目录

[1. Java基础： 4](#_Toc69765274)

[1.1. JDK和JRE： 4](#_Toc69765275)

[1.2. 什么是OOP: 4](#_Toc69765276)

[1.3. 重载和重写的区别： 4](#_Toc69765277)

[1.4. Java三大特性： 5](#_Toc69765278)

[1.5. String、Stringbuffer、StringBuilder的区别： 5](#_Toc69765279)

[1.6. 用final修饰的类还有哪写？ 5](#_Toc69765280)

[1.7. final关键字的用法： 5](#_Toc69765281)

[1.8. finally的用法： 5](#_Toc69765282)

[1.9. 接口和抽象类的区别： 6](#_Toc69765283)

[1.10. 成员变量与局部变量的区别： 6](#_Toc69765284)

[1.11. 静态方法与实例方法区别： 6](#_Toc69765285)

[1.12. ==与equals()区别： 6](#_Toc69765286)

[1.1.3. 包装类与常量池; 6](#_Toc69765287)

[1.14. 装箱和拆箱的过程： 7](#_Toc69765288)

[2. java集合 8](#_Toc69765289)

[2.1. 集合与数组的区别： 8](#_Toc69765290)

[2.2. ArrayList、LinkedList、Vector 的区别： 8](#_Toc69765291)

[2.3. HashSet、TreeSet的区别： 8](#_Toc69765292)

[2.4. comparable 和 comparator的区别？ 8](#_Toc69765293)

[2.5. HashSet去重原理： 8](#_Toc69765294)

[2.6. hashCode和equals() 8](#_Toc69765295)

[2.7. HashMap、HashTable、TreeMap的区别： 9](#_Toc69765296)

[2.8. ConcurrentHashMap 和 Hashtable 的区别 9](#_Toc69765297)

[2.9. ArrayList的扩容机制： 9](#_Toc69765298)

[2.10. 为什么 ArrayList 的 elementData 加上 transient 修饰 9](#_Toc69765299)

[2.11. HashMap扩容： 9](#_Toc69765300)

[2.12. HashMap原理： 10](#_Toc69765301)

[2.13. hashMap如何解决hash冲突 10](#_Toc69765302)

[2.14. HashMap为什么不直接使用hashCode()直接作为table的下标？ 10](#_Toc69765303)

[2.15. HashMap 的长度为什么是2的幂次方 10](#_Toc69765304)

[2.16. HashMap为什么是是不安全的/死锁问题： 10](#_Toc69765305)

[3. 多线程并发： 15](#_Toc69765306)

[3.1. 说说并发与并行的区别? 15](#_Toc69765307)

[3.2. 线程的基本状态： 15](#_Toc69765308)

[3.3. Java序列化化中，有些字段不想被序列化，怎么办？ 15](#_Toc69765309)

[3.4. 程序计数器、虚拟机栈为什么是线程私有的 15](#_Toc69765310)

[3.5 Sleep和wait方法的区别及共同点： 15](#_Toc69765311)

[3.6. 为什么调用start()方法执行run()方法，而不能直接调用run() 15](#_Toc69765312)

[3.7. 多线程一定比单线程要快么？什么时候使用多线程？谈谈多线程的了解 16](#_Toc69765313)

[3.7. 项目中使用多线程栗子： 16](#_Toc69765314)

[3.8. 什么是上下文切换? 16](#_Toc69765315)

[3.9. 为什么使用线程池： 16](#_Toc69765316)

[3.10. 如何创建线程池： 16](#_Toc69765317)

[3.11. execute()与submit()的使用区别： 16](#_Toc69765318)

[3.12. 使用多线程可能带来什么问题?如何保证线程安全？ 17](#_Toc69765319)

[3.13. 说说线程的生命周期和状态? 17](#_Toc69765320)

[3.15. 说一说自己对于 synchronized 关键字的了解： 17](#_Toc69765321)

[3.16. Synchronized修饰的代码/代码块异常时： 17](#_Toc69765322)

[3.17. 谈谈 synchronized和ReentrantLock 的区别 17](#_Toc69765323)

[3.18. Volatile关键字的作用 17](#_Toc69765324)

[3.19. 双重校验锁的单例模式： 17](#_Toc69765325)

[3.20. 说说 synchronized 关键字和 volatile 关键字的区别： 17](#_Toc69765326)

[3.21. ThreadLocal： 18](#_Toc69765327)

[3.22. ThreadLocal 内存泄露问题： 18](#_Toc69765328)

[3.23. 谈谈强引用、软引用、弱引用、虚引用区别 18](#_Toc69765329)

[4. dubbo/ActiveMQ 19](#_Toc69765330)

[4.1. 微服务、分布式、集群概念： 19](#_Toc69765331)

[4.2. 服务注册和发现： 19](#_Toc69765332)

[4.3. 注册中心全部宕机了dubbo服务还能调用吗？ 19](#_Toc69765333)

[4.4. dubbo集群容错模式： 19](#_Toc69765334)

[4.5. dubbo负载均衡策略： 19](#_Toc69765335)

[4.6. dubbo的协议： 20](#_Toc69765336)

[4.7.有用过MQ么： 20](#_Toc69765337)

[4.8. ActiveMQ消息确认机制： 20](#_Toc69765338)

[4.9. ActiveMQ消息失败怎么办： 21](#_Toc69765339)

[5. Mysql： 22](#_Toc69765340)

[5.1. 什么是事务： 22](#_Toc69765341)

[5.2. 事务的四大特性： 22](#_Toc69765342)

[5.3. MyISAM和InnoDB区别： 22](#_Toc69765343)

[5.4. 并发事务带来的问题： 22](#_Toc69765344)

[5.5. 事务的隔离级别： 22](#_Toc69765345)

[5.6. 大表优化： 23](#_Toc69765346)

[5.7. 分库分表之后,id 主键如何处理？ 23](#_Toc69765347)

[5.8. 三大范式： 23](#_Toc69765348)

[5.9. in 和 exists 区别： 23](#_Toc69765349)

[5.10. SQL优化： 23](#_Toc69765350)

[5.11. 唯一索引和联合索引的区别： 24](#_Toc69765351)

[6. Spring相关 25](#_Toc69765352)

[4.1. 什么IOC 25](#_Toc69765353)

[4.2. 什么是AOP： 25](#_Toc69765354)

[4.3. 动态代理： 25](#_Toc69765355)

[4.4. Spring IOC容器初始化过程： 25](#_Toc69765356)

[4.5. 设计模式： 25](#_Toc69765357)

[4.5.1.单例模式： 25](#_Toc69765358)

[4.5.2.代理模式： 26](#_Toc69765359)

[4.5.3.装饰者模式： 26](#_Toc69765360)

[4.6. Spring中Bean的作用域： 26](#_Toc69765361)

[4.7. Spring中单例Bean的线程安全问题： 26](#_Toc69765362)

[4.8. @Transactional的原理： 26](#_Toc69765363)

[4.9. @Transactional的失效场景： 26](#_Toc69765364)

[7. Redis 33](#_Toc69765365)

[5.1. 什么是redis，redis的类型数据，应用场景？ 33](#_Toc69765366)

[5.2. 为什么要用 redis/为什么要用缓存: 33](#_Toc69765367)

[5.3. 为什么要用 redis 而不用 map/guava 做缓存? 33](#_Toc69765368)

[5.4. Redis 为什么早期版本选择单线程？ 33](#_Toc69765369)

[5.5. Redis 为什么这么快？ 33](#_Toc69765370)

[5.6. redis 和 memcached 的区别： 33](#_Toc69765371)

[5.7. 一个字符串类型的值能存储最大容量是多少？ 33](#_Toc69765372)

[5.8. Redis的批量处理： 34](#_Toc69765373)

[5.9. redis 过期键的删除策略？ 34](#_Toc69765374)

[5.10. redis 内存淘汰机制： 34](#_Toc69765375)

[5.11. 什么是缓存雪崩？有哪些解决办法？ 34](#_Toc69765376)

[5.12. 什么是缓存穿透？有哪些解决办法？ 34](#_Toc69765377)

[5.13. 布隆过滤器： 35](#_Toc69765378)

[5.14. 如何解决 Redis 的并发竞争 Key 问题： 35](#_Toc69765379)

[5.15. 如何保证缓存与数据库双写时的数据一致性? 35](#_Toc69765380)

[5.16. Redis持久化方式：快照RDB和AOF 35](#_Toc69765381)

[5.17. RDB优点和缺点： 35](#_Toc69765382)

[5.18. AOF优缺点： 35](#_Toc69765383)

[5.19. AOF流程： 35](#_Toc69765384)

[8. 易宝外派华为的外包： 39](#_Toc69765385)

[A. 反射机制有什么作用，优缺点 39](#_Toc69765386)

[B. HashMap和HashTable的区别 39](#_Toc69765387)

[C. ArrayList和LinkList的区别 39](#_Toc69765388)

[D. 类加载顺序 39](#_Toc69765389)

[E. 什么是IOC以及应用场景 39](#_Toc69765390)

[F. 什么是AOP以及应用场景 39](#_Toc69765391)

[G. SpringMVC的流程 39](#_Toc69765392)

[H. 多线程的创建方式 39](#_Toc69765393)

[I. 垃圾回收机制的算法 39](#_Toc69765394)

