千锋郑州 JavaEE 葵花宝典

V5.0

此典在手，就业无忧！

一、Java 基础.....1
1. final 关键字的作用？.....1
2. Java 集合 list,set,queue,map,stack 的特点与用法？.....1
3. 说出 ArrayList,Vector, LinkedList 的存储性能和特性？.....1
4. 内存泄漏和内存溢出？.....1
5. 反射中,Class.forName()和 ClassLoader.loadClass()的区别？.....2
6. int 和 Integer 的区别？.....2
7. String,StringBuilder,StringBuffer 区别？.....2
8. Hashtable 和 Hashmap 的区别？.....2
9. 说几个常见的异常？.....2
10. 方法重载的规则？.....3
11. 方法重写的规则？.....3
12. throw 和 throws 的区别?.....3
13. 抽象类和接口的区别?.....3
14. Java 的基本类型和字节大小?.....4
15. 访问修饰符的区别访问级别？.....4
16. HashSet 的底层实现是什么？.....4
17. 为什么重写 equals 时必须重写 hashCode 方法？.....4
18. HashSet 和 TreeSet 有什么区别？.....4
19. Java 中的四大引用分别是什么？.....4
20. 数组在内存中如何分配?.....5
21. Java 中怎么创建一个不可变对象？.....5
22. Java 中++操作符是线程安全的吗？.....5
23. new 一个对象的过程和 clone 一个对象的过程？.....5
24. Java 中==和 equals()的区别？.....6
25. final,finalize 和 finally 的不同之处？.....6
26. Java 的多态是什么,表现在哪里?.....6
27. 静态类型有什么特点？.....6
28. Java 创建对象的几种方式？.....6
29. Object 中有哪些公共方法?.....6
30. &和&&的区别？.....7
31. 在 java 源文件中可以有多个类吗内部类除外？.....7

32. 如何正确的退出多层嵌套循环？ 7
33. 内部类有什么作用？ 7
34. 深拷贝和浅拷贝的区别是什么？ 7
35. String 是基本数据类型吗？ 7
36. static 的用法？ 7
37. 什么是值传递和引用传递？ 8
38. 重载和重写的区别？ 8
39. 成员变量和局部变量的区别有哪些？ 8
40. 静态方法和实例方法有何不同？ 8
41. 多态的优点？ 8
42. 多态存在的三个必要条件？ 9
43. TreeMap,HashMap,LindedHashMap 的区别？ 9
44. Java 面向对象的特征有哪些方面？ 9
46. 什么是反射？ 9
47. 反射的作用？ 10
48. 获取 class 的三种方式？ 10
49. break 和 continue 的区别？ 10
50. Collection 和 Collections 的区别？ 10
51. Error 和 Exception 有什么区别？ 10
52. Comparable 和 Comparator 接口的区别？ 10
53. switch 能否作用在 byte,long,String 上？ 10
54. jdk 中哪些类是不能继承的？ 10
55. JDK 和 JRE 的区别是什么？ 11
56. 是否可以在 static 环境中访问非 static 变量？ 11
57. Java 支持多继承么？ 11
58. 什么是迭代器(Iterator)？ 11
59. Iterator 和 ListIterator 的区别是什么？ 11
60. Enumeration 接口和 Iterator 接口的区别有哪些？ 11
61. 字符串常量池到底存在于内存空间的哪里？ 11
62. Java 中的编译期常量是什么,使用它又什么风险？ 12
63. 用哪两种方式来实现集合的排序？ 12
65. ArrayList 源码分析？ 12
66. HashMap 源码分析？ 12
67. ConcurrentHashMap 源码分析？ 13
68. super 和 this 的共同点与区别？ 13
69. Concurrenthashmap 为什么是线程安全的？ 14
70. 异常的分类? 14
71. 遍历集合的时候能否增删元素？ 15
72. JUC 中线程安全的集合？ 15
二、Java IO 16
1. IO 里面的常见类,字节流,字符流,接口,实现类,方法阻塞？ 16
2. 谈谈对 NIO 的认知？ 16
3. 字节流和字符流的区别？ 16
4. NIO 和传统的 IO 有什么区别？ 16

5. BIO 和 NIO 和 AIO 的区别以及应用场景？ 17
6. 什么是 Java 序列化,如何实现 Java 序列化？ 17
7. PrintStream,BufferedWriter,PrintWriter 的比较？ 17
8. 什么是节点流,什么是处理流,有什么好处？ 18
9. Java 中有几种类型的流？ 18
10. 如何实现对象克隆？ 18
11. 什么是缓冲区,有什么作用？ 18
12. 什么是阻塞 IO,什么是非阻塞 IO？ 18
13. NIO 的缓冲区以及非阻塞？ 19
三、Java Web 20
1. session 和 cookie 的区别？ 20
2. servlet 的生命周期？ 20
3. 什么是 webservice？ 20
4. jsp 和 servlet 的区别,共同点,各自应用的范围？ 20
5. 转发 forward 和重定向 redirect 的区别？ 21
6. request.getAttribute()和 request.getParameter()有何区别？ 21
7. jsp 静态包含和动态包含的区别？ 21
8. MVC 的各个部分都有哪些技术来实现？如何实现？ 21
9. jsp 有哪些内置对象,作用分别是什么？ 22
10. Http 请求的 get 和 post 方法的区别? 22
11. tomcat 容器是如何创建 servlet 类实例？用到了什么原理？ 22
12. JDBC 访问数据库的基本步骤是什么？ 23
13. 为什么要使用 PreparedStatement？ 23
14. 数据库连接池的原理,为什么要使用连接池？ 23
15. execute,executeQuery,executeUpdate 的区别是什么？ 23
16. JDBC 的 ResultSet 是什么？ 24
17. 什么是 Servlet？ 24
18. JSP 常用的指令？ 24
19. JSP 的四大范围？ 24
20. BS 与 CS 的联系与区别？ 24
21. 说出 Servlet 的生命周期? 25
22. 如何防止表单重复提交？ 25
23. request 作用？ 26
24. 响应乱码？ 26
25. Cookie 对象的缺陷？ 26
26. Session 的运行机制？ 26
27. HttpSession 钝化和活化？ 26
28. Filter 的工作原理？ 27
29. Filter 链是什么？ 27
30. 监听器类型？ 27
31. Servlet,Filter,Listener 启动顺序？ 27
32. Tomcat 的 IO 模型:BIO,NIO,AIO,APR? 28
33. 如何获取新增数据的 ID？ 28
34. 常用的日志框架? 28

35.加密算法算法有哪些？ 29
四、JVM 31
1.Java 的内存划分？ 31
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？ 32
3.如何判断一个对象应该被回收？ 32
4.GC 触发的条件? 33
5.可以作为 GCRoots 的对象有哪些？ 33
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？ 33
7.双亲委派模型？ 33
8.为什么需要双亲委派模型？ 33
9.怎么打破双亲委派模型？ 33
10.导致 Full GC 一般有哪些情况？ 34
11.Minor GC,Full GC 触发条件？ 34
12.JVM 性能调优？ 34
13.Java 内存模型？ 34
14.Java 中堆和栈有什么区别？ 35
15.垃圾回收算法有哪些,简述其原理？ 35
16.解释栈(stack)堆(heap)和方法区(method area)的用法？ 36
17.类的加载过程是什么? 36
18.类加载器有哪些? 37
19.Java 对象创建过程? 37
20.Java 中类的生命周期是什么? 37
21.都有哪些垃圾回收器? 37
22.JVM 调优命令？ 38
23.JVM 调优工具？ 38
24.描述一下 JVM 加载 class 的原理？ 38
25.GC 是什么,为什么要有 GC？ 38
26.垃圾回收器的基本原理是什么？ 39
27.Java 中的引用类型有几种？ 39
28.GC 的回收器有哪些？ 39
29.GC 相关参数-调优？ 40
30.Java 堆空间内存溢出？ 41
31.GC 开销超过限制,引起 OOM? 41
32.请求的数组大引起 OOM,怎么解决？ 41
33.永久代(Perm gen)空间,引起 OOM? 41
34.Metaspace 元空间耗尽,引起 OOM? 41
35.无法新建本机线程？ 42
五、多线程 42
1.线程调用 start()和 run()的区别? 42
2.线程 B 怎么知道线程 A 修改了变量? 42
3.synchronized 和 Volatile,CAS 比较? 42
4.线程间通信,wait 和 notify 的理解和使用? 43
5.定时线程的使用? 43
6.线程同步的方法? 43

7. 进程和线程的区别?.....43
8. 什么叫线程安全? 43
9. 线程的几种状态?.....43
10. volatile 变量和 atomic 变量有什么不同? 44
11. Java 中什么是静态条件? 44
12. Java 中如何停止一个线程? 44
13. 线程池的优点?.....44
14. volatile 的理解?.....45
15. 实现多线程有几种方式?.....45
16. Java 中 notify 和 notifyAll 有什么区别? 45
17. 什么是乐观锁和悲观锁? 45
18. 线程的创建方式?.....45
19. 线程池的作用?.....45
20. wait 和 sleep 的区别?.....46
21. 产生死锁的条件?.....46
22. 请写出实现线程安全的几种方式?.....46
23. 守护线程是什么,它和非守护线程的区别?.....46
24. 什么是多线程的上下文切换?.....46
25. Callable 和 Runnable 的区别是什么?.....46
26. 线程阻塞有哪些原因? 46
27. synchronized 和 Lock 的区别?.....47
28. ThreadLocal 是什么,有什么作用? 47
29. 交互方式分为同步和异步两种?.....47
30. 什么是线程? 47
31. 什么是 FutureTask? 47
32. Java 中 interrupted 和 isInterruptedd 方法的区别? 48
33. 死锁的原因?.....48
34. 什么是自旋锁? 48
35. 怎么唤醒一个阻塞的线程?.....48
36. 提交任务时,线程池队列已满,会发生什么?.....48
37. 什么是线程局部变量? 48
38. 使用 volatile 关键字的场景?.....49
39. 线程池的 7 大参数什么意思? 49
40. 线程池的类型?.....49
41. 线程池的阻塞队列有哪些? 49
42. 线程池的拒绝策略都有哪些?.....50
42. 线程的生命周期?.....50
44. 线程的调度方法 wait 和 join 与 notify.....50
45. 公平锁与非公平锁?.....51
46. 可重入锁与不可重入锁?.....51
46. synchronized 的三种用法与区别?.....51
47. CPU 的调度算法?.....52
50. 项目中使用了哪种线程池，使用场景是什么?.....52
51. ReentrantLock 的理解? 52

52. Lock 接口的主要方法？ 52
53. ReentrantLock 与 synchronized 的区别？ 53
54. tryLock 和 lock 和 lockInterruptibly 的区别？ 53
55. ReadWriteLock 读写锁是干什么的？ 54
56. 共享锁和独占锁是什么？ 54
57. 你对偏向锁是理解是什么？ 54
58. Java 中用到的线程调度？ 55
59. 进程调度算法有哪些？ 55
60. 什么 CAS？ 57
61. 为什么 wait\notify 是在 Object 而不是 Thread 中？ 57
62. 锁消除和锁粗化的理解是什么？ 57
63. 线程池的五大状态？ 58
64.Synchonized 的实现原理？ 58
65.Java 中锁的升级过程？ 59
六、设计模式 60
1. 说一下你熟悉的设计模式？ 60
2. 简单工厂和抽象工厂的区别? 60
3. 设计模式的优点? 60
4. 设计模式的六大基本原则? 60
5. 单例模式？ 60
6. 设计模式的分类？ 61
7. 设计模式的每种模式的功能？ 61
8.UML 是什么？ 62
9.桥接模式是什么？ 62
10. 享元模式 63
11. 策略模式是什么？ 64
12.代理模式是什么？ 65
七、开源框架 66
1. Hibernate 和 Mybatis 的区别？ 66
2. MyBatis 的优点？ 66
3. MyBatis 框架的缺点？ 66
4. SpringMVC 工作流程？ 66
5. MyBatis 框架使用的场合? 67
6. Spring 中 beanFactory 和 ApplicationContext 的联系和区别? 67
7. SpringIOC 注入的几种方式? 68
8. 拦截器与过滤器的区别? 68
9. SpringIOC 是什么? 68
10. AOP 有哪些实现方式？ 68
11. 解释一下代理模式？ 68
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？ 69
13. Spring bean 的生命周期？ 69
14. Spring 框架中都用到了哪些设计模式？ 69
15. Spring 中的事件处理？ 69
16. 使用 Sping 框架的好处是什么？ 70

17. 解释 Spring 支持的几种 bean 的作用域？ 70
18. 在 Spring 中如何注入一个 java 集合？ 70
19. 什么是 Spring bean？ 70
20. 什么是 Spring 自动装配？ 71
21. 自动装配有哪些方式？ 71
22. 自动装配有什么局限？ 71
23. Spring 的重要注解？ 71
24. @Component,@Controller,@Repository,@Service 有何区别？ 71
25. 列举 Spring 支持的事务管理类型? 72
26. Spring 框架的事物管理有哪些优点? 72
27. Spring AOP（面向切面）编程的原理？ 72
28. Spring MVC 框架有什么用？ 72
29. 介绍一下 WebApplicationContext? 72
30. SpringMVC 和 struts2 的区别有哪些? 73
31. Mybatis 中#{}和${}的区别是什么? 73
32. Spring 中@Autowire 与@Resource 的区别？ 73
33. 什么是 IOC,什么是 DI？ 74
34. Spring 运行原理? 74
35. Spring 的事务传播行为？ 74
36.Spring 的循环依赖? 75
37. Spring 单例的 Bean 是线程安全的吗? 75
38.Mybatis 的缓存策略：一级缓存与二级缓存? 76
39.@Controller 与@RestController 的区别? 76
八、分布式相关 76
1. Redis 和 Memcache 的区别？ 76
2. 使用 Redis 有哪些好处？ 77
3. 什么是 Redis 持久化,rdb 和 aof 的比较? 77
4. Redis 最适合的场景? 77
5. Redis 哈希槽的概念? 78
6. Redis 和数据库的数据一致性如何保证？ 78
7. Redis 的淘汰策略有哪些? 78
8. Redis 有哪些数据结构? 78
9. Redis 缓存穿透、缓存雪崩、缓存击穿? 78
10. Redis 如何实现高并发? 79
11. Redis 如何实现高可用? 79
12. Redis 单线程还能处理速度那么快？ 79
13. 为什么 Redis 的操作是原子性的,怎么保证原子性？ 79
14. Redis 的主从复制的实现过程? 79
15. Redis 的哨兵机制的作用? 80
16. Redis 常见的性能问题和解决方案? 80
17. 分布式缓存? 80
18. Redis 的过期策略？ 81
19. Redis 是如何发现 hot 和 bigkey 的？ 81
20. RedLock（红锁）分布式实现原理？ 82

