new申请的cookie块会链接成双向链表。 对于自定义类型，new会先申请上述的大小空间，然后调用自定义类型的构造函数，对object所在空间进行构造。

6、既然有了malloc/free，为什么还要new/delete?

详见：c++中有了malloc/free，为什么还需要new/delete?

- malloc/free是c/c++中的标准库函数，new/delete是c++中的运算符。它们都用于申请动态内存和释放内存。
- 对于非内部数据对象（如类对象），只用malloc/free无法满足动态对象的要求。这是因为对象在创建的同时需要自动执行构造函数，对象在消亡之前要自动执行析构函数，而由于malloc/free是库函数而不是运算符，不在编译器的控制权限之内，也就不饿能自动执行构造函数和析构函数。因此，不能将执行构造函数和析构函数的任务强加给malloc/free。所以，在c++中需要一个能完成动态内存分配和初始化工作的运算符new，以及一个能完成清理和释放内存工作的运算符delete。

7、C和C++的区别

- C面向过程， c++面向对象。
  c++有封装，继承和多态的特性。封装隐藏了实现细节，使得代码模块化。继承通过子类继承父类的方法和属性，实现了代码重用。多态则是“一个接口，多个实现”，通过子类重写父类的虚函数，实现接口重用。
- C和C++内存管理的方法不一样，C使用malloc/free，C++除此之外还用new/delete
- C++中还有函数重载和引用等概念，C中没有

8、delete和delete[]区别

- delete只调用一次析构函数，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
- 调用new []之后，释放内存使用delete[]，没有指定需要析构的对象的个数，自己设计编译器的话怎么实现operator delete。
  就是申请内存时可以多分配几个字节用来存放对象个数，delete[]的时候可以先调整指针位置来获取这个值。
  《深入探索c++对象模型》中解决方法：为vec_new()所传回的每一个内存区块配置一个额外的word，然后把元素个数藏在这个word当中

9、 C++、Java的联系与区别，包括语言特性、垃圾回收、应用场景等

- C++ 和Java都是面向对象的语言，C++是编译成可执行文件直接运行的，JAVA是编译之后在JAVA虚拟机上运行的，因此JAVA有良好的跨平台特性，但是执行效率没有C++ 高。
- C++的内存管理由程序员手动管理，JAVA的内存管理是由Java虚拟机完成的，它的垃圾回收使用的是标记-回收算法
- C++有指针，Java没有指针，只有引用
- JAVA和C++都有构造函数，但是C++有析构函数但是Java没有

10、c++和python区别

1. python是一种脚本语言，是解释执行的，而C++是编译语言，是需要编译后在特定平台运行的。python可以很方便的跨平台，但是效率没有C++高。
2. python使用缩进来区分不同的代码块，C++使用花括号来区分
3. C++中需要事先定义变量的类型，而python不需要，python的基本数据类型只有数字，布尔值，字符串，列表，元组等等
4. python的库函数比C++的多，调用起来很方便

11、Struct和class的区别

- struct的成员的访问权限默认是public，而class的成员默认是private；
- struct的继承默认是publilc继承，而class的默认继承是private继承；
- class可以在作为模板，而struct不可以

12、define和const的联系与区别

- 联系：他们都是定义常量的一种方法。
- 区别：
  - define定义的变量没有类型，只是进行简单的替换，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。
    - define定义的常量实在预处理阶段进行替换，而const在编译阶段确定它的值。
    - define不会进行安全类型检查，而const会进行类型安全检查，安全性更高；
    - const可以定义函数而define不可以。

13、在c++中const的用法（定义、用途）

- const修饰类的成员变量目标是常量不能被修改；
- const修饰类的成员函数，表示该函数不会修改类的数据成员，不会调用其他非const的成员函数。

14、C++中static的用法和意义

详见：c++中static的用法详解
C++内存空间：静态存储区、栈、堆、文字常量区、程序代码区
static的意思是静态的，用来修饰变量，函数和类成员。

- 变量：被static修饰的变量就是静态变量，它会在程序运行过程中一直存在，会被放在静态存储区。局部静态变量的作用域在函数体内，全局静态变量的作用域在这个文件内。
- 函数：被static修饰过的函数就是静态函数，静态函数只能在本文件中使用，不能被其他文件调用，也不会和其他文件中的同名函数冲突。
- 类：在类中，被static修饰的成员变量是类静态成员，这个静态成员会被类的多个对象共用。被static修饰的成员函数也属于静态成员，不是属于某个对象的，访问这个静态函数不需要引用对象名，而是通过引用类名来访问。

补充：

1. 全局静态变量：
   定义：.在全局变量之前加上关键字static，全局变量就被定义成为一个全局静态变量。
   说明：
   1）内存中的位置：静态存储区（静态存储区在整个程序运行期间都存在）
   2）初始化：未经初始化的全局静态变量会被程序自动初始化为0（自动对象的值是任意的，除非它被显示初始化）
   3）作用域：全局静态变量在声明它的文件之外是不可见的。
   全局静态变量的好处：
   1）不会被其他文件所访问，修改；
   2）其他文件中可以使用相同名字的变量，不会发生冲突。

2. 局部静态变量：
   定义：在局部变量之前加上关键字static，局部变量就被定义成为一个局部静态变量。
   说明：
   1）内存中的位置：静态存储区
   2）初始化：未经初始化的局部静态变量会被程序自动初始化为0（自动对象的值是任意的，除非他被显示初始化）
   3）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。

注意：当static用来修饰局部变量的时候，它就改变了局部变量的存储位置，从原来的栈中存放改为静态存储区。但是局部静态变量在离开作用域之后，并没有被销毁，而是仍然驻留在内存当中，直到程序结束，只不过我们不能再对它进行访问。
当static用来修饰全局变量的时候，它就改变了全局变量的作用域（在声明它的文件之外是不可见的），但是没有改变它的存放位置，还是在静态存储区中。

15、计算以下几个类的大小

复制代码

```
class A {};
int main(){
    cout<<sizeof(A)<<endl;// 输出 1;
    A a;
    cout<<sizeof(a)<<endl;// 输出 1;
    return 0;
}
```

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。

复制代码

```
class A { virtual Fun(){} };
int main(){
    cout<<sizeof(A)<<endl;// 输出 4(32位机器)/8(64位机器);
    A a;
    cout<<sizeof(a)<<endl;// 输出 4(32位机器)/8(64位机器);
    return 0;
}
```

因为有虚函数的类对象中都有一个虚函数表指针 __vptr，其大小是4字节

复制代码

```
class A { static int a; };
int main(){
    cout<<sizeof(A)<<endl;// 输出 1;
    A a;
    cout<<sizeof(a)<<endl;// 输出 1;
    return 0;
}
```

静态成员存放在静态存储区，不占用类的大小, 普通函数也不占用类大小

复制代码

```
class A { int a; };
int main(){
    cout<<sizeof(A)<<endl;// 输出 4;
    A a;
    cout<<sizeof(a)<<endl;// 输出 4;
    return 0;
```

```
}

class A { static int a; int b; };;
int main(){
    cout<<sizeof(A)<<endl;// 输出 4;
    A a;
    cout<<sizeof(a)<<endl;// 输出 4;
    return 0;
}
```

静态成员a不占用类的大小，所以类的大小就是b变量的大小 即4个字节

1.16、定义和声明的区别

声明是告诉编译器变量的类型和名字，不会为变量分配空间
定义就是对这个变量和函数进行内存分配和初始化。需要分配空间，同一个变量可以被声明多次，但是只能被定义一次

1.17、typdef和define区别

（#define是预处理命令，在预处理是执行简单的替换，不做正确性的检查
typedef是在编译时处理的，它是在自己的作用域内给已经存在的类型一个别名

1.18、被free回收的内存是立即返还给操作系统吗？为什么

- 不是的，被free回收的内存会首先被ptmalloc使用双链表保存起来，当用户下一次申请内存的时候，会尝试从这些内存中寻找合适的返回。这样就避免了频繁的系统调用，占用过多的系统资源。同时ptmalloc也会尝试对小块内存进行合并，避免过多的内存碎片。

1.19、引用作为函数参数以及返回值的好处

对比值传递，引用传参的好处：
1）在函数内部可以对此参数进行修改
2）提高函数调用和运行的效率（因为没有了传值和生成副本的时间和空间消耗）
如果函数的参数实质就是形参，不过这个形参的作用域只是在函数体内部，也就是说实参和形参是两个不同的东西，要想形参代替实参，肯定有一个值的传递。函数调用时，值的传递机制是通过“形参=实参”来对形参赋值达到传值目的，产生了一个实参的副本。即使函数内部有对参数的修改，也只是针对形参，也就是那个副本，实参不会有任何更改。函数一旦结束，形参生命也宣告终结，做出的修改一样没对任何变量产生影响。

用引用作为返回值最大的好处就是在内存中不产生被返回值的副本。
但是有以下的限制：
1）不能返回局部变量的引用。因为函数返回以后局部变量就会被销毁
2）不能返回函数内部new分配的内存的引用。虽然不存在局部变量的被动销毁问题，可对于这种情况（返回函数内部new分配内存的引用），又面临其它尴尬局面。例如，被函数返回的引用只是作为一个临时变量出现，而没有被赋予一个实际的变量，那么这个引用所指向的空间（由new分配）就无法释放，造成memory leak
3）可以返回类成员的引用，但是最好是const。因为如果其他对象可以获得该属性的非常量的引用，那么对该属性的单纯赋值就会破坏业务规则的完整性。

1.20、友元函数和友元类

详见：友元(友元函数、友元类和友元成员函数) C++

友元提供了不同类的成员函数之间、类的成员函数和一般函数之间进行数据共享的机制。通过友元，一个不同函数或者另一个类中的成员函数可以访问类中的私有成员和保护成员。友元的正确使用能提高程序的运行效率，但同时也破坏了类的封装性和数据的隐藏性，导致程序可维护性变差。

1.21、说一下volatile关键字的作用

volatile的意思是“脆弱的”，表明它修饰的变量的值十分容易被改变，所以编译器就不会对这个变量进行优化（CPU的优化是让该变量存放到CPU寄存器而不是内存），进而提供稳定的访问。每次读取volatile的变量时，系统总是会从内存中读取这个变量，并且将它的值立刻保存。

1.22、STL中的sort()算法是用什么实现的，stable_sort()呢

STL中的sort是用快速排序和插入排序结合的方式实现的，stable_sort()是归并排序。

1.23、vector会迭代器失效吗？什么情况下会迭代器失效?

vector迭代器失效的几种情况

- 会
- 当vector在插入的时候，如果原来的空间不够，会将申请新的内存并将原来的元素移动到新的内存，此时指向原内存地址的迭代器就失效了，first和end迭代器都失效
- 当vector在插入的时候，end迭代器肯定会失效
- 当vector在删除的时候，被删除元素以及它后面的所有元素迭代器都失效。

1.24、为什么C++没有实现垃圾回收?

- 首先，实现一个垃圾回收器会带来额外的空间和时间开销。你需要开辟一定的空间保存指针的引用计数和对他们进行标记mark。然后需要单独开辟一个线程在空闲的时候进行free操作。
- 垃圾回收会使得C++不适合进行很多底层的操作。

1.25、有一个类A，里面有个类B类型的b，还有一个B类型的*b，什么情况下要用到前者，什么情况下用后者?

- 一个具体的类和一个类的指针，主要差别就是占据的内存大小和读写速度。类占据的内存大，但是读写速度快。类指针内存小，但是读写需要解引用。所以可知，以搜索为主的场景中，应当使用类。以插入删除为主的场景中，应当使用类指针。

2、STL

2.1、C++的STL介绍

C++ STL从广义来讲包括了三类：算法，容器和迭代器。

- 算法包括排序，复制等常用算法，以及不同容器特定的算法。

- 容器就是数据的存放形式，包括序列式容器和关联式容器，序列式容器就是list，vector等，关联式容器就是set，map等。
- 迭代器就是在不暴露容器内部结构的情况下对容器的遍历。

2.2、STL源码中的hash表的实现

- STL中的hash表就unordered_map。使用的是哈希进行实现（注意与map的区别）。它记录的键是元素的哈希值，通过对比元素的哈希值来确定元素的值。
  unordered_map的底层实现是hashtable，采用开链法（也就是用桶）来解决哈希冲突，默认的桶大小是10.

- 哈希表的底层实现和扩容
  HashMap实现原理和扩容机制

2.3、解决哈希冲突的方式?

参考：哈希冲突及四种解决方法

1. 开放地址方法：
   当发生地址冲突时，按照某种方法继续探测哈希表中的其他存储单元，直到找到空位置为止。
   （1）线性探测
   按顺序决定值时，如果某数据的值已经存在，则在原来值的基础上往后加一个单位，直至不发生哈希冲突。
   （2）再平方探测
   按顺序决定值时，如果某数据的值已经存在，则在原来值的基础上先加1的平方个单位，若仍然存在则减1的平方个单位。随之是2的平方，3的平方等等。直至不发生哈希冲突。
   （3）伪随机探测
   按顺序决定值时，如果某数据已经存在，通过随机函数随机生成一个数，在原来值的基础上加上随机数，直至不发生哈希冲突。
2. 链式地址法（HashMap的哈希冲突解决方法）
   对于相同的值，使用链表进行连接。使用数组存储每一个链表。
   优点：
   （1）拉链法处理冲突简单，且无堆积现象，即非同义词决不会发生冲突，因此平均查找长度较短；
   （2）由于拉链法中各链表上的结点空间是动态申请的，故它更适合于造表前无法确定表长的情况；
   （3）开放定址法为减少冲突，要求装填因子α较小，故当结点规模较大时会浪费很多空间。而拉链法中可取α≥1，且结点较大时，拉链法中增加的指针域可忽略不计，因此节省空间；
   （4）在用拉链法构造的散列表中，删除结点的操作易于实现。只要简单地删去链表上相应的结点即可。
   缺点：
   指针占用较大空间时，会造成空间浪费，若空间用于增大散列表规模进而提高开放地址法的效率。
3. 再哈希法：当发生哈希冲突时使用另一个哈希函数计算地址值，直到冲突不再发生。这种方法不易产生聚集，但是增加计算时间，同时需要准备许多哈希函数。
4. 建立公共溢出区：采用一个溢出表存储产生冲突的关键字。如果公共溢出区还产生冲突，再采用处理冲突方法处理。

2.4 STL中unordered_map和map的区别

- unordered_map是使用哈希实现的，占用内存比较多，查询速度比较快，是常数时间复杂度。它内部是无序的，需要实现==操作符。
- map底层是采用红黑树实现的，插入删除查询时间复杂度都是O(log(n))，它的内部是有序的，因此需要实现比较操作符(<)。

2.5 STL中vector的实现

- STL中的vector是封装了动态数组的顺序容器。不过与动态数组不同的是，vector可以根据需要自动扩大容器的大小。具体策略是每次容量不够用时重新申请一块大小为原来容量两倍的内存，将原容器的元素拷贝至新容器，并释放原空间，返回新空间的指针。
  在原来空间不够存储新值时，每次调用push_back方法都会重新分配新的空间以满足新数据的添加操作。如果在程序中频繁进行这种操作，还是比较消耗性能的。

2.6 vector使用的注意点及其原因，频繁对vector调用push_back()对性能的影响和原因

- 如果需要频繁插入，最好先指定vector的大小，因为vector在容器大小不够用的时候会重新申请一块大小为原容器两倍的空间，并将原容器的元素拷贝到新容器中，并释放原空间，这个过程是十分耗时和耗内存的。频繁调用push_back()会使得程序花费很多时间在vector扩容上，会变得很慢。这种情况可以考虑使用list。

2.7 C++中vector和list的区别

- vector和数组类似，拥有一段连续的内存空间。vector申请的是一段连续的内存，当插入新的元素内存不够时，通常以2倍重新申请更大的一块内存，将原来的元素拷贝过去，释放旧空间。因为内存空间是连续的，所以在进行插入和删除操作时，会造成内存块的拷贝，时间复杂度为o(n)。
- list是由双向链表实现的，因此内存空间是不连续的。只能通过指针访问数据，所以list的随机存取非常没有效率，时间复杂度为o(n); 但由于链表的特点，能高效地进行插入和删除。
- vector拥有一段连续的内存空间，能很好的支持随机存取，因此vector::iterator支持“+”，“+=”，“<”等操作符。
- list的内存空间可以是不连续，它不支持随机访问，因此list::iterator则不支持“+”、“+=”、“<”等
- vector::iterator和list::iterator都重载了“++”运算符。
- 总之，如果需要高效的随机存取，而不在乎插入和删除的效率，使用vector;
  如果需要大量的插入和删除，而不关心随机存取，则应使用list。

2.8、string的底层实现

- string继承自basic_string,其实是对char*进行了封装，封装的string包含了char*数组，容量，长度等等属性。
- string可以进行动态扩展，在每次扩展的时候另外申请一块原空间大小两倍的空间（2*n），然后将原字符串拷贝过去，并加上新增的内容。

2.9 set，map和vector的插入复杂度

- map, set, multimap, and multiset
  上述四种容器采用红黑树实现，红黑树是平衡二叉树的一种。不同操作的时间复杂度近似为:
  插入: O(logN)
  查看:O(logN)
  删除:O(logN)

- hash_map, hash_set, hash_multimap, and hash_multiset
  上述四种容器采用哈希表实现，不同操作的时间复杂度为：
  插入:O(1)，最坏情况O(N)。
  查看:O(1)，最坏情况O(N)。
  删除:O(1)，最坏情况O(N)。

- vector的复杂度
  查看：O（1）
  插入：O（N）
  删除：O（N）

- list复杂度
  查看：O（N）
  插入：O（1）
  删除：O（1）

set,map的插入复杂度就是红黑树的插入复杂度，是log(N)。
unordered_set,unordered_map的插入复杂度是常数，最坏是O(N).
vector的插入复杂度是O(N),最坏的情况下（从头插入）就要对所有其他元素进行移动，或者扩容重新拷贝

数据结构	查找		插入		删除		遍历
	平均	最坏	平均	最坏	平均	最坏	
数组	O(N)	O(N)	O(N)	O(N)	O(N)	O(N)	--
有序数组	O(logN)	O(n)	O(N)	O(N)	O(N)	O(N)	O(N)
链表	O(N)	O(N)	O(1)	O(1)	O(1)	O(1)	--
有序链表	O(N)	O(N)	O(N)	O(N)	O(1)	O(1)	O(N)
二叉查找树	O(logN)	O(N)	O(logN)	O(N)	O(logN)	O(N)	O(N)
红黑树	O(logN)	O(logN)	O(logN)	O(logN)	O(logN)	O(logN)	O(N)
平衡树	O(logN)	O(logN)	O(logN)	O(logN)	O(logN)	O(logN)	O(N)
二叉堆/优先队列	O(1)	O(1)	O(logN)	O(logN)	O(logN)	O(logN)	O(N)
哈希表	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(N)

2.10、set、map特性与区别

- set：set是一种关联式容器，特性如下：

1. set低层以红黑树为低层容器；（低层实现：红黑树）
2. 所得元素的只有key没有value，value就是key；
3. 不允许出现键值重复；
4. 所有元素都会被自动排序
5. 不能通过迭代器来改变set的值，因为set的值就是键

- map：map是一种关联式容器，特性如下：

1. map以红黑树作为底层容器（低层实现：红黑树）

2. 所有元素都是键key+值value存在
3. 不允许键key重复
4. 所有元素是通过键进行自动排序的
5. map的键是不能修改的，但是其键对应的值是可以修改的

- uunordered_map

1. 低层实现：哈希表

2.11、map、set为什么要用红黑树实现

- 红黑树是一种二叉查找树，但在每个节点上增加一个存储为用于表示节点的颜色，可以是红或者黑。通过对任何一条从根到叶子节点的路径上各个节点的着色方式的限制，**红黑树确保没有一条路径会比其他路径长出两倍**，因此，红黑树是一种弱平衡树，但又相对与要求严格的AVL树来说，他的旋转次数较少，所以对于搜索，插入，删除操作比较多的情况下，通常使用红黑树。

2.12 STL里迭代器什么情况下失效，具体说几种情况

- 对于序列式容器(如vector,deque)，序列式容器就是数组式容器，删除当前的iterator会使后面所有元素的iterator都失效。这是因为vetor,deque使用了连续分配的内存，删除一个元素导致后面所有的元素会向前移动一个位置。

- 对于关联容器(如map, set,multimap,multiset)，删除当前的iterator，仅仅会使当前的iterator失效，只要在erase时，递增当前iterator即可。这是因为map之类的容器，使用了红黑树来实现，插入、删除一个结点不会对其他结点造成影响。

- 对于链表式容器(如list)，删除当前的iterator，仅仅会使当前的iterator失效，这是因为list之类的容器，使用了链表来实现，插入、删除一个结点不会对其他结点造成影响。

2.16、vector用swap来缩减空间

详见：vector用swap来缩减空间
容器v1只有两个元素，却有着很大的容量，会造成存储浪费。
所以我们
（1）用v1初始化一个临时对象，临时对象会根据v1的元素个数进行初始化；
（2）交换临时对象和v1；
（3）临时对象交换后销毁，v1原来的空间也销毁了；v1就指向现在的空间，明显占用空间减少。

3、c++特性

3.1、c++的重载和重写

- 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。
- 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。

3.2、C ++内存管理（热门问题）

详见：C++内存空间：静态存储区、栈、堆、文字常量区、程序代码区

在c++中，内存被分为五个区，分别是栈、堆、全局/局部静态存储区、常量存储区 和 代码区。

- 栈：在执行函数时，函数内局部变量的存储单元都可以在栈上创建，函数执行结束时这些存储单元自动被释放。栈内存分配运算内置于处理器的指令集中，效率很高但分配的内存有限。
- 堆：由new分配的内存块，们的释放编译器不去管，由我们的应用程序去控制，一般一个new就要对应一个delete。如果程序员没有释放掉，那么在程序结束后，操作系统会自动回收。
- 局/静态存储区，内存在程序编译的时候就已经分配好，这块内存在程序的整个运行期间都存在。它主要存放静态数据（局部static变量，全局static变量）、全局变量和常量。
- 常量存储区，这是一块比较特殊的存储区，他们里面存放的是常量字符串，不允许修改。
- 代码区，存放程序的二进制代码。

3.3、面向对象的三大特性

面向对象的三大特性是：封装，继承和多态。

- 封装隐藏了类的实现细节和成员数据，实现了代码模块化，如类里面的private和public；
- 继承使得子类可以复用父类的成员和方法，实现了代码重用；
- 多态则是“一个接口，多个实现”，通过父类调用子类的成员，实现了接口重用，如父类的指针指向子类的对象。

