

Java 面试宝典

一、Java 基础.....	11
1. final 关键字的作用?	11
2. abstract class 和interface 有什么区别?.....	11
3. Java 集合类: list、set、queue、map、stack 的特点与用法?	11
4. 说出 ArrayList, Vector, LinkedList 的存储性能和特性?	12
5. 内存泄漏和内存溢出?	12
6. 反射中, Class.forName()和 ClassLoader.loadClass()的区别?	12
7. Int 和Integer 的区别?	12
8. String、StringBuilder、StringBuffer 区别?.....	12
9. Hashtable 和HashMap 的区别?	13
10. 说几个常见的编译时异常?	13
11. 方法重载的规则?	13
12. 方法重写的规则?	13
13. throw 和throws 的区别?.....	13
14. 抽象类和接口的区别?.....	14
15. Java 的基础类型和字节大小?.....	14
16. 四个访问修饰符合访问级别?	14
18. HashSet 的底层实现是什么?	14
19. 抽象类的意义?	14
20. 你为什么重写 equals 时必须重写 hashCode 方法?	15
21. HashSet 和TreeSet 有什么区别?	15
22. 强引用和软引用和弱引用以及虚引用?	15
23. 数组在内存中如何分配?.....	15
24. Java 中怎么创建一个不可变对象?	16
25. Java 中 ++ 操作符是线程安全的吗?	16
26. new 一个对象的过程和 clone 一个对象的过程?	16
27. Java 中==和 equals()的区别?	16
28. final、finalize 和 finally 的不同之处?	16
29. Java 的多态表现在哪里?.....	16
30. 静态类型有什么特点?	16
31. Java 创建对象的几种方式?	17
32. Object 中有哪些公共方法?.....	17
33. &和&&的区别?	17
34. 在.java 源文件中可以有多个类吗(内部类除外)?	17
35. 如何正确的退出多层嵌套循环?	17
36. 内部类有什么作用?	17
37. 深拷贝和浅拷贝的区别是什么?	17
38. String 是基本数据类型吗?	18
39. static 的用法?	18

40. 什么是值传递和引用传递?	18
41. 重载和重写的区别?	18
42. 成员变量和局部变量的区别有哪些?	18
43. 静态方法和实例方法有何不同?	19
44. 什么是多态?	19
45. 多态的优点?	19
46. 多态存在的三个必要条件?	19
47. TreeMap、HashMap、LindedHashMap 的区别?	19
48. Java (OOP) 面向对象的特征有哪些方面?	19
49. 列出一些常见的运行时异常?	10
50. 什么是反射?	10
51. 反射的作用?	10
52. 获取 class 的三种方式?	10
53. break 和continue 的区别?	10
54. 运行时异常与一般异常有何异同?	11
55. List、Map、Set 三个接口存取元素时, 各有什么特点?	11
56. Collection 和Collections 的区别?	11
57. Error 和Exception 有什么区别?	11
58. EJB 的生命周期, 以及如何管理事务?	11
59. Comparable 和Comparator 接口的区别?	11
60. switch 能否作用在 byte、long、string 上?	11
61. jdk 中哪些类是不能继承的?	12
62. Set 里的元素是不能重复的, 那么用什么方法来区分重复与否呢?	12
63. JDK 和JRE 的区别是什么?	12
64. 是否可以在 static 环境中访问非 static 变量?	12
65. Java 支持多继承么?	12
66. 什么是迭代器 (Iterator)?	12
67. Iterator 和ListIterator 的区别是什么?	12
68. Enumeration 接口和 Iterator 接口的区别有哪些?	13
69. List, Set, Map 是否继承自 Collection 接口?	13
70. 字符串常量池到底存在于内存空间的哪里?	13
71. Java 中的编译期常量是什么? 使用它又什么风险?	13
72. 用哪两种方式来实现集合的排序?	13
73. 说出 JDK 1.7 中的三个新特性?	13
74. 说出 5 个 JDK 1.8 引入的新特性?	14
75. ArrayList 源码分析?	14
76. HashMap 源码分析?	14
77. ConcurrentHashMap 源码分析?	14
二、Java IO	15
1. IO 里面的常见类, 字节流、字符流、接口、实现类、方法阻塞?	15
2. 谈谈对 NIO 的认知?	15
3. 字节流和字符流的区别?	15
4. NIO 和传统的 IO 有什么区别?	15
5. BIO 和NIO 和AIO 的区别以及应用场景?	16

6. 什么是 Java 序列化, 如何实现 Java 序列化?	16
7. PrintStream、BufferedWriter、PrintWriter 的比较?	16
8. 什么是节点流, 什么是处理流, 各有什么好处, 处理流的创建有什么特征?	17
9. 什么是 IO 流?	17
10. 有哪些可用的 Filter 流?	17
11. Java 中有几种类型的流?	17
12. 如何实现对象克隆?	17
13. 什么是缓冲区? 有什么作用?	17
14. 什么是阻塞 IO? 什么是非阻塞 IO?	17
三、Java Web.....	18
1. session 和 cookie 的区别?	18
2. session 和 cookie 联系?	18
3. servlet 的生命周期?	19
4. 什么是 webservice?	20
5. jsp 和 servlet 的区别、共同点、各自应用的范围?	20
6. 转发 (forward) 和重定向 (redirect) 的区别?	20
7. request.getAttribute() 和 request.getParameter() 有何区别?	20
8. jsp 静态包含和动态包含的区别?	20
9. MVC 的各个部分都有哪些技术来实现? 如何实现?	20
10. jsp 有哪些内置对象? 作用分别是什么?	20
11. Http 请求的 get 和 post 方法的区别。.....	20
12. tomcat 容器是如何创建 servlet 类实例? 用到了什么原理?	21
13. JDBC 访问数据库的基本步骤是什么?	21
14. 为什么要使用 PreparedStatement?	21
15. 数据库连接池的原理。为什么要使用连接池?	21
16. execute, executeQuery, executeUpdate 的区别是什么?	22
17. JDBC 的 ResultSet 是什么?	22
18. 什么是 Servlet?	22
19. doGet 和 doPost 方法有什么区别?	22
20. JSP 有哪些动作? 分别是什么?	23
21. JSP 常用的指令?	23
22. 页面间对象传递的方法?	23
23. JSP 中动态 INCLUDE 与静态 INCLUDE 的区别?	23
24. JSP 的四大范围?	23
25. BS 与 CS 的联系与区别?	24
26. 说出 Servlet 的生命周期, 并说出 Servlet 和 CGI 的区别?.....	24
27. 如何防止表单重复提交?	25
28. request 作用?	25
29. get 请求中文乱码?	25
30. post 请求中文乱码问题?	25
31. 响应乱码?	25
32. Cookie 对象的缺陷?	26
33. Session 的运行机制?	26
34. 钝化和活化?	26

35. Filter 的工作原理?	26
36. Filter 链是什么?	27
37. 监听器类型?	27
38. Servlet Filter Listener 启动顺序?	27
四、JVM	27
1. Java 的内存划分?	27
2. 什么是 Java 虚拟机? 为什么 Java 被称作是无关平台的编程语言?	28
3. 如何判断一个对象应该被回收?	28
4. GC 触发的条件?	28
5. 可以作为 GCRoots 的对象有哪些?	28
6. JVM 中一次完整的 GC 流程是怎样的, 对象如何晋升到老年代?	29
7. 双亲委派模型?	29
10. 导致 Full GC 一般有哪些情况?	29
11. Minor GC, Full GC 触发条件?	30
12. JVM 性能调优?	30
13. Java 内存模型?	30
14. Java 中堆和栈有什么区别?	31
15. 常见的垃圾回收算法有哪些? 简述其原理?	32
16. 解释栈(stack)、堆(heap)和方法区(method area)的用法?	32
17. 什么是类的加载?	32
18. 类加载器有哪些?	33
19. Java 对象创建过程?	33
20. Java 中类的生命周期使什么?	33
21. 都有哪些垃圾回收器?	33
22. JVM 调优命令?	34
23. JVM 调优工具?	34
24. 描述一下 JVM 加载 class 文件的原理机制?	34
25. GC 是什么? 为什么要有 GC?	34
26. 垃圾回收器的基本原理是什么?	34
27. Java 中的引用类型有几种?	35
五、多线程	35
1. Java 创建线程之后, 直接调用 start() 方法和 run() 的区别?	35
2. 线程 B 怎么知道线程 A 修改了变量?	35
3. synchronized 和 Volatile、CAS 比较?	35
4. 线程间通信, wait 和 notify 的理解和使用?	36
5. 定时线程的使用?	36
6. 线程同步的方法?	36
7. 进程和线程的区别?	36
8. 什么叫线程安全?	36
9. 线程的几种状态?	37
10. volatile 变量和 atomic 变量有什么不同?	37
11. Java 中什么是静态条件?	37
12. Java 中如何停止一个线程?	37
13. 线程池的优点?	37

14. volatile 的理解?	38
15. 实现多线程有几种方式?	38
16. Java 中notify 和notifyAll 有什么区别?	38
17. 什么是乐观锁和悲观锁?	38
18. 线程的创建方式?	38
19. 线程池的作用?	39
20. wait 和sleep 的区别?	39
21. 产生死锁的条件?	39
22. 请写出实现线程安全的几种方式?	39
23. 守护线程是什么? 它和非守护线程的区别?	39
24. 什么是多线程的上下文切换?	39
25. Callable 和Runnable 的区别是什么?	39
26. 线程阻塞有哪些原因?	40
27. synchronized 和Lock 的区别?	40
28. ThreadLocal 是什么? 有什么作用?	40
29. 交互方式分为同步和异步两种?	40
30. 什么是线程?	40
31. 什么是 FutureTask?	40
32. Java 中interrupted 和isInterruptedd 方法的区别?	40
33. 死锁的原因?	40
34. 什么是自旋?	40
35. 怎么唤醒一个阻塞的线程?	40
36. 如果提交任务时, 线程池队列已满, 这时会发生什么?	40
37. 什么是线程局部变量?	41
38. 使用 volatile 关键字的场景?	41
39. 线程池的工作原理, 几个重要参数?	41
40. 线程池的类型?	41
41. 线程池的阻塞队列有哪些?	42
42. 线程池的拒绝策略都有哪些?	42
六、设计模式	42
1. 说一下你熟悉的设计模式?	42
2. 简单工厂和抽象工厂的区别?	42
3. 设计模式的优点?	43
4. 设计模式的六大基本原则?	43
5. 单例模式?	43
6. 设计模式的分类?	44
7. 23 种设计模式的具体的每种模式的功能?	45
7. UML 是什么?	47
8. 桥接模式是什么?	47
七、开源框架	49
1. hibernate 和mybatis 的区别?	49
2. MyBatis 的优点?	50
3. MyBatis 框架的缺点?	50
4. SpringMVC 工作流程?	50

5. MyBatis 框架使用的场合?.....	50
6. Spring 中beanFactory 和ApplicationContext 的联系和区别?.....	50
7. SpringIOC 注入的几种方式?.....	50
8. 拦截器与过滤器的区别?.....	50
9. SpringIOC 是什么?.....	50
10. AOP 有哪些实现方式?.....	50
11. 解释一下代理模式?.....	51
12. Mybatis 是如何 sql 执行结果封装为目标对象? 都有哪些映射形式?.....	51
13. Spring bean 的生命周期?.....	51
14. Spring 框架中都用到了哪些设计模式?.....	52
15. Spring 中的事件处理?.....	52
16. 使用 Spring 框架的好处是什么?.....	52
17. 解释 Spring 支持的几种 bean 的作用域?.....	52
18. 在 Spring 中如何注入一个 java 集合?.....	52
19. 什么是 Spring bean?.....	53
20. 什么是 spring 自动装配?.....	53
21. 自动装配有哪些方式?.....	53
22. 自动装配有什么局限?.....	53
23. Spring 的重要注解?.....	53
24. @Component, @Controller, @Repository, @Service 有何区别?.....	54
25. 列举 spring 支持的事务管理类型?.....	54
26. Spring 框架的事物管理有哪些优点?.....	54
27. Spring AOP (面向切面) 编程的原理?.....	54
28. Spring MVC 框架有什么用?.....	55
29. 介绍一下 WebApplicationContext?.....	55
30. SpringMVC 和struts2 的区别有哪些?.....	55
31. Mybatis 中#{ }和\${ }的区别是什么?.....	56
32. Spring 中@Autowired 与@Resource 的区别?.....	56
33. 什么是控制反转 (IOC), 什么是依赖注入 (DI)?.....	56
34. Spring 运行原理?.....	56
35. Spring 的事务传播行为?.....	56
36. SpringMVC 的运行流程?.....	57
八、分布式相关.....	58
1. Redis 和Memcache 的区别?.....	58
2. 使用 Redis 有哪些好处?.....	58
3. 什么是 redis 持久化? rdb 和aof 的比较?.....	58
4. Redis 最适合的场景?.....	58
5. redis 哈希槽的概念?.....	59
6. 怎么理解 Redis 事务?.....	59
7. redis 的淘汰策略有哪些?.....	59
8. redis 有哪些数据结构?.....	59
9. redis 缓存穿透、缓存雪崩、缓存击穿?.....	59
10. redis 如何实现高并发?.....	60
11. redis 如何实现高可用?.....	60

12. redis 单线程还能处理速度那么快?	60
13. 为什么 Redis 的操作是原子性的, 怎么保证原子性?	60
14. redis 的主从复制的实现过程?.....	61
15. redis 的哨兵机制的作用?.....	61
16. redis 常见的性能问题和解决方案?.....	61
17. 分布式缓存?.....	61
18. 什么是 Nginx?	61
19. nginx 相对于 apache 的优点?.....	61
20. Nginx 优化的方式?	62
21. Nginx 如何处理一个请求的?.....	62
22. Nginx 是如何实现高并发的?.....	62
23. Nginx 的进程模型?.....	63
24. Nginx 负载均衡的 4 种分配方式?.....	63
25. 为什么要用 Nginx?	63
26. 什么是正向代理?.....	63
27. 什么是反向代理?.....	64
28. 什么是负载均衡?.....	64
29. Nginx 的调度算法有哪些?	64
30. Nginx 负载均衡调度状态?.....	64
31. 可以从哪些方面来优化 nginx 服务?	65
32. 为什么要用 MQ?	65
33. 使用 MQ 会有什么问题?	65
34. 怎么保证 MQ 的高可用?	65
35. MQ 的优缺点?	66
36. Kafka、ActiveMQ、RabbitMQ、RocketMQ 都有什么区别?	66
37. 如何设置消息的过期时间?	66
38. 消息的持久化是如何实现的?	67
39. Zookeeper 是什么?	67
40. Zookeeper 的应用场景?	67
41. 四种类型的数据节点 Znode?	67
42. Zookeeper Watcher 机制?.....	67
43. Zookeeper 下 Server 工作状态?.....	68
44. Zookeeper 是如何保证事务的顺序一致性的?	68
45. ZK 节点宕机如何处理?	68
46. Zookeeper 有哪几种部署模式?	69
47. Dubbo 内置了哪几种容器?	69
48. Dubbo 里面有哪几种角色?	69
49. Dubbo 有哪几种集群容错方案, 默认是那种?	69
50. Dubbo 有哪几种负载均衡策略, 默认是哪种?	69
51. Dubbo 的管理控制台能做什么?	69
52. 什么是 Spring Boot?	69
53. Spring Boot 有哪些优点?	70
54. 什么是 JavaConfig?	70
55. 如何重新加载 Spring Boot 上的更改, 而无需重新启动服务器?	70

56. Spring Boot 中的监视器是什么?	72
57. 如何在 Spring Boot 中禁用 Actuator 端点安全性?	72
59. 什么是 WebSocket?	72
60. 什么是 Swagger? 你用 Spring Boot 实现了它吗?	72
61. 什么是 Apache Kafka?	72
62. 什么是 Spring Cloud?	72
63. 使用 Spring Cloud 有什么优势?	73
64. 服务注册和发现是什么意思? Spring Cloud 如何实现?	73
65. 什么是 Netflix Feign? 它的优点是什么?	73
66. 什么是 Spring Cloud Bus? 我们需要它吗?	74
67. 什么是 Hystrix 断路器? 我们需要它吗?	75
九、网络通信.....	76
1. http 协议的状态码有哪些? 含义是什么?	76
2. Http 的请求报文组成?	77
3. 一次完整的 Http 请求是怎样的?	77
4. Tcp 和Udp 的区别?	77
5. SSL 协议的三个特性	77
6. TCP 的三次握手与四次挥手.....	77
7. 为什么连接的时候是三次握手, 关闭的时候却是四次握手?	78
8. 如果已经建立了连接, 但是客户端突然出现故障了怎么办?	78
9. http 中重定向和请求转发的区别?	79
10. GET 和POST 的区别?	79
6. Http 与 Https 的区别.....	79
12. 什么是 Http 协议无状态协议?怎么解决 Http 协议无状态协议?.....	79
13. HTTPS 工作原理	81
十、数据库.....	81
1. MySql 的存储引擎有哪些, 区别是什么	81
2. 触发器的作用?	81
3. 什么是存储过程? 用什么来调用?	81
4. 存储过程的优缺点。	80
5. SQL 优化的具体操作	80
6. 什么叫视图? 游标是什么?	80
7. 视图的优缺点.....	80
8. 事务的四个特性.....	80
9. 数据库乐观锁, 悲观锁的区别, 怎么实现.....	81
10. 事务的并发问题.....	81
12. 非关系型数据库和关系型数据库区别, 优势比较.....	83
13. 数据库的五大范式.....	83
14. 什么是内连接、外连接、交叉连结、笛卡尔积等?	84
15. SQL 语言分类	84
16. count(*)、count(1)、count(column)的区别.....	84
17. 什么是索引?	84
18. 索引的作用?	85
19. 索引的优缺点.....	85

20. 什么样的字段适合建索引?	85
21. Hash 索引和 B+树索引的区别?	85
22. MySQL 三种锁的级别	85
23. 为什么不都用 Hash 索引而使用 B+树索引?	85
24. B 树和 B+树的区别.....	86
25. 为什么说 B+比 B 树更适合实际应用中操作系统的文件索引和数据库索引?	86
26. 聚集索引和非聚集索引区别?.....	86
十一、算法.....	86
1. 冒泡排序.....	86
2. 两个有序数组的合并排序.....	88
3. 一个数组的倒序.....	88
4. 计算一个正整数的正平方根.....	88
5. 快速排序算法?	89
6. 二叉树的遍历算法.....	90
6. DFS, BFS 算法	91
7. 时间类型转换。.....	90
8. 逆波兰计算器.....	91
9. 阶乘.....	92
10. 兔子问题 斐波那契数列.....	93
11. 判断 101-200 之间有多少个素数, 并输出所有素数。	93
12 水仙花数.....	94
13. 正整数分解质因数.....	94
14. 最大公约数和最小公倍数.....	95
15. 完数.....	96
16. 完全平方数.....	96
17. 猴子吃桃问题.....	97
18. 两个乒乓球队进行比赛.....	97
19 求 $1+2!+3!+\dots+20!$ 的和.....	98
20. 有 5 个人坐在一起.....	98
22. 100 之内的素数.....	99
23. 矩阵对角线元素之和.....	100
24. 杨辉三角形.....	101
25. 有 n 个人围成一圈.....	102
26. 海滩上有一堆桃子.....	103
27. 冒泡排序.....	106
28. 快速排序.....	107
29. 选择排序.....	107
30. 插入排序.....	108
十二、并发与性能调优.....	109
1. 每秒钟 5k 个请求, 查询手机号的笔试题, 设计算法?请求再多, 比如 5w, 如何设计整个系统?109	
2. 高并发情况下, 我们系统是如何支撑大量的请求的?	110
3. 集群如何同步会话状态?	110
4. 负载均衡的原理?	110
5. 怎么提高并发量, 请列举你所知道的方案?	111

6. 系统的用户量有多少？多用户并发访问时如何解决？	111
7. 如果有一个特别大的访问量，到数据库上，怎么做优化？	111
8. 大面积并发，在不增加服务器，如何解决服务器响应不及时问题？	112

一、Java 基础

1. final 关键字的作用？

被 final 修饰的类不可以被继承，被 final 修饰的方法不可以被重写，被 final 修饰的变量不可以被改变。如果修饰引用，那么表示引用不可变，引用指向的内容可变。被 final 修饰的方法，JVM 会尝试将其内联，以提高运行效率，被 final 修饰的常量，在编译阶段会存入常量池中。

2. abstract class 和 interface 有什么区别？

声明方法的存在而不去实现它的类被叫做抽象类（abstract class），它用于要创建一个体现某些基本行为的类，并为该类声明方法，但不能在该类中实现该方法的情况。不能创建 abstract 类的实例。然而可以创建一个变量，其类型是一个抽象类，并让它指向具体子类的一个实例。不能有抽象构造函数或抽象静态方法。Abstract 类的子类为它们父类中的所有抽象方法提供实现，否则它们也是抽象类。取而代之，在子类中实现该方法。知道其行为的其它类可以在类中实现这些方法。

接口（interface）是抽象类的变体。在接口中，所有方法都是抽象的。多继承性可通过实现这样的接口而获得。接口中的所有方法都是抽象的，没有一个有程序体。接口只可以定义 static final 成员变量。接口的实现与子类相似，除了该实现类不能从接口定义中继承行为。当类实现特殊接口时，它定义（即将程序体给予）所有这种接口的方法。然后，它可以在实现了该接口的类的任何对象上调用接口的方法。由于有抽象类，它允许使用接口名作为引用变量的类型。通常的动态联编将生效。引用可以转换到接口类型或从接口类型转换， instanceof 运算符可以用来决定某对象的类是否实现了接口。

3. Java 集合类：list、set、queue、map、stack 的特点与用法？

Map

Map 是键值对，键 Key 是唯一不能重复的，一个键对应一个值，值可以重复。

TreeMap 可以保证顺序，HashMap 不保证顺序，即为无序的，Map 中可以将 Key 和 Value 单独抽取出来，其中 KeySet() 方法可以将所有的 keys 抽取成一个 Set，而 Values() 方法可以将 map 中所有的 values 抽取成一个集合。

Set

不包含重复元素的集合，set 中最多包含一个 null 元素，只能用 Iterator 实现单项遍历，Set 中没有同步方法。

List

有序的可重复集合，可以在任意位置增加删除元素，用 Iterator 实现单向遍历，也可用 ListIterator 实现双向遍历。

Queue

Queue 遵从先进先出原则，使用时尽量避免 add() 和 remove() 方法，而是使用 offer() 来添加元素，使用 poll() 来移除元素，它的优点是可以通过返回值来判断是否成功，LinkedList 实现了 Queue 接口，Queue 通常不允许插入 null 元素。

Stack

Stack 遵从后进先出原则，Stack 继承自 Vector，它通过五个操作对类 Vector 进行扩展，

允许将向量视为堆栈，它提供了通常的 push 和 pop 操作，以及取堆栈顶点的 peek() 方法、测试堆栈是否为空的 empty 方法等。

用法

如果涉及堆栈，队列等操作，建议使用 List。

对于快速插入和删除元素的，建议使用 LinkedList。

如果需要快速随机访问元素的，建议使用 ArrayList。

4. 说出 ArrayList, Vector, LinkedList 的存储性能和特性?

ArrayList 和 Vector 都是使用数组方式存储数据，此数组元素数大于实际存储的数据以便增加和插入元素，它们都允许直接按序号索引元素，但是插入元素要涉及数组元素移动等内存操作，所以索引数据快而插入数据慢，Vector 由于使用了 synchronized 方法（线程安全），通常性能上较 ArrayList 差，而 LinkedList 使用双向链表实现存储，按序号索引数据需要进行前向或后向遍历，但是插入数据时只需要记录本项的前后项即可，所以插入速度较快。

5. 内存泄漏和内存溢出?

内存泄漏 (memoryleak)，是指应用程序在申请内存后，无法释放已经申请的内存空间，一次内存泄漏危害可以忽略，但如果任其发展最终会导致内存溢出 (outofmemory)。如读取文件后流要进行及时的关闭以及对数据库连接的释放。

内存溢出 (outofmemory) 是指应用程序在申请内存时，没有足够的内存空间供其使用。如我们在项目中对于大批量数据的导入，采用分批量提交的方式。

6. 反射中，Class.forName() 和 ClassLoader.loadClass() 的区别?

Class.forName(className) 方法，内部实际调用的方法是 Class.forName(className,true,classloader);

第 2 个 boolean 参数表示类是否需要初始化，Class.forName(className) 默认是需要初始化，一旦初始化，就会触发目标对象的 static 块代码执行，static 参数也也会被再次初始化，ClassLoader.loadClass(className) 方法，内部实际调用的方法是 ClassLoader.loadClass(className,false); 第 2 个 boolean 参数，表示目标对象是否进行链接，false 表示不进行链接，由上面介绍可以，不进行链接意味着不进行包括初始化等一些列步骤，那么静态块和静态对象就不会得到执行

7. Int 和 Integer 的区别?

Integer 是 int 的包装类型，在拆箱和装箱中，二者自动转换。int 是基本类型，直接存数值；而 integer 是对象；用一个引用指向这个对象。由于 Integer 是一个对象，在 JVM 中对象需要一定的数据结构进行描述，相比 int 而言，其占用的内存更大一些。

8. String、StringBuilder、StringBuffer 区别?

String	字符串常量	不可变	使用字符串拼接时是不同的 2 个空间
StringBuffer	字符串变量	可变	线程安全 字符串拼接直接在字符串后追加
StringBuilder	字符串变量	可变	非线程安全 字符串拼接直接在字符串后追加

1. **StringBuilder** 执行效率高于 **StringBuffer** 高于 **String**.
2. **String** 是一个常量，是不可变的，所以对于每一次+=赋值都会创建一个新的对象，**StringBuffer** 和 **StringBuilder** 都是可变的，当进行字符串拼接时采用 **append** 方法，在原来的基础上进行追加，所以性能比 **String** 要高，又因为 **StringBuffer** 是线程安全的而 **StringBuilder** 是线程非安全的，所以 **StringBuilder** 的效率高于 **StringBuffer**.
3. 对于大数据量的字符串的拼接，采用 **StringBuffer, StringBuilder**.

9. Hashtable 和 HashMap 的区别?

- 1、**HashTable** 线程安全，**HashMap** 非线程安全
- 2、**Hashtable** 不允许 null 值(key 和 value 都不可以)，**HashMap** 允许 null 值(key 和 value 都可以)。
- 3、两者的遍历方式大同小异，**Hashtable** 仅仅比 **HashMap** 多一个 **elements** 方法。

10. 说几个常见的编译时异常?

SQLException 提供有关数据库访问错误或其他错误的信息的异常。

IOException 表示发生了某种 I / O 异常的信号。此类是由失败或中断的 I / O 操作产生的一般异常类

FileNotFoundException 当试图打开指定路径名表示的文件失败时，抛出此异常。

ClassNotFoundException 找不到具有指定名称的类的定义。

EOFException 当输入过程中意外到达文件或流的末尾时，抛出此异常。

11. 方法重载的规则?

方法名一致，参数列表中参数的顺序，类型，个数不同。

重载与方法的返回值无关，存在于父类和子类，同类中。

可以抛出不同的异常，可以有不同修饰符。

12. 方法重写的规则?

参数列表、方法名、返回值类型必须完全一致，构造方法不能被重写；声明为 **final** 的方法不能被重写；声明为 **static** 的方法不存在重写(重写和多态联合才有意义)；访问权限不能比父类更低；重写之后的方法不能抛出更宽泛的异常

13. throw 和 throws 的区别?

throw:

throw 语句用在方法体内，表示抛出异常，由方法体内的语句处理。**throw** 是具体向外抛出异常的动作，所以它抛出的是一个异常实例，执行 **throw** 一定是抛出了某种异常。**throws:**

throws 语句是用于方法声明后面，表示如果抛出异常，由该方法的调用者来进行异常的处理。**throws** 主要是声明这个方法会抛出某种类型的异常，让它的使用者要知道需要捕获的

异常的类型。throws 表示出现异常的一种可能性，并不一定会发生这种异常。

14. 抽象类和接口的区别？

- 1、接口中所有的方法隐含的都是抽象的。而抽象类则可以同时包含抽象和非抽象的方法。
- 2、类可以实现很多个接口，但是只能继承一个抽象类
- 3、类如果实现一个接口，它必须实现接口声明的所有方法。但是，类可以不实现 抽象类声明的所有方法，当然，在这种情况下，类也必须得声明成是抽象的。
- 4、抽象类可以在不提供接口方法实现的情况下实现接口。
- 5、Java 接口中声明的变量默认都是 final 的。抽象类可以包含非 final 的变量。
- 6、Java 接口中的成员函数默认是 public 的。抽象类的成员函数可以是 private, protecte 或者是 public 。
- 7、接口是绝对抽象的，不可以被实例化(java 8 已支持在接口中实现默认的方法)。抽象类也不可以被实例化，但是，如果它包含 main 方法的话是可以被调用的。

15. Java 的基础类型和字节大小？

布尔型 boolean 8 位；字节型 byte 8 位；字符型 char 16 位；
短整型 short 16 位；整型 int 32 位；长整型 long 64 位；
浮点型 float 32 位；双精度 double 64 位；

16. 四个访问修饰符合访问级别？

Protected、public、没有访问修饰符、private

17. String 和 StringBuffer 的区别？

String 和 StringBuffer 主要区别是性能:String 是不可变对象,每次对 String 类型进行操作都等同于产生了一个新的 String 对象,然后指向新的 String 对象. 所以尽量不要对 String 进行大量的拼接操作,否则会产生很多临时对象,导致 GC 开始工作,影响系统性能. StringBuffer 是对对象本身操作,而不是产生新的对象,因此在有大量拼接的情况下,我们建议使用 StringBuffer(线程安全).

18. HashSet 的底层实现是什么？

HashSet 的实现是依赖于 HashMap 的, HashSet 的值都是存储在 HashMap 中的。在 HashSet 的构造法中会初始化一个 HashMap 对象, HashSet 不允许值重复。因此, HashSet 的值是作为 HashMap 的 key 存储在 HashMap 中的, 当存储的值已经存在时返回 false。

19. 抽象类的意义？

抽象类的意义可以用三句话来概括：

- 1、为其他子类提供一个公共的类型
- 2、封装子类中重复定义的内容

3、定义抽象方法,子类虽然有不同的实现,但是定义时一致的

20. 你为什么重写 equals 时必须重写 hashCode 方法?

hashCode() 的作用是获取哈希码,也称为散列码;它实际上是返回一个 int 整数。这个哈希码的作用是确定该对象在哈希表中的索引位置。如果两个对象相等,则 hashCode 一定也是相同的。如果两个对象相等,对两个对象分别调用 equals 方法都返回 true。如果两个对象有相同的 hashCode 值,它们也不一定是相等的。因此,equals 方法被覆盖过,则 hashCode 方法也必须被覆盖。

hashCode() 的默认行为是对堆上的对象产生独特值。如果没有重写 hashCode(), 则该 class 的两个对象无论如何都不会相等(即使这两个对象指向相同的数据)。

21. HashSet 和 TreeSet 有什么区别?

HashSet 是由一个 hash 表来实现的,因此,它的元素是无序的。add(), remove(), contains() 方法的时间复杂度是 $O(1)$ 。TreeSet 是由一个树形的结构来实现的,它里面的元素是有序的。因此,add(), remove(), contains() 方法的时间复杂度是 $O(\log n)$ 。

22. 强引用和软引用和弱引用以及虚引用?

1、强引用

最普遍的一种引用方式,如 `String s = "abc"`,变量 s 就是字符串“abc”的强引用,只要强引用存在,则垃圾回收器就不会回收这个对象。

2、软引用(SoftReference)

用于描述还有用但非必须的对象,如果内存足够,不回收,如果内存不足,则回收。一般用于实现内存敏感的高速缓存,软引用可以和引用队列 ReferenceQueue 联合使用,如果软引用的对象被垃圾回收,JVM 就会把这个软引用加入到与之关联的引用队列中。

3、弱引用(WeakReference)

弱引用和软引用大致相同,弱引用与软引用的区别在于:只具有弱引用的对象拥有更短暂的生命周期。在垃圾回收器线程扫描它所管辖的内存区域的过程中,一旦发现了只具有弱引用的对象,不管当前内存空间是否足够与否,都会回收它的内存。

4、虚引用(PhantomReference)

就是形同虚设,与其他几种引用都不同,虚引用并不会决定对象的生命周期。如果一个对象仅持有虚引用,那么它就和没有任何引用一样,在任何时候都可能被垃圾回收器回收。虚引用主要用来跟踪对象被垃圾回收器回收的活动。

虚引用与软引用和弱引用的一个区别在于:

虚引用必须和引用队列 (ReferenceQueue) 联合使用。当垃圾回收器准备回收一个对象时,如果发现它还有虚引,就会在回收对象的内存之前,把这个虚引用加入到与之关联的引用队列中。

23. 数组在内存中如何分配?

当一个对象使用 new 关键字创建的时候,会在堆上分配内存空间,然后才返回到对象的引用。这对数组来说也是一样的,因为数组也是一个对象,简单的值类型的数组,每个数组成员是一个引用(指针)引用到栈上的空间。

24. Java 中怎么创建一个不可变对象？

1. 对象的状态在构造函数之后都不能被修改,任何修改应该通过创建一个新对象来实现.
2. 所有的对象属性应该都设置为 final
3. 对象创建要正确,例如:对象的应用不能在构造函数中被泄露出去
4. 对象要设置为 final,确保不要继承的 Class 修改了 immutability 特性

25. Java 中 ++ 操作符是线程安全的吗？

不是线程安全的操作。它涉及到多个指令,如读取变量值,增加,然后存储回内存,这个过程可能会出现多个线程交差。

26. new 一个对象的过程和 clone 一个对象的过程？

new 操作符的本意是分配内存。程序执行到 new 操作符时,首先去看 new 操作符后面的类型,因为知道了类型,才能知道要分配多大的内存空间。分配完内存之后,再调用构造函数,填充对象的各个域,这一步叫做对象的初始化,构造方法返回后,一个对象创建完毕,可以把他的引用(地址)发布到外部,在外部就可以使用这个引用操纵这个对象。

clone 在第一步是和 new 相似的,都是分配内存,调用 clone 方法时,分配的内存和原对象(即调用 clone 方法的对象)相同,然后再使用原对象中对应的各个域,填充新对象的域,填充完成之后,clone 方法返回,一个新的相同的对象被创建,同样可以把这个新对象的引用发布到外部。

27. Java 中==和 equals() 的区别？

使用==比较原生类型如: boolean、int、char 等等,使用 equals() 比较对象。

- 1、==是判断两个变量或实例是不是指向同一个内存空间。equals 是判断两个变量或实例所指向的内存空间的值是不是相同。
- 2、==是指对内存地址进行比较。equals() 是对字符串的内容进行比较。
- 3、==指引用是否相同,equals() 指的是值是否相同。

28. final、finalize 和 finally 的不同之处？

final 用于声明属性,方法和类,分别表示属性不可变,方法不可覆盖,类不可继承。

finally 是异常处理语句结构的一部分,表示总是执行。

finalize 是 Object 类的一个方法,在垃圾收集器执行的时候会调用被回收对象的此方法,可以覆盖此方法提供垃圾收集时的其他资源回收,例如关闭文件等。

29. Java 的多态表现在哪里？

多态要有动态绑定,否则就不是多态,方法重载也不是多态(因为方法重载是编译期决定好的,没有后期也就是运行期的动态绑定)当满足这三个条件:1、有继承 2、有重写 3、要有父类引用指向子类对象

30. 静态类型有什么特点？

- 1、静态的属性：随着类的加载而加载，该属性不在属于某个对象，属于整个类
- 2、静态的方法：直接用类名调用，静态方法里不能访问非静态成员变量
- 3、静态类：不能直接创建对象，不可被继承

31. Java 创建对象的几种方式？

new 创建新对象;通过反射机制;采用 clone 机制;通过序列化机制

32. Object 中有哪些公共方法？

Object 是所有类的父类，任何类都默认继承 Object clone 保护方法，实现对象的浅复制，只有实现了 Cloneable 接口才可以调用该方法，否则抛出 CloneNotSupportedException 异常。equals 在 Object 中与一是一样的，子类一般需要重写该方法。hashCode 该方法用于哈希查找，重写了 equals 方法一般都要重写 hashCode 方法。这个方法在一些具有哈希功能的 Collection 中用到。getClass final 方法，获得运行时类型 wait 使当前线程等待该对象的锁，当前线程必须是该对象的拥有者，也就是具有该对象的锁。wait() 方法一直等待，直到获得锁或者被中断。wait(long timeout) 设定一个超时间隔，如果在规定时间内没有获得锁就返回。

33. &和&&的区别？

&是位运算符，表示按位与运算，&&是逻辑运算符，表示逻辑与（and）。

34. 在. java 源文件中可以有多个类吗（内部类除外）？

一个. java 源文件中可以包括多个类（不是内部类），但是单个文件中只能有一个 public 类，并且该 public 类必须与文件名相同

35. 如何正确的退出多层嵌套循环？

- 1、使用标号和 break;
- 2、通过在外层循环中添加标识符

36. 内部类有什么作用？

- 1、内部类可以很好的实现隐藏，一般的非内部类，是不允许有 private 与 protected 权限的，但内部类可以
- 2、内部类拥有外围类的所有元素的访问权限
- 3、可是实现多重继承
- 4、可以避免修改接口而实现同一个类中两种同名方法的调用

37. 深拷贝和浅拷贝的区别是什么？

浅拷贝:被复制对象的所有变量都含有与原来的对象相同的值,而所有的对其他对象的引用 仍然指向原来的对象.换言之,浅拷贝仅仅复制所考虑的对象,而不复制它所引用的对象.