21.基于 Redis 分布式锁？ 82
23.Mysql 与 Redis 的数据一致性？ 82
24. 什么是 Nginx? 83
25. Nginx 相对 apache 的优点? 83
26. Nginx 优化的方式？ 83
27. Nginx 如何处理一个请求的? 84
28. Nginx 是如何实现高并发的? 84
29. Nginx 的进程模型? 84
30. Nginx 负载均衡的 4 种分配方式? 84
31. 为什么要用 Nginx？ 85
32. 四层负载均衡与七层负载均衡 85
33. 负载均衡策略/算法有哪些？ 86
34. Nginx 如何实现反向代理负载均衡？ 87
35. 什么是正向代理? 87
36. 什么是反向代理? 87
37. 什么是负载均衡? 87
38. Nginx 的调度算法有哪些？ 87
39. Nginx 负载均衡调度状态? 88
40. 可以从哪些方面来优化 nginx 服务？ 88
41. 为什么要用 MQ? 89
42. 使用 MQ 会有什么问题？ 89
43. 怎么保证 MQ 的高可用？ 89
44. MQ 的优缺点? 89
45. Kafka,ActiveMQ,RabbitMQ,RocketMQ 都有什么区别？ 89
46. 如何设置消息的过期时间？ 90
47. 消息的持久化是如何实现的？ 90
48. 如何保证消息的幂等性？ 90
49. 什么是 RabbitMQ 的死信队列？ 90
50. 什么是延迟队列,延迟队列的实现方式 91
51. 如何保证消息不丢失？ 91
52. Zookeeper 是什么？ 92
53. Zookeeper 的应用场景? 92
54. 四种类型的数据节点 Znode? 92
55. Zookeeper Watcher 机制? 92
56. Zookeeper 下 Server 工作状态? 93
57. Zookeeper 是如何保证事务的顺序一直性的？ 93
58. ZK 节点宕机如何处理？ 93
59. Zookeeper 有哪几种部署模式？ 94
60. Zookeeper 角色 94
61. Zookeeper 工作原理（原子广播）？ 94
62. 什么是 Dubbo？ 95
63. 为什么要用 Dubbo？ 95
64. Dubbo 和 Spring Cloud 有什么区别？ 95
65. dubbo 都支持什么协议，推荐用哪种？ 95

66. Dubbo 需要 Web 容器吗？ 95
67. Dubbo 内置了哪几种容器？ 95
68. Dubbo 里面有哪几种角色？ 96
69. Dubbo 有哪几种集群容错方案，默认是那种？ 96
70. Dubbo 有哪几种负载均衡策略，默认是哪种？ 96
71. Dubbo 的管理控制台能做什么？ 96
72.Dubbo 默认使用什么注册中心，还有别的选择吗？ 96
73.Dubbo 有哪几种配置方式？ 96
74.Dubbo 核心的配置有哪些？ 96
75.Dubbo 推荐使用什么序列化框架，你知道的还有哪些？ 97
76.Dubbo 默认使用的是什么通信框架，还有别的选择吗？ 97
77.Dubbo 支持服务多协议吗？ 97
78. 什么是 Spring Boot？ 97
79. Spring Boot 有哪些优点？ 97
80. 什么是 JavaConfig？ 97
81. 如何重新加载 Spring Boot 上的更改，而无需重新启动服务器？ 98
82. Spring Boot 中的监视器是什么？ 98
83. 如何在 Spring Boot 中禁用 Actuator 端点安全性？ 98
84. 什么是 WebSocket？ 98
85. 什么是 Swagger？你用 Spring Boot 实现了它吗？ 99
86. 什么是 Apache Kafka？ 99
87. 什么是 Spring Cloud？ 99
88. 使用 Spring Cloud 有什么优势？ 99
89. 服务注册和发现是什么意思？Spring Cloud 如何实现？ 99
90. 什么是 Netflix Feign？它的优点是什么？ 100
91. 什么是 Spring Cloud Bus？我们需要它吗？ 100
92. 什么是 Hystrix 断路器？我们需要它吗？ 102
93. 什么是 ELK？ 102
94. 你对微服务的认知？ 103
95. SpringCloud 的常用组件？ 103
96. 服务是如何被注册到 Nacos 的 103
97. Springboot 支持松绑定么？什么是松绑定？ 103
98. 什么是 CAP？ 103
99.@ComponentScan 注解的作用？ 104
100.@SpringBootApplication 注解的作用？ 104
101.@EnableAutoConfiguration(自动装配)注解的作用？ 104
102. Docker 是什么？ 105
103. linux 怎么查看系统负载情况？ 105
104. 什么是 OAuth2？ 106
105. OAuth2 的四种授权模式 106
106.RedLock 的实现原理？ 107
九、网络通信 109
1. http 协议的状态码有哪些,含义是什么？ 109
2. Http 的请求报文组成？ 110

3. 一次完整的 Http 请求是怎样的？ 110
4. TCP 和 UDP 的区别？ 110
5. SSL 协议的三个特性? 110
6. TCP 的三次握手与四次挥手? 110
7. TCP 为什么连接是三次握手,关闭却是四次握手？ 111
8. 建立了连接,但是客户端突然出现故障了怎么办？ 111
9. http 中重定向和请求转发的区别？ 111
10.Http 与 Https 的区别? 112
11. 什么是无状态协议?怎么解决 Http 协议无状态协议? 112
13. HTTPS 工作原理? 112
十、数据库 112
1. MySql 的存储引擎有哪些,区别是什么? 112
2. 触发器的作用？ 113
3. 什么是存储过程？用什么来调用？ 113
4. 存储过程的优缺点? 113
5.SQL 优化的具体操作? 113
6. 什么叫视图,游标是什么？ 114
7. 视图的优缺点？ 114
8. 事务的四个特性？ 114
9. 数据库乐观锁,悲观锁的区别,怎么实现？ 115
10. 事务的并发问题？ 115
11.MySQL 的 MyISAM 与 InnoDB 存储引擎在事务,锁,场景? 117
12. 非关系型和关系型数据库区别,优势比较？ 117
13. 数据库的五大范式和 BCNF？ 117
14. 什么是内连接,外连接,交叉连结,笛卡尔积等？ 118
15. SQL 语言分类? 119
16. count(*)count(1)count(column)的区别？ 119
17. 什么是索引？ 119
18. 索引的作用？ 119
19. 索引的优缺点？ 119
20. 什么样的字段适合建索引？ 120
21. Hash 索引和 B+树索引的区别？ 120
22. MySQL 三种锁的级别？ 120
23. 为什么不都用 Hash 索引而使用 B+树索引？ 120
24. B 树和 B+树的区别？ 120
25. 为什么 B+比 B 树更适合作为文件索引和数据库索引？ 120
26. 聚集索引和非聚集索引区别? 121
27. 事务的隔离级别？ 121
28. 索引的分类？ 122
29. 索引的最佳左匹配？ 122
30. 什么是幂等性？ 122
31. ACID 是什么?可以详细说一下吗? 122
32. 统计过慢查询吗?对慢查询都怎么优化过? 122
十一、算法 123

1. 冒泡排序？.....123
2.实现两个有序数组的合并排序？.....124
3.实现一个数组的倒序？.....125
4.计算一个正整数的正平方根？.....125
5.快速排序算法？.....125
6.二叉树的遍历算法？.....126
7. DFS, BFS 算法？.....127
8. 时间类型转换？.....128
9. 逆波兰计算器？.....129
10.请实现阶乘？.....130
11.兔子问题或者斐波那契数列？.....131
12.判断 101-200 之间素数,并输出所有素数？.....131
13.请实现水仙花数？.....131
14. 正整数分解质因数？.....132
15. 最大公约数和最小公倍数？.....132
16. 请实现完数？.....134
17. 请实现完全平方数?.....134
18. 猴子吃桃问题？.....134
19. 请实现两个乒乓球队进行比赛？.....135
20.求 1+2!+3!+...+20!的和？.....135
21. 有 5 个人坐在一起？.....136
22.回文数的实现？.....136
23. 100 之内的素数？.....137
24. 矩阵对角线元素之和？.....138
25. 请实现杨辉三角形？.....138
26.有 n 个人围成一圈顺序排号？.....139
27.海滩上有一堆桃子,5 只猴子来分？.....140
28.一个偶数总能表示为两个素数之和？.....141
29.实现快速排序？.....142
30.请实现选择排序？.....143
31.请实现插入排序？.....144
32. 请实现二分查找？.....144
33. 请实现希尔排序？.....145
十二、并发与性能调优.....147
1. 每秒钟 5k 个请求,查询手机号的笔试题,设计算法?.....147
2.高并发下,我们系统是如何支撑请求？.....147
3.集群如何同步会话状态？.....148
4.负载均衡的原理？.....149
5.怎么提高并发量,请列举你所知道的方案？.....149
6.系统的用户量有多少,多用户并发访问时如何解决？.....149
7.如果有一个特别大的访问量,到数据库上怎么做优化？.....150
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？.....150
9. 秒杀系统的设计与实现？.....151
十三、开放性问题.....151

1.工作上有哪些有成就感,又有哪些不能接受的地方？.....151
2.跟产品的关系怎么样？.....151
3.你关注外包么,那在做外包的时候,最主要关注的点在哪里？.....151
4.于你个人而言,你觉得软件开发这个行业是怎样一个行业？.....151
5.在工作中有没有遇到过什么比较有趣的问题,最后是怎么解决的？.....151
6.有参与过需求分析会议么？.....151
7.从技术角度来说,你觉得你跟前同事比怎么样？.....152
8.工作中觉得哪方面欠缺？.....152
9.项目出现许多 BUG,你一般会怎么解决？.....152
10.你们公司 Git 的分支？.....152
十四、IDEA 快捷键大全.....152

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.Java 集合 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。

3.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。

4.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。

5.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

6.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.

7.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.

8.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。

9.说几个常见的异常？

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。

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）

10.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。

11.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

12.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 语句是用在方法声明后面，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。

13.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、类如果要实现一个接口，它必须要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。

14. Java 的基本类型和字节大小?

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

15. 访问修饰符的区别访问级别？

public：公共的，都可以访问
protected：受保护的，只要同类中、同包下、子类中可以访问
没有访问修饰符：只能同类中、同包下可以访问
private：私有的，只能同类中可以访问

16. 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。

17. 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).

18. 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(logn)。

19. Java 中的四大引用分别是什么？

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。

20.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。

21.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

22.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。

23.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。

24. Java 中==和 equals()的区别？

本质上没有区别，因为 equals 的内部就是使用的==，但是用法上是有些区别的。
使用==比较原生类型如：boolean、int、char 等等，使用 equals()比较对象。
==是判断两个变量或实例是不是指向同一个内存空间。可以用于判断基本类型是否相同。
equals 是一个方法，只能是对象才可以调用，默认是比较的对象的地址是否相同，但是某些类的 equals 本身发生了重写，比如 String 的 equals 是比较的内容。

25.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。

26.Java 的多态是什么,表现在哪里?

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）
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。

27.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

28.Java 创建对象的几种方式？

1.new 创建新对象;
2.通过反射机制;
3.采用 clone 机制;
4.通过序列化机制

29.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==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。

30.&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）。

31.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

32.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
3、return

33.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

34.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.

35.String 是基本数据类型吗？

不是的，基本数据类型只包括 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

36.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。

37.什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。

38.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)。

39.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。

40.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

41.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。

42.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）

43.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。

44.Java 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）。

46.什么是反射？

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。

47.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

48.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

49.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。

50.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。

51.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。

52.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。

53.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）

54.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。

55. 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。

56. 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。

57. 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。

58. 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。

59. 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。

60. 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。

61. 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

62.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。

63.用哪两种方式来实现集合的排序？

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。

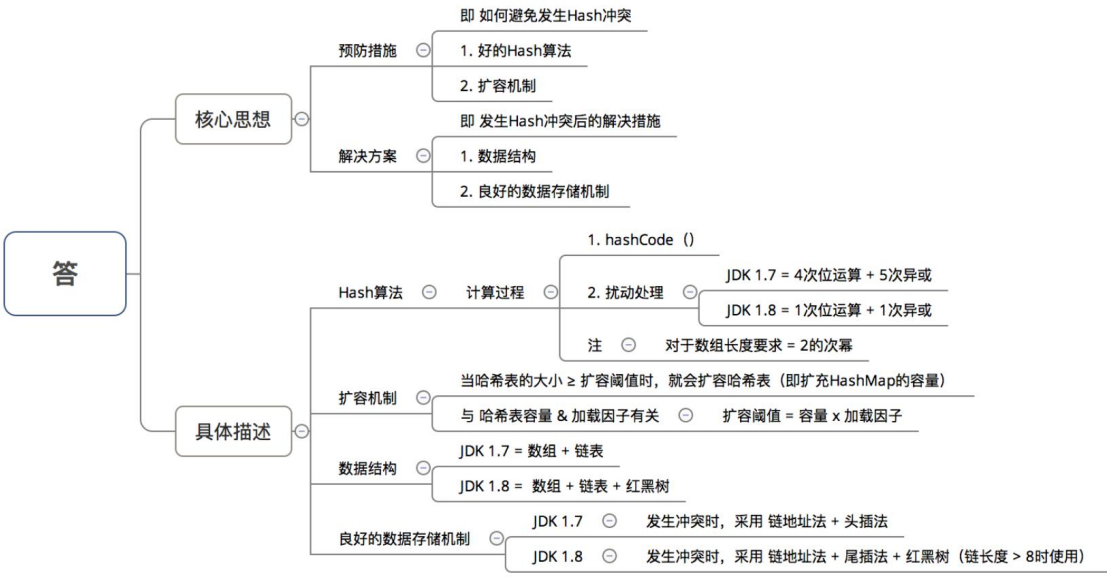
64.说出一些 JDK1.8 的新特性？

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。

65.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，然后将旧数组内容复制到新创建的更大的数组中。
（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

66.HashMap 源码分析？



67.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，这需要实现上的保证。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。

68.super 和 this 的共同点与区别？

super(参数)：调用基类中的某一个构造函数（应该为构造函数中的第一条语句）
this(参数)：调用本类中另一种形成的构造函数（应该为构造函数中的第一条语句）
super： 它引用当前对象的直接父类中的成员(用来访问直接父类中被隐藏的父类中成员数据或函数，基类与派生类中有相同成员定义时如：super.变量名 super.成员函数据名（实参）
this：它代表当前对象名（在程序中易产生二义性之处，应使用 this 来指明当前对象；如果函数的形参与类中的成员数据同名，这时需用 this 来指明成员变量名）
调用 super()必须写在子类构造方法的第一行，否则编译不通过。每个子类构造方法的第一条语句，都是隐含地调用 super()，如果父类没有这种形式的构造函数，那么在编译的时候就会报错。
super() 和 this() 类似,区别是，super() 从子类中调用父类的构造方法，this() 在同一类

内调用其它方法。

super() 和 this() 均需放在构造方法内第一行。

尽管可以用 this 调用一个构造器，但却不能调用两个。

this 和 super 不能同时出现在一个构造函数里面，因为 this 必然会调用其它的构造函数，其它的构造函数必然也会有 super 语句的存在，所以在同一个构造函数里面有相同的语句，就失去了语句的意义，编译器也不会通过。

this() 和 super() 都指的是对象，所以，均不可以在 static 环境中使用。包括：static 变量,static 方法，static 语句块。

从本质上讲，this 是一个指向本对象的指针，然而 super 是一个 Java 关键字。

69.Concurrenthashmap 为什么是线程安全的？

1.7 之前是因为数组加链表形式实现的，它是一个个大数组 Segment(可以理解为数据库)和小数组 HashEntry(每张表）数组构成的 ，大数组 Segment 本身是基于 ReentrantLock 实现的加锁和释放锁的操作，这样就能保证多个线程同时访问 ConcurrentHashMap 时，同一时间只有一个线程能操作相应的节点，这样就保证了 ConcurrentHashMap 的线程安全了

1.8 因为数组加链表形式实现的**所以它的效率会很慢

而 JDK1.8 则使用了数组+链表/红黑树的方式优化了 ConcurrentHashMap 的实现 ，

链表升级为红黑树的规则：当链表长度大于 8，并且数组的长度大于 64 时，链表就会升级为红黑树的结构。

CAS+volatile 或 synchronized 的方式来保证线程安全的，

它是通过 CAS 或 synchronized 来实现线程安全的，并且它的锁粒度更小，查询性能也更高。

在 jdk1.8 中，添加元素的时候首先会判断容器是否为空，如果为空，使用 volatile 加 cas 来初始化，如果容器不是空的就根据存储的元素计算该位置是不是空的，如果是空的，就使用 cas 设置该节点，就用如果不为空则使用 synchronize 加锁，只让一个线程操作 ，遍历桶中的数据，替换或新增节点到桶中，最后再判断是否需要转为红黑树，这样就能保证并发访问时的线程安全了。

我们可以简单的认为在 JDK1.8 中，ConcurrentHashMap 是在头节点加锁来保证线程安全的，锁的粒度相比 Segment 来说更小了，发生冲突和加锁的频率降低了，并发操作的性能就提高了。而且 JDK1.8 使用的是红黑树优化了之前的固定链表，那么当数据量比较大的时候，查询性能也得到了很大的提升。

70.异常的分类?

