目录

[线程相关 5](#_Toc488413032)

[Java线程安全（并发）的实现方式？ 5](#_Toc488413033)

[线程状态转换 5](#_Toc488413034)

[并发依赖特性 6](#_Toc488413035)

[线程池 6](#_Toc488413036)

[线程池种类 6](#_Toc488413037)

[影响单台服务器并发量的因素有哪些 7](#_Toc488413038)

[I/O密集型和CPU密集型的任务线程开多少合适？ 7](#_Toc488413039)

[blocked和waiting的区别 7](#_Toc488413040)

[Java内存模型 7](#_Toc488413041)

[内存屏障 7](#_Toc488413042)

[happens-before 8](#_Toc488413043)

[synchronized关键字 8](#_Toc488413044)

[原理 8](#_Toc488413045)

[锁(Lock) 8](#_Toc488413046)

[AQS 8](#_Toc488413047)

[Volatile关键字 9](#_Toc488413048)

[JAVA API相关 9](#_Toc488413049)

[java.util.concurrent.atomic包 9](#_Toc488413050)

[fork/join结构 10](#_Toc488413051)

[JAVA IO 11](#_Toc488413052)

[在有return的情况下try catch finally的执行顺序 11](#_Toc488413053)

[error和exception的区别 12](#_Toc488413054)

[checked异常和unchecked异常 12](#_Toc488413055)

[多线程的实现方式 12](#_Toc488413056)

[线程退出的方法 12](#_Toc488413057)

[守护线程（daemon） 12](#_Toc488413058)

[序列化与反序列化 12](#_Toc488413059)

[内存溢出和内存泄漏 13](#_Toc488413060)

[servlet生命周期及方法 13](#_Toc488413061)

[JSP工作原理 13](#_Toc488413062)

[JSP动态include和静态include 13](#_Toc488413063)

[Java Collection框架 14](#_Toc488413064)

[集合框架的内容 14](#_Toc488413065)

[HashMap内部实现 14](#_Toc488413066)

[put方法 14](#_Toc488413067)

[get方法 14](#_Toc488413068)

[hash函数 14](#_Toc488413069)

[扩容时元素在新表中的位置 14](#_Toc488413070)

[LinkedHashMap 15](#_Toc488413071)

[TreeMap 15](#_Toc488413072)

[ConcurrentHashMap 15](#_Toc488413073)

[ArrayList 16](#_Toc488413074)

[LinkedList 16](#_Toc488413075)

[CopyOnWriteArrayList 16](#_Toc488413076)

[ArrayDeque 17](#_Toc488413077)

[PriorityQueue 17](#_Toc488413078)

[BlockingQueue 17](#_Toc488413079)

[JVM 17](#_Toc488413080)

[虚拟机参数 18](#_Toc488413081)

[虚拟机区域 18](#_Toc488413082)

[GC算法 18](#_Toc488413083)

[标记-清除算法 18](#_Toc488413084)

[标记-整理算法 18](#_Toc488413085)

[复制算法 18](#_Toc488413086)

[GC收集器 19](#_Toc488413087)

[GC Roots有哪些？ 19](#_Toc488413088)

[类加载过程 20](#_Toc488413089)

[双亲委托模型 20](#_Toc488413090)

[J2EE 20](#_Toc488413091)

[redirect和forward的区别 20](#_Toc488413092)

[servlet不是线程安全的 20](#_Toc488413093)

[数据库 21](#_Toc488413094)

[数据库范式 21](#_Toc488413095)

[数据库四大特性(ACID) 21](#_Toc488413096)

[having和where的区别 21](#_Toc488413097)

[数据库死锁 21](#_Toc488413098)

[数据库索引存储结构 22](#_Toc488413099)

[Mysql索引实现 22](#_Toc488413100)

[Mysql数据库的四种隔离级别 22](#_Toc488413101)

[Join 23](#_Toc488413102)

[SQL语句优化（主要避免全表扫描） 23](#_Toc488413103)

[drop、delete和truncate的区别 23](#_Toc488413104)

[数据库优化方法 24](#_Toc488413105)

[数据库拆分 24](#_Toc488413106)

[数据库连接池 24](#_Toc488413107)

[NoSQL（redis） 24](#_Toc488413108)

[Sentienl 24](#_Toc488413109)

[集群 24](#_Toc488413110)

[常用命令 25](#_Toc488413111)

[消息队列 25](#_Toc488413112)

[两种消息队列实现方式 25](#_Toc488413113)

[数据结构和算法 26](#_Toc488413114)

[红黑树 26](#_Toc488413115)

[快速排序 26](#_Toc488413116)

[归并排序(Collections.sort()内部实现) 27](#_Toc488413117)

[Dijkstra算法 28](#_Toc488413118)

[计算机基础 28](#_Toc488413119)

[进程和线程的区别？ 28](#_Toc488413120)

[同步与阻塞的关系 28](#_Toc488413121)

[设计模式 29](#_Toc488413122)

[生产者消费者模式 29](#_Toc488413123)

[单例模式 30](#_Toc488413124)

[动态代理模式 30](#_Toc488413125)

[装饰器模式 31](#_Toc488413126)

[适配器模式 31](#_Toc488413127)

[Java 8新特性 31](#_Toc488413128)

[函数式编程 32](#_Toc488413129)

[回调函数、函数式编程、面向对象的区别 32](#_Toc488413130)

[Collection的内部迭代和之前的外部迭代的区别？ 32](#_Toc488413131)

[Spring 32](#_Toc488413132)

[Spring注入的方式 32](#_Toc488413133)

[IoC依赖注入的好处 33](#_Toc488413134)

[AOP 33](#_Toc488413135)

[Repository、Component、Service、Controller注解 33](#_Toc488413136)

[Spring MVC 34](#_Toc488413137)

[请求处理过程 34](#_Toc488413138)

[使用Spring MVC遇到的问题 34](#_Toc488413139)

[Hibernate 35](#_Toc488413140)

[Hibernate六大核心接口 35](#_Toc488413141)

[query的list和iterate的区别 35](#_Toc488413142)

[get和load有什么区别 35](#_Toc488413143)

[update与lock的区别 36](#_Toc488413144)

[merge()方法 36](#_Toc488413145)

[save、persist和saveOrUpdate方法的不同点 36](#_Toc488413146)

[hibernate中的命名查询 36](#_Toc488413147)

[sessionFactory 37](#_Toc488413148)

[Session 37](#_Toc488413149)

[hibernate中对象的三种状态 37](#_Toc488413150)

[Hibernate的三种缓存级别 38](#_Toc488413151)

[hibernate二级缓存策略 38](#_Toc488413152)

[hibernate锁 39](#_Toc488413153)

[hibernate事务管理 39](#_Toc488413154)

[hibernate的N+1查询问题 39](#_Toc488413155)

[MyBatis 40](#_Toc488413156)

[mybatis创建sqlsession主要步骤 40](#_Toc488413157)

[${}和#{}的区别 40](#_Toc488413158)

[mybatis中的映射文件配合Dao的工作原理 41](#_Toc488413159)

[mybatis如何分页，分页插件的原理 41](#_Toc488413160)

[mybatis动态sql，实现原理 41](#_Toc488413161)

[mybatis如何将sql执行结果封装并返回 41](#_Toc488413162)

[mybatis支持延迟加载的原理是什么 41](#_Toc488413163)

[mybatis的三种executor执行器的区别 41](#_Toc488413164)

[Cache标签的作用 42](#_Toc488413165)

[参数传递 42](#_Toc488413166)

[动态sql 42](#_Toc488413167)

[mybatis配置文件 42](#_Toc488413168)

[计算机网络 43](#_Toc488413169)

[滑动窗口 43](#_Toc488413170)

[TCP/IP协议族 43](#_Toc488413171)

[OSI七层 43](#_Toc488413172)

[OSI模型与TCP/IP模型的对比 44](#_Toc488413173)

[TCP、UDP的区别 44](#_Toc488413174)

[TCP状态转换 45](#_Toc488413175)

[常见算法 45](#_Toc488413176)

[蓄水池抽样 45](#_Toc488413177)

[海量数据找频繁前N个数据 45](#_Toc488413178)

[海量数据找第K大/中位数数据 46](#_Toc488413179)

[bitmap进行数据的判重以及集合求交集 46](#_Toc488413180)

[找出数组中的第N大的数 46](#_Toc488413181)

[Tomcat 46](#_Toc488413182)

[tomcat处理http请求的过程 46](#_Toc488413183)

[session和cookies的区别 47](#_Toc488413184)

[杂项 47](#_Toc488413185)

[分布式与集群的区别 47](#_Toc488413186)

[分布式锁的redis实现 48](#_Toc488413187)

[zookeeper 48](#_Toc488413188)

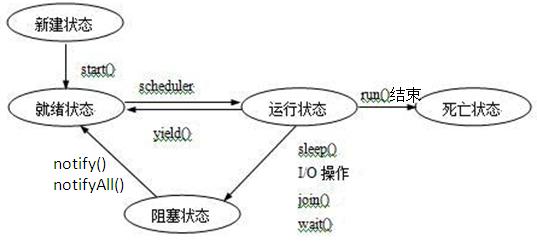
# 线程相关

## Java线程安全（并发）的实现方式？

虚拟机主要的内存分布有堆和栈两种，其中栈作为线程私有内存区域，不会与其他线程共享也就不会有并发安全的问题。但由于堆内对象被所有线程共享，多线程在并发环境下容易得到与串行执行不一样的结果。造成这样结果的原因是，由于JVM有主内存和工作内存的分别，线程执行时首先从主内存获取一份拷贝到本地工作内存再开始工作，工作完成将本地内存同步到主内存，但由于各线程工作内存存在隔离，一个线程对共享变量的修改不能立即被其他线程感知，这样线程对主内存修改的先后顺序将会对结果产生影响。

要实现线程安全，可以采用：1、将要被多线程共享的对象或者代码片段加锁(synchronized)，保证任何时间仅有一个线程在操作，同时synchronized提供了可见性的保证，在线程在释放锁后，保证加锁期间都对象的修改能够被其他线程感知；2、使用原子操作，JDK1.5之后新增的JUC.atomic包内提供了Boolean的原子操作、以及integer、long及它们的数组的原子操作，还有reference类型的原子操作，这些类使用的时候不需要加锁，原子操作的特性保证并发的安全性；3、使用ThreadLocal，每个Thread都有一个ThreadLocal. ThreadLocalMap字段，对ThreadLocal的修改只能被线程内部所共享，实现线程隔离，也就不会出现线程不安全的问题。

