Java整体知识架构详解-之进阶篇一

- 开发必备
  - 前端长连接
    - WebSocket
      - 最流行的web前端长连接技术
      - 可能遇到的问题
        - tomcat已经对WebSocket端点集成，所以在使用tomcat部署war包的时候，不需要加上ServerEndpointExporter实例；如果用springboot的jar包形式启动，忽略这个问题
        - 没过一段时间自动断开，这应该是它的空闲超时机制，当空闲时间超过一定值会自动断开，nginx的超时时间设置也会导致这个问题，可以采用定时重连，或者在close方法里进行断线重连
    - Netty
      - Netty也可以实现WebSocket长连接
  - Mybatis
    - 缓存
      - 一级缓存
        - mybatis的一级缓存是sqlSession级别的，它会创建一个私有的hashmap作为缓存容器
        - mybatis默认开启一级缓存
        - 在同一个sqlSession中，执行两次相同的查询sql，第二次会从缓存读取；
        - 在同一个sqlSession中，任何增删改都会导致缓存失效
        - 当一个sqlSession结束后，缓存也会清除
      - 二级缓存
        - mybatis二级缓存是mapper级别的，同一个mapper中，多个sqlSession共用同一个二级缓存区域
        - mybatis默认不开启二级缓存
        - 在同一个mapper中，不同的sqlSession两次执行相同的sql语句，第二次之后也从缓存中读取
        - 同一个mapper中，不同sqlSession执行任何增删改都会导致整个mapper二级缓存清除，所以对于存在大量增删改的数据表并不适合开启二级缓存
- 性能优化
  - MySQL性能优化
    - count(主键)往往是性能最优的
    - 索引失效的情况
      - 字符串不加单引号，会失效
      - 使用的索引列不是复合索引列表中的第一部分，会失效
      - 避免where子句中对字段进行null值判断，否则可能导致mysql放弃使用索引全表扫描

- 字段尽量加上默认值，比如0，''
- 尽量避免where子句中使用!=或<>操作符
- 避免where子句中使用or来连接条件（若or前的条件有索引，后面的列没有索引，那么涉及的索引都不会被用到），可以用union all来代替
- 避免在where条件中like以%打头
- 避免在where子句中对字段进行表达式操作
- 避免where子句中对=左边字段进行函数操作

- 开启mysql慢查询，配置慢查询时间，超过执行时间的语句将会记下来
- 需要事务处理的表使用innoDB，不需要的可以使用MyIsam，查询效率更高
- group by后面的字段会默认进行排序，如果不想有这个损耗，可以用order by null来避免
- 重复执行相同的sql语句，传入参数中条件最好用外部传入，这样可以使用到mysql自己的缓存，比如where条件里包含日期字段=curdate()，这个curdate()最好外部传入，否则mysql每次都会执行一遍
- SQL慢的原因
  - 偶尔慢
    - 数据库进行数据增删改操作会进行缓存，不会立即刷新到磁盘，当数据一直频繁增删改时，可能导致刷盘的频率加快，在刷新的时候会造成停顿延迟
    - 遇到表锁或行锁，需要等待锁的释放，偶尔会慢
  - 一直很慢
    - 字段加了索引，但没有命中索引，有可能是多字段联合索引，a,b,c的顺序，但条件只用到b，这时可能索引不起作用
    - 字段没加索引，数据量过大时造成全表查询，就慢了

- JVM调优
  - 堆设置主要设置初始堆内存，最大堆内存，年轻代和年老代比例，年轻代中Eden区和Survivor区的比值
  - 年轻代过小会导致Minor GC过于频繁，影响系统性能；年老代过小会导致内存溢出；所以要根据业务需求适当配比年轻代和老年代
  - 垃圾收集器也可以根据实际情况进行适当的选择，现在常用的为CMS（追求最短时间停顿）和G1收集器（适用大容量内存（大于4G））

- Web安全
  - SQL注入攻击
    - 避免直接拼接SQL，尽量使用成熟框架，比如MyBatis，xml中尽量不要使用$开头参数注入
    - 避免直接拼接SQL，尽量使用成熟框架，比如MyBatis，xml中尽量不要使用$开头参数注入
  - CSRF攻击
    - 概述
      - 跨站点请求伪造，CSRF攻击常常还伴随着XSS攻击，本质是伪造用户请求达到非本用户访问该用户才有权限的接口
    - 解决方案
      - 强制用户和服务器交互，填写验证码
      - 使用POST请求，更加不容易伪造，更易暴露

