一、CPP 基础

01 语言理解

1. C 和 C++ 的区别

- 编程范式：C++ 是面向对象的语言言，而 C 是面向过程的语言；
- 语法：C++ 引入 new/delete 运算符，取代了C中的 malloc/free 库函数；C++引入引用的概念，而而C中没有；新增了 bool 数据类型
- 函数：C++引入函数重载和模板，而 C 中没有
- 类：C++引入类的概念，并增加了虚函数的概念，由于实现多态

2. 面向对象和面向过程编程的区别

面向过程编程的概念：是进行以模块功能和处理过程设计为主的详细设计的基本原则

结构化程序设计的不足：随着程序规模的扩大，难以**理解**、难以**扩充**、难以**查错**、难以**重用。**

面向对象编程的概念：客观事物的**抽象**过程，归纳客观事物的**属性**，归纳客观事物的**行为**

面向对象程序设计的四个特点：抽象、封装、继承、多态。**封装**：通过某种语法形式，将客观事物抽象的数据结构和对应操作的函数捆绑在一起，形成一个**类 (class)**，从而使得数据结构和操作函数呈现出清晰的关系，这称为**封装。**

3. C++ 和 Java 的区别

- 指针：Java 语言让程序员没法找到指针来直接访问内存，**没有指针的概念**，并有内存的自动管理功能，从而有效的防止了 C++ 语言中的指针操作失误的影响。
- 垃圾回收：自动内存管理，Java 自动进行无用内存回收操作，不需要程序员进行手动删除。而 C++ 中必须由程序员释放内存资源 delete/delete[]
- 类：Java 中取消了 C++ 中的 struct 和 union

4. C++ 和 Python 的区别

- 执行方式不同：c++ 是编译型语言需要经过编译才能运行；Python是解释型的脚本语言，不需要经过编译。
- 语言类型不同：c++属于静态类型；Python属于动态类型，但都是强类型语言不能进行隐式转换。
- 运行效率不同：c++的运行效率要比Python快。

02 内存管理

1. C++ 中内存分配情况

栈区：由**编译器自动**管理分配和回收，存放**局部变量和函数参数。**

堆区：由**程序员手动**管理，需要手动通过代码 new malloc delete free 完成内存空间的分配和回收，空间较大，一般比较**复杂的数据类型**都是放在堆中，但可能会出现内存泄漏和空闲碎片的情况。

静态区：分为初始化和未初始化两个相邻区域，存储初始化和未初始化的**全局变量和静态变量。**（全局变量：出现在代码块{}之外的变量就是全局变量；静态变量：是指内存位置在程序执行期间一直不改变的变量，用关键字static修饰。）

代码区：存放程序的二**进制代码**，这块内存在程序运行期间是不变的。

2. 堆和栈区别

栈

- 由**编译器自动管理**，在需要时由编译器自动分配空间，在不需要时候自动回收空间，一般保存的是局部变量和函数参数等。
- **连续的内存空间**，在函数调用的时候，首先入栈的主函数的下一条可执行指令的地址，然后是函数的各个参数。大多数编译器中，参数是从右向左入栈（原因在于采用这种顺序，是为了让程序员在使用C/C++的"函数参数长度可变"这个特性时更方便。如果是从左向右压栈，第一个参数（即描述可变参数表各变量类型的那个参数）将被放在栈底，由于可变参的函数第一步就需要解析可变参数表的各参数类型，即第一步就需要得到上述参数，因此，将它放在栈底是很不方便的。）本次函数调用结束时，局部变量先出栈，然后是参数，最后是栈顶指针最开始存放的地址，程序由该点继续运行，**不会产生碎片**。
- 栈是高地址向低地址扩展，栈低高地址，**空间较小**。

堆

- 由**程序员管理**，需要手动 new malloc delete free 进行分配和回收，如果不进行回收的话，会造成内存泄漏的问题。
- **不连续的空间**，实际上系统中有一个空闲**链表**，当有程序申请的时候，系统遍历空闲链表找到第一个大于等于申请大小的空间分配给程序，一般在分配程序的时候，也会空间头部写入内存大小，方便 delete 回收空间大小。当然如果有剩余的，也会将剩余的插入到空闲链表中，这也是产生内存碎片的原因。
- 堆是低地址向高地址扩展，**空间交大**，较为灵活。

3. malloc / free 和 new / delete 的区别

相同点：

都可以用来在堆上分配和回收空间。

不同点：

- `malloc/free` 是 C/C++ 语言的标准**库函数**；`new/delete` 是 C++ 中的**运算符。**
- `malloc` 返回类型是 `void*`，使用时需要类型转换；而 `new` 的返回类型是指向**对象类型的指针。**
- `malloc/free` 只分配和回收**指定大小**的堆内存空间，而 `new/delete` 可以**根据对象类型**分配和回收合适的堆内存空间。
- 使用 `new/delete` 将调用构造函数或析构函数初始化或析构对象，而 `malloc/free` 没有该操作。

new/delete 的执行过程：

- 执行 new 实际上执行两个过程：
  - 分配未初始化的内存空间（malloc）。如果分配空间出现问题，则抛出 `std::bad_alloc` 异常，或被某个设定的异常处理函数捕获处理。
  - 使用对象的构造函数对空间进行初始化；返回空间的首地址。如果在构造对象时出现异常，则自动调用 delete 释放内存。
- 执行 delete 实际上也有两个过程：
  - 使用析构函数对对象进行析构；
  - 回收内存空间（free）。

为什么有了 malloc / free 还需要 new / delete?

因为对于非内部数据类型（复杂数据结构）而言，光用 `malloc/free` 无法满足动态对象的要求。对象在创建的同时需要自动执行构造函数，对象在消亡以前要自动执行析构函数。

由于 `mallo / free` 是库函数而不是运算符，不在编译器控制权限之内，不能够把执行的构造函数和析构函数的任务直接强加于 `malloc/free`，所以有了 `new / delete` 运算符对其进行进一步的扩展。

4. 什么是内存泄漏和内存溢出

内存泄漏：

内存泄漏简单的说就是申请了一块内存空间，**使用完毕后没有释放掉**，占用了有用内存。 它的一般表现方式是程序运行时间越长，占用内存越多，最终用尽全部内存，整个系统崩溃。由程序申请的一块内存，且没有任何一个指针指向它，那么这块内存就泄漏了。

内存溢出：

内存溢出是指程序在申请内存时，**没有足够的内存空间供其使用。**内存泄漏的堆积最终会导致内存溢出内存溢出，就是程序申请的内存空间超过了系统实际可分配的空间，此时系统无法满足程序的需求，就会报内存溢出的错误。

如何检测内存泄漏：

- 首先可以通过观察猜测是否可能发生内存泄漏，Linux 中使用 swap 命令观察还有多少可用的交换空间，在一两分钟内键入该命令三到四次，看看可用的交换区是否在减少。
- 还可以使用其他一些 `/usr/bin/stat` 工具如 netstat、vmstat 等。如发现波段有内存被分配且从不释放，一个可能的解释就是有个进程出现了内存泄漏。
- 当然也有用于内存调试，内存泄漏检测以及性能分析的软件开发工具 valgrind 这样的工具来进行内存泄漏的检测。

5. 结构体内存对齐方式与作用

结构体作为一种复合数据类型，其构成元素既可以是基本数据类型的变量，也可以是一些复合型数据。对此，编译器会自动进行成员变量的对齐以提高运算效率。默认情况下，按自然对齐条件分配空间。各个成员按照它们被声明的顺序在内存中顺序存储，第一个成员的地址和整个结构的地址相同，向结构体成员中size最大的成员对齐。

经过内存对齐之后，CPU 的内存访问速度大大提升。因为 CPU 把内存当成是一块一块的，块的大小可以是 2，4，8，16 个字节，因此 CPU 在读取内存的时候是一块一块进行读取的，块的大小称为内存读取粒度。比如说 CPU 要读取一个 4 个字节的数据到寄存器中（假设内存读取粒度是 4），如果数据是从 0 字节开始的，那么直接将 0-3 四个字节完全读取到寄存器中进行处理即可。

6. 什么是栈溢出

- **局部数组过大：**当函数内部的数组过大时，有可能导致堆栈溢出。
- **递归调用层次太多：**递归函数在运行时会执行压栈操作，当压栈次数太多时，也会导致堆栈溢出。
- **指针或数组越界：**这种情况最常见，例如进行字符串拷贝，或处理用户输入等等。当拷贝的字符串大大的超过变量空间就会破坏堆栈

解决这类问题的办法有两个

- 增大栈空间
- 改用动态分配，使用堆（heap）而不是栈（stack）

7. 什么是堆破坏

堆破坏是指没控制好自己的指针，把不属于你分配的那块内存给写覆盖。

一般来说，堆破坏往往都是**写数据越界造成的**，所以微软在堆分配上，给程序员门额外提供了2种堆分配模式--完全页堆（full page heap），准页堆(normal page heap)，用来检测堆被写越界的情况。

03 指针与引用

1. 指针和引用的区别

引用的概念：某个变量的引用，等价于这个变量，相当于该变量的一个**别名**；**本质上来说引用是指针常量，可以修改指向的值，但是不能修改地址。**

指针和引用的区别：

- 在声明引用时一定要对其**初始化**；而指针**不需要初始化**
- 引用经初始化后，就不可以再和其他对象绑定在一起了，一直引用该对象（从一而终），**不能被改变**；指针**可以改变**其所指向的对象。
- 引用只能引用变量，不能引用常量和表达式，且**不存在指向空值**的引用；但是指针**可以指向空值。**

04 基础语法

1. const 关键字的作用

- 修饰变量，说明该变量不可以被改变；
- 修饰指针，分为指向常量的指针（pointer to const）和自身是常量的指针（常量指针，const pointer）；
- 修饰引用，指向常量的引用（reference to const），用于形参类型，即避免了拷贝，又避免了函数对值的修改，相当于常指针常量；
- 修饰成员函数，说明该成员函数内不能修改成员变量。
- 修饰对象，常量对象只能调用常量函数。

2. 宏定义 #define 和 const 的区别

区别	#define	const
编译器处理方式	预处理器处理	编译器处理
类型安全检查	无类型安全检查	有类型安全检查
存储方式	存储在代码段、代码字符替换、不分配内存	存储在数据段、常量声明、分配内存
定义域	不受定义域限制	只在定义域内有效

- 编译器处理方式不同： #define 宏是在预处理阶段展开，不能对宏定义进行调试；而 const 常量量是在编译阶段使用；
- 类型和安全检查不同： #define 宏没有类型，不不做任何类型检查，仅仅是代码展开，可能产生生边际效应等错误；而 const 常量有具体类型，在编译阶段会执行类型检查；
- 存储方式不同： #define 宏仅仅是代码展开，在多个地方进行字符串替换，不会分配内存，存储于程序的代码段中；而 const 常量会分配内存，但只维持一份拷贝，存储于程序的数据段中。
- 定义域不不同： #define 宏不受定义域限制，而 const 常量只在定义域内有效。

3. 宏定义 #define 和 typedef 的区别

4. static 关键字的作用

- 修饰局部变量：修改局部变量的**存储区域和生命周期**，使其存储在静态区；只能在首次函数调用中进行首次初始化，之后的函数调用不再进行初始化；其生命周期与程序相同，但其作用域为局部作用域，并不能一直被访问。
- 修饰全局变量：修改全局变量的**作用域范围**，使变量仅能在文件内可访问，不加修饰可以在整个工程中被访问。该变量在整个**文件内**都可被访问，但是对之外的文件不可见。
- 修饰普通函数：表明函数的作用范围，仅在定义该函数的**文件内**才能使用。在多人开发项目时，为了防止与他人命名空间里的函数重名，可以将函数定位为 static。
- 修饰成员变量：修饰成员变量使所有的对象只维持**一份拷贝**，实现不同对象之间的**数据共享**；而且不需要生成对象就可以访问该成员，一般在类外部完成初始化。
- 修饰成员函数：修饰成员函数使得不需要生成对象就可以访问该函数，但是在 static 函数**只能访问静态成员。**

5. 为什么 static 成员要在类外部完成初始化

因为被static声明的类静态成员变量，其实体远在 `main()` 函数开始之前就已经在全局数据段中存在了，其生命期和类对象是异步的。

这时类对象的生命期还没有开始，如果要在类内初始化类静态成员变量，那么其初始化操作就需要依赖于类的实例化，这样就无法实现**不生成对象就可以访问该成员变量的静态语意了。** 静态语意说明即使没有类实体的存在，其静态成员变量的实体也是存在并且可访问的。

05 面向对象

1. 面向对象的三大特性

- 封装：将客观事物封装成抽象的类，而类可以把自己的数据和方法暴露给可信的类或者对象，对不不可信的类或对象则进行信息隐藏。更抽象的说：**封装就是将完成一个功能所需要的所有东西都放在一起，对外部只提供调用的接口。**
- 继承：可以使用现有类的所有功能，并且无需重新编写原来的类即可对功能进行拓展；**即子类继承父类的特性，并在此基础上进行扩展。**
- 多态：一个类实例的相同方法在不同情形下有不同的表现形式，使不同内部结构的对象可以共享相同的外部接口。即**一个函数，多种实现**，多态性的表现在于程序运行时根据调用对象的不同，调用不同的函数。多态与非多态的实质区别就是函数地址是早绑定还是晚绑定的。如果函数的调用，在编译器编译期间就可以确定函数的调用地址，并产生代码，则是静态的，即地址早绑定。而如果函数调用的地址不能在编译器期间确定，需要在运行时才确定，这就属于晚绑定。

2. 怎么实现多态

多态分为静态多态和动态多态，其中：

- 静态多态是通过**重载**和**模板**技术实现的，在**编译期间确定**；
- 动态多态是通过**虚函数和继承关系**实现的，执行动态绑定，在**运行期间确定。**

3. 什么是虚函数，它是怎么实现的

在类的定义中，有 `virtual` 关键字的成员函数就是虚函数。

虚函数的实现：在有虚函数的类中，类的最开始部分是一个虚函数表的指针，这个指针指向一个虚函数表，表中放了虚函数的地址。当子类继承了父类的时候也会继承其虚函数表，当子类重写父类中虚函数时候，会将其继承到的虚函数表中的地址替换为重新写的函数地址。因为虚函数表和查询的存在，使用虚函数，会增加访问内存开销，降低效率。

