面试题总结：

1、TCP/IP三次握手，四次挥手?

TCP提供了一种可靠、面向连接、字节流、传输层的服务，采用三次握手建立一个连接。采用4次挥手来关闭一个连接。一个TCP连接由一个4元组构成，分别是两个IP地址和两个端口号。一个TCP连接通常分为三个阶段：启动、数据传输、退出（关闭）。

ACK —— 确认，使得确认号有效。

RST —— 重置连接（经常看到的reset by peer）就是此字段搞的鬼。

SYN —— 用于初如化一个连接的序列号。

FIN —— 该报文段的发送方已经结束向对方发送数据。

原因："3次握手"的作用就是`双方都能明确自己和对方的收、发能力是正常的`。

三次握手：

通俗版：

`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。 从客户端的视角来看，我接到了服务端发送过来的响应数据包，说明服务端接收到了我在第一次握手时发送的网络包，并且成功发送了响应数据包，这就说明，服务端的接收、发送能力正常。而另一方面，我收到了服务端的响应数据包，说明我第一次发送的网络包成功到达服务端，这样，我自己的发送和接收能力也是正常的。

`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，服务端的发送、接收能力是正常的。 第一、二次握手后，服务端并不知道客户端的接收能力以及自己的发送能力是否正常。而在第三次握手时，服务端收到了客户端对第二次握手作的回应。从服务端的角度，我在第二次握手时的响应数据发送出去了，客户端接收到了。所以，我的发送能力是正常的。而客户端的接收能力也是正常的。

专业版：

- 第一次握手(SYN=1, seq=x):

  客户端发送一个 TCP 的 SYN 标志位置1的包，指明客户端打算连接的服务器的端口，以及初始序号X,保存在包头的序列号(Sequence Number)字段里。

  发送完毕后，客户端进入 `SYN_SEND` 状态。

- 第二次握手(SYN=1, ACK=1, seq=y, ACKnum=x+1):

  服务器发回确认包(ACK)应答。即 SYN 标志位和 ACK 标志位均为1。服务器端选择自己 ISN 序列号，放到 Seq 域里，同时将确认序号(Acknowledgement Number)设置为客户的 ISN 加1，即 X+1。 发送完毕后，服务器端进入 `SYN_RCVD` 状态。

- 第三次握手(ACK=1，ACKnum=y+1)

  客户端再次发送确认包(ACK)，SYN 标志位为0，ACK 标志位为1，并且把服务器发来 ACK 的序号字段+1，放在确定字段中发送给对方，并且在数据段放写ISN的+1

  发送完毕后，客户端进入 `ESTABLISHED` 状态，当服务器端接收到这个包时，也进入 `ESTABLISHED` 状态，TCP 握手结束。

视角	客收	客发	服收	服发
客视角	二	一 + 二	一 + 二	二
服视角	二 + 三	一	一	二 + 三

四次挥手：

TCP连接是双向传输的对等的模式，就是说双方都可以同时向对方发送或接收数据。

当有一方要关闭连接时，会发送指令告知对方，我要关闭连接了。这时对方会回一个ACK，此时一个方向的连接关闭。但是另一个方向仍然可以继续传输数据，等到发送完了所有的数据后，会发送一个FIN段来关闭此方向上的连接。接收方发送ACK确认关闭连接。

注意：接收到FIN报文的一方只能回复一个ACK, 它是无法马上返回对方一个FIN报文段的，因为结束数据传输的“指令”是上层应用层给出的，我只是一个“搬运工”，我无法了解“上层的意志”。

专业版：

- 第一次挥手(FIN=1，seq=x)

  假设客户端想要关闭连接，客户端发送一个 FIN 标志位置为1的包，表示自己已经没有数据可以发送了，但是仍然可以接受数据。发送完毕后，客户端进入 `FIN_WAIT_1` 状态。

- 第二次挥手(ACK=1，ACKnum=x+1)

  服务器端确认客户端的 FIN 包，发送一个确认包，表明自己接受到了客户端关闭连接的请求，但还没有准备好关闭连接。发送完毕后，服务器端进入 `CLOSE_WAIT` 状态，客户端接收到这个确认包之后，进入 `FIN_WAIT_2` 状态，等待服务器端关闭连接。

- 第三次挥手(FIN=1，seq=y)