Throwable 是 Java 语言中所有错误或异常的超类。下一层分为 Error 和 Exception

Error

1.Error 类是指 java 运行时系统的内部错误和资源耗尽错误。应用程序不会抛出该类对象。如果出现了这样的错误，除了告知用户，剩下的就是尽力使程序安全的终止。

Exception（RuntimeException、CheckedException）

2.Exception 又有两个分支，一个是运行时异常 RuntimeException，一个是 CheckedException。

RuntimeException 如： NullPointerException 、 ClassCastException ；一个是检查异常 CheckedException，如 I/O 错误导致的 IOException、SQLException。 RuntimeException 是那些可能在 Java 虚拟机正常运行期间抛出的异常的超类。 如果出现 RuntimeException，那么

一定是程序员的错误.

检查异常 CheckedException：一般是外部错误，这种异常都发生在编译阶段，Java 编译器会强制程序去捕获此类异常，即会出现要求你把这段可能出现异常的程序进行 try catch，该类异常一般包括几个方面：

1.试图在文件尾部读取数据

2.试图打开一个错误格式的 URL

3.试图根据给定的字符串查找 class 对象，而这个字符串表示的类并不存在

71.遍历集合的时候能否增删元素？

1.foreach 遍历，iterator 遍历都不能在遍历的过程中使用 list.remove 或 list.add 操作，会报并发修改异常，遍历删除后加个 break 即可解决。

2.iterator 遍历过程中如果需要删除可以使用 iterator 提供的 remove()方法。

3.遍历根据元素索引删除是可行的。

72.JUC 中线程安全的集合？

依靠 Synchronized 保证线程安全的集合(并非 JUC)：Vector,stack,hashtable

以上三种集合效率较低

1.CopyOnWriteArrayList

2.ConcurrentHashMap

3.CopyOnWriteArraySet

4.ArrayBlockingQueue

5.LinkedBlockingQueue

CopyOnWriteArrayList 和 CopyOnWriteArraySet 的原理是当我们往一个容器添加元素的时候，不直接往当前容器添加，而是先将当前容器进行 Copy，复制出一个新的容器，然后新的容器里添加元素，添加完元素之后，再将原容器的引用指向新的容器。

这时候会抛出来一个新的问题，也就是数据不一致的问题。如果写线程还没来得及写会内存，其他的线程就会读到了脏数据。==这就是 CopyOnWriteArrayList 的思想和原理。就是拷贝一份。

它最适合于具有以下特征的应用程序：

1.List 大小通常保持很小，只读操作远多于可变操作，需要在遍历期间防止线程间的冲突。

2.它是线程安全的。

3.因为通常需要复制整个基础数组，所以可变操作（add()、set() 和 remove()等等）的开销很大。

4.迭代器支持 hasNext(), next()等不可变操作，但不支持可变 remove()等操作。

5.使用迭代器进行遍历的速度很快，并且不会与其他线程发生冲突。在构造迭代器时，迭代器依赖于不变的数组快照。

6.独占锁效率低：采用读写分离思想解决

7.写线程获取到锁，其他写线程阻塞

8.复制思想

ConcurrentHashMap 的原理是因为 ConcurrentHashMap 是由一个 Segment 数组和多个 HashEntry 数组组成，Segment 是一种可重入锁 ReentrantLock),ConcurrentHashMap 里面扮演锁的角色；HashEntry 则用于存储键值对数据。

那么为什么他的效率高呢？

这就是我之前说为为何同步锁根本扛不住很大并发的原因。因为 HashTable 之流，是很多线程去争夺一把锁。而 ConcurrentHashMap 的锁的粒度很小，每个分段去分一把锁。所以效率高。

BlockQueue 的原理是

1、 使用锁机制。使用了 ReentrantLock，可重入锁，在做关键操作之前，先调用 ReentrantLock 的 lockInterruptibly 方法进行上锁，在执行完成之后，调用 unlock 方法解锁。

2、 因为 Blocking 机制，队列满了，就要等有空余空间才能 put 新元素。使用了线程安全的 AtomicInteger 类来进行当前元素个数的计数工作，并与 capacity 容量进行比较。

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改方法时，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,有什么好处?

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.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。

10.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

11.什么是缓冲区,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。

12.什么是阻塞 IO,什么是非阻塞 IO?

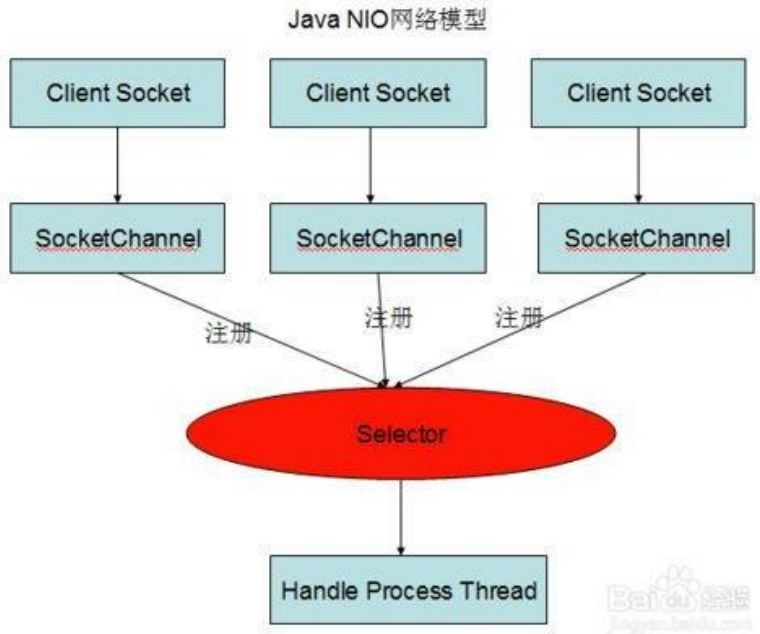
IO 操作包括：对硬盘的读写、对 socket 的读写以及外设的读写。
当用户线程发起一个 IO 请求操作（本文以读请求操作为例），内核会去查看要读取的数据是否就绪，对于阻塞 IO 来说，如果数据没有就绪，则会一直在那等待，直到数据就绪；对于非阻塞 IO 来说，如果数据没有就绪，则会返回一个标志信息告知用户线程当前要读的数据没有就绪。当数据就绪之后，便将数据拷贝到用户线程，这样才完成了一个完整的 IO 读请求操作，也就是说一个完整的 IO 读请求操作包括两个阶段：
1）查看数据是否就绪；
2）进行数据拷贝（内核将数据拷贝到用户线程）。
那么阻塞（blocking IO）和非阻塞（non-blocking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

13.NIO 的缓冲区以及非阻塞？

1. NIO 的介绍

NIO 主要有三大核心部分：Channel(通道)，Buffer(缓冲区)， Selector。传统 IO 基于字节流和字符流进行操作，而 NIO 基于 Channel 和 Buffer(缓冲区)进行操作，数据总是从通道读取到缓冲区中，或者从缓冲区写入到通道中。Selector(选择区)用于监听多个通道的事件（比如：连接打开， 数据到达）。因此，单个线程可以监听多个数据通道。



NIO 和传统 IO 之间第一个最大的区别是，IO 是面向流的，NIO 是面向缓冲区的。

2. NIO 的缓冲区

Java IO 面向流意味着每次从流中读一个或多个字节，直至读取所有字节，它们没有被缓存在任何地方。此外，它不能前后移动流中的数据。如果需要前后移动从流中读取的数据，需要先将它缓 存到一个缓冲区。NIO 的缓冲导向方法不同。数据读取到一个它稍后处理的缓冲区，需要时可在缓冲区中前后移动。这就增加了处理过程中的灵活性。但是，还需要检查是否该缓冲区中包含所 有您需要处理的数据。而且，需确保当更多的数据读入缓冲区时，不要覆盖缓冲区里尚未处理的 数据。

3.NIO 的非阻塞

IO 的各种流是阻塞的。这意味着，当一个线程调用 read() 或 write()时，该线程被阻塞，直到有一些数据被读取，或数据完全写入。该线程在此期间不能再干任何事情了。 NIO 的非阻塞模式， 使一个线程从某通道发送请求读取数据，但是它仅能得到目前可用的数据，如果目前没有数据可用时，就什么都不会获取。而不是保持线程阻塞，所以直至数据变的可以读取之前，该线程可以继续做其他的事情。 非阻塞写也是如此。一个线程请求写入一些数据到某通道，但不需要等待它完全写入，这个线程同时可以去做别的事情。 线程通常将非阻塞 IO 的空闲时间用于在其它通道上执行 IO 操作，所以一个单独的线程现在可以管理多个输入和输出通道（channel）。

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 确切的说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.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。

3.什么是 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。

4.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.

5.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.

6.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

7.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

8.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），"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。

9. 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。

10. 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 表示可能修改变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。

11. 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。

12.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。

13.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。

14.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

15.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。

16. 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。

17. 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 都必须要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。

18. 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

19. 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。

20. 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。

21.说出 Servlet 的生命周期?

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。

22.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

23.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

24.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。

25.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

26.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

27.HttpSession 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

28.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

29.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。

30.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。

31.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。

32.Tomcat 的 IO 模型:BIO,NIO,AIO,APR?

BIO:

bio(blocking I/O)，顾名思义，即阻塞式 I/O 操作，表示 Tomcat 使用的是传统的 Java I/O 操作(即 java.io 包及其子包)。Tomcat 在默认情况下，就是以 bio 模式运行的。遗憾的是，就一般而言，bio 模式是三种运行模式中性能最低的一种。我们可以通过 Tomcat Manager 来查看服务器的当前状态。

NIO:

是 Java SE 1.4 及后续版本提供的一种新的 I/O 操作方式(即 java.nio 包及其子包)。Java nio 是一个基于缓冲区、并能提供非阻塞 I/O 操作的 Java API,因此 nio 也被看成是 non-blocking I/O 的缩写。它拥有比传统 I/O 操作(bio)更好的并发运行性能。

APR:

(Apache Portable Runtime/Apache 可移植运行库)，是 Apache HTTP 服务器的支持库。你可以简单地理解为，Tomcat 将以 JNI 的形式调用 Apache HTTP 服务器的核心动态链接库来处理文件读取或网络传输操作，从而大大地提高 Tomcat 对静态文件的处理性能。 Tomcat apr 也是在 Tomcat 上运行高并发应用的首选模式。

33.如何获取新增数据的 ID?

有时候因为新增的需求需要获取刚刚新增的数据的自增的主键 ID，可以使用使用 PreparedStatement.RETURN_GENERATED_KEYS （关键看你使用哪个接口与数据库交互，都有 RETURN_GENERATED_KEYS 这个方法）可以获取刚刚插入自增 ID 值。

34.常用的日志框架?

1.Slf4j

slf4j 的全称是 Simple Loging Facade For Java，即它仅仅是一个为 Java 程序提供日志输出的统一接口，并不是一个具体的日志实现方案，就比如 JDBC 一样，只是一种规则而已。所以单独的 slf4j 是不能工作的，必须搭配其他具体的日志实现方案，比如 apache 的 org.apache.log4j.Logger，jdk 自带的 java.util.logging.Logger 等。

2.Log4j

Log4j 是 Apache 的一个开源项目，通过使用 Log4j，我们可以控制日志信息输送的目的地是控制台、文件、GUI 组件，甚至是套接口服务器、NT 的事件记录器、UNIX Syslog 守护进程等；我们也可以控制每一条日志的输出格式；通过定义每一条日志信息的级别，我们能够更加细致地控制日志的生成过程。Log4j 由三个重要的组成构成:日志记录器(Loggers),输出端(Appenders)和日志格式化器(Layout)。

1. Logger：控制要启用或禁用哪些日志记录语句，并对日志信息进行级别限制
2. Appenders ： 指定了日志将打印到控制台还是文件中
3. Layout ： 控制日志信息的显示格式

Log4j 中将要输出的 Log 信息定义了 5 种级别，依次为 DEBUG、INFO、WARN、ERROR 和 FATAL，当输出时，只有级别高过配置中规定的 级别的信息才能真正的输出，这样就很方便的来配置不同情况下要输出的内容，而不需要更改代码。

