前言（必读）

对于 C++ 的学习，很多人可能学了之后，不知道自己处于哪个阶段，也不到究竟要学到哪个程度，帅地觉得，验证自己学得如何最好的面试，就是尝试去面试，而面试无非就是问你一些面试题，所以呢，帅地整理了这些 C++ 面试题，从 **C++ 基础，集合，面向对象再到内存管理**，并且附带了详细的答案，无论是想面试还是想看看自己学得如何，那么这份面试题，都值得你去学习。

当然，如果单单只会 C++，是很难进大公司的，所以计算机基础之类的也得学，所以本 PDF 还包括了**Redis，计算机网络，操作系统，MySQL**等通用基础知识。

你现在手里的这份可能不是最新版，因为帅地看到好的面试题，就会整理进去，强烈建议你去看最新版的，微信搜索关注「**帅地玩编程**」，回复「**Java面试题**」，即可获取最新版的 PDF 哦，扫码直达

关注后，回复「**1023**」，即可获取最新版 PDF。

C++ 基础

1、C和C++有什么区别?

- C++是面向对象的语言，而C是面向过程的语言；
- 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C中没有；
- C++引入类的概念，而C中没有；
- C++引入函数重载的特性，而C中没有

2、a和&a有什么区别?

假设数组int a[10]; int (*p)[10] = &a;其中：

- a是数组名，是数组首元素地址，+1表示地址值加上一个int类型的大小，如果a的值是0x00000001，加1操作后变为0x00000005。*(a + 1) = a[1]。
- &a是数组的指针，其类型为int (*)[10]（就是前面提到的数组指针），其加1时，系统会认为是数组首地址加上整个数组的偏移（10个int型变量），值为数组a尾元素后一个元素的地址。

- 若(int *)p ，此时输出 *p时，其值为a[0]的值，因为被转为int *类型，解引用时按照int类型大小来读取。

3、static关键字有什么作用?

- 修饰局部变量时，使得该变量在静态存储区分配内存；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；
- 修饰全局变量时，使得该变量在静态存储区分配内存；在声明该变量的整个文件中都是可见的，而在文件外是不可见的；
- 修饰函数时，在声明该函数的整个文件中都是可见的，而在文件外是不可见的，从而可以在多人协作时避免同名的函数冲突；
- 修饰成员变量时，所有的对象都只维持一份拷贝，可以实现不同对象间的数据共享；不需要实例化对象即可访问；不能在类内部初始化，一般在类外部初始化，并且初始化时不加`static`；
- 修饰成员函数时，该函数不接受`this`指针，只能访问类的静态成员；不需要实例化对象即可访问。

4、#define和const有什么区别?

- 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`常量是在编译阶段使用；
- 类型和安全检查不同：`#define`宏没有类型，不做任何类型检查，仅仅是代码展开，可能产生边际效应等错误，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。
- 定义域不同：`#define`宏不受定义域限制，而`const`常量只在定义域内有效。

5、对于一个频繁使用的短小函数，应该使用什么来实现？有什么优缺点?

应该使用`inline`内联函数，即编译器将`inline`内联函数内的代码替换到函数被调用的地方。

优点：

- 在内联函数被调用的地方进行代码展开，省去函数调用的时间，从而提高程序运行效率；
- 相比于宏函数，内联函数在代码展开时，编译器会进行语法安全检查或数据类型转换，使用更加安全；

缺点：

- 代码膨胀，产生更多的开销；
- 如果内联函数内代码块的执行时间比调用时间长得多，那么效率的提升并没有那么大；
- 如果修改内联函数，那么所有调用该函数的代码文件都需要重新编译；
    - 内联声明只是建议，是否内联由编译器决定，所以实际并不可控。

6、什么是智能指针？智能指针有什么作用？分为哪几种？各自有什么样的特点?

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。

分为`auto_ptr`、`unique_ptr`、`shared_ptr`和`weak_ptr`四种，各自的特点：

- 对于`auto_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；但`auto_ptr`在C++11中被摒弃，其主要问题在于：
  - 对象所有权的转移，比如在函数传参过程中，对象所有权不会返还，从而存在潜在的内存崩溃问题；
  - 不能指向数组，也不能作为STL容器的成员。
- 对于`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，因为无法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；
- 对于`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；
- 对于`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，从而导致死锁问题。而`weak_ptr`是对对象的一种弱引用，可以绑定到`shared_ptr`，但不会增加对象的引用计数。

7、shared_ptr是如何实现的?

1. 构造函数中计数初始化为1；
2. 拷贝构造函数中计数值加1；
3. 赋值运算符中，左边的对象引用计数减1，右边的对象引用计数加1；
4. 析构函数中引用计数减1；
5. 在赋值运算符和析构函数中，如果减1后为0，则调用`delete`释放对象。

8、右值引用有什么作用?

右值引用的主要目的是为了实现转移语义和完美转发，消除两个对象交互时不必要的对象拷贝，也能够更加简洁明确地定义泛型函数

9、悬挂指针与野指针有什么区别?

- 悬挂指针：当指针所指向的对象被释放，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挂指针；
- 野指针：未初始化的指针被称为野指针。

10、静态链接和动态链接有什么区别?

- 静态链接是在编译链接时直接将需要的执行代码拷贝到调用处；

  优点在于程序在发布时不需要依赖库，可以独立执行，缺点在于程序的体积会相对较大，而且如果静态库更新之后，所有可执行文件需要重新链接；

- 动态链接是在编译时不直接拷贝执行代码，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代码时，在共享执行内存中寻找已经加载的动态库可执行代码，实现运行时链接；

  优点在于多个程序可以共享同一个动态库，节省资源；

  缺点在于由于运行时加载，可能影响程序的前期执行性能。

11、变量的声明和定义有什么区别

变量的定义为变量分配地址和存储空间， 变量的声明不分配地址。一个变量可以在多个地方声明， 但是只在一个地方定义。加入extern 修饰的是变量的声明，说明此变量将在文件以外或在文件后面部分定义。

说明：很多时候一个变量，只是声明不分配内存空间，直到具体使用时才初始化，分配内存空间， 如外部变量。

```
int main()
{
   extern int A;
   //这是个声明而不是定义，声明A是一个已经定义了的外部变量
   //注意：声明外部变量时可以把变量类型去掉如：extern A;
   dosth(); //执行函数
}
int A; //是定义，定义了A为整型的外部变量
```

12、简述#ifdef、#else、#endif和#ifndef的作用

利用#ifdef、#endif将某程序功能模块包括进去，以向特定用户提供该功能。在不需要时用户可轻易将其屏蔽。

```
#ifdef MATH
#include "math.c"
#endif
```

在子程序前加上标记，以便于追踪和调试。

```
#ifdef DEBUG
printf ("Indebugging......!");
#endif
```

应对硬件的限制。由于一些具体应用环境的硬件不一样，限于条件，本地缺乏这种设备，只能绕过硬件，直接写出预期结果。

「注意」：虽然不用条件编译命令而直接用if语句也能达到要求，但那样做目标程序长（因为所有语句都编译），运行时间长（因为在程序运行时间对if语句进行测试）。而采用条件编译，可以减少被编译的语句，从而减少目标程序的长度，减少运行时间。

13、写出int 、bool、 float 、指针变量与 “零值”比较的if 语句

```
//int与零值比较
if ( n == 0 )
if ( n != 0 )

//bool与零值比较
if   (flag) //   表示flag为真
if   (!flag) //   表示flag为假

//float与零值比较
```

```
const float EPSINON = 0.00001;
if ((x >= - EPSINON) && (x <= EPSINON) //其中EPSINON是允许的误差（即精度）。
//指针变量与零值比较
if (p == NULL)
if (p != NULL)
```

14、结构体可以直接赋值吗

声明时可以直接初始化，同一结构体的不同对象之间也可以直接赋值，但是当结构体中含有指针“成员”时一定要小心。

「注意」：当有多个指针指向同一段内存时，某个指针释放这段内存可能会导致其他指针的非法操作。因此在释放前一定要确保其他指针不再使用这段内存空间。

15、sizeof 和strlen 的区别

- sizeof是一个操作符，strlen是库函数。
- sizeof的参数可以是数据的类型，也可以是变量，而strlen只能以结尾为‘\0’的字符串作参数。
- 编译器在编译时就计算出了sizeof的结果，而strlen函数必须在运行时才能计算出来。并且sizeof计算的是数据类型占内存的大小，而strlen计算的是字符串实际的长度。
- 数组做sizeof的参数不退化，传递给strlen就退化为指针了

16、C 语言的关键字 static 和 C++ 的关键字 static 有什么区别

在 C 中 static 用来修饰局部静态变量和外部静态变量、函数。而 C++中除了上述功能外，还用来定义类的成员变量和函数。即静态成员和静态成员函数。

「注意」：编程时 static 的记忆性，和全局性的特点可以让在不同时期调用的函数进行通信，传递信息，而 C++的静态成员则可以在多个对象实例间进行通信，传递信息。

17、volatile有什么作用

- 状态寄存器一类的并行设备硬件寄存器。
- 一个中断服务子程序会访问到的非自动变量。
- 多线程间被几个任务共享的变量。

「注意」：虽然volatile在嵌入式方面应用比较多，但是在PC软件的多线程中，volatile修饰的临界变量也是非常实用的。

18、一个参数可以既是const又是volatile吗

可以，用const和volatile同时修饰变量，表示这个变量在程序内部是只读的，不能改变的，只在程序外部条件变化下改变，并且编译器不会优化这个变量。每次使用这个变量时，都要小心地去内存读取这个变量的值，而不是去寄存器读取它的备份。

注意：在此一定要注意const的意思，const只是不允许程序中的代码改变某一变量，其在编译期发挥作用，它并没有实际地禁止某段内存的读写特性。

19、全局变量和局部变量有什么区别？操作系统和编译器是怎么知道的?

- 全局变量是整个程序都可访问的变量，谁都可以访问，生存期在整个程序从运行到结束（在程序结束时所占内存释放）；
- 而局部变量存在于模块（子程序，函数）中，只有所在模块可以访问，其他模块不可直接访问，模块结束（函数调用完毕），局部变量消失，所占据的内存释放。
- 操作系统和编译器，可能是通过内存分配的位置来知道的，全局变量分配在全局数据段并且在程序开始运行的时候被加载.局部变量则分配在堆栈里面。

20、简述strcpy、sprintf 与memcpy 的区别

- 操作对象不同，strcpy 的两个操作对象均为字符串，sprintf 的操作源对象可以是多种数据类型， 目的操作对象是字符串，memcpy 的两个对象就是两个任意可操作的内存地址，并不限于何种数据类型。
- 执行效率不同，memcpy 最高，strcpy 次之，sprintf 的效率最低。
- 实现功能不同，strcpy 主要实现字符串变量间的拷贝，sprintf 主要实现其他数据类型格式到字 符串的转化，memcpy 主要是内存块间的拷贝。

「注意」：strcpy、sprintf 与memcpy 都可以实现拷贝的功能，但是针对的对象不同，根据实际需求，来 选择合适的函数实现拷贝功能。

21、请解析((void ()( ))0)( )的含义

- void (*0)( )：是一个返回值为void，参数为空的函数指针0。
- (void (*)( ))0：把0转变成一个返回值为void，参数为空的函数指针。
- **(void ()**( ))0：在上句的基础上加*表示整个是一个返回值为void，无参数，并且起始地址为0的函数的名字。
- (**(void ()**( ))0)( )：这就是上句的函数名所对应的函数的调用。

22、C语言的指针和引用和c++的有什么区别?

- 指针有自己的一块空间，而引用只是一个别名；
- 使用sizeof看一个指针的大小是4，而引用则是被引用对象的大小；
- 作为参数传递时，指针需要被解引用才可以对对象进行操作，而直接对引 用的修改都会改变引用所指向的对象；
- 可以有const指针，但是没有const引用；
- 指针在使用中可以指向其它对象，但是引用只能是一个对象的引用，不能 被改变；
- 指针可以有多级指针（**p），而引用止于一级；
- 指针和引用使用++运算符的意义不一样；
- 如果返回动态内存分配的对象或者内存，必须使用指针，引用可能引起内存泄露。

23、typedef 和define 有什么区别

- 用法不同：typedef 用来定义一种数据类型的别名，增强程序的可读性。define 主要用来定义 常量，以及书写复杂使用频繁的宏。
- 执行时间不同：typedef 是编译过程的一部分，有类型检查的功能。define 是宏定义，是预编译的部分，其发生在编译之前，只是简单的进行字符串的替换，不进行类型的检查。
- 作用域不同：typedef 有作用域限定。define 不受作用域约束，只要是在define 声明后的引用 都是正确的。
- 对指针的操作不同：typedef 和define 定义的指针时有很大的区别。

「注意」：typedef 定义是语句，因为句尾要加上分号。而define 不是语句，千万不能在句尾加分号。

24、指针常量与常量指针区别

指针常量是指定义了一个指针，这个指针的值只能在定义时初始化，其他地方不能改变。常量指针 是指定义了一个指针，这个指针指向一个只读的对象，不能通过常量指针来改变这个对象的值。指针常量强调的是指针的不可改变性，而常量指针强调的是指针对其所指对象的不可改变性。

「注意」：无论是指针常量还是常量指针，其最大的用途就是作为函数的形式参数，保证实参在被调用 函数中的不可改变特性。

25、简述队列和栈的异同

队列和栈都是线性存储结构，但是两者的插入和删除数据的操作不同，队列是“先进先出”，栈是 “后进先出”。

「注意」：区别栈区和堆区。堆区的存取是“顺序随意”，而栈区是“后进先出”。栈由编译器自动分 配释放 ，存放函数的参数值，局部变量的值等。其操作方式类似于数据结构中的栈。堆一般由程序员 分配释放， 若程序员不释放，程序结束时可能由OS 回收。分配方式类似于链表。它与本题中的堆和栈是两回事。堆栈只是一种数据结构，而堆区和栈区是程序的不同内存存储区域。

26、设置地址为0x67a9 的整型变量的值为0xaa66

```
int *ptr;
ptr = (int *)0x67a9;
*ptr = 0xaa66;
```

「注意」：这道题就是强制类型转换的典型例子，无论在什么平台地址长度和整型数据的长度是一样的， 即一个整型数据可以强制转换成地址指针类型，只要有意义即可。

27、C语言的结构体和C++的有什么区别

- C语言的结构体是不能有函数成员的，而C++的类可以有。
- C语言的结构体中数据成员是没有private、public和protected访问限定的。而C++的类的成员有这些访问限定。
- C语言的结构体是没有继承关系的，而C++的类却有丰富的继承关系。

「注意」：虽然C的结构体和C++的类有很大的相似度，但是类是实现面向对象的基础。而结构体只可以简单地理解为类的前身。

28、简述指针常量与常量指针的区别

- 指针常量是指定义了一个指针，这个指针的值只能在定义时初始化，其他地方不能改变。常量指针是指定义了一个指针，这个指针指向一个只读的对象，不能通过常量指针来改变这个对象的值。
- 指针常量强调的是指针的不可改变性，而常量指针强调的是指针对其所指对象的不可改变性。

「注意」：无论是指针常量还是常量指针，其最大的用途就是作为函数的形式参数，保证实参在被调用函数中的不可改变特性。

29、如何避免“野指针”

- 指针变量声明时没有被初始化。解决办法：指针声明时初始化，可以是具体的地址值，也可让它指向NULL。
- 指针p被free或者delete之后，没有置为NULL。解决办法：指针指向的内存空间被释放后指针应该指向NULL。
- 指针操作超越了变量的作用范围。解决办法：在变量的作用域结束前释放掉变量的地址空间并且让指针指向NULL。

30、句柄和指针的区别和联系是什么?

句柄和指针其实是两个截然不同的概念。Windows系统用句柄标记系统资源，隐藏系统的信息。你只要知道有这个东西，然后去调用就行了，它是个32it的uint。指针则标记某个物理内存地址，两者是不同的概念。

31、说一说extern“C”

extern "C"的主要作用就是为了能够正确实现C++代码调用其他C语言代码。加上extern "C"后，会指示编译器这部分代码按C语言（而不是C++）的方式进行编译。由于C++支持函数重载，因此编译器编译函数的过程中会将函数的参数类型也加到编译后的代码中，而不仅仅是函数名；而C语言并不支持函数重载，因此编译C语言代码的函数时不会带上函数的参数类型，一般只包括函数名。

这个功能十分有用处，因为在C++出现以前，很多代码都是C语言写的，而且很底层的库也是C语言写的，为了更好的支持原来的C代码和已经写好的C语言库，需要在C++中尽可能的支持C，而extern "C"就是其中的一个策略。

- C++代码调用C语言代码
- 在C++的头文件中使用
- 在多个人协同开发时，可能有的人比较擅长C语言，而有的人擅长C++，这样的情况下也会有用到

32、对c++中的smart pointer四个智能指针：shared_ptr,unique_ptr,weak_ptr,auto_ptr的理解

C++里面的四个智能指针: auto_ptr, shared_ptr, weak_ptr, unique_ptr 其中后三个是c++11支持，并且第一个已经被11弃用。

智能指针的作用是管理一个指针，因为存在以下这种情况：申请的空间在函数结束时忘记释放，造成内存泄漏。使用智能指针可以很大程度上的避免这个问题，因为智能指针就是一个类，当超出了类的作用域是，类会自动调用析构函数，析构函数会自动释放资源。所以智能指针的作用原理就是在函数结束时自动释放内存空间，不需要手动释放内存空间。

- auto_ptr（c++98的方案，cpp11已经抛弃）

采用所有权模式。

```
auto_ptr< string> p1 (new string ("I reigned lonely as a cloud."));
auto_ptr<string> p2;
p2 = p1; //auto_ptr不会报错.
```

此时不会报错，p2剥夺了p1的所有权，但是当程序运行时访问p1将会报错。所以auto_ptr的缺点是：存在潜在的内存崩溃问题!

- unique_ptr（替换auto_ptr）

unique_ptr实现独占式拥有或严格拥有概念，保证同一时间内只有一个智能指针可以指向该对象。它对于避免资源泄露(例如“以new创建对象后因为发生异常而忘记调用delete”)特别有用。

采用所有权模式。

```
unique_ptr<string> p3 (new string ("auto"));   //#4
unique_ptr<string> p4;                         //#5
p4 = p3;//此时会报错！！
```

编译器认为p4=p3非法，避免了p3不再指向有效数据的问题。因此，unique_ptr比auto_ptr更安全。

另外unique_ptr还有更聪明的地方：当程序试图将一个 unique_ptr 赋值给另一个时，如果源 unique_ptr 是个临时右值，编译器允许这么做；如果源 unique_ptr 将存在一段时间，编译器将禁止这么做，比如：

```
unique_ptr<string> pu1(new string ("hello world"));
unique_ptr<string> pu2;
pu2 = pu1;                                     // #1 not allowed
unique_ptr<string> pu3;
pu3 = unique_ptr<string>(new string ("You"));   // #2 allowed
```

其中#1留下悬挂的unique_ptr(pu1)，这可能导致危害。而#2不会留下悬挂的unique_ptr，因为它调用 unique_ptr 的构造函数，该构造函数创建的临时对象在其所有权让给 pu3 后就会被销毁。这种随情况而已的行为表明，unique_ptr 优于允许两种赋值的auto_ptr 。

「注意」：如果确实想执行类似与#1的操作，要安全的重用这种指针，可给它赋新值。C++有一个标准库函数std::move()，让你能够将一个unique_ptr赋给另一个。例如：

```
unique_ptr<string> ps1, ps2;
ps1 = demo("hello");
ps2 = move(ps1);
ps1 = demo("alexia");
cout << *ps2 << *ps1 << endl;
```

- shared_ptr

shared_ptr实现共享式拥有概念。多个智能指针可以指向相同对象，该对象和其相关资源会在“最后一个引用被销毁”时候释放。从名字share就可以看出了资源可以被多个指针共享，它使用计数机制来表明资源被几个指针共享。可以通过成员函数use_count()来查看资源的所有者个数。除了可以通过new来构造，还可以通过传入auto_ptr, unique_ptr,weak_ptr来构造。当我们调用release()时，当前指针会释放资源所有权，计数减一。当计数等于0时，资源会被释放。

shared_ptr 是为了解决 auto_ptr 在对象所有权上的局限性(auto_ptr 是独占的), 在使用引用计数的机制上提供了可以共享所有权的智能指针。

成员函数：

use_count 返回引用计数的个数

unique 返回是否是独占所有权(use_count 为 1)

swap 交换两个 shared_ptr 对象(即交换所拥有的对象)

reset 放弃内部对象的所有权或拥有对象的变更, 会引起原有对象的引用计数的减少

get 返回内部对象(指针), 由于已经重载了()方法, 因此和直接使用对象是一样的.如 shared_ptrsp(new int(1)); sp 与 sp.get()是等价的

- weak_ptr

weak_ptr 是一种不控制对象生命周期的智能指针, 它指向一个 shared_ptr 管理的对象. 进行该对象的内存管理的是那个强引用的 shared_ptr. weak_ptr只是提供了对管理对象的一个访问手段。weak_ptr 设计的目的是为配合 shared_ptr 而引入的一种智能指针来协助 shared_ptr 工作, 它只可以从一个 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。weak_ptr是用来解决shared_ptr相互引用时的死锁问题,如果说两个shared_ptr相互引用,那么这两个指针的引用计数永远不可能下降为0,资源永远不会释放。它是对对象的一种弱引用，不会增加对象的引用计数，和shared_ptr之间可以相互转化，shared_ptr可以直接赋值给它，它可以通过调用lock函数来获得shared_ptr。

```
class B;
class A
{
public:
shared_ptr<B> pb_;
~A()
{
     cout<<"A delete
";
}
};
class B
{
public:
shared_ptr<A> pa_;
~B()
{
    cout<<"B delete
";
}
};
void fun()
{
    shared_ptr<B> pb(new B());
    shared_ptr<A> pa(new A());
    pb->pa_ = pa;
    pa->pb_ = pb;
    cout<<pb.use_count()<<endl;
    cout<<pa.use_count()<<endl;
}
int main()
{
    fun();
    return 0;
}
```

可以看到fun函数中pa ，pb之间互相引用，两个资源的引用计数为2，当要跳出函数时，智能指针pa，pb析构时两个资源引用计数会减一，但是两者引用计数还是为1，导致跳出函数时资源没有被释放（A B的析构函数没有被调用），如果把其中一个改为weak_ptr就可以了，我们把类A里面的shared_ptr pb*; 改为weak_ptr pb*; 运行结果如下，这样的话，资源B的引用开始就只有1，当pb析构时，B的计数变为0，B得到释放，B释放的同时也会使A的计数减一，同时pa析构时使A的计数减一，那么A的计数为0，A得到释放。

「注意」：不能通过weak_ptr直接访问对象的方法，比如B对象中有一个方法print(),我们不能这样访问，pa->pb->*print(); 英文pb*是一个weak_ptr，应该先把它转化为shared_ptr,如：shared_ptr p = pa->pb_.lock(); p->print();

33、C++的顶层const和底层const？

- 底层const是代表对象本身是一个常量（不可改变）；
- 顶层const是代表指针的值是一个常量,而指针的值(即对象的地址)的内容可以改变（指向的不可改变）；

C++ 面向对象

> 你现在手里的这份可能不是最新版，因为帅地看到好的面试题，就会整理进去，强烈建议你去看最新版的，微信搜索关注「**帅地玩编程**」，回复「**1023**」，即可获取最新版的 PDF 哦，扫码直达

关注后，回复「**1023**」，即可获取最新版 PDF。

1、面向对象的三大特征是哪些？各自有什么样的特点？

- 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，对不可信的类或对象则进行信息隐藏。
- 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；
- 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。

2、C++中类成员的访问权限

C++通过 public、protected、private 三个关键字来控制成员变量和成员函数的访问权限，它们分别表示公有的、受保护的、私有的，被称为成员访问限定符。在类的内部（定义类的代码内部），无论成员被声明为 public、protected 还是 private，都是可以互相访问的，没有访问权限的限制。在类的外部（定义类的代码之外），只能通过对象访问成员，并且通过对象只能访问 public 属性的成员，不能访问 private、protected 属性的成员

3、多态的实现有哪几种?

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。

4、动态多态有什么作用？有哪些必要条件?

动态多态的作用：

- 隐藏实现细节，使代码模块化，提高代码的可复用性；
- 接口重用，使派生类的功能可以被基类的指针/引用所调用，即向后兼容，提高代码的可扩充性和可维护性。

动态多态的必要条件：

- 需要有继承；
- 需要有虚函数覆盖；
- 需要有基类指针/引用指向子类对象。

5、动态绑定是如何实现的?

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，从而实现动态绑定。

6、纯虚函数有什么作用？如何实现?

定义纯虚函数是为了实现一个接口，起到规范的作用，想要继承这个类就必须覆盖该函数。

实现方式是在虚函数声明的结尾加上`= 0`即可。

7、虚函数表是针对类的还是针对对象的？同一个类的两个对象的虚函数表是怎么维护的?

虚函数表是针对类的，类的所有对象共享这个类的虚函数表，因为每个对象内部都保存一个指向该类虚函数表的指针`vptr`，每个对象的`vptr`的存放地址都不同，但都指向同一虚函数表。

8、为什么基类的构造函数不能定义为虚函数?

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。

9、为什么基类的析构函数需要定义为虚函数?

为了实现动态绑定，基类指针指向派生类对象，如果析构函数不是虚函数，那么在对象销毁时，就会调用基类的析构函数，只能销毁派生类对象中的部分数据，所以必须将析构函数定义为虚函数，从而在对象销毁时，调用派生类的析构函数，从而销毁派生类对象中的所有数据。

10、构造函数和析构函数能抛出异常吗?

- 从语法的角度来说，构造函数可以抛出异常，但从逻辑和风险控制的角度来说，尽量不要抛出异常，否则可能导致内存泄漏。
- 析构函数不可以抛出异常，如果析构函数抛出异常，则异常点之后的程序，比如释放内存等操作，就不会被执行，从而造成内存泄露的问题；而且当异常发生时，C++通常会调用对象的析构函数来释放资源，如果此时析构函数也抛出异常，即前一个异常未处理又出现了新的异常，从而造成程序崩溃的问题。

11、如何让一个类不能实例化?

将类定义为抽象类（也就是存在纯虚函数）或者将构造函数声明为`private`。

12、多继承存在什么问题？如何消除多继承中的二义性?

1. 增加程序的复杂度，使得程序的编写和维护比较困难，容易出错；
2. 在继承时，基类之间或基类与派生类之间发生成员同名时，将出现对成员访问的不确定性，即同名二义性；

   消除同名二义性的方法：

   - 利用作用域运算符`::`，用于限定派生类使用的是哪个基类的成员；
   - 在派生类中定义同名成员，覆盖基类中的相关成员；
3. 当派生类从多个基类派生，而这些基类又从同一个基类派生，则在访问此共同基类的成员时，将产生另一种不确定性，即路径二义性；

   消除路径二义性的方法：

   - 消除同名二义性的两种方法都可以；
   - 使用虚继承，使得不同路径继承来的同名成员在内存中只有一份拷贝。

13、如果类`A`是一个空类，那么`sizeof(A)`的值为多少？为什么?

`sizeof(A)`的值为1，因为编译器需要区分这个空类的不同实例，分配一个字节，可以使这个空类的不同实例拥有独一无二的地址。

14、 覆盖和重载之间有什么区别?

- 覆盖是指派生类中重新定义的函数，其函数名、参数列表、返回类型与父类完全相同，只是函数体存在区别；覆盖只发生在类的成员函数中；
- 重载是指两个函数具有相同的函数名，不同的参数列表，不关心返回值；当调用函数时，根据传递的参数列表来判断调用哪个函数；重载可以是类的成员函数，也可以是普通函数。

15、拷贝构造函数和赋值运算符重载之间有什么区别?

- 拷贝构造函数用于构造新的对象；

```
Student s;
Student s1 = s; // 隐式调用拷贝构造函数
Student s2(s);  // 显式调用拷贝构造函数
```

- 赋值运算符重载用于将源对象的内容拷贝到目标对象中，而且若源对象中包含未释放的内存需要先将其释放；

```
Student s;
Student s1;
s1 = s; // 使用赋值运算符
```

一般情况下，类中包含指针变量时需要重载拷贝构造函数、赋值运算符和析构函数。

16、对虚函数和多态的理解

多态的实现主要分为静态多态和动态多态，静态多态主要是重载，在编译的时候就已经确定；动态多态是用虚函数机制实现的，在运行期间动态绑定。举个例子：一个父类类型的指针指向一个子类对象时候，使用父类的指针去调用子类中重写了的父类中的虚函数的时候，会调用子类重写过后的函数，在父类中声明为加了virtual关键字的函数，在子类中重写时候不需要加virtual也是虚函数。

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，实际的虚函数在代码段(.text)中。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。使用了虚函数，会增加访问内存开销，降低效率。

17、请你来说一下C++中struct和class的区别

在C++中，class和struct做类型定义是只有两点区别：

- 默认继承权限不同，class继承默认是private继承，而struct默认是public继承
- class还可用于定义模板参数，像typename，但是关键字struct不能同于定义模板参数 C++保留struct关键字，原因
- 保证与C语言的向下兼容性，C++必须提供一个struct
- C++中的struct定义必须百分百地保证与C语言中的struct的向下兼容性，把C++中的最基本的对象单元规定为class而不是struct，就是为了避免各种兼容性要求的限制
- 对struct定义的扩展使C语言的代码能够更容易的被移植到C++中

18、说说强制类型转换运算符

「static_cast」

- 用于非多态类型的转换
- 不执行运行时类型检查（转换安全性不如 dynamic_cast）
- 通常用于转换数值数据类型（如 float -> int）
- 可以在整个类层次结构中移动指针，子类转化为父类安全（向上转换），父类转化为子类不安全（因为子类可能有不在父类的字段或方法）

「dynamic_cast」

- 用于多态类型的转换
- 执行行运行时类型检查
- 只适用于指针或引用
- 对不明确的指针的转换将失败（返回 nullptr），但不引发异常
- 可以在整个类层次结构中移动指针，包括向上转换、向下转换

「const_cast」

- 用于删除 const、volatile 和 __unaligned 特性（如将 const int 类型转换为 int 类型 ） reinterpret_cast
- 用于位的简单重新解释
- 滥用 reinterpret_cast 运算符可能很容易带来风险。除非所需转换本身是低级别的，否则应- 使用其他强制转换运算符之一。
- 允许将任何指针转换为任何其他指针类型（如 char* 到 int* 或 One_class* 到 Unrelated_class* 之类的转换，但其本身并不安全）
- 也允许将任何整数类型转换为任何指针类型以及反向转换。
- reinterpret_cast 运算符不能丢掉 const、volatile 或 __unaligned 特性。
- reinterpret_cast 的一个实际用途是在哈希函数中，即，通过让两个不同的值几乎不以相同的索引结尾的方式将值映射到索引。

「bad_cast」

- 由于强制转换为引用类型失败，dynamic_cast 运算符引发 bad_cast 异常。

bad_cast 使用

```
try {
    Circle& ref_circle = dynamic_cast<Circle&>(ref_shape);
}
catch (bad_cast b) {
    cout << "Caught: " << b.what();
}
```

19、简述类成员函数的重写、重载和隐藏的区别

（1）重写和重载主要有以下几点不同。

- 范围的区别：被重写的和重写的函数在两个类中，而重载和被重载的函数在同一个类中。
- 参数的区别：被重写函数和重写函数的参数列表一定相同，而被重载函数和重载函数的参数列表一 定不同。
- virtual 的区别：重写的基类中被重写的函数必须要有virtual 修饰，而重载函数和被重载函数可以被 virtual 修饰，也可以没有。

（2）隐藏和重写、重载有以下几点不同。

- 与重载的范围不同：和重写一样，隐藏函数和被隐藏函数不在同一个类中。
- 参数的区别：隐藏函数和被隐藏的函数的参数列表可以相同，也可不同，但是函数名肯定要相同。当参数不相同时，无论基类中的参数是否被virtual 修饰，基类的函数都是被隐藏，而不是被重写。

「注意」：虽然重载和覆盖都是实现多态的基础，但是两者实现的技术完全不相同，达到的目的也是完 全不同的，覆盖是动态态绑定的多态，而重载是静态绑定的多态。

20、类型转换分为哪几种？各自有什么样的特点?

- `static_cast`：用于基本数据类型之间的转换、子类向父类的安全转换、`void*`和其他类型指针之间的转换；
- `const_cast`：用于去除`const`或`volatile`属性；
- `dynamic_cast`：用于子类和父类之间的安全转换，可以实现向上向下转换，因为编译器默认向上转换总是安全的，而向下转换时，`dynamic_cast`具有类型检查的功能；

  `dynamic_cast`转换失败时，对于指针会返回目标类型的`nullptr`，对于引用会返回`bad_cast`异常；
- `reinterpret_cast`：用于不同类型指针之间、不同类型引用之间、指针和能容纳指针的整数类型之间的转换。

21、RTTI是什么？其原理是什么?

RTTI即运行时类型识别，其功能由两个运算符实现：

- `typeid`运算符，用于返回表达式的类型，可以通过基类的指针获取派生类的数据类型；
- `dynamic_cast`运算符，具有类型检查的功能，用于将基类的指针或引用安全地转换成派生类的指针或引用。

22、说一说c++中四种cast转换

C++中四种类型转换是：static_cast, dynamic_cast, const_cast, reinterpret_cast

1、const_cast

- 用于将const变量转为非const

2、static_cast

- 用于各种隐式转换，比如非const转const，void*转指针等, static_cast能用于多态向上转化，如果向下转能成功但是不安全，结果未知；

3、dynamic_cast

用于动态类型转换。只能用于含有虚函数的类，用于类层次间的向上和向下转化。只能转指针或引用。向下转化时，如果是非法的***对于指针返回NULL，对于引用抛异常***。要深入了解内部转换的原理。

- 向上转换：指的是子类向基类的转换
- 向下转换：指的是基类向子类的转换

它通过判断在执行到该语句的时候变量的运行时类型和要转换的类型是否相同来判断是否能够进行向下转换。

4、reinterpret_cast

- 几乎什么都可以转，比如将int转指针，可能会出问题，尽量少用；

5、为什么不使用C的强制转换?