[J. Java的内存区域 39](#_Toc69765395)

[K. SpringBoot的核心注解 39](#_Toc69765396)

[L. Zookeeper的读写机制： 40](#_Toc69765397)

[M. Jdk1.8的新特性有哪些 40](#_Toc69765398)

[N. SQL方面的优化 40](#_Toc69765399)

[O. 索引的类型： 40](#_Toc69765400)

[P. 联合索引最左原则： 40](#_Toc69765401)

[Q. 索引失效的场景： 40](#_Toc69765402)

[R. Join连接中on条件里的and和where条件里的and的区别 40](#_Toc69765403)

[9. 中软外派华为的外包： 40](#_Toc69765404)

[A. SSM架构的优缺点： 40](#_Toc69765405)

[B. 建表规范： 41](#_Toc69765406)

[C. PO和VO: 41](#_Toc69765407)

[PO是持久层对象，属性对应数据库的字段 41](#_Toc69765408)

[VO是值对象，和PO差不到，只是属性可以不是数据库的字段 41](#_Toc69765409)

[D. Controller的主要注解: 41](#_Toc69765410)

[E. Restful接口： 41](#_Toc69765411)

[F. 依赖注入和控制反正有什么关系： 41](#_Toc69765412)

[G. Oracle的分页： 41](#_Toc69765413)

[H. 多表的关联查询: 41](#_Toc69765414)

[I. 组函数有哪些： 41](#_Toc69765415)

[J. 组合性的查询： 41](#_Toc69765416)

[K. String、数组、集合获取长度的方法 41](#_Toc69765417)

[L. String、StringBuilder、StringBuffer的区别： 42](#_Toc69765418)

[M. 常用的设计模式： 42](#_Toc69765419)

[N. 单例模式时，通过new，会怎么样 42](#_Toc69765420)

[O. 表单如何防止重复提交 42](#_Toc69765421)

[P. 前后端分离跨域问题解决 42](#_Toc69765422)

1. Java基础：

1.1. JDK和JRE：

JRE就是java运行时环境，包括JVM、java类库等等，可以运行已编译的java程序。

JDK拥有JRE拥有的一切，此外还有javac编译器和一些根据，能够创建和编译程序。

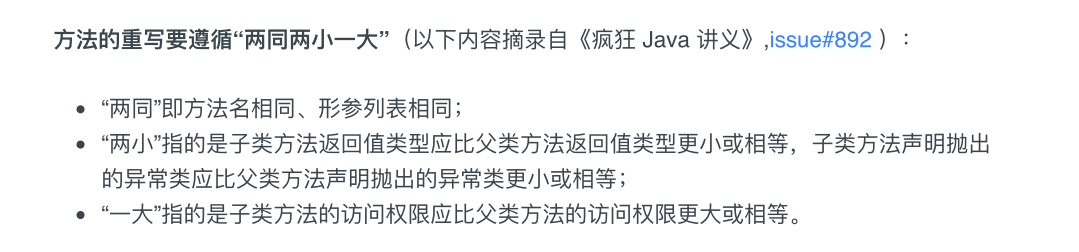
1.2. 什么是OOP:

OOP就是面向对象编程，java将事物抽象出来，比如一个人，他的性别、姓名等等属性都将它抽象到一个person类里，方便我们使用；面向对象具有封装、继承、多态三大特性，使用时更灵活，更易于维护。

1.3. 重载和重写的区别：

重载就是类中方法名一样，但参数类型，个数，位置，不一样；像类的构造方法就是很好的示例。

重写发生在实现接口或者继承抽象类后，重写实现的接口方法或抽象方法。



1.4. Java三大特性：

封装：就是把对象的属性私有化，比如人的性别，年龄等，将这些属性封装到实体类里。

继承：在已有类的基础上新建的类，已有类又叫父类，新类叫子类；特点--①子类拥有父类的所有属性和方法（包括私有属性和私有方法），但子类无法访问私有属性和私有方法。②子类可以拥有自己的属性和方法，即对父类进行扩展；③子类可以重写父类的方法；④一个子类只能继承一个父类，但能多重继承。

多态：一个接口/抽象方法，多个实现方法；具体调用哪个类的实现方法，在程序运行时才能决定。多态的表现—继承和接口方法的重写

1.5. String、Stringbuffer、StringBuilder的区别：

String和StringBuffer都是线程安全的，而StringBuilder是线程不安全的；因为String是final修饰的，而StringBuffer的方法是同步的。

String是不可变的，每次修改对应的是引用地址的变化；而StringBuffer、StringBuilder是可变的，可以对自身进行修改

1.6. 用final修饰的类还有哪写？

String还有包装类都是final修饰的

1.7. final关键字的用法：

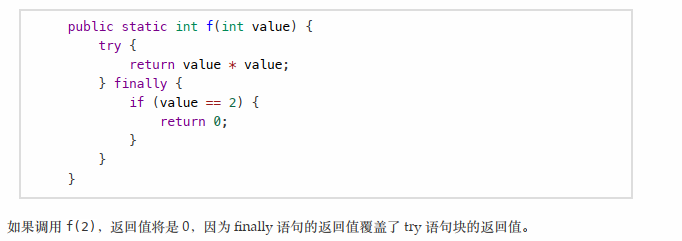
修饰类，表示该类不能被继承，类中方法隐式指定final

修饰方法，表示改方法不能被重写

修饰基本类型变量，则该变量值不可变；如果修饰引用类型变量，则引用地址不可变

1.8. finally的用法：

当try和finally中都有return语句时，在方法返回前finally语句内容将被执行，并覆盖try中的返回值。



1.9. 接口和抽象类的区别：

接口是用interface修饰的，抽象类是用abstract修饰的

接口中的变量只能用static、final修饰；抽象类没有这个限制

接口中不能有实现（java8之后可以有default实现），但抽象类可以有普通方法

接口方法是用public修饰的，抽象方法还可以用protected、default修饰

一个类可以实现多个接口，但只能继承一个抽象类

1.10. 成员变量与局部变量的区别：

成员变量属于类/实例，存在堆中，局部变量属于方法，存在栈中

成员变量可以有修饰符，局部变量只能final修饰

成员变量会自动赋默认值，局部变量不会

成员变量随对象创建而存在，局部变量随方法调用完毕而消失

1.11. 静态方法与实例方法区别：

静态方法属于类，访问时类.方法；实例方法属于对象，访问时对象.方法

静态方法只能访问静态属性和静态方法；实例方法没有限制

1.12. ==与equals()区别：

==判断对象地址是否相等，基本类型比较的是值，引用类型比较的是内存地址

没有重写equals时，等价于==；

重写了equals，比较内容是否相等，相等返回true

1.1.3. 包装类与常量池;

Java 基本类型的包装类的大部分都实现了常量池技术

Byte,Short,Integer,Long默认创建数值范围为[-128，127]

Character创建了数值在[0,127]范围的缓存数据

Boolean 直接返回True Or False

如果超出对应范围仍然会去创建新的对象

Integer i1 = 33;

Integer i2 = 33;

System.out.println(i1 == i2);// 输出 true

Integer i11 = 333;

Integer i22 = 333;

System.out.println(i11 == i22);// 输出 false

Double i3 = 1.2;

Double i4 = 1.2;

System.out.println(i3 == i4);// 输出 false

Integer i4 = new Integer(40);

Integer i5 = new Integer(40);

Integer i6 = new Integer(0);

System.out.println("i4=i5+i6 " + (i4 == i5 + i6)); //true

System.out.println("40=i5+i6 " + (40 == i5 + i6)); //true

i5 和 i6 进行自动拆箱操作，进行数值相加，即 i4 == 40。然后 Integer 对象无法与数值进行直接比较，所以 i4 自动拆箱转为 int 值 40，最终这条语句转为 40 == 40 进行数值比较。

1.14. 装箱和拆箱的过程：

装箱：好比integer=10，装箱实际上是调用静态方法valueOf()，如果int值在[-128，127]，则会从常量池中获取，如果不在，则直接new Integer()

拆箱：通过调用initValue()来实现拆箱

1. java集合

2.1. 集合与数组的区别：

数组长度是固定的，集合长度是可变的

数组存储的元素类型必须是同一个数据类型；集合可以储存不同的数据类型

数组可以储存基本数据类型，也可以储存引用数据类型；集合只能储存引用数据类型（基本类型数据在储存的时候自动进行装箱，所以也能储存进去）

2.2. ArrayList、LinkedList、Vector 的区别：

ArrayList和Vector底层实现都是object数组，支持随机访问，所以能够快速获取元素，但增删就会比较慢

而LinkedList是双向链表，不支持随机访问，所以增删比较快，获取指定元素比较慢

此外Vector是同步的，是线程安全的，效率会比ArrayList低。

LinkedList和ArrayList是不同步的，是线程不安全的

2.3. HashSet、TreeSet的区别：

HashSet底层实现是hashMap，是无序不允许重复的，允许有null

TreeSet底层实现是红黑树，是有序不允许重复的，也不运允许有null；有序是指其支持两种排序方式，实现Comparable接口的自然排序和Comparator的定制排序，默认是自然排序。

