java面试笔记

# java基础

## 基本数据类型和引用数据类型

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 数据类型 | 大小 | 默认值 | 范围 |  |  |
| byte | 1K | 0 | -128-127 |  |  |
| short | 2K | 0 | -32768-32768 |  |  |
| int | 4K | 0 |  |  |  |
| long | 8K | 0L |  |  |  |
| float | 4K | 0F |  |  |  |
| double | 8K | 0 |  |  |  |
| char | 2K | “” |  |  |  |
| boolean | 1K | false |  |  |  |

## 常见java集合

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 类型 | 具体类型 | 顺序 | null | 线程 | 元素重复 | 备注 |
| Set | HashSet | 无序 | 可以 | 不安全 | 不重复 | 基于HashMap实现 |
| LinkedHashSet | 有序 |  |  |  | LinkedHashSet是HashSet的子类 |
| TreeSet | 有序 |  |  |  | 要求元素实现comparable接口或传入comparator； |
| List | ArrayList | 有序 |  | 不安全 | 可重复 | 插入效率低，访问效率高 |
| Vector | 有序 |  | 安全 |  |  |
| LinkedList | 有序 |  | 不安全 |  | 插入效率高，随机访问效率低 |
| Map | HashMap | 无序 | key=null | 不安全 |  | 数组+链表+红黑树实现，采用哈希表来存储 |
| HashTable | 无序 | key!=null | 安全 |  |  |
| TreeMap | 有序 |  |  |  | TreeMap默认升序，基于红黑树实现 |
| LinkedHashMap | 有序 |  |  |  | LinkedHashMap记录插入顺序 |
| 总结 |  |  |  |  |  |  |

## 类的实例化顺序：

父类静态变量-父类静态代码块-子类静态变量-子类静态代码块-父类非静态变量-父类构造函数-子类非静态变量-子类构造函数。

## 抽象类和接口：

1、抽象类和接口都不能直接实例化，如果要实例化，抽象类变量必须指向实现所有抽象方法的子类对象，接口变量必须指向实现所有接口方法的类对象。   
2、抽象类要被子类继承，接口要被类实现。   
3、接口只能做方法申明，抽象类中可以做方法申明，也可以做方法实现   
4、接口里定义的变量只能是公共的静态的常量，抽象类中的变量是普通变量。   
5、抽象类里的抽象方法必须全部被子类所实现，如果子类不能全部实现父类抽象方法，那么该子类只能是抽象类。同样，一个实现接口的时候，如不能全部实现接口方法，那么该类也只能为抽象类。   
6、抽象方法只能申明，不能实现。abstract void abc();不能写成abstract void abc(){}。   
7、抽象类里可以没有抽象方法   
8、如果一个类里有抽象方法，那么这个类只能是抽象类   
9、抽象方法要被实现，所以不能是静态的，也不能是私有的。   
10、接口可继承接口，并可多继承接口，但类只能单根继承

## 反射

## 代理

## final关键字

1. final关键字可以用于成员变量、本地变量、方法以及类。
2. final本地变量必须声明时赋值；final成员变量必须在声明时初始化或构造器中初始化。
3. final变量不能再次赋值，final方法不能被重写，final类不能被继承
4. 在匿名类中所有变量都必须是final变量。
5. 接口中声明的所有变量本身是final的。
6. final和abstract这两个关键字是反相关的，final类就不可能是abstract的。
7. final方法在编译阶段绑定，称为静态绑定，故final方法比非final方法要快
8. 按照Java代码惯例，final变量就是常量，而且通常常量名要大写：

## 访问修饰符

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 修饰符 | 同一个类 | 同一个包 | 不同包子类 | 不同包非子类 |
| private | true |  |  |  |
| default | true | true |  |  |
| protected | true | true | true |  |
| public | true | true | true | true |

## 异常

|  |  |  |
| --- | --- | --- |
| 异常类型 |  | 备注 |
| java.lang.Exception | Checked异常 | 处理方法：try...catch块/抛出 |
| Runtime异常 | ArithmeticException、ArrayStoreExcetpion、ClassCastException、IndexOutOfBoundsException、NullPointerException |
| java.lang.Error |  | 由Java虚拟机生成并抛出，包括动态链接失败，虚拟机错误等。程序对其不做处理 |