深拷贝:被复制对象的所有变量都含有与原来的对象相同的值.而那些引用其他对象的变量 将指向被复制过的新对象.而不再是原有的那些被引用的对象.换言之,深拷贝把要复制的对象所引用的对象都复制了一遍.

38. String 是基本数据类型吗?

基本数据类型包括 byte、int、char、long、float、double、boolean 和 short。

java.lang.String 类是 final 类型的,因此不可以继承这个类、不能修改这个类。为了提高效率节省空间,我们应该用 StringBuffer 类

39. static 的用法?

Static 可以修饰内部类、方法、变量、代码块;Static 修饰的类是静态内部类;Static 修饰的方法是静态方法,表示该方法属于当前类的,而不属于某个对象的,静态方法也不能被重写,可以直接使用类名来调用。在 static 方法中不能使用 this 或者 super 关键字。Static 修饰变量是静态变量或者叫类变量,静态变量被所有实例所共享,不会依赖于对象。静态变量在内存中只有一份拷贝,在 JVM 加载类的时候,只为静态分配一次内存。

Static 修饰的代码块叫静态代码块,通常用来做程序优化的。静态代码块中的代码在整个类加载的时候只会执行一次。静态代码块可以有多个,如果有多个,按照先后顺序依次执行。

40. 什么是值传递和引用传递?

对象被值传递,意味着传递了对象的一个副本。因此,就算是改变了对象副本,也不会影响 源对象的值,对象被引用传递,意味着传递的并不是实际的对象,而是对象的引用。因此, 外部对引用对象所做的改变会反映到所有的对象上。

41. 重载和重写的区别?

方法的重写 Overriding 和重载 Overloading 是 Java 多态性的不同表现。重写 Overriding 是父类与子类之间多态性的一种表现,重载 Overloading 是一个类中多态性的一种表现。如果在子类中定义某方法与其父类有相同的名称和参数,我们说该方法被重写 (Overriding)。子类的对象使用这个方法时,将调用子类中的定义,对它而言,父类中的定义如同被“屏蔽”了。如果在一个类中定义了多个同名的方法,它们或有不同的参数个数或有不同的参数类型, 则称为方法的重载 (Overloading)。

42. 成员变量和局部变量的区别有哪些?

- 1、从语法形式上,看成员变量是属于类的,而局部变量是在方法中定义的变量或是方法的参数;成员变量可以被 public,private,static 等修饰符所修饰,而局部变量不能被访问控制修饰符及 static 所修饰;成员变量和局部变量都能被 final 所修饰;
- 2、从变量在内存中的存储方式来看,成员变量是对象的一部分,而对象存在于堆内存,局部变量存在于栈内存
- 3、从变量在内存中的生存时间上看,成员变量是对象的一部分,它随着对象的创建而存在, 而局部变量随着方法的调用而自动消失。

4、成员变量如果没有被赋初值，则会自动以类型的默认值而赋值（一种情况例外被 final 修饰但没有被 static 修饰的成员变量必须显示地赋值）；而局部变量则不会自动赋值。

43. 静态方法和实例方法有何不同？

静态方法和实例方法的区别主要体现在两个方面：

在外部调用静态方法时，可以使用“类名.方法名”的方式，也可以使用“对象名.方法名”的方式。而实例方法只有后面这种方式。也就是说，调用静态方法可以无需创建对象。

静态方法在访问本类的成员时，只允许访问静态成员（即静态成员变量和静态方法），而不允许访问实例成员变量和实例方法；实例方法则无此限制

44. 什么是多态？

允许不同类的对象对同一消息做出响应。即同一消息可以根据发送对象的不同而采用多种不同的行为方式。（发送消息就是函数调用）

45. 多态的优点？

可替换性（substitutability）。多态对已存在代码具有可替换性。例如，多态对圆 Circle 类工作，对其他任何圆形几何体，如圆环，也同样工作。

可扩充性（extensibility）。多态对代码具有可扩充性。增加新的子类不影响已存在类的多态性、继承性，以及其他特性的运行和操作。实际上新加子类更容易获得多态功能。

46. 多态存在的三个必要条件？

要有继承。

要有方法的重写。

父类引用指向子类对象（对于父类中定义的方法，如果子类中重写了该方法，那么父类类型的引用将会调用子类中的这个方法，这就是动态连接）

47. TreeMap、HashMap、LindedHashMap 的区别？

LinkedHashMap 可以保证 HashMap 集合有序。存入的顺序和取出的顺序一致。TreeMap 实现 SortMap 接口，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 TreeMap 时，得到的记录是排过序的。HashMap 不保证顺序，即为无序的，具有很快的访问速度。HashMap 最多只允许一条记录的键为 Null；允许多条记录的值为 Null；HashMap 不支持线程的同步。

48. Java（OOP）面向对象的特征有哪些方面？

1) 抽象：抽象是将一类对象的共同特征总结出来构造类的过程，包括数据抽象和行为抽象两方面。抽象只关注对象有哪些属性和行为，并不关注这些行为的细节是什么。

2) 继承：继承是从已有类得到继承信息创建新类的过程。提供继承信息的类被称为父类（超类、基类）；得到继承信息的类被称为子类（派生类）。继承让变化中的软件系统有了一定的延续性，同时继承也是封装程序中可变因素的重要手段

3) 封装：通常认为封装是把数据和操作数据的方法绑定起来，对数据的访问只能通过已定义的接口。面向对象的本质就是将现实世界描绘成一系列完全自治、封闭的对象。我们在类中编写的方法就是对实现细节的一种封装；我们编写一个类就是对数据和数据操作的封装。可以说，封装就是隐藏一切可隐藏的东西，只向外界提供最简单的编程接口。

4) 多态性：多态性是指允许不同子类型的对象对同一消息作出不同的响应。简单的说就是用同样的对象引用调用同样的方法但是做了不同的事情。多态性分为编译时的多态性和运行时的多态性。如果将对象的方法视为对象向外界提供的服务，那么运行时的多态性可以解释为：当 A 系统访问 B 系统提供的服务时，B 系统有多种提供服务的方式，但一切对 A 系统来说都是透明的。方法重载（overload）实现的是编译时的多态性（也称为前绑定），而方法重写（override）实现的是运行时的多态性（也称为后绑定）。运行时的多态是面向对象最精髓的东西，要实现多态需要做两件事：1. 方法重写（子类继承父类并重写父类中已有的或抽象的方法）；2. 对象造型（用父类型引用引用子类型对象，这样同样的引用调用同样的方法就会根据子类对象的不同而表现出不同的行为）。

49. 列出一些常见的运行时异常？

ArithmeticException（算术异常）
ClassCastException（类转换异常）
IllegalArgumentException（非法参数异常）
IndexOutOfBoundsException（下标越界异常）
NullPointerException（空指针异常）
SecurityException（安全异常）

50. 什么是反射？

反射就是动态加载对象，并对对象进行剖析。在运行状态中，对于任意一个类，都能够知道这个类的所有属性和方法；对于任意一个对象，都能够调用它的任意一个方法，这种动态获取信息以及动态调用对象方法的功能成为 Java 反射机制。

51. 反射的作用？

- 1) 在运行时判断任意一个对象所属的类
- 2) 在运行时构造任意一个类的对象
- 3) 在运行时判断任意一个类所具有的成员变量和方法
- 4) 在运行时调用任意一个对象的方法

52. 获取 class 的三种方式？

对象调用 getClass() 方法来获取；类名.class 的方式得到；通过 Class 对象的 forName() 静态方法来获取

53. break 和 continue 的区别？

break 和 continue 都是用来控制循环的语句。break 用于完全结束一个循环，跳出循环体执行循环后面的语句。continue 用于跳过本次循环，继续下次循环。

54. 运行时异常与一般异常有何异同？

异常表示程序运行过程中可能出现的非正常状态，运行时异常表示虚拟机的通常操作中可能遇到的异常，是一种常见运行错误。java 编译器要求方法必须声明抛出可能发生的非运行时异常，但是并不要求必须声明抛出未被捕获的运行时异常。

55. List、Map、Set 三个接口存取元素时，各有什么特点？

List 以特定索引来存取元素，可以有重复元素。Set 不能存放重复元素（用对象的 equals() 方法来区分元素是否重复）。Map 保存键值对（key-value pair）映射，映射关系可以是一对一或多对一

56. Collection 和 Collections 的区别？

Collection 是集合类的上级接口，继承与他的接口主要有 Set 和 List。Collections 是针对集合类的一个帮助类，他提供一系列静态方法实现对各种集合的搜索、排序、线程安全化等操作。

57. Error 和 Exception 有什么区别？

error 表示恢复不是不可能但很困难的情况下的一种严重问题。比如说内存溢出。不可能指望程序能处理这样的情况。exception 表示一种设计或实现问题。也就是说，它表示如果程序运行正常，从不会发生的情况。

58. EJB 的生命周期，以及如何管理事务？

SessionBean: Stateless Session Bean 的生命周期是由容器决定的，当客户机发出请求要建立一个 Bean 的实例时，EJB 容器不一定要创建一个新的 Bean 的实例供客户机调用，而是随便找一个现有的实例提供给客户机。当客户机第一次调用一个 Stateful Session Bean 时，容器必须立即在服务器中创建一个新的 Bean 实例，并关联到客户机上，以后此客户机调用 Stateful Session Bean 的方法时容器会把调用分派到与此客户机相关联的 Bean 实例。

EntityBean: Entity Beans 能存活相对较长的时间，并且状态是持续的。只要数据库中的数据存在，Entity beans 就一直存活。而不是按照应用程序或者服务进程来说的。即使 EJB 容器崩溃了，Entity beans 也是存活的。Entity Beans 生命周期能够被容器或者 Beans 自己管理。EJB 通过以下技术管理实务：对象管理组织（OMG）的对象实务服务（OTS），Sun Microsystems 的 Transaction Service(JTS)、Java Transaction API(JTA)，开发组（X/Open）的 XA 接口。

59. Comparable 和 Comparator 接口的区别？

Comparable 接口只包含一个 compareTo()方法。这个方法可以个给两个对象排序。具体来说，它返回负数，0，正数来表明输入对象小于，等于，大于已经存在的对象。Comparator 接口包含 compare()和 equals()两个方法。

60. switch 能否作用在 byte、long、string 上？

switch 可作用在 char、byte、short、int
switch 可作用于 char、byte、short、int 的包装类上
switch 不可作用于 long、double、float、boolean，包括他们的包装类
switch 中可以是字符串类型，String（Java1.7 以后才可以作用在 String 上）
switch 可以是枚举类型（JDK1.5 之后）

61. jdk 中哪些类是不能继承的？

不能继承的是类是那些用 final 关键字修饰的类。一般比较基本的类型或防止扩展类无意间破坏原来方法的实现的类型都应该是 final 的，在 jdk 中 System、String、StringBuffer 等都是基本类型。

62. Set 里的元素是不能重复的，那么用什么方法来区分重复与否呢？

Set 里的元素是不能重复的，元素重复与否是使用 equals() 方法进行判断的。
equals() 和 == 方法决定引用值是否指向同一对象，equals() 在类中被覆盖，为的是当两个分离的对象的内容和类型相配的话，返回真值。

63. JDK 和 JRE 的区别是什么？

Java 运行时环境(JRE)是将要执行 Java 程序的 Java 虚拟机。它同时也包含了执行 applet 需要的浏览器插件。Java 开发工具包(JDK)是完整的 Java 软件开发包，包含了 JRE，编译器和其他的工具(比如：JavaDoc，Java 调试器)，可以让开发者开发、编译、执行 Java 应用程序。

64. 是否可以在 static 环境中访问非 static 变量？

static 变量在 Java 中是属于类的，它在所有的实例中的值是一样的。当类被 Java 虚拟机载入的时候，会对 static 变量进行初始化。如果你的代码尝试不用实例来访问非 static 的变量，编译器会报错，因为这些变量还没有被创建出来，还没有跟任何实例关联上。

65. Java 支持多继承么？

不支持，Java 不支持多继承。每个类都只能继承一个类，但是可以实现多个接口。

66. 什么是迭代器(Iterator)？

Iterator 接口提供了很多对集合元素进行迭代的方法。每一个集合类都包含了可以返回迭代器实例的

迭代方法。迭代器可以在迭代的过程中删除底层集合的元素。

克隆(cloning)或者是序列化(serialization)的语义和含义是跟具体的实现相关的。因此，应该由集合类的具体实现来决定如何被克隆或者是序列化。

67. Iterator 和 ListIterator 的区别是什么？

下面列出了他们的区别：

Iterator 可用来遍历 Set 和 List 集合，但是 ListIterator 只能用来遍历 List。

Iterator 对集合只能是前向遍历，ListIterator 既可以前向也可以后向。

ListIterator 实现了 Iterator 接口，并包含其他的功能，比如：增加元素，替换元素，获取前一个和后一个元素的索引，等等。

68. Enumeration 接口和 Iterator 接口的区别有哪些？

Enumeration 速度是 Iterator 的 2 倍，同时占用更少的内存。但是， Iterator 远远比 Enumeration 安全，因为其他线程不能够修改正在被 iterator 遍历的集合里面的对象。同时， Iterator 允许调用者删除底层集合里面的元素，这对 Enumeration 来说是不可能的。

69. List, Set, Map 是否继承自 Collection 接口？

只有 List 和 Set 接口继承于 Collection 接口， Map 是与 Collection 并列的接口概念

70. 字符串常量池到底存在于内存空间的哪里？

jdk 6.0 字符串常量池在方法区，方法区的具体体现可以看做是堆中的永久区。

jdk 7.0 java 虚拟机规范中不再声明方法区，字符串常量池存放在堆空间中 jdk

8.0 java 虚拟机规范中又声明了元空间，字符串常量池存放在元空间中

71. Java 中的编译期常量是什么？使用它又什么风险？

公共静态不可变（`public static final`）变量也就是我们所说的编译期常量，这里的 `public` 可选的。实际上这些变量在编译时会被替换掉，因为编译器知道这些变量的值，并且知道这些变量在运行时不能改变。这种方式存在的一个问题是你使用了一个内部的或第三方库中的公有编译时常量，但是这个值后面被其他人改变了，但是你的客户端仍然在使用老的值，甚至你已经部署了一个新的 jar。为了避免这种情况， 当你在更新依赖 JAR 文件时，确保重新编译你的程序。

72. 用哪两种方式来实现集合的排序？

你可以使用有序集合，如 `TreeSet` 或 `TreeMap`，你也可以使用有顺序的的集合，如 `list`，然后通过 `Collections.sort()` 来排序。

73. 说出 JDK 1.7 中的三个新特性？

虽然 JDK 1.7 不像 JDK 5 和 8 一样的大版本，但是，还是有很多新的特性，如 `try-with-resource` 语句，这样你在使用流或者资源的时候，就不需要手动关闭，Java 会自动关闭。
`Fork-Join` 池某种程度上实现 Java 版的 `Map-reduce`。允许 `Switch` 中有 `String` 变量和文本。
菱形操作符(`<>`)用于类型推断，不再需要在变量声明的右边申明泛型，因此可以写出可读写更强、更简洁的代码。另一个值得一提的特性是改善异常处理，如允许在同一个 `catch` 块中捕获多个异常。

74. 说出 5 个 JDK 1.8 引入的新特性？

Java 8 在 Java 历史上是一个开创新的版本，下面 JDK 8 中 5 个主要的特性： Lambda 表达式，允许像对象一样传递匿名函数 Stream API，充分利用现代多核 CPU，可以写出很简洁的代码 Date 与 Time API，最终，有一个稳定、简单的日期和时间库可供你使用 扩展方法，现在，接口中可以有静态、默认方法。 重复注解，现在你可以将相同的注解在同一类型上使用多次。

75. ArrayList 源码分析？

(1) ArrayList 是一种变长的集合类，基于定长数组实现，使用默认构造方法初始化出来的容量是 10（1.7 之后都是延迟初始化，即第一次调用 add 方法添加元素的时候才将 elementData 容量初始化为 10）。

(2) ArrayList 允许空值和重复元素，当往 ArrayList 中添加的元素数量大于其底层数组容量时，其会通过扩容机制重新生成一个更大的数组。ArrayList 扩容的长度是原长度的 1.5 倍

(3) 由于 ArrayList 底层基于数组实现，所以其可以保证在 O(1) 复杂度下完成随机查找操作。

(4) ArrayList 是非线程安全类，并发环境下，多个线程同时操作 ArrayList，会引发不可预知的异常或错误。

(5) 顺序添加很方便

(6) 删除和插入需要复制数组，性能差（可以使用 LinkindList）

(7) Integer.MAX_VALUE - 8：主要是考虑到不同的 JVM,有的 JVM 会在加入一些数据头,当扩容后的容量大于MAX_ARRAY_SIZE,我们会去比较最小需要容量和MAX_ARRAY_SIZE 做比较,如果比它大,只能取 Integer.MAX_VALUE,否则是 Integer.MAX_VALUE -8。 这个是从 jdk1.7 开始才有的

76. HashMap 源码分析？

jdk1.8 之前 list + 链表

jdk1.8 之后 list + 链表（当链表长度到 8 时，转化为红黑树）

HashMap 的扩容因子

默认 0.75，也就是会浪费 1/4 的空间，达到扩容因子时，会将 list 扩容一倍，0.75 是时间与空间一个平衡值；

77. ConcurrentHashMap 源码分析？

ConcurrentHashMap 所使用的锁分段技术，首先将数据分成一段一段的存储，然后给每一段数据配一把锁，当一个线程占用锁访问其中一个段数据的时候，其他段的数据也能被其他线程访问。有些方法需要跨段，比如 size()和 containsValue()，它们可能需要锁定整个表而不仅仅是某个段，这需要按顺序锁定所有段，操作完毕后，又按顺序释放所有段的锁。这里“按顺序”是很重要的，否则极有可能出现死锁，在 ConcurrentHashMap 内部，段数组是 final 的，并且其成员变量实际上也是 final 的，但是，仅仅是将数组声明为final 的并不保证数组成员也是 final 的，这要实现上的保证。这可以确保不会出现死锁，因为获得锁的顺序是固定的。

ConcurrentHashMap 是由 Segment 数组结构和 HashEntry 数组结构组成。Segment 是一种可重入锁 ReentrantLock，在 ConcurrentHashMap 里扮演锁的角色，HashEntry 则用于存储键值对数据。一个 ConcurrentHashMap 里包含一个 Segment 数组，Segment 的结构和 HashMap 类似，是一种数组和链表结构，一个 Segment 里包含一个 HashEntry 数组，每个 HashEntry 是一个链表结构的元素，每个 Segment 守护者一个 HashEntry 数组里的元素，当对 HashEntry 数组的数据进行修改时，必须首先获得它对应的 Segment 锁。

二、Java IO

1. IO 里面的常见类，字节流、字符流、接口、实现类、方法阻塞？

输入流就是从外部文件输入到内存，输出流主要是从内存输出到文件。

IO 里面常见的类，第一印象就只知道 IO 流中有很多类，IO 流主要分为字符流和字节流。字符流中有抽象类 InputStream 和 OutputStream，它们的子类 FileInputStream，FileOutputStream, BufferedOutputStream 等。字符流 BufferedReader 和 Writer 等。都实现了 Closeable, Flushable, Appendable 这些接口。程序中的输入输出都是以流的形式保存的，流中保存的实际上全都是字节文件。

java 中的阻塞式方法是指在程序调用该方法时，必须等待输入数据可用或者检测到输入结束或者抛出异常，否则程序会一直停留在该语句上，不会执行下面的语句。比如 read() 和 readLine() 方法。

2. 谈谈对 NIO 的认知？

对于 NIO，它是非阻塞式，核心类：

1. Buffer 为所有的原始类型提供 (Buffer) 缓存支持。
2. Charset 字符集编码解码解决方案
3. Channel 一个新的原始 I/O 抽象，用于读写 Buffer 类型，通道可以认为是一种连接，可以是到特定设备，程序或者是网络的连接。

3. 字节流和字符流的区别？

字符流和字节流的使用非常相似，但是实际上字节流的操作不会经过缓冲区（内存）而是直接操作文本本身的，而字符流的操作会先经过缓冲区（内存）然后通过缓冲区再操作文件以字节为单位输入输出数据，字节流按照 8 位传输
以字符为单位输入输出数据，字符流按照 16 位传输

4. NIO 和传统的 IO 有什么区别？

- 1、传统 IO 一般是一个线程等待连接，连接过来之后分配给 processor 线程，processor 线程与通道连接后如果通道没有数据过来就会阻塞（线程被动挂起）不能做别的事情。NIO 则不同，首先，在 selector 线程轮询的过程中就已经过滤掉了不感兴趣的事件，其次，在 processor 处理感兴趣事件的 read 和 write 都是非阻塞操作即直接返回的，线程没有被挂起。
- 2、传统 io 的管道是单向的，nio 的管道是双向的。

3、两者都是同步的，也就是 java 程序亲力亲为的去读写数据，不管传统 io 还是 nio 都需要 read 和 write 方法，这些都是 java 程序调用的而不是系统帮我们调用的，nio2.0 里这点得到了改观，即使用异步非阻塞 AsynchronousXXX 四个类来处理。

5. BIO 和 NIO 和 AIO 的区别以及应用场景？

同步：java 自己去处理 io。

异步：java 将 io 交给操作系统去处理，告诉缓存区大小，处理完成回调。

阻塞：使用阻塞 IO 时，Java 调用会一直阻塞到读写完成才返回。

非阻塞：使用非阻塞 IO 时，如果不能立马读写，Java 调用会马上返回，当 IO 事件分发器通知可读写时在进行读写，不断循环直到读写完成。

BIO：同步并阻塞，服务器的实现模式是一个连接一个线程，这样的模式很明显的一个缺陷是：由于客户端连接数与服务器线程数成正比关系，可能造成不必要的线程开销，严重的还将导致服务器内存溢出。当然，这种情况可以通过线程池机制改善，但并不能从本质上消除这个弊端。

NIO：在 JDK1.4 以前，Java 的 IO 模型一直是 BIO，但从 JDK1.4 开始，JDK 引入的新的 IO 模型 NIO，它是同步非阻塞的。而服务器的实现模式是多个请求一个线程，即请求会注册到多路复用器 Selector 上，多路复用器轮询到连接有 IO 请求时才启动一个线程处理。AIO：

JDK1.7 发布了 NIO2.0，这就是真正意义上的异步非阻塞，服务器的实现模式为多个有效请求一个线程，客户端的 IO 请求都是由 OS 先完成再通知服务器应用去启动线程处理（回调）。

应用场景：并发连接数不多时采用 BIO，因为它编程和调试都非常简单，但如果涉及到高并发的情况，应选择 NIO 或 AIO，更好的建议是采用成熟的网络通信框架 Netty。

6. 什么是 Java 序列化，如何实现 Java 序列化？

序列化就是一种用来处理对象流的机制，将对象的内容进行流化。可以对流化后的对象进行读写操作，可以将流化后的对象传输于网络之间。序列化是为了解决在对象流读写操作时所引发的问题

序列化的实现：将需要被序列化的类实现 Serialize 接口，没有需要实现的方法，此接口只是为了标注对象可被序列化的，然后使用一个输出流（如：FileOutputStream）来构造一个 ObjectOutputStream(对象流)对象，再使用 ObjectOutputStream 对象的 write(Object obj)方法就可以将参数 obj 的对象写出

7. PrintStream、BufferedWriter、PrintWriter 的比较？

PrintStream 类的输出功能非常强大，通常如果需要输出文本内容，都应该将输出流包装成 PrintStream 后进行输出。它还提供其他两项功能。与其他输出流不同，PrintStream 永远不会抛出 IOException；而是，异常情况仅设置可通过 checkError 方法测试的内部标志。另外，为了自动刷新，可以创建一个 PrintStream

2、BufferedWriter:将文本写入字符输出流，缓冲各个字符从而提供单个字符，数组和字符串的高效写入。通过 write()方法可以将获取到的字符输出，然后通过 newLine()进行换行操作。BufferedWriter 中的字符流必须通过调用 flush 方法才能将其刷出去。并且 BufferedWriter 只能对字符流进行操作。如果要对字节流操作，则使用 BufferedInputStream

3、PrintWriter 的 println 方法自动添加换行，不会抛异常，若关心异常，需要调用 checkError

方法看是否有异常发生，`PrintWriter` 构造方法可指定参数，实现自动刷新缓存（`autoflush`）

8. 什么是节点流，什么是处理流，各有什么好处，处理流的创建有什么特征？

节点流 直接与数据源相连，用于输入或者输出

处理流：在节点流的基础上对之进行加工，进行一些功能的扩展 处理流的构造器必须要 传入节点流的子类

9. 什么是 IO 流？

它是一种数据的流从源头流到目的地。比如文件拷贝，输入流和输出流都包括了。输入流从 文件中读取数据存储到进程(process)中，输出流从进程中读取数据然后写入到目标文件。

10. 有哪些可用的 Filter 流？

在 `java.io` 包中主要由 4 个可用的 filter Stream。两个字节 filter stream，两个字符 filter stream。分别是 `FilterInputStream`, `FilterOutputStream`, `FilterReader` and `FilterWriter`.这些类是抽象类，不能被实例化的。

11. Java 中有几种类型的流？

按照流的方向：输入流（`InputStream`）和输出流（`OutputStream`）

按照实现功能分：节点流（可以从或向一个特定的地方（节点）读写数据。如 `FileReader`）和处理流（是对一个已存在的流的连接和封装，通过所封装的流的功能调用实现数据读写。

如 `BufferedReader`。处理流的构造方法总是要带一个其他的流对象做参数。一个流对象经过其他流的多次包装，称为流的链接。）

按照处理数据的单位：字节流和字符流。字节流继承于 `InputStream` 和 `OutputStream`，字符流继承于 `InputStreamReader` 和 `OutputStreamWriter`。

12. 如何实现对象克隆？

有两种方式：

- 1). 实现 `Cloneable` 接口并重写 `Object` 类中的 `clone()`方法；
- 2). 实现 `Serializable` 接口，通过对象的序列化和反序列化实现克隆，可以实现真正的深度克隆

13. 什么是缓冲区？有什么作用？

缓冲区就是一段特殊的内存区域，很多情况下当程序需要频繁地操作一个资源（如文件或数据库）则性能会很低，所以为了提升性能就可以将一部分数据暂时读写到缓存区，以后直接从此区域中读写数据即可，这样就可以显著的提升性能。

对于 Java 字符流的操作都是在缓冲区操作的，所以如果我们想在字符流操作中主动将缓冲区刷新到文件则可以使用 `flush()` 方法操作。

14. 什么是阻塞 IO？什么是非阻塞 IO？

IO 操作包括：对硬盘的读写、对 socket 的读写以及外设的读写。

当用户线程发起一个 IO 请求操作（本文以读请求操作为例），内核会去查看要读取的数据是否就绪，对于阻塞 IO 来说，如果数据没有就绪，则会一直在那等待，直到数据就绪；对于非阻塞 IO 来说，如果数据没有就绪，则会返回一个标志信息告知用户线程当前要读的数据没有就绪。当数据就绪之后，便将数据拷贝到用户线程，这样才完成了一个完整的 IO 读请求操作，也就是说一个完整的 IO 读请求操作包括两个阶段：

- 1) 查看数据是否就绪；
- 2) 进行数据拷贝（内核将数据拷贝到用户线程）。

那么阻塞（blocking IO）和非阻塞（non-blocking IO）的区别就在于第一个阶段，如果数据没有就绪，在查看数据是否就绪的过程中是一直等待，还是直接返回一个标志信息。

Java 中传统的 IO 都是阻塞 IO，比如通过 socket 来读数据，调用 read()方法之后，如果数据没有就绪，当前线程就会一直阻塞在 read 方法调用那里，直到有数据才返回；而如果是非阻塞 IO 的话，当数据没有就绪，read()方法应该返回一个标志信息，告知当前线程数据没有就绪，而不是一直在那里等待。

三、Java Web

1. session 和 cookie 的区别？

session 是存储在服务器端，cookie 是存储在客户端的，所以安全来讲 session 的安全性要比 cookie 高，然后我们获取 session 里的信息是通过存放在会话 cookie 里的 sessionid 获取的。又由于 session 是存放在服务器的内存中，所以 session 里的东西不断增加会造成服务器的负担，所以会把很重要的信息存储在 session 中，而把一些次要东西存储在客户端的 cookie 里，然后 cookie 确切的分为两大类分为会话 cookie 和持久化 cookie，会话 cookie 确切的说是，存放在客户端浏览器的内存中，所以说他的生命周期和浏览器是一致的，浏览器关了会话 cookie 也就消失了，然而持久化 cookie 是存放在客户端硬盘中，而持久化 cookie 的生命周期就是我们在设置 cookie 时候设置的那个保存时间，然后我们考虑一问题当浏览器关闭时 session 会不会丢失，从上面叙述分析 session 的信息是通过会话 cookie 的 sessionid 获取的，当浏览器关闭的时候会话 cookie 消失所以我们的 sessionid 也就消失了，但是 session 的信息还存在服务器端，这时我们只是查不到所谓的 session 但它并不是不存在。那么，session 在什么情况下丢失，就是在服务器关闭的时候，或者是 session 过期(默认时间是 30 分钟)，再或者调用了 invalidate() 的或者是我们想要 session 中的某一条数据消失调用 session.removeAttribute()方法，然后 session 在什么时候被创建呢，确切的说是通过调用 getSession()来创建，这就是 session 与 cookie 的区别。

2、session 和 cookie 联系？

session 是通过 cookie 来工作的 session 和 cookie 之间是通过 \$_COOKIE['PHPSESSID']来联系的，通过 \$_COOKIE['PHPSESSID']可以知道 session 的 id，从而获取到其他的信息。在购物网站中通常将用户加入购物车的商品联通 session_id 记录到数据库中，当用户再次访问是，通过 sessionid 就可以查找到用户上次加入购物车的商品。因为 sessionid 是唯一的，记录到数据库中就可以根据这个查找了。

3. servlet 的生命周期?

Servlet 生命周期可以分成四个阶段：加载和实例化、初始化、服务、销毁。

当客户第一次请求时，首先判断是否存在 Servlet 对象，若不存在，则由 Web 容器创建对象，而后调用 `init()` 方法对其初始化，此初始化方法在整个 Servlet 生命周期中只调用一次。

完成 Servlet 对象的创建和实例化之后，Web 容器会调用 Servlet 对象的 `service()` 方法来处理请求。

当 Web 容器关闭或者 Servlet 对象要从容器中被删除时，会自动调用 `destory()` 方法。

4. 什么是 webservice?

从表面上看，WebService 就是一个应用程序向外界暴露出一个能通过 Web 进行调用的 API，也就是说能用编程的方法通过 Web 来调用这个应用程序。我们把调用这个 WebService 的应用程序叫做客户端，而把提供这个 WebService 的应用程序叫做服务端。从深层次看，WebService 是建立可互操作的分布式应用程序的新平台，是一个平台，是一套标准。它定义了应用程序如何在 Web 上实现互操作性，你可以用任何你喜欢的语言，在任何你喜欢的平台上写 Web service，只要我们可以通过 Web service 标准对这些服务进行查询和访问。

5. jsp 和 servlet 的区别、共同点、各自应用的范围?

JSP 是 Servlet 技术的扩展，本质上就是 Servlet 的简易方式。JSP 编译后是“类 servlet”。Servlet 和 JSP 最主要的不同点在于，Servlet 的应用逻辑是在 Java 文件中，并且完全从表示层中的 HTML 里分离开来。而 JSP 的情况是 Java 和 HTML 可以组合成一个扩展名为 .jsp 的文件。JSP 侧重于视图，Servlet 主要用于控制逻辑。在 struts 框架中，JSP 位于 MVC 设计模式的视图层，而 Servlet 位于控制层。

6. 转发 (forward) 和重定向 (redirect) 的区别?

1. 从地址栏显示来说

forward 是服务器请求资源,服务器直接访问目标地址的URL,把那个URL 的响应内容读取过来,然后把这些内容再发给浏览器.浏览器根本不知道服务器发送的内容从哪里来的,所以它的地址栏还是原来的地址.

redirect 是服务端根据逻辑,发送一个状态码,告诉浏览器重新去请求那个地址.所以地址栏显示的是新的 URL.

2. 从数据共享来说

forward:转发页面和转发到的页面可以共享 request 里面的数据.

redirect:不能共享数据.

3. 从运用地方来说

forward:一般用于用户登陆的时候,根据角色转发到相应的模块.

redirect:一般用于用户注销登陆时返回主页面和跳转到其它的网站等

4. 从效率来说

forward: 高 .

redirect:低.

7. request.getAttribute() 和 request.getParameter() 有何区别?

