# 一．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

# 二．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

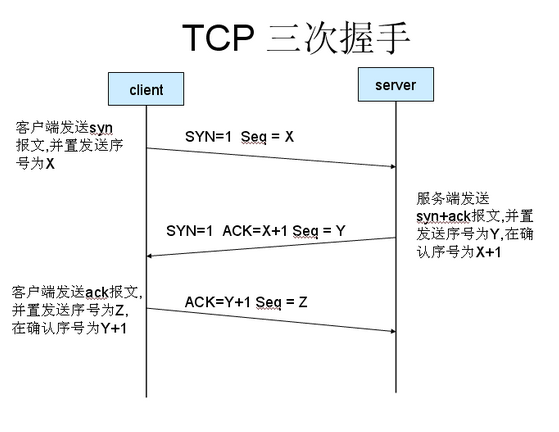
释放锁：DEL foo.lock 直接删除这个key，就表示释放锁。

# 三．计算机网络的基础知识

### TCP的三次握手、四次挥手

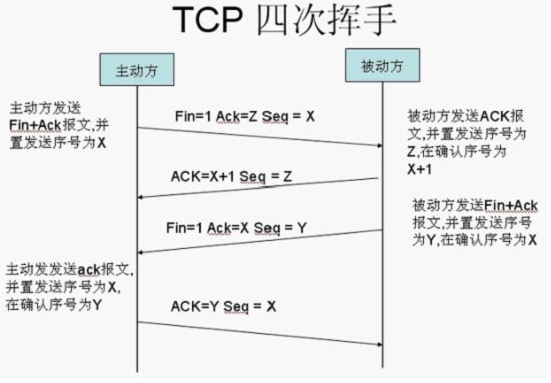
**连接的三次握手**

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。

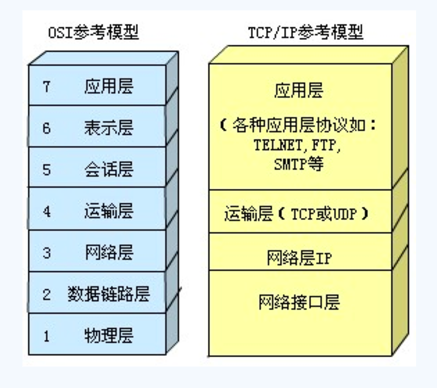
### TCP和UDP的区别

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

# 四．Java并发与多线程

# 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。

# 六． 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：

### 2. 工厂设计模式

### 3. 适配器设计模式

### 4. 责任链模式

### 5. 观察者模式

### 代理模式

### 建造者模式

# 七. 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。

# 八. 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。