## 线程状态转换



Thread.yield()：让出CPU时间片执行时间，和其他线程一起参与调度分配。

Thread.sleep()：线程等待一段时间，不释放锁。

Object.wait()：线程挂起，等待其他线程显式唤起。

## 并发依赖特性

**原子性**，任何时候只有一个线程操作的叫做原子操作，java内存模型通过read、load、assign、use、store和write六个操作保证变量的原子操作；**可见性**，一个线程修改了共享变量的值，其他线程可以立即得知这个修改，java实现有volatile、synchronized以及final来保证可见性；**有序性**，线程内部，所有操作是有序执行的，java通过volatile和synchronized保证操作的有序性，volatile禁止指令重排序优化实现有序性，synchronized通过一个变量一个时刻只允许一个线程操作实现有序性

## 线程池

corePoolSize：核心线程池数量，maximumPoolSize：最大线程池数量，大于核心池的数量相当于线程池面对突然大任务量的一种补救。

当有任务到来的时候，且当前线程不大于corePoolSize，则会尝试创建一个线程执行该任务

当任务到来的时候，线程数量大于corePoolSize，则会尝试将任务加入缓冲队列中，这个缓冲队列一般使用的ArrayBlockingQueue/LinkedListBlockingQueue实现，若加入失败，则说明任务缓冲队列已满，这时尝试创建新线程执行任务，当线程数量大于maximumPoolSize时候，则执行拒绝策略。

**线程池作用**：减少重复创建、销毁线程的消耗；提高响应任务的效率，任务到达时，可以不用等待线程创建即可执行；便于线程的管理。

**线程池的状态**：

running：创建线程后的状态。

shutdown：调用shutdown方法后的状态，不接受新任务，等待线程池内任务执行完毕，若缓冲队列存在任务将不会被执行。

stop：shutdownnow方法后的状态，不接受新任务，尝试终止正在执行的任务。

terminate：所有线程被销毁，缓存队列被清空或者执行结束后。

## 线程池种类

**newFixedThreadPool**：创建一个指定工作数量的线程池，每当提交一个任务就创建一个工作线程，如果线程数达到线程池初始的最大数，将后来的任务放在任务等待队列中。

**newCachedThreadPool**：可缓存的线程池，特点的线程数没有限制，如线程池中线程 长时间处于空闲状态将被回收。

**newSingleThreadPool**：创建一个单线程的executor，即只创建唯一的工作线程执行任务，特点是可以保证任务的执行顺序。

**newScheduleThreadPool**：创建定长的线程池，支持定时以及周期的任务执行。

## 影响单台服务器并发量的因素有哪些

**从并发编程的挑战上有：**

避免频繁的上下文切换，CPU通过时间片分配算法来执行任务，当前任务执行一个时间片后会切换到下一个任务，获得时间片后加载任务状态，重新执行，频繁的上下文切换影响线程的执行速度，使用无锁并发编程，避免多线程竞争锁时，引起的上下文切换，比如使用concurrentHashMap，CAS算法，如atomic包使用CAS算法来更新数据，不加锁，避免创建不需要的线程

**资源限制方面：**

服务器带宽，磁盘读写速度和CPU的处理速度

## I/O密集型和CPU密集型的任务线程开多少合适？

**CPU密集型**的任务往往load都很高，应该充分发挥多核的优势，使得全部核心都参与工作，同时也要避免过多的线程，引起频繁的线程上下文切换，增加系统开销。线程数通常比核心数稍高一点。

**I/O密集型**的任务通常的情况是分配给该线程的时间片还未用完，线程就因为等待IO操作而被阻塞，白白浪费了CPU时间，此时应该多开一些线程，让线程在等待IO操作时，可以切换到其他线程继续工作。线程数=核心数/(1-阻塞系数)(阻塞系数一般在0.8-0.9之间)。

## blocked和waiting的区别

blocked是在临界区外等待，waiting是在临界区内调用对象的wait的方法，等待其他线程调用该对象notify方法，time waiting 是有时间的等待sleep或者wait

# Java内存模型

Java内存模型是为了保证线程之间安全的共享主内存的共享变量，因此提供了一些约束来保证这样的共享安全。如通过内存屏障来避免重排序导致多线程环境下程序的不可预测的行为。以及提供一些关键字如volatile、synchronized等来辅助实现线程安全。

## 内存屏障

是一种指令，为了优化程序性能，编译期和CPU能够重排序指令，内存屏障即阻止任何其他指令与其重排序，另外一个效果就是强制使得其他线程持有的变量副本失效，强制从主存中读取

## happens-before

提供一些规则，保证内存可见性，对于有happens-before的两个操作，不一定要求一个操作在另一个操作之前发生，但之前的操作的结果一定是对后一个操作可见的

# synchronized关键字

synchronized可以修饰成员方法、静态方法和方法块，同步的方式获得锁，java中每一个对象都可以看做是一个锁，同步方法的锁是当前对象；静态同步方法的锁是当前对象类的class对象；同步方法块的锁是synchronized括号里配置的对象。

偏向锁：使用了一种等待竞争才会释放锁的机制，当其他线程尝试竞争偏向锁，持有偏向锁的线程才会释放锁，适合只有一个线程执行同步代码块的情况。

轻量级锁：竞争的线程不会阻塞，锁竞争失败则自旋。适合同步块执行时间较短，追求响应时间的情况。

重量级锁：锁竞争失败则阻塞，适合同步块执行时间较长，吞吐量要求高的系统。

## 原理

Java规范规定使用进入和退出monitor对象来实现方法同步和代码块同步，使用monitorenter和monitorexit指令实现，monitorenter放在同步开始位置，monitorexit放置在同步结束或者异常位置。任何一个对象都有一个monitor与之关联，线程执行到monitorenter将会尝试获取对象monitor的所有权。

## 锁(Lock)

JUC提供了API级别的并发加锁实现，与synchronized隐式释放锁不同，Lock需要显式释放，最好是在finally块中释放，防止方法异常退出使得锁无法释放。同时实现synchronized没有的非阻塞加锁以及公平锁的功能。通过lock.newCondition可以将锁对象与监视器分开，区别synchronized一个锁只有一套监视器方法。

Lock在加锁时首先getState检查当前锁状态，是否加锁，否则cas将当前线程设置为锁的独占线程；若已经加锁，则判断当前线程是否是该锁的独占线程，是的话将该锁的重入数加1，同时设置state。

## AQS

AQS是一个同步队列抽象类，内部维护了一个volatile state的成员变量以及一个存放资源等待线程的双向链表。

**获得资源**的大致过程是，首先tryAcquire，如果不成功则将当前线程放入等待队列的尾部，并标记为独占模式，然后使线程在等待队列中获取资源，直至获取其他线程释放资源后唤醒自己，自己拿到资源后即可返回。



**释放资源**过程：release方法有个int型的参数，指定资源释放的数量，如果彻底释放了，将会唤醒等待队列的其他线程来获取资源。

# Volatile关键字

**提供的保证**：提供顺序及可见性的保证，Volatile关键字有禁止指令重排序的语义，从而保证了在并发环境下，不会出现虚拟机优化代码执行而产生问题。

**可见性的原因**：线程对于volatile变量的更新会使得，其他线程持有的该变量的副本失效，因此使用该变量时，需要重新从主存中读取，保证一个线程对变量的修改，能够被其他线程感知。

**使用volatile变量的实例**：一个资源准备线程，其余的工作线程，等待资源准备线程修改状态量，才开始工作。

# JAVA API相关

## java.util.concurrent.atomic包

提供了四种原子更新方式，分别是原子更新基本类型，原子更新基本类型数组，原子更新引用和原子更新字段，可以看作是基于Unsafe的包装类，依赖于处理器底层提供的原子指令实现，因此并不是一定保证线程不被阻塞。

更新基本类型：AtomicInteger，方法有addAndGet（原子与传入的参数相加，返回新值），compareAndSet（如果与期望值一致则设置后返回true，否则返回false），getAndIncrement（自增，返回旧值），lazySet（多核下多个CPU缓存之间同步一个内存值代价比较昂贵，该方法导致其他线程在一小段时间内还是可以读到旧值），getAndSet（原子设置新值，返回旧值）。

原子更新基本类型数组：将通过构造函数传入的参数复制一份，对数组的改变不影响原始数组。

原子更新引用类型：即用一个对象的内容替换另一个对象的内容。

原子更新字段：AtomicIntegerFieldUpdater等

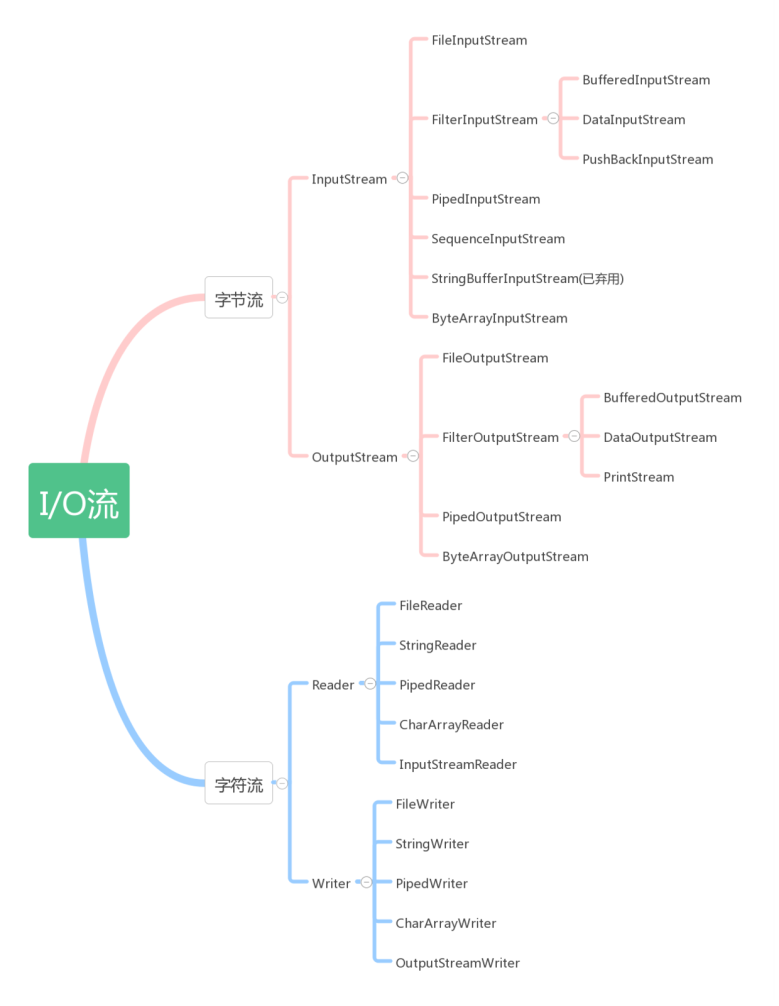
## fork/join结构

主要是用来完成将大任务分割(fork)成小任务，然后再将计算结果合并(join)的到一起的事情。fork/join使用两个类来完成任务

ForkJoinTask：主要使用的ForkJoin框架，提供了任务执行中的fork/join机制，提供了两个子类：RecursiveAction和RecursiveTask，区别是前者没有返回值，后者有，通过继承这两个类，然后重写compute方法，实现任务执行过程。