3.LogBack

简单地说，Logback 是一个 Java 领域的日志框架。它被认为是 Log4J 的继承人。

Logback 主要由三个模块组成：logback-core，logback-classic。logback-access

logback-core 是其它模块的基础设施，其它模块基于它构建，显然，logback-core 提供了一些关键的通用机制。

logback-classic 的地位和作用等同于 Log4J，它也被认为是 Log4J 的一个改进版，并且它实现了简单日志门面 SLF4J；

logback-access 主要作为一个与 Servlet 容器交互的模块，比如说 tomcat 或者 jetty，提供一些与 HTTP 访问相关的功能。

4.Logback 优点

同样的代码路径，Logback 执行更快

更充分的测试

原生实现了 SLF4J API（Log4J 还需要有一个中间转换层）

内容更丰富的文档

支持 XML 或者 Groovy 方式配置

配置文件自动热加载

从 IO 错误中优雅恢复

自动删除日志归档

自动压缩日志成为归档文件

支持 Prudent 模式，使多个 JVM 进程能记录同一个日志文件

支持配置文件中加入条件判断来适应不同的环境

更强大的过滤器

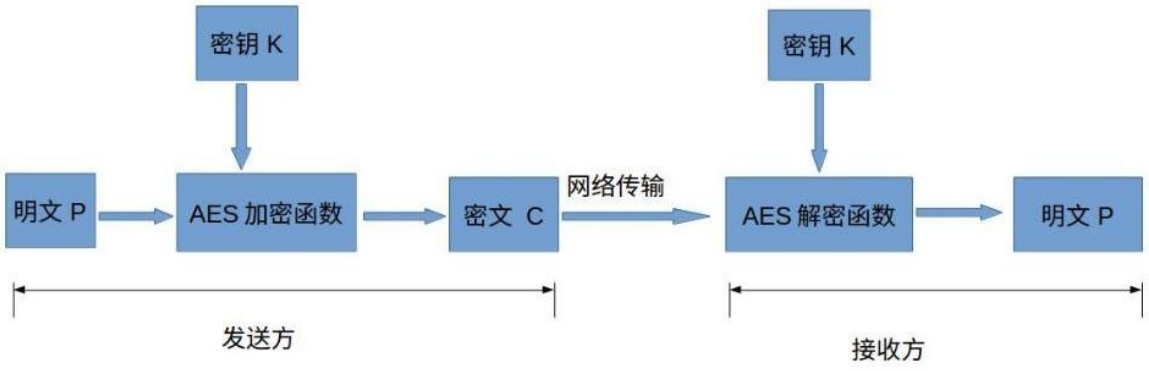
支持 SiftingAppender（可筛选 Appender）

异常栈信息带有包信息

35.加密算法算法有哪些？

AES

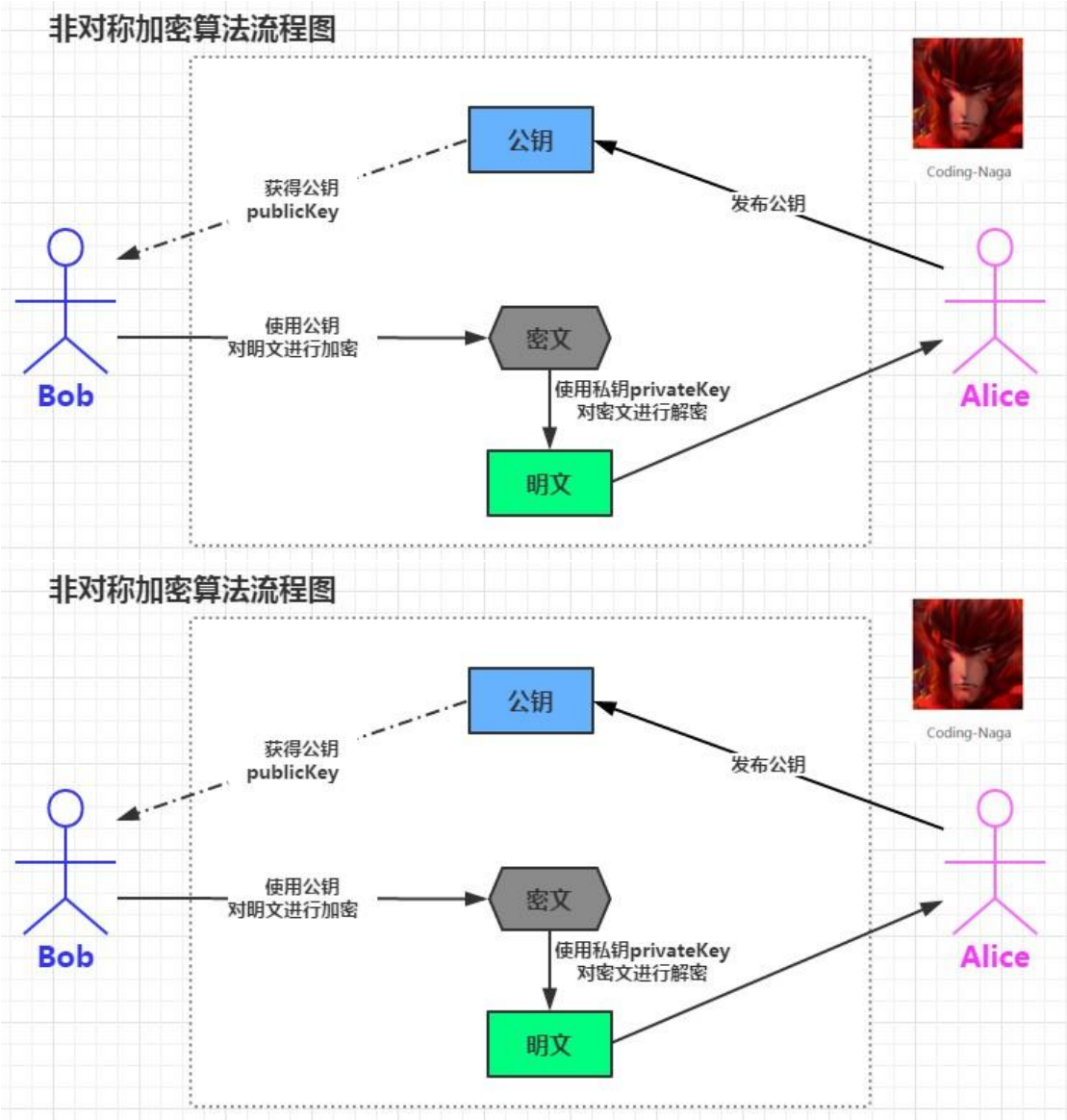
高级加密标准(AES,Advanced Encryption Standard)为最常见的对称加密算法(微信小程序加密传输就是用这个加密算法的)。对称加密算法也就是加密和解密用相同的密钥，具体的加密流程如下图：



RSA

RSA 加密算法是一种典型的非对称加密算法，它基于大数的因式分解数学难题，它也是应用最广泛的非对称加密算法。

非对称加密是通过两个密钥（公钥-私钥）来实现对数据的加密和解密的。公钥用于加密，私钥用于解密。



CRC

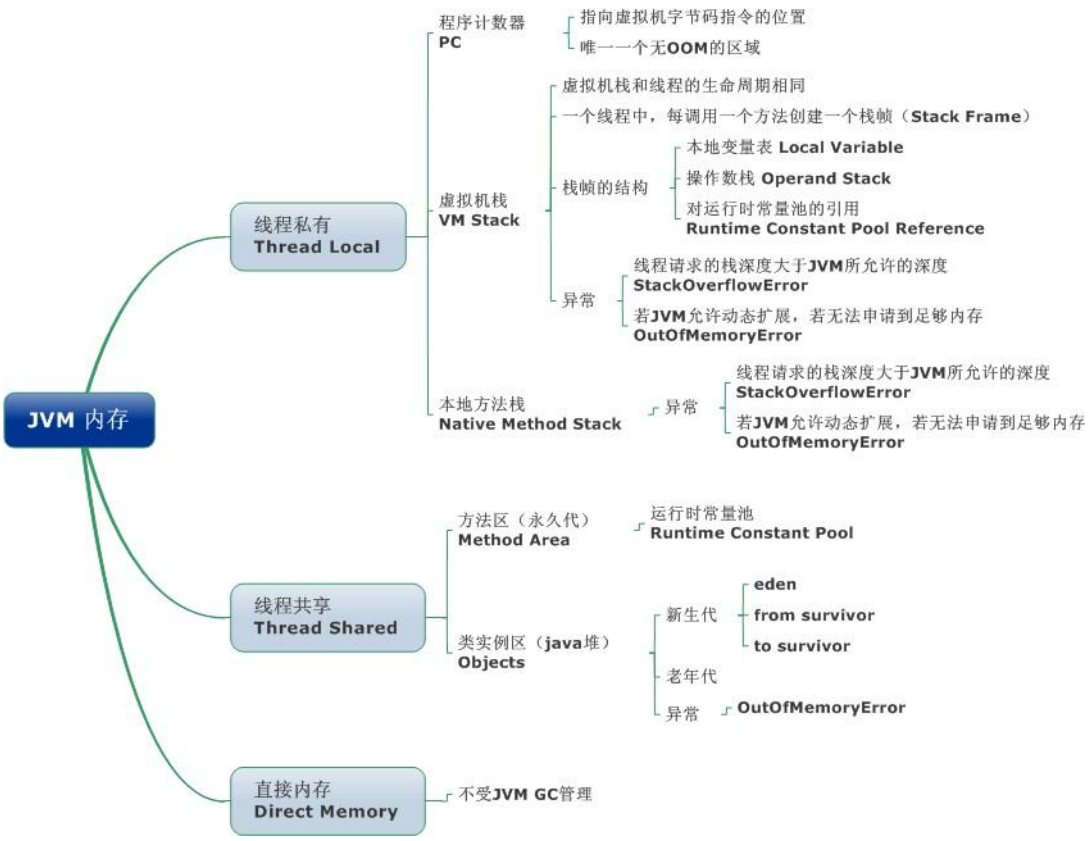
循环冗余校验(Cyclic Redundancy Check, CRC)是一种根据网络数据包或电脑文件等数据产生简短固定位数校验码的一种散列函数，主要用来检测或校验数据传输或者保存后可能出现的错误。它是利用除法及余数的原理来作错误侦测的。

MD5 摘要算法，严格来说不叫加解密算法

MD5 常常作为文件的签名出现，我们在下载文件的时候，常常会看到文件页面上附带一个扩展名为.MD5 的文本或者一行字符，这行字符就是就是把整个文件当作原数据通过 MD5 计算后的值，我们下载文件后，可以用检查文件 MD5 信息的软件对下载到的文件在进行一次计算。两次结果对比就可以确保下载到文件的准确性。　另一种常见用途就是网站敏感信息加密，比如用户名密码， 支付签名等等。随着 https 技术的普及，现在的网站广泛采用前台明文传输到后台，MD5 加密（使用偏移量）的方式保护敏感数据保护站点和数据安全。

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）

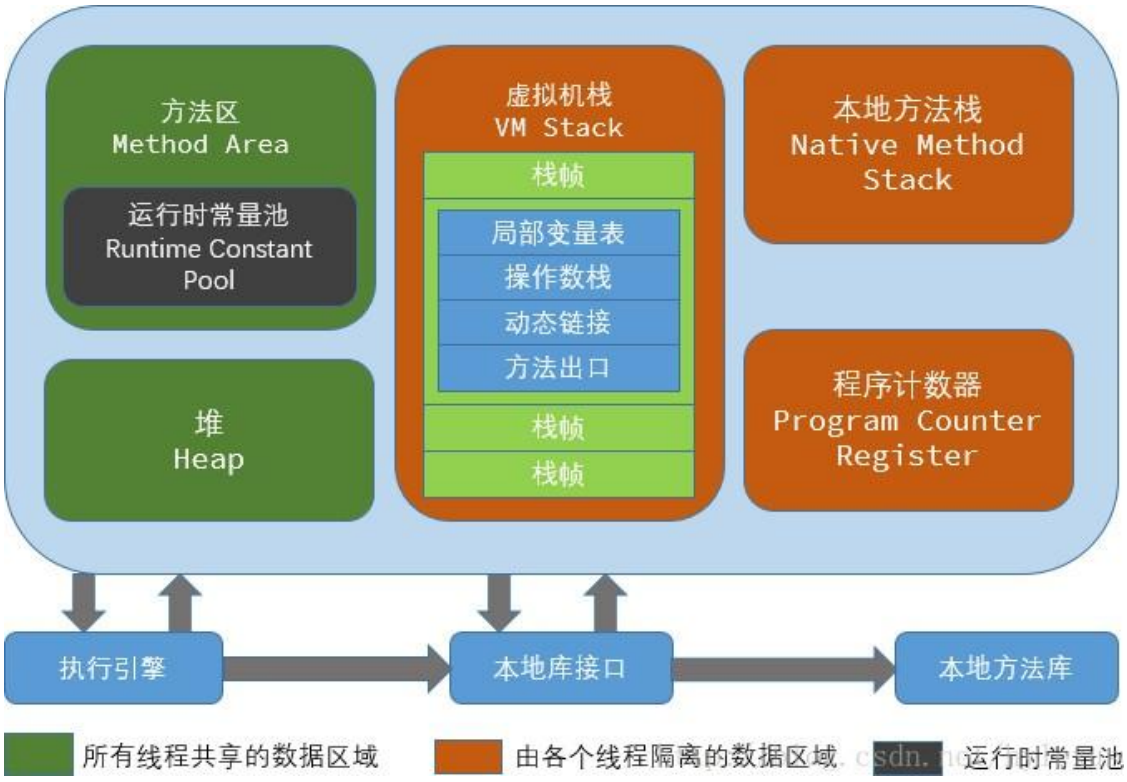
理所当然，堆也是垃圾收集器重点照顾的区域，所以堆内空间还会被不同的垃圾收集器进行进一步的细分，最有名的就是新生代、老年代的划分。

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就会成为可回收对象。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生代，占据了旧生代剩余空间，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生代过小（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 直接到达旧生代

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 Sp3ace 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 新生代和老生代占比

-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 年老代用-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. 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.类的加载过程是什么？

加载

加载一个 Class 需要完成以下 3 件事：

通过 Class 的全限定名获取 Class 的二进制字节流

将 Class 的二进制内容加载到虚拟机的方法区

在内存中生成一个 java.lang.Class 对象表示这个 Class

获取 Class 的二进制字节流这个步骤有多种方式：

从 zip 中读取，如：从 jar、war、ear 等格式的文件中读取 Class 文件内容

从网络中获取，如：Applet

动态生成，如：动态代理、ASM 框架等都是基于此方式

由其他文件生成，典型的是从 jsp 文件生成相应的 Class

校验

验证一个 Class 的二进制内容是否合法，主要包括 4 个阶段：

文件格式验证，确保文件格式符合 Class 文件格式的规范。如：验证魔数、版本号等。

元数据验证，确保 Class 的语义描述符合 Java 的 Class 规范。如：该 Class 是否有父类、是否错误继承了 final 类、是否一个合法的抽象类等。

字节码验证，通过分析数据流和控制流，确保程序语义符合逻辑。如：验证类型转换是合法的。

符号引用验证，发生于符号引用转换为直接引用的时候（转换发生在解析阶段）。如：验证引用的类、成员变量、方法的是否可以被访问（IllegalAccessError），当前类是否存在相应的方法、成员等（NoSuchMethodError、NoSuchFieldError）。

准备

在准备阶段，虚拟机会在方法区中为 Class 分配内存，并设置 static 成员变量的初始值为默认值。注意这里仅仅会为 static 变量分配内存（static 变量在方法区中），并且初始化 static 变量的值为其所属类型的默认值

解析

解析阶段，虚拟机会将常量池中的符号引用替换为直接引用，解析主要针对的是类、接口、方法、成员变量等符号引用。在转换成直接引用后，会触发校验阶段的符号引用验证，验证转换之后的直接引用是否能找到对应的类、方法、成员变量等。这里也可见类加载的各个阶段在实际过程中，可能是交错执行。

初始化

初始化阶段即开始在内存中构造一个 Class 对象来表示该类

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 DK\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 中类的生命周期是什么?

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 的原理？

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（Gabage 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

28.GC 的回收器有哪些？

1. 并发标记扫描收回器（CMS Collector）
管理新生代方式和 Paralel 和 Serial GC 相同。而是在老年代中并发处理。尽量减少停顿时间。CMS 收集器（Concurrent Mark Sweep）是一种以获取最短回收停顿时间为目标的收集器，是基于“标记-清除”算法
2 .G1 垃圾回收器 G1 (Garbage first)

是 JDK7 的新特性。jdk7 以上都可以自主设置 JVM GC 类型。G1 会将堆内存划分成相互独立的区块（默认 1024），每一块都可能是不连续的 O（old 区）,Y(young 区)区块（相对于 CMS 中 O,Y 区块是连续的）。G1 会第一时间处理垃圾最多的区块。这个是 garbage First 的原因之一。
优点：并发与并行、分代收集、空间整合、可预测的停顿

29.GC 相关参数-调优？

1.分析工具
所有调优都是建立在对堆内存的使用情况有一定的了解的基础上,所以我们首先需要使用一些工具对我们的 java 应用进行监控和分析，来了解到底出现了什么问题：
GC 日志
-XX:+PrintGCDetails：打印 GC 日志
-XX:+PrintGCTimeStamps：GC 日志打印时间戳
堆实时监控
使用 jstat -gcutil PID 1000 指令：每秒打印堆内存实时信息及 GC 情况
2.调整堆大小
考虑维度：
GC 频率：过频繁时需要增大堆大小
单次 GC 耗时：耗时过长需要考虑控制堆大小
GC 在整个时间中所占百分比
FullGC 后剩余可用空间：普遍经验为 70%可用
调优参数：
-Xms=N、-Xmx=N：堆初始值和最大值
-Xmn=N：设置固定新生代大小
-XX:NewRatio=N：设置新生代与老年代空间占用比（此值为分母）
-XX:NewSize=N、-XX:MaxNewSize=N：设置新生代初始和最大值
3.永久代和元空间
元空间默认会使用尽可能多的空间，因此不用过于关注
永久代（jdk7 及之前）
-XX:PermSize=N、-XXMaxPer mSize=N：设置固定永久代大小和最大值
元空间（jdk8 以后）
-XX:MetaspaceSize=N、-XX:MaxMetaspaceSize=N：设置固定元空间大小和最大值
4.控制并发
控制线程数：-XX:ParallelGCThreads=N，影响多线程操作：
Parallel 收集新生代和老年代
CMS：新生代收集（ParNewGC）、并发收集 STW 阶段（非 FullGC）
G1 新生代收集、STW 阶段（非 FullGC）
默认线程数计算（n 为 cpu 线程数）：ParallelGCThreads = 8 + ((N-8) * 5 / 8)
当机器 CPU 线程数比较多，此时默认线程数会比较大，而又同时运行多个 jvm 实例，这时需要手动控制线程数，避免过多垃圾回收线程并发运行。
5.自适应调整
JVM 会自己根据以往的性能历史进行性能参数调整
-XX:-UseAdaptiveSizePolicy：关闭自适应调整功能（默认开启）
-XX:+PrintAdaptiveSizePolicy：打印自动调整信息

30.Java 堆空间内存溢出？

原因：
1.无法在 Java 堆中分配对象
2.吞吐量增加
3.应用程序无意中保存了对象引用，对象无法被 GC 回收
4.应用程序过度使用 finalizer。finalizer 对象不能被 GC 立刻回收。finalizer 由结束队列服务的守护线程调用，有时 finalizer 线程的处理能力无法跟上结束队列的增长。
解决：
1.使用 -Xmx 增加堆大小
2.修复应用程序中的内存泄漏

31.GC 开销超过限制,引起 OOM?