3.4、多态的实现

- C++ 多态包括编译时多态和运行时多态，编译时多态体现在函数重载和模板上，运行时多态体现在虚函数上。
- 虚函数：在基类的函数前加上virtual关键字，在派生类中重写该函数，运行时将会根据对象的实际类型来调用相应的函数。如果对象类型是派生类，就调用派生类的函数；如果对象类型是基类，就调用基类的函数。

3.4.2、函数重载的时候怎么解决同名冲突

- 对于C++，同名函数会根据参数类型和数量的不同，编译成不同的函数名，这样在链接阶段就可以正确的区分，从而实现重载。

3.5、c++虚函数相关（虚函数表，虚函数指针），虚函数的实现原理

- C++的虚函数是实现多态的机制。它是通过虚函数表实现的，虚函数表是每个类中存放虚函数地址的指针数组，类的实例在调用函数时会在虚函数表中寻找函数地址进行调用，如果子类覆盖了父类的函数，则子类的虚函数表会指向子类实现的函数地址，否则指向父类的函数地址。一个类的所有实例都共享同一张虚函数表。

- 虚表指针放在类的开头。通过对虚表指针的解引用找到虚表。

- 如果多重继承和多继承的话，子类的虚函数表长什么样子？
  多重继承的情况下越是祖先的父类的虚函数更靠前，多继承的情况下越是靠近子类名称的类的虚函数在虚函数表中更靠前。

- 虚表是一个指针数组，其元素是虚函数的指针，每个元素对应一个虚函数的函数指针。需要指出的是，普通的函数即非虚函数，其调用并不需要经过虚表，所以虚表的元素并不包括普

通函数的函数指针。
虚表内的条目，即虚函数指针的赋值发生在编译器的编译阶段，也就是说在代码的编译阶段，虚表就可以构造出来了。

- 虚表是属于类的，而不是属于某个具体的对象，一个类只需要一个虚表即可。同一个类的所有对象都使用同一个虚表。

- 为了指定对象的虚表，对象内部包含一个虚表的指针，来指向自己所使用的虚表。为了让每个包含虚表的类的对象都拥有一个虚表指针，编译器在类中添加了一个指针，*__vptr，用来指向虚表。这样，当类的对象在创建时便拥有了这个指针，且这个指针的值会自动被设置为指向类的虚表。

- 上面指出，一个继承类的基类如果包含虚函数，那个这个继承类也有拥有自己的虚表，故这个继承类的对象也包含一个虚表指针，用来指向它的虚表。

- 对象的**虚表指针用来指向自己所属类的虚表，虚表中的指针会指向其继承的最近的一个类的虚函数**

3.6编译器处理虚函数

- 如果类中有虚函数，就将虚函数的地址记录在类的虚函数表中。派生类在继承基类的时候，如果有重写基类的虚函数，就将虚函数表中相应的函数指针设置为派生类的函数地址，否则指向基类的函数地址。
为每个类的实例添加一个虚表指针（vptr），虚表指针指向类的虚函数表。实例在调用虚函数的时候，通过这个虚函数表指针找到类中的虚函数表，找到相应的函数进行调用。

3.7、基类的析构函数一般写成虚函数的原因

- 首先析构函数可以为虚函数，当析构一个指向子类的父类指针时，编译器可以根据虚函数表寻找到子类的析构函数进行调用，从而正确释放子类对象的资源。
- 如果析构函数不被声明成虚函数，则编译器实施静态绑定，在删除指向子类的父类指针时，只会调用父类的析构函数而不调用子类析构函数，这样就会造成子类对象析构不完全造成内存泄漏。

3.8、构造函数为什么一般不定义为虚函数

- 1）因为创建一个对象时需要确定对象的类型，而虚函数是在运行时确定其类型的。而在构造一个对象时，由于对象还未创建成功，编译器无法知道对象的实际类型，是类本身还是类的派生类等等
- 2）虚函数的调用需要虚函数表指针，而该指针存放在对象的内存空间中；若构造函数声明为虚函数，那么由于对象还未创建，还没有内存空间，更没有虚函数表地址用来调用虚函数即构造函数了

3.9构造函数或者析构函数中调用虚函数会怎样

- 在构造函数中调用虚函数，由于当前对象还没有构造完成，此时调用的虚函数指向的是基类的函数实现方式。
- 在析构函数中调用虚函数，此时调用的是子类的函数实现方式。

3.10、纯虚函数

- 纯虚函数是只有声明没有实现的虚函数，是对子类的约束，是接口继承
  包含纯虚函数的类是抽象类，它不能被实例化，只有实现了这个纯虚函数的子类才能生成对象
  使用场景：当这个类本身产生一个实例没有意义的情况下，把这个类的函数实现为纯虚函数，比如动物可以派生出老虎兔子，但是实例化一个动物对象就没有意义。并且可以规定派生的子类必须重写某些函数的情况下可以写成纯虚函数。

3.11、静态绑定和动态绑定

详见：C++中的静态绑定和动态绑定

- 静态绑定也就是将该对象相关的属性或函数绑定为它的静态类型，也就是它在声明的类型，在编译的时候就确定。在调用的时候编译器会寻找它声明的类型进行访问。

- 动态绑定就是将该对象相关的属性或函数绑定为它的动态类型，具体的属性或函数在运行期确定，通常通过虚函数实现动态绑定。

3.12、深拷贝和浅拷贝

- 浅拷贝就是将对象的指针进行简单的复制，原对象和副本指向的是相同的资源。

- 而深拷贝是新开辟一块空间，将原对象的资源复制到新的空间中，并返回该空间的地址。
  深拷贝可以避免重复释放和写冲突。例如使用浅拷贝的对象进行释放后，对原对象的释放会导致内存泄漏或程序崩溃。

3.13、对象复用的了解，零拷贝的了解

- 对象复用指得是设计模式，对象可以采用不同的设计模式达到复用的目的，最常见的就是继承和组合模式了。

- 零拷贝指的是在进行操作时，避免CPU从一处存储拷贝到另一处存储。在Linux中，我们可以减少数据在内核空间和用户空间的来回拷贝实现，比如通过调用mmap()来代替read调用。

  用程序调用mmap()，磁盘上的数据会通过DMA被拷贝的内核缓冲区，接着操作系统会把这段内核缓冲区与应用程序共享，这样就不需要把内核缓冲区的内容往用户空间拷贝。应用程序再调用write(),操作系统直接将内核缓冲区的内容拷贝到socket缓冲区中，这一切都发生在内核态，最后，socket缓冲区再把数据发到网卡去。

3.14、c++的所有构造函数

- 默认构造函数是当类没有实现自己的构造函数时，编译器默认提供的一个构造函数。
- 重载构造函数也称为一般构造函数，一个类可以有多个重载构造函数，但是需要参数类型或个数不相同。可以在重载构造函数中自定义类的初始化方式。
- 拷贝构造函数是在发生对象复制的时候调用的。

3.15、什么时候调用拷贝构造函数

详见：C++拷贝构造函数详解

- 对象以值传递的方式传入函数参数

如 void func(Dog dog){};

- 对象以值传递的方式从函数返回

  如 Dog func(){ Dog d; return d;}

- 对象需要通过另外一个对象进行初始化

3.16 结构体内存对齐方式和为什么要进行内存对齐?

因为结构体的成员可以有不同的数据类型，所占的大小也不一样。同时，由于CPU读取数据是按块读取的，内存对齐可以使得CPU一次就可以将所需的数据读进来。

对齐规则：

- 第一个成员在与结构体变量偏移量为0的地址
- 其他成员变量要对齐到某个数字（对齐数）的整数倍的地址处。
- 对齐数=编译器默认的一个对齐数 与 该成员大小的较小值。
- linux 中默认为4
- vs 中的默认值为8
- 结构体总大小为最大对齐数的整数倍（每个成员变量除了第一个成员都有一个对齐数）

3.17、内存泄露的定义，如何检测与避免?

- 动态分配内存所开辟的空间，在使用完毕后未手动释放，导致一直占据该内存，即为内存泄漏。

几种原因：

- 类的构造函数和析构函数中new和delete没有配套；
- 在释放对象数组时没有使用delete[]，使用了delete；
- 没有将基类的析构函数定义为虚函数，当基类指针指向子类对象时，如果基类的析构函数不是virtual，那么子类的析构函数将不会被调用，子类的资源没有正确释放，因此造成内存泄露
- 没有正确的清楚嵌套的对象指针

避免方法：

- malloc/free要配套
- 使用智能指针；
- 将基类的析构函数设为虚函数；

3.18、c++智能指针

C++中的智能指针有auto_ptr,shared_ptr,weak_ptr和unique_ptr。智能指针其实是将指针进行了封装，可以像普通指针一样进行使用，同时可以自行进行释放，避免忘记释放指针指向的内存地址造成内存泄漏。

- auto_ptr是较早版本的智能指针，在进行指针拷贝和赋值的时候，新指针直接接管旧指针的资源并且将旧指针指向空，但是这种方式在需要访问旧指针的时候，就会出现问题。
- unique_ptr是auto_ptr的一个改良版，不能赋值也不能拷贝，保证一个对象同一时间只有一个智能指针。

- shared_ptr可以使得一个对象可以有多个智能指针，当这个对象所有的智能指针被销毁时就会自动进行回收。（内部使用计数机制进行维护）
- weak_ptr是为了协助shared_ptr而出现的。它不能访问对象，只能观测shared_ptr的引用计数，防止出现死锁。

补充：

- share_ptr原理：
  shared_ptr是可以共享所有权的指针。如果有多个shared_ptr共同管理同一个对象时，只有这些shared_ptr全部与该对象脱离关系之后，被管理的对象才会被释放。
  shared_ptr的管理机制其实并不复杂，就是对所管理的对象进行了引用计数，当新增一个shared_ptr对该对象进行管理时，就将该对象的引用计数加一；减少一个shared_ptr对该对象进行管理时，就将该对象的引用计数减一，如果该对象的引用计数为0的时候，说明没有任何指针对其管理，才调用delete释放其所占的内存。
  参考：[C++] Boost智能指针——boost::shared_ptr（使用及原理分析）

3.19、 inline关键字说一下 和宏定义有什么区别

- inline是内联的意思，可以定义比较小的函数。因为函数频繁调用会占用很多的栈空间，进行入栈出栈操作也耗费计算资源，所以可以用inline关键字修饰频繁调用的小函数。编译器会在编译阶段将代码体嵌入内联函数的调用语句块中。

1. 内联函数在编译时展开，而宏在预编译时展开
2. 在编译的时候，内联函数直接被嵌入到目标代码中去，而宏只是一个简单的文本替换。
3. 内联函数可以进行诸如类型安全检查、语句是否正确等编译功能，宏不具有这样的功能。
4. 宏不是函数，而inline是函数
5. 宏在定义时要小心处理宏参数，一般用括号括起来，否则容易出现二义性。而内联函数不会出现二义性。
6. inline可以不展开，宏一定要展开。因为inline指示对编译器来说，只是一个建议，编译器可以选择忽略该建议，不对该函数进行展开。
7. 宏定义在形式上类似于一个函数，但在使用它时，仅仅只是做预处理器符号表中的简单替换，因此它不能进行参数有效性的检测，也就不能享受C++编译器严格类型检查的好处，另外它的返回值也不能被强制转换为可转换的合适的类型，这样，它的使用就存在着一系列的隐患和局限性。

3.20、模板的用法与适用场景 实现原理

用template <typename t="">关键字进行声明，接下来就可以进行模板函数和模板类的编写了</typename>

编译器会对函数模板进行两次编译：第一次编译在声明的地方对模板代码本身进行编译，这次编译只会进行一个语法检查，并不会生成具体的代码。第二次编译时对代码进行参数替换后再进行编译，生成具体的函数代码。

3.21、 成员初始化列表的概念，为什么用成员初始化列表会快一些（性能优势）？

成员初始化列表就是在类或者结构体的构造函数中，在参数列表后以冒号开头，逗号进行分隔的一系列初始化字段。
复制代码

```
class A{
int id;
string name;
FaceImage face;
A(int& inputID,string& inputName,FaceImage&
inputFace):id(inputID),name(inputName),face(inputFace){} // 成员初始化列表
};
```

因为使用成员初始化列表进行初始化的话，会直接使用传入参数的拷贝构造函数进行初始化，省去了一次执行传入参数的默认构造函数的过程，否则会调用一次传入参数的默认构造函数。所以使用成员初始化列表效率会高一些。

另外，有三种情况是必须使用成员初始化列表进行初始化的：

- 常量成员的初始化，因为常量成员只能初始化不能赋值
- 引用类型
- 没有默认构造函数的对象必须使用成员初始化列表的方式进行初始化
  详见：C++ 初始化列表

3.22、C++的调用惯例（简单一点C++函数调用的压栈过程）

函数的调用过程
1） 从栈空间分配存储空间
2）从实参的存储空间复制值到形参栈空间
3）进行运算
形参在函数未调用之前都是没有分配存储空间的，在函数调用结束之后，形参弹出栈空间，清除形参空间。
数组作为参数的函数调用方式是地址传递，形参和实参都指向相同的内存空间，调用完成后，形参指针被销毁，但是所指向的内存空间依然存在，不能也不会被销毁。
当函数有多个返回值的时候，不能用普通的 return 的方式实现，需要通过传回地址的形式进行，即地址/指针传递。

3.23 C++的四种强制转换

四种强制类型转换操作符分别为：static_cast、dynamic_cast、const_cast、reinterpret_cast

- 1）static_cast：
  用于各种隐式转换。具体的说，就是用户各种基本数据类型之间的转换，比如把int换成char，float换成int等。以及派生类（子类）的指针转换成基类（父类）指针的转换。

  特性与要点：

  1. *它没有运行时类型检查，所以是有安全隐患的。*
  2. *在派生类指针转换到基类指针时，是没有任何问题的，在基类指针转换到派生类指针的时候，会有安全问题。*
  3. *static_cast不能转换const，volatile等属性*

- 2）dynamic_cast：
  用于动态类型转换。具体的说，就是在基类指针到派生类指针，或者派生类到基类指针的转换。
  dynamic_cast能够提供运行时类型检查，只用于含有虚函数的类。
  dynamic_cast如果不能转换返回NULL。
- 3）const_cast：
  用于去除const常量属性，使其可以修改 ，也就是说，原本定义为const的变量在定义后就不能进行修改的，但是使用const_cast操作之后，可以通过这个指针或变量进行修改; 另外还有volatile属性的转换。
- 4）reinterpret_cast
  几乎什么都可以转，用在任意的指针之间的转换，引用之间的转换，指针和足够大的int型之间的转换，整数到指针的转换等。但是不够安全。

4、调试程序

4.1、调试程序的方法

- 通过设置断点进行调试
- 打印log进行调试
- 打印中间结果进行调试

4.2、 遇到coredump要怎么调试

- coredump是程序由于异常或者bug在运行时异常退出或者终止，在一定的条件下生成的一个叫做core的文件，这个core文件会记录程序在运行时的内存，寄存器状态，内存指针和函数堆栈信息等等。对这个文件进行分析可以定位到程序异常的时候对应的堆栈调用信息。

- 使用gdb命令对core文件进行调试
  gdb [可执行文件名] [core文件名]

4.3、一个函数或者可执行文件的生成过程或者编译过程是怎样的

预处理、编译、汇编、链接

- 1）预处理： 对预处理命令进行替换等预处理操作
  主要处理源代码文件中的以“#”开头的预编译指令。处理规则见下
  1、删除所有的#define，展开所有的宏定义。
  2、处理所有的条件预编译指令，如“#if”、“#endif”、“#ifdef”、“#elif”和“#else”。
  3、处理“#include”预编译指令，将文件内容替换到它的位置，这个过程是递归进行的，文件中包含其他文件。
  4、删除所有的注释，“//”和“/**/”。
  5、保留所有的#pragma 编译器指令，编译器需要用到他们，如：#pragma once 是为了防止有文件被重复引用。
  6、添加行号和文件标识，便于编译时编译器产生调试用的行号信息，和编译时产生编译错误或警告是能够显示行号。

- 2）编译：代码优化和生成汇编代码
  把预编译之后生成的xxx.i或xxx.ii文件，进行一系列词法分析、语法分析、语义分析及优化后，生成相应的汇编代码文件。
  1、词法分析：利用类似于“有限状态机”的算法，将源代码程序输入到扫描机中，将其中

的字符序列分割成一系列的记号。
2、语法分析：语法分析器对由扫描器产生的记号，进行语法分析，产生语法树。由语法分析器输出的语法树是一种以表达式为节点的树。
3、语义分析：语法分析器只是完成了对表达式语法层面的分析，语义分析器则对表达式是否有意义进行判断，其分析的语义是静态语义——在编译期能分期的语义，相对应的动态语义是在运行期才能确定的语义。
4、优化：源代码级别的一个优化过程。
5、目标代码生成：由代码生成器将中间代码转换成目标机器代码，生成一系列的代码序列——汇编语言表示。
6、目标代码优化：目标代码优化器对上述的目标机器代码进行优化：寻找合适的寻址方式、使用位移来替代乘法运算、删除多余的指令等。

- 3）汇编：将汇编代码转化为机器语言
将汇编代码转变成机器可以执行的指令(机器码文件)。 汇编器的汇编过程相对于编译器来说更简单，没有复杂的语法，也没有语义，更不需要做指令优化，只是根据汇编指令和机器指令的对照表一一翻译过来，汇编过程有汇编器as完成。经汇编之后，产生目标文件(与可执行文件格式几乎一样)xxx.o(Windows下)、xxx.obj(Linux下)。

- 4）链接：将目标文件彼此链接起来
将不同的源文件产生的目标文件进行链接，从而形成一个可以执行的程序。链接分为静态链接和动态链接：
1、静态链接：
函数和数据被编译进一个二进制文件。在使用静态库的情况下，在编译链接可执行文件时，链接器从库中复制这些函数和数据并把它们和应用程序的其它模块组合起来创建最终的可执行文件。
空间浪费：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，所以如果多个程序对同一个目标文件都有依赖，会出现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。
运行速度快：但是静态链接的优点就是，在可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。

2、动态链接：
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。
共享库：就是即使需要每个程序都依赖同一个库，但是该库不会像静态链接那样在内存中存在多分，副本，而是这多个程序在执行时共享同一份副本；
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。当程序下一次运行时，新版本的目标文件会被自动加载到内存并且链接起来，程序就完成了升级的目标。
性能损耗：因为把链接推迟到了程序运行时，所以每次执行程序都需要进行链接，所以性能会有一定损失。

- 进程的加载过程：
详见：进程的加载过程
进程的执行过程需要经过三大步骤：编译，链接和装入。

- 编译：将源代码编译成若干模块；

- 链接：将编译后的模块和所需的库函数进行链接。
链接包括三种形式：静态链接，装入时动态链接（将编译后的模块在链接时一边链接一边装入），运行时动态链接（在执行时才把需要的模块进行链接）

- 装入：将模块装入内存运行
将进程装入内存时，通常使用分页技术，将内存分成固定大小的页，进程分为固定大小的

块，加载时将进程的块装入页中，并使用页表记录。减少外部碎片。
通常操作系统还会使用虚拟内存的技术将磁盘作为内存的扩充。

4.4、vs2013检查内存泄漏

详见：VS 查看是否有内存泄露的方法 定位位置
在main函数中调用下面的函数：
_CrtSetDbgFlag(_CRTDBG_ALLOC_MEM_DF|_CRTDBG_LEAK_CHECK_DF);

执行后将在输出窗口出未释放的指针的位置。

5、c11新特性

5.1、C11新特性

- 自动类型推导auto：auto的自动类型推导用于从初始化表达式中推断出变量的数据类型。通过auto的自动类型推导，可以大大简化我们的编程工作。
- nullptr：nullptr是为了解决原来C++中NULL的二义性问题而引进的一种新的类型，因为NULL实际上代表的是0，而nullptr是void*类型的
- lambda表达式：它类似Javascript中的闭包，它可以用于创建并定义匿名的函数对象，以简化编程工作。Lambda的语法如下：
  函数对象参数mutable或exception声明->返回值类型{函数体}
- thread类和mutex类
- 新的智能指针 unique_ptr和shared_ptr

详见：【面试知识整理】CPP——C++ 11的新特性

二、计算机网络基础

1、 计算机体系模型

1.1、各层模型

复制代码

```
七层模型：物理层、数据链路层、网络层、传输层、会话层、应用层
tcp/ip四层模型：网络接口层、网络层、传输层、应用层
五层模型：物理层、数据链路层、网络层、传输层、应用层
```

1.2、各层常用的协议：

复制代码

```
网络层：IP、ICMP、ARP、RARP、OSPF、IPX、RIP、IGRP
传输层：TCP、UDP
应用层：DNS、Telnet、SMTP、HTTP
```

1.3、展开讲讲各个协议作用

复制代码

```
IP（Internet Protocol，网际协议）是为计算机网络相互连接进行通信而设计的协议。
ARP（Address Resolution Protocol，地址解析协议）
ICMP（Internet Control Message Protocol，网际控制报文协议）：
ICMP 是 TCP/IP 模型中网络层的重要成员，ping 和 tracert是两个常用网络管理命令，
        ping： 用来测试网络可达性，
        traceroute（路由追踪）： 用来检测发出数据包的主机到目标主机之间所经过的网关数量的工具。
IGMP（Internet Group Management Protocol，网际组管理协议）
```

参考：Linux命令：traceroute命令（路由跟踪）

1.4、对路由协议的了解与介绍。内部网关协议IGP包括RIP，OSPF，和外部网关协议EGP和BGP.

