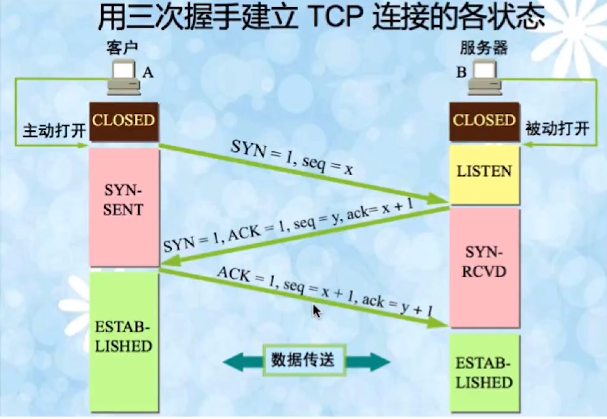
# 

# 1.计算机网络

### 1.1 说说TCP的三次握手



**第一次握手**：建立连接时，客户端发送SYN包（seq=x）到服务器，并且，进入SYN\_SEND状态，等待服务器确认

**第二次握手**：服务器收到SYN包，必须确认客户端的SYN（ack=x+1）同时自己也发送一个SYN包（seq=y），即SYN+ACK包，此时服务器进入SYN\_RECV状态

**第三次握手**：客户端收到服务器的SYN+ACK包，向服务器发送确认包（ack=y+1），此包发送完毕，客户端和服务端进入ESTABLISHED状态，完成三次握手

### 1.2 首次握手的隐患--SYN超时

Server收到Client的SYN，回复SYN-ACK后未收到ACK确认

Server不断重试直到超时，Linux默认重试5次等待63秒断开连接

这样可能造成SYN-Flood攻击，恶意程序把SYN队列耗尽

解决方案：

SYN队列满后，通过tcp-syncookies参数会发送SYN Cookie，若为正常连接的Client会回发SYN Cookie，建立连接

### 1.3 谈谈TCP的四次挥手



**第一次挥手**：Client发送一个FIN（seq=u），关闭Client到Server的数据传送，Client进入FIN\_WAIT\_1状态

**第二次挥手**：Server收到FIN后发送一个ACK（ack=u+1）给Client，Server进入CLOSE\_WAIT状态

**第三次挥手**：Serrver发送一个FIN，用来关闭Server到Client的数据传输，Server进去LAST\_ACK状态

**第四次挥手**：Client收到FIN后，Client进去TIME\_WAIT状态，接着发送一个ACK给Server（确认序号未收到序号+1），Server进去CLOSEDD状态，完成四次挥手

### 1.4 挥手为什么会有TIME\_WAIT状态

确保有足够时间让对方接收到ACK包

避免新旧连接混淆

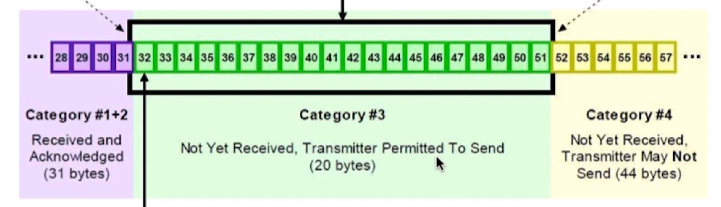
### 1.5 为什么需要四次挥手才能断开连接

因为TCP全双工，发送方和接收方都需要FIN报文和ACK报文

### 1.6 TCP和UDP的区别

* 面向连接vs面向非连接
* TCP可靠性
* TCP有序
* UDP速度快
* TCP重量级，TCP头20字节，UDP8字节

### 1.7 TCP的滑动窗口



TCP接收方缓存内有三种状态

1 已接收并发送了ack确认

2 未接收但可以接收状态（称为接收窗口）

3 未接收状态

TCP的传输可靠性来自确认重传，TCP滑动窗口可靠性也是来自确认重传，

发送方只有收到接受方对于本段发送窗口内字节的ack确认才会移动发送窗口的左边界，接收窗口只有在前面所有段都确认的情况下才会移动左边界，当前面还有字节未接收，收到后面的字节的情况下，窗口是不会移动的，并不会对后面字节进行确认，保证前面的数据会重传

### 1.8 在浏览器地址栏键入URL，按下回车之后经历的流程

1. DNS解析，由近到远依次是浏览器缓存，系统缓存，路由器缓存，IPS服务器缓存，域名服务器缓存，顶级域名服务器缓存
2. TCP连接
3. 发送HTTP请求
4. 服务器处理请求返回HTTP报文
5. 浏览器解析渲染页面，结束连接

### 1.9 常见状态码

200：正常返回信息

302：重定向

400：客户端请求有语法错误，不能被服务器理解

401：请求未经授权，这个状态码必须和WWW-Authenticate报头一起使用

403：服务器收到请求，但是拒绝提供服务

404：请求资源不存在

500：服务器发生了不可预期的错误

503：服务器当前不能处理客户端的请求，一段时间后可能恢复

### 1.10 GET和POST的区别

GET在浏览器回退时是无害的，而POST会再次提交请求。

GET产生的URL地址可以被Bookmark，而POST不可以。

GET请求会被浏览器主动cache，而POST不会，除非手动设置。

GET请求只能进行url编码，而POST支持多种编码方式。

GET请求参数会被完整保留在浏览器历史记录里，而POST中的参数不会被保留。

GET请求在URL中传送的参数是有长度限制的，而POST么有。

对参数的数据类型，GET只接受ASCII字符，而POST没有限制。

GET比POST更不安全，因为参数直接暴露在URL上，所以不能用来传递敏感信息。

GET参数通过URL传递，POST放在Request body中。

对于GET方式的请求，浏览器会把http header和data一并发送出去，服务器响应200（返回数据）,而对于POST，浏览器先发送header，服务器响应100 continue，浏览器再发送data，服务器响应200 ok（返回数据）。

### 1.11 Cookie和Session的区别

**Cookie**

是由服务器发送给客户端的特殊信息，以文本的形式存放在客户端

客户端再次请求时，会把Cookie回发

服务器收到Cookie后，会解析Cookie生成与客户端相对应的内容

**Session**

服务器端机制，在服务器上保存信息

解析客户端请求并操作sessionId，按需保存状态信息

**Session的实现方式**

Cookie实现

URL回写实现

**区别**：

Cookie数据存放在客户的浏览器上，Session数据存放在服务器上

Session相对于Cookie更加安全

Session会增加服务器负担

### 1.12 HTTP和HTTPS的区别

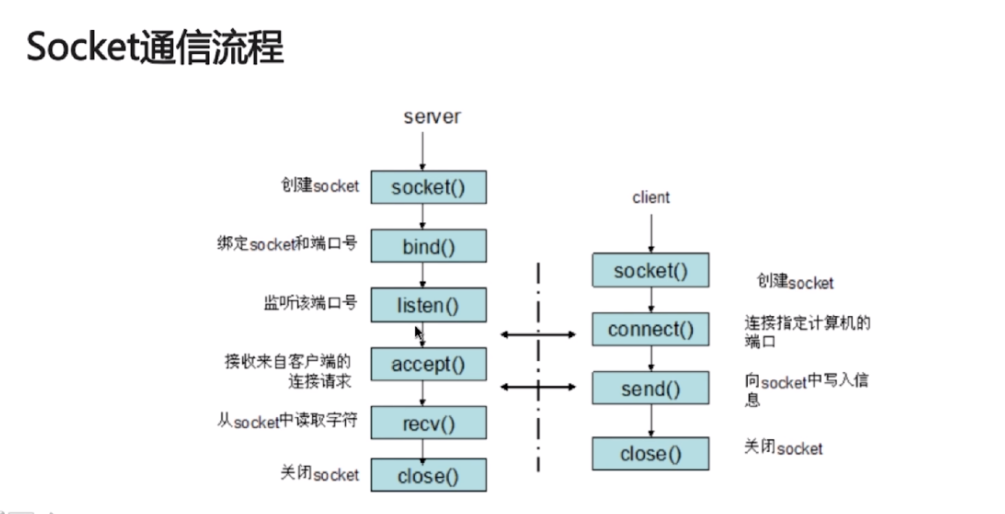
HTTPS需要到CA申请证书，HTTP不需要

HTTPS密文传输，HTTP明文传输

连接方式不同，HTTPS默认使用443端口，HTTP默认使用80端口

HTTPS=HTTP+加密+认证+完整性保护，HTTPS较HTTP安全

### 1.13 Socket简介



### 1.14 Socket编程题

编写一个网络应用程序，有客户端和服务端，客户端向服务器发送一个字符串，服务器收到该字符串后将其打印在命令行上，然后向客户端返回该字符串的长度，最后客户端输出服务器端返回的该字符串的长度，分别用TCP和UDP方式实现