原因：
1. Java 进程 98%的时间在进行垃圾回收，恢复了不到 2%的堆空间，最后连续 5 个（编译时常量）垃圾回收一直如此。
解决：
1. 使用 -Xmx 增加堆大小
2. 使用 -XX:-UseGCOverheadLimit 取消 GC 开销限制

32.请求的数组大引起 OOM,怎么解决？

原因：
1. 应用程序试图分配一个超过堆大小的数组
解决：
1. 使用 -Xmx 增加堆大小
2. 修复应用程序中分配巨大数组的 bug

33.永久代(Perm gen)空间,引起 OOM?

原因：
Perm gen 空间包含：
类的名字、字段、方法
与类相关的对象数组和类型数组
JIT 编译器优化
1.当 Perm gen 空间用尽时，将抛出异常。
通过 JDK bin 目录下自带的 jconsole 工具
解决：
1.使用 -XX: MaxPermSize 增加 Permgen 大小
2.不重启应用部署应用程序可能会导致此问题。重启 JVM 解决

34.Metaspace 元空间耗尽,引起 OOM?

原因：

1.从 Java 8 开始 Perm gen 改成了 Metaspace，在本机内存中分配 class 元数据（称为 metaspace）。如果 metaspace 耗尽，则抛出异常

解决：

1.通过命令行设置 -XX: MaxMetaSpaceSize 增加 metaspace 大小

取消 -XX: maxmetsspacedize

减小 Java 堆大小,为 MetaSpace 提供更多的可用空间

2.为服务器分配更多的内存

3.可能是应用程序 bug，修复 bug

35.无法新建本机线程？

原因：

1. 内存不足，无法创建新线程。由于线程在本机内存中创建，报告这个错误表明本机内存空间不足

解决：

1.为机器分配更多的内存

2.减少 Java 堆空间

3.修复应用程序中的线程泄漏。

4.增加操作系统级别的限制

ulimit -a

用户进程数增大 (-u) 1800

使用 -Xss 减小线程堆栈大小

五、多线程

1.线程调用 start()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。
notityAll():唤醒所有处入等待状态的线程，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的进程产生影响，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，所以多进程的程序要比多线程的程序健壮，但是在进程切换时，耗费的资源较大，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个线程在同时运行，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，在末使用完之前，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，是否存在着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. 什么是自旋锁？

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. 提交任务时,线程池队列已满,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 7 大参数什么意思？

ThreadPoolExecutor(intcorePoolSize,intmaximumPoolSize,longkeepAliveTime,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。

42. 线程的生命周期?

当线程被创建并启动以后，它既不是一启动就进入了执行状态，也不是一直处于执行状态。 在线程的生命周期中，它要经过新建(New)、就绪（Runnable）、运行（Running）、阻塞 (Blocked)和死亡(Dead)5 种状态。尤其是当线程启动以后，它不可能一直"霸占"着 CPU 独自 运行，所以 CPU 需要在多条线程之间切换，于是线程状态也会多次在运行、阻塞之间切换
这 5 种状态如下
1）新建（New）：创建后尚未启动的线程处于这种状态
（2）运行（Runable）：Runable 包括了操作系统线程状态的 Running 和 Ready，也就是处于此状态的线程有可能正在执行，也有可能正在等待着 CPU 为它分配执行时间。
（3）等待（Wating）：处于这种状态的线程不会被分配 CPU 执行时间。等待状态又分为无限期等待和有限期等待，处于无限期等待的线程需要被其他线程显示地唤醒，没有设置 Timeout 参数的 Object.wait()、没有设置 Timeout 参数的 Thread.join()方法都会使线程进入无限期等待状态；有限期等待状态无须等待被其他线程显示地唤醒，在一定时间之后它们会由系统自动唤醒，Thread.sleep()、设置了 Timeout 参数的 Object.wait()、设置了 Timeout 参数的 Thread.join()方法都会使线程进入有限期等待状态。
（4）阻塞（Blocked）：线程被阻塞了，“阻塞状态”与”等待状态“的区别是：”阻塞状态“在等待着获取到一个排他锁,这个时间将在另外一个线程放弃这个锁的时候发生;而”等待状态“则是在等待一段时间或者唤醒动作的发生。在程序等待进入同步区域的时候,线程将进入这种状态。
（5）结束（Terminated）：已终止线程的线程状态，线程已经结束执行。

44.线程的调度方法 wait 和 join 与 notify

JDK 中提供了三个版本的方法，

（1）wait()方法的作用是将当前运行的线程挂起（即让其进入阻塞状态），直到 notify 或 notifyAll 方法来唤醒线程.

（2）wait(long timeout)，该方法与 wait()方法类似，唯一的区别就是在指定时间内，如果没有 notify 或 notifAll 方法的唤醒，也会自动唤醒。

（3）至于 wait(long timeout,long nanos)，本意在于更精确的控制调度时间，不过从目前版本来看，该方法貌似没有完整的实现该功能，其源码(JDK1.8)如下：

有了对 wait 方法原理的理解，notify 方法和 notifyAll 方法就很容易理解了。既然 wait 方式是通过对象的 monitor 对象来实现的，所以只要在同一对象上去调用 notify/notifyAll 方法，就可以唤醒对应对象 monitor 上等待的线程了。notify 和 notifyAll 的区别在于前者只能唤醒 monitor 上的一个线程，对其他线程没有影响，而 notifyAll 则唤醒所有的线程，

join 方法的作用是父线程等待子线程执行完成后再执行，换句话说就是将异步执行的线程合并为同步的线程。JDK 中提供三个版本的 join 方法，其实现与 wait 方法类似，join()方法实际上执行的 join(0)，而 join(long millis, int nanos)也与 wait(long millis, int nanos)的实现方式一致，暂时对纳秒的支持也是不完整的。

45.公平锁与非公平锁?

公平锁

是指多个线程按照申请锁的顺序来获取锁，类似于排队买饭，先来后到，先来先服务，就是公平的，也就是队列

非公平锁

是指多个线程获取锁的顺序，并不是按照申请锁的顺序，有可能申请的线程比先申请的线程优先获取锁，在高并发环境下，有可能造成优先级翻转，或者饥饿的线程（也就是某个线程一直得不到锁），类似于允许排队加塞。。。

如何创建

并发包中 ReentrantLock 的创建可以指定析构函数的 boolean 类型来得到公平锁或者非公平锁，默认是非公平锁

46.可重入锁与不可重入锁?

可重入锁指的是可重复可递归调用的锁，在外层使用锁之后，在内层仍然可以使用，并且不发生死锁（前提是同一个对象或者类）

不可重入锁，与可重入锁相反，不可递归调用，递归调用就会发生死锁。

46. synchronized 的三种用法与区别?

修饰普通方法 一个对象中的加锁方法只允许一个线程访问。但要注意这种情况下锁的是访问该方法的实例对象， 如果多个线程不同对象访问该方法，则无法保证同步。

修饰静态方法 由于静态方法是类方法， 所以这种情况下锁的是包含这个方法的类，也就是类对象；这样如果多个线程不同对象访问该静态方法，也是可以保证同步的。

修饰代码块 其中普通代码块 如 Synchronized（obj） 这里的 obj 可以为类中的一个属性、也可以是当前的对象，它的同步效果和修饰普通方法一样；Synchronized 方法 （obj.class）静态代码块它的同步效果和修饰静态方法类似。

47. CPU 的调度算法?

XEN 的 CPU 调度算法主要有 3 中，BVT(borrowed virtual time) 调度算法，SEDF (Ssimple earliest deadline first)调度算法，Credit 调度算法

50.项目中使用了哪种线程池，使用场景是什么?

在项目中 我们对于数据查询加载使用线程池 比较多，比如商品的详情加载，商品的 sku 信息和 spu 信息 是 根据代码的顺序加载的，而且其中难免会有订单服务，远程调用商品服务查询 sku 信息这种耗时的程序，打个比方，查询商品的订单编号 = 50ms，查询商品的 sku 信息远程调用 =100ms，多个查询 多个远程调用 加 多个数据的封装返回 前端 接收 展示，一套下来 难免有个 2-3s 的 时间 这还不包括别的，总体下来 要有个 5-6s ，但是在现在,去打开一个网页 如果 4s 之内不给我出现响应，我直接就给关了 ，所以这就很考研我们代码的加载性能，我们在项目中对这些 场景 使用线程池，进行异步加载的形式，调用多个线程进行异步加载，大大减少了代码的运行时间。
后台系统实现数据的批量导入，采用线程池实现数据分片处理

51.ReentrantLock 的理解？

ReentantLock 继承接口 Lock 并实现了接口中定义的方法，他是一种可重入锁，除了能完成 synchronized 所能完成的所有工作外，还提供了诸如可响应中断锁、可轮询锁请求、定时锁等避免多线程死锁的方法。
非公平锁
JVM 按随机、就近原则分配锁的机制则称为不公平锁，ReentrantLock 在构造函数中提供了是否公平锁的初始化方式，默认为非公平锁。非公平锁实际执行的效率要远远超出公平锁，除非 程序有特殊需要，否则最常用非公平锁的分配机制。
公平锁
公平锁指的是锁的分配机制是公平的，通常先对锁提出获取请求的线程会先被分配到锁，ReentrantLock 在构造函数中提供了是否公平锁的初始化方式来定义公平锁。

52.Lock 接口的主要方法？

1.void lock(): 执行此方法时，如果锁处于空闲状态，当前线程将获取到锁. 相反，如果锁已经被其他线程持有，将禁用当前线程，直到当前线程获取到锁.
2.boolean tryLock()：如果锁可用，则获取锁，并立即返回 true，否则返回 false. 该方法和 lock()的区别在于，tryLock()只是"试图"获取锁，如果锁不可用，不会导致当前线程被禁用，当前线程仍然继续往下执行代码. 而 lock()方法则是一定要获取到锁，如果锁不可用，就一直等待，在未获得锁之前,当前线程并不继续向下执行.
3.void unlock()：执行此方法时，当前线程将释放持有的锁. 锁只能由持有者释放，如果线程并不持有锁，却执行该方法，可能导致异常的发生.
4.Condition newCondition()：条件对象，获取等待通知组件。该组件和当前的锁绑定， 当前线程只有获取了锁，才能调用该组件的 await()方法，而调用后，当前线程将缩放锁。
5.getHoldCount() ：查询当前线程保持此锁的次数，也就是执行此线程执行 lock 方法的次数。
6.getQueueLength（）：返回正等待获取此锁的线程估计数，比如启动 10 个线程，1 个线程获

得锁，此时返回的是 9

7.getWaitQueueLength：（Condition condition）返回等待与此锁相关的给定条件的线程估计数。比如 10 个线程，用同一个 condition 对象，并且此时这 10 个线程都执行了 condition 对象的 await 方法，那么此时执行此方法返回 10

8.hasWaiters(Condition condition)： 查询是否有线程等待与此锁有关的给定条件(condition)，对于指定 contidion 对象，有多少线程执行了 condition.await 方法

9.hasQueuedThread(Thread thread)：查询给定线程是否等待获取此锁

10.hasQueuedThreads()：是否有线程等待此锁

11.isFair()：该锁是否公平锁

12.isHeldByCurrentThread()： 当前线程是否保持锁锁定，线程的执行 lock 方法的前后分别是 false 和 true

13.isLock()：此锁是否有任意线程占用

14.lockInterruptibly（）：如果当前线程未被中断，获取锁

15.tryLock（）：尝试获得锁，仅在调用时锁未被线程占用，获得锁

16.tryLock(long timeout TimeUnit unit)：如果锁在给定等待时间内没有被另一个线程保持，则获取该锁。

53.ReentrantLock 与 synchronized 的区别？

1.两者的共同点：

1.都是用来协调多线程对共享对象、变量的访问

2.都是可重入锁，同一线程可以多次获得同一个锁

3.都保证了可见性和互斥性

2.两者的不同点：

1.ReentrantLock 显示的获得、释放锁，synchronized 隐式获得释放锁

2.ReentrantLock 可响应中断、可轮回，synchronized 是不可以响应中断的，为处理锁的不可用性提供了更高的灵活性

3.ReentrantLock 是 API 级别的，synchronized 是 JVM 级别的

4.ReentrantLock 可以实现公平锁

5.ReentrantLock 通过 Condition 可以绑定多个条件

6.底层实现不一样， synchronized 是同步阻塞，使用的是悲观并发策略，lock 是同步非阻塞，采用的是乐观并发策略

7.Lock 是一个接口，而 synchronized 是 Java 中的关键字，synchronized 是内置的语言实现。

8.synchronized 在发生异常时，会自动释放线程占有的锁，因此不会导致死锁现象发生； 而 Lock 在发生异常时，如果没有主动通过 unLock()去释放锁，则很可能造成死锁现象， 因此使用 Lock 时需要在 finally 块中释放锁。

9.Lock 可以让等待锁的线程响应中断，而 synchronized 却不行，使用 synchronized 时， 等待的线程会一直等待下去，不能够响应中断。

10.通过 Lock 可以知道有没有成功获取锁，而 synchronized 却无法办到。

11.Lock 可以提高多个线程进行读操作的效率，既就是实现读写锁等。

54.tryLock 和 lock 和 lockInterruptibly 的区别？

1.tryLock 能获得锁就返回 true，不能就立即返回 false，tryLock(long timeout,TimeUnit

unit)，可以增加时间限制，如果超过该时间段还没获得锁，返回 false
2.lock 能获得锁就返回 true，不能的话一直等待获得锁
3.lock 和 lockInterruptibly，如果两个线程分别执行这两个方法，但此时中断这两个线程，lock 不会抛出异常，而 lockInterruptibly 会抛出异常。