ForkJoinPool：ForkJoinTask通过ForkJoinPool来执行，形如ForkJoinPool.submit(ForkJoinTask),任务分割出的子任务会添加到当前工作线程所维护的双端队列头部，当一个队列没有任务时，会随机从其他工程线程的队列的尾部获取一个任务。

## JAVA IO



## 在有return的情况下try catch finally的执行顺序

1、不管有没有出现异常，finally都会执行

2、try和catch中有return时，finally依然会执行

3、finally是在return值确定后执行的，此时先把要返回的值保存起来

4、finally中包含了return程序会提前退出

## error和exception的区别

error表示系统级别的错误和程序不需要的处理的异常，是java环境中的内部错误或者硬件问题

exception表示需要程序处理的异常，引起此类异常的主要是因为程序设计的问题

## checked异常和unchecked异常

checked exception继承exception类，例如IO异常，FileNotFoundException

unchecked exception继承runtimeException

## 多线程的实现方式

1、继承Thread类，重写run方法

2、实现Runnable接口，实现run方法

3、如果使得线程有返回值，应该实现Callable接口，executorService.submit方法的参数即是Callable对象，返回值是future对象，可以通过future获得该线程测返回值

## 线程退出的方法

1、正常退出

2、thread.stop方法退出，可能会造成意外的情况

3、thread.interrupt，如果当前线程处于阻塞状态，则会抛出interrupt异常

## 守护线程（daemon）

在java中是低级别的线程，具有最低的优先级，当系统只剩守护线程时，系统就会退出，典型的守护线程即是java的垃圾回收线程，它常驻后台监控以及回收资源，当系统所有非守护线程均退出时，垃圾回收线程也就无事可做，主动退出。

## 序列化与反序列化

序列化把对象以一种格式形式保存下来，方便传输，反序列化执行序列化的逆过程，java中使用objectInputStream和objectOutputStream代表输入和输出流，主要方法是writeObject()和readObject()方法

## 内存溢出和内存泄漏

内存溢出是没有足够的空间来分配内存，重现例子是，一次往list加入大量的对象

内存泄漏是指分配内存空间，却没有释放，堆积到一定程度一定会造成内存溢出，所以内存泄漏是内存溢出的一种途径，重现是建立连接而不释放

## servlet生命周期及方法

加载-实例化(只在第一次请求时加载和实例化)-服务-销毁

init()在首次请求servlet执行，用来初始化servlet，无论有多少客户机访问servlet，都不会重复执行

service()负责响应客户端的请求，每一个请求都会调用一个service方法，同时传入servletRequest和servletResponse作为参数，httpservlet会根据请求方法调用响应的do方法

destroy()服务器停止，servlet卸载时执行，用于释放占用的资源

## JSP工作原理

当浏览器向服务器请求一个JSP页面时，服务器会检查这个JSP文件是否被更新或者是创建后第一次被访问，如果是的话，容器的JSP引擎就会将JSP文件转换为一个代表servlet类的JAVA源代码文件，紧接着会被编译成字节码文件，和servlet一样的执行过程

## JSP动态include和静态include

动态include是使用jsp:include标签实现，它会动态的检查所引入页面的变化，如果发生变化，下一次请求会将包含页面的新内容

静态include是使用<%@ include=””%>实现，它不会检查被包含页面的内容变化，适合引入那些内容不常改变的页面，比如导航栏、页脚等内容

# Java Collection框架

## 集合框架的内容

其中包括Collection接口和Map接口。Collection是一个标准接口，规定子类需要实现的方法，主要方法有add、addAll、contains、toArray、remove，Collection子类提供三种不同的实现，有set、list、queue，遍历Collection主要有foreach、iterate方法，其中foreach方法遍历时，不允许同时remove修改元素，否则会抛出ConcurrentModificationEeception；Map主要方法有put、get等方法。

## HashMap内部实现

## put方法

首先对key做hash（hash函数是(key.hash() ^ key.hashcode() >>> 16)），得到hash计算key的位置（(n-1)&hash）；

如果没有碰撞生成newNode直接放到bucket里面，有碰撞以链表的形式放入bucket中；

如果一个bucket元素过多（大于treeify\_threshold），使用treefyBin转换成红黑树；

如果节点已存在用新值替换old值；

如果bucket满了(超过capecity\*loadfactor)，则触发扩容操作。

## get方法

如果key是bucket里的第一个节点，直接命中；

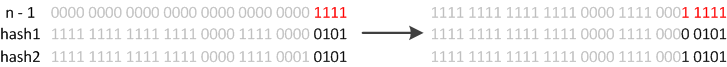
如果有冲突，则通过key.equals(k)去查找对应的entry；(树结构查找效率为o(logn)，链表查找效率是o(n));

## hash函数

把hash值的高16位与低16位做异或，使得hash值的高低位都参与运算，减少碰撞的发生，然后再与数组长度做与运算。

## 扩容时元素在新表中的位置

数组每次扩容为原来的二倍(n << 1)，之前元素在新数组中的位置要么不变，要么移动2的幂次方个位置，这是因为数组扩容二倍,较之前二进制高位多了个1，再与hash值做异或之前看之前该高是否为1，这样既不用重新计算hash，同时也能随机分布元素。



## LinkedHashMap

扩展HashMap，每个Entry增加双向链表，号称是最占内存的数据结构。

支持iterator（）时按Entry的插入顺序来排序（如果设置accessOrder属性为true，则所有读写访问都排序）。

插入时，Entry把自己加到Header Entry的前面去。如果所有读写访问都要排序，还要把前后Entry的before/after拼接起来以在链表中删除掉自己，所以此时读操作也是线程不安全的了。

## TreeMap

以红黑树实现，红黑树又叫自平衡二叉树：对于任一节点而言，其到叶节点的每一条路径都包含相同数目的黑结点。

上面的规定，使得树的层数不会差的太远，使得所有操作的复杂度不超过 O（lgn），但也使得插入，修改时要复杂的左旋右旋来保持树的平衡。

支持iterator（）时按Key值排序，可按实现了Comparable接口的Key的升序排序，或由传入的Comparator控制。可想象的，在树上插入/删除元素的代价一定比HashMap的大。

支持SortedMap接口，如firstKey（），lastKey（）取得最大最小的key，或sub（fromKey, toKey）, tailMap（fromKey）剪取Map的某一段。

## ConcurrentHashMap

并发优化的HashMap。

在JDK5里的经典设计，默认16把写锁（可以设置更多），有效分散了阻塞的概率。数据结构为Segment[]，每个Segment一把锁。Segment里面才是哈希桶数组。Key先算出它在哪个Segment里，再去算它在哪个哈希桶里。

也没有读锁，因为put/remove动作是个原子动作（比如put的整个过程是一个对数组元素/Entry 指针的赋值操作），读操作不会看到一个更新动作的中间状态。

但在JDK8里，Segment[]的设计被抛弃了，改为精心设计的，只在需要锁的时候加锁。

**初始化**：当有第一个元素到来时ConcurrentHashMap才进行初始化，首先判断sizeCtl，如果sizeCtl小于0，说明其他线程在进行扩容或者初始化，当前线程yield让出时间片，否则，使用cas操作将sizeCtl值置为-1

**put操作**：首先根据key计算出hash，然后获得该node在hash上索引位置，使用Unsafe.getObjectVolatile获取指定数组位置上的值，如果值为null，说明是第一次插入该位置，同样利用Unsafe.compareAndSwapObject插入节点，若失败则自旋重新尝试插入，如果不为空，使用synchronized对节点加锁插入

**扩容**：在扩容过程中，原始位置上放入的是forwardingNode，hash值为-1，存储扩容的nextTable引用

**get操作**：获得所在的数组下标，然后获得节点

## ArrayList

以数组实现。节约空间，但数组有容量限制。超出限制时会增加50%容量，用System.arraycopy（）复制到新的数组。因此最好能给出数组大小的预估值。默认第一次插入元素时创建大小为10的数组。

按数组下标访问元素－get（i）、set（i,e） 的性能很高，这是数组的基本优势。

如果按下标插入元素、删除元素－add（i,e）、 remove（i）、remove（e），则要用System.arraycopy（）来复制移动部分受影响的元素，性能就变差了。

越是前面的元素，修改时要移动的元素越多。直接在数组末尾加入元素－常用的add（e），删除最后一个元素则无影响。

## LinkedList

以双向链表实现。链表无容量限制，但双向链表本身使用了更多空间，每插入一个元素都要构造一个额外的Node对象，也需要额外的链表指针操作。

按下标访问元素－get（i）、set（i,e） 要悲剧的部分遍历链表将指针移动到位 （如果i>数组大小的一半，会从末尾移起）。

插入、删除元素时修改前后节点的指针即可，不再需要复制移动。但还是要部分遍历链表的指针才能移动到下标所指的位置。

只有在链表两头的操作－add（）、addFirst（）、removeLast（）或用iterator（）上的remove（）倒能省掉指针的移动。

Apache Commons 有个TreeNodeList，里面是棵二叉树，可以快速移动指针到位。

## CopyOnWriteArrayList

并发优化的ArrayList。基于不可变对象策略，在修改时先复制出一个数组快照来修改，改好了，再让内部指针指向新数组。

因为对快照的修改对读操作来说不可见，所以读读之间不互斥，读写之间也不互斥，只有写写之间要加锁互斥。但复制快照的成本昂贵，典型的适合读多写少的场景。

虽然增加了addIfAbsent（e）方法，会遍历数组来检查元素是否已存在，性能可想像的不会太好。

## ArrayDeque

以循环数组实现的双向Queue。大小是2的倍数，默认是16。

为了支持FIFO，即从数组尾压入元素（快），从数组头取出元素（超慢），就不能再使用普通ArrayList的实现了，改为使用循环数组。

有队头队尾两个下标：弹出元素时，队头下标递增；加入元素时，队尾下标递增。如果加入元素时已到数组空间的末尾，则将元素赋值到数组[0]，同时队尾下标指向0，再插入下一个元素则赋值到数组[1]，队尾下标指向1。如果队尾的下标追上队头，说明数组所有空间已用完，进行双倍的数组扩容。