# 数据库知识

### 2.1 如何设计一个关系型数据库

存储模块(文件系统)

存储管理(逻辑关系转换成物理存储关系),缓存模块,SQL解析,日志管理,权限划分,容灾机制,索引管理,锁管理

### 2.2 为什么要使用索引

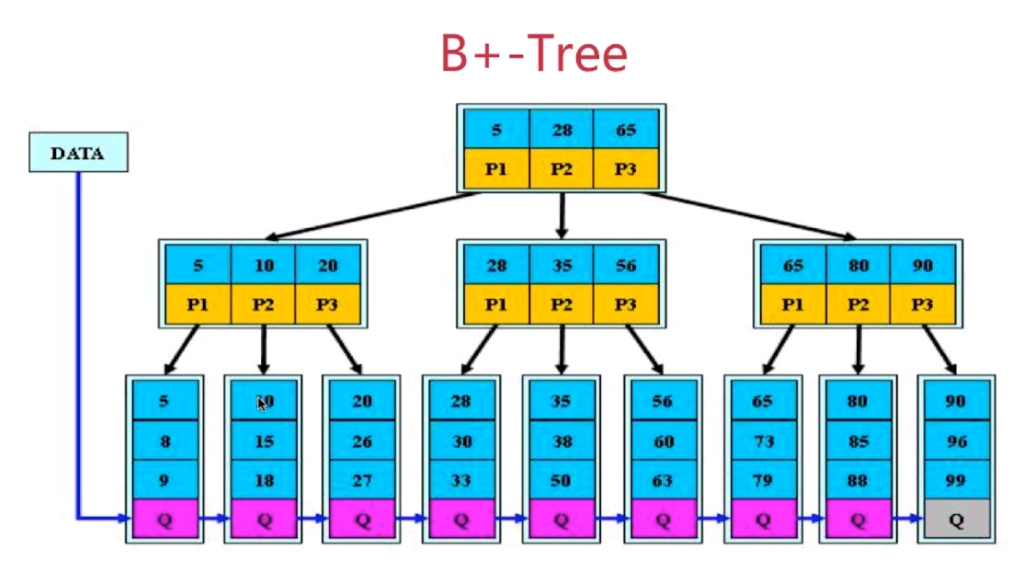
在数据量较大的情况下，为加快查询效率，尽量避免全表扫描，所以使用索引

### 2.3 什么样的信息能够成为索引

主键，唯一键，以及普通键

### 2.4 索引的数据结构

二叉树，B - Tree树，B+ - 树，Hash，BitMap



**B+ Tree**

非叶子节点的子树指针与关键字个数相同

非叶子节点的子树指针P[i],指向关键字值[k[i],k[i+1])

非叶子节点仅用来索引，数据保存在叶子节点中

所有叶子节点均有一个链指针指向下一个叶子节点

优点：

B+Tree的磁盘读写代价更低

B+Tree的查询效率更加稳定，所有叶子查询效率相同稳定O(logn)

B+Tree更有利于数据库扫描

**Hash索引的缺点**

仅仅能满足 = ，in，不能范围查询

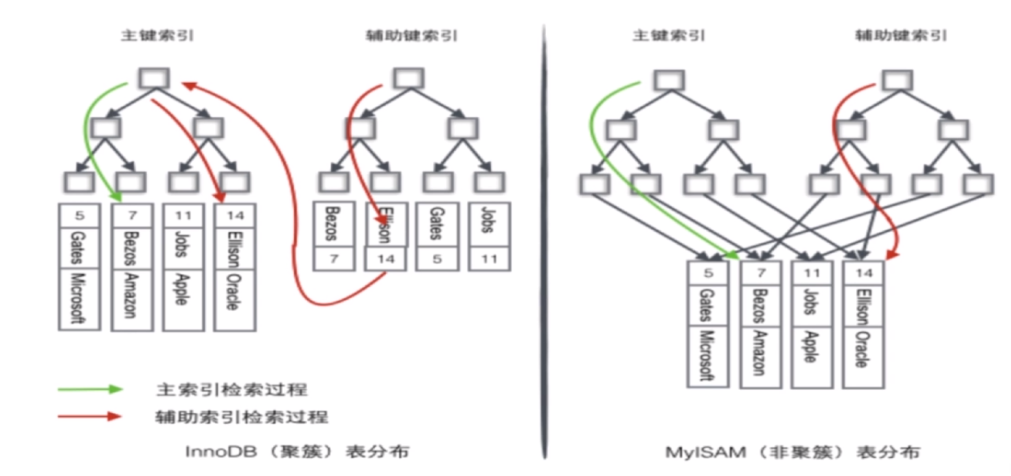
无法避免数据排序操作

不能利用部分索引查询（不支持利用组合索引中的部分索引）

不能避免表扫描（可能不同索引有相同Hash）

遇到大量Hash相等的情况后，性能不高

### 2.5 密集索引和稀疏索引的区别



**聚簇索引**，就是将索引和数据放到一起，找到索引也就找到了数据，B+树索引就是一种聚簇索引

**非聚簇索引**就是将数据和索引分开，查找时需要先查找到索引，然后通过索引回表找到相应的数据

InnoDB有且只有一个聚簇索引，而MyISAM中都是非聚簇索引。

### 2.6 为什么InnoDB只有一个聚簇索引，而不将所有索引都使用聚簇索引？

因为聚簇索引是将索引和数据都存放在叶子节点中，如果所有的索引都用聚簇索引，则每一个索引都将保存一份数据，会造成数据的冗余，在数据量很大的情况下，这种数据冗余是很消耗资源的

### 2.7 索引是越多越好吗

数据量小的表不需要建立索引，建立会额外增加索引开销

数据变更需要维护索引，更多的索引意味着更多的维护成本

更多的索引也意味着需要更多的空间

### 2.8 MyISAM和InnoDB关于锁方面的区别

MyISAM默认使用表级锁，不支持行级锁

InnoDB默认使用行级锁，也支持表级锁

MyISAM在读取数据时会给整张表加上一个读锁，读取未完成时，另一个Session进行增删改时会给表加上一个写锁，因为读锁未释放，所以写锁必须等待

显式加上锁：lock tables xxx read | write

释放锁：unlock tables

**读锁也叫共享锁（lock in share mode），写锁也叫排他锁（for update）**

先上共享锁，可以再上共享锁，不能上排他锁

先上排他锁，不可以再加任何锁

InooDB不走索引的时候使用的是表级锁，走索引的时候用的是行级锁

### 2.9 MyISAM和InnoDB适合的场景

**MyISAM**

频繁执行全表count语句

对数据查询频繁，增删改频率高

不需要事务

**InnoDB**

数据增删改查都相当频繁

可靠性要求比较高，要求支持事务

### 2.10 数据库事务的四大特性

ACID

* **原子性**（Atomic）
* **一致性**（Consistency）
* **隔离性**（Isolation）
* **持久性**（Durability）

### 2.11 事务并发访问引起的问题

**丢失更新**：一个事务的更新覆盖了另一个事务的更新

eg：两个事务同时访问一个账户，Session1查询余额为100，Session2查询余额为100并存入20，余额变为120并提交事务，Session1取出了10元，但是回滚事务，余额变为100，Session2丢失更新

**脏读**：一个事务读取到另一个事务未提交的数据

eg：两个事务同时访问同一账户，Session1查询余额未1000，取出100后，查询余额为900，

然后Session2进行查询余额为900，Session1由于原因回滚事务，实际余额还为1000，但是Session2并不知道，还在900上操作

**不可重复读**：事务A多次读取同一数据，事务B在事务A多次读取的过程中对数据进行了更新并提交，导致事务A多次读取数据不一致

**幻读**：事务A读取与搜索条件匹配的若干行，事务B已插入或删除修改事务A的结果集

eg：Session1和Session2同时开启事务，Session1查询出全表共有3条数据，Session2新增一条数据并提交，Session1将所有数据更新，却发现跟新成功了4条