- IGP（内部网关协议）：**是在一个自治网络内网关（主机和路由器）间交换路由信息的协议。**路由信息能用于网间协议（IP）或者其它网络协议来说明路由传送是如何进行的。IGP协议包括RIP、OSPF、IS-IS、IGRP、EIGRP。
    - 内部网关协议分类：参考：内部网关协议
    - 距离矢量路由协议：

      距离矢量是指以距离和方向构成的矢量来通告路由信息。距离按跳数等度量来定义，方向则是下一跳的路由器或送出接口。距离矢量协议通常使用贝尔曼-福特 (Bellman-Ford) 算法来确定最佳路径。尽管贝尔曼-福特算法最终可以累积足够的信息来维护可到达网络的数据库，但路由器无法通过该算法了解网际网络的确切拓扑结构。路由器仅了解从邻近路由器接收到的路由信息。
      距离矢量协议适用于以下情形：
      （1） 网络结构简单、扁平，不需要特殊的分层设计。
      （2）管理员没有足够的知识来配置链路状态协议和排查故障。
      （3）特定类型的网络拓扑结构，如集中星形（Hub-and-Spoke）网络。
      （4）无需关注网络最差情况下的收敛时间。

    - 链路状态路由协议

      配置了链路状态路由协议的路由器可以获取所有其它路由器的信息来创建网络的“完整视图”（即拓扑结构）。并在拓扑结构中选择到达所有目的网络的最佳路径（链路状态路由协议是触发更新，就是说有变化时就更新）。
      链路状态协议适用于以下情形：
      （1）网络进行了分层设计，大型网络通常如此。
      （2）管理员对于网络中采用的链路状态路由协议非常熟悉。
      （3）网络对收敛速度的要求极高。

- RIP：路由信息协议(Route Information Protocol)”的简写，主要传递路由信息，通过每隔30秒广播一次路由表，维护相邻路由器的位置关系，同时根据收到的路由表信息使用动态规划的方式计算自己的路由表信息。RIP是一个**距离矢量路由协议**,最大跳数为16跳,16跳以及超过16跳的网络则认为目标网络不可达。

- OSPF:OSPF(Open Shortest Path First开放式最短路径优先）是一个内部网关协议(Interior Gateway Protocol，简称IGP），用于在单一自治系统（autonomous system,AS）内决策路由。是对链路状态路由协议的一种实现，隶属内部网关协议（IGP），故运作于自治系统内部.OSPF详解

1.5、介绍一下ping的过程，分别用到了哪些协议

详见：ping 原理与ICMP协议

ping通过ICMP协议来进行工作。ICMP：网络报文控制协议

- 首先，ping命令会构建一个ICMP请求数据包，然后由ICMP协议将这个数据包连同目的IP地址，源IP地址一起交给IP协议；
- 然后IP协议会构建一个IP数据包并在映射表中查找目的IP对应的MAC地址，将其交给数据链路层。
- 数据链路层会构建一个数据帧，附上源mac地址和目的mac地址发送出去；
- 目的主机收到数据帧后，会检查数据包上的mac地址和本机mac地址是否相符，如果相符，就把其中的信息提取出来交给IP协议，IP协议提取信息后交给ICMP协议，然后构建一个ICMP应答包，用相同的过程发回去。

1.6、DNS的工作过程和原理

- DNS解析有两种方式：递归查询和迭代查询
- **递归查询**：用户先向本地域名服务器查询，如果本地域名服务器的缓存没有IP地址映射记录，就向根域名服务器查询，根域名服务器就会向顶级域名服务器查询，顶级域名服务器向权限域名服务器查询，查到结果后依次返回。
- **迭代查询**：用户向本地域名服务器查询，如果没有缓存，本地域名服务器会向根域名服务器查询，根域名服务器返回顶级域名服务器的地址，本地域名服务器再向顶级域名服务器查询，得到权限域名服务器的地址，本地域名服务器再向权限域名服务器查询得到结果。

1.7、IP寻址的过程（ARP协议工作）

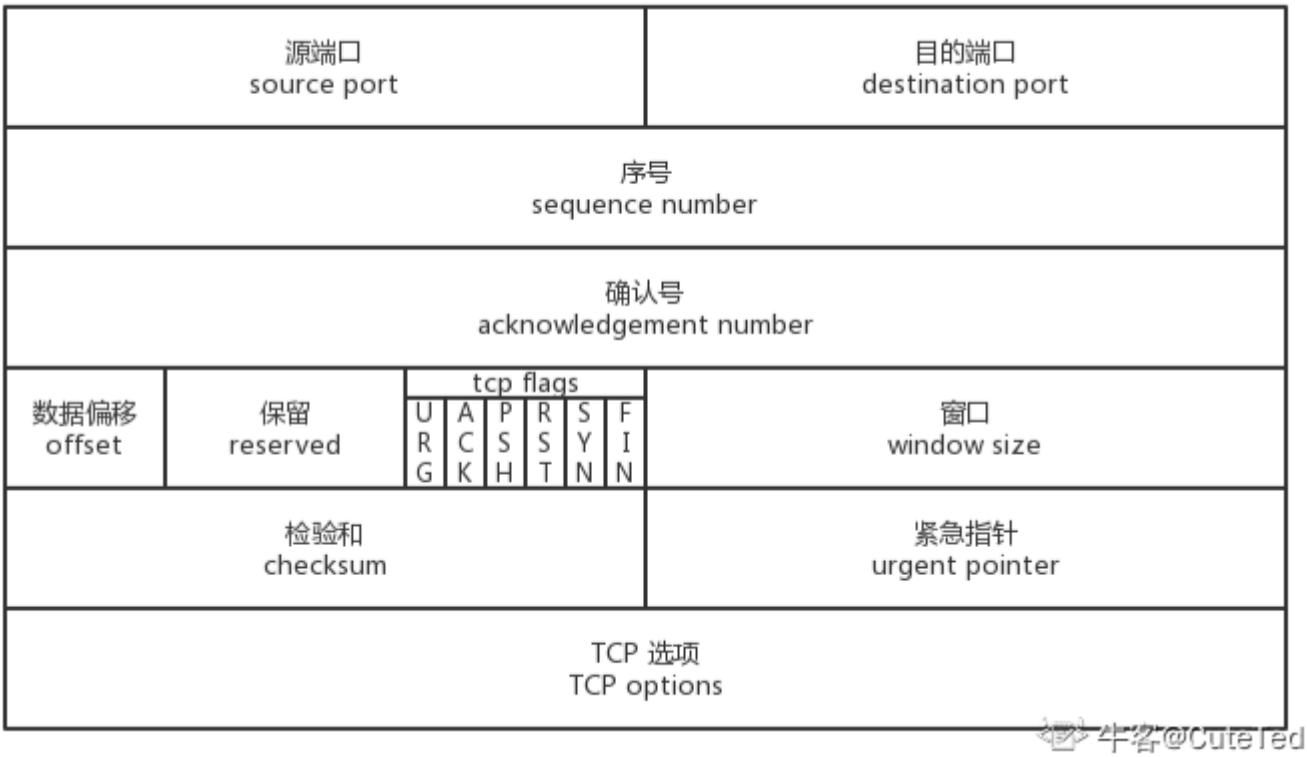
- IP寻址的工作原理，（包括本地网络寻址和非本地网络寻址）
- 本地网络寻址：
  本地网络实现IP 寻址，也就是我们所说的同一网段通信过程，现在我们假设有2个主机，他们是属于同一个网段。主机A和主机B，首先主机A通过本机的hosts表或者wins系统或dns系统先将主机B的计算机名 转换为Ip地址，然后用自己的 Ip地址与子网掩码计算出自己所出的网段，比较目的主机B的ip地址与自己的子网掩码，发现与自己是出于相同的网段，于是在自己的ARP缓存中查找是否有主机B 的mac地址，如果能找到就直接做数据链路层封装并且通过网卡将封装好的以太网帧发送有物理线路上去：如果arp缓存中没有主机B的的mac地址，主机A将启动arp协议通过在本地网络上的arp广播来查询主机B的mac地址，获得主机B的mac地址厚写入arp缓存表，进行数据链路层的封装，发送数据。
- 非本地网络寻址：不同的数据链路层网络必须分配不同网段的Ip地址并且由路由器将其连接起来。主机A通过本机的hosts表或wins系统或dns系统先主机B的计算机名转换为IP地址，然后用自己的Ip地址与子网掩码计算出自己所处的网段，比较目的目的主机B的Ip地址，发现与自己处于不同的网段。于是主机A将知道应该将次数据包发送给自己的缺省网关，即路由器的本地接口。主机A在自己的ARP缓存中查找是否有缺省网关的MAC地址，如果能够找到就直接做数据链路层封装并通过网卡 将封装好的以太网数据帧发送到物理线路上去，如果arp缓存表中没有缺省网关的Mac地址，主机A将启动arp协议通过在本地网络上的arp广播来查询缺省网关的mac地址，获得缺省网关的mac地址后写入arp缓存表，进行数据链路层的封装，发送数据。数据帧到达路由器的接受接口后首先解封装，变成ip数据包，对ip 包

进行处理，根据目的Ip地址查找路由表，决定转发接口后做适应转发接口数据链路层协议帧的封装，并且发送到下一跳路由器，次过程继续直至到达目的的网络与目的主机。
详见：ARP协议的工作机制详解

2、TCP相关

2.1、TCP/UDP/IP 报文结构：

- TCP头部 + TCP数据部分
  TCP头部结构： 源端口 + 目的端口 + **序号（seq）** + **确认号（ack）** + **tcp flag（ACK + SYN + FIN）** + 偏移位 + 校验和 + TCP选项
- UDP头部结构： 源端口 + 目的端口 + 长度 + 校验和
- IP数据报头部结构： ID域标识 + 标志 + 片位移 + 源IP地址 + 目的IP地址 + 协议 + 校验和 + 总长度等等；
- 参考： IP、TCP、UDP首部详解



2.2、TCP和UDP的区别

- TCP是面向连接的协议，提供的是可靠传输，在收发数据前需要进行三次握手协议完成连接，使用ACK对手发的数据进行正确性检验。// UDP是面向无连接的协议，尽最大努力交付，不管对方是否收到了数据或者收到的数据是否正确。
- TCP提供**流量控制**和**拥塞控制**。// UDP不提供。
- TCP是全双工的可靠信道 // UDP则是不可靠信道
- TCP对系统资源的要求高于UDP，所以速度要慢于UDP。
- TCP包没有边界，会出现黏包问题，实际上TCP把数据看成是一连串无结构的字节流。// UDP面向报文，不会出现黏包问题。
- 在应用方面，强调数据完整性和正确性用TCP // 要求性能和速度时使用UDP。

2.3、TCP和UDP相关的协议和端口号

- TCP族的协议有： HTTP，HTTPS，SMTP,FTP,

- UDP族的协议有： DNS,DHCP
- TCP,UDP相关协议和端口号

2.4、TCP如何保证可靠传输

- 校验和
  发送的数据包的二进制相加然后取反，目的是检验数据在传输过程中是否有变化。如果收到段的校验和有差错，TCP将丢弃这个报文段和不确认接收到此报文段。

- 确认应答 + 序列号
  TCP给发送的每一个数据包进行编号，接收方对数据包进行排序，把有序数据传递给应用层。

- 超时重传
  当TCP发出一个段后，会启动一个计时器，等待目的端口确认接收到这个报文段。如果不能及时收到一个接收确认，将重传这个报文段。

- 流量控制
  TCP连接的每一方都有固定大小的缓冲空间，TCP的接收端只允许发送端发送接收端缓冲区能够接纳的数据。当接收方来不及处理发送方的数据时，能够提示发送方降低发送的速率，防止包丢失。
  TCP用的流量控制协议是可变大小的滑动窗口协议。接收方有**即时窗口**（滑动窗口 rwnd），随ACK报文发送。

- 拥塞控制
  当网络拥塞时，减少数据的发送。
  发送方有**拥塞窗口**，随ACK报文发送。

2.5、使用UDP如何实现可靠传输

- UDP如何实现可靠传输
  因为UDP是面向无连接到的协议，在传输层上无法保证可靠传输。如果想要实现可靠传输，只能在应用层下手。需要实现seq/ack机制，重传机制和窗口确认机制。
  即要接收方收到UDP之后回复个确认包，发送方有个机制，收不到确认包就要重新发送，每个包有递增的序号，接收方发现中间丢了包就要发重传请求，当网络太差时候频繁丢包，防止越丢包越重传的恶性循环，要有个发送窗口的限制，发送窗口的大小根据网络传输情况调整，调整算法要有一定自适应性。

2.6、TCP拥塞控制，算法名字（极其重要）（TCP发送方）

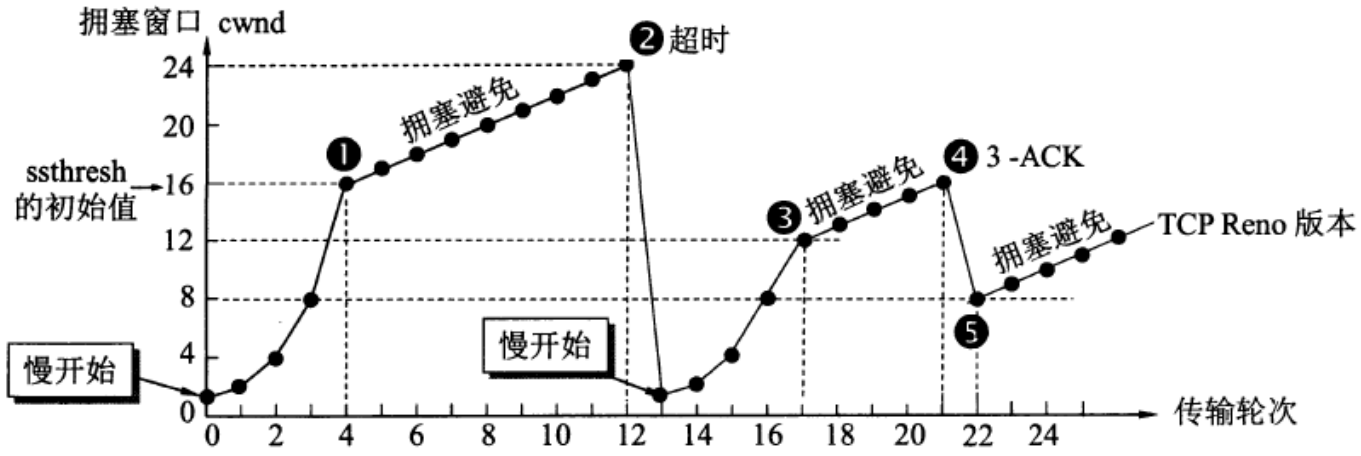


图 5-25　TCP 拥塞窗口 cwnd 在拥塞控制时的变化情况

- 防止过多的数据注入到网络中，这样可以是网络中的路由器或者链路不致过载，拥塞控制是控制发送者的流量。拥塞控制有四种算法：**慢启动，拥塞避免，快速重传、快速恢复。**
- 发送方发送方维持一个拥塞窗口 cwnd (congestion window)的状态变量。拥塞窗口的大小取决于网络的拥塞程度，并且动态地在变化。发送方让自己的发送窗口等于**拥塞窗口和接受窗口的较小值**。

1. **慢启动**：思路是当主机开始发送数据时，先以较小的拥塞窗口进行发送，然后每次翻倍。即，由小到大（指数级别）增加拥塞窗口的大小。此外，为了防止拥塞窗口cwnd增长速度过快引起网络堵塞，还需设置一个慢启动阈值ssthresh状态变量，当拥塞窗口大于阈值ssthresh时，停止使用慢启动改用拥塞避免算法。
2. **拥塞避免：** 让拥塞窗口cwnd慢慢增大，即每经过一个往返时间RTT就让发送方的拥塞窗口cwnd加一。
3. **快速重传： 当发送端连续收到三个重复的ack时**，表示该数据段已经丢失，需要重发。此时慢启动阈值ssth变为原来一半，拥塞窗口cwnd变为ssth，然后+1+1的发（每一轮rtt+1）。
4. **快速恢复：** 当超过设定时间没有收到某个报文段的ack时，表示网络拥塞，慢启动阈值ssth变为原来一半，拥塞窗口cwnd=1，进入慢启动阶段。

2.7、流量控制，采用滑动窗口法存在的问题（死锁可能，糊涂窗口综合征）

- **定义：**流量控制即让发送方发送速率不要过快，让接收方有来得及接收，可以通过TCP报文中的窗口大小字段来控制发送方的发送窗口不大于接收方发回的窗口大小，这样就可以实现流量控制

- **可能存在的问题：** 特殊情况下，接收方没有足够的缓存可以使用，就会发送零窗口大小的报文，此时发送方接收到这个报文后，停止发送。过了一段时间，接收方有了足够的缓存，开始接收数据了，发送了一个非零窗口的报文，这个报文在传输过程中丢失，那么发送方的发送窗口一直为零导致死锁发生。

- **解决方法：** 为了解决这个问题，TCP为每一个连接设定一个持续计时器（persistence timer），只要TCP的一方接收到了对方的零窗口通知，就会启动这个计时器，周期性的发送零窗口探测报文段，对方在确认这个报文时给出现在的窗口大小。（注意：TCP规定，即便是在零窗口状况下，也必须接收以下几种报文段：零窗口探测报文、确认段报文、携带紧急数据的报文段）。

- 在什么情况下会减慢拥塞窗口增长的速度

    1. 到达慢开始门限ssthresh，采用拥塞避免算法；

2. 出现丢包现象，就进入快速恢复；
3. 当连续收到三个重复确认，进入快速重传。

2.8、TCP滑动窗口协议

- 详见：TCP-IP详解：滑动窗口（Sliding Window）
- TCP的滑动窗口协议用来控制发送方和接收方的发送速率，避免拥塞的发生。**滑动窗口即接收方缓冲区的大小，用来告知发送方对它的接收还有多大的缓存空间。**在接收方的滑动窗口已知的情况下，当接收方确认了连续的数据序列之后，发送方的滑动窗口向后滑动，发送下一个数据序列。

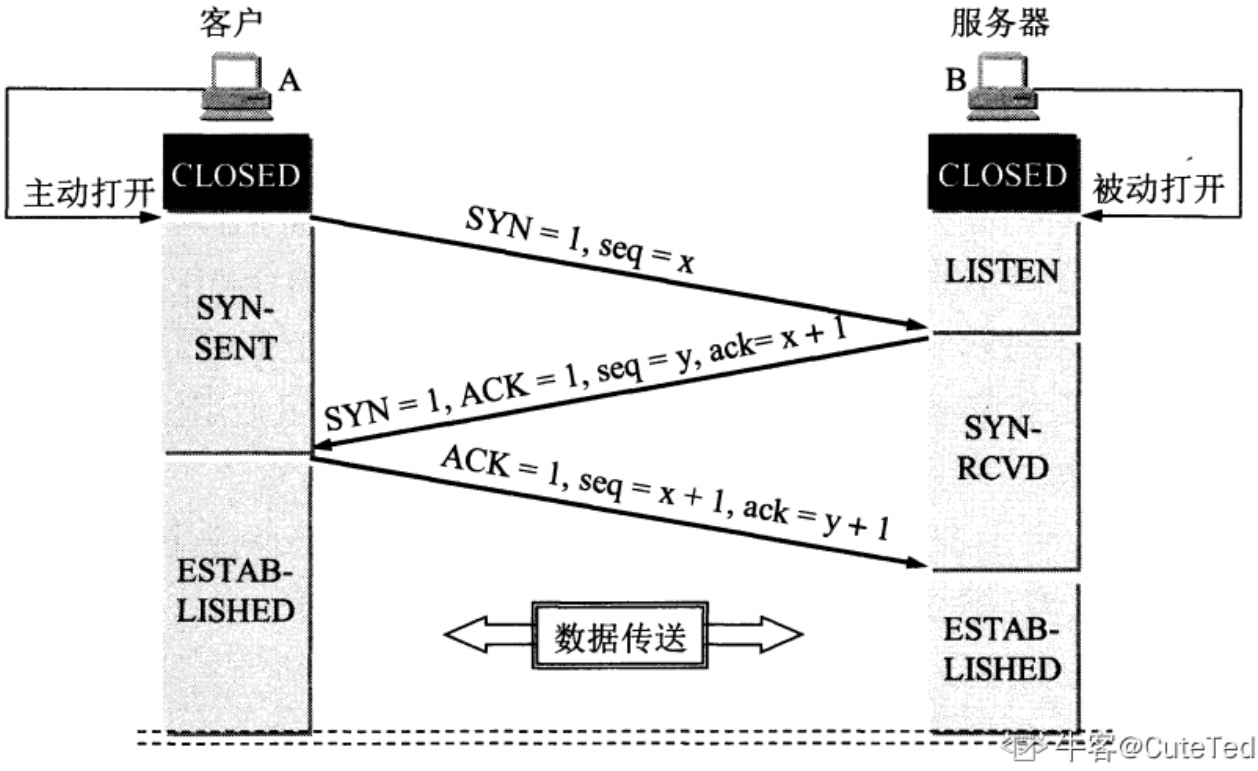
2.9、拥塞控制和流量控制的区别

- **拥塞控制**是防止过多的数据注入到网络中，导致网络发生拥塞；
- **流量控制**是防止发送方一下子发送过多的数据到接收方，导致接收方缓存放不下。
- 两种算法都是对发送方的行为进行控制。

2.10 TCP的三次握手和四次挥手（热门问题）

2.10.1. 三次握手

- **第一次握手**：client给server段发送建立连接请求，在这个报文中，包含了SYN=1，seq=任意值x，发送后client处于SYN_SNET，状态；
- **第二次握手**：server段接收到这个请求后，进行资源分配，同时返回一个ACK报文，这个报文中包含了ACK=1，SYN=1，ack=x+1，seq=y，此时server端处于**SYN-RCVD**状态；
- **第三次握手**： client接收到server发送的ACK信息后，可以看到ack=x+1知道server接收到了消息，也给server回一个ACK报文，报文中包含ACK=1，ack=y+1，seq=x+1。这样，三次握手以后，连接建立了，client端进入**established** 状态。



2.10.2. 四次挥手

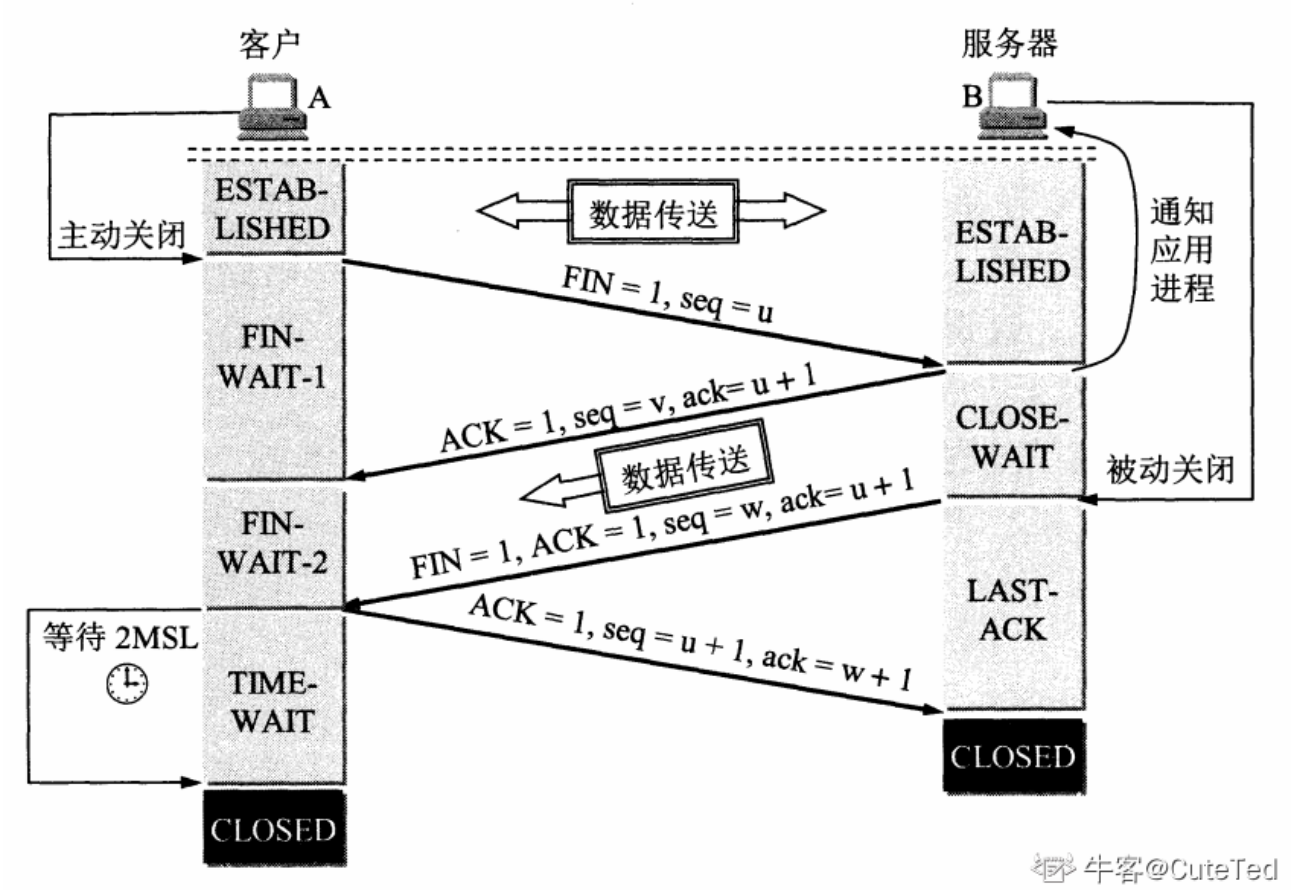
TCP断开连接一般都是一方主动，一方被动的。这里假设client主动，server被动