## PriorityQueue

用平衡二叉最小堆实现的优先级队列，不再是FIFO，而是按元素实现的Comparable接口或传入Comparator的比较结果来出队，数值越小，优先级越高，越先出队。但是注意其iterator（）的返回不会排序。

平衡最小二叉堆，用一个简单的数组即可表达，可以快速寻址，没有指针什么的。最小的在queue[0] ，比如queue[4]的两个孩子，会在queue[2\*4+1] 和 queue[2\*（4+1）]，即queue[9]和queue[10]。

入队时，插入queue[size]，然后二叉地往上比较调整堆。

出队时，弹出queue[0]，然后把queque[size]拿出来二叉地往下比较调整堆。

初始大小为11，空间不够时自动50%扩容。

## BlockingQueue

BlockingQueue，一来如果队列已空不用重复的查看是否有新数据而会阻塞在那里，二来队列的长度受限，用以保证生产者与消费者的速度不会相差太远。当入队时队列已满，或出队时队列已空，不同函数的效果见下表：

立刻报异常 立刻返回布尔 阻塞等待 可设定等待时间

入队 add（e） offer（e） put（e） offer（e, timeout, unit）

出队 remove（） poll（） take（） poll（timeout, unit）

查看 element（） peek（） 无 无

# JVM

Java虚拟机的运行时内存结构主要有方法区、堆、虚拟机栈、本地方法栈以及程序计数器组成。其中方法区主要保存的是类加载信息、字面量、常量，符号引用等信息；堆是线程共享的，按照对象存活年龄分为年轻代、老年代；虚拟机栈是线程私有，主要保存了局部变量表、操作数栈等信息；本地方法栈和虚拟机栈一样，保存的是本地方法信息；程序计数器类似于行指针，指示当前代码执行位置。

## 虚拟机参数

-xmx：最大堆内存

-xms：最小堆内存

-xss：栈大小（1.5之前每个栈为256k，之后为1M）

-xmn：最大新生代大小

-xx:SurvivorRatio 设置eden区与survivor区的比值

-xx:MaxDirectMemorySize 设置直接内存大小

## 虚拟机区域

**java堆**：存放对象实例和数组，是虚拟机垃圾回收的主要对象，分为新生代和老年代

**方法区**：被虚拟机加载的类信息、常量、静态变量，符号引用，字面量（后两个是运行时常量池的内容）

**虚拟机栈**：线程私有，存放java方法的栈帧，栈帧主要记录方法的局部变量表、操作数栈、动态链接、方法出口等信息

**本地方法栈**：和虚拟机栈类似，关于native方法的栈

**程序计数器**: 相当于行号，指示了下一条需要执行的字节码指令，线程私有，各线程互不影响

## GC算法

## 标记-清除算法

在GC阶段，通过GC Root枚举，找到那些不能直接关联到GC Root的对象，即为需要清除的对象，然后在清除阶段直接将这些对象所在内存回收，这样做可能会导致大量的内存碎片，下一次没有足够的内存分配时候，会再进行一次GC过程。

## 标记-整理算法

GC阶段和标记清除算法一致，但在回收对象内存时，将不需要回收的对象统一向一侧移动，适合对象存活时间比较久的区域，比如老年代。

## 复制算法

仅在一块区域分配内存，回收内存时将存活的对象复制到另一块区域，分配内存的区域直接回收，此算法使得某一块内存总不能参与对象内存分配，适合一次回收后，较少对象存活的区域，比如年轻代。

## GC收集器

**Serial收集器**：单线程，Stop the World，在使用GC垃圾收集的时候需要暂停全部的用户线程，造成GC停顿，适合client模式下的新生代的垃圾收集。

**ParNew收集器**：serial收集器的多线程版本，server模式下的默认新生代收集器，因为除了serial外，只有它可以配合CMS收集器，使用-XX:+UseParNewGC强制使用ParNew

**Parallel Scavenge收集器**：虽然和ParNew一样是使用复制算法的新生代多线程收集器，但它主要关注线程的吞吐量（吞吐量=用户线程时间/(用户线程时间+垃圾收集时间)）

**Serial Old收集器**：Serial 的老年代版本，client模式下，可以搭配全部的新生代收集器

**Parallel old收集器**：Parallel Scavenge的老年代版本，注重吞吐量，CPU资源敏感

**CMS收集器**（Concurrent Mark Sweep）:老年代的垃圾收集器，目标是减少甚至消除垃圾收集时，用户线程的停顿时间，垃圾收集过程分为1、初始标记，2、并发标记，3、重新标记，4、并发清除，初始标记阶段仅枚举GC Roots能直接关联的对象，虽然暂停用户线程，但速度较快。并发标记阶段，GC Roots过程，进行全部的存活对象枚举，速度较慢，但与用户线程同时进行。重新标记更改并发标记时改变的引用关系，进行STW，但速度较快

CMS存在的问题：1、CPU资源敏感，并发标记阶段，并发标记线程会占用一部分线程资源，总吞吐量降低

2、无法处理浮动垃圾，即并发标记过程产生的垃圾，可能出现Concurrent Mode Failure导致另一次Full GC的产生。3、标记清除算法会存在内存碎片的问题，可以内存碎片的整理功能，但是停顿时间就过长了

**G1收集（Garbage First）**：同CMS一样是关注垃圾收集时的用户线程停顿时间。可以同时用于年轻代和老年代。将内存划分为不同的Region，评估各region的进行回收的价值大小，优先回收价值大的region。G1使用Remember Set记录region之间以及其他收集器年轻代和老年代之间的对象应用，避免扫描全堆

G1收集器运作过程：1、初始标记，2、并发标记，3、最终标记，4、筛选回收

过程大致与CMS类似，但进行可达性分析时范围限制在region的Remember Set中，最终标记也是为了发现并发标记阶段引用的改变情况，最终筛选回收则回收那些垃圾收集价值大的region

## GC Roots有哪些？

虚拟机栈中引用的对象；

方法区中静态属性/常量引用的对象

native方法引用的对象

## 类加载过程

**加载**：通过类的全限定名获取这个类二进制字节流；将类的静态数据结构转变为运行时数据结构；在方法区生成代表这个类的class对象，作为运行期这个类各种信息的访问入口

**验证**：保证class中的字节流信息的正确性

**准备**：为类变量分配内存并设置初始值（数据零值，不是程序设置的值），final static会被设置成程序指定的值

**解析**：将常量池的符号引用转变为直接引用的过程

**初始化**：真正执行类中定义的java代码，执行类构造器的<clinit>()方法，该方法包括类变量的赋值操作以及静态语句块

## 双亲委托模型

类加载器收到类加载请求首先将请求委托给父类加载器，这样所有的类加载请求都会到达启动类加载器，只有当父类加载器反馈无法完成类类加载请求时，子类才会尝试加载

双亲委托模型是通过loadclass实现的，首先检查类是否已经被加载过，如果没有则调用父类加载器的loadclass方法，如果父类加载器为空，则默认使用启动类加载器加载，如果加载失败抛出ClassNotFoundException

# J2EE

## redirect和forward的区别

redirect（重定向）：浏览器地址改变，不携带上一次请求的request

forward（转发）：浏览器地址改变，携带上一次请求的request

## servlet不是线程安全的

web容器在启动的时候，只会实例化单个servlet一次，而当servlet有实例变量的时候，多个请求并发的调用一个servlet可能会出现线程安全问题。解决办法是尽量不使用servlet的实例变量，以及必要的时候加锁。

# 数据库

## 数据库范式

1NF：列不可再分

2NF：表必须有主键并且没有包含在主键中的列，必须完全依赖于主键，不能部分依赖

（一个学生，上一门课可以决定一个教材，其中可以看出教材对于主键（学生，课程）是部分依赖的，部分依赖于课程，当课程换教材后改动量会很大，总之不满足2NF会存在插入，删除，修改异常）

3NF：非主键直接依赖主键，不能出现间接依赖，这样会造成冗余（实际应用中，适当冗余也是有利的），3NF要求字段没有冗余。

**理解：**范式越高，数据冗余越少，但是表链接时效率更低。范式低，数据冗余高，但是查询会更快，具体的时候还是看一个具体的空间和时间的折中。

## 数据库四大特性(ACID)

**原子性**：同一个事务里的操作要么全做，要么全不做

**一致性**：事务使数据库从一个一致性状态转变为另一个一致性状态

**隔离性**：当多个用户并发访问数据库时，数据库为每一个用户开启的事务不能被其他事务干扰，多个并发事务要相互隔离

**持久性**：事务一旦提交，对数据库的改变就是永久的，即使数据库遇到故障也不会丢失提交事务的操作

## having和where的区别

where从表读出数据的行数，having返回客户端的行数

where可以使用索引，having不使用索引，只能在临时表中操作

## 数据库死锁

**必要条件**：1、资源独占，2、请求且不释放资源，3、不剥夺资源，必须主动释放，4、循环等待

**避免死锁的方法**：1、要求事务一次将要用到的所有资源加锁，否则不执行，2、预先规定一个封锁顺序，所有事务都必须按照这个顺序对资源加锁

## 数据库索引存储结构

**B-Tree数据结构**：每个非叶子节点由n个指针和n-1个key组成；每个叶子节点最少包含一个key和两个指针，最多包含2d-1个key和2d个指针，叶节点的指针全为null；一个节点中的key从左向右由高到底排列；每个节点的两侧均为指针，中间是数据，交叉排列；若一个节点最左边的指针指向一个非空节点，那么该节点所有key均小于该指针相邻key的值，同样，最右则均大于，若指针位于中间位置则指向节点key值位于左右相邻key之间。