2.4. comparable 和 comparator的区别？

comparable接口是出自java.lang包，它有一个compareTo(Object obj)方法用来排序

comparator接口是出自java.util 包，它有一个compare(obj1, obj2)方法用来排序

2.5. HashSet去重原理：

当hashSet.add()的对象的时候，先判断该对象的hashcode和其他对象的是否一样，若不一样，则该对象add成功；若一样，则调用equals()来判断是否是同一个对象，如果二者相同，则add失败。

add的对象必须重写hashCode()和equals()方法。因为每new一次，对象的hashCode是不一样的。

2.6. hashCode和equals()

两个对象有相同的hashcode，它们不一定相等

两个对象相等，则hashcode一定相同

2.7. HashMap、HashTable、TreeMap的区别：

HashMap线程不同步，允许 key 和 Vale 是 null，但是只允许一个 key 为 null，且这个元素存放在哈希表 0 角标位置。

HashTable是线程安全的，它是遗留类，不推荐使用。现在可以使用 ConcurrentHashMap 来支持线程安全，并且 ConcurrentHashMap 的效率会更高，因为 ConcurrentHashMap 引入了分段锁。

TreeMap的Key就是一个TreeSet，是无序不可重复的

2.8. ConcurrentHashMap 和 Hashtable 的区别

JDK1.7的 ConcurrentHashMap 底层采用 分段的数组 链表 实现，JDK1.8 采用数据结构跟HashMap1.8的结构一样，数组 链表/红黑二叉树；Hashtable采用 数组 链表的形式。

线程安全实现方式的区别：ConcurrentHashMap采用note+CAS+synchronize的方式；Hashtable采用synchronzied

2.9. ArrayList的扩容机制：

如果新建ArrayList时不指定大小，会自动分配一个空的数组

当第一次调用add方法时，当前数组容量和默认容量之间取最大值，作为最小扩容容量

然后在内部的grow方法中，进行扩容，第一次add的时候，会将arrayList的容量直接扩到默认容量10。

未超过容量时add，不会触发grow方法扩容

当超过当前数组容量时add，触发grow方法，将容量扩容到原来的1.5倍。

2.10. 为什么 ArrayList 的 elementData 加上 transient 修饰

用transient修饰的变量，在序列化的时候不会被序列化

底层数组并非每一个元素都有值，那些空的元素就不必序列化，用transient修饰，能加快序列化的速度，减少序列化文件大小

ArrayList实现了 Serializable 接口，并重写了writeObject方法

在序列化的时候先调用 defaultWriteObject() 方法序列化 ArrayList 中的非 transient 变量，然后遍历elementData，序列化已存入的元素。

2.11. HashMap扩容：

创建时未指定初始容量，在第一次put的时候，进行初始化扩容，默认容量16，此时扩容阈值12（16\*0.75）；当超过阈值后，会再次进行扩容，每次扩容容量为原来的2倍，阈值也为原来的2倍

当指定初始化容量时，在构造方法中会将容量转换为2的幂次方大小，比如21转换成32；第一次put时，采用转换后的容量，此时阈值为0.75\*容量；当超过阈值时，就会扩容。

2.12. HashMap原理：

当put数据时，通过hash(Key)来计算出数据存放在node数组中的下标位置，

如果当前位置没有数据，则直接保存K,V数据；

如果当前位置已存在数据（hash冲突时），会去判断Key值是否相等

相等，则直接替换原来的Value；否则就尾插成链表

当链表长度大于8时，若数组长度小于64，则先进行扩容；否则就链表转红黑树

链表转红黑树后，如果数组节点hash冲突，直接插入树的新节点

所以，hashMap的底层结构是【数组+链表+红黑树】

2.13. hashMap如何解决hash冲突

使用链地址法来链接拥有相同hash()值得数据

底层的hash方法对key的hashcode进行了两次扰动，降低hash冲突概率

引入红黑树降低遍历时间

2.14. HashMap为什么不直接使用hashCode()直接作为table的下标？

减少碰撞，进一步降低hash冲突的几率

2.15. HashMap 的长度为什么是2的幂次方

为了尽量把数据分配均匀，减少hash冲突

2.16. HashMap为什么是是不安全的/死锁问题：

HashMap在jdk1.8之前多线程同时进行put操作，并且同时进行扩容的时候可能会出现环形链表，导致死循环的发生。

1.8后在链表转换树或者对树进行操作的时候会出现线程安全的问题。

HashMap源码(jdk1.8)

1.变量：

**static final int *DEFAULT\_INITIAL\_CAPACITY*** = 1 << 4**;** // 默认容量16

**static final int *MAXIMUM\_CAPACITY*** = 1 << 30**;** // 最大容量

**static final float *DEFAULT\_LOAD\_FACTOR*** = 0.75f**;** // 默认负载因子0.75

**static final int *TREEIFY\_THRESHOLD*** = 8**;** // 链表转红黑树阈值8

**int** threshold**;** // 扩容边界值

**transient** Node<K**,**V>[] table**;** // 存放数据的数组(哈希桶数组)。

2.hashMap构造函数：

a.指定容量和负载因子。

**public** HashMap(**int** initialCapacity**, float** loadFactor) **{  
 if** (initialCapacity < 0)  
 **throw new** IllegalArgumentException("Illegal initial capacity: " +initialCapacity)**;  
 if** (initialCapacity > ***MAXIMUM\_CAPACITY***)  
 initialCapacity = ***MAXIMUM\_CAPACITY*;  
 if** (loadFactor <= 0 || Float**.***isNaN*(loadFactor))  
 **throw new** IllegalArgumentException("Illegal load factor: " +loadFactor)**;  
 this.*loadFactor*** = loadFactor**;  
 this.**threshold = *tableSizeFor*(initialCapacity)**;  
}**// *tableSizeFor(initialCapacity)*使容量变为2的幂次方，比如：

initialCapacity=7，threshold=8

initialCapacity=11，threshold=16

b.指定容量,负载因子默认

**public** HashMap(**int** initialCapacity) **{  
 this**(initialCapacity**, *DEFAULT\_LOAD\_FACTOR***)**;  
}**

c.不指定容量，负载因子默认

**public** HashMap() **{  
 this.*loadFactor*** = ***DEFAULT\_LOAD\_FACTOR*;** // all other fields defaulted  
**}**

3.hashMap底层结构：

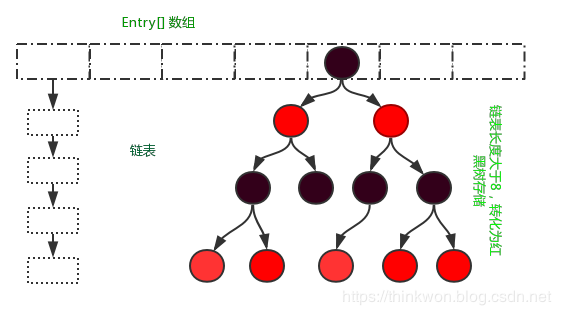
HashMap底层结构：**Node<K,V>[]数组+链表+红黑树**

hashMap.put(K，V);每put一次数据，就是往数组里存数据

put进去的数组位置i，由*hash*(key)决定。tab[i = (n - 1) & hash]

**static final int** hash(Object key) **{  
 int** h**;  
 return** (key == **null**) ? 0 : (h = key**.**hashCode()) ^ (h >>> 16)**;  
}**

**//key的hashCode与其膏16位做异或 ；扰动处理 = 2次扰动 = 1次位运算 + 1次异或运算**



4.put操作过程及扩容时机：

**public** V put(K key**,** V value) **{  
 return** putVal(*hash*(key)**,** key**,** value**, false, true**)**;  
}**

putVal具体方法看源码，有点长。

a.场景1

HashMap<String**,** Object> map = **new** HashMap<>()**;**map**.**put("key"**,**"66666")**;**

1).通过构造函数，负载因子默认0.75

2).判断初始化数组大小

**if** ((tab = table) == **null** || (n = tab**.*length***) == 0)  
 n = (tab = resize())**.*length*;**

//初次put，table为null，对底层数组进行扩容

//扩容后hashMap数组容量16，扩容边界值12（=16\*0.75）；n=16

resize() ：

newCap = ***DEFAULT\_INITIAL\_CAPACITY*;**newThr = (**int**)(***DEFAULT\_LOAD\_FACTOR*** \* ***DEFAULT\_INITIAL\_CAPACITY***)**;**

3).确定put进去的元素存放哈希桶数组Node<K**,**V>[] table哪个位置（桶）

**if** ((p = tab[i = (n - 1) & hash]) == **null**)  
 tab[i] = newNode(hash**,** key**,** value**, null**)**;**

//根据扰动后的hash值和数组长度，确认存放位置

//如果当前位置为null（hash不冲突），直接储存数据

++modCount**;  
if** (++size > threshold)  
 resize()**;**afterNodeInsertion(evict)**;  
return null;**

//put进去成功，size加1，判断是否超过扩容边界值，超过则扩容

//当put第13次时，满足条件，进行扩容；hashMap数组容量32，扩容边界值24

//0001 0000 :16

//0010 0000 :32

4).hash冲突处理：

①桶中第一个元素(数组中的结点)的hash值相等，key相等

**else {** Node<K**,**V> e**;** K k**;  
 if** (p**.*hash*** == hash &&  
 ((k = p**.*key***) == key || (key != **null** && key**.**equals(k))))  
 e = p**;**

