目录

[一、计算机网络 1](#_Toc256484177)

[1. TCP和UDP的区别 1](#_Toc729883310)

[2. TCP的可靠传输 2](#_Toc722199506)

[3. 简述TCP的三报文握手 2](#_Toc429524498)

[4. 简述TCP的四报文挥手 3](#_Toc1325700319)

[5. TCP的流量控制和拥塞控制 4](#_Toc902423808)

[6. http输入url之后到最后的页面展示，经历的过程 5](#_Toc1507425942)

[7. Socket系统编程接口 6](#_Toc1443223535)

[8. 最大传送单元MTU 6](#_Toc430159880)

[9. Http和Https区别 6](#_Toc1267147358)

[10. Cookie和Session的区别 6](#_Toc350318607)

[二、数据结构与算法 8](#_Toc1552151422)

[1. 中缀表达式/后缀表达式 8](#_Toc1525089445)

[2. 链表奇数位升序，偶数位降序，如何改为有序链表： 8](#_Toc1960975170)

[3. HashMap的实现？在Java7和Java8中的区别？ 9](#_Toc678151681)

[4. HashMap和Hashtable的区别 10](#_Toc999587938)

[5. ConcurrentHashMap、Collections.SynchronizedMap和Hashtable的区别？分别是怎么实现线程安全的 11](#_Toc309903485)

[6. b树/b+树 12](#_Toc900028420)

[7. 选择问题 14](#_Toc2050329119)

[8. 排序 14](#_Toc1358903271)

[9. 红黑树 17](#_Toc598689852)

[X. 智力题——赛马问题 17](#_Toc1219456369)

[三、Java知识点 18](#_Toc1966750462)

[1. 请说明Java中的方法覆盖(Overriding)和方法重载(Overloading)是什么意思？ 18](#_Toc1106720210)

[2. 如何理解面向对象？如何理解继承和多态？ 18](#_Toc1290702803)

[3. Java多线程 19](#_Toc1109691674)

[4. Java修饰符总结 28](#_Toc1839974370)

[5. 为什么说Java是平台无关的语言 29](#_Toc684719790)

[6. Java内存模型 30](#_Toc1868129904)

[7. 面向过程和面向对象 31](#_Toc1448377388)

[8. 面向对象的六大原则 32](#_Toc1151621371)

[四、设计模式 33](#_Toc30271986)

[1. 创建型模式 33](#_Toc1975128010)

[五、MySQL 39](#_Toc174248744)

[1. 数据库事务 39](#_Toc1578429547)

[2. 数据库隔离级别（这里有错误，还是以《MySQL技术内幕》为准，不要看） 39](#_Toc799905038)

[3. 多版本并发控制 40](#_Toc756343446)

[4. 数据库索引 41](#_Toc908590329)

[5. InnoDB和MyISAM存储引擎的区别 43](#_Toc2068929333)

[6. 存储过程及其优点 44](#_Toc440087507)

[7. 视图 44](#_Toc617049881)

**一、计算机网络**

**1. TCP和UDP的区别**

TCP和UDP都是运输层协议，为应用进程提供端到端的逻辑通信。

（1）、

UDP是无连接的协议，在传输数据之前不需要先建立连接，数据传输之后也没有连接释放。并且UDP不提供数据报的可靠交付，而是尽最大努力的交付数据报。

TCP则是面向连接的协议。在传输数据之前必须先建立连接，数据传送结束之后要释放连接。因此TCP提供的是可靠的，面向连接的数据传输，无差错、不丢失、不重复、不失序。它包括可靠传输（停止等待协议、自动重传请求ARQ）、流量控制，拥塞控制和连接管理等技术。

（2）、

UDP没有拥塞控制，因此网络出现拥塞也不会降低源主机的发送数据的速率，可以保证数据传输的实时性。即使发生了个别报文段的丢失也可以忍受，因为主要保证数据的实时性。

TCP可以进行网络的拥塞控制，根据当前网络的拥塞情况调整发送方的发送窗口。

（3）、

UDP支持一对一，一对多，多对一和多对多的交互通信。

TCP是面向连接的，提供的是应用进程端到端的逻辑通信，通信连接只能由两个端点，因此只能是点对点的通信。

（4）、

UDP是面向报文的。应用层交给的报文简单添加首部就交付IP层，IP层上交的数据报去除首部就上交应用进程。即，UDP一次交付一个完整的报文。

TCP是面向字节流的。发送方应用进程的字节流会先写入TCP的发送缓存，接收方应用进程从TCP接受缓存中读取字节流。TCP把进程输出的数据看成是无结构的字节流，它会把要传输的每个字节都进行编号然后再传输，确保这些字节流按序并完整传输。TCP报文段一次发送的字节数并不是由应用层决定的，而是根据接收方给出的窗口值以及当前的网络拥塞程度来决定一个报文段包含多少子节。

（5）、

UDP因为提供的是尽最大可能交付数据，它的数据报首部很小。首部8字节。源端口、目的端口、长度和检验和。

TCP因为要提供可靠的面向连接的服务，因此首部要大很多。首部固定长度20字节，以及不定长的选项字段。

什么是可靠传输？确保传输的报文没有差错，差错包括比特差错，以及报文丢失、重复和失序，还有确保报文交付的时限。IP数据报的格式中首部有一个首部检验和字段，但是只检验数据报的首部是否有比特差错，不包括数据部分。但是TCP和UDP的首部也有检验和字段，检验范围包括首部和数据部分。

**2. TCP的可靠传输**

理想的传输需要满足：传输不产生差错，以及无论发送方以多快速率发送数据，接收方总能来得及处理数据。但是实际的传输网络无法具备理想的传输。因此TCP可靠的传输协议能够使得当传输的数据发生差错时能够让发送方重传数据；当接收方缓存不足不足以接受数据时能够告知发送方降低发送数据的速度。这就是TCP可靠传输的基本方法。

TCP使用流水线传输，使用连续自动重传请求协议（连续ARQ协议）和滑动窗口协议。发送方维持一个发送窗口，窗口内的字节数据都可以连续发送，接收方对收到的分组采用累计确认的方式，对按序到达的最后一个分组发送确认。发送方不需要每发送一个分组就停止发送等待对方确认，对于已经确认的分组就将其移出滑动窗口。对于每个分组都设置一个超时重传计时器，超过一定时间没有收到确认就重传已发送的报文段。同时，TCP首部有窗口字段，指出了接收方目前可接受的窗口大小，可根据此来调整发送方的发送窗口大小，保证接收方能够来得及接受和处理数据。

**3. 简述TCP的三报文握手**

TCP是面向连接的协议。TCP运输连接的建立和释放是使用TCP协议进行数据传输的关键过程。TCP连接的建立采用客户服务器方式，建立连接的过程叫做握手，握手需要在客户和服务器之间交换三个TCP报文段，称为TCP建立过程的三报文握手。

起初，服务器TCP进程处于CLOSED（关闭）状态。若服务器准备好接受连接请求，便处于LISTEN（监听）状态，等待客户TCP进程的连接请求；

（1）客户向服务器发送连接请求报文：SYN=1，选择一个初始序号seq=x。SYN同步报文段（SYN=1的报文段，表示连接建立或连接接受报文段）不能携带数据，但消耗一个序号（seq=x）。客户进入SYN-SENT（同步已发送）状态；

（2）服务器收到连接请求报文段，若同意连接，则向客户发送确认报文段（确认客户发送数据的起始序号seq=x）：SYN=1，ACK=1，ack=x+1（消耗了客户给服务器传输数据的序号x），也为自己传输数据选择一个初始序号seq=y。SYN为1同样不能携带数据，并消耗一个序号（seq=y）。服务器进入SYN-RCVD（同步接受）状态；

（3）客户收到服务器的确认还要向服务器再次发送确认（确认服务器发送数据的起始序号seq=y）：ACK=1，ack=y+1（消耗了服务器给客户传输数据时的序号y），seq=x+1。ACK报文段可以携带数据，如若不携带则不消耗序号（seq=x+1）。客户进入ESTABLISHED（连接已建立）状态；

服务器收到确认报文段，也进入ESTABLISHED状态。

为什么客户最后还要发送一次确认？

防止客户发送的已失效的连接请求报文延误到客户服务器的TCP连接释放之后才又到达服务器，让服务器误认为客户又发出了新的连接请求，于是又向客户发送了确认报文段同意连接。如果不采用三报文握手，则服务器发送了确认报文段之后就已经建立TCP连接开始等待接受或发送数据。而客户并没有重新发出连接请求报文段，不会对服务器突然发出的确认报文段进行反应，也不会等待服务器发送数据。这样会白白浪费服务器的资源。

但是如果客户最后还要发送对服务器确认的确认才能建立TCP连接，那么由于客户不会发送确认，服务器没有收到客户的确认就不会建立连接。

**4. 简述TCP的四报文挥手**

TCP的四报文挥手是TCP连接释放的过程。

起初，通信双方都是ESTABLISHED（连接建立）状态。

（1）TCP通信的双方都可以释放连接。假设是客户要求释放TCP连接：FIN=1，seq=u。此时客户进入FIN-WAIT-1（终止等待1）状态。FIN报文段即使不携带数据也消耗一个序号。

（2）服务器收到连接释放报文段后发出确认：ACK=1，ack=u+1，seq=v。服务器进入CLOSE-WAIT（关闭等待）状态。此时，从客户到服务器方向的连接就释放了，TCP连接处于半关闭状态。但是由于TCP提供全双工的通信，数据传输是双向的。服务器到客户方向的连接并未关闭。

客户收到来自服务器的确认后，进入FIN-WAIT-2（终止等待2）状态，等待服务器传输完数据后发出连接释放报文段。

（3）服务器发出连接释放报文段：FIN=1，seq=w，并重复已经发送过的确认号ack=u+1，ACK=1。服务器进入LAST-ACK（最后确认）状态，等待客户的确认。同样，FIN报文段要消耗一个序号。

（4）客户收到服务器的连接释放报文段后发出确认：ACK=1，ack=w+1，seq=u+1。客户进入TIME-WAIT（时间等待）状态，等待2MSL（MSL即最长报文段寿命）时间后进入CLOSED状态。

服务器收到客户的确认之后就进入CLOSED状态。

客户为什么要在TIME-WAIT状态等待2MSL的时间之后再进入CLOSED状态？

为了防止客户最后发送的确认报文段丢失而使得服务器收不到对连接释放的确认，于是处于LAST-ACK状态的服务器会重传连接释放报文段。客户在等待的这段时间内如果收到重传的连接释放报文段就会重传确认。最后使得双方都进入CLOSED状态。如果客户在发送完确认之后就立即从TIME-WAIT状态进入CLOSED状态，就无法收到重传的连接释放报文段并重传确认，使得服务器无法正常进入CLOSED状态。

**5. TCP的流量控制和拥塞控制**

**流量控制**指的是控制发送方发送数据的速率，使得接收方来得及接受数据。发送方发送数据的速率不仅要考虑信道的传输速率、当前网络的拥塞状况以及接收方的数据接收和处理能力。TCP利用滑动窗口实现流量控制（通过滑动窗口协议可以实现可靠传输和流量控制）。发送方的发送窗口不能超过接收方给出的接受窗口大小。并且发送方的发送窗口还受到当前网络拥塞状况的限制。

（补充：当接收方给出的窗口减小到0时，则发送方就要停止发送数据，直到接收方重新发送一个新的窗口值。若此时的携带新窗口值的报文段丢失，则此时发送方会一直等待接收方的非零窗口的报文段，而接收方却一直等待发送方发送数据，因此陷入停滞。因此TCP为每一个连接设置一个零窗口计时器。发送方收到零窗口报文段之后的一定时间后仍然没有收到非零窗口的报文段，则发送一个零窗口探测报文段。TCP规定，即使接受方窗口为0，也必须接受零窗口探测报文段、确认报文段以及携带紧急数据的报文段，即URG置为1的报文段）

**拥塞控制**指的是防止网络的负荷超过网络本身的承载力。网络的链路宽度、交换节点中的缓存和处理能力等等都是网络资源。当网络的负荷超过网络能提供的资源时就会发生拥塞，例如节点的缓存或者处理能力不足就会导致分组的丢失。分组的丢失往往是网络拥塞的征兆。由于TCP的超时重传机制，又会导致大量重传的分组注入网络中，使得拥塞恶化，网络的吞吐量下降。