**B+树数据结构**：与B树主要不用是，B+树内节点只存储key，而不存储data，页节点存储data

**索引结构的好坏：**主要是看查找效率，由于索引可能很大，存储在磁盘上，磁盘I/O成为索引效率的主要瓶颈，使用B树结构可以为每一个节点以磁盘页为单位分配存储空间，这样不仅使得逻辑上相邻的数据在物理上也是相邻的，同时，一次磁盘读操作可以将节点全部读出，起到了局部性原理中预读的效果，加快索引的查找速度。对于B+树比B-树有更好的索引效率，因为B+树内节点不保存数据，只保存key，因此一页可以包含更多的key。

## Mysql索引实现

MyISAM索引实现：B+树存储结构，非聚集索引，索引文件与数据文件分开，叶节点data存放的数据的内存地址。

InnoDB索引实现：B+树存储结构，聚集索引，索引文件既是数据文件，叶节点存放的实际的数据，使用Innodb存储引擎，表必须有一个主键，若没有主键则会隐含一个自增主键，对于辅助索引，在叶子节点的data域存放的是主键索引，然后用主键到主索引中检索获得记录。尽量使用单调列作为主键，方便插入索引。

MySql无法使用索引的情况有：查询条件含有函数；查询值有前缀百分号

## Mysql数据库的四种隔离级别

**serializable**（串行）：避免脏读、不可重复读和幻读的发生

repeatable read（可重复读，mysql默认级别）：可避免脏读、不可重复读

**read committed**（读已提交）：可避免脏读的发生

**read uncommitted**（读未提交）：任何情况都无法保证

脏读、不可重复读、幻读指的都是在同一个事务内发生的情况

**脏读：**指一个事务读到另一个事务未提交的数据。

**不可重复读：**指一个事务第一次读取数据后，数据被另一事务修改，前一事务再读取时发现与第一次读的不一样，发生了不可重复读（这种情况要求前一事务在后一事务完成修改后再读，即可以避免不可重复读的发生）

**幻读：**指前一事务对数据库表整体数据操作，另一事务向表中插入一行，前一事务再次读的时候发现自己的修改对某些数据没有起作用，发生了幻读（这种情况可以要求前一事务在对数据库表完后修改之前，任何事务都不能插入数据，即可避免幻读的发生）

## Join

**inner join**：产生的结果集中，是两个表的交集

**full outer join：** 产生两个表的并集

**lefer outer join**： 产生A表的完全集，B表中匹配的有值，没有匹配的以null值取代

**right outer join**： 产生B表的完全集，A表中匹配的有值，没有匹配的以null值取代

**union**用于合并两个或多个select语句的结果集，连接的select语句必须拥有相同的列，列必须拥有相似的数据类型，列的顺序还必须一致，只列出不同的记录，**union all**取全部结果

## SQL语句优化（主要避免全表扫描）

1、在where、order by子句上建立索引

2、最好不要给数据库留NULL，尽可能使用NOT NULL，NULL可能也会占用空间，如：char(100)，不论是否插入值，都占用100个字符

3、尽量避免where子句对字段进行null值判断，这样会使用引擎放弃使用索引，而进行全表扫描

4、尽量避免在where子句上使用!=和<>操作符，否则也会放弃索引进行全表扫描

5、尽量避免在where中使用or来连接条件，如果一个字段有索引，另一个没有则也会放弃索引，全表扫描，(这样的情况可以将or换成两个查询的union，记住union是或的意思)

6、in和not in也可能会导致全表扫描，能用between就不要用in，很多时候可以使用exist代替in

7、where子句的查询中有%abc%,也会导致全表查询

8、where中使用参数也会导致全表扫描，参数在运行期才可知，引擎不能将访问策略的选择推迟到运行时，必须编译时选择，这样可以强制使用索引；避免where中对字段进行表达式操作，同样是为了避免全表扫描

9、避免对字段进行函数操作，避免全表扫描

drop、delete和truncate的区别

delete是DML，执行过程每次删除一行，不会删除索引，操作记录会被放在事务日志中，可回滚，如有触发器，则会激活触发器

drop删除表，表占用的所有空间都将被删除，不可以回滚

truncate清空数据库，一次性的删除所有表数据，不可回滚，不删除索引，不会触发与表有关的触发器

## 数据库优化方法

建立索引

使用缓存

数据库拆分

## 数据库拆分

1.按功能拆分（垂直拆分）

2.按表中的某一字段范围拆分（水平拆分）

3.hash拆分：hash取模决定使用哪台数据库，平均分配数据

4.基于路由表拆分：按照用户的某一属性拆分

## 数据库连接池

**DBCP**：可设置最大最小连接，连接等待时间，在大并发量的压力下稳定性有所下降，不提供连接池监控

**C3P0**：与hibernate一起发布，在大并发量的压力下稳定性也有一定保证，不提供连接池监控，有空闲连接自动回收功能

**Proxool**：连接池基本功能具备，持续运行的稳定性有一定问题，优势是有连接池监控

**BoneCP**：效率高，速度快。

# NoSQL（redis）

## Sentienl

主要执行的是监控（主从服务器是否可用）、提醒（当服务器出现故障发出通知）、自动故障转移（设置新的主服务器）

## 集群

自动将数据切分到多个节点的能力；当集群中的一部分节点失效或者无法进行通讯时，仍然可以继续处理命令请求

## 常用命令

LPUSH：放置list数据类型

SORT：排序，默认排浮点数，对于字符串排序，可以在最后加ALPHA

SORT…BY…GET：对某列通过某列进行排序，取排序结果的某值

可以给SORT命令加一个store选项，存储在某一个键里面，同时可以用expire设置过期时间，以此来生成SORT操作的结果缓存

TYPE：用来查看key所存储的值的类型，结果有none(key不存在)，string，list，set，zset(有序集合)，hash

SCAN:迭代命令，增量迭代，每次SCAN都会返回一个游标，用于下次增量迭代在上次迭代基础上继续进行，该命令提供的保证仅仅是，保证一个元素在迭代过程存在数据库中，它一定可以在某次迭代被返回，而不能保证不被多次返回。

SCAN命令选项:match用于筛选返回值的模式，count用于指定返回的数目

SETBIT：用来设置比特位

BITCOUNT:用来计算比特位为1的个数

**redis相比memcached的优势**：redis支持更多的数据类型；redis速度快；redis可以持久化其数据；

redis持久化方法

rdb文件；aof文件，append only file，用于数据库状态的恢复，三种可选的方式写日志：每次写操作都保证fsync()执行、每次调用一次fsync()、从不调用，让系统自己来同步

# 消息队列

两个独立的系统可能存在依赖关系，一个系统的短时延迟可能会波及到另一个系统上，消息队列采用推-拉的形式将两个系统解耦合，一个系统向消息队列发送消息，另一系统订阅消息。

## 两种消息队列实现方式

订阅者/发布者以及生产者/消费者

订阅者/发布者：

使用pub/sub指令，是现实是publisher调用redis的publish方法往特定的channel发送消息，subscriber在初始化时候订阅该channel，一旦有消息就立即接收

生产者/消费者：

redis借助list实现，publisher调用redis往特定的key里塞入消息，subscriber调用brpop监听这个key

# 数据结构和算法

## 红黑树

**定义**：是一颗二叉查找树，普通二叉查找树可能会退化成链表，但是红黑树可以保证查找时间o(logn)的效率，性质如下：

1、树的节点不是黑就是红

2、根节点是黑的

3、若一节点是红的，那么它的孩子是黑的

4、每个叶节点（尾部nil指针或者null节点）都是黑的

5、对于任意节点到叶子节点尾端的nil指针包含了相同数目的黑节点

插入/删除：

## 快速排序

**常规实现**：

|  |
| --- |
| public void sort(int[] array, int start, int stop) {  if (start >= stop) return;  int i = start, j = stop;  int x = array[start];  while (i < j) {  while (i < j && array[j] >= x) --j;  if (i < j) {  array[i] = array[j];  ++i;  }  while (i < j && array[i] < x) ++i;  if (i < j) {  array[j] = array[i];  --j;  }  }  array[i] = x;  sort(array, start, i-1);  sort(array, i, stop);  } |

**快排的基准选择**：从基准选择上分为三种快排，一种是固定基准，每次选取第一个或者固定位置下标的值作为基准，这种情况在数组随机性较好的时候，能体现很好的效果，但是碰到有序数组（升序、降序、重复数组）时，快排退化成冒泡排序；一种是随机选择基准，可以克服快速排序在升序或降序数组上的退化问题；另外一种是三数取中的方法，理想状态是选取的基准值可以平均分割数组，三数取中就是近似达到这样一个效果。

**优化方法**：1、利用插入排序在数组长度小及数组部分有序情况下效率比快排好的特点，当待排序数组长度分割到一定大小时（《数据结构与算法分析》Mark Allen Weiness）。

2、在一次分割结束后将key相等的元素聚在一起，再次排序时，可以不用对key相等的元素再进行分割。

3、多线程处理子序列。

## 归并排序(Collections.sort()内部实现)

**基本思想**：将两个有序数组合并，可以发现，单个元素本身就是有序的，因此可以通过递归合并两个有序数组，获得最终的有序数组

**基本实现**：

|  |
| --- |
| public void mergeSort(int[] array, int first, int last, int[] temp) {  if (first < last) {  int mid = (first + last) / 2;  mergeSort(array, first, mid, temp);  mergeSort(array, mid+1, last, temp);  mergeArray(array, first, mid, last, temp);  }  }    private void mergeArray(int[] array, int first, int mid, int last, int[] temp) {  int i = first, j = mid + 1;  int m = mid, n = last;  int k = 0;    while (i <= m && j <= n) {  if (array[i] <= array[j]) {  temp[k++] = array[i++];  } else {  temp[k++] = array[j++];  }  }  while (i <= m) {  temp[k++] = array[i++];  }  while (j <= n) {  temp[k++] = array[j++];  }  for (int t = 0; t < k; ++t) {  array[first+t] = temp[k];  }  } |