- C的强制转换表面上看起来功能强大什么都能转，但是转化不够明确，不能进行错误检查，容易出错。

23、C++的空类有哪些成员函数

- 缺省构造函数。
- 缺省拷贝构造函数。
- 省析构函数。
- 赋值运算符。
- 取址运算符。
- 取址运算符 const 。

「注意」：有些书上只是简单的介绍了前四个函数。没有提及后面这两个函数。但后面这两个函数也是 空类的默认函数。另外需要注意的是，只有当实际使用这些函数的时候，编译器才会去定义它们。

24、模板函数和模板类的特例化

「引入原因」

编写单一的模板，它能适应多种类型的需求，使每种类型都具有相同的功能，但对于某种特定类型，如果要实现其特有的功能，单一模板就无法做到，这时就需要模板特例化

「定义」对单一模板提供的一个特殊实例，它将一个或多个模板参数绑定到特定的类型或值上

（1）模板函数特例化

必须为原函数模板的每个模板参数都提供实参，且使用关键字template后跟一个空尖括号对<>，表明将原模板的所有模板参数提供实参，举例如下：

```
template<typename T> //模板函数
int compare(const T &v1,const T &v2)
{
    if(v1 > v2) return -1;
    if(v2 > v1) return 1;
    return 0;
}
//模板特例化,满足针对字符串特定的比较，要提供所有实参，这里只有一个T
template<>
int compare(const char* const &v1,const char* const &v2)
{
    return strcmp(p1,p2);
}
```

「本质」特例化的本质是实例化一个模板，而非重载它。特例化不影响参数匹配。参数匹配都以最佳匹配为原则。例如，此处如果是compare(3,5)，则调用普通的模板，若为compare(“hi”,”haha”)则调用特例化版本（因为这个cosnt char*相对于T，更匹配实参类型），注意二者函数体的语句不一样了，实现不同功能。

「注意」模板及其特例化版本应该声明在同一个头文件中，且所有同名模板的声明应该放在前面，后面放特例化版本。

（2）类模板特例化

原理类似函数模板，不过在类中，我们可以对模板进行特例化，也可以对类进行部分特例化。对类进行特例化时，仍然用template<>表示是一个特例化版本，例如：

```
template<>
class hash<sales_data>
{
    size_t operator()(sales_data& s);
    //里面所有T都换成特例化类型版本sales_data
    //按照最佳匹配原则，若T != sales_data，就用普通类模板，否则，就使用含有特定功能的特例化版本。
};
```

「类模板的部分特例化」

不必为所有模板参数提供实参，可以指定一部分而非所有模板参数，一个类模板的部分特例化本身仍是一个模板，使用它时还必须为其特例化版本中未指定的模板参数提供实参(特例化时类名一定要和原来的模板相同，只是参数类型不同，按最佳匹配原则，哪个最匹配，就用相应的模板)

「特例化类中的部分成员」

可以特例化类中的部分成员函数而不是整个类，举个例子：

```
template<typename T>
class Foo
{
    void Bar();
    void Barst(T a)();
};

template<>
void Foo<int>::Bar()
{
    //进行int类型的特例化处理
    cout << "我是int型特例化" << endl;
}

Foo<string> fs;
Foo<int> fi;//使用特例化
fs.Bar();//使用的是普通模板，即Foo<string>::Bar()
fi.Bar();//特例化版本，执行Foo<int>::Bar()
//Foo<string>::Bar()和Foo<int>::Bar()功能不同
```

23、为什么析构函数一般写成虚函数

由于类的多态性，基类指针可以指向派生类的对象，如果删除该基类的指针，就会调用该指针指向的派生类析构函数，而派生类的析构函数又自动调用基类的析构函数，这样整个派生类的对象完全被释放。如果析构函数不被声明成虚函数，则编译器实施静态绑定，在删除基类指针时，只会调用基类的析构函数而不调用派生类析构函数，这样就会造成派生类对象析构不完全，造成内存泄漏。所以将析构函数声明为虚函数是十分必要的。在实现多态时，当用基类操作派生类，在析构时防止只析构基类而不析构派生类的状况发生，要将基类的析构函数声明为虚函数。举个例子：

```
#include <iostream>
using namespace std;
```

```
class Parent{
public:
    Parent(){
        cout << "Parent construct function"  << endl;
    };
    ~Parent(){
        cout << "Parent destructor function" <<endl;
    }
};

class Son : public Parent{
public:
    Son(){
        cout << "Son construct function"  << endl;
    };
    ~Son(){
        cout << "Son destructor function" <<endl;
    }
};

int main()
{
    Parent* p = new Son();
    delete p;
    p = NULL;
    return 0;
}
//运行结果:
//Parent construct function
//Son construct function
//Parent destructor function
```

将基类的析构函数声明为虚函数:

```
#include <iostream>
using namespace std;

class Parent{
public:
    Parent(){
        cout << "Parent construct function"  << endl;
    };
    virtual ~Parent(){
        cout << "Parent destructor function" <<endl;
    }
};

class Son : public Parent{
```

```
public:
    Son(){
        cout << "Son construct function"  << endl;
    };
    ~Son(){
        cout << "Son destructor function" <<endl;
    }
};

int main()
{
    Parent* p = new Son();
    delete p;
    p = NULL;
    return 0;
}
//运行结果:
//Parent construct function
//Son construct function
//Son destructor function
//Parent destructor function
```

24、拷贝初始化和直接初始化，初始化和赋值的区别?

- ClassTest ct1("ab"); 这条语句属于直接初始化，它不需要调用复制构造函数，直接调用构造函数 ClassTest(constchar *pc)，所以当复制构造函数变为私有时，它还是能直接执行的。
- ClassTest ct2 = "ab"; 这条语句为复制初始化，它首先调用构造函数 ClassTest(const char* pc) 函数创建一个临时对象，然后调用复制构造函数，把这个临时对象作为参数，构造对象ct2；所以当复制构造函数变为私有时，该语句不能编译通过。
- ClassTest ct3 = ct1;这条语句为复制初始化，因为 ct1 本来已经存在，所以不需要调用相关的构造函数，而直接调用复制构造函数，把它值复制给对象 ct3；所以当复制构造函数变为私有时，该语句不能编译通过。
- ClassTest ct4（ct1）;这条语句为直接初始化，因为 ct1 本来已经存在，直接调用复制构造函数，生成对象 ct3 的副本对象 ct4。所以当复制构造函数变为私有时，该语句不能编译通过。

要点就是拷贝初始化和直接初始化调用的构造函数是不一样的，但是当类进行复制时，类会自动生成一个临时的对象，然后再进行拷贝初始化。

C++ STL

> 你现在手里的这份可能不是最新版，因为帅地看到好的面试题，就会整理进去，强烈建议你去看最新版的，微信搜索关注「**帅地玩编程**」，回复「**1023**」，即可获取最新版的 PDF 哦，扫码直达

关注后，回复「**1023**」，即可获取最新版 PDF。

1、什么是STL?

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、什么时候需要用hash_map，什么时候需要用map?

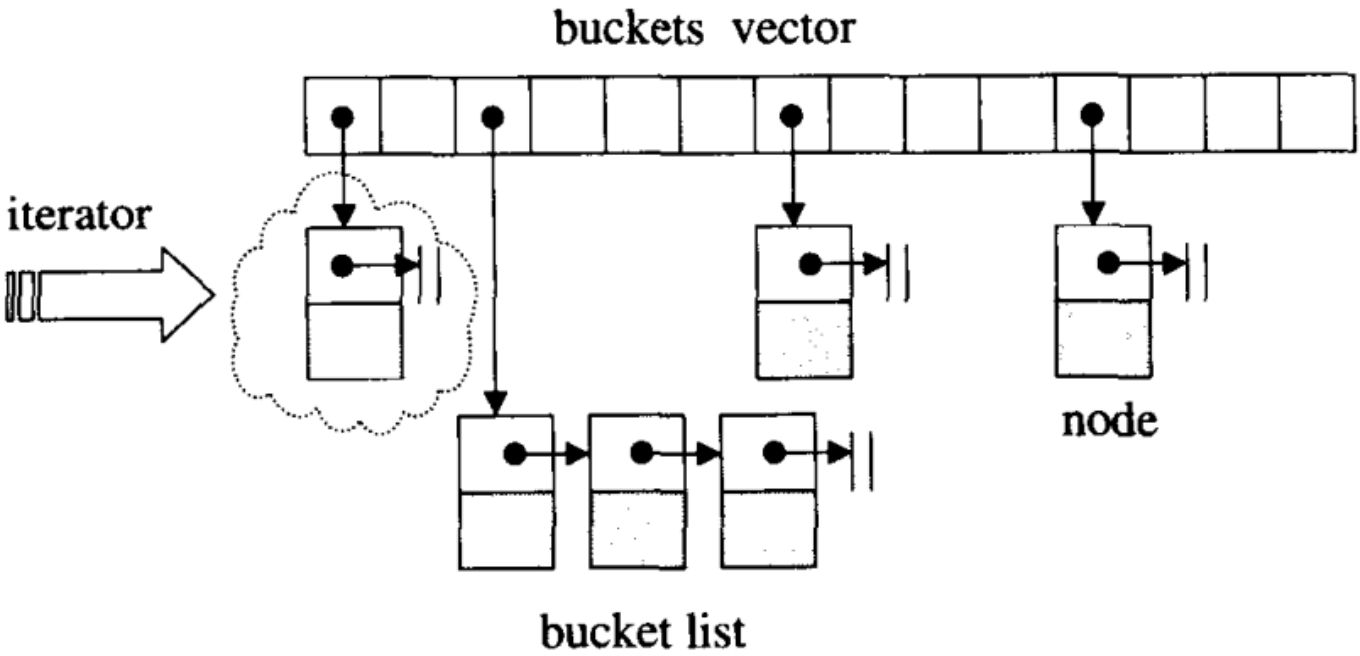
总体来说，hash_map 查找速度会比 map 快，而且查找速度基本和数据数据量大小，属于常数级别;而 map 的查找速度是 log(n) 级别。

并不一定常数就比 log(n) 小，hash 还有 hash 函数的耗时，明白了吧，如果你考虑效率，特别是在元素达到一定数量级时，考虑考虑 hash_map。但若你对内存使用特别严格，希望程序尽可能少消耗内存，那么一定要小心，hash_map 可能会让你陷入尴尬，特别是当你的 hash_map 对象特别多时，你就更无法控制了。而且 hash_map 的构造速度较慢。

现在知道如何选择了吗？权衡三个因素: 查找速度, 数据量, 内存使用 。

3、STL中hashtable的底层实现?

STL中的hashtable使用的是**开链法**解决hash冲突问题，如下图所示。



hashtable中的bucket所维护的list既不是list也不是slist，而是其自己定义的由hashtable_node数据结构组成的linked-list，而bucket聚合体本身使用vector进行存储。hashtable的迭代器只提供前进操作，不提供后退操作

在hashtable设计bucket的数量上，其内置了28个质数[53, 97, 193,...,429496729]，在创建hashtable时，会根据存入的元素个数选择大于等于元素个数的质数作为hashtable的容量（vector的长度），其中每个bucket所维护的linked-list长度也等于hashtable的容量。如果插入hashtable的元素个数超过了bucket的容量，就要进行重建table操作，即找出下一个质数，创建新的buckets vector，重新计算元素在新hashtable的位置。

4、vector 底层原理及其相关面试题

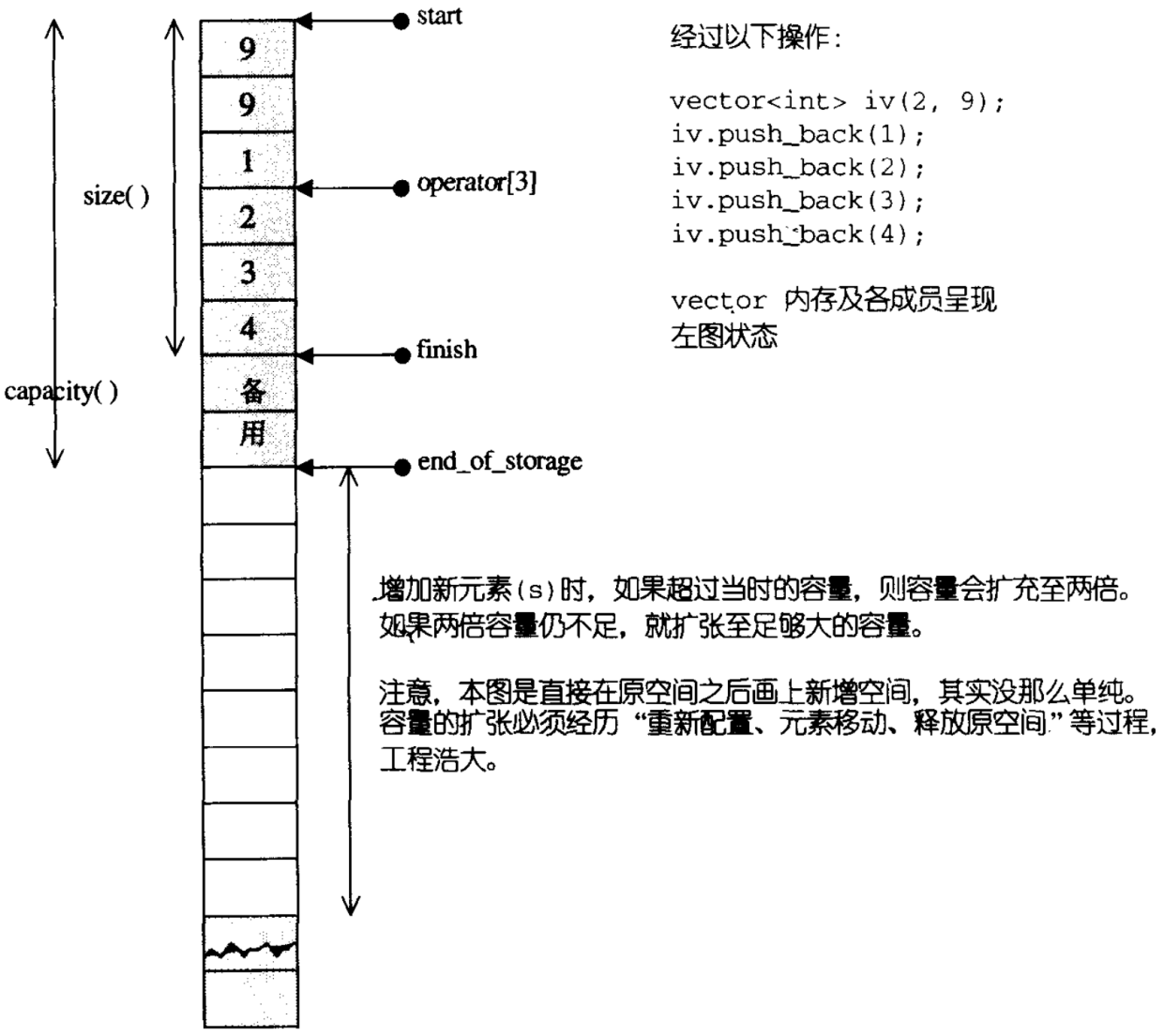
（1）vector的底层原理

vector底层是一个动态数组，包含三个迭代器，start和finish之间是已经被使用的空间范围，end_of_storage是整块连续空间包括备用空间的尾部。

当空间不够装下数据（vec.push_back(val)）时，会自动申请另一片更大的空间（1.5倍或者2倍），然后把原来的数据拷贝到新的内存空间，接着释放原来的那片空间【vector内存增长机制】。

当释放或者删除（vec.clear()）里面的数据时，其存储空间不释放，仅仅是清空了里面的数据。

因此，对vector的任何操作一旦引起了空间的重新配置，指向原vector的所有迭代器会都失效了。



https://blog.csdn.net/WizardtoH

（2）vector中的reserve和resize的区别

reserve是直接扩充到已经确定的大小，可以减少多次开辟、释放空间的问题（优化push_back），就可以提高效率，其次还可以减少多次要拷贝数据的问题。reserve只是保证vector中的空间大小（capacity）最少达到参数所指定的大小n。**reserve()只有一个参数。**

resize()可以改变有效空间的大小，也有改变默认值的功能。capacity的大小也会随着改变。resize()可以有多个参数。

（3）vector中的size和capacity的区别

size表示当前vector中有多少个元素（finish - start），而capacity函数则表示它已经分配的内存中可以容纳多少元素（end_of_storage - start）。

（4）vector的元素类型可以是引用吗?

vector的底层实现要求连续的对象排列，引用并非对象，没有实际地址，因此vector的元素类型不能是引用。

（5）vector迭代器失效的情况

当插入一个元素到vector中，由于引起了内存重新分配，所以指向原内存的迭代器全部失效。

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，那么也造成迭代器失效。erase方法会返回下一个有效的迭代器，所以当我们要删除某个元素时，需要it=vec.erase(it);。

（6）正确释放vector的内存(clear(), swap(), shrink_to_fit())

vec.clear()：清空内容，但是不释放内存。

vector().swap(vec)：清空内容，且释放内存，想得到一个全新的vector。

vec.shrink_to_fit()：请求容器降低其capacity和size匹配。

vec.clear();vec.shrink_to_fit();：清空内容，且释放内存。

（7）vector 扩容为什么要以1.5倍或者2倍扩容?

根据查阅的资料显示，考虑可能产生的堆空间浪费，成倍增长倍数不能太大，使用较为广泛的扩容方式有两种，以2倍的方式扩容，或者以1.5倍的方式扩容。

以2倍的方式扩容，导致下一次申请的内存必然大于之前分配内存的总和，导致之前分配的内存不能再被使用，所以最好倍增长因子设置为(1,2)之间：

$$k\sum_{i=0}^{n}2^{i} = k(2^{n+1}-1) < k2^{n+1}$$

（8）vector的常用函数

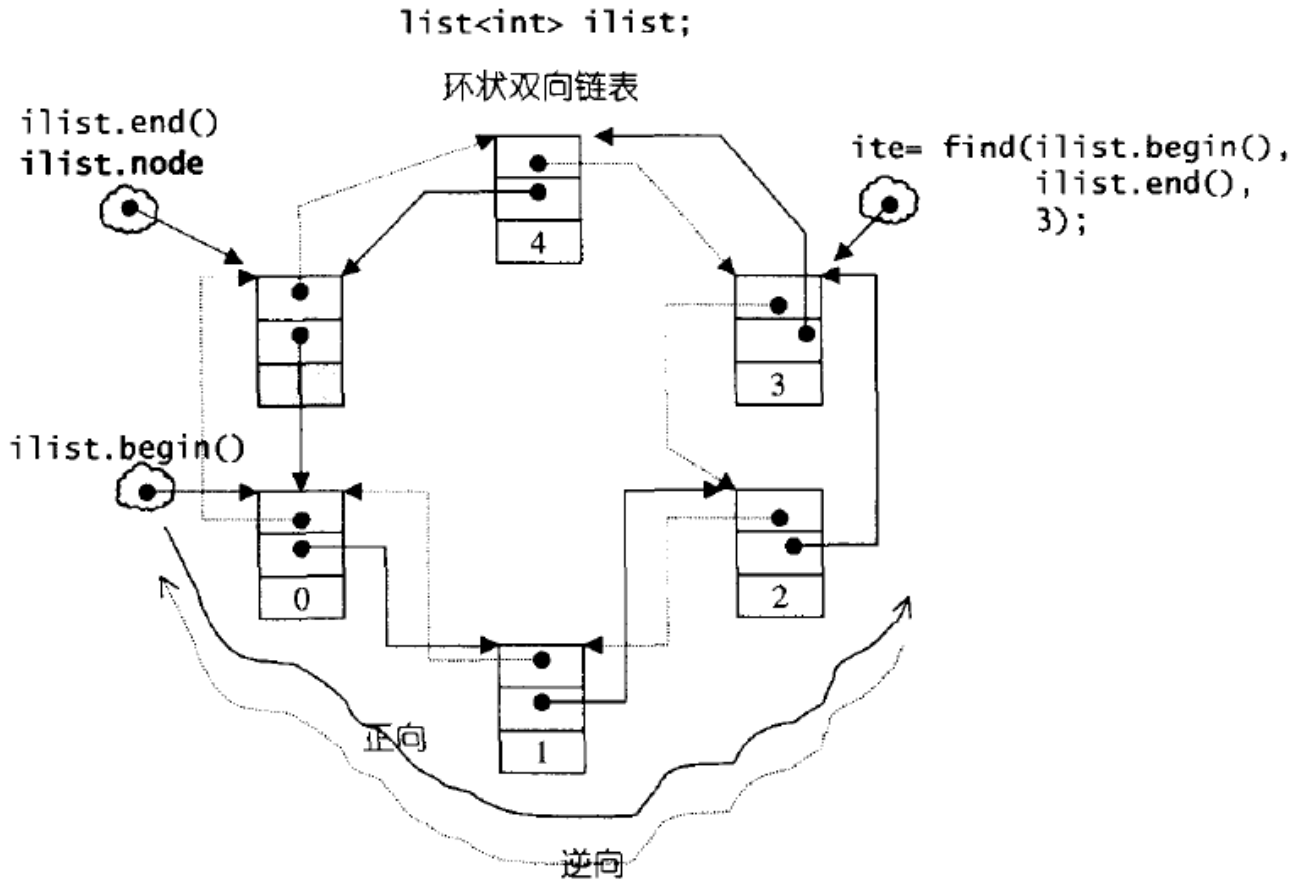
```
vector<int> vec(10,100);          创建10个元素,每个元素值为100
vec.resize(r,vector<int>(c,0));   二维数组初始化
reverse(vec.begin(),vec.end())    将元素翻转
sort(vec.begin(),vec.end());      排序，默认升序排列
vec.push_back(val);               尾部插入数字
vec.size();                       向量大小
find(vec.begin(),vec.end(),1);    查找元素
iterator = vec.erase(iterator)    删除元素
```

5、list 底层原理及其相关面试题

（1）vector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。



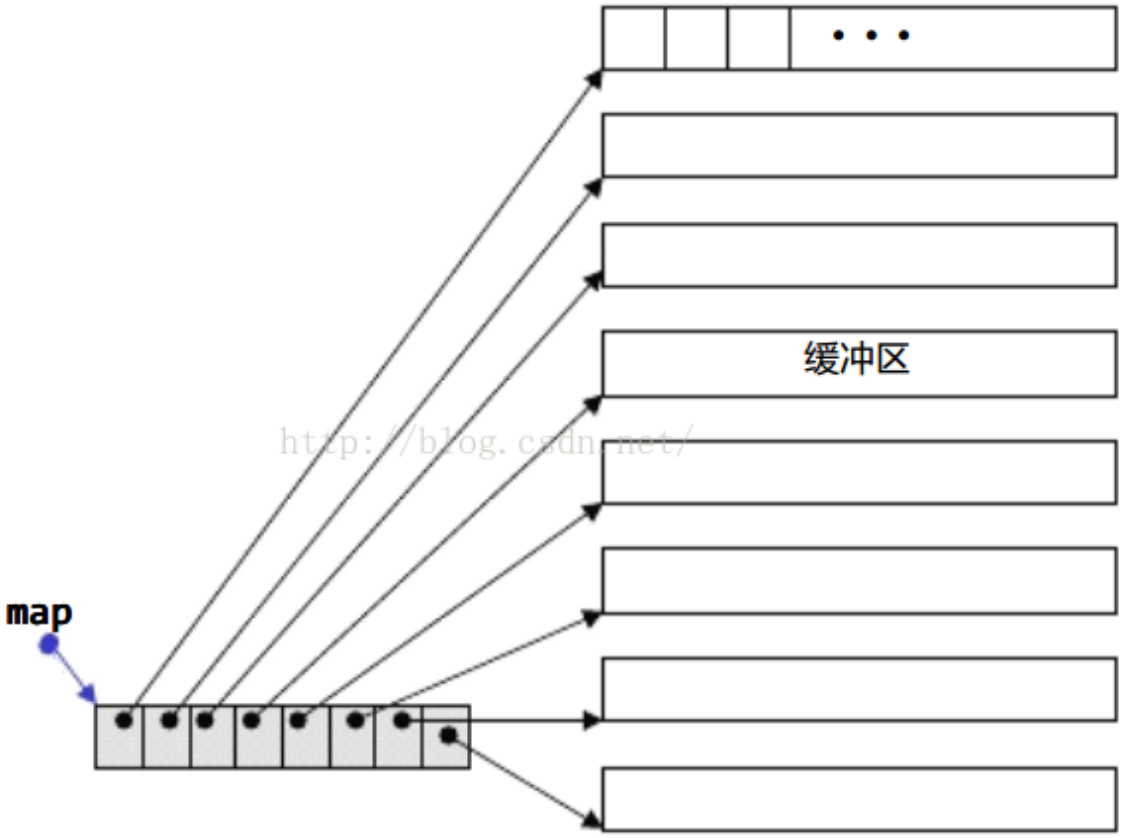
（2）list的常用函数

```
list.push_back(elem)   在尾部加入一个数据
list.pop_back()          删除尾部数据
list.push_front(elem) 在头部插入一个数据
list.pop_front()       删除头部数据
list.size()              返回容器中实际数据的个数
list.sort()              排序，默认由小到大
list.unique()            移除数值相同的连续元素
list.back()              取尾部迭代器
list.erase(iterator)     删除一个元素，参数是迭代器，返回的是删除迭代器的下一个位置
```

6、deque底层原理及其相关面试题

（1）deque的底层原理

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跟删除操作都有理想的时间复杂度。



（2）什么情况下用vector，什么情况下用list，什么情况下用deque

vector可以随机存储元素（即可以通过公式直接计算出元素地址，而不需要挨个查找），但在非尾部插入删除数据时，效率很低，适合对象简单，对象数量变化不大，随机访问频繁。除非必要，我们尽可能选择使用vector而非deque，因为deque的迭代器比vector迭代器复杂很多。

list不支持随机存储，适用于对象大，对象数量变化频繁，插入和删除频繁，比如写多读少的场景。

需要从首尾两端进行插入或删除操作的时候需要选择deque。

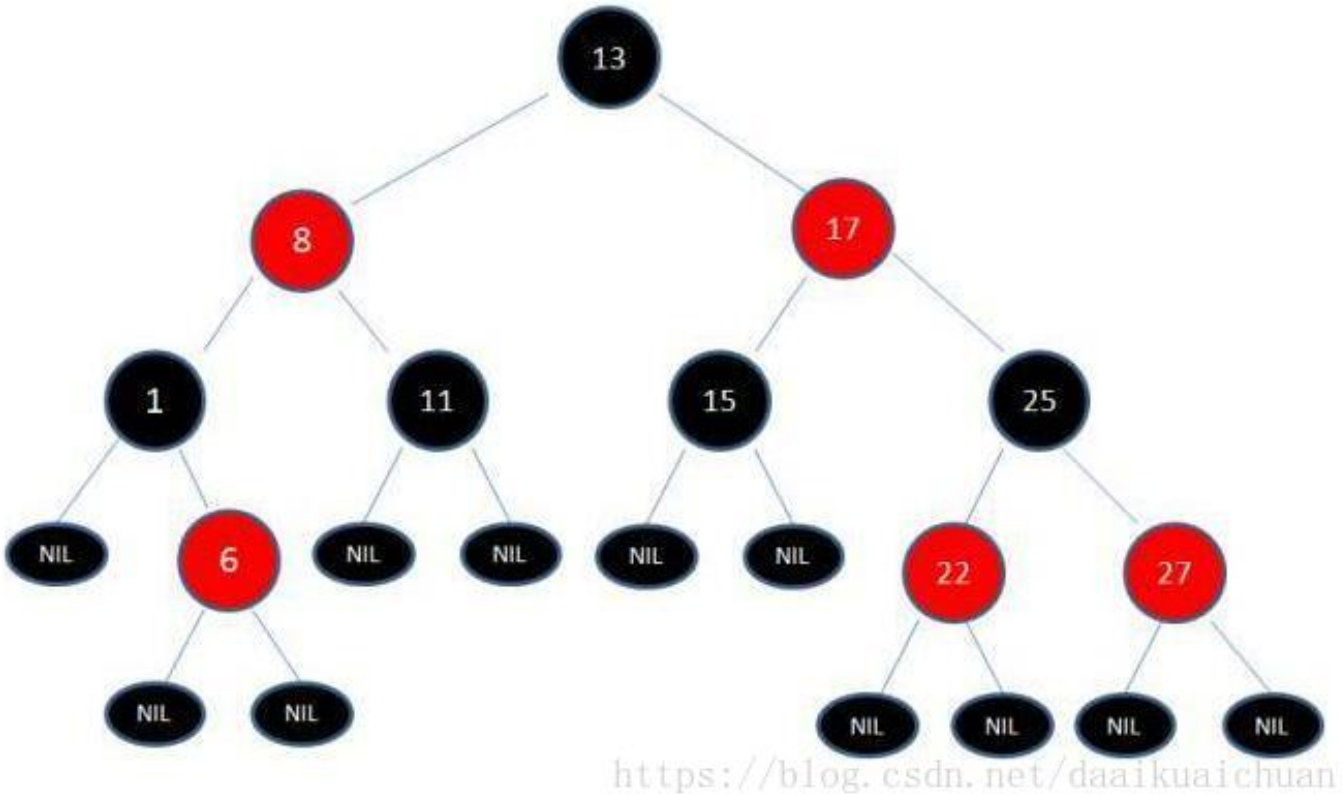
（3）deque的常用函数

```
deque.push_back(elem) 在尾部加入一个数据。
deque.pop_back()      删除尾部数据。
deque.push_front(elem)  在头部插入一个数据。
deque.pop_front()     删除头部数据。
deque.size()          返回容器中实际数据的个数。
deque.at(idx)         传回索引idx所指的数据，如果idx越界，抛出out_of_range。
```

7、map 、set、multiset、multimap 底层原理及其相关面试题

（1）map 、set、multiset、multimap的底层原理

map 、set、multiset、multimap的底层实现都是**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。



红黑树的特性：

1. 每个结点或是红色或是黑色；
2. 根结点是黑色；
3. 每个叶结点是黑的；
4. 如果一个结点是红的，则它的两个儿子均是黑色；
5. 每个结点到其子孙结点的所有路径上包含相同数目的黑色结点。

红黑树详解具体看这篇：[别再问我什么是红黑树了]

对于STL里的map容器，count方法与find方法，都可以用来判断一个key是否出现，mp.count(key) > 0统计的是key出现的次数，因此只能为0/1，而mp.find(key) != mp.end()则表示key存在。

（2）map 、set、multiset、multimap的特点

set和multiset会根据特定的排序准则自动将元素排序，set中元素不允许重复，multiset可以重复。

map和multimap将key和value组成的pair作为元素，根据key的排序准则自动将元素排序（因为红黑树也是二叉搜索树，所以map默认是按key排序的），map中元素的key不允许重复，multimap可以重复。

map和set的增删改查速度为都是logn，是比较高效的。

（3）为何map和set的插入删除效率比其他序列容器高，而且每次insert之后，以前保存的iterator不会失效?

因为存储的是结点，不需要内存拷贝和内存移动。

因为插入操作只是结点指针换来换去，结点内存没有改变。而iterator就像指向结点的指针，内存没变，指向内存的指针也不会变。

（4）为何map和set不能像vector一样有个reserve函数来预分配数据?

因为在map和set内部存储的已经不是元素本身了，而是包含元素的结点。也就是说map内部使用的Alloc并不是map<Key, Data, Compare, Alloc>声明的时候从参数中传入的Alloc。

（5）map 、set、multiset、multimap的常用函数

```
it map.begin()            返回指向容器起始位置的迭代器（iterator）
it map.end()              返回指向容器末尾位置的迭代器
bool map.empty()          若容器为空，则返回true，否则false
it map.find(k)            寻找键值为k的元素，并用返回其地址
int map.size()            返回map中已存在元素的数量
map.insert({int,string}) 插入元素
for (itor = map.begin(); itor != map.end();)
{
    if (itor->second == "target")
        map.erase(itor++) ; // erase之后，令当前迭代器指向其后继。
    else
        ++itor;
}
```

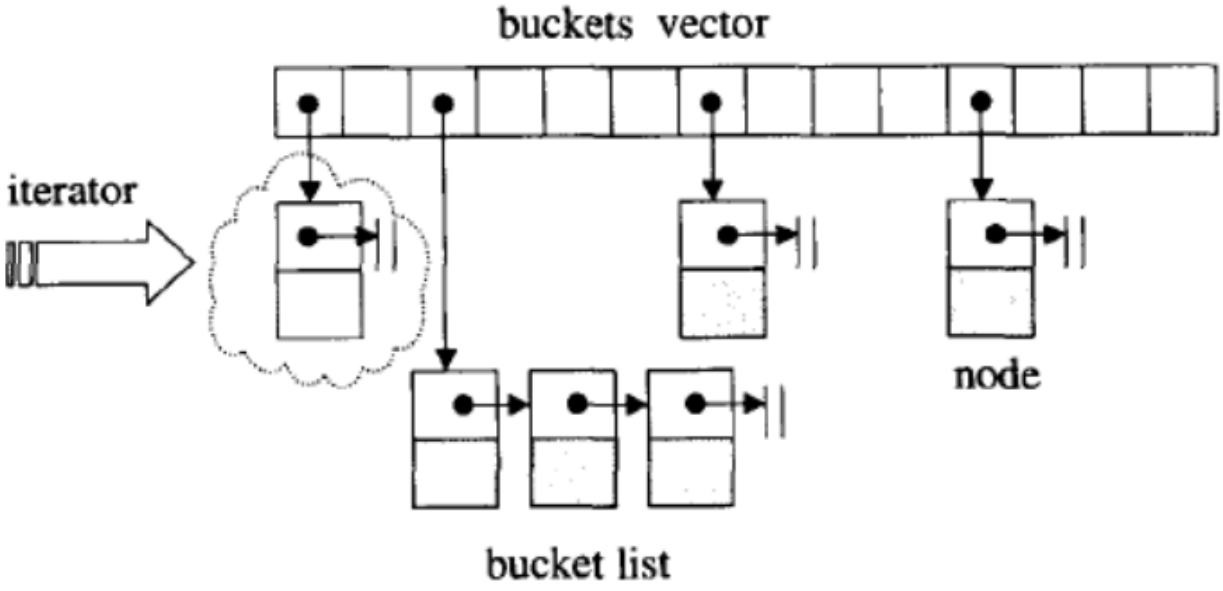
8、unordered_map、unordered_set 底层原理及其相关面试题

（1）unordered_map、unordered_set的底层原理

unordered_map的底层是一个防冗余的哈希表（采用除留余数法）。哈希表最大的优点，就是把数据的存储和查找消耗的时间大大降低，时间复杂度为O(1)；而代价仅仅是消耗比较多的内存。

使用一个下标范围比较大的数组来存储元素。可以设计一个函数（哈希函数（一般使用除留取余法），也叫做散列函数），使得每个元素的key都与一个函数值（即数组下标，hash值）相对应，于是用这个数组单元来存储这个元素；也可以简单的理解为，按照key为每一个元素“分类”，然后将这个元素存储在相应“类”所对应的地方，称为桶。

但是，不能够保证每个元素的key与函数值是一一对应的，因此极有可能出现对于不同的元素，却计算出了相同的函数值，这样就产生了“冲突”，换句话说，就是把不同的元素分在了相同的“类”之中。 一般可采用拉链法解决冲突：



以开链（separate chaining）法完成的 hash table。

（2）哈希表的实现

```
#include <iostream>
#include <vector>
#include <list>
#include <random>
#include <ctime>
using namespace std;

const int hashsize = 12;

//定一个节点的结构体
template <typename T, typename U>
struct HashNode
{
    T _key;
    U _value;
};

//使用拉链法实现哈希表类
template <typename T, typename U>
class HashTable
{
public:
    HashTable() : vec(hashsize) {}//类中的容器需要通过构造函数来指定大小
    ~HashTable() {}
    bool insert_data(const T &key, const U &value);
    int hash(const T &key);
    bool hash_find(const T &key);
private:
```

```
    vector<list<HashNode<T, U>>> vec;//将节点存储到容器中
};

//哈希函数（除留取余）
template <typename T, typename U>
int HashTable<T, U>::hash(const T &key)
{
    return key % 13;
}

//哈希查找
template <typename T, typename U>
bool HashTable<T, U>::hash_find(const T &key)
{
    int index = hash(key);//计算哈希值
    for (auto it = vec[index].begin(); it != vec[index].end(); ++it)
    {
        if (key == it -> _key)//如果找到则打印其关联值
        {
            cout << it->_value << endl;//输出数据前应该确认是否包含相应类型
            return true;
        }
    }
    return false;
}

//插入数据
template <typename T, typename U>
bool HashTable<T, U>::insert_data(const T &key, const U &value)
{
    //初始化数据
    HashNode<T, U> node;
    node._key = key;
    node._value = value;
    for (int i = 0; i < hashsize; ++i)
    {
        if (i == hash(key))//如果溢出则把相应的键值添加进链表
        {
            vec[i].push_back(node);
            return true;
        }
    }
}

int main(int argc, char const *argv[])
{
    HashTable<int, int> ht;
    static default_random_engine e;
    static uniform_int_distribution<unsigned> u(0, 100);
```

```
    long long int a = 10000000;
    for (long long int i = 0; i < a; ++i)
        ht.insert_data(i, u(e));
    clock_t start_time = clock();
    ht.hash_find(114);
    clock_t end_time = clock();
    cout << "Running time is: " << static_cast<double>(end_time - start_time) /
CLOCKS_PER_SEC * 1000 <<
        "ms" << endl;//输出运行时间。
    system("pause");
    system("pause");
    return 0;
}
```

（3）unordered_map 与map的区别？使用场景?

