# JVM

### 1. JVM运行时的数据区域：

（1）程序计数器--线程私有

（2）Java虚拟机栈--线程私有

每个Java方法执行时都会创建一个栈帧用于存储局部变量表、操作数栈、动态链接、方法出口等信息。每一个Java方法从调用到执行完成的过程就对应着一个栈帧在虚拟机栈中从入栈到出栈的过程。

（3）本地方法栈（也叫native栈，线程私有）

1. 虚拟机栈为虚拟机执行Java方法服务，

2. 本地方法栈则为虚拟机用到的Native方法服务。

（4）Java堆（线程共享区域-虚拟机启动时创建）--也是GC管理的地方

（5）方法区（线程共享区-虚拟机启动时创建）

用于存储以被虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码等数据。

## 对象创建与定位访问：

主要通过new关键字来实现：首先判断有没有类加载，没有加载过就先加载类--》为新生对象分配内存--》内存初始化--》对象的必要设置

对象内存布局：对象头、实例数据、对齐填充

访问定位：通过栈上的 reference 数据来操作堆上面具体对象。

Hotspot使用的是直接指针。reference直接指向Java堆上的指针。

## GC

### GC--判断对象是否可回收

（1）引用技术算法：引用一次加1，失效就减1. 但是不能解决互相引用的情况。

（2）可达性分析算法：通过一系列称为“GC Roots”的对象作为起始点，从这些节点向下搜索，搜索所走过的路径称为引用链，当一个对象到GC Roots没有任何引用链（即GC Roots到对象不可达）时，则证明此对象是不可用的。

### 3.2 GC--对象回收算法

新生代：主要用来存放新生的对象。

老年代：主要用来存放应用程序中生命周期长的内存区域。

持久代：主要存放的是Java类的类信息，与垃圾收集要收集的Java对象关系不大。

（1）标记-清除算法：首先标记出所有需要回收的对象，标记完成后统一回收所有被标记的对象。

（2）复制算法----新生代才采用的算法：它将可用的内存分为两块，每次只用其中一块，当这一块内存用完了，就将还存活着的对象复制到另外一块上面，然后再把已经使用过的内存空间一次性清理掉。

（3）标记整理算法----老年代常采用算法：过程与标记-清除算法一样，不过不是直接对可回收对象进行清理，而是让所有存活对象都向一端移动，然后直接清理掉边界以外的内存。

（4）分代收集算法：现代商用虚拟机基本都采用分代收集算法来进行垃圾回收，在Java中一般把堆分为新生代和老年代，然后根据各块的特点采用最适当的收集算法。其实也就是上面几种算法的结合使用。

### 3.3 GC 算法中存在的问题：

#### （1）Stop the world

在对象存活算法的可达性分析算法中从GC Roots节点找引用链这个操作为例。其中可达性分析对执行时间的敏感点的一个体现就是GC停顿上面，因为这项分析工作必须在一个能确保一致性的快照中进行–这里的一致性是指在整个分析期间整个执行系统看起来就像被冻结在某个时间点上，不可以出现分析过程中对象引用关系还在不断变化的情况，该点不满足就无法得到保证。这一点是导致GC进行时必须停顿所有[**Java**](http://lib.csdn.net/base/javase)执行线程（sun称为 “Stop the world”）的其中一个重要原因。

#### （2）安全点

在HotSpot虚拟机中，借助于OopMap的协助下，可以快速且准确的完成GCRoots的枚举。但是不可能为每条指令都生成OopMap，这样空间成本就非常高。所以HotSpot中只是在“特定的位置”记录了这些信息，这些位置称为安全点，即程序执行时并非在所有地方都停顿下来开始GC，只有到达安全点时才能暂停。

safepoint的选择不能太少，不然GC等待时间会过长，也不能过于频繁以导致过大增大运行时的负荷。

### 3.4 Hotspot中的垃圾收集器

Hotspot中常见的垃圾收集器如下图所示：



#### （1）Serial收集器—新生代

这个收集器是一个采用复制算法的单线程的收集器，单线程一方面意味着它只会使用一个CPU或一条线程去完成垃圾收集工作，另一方面也意味着它进行垃圾收集时必须暂停其他线程的所有工作，直到它收集结束为止。

#### （2）ParNew收集器—新生代

ParNew收集器其实就是Serial收集器的多线程版本，除了使用多条线程进行垃圾收集之外，其余行为都与Serial收集器完全一样,包括使用的也是复制算法。

#### （3）Parallel收集器—新生代

Parallel收集器也是一个新生代收集器，也是用复制算法的收集器，也是并行的多线程收集器,但是它的特点是它的关注点和其他收集器不同。介绍这个收集器主要还是介绍吞吐量的概念。CMS等收集器的关注点是尽可能缩短垃圾收集时用户线程的停顿时间，而Parallel收集器的目标则是打到一个可控制的吞吐量。所谓吞吐量的意思就是CPU用于运行用户代码时间与CPU总消耗时间的比值，即吞吐量=运行用户代码时间/（运行用户代码时间+垃圾收集时间），虚拟机总运行100分钟，垃圾收集1分钟，那吞吐量就是99%。另外，Parallel收集器是虚拟机运行在Server模式下的默认垃圾收集器。

#### （4）Serial Old收集器—老年代

Serial收集器的老年代版本，同样是一个单线程收集器，使用“标记-整理算法”，

#### （5）Parallel Old收集器—老年代

Parallel收集器的老年代版本，使用多线程和“标记-整理”算法。

#### （6）CMS收集器—老年代

CMS(Concurrent Mark Sweep)收集器是一种以获取最短回收停顿时间为目标的老年代收集器。

### 3.5 常识概念：

#### （1）内存溢出和内存泄露的区别

内存溢出:指的是程序在申请内存的时候，没有足够大的空间可以分配了。

内存泄露:指的是程序在申请内存之后，没有办法释放掉已经申请到内存，它始终占用着内存，即被分配的对象可达但无用。内存泄露一般都是因为内存中有一块很大的对象，但是无法释放。

#### （2）Minor GC和Full GC的区别

新生代GC（Minor GC）:指发生在新生代的垃圾收集动作，因为大多数Java对象存活率都不高，所以Minor GC非常频繁，一般回收速度也比较快

老年代GC(Full GC）:指发生在老年代的垃圾收集动作，出现了Full GC，经常会伴随至少一次的Minor GC（但并不是绝对的）。Full GC的速度一般要比Minor GC慢上10倍以上

## 类加载机制

### 4.1 JVM有哪些类加载器：

1.启动类加载器--加载位置 ：加载$JAVA\_HOME中jre/lib/rt.jar里所有的class或则被-Xbootclasspath参数指定的路径中

2.扩展类加载器--加载位置 ：加载JAVA\_HOME/lib/ext目录中的，或者被java.ext.dirs系统变量指定所指定的路径中所有类库，

应用程序类加载器--加载位置 ：classpath中指定的jar包及目录中class；

4.自定义加载器—自定义的类加载器，手动加载，只需要继承自ClassLoader然后实现loadClass()方法。

### 4.2 双亲委派模型



双亲委派模型的工作过程是：   
1. 如果一个类加载器收到了类加载的请求，它首先不会自己去尝试加载这个类，而是把这个请求委派给父类加载器去完成，每一个层次的类加载器都是如此。   
2. 所有的加载请求最终都会传送到顶层的启动类加载器中，只有当父加载器反馈自己无法完成这这个加载请求（它的搜索范围中没有找到所需的类）时，子加载器才会尝试自己去加载。