4. 动态多态的作用

在面向对象的程序设计中使用多态能够**增强程序的可扩充性和代码可复用性**，即程序需要修改或增加功能的时候，避免大量代码的改动和增加。

但是多态的机制会增加程序执行时在时间和空间上的开销，空间上是由于存在虚函数的类的每个对象在创建时都会多出 4 个字节的额外空间开销用于**存放虚函数表的地址**；时间上是由于在**查虚函数表**的过程中需要消耗一定的时间。

5. 动态多态的实现原理

多态的关键在于通过**基类指针**或**基类引用**调用虚函数时，编译时还不确定该语句调用的是基类函数还是派生类函数，直到运行时才能确定，这种机制也称为**动态联编**。

具体过程：当编译器发现类中有虚函数时，会创建一张虚函数表，把虚函数的函数入口地址放到虚函数表中，并且在对象中增加一个指针 `vptr` ，用于指向类的虚函数表。当派生类覆盖基类的虚函数时，会将虚函数表中对应的指针进行替换，从而调用派生类中覆盖后的虚函数，实现动态联编。

6. struct 和 class 的区别

C++ 中保留了 C 语言的 struct 关键字，并且加以扩充。在 C 语言中，struct 只能包含成员变量，不能包含成员函数。而在 C++ 中，struct 类似于 class，既可以包含成员变量，又可以包含成员函数，能实现继承和多态。

C++ 中的 struct 和 class 基本是通用的，唯有几个细节不同：

- 使用 class 时，类中的成员默认都是 private 属性的；而使用 struct 时，结构体中的成员默认都是 public 属性的。
- class 继承默认是 private 继承，而 struct 继承默认是 public 继承
- class 关键字还可以代替 `typename` 用于模板参数声明，而 struct 不可以

7. 什么是纯虚函数，它有什么作用

纯虚函数是没有函数体的虚函数，`virtual void Print() = 0;`

定义纯虚函数是为了实现接口，这种接口又称为**抽象类**，即包含纯虚函数的类。抽象类是起到规范的作用，想要继承这个类就必须实现所有纯虚函数，否则派生类仍为抽象类。

8. 抽象类和接口的区别

抽象类和接口都不能被实例化，但是他们的用途不一样：

- 抽象类主要目的是实现**代码复用**，即不同类具有某些相同属性和行为时，可以将这些相同属性抽象出来构成一个抽象类，这些不同的类可以通过继承该类实现代码复用。
- 接口的主要作用是**约束类的行为**，即一种强制不同类具有相同行为的机制。
- 一个类只能继承一个抽象类，但是一个类可以实现多个接口

9. 基类构造函数为什么不能是虚函数

虚函数的调用依赖于虚函数表，而指向虚函数表的指针 vptr 需要在构造函数中进行初始化，所以无法调用定义为虚函数的构造函数。

10. 为什么不能在构造函数中调用虚函数

因为派生类调用构造函数期间会先调用基类构造函数，如果基类构造函数中存在多态并调用了派生类的虚函数，这时派生类还未调用构造函数完成初始化，这将导致错误的运行结果。

11. 析构函数为什么一般写成虚函数

当通过基类指针销毁派生类对象时，通常只有基类的析构函数被调用，只能销毁派生类对象中的部分数据，这就出现了析构不完整的情况。为此，可以将基类的析构函数声明为虚函数，派生类的析构函数就不需要进行虚函数声明，自动成为虚函数；这时，通过基类指针析构派生类对象时调用的就是派生类的析构函数，派生类的析构函数中会调用基类的析构函数，从而销毁派生类对象中的所有数据。

一般来说，一个类定义了虚函数，则应该将该类的析构函数声明为虚函数；另外，如果一个类要被作为基类使用，则也应该将该类的析构函数声明为虚函数。

12. 深拷贝和浅拷贝的区别

浅拷贝只复制指向某个对象的**指针**，而不复制对象本身，新旧对象还是**共享一块内存**；而深拷贝会**创造**一个相同的对象，新对象与原对象**不共享内存**，修改新对象不会影响原对象。

当出现类的等号赋值时，会调用拷贝函数，在未定义显示拷贝构造函数的情况下， 系统会调用默认的拷贝函数 - 即浅拷贝，它能够完成成员的一一复制。当数据成员中没有指针时，浅拷贝是可行的。

但当数据成员中有指针时，如果采用简单的浅拷贝，则两类中的两个指针指向同一个地址，当对象快要结束时，会调用两次析构函数，而导致指野指针的问题。这时必需采用深拷贝。深拷贝与浅拷贝之间的区别就在于深拷贝会在堆内存中另外申请空间来存储数据，从而也就解决来野指针的问题。

简而言之，当**数据成员中有指针**时，必需要用深拷贝更加安全。

13. 重载、重写与重定义

- **重载**：是指同一可访问区内被声明的几个具有**不同参数列表**（参数的类型，个数，顺序不同）的同名函数，根据参数列表确定调用哪个函数，重载不关心函数返回类型。
- **重写(覆盖)**：是指派生类中存在重新定义的函数。**其函数名，参数列表，返回值类型，所有都必须同基类中被重写的函数一致。**只有**函数体不同**（花括号内），派生类调用时会调用派生类的重写函数，不会调用被重写函数。重写的基类中被重写的函数必须为虚函数。
- **重定义(隐藏)**：是指派生类的函数屏蔽了与其**同名非虚函数**的基类函数；注意只要同名函数，不管参数列表是否相同，基类函数都会被隐藏。

06 C++ STL

1. vector 与 array 的区别

vector 和 array 的数据组织和操作方式相近，两者的区别在于对空间运用的灵活性。

array 是静态空间，一旦配置就不可变更，如果需要扩大空间就需要自行重新配置新的空间，然后将元素从旧址搬到新址，并把旧空间释放掉。

vector 是动态空间，随着元素的加入，他的内部机制会自行扩充空间以容纳新元素，不需要手动配置空间，增强了内存的合理使用和运用的灵活性。

2. vector 的底层原理

vector底层数据结构是一个**线性连续空间**，包含三个迭代器器，`start` 和 `finish` 之间是已经被使用用的空间范围，`end_of_storage` 是整块连续空间包括备用用空间的尾部。

运用 `star, finish, end_of_storage` 三个迭代器，vector 可以提供如下机能：

- 首尾标示：`begin(), end()`
- 已使用大小：`size()`
- 总容量：`capacity()`
- 判空：`empty()`
- 括号运算符重载：`operator[]()`
- 最前端和最后端元素值：`front(), back()`

3. vector 的扩容机制

vector 实现的关键在于其对大小的控制以及重新配置是的数据移动效率。当 vector 的空间满载时，需要新插入一个元素，如果这时仅仅扩充一个元素的空间，会大大增加系统开销。

因为扩容往往不是直接增加空间，而是要经过**配置新空间 >> 数据移动 >> 释放旧空间**的繁琐过程，往往需要很高的时间成本。所以，为了减少扩容次数，当新增元素是，超过了当前的容量，则会将 vector 的容量根据扩容因子（一般为1.5 或 2）进行扩容，预留更多的备用空间。

为什么扩容因子一般为 1.5 或 2？

如果扩容因子过大，将导致堆空间浪费

4. vector resize() 与 reserve() 的区别

resize()

resize() 函数的作用是改变 vector 元素个数，包括两个参数 `resize(n,m)`，其中 第二个参数可以省略，n 代表改变元素个数为 n ,m 代表初始化为 m 。

resize() 可以改变有效空间的大小。如果 n 比 vector 的 size 小，结果是 size 减小到 n ， 然后**删除** n 之后的数据。

resize() 也可以改变总空间的大小。如果 n 比 vector 的 capacity 大，结果是先增加容量，然后再增加 size 并初始化， capacity 和 size 均会被改变

reserve()

reserve() 函数的作用是改变总空间的大小，只有一个参数 `reverse(n)`，n 代表改变扩容到容量 n 。

- 如果 n 的值比大于 vector 的 size，则 capacity 增容到 n ， size 不变。
- 如果 n 的值比小于 vector 的 size，则 capacity ， size 都不变。

当将 reserve()，resize() 用于扩容时，可以直接扩充到已经确定的大大小，可以减少多次开辟、释放空间的问题，从而能够提高效率，减少多次数据拷贝的问题。

5. vector 迭代器失效的情况

空间重新配置导致失效：vector 空间的动态增加，并不是在原空间之后增加连续的新空间，而是直接重新配置一块更大的新空间，然后将数据拷贝到新空减，并将旧空间释放。所以，当对 vector 的操作引起空间重新配置时，指向原 vector 的迭代器全部失效了。

元素被删除导致失效：当删除容器中一个元素后，该迭代器所指向的元素已经被删除，也将造成该迭代器失效。erase 方法会返回下一一个有效的迭代器，所以当我们要删除某个元素时，需要 `it=vec.erase(it);`。

6. vector 有了 push_back() 还要 emplace_back

在引入右值引用，移动构造函数，移动复制运算符之前，通常使用push_back()向容器中加入一个右值元素(临时对象)的时候，首先会调用构造函数构造这个临时对象，然后需要调用拷贝构造函数这个临时对象放入容器中。原来的临时变量释放。这样造成的问题是临时变量申请的资源就浪费。

引入右值引用，移动构造函数后，push_back()右值时就会调用构造函数和移动构造函数。

在这上面有进一步优化的空间就是使用emplace_back，在容器尾部添加一个元素，这个**元素原地构造，不需要触发拷贝构造和移动构造**。而且调用形式更加简洁，直接根据参数初始化临时对象的成员。

```
vector<_Tp, _Alloc>::emplace_back(_Args &&... __args) {
    if (this->_M_impl._M_finish != this->_M_impl._M_end_of_storage) {
        // 同样判断容器是否满了，没满的话，执行构造函数，对元素进行构造，并执行类型转换
        _Alloc_traits::construct(this->_M_impl, this->_M_impl._M_finish,
                                 std::forward<_Args>(__args)...);
        ++this->_M_impl._M_finish; // 更新当前容器大小
    } else
        // 满了的话重新申请内存空间，将新的元素继续构造进来，并且进行类型转换
        _M_realloc_insert(end(), std::forward<_Args>(__args)...);
#if __cplusplus > 201402L
    return back(); // 在 C++14版本之后，添加返回值，返回最后一个元素的引用
#endif
}
```

https://zhuanlan.zhihu.com/p/183861524

- 如果插入vector的类型的构造函数接受多个参数，那么push_back只能接受该类型的**对象**（实例），emplace_back 还能接受该类型的构造函数的参数（多个参数）。
- 如果**传入的是构造函数能接受的参数**，emplace_back() 不需要调用构造函数就可以直接进行原地构造，性能更好。如果传入的是对象实例，这两种方法没什么区别都要调用拷贝构造函数，如果是临时对象就都调用移动构造函数。

7. list 的底层原理

list 的底层是一个**双向链表**，以结点为单位存放数据，结点的地址在内存中不一定连续，每次插入或删除一个元素，就配置或释放一个元素空间。

list 不支持随机存取，适合需要大量的插入和删除，而不关心随即存取的应用场景。

8. deque 的底层原理

deque 的底层是**分段连续线性空间**，他是由一段一段的定量连续空间构成的，一旦有需要就在 deque 的头或尾部配置一段定量连续空间，串接在整个 deque 的头部或尾部。

deque 的最大任务就是维护这些分段连续空间是**整体连续的假象**，并提供了随机存取的接口。通过这种方式避免了 vector 扩容过程中的时间成本与资源消耗，而代价则是复杂的迭代器架构。

数据结构上，deque 使用一小块被称为 map 的连续空间存储指向不同分段连续线性空间的指针，而这些被指向的连续线性空间被称为缓冲区，它们作为存储 deque 元素的主体。

deque 是一个双向开口的连续线性空间（**双端队列**），在头尾两端进行元素的插入和删除操作都有较好的时间复杂度。

9. vector, list, deque 的使用场景

- vector 适用于随机访问频繁，元素对象简单且通过尾部插入的场景。
- deque 虽然功能上比 vector 强大，且支持随机访问，但是其迭代器十分复杂，尽量避免使用 deque。
- list 不支持随机存取，但是适用于对象大，对象数量变化频繁，插入和删除操作较多的使用场景。

10. set, map 的底层原理

set, map的底层实现都是红黑树，红黑树是一种平衡二叉搜索树，有较好的自动排序效果，能够实现 `O(logN)` 的时间复杂度内的搜索、插入和删除操作。

红黑树的性质：

- 所有节点都是红色或者黑色
- 根节点是黑色的
- 每个叶子节点都是黑色的空节点（NULL）
- 父子节点不能同时为红色
- 任意节点到达空节点（NULL）的路径上的黑色节点的数量必须相同

当对红黑树进行插入或删除操作导致上述性质被破坏时，红黑树就出现了不平衡状态，必须要调整节点颜色并旋转树形恢复平衡。

11. 红黑树和 AVL 树的区别

红黑树和 AVL 树都是二叉搜索树，二叉搜索树的查找、插入和删除操作的时间复杂度都是与其自身的高度相关的，平均情况下时间复杂度为 `O(logN)`。但是二叉搜索树可能出现向右倾斜的情况，导致二叉搜索树退化为一个链表，这时二叉搜索树的查找、插入和删除操作的时间复杂度就变成了 `O(N)`。

为了解决这一问题，提出了平衡二叉搜索树，AVL 是一种高度平衡的二叉搜索树，它的平衡条件是**所有非叶子节点的左右子树均为平衡二叉树，且左右子树的高度差的绝对值不超过 1**。这就实现了最坏情况下 AVL 树的查找、插入和删除操作的时间复杂度为 `O(logN)`。

但是由于 AVL 树严格的平衡条件，使得每次插入或删除操作时都可能破坏平衡状态，一旦平衡被破坏就需要通过繁琐的树形旋转进行调整。所以，AVL 树不适用于频繁插入或删除操作的场景。

红黑树在这一问题上进行了优化，它的平衡条件没有 AVL 树那么严格，所以插入或删除操作并不容易被打破，即使平衡被破坏它也可以通过更加简易的颜色和树形调整完成平衡恢复。所以红黑树能够更好的应对频繁插入或删除操作的场景。

12. 红黑树的应用场景

- STL 中 set 和 map 底层实现都是红黑树
- epoll 模型的底层数据结构也是红黑树
- linux系统中 CFS **进程调度算法**，也用到红黑树