1、`request.getParameter()`取得是通过容器的实现来取得通过类似 post, get 等方式传入的数

据。

2、request.setAttribute()和 getAttribute()只是在 web 容器内部流转，仅仅是请求处理阶段。

3、getAttribute 是返回对象,getParameter 返回字符串

4、getAttribute()一向是和 setAttribute()一起使用的，只有先用 setAttribute()设置之后，才能够通过 getAttribute()来获得值，它们传递的是 Object 类型的数据。而且必须在同一个 request 对象中使用才有效。而 getParameter()是接收表单的 get 或者 post 提交过来的参数

8. jsp 静态包含和动态包含的区别？

1. 两者格式不同，静态包含：<%@ include file="文件" %>，而动态包含：<jsp: include page = "文件" />。

2. 包含时间不同，静态包含是先将几个文件合并，然后再被编译，缺点就是如果含有相同的 标签，会出错。动态包含是页面被请求时编译，将结果放在一个页面。

3. 生成的文件不同，静态包含会生成一个包含页面名字的 servlet 和 class 文件；而动态包含会各自生成对应的 servlet 和 class 文件

4. 传递参数不同，动态包含能够传递参数，而静态包含不能

9. MVC 的各个部分都有哪些技术来实现？如何实现？

MVC 是 Model—View—Controller 的简写。"Model" 代表的是应用的业务逻辑（通过 JavaBean，EJB 组件实现），"View" 是应用的表示面（由 JSP 页面产生），"Controller" 是提供应用的处理过程控制（一般是一个 Servlet），通过这种设计模型把应用逻辑，处理过程和显示逻辑分成不同的组件实现。这些组件可以进行交互和重用。

10. jsp 有哪些内置对象？作用分别是什么？

JSP 有 9 个内置对象：

request：封装客户端的请求，其中包含来自 GET 或 POST 请求的参数；

response：封装服务器对客户端的响应；

pageContext：通过该对象可以获取其他对象；

session：封装用户会话的对象；

application：封装服务器运行环境的对象；

out：输出服务器响应的输出流对象；

config：Web 应用的配置对象；

page：JSP 页面本身（相当于 Java 程序中的 this）；

exception：封装页面抛出异常的对象。

11. Http 请求的 get 和 post 方法的区别。

1、Get 是向服务器发索取数据的一种请求，而 Post 是向服务器提交数据的一种请求

2、Get 是获取信息，而不是修改信息，类似数据库查询功能一样，数据不会被修改

3、Get 请求的参数会跟在 url 后进行传递，请求的数据会附在 URL 之后，以?分割 URL 和传输数据，参数之间以&相连,%XX 中的 XX 为该符号以 16 进制表示的 ASCII，如果数据是英文字母/数字，原样发送，如果是空格，转换为+，如果是中文/其他字符，则直接把字符串用

BASE64 加密。

4、Get 传输的数据有大小限制，因为 GET 是通过 URL 提交数据，那么 GET 可提交的数据量就跟 URL 的长度有直接关系了，不同的浏览器对 URL 的长度的限制是不同的。

5、GET 请求的数据会被浏览器缓存起来，用户名和密码将明文出现在 URL 上，其他人可以查到历史浏览记录，数据不太安全。

在服务器端，用 `Request.QueryString` 来获取 Get 方式提交来的数据

Post 请求则作为 http 消息的实际内容发送给 web 服务器，数据放置在 HTML Header 内提交，Post 没有限制提交的数据。Post 比 Get 安全，当数据是中文或者不敏感的数据，则用 `get`，因为使用 `get`，参数会显示在地址，对于敏感数据和不是中文字符的数据，则用 `post`。

6、POST 表示可能修改服务器上的资源的请求，在服务器端，用 Post 方式提交的数据只能用 `Request.Form` 来获取。

12. tomcat 容器是如何创建 servlet 类实例？用到了什么原理？

当容器启动时，会读取在 `webapps` 目录下所有的 web 应用中的 `web.xml` 文件，然后对 xml 文件进行解析，并读取 `servlet` 注册信息。然后，将每个应用中注册的 `servlet` 类都进行加载，并通过反射的方式实例化。（有时候也是在第一次请求时实例化）

在 `servlet` 注册时加上 `<load-on-startup>1</load-on-startup>` 如果为正数，则在一开始就实例化，如果不写或为负数，则第一次请求实例化。

13. JDBC 访问数据库的基本步骤是什么？

第一步：`Class.forName()`加载数据库连接驱动；

第二步：`DriverManager.getConnection()`获取数据连接对象；

第三步：根据 SQL 获取 sql 会话对象，有 2 种方式 `Statement`、`PreparedStatement`；

第四步：执行 SQL，执行 SQL 前如果有参数值就设置参数值 `setXXX()`；

第五步：处理结果集；

第六步：关闭结果集、关闭会话、关闭连接。

14. 为什么要使用 PreparedStatement？

`PreparedStatement` 接口继承 `Statement`，`PreparedStatement` 实例包含已编译的 SQL 语句，所以其执行速度要快于 `Statement` 对象。

作为 `Statement` 的子类，`PreparedStatement` 继承了 `Statement` 的所有功能。三种方法 `execute`、`executeQuery` 和 `executeUpdate` 已被更改以使之不再需要参数。

在 JDBC 应用中，多数情况下使用 `PreparedStatement`，原因如下：

代码的可读性和可维护性。`Statement` 需要不断地拼接，而 `PreparedStatement` 不会。

`PreparedStatement` 尽最大可能提高性能。DB 有缓存机制，相同的预编译语句再次被调用不会再次需要编译。

最重要的一点是极大地提高了安全性。`Statement` 容易被 SQL 注入，而 `PreparedStatement` 传入的内容不会和 sql 语句发生任何匹配关系。

15. 数据库连接池的原理。为什么要使用连接池？

- 1、数据库连接是一件费时的操作，连接池可以使多个操作共享一个连接。
- 2、数据库连接池的基本思想就是为数据库连接建立一个“缓冲池”。预先在缓冲池中放入一定数量的连接，当需要建立数据库连接时，只需从“缓冲池”中取出一个，使用完毕之后再放回去。我们可以通过设定连接池最大连接数来防止系统无尽的与数据库连接。更为重要的是 我们可以通过连接池的管理机制监视数据库的连接的数量、使用情况，为系统开发，测试及 性能调整提供依据。
- 3、使用连接池是为了提高对数据库连接资源的管理

16. execute, executeQuery, executeUpdate 的区别是什么？

- 1、Statement 的 execute(String query)方法用来执行任意的 SQL 查询，如果查询的结果是一个 ResultSet，这个方法就返回 true。如果结果不是 ResultSet，比如 insert 或者 update 查询，它就会返回 false。 我们可以通过它的 getResultSet 方法来获取 ResultSet， 或者通过 getUpdateCount()方法来获取更新的记录条数。
- 2、Statement 的 executeQuery(String query)接口用来执行 select 查询，并且返回 ResultSet。即使查询不到记录返回的 ResultSet 也不会为 null。我们通常使用 executeQuery 来执行查询语句， 这样的话如果传进来的是 insert 或者 update 语句的话， 它会抛出错误信息为“executeQuery method can not be used for update”的 java.util.SQLException。 ,
- 3、Statement 的 executeUpdate(String query)方法用来执行 insert 或者 update/delete (DML) 语句，或者 什么也不返回，对于 DDL 语句，返回值是 int 类型，如果是 DML 语句的话，它就是更新的条数，如果是 DDL 的话，就返回 0。

只有当你不确定是什么语句的时候才应该使用 execute() 方法， 否则应该使用 executeQuery 或者 executeUpdate 方法。

17. JDBC 的 ResultSet 是什么？

在查询数据库后会返回一个 ResultSet，它就像是查询结果集的一张数据表。

ResultSet 对象维护了一个游标，指向当前的数据行。开始的时候这个游标指向的是第一行。如果调用了 ResultSet 的 next()方法游标会下移一行，如果没有更多的数据了，next()方法会返回 false。可以在 for 循环中用它来遍历数据集。

默认的 ResultSet 是不能更新的，游标也只能往下移。也就是说你只能从第一行到最后一行遍历一遍。不过也可以创建可以回滚或者可更新的 ResultSet

当生成 ResultSet 的 Statement 对象要关闭或者重新执行或是获取下一个 ResultSet 的时候，ResultSet 对象也会自动关闭。

可以通过 ResultSet 的 getter 方法，传入列名或者从 1 开始的序号来获取列数据。

18. 什么是 Servlet？

Servlet 是使用 Java Servlet 应用程序接口 (API) 及相关类和方法的 Java 程序，所有的 Servlet 都必须要实现的核心接口是 javax.servlet.Servlet。每一个 servlet 都必须直接或者间接实现这个接口，或者继承 javax.servlet.GenericServlet 或 javax.servlet.HttpServlet。

Servlet 主要用于处理客户端传来的 HTTP 请求，并返回一个响应。

19. doGet 和 doPost 方法有什么区别？

doGet: GET 方法会把名值对追加在请求的 URL 后面。因为 URL 对字符数目有限制，进而限制了用在客户端请求的参数值的数目。并且请求中的参数值是可见的，因此，敏感信息不能用这种方式传递。

doPOST: POST 方法通过把请求参数值放在请求体中来克服GET 方法的限制，因此，可以发送的参数数目是没有限制的。最后，通过 POST 请求传递的敏感信息对外部客户端是不可见的。

20. JSP 有哪些动作？分别是什么？

JSP 共有以下 6 种基本动作

jsp:include: 在页面被请求的时候引入一个文件。

jsp:useBean: 寻找或者实例化一个 **JavaBean**。

jsp:setProperty: 设置 **JavaBean** 的属性。

jsp:getProperty: 输出某个 **JavaBean** 的属性。

jsp:forward: 把请求转到一个新的页面。

jsp:plugin: 根据浏览器类型为 Java 插件生成 **OBJECT** 或 **EMBED** 标记

21. JSP 常用的指令？

page: 针对当前页面的指令。

include: 包含另一个页面

taglib: 定义和访问自定义标签

22. 页面间对象传递的方法？

request、**session**、**application**、**cookie**

23. JSP 中动态 INCLUDE 与静态 INCLUDE 的区别？

动态 include 用于 **jsp: include** 动作实现`<jsp: include page = "include.jsp" flush = "true"/>`它总是会检查所含文件的变化，适用于包含动态页面，并且可以带参数。

静态include 用 **include** 伪码实现，不会检查所含文件的变化，适用于包含静态页面`<%@include file= "include.html" %>`.

24. JSP 的四大范围？

JSP 中的四种作用域包括 **page**、**request**、**session** 和 **application**，具体来说：

page 代表与一个页面相关的对象和属性。

request 代表与 Web 客户机发出的一个请求相关的对象和属性。一个请求可能跨越多个页面，涉及多个 Web 组件；需要在页面显示的临时数据可以置于此作用域。

session 代表与某个用户与服务器建立的一次会话相关的对象和属性。跟某个用户相关的数据应该放在用户自己的 **session** 中。

application 代表与整个 Web 应用程序相关的对象和属性，它实质上是跨越整个 Web 应用程序，包括多个页面、请求和会话的一个全局作用域。

25. BS 与 CS 的联系与区别？

1. 硬件环境不同:

C/S 一般建立在专用的网络上, 小范围里的网络环境, 局域网之间再通过专门服务器提供连接和数据交换服务.

B/S 建立在广域网之上的, 不必是专门的网络硬件环境, 例与电话上网, 租用设备. 信息自己管理. 有比 C/S 更强的适应范围, 一般只要有操作系统和浏览器就行

2. 对安全要求不同
C/S 一般面向相对固定的用户群, 对信息安全的控制能力很强. 一般高度机密的信息系统采用 C/S 结构适宜. 可以通过 B/S 发布部分可公开信息.

B/S 建立在广域网之上, 对安全的控制能力相对弱, 可能面向不可知的用户。3. 对程序架构不同

C/S 程序可以更加注重流程, 可以对权限多层次校验, 对系统运行速度可以较少考虑.

B/S 对安全以及访问速度的多重的考虑, 建立在需要更加优化的基础之上. 比 C/S 有更高的要求 B/S 结构的程序架构是发展的趋势, 从 MS 的 .Net 系列的 BizTalk 2000 Exchange 2000 等, 全面支持网络的构件搭建的系统. SUN 和 IBM 推的 JavaBean 构件技术等, 使 B/S 更加成熟. 4. 软件重用不同

C/S 程序可以不可避免的整体性考虑, 构件的重用性不如在 B/S 要求下的构件的重用性好.

B/S 对的多重结构, 要求构件相对独立的功能. 能够相对较好的重用. 就入买来的餐桌可以再利用, 而不是做在墙上的石头桌子

5. 系统维护不同
C/S 程序由于整体性, 必须整体考察, 处理出现的问题以及系统升级. 升级难. 可能是再做一个全新的系统

B/S 构件组成, 方面构件个别的更换, 实现系统的无缝升级. 系统维护开销减到最小. 用户从网上自己下载安装就可以实现升级. 6. 处理问题不同

C/S 程序可以处理用户面固定, 并且在相同区域, 安全要求高需求, 与操作系统相关. 应该都是相同的系统

B/S 建立在广域网上, 面向不同的用户群, 分散地域, 这是 C/S 无法作到的. 与操作系统平台关系最小. 7. 用户接口不同

C/S 多是建立的 Window 平台上, 表现方法有限, 对程序员普遍要求较高

B/S 建立在浏览器上, 有更加丰富和生动的表现方式与用户交流. 并且大部分难度减低, 减低开发成本. 8. 信息流不同

C/S 程序一般是典型的中央集权的机械式处理, 交互性相对低
B/S 信息流向可变化, B-B B-C B-G 等信息、流向的变化, 更像交易中心。

26. 说出 Servlet 的生命周期, 并说出 Servlet 和 CGI 的区别？

Web 容器加载 Servlet 并将其实例化后, Servlet 生命周期开始, 容器运行其 init 方法进行 Servlet 的初始化, 请求到达时运行其 service 方法, service 方法自动派遣运行与请求对应的 doXXX 方法 (doGet, doPost) 等, 当服务器决定将实例销毁的时候调用其 destroy 方法。与 cgi 的区别在于 servlet 处于服务器进程中, 它通过多线程方式运行其 service 方法, 一个实例可以服务于多个请求, 并且其实例一般不会销毁, 而 CGI 对每个请求都产生新的进程, 服务完成后就销毁, 所以效率上低于 servlet。

27. 如何防止表单重复提交？

针对于重复提交的整体解决方案：

1. 用 `redirect`（重定向）来解决重复提交的问题
2. 点击一次之后，按钮失效
3. 通过 `loading` (`Loading` 原理是在点击提交时，生成 `Loading` 样式，在提交完成之后隐藏该样式)
4. 自定义重复提交过滤器

28. `request` 作用？

- 1、获取请求参数 `getParameter()`
- 2、获取当前 Web 应用的虚拟路径 `getContextPath`
- 3、转发 `getRequestDispatcher(路径).forward(request,response);`
- 4、它还是一个域对象

29. `get` 请求中文乱码？

- 1、乱码的根本原因：

浏览器的编码方式 `UTF-8` 和 服务器的解码方式 `ISO-8859-1` 不一样

- 2、解决方法：

1) 第一种方式 使用 `URLEncoder` 和 `URLDecoder` 两个类 编解码。先以 `iso-8859-1` 进行编码，然后再以 `utf-8` 进行解码

2) 第二种方式 使用 `String` 类的方法进行编解码

3) 第三种方式 更改 `server.xml` 配置文件。

`GET` 请求是在 `URL` 地址栏中传递请求参数的，它会被 `Tomcat` 服务器自动解码，而 `Tomcat` 服务器默认的字符集也是 `ISO-8859-1`，所以我们需要修改 `Tomcat` 服务器的字符集为 `UTF-8`。

30. `post` 请求中文乱码问题？

- 1、`post` 请求方式乱码的原因是：因为 `post` 是以二进制流的形式发送到的服务器。服务器收到数据后。默认以 `iso-8859-1` 进行编码。

- 2、`post` 请求乱码解决，只需要在获取请求参数之前调用

`request.setCharacterEncoding("UTF-8");` 方法设置字符集 即可。

31. 响应乱码？

- 1、原因：
由服务器编码，默认使用 `ISO-8859-1` 进行编码
由浏览器解码，默认使用 `GBK` 进行解码

- 2、解决方案

方法 1：设置响应头

`response.setHeader("Content-Type","text/html;charset=utf-8");`

方法 2：设置响应的内容类型

```
response.setContentType("text/html;charset=utf-8");
```

通过这种方式可以在响应头中告诉浏览器响应体的编码方式是 UTF-8; 同时服务器也会采用该字符集进行编码

但需要注意的是, 两种方法一定要在 `response.getWriter()`之前进行。

32. Cookie 对象的缺陷?

- 1、Cookie 是明文的, 不安全
 - 2、不同的浏览器对 Cookie 对象的数量和大小有限制
 - 3、Cookie 对象携带过多费流量
 - 4、Cookie 对象中的 value 值只能是字符串, 不能放对象
- 网络中传递数据只能是字符串

33. Session 的运行机制?

- 1、在服务器端创建 Session 对象, 该对象有一个全球唯一的 ID
- 2、在创建 Session 对象的同时创建一个特殊的 Cookie 对象, 该 Cookie 对象的名字是 JSESSIONID, 该 Cookie 对象的 value 值是 Session 对象的那个全球唯一的 ID, 并且会将这个特殊的 Cookie 对象携带发送给浏览器
- 3、以后浏览器再发送请求就会携带这个特殊的 Cookie 对象
- 4、服务器根据这个特殊的 Cookie 对象的 value 值在服务器中寻找对应的 Session 对象, 以此来区分不同的用户

34. 钝化和活化?

- 1、Session 与 session 域中的对象一起从内存中被序列化到硬盘上的过程我们称为钝化。服务器关闭时会发生钝化。
- 2、Session 与 session 域中的对象一起从硬盘上反序列化到内存中的过程我们称为活化。服务器再次开启时会发生活化。
- 3、要保证 session 域中的对象能和 Session 一起被钝化和活化, 必须保证对象对应的类实现 Serializable 接口

35. Filter 的工作原理?

Filter 接口中有一个 `doFilter` 方法, 当我们编写好 Filter, 并配置对哪个 web 资源进行拦截后, WEB 服务器每次在调用 web 资源的 `service` 方法之前, 都会先调用一下 filter 的 `doFilter` 方法, 因此, 在该方法内编写代码可达到如下目的:

调用目标资源之前, 让一段代码执行。

是否调用目标资源 (即是否让用户访问 web 资源)。

调用目标资源之后, 让一段代码执行。

web 服务器在调用 `doFilter` 方法时, 会传递一个 `filterChain` 对象进来, `filterChain` 对象是 filter 接口中最重要的一个对象, 它也提供了一个

`doFilter` 方法, 开发人员可以根据需求决定是否调用此方法, 调用该方法, 则 web 服务器就会调用 web 资源的 `service` 方法, 即 web 资源就会被访问, 否则 web 资源不会被访问

36. Filter 链是什么？

在一个 web 应用中，可以开发编写多个 Filter，这些 Filter 组合起来称之为一个 Filter 链。web 服务器根据 Filter 在 web.xml 文件中的注册顺序，决定先调用哪个 Filter，当第一个 Filter 的 doFilter 方法被调用时，web 服务器会创建一个代表 Filter 链的 FilterChain 对象传递给该方法。在 doFilter 方法中，开发人员如果调用了 FilterChain 对象的 doFilter 方法，则 web 服务器会检查 FilterChain 对象中是否还有 filter，如果有，则调用第 2 个 filter，如果没有，则调用目标资源。

37. 监听器类型？

按监听的对象划分：servlet2.4 规范定义的事件有三种：

- 1.用于监听应用程序环境对象（ServletContext）的事件监听器
 - 2.用于监听用户会话对象（HttpSession）的事件监听器
 - 3.用于监听请求消息对象（ServletRequest）的事件监听器
- 按监听的事件类项划分

- 1.用于监听域对象自身的创建和销毁的事件监听器
- 2.用于监听域对象中的属性的增加和删除的事件监听器
- 3.用于监听绑定到 HttpSession 域中的某个对象的状态的事件监听器

在一个 web 应用程序的整个运行周期内，web 容器会创建和销毁三个重要的对象，ServletContext, HttpSession, ServletRequest。

38. Servlet Filter Listener 启动顺序？

启动的顺序为 listener->Filter->servlet.

简单记为：理(Listener)发(Filter)师(servlet).

执行的顺序不会因为三个标签在配置文件中的先后顺序而改变。

四、JVM

1. Java 的内存划分？

程序计数器（PC，Program Counter Register）。在 JVM 规范中，每个线程都有它自己的程序计数器，并且任何时间一个线程都只有一个方法在执行，也就是所谓的当前方法。程序计数器会存储当前线程正在执行的 Java 方法的 JVM 指令地址；或者，如果是在执行本地方法，则是未指定值（undefined）。（唯一不会抛出 OutOfMemoryError）

第二，Java 虚拟机栈（Java Virtual Machine Stack），早期也叫 Java 栈。每个线程在创建时都会创建一个虚拟机栈，其内部保存一个个的栈帧（Stack Frame），对应着一次次的 Java 方法调用。

前面谈程序计数器时，提到了当前方法；同理，在一个时间点，对应的只会有一个活动的栈帧，通常叫作当前帧，方法所在的类叫作当前类。如果在该方法中调用了其他方法，对应的新的栈帧会被创建出来，成为新的当前帧，一直到它返回结果或者执行结束。JVM 直接对 Java 栈的操作只有两个，就是对栈帧的压栈和出栈。

栈帧中存储着局部变量表、操作数（operand）栈、动态链接、方法正常退出或者异常退出的定义等。

第三，堆（Heap），它是 Java 内存管理的核心区域，用来放置 Java 对象实例，几乎所有创建的 Java 对象实例都是被直接分配在堆上。堆被所有的线程共享，在虚拟机启动时，我们指定的“Xmx”之类参数就是用来指定最大堆空间等指标。

（编译器通过逃逸分析，确定对象是在栈上分配还是在堆上分配）

理所当然，堆也是垃圾收集器重点照顾的区域，所以堆内空间还会被不同的垃圾收集器进行进一步的细分，最有名的就是新生代、老年代的划分。

第四，方法区（Method Area）。这也是所有线程共享的一块内存区域，用于存储所谓的元（Meta）数据，例如类结构信息，以及对应的运行时常量池、字段、方法代码等。

由于早期的 Hotspot JVM 实现，很多人习惯于将方法区称为永久代（Permanent Generation）。

Oracle JDK 8 中将永久代移除，同时增加了元数据区（Metaspace）。

第五，运行时常量池（Run-Time Constant Pool），这是方法区的一部分。如果仔细分析过反编译的类文件结构，你能看到版本号、字段、方法、超类、接口等各种信息，还有一项信息就是常量池。Java 的常量池可以存放各种常量信息，不管是编译期生成的各种字面量，还是需要在运行时决定的符号引用，所以它比一般语言的符号表存储的信息更加宽泛。

第六，本地方法栈（Native Method Stack）。它和 Java 虚拟机栈是非常相似的，支持对本地方法的调用，也是每个线程都会创建一个。在 Oracle Hotspot JVM 中，本地方法栈和 Java 虚拟机栈是在同一块儿区域，这完全取决于技术实现的决定，并未在规范中强制。

2. 什么是 Java 虚拟机？为什么 Java 被称作是无关平台的编程语言？

Java 虚拟机是一个可以执行 Java 字节码的虚拟机进程。Java 源文件被编译成能被 Java 虚拟机执行的字节码文件。Java 被设计成允许应用程序可以运行在任意的平台，而不需要程序员为每一个平台单独重写或者是重新编译。Java 虚拟机让这个变为可能，因为它知道底层硬件平台的指令长度和其他特性。

3. 如何判断一个对象应该被回收？

1) 在 Java 中采取了 可达性分析法

通过一系列的“GC Roots”对象作为起点进行搜索，如果在“GC Roots”和一个对象之间没有可达路径，则称该对象是不可达的，不过要注意的是被判定为不可达的对象不一定会成为可回收对象。被判定为不可达的对象要成为可回收对象必须至少经历两次标记过程，如果在这两次标记过程中仍然没有逃脱成为可回收对象的可能性，则基本上就真的成为可回收对象了。

2) 虚拟机栈中引用的对象、方法区类静态属性引用的对象、方法区常量池引用的对象、本地方法栈 JNI 引用的对象

4. GC 触发的条件？

1) 程序调用 System.gc 时可以触发；（2）系统自身来决定 GC 触发的时机

5. 可以作为 GCRoots 的对象有哪些？

虚拟机栈中引用的对象

方法区中类静态属性引用的对象

方法区中常量引用的对象

本地方法栈中引用的对象

6. JVM 中一次完整的 GC 流程是怎样的，对象如何晋升到老年代？

Java 堆 = 老年代 + 新生代

新生代 = Eden + S0 + S1

当 Eden 区的空间满了，Java 虚拟机会触发一次 Minor GC，以收集新生代的垃圾，存活下来的对象，则会转移到 Survivor 区。

大对象（需要大量连续内存空间的 Java 对象，如那种很长的字符串）直接进入老年态；

如果对象在 Eden 出生，并经过第一次 Minor GC 后仍然存活，并且被 Survivor 容纳的话，年龄设为 1，每熬过一次 Minor GC，年龄+1，若年龄超过一定限制（15），则被晋升到老年态。即长期存活的对象进入老年态。

老年代满了而无法容纳更多的对象，Minor GC 之后通常就会进行 Full GC，Full GC 清理整个内存堆 – 包括年轻代和年老代。

Major GC 发生在老年代的 GC，清理老年区，经常会伴随至少一次 Minor GC，比 Minor GC 慢 10 倍以上。

7. 双亲委派模型？

双亲委派模型工作过程是：

如果一个类加载器收到类加载的请求，它首先不会自己去尝试加载这个类，而是把这个请求委派给父类加载器完成。每个类加载器都是如此，只有当父加载器在自己的搜索范围内找不到指定的类时（即 `ClassNotFoundException`），子加载器才会尝试自己去加载。

8. 为什么需要双亲委派模型？

防止内存中出现多份同样的字节码

9. 怎么打破双亲委派模型？

打破双亲委派机制则不仅要继承 `ClassLoader` 类，还要重写 `loadClass` 和 `findClass` 方法。

10. 导致 Full GC 一般有哪些情况？

1). 新生代设置过小

一是新生代 GC 次数非常频繁，增大系统消耗；二是导致大对象直接进入旧世代，占据了旧世代剩余空间，诱发 Full GC

2). 新生代设置过大

一是新生代设置过大会导致旧世代过小（堆总量一定），从而诱发 Full GC；二是新生代 GC 耗时大幅度增加

3). Survivor 设置过小

导致对象从 eden 直接到达旧世代

4). Survivor 设置过大

导致 eden 过小，增加了 GC 频率

一般说来新生代占整个堆 1/3 比较合适

GC 策略的设置方式

- 1). 吞吐量优先 可由-XX:GCTimeRatio=n 来设置
- 2). 暂停时间优先 可由-XX:MaxGCPauseRatio=n 来设置

11. Minor GC, Full GC 触发条件?

Minor GC 触发条件: 当 Eden 区满时, 触发 Minor GC。

Full GC 触发条件:

- (1) 调用 System.gc 时, 系统建议执行 Full GC, 但是不必然执行
- (2) 老年代空间不足
- (3) 方法区空间不足
- (4) 通过 Minor GC 后进入老年代的平均大小大于老年代的可用内存
- (5) 由 Eden 区、From Space 区向 To Space 区复制时, 对象大小大于 To Space 可存, 则把该对象转存到老年代, 且老年代的可用内存小于该对象大小

12. JVM 性能调优?

1、设定堆内存大小

-Xmx: 堆内存最大限制。

2、设定新生代大小。 新生代不宜太小, 否则会有大量对象涌入老年代

-XX:NewSize: 新生代大小

-XX:NewRatio 新生代和老年代占比

-XX:SurvivorRatio: 伊甸园空间和幸存者空间的占比

3、设定垃圾回收器 年轻代用 -XX:+UseParNewGC 老年代用-XX:+UseConcMarkSweepGC

13. Java 内存模型?

Java 内存模型定义了多线程之间共享变量的可见性以及如何在需要的时候对共享变量进行同步。JMM 内部的实现通常是依赖于所谓的内存屏障, 通过禁止某些重排序的方式, 提供内存可见性保证, 也就是实现了各种 happen-before 规则。

与 JVM 内存模型不同。

Java 内存模型即 Java Memory Model, 简称 JMM。JMM 定义了 Java 虚拟机(JVM)在计算机内存(RAM)中的工作方式。JVM 是整个计算机虚拟模型, 所以 JMM 是隶属于 JVM 的。

Java 内存模型定义了多线程之间共享变量的可见性以及如何在需要的时候对共享变量进行同步。

Java 线程之间的通信采用的是过共享内存模型, 这里提到的共享内存模型指的就是 Java 内存模型(简称 JMM), JMM 决定一个线程对共享变量的写入何时对另一个线程可见。从抽象的角度来看, JMM 定义了线程和主内存之间的抽象关系: 线程之间的共享变量存储在主内存(main memory)中, 每个线程都有一个私有的本地内存(local memory), 本地内存中存储了该线程以读/写共享变量的副本。

14. Java 中堆和栈有什么区别？

最主要的区别就是栈内存用来存储局部变量和方法调用。

而堆内存用来存储 Java 中的对象。无论是成员变量，局部变量，还是类变量，它们指向的对象都存储在堆内存中。

独有还是共享

栈内存归属于单个线程，每个线程都会有一个栈内存，其存储的变量只能在其所属线程中可见，即栈内存可以理解成线程的私有内存。

而堆内存中的对象对所有线程可见。堆内存中的对象可以被所有线程访问。

异常错误

如果栈内存没有可用的空间存储方法调用和局部变量，JVM 会抛出

`java.lang.StackOverflowError`。

而如果是堆内存没有可用的空间存储生成的对象，JVM 会抛出 `java.lang.OutOfMemoryError`。

空间大小

栈的内存要远远小于堆内存，如果你使用递归的话，那么你的栈很快就会充满。如果递归没有及时跳出，很可能发生 `StackOverflowError` 问题。

15. 常见的垃圾回收算法有哪些？简述其原理？

GC 最基础的算法有三种：标记-清除算法、复制算法、标记-压缩算法，我们常用的垃圾回收器一般都采用分代收集算法。

标记-清除算法，“标记-清除”（Mark-Sweep）算法，如它的名字一样，算法分为“标记”和“清除”两个阶段：首先标记出所有需要回收的对象，在标记完成后统一回收掉所有被标记的对象。

复制算法，“复制”（Copying）的收集算法，它将可用内存按容量划分为大小相等的两块，每次只使用其中的一块。当这一块的内存用完了，就将还存活着的对象复制到另外一块上面，然后再把已使用过的内存空间一次清理掉。

标记-压缩算法，标记过程仍然与“标记-清除”算法一样，但后续步骤不是直接对可回收对象进行清理，而是让所有存活的对象都向一端移动，然后直接清理掉端边界以外的内存

分代收集算法，“分代收集”（Generational Collection）算法，把 Java 堆分为新生代和老年代，这样就可以根据各个年代的特点采用最适当的收集算法。

16. 解释栈(stack)、堆(heap)和方法区(method area)的用法？

通常我们定义一个基本数据类型的变量，一个对象的引用，还有就是函数调用的现场保存都使用 JVM 中的栈空间；而通过 `new` 关键字和构造器创建的对象则放在堆空间，堆是垃圾收集器管理的主要区域，由于现在的垃圾收集器都采用分代收集算法，所以堆空间还可以细分为新生代和老生代，再具体一点可以分为 `Eden`、`Survivor`（又可分为 `From Survivor` 和 `To Survivor`）、`Tenured`；方法区和堆都是各个线程共享的内存区域，用于存储已经被 JVM 加载的类信息、常量、静态变量、JIT 编译器编译后的代码等数据；程序中的字面量（literal）如直接书写的 `100`、“`hello`”和常量都是放在常量池中，常量池是方法区的一部分，。栈空间操作起来最快但是栈很小，通常大量的对象都是放在堆空间，栈和堆的大小都可以通过 JVM 的启动参数来进行调整，栈空间用光了会引发 `StackOverflowError`，而堆和常量池空间不足则会引发 `OutOfMemoryError`。

17. 什么是类的加载？

类的加载指的是将类的.class 文件中的二进制数据读入到内存中，将其放在运行时数据区的方法区内，然后在堆区创建一个 `java.lang.Class` 对象，用来封装类在方法区内的数据结构。类的加载的最终产品是位于堆区中的 `Class` 对象，`Class` 对象封装了类在方法区内的数据结构，并且向 Java 程序员提供了访问方法区内的数据结构的接口。

18. 类加载器有哪些？

启动类加载器：Bootstrap ClassLoader，负责加载存放在 `JDK\jre\lib` (JDK 代表 JDK 的安装目录，下同) 下，或被 `-Xbootclasspath` 参数指定的路径中的，并且能被虚拟机识别的类库

扩展类加载器：Extension ClassLoader，该加载器由 `sun.misc.Launcher$ExtClassLoader` 实现，它负责加载 `JDK\jre\lib\ext` 目录中，或者由 `java.ext.dirs` 系统变量指定的路径中的所有类库（如 `javax.*` 开头的类），开发者可以直接使用扩展类加载器。

应用程序类加载器：Application ClassLoader，该类加载器由 `sun.misc.Launcher$AppClassLoader` 来实现，它负责加载用户类路径（`ClassPath`）所指定的类，开发者可以直接使用该类加载器

19. Java 对象创建过程？

1. JVM 遇到一条新建对象的指令时首先去检查这个指令的参数是否能在常量池中定义到一个类的符号引用。然后加载这个类（类加载过程在后边讲）
2. 为对象分配内存。一种办法“指针碰撞”、一种办法“空闲列表”，最终常用的办法“本地线程缓冲分配(TLAB)”
3. 将除对象头外的对象内存空间初始化为 0
4. 对对象头进行必要设置

20. Java 中类的生命周期使什么？

- 1、加载，查找并加载类的二进制数据，在 Java 堆中也创建一个 `java.lang.Class` 类的对象
- 2、连接，连接又包含三块内容：验证、准备、初始化。
 - 1) 验证，文件格式、元数据、字节码、符号引用验证；
 - 2) 准备，为类的静态变量分配内存，并将其初始化为默认值；
 - 3) 解析，把类中的符号引用转换为直接引用
- 3、初始化，为类的静态变量赋予正确的初始值
- 4、使用，`new` 出对象程序中使用
- 5、卸载，执行垃圾回收

21. 都有哪些垃圾回收器？

Serial 收集器，串行收集器是最古老，最稳定以及效率高的收集器，可能会产生较长的停顿，只使用一个线程去回收。

ParNew 收集器，ParNew 收集器其实就是 Serial 收集器的多线程版本。

Parallel 收集器，Parallel Scavenge 收集器类似 ParNew 收集器，Parallel 收集器更关注系统的吞吐量。

Parallel Old 收集器，Parallel Old 是 Parallel Scavenge 收集器的老年代版本，使用多线程和“标记—整理”算法

CMS 收集器，CMS（Concurrent Mark Sweep）收集器是一种以获取最短回收停顿时间为目标的收集器。

G1 收集器, G1 (Garbage-First)是一款面向服务器的垃圾收集器,主要针对配备多颗处理器及大容量内存的机器. 以极高概率满足 GC 停顿时间要求的同时,还具备高吞吐量性能特征

22. JVM 调优命令?

Sun JDK 监控和故障处理命令有 jps jstat jmap jhat jstack jinfo

- 1、jps, JVM Process Status Tool,显示指定系统内所有的 HotSpot 虚拟机进程。
- 2、jstat, JVM statistics Monitoring 是用于监视虚拟机运行时状态信息的命令, 它可以显示出虚拟机进程中的类装载、内存、垃圾收集、JIT 编译等运行数据。
- 3、jmap, JVM Memory Map 命令用于生成 heap dump 文件
- 4、jhat, JVM Heap Analysis Tool 命令是与 jmap 搭配使用, 用来分析 jmap 生成的 dump, jhat 内置了一个微型的 HTTP/HTML 服务器, 生成 dump 的分析结果后, 可以在浏览器中查看
- 5、jstack, 用于生成 java 虚拟机当前时刻的线程快照。
- 6、jinfo, JVM Configuration info 这个命令作用是实时查看和调整虚拟机运行参数。

23. JVM 调优工具?

常用调优工具分为两类,jdk 自带监控工具: jconsole 和 jvisualvm, 第三方有: MAT(Memory AnalyzerTool)、GChisto。

- 1、jconsole, Java Monitoring and Management Console 是从 java5 开始, 在 JDK 中自带的 java 监控和管理控制台, 用于对 JVM 中内存, 线程和类等的监控
- 2、jvisualvm, jdk 自带全能工具, 可以分析内存快照、线程快照; 监控内存变化、GC 变化等。
- 3、MAT, Memory Analyzer Tool, 一个基于 Eclipse 的内存分析工具, 是一个快速、功能丰富的 Java heap 分析工具, 它可以帮助我们查找内存泄漏和减少内存消耗
- 4、GChisto, 一款专业分析 gc 日志的工具

24. 描述一下 JVM 加载 class 文件的原理机制?