55.ReadWriteLock 读写锁是干什么的？

为了提高性能，Java 提供了读写锁，在读的地方使用读锁，在写的地方使用写锁，灵活控制，如果没有写锁的情况下，读是无阻塞的,在一定程度上提高了程序的执行效率。读写锁分为读锁和写锁，多个读锁不互斥，读锁与写锁互斥，这是由 jvm 自己控制的，你只要上好相应的锁即可。
读锁
如果你的代码只读数据，可以很多人同时读，但不能同时写，那就上读锁
写锁
如果你的代码修改数据，只能有一个人在写，且不能同时读取，那就上写锁。总之，读的时候上读锁，写的时候上写锁！
Java 中读写锁有个接 口 java.util.concurrent.locks.ReadWriteLock ，也有 具体的实 现 ReentrantReadWriteLock。

56.共享锁和独占锁是什么？

java 并发包提供的加锁模式分为独占锁和共享锁。
独占锁
独占锁模式下，每次只能有一个线程能持有锁，ReentrantLock 就是以独占方式实现的互斥锁。独占锁是一种悲观保守的加锁策略，它避免了读/读冲突，如果某个只读线程获取锁，则其他读线程都只能等待，这种情况下就限制了不必要的并发性，因为读操作并不会影响数据的一致性。
共享锁
共享锁则允许多个线程同时获取锁，并发访问 共享资源，如：ReadWriteLock。共享锁则是一种乐观锁，它放宽了加锁策略，允许多个执行读操作的线程同时访问共享资源。
1.AQS 的内部类 Node 定义了两个常量 SHARED 和 EXCLUSIVE,他们分别标识 AQS 队列中等待线程的锁获取模式。
2.java 的并发包中提供了 ReadWriteLock，读-写锁。它允许一个资源可以被多个读操作访问，或者被一个 写操作访问，但两者不能同时进行。

57.你对偏向锁是理解是什么？

Hotspot 的作者经过以往的研究发现大多数情况下锁不仅不存在多线程竞争,而且总是由同一线程多次获得。偏向锁的目的是在某个线程获得锁之后，消除这个线程锁重入（CAS）的开销，看起来让这个线程得到了偏护。引入偏向锁是为了在无多线程竞争的情况下尽量减少不必要的轻量级锁执行路径，因为轻量级锁的获取及释放依赖多次 CAS 原子指令，而偏向锁只需要在置换 ThreadID 的时候依赖一次 CAS 原子指令（由于一旦出现多线程竞争的情况就必须撤销偏向锁，所以偏向锁的撤销操作的性能损耗必须小于节省下来的 CAS 原子指令的性能消耗）。上面说过，轻量级锁是为了在线程交替执行同步块时提高性能,而偏向锁则是在只有一个线程执行同步块时进 一步提高性能。

58. Java 中用到的线程调度？

1.抢占式调度：
抢占式调度指的是每条线程执行的时间、线程的切换都由系统控制，系统控制指的是在系统某种运行机制下，可能每条线程都分同样的执行时间片，也可能是某些线程执行的时间片较长，甚至某些线程得不到执行的时间片。在这种机制下，一个线程的堵塞不会导致整个进程堵塞。
2.协同式调度：
协同式调度指某一线程执行完后主动通知系统切换到另一线程上执行，这种模式就像接力赛一样， 一个人跑完自己的路程就把接力棒交接给下一个人，下个人继续往下跑。线程的执行时间由线程本身控制，线程切换可以预知，不存在多线程同步问题，但它有一个致命弱点：如果一个线程编写有问题，运行到一半就一直堵塞，那么可能导致整个系统崩溃。
3.JVM 的线程调度实现（抢占式调度）
java 使用的线程调使用抢占式调度，Java 中线程会按优先级分配 CPU 时间片运行，且优先级越高越优先执行，但优先级高并不代表能独自占用执行时间片，可能是优先级高得到越多的执行时间片，反之，优先级低的分到的执行时间少但不会分配不到执行时间。
4.线程让出 cpu 的情况：
1.当前运行线程主动放弃 CPU，JVM 暂时放弃 CPU 操作（基于时间片轮转调度的 JVM 操作系统不会让线程永久放弃 CPU，或者说放弃本次时间片的执行权），例如调用 yield()方法。
2.当前运行线程因为某些原因进入阻塞状态，例如阻塞在 I/O 上。
3.当前运行线程结束，即运行完 run()方法里面的任务。

59. 进程调度算法有哪些？

1.优先调度算法
1）.先来先服务调度算法（FCFS）
当在作业调度中采用该算法时，每次调度都是从后备作业队列中选择一个或多个最先进入该队列的作业，将它们调入内存，为它们分配资源、创建进程，然后放入就绪队列。在进程调度中采 用 FCFS 算法时，则每次调度是从就绪队列中选择一个最先进入该队列的进程，为之分配处理机，使之投入运行。该进程一直运行到完成或发生某事件而阻塞后才放弃处理机，特点是：算法比较简单，可以实现基本上的公平。
2）.短作业(进程)优先调度算法
短作业优先(SJF)的调度算法是从后备队列中选择一个或若干个估计运行时间最短的作业，将它们 调入内存运行。而短进程优先(SPF)调度算法则是从就绪队列中选出一个估计运行时间最短的进程， 将处理机分配给它，使它立即执行并一直执行到完成，或发生某事件而被阻塞放弃处理机时再重 新调度。该算法未照顾紧迫型作业。

2.高优先权优先调度算法
为了照顾紧迫型作业，使之在进入系统后便获得优先处理，引入了最高优先权优先(FPF)调度 算法。当把该算法用于作业调度时，系统将从后备队列中选择若干个优先权最高的作业装入内存。当用于进程调度时，该算法是把处理机分配给就绪队列中优先权最高的进程。
1）.非抢占式优先权算法
在这种方式下，系统一旦把处理机分配给就绪队列中优先权最高的进程后，该进程便一直执行下去，直至完成；或因发生某事件使该进程放弃处理机时。这种调度算法主要用于批处理系统中；也可用于某些对实时性要求不严的实时系统中。

2）.抢占式优先权调度算法

在这种方式下，系统同样是把处理机分配给优先权最高的进程，使之执行。但在其执行期间，只要又出现了另一个其优先权更高的进程，进程调度程序就立即停止当前进程(原优先权最高的进程) 的执行，重新将处理机分配给新到的优先权最高的进程。显然，这种抢占式的优先权调度算法能 更好地满足紧迫作业的要求，故而常用于要求比较严格的实时系统中，以及对性能要求较高的批 处理和分时系统中。

3）.高响应比优先调度算法

在批处理系统中，短作业优先算法是一种比较好的算法，其主要的不足之处是长作业的运行得不到保证。如果我们能为每个作业引入前面所述的动态优先权，并使作业的优先级随着等待时 间的增加而以速率 a 提高，则长作业在等待一定的时间后，必然有机会分配到处理机。该优先权的变化规律可描述为：

(1) 如果作业的等待时间相同，则要求服务的时间愈短，其优先权愈高，因而该算法有利于短作业。

(2) 当要求服务的时间相同时，作业的优先权决定于其等待时间，等待时间愈长，其优先权愈高，因而它实现的是先来先服务。

(3) 对于长作业，作业的优先级可以随等待时间的增加而提高，当其等待时间足够长时，其优先级便可升到很高，从而也可获得处理机。简言之，该算法既照顾了短作业，又考虑了作业到 达的先后次序，不会使长作业长期得不到服务。因此，该算法实现了一种较好的折衷。当然，在 利用该算法时，每要进行调度之前，都须先做响应比的计算，这会增加系统开销。

3.基于时间片的轮转调度算法

1）.时间片轮转法

在早期的时间片轮转法中，系统将所有的就绪进程按先来先服务的原则排成一个队列，每次调度时，把 CPU 分配给队首进程，并令其执行一个时间片。时间片的大小从几 ms 到几百 ms。当执行的时间片用完时，由一个计时器发出时钟中断请求，调度程序便据此信号来停止该进程的执行，并将它送往就绪队列的末尾；然后，再把处理机分配给就绪队列中新的队首进程，同时也让它执行一个时间片。这样就可以保证就绪队列中的所有进程在一给定的时间内均能获得一时间片的处理机执行时间。

2）.多级反馈队列调度算法

(1) 应设置多个就绪队列，并为各个队列赋予不同的优先级。第一个队列的优先级最高，第二个队列次之，其余各队列的优先权逐个降低。该算法赋予各个队列中进程执行时间片的大小也各不相同，在优先权愈高的队列中，为每个进程所规定的执行时间片就愈小。例如，第二个队列的时间片要比第一个队列的时间片长一倍，……，第 i+1 个队列的时间片要比第 i 个队列的时间片长一倍。

(2) 当一个新进程进入内存后，首先将它放入第一队列的末尾，按 FCFS 原则排队等待调度。当轮到该进程执行时，如它能在该时间片内完成，便可准备撤离系统；如果它在一个时间片结束时尚未完成，调度程序便将该进程转入第二队列的末尾，再同样地按 FCFS 原则等待调度执行；如果它在第二队列中运行一个时间片后仍未完成，再依次将它放入第三队列，……，如此下去，当一个长作业(进程)从第一队列依次降到第 n 队列后，在第 n 队列便采取按时间片轮转的方式运行。

(3) 仅当第一队列空闲时，调度程序才调度第二队列中的进程运行；仅当第 1～(i-1)队列均空时， 才会调度第 i 队列中的进程运行。如果处理机正在第 i 队列中为某进程服务时，又有新进程进入优先权较高的队列(第 1～(i-1)中的任何一个队列)，则此时新进程将抢占正在运行进程的处理机，即由调度程序把正在运行的进程放回到第 i 队列的末尾，把处理机分配给新到的高优先权进程。

在多级反馈队列调度算法中,如果规定第一个队列的时间片略大于多数人机交互所需之处理时间时，便能够较好的满足各种类型用户的需要。

60.什么 CAS?

CAS（Compare And Swap/Set）比较并交换，CAS 算法的过程是这样：它包含 3 个参数 CAS(V,E,N)。V 表示要更新的变量(内存值)，E 表示预期值(旧的)，N 表示新值。当且仅当 V 值等于 E 值时，才会将 V 的值设为 N,如果 V 值和 E 值不同,则说明已经有其他线程做了更新，则当前线程什么都不做。最后，CAS 返回当前 V 的真实值。

CAS 操作是抱着乐观的态度进行的(乐观锁)，它总是认为自己可以成功完成操作。当多个线程同时使用 CAS 操作一个变量时，只有一个会胜出，并成功更新，其余均会失败。失败的线程不会被挂起，仅是被告知失败，并且允许再次尝试，当然也允许失败的线程放弃操作。基于这样的原理， CAS 操作即使没有锁，也可以发现其他线程对当前线程的干扰，并进行恰当的处理。

61.为什么 wait\notify 是在 Object 而不是 Thread 中?

1)wait 和 notify 是 Java 中两个线程之间的通信机制。

对语言设计者而言,如果不能通过 synchronized 实现通信此机制,

同时又要确保这个机制对每个对象可用，那么 Object 类则是的正确声明位置

2)每个对象都可上锁,这是在 Object 类而不是 Thread 类中声明 wait 和 notify 的另一个原因。

3)在 Java 中为了进入代码的临界区,线程需要锁定并等待锁定,他们不知道哪些线程持有锁,而只是知道锁被某个线程持有,并且他们应该等待取得锁,而不是去了解哪个线程在同步块内,并请求它们释放锁定。

4)Java 是基于 Hoare 的监视器的思想。所有对象都有一个监视器。

线程在监视器上等待为执行等待，我们需要 2 个参数：

一个线程,一个监视器(任何对象)

在 Java 设计中，线程不能被指定，它总是运行当前代码的线程。

但是我们可以指定监视器(这是我们称之为等待的对象)

62.锁消除和锁粗化的理解是什么?

锁消除

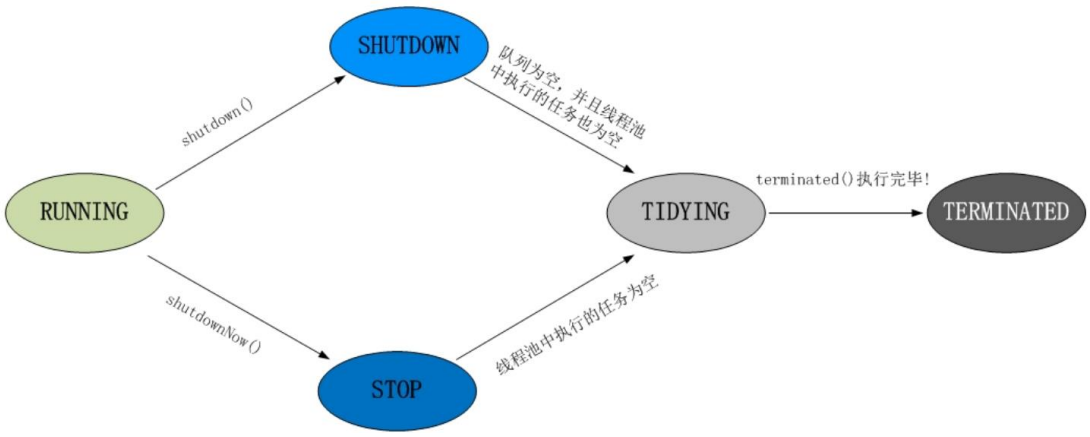
锁消除就是字面意思，虚拟机会根据自己的代码检测结果取消一些加锁逻辑。虚拟机通过检测会发现一些代码中不可能出现数据竞争，但是代码中又有加锁逻辑，为了提高性能，就消除这些锁。如果一段代码中，在堆上的所有数据都不会被其他线程访问到，那就可以把它们当成线程私有数据，自然就不需要同步加锁了。

锁粗化

原则上，我们在编写代码的时候，总是推荐将同步块的作用范围限制得尽量小---只在共享数据的实际作用域内才进行同步,这样是为了使得需要同步的操作数量尽可能变少,即使存在锁竞争,等待锁的线程也能进可能快地拿到锁。

大多数情况下,上面的原则都是对的,但是如果一系列的连续操作都对同一个对象反复加锁解锁,甚至加锁操作时出现在循环体中，那即使没有线程竞争，频繁地进行互斥同步操作也会导致不必要的性能损耗。

63. 线程池的五大状态？



1、RUNNING

状态说明：线程池处在 RUNNING 状态时，能够接收新任务，以及对已添加的任务进行处理。

状态切换：线程池的初始化状态是 RUNNING。换句话说，线程池被一旦被创建，就处于 RUNNING 状态，并且线程池中的任务数为 0!