13. B 树和 AVL 树的区别

- B 树是多叉搜索树，AVL 树是二叉搜索树
- B 树一个节点中可以存储多个关键字，AVL 树一个节点通常只放一个关键字

14. B 树和 B+ 树的区别

- B+树内节点不存储数据，所有 data 存储在叶节点导致查询时间复杂度固定为 `O(logN)`。而B-树查询时间复杂度不固定，与 key 在树中的位置有关，最好为`O(1)`。
- B+树内节点无 data 域，每个节点能索引的范围更大更精确。由于B树的节点都存了 key 和 data，而B+树只有叶子节点存data，非叶子节点都只是索引值，没有实际的数据，这就时B+树在一次 IO 里面，能读出的索引值更多，从而减少查询时候需要的IO次数。
- B+ 树是基于 B 树和**叶子节点顺序访问指针**进行实现，它具有 B 树的平衡性，并且通过顺序访问指针来提高**范围查询**的性能。

15. 红黑树和哈希表的区别

- **有序性**：红黑树是有序的，Hash是无序的。
- **搜索效率**：红黑树查找和删除的时间复杂度都是`O(logN)`，Hash查找和删除的时间复杂度都是`O(1)`，由于是通过哈希函数实现，所以Hash的查找效率在一般情况下与数据量无关，而红黑树则不同。
- **内存消耗**：红黑树占用的内存更小（仅需要为其存在的节点分配内存），而 Hash 事先应该分配足够的内存存储散列表，即使有些槽可能弃用，如果分配内存小导致哈希冲突情况增多，如果使用开链法解决冲突，过长的链表导致效率下降。

16. unordered_set, unordered_map 的底层实现原理

底层使用 hashtable + buket 的实现原理，hashtable 可以看作是一个数组或者 vector 之类的连续内存存储结构，它可以通过下标来快速定位时间复杂度为 O(1) 处理。其 hash 冲突的方法就是在相同 hash 值的元素位置下面挂 buket（桶），桶使用链表结构实现。

17. 哈希函数的实现原理

哈希法又称散列法、杂凑法以及关键字地址计算法等，相应的表称为哈希表。这种方法的基本思想是：首先在**元素的关键字 *k* 和元素的存储位置 *p* 之间建立一个对应关系 *f*，**使得*p=f(k)*，*f*称为哈希函数。创建哈希表时，把关键字为 *k* 的元素直接存入地址为 *f(k)* 的单元；以后当查找关键字为 *k* 的元素时，再利用哈希函数计算出该元素的存储位置 *p=f(k)*，从而达到按关键字直接存取元素的目的。

18. 哈希冲突的解决方法

哈希冲突：由于哈希算法被计算的数据是无限的，而计算后的结果范围有限，因此总会存在不同的数据经过计算后得到的值相同，这就是哈希冲突。

解决哈希冲突的方法：

1. **开放定址法：**从发生冲突的那个单元起，按照一定的次序，从哈希表中找到一个空闲的单元。然后把发生冲突的元素存入到该单元的一种方法。
    1. **线性探查法**是开放定址法中最简单的冲突处理方法，它从发生冲突的单元起，**依次判断下一个单元是否为空**，当达到最后一个单元时，再从表首依次判断。直到碰到空闲的单元或者探查完全部单元为止。
    2. **线性步长探测法**是在线性探查法的基础上将探测步长从 1 该无 Q，依次判断找到空闲地址。

2. **拉链法：**链接地址法的思路是**将哈希值相同的元素构成一个同义词的单链表**，并将单链表的头指针存放在哈希表的第 i 个单元中，查找、插入和删除主要在同义词链表中进行。链表法适用于经常进行插入和删除的情况。
3. **再哈希法：**就是同时构造多个不同的哈希函数，当 H1 发生冲突时，再用 H2 计算哈希值，直到冲突不再产生，这种方法不易产生聚集，但是增加了计算时间。

19. 怎么实现哈希扩容

`rehash()` 函数重建hash表，将插槽的数量扩展的 n，如果 n 小于目前插槽数量，这个函数并不起作用。

20. sort 方法的底层原理

sort 算法的底层主要是快速排序和插入排序结合实现的。数据量大时采用快速排序进行分段递归排序，一旦分段后的数据两小于某个门槛，为了避免快排的递归调用带来的过大负荷，就改用插入排序。如果快排分割行为恶化为`O(N^2)`时间复杂度时，将导致递归层次过深，此时则改用堆排序。

07 C++ 11 新特性

1. 没有垃圾回收机制会产生什么问题

没有垃圾回收机制会带来了很多内存资源管理不当的问题，例如：

- **野指针**-指向了内存资源已经被释放的空间并被继续使用；
- **重复释放内存**-内存资源在已经被释放的情况下，被试图再次释放导致程序崩溃；
- **内存泄漏**-没有及时释放不再使用的内存资源，致使程序运行过程中占用的内存资源不断累加，最终导致程序崩溃。

2. 野指针和悬空指针的区别

- 野指针：**未初始化的指针**被称为野指针
- 悬空指针：当指针所**指向的对象已经被释放**，但是该指针没有任何改变，以至于其仍然指向已经被回收的内存地址，这种情况下该指针被称为悬空指针

3. 什么是智能指针

智能指针是一个RAII类模型，用于动态分配内存，其设计思想是将基本类型**指针封装为（模板）类**对象指针，并在离开作用域时**调用析构函数**，使用 delete 删除指针所指向的内存空间。

智能指针的作用是，能够处理内存泄漏问题和空悬指针问题。

4. 常用的智能指针有哪几种

- **shared_ptr**： 实现**共享式**拥有的概念，即**多个智能指针可以指向相同的对象**，该对象及相关资源会在其所指对象不再使用之后，自动释放与对象相关的资源；
- **unique_ptr**： 实现**独占式**拥有的概念，同一时间只能有一个智能指针可以指向该对象，不可以进行拷贝构造和拷贝赋值，但是可以进行移动构造和移动赋值；
- **weak_ptr**： 解决 shared_ptr 相互引用时，两个指针的引用计数永远不会下降为0，从而导致死锁的**循环引用问题。**而 weak_ptr 是对象的一种弱引用，**可以绑定到 shared_ptr ，但不会增加对象的引用计数。**

5. shared_ptr 的实现原理

C++ 智能指针底层是采用**引用计数的方式实现**的。简单的理解，智能指针在**申请**堆内存空间的同时，会为其配备一个整形值（初始值为 1）；每当有**新对象使用**此堆内存时，该整形值 +1；反之，每当使用此堆内存的对象**被释放**时，该整形值减 1。当堆空间对应的整形值为 0 时，即表明不再有对象使用它，该堆空间就会被释放掉。

6. weak_ptr是如何解除shared_ptr循环引用的

循环引用的问题：shared_ptr 所管理的对象形成环状的引用，该被调用的析构函数没有被调用，其引用计数无法抵达 0 ，而存在内存泄漏的情况。

> `weak_ptr` 可以从一个 `shared_ptr` 或另一个 `weak_ptr` 对象构造，它的构造和析构不会引起引用记数的增加或减少。形成环状引用的一个对象使用 `weak_ptr` 对象构造就可以打破循环，析构顺序是 `weak_ptr` 对象被优先析构。

7. shared_ptr 和 weak_ptr 是线程安全的吗

shared_ptr 的引用计数在实现上是遵循**原子性**的，所以是线程安全的。

8. shared_ptr 和 weak_ptr 指向的对象是线程安全的吗

但是智能指针其**指向对象的读写**则不是线程安全的。shared_ptr 包含两个主要数据成员，一个是指向对象的指针，一个是引用计数管理对象。当智能指针发生拷贝的时候，先拷贝的是指针，然后拷贝引用计数，这两个操作并不是原子操作，所以无法保证其指向对象的线程安全。

9. Lambda 表达式是闭包的吗

闭包就是能够读取其他函数内部变量的函数

首先 lambda 表达式是无状态的，因为lambda表达式的本质是函数，它的作用就是在给定输入参数的情况下，输出固定的结果。

如果 lambda 表达式中**引用的方法了的局部变量**，则 lambda 表达式就变成了闭包，因为这个时候lambda表达式是有状态的。

10. Lambda 表达式的实现原理

当编写了一个lambda之后，编译器将该表达式翻译成一个匿名类的匿名对象。该类含有一个重载的函数调用运算符。

使用 Lambda 表达式可以减少程序中函数对象类的数量

就向一些临时变量一样，也存在临时函数的情况。有些简单函数或函数对象在整个程序中可能只需要被调用或使用一次。这样一次性的函数，如果为其单独声明函数或者编写一个类，可能降低程序的可读性。而C++11中提供的 Lambda 表达式提供了避免这一问题的方法，使用 Lambda 表达式构建匿名函数。

11. 什么是右值引用

右值是指那些在表达式执行结束后不再存在的数据，也就是临时性的数据；**有名称的、可以获取到存储地址的表达式即为左值；反之则是右值。**

C++11中提出了右值引用使用 && 表示，和声明左值引用一样，右值引用也必须立即进行初始化操作，且只能使用右值进行初始化。**提出右值引用的主要目的是提高程序运行的效率，减少需要进行深拷贝的对象进行深拷贝的次数。**

12. 什么是移动语义

STL.中一个 vector 对象管理的动态内存中的元素数量是会变得，**当分配的内存不够存放对象时，就要重新分配一块更大的内存，然后把原来的对象挪到新的内存中放着，再把原来的内存中的元素挨个销毁，最后将原来的内存空间释放。**

这个过程包含**开辟新空间、拷贝数据、释放旧空间**，这个繁琐的过程既然旧空间要被释放，如果不是需要开辟更大的空间，那么新对象直接接管旧空间将极大的提高性能。

移动语义就是**当原对象拷贝后要就立即被销毁，则此时使用移动而非拷贝对象会大幅提高性能，前提是此类型的对象可以移动。**（移动就是把指针拷贝一下，然后将原来的指针置为空）

https://blog.csdn.net/m0_43436602/article/details/112199418

13. 移动操作的条件

- **移动后源对象必须可正常析构：**从一个对象移动数据并不会销毁此对象，但有时在移动操作后，源对象会被摧毁，故我们必须确保，移后源对象处于一个可析构的状态，即析构这个对象不会对移动构造的新对象造成任何影响——例如string的移动，将移后源对象的指针成员置为空，来避免析构的时候把交出去的内存给释放了。
- **移动操作还必须保证移后源对象仍然是有效的：**对象有效就是仍然可以为其赋值，且对象可以安全的使用。虽然我们保证移后源对象是有效的，但其中留下的值并无要求（值是多少都有可能，这依赖于类型设计者是怎么设计的），故我们在程序中不应再使用移后源对象。

14. 拷贝构造函数和移动拷贝构造函数的区别

移动构造函数传入的参数是一个**右值引用用&&标出。**一般来说左值可以通过使用std:move方法强制转换为右值。

拷贝构造函数是先将传入的参数对象**进行一次深拷贝**，再传给新对象。这就会有一次拷贝对象的开销，并且进行了深拷贝，就需要给对象分配地址空间。

而移动构造函数就是为了解决这个**拷贝开销**而产生的。移动构造函数首先将传递参数的内存地址空间接管，然后**将内部所有指针设置为nullptr**，并且**在原地址上进行新对象的构造**，最后调用原对象的的析构函数，这样做既不会产生额外的拷贝开销，也不会给新对象分配内存空间。

对于指针参数来讲，需要注意的是，移动构造函数是对**传递参数进行一次浅拷贝。**也就是说如果参数为指针变量，进行拷贝之后将会有两个指针指向同一地址空间，这个时候如果前一个指针对象进行了析构，则后一个指针将会变成野指针，从而引发错误。**所以当变量是指针的时候，要将指针置为空**，这样在调用析构函数的时候会进行判断指针是否为空，如果为空则不回收指针的地址空间，这样就不会释放掉前一个指针。

二、操作系统

操作系统基础

1. 操作系统的基本特点

并发：是在计算机系统中**同时存在多个程序**，宏观上看，这些程序是同时向前推进的。 在单CPU上，这些并发执行的程序是交替在CPU上运行的。程序并发性体现在两个方面： 用户程序与用户程序之间的并发执行。 用户程序与操作系统程序之间的并发。

共享：资源共享是操作系统程序和**多个用户程序共用系统中的资源。**

虚拟：是指通过技术**将一个物理实体变成若干个逻辑上的对应物。**在操作系统中虚拟的实现主要是通过分时的使用方法。显然，如果n是某一个物理设备所对应的虚拟逻辑设备数，则虚拟设备的速度必然是物理设备速度的1/n。

异步（不确定性）：同一程序和数据的**多次运行可能得到不同的结果**；程序的运行时间、运行顺序也具有不确定性；外部输入的请求、运行故障发生的时间难以预测。这些都是不确定性的表现。

随机性：是指操作系统的运行是在一个随机的环境中，一个设备可能在任何时间向处理机发出中断请求，系统无法知道运行着的程序会在什么时候做什么事情。

2. 并行与并发的区别

一个人"同时"做多件事，就是并发；多个人"同时"做多件事，就是并行

并发：指在同一时刻只能有一条指令执行，但多个进程指令被快速的**轮换执行**，使得在宏观上具有多个进程同时执行的效果，但在微观上并不是同时执行的，只是把时间分成若干段，使多个进程快速交替的执行。所以并发编程的目标是充分的利用处理器的每一个核，以达到最高的处理性能。

并行：指在同一时刻，有多条指令在多个处理器上**同时执行。**所以无论从微观还是从宏观来看，二者都是一起执行的。

并发与并行的区别：

- 并发，指的是多个事情，在同一时间段内同时发生了；并行，指的是多个事情，在同一时间点上同时发生了。
- 并发的多个任务之间是互相抢占资源的；并行的多个任务之间是不互相抢占资源的。
- 只有在多CPU的情况中，才会发生并行；否则，看似同时发生的事情，其实都是并发执行的。

进程管理

1. 什么是进程

进程是**一个具有独立功能的程序关于某个数据集合的一次运行活动。**它可以申请和拥有系统资源，是一个动态的概念，是一个活动的实体。操作系统会以进程为单位，分配系统资源（CPU时间片、内存等资源），进程是**资源分配的最小单位**，且每个进程拥有独立的地址空间。

2. 什么是线程

线程是进程的一个实体，是进程的一条执行路径；比进程更小的独立运行的基本单位，线程也被称为**轻量级进程**，是操作系统调度（CPU调度）执行的最小单位。一个标准的线程由线程ID，当前指令指针PC，寄存器和堆栈组成。一个进程可以有一个或多个线程，各个线程之间共享程序的内存空间(也就是所在进程的内存空间)。