## 内存模型

Java内存模型定义了一种多线程访问Java内存的规范。Java内存模型要完整讲不是这里几句话能说清楚的，我简单总结一下Java内存模型的几部分内容：

（1）Java内存模型将内存分为了主内存和工作内存。类的状态，也就是类之间共享的变量，是存储在主内存中的，每次Java线程用到这些主内存中的变量的时候，会读一次主内存中的变量，并让这些内存在自己的工作内存中有一份拷贝，运行自己线程代码的时候，用到这些变量，操作的都是自己工作内存中的那一份。在线程代码执行完毕之后，会将最新的值更新到主内存中去

（2）定义了几个原子操作，用于操作主内存和工作内存中的变量的交互

（3）定义了volatile变量的使用规则

（4）原子性、可见性、有序性

（5）happens-before，即先行发生原则，定义了操作A必然先行发生于操作B的一些规则，比如在同一个线程内控制流前面的代码一定先行发生于控制流后面的代码、一个释放锁unlock的动作一定先行发生于后面对于同一个锁进行锁定lock的动作等等，只要符合这些规则，则不需要额外做同步措施，如果某段代码不符合所有的happens-before规则，则这段代码一定是线程非安全的。

# 分布式

### 1.大型网站性能逐步优化过程：

（1）应用和数据的分离:也就是专门的应用服务器、数据库服务器、文件服务器

（2）使用缓存，专门的缓存服务器(集群) 改善数据访问性能。注意这里的数据访问二八定律。

（3）应用服务器集群，改善并发处理请求能力。

（4）数据库的读写分离改善数据库访问压力(主从热备功能)

（5）CDN加速和方向代理提高访问速度。（都是基于缓存

1. CDN部署在网络提供商的机房，使用户请求网站服务时可以从距离自己最近的网络提供商机房获取数据）

2. 反向代理则部署在网站的机房中心，当用户请求到达机房中心后，首先访问的服务器是反向代理服务器，如果反向代理服务器中缓存着用户请求的资源，就将其直接返回给用户，就不用经过应用服务器。

不管是CDN还是反向代理的目的都是为了尽早的把数据返回给用户，一方面加快用户访问速度，也能减轻后端服务器的负载压力。

（6）分布式数据库服务器

（7）业务拆分

（8）分布式服务，使用消息队列。

记住：首先是使用业务手段解决业务问题，再用技术手段解决。

### 2.高性能：

（1）前端优化：

1. 使用浏览器缓存、静态资源使用独立的域名(减少cookie传输)、JS压缩

2. CDN加速：使用CDN缓存的一般都是一些静态资源，比如：图片，CSS，Script脚本，静态网页等等。这些文件访问频率极高，将其缓存在CDN可极大的改善网页的打开速度。也就是实现所谓的动静（资源）分离

3. 反向代理：配置缓存功能加快web请求，当用户第一次请求静态内容时，静态内容就被缓存在反向代理服务器上。这样当其他用户访问时就可以直接从反向代理服务器直接返回，加快web请求的响应速度，减轻后端服务器压力。

（2）应用服务器优化：缓存、集群、异步。

1. 缓存：分布式缓存存在一致性Hash问题，

2. 集群：负载均衡：负载均衡算法

3. 异步：消息队列（存在延迟）， 优点：提高可用性、加快访问速度、消除峰值

### 3.高可用：服务器硬件故障时服务依旧可用

session集群：session定位？

高可用数据：

### 4. CAP理论（一致性、可用性、分区容错性）

A是必须要保证的，所以对于CP基本是放弃的； 对于分布式系统一般是保证AP放弃强一致性C（对于分布式系统来说P是基本要求）

### 5. BASE理论：

Basically Available（基本可用）、

Soft state（软状态：允许系统中的数据存在中间状态）

Eventually consistent（数据最终一致性）

### 6. 伸缩性：应用服务器集群

（1）http重定向负载均衡：利用Http的重定向功能，客户端重定向到实际应用服务器

（2）DNS域名解析负载均衡：DNS负载均衡服务器解析请求 并返回实际的IP地址，然后再次访问实际应用服务器；

常规用法是：DNS域名解析作为第一级负载均衡，获得的一组服务器是同样提供负载均衡的内网服务器，然后内网的负载均衡服务器再在内网进行第二级负载均衡，分发到实际Web应用服务器。

（3）反向代理负载均衡：反向代理服务器除了缓存资源外还可以提供负载均衡。

（4）IP负载均衡：在网络层修改请求的目标地址进行负载均衡。

（5）数据链路层负载均衡：在通信协议的数据链路层修改Mac地址进行负载均衡。

### 7. 路由算法：

（1）余数Hash算法：扩容时存在大量的命中失效

（2）一致性Hash算法：使用Hash环均匀分布； 为了解决一致性hash 算法带来的负载不均衡问题，通过加入加一层虚拟层来解决。

### 8. 负载均衡算法：

(1) 轮询法（Round Robin）

(2)随机法（Random）

(3)源地址Hash法（Hash）

(4)加权轮询法（Weight Round Robin）

(5)加权随机法（Weight Random）

(6)最小连接数法（Least Connections）

### 9. 常用的hash算法有哪些？

1）余数Hash算法

2）一致性Hash算法

### 10. 解决hash冲突的方法有哪些：

1）开放定址法

2）再哈希法

3）链地址法

4）建立一个公共的溢出区。

### 11. 什么是一致性hash

一致性hash是一种分布式hash算法。分布式集群中，对机器的添加删除，或者机器故障后自动脱离集群这些操作是分布式集群管理最基本的功能。如果采用常用的余数hash算法会导致大部分的原有数据失效，这是非常致命的。一致性hash算法，利用一个hash环的数据结构来实现，将集群中的不同机器通过计算其hash值均匀的分布在hash环上。当一个任意的key计算出其hash值之后，只需要在环上寻找顺时针下最近的结点。如果对于集群中任意一个机器的删除或则增加，只会使很小的一部分的数据失效，大部分数据还是有效的。当机器的增加或则删除导致的负载不均衡，我们可以通过增加一个虚拟层来解决。



### 12. 怎么理解分布式锁？（秒杀为例）

分布式锁：就是控制分布式系统中同步访问共享资源的方式。（比如淘宝秒杀）

在分布式场景中，数据的一致性是一个很重要的内容，从分布式的CAP理论我们可以知道，我们一般都是牺牲强一致性提高可用性，从而实现最终一致性。保证最终一致性就需要用到分布式锁了。对于Java来说提供的锁机制都是针对同一主机而言的，但是项目分布在一个集群上时，这些锁机制就失效了。

分布式锁其实就是要保证在分布式环境下，同一个方法在同一时间只能被一台机器上的一个线程执行。该锁应该是：可重入、阻塞、高可用、高性能。

对于分布式锁的实现一般有如下几种方式

1. 基于数据库：

直接创建一个锁表，通过操作数据表中的数据实现，当我们要锁住某个方法或资源时，我们就在该表中增加一条记录，想要释放锁的时候就删除这条记录。注意这里的记录必须以该资源作为唯一标识(主键)

该方法有一些缺点：依赖数据库的高可用高性能，一旦解锁失败会导致锁永远不被释放。

2）基于缓存：比如redis

redis是单进程单线程模式，采用队列模式将并发访问编程串行。下面利用Redis提供的SETNX、EXPIRE、DEL命令很方便实现分布式锁。EXPIRE设置失效时间。