拥塞控制的主要过程：（1）监测网络何时、何处发生拥塞，用什么方法来监测，用什么指标可以标志网络发生了拥塞，是什么因素导致拥塞的产生。（2）拥塞发生时如何通知产生分组的源站。（3）如何调整网络来解决拥塞问题。

TCP拥塞控制算法：慢开始、拥塞避免、快重传、快恢复。

（1）慢开始和拥塞避免：基于拥塞窗口的拥塞控制。发送方维持一个拥塞窗口cwnd，拥塞窗口大小取决于当前网络的拥塞程度而动态变化。当出现了超时就判断出现了网络拥塞，从而减小拥塞窗口。算法的关键在于合理的调整拥塞窗口的大小。

慢开始：初始拥塞窗口设置为不超过2~4个发送方最大报文段长度（SMSS），每收到一个对新报文段的确认就使拥塞窗口增加最多一个SMSS的数值。每经过一个传输轮次（往返时间RTT），拥塞窗口cwnd翻倍。拥塞窗口的大小是指数增大的，但是“慢”的意思在于初始的拥塞窗口设置的非常小，起到试探网络拥塞情况的作用。但为了防止拥塞窗口增长过大，设置了慢开始门限（ssthresh）。当cwnd超过ssthresh就改用拥塞避免算法。

拥塞避免：经过一个传输轮次/往返时间RTT，拥塞窗口只增加一个SMSS大小，拥塞窗口cwnd呈线性增长，称为（cwnd）加法增大（Additive Increase）。当出现超时时就判断为拥塞发生，调整ssthresh为当前拥塞窗口的一半：ssthresh=cwnd/2，同时cwnd=初始拥塞窗口大小，重新进入慢开始阶段。这称为（ssthresh）乘法减小（Multiplicative Decrease）。合称为AIMD算法。

（2）快重传和快恢复：解决网络拥塞的误判。由于个别报文段的丢失引起对网络拥塞的误判而启动了慢开始，降低了传输效率。

快重传：让发送方尽早知道发生了个别报文段的丢失，而并没有发生拥塞。快重传算法要求接收方即使收到失序的报文段也要发送对已收到的报文段的重复确认。发送方只要一连收到三个重复确认，就可知该确认的报文段之后的报文段已丢失，接收方没有收到无法给出确认，应当立即进行重传，而不会出现超时。如果这时网络没有发生拥塞，则发送方也会顺利收到多个重复确认而立即进行重传。如果发生了拥塞，则无法收到重复的确认，因此造成超时，就可启动慢开始。

快恢复：在快重传之后，不启动慢开始，而是执行快恢复。将ssthresh减小为当前拥塞窗口的一半：ssthresh=cwnd/2，同时设cwnd=ssthresh，并开始执行拥塞避免算法。

**流量控制和拥塞控制的区别**：*拥塞控制*是防止网络过载，使得网络的负荷在网络的承受范围内。拥塞控制是一个全局性的控制过程，涉及网络的各个组成部分如主机和路由器等，以及降低网络性能的所有因素。而*流量控制*往往是针对点对点通信量的控制，是端到端的控制过程，即接受端控制发送端的发送速率，使得接收端来得及接受和处理数据。

**6. http输入url之后到最后的页面展示，经历的过程**

（1）DNS域名解析

首先要知道网页所在的服务器的IP地址，需要从网站的域名解析到网站的IP地址。这需要DNS域名解析，向域名服务器发送域名解析请求（使用UDP用户数据报），获得域名对应的IP地址后即可进行通信。

（2）建立TCP连接

浏览器向服务器发起TCP连接，进行三报文握手。

（3）发送HTTP请求报文

与服务器建立TCP连接之后就可以发送HTTP请求报文，端口号Http：80，Https：443。

（补充：IP数据报首部中的协议号指的是直接使用IP层的协议，IP：4，TCP：6，UDP：17，IPv6：41）

（4）服务器处理请求并返回HTTP回答报文

（5）浏览器解析渲染页面

**7. Socket系统编程接口**

**8. 最大传送单元MTU**

每一种数据链路层都规定了数据帧中的**数据部分**（即IP数据报）的长度上限——最大传送单元（Maximum Transfer Unit, MTU），常用的以太网MTU默认值为1500字节。

IP数据报如果长度超过了数据链路层的MTU就要进行分片。IP协议规定，互联网中的所有主机和路由器必须能够接受长度不超过576字节的数据报。

TCP的选项中有一个最大报文段长度（Maximum Segment Size, MSS），是每一个TCP报文段中的**数据部分**的最大长度。TCP通信双方都可以规定自己的MSS，如果没有MSS选项，则MSS默认值为536字节，加上20字节的TCP固定首部，加上20字节的IP数据报固定首部，为576字节。

**9. Http和Https区别**

**10. Cookie和Session的区别**

Cookie：Cookie是一个键值对，由服务器创建并发送给浏览器，浏览器将其保存到内存或硬盘上，等到再次发送请求时附带Cookie发送给服务器端，服务器根据浏览器发回的Cookie来得到用户的信息。

Session：Session基于Cookie或URL重写来实现会话跟踪。利用Session，服务器可以将用户的一系列请求联系起来，并跟踪用户的操作信息。用户第一次连接到服务器时，服务器为其创建一个Session，并为其分配一个唯一的标识：Session ID，并将其作为一个Cookie发送给浏览器，Cookie键为JSESSIONID，此Cookie称为会话Cookie。浏览器会在内存中保存这个Session ID。当用户再次请求时会附带这个Cookie，服务器从请求中读取Session ID，根据Session ID找到对应的Session，其中就包含了用户的状态信息。Session通过HashMap数据结构来保存Session ID和对应的Session。

Session是在服务器端保存用户信息，而Cookie是在浏览器端保存信息。

当关闭浏览器之后，再次打开浏览器访问Servlet，这时开启了一个新的会话，那么保存JSESSIONID的Cookie就删除了。保存Session ID的Cookie是存储在浏览器的内存中，一旦浏览器关闭，此Cookie就删除了。这是因为Session ID Cookie创建时会设置：cookie.setMaxAge(-1), 当设置Cookie生命值为负数时，那么此Cookie只能保存在内存中，当浏览器关闭，就代表这浏览器进程结束，那么Session ID Cookie就清除了。

对于其他的Cookie，当设置的setMaxAge为正数时，可以将其保存到硬盘中，因此即使浏览器关闭之后再重新打开，依然可以读取到硬盘中的Cookie。

**二、数据结构与算法**

**1. 中缀表达式/后缀表达式**

（1）计算后缀（postfix）/逆波兰（reverse Polish）表达式：

计算一个后缀表达式的值：使用一个栈，从左到右遍历后缀表达式，当遇到一个数就将其入栈；在遇到一个运算符时就从栈弹出两个数进行运算，并将结果入栈。

后缀表达式优点：无需考虑运算符的优先级规则

（2）中缀表达式（infix）转换为后缀表达式

中缀表达式即标准形式的表达式，包含括号。中缀到后缀的转换依然使用一个栈，从左到右遍历中缀表达式

1）遇到一个数立即输出

2）遇到一个操作符时，从栈中弹出元素，直到栈顶元素的优先级严格低于当前操作符，再将当前操作符入栈。

3）+和-具有相同的优先级，\*和/具有相同的优先级，\*和/的优先级高于+和-

4）当遍历到一个左括号时，左括号具有最高的优先级，因此不会弹出栈顶元素，直接入栈。当左括号在栈中时，它具有最低的优先级，保证不会被除右括号外的其他操作符弹出。

5）在遇到一个右括号时，会持续弹出并输出栈顶元素直到一个左括号被弹出。弹出的左括号不输出。

（3）表达式树（expression tree）

表达式树的树叶是操作数（operand），常数或变量名。其他节点为操作符（operator）。对于二元操作符，表达式树为二叉树。（一个节点也可能会有多于两个的子节点，也可能只有一个子节点，对应一元操作符）

表达式树中序遍历即得到中缀表达式，后序遍历即得到后缀表达式，先序遍历得到前缀（prefix）表达式。

由后缀表达式构建表达式树类似后缀表达式的求值算法：从左往右遍历后缀表达式，遇到操作数则建立一个单节点树入栈；如果遇到操作符则从栈中弹出两棵树作为该操作符的左右子树，并将新树入栈。（虽然不是很重要，但可以将先弹出的树作为右子树）。

由后缀表达式构建表达式树，即由表达式树的后续遍历还原表达式树，可以这么做的前提是我们知道表达式树的每个节点（除了树叶）都有两个子节点，即根为操作符，左右子节点为操作数。

**2. 链表奇数位升序，偶数位降序，如何改为有序链表**：

总体思路：（1）先将奇偶数位的链分成两个链表，（2）再将偶数位链表反转，（3）最后将两个有序链表组合成一个有序链表。

（1）prev指向head，curr指向head.next。prev负责串起奇数位节点，curr串起偶数位节点。

while (prev != null && curr != null) {

prev.next = curr.next;

prev = prev.next;

if (prev != null) {  
 curr.next = prev.next;

}

else {  
 curr.next = null;

}

}

（2） 反转链表，LeetCode.206.

（3）合并两个有序链表，LeetCode.21. 双指针分别指向两个链表头，可以使用一个pointer指针，始终将pointer.next指向较小的节点，同时移动指针。也可以使用最简单的递归。

**3. HashMap的实现？在Java7和Java8中的区别？**

HashMap是一种映射表，存储键/值对的容器，可以通过键快速映射到键对应的值。HashMap将键和值组成键值对一起存储（Map.Entry<K,V>），存放在散列表内。它通过键的哈希值得到键值对在数组中的存放位置，可以通过键快速增删查改键所对应的键值对。HashMap通过分离链接法来解决散列冲突，对于不同的键而散列到同一位置的情况，HashMap将散列到同一位置的键值对存放在一个链表内。由于在散列表内存放的是键值对，键值节点内不仅存放键/值，还存放该键值对的下一个节点的指针，这样就可以构成链表。对于散列到同一位置的键值对，通过键的equals方法可以确定键值对在链表内的位置。可见，equals方法和hashCode方法对于HashMap来说是非常重要的。存放于HashMap中的键和值都必须提供equals方法和hashCode方法。

* **Java 8中HashMap相对于Java 7的区别**：

（1）引入了红黑树，Java 8 HashMap的数据结构=数组+链表+红黑树。

Java 7中键值对存放在Entry<K, V>内，而Java 8是使用结点Node<K, V>中。Node可以转换成TreeNode，树节点TreeNode可以组织成红黑树来替代链表优化冲突节点的查询。

HashMap.TreeNode<K, V> extends LinkedHashMap.Entry<K, V>

LinkedHashMap.Entry<K,V> extends HashMap.Node<K,V>

HashMap.Node<K,V> implements Map.Entry<K,V>

所以HashMap的红黑树节点既是树节点又可以当作为链表节点。当散列到同一位置的节点数超过8（static final int TREEIFY\_THRESHOLD=8）就将链表转换为红黑树，当红黑树节点少于6（static final int UNTREEIFY\_THRESHOLD = 6）就将红黑树转换为链表。利用红黑树快速增删查改的特点，避免冲突的链表过长而导致查询效率低，时间复杂度由O(N)降为O(logN)。

（2）存储过程

Java 8：接受需要存储的键值对->根据key计算哈希值hash->根据hash计算当前容量下的散列位置->若无冲突，直接放入该位置->若发生冲突，判断当前位置节点是链节点Node还是树节点TreeNode，如果是树节点就按照红黑树的插入方式插入键值对，否则将键值对插入链表末尾，即尾插法->在移动到链表末尾的过程如果计算链表节点个数超出阈值就将链表转换为红黑树（所以要使用尾插法，需要统计当前链表的长度）->结束。

Java 7：发生冲突时将新的键值对插入链表头部，即头插法。

（3）扩容之后再散列的计算方式：

Java 8中扩容之后再散列的位置只有两种，要么在原位置，要么在原位置+旧容量大小的位置。因此将原链表分为两个链表，再各自放在两个位置即可。

Java 7中则需要按照初始插入的步骤，计算哈希值并根据新的容量计算散列的位置，最后存放到相应的位置。

（4）哈希值的计算方法（这个可不提）

Java 7：使用key的hashCode进行4次位运算+5次异或运算

Java 8：使用key的hashCode进行1次位运算+1次异或运算

**4. HashMap和Hashtable的区别**

（1）Hashtable的方法是同步的，Hashtable的方法使用了synchronized关键字，线程在调用方法时需要先获得对象的内部锁，执行较慢。而HashMap不是同步的，多线程下并发访问HashMap要手动同步。