  服务器端准备好关闭连接时，向客户端发送结束连接请求，FIN 置为1。发送完毕后，服务器端进入 `LAST_ACK` 状态，等待来自客户端的最后一个ACK。

- 第四次挥手(ACK=1，ACKnum=y+1)

  客户端接收到来自服务器端的关闭请求，发送一个确认包，并进入 `TIME_WAIT` 状态，等待可能出现的要求重传的 ACK 包。服务器端接收到这个确认包之后，关闭连接，进入 `CLOSED` 状态。客户端等待了某个固定时间（两个最大段生命周期，2MSL，2 Maximum Segment Lifetime）之后，没有收到服务器端的 ACK ，认为服务器端已经正常关闭连接，于是自己也关闭连接，进入 `CLOSED` 状态。

Plus，利用数据包的选项来传输特殊的信息，交换初始序列号ISN。

3次握手是指发送了3个报文段，4次挥手是指发送了4个报文段。注意：SYN和FIN段都是会利用重传进行可靠传输的。

2、二叉树的前中后序

```
//递归
void preorder(TreeNode *p, vector<int>& result) {
    if (p == NULL) {return;}
    result.push_back(p->val);//1
    preorder(p->left, result);//2
    preorder(p->right, result);//3
    //后序2，3，1 //中序2，1，3
}
vector<int> preorderTraversal(TreeNode* root) {
```

```
    vector<int> result;
    if (root == nullptr) {return result;}
    preorder(root, result);
    return result;
}

//递推
//前序
vector<int> preorderTraversal(TreeNode* root) {
    TreeNode *p = root;
    vector<int> result;
    if (!p) {return result;}

    stack<TreeNode *> q;
    while (p || !q.empty()) {
        if (p) {
            result.push_back(p->val);
            q.push(p);
            p = p->left;
        }
        else {
            p = q.top();
            q.pop();
            p = p->right;
        }
    }
    return result;
}
//后序
 vector<int> postorderTraversal(TreeNode* root) {
    TreeNode *p = root;
    vector<int> result;
    if (!p) {return result;}

    TreeNode *top, *last = NULL;
    stack<TreeNode *> q;
    while (p || !q.empty()) {
        if (p) {
            q.push(p);
            p = p->left;
        } else {
            top = q.top();
            if (top->right == NULL || top->right == last) {
                q.pop();
                result.push_back(top->val);
                last = top;
            } else {
                p = top->right;
            }
        }
    }
    return result;
}
//中序
vector<int> inorderTraversal(TreeNode* root) {
    TreeNode *p = root;
    vector<int> result;
    if (!p) {return result;}
```

```
    stack<TreeNode *> q;
    while (p || !q.empty()) {
        if (p) {
            q.push(p);
            p = p->left;
        }
        else {
            p = q.top();
            result.push_back(p->val);
            q.pop();
            p = p->right;
        }
    }
    return result;
}
```

3、哈希碰撞解决

当两个不同的输入值对应一个输出值时，就会产生“碰撞”，这个时候便需要解决冲突。

常见的冲突解决方法有开放定址法，**链地址法**，建立公共溢出区等。

4、快速排序

```
#include <iostream>
using namespace std;
int partition(vector<int>& arr, int left, int right) {
    ///也可以弄个随机数，与最右交换过来，最后换回去
    int key = arr[right]; ///最右为轴点
    int k = left;
    for(int i = left; i < right; ++i) {
        if(arr[i] < key) {
            swap(arr[i], arr[k++]); //先交换arr[k],k++
        }
    }
    swap(arr[right], arr[k]);
    return k;
}

void quicksort(vector<int>& arr, int left, int right) {
    //if(left < right) {
    //    int i = partition(arr, left, right);
    //    quicksort(arr, left, i-1);
    //    quicksort(arr, i+1, right);
    //}
    //return;
    if(left == right) return;
    int index = partition(arr, left, right);
    if(index > left)  quicksort(arr,left,index-1);
    if(index < right) quicksort(arr,index+1,right);
}
```