2、SHUTDOWN

状态说明：线程池处在 SHUTDOWN 状态时，不接收新任务，但能处理已添加的任务。

状态切换：调用线程池的 shutdown()接口时，线程池由 RUNNING -> SHUTDOWN。

3、STOP

状态说明：线程池处在 STOP 状态时，不接收新任务，不处理已添加的任务，并且会中断正在处理的任务。

状态切换：调用线程池的 shutdownNow()接口时，线程池由(RUNNING or SHUTDOWN) -> STOP。

4、TIDYING

状态说明：当所有的任务已终止，ctl 记录的”任务数量”为 0，线程池会变为 TIDYING 状态。

当线程池变为 TIDYING 状态时，会执行钩子函数 terminated()。terminated()在 ThreadPoolExecutor 类中是空的，若用户想在线程池变为 TIDYING 时，进行相应的处理；可以通过重载 terminated()函数来实现。

状态切换：当线程池在 SHUTDOWN 状态下，阻塞队列为空并且线程池中执行的任务也为空时，就会由 SHUTDOWN -> TIDYING。

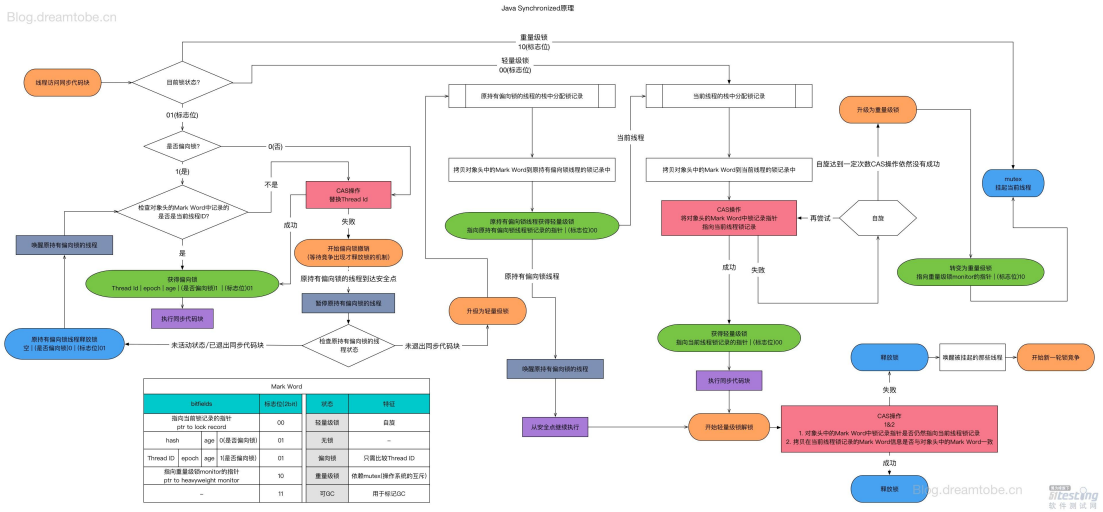
当线程池在 STOP 状态下，线程池中执行的任务为空时，就会由 STOP -> TIDYING。

5、TERMINATED

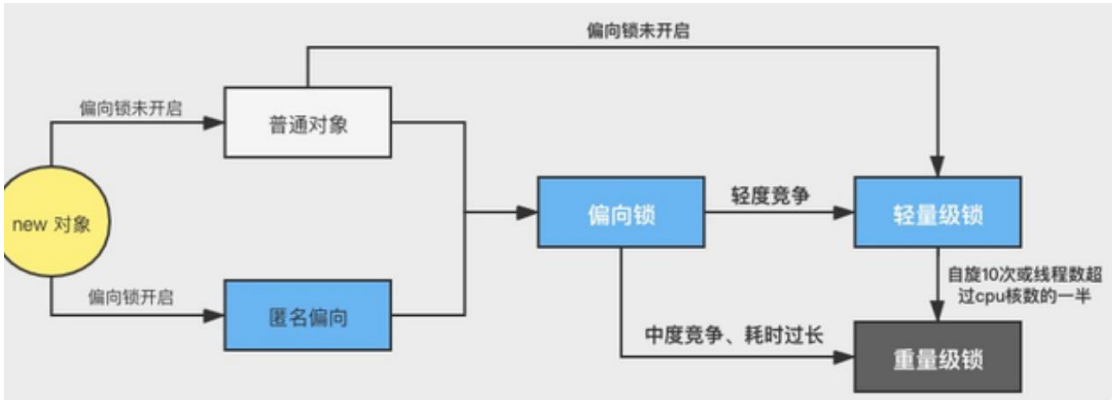
状态说明：线程池彻底终止，就变成 TERMINATED 状态。

状态切换：线程池处在 TIDYING 状态时，执行完 terminated()之后，就会由 TIDYING -> TERMINATED。

64. Synchonized 的实现原理？



65. Java 中锁的升级过程？



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、IODH

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，在运行时刻是可以变化的，更具动态性。GoF 中除了以上 4 种，其他的都是对象模式。

7.设计模式的每种模式的功能？

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多种访问方式，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。

8.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，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。

9.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();
}
//适配者接口
class Adaptee
{
    public void specificRequest()
    {
        System.out.println("适配者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();
    }
}
```

10. 享元模式

意图：运用共享技术有效地支持大量细粒度的对象。

主要解决：在有大量对象时，有可能会造成内存溢出，我们把其中共同的部分抽象出来，如果有相同的业务请求，直接返回在内存中已有的对象，避免重新创建。

何时使用： 1、系统中有大量对象。 2、这些对象消耗大量内存。 3、这些对象的状态大部分可以外部化。 4、这些对象可以按照内蕴状态分为很多组，当把外蕴对象从对象中剔除出来时，每一组对象都可以用一个对象来代替。 5、系统不依赖于这些对象身份，这些对象是不可分辨的。

如何解决：用唯一标识码判断，如果在内存中有，则返回这个唯一标识码所标识的对象。

关键代码：用 HashMap 存储这些对象。

应用实例： 1、JAVA 中的 String，如果有则返回，如果没有则创建一个字符串保存在字符串缓存池里面。 2、数据库的数据池。

优点：大大减少对象的创建，降低系统的内存，使效率提高。

缺点：提高了系统的复杂度，需要分离出外部状态和内部状态，而且外部状态具有固有化的性质，不应该随着内部状态的变化而变化，否则会造成系统的混乱。

使用场景： 1、系统有大量相似对象。 2、需要缓冲池的场景。

注意事项： 1、注意划分外部状态和内部状态，否则可能会引起线程安全问题。 2、这些类必须有一个工厂对象加以控制。

11.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();
    }
}
```

12. 代理模式是什么？

代理模式给某一个对象提供一个代理对象，并由代理对象控制对原对象的引用。通俗的来讲代理模式就是我们生活中常见的中介。

中介隔离作用：在某些情况下，一个客户类不想或者不能直接引用一个委托对象，而代理类对象可以在客户类和委托对象之间起到中介的作用，其特征是代理类和委托类实现相同的接口。

开闭原则，增加功能：代理类除了是客户类和委托类的中介之外，我们还可以通过给代理类增加额外的功能来扩展委托类的功能，这样做我们只需要修改代理类而不需要再修改委托类，符合代码设计的开闭原则。代理类主要负责为委托类预处理消息、过滤消息、把消息转发给委托类，以及事后对返回结果的处理等。代理类本身并不真正实现服务，而是同过调用委托类的相关方法，来提供特定的服务。真正的业务功能还是由委托类来实现，但是可以在业务功能执行的前后加入一些公共的服务。例如我们想给项目加入缓存、日志这些功能，我们就可以使用代理类来完成，

而没必要打开已经封装好的委托类。

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。

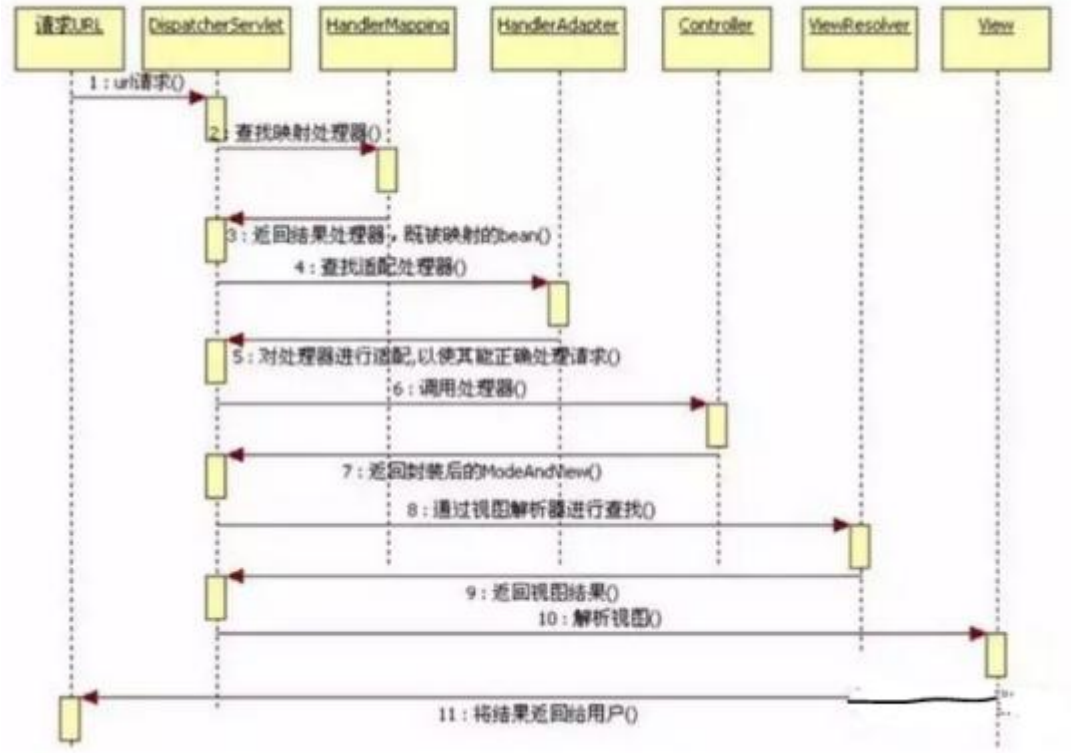
2.MyBatis 的优点？

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 ViewReslover 视图解析器
9、ViewReslover 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
p 标签注入
注解注入

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。
JDK 动态代理、CGLIB

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，逐一定义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()。
如果 bean 实现 BeanFactoryAware 接口,工厂通过传递自身的实例来调用 setBeanFactory()。
4.如果存在与 bean 关联的任何 BeanPostProcessors 则调 preProcessBeforeInitialization()方法。
5.如果为 bean 指定了 init 方法(<bean>的 init-method 属性)那么将调用它。
6.最后，如果存在与 bean 关联的任何 BeanPostProcessors，则将调用 postProcessAfterInitialization()方法。
7.如果 bean 实现 DisposableBean 接口，当 spring 容器关闭时，会调用 destory()。
8.如果为 bean 指定了 destroy 方法（ <bean> 的 destroy-method 属性），那么将调用它。

14.Spring 框架中都用到了哪些设计模式？

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。
模板模式， 用来解决重复性代码。
前端控制器，Spring 提供了 DispatcherSerclet 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，它是惯穿于 BeanFactory/ApplicationContext 接口的核心理念。

15.Spring 中的事件处理？

1、Spring 的核心是 ApplicatonContext,它负责管理 bean 的完整的生命周期。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的该消息，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 Sping 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，代码污染级别级别。

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 实例的实力化,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 Sesisonn，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：

lt;List&: 该标签用来装配 有重复值的 list 值

lt;set&: 该标签用来装配没有重复值的 set 值

lt;map&：该标签科以用来注入键值对

lt;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、它为编程式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，同时在所有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 ModeMap 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 在是在调用时初始化，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。所以可以拼接 SQL 关键字

使用#{}可以有效的防止 SQL 注入，提高系统安全性。

32.Spring 中@Autowire 与@Resource 的区别?

@Autowire 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 IOC,什么是 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. Spring 的循环依赖?

候选者：首先A对象实例化，然后对属性进行注入，发现依赖B对象

候选者：B对象此时还没创建出来，所以转头去实例化B对象

候选者：B对象实例化之后，发现需要依赖A对象，那A对象已经实例化了嘛，所以B对象最终能完成创建

候选者：B对象返回到A对象的属性注入的方法上，A对象最终完成创建

候选者：上面就是大致的过程；

面试官：**听起来你还会原理哦?**

候选者：Absolutely

候选者：至于原理，其实就是用到了三级的缓存

候选者：所谓的三级缓存其实就是三个Map...首先明确一定，我对这里的三级缓存定义是这样的：

候选者：singletonObjects（一级，日常实际获取Bean的地方）；

候选者：earlySingletonObjects（二级，还没进行属性注入，由三级缓存放进来）；

候选者：singletonFactories（三级，Value是一个对象工厂）；

候选者：再回到刚才讲述的过程中，A对象实例化之后，属性注入之前，其实会把A对象放入三级缓存中

候选者：key是BeanName，Value是ObjectFactory

候选者：等到A对象属性注入时，发现依赖B，又去实例化B时

候选者：B属性注入需要去获取A对象，这里就是从三级缓存里拿出ObjectFactory，从ObjectFactory得到对应的Bean（就是对象A）

候选者：把三级缓存的A记录给干掉，然后放到二级缓存中

候选者：显然，二级缓存存储的key是BeanName，value就是Bean（这里的Bean还没做完属性注入相关的工作）

候选者：等到完全初始化之后，就会把二级缓存给remove掉，塞到一级缓存中

候选者：我们自己去getBean的时候，实际上拿到的是一级缓存的

候选者：大致的过程就是这样

面试官：**那我想问一下，为什么是三级缓存?**

候选者：首先从第三级缓存说起（就是key是BeanName，Value为ObjectFactory）

37. Spring 单例的 Bean 是线程安全的吗?

结论： 不是线程安全的

Spring 容器中的 Bean 是否线程安全，容器本身并没有提供 Bean 的线程安全策略，因此可以说 Spring 容器中的 Bean 本身不具备线程安全的特性，但是具体还是要结合具体 scope 的 Bean 去研究

38.Mybatis 的缓存策略：一级缓存与二级缓存?

一级缓存：

SqlSession(连接对象)级别的缓存，同一个 SqlSession 的发起多次同构查询，会将数据保存在一级缓存中。

在 sqlsession 中有一个数据结构是 map 结构，这个区域就是一级缓存区域，一级缓存区域中的 key 是由 sql 语句方法参数 statement 等组成的一个唯一值对应的 value 就是缓存内容，一级缓存 map 的生命周期和当前 sqlSession 一致

注意：无需任何配置，默认开启一级缓存

二级缓存：

SqlSessionFactory(连接对象的工厂)级别的缓存，同一个 SqlSessionFactory 构建的 SqlSession 发起的多次同构查询，会将数据保存在二级缓存中。

二级缓存指的是同一个 namespace 下的 mapper 二级缓存的数据结构,也是一个 map 二级缓存 需要 手动开启

39.@Controller 与@RestController 的区别?