（2）HashMap允许null键和null值，null键的hash值为0。而Hashtable的键和值都不允许null值。

（3）哈希值的使用不同。Hashtable直接使用key的hashCode，而HashMap根据key的hashCode重新计算hash值。HashMap将key的hashCode与其高位16位进行异或运算得到hash值。

（4）Hashtable中的散列数组默认大小是11，扩容的方式是oldCap \* 2 + 1。而HashMap中数组默认大小是16，且容量始终是2的指数。

**5. ConcurrentHashMap、Collections.SynchronizedMap和Hashtable的区别？分别是怎么实现线程安全的**

HashMap是非线程安全的映射表，同样非线程安全的集合类还有ArrayList等。java.util.Collections中针对不同的集合类都提供了同步包装器，它是Collections中的静态方法，它将对应的非线程安全的集合类包装成线程安全的集合类。比如对于非线程安全的HashMap，有对应的Collections.synchronizedMap(Map<K, V> m)方法，将HashMap包装成线程安全的Map。这个方法返回一个Map<K, V>接口的实例，实际上是Collections中的私有的静态内部类/嵌套类Collections.SynchronizedMap<K, V>，它将HashMap包装在内部，并将HashMap的方法放到synchronized同步块中，即将HashMap对象的方法进行同步，同步的锁是包装后的SynchronizedMap对象本身。这样获得的线程安全集合的性能比java.util.concurrent包中的并发集合要低。

Hashtable同样是线程安全的映射散列表，它是在对应的方法上添加synchronized关键字来实现线程同步，线程在调用Hashtable对象上的方法时必须先获得对象上的内部锁，则执行起来较慢，并发效率不高。为了保证线程安全，同时获得较高的并发效率，可以使用java.util.concurrent.ConcurrentHashMap。

ConcurrentHashMap将整个Hash桶进行了分段，也即将整个散列表分成了几个小的片段segment，而且每个segment上都有一个锁，在插入元素时需要先找到应该插入到哪一个segment，然后获取该segment的锁，再在该segment上进行插入。ConcurrentHashMap提高了锁的数量，让锁的粒度更加精细，并发性能更好。

ConcurrentHashMap和Hashtable对比：Hashtable在多线程的并发环境下并发效率很低的原因在于，所有线程访问Hashtable都需要竞争同一把锁，即使有些线程的操作互不影响。而ConcurrentHashMap采用的分段锁方法，将容器中的数据进行分段，对不同的数据段分别进行加锁，当一个线程占用其中一段数据的锁时，其他线程可以并发访问其他数据段的锁，线程间的锁竞争与冲突大大减小，从而提高了并发访问效率。

同时，ConcurrentHashMap的get方法的高效之处在于，整个get过程不需要加锁，除非读到的值为空才会加锁重读。get方法里将要使用的共享变量都定义为volatile，比如用于统计当前segment大小的count字段，以及用于存储值的HashEntry的value。定义成volatile得变量，能够在线程之间保证可见性，能够被多线程同时读，并且保证不会读到过期的值。但是只能被单线程写入。在get操作中只需要读取而无需写，可以不用加锁。

ConcurrentHashMap的put方法首先定位到segment，获得segment的锁，在进行插入操作。插入操作需要两个步骤，第一步判断是否需要对segment里的HashEntry数组扩容，第二步定位散列位置并存放数据。

**6. b树/b+树**

b树/b+树数据结构为的是在磁盘I/O中降低查询的访问次数，使树的深度尽可能小。二叉搜索树，AVL树以及红黑树都是基于内存的数据结构，当用于磁盘读写的时候，树的深度依然太深了。要降低树的深度就需要增加树的分支。

（1）b+树：多路平衡查找树。对于M阶b+树：

b+树的每个节点都代表一个磁盘区块

数据项都存放在树叶上，且所有的树叶都在同一深度上，每个树叶都有L/2（向上取整）个数据项（设每个磁盘区块最多能够容纳L个数据项）

树的根或者是树叶，或者有至少两个儿子

非根非叶节点至少要有M/2（向上取整）个子节点。M叉b+树每个非叶节点都需要最多M-1个关键字来决定选取的分支，关键字i代表子树i+1中的数据项的最小值。

每一个叶节点都包含指向下一个叶子节点的指针。由于是查找树，且叶子节点都处于同一深度，所以叶子节点是顺序排列在叶节点层的，所以可以方便地顺序遍历叶节点。

插入：

如果叶节点未满，则直接插入树叶中合适的位置，保持有序性

如果叶节点已满，则此时有L+1项，则分裂成两个树叶节点，并将其连接到父节点，更新父节点的关键字。父节点增加一个子节点。

如果父节点也已满，则继续向上分裂父节点直到不用分裂，或者到达根节点。分裂到树根时得到两个节点，则新建一个根，并以分裂的两个节点作为新根的子节点。b+树的高度加一

删除：

如果叶节点删除一个元素之后未低于L/2，则直接删除

如果叶节点删除元素之后低于最小值，则从相邻节点借一个元素

如果相邻节点也已经达到最小值，则与相邻节点合并成一个满节点。此时父节点减少一个子节点

如果父节点也低于最小值，则同样地，或者向相邻节点借一个子节点，或者与相邻节点合并。直到根只剩下一个子节点，则删除根节点，并以根的子节点作为新的根

（2）b树：

b树与b+树类似，或者说b+树来源于b树。b树就是二叉查找树的推广，对b树进行中序遍历依然可以得到关键字的升序排列。b树同样满足：

所有叶节点在同一深度

根节点或者为树叶，或者至少有两个子节点

非根非叶节点至少有M/2个子节点

插入：

当某个节点已满，则分裂为两个节点，并将其中关键字的中位数上升到父节点中。如果父节点也满，则依然对父节点执行分裂，并将关键字的中位数上移到父节点。直到根节点，将中间关键字向上移动成为新的根。

删除：

如果删除的元素在最底层的叶节点，则直接删除。（也可能会低于最小值）

如果删除的是非叶节点中的关键字，那么需要有其他的关键字填补该删除的位置。跟二叉搜索树类似，需要选关键字的左子树的最大值或者右子树的最小值来填补到当前删除的关键字位置

当删除之后节点关键字个数低于最小值，需要向相邻节点借一个元素：将父节点中的关键字下移到当前节点，再将相邻节点中的最小关键字上移到父节点中。

如果相邻节点也达到了最低值，则与相邻节点连同父节点中的父关键字一起合并为一个节点。此时父节点中的关键字减少一个。

父节点中关键字个数也低于最小值时重复上述步骤，直到根节点。当根只有一个关键字，即两个子节点时，唯一的关键字被合并到两个子节点中，则b树的高度减一。

b树与b+树不同的地方在于：

b树的每个节点都存放数据，所以b树的每个节点都会有两个指针，一个指向该节点所保存的数据项，另一个指针指向其子节点所在的位置。而b+树的数据项都存放在叶节点，非叶节点只作为索引保存了子节点的指针，因此**b+树的节点能够保存更多的关键字**，一次性读入内存中的需要查找的关键字也就越多，相对来说I/O读写次数也就降低了。

b+树**范围查询**效率更高：b树需要中序遍历整个树，而b+树只需要遍历叶结点中的链表。由于b树的每个节点都存放数据项，因此在做范围查询的时候性能不如b+树。b+树的所有数据都在叶子节点，因此范围相近的数据都聚集在相同或相邻的叶节点，同时叶节点还有指向下一个叶节点的指针，可以很方便的进行范围查询。而b树每个节点都是数据项，只是范围相近的数据可以在同一个节点内，但同一范围内的数据项并不聚集在一处，因此不利于范围查询

b+树的叶子节点有指向下一个叶节点的指针，方便数据项的**顺序遍历**

b+tree的查询效率更加稳定。由于非叶节点并不是最终指向数据内容的结点，而只是叶子结点中关键字的索引。所以任何关键字的查找必须走一条从根结点到叶子结点的路。所有关键字查询的路径长度相同，导致每一个数据的查询效率相当

（3）b\*树

b\*树是b+树的变体，在b+树的非根和非叶子结点再增加指向兄弟的指针；b\*树定义了非叶子结点关键字个数至少为(2/3)M，即块的最低使用率为2/3（代替b+树的1/2M）。

b+树的分裂：当一个结点满时，分配一个新的结点，并将原结点中1/2的数据复制到新结点，最后在父结点中增加新结点的指针。b+树的分裂只影响原结点和父结点，而不会影响兄弟结点，所以它不需要指向兄弟的指针。

b\*树的分裂：当一个结点满时，如果它的下一个兄弟结点未满，那么将一部分数据移到兄弟结点中，再在原结点插入关键字，最后修改父结点中兄弟结点的关键字（因为兄弟结点的关键字范围改变了）。如果兄弟也满了，则在原结点与兄弟结点之间增加新结点，并各复制1/3的数据到新结点，最后在父结点增加新结点的指针。

所以，b\*树分配新结点的概率比b+树要低，空间使用率更高。

**7. 选择问题**

**8. 排序**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 排序方法 | 描述 | 空间 | 平均时间 | 最坏时间 |
| **选择**排序 | 每次查找当前最小的元素与头部元素互换 | O(1) | O(N2) |  |
| **冒泡**排序 | 遍历一趟数组，如果相邻元素为逆序则交换两个元素，最大值会逐渐移到数组末尾 | O(1) | O(N2) |  |
| **插入**排序 | 将位置i的元素插入到前i个已排序元素中的合适位置，使前i+1个元素有序 | O(1) | O(N2) |  |
| **希尔**排序 | 又称缩减增量排序，实际上为间隔一定距离的插入排序。增量序列的最后一个一定为1。所有相隔hk的元素都已排序则称数组为hk排序。希尔排序基于：hk排序的数组始终保持hk排序性。一趟hk排序就是对hk个独立的子数组进行一次插入排序。希尔排序的时间依赖于增量序列的选择。  Shell增量序列：ht=N/2, hk=hk+1/2  Hibbard增量序列：1，3，7…2k-1 | O(1) |  | Shell增量序列：O(N2)  Hibbard增量序列：O(N3/2) |
| **堆**排序 | 将原数组构建成最大堆，从最后一个元素的父节点nums.length/2-1开始从右往左从下至上下滤。构建好最大堆之后，将堆顶元素与末尾元素对调等价于删除堆顶最大值，同时将交换后的堆顶元素下滤保持堆序性。重复上述步骤，同时注意每次删除最大值后堆顶元素个数减少1。 | O(1) | O(NlogN) |  |
| **归并**排序 | 数组两端为left和right，mid=(left+right)/2。递归排序left到mid，mid+1到right两个部分，再将两部分合并。这里需要一个额外的数组，两段子数组从左到右，将较小的复制到临时数组，同时移动指针。当两个子数组合并到临时数组后要复制到原数组，此时需要根据提前记录总的元素个数以及右半个数组的末端来定位复制数组。 | O(N)归并排序使用线性附加内存，但是比较次数是最少的 |  | O(NlogN) |
| **快速**排序 | 选取枢纽元：三数中值分割法（median3）  根据枢纽元进行分割（partition）  递归调用排序小于枢纽元的部分和大于枢纽元的部分 | O(1) | O(NlogN) | O(N2) |
| **桶**排序 | 输入的数据必须为非负整数，最大值小于M，则创建一个大小为M的数组count。对于元素Ai，count[Ai]++，读取完毕之后顺序扫描count数组。 | O(M) | O(M+N) |  |
| **基数**排序 | 使用多次桶排序。从低位到高位，依次对每一位进行桶排序。这样保证高位相同的元素可以按照低位数字的顺序正确排序。对于按低位进行桶排序的数，每个桶之间的数的大小关系并没有完全确定，因为可以由更高位数字确定大小关系，今次需要再次进行桶排序，但是依然只需要之前的10个桶对更高位进行排序。但是如果是从高位到低位进行桶排序，由于高位排序已经完全确定了桶之间的数的大小关系，如果对更低位的数重新进行桶排序会完全打乱大小顺序。因此需要的是对每个桶内的数字再次对更低位进行桶排序，这样对每个桶需要再开辟10个桶。总的需要10n个桶，n为最大数字的位数。基数排序还适用于字符串排序。可排序定长或不定长的字符串 | O(N) | 数字排序：O(10N);字符串排序：O(NL)，字符串长度都为L。即与字符总个数呈线性关系 |  |
| **计数基数**排序 | 计数基数排序目的是不使用ArrayList数组。将字符串按某位进行桶排序之后，需要从左到右复制到原数组中。可以使用一个count数组来保存每个桶中元素复制到原数组中时的偏移量，这个偏移量就是所有当前字符小于该桶所代表字符的字符串个数。但是这就需要另一个临时数组，用来将原数组中的字符串按照当前桶排序的结果复制到临时数组中。 | O(N) | O(NL)，关于所有字符串中字符总个数是线性的。 |  |
| **外部**排序 | 当要排序的数据无法一次性装入内存时，需要将数据从磁盘一部分读入内存进行排序再写入磁盘。而磁盘上的数据访问时间要比排序的时间长的多。这叫外部排序。外部排序的基本算法是归并排序。假设内存一次性可以容纳及排序的数据个数是M，从磁盘读入M个数据并在内存将其排序后交替写入两个磁盘中，每组M个叫做一个顺串。接着合并两个排序的顺串，则不需要全部读入内存，使用两个指针指向两个顺串的开头，将其中的较小者写入另一个磁盘即可。 | 对于外部排序以及不适用 | 对于外部排序，时间复杂度的分析方法已经不再适用。 |  |