直接替换相同key的value值

**if** (e != **null**) **{** // existing mapping for key  
 V oldValue = e**.**value**;  
 if** (!onlyIfAbsent || oldValue == **null**)  
 e**.**value = value**;** afterNodeAccess(e)**;  
 return** oldValue**;  
 }**

②桶中第一个元素(数组中的结点)的hash值相等，key不相等，加在链表

**else {  
 for** (**int** binCount = 0**; ;** ++binCount) **{  
 if** ((e = p**.**next) == **null**) **{** p**.**next = newNode(hash**,** key**,** value**, null**)**;  
 if** (binCount >= ***TREEIFY\_THRESHOLD*** - 1) // -1 for 1st  
 treeifyBin(tab**,** hash)**;  
 break;  
 }  
 if** (e**.*hash*** == hash &&  
 ((k = e**.*key***) == key || (key != **null** && key**.**equals(k))))  
 **break;** p = e**;  
 }**

**// e = p.next 数组节点的下一个链表节点，如果为null，直接将数据存储在该链表节点**

**//如果不为null,判断该链表节点的key是否与存入的相等，相等同上述，直接替换key的value值**

**//如果key不相等，循环下一链表节点**e = p**.**next，**重复上述判断**

**//当链表长度**binCount >= ***TREEIFY\_THRESHOLD*** – 1 ；**大于转红黑树阈值8，执行转红黑树操作**

**final void** treeifyBin(Node<K**,**V>[] tab**, int** hash) **{  
 int** n**,** index**;** Node<K**,**V> e**;  
 if** (tab == **null** || (n = tab**.*length***) < ***MIN\_TREEIFY\_CAPACITY***)  
 resize()**;  
 //如果数组长度小于64，会先扩容，否则才会转红黑树**

③桶中第一个元素(数组中的结点)的hash值相等，key不相等，加在红黑树

**else if** (p **instanceof** TreeNode)  
 e = ((TreeNode<K**,**V>)p)**.**putTreeVal(**this,** tab**,** hash**,** key**,** value)**;**

**//****链表转红黑树后，如果数组节点hash冲突，直接插入树的新节点**

1. 多线程并发：

3.1. 说说并发与并行的区别?

并发：就是同一时间段，访问同一个功能。比如双十一零点的下单情况。

并行：就是单位时间内，多个任务同时执行。好比同时吃饭，同时打电话。

3.2. 线程的基本状态：

New：初始状态，线程创建，未start()之前

Runnable：运行状态，调用start()后，线程开始运行，处于运行状态

WAITING：等待状态，调用wait()后，线程进入等待状态，需要依靠其他线程通知才能回到runnable状态

TIME\_WAITING：超时等待状态：在等待基础上增加超时限制，达到超时时间后，自动回到runnable状态

BLOCKED：阻塞状态，线程调用同步方法，在没有获得锁的情况，会进入等待状态

TERMINATED：终止状态，线程执行完毕后，进入终止状态。

3.3. Java序列化化中，有些字段不想被序列化，怎么办？

使用transient修饰，transient只能修饰变量，不能修饰类和方法

transient作用：阻止实例中用此关键字修饰的变量序列化，当对象反序列化时，被transient修饰的变量不会被持久化和恢复。

3.4. 程序计数器、虚拟机栈为什么是线程私有的

程序计数器主要是记录各个线程正在执行字节码指令的地址，线程私有是为了线程切换后恢复到正确的位置。

虚拟机栈的栈帧储存着局部变量表，操作数栈等，为了保证线程中的局部变量不被别的线程访问到，所以是私有的。

3.5 Sleep和wait方法的区别及共同点：

主要区别： sleep()没有释放锁，wait()释放了锁

sleep()是Thread的方法，wait()是Object的方法

sleep(time),线程到时会苏醒，wait()，需要调用notify()方法唤醒

二者都可以是线程进入等待状态

3.6. 为什么调用start()方法执行run()方法，而不能直接调用run()

调用run(),是在主线程中调用了普通方法，并没有执行多线程。

start()会启动线程，然后自动执行run()的内容，这才是多线程工作

3.7. 多线程一定比单线程要快么？什么时候使用多线程？谈谈多线程的了解

当任务多，且任务执行耗时较长，我们可以使用多线程去处理任务，提高系统并发能力

创建线程的方式有四种，通过继承Thread类、实现Runnable或者Callable接口，还有就是通过线程池来使用多线程；

但并不是多线 程一定比单线程要快，多线程会存在上下文切换，频繁切换会影响性能

3.7. 项目中使用多线程栗子：

保单项目中，采用多线程来处理的保单，被保人的保单数据保存在List里，通过多线程处理每个保单的保额，最后累计获得累计的保额。线程任务实现Callable接口，返回值在Future对象里

3.8. 什么是上下文切换?

CPU为每个线程分配CPU时间片来实现多线程的，如果在时间片内没执行完线程任务，那么会保存当前任务状态，然后切换别的任务执行；下次被切换回来的时候就能够加载该任务状态。所以，任务保存到再加载的过程就是上下文切换。

3.9. 为什么使用线程池：

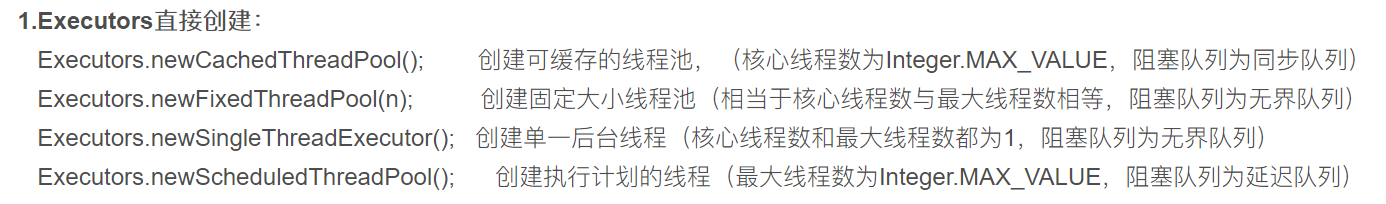
线程池和数据库连接池一样，这种池化技术无非就是用空间换时间。

避免创建线程和销毁线程的开销，降低资源消耗，提高响应速度。

3.10. 如何创建线程池：

通过new ThreadPoolExecutor来创建

也可以通过Executors来创建



3.11. execute()与submit()的使用区别：

execute(Runnable command)：执行的任务实现Runnable接口，提交任务的没有返回值

submit()：执行的任务可以是Runnable接口，也可以是Callable接口；提交的Callable接口的任务会返回Future对象，通过该对象的get方法可以获取返回值。

execute会直接打印异常信息

submit会将异常信息存储在返回future对象里，只有get的时候，才会出现异常信息。

3.12. 使用多线程可能带来什么问题?如何保证线程安全？

线程安全、死锁、内存泄漏等问题

通过synchronized/Lock加锁、使用线程安全的类（JUC并发包的类），比如原子类AtomicInteger

3.13. 说说线程的生命周期和状态?

线程的状态主要有六个，初始状态（NEW）、运行状态（RUNNABLE）、等待状态（WAITING）、超时等待状态（TIME\_WAITING）、阻塞状态（BLOCKED）以及终止状态（TERMINATED）

3.15. 说一说自己对于 synchronized 关键字的了解：

Synchronized保证线程安全，保证它修饰的方法或代码块在任意时刻只有一个线程执行

synchronized修饰静态方法和代码块，会给类加锁；修饰实例方法，会给实例对象加锁。

3.16. Synchronized修饰的代码/代码块异常时：

会释放锁，并打印异常，别的线程可以去竞争锁

3.17. 谈谈 synchronized和ReentrantLock 的区别

两者都是可重入锁：可以获取自己的内部锁

synchronized 依赖于 JVM 而 ReentrantLock 依赖于 API

synchronized 会自动释放锁，Lock要手动释放锁。

ReentrantLock 比 synchronized 增加了一些高级功能，比如实现公平锁

3.18. Volatile关键字的作用

volatile 可以禁止 JVM 的指令重排，还可以保证变量的可见性

用 volatile 修饰 long 和 double 变量，使其能按原子类型来读写

3.19. 双重校验锁的单例模式：

<https://blog.csdn.net/weixin_43901882/article/details/108913317>

3.20. 说说 synchronized 关键字和 volatile 关键字的区别：

volatile关键字只能用于变量，synchronized关键字可以修饰方法以及代码块

多线程访问volatile关键字不会发生阻塞，而synchronized关键字可能会发生阻塞

3.21. ThreadLocal：

ThreadLocal类主要为每个线程绑定自己的值，往ThreadLocal里Set变量值，最终变量是放在ThreadLocalMap里头。Key为线程ThreadLocal对象。

3.22. ThreadLocal 内存泄露问题：

ThreadLocalMap 中使用的 key 是弱引用的,而 value 是强引用。

ThreadLocal 没有被外部强引用的情况下，在垃圾回收的时候，key 会被清理掉，而 value 不会被清理掉。所以使用完 ThreadLocal方法后 最好手动调用remove()方法，会清除key 为 null 的记录。