- XSS攻击
  - 概述
    - 脚本攻击，比如用户提交的评论内容里包含了一段脚本，提交后，打开该页面会自动运行该脚本，这就是没处理好XSS问题
  - 解决方案
    - 在服务请求的时候加一层过滤即可，网上方案很多，只要意识到这点很容易解决，像Spring Security也提供了快捷的解决方案
- DDOS攻击
  - 概述
    - 本质上是流量攻击，服务器带宽受不住就崩溃了，这也是最难防御的攻击
  - 改善方案
    - nginx限流，对同一个ip的大量请求进行限制
- 权限验证
  - Spring Security
    - 权限控制颗粒度较细，实现复杂
  - Shiro
    - 权限控制颗粒度较粗，实现简单
  - Oauth2.0
    - 接口授权协议，可以用spring-security-oauth2来实现
- 防盗链
  - 使用Refered请求头判断，是否来自自己的域名
  - nginx限制ip
- 跨域
  - 概述
    - 跨域本质上是浏览器自己的防护
  - 解决方案
    - 前端使用JSONP，只能是POST请求
    - 后端接口添加header请求头设置Access-Control-Allow-Origin
    - nginx域名转发
    - 现在常用的Springboot添加注解

- JDK8
  - Stream流
    - 概念
      - 一系列链式操作，中间操作返回流，直到终止操作结束，终止操作每调用中间操作是不会调用的
        - 例子
    - 流的创建
      - 类型
        - 集合
          - list.stream()

- list.parallelStream()
- 数组
  - Arrays.stream(int[])
- 数字流
  - IntStream，LongStream等
- 无限流
  - Random.ints().limit(int) 产生随机值
  - Stream.generate()
- 例子

流中间操作

- 无状态操作
  - map/mapToxx
    - 转换比如String转换成int不同类型
  - filterMap/filterMapToxxx
    - 深入对象里参数，比如对象中的集合，可以直接拿出
  - filter
    - 过滤器
  - peek
    - 类似foreach，只是这是中间操作
  - unordered
- 有状态操作
  - distinct
    - 去重
  - sorted
    - 排序
  - limit/skip
    - 限制/跳过
- 例子

流终止操作

- 非短路操作
  - forEach/forEachOrdered
  - collect/toArray
  - reduce
  - min/max/count
- 短路操作
  - findFirst/findAny
  - allMatch/anyMatch/noneMatch
- 例子

并行流

- parallel默认多线程

- 设置多个线程
- 用自己创建的线程并行操作
- 例子
- 收集器
  - 统计信息汇总
  - 分组
  - 分块
  - 分组后继续统计
  - 例子
- stream并行机制
  - 所有操作是链式调用，一个元素只迭代一次
  - 每一个中间操作返回一个新的流，流里面有一个属性sourceStage指向同一个地方，就是Head；Head -> nextStage -> nextStage ->...->null
  - 有状态操作（两个参数）会把无状态操作阶段单独处理
  - 并行环境下，有状态的中间操作不一定能并行操作
  - parallel/sequetial 这两个操作也是中间操作（也是返回stream），但是他们不创建流，他们只修改Head的并行标志
  - 例子
- lambda
  - 通过流来实现函数功能，比如
  - 接口的lambda表达式实现类，比如
  - 函数接口
    - 实现例子
    - 接口类型
      - Function<T,R>
        - 表示输入T，返回R
      - Predicate<T>
        - 输入T，返回类型Boolean
      - Consumer<T>
        - 输入T，没有返回值，消费完了就没了
      - Supplier<T>
        - 没有输入参数，返回T，属于类似生产者，提供数据
      - UnaryOperator<T>
        - 一元函数，输入输出类型相同
      - BinaryOperator<T>
        - 二元函数，输入两个相同类型的值，输入输出类型相同
      - BiFunction<T,U,R>
        - 两个输入参数T，U返回R
    - 引用传递
      - 代码例子

- 类型推断
  - 代码例子
- 变量引用
  - 代码例子
- 级联表达式和柯里化
  - 代码例子

- 日期
  - 日期格式化
    - 替换SimpleDateFormat
    - 这里特别提醒一点，不要想着把java8的LocalDate转换成Date，直接使用LocalDate替换你以前用的Date就行了，包括数据库对应的Date属性

- JDK9 Reactive Stream
  - 发布订阅
    - Publisher
    - Subscriber
    - 例子
  - 中间处理器
    - Processor
    - 例子
  - 运行机制
    - 可以调节生产者生产数据的速度，不至于消费者消费不过来，原理是订阅者缓冲池满了，发布者发布消息会阻塞，直到订阅者消费一条数据，发布者才能继续发送；而订阅者这边则是缓存了一部分请求在缓冲池，逐步拿取

- 并发编程
  - 多线程实现方式
    - 实现Runnable接口
      - 创建单个线程的常用方式，实现接口即可
    - 继承Thread
      - 通过继承方式实现的线程，由于java的单一继承性，这种方式有局限性
    - Callable+FutureTask
      - 一个线程执行有返回结果
    - Callable+ExecutorService+Future
      - 多线程执行有返回结果的任务
  - 线程池
    - 线程池参数
      - corePoolSize
        - 线程的核心线程数，当任务到来时，若创建的线程数量还未达到核心线程数，则会创建新的线程来处理
        - 当任务到来时，若创建的线程数大于等于核心线程数且小于最大线程数，则只有当workQueue满是才创建新的线程去处理任务