加锁：SETNX profuct.lock <时间戳> 这里的value并不重要，只要key存在就是加锁了

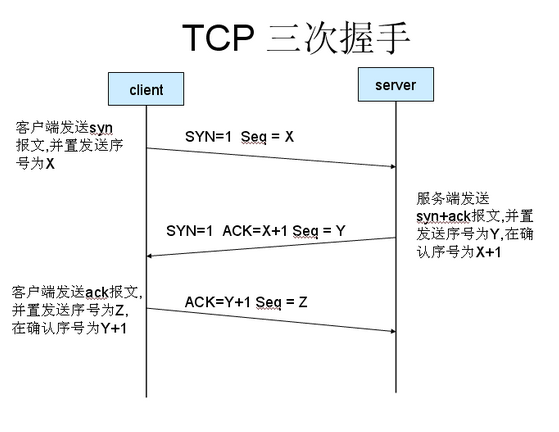
释放锁：DEL foo.lock 直接删除这个key，就表示释放锁。

# 计算机网络的基础知识

### TCP的三次握手、四次挥手

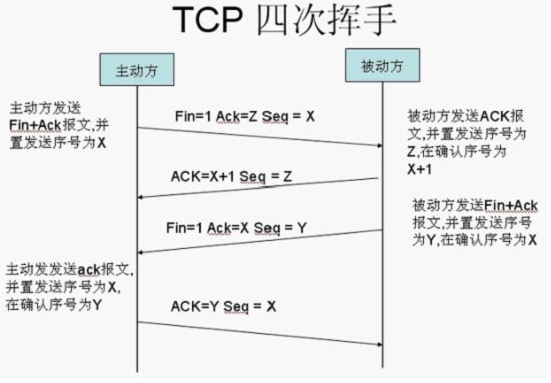
连接的三次握手

1. 客户端发送SYN信号，表示告诉服务器，我要建立连接
2. 服务器返回ACK=1和SYN=1给客户端，表示我已经接收到了客户端的请求，并接收了请求。
3. 客户端收到来自服务端响应后，知道服务端已经接受连接请求，再次发送ACK给服务器，确认服务端的SYN，其实也就是一个确认的再确认。



终止的四次挥手：客户端和服务端均可主动发起挥手操作终止连接

1. 主动方A发送一个FIN(终止信号)给被动方B，表示要终结主动方A到被动方B的连接；
2. 被动方B收到了FIN信号，返回ACK信号给主动方，表示从主动方到被动方的连接关闭了，也就是主动方不能再发送数据给被动方。
3. 被动方B在发送完数据后，给主动方A发送一个FIN信号，请求要终结被动方B到主动方A的连接；
4. 主动方A收到了FIN信号，返回ACK信号给被动方B，表示从被动方B到主动方A的连接关闭了，也就是被动方不能再发送数据给主动方。



### 为什么TCP建立连接是三次，而断开连接要四次？

连接是三次的原因：

为了防止失效的连接请求报文段突然由传送到主机B，因而产生错误。

断开时四次的原因：

这是由于TCP的半关闭造成的，由于TCP 是全双工的，表示可以同时在两个方向上即可以接收数据也可以发送数据。所以我们关闭连接必须在每个方向上面单独关闭，这个单方向的关闭就叫半关闭。所以挥手时其实就是2次断开连接，所以是四次。

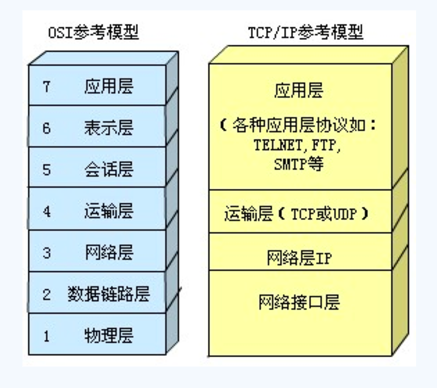
### TCP和UDP的区别

TCP和UDP都是传输层的协议。

TCP：面向连接、传输可靠(保证数据正确性,保证数据顺序)、用于一次传输大量数据(流模式)、速度慢，建立连接需要开销较多(时间，系统资源)。比如FTP、SMTP之类

UDP：面向非连接、传输不可靠、用于一次传输少量数据(数据包模式)、速度快。比如QQ

### TCP/IP的分层模型：



### http中GET和POST的区别

1. get是从服务器上获取数据，post是向服务器传送数据。

2. get是把参数数据队列加到提交表单的ACTION属性所指的URL中，值和表单内各个字段一一对应，在URL中可以看到。post是通过HTTP post机制，将表单内各个字段与其内容放置在HTML HEADER内一起传送到ACTION属性所指的URL地址，用户看不到这个过程。

3. get传送的数据量较小，不能大于2KB。post传送的数据量较大，一般被默认为不受限制。

### http和https的区别

1）https协议需要申请证书，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。

### Cookie和session的区别

Cookie和session都是保存回话的两种技术：

1）cookie保存在客户端，session保存在服务器端；

2）cookie目的可以跟踪会话，也可以保存用户喜好或者保存用户名密码，session用来跟踪会话。

3）cookie不是很安全，别人可以分析存放在本地的COOKIE并进行COOKIE欺骗，考虑到安全应当使用session。

4）session会在一定时间内保存在服务器上。当访问增多，会比较占用你服务器的性能，考虑到减轻服务器性能方面，应当使用COOKIE。

5）这个地方可以扩展到分布式session服务器。

### 8. http状态码的3xx, 4xx, 5xx分别指什么？

3xx：重定向

4xx：客户端/请求错误

5xx：服务器内部错误

# Java并发与多线程

### 1. 多线程有什么用？

1）发挥多核CPU的优势，充分“压榨”CPU的性能

2）防止阻塞。

3）便于建模，比如一个大的任务A，单线程完成要考虑很多，如果分成很多小任务，多线程来完成就简单很多了。

### 2.创建线程的方式

一般有三种：   
1）继承Thread类   
2）实现Runnable接口   
3）实现Callable接口   
一般来说我都是通过实现Runnable或则Callable接口创建线程，也觉得这种方式好一些，毕竟Java一直提倡面向接口编程。

### start()方法和run()方法的区别

start()是Thread里面的方法，在调用start()方法时会立刻启动线程，然后会根据CPU的调度而自动的调用run()方法，但是调用start()方法会立即返回，不会阻塞调用的主线程，从而真正实现多线程。

1）start() 方法来启动线程，真正实现了多线程运行。这时主线程无需等待run方法体代码执行完毕，就可以直接继续执行下面的代码；通过调用Thread类的start()方法来启动一个线程， 这时此线程是处于就绪状态， 并没有运行。 但是通过Thread类调用方法run()来完成其运行操作的， 这里方法run()称为线程体，它包含了要执行的这个线程的内容， Run方法运行结束， 此线程终止。然后CPU再调度其它线程。

2）run()方法当作普通方法的方式调用。程序还是要顺序执行，要等待run方法体执行完毕后，才可继续执行下面的代码； 程序中只有主线程——这一个线程， 其程序执行路径还是只有一条， 这样就没有达到多线程的目的。

### 4.Runnable接口和Callable接口的区别

1）Runnable接口里面的run()方法是没有返回值的，所以实现Runnable接口一般是执行不带返回值的任务。

2）Callable接口里面的call()方法是有返回值的，返回值是一个泛型，和Future、FutureTask配合可以用来获取异步执行的结果。 有没有返回值也是这两个接口的主要区别，Callable接口的功能更加强大，使用Callable+Future/FutureTask可以获取多线程运行的结果，可以在等待时间太长没获取到需要的数据的情况下取消该线程的任务，非常有用。