3.23. 谈谈强引用、软引用、弱引用、虚引用区别

强引用：JVM的内存不足时，宁愿OOM也不愿意回收具有强引用的对象

软引用：如果内存足够，GC不会回收它；如果内存不足，就会回收这些对象的内存

弱引用：一旦发现具有弱引用的对象，不管当前内存空间是否足够，都会回收它的内存

虚引用：在任何时候都可能被GC回收

1. dubbo/ActiveMQ

4.1. 微服务、分布式、集群概念：

微服务：就是分散功能

将一个单体web应用拆分多个模块，每一个模块就是一个独立的微服务；

微服务之间通过rpc/webservice进行交互

分布式：就是分散部署，将拆分的模块分别部署在不同的机器

集群：就是同一模块部署多个机器

微服务往往是分布式部署的，但分布式部署的应用不一定是微服务架构的

比如集群部署，它是把相同应用复制到不同服务器上，但是逻辑功能上还是单体应用

4.2. 服务注册和发现：

采用dubbo+zookeeper做服务注册和发现：

Zookeeper作为服务注册中心，服务提供的接口注册到zookeeper上，消费者就能从注册中心调用（订阅）接口

此外还可以用springboot+eureka做服务注册和发现

4.3. 注册中心全部宕机了dubbo服务还能调用吗？

能。

服务消费者在启动时，注册中心会返回提供者地址列表给消费者，消费者会将列表缓存到本地，所以就算注册中心都宕机了，dubbo服务还能通过本地缓存通讯

4.4. dubbo集群容错模式：

<dubbo:service cluster="failsafe" />

Failover：失败自动切换，默认配置，当出现失败，重试其它服务器。

Failfast：快速失败，只发起一次调用，失败立即报错

Failsafe：失败安全，出现异常时，直接忽略

Failback：失败自动恢复，后台记录失败请求，定时重发

Forking：并行调用多个服务器，只要一个成功即返回

Broadcast：广播调用所有提供者，逐个调用，任意一台报错则报错。

4.5. dubbo负载均衡策略：

<dubbo:service interface="..." loadbalance="roundrobin" />

Random：随机调用，默认配置

RoundRobin：轮询调用

LeastActive：最少活跃调用数模式，慢的提供者收到更少请求

ConsistentHash：一致性 Hash模式，相同参数的请求总是发到同一提供者

4.6. dubbo的协议：

<dubbo:protocol name="dubbo" port="20880" />

<dubbo:protocol name="rmi" port="1099" />

dubbo：默认协议，采用单一长连接和NIO异步通讯，适合数据小，并发大的服务调用

此外，还支持rmi、http、rest、redis、webservice、hessian等9种协议

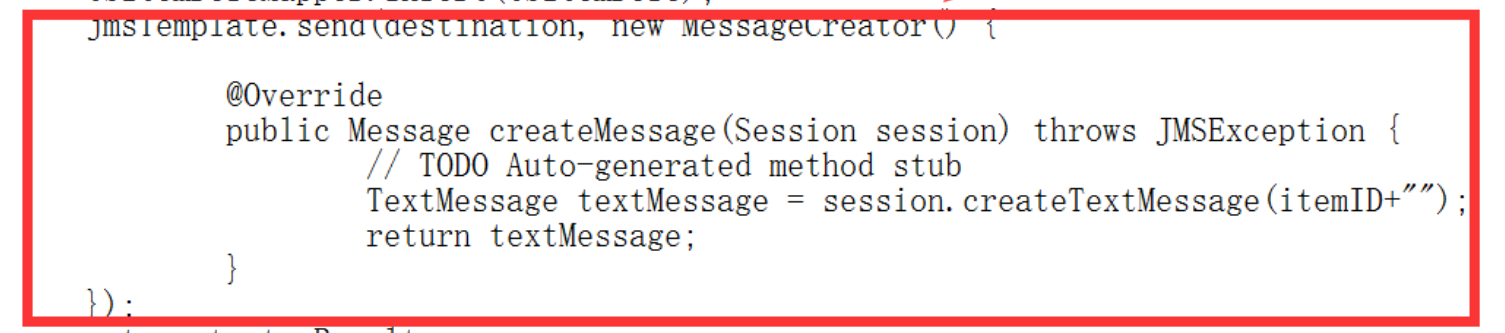
4.7.有用过MQ么：

项目中没怎么用过，但写过简单的demo；

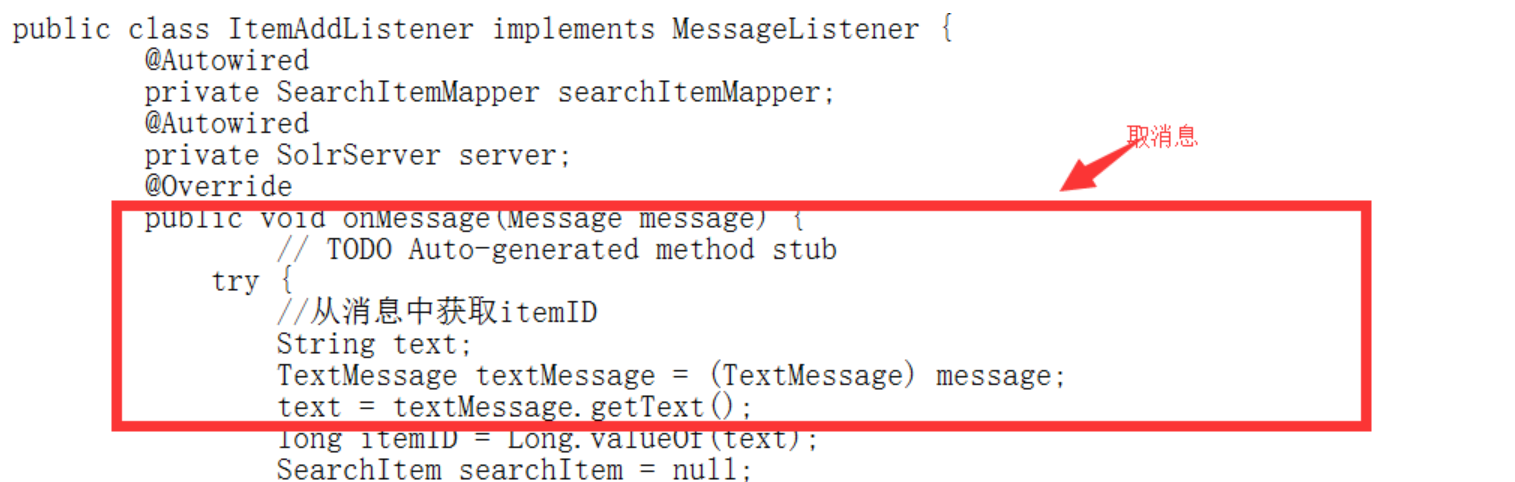
生产者通过jmsTemplate来发送消息到指定队列destination，消费者监听这个队列，有数据则取出消息数据去处理。

<https://www.cnblogs.com/loong-hon/p/9661946.html>

发送消息：jmsTemplate.send(destination,Message)



监听类实现MessageLitscener接口，重新写onMessage方法



4.8. ActiveMQ消息确认机制：

监听容器中，有个sessionAcknowledgeMode 属性，可以配置5个值

AUTO\_ACKNOWLEDGE = 1 ：自动确认

CLIENT\_ACKNOWLEDGE = 2：客户端手动确认

DUPS\_OK\_ACKNOWLEDGE = 3： 自动批量确认

SESSION\_TRANSACTED = 0：事务提交并确认

INDIVIDUAL\_ACKNOWLEDGE = 4：单条消息确认

4 需要手动确认消息，也就是手动触发message.acknowleged()方法



4.9. ActiveMQ消息失败怎么办：

可以设置失败重试，默认失败6次就放入死信队列

如果消息允许失败丢失，则设置指定失败重试次数即可

如果消息不允许丢失，可以将失败的消息持久化到数据库，然后再定时取出来执行

1. Mysql：

5.1. 什么是事务：

事务是逻辑上的一组操作，要么都执行，要么都不执行

5.2. 事务的四大特性：

原子性：DML操作要么都执行，要么不执行

一致性：事务执行前后，数据保持一致

隔离性：并发访问数据库，各并发事务之间保持独立

持久性：数据一旦提交，永久保存在数据库

5.3. MyISAM和InnoDB区别：

MyISAM是MySQL的默认数据库引擎（5.5版之前），不支持行级锁，不提供事务支持，不支持外键，破溃后无法安全恢复。

MySQL 5.5版本后默认的存储引擎为InnoDB。默认行级锁，提供事务支持，外键、崩溃安全恢复以及MVCC。

5.4. 并发事务带来的问题：

脏读：读取到别的事务还没有提交的数据

不可重复读：一个事务内多次读同一数据，这个事务还未结束，另一事物修改了数据，使得前一个事务查询不一致。

幻读：一个事务内多次读同一数据，这个事务还未结束，另一个事务添加或删除(行)数据，使得前一个事务查询不一致（多了或少了行）。

丢失修改：事务1读取某表中的数据A=20，事务2也读取A=20，事务1修改A=A-1，事务2也修改A=A-1，最终结果A=19，事务1的修改被丢失。

5.5. 事务的隔离级别：

READ-UNCOMMITTED (读取未提交): 最低的隔离级别，允许读取尚未提交的数据变更，可能会导致脏读、幻读或不可重复读。

READ-COMMITTEE D(读取已提交)：允许读取并发事务已经提交的数据，可以阻止脏读，但是幻读或不可重复读仍有可能发生。

REPEATABLE-READ(可重复读)： 对同一字段的多次读取结果都是一致的，除非数据是被本身事务自己所修改，可以阻止脏读和不可重复读，但幻读仍有可能发生。

SERIALIZABLE(可串行化)： 最高的隔离级别，完全服从ACID的隔离级别。所有的事务依次逐个执行，这样事务之间就完全不可能产生干扰，也就是说，该级别可以防止脏读、不可重复读以及幻读。

| **隔离级别** | **脏读** | **不可重复读** | **幻读** |
| --- | --- | --- | --- |
| READ-UNCOMMITTED | √ | √ | √ |
| READ-COMMITTED | × | √ | √ |
| REPEATABLE-READ | × | × | √ |
| SERIALIZABLE | × | × | × |