3. 进程与线程的区别

- 同一进程的线程**共享**本进程的系统资源（CPU时间片、内存等资源），而进程之间则是**独**立的系统资源
- 线程是 **CPU 调度**的最小单位，而进程是操作系统**分配资源**的最小单位
- **多进程**比**多线程健壮**：一个进程崩溃后，在保护模式下不会对其他进程产生影响，但是一个线程崩溃整个进程崩溃
- **多线程效率优于多进程**：进程切换，消耗的资源大，所以涉及到频繁的切换，使用线程要好于进程
- 每个独立的进程有一个程序的入口、程序出口。但是线程不能独立执行，必须依存在应用程序中，由应用程序提供多个线程执行控制。

进程与线程联系的形象理解：

进程和线程的关系类似于公路交通中的多向多车道公路，一个方向的公路视为一个**进程**，一个方向中多个车道线分割开的车道视为多个进程。一个方向的公路由一个或多个车道组成，他们共享道路资源，但不同方向的公路之间是相对独立的。这些不同方向和不同车道都可以**并发**运行，相同方向的不同车道间可以快捷协作同步，而不同方向中的不同车道则需要交通信号灯进行消息通行实现同步执行。

进程与线程所占有的系统资源：

线程 **私有**：线程栈，寄存器，程序寄存器 **共享**：堆，地址空间，全局变量，静态变量 进程 **私有**：地址空间，堆，全局变量，栈，寄存器 **共享**：代码段，公共数据，进程目录，进程ID

4. 线程的特点

- 线程在程序中是独立的、**并发的执行流**，但是进程中的线程之间的隔离程度要小；
- 线程比进程更具有**更高的性能**，这是由于同一个进程中的线程都有共性：多个线程将共享同一个进程虚拟空间；
- 当操作系统创建一个进程时，必须为进程分配独立的内存空间，并分配大量相关资源，而线程创建的**资源消耗要小**很多；

5. 多进程与多线程的区别

- **多进程**比**多线程健壮**：一个进程崩溃后，在保护模式下不会对其他进程产生影响，但是一个线程崩溃整个进程崩溃
- **多线程效率优于多进程**：进程切换，消耗的资源大，所以涉及到频繁的切换，使用线程要好于进程

基于各自特点多进程和多线程有这不同的使用场景:

1. 需要频繁创建销毁的优先用线程；
2. 需要进行大量计算的优先使用线程；
3. 强相关的处理用线程，弱相关的处理用进程；
4. 可能要扩展到多机分布的用进程，多核分布的用线程；

6. 什么是线程同步

线程同步是指多线程通过特定的设置来**控制**线程之间的**执行顺序**，也可以说在线程之间通过同步建立起执行顺序的关系；主要四种方式，**临界区、互斥对象、信号量、事件对象**；其中临界区和互斥对象主要用于**互斥控制**，信号量和事件对象主要用于**同步控制**。

7. 线程的同步方式

互斥控制

- 临界区：通过对多线程的串行化来访问公共资源或一段代码，速度快、适合控制数据访问。在**任意一个时刻只允许一个线程对共享资源进行访问**，如果有多个线程试图访问公共资源，那么在有一个线程进入后，其他试图访问公共资源的线程将被**挂起**，并一直等到进入临界区的线程离开，临界区在被释放后，其他线程才可以抢占。
- 互斥对象：互斥对象和临界区很像，采用互斥对象机制，只有拥有互斥对象的线程才有访问公共资源的权限。因为互斥对象只有一个，所以能保证公共资源不会同时被多个线程同时访问。当前拥有互斥对象的线程处理完任务后必须将线程交出，以便其他线程访问该资源。

同步控制

- 信号量：它允许多个线程在同一时刻访问同一资源，但是需要限制在同一时刻访问此资源的最大线程数目。在用 `CreateSemaphore()` 创建信号量时即要同时指出允许的最大资源计数和当前可用资源计数。一般是将当前可用资源计数设置为最大资源计数，每增加一个线程对共享资源的访问，当前可用资源计数就会减 1 ，只要当前可用资源计数是大于 0 的，就可以发出信号量信号。但是当前可用计数减小到0 时则说明当前占用资源的线程数已经达到了所允许的最大数目，不能在允许其他线程的进入，此时的信号量信号将无法发出。线程在处理完共享资源后，应在离开的同时通过

`ReleaseSemaphore ()`函数将当前可用资源计数加1 。在任何时候当前可用资源计数决不可能大于 最大资源计数。
- 事件对象：通过通知操作的方式来保持线程的同步，还可以方便实现对多个线程的优先级比较的操作。

8. 什么是线程安全

线程安全是多线程编程时的计算机程序代码中的一个概念。在拥有**共享数据**的**多条**线程**并行执行**的程序中，线程安全的代码会通过**同步机制**保证各个线程都可以正常且**正确的执行**，不会出现数据污染等意外情况。没有**数据污染**，即每次程序运行结果和运行的结果是一样的，而且其他的变量的值也和预期的是一样的，这就是线程安全的。

线程不安全问题都是由**全局变量**及静态变量引起的。若每个线程中对全局变量、静态变量只有读操作，而无写操作，一般来说，这个全局变量是线程安全的；若有多个线程同时执行写操作，一般都需要有同步机制，否则的话就可能出现数据污染，从而影响线程安全。

9. 线程安全的实现方式

- **加锁**：利用`Synchronized`或者`ReenTrantLock`来对不安全对象进行加锁，来**实现线程执行的串行化**，从而保证多线程同时操作对象的安全性，一个是语法层面的**互斥锁**，一个是`API`层面的互斥锁。
- **非阻塞同步**。即先进性操作（**先到先得，冲突排队重试**），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，产生冲突，那就再采取其他措施（最常见的措施就是不断地重试，知道成功为止）。这种方法需要硬件的支持，因为我们需要操作和冲突检测这两个步骤具备原子性。通常这种指令包括`CASSC,FAI TAS`等。
- **线程本地化**：一种无同步的方案，利用`Threadlocal`来为每一个线程创造一个**共享变量的副本**，避免几个线程同时操作一个对象时发生线程安全问题。

10. 进程间通信的方式

1. 管道 管道，通常指无名管道。 ① 半双工的，具有固定的读端和写端； ② 只能用于具有亲属关系的进程之间的通信； ③ 可以看成是一种特殊的文件，对于它的读写也可以使用普通的read、write函数。但是它不是普通的文件，并不 属于其他任何文件系统，只能用于内存中。 ④ Int pipe(int fd[2]);当一个管道建立时，会创建两个文件文件描述符，要关闭管道只需将这两个文件描述符关闭即可。
2. FiFO（有名管道） ① FIFO可以再无关的进程之间交换数据，与无名管道不同； ② FIFO有路径名与之相关联，它以一种特殊设备文件形式存在于文件系统中； ③ Int mkfifo(const char* pathname,mode_t mode);
3. 消息队列 ① 消息队列，是消息的连接表，存放在内核中。一个消息队列由一个标识符来标识； ② 消息队列是面向记录的，其中的消息具有特定的格式以及特定的优先级； ③ 消息队列独立于发送与接收进程。进程终止时，消息队列及其内容并不会被删除； ④ 消息队列可以实现消息的随机查询
4. 信号量 ① 信号量是一个计数器，信号量用于实现进程间的互斥与同步，而不是用于存储进程间通信数据； ② 信号量用于进程间同步，若要在进程间传递数据需要结合共享内存； ③ 信号量基于操作系统的PV操作，程序对信号量的操作都是原子操作；
5. 共享内存 ① 共享内存，指两个或多个进程共享一个给定的存储区； ② 共享内存是最快的一种进程通信方式，因为进程是直接对内存进行存取； ③ 因为多个进程可以同时操作，所以需要进行同步； ④ 信号量+共享内存通常结合在一起使用。
6. 套接字（Socket）：更为一般的进程间通信机制，可用于不同机器之间的进程间通信。起初是由Unix系统的BSD分支开发出来的，但现在一般可以移植到其它类Unix系统上：Linux和System V的变种都支持套接字。

11. 进程创建过程

进程调用fork系统调用创建子进程，当控制转移到内核中的fork代码后，内核进行如下操作：

- 系统会分配新的内存块和内核数据结构给子进程
- 将父进程的部分数据结构**写时拷贝**给子进程
- 系统添加子进程到系统进程列表中
- fork 返回，调用器开始调度

当进程调用 fork 完成后，会**有两个返回值**，子进程返回 0；父进程返回子进程的 pid，一般大于0；如果创建失败就返回 -1。通过返回值判断进程是父进程还是子进程，另外也有两个函数 getpid，getppid 分别获取子进程 pid 和父进程 pid。至于父进程和子进程的执行顺序是未知的，这是由调度器决定。

12. 什么是死锁

死锁，是指多个进程在运行过程中因**竞争资源**而造成的一种僵局，当进程处于这种僵持状态时，若无无外力作用，它们都将无法再向前推进。进程各自保持已经获得的资源不释放，并去试图获取另一进程已经持有的资源，陷入相互等待状态。

13. 死锁产生的原因

由于系统中存在一些不可剥夺资源，而当两个或两个以上进程**占有自身资源**，**并请求对方资源**时，会导致每个进程都无法向前推进，这就是死锁。

所以导致死锁的原因有：

- 竞争资源，持有的资源不可剥夺：当系统把这类资源分配给某进程后，再不能强行收回，只能在进程用完后自行释放。
- 进程推荐顺序不当：进程 A 和 进程 B 互相等待对方的数据，（不安全状态）

14. 死锁产生的必要条件

1. 互斥条件：进程要求对所分配的资源进行排它性控制，即在一段时间内某资源仅为一进程所占用用。
2. 请求和保持条件：当进程因请求资源而阻塞时，对已获得的资源保持不放。
3. 不剥夺条件：进程已获得的资源在未使用完之前，不能剥夺，只能在使用完时由自己释放。
4. 循环等待条件：在发生死锁时，必然存在一个进程--资源的环形链。

15. 银行家算法

如果没有死锁发生，并且即使所有进程突然请求对资源的最大需求，也仍然存在某种调度次序能够使得每一个进程运行完毕，则称该状态是安全的。

安全状态的检测与死锁的检测类似，因为安全状态必须要求不能发生死锁。下面的银行家算法与死锁检测算法非常类似。

主存管理

1. 物理地址、逻辑地址

物理地址：它是地址转换的最终地址，进程在运行时执行指令和访问数据最后都要通过物理地址从主存中存取，是**内存单元真正的地址。**

逻辑地址：是指程序员看到的地址。事实上，逻辑地址并不一定是元素存储的真实地址，即数组元素的物理地址，而且并非是连续的，只是**操作系统通过地址映射，将逻辑地址映射成连续的**，这样更符合人们的直观思维。

2. 虚拟内存

虚拟内存：是计算机系统内存管理的一种技术。它使得应用程序认为它拥有连续的可用的内存地址空间），而实际上，它通常是被分隔成多个物理内存碎片，还有部分暂时存储在外部磁盘存储器上，在需要时进行数据交换。

3. 页式管理

4. 页面置换算法

1. **最佳置换算法（OPT）**：在知道页面请调顺序的情况下，将**未来最远**将使用的页淘汰，是一种最优的方案，可以证明缺页数最小。
2. **先进先出置换算法（FIFO）**：先进先出，即淘汰最早调入的页面。
3. **最久未使用置换算法（LRU）**：选择最长时间未被使用的那一页淘汰。
4. **最不经常使用置换算法（LFU）**：将最近使用次数最少的页淘汰。

5. 段式管理

6. 段式管理和页式管理的区别

区别	页式管理	段式管理
用户进程地址空间	一维地址空间	二维地址空间，段名+段内地址
用户可见性	页是信息的物理单位，目的实现离散分配内存，完全是系统行为，对用户是不可见的	段是信息的逻辑单位，目的是更好满足用户需求，对用户是可见的，编程时要给出段名
空间大小	空间大小固定	空间大小可变

分段更容易实现信息的共享和保护，但是如果段长过大，很难分配很大的连续内存空间；分页管理可以提高内存空间的利用率，但是不易实现信息的共享和保护。

访问一个逻辑地址：

- 分页需要两次访存：第一次访存--查内存中的页表；第二次访存--访问目标内存单元。
- 分段需要两次访存：第一次访存--查内存中的段表；第二次访存--访问目标内存单元。但是分段可以通过**快表机制**，将近期访问过的段表项放到快表中，命中就可以减少访问次数，加快地址变换速度。

7. 段页式管理

IO 管理

1. 什么是外中断和内中断

外中断：是指由 CPU 执行指令以外的事件引起，如 I/O 完成中断，表示设备输入/输出处理已经完成，处理器能够发送下一个输入/输出请求。此外还有时钟中断、控制台中断等。

内中断（异常）：是由 CPU 执行指令的内部事件引起，如非法操作码、地址越界、算术溢出等。

程序编译

1. 程序的执行过程

1. **预编译：**主要处理源代码文件中的以“#”开头的预编译指令，展开宏定义，删除注释。
2. **编译：**把预编译生成的文件，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。
3. **汇编：**将汇编代码根据对照表转变成机器可以执行的机器指令(机器码文件)。
4. **链接：**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。

2. 静态链接和动态链接

静态链接：就是在编译链接时**直接将需要的执行代码拷贝到调用处。**优点就是在程序发布的时候就不不需要依赖库，也就是不再需要带着库一块发布，程序可以独立执行，但是体积可能会相对大一些。

动态链接：就是在编译的时候**不直接拷贝**可执行代码，而是通过记录一系列符号和参数，**在程序运行或加载**时将这些信息传递给操作系统，操作系统负责将需要的动态库加载到内存中，然后程序在运行到指定的代码时，去**共享执行内存**中已经加载的动态库可执行代码，最终达到运行时连接的目的。优点是多个程序可以共享同一段代码，而不需要在磁盘上存储多个拷贝，缺点是由于是运行时加载，可能会影响程序的前期执行性能。

Linux

1. Linux 系统中进程管理有哪些命令

- `ps`：显示系统响应命令时的用户的进程信息
- `top`：动态监视系统任务的工具，相当于任务管理器
- `kill`：向某个进程传送一个信号 `kill -signal PID`
- `nice`：改变进程的优先级，使得系统分配资源更加公平

2. top 命令有哪些字段都是什么含义

第一行：概况

`HH:mm:ss`：当前的系统时间。 `up xxx days, HH:mm`：从本次开机到现在经过的时间。 `x user`：当前有几个用户登录到该机器。 `load average`：系统1分钟、5分钟、15分钟内的平均负载值。

