**C++相关：**

**描述.cpp转为.exe文件的编译过程？**

整体过程：预处理（.i）->编译（.asm?不确定）->汇编（.obj）->链接（.exe）

其中

预处理是对#后跟随的比如include/define的代码替换，预处理后输出.i文件；

编译对.i文件进行语法、语义分析等，输出汇编代码；

汇编将汇编代码编译为二进制机器码，生成.obj文件；

链接将零散的.obj文件组织起来，整合为.exe文件。

**声明和定义的区别？**

声明是告诉编译器变量或函数的类型和名称等；定义是告诉编译器变量的值，函数具体干什么。

分配内存：声明不分配内存，定义分配内存；

次数：可以声明多次，只能定义一次；

**为什么需要声明？**

1. 避免重复定义：通过声明变量或函数，可以在多个文件中共享同一个标识符，避免多次定义同一个实体而导致链接错误。
2. 分离接口和实现：通过声明，可以将接口（声明）与实现（定义）分离，提高代码的模块化和可重用性。

**const和define的区别？**

1. 类型和安全检查：define是简单的文本替换，没有类型检查，const有；

2. 作用域：define的作用域是从定义到文件尾，常量作用域受到代码块限制；

3. 处理阶段：define在预处理阶段展开，const在编译阶段分配内存，值在编译期（编译期常量）或运行期（运行期常量）确定；

**全局变量和静态全局变量的区别？**

全局变量的作用域是程序中的所有源文件，当然，其他不包含全局变量定义的源文件需要用extern 关键字再次声明这个全局变量；

静态全局变量的只作用在定义的源文件中，不能作用到其他文件。

**关键字volatile？**

与const相反，用来修饰变量，表示变量随时可能改变，每次读取都直接从变量地址中读取；如果没有volatile，编译期可能会优化读取和存储，暂时使用寄存器中的值。

**static和this之间的联系？**

static用于定义静态成员，使得成员可在类的所有对象之间共享；

this是指向当前对象的指针，用于在成员函数中引用当前对象。

**关键字explicit？**

作用于单参构造函数（或除了第一个参数以外的其他参数都有默认值的构造函数），表明构造函数是显示的，禁止了隐式转换,以及禁止调用拷贝构造函数。

例子：explicit Myclass(string s){}

Myclass a = “aa”;正确

Myclass a = 10;错误

**struct和class的区别？**

1. struct表示一组相关的数据（在c中不能包含成员函数），class表示封装了数据和操作的对象；

2. struct成员默认public，class默认private，继承同理；

3. struct不能使用模板，class可以；

**静态成员变量为什么不能在类内初始化？**

在类外定义和初始化是保证static成员变量只被定义一次的好方法；

但静态常量成员变量可在类内初始化。

**constexpr是什么，和const的区别？**

c++11关键字，声明编译期常量和允许在编译期求值的函数；

1. 声明常量：确保在编译期被计算并具有确定的值；

2. 声明函数：在编译期求值，前提是计算所依赖的值也必须能在编译期确定；

区别：

1. const没有区分编译期和运行期常量，即const修饰的变量可以在运行时才初始化，而constexpr一定会在编译期初始化；

2. const的语义为只读，保证修饰的变量在运行时不允许更改，constexpr的语义才更接近常量；

**分别禁止在堆栈创建对象**

1. 禁止在栈上创建：把析构函数设置为私有（我的理解：栈由操作系统管理，栈上创建的对象，其构造和析构由操作系统完成，操作系统用不了私有的析构，就没办法管理这个类的对象，所以没办法在栈上创建私有析构的类对象）。
2. 禁止在堆上创建：把new和delete重载为私有。

**介绍一下智能指针？**

通过raii机制自动管理动态内存，解决动态分配内存导致的内存泄漏和多次释放同一内存的问题；

unique\_ptr通过禁止拷贝语义来持有对象的独有权；如果强行拷贝，在释放资源时会造成对同一内存指针多次释放而导致程序崩溃

shared\_ptr允许多个指针共享一个对象，通过引用计数决定指针析构时是否释放对象；

weak\_ptr协助shared\_ptr工作，构造和析构不引起引用计数的变化，用来观察引用计数和解决循环引用问题。

**为什么智能指针可以自动释放内存？计数器怎么实现，用什么数据结构和方法？**

智能指针通过raii机制实现自动释放内存，其封装了对象的指针和析构操作，使得智能指针离开作用域时，能够自动调用析构函数来释放内存；

以shared\_ptr为例，当创建一个shared\_ptr对象时，其内部会分配两个指针：一个指向管理的对象，另一个指向（用原子整型变量记录的）资源引用计数，通过拷贝构造shared指针时，它们会共享同一个shared引用计数，使其计数+1。shared\_ptr离开作用域时，自动调用析构函数，析构函数将计数器减一，当减为0时释放内存。

**智能指针的缺点；什么是循环引用？怎么解决？**

两个对象互相使用其内部的shared\_ptr指向对方，导致计数器错误，造成内存泄漏；

案例：假设有两个shared\_ptr指针sp1和sp2分别指向A、B类的对象，此时引用计数为1，当类中又分别有一个shared\_ptr指向对方，此时引用计数为2，当sp1和sp2离开作用域时引用计数变为1，却无法正确释放内存导致内存泄露。因此需要在内部使用weak\_ptr指向对方，来解决循环引用造成的内存泄漏。

**讲讲红黑树？**

自平衡的二叉搜索树，在每个节点上增加额外属性来存储节点的颜色。

满足性质（确保平衡性和搜索效率）：

1.每个节点要么红要么黑

2.根节点黑

3.每个叶子结点黑

4.如果一个节点是红，则其两个子节点都是黑（不能有两个连续红节点）

5.从任一节点到其每个叶子的所有路径都包含相同数量的黑节点（确保最长路径不会超过最短路径的两倍）。

平衡性保证最坏情况下的时间复杂度0（logn），C++STL中的map和set等容器的底层实现是红黑树。

**为什么红黑树高效？**

1.平衡性：不严格平衡，相比于严格平衡的AVL树，增删节点的代价较低，同时也能使树高度不会过高，保证增删查的时间复杂度为O（logn）；

2.二叉搜索树性质：有序性使得查找操作更高效；

3.良好的平均性能：最坏情况的时间复杂度也是O（logn），增删查都能在较短时间内完成；

**内存泄露**

一般来说内存泄漏是指堆内存泄漏，即程序在动态分配内存并运行结束之后，没有显示的释放动态内存而导致的泄露，会导致系统中可用的内存减少，性能下降或崩溃。

造成内存泄漏可能的原因有：使用new或malloc后没有相应的delete或free来释放内存；智能指针引起的循环引用；建立的socket连接、数据库连接在程序结束没手动释放等。

**C++面向对象的三个特性？**

封装：将数据和操作封装在一个类中，隐藏实现细节，只暴露必要的接口供外部使用，可以保护数据的完整性；

继承：是实现代码重用的机制，一个类可以继承另一个类的方法和数据，建立类与类之间的关系；

多态：同一操作在不同对象上产生不同的行为（重载实现编译时多态，重写虚函数实现运行时多态）。

**友元**

友元可以是一个函数，它定义在类的外部，不属于类的成员函数但有权访问类的所有私有成员和保护成员。

也可以声明一个友元类，此时友元类可以访问当前整个类中的所有成员。（包括变量和函数）

**菱形继承问题？**

菱形继承：一个类继承了基类相同的两个派生类；

问题：数据冗余（基类的成员有两份）、二义性（调用基类成员函数时，不知道调用哪个）；

解决：虚拟继承，class B : virtual public A。

**如何实现多态？虚函数**

在基类声明或定义虚函数，在派生类中重写该函数，运行时将会根据所指对象的实际类型来调用相应的函数；

底层通过虚函数和虚函数表实现；

每个有virtual或重写了virtual函数的类内部都有虚函数表，表中存储了指向实际函数的指针，函数重写时新的函数指针会替代掉基类对应位置的函数地址，编译器为每个有虚函数表的类的对象的内存起始位置自动创建了一个虚表指针，指向所属类的虚表，运行时在表中动态查找调用的函数。

**纯虚函数：**作为单纯的接口表示该函数没有具体的实现，[要求子类必须复写该函数，以实现多态性](https://zhuanlan.zhihu.com/p/37331092" \t "https://www.bing.com/_blank)

**类中的内存布局？**

1. 如果继承了：先放父类的数据拷贝（包括虚表指针）；

2. 再放子类的数据（父类如果有虚函数，数据前是虚表指针）；虚表先放父类的虚函数，再放子类的虚函数；

注意：对象内存中不存储成员函数，操作是共享的！a.f() = f(&a)；

基于这个内存布局，可以通过对象指针的偏移访问成员变量（包括private的）：

假设类A中有两个int类型和一个double类型的成员变量，那么

A a;

int\* ptr1 = (int\*)&a; // ptr1指向了第一个成员变量

int\* ptr2 = ptr1 + 1; //ptr2指向第二个成员变量

double\* ptr3 = (double\*)(ptr2 + 1); // ptr2偏移一个int大小，到了第三个成员变量的起始位置

注意：指针+1的运算含义为偏移当前指针指向的对象的大小；如果有虚表指针，指向第一个成员变量需要偏移8字节（64位操作系统）或4字节（32位操作系统）；

**内存对齐是什么？为什么？规则？**

元素按照定义顺序一个一个放到内存中去的，但并不是紧密排列的，这就是内存对齐；

原因：

1. 平台原因(移植原因)：不是所有的硬件平台都能访问任意地址上的任意数据的；某些硬件平台只能在某些地址处取某些特定类型的数据，否则抛出异常；

2. 性能原因：访问未对齐的内存，处理器需要作两次内存访问；而对齐的内存访问仅需要一次访问；（可以举例：int变量存放在地址1~5，处理器访问该变量时需要先读取第一个4字节块，剔除不想要的0地址的字节，再从地址4开始读取下一个4字节块，剔除不想要的567地址字节，留下的数据合并放入寄存器）

内存对齐规则：

1. 每个数据成员存储的起始位置从其大小的整数倍开始；

2. 结构体作为成员，存储位置从其内部最大元素大小的整数倍地址开始；（空类占1字节）

3. 类的总大小必须是内部最大成员长度的整数倍；

**构造函数或析构函数中可以调用虚函数吗？**

语法上可以，但效果往往达不到需要的目的；

1. 构造函数：派生类对象构造期进入基类的构造函数时，对象类型变成基类类型，调用的虚函数是基类的，无法多态调用；

2. 析构函数：派生类先析构，这时如果调用了虚函数，只会执行基类的虚函数，无法多态调用。

**C++的资源管理？**

内存、锁、文件、Socket等等都是资源，C++中可以通过RAII的方式管理这些资源。其基本思路为：用类来封装资源，在类的构造函数中获取资源，在类的析构函数中释放资源。需要使用资源时，把相应类实例化出一个对象，当对象超出作用域时，其析构函数会被调用，从而释放资源。智能指针、unique\_lock就是RAII的实现范例。