### 5. CyclicBarrier和CountDownLatch的区别

这两个类都是J.U.C 并发包下面非常有用的类：

1）CountDownLatch(闭锁)相当于一扇门：在闭锁到达结束状态之前（闭锁的结束状态也就是闭锁的计数器减到零），这扇门一直是关闭的，并且没有任何线程可以通过；当闭锁到达结束状态时，这扇门会打开并允许所有的等待线程(在闭锁上调用了await()方法的线程)通过。并且当闭锁到达结束状态后不可逆转，这扇门会一直保持打开的状态。应用场景：确保某些活动直到其他活动都结束了才继续执行。

2）CyclicBarrier 栏珊，它允许一组线程互相等待，直到都到达某个公共屏障点 (common barrier point)。在涉及一组固定大小的线程的程序中，这些线程必须不时地互相等待，此时 CyclicBarrier 很有用。因为该 barrier 在释放等待线程后可以重用，所以称它为循环 的 barrier。CyclicBarrier就象它名字的意思一样，可看成是个障碍， 所有的线程必须到齐后才能一起通过这个障碍。

### 6. volatile关键字的作用

简单点说就是提供了可视性机制。其实在理解volatile关键字的作用的前提是要理解Java内存模型，关于Java内存模型了，可以参见第31点，volatile关键字的作用主要有两个：

1）可视性：多线程主要围绕可见性和原子性两个特性而展开，使用volatile关键字修饰的变量，保证了其在多线程之间的可见性，即每次读取到volatile变量，一定是最新的数据

2）防止重排序：代码底层执行不像我们看到的高级语言----Java程序这么简单，它的执行是Java代码-->字节码-->根据字节码执行对应的C/C++代码-->C/C++代码被编译成汇编语言-->和硬件电路交互，现实中，为了获取更好的性能JVM可能会对指令进行重排序，多线程下可能会出现一些意想不到的问题。使用volatile则会对禁止语义重排序，当然这也一定程度上降低了代码执行效率。 volatile的一个重要作用就是和CAS结合，保证了原子性与可视性，详细的可以参见java.util.concurrent.atomic包下的类，比如AtomicInteger。

### 什么是线程安全

这个问题没有标准答案，每本书给出的解释也不一样。就我的理解来说：一段代码在单线程和多线程并发下永远都能获得一样的结果，那么就是线程安全的。

有值得一提的地方，就是线程安全也是有几个级别的：

1）不可变(对象) String、Integer、Long这些，都是final类型的类，还有final类型的简单变量。任何一个线程都改变不了它们的值，要改变除非新创建一个，因此这些不可变对象不需要任何同步手段就可以直接在多线程环境下使用。

2）绝对线程安全 不管运行时环境如何，调用者都不需要额外的同步措施。要做到这一点通常需要付出许多额外的代价，Java中标注自己是线程安全的类，实际上绝大多数都不是线程安全的，不过绝对线程安全的类，Java中也有，比方说CopyOnWriteArrayList、CopyOnWriteArraySet

3）相对线程安全 相对线程安全也就是我们通常意义上所说的线程安全，像Vector这种，add、remove方法都是原子操作，不会被打断，但也仅限于此，如果有个线程在遍历某个Vector、有个线程同时在add这个Vector，99%的情况下都会出ConcurrentModificationException，也就是fail-fast机制。

4）线程非安全 这个就没什么好说的了，ArrayList、LinkedList、HashMap等都是线程非安全的类，没有加任何保证线程安全的措施。

### 8. 一个线程如果出现了运行时异常会怎么样

如果这个异常没有被捕获的话，这个线程就停止执行了。另外重要的一点是：如果这个线程持有某个某个对象的监视器，那么这个对象锁会被立即释放.

### 9. 如何在两个线程之间共享数据

通过线程之间共享对象就可以了，然后通过wait/notify/notifyAll、await/signal/signalAll进行唤起和等待，这里需要注意的是要保证多线程环境下的数据安全性。比方说阻塞队列BlockingQueue就是为线程之间共享数据而设计的。

### 10.sleep方法和wait方法有什么区别

1）相同点：sleep方法和wait方法都可以用来放弃CPU一定的时间。

2）不同点：在于如果线程持有某个对象的锁，sleep方法不会放弃这个对象的锁；wait方法会放弃这个对象的锁。

### 11. 生产者消费者模型的作用是什么

1）通过平衡生产者的生产能力 和 消费者的消费能力来提升整个系统的运行效率,这是生产者消费者模型最重要的作用。

2）解耦，这是生产者消费者模型附带的作用，解耦意味着生产者和消费者之间的联系少，联系越少越可以独自发展而不需要收到相互的制约。

### 12．ThreadLocal有什么用？

简单说ThreadLocal就是一种以 空间换时间 的做法，在每个Thread里面维护了一个以开地址法实现的ThreadLocal.ThreadLocalMap，把数据进行隔离，数据不共享，自然就没有线程安全方面的问题了。

### 13.为什么wait()方法和notify()/notifyAll()方法要在同步块中被调用

这是JDK强制的，wait()方法和notify()/notifyAll()方法在调用前都必须先获得对象的锁。

### 14. wait()方法和notify()/notifyAll()方法在放弃对象锁时有什么区别

wait()方法立即释放对象锁，notify()/notifyAll()方法则会等待线程剩余代码执行完毕才会放弃对象的锁。

### 15. 为什么要使用线程池

1）避免频繁地创建和销毁线程，达到线程对象的重用。（创建和销毁线程的开销还是很大的）

2）使用线程池还可以根据项目灵活地控制并发的数目。

### 16. 怎么检测一个线程是否持有对象监视器

有方法可以判断某个线程是否持有对象的锁：Thread类提供了一个holdsLock(Object obj)方法，当且仅当对象obj的监视器被某条线程持有的时候才会返回true，注意这是一个static方法，这意味着"某条线程"指的是当前线程。

### 17. synchronized和ReentrantLock的区别

synchronized是和if、else、for、while一样的关键字，ReentrantLock是类，这是二者的本质区别。既然ReentrantLock是类，那么它就提供了比synchronized更多更灵活的特性，可以被继承、可以有方法、可以有各种各样的类变量，ReentrantLock比synchronized的扩展性体现在几点上：

1）ReentrantLock可以对获取锁的等待时间进行设置，这样就避免了死锁 ；

2）ReentrantLock可以获取各种锁的信息 ；

3）ReentrantLock可以灵活地实现多路通知 另外，二者的锁机制其实也是不一样的。ReentrantLock底层是基于AQS实现加锁，synchronized操作的是对象头中mark word。

### 18. ConcurrentHashMap的并发度是多少

看看源码就知道ConcurrentHashMap使用了分段锁，默认的并发度是16。ConcurrentHashMap的并发度就是segment的大小，默认为16，这意味着最多同时可以有16条线程操作ConcurrentHashMap，这也是ConcurrentHashMap对Hashtable的最大优势。