第二行：进程计数（Tasks）

`total`：进程总数。 `running`：正在运行的进程数，对应状态TASK_RUNNING。 `sleeping`：睡眠的进程数，对应状态TASK_INTERRUPTIBLE和TASK_UNINTERRUPTIBLE。 `stopped`：停止的进程数，对应状态TASK_STOPPED。 `zombie`：僵尸进程数，对应状态TASK_ZOMBIE。

第三行：CPU使用率（%Cpu(s)）

`us`：进程在用户空间（user）消耗的CPU时间占比，不包含调整过优先级的进程。 `sy`：进程在内核空间（system）消耗的CPU时间占比。 `ni`：调整过用户态优先级的（niced）进程的CPU时间占比。 `id`：空闲的（idle）CPU时间占比。 `wa`：等待（wait）I/O完成的CPU时间占比。 `hi`：处理硬中断（hardware interrupt）的CPU时间占比。 `si`：处理软中断（software interrupt）的CPU时间占比。 `st`：当Linux系统是在虚拟机中运行时，等待CPU资源的时间（steal time）占比。

第四五行：物理内存和交换空间（Mem/Swap）

以物理内存为例。**free命令也会打印**出类似的信息。 `total`：内存总量。 `free`：空闲内存量。 `used`：使用中的内存量。 `buff/cache`：缓存和page cache占用的内存量。

以下所有行：进程详细信息

`PID`：进程ID。 `USER`：进程所有者的用户名。 `PR`：从系统内核角度看的进程调度优先级。 `NI`：进程的nice值，即从用户空间角度看的进程优先级。值越低，优先级越高。 `VIRT`：进程申请使用的虚拟内存量。 `RES`：进程使用的驻留内存（即未被swap out的内存）量。 `SHR`：进程使用的共享内存量。 `S`：进程状态。R=running，S=interruptible sleeping，D=uninterruptible sleeping，T=stopped，Z=zombie。 `%CPU`：进程在一个更新周期内占用的CPU时间比例。 `%MEM`：进程占用的物理内存比例。 `TIME+`：进程创建后至今占用的CPU时间长度。 `COMMAND`：运行进程使用的命令。

3. 什么是僵尸进程

当前进程中生成一个子进程，一般需要调用fork这个系统调用，fork这个函数的特别之处在于一次调用，两次返回，一次返回值大于 0 返回到父进程中，一次返回值为 0 返回到子进程。

如果**子进程先于父进程退出**，同时**父进程又没有调用** `wait/waitpid`，则该**子进程将成为僵尸进程**。通过 ps 命令，我们可以看到该进程的状态为Z (表示僵死)。

4. 如何避免出现僵尸进程

为了防止产生僵尸进程，在fork子进程之后我们都要wait它们

- **通过信号机制来避免僵尸进程**：在父进程 fork() 之前建立一个捕获SIGCHLD信号的**信号处理函数**，并在此 handler 函数中**调用 waitpid() 等待**子进程结束，这样子进程退出的时候向父进程发送 SIGNAL 通知，内核才能获得子进程退出信息从而释放那个进程描述符。
- **两次fork()来避免僵尸进程 (托孤)**：父进程首先调用 fork 创建一个子进程，然后使用 `waitpid()` 等待子子进程退出，子进程再 fork 一个孙进程后退出。这样子进程退出后会被父进程等待回收，而对于孙子进程其父**进程已经退出**所以孙进程成为一个孤儿进程，孤儿进程由 init 进程接管，孙进程结束后，init 会等待回收。

两次 fork 是怎么解决僵尸进程的:

之所以会产生僵尸进程，是因为在进程后终止，除了回收分配到的内存和资源外，还要保留一部分信息供感兴趣者使用，一般是父进程。但是，如果在子进程退出后，父进程没有处理子进程发出的用于处理退出信息的SIGCHLD信号，则会导致子进程一直处于EXIT_ZOMBIE状态而成为僵尸进程。

```
1 进程的**僵尸状态生命周期**为：进程终止到父进程调用waitpid处理SIGCHLD信号。
```

采用fork两次的方式，由爷爷产生父亲，父亲产生孙子，并且父亲进程在产生孙子以后就退出，这会导致孙子进程被祖宗进程init收养，而init进程是一定会处理进城的退出信息的，这就避免了孙子进程变为僵尸进程。

5. 什么是孤儿进程

如果父进程先退出，子进程还没退出，那么子进程的父进程将变为 init 进程。（注：任何一个进程都必须有父进程）。

一个父进程退出，而它的一个或多个子进程还在运行，那么那些子进程将成为孤儿进程。孤儿进程将被 init 进程 (进程号为1) 所收养，并由 init 进程对它们完成状态收集工作。

6. 什么是守护进程

指在后台运行的，没有控制终端与之相连的进程。它独立于控制终端，周期性地执行某种任务。Linux的大多数服务器就是用守护进程的方式实现的，如 web 服务器器进程http等。

7. fork() 系统调用的返回值，父进程和子进程有什么差异

当进程调用 fork 完成后 ，会**有两个返回值**，子进程返回 0；父进程返回子进程的 pid，一般大于0；如果创建失败就返回 -1。通过返回值判断进程是父进程还是子进程，另外也有两个函数 getpid，getppid 分别获取子进程 pid 和父进程 pid。至于父进程和子进程的执行顺序是未知的，这是由调度器决定。

8. Linux 系统中网络管理有哪些命令

9. Linux 系统中磁盘管理有哪些命令

三、计算机网络

计算机网络概述

1. OSI 七层协议模型

- **物理层**：简单的说，物理层（Physical Layer）确保原始的数据可在各种**物理媒体上传输。**在这一层上面规定了激活，维持，关闭通信端点之间的机械性，电气特性，功能特性，，为上层协议提供了一个传输数据的物理媒体，这一层传输的是 **bit 流。**IEEE 802.2
- **数据链路层**：数据链路层（Data Link Layer）在不可靠的物理介质上提供**可靠的传输。**该层的作用包括：物理地址寻址、数据的成帧、流量控制、数据的检错、重发等。这一层中将 bit 流封装成 **frame 帧。**ARP, MAC
- **网络层**：网络层（Network Layer）负责对子网间的数据包进行路由选择。此外，网络层还可以实现拥塞控制、网际互连等功能。在这一层，数据的单位称为**数据包**（packet）。IP
- **传输层**：传输层是第一个端到端，即主机到主机的层次。传输层负责**将上层数据分段并提供端到端的、可靠的或不可靠的传输。**此外，传输层还要处理端到端的差错控制和流量控制问题。在这一层，数据的单位称为**数据段**（segment）。TCP, UDP
- **会话层**：这一层管理主机之间的会话进程，即负责建立、管理、终止进程之间的会话。会话层还利用在数据中插入校验点来实现数据的同步，访问验证和会话管理在内的建立和维护应用之间通信的机制。如**服务器验证用户登录便是由会话层完成的。**使通信会话在通信失效时从校验点继续恢复通信。比如说**建立会话**，如 session 认证、断点续传。DNS
- **表示层**：这一层主要解决用户信息的语法表示问题。它将欲交换的数据从适合于某一用户的抽象语法，转换为适合于 OSI 系统内部使用的传送语法。即**提供格式化的表示和转换数据服务。**数据的压缩和解压缩， 加密和解密等工作都由表示层负责。比如说图像、视频编码解，数据加密。 Telnet
- **应用层**：这一层为操作系统或网络应用程序**提供访问网络服务的接口。**HTTP

运输层

1. UDP 与 TCP 的区别

用户数据报协议 UDP（User Datagram Protocol）

- 是**无连接**的，
- 尽最大可能交付，即**不可靠交付**
- **没有拥塞控制**，源主机以恒定的速率发送数据
- **面向报文**（对于应用程序传下来的报文不合并也不拆分，只是添加 UDP 首部），
- 支持一对一、一对多、多对一和**多对多**的交互通信
- 首部开销小

传输控制协议 TCP（Transmission Control Protocol）

- 是**面向连接**的，
- 提供**可靠交付**，
- **有**流量控制，拥塞控制，
- **面向字节流**（把应用层传下来的报文看成字节流，把字节流组织成大小不等的数据块），
- 每一条 TCP 连接只能是点对点的（**一对一**），
- 提供**全双工通信**，TCP 连接的两端在任何时候都可以发送和接收数据

2. TCP 报文段的首部格式

- **源端口**和**目的端口**（16bit+16bit）：源端口号和目的端口号
- **序号**（32bit）：传输方向上字节流的字节编号。初始时序号会被设置一个随机的初始值（ISN），之后每次发送数据时，序号值 = ISN + 数据在整个字节流中的偏移。假设A -> B且ISN = 1024，第一段数据512字节已经到B，则第二段数据发送时序号为1024 + 512。TCP 是面向字节流的，在一个 TCP 连接中传送的字节流中的**每一个字节都按顺序编号**，用**于解决网络包乱序问题。**
- **确认号**（32bit）：接收方对发送方 TCP 报文段的响应，其值是收到的序号值 + 1。确认号 = N，表示到序号 N-1 为止都所有数据都已经正确收到，期望收到发送方的下一个报文断的第一个数据字节的序号为 N。
- **数据偏移**（4bit）：报文段数据起始处距离整个报文段起始处的距离即**首部长度**，标识首部有多少个4字节 * 首部长，最大为15，即60字节。
- **标志位**（6bit）：
  - 紧急 URG（urgent）：标志紧急指针是否有效。
  - 确认 ACK（acknowledgment）：标志确认号是否有效（确认报文段）。用于**解决丢包**问题。
  - 推送 PSH（push）：提示接收端立即从**缓冲读走**数据。
  - 复位 RST（reset）：表示要求对方**重新建立**连接（复位报文段）。
  - 同步 SYN（synchronization）：表示请求**建立**一个连接（连接报文段）。
  - 终止 FIN（finish）：表示**关闭连接**（断开报文段）。
- **窗口**（16bit）：接收窗口。用于告知对方（发送方）本方的**缓冲**还能接收多少字节数据。用于**流量控制。**
- **校验和**（16bit）：接收端用 CRC 检验整个报文段有无**损坏。**

3. TCP 三次握手

1. 客户端 SYN-SENT：客户端发含 `SYN = 1, seq = x` 的建立连接包到服务器
2. 服务器 SYN-RCVD：服务器发含 `SYN = 1, ACK = 1, seq = y, ack = x+1` 的建立连接包到客户端

3. 客户端 ESTABLISHED：客户端发含 `ACK = 1，seq = x+1，ack = y+1` 的确认包到服务器，服务器 ESTABLISHED

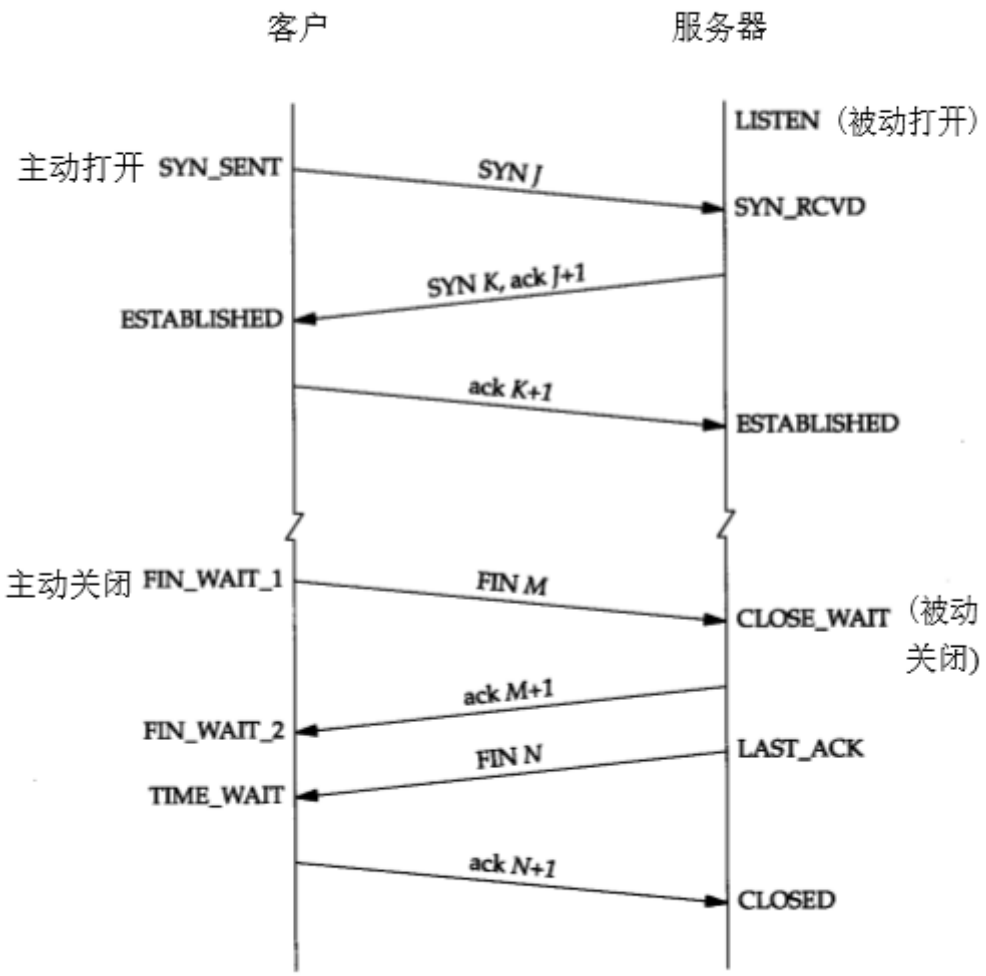
4. TCP 四次挥手

1. 客户端 FIN-WAIT-1：客户端发送 `FIN = 1，seq = u` 的关闭连接包到服务器
2. 服务器 CLOSE-WAIT：服务器发送 `ACK = 1，seq = v，ack = u+1` 的确认包到客户端。这两步完成了客户端的连接关闭，客户端进入 FIN-WAIT-2 状态。此时整个连接处于**半关闭状态**，服务器还未释放连接，若服务器这段时间内向客户端发送数据，客户端仍要接收。
3. 服务器 CLOSE-WAIT：服务器发送 `FIN = 1，ACK = 1，seq = w，ack = u+1` 的关闭连接包到客户端。
4. 客户端 FIN-WAIT-2：客户端发送 `ACK = 1，seq = u+1，ack = w+1` 的确认包到服务器。3、4 两步完成了服务器的连接关闭，在此之后客户端进入持续 2MSL 的时间等待状态 TIME-WAIT。（`MSL（Maximum Segment Lifetime）`，**报文最大生存时间**，它是任何报文在网络上存在的最长时间，超过这个时间报文将被丢弃。）