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 事务隔离级别 | 丢失更新 | 脏读 | 不可重复读 | 幻读 |
| 未提交读 (Read Uncommitted) | **×** | **√** | **√** | **√** |
| 已提交读 (Read Committed) | **×** | **×** | **√** | **√** |
| 可重复读 (Repeated Read) | **×** | **×** | **×** | **√** |
| 串 行 化 (Serializable) | **×** | **×** | **×** | **×** |

实际上MYSQL的 Repeated Read也可以避免幻读

### 2.12 什么情况索引会失效

1、在索引列上做任何的操作（计算、函数、类型转换），会导致索引失效而转向全表扫描

2、组合索引中，如果中间某个字段使用了范围条件，则右边的列索引失效

3、mysql在使用不等于（！= 或者<>）的时候，无法使用索引列会导致全表扫描

4、is null ，is not null 也无法使用索引

5、like通配符必须放在索引列的右边，否则索引失效，编程全表扫描

6、字符串不加单引号索引失效

7、少用or，用它连接索引会失效

**2.13 数据库三范式**

1NF：列不可再分，一列只能存储一个数据，不能再次拆分，强调原子性

2NF：不可以把多种数据保存在同一张表中，即一张表只能描述一种数据，强调唯一性。

3NF：消除字段冗余。

# JVM

### 3.1 Java如何实现平台无关性

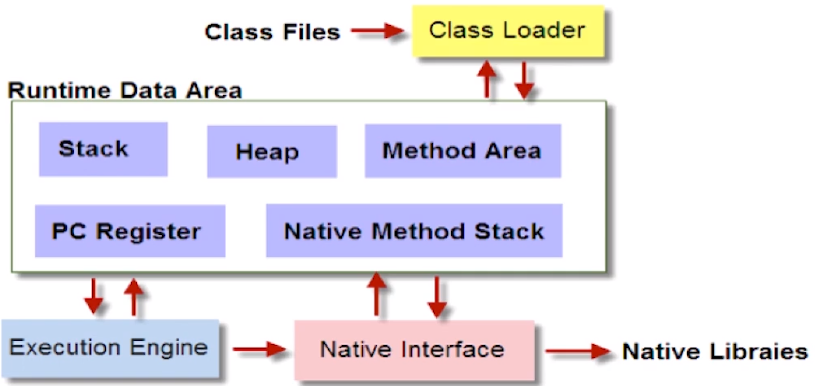
Java源码首先会被编译成字节码，再由不同平台的jvm进行解析java语言在不同平台运行时不需重新编译，虚拟机在执行字节码的时候，会把字节码转换成具体平台上的机器指令

### 3.2 为什么jvm不直接将源码解析成机器码去执行

检查工作：每次执行时需要进行语法，句法的检查，需要重新编译

兼容性：也可以由其他语言生成字节码，在jvm中运行，如ruby

### 3.3 Jvm组成部分



Class Loader：依据特定格式加载class文件

Execution Engine：对命令进行解析

Native Interface：融合不同开发语言的库为Java所用

Runtime Data Area：jvm内存空间结构模型

### 3.4 什么是反射

Java反射机制是指在运行状态中，动态的调取类的信息，以及动态调用类的方法的功能称为java反射

### 3.5 写一个反射的例子

public class Robot {  
 private String name;  
 private void sayHello(String name){  
 System.*out*.println("hello my name is "+name);  
 }  
}

public class RobotTest {  
 public static void main(String[] args) throws Exception {  
 Class rc = Class.*forName*("test.reflex.Robot");  
 //动态创建对象  
 Robot robot = (Robot) rc.newInstance();  
 Method sayHello = rc.getDeclaredMethod("sayHello",String.class);  
 sayHello.setAccessible(true);  
 sayHello.invoke(robot,"mali");  
 }  
}

### 3.6 类装载的过程

**加载**：通过ClassLoader加载Class文件，生成Class对象

**链接**：1.校验：检查加载class的正确性和安全性

1. 准备：为类变量分配空间并设置初始值（默认值）
2. 解析：将jvm常量池内的符号引用转换为直接引用

**初始化**：执行类变量复制，并执行静态代码块

### 3.7 谈谈ClassLoader

ClassLoader主要工作在Class装载的加载阶段，其作用是从系统外部获得Class二进制数据流，所有的Class都是由ClassLoader加载的，ClassLoader负责将Class文件中的二进制数据装载进系统，然后交给jvm进行连接初始化等操作

### 3.8 ClassLoader的种类

BootStrapClassLoader：C++编写，加载核心库java.\*

ExtClass Loader：java编写，加载扩展库javax.\*

App Class Loader：java编写，加载程序所在目录即classpath

自定义Class Loader：java编写，定制化加载

### 3.9 谈谈ClassLoader的双亲委派机制

1-类加载器收到类加载的请求；

2-把这个请求委托给父加载器去完成，一直向上委托，直到启动类加载器；

3-启动器加载器检查能不能加载（使用findClass()方法），能就加载（结束）；否则，抛出异常，通知子加载器进行加载。

4-重复步骤三

### 3.10 为什么要使用双亲委派机制加载类

避免多份同样的字节码重复加载

### 3.11 类的加载载方式LoadClass和Class.forName区别

**隐式加载**：new()

**显示加载**：loadClass(),Class.forName()

**loadClass(),Class.forName()区别**：

loadClass方法加载的类只完成了加载过程，没有执行链接和初始化

Class.forName方法的到的是已经初始化完成的类

### 3.12 JVM类加载顺序

1.父类静态（静态代码块、静态成员变量），子类静态（静态代码块、静态成员变量）

2.父类非静态（构造代码块、非静态成员变量），父类构造函数

3.子类非静态（构造代码块、非静态成员变量），子类构造函数

### 3.13 JVM内存模型-jdk8

###### 线程私有部分

**程序计数器**：

当前线程所执行字节码得行号指示器（逻辑）

改变计数器得值来选取下一条所需要得字节码指令

每条线程需要独立得程序计数器，和线程一对一，所以是线程私有

**虚拟机栈**

**本地方法栈**

###### 线程共有部分

**元空间**：java8元空间代替了永久代，元空间存储着类文件在jvm运行时数据结构 以及Class相关内容

**堆**：对象实例分配区域，GC管理的主要区域

Jdk7之后字符串常量池从方法区被移动到了堆中

###### metaSpace相比PermGen的优势

元空间使用的是本地内存，永久代使用的是jvm内存

字符串常量池存在永久代中，容易出现性能问题和内存溢出

类和方法信息大小难以确定，给永久代大小指定带来困难

永久代会为GC带来不必要的复杂性

### 3.14 Jvm三大性能调优参数-Xms -Xmx -Xss的含义

-Xss：规定了每个线程虚拟机栈的大小，一般256k足够

-Xms：规定了初始java堆的大小

-Xmx：堆能达到最大值

-Xmn: 新生代内存大小

通常-Xms和-Xmx设置为一样，因为堆扩容时影响程序的稳定性

### 3.15 Java内存模型中栈和堆的区别

管理方式：栈自动释放，堆GC释放

空间大小：堆远大于栈

内存碎片：堆产生的内存碎片远大于堆

分配相关：栈支持动态分配内存和静态分配内存，堆仅支持动态分配

效率：栈的效率远大于堆

### 3.16 JVM常见的内存溢出问题

1. java.lang.OutOfMemoryError: Java heap space ----JVM Heap（堆）溢出

JVM在启动的时候会自动设置JVM Heap的值，其初始空间(即-Xms)是物理内存的 1/64，最大空间(-Xmx)不可超过物理内存。可以利用JVM提供的-Xmn -Xms -Xmx等选项可进行设置。Heap的大小是Young Generation 和Tenured Generaion 之和。

在JVM中如果98％的时间是用于GC，且可用的Heap size 不足2％的时候将抛出 此异常信息。

1. java.lang.OutOfMemoryError: PermGen space ---- PermGen space溢出

PermGen space的全称是Permanent Generation space，是指内存的永久保存区域。

为什么会内存溢出，这是由于这块内存主要是被JVM存放Class和Meta信息的， Class在被Load的时候被放入PermGen space区域，它和存放Instance的Heap区 域不同,sun的 GC不会在主程序运行期对PermGen space进行清理，所以如果你的 APP会载入很多CLASS的话，就很可能出现PermGen space溢出。