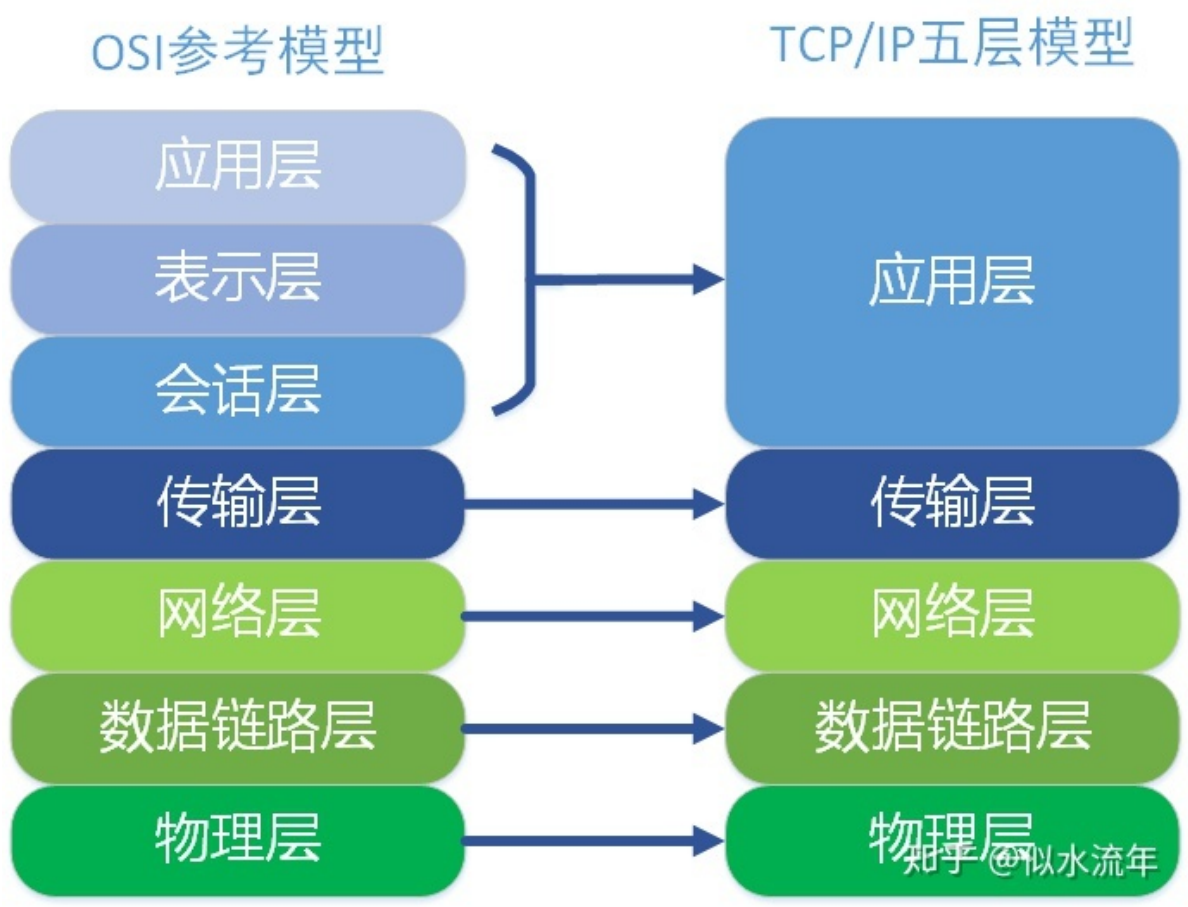
5、归并排序

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

void merge(vector<int>& A, vector<int> L, vector<int> R) {
    int l = L.size();
    int r = R.size();
    int i = 0;
    int j = 0;
    int k = 0;
    while(i < l && j < r) {
        if(L[i] < R[j]) {
            A[k++] = L[i++];
        } else {
            A[k++] = R[j++];
        }
    }
    while(i < l) A[k++] = L[i++];
    while(j < r) A[k++] = R[j++];
}