**9. 红黑树**

**X. 智力题——赛马问题**

**三、Java知识点**

**1. 请说明Java中的方法覆盖(Overriding)和方法重载(Overloading)是什么意思？**

* 方法重写发生在通过继承而相关的不同类中；方法重载可以发生在同一个类中，也可以发生在**由于继承而相关的不同类**中。**注意**：如果子类中的某个方法和父类中某个方法名字相同，但是参数列表不同，这也是方法重载！
* 方法重写具有同样的签名；方法重载具有同样的名字，但是不同的参数列表。

**方法重载**：

（1）Java中的方法重载发生在**同一个类**内或**具有继承关系的不同类中**两个或者是多个方法的**方法名相同**但是**参数列表不同**的情况。被重载的方法具有相同的方法名但必须具有不同的参数列表。不能基于不同修饰符或返回值类型来重载方法。

（2）重载方法使得你可以使用同样的名字来定义不同方法，只要它们的签名是不同的。执行同样功能但是具有不同参数类型，处理不同类型的数据的方法应该使用同样的名字，进行方法重载。

（3）重载方法可以使得程序更加清楚，以及更加具有可读性。

**方法重写**：

（1）方法覆盖也叫方法重写，是指子类继承自父类，同时继承了父类的属性和方法，但有时子类并不想使用父类中定义的方法，需要重新实现该方法，需要在子类中使用和父类一样的签名以及一样的返回值类型来对该方法进行重新定义，这就叫方法重写。

（2）父类中的私有方法不能重写/覆盖。如果子类中定义的方法在父类中是私有的，那么这两个方法完全没有关系。

（3）实例方法可以重写，但是静态方法不能重写。如果父类中定义的静态方法在子类中被重新定义，那么在父类中定义的静态方法将被隐藏。

（4）方法重写不能降低父类中方法的可访问性

（5）方法重写可以返回父类中同名方法的返回类型的子类。

**2. 如何理解面向对象？如何理解继承和多态？**

如何理解多态：按照多态的定义，多态指的是子类继承自父类，则子类的实例也必定是父类的实例，父类的变量可以引用子类的实例，也即可以向需要父类对象的地方传递其子类的实例。从宏观语义上面来理解，多态表示的是父类与子类的包含与被包含的关系。父类具有更加广泛通用的概念，子类包含在父类的范围内，因此也是属于父类的一元。子类可以看作是父类的特殊化，可以根据不同的特点划分出不同的子类，因此说父类的实例不一定是子类的实例，因为可以有父类派生出无数的子类，它们之间互不相同。当使用一个父类的变量引用一个子类的对象时，就是将子类的对象放在父类的层面进行操作，而一定程度上剥离了子类的特殊性，把他放在一个更加宏观宽泛的角度来对待。可以对该变量调用父类的方法，因为子类拥有父类的所有共同的属性和行为。但是，通过方法重写，可以让不同的子类对象根据自身的不同而呈现出不同的行为，也即根据子类对象的特殊性调用最接近子类的方法。这就是多态的表现。

**3. Java多线程**

* 多线程是Java最基本的一种并发模型。

**（1）程序、进程与线程**

**程序**：程序是一组指令的有序集合，是指令、数据及其组织形式的描述，是一个静态的实体。

**进程**：进程是正在执行中的程序，是具有一定独立功能的程序关于某个数据集合上的一次运行活动。进程的组成包括程序、数据和进程控制块（Processing Control Block， PCB），是一个动态的实体。

程序与进程并不是一一对应的，一个程序执行在不同的数据集上就成为不同的进程，可以用进程控制块唯一地标识一个进程。

**线程**：进程内部需要同时执行多个子任务，这些子任务称为线程。线程不能单独执行，必须依赖于进程而存在。一个进程至少有一个线程，进程内的多个线程共享进程内的资源。进程是线程的容器。操作系统调度的最小任务单位不是进程而是线程。

一个Java程序实际上是一个JVM进程，JVM进程用一个主线程来执行main()方法，在main()方法内部可以启动多个线程。

一个应用程序既可以有多个进程，也可以有多个线程。实现多任务的方法可以有：多进程（每个进程只有一个线程）、多线程（一个进程有多个线程）、多进程+多线程。

**（2）进程与线程**

进程（Process）是系统进行**资源**分配和**资源**调度的基本单位，线程（Thread）是操作系统调度和分派的基本任务单位。进程有自己的独立地址空间，而线程也有自己线程私有数据，比如虚拟机栈以及寄存器资源等，但是多线程共享所属进程的地址空间与全局变量等资源。线程是比进程更轻量级的调度执行单位，线程的引入，可以把一个进程的资源分配和执行调度分开，各个线程既可以共享进程资源，如内存地址与文件I/O等，又可以独立调度，线程是CPU调度的基本单位。

每个进程拥有独立的资源，而线程共享进程的数据。共享数据使得线程之间的通信比进程之间通信更快速。

创建进程比创建线程的开销大。

多进程稳定性比多线程高。在多进程情况下，一个进程崩溃不会影响其他进程。而在多线程下，任何一个线程崩溃导致整个进程崩溃。

**（3）创建线程**

(i) 重写Thread中的run()方法：定义一个Thread的子类并重写run()方法，或使用匿名内部类：Thread t = new Thread() { public void run() {…}}；

(ii) 将Runnable的实例传递给Thread的构造器，创建Thread的实例。可以定义一个类实现Runnbale接口，并定义接口的run()方法，或使用匿名内部类或lambda表达式。

(iii) Callable<V>接口与Runnable类似，封装异步执行的任务，但是有返回值，可以用Future<V>保存计算的结果。但是由于Thread构造器只接受Runnable的实例，所以需要用FutureTask<V>类，它实现了Runnable和Future<V>接口，可以将Callable<V>实例包装成Runnable和Future<V>的实例，可将FutureTask<V>对象传递给Thread构造器。其实跟第二种方法是一样的。

**（4）中断线程**

(i) Thread.currentThread().interrupt()请求终止线程，注意interrupted方法只是向t请求终止，线程t可以选择如何响应中断，这要看具体的代码。可以将要执行的代码放在while(! t.isInterrupted()) {…}中。

(ii) t.isInterrupted()方法：对一个线程调用interrupt方法会将线程的中断状态置为true，t.isInterrupted()方法返回true。线程的while循环如果循环检测中断状态，就能响应interrupt的请求而立即结束执行。

(iii) 如果线程处于等待状态（调用sleep/wait/join方法时）则无法中断线程或检测中断状态，否则会抛出InterruptedException异常。

(iv) 在线程中断状态被置为true时调用sleep方法会清除中断状态并抛出InterruptedException异常。

(v) Thread.interrupted()静态方法，检测当前线程是否中断，如果是会清除该线程的中断状态。而isInterrupted实例方法不会改变中断状态。

**（5）volatile关键字**

(i) 为什么要使用volatile关键字

线程间共享的变量需要使用volatile关键字标记，确保每个线程都能读取更新后的变量值。

在Java内存模型中，Java虚拟机中变量的值保存在主内存中，而线程访问变量时，会从主内存获取一个变量的副本到自己的工作内存中。如果线程修改了变量的值，虚拟机会在某个时刻将修改后的值回写到主内存中，但是更新的时间却是不确定的。这会导致写线程更新了变量的值，而读线程读取的仍然是未更新的值，造成多线程之间共享的变量不一致。

volatile关键字指示线程在每次访问变量时都进行一次刷新，总是获取主内存的最新值，而每次修改变量后总是立刻写到主内存。这样使得不同的线程都能及时看到变量的最新值，保证变量修改的实时可见性。也就是强制线程从主内存读写。

volatile关键字解决的是可见性问题：当一个线程修改了某个共享变量的值，其他线程能够立刻看到修改后的值。但是volatile关键字不提供原子性。在变量的读取、操作和写入之间不保证不被中断。

(ii) 使用volatile的条件：

* 对变量的写入操作不依赖于变量的当前值，或者确保只有单线程更新变量的值。否则volatile关键字不保证变量的读取、运算、重新写入不被中断。
* 该变量没有包含在具有其他变量的不变式中。总之，volatile不提供原子性。
* volatile是针对实例域的多线程同步**读写**操作而提供的一种免锁机制。如果要访问的变量已在synchronized同步块中或为常量，则没有必要使用volatile。

(iii) volatile和synchronized的区别：

* volatile不会进行加锁操作。volatile变量是一种较弱的同步机制，在访问volatile变量时不会进行加锁操作，因此也就不会阻塞其他线程。所以volatile是一种比synchronized关键字更轻量级的同步机制。
* volatile不如synchronized安全。仅当volatile变量能简化代码时才使用它，一般来说使用synchronized会更安全。
* volatile只能确保内存可见性，而不保证原子性。synchronized关键字既可以保证可见性也可以保证原子性。

**（6）线程同步**

多线程同时读写**共享变量**时会导致数据出错，这是由于线程的调度由操作系统决定，任何线程都有可能在任何指令处被中断。假如某线程在读取了某个数据之后被暂时中断，而之后共享变量被其他并发的线程所修改，此时中断后的线程继续执行修改数据并写回，但是此时变量值已经改变，必须重新刷新获得最新的值，否则会覆盖之前的修改，也就意味着某些修改被抹去了，从而造成错误（所以volatile保证了修改的实时可见性）。

线程安全问题来自于操作的非原子性。原子性用于标识事务是否**完全地**完成，一个事务的任何更新要在系统上完全完成，如果由于某种原因出错，事务不能完成它的全部任务，系统将返回到事务未开始的状态。也即一个任务**要么完整的被执行，要么完全不执行**。原子性达到的目标：就是能使一个程序被完整的执行。而原子操作：不可再分的不可被中断的一个或一系列的操作。

可以通过加锁来保证一段代码指令的原子性。即使线程在执行期间被中断，其他线程无法获得锁就无法进入临界区，只有执行线程执行完将锁释放，其他线程才有机会获得锁并执行，保证任何时刻最多只有一个线程能执行临界区的指令。但是对于JVM规定的原子操作则不需要加锁同步。

线程同步的“同步”二字指的是线程间的协同步调，协同与互相配合，而不是指同时进行。同步指的是多线程在访问共享资源时通过某种机制来控制线程的执行顺序，保证共享数据在同一时刻只被一个或一些线程使用。

**（7）synchronized关键字**

使用synchronized(lock)把lock对象当作锁，lock可以是任何类型共享的实例。JVM保证同一个锁在任意时刻只能被一个线程获得。任何线程在执行同步块内的代码时必须先获得lock，执行结束后会自动释放锁，即使抛出异常也会正确释放锁。synchronized块无法并发执行。

当synchronized(lock)中的lock为this时：synchronized(this)表示将代码块所在的整个对象作为锁，对整个对象加锁，则可以用synchronized修饰同步块所在的方法，称为同步方法。从Java 1.0开始，每个Java对象都有一个内部锁。用synchronized关键字声明的方法即表示该方法使用了对象的内部锁，调用该方法必须获得内部的对象锁。使用synchronized无需使用显示的锁对象。