### 19. ReadWriteLock是什么？

首先明确一下，不是说ReentrantLock不好，只是ReentrantLock某些时候有局限。如果使用ReentrantLock，可能加锁本身是为了防止线程A在写数据、线程B在读数据造成的数据不一致。但这样，如果线程C在读数据、线程D也在读数据，读数据是不会改变数据的，没有必要加锁，但是还是加锁了，降低了程序的性能。 因为这个，才诞生了读写锁ReadWriteLock。ReadWriteLock是一个读写锁接口，ReentrantReadWriteLock是ReadWriteLock接口的一个具体实现，实现了读写的分离，读锁是共享的，写锁是独占的，读和读之间不会互斥，读和写、写和读、写和写之间才会互斥，提升了读写的性能。

### 20. FutureTask是什么

FutureTask表示一个异步运算的任务。FutureTask里面可以传入一个Callable的具体实现类，可以对这个异步运算的任务的结果进行等待获取、判断是否已经完成、取消任务等操作。当然，由于FutureTask也是Runnable接口的实现类，所以FutureTask也可以放入线程池中。

### 21. Java编程写一个会导致死锁的程序

死锁：线程A和线程B相互等待对方持有的锁导致程序无限死循环下去。实践中碰到死锁的问题基本上是看不出来的， 真正理解什么是死锁，这个问题其实不难，几个步骤：

1）两个线程里面分别持有两个Object对象：lock1和lock2。这两个lock作为同步代码块的锁。 2）线程1的run()方法中同步代码块先获取lock1的对象锁，Thread.sleep(xxx)，时间不需要太多，50毫秒差不多了，然后接着获取lock2的对象锁。这么做主要是为了防止线程1启动一下子就连续获得了lock1和lock2两个对象的对象锁 。

3）线程2的run)(方法中同步代码块先获取lock2的对象锁，接着获取lock1的对象锁，当然这时lock1的对象锁已经被线程1锁持有，线程2肯定是要等待线程1释放lock1的对象锁的 这样，线程1"睡觉"睡完，线程2已经获取了lock2的对象锁了，线程1此时尝试获取lock2的对象锁，便被阻塞，此时一个死锁就形成了。

### 22. 怎么唤醒一个阻塞的线程

1）如果线程是因为调用了wait()、sleep()或者join()方法而导致的阻塞，可以中断线程，并且通过抛出InterruptedException来唤醒它；

2）如果线程遇到了IO阻塞就无能为力，因为IO是操作系统实现的，Java代码并没有办法直接接触到操作系统。

### 23. 不可变对象对多线程有什么帮助

不可变对象保证了对象的内存可见性，而且不可变对象天生就是线程安全的。

### 24. 什么是多线程的上下文切换

多线程的上下文切换是指CPU控制权由一个已经正在运行的线程切换到另外一个就绪并等待获取CPU执行权的线程的过程，这其中涉及到运行环境的保存与切换，存在时间开销。

### 25. 如果你提交任务时，线程池队列已满，这时会发生什么？

1）如果使用的是无界队列（比如LinkedBlockingQueue），就没什么影响，可以直接添加任务到阻塞队列中等待执行，因为LinkedBlockingQueue可以近乎认为是一个无穷大的队列，可以无限存放任务。

2）如果使用的是有界队列。比方说ArrayBlockingQueue的话，任务首先会被添加到ArrayBlockingQueue中，ArrayBlockingQueue满了，则会使用拒绝策略RejectedExecutionHandler处理满了的任务，默认是AbortPolicy。

### 26. Java中用到的线程调度算法是什么

抢占式。一个线程用完CPU之后，操作系统会根据线程优先级、线程饥饿情况等数据算出一个总的优先级并分配下一个时间片给某个线程执行。

### 27. Thread.sleep(0)的作用是什么

简单点说就是当前线程主动让出一次CPU：。

由于Java采用抢占式的线程调度算法，因此可能会出现某条线程常常获取到CPU控制权的情况，为了让某些优先级比较低的线程也能获取到CPU控制权，可以使用Thread.sleep(0)手动触发一次操作系统分配时间片的操作，这也是平衡CPU控制权的一种操作。

### 28. 什么是自旋？

很多synchronized里面的代码只是一些很简单的代码，执行时间非常快，此时等待的线程都加锁可能是一种不太值得的操作，因为线程阻塞涉及到用户态和内核态切换的问题。既然synchronized里面的代码执行得非常快，不妨让等待锁的线程不要被阻塞，而是在synchronized的边界做忙循环，这就是自旋。如果做了多次忙循环发现还没有获得锁，再阻塞，这样可能是一种更好的策略。

### 29. 什么是Java内存模型

Java内存模型定义了一种多线程访问Java内存的规范，简单总结一下Java内存模型的几部分内容：

（1）Java内存模型将内存分为了主内存和工作内存。类的状态，也就是类之间共享的变量，是存储在主内存中的，每次Java线程用到这些主内存中的变量的时候，会读一次主内存中的变量，并让这些内存在自己的工作内存中有一份拷贝，运行自己线程代码的时候，用到这些变量，操作的都是自己工作内存中的那一份。在线程代码执行完毕之后，会将最新的值更新到主内存中去

（2）定义了几个原子操作，用于主内存和工作内存中的变量的交互。

（3）定义了volatile变量的使用规则 。

（4）原子性、可见性、有序性。

（5）happens-before，即先行发生原则，定义了操作A必然先行发生于操作B的一些规则，比如在同一个线程内控制流前面的代码一定先行发生于控制流后面的代码、一个释放锁unlock的动作一定先行发生于后面对于同一个锁进行锁定lock的动作等等，只要符合这些规则，则不需要额外做同步措施，如果某段代码不符合所有的happens-before规则，则这段代码一定是线程非安全的。

### 30．什么是CAS

CAS，全称为Compare and Swap，即比较-替换。假设有三个操作数：内存值V、旧的预期值A、要修改的值B，当且仅当预期值A和内存值V相同时，才会将内存值修改为B并返回true，否则什么都不做并返回false。当然CAS一定要volatile变量配合，这样才能保证每次拿到的变量是主内存中最新的那个值，否则旧的预期值A对某条线程来说，永远是一个不会变的值A，只要某次CAS操作失败，永远都不可能成功。 比如原子类：AtomicInteger 就是volatile + 硬件级别的CAS实现的。

### 31. 什么是乐观锁和悲观锁

其实这两种锁就是对线程安全的最悲观和最乐观的假设。

（1）乐观锁：就像它的名字一样，对于并发间操作产生的线程安全问题持乐观状态，乐观锁认为竞争不总是会发生，因此它不需要持有锁，将比较-替换这两个动作作为一个原子操作尝试去修改内存中的变量，如果失败则表示发生冲突，那么就应该有相应的重试逻辑。

（2）悲观锁：还是像它的名字一样，对于并发间操作产生的线程安全问题持悲观状态，悲观锁认为竞争总是会发生，因此每次对某资源进行操作时，都会持有一个独占的锁，就像synchronized，不管三七二十一，直接上了锁就操作资源了。

### 32. 单例模式的线程安全性

老生常谈的问题了，首先要说的是单例模式的线程安全意味着：某个类的实例在多线程环境下只会被创建一次出来。单例模式有很多种的写法，我总结一下：

（1）饿汉式单例模式的写法：线程安全 ；

（2）懒汉式单例模式的写法：非线程安全 ；

（3）双检锁(DCL)单例模式的写法：线程安全 。

（4）静态内部类。

### 33. Semaphore有什么作用

常用作并发数控制器。Semaphore就是一个信号量，它的作用是限制某段代码块的并发数。Semaphore有一个构造函数，可以传入一个int型整数n，表示某段代码最多只有n个线程可以访问，如果超出了n，那么请等待，等到某个线程执行完毕这段代码块，下一个线程再进入。由此可以看出如果Semaphore构造函数中传入的int型整数n=1，相当于变成了一个synchronized了。