5. 为什么客户端在 TIME-WAIT 状态等待 2MSL？

- **保证连接正确关闭**：等待足够的时间以确保最后的 ACK 能让服务器接收。因为客户端发送的确认包可能丢失，服务端在 LAST-ACK 状态收不到确认包时会进行超时重传关闭连接包给客户端，这时就能保证客户端在 2MSL 的等待时间内重传一次确认包，保证服务器正确关闭连接。如果客户端仅发完确认包后不仅入时间等待状态，而是立即释放连接，则无法收到服务器重传的关闭连接包，导致其无法正确关闭连接。
- **防止已失效的连接请求报文段出现在本连接中**：`2MSL` 的时间等待足以让两个方向上的数据包都被丢弃，使得原来连接的数据包在网络中都自然消失，再出现的数据包一定都是新建立连接所产生的。如果没有时间等待状态，将使得具备相同序号的旧连接被接收，导致数据错乱。

6. TCP 的连接状态



- CLOSED：初始状态。
- LISTEN：服务器处于监听状态。
- SYN_SEND：客户端 socket 执行CONNECT连接，发送SYN包，进入此状态。
- SYN_RECV：服务端收到 SYN 包并发送服务端 SYN 包，进入此状态。
- ESTABLISH：表示连接建立。客户端发送了最后一个ACK包后进入此状态，服务端接收到ACK包后进入此状态。
- FIN_WAIT_1：终止连接的一方（通常是客户机）发送了FIN报文后进入。等待对方FIN。
- CLOSE_WAIT：（假设服务器）接收到客户机FIN包之后等待关闭的阶段。在接收到对方的FIN包之后，自然是需要立即回复ACK包的，表示已经知道断开请求。但是本方是否立即断开连接（发送FIN包）取决于是否还有数据需要发送给客户端，若有，则在发送 FIN 包之前均为此状态。
- FIN_WAIT_2：此时是半连接状态，即有一方要求关闭连接，等待另一方关闭。客户端接收到服务器的ACK包，但并没有立即接收到服务端的FIN包，进入FIN_WAIT_2状态。
- LAST_ACK：服务端发动最后的FIN包，等待最后的客户端ACK响应，进入此状态。
- TIME_WAIT：客户端收到服务端的FIN包，并立即发出ACK包做最后的确认，在此之后的2MSL时间称为TIME_WAIT状态。

7. TCP 四次挥手中的等待状态

- FIN_WAIT_1：客户端发送了 FIN 报文后进入 FIN_WAIT_1 状态，并等待服务器确认。
- FIN_WAIT_2：
  - 此时是**半关闭状态**，即有一方要求关闭连接，等待另一方关闭。客户端接收到服务器的 ACK 包，但并没有立即接收到服务端的FIN包，进入 FIN_WAIT_2 状态。
  - 该状态中**服务器还未释放连接**，若服务器这段时间内向客户端发送数据，客户端仍要接收，但是已经没有发送数据能力。

- CLOSE_WAIT：发出被动关闭连接确认报文段之后，发出主动关闭连接报文段之前处于的状态。
    - 被动关闭连接一方接收到 FIN 包会立即回应 ACK 包表示已接收到断开请求。
    - 被动关闭连接一方如果还有**剩余数据要发送**就会进入 CLOSED_WAIT 状态。
- TIME_WAIT：客户端收到服务端的FIN包，并立即发出ACK包做最后的确认，在此之后的2MSL时间称为TIME_WAIT状态。
    - 如果客户端直接进入CLOSED状态，如果服务端没有接收到最后一次ACK包会在超时之后重新再发 FIN 包，此时因为客户端已经CLOSED，所以服务端就不会收到ACK而是收到RST。所 TIME_WAIT 状态目的是防止最后一次握手数据没有到达对方而触发重传FIN准备的。
    - 在 2MSL 时间内，同一个socket不能再被使用，否则有可能会和旧连接数据混淆（如果新连接和旧连接的socket相同的话）。

8. 为什么握手是三次，挥手是四次

- 对于握手：握手只需要确认双方通信时的初始化序号，保证通信不会乱序。（**第三次握手**必要性：假设服务端的确认丢失，连接并未断开，客户机超时重发连接请求，这样服务器会对同一个客户机保持多个连接，造成资源浪费。）
- 对于挥手：TCP 是**全双工通信**的，所以发送方和接收方都需要 FIN 和 ACK。只不过有一方是被动的，所以看上去就成了4次挥手。

9. 什么是超时重传、RTT 和 RTO

- **超时重传：**发送端发送报文后若长**时间未收到确认**的报文则需要重发该报文。可能有以下几种情况：
    - 发送的数据没能到达接收端，所以对方没有响应
    - 接收端接收到数据，但是ACK报文在返回过程中丢失
    - 接收端拒绝或丢弃数据
- **报文段的往返时间 RTT：** 数据从发送到接收到对方响应之间的时间间隔，即数据报在网络中**一个往返用时**，其大小不稳定。
- **超时重传时间 RTO：**从上一次发送数据，因为长期没有收到 ACK 响应，到下一次重发之间的时间，就是**重传间隔。**
    - 通常每次重传RTO是前一次重传间隔的两倍，计量单位通常是 RTT，例：1RTT，2RTT，4RTT， 8RTT……
    - 重传次数到达上限之后停止重传

10. TCP 滑动窗口

- TCP是双工协议，双方可以同时通信，所以发送方接收方各自维护一个**发送窗和接收窗。**
    - 发送窗：用来限制发送方可以发送的数据大小，其中发送窗口的大小由接收端返回的TCP报文段中窗口字段来控制，接收方通过此字段告知发送方自己的缓冲（受系统、硬件等限制）大小。
    - 接收窗：用来标记可以接收的数据大小。
- TCP 是流数据
    - 发送出去的数据流可以被分为以下四部分：已发送且被确认部分 | **已发送未被确认部分 | 未发送但可发送部分** | 不可发送部分，其中发送窗 = 已发送未确认部分 + 未发但可发送部分。
    - 接收到的数据流可分为：已接收 | **未接收但准备接收** | 未接收不准备接收。接收窗 = 未接收但准备接收部分。
- 发送窗内数据只有当接收到接收端某段发送数据的 ACK 响应时才移动发送窗，**左边缘**紧贴刚被确认的数据。接收窗也只有接收到数据且**最左侧**连续时才移动接收窗口。

TCP 基于滑动窗口的重发机制

- 滑动窗口机制，确立收发的边界，能让发送方知道已经发送了多少（已确认）、尚未确认的字节数、尚待发送的字节数；让接收方知道（已经确认收到的字节数）。
- 选择重传，用于对传输出错的序列进行重传。

11. 什么是流量控制

流量控制目的是接收方通过 TCP 头窗口字段告知发送方本方可接收的最大数据量，用以**解决发送速率过快导致接收方不能接收的问题**，所以流量控制是**点对点控制。**

12. 什么是拥塞控制

拥塞控制目的是防止数据被过多注网络中导致网络资源（路由器、交换机等）过载。因为拥塞控制涉及网络链路全局，所以属于**全局控制**，控制拥塞使用拥塞窗口。

13. TCP 的拥塞控制算法

TCP 主要通过四个算法来进行拥塞控制：

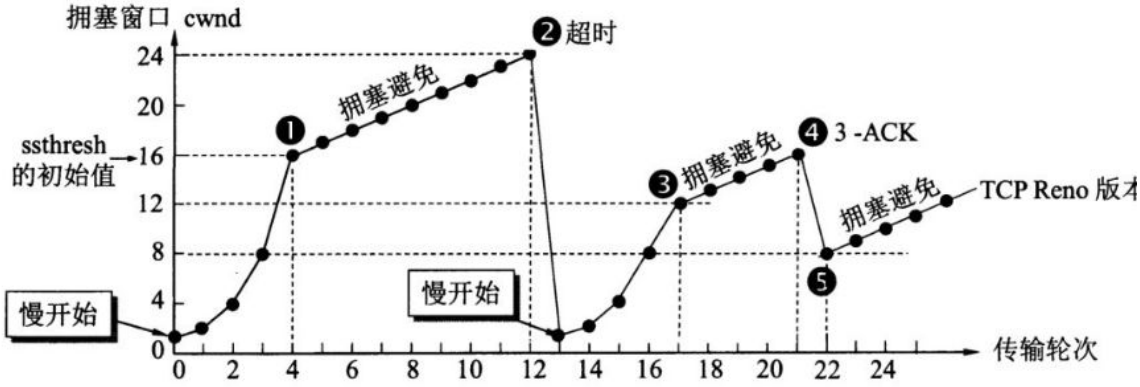
慢开始：cwnd = 1, 每个轮次 cwnd 加倍增长

拥塞避免：cwnd >= ssthresh, cwnd 线性增长，每个轮次只将 cwnd 加 1

快重传：三个重复确认立即重传丢失的报文段

快恢复：ssthresh = cwnd / 2, cwnd = ssthresh

发送方需要维护一个叫做拥塞窗口（cwnd）的状态变量，注意拥塞窗口与发送方窗口的区别：拥塞窗口只是一个状态变量，实际决定发送方能发送多少数据的是发送方窗口。



TCP 拥塞窗口 cwnd 在拥塞控制时的变化情况

慢开始与拥塞避免

发送的最初执行**慢开始，令 cwnd = 1**，发送方只能发送 1 个报文段；当收到确认后，将 cwnd 加倍，因此之后发送方能够发送的报文段数量为：2、4、8 ...

注意到**慢开始每个轮次都将 cwnd 加倍**，这样会让 cwnd 增长速度非常快，从而使得发送方发送的速度增长速度过快，网络拥塞的可能性也就更高。

设置一个**慢开始门限 ssthresh**，当 cwnd >= ssthresh 时，进入**拥塞避免，每个轮次只将 cwnd 加 1。**

如果出现了**超时**，则令 ssthresh = cwnd / 2，然后重新执行慢开始。

快重传与快恢复

在接收方，要求每次接收到报文段都应该对最后一个已收到的有序报文段进行确认。例如已经接收到 M1 和 M2，此时收到 M4，应当发送对 M2 的确认。

在发送方，如果收到**三个重复确认**，那么可以知道下一个报文段丢失，此时执行快重传，立即重传下一个报文段。例如收到三个 M2，则 M3 丢失，立即重传 M3。

在这种情况下，只是丢失个别报文段，而不是网络拥塞。因此执行快恢复，令 ssthresh = cwnd / 2 ，cwnd = ssthresh，注意到此时直接进入拥塞避免。

慢开始和快恢复的快慢指的是 cwnd 的设定值，而不是 cwnd 的增长速率。慢开始 cwnd 设定为 1，而**快恢复 cwnd 设定为 ssthresh。**

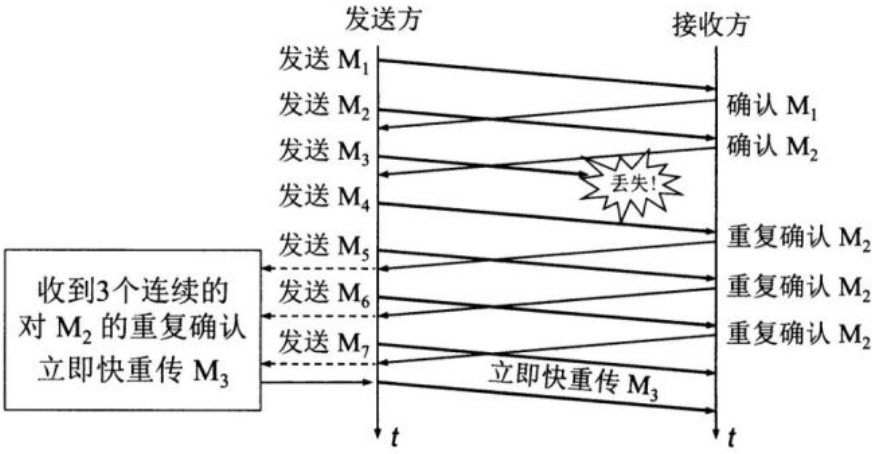


图 5-26　快重传的示意图

超时与3ACK

- 超时：判断为出现网络拥塞，令 **ssthresh = cwnd / 2**，然后重新执行慢开始 **cwnd = 1**
- 3ACK：三个重复确认避免判断为超时情况，发送方也就不会误认为出现了网络拥塞，这时执行快重传立即重传丢失的报文段，接着执行快恢复令 **ssthresh = cwnd / 2 ， cwnd = ssthresh**，避免直接判断为拥塞情况重新执行慢开始降低传输效率。

慢开始门限用法

- cwnd < ssthresh : 执行慢开始
- cwnd > ssthresh : 停止慢开始 指向拥塞避免
- cwnd = ssthresh : 可以使用慢开始，也可以使用拥塞避免

14. TCP 如何实现提供可靠数据传输

- 建立连接（标志位）： 通信前确认通信实体存在。
- 序号机制（序号、确认号）： 确保了数据是按序、完整到达。
- 数据校验（校验和）： CRC 校验全部数据。
- 超时重传（定时器）： 保证因链路故障未能到达数据能够被多次重发。
- 窗口机制（窗口）： 提供流量控制，避免过量发送。
- 拥塞控制：提供拥塞控制，避免网络拥塞。

15. TCP 丢包与粘包

如果是 TCP 协议，在大多数场景下，是不存在丢包和包乱序问题的，TCP 通信是可靠通信方式，TCP 协议栈通过序列号和包重传确认机制保证数据包的有序和一定被正确发到目的地；如果是 UDP 协议，如果不能接受少量丢包，那就要自己在 UDP 的基础上实现类似 TCP 这种有序和可靠传输机制了（例如 RTP 实时传输协议（序号）、RUDP 实时控制协议（流量控制，拥塞控制））。

粘包的产生：

- 发送方产生粘包：采用TCP协议传输数据的客户端与服务器经常是保持一个**长连接的状态**（一次连接发一次数据不存在粘包），双方在连接不断开的情况下，可以一直传输数据；但当发送的**数据包过小**时，那么TCP协议默认的会**启用Nagle算法**，将这些较小的数据包进行**合并发送**（缓冲区数据

发送是一个堆压的过程）；这个合并过程就是在发送缓冲区中进行的，也就是说数据发送出来它已经是**粘包的状态**了；
- 接收方产生粘包：接收方采用TCP协议接收数据时，数据从网络模型底层开始逐层向上传递，传到传输层后TCP将数据**放置在接收缓冲区**，然后由**应用层主动获取。**如果应用层**读**缓冲区数据的速度**小于**传输层**写入**数据的速度，就会出现粘包情况，第一个数据包每读完第二个数据包就已经到达，并有一部分放入了缓冲区末尾。