void mergesort(vector<int>& arr) {
    int n = arr.size();
    if(n < 2) return;
    int mid = n/2;
    int i;
    vector<int> L(mid);
    vector<int> R(n - mid);
    for(i = 0; i < mid; ++i) {
        L[i] = arr[i];
    }
    for(;i < n; ++i) {
        R[i-mid] = arr[i];
    }
    mergesort(L);
    mergesort(R);
    merge(arr, L, R);
}
```

6、OSI七层模型与TCP/IP五层模型



1） 物理层：利用传输介质为数据链路层提供物理连接，实现比特流的透明传输

2）数据链路层：通过各种控制协议，将有差错的物理信道变为无差错的、能可靠传输数据帧的数据链路

3）网络层提供路由和寻址的功能，使两终端系统能够互连且决定最佳路径，并具有一定的拥塞控制和流量控制的能力。

4）传输层：向用户提供可靠的端到端的差错和流量控制，保证报文的正确传输。传输层的作用是向高层屏蔽下层数据通信的细节，即向用户透明地传送报文

5）会话层：任务就是向两个实体的表示层提供建立和使用连接的方法。

6）表示层：它对来自应用层的命令和数据进行解释，对各种语法赋予相应的含义，并按照一定的格式传送给会话层

7）应用层：应用层为用户提供的服务和协议有：文件服务、目录服务、文件传输服务（FTP）、远程登录服务（Telnet）、电子邮件服务（E-mail）、打印服务、安全服务、网络管理服务、数据库服务等

传输层和网络层的区别：

- 网络层为不同主机提供通信服务，而传输层为不同主机的不同应用提供通信服务
- 网络层只对报文头部进行差错检测，而传输层对整个报文进行差错检测

每一层的协议如下：

物理层：RJ45、CLOCK、IEEE802.3 （中继器，集线器）

数据链路：PPP、FR、HDLC、VLAN、MAC （网桥，交换机）

网络层：IP、ICMP、ARP、RARP、OSPF、IPX、RIP、IGRP、 （路由器）

传输层：TCP、UDP、SPX

会话层：NFS、SQL、NETBIOS、RPC

表示层：JPEG、MPEG、ASII

应用层：FTP、DNS、Telnet、SMTP、HTTP、WWW、NFS

7、TCP 与 UDP 的区别

1. TCP 面向连接，UDP 是无连接的；
2. TCP 提供可靠的服务，也就是说，通过 TCP 连接传送的数据，无差错，不丢失，不重复，且按序到达；UDP 尽最大努力交付，即不保证可靠交付
3. TCP 的逻辑通信信道是全双工的可靠信道；UDP 则是不可靠信道
4. 每一条 TCP 连接只能是点到点的；UDP 支持一对一，一对多，多对一和多对多的交互通信
5. TCP 面向字节流（可能出现黏包问题），实际上是 TCP 把数据看成一连串无结构的字节流；UDP 是面向报文的（不会出现黏包问题）
6. 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 IP 电话，实时视频会议等）
7. TCP 首部开销20字节；UDP 的首部开销小，只有 8 个字节

8、http和https区别

超文本传输协议HTTP协议被用于在Web浏览器和网站服务器之间传递信息，HTTP协议以明文方式发送内容，不提供任何方式的数据加密，如果攻击者截取了Web浏览器和网站服务器之间的传输报文，就可以直接读懂其中的信息，因此，HTTP协议不适合传输一些敏感信息，比如：信用卡号、密码等支付信息。

为了解决HTTP协议的这一缺陷，需要使用另一种协议：安全套接字层超文本传输协议HTTPS，为了数据传输的安全，HTTPS在HTTP的基础上加入了SSL协议，SSL依靠证书来验证服务器的身份，并为浏览器和服务器之间的通信加密。

HTTP和HTTPS的基本概念

HTTP：是互联网上应用最为广泛的一种网络协议，是一个客户端和服务器端请求和应答的标准（TCP），用于从WWW服务器传输超文本到本地浏览器的传输协议，它可以使浏览器更加高效，使网络传输减少。

HTTPS：是以安全为目标的HTTP通道，简单讲是HTTP的安全版，即HTTP下加入SSL层，HTTPS的安全基础是SSL，因此加密的详细内容就需要SSL。

HTTPS和HTTP的区别主要如下：

1、https协议需要到ca申请证书，一般免费证书较少，因而需要一定费用。

2、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。