- **第一次挥手**：当客户端没有数据要继续 发送给服务器端时，会发送一个**FIN报文**，报文中包含FIN=1，seq=u，告诉server：“我已经没有数据要发给你了，但是你要是还想给我发数据的话，你就接着发，但是你得告诉我你收到我的关闭信息了”，然后进入FIN-WAIT-1状态**；

- **第二次挥手**：服务器在在接受到以上报文后，会告诉客户端：“我收到你的FIN消息了，但是你等我发完的”。此时给client会回复一个ACK=1，seq=v，ack=u+1，的**ACK报文。**服务器进入**CLOSE_WAIT状态。**

- **第三次挥手**：当server发完所有数据时，他会给client发送一个**FIN报文**，告诉client说“我传完数据了，现在要关闭连接了”，报文中包含：ACK=1，FIN=1，seq=w，ack=u+1。然后服务器端进入**LAST-ACK状态。**

- **第四次挥手**：
  当client收到这个消息时，他会给server发**ACK报文**，但是它不相信网络，怕server收不到信息，它会进入**TIME_WAIT状态**，万一server没收到ACK消息它可以可以重传，而当server收到这个ACK信息后，就正式关闭了tcp连接，处于**CLOSED状态**，而client等待了**2MSL**这样长时间后还没等到消息，它知道server已经关闭连接了，于是乎他自己也断开了，



2.10.3. **为什么使用三次握手，两次握手可不可以？（重点，热门问题）**

- 因为只使用两次握手的话，服务器端无法确认客户端是否接收到上一个ACK报文，是否做好准备进入接收状态。

2.10.4. **TIME_WAIT的意义（为什么要等于2MSL）**

- TIME_WAIT是指四次挥手中客户端接收了服务端的FIN报文并发送ACK报文给服务器后，仍然需要等待2MSL时间的过程。虽然按道理，四个报文都发送完毕，我们可以直接进入CLOSE状态了，但是我们必须假象网络是不可靠的，有可以最后一个ACK丢失。如果客户端

发送的ACK发生丢失，服务器会再次发送FIN报文给客户端，所以TIME_WAIT状态就是用来重发可能丢失的ACK报文。

2.11、建立TCP服务器连接的各个系统调用

- 建立TCP服务器连接的过程中主要通过以下的系统调用序列来获取某些函数，这些系统调用主要包括有：
  socket() 创建套接字
  bind() 绑定本机端口
  connent() 建立连接 （TCP三次握手在调用这个函数时进行）
  lisiten() 监听端口
  accept() 接收连接
  recv()、read()、recvfrom() 数据接收
  send()、write()、sendto() 数据发送
  close()、shutdown() 关闭套接字 使用close()时，只有当套接字的引用计数为0的时候才会终止连接，而用shutdown()就可以直接关闭连接
  参考：
  建立TCP 服务器的系统调用
  网络编程Socket之TCP之close/shutdown详解
  TCP连接与断开详解（socket通信）

2.12、TCP三次握手时第一次的seq序号是如何产生的?

- 第一次的序号是随机序号，但也不是完全随机，它是使用一个ISN算法得到的。
- seq = C + H (源IP地址，目的IP地址，源端口，目的端口)。其中，C是一个计时器，每隔一段时间值就会变大，H是消息摘要算法，输入是一个四元组（源IP地址，目的IP地址，源端口，目的端口）。

2.13、一个机器能够使用的端口号上限是多少，为什么？可以改变吗？那如果想要用的端口超过这个限制怎么办?

- 65536.因为TCP的报文头部中源端口号和目的端口号的长度是16位，也就是可以表示2^16=65536个不同端口号，因此TCP可供识别的端口号最多只有65536个。但是由于0到1023是知名服务端口，所以实际上还要少1024个端口号。
- 而对于服务器来说，可以开的端口号与65536无关，其实是受限于Linux可以打开的文件数量，并且可以通过MaxUserPort来进行配置。

2.14、服务器出现大量close_wait的连接的原因以及解决方法

- close_wait状态是在TCP四次挥手的时候服务器收到FIN但是没有发送自己的FIN时出现的，服务器出现大量close_wait状态的原因有两种：

  1. 服务器内部业务处理占用了过多时间，都没能处理完业务；或者还有数据需要发送；或者服务器的业务逻辑有问题，没有执行close()方法；
  2. 服务器的父进程派生出子进程，子进程继承了socket，收到FIN的时候子进程处理但父进程没有处理该信号，导致socket的引用不为0无法回收。

- 解决方法：

  1. 停止应用程序
  2. 修改程序里的bug

2.15、TCP的黏包和避免

- 黏包：因为TCP为了减少额外开销，采取的是流式传输，所以接收端在一次接收的时候有可能一次接收多个包。而TCP粘包就是发送方的若干个数据包到达接收方的时候粘成了一个包。多个包首尾相接，无法区分。

- 导致TCP粘包的原因有三方面：

    - 发送端等待缓冲区满才进行发送，造成粘包
    - 接收方来不及接收缓冲区内的数据，造成粘包
    - 由于TCP协议在发送较小的数据包的时候，会将几个包合成一个包后发送

- 避免黏包的措施：

    - 发送定长包。如果每个消息的大小都是一样的，那么在接收对等方只要累计接收数据，直到数据等于一个定长的数值就将它作为一个消息。
    - 包头加上包体长度。包头是定长的 4 个字节，说明了包体的长度。接收对等方先接收包头长度，依据包头长度来接收包体。
    - 在数据包之间设置边界，如添加特殊符号 \r\n 标记。FTP 协议正是这么做的。但问题在于如果数据正文中也含有 \r\n，则会误判为消息的边界。
    - 使用更加复杂的应用层协议。

2.16、TCP的封包与拆包

- 封包：封包就是在发送数据报的时候为每个TCP数据包加上一个包头，将数据报分为包头和包体两个部分。包头是一个固定长度的结构体，里面包含该数据包的总长度。
- 拆包：接收方在接收到报文后提取包头中的长度信息进行截取。

详见：TCP的封包与拆包

3、HTTP协议

3.1、网页的解析过程与实现方法

- 浏览器解析服务器响应过程:
- 首先是html文档解析，浏览器会将html文档生成解析树，也就是DOM树，它由dom元素以及属性节点组成。
- 然后浏览器加载过程中如果遇到了外部的css文件或者图片资源，还会另外发送请求来获取css文件和资源，这个请求通常是异步的，不会影响html文档的加载。
- 如果浏览器在加载时遇到了js文件，则会挂起渲染的线程，等js文件加载解析完毕才恢复html的渲染线程。
- 然后是css解析，将css文件解析为央视表对象来渲染DOM树。

3.2、在浏览器中输入URL后执行的全部过程（如www.baidu.com）（重点，热门问题）

- 1、首先是域名解析，客户端使用DNS协议将URL解析为对应的IP地址；
- 2、然后建立TCP连接，客户端与服务器端通过三次握手建立TCP连接；
- 3、接着是http连接，客户端向服务器发送http连接请求；（http连接无需额外连接，直接通过已经建立的TCP连接发送）
- 4、服务器对客户端发来的http请求进行处理，并返回响应；
- 5、客户端接收到http响应时，将结果渲染展示给用户。

3.3、网络层分片的原因

- 因为在链路层中帧的大小通常都有限制，如在以太网中最大大小（MTU）就是1500字节。如果IP数据包加上头部后大小超过1500字节，就需要分片。
- IP分片和完整IP报文拥有差不多的IP头，16位ID域对于每个分片都是一致的，这样才能在重新组装时识别出来自同一报文的分片。在IP头中，16标识号唯一记录了一个IP包的ID，具有同一ID的IP分片将会重新组装起来；而13位片位移则记录了某IP分片相对整个IP包的位置，而这两个标识中间的三位标志则标志着该分片后是否还有新的分片。这三个标志就组成了IP分片的所有信息，接收方就可以利用这些信息对IP数据进行重新组织。

3.4、http协议和TCP的区别和联系

- 联系：HTTP是建立在TCP基础上的。当浏览器需要从服务器获取网页数据时，会发出一次http请求。http会通过TCP建立从客户端到服务器的连接通道，当本次请求的数据传输完毕后，http会立即断开TCP连接，这个过程非常短暂。
- 区别：两个协议位于不同的层次。tcp协议是传输层的，定义了数据传输和连接的规范。http协议是应用层的。定义了数据的内容的规范。

3.5、http/1.0和http:1.1的区别。

- HTTP 协议老的标准是 HTTP/1.0 ，目前最通用的标准是 HTTP/1.1 。
- HTTP1.0 只保持短暂的连接，浏览器的每次请求都需要与服务器建立一个 TCP 连接，但是最新的http/1.0加入了长连接，只需要在客户端给服务器发送的http报文头部加入Connection:keep-alive
- HTTP 1.1 支持持久连接，默认进行持久连接，在一个 TCP 连接上可以传送多个 HTTP 请求和响应，减少了建立和关闭连接的消耗和延迟。

3.5、http的请求方法有哪些？get和post的区别

- HTTP的请求方法包括GET、POST、PUT、DELETE四种基本方法。
- GET和POST的区别：

    1. get方法不会修改服务器上的资源，它的查询是没有副作用的，而post有可能会修改服务器上的资源
    2. get可以保存为书签，可以用缓存来优化，而post不可以
    3. get把请求附在url上，而post把参数附在http包的包体中
    4. 浏览器和服务器一般对get方法所提交的url长度有限制，一般是1k或者2k，而对post方法所传输的参数大小限制为80k到4M不等
    5. post可以传输二进制编码的信息，get的参数一般只支持ASCII
       参考：HTTP请求方式中8种请求方法

3.6、http状态码含义

-200 - 请求成功
-301 - 资源（网页等）被永久转移到其它URL
-404 - 请求的资源（网页等）不存在
-500 - 内部服务器错误
-400 - 请求无效
-403 - 禁止访问

参考：HTTP状态码的含义

3.7、 http和https的区别，由http升级为https需要做哪些操作

- 区别：

1. http 是超文本传输协议，信息是明文传输， https 则是具有安全性的 ssl 加密传输协议
2. http 和 https 使用的是完全不同的连接方式，用的端口也不一样，前者是 80 ，后者是 443
3. http 的连接很简单，是无状态的； HTTPS 协议是由 SSL+HTTP 协议构建的可进行加密传输、身份认证的网络协议，比http 协议安全。
4. https 协议需要到 ca 申请证书，一般免费证书较少，因而需要一定费用

参考：HTTP与HTTPS的区别

3.7.2、HTTP2.0有哪些改动?

多路复用允许同时通过单一的 HTTP/2 连接发起多重的请求-响应消息。
二进制分帧：在应用层(HTTP/2)和传输层(TCP or UDP)之间增加一个二进制分帧层。将所有传输的信息分割为更小的消息和帧（frame）,并对它们采用二进制格式的编码 ，其中 HTTP1.x 的首部信息会被封装到 HEADER frame，而相应的 Request Body 则封装到 DATA frame 里面。二进制分帧主要作用是二进制码鲁棒性高，增强了通信的稳定性。
首部压缩：http1.x的header由于cookie和user agent很容易膨胀，而且每次都要重复发送。http2.0使用encoder来减少需要传输的header大小
服务端推送：http2.0能通过push的方式将客户端需要的内容预先推送过去

3.8、https的具体实现，怎么确保安全性

SSL是传输层的协议

https包括**非对称加密**和**对称加密**两个阶段，在客户端与服务器建立连接的时候使用非对称加密，连接建立以后使用的是对称加密

1. 客户端使用https的URL访问web服务器，要求与服务器建立SSL连接；
2. 服务器接收到客户端请求后，发送一份公钥给客户端，私钥自己保存；
3. 客户端收到公钥后，将自己生成的对称加密的会话密钥进行加密，并传给网站；
4. 网站收到秘文后用私钥解密出会话密钥；
5. Web服务器利用会话密钥加密与客户端之间的通信，这个过程称为对称加密的过程。

注意：服务器第一次传给客户端的公钥其实是CA对网站信息进行加密的数字证书；

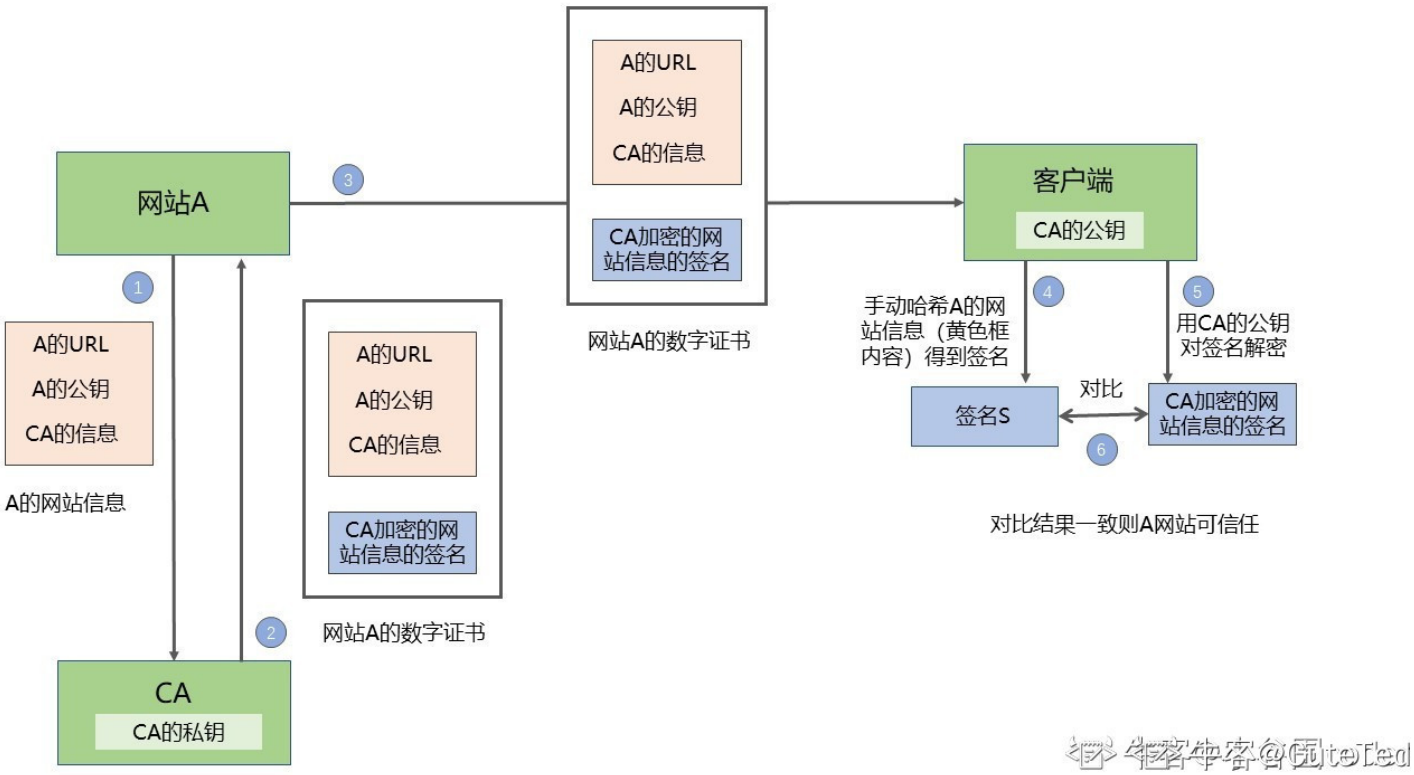
3.9、对称密码和非对称密码体系

- 对称加密：加密和解密用相同的密钥；
    - 优点：计算量小，算法速度快，加密效率高
    - 缺点：密钥容易泄漏。不同的会话需要不同的密钥，管理起来很费劲
    - 常用算法：DES，3DES，IDEA，CR4，CR5，CR6，AES

- 非对称加密：需要公钥和私钥，公钥用来加密，私钥用来解密
    - 优点：安全，不怕泄漏
    - 缺点：速度慢
    - 常用算法：RSA，ECC，DSA

3.10、数字证书的了解（高频）

- 权威CA使用私钥将网站A的信息和消息摘要（签名S）进行加密打包形成数字证书。公钥给客户端。
- 网站A将自己的信息和数字证书发给客户端，客户端用CA的公钥对数字证书进行解密，得到签名S，与手动将网站的信息进行消息摘要得到的结果S*进行对比，如果签名一致就证明网站A可以信任。



4、网络编程

4.1、epoll

详见：如果这篇文章说不清epoll的本质，那就过来掐死我吧！ （3）
罗培羽

系统调用	select	poll	epoll
事件集合	用户通过 3 个参数分别传入感兴趣的可读、可写及异常等事件，内核通过对这些参数的在线修改来反馈其中的就绪事件。这使得用户每次调用 select 都要重置这 3 个参数	统一处理所有事件类型，因此只需一个事件集参数。用户通过 pollfd.events 传入感兴趣的事件，内核通过修改 pollfd.revents 反馈其中就绪的事件	内核通过一个事件表直接管理用户感兴趣的所有事件。因此每次调用 epoll_wait 时，无须反复传入用户感兴趣的事件。epoll_wait 系统调用的参数 events 仅用来反馈就绪的事件
应用程序索引就绪文件描述符的时间复杂度	O(n)	O(n)	O(1)
最大支持文件描述符数	一般有最大值限制	65 535	65 535
工作模式	LT	LT	支持 ET 高效模式
内核实现和工作效率	采用轮询方式来检测就绪事件，算法时间复杂度为 O(n)	采用轮询方式来检测就绪事件，算法时间复杂度为 O(n)	采用回调方式来检测就绪事件，算法时间复杂度为 O(1)

三、操作系统

1、进、线程

1.1、进程与线程的区别与联系

1. **拥有资源**：进程时资源分配的基本单位，而线程时CPU分配和调度的基本单位。
   进程在执行过程中拥有独立的内存单元，而多个线程共享进程的内存。（资源分配给进程，同一进程的所有线程共享该进程的所有资源。同一进程中的多个线程共享代码段（代码和常量），数据段（全局变量和静态变量），扩展段（堆存储）。但是每个线程拥有自己的栈段，栈段又叫运行时段，用来存放所有局部变量和临时变量。
2. **调度**：线程时实现独立调度的基本单位。在同一进程中，线程的切换不会引起进程切换，从一个进程中的线程切换到另一个进程中的线程时，会引起进程切换。
3. **系统开销**：由于创建或撤销进程时，系统都要为之分配或回收资源，如内存空间、I/O 设备等，所付出的开销远大于创建或撤销线程时的开销。类似地，在进行进程切换时，涉及当前执行进程 CPU 环境的保存及新调度进程 CPU 环境的设置，而线程切换时只需保存和设置少量寄存器内容，开销很小。
4. **通信**：程间可以通过直接读写同一进程中的数据进行通信，但是进程通信需要借助 IPC (Inter-Process Communication)。

1.2、 Linux理论上最多可以创建多少个进程？一个进程可以创建多少线程，和什么有关

- 32768. 因为进程的pid是用pid_t来表示的，pid_t的最大值是32768.所以理论上最多有32768个进程。

- 至于线程。进程最多可以创建的线程数是根据分配给调用栈的大小，以及操作系统（32位和64位不同）共同决定的。Linux32位下是300多个。

1.3、进程之间的通信方式

进程通信的方式主要有六种：管道，信号量，消息队列，信号，共享内存，套接字

- 管道：

  管道是半双工的，双方需要通信的时候，需要建立两个管道。管道的实质是一个内核缓冲区，进程以先进先出的方式从缓冲区存取数据：管道一端的进程顺序地将进程数据写入缓冲区，另一端的进程则顺序地读取数据，该缓冲区可以看做一个循环队列，读和写的位置都是自动增加的，一个数据只能被读一次，读出以后再缓冲区都不复存在了。当缓冲区读空或者写满时，有一定的规则控制相应的读进程或写进程是否进入等待队列，当空的缓冲区有新数据写入或慢的缓冲区有数据读出时，就唤醒等待队列中的进程继续读写。管道是最容易实现的
  匿名管道pipe和命名管道除了建立，打开，删除的方式不同外，其余都是一样的。匿名管道只允许有亲缘关系的进程之间通信，也就是父子进程之间的通信，命名管道允许具有非亲缘关系的进程间通信。
  管道的底层实现

- 信号量：

  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。信号量只有等待和发送两种操作。等待(P(sv))就是将其值减一或者挂起进程，发送(V(sv))就是将其值加

一或者将进程恢复运行。

- 信号：

  信号是Linux系统中用于进程之间通信或操作的一种机制，信号可以在任何时候发送给某一进程，而无须知道该进程的状态。如果该进程并未处于执行状态，则该信号就由内核保存起来，直到该进程恢复执行并传递给他为止。如果一个信号被进程设置为阻塞，则该信号的传递被延迟，直到其阻塞被取消时才被传递给进程。 信号是开销最小的。