- 当corePoolSize和maximumPoolSize相同，则创建的线程池大小固定，判断workQueue未满则加入workQueue中等待空闲线程去处理
- 当运行的线程数等于maximumPoolSize，且workQueue已经满了，则通过handler所指定的策略来处理任务

maximumPoolSize
- 最大线程数

workQueue
- 等待队列，当任务提交时，线程池所有线程正忙，会将该任务加入worker队列中等待

KeepAliveTime
- 线程池维护线程所允许的空闲时间，这个时间针对的是超过corePoolSize数量的线程，它们执行完线程不会立即销毁，直到超过KeepAliveTime所指定的时间

threadFactory
- 用来创建线程，默认创建的线程拥有相同的优先级且是非守护线程，同时设置线程的名称

handler
- 概述
  - 该参数用来表示超过线程池执行任务限制的任务，我们所采取的策略，有四种实现
- 策略
  - AbortPolicy
    - 直接抛出异常，默认策略
  - CallerRunsPolicy
    - 用调用者所在的线程来执行任务
  - DiscardOldesPolicy
    - 丢弃阻塞队列中靠前的任务，并执行当前任务
  - DiscardPolicy
    - 直接丢弃任务

优点
- 降低资源消耗，重复利用已创建的线程降低线程创建和销毁造成的消耗
- 提高响应速度，任务到达是不需要等待线程创建就能立即执行
- 提高线程的可管理性，线程是稀缺资源不能无限制创建，通过线程池统一分配可以调优和监控

注意点
- shutdown和shutdownNow
  - shutdown会将正在执行的任务执行完后再关闭，没被执行的任务则**中断**
  - shutdownNow会中断现在执行着的任务，没被执行的任务则**返回**

线程知识点
- 线程中断interrupted
  - 调用interrupted并不是说该线程立即中断了，需要配置isInterrupted来使用
- ThreadLocal
  - 用空间换时间，表示把变量在每个线程存储一份进行操作

- NIO
  - 核心组成
    - Buffer
      - 一块缓存区，存储字节数组，维护一些变量实现数据反复利用
    - Channel
      - 双向可异步读写，数据读到缓存Buffer中
    - Selector
      - 通过单一线程控制一个Selector，并把请求数据转发到多个Channel通道进行传输
  - 概述
    - NIO操作面向缓冲区，数据从Channel读取到Buffer缓冲区，随后在Buffer中处理数据
- 原子操作
  - 基本类型
    - AtomicBoolean
    - AtomicInteger
    - AtomicLong
  - 数组
    - AtomicIntegerArray
    - AtomicLongArray
    - AtomicReferenceArray
  - 引用类型
    - AtomicReference
      - 包装对象进行cas操作
    - AtomicReferenceFieldUpdater
      - 对对象中的字段进行cas操作
- 并发工具类
  - CyclicBarrier
    - 栅栏，若设置5个栅栏表示当只有5条线程执行到await方法时才会继续执行
  - CountDownLatch
    - 计数器，使用await阻塞，当计数器减为0时，才可以唤醒线程继续执行下去
  - semphore
    - 信号量，有几个信号量就表示允许几个线程同时执行，超出的执行线程会阻塞
- 锁
  - 互斥锁
    - 比如去银行办业务，你需要取号，然后去等待区等待，此时你就是一个等待线程，而窗口前正在办业务的就是正在执行的线程，而窗口就是共享资源，同一时间只能有一条线程使用共享资源，你们是互斥的
  - 自旋锁
    - 比如你去银行取钱，门口有ATM机，你就可以直接去ATM前面排队等待，你需要时刻关注你前面还有没有人，当没有人了也就轮到你了，这就是自旋
  - 锁消除

- 还是去银行取钱，不是任何时候你都要去取号排队，当确保没人和你争抢窗口的时候，你可以直接坐上去办理业务，不需要取号，这就是锁消除，实际情况下何时会做这种优化呢？JIT编译器会判断当同步块所使用的锁对象只能够被一个线程访问时，会做优化
- 锁粗化
  - 还是拿银行取钱的例子吧，比如一个脑抽，它取号后等待取钱，轮到后取了1000元钱继续去取号，继续等待轮到后再取一笔1000元钱再去取号，一直重复了5次，那么它拿5000元的时间就被拖得很长了，你会想为什么不一次性取个5000呢，这就是代码中可能出现的问题，当JIT发现一系列连续的操作都对同一个对象反复加锁和解锁，甚至加锁操作出现在循环体中的时候，会将加锁同步的范围扩散（粗化）到整个操作序列的外部。看右边代码示例