1. java.lang.StackOverflowError ---- 栈溢出

通常是程序错误，比如递归太多层数

# 4.GC

### 4.1 判断对象是否为垃圾的算法

引用计数法：如果一个对象没有任何引用与之相关联，那么这个对象就可能成为被回收对象，但是这种方法无法解决循环引用

可达性分析法：java中采用这种方法，通过一系列的GC Roots对象作为起点进行搜索，如果GC Roots和一个对象没有可达路径，则这个对象在经过至少两次的标记后就可能被回收

### 4.2 垃圾回收算法

**标记-清除算法（比较适合老年代）**

这种方式容易产生内存碎片，碎片太多导致在为大对象分配内存空间时内存不够而提前触发下一次GC操作

**复制算法（比较适合年轻代）**

将内存按容量分为等量的两块，每次使用其中的一块，当一块的内存用完之后，就将存活的对象放到另一块内存之上，一次性清掉以用内存

优点：这中方式不容易产生内存碎片，顺序分配内存简单高效，适用对象存货率低的场景

缺点：浪费内存空间，如果存活对象过多，那么复制算法的效率大大降低

**标记-整理算法（比较适合老年代）**

为了解决复制算法的缺陷，标记整理和标记清除一样，但是在标记完成之后是将存活的对象移动到内存的一端，然后清掉端边界以外的内存

**分代收集算法**

根据对象的存活周期，将堆区划分为新生代和老年代

新生代(1/3堆空间)：Minor GC是发生在新生代的垃圾收集动作，采用复制算法

新生代分为3块(Eden(8/10)、From Survivor(1/10)、To Survivor(1/10))

老年代(2/3堆空间)：Full GC是发生在老年代的垃圾收集动作，采用的标记-整理算法

对象出生在Eden或者是其中一个Survivor区（假设是From）经过一次Minor GC之后，若对象还存活并且能被另一块Survivor（To）所容纳，那么使用复制算法将Eden 喝 From Survivor中存活的对象复制到另一块Survivor中（To），并将他们的年龄设置位1，以后对象在Survivor中每熬过一次Minor GC他的年龄就会加1，当年龄到达15（默认），这些对象就会成为老年代

### 4.3 触发Full GC的条件

老年代空间不足

Jdk7及以前版本，永久代空间不足

Minor GC晋升到老年代的大小大于老年代的剩余空间

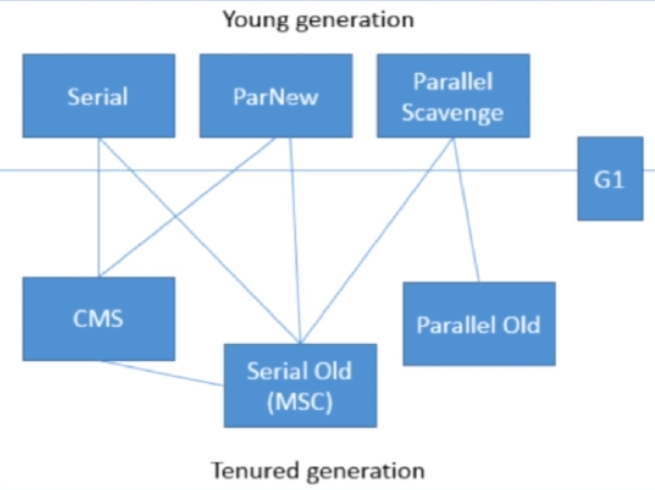
调用System.gc(),只是提醒

使用RMI来进行RPC或管理的jdk应用，每小时执行一次Full GC

### 4.4垃圾收集器的种类

Jvm两种运行模式：server;client

**垃圾收集器之间的关系**：



**新生代**：

**Serial收集器**：（-XX:+UseSerialGC）

使用复制算法，单线程运行，client 模式新生代默认收集器，收集时，暂停其他所有工作线 程

**ParNew收集器**：(-XX:+UseParNewGC)

使用复制算法，多线程运行，其他行为特点与Serial一样, server模式虚拟机首选新生 代垃圾收集器，在单核cpu情况下，并不会比Serial有更好效果，默认开启线程数与cpu 数量相同

**Parallel Scavenge收集器**：(-XX:+UseParallelGC)

使用复制算法，多线程运行，比起关注用户线程停顿时间，更关注吞吐量，server模式默认 垃圾垃圾收集器

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）

**老年代**：

**Serial Old收集器**：（-XX:+UseSerialOldGC）

使用标记-整理算法，单线程运行，client模式默认老年代收集器

**CMS收集器**：(-XX:+UseConcMarkSweepGC)

使用标记-清除算法，由于使用标记清除算法，容易产生内存碎片，当产生一个大对象的时候， 没有连续的内存空间，只能提前 触发GC

**Parallel Old收集器**：(-XX:+UseParallelOldGC)

使用标记-整理算法，多线程运行，吞吐量优先

**Garbage First收集器**：(-XX:+UseG1GC)

能用于新生代和老年代的收集器,复制+标记-整理算法

将整个java堆内存划分为多个大小相等的Region，新生代和老年代不在物理隔离

### 4.5 Object中的finalize()方法的作用是否与C++的构析函数相同

1. 与C++中的构析函数不同，析构函数调用时机确定，即对象离开作用域后就会被删除掉，而 finalize()的调用不确定性

2. 如果对象覆盖finalize()方法，在经过两次垃圾标记后，未被引用的对象会放置于F-Queue队列，等待一个低优先级的线程执行finalize()方法

3. 方法执行随时可能终止

4. 给与对象最后一次重生的机会

### 4.6 强引用，软引用，弱引用，虚引用有什么作用

# 5. Java多线程与并发

### 5.1 进程和线程的区别

进程是资源分配的最小单位，线程是cpu调度的最小单位

线程不能看做独立应用，而进程可看做独立应用

进程有独立地址空间，线程没有独立的地址空间

进程的切换比线程的切换开销大

### 5.2 Thread中的start和run方法的区别

调用start()方法会创建一个新的线程并启动

Run()方法知识Thread的一个普通方法调用

### 5.3 Thread和Runnable的关系

Thread是实现了Runnable接口的类，使run方法支持多线程

因为单一继承原则，推荐使用Runnable接口

### 5.4 如何给run()方法传参

构造函数传参

成员变量传参

回调函数传参

### 5.5 如何实现处理线程的返回值

**主线程等待法**

**使用Thread类中的join方法阻塞当前线程等待子线程执行完毕**

public class CycleWait implements Runnable {  
 private String value;  
 @Override  
 public void run() {  
 try {  
 Thread.*currentThread*().*sleep*(3000);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 value = "hello world";  
 }  
 public static void main(String[] args) throws InterruptedException {  
 CycleWait cycleWait = new CycleWait();  
 Thread t = new Thread(cycleWait);  
 t.start();  
 //主线程等待法  
 while (cycleWait.value == null) {  
 Thread.*currentThread*().*sleep*(100);  
 }  
 //join  
 t.join();  
 System.*out*.println(cycleWait.value);  
 }  
}

**使用Callable接口，通过FutureTask或线程池获取**

public class CallableTest implements Callable<String> {  
 @Override  
 public String call() throws Exception {  
 String str = "hello world";  
 Thread.*currentThread*().*sleep*(3000);  
 return str;  
 }  
 public static void main(String[] args) throws ExecutionException, InterruptedException {  
 *futureTaskTest*();  
 *threadPoolTest*();  
 }  
 //使用FutureTask接收线程的返回值  
 public static void futureTaskTest() throws ExecutionException, InterruptedException {  
 FutureTask task = new FutureTask(new CallableTest());  
 Thread thread = new Thread(task);  
 thread.start();  
 if (!task.isDone()) {  
 System.*out*.println("未执行完成");  
 }  
 System.*out*.println(task.get());  
 }  
 //使用线程池接收返回值  
 public static void threadPoolTest() throws ExecutionException, InterruptedException {  
 ExecutorService es = Executors.*newCachedThreadPool*();  
 Future<String> future = es.submit(new CallableTest());  
 if (!future.isDone()) {  
 System.*out*.println("未执行完成");  
 }  
 System.*out*.println(future.get());  
 }  
}