3、http和https使用的是完全不同的连接方式，用的端口也不一样，前者是80，后者是443。

4、http的连接很简单，是无状态的；HTTPS协议是由SSL+HTTP协议构建的可进行加密传输、身份认证的网络协议，比http协议安全。

9、动态规划和贪心算法的区别

贪心算法
基本思想：贪心算法并不从整体最优上加以考虑，它所做的选择只是在某种意义上的局部最优解。
基本要素：最优子结构性质和贪心选择性质。

动态规划

基本思想：将待求解的问题分解成若干个子问题，先求解子问题，然后从这些子问题的解得到原问题的解。

基本要素：最优子结构性质和重叠子问题性质

贪心算法与动态规划的区别

共同点：两者都具有最优子结构性质

不同点：

1) 动态规划算法中，每步所做的选择往往依赖于相关子问题的解，因而只有在解出相关子问题时才能做出选择。而贪心算法，仅在当前状态下做出最好选择，即局部最优选择，然后再去解做出这个选择后产生的相应的子问题。

2) 动态规划算法通常以自底向上的方式解各子问题，而贪心算法则通常自顶向下的方式进行。

10、多进程和多线程的区别

多进程和多线程的区别

维度	多进程	多线程	总结
数据共享、同步	数据是分开的，共享复杂，需要用IPC；同步简单	多线程共享进程数据，共享简单；同步复杂	各有优势
内存、CPU	占用内存多，切换复杂，CPU利用率低	占用内存少，切换简单，CPU利用率高	线程占优
创建销毁、切换	创建销毁、切换复杂，速度慢	创建销毁、切换简单，速度快	线程占优
编程调试	编程简单，调试简单	编程复杂，调试复杂	进程占优
可靠性	进程间不会相互影响	一个线程挂掉将导致整个进程挂掉	进程占优
分布式	适应于多核、多机分布；如果一台机器不够，扩展到多台机器比较简单	适应于多核分布	线程占优

协程

协程和线程的比较

比较项	线程	协程
占用资源	初始单位为1MB,固定不可变	初始一般为 2KB，可随需要而增大
调度所属	由 OS 的内核完成	由用户完成
切换开销	涉及模式切换(从用户态切换到内核态)、16个寄存器、PC、SP...等寄存器的刷新等	只有三个寄存器的值修改 - PC / SP / DX.
性能问题	资源占用太高，频繁创建销毁会带来严重的性能问题	资源占用小,不会带来严重的性能问题
数据同步	需要用锁等机制确保数据的一直性和可见性	不需要多线程的锁机制，因为只有一个线程，也不存在同时写变量冲突，在协程中控制共享资源不加锁，只需要判断状态就好了，所以执行效率比多线程高很多。

进程与线程

对于有线程系统：

- 进程是资源分配的独立单位
- 线程是资源调度的独立单位

11、设计模式

设计模式（Design pattern）代表了最佳的实践，通常被有经验的面向对象的软件开发人员所采用。设计模式是软件开发人员在软件开发过程中面临的一般问题的解决方案。这些解决方案是众多软件开发人员经过相当长的一段时间的试验和错误总结出来的。

设计模式是一套被反复使用的、多数人知晓的、经过分类编目的、代码设计经验的总结。使用设计模式是为了重用代码、让代码更容易被他人理解、保证代码可靠性。 毫无疑问，设计模式于己于他人于系统都是多赢的，设计模式使代码编制真正工程化，设计模式是软件工程的基石，如同大厦的一块块砖石一样。项目中合理地运用设计模式可以完美地解决很多问题，每种模式在现实中都有相应的原理来与之对应，每种模式都描述了一个在我们周围不断重复发生的问题，以及该问题的核心解决方案，这也是设计模式能被广泛应用的原因。

12、栈和堆

栈：由操作系统自动分配释放，存放函数的参数值、局部变量等的值，用于维护函数调用的上下文
堆：一般由程序员分配释放，若程序员不释放，程序结束时可能由操作系统回收，用来容纳应用程序动态分配的内存区域

13、四个智能指针

C++ 11

1. shared_ptr
2. unique_ptr
3. weak_ptr
4. auto_ptr（被 C++11 弃用）

- Class shared_ptr 实现共享式拥有（shared ownership）概念。多个智能指针指向相同对象，该对象和其相关资源会在 “最后一个 reference 被销毁” 时被释放。为了在结构较复杂的情景中执行上述工作，标准库提供 weak_ptr、bad_weak_ptr 和 enable_shared_from_this 等辅助类。
- Class unique_ptr 实现独占式拥有（exclusive ownership）或严格拥有（strict ownership）概念，保证同一时间内只有一个智能指针可以指向该对象。你可以移交拥有权。它对于避免内存泄漏（resource leak）——如 new 后忘记 delete ——特别有用。

shared_ptr