MySQL InnoDB 存储引擎的默认支持的隔离级别是 REPEATABLE-READ（可重读）

InnoDB 存储引擎在分布式事务情况下一般会用到 SERIALIZABLE(可串行化) 隔离级别。

5.6. 大表优化：

限定数据范围：禁止不带任何限制数据范围条件的查询语句

读/写分离：经典的数据库拆分方案，主库负责写，从库负责读；

垂直拆分：数据表列的拆分，把一张列比较多的表拆分为多张表

水平拆分：水平拆分是指数据表行的拆分，水平拆分最好分库 。

5.7. 分库分表之后,id 主键如何处理？

分成多个表之后，每个表都是从 1 开始累加，这样是不对的，我们需要一个全局唯一的 id 来支持。可以使用UUID、数据库自增 id 、redis 生成 id等等。

5.8. 三大范式：

第一范式：每个列都不可以再拆分。

第二范式：要求每个表只描述一件事情

第三范式：表的各列必须和主键直接相关

5.9. in 和 exists 区别：

exists是将外表的值传递给内表，内表查询到数据，就将外表当前行数据保留，依次循环判断。一般内表数据比较多的时候用exists，内表数据比较少时用in

5.10. SQL优化：

查看执行计划，看是否使用了索引，还有就是SQL方面的优化，比如：

避免使用SELECT \*，而是直接select 具体字段

能用union all 就不用union

where 后面避免使用!=，也要避免字段进行函数操作。

5.11. 唯一索引和联合索引的区别：

当表中只有一个主键时，他是惟一索引

当表中有多个主键时，称为联合主键，两盒主键保证唯一的索引

联合索引要保证最左原则

联合索引(A,B,C)，如果A索引没生效，那么B索引也不生效

联合索引(A,B,C)，相当于创建了（A）、(A,B)和(A,B,C)三个索引

1. Spring相关

4.1. 什么IOC

IOC就是控制反转，就是将对象的控制权交给Spring去管理，创建对象是不用手动去New，而是直接从容器中获取。

4.2. 什么是AOP：

AOP就是面向切面编程，由动态代理实现的，可以对原业务进行增强，而不需要改变原有逻辑代码

需要增强的方法定义为切点（@PointCut），通过@Before、@After、@Around来对切点方法进行增强处理。

4.3. 动态代理：

动态代理实现有jdk动态代理和cglib动态代理

Jdk动态代理：

只能对实现接口的类代理

代理类实现invocationHandle接口

通过Proxy.newProxyInstance()方法来创建代理类

Cglib动态代理：

能够直接对类进行代理

4.4. Spring IOC容器初始化过程：

资源定位：根据配置文件路径找到配置文件

载入：解析配置文件

注册：将解析后的内容放入容器

4.5. 设计模式：

设计模式有23种，符合开闭原则，也就是对扩展开发，对修改关闭

常见的设计模式有单例模式、工厂模式、代理模式、装饰器模式

4.5.1.单例模式：

单例模式就是确保一个类只有一个实例

它的实现有懒汉式、饿汉式

如果需要保证线程安全，可以使用双重校验的单例模式或者静态内部类的方式。

单例模式的要点是要将类的构造函数私有化，此次对外提供一个公共方法获取对象。

4.5.2.代理模式：

代理类对原有真实对象进行代理，可执行原有业务并做出增强处理。比如AOP

Jdk动态代理：

代理类实现invocationHandle接口

通过Proxy.newProxyInstance()方法来创建代理类

Cglib动态代理：能够直接对类进行代理

4.5.3.装饰者模式：

装饰者模式会有个抽象的装饰类，通过对抽象装饰类进行实现，可以保留原有对象功能，对对象扩展一些新的功能。

与直接继承相比，他能够有多种组合的实现方式。

4.6. Spring中Bean的作用域：

单例：singleton | 原型：prototype | request | session | global-session

4.7. Spring中单例Bean的线程安全问题：

Spring中默认创建的Bean是单例的，这是线程不安全的；

解决：如果Bean允许使用原型，则使用原型bean

4.8. @Transactional的原理：

声明式注解事务，是Spring AOP的体现之一，也就是说它也是级域动态代理实现的

Spring创建Bean时，会将切面类生成代理对象，它注解的事务相当于切点，对象调用事务方法，实际是代理对象调用，在调用之前、之后会执行一些增强操作

比如调用之前创建连接开启事务，调用之后提交事务或回滚

4.9. @Transactional的失效场景：

注解的方法为非Public方法时，会失效，因为生成不了代理对象

没有注解修饰的方法A调用了有注解的方法B，方法B异常时不会回滚

注解的方法catch了运行时异常，但没有手动抛异常/回滚，事务不会自动回滚

抛出受检异常，比如IO异常，但没有指定rollbackFor属性，事务也不会回滚

Spring事务只会发生威捕获的运行时异常才会回滚

类加载

类加载就是将编译好的字节码文件，加载到运行时内存，并对加载的数据进行校验、解析、初始化，最终形成能被虚拟机直接使用的java类型。

生命周期：

加载、连接（验证、准备、解析）、初始化、使用、卸载

加载和连接可以说是异步的。

加载：

通过类的的完全限定名获取该类二进制字节流

将字节流的静态储存结果转换为方法区的运行时数据结构

内存中生成一个代表改类的class对象。

验证：

验证字节码文件的字节流信息符不符合虚拟机要求，比如文件格式验证、元数据验证、字节码验证等。

准备：

该阶段正式为类变量（static修饰的变量）分配内存，并设置类变量的初始值（默认值，比如int 为0，boolean为false），但如果类变量被final修饰，该直接为类变量设置常量值。

该阶段不为实例变量分配内存，实例变量在对象实例化时随对象一起分配到Java堆中。

解析：

改阶段将常量池中的符号引用替换为直接引用的过程。

符号引用就是一组符号来描述目标

直接引用就是直接指向目标的指针、相对偏移量或一个间接定位到目标的句柄

初始化：

初始化是类加载的最后一步，也是真正执行类中定义的 Java 程序代码(字节码)，初始化阶段是执行类构造器 <clinit> ()方法的过程。

类加载顺序：

父类静态-子类静态

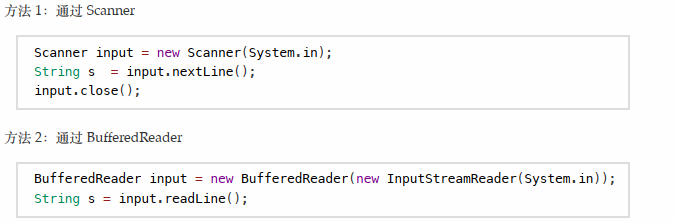
父类非静态-父类构造-子类非静态-子类构造

均自上而下执行。

双亲委派：

双亲委派的意思是如果一个类加载器需要加载类，那么首先它会把这个类请求委派给父类加载器去完成，每一层都是如此。一直递归到顶层，当父加载器无法完成这个请求时，子类才会尝试去加载。保证了程序的稳定性，避免重复

获取键盘输入值的常用方法：



Java的IO流分类：

按流的流向分：有输入流和输出流—InputStream/OutputStream

按操作单位分：有字节流和字符流—Reader/Writer

按流的功能分：有节点流和处理流—

JVM内存区域

JVM

JVM就是java虚拟机，是运行java字节码文件的虚拟机。

JVM有针对不同系统的特定实现，只要字节码文件相同，执行结果就相同。

JAVA内存区域：

Jdk1.8之前，JVM内存区域分为线程共有的堆、方法区；线程私有的程序计数器、虚拟机栈、本地方法栈

Jdk1.8之后，JVM去除了方法区，取而代之的是元空间。

程序计数器：

程序计数器相当于当前线程执行字节码文件的行号指示器。主要记录当前线程执行的位置，使线程切换回正确位置；还有就是实现代码的流程控制，如：顺序执行、选择、循环、异常处理。

虚拟机栈：

虚拟机栈是描述java方法执行的内存模型，每个方法执行都会创建一个栈帧，每个栈帧包含了局部变量表、操作数栈、动态链接、等信息。

每次调用方法都会创建一个栈帧，每个栈帧是先进后出。栈帧在虚拟机的入栈到出栈，就是Java方法从调用到执行完成的过程

局部变量表存放了编译期可知的基本类型数据（byte、shot、int、long、boolean、char、float、double）。对象引用（reference类型，可能是指向一个对象地址的引用指针，也可能是一个对象的句柄）和 returnAddress 类型（指向了一条字节码指令的地址）。