## Dijkstra算法

适用情况：求某一点到其他所有点的最短距离

核心思想：维护两个列表，第一个表示已经确定最短路径的顶点集，第二组为尚未确定最短路径的顶点集。算法步骤是，每一次从第二组顶点集内取出顶点权值（到起点的距离）最小的那个顶点放到第一组顶点集中，同时将该顶点作为中间点，更新第二组顶点的权值。直至第二组顶点集为空为止。

# 计算机基础

## 进程和线程的区别？

它们是作为操作系统管理资源的不同形式，在引入线程之前，进程是资源分配和调度以及分派的基本单位，引入线程后，进程只作为系统资源分配的基本单位，进程里可以有多个线程，线程作为系统调度的基本单位，相比于线程，进程切换时，耗费资源较大，效率要差一些。

## 同步与阻塞的关系

同步是目的，阻塞是手段，应用程序发起IO请求，等待返回结果是同步，立即返回是异步，阻塞是在内核状态下完成的

**同步阻塞**（BIO）：以服务器模式为例，客户端有连接请求服务端就启动一个线程处理，不管连接上有否有数据传输

**同步非阻塞**（NIO）：服务器实现是，客户端的所有请求都会注册到一个多路复用器上，多路复用器轮询到连接有I/O请求时才启动一个线程进行处理，客户端也需要不停的去询问IO操作是否就绪

**异步阻塞**（NIO）：应用程序发起IO请求后立即返回，等到内核操作完成后会通知应用程序（与同步的区别就是应用程序不需要等待或者主动询问IO是否完成）

**异步非阻塞**（即AIO，它是NIO的升级版）

# 设计模式

## 生产者消费者模式

使用lock配合condition实现：

|  |
| --- |
| public class ProduceAndConsume {  private final Lock lock = new ReentrantLock();  private final Condition notFull = lock.newCondition();  private final Condition notEmpty = lock.newCondition();  private Object[] resources = new Object[100];  private int putp = 0, getp = 0, count = 0;    public void produce(Object x) throws InterruptedException {  lock.lock();  try {  while (count == resources.length) {  notFull.await();  }  resources[putp++] = x;  ++count;  if (putp == resources.length) putp = 0;  notEmpty.signal();  } finally {  lock.unlock();  }  }    public Object consume() throws InterruptedException {  Object x = null;  lock.lock();    try {  while (count == 0) {  notEmpty.await();  }  x = resources[getp++];  if (getp == resources.length) getp = 0;  --count;  notFull.signal();  } finally {  lock.unlock();  }  return x;  }  } |

## 单例模式

实时加载和延迟加载实现

|  |
| --- |
| public class Singleton {  }  class RushSingleton {  private static Singleton singleton = new Singleton();  public Singleton getSingleton() {  return singleton;  }  }  class LazySingleton {  private static Singleton singleton;  public Singleton getSingleton() {  if (singleton == null) {  synchronized (this) {  if (singleton == null)  singleton = new Singleton();  }  }  return singleton;  }  } |

以上延迟加载使用了Double-checked，这可能是不安全的，因为经过指令重排序，当其他线程调用getSingleton方法时，可能会得到一个没有被初始化完成的单例对象。解决方法可以在方法上加synchronized；或者将属性设置为volatile；或者将要使用的单例对象单独作为一个类的静态属性，这样既保证了延迟加载，也不会出现Double-checked的问题；

## 动态代理模式

|  |
| --- |
| public class Client {  public static void main(String[] args) {  RealSubject realSubject = new RealSubject();  InvocationHandler invocationHandler = new DynamicSubject(realSubject);  Class<?> classType = invocationHandler.getClass();  Subject subject = (Subject)Proxy.newProxyInstance(classType.getClassLoader(),  realSubject.getClass().getInterfaces(), invocationHandler);  subject.request();  }  } |

## 装饰器模式

由三部分组成，**抽象组件**、**具体组件**、**装饰器**，其中具体组件和装饰器实现抽象组件，同时装饰器持有对抽象组件的引用，就是通过这个引用来为具体组件添加装饰功能，应该同时实现抽象组件，将具体组件的对象最为传递到装饰器中，装饰器可以重写抽象组件和具体组件共同的方法，以此添加上装饰功能。

**优点**是：装饰模式动态的将责任附件到对象上，若要扩展功能，提供比继承更灵活的替代方案

**缺点**：虽然扩展性较高，但是不是十分的简洁，无法知道一个类究竟被装饰了多少层

## 适配器模式

要求：1、适配器必须实现原有的旧的接口；

2、适配器对象中持有对新接口的引用，当调用旧接口时，将这个调用委托给实现新接口的对象来处理，也就是在适配器对象中组合了一个新接口

举例：使用外国的插座给国标电插口，这样实现适配器模式就是新建一个适配器，这个适配器实现了外国的插座，同时持有国标的电插口，当进行充电时，将调用委托给国标电插口充电

# Java 8新特性

**接口可以有默认方法**，使用default指定即可，默认方法在子类上可以直接使用；

**lambda表达式**，一个lambda表达式会映射到一个单方法接口上，即每一个lambda表达式都对应一个类型，通常是接口类型

lambda可以使用::关键字引用方法和构造函数

lambda可以直接访问外层局部变量，这样外层局部变量就具有隐性的final语义，不要试图修改

lambda内部对于实例字段以及静态变量是即可以读又可以写

**stream接口**，可以将collection变为stream，操作分为中间操作和最终操作，提供了一些有用的默认方法

**提供了全新的Date API Clock/Timezones等**

## 函数式编程

Java8提供了lambda表达式来实现函数式编程，现在只需要告知程序做什么即可，类库实现将根据运行环境自动进行优化，比如常见的集合迭代，之前使用外部迭代是串行的，若想利用多核的优势，必须自己实现，还要考虑到线程安全的影响，而java8的collection提供了集合的内部迭代方法，将迭代的控制权交由类库实现，可以根据运行时环境使用像并行、乱序、懒加载等优化手段。

## 回调函数、函数式编程、面向对象的区别

Java中的回调函数可能需要大量冗余的代码实现一个匿名内部类，而通过函数式编程，传入一个lambda表达式即可，代码较为简洁。

函数式编程理论基础是lambda演算，不支持赋值操作，不会有任何副作用。

面向对象则是一种编程的范式。

## Collection的内部迭代和之前的外部迭代的区别？

外部迭代是使用固定的顺序遍历集合，是串行的，若想利用多核优势，需要考虑线程安全的影响，而java 8之后的collection接口提供了forEach方法实现了内部迭代，此时遍历集合的是由jdk来控制了，内部实现可以根据运行环境来决定使用并行或者乱序或者懒加载的形式实现。

# Spring

## Spring注入的方式

注入bean三种方式：**构造器方法**、**静态工厂方法**和**实例工厂方法**，使用工厂方法时需要指定实例化工厂bean以及factory-method

**set注入**

<bean name="springAction" class="com.bless.springdemo.action.SpringAction">

<property name="springDao" ref="springDao"></property>

</bean>

**构造器注入**

<bean name="springAction" class="com.bless.springdemo.action.SpringAction">

<constructor-arg ref="springDao"></constructor-arg>

<constructor-arg ref="user"></constructor-arg>

</bean>

**静态工厂注入**

**实例工厂注入**

以上两个工厂方法都需要指定工厂方法

<!—1、定义实例工厂Bean -->

<bean id="beanInstanceFactory" class="cn.javass.spring.chapter2.HelloApiInstanceFactory"/>

<!—2、使用实例工厂Bean创建Bean -->

<bean id="bean4" factory-bean="beanInstanceFactory" factory-method="newInstance">

<constructor-arg index="0" value="Hello Spring!"></constructor-arg></bean>

## IoC依赖注入的好处

面向接口编程，代码更清晰；容器管理对象实例化以及注入依赖资源，代码松耦合；容器管理对象，提高对象的复用性

## AOP

面向切面编程，是面向对象编程的一种补充，主要功能可以用于日志记录、性能统计、安全控制等，内部实现使用JDK动态代理或者cglib代理实现。

由于JDK动态代理限制了被代理类必须要有实现的接口，只能代理那些有上层接口的类；cglib通过生成子类，加入代理逻辑，因此没有实现接口的限制。Spring默认使用jdk代理的方式实现切面。

## Repository、Component、Service、Controller注解

@Repository：用于将数据访问层的（DAO层）的类标识为bean，不需要再在xml文件中手动配置了，同时spring提供了不依赖底层数据库实现的异常封装。

@Component：是一个泛化的概念，仅仅表示一个组件 (Bean) ，可以作用在任何层次。

@Service：通常作用在业务层，但是目前该功能与 @Component 相同。

@Controller：通常作用在控制层，但是目前该功能与 @Component 相同。

# Spring MVC

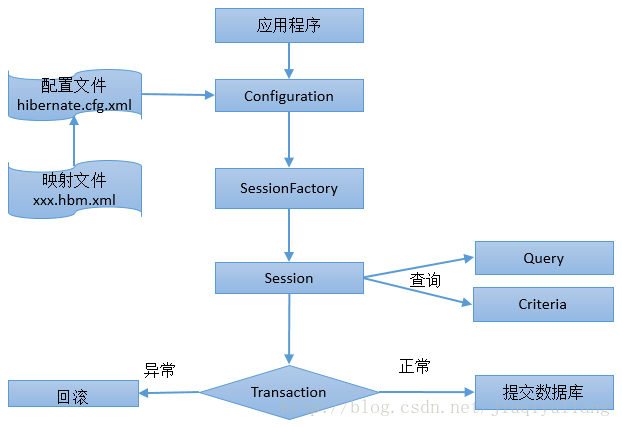
## 请求处理过程



## 使用Spring MVC遇到的问题

**日期转换失败**：通过使指定控制器继承BaseCommandController,并重写initBinder()方法，注册期望的String->Date转换的PropertyEditor（CustomDateEditor）

# Hibernate



## Hibernate六大核心接口

**Configuration**接口:负责配置并启动Hibernate

**SessionFactory**接口:负责初始化Hibernate

**Session**接口:负责持久化对象的CRUD操作

**Transaction**接口:负责事务

**Query**接口和**Criteria**接口:负责执行各种数据库查询

Criteria接口可以有session创建，可以在不使用sql/hql情况下进行数据查询，通过add方法将查询条件加入Criteria实例中（查询条件使用Restrictions表示）