多个智能指针可以共享同一个对象，对象的最末一个拥有着有责任销毁对象，并清理与该对象相关的所有资源。

- 支持定制型删除器（custom deleter），可防范 Cross-DLL 问题（对象在动态链接库（DLL）中被 new 创建，却在另一个 DLL 内被 delete 销毁）、自动解除互斥锁

weak_ptr

weak_ptr 允许你共享但不拥有某对象，一旦最末一个拥有该对象的智能指针失去了所有权，任何 weak_ptr 都会自动成空（empty）。因此，在 default 和 copy 构造函数之外，weak_ptr 只提供 “接受一个 shared_ptr” 的构造函数。

- 可打破环状引用（cycles of references，两个其实已经没有被使用的对象彼此互指，使之看似还在 “被使用” 的状态）的问题

unique_ptr

unique_ptr 是 C++11 才开始提供的类型，是一种在异常时可以帮助避免资源泄漏的智能指针。采用独占式拥有，意味着可以确保一个对象和其相应的资源同一时间只被一个 pointer 拥有。一旦拥有着被销毁或编程 empty，或开始拥有另一个对象，先前拥有的那个对象就会被销毁，其任何相应资源亦会被释放。

- unique_ptr 用于取代 auto_ptr

auto_ptr

被 c++11 弃用，原因是缺乏语言特性如 “针对构造和赋值” 的 `std::move` 语义，以及其他瑕疵。

auto_ptr 与 unique_ptr 比较

- auto_ptr 可以赋值拷贝，复制拷贝后所有权转移；unqiue_ptr 无拷贝赋值语义，但实现了`move`语义；
- auto_ptr 对象不能管理数组（析构调用 `delete`），unique_ptr 可以管理数组（析构调用 `delete[]`）；

14、线程共享资源

> a. 堆 由于堆是在进程空间中开辟出来的，所以它是理所当然地被共享的；因此new出来的都是共享的（16位平台上分全局堆和局部堆，局部堆是独享的）
>
> b. 全局变量 它是与具体某一函数无关的，所以也与特定线程无关；因此也是共享的
>
> c. 静态变量 虽然对于局部变量来说，它在代码中是"放"在某一函数中的，但是其存放位置和全局变量一样，存于堆中开辟的.bss和.data段，是共享的
>
> d. 文件等公用资源 这个是共享的，使用这些公共资源的线程必须同步。
>
> e. 地址空间
>
> f. 子进程、闹铃、信号及信号服务程序、记账信息
>
> g. 进程代码段

线程独享资源

> 栈
>
> 寄存器
>
> 程序计数器
>
> 状态字
>
> 线程优先级
>
> 错误返回码

15、Hashmap和Hashtable的区别

HashMap 不是线程安全的

HashMap 是 map 接口的实现类，是将键映射到值的对象，其中键和值都是对象，并且不能包含重复键，但可以包含重复值。HashMap 允许 null key 和 null value，而 HashTable 不允许。

HashTable 是线程安全 Collection。

HashMap 是 HashTable 的轻量级实现，他们都完成了Map 接口，主要区别在于 HashMap 允许 null key 和 null value, 由于非线程安全，效率上可能高于 Hashtable。

区别如下：

- HashMap允许将 null 作为一个 entry 的 key 或者 value，而 Hashtable 不允许。
- HashMap 把 Hashtable 的 contains 方法去掉了，改成 containsValue 和 containsKey。因为 contains 方法容易让人引起误解。
- HashTable 继承自 Dictionary 类，而 HashMap 是 Java1.2 引进的 Map interface 的一个实现。
- HashTable 的方法是 Synchronize 的，而 HashMap 不是，在多个线程访问 Hashtable 时，不需要自己为它的方法实现同步，而 HashMap 就必须为之提供外同步。
- Hashtable 和 HashMap 采用的 hash/rehash 算法都大概一样，所以性能不会有很大的差异。

16、volatile 关键字

是一种类型修饰符，用它声明的类型变量表示可以被某些编译器未知的因素更改，比如：操作系统、硬件或者其它线程等。

遇到这个关键字声明的变量，编译器对访问该变量的代码就不再进行优化，从而可以提供对特殊地址的稳定访问。

当要求使用 volatile 声明的变量的值的时候，系统总是重新从它所在的内存读取数据，即使它前面的指令刚刚从该处读取过数据。而且读取的数据立刻被保存。