将静态方法声明为synchronized，则调用静态方法获得的是类对象的内部锁，即锁住的是该类的Class实例，其他线程无法调用同一个类的所有同步静态方法。

**（8）条件对象Condition**

条件对象：解决线程间的协调问题。当线程获得一个锁时，还需要满足某些条件才能继续执行，否则进入等待状态，等待其他线程唤醒。

一个锁对象可以有一个或多个相关的条件对象：

Lock lock = new ReentrantLock(); Condition condition = lock.newCondition();

当满足某个条件时：while(!notOkToProceed) { condition.await() }; 则进入该条件的等待集而进入等待状态，同时会释放已获得的锁。await()方法也可以指定等待的时间，在一定的时间内如果没有被其他线程激活则自己唤醒去执行其他任务。等待其他线程调用condition.signalAll/signal方法激活等待的线程。被激活的线程从await()方法返回后会重新试图获得锁，如果重新获得了锁会从被阻塞的地方继续执行。所以应该使用while语句进行条件的判断，当线程重新获得锁之后应该重新判断是否满足条件，满足条件才可以继续执行。如果只使用if语句，那么在被激活的线程重新获得锁之后会不进行判断而继续执行。所以被唤醒之后重新获得锁的线程应该总是再次执行判断。

对于使用synchronized关键字的方法，内部对象锁只有一个相关的条件对象。可直接调用wait()/notify()/notifyAll()方法，这些方法是Object类的final且native的方法，由JVM的C代码实现，必须在synchronized方法/块内才能调用wait/notify/notifyAll方法。

**（9）读写锁 悲观锁 乐观锁**

ReadWriteLock rwlock = new ReentrantReadWriteLock();

Lock rlock = rwlock.readLock(); Lock wlock = rwlock.writeLock();

读写锁保证(i)同时只允许一个线程写入，此时其他线程既不能写入也不能读取；(ii)多个线程允许同时读取，此时不能写入。

ReadWriteLock允许多线程同时读，但是在读的过程中不允许写。这是一种悲观的读锁。

悲观锁：读的过程中拒绝排斥写入，必须等待读取完毕。

乐观锁：乐观地估计读的过程中大概率不会有写入。乐观锁的并发效率更高，但是存在读取的时候发生了写入导致读取的数据不一致，需要进行检测，如果确实发生了写入则进行重读。

Java 8新的读写锁：StampedLock，同样，写的过程中不允许读取，但是读的过程中允许获得写锁。但是需要额外得代码来判断读的过程是否有写入。这是一种乐观的读锁。

为什么乐观的读锁可以提高并发效率呢？个人理解：获取读锁进行读取数据之后往往不会马上释放锁，可能还会有其他操作，比如赋值和计算等。等到赋值计算完成返回之后才会释放锁。我们只要估计在读取数据的过程中没有发生数据的写入，则在完成后续赋值以及计算返回结果的过程中，就可以允许其他线程获得写锁写入数据。但是存在可能，即在读取的过程中被中断，后续读取的是已经被其他线程更新后的新值，造成读取的数据不一致的情况。此时就需要进行重读。重读时要先获得一个悲观读锁进行读取。

**（10）阻塞队列Blocking Queue**

对于多线程同步问题，可以不需要显式的线程同步（加锁或使用synchronized），而可以使用队列数据结构来实现同步。使用生产者-消费者模式，生产者线程向队列尾插入元素，而消费者线程从队列头取出元素，以此安全地从一个线程向另一个线程传递数据。当然，对于队列来说，线程安全的队列的实现仍然是需要考虑的。当试图向已满的队列添加元素时，或从已空的队列中取出元素时，阻塞队列会导致线程阻塞。因此可以将阻塞队列当作线程管理工具。

阻塞队列介绍：

LinkedBlockingQueue/LinkedBlockingDeque: 没有容量上界/或可指定最大容量

ArrayBlockingQueue: 构造时指定容量，可指定公平性

PriorityBlockingQueue: 带优先级的阻塞队列，没有容量上限/或指定容量

阻塞队列方法：

(i) 抛出异常的方法：add/element/remove

(ii) 返回null或boolean的方法：offer (队列满则返回false)/peek (队列空返回null)/poll (队列空返回null)

(iii) 阻塞的方法：put (队列满则阻塞)/take (队列空则阻塞)

**（11）Atomic类**

java.util.concurrent.atomic包中提供了能够不使用锁而实现原子性操作的类，如AtomicInteger类，可以使用incrementAndGet()/decrementAndGet()方法来以原子方式将AtomicInteger进行自增或自减并返回自增后的值，也就是在获得AtomicInteger的原值、增减一以及更新新值能够保证即使在多线程下也能计算正确的值。除了自增自减，还可以通过Atomic类的compareAndSet()方法来进行原子更新：

do {

oldValue = atomicObj.get();

newValue = method(oldValue, observed);

} while ( !atomicObj.compareAndSet(oldValue, newValue));

它将读取到的旧值oldValue与atomicObj的当前值进行比较，如果没有变化则说明没有其他线程对其进行更新，则将其设置为newValue，否则，则说明其他线程对其进行了更新，则不执行任何操作，compareAndSet方法返回false，并重新读取更新后的值并再次尝试修改，但是在修改失败后始终不会阻塞。最终会在没有其他线程修改的情况下成功将新值替换原来的值。

通常情况下，我们并不需要直接用do ... while循环调用compareAndSet实现复杂的并发操作，而是用incrementAndGet()这样的封装好的方法，因此，使用起来非常简单。可以使用updateAndGet/accumulateAndGet方法实现更加复杂的更新并返回新的值，需要传递另一个操作数以及一个lambda表达式来指示所需要的操作。或者使用getAndUpdate/getAndAccumulate方法返回原值。

当有大量线程要访问Atomic类的值时，存在高度竞争的情况下，Atomic类的更新需要太多重试，性能会降低。可以使用Java 8提供的LongAdder和LongAccumulator类，它们将LongAdder的当前值划分为多个加数，多线程分别更新不同的加数，并且当线程个数增加时会自动提供新的加数。如果要获取当前值需要将各个加数求和，因此需要在所有任务完成得最后才需要当前值。

**（12）CAS机制**

Atomic类是通过无锁的非阻塞方式实现的线程安全的原子操作类，它不依靠同步就可以进行变量的原子操作。Atomic类的方法是使用Unsafe类的compareAndSwapInt/Long方法来进行原子更新，它的原理是**CAS机制**：Compare and Swap，即“比较并交换”。CAS包含3个操作数：需要读写的内存位置V、进行比较的旧的预期值A以及拟写入的新值B。当且仅当内存位置V的值等于A的值时，也即变量的旧的值与内存地址V的实际值相同时，CAS才会通过原子方式将内存位置V的值更新为新的值B，否则不会执行任何操作。无论V位置的值是否等于A都会返回V原有的值。CAS的典型使用模式是：首先从内存位置V读取值A，并根据A计算新值B，然后再通过CAS以原子方式将V中的值由A变成B。CAS能检测到其他线程的干扰，因此不使用锁也能够实现变量的原子的读-改-写操作序列。

CAS是一种乐观的技术，它不进行加锁而是乐观地估计能成功执行更新操作而不会与其他线程冲突，如果有其他线程在最近一次检查前更新了该变量，那么CAS能检测到这个错误，使得更新失败并重试。当多个线程尝试使用CAS同时更新同一个变量时，只有其中一个线程能够更新变量，而其他线程都会失败，但是并不会被阻塞，失败的线程可以决定是否重新尝试，或者执行其他的操作，或者干脆不执行任何操作。

**（13）Callable和Future**

Runnable接口：void run(); 封装一个异步运行的任务，没有参数和返回值，即没有任务处理的返回结果。

Callable<V>接口：V call() throws Exception; 与Runnable类似，但是有指定类型的返回值。它是一个泛型接口，有一个参数化的类型，类型参数是返回值的类型。

Future<V>接口：保存任务执行的结果。get()方法获得计算的结果，任务完成之前调用会阻塞，直到任务完成。isDone()查询任务是否执行完成。cancel()取消任务，isCancelled()查询是否被取消。

FutureTask<V>包装类：实现了Future和Runnable接口，可将一个Callable对象包装成FutureTask对象，可以将FutureTask对象传递给线程执行，此时它相当于Runnable对象，也可以对FutureTask对象调用get()方法获得任务执行完成的结果。

**（14）执行器Executors与线程池**

使用线程池的原因：创建线程需要与操作系统交互，需要操作系统资源，而且频繁创建和结束线程需要消耗时间，降低性能。可以使用线程池，它包含一组线程，可以将任务交给线程池，空闲的线程会执行任务，执行完任务的线程不会死亡，如果没有任务需要处理则会在线程池进入等待状态，等待新的任务。如果任务数大于可用的线程数，则会创建新线程或将任务放到一个队列中等待执行。使用线程池可以复用线程，减少并发线程的数目。

Executors执行器类的静态工厂方法可以返回实现了ExecutorService接口的ThreadPoolExecutor类线程池对象：

Executors.newCachedThreadPool(): 必要时创建新线程，空闲线程会保留60秒；

Executors.newFixedThreadPool(): 包含固定数目的线程，空闲线程会一直保留；

Executors.newSingleThreadExecutor(): 只有一个线程的线程池。

使用threadPoolObj.submit()向线程池提交Runnable/Callable对象，返回一个Future对象用来查询任务的执行状态或者得到任务执行的结果，或者取消任务。使用threadPoolObj.shutdown()关闭线程池，被关闭的线程池不会再接受新的任务，等待任务都完成之后再关闭线程池。而shutdownNow()会立刻停止正在执行的任务。awaitTermination()方法会等待指定的时间关闭线程池。

Executors类的下面两个静态方法返回实现了ScheduledExecutorService接口的线程池对象：

Executor.newScheduledThreadPool(): 需要预定执行或重复执行任务而使用的线程池，可以初始的指定延迟后执行一次（schedule()），或者固定时间间隔触发一次任务（scheduleAtFixedRate()），或者任务执行完后等待固定时间间隔再次执行（scheduleWithFixedDelay()）。

Executors.newSingleThreadScheduledExecutor(): 用于预定执行或重复执行任务的单线程池

使用线程池的另一个优点是：可以控制一组相关的任务。可以向执行器对象的invokeAny()方法提交一个Callable对象的集合，它会返回已经完成了的任务的其中一个结果，即Future对象。而invokeAll方法可以接收一个Callable对象的集合，返回一个Future对象的列表，代表所有任务的执行结果。这个Future列表按照对应的Callable对象的顺序，如果某个任务执行比较费时，则其他任务就不得不等待。

可以用常规的执行器来构造一个ExecutorCompletionService对象，将任务一个一个提交submit()给ExecutorCompletionService对象，它会维持一个Future对象的阻塞队列，队列中按照任务执行完成的先后顺序入队Future对象，其中包含了已经提交的任务的执行结果。可以使用take()方法从ExecutorCompletionService对象中移除并返回一个已经完成的任务的结果，即Future对象，如果没有任何已经完成的结果则阻塞。而使用poll()方法在没有可用的结果时会返回null。

**（15）Fork-Join线程池**

ForkJoinPool线程池可以将任务分解成多个子任务并行执行，可以通过调用invoke(task)执行任务，而这个任务必须继承自RecursiveTask<V>抽象类（任务会返回一个类型为V的结果）或者RecursiveAction抽象类（没有返回结果，这两个抽象类都继承自ForkJoinTask<V>抽象类，而ForkJoinTask<V>实现了Future<V>接口），重写RecursiveTask<V>/RecursiveAction抽象类中的compute方法来划分子任务：

protected V compute() {}

在compute中规定了分裂子任务的条件，当需要分裂子任务时在compute内部递归创建两个子任务，并调用ForkJoinTask<V>中的invokeAll(subtask1, subtask2)方法并行执行子任务，最后调用子任务的join()方法返回任务执行的结果，将子任务的结果组合返回得到总结果。

Arrays.parallelSort(array)方法内部就是通过ForkJoin对大数组进行拆分并行排序。

**（16）ThreadLocal类**

可以在一个类或方法内定义一个ThreadLocal类，在每个线程执行方法时ThreadLocal会为每个线程生成线程独有的实例，即生成一个线程的局部变量，这个局部变量不用在同一个线程的所有方法之间传递，一个线程中的所有执行的方法都可以通过调用ThreadLocal类实例的get()方法获得属于当前线程的实例，从而可以避免方法之间传递共享对象就可以实现**在线程内传递状态和信息**。

ThreadLocal类实例通常设置为静态变量，通过：

public static ThreadLocal<V> threadLocalObj = new ThreadLocal<>();