**解释一下浅拷贝和深拷贝**

浅拷贝：复制的对象中引用了外部内容（如分配在堆上的数据），在复制这个对象的时候会让新旧两个对象指向同一个外部内容。

深拷贝：复制的对象会引用外部对象的独立复制。

**浮点数是怎么存储的**

**float类型**：

[单精度浮点数](https://www.zhihu.com/search?q=%E5%8D%95%E7%B2%BE%E5%BA%A6%E6%B5%AE%E7%82%B9%E6%95%B0&search_source=Entity&hybrid_search_source=Entity&hybrid_search_extra=%7b" \t "https://zhuanlan.zhihu.com/p/_blank)float(32bit)表示为： SEEE EEEE EMMM MMMM MMMM MMMM MMMM MMMM

1）符号位(S)1 bit;

2）阶码(E)移码表示8 bit，阶码的偏移量为127;

3）尾数(M)原码表示23 bit，用小数表示，小数点放在尾数域的最前面

**[双精度浮点数](https://www.zhihu.com/search?q=%E5%8F%8C%E7%B2%BE%E5%BA%A6%E6%B5%AE%E7%82%B9%E6%95%B0&search_source=Entity&hybrid_search_source=Entity&hybrid_search_extra=%7b" \t "https://zhuanlan.zhihu.com/p/_blank)double(64bit)**

1)符号位(S)1 bit;

2)阶码(E)11 bit，阶码的偏移量为1023(3FFH);

3)尾数(M)52 bit，用小数表示，小数点放在尾数域的最前面.

**new和malloc的区别？**

new是C++的运算符，分配内存并调用构造函数；malloc是C语言函数库，只分配指定大小的内存；

new返回具体指针类型；malloc返回void\*，使用时一般需要进行类型转换；

new分配失败时抛出bad\_alloc异常；malloc分配失败时返回NULL；

**new operator、operator new、placement new之间？**

new运算符（不能被重载）：调用operator new分配内存、调用构造函数、返回相应指针；

operator new（可重载）：

1. void\* operator new (std::size\_t size) throw (std::bad\_alloc); // A\* a = new A;

2. void\* operator new (std::size\_t size, const std::nothrow\_t& nothrow\_constant) throw(); // A\* a = new(std::nothrow) A;

3. void\* operator new (std::size\_t size, void\* ptr) throw(); // new (p)A();

operator new在c++中有三个重载，一个是抛出bad\_alloc异常的，一个是不抛异常的，还有一个是placement new，示意在预先分配好的内存（由p指向内存起始地址）中构造对象，placement new本身不再分配内存。

**malloc如何分配内存？**

malloc是C库中的函数，通过系统调用申请堆内存，在返回给用户的地址前会多出16个字节，用于保存内存块的描述信息（free、sizeof等可以通过这个信息判断内存块大小）：

1. brk()：将堆顶指针向高地址移动，获得新的内存空间（而且会预分配更大的空间）【申请内存小于128KB】，free释放内存时，将该内存缓存在malloc的内存池中，下次申请同样内存空间就不用系统调用了；

2. mmap()：在文件映射区分配一块内存空间【申请内存大于128KB】，free释放内存时，会把内存还给操作系统；

为什么不全部用brk()？为什么不全部用mmap()？

全用brk容易产生碎片，导致内存泄漏；

全用mmap，每次都要系统调用，第一次访问申请的内存都会触发缺页中断；

**C++的内存管理方式？**

程序运行时，会开辟一个内存空间包括几个区域，地址从大到小依次是

命令行参数和环境变量；

栈：存放局部变量、函数参数、调用、返回等，向下增长；

共享区：给堆栈增长留存的空间；

堆：动态申请内存，向上增长；

BSS段：存放未初始化的全局变量和静态变量；

数据段：存放已初始化的全局变量和静态变量；

代码段：存放执行代码和只读的常量；

**引用和指针的区别？**

都用于间接访问变量，区别：

指针是一个变量，它保存了另一个变量的内存地址，可以被重新赋值，指向不同的变量；

引用是另一个变量的别名，与原变量共享内存地址，在初始化后不能更改，始终指向同一个变量。

**\*\*右值引用**

右值引用实现了移动语义，允许高效地将资源从一个对象移动到另一个对象；

在没有移动语义之前，转移对象资源需要先将资源拷贝到指定对象中，再释放原资源，而移动语义通过改变资源的所有权节省了拷贝和释放的操作；

写移动构造/赋值函数时需要注意，将源对象的所有指向资源的指针置空，因为源对象可能要析构了，不置空就会析构掉要用的资源；

有些局部变量（左值）本身生命周期短，将其转为右值进行移动构造更高效：

for(int i=0;i<1000;i++){

MyString tmp("hello");

vecStr2.push\_back(std::move(tmp));}

提出完美转发的背景：希望一个函数的参数既能传左值又能传右值，要么重载几次，要么使用常量左值引用，都有缺点：重载次数是2的参数数量次方，设定为常量就不能修改了；

完美转发：std::forward，是万能引用的一个应用场景，通过万能引用和引用折叠推导出参数的原属性并转发，使其保持原来的左右值属性不变：

万能引用通过引用折叠的推导规则【遇左则左】来推导引用对象的类型，因此万能引用必须使用直接推导，包括auto&&和模板推导；

如定义了void print(T & t)和void print(T && t)，调用print(1)，传入右值，但是会被隐式转换为左值处理；而如果定义了模板函数（将参数完美转发给其他函数）

template<typename T>

void testForward(T && v){

print(std::forward<T>(v)); // 等价于print(static\_cast<T&&>(v));对v再做一次万能引用的推导

}

调用testForward(1)，则会调用print(T && t)。

**\*\*Lambda表达式的理解？有什么好处？**

Lambda表达式是一种匿名函数，可以在需要函数的地方直接定义函数体，而无需显式地定义函数名称；它由捕获方式、参数列表、返回类型和函数体组成。

lambda表达式底层实现是一个类，它重载了operator()运算符，使得它可以像函数一样被调用。一个lambda表达式产生一个临时对象，可以作为函数指针传入。

只有[]和{}必选；

好处：

1. 简洁性：在需要函数对象的地方直接定义，代码紧凑易读；

2. 便利性：方便地捕获作用域内的任何动态变量；

3. 匿名性：很多在STL算法中只用一次的函数对象通过lambda实现，可以避免命名冲突和代码冗长；

使用：STL算法中，如数组排序、简单线程创建。

sort(arr, arr + 6, [](const int& a, const int& b){return a > b;});

Thread t1([](int cnt){while(cnt){cout<<cnt<<endl; cnt--;}, 5);

**不稳定的排序算法：快排、堆排、选择排序、希尔排序**

**内联函数**

用于告诉编译器在内联函数调用时将函数代码直接插入到调用函数的地方，而不是通过函数调用的方式执行。这样可以减少函数调用的开销，提高程序执行的效率。

适用于简单短小的函数或者频繁调用的函数；过大的函数体会导致代码膨胀，增加可执行文件的大小

**什么是STL？**

STL是标准模板库，由容器、算法、迭代器、适配器、空间配置器、仿函数组成；

优势：高可重用性、高性能、高移植性、跨平台；

**迭代器是什么？**

迭代器是算法访问容器的统一接口，所有容器都有专属迭代器；

从实现角度看，迭代器是重载了指针相关操作符的类模板；

【vector的迭代器是原生指针，因为维护的是线性空间】

总的来说，迭代器相当于一个泛化的指针，通过派生统一的接口以及通过多态重写这些接口，使得迭代器可以适应不同类型的容器。

**push\_back和emplace\_back()的区别是什么？**

当动态数组vector<int> nums中的空间足够时emplace\_back会比push\_back少一次构造和析构操作：

1.nums.push\_back(1)传入右值时：

1）.调用构造函数构造出值A；

2）.调用拷贝构造函数将A拷贝到nums尾部数据空间

3）.调用A的析构函数

2.nums.emplace\_back(1);传入右值时会：

1）.调用构造函数直接在nums的尾部的内存地址处构造对象。

类似的操作：通过std::move()函数将一个左值强制转化为右值引用，高效移动资源

int num = 2; nums.push\_back(move(num));

注意，在转移语义后num内不再存储任何东西。

**array和普通数组的区别？**

1. 大小固定：std::array是一个固定大小的容器，其大小在编译时确定，一旦创建，大小不能改变。普通的数组也具有固定的大小，但可以通过重新分配内存来改变其大小。

2. 安全性检查：std::array提供了**边界检查**，确保访问容器元素时不会越界。它提供了成员函数at()来进行访问，如果索引超出范围，会抛出std::out\_of\_range异常。普通的数组没有内置的边界检查，如果访问超出边界的元素，可能会导致未定义的行为。

3. 传递性：std::array可以像其他STL容器一样进行传递和返回，可以作为函数参数或函数返回值。普通的数组在传递时通常会退化为指针，丢失了原始数组的大小信息。

4. STL算法支持：std::array可以与STL算法无缝地集成，可以直接使用STL算法对其进行排序、查找等操作。普通的数组需要自己编写循环或使用指针来实现类似的操作。

5. 成员函数和操作符重载：std::array提供了一些有用的成员函数和操作符重载，如size()、empty()、operator[]等，使得操作更加方便和直观。普通的数组没有这些成员函数和操作符重载。

**各容器的底层实现？**

顺序容器：

vector:基于数组实现，采用连续的内存块来存储元素；通过固定/加倍的方法扩展空间。它维护三个指针first【起始位】、last【最后一个元素的末尾】、end【占内存的末尾】；

list:基于双向链表实现，每个元素放在一块内存中，内存空间不是连续的，它维护头指针、尾指针；

deque:存储于几段连续的空间中，可以直接再前后进行扩展，但查询效率没有vector块，它维护map【作主控，元素为指向一片连续空间的指针】、map\_size、start【迭代器】、finish；

关联容器：

map和set：底层采用红黑树构建，键具有唯一性，查找效率为O(logn)，没无序map和无序set快

unordered\_map、unordered\_set底层基于“哈希表”构建，占用空间大，但查找速度快O(1)-O(n)

容器适配器：

基于双端队列deque实现的stack和queue；基于完全二叉树构建的优先队列“priority\_queue”