### 34. Hashtable的size()方法中明明只有一条语句"return count"，为什么还要做同步？

（1）同一时间只能有一条线程执行固定类的同步方法，但是对于类的非同步方法，可以多条线程同时访问。所以，这样就有问题了，可能线程A在执行Hashtable的put方法添加数据，线程B则可以正常调用size()方法读取Hashtable中当前元素的个数，那读取到的值可能不是最新的，可能线程A添加了完了数据，但是没有对size++，线程B就已经读取size了，那么对于线程B来说读取到的size一定是不准确的。而给size()方法加了同步之后，意味着线程B调用size()方法只有在线程A调用put方法完毕之后才可以调用，这样就保证了线程安全性。

（2）CPU执行代码，执行的不是Java代码，这点很关键，一定得记住。Java代码最终是被翻译成汇编代码执行的，汇编代码才是真正可以和硬件电路交互的代码。即使你看到Java代码只有一行，甚至你看到Java代码编译之后生成的字节码也只有一行，也不意味着对于底层来说这句语句的操作只有一个。一句"return count"假设被翻译成了三句汇编语句执行，完全可能执行完第一句，线程就切换了。

### 35. 线程类的构造方法、静态块是被哪个线程调用的？

线程类的构造方法、静态块是被new这个线程类所在的线程所调用的，而run方法里面的代码才是被线程自身所调用的。

举个例子，假设Thread2中new了Thread1，main函数中new了Thread2，那么：

（1）Thread2的构造方法、静态块是main线程调用的，Thread2的run()方法是Thread2自己调用的。

（2）Thread1的构造方法、静态块是Thread2调用的，Thread1的run()方法是Thread1自己调用的。

### 36. 同步方法和同步块，哪个是更好的选择？

1）同步块，这意味着同步块之外的代码是异步执行的，这比同步整个方法更提升代码的效率。请知道一条原则：同步的范围越小越好。

2）虽说同步的范围越少越好，但是在Java虚拟机中还是存在着一种叫做锁粗化的优化方法，这种方法就是把同步范围变大。这是有用的，比方说StringBuffer，它是一个线程安全的类，自然最常用的append()方法是一个同步方法，我们写代码的时候会反复append字符串，这意味着要进行反复的加锁->解锁，这对性能不利，因为这意味着Java虚拟机在这条线程上要反复地在内核态和用户态之间进行切换，因此Java虚拟机会将多次append方法调用的代码进行一个锁粗化的操作，将多次的append的操作扩展到append方法的头尾，变成一个大的同步块，这样就减少了加锁-->解锁的次数，有效地提升了代码执行的效率。

### 37. 什么是AQS？

1）简单说一下AQS，AQS全称为AbstractQueuedSychronizer，翻译过来应该是抽象队列同步器。

2）如果说java.util.concurrent的基础是CAS的话，那么AQS就是整个Java并发包的核心了，ReentrantLock、CountDownLatch、Semaphore等等都用到了它。AQS实际上以双向队列的形式连接所有的Entry，比方说ReentrantLock，所有等待的线程都被放在一个Entry中并连成双向队列，前面一个线程使用ReentrantLock好了，则双向队列实际上的第一个Entry开始运行。

3）AQS定义了对双向队列所有的操作，而只开放了tryLock和tryRelease方法给开发者使用，开发者可以根据自己的实现重写tryLock和tryRelease方法，以实现自己的并发功能。

### 38. 高并发、任务执行时间短的业务怎样使用线程池？并发不高、任务执行时间长的业务怎样使用线程池？并发高、业务执行时间长的业务怎样使用线程池？

（1）高并发、任务执行时间短的业务，线程池线程数可以设置为CPU核数+1，减少线程上下文的切换 。

（2）并发不高、任务执行时间长的业务要区分开看：

a）假如是业务时间长集中在IO操作上，也就是IO密集型的任务，因为IO操作并不占用CPU，所以不要让所有的CPU闲下来，可以加大线程池中的线程数目，让CPU处理更多的业务 。

b）假如是业务时间长集中在计算操作上，也就是计算密集型任务，这个就没办法了，和（1）中一样吧，线程池中的线程数设置得少一些，减少线程上下文的切换

（3）并发高、业务执行时间长，解决这种类型任务的关键不在于线程池而在于整体架构的设计，看看这些业务里面某些数据是否能做缓存是第一步，增加服务器是第二步，至于线程池的设置，设置参考（2）。最后，业务执行时间长的问题，也可能需要分析一下，看看能不能使用中间件对任务进行拆分和解耦。

### 39 wait方法是否必须用在while循环体中

对于从wait中被notify的进程来说，它在被notify之后还需要重新检查是否符合执行条件，如果不符合，就必须再次被wait，如果符合才能往下执行。所以：wait方法应该使用循环模式来调用。按照上面的生产者和消费者问题来说：错误情况一：如果有两个生产者A和B，一个消费者C。当存储空间满了之后，生产者A和B都被wait，进入等待唤醒队列。当消费者C取走了一个数据后，如果调用了notifyAll（），注意，此处是调用notifyAll（），则生产者线程A和B都将被唤醒，如果此时A和B中的wait不在while循环中而是在if中，则A和B就不会再次判断是否符合执行条件，都将直接执行wait（）之后的程序，那么如果A放入了一个数据至存储空间，则此时存储空间已经满了；但是B还是会继续往存储空间里放数据，错误便产生了。错误情况二：如果有两个生产者A和B，一个消费者C。当存储空间满了之后，生产者A和B都被wait，进入等待唤醒队列。当消费者C取走了一个数据后，如果调用了notify（），则A和B中的一个将被唤醒，假设A被唤醒，则A向存储空间放入了一个数据，至此空间就满了。A执行了notify（）之后，如果唤醒了B，那么B不会再次判断是否符合执行条件，将直接执行wait（）之后的程序，这样就导致向已经满了数据存储区中再次放入数据。错误产生。

### 40 Runnable 和 Thread的区别

在程序开发中只要是多线程肯定永远以实现Runnable接口为主，因为实现Runnable接口相比继承Thread类有如下好处：

* 避免了继承的局限，一个类可以继承多个接口。
* 适合于资源的共享

以卖票程序为例，通过Thread类完成要卖出30张票。虽然现在程序中有三个线程，但是一共卖了10张票，也就是说使用Runnable实现多线程可以达到资源共享目的。

### 41. 同步、阻塞与异步的辨析？

同步和异步关注的是消息通信机制

1）同步：就是在发出一个调用时，在没有得到结果之前，该调用就不返回。但是一旦调用返回，就得到返回值了。（调用者一直等待调用结果）

2）异步：调用在发出之后，这个调用就直接返回了，所以没有返回结果。但是等待调用的事件被处理完之后在通知调用者。

阻塞和非阻塞关注的是程序等待调用结果时的状态。

1）阻塞：阻塞调用是指调用结果返回之前，当前线程会被挂起。函数只有在得到结果之后才会返回。

2）非阻塞指：指调用在不能立刻得到结果之前，该调用不会阻塞当前线程，而是立即返回。

# Java基础

### Java的4个基本特性，并说说对多态的理解。

1）抽象、封装、继承、多态。

多态的理解：

1. 多态分为编译时多台和运行时多态。编译时多台一般都是通过重载方法的方式实现，根据参数列表的不同来区分不同函数； 运行时多态一般就是 面向接口的编程 和 (基于继承的多态)父类引用子类的实例，也就是父类对象指向子类的实例。