## JVM

|  |  |
| --- | --- |
| 垃圾回收算法 | 原理 |
| 标记—清除法 | 标记阶段，标记所有要回收的对象。清除阶段紧随标记阶段，将标记阶段确定的对象清除。  基础的收集算法，标记和清除阶段的效率不高，而且清除后会产生大量的不连续空间 |
| 复制算法 | 把内存分成大小相等的两块，每次使用其中一块，实现简单，运行效率高，但内存利用率不高 |
| 标记—整理法 | 基于标记—清除法，但会把存活对象往内存的一端移动，然后直接回收边界以外的内存 |
| 对象死亡算法 | **原理** |
| 引用计数器 | 给每个对象设置计数器，当引用的时候，计数器+1，当引用失效时，计数器-1，计数器为0时，JVM就认为对象不再被使用，是“垃圾”了 |
| 可达性分析 | GC Roots对象作为起点，从这些节点开始往下搜索，搜索通过的路径成为引用链，当一个对象没有被GC Roots的引用链连接的时候，说明这个对象是不可用的 |
| 垃圾收集器 | **原理** |
| Serial | 新生代收集器，单线程，使用复制算法；进行垃圾收集时，必须暂停其他所有的工作线程 |
| Serial Old | 老年代收集器，单线程，使用“标记－整理”算法 |
| ParNew | 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，其余行为与Serial收集器一样 |
| Parallel Scavenge | 新生代收集器，使用复制算法，并行多线程；吞吐量优先 |
| Parallel Old | Parallel Scavenge收集器的老年代版本，使用多线程和“标记－整理”算法 |
| CMS | 基于“标记—清除”算法，响应优先，分代 |
| G1 | 基于“标记—整理”算法，分区 |

## 多线程

## 零碎知识点

|  |
| --- |
| 零碎知识点： |
| equals相等两个对象，则hashcode一定要相等。但是hashcode相等的两个对象不一定equals相等 |
|  |
|  |
|  |
|  |
|  |
|  |

# 设计模式

## 单例模式

https://my.oschina.net/dyyweb/blog/609021

# 数据结构和算法

# 框架

## Struts2 vs SpringMVC

|  |  |  |
| --- | --- | --- |
|  | SpringMVC | Struts2 |
| 实现方式 | 采用Servlet（DispatcherServlet） | 采用Filter（StrutsPrepareAndExecuteFilter） |
|  | SpringMVC是方法级别的拦截，一个方法对应一个Request上下文 | Struts2框架是类级别的拦截，每次请求就会创建一个Action，一个Action对应一个request，response上下文 |
| 性能 | 零配置，基于方法的拦截，加载一次单例模式bean注入，效率高 | 类级别的拦截，每次请求对应实例一个新的Action，需要加载所有的属性值注入，效率低 |
| 拦截机制 | 用的是独立的Aop方式 | 有自己的拦截Interceptor机制 |

## Hibernate vs MyBatis

|  |  |  |
| --- | --- | --- |
|  | Hibernate | MyBatis |
| 优点 | 1、hibernate是全自动，hibernate完全可以通过对象关系模型实现对数据库的操作，拥有完整的JavaBean对象与数据库的映射结构来自动生成sql。  2、功能强大，数据库无关性好，O/R映射能力强，需要写的代码很少，开发速度很快。  3、有更好的二级缓存机制，可以使用第三方缓存。  4、数据库移植性良好。  5、hibernate拥有完整的日志系统，hibernate日志系统非常健全，涉及广泛，包括sql记录、关系异常、优化警告、缓存提示、脏数据警告等 | 1、易于上手和掌握，提供了数据库查询的自动对象绑定功能，而且延续了很好的SQL使用经验，对于没有那么高的对象模型要求的项目来说，相当完美。  2、sql写在xml里，便于统一管理和优化， 解除sql与程序代码的耦合。  3、提供映射标签，支持对象与数据库的orm字段关系映射  4、 提供对象关系映射标签，支持对象关系组建维护  5、提供xml标签，支持编写动态sql。  6、速度相对于Hibernate的速度较快 |
| 缺点 | 1、学习门槛高，精通门槛更高，程序员如何设计O/R映射，在性能和对象模型之间如何取得平衡，以及怎样用好Hibernate方面需要的经验和能力都很强才行  2、hibernate的sql很多都是自动生成的，无法直接维护sql；虽然有hql查询，但功能还是不及sql强大，见到报表等变态需求时，hql查询要虚，也就是说hql查询是有局限的；hibernate虽然也支持原生sql查询，但开发模式上却与orm不同，需要转换思维，因此使用上有些不方便。总之写sql的灵活度上hibernate不及mybatis。 | 1、关联表多时，字段多的时候，sql工作量很大。  2、sql依赖于数据库，导致数据库移植性差。  3、由于xml里标签id必须唯一，导致DAO中方法不支持方法重载。  4、对象关系映射标签和字段映射标签仅仅是对映射关系的描述，具体实现仍然依赖于sql。  5、DAO层过于简单，对象组装的工作量较大。  6、不支持级联更新、级联删除。  7、Mybatis的日志除基本记录功能外，其它功能薄弱  8、编写动态sql时,不方便调试，尤其逻辑复杂时。  9、提供的写动态sql的xml标签功能简单，编写动态sql仍然受限，且可读性低。 |
|  |  |  |