**vector中大小和容量的区别？**

size表示当前vector中实际存储的元素数量；

capacity表示当前vector在重新分配内存之前可以容纳的元素数量

**sort的底层原理？**

1. 采用正常的快排，效率O(nlogn)；

2. 一旦分段后的数据量小于某个阈值（一般16），此时使用插入排序，因为分段基本有序，所以效率为O(n)；

3. 如果递归层次过深（递归需要将局部变量压栈，过深会造成栈溢出），改用堆排序，效率O(nlogn)；

**堆排序和快排的对比**

堆排序和快排虽然效率上都是O(nlogn)，但平均表现快排比堆排序快2到5倍左右，原因在于快排存取模型的局部性更强（快排每次排序都会使数据尽量移动到最终位置，排序会在局部范围进行；而堆排序是所有未排序数据都可能访问到），更利于缓存发挥作用；

堆排序好就好在最坏情况效率也是O(nlogn)。

**CPP中构造函数的种类**

默认构造函数：无参，类中没定义时编译器会自动生成。

有参构造函数：接受参数用来初始化对象的成员变量。

拷贝构造函数：通过复制已存在的对象来初始化一个新的对象。

移动构造函数：用于将临时对象的资源直接移动到新的对象中，通常使用右值引用实现

析构函数：在对象声明周期结束时执行释放资源、关闭连接等操作。

**用过设计模式吗？大致说说**

设计模式是在软件设计中广泛使用的解决问题的可重复方案；

按设计目的分三类：创建型模式（如何创建对象）、结构型模式（如何组合对象）、行为型模式（如何处理对象之间的动态通信和职责分配）；

常见的设计模式：

1. 单例模式：保证一个类只有一个实例，并提供全局访问点；

2. 工厂模式：将对象的创建和使用分离，客户端只要知道产品接口；

3. 适配器模式：将一个类的接口转换成客户端所期望的另一个接口，使得原本由于接口不兼容而不能一起工作的类可以一起工作；

4. 装饰器模式：动态地给一个对象添加一些额外的职责，同时又不改变其结构；

5. 策略模式：定义一系列算法，将每个算法封装起来并可互换使用，使得算法的变化不会影响到使用算法的客户端；

6. 观察者模式：对象间的一种一对多的依赖关系，当一个对象状态发生改变时，所有依赖它的对象（观察者）都会得到通知并自动更新；

**设计原则**

1. 单一职责：缩小类的粒度，功能明确单一；

2. 开闭原则：对扩展开放，对修改关闭；

3. 里氏替换：子类能够完全替换父类，不会改变父类定义的行为；

4. 接口隔离：不该强迫客户实现它们不需要的接口，即接口方法要尽量细化；

5. 依赖倒置：上层避免依赖下层的实现细节，而应该依赖于抽象（好处是可以灵活替换抽象类的各种具体子类）；

6. 组合优于继承：组合耦合度低。

**什么是单例模式？**

单例模式是一种创建型设计模式，保证一个类只有一个实例，并提供一个全局访问点（静态方法）来访问该实例；

优势：全局控制、节省资源、懒加载（懒汉模式）；

结构：**私有**构造函数【防止外部实例化】、**私有静态**实例变量【唯一实例】、**公有静态**方法【获取实例】；

应用场景：创建管理线程池、数据库连接池、日志。

**什么是策略模式？**

策略模式是行为型设计模式，定义了一系列算法（不同实现完成相同工作）并封装，使其可以相互替换；

优势：算法的变化不影响使用算法的客户；

结构：策略类【定义所有支持的算法的公共接口】、具体策略类【实现策略接口】、上下文【包含策略实例，需要时调用策略对象的方法】；

应用场景：需要动态地在几种算法中选择一种、代码存在大量条件判断且条件区别仅在行为时可以使用策略模式。

**计网相关：**

**OSI七层模型？**

应用层：网络体系结构中的最顶层，提供用户接口和应用程序之间的通信服务。在这一层，用户可以访问各种网络应用程序，如电子邮件、文件传输和远程登录。

表示层：负责数据的格式化、加密和压缩，以确保数据在不同系统之间的交换是有效的和安全的。它还提供了数据格式转换和语法转换的功能。

会话层：管理应用程序之间的通信会话，负责建立、维护和终止会话。它还提供了数据的同步和检查点恢复功能，以确保通信的完整性和持续性。

传输层：传输层为应用程序提供端到端的数据传输服务，负责数据的分段、传输控制、错误恢复和流量控制。它主要使用 TCP（传输控制协议）和 UDP（用户数据报协议）来实现这些功能

网络层：网络层负责数据包的路由和转发，以及网络中的寻址和拥塞控制。它选择最佳的路径来传输数据包，以确保它们能够从源主机到目标主机进行传输。

数据链路层：数据链路层提供点对点的数据传输服务，负责将原始比特流转换为数据帧，并检测和纠正传输中出现的错误。它还控制访问物理媒介的方式，以及数据帧的传输和接收。

物理层：物理层在物理媒介上传输原始比特流，定义了连接主机的硬件设备和传输媒介的规范。它确保比特流能够在网络中准确地传输，例如通过以太网、光纤和无线电波等媒介。

**浏览器键入网址(url)的全过程？**

1. 检查浏览器缓存中是否存在该资源，有则返回，无则下一步网络请求；

2. DNS解析：将域名转换成ip地址；

3. TCP连接：浏览器向服务器发起TCP连接请求，建立连接；

4. 浏览器发送http/https请求；

5. 服务器根据请求内容返回相应的响应数据；

6. 浏览器解析响应：如果状态码为301、302，会重定向到新地址；如果数据类型是字节流，则将请求提交给下载管理器；如果数据类型是HTML，进入下一步渲染；

7. 浏览器渲染：解析HTML文件，创建DOM树，解析CSS进行样式计算，然后将DOM和CSS合并构建渲染树，最后布局和绘制渲染树完成页面展示。

**DNS怎么工作？**

浏览器接收到网址后，首先查找自己的缓存，没有就查找本机域名解析文件hosts，还没有就向本地域名服务器查询（递归查询）；

本地域名服务器先在缓存中查找，以迭代查询的方式依次在根域名服务器->顶级域名服务器->权威域名服务器查找，最后返回ip给浏览器；

**\*\*DNS什么时候使用tcp协议？**

结论：报文大于512字节时可能会导致UDP报文分片，更容易丢失，所以这个时候使用TCP；

1. UDP 协议的额外开销小、有着更好的性能表现，在数据包较小的时候使用；

2.由于 DNSSEC（安全扩展协议，确保DNS查询的可信、完整）和IPv6 的引入，DNS 查询报文迅速膨胀，经常超过 MTU ，造成数据的分片和丢失，这时需要更加可靠的 TCP 协议完成数据的传输；

3. 随着 DNS 查询中包含的数据不断增加，TCP 协议头以及三次握手带来的额外开销比例逐渐降低，不再是报文占比的主要成分；

**TCP、UDP、http报文的格式？**

tcp报文格式（至少20字节）：

源端口号 | 目标端口号

序列号

确认应答号

首部长度 保留 控制位 | 窗口大小

校验和 | 紧急指针

udp报文格式（8字节）：

源端口号 | 目标端口号

UDP长度 | UDP检验和

http报文格式：请求/状态行、请求/响应头【常用：host、connection、content-type、content-encoding、content-length】、空行、请求/响应体

**\*\*TCP和UDP的区别？**

1. 面向连接、不面向连接；

2. **服务对象**；

3. 可靠性：确认应答、重传机制、流量控制、拥塞控制；不可靠

4. 传输方式：TCP是面向字节流的流式传输，没有边界；UDP面向数据包的传输，会一个包一个包发送，有边界，可能乱序丢包；

5. **首部开销**：TCP至少20字节，UDP8字节

6. **分片**：TCP数据如果大于MSS（最大分段大小），则会在传输层分片；UDP数据如果大于MTU（IP层的最大传输单元），则会在IP层分片；

7. 应用场景：TCP用于网页浏览、文件传输（可靠性）；UDP常用于DNS，音视频通信，广播通信（低延迟）；

**\*\*TCP怎么保证可靠传输？**

1. 连接管理：三次握手和四次挥手；

2. 序列号：保证有序性，防止数据丢失，避免数据重复；

3. 确认应答：接收方接收数据后会回传ACK报文，告知接收情况，若发送端在指定时间内未收到确认应答则会触发超时重传；

4. **校验和**：覆盖首部和数据，接收方重新计算校验和并对比，检测是否存在错误或损坏；

5. 重传机制：超时重传、快速重传、SACK（选择性确认，嵌套在快速重传中，告诉发送端都收到了哪些消息）；

6. 流量控制（糊涂窗口、窗口关闭现象）：根据接收端的处理能力决定发送数据的速度；

7. 拥塞控制：根据网络拥塞状况决定发送数据的速度。

**服务端如果没有打开，客户端发起TCP连接会发生什么事？**

如果服务端完全没有打开，客户端发出SYN报文后收不到回应，超时重传次数到达最大值后断开连接；

如果服务端绑定了ip和端口，但没有监听，客户端发出SYN报文后，服务端回RST报文。

**为什么TCP在传输层分片，UDP在IP层分片？**

IP层分片一旦丢失一片，整个IP报文就失效了；

UDP不可靠，失效也就不管了；

TCP需要保证可靠传输，在IP分层效率低，一个分片丢失就需要全部重传，在传输层分片可以只传丢失的那个分片。

**描述TCP三次握手，为什么不是两次？各次握手丢失发生什么情况？**

1. 【第一次】客户端发送SYN报文，其中随机初始化序列号client\_isn=x，然后处于SYN-SENT状态；

2. 【第二次】服务端收到SYN报文后，发送SYN+ACK报文，其中随机初始化序列号server\_isn=y，确认应答号x+1，然后处于SYN-RCVD状态；

3. 【第三次】客户端收到SYN+ACK报文后，发送ACK报文，其中确认应答号y+1，处于ESTABLISHED状态；（可以携带数据）

4. 服务器收到ACK后，进入ESTABLISHED状态。

不是两次的原因：

1. 避免历史连接（主因）：如果没有第三次确认，服务器遇到旧的SYN会直接建立连接，没有中间态（SYN-RCVD）等待客户端的第三次握手，就无法确认SYN是否是旧的；