### 5.6 线程的状态

6个状态

**新建(New)**:创建后尚未启动的线程状态

**运行(Runnable)**:包含Running和Ready，可能正在执行，也可能等待cpu分配时间

**无限期等待(Waiting)**:不会被分配cpu执行时间，需要显示唤醒

没有设置Timeout参数的Object.wait()方法

没有设置Timeout参数的Thread.join()方法

LockSuppot.pack()方法

**限期等待(Timed Waiting)**:在一定时间后会有系统自己唤醒

Thread.sleep()方法

设置Timeout参数的Object.wait()方法

设置Timeout参数的Thread.join()方法

LockSuppot.packNanos()方法，LockSuppot.packUntil()方法

**阻塞(Block)**:等待获取排他锁

**结束(Terminated)**:线程已结束

### 5.7 sleep和wait的区别

sleep是Thread中的方法，wait是Object中定义的方法

sleep可以在任何地方使用

wait只能在synchronized方法或synchronized块中使用

sleep只会让出cpu，不会释放锁，wait不仅让出cpu，还会释放锁

### 5.8 锁池和等待池

**锁池**：

假设线程A已经拥有了某个对象（不是类）的锁，而其他现线程B、C想调用这个对象的某个synchrronized方法（或者块），由于B、C线程进入对象的synchronized方法（或者块）之前必须获得该对象锁的拥有权，而恰巧该对象的锁目前正在被线程A占用，此时B、C线程就会被阻塞，进去一个贷方去等待锁的释放，这个地方便是该对象的锁池

**等待池**：

假设线程A调用了某个对象的wait（）方法，线程A就会释放该对象的锁，同时线程A就会进去该对象的等待池中，进去等待池中的对象不会去竞争该对象的锁

### 5.9 notify和notifyAll的区别

notifyAll会让所有处于等待池的线程全部进入锁池去竞争获取锁的机会

notify只会随机选取一个处于等待池中的线程进入锁池去竞争获取锁的机会

### 5.10 yield

Thread.yield()函数被调用时，会给线程调度器一个当前线程愿意让出cpu使用的暗示，但是线程调度器可能会忽略这个暗示

### 5.11 如何中断线程

调用interrupt()函数，通知线程该中断了

如果线程处于阻塞状态，那么线程将立即退出被阻塞状态，并抛出一个InterruptedException

如果线程处于正常活动状态，那么会将该线程的中断标记设置未true，被设置中断标志的线程将继续正常运行

public static void main(String[] args) throws InterruptedException {  
 Thread t = new Thread(new Runnable() {  
 @Override  
 public void run() {  
 try {  
 while (!Thread.*currentThread*().isInterrupted()) {  
 System.*out*.println("===");  
 Thread.*sleep*(1000);  
 }  
 } catch (InterruptedException e) {  
 System.*out*.println(Thread.*currentThread*().getState());  
 }  
 }  
 });  
 t.start();  
 Thread.*currentThread*().*sleep*(3000);  
 t.interrupt();  
}

# 6 并发-原理

### 6.1 synchronized介绍

互斥锁的特性

**互斥性**：即在同一时间只允许一个线程持有某个对象锁，通过这种特性来实现多线程的协调机制，这样在同一时间只有一个线程对同步的代码块进行访问操作，互斥性也称未原子性

**可见性**：必须确保在锁被释放之前，对共享变量所做的修改，对于随后获得该锁的另一个线程是可见的，即在或的锁时应获得最新的共享变量值，否则另一个线程可能是在本地缓存的副本上进行操作，从而引起不一致

Synchronized锁的不是代码，锁的是对象

**总结**：

1. 有线程访问对象的同步代码块时，另外的线程可以访问该对象的非同步代码块
2. 若锁住的是同一个对象，一个线程访问对象的同步代码块时，另一个访问该同步代码块的线程被阻塞
3. 若锁住的是同一个对象，一个线程访问对象的同步方法时，另一个访问该同步方法的线程被阻塞
4. 若锁住的是同一个对象，一个线程访问对象的同步代码块时，另一个访问对象同步方法的线程会被阻塞，反之亦然
5. 同一个类不同对象的对象锁互不干扰
6. 类锁是特殊的对象锁，表现和1、2、3、4一致，而由于一个类只有一把类锁，所以同一个类的不同对象使用类锁将会是同步的
7. 类锁和对象锁互不干扰

### 6.2 什么是重入

当一个线程试图操作一个由其他线程持有的对象锁的临界资源时，将会处于阻塞状态，但是当一个线程再次请求自己持有对象锁的临界资源时，是允许的，这种情况属于重入

### 6.3 自旋锁与自适应自旋锁

**自旋锁**：

许多情况下，共享数据的锁定状态持续时间较短，切换线程不值得

通过让线程执行忙循环等待锁释放，但是又不像sleep()一样放弃cpu执行时间

若锁被其他线程占用时间较长，会带来许多性能开销

**自适应自旋锁**：(jdk6)

自旋次数不在固定

由前一次在同一个锁上自旋时间及锁的拥有者的状态来决定，如果在同一个锁对象上，自旋等待成功获取到了锁，并且持有锁的线程正在运行，jvm会认为该锁自旋获取成功的可能性很大，会自动增加等待时间，如果某个锁自旋很少成功，那以后将可能省略自旋过程，以免浪费cpu资源

### 6.4 锁消除和锁粗化

**锁消除**：

JIT编译时，对运行上下文进行扫描，去除不可能存在竞争的锁

public static void test(String s1,String s2){  
 StringBuffer stringBuffer = new StringBuffer();  
 stringBuffer.append(s1).append(s2);  
}

不同线程调用StringBuffer的append方法是，由于stringBuffer对象属于本地变量，只会在test方法中使用，所以jvm会自动消除append方法中的锁，为了提升效率

**锁粗化**：

通过扩大加锁范围，避免反复加锁解锁

### 6.5 synchronized的四种状态

无锁、偏向锁、轻量级锁、重量级锁

**无锁**：此时目标共享数据没有被任何线程占有

**偏向锁**：大多数情况下，锁不存在多线程竞争，总是由同一线程多次获得

核心思想：如果一个线程获得了锁，会在对象头和栈帧的所记录里存储锁偏向的线程ID，

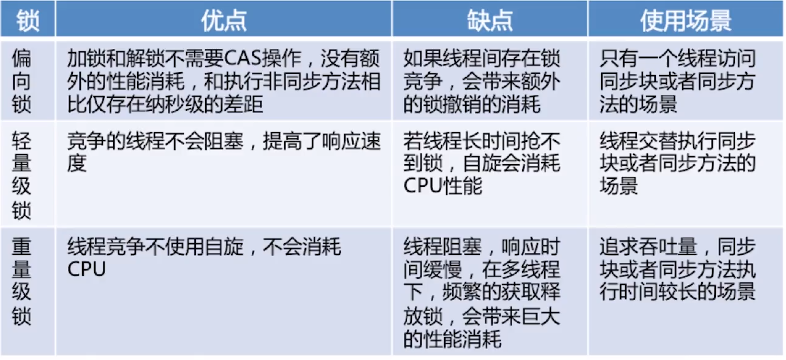
以后该线程在进入和退出同步块操作时，不需要进行CAS操作加锁解锁

不适用锁竞争比较激烈的多线程场合

**轻量级锁**：偏向锁运行在一个线程进入同步块的情况下，当第二个线程加入锁争用的时候，偏向锁就会升级未轻量级锁

适用场景：线程交替执行同步块

如果存在同一时间多个线程访问同一个锁的情况，就会导致轻量级锁膨胀未重量级锁



### 6.6 synchronized和ReentrantLock的区别

ReentrantLock：

位于java.util.concurrent.locks包

和CountDownLatch、FutureTask、Semaphore一样基于AQS实现

能够实现比synchronized更细粒度的控制，如控制公平性

调用lock()后，必须调用unlock()释放锁

性能未必比synchronized高，并且也是可重入的

ReentrantLock公平性设置

ReentrantLock lock = new ReentrantLock(true);//慎用

参数未true时，倾向将锁赋予等待时间最久的线程

synchronized 是非公平锁

**区别**：

synchronized是关键字，ReentrantLock是类

ReentrantLock可以对获取锁的等待时间进行设置，避免死锁

ReentrantLock可以获取各种锁信息

机制：sync操作Mark Word，lock调用Unsafe类的park()方法

## 6.7 volatile和synchronized的区别

1.volatile本质是告诉jvm当前变量在寄存器（工作内存）中的值是不确定的，需要从主存中读取；synchronized则是锁定当前变量，只有当前线程可以访问该变量，其他线程被阻塞知道该线程完成变量操作为止