构造函数：unordered_map 需要hash函数，等于函数;map只需要比较函数(小于函数).

存储结构：unordered_map 采用hash表存储，map一般采用红黑树(RB Tree) 实现。因此其memory数据结构是不一样的。

总体来说，unordered_map 查找速度会比map快，而且查找速度基本和数据数据量大小，属于常数级别;而map的查找速度是log(n)级别。并不一定常数就比log(n)小，hash还有hash函数的耗时，明白了吧，如果你考虑效率，特别是在元素达到一定数量级时，考虑考虑unordered_map 。但若你对内存使用特别严格，希望程序尽可能少消耗内存，那么一定要小心，unordered_map 可能会让你陷入尴尬，特别是当你的unordered_map 对象特别多时，你就更无法控制了，而且unordered_map 的构造速度较慢。

（4）unordered_map、unordered_set的常用函数

```
unordered_map.begin()       返回指向容器起始位置的迭代器（iterator）
unordered_map.end()         返回指向容器末尾位置的迭代器
unordered_map.cbegin()      返回指向容器起始位置的常迭代器（const_iterator）
unordered_map.cend()        返回指向容器末尾位置的常迭代器
unordered_map.size()        返回有效元素个数
unordered_map.insert(key)   插入元素
unordered_map.find(key)     查找元素，返回迭代器
unordered_map.count(key)    返回匹配给定主键的元素的个数
```

9、迭代器的底层机制和失效的问题

1、迭代器的底层原理

迭代器是连接容器和算法的一种重要桥梁，通过迭代器可以在不了解容器内部原理的情况下遍历容器。它的底层实现包含两个重要的部分：萃取技术和模板偏特化。

萃取技术（traits）可以进行类型推导，根据不同类型可以执行不同的处理流程，比如容器是vector，那么traits必须推导出其迭代器类型为随机访问迭代器，而list则为双向迭代器。

例如STL算法库中的distance函数，distance函数接受两个迭代器参数，然后计算他们两者之间的距离。显然对于不同的迭代器计算效率差别很大。比如对于vector容器来说，由于内存是连续分配的，因此指针直接相减即可获得两者的距离；而list容器是链式表，内存一般都不是连续分配，因此只能通过一级一级调用next()或其他函数，每调用一次再判断迭代器是否相等来计算距离。vector迭代器计算distance的效率为O(1),而list则为O(n),n为距离的大小。

使用萃取技术（traits）进行类型推导的过程中会使用到模板偏特化。模板偏特化可以用来推导参数，如果我们自定义了多个类型，除非我们把这些自定义类型的特化版本写出来，否则我们只能判断他们是内置类型，并不能判断他们具体属于是个类型。

```
template <typename T>
struct TraitsHelper {
     static const bool isPointer = false;
};
template <typename T>
struct TraitsHelper<T*> {
     static const bool isPointer = true;
};

if (TraitsHelper<T>::isPointer)
     ..... // 可以得出当前类型int*为指针类型
else
     ..... // 可以得出当前类型int非指针类型
```

2、一个理解traits的例子

```
// 需要在T为int类型时，Compute方法的参数为int，返回类型也为int，
// 当T为float时，Compute方法的参数为float，返回类型为int
template <typename T>
class Test {
public:
     TraitsHelper<T>::ret_type Compute(TraitsHelper<T>::par_type d);
private:
     T mData;
};

template <typename T>
struct TraitsHelper {
     typedef T ret_type;
     typedef T par_type;
};

// 模板偏特化，处理int类型
template <>
struct TraitsHelper<int> {
     typedef int ret_type;
     typedef int par_type;
```

```
};

// 模板偏特化，处理float类型
template <>
struct TraitsHelper<float> {
     typedef float ret_type;
     typedef int par_type;
};
```

当函数，类或者一些封装的通用算法中的某些部分会因为数据类型不同而导致处理或逻辑不同时，traits会是一种很好的解决方案。

3、迭代器的种类

输入迭代器：是只读迭代器，在每个被遍历的位置上只能读取一次。例如上面find函数参数就是输入迭代器。

输出迭代器：是只写迭代器，在每个被遍历的位置上只能被写一次。

前向迭代器：兼具输入和输出迭代器的能力，但是它可以对同一个位置重复进行读和写。但它不支持operator–，所以只能向前移动。

双向迭代器：很像前向迭代器，只是它向后移动和向前移动同样容易。

随机访问迭代器：有双向迭代器的所有功能。而且，它还提供了“迭代器算术”，即在一步内可以向前或向后跳跃任意位置， 包含指针的所有操作，可进行随机访问，随意移动指定的步数。支持前面四种Iterator的所有操作，并另外支持it + n、it - n、it += n、 it -= n、it1 - it2和it[n]等操作。

4、迭代器失效的问题

（1）插入操作

对于vector和string，如果容器内存被重新分配，iterators,pointers,references失效；如果没有重新分配，那么插入点之前的iterator有效，插入点之后的iterator失效；

对于deque，如果插入点位于除front和back的其它位置，iterators,pointers,references失效；当我们插入元素到front和back时，deque的迭代器失效，但reference和pointers有效；

对于list和forward_list，所有的iterator,pointer和refercnce有效。

（2）删除操作

对于vector和string，删除点之前的iterators,pointers,references有效；off-the-end迭代器总是失效的；

对于deque，如果删除点位于除front和back的其它位置，iterators,pointers,references失效；当我们插入元素到front和back时，off-the-end失效，其他的iterators,pointers,references有效；

对于list和forward_list，所有的iterator,pointer和refercnce有效。

对于关联容器map来说，如果某一个元素已经被删除，那么其对应的迭代器就失效了，不应该再被使用，否则会导致程序无定义的行为。

10、Vector如何释放空间?

由于vector的内存占用空间只增不减，比如你首先分配了10,000个字节，然后erase掉后面9,999个，留下一个有效元素，但是内存占用仍为10,000个。所有内存空间是在vector析构时候才能被系统回收。empty()用来检测容器是否为空的，clear()可以清空所有元素。但是即使clear()，vector所占用的内存空间依然如故，无法保证内存的回收。

如果需要空间动态缩小，可以考虑使用deque。如果vector，可以用swap()来帮助你释放内存。

```
vector(Vec).swap(Vec); //将Vec的内存清除；  vector().swap(Vec); //清空Vec的内存;
```

11、如何在共享内存上使用STL标准库?

1. 想像一下把STL容器，例如map, vector, list等等，放入共享内存中，IPC一旦有了这些强大的通用数据结构做辅助，无疑进程间通信的能力一下子强大了很多。

我们没必要再为共享内存设计其他额外的数据结构，另外，STL的高度可扩展性将为IPC所驱使。STL容器被良好的封装，默认情况下有它们自己的内存管理方案。

当一个元素被插入到一个STL列表(list)中时，列表容器自动为其分配内存，保存数据。考虑到要将STL容器放到共享内存中，而容器却自己在堆上分配内存。

一个最笨拙的办法是在堆上构造STL容器，然后把容器复制到共享内存，并且确保所有容器的内部分配的内存指向共享内存中的相应区域，这基本是个不可能完成的任务。

2. 假设进程A在共享内存中放入了数个容器，进程B如何找到这些容器呢?

一个方法就是进程A把容器放在共享内存中的确定地址上（fixed offsets），则进程B可以从该已知地址上获取容器。另外一个改进点的办法是，进程A先在共享内存某块确定地址上放置一个map容器，然后进程A再创建其他容器，然后给其取个名字和地址一并保存到这个map容器里。

进程B知道如何获取该保存了地址映射的map容器，然后同样再根据名字取得其他容器的地址。

12、map插入方式有哪几种?

1. 用insert函数插入pair数据，

```
mapStudent.insert(pair<int, string>(1, "student_one")); Copy to clipboardErrorCopied
```

2. 用insert函数插入value_type数据

```
mapStudent.insert(map<int, string>::value_type (1, "student_one"));Copy to
clipboardErrorCopied
```

3. 在insert函数中使用make_pair()函数

```
mapStudent.insert(make_pair(1, "student_one")); Copy to clipboardErrorCopied
```

4. 用数组方式插入数据

```
mapStudent[1] = "student_one"; Copy to clipboard
```

13、为什么vector的插入操作可能会导致迭代器失效?

vector动态增加大小时，并不是在原空间后增加新的空间，而是以原大小的两倍在另外配置一片较大的新空间，然后将内容拷贝过来，并释放原来的空间。由于操作改变了空间，所以迭代器失效。

14、`vector`的`reserve()`和`resize()`方法之间有什么区别?

首先，`vector`的容量`capacity()`是指在不分配更多内存的情况下可以保存的最多元素个数，而`vector`的大小`size()`是指实际包含的元素个数；

其次，`vector`的`reserve(n)`方法只改变`vector`的容量，如果当前容量小于`n`，则重新分配内存空间，调整容量为`n`；如果当前容量大于等于`n`，则无操作；

最后，`vector`的`resize(n)`方法改变`vector`的大小，如果当前容量小于`n`，则调整容量为`n`，同时将其全部元素填充为初始值；如果当前容量大于等于`n`，则不调整容量，只将其前`n`个元素填充为初始值。

15、标准库中有哪些容器？分别有什么特点?

标准库中的容器主要分为三类：顺序容器、关联容器、容器适配器。

- 顺序容器包括五种类型：
  - `array<T, N>`数组：固定大小数组，支持快速随机访问，但不能插入或删除元素；
  - `vector<T>`动态数组：支持快速随机访问，尾位插入和删除的速度很快；
  - `deque<T>`双向队列：支持快速随机访问，首尾位置插入和删除的速度很快；（可以看作是`vector`的增强版，与`vector`相比，可以快速地在首位插入和删除元素）
  - `list<T>`双向链表：只支持双向顺序访问，任何位置插入和删除的速度都很快；
  - `forward_list<T>`单向链表：只支持单向顺序访问，任何位置插入和删除的速度都很快。
- 关联容器包含两种类型：
  - map容器：
    - `map<K, T>`关联数组：用于保存关键字-值对；
    - `multimap<K, T>`：关键字可重复出现的`map`；
    - `unordered_map<K, T>`：用哈希函数组织的`map`；
    - `unordered_multimap<K, T>`：关键词可重复出现的`unordered_map`；
  - set容器：
    - `set<T>`：只保存关键字；
    - `multiset<T>`：关键字可重复出现的`set`；
    - `unordered_set<T>`：用哈希函数组织的`set`；
    - `unordered_multiset<T>`：关键词可重复出现的`unordered_set`；
- 容器适配器包含三种类型：
  - `stack<T>`栈、`queue<T>`队列、`priority_queue<T>`优先队列。

16、容器内部删除一个元素

1. 顺序容器（序列式容器，比如vector、deque）

erase迭代器不仅使所指向被删除的迭代器失效，而且使被删元素之后的所有迭代器失效(list除外)，所以不能使用erase(it++)的方式，但是erase的返回值是下一个有效迭代器；

It = c.erase(it);

2. 关联容器(关联式容器，比如map、set、multimap、multiset等)

erase迭代器只是被删除元素的迭代器失效，但是返回值是void，所以要采用erase(it++)的方式删除迭代器；

c.erase(it++)

17、vector越界访问下标，map越界访问下标？vector删除元素时会不会释放空间?

1. 通过下标访问vector中的元素时会做边界检查，但该处的实现方式要看具体IDE，不同IDE的实现方式不一样，确保不可访问越界地址。
2. map的下标运算符[]的作用是：将key作为下标去执行查找，并返回相应的值；如果不存在这个key，就将一个具有该key和value的某人值插入这个map。
3. erase()函数，只能删除内容，不能改变容量大小;

erase成员函数，它删除了itVect迭代器指向的元素，并且返回要被删除的itVect之后的迭代器，迭代器相当于一个智能指针;clear()函数，只能清空内容，不能改变容量大小;如果要想在删除内容的同时释放内存，那么你可以选择deque容器。

18、map中[ ]与find的区别?

1. map的下标运算符[]的作用是：将关键码作为下标去执行查找，并返回对应的值；如果不存在这个关键码，就将一个具有该关键码和值类型的默认值的项插入这个map。
2. map的find函数：用关键码执行查找，找到了返回该位置的迭代器；如果不存在这个关键码，就返回尾迭代器。

19、STL内存优化?

STL内存管理使用二级内存配置器。

(1) 第一级配置器：

第一级配置器以malloc()，free()，realloc()等C函数执行实际的内存配置、释放、重新配置等操作，并且能在内存需求不被满足的时候，调用一个指定的函数。一级空间配置器分配的是大于128字节的空间，如果分配不成功，调用句柄释放一部分内存，如果还不能分配成功，抛出异常。

第一级配置器只是对malloc函数和free函数的简单封装，在allocate内调用malloc，在deallocate内调用free。同时第一级配置器的oom_malloc函数，用来处理malloc失败的情况。

(2) 第二级配置器：

第一级配置器直接调用malloc和free带来了几个问题：

- 内存分配/释放的效率低
- 当配置大量的小内存块时，会导致内存碎片比较严重
- 配置内存时，需要额外的部分空间存储内存块信息，所以配置大量的小内存块时，还会导致额外内存负担

如果分配的区块小于128bytes，则以内存池管理，第二级配置器维护了一个自由链表数组，每次需要分配内存时，直接从相应的链表上取出一个内存节点就完成工作，效率很高

自由链表数组：自由链表数组其实就是个指针数组，数组中的每个指针元素指向一个链表的起始节点。数组大小为16，即维护了16个链表，链表的每个节点就是实际的内存块，相同链表上的内存块大小都相同，不同链表的内存块大小不同，从8一直到128。如下所示，obj为链表上的节点，free_list就是链表数组。

内存分配：allocate函数内先判断要分配的内存大小，若大于128字节，直接调用第一级配置器，否则根据要分配的内存大小从16个链表中选出一个链表，取出该链表的第一个节点。若相应的链表为空，则调用refill函数填充该链表。默认是取出20个数据块。

填充链表 refill：若allocate函数内要取出节点的链表为空，则会调用refill函数填充该链表。refill函数内会先调用chunk_alloc函数从内存池分配一大块内存，该内存大小默认为20个链表节点大小，当内存池的内存也不足时，返回的内存块节点数目会不足20个。接着refill的工作就是将这一大块内存分成20份相同大小的内存块，并将各内存块连接起来形成一个链表。

内存池：chunk_alloc函数内管理了一块内存池，当refill函数要填充链表时，就会调用chunk_alloc函数，从内存池取出相应的内存。

- 在chunk_alloc函数内首先判断内存池大小是否足够填充一个有20个节点的链表，若内存池足够大，则直接返回20个内存节点大小的内存块给refill；
- 若内存池大小无法满足20个内存节点的大小，但至少满足1个内存节点，则直接返回相应的内存节点大小的内存块给refill；
- 若内存池连1个内存节点大小的内存块都无法提供，则chunk_alloc函数会将内存池中那一点点的内存大小分配给其他合适的链表，然后去调用malloc函数分配的内存大小为所需的两倍。若malloc成功，则返回相应的内存大小给refill；若malloc失败，会先搜寻其他链表的可用的内存块，添加到内存池，然后递归调用chunk_alloc函数来分配内存，若其他链表也无内存块可用，则只能调用第一级空间配置器。

20、频繁对vector调用push_back()对性能的影响和原因?

在一个vector的尾部之外的任何位置添加元素，都需要重新移动元素。而且，向一个vector添加元素可能引起整个对象存储空间的重新分配。重新分配一个对象的存储空间需要分配新的内存，并将元素从旧的空间移到新的空间。

C++内存管理

> 你现在手里的这份可能不是最新版，因为帅地看到好的面试题，就会整理进去，强烈建议你去看最新版的，微信搜索关注「**帅地玩编程**」，回复「**1023**」，即可获取最新版的 PDF 哦，扫码直达

关注后，回复「**1023**」，即可获取最新版 PDF。

1、new/delete和malloc/free之间有什么关系?

```
 int *p = new int[2];
int *q = (int *)malloc(2*sizeof(int));
```

- new与delete直接带具体类型的指针，malloc和free返回void类型的指针。
- new类型是安全的，而malloc不是。例如int *p = new float[2];就会报错；而int p = malloc(2sizeof(int))编译时编译器就无法指出错误来。
- new一般分为两步：new操作和构造。new操作对应与malloc，但new操作可以重载，可以自定义内存分配策略，不做内存分配，甚至分配到非内存设备上，而malloc不行。
- new调用构造函数，malloc不能；delete调用析构函数，而free不能。
- malloc/free需要库文件stdlib.h的支持，new/delete则不需要!

「注意」：delete和free被调用后，内存不会立即回收，指针也不会指向空，delete或free仅仅是告诉操作系统，这一块内存被释放了，可以用作其他用途。但是由于没有重新对这块内存进行写操作，所以内存中的变量数值并没有发生变化，出现野指针的情况。因此，释放完内存后，应该讲该指针指向NULL。

2、delete与delete []有什么区别?

- 对于简单类型来说，使用`new`分配后，不管是数组数组还是非数组形式，两种方式都可以释放内存：

```
int *a = new int(1);
delete a;
int *b = new int(2);
delete [] b;
int *c = new int[11];
delete c;
int *d = new int[12];
delete [] d;
```

- 对于自定义类型来说，就需要对于单个对象使用`delete`，对于对象数组使用`delete []`，逐个调用数组中对象的析构函数，从而释放所有内存；

  如果反过来使用，即对于单个对象使用`delete []`，对于对象数组使用`delete`，其行为是未定义的；

- 所以，最恰当的方式就是如果用了`new`，就用`delete`；如果用了`new []`，就用`delete []`。

3、内存块太小导致`malloc`和`new`返回空指针，该怎么处理?

- 对于`malloc`来说，需要判断其是否返回空指针，如果是则马上用`return`语句终止该函数或者`exit`终止该程序；
- 对于`new`来说，默认抛出异常，所以可以使用`try...catch...`代码块的方式：

```
try {
    int *ptr = new int[10000000];
} catch(bad_alloc &memExp) {
    cerr << memExp.what() << endl;
}
```

还可以使用`set_new_handler`函数的方式：

```
void no_more_memory() {
    cerr << "Unable to satisfy request for memory" << endl;
    abort();
}
int main() {
    set_new_handler(no_more_memory);
    int *ptr = new int[10000000];
}
```

在这种方式里，如果`new`不能满足内存分配请求，`no_more_memory`会被反复调用，所以`new_handler`函数必须完成以下事情：

- 让更多内存可被使用：可以在程序一开始执行就分配一大块内存，之后当`new_handler`第一次被调用，就将这些内存释放还给程序使用；
- 使用另一个`new_handler`；
- 卸除`new_handler`：返回空指针，这样`new`就会抛出异常；
- 直接抛出`bad_alloc`异常；
- 调用`abort`或`exit`。

4、内存泄漏的场景有哪些？如何判断内存泄漏？如何定位内存泄漏?

内存泄漏的场景：

- `malloc`和`free`未成对出现；`new/new []`和`delete/delete []`未成对出现；
  - 在堆中创建对象分配内存，但未显式释放内存；比如，通过局部分配的内存，未在调用者函数体内释放：

```
char* getMemory() {
    char *p = (char *)malloc(30);
    return p;
}
int main() {
    char *p = getMemory();
    return 0;
}
```

    - 在构造函数中动态分配内存，但未在析构函数中正确释放内存；
- 未定义拷贝构造函数或未重载赋值运算符，从而造成两次释放相同内存的做法；比如，类中包含指针成员变量，在未定义拷贝构造函数或未重载赋值运算符的情况下，编译器会调用默认的拷贝构造函数或赋值运算符，以逐个成员拷贝的方式来复制指针成员变量，使得两个对象包含指向同一内存空间的指针，那么在释放第一个对象时，析构函数释放该指针指向的内存空间，在释放第二个对象时，析构函数就会释放同一内存空间，这样的行为是错误的；
- 没有将基类的析构函数定义为虚函数。

判断和定位内存泄漏的方法：在Linux系统下，可以使用valgrind、mtrace等内存泄漏检测工具。

5、内存的分配方式有几种?

- 在栈上分配：在执行函数时，局部变量的内存都可以在栈上分配，函数结束时会自动释放；栈内存的分配运算内置于处理器的指令集中，效率很高，但分配的内存容量有限；
- 从堆上分配：由`new`分配/`delete`释放的内存块，也称为动态内存分配，程序员自行申请和释放内存，使用灵活；
- 从自由存储区分配：由`malloc`分配/`free`释放的内存块，与堆类似；
- 从常量存储区分配：特殊的存储区，存放的是常量，不可修改；
- 从全局/静态存储区分配：编译期间分配内存，整个程序运行期间都存在，如全局变量、静态变量等。

6、堆和栈有什么区别?

- 分配和管理方式不同：
    - 堆是动态分配的，其空间的分配和释放都由程序员控制；
    - 栈是由编译器自动管理的，其分配方式有两种：静态分配由编译器完成，比如局部变量的分配；动态分配由`alloca()`函数进行分配，但是会由编译器释放；
- 产生碎片不同：
    - 对堆来说，频繁使用`new/delete`或者`malloc/free`会造成内存空间的不连续，产生大量碎片，是程序效率降低；
    - 对栈来说，不存在碎片问题，因为栈具有先进后出的特性；
- 生长方向不同：
    - 堆是向着内存地址增加的方向增长的，从内存的低地址向高地址方向增长；
    - 栈是向着内存地址减小的方向增长的，从内存的高地址向低地址方向增长；
- 申请大小限制不同：
    - 栈顶和栈底是预设好的，大小固定；
    - 堆是不连续的内存区域，其大小可以灵活调整

7、静态内存分配和动态内存分配有什么区别?

- 静态内存分配是在编译时期完成的，不占用CPU资源；动态内存分配是在运行时期完成的，分配和释放需要占用CPU资源；
- 静态内存分配是在栈上分配的；动态内存分配是在堆上分配的；
- 静态内存分配不需要指针或引用类型的支持；动态内存分配需要；
- 静态内存分配是按计划分配的，在编译前确定内存块的大小；动态内存分配是按需要分配的；
- 静态内存分配是把内存的控制权交给了编译器；动态内存分配是把内存的控制权给了程序员；
- 静态内存分配的运行效率比动态内存分配高，动态内存分配不当可能造成内存泄漏。

8、如何构造一个类，使得只能在堆上或只能在栈上分配内存?

- 只能在堆上分配内存：将析构函数声明为`private`；
- 只能在栈上生成对象：将`new`和`delete`重载为`private`。

9、浅拷贝和深拷贝有什么区别?

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。

10、字节对齐的原则是什么?

- 从偏移为0的位置开始存储；
- 如果没有定义`#pragma pack(n)`
  - `sizeof`的最终结果必然是结构内部最大成员的整数倍，不够补齐；
  - 结构内部各个成员的首地址必然是自身大小的整数倍；
- 如果定义了`#pragma pack(n)`
  - `sizeof`的最终结果必然必然是`min[n,结构内部最大成员]`的整数倍，不够补齐；
  - 结构内部各个成员的首地址必然是`min[n,自身大小]`的整数倍。

11、结构体内存对齐问题

请写出以下代码的输出结果：

```
#include<stdio.h>
struct S1
{
    int i:8;
    char j:4;
    int a:4;
    double b;
};

struct S2
{
    int i:8;
    char j:4;
```

```
    double b;
    int a:4;
};

struct S3
{
    int i;
    char j;
    double b;
    int a;
};

int main()
{
    printf("%d\n",sizeof(S1));  // 输出8
    printf("%d\n",sizeof(S1);  // 输出12
    printf("%d\n",sizeof(Test3));  // 输出8
    return 0;
}

sizeof(S1)=16
sizeof(S2)=24
sizeof(S3)=32
```

「说明」：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类型数据。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。

许多实际的计算机系统对基本类型数据在内存中存放的位置有限制，它们会要求这些数据的首地址的值是某个数k（通常它为4或8）的倍数，而这个k则被称为该数据类型的对齐模数。

12、在C++中，使用malloc申请的内存能否通过delete释放？使用new申请的内存能否用free?

不能，malloc /free主要为了兼容C，new和delete 完全可以取代malloc /free的。malloc /free的操作对象都是必须明确大小的。而且不能用在动态类上。new 和delete会自动进行类型检查和大小，malloc/free不能执行构造函数与析构函数，所以动态对象它是不行的。当然从理论上说使用malloc申请的内存是可以通过delete释放的。不过一般不这样写的。而且也不能保证每个C++的运行时都能正常。

网络编程

> 你现在手里的这份可能不是最新版，因为帅地看到好的面试题，就会整理进去，强烈建议你去看最新版的，微信搜索关注「**帅地玩编程**」，回复「**1023**」，即可获取最新版的 PDF 哦，扫码直达

关注后，回复「**1023**」，即可获取最新版 PDF。

史上最全，不接受反驳！！！！！！！

1、什么是IO多路复用

I/O多路复用的本质是使用select,poll或者epoll函数，挂起进程，当一个或者多个I/O事件发生之后，将控制返回给用户进程。以服务器编程为例，传统的多进程(多线程)并发模型，在处理用户连接时都是开启一个新的线程或者进程去处理一个新的连接，而I/O多路复用则可以在一个进程(线程)当中同时监听多个网络I/O事件，也就是多个文件描述符。select、poll 和 epoll 都是 Linux API 提供的 IO 复用方式。

2、epool中et和lt的区别与实现原理

LT：水平触发，效率会低于ET触发，尤其在大并发，大流量的情况下。但是LT对代码编写要求比较低，不容易出现问题。LT模式服务编写上的表现是：只要有数据没有被获取，内核就不断通知你，因此不用担心事件丢失的情况。

ET：边缘触发，效率非常高，在并发，大流量的情况下，会比LT少很多epoll的系统调用，因此效率高。但是对编程要求高，需要细致的处理每个请求，否则容易发生丢失事件的情况。

3、tcp连接建立的时候3次握手，断开连接的4次握手的具体过程

三次握手 --- 第一次握手是客户端connect连接到server，server accept client的请求之后，向client端发送一个消息，相当于说我都准备好了，你连接上我了，这是第二次握手，第3次握手就是client向server发送的，就是对第二次握手消息的确认。之后client和server就开始通讯了。

四次握手 --- 断开连接的一端发送close请求是第一次握手，另外一端接收到断开连接的请求之后需要对close进行确认，发送一个消息，这是第二次握手，发送了确认消息之后还要向对端发送close消息，要关闭对对端的连接，这是第3次握手，而在最初发送断开连接的一端接收到消息之后，进入到一个很重要的状态time_wait状态，这个状态也是面试官经常问道的问题，最后一次握手是最初发送断开连接的一端接收到消息之后。对消息的确认。

4、connect方法会阻塞，请问有什么方法可以避免其长时间阻塞?

最通常的方法最有效的是加定时器；也可以采用非阻塞模式。

或者考虑采用异步传输机制，同步传输与异步传输的主要区别在于同步传输中，如果调用recvfrom后会一致阻塞运行，从而导致调用线程暂停运行；异步传输机制则不然，会立即返回。

5、网络中，如果客户端突然掉线或者重启，服务器端怎么样才能立刻知道?

若客户端掉线或者重新启动，服务器端会收到复位信号，每一种tcp/ip得实现不一样，控制机制也不一样。

6、在子网210.27.48.21/30种有多少个可用地址？分别是什么?

简: 30表示的是网络号(network number)是30位，剩下2位中11是广播(broadcast)地址，00是multicast地址，只有01和10可以作为host address。

详: 210.27.48.21/30代表的子网的网络号是30位，即网络号是210.27.48.21 & 255.255.255.251=210.27.48.20，此子网的地址空间是2位，即可以有4个地址：210.27.48.20, 210.27.48.21, 210.27.48.22, 210.27.48.23。第一个地址的主机号(host number/id)是0，而主机号0代表的是multicast地址。

最后一个地址的最后两位是11，主机号每一位都为1代表的是广播(broadcast)地址。所以只有中间两个地址可以给host使用。

其实那个问题本身不准确，广播或multicast地止也是可以使用的地址，所以回答4也应该正确，当然问的人也可能是想要你回答2。我个人觉得最好的回答是一个广播地址，一个multicast地址，2个unicast地址。

7、TTL是什么？有什么用处，通常那些工具会用到它？（ping? traceroute? ifconfig? netstat?）

简: TTL是Time To Live，一般是hup count，每经过一个路由就会被减去一，如果它变成0，包会被丢掉。它的主要目的是防止包在有回路的网络上死转，浪费网络资源。ping和traceroute用到它。

详: TTL是Time To Live，目前是hup count，当包每经过一个路由器它就会被减去一，如果它变成0，路由器就会把包丢掉。IP网络往往带有环(loop)，比如子网A和子网B有两个路由器相连，它就是一个loop。TTL的主要目的是防止包在有回路的网络上死转，因为包的TTL最终后变成0而使得此包从网上消失(此时往往路由器会送一个ICMP包回来，traceroute就是根据这个做的)。ping会送包出去，所以里面有它，但是ping不一定非要不可它。traceroute则是完全因为有它才能成的。ifconfig是用来配置网卡的，netstat -rn 是用来列路由表的，所以都用不着它

8、路由表示做什么用的？在linux环境中怎么来配置一条默认路由?

简: 路由表是用来决定如何将包从一个子网传送到另一个子网的，换局话说就是用来决定从一个网卡接收到的包应该送的哪一张网卡上的。在Linux上可以用“route add default gw <默认路由器IP>”来配置一条默认路由。

详: 路由表是用来决定如何将包从一个子网传送到另一个子网的，换局话说就是用来决定从一个网卡接收到的包应该送的哪一张网卡上的。路由表的每一行至少有目标网络号、netmask、到这个子网应该使用的网卡。当路由器从一个网卡接收到一个包时，它扫描路由表的每一行，用里面的netmask和包里的目标IP地址做并逻辑运算(&)找出目标网络号，如果此网络号和这一行里的网络号相同就将这条路由保留下来做为备用路由，如果已经有备用路由了就在这两条路由里将网络号最长的留下来，另一条丢掉，如此接着扫描下一行直到结束。如果扫描结束任没有找到任何路由，就用默认路由。确定路由后，直接将包送到对应的网卡上去。在具体的实现中，路由表可能包含更多的信

息为选路由算法的细节所用。

题外话：路由算法其实效率很差，而且不scalable，解决办法是使用IP交换机，比如MPLS。

在Linux上可以用“route add default gw <默认路由器IP>”来配置一条默认路由。

9、在网络中有两台主机A和B，并通过路由器和其他交换设备连接起来，已经确认物理连接正确无误，怎么来测试这两台机器是否连通？如果不通，怎么来判断故障点？怎么排除故障？

测试这两台机器是否连通：从一台机器ping另一台机器，如果ping不通，用traceroute可以确定是哪个路由器不能连通，然后再找问题是在交换设备/hup/cable等。

10、网络编程中设计并发服务器，使用多进程 与 多线程 ，请问有什么区别？

答案一：

1）进程：子进程是父进程的复制品。子进程获得父进程数据空间、堆和栈的复制品。

2）线程：相对与进程而言，线程是一个更加接近与执行体的概念，它可以与同进程的其他线程共享数据，但拥有自己的栈空间，拥有独立的执行序列。

两者都可以提高程序的并发度，提高程序运行效率和响应时间。

线程和进程在使用上各有优缺点：线程执行开销小，但不利于资源管理和保护；而进程正相反。同时，线程适合于在SMP机器上运行，而进程则可以跨机器迁移。

答案二：

根本区别就一点：用多进程每个进程有自己的地址空间(address space)，线程则共享地址空间。所有其它区别都是由此而来的：

1）速度：线程产生的速度快，线程间的通讯快、切换快等，因为他们在同一个地址空间内。 2）资源利用率：线程的资源利用率比较好也是因为他们在同一个地址空间内。 3）同步问题：线程使用公共变量/内存时需要使用同步机制还是因为他们在同一个地址空间内。

11、网络编程的一般步骤

对于TCP连接：

1.服务器端1）创建套接字create；2）绑定端口号bind；3）监听连接listen；4）接受连接请求accept，并返回新的套接字；5）用新返回的套接字recv/send；6）关闭套接字。

2.客户端1）创建套接字create; 2）发起建立连接请求connect; 3）发送/接收数据send/recv；4）关闭套接字。

TCP总结：

Server端：create -- bind -- listen-- accept-- recv/send-- close Client端：create------- conncet------send/recv------close.