- 共享内存

  共享内存允许两个或多个进程共享一个给定的存储区，这一段存储区可以被两个或两个以上的进程映射至自身的地址空间中，就像由malloc()分配的内存一样使用。一个进程写入共享内存的信息，可以被其他使用这个共享内存的进程，通过一个简单的内存读取读出，从而实现了进程间的通信。共享内存的效率最高，缺点是没有提供同步机制，需要使用锁等其他机制进行同步。

- 消息队列：

  消息队列就是一个消息的链表，是一系列保存在内核中消息的列表。用户进程可以向消息队列添加消息，也可以向消息队列读取消息。
  消息队列与管道通信相比，其优势是对每个消息指定特定的消息类型，接收的时候不需要按照队列次序，而是可以根据自定义条件接收特定类型的消息。
  可以把消息看做一个记录，具有特定的格式以及特定的优先级。对消息队列有写权限的进程可以向消息队列中按照一定的规则添加新消息，对消息队列有读权限的进程可以从消息队列中读取消息。

- 套接字：

  套接口也是一种进程间通信机制，与其他通信机制不同的是，它可用于不同设备及其间的进程通信。

1.4、进程调度方法

详见：操作系统的常见进程调度算法

- 先来先服务（FCFS）：按照作业到达任务队列的顺序调度FCFS是非抢占式的，易于实现，效率不高，性能不好，有利于长作业（CPU繁忙）而不利于短作业（I/O繁忙）

- 短作业优先（SHF）：次从队列里选择预计时间最短的作业运行。SJF是非抢占式的，优先照顾短作业，具有很好的性能，降低平均等待时间，提高吞吐量。但是不利于长作业，长作业可能一直处于等待状态，出现饥饿现象；完全未考虑作业的优先紧迫程度，不能用于实时系统。

- 最短剩余时间优先：该算法首先按照作业的服务时间挑选最短的作业运行，在该作业运行期间，一旦有新作业到达系统，并且该新作业的服务时间比当前运行作业的剩余服务时间短，则发生抢占；否则，当前作业继续运行。该算法确保一旦新的短作业或短进程进入系统，能够很快得到处理。

- 时间片轮转 用于分时系统的进程调度。基本思想：系统将CPU处理时间划分为若干个时间片（q），进程按照到达先后顺序排列。每次调度选择队首的进程，执行完1个时间片q后，

计时器发出时钟中断请求，该进程移至队尾。以后每次调度都是如此。该算法能在给定的时间内响应所有用户的而请求，达到分时系统的目的。

- 优先级调度：为每个进程分配一个优先级，按优先级进行调度。为了防止低优先级的进程永远等不到调度，可以随着时间的推移增加等待进程的优先级。

- 多级反馈队列：时间片轮转算法对于需要运行较长时间的进程很不友好，假设有一个进程需要执行 100 个时间片，如果采用时间片轮转调度算法，那么需要交换 100 次。因此发展出了多级反馈队列的调度方式。
  多级队列是为这种需要连续执行多个时间片的进程考虑，它设置了多个就绪队列，每个队列时间片大小都不同，例如 ： 1, 2, 4, 8, ... 这样呈指数增长。如果进程在第一个队列没执行完，就会被移到下一个队列。
  在这种情况下，一个需要 100 个时间片才能执行完的进程只需要交换 7 次就能执行完 (1 + 2 + 4 + 8 + 16 + 32 + 64 = 127 > 100）。

1.5、进程的执行过程

详见：进程的加载过程
进程的执行过程需要经过三大步骤：编译，链接和装入。
程序如何运行：编译、链接、装入

- 编译：将源代码编译成若干模块；

- 链接：将编译后的模块和所需的库函数进行链接。
  链接包括三种形式：静态链接，装入时动态链接（将编译后的模块在链接时一边链接一边装入），运行时动态链接（在执行时才把需要的模块进行链接）

- 装入：将模块装入内存运行
  将进程装入内存时，通常使用分页技术，将内存分成固定大小的页，进程分为固定大小的块，加载时将进程的块装入页中，并使用页表记录。减少外部碎片。
  通常操作系统还会使用虚拟内存的技术将磁盘作为内存的扩充。

1.6 多线程和多进程的选择

详见:多进程多线程的区别和选择（总结）

- 频繁修改：需要频繁创建和销毁的优先使用多线程
- 计算量：需要大量计算的优先使用多线程 因为需要消耗大量CPU资源且切换频繁，所以多线程好一点
- 相关性：任务间相关性比较强的用多线程，相关性比较弱的用多进程。因为线程之间的数据共享和同步比较简单。
- 多分布：可能要扩展到多机分布的用多进程，多核分布的用多线程。

1.7、孤儿进程、僵尸进程

详见：孤儿进程与僵尸进程[总结]

- 孤儿进程是父进程退出后它的子进程还在执行，这时候这些子进程就成为孤儿进程。孤儿进程会被init进程收养并完成状态收集。
- 僵尸进程是指子进程完成并退出后父进程没有使用wait()或者waitpid()对它们进行状态收集，这些子进程的进程描述符（PCB）仍然会留在系统中。这些子进程就成为僵尸进程。

1.8、协程

- 协程和微线程是一个东西。
  大多数web服务跟互联网服务本质上大部分都是 IO 密集型服务，IO 密集型服务的瓶颈不在CPU处理速度，而在于尽可能快速的完成高并发、多连接下的数据读写。以前有两种解决方案：
  多进程：存在频繁调度切换问题，同时还会存在每个进程资源不共享的问题，需要额外引入进程间通信机制来解决。
  多线程：高并发场景的大量 IO 等待会导致多线程被频繁挂起和切换，非常消耗系统资源，同时多线程访问共享资源存在竞争问题。
  此时协程出现了，协程 Coroutines 是一种比线程更加轻量级的微线程。类比一个进程可以拥有多个线程，一个线程也可以拥有多个协程。可以简单的把协程理解成子程序调用，每个子程序都可以在一个单独的协程内执行。

- 协程就是子程序在执行时中断并转去执行别的子程序，在适当的时候又返回来执行。
  这种子程序间的跳转不是函数调用，也不是多线程执行，所以省去了线程切换的开销，效率很高，并且不需要多线程间的锁机制，不会发生变量写冲突
  协程运行在线程之上，当一个协程执行完成后，可以选择主动让出，让另一个协程运行在当前线程之上。协程并没有增加线程数量，只是在线程的基础之上通过分时复用的方式运行多个协程，而且协程的切换在用户态完成，切换的代价比线程从用户态到内核态的代价小很多，一般在Python、Go中会涉及到协程的知识，尤其是现在高性能的脚本Go。

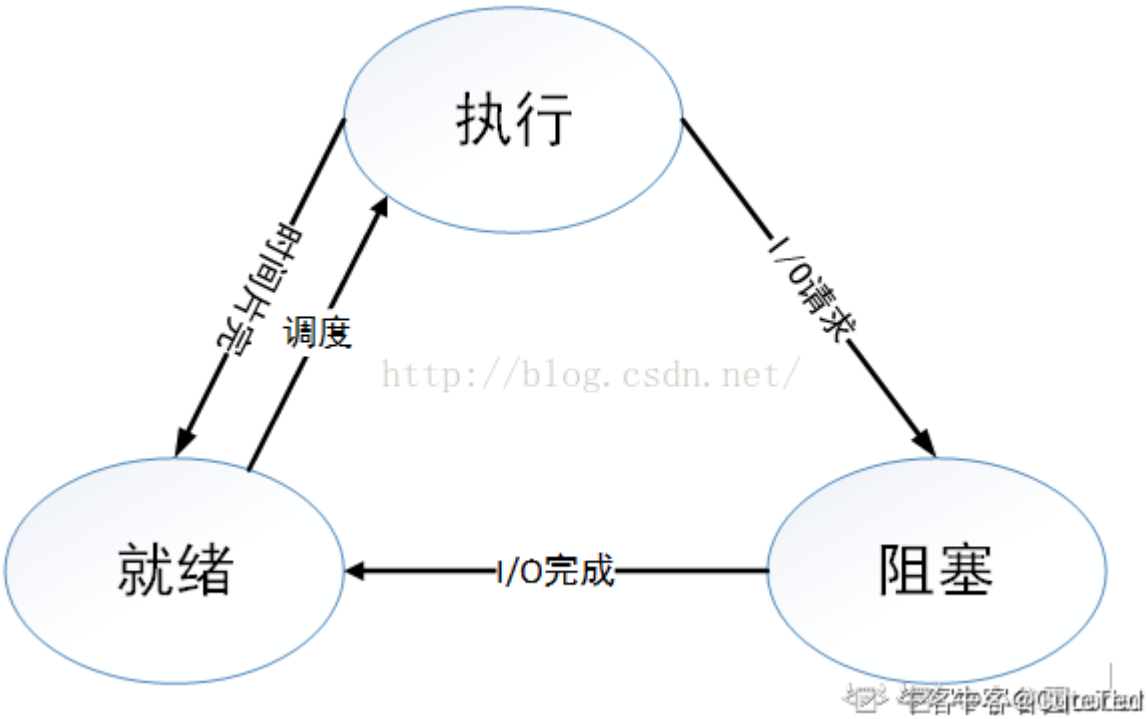
1.9、那协程的底层是怎么实现的，怎么使用协程?

协程进行中断跳转时将函数的上下文存放在其他位置中，而不是存放在函数堆栈里，当处理完其他事情跳转回来的时候，取回上下文继续执行原来的函数。

1.10、进程的状态和转换图

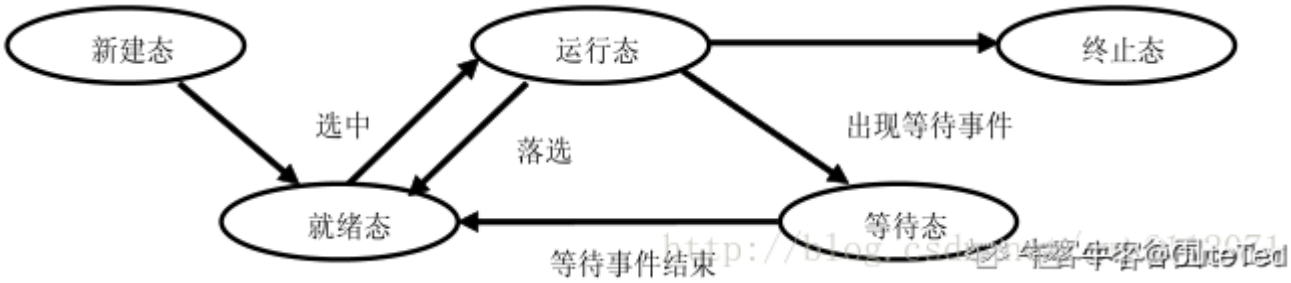
- 三态模型
  三态模型包括三种状态：

1. 执行：进程分到CPU时间片，可以执行
2. 就绪：进程已经就绪，只要分配到CPU时间片，随时可以执行
3. 阻塞：有IO事件或者等待其他资源



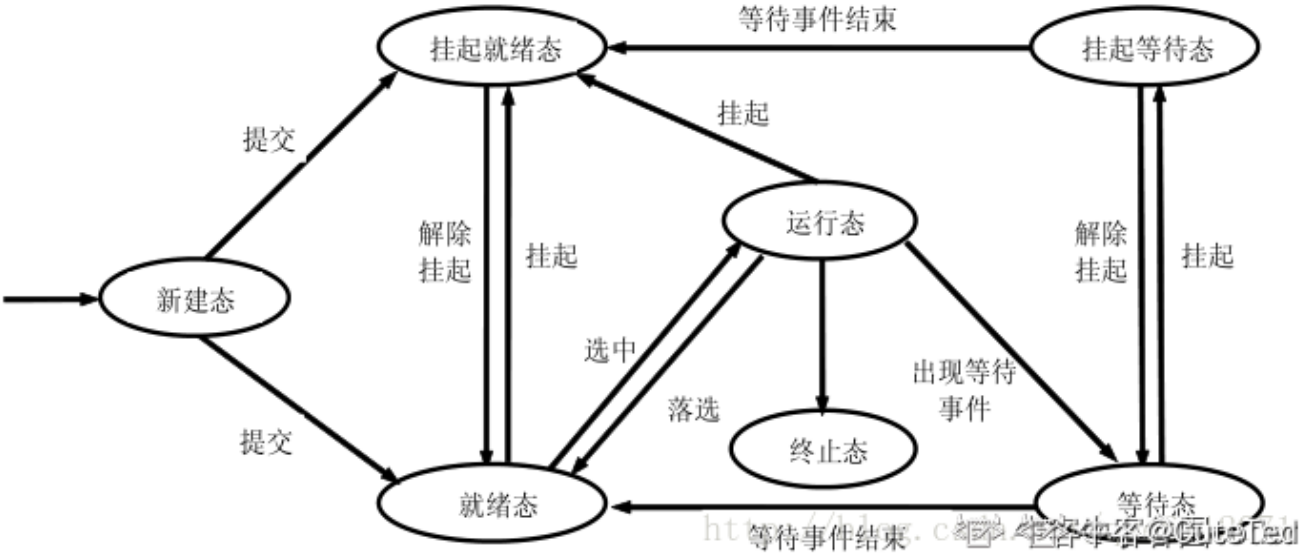
- 五态模型

1. 新建态：进程刚刚创建。
2. 就绪态：
3. 运行态：
4. 等待态：出现等待事件
5. 终止态：进程结束



- 七态模型

1. 新建态
2. 就绪挂起态
3. 就绪态
4. 运行态
5. 等待态
6. 挂起等待态
7. 终止态



1.10、fork和vfork

- fork基础知识：

  fork:创建一个和当前进程映像一样的进程可以通过fork( )系统调用：
  /#include <sys/types.h>
  /#include <unistd.h>
  pid_t fork(void);
  成功调用fork( )会创建一个新的进程，它几乎与调用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。
  最常见的fork( )用法是创建一个新的进程，然后使用exec( )载入二进制映像，替换当前进程的映像。这种情况下，派生（fork）了新的进程，而这个子进程会执行一个新的二进制可执行文件的映像。这种“派生加执行”的方式是很常见的。

- vfork的基础知识：

  在实现写时复制之前，Unix的设计者们就一直很关注在fork后立刻执行exec所造成的地址空间的浪费。BSD的开发者们在3.0的BSD系统中引入了vfork( )系统调用。
  /#include <sys/types.h>
  /#include <unistd.h>
  /pid_t vfork(void);
  除了子进程必须要立刻执行一次对exec的系统调用，或者调用_exit( )退出，对vfork( )的成功调用所产生的结果和fork( )是一样的。vfork( )会挂起父进程直到子进程终止或者运行了一个新的可执行文件的映像。通过这样的方式，vfork( )避免了地址空间的按页复制。在这个过程中，父进程和子进程共享相同的地址空间和页表项。实际上vfork( )只完成了一件事：复制内部的内核数据结构。因此，子进程也就不能修改地址空间中的任何内存。

- 补充知识点：写时复制

  Linux采用了写时复制的方法，以减少fork时对父进程空间进程整体复制带来的开销。
  写时复制是一种采取了惰性优化方法来避免复制时的系统开销。它的前提很简单：如果有多个进程要读取它们自己的那部门资源的副本，那么复制是不必要的。每个进程只要保存一个指向这个资源的指针就可以了。只要没有进程要去修改自己的“副本”，就存在着这样的幻觉：每个进程好像独占那个资源。从而就避免了复制带来的

负担。如果一个进程要修改自己的那份资源“副本”，那么就会复制那份资源，并把复制的那份提供给进程。不过其中的复制对进程来说是透明的。这个进程就可以修改复制后的资源了，同时其他的进程仍然共享那份没有修改过的资源。所以这就是名称的由来：在写入时进行复制。
写时复制的主要好处在于：如果进程从来就不需要修改资源，则不需要进行复制。惰性算法的好处就在于它们尽量推迟代价高昂的操作，直到必要的时刻才会去执行。
在使用虚拟内存的情况下，写时复制（Copy-On-Write）是以页为基础进行的。所以，只要进程不修改它全部的地址空间，那么就不必复制整个地址空间。在fork( )调用结束后，父进程和子进程都相信它们有一个自己的地址空间，但实际上它们共享父进程的原始页，接下来这些页又可以被其他的父进程或子进程共享。
写时复制在内核中的实现非常简单。与内核页相关的数据结构可以被标记为只读和写时复制。如果有进程试图修改一个页，就会产生一个缺页中断。内核处理缺页中断的方式就是对该页进行一次透明复制。这时会清除页面的COW属性，表示着它不再被共享。
现代的计算机系统结构中都在内存管理单元（MMU）提供了硬件级别的写时复制支持，所以实现是很容易的。
在调用fork( )时，写时复制是有很大优势的。因为大量的fork之后都会跟着执行exec，那么复制整个父进程地址空间中的内容到子进程的地址空间完全是在浪费时间：如果子进程立刻执行一个新的二进制可执行文件的映像，它先前的地址空间就会被交换出去。写时复制可以对这种情况进行优化.

- fork和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( )的父子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exit之后父进程才可能被调度运行。
3. vfork( )保证子进程先运行，在它调用exec或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于父进程的进一步动作，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。

i、多线程相关

i.1、多线程线程同步

参考：C++线程同步的四种方式

实现同步的方法：
参考：C++多线程并发（二）---线程同步之互斥锁
C++多线程并发（三）---线程同步之条件变量

线程之间通信的两个基本问题是互斥和同步。

- 线程同步是指线程之间所具有的一种制约关系，一个线程的执行依赖另一个线程的消息，当它没有得到另一个线程的消息时应等待，直到消息到达时才被唤醒。
- 线程互斥是指对于共享的操作系统资源（指的是广义的”资源”，而不是Windows的.res文件，譬如全局变量就是一种共享资源），在各线程访问时的排它性。当有若干个线程都要使用某一共享资源时，任何时刻最多只允许一个线程去使用，其它要使用该资源的线程必须等待，直到占用资源者释放该资源。

i.1.2、C++11多线程之条件变量

- 条件变量是线程的另外一种有效同步机制。这些同步对象为线程提供了交互的场所（一个线程给另外的一个或者多个线程发送消息），我们指定在条件变量这个地方发生，一个线程用于修改这个变量使其满足其它线程继续往下执行的条件，其它线程则等待接收条件已经发生改变的信号。当条件变量同互斥锁一起使用时，条件变量允许线程以一种无竞争的方式等待任意条件的发生。

- 为何引入条件变量
  前一章介绍了多线程并发访问共享数据时遇到的数据竞争问题，我们通过互斥锁保护共享数据，保证多线程对共享数据的访问同步有序。但如果一个线程需要等待一个互斥锁的释放，该线程通常需要轮询该互斥锁是否已被释放，我们也很难找到适当的轮训周期，如果轮询周期太短则太浪费CPU资源，如果轮询周期太长则可能互斥锁已被释放而该线程还在睡眠导致发生延误。
  这就引入了条件变量来解决该问题：条件变量使用“通知—唤醒”模型，生产者生产出一个数据后通知消费者使用，消费者在未接到通知前处于休眠状态节约CPU资源；当消费者收到通知后，赶紧从休眠状态被唤醒来处理数据，使用了事件驱动模型，在保证不误事儿的情况下尽可能减少无用功降低对资源的消耗。
  （参考：https://blog.csdn.net/m0_37621078/article/details/89766449）

i.2、多线程互斥

同步机制：

1. 事件(Event);
2. 信号量(semaphore);
3. 互斥量(mutex);
4. 临界区(Critical section)。

i.3、多线程怎么实现线程安全

详见：【多线程】如何保证线程安全

- 线程安全就是多线程访问时，采用了加锁机制，当一个线程访问该类的某个数据时，进行保护，其他线程不能进行访问直到该线程读取完，其他线程才可使用。不会出现数据不一致或者数据污染。

- 说说线程同步方式有哪些？（线程安全性）
  线程间的同步方式包括互斥锁、信号量、条件变量、读写锁：
  互斥锁：采用互斥对象机制，只有拥有互斥对象的线程才可以访问。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。
  信号量：计数器，允许多个线程同时访问同一个资源。
  条件变量：通过条件变量通知操作的方式来保持多线程同步。
  读写锁：读写锁与互斥量类似。但互斥量要么是锁住状态，要么就是不加锁状态。读写锁一次只允许一个线程写，但允许一次多个线程读，这样效率就比互斥锁要高。

-如何实现线程安全：
保证线程安全以是否需要同步手段分类，分为同步方案和无需同步方案。

- 1、互斥同步（阻塞同步）
  互斥同步是最常见的一种并发正确性保障手段。同步是指在多线程并发访问共享数据时，保证共享数据在同一时刻只被一个线程使用（同一时刻，只有一个线程在操作共享数据）。而互斥是实现同步的一种手段，临界区、互斥量和信号量都是主要的互斥实现方式。因此，在这4个字里面，互斥是因，同步是果；互斥是方法，同步是目的。

互斥同步最主要的问题就是进行线程阻塞和唤醒所带来的性能问题，因此这种同步也成为阻塞同步。从处理问题的方式上说，互斥同步属于一种悲观的并发策略，总是认为只要不去做正确地同步措施（例如加锁），那就肯定会出现问题，无论共享数据是否真的会出现竞争，它都要进行加锁。

- 非阻塞同步
  随着硬件指令集的发展，出现了基于冲突检测的乐观并发策略，通俗地说，就是先进行操作，如果没有其他线程争用共享数据，那操作就成功了；如果共享数据有争用，产生了冲突，那就再采用其他的补偿措施。（最常见的补偿错误就是不断地重试，直到成功为止），这种乐观的并发策略的许多实现都不需要把线程挂起，因此这种同步操作称为非阻塞同步。

i.4、线程池

详见：线程池原理及创建（C++实现）

- 传统多线程方案中我们采用的服务器模型则是一旦接受到请求之后，即创建一个新的线程，由该线程执行任务。任务执行完毕后，线程退出，这就是是“即时创建，即时销毁”的策略。尽管与创建进程相比，创建线程的时间已经大大的缩短，但是如果提交给线程的任务是执行时间较短，而且执行次数极其频繁，那么服务器将处于不停的创建线程，销毁线程的状态。
- 线程池采用预创建的技术，在应用程序启动之后，将立即创建一定数量的线程(N1)，放入空闲队列中。这些线程都是处于阻塞（Suspended）状态，不消耗CPU，但占用较小的内存空间。当任务到来后，缓冲池选择一个空闲线程，把任务传入此线程中运行。当N1个线程都在处理任务后，缓冲池自动创建一定数量的新线程，用于处理更多的任务。在任务执行完毕后线程也不退出，而是继续保持在池中等待下一次的任务。当系统比较空闲时，大部分线程都一直处于暂停状态，线程池自动销毁一部分线程，回收系统资源。