2. volatile仅能适用在变量级别；synchronized可以在变量、方法、和类级别使用

3.volatile仅能保证变量修改的可见性，不能保证原子性；synchronized则可以保证变量修改的原子性和可见性

4.volatile不会造成线程阻塞；synchronized可能造成线程阻塞

5.volatile标记的变量不会被编译器优化；synchronized编辑的变量可以被编译器优化

## 6.8 CAS(Compare and Swap)

一种高效实现线程安全的方法

* 支持原子更新操作，适用与计数器，序列发生器等
* 属于乐观锁机制，号称lock-free
* CAS操作失败是由开发者决定继续尝试，还是执行其他操作
* JUC的atomic包提供了常用的原子性数据类型以及引用、数组等相关原子类型和更新操作工具，是很多线程安全程序的首选
* Unsafe类索然提供了CAS服务，但因能够操作任意内存地址读写而存在隐患

缺点：

* 若循环时间长，开销很大
* 只能共享变量的原子操作

### 6.9 线程池

利用Executors创建不同的线程池满足不同场景的需求

1.newFixedThreadPool(int nThreads)

指定工作线程数量的线程池

2.newCachedThreadPool()

处理大量短时间工作任务的线程池

(1)试图缓存线程并重用，党务缓存线程可用就会创建新的工作线程

(2)如果线程闲置时间超过阈值，则会被终止并移出缓存

(3)系统长时间闲置的时候，不会消耗什么资源

3. newSingleThreadExecutor()

创建唯一的工作线程来执行任务，线程如果异常结束，会有另一个线程取代它

这个比较适合需要保证队列中任务顺序执行的场景

4.newSingleThreadScheduledExecutor()和newScheduledThreadPool(int corePoolSize)

定时或者周期工作调度，浪者区别是单一工作线程还是多个线程

5.newWorkStealingPool()--jdk8

内部构件ForkJoinPool，利用working-stealing算法，并行执行任务，不保证处理顺序

**使用线程池的好处**：

降低资源消耗

提高线程可管理性

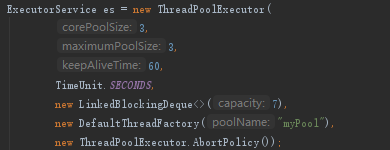
**创建线程池**：

方式1：

ExecutorService executorService =Executors.*newFixedThreadPool*(3);

*\* 使用该种方式创建线程池容易造成OOM（out of memory）  
\* 使用该种方式 LinkedBlockingDeque的默认容量为无限制*

方式2：



参数：

corePoolSize:核心线程数量

maximumPoolSize:线程不够时能够创建最大线程数量

keepAliveTime：当池中线程数大于核心线程数时，该时间为余下线程（存活线程总数-核心线程数）的最大空闲存活时间

unit：keepAliveTime时间单位

workQueue:任务等待队列

ThreadFactory：创建线程的线程工厂，使用DefaultThreadFactory创建的线程具有相同的优先级

Handler：线程池饱和策略

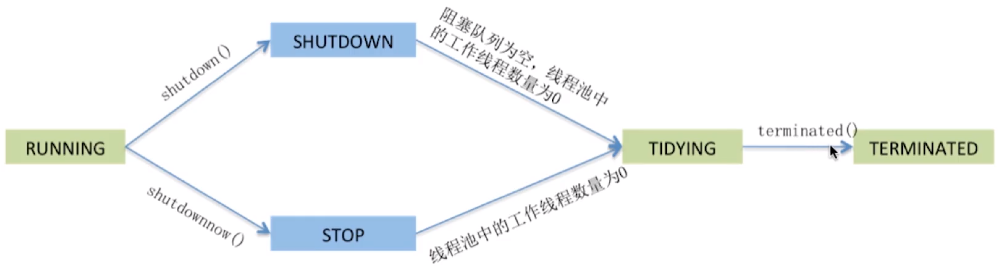
AbortPolicy：直接抛出异常，也是默认的拒绝策略。

CallerRunsPolicy：直接在调用线程中执行该任务。

DiscardPolicy：直接将该任务丢弃。

DiscardOldestPolicy：丢弃最老的未被执行的任务，并执行当前任务

### 6.10 线程池的状态

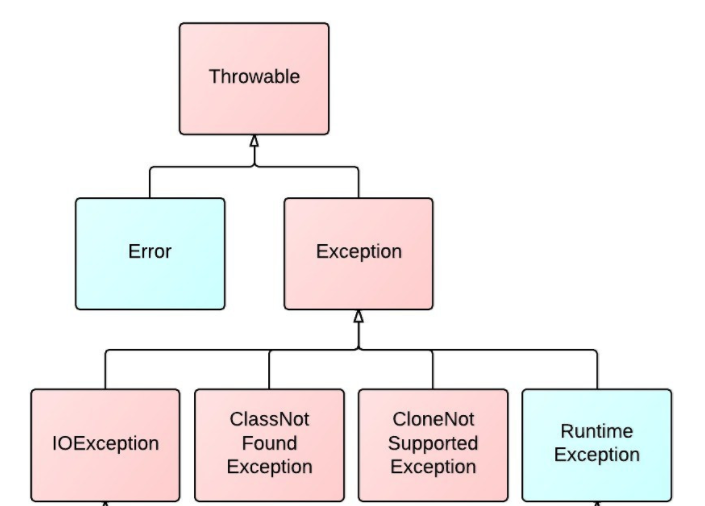


* Running: 能够接收新任务，并且也能处理阻塞队列中的任务
* Shutdown:不再接收新任务，但可以处理存量任务
* Stop:不在接受新任务，也不处理存量任务
* Tidying:所有任务都终止
* Terminated：terminated()方法执行完后进入该状态

# 7.java常用类库

### 7.1 String，StringBuffer，StringBuilder的区别

### 7.2 java异常架构



**Error和Exception的区别**

Error：程序无法处理的系统错误，编译器不做检查

Exception：程序可以处理的异常，捕获后可能恢复

**CheckedException和UnCheckedException的区别**

CheckedException：图中粉红色的是受检查的异常,其必须被 try{}catch语句块所捕获,或者在方法签名里通过throws子句声明.受检查的异常必须在编译时被捕捉处理,命名为 Checked Exception 是因为Java编译器要进行检查,Java虚拟机也要进行检查,以确保这个规则得到遵守.

UnCheckedException：也就是RuntimeException运行时异常

**常见的RuntimeException**

NullPointerException - 空指针引用异常

ClassCastException - 类型强制转换异常。

IllegalArgumentException - 传递非法参数异常。

ArithmeticException - 算术运算异常

ArrayStoreException - 向数组中存放与声明类型不兼容对象异常

IndexOutOfBoundsException - 下标越界异常

NegativeArraySizeException - 创建一个大小为负数的数组错误异常

NumberFormatException - 数字格式异常

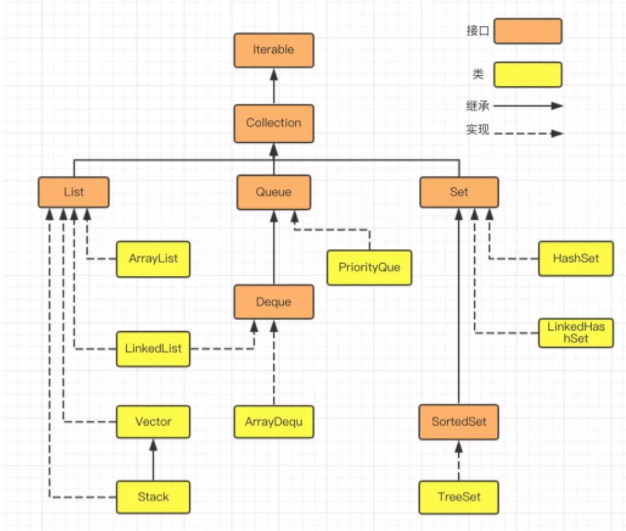
**Finally执行顺序**

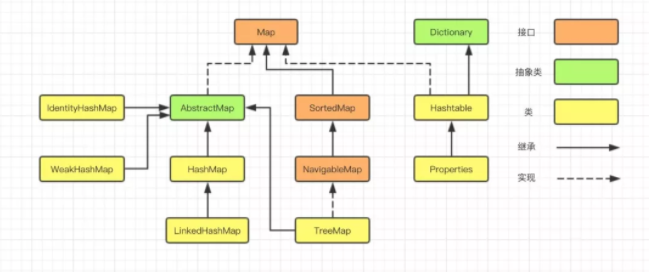
Finnally优先于return语句执行

**try{}catch()的效率并不如if()**

### 7.3 java集合框架

##### 7.3.1集合框架图





##### 7.3.2 List

**ArrayList**是一个动态数组结构，支持随机存取，尾部插入删除方便，内部插入删除效率低（因为要移动数组元素）；如果内部数组容量不足则自动扩容，因此当数组很大时，效率较低。

**LinkedList**是一个双向链表结构，在任意位置插入删除都很方便，但是不支持随机取值，每次都只能从一端开始遍历，直到找到查询的对象，然后返回；不过，它不像 ArrayList 那样需要进行内存拷贝，因此相对来说效率较高，但是因为存在额外的前驱和后继节点指针，因此占用的内存比 ArrayList 多一些。