JVM 中类的装载是由类加载器 (ClassLoader) 和它的子类来实现的, Java 中的类加载器是一个重要的 Java 运行时系统组件, 它负责在运行时查找和装入类文件中的类。类的加载是指把类的.class 文件中的数据读入到内存中, 通常是创建一个字节数组读入.class 文件

25. GC 是什么? 为什么要有 GC?

GC 是垃圾收集的意思 (Gabbage Collection), 内存处理是编程人员容易出现问题的地方, 忘记或者错误的内存回收会

导致程序或系统的不稳定甚至崩溃, Java 提供的 GC 功能可以自动监测对象是否超过作用域从而达到自动回收内存的目的,

Java 语言没有提供释放已分配内存的显示操作方法。

26. 垃圾回收器的基本原理是什么?

对于 GC 来说, 当程序员创建对象时, GC 就开始监控这个对象的地址、大小以及使用情况。通常, GC 采用有向图的方式记录和管理堆(heap)中的所有对象。通过这种方式确定哪些对象

是"可达的", 哪些对象是"不可达的"。当 GC 确定一些对象为"不可达"时, GC 就有责任回收这些内存空间。可以。程序员可以手动执行 `System.gc()`, 通知 GC 运行, 但是 Java 语言规范并不保证 GC 一定会执行。

27. Java 中的引用类型有几种?

1、强引用

如果一个对象具有强引用, 它就不会被垃圾回收器回收。即使当前内存空间不足, JVM 也不会回收它, 而是抛出 `OutOfMemoryError` 错误, 使程序异常终止。如果想中断强引用和某个对象之间的关联, 可以显式地将引用赋值为 `null`, 这样一来的话, JVM 在合适的时间就会回收该对象。

2、软引用

在使用软引用时, 如果内存的空间足够, 软引用就能继续被使用, 而不会被垃圾回收器回收; 只有在内存空间不足时, 软引用才会被垃圾回收器回收

3、弱引用

具有弱引用的对象拥有的生命周期更短暂。因为当 JVM 进行垃圾回收, 一旦发现弱引用对象, 无论当前内存空间是否充足, 都会将弱引用回收。不过由于垃圾回收器是一个优先级较低的线程, 所以并不一定能迅速发现弱引用对象

4、虚引用

顾名思义, 就是形同虚设, 如果一个对象仅持有虚引用, 那么它相当于没有引用, 在任何时候都可能被垃圾回收器回收。

虚引用必须和引用队列关联使用, 当垃圾回收器准备回收一个对象时, 如果发现它还有虚引用, 就会把这个虚引用加入到与之关联的引用队列中。程序可以通过判断引用队列中是否已经加入了虚引用, 来了解被引用的对象是否将要被垃圾回收。如果程序发现某个虚引用已经被加入到引用队列, 那么就可以在所引用的对象的内存被回收之前采取必要的行动

五、多线程

1. Java 创建线程之后, 直接调用 `start()` 方法和 `run()` 的区别?

启动一个线程是调用 `start()` 方法, 使线程所代表的虚拟处理机处于可运行状态, 这意味着它可以由 JVM 调度并执行。这并不意味着线程就会立即运行。`run()` 方法可以产生必须退出的标志来停止一个线程。

2. 线程 B 怎么知道线程 A 修改了变量?

`volatile` 修饰变量

`synchronized` 修饰修改变量的方法

`wait/notify`

`while` 轮询

3. `synchronized` 和 `Volatile`、`CAS` 比较?

synchronized 是悲观锁，属于抢占式，会引起其他线程阻塞。

volatile 提供多线程共享变量可见性和禁止指令重排序优化。

CAS 是基于冲突检测的乐观锁（非阻塞）

4. 线程间通信，wait 和 notify 的理解和使用？

1 wait 和 notify 必须配合 synchronized 关键字使用。

2 wait 方法释放锁，notify 方法不释放锁。

3 还要注意一点 就是涉及到线程之间的通信，就肯定会用到 validate 修饰。

5. 定时线程的使用？

1、普通线程死循环

2、使用定时器 timer

3、使用定时调度线程池 ScheduledExecutorService

6. 线程同步的方法？

wait():使一个线程处于等待状态，并且释放所持有的对象的 lock。

sleep():使一个正在运行的线程处于睡眠状态，是一个静态方法，调用此方法要捕捉 InterruptedException 异常。

notify():唤醒一个处于等待状态的线程，注意的是在调用此方法的时候，并不能确切的唤醒某一个等待状态的线程，而是由 JVM 确定唤醒哪个线程，而且不是按优先级。

notifyAll():唤醒所有处于等待状态的线程，注意并不是给所有唤醒线程一个对象的锁，而是让它们竞争。

7. 进程和线程的区别？

1、调度：线程作为调度和分配的基本单位，进程作为拥有资源的基本单位。

2、并发性：不仅进程之间可以并发执行，同一个进程的多个线程之间也可以并发执行。

3、拥有资源：进程是拥有资源的一个独立单位，线程不拥有系统资源，但可以访问隶属于进程的资源。

4、系统开销：在创建或撤销进程的时候，由于系统都要为之分配和回收资源，导致系统的明显大于创建或撤销线程时的开销。但进程有独立的地址空间，进程崩溃后，在保护模式下不会对其他进程产生影响，而线程只是一个进程中的不同的执行路径。线程有自己的堆栈和局部变量，但线程之间没有单独的地址空间，一个线程死掉就等于整个进程死掉，所以多线程的程序要比多进程的程序健壮，但是在进程切换时，耗费资源较大，效率要差些。

8. 什么叫线程安全？

如果你的代码所在的进程中有多线程在同时运行，而这些线程可能会同时运行这段代码。

如果每次运行结果和单线程运行的结果是一样的，而且其他的变量的值也和预期的是一样的，就是线程安全的。一个线程安全的计数器类的同一个实例对象在被多个线程使用的情况下也不会出现计算失误。很显然你可以将集合类分成两组，线程安全和非线程安全的。

9. 线程的几种状态?

- 1、新建状态(New): 新创建了一个线程对象。
- 2、就绪状态(Runnable): 线程对象创建后, 其他线程调用了该对象的 `start()`方法。该状态的线程位于“可运行线程池”中, 变得可运行, 只等待获取 CPU 的使用权。即在就绪状态的进程除 CPU 之外, 其它的运行所需资源都已全部获得。
- 3、运行状态(Running): 就绪状态的线程获取了 CPU, 执行程序代码。
- 4、阻塞状态(Blocked): 阻塞状态是线程因为某种原因放弃 CPU 使用权, 暂时停止运行。直到线程进入就绪状态, 才有机会转到运行状态。

阻塞的情况分三种:

- (1)、等待阻塞: 运行的线程执行 `wait()`方法, 该线程会释放占用的所有资源, JVM 会把该线程放入“等待池”中。进入这个状态后, 是不能自动唤醒的, 必须依靠其他线程调用 `notify()`或 `notifyAll()`方法才能被唤醒,
 - (2)、同步阻塞: 运行的线程在获取对象的同步锁时, 若该同步锁被别的线程占用, 则 JVM 会把该线程放入“锁池”中。
 - (3)、其他阻塞: 运行的线程执行 `sleep()`或 `join()`方法, 或者发出了 I/O 请求时, JVM 会把该线程置为阻塞状态。当 `sleep()`状态超时、`join()`等待线程终止或者超时、或者 I/O 处理完毕时, 线程重新转入就绪状态。
- 5、死亡状态(Dead): 线程执行完了或者因异常退出了 `run()`方法, 该线程结束生命周期。

10. volatile 变量和 atomic 变量有什么不同?

volatile 变量和 atomic 变量看起来很像, 但功能却不一样。Volatile 变量可以确保先行关系, 即写操作会发生在后续的读操作之前, 但它并不能保证原子性。例如用 volatile 修饰 count 变量那么 `count++` 操作就不是原子性的。而 AtomicInteger 类提供的 atomic 方法可以让这种操作具有原子性如 `getAndIncrement()`方法会原子性 的进行增量操作把当前值加一, 其它数据类型和引用变量也可以进行相似操作。

11. Java 中什么是静态条件?

竞态条件会导致程序在并发情况下出现一些 bugs。多线程对一些资源的竞争的时候就会产生竞态条件, 如果首先要执行的程序竞争失败排到后面执行了, 那么整个程序就会出现一些不确定的 bugs。这种 bugs 很难发现而且会重复出现, 因为线程间的随机竞争。

12. Java 中如何停止一个线程?

Java 提供了很丰富的 API 但没有为停止线程提供 API。JDK 1.0 本来有一些像 `stop()`, `suspend()`和 `resume()`的控制方法但是由于潜在的死锁威胁因此在后续的 JDK 版本中他们被弃用了, 之后 Java API 的设计者就没有提供一个兼容且线程安全的方法来停止一个线程。当 `run()` 或者 `call()` 方法执行完的时候线程会自动结束, 如果要手动结束一个线程, 你可以用 volatile 布尔变量来退出 `run()`方法的循环或者是取消任务来中断线程。

13. 线程池的优点?

- 1) 重用存在的线程, 减少对象创建销毁的开销。

- 2) 可有效的控制最大并发线程数，提高系统资源的使用率，同时避免过多资源竞争，避免堵塞。
- 3) 提供定时执行、定期执行、单线程、并发数控制等功能。

14. volatile 的理解?

volatile 关键字的两层语义

一旦一个共享变量（类的成员变量、类的静态成员变量）被 volatile 修饰之后，那么就具备了两层语义：

- 1) 保证了不同线程对这个变量进行操作时的可见性，即一个线程修改了某个变量的值，这新值对其他线程来说是立即可见的。
- 2) 禁止进行指令重排序。

用 volatile 修饰之后，变量的操作：

第一：使用 volatile 关键字会强制将修改的值立即写入主存；

第二：使用 volatile 关键字的话，当线程 2 进行修改时，会导致线程 1 的工作内存中缓存变量 stop 的缓存行无效（反映到硬件层的话，就是 CPU 的 L1 或者 L2 缓存中对应的缓存行无效）；

第三：由于线程 1 的工作内存中缓存变量 stop 的缓存行无效，所以线程 1 再次读取变量 stop 的值时会去主存读取。

15. 实现多线程有几种方式?

在语言层面有两种方式。java.lang.Thread 类的实例就是一个线程但是它需要调用 java.lang.Runnable 接口来执行，由于线程类本身就是调用的 Runnable 接口所以你可以继承 java.lang.Thread 类或者直接调用 Runnable 接口来重写 run()方法实现线程。

16. Java 中 notify 和 notifyAll 有什么区别?

notify()方法不能唤醒某个具体的线程，所以只有一个线程在等待的时候它才有用武之地。而 notifyAll()唤醒所有线程并允许他们争夺锁确保了至少有一个线程能继续运行。

17. 什么是乐观锁和悲观锁?

1) 乐观锁：就像它的名字一样，对于并发间操作产生的线程安全问题持乐观状态，乐观锁认为竞争不总是会发生，因此它不需要持有锁，将比较-替换这两个动作作为一个原子操作尝试去修改内存中的变量，如果失败则表示发生冲突，那么就应该有相应的重试逻辑。

2) 悲观锁：还是像它的名字一样，对于并发间操作产生的线程安全问题持悲观状态，悲观锁认为竞争总是会发生，因此每次对某资源进行操作时，都会持有一个独占的锁，就像 synchronized，不管三七二十一，直接上了锁就操作资源了。

18. 线程的创建方式?

方式一：继承 Thread 类

方式二：实现 Runnable 接口

方式三：实现 `Callable` 接口

方式四：使用线程池的方式

19. 线程池的作用？

创建线程要花费昂贵的资源和时间，如果任务来了才创建线程那么响应时间会变长，而且一个进程能创建的线程数有限。为了避免这些问题，在程序启动的时候就创建若干线程来响应处理，它们被称为线程池，里面的线程叫工作线程。从 `JDK1.5` 开始，`Java API` 提供了 `Executor` 框架让你可以创建不同的线程池。比如单线程池，每次处理一个任务；数目固定的线程池或者是缓存线程池（一个适合很多生存期短的任务的程序的可扩展线程池）。

20. `wait` 和 `sleep` 的区别？

`sleep` 是线程类（`Thread`）的方法，导致此线程暂停执行指定时间，给执行机会给其他线程，但是监控状态依然保持，到时后会自动恢复。调用 `sleep` 不会释放对象锁。

`wait` 是 `Object` 类的方法，对此对象调用 `wait` 方法导致本线程放弃对象锁，进入等待此对象的等待锁定池，只有针对此对象发出 `notify` 方法（或 `notifyAll`）后本线程才进入对象锁定池准备获得对象锁进入运行状态。

21. 产生死锁的条件？

- 1、互斥条件：一个资源每次只能被一个进程使用。
- 2、请求与保持条件：一个进程因请求资源而阻塞时，对已获得的资源保持不放。
- 3、不剥夺条件：进程已获得的资源，在未使用完之前，不能强行剥夺。
- 4、循环等待条件：若干进程之间形成一种头尾相接的循环等待资源关系。

22. 请写出实现线程安全的几种方式？

方式一：使用同步代码块

方式二：使用同步方法

方式三：使用 `ReentrantLock`

23. 守护线程是什么？它和非守护线程的区别？

程序运行完毕，`jvm` 会等待非守护线程完成后关闭，但是 `jvm` 不会等待守护线程。守护线程最典型的例子就是 `GC` 线程。

24. 什么是多线程的上下文切换？

多线程的上下文切换是指 `CPU` 控制权由一个已经正在运行的线程切换到另外一个就绪并等待获取 `CPU` 执行权的线程的过程。

25. `Callable` 和 `Runnable` 的区别是什么？

两者都能用来编写多线程,但实现 `Callable` 接口的任务线程能返回执行结果,而实现 `Runnable` 接口的任务线程不能返回结果.`Callable` 通常需要和`Future/FutureTask` 结合使用,用于获取异步计算结果.

26. 线程阻塞有哪些原因?

1、`sleep()` 允许 指定以毫秒为单位的一段时间作为参数,它使得线程在指定的时间内进入阻塞状态,不能得到 CPU 时间,指定的时间一过,线程重新进入可执行状态。典型地, `sleep()`被用在等待某个资源就绪的情形:测试发现条件不满足后,让线程阻塞一段时间后重新测试,直到条件满足为止

2、`suspend()` 和 `resume()` 两个方法配套使用, `suspend()`使得线程进入阻塞状态,并且不会自动恢复,必须其对应的 `resume()` 被调用,才能使得线程重新进入可执行状态。典型地, `suspend()` 和 `resume()` 被用在等待另一个线程产生的结果的情形:测试发现结果还没有产生后,让线程阻塞,另一个线程产生了结果后,调用 `resume()` 使其恢复。

3、`yield()` 使当前线程放弃当前已经分得的 CPU 时间,但不使当前线程阻塞,即线程仍处于可执行状态,随时可能再次分得 CPU 时间。调用 `yield()` 的效果等价于调度程序认为该线程已执行了足够的时间从而转到另一个线程

4、`wait()` 和 `notify()` 两个方法配套使用, `wait()` 使得线程进入阻塞状态,它有两种形式,一种允许 指定以毫秒为单位的一段时间作为参数,另一种没有参数,前者当对应的 `notify()` 被调用或者超出指定时间时线程重新进入可执行状态,后者则必须对应的 `notify()` 被调用。

27. `synchronized` 和 `Lock` 的区别?

主要相同点: `Lock` 能完成 `synchronized` 所实现的所有功能

主要不同点: `Lock` 有比 `synchronized` 更精确的线程语义和更好的性能。`synchronized` 会自动释放锁,而 `Lock` 一定要求程序员手工释放,并且必须在 `finally` 从句中释放。

28. `ThreadLocal` 是什么? 有什么作用?

`ThreadLocal` 是一个本地线程副本变量工具类。主要用于将私有线程和该线程存放的副本对象做一个映射,各个线程之间的变量互不干扰,在高并发场景下,可以实现无状态的调用,特别适用于各个线程依赖不通的变量值完成操作的场景。

简单说 `ThreadLocal` 就是一种以空间换时间的做法,在每个 `Thread` 里面维护了一个以开地址法实现的 `ThreadLocal.ThreadLocalMap`,把数据进行隔离,数据不共享,自然就没有线程安全方面的问题了。

29. 交互方式分为同步和异步两种?

同步交互:指发送一个请求,需要等待返回,然后才能够发送下一个请求,有个等待过程;

异步交互:指发送一个请求,不需要等待返回,随时可以再发送下一个请求,即不需要等待。

区别:一个需要等待,一个不需要等待,在部分情况下,我们的项目开发中都会优先选择不需要等待的异步交互方式。

30. 什么是线程?

线程是操作系统能够进行运算调度的最小单位，它被包含在进程之中，是进程中的实际运作单位。程序员可以通过它进行多处理器编程，你可以使用多线程对运算密集型任务提速

31. 什么是 FutureTask?

在 Java 并发程序中 FutureTask 表示一个可以取消的异步运算。它有启动和取消运算、查询运算是否完成和取回运算结果等方法。只有当运算完成的时候结果才能取回，如果运算尚未完成 get 方法将会阻塞。一个 FutureTask 对象可以对调用了 Callable 和 Runnable 的对象进行包装，由于 FutureTask 也是调用了 Runnable 接口所以它可以提交给 Executor 来执行。

32. Java 中 interrupted 和 isInterruptedd 方法的区别?

interrupted() 和 isInterrupted() 的主要区别是前者会将中断状态清除而后者不会。Java 多线程的中断机制是用内部标识来实现的，调用 Thread.interrupt() 来中断一个线程就会设置中断标识为 true。当中断线程调用静态方法 Thread.interrupted() 来检查中断状态时，中断状态会被清零。而非静态方法 isInterrupted() 用来查询其它线程的中断状态且不会改变中断状态标识。简单的说就是任何抛出 InterruptedException 异常的方法都会将中断状态清零。无论如何，一个线程的中断状态有可能被其它线程调用中断来改变。

33. 死锁的原因?

1) 是多个线程涉及到多个锁，这些锁存在着交叉，所以可能会导致了一个锁依赖的闭环。例如：线程在获得了锁 A 并且没有释放的情况下去申请锁 B，这时，另一个线程已经获得了锁 B，在释放锁 B 之前又要先获得锁 A，因此闭环发生，陷入死锁循环。

2) 默认的锁申请操作是阻塞的。

所以要避免死锁，就要在一遇到多个对象锁交叉的情况，就要仔细审查这几个对象的类中的所有方法，是否存在导致锁依赖的环路的可能性。总之是尽量避免在一个同步方法中调用其它对象的延时方法和同步方法。

34. 什么是自旋

很多 synchronized 里面的代码只是一些很简单的代码，执行时间非常快，此时等待的线程都加锁可能是一种不太值得的操作，因为线程阻塞涉及到用户态和内核态切换的问题。既然 synchronized 里面的代码执行得非常快，不妨让等待锁的线程不要被阻塞，而是在 synchronized 的边界做忙循环，这就是自旋。如果做了多次忙循环发现还没有获得锁，再阻塞，这样可能是一种更好的策略。

35. 怎么唤醒一个阻塞的线程?

如果线程是因为调用了 wait()、sleep() 或者 join() 方法而导致的阻塞，可以中断线程，并且通过抛出 InterruptedException 来唤醒它；如果线程遇到了 IO 阻塞，无能为力，因为 IO 是操作系统实现的，Java 代码并没有办法直接接触到操作系统。

36. 如果提交任务时，线程池队列已满，这时会发生什么?

许多程序员会认为该任务会阻塞直到线程池队列有空位。事实上如果一个任务不能被调度执行那么 `ThreadPoolExecutor's submit()` 方法将会抛出一个 `RejectedExecutionException` 异常。

37. 什么是线程局部变量?

线程局部变量是局限于线程内部的变量，属于线程自身所有，不在多个线程间共享。Java 提供 `ThreadLocal` 类来支持线程局部变量，是一种实现线程安全的方式。但是在管理环境下（如 web 服务器）使用线程局部变量的时候要特别小心，在这种情况下，工作线程的生命周期比任何应用变量的生命周期都要长。任何线程局部变量一旦在工作完成后没有释放，Java 应用就存在内存泄露的风险。

38. 使用 `volatile` 关键字的场景?

`synchronized` 关键字是防止多个线程同时执行一段代码，那么就会很影响程序执行效率，而 `volatile` 关键字在某些情况下性能要优于 `synchronized`，但是要注意 `volatile` 关键字是无法替代 `synchronized` 关键字的，因为 `volatile` 关键字无法保证操作的原子性。通常来说，使用 `volatile` 必须具备以下 2 个条件：

- 1) 对变量的写操作不依赖于当前值
- 2) 该变量没有包含在具有其他变量的不变式中

39. 线程池的工作原理，几个重要参数?

`ThreadPoolExecutor(int corePoolSize, int maximumPoolSize, long keepAliveTime, TimeUnit unit, BlockingQueue workQueue, ThreadFactory threadFactory, RejectedExecutionHandler handler)`
参数说明：

`corePoolSize` 核心线程数

`maximumPoolSize` 最大线程数，一般大于等于核心线程数

`keepAliveTime` 线程存活时间(针对最大线程数大于核心线程数时，非核心线程)

`unit` 存活时间单位，和线程存活时间配套使用

`workQueue` 任务队列

`threadFactory` 创建线程的工程

`handler` 拒绝策略

40. 线程池的类型?

五种线程池：

```
ExecutorService threadPool = null;
threadPool = Executors.newCachedThreadPool();//有缓冲的线程池，线程数 JVM 控制
threadPool = Executors.newFixedThreadPool(3);//固定大小的线程池
threadPool = Executors.newScheduledThreadPool(2);
threadPool = Executors.newSingleThreadExecutor();//单线程的线程池，只有一个线程在工作
threadPool = new ThreadPoolExecutor();//默认线程池，可控制参数比较多
```

41. 线程池的阻塞队列有哪些？

三种阻塞队列：

```
BlockingQueue<Runnable> workQueue = null;  
workQueue = new ArrayBlockingQueue<>(5); //基于数组的先进先出队列，有界  
workQueue = new LinkedBlockingQueue<>(); //基于链表的先进先出队列，无界  
workQueue = new SynchronousQueue<>(); //无缓冲的等待队列，无界
```

42. 线程池的拒绝策略都有哪些？

四种拒绝策略

等待队列已经排满了，再也塞不下新任务，同时线程池中线程也已经达到 `maximumPoolSize` 数量，无法继续为新任务服务，这个时候就需要使用拒绝策略来处理。

```
RejectedExecutionHandler rejected = null;  
rejected = new ThreadPoolExecutor.AbortPolicy(); //默认，队列满了丢任务抛出异常,直接  
抛出 RejectedExecutionException 异常阻止系统正常运行。
```

```
rejected = new ThreadPoolExecutor.DiscardPolicy(); //队列满了丢任务不异常,直接丢弃任  
务，不予任何处理也不抛出异常。如果允许任务丢失，这是最好的一种方案。
```

```
rejected = new ThreadPoolExecutor.DiscardOldestPolicy(); //将最早进入队列的任务删，之  
后再尝试加入队列，抛弃队列中等待最久的任务，然后把当前任务加入队列中尝试再次提交  
当前任务。
```

```
rejected = new ThreadPoolExecutor.CallerRunsPolicy(); //如果添加到线程池失败，那么主线  
程会自己去执行该任务,调用者运行”一种调节机制，该策略既不会丢弃任务，也不会抛出异常，  
而是将某些任务回退给调用者，从而降低新任务的流量。
```

六、设计模式

1. 说一下你熟悉的设计模式？

单例模式：保证被创建一次，节省系统开销。

工厂模式（简单工厂、抽象工厂）：解耦代码。

观察者模式：定义了对象之间的一对多的依赖，这样一来，当一个对象改变时，它的所有的 依赖者都会收到通知并自动更新。

外观模式：提供一个统一的接口，用来访问子系统中的一群接口，外观定义了一个高层的接 口，让子系统更容易使用。

模版方法模式：定义了一个算法的骨架，而将一些步骤延迟到子类中，模版方法使得子类可 以在不改变算法结构的情况下，重新定义算法的步骤。

状态模式：允许对象在内部状态改变时改变它的行为，对象看起来好像修改了它的类。

2. 简单工厂和抽象工厂的区别？

简单工厂：用来生产同一等级结构中的任意产品，对于增加新的产品，无能为力。 工厂方法：用来生产同一等级结构中的固定产品，支持增加任意产品。
抽象工厂：用来生产不同产品族的全部产品，对于增加新的产品，无能为力；支持增加产品族。

3. 设计模式的优点？

设计模式可在多个项目中重用。

设计模式提供了一个帮助定义系统架构的解决方案。

设计模式吸收了软件工程的经验。

设计模式为应用程序的设计提供了透明性。

设计模式是被实践证明切实有效的，由于它们是建立在专家软件开发人员的知识和经验之上的。

4. 设计模式的六大基本原则？

(1) 单一原则(Single Responsibility Principle)：一个类只负责一项职责，尽量做到类的只有一个行为原因引起变化；

(2) 里氏替换原则（LSP liskov substitution principle）：子类可以扩展父类的功能，但不能改变原有父类的功能；

(3) 依赖倒置原则（dependence inversion principle）：面向接口编程；

(4) 接口隔离（interface segregation principle）：建立单一接口；

(5) 迪米特原则（law of demeter LOD）：最少知道原则，尽量降低类与类之间的耦合；

(6) 开闭原则（open closed principle）：用抽象构建架构，用实现扩展原则；

5. 单例模式？

单例就是该类只能返回一个实例。

单例所具备的特点：

1. 私有化的构造函数
2. 私有的静态的全局变量
3. 公有的静态的方法

单例分为懒汉式、饿汉式和双层锁式

饿汉式：

```
public class Singleton1 {
    private Singleton1() {}
    private static Singleton1 single = new Singleton1();
    public static Singleton1 getInstance() {
        return single;
    }
}
```

懒汉式：

```
public class Singleton2
{ private Singleton2()
{}
}
```

```

        private static Singleton2 single=null;
        public static Singleton2 getInstance() {
            if (single == null) {
                single = new Singleton2();
            }
            return single;
        }
    }
}
线程安全：
public class Singleton3
{ private Singleton3()
{}
private static Singleton3 single ;
public static Singleton3 getInstance() {
    if(null ==
        single){ synchronized(
            single ){
                if(null == single){
                    single = new Singleton3();
                }
            }
        }
    return single;
}
}

```

6. 设计模式的分类？

1. 根据目的来分

根据模式是用来完成什么工作来划分，这种方式可分为创建型模式、结构型模式和行为型模式 3 种。

创建型模式：用于描述“怎样创建对象”，它的主要特点是“将对象的创建与使用分离”。GoF 中提供了单例、原型、工厂方法、抽象工厂、建造者等 5 种创建型模式。

结构型模式：用于描述如何将类或对象按某种布局组成更大的结构，GoF 中提供了代理、适配器、桥接、装饰、外观、享元、组合等 7 种结构型模式。

行为型模式：用于描述类或对象之间怎样相互协作共同完成单个对象都无法单独完成的任务，以及怎样分配职责。GoF 中提供了模板方法、策略、命令、职责链、状态、观察者、中介者、迭代器、访问者、备忘录、解释器等 11 种行为型模式。

2. 根据作用范围来分

根据模式是主要用于类上还是主要用于对象上来分，这种方式可分为类模式和对象模式两种。

类模式：用于处理类与子类之间的关系，这些关系通过继承来建立，是静态的，在编译时刻便确定下来了。GoF 中的工厂方法、（类）适配器、模板方法、解释器属于该模式。

对象模式：用于处理对象之间的关系，这些关系可以通过组合或聚合来实现，在运行时是可以变化的，更具动态性。GoF 中除了以上 4 种，其他的都是对象模式。

7.23 种设计模式的具体的每种模式的功能？

单例（Singleton）模式：某个类只能生成一个实例，该类提供了一个全局访问点供外部获取 该实例，其拓展是有限多例模式。

原型（Prototype）模式：将一个对象作为原型，通过对其进行复制而克隆出多个和原型类似的新实例。

工厂方法（Factory Method）模式：定义一个用于创建产品的接口，由子类决定生产什么产品。

抽象工厂（AbstractFactory）模式：提供一个创建产品族的接口，其每个子类可以生产一系列相关的产品。

建造者（Builder）模式：将一个复杂对象分解成多个相对简单的部分，然后根据不同需要分别创建它们，最后构建成该复杂对象。

代理（Proxy）模式：为某对象提供一种代理以控制对该对象的访问。即客户端通过代理间接地访问该对象，从而限制、增强或修改该对象的一些特性。

适配器（Adapter）模式：将一个类的接口转换成客户希望的另外一个接口，使得原本由于接口不兼容而不能一起工作的那些类能一起工作。

桥接（Bridge）模式：将抽象与实现分离，使它们可以独立变化。它是用组合关系代替 继承关系来实现，从而降低了抽象和实现这两个可变维度的耦合度。

装饰（Decorator）模式：动态的给对象增加一些职责，即增加其额外的功能。

外观（Facade）模式：为多个复杂的子系统提供一个一致的接口，使这些子系统更加容易被访问。

享元（Flyweight）模式：运用共享技术来有效地支持大量细粒度对象的复用。

组合（Composite）模式：将对象组合成树状层次结构，使用户对单个对象和组合对象具有一致的访问性。

模板方法（TemplateMethod）模式：定义一个操作中的算法骨架，而将算法的一些步骤延迟到子类中，使得子类可以不改变该算法结构的情况下重定义该算法的某些特定步骤。

策略（Strategy）模式：定义了一系列算法，并将每个算法封装起来，使它们可以相互替换，且算法的改变不会影响使用算法的客户。

命令（Command）模式：将一个请求封装为一个对象，使发出请求的责任和执行请求的责任分割开。

职责链（Chain of Responsibility）模式：把请求从链中的一个对象传到下一个对象，直到请求被响应为止。通过这种方式去除对象之间的耦合。

状态（State）模式：允许一个对象在其内部状态发生改变时改变其行为能力。

观察者（Observer）模式：多个对象间存在一对多关系，当一个对象发生改变时，把这种改变通知给其他多个对象，从而影响其他对象的行为。

中介者（Mediator）模式：定义一个中介对象来简化原有对象之间的交互关系，降低系统中对象间的耦合度，使原有对象之间不必相互了解。

迭代器（Iterator）模式：提供一种方法来顺序访问聚合对象中的一系列数据，而不暴露聚合对象的内部表示。

访问者（Visitor）模式：在不改变集合元素的前提下，为一个集合中的每个元素提供一种访问方式，即每个元素有多个访问者对象访问。

备忘录（Memento）模式：在不破坏封装性的前提下，获取并保存一个对象的内部状态，以便以后恢复它。

解释器（Interpreter）模式：提供如何定义语言的文法，以及对语言句子的解释方法，即解释器。

7. UML 是什么？

统一建模语言（Unified Modeling Language，UML）是用来设计软件蓝图的可视化建模语言，1997 年被国际对象管理组织（OMG）采纳为面向对象的建模语言的国际标准。它的特点是简单、统一、图形化、能表达软件设计中的动态与静态信息。

统一建模语言能为软件开发的所有阶段提供模型化和可视化支持。而且融入了软件工程领域的新思想、新方法和新技术，使软件设计人员沟通更简明，进一步缩短了设计时间，减少开发成本。它的应用领域很宽，不仅适合于一般系统的开发，而且适合于并行与分布式系统的建模。

UML 从目标系统的不同角度出发，定义了用例图、类图、对象图、状态图、活动图、时序图、协作图、构件图、部署图等 9 种图。

8. 桥接模式是什么？

桥接（Bridge）模式的定义如下：将抽象与实现分离，使它们可以独立变化。它是用组合关系代替继承关系来实现，从而降低了抽象和实现这两个可变维度的耦合度。

桥接（Bridge）模式的优点是：

- 由于抽象与实现分离，所以扩展能力强；

- 其实现细节对客户透明。

缺点是：由于聚合关系建立在抽象层，要求开发者针对抽象化进行设计与编程，这增加了系统的理解与设计难度

类适配器：

```
package adapter;
//目标接口
interface Target
{
    public void request();
}
//适配器接口
class Adaptee
{
    public void specificRequest()
    {
        System.out.println("适配器中的业务代码被调用！");
    }
}
//类适配器类
class ClassAdapter extends Adaptee implements Target
{
    public void request()
    {
        specificRequest();
    }
}
//客户端代码
public class ClassAdapterTest
{
    public static void main(String[] args)
    {
        System.out.println("类适配器模式测试：");
        Target target = new ClassAdapter();
        target.request();
    }
}
```

对象适配器：

```

package adapter;
//对象适配器类
class ObjectAdapter implements Target
{
    private Adaptee adaptee;
    public ObjectAdapter(Adaptee adaptee)
    {
        this.adaptee=adaptee;
    }
    public void request()
    {
        adaptee.specificRequest();
    }
}
//客户端代码
public class ObjectAdapterTest
{
    public static void main(String[] args)
    {
        System.out.println("对象适配器模式测试: ");
        Adaptee adaptee = new Adaptee();
        Target target = new ObjectAdapter(adaptee);
        target.request();
    }
}

```

9. 策略模式是什么？

策略（Strategy）模式的定义：该模式定义了一系列算法，并将每个算法封装起来，使它们可以相互替换，且算法的变化不会影响使用算法的客户。策略模式属于对象行为模式，它通过对算法进行封装，把使用算法的责任和算法的实现分割开来，并委派给不同的对象对这些算法进行管理。

策略模式的主要优点如下

多重条件语句不易维护，而使用策略模式可以避免使用多重条件语句。

策略模式提供了一系列的可供重用的算法族，恰当使用继承可以把算法族的公共代码转移到父类里面，从而避免重复的代码。

策略模式可以提供相同行为的不同实现，客户可以根据不同时间或空间要求选择不同的。

策略模式提供了对开闭原则的完美支持，可以在不修改原代码的情况下，灵活增加新算法。

策略模式把算法的使用放到环境类中，而算法的实现移到具体策略类中，实现了二者的分离。

其主要缺点如下

客户端必须理解所有策略算法的区别，以便适时选择恰当的算法类。

策略模式造成很多的策略类

```

package strategy;
public class StrategyPattern
{
    public static void main(String[] args)
    {
        Context c=new Context();
        Strategy s=new ConcreteStrategyA();
        c.setStrategy(s);
        c.strategyMethod();
        s=new ConcreteStrategyB();
        c.setStrategy(s);
        c.strategyMethod();
    }
}
//抽象策略类
interface Strategy
{
    public void strategyMethod();    //策略方法
}
//具体策略类A
class ConcreteStrategyA implements Strategy
{
    public void strategyMethod()
    {
        System.out.println("具体策略A的策略方法被访问!");
    }
}
//具体策略类B
class ConcreteStrategyB implements Strategy
{
    public void strategyMethod()
    {
        System.out.println("具体策略B的策略方法被访问!");
    }
}
//环境类
class Context
{
    private Strategy strategy;
    public Strategy getStrategy()
    {
        return strategy;
    }
    public void setStrategy(Strategy strategy)
    {
        this.strategy=strategy;
    }
    public void strategyMethod()
    {
        strategy.strategyMethod();
    }
}

```

七、开源框架

1. hibernate 和 mybatis 的区别?