2、操作系统内存管理

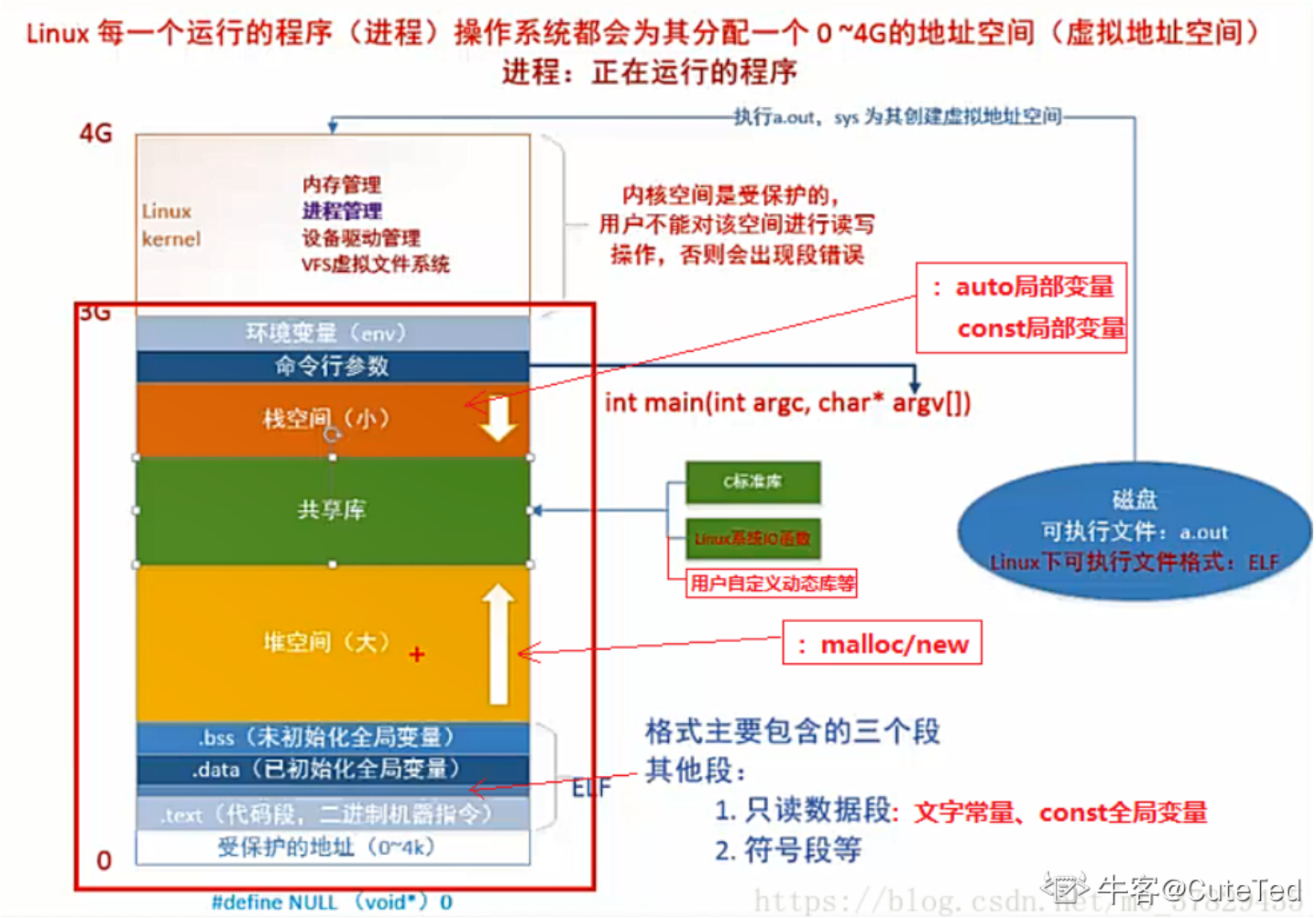
2.1、操作系统的内存管理

详见：
物理内存管理：操作系统内存管理(思维导图详解)
linux虚拟内存管理：Linux 内存管理
【操作系统】总结三（内存管理）
操作系统—内存管理
C/C++内存管理详解

操作系统内存管理：总的来说，操作系统内存管理包括物理内存管理和虚拟内存管理。
操作系统的内存管理包括物理内存管理和虚拟内存管理

- 物理内存管理包括交换与覆盖，分页管理，分段管理和段页式管理等；
- 虚拟内存管理包括虚拟内存的概念，页面置换算法，页面分配策略等；

物理内存管理：
包括程序装入等概念、交换技术、连续分配管理方式和非连续分配管理方式（分页、分段、段页式）。
虚拟内存管理：
虚拟内存管理包括虚拟内存概念、请求分页管理方式、页面置换算法、页面分配策略、工作集和抖动。
这个系列主要使用linux内存管理来具体说明。



2.1.1、物理内存管理

- **程序装入概念**
  详见：程序如何运行：编译、链接、装入

- **交换技术**

交换 (swapping)技术在多个程序并发执行时，可以将暂时不能执行的程序（进程）送到外存中，从而获得空闲内存空间来装入新程序（进程），或读人保存在外存中而处于就绪状态的程序。交换单位为整个进程的地址空间。交换技术常用于多道程序系统或小型分时系统中，因为这些系统大多采用分区存储管理方式。与分区式存储管理配合使用又称作“对换”或“滚进 / 滚出”（roll-in / roll-out)。

原理：暂停执行内存中的进程，将整个进程的地址空间保存到外存的交换区中（换出swap out），而将外存中由阻塞变为就绪的进程的地址空间读入到内存中，并将该进程送到就绪队列（换入swap in）。
交换技术优点之一是增加并发运行的程序数目，并给用户提供适当的响应时间；与覆盖技术相比交换技术另一个显著的优点是不影响程序结构。交换技术本身也存在着不足，例如：对换人和换出的控制增加处理器开销；程序整个地址空间都进行对换，没有考虑执行过程中地址访问的统计特性。

- **覆盖技术**
  引入覆盖 (overlay)技术的目标是在较小的可用内存中运行较大的程序。这种技术常用于多道程序系统之中，与分区式存储管理配合使用。
  覆盖技术的原理：一个程序的几个代码段或数据段，按照时间先后来占用公共的内存空间。将程序必要部分(常用功能)的代码和数据常驻内存；可选部分(不常用功能)平时存放在外存(覆盖文件)中，在需要时才装入内存。不存在调用关系的模块不必同时装入到内存，从而可以相互覆盖。

- **连续分配管理方式**
  连续分配是指为一个用户程序分配连续的内存空间。连续分配有单一连续存储管理和分区式储管理两种方式。

  - **单一连续存储管理**
    在这种管理方式中，内存被分为两个区域：系统区和用户区。应用程序装入到用户区，可使用用户区全部空间。其特点是，最简单，适用于单用户、单任务的操作系统。
  - **分区式存储管理**
    为了支持多道程序系统和分时系统，支持多个程序并发执行，引入了分区式存储管理。分区式存储管理是把内存分为一些大小相等或不等的分区，操作系统占用其中一个分区，其余的分区由应用程序使用，每个应用程序占用一个或几个分区。分区式存储管理虽然可以支持并发，但难以进行内存分区的共享。
    分区式存储管理引人了两个新的问题：**内碎片和外碎片**。
    内碎片是占用分区内未被利用的空间，外碎片是占用分区之间难以利用的空闲分区(通常是小空闲分区)。
    为实现分区式存储管理，操作系统应维护的数据结构为分区表或分区链表。表中各表项一般包括每个分区的起始地址、大小及状态(是否已分配)。

    *1.**固定分区**：固定式分区的特点是把内存划分为若干个固定大小的连续分区。分区大小可以相等：这种作法只适合于多个相同程序的并发执行(处理多个类型相同的对象)。分区大小也可以不等：有多个小分区、适量的中等分区以及少量的大分区。根据程序的大小，分配当前空闲的、适当大小的分区。*
    *2.**动态分区(dynamic partitioning)***
    动态分区的特点是动态创建分区：在装入程序时按其初始要求分配，或在其执行过程中通过系统调用进行分配或改变分区大小。与固定分区相比较其优点是：没有内碎片。但它却引入了另一种碎片——外碎片。动态分区的分区分配就是寻找某个空闲分区，其大小需大于或等于程序的要求。若是大于要求，则将该分区分割成两个分区，其中一个分区为要求的大小并标记为“占用”，而另一个分区为余下部分并标记为“空闲”。分区分配的先后次序通常是从内存低端到高端。
    常用的分区方法：
    *(1). **最先适配法(nrst-fit)**：按分区在内存的先后次序从头查找，找到符合要求的第一个分区进行分配。该算法的分配和释放的时间性能较好，较大的空闲分区可以被保留在内存高端。但随着低端分区不断划分会产生较多小分区，每次分配时查找时间开销便会增大。*
    *(2). **下次适配法(循环首次适应算法 next fit)**：按分区在内存的先后次序，从上次分配的分区起查找(到最后{区时再从头开始}，找到符合要求的第一个分区进行分配。该算法的分配和释放的时间性能较好，使空闲分区分布得更均匀，但较大空闲分区不易保留。*
    *(3). **最佳适配法(best-fit)**：按分区在内存的先后次序从头查找，找到其大小与要求相差最小的空闲分区进行分配。从个别来看，外碎片较小；但从整体来看，会形成较多外碎片优点是较大的空闲分区可以被保留。*
    *(4). **最坏适配法(worst- fit)**：按分区在内存的先后次序从头查找，找到最大的空闲分区进行分配。基本不留下小空闲分区，不易形成外碎片。但由于较大的空闲分区不被保留，当对内存需求较大的进程需要运行时，其要求不易被满足。*

- 非连续分配管理方式（分页、分段、段页式）
  在前面的几种存储管理方法中，为进程分配的空间是连续的，使用的地址都是物理地址。如果允许将一个进程分散到许多不连续的空间，就可以避免内存紧缩，减少碎片。基于这一思

想，通过引入进程的逻辑地址，把进程地址空间与实际存储空间分离，增加存储管理的灵活性。地址空间和存储空间两个基本概念的定义如下：

地址空间：将源程序经过编译后得到的目标程序，存在于它所限定的地址范围内，这个范围称为地址空间。地址空间是逻辑地址的集合。

存储空间:指主存中一系列存储信息的物理单元的集合，这些单元的编号称为物理地址存储空间是物理地址的集合。

根据分配时所采用的基本单位不同，可将离散分配的管理方式分为以下三种：
页式存储管理、**段式存储管理**和**段页式存储管理**。其中段页式存储管理是前两种结合的产物。

- **页式存储**

  基本原理：*将程序的逻辑地址空间划分为固定大小的页(page)，而物理内存划分为同样大小的页框(page frame)。程序加载时，可将任意一页放人内存中任意一个页框，这些页框不必连续，从而实现了离散分配。在页式存储管理方式中地址结构由两部构成，前一部分是页号，后一部分为页内地址w（位移量）。*
  优点:
  1）没有外碎片，每个内碎片不超过页大比前面所讨论的几种管理方式的最大进步是，
  2）一个程序不必连续存放。
  3）便于改变程序占用空间的大小(主要指随着程序运行，动态生成的数据增多，所要求的地址空间相应增长)。
  缺点是：*要求程序全部装入内存，没有足够的内存，程序就不能执行。*

  数据结构：*在页式系统中进程建立时，操作系统为进程中所有的页分配页框。当进程撤销时收回所有分配给它的页框。在程序的运行期间，如果允许进程动态地申请空间，操作系统还要为进程申请的空间分配物理页框。操作系统为了完成这些功能，必须记录系统内存中实际的页框使用情况。操作系统还要在进程切换时，正确地切换两个不同的进程地址空间到物理内存空间的映射。这就要求操作系统要记录每个进程页表的相关信息。为了完成上述的功能，一个页式系统中，一般要采用如下的数据结构。*
  进程页表：*完成逻辑页号(本进程的地址空间)到物理页面号(实际内存空间，也叫块号)的映射。每个进程有一个页表，描述该进程占用的物理页面及逻辑排列顺序。*
  物理页面表：*整个系统有一个物理页面表，描述物理内存空间的分配使用状况，其数据结构可采用位示图和空闲页链表。*
  请求表：*整个系统有一个请求表，描述系统内各个进程页表的位置和大小，用于地址转换也可以结合到各进程的PCB(进程控制块)里。*

  页式管理地址变换:
  在页式系统中，指令所给出的地址分为两部分：逻辑页号和页内地址。
  原理：*CPU中的内存管理单元(MMU)按逻辑页号通过查进程页表得到物理页框号，将物理页框号与页内地址相加形成物理地址(见图4-4)。*
  逻辑页号，页内偏移地址 - >查进程页表，得物理页号 - >物理地址：

- **段式存储**

  基本原理：*在段式存储管理中，将程序的地址空间划分为若干个段(segment)，这样每个进程有一个二维的地址空间。在前面所介绍的动态分区分配方式中，系统为整个进程分配一个连续的内存空间。而在段式存储管理系统中，则为每个段分配一个连续的分区，而进程中的各个段可以不连续地存放在内存的不同分区中。程序加载时，操作系统为所有段分配其所需内存，这些段不必连续，物理内存的管理采用动态分区的*

管理方法。
***在为某个段分配物理内存时**，可以采用首先适配法、下次适配法、最佳适配法等方法。*
***优点**是：没有内碎片，外碎片可以通过内存紧缩来消除；便于实现内存共享。*
***缺点**：与页式存储管理的缺点相同，进程必须全部装入内存。*

***数据结构**：为了实现段式管理，操作系统需要如下的数据结构来实现进程的地址空间到物理内存空间的映射，并跟踪物理内存的使用情况，以便在装入新的段的时候，合理地分配内存空间。*
***进程段表**：描述组成进程地址空间的各段，可以是指向系统段表中表项的索引。每段有段基址(baseaddress)，即段内地址。*
***系统段表**：系统所有占用段（已经分配的段）。*
***空闲段表**：内存中所有空闲段，可以结合到系统段表中。*

***段式管理地址变换**：在段式 管理系统中，整个进程的地址空间是二维的，即其逻辑地址由段号和段内地址两部分组成。为了完成进程逻辑地址到物理地址的映射，处理器会查找内存中的段表，由段号得到段的首地址，加上段内地址，得到实际的物理地址。*

- 页式和段式管理区别

页式和段式系统有许多相似之处。比如，两者都采用离散分配方式，且都通过地址映射机构来实现地址变换。但概念上两者也有很多区别，主要表现在：
1)、需求：是信息的物理单位，分页是为了实现离散分配方式，以减少内存的碎片，提高内存的利用率。或者说，分页仅仅是由于系统管理的需要，而不是用户的需要。段是信息的逻辑单位，它含有一组其意义相对完整的信息。分段的目的是为了更好地满足用户的需要。
一条指令或一个操作数可能会跨越两个页的分界处，而不会跨越两个段的分界处。
2)、大小：页大小固定且由系统决定，把逻辑地址划分为页号和页内地址两部分，是由机器硬件实现的。段的长度不固定，且决定于用户所编写的程序，通常由编译系统在对源程序进行编译时根据信息的性质来划分。
3)、逻辑地址表示：页式系统地址空间是一维的，即单一的线性地址空间，程序员只需利用一个标识符，即可表示一个地址。分段的作业地址空间是二维的，程序员在标识一个地址时，既需给出段名，又需给出段内地址。
4)、比页大，因而段表比页表短，可以缩短查找时间，提高访问速度。

2.1.2、Linux的虚拟内存管理

- **1、基础概念**：

***物理内存**，真实存在的插在主板内存槽上的内存条的容量的大小.*
内存是由若干个存储单元组成的，每个存储单元有一个编号，这种编号可唯一标识一个存储单元，称为内存地址（或物理地址）。我们可以把内存看成一个从0字节一直到内存最大容量逐字节编号的存储单元数组，即每个存储单元与内存地址的编号相对应。

***虚拟内存地址**：就是每个进程可以直接寻址的地址空间，不受其他进程干扰。每个指令或数据单元都在这个虚拟空间中拥有确定的地址。*
***虚拟内存**：就是进程中的目标代码，数据等虚拟地址组成的虚拟空间。*
***虚拟内存与物理内存的区别**：虚拟内存就与物理内存相反，是指根据系统需要从硬盘虚拟地匀出来的内存空间，是一种计算机系统内存管理技术，属于计算机程序，而物理内存为硬件。因为有时候当你处理大的程序时候系统内存不够用，此时就会把硬盘*

当内存来使用，来交换数据做缓存区，不过物理内存的处理速度是虚拟内存的30倍以上。

***逻辑地址**：源程序经过汇编或编译后，形成目标代码，每个目标代码都是以0为基址顺序进行编址的，原来用符号名访问的单元用具体的数据——单元号取代。这样生成的目标程序占据一定的地址空间，称为作业的逻辑地址空间，简称逻辑空间。*
在逻辑空间中每条指令的地址和指令中要访问的操作数地址统称为逻辑地址。即应用程序中使用的地址。要经过寻址方式的计算或变换才得到内存中的物理地址。

***线性地址**：线性地址是逻辑地址到物理地址变换之间的中间层。程序代码会产生逻辑地址，或者说是段中的偏移地址，加上相应段的基地址就生成了一个线性地址。如果启用了分页机制，那么线性地址可以再经变换以产生一个物理地址。若没有启用分页机制，那么线性地址直接就是物理地址。*
*跟逻辑地址类似，它也是一个不真实的地址，如果逻辑地址是对应的**硬件平台段式管理**转换前地址的话，那么线性地址则对应了**硬件页式内存**的转换前地址。*
CPU将一个虚拟内存空间中的地址转换为物理地址，需要进行两步：首先将给定一个逻辑地址（其实是段内偏移量=），CPU要利用其段式内存管理单元，先将为个逻辑地址转换成一个线程地址，再利用其页式内存管理单元，转换为最终物理地址。

- **3、地址映射**：

  *** 虚拟地址向线性地址的转换 **:*

  *** 线性地址向物理地址的转换 **:*
  Linux的每个用户进程都可以访问4 GB的线 性地址空间， 而实际的物理内存可能远远少于4GB． 采用分页机制。Linux仅把可执行映像的一小部分 装入物理 内存。当需要访问未装入的页面时． 系统产生一个缺页中断， 把需要的页读入 物理内存。

- **4、虚拟地址管理**：

  每个用户进程都可以有4 GB的虚存空 间． 为了更好地管 理这部分虚存空间．Linux主要定义了如下三个数据结构：
  struct vm_area_struct ,
  struct vm_operations_struct
  struct vmm_struct
  虚存段(vm_area_struct)． 简称 vma是某个进程的一段 连续的虚存空间． 一个进程通常占用几个vma段． 例如代码段 、 数据段、 堆栈段等 。 vma不仅可以代表一段内存区间， 也可 以对应于一个文件、 共享内存或者对换设备。
  每一个进程的所有vma由一个双向链表管理。 为了提高对vma的查询、 插入、 删除等操作的效率． Linux把系统中所 有进程的 vma组成了一棵 AVL树。 这是一棵平衡二叉树． 当 vma数量特别 大时。 利用这棵 AVL树查找 v ma的效率得到 明显提 高 。
  不同的 vma可能需要不同的操作处理方式． 但同时考虑到统接口的统 一 性． Linux采 用vm_operations_struct结 构和面向对象的思想来定义操作方式． 一个 vm_operations_struct结构体是一组 函数指针， 对于不同的 vma． 它可能指向 不同的处理 函数．例如当发生缺页错误时． 共 享内存和代码 段 的 readpage所 指 向的页面读入函数可能就不同 。
  内存管理中另外 一个 非 常重要的数 据 结 构是vmm_struct 结构体 ． 进程 的 task_struct中的mm成员指向 它． 当前运行进程的整个虚拟空间都 由它来管理和描述 ． 它不仅包含该进程的映像信 息． 而且它的 mma p成员项指向该进 程所有vma组成的链 表 。 它的 mmap_avl 成 员 项 指 向整个系统 的 AVL树 。

- **5、swap对换空间**：

32位Linux系统的每个进程可以有4 GB的虚拟 内存空间 . 而且系统中还要同时存在多个进程 , 但是 , 事实上大多数计算机都没有这么多物理内存空间 , 当系统中的物理内存紧缺时 . 就需要利用对换空间把一部分未来可能不用的页面从物理内存中移到对换设备或对换文件中。
Linux采用两种方式保存换出的页面 :
一种是利用整个块设备 , 如硬盘的一个分区 . 即对换设备,
另一种是利用文件系统中固定长度的文件 . 即对换文件。 它们统称为对换空间。

- **6、分页机制管理**:

  Linux使用分页管理机制来更加有效地利用物理内存．当创建一个进程时．仅仅把当前进程的一小部分真正装入内存．其余部分需要访问时．处理器产生一个页故障．由缺页中断服务程序根据缺页虚拟地址和出错码调用写拷贝函数do—wp—page、此地址所属的vma的vm—ops指向的nopage、do—swap—page．swap—in等函数将需要的页换入物理内存。

  - ***缺页中断和页面换入***
    页面换入主要由缺页中断服务入口函数do—page—fault来实现。当系统中产生页面故障时．如果虚拟内存地址有效．则产生错误的原因有如下两种:
    虚拟内存地址对应的物理页不在内存中。那么它必然在磁盘或对换空间中．如果在磁盘上．那么我们调用do—nO—page函数．而do—no—page调用vma一>vm—ops一>nopage()函数建立页面映射．从对换空间或磁盘中调入页面．或者通过do—swap—page()函数调用swap—in()来换入页面。
    该虚拟地址对应的物理页在内存。但是被写保护．如果这种情况发生在一个共享页面上．则需要“写拷贝“函数do—wp—page来换入页面．do—wp—page函数首先调用一get—free—page获得一新页面．然后调用copy—COW—page拷贝页面的内容．当然还要调用相应的刷新函数刷新TLB和缓存等。

  - ***页交换进程和页面换出***
    正如我们上面所描述的．系统使用kswapd守护进程来定期地换出页面。使系统中有足够的空闲物理内存页。

2.2缺页中断

- 定义：现代操作系统通过虚拟内存技术来扩大物理内存，虚拟内存每一页都映射在物理内存或磁盘上所以虚拟内存会比物理内存大，程序里访问的是虚拟地址，当程序访问页映射在磁盘上时，就会发生缺页中断，调用中断处理程序将页载入物理内存。例如：32位Linux的每个用户进程都可以访问4GB的线性地址空间， 而实际的物理内存可能远远少于4GB． 采用分页机制 ， Linux仅把可执行映像的一小部分装入物理内存。当需要访问未装入的页面时 ．系统产生一个缺页中断，把需要的页读入物理内存。