对于UDP连接：

1.服务器端:1）创建套接字create；2）绑定端口号bind；3）接收/发送消息recvfrom/sendto；4）关闭套接字。
2.客户端:1）创建套接字create；2）发送/接收消息sendto/recvfrom；3）关闭套接字.

UDP总结:

Server端：create----bind ----recvfrom/sendto----close Client端：create---- sendto/recvfrom----close.

12、TCP的重发机制是怎么实现的?

1）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）。

2）选择重传，用于对传输出错的序列进行重传。

13、TCP为什么不是两次连接？而是三次握手?

如果A与B两个进程通信，如果仅是两次连接。可能出现的一种情况就是：A发送完请报文以后，由于网络情况不好，出现了网络拥塞，即B延时很长时间后收到报文，即此时A将此报文认定为失效的报文。

B收到报文后，会向A发起连接。此时两次握手完毕，B会认为已经建立了连接可以通信，B会一直等到A发送的连接请求，而A对失效的报文回复自然不会处理。依次会陷入B忙等的僵局，造成资源的浪费。

14、socket编程，如果client断电了，服务器如何快速知道?

使用定时器（适合有数据流动的情况）； 使用socket选项SO_KEEPALIVE（适合没有数据流动的情况）；

15、fork()一子进程程后 父进程乩全局变量能不能使用?

fork后子进程将会拥有父进程的几乎一切资源，父子进程的都各自有自己的全局变量。不能通用，不同于线程。对于线程，各个线程共享全局变量。

16、4G的long型整数中找到一个最大的，如何做?

要找到最大的肯定要遍历所有的数的，而且不能将数据全部读入内存，可能不足。算法的时间复杂度肯定是O（n） 感觉就是遍历，比较。。。。还能怎么改进呢？？？？

可以改进的地方，就是读入内存的时候，一次多读些。。。。

需要注意的就是每次从磁盘上尽量多读一些数到内存区，然后处理完之后再读入一批。减少IO次数，自然能够提高效率。

而对于类快速排序方法，稍微要麻烦一些： 分批读入，假设是M个数，然后从这M个数中选出n个最大的数缓存起来，直到所有的N个数都分批处理完之后，再将各批次缓存的n个数合并起来再进行一次类快 速排序得到最终的n个最大的数就可以了。

在运行过程中，如果缓存数太多，可以不断地将多个缓存合并，保留这些缓存中最大的n个数即可。由于类快速排序的时 间复杂度是O（N），这样分批处理再合并的办法，依然有极大的可能会比堆和败者树更优。当然，在空间上会占用较多的内存。

此题还有个变种，就是寻找K个最大或者最小的数。有以下几种算法：

容量为K的最大堆/最小堆，假设K可以装入内存；

如果N个数可以装入内存，且都小于MAX，那么可以开辟一个MAX大的数组，类似计数排序。。。从数组尾部扫描K个最大的数，头部扫描K个最小的数。

17、tcp三次握手的过程，accept发生在三次握手哪个阶段?

client 的 connect 引起3次握手

server 在socket， bind， listen后，阻塞在accept，三次握手完成后，accept返回一个fd，因此accept发生在三次握手之后。

18、tcp流， udp的数据报，之间有什么区别，为什么TCP要叫做数据流?

TCP本身是面向连接的协议，S和C之间要使用TCP，必须先建立连接，数据就在该连接上流动，可以是双向的，没有边界。所以叫数据流 ，占系统资源多

UDP不是面向连接的，不存在建立连接，释放连接，每个数据包都是独立的包，有边界，一般不会合并。

TCP保证数据正确性，UDP可能丢包，TCP保证数据顺序，UDP不保证

19、socket在什么情况下可读?

1. 接收缓冲区有数据，一定可读
2. 对方正常关闭socket，也是可读
3. 对于侦听socket，有新连接到达也可读 4.socket有错误发生，且pending

20、TCP通讯中，select到读事件，但是读到的数据量是0，为什么，如何解决?

select 返回0代表超时。select出错返回-1。

select到读事件，但是读到的数据量为0，说明对方已经关闭了socket的读端。本端关闭读即可。

当select出错时，会将接口置为可读又可写。这时就要通过判断select的返回值为-1来区分。

21、说说IO多路复用优缺点?

IO多路复用优点：

1.相比基于进程的模型给程序员更多的程序行为控制。 2.IO多路复用只需要一个进程就可以处理多个事件，单个进程内数据共享变得容易，调试也更容易。 3.因为在单一的进程上下文当中，所以不会有多进程多线程模型的切换开销。

IO多路复用缺点：

1.业务逻辑处理困难，编程思维不符合人类正常思维。 2.不能充分利用多核处理器。

22、说说select机制的缺点

每次调用select，都需要把监听的文件描述符集合fd_set从用户态拷贝到内核态，从算法角度来说就是O(n)的时间开销。

每次调用select调用返回之后都需要遍历所有文件描述符，判断哪些文件描述符有读写事件发生，这也是O(n)的时间开销。

内核对被监控的文件描述符集合大小做了限制，并且这个是通过宏控制的，大小不可改变(限制为1024)。

23、说一下epoll的好处

epoll解决了select和poll在文件描述符集合拷贝和遍历上的问题，能够在一个进程当中监听多个文件描述符，并且十分高效。

24、epoll需要在用户态和内核态拷贝数据么?

在注册监听事件时从用户态将数据传入内核态；当返回时需要将就绪队列的内容拷贝到用户空间。

25、epoll的实现知道么？在内核当中是什么样的数据结构进行存储，每个操作的时间复杂度是多少?

在内核当中是以红黑树的方式组织监听的事件，查询开销是O(logn)。采用回调的方式检测就绪事件，时间复杂的位O(1);

计算机网络

> 你现在手里的这份可能不是最新版，因为帅地看到好的面试题，就会整理进去，强烈建议你去看最新版的，微信搜索关注「**帅地玩编程**」，回复「**1023**」，即可获取最新版的 PDF 哦，扫码直达

关注后，回复「**1023**」，即可获取最新版 PDF。

史上最全，不接受反驳！！！！！！！

1、说一说三次握手

当面试官问你为什么需要有三次握手、三次握手的作用、讲讲三次三次握手的时候，我想很多人会这样回答：

首先很多人会先讲下握手的过程：

1、第一次握手：客户端给服务器发送一个 SYN 报文。

2、第二次握手：服务器收到 SYN 报文之后，会应答一个 SYN+ACK 报文。

3、第三次握手：客户端收到 SYN+ACK 报文之后，会回应一个 ACK 报文。

4、服务器收到 ACK 报文之后，三次握手建立完成。

作用是为了确认双方的接收与发送能力是否正常。

> **这里我顺便解释一下为啥只有三次握手才能确认双方的接受与发送能力是否正常，而两次却不可以**：
> 第一次握手：客户端发送网络包，服务端收到了。这样服务端就能得出结论：客户端的发送能力、服务端的接收能力是正常的。
>
> 第二次握手：服务端发包，客户端收到了。这样客户端就能得出结论：服务端的接收、发送能力，客户端的接收、发送能力是正常的。不过此时服务器并不能确认客户端的接收能力是否正常。
>
> 第三次握手：客户端发包，服务端收到了。这样服务端就能得出结论：客户端的接收、发送能力正常，服务器自己的发送、接收能力也正常。

因此，需要三次握手才能确认双方的接收与发送能力是否正常。

这样回答其实也是可以的，但我觉得，这个过程的我们应该要描述的更详细一点，因为三次握手的过程中，双方是由很多状态的改变的，而这些状态，也是面试官可能会问的点。所以我觉得在回答三次握手的时候，我们应该要描述的详细一点，而且描述的详细一点意味着可以扯久一点。加分的描述我觉得应该是这样：

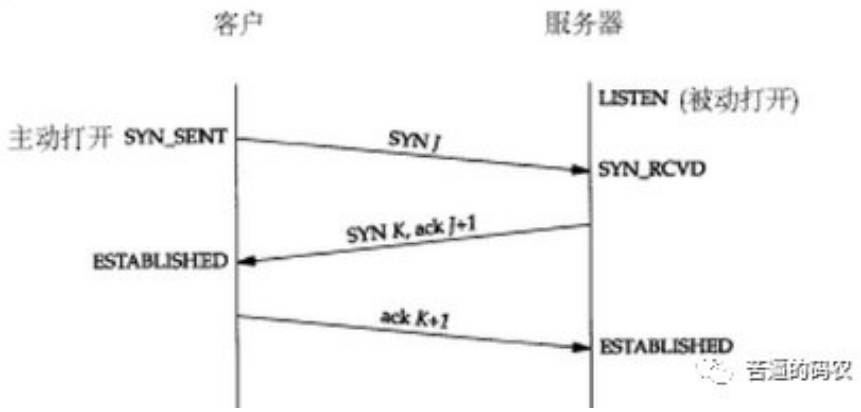
刚开始客户端处于 closed 的状态，服务端处于 listen 状态。然后

1、第一次握手：客户端给服务端发一个 SYN 报文，并指明客户端的初始化序列号 **ISN(c)**。此时客户端处于 **SYN_Send** 状态。

2、第二次握手：服务器收到客户端的 SYN 报文之后，会以自己的 SYN 报文作为应答，并且也是指定了自己的初始化序列号 ISN(s)，同时会把客户端的 ISN + 1 作为 ACK 的值，表示自己已经收到了客户端的 SYN，此时服务器处于 **SYN_REVD** 的状态。

3、第三次握手：客户端收到 SYN 报文之后，会发送一个 ACK 报文，当然，也是一样把服务器的 ISN + 1 作为 ACK 的值，表示已经收到了服务端的 SYN 报文，此时客户端处于 **establised** 状态。

4、服务器收到 ACK 报文之后，也处于 **establised 状态**，此时，双方以建立起了链接



三次握手的作用

三次握手的作用也是有好多的，多记住几个，保证不亏。例如：

1、确认双方的接受能力、发送能力是否正常。

2、指定自己的初始化序列号，为后面的可靠传送做准备。

1、（ISN）是固定的吗

三次握手的一个重要功能是客户端和服务端交换ISN(Initial Sequence Number), 以便让对方知道接下来接收数据的时候如何按序列号组装数据。

如果ISN是固定的，攻击者很容易猜出后续的确认号，因此 ISN 是动态生成的。

2、什么是半连接队列

服务器第一次收到客户端的 SYN 之后，就会处于 SYN_RCVD 状态，此时双方还没有完全建立其连接，服务器会把此种状态下请求连接放在一个队列里，我们把这种队列称之为**半连接队列**。当然还有一个**全连接队列**，就是已经完成三次握手，建立起连接的就会放在全连接队列中。如果队列满了就有可能会出现丢包现象。

> 这里在补充一点关于**SYN-ACK 重传次数**的问题： 服务器发送完SYN－ACK包，如果未收到客户确认包，服务器进行首次重传，等待一段时间仍未收到客户确认包，进行第二次重传，如果重传次数超 过系统规定的最大重传次数，系统将该连接信息从半连接队列中删除。注意，每次重传等待的时间不一定相同，一般会是指数增长，例如间隔时间为 1s, 2s, 4s, 8s,

3、三次握手过程中可以携带数据吗

很多人可能会认为三次握手都不能携带数据，其实第三次握手的时候，是可以携带数据的。也就是说，第一次、第二次握手不可以携带数据，而第三次握手是可以携带数据的。

为什么这样呢？大家可以想一个问题，假如第一次握手可以携带数据的话，如果有人要恶意攻击服务器，那他每次都在第一次握手中的 SYN 报文中放入大量的数据，因为攻击者根本就不理服务器的接收、发送能力是否正常，然后疯狂着重复发 SYN 报文的话，这会让服务器花费很多时间、内存空间来接收这些报文。也就是说，第一次握手可以放数据的话，其中一个简单的原因就是会让服务器更加容易受到攻击了。

而对于第三次的话，此时客户端已经处于 established 状态，也就是说，对于客户端来说，他已经建立起连接了，并且也已经知道服务器的接收、发送能力是正常的了，所以能携带数据页没啥毛病。

2、说一说四次挥手

四次挥手也一样，千万不要对方一个 FIN 报文，我方一个 ACK 报文，再我方一个 FIN 报文，我方一个 ACK 报文。然后结束，最好是说的详细一点，例如想下面这样就差不多了，要把每个阶段的**状态**记好，我上次面试就被问了几个了，呵呵。我答错了，还以为自己答对了，当时还解释的头头是道，呵呵。

刚开始双方都处于 establised 状态，假如是客户端先发起关闭请求，则：

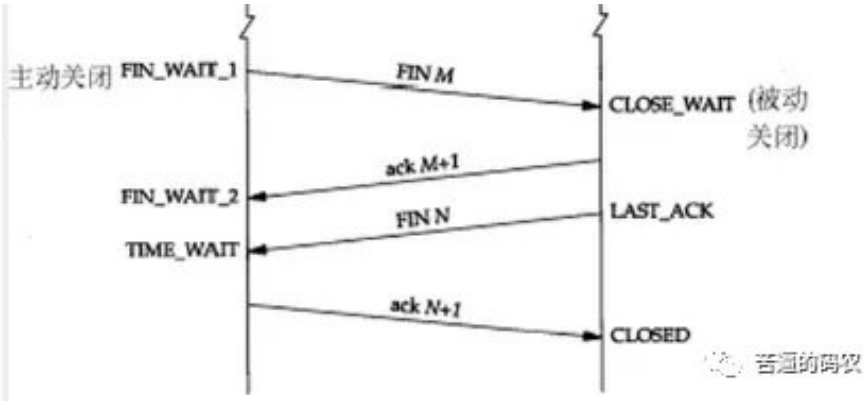
1、第一次挥手：客户端发送一个 FIN 报文，报文中会指定一个序列号。此时客户端处于**CLOSED_WAIT1**状态。

2、第二次握手：服务端收到 FIN 之后，会发送 ACK 报文，且把客户端的序列号值 + 1 作为 ACK 报文的序列号值，表明已经收到客户端的报文了，此时服务端处于 **CLOSE_WAIT2**状态。

3、第三次挥手：如果服务端也想断开连接了，和客户端的第一次挥手一样，发给 FIN 报文，且指定一个序列号。此时服务端处于 **LAST_ACK** 的状态。

4、第四次挥手：客户端收到 FIN 之后，一样发送一个 ACK 报文作为应答，且把服务端的序列号值 + 1 作为自己 ACK 报文的序列号值，此时客户端处于 **TIME_WAIT** 状态。需要过一阵子以确保服务端收到自己的 ACK 报文之后才会进入 CLOSED 状态

5、服务端收到 ACK 报文之后，就处于关闭连接了，处于 CLOSED 状态。



这里特别需要主要的就是**TIME_WAIT**这个状态了，这个是面试的高频考点，就是要理解，为什么客户端发送 ACK 之后不直接关闭，而是要等一阵子才关闭。这其中的原因就是，要确保服务器是否已经收到了我们的 ACK 报文，如果没有收到的话，服务器会重新发 FIN 报文给客户端，客户端再次收到 FIN 报文之后，就知道之前的 ACK 报文丢失了，然后再次发送 ACK 报文。

至于 TIME_WAIT 持续的时间至少是一个报文的来回时间。一般会设置一个计时，如果过了这个计时没有再次收到 FIN 报文，则代表对方成功就是 ACK 报文，此时处于 CLOSED 状态。

这里我给出每个状态所包含的含义，有兴趣的可以看看。

LISTEN - 侦听来自远方TCP端口的连接请求；

SYN-SENT -在发送连接请求后等待匹配的连接请求；

SYN-RECEIVED - 在收到和发送一个连接请求后等待对连接请求的确认；

ESTABLISHED- 代表一个打开的连接，数据可以传送给用户；

FIN-WAIT-1 - 等待远程TCP的连接中断请求，或先前的连接中断请求的确认；

FIN-WAIT-2 - 从远程TCP等待连接中断请求；

CLOSE-WAIT - 等待从本地用户发来的连接中断请求；

CLOSING -等待远程TCP对连接中断的确认；

LAST-ACK - 等待原来发向远程TCP的连接中断请求的确认；

TIME-WAIT -等待足够的时间以确保远程TCP接收到连接中断请求的确认；

CLOSED - 没有任何连接状态；

3、说一说POST与GET有哪些区别

使用场景

GET 用于获取资源，而 POST 用于传输实体主体。

参数

GET 和 POST 的请求都能使用额外的参数，但是 GET 的参数是以查询字符串出现在 URL 中，而 POST 的参数存储在实体主体中。不能因为 POST 参数存储在实体主体中就认为它的安全性更高，因为照样可以通过一些抓包工具（Fiddler）查看。

因为 URL 只支持 ASCII 码，因此 GET 的参数中如果存在中文等字符就需要先进行编码。例如 `中文` 会转换为 `%E4%B8%AD%E6%96%87`，而空格会转换为 `%20`。POST 参数支持标准字符集。

```
GET /test/demo_form.asp?name1=value1&name2=value2 HTTP/1.1Copy to clipboardErrorCopied
POST /test/demo_form.asp HTTP/1.1
Host: w3schools.com
name1=value1&name2=value2Copy to clipboardErrorCopied
```

安全性

安全的 HTTP 方法不会改变服务器状态，也就是说它只是可读的。

GET 方法是安全的，而 POST 却不是，因为 POST 的目的是传送实体主体内容，这个内容可能是用户上传的表单数据，上传成功之后，服务器可能把这个数据存储到数据库中，因此状态也就发生了改变。

安全的方法除了 GET 之外还有：HEAD、OPTIONS。

不安全的方法除了 POST 之外还有 PUT、DELETE。

幂等性

幂等的 HTTP 方法，同样的请求被执行一次与连续执行多次的效果是一样的，服务器的状态也是一样的。换句话说就是，幂等方法不应该具有副作用（统计用途除外）。

所有的安全方法也都是幂等的。

在正确实现的条件下，GET，HEAD，PUT 和 DELETE 等方法都是幂等的，而 POST 方法不是。

GET /pageX HTTP/1.1 是幂等的，连续调用多次，客户端接收到的结果都是一样的：

```
GET /pageX HTTP/1.1
GET /pageX HTTP/1.1
GET /pageX HTTP/1.1
GET /pageX HTTP/1.1Copy to clipboardErrorCopied
```

POST /add_row HTTP/1.1 不是幂等的，如果调用多次，就会增加多行记录：

```
POST /add_row HTTP/1.1   -> Adds a 1nd row
POST /add_row HTTP/1.1   -> Adds a 2nd row
POST /add_row HTTP/1.1   -> Adds a 3rd rowCopy to clipboardErrorCopied
```

DELETE /idX/delete HTTP/1.1 是幂等的，即使不同的请求接收到的状态码不一样：

```
DELETE /idX/delete HTTP/1.1   -> Returns 200 if idX exists
DELETE /idX/delete HTTP/1.1   -> Returns 404 as it just got deleted
DELETE /idX/delete HTTP/1.1   -> Returns 404Copy to clipboardErrorCopied
```

可缓存

如果要对响应进行缓存，需要满足以下条件：

- 请求报文的 HTTP 方法本身是可缓存的，包括 GET 和 HEAD，但是 PUT 和 DELETE 不可缓存，POST 在多数情况下不可缓存的。
- 响应报文的状态码是可缓存的，包括：200, 203, 204, 206, 300, 301, 404, 405, 410, 414, and 501。
- 响应报文的 Cache-Control 首部字段没有指定不进行缓存。

XMLHttpRequest

为了阐述 POST 和 GET 的另一个区别，需要先了解 XMLHttpRequest：

> XMLHttpRequest 是一个 API，它为客户端提供了在客户端和服务器之间传输数据的功能。它提供了一个通过 URL 来获取数据的简单方式，并且不会使整个页面刷新。这使得网页只更新一部分页面而不会打扰到用户。XMLHttpRequest 在 AJAX 中被大量使用。

- 在使用 XMLHttpRequest 的 POST 方法时，浏览器会先发送 Header 再发送 Data。但并不是所有浏览器会这么做，例如火狐就不会。

- 而 GET 方法 Header 和 Data 会一起发送。

另外，已经把这些整理成 PDF 了，送给大家：500道面试题必知必会（附答案）

4、面试官：说一说TCP与UDP的区别

TCP协议的主要特点

（1）TCP是面向连接的运输层协议；所谓面向连接就是双方传输数据之前，必须先建立一条通道，例如三次握手就是建议通道的一个过程，而四次挥手则是结束销毁通道的一个其中过程。

（2）每一条TCP连接只能有两个端点（即两个套接字），只能是点对点的；

（3）TCP提供可靠的传输服务。传送的数据无差错、不丢失、不重复、按序到达；

（4）TCP提供全双工通信。允许通信双方的应用进程在任何时候都可以发送数据，因为两端都设有发送缓存和接受缓存；

（5）面向字节流。虽然应用程序与TCP交互是一次一个大小不等的数据块，但TCP把这些数据看成一连串无结构的字节流，它不保证接收方收到的数据块和发送方发送的数据块具有对应大小关系，例如，发送方应用程序交给发送方的TCP10个数据块，但就受访的TCP可能只用了4个数据块久保收到的字节流交付给上层的应用程序，但字节流完全一样。

TCP的可靠性原理

可靠传输有如下两个特点:

a.传输信道无差错,保证传输数据正确;

b.不管发送方以多快的速度发送数据,接收方总是来得及处理收到的数据;

（1）首先，采用三次握手来建立TCP连接，四次握手来释放TCP连接，从而保证建立的传输信道是可靠的。

（2）其次，TCP采用了连续ARQ协议（回退N，Go-back-N；超时自动重传）来保证数据传输的正确性，使用滑动窗口协议来保证接方能够及时处理所接收到的数据，进行流量控制。

（3）最后，TCP使用慢开始、拥塞避免、快重传和快恢复来进行拥塞控制，避免网络拥塞。

UDP协议特点

（1）UDP是无连接的传输层协议；

（2）UDP使用尽最大努力交付，不保证可靠交付；

（3）UDP是面向报文的，对应用层交下来的报文，不合并，不拆分，保留原报文的边界；

（4）UDP没有拥塞控制，因此即使网络出现拥塞也不会降低发送速率；

（5）UDP支持一对一　一对多　多对多的交互通信；

（6）UDP的首部开销小，只有 8 字节.

TCP和UDP的区别

(1)TCP是可靠传输,UDP是不可靠传输;

(2)TCP面向连接,UDP无连接;

(3)TCP传输数据有序,UDP不保证数据的有序性;

(4)TCP不保存数据边界,UDP保留数据边界;

(5)TCP传输速度相对UDP较慢;

(6)TCP有流量控制和拥塞控制,UDP没有;

(７)TCP是重量级协议,UDP是轻量级协议;

(８)TCP首部较长２０字节,UDP首部较短８字节;

基于TCP和UDP的常用协议

HTTP、HTTPS、FTP、TELNET、SMTP(简单邮件传输协议)协议基于可靠的TCP协议。TFTP、DNS、DHCP、TFTP、SNMP(简单网络管理协议)、RIP基于不可靠的UDP协议

TCP 和 UDP 的常用场景，这个问的好挺多的，例如我当时面试时，就被问过：QQ 登录的过程中，用到了 TCP 和 UDP，QQ 通话呢?

5、面试题：说一说HTTP1.0，1.1，2.0 的区别

HTTP/1.0

1996年5月，HTTP/1.0 版本发布，为了提高系统的效率，**HTTP/1.0规定浏览器与服务器只保持短暂的连接，浏览器的每次请求都需要与服务器建立一个TCP连接，服务器完成请求处理后立即断开TCP连接**，服务器不跟踪每个客户也不记录过去的请求。

这种方式就好像我们打电话的时候，只能说一件事儿一样，说完之后就要挂断，想要说另外一件事儿的时候就要重新拨打电话。

HTTP/1.0中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。

我们知道TCP连接的建立需要三次握手，是很耗费时间的一个过程。所以，HTTP/1.0版本的性能比较差。

> HTTP1.0 其实也可以强制开启长链接，例如接受`Connection: keep-alive`这个字段，但是，这不是标准字段，不同实现的行为可能不一致，因此不是根本的解决办法。

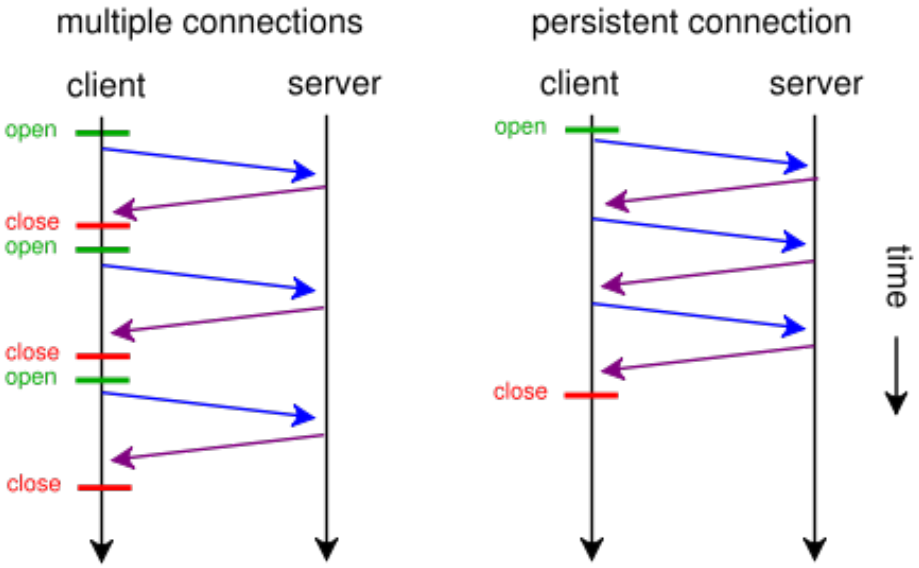
HTTP/1.1

为了解决HTTP/1.0存在的缺陷，HTTP/1.1于1999年诞生。相比较于HTTP/1.0来说，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。**

由于之前打一次电话只能说一件事儿，效率很低。后来人们提出一种想法，就是电话打完之后，先不直接挂断，而是持续一小段时间，这一小段时间内，如果还有事情沟通可以再次进行沟通。

客户端和服务器发现对方一段时间没有活动，就可以主动关闭连接。或者客户端在最后一个请求时，主动告诉服务端要关闭连接。

HTTP/1.1版还引入了管道机制（pipelining），即在同一个TCP连接里面，客户端可以同时发送多个请求。这样就进一步改进了HTTP协议的效率。



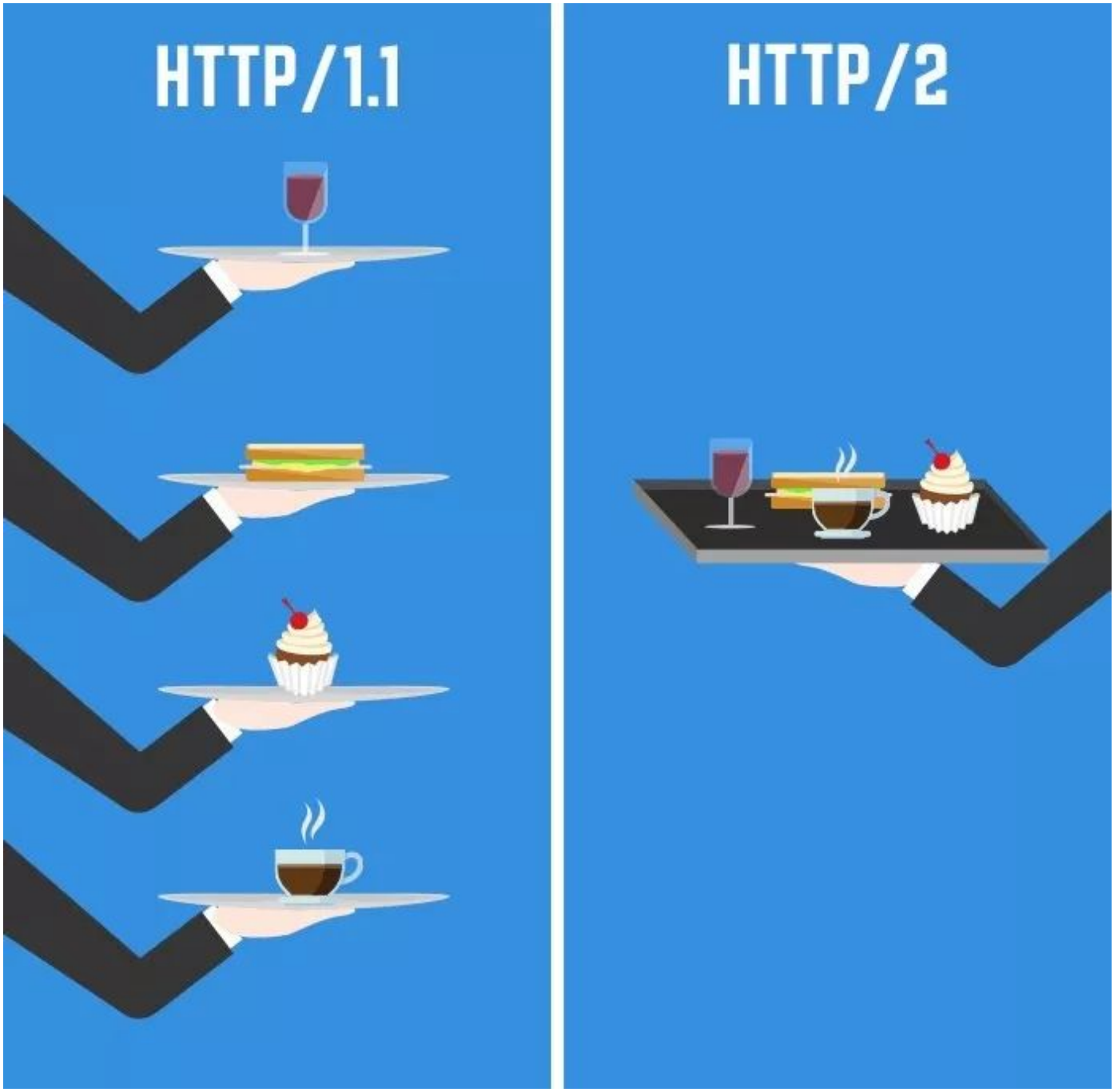
有了持久连接和管道，大大的提升了HTTP的效率。但是服务端还是顺序执行的，效率还有提升的空间。

HTTP/2

HTTP/2 是 HTTP 协议自 1999 年 HTTP 1.1 发布后的首个更新，主要基于 SPDY 协议。

HTTP/2 为了解决HTTP/1.1中仍然存在的效率问题，HTTP/2 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。能这样做有一个前提，就是HTTP/2进行了**二进制分帧**，即 HTTP/2 会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。

也就是说，老板可以同时下达多个命令，员工也可以收到了A请求和B请求，于是先回应A请求，结果发现处理过程非常耗时，于是就发送A请求已经处理好的部分， 接着回应B请求，完成后，再发送A请求剩下的部分。A请求的两部分响应在组合到一起发给老板。



而这个负责拆分、组装请求和二进制帧的一层就叫做**二进制分帧层**。

除此之外，还有一些其他的优化，比如做Header压缩、服务端推送等。

Header压缩就是压缩老板和员工之间的对话。

服务端推送就是员工事先把一些老板可能询问的事情提现发送到老板的手机（缓存）上。这样老板想要知道的时候就可以直接读取短信（缓存）了。

目前，主流的HTTP协议还是HTTP/1.1 和 HTTP/2。并且各大网站的HTTP/2的使用率也在逐年增加。

6、什么是SQL 注入？举个例子?

SQL注入就是通过把SQL命令插入到Web表单提交或输入域名或页面请求的查询字符串，最终达到欺骗服务器执行恶意的SQL命令。

1). SQL注入攻击的总体思路

(1). 寻找到SQL注入的位置　　(2). 判断服务器类型和后台数据库类型　　(3). 针对不通的服务器和数据库特点进行SQL注入攻击

2). SQL注入攻击实例

比如，在一个登录界面，要求输入用户名和密码，可以这样输入实现免帐号登录：

```
用户名:  'or 1 = 1 --
密 码:
```

用户一旦点击登录，如若没有做特殊处理，那么这个非法用户就很得意的登陆进去了。这是为什么呢?

下面我们分析一下：从理论上说，后台认证程序中会有如下的SQL语句：

```
String sql = "select * from user_table where username=' "+userName+" ' and password='
"+password+" '";
```

因此，当输入了上面的用户名和密码，上面的SQL语句变成：

```
SELECT * FROM user_table WHERE username=''or 1 = 1 -- and password=''
```

分析上述SQL语句我们知道，username=' or 1=1 这个语句一定会成功；然后后面加两个-，这意味着注释，它将后面的语句注释，让他们不起作用。这样，上述语句永远都能正确执行，用户轻易骗过系统，获取合法身份。

3). 应对方法

(1). 参数绑定

使用预编译手段，绑定参数是最好的防SQL注入的方法。目前许多的ORM框架及JDBC等都实现了SQL预编译和参数绑定功能，攻击者的恶意SQL会被当做SQL的参数而不是SQL命令被执行。在mybatis的mapper文件中，对于传递的参数我们一般是使用#和$来获取参数值。当使用#时，变量是占位符，就是一般我们使用javajdbc的PrepareStatement时的占位符，所有可以防止sql注入；当使用$时，变量就是直接追加在sql中，一般会有sql注入问题。

(2). 使用正则表达式过滤传入的参数

7、谈一谈 XSS 攻击，举个例子?