相同点:

- 1) 都属于 ORM 框架
- 2) 都是对 jdbc 的包装
- 3) 都属于持久层的框架

不同点:

- 1) hibernate 是面向对象的, mybatis 是面向 sql 的;
- 2) hibernate 全自动的 orm, mybatis 是半自动的 orm;

- 3)hibernate 查询映射实体对象必须全字段查询，mybatis 可以不用；
- 4) hibernate 级联操作，mybatis 则没有；
- 5) hibernate 编写 hql 查询数据库大大降低了对象和数据库的耦合性，mybatis 提供动态 sql，需要手写 sql，与数据库之间的耦合度取决于程序员所写的 sql 的方法，所以 hibernate 的移植性要远大于 mybatis。
- 6) hibernate 有方言夸数据库，mybatis 依赖于具体的数据库。
- 7) hibernate 拥有完整的日志系统，mybatis 则相对比较欠缺。

2. MyBatis 的优点？

- 1、基于 SQL 语句编程，相当灵活，不会对应用程序或者数据库的现有设计造成任何影响，SQL 写在 XML 里，解除 sql 与程序代码的耦合，便于统一管理；提供 XML 标签，支持编写动态 SQL 语句，并可重用。
- 2、与 JDBC 相比，减少了 50%以上的代码量，消除了 JDBC 大量冗余的代码，不需要手动开关连接；
- 3、很好的与各种数据库兼容（因为 MyBatis 使用 JDBC 来连接数据库，所以只要 JDBC 支持的数据库 MyBatis 都支持）。
- 4、能够与 Spring 很好的集成；
- 5、提供映射标签，支持对象与数据库的 ORM 字段关系映射；提供对象关系映射标签，支持对象关系组件维护。

3. MyBatis 框架的缺点？

- （1）SQL 语句的编写工作量较大，尤其当字段多、关联表多时，对开发人员编写 SQL 语句的功底有一定要求。
- （2）SQL 语句依赖于数据库，导致数据库移植性差，不能随意更换数据库。

4. SpringMVC 工作流程？

- 1、用户发送请求至前端控制器 DispatcherServlet
- 2、DispatcherServlet 收到请求调用 HandlerMapping 处理器映射器。
- 3、处理器映射器根据请求 url 找到具体的处理器，生成处理器对象及处理器拦截器(如果有则生成)一并返回给 DispatcherServlet。
- 4、DispatcherServlet 通过 HandlerAdapter 处理器适配器调用处理器
- 5、执行处理器(Controller，也叫后端控制器)。
- 6、Controller 执行完成返回 ModelAndView
- 7、HandlerAdapter 将 controller 执行结果 ModelAndView 返回给 DispatcherServlet
- 8、DispatcherServlet 将 ModelAndView 传给 ViewResolver 视图解析器
- 9、ViewResolver 解析后返回具体 View
- 10、DispatcherServlet 对 View 进行渲染视图（即将模型数据填充至视图中）。
- 11、DispatcherServlet 响应用户

5. MyBatis 框架使用的场合？

- （1）MyBatis 专注于 SQL 本身，是一个足够灵活的 DAO 层解决方案。

(2) 对性能的要求很高, 或者需求变化较多的项目, 如互联网项目, **MyBatis** 将是不错的选择。

6. Spring 中 **beanFactory** 和 **ApplicationContext** 的联系和区别?

BeanFactory 是 **spring** 中较为原始的 **Factory**, 无法支持 **spring** 的许多插件, 如 **AOP** 功能、**Web** 应用等。

ApplicationContext 接口是通过 **BeanFactory** 接口派生而来的, 除了具备 **BeanFactory** 接口的功能外, 还具备资源访问、事件传播、国际化消息访问等功能。

总体区别如下:

- 1) 使用 **ApplicationContext**, 配置 **bean** 默认配置是 **singleton**, 无论是否使用, 都会被实例化。优点是预先加载, 缺点是浪费内存;
- 2) 使用 **BeanFactory** 实例化对象时, 配置的 **bean** 等到使用的时候才会被实例化。优点是节约内存, 缺点是速度比较慢, 多用于移动设备的开发;
- 3) 没有特殊要求的情况下, 应该使用 **ApplicationContext** 完成, **ApplicationContext** 可以实现 **BeanFactory** 所有可实现的功能, 还具备其他更多的功能。

7. SpringIOC 注入的几种方式?

构造器注入

set 方法注入

接口注入

8. 拦截器与过滤器的区别?

- 1、拦截器是基于 **java** 的反射机制的, 而过滤器是基于函数回调
- 2、拦截器不依赖与 **servlet** 容器, 过滤器依赖与 **servlet** 容器。
- 3、拦截器只能对 **action** 请求起作用, 而过滤器则可以对几乎所有的请求起作用。
- 4、拦截器可以访问 **action** 上下文、值栈里的对象, 而过滤器不能访问。
- 5、在 **action** 的生命周期中, 拦截器可以多次被调用, 而过滤器只能在容器初始化时被调用一次

9. SpringIOC 是什么?

Spring IOC 负责创建对象, 管理对象 (通过依赖注入 (**DI**)), 装配对象, 配置对象, 并且管理这些对象的整个生命周期。

10. AOP 有哪些实现方式?

实现 **AOP** 的技术, 主要分为两大类:

静态代理 - 指使用 **AOP** 框架提供的命令进行编译, 从而在编译阶段就可生成 **AOP** 代理类, 因此也称为编译时增强;

编译时编织 (特殊编译器实现)

类加载时编织 (特殊的类加载器实现)。

动态代理 - 在运行时在内存中“临时”生成 **AOP** 动态代理类, 因此也被称为运行时增强。

11. 解释一下代理模式？

1、代理模式：代理模式就是本该我做的事，我不做，我交给代理人去完成。就比如，我生产了一些产品，我自己不卖，我委托代理商帮我卖，让代理商和顾客打交道，我自己负责主要产品的生产就可以了。代理模式的使用，需要有本类，和代理类，本类和代理类共同实现统一的接口。然后在 main 中调用就可以了。本类中的业务逻辑一般是不会变动的，在我们需要的时候可以不断的添加代理对象，或者修改代理类来实现业务的变更。

2、代理模式可以分为：静态代理 优点：可以做到在不修改目标对象功能的前提下，对目标功能扩展 缺点：因为本来和代理类要实现统一的接口，所以会产生很多的代理类，类太多，一旦接口增加方法，目标对象和代理对象都要维护。动态代理（JDK 代理/接口代理）代理对象，不需要实现接口，代理对象的生成，是利用 JDK 的 API，动态的在内存中构建代理对象，需要我们指定代理对象/目标对象实现的接口的类型。Cglib 代理 特点：在内存中构建一个子类对象，从而实现对目标对象功能的扩展。

3、使用场景：修改代码的时候。不用随便去修改别人已经写好的代码，如果需要修改的话，可以通过代理的方式来扩展该方法。隐藏某个类的时候，可以为其提供代理类 当我们要扩展某个类功能的时候，可以使用代理类 当一个类需要对不同的调用者提供不同的调用权限的时候，可以使用代理类来实现。减少本类代码量的时候。需要提升处理速度的时候。就比如我们在访问某个大型系统的时候，一次生成实例会耗费大量的时间，我们可以采用代理模式，当用来需要的时候才生成实例，这样就能提高访问的速度。

12. Mybatis 是如何 sql 执行结果封装为目标对象？都有哪些映射形式？

第一种是使用<resultMap>标签，逐一数据库列名和对象属性名之间的映射关系。 第

二种是使用 sql 列的别名功能，将列的别名书写为对象属性名。

有了列名与属性名的映射关系后，Mybatis 通过反射创建对象，同时使用反射给对象的属性逐一赋值并返回，那些找不到映射关系的属性，是无法完成赋值的。

13. Spring bean 的生命周期？

1、Spring 容器根据配置中的 bean 定义中实例化 bean。

2、Spring 使用依赖注入填充所有属性，如 bean 中所定义的配置。

3、如果 bean 实现 BeanNameAware 接口，则工厂通过传递 bean 的 ID 来调用 setBeanName()。

4、如果 bean 实现 BeanFactoryAware 接口，工厂通过传递自身的实例来调用 setBeanFactory()。

5、如果存在与 bean 关联的任何 BeanPostProcessors，则调用 preProcessBeforeInitialization() 方法。

6、如果为 bean 指定了 init 方法（<bean> 的 init-method 属性），那么将调用它。

7、最后，如果存在与 bean 关联的任何 BeanPostProcessors，则将调用 postProcessAfterInitialization() 方法。

8、如果 bean 实现 DisposableBean 接口，当 spring 容器关闭时，会调用 destroy()。

9、如果为 bean 指定了 destroy 方法（<bean> 的 destroy-method 属性），那么将调用它。

14. Spring 框架中都用到哪些设计模式？

代理模式，在 AOP 中被使用最多。

单例模式，在 Spring 配置文件中定义 bean 的时候默认的是单例模式。

工厂模式，BeanFactory 用来创建对象的实例。

模板方法，用来解决重复性代码。

前端控制器，Spring 提供了 DispatcherServlet 来对请求进行分发。

视图帮助，Spring 提供了一系列的 JSP 标签。

依赖注入，它是贯穿于 BeanFactory/Application Context 接口的核心理念。

15. Spring 中的事件处理？

1、Spring 的核心是 ApplicationContext,它负责管理 bean 的完整生命周期。Spring 提供了以下内置事件：ContextRefreshedEvent ContextStartedEvent ContextStoppedEvent ContextClosedEvent RequestHandleEvent

2、由于 Spring 的事件处理是单线程的，所以如果一个事件被发布，直至并且除非所有的接收者得到该消息，该进程被阻塞并且流程将不会继续。因此，如果事件处理被使用，在设计应用程序时应注意。

3、监听上下文事件

4、自定义事件

16. 使用 Spring 框架的好处是什么？

1、简化开发，解耦，集成其它框架。

2、低侵入式设计，代码污染级别低。

3、Spring 的 DI 机制降低了业务对象替换的复杂性，提高了软件之间的解耦。

4、Spring AOP 支持将一些通用的任务进行集中式的管理，例如：安全，事务，日志等，从而使代码能更好的复用。

17. 解释 Spring 支持的几种 bean 的作用域？

当通过 Spring 容器创建一个 Bean 实例的时候，不仅可以完成 bean 实例的实例化，还可以为 bean 指定作用域。Spring bean 元素的支持以下 5 种作用域：

Singleton：单例模式，在整个 Spring IOC 容器中，使用 singleton 定义的 bean 将只有一个实例。

Prototype：多例模式，每次通过容器中的 getBean 方法获取 prototype 定义的 beans 时，都会产生一个新的 bean 的实例。

Request：对于每次 Http 请求，使用 request 定义的 bean 都会产生一个新的实例，只有在 web 应用时候，该作用域才会有效。

Session：对于每次 Http Session，使用 session 定义的 Bean 都将产生一个新的实例。

Globalsession：每个全局的 Http Session，使用 session 定义的本都将产生一个新的实例。

18. 在 Spring 中如何注入一个 java 集合？

Spring 提供理论四种集合类的配置元素：

`<list>`: 该标签用来装配 有重复值的 list 值
`<set>`: 该标签用来装配没有重复值的 set 值
`<map>`: 该标签科以用来注入键值对
`<props>`: 该标签用来支持注入键值对和字符串类型键值对。

19. 什么是 Spring bean?

它们是构成用户应用程序主干的对象。

Bean 由 Spring IoC 容器管理。

它们由 Spring IoC 容器实例化，配置，装配和管理。

Bean 是基于用户提供给容器的配置元数据创建。

20. 什么是 spring 自动装配?

就是将一个 Bean 注入到其它的 Bean 的 Property 中，默认情况下，容器不会自动装配，需要我们手动设定。Spring 可以通过向 Bean Factory 中注入的方式来搞定 bean 之间的依赖关系，达到自动装配的目的。

自动装配建议少用，如果要使用，建议使用 ByName

21. 自动装配有哪些方式?

- 1、no - 这是默认设置，表示没有自动装配。应使用显式 bean 引用进行装配。
- 2、byName - 它根据 bean 的名称注入对象依赖项。它匹配并装配其属性与 XML 文件中由相同名称定义的 bean。
- 3、byType - 它根据类型注入对象依赖项。如果属性的类型与 XML 文件中的一个 bean 名称匹配，则匹配并装配属性。
- 4、构造函数 - 它通过调用类的构造函数来注入依赖项。它有大量的参数。
- 5、autodetect - 首先容器尝试通过构造函数使用 autowire 装配，如果不能，则尝试通过 byType 自动装配。

22. 自动装配有什么局限?

- 1、覆盖的可能性 - 您始终可以使用 `<constructor-arg>` 和 `<property>` 设置指定依赖项，这将覆盖自动装配。
- 2、基本元数据类型 - 简单属性（如原数据类型，字符串和类）无法自动装配。
- 3、令人困惑的性质 - 总是喜欢使用明确的装配，因为自动装配不太精确。

23. Spring 的重要注解?

`@Controller` - 用于 Spring MVC 项目中的控制器类。

`@Service` - 用于服务类。

`@RequestMapping` - 用于在控制器处理程序方法中配置 URI 映射。

`@ResponseBody` - 用于发送 Object 作为响应，通常用于发送 XML 或 JSON 数据作为响应。

`@PathVariable` - 用于将动态值从 URI 映射到处理程序方法参数。

`@Autowired` - 用于在 spring bean 中自动装配依赖项。

@Qualifier - 使用 @Autowired 注解，以避免在存在多个 bean 类型实例时出现混淆。

@Scope - 用于配置 spring bean 的范围。

@Configuration, @ComponentScan 和 @Bean - 用于基于 java 的配置。

@Aspect, @Before, @After, @Around, @Pointcut - 用于切面编程（AOP）。

24. @Component, @Controller, @Repository, @Service 有何区别？

1、@Component: 这将 java 类标记为 bean。它是任何 Spring 管理组件的通用构造型。spring 的组件扫描机制现在可以将其拾取并将其拉入应用程序环境中。

2、@Controller: 这将一个类标记为 Spring Web MVC 控制器。标有它的 Bean 会自动导入到 IoC 容器中。

3、@Service: 此注解是组件注解的特化。它不会对 @Component 注解提供任何其他行为。您可以在服务层类中使用 @Service 而不是 @Component，因为它以更好的方式指定了意图。

4、@Repository: 这个注解是具有类似用途和功能的 @Component 注解的特化。它为 DAO 提供了额外的好处。它将 DAO 导入 IoC 容器，并使未经检查的异常有资格转换为 Spring DataAccessException。

25. 列举 spring 支持的事务管理类型？

Spring 支持两种类型的事务管理：

程序化事务管理：在此过程中，在编程的帮助下管理事务。它为您提供极大的灵活性，但维护起来非常困难。

声明式事务管理：在此，事务管理与业务代码分离。仅使用注解或基于 XML 的配置来管理事务。

26. Spring 框架的事物管理有哪些优点？

1、它为不同的事务 API(如 JTA, JDBC, Hibernate, JPA, 和 JDO)提供了统一的编程模型。

2、它为程式化事务管理提供了一个简单的 API 而非一系列复杂的事务 API(如 JTA)。

3、它支持声明式事务管理。

4、它可以和 Spring 的多种数据访问技术很好的融合。

27. Spring AOP（面向切面）编程的原理？

1、AOP 面向切面编程，它是一种思想。它就是针对业务处理过程中的切面进行提取，以达到优化代码的目的，减少重复代码的目的。 就比如，在编写业务逻辑代码的时候，我们习惯性的都要写：日志记录，事物控制，以及权限控制等，每一个子模块都要写这些代码，代码明显存在重复。这时候，我们运用面向切面的编程思想，采用横切技术，将代码中重复的部分，不影响主业务逻辑的部分抽取出来，放在某个地方进行集中式的管理，调用。 形成日志切面，事物控制切面，权限控制切面。 这样，我们就只需要关系业务的逻辑处理，即提高了工作的效率，又使得代码变的简洁优雅。这就是面向切面的编程思想，它是面向对象编程思想的一种扩展。

2、AOP 的使用场景： 缓存、权限管理、内容传递、错误处理、懒加载、记录跟踪、优化、校准、调试、持久化、资源池、同步管理、事物控制等。 AOP 的相关概念： 切面（Aspect）

连接点(JoinPoint) 通知 (Advice) 切入点 (Pointcut) 代理 (Proxy): 织入 (Weaving)
3、Spring AOP 的编程原理? 代理机制 JDK 的动态代理: 只能用于实现了接口的类产生代理。
Cglib 代理: 针对没有实现接口的类产生代理, 应用的是底层的字节码增强技术, 生成当前类的子类对象。

28. Spring MVC 框架有什么用?

Spring Web MVC 框架提供 模型-视图-控制器 架构和随时可用的组件, 用于开发灵活且松散耦合的 Web 应用程序。 MVC 模式有助于分离应用程序的不同方面, 如输入逻辑, 业务逻辑和 UI 逻辑, 同时在这些元素之间提供松散耦合。

29. 介绍一下 WebApplicationContext?

WebApplicationContext 是 ApplicationContext 的扩展。它具有 Web 应用程序所需的一些额外功能。它与普通的 ApplicationContext 在解析主题和决定与哪个 servlet 关联的能力方面有所不同。

30. SpringMVC 和 struts2 的区别有哪些?

一、拦截机制的不同

Struts2 是类级别的拦截, 每次请求就会创建一个 Action, 和 Spring 整合时 Struts2 的 ActionBean 注入作用域是原型模式 prototype, 然后通过 setter, getter 吧 request 数据注入到属性。Struts2 中, 一个 Action 对应一个 request, response 上下文, 在接收参数时, 可以通过属性接收, 这说明属性参数是让多个方法共享的。Struts2 中 Action 的一个方法可以对应一个 url, 而其类属性却被所有方法共享, 这也就无法用注解或其他方式标识其所属方法了, 只能设计为多例。

SpringMVC 是方法级别的拦截, 一个方法对应一个 Request 上下文, 所以方法直接基本上是独立的, 独享 request, response 数据。而每个方法同时又何一个 url 对应, 参数的传递是直接注入到方法中的, 是方法所独有的。处理结果通过 ModelAndView 返回给框架。在 Spring 整合时, SpringMVC 的 Controller Bean 默认单例模式 Singleton, 所以默认对所有的请求, 只会创建一个 Controller, 有应为没有共享的属性, 所以是线程安全的, 如果要改变默认的作用域, 需要添加@Scope 注解修改。

Struts2 有自己的拦截 Interceptor 机制, SpringMVC 这是用的是独立的 Aop 方式, 这样导致 Struts2 的配置文件量还是比 SpringMVC 大。

二、底层框架的不同

Struts2 采用 Filter (StrutsPrepareAndExecuteFilter) 实现, SpringMVC (DispatcherServlet) 则采用 Servlet 实现。Filter 在容器启动之后即初始化; 服务停止以后坠毁, 晚于 Servlet。Servlet 是在调用时初始化, 先于 Filter 调用, 服务停止后销毁。

三、性能方面

Struts2 是类级别的拦截, 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, SpringMVC 实现了零配置, 由于 SpringMVC 基于方法的拦截, 有加载一次单例模式 bean 注入。所以, SpringMVC 开发效率和性能高于 Struts2。

四、配置方面

spring MVC 和 Spring 是无缝的。从这个项目的管理和安全上也比 Struts2 高。

31. Mybatis 中#{ }和\${ }的区别是什么?

#{ }是预编译处理，\${ }是字符串替换。

Mybatis 在处理#{ }时，会将 sql 中的#{ }替换为?号，调用 PreparedStatement 的 set 方法来赋值；

Mybatis 在处理\${ }时，就是把\${ }替换成变量的值。使

用#{ }可以有效的防止 SQL 注入，提高系统安全性。

32. Spring 中@Autowired 与@Resource 的区别?

@Autowired 默认按照类型装配，默认情况下它要求依赖对象必须存在如果允许为 null，可以设置它 required 属性为 false，如果我们想使用按照名称装配，可以结合@Qualifier 注解一起使用；

@Resource 默认按照名称装配，当找不到与名称匹配的 bean 才会按照类型装配，可以通过 name 属性指定，如果没有指定 name 属性，当注解标注在字段上，即默认取字段的名称作为 bean 名称寻找依赖对象，当注解标注在属性的 setter 方法上，即默认取属性名作为 bean 名称寻找依赖对象

33. 什么是控制反转（IOC），什么是依赖注入（DI）？

IOC：就是对象之间的依赖关系由容器来创建，对象之间的关系本来是由我们开发者自己创建和维护的，在我们使用 Spring 框架后，对象之间的关系由容器来创建和维护，将开发者做的事让容器做，这就是控制反转。BeanFactory 接口是 Spring IOC 容器的核心接口。

DI：我们在使用 Spring 容器的时候，容器通过调用 set 方法或者是构造器来建立对象之间的依赖关系。

控制反转是目标，依赖注入是我们实现控制反转的一种手段。

34. Spring 运行原理?

- 1、内部最核心的就是 IOC 了，之前是 new 对象，现在可以直接从容器中获取，动态注入，这其实就是利用 java 里的反射。反射其实就是在运行时动态的去创建、调用对象，Spring 就是在运行时，根据 xml Spring 的配置文件来动态的创建对象，和调用对象里的方法的。
- 2、Spring 另一个核心就是 AOP 面向切面编程，可以为某一类对象 进行监督和控制（也就是在调用这类对象的具体方法的前后去调用你指定的 模块）从而达到对一个模块扩充的功能。这些都是通过配置类达到的。（日志、事务等）
- 3、Spring 目的：就是让对象与对象（模块与模块）之间的关系没有通过代码来关联，都是通过配置类说明管理的（Spring 根据这些配置 内部通过反射去动态的组装对象）要记住：Spring 是一个容器，凡是在容器里的对象才会有 Spring 所提供的这些服务和功能。
- 4、Spring 里用的最经典设计模式：模板方法模式。（有兴趣同学可以了解一下）、核心容器组件是 BeanFactory，它是工厂模式的实现。BeanFactory 使用控制反转（IOC）模式将应用程序的配置和依赖性规范与实际的应用程序代码分开。

35. Spring 的事务传播行为?

PROPAGATION(蔓延、传播、传输)

事务传播行为类型	说明
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。这是默认的事务传播行为
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。（一个新的事务将启动，而且如果有一个现有的事务在运行的话，则这个方法将在运行期被挂起，直到新的事务提交或者回滚才恢复执行。）
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。（外层事务抛出异常回滚，那么内层事务必须回滚，反之内层事务并不影响 外层事务）

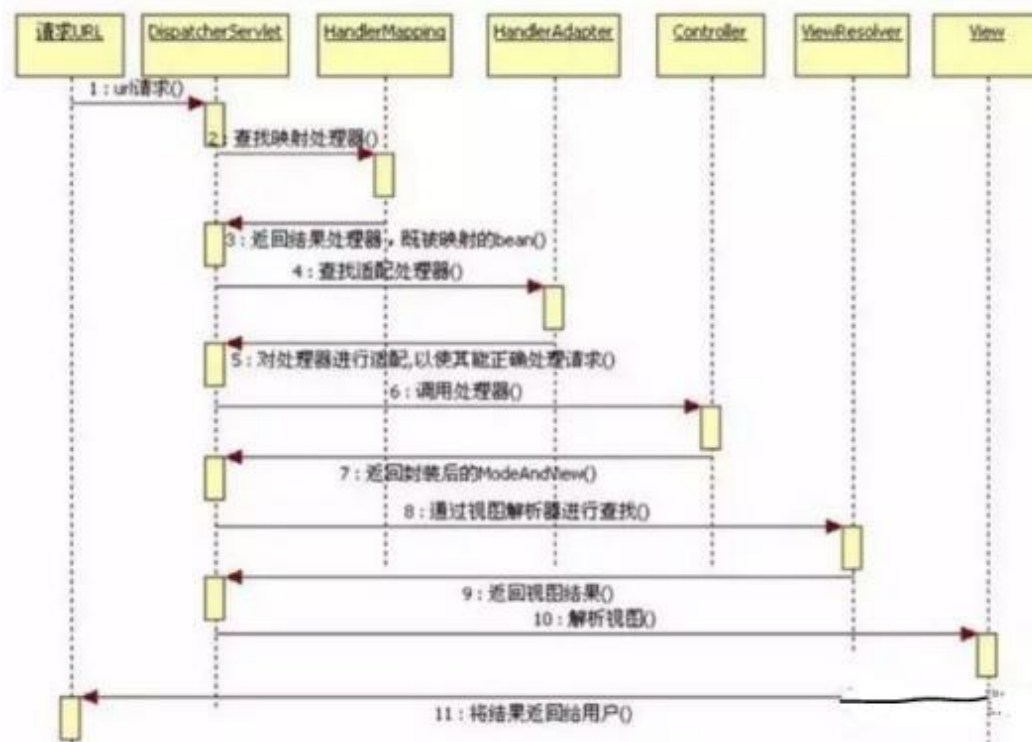
36. SpringMVC 的运行流程？

DispatcherServlet 前置控制器

|HandlerMapping 请求映射（到 Controller）

|HandlerAdapter 请求映射（到 Controller 类的方法上）|Controller 控制器|

HandlerInterceptor 拦截器|ViewResolver 视图映射|View 视图处理



八、分布式相关

1. Redis 和 Memcache 的区别?

- 1、存储方式 Memecache 把数据全部存在内存之中，断电后会挂掉，数据不能超过内存大小。 Redis 有部份存在硬盘上，redis 可以持久化其数据
- 2、数据支持类型 memcached 所有的值均是简单的字符串，redis 作为其替代者，支持更为丰富的数据类型，提供 list, set, zset, hash 等数据结构的存储
- 3、使用底层模型不同 它们之间底层实现方式 以及与客户端之间通信的应用协议不一样。Redis 直接自己构建了 VM 机制，因为一般的系统调用系统函数的话，会浪费一定的时间去移动和请求。
- 4、value 值大小不同：Redis 最大可以达到 1gb；memcache 只有 1mb。
- 5、redis 的速度比 memcached 快很多
- 6、Redis 支持数据的备份，即 master-slave 模式的数据备份。

2. 使用 Redis 有哪些好处?

- (1) 速度快，因为数据存在内存中，类似于 HashMap，HashMap 的优势就是查找和操作的时间复杂度都是 $O(1)$
- (2) 支持丰富数据类型，支持 string, list, set, sorted set, hash
- (3) 支持事务，操作都是原子性，所谓的原子性就是对数据的更改要么全部执行，要么全部不执行
- (4) 丰富的特性：可用于缓存，消息，按 key 设置过期时间，过期后将会自动删除

3. 什么是 redis 持久化? rdb 和 aof 的比较?

持久化就是把内存的数据写到磁盘中去，防止服务宕机了内存数据丢失。

比较：

- 1、aof 文件比 rdb 更新频率高，优先使用 aof 还原数据。
- 2、aof 比 rdb 更安全也更大
- 3、rdb 性能比 aof 好
- 4、如果两个都配了优先加载 AOF

4. Redis 最适合的场景?

(1)、会话缓存 (Session Cache)

最常用的一种使用 Redis 的情景是会话缓存 (session cache)。用 Redis 缓存会话比其他存储 (如 Memcached) 的优势在于：Redis 提供持久化。

(2)、全页缓存 (FPC)

除基本的会话 token 之外，Redis 还提供很简便的 FPC 平台。回到一致性问题，即使重启了 Redis 实例，因为有磁盘的持久化，用户也不会看到页面加载速度的下降，这是一个极大改进，类似 PHP 本地 FPC。

再次以 Magento 为例，Magento 提供一个插件来使用 Redis 作为全页缓存后端。

此外，对 WordPress 的用户来说，Pantheon 有一个非常好的插件 wp-redis，这个插件能帮

助你以最快速度加载你曾浏览过的页面。

(3)、队列

Redis 在内存存储引擎领域的一大优点是提供 list 和 set 操作，这使得 Redis 能作为一个很好的消息队列平台来使用。Redis 作为队列使用的操作，就类似于本地程序语言（如 Python）对 list 的 push/pop 操作。

(4)、排行榜/计数器

Redis 在内存中对数字进行递增或递减的操作实现的非常好。集合（Set）和有序集合（Sorted Set）也使得我们在执行这些操作的时候变的非常简单，Redis 只是正好提供了这两种数据结构。所以，我们要从排序集合中获取到排名最靠前的 10 个用户 - 我们称之为“user_scores”，我们只需要像下面一样执行即可：

(5)、发布/订阅

最后（但肯定不是最不重要的）是 Redis 的发布/订阅功能。发布/订阅的使用场景确实非常多。我已看见人们在社交网络连接中使用，还可作为基于发布/订阅的脚本触发器，甚至用 Redis 的发布/订阅功能来建立聊天系统！

5. redis 哈希槽的概念？

Redis 集群没有使用一致性 hash,而是引入了哈希槽的概念，Redis 集群有 16384 个哈希槽，每个 key 通过 CRC16 校验后对 16384 取模来决定放置哪个槽，集群的每个节点负责一部分 hash 槽。

6. 怎么理解 Redis 事务？

事务是一个单独的隔离操作：事务中的所有命令都会序列化、按顺序地执行，事务在执行的过程中，不会被其他客户端发送来的命令请求所打断。事务是一个原子操作：事务中的命令要么全部被执行，要么全部都不执行。

7. redis 的淘汰策略有哪些？

noeviction:返回错误当内存限制达到并且客户端尝试执行会让更多内存被使用的命令（大部分的写入指令，但 DEL 和几个例外）

allkeys-lru: 尝试回收最少使用的键（LRU），使得新添加的数据有空间存放。

volatile-lru: 尝试回收最少使用的键（LRU），但仅限于在过期集合的键,使得新添加的数据有空间存放。

allkeys-random: 回收随机的键使得新添加的数据有空间存放。

volatile-random: 回收随机的键使得新添加的数据有空间存放，但仅限于在过期集合的键。

volatile-ttl: 回收在过期集合的键，并且优先回收存活时间（TTL）较短的键,使得新添加的数据有空间存放。

8. redis 有哪些数据结构？

String、List、Set、Zset(Sorted Set)、hash

9. redis 缓存穿透、缓存雪崩、缓存击穿？

缓存穿透：无效 ID，在 redis 缓存中查不到，去查询 DB，造成 DB 压力增大。

解决方法：

1、解决方法 1：布隆过滤器，提供一个很大的 Bit-Map，提供多个 hash 函数，分别对查询参数值【比如 UUID】，进行求 hash，然后分别对多个 hash 结果，在对应位置对比是否全为 1 或者某个位置为 0，一旦有一个位置标识为 0，表示本次查询 UUID，不存在于缓存，再去查询 DB，起到一个再过滤的效果。

2、解决方法 2：把无效的 ID，也在 redis 缓存起来，并设置一个很短的超时时间。

缓存雪崩：缓存同一时间批量失效，导致大量的访问直接访问 DB

解决方法：

在做缓存时候，就做固定失效时间+随机时间段，保证所有的缓存不会同一时间失效 缓存击穿：在缓存失效的时候，会有高并发访问失效的缓存【热点数据】

解决方法：

最简单的解决方法，就是将热点数据设置永不超时！

第二个解决方法：对访问的 Key 加上互斥锁，请求的 Key 如果不存在，则加锁，去数据库取，新请求过来，如果相同 Key，则暂停 10s 再去缓存取值；如果 Key 不同，则直接去缓存取！

10. redis 如何实现高并发？

redis 通过一主多从，主节点负责写，从节点负责读，读写分离，从而实现高并发。

11. redis 如何实现高可用？

主备切换，哨兵集群，主节点宕机的情况下，自动选举出一个从节点变成主节点，从而保证了 redis 集群的高可用。

12. redis 单线程还能处理速度那么快？

首先，redis 是单进程单线程的 k-v 内存型可持久化数据库。

单线程还能处理速度很快的原因：

- 1、redis 操作是基于内存的，内存的读写速度非常快
- 2、正是由于 redis 的单线程模式，避免了线程上下文切换的损耗
- 3、redis 采用的 IO 多路复用技术，可以很好的解决多请求并发的的问题。多路代表多请求，复用代表多个请求重复使用同一个线程。

13. 为什么 Redis 的操作是原子性的，怎么保证原子性？

对于 Redis 而言，命令的原子性指的是：一个操作的不可再分，操作要么执行，要么不执行。

Redis 的操作之所以是原子性的，是因为 Redis 是单线程的。

Redis 本身提供的所有 API 都是原子操作，Redis 中的事务其实是要保证批量操作的原子性。多个命令在并发中也是原子性的吗？

不一定，将 get 和 set 改成单命令操作，incr。使用 Redis 的事务，或者使用 Redis+Lua== 的方式实现。

14. redis 的主从复制的实现过程?

- 1、从服务发送一个 `sync` 同步命令给主服务要求全量同步
- 2、主服务接收到从服务的 `sync` 同步命令时，会 `fork` 一个子进程后台执行 `bgsave` 命令（非阻塞）快照保存，生成 `RDB` 文件，并将 `RDB` 文件发送给从服务
- 3、从服务再将接收到的 `RDB` 文件载入自己的 `redis` 内存
- 4、待从服务将 `RDB` 载入完成后，主服务再将缓冲区所有写命令发送给从服务
- 5、从服务在将主服务所有的写命令载入内存从而实现数据的完整同步
- 6、从服务下次在需要同步数据时只需要发送自己的 `offset` 位置（相当于 `mysql binlog` 的位置）即可，只同步新增加的数据，再不需要全量同步

15. redis 的哨兵机制的作用?

- 1、监控: Sentinel 会不断的检查主服务器和从服务器是否正常运行。
- 2、通知: 当被监控的某个 `redis` 服务器出现问题，Sentinel 通过 `API` 脚本向管理员或者其他的应用程序发送通知。
- 3、自动故障转移: 当主节点不能正常工作时，Sentinel 会开始一次自动的故障转移操作，它会将与失效主节点是主从关系的其中一个从节点升级为新的主节点，并且将其他的从节点指向新的主节点。

16. redis 常见的性能问题和解决方案?

- (1) Master 最好不要做任何持久化工作，如 `RDB` 内存快照和 `AOF` 日志文件
- (2) 如果数据比较重要，某个 `Slave` 开启 `AOF` 备份数据，策略设置为每秒同步一次
- (3) 为了主从复制的速度和连接的稳定性，Master 和 Slave 最好在同一个局域网内
- (4) 尽量避免在压力很大的主库上增加从库
- (5) 主从复制不要用图状结构，用单向链表结构更为稳定，即： Master <- Slave1 <- Slave2 <- Slave3...

17. 分布式缓存?

硬盘上的数据，缓存在别的计算机上（非程序运行的计算机）的内存上，而且可以缓存的计算机的个数不止一个，可以使用 `n` 个用户通过访问 `http` 服务器，然后访问应用服务器资源，应用服务器调用后端的数据库，在第一次访问的时候，直接访问数据库，然后将要缓存的内容放入到 `memcached` 集群，集群规模根据缓存文件的大小而定。在第二次访问的时候就直接进入缓存读取，不需要进行数据库的操作。这个适合数据变化不频繁的场景，比如：互联网站显示的榜单，阅读排行等。

18. 什么是 Nginx?

Nginx 是一个高性能的 `HTTP` 和反向代理服务器，及电子邮件代理服务器，同时也是一个非常高效的反向代理、负载平衡。

19. nginx 相对于 apache 的优点?