16. 怎么解决粘包问题

解决发送方粘包：

- **发送方产生粘包是因为Nagle算法合并小数据包，那么可以禁用掉该算法**
- TCP提供了强制数据立即传送的操作指令push，当填入数据后调用操作指令就可以立即将数据发送，而不必等待发送缓冲区填充自动发送
- 数据包中加头，头部信息为整个数据的长度（最广泛最常用）

解决接收方粘包：

- **解析数据包头部信息，根据长度来接收。**在一条TCP连接上，数据的流式传输在接收缓冲区里是有序的，其主要的问题就是第一个包的包尾与第二个包的包头共存接收缓冲区，所以根据长度读取是十分合适的。
- 自定义数据格式：在**数据中放入开始、结束标识**；解析时根据格式抓取数据，缺点是数据内不能含有开始或结束标识
- 短连接传输，建立一次连接只传输一次数据就关闭；（不推荐）

会话层

1. DNS 的解析过程

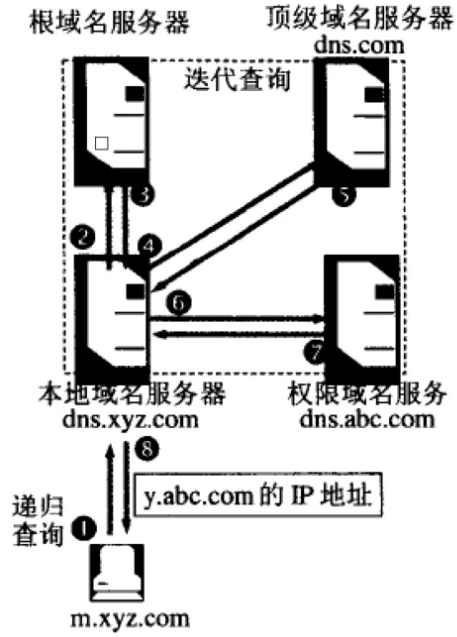
递归查询：

主机向**本地域名服务器**的查询一般都是采用**递归查询。**所谓递归查询就是：如果主机所询问的本地域名服务器不知道被查询的域名的 IP 地址，那么本地域名服务器就以 DNS 客户的身份，向根域名服务器器继续发出查询请求报文，即**代替主机继续查询**，而不是让主机自己进行下一步查询。因此，递归查询返回的查询结果或者找到或者没找到，即查询主机的IP或报错信息。

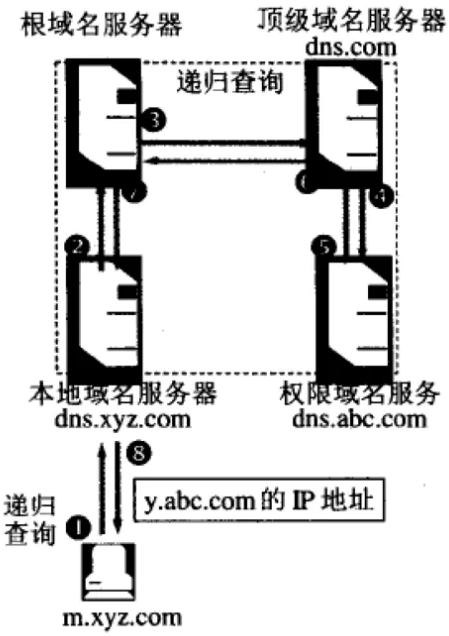
迭代查询：

本地域名服务器向**根域名服务器**的查询一般采用**迭代查询。**迭代查询的特点：当根域名服务器收到本地域名服务器发出的迭代查询请求报文时，要么给出所要查询的 IP 地址，要么告知下一步应当查询的域名服务器 IP 地址，然后让**本地服务器自行进行后续的查询。**

根域名服务器通常是把自己知道的顶级域名服务器的 IP 地址告诉本地域名服务器，让本地域名服务器再向顶级域名服务器查询。顶级域名服务器在收到本地域名服务器的查询请求后，要么给出所要查询的 IP 地址，要么告知下一步应当查询的权限域名服务器 IP 地址。最后，本地域名服务器得到了所要解析的 IP 地址或报错，然后把这个结果返回给发起查询的主机。



(a)



(b)

应用层

1. 统一资源定位符 URL 是什么

URL 是用来个表示从互联网上得到的资源位置和访问这些资源的方法，**URL 给资源的位置提供了一种抽象的识别方法。**

URL 一般形式由四个部分组成：`<协议>://<主机>:<端口>/<路径>`，携带参数的情况 `<协议>://<主机>:<端口>/<路径>?<参数>`

2. 浏览器中输入网址之后到显示页面的过程

1. 输入 URL，如果浏览器有缓存且未过期，则直接使用缓存渲染页面
2. 浏览器解析 URL，获取主机域名等信息
3. DNS 解析，根据主机域名获取主机 IP 地址
4. 根据 IP 地址建立 TCP 连接
5. 建立连接之后，向服务端发送 HTTP 请求
6. 服务端处理 HTTP 请求，处理之后将请求的资源作为响应体，通过 HTTP 响应返回给浏览器
7. 浏览器对 HTTP 响应内容进行解析，根据状态码进行对应处理，根据响应头字段判断是否启用缓存（cache-control 可缓存，no-store 不可缓存）
8. 并将请求到的资源渲染出页面
9. 连接结束

3. 常用的 HTTP 方法

GET：用于请求访问已经被URI（统一资源标识符）识别的资源，可以通过URL传参给服务器 **POST**：用于传输信息给服务器，主要功能与GET方法类似，但一般推荐使用POST方式。 **PUT**： 传输文件，报文主体中包含文件内容，保存到对应URI位置。 **HEAD**： 获得报文首部，与GET方法类似，只是不返回报文主体，一般用于验证URI是否有效。 **DELETE**：删除文件，与PUT方法相反，删除对应URI位置的文件。 **OPTIONS**：查询相应URI支持的HTTP方法。

4. GET 方法和 POST 方法的区别

- 使用场景：GET 用于从指定的资源**请求**数据，而 POST 用于向指定的资源**提交**要被处理的数据。
- 传递参数：
  - GET 方法的参数是以查询字符串的形式放在 URL 后，而 POST 则是将参数存储在请求实体中。
  - 另外 URL 仅支持 ASCII 码所以如果参数中出现中文等其他字符需要进行编码，否在会出现乱码情况；而 POST 支持标准字符集，可以正确的传递中文等字符。
  - GET 传递的参数放在 URL 中是完全暴露的，存在数据安全的问题；而 POST 方法的参数则存在在请求实体中。
- 安全性：GET 方法是安全的因为它仅请求资源，不会改变服务器状态；POST 方法则需要提交请求实体，其中的内容可能会改变服务端数据内容，从而改变了服务器状态态，所以是不安全的。
- 缓存：GET 请求类似于查找的过程，用户获取数据，可以不用每次都与数据库连接，所以**可以使用缓存。**POST 做的一般是修改和删除的工作，所以必须与数据库交互，所以**不能使用缓存。**
- 幂等性

5. HTTP 请求报文和响应报文的格式

请求报文：请求行+请求头部+空行+请求主体

```
POST /user HTTP/1.1                                  // 请求行
User-Agent：产生请求的浏览器类型(火狐还是谷歌浏览器)
Host：请求的主机名，即请求的网址
connection：连接方式(close 或 keepalive)
Cookie：存储于客户端扩展字段，向同一域名的服务端发送属于该域的cookie
Content-Type：代表发送端（客户端|服务器）发送的实体数据的数据类型
Accept：代表发送端（客户端）希望接受的数据类型     // 以上是请求头
（此处必须有一空行 |                                 // 空行分割header和请求内容
name=world                                           // 请求体(可选，如get请求时可选)
```

响应报文：状态行+响应头部+空行+响应主体

```
HTTP/1.1 200 OK                                      // 状态行
Date标头：响应产生的时间
Age标头:（从最初创建开始）响应持续时间
Server标头: 向客户端标明服务器程序名称和版本
Content-Length标头：响应实体的长度
Content-Tyep标头：响应实体的类型                    // 以上是请响应头
（此处必须有一空行 |                                 // 空行分割header和请求内容
name=world                                           // 响应体
```

6. HTTP 常用的状态码

状态码分类

1xx：表示目前是协议的中间状态，**正在处理**还需要后续请求

2xx：表示**请求成功**

3xx：表示**重定向**状态，需要重新请求

4xx：表示**请求报文错误**

5xx：表示**服务端错误**

常用状态码

101 切换请求协议，从 HTTP 切换到 WebSocket

200 请求成功，有响应体

301 **永久**重定向：会缓存

302 **临时**重定向：不会缓存

304 协商缓存命中

403 服务器**禁止访问**

404 资源**未找到**

400 **请求错误**

500 服务端错误

503 服务器**繁忙**

7. HTTP 301 和 302 状态码的区别

重定向：多个域名跳转至同一域名

301 重定向是页面**永久性转移**，搜索引擎在抓取新内容的同时也将旧的网址替换成重定向之后的网址

302 重定向是页面**暂时性转移**，搜索引擎会抓取新的内容而保存旧的网址并认为新的网址只是暂时的。

8. HTTP 500 状态码的具体场景

- 运行的**用户数过多**，对服务器造成的压力过大，服务器无法响应
- 操作涉及数据库，大数据量的情况下导致数据库中表空间已满，或者数据库连接池较小无法满足数据的存取等

9. HTTP 的优化方案

1. TCP 复用：一个客户端的多个HTTP请求通过一个TCP连接进行处理（HTTP 1.1 长连接）
2. 压缩：将文本数据进行压缩，减少带宽（HTTP 2.0 压缩优化）
3. 内容缓存：将经常用到的内容进行缓存起来，客户端可以直接在内存中获取相应的数据（Session 和 Cookie）
4. 安全性：SSL 加速（SSL Acceleration），使用 SSL 协议对 HTTP 协议进行加密，在通道内加密并加速（HTTPS）

10. HTTP 1.0, 1.1, 2.0 的区别

HTTP 1.0：浏览器与服务器只保持短暂的连接，连接**无法复用。**也就是说每个TCP连接只能发送一个请求。发送数据完毕，连接就关闭，如果还要请求其他资源，就必须再新建一个连接。这种方式就好像打电话的时候，只能说一件事，说完之后就要挂断；想要说另外一件事的时候就要重新拨打电话。

HTTP 1.1：相对于 1.0 最主要的改进就是引入了**持久连接。**所谓的持久连接即TCP连接默认不关闭，可以被多个请求复用，提升了 HTTP 的效率。

HTTP 2.0： 2.0 采用了**多路复用。**即在一个连接里，客户端和浏览器都可以同时发送多个请求或回应，而且不用按照顺序一一对应。能实现多路复用的基础是**二进制分帧**，将请求进一步**拆分**并进行二进制编码，最终响应结果由多个子响应**组装**而成。除此之外，2.0 还进行了压缩优化，将文本数据进行压缩，减少带宽。

HTTP2.0的多路复用和HTTP1.X中的长连接复用有什么区别?

https://zhuanlan.zhihu.com/p/348307320

- HTTP/1.* 一次请求-响应，建立一个连接，用完关闭；每一个请求都要建立一个连接
- HTTP/1.1 Pipeling解决方式为，若干个请求排队串行化单线程处理，后面的请求等待前面请求的返回才能获得执行机会，因为传输格式是文本的，一旦有某请求超时等，后续请求只能被阻塞，毫无办法，也就是人们常说的**线头阻塞**
- HTTP/2多个请求可同时在一个连接上**并行执行**（由于支持二进制的格式，可以无序）某个请求任务耗时严重，不会影响到其它连接的正常执行

11. Cookie 和 Session 的区别

由于 HTTP 协议是**无状态协议**，所以服务端需要记录用户的状态时，就需要用某种机制来识具体的用户，这种机制就是服务端为特定的用户创建特定的 Session 用于标识和跟踪特定用户。

Session 是在**服务端保存**的一个数据结构，用来跟踪用户的状态，这个数据可以保存在集群、数据库和文件中。Cookie 是**客户端保存**用户信息的一种机制，用来记录用户的一些信息，也是实现 Session 的一种方式。

Cookie 要参与服务端通信，客户端每次向服务端发送 HTTP 请求是 cookie 记录的用户信息都被存储于**请求头部**中；所以 Cookie 的存储的数据大小不能超过 4K，否则将会带来性能问题。而 Session 不参与服务端通信，有着更大的数据存储大小，一般不超过 5M。

Session 和 Cookie 的有效期也不同，Session 在当前会话下有效，关闭页面或者浏览器时会被清空；Cookie 在设置的有效期内有效，当超过有效期便会自动失效。

12. HTTP 与 HTTPS 的区别

HTTP 的中文叫做超文本传输协议,它负责完成客户端到服务端的一系列操作，是专门用来传输注入HTML的超媒体文档等 web 内容的协议，它是**基于传输层的 TCP** 协议的应用层协议。

HTTPS (Hyper Text Transfer Protocol over SecureSocket Layer) 是基于安全套接字的 HTTP 协议，在 HTTP 的基础上通过传输加密和身份认证保证了传输过程的安全性，也可以理解为是 HTTP + SSL / TLS (数字证书) 的组合。

http和https的区别:

- URL 标识：HTTP 的 URL 以 http:// 开头，而 HTTPS 的 URL 以 https:// 开头
- 开销：HTTP 无需认证证书，而 HTTPS 需要认证证书
- 安全性：HTTP 连接较为简单，是无状态的；而 HTTPS 通过传输加密和身份认证保证了传输过程的安全性
- 端口：HTTP 标准端口是 80 ，而 HTTPS 的标准端口是 443
- 资源消耗：HTTP 是**明文传输**，HTTPS 使用了 SSL 加密传输协议，需要消耗更多的系统资源

小结：简单来说 http 是用来进行 html 等超媒体传输的，但是http不安全，为了安全使用证书 SSL 和 HTTP 的方式进行数据传输，也就是 HTTPS。

13. HTTPS 的工作过程

1. **客户端访问 HTTPS 连接**：客户端发送自己支持的加密规则给服务器，告知服务器 HTTPS 连接开始；
2. **服务端发送证书（公钥）给客户端**：服务器先将收到的加密规则和自己的比对，如果不符合直接拒绝连接。若符合则把符合的加密规则和证书发送给客户端，证书中包含证书的颁发机构、加密公钥等信息；
3. **客户端验证收到的证书**：
    1. **验证证书**的合法性；
    2. **密钥加密**：通过证书验证后，客户端会生成一串随机数即**密钥**，并用证书中的**公钥进行加密**；