会有栈溢出和OOM溢出

本地方法栈：

如同虚拟机栈在虚拟机执行java方法服务一样，本地方法栈在虚拟机使用native方法服务。也会有会有栈溢出和OOM溢出。

堆：

Java堆就是用来存放对象实例的，该区域也是GC所管理的主要区域。

Jdk1.7之前java堆可分为年轻代、老年代和永久代，

Jdk1.8之后，永久代就去除了，取而代之是元空间，使用直接内存。

年轻代再细分可分为Eden、From survivo、To survivo空间。

方法区：  
 用于存储被虚拟机加载的类信息（类版本、字段、方法等）、变量，如类变量在类加载是就被加载到方法区中。

方法区和永久代关系就像java接口和实现类的关系，类实现接口，而永久代就是java虚拟机对虚拟机规范的方法区的一种实现。

运行时常量池：

运行时常量池是方法区一部分，用于存放编译期间产生的各种字面量和符号引用。

Jdk1.7之后，字符串常量池被拿到了堆中，其他的运行时常量池还是在方法区里。

对象的创建：

类加载检查-分配内存-初始化零值-设置对象头-执行init方法

类加载检查：

遇到一条new指令时，首先去检查这个指令的参数是否在运行时常量池定位到一个符号引用（类加载的加载阶段是否执行了），然后检查符号引用代表的类是否已经加载、解析、初始化（没有的话就执行相应的类加载过程）。

分配内存：

类加载检查通过后，为新生的对象分配内存。分配方式有指针碰撞和空闲列表。选择哪种分配方式由java堆是否规则决定；而java堆是否规则由GC采用算法是“标记-清除”还是“标记-整理”决定。

内存分配并发问题（线程安全）：

虚拟机正在给A分配内存，同时，同一块内存又被对象B分配情况。虚拟机有两种解决方式:

CAS+失败重试；CAS是乐观锁的一种实现方式，就是每次都假设没有冲突，如果冲突了就重试。

TLAB:每个线程在JAVA堆中预先分配一小块内存，称为本地分配缓存，每个线程对象分配内存时，先从TLAB上分配，如果没有内存了，则采用CAS+失败重试的方式。

初始化零值：

内存分配后，将分配的内存空间初始化零值，这保证了实例字段不赋值也能使用

设置对象头：

初始化零值后，虚拟机对对象进行必要的设置，比如对象是哪个类的实例，GC分代年龄等等，这些信息都存放在对象头里。

执行init方法：

执行< init >方法进行对象的初始化，按照程序员的意愿初始化对象，至此一个真正可用的对象才算产生。

String s1 = new String("abc");这句话创建了几个字符串对象？

1个或2个

如果字符串常量池没有”abc”，那么会在池中创建（此时没有引用），再在堆中创建

如果字符串常量池中有”abc”，那么只在对内存中创建对象。

Java垃圾回收

<https://mp.weixin.qq.com/s/vitwq4oXiqALD0VhA4JCPw>

1. Redis

5.1. 什么是redis，redis的类型数据，应用场景？

Redis是开源的高性能 非关系型 (NoSQL) 的 键值对数据库，速度快，简单稳定。

Redis有五种数据类型String、List、Hash、Set、Zset，常用的是String

String主要作用缓存，分布式锁、session共享

List主要lpush+brpop实现消息队列，信息列表

Hash可以用来存储表结构数据

Set主要是社交类型应用，比如spop从集合抽奖，sadd/srem点赞/取消赞，以及交集共同关注的人。

Zset主要用zrank来做排行榜系统

5.2. 为什么要用 redis/为什么要用缓存:

当然是为了提高性能，在高并发等场景，频繁数据库，效率肯定慢，采用缓存，将请求从缓存中获取数据，能减缓数据库压力，提高性能。

5.3. 为什么要用 redis 而不用 map/guava 做缓存?

后两者属于本地缓存，如果数据较大，会占相当大的内存；此次多实例下，每个实例都保存一份缓存，缓存不具备一致性

Redis是分布式缓存，在多实例下，都共享一份缓存，缓存具备一致性。

5.4. Redis 为什么早期版本选择单线程？

因为 Redis 是基于内存的操作，CPU 不是 Redis 的瓶颈，而且单线程容易实现。

Redis6.0之后支持IO多线程，但是命令执行还是单线程的

5.5. Redis 为什么这么快？

数据放在内存（主要原因）、单线程多路 I/O 复用模型、C语言实现

5.6. redis 和 memcached 的区别：

redis支持丰富的数据类型，后者只支持String

Redis的持久化可以在磁盘中，而不支持持久化，数据全在内存中

Redis使用单线程的多路IO复用模型，后者是多线程的非阻塞IO复用网络模型

5.7. 一个字符串类型的值能存储最大容量是多少？

512M

5.8. Redis的批量处理：

String类型时，批量查询可以使用mget，批量插入使用mset

或者使用pipeline来批量操作

但是pipeline不是原子性的，而mset、mget是原子性的。

5.9. redis 过期键的删除策略？

定期删除：redis每隔100ms就随机抽取一些设置过期时间的key，检查是否过期，过期了就删除。

惰性删除：就是需要手动查一下key，才会被redis删除。

5.10. redis 内存淘汰机制：

大量过期key堆积在内存里，导致redis内存会耗尽，redis 内存淘汰机制会解决，redis 提供 6种数据淘汰策略：

volatile-lru：从设置过期时间的数据集中淘汰最少使用的数据

volatile-ttl：从设置过期时间的数据集中淘汰最近过期的数据

volatile-random：从设置过期时间的数据集中随机淘汰数据

allkeys-lru：内存不足时，移除最近很少使用的key

allkeys-random：内存不足时，随机淘汰数据

no-eviction：内存不足时，不淘汰，报错

5.11. 什么是缓存雪崩？有哪些解决办法？

缓存雪崩：就是同一时间大量的缓存（不同key）失效，后面请求全都落到数据库上，造成短时间数据库压力升高而崩掉。

解决： 尽量保证redis的高可用，避免机器宕机造成的缓存失效

适当使用本地缓存和限流降级

5.12. 什么是缓存穿透？有哪些解决办法？

大量请求不在缓存中的key，使请求没经过缓存直接请求到数据库，造成数据库压力上升而崩掉。

解决： 可以对参数做校验，不合法参数直接处理掉

缓存无效的key，并设置过期时间

还有就是使用布隆过滤器

5.13. 布隆过滤器：

维护一个数组，key在add进数组时，都经过相同的hash方法，hash方法计算出数组位置，并将该位置置为1。

Key经过相同hash算法，计算出数组位置，判断该位置是不是==1，如果不等（false），这个key肯定不在布隆过滤器的数组，那么redis缓存就可以直接返回null，不用查询数据库。

5.14. 如何解决 Redis 的并发竞争 Key 问题：

Redis 的并发竞争 Key 的问题也就是多个系统同时对一个 key 进行操作

解决方案就是使用分布式锁

Redis分布式锁：基于setnx只能set成功同一个key的特性

Zookeeper分布式锁：基于唯一的瞬时有序节点

5.15. 如何保证缓存与数据库双写时的数据一致性?