- *缺页中断**：即指的是当应用程序试图访问已映射在虚拟地址空间中，但是并未被加载在物理内存中的一个分页时，产生一个页不存在的中断，需要操作系统将其调入物理内存后再进行访问。在这个时候，被内存映射的文件（映像）实际上成了一个分页交换文件。

- 缺页中断的次数
  中断次数=进程的物理块数+页面置换次数。
  缺页中断率=（缺页中断次数 / 总访问页数 ）

- 页面置换算法
  当发生缺页中断时，如果操作系统内存中没有空闲页面，则操作系统必须在内存选择一个页面将其移出内存，以便为即将调入的页面让出空间。而用来选择淘汰哪一页的规则叫做页面置换算法。

几种缺页中断算法（FIFO，LRU与LFU）：

- 1)、先进先出（FIFO）
  优先淘汰最早进入内存的页面，亦即在内存中驻留时间最久的页面。该算法实现简单，只需把调入内存的页面根据先后次序链接成队列，设置一个指针总指向最早的页面。但该算法与进程实际运行时的规律不适应，因为在进程中，有的页面经常被访问。

- 2)、最近最久未使用(LRU)置换算法
  选择最近最长时间未访问过的页面予以淘汰，它认为过去一段时间内未访问过的页面，在最近的将来可能也不会被访问。该算法为每个页面设置一个访问字段，来记录页面自上次被访问以来所经历的时间，淘汰页面时选择现有页面中值最大的予以淘汰。

- 3)、LFU（最不经常访问淘汰算法）
  思想：如果数据过去被访问多次，那么将来被访问的频率也更高。
  实现：每个数据块一个引用计数，所有数据块按照引用计数排序，具有相同引用计数的数据块则按照时间排序。每次淘汰队尾数据块。

2.3、实现一个LRU算法

- LRU是什么?
  详见：
  LRU原理和Redis实现——一个今日头条的面试题
  按照英文的直接原义就是Least Recently Used,最近最久未使用法，它是按照一个非常著名的计算机操作系统基础理论得来的：最近使用的页面数据会在未来一段时期内仍然被使用,已经很久没有使用的页面很有可能在未来较长的一段时间内仍然不会被使用。基于这个思想,会存在一种缓存淘汰机制，每次从内存中找到最久未使用的数据然后置换出来，从而存入新的数据！它的主要衡量指标是使用的时间，附加指标是使用的次数。在计算机中大量使用了这个机制，它的合理性在于优先筛选热点数据，所谓热点数据，就是最近最多使用的数据！因为，利用LRU我们可以解决很多实际开发中的问题，并且很符合业务场景。

用到：哈希 + 双向链表
复制代码

```
class LRUCache {
        int cap;
        list<pair<int,int>> l;// front:new back:old 存放值 新的放前面，因为前面的可以取得有效的迭代器
        map<int,list<pair<int,int> >::iterator > cache;// 存放键，迭代器
public:
        LRUCache(int capacity) {
                cap=capacity;
        }

        int get(int key) {
                auto mapitera = cache.find(key);
                if(mapitera==cache.end()){
```

```
                return -1;
            }else{// found
                list<pair<int,int>>::iterator listItera = mapitera->second;
                int value = (*listItera).second;

                l.erase(listItera);
                l.push_front({key,value});
                cache[key]=l.begin();

                return value;
            }
        }

        void put(int key, int value) {
            auto itera = cache.find(key);
            if(itera!=cache.end()){// exist
                list<pair<int,int>>::iterator listItera = itera->second;

                l.erase(listItera);
                l.push_front({key,value});
                cache[key]=l.begin();

            }else{// not exist
                if(cache.size()>=cap){
                    pair<int,int> oldpair = l.back();
                    l.pop_back();
                    cache.erase(oldpair.first);
                }
                l.push_front({key,value});
                cache[key]=l.begin();
            }
        }
};

/**
 * Your LRUCache object will be instantiated and called as such:
 * LRUCache* obj = new LRUCache(capacity);
 * int param_1 = obj->get(key);
 * obj->put(key,value);
 */
```

2.4、内核空间和用户空间是怎样区分的

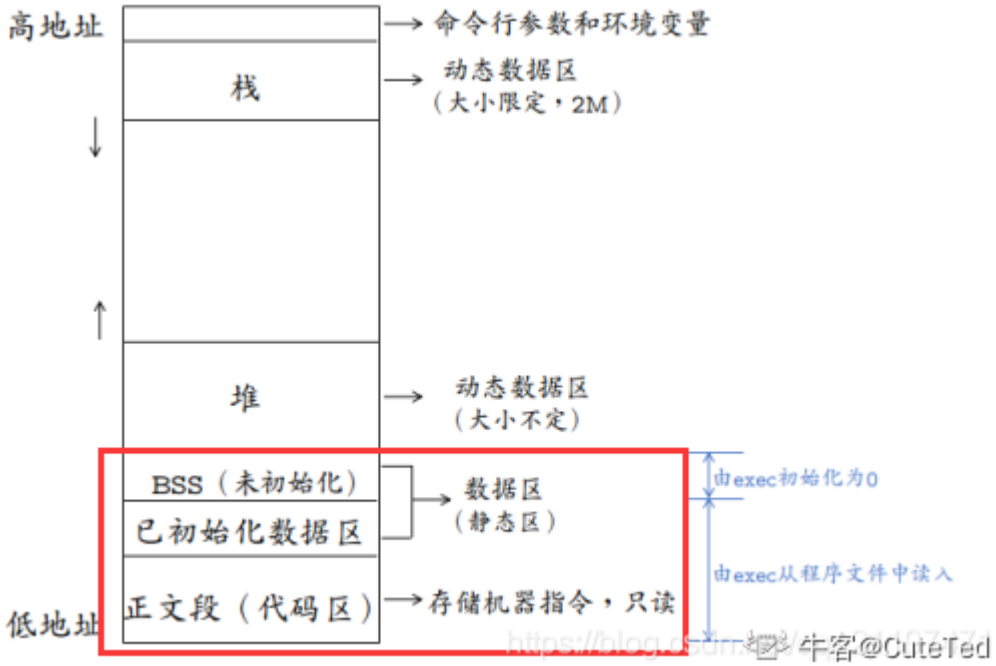
- 在Linux中虚拟地址空间范围为0到4G，最高的1G地址（0xC0000000到0xFFFFFFFF）供内核使用，称为内核空间，低的3G空间（0x00000000到0xBFFFFFFF）供各个进程使用，就是用户空间。
- 内核空间中存放的是内核代码和数据，而进程的用户空间中存放的是用户程序的代码和数据。

2.5、进程内存结构（操作系统中程序的内存结构说明）

- PCB就是进程控制块，是操作系统中的一种数据结构，用于表示进程状态，操作系统通过PCB对进程进行管理。
  PCB中包含有：进程标识符，处理器状态，进程调度信息，进程控制信息

进程地址空间内有：

1. 代码段text：存放程序的二进制代码。通常是指用来存放程序执行代码的一块内存区域。这部分区域的大小在程序运行前就已经确定，并且内存区域通常属于只读, 某些架构也允许代码段为可写，即允许修改程序。在代码段中，也有可能包含一些只读的常数变量，例如字符串常量等。
2. 初始化的数据Data：已经初始化的变量和数据。通常是指用来存放程序中已初始化的全局变量的一块内存区域。数据段属于静态内存分配。
3. 未初始化的数据BSS：还没有初始化的数据。（bss segment）通常是指用来存放程序中未初始化的全局变量的一块内存区域。BSS是英文Block Started by Symbol的简称。BSS段属于静态内存分配。bss段（未进行初始化的数据）的内容并不存放在磁盘上的程序文件中。其原因是内核在程序开始运行前将它们设置为0。需要存放在程序文件中的只有正文段和初始化数据段。text段和data段在编译时已经分配了空间，而BSS段并不占用可执行文件的大小，它是由链接器来获取内存的。
4. 堆：堆是用于存放进程运行中被动态分配的内存段，它的大小并不固定，可动态扩张或缩减。当进程调用malloc等函数分配内存时，新分配的内存就被动态添加到堆上；当利用free等函数释放内存时，被释放的内存从堆中被剔除。
   这里区别 堆区：用于动态分配内存，位于BSS和栈中间的地址区域。由程序员申请分配和释放。堆是从低地址位向高地址位增长，采用链式存储结构。频繁的malloc/free造成内存空间的不连续，产生碎片。当申请堆空间时库函数是按照一定的算法搜索可用的足够大的空间。因此堆的效率比栈要低的多。
5. 栈：栈又称堆栈， 是用户存放程序临时创建的局部变量，也就是说我们函数括弧“{}”中定义的变量（但不包括static声明的变量，static意味着在数据段中存放变量）。除此以外，在函数被调用时，其参数也会被压入发起调用的进程栈中，并且待到调用结束后，函数的返回值也会被存放回栈中。由于栈的先进先出特点，所以栈特别方便用来保存/恢复调用现场。从这个意义上讲，我们可以把堆栈看成一个寄存、交换临时数据的内存区。
   这里区别 栈区：由编译器自动释放，存放函数的参数值、局部变量等。每当一个函数被调用时，该函数的返回类型和一些调用的信息被存放到栈中。然后这个被调用的函数再为他的自动变量和临时变量在栈上分配空间。每调用一个函数一个新的栈就会被使用。栈区是从高地址位向低地址位增长的，是一块连续的内存区域，最大容量是由系统预先定义好的，申请的栈空间超过这个界限时会提示溢出，用户能从栈中获取的空间较小。



一个程序在内存上有BSS段，Date段、text段三个组成的。在没有调入内存前，可执行程序分为代码段，数据区和未初始化数据三部分。

2.6、在执行malloc申请内存的时候，操作系统是怎么做的？/内存分配的原理说一下/malloc函数底层是怎么实现的？/进程是怎么分配内存的？

详见：进程分配内存的两种方式--brk() 和mmap()（不设计共享内存）

从操作系统层面上看，malloc是通过两个系统调用来实现的： brk和mmap

- brk是将进程数据段(.data)的最高地址指针向高处移动，这一步可以扩大进程在运行时的堆大小
- mmap是在进程的虚拟地址空间中寻找一块空闲的虚拟内存，这一步可以获得一块可以操作的堆内存。
  通常，分配的内存小于128k时，使用brk调用来获得虚拟内存，大于128k时就使用mmap来获得虚拟内存。

进程先通过这两个系统调用获取或者扩大进程的虚拟内存，获得相应的虚拟地址，在访问这些虚拟地址的时候，通过缺页中断，让内核分配相应的物理内存，这样内存分配才算完成。

2.6什么是字节序？怎么判断是大端还是小端？有什么用？

详见：字节序：大端法和小端法

字节序是对象在内存中存储的方式，大端即为最高有效位在前面，小端即为最低有效位在前面。
判断大小端的方法：使用一个union数据结构

3、锁

3.1死锁

产生的必要条件：

- 互斥条件：资源同时只能由一个进程占有。
- 不可抢占：不能抢占其他进程的资源，只能等对方开放；
- 占有且申请： 占有且申请：申请其他资源时不放开自己已占有的资源。
- 循环等待：手中拿着对方想要的资源同时向对方要资源。

产生死锁的原因主要是：

- 因为系统资源不足。
- 进程运行推进的顺序不合适。
- 资源分配不当等。

死锁的恢复：

1. 重新启动：是最简单、最常用的死锁消除方法，但代价很大，因为在此之前所有进程已经完成的计算工作都将付之东流，不仅包括死锁的全部进程，也包括未参与死锁的全部进程。
2. 终止进程(process termination)：终止参与死锁的进程并回收它们所占资源。
   (1) 一次性全部终止；(2) 逐步终止(优先级，代价函数)
3. 剥夺资源(resource preemption):剥夺死锁进程所占有的全部或者部分资源。
   (1) 逐步剥夺：一次剥夺死锁进程所占有的一个或一组资源，如果死锁尚未解除再继续剥夺，直至死锁解除为止。
   (2) 一次剥夺：一次性地剥夺死锁进程所占有的全部资源。
4. 进程回退(rollback):让参与死锁的进程回退到以前没有发生死锁的某个点处，并由此点开始继续执行，希望进程交叉执行时不再发生死锁。但是系统开销很大：
   (1) 要实现“回退”，必须“记住”以前某一点处的现场，而现场随着进程推进而动态变化，需要花费大量时间和空间。
   (2) 一个回退的进程应当“挽回”它在回退点之间所造成的影响，如修改某一文件，给其它进程发送消息等，这些在实现时是难以做到的

3.2、死锁预防

- 打破占有且申请： 可以实行资源预先分配策略。即进程在运行前一次性地向系统申请它所需要的全部资源。如果某个进程所需的全部资源得不到满足，则不分配任何资源，此进程暂不运行。只有当系统能够满足当前进程的全部资源需求时，才一次性地将所申请的资源全部分配给该进程。

- 打破循环等待： 实行资源有序分配策略。采用这种策略，即把资源事先分类编号，按号分配，使进程在申请，占用资源时不会形成环路。所有进程对资源的请求必须严格按资源序号递增的顺序提出。进程占用了小号资源，才能申请大号资源，就不会产生环路，从而预防了死锁。

- 安全序列：安全序列是指对当前申请资源的进程排出一个序列，保证按照这个序列分配资源完成进程，不会发生 “酱油和醋” 的尴尬问题。
  我们假设有进程 P1, P2, ... Pn，则安全序列要求满足：Pi (1<=i<=n) 需要资源 <= 剩余资源 + 分配给Pj(1 <= j < i) 资源为什么等号右边还有已经被分配出去的资源？想想银行家那个问题，分配出去的资源就好比第二个开发商，人家能还回来钱，咱得把这个考虑在内。
  银行家算法。

3.4、如何实现一个mutex互斥锁

详见：互斥锁mutex的简单实现

实现mutex最重要的就是实现它的lock()方法和unlock()方法。我们保存一个全局变量flag，flag=1表明该锁已经锁住，flag=0表明锁没有锁住。

实现lock()时，使用一个while循环不断检测flag是否等于1，如果等于1就一直循环。然后将flag设置为1；unlock()方法就将flag置为0；
复制代码

```
static int flag=0;


void lock(){
    while(TestAndSet(&flag,1)==1);
    //flag=1;
}
void unlock(){
    flag=0;
}
//因为while有可能被重入，所以可以用TestandSet()方法。
int TestAndSet(int *ptr, int new) {
        int old = *ptr;
        *ptr = new;
        return old;
}
```

3.5、互斥锁和读写锁区别

- **互斥锁**：mutex，用于保证在任何时刻，都只能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。

- **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。但是同一时刻只能有一个线程可以获得写锁。其它获取写锁失败的线程都会进入睡眠状态，直到写锁释放时被唤醒。 注意：写锁会阻塞其它读写锁。当有一个线程获得写锁在写时，读锁也不能被其它线程获取；写者优先于读者（一旦有写者，则后续读者必须等待，唤醒时优先考虑写者）。适用于读取数据的频率远远大于写数据的频率的场合。

- **互斥锁和读写锁的区别：**
  1）读写锁区分读者和写者，而互斥锁不区分
  2）互斥锁同一时间只允许一个线程访问该对象，无论读写；读写锁同一时间内只允许一个写者，但是允许多个读者同时读对象。

3.6、Linux的4种锁机制

- 互斥锁：mutex，用于保证在任何时刻，都只能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

- 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。但是同一时刻只能有一个线程可以获得写锁。其它获取写锁失败的线程都会进入睡眠状态，直到写锁释放时被唤醒。 注意：写锁会阻塞其它读写锁。当有一个线程获得写锁在写时，读锁也不能被其它线程获取；写者优先于读者（一旦有写者，则后续读者必须等待，唤醒时优先考虑写者）。适用于读取数据的频率远远大于写数据的频率的场合。

- 自旋锁：spinlock，在任何时刻同样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，而是会在原地自旋，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。但如果加锁时间过长，则会非常浪费CPU资源。

- 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，不会导致锁竞争，因此就不用考虑死锁问题了。而对于写者的同步开销较大，它需要复制被修改的数据，还必须使用锁机制同步并行其它写者的修改操作。在有大量读操作，少量写操作的情况下效率非常高。

4、高并发

4.1、服务器高并发的解决方案

1. 应用数据和静态资源分离
   将静态资源（图片，视频，js，css等）单独保存到专门的静态资源服务器中，在客户端访问的时候从静态资源服务器中返回静态资源，从主服务器中返回应用数据。

2. 客户端缓存
   因为效率最高，消耗资源最小的就是纯静态的html页面，所以可以把网站上的页面尽可能用静态的来实现，在页面过期或者有数据更新之后再将页面重新缓存。或者先生成静态页面，然后用ajax异步请求获取动态数据。

3. 集群和分布式
   （集群是所有的服务器都有相同的功能，请求哪台都可以，主要起分流作用）
   （分布式是将不同的业务放到不同的服务器中，处理一个请求可能需要使用到多台服务器，起到加快请求处理的速度。）
   可以使用服务器集群和分布式架构，使得原本属于一个服务器的计算压力分散到多个服务器上。同时加快请求处理的速度。

4. 反向代理
   在访问服务器的时候，服务器通过别的服务器获取资源或结果返回给客户端。

四、数据库

参考：我的数据库面试笔记

1、数据库基础知识

1.1、关系型非关系型数据的区别

- 关系型数据库的优点：
  - 容易理解。因为它采用了关系模型来组织数据。
  - 可以保持数据的一致性。
  - 数据更新的开销比较小。
  - 支持复杂查询（带where子句的查询）

- 非关系数据库的优点
  - 不需要经过sql层的解析，读写效率高；

- 基于键值对，数据的扩展性好；
- 可以支持多种类型数据的存储，如图片，文档等。

1.2、什么是非关型数据库

- 非关系型的数据库也叫nosql，采用键值对的形式进行存储。读写性能很高，易于扩展。如：Redis,Mongodb，hbase等等。
- 适合非关系型数据库的场景：
  - 日志系统；
  - 地理位置存储；
  - 数据量大；
  - 高可用

1.3、数据库的索引类型

数据库的索引类型可以分为逻辑分类 和 物理分类

- 逻辑分类：
  - 主键索引：当关系表中定义主键时会自主创建主键索引。每张表的主键索引只能有一个，要求主键中的每一个值都唯一，即不可重复，也不能有空值。
  - 唯一索引：数据列不能有重复，可以有空值。一张表可以有多个唯一索引，但每个唯一索引只能有一列。如身份证号，卡号。
  - 普通索引：一张表可以有多个普通索引，可以重复可以为空值。
  - 全文索引：可以加快模糊查询，不常用。

- 物理分类
  - 聚集索引（聚簇索引）：数据在物理存储中的顺序跟索引中数据的逻辑顺序相同，比如以ID建立聚集索引，数据库中id从小到大排列，那么物理存储中该数据的内存地址值也按照从小到大存储。一般是表中的主键索引，如果没有主键索引就会以第一个非空的唯一索引作为聚集索引。一张表只能有一个聚集索引。
  - 非聚集索引：数据在物理存储中的顺序跟索引中数据的逻辑顺序不同。非聚集索引因为无法定位数据所在的行，所以需要扫描两遍索引树。第一遍扫描非聚集索引的索引树，确定该数据的主键ID，然后到主键索引（聚集索引）中寻找相应的数据。

1.4、数据库的事务是怎么实现的?

详见：数据库事务的概念及其实现原理

- 事务是一组逻辑操作的集合。实现事务就是要保证可靠性和并发隔离(ACID)。这些主要靠日志恢复和并发控制完成的。
  - 日志恢复：数据库里有两个日志，一个是redo log，一个是undo log。redo log记录的是已经成功提交的事务操作信息，用来恢复数据，保证事务的持久性。undo log记录的是事务修改之前的数据信息，用来回滚数据，保证事务的原子性。
  - 并发控制：并发控制主要靠读写锁和MVCC（多版本并发控制）来实现。读写锁包括共享锁和排他锁，保证事务的隔离性。MVCC通过为数据添加时间戳来实现。

1.5、 数据库中事务的ACID（四大特性都要能够举例说明，理解透彻，比如原子性和一致性的关联，隔离性不好会出现的问题）

数据库事务是指逻辑上对数据的一种操作，这个事务要么全部成功，要么全部失败

- A：atom 原子性
  事务是一个不可分割的工作单位，这组操作要么全部发生，要么全部不发生。
- C：consistency 一致性
  数据库事务的一致性是指：在事务开始以前，数据库的数据有一个一致的状态。在事务完成以后，数据库中的事务也应该保持这种一致性。事务应该将数据从一个一致性状态转移到另一个一致性状态。如：在银行转账操作前后两个账户的总额应当不变。
- I：isolation 隔离性
  数据库事务的隔离性要求数据库中的事务不会受另一个开发执行的事务的影响，对于数据库中同时执行的每个事务来说，其他事务要么还没开始执行，要么已经执行结束。
  D：durability 持久性
  数据库事务的持久性要求书屋对数据库的改变时永久的，哪怕数据库发生损坏都不会影响到已发生的事务。
  如果事务没有完成，数据库因故断电了，那么重启后也应该是没有执行事务的状态，如果事务已经完成后数据库断电了，那么重启后就应该是事务执行完成后的状态。

1.6、脏读、不可重复读、幻读

详见：数据库的事务隔离级别总结

- 脏读： 一个事务在处理过程中读取了另外一个还没提交的事务的数据。
- 不可重复读：对于数据库的某一个字段，一个事务多次拆线呢却返回了不同的值，只是由于早查询的间隔中，该字段被其他事务修改并提交了。
- 幻读：事务多次读取同一个范围的时候，查询结果的记录数不一样，这是由于在查询的间隔中，另一个事务新增或删除了数据。
- 避免不可重复读需要锁行，避免幻读则需要锁表。

1.7、数据库的隔离级别，mysql和Oracle的隔离级别分别是什么（重点）

为了保证数据库事务一致性，解决脏读，不可重复读和幻读的问题，数据库的隔离级别一共有四种隔离级别：

- 读未提交 Read Uncommitted: 最低级别的隔离，不能解决以上问题
- 读已提交 Read committed: 可以避免脏读的发生
- 可重复读 Reapeatable read: 确保事务可以多次从一个字段中读取相同的值，在该事务执行期间，禁止其他事务对此字段的更新，可以避免脏读和不可重复读。 通过锁行来实现
- 串行化 Serializaion 最严格的事务隔离机制，要求所有事务被串行执行，可以避免以上所有问题。 通过锁表来实现
  Oracle的默认隔离级别是**读已提交**，实现了四种隔离级别中的读已提交和串行化隔离级别
  MySQL的默认隔离级别是可重复读，并且实现了所有四种隔离级别

1.8、数据库的三大范式

详见：数据库设计三大范式

- 第一范式(确保每列保持原子性)
  第一范式是最基本的范式。如果数据库表中的所有字段值都是不可分解的原子值，就说明该数据库表满足了第一范式。
- 第二范式(确保表中的每列都和主键相关)
  在满足第一范式的前提下，（主要针对联合主键而言）第二范式需要确保数据库表中的每一

列都和主键的所有成员直接相关，由整个主键才能唯一确定，而不能只与主键的某一部分相关或者不相关。
- 第三范式(确保非主键的列没有传递依赖)
  在满足第二范式的前提下，第三范式需要确保数据表中的每一列数据都和主键直接相关，而不能间接相关。非主键的列不能确定其他列，列与列之间不能出现传递依赖。
- BCNF范式（确保主键之间没有传递依赖）
  主键有可能是由多个属性组合成的复合主键，那么多个主键之间不能有传递依赖。也就是复合主键之间谁也不能决定谁，相互之间没有关系。

1.9、数据库的锁的种类，加锁的方式

以MySQL为例

- 按照类型来分有乐观锁和悲观锁
- 根据粒度来分有行级锁，页级锁，表级锁（粒度一个比一个大） （仅BDB，Berkeley Database支持页级锁）
- 根据作用来分有共享锁（读锁）和排他锁（写锁）

1.10、 什么是共享锁和排他锁

- 共享锁是读操作的时候创建的锁，一个事务对数据加上共享锁之后，其他事务只能对数据再加共享锁，不能进行写操作直到释放所有共享锁。
- 排他锁是写操作时创建的锁，事务对数据加上排他锁之后其他任何事务都不能对数据加任何的锁（即其他事务不能再访问该数据）

1.11、 乐观锁与悲观锁

- 一般的数据库都会支持并发操作，在并发操作中为了避免数据冲突，所以需要对数据上锁，乐观锁和悲观锁就是两种不同的上锁方式。

- 悲观锁假设数据在并发操作中一定会发生冲突，所以在数据开始读取的时候就把数据锁住。而乐观锁则假设数据一般情况下不会发生冲突，所以在数据提交更新的时候，才会检测数据是否有冲突。

- 悲观锁的实现：悲观锁有行级锁和页级锁两种形式。行级锁对正在使用的单条数据进行锁定，事务完成后释放该行数据，而页级锁则对整张表进行锁定，事务正在对该表进行访问的时候不允许其他事务并行访问。
  悲观锁要求在整个过程中一直与数据库有一条连接，因为上一个事务完成后才能让下一个事务执行，这个过程是串行的。

- 乐观锁有三种常用的实现形式：

  - 一种是在执行事务时把整个数据都拷贝到应用中，在数据更新提交的时候比较数据库中的数据与新数据，如果两个数据一摸一样则表示没有冲突可以直接提交，如果有冲突就要交给业务逻辑去解决。
  - 一种是使用版本戳来对数据进行标记，数据每发生一次修改，版本号就增加1。某条数据在提交的时候，如果数据库中的版本号与自己的一致，就说明数据没有发生修改，否则就认为是过期数据需要处理。
  - 最后一种采用时间戳对数据最后修改的时间进行标记。

2、MySQL

2.1 说一下MySQL执行一条查询语句的内部执行过程?