## query的list和iterate的区别

iterate查询时首先回去数据库获得要查询结果的全部id，然后按照id依次查找，先查看该id对应的对象是否在缓存中，若存在则直接取缓存对象，否则发送sql语句查询，并且将查询到的对象放入缓存

list查询时不访问缓存，而是直接使用一条select语句将全部对象查询出来，同样的将结果放入缓存中

get和load有什么区别  
如果在缓存中没有找到相应的对象，get会访问数据库并返回一个初始化好的对象，过程中可能涉及多个数据库的调用；在hbm文件的class标签被加上了lazy后，而load方法在缓存中如果没有发现对象，只会返回一个代理对象，只有对象的getId之外的其他方法被调用时，才会真正访问数据库，执行查询操作。当数据库也没找到时，get方法返回null；load抛出异常。

## update与lock的区别

都是将一个游离对象与一个session实例关联起来，lock方法在执行时，会立即使用查询语句查询版本号进行检查，并不会执行一个update操作，而update不会执行版本检查，在session.flush()时，会先进行版本检查，再执行更新操作。

## merge()方法

对脱管状态对象的操作，执行时会从session缓存中或者数据库中查找与参数持久化标识一致的对象，对比之后如果有改变则会执行update语句，但对象仍旧是脱管状态。

## save、persist和saveOrUpdate方法的不同点

save()立即执行sql insert，返回值是一个持久化标识符serializable对象；

persist()的功能和save()方法类似，都是保存持久化对象，但是persist()方法的返回值是void，并且不保证立即执行，persist()方法可能在flush()时，才会把标识符填入到持久化实例中，并且不更新缓存；

saveOrUpdate()：可以进行insert和update操作，返回值为空

## hibernate中的命名查询

避免在java代码中写HQL语句，破坏代码的可读性，命名查询可以写在\*.hbm.xml配置文件的class标签外，使用query标签定义，查询语句使用<![CDATA[from 持久类 where name = :name]]>形式定义，调用的时候直接使用session.getNameQuery(“”);命名查询里的命名参数的替换使用query.setString(“命名参数名”, ”值”);

当声明在class标签内部时，获取该命名查询需要指定(包+类+配置名)

## sessionFactory

负责初始化hibernate，是线程安全的，一个应用程序只有一个sessionFactory，是用于创建session对象的工厂，应用程序通过它来获得session实例。

## Session

负责持久化对象的CRUD操作；作为持久化对象的缓冲区，缓存中对象被改变后不是立即清理并更新，而是在特定时间、条件下进行的；负责维护数据库的连接，不是线程安全的，多线程之间不能共享session，session使用完之后应当立即关闭，每个session实例与一个数据库事务绑定。

## hibernate中对象的三种状态

**自由状态**：普通的java对象，未与任何session关联。

**持久化状态**：持久化对象实例在数据库中有对应的记录，持有持久化标识，仅与一个session相关联，在session中对持久化对象的更新不会立即同步到数据库中，必须事务提交之后才在数据库中运行真正的SQL语句，使用save、update和saveOrUpdate将一个自由状态对象转换为持久化对象，使用get、load、find、iterate方法查询到的对象也是持久化对象。

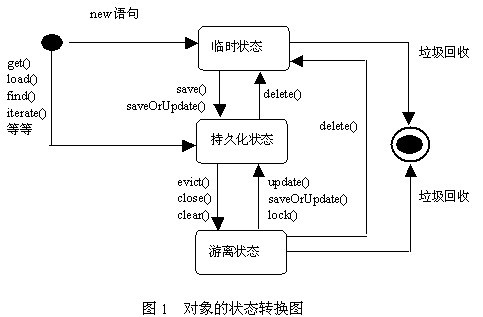
**游离状态**：已进行过持久化并且与session关联的对象，在session关闭之后，就变为游离状态，本质上和自由状态的对象一致，只有对了一个数据库标识id，造成与session关联消失的方法有clear、evict、close。

**自由状态->持久化状态**：new出来一个对象后，即是自由状态，通过save、saveOrUpdate、persist方法将自由状态的对象转变持久化状态的对象，这时候对象仅存在session缓冲中，拥有了持久化标识符，并且和session相关联。

**持久化状态->游离状态**：调用session的evict、close、clear方法将持久化状态的对象变为游离态，这时对象不再和session相关联，并且脱离session清理缓存时，总会将缓存对象在数据库中update，确保托管对象与数据库记录一致。

**游离态->持久态**：对游离态对象调用update、saveOrUpdate、lock方法可以变为持久态对象。

**持久态/游离态->自由态**：delete方法。



## Hibernate的三种缓存级别

**hibernate的一级缓存**：与session的生命周期一致，当前session一旦关闭，缓存消失一级缓存只存实体对象的 ，它不会缓存一般的对象属性（查询缓存可以），即当获得对象后，就将该对象的缓存起来，如果在同一session中再去获取这个对象 时，它会先判断缓存中有没有该对象的ID，如果有就直接从缓存中取出，反之则去数据库中取，取的同时将该对象的缓存起来，get、load、iterate方法可以 支持一级缓存。

**hibernate的二级缓存**：缓存实体对象，方法与一级缓存一样，只是生命周期不一致，在sessionFactory级别维护的缓存，节省数据库往返调用来提高性能，针对整个应用而不是特定的session。

**hibernate查询缓存**：针对普通属性结果集的缓存，查询缓存时保存sql查询的结果，之后相同sql及相同参数的查询就可以从缓存中拿到结果，可以和二级缓存一起使用。

## hibernate二级缓存策略

**事务缓存**：对于经常被读，但很少修改的数据，可以采用这个策略，可以防止脏读（读到别的事务未提交的数据）和不可重复读（两次select得到不一样的结果）。

**读/写缓存（read-write）**：如果程序需要更新数据，可以采取这种策略，防止脏读。

**不严格的读/写缓存（nostrict-read-write）**：偶尔更新数据，且允许脏读，可以采用这个策略。

**只读缓存（read-only）**：从来不会修改的数据，采用这种策略。

## hibernate锁

**悲观锁**：生成查询query时,通过query.setLockMode()加锁。

其中LockMode.None、LockMode.WRITE、LockMode.READ是hibernate内部实现，分别为不加锁、加写锁、加读锁，此外还有依赖数据库的锁配置LockMode.UPGRADE、LockMode.UPGRADE\_NOWAIT，利用数据库的for update实现，使用select \* from xx where xx for update表示在此次事务提交之前，锁定所有满足条件的记录。

**乐观锁**：通过实体的映射文件上添加optimistic-lock属性实现，可选值有none、version、dirty、all，分别表示无乐观锁、通过版本机制实现乐观锁、检查变动的属性实现乐观锁、检查全部属性实现乐观锁。

## hibernate事务管理

当hibernate自己处理事务时，通过sessionFactory.openSession()来开启会话，需要手动提交事务以及关闭session。

可以使用spring来管理事务，依次配置数据源，配置sessionFactory，使用sessionFactory绑定transactionManager，使用transactionManager绑定事务通知，使用aop拦截方法执行

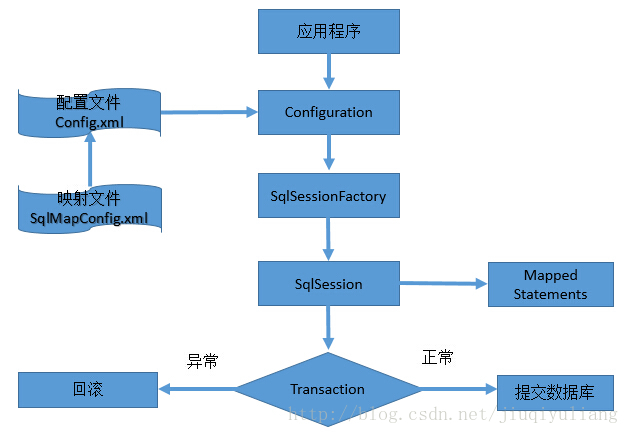
## hibernate的N+1查询问题

**一对多关联**：查询某个对象一次，需要查询与它关联的子表的N个对象。

**多对一关联**：在多的一方通过一次查询获得N个对象，由于关联的存在，也会将N个对象对应的一方查询出来。

**iterate查询**：首先一条sql语句从数据库查询对象ID，然后通过ID依次从缓存中查找，缓存中中找不到再去数据库里查找。

# MyBatis



## mybatis创建sqlsession主要步骤

1、从配置（configuration）中获取environment

2、从environment中获取datasource

3、从environment中获取sqlsessionFactory

4、从datasource中获取数据库连接对象connection

5、在取得的数据库连接对象上创建transaction

6、创建executor对象（sqlsession所有的操作都由它完成）

7、创建sqlsession

## ${}和#{}的区别

**${}**是配置文件的占位符，**#{}**是sql文件的占位符，促使mybatis生成PreparaedStatement来将其替换为?，使用反射取得参数的值进行填充。

## mybatis中的映射文件配合Dao的工作原理

mybatis每个Dao都是一个Mapper接口，Mapper接口没有实现类，调用接口方法时，使用接口的全限定名+方法名作为key值，定位到唯一的MappedStatement(配置文件下的select、update、insert和delete都会被映射成一个MappedStatement)，然后通过JDK动态代理的方法，执行sql语句，返回结果。

## mybatis如何分页，分页插件的原理

mybatis通过使用RowBounds对象分页，是针对result结果集的内存分页

分页插件的原理是根据mybatis提供的插件接口，实现自定义的插件，在插件中拦截sql语句，根据数据库方言，添加物理分页语句进行分页

## mybatis动态sql，实现原理

trim|choose|when|otherwise|where|set|if|foreach|bind

原理是使用OGNL从参数对象中计算表达式的值，根据表达式的值拼接sql，完成动态语法

## **mybatis如何将sql执行结果封装并返回**

一种使用resultMap，在result里使用属性名到列名的映射，另一种使用自动的列名到属性名的映射。

## mybatis支持延迟加载的原理是什么

mybatis仅支持association（一对一）关联对象和collection（一对多）关联对象的延迟加载，原理是，创建目标对象代理对象后，调用目标方法被拦截，若发现某关联属性的值为空，则会发送实现准备好的sql语句，获取关联属性对象实体。

## mybatis的三种executor执行器的区别

有simpleExecutor(每一次update或者select就开启一个statement对象，用完关闭)、reuseExecutor（statement可以重复使用，使用完不关闭，放置在Map中）、BatchExecutor（执行批处理），executor生命周期都限制在sqlSession中，可以在配置文件中指定使用哪种executor，也可以在DefaultSqlSessionFactory中指定。