轻量级，同样起 web 服务，比 apache 占用更少的内存及资源
抗并发，nginx 处理请求是异步非阻塞的，而 apache 则是阻塞型的，在高并发下 nginx 能保持低资源低消耗高性能
高度模块化的设计，编写模块相对简单
社区活跃，各种高性能模块出品迅速啊
rewrite，比 nginx 的 rewrite 强大
模块超多，基本想到的都可以找到
少 bug，nginx 的 bug 相对较多

20. Nginx 优化的方式?

Nginx 运行工作进程数量

Nginx 运行工作进程个数一般设置 CPU 的核心或者核心数 x2

Nginx 运行 CPU 亲和力

比如 4 核配置:

```
worker_processes 4;
```

```
worker_cpu_affinity 0001 0010 0100 1000
```

Nginx 最大打开文件数

```
worker_rlimit_nofile 65535;
```

Nginx 事件处理模型

```
events {  
    use epoll;  
    worker_connections 65535;  
    multi_accept on;  
}
```

nginx 采用 epoll 事件模型，处理效率高。

开启高效传输模式

连接超时时间

主要目的是保护服务器资源，CPU，内存，控制连接数，因为建立连接也是需要消耗资源的

21. Nginx 如何处理一个请求的?

首先，nginx 在启动时，会解析配置文件，得到需要监听的端口与 ip 地址，然后在 nginx 的 master 进程里面先初始化好这个监控的 socket，再进行 listen,然后再 fork 出多个子进程出来，子进程会竞争 accept 新的连接。此时，客户端就可以向 nginx 发起连接了。当客户端与 nginx 进行三次握手，与 nginx 建立好一个连接后,此时，某一个子进程会 accept 成功，然后创建 nginx 对连接的封装，即 ngx_connection_t 结构体,接着，根据事件调用相应的事件处理模块，如 http 模块与客户端进行数据的交换。最后，nginx 或客户端来主动关掉连接，到此，一个连接就寿终正寝了

22. Nginx 是如何实现高并发的?

nginx 之所以可以实现高并发，与它采用的 epoll 模型有很大的关系。epoll 模型采用异步非阻塞的事件处理机制。这种机制可让 nginx 进程同时监控多个事件。

简单来说，就是异步非阻塞，使用了 `epoll` 模型和大量的底层代码优化。如果深入一点的话，就是 `nginx` 的特殊进程模型和事件模型的设计，才使其可以实现高并发。

23. Nginx 的进程模型？

它是采用一个 `master` 进程和多个 `worker` 进程的工作模式。

- 1、`master` 进程主要负责收集、分发请求。当一个请求过来时，`master` 拉起一个 `worker` 进程负责处理这个请求。；
- 2、`master` 进程也要负责监控 `worker` 的状态，保证高可靠性；
- 3、`worker` 进程议案设置为和 CPU 核心数一致或者其二倍。`nginx` 的 `worker` 进程和 `Apache` 的不一样。`apache` 的进程在同一时间只能处理一个请求，所以它会开启很多个进程，几百甚至几千个。而 `nginx` 的 `worker` 进程在同一时间可以处理的请求数只受内存限制，因此可以处理更多请求。

24. Nginx 负载均衡的 4 种分配方式？

1、轮询（默认）

每个请求按时间顺序逐一分配到不同的后端服务器，如果后端服务器 `down` 掉，能自动剔除。

2、weight

指定轮询几率，`weight` 和访问比率成正比，用于后端服务器性能不均的情况。

2、ip_hash

每个请求按访问 `ip` 的 `hash` 结果分配，这样每个访客固定访问一个后端服务器，可以解决的问题。

3、fair（第三方）

按后端服务器的响应时间来分配请求，响应时间短的优先分配。

4、url_hash（第三方）

按访问 `url` 的 `hash` 结果来分配请求，使同样的 `url` 定向到同一个后端服务器，后端服务器为缓存时比较有效

25. 为什么要用 Nginx？

跨平台、配置简单，非阻塞、高并发连接：处理 2-3 万并发连接数，官方监测能支持 5 万并发，

内存消耗小：开启 10 个 `nginx` 才占 150M 内存，`nginx` 处理静态文件好，耗费内存少，

内置的健康检查功能：如果有一个服务器宕机，会做一个健康检查，再发送的请求就不会发送到宕机的服务器了。重新将请求提交到其他的节点上。

节省宽带：支持 `GZIP` 压缩，可以添加浏览器本地缓存

稳定性高：宕机的概率非常小

接收用户请求是异步的：浏览器将请求发送到 `nginx` 服务器，它先将用户请求全部接收下来，再一次性发送给后端 `web` 服务器，极大减轻了 `web` 服务器的压力，一边接收 `web` 服务器的返回数据，一边发送给浏览器客户端，网络依赖性比较低，只要 `ping` 通就可以负载均衡，可以有多台 `nginx` 服务器使用 `dns` 做负载均衡，事件驱动：通信机制采用 `epoll` 模型（`nio2` 异步非阻塞）

26. 什么是正向代理？

一个位于客户端和原始服务器之间的服务器，为了从原始服务器取得内容，客户端向代理发送一个请求并指定目标(原始服务器)，然后代理向原始服务器转交请求并将获得的内容返回给客户端。客户端才能使用正向代理

正向代理总结就一句话：代理端代理的是客户端

27. 什么是反向代理？

反向代理是指以代理服务器来接受 internet 上的连接请求，然后将请求，发给内部网络上的服务器,并将从服务器上得到的结果返回给 internet 上请求连接的客户端，此时代理服务器对外就表现为一个反向代理服务器

反向代理总结就一句话：代理端代理的是服务端

28. 什么是负载均衡？

负载均衡即是代理服务器将接收的请求均衡的分发到各服务器中，负载均衡主要解决网络拥塞问题，提高服务器响应速度，服务就近提供，达到更好的访问质量，减少后台服务器大并发压力

29. Nginx 的调度算法有哪些？

1、sticky: 通过 nginx-sticky 模块，来实现 cookie 黏贴的方式将来自同一个客户端的请求发送到同一个后端服务器上处理，这样一定程度上可以解决多个后端服务器的 session 会话同步的问题；

2、round-robin (RR)：轮询，每个请求按时间顺序依次分配到不同的后端服务器，如果后端某台服务器死机，自动剔除故障系统，使用户访问不受影响；

3、weight: 轮询权重，weight 的值越大分配到的访问概率就越高，主要用于后端每台服务器性能不均衡的情况下，或者仅仅为在主从的情况下设置不同的权重，达到合理有效的利用主机资源。

4、least_conn: 请求被发送到当前活跃连接最少的 realserver 上，会考虑到 weight 的值；

5、ip_hash: 每个请求按照 IP 的哈希结果分配，使来自同一个 IP 的访客固定访问后端服务器，可以有效的解决动态网页存在的 session 共享问题。

6、fair: 比 weight、ip_hash 更加智能的负载均衡算法，fair 算法可以根据页面的大小和加载时间长短智能地进行负载均衡，也就是根据后端服务器的响应时间来分配请求，相应时间短的优先分配。nginx 本身不支持 fair，如果需要使用这种调度算法，则必须安装 upstream_fair 模块。

7、url_hash: 按访问的 URL 的哈希结果来分配请求，使每个 URL 定向到后端服务器，可以进一步提高后端缓存服务器的效率。同样，nginx 本身不支持 url_hash，如果需要这种调度算法，则必须安装 nginx 的 hash 软件包。

30. Nginx 负载均衡调度状态？

常用的状态有：

1、down: 表示当前的 server 暂时不参与负载均衡；

2、backup: 预留的备份机器。当其他所有的非 backup 机器出现故障或者繁忙的时候，才会请求 backup 机器，因此这台机器的访问压力最低；

3、`max_fails`: 允许请求失败的次数，默认为 1，当超过最大次数时，返回 `proxy_next_upstream` 模块定义的错误；

4、`fail_timeout`: 请求失败超时时间，在经历了 `max_fails` 次失败后，暂停服务的时间。`max_fails` 和 `fail_timeout` 可以一起使用。

31. 可以从哪些方面来优化 nginx 服务？

- 1、配置 nginx 的 proxy 缓存；
- 2、对静态页面开启压缩功能，如 br 压缩或者 gzip 压缩；
- 3、调整 nginx 运行工作进程个数，最多开启 8 个，8 个以上性能就不会再提升了，而且稳定性变得更低，所以 8 个足够用了；
- 4、调整 nginx 运行 CPU 的亲和力；
- 5、修改 nginx 最多可打开的文件数，若超过系统限制的最多打开文件数（`ulimit -n` 命令查看系统的最多打开文件数），还需要修改系统默认的文件数；
- 6、修改单个 worker 的最大连接数；
- 7、开启高效传输；
- 8、设置连接超时时间，以便保护服务器资源，因为建立连接也是需要消耗资源的；
- 9、优化 fastCGI 的一个超时时间，也可以根据实际情况对其配置缓存动态页面；
- 10、expires 缓存调优，主要针对图片、css、js 等元素更改较少的情况下使用。
- 11、配置防盗链；
- 12、优化内核参数，如进程可以同时打开的最大句柄数；开启 tcp 重用机制，以便允许 TIME_WAIT sockets 重新用于新的 TCP 连接

32. 为什么要用 MQ？

- 1、解耦：如果多个模块或者系统中，互相调用很复杂，维护起来比较麻烦，但是这个调用又不是同步调用，就可以运用 MQ 到这个业务中。
- 2、异步：这个很好理解，比如用户的操作日志的维护，可以不用同步处理，节约响应时间。
- 3、削峰：在高峰期的时候，系统每秒的请求量达到 5000，那么调用 MySQL 的请求也是 5000，一般情况下 MySQL 的请求大概在 2000 左右，那么在高峰期的时候，数据库就被打垮了，那系统就不可用了。此时引入 MQ，在系统 A 前面加个 MQ，用户请求先到 MQ，系统 A 从 MQ 中每秒消费 2000 条数据，这样就把本来 5000 的请求变为 MySQL 可以接受的请求数量了，可以保证系统不挂掉，可以继续提供服务。MQ 里的数据可以慢慢的把它消费掉。

33. 使用 MQ 会有什么问题？

- （1）降低了系统可用性 （2）增加了系统的复杂性

34. 怎么保证 MQ 的高可用？

RabbitMQ 是比较有代表性的，因为是基于主从做高可用性的。以他为例，自行查阅以下模式。

rabbitmq 有三种模式：单机模式、普通集群模式、镜像集群模式。

35. MQ 的优缺点?

在特殊场景下有其对应的好处，解耦、异步、削峰。

缺点有以下几个：

系统可用性降低

系统引入的外部依赖越多，越容易挂掉。万一 MQ 挂了，MQ 一挂，整套系统崩溃，你不就完了？

系统复杂度提高

硬生生加个 MQ 进来，你怎么保证消息没有重复消费？怎么处理消息丢失的情况？怎么保证消息传递的顺序性？问题一大堆。

一致性问题

A 系统处理完了直接返回成功了，人都以为你这个请求就成功了；但是问题是，要是 BCD 三个系统那里，BD 两个系统写库成功了，结果 C 系统写库失败了，咋整？你这数据就不一致了。

36. Kafka、ActiveMQ、RabbitMQ、RocketMQ 都有什么区别？

对于吞吐量来说 kafka 和 RocketMQ 支撑高吞吐，ActiveMQ 和 RabbitMQ 比他们低一个数量级。对于延迟量来说 RabbitMQ 是最低的。

1. 从社区活跃度

按照目前网络上的资料，RabbitMQ、activeM、ZeroMQ 三者中，综合来看，RabbitMQ 是首选。

2. 持久化消息比较

ActiveMq 和 RabbitMq 都支持。持久化消息主要是指我们机器在不可抗力因素等情况下挂掉了，消息不会丢失的机制。

3. 综合技术实现

可靠性、灵活的路由、集群、事务、高可用的队列、消息排序、问题追踪、可视化管理工具、插件系统等等。

RabbitMq/Kafka 最好，ActiveMq 次之，ZeroMq 最差。当然 ZeroMq 也可以做到，不过自己必须手动写代码实现，代码量不小。尤其是可靠性中的：持久性、投递确认、发布者证实和高可用性。

4. 高并发

毋庸置疑，RabbitMQ 最高，原因是它的实现语言是天生具备高并发高可用的 erlang 语言。

5. 比较关注的比较，RabbitMQ 和 Kafka

RabbitMq 比 Kafka 成熟，在可用性上，稳定性上，可靠性上，RabbitMq 胜于 Kafka（理论上）。另外，Kafka 的定位主要在日志等方面，因为 Kafka 设计的初衷就是处理日志的，可以看做是一个日志（消息）系统一个重要组件，针对性很强，所以如果业务方面还是建议选择 RabbitMq。

还有就是，Kafka 的性能（吞吐量、TPS）比 RabbitMq 要高出来很多。

37. 如何设置消息的过期时间？

设置队列属性，队列中所有消息都有相同的过期时间

对消息本身进行单独设置，每条消息的 TTL 可以不同

如果两种方法一起使用，则消息的 TTL 以两者之间较小的那个数值为准

38. 消息的持久化是如何实现的？

RabbitMQ 的持久化分为:交换器的持久化、队列的持久化和消息的持久化。交换器和队列的持久化都是通过在声明时将 `durable` 参数置为 `true` 实现的。消息的持久化是在发送消息指定 `deliveryMode` 为 2 实现的。

39. Zookeeper 是什么？

ZooKeeper 是一个开放源码的分布式协调服务，它是集群的管理者，监视着集群中各个节点的状态。根据节点提交的反馈进行下一步合理操作。最终，将简单易用的接口和性能高效、功能稳定的系统提供给用户。

40. Zookeeper 的应用场景？

- 数据发布/订阅
- 负载均衡
- 命名服务
- 分布式协调/通知
- 集群管理
- Master 选举
- 分布式锁
- 分布式队列

41. 四种类型的数据节点 Znode？

PERSISTENT-持久节点

除非手动删除，否则节点一直存在于 Zookeeper 上。

EPHEMERAL-临时节点

临时节点的生命周期与客户端会话绑定，一旦客户端会话失效（客户端与 zookeeper 连接断开不一定会话失效），那么这个客户端创建的所有临时节点都会被移除。

PERSISTENT_SEQUENTIAL-持久顺序节点

基本特性同持久节点，只是增加了顺序属性，节点名后边会追加一个由父节点维护的自增整型数字。

EPHEMERAL_SEQUENTIAL-临时顺序节点

基本特性同临时节点，增加了顺序属性，节点名后边会追加一个由父节点维护的自增整型数字。

42. Zookeeper Watcher 机制？

1、一次性

无论是服务端还是客户端，一旦一个 Watcher 被触发，Zookeeper 都会将其从相应的存储中移除。这样的设计有效的减轻了服务端的压力，不然对于更新非常频繁的节点，服务端会不断的向客户端发送事件通知，无论对于网络还是服务端的压力都非常大。

2、客户端串行执行

客户端 `Watcher` 回调的过程是一个串行同步的过程。

3、轻量

3.1 `Watcher` 通知非常简单，只会告诉客户端发生了事件，而不会说明事件的具体内容。

3.2 客户端向服务端注册 `Watcher` 的时候，并不会把客户端真实的 `Watcher` 对象实体传递到服务端，仅仅是在客户端请求中使用 `boolean` 类型属性进行了标记。

4、`watcher event` 异步发送 `watcher` 的通知事件从 `server` 发送到 `client` 是异步的，这就存在一个问题，不同的客户端和服务端之间通过 `socket` 进行通信，由于网络延迟或其他因素导致客户端在不通的时刻监听到事件，由于 `Zookeeper` 本身提供了 `ordering guarantee`，即客户端监听事件后，才会感知它所监视 `znode` 发生了变化。所以我们使用 `Zookeeper` 不能期望能够监控到节点每次的变化。`Zookeeper` 只能保证最终的一致性，而无法保证强一致性。

5、注册 `watcher` `getData`、`exists`、`getChildren`

6、触发 `watcher` `create`、`delete`、`setData`

7、当一个客户端连接到一个新的服务器上时，`watch` 将会被以任意会话事件触发。当与一个服务器失去连接的时候，是无法接收到 `watch` 的。而当 `client` 重新连接时，如果需要的话，所有先前注册过的 `watch`，都会被重新注册。通常这是完全透明的。只有在一个特殊情况下，`watch` 可能会丢失：对于一个未创建的 `znode` 的 `exist watch`，如果在客户端断开连接期间被创建了，并且随后在客户端连接上之前又删除了，这种情况下，这个 `watch` 事件可能会被丢失。

43. Zookeeper 下 Server 工作状态？

服务器具有四种状态，分别是 `LOOKING`、`FOLLOWING`、`LEADING`、`OBSERVING`。
`LOOKING`：寻找 `Leader` 状态。当服务器处于该状态时，它会认为当前集群中没有 `Leader`，因此需要进入 `Leader` 选举状态。

`FOLLOWING`：跟随者状态。表明当前服务器角色是 `Follower`。

`LEADING`：领导者状态。表明当前服务器角色是 `Leader`。

`OBSERVING`：观察者状态。表明当前服务器角色是 `Observer`。

44. Zookeeper 是如何保证事务的顺序一致性的？

`zookeeper` 采用了全局递增的事务 `Id` 来标识，所有的 `proposal`（提议）都在被提出的时候加上了 `zxid`，`zxid` 实际上是一个 64 位的数字，高 32 位是 `epoch`（时期；纪元；世；新时代）用来标识 `leader` 周期，如果有新的 `leader` 产生出来，`epoch` 会自增，低 32 位用来递增计数。当新产生 `proposal` 的时候，会依据数据库的两阶段过程，首先会向其他的 `server` 发出事务执行请求，如果超过半数的机器都能执行并且能够成功，那么就会开始执行。

45. ZK 节点宕机如何处理？

`Zookeeper` 本身也是集群，推荐配置不少于 3 个服务器。`Zookeeper` 自身也要保证当一个节点宕机时，其他节点会继续提供服务。

如果是一个 `Follower` 宕机，还有 2 台服务器提供访问，因为 `Zookeeper` 上的数据是有多个副本的，数据并不会丢失；

如果是一个 `Leader` 宕机，`Zookeeper` 会选举出新的 `Leader`。

`ZK` 集群的机制是只要超过半数的节点正常，集群就能正常提供服务。只有在 `ZK` 节点挂得太

多，只剩一半或不到一半节点能工作，集群才失效。

所以

3 个节点的 cluster 可以挂掉 1 个节点(leader 可以得到 2 票>1.5)

2 个节点的 cluster 就不能挂掉任何 1 个节点了(leader 可以得到 1 票<=1)

46. Zookeeper 有哪几种部署模式？

部署模式：单机模式、伪集群模式、集群模式。

47. Dubbo 内置了哪几种容器？

Spring Container

Jetty Container

Log4j Container

48. Dubbo 里面有哪几种角色？

Provider：暴露服务的服务提供方

Consumer：调用远程服务的服务消费方

Registry：服务注册与发现的注册中心

Monitor：统计服务的调用次数和调用时间的监控中心

Container：服务运行容器

49. Dubbo 有哪几种集群容错方案，默认是那种？

Failover Cluster：失败自动切换，自动重试其他服务器（默认）

Failfast Cluster：快速失败，立即报错，只发起一次调用

Failsafe Cluster：失败安全，出现异常时，直接忽略

Failback Cluster：失败自动恢复，记录失败请求，定时重发

Forking Cluster：并行调用多个服务器，只要一个成功即返回

Broadcast Cluster：广播逐个调用所有提供者，任意一个报错则报错

50. Dubbo 有哪几种负载均衡策略，默认是哪种？

Random LoadBalance:随机，按权重设置随机概率(默认)

RoundRobin LoadBalance：轮询，按公约后的权重设置轮询比率

LeastActive LoadBalance：最少活跃调用次数，相同活跃数的随机

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

51. Dubbo 的管理控制台能做什么？

管理控制台主要包含：路由规则，动态配置，服务降级，访问控制，权重调整，负载均衡，等管理功能。

52. 什么是 Spring Boot？

做真实的自己，用匠心做教育

多年来，随着新功能的增加，spring 变得越来越复杂。只需访问 <https://spring.io/projects> 页面，我们就会看到可以在我们的应用程序中使用的所有 Spring 项目的不同功能。如果必须启动一个新的 Spring 项目，我们必须添加构建路径或添加 Maven 依赖关系，配置应用程序服务器，添加 spring 配置。因此，开始一个新的 spring 项目需要很多努力，因为我们现在必须从头开始做所有事情。

Spring Boot 是解决这个问题的方法。Spring Boot 已经建立在现有 spring 框架之上。使用 spring 启动，我们避免了之前我们必须做的所有样板代码和配置。因此，Spring Boot 可以帮助我们以最少的工作量，更加健壮地使用现有的 Spring 功能。

53. Spring Boot 有哪些优点？

减少开发，测试时间和努力。

使用 JavaConfig 有助于避免使用 XML。避免大量的 Maven 导入和各种版本冲突。提供意见发展方法。

通过提供默认值快速开始开发。

没有单独的 Web 服务器需要。这意味着你不再需要启动 Tomcat，Glassfish 或其他任何东西。需要更少的配置 因为没有 web.xml 文件。只需添加用 @ Configuration 注释的类，然后添加用 @Bean 注释的方法，Spring 将自动加载对象并像以前一样对其进行管理。您甚至可以将 @Autowired 添加到 bean 方法中，以使 Spring 自动装入需要的依赖关系中。

基于环境的配置 使用这些属性， 您可以将您正在使用的环境传递到应用程序：

-Dspring.profiles.active = {environment}。 在加载主应用程序属性文件后， Spring 将在（application{environment}.properties）中加载后续的应用程序属性文件。

54. 什么是 JavaConfig？

Spring JavaConfig 是 Spring 社区的产品，它提供了配置 Spring IoC 容器的纯 Java 方法。因此它有助于避免使用 XML 配置。使用 JavaConfig 的优点在于：

面向对象的配置。由于配置被定义为 JavaConfig 中的类，因此用户可以充分利用 Java 中的面向对象功能。一个配置类可以继承另一个，重写它的 @Bean 方法等。

减少或消除 XML 配置。基于依赖注入原则的外化配置的好处已被证明。但是，许多开发人员不希望在 XML 和 Java 之间来回切换。JavaConfig 为开发人员提供了一种纯 Java 方法来配置与 XML 配置概念相似的 Spring 容器。从技术角度来讲，只使用 JavaConfig 配置类来配置容器是可行的，但实际上很多人认为将 JavaConfig 与 XML 混合匹配是理想的。

类型安全和重构友好。JavaConfig 提供了一种类型安全的方法来配置 Spring 容器。由于 Java 5.0 对泛型的支持，现在可以按类型而不是按名称检索 bean，不需要任何强制转换或基于字符串的查找。

55. 如何重新加载 Spring Boot 上的更改，而无需重新启动服务器？

这可以使用 DEV 工具来实现。通过这种依赖关系，您可以节省任何更改，嵌入式 tomcat 将重新启动。Spring Boot 有一个开发工具（DevTools）模块，它有助于提高开发人员的生产力。Java 开发人员面临的一个主要挑战是将文件更改自动部署到服务器并自动重启服务器。开发人员可以重新加载 Spring Boot 上的更改，而无需重新启动服务器。这将消除每次手动部署

更改的需要。**Spring Boot** 在发布它的第一个版本时没有这个功能。这是开发人员最需要的功

能。DevTools 模块完全满足开发人员的需求。该模块将在生产环境中被禁用。它还提供 H2 数据库控制台以更好地测试应用程序。

56. Spring Boot 中的监视器是什么？

Spring boot actuator 是 spring 启动框架中的重要功能之一。Spring boot 监视器可帮助您访问生产环境中正在运行的应用程序的当前状态。有几个指标必须在生产环境中进行检查和监控。即使一些外部应用程序可能正在使用这些服务来向相关人员触发警报消息。监视器模块公开了一组可直接作为 HTTP URL 访问的 REST 端点来检查状态。

57. 如何在 Spring Boot 中禁用 Actuator 端点安全性？

默认情况下，所有敏感的 HTTP 端点都是安全的，只有具有 ACTUATOR 角色的用户才能访问它们。安全性是使用标准的 `HttpServletRequest.isUserInRole` 方法实施的。我们可以使用 `management.security.enabled = false` 来禁用安全性。只有在执行机构端点在防火墙后访问时，才建议禁用安全性。

59. 什么是 WebSocket？

WebSocket 是一种计算机通信协议，通过单个 TCP 连接提供全双工通信信道。

WebSocket 是双向的 - 使用 WebSocket 客户端或服务器可以发起消息发送。

WebSocket 是全双工的 - 客户端和服务器通信是相互独立的。

单个 TCP 连接 - 初始连接使用 HTTP，然后将此连接升级到基于套接字的连接。然后这个单一连接用于所有未来的通信

Light - 与 http 相比，WebSocket 消息数据交换要轻得多。

60. 什么是 Swagger？你用 Spring Boot 实现了它吗？

Swagger 广泛用于可视化 API，使用 Swagger UI 为前端开发人员提供在线沙箱。Swagger 是用于生成 RESTful Web 服务的可视化表示的工具，规范和完整框架实现。它使文档能够以与服务器相同的速度更新。当通过 Swagger 正确定义时，消费者可以使用最少量的实现逻辑来理解远程服务并与其进行交互。因此，Swagger 消除了调用服务时的猜测。

61. 什么是 Apache Kafka？

Apache Kafka 是一个分布式发布 - 订阅消息系统。它是一个可扩展的，容错的发布 - 订阅消息系统，它使我们能够构建分布式应用程序。这是一个 Apache 顶级项目。Kafka 适合离线和在线消息消费。

62. 什么是 Spring Cloud？

Spring cloud 流应用程序启动器是基于 Spring Boot 的 Spring 集成应用程序，提供与外部系统的集成。Spring cloud Task，一个生命周期短暂的微服务框架，用于快速构建执行有限数据处理的应用程序。

63. 使用 Spring Cloud 有什么优势？

使用 Spring Boot 开发分布式微服务时，我们面临以下问题

- 1、与分布式系统相关的复杂性-这种开销包括网络问题，延迟开销，带宽问题，安全问题。
- 2、服务发现-服务发现工具管理群集中的流程和服务如何查找和互相交谈。它涉及一个服务目录，在该目录中注册服务，然后能够查找并连接到该目录中的服务。
- 3、冗余-分布式系统中的冗余问题。
- 4、负载均衡 --负载均衡改善跨多个计算资源的工作负荷，诸如计算机，计算机集群，网络链路，中央处理单元，或磁盘驱动器的分布。
- 5、性能-问题 由于各种运营开销导致的性能问题。
- 6、部署复杂性-Devops 技能的要求。

64. 服务注册和发现是什么意思？Spring Cloud 如何实现？

当我们开始一个项目时，我们通常在属性文件中进行所有的配置。随着越来越多的服务开发和部署，添加和修改这些属性变得更加复杂。有些服务可能会下降，而某些位置可能会发生变化。手动更改属性可能会产生问题。Eureka 服务注册和发现可以在这种情况下提供帮助。由于所有服务都在 Eureka 服务器上注册并通过调用 Eureka 服务器完成查找，因此无需处理服务地点的任何更改和处理。

65. 什么是 Netflix Feign？它的优点是什么？

Feign 是受到 Retrofit, JAXRS-2.0 和 WebSocket 启发的 java 客户端联编程序。Feign 的第一个目标是将约束分母的复杂性统一到 http apis，而不考虑其稳定性。在 employee-consumer 的例子中，我们使用了 employee-producer 使用 REST 模板公开的 REST 服务。

但是我们必须编写大量代码才能执行以下步骤

- 使用功能区进行负载均衡。
- 获取服务实例，然后获取基本 URL。
- 利用 REST 模板来使用服务。前面的代码如下

```

@Controller
public class ConsumerControllerClient {
    @Autowired
    private LoadBalancerClient loadBalancer;
    public void getEmployee() throws RestClientException, IOException {
        ServiceInstance serviceInstance=loadBalancer.choose("employee-producer");
        System.out.println(serviceInstance.getUri());
        String baseUrl=serviceInstance.getUri().toString();
        baseUrl=baseUrl+"/employee";
        RestTemplate restTemplate = new RestTemplate();
        ResponseEntity<String> response=null;
        try{
            response=restTemplate.exchange(baseUrl,
                HttpMethod.GET, getHeaders(),String.class);
        }catch (Exception ex)
        {
            System.out.println(ex);
        }
        System.out.println(response.getBody());
    }
}

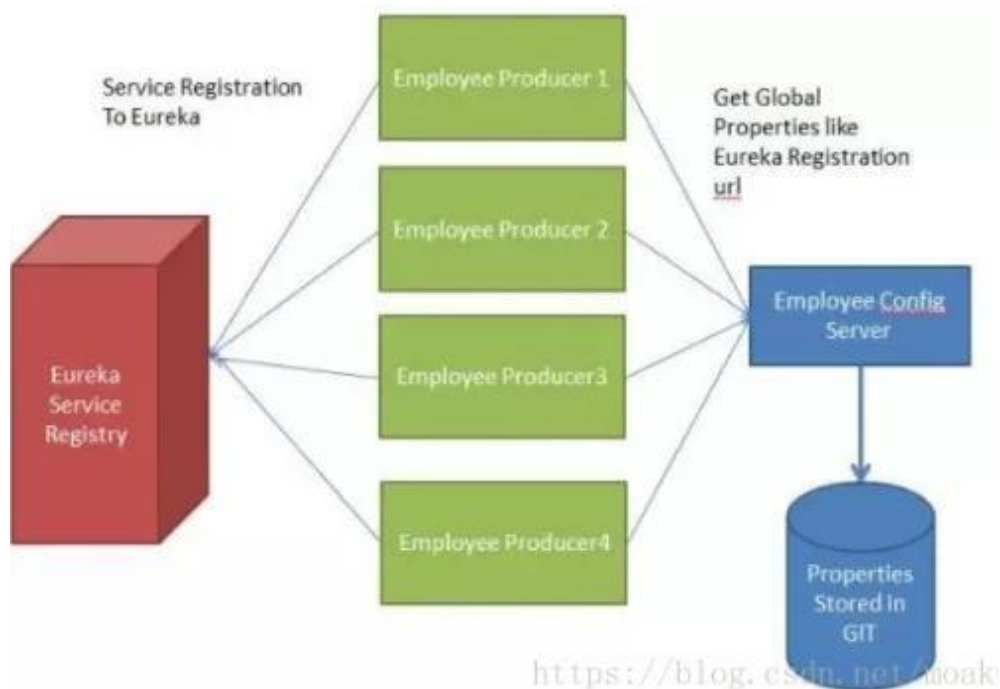
```

之前的代码，有像 NullPointerException 这样的例外的机会，并不是最优的。我们将看到如何使用 Netflix Feign 使呼叫变得更加轻松和清洁。如果 Netflix Ribbon 依赖关系也在类路径中，那么 Feign 默认也会负责负载均衡。

66. 什么是 Spring Cloud Bus? 我们需要它吗?

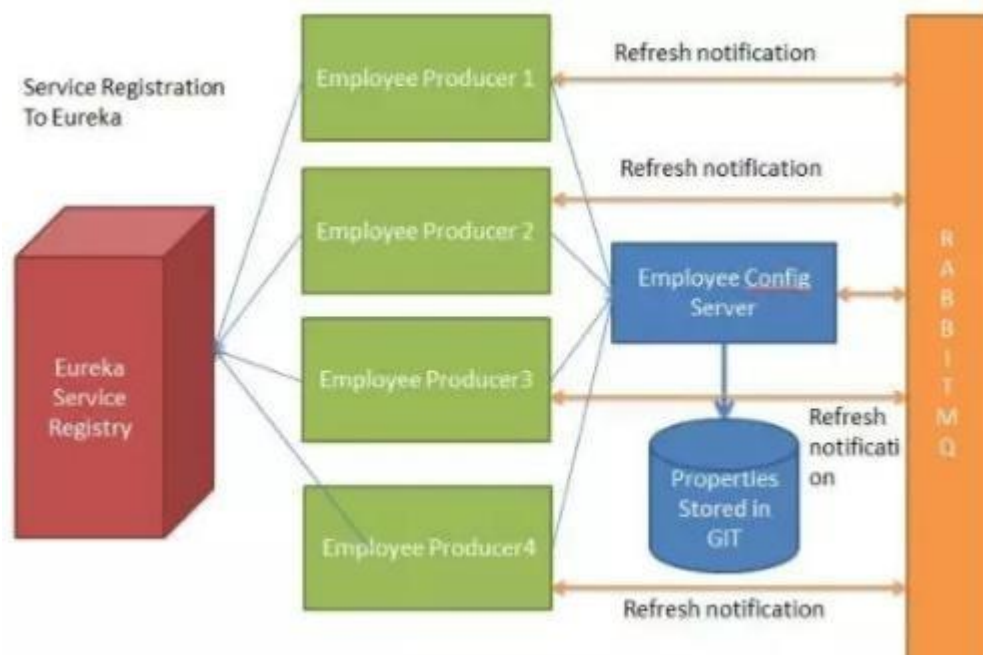
考虑以下情况：我们有多多个应用程序使用 Spring Cloud Config 读取属性，而 Spring Cloud Config 从 GIT 读取这些属性。

下面的例子中多个员工生产者模块从 Employee Config Module 获取 Eureka 注册的财产。



如果假设 GIT 中的 Eureka 注册属性更改为指向另一台 Eureka 服务器，会发生什么情况。在这种情况下，我们将不得不重新启动服务以获取更新的属性。

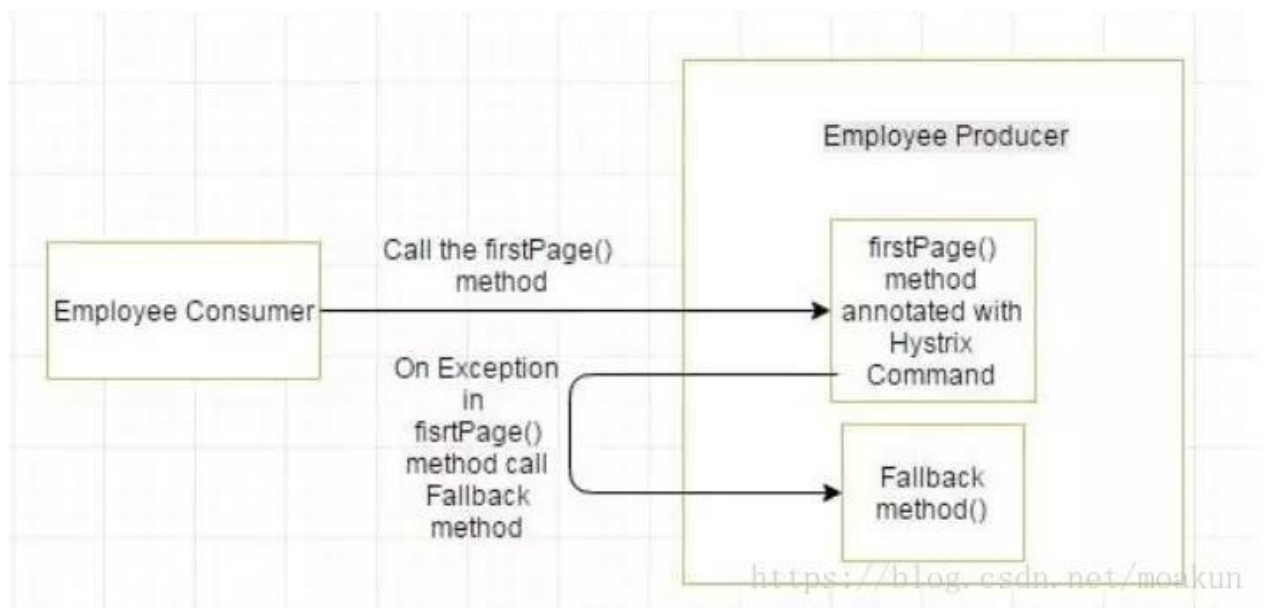
还有另一种使用执行器端点/刷新的方式。但是我们将不得不为每个模块单独调用这个 url。例如，如果 Employee Producer1 部署在端口 8080 上，则调用 `http://localhost:8080/refresh`。同样对于 Employee Producer2 `http://localhost:8081/refresh` 等等。这又很麻烦。这就是 Spring Cloud Bus 发挥作用的地方。



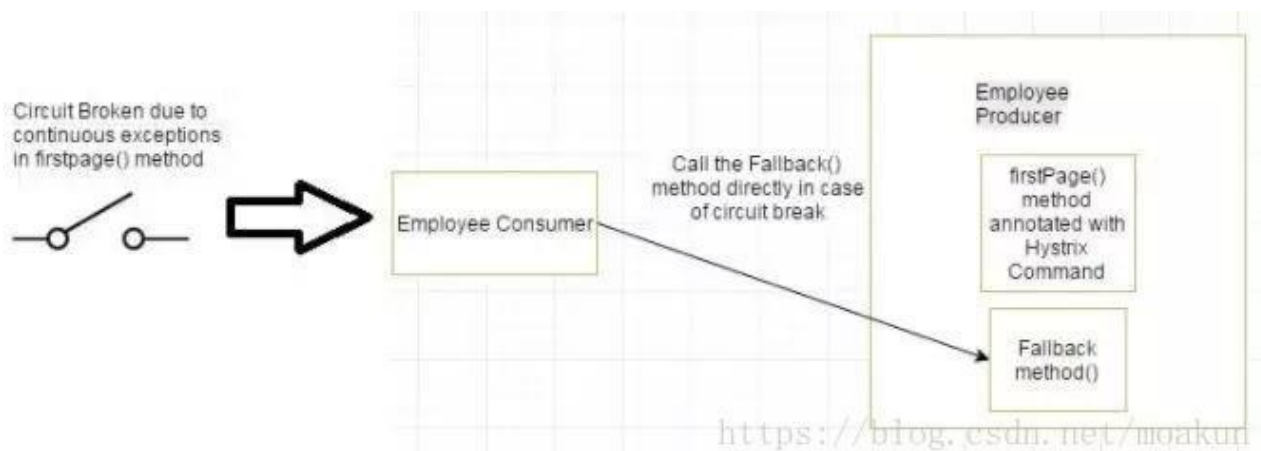
Spring Cloud Bus 提供了跨多个实例刷新配置的功能。因此，在上面的示例中，如果我们刷新 Employee Producer1，则会自动刷新所有其他必需的模块。如果我们有多个微服务启动并运行，这特别有用。这是通过将所有微服务连接到单个消息代理来实现的。无论何时刷新实例，此事件都会订阅到侦听此代理的所有微服务，并且它们也会刷新。可以通过使用端点/总线/刷新来实现对任何单个实例的刷新。

67. 什么是 Hystrix 断路器？我们需要它吗？

由于某些原因，employee-consumer 公开服务会引发异常。在这种情况下使用 Hystrix 我们定义了一个回退方法。如果在公开服务中发生异常，则回退方法返回一些默认值。



如果 firstPage method() 中的异常继续发生，则 Hystrix 电路将中断，并且员工使用者将一起跳过 firstPage 方法，并直接调用回退方法。断路器的目的是给第一页方法或第一页方法可能调用的其他方法留出时间，并导致异常恢复。可能发生的情况是，在负载较小的情况下，导致异常的问题有更好的恢复机会。



九、网络通信

1. http 协议的状态码有哪些？含义是什么？

200 OK 客户端请求成功

301 Moved Permanently (永久移除)，请求的 URL 已移走。Response 中应该包含一个 Location URL，说明资源现在所处的位置

302 found 重定向

400 Bad Request 客户端请求有语法错误，不能被服务器所理解

401Unauthorized 请求未经授权，这个状态代码必须和 WWW-Authenticate 报头域一起使用
403 Forbidden 服务器收到请求，但是拒绝提供服务
404 Not Found 请求资源不存在，eg: 输入了错误的 URL
500 Internal Server Error 服务器发生不可预期的错误
503 Server Unavailable 服务器当前不能处理客户端的请求，一段时间后可能恢复正常

2. Http 的请求报文组成?

请求行:

- 1、是请求方法，GET 和 POST 是最常见的HTTP 方法，除此以外还包括 DELETE、HEAD、OPTIONS、PUT、TRACE。
- 2、为请求对应的 URL 地址，它和报文头的 Host 属性组成完整的请求 URL。
- 3、是协议名称及版本号。

请求头:

是 HTTP 的报文头，报文头包含若干个属性，格式为“属性名:属性值”，服务端据此获取客户端的信息。

与缓存相关的规则信息，均包含在 header 中

请求体:

是报文体，它将一个页面表单中的组件值通过 param1=value1¶m2=value2 的键值对形式编码成一个格式化串，它承载多个请求参数的数据。不但报文体可以传递请求参数，请求 URL 也可以通过类似于“/chapter15/user.html? param1=value1¶m2=value2”的方式传递请求参数。

3. 一次完整的 Http 请求是怎样的?