XSS是一种经常出现在web应用中的计算机安全漏洞，与SQL注入一起成为web中最主流的攻击方式。

XSS是指恶意攻击者利用网站没有对用户提交数据进行转义处理或者过滤不足的缺点，进而添加一些脚本代码嵌入到web页面中去，使别的用户访问都会执行相应的嵌入代码，从而盗取用户资料、利用用户身份进行某种动作或者对访问者进行病毒侵害的一种攻击方式。

1). XSS攻击的危害

盗取各类用户帐号，如机器登录帐号、用户网银帐号、各类管理员帐号

控制企业数据，包括读取、篡改、添加、删除企业敏感数据的能力

盗窃企业重要的具有商业价值的资料

非法转账

强制发送电子邮件

网站挂马

控制受害者机器向其它网站发起攻击

2). 原因解析

主要原因：过于信任客户端提交的数据！

解决办法：不信任任何客户端提交的数据，只要是客户端提交的数据就应该先进行相应的过滤处理然后方可进行下一步的操作。

进一步分析细节：客户端提交的数据本来就是应用所需要的，但是恶意攻击者利用网站对客户端提交数据的信任，在数据中插入一些符号以及javascript代码，那么这些数据将会成为应用代码中的一部分了，那么攻击者就可以肆无忌惮地展开攻击啦，因此我们绝不可以信任任何客户端提交的数据！！！

3). XSS 攻击分类

(1). 反射性XSS攻击 (非持久性XSS攻击)

漏洞产生的原因是攻击者注入的数据反映在响应中。一个典型的非持久性XSS攻击包含一个带XSS攻击向量的链接(即每次攻击需要用户的点击)，例如，正常发送消息：

```
http://www.test.com/message.php?send=Hello,World!
```

接收者将会接收信息并显示Hello,World；但是，非正常发送消息：

```
http://www.test.com/message.php?send=<script>alert(‘foolish!’)</script>!
```

接收者接收消息显示的时候将会弹出警告窗口！

(2). 持久性XSS攻击 (留言板场景)

XSS攻击向量(一般指XSS攻击代码)存储在网站数据库，当一个页面被用户打开的时候执行。也就是说，每当用户使用浏览器打开指定页面时，脚本便执行。

与非持久性XSS攻击相比，持久性XSS攻击危害性更大。从名字就可以了解到，持久性XSS攻击就是将攻击代码存入数据库中，然后客户端打开时就执行这些攻击代码。

例如，留言板表单中的表单域：

```
<input type="text" name="content" value="这里是用户填写的数据">
```

正常操作流程是：用户是提交相应留言信息 —— 将数据存储到数据库 —— 其他用户访问留言板，应用去数据并显示；而非正常操作流程是攻击者在value填写:

```
<script>alert('foolish!');</script> <!--或者html其他标签（破坏样式。。。）、一段攻击型代码-->
```

并将数据提交、存储到数据库中；当其他用户取出数据显示的时候，将会执行这些攻击性代码。

4). 修复漏洞方针

漏洞产生的根本原因是 太相信用户提交的数据，对用户所提交的数据过滤不足所导致的，因此解决方案也应该从这个方面入手，具体方案包括：

将重要的cookie标记为http only, 这样的话Javascript 中的document.cookie语句就不能获取到cookie了（如果在cookie中设置了HttpOnly属性，那么通过js脚本将无法读取到cookie信息，这样能有效的防止XSS攻击）；

表单数据规定值的类型，例如：年龄应为只能为int、name只能为字母数字组合。。。。

对数据进行Html Encode 处理

过滤或移除特殊的Html标签，例如:

```
<script>, <iframe> , < for <, > for>, &quot for
```

过滤JavaScript 事件的标签，例如 “onclick=”, “onfocus” 等等。

需要注意的是，在有些应用中是允许html标签出现的，甚至是javascript代码出现。因此，我们在过滤数据的时候需要仔细分析哪些数据是有特殊要求（例如输出需要html代码、javascript代码拼接、或者此表单直接允许使用等等），然后区别处理！

8、在交互过程中如果数据传送完了，还不想断开连接怎么办，怎么维持?

在 HTTP 中响应体的 Connection 字段指定为 keep-alive

```
connetion:keep-alive;
```

9、GET请求中URL编码的意义

我们知道，在GET请求中会对URL中非西文字符进行编码，这样做的目的就是为了 **避免歧义**。看下面的例子，

针对“name1=value1&name2=value2”的例子，我们来谈一下数据从客户端到服务端的解析过程。首先，上述字符串在计算机中用ASCII吗表示为：

```
    6E616D6531 3D 76616C756531 26 6E616D6532 3D 76616C756532    6E616D6531: name1    3D: =
    76616C756531: value1    26: &    6E616D6532: name2    3D: =    76616C756532: value2
```

服务端在接收到该数据后就可以遍历该字节流，一个字节一个字节的吃，当吃到3D这字节后，服务端就知道前面吃得字节表示一个key，再往后吃，如果遇到26，说明从刚才吃的3D到26子节之间的是上一个key的value，以此类推就可以解析出客户端传过来的参数。

现在考虑这样一个问题，如果我们的参数值中就包含 = 或 & 这种特殊字符的时候该怎么办？比如，“name1=value1”，其中value1的值是“va&lu=e1”字符串，那么实际在传输过程中就会变成这样“name1=va&lu=e1”。这样，我们的本意是只有一个键值对，但是服务端却会解析成两个键值对，这样就产生了歧义。

那么，如何解决上述问题带来的歧义呢？解决的办法就是对参数进行URL编码：例如，我们对上述会产生歧义的字符进行URL编码后结果：“name1=va%26lu%3D”，这样服务端会把紧跟在“%”后的字节当成普通的字节，就是不会把它当成各个参数或键值对的分隔符

另外，已经把这些整理成 PDF 了，送给大家：500道面试题必知必会（附答案）

10、HTTP 哪些常用的状态码及使用场景?

状态码分类

1xx：表示目前是协议的中间状态，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：服务器端错误

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 服务器端错误

503 服务器繁忙

11、HTTP 如何实现长连接？在什么时候会超时?

通过在头部（请求和响应头）设置 Connection: keep-alive，HTTP1.0协议支持，但是默认关闭，从HTTP1.1协议以后，连接默认都是长连接

1、HTTP 一般会有 httpd 守护进程，里面可以设置 keep-alive timeout，当 tcp 链接闲置超过这个时间就会关闭，也可以在 HTTP 的 header 里面设置超时时间

2、TCP 的 keep-alive 包含三个参数，支持在系统内核的 net.ipv4 里面设置：当 TCP 链接之后，闲置了 tcp_keepalive_time，则会发生侦测包，如果没有收到对方的 ACK，那么会每隔 tcp_keepalive_intvl 再发一次，直到发送了 tcp_keepalive_probes，就会丢弃该链接。

（1）tcp_keepalive_intvl = 15 （2）tcp_keepalive_probes = 5 （3）tcp_keepalive_time = 1800

实际上 HTTP 没有长短链接，只有 TCP 有，TCP 长连接可以复用一个 TCP 链接来发起多次 HTTP 请求，这样可以减少资源消耗，比如一次请求 HTML，可能还需要请求后续的 JS/CSS/图片等

12、HTTP状态码301和302的区别，都有哪些用途?

一. 301重定向的概念

301重定向（301 Move Permanently），指页面永久性转移，表示为资源或页面永久性地转移到了另一个位置。301是HTTP协议中的一种状态码，当用户或搜索引擎向服务器发出浏览请求时，服务器返回的HTTP数据流中头信息（header）中包含状态码 301 ，表示该资源已经永久改变了位置。

301重定向是一种非常重要的"自动转向"技术，网址重定向最为可行的一种方法。

二. 哪些情况需要做301重定向?

网页开发过程中，时常会遇到网站目录结构的调整，将页面转移到一个新地址；网页扩展名的改变，这些变化都会导致网页地址发生改变，此时用户收藏夹和搜索引擎数据库中的旧地址是一个错误的地址，访问之后会出现404页面，直接导致网站流量的损失。或者是我们需要多个域名跳转至同一个域名，例如本站主站点域名为 www.conimi.com ，而还有一个域名 www.nico.cc，由于对该域名设置了301重定向，当输入www.nico.cc 时，自动跳转至 www.conimi.com 。

三. 301重定向有什么优点?

有利于网站首选域的确定，对于同一资源页面多条路径的301重定向有助于URL权重的集中。例如 www.conimi.com和 conimi.com 是两个不同的域名，但是指向的内容完全相同，搜索引擎会对两个域名收录情况不同，这样导致网站权重和排名被分散；对conimi.com 做301重定向跳转至www.conimi.com 后，权重和排名集中到www.conimi.com，从而提升自然排名。

四. 302重定向又是什么鬼?

302重定向（302 Move Temporarily），指页面暂时性转移，表示资源或页面暂时转移到另一个位置，常被用作网址劫持，容易导致网站降权，严重时网站会被封掉，不推荐使用。

五. 301与302的区别

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。

13、IP地址有哪些分类?

A类地址(1~126)：网络号占前8位，以0开头，主机号占后24位。

B类地址(128~191)：网络号占前16位，以10开头，主机号占后16位。

C类地址(192~223)：网络号占前24位，以110开头，主机号占后8位。

D类地址(224~239)：以1110开头，保留位多播地址。

E类地址(240~255)：以1111开头，保留位今后使用

这种两级的 IP 地址可以记为：

$$\text{IP 地址} ::= \{ <\text{网络号}>, <\text{主机号}> \} \tag{4-1}$$

式(4-1)中的符号“::=”表示“**定义为**”。图 4-5 给出了各种 IP 地址的网络号字段和主机号字段，这里 A 类、B 类和 C 类地址都是**单播地址**（一对一通信），是最常用的。

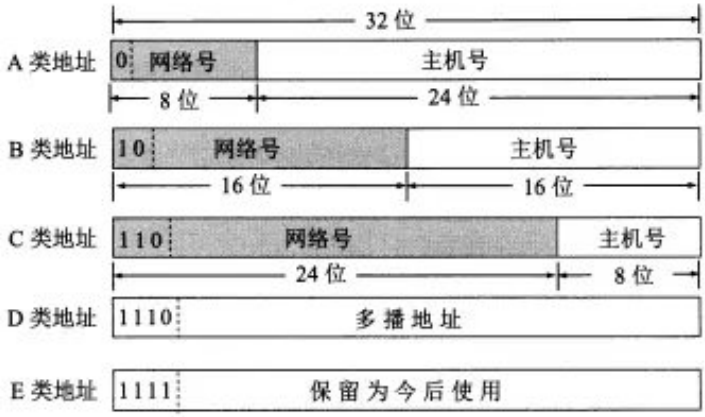


图 4-5　IP 地址中的网络号字段和主机号字段

从图 4-5 可以看出：

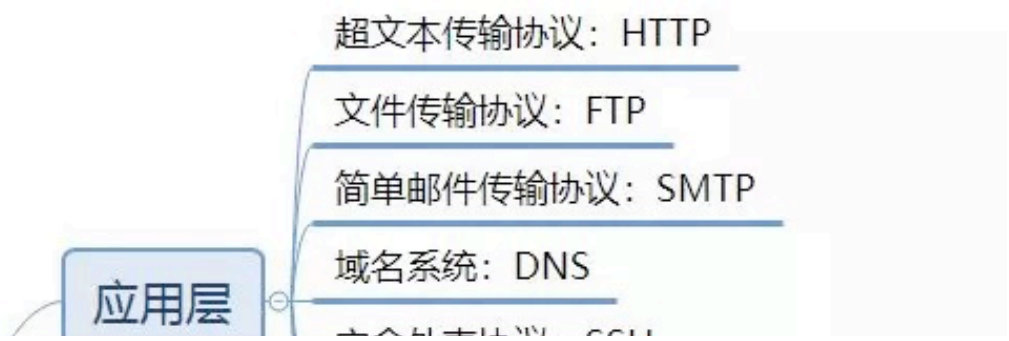
- A 类、B 类和 C 类地址的网络号字段（在图中这个字段是灰色的）分别为 1 个、2 个和 3 个字节长，而在网络号字段的最前面有 1~3 位的**类别位**，其数值分别规定为 0，10 和 110。
- A 类、B 类和 C 类地址的主机号字段分别为 3 个、2 个和 1 个字节长。
- D 类地址（前 4 位是 1110）用于**多播**（一对多通信）。我们将在 4.6 节讨论 IP 多播。
- E 类地址（前 4 位是 1111）保留为以后用。

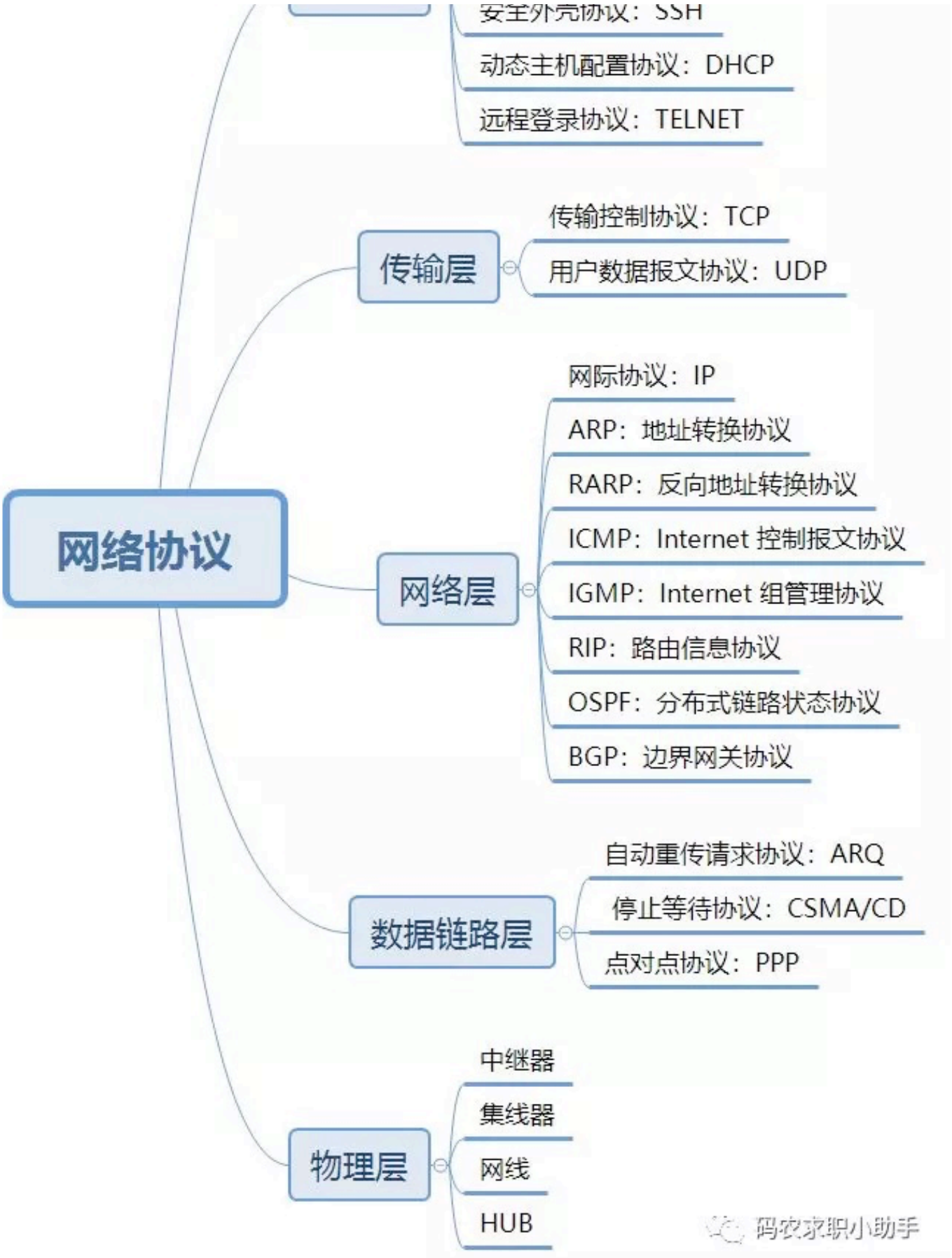
表 4-2　IP 地址的指派范围

网络类别	最大可指派的网络数	第一个可指派的网络号	最后一个可指派的网络号	每个网络中的最大主机数
A	126 (2^7-2)	1	126	16777214
B	16383 ($2^{14}-1$)	128.1	191.255	65534
C	2097151 ($2^{21}-1$)	192.0.1	223.255.255	254

14、简单说下每一层对应的网络协议有哪些?

计算机五层网络体系中涉及的协议非常多，下面就常用的做了列举：




网络协议
安全外壳协议：SSH
动态主机配置协议：DHCP
远程登录协议：TELNET
传输层
传输控制协议：TCP
用户数据报文协议：UDP
网络层
网际协议：IP
ARP：地址转换协议
RARP：反向地址转换协议
ICMP：Internet 控制报文协议
IGMP：Internet 组管理协议
RIP：路由信息协议
OSPF：分布式链路状态协议
BGP：边界网关协议
数据链路层
自动重传请求协议：ARQ
停止等待协议：CSMA/CD
点对点协议：PPP
物理层
中继器
集线器
网线
HUB
码农求职小助手

15、ARP 协议的工作原理?

网络层的 ARP 协议完成了 IP 地址与物理地址的映射。首先，每台主机都会在自己的 ARP 缓冲区中建立一个 ARP 列表，以表示 IP 地址和 MAC 地址的对应关系。当源主机需要将一个数据包要发送到目的主机时，会首先检查自己 ARP 列表中是否存在该 IP 地址对应的 MAC 地址：如果有，就直接将数据包发送到这个 MAC 地址；如果没有，就向本地网段发起一个 ARP 请求的广播包，查询此目的主机对应的 MAC 地址。

此 ARP 请求数据包里包括源主机的 IP 地址、硬件地址、以及目的主机的 IP 地址。网络中所有的主机收到这个 ARP 请求后，会检查数据包中的目的 IP 是否和自己的 IP 地址一致。如果不相同就忽略此数据包；如果相同，该主机首先将发送端的 MAC 地址和 IP 地址添加到自己的 ARP 列表中，如果 ARP 表中已经存在该 IP 的信息，则将其覆盖，然后给源主机发送一个 ARP 响应数据包，告诉对方自己是它需要查找的 MAC 地址；源主机收到这个 ARP 响应数据包后，将得到的目的主机的 IP 地址和 MAC 地址添加到自己的 ARP 列表中，并利用此信息开始数据的传输。如果源主机一直没有收到 ARP 响应数据包，表示 ARP 查询失败

另外，已经把这些整理成 PDF 了，送给大家：500道面试题必知必会（附答案）

16、TCP 的主要特点是什么?

1. TCP 是面向连接的。（就好像打电话一样，通话前需要先拨号建立连接，通话结束后要挂机释放连接）；
2. 每一条 TCP 连接只能有两个端点，每一条 TCP 连接只能是点对点的（一对一）；
3. TCP 提供可靠交付的服务。通过 TCP 连接传送的数据，无差错、不丢失、不重复、并且按序到达；
4. TCP 提供全双工通信。TCP 允许通信双方的应用进程在任何时候都能发送数据。TCP 连接的两端都设有发送缓存和接收缓存，用来临时存放双方通信的数据；
5. 面向字节流。TCP 中的“流”（Stream）指的是流入进程或从进程流出的字节序列。“面向字节流”的含义是：虽然应用程序和 TCP 的交互是一次一个数据块（大小不等），但 TCP 把应用程序交下来的数据仅仅看成是一连串的无结构的字节流。

17、UDP 的主要特点是什么?

1. UDP 是无连接的；
2. UDP 使用尽最大努力交付，即不保证可靠交付，因此主机不需要维持复杂的链接状态（这里面有许多参数）；
3. UDP 是面向报文的；
4. UDP 没有拥塞控制，因此网络出现拥塞不会使源主机的发送速率降低（对实时应用很有用，如 直播，实时视频会议等）；
5. UDP 支持一对一、一对多、多对一和多对多的交互通信；
6. UDP 的首部开销小，只有 8 个字节，比 TCP 的 20 个字节的首部要短。

18、TCP 和 UDP 分别对应的常见应用层协议有哪些?

1. TCP 对应的应用层协议

FTP：定义了文件传输协议，使用 21 端口。常说某某计算机开了 FTP 服务便是启动了文件传输服务。下载文件，上传主页，都要用到 FTP 服务。

Telnet：它是一种用于远程登陆的端口，用户可以以自己的身份远程连接到计算机上，通过这种端口可以提供一种基于 DOS 模式下的通信服务。如以前的 BBS 是-纯字符界面的，支持 BBS 的服务器将 23 端口打开，对外提供服务。

SMTP：定义了简单邮件传送协议，现在很多邮件服务器都用的是这个协议，用于发送邮件。如常见的免费邮件服务中用的就是这个邮件服务端口，所以在电子邮件设置-中常看到有这么 SMTP 端口设置这个栏，服务器开放的是 25 号端口。

POP3：它是和 SMTP 对应，POP3 用于接收邮件。通常情况下，POP3 协议所用的是 110 端口。也是说，只要你有相应的使用 POP3 协议的程序（例如 Fo-xmail 或 Outlook），就可以不以 Web 方式登陆进邮箱界面，直接用邮件程序就可以收到邮件（如是163 邮箱就没有必要先进入网易网站，再进入自己的邮-箱来收信）。

HTTP：从 Web 服务器传输超文本到本地浏览器的传送协议。

2. UDP 对应的应用层协议

DNS：用于域名解析服务，将域名地址转换为 IP 地址。DNS 用的是 53 号端口。

SNMP：简单网络管理协议，使用 161 号端口，是用来管理网络设备的。由于网络设备很多，无连接的服务就体现出其优势。

TFTP(Trival File Transfer Protocal)：简单文件传输协议，该协议在熟知端口 69 上使用 UDP 服务。

19、为什么 TIME-WAIT 状态必须等待 2MSL 的时间呢?

1、为了保证 A 发送的最后一个 ACK 报文段能够到达 B。这个 ACK 报文段有可能丢失，因而使处在 LAST-ACK 状态的 B 收不到对已发送的 FIN + ACK 报文段的确认。B 会超时重传这个 FIN+ACK 报文段，而 A 就能在 2MSL 时间内（超时 + 1MSL 传输）收到这个重传的 FIN+ACK 报文段。接着 A 重传一次确认，重新启动 2MSL 计时器。最后，A 和 B 都正常进入到 CLOSED 状态。如果 A 在 TIME-WAIT 状态不等待一段时间，而是在发送完 ACK 报文段后立即释放连接，那么就无法收到 B 重传的 FIN + ACK 报文段，因而也不会再发送一次确认报文段，这样，B 就无法按照正常步骤进入 CLOSED 状态。

2、 防止已失效的连接请求报文段出现在本连接中。A 在发送完最后一个 ACK 报文段后，再经过时间 2MSL，就可以使本连接持续的时间内所产生的所有报文段都从网络中消失。这样就可以使下一个连接中不会出现这种旧的连接请求报文段。

20、保活计时器的作用?

除时间等待计时器外，TCP 还有一个保活计时器（keepalive timer）。设想这样的场景：客户已主动与服务器建立了 TCP 连接。但后来客户端的主机突然发生故障。显然，服务器以后就不能再收到客户端发来的数据。因此，应当有措施使服务器不要再白白等待下去。这就需要使用保活计时器了。

服务器每收到一次客户的数据，就重新设置保活计时器，时间的设置通常是两个小时。若两个小时都没有收到客户端的数据，服务端就发送一个探测报文段，以后则每隔 75 秒钟发送一次。若连续发送 10个 探测报文段后仍然无客户端的响应，服务端就认为客户端出了故障，接着就关闭这个连接。

21、TCP 协议是如何保证可靠传输的?

1. 数据包校验：目的是检测数据在传输过程中的任何变化，若校验出包有错，则丢弃报文段并且不给出响应，这时 TCP 发送数据端超时后会重发数据；
2. 对失序数据包重排序：既然 TCP 报文段作为 IP 数据报来传输，而 IP 数据报的到达可能会失序，因此 TCP 报文段的到达也可能会失序。TCP 将对失序数据进行重新排序，然后才交给应用层；
3. 丢弃重复数据：对于重复数据，能够丢弃重复数据；
4. 应答机制：当 TCP 收到发自 TCP 连接另一端的数据，它将发送一个确认。这个确认不是立即发送，通常将推迟几分之一秒；
5. 超时重发：当 TCP 发出一个段后，它启动一个定时器，等待目的端确认收到这个报文段。如果不能及时收到

一个确认，将重发这个报文段；

6. 流量控制：TCP 连接的每一方都有固定大小的缓冲空间。TCP 的接收端只允许另一端发送接收端缓冲区所能接纳的数据，这可以防止较快主机致使较慢主机的缓冲区溢出，这就是流量控制。TCP 使用的流量控制协议是可变大小的滑动窗口协议。

22、谈谈你对停止等待协议的理解?

停止等待协议是为了实现可靠传输的，它的基本原理就是每发完一个分组就停止发送，等待对方确认。在收到确认后再发下一个分组；在停止等待协议中，若接收方收到重复分组，就丢弃该分组，但同时还要发送确认。主要包括以下几种情况：无差错情况、出现差错情况（超时重传）、确认丢失和确认迟到、确认丢失和确认迟到。

23、谈谈你对 ARQ 协议的理解?

自动重传请求 ARQ 协议

停止等待协议中超时重传是指只要超过一段时间仍然没有收到确认，就重传前面发送过的分组（认为刚才发送过的分组丢失了）。因此每发送完一个分组需要设置一个超时计时器，其重传时间应比数据在分组传输的平均往返时间更长一些。这种自动重传方式常称为自动重传请求 ARQ。

连续 ARQ 协议

连续 ARQ 协议可提高信道利用率。发送方维持一个发送窗口，凡位于发送窗口内的分组可以连续发送出去，而不需要等待对方确认。接收方一般采用累计确认，对按序到达的最后一个分组发送确认，表明到这个分组为止的所有分组都已经正确收到了。

24、谈谈你对滑动窗口的了解?

TCP 利用滑动窗口实现流量控制的机制。滑动窗口（Sliding window）是一种流量控制技术。早期的网络通信中，通信双方不会考虑网络的拥挤情况直接发送数据。由于大家不知道网络拥塞状况，同时发送数据，导致中间节点阻塞掉包，谁也发不了数据，所以就有了滑动窗口机制来解决此问题。

TCP 中采用滑动窗口来进行传输控制，滑动窗口的大小意味着接收方还有多大的缓冲区可以用于接收数据。发送方可以通过滑动窗口的大小来确定应该发送多少字节的数据。当滑动窗口为 0 时，发送方一般不能再发送数据报，但有两种情况除外，一种情况是可以发送紧急数据，例如，允许用户终止在远端机上的运行进程。另一种情况是发送方可以发送一个 1 字节的数据报来通知接收方重新声明它希望接收的下一字节及发送方的滑动窗口大小。

25、谈下你对流量控制的理解?

TCP 利用滑动窗口实现流量控制。流量控制是为了控制发送方发送速率，保证接收方来得及接收。接收方发送的确认报文中的窗口字段可以用来控制发送方窗口大小，从而影响发送方的发送速率。将窗口字段设置为 0，则发送方不能发送数据。

另外，已经把这些整理成 PDF 了，送给大家：500道面试题必知必会（附答案）

26、谈下你对 TCP 拥塞控制的理解？使用了哪些算法?

拥塞控制和流量控制不同，前者是一个全局性的过程，而后者指点对点通信量的控制。在某段时间，若对网络中某一资源的需求超过了该资源所能提供的可用部分，网络的性能就要变坏。这种情况就叫拥塞。

拥塞控制就是为了防止过多的数据注入到网络中，这样就可以使网络中的路由器或链路不致于过载。拥塞控制所要做的都有一个前提，就是网络能够承受现有的网络负荷。拥塞控制是一个全局性的过程，涉及到所有的主机，所有的路由器，以及与降低网络传输性能有关的所有因素。相反，流量控制往往是点对点通信量的控制，是个端到端的问题。流量控制所要做到的就是抑制发送端发送数据的速率，以便使接收端来得及接收。

为了进行拥塞控制，TCP 发送方要维持一个拥塞窗口(cwnd) 的状态变量。拥塞控制窗口的大小取决于网络的拥塞程度，并且动态变化。发送方让自己的发送窗口取为拥塞窗口和接收方的接受窗口中较小的一个。

TCP 的拥塞控制采用了四种算法，即：慢开始、拥塞避免、快重传和快恢复。在网络层也可以使路由器采用适当的分组丢弃策略（如：主动队列管理 AQM），以减少网络拥塞的发生。

- **慢开始：**

慢开始算法的思路是当主机开始发送数据时，如果立即把大量数据字节注入到网络，那么可能会引起网络阻塞，因为现在还不知道网络的符合情况。经验表明，较好的方法是先探测一下，即由小到大逐渐增大发送窗口，也就是由小到大逐渐增大拥塞窗口数值。cwnd 初始值为 1，每经过一个传播轮次，cwnd 加倍。

- **拥塞避免：**

拥塞避免算法的思路是让拥塞窗口 cwnd 缓慢增大，即每经过一个往返时间 RTT 就把发送方的 cwnd 加 1。

- **快重传与快恢复：**

在 TCP/IP 中，快速重传和快恢复（fast retransmit and recovery，FRR）是一种拥塞控制算法，它能快速恢复丢失的数据包。

没有 FRR，如果数据包丢失了，TCP 将会使用定时器来要求传输暂停。在暂停的这段时间内，没有新的或复制的数据包被发送。有了 FRR，如果接收机接收到一个不按顺序的数据段，它会立即给发送机发送一个重复确认。如果发送机接收到三个重复确认，它会假定确认件指出的数据段丢失了，并立即重传这些丢失的数据段。

有了 FRR，就不会因为重传时要求的暂停被耽误。当有单独的数据包丢失时，快速重传和快恢复（FRR）能最有效地工作。当有多个数据信息包在某一段很短的时间内丢失时，它则不能很有效地工作。

27、什么是粘包?

在进行 Java NIO 学习时，可能会发现：如果客户端连续不断的向服务端发送数据包时，服务端接收的数据会出现两个数据包粘在一起的情况。

1. TCP 是基于字节流的，虽然应用层和 TCP 传输层之间的数据交互是大小不等的数据块，但是 TCP 把这些数据块仅仅看成一连串无结构的字节流，没有边界；
2. 从 TCP 的帧结构也可以看出，在 TCP 的首部没有表示数据长度的字段。

基于上面两点，在使用 TCP 传输数据时，才有粘包或者拆包现象发生的可能。一个数据包中包含了发送端发送的两个数据包的信息，这种现象即为粘包。

接收端收到了两个数据包，但是这两个数据包要么是不完整的，要么就是多出来一块，这种情况即发生了拆包和粘包。拆包和粘包的问题导致接收端在处理的时候会非常困难，因为无法区分一个完整的数据包。

28、TCP 黏包是怎么产生的?

- **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于的小时，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 协议接收数据时的过程是这样的：数据到接收方，从网络模型的下方传递至传输层，传输层的 TCP 协议处理是将其放置接收缓冲区，然后由应用层来主动获取（C 语言用 recv、read 等函数）；这时会出现一个问题，就是我们在程序中调用的读取数据函数不能及时的把缓冲区中的数据拿出来，而下一个数据又到来并有一部分放入的缓冲区末尾，等我们读取数据时就是一个粘包。（放数据的速度 > 应用层拿数据速度）

29、怎么解决拆包和粘包?

分包机制一般有两个通用的解决方法：

1. 特殊字符控制；
2. 在包头首都添加数据包的长度。

如果使用 netty 的话，就有专门的编码器和解码器解决拆包和粘包问题了。

tips：UDP 没有粘包问题，但是有丢包和乱序。不完整的包是不会有的，收到的都是完全正确的包。传送的数据单位协议是 UDP 报文或用户数据报，发送的时候既不合并，也不拆分。

30、forward 和 redirect 的区别?

Forward 和 Redirect 代表了两种请求转发方式：直接转发和间接转发。

直接转发方式（Forward）：客户端和浏览器只发出一次请求，Servlet、HTML、JSP 或其它信息资源，由第二个信息资源响应该请求，在请求对象 request 中，保存的对象对于每个信息资源是共享的。

间接转发方式（Redirect）：实际是两次 HTTP 请求，服务器端在响应第一次请求的时候，让浏览器再向另外一个 URL 发出请求，从而达到转发的目的。

- 举个通俗的例子：

直接转发就相当于：“A 找 B 借钱，B 说没有，B 去找 C 借，借到借不到都会把消息传递给 A”；

间接转发就相当于："A 找 B 借钱，B 说没有，让 A 去找 C 借"。

31、HTTP 方法有哪些?

客户端发送的 请求报文 第一行为请求行，包含了方法字段。

1. GET：获取资源，当前网络中绝大部分使用的都是 GET；
2. HEAD：获取报文首部，和 GET 方法类似，但是不返回报文实体主体部分；
3. POST：传输实体主体
4. PUT：上传文件，由于自身不带验证机制，任何人都可以上传文件，因此存在安全性问题，一般不使用该方法。

5. PATCH：对资源进行部分修改。PUT 也可以用于修改资源，但是只能完全替代原始资源，PATCH 允许部分修改。
6. OPTIONS：查询指定的 URL 支持的方法；
7. CONNECT：要求在与代理服务器通信时建立隧道。使用 SSL（Secure Sockets Layer，安全套接层）和 TLS（Transport Layer Security，传输层安全）协议把通信内容加密后经网络隧道传输。
8. TRACE：追踪路径。服务器会将通信路径返回给客户端。发送请求时，在 Max-Forwards 首部字段中填入数值，每经过一个服务器就会减 1，当数值为 0 时就停止传输。通常不会使用 TRACE，并且它容易受到 XST 攻击（Cross-Site Tracing，跨站追踪）。

32、在浏览器中输入 URL 地址到显示主页的过程?

1. DNS 解析：浏览器查询 DNS，获取域名对应的 IP 地址：具体过程包括浏览器搜索自身的 DNS 缓存、搜索操作系统的 DNS 缓存、读取本地的 Host 文件和向本地 DNS 服务器进行查询等。对于向本地 DNS 服务器进行查询，如果要查询的域名包含在本地配置区域资源中，则返回解析结果给客户机，完成域名解析(此解析具有权威性)；如果要查询的域名不由本地 DNS 服务器区域解析，但该服务器已缓存了此网址映射关系，则调用这个 IP 地址映射，完成域名解析（此解析不具有权威性）。如果本地域名服务器并未缓存该网址映射关系，那么将根据其设置发起递归查询或者迭代查询；
2. TCP 连接：浏览器获得域名对应的 IP 地址以后，浏览器向服务器请求建立链接，发起三次握手；
3. 发送 HTTP 请求：TCP 连接建立起来后，浏览器向服务器发送 HTTP 请求；
4. 服务器处理请求并返回 HTTP 报文：服务器接收到这个请求，并根据路径参数映射到特定的请求处理器进行处理，并将处理结果及相应的视图返回给浏览器；
5. 浏览器解析渲染页面：浏览器解析并渲染视图，若遇到对 js 文件、css 文件及图片等静态资源的引用，则重复上述步骤并向服务器请求这些资源；浏览器根据其请求到的资源、数据渲染页面，最终向用户呈现一个完整的页面。
6. 连接结束。

另外，已经把这些整理成 PDF 了，送给大家：500道面试题必知必会（附答案）

33、DNS 的解析过程?