2. 同步双方初始序列号；

3. 避免资源浪费：两次握手时服务端每次收到SYN只能主动先建立连接；

不是四次的原因：

中间服务端ACK+SYN两步合成了一步；

握手丢失发生情况：

第一次丢失：客户端超时重传；

第二次丢失：客户端和服务端超时重传；

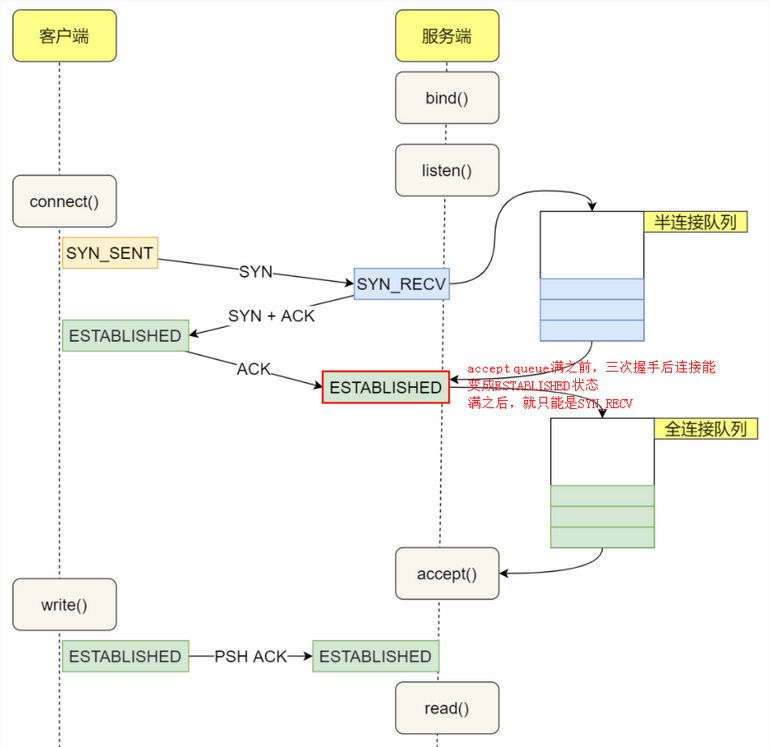
第三次丢失：服务端超时重传。（客户端由于发送的是ACK报文，ACK不会超时重传）

**TCP三次握手和网络编程接口之间的联系？**

1. 服务器调用bind、listen准备好监听连接；bind()[函数](http://baike.baidu.com/view/15061.htm" \t "https://blog.csdn.net/u011430225/article/details/_blank)通过给一个[未命名](http://baike.baidu.com/view/1418045.htm" \t "https://blog.csdn.net/u011430225/article/details/_blank)[套接口](http://baike.baidu.com/view/567586.htm" \t "https://blog.csdn.net/u011430225/article/details/_blank)分配一个本地名字来为[套接口](http://baike.baidu.com/view/567586.htm" \t "https://blog.csdn.net/u011430225/article/details/_blank)建立（[主机地址](http://baike.baidu.com/view/547482.htm" \t "https://blog.csdn.net/u011430225/article/details/_blank)/[端口号](http://baike.baidu.com/view/642103.htm" \t "https://blog.csdn.net/u011430225/article/details/_blank)组成的）本地捆绑

2. 客户端调用connect发送syn报文，开始进行TCP三次握手；

3. 服务端维护一个半连接队列和全连接队列，没建立完的连接放在半连接队列中（针对半连接队列的网络攻击是syn floods攻击，攻击方不停发送syn占满队列，使得其他请求无法进来），建立完成的放在全连接队列中（listen()中的backlog参数表示全连接队列长度）；

4. 服务器调用accept从全连接队列中取出连接，返回给进程；

**描述TCP四次挥手，为什么是四次？挥手丢失发生什么？其中time\_wait的作用？为什么最后等2MSL？**

1. 【第一次】客户端发送FIN报文，进入FIN\_WAIT\_1状态；

2. 【第二次】服务端收到后，发送ACK报文，进入CLOSE\_WAIT状态；

3. 客户端收到后，进入FIN\_WAIT\_2状态；

4. 【第三次】服务端发送FIN报文，进入LAST\_ACK状态；

5. 【第四次】客户端收到后，发送ACK报文，进入TIME\_WAIT状态；

6. 服务端收到后，进入CLOSE状态；

7. 客户端经过2MSL后，进入CLOSE状态；

**注意：如果服务端没有数据要发送&&开启TCP延迟确认机制（等等看有没有DATA，有的话DATA+ACK一起发送，如果等待时间内收到第二份数据，立刻发送ACK），第二、三次挥手会合并；**

四次的原因：

TCP是全双工通信，两次握手可以释放一端到另一端的连接，完全释放需要四次；

挥手丢失发生情况：

第一次丢失：客户端超时重传；

第二次丢失：客户端超时重传；（ACK报文不会重传）

第三次丢失：服务端超时重传；

第四次丢失：服务端超时重传；

TIME\_WAIT作用：

1. 防止历史数据被后面相同四元组的连接 错误接收（给时间让历史数据被丢弃）；

2. 保证被动关闭连接的一方被正确关闭（确保ACK让被动方收到）；

2MSL的原因：

1. MSL的时间让历史数据被丢弃；

2. 当ACK没有被正确接收时，需要等待另一端超时重传FIN报文并重新回应，来回最长需要2MSL时间。

**Time\_wait过多有什么危害？**

客户端：如果time\_wait占满了所有端口资源，就无法再跟目的ip和目的端口都一样的服务端建立连接；

服务端：不会导致端口资源受限，因为只监听一个端口（理论上可以与很多不同的客户端建立连接），但tcp连接过多会占用系统资源。

**怎么处理time\_wait过多的情况？**

一般来说服务端主动断开连接才会出现time\_wait过多的情况，而主动断开连接的情况有：

1. http没有使用长连接；

解决方案为双端connect字段设置为keep-alive；

2. http长连接超时：大多数web服务软件都会提供参数指定连接超时的时间，比如nginx提供keepalive\_timeout，time\_wait过多可能是因为大量客户端在建立连接后没有把数据成功发送到服务端；

解决方案：检查网络问题；

3. http长连接请求数量达到上限：web服务软件有参数定义http长连接能处理的最大请求数，比如nginx的keepalive\_requests，请求数超过该值则会断开长连接；当QPS（每秒请求数）过高，就会出现服务端频繁主动断开连接的现象；

解决方案：参数调大。

**TCP、UDP数据包大小的限制？**

数据链路层，物理特性决定帧最大1500字节，即MTU最大为1500；

路由器中MTU可能设置成不同的值，Internet（非局域网）标准MTU为576；

UDP包的大小=MTU – IP头（20）-UDP头（8）

UDP头中的报文长度16位（2字节），所以UDP包含报头在内的数据报的最大长度为2^16-1=65535字节

**\*\*什么是滑动窗口？**

滑动窗口协议是传输层进行流量控制的方式；

1. 允许发送方连续发送多个分组，而不必每发送一个分组就等待确认，增加了数据传输的速率；

2. 接收方可以通过窗口大小控制发送方的发送速率，避免发送方发送速率过快导致接收方不能接收的问题；

发送窗口：限制发送方可发送的数据大小，由接收方返回的窗口大小字段控制；接收到ACK响应时移动窗口；

接收窗口：标记可以接收的数据大小；接收到数据且左侧连续才移动窗口。

**\*\*细说拥塞控制？**

**拥塞控制避免发送方的数据填满整个网络，只要发生重传则认为网络出现拥塞**；

发送窗口=min（拥塞窗口，接收窗口）；

拥塞控制四个算法：

1. 慢启动：拥塞窗口从初始值开始（不一定是1），发送方每收到一个ACK，cwnd++（指数增长），直到涨到慢启动门限ssthreash；

2. 拥塞避免：每当收到一个ACK，cwnd增加1/cwnd（线性增长）；

3. 拥塞发生：

ssthresh=cwnd/2，cwnd=初始值【出现超时重传】；

ssthresh=cwnd/2，cwnd=ssthresh【出现快速重传：发送方收到了三次重复的ACK，就会马上重传】；

4. 快速恢复：和快速重传同时使用，线性增长。

**http和https的区别**

1）、http协议传输的数据都是未加密的，也就是明文的，因此使用HTTP协议传输隐私信息非常不安全， 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，要比http协议安全。

2）、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。

3）、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

**https对比http为什么更安全？**

1. 身份验证： https使用证书对服务器进行身份验证，防止中间人攻击；

2. 数据加密：https使用SSL/TLS协议对数据加密，防止传输过程中被窃听或篡改；

3. 数据完整性验证：https使用消息摘要算法验证数据的完整性；

**长连接和短连接，它们各有什么优缺点？**

短连接

客户端和服务器每进行一次http连接就会建立一次tcp连接，任务结束就中断

优点：管理简单，不要额外的控制手段

缺点：客户端请求频繁时，将在tcp的连接和关闭操作上浪费时间和带宽。

长连接：

当一个网页打开完成后，客户端和服务器之间用于传输http数据的TCP连接不会关闭，客户端再次访问这个服务器时，会继续使用这一条已经建立的连接

优点：可以省去较多的TCP建立和关闭的操作，减少浪费，节约时间，适用于频繁请求资源的用户

缺点：存在存活功能的探测周期长的问题，遇到恶意连接时保活功能不够；随着客户端连接的增多，服务器可能扛不住，需要采取额外的策略来管理连接。

**SSL/TLS握手过程？**

是一个使用非对称加密协商对称加密密钥的过程，为什么是这个过程呢？因为对称加密密钥运算速度比非对称加密快，但是对称密钥必须保密，无法做到安全交换；而非对称加密使用两个密钥：公钥任意分发而私钥保密，解决了密钥交换的问题【对称加密和非对称加密的区别】；

具体过程：

1.客户端->Client Hello：支持的TLS协议版本、支持的密码套件列表、客户端随机数；

2.服务端->Server Hello：确认TLS版本、确认的密码套件、服务器数字证书、服务端随机数；

3.客户端（CA公钥验证证书，取出服务器公钥）：pre-master随机数（服务器公钥加密）、加密算法改变通知、握手结束（将之前所有内容的数据做摘要，供服务端校验）；