3. **握手信息加密**：用约定好的 hash 算法生成握手消息，并用生成的密钥进行加密，最后将哈希值和握手信息一起发送给服务端。

4. **服务端接收加密握手信息**：
    1. **验证加密信息**：服务端用**私钥解析出密钥**，并用密钥解析握手消息，验证哈希值是否和客户端发来的一致；
    2. **握手信息加密**：完成信息验证之后，服务端也使用同样的方式加密握手消息，并和哈希值一起发给客户端。
5. **客户端接收加密握手信息**：客户端用原先生成的密码解密握手信息，如果计算得到的哈希值一致，则握手成功。

14. 什么是对称加密和非对称加密

- 对称加密算法在加密和解密时使用的是同一个秘钥；
- 非对称加密算法需要两个密钥来进行加密和解密，这两个密钥是公开密钥（public key，简称公钥）和私有密钥（private key，简称私钥）。RSA 非对称加密算法

15. 什么是数字证书

对称加密中，双方都使用公钥进行解密。虽然数字签名可以保证数据不被替换，但是数据是由公钥加密的，如果**公钥也被替换**，则仍然可以伪造数据，因为用户不知道对方提供的公钥其实是假的。所以为了保证发送方的公钥是真的，CA 证书机构会负责颁发一个证书，**里面的公钥保证是真的**，用户请求服务器时，服务器将证书发给用户，这个证书是经由系统内置证书的备案的。

四、数据库系统

01 数据库系统基础概念

1. 关系型数据库与非关系型数据库的区别

2. 什么是数据库索引

索引是为了提高数据的查询效率，就像书的目录一样。减少磁盘 IO 次数

同样索引也会带来很多负面影响：

- 创建索引和维护索引需要耗费时间，这个时间随着数据量的增加而增加；
- 索引需要占用物理空间，不光是表需要占用数据空间，每个索引也需要占用物理理空间；当对表进行增、删、改、的时候索引也要动态维护，这样就降低了数据的维护速度。

3. 可以用于实现索引的常用数据结构有哪些

索引的数据结构和具体存储引擎的实现有关，，在 MySQL 中使用较多的索引有 Hash 索引、B+树索引等。经常使用的 InnoDB 存储引擎的默认索引实现为 B+ 树索引。

4. B 树和 B+ 树的区别

1. B+ 树是基于 B 树和**叶子节点顺序访问指针**进行实现，它具有 B 树的平衡性，并且通过顺序访问指针来提高**范围查询**的性能。
2. 在 B+ 树中，一个节点中的 key 从左到右非递减排列，如果某个指针的左右相邻 key 分别是 key[i] 和 key[i+1]，且不为 null，则该指针指向节点的所有 key 大于等于 key[i] 且小于等于 key[i+1]。
3. 进行查找操作时，首先在根节点进行二分查找，找到一个 key 所在的指针，然后递归地在指针所指向的节点进行查找。直到查找到叶子节点，然后在叶子节点上进行二分查找，找出 key 所对应的

data。
4. 插入、删除操作会破坏平衡树的平衡性，因此在插入删除操作之后，需要对树进行一个分裂、合并、旋转等操作来维护平衡性。

5. 为什么用 B+ 树不用 B 树

用 B+ 树而不用 B 树考虑的是 IO 对性能的影响，B 树的每个节点都存储数据，而 B+ 树只有叶子节点才存储数据，所以查找相同数据量的情况下，B 树的高**度更高**，**IO 更频繁。**数据库索引是存储在磁盘上的，当数据量大时，就不能把整个索引全部加载到内存了，只能逐一加载每一个磁盘页（对应索引树的节点）。

6. 聚簇索引和非聚簇索引的区别

聚簇索引是对磁盘上实际数据重新组织以按指定的一个或多个列的值排序的算法。特点是存储数据的顺序和索引顺序一致。一般情况下主键会默认创建聚簇索引，且一张表只允许存在一个聚簇索引。

聚簇索引和非非聚簇索引的区别：

聚簇索引的叶子节点就是数据节点，而非聚簇索引的叶子节点仍然是索引节点，只不过有指向对应数据块的指针。

7. 什么是事务，它有什么特性

数据库事务(transaction)是访问并可能操作各种数据项的一个数据库操作序列，这些操作**要么全部执行，要么全部不执行**，是一个不可分割的工作单位。

事务的四大特性：

- 原子性：原子性是指包含事务的操作要么全部执行成功，要么全部失败回滚。 (undo log)
- 一致性：一致性指事务在执行前后状态是一致的。
- 隔离性：一个事务所进行的修改在最终提交之前，对其他事务是不可见的。 (锁与MVCC)
- 持久性：数据一旦提交，其所作的修改将永久地保存到数据库中。 (redo log)

8. MySQL 事务特性实现原理

原子性：事务的所有修改操作(增、删、改)的相反操作都会写入`undo log`，事务没有成功提交，系统则会执行`undo log`中相应的撤销操作，达到事务回滚的目的。

一致性：一致性指事务在执行前后状态是一致的。

隔离性：一个事务所进行的修改在最终提交之前，对其他事务是不可见的。 (锁与MVCC)

持久性：数据一旦提交，其所作的修改将永久地保存到数据库中。 事务的所有修改操作(增、删、改)，数据库都会生成一条 redo 日志记录到redo log。区别于undo log记录 SQL 语句、redo log记录的是事务对数据库的哪个数据页做了什么修改，属于物理日志。redo日志应用场景：数据库系统直接崩溃，需要进行恢复，一般数据库都会使用按时间点备份的策略，首先将数据库恢复到最近备份的时间点状态，之后读取该时间点之后的redo log记录，重新执行相应记录，达到最终恢复的目的。

9. 什么是脏读、幻读和不可重复读

当多个事务并发执行时，可能会出现以下问题：

- 脏读：事务A更新了数据，但还没有提交，这时事务B读取到事务A更新后的数据，然后事务A回滚了，事务B读取到的数据就成为脏数据了。
- 不可重复读：事务A对数据进行多次读取，事务B在事务A多次读取的过程中执行了更新操作并提交了，导致事务A多次读取到的数据并不一致。

- 幻读：事务A在读取数据后，事务B向事务A读取的数据中插入了几条数据，事务A再次读取数据时发现多了几条数据，和之前读取的数据不一致。
- 丢失修改：事务A和事务B都对同一个数据进行修改，事务A先修改，事务B随后修改，事务B的修改覆盖了事务A的修改。

不可重复度和幻读看起来比较像，它们主要的区别是：在**不可重复读**中，发现数据不一致主要是**数据被更新**了。在**幻读**中，发现数据不一致主要是**数据增多或者减少**了。

10. 数据库的隔离级别

- 未提交读：一个事务在提交前，它的修改对其他事务也是可见的。
- 提交读：一个事务提交之后，它的修改才能被其他事务看到。
- 可重复读：在同一个事务中多次读取到的数据是一致的。
- 串行化：需要加锁实现，会强制事务串行执行。

隔离级别	脏读	不可重复读	幻读
未提交读	允许	允许	允许
提交读	不允许	允许	允许
可重复读	不允许	不允许	允许
串行化	不允许	不允许	不允许

MySQL的默认隔离级别是可重复读

02 MySQL

1. MySQL 的锁机制

当数据库有并发事务的时候，**保证数据访问顺序**的机制称为锁机制。

隔离级别	实现方式
未提交读	总是读取最新的数据，无需加锁
提交读	读取数据时加共享锁，**读取数据后**释放共享锁
可重复读	读取数据时加共享锁，**事务结束**后释放共享锁
串行化	锁定整个范围的键，一直持有锁直到事务结束

2. MVCC 是什么

多版本并发控制（MVCC）是一种用来**解决读-写冲突**的无锁并发控制，也就是为事务分配单向增长的时间戳，为每个修改保存一个版本，版本与事务时间戳关联，读操作只读该事务开始前的数据库的快照。

在并发读写数据库时，可以做到在读操作时不用阻塞写操作，写操作也不用阻塞读操作，提高了数据库并发读写的性能；同时还可以解决脏读，幻读，不可重复读等事务隔离问题，但不能解决更新丢失问题。

行级锁是一个**悲观锁**，而MVCC是一个**乐观锁**，乐观锁在一定程度上可以避免加锁操作，因此开销更低。MVCC，通过在每行记录后面保存两个隐藏的列来实现：一个保存了**行的创建时间**，一个保存**行的过期时间**（删除时间）。当然，这里的时间并不是时间戳，而是系统版本号，每开始一个新的事务，系统版本号就会递增。

3. MySQL 的行锁和表锁

- 表级锁：开销小，加锁快，不会出现死锁。锁定粒度大，发生锁冲突的概率最高，并发量最低。
- 行级锁：开销大，加锁慢，会出现死锁。锁定粒度小，发生锁冲突的概率小，并发量最高。

4. MySQL 共享锁和排他锁（读写锁）

- 共享锁：共享锁又称读锁，简写为S锁，一个事务对一个数据对象加了S锁，可以对这个数据对象进行读取操作，但不能进行更新操作。并且在加锁期间其他事务只能对这个数据对象加S锁，不能加X锁。
- 排他锁：排他锁又称为写锁，简写为X锁，一个事务对一个数据对象加了X锁，可以对这个对象进行读取和更新操作，加锁期间，其他事务不能对该数据对象进行加X锁或S锁。

03 Redis

1. 什么是 Redis

Redis 是一个内存中的数据结构存储系统，它可以用作数据库、缓存和消息中间件。

Redis 可以以键值对的形式作为一种 NoSql 数据库，因为 Redis 将所有数据都存放在**内存**中，所以其可达到的数据**读写性能**非常惊人。

除了性能惊人，Redis还具备**可持久化**的特点，它可以将内存中的数据利用快照和日志的形式保存到磁盘上。另外，Redis还提供了键过期，发布订阅，事务，流水线等其他功能。

2. Redis 的特性

\1. 读写速度块

- Redis将所有数据都存放在内存中

- Redis使用C语言编写，接近底层硬件性能好

- Redis使用单线程架构，减少CPU切换的性能消耗，也不用考虑锁的问题

\2. 支持多种数据结构

- 基于键值对的数据结构服务器

- 支持数据结构：字符串（strings）， 哈希表（hashes）， 列表（lists）， 集合（sets）， 有序集合（sorted sets）等

\3. 功能丰富，可扩展性强

- Redis提供键过期功能，可以实现缓存

- Redis提供发布订阅功能，可以作为消息系统

- Redis提供管道Pipeline功能，客户端可以将一批命令一次性传到Redis中，减少网络开销

\4. 简单稳定

- 代码简单，开源版本中代码行仅在5万行左右

- 处理模型简单，采用单线程模型，所以使得Redis服务端处理模型变得简单

- 不依赖操作系统中的任何类库，独立性好

\5. 可持久化

- RDB方式

- AOF方式

\6. 主从复制

\7. 高可用和分布式

- Redis哨兵

- Redis集群

3. Redis 的应用场景

支撑功能	应用场景
键过期功能	缓存session会话，缓存mysql数据
列表，有序集合	热度排名排行榜，发布时间排行榜
集合	共同好友，共同兴趣，广告投放
计数器	文章浏览量，视频播放量，评论留言量
发布订阅	消息队列系统，缓存ELK日志收集

4. Redis 的持久化方式

Redis 提供了不同级别的持久化方式:

- **RDB持久化方式** 能够在指定的时间间隔能对数据进行快照存储
- **AOF持久化方式** 记录每次对服务器写的操作，当服务器重启的时候会重新执行这些命令来恢复原始的数据。AOF命令以redis协议追加保存每次写的操作到文件末尾。

5. 什么是缓存雪崩、缓存穿透和缓存击穿

下图是一个正常的系统架构图，其中缓存的作用是**减轻数据库的压力，提升系统的性能**，无论是缓存雪崩、缓存击穿还是缓存穿透都是缓存失效了导致数据库压力过大。

缓存雪崩：缓存雪崩是指在**某一个时刻**出现**大规模**的缓存失效的情况，大量的请求**直接打在数据库上面**，可能会导致数据库宕机，如果这时重启数据库并不能解决根本问题，会再次造成缓存雪崩。

缓存击穿：缓存雪崩是大规模的 key 失效，而缓存击穿是一个**热点**的 Key，有大并发集中对其进行访问，突然间这个Key失效了，导**致大并发全部打在数据库上**，导致数据库压力剧增，这种现象就叫做缓存击穿。

缓存穿透：缓存穿透是指用户的请求**没有经过缓存**而直接请求到数据库上了，比如用户请求的key在Redis中不存在，或者用户恶意伪造大量不存在的 key 进行请求，都可以**绕过缓存**，导致数据库压力太大挂掉。

6. 缓存雪崩的解决方法

缓存雪崩是同一时刻出现大规模的缓存失效导致的，其成因主要有两种情况：（1）**Redis 整体宕机**导致缓存失效；（2）多个 key 被设置了相同的**过期时间。**

针对上述成因可以提供如下解决方案:

- 为避免 Redis 宕机造成缓存雪崩，可以搭建 Redis 集群，提高缓存可靠性
- 尽量不要设置相同的过期时间，可以在原有的过期时间加上随机数

7. 缓存击穿的解决方法

缓存击穿是大量并发请求集中访问的热点 Key 突然缓存失效，而其失效的最直接原因可能是 **key 过期**，所以可以延长或不设置热点 Key 的过期时间，避免其过期失效。

8. 缓存穿透的解决方法

缓存穿透是用户请求不存在的数据，导致缓存不命中直接请求数据库。这种情况的成因可能是：（1）Redis 确实不存在请求的 Key；（2）用户恶意请求大量不存在的 Key

针对上述成因可以提供如下解决方案：

- 缓存空值：如果某个 Key 在Redis和数据库中都不存在，则把该 Key 存入 Redis 并将值置为空值
- 参数校验：对用户 Id 进行校验，拦截不合法的用户请求
- 布隆过滤器：判断 Key 是否存在，如果判断不存在那么一定不存在；如果判断存在，并不能保证一定存在。

9. 布隆过滤器的实现原理

多个哈希函数的值查询结果都是 1 才表示数据存在，只要有一个 0 就不存在。

哈希函数个数越多，误判概率越低。

Redis面试八股