**Vector**也是一个动态数组结构，jdk1.1就引入进来的类，之后在jdk1.2里引进ArrayList，ArrayList大部分的方法和Vector比较相似，两者是不同的，Vector是允许同步访问的，Vector中的操作是线程安全的，但是效率低，而ArrayList所有的操作都是异步的，执行效率高，但不安全！

Vector，现在用的很少了，因为里面的get、set、add等方法都加了synchronized，所以，执行效率会比较低，如果需要在多线程中使用，可以采用下面语句创建ArrayList对象

List<Object> list =Collections.synchronizedList(new ArrayList<Object>());

也可以考虑使用复制容器java.util.concurrent.CopyOnWriteArrayList进行操作，例如：

Final CopyOnWriteArrayList<Object> cowList = new CopyOnWriteArrayList<>(Object);

#### **Stack**是Vector的一个子类，本质也是一个动态数组结构，不同的是，它的数据结构是先进后出，取名叫栈！

关于Stack，现在用的也很少，因为有个ArrayDeque双端队列，可以替代Stack所有的功能，并且执行效率比它高！

##### 7.3.3 Map

**HashMap** 继承自AbstractMap，key 不可重复，因为使用的是哈希表存储元素，所以输入的数据与输出的数据，顺序基本不一致，另外，HashMap的key、value均可为null，但是只能有一个key为null

java8以前用数组+链表实现,如果极端情况数据key的hashcode都一样，性能就会恶化O(1)->O(n)

Java8之后用数组+链表+红黑树实现，性能最差情况未O(log n)

默认容量16，装载因子0.75，扩容为以前2倍

当链表大小大于8并且整个hashmap的元素大于64就会由链表改造成为红黑树

当链表大小小于6就会由红黑树转为链表

**HashTable** 早期提供的哈希表的实现，线程安全串行执行，性能较差，key、value不能为null

**ConcurrentHashMap**

早期的ConcurrentHashMap由分段锁实现

当前的ConcurrentHashMap由CAS+synchronized使锁更细化

ConcurrentHashMap不允许null键

put方法执行时，通过hash定位数组的索引坐标，是否存在Node节点，如果没有则用CAS进行添加（链表头节点），如果添加失败则进入下一次循环，如果头节点不为空，则尝试获取头节点的同步锁，在进行操作

**LinkedHashMap**

HashMap 的子类，内部使用链表数据结构来记录插入的顺序，使得输入的记录顺序和输出的记录顺序是相同的。LinkedHashMap与HashMap最大的不同处在于，LinkedHashMap输入的记录和输出

的记录顺序是相同的

**TreeMap**能够把它保存的记录根据键排序，默认是按键值的升序排序，也可以指定排序的比较器，当用 Iterator 遍历时，得到的记录是排过序的；如需使用排序的映射，建议使用 TreeMap

##### 7.3.4 Set

#### **HashSet** 底层是基于 HashMap 的k实现的，元素不可重复，特性同 HashMap

#### **LinkedHashSet**底层也是基于 LinkedHashMap 的k实现的，一样元素不可重复，特性同 LinkedHashMap

#### **TreeSet**也是基于 TreeMap 的k实现的，同样元素不可重复，特性同 TreeMap

Set集合的实现，基本都是基于Map中的键做文章，使用Map中键不能重复、无序的特性

##### 7.3.5 hashcode 作用

对象的散列值，Object类中的方法，

像Hash开头的类，HashMap、HashSet、Hash等等

哈希代码值可以提高性能

实现了hashCode一定要实现equals，因为底层HashMap就是通过这2个方法判断重复对象的，先判断key的hashCode是否相等，相等进行equals判断，equals为true覆盖，不为true以链表的形式插入

如果两对象equals()是true,那么它们的hashCode()值一定相等

如果两对象的hashCode()值相等，它们的equals不一定相等（hash冲突）

##### 7.3.6 Arraylist扩容方式

ArrayList默认容量10

Jdk1.6扩容算法 新容量=旧容量\*1.5+1

Jdk1.8扩容算法 新容量=旧容量\*1.5

扩容后检查当前容量是否大于需求量，若扩容后的容量还是小于需求量，那么就把需求量当做数组的新容量,然后调用Arrays.copyOf()复制

Arrays.copyOf()不仅仅只是拷贝数组中的元素，在拷贝元素时，会创建一个新的数组对 象。而System.arrayCopy只拷贝已经存在数组元素。

Arrays.copyOf()该方法的底层还是调用了System.arrayCopyOf()方法。

System.arrayCopy如果改变目标数组的值原数组的值也会随之改变。

**7.3.7 final的四种用法**

**7.3.8 序列化是什么，底层怎么实现**

# Java IO

**基本概念**

同步异步针对的是被调用者B

**同 步**：A调用B，B的处理是同步的，直到B处理完成之后才会通知A

**异 步**：A调用B，B的处理是异步的，B在接到请求后先告诉A我已经接到请求了，然后异步去处理，处理完之后通过回调等方式再通知A

阻塞非阻塞针对的是调用者A

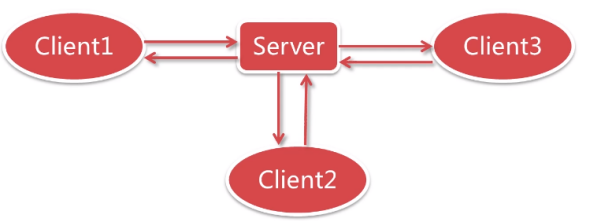
**阻 塞**：A调用B，A一直等待B的返回，期间处于阻塞状态

**非阻塞**：A调用B，A调用成功后不用一直等待B返回，可以去做其他任务

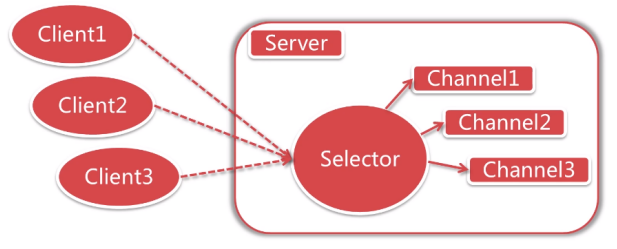
**Java中的3种IO模型**

**BIO** (Blocking I/O)同步阻塞IO

每有一个客户端连接就需要和Server建立一个线程



**NIO** (New I/O)同步非阻塞IO，也支持同步阻塞IO



通道 和 缓冲区 是 NIO 中的核心对象，几乎在每一个 I/O 操作中都要使用它们。

**缓冲区**：Buffer 是一个对象， 它包含一些要写入或者刚读出的数据。 在 NIO 中加入 Buffer 对象，体现了新库与原 I/O 的一个重要区别。在面向流的 I/O 中，数据直接写入或者将数据直接读到 Stream 对象中。

在 NIO 库中，所有数据都是用缓冲区处理的。在读取数据时，它是直接读到缓冲区中的。在写入数据时，它是写入到缓冲区中的。任何时候访问 NIO 中的数据，都是将它放到缓冲区中。

缓冲区实质上是一个数组。通常它是一个字节数组，但是也可以使用其他种类的数组。但是一个缓冲区不 仅仅 是一个数组。缓冲区提供了对数据的结构化访问，而且还可以跟踪系统的读/写进程。