域名解析 --> 发起 TCP 的 3 次握手 --> 建立 TCP 连接后发起 http 请求 --> 服务器响应 http 请求，浏览器得到 html 代码 --> 浏览器解析 html 代码，并请求 html 代码中的资源（如 js、css、图片等） --> 浏览器对页面进行渲染呈现给用户

4. Tcp 和Udp 的区别?

1. 基于连接与无连接;
2. 对系统资源的要求（TCP 较多，UDP 少）;
3. UDP 程序结构较简单;
4. 流模式与数据报模式；
5. TCP 保证数据正确性，UDP 可能丢包，TCP 保证数据顺序，UDP 不保证。

5. SSL 协议的三个特性

私密性：在握手协议定义了会话密钥后，所有的消息都被加密。

确认性：尽管会话的客户端认证是可选的，但是服务器端始终是被认证的。

可靠性：传送的消息包括消息完整性检查。

6. TCP 的三次握手与四次挥手

第一次握手：建立连接时，客户端发送 syn 包（syn=x）到服务器，并进入 SYN_SENT 状态，等待服务器确认；SYN：同步序列编号（Synchronize Sequence Numbers）。

第二次握手：服务器收到 syn 包，必须确认客户的 SYN（ack=x+1），同时自己也发送一个 SYN 包（syn=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(ack=y+1)，此包发送完毕，客户端和服务器进入 ESTABLISHED（TCP 连接成功）状态，完成三次握手。

1) 客户端进程发出连接释放报文，并且停止发送数据。释放数据报文首部，FIN=1，其序列号为 seq=u（等于前面已经传送过来的数据的最后一个字节的序号加 1），此时，客户端进入 FIN-WAIT-1（终止等待 1）状态。TCP 规定，FIN 报文段即使不携带数据，也要消耗一个序号。

2) 服务器收到连接释放报文，发出确认报文，ACK=1，ack=u+1，并且带上自己的序列号 seq=v，此时，服务端就进入了 CLOSE-WAIT（关闭等待）状态。TCP 服务器通知高层的应用进程，客户端向服务器的方向就释放了，这时候处于半关闭状态，即客户端已经没有数据要发送了，但是服务器若发送数据，客户端依然要接受。这个状态还要持续一段时间，也就是整个 CLOSE-WAIT 状态持续的时间。

3) 客户端收到服务器的确认请求后，此时，客户端就进入 FIN-WAIT-2（终止等待 2）状态，等待服务器发送连接释放报文（在这之前还需要接受服务器发送的最后的的数据）。

4) 服务器将最后的数据发送完毕后，就向客户端发送连接释放报文，FIN=1，ack=u+1，由于在半关闭状态，服务器很可能又发送了一些数据，假定此时的序列号为 seq=w，此时，服务器就进入了 LAST-ACK（最后确认）状态，等待客户端的确认。

5) 客户端收到服务器的连接释放报文后，必须发出确认，ACK=1，ack=w+1，而自己的序列号是 seq=u+1，此时，客户端就进入了 TIME-WAIT（时间等待）状态。注意此时 TCP 连接还没有释放，必须经过 2 * MSL（最长报文段寿命）的时间后，当客户端撤销相应的 TCB 后，才进入 CLOSED 状态。

6) 服务器只要收到了客户端发出的确认，立即进入 CLOSED 状态。同样，撤销 TCB 后，就结束了这次的 TCP 连接。可以看到，服务器结束 TCP 连接的时间要比客户端早一些。

7. 为什么连接的时候是三次握手，关闭的时候却是四次握手？

因为当 Server 端收到 Client 端的 SYN 连接请求报文后，可以直接发送 SYN+ACK 报文。其中 ACK 报文是用来应答的，SYN 报文是用来同步的。但是关闭连接时，当 Server 端收到 FIN 报文时，很可能并不会立即关闭 SOCKET，所以只能先回复一个 ACK 报文，告诉 Client 端，“你发的 FIN 报文我收到了”。只有等到我 Server 端所有的报文都发送完了，我才能发送 FIN 报文，因此不能一起发送。故需要四次握手。

8. 如果已经建立了连接，但是客户端突然出现故障了怎么办？

TCP 还设有一个保活计时器，显然，客户端如果出现故障，服务器不能一直等下去，白白浪费资源。服务器每收到一次客户端的请求后都会重新复位这个计时器，时间通常是设置为 2 小时，若两小时还没有收到客户端的任何数据，服务器就会发送一个探测报文段，以后每隔 75 秒钟发送一次。若一连发送 10 个探测报文仍然没反应，服务器就认为客户端出了故障，接着就关闭连接。

9. http 中重定向和请求转发的区别？

本质区别：转发是服务器行为，重定向是客户端行为。

重定向特点：两次请求，浏览器地址发生变化，可以访问自己 web 之外的资源，传输的数据会丢失。

请求转发特点：一次强求，浏览器地址不变，访问的是自己本身的 web 资源，传输的数据不会丢失。

10. GET 和 POST 的区别？

1、GET 请求的数据会附在 URL 之后（就是把数据放置在 HTTP 协议头中），以?分割 URL 和传输数据，参数之间以&相连，如：login.action?name=zhagnsan&password=123456。POST 把提交的数据则放置是在 HTTP 包的包体中。

2、GET 方式提交的数据最多只能是 1024 字节，理论上 POST 没有限制，可传较大量的数据。其实这样说是错误的，不准确的：“GET 方式提交的数据最多只能是 1024 字节”，因为 GET 是通过 URL 提交数据，那么 GET 可提交的数据量就跟 URL 的长度有直接关系了。而实际上，URL 不存在参数上限的问题，HTTP 协议规范没有对 URL 长度进行限制。这个限制是特定的浏览器及服务器对它的限制。IE 对 URL 长度的限制是 2083 字节(2K+35)。对于其他浏览器，如 Netscape、FireFox 等，理论上没有长度限制，其限制取决于操作系统的支持

3、POST 的安全性要比 GET 的安全性高。注意：这里所说的安全性和上面 GET 提到的“安全”不是同个概念。上面“安全”的含义仅仅是不作数据修改，而这里安全的含义是真正的 Security 的含义，比如：通过 GET 提交数据，用户名和密码将明文出现在 URL 上，因为登录页面有可能被浏览器缓存，其他人查看浏览器的历史纪录，那么别人就可以拿到你的账号和密码了，除此之外，使用 GET 提交数据还可能会造成 Cross-site request forgery 攻击。

4、Get 是向服务器发索取数据的一种请求，而 Post 是向服务器提交数据的一种请求，在 FORM（表单）中，Method 默认为"GET"，实质上 GET 和 POST 只是发送机制不同，并不是一个取一个发。

6. Http 与 Https 的区别

- 1、HTTP 的 URL 以 http:// 开头，而 HTTPS 的 URL 以 https:// 开头
- 2、HTTP 是不安全的，而 HTTPS 是安全的
- 3、HTTP 标准端口是 80，而 HTTPS 的标准端口是 443
- 4、在 OSI 网络模型中，HTTP 工作于应用层，而 HTTPS 的安全传输机制工作在传输层
- 5、HTTP 无法加密，而 HTTPS 对传输的数据进行加密
- 6、HTTP 无需证书，而 HTTPS 需要 CA 机构 wosign 的颁发的 SSL 证书

12. 什么是 Http 协议无状态协议?怎么解决 Http 协议无状态协议?

无状态协议对于事务处理没有记忆能力。缺少状态意味着如果后续处理需要前面的信息也就是说，当客户端一次 HTTP 请求完成以后，客户端再发送一次 HTTP 请求，HTTP 并不知道当前客户端是一个“老用户”。

可以使用 Cookie 来解决无状态的问题，Cookie 就相当于一个通行证，第一次访问的时候给客户端发送一个 Cookie，当客户端再次来的时候，拿着 Cookie(通行证)，那么服务器就

知道这个是“老用户”。

13. HTTPS 工作原理

一、首先 HTTP 请求服务端生成证书，客户端对证书的有效期、合法性、域名是否与请求的域名一致、证书的公钥（RSA 加密）等进行校验；

二、客户端如果校验通过后，就根据证书的公钥的有效，生成随机数，随机数使用公钥进行加密（RSA 加密）；

三、消息体产生的后，对它的摘要进行 MD5（或者 SHA1）算法加密，此时就得到了 RSA 签名；

四、发送给服务端，此时只有服务端（RSA 私钥）能解密。

五、解密得到的随机数，再用 AES 加密，作为密钥（此时的密钥只有客户端和服务端知道）。

十、数据库

1. MySQL 的存储引擎有哪些，区别是什么

MySQL 常见的三种存储引擎为 InnoDB、MyISAM 和 MEMORY。

1、事务安全：

InnoDB 支持事务安全，MyISAM 和 MEMORY 两个不支持。

2、存储限制：

InnoDB 有 64TB 的存储限制，MyISAM 和 MEMORY 要是具体情况而定。

3、空间使用：

InnoDB 对空间使用程度较高，MyISAM 和 MEMORY 对空间使用程度较低。

4、内存使用：

InnoDB 和 MEMORY 对内存使用程度较高，MyISAM 对内存使用程度较低。

5、插入数据的速度：

InnoDB 插入数据的速度较低，MyISAM 和 MEMORY 插入数据的速度较高。

6、对外键的支持：

InnoDB 对外键支持情况较好，MyISAM 和 MEMORY 两个不支持外键。

2. 触发器的作用？

触发器是一种特殊的存储过程，主要是通过事件来触发而被执行的。它可以强化约束，来维护数据的完整性和一致性，可以跟踪数据库内的操作从而不允许未经许可的更新和变化。可以联级运算。如，某表上的触发器上包含对另一个表的数据操作，而该操作又会导致该表触发器被触发。

3. 什么是存储过程？用什么来调用？

存储过程是一个预编译的 SQL 语句，优点是允许模块化的设计，就是说只需创建一次，以后在该程序中就可以调用多次。如果某次操作需要执行多次 SQL，使用存储过程比单纯 SQL 语句执行要快。

调用：

1) 可以用一个命令对象来调用存储过程。

2) 可以供外部程序调用，比如：java 程序。

4. 存储过程的优缺点。

优点：

- 1) 存储过程是预编译过的，执行效率高。
- 2) 存储过程的代码直接存放于数据库中，通过存储过程名直接调用，减少网络通讯。
- 3) 安全性高，执行存储过程需要有一定权限的用户。
- 4) 存储过程可以重复使用，可减少数据库开发人员的工作量。

缺点：移植性差

5. SQL 优化的具体操作

1、尽量避免使用 `select *`，返回无用的字段会降低查询效率。

优化方式：使用具体的字段代替`*`，只返回使用到的字段。

2、尽量避免使用 `in` 和 `not in`，会导致数据库引擎放弃索引进行全表扫描。

优化方式：如果是连续数值，可以用 `between` 代替。如果是子查询，可以用 `exists` 代替。

3、尽量避免在字段开头模糊查询，会导致数据库引擎放弃索引进行全表扫描。

优化方式：尽量在字段后面使用模糊查询。

4、尽量避免进行 `null` 值的判断，会导致数据库引擎放弃索引进行全表扫描。

优化方式：可以给字段添加默认值 `0`，对 `0` 值进行判断。

6. 什么叫视图？游标是什么？

视图：

是一种虚拟的表，具有和物理表相同的功能。可以对视图进行增，改，查，操作，试图 通常是有有一个表或者多个表的行或列的子集。对视图的修改会影响基本表。它使得我们获取 数据更容易，相比多表查询。

游标：

是对查询出来的结果集作为一个单元来有效的处理。游标可以定在该单元中的特定行， 从结果集的当前行检索一行或多行。可以对结果集当前行做修改。一般不使用游标，但是需要逐条处理数据的时候，游标显得十分重要。

7. 视图的优缺点

优点：

- 1 对数据库的访问，因为视图可以有选择性的选取数据库里的一部分。
- 2) 用户通过简单的查询可以从复杂查询中得到结果。
- 3) 维护数据的独立性，试图可从多个表检索数据。
- 4) 对于相同的数据可产生不同的视图。

缺点：

性能：查询视图时，必须把视图的查询转化成对基本表的查询，如果这个视图是由一个复杂的多表查询所定义，那么，那么就无法更改数据

8. 事务的四个特性

原子性 (Atomicity) :

原子性是指事务包含的所有操作要么全部成功, 要么全部失败回滚, 因此事务的操作如果成功就必须完全应用到数据库, 如果操作失败则不能对数据库有任何影响。

一致性 (Consistency) :

事务开始前和结束后, 数据库的完整性约束没有被破坏。比如 A 向 B 转账, 不可能 A 扣了钱, B 却没收到。

隔离性 (Isolation) :

隔离性是当多个用户并发访问数据库时, 比如操作同一张表时, 数据库为每一个用户开启的事务, 不能被其他事务的操作所干扰, 多个并发事务之间要相互隔离。同一时间, 只允许一个事务请求同一数据, 不同的事务之间彼此没有任何干扰。比如 A 正在从一张银行卡中取钱, 在 A 取钱的过程结束前, B 不能向这张卡转账。

持久性 (Durability) :

持久性是指一个事务一旦被提交了, 那么对数据库中的数据的改变就是永久性的, 即便是在数据库系统遇到故障的情况下也不会丢失提交事务的操作。

9. 数据库乐观锁, 悲观锁的区别, 怎么实现

悲观锁 (Pessimistic Lock), 顾名思义, 就是很悲观, 每次去拿数据的时候都认为别人会修改, 所以每次在拿数据的时候都会上锁, 这样别人想拿这个数据就会阻塞挂起直到它拿到锁。传统的关系型数据库里边就用到了很多这种锁机制, 比如行锁, 表锁, 读锁, 写锁等, 都是在做操作之前先上锁。

乐观锁 (Optimistic Lock), 顾名思义, 就是很乐观, 每次去拿数据的时候都认为别人不会修改数据, 所以不会上锁, 但是在更新的时候会判断一下在此期间别人有没有去更新这个数据, 乐观锁适用于多读的应用类型, 这样可以提高吞吐量。

两种锁各有优缺点, 乐观锁适用于写比较少的情况下, 即冲突真的很少发生的时候, 这样就可以省去锁的开销, 加大系统的整个吞吐量。但如果经常产生冲突, 上层应用会不断的进行 retry, 这样反倒是降低了性能, 所以这种情况下用悲观锁就比较合适。

10. 事务的并发问题

1、脏读: 事务 A 读取了事务 B 更新的数据, 然后 B 回滚操作, 那么 A 读取到的数据是脏数据

2、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, 对数据作了更新并提交, 导致事务 A 多次读取同一数据时, 结果因此本事务先后两次读到的数据结果会不一致。

3、幻读: 幻读解决了不重复读, 保证了同一个事务里, 查询的结果都是事务开始时的状态 (一致性)。

11. MySQL 的 MyISAM 与 InnoDB 两种存储引擎在, 事务、锁级别, 各自的适用场景?

事务处理上方面

MyISAM: 强调的是性能, 每次查询具有原子性, 其执行速度比 InnoDB 类型更快, 但是不提供事务支持。

InnoDB: 提供事务支持事务, 外部键等高级数据库功能。具有事务(commit)、回滚(rollback) 和崩溃修复能力 (crash recovery capabilities) 的事务安全 (transaction-safe)

(ACID compliant))型表。

锁级别

MyISAM: 只支持表级锁，用户在操作 **MyISAM** 表时，**select**，**update**，**delete**，**insert** 语句都会给表自动加锁，如果加锁以后的表满足 **insert** 并发的情况下，可以在表的尾部插入新的数据。

InnoDB: 支持事务和行级锁，是 **innodb** 的最大特色。行锁大幅度提高了多用户并发操作的新能。但是 **InnoDB** 的行锁，只是在 **WHERE** 的主键是有效的，非主键的 **WHERE** 都会锁全表的。

12. 非关系型数据库和关系型数据库区别，优势比较

非关系型数据库的优势：

性能：**NOSQL** 是基于键值对的，可以想象成表中的主键和值的对应关系，而且不需要经过 **SQL** 层的解析，所以性能非常高。

可扩展性：同样也是因为基于键值对，数据之间没有耦合性，所以非常容易水平扩展。关系型数据库的优势：

复杂查询：可以用 **SQL** 语句方便的在一个表以及多个表之间做非常复杂的数据查询。

事务支持：使得对于安全性能很高的数据访问要求得以实现。

13. 数据库的五大范式

第一范式:(确保每列保持原子性)所有字段值都是不可分解的原子值。

第一范式是最基本的范式。如果数据库表中的所有字段值都是不可分解的原子值，就说明该数据库表满足了第一范式。

第一范式的合理遵循需要根据系统的实际需求来定。比如某些数据库系统中需要用到“地址”这个属性，本来直接将“地址”属性设计成一个数据库表的字段就行。但是如果系统经常会访问“地址”属性中的“城市”部分，那么就非要将“地址”这个属性重新拆分为省份、城市、详细地址等多个部分进行存储，这样在对地址中某一部分操作的时候将非常方便。这样设计才算满足了数据库的第一范式，如下表所示。

上表所示的用户信息遵循了第一范式的要求，这样在对用户使用城市进行分类的时候就非常方便，也提高了数据库的性能。

第二范式:(确保表中的每列都和主键相关)在一个数据库表中，一个表中只能保存一种数据，不可以把多种数据保存在同一张数据库表中。

第二范式在第一范式的基础之上更进一层。第二范式需要确保数据库表中的每一列都和主键相关，而不能只与主键的某一部分相关（主要针对联合主键而言）。也就是说在一个数据库表中，一个表中只能保存一种数据，不可以把多种数据保存在同一张数据库表中。

比如要设计一个订单信息表，因为订单中可能会有多种商品，所以要将订单编号和商品编号作为数据库表的联合主键。

第三范式:(确保每列都和主键列直接相关,而不是间接相关)数据表中的每一列数据都和主键直接相关，而不能间接相关。

第三范式需要确保数据表中的每一列数据都和主键直接相关，而不能间接相关。

比如在设计一个订单数据表的时候，可以将客户编号作为一个外键和订单表建立相应的关系。而不可以再在订单表中添加关于客户其它信息（比如姓名、所属公司等）的字段。

BCNF:符合 **3NF**，并且，主属性不依赖于主属性。

若关系模式属于第二范式，且每个属性都不传递依赖于键码，则 **R** 属于 **BC** 范式。

通常 BC 范式的条件有多种等价的表述：每个非平凡依赖的左边必须包含键码；每个决定因素必须包含键码。

BC 范式既检查非主属性，又检查主属性。当只检查非主属性时，就成了第三范式。满足 BC 范式的关系都必然满足第三范式。

还可以这么说：若一个关系达到了第三范式，并且它只有一个候选码，或者它的每个候选码都是单属性，则该关系自然达到 BC 范式。

一般，一个数据库设计符合 3NF 或 BCNF 就可以了。

第四范式:要求把同一表内的多对多关系删除。

第五范式:从最终结构重新建立原始结构。

14. 什么是内连接、外连接、交叉连结、笛卡尔积等？

内连接: 只连接匹配的行

左外连接: 包含左边表的全部行（不管右边的表中是否存在与它们匹配的行），以及右边表中全部匹配的行

右外连接: 包含右边表的全部行（不管左边的表中是否存在与它们匹配的行），以及左边表中全部匹配的行

例如 1:

```
SELECT a,b. FROM luntan LEFT JOIN usertable as b ON a.username=b.username
```

例如 2:

```
SELECT a,b. FROM city as a FULL OUTER JOIN user as b ON a.username=b.username
```

全外连接: 包含左、右两个表的全部行，不管另外一边的表中是否存在与它们匹配的行。交叉连接: 生成笛卡尔积—它不使用任何匹配或者选取条件，而是直接将一个数据源中的

每个行与另一个数据源的每个行都一一匹配

15. SQL 语言分类

数据查询语言 DQL

数据操纵语言 DML

数据定义语言 DDL

数据控制语言 DCL。

16. count(*)、count(1)、count(column)的区别

count(*)对行的数目进行计算,包含 NULL

count(column)对特定的列的值具有的行数进行计算,不包含 NULL 值。

count()还有一种使用方式,count(1)这个用法和 count(*)的结果是一样的。

17. 什么是索引？

数据库索引，是数据库管理系统中一个排序的数据结构，索引的实现通常使用 B 树及其变种 B+树。

在数据之外，数据库系统还维护着满足特定查找算法的数据结构，这些数据结构以某种方式引用（指向）数据，这样就可以在这些数据结构上实现高级查找算法。这种数据结构，就是索引。

18. 索引的作用？

协助快速查询、更新数据库表中数据。

为表设置索引要付出代价的：

一是增加了数据库的存储空间

二是在插入和修改数据时要花费较多的时间(因为索引也要随之变动)。

19. 索引的优缺点

创建索引可以大大提高系统的性能（优点）：

1.通过创建唯一性索引，可以保证数据库表中每一行数据的唯一性。

2.可以大大加快数据的检索速度，这也是创建索引的最主要的原因。

3.可以加速表和表之间的连接，特别是在实现数据的参考完整性方面特别有意义。

4.在使用分组和排序子句进行数据检索时，同样可以显著减少查询中分组和排序的时间。

5.通过使用索引，可以在查询的过程中，使用优化隐藏器，提高系统的性能。

增加索引也有许多不利的方面(缺点)：

1.创建索引和维护索引要耗费时间，这种时间随着数据量的增加而增加。

2.索引需要占物理空间，除了数据表占数据空间之外，每一个索引还要占一定的物理空间，如果要建立聚簇索引，那么需要的空间就会更大。

3.当对表中的数据进行增加、删除和修改的时候，索引也要动态的维护，这样就降低了数据的维护速度。

20. 什么样的字段适合建索引？

唯一、不为空、经常被查询的字段

21. Hash 索引和 B+树索引的区别？

hash 索引，等值查询效率高，不能排序,不能进行范围查询；B+树数据有序,范围查询

22. MySQL 三种锁的级别

表级锁：开销小，加锁快；不会出现死锁；锁定粒度大，发生锁冲突的概率最高,并发度最低。

行级锁：开销大，加锁慢；会出现死锁；锁定粒度最小，发生锁冲突的概率最低,并发度也最高。

页面锁：开销和加锁时间界于表锁和行锁之间；会出现死锁；锁定粒度界于表锁和行锁 之间，并发度一般

23. 为什么不都用 Hash 索引而使用 B+树索引？

索引查找过程中就要产生磁盘 I/O 消耗,主要看 IO 次数，和磁盘存取原理有关。 根据 B-Tree 的定义，可知检索一次最多需要访问 h 个节点。数据库系统的设计者巧妙利用了磁盘预读原理， 将一个节点的大小设为等于一个页，这样每个节点只需要一次 I/O 就可以完全

24. B 树和 B+树的区别

1、树，每个节点都存储 key 和 data，所有节点组成这棵树，并且叶子节点指针为 nul，叶子结点不包含任何关键字信息。

2、B+树，所有的叶子结点中包含了全部关键字的信息，及指向含有这些关键字记录的指针，且叶子结点本身依关键字的大小自小而大的顺序链接，所有的非终端结点可以看成是索引部分，结点中仅含有其子树根结点中最大（或最小）关键字。（而 B 树的非终端节点也包含需要查找的有效信息）

25. 为什么说 B+比 B 树更适合实际应用中操作系统的文件索引和数据库索引？

1.B+的磁盘读写代价更低

B+的内部结点并没有指向关键字具体信息的指针。因此其内部结点相对 B 树更小。如果把所有同一内部结点的关键字存放在同一盘块中，那么盘块所能容纳的关键字数量也越多。一次性读入内存中的需要查找的关键字也就越多。相对来说 IO 读写次数也就降低了。

2.B+tree 的查询效率更加稳定

由于非终结点并不是最终指向文件内容的结点，而只是叶子结点中关键字的索引。所以任何关键字的查找必须走一条从根结点到叶子结点的路。所有关键字查询的路径长度相同，导致每一个数据的查询效率相当。

26. 聚集索引和非聚集索引区别？

聚合索引(clustered index):

聚集索引表记录的排列顺序和索引的排列顺序一致，所以查询效率高，只要找到第一个索引值记录，其余就连续性的记录在物理也一样连续存放。聚集索引对应的缺点就是修改慢，因为为了保证表中记录的物理和索引顺序一致，在记录插入的时候，会对数据页重新排序。聚集索引类似于新华字典中用拼音去查找汉字，拼音检索表于书记顺序都是按照 a~z 排列的，就像相同的逻辑顺序于物理顺序一样，当你需要查找 a,ai 两个读音的字，或是想一次寻找多个傻(sha)的同音字时，也许向后翻几页，或紧接着下一行就得到结果了。

非聚合索引(nonclustered index):

非聚集索引指定了表中记录的逻辑顺序，但是记录的物理和索引不一定一致，两种索引都采用 B+树结构，非聚集索引的叶子层并不和实际数据页相重叠，而采用叶子层包含一个指向表中的记录在数据页中的指针方式。非聚集索引层次多，不会造成数据重排。非聚集索引类似在新华字典上通过偏旁部首来查询汉字，检索表也许是按照横、竖、撇来排列的，但是由于正文中是 a~z 的拼音顺序，所以就类似于逻辑地址于物理地址的不对应。同时适用的情况就在于分组，大数目的不同值，频繁更新的列中，这些情况即不适合聚集索引。

根本区别：

聚集索引和非聚集索引的根本区别是表记录的排列顺序和与索引的排列顺序是否一致。

十一、算法

1. 冒泡排序

```

public static void sort() {
    Scanner input = new Scanner(System.in);
    int sort[] = new int[10];
    int temp;
    System.out.println("请输入10个排序的数据: ");
    for (int i = 0; i < sort.length; i++) {
        sort[i] = input.nextInt();
    }
    for (int i = 0; i < sort.length - 1; i++) {
        for (int j = 0; j < sort.length - i - 1; j++) {
            if (sort[j] < sort[j + 1]) {
                temp = sort[j];
                sort[j] = sort[j + 1];
                sort[j + 1] = temp;
            }
        }
    }
    System.out.println("排列后的顺序为: ");
    for(int i=0;i<sort.length;i++){
        System.out.print(sort[i]+" ");
    }
}
public static void main(String[] args) {
    sort();
}
}

```

```
public class Sort {
```

2. 两个有序数组的合并排序

```
public class MyClass {
    public static void main(String[] args) {
        int[] num1 = new int[]{1, 2, 4, 6, 7, 123, 411, 5334, 1414141, 1314141414};
        int[] num2 = new int[]{0, 2, 5, 7, 89, 113, 5623, 6353, 134134};
        //变量用于存储两个集合应该被比较的索引（存入新集合就加一）
        int a = 0;
        int b = 0;
        int[] num3 = new int[num1.length + num2.length];
        for (int i = 0; i < num3.length; i++) {
            if (a < num1.length && b < num2.length) { //两数组都未遍历完，相互比较后加入
                if (num1[a] > num2[b]) {
                    num3[i] = num2[b];
                    b++;
                } else {
                    num3[i] = num1[a];
                    a++;
                }
            } else if (a < num1.length) { //num2已经遍历完，无需比较，直接将剩余num1加入
                num3[i] = num1[a];
                a++;
            } else if (b < num2.length) { //num1已经遍历完，无需比较，直接将剩余num2加入
                num3[i] = num2[b];
                b++;
            }
        }
        System.out.println("排序后:" + Arrays.toString(num3));
    }
}
```

3. 一个数组的倒序

```
public static int[] reverse(int[] a) {
    int[] b=a;
    for(int start=0,end=b.length-1;start<end;start++,end--) {
        int temp=b[start];
        b[start]=b[end];
        b[end]=temp;
    }
    return b;
}
```

4. 计算一个正整数的正平方根

```

public static double MySqrt(int value, double t){
    if (value < 0 || t<0)
        return 0;
    double left = 0;
    double right = value;
    double mid = (right + left) / 2;
    double offset = 2*t ;
    while (offset>t){
        double temp = mid*mid;
        if (temp > value){
            right = (left + right) / 2;
            offset = temp - value;
        }
        if (temp <= value){
            left = (left + right) / 2;
            offset = value - temp;
        }
        mid = (left + right) / 2;
    }
    return mid;
}

```

5. 快速排序算法？


```

public class QuickTest {
    public static void sort(int[] a){
        if (a.length>0) {
            sort(a,0,a.length-1);
        }
    }

    public static void sort(int[] a,int low,int height){
        int i=low;
        int j=height;
        if (i>j) { //放在k之前，防止下标越界
            return;
        }
        int k=a[i];

        while (i<j) {
            while (i<j&& a[j]>k) { //找出小的数
                j--;
            }
            while (i<j&& a[i]<=k) { //找出大的数
                i++;
            }
            if (i<j){ //交换
                int swap=a[i];
                a[i]=a[j];
                a[j]=swap;
            }
        }
        //交换K
        k=a[i];
        a[i]=a[low];
        a[low]=k;

        //对左边进行排序,递归算法
        sort(a, low, i-1);
        //对右边进行排序
        sort(a, i+1,height);
    }
}

```

6. 二叉树的遍历算法

```

public class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;

    TreeNode(int x) {
        val = x;
    }
}

// 用递归的方法进行先序遍历
public void qinaxuDigui(TreeNode treeNode) {
    qianxuNumList.add(treeNode.val);
    if (treeNode.left != null) {
        qinaxuDigui(treeNode.left);
    }
    if (treeNode.right != null) {
        qinaxuDigui(treeNode.right);
    }
}

// 用递归的方法进行中序遍历
public void zhongxuDigui(TreeNode treeNode) {
    if (treeNode.left != null) {
        zhongxuDigui(treeNode.left);
    }
    zhongxuNumList.add(treeNode.val);
    if (treeNode.right != null) {
        zhongxuDigui(treeNode.right);
    }
}

// 用递归的方法进行后序遍历
public void houxuDigui(TreeNode treeNode) {
    if (treeNode.left != null) {
        houxuDigui(treeNode.left);
    }
    if (treeNode.right != null) {
        houxuDigui(treeNode.right);
    }
    houxuNumList.add(treeNode.val);
}
}

```

6. DFS, BFS 算法

DFS (深度优先遍历)

BFS (广度优先遍历)

```

public class Graph {
    //A, B, C,D, E, F
    static int[][] graph = new int[][]{{0, 0, 1, 1, 0, 0},
        {0, 0, 1, 0, 0, 0},
        {1, 1, 0, 0, 0, 0},
        {0, 0, 1, 0, 1, 0},
        {0, 0, 0, 1, 0, 1},
        {0, 0, 0, 0, 1, 0}};
    int[] help = new int[graph.length]; //用来记录已经遍历过的元素

    //DFS(深度优先遍历)同样适用于有向图 A->C->B->D->E->F 即 0->2->1->3->4->5
    public void dfsTraversing(int node, int[][] graph) {
        help[node]=1;
        System.out.println(node);
        for (int i = 0; i < graph[node].length; ++i) {
            if (help[i]==0&&i != node&&graph[node][i]==1) {
                dfsTraversing(i, graph);
            }
        }
    }

    //BFS(广度优先遍历)同样适用于有向图 A->C->D->B->E->F 即0->2->3->1->4->5
    public void bfsTraversing(int[][] graph) {
        int[] queue=new int[graph.length];
        int cnt=1;
        queue[0]=0; //将A作为起始顶点加入队列
        help[0]=1;
        System.out.println(0);
        for (int i=0;i<cnt;++i){
            for (int j=0;j<graph[queue[i]].length;++j){
                if (queue[i]!=j&&graph[queue[i]][j]==1&&help[j]==0){
                    help[queue[i]]=1;
                    queue[cnt++]=j;
                    System.out.println(j);
                }
            }
        }
    }
}