## Cache标签的作用

mybatis的二级缓存时默认不开启的，可以在映射文件中使用<cache>标签开启，有效果：

所有映射文件的select语句都将被缓存；所有insert、update、delete会清空缓存；缓存使用“最近最少使用”算法来回收；每个缓存可以存储指定多个列表或对象的引用

如：<cache eviction="FIFO" flushInterval="60000" size="512" readOnly="true"> </cache>

可以在不同的命名空间共享一个缓存配置，使用<cache-ref>

## 参数传递

传入参数需要设置参数类型，如果是基本数据类型，那只能传递一个参数，可以使用实体类型传递，也可以使用Map传递，sql语句通过#{Map的键名获取值}

如果要传递多个参数，也可已在参数接口上添加@Param注解如:

public List<Entity> getXXX(@Param(para1) Type1 param1, @Param(para2) Type2 param2, ……)

调用该方法时设置这些参数

## 动态sql

**<where>**：动态sql标签，与if搭配，避免if全不匹配的时候出现 where and的情况，同样的适用用<update>内的<set>标签去除多余逗号，也可以使用trim实现效果

可以使用**<choose><when><otherwise>**来实现选择条件执行的效果，类似于switch、case、default

**<if test=”判断”>**：如果判断成功则输出内部的sql语句

## mybatis配置文件

**properties 属性**：属性配置，指定resource，内部可以添加属性，也可以覆盖source内设置的属性< properties resource=””></ properties >

**settings属性**：全局设置，包括设置缓存、懒加载、自动生成主键等行为

**typeAliases类型别名**：关联XML配置，简写java类名

**typeHandlers类型转换**：继承TypeHandler实现自定义的类型处理器

**ObjectFactory对象工厂**：继承DefaultObjectFactory ，重写自己的对象工厂

**plugins插件**：可以拦截的方法有Executor、ParameterHandler、ResultSetHandler、StatementHandler

**environments环境**：可以配置多个环境，映射多个数据库

**mappers映射器**：映射SQL语句的引用的确切位置

# 计算机网络

## 滑动窗口

**发送方的滑动窗口**：发送方维护一个最近发送帧的序号和最近一个收到的确认帧的序号，发送方最多存储窗口大小个帧，在得到确认前必须准备重发。

**接收方的窗口**：接收维护一个最近收到的帧的序号和一个最近一个确认的帧的序号，当到来的帧序号在这两个帧序号之外时，抛弃处理。

## TCP/IP协议族

三个层次：**网络层**、**传输层**、**应用层**

**网络层协议**：IP，ICMP，ARP，RARP，BOOTP

**传输层协议**：TCP，UDP

**应用层协议**：TCP包括FTP,HTTP,TELNET,SMTP；UDP包括DNS，TFTP

**短连接**：建立连接，接收数据，关闭连接，适合用户较多，维护长连接或导致服务端资源耗尽的情况，如http服务

**长连接**：建立连接后不管是否使用都保持连接，适合那些对连接建立关闭时间敏感的应用，如数据库连接后一般不关闭

## OSI七层

**物理层**：通过物理设备传输比特（中继器、集线器、网关）

速记：RJ45等将数据转化成0和1。

**数据链路层**：可以看作是数据通道，为上层提供带有差错检测的不可靠的传输服务，将比特组装成帧和点到点的传递（网桥、交换机），PPP协议，点对点协议，用来用户计算机和服务器进行通信所用的协议。

速记：规定了0和1的分包形式，确定了网络数据包的形式。

**网络层**：负责数据报从源到目的地的传递和网际互联（路由器）

速记：可以理解为，此处需要确定计算机的位置。

**传输层**：负责端到端的报文传递（TCP、UDP）

速记：每一个应用程序都会在网卡注册一个端口号，该层就是端口与端口的通信！常用的（TCP/IP）协议

**会话层**：建立、管理和终止会话（会话协议数据单元SPDU）

速记：建立一个连接（自动的手机信息、自动的网络寻址）

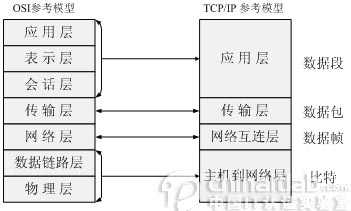
**表示层**：对数据进行翻译、加密和压缩（表示协议数据单元PPDU）

速记：可以理解为：解决不同系统之间的通信，eg：Linux下的QQ和Windows下的QQ可以通信

**应用层**：允许访问OSI环境的手段（HTTP, HTTPS, FTP, POP3, SMTP, Telnet等）

速记：规定数据的传输协议

## OSI模型与TCP/IP模型的对比



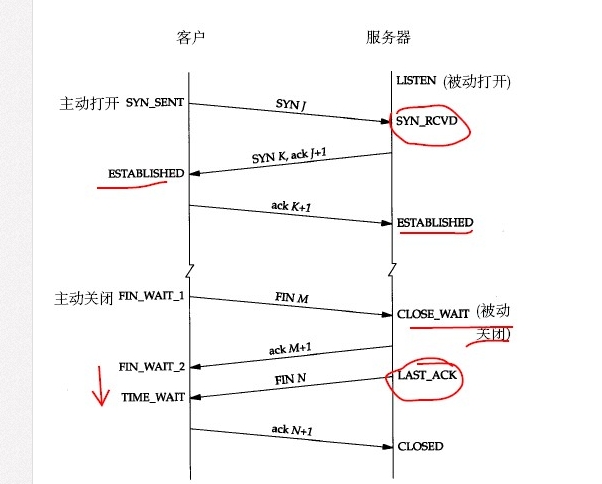
## TCP、UDP的区别

**TCP**：面向连接的可靠的数据传输，保证了报文段到达目的地，以及报文段到达的顺序。

**UDP**：无连接的数据传输协议，不保证数据报正确到达目的地，以及达到的顺序。

由于UDP不需要建立连接以及报文段确认，传输速度要好于TCP，但是当网络环境较差时，UDP丢包严重

## TCP状态转换



# 常见算法

## 蓄水池抽样

**背景**：从海量不确定数量的数据下等概率随机抽样，假设抽取K个数据。

**方法**：先选择前K个元素作为被选中的元素，然后后面数据以k/x的概率被选中，然后等概率（1/X）的替换掉被选中的元素，X表示元素的序号。

可以证明使用以上方法可以使得每个元素等概率的出现在蓄水池中（即若从N中随机选择K个元素，那么每个元素被选中的概率是K/N）。

## 海量数据找频繁前N个数据

若数据可以直接读入内存（十亿级别，即几个G），则可以使用hashmap方式统计出现次数，然后在堆中找出排在前面的N个最频繁数据。

若数据太大无法读入内存，则可以使用hash取模的方法，将大文件映射成不同的小文件，使得同样的数据不可能映射到不同的文件中，然后分别计算小文件中频繁N，最后归并这些小文件。

## 海量数据找第K大/中位数数据

多层划分的思想，将数据分散到多个区域中，若找第K大，则首先可以确定该数据位于哪一个区域，以及该区域的第几大数，然后再对该区域计算

## bitmap进行数据的判重以及集合求交集

如：给40亿不重复的数值，未排序，再给出一个值，判断这个值是否在这40亿个数中

思想：申请512M内存，读入40亿个数，再读入要查询的数，查看相应bit位是否为1,1则表示存在

## 找出数组中的第N大的数

利用快速排序的思想，一次快排后看基准值的下标就是第i大的数，如果i==n，则找到第N大，否则i大于n时，在左边继续找，i小于n时，在右边继续找。

# Tomcat

## tomcat处理http请求的过程

客户发送请求，根据请求的端口，被在该端口监听请求的Connector获得

Connector把请求交由该Connector所在service的engine处理，等待engine响应

engine获得请求，根据请求匹配它所拥有的所有虚拟主机host

host获得请求，开始匹配它所有拥有的context（应用程序）

context根据url pattern匹配对应的servlet，同时构造HttpServletRequest和HttpServletResponse对象，作为参数调用servlet的doget和dopost方法

context把执行完后的HttpServletResponse对象返回给engine

engine把HttpServletResponse对象返回给Connector

Connector把HttpServletResponse对象返回给客户端

## session和cookies的区别

session是存在服务器端，cookie是存储在客户端的，cookie分为会话cookie和持久化cookie，会话cookie存储在客户端浏览器的内存中，服务器通过会话cookie获取session的sessionid，浏览器关闭那么会话cookie也就消失了，所以sessionid也即消失了，但是服务端的session并没有消失了。

# 杂项

## 分布式与集群的区别

**分布式**（distributed）是指在多台不同的服务器中部署不同的服务模块，通过远程调用协同工作，对外提供服务。

**集群**（cluster）是指在多台不同的服务器中部署相同应用或服务模块，构成一个集群，通过负载均衡设备对外提供服务。

**分布式应用和服务**

将应用和服务进行分层和分割，然后将应用和服务模块进行分布式部署。这样做不仅可以提高并发访问能力、减少数据库连接和资源消耗，还能使不同应用复用共同的服务，使业务易于扩展。

**分布式静态资源**

对网站的静态资源如JS、CSS、图片等资源进行分布式部署可以减轻应用服务器的负载压力，提高访问速度。

**分布式数据和存储**

大型网站常常需要处理海量数据，单台计算机往往无法提供足够的内存空间，可以对这些数据进行分布式存储。

**分布式计算**

随着计算技术的发展，有些应用需要非常巨大的计算能力才能完成，如果采用集中式计算，需要耗费相当长的时间来完成。分布式计算将该应用分解成许多小的部分，分配给多台计算机进行处理。这样可以节约整体计算时间，大大提高计算效率。

## 分布式锁的redis实现

使用redis的SETNX命令，这个命令在key不存在的时候，返回1，存在的时候返回0，可以在并发的任务尝试设置key值，返回1则继续逻辑处理，返回0则考虑是阻塞还是循环尝试加锁，直至成功。

## zookeeper

zookeeper是一个分布式的、分层级的文件系统，保证分布式系统的最终一致性。可以用来构建可靠的分布式系统，提供的功能如领导人选举、配置服务等，还可以实现一些分布式数据结构，比如分布式锁、队列等。