【服务端和客户端有三个随机数后，用双方协商的加密算法生成会话密钥】

4.服务端：加密算法改变通知、握手结束（将之前所有内容的数据做摘要，供客户端校验）。

**对称加密和非对称加密的区别？**

对称加密只使用一个密钥进行加解密，优点是运算**速度快**，缺点是密钥**必须保密**，无法做到安全的密钥交换。

非对称加密使用两个密钥：公钥和私钥。公钥可以任意分发而私钥保密，解决了密钥交换问题但**速度慢**。公钥和私钥都可以用来加密和解密，流程不同，目的不同：

**公加私解：**保证内容传输的安全，因为被公钥加密的内容，其他人是无法解密的，只有持有私钥的人，才能解密出实际的内容；

**私加公解：**保证消息不会被冒充，因为私钥是不可泄露的，如果公钥能正常解密出私钥加密的内容，就能证明这个消息是来源于持有私钥身份的人发送的。（客户端校验服务端证书时用CA公钥解密）

一般不用非对称加密来加密实际的传输内容，因为计算耗费性能。

**签发证书和验证证书的过程？**

签发：

1. CA把证书持有者的公钥、用途、颁发者、有效时间等信息打包，使用hash算法得出hash值；

2. CA使用自己的私钥对hash值加密，生成签名，并添加在证书上，最终形成数字证书；

验证：

1. 客户端使用同样的hash算法得到hash值H1；

2. 客户端使用浏览器或操作系统中集成的CA公钥解密签名，得到hash值H2；

3. 若H1=H2，则证书可信赖。

证书链：

一般服务器（以百度举例）向CA申请的证书不是由根证书签发的，而是由中间证书签发的（客户端没有中间证书的公钥），因此客户端验证过程可以分为三步：

1. 收到百度的证书，发现该证书的签发者不是根证书，则向CA请求相应的中间证书；

2. 中间证书由根证书签发，则利用已有的根证书公钥验证中间证书；

3. 通过中间证书公钥验证百度证书【即上述的验证过程】。

**了解哪些对称/非对称加密算法？**

对称：AES（最流行、广泛使用）

非对称：RSA（用于数据加密和数字签名）、ECC（适用于移动设备等资源受限的环境）

**怎么防止下载的文件被劫持和篡改？**

使用https下载：通过TLS加密数据，防止中间人攻击和篡改；

验证文件完整性：下载文件后使用哈希算法（MD5等）计算文件哈希值，与官方提供值对比；

**\*\*get和post的区别**

1. 作用：get从服务器获取资源；post向服务器提交数据；

2. 参数传递方式：get请求参数一般写在URL，只接受ASCII字符；post放在请求体，数据类型无要求；

3. 安全性：get请求参数暴露在URL中，更不安全（但是http明文传输，其实两者都不安全）；

4. **传送数据量**：get较小，不大于2kb；post大，不受限制；

5. **编码方式**：get只能URL编码；post支持多种编码方式；

6. 缓存机制：get会被主动cache，post不会除非手动设置；**get参数被保留，post不会；get在回退时无害，post会再次提交请求；**

7. 幂等（多次执行相同操作，结果相同）：get幂等；post会修改服务器资源，不幂等；

**\*\*http协议的发展过程**

http1.0->http1.1：

1. 支持长连接；

2. 引入更多缓存控制策略；

3. 请求管道化：请求不必等到响应才能发下一个，可以一起发过去，减少整体响应时间；

4. 带宽优化：请求头引入range，允许只请求资源的某个部分，返回码为206；

http1.1的问题：

1. 首部未经压缩就发送，而且发送相同首部造成的浪费比较多；

2. 同一个连接中，服务端按请求的顺序响应，如果当前请求响应得慢，会导致后续请求一直得不到响应，造成队头阻塞；

3. 请求只能从客户端开始，服务器被动响应。

http1.1->http2.0：

1. 二进制分帧：在应用层和传输层之间增加二进制分帧，头信息和数据体全是二进制，改善传输性能（’2’’0’’0’在文本中占用3字节，在二进制中对其编码只要1字节）；

2. 首部压缩：使用HPACK算法压缩头（客户端服务器维护头信息表，所有字段存入表并生成索引号，以后遇到表中有的头就只发送索引号）；

3. 多路复用：多个stream复用在一个TCP连接上；使用stream id区分不同的http请求，不同stream的帧可以乱序发送，接收端通过stream id有序组装成http消息，因此可以并发不同的stream，也就是可以并发响应同一连接上的不同请求；

4. 服务器推送：可以主动向客户端发送消息，客户端建立的stream的id是奇数，服务端的id是偶数。

http2的问题：

http2通过stream的并发能力解决了http1的在应用层的队头阻塞问题，但TCP层的队头阻塞问题仍然没有解决：

TCP是字节流协议，必须保证收到的字节数据完整且连续，这样内核才会将缓冲区的数据返回给http应用；

当前一个字节没有到达时，后收到字节就只能放在内核缓冲区里，等前一个字节到达时，http2应用层才能从内核中拿到数据；

**http队头阻塞问题？**

1. http1.1的实现基于请求响应模型：同一个连接中，服务端按请求的顺序响应，如果当前请求响应得慢，会导致后续请求一直得不到响应，造成队头阻塞；

2. http2通过stream的并发能力解决了http1的在应用层的队头阻塞问题，但TCP层的队头阻塞问题仍然没有解决：

TCP是字节流协议，必须保证收到的字节数据完整且连续，这样内核才会将缓冲区的数据返回给http应用；

当前一个字节没有到达时，后收到字节就只能放在内核缓冲区里，等前一个字节到达时，http2应用层才能从内核中拿到数据；

也就是tcp层一旦发生丢包，就会阻塞所有http请求；

3. http3把下层的tcp改成了udp，在http层使用quic协议实现类似tcp的可靠性传输；udp不管顺序，也不管丢包，所以不会出现传输层队头阻塞的问题。

**有了http协议，为什么还要有websocket？**

举例来说，我们想了解今天的天气，只能是客户端向服务器发出请求，服务器返回查询结果，http协议用成了半双工，它做不到服务器主动向客户端推送信息。如果服务器有连续的状态变化，客户端要获知就只能使用"轮询"方法：每隔一段时候，就发出一个询问，了解服务器有没有新的信息。最典型的场景就是聊天室。轮询的效率低，非常浪费资源

websocket是一种在客户端和服务器之间建立长连接的协议，与http协议相比，它能够实现双向通信，服务器可以主动向客户端推送信息，简化了通信流程，降低了服务器压力。

websocket主要解决服务器主动推送大量数据给客户端的场景：微信小程序、网页游戏等；

http解决服务器主动推送的方式：

定时轮询：客户端以1到2秒的时间间隔不断发出请求，看看码有没有被扫；缺点为消耗带宽，用户感到卡顿（网页扫码登录）；

长轮询：超时设置得很大如30秒，在30秒内服务器收到了扫码请求，就马上返回给客户端（百度网盘扫码登录）；

websocket是什么：在传输层上，TCP是全双工的，但是被http协议用成了半双工，所以需要新的应用层协议websocket；

怎么连接（两次握手）：

1. http请求中带上特殊的header头（connection：upgrade，upgrade：websocket，随机生成的base64码）发给服务器【第一次】；

2. 服务器用公开算法将base64码转成一段字符串放在响应头中，发给客户端【第二次】；

3. 浏览器用同样的公开算法将base64码转成字符串与服务器的比较，如果一致则验证通过，握手成功。（升级后和http没关系）

**服务器响应后，怎么禁止客户端进行缓存？**

Cache-Control字段设置为no-store；

no-cache表示强制协商缓存；

max-age=N表示缓存多长时间；

**\*\*cookie和session的区别？**

cookie和session都用于管理用户的状态和身份。

cookie：存放在客户端，请求时会将其放在请求头，服务端根据cookie得到客户端特有信息，从而动态生成对应内容；

session：把客户端信息记在服务器中，**维护用户登录状态、存储用户临时数据和上下文信息**；

区别：

1. 存储位置；

2. 安全性：cookie存在客户端，容易被篡改；session存在服务端，更难被访问和修改；

3. 数据容量：cookie小（4kb），session大；

4. 有效期：cookie可设置为长时间保持（比如默认登录功能），session一般有效期短，客户端关闭或session超时都会失效；

5. 传输方式：cookie每次请求都会被自动发送至服务器，**session id通常通过cookie或url参数传递。**

**cookie里存哪些字段？**

name、value、

domain（可以访问cookie的域名）、

path（可以访问cookie的页面路径）、

expires/max-age（cookie超时时间）、

size（cookie大小）、

secure（是否只能通过https传递cookie）。

**Ping用什么协议，在哪层？**

ICMP（Internet Control Message Protocol）协议，网络层；

ICMP是面向无连接的协议，用于在主机与路由器之间传递控制信息。

**操作系统相关：**

**操作系统由哪些部分组成？**

内核、文件系统、设备驱动程序、Shell（用户与操作系统交互的接口）

**用户态和内核态的区别？**

用户态是应用程序运行的环境，受限制，无法直接访问底层硬件和资源；

内核态是操作系统内核执行的环境，具有更高的特权级别，可以直接访问硬件和资源。

**\*\*用户态陷入内核态的方式？**

1. 系统调用：

系统调用是操作系统提供给应用程序用于操作底层硬件、资源的接口，保证系统的安全稳定；

一般应用程序在用户态运行，不能执行I/O操作、内存管理、进程管理、网络通信等，这些功能需要在内核态下才能执行，于是应用程序通过系统调用安全可控地执行这些功能；

2. 中断：

CPU在执行指令时，收到中断信号后会转而执行中断处理程序，再返回到原指令流中继续执行；

3. 异常：

CPU执行某些指令时发生错误或故障（比如访问非法内存、除数为0）时，就会陷入异常。

**用户态陷入内核态的开销？**

1. 上下文的切换：保存当前用户态的上下文信息（程序计数器、寄存器、堆栈指针），以便返回用户态的时候能正确恢复；

2. 特权级别的改变：需要执行额外的指令和权限检查；

3. 内存或缓存失效：内核态的数据代码一般和用户态的分开存放在不同的内存区域，当从用户态切换到内核态时，可能需要加载新的指令和数据到内存或缓存中，导致原先在内存或缓存中的用户态数据失效。

**处理并发常用的锁？**

互斥锁、自旋锁、读写锁（都属于悲观锁）

**互斥锁：**加锁失败时，会用「线程切换」来应对，当加锁失败的线程再次加锁成功后的这一过程，会有两次线程上下文切换的成本，性能损耗比较大。适用于线程执行时间长；系统中线程数量过多的情况（自旋会增加系统负担）

**自旋锁：**加锁失败时，一直忙等待，直到获取到锁，并不会主动产生线程切换，如果**被锁住的代码执行时间很短**，那这个忙等待的时间相对应也很短。适用于线程执行时间短的情况。

**读写锁：**如果能区分读操作和写操作的场景，那读写锁就更合适了，它允许多个读线程可以同时持有读锁，提高了读的并发性。根据偏袒读方还是写方，可以分为**读优先锁**和**写优先锁**，读优先锁并发性很强，但是写线程会被饿死，而写优先锁会优先服务写线程，读线程也可能会被饿死，那为了避免饥饿的问题，于是就有了**公平读写锁**，它是用队列把请求锁的线程排队，并保证先入先出的原则来对线程加锁，这样便保证了某种线程不会被饿死，通用性也更好点。

互斥锁和自旋锁都是最基本的锁，读写锁可以根据场景来选择这两种锁其中的一个进行实现。

另外，互斥锁、自旋锁、读写锁都属于悲观锁，悲观锁认为并发访问共享资源时，冲突概率可能非常高，所以在访问共享资源前，都需要先加锁。

相反的，如果并发访问共享资源时，冲突概率非常低的话，就可以使用乐观锁，它的工作方式是，在访问共享资源时，不用先加锁，修改完共享资源后，再验证这段时间内有没有发生冲突，如果没有其他线程在修改资源，那么操作完成，如果发现有其他线程已经修改过这个资源，就放弃本次操作。