写完数据库后写缓存；更新数据库后删缓存。

5.16. Redis持久化方式：快照RDB和AOF

RDB触发方式:手动save/bgsave命令，redis.conf配置的自动触发快照、全量复制也自动触发

5.17. RDB优点和缺点：

优点：快照文件远小于内存文件，RDB数据恢复比AOF快

缺点：耗时、耗性能（bgsave每次fork都要创建子线程），不适合实时持久化。

5.18. AOF优缺点：

AOF能够保证实时持久化，但是文件会比较大，恢复时也没RDB快。

5.19. AOF流程：

所有写入命令追加到缓冲区，缓冲区再同步到磁盘，必要时重写AOF文件

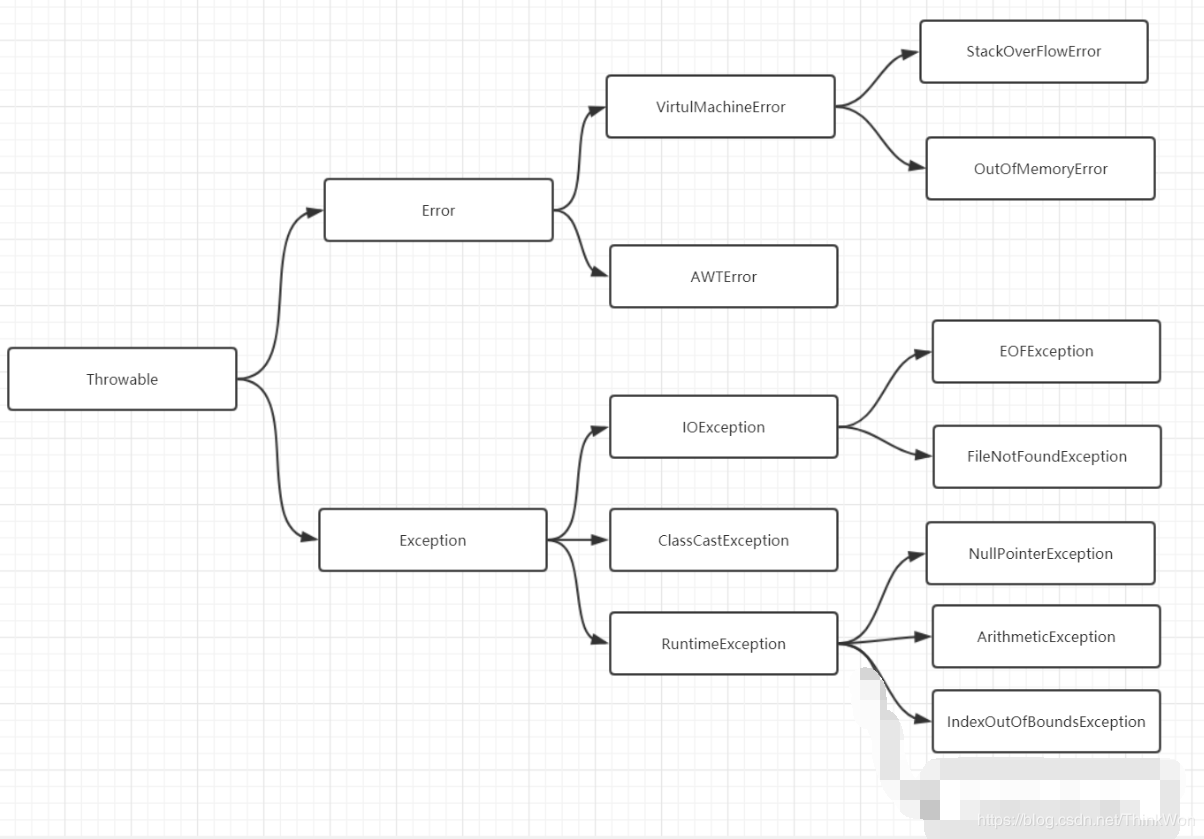
写入缓冲区的原因：

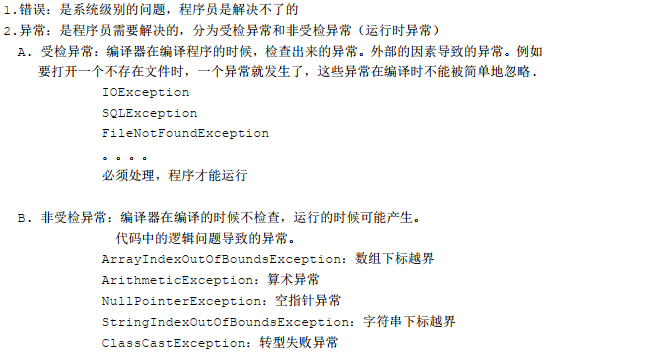
写入缓冲区比直接写入磁盘更能保证效率，缓冲区还有多种同步策略

AOF重写的目的：减少磁盘占用量，加速恢复速度。

异常

异常架构：





异常处理方式：

throws 关键字抛出的异常。

try-catch 语句块中捕获多个异常类型

自定义异常：

如果希望写一个检查性异常类，则需要继承 Exception 类。

如果你想写一个运行时异常类，那么需要继承 RuntimeException 类。

1. 易宝外派华为的外包：
2. 反射机制有什么作用，优缺点

作用：可以在运行的时候动态访问对象的属性、方法、

优点：能够动态访问，代码比较灵活

缺点：性能比直接的java要慢

1. HashMap和HashTable的区别
2. ArrayList和LinkList的区别
3. 类加载顺序
4. 什么是IOC以及应用场景

IOC就是控制反转，就是将对象的控制权交给Spring去管理，这样我们获取对象的时候就不用new，而是直接从Spring容器中获取

应用场景就是依赖注入，直接从Spring容器中获取对象

1. 什么是AOP以及应用场景

AOP就是面向切面编程，由动态代理实现的，可以对原业务进行增强，而不需要改变原有逻辑。

一般应用在事务、日志

1. SpringMVC的流程

前端请求会先发送到前端控制器，前端控制器将请求发送给处理器映射器，

处理器映射器返回处理器执行链（包括处理器对象和处理器拦截器）给前端控制器，

前端控制器根据处理器执行链调用处理器适配器，执行参数封装等操作

然后调用处理器，返回ModelAndView对象给处理器适配器

处理器适配器将ModelAndVliew对方返回给前端控制器

前端控制器将ModelAndView传给视图解析器，解析View给前端控制器

前端控制器将View渲染后返回给前端

1. 多线程的创建方式
2. 垃圾回收机制的算法
3. Java的内存区域
4. SpringBoot的核心注解

@SpringBootApplication

@EnableAutoConfiguration

@Configuration

@SpringBootConfiguration

@ComponentScan

@Bean

1. Zookeeper的读写机制：

读服务（Read Requst），是直接由对应Server的本地副本来进行服务的

写请求会先转发到Leader节点，然后更新持久化到本地，

Leader将更新提议发给Followers

Followers接受请求，持久化到本地，返回一个ACK给Leader

Leader接收到半数以上的ACK时，Leader将广播commit消息。

1. Jdk1.8的新特性有哪些

Lambda表达式、Stream、时间日期相关的类等等

1. SQL方面的优化
2. 索引的类型：

普通索引、主键索引、唯一索引、联合索引

1. 联合索引最左原则：

联合索引(A,B,C)，如果A索引没生效，那么B索引也不生效

联合索引(A,B,C)，相当于创建了（A）、(A,B)和(A,B,C)三个索引

1. 索引失效的场景：

where中包含or时，可能会导致索引失效

where语句中索引使用负向查询，可能会导致索引失效：NOT、!=

在索引列上使用内置函数，一定会导致索引失效

对索引列进行运算，一定会导致索引失效

like通配符可能会导致索引失效

联合索引中，where中索引列违背最左原则，一定会导致索引失效

1. Join连接中on条件里的and和where条件里的and的区别

On条件里的and，是作为限制连接表显示的数据。比如

A 左连接 B on (xx and xx)，则满足on条件的B表数据显示，不满足的为null

Where后面的and是值得条件，只显示满足where 条件的数据，此时左连接或右连接会失效。

1. 中软外派华为的外包：
2. SSM架构的优缺点：

SSM就是Spring、SpringMVC、MyBatis的集成框架

优点自然就是拥有Spring的IOC、AOP特性，实现依赖注入和事务

还有就是拥有SpringMVC的分层开发模式，开发有条理

通过MyBatis持久化，支持对象和字段的关系映射，能够书写比较复杂的SQL

缺点的话就是

1. 建表规范：

数据库和表的字符集统一使用utf-8，兼容性更好

表的字段尽可能都为not null

等等

1. PO和VO:

PO是持久层对象，属性对应数据库的字段

VO是值对象，和PO差不到，只是属性可以不是数据库的字段

1. Controller的主要注解:

@Controller：声明一个类是控制器

@ RequestMapping：指明处理器的访问路径

@RequestParam：接受请求参数

@ResponseBody：将处理器的返回值格式化为json格式返回

@RequestBody：将前端请求参数转换为实体类对象

@Autoview：注入实体类对象

1. Restful接口：

Restful就是一种设计风格，常见的就是将请求参数放入访问路径里面，通过@PathVariable能实现。

/ttt/abc/id

/ttt/{path}/id (@PathVariable(name=”path”) param) -- abc

1. 依赖注入和控制反正有什么关系：

依赖注入是基于控制反正的，容器里面有对象才能注入。

1. Oracle的分页：

Select \* from(select rownum rn, emp.\* f rom emp where rownum <= num2)

where rn >= num1;

1. 多表的关联查询:

等值、内链接、外连接、自查询

1. 组函数有哪些：

max、min、avg、sum、count

1. 组合性的查询：

Group by用到组函数话，select后面不是组函数的列要放在Group by后面

Select a ，count(b) from where C group by a;

1. String、数组、集合获取长度的方法

String是length方法、数组是length属性、集合是size方法

1. String、StringBuilder、StringBuffer的区别：

String、StringBuilder是线程安全的，StringBuffer是线程不安全的

(String是final修饰的，所以线程安全；StringBuilder方法都是同步的)

String是不可变的，每次修改都会生成新的对象

SB是可变的，每次修改都是修改自身，不会新建对象。

1. 常用的设计模式：

常见的就单例模式、代理模式、装饰器模式、工厂模式

单例模式就是保证一个类只有一个实体类，具体实现由懒汉式和饿汉式，实现的要点是类的构造函数私有化，并对外提供一个获取实例对象的方法。

代理模式就是对原有对象进行代理，对原有功能进行增强，比如在方法调用前增加功能。

装饰器模式就是抽象出一个装饰类，并对抽象的装饰类进行实现，可以有多种的组合实现，比如有ABC,三个实现，则可以new A(new B()),也可以new A(new C())

1. 单例模式时，通过new，会怎么样

单例模式的构造函数私有化了，所以不能new来获取对象，如果new的话，肯定会报错。

1. 表单如何防止重复提交

前台页面进行提交后禁掉提交按钮

在session中存放一个特殊标志，放在隐藏域里；通过比较标志，防止重复提交

在数据库里添加约束：在数据库里添加唯一约束，防止出现重复数据

1. 前后端分离跨域问题解决

Nginx服务器反向代理

在服务端设置header

在 Controller的基类中加上 @CrossOrigin