**缓冲区类型**：最常用的缓冲区类型是 ByteBuffer。一个 ByteBuffer 可以在其底层字节数组上进行 get/set 操作(即字节的获取和设置)。

当然也有其他类型：

ByteBuffer、CharBuffer、ShortBuffer、IntBuffer、LongBuffer、FloatBuffer、DoubleBuffer

每一个 Buffer 类都是 Buffer 接口的一个实例。 除了 ByteBuffer，每一个 Buffer 类都有完全一样的操作，只是它们所处理的数据类型不一样。因为大多数标准 I/O 操作都使用 ByteBuffer，所以它具有所有共享的缓冲区操作以及一些特有的操作。

**通道**：Channel是一个对象，可以通过它读取和写入数据。拿 NIO 与原来的 I/O 做个比较，通道就像是流。

所有数据都通过 Buffer 对象来处理。不会将字节直接写入通道中，相反，您是将数据写入包含一个或者多个字节的缓冲区。同样，您不会直接从通道中读取字节，而是将数据从通道读入缓冲区，再从缓冲区获取这个字节。

通道类型：通道与流的不同之处在于通道是双向的。而流只是在一个方向上移动(一个流必须是 InputStream或者 OutputStream 的子类)， 而 通道 可以用于读、写或者同时用于读写。

因为它们是双向的，所以通道可以比流更好地反映底层操作系统的真实情况。特别是在 UNIX 模型中，底层操作系统通道是双向的。

**AIO** (Asynchronous I/O) 异步非阻塞IO

# Spring

### IOC和DI的区别

**IOC**： Inverse of Control反转控制的概念，就是将原本在程序中手动创建UserService对象的控制权，交由S p r i n g框架管理，简单说，就是创建UserService对象控制权被反转到了

Sprin框架

**DI**：Dependency Injection 依赖注入，在Srping创建对象是，动态将以来的对象注入到Bean组件

### BeanFactory 接口和 ApplicationContext 接口有什么区别

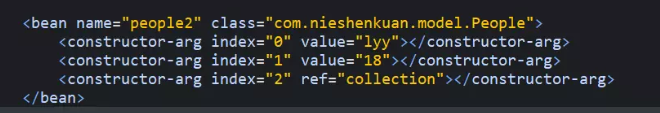
1. ApplicationContext接口继承BeanFactory，Srping的核心工厂是BeanFactory，BeanFactory采用延迟加载，第一次getBean时候才会初始化Bean，ApplicationContext是会在加载配置文件的时候初始化Bean
2. ApplicationContext是对BeanFactory的扩展，它可以进行国际化处理，时间传递和Bean的自动装配以及不同应用层的Context实现

### Spring配置bean实例化有哪些方法

1. 使用空构造器实例化，使用此种方式，class属性指定的类必须有空构造器



使用有参数的构造器进行定义，可以使用<constructor-arg>标签指定构造器参数值，index表示位置，value表示常量值也可以指引用，指定引用使用ref来引用另一个Bean的定义



2.使用静态工厂方式实例化Bean，使用这种方式除了指定必须的class属性，还要指定factory-method属性来指定实例化Bean的方法，而且使用静态工厂方法也允许指定方法参数，spring IoC容器将调用此属性指定的方法来获取Bean



3.使用实例工厂方法实例化Bean，使用这种方式不能指定class属性，此时必须使用factory-bean属性来指定工厂Bean，factory-method属性指定实例化Bean的方法，而且使用实例工厂方法允许指定方法参数，方式和使用构造器方式一样



### spring Bean 的生命周期

### Spring Bean 的作用域

Singleton：Spring只会为每一个bean创建一个实例

Prototype：每一次请求（将其注入到另一个本案中，或执行getBean()方法都会产生一个bean实例）

Request：针对每一个HTTP请求都会产生一个bean，该bean仅在当前request请求内 有效，仅适用于WebApplicationContext环境

Session：针对每一个HTTP请求都会产生一个bean，该bean仅在当前session有效， 仅适用于WebApplicationContex环境

GlobalSession：全局session有效，这个是针对portlet的web应用的，具体可以看看Portlet是如何定义了全局Session的。这个有效范围是全局portlet Session的生命周期范围内，如果不是portlet的web应用，那么这个等同于session作用域

### Spring Bean的注入方式

setter方法注入，构造器注入，注解注入

### Spring如何处理线程并发问题

通过ThreadLocal解决，ThreadLocal会为每个线程提供一个独立的变量副本，从而隔离了多个线程对数据的访问冲突，概括起来对于多线程资源共享问题，同步机制采用了以时间换空间的形式，ThreadLocal采用了以空间换时间的方式

### ThreadLocal实现原理

### SpringAOP几个名词

**连接点**(JoinPoint)：就是spring允许通知Advice的地方，如每个方法的前后，或者抛出异常时等，spring只支持方法的连接点，而AspectJ还可以在构造器或者属性注入时都行

**切入点**(Pointcut)：目标对象,将要和已经 增强的方法

一个类里，有15个方法，那就有十几个连接点，但是并不需要在所有方法都使用通知，只是想让其中几个，在调用这几个方法之前、之后或者抛出异常时干点什么，那么就用切入点来定义这几个方法，让切入点来筛选连接点，选中需要的方法。

**切面**(Aspect)：切入点+通知

**通知**(Advice)：

通知类型：

前置通知@Before，方法执行之前执行

后置通知@After，方法执行之后执行，无论是否发生异常

返回通知@AfterReturning，犯法正常结束后执行

异常通知@AfterThrowing，方法抛出异常后执行

环绕通知@Around，围绕方法执行



### AOP的应用场景

场景一： 记录日志

场景二： 监控方法运行时间 （监控性能）

场景三： 权限控制

场景四： 缓存优化 （第一次调用查询数据库，将查询结果放入内存对象， 第二次调用， 直接从内存对象返回，不需要查询数据库 ）

场景五： 事务管理 （调用方法前开启事务， 调用方法后提交关闭事务 ）

### AOP实现原理

动态代理：

Jdk代理：又叫接口代理

Cglib代理：又叫子类代理

### Spring事务传播行为

|  |  |
| --- | --- |
| PROPAGATION\_REQUIRED | 如果当前没有事务，就新建一个事务，如果已经存在一个事务中，加入到这个事务中。这是最常见的选择。 |
| PROPAGATION\_SUPPORTS | 支持当前事务，如果当前没有事务，就以非事务方式执行。 |
| PROPAGATION\_MANDATORY | 使用当前的事务，如果当前没有事务，就抛出异常。 |
| PROPAGATION\_REQUIRES\_NEW | 新建事务，如果当前存在事务，把当前事务挂起。 |
| PROPAGATION\_NOT\_SUPPORTED | 以非事务方式执行操作，如果当前存在事务，就把当前事务挂起。 |
| PROPAGATION\_NEVER | 以非事务方式执行，如果当前存在事务，则抛出异常。 |
| PROPAGATION\_NESTED | 如果当前存在事务，则在嵌套事务内执行。如果当前没有事务，则执行与PROPAGATION\_REQUIRED类似的操作。 |

### @Transaction注解什么情况会失效

1. 在非public方法中
2. 抛出的异常为checkedException，可以用rollbackFor来指定类型
3. 异常提前被catch掉

# 综合问题

### tomcat集群怎么保证同步

### 怎么解决超卖问题

### mybaits二级缓存

Mybatis的一级缓存共享范围就是sqlSession内部，如果多个sqlsession需要共享缓存，则需要开启mybatis的二级缓存，二级缓存的区域是根据mapper的namespace划分的，如果两个mapper的namespace一样，那么他们就共享一个mapper缓存

开启方式：mybatis配置文件中配置cacheEnable=true同时在mapper中添加<cache>标签，并且pojo必须实现序列化接口

<cache>标签的属性：

eviction：缓存回收策略

Lru：最近最少使用对象回收

Fifo：先进先出

Soft：软引用先回收

Weak：弱引用先回收

flushinterval：缓存刷新间隔，默认不清空

readonly：是否只读，mybatis认为从缓存读取操作都是只读的，不会修改数据

size：缓存存放元素个数

type：自定义缓存的全类名

blocking：若缓存中不存在某个key，是否一直blocking，直到有数据存入

当然在同一个namespace中如果有insert、update、delete则需要更新缓存

<insert id="insertUser" parameterType="User" flushCache="true">