**\*\*中断的作用？**

1. 外设异步通知CPU（硬件中断，分为可屏蔽与不可屏蔽）；

2. CPU之间发送消息（处理器间中断）；

3. 处理CPU异常（软件中断）：CPU发现异常会向自己发送中断信号；

CPU异常按是否需要修复以及能否修复分3类：

1. 陷阱（trap）【有意为之的异常，用来主动陷入内核执行系统调用】，不需要修复，中断处理完成后，继续执行下一条指令；

2. 故障（fault）【如缺页】，需要修复，可能修复，中断处理完成后，执行之前的指令；

3. 中止（abort），需要修复，无法修复，中断处理完成后，进程崩溃。

**中断和异常的区别？**

1. 中断是异步的（与当前执行的指令无关），异常是同步的（与当前执行的指令有关）；

2. 中断可以被屏蔽，异常不行（确保程序的正确性和系统的稳定性）；

**硬中断和软中断？**

为了避免由于中断处理程序执行时间过长，丢失了其他其他中断请求，Linux 将中断处理程序分为两个阶段：

1. 硬中断，由硬件触发中断，用来快速处理中断，一般会关闭中断请求，负责处理和硬件紧密相关的事情；

2. 软中断，由内核触发中断，用来异步处理一阶段未完成的工作，一般以内核线程的方式运行；

**网卡接收网络包的过程？**

1. 网卡收到网络包后，硬中断告知CPU，通过DMA方式将接收到的数据写入内存；（这个过程是硬中断，此时禁止了网卡中断，避免频繁硬中断降低内核的工作效率）

2. 通过硬件中断通知内核有新数据；

3. 内核调用对应的中断处理程序来处理该事件，包括解析头部等；（软中断）

**进程和线程的区别？（基于是否共享内存产生的差异）**

进程是系统进行资源分配的基本单位；

线程是系统运算调度的基本单位，是进程内的执行单元；

资源开销：每个进程有独立的内存空间，上下文切换开销大；线程共享内存空间，开销小；

通信同步：进程间相互隔离，需要管道、消息队列、共享内存机制；线程直接访问共享内存；

稳定性：一个进程崩溃不会影响其他进程，线程会。

**协程是什么？**

用户态的轻量级线程，同一进程内，同一时间只能执行一个协程（其他协程休眠），切换开销小，无锁同步；

场景：长时间I/O操作阻塞，可以切换至其他协程运行（如果是线程切换开销大）。

**进程和线程切换开销对比**

1. 上下文切换的开销：

进程切换需要保存和恢复更多的上下文信息。上下文信息包括进程的程序计数器、寄存器状态、堆栈、内存管理单元信息等，这些信息的保存和恢复需要消耗较多的时间。

线程切换只需要保存和恢复程序计数器、寄存器、线程栈，因此线程切换的开销较小。

1. 地址空间的切换：

进程有独立的地址空间，进程切换时需要切换地址空间的映射关系，这涉及到页表的切换和 TLB（Translation Lookaside Buffer）的刷新等操作，开销较大。

线程共享同一进程的地址空间，线程切换不涉及地址空间的切换，因此开销较小。

1. 资源的切换和管理：

进程有独立的资源，包括文件描述符、打开的文件、信号处理器等，进程切换时需要切换和管理这些资源，开销较大。

线程共享同一进程的资源，线程切换不涉及对资源的切换和管理，因此开销较小。

**线程有哪些资源是独有的？**

线程的栈区、程序计数器、使用的寄存器、栈指针；

但线程的栈区不严格隔离，线程A如果拿到线程B的局部变量地址，可以改变；

统称线程上下文。

**Linux如何查看线程内存、cpu占用情况**

首先通过top命令找到进程对应的进程pid

通过top -H -p 进程pid查看线程 cat /proc/进程pid/status查看进程

**\*\*fork()的运行原理？怎么区分父子进程？**

fork是系统调用，复制当前进程，创建一个新的子进程，使得父子进程拥有相同的程序代码、数据和**上下文**，但各自在独立的地址空间中运行；

在父进程和子进程中，fork() 返回值不同：在父进程中，fork() 返回子进程的 PID，而在子进程中，fork() 返回值是 0；

linux通过写时拷贝实现fork：先让父子进程共享同一个地址空间，在需要写入的时候才复制地址空间，从而使父子进程拥有各自的地址空间；

注意：子进程是从fork的返回开始执行，而不是将代码从头开始执行！（因为复制了上下文）

**僵尸进程和孤儿进程是什么？**

每个子进程在退出后，并非马上消失，而是留下进程号、退出状态、运行时间等信息供父进程获取，这时的子进程是僵尸进程；

僵尸进程需要等待父进程通过wait来释放；

孤儿进程是没有父进程的子进程，当父进程在子进程结束前退出时，所有子进程成为孤儿进程，init进程变成它们的父进程，由init进程调用wait()来释放孤儿进程；

僵尸进程若不处理会占用资源，孤儿进程由init进程自动处理，没有危害。

**写时拷贝技术（COW）？**

介绍：COW将拷贝操作延迟到第一次写入时进行，也叫隐式共享；

原因：当通过fork()创建子进程时，需要将父进程内存空间中的内容全部复制到子进程中，操作耗时且占用物理内存；如果子进程拷贝内存后立刻使用exec()加载新程序，那么之前的拷贝操作是多余的；因此引入COW，只有在第一次写入时才拷贝相应内存页，节省了不必要的拷贝操作；

补充exec()：使另一个可执行程序替换当前的进程，除了pid不变，其他数据段、代码段、堆栈等都被替换为新程序的，从头开始执行；

实现方式：

1. fork()之后，内核将父进程所有内存页标记只读，并且内存页的引用数增加，子进程只拷贝父进程的页表（即fork的开销只有拷贝页表和分配pid）；

2. 当某个进程尝试写入某个内存页时，就会触发缺页异常，陷入内核态；

3. 内核为尝试写入的进程拷贝相应的内存页，并恢复原内存页的可写权限，就可以正常执行写操作；

4. 如果内存页只有一个引用，则可以直接修改。

**进程调度方式以及优缺点？**

1. 先来先服务：进程按照先来后到的顺序依次执行；

优点：公平；

缺点：如果长作业先运行，短作业等待时间就会很长；

2. 最短作业优先：优先选择运行时间最短的进程执行；

优点：短作业响应速度快；

缺点：产生长作业饥饿问题；

3. 高响应比优先：计算每个进程的响应比，计算方式为（等待时间+要求服务时间）/要求服务时间，长作业的响应比会随着等待时间的增加而提高，从而获得执行机会；

优点：兼顾了长作业和短作业，解决了长作业饥饿问题；

4. 时间片轮转：将相等长度的时间片依次分配给每个进程执行，不考虑任何优先级；

优点：简单公平；

缺点：时间片太短造成频繁的进程上下文切换，太长使得短作业响应时间变长；

5. 多级反馈队列：

【多级】表示有多个队列，每个队列优先级从高到低，优先级越高时间片越短；

进程如果在当前队列的时间片内没有执行完毕，则加入次优先级队列；

【反馈】表示如果有新的进程加入优先级高的队列，则优先执行这个进程；

优点：兼顾长短作业，同时有较好的响应时间。

**进程的通信方式？**

1. 管道：单向通信，适用于有亲缘关系的进程之间通信；半双工；

2. 命名管道：通过在创建管道文件，允许无亲缘关系的进程之间通信；半双工；

3. 消息队列：进程间异步通信，克服信号传递信息少、管道只能承载无格式字节流等缺点；

4. 信号：通知进程某个事件已经发生，迫使进程执行信号处理程序；

5. 共享内存：多个进程共享一块内存区域，读写不需要陷入内核态，实现高效的数据交换；

6. 信号量：计数器，控制多个进程对共享资源的访问，主要作为进程间、同一进程不同线程之间的同步手段；

7. 套接字：不同主机之间的进程通信，可实现网络通信。

**匿名管道和命名管道的区别？**

1. 本质：匿名管道是内存中的一块缓冲区，命名管道是存储在硬盘的fifo文件；

2. 使用对象：匿名管道只能用在父子进程（能拿到管道地址就能读写），命名管道任意进程；

3. 创建：匿名管道pipe()，命名管道mkfifo()；

4. 使用：匿名管道可直接读写，命名管道需要先open()相应文件；

**线程的通信方式？**

1. 互斥锁；

2. 条件变量；

3. 信号量；

4. 读写锁；

**条件变量的虚假唤醒？**

当一个正在等待条件变量的线程由于条件变量被触发而唤醒时，却发现它等待的条件（共享数据）没有满足(也就是没有共享数据)；

**什么是死锁？死锁条件？解决办法？**

死锁是两个或多个线程由于相互等待而永远被阻塞的情况；

同时满足：互斥条件、请求保持条件、不可剥夺条件、循环等待条件（多个进程间形成循环等待资源的链）；

破坏请求保持条件：一次性申请所有资源；

破坏不可剥夺条件：当申请不到资源时，主动释放自己占有的资源；

破坏循环等待条件：按序申请资源；

**\*\*什么是虚拟内存？为什么需要？**

虚拟内存是内存管理技术，在每个进程创建加载时，会分配一个连续虚拟地址空间，通过映射与物理地址空间对应；

作用：

1. 内存扩展：虚拟内存可以使程序的运行内存超过物理内存大小，程序运行符合局部性原理，当物理内存不足时，可以将不常用的数据暂时移到硬盘swap区域上，释放内存供其他进程使用；