1. 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，向根域名服务器继续发出查询请求报文(即替主机继续查询)，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是所要查询的 IP 地址，或者是报错，表示无法查询到所需的 IP 地址。
2. 本地域名服务器向根域名服务器的查询的迭代查询。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，要么告诉本地服务器：“你下一步应当向哪一个域名服务器进行查询”。然后让本地服务器进行后续的查询。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，要么告诉本地服务器下一步应当向哪一个权限域名服务器进行查询。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。

34、谈谈你对域名缓存的了解?

为了提高 DNS 查询效率，并减轻服务器的负荷和减少因特网上的 DNS 查询报文数量，在域名服务器中广泛使用了高速缓存，用来存放最近查询过的域名以及从何处获得域名映射信息的记录。

由于名字到地址的绑定并不经常改变，为保持高速缓存中的内容正确，域名服务器应为每项内容设置计时器并处理超过合理时间的项（例如：每个项目两天）。当域名服务器已从缓存中删去某项信息后又被请求查询该项信息，就必须重新到授权管理该项的域名服务器绑定信息。当权限服务器回答一个查询请求时，在响应中都指明绑定有效存在的时间值。增加此时间值可减少网络开销，而减少此时间值可提高域名解析的正确性。

不仅在本地域名服务器中需要高速缓存，在主机中也需要。许多主机在启动时从本地服务器下载名字和地址的全部数据库，维护存放自己最近使用的域名的高速缓存，并且只在从缓存中找不到名字时才使用域名服务器。维护本地域名服务器数据库的主机应当定期地检查域名服务器以获取新的映射信息，而且主机必须从缓存中删除无效的项。由于域名改动并不频繁，大多数网点不需花精力就能维护数据库的一致性。

35、谈下你对 HTTP 长连接和短连接的理解？分别应用于哪些场景?

在 HTTP/1.0 中默认使用短连接。也就是说，客户端和服务器每进行一次 HTTP 操作，就建立一次连接，任务结束就中断连接。当客户端浏览器访问的某个 HTML 或其他类型的 Web 页中包含有其他的 Web 资源（如：JavaScript 文件、图像文件、CSS 文件等），每遇到这样一个 Web 资源，浏览器就会重新建立一个 HTTP 会话。

而从 HTTP/1.1 起，默认使用长连接，用以保持连接特性。使用长连接的 HTTP 协议，会在响应头加入这行代码：

```
Connection:keep-alive
```

在使用长连接的情况下，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。

Keep-Alive 不会永久保持连接，它有一个保持时间，可以在不同的服务器软件（如：Apache）中设定这个时间。实现长连接需要客户端和服务端都支持长连接。

36、HTTPS 的工作过程?

1、 客户端发送自己支持的加密规则给服务器，代表告诉服务器要进行连接了；

2、 服务器从中选出一套加密算法和 hash 算法以及自己的身份信息（地址等）以证书的形式发送给浏览器，证书中包含服务器信息，加密公钥，证书的办法机构；

3、客户端收到网站的证书之后要做下面的事情：

- 验证证书的合法性；
- 果验证通过证书，浏览器会生成一串随机数，并用证书中的公钥进行加密；
- 用约定好的 hash 算法计算握手消息，然后用生成的密钥进行加密，然后一起发送给服务器。

4、服务器接收到客户端传送来的信息，要做下面的事情：

- 4.1 用私钥解析出密码，用密码解析握手消息，验证 hash 值是否和浏览器发来的一致；
- 4.2 使用密钥加密消息；

5、如果计算法 hash 值一致，握手成功。

37、HTTP 和 HTTPS 的区别?

1. 开销：HTTPS 协议需要到 CA 申请证书，一般免费证书很少，需要交费；
2. 资源消耗：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，需要消耗更多的 CPU 和内存资源；
3. 端口不同：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；
4. 安全性：HTTP 的连接很简单，是无状态的；HTTPS 协议是由 TSL+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

38、HTTPS 的优缺点?

优点：

1. 使用 HTTPS 协议可认证用户和服务器，确保数据发送到正确的客户机和服务器；
2. 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，可防止数据在传输过程中不被窃取、改变，确保数据的完整性；
3. HTTPS 是现行架构下最安全的解决方案，虽然不是绝对安全，但它大幅增加了中间人攻击的成本。

缺点：

1. HTTPS 协议握手阶段比较费时，会使页面的加载时间延长近 50%，增加 10% 到 20% 的耗电；
2. HTTPS 连接缓存不如 HTTP 高效，会增加数据开销和功耗，甚至已有的安全措施也会因此而受到影响；
3. SSL 证书需要钱，功能越强大的证书费用越高，个人网站、小网站没有必要一般不会用；
4. SSL 证书通常需要绑定 IP，不能在同一 IP 上绑定多个域名，IPv4 资源不可能支撑这个消耗；
5. HTTPS 协议的加密范围也比较有限，在黑客攻击、拒绝服务攻击、服务器劫持等方面几乎起不到什么作用。最关键的，SSL 证书的信用链体系并不安全，特别是在某些国家可以控制 CA 根证书的情况下，中间人攻击一样可行。

39、什么是数字签名?

为了避免数据在传输过程中被替换，比如黑客修改了你的报文内容，但是你并不知道，所以我们让发送端做一个数字签名，把数据的摘要消息进行一个加密，比如 MD5，得到一个签名，和数据一起发送。然后接收端把数据摘要进行 MD5 加密，如果和签名一样，则说明数据确实是真的。

40、什么是数字证书?

对称加密中，双方使用公钥进行解密。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。

41、Cookie 和 Session 有什么区别?

1、由于HTTP协议是无状态的协议，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，这个机制就是Session.典型的场景比如购物车。

当你点击下单按钮时，由于HTTP协议无状态，所以并不知道是哪个用户操作的，所以服务端要为特定的用户创建了特定的Session，用用于标识这个用户，并且跟踪用户，这样才知道购物车里面有几本书。

这个Session是保存在服务端的，有一个唯一标识。在服务端保存Session的方法很多，内存、数据库、文件都有。集群的时候也要考虑Session的转移，在大型的网站，一般会有专门的Session服务器集群，用来保存用户会话，这个时候 Session 信息都是放在内存的，使用一些缓存服务比如Memcached之类的来放 Session。

2、思考一下服务端如何识别特定的客户？这个时候Cookie就登场了。每次HTTP请求的时候，客户端都会发送相应的Cookie信息到服务端。实际上大多数的应用都是用 Cookie 来实现Session跟踪的，第一次创建Session的时候，服务端会在HTTP协议中告诉客户端，需要在 Cookie 里面记录一个Session ID，以后每次请求把这个会话ID发送到服务器，我就知道你是谁了。

有人问，如果客户端的浏览器禁用了 Cookie 怎么办？一般这种情况下，会使用一种叫做URL重写的技术来进行会话跟踪，即每次HTTP交互，URL后面都会被附加上一个诸如 sid=xxxxx 这样的参数，服务端据此来识别用户。

3、Cookie其实还可以用在一些方便用户的场景下，设想你某次登陆过一个网站，下次登录的时候不想再次输入账号了，怎么办？这个信息可以写到Cookie里面，访问网站的时候，网站页面的脚本可以读取这个信息，就自动帮你把用户名给填了，能够方便一下用户。这也是Cookie名称的由来，给用户的一点甜头。

所以，总结一下：

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、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。

另外，已经把这些整理成 PDF 了，送给大家：500道面试题必知必会（附答案）

整理不容易，如果觉得有帮助，记得给 @帅地 一个赞啊

操作系统

> 你现在手里的这份可能不是最新版，因为帅地看到好的面试题，就会整理进去，强烈建议你去看最新版的，微信搜索关注「**帅地玩编程**」，回复「**1023**」，即可获取最新版的 PDF 哦，扫码直达

关注后，回复「**1023**」，即可获取最新版 PDF。

1、简单说下你对并发和并行的理解?

1. 并行是指两个或者多个事件在同一时刻发生；而并发是指两个或多个事件在同一时间间隔发生；
2. 并行是在不同实体上的多个事件，并发是在同一实体上的多个事件；

2、同步、异步、阻塞、非阻塞的概念

同步：当一个同步调用发出后，调用者要一直等待返回结果。通知后，才能进行后续的执行。

异步：当一个异步过程调用发出后，调用者不能立刻得到返回结果。实际处理这个调用的部件在完成后，通过状态、通知和回调来通知调用者。

阻塞：是指调用结果返回前，当前线程会被挂起，即阻塞。

非阻塞：是指即使调用结果没返回，也不会阻塞当前线程。

3、进程和线程的基本概念

进程：进程是系统进行资源分配和调度的一个独立单位，是系统中的并发执行的单位。

线程：线程是进程的一个实体，也是 CPU 调度和分派的基本单位，它是比进程更小的能独立运行的基本单位，有时又被称为轻权进程或轻量级进程。

4、进程与线程的区别?

1. 进程是资源分配的最小单位，而线程是 CPU 调度的最小单位；
2. 创建进程或撤销进程，系统都要为之分配或回收资源，操作系统开销远大于创建或撤销线程时的开销；
3. 不同进程地址空间相互独立，同一进程内的线程共享同一地址空间。一个进程的线程在另一个进程内是不可见的；
4. 进程间不会相互影响，而一个线程挂掉将可能导致整个进程挂掉；

5、为什么有了进程，还要有线程呢?

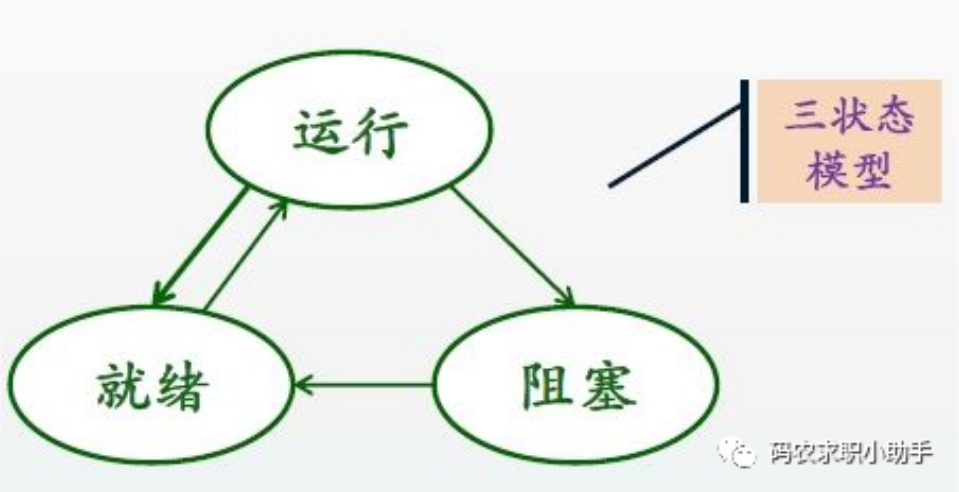
进程可以使多个程序并发执行，以提高资源的利用率和系统的吞吐量，但是其带来了一些缺点：

1. 进程在同一时间只能干一件事情；
2. 进程在执行的过程中如果阻塞，整个进程就会被挂起，即使进程中有些工作不依赖与等待的资源，仍然不会执行。

基于以上的缺点，操作系统引入了比进程粒度更小的线程，作为并发执行的基本单位，从而减少程序在并发执行时所付出的时间和空间开销，提高并发性能。

6、进程的状态转换

进程包括三种状态：就绪态、运行态和阻塞态。



1. 就绪 —> 执行：对就绪状态的进程，当进程调度程序按一种选定的策略从中选中一个就绪进程，为之分配了处理机后，该进程便由就绪状态变为执行状态；
2. 执行 —> 阻塞：正在执行的进程因发生某等待事件而无法执行，则进程由执行状态变为阻塞状态，如进程提出输入/输出请求而变成等待外部设备传输信息的状态，进程申请资源（主存空间或外部设备）得不到满足时变成等待资源状态，进程运行中出现了故障（程序出错或主存储器读写错等）变成等待干预状态等等；
3. 阻塞 —> 就绪：处于阻塞状态的进程，在其等待的事件已经发生，如输入/输出完成，资源得到满足或错误处理完毕时，处于等待状态的进程并不马上转入执行状态，而是先转入就绪状态，然后再由系统进程调度程序在适当的时候将该进程转为执行状态；
4. 执行 —> 就绪：正在执行的进程，因时间片用完而被暂停执行，或在采用抢先式优先级调度算法的系统中,当有更高优先级的进程要运行而被迫让出处理机时，该进程便由执行状态转变为就绪状态。

7、进程间的通信方式有哪些?

进程间通信（IPC，InterProcess Communication）是指在不同进程之间传播或交换信息。IPC 的方式通常有管道（包括无名管道和命名管道）、消息队列、信号量、共享存储、Socket、Streams 等。其中 Socket 和 Streams 支持不同主机上的两个进程 IPC。

管道

1. 它是半双工的，具有固定的读端和写端；
2. 它只能用于父子进程或者兄弟进程之间的进程的通信；
3. 它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 等函数。但是它不是普通的文件，并不属于其他任何文件系统，并且只存在于内存中。

命名管道

1. FIFO 可以在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。

消息队列

1. 消息队列，是消息的链接表，存放在内核中。一个消息队列由一个标识符 ID 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，消息不一定要以先进先出的次序读取，也可以按消息的类型读取。

信号量

1. 信号量（semaphore）是一个计数器。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；
4. 每次对信号量的 PV 操作不仅限于对信号量值加 1 或减 1，而且可以加减任意正整数；
5. 支持信号量组。

共享内存

1. 共享内存（Shared Memory），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 IPC，因为进程是直接对内存进行存取。

8、进程的调度算法有哪些?

调度算法是指：根据系统的资源分配策略所规定的资源分配算法。常用的调度算法有：先来先服务调度算法、时间片轮转调度法、短作业优先调度算法、最短剩余时间优先、高响应比优先调度算法、优先级调度算法等等。

- **先来先服务调度算法**

先来先服务调度算法是一种最简单的调度算法，也称为先进先出或严格排队方案。当每个进程就绪后，它加入就绪队列。当前正运行的进程停止执行，选择在就绪队列中存在时间最长的进程运行。该算法既可以用于作业调度，也可以用于进程调度。先来先去服务比较适合于常作业（进程），而不利于段作业（进程）。

- **时间片轮转调度算法**

时间片轮转调度算法主要适用于分时系统。在这种算法中，系统将所有就绪进程按到达时间的先后次序排成一个队列，进程调度程序总是选择就绪队列中第一个进程执行，即先来先服务的原则，但仅能运行一个时间片。

- **短作业优先调度算法**

短作业优先调度算法是指对短作业优先调度的算法，从后备队列中选择一个或若干个估计运行时间最短的作业，将它们调入内存运行。 短作业优先调度算法是一个非抢占策略，他的原则是下一次选择预计处理时间最短的进程，因此短进程将会越过长作业，跳至队列头。

- **最短剩余时间优先调度算法**

最短剩余时间是针对最短进程优先增加了抢占机制的版本。在这种情况下，进程调度总是选择预期剩余时间最短的进程。当一个进程加入到就绪队列时，他可能比当前运行的进程具有更短的剩余时间，因此只要新进程就绪，调度程序就能可能抢占当前正在运行的进程。像最短进程优先一样，调度程序正在执行选择函数是必须有关于处理时间的估计，并且存在长进程饥饿的危险。

- **高响应比优先调度算法**

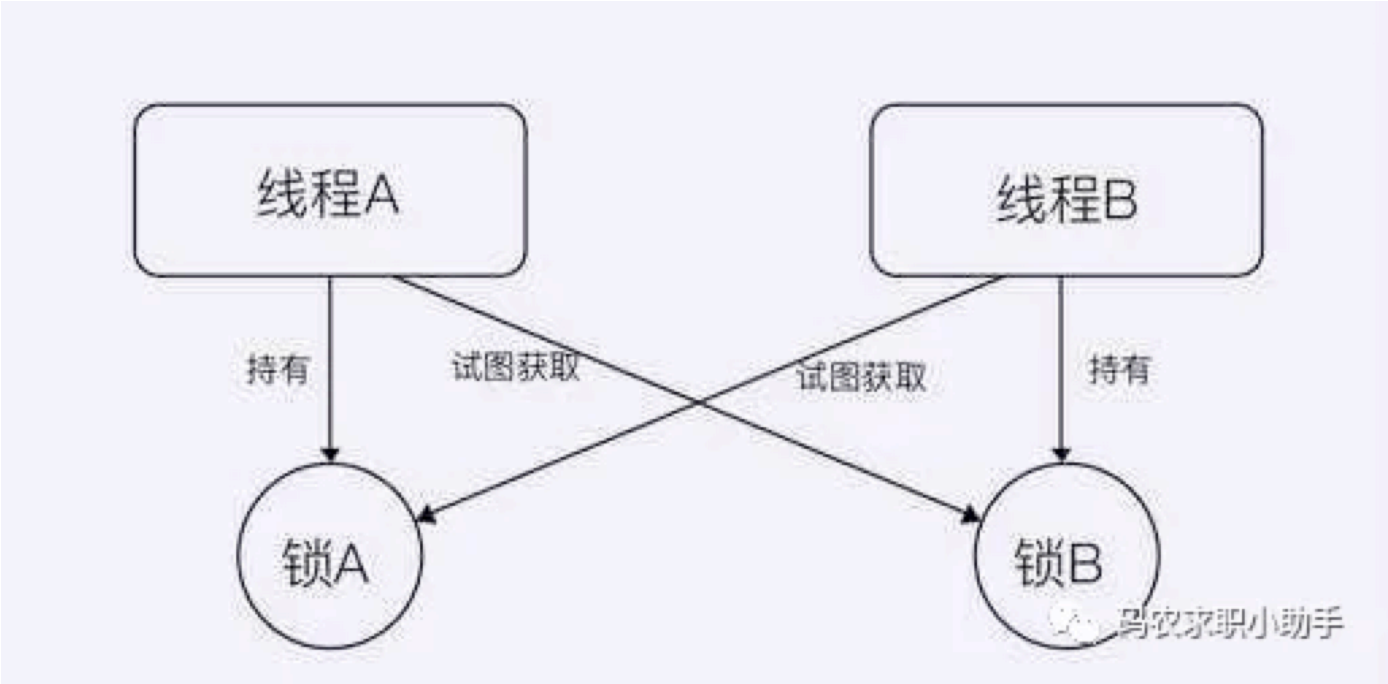
高响应比优先调度算法主要用于作业调度，该算法是对 先来先服务调度算法和短作业优先调度算法的一种综合平衡，同时考虑每个作业的等待时间和估计的运行时间。在每次进行作业调度时，先计算后备作业队列中每个作业的响应比，从中选出响应比最高的作业投入运行。

- **优先级调度算法**

优先级调度算法每次从后备作业队列中选择优先级最高的一个或几个作业，将它们调入内存，分配必要的资源，创建进程并放入就绪队列。在进程调度中，优先级调度算法每次从就绪队列中选择优先级最高的进程，将处理机分配给它，使之投入运行。

9、什么是死锁?

死锁，是指多个进程在运行过程中因争夺资源而造成的一种僵局，当进程处于这种僵持状态时，若无外力作用，它们都将无法再向前推进。 如下图所示：如果此时有一个线程 A，已经持有了锁 A，但是试图获取锁 B，线程 B 持有锁 B，而试图获取锁 A，这种情况下就会产生死锁。



10、产生死锁的原因?

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。

- **竞争资源**

例如：系统中只有一台打印机，可供进程 A 使用，假定 A 已占用了打印机，若 B 继续要求打印机打印将被阻塞。

系统中的资源可以分为两类：

1. 可剥夺资源：是指某进程在获得这类资源后，该资源可以再被其他进程或系统剥夺，CPU 和主存均属于可剥夺性资源；
2. 不可剥夺资源，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，如磁带机、打印机等。

- **进程推进顺序不当**

例如：进程 A 和 进程 B 互相等待对方的数据。

11、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。
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. 环路等待条件：在发生死锁时，必然存在一个进程--资源的环形链。

12、解决死锁的基本方法?

1. 预防死锁
2. 避免死锁
3. 检测死锁
4. 解除死锁

13、怎么预防死锁?

1. 破坏请求条件：一次性分配所有资源，这样就不会再有请求了；
2. 破坏请保持条件：只要有一个资源得不到分配，也不给这个进程分配其他的资源：
3. 破坏不可剥夺条件：当某进程获得了部分资源，但得不到其它资源，则释放已占有的资源；
4. 破坏环路等待条件：系统给每类资源赋予一个编号，每一个进程按编号递增的顺序请求资源，释放则相反。

14、怎么避免死锁?

1. 安全状态

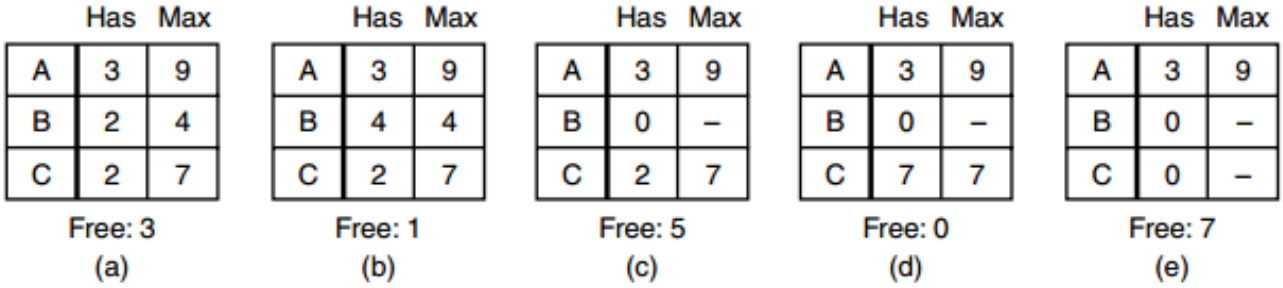


Figure 6-9. Demonstration that the state in (a) is safe.

图 a 的第二列 Has 表示已拥有的资源数，第三列 Max 表示总共需要的资源数，Free 表示还有可以使用的资源数。从图 a 开始出发，先让 B 拥有所需的所有资源（图 b），运行结束后释放 B，此时 Free 变为 5（图 c）；接着以同样的方式运行 C 和 A，使得所有进程都能成功运行，因此可以称图 a 所示的状态时安全的。

定义：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，可以结合着做参考对比。

2. 单个资源的银行家算法

一个小城镇的银行家，他向一群客户分别承诺了一定的贷款额度，算法要做的是判断对请求的满足是否会进入不安全状态，如果是，就拒绝请求；否则予以分配。

	Has	Max
A	0	6
B	0	5
C	0	4
D	0	7

Free: 10

(a)

	Has	Max
A	1	6
B	1	5
C	2	4
D	4	7

Free: 2

(b)

	Has	Max
A	1	6
B	2	5
C	2	4
D	4	7

Free: 1

(c)

Figure 6-11. Three resource allocation states: (a) Safe. (b) Safe. (c) Unsafe.

上图 c 为不安全状态，因此算法会拒绝之前的请求，从而避免进入图 c 中的状态。

3. 多个资源的银行家算法

Process	Tape drives	Plotters	Printers	Blu-rays
A	3	0	1	1
B	0	1	0	0
C	1	1	1	0
D	1	1	0	1
E	0	0	0	0

Resources assigned

Process	Tape drives	Plotters	Printers	Blu-rays
A	1	1	0	0
B	0	1	1	2
C	3	1	0	0
D	0	0	1	0
E	2	1	1	0

Resources still assigned

E = (6342)
P = (5322)
A = (1020)

Figure 6-12. The banker's algorithm with multiple resources.

上图中有五个进程，四个资源。左边的图表示已经分配的资源，右边的图表示还需要分配的资源。最右边的 E、P 以及 A 分别表示：总资源、已分配资源以及可用资源，注意这三个为向量，而不是具体数值，例如 A=(1020)，表示 4 个资源分别还剩下 1/0/2/0。

检查一个状态是否安全的算法如下：

- 查找右边的矩阵是否存在一行小于等于向量 A。如果不存在这样的行，那么系统将会发生死锁，状态是不安全的。
- 假若找到这样一行，将该进程标记为终止，并将其已分配资源加到 A 中。
- 重复以上两步，直到所有进程都标记为终止，则状态时安全的。

如果一个状态不是安全的，需要拒绝进入这个状态。

15、怎么解除死锁?

1. 资源剥夺：挂起某些死锁进程，并抢占它的资源，将这些资源分配给其他死锁进程（但应该防止被挂起的进程长时间得不到资源）；
2. 撤销进程：强制撤销部分、甚至全部死锁进程并剥夺这些进程的资源（撤销的原则可以按进程优先级和撤销进程代价的高低进行）；
3. 进程回退：让一个或多个进程回退到足以避免死锁的地步。进程回退时自愿释放资源而不是被剥夺。要求系统保持进程的历史信息，设置还原点。

16、什么是缓冲区溢出？有什么危害?

缓冲区为暂时置放输出或输入资料的内存。缓冲区溢出是指当计算机向缓冲区填充数据时超出了缓冲区本身的容量，溢出的数据覆盖在合法数据上。造成缓冲区溢出的主要原因是程序中没有仔细检查用户输入是否合理。计算机中，缓冲区溢出会造成的危害主要有以下两点：程序崩溃导致拒绝服务和跳转并且执行一段恶意代码。

17、分页与分段的区别?

1. 段是信息的逻辑单位，它是根据用户的需要划分的，因此段对用户是可见的 ；页是信息的物理单位，是为了管理主存的方便而划分的，对用户是透明的；
2. 段的大小不固定，有它所完成的功能决定；页大大小固定，由系统决定；
3. 段向用户提供二维地址空间；页向用户提供的是一维地址空间；
4. 段是信息的逻辑单位，便于存储保护和信息的共享，页的保护和共享受到限制。

18、物理地址、逻辑地址、虚拟内存的概念

1. 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。
2. 逻辑地址：是指计算机用户看到的地址。例如：当创建一个长度为 100 的整型数组时，操作系统返回一个逻辑上的连续空间：指针指向数组第一个元素的内存地址。由于整型元素的大小为 4 个字节，故第二个元素的地址时起始地址加 4，以此类推。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（在内存条中所处的位置），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。
3. 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（一个连续完整的地址空间），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。

19、页面置换算法有哪些?

请求调页，也称按需调页，即对不在内存中的“页”，当进程执行时要用时才调入，否则有可能到程序结束时也不会调入。而内存中给页面留的位置是有限的，在内存中以帧为单位放置页面。为了防止请求调页的过程出现过多的内存页面错误（即需要的页面当前不在内存中，需要从硬盘中读数据，也即需要做页面的替换）而使得程序执行效率下降，我们需要设计一些页面置换算法，页面按照这些算法进行相互替换时，可以尽量达到较低的错误率。常用的页面置换算法如下：

- **先进先出置换算法（FIFO）**

先进先出，即淘汰最早调入的页面。

- **最佳置换算法（OPT）**

选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。

- **最近最久未使用（LRU）算法**

即选择最近最久未使用的页面予以淘汰

- **时钟（Clock）置换算法**

时钟置换算法也叫最近未用算法 NRU（Not RecentlyUsed）。该算法为每个页面设置一位访问位，将内存中的所有页面都通过链接指针链成一个循环队列。

20、谈谈你对动态链接库和静态链接库的理解?

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需要的依赖库，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

21、外中断和异常有什么区别?

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。

而异常时由 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。

22、一个程序从开始运行到结束的完整过程，你能说出来多少?

四个过程：

（1）预编译 主要处理源代码文件中的以“#”开头的预编译指令。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 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(Linux下)、xxx.obj(Windows下)。

（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：

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份副本，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。

23、介绍一下几种典型的锁?

读写锁

- 多个读者可以同时进行读
- 写者必须互斥（只允许一个写者写，也不能读者写者同时进行）
- 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）

互斥锁

一次只能一个线程拥有互斥锁，其他线程只有等待

互斥锁是在抢锁失败的情况下主动放弃CPU进入睡眠状态直到锁的状态改变时再唤醒，而操作系统负责线程调度，为了实现锁的状态发生改变时唤醒阻塞的线程或者进程，需要把锁交给操作系统管理，所以互斥锁在加锁操作时涉及上下文的切换。互斥锁实际的效率还是可以让人接受的，加锁的时间大概100ns左右，而实际上互斥锁的一种可能的实现是先自旋一段时间，当自旋的时间超过阀值之后再将线程投入睡眠中，因此在并发运算中使用互斥锁（每次占用锁的时间很短）的效果可能不亚于使用自旋锁

条件变量

互斥锁一个明显的缺点是他只有两种状态：锁定和非锁定。而条件变量通过允许线程阻塞和等待另一个线程发送信号的方法弥补了互斥锁的不足，他常和互斥锁一起使用，以免出现竞态条件。当条件不满足时，线程往往解开相应的互斥锁并阻塞线程然后等待条件发生变化。一旦其他的某个线程改变了条件变量，他将通知相应的条件变量唤醒一个或多个正被此条件变量阻塞的线程。总的来说**互斥锁是线程间互斥的机制，条件变量则是同步机制。**

自旋锁

如果进线程无法取得锁，进线程不会立刻放弃CPU时间片，而是一直循环尝试获取锁，直到获取为止。如果别的线程长时期占有锁，那么自旋就是在浪费CPU做无用功，但是自旋锁一般应用于加锁时间很短的场景，这个时候效率比较高。

24、什么是用户态和内核态

用户态和内核态是操作系统的两种运行状态。

- `内核态`：处于内核态的 CPU 可以访问任意的数据，包括外围设备，比如网卡、硬盘等，处于内核态的 CPU 可以从一个程序切换到另外一个程序，并且占用 CPU 不会发生抢占情况，一般处于特权级 0 的状态我们称之为内核态。
- `用户态`：处于用户态的 CPU 只能受限的访问内存，并且不允许访问外围设备，用户态下的 CPU 不允许独占，也就是说 CPU 能够被其他程序获取。

> 那么为什么要有用户态和内核态呢?

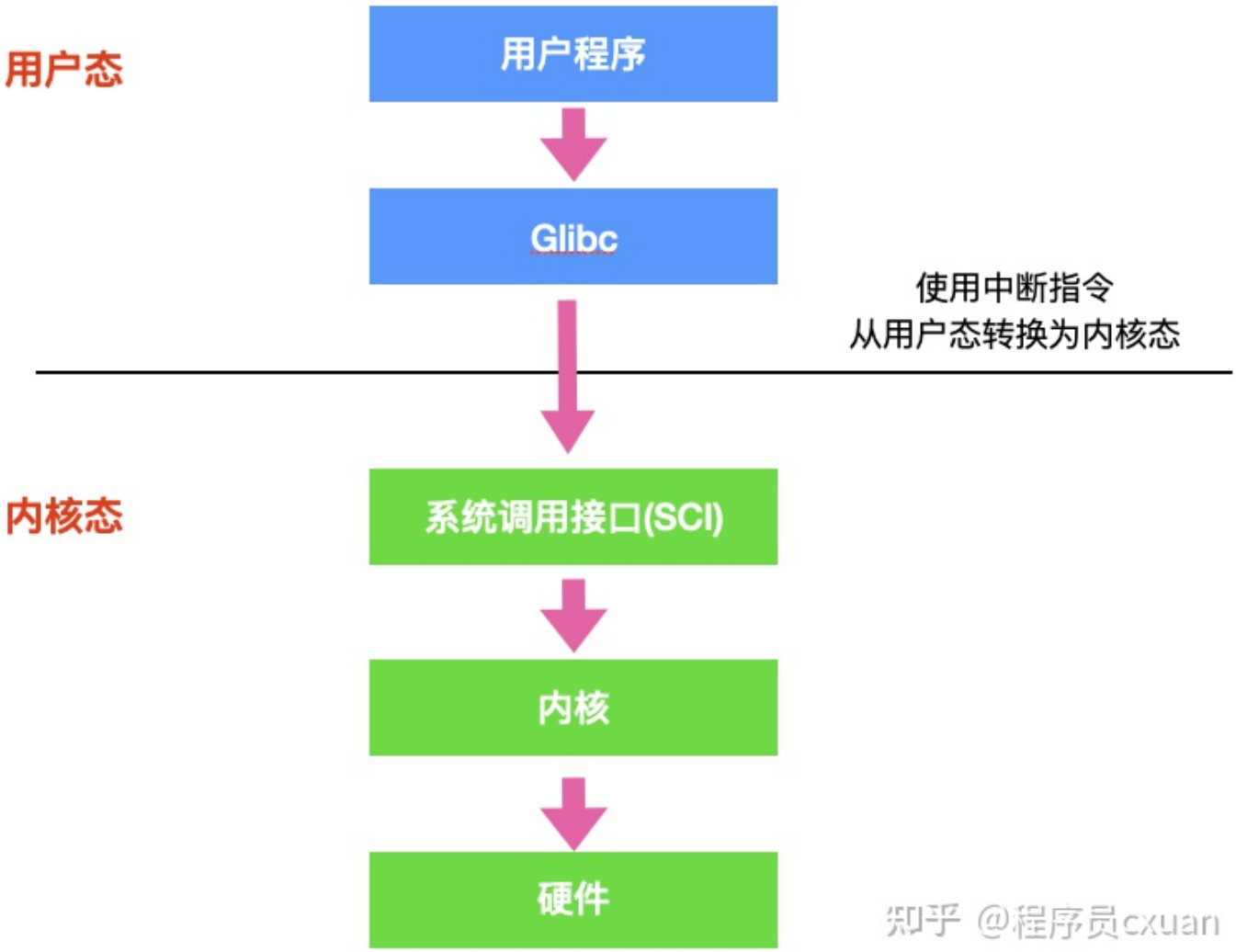
这个主要是访问能力的限制的考量，计算机中有一些比较危险的操作，比如设置时钟、内存清理，这些都需要在内核态下完成，如果随意进行这些操作，那你的系统得崩溃多少次。

25、用户态和内核态是如何切换的?