## 框架原理

|  |
| --- |
| MyBatis原理： |
| 初始化过程：  首先会创建SqlSessionFactory建造者对象，然后由它进行创建SqlSessionFactory。这里用到的是建造者模式；  XMLConfigBuilder对象会进行XML配置文件的解析，实际为configuration节点的解析操作；  在configuration节点下会依次解析properties/settings/.../mappers等节点配置。  在解析environments节点时，会根据transactionManager的配置来创建事务管理器，根据dataSource的配置来创建DataSource对象，这里面包含了数据库登录的相关信息。  在解析mappers节点时，会读取该节点下所有的mapper文件，然后进行解析，并将解析后的结果存到configuration对象中  解析完MyBatis配置文件后，configuration就初始化完成了，然后根据configuration对象来创建SqlSession，到这里时，MyBatis的初始化完成；  查询过程：  创建sqlSession的过程其实就是根据configuration中的配置来创建对应的类，然后返回创建的sqlSession对象。调用selectOne方法进行SQL查询，selectOne方法最后调用的是selectList，在selectList中，会查询configuration中存储的MappedStatement对象，mapper文件中一个sql语句的配置对应一个MappedStatement对象，然后调用执行器进行查询操作  执行器在query操作中，优先会查询缓存是否命中，命中则直接返回，否则从数据库中查询  真正的doQuery操作是由SimplyExecutor代理来完成的，该方法中有2个子流程，一个是SQL参数的设置，另一个是SQL查询操作和结果集的封装  子流程1 SQL查询参数的设置：首先获取数据库connection连接，然后准备statement，然后就设置SQL查询中的参数值。打开一个connection连接，在使用完后不会close，而是存储下来，当下次需要打开连接时就直接返回  子流程2 SQL查询结果集的封装：ResultSetWrapper是ResultSet的包装类，调用getFirstResultSet方法获取第一个ResultSet，同时获取数据库的MetaData数据，包括数据表列名、列的类型、类序号等，这些信息都存储在ResultSetWrapper类中了。然后调用handleResultSet方法来来进行结果集的封装 |

# 数据库

## InnoDB vs MyISAM

|  |  |
| --- | --- |
| InnoDB | MyISAM |
| 支持事务处理等 不加锁读取 支持外键 支持行锁 不支持FULLTEXT类型的索引 不保存表的具体行数，扫描表来计算有多少行 DELETE 表时，是一行一行的删除 InnoDB 把数据和索引存放在表空间里面 跨平台可直接拷贝使用 InnoDB中必须包含AUTO\_INCREMENT类型字段的索引 表格很难被压缩 | 不支持事务，回滚将造成不完全回滚，不具有原子性 不支持外键 支持全文搜索 保存表的具体行数  DELETE 表时，先drop表，然后重建表 MyISAM 表被存放在三个文件 。frm 文件存放表格定义。 数据文件是MYD (MYData) 。 索引文件是MYI (MYIndex)引伸 跨平台很难直接拷贝 MyISAM中可以使AUTO\_INCREMENT类型字段建立联合索引 表格可以被压缩  索引文件和数据文件是分离的，索引文件仅保存数据记录的地址 |

## 事务的隔离级别

|  |  |
| --- | --- |
| 隔离级别 |  |
|  |  |
|  |  |
|  |  |

## 删除比较

|  |  |  |
| --- | --- | --- |
| delete | truncate | drop |
| 每次从表中删除一行，同时将该行的删除操作作为事务记录在日志中,以便进行进行回滚;  不会减少表或索引所占用的空间;  事务提交后才生效，会触发触发器  断点自增 | 一次性地从表中删除所有的数据并不把单独的删除操作记录记入日志保存，删除行是不能恢复的。并且在删除的过程中不会激活与表有关的删除触发器。执行速度快;  表和索引所占空间会恢复初始大小  DDL(数据定义),执行后会自动提交  truncate无条件  自增重新计数 | 将表所占用的空间全释放掉;  触发器/索引/依赖于该表的存储过程/函数将被保留，但其状态会变为：invalid  DDL(数据定义),执行后会自动提交 |