2. 内存隔离：一个进程无法直接访问另一个进程的内存空间；

3. 内存访问安全：页表里的页表项中除了物理地址之外，还有一些标记位，比如控制页的读写权限，标记该页是否存在等。

**内部碎片和外部碎片的描述？**

内部碎片：已分配给进程的内存块中存在未被有效利用的空间；（分页）

外部碎片：已分配内存块之间的零散的无法被利用的未分配空间；（分段）

**软链接和硬链接？**

概念：

软链接：（创建新inode）创建的指向链接目标路径的特殊文件；

硬链接：（不创建新inode）在索引节点inode的links字段+1，删除硬链接只会在links字段-1，直到links=0才删除源文件；不能跨文件系统；

场景：

软链接：快捷方式、切换不同版本的目标程序；

硬链接：文件备份、文件共享、不同角度分类文件；

**五种IO模型？**

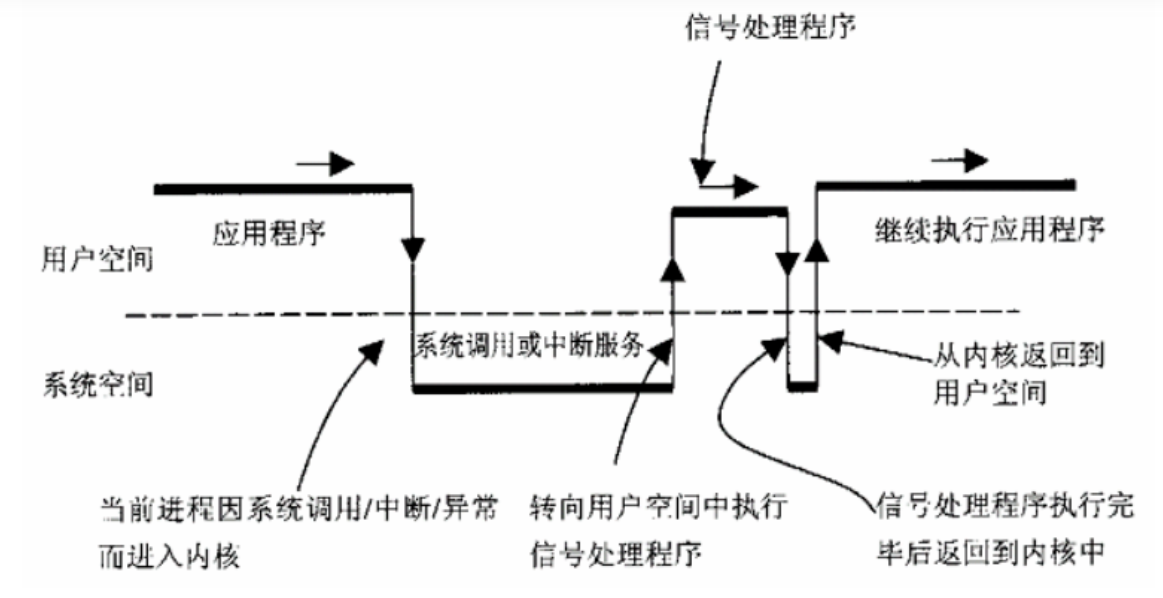
IO时内核一般有两个阶段：内核准备数据、将数据拷贝到用户缓存（读）；

1. 阻塞IO：用户线程发起IO后等待两个阶段的完成；

2. 非阻塞IO：当内核没准备好数据时直接返回，当下次轮询发现内核准备就绪后等待第二阶段的完成；

3. IO复用：使用select/poll/epoll函数同时监听多个fd是否准备就绪，用户线程通过函数获得就绪fd并处理；（不必为每个fd创建监控线程）

4. 信号驱动IO：当用户发起IO时，给对应socket注册信号处理函数，然后继续执行，当内核数据准备就绪时，发送信号给用户，通知用户线程调用信号处理函数读取数据；（优化无用的轮询）

5. 异步IO：用户线程发起IO，内核完成两个阶段后发出信号通知用户线程。

**select/poll/epoll？**

select和poll都是在用户态中通过线性结构来存储进程关注的socket集合，在调用时将集合拷贝到内核中，内核遍历查看socket是否发生事件，发生了就标记，再把集合拷贝回用户态，由用户遍历找到发生事件的socket；

不同的是select支持的fd个数受内核限制，最大值为1024，poll使用链表组织fd，只受系统文件描述符限制；

epoll：在内核中通过红黑树存储socket集合，使用事件驱动机制维护一个记录就绪事件的链表：每当有socket发生事件时，将其添加到就绪链表中；当调用epoll\_wait时可以直接获得发生事件的socket，不需要遍历；

随着连接数增加，select/poll遍历和拷贝的代价越来越大，而epoll基本不受影响，而且epoll可以使用效率更高的ET触发模式（接下面的八股）。

**LT/ET触发模式？**

LT（Level Trigger，水平触发）

当epoll\_wait检测到有事件发生并将此事件通知给应用程序后，应用程序可以不立刻处理该事件，当应用程序下一次调用epoll\_wait时，epoll函数在仍会告知此事件，直到该事件被处理为止。

ET（Edge Trigger，边缘触发）

当epoll\_wait检测到有事件发生并将此事件通知给应用程序后，应用程序必须立刻处理该事件 ，因为后续的epoll\_wait调用将不再重复通知此事件。

ET比LT好在：epoll\_wait系统调用次数减少，效率高。

select/poll只能实现LT。

**epoll用红黑树管理，为什么不用哈希表？**

1. 红黑树容易扩容缩容，而哈希表扩容缩容需要重新分配桶数组的动态内存，计算所有元素的新哈希值，将元素按照新的哈希值重新放回桶中；

2. 哈希表的查找插入删除操作的平均时间复杂度是O(1)，但最坏情况下可能会退化为链表，导致复杂度为O(n)；

**\*\*Reactor和Proactor？**

两者都是基于事件分发的网络编程模式，

Reactor是非阻塞同步网络模式，感知的是就绪可读写的事件；在每次感知到有事件发生，比如读就绪事件，需要应用进程主动将socket缓存中的数据读到进程内存后才能进行下一步的业务处理，这个过程是同步的；

Proactor是异步网络模式，感知的是已完成的读写事件；在发起异步读写请求时，读写过程由内核执行，读写完成后内核直接通知应用进程处理数据；

**数据库相关：**

**CAP定理？**

一致性：分布式系统所有节点上，对于同一数据的读操作都应该返回相同的结果；

可用性：面对用户请求时能够及时响应并返回结果；

分区容错性：网络分区（节点之间无法相互通信）的情况下仍然能够正常运行；

**数据库范式？**

第一范式：强调列的原子性，列不能再分成其他几列；

第二范式：非主属性完全依赖主属性，而不是依赖于主属性的一部分；

第三范式：任何非主属性不依赖于其他非主属性，在2NF基础上消除传递依赖。

**MySQL语句执行流程？**

1. 客户端连接数据库；

2. 解析第一个字段，如果是查询语句，就去查询缓存（8.0后移除，因为对于更新频繁的表缓存命中率低）；

3. 解析SQL：包括词法分析和语法分析，词法分析识别出关键字和非关键字，语法分析根据语法规则判断语法是否有误，如果没问题就构建语法树，方便后续模块获取SQL类型、表名、字段、where条件等；

4. 执行SQL：包括预处理、优化、执行阶段；

预处理：检查表或字段是否存在、将select\*中的\*替换为表上所有列；

优化：确定效率最高的执行计划，比如表里有多个索引的时候，优化器基于查询成本选择索引；

执行：根据执行计划从存储引擎读取记录，返回给客户端。

**\*\*undo log、redo log、 bin log是用来干什么的？怎么工作的？**

1. undo log：是innodb生成的日志，实现事务的原子性，主要用于事务回滚和mvcc；

工作方式：每当存储引擎对一条记录进行增删改时，就要把回滚需要的信息记录到undo log中（比如插入记录时就要把主键记下来，回滚删了主键对应的记录就好）；

undo log和数据页一样，都需要通过redo log保证持久化；

在说明redo log前先讲讲缓冲池，innodb为缓冲池申请了一片连续的内存空间，以页为单位存储数据，包括索引页、数据页、undo页等等，这样可以减少磁盘io，提高数据库的读写性能；

但是缓冲池基于内存，是不可靠的，万一断电，没来得及落盘的脏页数据就会丢失；为了防止这种情况，innodb使用write-ahead logging技术：先将修改以redo log的形式记录，将redo log刷盘，再在适当的时候将脏页刷新到磁盘中；

2. redo log：是innodb生成的物理日志（记录在某个数据页做了什么修改），保存的是未被刷入磁盘的脏页日志，只要先将redo log持久化到硬盘中，就不需要马上将脏页落盘；当系统崩溃时，可以通过redo log恢复数据；

工作方式：redo log在缓冲池中也有一个redo log buffer，落盘时机有：

1. mysql正常关闭；

2. redo log buffer中记录的写入量大于buffer内存空间的一半；

3. 每隔1s；

4. 事务提交时；

总结redo log的用处：Redo log是MySQL中用于保证持久性的重要机制之一。它通过以下方式来保证持久性：

Write-ahead logging（WAL）：在事务提交之前，将事务所做的修改操作记录到redo log中，然后再将数据写入磁盘。这样即使在数据写入磁盘之前发生了宕机，系统可以通过redo log中的记录来恢复数据。

redo log的顺序写入：redo log采用追加写入的方式，将redo日志记录追加到文件末尾，而不是随机写入，提高写入性能。

3. bin log：是server层生成的二进制日志（所有存储引擎可用），记录所有表结构、数据修改的日志，用于备份、主从复制；

工作方式：MySQL在完成一条更新操作后，server层会生成一条binlog，保存在server层的binlog cache，事务提交时统一将事务产生的binlog刷新到磁盘中的binlog文件；

**Q：redo log和binlog的区别？**

redo log用于掉电等故障恢复；

binlog用于备份恢复、主从复制；

当需要恢复整个数据库时，不能用redo log恢复，因为redo log是循环写，只记录未落盘的脏页日志；而binlog保存的是全量日志。

**什么是非关系型数据库？**

采用键值对、文档、图等形式存储，读写性能高，易扩展，为了解决关系数据库在处理大规模数据和高并发访问面临的挑战而设计；

redis是键值对型数据库。

**MySQL有哪些存储引擎？**

InnoDB和MyISAM，MySQL5.1后默认使用InnoDB；

区别：

1. innodb提供事务支持，遵循acid属性，myisam不支持，但执行速度更快；

2. innodb是聚簇索引（数据和索引绑定），myisam是非聚簇索引（索引只保存数据的指针）；

3. innodb支持外键（表的一个字段，值为另一个表的主键），myisam不支持；

4. innodb最小锁粒度是行锁（提高了并发性能），myisam是表锁；

5. InnoDB支持崩溃恢复和故障转移，而MyISAM在崩溃后无法保证数据的完整性；

总的来说，如果需要支持事务、并发性能高、要求数据完整性，就选择innodb，以查询为主，要求速度快就选择myisam。

**什么是索引？索引的分类？**

索引是帮助存储引擎快速获取数据的数据结构，即数据的目录；

按数据结构：b+树索引、hash索引、full-text索引；

按物理存储：聚簇索引、二级索引；

按字段特性：主键索引、唯一索引、普通索引、前缀索引；

按字段个数：单列索引、联合索引。

**Mysql创建索引需要注意什么？**

Mysql支持多种索引类型，要根据实际需求选择合适的索引类型，因为索引会增加数据插入、更新和删除的成本，所以不能为每个列都创建索引，而是要在经常查询的列上创建索引，也根据需求选择联合多个列构建索引。