- 连接器：客户端首先通过连接器连接到MySQL服务器。
- 缓存：连接器经过权限验证后，先查询之前是否有执行过此语句（有缓存），若有则，直接返回缓存数据，若无，进入分析器。
- 分析器：分析器会对查询语句进行语法分析和词法分析，判断 SQL 语法是否正确，如果查询语法错误会直接返回给客户端错误信息，如果语法正确则进入优化器。
- 优化器：优化器是对查询语句进行优化处理，例如一个表里面有多个索引，优化器会判别哪个索引性能更好。
- 执行器：优化器执行完就进入执行器，执行器就开始执行语句进行查询比对了，直到查询到满足条件的所有数据，然后进行返回。
  详见： 当程序执行一条查询语句时，MySQL内部到底发生了什么？ （说一下 MySQL 执行一条查询语句的内部执行过程?

2.2、MySQL怎么建立索引，怎么建立主键索引，怎么删除索引?

- 建立索引： alter table 或者 create index
  复制代码

```
alter table table_name add primary key(column_list) #添加一个主键索引
alter table table_name add index (column_list)          #添加一个普通索引
alter table table_name add unique (column_list)        #添加一个唯一索引
create index index_name on table_name (column_list)    #创建一个普通索引
create unique index_name on table_name (column_list)   #创建一个唯一索引
```

- Mysql删除索引同样也有两种方式：alter table 和 drop index
  复制代码

```
alter table table_name drop index index_name      #删除一个普通索引
alter table table_name drop primary key               #删除一个主键索引
drop index index_name on table table_name
```

2.3、MySQL的优化（高频）

- 高频访问：
  分表分库：将数据库表进行水平拆分，减少表的长度
  增加缓存： 在web和DB之间加上一层缓存层
  增加数据库的索引：在合适的字段加上索引，解决高频访问的问题
- 并发优化：
  主从读写分离：只在主服务器上写，从服务器上读
  负载均衡集群：通过集群或者分布式的方式解决并发压力

2.4 MySQL数据库引擎介绍，innodb和myisam的特点和区别

- InnoDB： InnoDB是mysql的默认引擎，支持事务和外键，支持容灾恢复。适合更新频繁和多并发的表 行级锁
- MyISAM： 插入和查询速度比较高，支持大文件，但是不支持事务，适合在web和数据仓库场景下使用 表级锁
- MEMORY： memory将表中的数据保存在内存里，适合数据比较小而且频繁访问的场景
- CSV
- blackhole

3、索引

3.1、索引的优缺点，什么时候使用索引，什么时候不能使用索引（重点）

- 什么时候适合使用索引
    - 经常搜索的列上建索引；
    - 作为主键的列上需要建索引
    - 经常需要连接（where）的列上
    - 经常需要排序的列
    - 进场需要范围插着列
- 那些列不适合建索引
    - 很少查询的列
    - 更新很频繁的列
    - 数据的可取值比较少的列

3.2、索引的低层实现（重点）

- 数据库的索引是用B+树实现的；
- B+树是一种特殊的平衡多路树，是B树的优化改进版本，它把所有的数据都存放在叶节点上，中间节点保存的是索引。这样一来相对于B树来说，减少了数据对中间节点的空间占用，使得中间节点可以存放更多的指针，使得树变得更矮，深度更小，从而减少查询的磁盘IO次数，提高查询效率。另一个是由于叶节点之间有指针连接，所以可以进行范围查询，方便区间访问。
- 而红黑树是二叉的，他的深度相对于B+树来说更大，更大的深度意味着查找的次数更多，更频繁的磁盘IO，所以红黑树更适合在内存中进行查找。

3.3、B树和B+树二点区别（重点）

1. 关键字的数量不同；B+树中分支结点有m个关键字，其叶子结点也有m个，其关键字只是起到了一个索引的作用，但是B树虽然也有m个子结点，但是其只拥有m-1个关键字。
2. 存储的位置不同；B+树中的数据都存储在叶子结点上，也就是其所有叶子结点的数据组合起来就是完整的数据，但是B树的数据存储在每一个结点中，并不仅仅存储在叶子结点上。
3. 分支结点的构造不同；B+树的分支结点仅仅存储着关键字信息和儿子的指针（这里的指针指的是磁盘块的偏移量），也就是说内部结点仅仅包含着索引信息。
4. 查询不同；B树在找到具体的数值以后，则结束，而B+树则需要通过索引找到叶子结点中的数据才结束，也就是说B+树的搜索过程中走了一条从根结点到叶子结点的路径。

- B+树优点：由于B+树的数据都存储在叶子结点中，分支结点均为索引，方便扫库，只需要扫一遍叶子结点即可，但是B树因为其分支结点同样存储着数据，我们要找到具体的数据，需要进行一次中序遍历按序来扫，所以B+树更加适合在区间查询的情况，所以通常B+树用于数据库索引，而B树则常用于文件索引。

3.4、索引最左前缀/最左匹配

- 假如我们对a b c三个字段建立了联合索引，在联合索引中，从最左边的字段开始，任何连续的索引都能匹配上，当遇到范围查询的时候停止。比如对于联合索引index(a,b,c),能匹配a,ab,abc三组索引。并且对查询时字段的顺序没有限制，也就是a,b,c; b,a,c; c,a,b; c,b,a都可以匹配。

3.5 各种树形结构

参考：浅谈AVL树,红黑树,B树,B+树原理及应用

3.5.1 AVL树（平衡二叉树）

- 红黑树是在AVL树的基础上提出来的。
  平衡二叉树又称为AVL树，是一种特殊的二叉排序树。其左右子树都是平衡二叉树，且左右子树高度之差的绝对值不超过1。
  AVL树中所有结点为根的树的左右子树高度之差的绝对值不超过1。
  将二叉树上结点的左子树深度减去右子树深度的值称为平衡因子BF，那么平衡二叉树上的所有结点的平衡因子只可能是-1、0和1。只要二叉树上有一个结点的平衡因子的绝对值大于1，则该二叉树就是不平衡的。

3.5.2 红黑树

详见：红黑树之原理和算法详细介绍

- 对红黑树的理解：通过特殊的要求实现了比较奇怪的结构（数据存放方式），这种结构（存放方式）可以让查找数据变得特别有效，查找方式和常规的二叉查找树查找同。
  红黑树是在AVL树的基础上发展而来的。红黑树是一种二叉查找树，但在每个节点增加一个存储位表示节点的颜色，可以是红或黑（非红即黑）。通过对任何一条从根到叶子的路径上各个节点着色的方式的限制，红黑树确保没有一条路径会比其它路径长出两倍，因此，红黑树是一种弱平衡二叉树，相对于要求严格的AVL树来说，它的旋转次数少，所以对于搜索，插入，删除操作较多的情况下，通常使用红黑树。

- 红黑树较AVL树的优点：
  AVL 树是高度平衡的，频繁的插入和删除，会引起频繁的rebalance，导致效率下降；红黑树不是高度平衡的，算是一种折中，插入最多两次旋转，删除最多三次旋转。
  所以红黑树在查找，插入删除的性能都是O(logn)，且性能稳定，所以STL里面很多结构包括map底层实现都是使用的红黑树。

- 红黑树旋转：
  旋转：红黑树的旋转是一种能保持二叉搜索树性质的搜索树局部操作。有左旋和右旋两种旋转，通过改变树中某些结点的颜色以及指针结构来保持对红黑树进行插入和删除操作后的红黑性质。
  左旋：对某个结点x做左旋操作时，假设其右孩子为y而不是T.nil：以x到y的链为“支轴”进行。使y成为该子树新的根结点，x成为y的左孩子，y的左孩子成为x的右孩子。
  右旋：对某个结点x做右旋操作时，假设其左孩子为y而不是T.nil：以x到y的链为“支轴”进行。使y成为该子树新的根结点，x成为y的右孩子，y的右孩子成为x的左孩子。

3.5.3 B-树

3.5.4 B+树

B+是一种多路搜索树，主要为磁盘或其他直接存取辅助设备而设计的一种平衡查找树，在B+树中，每个节点的可以有多个孩子，并且按照关键字大小有序排列。所有记录节点都是按照键值的大小顺序存放在同一层的叶节点中。相比B树，其具有以下几个特点：

1. 每个节点上的指针上限为2d而不是2d+1（d为节点的出度）
2. 内节点不存储data,只存储key
3. 叶子节点不存储指针

五、linux

0、常用的Linux命令

参考：初窥Linux 之 我最常用的20条命令

1. cd
2. ls
3. find
4. grep:该命令常用于分析一行的信息，若当中有我们所需要的信息，就将该行显示出来，该命令通常与管道命令一起使用，用于对一些命令的输出进行筛选加工等等，它的简单语法为.
5. cp:该命令用于复制文件，copy之意，它还可以把多个文件一次性地复制到一个目录下
6. mv:该命令用于移动文件、目录或更名，move之意，
7. rm:该命令用于删除文件或目录，

-f： 就是force的意思，忽略不存在的文件，不会出现警告消息
-i： 互动模式，在删除前会询问用户是否操作
-r： 递归删除，最常用于目录删除，它是一个非常危险的参数
8. ps命令:该命令用于将某个时间点的进程运行情况选取下来并输出
ps aux # 查看系统所有的进程数据
ps ax # 查看不与terminal有关的所有进程
9. kill命令:该命令用于向某个工作（%jobnumber）或者是某个PID（数字）传送一个信号，它通常与ps和jobs命令一起使用，10. 11、file: 该命令用于判断接在file命令后的文件的基本数据，因为在Linux下文件的类型并不是以后缀为分的，所以这个命令对我们来说就很有用了
11. tar命令: 该命令用于对文件进行打包，默认情况并不会压缩，如果指定了相应的参数，它还会调用相应的压缩程序（如gzip和bzip等）进行压缩和解压。
12. cat命令
该命令用于查看文本文件的内容，后接要查看的文件名，通常可用管道与more和less一起使用，从而可以一页页地查看数据。
13. chmod命令
该命令用于改变文件的权限，
14. vim命令
该命令主要用于文本编辑，它接一个或多个文件名作为参数，如果文件存在就打开，如果文件不存在就以该文件名创建一个文件。

1、Linux的I/O模型介绍以及同步异步阻塞非阻塞的区别（超级重要）

详见：IO同步、异步与多路复用
同步IO、异步IO、阻塞IO、非阻塞IO之间的联系与区别
IO过程包括两个阶段：（1）内核从IO设备读写数据 （2）进程从内核复制数据

- 阻塞：调用IO操作的时候，如果缓冲区空或者满了，调用的进程或者线程就会处于阻塞状态直到IO可用并完成数据拷贝。
- 非阻塞：调用IO操作的时候，内核会马上返回结果，如果IO不可用，会返回错误，这种方式下进程需要不断轮询直到IO可用为止，但是当进程从内核拷贝数据时是阻塞的。
- IO多路复用就是同时监听多个描述符，一旦某个描述符IO就绪（读就绪或者写就绪），就能够通知进程进行相应的IO操作，否则就将进程阻塞在select或者epoll语句上。
- 同步IO：同步IO模型包括阻塞IO，非阻塞IO和IO多路复用。特点就是当进程从内核复制数据的时候都是阻塞的。
- 异步IO：在检测IO是否可用和进程拷贝数据的两个阶段都是不阻塞的，进程可以做其他事情，当IO完成后内核会给进程发送一个信号。

2、EPOLL的介绍和了解

Epoll是Linux进行IO多路复用的一种方式，用于在一个线程里监听多个IO源，在IO源可用的时候返回并进行操作。它的特点是基于事件驱动，性能很高。
epoll将文件描述符拷贝到内核空间后使用红黑树进行维护，同时向内核注册每个文件描述符的回调函数，当某个文件描述符可读可写的时候，将这个文件描述符加入到就绪链表里，并唤起进程，返回就绪链表到用户空间，由用户程序进行处理。
Epoll有三个系统调用：epoll_create(),epoll_ctl()和epoll_wait()。

- eoll_create()函数在内核中初始化一个eventpoll对象，同时初始化红黑树和就绪链表。
- epoll_ctl()用来对监听的文件描述符进行管理。将文件描述符插入红黑树，或者从红黑树中删除，这个过程的时间复杂度是log(N)。同时向内核注册文件描述符的回调函数。
- epoll_wait()会将进程放到eventpoll的等待队列中，将进程阻塞，当某个文件描述符IO可用时，内核通过回调函数将该文件描述符放到就绪链表里，epoll_wait()会将就绪链表里的文件描述符返回到用户空间。

epoll提供了三个函数，epoll_create、epoll_ctl和epoll_wait。
首先创建一个epoll对象，然后使用epoll_ctl对这个对象进行操作（添加、删除、修改），把需要监控的描述符加进去，这些描述符将会以epoll_event结构体的形式组成一颗红黑树，接着阻塞在epoll_wait，进入大循环，当某个fd上有事件发生时，内核将会把其对应的结构体放入一个链表中，返回有事件发生的链表。

epoll为什么高效：
（1）select，poll实现需要自己不断轮询所有fd集合，直到设备就绪，期间可能要睡眠和唤醒多次交替。而epoll只要判断一下就绪链表是否为空就行了，这节省了大量的CPU时间。
（2）select，poll每次调用都要把fd集合从用户态往内核态拷贝一次，并且要把当前进程往设备等待队列中挂一次，而epoll只要一次拷贝，而且把当前进程往等待队列上挂也只挂一次，这也能节省不少的开销。

3、IO复用的三种方法（select,poll,epoll）深入理解，包括三者区别，内部原理实现?

详见：
select、poll、epoll之间的区别总结[整理]
select、poll、epoll之间的区别(搜狗面试)

- （1）select==>时间复杂度O(n)
  select的方法介绍：select把所有监听的文件描述符拷贝到内核中，挂起进程。当某个文件描述符可读或可写的时候，中断程序唤起进程，select将监听的文件描述符再次拷贝到用户空间，然select后遍历这些文件描述符找到IO可用的文件。下次监控的时候需要再次拷贝这些文件描述符到内核空间。select支持监听的描述符最大数量是1024.
  它仅仅知道了，有I/O事件发生了，却并不知道是哪那几个流（可能有一个，多个，甚至全部），我们只能无差别轮询所有流，找出能读出数据，或者写入数据的流，对他们进行操作。所以select具有O(n)的无差别轮询复杂度，同时处理的流越多，无差别轮询时间就越长。
- 缺点是：
  1、单个进程可监视的fd数量被限制，即能监听端口的大小有限。
  一般来说这个数目和系统内存关系很大，具体数目可以cat /proc/sys/fs/file-max察看。32位机默认是1024个。64位机默认是2048.
  2、 对socket进行扫描时是线性扫描，即采用轮询的方法，效率较低：
  当套接字比较多的时候，每次select()都要通过遍历FD_SETSIZE个Socket来完成调度,不管哪个Socket是活跃的,都遍历一遍。这会浪费很多CPU时间。如果能给套接字注册某个回调函数，当他们活跃时，自动完成相关操作，那就避免了轮询，这正是epoll与kqueue做的。

3、需要维护一个用来存放大量fd的数据结构，这样会使得用户空间和内核空间在传递该结构时复制开销大

- （2）poll==>时间复杂度O(n)
  poll使用链表保存文件描述符，其他的跟select没有什么不同。
  poll本质上和select没有区别，它将用户传入的数组拷贝到内核空间，然后查询每个fd对应的设备状态， 但是它没有最大连接数的限制，原因是它是基于链表来存储的.

- 缺点：
  1、大量的fd的数组被整体复制于用户态和内核地址空间之间，而不管这样的复制是不是有意义。

- （3）epoll==>时间复杂度O(1)
  epoll可以理解为event poll，不同于忙轮询和无差别轮询，epoll会把哪个流发生了怎样的I/O事件通知我们。所以我们说epoll实际上是事件驱动（每个事件关联上fd）的，此时我们对这些流的操作都是有意义的。（复杂度降低到了O(1)）
  epoll将文件描述符拷贝到内核空间后使用红黑树进行维护，同时向内核注册每个文件描述符的回调函数，当某个文件描述符可读可写的时候，将这个文件描述符加入到就绪链表里，并唤起进程，返回就绪链表到用户空间。

- epoll的优点：
  1、没有最大并发连接的限制，能打开的FD的上限远大于1024（1G的内存上能监听约10万个端口）；
  2、效率提升，不是轮询的方式，不会随着FD数目的增加效率下降。只有活跃可用的FD才会调用callback函数；
  即Epoll最大的优点就在于它只管你“活跃”的连接，而跟连接总数无关，因此在实际的网络环境中，Epoll的效率就会远远高于select和poll。
  3、 内存拷贝，利用mmap()文件映射内存加速与内核空间的消息传递；即epoll使用mmap减少复制开销。

4、Epoll的ET模式和LT模式（ET的非阻塞）

- ET是边缘触发模式，在这种模式下，只有当描述符从未就绪变成就绪时，内核才会通过epoll进行通知。然后直到下一次变成就绪之前，不会再次重复通知。也就是说，如果一次就绪通知之后不对这个描述符进行IO操作导致它变成未就绪，内核也不会再次发送就绪通知。优点就是只通知一次，减少内核资源浪费，效率高。缺点就是不能保证数据的完整，有些数据来不及读可能就会无法取出。
- LT是水平触发模式，在这个模式下，如果文件描述符IO就绪，内核就会进行通知，如果不对它进行IO操作，只要还有未操作的数据，内核都会一直进行通知。优点就是可以确保数据可以完整输出。缺点就是由于内核会一直通知，会不停从内核空间切换到用户空间，资源浪费严重。

5、coredump是什么 怎么才能coredump

- coredump是程序由于异常或者bug在运行时异常退出或者终止，在一定的条件下生成的一个叫做core的文件，这个core文件会记录程序在运行时的内存，寄存器状态，内存指针和函数堆栈信息等等。对这个文件进行分析可以定位到程序异常的时候对应的堆栈调用信息。
- coredump产生的条件

1. shell资源控制限制，使用 ulimit -c 命令查看shell执行程序时的资源 ，如果为0，则不会产生coredump。可以用ulimit -c unlimited设置为不限大小。
2. 读写越界，包括：数组访问越界，指针指向错误的内存，字符串读写越界

3. 使用了线程不安全的函数，读写未加锁保护
4. 错误使用指针转换
5. 堆栈溢出

6、tcpdump常用命令

用简单的话来定义tcpdump，就是：dump the traffic on a network，根据使用者的定义对网络上的数据包进行截获的包分析工具。 tcpdump可以将网络中传送的数据包的“头”完全截获下来提供分析。它支持针对网络层、协议、主机、网络或端口的过滤，并提供and、or、not等逻辑语句来帮助你去掉无用的信息。

六、场景题目

1、如何实现一个大数运算

- 如果出现了大数的处理，所谓大数，应该是位数达到了数十位甚至几百位。由于没有可用的数据类型来存储数据，首先，数据的接收应该使用字符数组的方法，然后再转换。
  复制代码

```
1、接收
void Input(int a[])
{
    char ch[100];
    int i,j=0;
    scanf("%s",ch);getchar();
    int k=strlen(ch);
    for (i=k-1;i>=0;i--)   //倒叙储存来使低位放在低下标，方便运算
            a[j++]= ch[i]-'0';
}
2、加法
void jia(int a[],int b[])
{
    int i,j,k;
    for (i=0;i<MAX;i++)
    {
            if (a[i]+b[i]>=10)
            {
                    a[i+1]++;
                    a[i]=(a[i]+b[i])%10;
            }
            else
                    a[i]=a[i]+b[i];
    }
}
3、乘法
单位乘法（这种方法可以用于乘一个变量可以存储的类型，比如一个并不算大的数的阶乘，大数乘小数）
void chengl(int *a,int b)
```