- 算法
  - 海量数据处理问题
    - 分而治之/Hash映射+HashMap统计+堆/快速/归并排序
      - 例如海量日志提取访问百度次数最多的IP地址
    - 多层划分
    - Bloom filter/Bitmap
    - Trie树/数据库/倒排索引
    - 外排序
  - 排序算法
    - 排序算法类别
      - 选择排序
        - 稳定性
          - 不稳定
        - 时间复杂度
          - O(n2)
        - 概述
          - 先遍历原数组，取第一个数，和后续的数据对比，找到较小的数的位置，在继续和后续数据对比，直到找到最小的，和第一个数交换位置，再取第二个数重复上述步骤
      - 插入排序
        - 稳定性
          - 稳定
          - 稳定
        - 时间复杂度
          - O(n2)
          - O(n2)
          - O(n2)
        - 时间复杂度
        - 概述
          - 从左至右相邻两个数进行对比，后个数小往前移，不断往前两个两个对比，直到小的数比前一个数大，再回到开始时对比的两数继续往后两两对比

- 冒泡排序
  - 稳定性
    - 稳定
  - 时间复杂度
    - O(n2)
  - 概述
    - 遍历两两对比，将最大的数往后冒
- 快速排序
  - 稳定性
    - 不稳定
  - 时间复杂度
    - O(nlogn)
  - 概述
    - 随意从数组中抽个值作为基点A，第一个数位置记为i，最后一个数记为j，i向右移动，j向左移动，当碰到i对应的数大于基数A，停下i；当j对应的数碰到小于基数A停下，交换i，j对应的数；知道i，j碰到一起，判断碰到一起的数大于基数A，，且在A左侧则两者交换位置；以碰到位置将数组分成两组，执行相同的上述步骤，最后数组就是从小到大排列了
- 归并排序
  - 稳定性
    - 稳定
  - 时间复杂度
    - O(nlogn)
  - 概述
    - 分而治之的思想，先全部打散了，再用一个大数组一遍排序
- 堆排序
  - 稳定性
    - 不稳定
  - 时间复杂度
    - O(nlogn)
  - 通过最大堆顶的形式，每次堆排序的最大堆顶值和末尾倒数没和堆顶交换过的数交换，经过数次堆排序得出结果
- 希尔排序
  - 稳定性
    - 不稳定
  - 时间复杂度
    - O(nlogn)
  - 概述
    - 对插入排序的优化算法，通过调整步长进行对比交换
- 二叉树排序

- 稳定性
  - 稳定
- 时间复杂度
  - O(nlogn)
- 需要额外空间，树形结构的排列

- 计数排序
  - 稳定性
    - 稳定
  - 时间复杂度
    - O(n+k)
  - 需要额外存储空间，k为序列中Max-Min+1

- 桶排序
  - 稳定性
    - 稳定
  - 时间复杂度
    - O(n)
  - 概述
    - 需要O(k)额外存储空间

- 基数排序
  - 稳定性
    - 不稳定
  - 时间复杂度
    - O(n·k)
  - 概述
    - 需要O(n)额外存储空间（k为特征个数）

- 稳定性
  - 简单的说就是排序前后，值相同的两个数前后位置会不会倒过来，倒过来了就不稳定，还是原来的顺序就稳定

- UML
  - 概述
    - 架构师必备技能，画出实体间的耦合关系
  - 关联关系
    - 1. 单向关联
      - **单向不封闭空心箭头**表示，**has**的关系，比如顾客和他地址的关系，顾客有地址这属性
    - 2. 双向关系
      - **不带箭头直线**便是，双方各自持有对方类型的成员变量，比如顾客持有产品列表属性，而产品列表持有哪些顾客购买属性
    - 3. 自关联
      - **单向不封闭空心指向自身箭头**表示，也就是说自己**包含**自己类型的成员变量，比如链表节点，它包含的自己类型的下一个链表节点

- 4. 聚合关系
  - 从**空心菱形出发到不封闭空心的实线箭头**表示，强调整体**包含**部分，但部分可以脱离整体存在，比如汽车包含发动机，但发动机也是个实体可以单独存在
- 5. 组合关系
  - 从**实心菱形到不封闭空心的实线箭头**表示，这里的部分不能脱离整体，强调**has**的概念，比如嘴是头的一部分，不能脱离头单独存在
- 6. 依赖关系
  - **虚线箭头**表示，表示依赖对象才可执行，方法需要传入**依赖**的对象，比如说司机开车的动作，需要依赖车这个对象
- 7. 继承关系
  - 对象继承extends后面的类为被指对象，继承者**封闭空心实线箭头**指向被继承对象
- 8. 接口实现关系
  - implement关键字后方接口为被指向对象，实现对象用**虚线空心箭头**指向接口