方法创建一个ThreadLocal实例，线程调用threadLocalObj.set(instance)就可以将一个实例关联到ThreadLocal中，当前线程的所有方法都可以通过调用threadLocalObj.get()获取到属于当前线程的实例instance。

也可以通过调用:

public static ThreadLocal<V> threadLocalObj = ThreadLocal.withInitial(Supplier<? extends V> supplier)方法向ThreadLocal对象传递一个初始化方法，当某个线程首次调用threadLocalObj.get()方法时会调用初始化方法生成一个属于当前线程的实例，在此之后调用get()方法都会返回属于当前线程的实例。

Supplier<V>接口：V get(); 可以为ThreadLocal提供一个构造方法用于创建一个实例。

**（17）synchronized和lock的区别**

synchronized是JVM层面提供的同步，依靠编译器和JVM协作实现的同步。而lock是一个类，通过lock对象的lock和unlock方法来加锁和解锁。

synchronized使用相对简单，只需添加关键字即可实现同步，线程进入同步方法或同步块之后其他线程阻塞，直到线程执行完毕退出或者抛出异常，即使抛出异常同步锁也会自动释放，这是通过编译器来自动生成异常处理表和成对的monitorenter和monitorexit指令来实现的。而使用lock需要手动调用unlock来释放锁，通常是在try-catch-finally中来保证释放锁

synchronized虽然使用简单，但是没有lock灵活。synchronized只有一个对应的条件对象，在synchronized内使用的wait/notify方法是继承自Object内的方法。而一个锁lock实例可以有多个条件对象。同时，如果线程在同步方法/块内一直循环，那么其他线程都会一直阻塞，但是可以使用lock的tryLock方法来尝试获得锁，否则就执行其他任务。所以lock使用较为灵活。

synchronized是可重入的、重量级的非公平锁，而lock也是可重入的，但是可以实现公平锁策略。

**4. Java修饰符总结**

* 可见性修饰符：public、protected、（default）、private。其中protected和private只可以用于类成员（包括内部类），而常规类只可以有包可见性（default）或公有可见性（public）。
* 其它修饰符：static（只可以用于类成员）、final（可用于常规类或类成员）

（1）常规类修饰符

* public: 公有可见性
* (default)：包可见性
* final：表示类不可扩展

（2）类成员修饰符

(i) 数据域：

* public：公有可见性
* protected：包和子类可见性
* (default)：包可见性
* private：仅类内可见
* final：常量域，必须初始化
* static：表示静态域/类域，该域属于整个类

(ii) 方法：

* public：公有可见性
* protected：包及子类可见性
* (default)：包内可见性
* private：类内可见性
* final：表示该方法不可重写
* static：静态方法，属于整个类

(iii) 内部类：

* public：公有可见性
* private：外部类内私有可见性

public/private内部类，包含一个对外部类对象的引用，可以访问自身的数据域，也可以访问创建它的外部类的对象的数据域。

内部类的静态域必须是final。静态域是属于整个类的，对于该类的所有实例都共享一个静态域，即对该类的所有实例，静态域都是一样的。但是对于每个外部类对象，都可以有一个单独的内部类实例，如果要对这些外部类对象内的内部类对象来说，静态域始终都是唯一的，就必须定义为final。

内部类不能有static方法。

* static：静态内部类/嵌套类，不包含对外部类对象的引用，不可访问外部类的数据域。

**5. 为什么说Java是平台无关的语言**

Java的平台无关性指的是Java语言程序编译后形成的特定的存储格式——字节码存储格式，它是与操作系统和机器指令集无关的平台中立的格式。

Java程序经编译后由Java虚拟机执行，而Java虚拟机有许多运行在各种不同平台上的版本，但是这些虚拟机都可以载入和执行同一种平台无关的Class类文件，这种Class类文件采用的是各种不同平台的虚拟机都统一使用的程序存储格式——字节码存储格式。

**虚拟机**和**字节码存储格式**是构成Java语言平台无关性的基础，可以实现一次编写，到处运行。

实际上，Java虚拟机并不是跟Java语言绑定的，而是只跟Class类文件格式这种特定的二进制文件格式关联。只要某种语言的编译器能够将程序代码编译成规范的存储字节码的Class类文件，就可以运行在Java虚拟机上，而不关心字节码文件是来自于那种语言。字节码文件可以从任何途径产生，甚至可以使用十六进制编辑器直接编写Class文件。因此虚拟机以及体系结构中立的字节码存储格式同样也可以实现语言无关性。

Java虚拟机和不依赖特定硬件和操作系统的平台中立的Class文件格式是Java技术体系实现平台无关、语言无关特性的基础和支柱。

**6. Java内存模型**

（1）Java内存模型

处理器运算速度与内存读写速度之间的巨大差异，使得不得不在内存与处理器之间加入读写速度接近处理器的高速缓存（Cache）作为缓冲。CPU将需要的数据读取到高速缓存，运算结束后将结果从缓存同步回内存。

导致的问题：缓存一致性。多处理器有各自的高速缓存，但是共享同一主内存，那么就会出现缓存中数据的一致性问题。

Java内存模型是对内存和高速缓存读写访问过程的抽象，定义虚拟机中将变量存储到内存和从内存中读取变量的过程，目的是屏蔽不同硬件和操作系统的内存访问的差异，使得Java程序在各种平台下都能达到一致的内存访问效果。

Java内存模型中，所有变量（指的是实例/静态字段、数组对象元素等线程共享变量，但不包括线程私有的局部变量和方法参数）都存储在主内存中，而每条线程有自己的工作内存，可以类比为处理器各自的高速缓存。线程使用到的变量需要从主内存读取变量的拷贝值到自己工作内存中，线程对变量的所有读取、赋值操作都必须在工作内存中进行，而不得直接读写主内存中变量。不同线程间的工作内存相互独立，线程间变量值的传递都需要通过主内存。

（2）主内存与工作内存之间的交互操作

lock（锁定）/unlock（解锁）：作用于主内存变量

read（读取）：作用于主内存变量，将变量值从主内存传输到工作内存

load（载入）：作用于工作内存变量，将read得到的值存入工作内存中的变量副本中

use（使用）：作用于工作内存变量，将变量值传递给执行引擎

assign（赋值）：作用于工作内存变量，将执行引擎返回的值赋给工作内存变量

store（存储）：作用于工作内存变量，将工作内存变量值传送到主内存

write（写入）：将store传输到主内存的值存入主内存中的变量

Java内存模型要求read->load，store->write顺序执行，但没有保证是连续执行的。同时要求线程不得丢弃它最近的assign操作，也即变量在工作内存中改变了之后必须同步回主内存。但是由于并不保证store和write的连续执行，所以并不能保证变量能立刻同步回主内存。

（3）volatile变量的特殊规则

volatile关键字是虚拟机提供的最轻量级的同步机制。

volatile变量具有两种特性：

**可见性**：volatile变量保证对所有线程的可见性。当某一线程修改了变量值，其他线程可以立即看到更新的值。普通变量的值在线程间传递需要通过主内存。而volatile变量依然需要从主内存中获得拷贝值，但是由于它的特性使其看起来好像直接在主内存中进行读写访问。

需要明确的是，volatile变量在各个线程的工作内存中也会有不一致的情况，但是由于每次使用之前都要先从主内存刷新以获取最新的值，执行引擎看不到刷新前不一致的情况，所以说volatile变量在各个线程的工作内存中不存在不一致的情况。

volatile变量保证了变量更新的实时可见性，但是基于volatile变量的运算并非原子操作，所以volatile变量的运算在并发下并不是安全的。

**有序性**：禁止指令重排序。volatile变量的assign赋值操作后面会生成一个lock内存屏障，防止重排序之后后面的指令重排序到内存屏障之前的位置。并且在内存屏障之前的指令会使得工作内存中的变量写入主内存中，并无效化其他工作内存中的变量，所以使得其他线程必须从主内存中重新读取变量值，让volatile变量的修改对其他线程立即可见。

Java内存模型中对volatile变量定义的特殊规则：

线程对volatile变量的use操作之前必须先进行read和load操作，同时也只有进行use操作时才能够对变量执行load操作。也即线程对volatile变量的read、load、use操作必须关联，并连续出现。在工作内存中，每次使用volatile变量之前都必须先从主内存刷新最新的值，保证能看见其他线程对volatile变量所作的修改。

线程在对volatile变量执行assign操作之后必须是store操作，同时也只有在进行assign操作之后才可以执行store动作。也即线程对volatile变量的assign、store、write操作相互关联，必须连续出现。在工作内存中，每次修改volatile变量的值之后都必须立刻同步回主内存，保证其他线程可以看到自己对变量的修改。

**7. 面向过程和面向对象**

面向过程是在面向对象出现之后提出的，对面向对象之前的编程方式的概括，或者称为结构化编程。在面向对象之前，程序基本都可以使用三种基本流程来实现，即顺序、选择和循环，程序设计的重点在于编写实现不同目的的函数/过程，并将数据访问局部化，这称为面向过程。我觉得这可以是封装思想的雏形。面向过程开发是一种利用计算机，并且从计算机的思维来解决问题的一种抽象，它将程序分为数据和处理数据的操作两部分，也即核心在于数据结构和算法的开发和优化。而在结构化编程的过程中，将数据结构和相应的操作绑定到一起就形成了抽象数据类型，这可以说是面向对象思想的起源。面向对象思想会将数据以及数据对应的操作或者行为逻辑封装到一起，作为一个整体。面向对象的重点在于对象的设计上，也就是类的设计上，并且考虑各个对象之间的关系。它是一种更符合现实世界思维方式的一种抽象。

面向过程是流程化、具体化的，分析解决问题需要的数据和步骤，并编写函数来一步步解决问题。而面向对象是模型化、抽象化的，它会把问题分解抽象成不同的对象，并把重点放在了对象的功能设计上，关注抽象出来的对象应当具有什么功能。对象设计好之后，通过不同对象模块之间的组合关联，来共同解决问题。面向对象思想可以保证更好的可维护性、可重用性和可扩展性。当需求发生变化，面向过程的设计中需要改动整体的代码, 同时，面向过程也不适合解决复杂的问题。而在面向对象中，只需要针对特定的对象做出修改，其他模块无需改动，并且修改对于其他模块是透明的，不会影响到其他模块的使用。在处理复杂的问题时，有些对象依赖于其他对象的功能，那么就可以进行聚合；有些对象具有的属性和功能在另一个对象中也有，但是更加具体，功能也更加丰富，那么就可以使用继承，使得对象的类型继承自另一个类型，使其可以有不同的类型，可以被当成不同的类型的对象；某些对象的使用方法是一致的，那么可以通过接口对他们进行一致的调用，而其具体的实现不可访问（封装）也无需关心，因为虽然使用的方法一致，但是不同的实现却会表现出不同的行为，这就是多态。面向对象的三个核心要素：封装、继承、多态。

**8. StringBuilder和StringBuffer**

8.1 相同点和差异

StringBuilder和StringBuffer都继承自AbstractStringBuilder，实现方法都是重用父类的方法除了toString方法。跟String一样，内部都使用字符数组来存放字符串。

两者区别就在于，StringBuffer每个方法都添加了synchronized关键字，并且有一个名为toStringCache的char[]来缓存最近一次的toString()方法返回的值，一旦StringBuffer类发生改变，toStringCache就置为null。当调用toString()时toStringCache为null，toStringCache会获得此时内部数组value的一个拷贝，同时用toStringCache构造一个String返回，而且此时新构造的String跟toStringCache共享一个字符数组。而StringBuilder的toString()方法重新创建一个String，而且这个String的内部字符数组是StringBuilder内部数组的拷贝，而非共享。

8.2 扩容策略

StringBuilder在每次插入新字符时都会调用ensureCapacityInternal(count + len)方法来保证内部数组的容量足够插入新字符。count为当前字符串长度，len为新插入的字符数量。当count+len大于当前数组的容量value.length时，会调用Arrays.copyOf(value, newCapacity(count+len))方法来新建数组。newCapacity(int minCapacity)方法返回数组的新容量，该容量是minCapacity和（旧容量 \* 2 + 2）中的较大值。