### String、StringBuffer、StringBuilder区别？

1. 在JDK源码中，String类是被修饰成final的，所以String类是不可变的。我们对于String对象的操作都会返回一个新的String对象。所以对String进行字符串的拼接的时候会创建大量的String实例对象，这是一个非常耗时的操作，所以String直接拼接字符串效率很低。
2. StringBuffer和StringBuilder是可变的。他们两个都是继承于AbstractStringBuilder，内部是基于char[]字符数组实现的，其对于字符串的拼接等操作是基于数组，所以效率很高。此外StringBuilder是非线程安全的，StringBuffer是线程安全的。StringBuffer内部的所有方法都是使用synchronized锁实现的同步方法，所以在多线程环境下使用StringBuffer。StringBuilder的方法没有使用任何同步机制，所以是非线程安全的。注意在单线程环境下StringBuilder效率比StringBuffer高，因为没有加锁释放锁的时间。

### Object类有哪些方法，分别是什么作用？

1）protected Object clone() 创建并返回此对象的一个副本。   
2）boolean equals(Object obj) 指示其他某个对象是否与此对象“相等”。   
3）protected void finalize() 当垃圾回收器确定不存在对该对象的更多引用时，由对象的垃圾回收器调用此方法。   
4）Class<?> getClass()返回此Object 的运行时类对象。   
5）int hashCode() 返回该对象的哈希码值。   
6）void notify() 唤醒在此对象监视器上等待的单个线程。   
7）void notifyAll() 唤醒在此对象监视器上等待的所有线程。   
8）String toString() 返回该对象的字符串表示。   
9）void wait() 在其他线程调用此对象notify()方法或 notifyAll()方法前，导致当前线程等待。   
10）void wait(long timeout) 在其他线程调用此对象的 notify() 方法或 notifyAll() 方法，或者超过指定的时间量前，导致当前线程等待。   
11）void wait(long timeout, int nanos) 在其他线程调用此对象的 notify() 方法或 notifyAll() 方法，或者其他某个线程中断当前线程，或者已超过某个实际时间量前，导致当前线程等待。

### Static关键字的作用

静态变量、静态方法、静态代码快、内部静态类。

### 5. 抽象类与接口的区别

1）抽象类与接口都不能实例化；

1. 抽象类要被子类继承，接口要被类实现；

3）接口只能声明方法，抽象类即可以声明方法也可以实现方法；

4）接口里面的变量只能是公共静态常量，抽象类里面可以定义变量；

5）抽象类必须被子类实现所有抽象方法，不然子类也是抽象类；

6）接口可以多继承接口。但类只能单继承。

### 6. 内存溢出和内存泄露的区别

1. 内存溢出:指的是程序在申请内存的时候，没有足够大的空间可以分配了。
2. 内存泄露:指的是程序在申请内存之后，没有办法释放掉已经申请到内存，它始终占用着内存，即被分配的对象可达但无用。内存泄露一般都是因为内存中有一块很大的对象，但是无法释放。

### NIO、BIO、AIO分别指什么

BIO：同步阻塞IO，一个连接一个线程；

AIO：异步非阻塞IO，一个有效请求一个线程；

NIO：同步非阻塞IO，一个请求一个线程

### Try catch finally结构的代码，try块里面有直接return语句，finally还执行吗？

finally会执行！不管try块里面有没有return语句，finally里面的语句肯定是会执行的。

### 10 如果我们在finally块中改变某个返回值的变量的值，最后的return结果会变吗？

不会，比如下面的例子：

public static int i=0;

public static int testtest(){

try {

i=10;

System.out.println("try");

return i;

}finally {

i=100;

System.out.println("finally: " + i);

}

}

@org.junit.Test

public void test1(){

System.out.println("test:"+ testtest());

System.out.println(“i= ”i);

}

输出就是：

try

finally: 100

test:10

i= 100

根据输出我们就可以知道： 在执行return 操作的时候，其实return的值就已经准备好了（这里的准备好了应该类似于副本），无论你在finally怎么对变量改变都不会影响return的值。

### 11．静态内部类和非静态内部类的区别？

1）java允许我们在一个类里面定义静态内部类（nested class），把nested class封闭起来的类叫外部类。在java中，我们不能用static修饰顶级类（top level class）。只有内部类可以为static。

2）静态内部类和非静态内部类之间到底有什么不同呢？

（1）静态内部类不需要有指向外部类的引用。但非静态内部类需要持有对外部类的引用。

（2）静态内部类不能访问外部类的非静态成员，他只能访问外部类的静态成员。

非静态内部类能够访问外部类的静态和非静态成员。

（3）一个非静态内部类不能脱离外部类实体被创建，必须通过外部类的引用创建实例。一个非静态内部类可以访问外部类的数据和方法，因为他就在外部类里面。

### 12. Java的反射作用原理与作用：

1）原理：在程序运行的时候能够获取自身的的Class信息，比如属性和方法，且能够调用它的任意的方法和属性，这就为Java提供了动态的特性。

2）Java反射的作用：

1. 在运行时判断任意一个对象所属的类；   
2. 在运行时构造任意一个类的对象；   
3. 在运行时判断任意一个类所具有的成员变量和方法；   
4. 在运行时调用任意一个对象的方法；   
5. 生成动态代理。

### 13. Java的动态代理怎么理解？

**Java**的动态代理可以动态的创建代理并动态的处理对所代理方法的调用。在动态代理上所做的所有调用都会被重定向到单一的 调用处理器 上，它的工作是揭示调用的类型并确定相应的对策。常规套路是向 调用处理器 的构造器传递一个“实际”对象的引用，从而使得调用处理器在执行其中介任务时，可以将请求转发。

# Java的集合框架

### HashMap是怎么实现的？

数据结构上是基于：数组+单链表；

无序，特别说明这个无序指的是遍历HashMap的时候，得到的元素的顺序基本不可能是put的顺序。

### HashMap和HashTable的区别：

1. 继承的类不一样：

HashMap继承的是AbstractMap，Hashtable继承的是Dictionary。实现的接口一致(Map、Cloneable和Serializable)

2）初始容量不一样：

HashMap默认容量是16，且容量只能是2 的幂次方，这与计算key的hashcode算法有关，计算key的存储位置时，用的是i=hashcode&(table.length-1)， 只有容量是2的幂次方时才与hashcode%( table.length-1)等价，此外HashMap是在第一次put键值对时才初始化。

HashTable默认容量是11，容量没有2的幂次方限制，每次扩容大小是oldCap\*2+1;

3）HashMap 是非线程安全的，HashTable是线程安全的：

默认下，HashMap没有对方法进行同步，HashTable中方法均是使用synchronized进行了同步。

4）HashMap允许有一个key为null, 并且存储在table[0]位置，允许多个value为null；

但是HashTable不允许value 为空。

### 3. Vector、ArrayList、LinkedList的区别

1）Vector 和ArrayList最大的区别在于，Vector里面打核心方法都是synchronized方法，此外Vector每次扩容都是2倍扩容，而ArrayList是扩容50%。

2）ArrayList是基于数组实现的，里面的元素允许为空且允许重复，且基本有序(插入和遍历顺序一致)，但是是非线程安全的。还有一点需要注意的是ArrayList的subList()函数返回的并不是一个新的List对象，仅仅只是父类的一个视图，对subList的修改会影响到原本的父List。