所有的用户进程都是运行在用户态的，但是我们上面也说了，用户程序的访问能力有限，一些比较重要的比如从硬盘读取数据，从键盘获取数据的操作则是内核态才能做的事情，而这些数据却又对用户程序来说非常重要。所以就涉及到两种模式下的转换，即**用户态 -> 内核态 -> 用户态**，而唯一能够做这些操作的只有 `系统调用`，而能够执行系统调用的就只有 `操作系统`。

一般用户态 -> 内核态的转换我们都称之为 trap 进内核，也被称之为 `陷阱指令(trap instruction)`。

他们的工作流程如下：



- 首先用户程序会调用 `glibc` 库，glibc 是一个标准库，同时也是一套核心库，库中定义了很多关键 API。
- glibc 库知道针对不同体系结构调用 `系统调用` 的正确方法，它会根据体系结构应用程序的二进制接口设置用户进程传递的参数，来准备系统调用。
- 然后，glibc 库调用 `软件中断指令(SWI)` ，这个指令通过更新 `CPSR` 寄存器将模式改为超级用户模式，然后跳转到地址 `0x08` 处。
- 到目前为止，整个过程仍处于用户态下，在执行 SWI 指令后，允许进程执行内核代码，MMU 现在允许内核虚拟内存访问
- 从地址 0x08 开始，进程执行加载并跳转到中断处理程序，这个程序就是 ARM 中的 `vector_swi()` 。
- 在 vector_swi() 处，从 SWI 指令中提取系统调用号 SCNO，然后使用 SCNO 作为系统调用表 `sys_call_table` 的索引，调转到系统调用函数。
- 执行系统调用完成后，将还原用户模式寄存器，然后再以用户模式执行。

26、进程终止的方式

进程的终止

进程在创建之后，它就开始运行并做完成任务。然而，没有什么事儿是永不停歇的，包括进程也一样。进程早晚会发生终止，但是通常是由于以下情况触发的

- `正常退出(自愿的)`
- `错误退出(自愿的)`
- `严重错误(非自愿的)`

- `被其他进程杀死(非自愿的)`

正常退出

多数进程是由于完成了工作而终止。当编译器完成了所给定程序的编译之后，编译器会执行一个系统调用告诉操作系统它完成了工作。这个调用在 UNIX 中是 `exit` ，在 Windows 中是 `ExitProcess` 。面向屏幕中的软件也支持自愿终止操作。字处理软件、Internet 浏览器和类似的程序中总有一个供用户点击的图标或菜单项，用来通知进程删除它锁打开的任何临时文件，然后终止。

错误退出

进程发生终止的第二个原因是发现严重错误，例如，如果用户执行如下命令

```
cc foo.c
```

为了能够编译 foo.c 但是该文件不存在，于是编译器就会发出声明并退出。在给出了错误参数时，面向屏幕的交互式进程通常并不会直接退出，因为这从用户的角度来说并不合理，用户需要知道发生了什么并想要进行重试，所以这时候应用程序通常会弹出一个对话框告知用户发生了系统错误，是需要重试还是退出。

严重错误

进程终止的第三个原因是由进程引起的错误，通常是由于程序中的错误所导致的。例如，执行了一条非法指令，引用不存在的内存，或者除数是 0 等。在有些系统比如 UNIX 中，进程可以通知操作系统，它希望自行处理某种类型的错误，在这类错误中，进程会收到信号（中断），而不是在这类错误出现时直接终止进程。

被其他进程杀死

第四个终止进程的原因是，某个进程执行系统调用告诉操作系统杀死某个进程。在 UNIX 中，这个系统调用是 kill。在 Win32 中对应的函数是 `TerminateProcess`（注意不是系统调用）。

27、 守护进程、僵尸进程和孤儿进程

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服务器进程http等

创建守护进程要点：

（1）让程序在后台执行。方法是调用fork（）产生一个子进程，然后使父进程退出。

（2）调用setsid（）创建一个新对话期。控制终端、登录会话和进程组通常是从父进程继承下来的，守护进程要摆脱它们，不受它们的影响，方法是调用setsid（）使进程成为一个会话组长。setsid（）调用成功后，进程成为新的会话组长和进程组长，并与原来的登录会话、进程组和控制终端脱离。

（3）禁止进程重新打开控制终端。经过以上步骤，进程已经成为一个无终端的会话组长，但是它可以重新申请打开一个终端。为了避免这种情况发生，可以通过使进程不再是会话组长来实现。再一次通过fork（）创建新的子进程，使调用fork的进程退出。

（4）关闭不再需要的文件描述符。子进程从父进程继承打开的文件描述符。如不关闭，将会浪费系统资源，造成进程所在的文件系统无法卸下以及引起无法预料的错误。首先获得最高文件描述符值，然后用一个循环程序，关闭0到最高文件描述符值的所有文件描述符。

（5）将当前目录更改为根目录。

（6）子进程从父进程继承的文件创建屏蔽字可能会拒绝某些许可权。为防止这一点，使用unmask（0）将屏蔽字清零。

（7）处理SIGCHLD信号。对于服务器进程，在请求到来时往往生成子进程处理请求。如果子进程等待父进程捕获状态，则子进程将成为僵尸进程（zombie），从而占用系统资源。如果父进程等待子进程结束，将增加父进程的负担，影响服务器进程的并发性能。在Linux下可以简单地将SIGCHLD信号的操作设为SIG_IGN。这样，子进程结束时不会产生僵尸进程。

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。

僵尸进程

如果子进程先退出，父进程还没退出，那么子进程必须等到父进程捕获到了子进程的退出状态才真正结束，否则这个时候子进程就成为僵尸进程。

设置**僵尸进程的目**的是维护子进程的信息，以便父进程在以后某个时候获取。这些信息至少包括进程ID，进程的终止状态，以及该进程使用的CPU时间，所以当终止子进程的父进程调用wait或waitpid时就可以得到这些信息。如果一个进程终止，而该进程有子进程处于僵尸状态，那么它的所有僵尸子进程的父进程ID将被重置为1（init进程）。继承这些子进程的init进程将清理它们（也就是说init进程将wait它们，从而去除它们的僵尸状态）。

28、如何避免僵尸进程?

- 通过signal(SIGCHLD, SIG_IGN)通知内核对子进程的结束不关心，由内核回收。如果不想让父进程挂起，可以在父进程中加入一条语句：signal(SIGCHLD,SIG_IGN);表示父进程忽略SIGCHLD信号，该信号是子进程退出的时候向父进程发送的。
- 父进程调用wait/waitpid等函数等待子进程结束，如果尚无子进程退出wait会导致父进程阻塞。waitpid可以通过传递WNOHANG使父进程不阻塞立即返回。
- 如果父进程很忙可以用signal注册信号处理函数，在信号处理函数调用wait/waitpid等待子进程退出。
- 通过两次调用fork。父进程首先调用fork创建一个子进程然后waitpid等待子进程退出，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。

第一种方法忽略SIGCHLD信号，这常用于并发服务器的性能的一个技巧因为并发服务器常常fork很多子进程，子进程终结之后需要服务器进程去wait清理资源。如果将此信号的处理方式设为忽略，可让内核把僵尸子进程转交给init进程去处理，省去了大量僵尸进程占用系统资源。

29、常见内存分配内存错误

（1）内存分配未成功，却使用了它。

编程新手常犯这种错误，因为他们没有意识到内存分配会不成功。常用解决办法是，在使用内存之前检查指针是否为NULL。如果指针p是函数的参数，那么在函数的入口处用assert(p!=NULL)进行检查。如果是用malloc或new来申请内存，应该用if(p==NULL) 或if(p!=NULL)进行防错处理。

（2）内存分配虽然成功，但是尚未初始化就引用它。

犯这种错误主要有两个起因：一是没有初始化的观念；二是误以为内存的缺省初值全为零，导致引用初值错误（例如数组）。内存的缺省初值究竟是什么并没有统一的标准，尽管有些时候为零值，我们宁可信其无不可信其有。所以无论用何种方式创建数组，都别忘了赋初值，即便是赋零值也不可省略，不要嫌麻烦。

（3）内存分配成功并且已经初始化，但操作越过了内存的边界。

例如在使用数组时经常发生下标“多1”或者“少1”的操作。特别是在for循环语句中，循环次数很容易搞错，导致数组操作越界。

（4）忘记了释放内存，造成内存泄露。

含有这种错误的函数每被调用一次就丢失一块内存。刚开始时系统的内存充足，你看不到错误。终有一次程序突然挂掉，系统出现提示：内存耗尽。动态内存的申请与释放必须配对，程序中malloc与free的使用次数一定要相同，否则肯定有错误（new/delete同理）。

（5）释放了内存却继续使用它。常见于以下有三种情况：

- 程序中的对象调用关系过于复杂，实在难以搞清楚某个对象究竟是否已经释放了内存，此时应该重新设计数据结构，从根本上解决对象管理的混乱局面。
- 函数的return语句写错了，注意不要返回指向“栈内存”的“指针”或者“引用”，因为该内存在函数体结束时被自动销毁。
- 使用free或delete释放了内存后，没有将指针设置为NULL。导致产生“野指针”。

30、内存交换中，被换出的进程保存在哪里?

保存在磁盘中，也就是外存中。具有对换功能的操作系统中，通常把磁盘空间分为文件区和对换区两部分。文件区主要用于存放文件，主要追求存储空间的利用率，因此对文件区空间的管理采用离散分配方式;对换区空间只占磁盘空间的小部分，被换出的进程数据就存放在对换区。由于对换的速度直接影响到系统的整体速度，因此对换区空间的管理主要追求换入换出速度，因此通常对换区采用连续分配方式(学过文件管理章节后即可理解)。总之，对换区的I/O速度比文件区的更快。

31、原子操作的是如何实现的

处理器使用基于对缓存加锁或总线加锁的方式来实现多处理器之间的原子操作。首先处理器会自动保证基本的内存操作的原子性。处理器保证从系统内存中读取或者写入一个字节是原子的，意思是当一个处理器读取一个字节时，其他处理器不能访问这个字节的内存地址。Pentium 6和最新的处理器能自动保证单处理器对同一个缓存行里进行16/32/64位的操作是原子的，但是复杂的内存操作处理器是不能自动保证其原子性的，比如跨总线宽度、跨多个缓存行和跨页表的访问。但是，处理器提供总线锁定和缓存锁定两个机制来保证复杂内存操作的原子性。

（1）使用总线锁保证原子性 第一个机制是通过总线锁保证原子性。如果多个处理器同时对共享变量进行读改写操作（i++就是经典的读改写操作），那么共享变量就会被多个处理器同时进行操作，这样读改写操作就不是原子的，操作完之后共享变量的值会和期望的不一致。举个例子，如果i=1，我们进行两次i++操作，我们期望的结果是3，但是有可能结果是2，如图下图所示。

```
CPU1    CPU2
 i=1     i=1
 i+1     i+1
 i=2     i=2Copy to clipboardErrorCopied
```

原因可能是多个处理器同时从各自的缓存中读取变量i，分别进行加1操作，然后分别写入系统内存中。那么，想要保证读改写共享变量的操作是原子的，就必须保证CPU1读改写共享变量的时候，CPU2不能操作缓存了该共享变量内存地址的缓存。

处理器使用总线锁就是来解决这个问题的。**所谓总线锁就是使用处理器提供的一个LOCK＃信号，当一个处理器在总线上输出此信号时，其他处理器的请求将被阻塞住，那么该处理器可以独占共享内存。**

（2）使用缓存锁保证原子性 第二个机制是通过缓存锁定来保证原子性。在同一时刻，我们只需保证对某个内存地址的操作是原子性即可，但**总线锁定把CPU和内存之间的通信锁住了**，这使得锁定期间，其他处理器不能操作其他内存地址的数据，所以总线锁定的开销比较大，目前处理器在某些场合下使用缓存锁定代替总线锁定来进行优化。

频繁使用的内存会缓存在处理器的L1、L2和L3高速缓存里，那么原子操作就可以直接在处理器内部缓存中进行，并不需要声明总线锁，在Pentium 6和目前的处理器中可以使用“缓存锁定”的方式来实现复杂的原子性。

所谓“缓存锁定”是指内存区域如果被缓存在处理器的缓存行中，并且在Lock操作期间被锁定，那么当它执行锁操作回写到内存时，处理器不在总线上声言LOCK＃信号，而是修改内部的内存地址，并允许它的缓存一致性机制来保证操作的原子性，因为**缓存一致性机制会阻止同时修改由两个以上处理器缓存的内存区域数据，当其他处理器回写已被锁定的缓存行的数据时，会使缓存行无效，在如上图所示的例子中，当CPU1修改缓存行中的i时使用了缓存锁定，那么CPU2就不能使用同时缓存i的缓存行。**

但是有两种情况下处理器不会使用缓存锁定。 第一种情况是：当操作的数据不能被缓存在处理器内部，或操作的数据跨多个缓存行（cache line）时，则处理器会调用总线锁定。 第二种情况是：有些处理器不支持缓存锁定。对于Intel 486和Pentium处理器，就算锁定的内存区域在处理器的缓存行中也会调用总线锁定。

32、抖动你知道是什么吗？它也叫颠簸现象

刚刚换出的页面马上又要换入内存，刚刚换入的页面马上又要换出外存，这种频繁的页面调度行为称为抖动，或颠簸。产生抖动的主要原因是进程频繁访问的页面数目高于可用的物理块数(分配给进程的物理块不够)

为进程分配的物理块太少，会使进程发生抖动现象。为进程分配的物理块太多，又会降低系统整体的并发度，降低某些资源的利用率 为了研究为应该为每个进程分配多少个物理块，Denning 提出了进程工作集” 的概念

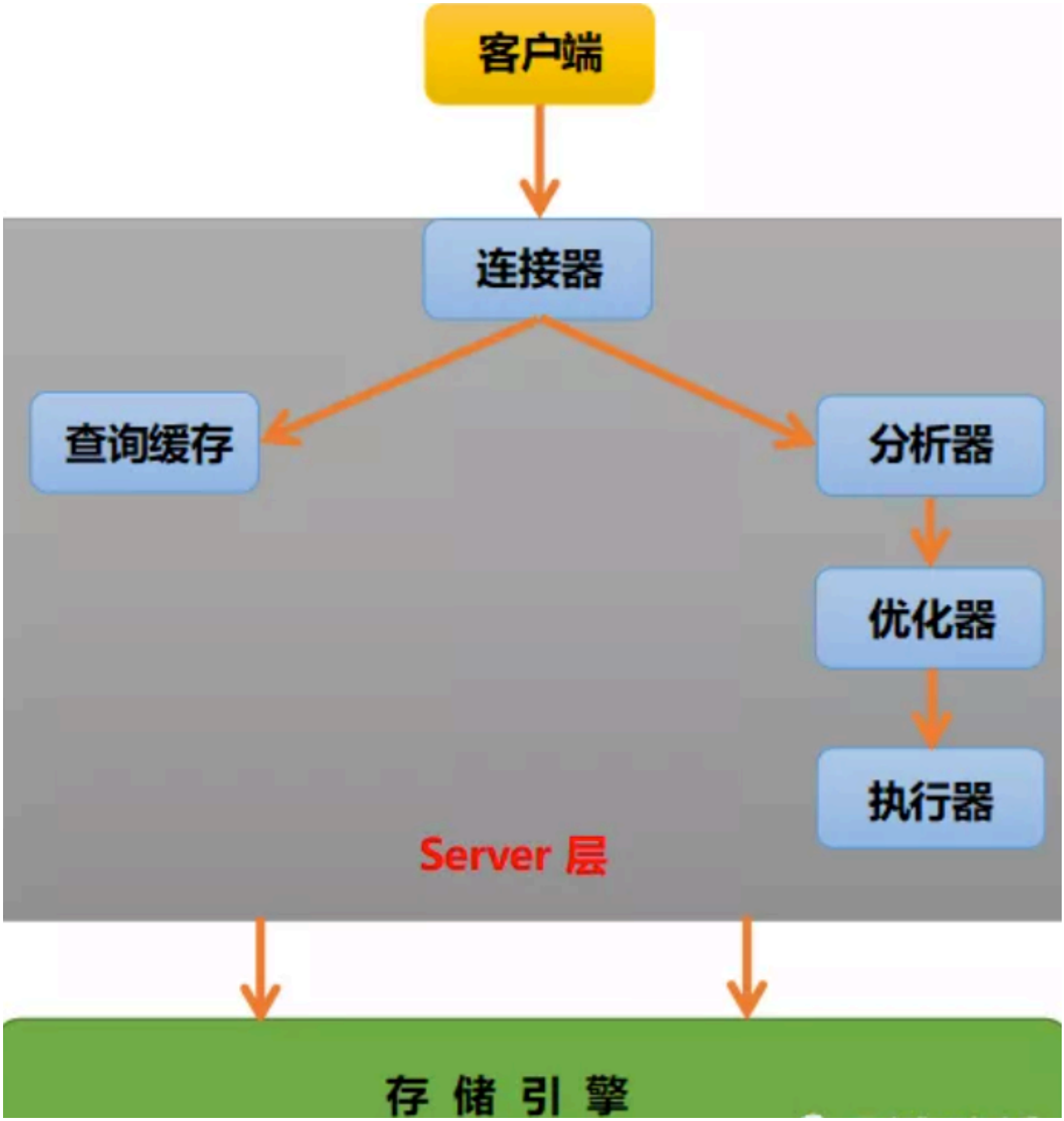
MySQL

> 你现在手里的这份可能不是最新版，因为帅地看到好的面试题，就会整理进去，强烈建议你去看最新版的，
> 微信搜索关注「**帅地玩编程**」，回复「**1023**」，即可获取最新版的 PDF 哦，扫码直达

关注后，回复「**1023**」，即可获取最新版 PDF。

1、请说下你对 MySQL 架构的了解?

- 先看下 MySQL 的基本架构图：



大体来说，MySQL 可以分为 Server 层和存储引擎两部分。

Server 层包括：连接器、查询缓存、分析器、优化器、执行器等，涵盖了 MySQL 的大多数核心服务功能，以及所有的内置函数（如：日期、时间、数学和加密函数等），所有跨存储引擎的功能都在这一层实现，比如：存储过程、触发器、视图等等。

存储引擎层负责：数据的存储和提取。其架构是插件式的，支持 InnoDB、MyISAM 等多个存储引擎。从 MySQL5.5.5 版本开始默认的是InnoDB，但是在建表时可以通过 engine = MyISAM 来指定存储引擎。不同存储引擎的表数据存取方式不同，支持的功能也不同。

从上图中可以看出，不同的存储引擎共用一个 Server 层，也就是从连接器到执行器的部分。

2、一条 SQL 语句在数据库框架中的执行流程?

1. 应用程序把查询 SQL 语句发送给服务器端执行；
2. 查询缓存，如果查询缓存是打开的，服务器在接收到查询请求后，并不会直接去数据库查询，而是在数据库的查询缓存中找是否有相对应的查询数据，如果存在，则直接返回给客户端。只有缓存不存在时，才会进行下面的操作；
3. 查询优化处理，生成执行计划。这个阶段主要包括解析 SQL、预处理、优化 SQL 执行计划；
4. MySQL 根据相应的执行计划完成整个查询；
5. 将查询结果返回给客户端。

详细过程可以看这篇博客https://blog.csdn.net/pcwl1206/article/details/86137408

3、数据库的三范式是什么?

1. 第一范式：强调的是列的原子性，即数据库表的每一列都是不可分割的原子数据项；
2. 第二范式：要求实体的属性完全依赖于主关键字。所谓完全依赖是指不能存在仅依赖主关键字一部分的属性；
3. 第三范式：任何非主属性不依赖于其它非主属性。

4、char 和 varchar 的区别?

char(n)： 固定长度类型，比如：订阅 char(10)，当你输入"abc"三个字符的时候，它们占的空间还是 10 个字节，其他 7 个是空字节。char 优点：效率高；缺点：占用空间；适用场景：存储密码的 md5 值，固定长度的，使用 char 非常合适。

varchar(n)： 可变长度，存储的值是每个值占用的字节再加上一个用来记录其长度的字节的长度。

所以，从空间上考虑 varcahr 比较合适；从效率上考虑 char 比较合适，二者使用需要权衡。

5、varchar(10) 和 varchar(20) 的区别?

varchar(10) 中 10 的涵义最多存放 10 个字符，varchar(10) 和 varchar(20) 存储 hello 所占空间一样，但后者在排序时会消耗更多内存，因为 order by col 采用 fixed_length 计算 col 长度。

6、谈谈你对索引的理解?

索引的出现是为了提高数据的查询效率，就像书的目录一样。一本500页的书，如果你想快速找到其中的某一个知识点，在不借助目录的情况下，那我估计你可得找一会儿。同样，对于数据库的表而言，索引其实就是它的“目录”。

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。

建立索引的原则：

1. 在最频繁使用的、用以缩小查询范围的字段上建立索引；

2. 在频繁使用的、需要排序的字段上建立索引。

不适合建立索引的情况：

1. 对于查询中很少涉及的列或者重复值比较多的列，不宜建立索引；
2. 对于一些特殊的数据类型，不宜建立索引，比如：文本字段（text）等。

7、索引的底层使用的是什么数据结构?

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。而我们经常使用的 InnoDB 存储引擎的默认索引实现为 B+ 树索引。

8、谈谈你对 B+ 树的理解?

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区间查询的性能。
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。

9、为什么 InnoDB 存储引擎选用 B+ 树而不是 B 树呢?

用 B+ 树不用 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）。

10、谈谈你对聚簇索引的理解?

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。

11、谈谈你对哈希索引的理解?

哈希索引能以 O(1) 时间进行查找，但是失去了有序性。无法用于排序与分组、只支持精确查找，无法用于部分查找和范围查找。

InnoDB 存储引擎有一个特殊的功能叫“自适应哈希索引”，当某个索引值被使用的非常频繁时，会在 B+ 树索引之上再创建一个哈希索引，这样就让 B+Tree 索引具有哈希索引的一些优点，比如：快速的哈希查找。

12、谈谈你对覆盖索引的认识?

如果一个索引包含了满足查询语句中字段与条件的数据就叫做覆盖索引。具有以下优点：

1. 索引通常远小于数据行的大小，只读取索引能大大减少数据访问量。
2. 一些存储引擎（例如：MyISAM）在内存中只缓存索引，而数据依赖于操作系统来缓存。因此，只访问索引可以不使用系统调用（通常比较费时）。
3. 对于 InnoDB 引擎，若辅助索引能够覆盖查询，则无需访问主索引。

13、索引的分类?

- **从数据结构角度**

1. 树索引 (O(log(n)))
2. Hash 索引

从物理存储角度

1. 聚集索引（clustered index）
2. 非聚集索引（non-clustered index）

从逻辑角度

1. 普通索引
2. 唯一索引
3. 主键索引
4. 联合索引
5. 全文索引

13、谈谈你对最左前缀原则的理解?

MySQL 使用联合索引时，需要满足最左前缀原则。下面举例对其进行说明：

```
1. 一个 2 列的索引（name, age），对（name）、（name, age）上建立了索引；
2. 一个 3 列的索引（name, age, sex），对（name）、（name, age）、（name, age, sex）上建立了索引。
```

1、 B+ 树的数据项是复合的数据结构，比如：(name, age, sex) 的时候，B+ 树是按照从左到右的顺序来建立搜索树的，比如：当(小明, 22, 男)这样的数据来检索的时候，B+ 树会优先比较 name 来确定下一步的所搜方向，如果 name 相同再依次比较 age 和 sex，最后得到检索的数据。但当 (22, 男) 这样没有 name 的数据来的时候，B+ 树就不知道第一步该查哪个节点，因为建立搜索树的时候 name 就是第一个比较因子，必须要先根据 name 来搜索才能知道下一步去哪里查询。

2、 当 (小明, 男) 这样的数据来检索时，B+ 树可以用 name 来指定搜索方向，但下一个字段 age 的缺失，所以只能把名字等于小明的数据都找到，然后再匹配性别是男的数据了， 这个是非常重要的性质，即索引的最左匹配特性。

关于最左前缀的补充：

1. 最左前缀匹配原则会一直向右匹配直到遇到范围查询（>、<、between、like）就停止匹配，比如：a = 1 and b = 2 and c > 3 and d = 4 如果建立 (a, b, c, d) 顺序的索引，d 是用不到索引的。如果建立 (a, b, d, c) 的索引则都可以用到，a、b、d 的顺序可以任意调整。
2. = 和 in 可以乱序，比如：a = 1 and b = 2 and c = 3 建立 (a, b ,c) 索引可以任意顺序，MySQL 的优化器会优化

成索引可以识别的形式。

14、怎么知道创建的索引有没有被使用到？或者说怎么才可以知道这条语句运行很慢的原因？

使用 Explain 命令来查看语句的执行计划，MySQL 在执行某个语句之前，会将该语句过一遍查询优化器，之后会拿到对语句的分析，也就是执行计划，其中包含了许多信息。可以通过其中和索引有关的信息来分析是否命中了索引，例如：possilbe_key、key、key_len 等字段，分别说明了此语句可能会使用的索引、实际使用的索引以及使用的索引长度。

16、什么情况下索引会失效？即查询不走索引？

下面列举几种不走索引的 SQL 语句：

1、索引列参与表达式计算：

```
SELECT 'sname' FROM 'stu' WHERE 'age' + 10 = 30;
```

2、 函数运算：

```
SELECT 'sname' FROM 'stu' WHERE LEFT('date',4) < 1990;
```

3、%词语%--模糊查询：

```
SELECT * FROM 'manong' WHERE `uname` LIKE '%码农%' -- 走索引

SELECT * FROM 'manong' WHERE `uname` LIKE "%码农%" -- 不走索引
```

4、 字符串与数字比较不走索引：

```
CREATE TABLE 'a' ('a' char(10));
EXPLAIN SELECT * FROM 'a' WHERE 'a'="1" -- 走索引
EXPLAIN SELECT * FROM 'a'WHERE 'a'=1 -- 不走索引，同样也是使用了函数运算
```

5、 查询条件中有 or ，即使其中有条件带索引也不会使用。换言之，就是要求使用的所有字段，都必须建立索引：

```
select * from dept where dname='xxx' or loc='xx' or deptno = 45;
```

6、正则表达式不使用索引。

7、 MySQL 内部优化器会对 SQL 语句进行优化，如果优化器估计使用全表扫描要比使用索引快，则不使用索引。

16、查询性能的优化方法?

减少请求的数据量

1. 只返回必要的列：最好不要使用 SELECT * 语句。
2. 只返回必要的行：使用 LIMIT 语句来限制返回的数据。
3. 缓存重复查询的数据：使用缓存可以避免在数据库中进行查询，特别在要查询的数据经常被重复查询时，缓存带来的查询性能提升将会是非常明显的。

减少服务器端扫描的行数

1. 最有效的方式是使用索引来覆盖查询。

17、InnoDB 和 MyISAM 的比较?

1. 事务：MyISAM不支持事务，InnoDB支持事务；
2. 全文索引：MyISAM 支持全文索引，InnoDB 5.6 之前不支持全文索引；
3. 关于 count*()：MyISAM会直接存储总行数，InnoDB 则不会，需要按行扫描。意思就是对于 select count()* from table; 如果数据量大，MyISAM 会瞬间返回，而 InnoDB 则会一行行扫描；
4. 外键：MyISAM 不支持外键，InnoDB 支持外键；
5. 锁：MyISAM 只支持表锁，InnoDB 可以支持行锁。

18、谈谈你对水平切分和垂直切分的理解?

- **水平切分**

水平切分是将同一个表中的记录拆分到多个结构相同的表中。当一个表的数据不断增多时，水平切分是必然的选择，它可以将数据分布到集群的不同节点上，从而缓存单个数据库的压力。

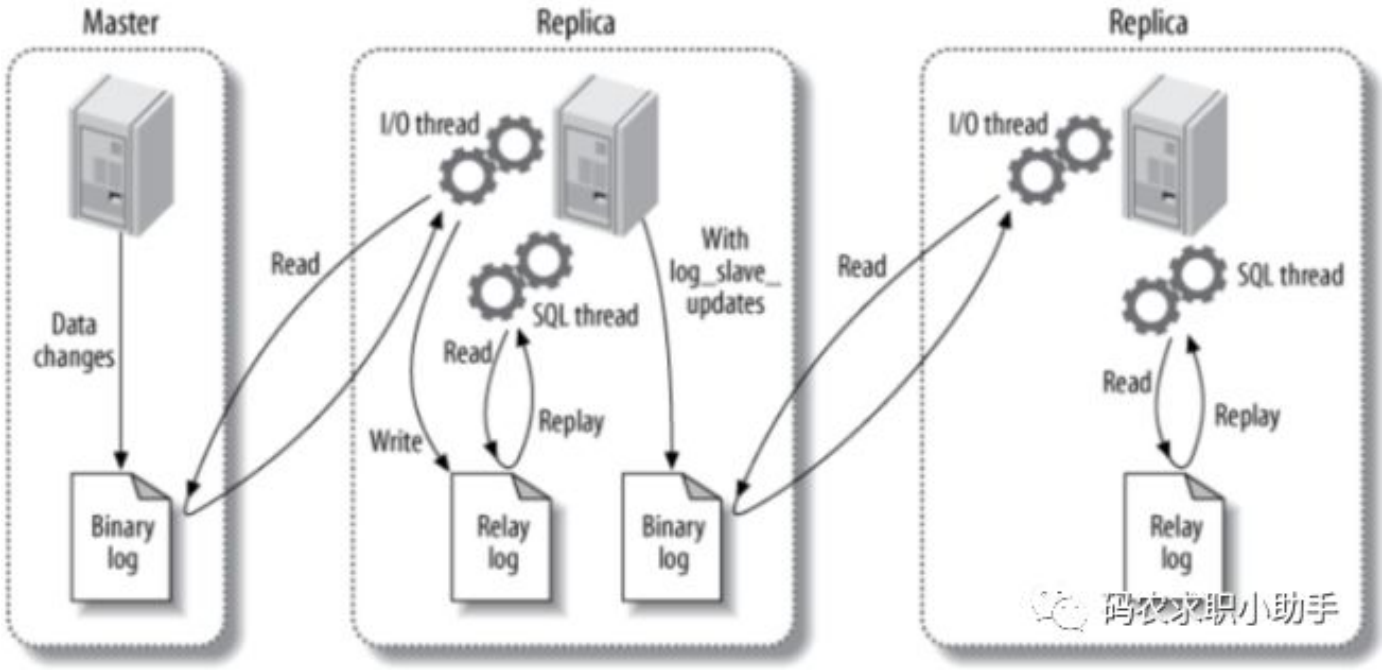
- **垂直切分**

垂直切分是将一张表按列切分成多个表，通常是按照列的关系密集程度进行切分，也可以利用垂直切分将经常被使用的列和不经常被使用的列切分到不同的表中。例如：将原来的电商数据库垂直切分成商品数据库、用户数据库等。

19、主从复制中涉及到哪三个线程?

主要涉及三个线程：binlog 线程、I/O 线程和 SQL 线程。

1. binlog 线程 ： 负责将主服务器上的数据更改写入二进制日志（Binary log）中。
2. I/O 线程 ： 负责从主服务器上读取二进制日志，并写入从服务器的重放日志（Relay log）中。
3. SQL 线程 ： 负责读取重放日志并重放其中的 SQL 语句。



20、主从同步的延迟原因及解决办法?

主从同步的延迟的原因：

假如一个服务器开放 N 个连接给客户端，这样有会有大并发的更新操作, 但是从服务器的里面读取 binlog 的线程仅有一个， 当某个 SQL 在从服务器上执行的时间稍长或者由于某个 SQL 要进行锁表就会导致主服务器的 SQL 大量积压，未被同步到从服务器里。这就导致了主从不一致， 也就是主从延迟。

主从同步延迟的解决办法：

实际上主从同步延迟根本没有什么一招制敌的办法， 因为所有的 SQL 必须都要在从服务器里面执行一遍，但是主服务器如果不断的有更新操作源源不断的写入，那么一旦有延迟产生，那么延迟加重的可能性就会原来越大。当然我们可以做一些缓解的措施。

1. 我们知道因为主服务器要负责更新操作， 它对安全性的要求比从服务器高，所有有些设置可以修改，比如 sync_binlog=1，innodb_flush_log_at_trx_commit = 1 之类的设置，而 slave 则不需要这么高的数据安全，完全可以将 sync_binlog 设置为 0 或者关闭 binlog、innodb_flushlog、innodb_flush_log_at_trx_commit 也可以设置为 0 来提高 SQL 的执行效率。
2. 增加从服务器，这个目的还是分散读的压力， 从而降低服务器负载。

21、谈谈你对数据库读写分离的理解?

读写分离常用代理方式来实现，代理服务器接收应用层传来的读写请求，然后决定转发到哪个服务器。主服务器处理写操作以及实时性要求比较高的读操作，而从服务器处理读操作。

读写分离能提高性能的原因在于：

1. 主从服务器负责各自的读和写，极大程度缓解了锁的争用；
2. 从服务器可以使用 MyISAM，提升查询性能以及节约系统开销；
3. 增加冗余，提高可用性。

22、请你描述下事务的特性?

1. 原子性：事务是最小的执行单位，不允许分割。事务的原子性确保动作要么全部完成，要么完全不起作用；
2. 一致性：执行事务前后，数据库从一个一致性状态转换到另一个一致性状态。
3. 隔离性：并发访问数据库时，一个用户的事物不被其他事务所干扰，各并发事务之间数据库是独立的；
4. 持久性：一个事务被提交之后。它对数据库中数据的改变是持久的，即使数据库 发生故障也不应该对其有任何影响。

23、谈谈你对事务隔离级别的理解?

1. READ_UNCOMMITTED（未提交读）: 最低的隔离级别，允许读取尚未提交的数据变更，可能会导致脏读、幻读或不可重复读；
2. READ_COMMITTED（提交读）: 允许读取并发事务已经提交的数据，可以阻止脏读，但是幻读或不可重复读仍有可能发生；
3. REPEATABLE_READ（可重复读）: 对同一字段的多次读取结果都是一致的，除非数据是被本身事务自己所修改，可以阻止脏读和不可重复读，但幻读仍有可能发生；
4. SERIALIZABLE（串行化）: 最高的隔离级别，完全服从 ACID 的隔离级别。所有的事务依次逐个执行，这样事务之间就完全不可能产生干扰，也就是说，该级别可以防止脏读、不可重复读以及幻读。但是这将严重影响程序的性能。通常情况下也不会用到该级别。

24、解释下什么叫脏读、不可重复读和幻读?

- **脏读:**

表示一个事务能够读取另一个事务中还未提交的数据。比如：某个事务尝试插入记录 A，此时该事务还未提交，然后另一个事务尝试读取到了记录 A。

- **不可重复读：**

是指在一个事务内，多次读同一数据。

- **幻读:**

指同一个事务内多次查询返回的结果集不一样。比如同一个事务 A 第一次查询时候有 n 条记录，但是第二次同等条件下查询却有 n+1 条记录，这就好像产生了幻觉。发生幻读的原因也是另外一个事务新增或者删除或者修改了第一个事务结果集里面的数据，同一个记录的数据内容被修改了，所有数据行的记录就变多或者变少了。

25、MySQL 默认的隔离级别是什么?