```

7. 时间类型转换。

```
public class DateFormat {  
    public static void fun() {  
        SimpleDateFormat sdf = new SimpleDateFormat("yyyy年MM月dd日");  
        String newDate;  
        try {  
            newDate = sdf.format(new SimpleDateFormat("yyyyMMdd")  
                .parse("20121115"));  
            System.out.println(newDate);  
        } catch (ParseException e) {  
            e.printStackTrace();  
        }  
    }  
    public static void main(String args[]) {  
        fun();  
    }  
}
```

8. 逆波兰计算器

```

public class PolandNotation {
    public static void main(String[] args) {
        //4*5-8+60+8/2
        String expression = "4 5 * 8 - 60 + 8 2 / +";
        List<String> list = getStrList(expression);
        System.out.println(list);
        //计算值: 得结果
        int res = calc(list);
        System.out.println(res);
    }
    public static List<String> getStrList(String exp){
        String arr[] = exp.split(" "); //将字符串遍历得到数组
        List<String> list = new ArrayList<>();
        for(String str : arr){
            list.add(str);
        }
        return list;
    }
    //计算表达式
    public static int calc(List<String> list ){
        //创建存放字符串的栈
        Stack<String> stack = new Stack<>();
        //遍历list
        for (int i = 0; i < list.size(); i++){
            //正则表达式匹配是否是数字
            if(list.get(i).matches("\\d+")){
                stack.push(list.get(i)); //是数字则放入栈中
            }else {
                int num2 = Integer.parseInt(stack.pop()); //弹出数字1
                int num1 = Integer.parseInt(stack.pop()); //弹出数字2
                int res = 0;
                //进行运算
                if(list.get(i).equals("+")){
                    res = num1 + num2;
                }else if(list.get(i).equals("-")){
                    res = num1 - num2;
                }else if(list.get(i).equals("*")){
                    res = num1 * num2;
                }else if(list.get(i).equals("/")){
                    res = num1 / num2;
                }else {
                    throw new RuntimeException("不是操作符号!");
                }
                stack.push(""+res);
            }
        }
        //留在栈中的值就是最后的计算表达式结果
        return Integer.parseInt(stack.pop());
    }
}

```

9. 阶乘

```

public class Multiply {
    public static int multiply(int num)
    { if (num < 0) {
        System.out.println("请输入大于 0 的数!");
    }
    }
}

```

```

        return -1;
    } else if (num == 0 || num == 1)
    { return 1;
    } else {
        return multiply(num - 1) * num;
    }
}
public static void main(String[] args)
{ System.out.println(multiply(10));
}
}

```

10. 兔子问题 斐波那契数列

古典问题：有一对兔子，从出生后第 3 个月起每个月都生一对兔子，小兔子长到第三个月后每个月又生一对兔子，假如兔子都不死，问每个月的兔子总数为多少？

```

// 这是一个斐波拉契数列问题
public static void main(String[] args)
{ System.out.println("第1个月的兔子数    1");
  System.out.println("第2个月的兔子数    1");
  int f1 = 1, f2 = 1, f, M=24;
  for(int i=3; i<=M; i++)
  { f = f2;
    f2 = f1 + f2;
    f1 = f;
    System.out.println("第" + i + "个月的兔子数 "+f2);
  }
}
}

```

11. 判断 101-200 之间有多少个素数，并输出所有素数。

程序分析：判断素数的方法：用一个数分别去除 2 到 sqrt(这个数)，如果能被整除， 则表明此数不是素数，反之是素数。

```
public class lianxi02 {
    public static void main(String[] args)
    {
        int count = 0;
        for(int i=101; i<200; i+=2)
        {
            boolean b = false;
            for(int j=2; j<=Math.sqrt(i); j++)
            {
                if(i % j == 0) { b = false; break; }
                else { b = true; }
            }
            if(b == true) {count ++;System.out.println(i);}
        }
        System.out.println( "素数个数是 " + count);
    }
}
```

12 水仙花数

打印出所有的 "水仙花数 "，所谓 "水仙花数 "是指一个三位数，其各位数字立方和等于该数本身。例如：153 是一个 "水仙花数 "，因为 $153=1$ 的三次方+ 5 的三次方+ 3 的三次方。

```
public class lianxi03 {
    public static void main(String[] args)
    {
        int b1, b2, b3;
        for(int m=101; m<1000; m++)
        {
            b3 = m / 100;
            b2 = m % 100 / 10;
            b1 = m % 10;
            if((b3*b3*b3 + b2*b2*b2 + b1*b1*b1) == m)
            {
                System.out.println(m+"是一个水仙花数");
            }
        }
    }
}
```

13. 正整数分解质因数

将一个正整数分解质因数。例如：输入 90,打印出 $90=2*3*3*5$ 。

程序分析：对 n 进行分解质因数，应先找到一个最小的质数 k，然后按下述步骤完成：

- (1)如果这个质数恰等于 n，则说明分解质因数的过程已经结束，打印出即可。
- (2)如果 $n \neq k$ ，但 n 能被 k 整除，则应打印出 k 的值，并用 n 除以 k 的商,作为新的正整数你

n,重复执行第一步。

(3) 如果 n 不能被 k 整除, 则用 k+1 作为 k 的值,重复执行第一步。

```
import java.util.*;
public class lianxi04{
    public static void main(String[] args)
    { Scanner s = new
      Scanner(System.in);
      System.out.print("请输入一个正整数      ");
      int n = s.nextInt();
      int k=2;
      System.out.print(n + "=" );
      while(k <= n) {
          if(k == n) {System.out.println(n);break;}
          else if( n % k == 0) {System.out.print(k + "*");n = n / k; }
              else k++;
          }
      }
}
```

14. 最大公约数和最小公倍数

输入两个正整数 m 和 n, 求其最大公约数和最小公倍数。

```
/** 在循环中, 只要余数不等于0, 用较大数除以较小的数, 将小的一个数作为下一轮循环的较大数, 取得的余数作为下一轮循环的较小数, 如此循环直到较小的数的值为0, 返回较大的数, 此数即为最大公约数, 最小公倍数为两数之积除以最大公约数 */
import java.util.*;
public class lianxi05 {
    public static void main(String[] args)
    { int a,b,m;
      Scanner s = new Scanner(System.in);
      System.out.print(" 键入一个整数: ");
      a = s.nextInt();
      System.out.print(" 再键入一个整数: ");
      b = s.nextInt();
      deff cd = new deff();
      m = cd.deff(a,b);
      int n = a * b / m;
      System.out.println(" 最大公约数 " + m);
      System.out.println(" 最小公倍数 " + n);
    }
}

class deff{
    public int deff(int x, int y) {
```



```

int t;
if(x < y)
{
    t = x;
    x = y;
    y = t;
}
while(y != 0) {
    if(x == y) return x;
    else {
        int k = x % y;
        x = y;
        y = k;
    }
}
return x;
}
}

```

15. 完数

一个数如果恰好等于它的因子之和，这个数就称为 "完数"。例如 $6=1+2+3$ 。编程找出 1000 以内的所有完数。

```

public class lianxi06 {
    public static void main(String[] args)
    {
        System.out.println("1 到1000 的完数有: ");
        for(int i=1; i<1000; i++) {
            int t = 0;
            for(int j=1; j<= i/2; j++)
            {
                if(i % j == 0) {
                    t = t + j;
                }
            }
            if(t == i)
            {
                System.out.print(i + " ");
            }
        }
    }
}

```

16. 完全平方数

一个整数，它加上 100 后是一个完全平方数，再加上 168 又是一个完全平方数，请问该数是多少？

```

public class lianxi07 {
public static void main(String[] args)
{
    for(int x = 1; x < 100000; x++)
    {
        if(Math.sqrt(x+100) % 1 == 0) {
            if(Math.sqrt(x+268) % 1 == 0) {
                System.out.println(x + "加100 是一个完全平方数, 再加168 又是一个完全平方数");
            }
        }
    }
}
}

```

17. 猴子吃桃问题

猴子吃桃问题：猴子第一天摘下若干个桃子，当即吃了一半，还不瘾，又多吃了一个 第二天早上又将剩下的桃子吃掉一半，又多吃了一个。以后每天早上都吃了前一天剩下的一半零一个。到第 10 天早上想再吃时，见只剩下一个桃子了。求第一天共摘了多少。

```

public class lianxi08 {
public static void main(String[] args)
{
    int x = 1;
    for(int i=2; i<=10; i++)
    {
        x = (x+1)*2;
    }
    System.out.println("猴子第一天摘了" + x + " 个桃子");
}
}

```

18. 两个乒乓球队进行比赛

两个乒乓球队进行比赛，各出三人。甲队为 a,b,c 三人，乙队为 x,y,z 三人。已抽签决定比赛名单。有人向队员打听比赛的名单。a 说他不和 x 比，c 说他不和 x,z 比，请编程找出三队赛手的名单。

```

public class lianxi09 {
static char[] m = { 'a', 'b', 'c' };
static char[] n = { 'x', 'y', 'z' };
public static void main(String[] args)
{
    for (int i = 0; i < m.length; i++)
    {
        for (int j = 0; j < n.length; j++) {
            if (m[i] == 'a' && n[j] == 'x') {
                continue;
            }
        }
    }
}
}

```

```

} else if (m[i] == 'a' && n[j] == 'y')
    { continue;
} else if ((m[i] == 'c' && n[j] == 'x')
    || (m[i] == 'c' && n[j] == 'z'))
    { continue;
} else if ((m[i] == 'b' && n[j] == 'z')
    || (m[i] == 'b' && n[j] == 'y'))
    { continue;
} else
    System.out.println(m[i] + " vs " + n[j]);
}
}
}
}

```

19 求 $1+2!+3!+\dots+20!$ 的和

求 $1+2!+3!+\dots+20!$ 的和

```

public class lianxi10 {
    public static void main(String[] args)
    { long sum = 0;
      long fac = 1;
      for(int i=1; i<=20; i++)
      { fac = fac * i;
        sum += fac;
      }
      System.out.println(sum);
    }
}

```

20. 有 5 个人坐在一起

有 5 个人坐在一起，问第五个人多少岁？他说比第 4 个人大 2 岁。问第 4 个人岁数，他说比第 3 个人大 2 岁。问第三个人，又说比第 2 人大两岁。问第 2 个人，说比第一个人大两岁。最后问第一个人，他说是 10 岁。请问第五个人多大？

```

public class lianxi11 {
    public static void main(String[] args)
    { int age = 10;
      for(int i=2; i<=5; i++)
      { age =age+2;
      }
    }
}

```

```

        System.out.println(age);
    }
}

```

21 回文数

一个 5 位数，判断它是不是回文数。即 12321 是回文数，个位与万位相同，十位与千位相同。

```

import java.util.*;
public class lianxi12 {
    public static void main(String[] args)
    {
        Scanner s = new
        Scanner(System.in); int a;
        do{
            System.out.print("请输入一个5      位正整数: ");
            a = s.nextInt();
        }while(a<10000||a>99999);
        String ss =String.valueOf(a);
        char[] ch = ss.toCharArray();
        if(ch[0]==ch[4]&&ch[1]==ch[3]){ System.o
        ut.println("这是一个回文数);}
        else {System.out.println("这不是一个回文数);}
        }
    }
}
//这个更好，不限位数
public class lianxi12 {
    public static void main(String[] args)
    {
        Scanner s = new
        Scanner(System.in); boolean is
        =true;
        System.out.print("请输入一个正整数: ");
        long a = s.nextLong();
        String ss = Long.toString(a);
        char[] ch = ss.toCharArray();
        int j=ch.length;
        for(int i=0; i<j/2; i++)
        { if(ch[i]!=ch[j-i-
        1]){is=false;}
        }
        if(is==true){System.out.println("这是一个回文数);}
        else {System.out.println("这不是一个回文数);}
        }
    }
}

```

22. 100 以内的素数

求 100 之内的素数

```
//使用除sqrt(n)的方法求出的素数不包括2 和3
public class lianxi13 {
    public static void main(String[] args)
    {
        boolean b=false;
        System.out.print(2 + " ");
        System.out.print(3 + " ");
        for(int i=3; i<100; i+=2) {
            for(int j=2; j<=Math.sqrt(i); j++)
            {
                if(i % j == 0) {b = false;
                    break;
                }
                else{b = true;}
            }
            if(b == true) {System.out.print(i + " ");}
        }
    }
}

//该程序使用除1 位素数得2 位方法，运行效率高通用性差。
public class lianxi13 {
    public static void main(String[] args)
    {
        int[] a = new int[]{2, 3, 5, 7};
        for(int j=0; j<4; j++)System.out.print(a[j] + " ");
        boolean b=false;
        for(int i=11; i<100; i+=2)
        {
            for(int j=0; j<4; j++) {
                if(i % a[j] == 0) {b = false;
                    break;
                }
                else{b = true;}
            }
            if(b == true) {System.out.print(i + " ");}
        }
    }
}
```

23. 矩阵对角线元素之和

求一个 3*3 矩阵对角线元素之和

```
import java.util.*;
public class lianxi14 {
    public static void main(String[] args)
    {
        Scanner s = new
        Scanner(System.in); int[][] a = new
        int[3][3];
```

```

for(int i=0; i<3; i++)
{ for(int j=0; j<3; j++)
{ a[i][j] = s.nextInt();
}
}
System.out.println("输入的3 * 3 矩阵是");
for(int i=0; i<3; i++) {
for(int j=0; j<3; j++)
{ System.out.print(a[i][j] + "
");
}
System.out.println();
}
int sum = 0;
for(int i=0; i<3; i++)
{ for(int j=0; j<3; j++)
{ if(i == j) {
sum += a[i][j];
}
}
}
System.out.println("对角线之和是 "+sum);
}

```

24. 杨辉三角形

打印出杨辉三角形（要求打印出 10 行如下图）

```

      1
     1 1
    1 2 1
   1 3 3 1
  1 4 6 4 1
 1 5 10 10 5 1
.....

```

```

public class lianxi15 {
public static void main(String[] args)
{ int[][] a = new int[10][10];
for(int i=0; i<10; i++)
{ a[i][i] = 1;
a[i][0] = 1;
}
for(int i=2; i<10; i++)
{ for(int j=1; j<i; j++)

```

```

        a[i][j] = a[i-1][j-1] + a[i-1][j];
    }
}

for(int i=0; i<10; i++) {
    for(int k=0; k<2*(10-i)-1; k++) {
        System.out.print(" ");
    }
    for(int j=0; j<=i; j++)
        { System.out.print(a[i][j] + " ");
        }
    System.out.println();
}
}
}

```

25. 有 n 个人围成一圈

有 n 个人围成一圈，顺序排号。从第一个人开始报数（从 1 到 3 报数），凡报到 3 的人退出圈子，问最后留下的是原来第几号的那位。

```

import java.util.Scanner;

public class lianxi16 {
    public static void main(String[] args)
    { Scanner s = new
      Scanner(System.in);
      System.out.print("请输入排成一圈的人数 ");
      int n = s.nextInt();
      boolean[] arr = new boolean[n];
      for(int i=0; i<arr.length; i++)
      { arr[i] = true;
      }
      int leftCount = n;
      int countNum = 0;
      int index = 0;
      while(leftCount > 1)
      { if(arr[index] == true)
        { countNum ++;
          if(countNum == 3)
          { countNum =0;
            arr[index] = false;
            leftCount --;
          }
        }
        index ++;
      }
    }
}

```

```

    index = 0;
}
}
for(int i=0; i<n; i++)
{ if(arr[i] == true) {
    System.out.println("原排在第+(i+1)+"位的人留下了。");
}
}
}
}

```

26. 海滩上有一堆桃子

海滩上有一堆桃子，五只猴子来分。第一只猴子把这堆桃子凭据分为五份，多了一个，这只猴子把多的一个扔入海中，拿走了一份。第二只猴子把剩下的桃子又平均分成五份，又多了一个，它同样把多的一个扔入海中，拿走了一份，第三、第四、第五只猴子都是这样做的，问海滩上原来最少有多少个桃子？

```

public class lianxi17{
public static void main (String[] args)
{ int i,m,j=0,k,count;
for(i=4;i<10000;i+=4)
{ count=0;
m=i;
for(k=0;k<5;k++)
{
j=i/4*5+1;
i=j;
if(j%4==0)
count++;
else break;
}
i=m;
if(count==4)
{System.out.println("原有桃子"+j+"个");
break;}
}
}
}
}

```

25 一个偶数总能表示为两个素数之和

一个偶数总能表示为两个素数之和。


```

//由于用sqrt(n)的方法求出的素数不包括2和3，
//因此在判断是否是素数程序中人为添加了一个3。
import java.util.*;
public class lianxi18{
public static void main(String[] args)
{ Scanner s = new Scanner(System.in);
int n,i;
do{
    System.out.print(" 请输入一个大于等于6      的偶数: ");
    n = s.nextInt();
    } while(n<6||n%2!=0); //判断输入是否是=6 偶数不是重新输入
fun fc = new fun();
    for(i=2;i<=n/2;i++){ if(fc.fun(i))=
        =1&&(fc.fun(n-i)==1))
        {int j=n-i;
            System.out.println(n+ " = "+i+ " + "+j);
        } //输出所有可能的素数对
    }
}
}
class fun{
public int fun (int a) //判断是否是素数的函数
{
int i,flag=0;
if(a==3){flag=1;return(flag);}
for(i=2;i<=Math.sqrt(a);i++){
    if(a%i==0) {flag=0;break;}
    else flag=1;}
return (flag) ;//不是素数返回0,是素数返回1
}
}
//解法二
import java.util.*;
public class lianxi18 {
public static void main(String[] args)
{ Scanner s = new Scanner(System.in);
int n;
do{
    System.out.print(" 请输入一个大于等于6      的偶数: ");
    n = s.nextInt();
    } while(n<6||n%2!=0); //判断输入是否是=6 偶数不是重新输入
for(int
    i=3;i<=n/2;i+=2){ if(f
un(i)&&fun(n-i)) {

```

```
System.out.println(n+" = "+i+" + "+(n-i));
```

```

    } //输出所有可能的素数对
  }
}
static boolean fun (int a){ //判断是否是素数的函数
flag=false;
if(a==3){flag=true;return(flag);}
for(int i=2;i<=Math.sqrt(a);i++){
    if(a%i==0) {flag=false;break;}
    else flag=true;}
return (flag);
}
}

```

27. 冒泡排序

冒泡排序

```

/**
 * 冒泡排序<br/>
 *
 * <li> 比较相邻的元素如果第一个比第二个大,就交换他们两个.</li>
 * <li> 对每一对相邻元素作同样的工作,从开始第一对到结尾的最后一对。在这一点,最后的元素应会是最大的数。</li>
 * <li> 针对所有的元素重复以上的步骤,除了最后一个。</li>
 * <li> 持续每次对越来越少的元素重复上面的步骤,直到没有任何一对数字需要比较。</li>
 *
 *
 * @param numbers
 *      需要排序的整型数组
 */
public static void bubbleSort(int[] numbers)
{
    int temp; // 记录临时中间值
    int size = numbers.length; // 数组大小
    for (int i = 0; i < size - 1; i++)
    {
        for (int j = i + 1; j < size; j++)
        {
            if (numbers[i] < numbers[j]) { // 交换两数的位置
                temp = numbers[i];
                numbers[i] = numbers[j];
                numbers[j] = temp;
            }
        }
    }
}
}

```

28. 快速排序

快速排序

```
/**
 * 快速排序<br/>
 * <ul>
 * <li>从数列中挑出一个元素，称为“基准”</li>
 * <li>重新排列数列，所有元素比基准值小的摆放在基准前面，所有元素比基准值大的摆在基准的后面
    (相等数可以摆任一边)。在这个分区操作之后
 * <li>该基准是它的最后位置。这个称为分割 (partition) 操作</li>
 * <li>递归地把小于基准值元素的子数列和大于基准值元素的子数列排序。</li>
 * </ul>
 *
 * @param numbers
 * @param start
 * @param end
 */
public static void quickSort(int[] numbers, int start, int end)
{
    if (start < end) {
        int base = numbers[start]; // 选定的基准值 (第一个数值作为基准值)
        int temp; // 记录临时中间值
        int i = start, j = end;
        do {
            while ((numbers[i] < base) && (i < end))
                i++;
            while ((numbers[j] > base) && (j > start))
                j--;
            if (i <= j) {
                temp = numbers[i];
                numbers[i] = numbers[j];
                numbers[j] = temp;
                i++;
                j--;
            }
        } while (i <= j);
        if (start < j)
            quickSort(numbers, start, j);
        if (end > i)
            quickSort(numbers, i, end);
    }
}
```

29. 选择排序

选择排序

```
/**
 * 选择排序 br/>
 * <li> 在排序序列中找到最小元素, 存放到排序序列的起始位置 /li>
 * <li> 再从剩余未排序元素中继续寻找最小元素, 然后放到排序序列末尾 </li>
 * <li> 以此类推, 直到所有元素排序完毕 </li>
 *
 *
 * @param numbers
 */
public static void selectSort(int[] numbers)
{
    int size = numbers.length, temp;
    for (int i = 0; i < size; i++)
    {
        int k = i;
        for (int j = size - 1; j > i; j--) {
            if (numbers[j] < numbers[k]) k = j;
        }
        temp = numbers[i];
        numbers[i] = numbers[k];
        numbers[k] = temp;
    }
}
```

30. 插入排序

插入排序

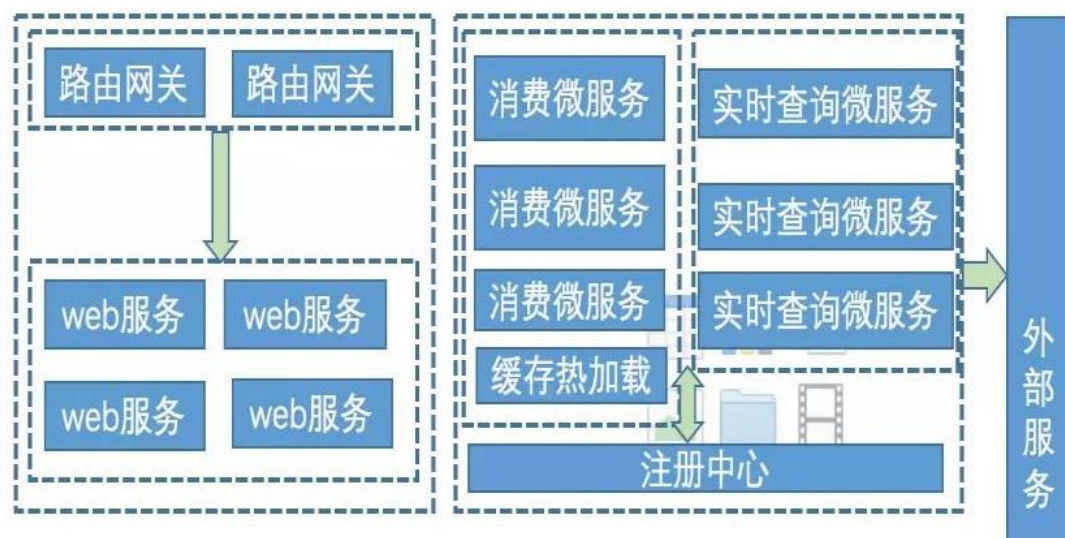
```
/**
 * 插入排序 br/>
 * <ul>
 * <li> 从第一个元素开始, 该元素可以认为是已经被排序 /li>
 * <li> 取出下一个元素, 在已经排序的元素序列中从后向前扫描 /li>
 * <li> 如果元素(已排序)大于新元素, 将该元素移到下一位置 /li>
 * <li> 重复步骤3, 直到找到已排序的元素小于或等于新元素的位置 /li>
 * <li> 将新元素插入到该位置中 /li>
 * <li> 重复步骤2 </li>
 * </ul>
 *
 *
 * @param numbers
 */
public static void insertSort(int[] numbers)
{
    int size = numbers.length, temp, j;
    for (int i = 1; i < size; i++) {
        temp = numbers[i];
```

```
for(j = i; j > 0 && temp < numbers[j-1]; j--)  
    numbers[j] = numbers[j-1];  
    numbers[j] = temp;  
}  
}
```

十二、并发与性能调优

1. 每秒钟 5k 个请求，查询手机号的笔试题，设计算法?请求再多，比如 5w，如何设计整个系统?

设计出每秒并 5K 的一个系统，根据网上的这个题目做以下梳理，众所周知一个好的架构需要考虑它的高可用和可伸缩，需要做服务的熔断、降级、隔离等等



架构设计原理:

- 1、路由网关-流量分发入口，不承载具体业务，简单点可以使用 **nginx**，如果是微服务可以使用 **zuul** 等（支持请求的分发、限流、下游依赖的发现，可以结合 **docker** 实现服务下游的 **web** 服务自动伸缩），如果采用 **nginx** 完成不了下游的伸缩发现，但是基本的限流和分发可以解决
- 2、web 服务-可以水平扩展，通过 **cache** 加速，查询手机号码号段对应的地区，对于缓存未命中的号段，直接丢入 **kafka** 队列，实时返回 **client** 端查询中的状态
- 3、消费微服务-完成 **kafka** 队列的消费，根据 **kafka topic** 的 **partition** 个数也可以实现水平扩展，负责把发送号段的查询请求至实时查询微服务，保存至 **cache**
- 4、实时查询微服务-因为是无状态服务，根据业务负载也可以实现水平扩展，且仅负责对外部运营商的查询，根据外部供应商的接口能力，也可以通过 **hystrix** 把该服务 **export** 出的接口做限流和熔断，这样影响面就不会波及外部合作伙伴
- 5、缓存预热服务-为提升体验，减少发出查询请求后的刷新等待时间，在服务发布前，可以预先先把一批号段通过请求实时查询微服务，并先保存起来

总结

如上设计：缓存读取不会形成瓶颈，队列生产不会形成瓶颈，唯一形成瓶颈的点有可能发生 在外部运营商接口，因此我们会对实时查询服务做限流和熔断，所以不会压垮运营商，但是 用户端的体验就糟些了，所以我们需要把缓存预热的功夫做足，改善体验。上面的设计在不同场景下需要进行微调，基本思想不会发生大的变化，把请求异步化，一天吃不成胖子，就 分多天吃，就是这个意思，当然还考察了服务的隔离、降级、可伸缩的特性！

2. 高并发情况下，我们系统是如何支撑大量的请求的？

1、尽量使用缓存技术，

包括用户缓存，信息缓存还有静态页面缓存，多花点内存来做缓存，可以大大减少与数据库 的交互次数和 tomcat 执行次数，减少不变的数据重复在 tomcat 和数据库中获取的次数。

2、同步转异步，

对于一些不需要即时结果的操作，可以使用 MQ 消息机制，达到同步转异步的效果，如秒杀系统，正常流程：先点击秒杀，然后往服务器发送请求，在页面等待响应，这样融入大量请求，服务器压力会特别大，搞不好服务器就会宕机。使用 MQ 消息队列实现异步的步骤为：点击秒杀，返回请稍后查看结果，请求去 MQ 队列中排队，等排队执行完成后返回给用户信息。这样就可以大大减少服务器的压力，提升用户体验度。

3、合并多个同类型请求为一个请求，

使用 SpringCloud 的 Hystrix 技术来实现。在服务提供者提供了返回单个对象和多个对象的接口，并且单个对象的查询并发数很高，服务提供者负载较高的时候，我们就可以使用请求合并来降低服务提供者的负载。

4、数据库方面，

搭建数据库集群，网站一般读的多写的少，可以按照网站的统计数据来找到一个合适的平衡点，来搭建主从数据库服务，可以实现一主多从，或者多主多从，来减轻单个数据库的压力。 可以按照每台数据库服务器的硬件条件，合理分配权重，配合 Mycat 达到负载均衡。

5、高质量代码，

合理的使用循环和递归，不要为了速度丢了内存，也不要为内存丢了速度，要看业务场景，来合理使用。减少自动处理逻辑，比如字符串拼接，每次拼接都会创建一个字符串放入常量池，这里可以按照业务场景来使用 StringBuilder 或者 StringBuffer 来进行字符串拼接，能手动处理就手动处理，代码中所有的临时对象，用完之后都赋值为 Null，这样可以减少 GC 的重复排查，效率就会有所提升。所有的资源用完都要回收，如：IO、数据库连接对象等，因为这些资源对 GC 不是特别友好。减少代码调用链，尽量不要让代码调用链超过 10，远程方法调用没事。提供过滤能力，把每个过滤器写的详细一点，把耦合度高的数据放入到同一个过滤器中，如果第一个过滤器没有通过那么后面的过滤器不执行，相对的业务也就不执行了， 效率也就提升了。

6、网络优化，

外网转内网，内网转局域网，外网转 VPN。配合公司内的网络运维人员，进行网络网段的切换，尽量让服务器群处于内网，或者局域网中，提供访问速度。服务器之间的通讯如果都是局域网内进行的，那么可想而知，访问速度肯定有所提升。

7、中间件处理

搭建 Tomcat 集群，通过 Nginx 代理 Tomcat 服务器做负载均衡，对每个 Tomcat 的调优，合理设置 Tomcat 的最大连接数，因为 Tomcat 的默认最大并发数为 200。适当的加大 Tomcat 的内存和最多线程数，设置 JVM 的堆大小为服务器可用内存的最大值的 80%。关闭 DNS 查询，开启 gzip 压缩。

搭建 MQ 集群，高并发的时候一个 MQ 来处理队列根本不够用，这时可以搭建集群来处理。增加 Nginx 的内存，加大 Nginx 缓存数据的范围。服务器操作系统都用 64 位的，因为 32 位的系统最大内存只能有 4G

图片服务器分离，搭建 vsftpd 服务器来存储图片数据，通过 Nginx 代理 vsftpd 存放路径就可以直接访问到图片，这样响应到页面的只是超链接，并不是图片，这样页面的响应会得到大大的提升。

3. 集群如何同步会话状态？

利用 Redis 同步 session

Redis 可以做分布式，正式因为这个功能他才可以用来做 session 同步。他可以把 web 服务器中的内存组合起来，形成一个“内存池”，不管是哪个服务器产生的 session 都可以存放于这个内存池中，其它的都可以使用。

以这种方式来同步 session，不会加大数据库的负担，安全性比 cookie 要大大提高，把 session 放到内存中，这样比从文件读取也要快很多。

4. 负载均衡的原理？

网站访问量已经越来越大，响应速度越来越慢。

考虑：

Scale Up（也就是 Scale vertically）纵向扩展，向上扩展：机器硬件升级，增加配置，如添加 CPU、内存。（往往需要购置新机器）->旧机器不能利用上。

Scale Out（也就是 Scale horizontally）横向扩展，向外扩展：向原有的 web、邮件系统添加一个新机器。->旧机器仍然可以发挥作用。

负载均衡技术为 scale out 服务。

Nginx 负载均衡器的特点是：

1. 工作在网络的 7 层之上，可以针对 http 应用做一些分流的策略，比如针对域名、目录结构；
2. Nginx 安装和配置比较简单，测试起来比较方便；
3. 也可以承担高的负载压力且稳定，一般能支撑超过上万次的并发；
4. Nginx 可以通过端口检测到服务器内部的故障，比如根据服务器处理网页返回的状态码、超时等等，并且会把返回错误的请求重新提交到另一个节点，不过其中缺点就是不支持 url 来检测；
5. Nginx 对请求的异步处理可以帮助节点服务器减轻负载；
6. Nginx 能支持 http 和 Email，这样就在适用范围上面小很多；
7. 默认有三种调度算法：轮询、weight 以及 ip_hash（可以解决会话保持的问题），还可以支持第三方的 fair 和 url_hash 等调度算法；

5. 怎么提高并发量，请列举你所知道的方案？

HTML 静态化 模板引擎

图片服务器分离 Nginx

数据库集群、库表散列 数据分片 Mycat

缓存 基于 Redis

镜像 是大型网站常采用的提高性能和数据安全性的方式

负载均衡

CDN 加速技术

6. 系统的用户量有多少？多用户并发访问时如何解决？

分布式是以缩短单个任务的执行时间来提升效率的，而集群则是通过提高单位时间内执行的任务数来提升效率。

集群主要分为：高可用集群(High Availability Cluster)，负载均衡集群(Load Balance Cluster，nginx 即可实现)，科学计算集群(High Performance Computing Cluster)。

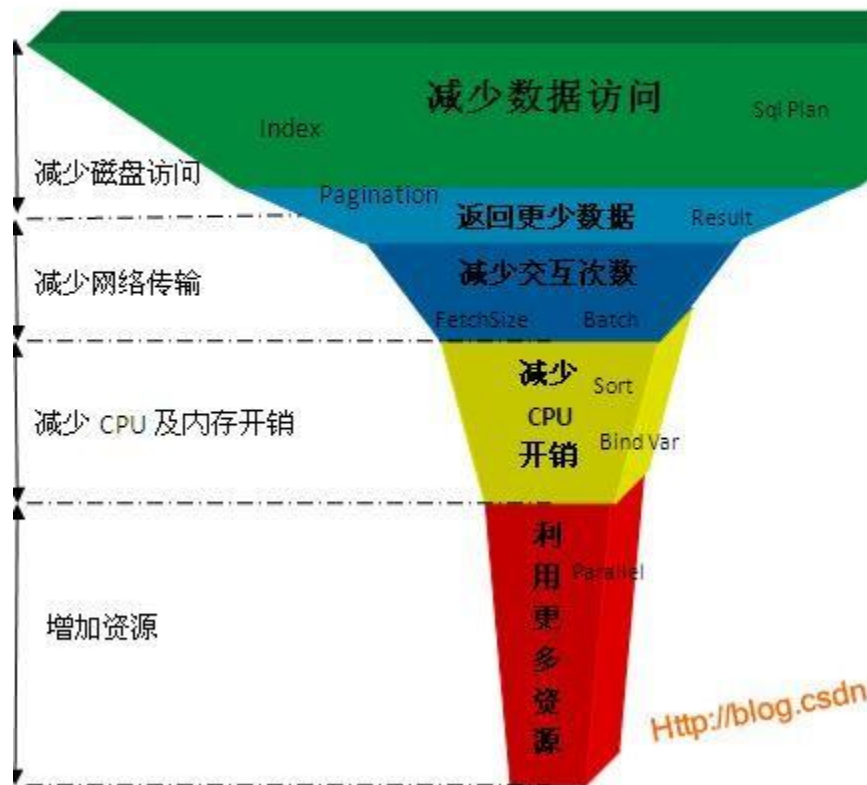
分布式是指将不同的业务分布在不同的地方；而集群指的是将几台服务器集中在一起，实现同一业务。分布式中的每一个节点，都可以做集群。而集群并不一定就是分布式的。

采用微服务架构，使用技术 Spring Cloud 的一站式解决方案

7. 如果有一个特别大的访问量，到数据库上，怎么做优化？

1. SQL 语句的优化处理 通过慢查询确认执行效率低下的 SQL 语句，进行拆解和索引的控制
2. 为数据库搭建集群，实现主从复制
3. 实现数据库的读写分离
4. 实现数据的分片处理
5. 采用数据库中间件 Mycat

数据库访问优化漏斗法则



8. 大面积并发，在不增加服务器，如何解决服务器响应不及时问题？

衡量服务器的并发能力

1. 吞吐率

吞吐率，单位时间里服务器处理的最大请求数，单位 req/s

2. 压力测试

使用 Jmeter，压力测试中关心的时间又细分以下 2 种：

用户平均请求等待时间（这里暂不把数据在网络的传输时间，还有用户 PC 本地的计算时间计算入内）

服务器平均请求处理时间

提高服务器的并发能力 1.

提高 CPU 并发计算能力

2. 考虑减少内存分配和释放

3. 考虑使用持久连接

4. 改进 I/O 模型

5. Sendfile Linux 提供 sendfile() 系统调用，可以讲磁盘文件的特定部分直接传送到代表客户端的 socket 描述符，加快了静态文件的请求速度，同时减少 CPU 和内存的开销。

6. 内存映射