3）LinkedList实现了Queue接口，所以可以作为一个FIFO的队列使用。LinkedList内部是基于双向链表实现的，元素可为空也可重复，并且遍历的时候是有序的。和ArrayList一样同样是非线程安全的。

4）LinkedList千万不能用for循环遍历，for循环遍历会比迭代器慢上n倍，因为LinkedList是基于双向链表实现的，for循环遍历，每次遍历数据都会把前面的数据走一遍，所以是非常耗时的。

### 4.HashMap 和 ConcurrentHashMap的区别

1）ConcurrentHashMap是基于分段锁实现的，具体可以理解成把一个大的Map拆分成N个小的HashTable（默认是16个），根据key.hashCode()来决定把key放到哪个HashTable。在ConcurrentHashMap中就是把Map分成了N个Segment，put和get的时候，都是先根据key.hashCode()算出放在哪个Segment中，然后对对应的Segment加锁，这样就不会影响其余Segment的并发访问。这样效率就提升了N倍，默认提升了16倍。其实也就是同时允许16个线程分别对16个Segment操作，只有写操作才需要锁住Segment，读线程基本不受限制。

2）基本上ConcurrentHashMap是HashMap 和 HashTable的结合。那么为什么ConcurrentHashMap里面的get()操作，也就是读操作不用加锁呢，除非读到的值是空的才会加锁重读？

原因是它的get方法里将要使用的共享变量都定义成volatile，如用于统计当前Segement大小的count字段和用于存储值的HashEntry的value。定义成volatile的变量，能够在线程之间保持可见性，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（有一种情况可以被多线程写，就是写入的值不依赖于原值），在get操作里只需要读不需要写共享变量count和value，所以可以不用加锁。之所以不会读到过期的值，是根据java内存模型的happen before原则，对volatile字段的写入操作优先于读操作，即使两个线程同时修改和获取volatile变量，get操作也能拿到最新的值，这是用volatile替换锁的经典应用场景。

3）ConcurrentHashMap不允许key或则value为空。

# 设计模式

### 1. 单例模式

1）定义： 保证一个类仅有一个实例，并提供一个访问的全局访问函数。

2）注意：构造器一定要是private的，不让外部调用构造器。

3）实现：

(1)饿汉式单例模式，在定义时就初始化；

(2)懒汉式单例模式（线程安全）：synchronized方法，synchronized同步块，ReentrantLock显示锁、基于DCL的双检查（volatile关键字）、使用静态内部类来实现单例。

### 2. 工厂设计模式

### 3. 适配器设计模式

### 4. 责任链模式

1）定义：避免请求发送者与接收者耦合在一起，让多个对象都有可能接收请求，将这些对象连接成一条链，并且沿着这条链传递请求，直到有对象处理它为止，这就是职责链模式。

2）在职责链模式中我们可以随时随地的增加或者更改一个处理者，甚至可以更改处理者的顺序，增加了系统的灵活性。处理灵活性是增加了，但是有时候可能会导致一个请求无论如何也得不到处理，它会被放置在链末端，这个既是职责链的优点是缺点

3）优点

(1)降低耦合度。它将请求的发送者和接受者解耦。

(2)简化了对象。使得对象不需要知道链的结构。

(3)增强给对象指派职责的灵活性。通过改变链内的成员或者调动它们的次序，允许动态地新增或者删除责任。

(4)增加新的请求处理类很方便。

4）缺点

  (1)不能保证请求一定被接收。

(2)系统性能将受到一定影响，而且在进行代码调试时不太方便；可能会造成循环调用。

(3)可能不容易观察运行时的特征，有碍于除错。

5)模式适用场景

(1)有多个对象可以处理同一个请求，具体哪个对象处理该请求由运行时刻自动确定。

(2)在不明确指定接收者的情况下，向多个对象中的一个提交一个请求。

(3)可动态指定一组对象处理请求。

### 5. 观察者模式

1）定义：观察者模式也叫发布—订阅模式，也就是时间监听机制。观察者模式定义了一种一对多的依赖关系，让多个观察者对象同时监听某一个主体对象，当这个主体对象在状态上发生改变时就会通知所有观察者对象（通过调用观察者接口的函数更新观察者状态），使他们能够自动更新自己。

2）观察者模式有两种模型：

（1）推模型

主题对象向观察者推送主题的详细信息，不管观察者是否需要。推送的信息通常是主题对象的全部或部分数据。

（2）拉模型

主题对象在通知观察者的时候，只传递少量信息。如果观察者需要更具体的信息，由观察者主动到主题对象中去获取，相当于是观察者从主题对象中拉数据。一般这种模型的实现中，会把主题对象自身通过update()方法传递给观察者，这样观察者在需要获取数据的时候，就可以通过这个引用来获取了。

1. 优点：

1、去重复代码，使得代码更清晰、更易读、更易扩展

2、解耦，使得代码可维护性更好，修改代码的时候可以尽量少改地方

使用观察者模式可以很好地做到这两点。增加观察者，直接new出观察者并注册到主题对象之后就完事了，删除观察者，主题对象调用方法删除一下就好了，其余都不用管。主题对象状态改变，内部会自动帮我们通知每一个观察者，是不是很方便。

4）观察者模式主要应用场景有：

1、对一个对象状态的更新需要其他对象同步更新

2、对象仅需要将自己的更新通知给其他对象而不需要知道其他对象的细节，如消息推送

### 代理模式

### 建造者模式

# 操作系统等相关的：

### 1. 进程间通信的方式有哪几种？

1）管道

2）有名管道

3）信号量

4）消息队列

5）共享内存；

6）套接字socket

### Linux下如何进行进程调度的？

主要有三种调度策略：

1. 分时条幅策略
2. 实时调度策略—FIFO
3. 实时调度策略—时间片轮转。

### Linux常用命令

1. 文件目录相关：ls、cd、pwd、mkdir、rmdir、mv、cp、touch、cat、
2. 文件查找类型的：which、find
3. 压缩和解压：tar、unzip、
4. 文件权限：chmod
5. 进程：ps、kill
6. 网络：ssh、ping、ipconfig、netstat、scp

### 操作系统什么情况下会死锁

一般都是存在资源竞争：比如获取一个资源必须获取两把锁A和B。线程甲已经获取了锁A，线程乙已经获取到了锁B，但是线程甲要获取锁B，线程乙要获取锁A，这时候互相等待，就会导致死锁。

# 数据结构

### 1． 快速排序的过程与复杂度：

1）首先选一个基准点，从数组两端扫描数组，设两个指示标志（lo指向起始位置，hi指向末尾），首先从后半部分开始，如果发现有元素比该基准点的值小，就交换lo和hi位置的值，然后从前半部分开始扫秒，发现有元素大于基准点的值，就交换lo和hi位置的值，如此往复循环，直到lo>=hi,然后把基准点的值放到hi这个位置，一次排序就完成了。再递归扫描前半部分和后半部分。

2）时间复杂度：一般情况下是O(nlogn)， 但是当序列本身有序时候，就是最坏情况，O(n^2)。

### 2. 什么是二叉平衡树，如何插入节点，删除节点，说出关键步骤。

1）二叉平衡树：或则是一颗空树 或则它的任意一个节点的左右两个子树的高度差的绝对值不超过1，且左右子树都是二叉平衡树。

2）插入：和二叉查找树类似，只是可能会破坏树的平衡性，然后进行相应的旋转操作。

3）删除：（1）叶节点直接删除。 （2）非叶节点，要处理失衡情况。