**9. Java中的单例模式**

Java中单例模式通过只提供private的构造方法来限制类的创建，再通过统一的getInstance方法作为获取单例的入口。

但是，只有这样其实还是不够的。因为在Java中，还有其它两种方法可以不通过new就可以获得类的实例，那就是clone和反射。

**9.1 clone方法**

一个类A要通过clone方法实现对象的克隆，需要两个条件，一是A重写Object中的clone方法，二是A实现Cloneable接口。

首先，clone方法在Object中声明为：

protected Object clone() throws CloneNotSupportedException

protected的可见性范围是包以及子类可见性，也即只能在Object类（clone方法是定义在Object内的）所在的包内以及Object的子类A内可访问（只能在Object的子类内调用父类的clone方法：super.clone()），所以如果在类A中不重写clone方法，那么该方法在其它类中是无法访问的。

其次，如果类A不实现Cloneable接口，那么在调用objA.clone()方法时会抛出异常：

Exception in thread "main" java.lang.CloneNotSupportedException: Singleton

为了避免单例类被通过clone方法创建出不止一个实例，单例类不要实现Cloneable接口。

**9.2 反射**

通过反射获得Singleton的private的构造方法，如果直接调用Constructor的newInstance()方法会抛出异常：

Exception in thread "main" java.lang.IllegalAccessException: Class Main can not access a member of class Singleton with modifiers "private"

但是通过Constructor的setAccessible(true)方法将private的构造方法设为可访问的，那么就可以调用newInstance方法创建出新的实例：

Constructor constructor = Singleton.class.getDeclaredConstructor();  
constructor.setAccessible(true);  
Singleton reflectionInstance = (Singleton) constructor.newInstance();  
System.*out*.println(reflectionInstance == Singleton.*getInstance*()); // 输出false

所以，通过反射的方法还是可以获得不止一个实例的。

**四、设计模式**

**1. 创建型模式**

* 创建型模式关注如何创建对象，核心思想是把对象创建和对象使用分离开，使得两者能相对独立地变换。

**（1）工厂方法**

定义创建对象的工厂接口，定义了抽象工厂方法来创建特定接口的产品对象，但是具体创建哪个实际类型的对象由工厂接口的实现类来决定。可以在工厂接口中定义静态方法获取工厂的实例，也就是工厂接口的实现类对象。通过调用工厂实例的具体工厂方法，返回实际的产品对象。

工厂方法的目的在于使得创建对象和使用对象分离，客户端总是引用工厂接口和产品接口，允许变换具体的子类，允许创建对象的代码独立变换，而不会影响到调用方。工厂方法能够隐藏创建对象的细节，允许对工厂方法内创建对象进行优化。

在大多数情况下，并不需要抽象工厂/工厂接口，而是直接使用静态方法返回产品对象，这叫做静态工厂方法。静态工厂方法对外封装了对象创建的细节，可以返回缓存的对象，从而提升速度并减少内存消耗。

**（2）抽象工厂**

工厂方法只定义创建对象的工厂接口，而需要创建的对象的产品接口是固定的，并不是由工厂接口定义的。

而抽象工厂不仅要定义工厂接口，也要定义产品接口。既定义了创建对象的抽象工厂，也定义所需要的抽象产品。可以根据需要实现不同的工厂以及工厂对应的产品对象。

抽象工厂模式让创建工厂以及工厂创建的对象产品与使用相互分离，可以切换到另一个工厂以及工厂对应的产品对象。抽象工厂模式实现的关键在于定义工厂接口与产品接口，但如何实现工厂与工厂所创建的产品对象可以根据需要由具体的实现类决定。可以在工厂接口中定义静态方法返回特定的工厂，客户端始终都只要引用工厂接口与产品接口，而无需知道具体的工厂与产品对象的实现细节。

总之：从工厂接口派生出不同的实际工厂；从产品接口派生出不同的产品对象；不同的产品对象由不同的实际工厂创建。客户端只引用工厂接口与产品接口。

**（3）单例模式**

* 保证一个类只有一个实例，并提供一个访问这个唯一实例的全局访问点。
* 懒汉式单例：在第一次调用的时候进行实例化，即延迟加载：。
* 饿汉式单例模式：在类加载的时候实例就被创建：以及静态内部类方法

线程安全的单例，也可以直接将INSTANCE设为public

public class Singleton {  
 private static final Singleton INSTANCE = new Singleton();

private Singleton() {

}

public static Singleton getInstance() {  
 return INSTANCE;

}

}

试图从JVM原理解释这个简单的单例为何是线程安全的：设多线程所在的类为D，由于多线程的代码调用了Singleton.getInstance()静态方法，而静态方法是通过解析调用的，也就是在D类加载的过程中的解析阶段就会把getInstance静态方法的符号引用解析为直接引用。而类方法的解析首先是类方法所在类的解析，首先加载静态方法所在的类即Singleton，其中包括了Singleton类的加载、验证、准备、解析和初始化过程。准备阶段为Singleton类的类变量进行分配内存空间并赋0值，即null。在初始化阶段调用Singleton的类构造器初始化类变量，也即为INSTANCE赋值new Singleton()。**总之，在多线程程序运行之前该静态方法已经解析成直接引用，而且在多线程启动时该类的类变量已经初始化。**总之一句话，实例化对象、调用静态方法或访问静态字段时必须先对类初始化。

基于此，也可以使用一个静态内部类来完成实例初始化：

private static class InnerSingleton {

private static final Singleton INSTANCE = new Singleton();

}

public static Singleton getInstance() {  
 return InnerSingleton.INSTANCE;

}

在访问内部类的静态字段时会先保证初始化内部类。跟是一样的。

只适合单线程下 延迟加载 的单例，在首次使用的时候才初始化唯一的实例：

public class Singleton {

private static final Singleton INSTANCE;

private Singleton() {  
 }

public static Singleton getInstance() {  
 if (INSTANCE == null) {  
 INSTANCE = new Singleton();

}

return INSTANCE;

}

}

试图从JVM原理解释，为何延迟加载并不是线程安全的。如上一条所述，线程代码所在的类在加载过程中就会解析Singleton.getInstance()方法，而解析方法会先解析方法所在的类，解析方法所在的类会加载这个类，在加载的过程中会对类的静态变量进行初始化。但是在延迟加载中静态变量并没有初始值，而是在getInstance方法中进行赋值。那么就会涉及到多线程的竞态问题，即多线程的冲突。

延迟加载的写法不是多线程安全的单例，在多线程竞态条件下可能会创建出多个实例，例如两个线程同时判断INSTANCE为null，则会同时创建出一个实例，则不再符合单例模式的要求。可以在判断为null时加上同步锁，可以使用synchronized或者使用锁，但是加锁会影响并发性能，因为只要第一次调用的时候需要同步，在第一次调用创建了单例之后就不需要同步检测INSTANCE是否为null了。因此使用Singleton模式时最好不要延迟加载，会使代码更简单。

public class Singleton {  
 private static final Singleton INSTANCE;

private Singleton() {

}

public synchronized static Singleton getInstance() {

if (INSTANCE == null) {  
 INSTANCE = new Singleton();

}

return INSTANCE;

}

}

理论可行但**实际不可行**的双重检查锁定（DCL）单例模式。使用双重检查来避免对除第一次调用之外的每次调用都进行同步：

public class Singleton {  
 private static final INSTANCE = null;

private Singleton() {

}

public static Singleton getInstance() {  
 if (INSTANCE == null) {  
 synchronized(Singleton.class) {  
 if (INSTANCE == null) {  
 INSTANCE = new Singleton();

}

}

}

return INSTANCE;

}

}

对象初始化操作并不是一个原子操作。初始化一个实例在java字节码中会有3个步骤：（1）申请内存空间（2）执行实例构造器方法（3）连接引用和实例，也就是将对象的引用赋给变量。这3个步骤后两个有可能会重排序，造成未初始化完全的对象发布而被其他线程获取，从而访问一个未初始化完全的对象。这个称为“引用逃逸”：其他线程通过逃逸的引用访问到初始化了一半的对象。

synchronized 同步块禁止重排序是通过内存屏障实现的。synchronized同步块通过monitorenter和monitorexit字节码指令实现。内存屏障只保证指令不会越过该屏障, 而synchronized块内部的指令仍然有可能发生重排序。

（根据Java内存模型，synchronized的语义不仅仅是在同一个信号上的互斥(mutex)，也包含线程和主存之间数据交互的同步，它确保在多处理器、多线程下对内存能有可预见的一致性视图。获取或释放锁会触发一次内存屏障（memory barrier）－－强迫线程本地内存和主存同步。当一个线程退出一个synchronized block时，触发一次写屏障（write barrier ）－－在释放锁前必须把所有在这个同步块里修改过的变量值刷新到主存；同样，进入一个synchronized block时，触发一次读屏障（read barrier）－－让本地内存失效，必须从主存中重新获取在这个同步块中将要引用的所有变量的值。）

虽然synchronized的锁对象是Singleton.class对象，当一个线程进入synchronized块时其他线程无法进入该类的静态方法或访问静态字段。但是可能在线程进入同步块时已经有其他线程进入getInstance方法并被挂起，在INSTANCE获得引用之后再次激活，此时判断INSTANCE非null，则还是有可能获得INSTANCE的引用。

volatile可以禁止指令重排序，从而避免这个问题。volatile保证volatile变量的赋值指令不会被指令重排序优化。volatile关键字是通过内存屏障来避免其他指令重排到屏障之前。volatile屏蔽指令重排序至少在JDK1.5以前是不能完全实现的，因而无法做到安全地使用DCL单例模式。但是现在貌似是可以安全使用的。但是仍然不推荐此方法，多核处理器下线程每次写volatile字段都会把工作内存及时刷新到主存，每次读都会从主存获取数据，因为要和主存交换数据，volatile的频繁读写会占用数据总线资源。

使用枚举类。Java的枚举类保证声明的类型只有特定个数的实例。枚举类可以有构造方法，但必须是private的，它只在构造枚举常量的时候被调用。同时可以包含一些方法和域：Singleton.INSTANCE.getName();

public enum Singleton {

INSTANCE; // 唯一实例

private String name = “Singleton”;

public String getName() {

return name;

}

public void setName(String name) {  
 this.name = name;

}

}

也可以定义一个构造器为唯一的实例INSTANCE的域赋值：

public enum Singleton {  
 INSTANCE(“Singleton”); // 唯一实例，调用构造器将name赋为”Singleton”

private String name;

private Singleton(String name) {  
 this.name = name;

}

public String getName() {  
 return name;

}

public void setName(String name) {  
 this.name = name;

}

}

(4)

**五、MySQL**

**1. 数据库事务**

（1）什么是事务

事务就是一组原子性的SQL查询。如果数据库引擎能够对数据库成功应用所有的语句，则执行整组查询。如果有任何原因导致无法全部执行，那么所有的语句都不会执行。事务内的语句，要么全部执行成功，要么全部执行失败。

事务处理：事务处理是一种机制，保护数据库的完整性。事务处理管理成批的MySQL操作，保证数据库不包含不完整的操作的结果。事务处理保证一组操作要么作为整体执行，要么完全不执行，而不会中途停止。如果整组操作没有错误发生，则将整组操作提交到数据库，否则进行回退，以恢复到事务开始之前的某个安全状态。

（2）事务的四大特性

事务处理必须具备四大特性：原子性（Atomicity）、一致性（Consistency）、隔离性（Isolation）和持久性（Durability），称为事务的ACID特性。

原子性

一个事务必须被视为一个不可分割的最小工作单元，事务中所有操作要么全部执行成功并提交，要么全部失败并回滚，而不能只执行其中的一部分操作。

一致性

数据库总是从一个一致性的状态转换到另一个一致性的状态。这里的一致指的是数据库整体的一致性，如果事务只执行一部分，那么就会使数据库处于不确定的且整体不一致的状态。事务开始之前和结束之后，数据库的完整性约束没有被破坏，即保证数据的完整性以及业务逻辑的一致性。比如说，不管银行转账事务是否成功执行，银行中的总金额必须保持不变。

隔离性

一个事务所做的修改在最终提交之前对其他事务不可见。隔离性保证各个事务的相互独立性。

持久性

一旦事务提交，则其所做的修改就会永久保存到数据库中。即使系统崩溃，修改的数据也不会丢失。

**2. 数据库隔离级别（这里有错误，还是以《MySQL技术内幕》为准，不要看）**

SQL标准定义了四种隔离级别，规定了一个事务中所做的修改对其他事务的可见性。

Read Uncommitted读未提交

在此隔离级别中，事务中的修改还没提交就对其他事务都是可见的。事务可以读取未提交的数据，这称为脏读。

脏读带来的问题：其他事务读取了还未提交的修改，如果事务进行了回滚，那么其他事务读取到的就是无效的脏数据。

Read Committed读已提交

Read Committed满足隔离性的定义，一个事务开始时只能看见已经提交的事务所作的修改，即一个事务在提交之前所做的任何修改对其他事务都是不可见的。

不可重复读：Read Committed也叫做不可重复读，指某个事务内前后两次对同一数据的查询中间，由于其他事务提交了修改，更改了数据，使得查询到的结果前后不一致。不可重复读针对的是同一数据的多次查询，由于其他事务的update操作而导致前后查询不一致。针对不可重复读可以使用行级锁，在读取之后才允许其他事务更新。