MySQL默认采用的 REPEATABLE_READ隔离级别。

Oracle 默认采用的 READ_COMMITTED 隔离级别。

26、谈谈你对MVCC 的了解?

数据库并发场景:

1. 读-读：不存在任何问题，也不需要并发控制；
2. 读-写：有线程安全问题，可能会造成事务隔离性问题，可能遇到脏读，幻读，不可重复读；
3. 写-写：有线程安全问题，可能会存在更新丢失问题。

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。

MVCC 可以为数据库解决以下问题：

1. 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；
2. 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。

27、说一下 MySQL 的行锁和表锁?

MyISAM 只支持表锁，InnoDB 支持表锁和行锁，默认为行锁。

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。锁力度小，发生锁冲突的概率小，并发度最高。

28、InnoDB 存储引擎的锁的算法有哪些?

1. Record lock：单个行记录上的锁；
2. Gap lock：间隙锁，锁定一个范围，不包括记录本身；
3. Next-key lock：record+gap 锁定一个范围，包含记录本身。

29、MySQL 问题排查都有哪些手段?

1. 使用 show processlist 命令查看当前所有连接信息；
2. 使用 Explain 命令查询 SQL 语句执行计划；
3. 开启慢查询日志，查看慢查询的 SQL。

30、MySQL 数据库 CPU 飙升到 500% 的话他怎么处理?

1. 列出所有进程 show processlist，观察所有进程，多秒没有状态变化的(干掉)；
2. 查看超时日志或者错误日志 (一般会是查询以及大批量的插入会导致 CPU与 I/O 上涨，当然不排除网络状态突然断了，导致一个请求服务器只接受到一半。

Redis

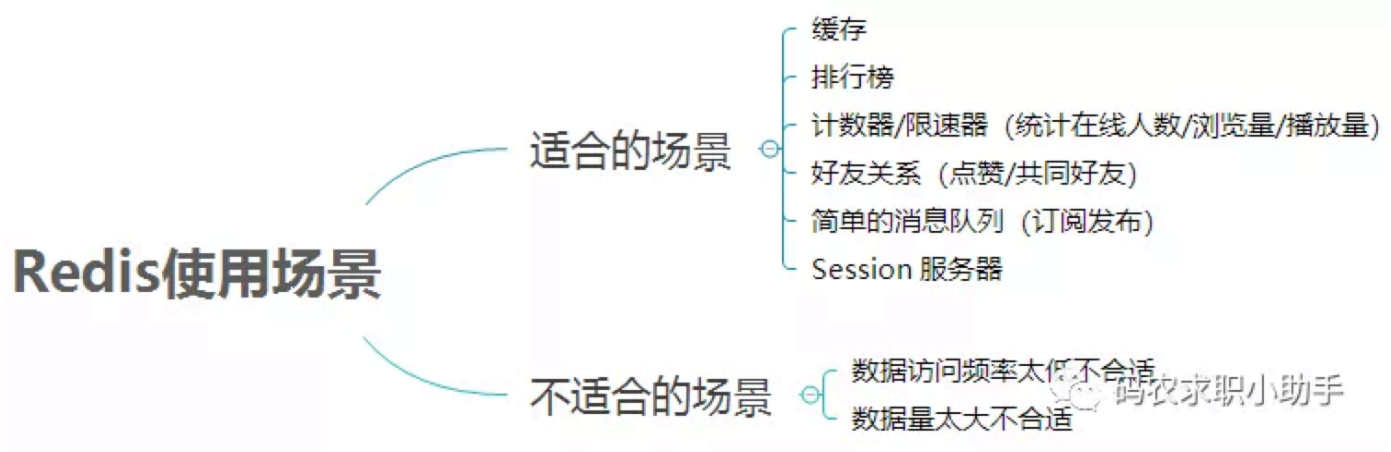
> 你现在手里的这份可能不是最新版，因为帅地看到好的面试题，就会整理进去，强烈建议你去看最新版的，微信搜索关注「**帅地玩编程**」，回复「**1023**」，即可获取最新版的 PDF 哦，扫码直达

关注后，回复「**1023**」，即可获取最新版 PDF。

1、谈下你对 Redis 的了解?

Redis（全称：Remote Dictionary Server 远程字典服务）是一个开源的使用 ANSI C 语言编写、支持网络、可基于内存亦可持久化的日志型、Key-Value 数据库，并提供多种语言的 API。

2、Redis 一般都有哪些使用场景?



Redis 适合的场景

1. 缓存：减轻 MySQL 的查询压力，提升系统性能；
2. 排行榜：利用 Redis 的 SortSet（有序集合）实现；
3. 计算器/限速器：利用 Redis 中原子性的自增操作，我们可以统计类似用户点赞数、用户访问数等。这类操作如果用 MySQL，频繁的读写会带来相当大的压力；限速器比较典型的使用场景是限制某个用户访问某个 API 的频率，常用的有抢购时，防止用户疯狂点击带来不必要的压力；
4. 好友关系：利用集合的一些命令，比如求交集、并集、差集等。可以方便解决一些共同好友、共同爱好之类的功能；
5. 消息队列：除了 Redis 自身的发布/订阅模式，我们也可以利用 List 来实现一个队列机制，比如：到货通知、邮件发送之类的需求，不需要高可靠，但是会带来非常大的 DB 压力，完全可以用 List 来完成异步解耦；
6. Session 共享：Session 是保存在服务器的文件中，如果是集群服务，同一个用户过来可能落在不同机器上，这就会导致用户频繁登陆；采用 Redis 保存 Session 后，无论用户落在那台机器上都能够获取到对应的 Session 信息。

Redis 不适合的场景

数据量太大、数据访问频率非常低的业务都不适合使用 Redis，数据太大会增加成本，访问频率太低，保存在内存中纯属浪费资源。

3、Redis 有哪些常见的功能?

1. 数据缓存功能
2. 分布式锁的功能
3. 支持数据持久化
4. 支持事务
5. 支持消息队列

4、Redis 支持的数据类型有哪些?

- **1. string 字符串**

字符串类型是 Redis 最基础的数据结构，首先键是字符串类型，而且其他几种结构都是在字符串类型基础上构建的。字符串类型实际上可以是字符串：简单的字符串、XML、JSON；数字：整数、浮点数；二进制：图片、音频、视频。

使用场景：缓存、计数器、共享 Session、限速。

- **2. Hash（哈希）**

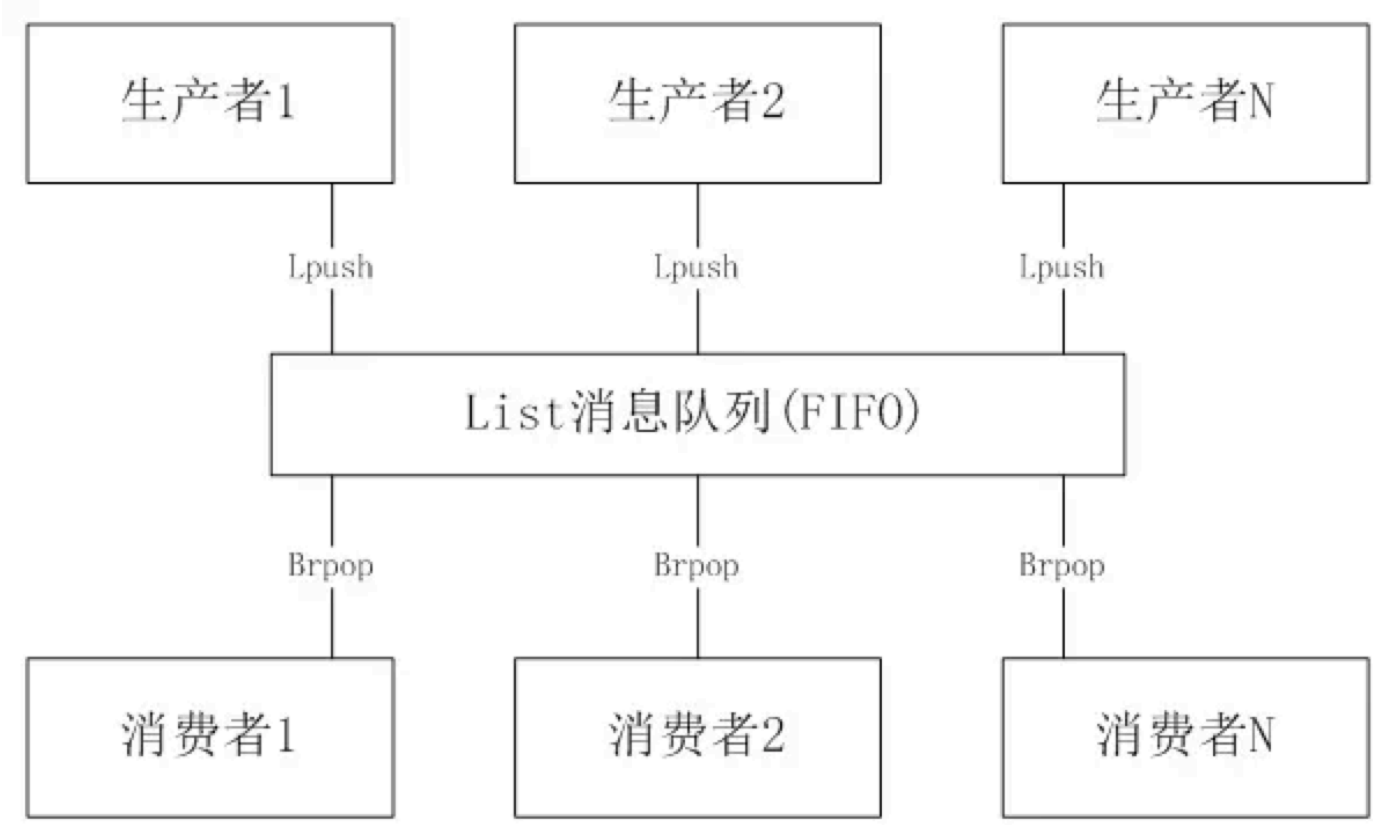
在 Redis中哈希类型是指键本身是一种键值对结构，如 value={{field1,value1},.....{fieldN,valueN}}

使用场景：哈希结构相对于字符串序列化缓存信息更加直观，并且在更新操作上更加便捷。所以常常用于用户信息等管理，但是哈希类型和关系型数据库有所不同，哈希类型是稀疏的，而关系型数据库是完全结构化的，关系型数据库可以做复杂的关系查询，而 Redis 去模拟关系型复杂查询开发困难且维护成本高。

- **3. List（列表）**

列表类型是用来储存多个有序的字符串，列表中的每个字符串成为元素，一个列表最多可以储存 2 ^ 32 - 1 个元素，在 Redis 中，可以队列表两端插入和弹出，还可以获取指定范围的元素列表、获取指定索引下的元素等，列表是一种比较灵活的数据结构，它可以充当栈和队列的角色。

使用场景：Redis 的 lpush + brpop 命令组合即可实现阻塞队列，生产者客户端是用 lpush 从列表左侧插入元素，多个消费者客户端使用 brpop 命令阻塞式的“抢”列表尾部的元素，多个客户端保证了消费的负载均衡和高可用性。



- **4. Set（集合）**

集合类型也是用来保存多个字符串的元素，但和列表不同的是集合中不允许有重复的元素，并且集合中的元素是无序的，不能通过索引下标获取元素，Redis 除了支持集合内的增删改查，同时还支持多个集合取交集、并集、差集。合理的使用好集合类型，能在实际开发中解决很多实际问题。

使用场景：如：一个用户对娱乐、体育比较感兴趣，另一个可能对新闻感兴趣，这些兴趣就是标签，有了这些数据就可以得到同一标签的人，以及用户的共同爱好的标签，这些数据对于用户体验以及曾强用户粘度比较重要。

- **5. zset（sorted set：**有序集合）**

有序集合和集合有着必然的联系，它保留了集合不能有重复成员的特性，但不同得是，有序集合中的元素是可以排序的，但是它和列表的使用索引下标作为排序依据不同的是：它给每个元素设置一个分数，作为排序的依据。

使用场景：排行榜是有序集合经典的使用场景。例如：视频网站需要对用户上传的文件做排行榜，榜单维护可能是多方面：按照时间、按照播放量、按照获得的赞数等。

5、Redis 为什么这么快?

1. 完全基于内存，绝大部分请求是纯粹的内存操作，非常快速；
2. 数据结构简单，对数据操作也简单；
3. 采用单线程，避免了不必要的上下文切换和竞争条件，也不存在多进程或者多线程导致的切换而消耗 CPU，不用去考虑各种锁的问题，不存在加锁释放锁操作，没有因为可能出现死锁而导致的性能消耗；
4. 使用多路 I/O 复用模型，非阻塞 IO。

6、什么是缓存穿透？怎么解决？

缓存穿透是指查询一个一定不存在的数据，由于缓存是不命中时需要从数据库查询，查不到数据则不写入缓存，这将导致这个不存在的数据每次请求都要到数据库去查询，造成缓存穿透。

解决办法：

1、缓存空对象：如果一个查询返回的数据为空（不管是数据不存在，还是系统故障），我们仍然把这个空结果进行缓存，但它的过期时间会很短，最长不超过五分钟。

缓存空对象带来的问题：

1. 空值做了缓存，意味着缓存中存了更多的键，需要更多的内存空间，比较有效的方法是针对这类数据设置一个较短的过期时间，让其自动剔除。
2. 缓存和存储的数据会有一段时间窗口的不一致，可能会对业务有一定影响。例如：过期时间设置为 5分钟，如果此时存储添加了这个数据，那此段时间就会出现缓存和存储数据的不一致，此时可以利用消息系统或者其他方式清除掉缓存层中的空对象。

2、布隆过滤器：将所有可能存在的数据哈希到一个足够大的 bitmap 中，一个一定不存在的数据会被这个 bitmap 拦截掉，从而避免了对底层存储系统的查询压力。

7、什么是缓存雪崩？该如何解决？

如果缓存集中在一段时间内失效，发生大量的缓存穿透，所有的查询都落在数据库上，造成了缓存雪崩。

解决办法：

1. 加锁排队：在缓存失效后，通过加锁或者队列来控制读数据库写缓存的线程数量。比如对某个 key 只允许一个线程查询数据和写缓存，其他线程等待；
2. 数据预热：可以通过缓存 reload 机制，预先去更新缓存，再即将发生大并发访问前手动触发加载缓存不同的 key，设置不同的过期时间，让缓存失效的时间点尽量均匀；
3. 做二级缓存，或者双缓存策略：Cache1 为原始缓存，Cache2 为拷贝缓存，Cache1 失效时，可以访问 Cache2，Cache1 缓存失效时间设置为短期，Cache2 设置为长期。
4. 在缓存的时候给过期时间加上一个随机值，这样就会大幅度的减少缓存在同一时间过期。

8、 怎么保证缓存和数据库数据的一致性？

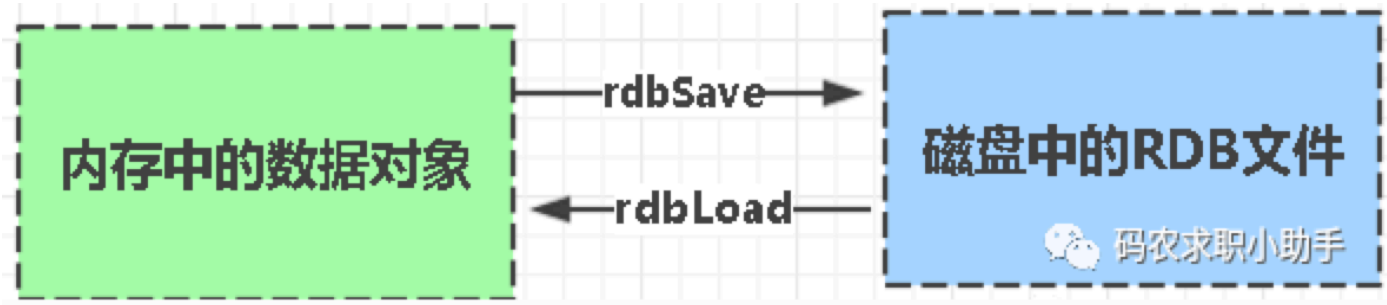
1. 从理论上说，只要我们设置了合理的键的过期时间，我们就能保证缓存和数据库的数据最终是一致的。因为只要缓存数据过期了，就会被删除。随后读的时候，因为缓存里没有，就可以查数据库的数据，然后将数据库查出来的数据写入到缓存中。除了设置过期时间，我们还需要做更多的措施来尽量避免数据库与缓存处于不一致的情况发生。
2. 新增、更改、删除数据库操作时同步更新 Redis，可以使用事物机制来保证数据的一致性。

9、Redis 持久化有几种方式？

持久化就是把内存的数据写到磁盘中去，防止服务宕机了内存数据丢失。Redis 提供了两种持久化方式：RDB（默认） 和 AOF。

RDB

RDB 是 Redis DataBase 的缩写。按照一定的时间周期策略把内存的数据以快照的形式保存到硬盘的二进制文件。即 Snapshot 快照存储，对应产生的数据文件为 dump.rdb，通过配置文件中的 save 参数来定义快照的周期。核心函数：rdbSave（生成 RDB 文件）和 rdbLoad（从文件加载内存）两个函数。



AOF

AOF 是 Append-only file 的缩写。Redis会将每一个收到的写命令都通过 Write 函数追加到文件最后，类似于 MySQL 的 binlog。当 Redis 重启是会通过重新执行文件中保存的写命令来在内存中重建整个数据库的内容。每当执行服务器（定时）任务或者函数时，flushAppendOnlyFile 函数都会被调用， 这个函数执行以下两个工作：

- WRITE：根据条件，将 aof_buf 中的缓存写入到 AOF 文件；
- SAVE：根据条件，调用 fsync 或 fdatasync 函数，将 AOF 文件保存到磁盘中。

RDB 和 AOF 的区别：

1. AOF 文件比 RDB 更新频率高，优先使用 AOF 还原数据；
2. AOF比 RDB 更安全也更大；
3. RDB 性能比 AOF 好；
4. 如果两个都配了优先加载 AOF。

10、Redis 怎么实现分布式锁?

Redis 为单线程模式，采用队列模式将并发访问变成串行访问，且多客户端对 Redis 的连接并不存在竞争关系。Redis 中可以使用 SETNX 命令实现分布式锁。一般使用 setnx(set if not exists) 指令，只允许被一个程序占有，使用完调用 del 释放锁。

11、Redis 淘汰策略有哪些?

1. volatile-lru：从已设置过期时间的数据集（server. db[i]. expires）中挑选最近最少使用的数据淘汰；
2. volatile-ttl：从已设置过期时间的数据集（server. db[i]. expires）中挑选将要过期的数据淘汰。
3. volatile-random：从已设置过期时间的数据集（server. db[i]. expires）中任意选择数据淘汰。
4. allkeys-lru：从数据集（server. db[i]. dict）中挑选最近最少使用的数据淘汰。
5. allkeys-random：从数据集（server. db[i]. dict）中任意选择数据淘汰。
6. no-enviction（驱逐）：禁止驱逐数据。

12、Redis 常见性能问题和解决方案?

1. Master 最好不要做任何持久化工作，如 RDB 内存快照和 AOF 日志文件。如果数据比较重要，某个 Slave 开启 AOF 备份数据，策略设置为每秒同步一次；
2. 为了主从复制的速度和连接的稳定性， Master 和 Slave 最好在同一个局域网内；
3. 主从复制不要用图状结构，用单向链表结构更为稳定，即：Master <- Slave1 <- Slave2 <- Slave3...

少走弯路，各类书籍推荐

少走弯路，帅地给大家推荐一些书籍，大部分都是帅地看过，质量有保证，并且还提供了**高清带目录**的电子版，节省大家寻找书籍的时间，看看目录

微信搜索公众号「**帅地玩编程****」的后台回复「**0000**」，就可以获取下载链接了，也可以直接扫码

> 下面是详细介绍

1、数据结构与算法

入门：《啊哈算法》，《数据结构与算法分析:xx语言描述版》

提升：《编程之美》，《剑指offer》，《程序员代码面试指南：IT 名企算法与数据结构题目最优解》，《算法4》

2、计算机网络

零基础先看《网络是怎样连接的》，之后看《计算机网络自动向上》，这两本足够了。

3、操作系统

感觉看《现代操作系统》就够了，如果零基础，想学操作系统和计算机组成原理，那么可以看《程序是怎样跑起来的》，之后看《现代操作系统》，再之后看《深入理解计算机操作系统》这本天书，这本天书包含了操作系统+计组的知识，挑着看就行。

4、MySQL

入门：《MySQL必知必会》，进阶：《MySQL技术内幕InnoDB存储引擎》，这两本差不多，但个人感觉还不足以应付，某些知识点结合部分文章应该就可以了。如果是为了面试，个人感觉没必要看《高性能MySQL》。

5、Go语言

入门：《学习Go 语言》，学习Go Web：《Go Web 编程》

6、C 语言

入门：《C Primer Plus》(可能零基础有点吃力，觉得吃力的就看《C程序员涉及语言》吧)

进阶：《C 和指针》，我觉得 C 语言，最核心的就是理解指针

7、C++

入门：《C++ Primer》

进阶：《深入探索C ++对象模型》、《more effective C++》、《C++编程思想》

8、Java

入门：《Java核心技术卷1》

进阶：《Java 编程思想》

多线程：《Java 并发编程的艺术》、《Java并发编程实践》

虚拟机：《深入理解Java虚拟机》

10、Linux

入门：《鸟哥的Linux私房菜》

11、Python

爬虫：《用 Python 写网络爬虫》

数据分析：《Python数据处理》、《Python数据分析实战》

Python：《编程小白的第一本Python入门书》、《Python网络编程基础》、《Python高级编程》

12、前端

HTML+css+JS：《HTML5与CSS3基础教程》、《JavaScript高级程序设计》

其他：《Node.js开发指南》

13、设计模式

《图解设计模式》

14、人工智能

《贝叶斯思维统计建模的Python学习法》、《TensorFlow实践与智能系统》

15、汇编

不用学太深，入门即可，看《汇编语言》（王爽这边就差不多了）

16、Git

入门：《快速入门Git》，貌似Git不用学的很深入，需要时在查询就可以了

以上所有书籍，可以关注微信公众号「**帅地玩编程**」，回复「**0000**」，即可无套路下载，扫码直达

关注后，后台回复「**0000**」即可领取。

200本计算机书籍免费下载（高清带目录完整PDF版）

计算机类的书籍那么贵，作为一个几个小时看完一本书且机不离身的程序员，天天买纸质书是不可能的了，所以对电子书的需求量还是挺多的。为了方便广大的小伙伴也能方便找到对应的电子书，我花费洪荒之力从各个搜索网站收集了几百本常用的电子书。

有人反馈百度云下载太慢，所以我部分书籍提供了蓝奏云下载，大家如果觉得有帮助，可以偷偷收藏起来哈，不过蓝奏云对文件的大小有限制，我开了 VIP，但还是只能上传 100 多MB 的文件。

书籍会在 GitHub 上实时更新，地址：https://github.com/iamshuaidi/CS-Book

大神刷题笔记

- Leetcode 1300道算法题解 蓝奏云直接下载(推荐)
- labuladong 的算法小抄 蓝奏云直接下载(推荐)
- 阿里巴巴Java面试问题大全 蓝奏云直接下载(推荐) 百度云下载链接 密码:8xzm
- 程序员面试宝典 蓝奏云直接下载(推荐) 百度云下载链接 密码:ko62
- 大厂面试真题 蓝奏云直接下载(推荐) 百度云下载链接 密码:gu92
- Java面试突击 蓝奏云直接下载(推荐) 百度云下载链接 密码:h44t

数据结构与算法相关书籍

- 挑战程序设计竞赛 百度云下载链接 密码:mxn7
- Java数据结构和算法 百度云下载链接 密码:lpym
- 算法图解 百度云下载链接 密码:7osf
- 算法导论 百度云下载链接 密码:p2tp
- 算法第四版 百度云下载链接 密码:rixw
- 数据结构与算法分析C语言描述版 百度云下载链接 密码:mn10
- 数据结构与算法分析Java语言描述版 百度云下载链接 密码:917n
- 数据结构与算法 Python语言描述_裘宗燕 百度云下载链接 密码:96gw
- 剑指offer 百度云下载链接 密码:a4rt
- 计算机程序设计艺术1-3卷 百度云下载链接 密码:i3nh
- 大话数据结构 百度云下载链接 密码:np2o
- 程序员代码面试指南：IT 名企算法与数据结构题目最优解 百度云下载链接 密码:20oh
- 编程珠玑 百度云下载链接 密码:4oow
- 编程之美 百度云下载链接 密码:4zme
- 啊哈算法 百度云下载链接 密码:h4id
- 程序员的算法趣题 百度云下载链接 密码:kk84

计算机基础

操作系统

- 30天填自制操作系统 蓝奏云直接下载(推荐) 百度云下载链接 密码:pxxr
- 操作系统之哲学原理 蓝奏云直接下载(推荐) 百度云下载链接 密码:hua7
- 程序是怎样跑起来的 蓝奏云直接下载(推荐) 百度云下载链接 密码:pbvh
- 深入理解计算机操作系统 百度云下载链接 密码:2toh
- 现代操作系统 蓝奏云直接下载(推荐) 百度云下载链接 密码:huk2

汇编语言

- 汇编语言（注：这边是王爽写的，我觉得写的很好，适合入门） 蓝奏云直接下载(推荐) 百度云下载链接 密码:qea3

计算机网络

- 计算机网络：自顶向下 蓝奏云直接下载(推荐) 百度云下载链接 密码:l77d
- 图解HTTP 蓝奏云直接下载(推荐) 百度云下载链接 密码:b42z
- 图解TC/IP 蓝奏云直接下载(推荐) 百度云下载链接 密码:5k3x
- 网络是怎样连接的 蓝奏云直接下载(推荐) 百度云下载链接 密码:g983
- HTTP权威指南 蓝奏云直接下载(推荐) 百度云下载链接 密码:tqj8
- UNIX网络编程 蓝奏云直接下载(推荐) 百度云下载链接 密码:4buy

计算机组成原理

- 隐匿在计算机软硬件背后的语言 蓝奏云直接下载(推荐) 百度云下载链接 密码:6jiq
- 大话计算机 蓝奏云直接下载(推荐) 百度云下载链接 密码:6j1o
- 计算机是怎样跑起来的 蓝奏云直接下载(推荐) 百度云下载链接 密码:er12

Python

1、Python基础

- 编程小白的第一本Python入门书 百度云下载链接 密码:s76b
- Python编程初学者指南 百度云下载链接 密码:g696
- Python高级编程 百度云下载链接 密码:qnuf
- Python编程入门经典 百度云下载链接 密码:j7gp
- Python灰帽子 百度云下载链接 密码:nfkt
- Python开发技术详解 百度云下载链接 密码:z0av
- Python开发实战 百度云下载链接 密码:ee5x
- Python网络编程基础 百度云下载链接 密码:m15q
- Python学习手册 百度云下载链接 密码:0yqf
- 精通Scrapy网络爬虫

2、数据分析与爬虫

- 数据科学入门 百度云下载链接 密码:1hz8
- 用Python写网络爬虫 百度云下载链接 密码:nlpa
- Python数据处理 百度云下载链接 密码:8eom
- Python数据分析实战 百度云下载链接 密码:idn1
- Python数据科学手册 百度云下载链接 密码:16u0
- Python数据可视化编程实战 百度云下载链接 密码:i7hp
- 精通Scrapy网络爬虫 百度云下载链接 密码:jb6u

Linux

- 精通正则表达式 百度云下载链接 密码:vp94
- 鸟哥的Linux私房菜基础篇和服务器篇 百度云下载链接 密码:ubg2
- 深入Linux内核架构 百度云下载链接 密码:fnh2
- Linux宝典 百度云下载链接 密码:nxhe
- Linux常用命令大全 百度云下载链接 密码:e0n2
- Linux防火墙 百度云下载链接 密码:sc4u
- Linux高级程序设计 百度云下载链接 密码:d4wq
- Linux环境编程 百度云下载链接 密码:xudv
- Linux命令详解词典 百度云下载链接 密码:yxuz
- 汇编语言基于linux环境第3版 百度云下载链接 密码:yq75

C语言

- 经典C程序100例 百度云下载链接 密码:tls2
- C Primer Plus 百度云下载链接 密码:5s85
- C程序设计语言（包括课后答案等） 百度云下载链接 密码:cfj4
- C和指针 百度云下载链接 密码:d8a0
- C语言编程精粹 百度云下载链接 密码:6mct
- C语言参考手册 百度云下载链接 密码:4hnx
- C语言函数大全 百度云下载链接 密码:xywt
- C语言解析教程 百度云下载链接 密码:6luf

- C语言深度剖析 百度云下载链接 密码:yu63
- C专家编程 百度云下载链接 密码:xlfm

C++

- C++ Primer 百度云下载链接 密码:js1a
- C++编程思想 百度云下载链接 密码:vi02
- C++对象模型 百度云下载链接 密码:v90k
- 深入探索C++对象模型 百度云下载链接 密码:3xuv
- C++ Templates 百度云下载链接 密码:4rvw
- C++编程规范-101条规则准则与最佳实践 百度云下载链接 密码:t43e
- C++沉思录中文第2版 百度云下载链接 密码:6emr
- C++大学教程 百度云下载链接 密码:n6ph
- C++设计新思维-泛型编程与设计之应用 百度云下载链接 密码:0el5
- Effective STL 中文版 百度云下载链接 密码:u7s1
- More Effective C++中文版 百度云下载链接 密码:xbxv
- STL源码剖析 百度云下载链接 密码:mxsh

前端

- 疯狂aJax讲义 百度云下载链接 密码:pce0
- Bootstrap实战 百度云下载链接 密码:rzhm
- HTML5揭秘 百度云下载链接 密码:vaam
- HTML5与CSS3基础教程 百度云下载链接 密码:2wxm
- HTML与CSS入门经典 百度云下载链接 密码:tsgm
- JavaScript DOM编程艺术 百度云下载链接 密码:gsbt
- JavaScript高级程序设计 百度云下载链接 密码:wbw0
- JavaScript高效图形编程 百度云下载链接 密码:tab1
- jQuery高级编程 百度云下载链接 密码:qwtr
- jQuery技术内幕 百度云下载链接 密码:pglf
- jQuery权威指南 百度云下载链接 密码:4vrw
- Node.js开发指南 百度云下载链接 密码:voze

人工智能

- 贝叶斯思维统计建模的Python学习法 百度云下载链接 密码:ztbe
- 机器学习实战 百度云下载链接 密码:cfqc
- Python机器学习及实践 百度云下载链接 密码:qq3q
- Tensorflow实战Google深度学习框架 百度云下载链接 密码:12kj
- TensorFlow实践与智能系统 百度云下载链接 密码:e668
- 深度学习_中文版 百度云下载链接 密码:01xp

设计模式

- 图解设计模式 百度云下载链接 密码:g50a
- 研磨设计模式 百度云下载链接 密码:h5fb
- Head First设计模式 百度云下载链接 密码:pxpq

Java

Java 基础

- 阿里巴巴Java开发手册 百度云下载链接 密码:g6lv
- 码出高效 百度云下载链接 密码:mbt9
- Head First Java 百度云下载链接 密码:d5ll
- Java8实战 百度云下载链接 密码:lvmb
- Java编程思想 百度云下载链接 密码:0add
- Java并发编程的艺术 百度云下载链接 密码:92jj
- Java并发编程实践 百度云下载链接 密码:i6w9
- Java从小白到大牛 百度云下载链接 密码:9auc
- Java核心技术1-2卷 百度云下载链接 密码:tr3s
- 深入理解Java虚拟机 百度云下载链接 密码:b6op

Java进阶

- 代码大全 百度云下载链接 密码:juhq
- 代码整洁之道 百度云下载链接 密码:hzn2
- 敏捷软件开发 百度云下载链接 密码:mmi4
- Effective Java(中文) 百度云下载链接 密码:4dcx
- Effective Java（英文） 百度云下载链接 密码:bhu4
- Java性能优化权威指南 百度云下载链接 密码:054x

JavaWeb

- 轻量级JavaEE企业应用实战 百度云下载链接 密码:8j3c
- 深入分析JavaWeb技术内幕 百度云下载链接 密码:31uu
- 深入剖析Tomcat 百度云下载链接 密码:y1yt
- Head First Servlet and JSP 百度云下载链接 密码:v8b8
- Maven实战 百度云下载链接 密码:s0q9
- Spring实战 百度云下载链接 密码:40fb
- Camel in Action 百度云下载链接 密码:plyn
- Spring 5 Recipes, 4th Edition 百度云盘下载链接 密码:slt0

Java工具

- Java se 11中文 api 百度云下载链接 密码：nkjm

数据库

- 高性能MySQL 百度云下载链接 密码:gh5t
- 深入浅出MySQL 百度云下载链接 密码:ju0h
- MongoDB权威指南 百度云下载链接 密码:llvx
- MySQL必知必会 百度云下载链接 密码:f1v9
- MySQL技术内幕InnoDB存储引擎 百度云下载链接 密码:6g04）
- SQL查询的艺术 百度云下载链接 密码:ndcv
- SQLite 权威指南 百度云下载链接 密码:ex3h

Go

- 学习Go语言 百度云下载链接 密码:5tri
- Go语言实战 百度云下载链接 密码:q0x3
- Go web编程 百度云下载链接 密码:pyw3
- C 程序设计语言第2版 百度云下载链接 密码:3yza

中间件

- redis实战 百度云下载链接 密码:ro48

未分类书籍

- 黑客与画家 百度云下载链接 密码:uux2
- 浪潮之巅 百度云下载链接 密码:xr66
- 奔跑吧，程序员：从零开始打造产品、技术和团队 百度云下载链接 密码:jxvj

Git

- 快速入门Git 百度云下载链接 密码:ofdd
- 专业git中文 百度云下载链接 密码:b3kx
- Git参考手册 百度云下载链接 密码:axou
- 《Pro Git》中文版 百度云下载链接 密码:l05a

免责声明

书籍全部来源于网络其他人的整理，我这里只是收集整理了他们的链接，如有侵权，马上联系我，我立马删除对应链接。我的邮箱：1326194964@qq.com