**mysql慢查询优化的方法？**

1. 通过explain执行结果，查看sql是否走索引（type字段是all表示全表扫描，key字段显示实际使用的索引），如果不走就考虑增加索引；

2. 建立联合索引，实现覆盖索引优化，减少回表；

3. 避免索引失效；

4. 联表查询最好手动指定小表驱动大表（有时候优化器会因为统计数据出问题而不去优化），减少表连接创建的次数；

5. 字段多的表分解成多个表，把字段使用频率低的放在单独的表中，只在需要时进行关联查询，减少了不必要的数据读取和处理；

**mysql为什么用b+树而不是b树、二叉树、hash？**

MySQL 默认的存储引擎 InnoDB 采用的是 B+ 作为索引的数据结构，原因有：

1. B+ 树的非叶子节点不存放实际的记录数据，仅存放索引，因此数据量相同的情况下，相比节点中存储索引和记录的 B 树，B+树的非叶子节点可以存放更多的索引，因此 B+ 树可以比 B 树更「矮胖」，查询底层节点的磁盘 I/O次数会更少，效率更高。

2. B+ 树有大量的冗余节点（所有非叶子节点都是冗余索引），这些冗余索引让 B+ 树在插入、删除的效率都更高，不会像 B 树那样会发生复杂的树的变化；

3. B+ 树叶子节点之间用双向链表连接了起来，有利于范围查询，而 B 树要实现范围查询，因此只能通过树的遍历来完成范围查询，这会涉及多个节点的磁盘 I/O 操作，范围查询效率不如 B+ 树。

二叉树的复杂度是O(log2N)，而b+树的复杂度是O(logmN)，其中m是b+树的阶数；实际应用中m一般大于100，这保证了即使数据达到千万级别，b+树的高度依然维持在3/4层，一次查询只需要3/4次磁盘io，相比之下二叉树的磁盘io要多得多；

hash做等值查询效率快，但不适合做范围查询。

**增加B+树的路数可以降低树的高度，那么无限增加树的路数是不是可以有最优的查找效率？**

无限增加树的路数是不可行的；

降低树高是为了减少磁盘io次数，增加路数意味着增加非叶子节点中存放的索引，非叶子节点存储在数据页中，而innodb的数据是按数据页为单位读写的，默认16kb，无限增加路数会导致分页，使得访问一个节点需要读取两页以上的数据页，io次数反而会增加。

**索引失效有哪些？**

1. 联合索引不满足最左匹配原则；

2. 对索引使用左或左右模糊匹配；

3. 对索引使用函数或表达式计算；

4. 对索引隐式类型转换（查询varchar字段不加引号）；

5. WHERE使用OR，且其中一个条件不是索引；

6. select \*；

**事务四大特性以及InnoDB怎么保证：**

1. 原子性：事务是不可分割的最小工作单元，要么全部完成要么全部没完成，如果过程中出错会回滚到原状态；【undo log回滚日志】

2. 一致性：事务执行前后，数据满足完整性约束，数据库保持一致性状态；【AID】

3. 隔离性：允许多个并发事务同时对数据读写，其执行互不干扰【MVCC或锁】

4. 持久性：事务一旦被提交，对数据的修改是永久的。【redo log 重做日志】

**并行事务引发的问题以及隔离级别？**

1. 脏读：读到其他事务未提交的数据；

2. 不可重复读：在一个事务内多次读取**同一数据**内容不一致；

3. 幻读：一个事务内多次查询符合条件的记录数量，前后两次查询的**记录数量**不一致；

隔离级别：

读未提交 < 读提交【解决脏读】 < 可重复读【解决脏读、不可重复读】 < 串行化【所有】

**MVCC多版本并发控制？**

用于在多个事务同时读取和修改数据库时提供一致性和隔离性，核心思想是在每个记录上通过隐藏字段维护多个版本，每个事务在读取数据时，根据自己的事务id选择合适的版本，这样每个事务可以看到一个一致性的数据库快照，不会受到其他正在并发执行的事务的影响。

读提交隔离级别：读数据时创建Read View；

可重复读：启动事务时创建Read View；

实现方式：

1. 每行记录末尾有两个隐藏列：trx\_id、roll\_pointer

2. 通过m\_ids（活跃事务列表），min\_trx\_id（m\_ids的最小值），max\_trx\_id（全局最大事务id+1）将所有事务基于当前执行的事务分为已提交、未提交、未开始状态；

3. 当前执行的事务对数据的可见条件：自己更新的、已提交的事务；

4. 其余不可见的记录会根据回滚指针读取首个trx\_id满足可见条件的记录。

**MySQL怎么解决幻读问题？**

快照读（普通select）：通过MVCC解决，启动事务时创建Read View，每次查询都在undo log版本链中找到事务开始时的数据；

当前读（除了快照读都是当前读，如update、insert等）：加next-key锁；

MVCC在可重复读隔离级别下无法完全解决幻读：

例1：A快照读，B插入，A当前读，A的两次读不一致；【尽量在事务开始时执行当前读，以对其加锁，防止其他事务修改】

例2（重要）：A快照读，B插入，A更新B插入的记录（更新一个本来自己快照中不存在的记录，该记录id变成了A，快照中可见了），A快照读，A的两次读不一致；

**两阶段锁是什么？**

锁在一个阶段被获取，另一个阶段被释放。

当事务正在执行时，事务持有的锁的数量会增长；在事务结束时，锁被释放，事务持有的锁的数量减少。

目的是实现单个数据库内的可串行化隔离级别。

**两阶段提交是什么？https://waylau.com/two-phase-commitment-protocol/**

是分布式事务中保证数据一致性的方法，即保证了分布式事务的原子性：所有节点要么全做要么全不做；

两阶段提交包含一个协调者和多个参与者；

准备阶段：协调者将通知事务参与者准备提交或取消事务，写本地的redo和undo日志，但不提交，然后进入表决过程。在表决过程中，参与者将告知协调者自己的决策：同意（参与者本地作业执行成功）或取消（本地作业执行故障）；

提交阶段：协调者将基于第一个阶段的投票结果决定提交或者取消事务。当且仅当所有的参与者同意提交事务，协调者才通知所有的参与者提交事务，否则协调者将通知所有的参与者取消事务。

**linux和docker相关：**

**linux常用命令**

cat /proc/cpuinfo 查看cpu核心数

**docker是什么？**

docker是开源的容器化平台，提供了轻量级、可移植、可扩展的容器化解决方案；

docker的核心概念是容器，容器是独立、可隔离的运行环境，包括应用程序和依赖项。

**docker用什么隔离？**

使用namespace隔离，namespace是linux内核提供的隔离机制，用于隔离进程的资源视图，包括进程、网络、挂载、用户命名空间等，使每个容器都有自己独立的空间；

使用control groups对资源进行限制，比如cpu、内存等。

**docker网络模式？**

1. host：和宿主机共享网络命名空间；

2. container：和其他容器共享网络命名空间；

3. none：容器有独立的网络命名空间，但没有进行任何网络配置；

4. bridge（默认）：主机上创建docker0虚拟网桥，docker0分配IP给容器使用，容器的默认网关为docker0的IP地址；同时主机上创建一对虚拟网卡，一端（eth0）放在容器中作为容器的网卡，另一端放在主机中，并将网卡加入到docker0网桥中。

**docker和k8s之间的关系？**

如何设计一个高性能、高并发、高可用、高可靠、可扩展的系统

1).使用CPP提供的高性能封装的操作函数，如emplace\_back()、move()等

2).可以使用操作系统提供的mmap内存映射，提高数据的传输效率：内存映射将用户空间的一段内存区映射到内核空间，用户和内核都可以直接对空间里的数据进行修改，在减少了“磁盘数据->内核空间->用户空间”中最后一步

3).池化技术：创建线程池、数据库/TCP连接池、对象池等，提高对象的复用，减少重复创建对象、销毁对象所需的系统开销。

4）.高效利用缓存：基于空间换时间的方法，解决高并发读情景。针对读密集型、热点数据进行缓存；还可以使用分布式缓存，将缓存数据通过hash算法分布在多台机器上，进一步缓解压力。

5）.异步化：在处理耗时的任务时，采用同步等待的方式会严重降低系统的吞吐量，可以通过异步化进行解决。

**Git相关命令的使用**

git add .

git commit -m ""

git push 将本地修改提交至远程库中

gitk 查看所有提交情况

git fetch + git merge 先将“远程仓库”拉到”本地远程连接“中，便于后续解决”本地远程连接“和”本地新版本“的冲突，然后合并更新”本地新版本“，最终再push到“远程仓库”

git status

git stash 暂存未提交的工作变动

git cherry-pick <提交号> 将单个提交所做的变动应用到当前分支中

**研电赛项目相关：**

**项目介绍？**

项目背景：我国人口老龄化程度加深，市面上面向老人的监护产品存在不足，独居老人发生意外的风险更高，所以我们设计了面向独居老人的智能居家监护系统；

功能：我们的系统基于云边端架构实现，由云端服务器、语音识别模块、姿态检测模块、机器人模块、微信小程序构成。其中云端服务器负责系统各模块之间的通信，语音识别模块负责识别老人的关键词；姿态检测模块对老人的姿态进行推理；机器人模块则根据检测识别模块发出的信息执行相应的行为；小程序是家人远程关注老人状态的方式；

主要工作：

1. 构思项目的功能需求，设计系统的总体架构，探索各个功能模块的技术可行性（包括在R329上部署离线语音库、DK610能否装上ubuntu作为控制机器人的上位机）；

2. 实现系统各个设备之间的通信；

3. 机器人的所有功能开发，包括房间分区巡逻、狭窄空间导航的优化、全局路径优化、部署目标检测算法实现自动寻人的功能。

**项目难点？**

1. 机器人操作系统ros依赖ubuntu，但比赛要求使用的开发板官方不提供ubuntu镜像；烧别的开发板的ubuntu镜像出现各种奇奇怪怪的问题；

解决方式：用别的厂商的CPU相同的开发板的内核和ubuntu构建脚本，把DK610内核里的设备树搬过去，就构建好了；

2. 处理各种细节上的报错：特别是将检测算法部署到机器人上，安装ncnn，将pytorch训练好的模型转成ncnn支持的模型都出现了各种疑难杂症。

**mqtt有什么优势？怎么实现安全传输？**

首先mqtt协议设计简单，使用二进制编码，减轻了网络带宽消耗，具有轻量级、低宽带消耗的优势；

其次mqtt的设计目标之一是低功耗，因此适用于资源受限的设备；

而且mqtt有很多开源实现和各种语言的客户端库，在各种平台设备上得到广泛支持；

总体来说它适用于各种规模的物联网部署。

项目中使用emqx作为mqtt代理服务器；

机器人和各感知设备间的通信用到了用emqx的客户端认证功能，在服务器设置用户名和密码，客户端需要提供正确的凭证才能建立连接；

微信小程序的安全性要求比较高，因此与小程序的连接使用了TLS/SSL加密，具体步骤为：申请阿里云服务器和域名，为域名配置证书和密钥，emqx服务器开启mqtt over websocket功能，目的是将mqtt协议包装成ws协议，再将提前准备好的域名配进nginx服务器反向代理，将ws协议升级成wss协议，最终完成mqtt服务器与小程序间的通信。