Repeatable Read可重复读

可重复读保证事务无法读取到未提交的修改，解决脏读的问题，同时也保证在同一事务内多次读取同样的记录的结果是一致的。

幻读：可重复读可以解决脏读和不可重复读的问题，但是无法解决幻读的问题。幻读指的是某个事务在读取某个范围内的记录（注意这里是一个范围内的数据，而不可重复读查询的是一个记录）时，由于其他事务在该范围内插入（注意这里是插入insert而不是不可重复读中的更新update）了新的记录，事务再次读取相同范围内的记录时出现了之前没有出现过的幻行。

如果只是锁定单个行，那么无法避免其他事务插入新的行，因此可重复读无法避免幻读。针对幻读可以使用行锁加间隙锁，锁定查询范围内的记录的间隙，避免其他事务的插入。

Serializable可串行化

可串行化是最高的隔离级别，它强制事务串行执行，避免了各个事务之间的干扰，因此避免了脏读、不可重复读以及幻读的问题。可串行化会对读取的每一行数据都加锁。

**3. 多版本并发控制**

考虑到并发性能，MySQL的大多数事务性存储引擎实现的都不是简单的行级锁，而是实现了多版本并发控制（MVCC）。MVCC是行级锁的一个变种，它在很多情况下避免加锁操作。这是不是也就是说在必要的时候会施加行级锁。MVCC可以实现非阻塞的读操作，也就是可以多事务同时读，而写操作也只会锁定必要的行。

MVCC的实现是通过保存数据在某个时间点的快照来实现的。使用 MVCC 读取的是快照中的数据，这样可以减少加锁所带来的开销。InnoDB的MVCC是通过在每行记录后保存两个隐藏的列：行的创建版本号以及删除版本号。创建版本号和删除版本号都是系统版本号。系统版本号每开始一个新的事务就会自增，事务开始时的系统版本号作为事务的版本号。

在Repeatable Read隔离级别下，select/update/insert/delete会如何操作：

Select：InnoDB只查找创建版本号小于等于事务版本号，删除版本号未定义或者大于当前事务版本号的行，作为一个快照。前者保证此行要么是在事务开始之前创建的，要么是本事务自己创建或修改过的行。而后者确保此行在事务开始之前没有被删除。

Insert：InnoDB为新插入的行保存事务版本号作为创建版本号

Delete：为删除的每一行保存当前的事务版本号作为删除版本号

Update：将更新的数据作为新行插入，保存事务版本号作为创建版本号，同时保存事务版本号到原来的行作为删除版本号。

保存版本号可以使大多数读操作都可以不用加锁，读数据操作简单，性能好。但是每行记录都需要额外的存储空间，需要额外的行版本号检查。

MVCC可以在读已提交和可重复读两种隔离级别下工作，而读未提交隔离级别总是读取最新的数据行，可串行化对所有读取的行进行加锁，所以这两个级别与MVCC不兼容。

InnoDB中分为快照读和当前读。快照读也即使用MVCC，读取的是数据的快照中的数据，是数据的历史版本，避免加锁的操作。对于简单的select操作属于快照读，不加锁。而执行特殊的读操作：insert/update/delete操作属于当前读，需要读取数据最新的版本，在当前读时需要加排他锁，其他事务都不能对加锁的数据进行修改。在对数据进行插入/更新/删除操作时隐式地使用到查询操作，则使用当前读的方式，通过施加行锁和间隙锁来避免出现幻读。间隙锁（next-key locking）不仅仅锁定查询涉及的行，还会对索引中的间隙进行锁定，以防止幻影行的插入。要注意的是，InnoDB行锁是通过给索引上的索引项加锁来实现的，如果不能使用索引查找和锁定行的话，MySQL会做全表扫描并锁住所有行（使用表锁）。

**4. 数据库索引**

索引在MySQL中也称为键key，是存储引擎用于快速找到记录的一种数据结构。索引可以包含一个或多个列，但只能高效使用索引的最左前缀列。在MySQL中，索引是在存储引擎中而不是服务器层实现的。

如果查询的列不是独立的，则MySQL就不会使用索引。独立的列是指，索引列不能是表达式的一部分，也不能是函数的参数。

（1）b+树索引

InnoDB存储引擎使用b+树索引。MyISAM使用前缀压缩的b+索引。

b+树索引的叶节点保存的是指向被索引的记录的指针。b+树对索引列是顺序组织存储的，很适合用来查找范围数据，用于查询中的order by/group by操作

b+树索引多个列时，对多个值进行排序的依据是定义索引时列的顺序。b+树索引适合用于：

全键值匹配：和索引中的所有列进行匹配

匹配最左前缀：只匹配索引的第一列

匹配列前缀：只匹配索引的第一列的前缀

匹配范围值：匹配索引第一列的某个范围

精确匹配某一列并部分匹配另一列：全部匹配索引第一列并部分匹配第二列

只访问索引：这是一种覆盖索引

b+树索引必须从索引的最左列开始查找，不能跳过索引中的列，否则无法使用索引。并且如果查询中有某个列的范围查询，则其右边的所有列查询都无法使用索引优化查询。

（2）哈希索引

哈希索引基于哈希表，并使用链表解决哈希冲突。只有精确匹配索引的所有列的查询才可以使用。对每一行数据，存储引擎根据所有的索引列计算哈希码，并将其存储在索引中，同时在哈希表中保存指向每个数据行的指针。

MySQL中，Memory存储引擎默认使用哈希索引，同时也支持b+树索引。

哈希索引只需存储哈希值，因此结构紧凑，查找速度快。但是也有如下缺点：

哈希索引只保存哈希值和指针，并没有存储行的字段值，所以不可避免要读取行来获得数据，哈希索引无法使用覆盖索引。

哈希索引只支持等值比较查询，无法用于范围查询，也无法用于排序。

哈希索引不支持部分索引列匹配查找，也不支持索引列的部分匹配查找。哈希索引始终使用索引列的全部列来计算哈希值。只使用部分索引列或者索引列的部分值无法计算哈希值。

当哈希冲突很多时会严重影响查询性能和索引维护成本

（3）聚簇/聚集索引

聚簇索引并不是一种单独的索引类型，而是一种数据存储方式。当表使用聚簇索引时，它的数据行重新组织并存放在了索引的叶子节点上。“聚簇”表示数据行与相邻的关键字紧凑地存储在一起。InnoDB存储引擎实际上在b+树索引的叶节点上存放的是数据行的全部数据而不是数据记录的指针，非叶节点依然包含的是索引值。聚簇索引也可以称为索引组织表，它将整个表的数据行按照指定顺序重新组织排列，变成索引表。因此，聚簇索引改变了数据行实际的物理地址，而非聚集索引只记录逻辑顺序，并不改变数据行的物理地址。在InnoDB中，聚簇索引就是表，它不需要额外的空间来存储数据行。

一个表只能有一个聚簇索引。InnoDB默认使用主键聚集数据。聚簇索引聚集数据有如下优点：

可以把相关数据保存聚集在一起，减少磁盘I/O次数

聚簇索引将索引和数据保存到一个b树中，数据访问更快

但同时，对于聚簇索引表的其他非聚簇索引（二级索引），需要两次索引查找。二级索引叶节点中保存的是数据行的主键值而不是行的物理地址（行指针），这意味着通过二级索引查找行，存储引擎需要先找到二级索引叶节点并获得对于行的主键值，再根据主键值去聚簇索引中查找相应的行。这就是两次b树查找。这样的方式使得当由于删除或插入操作导致页分裂或合并时，无需更新二级索引中的叶节点的行指针，减少二级索引的维护工作。

（4）覆盖索引

如果一个索引包含/覆盖所有需要查询的字段的值，那么就称之为覆盖索引。可以使用覆盖索引来直接获取所需的列值，而不需要再回表读取数据行。扫描的索引节点远小于数据行大小，如果只需要读取索引，就会极大减少数据访问量。

InnoDB的聚簇索引本身就是一种覆盖索引。如果二级索引能够覆盖查询，那么就可以避免对主键索引的二次查询。

覆盖索引需要存储索引列的值，而哈希索引、全文索引都不能存储索引列的值，因此只能使用b树索引来覆盖索引。

发起一个被索引覆盖的查询，也叫索引覆盖查询，就可以使用此索引作为覆盖索引。

**5. InnoDB和MyISAM存储引擎的区别**

* InnoDB支持事务，可以进行Commit和Rollback；
* MyISAM 只支持表级锁，而 InnoDB 还支持行级锁，提高了并发操作的性能；
* InnoDB 支持外键；
* MyISAM 崩溃后发生损坏的概率比 InnoDB 高很多，而且恢复的速度也更慢；
* MyISAM 支持压缩表和空间数据索引，InnoDB需要更多的内存和存储；
* InnoDB 支持在线热备份。

**6. 存储过程及其优点**

（1）什么是存储过程

存储过程简单来说就是为以后使用而实现保存的一条或多条MySQL语句的集合。存储过程是事先经过编译并存储在数据库中的一段完成特定功能的SQL语句的集合。想要实现相应的功能时，只需要调用这个存储过程（类似于函数，输入具有输出参数）。存储过程就是数据库SQL语句的代码封装和重用。

（2）存储过程的优点

将复杂的语句集合封装起来，可**简化**复杂的操作，提高代码的可重用性；

所有用户都使用同一存储过程，保证数据操作的一致性和完整性，防止出错；

**简化**代码的修改和维护，如果表定义和业务逻辑发生变动，只需要修改对应的存储过程即可，且修改对用户是透明的，用户无需知道其中的细节以及变化；

通过封装操作和数据，限制用户对数据的直接操作，控制访问权限，增加**安全性**；

**提高性能**，使用存储过程比使用单独的SQL语句要快。存储过程预先编译保存在数据库中，而不需要每次运行时编译，提高了执行**效率。**

**7. 视图**

（1）什么是视图

视图是虚拟的表，不存放任何数据，它包装了一个SQL查询语句，可以封装复杂的SQL查询。在使用视图时，它返回的是从其他表中查询得到的数据。MySQL可以像对待表一样使用视图。

（2）视图的实现方式

临时表算法（Temptable）

将视图定义中select语句的结果放到临时表中。当访问视图时直接访问对应的临时表：

CREATE VIEW view\_name AS SELECT \* FROM table\_name WHERE col = col\_value WITH CHECK OPTION;

CHECK OPTION：表示任何通过视图更新的行，都必须符合视图本身的WHERE条件定义，不能更新视图定义列以外的列。也就是更新的行都必须满足WHERE子句的条件，也不能插入WHERE子句范围外的新数据。

可以为一个基于简单查询的视图指定使用临时表算法：

CREATE ALGORITHM=TEMPTABLE VIEW v1 AS SELECT \* FROM table\_name;

合并算法（Merge）

实现视图更好的方式是重写含有视图的查询，将视图的定义SQL直接包含进查询的SQL中。

（3）使用视图的优点和限制

优点：

使用视图可以封装复杂的查询，重用SQL语句，提高代码重用性

简化复杂的SQL操作，不必知道查询的细节

保护数据，提供表特定部分的访问权限，而不是整张表的权限

限制：

视图不能创建索引，也不能有关联的触发器

MySQL不支持物化视图：物化视图指的是将视图的结果数据存放在一个可以查看的表中，并定期从原始表中刷新数据到这个表中。

可以更新视图，更新一个视图实际上更新的是视图定义中涉及的相关表。但是更新视图是有限制的，如果视图定义中有group by/having/distinct/union/join/聚集函数/子查询等等，则无法更新视图。所以一般应该将视图用于查询select，而不是update/insert/delete。另外，使用临时表算法实现的视图也都无法被更新。

如果视图中包含group by/distinct/union/聚集函数/子查询等，只要无法在原表记录和视图记录中建立一一映射的情况，MySQL都将使用临时表算法来实现视图。

MySQL并不会保存视图定义的原始SQL语句。