@RestController 注解相当于@ResponseBody ＋ @Controller 合在一起的作用。

1) 如果只是使用@RestController 注解 Controller,则 Controller 中的方法无法返回 jsp 页面，或者 html，配置的视图解析器 InternalResourceViewResolver 不起作用，返回的内容就是 Return 里的内容。

2) 如果需要返回到指定页面，则需要用 @Controller 配合视图解析器 InternalResourceViewResolver 才行。

如果需要返回 JSON，XML 或自定义 mediaType 内容到页面，则需要在对应的方法上加上 @ResponseBody 注解。

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
Reids 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.Redis 和数据库的数据一致性如何保证？

先解释一下 redis 和 数据库 怎么互动的 就是先 查 redis 没数据 去查数据库 存到 reids 然后如何保证？
首先 对于数据， 如果你选择强一致性，那就不能放缓存，所以，我们也就是仅仅能够保证最终的一致性，如果要保证强一致性，那还是别用缓存了，我们尽量的去保证 redis 和数据库 数据一致，只是说降低概率发生，而不能完全的避免，但是我们还是要说。

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、Bitmap、geospatial、hyperloglog

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：一个操作的不可以再分，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。

1.缓存雪崩

缓存雪崩我们可以简单的理解为：由于原有缓存失效，新缓存未到期间所有原本应该访问缓存的请求都去查询数据库了，而对数据库 CPU 和内存造成巨大压力，严重的会造成数据库宕机。从而形成一系列连锁反应，造成整个系统崩溃。一般有三种处理办法：
1.一般并发量不是特别多的时候，使用最多的解决方案是加锁排队。
2.给每一个缓存数据增加相应的缓存标记，记录缓存的是否失效，如果缓存标记失效，则更新数据缓存。
3.为 key 设置不同的缓存失效时间。

2.缓存穿透

缓存穿透是指用户查询数据，在数据库没有，自然在缓存中也不会有。这样就导致用户查询的时候，在 缓存中找不到，每次都要去数据库再查询一遍，然后返回空（相当于进行了两次无用的查询）。这样请 求就绕过缓存直接查数据库，这也是经常提的缓存命中率问题。有很多种方法可以有效地解决缓存穿透问题，最常见的则是采用布隆过滤器，将所有可能存在的数据哈 希到一个足够大的 bitmap 中，一个一定不存在的数据会被这个 bitmap 拦截掉，从而避免了对底层存储系统的查询压力。另外也有一个更为简单粗暴的方法，如果一个查询返回的数据为空（不管是数据不 存在，还是系统故障），我们仍然把这个空结果进行缓存，但它的过期时间会很短，最长不超过五分钟。 通过这个直接设置的默认值存放到缓存，这样第二次到缓冲中获取就有值了，而不会继续访问数据库。

3.缓存预热

缓存预热就是系统上线后，将相关的缓存数据直接加载到缓存系统。这样就可以避免在用户请求的时候，先查询数据库，然后再将数据缓存的问题！用户直接查询事先被预热的缓存数据！

4.缓存更新

缓存更新除了缓存服务器自带的缓存失效策略之外（Redis 默认的有 6 中策略可供选择），我们还可以根据具体的业务需求进行自定义的缓存淘汰，常见的策略有两种：

（1）定时去清理过期的缓存；

（2）当有用户请求过来时，再判断这个请求所用到的缓存是否过期，过期的话就去底层系统得到新数据并更新缓存。

5.缓存降级

当访问量剧增、服务出现问题（如响应时间慢或不响应）或非核心服务影响到核心流程的性能时，仍然　需要保证服务还是可用的，即使是有损服务。系统可以根据一些关键数据进行自动降级，也可以配置开　关实现人工降级。降级的最终目的是保证核心服务可用，即使是有损的。而且有些服务是无法降级的

（如加入购物车、结算）。

18.Redis 的过期策略？

key 的生存时间到了，Redis 会立即删除吗？不会立即删除。

有以下几种策略：

定期删除：Redis 每隔一段时间就去会去查看 Redis 设置了过期时间的 key，会再 100ms 的间隔中默认查看 3 个 key。

惰性删除：如果当你去查询一个已经过了生存时间的 key 时，Redis 会先查看当前 key 的生存时间，是否已经到了，直接删除当前 key，并且给用户返回一个空值。

19.Redis 是如何发现 hot 和 bigkey 的？

1.事前-预判在业务开发阶段，就要对可能变成 hot key ，big key 的数据进行判断，提前处理，这需要的是对产品业务的理解，对运营节奏的把握，对数据设计的经验。

2.事中-监控和自动处理

监控

在应用程序端，对每次请求 redis 的操作进行收集上报;不推荐，但是在运维资源缺少的场景下可以考虑。开发可以绕过运维搞定）；

在 proxy 层，对每一个 redis 请求进行收集上报;（推荐，改动涉及少且好维护）；

对 redis 实例使用 monitor 命令统计热点 key（不推荐，高并发条件下会有造成 redis 内存爆掉的隐患）；

机器层面，Redis 客户端使用 TCP 协议与服务端进行交互，通信协议采用的是 RESP。如果站在机器的角度，可以通过对机器上所有 Redis 端口的 TCP 数据包进行抓取完成热点 key 的统计（不推荐，公司每台机器上的基本组件已经很多了，别再添乱了）；

自动处理

通过监控之后，程序可以获取 big key 和 hot key，再报警的同时，程序对 big key 和 hot key 进行自动处理。或者通知程序猿利用一定的工具进行定制化处理（在程序中对特定的 key 执行前面提到的解决方案）

20.RedLock（红锁）分布式实现原理？

Redlock 是一种算法，Redlock 也就是 Redis Distributed Lock，可用实现多节点 redis 的分布式锁。
RedLock 官方推荐，Redisson 完成了对 Redlock 算法封装。
此种方式具有以下特性：
互斥访问：即永远只有一个 client 能拿到锁
避免死锁：最终 client 都可能拿到锁，不会出现死锁的情况，即使锁定资源的服务崩溃或者分区，仍然能释放锁。
容错性：只要大部分 Redis 节点存活（一半以上），就可以正常提供服务
RedLock 原理（了解）
获取当前 Unix 时间，以毫秒为单位。
依次尝试从 N 个实例，使用相同的 key 和随机值获取锁。在步骤 2，当向 Redis 设置锁时,客户端应该设置一个网络连接和响应超时时间，这个超时时间应该小于锁的失效时间。例如你的锁自动失效时间为 10 秒，则超时时间应该在 5-50 毫秒之间。这样可以避免服务器端 Redis 已经挂掉的情况下，客户端还在死死地等待响应结果。如果服务器端没有在规定时间内响应，客户端应该尽快尝试另外一个 Redis 实例。
客户端使用当前时间减去开始获取锁时间（步骤 1 记录的时间）就得到获取锁使用的时间。当且仅当从大多数（这里是 3 个节点）的 Redis 节点都取到锁，并且使用的时间小于锁失效时间时，锁才算获取成功。
如果取到了锁,key 的真正有效时间等于有效时间减去获取锁所使用的时间(步骤 3 计算的结果)。
如果因为某些原因，获取锁失败（没有在至少 N/2+1 个 Redis 实例取到锁或者取锁时间已经超过了有效时间），客户端应该在所有的 Redis 实例上进行解锁（即便某些 Redis 实例根本就没有加锁成功）。

21.基于 Redis 分布式锁？

1.获取锁的时候，使用 setnx（SETNX key val：当且仅当 key 不存在时，set 一个 key 为 val 的字符串，返回 1；若 key 存在，则什么都不做，返回 0）加锁，锁的 value 值为一个随机生成的 UUID，在释放锁的时候进行判断。并使用 expire 命令为锁添加一个超时时间，超过该时间则自动释放锁。
2.获取锁的时候调用 setnx，如果返回 0，则该锁正在被别人使用，返回 1 则成功获取锁。还设置一个获取的超时时间，若超过这个时间则放弃获取锁。
3.释放锁的时候，通过 UUID 判断是不是该锁，若是该锁，则执行 delete 进行锁释放。

23.Mysql 与 Redis 的数据一致性？

例如如果一个事务执行失败回滚了,但是如果采取了先写 Redis 的方式,就会造成 Redis 和 MySQL 数据库的不一致，再比如说，一个事务写入了 MySQL，但是此时还未写入 Redis，如果这时候有用户访问 Redis，则此时就会出现数据不一致。解决方案如下。

1、分别处理

针对某些对数据一致性要求不是特别高的情况下，可以将这些数据放入 Redis，请求来了直接查询 Redis，例如近期回复、历史排名这种实时性不强的业务。而针对那些强实时性的业务，例如虚拟货币、物品购买件数等等，则直接穿透 Redis 至 MySQL 上，等到 MySQL 上写入成功，再

同步更新到 Redis 上去。这样既可以起到 Redis 的分流大量查询请求的作用，又保证了关键数据的一致性。

2、高并发情况下

此时如果写入请求较多，则直接写入 Redis 中去，然后间隔一段时间，批量将所有的写入请求，刷新到 MySQL 中去；如果此时写入请求不多，则可以在每次写入 Redis，都立刻将该命令同步至 MySQL 中去。这两种方法有利有弊，需要根据不同的场景来权衡。

3、基于订阅 binlog 的同步机制

阿里巴巴的一款开源框架 canal，提供了一种发布/ 订阅模式的同步机制，通过该框架我们可以对 MySQL 的 binlog 进行订阅，这样一旦 MySQL 中产生了新的写入、更新、删除等操作，就可以把 binlog 相关的消息推送至 Redis，Redis 再根据 binlog 中的记录，对 Redis 进行更新。值得注意的是，binlog 需要手动打开，并且不会记录关于 MySQL 查询的命令和操作。

其实这种机制，很类似 MySQL 的主从备份机制，因为 MySQL 的主备也是通过 binlog 来实现的数据一致性。而 canal 正是模仿了 slave 数据库的备份请求，使得 Redis 的数据更新达到了相同的效果。如下图就可以看到 Slave 数据库中启动了 2 个线程，一个是 MySQL SQL 线程，这个线程跟 Matser 数据库中起的线程是一样的，负责 MySQL 的业务率执行，而另外一个线程就是 MySQL 的 I/O 线程，这个线程的主要作用就是同步 Master 数据库中的 binlog，达到数据备份的效果。而 binlog 就可以理解为一堆 SQL 语言组成的日志。

24.什么是 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。

25.Nginx 相对 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

26.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

27.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

28.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。

29.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。

30.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

31.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，

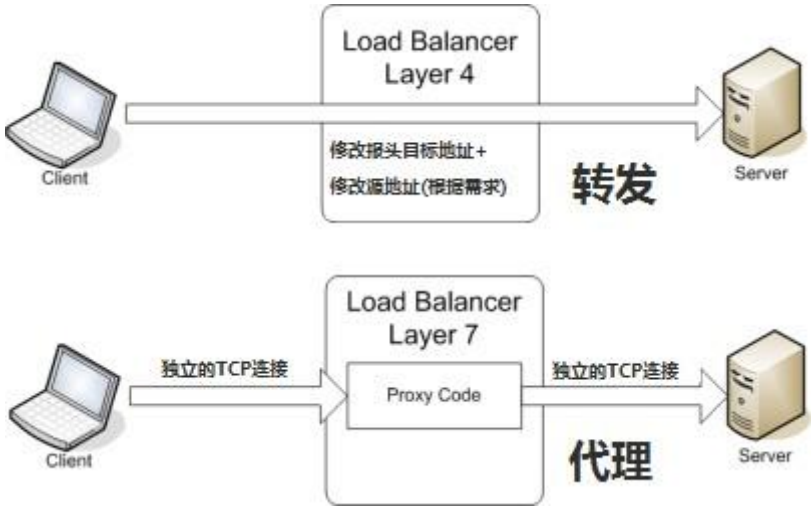
内置的健康检查功能：如果有一个服务器宕机，会做一个健康检查，再发送的请求就不会发送到宕机的服务器了。重新将请求提交到其他的节点上。

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）

32.四层负载均衡与七层负载均衡



1.四层负载均衡（目标地址和端口交换）

主要通过报文中的目标地址和端口，再加上负载均衡设备设置的服务器选择方式，决定最终选择的内部服务器。

以常见的 TCP 为例，负载均衡设备在接收到第一个来自客户端的 SYN 请求时，即通过上述方式选择一个最佳的服务器，并对报文中目标 IP 地址进行修改(改为后端服务器 IP），直接转发给该服务器。TCP 的连接建立，即三次握手是客户端和服务器直接建立的，负载均衡设备只是起到一个类似路由器的转发动作。在某些部署情况下，为保证服务器回包可以正确返回给负载均衡设备，在转发报文的同时可能还会对报文原来的源地址进行修改。实现四层负载均衡的软件有：

F5：硬件负载均衡器，功能很好，但是成本很高。lvs：重量级的四层负载软件。
nginx：轻量级的四层负载软件，带缓存功能，正则表达式较灵活。haproxy：模拟四层转发，较灵活。

2.七层负载均衡（内容交换）

所谓七层负载均衡，也称为“内容交换”，也就是主要通过报文中的真正有意义的应用层内容，再加上负载均衡设备设置的服务器选择方式，决定最终选择的内部服务器。

七层应用负载的好处，是使得整个网络更智能化。例如访问一个网站的用户流量，可以通过七层的方式，将对图片类的请求转发到特定的图片服务器并可以使用缓存技术；将对文字类的请求可以转发到特定的文字服务器并可以使用压缩技术。实现七层负载均衡的软件有：
haproxy：天生负载均衡技能，全面支持七层代理，会话保持，标记，路径转移； nginx：只在 http 协议和 mail 协议上功能比较好，性能与 haproxy 差不多； apache：功能较差
Mysql proxy：功能尚可。

33.负载均衡策略/算法有哪些？

1.轮循均衡（Round Robin）
每一次来自网络的请求轮流分配给内部中的服务器，从 1 至 N 然后重新开始。此种均衡算法适合于服务器组中的所有服务器都有相同的软硬件配置并且平均服务请求相对均衡的情况。

2.权重轮循均衡（Weighted Round Robin）
根据服务器的不同处理能力，给每个服务器分配不同的权值，使其能够接受相应权值数的服务请求。例如：服务器 A 的权值被设计成 1，B 的权值是 3，C 的权值是 6，则服务器 A、B、C 将分别接受到 10%、30%、60%的服务请求。此种均衡算法能确保高性能的服务器得到更多的使用率，避免低性能的服务器负载过重。
3.随机均衡（Random）
把来自网络的请求随机分配给内部中的多个服务器。
4.权重随机均衡（Weighted Random）
此种均衡算法类似于权重轮循算法，不过在处理请求分担时是个随机选择的过程。
5.响应速度均衡（Response Time 探测时间）
负载均衡设备对内部各服务器发出一个探测请求（例如 Ping），然后根据内部中各服务器对探测请求的最快响应时间来决定哪一台服务器来响应客户端的服务请求。此种均衡算法能较好的反映服务器的当前运行状态，但这最快响应时间仅仅指的是负载均衡设备与服务器间的最快响应时间，而不是客户端与服务器间的最快响应时间。
6.最少连接数均衡（Least Connection）
最少连接数均衡算法对内部中需负载的每一台服务器都有一个数据记录，记录当前该服务器正在处理的连接数量，当有新的服务连接请求时，将把当前请求分配给连接数最少的服务器，使均衡更加符合实际情况，负载更加均衡。此种均衡算法适合长时处理的请求服务，如 FTP。
7.处理能力均衡（CPU、内存）
此种均衡算法将把服务请求分配给内部中处理负荷（根据服务器 CPU 型号、CPU 数量、内存大小及当前连接数等换算而成）最轻的服务器，由于考虑到了内部服务器的处理能力及当前网络运行状况，所以此种均衡算法相对来说更加精确，尤其适合运用到第七层（应用层）负载均衡的情况下。

8.DNS 响应均衡（Flash DNS）

在此均衡算法下，分处在不同地理位置的负载均衡设备收到同一个客户端的域名解析请求，并在同一时间内把此域名解析成各自相对应服务器的 IP 地址并返回给客户端，则客户端将以最先收到的域名解析 IP 地址来继续请求服务，而忽略其它的 IP 地址响应。在种均衡策略适合应用在全局负载均衡的情况下，对本地负载均衡是没有意义的。

9.哈希算法

一致性哈希一致性 Hash，相同参数的请求总是发到同一提供者。当某一台提供者挂时，原本发往该提供者的请求，基于虚拟节点，平摊到其它提供者，不会引起剧烈变动。

10.IP 地址散列（保证客户端服务器对应关系稳定）

通过管理发送方 IP 和目的地 IP 地址的散列，将来自同一发送方的分组(或发送至同一目的地的分组)统一转发到相同服务器的算法。当客户端有一系列业务需要处理而必须和一个服务器反复通信时，该算法能够以流(会话)为单位，保证来自相同客户端的通信能够一直在同一服务器中进行处理。

11.URL 散列

通过管理客户端请求 URL 信息的散列，将发送至相同 URL 的请求转发至同一服务器的算法。

34.Nginx 如何实现反向代理负载均衡？

普通的负载均衡软件，（如 LVS）其实现的功能只是对请求数据包的转发、传递，从负载均衡下的节点服务器来看，接收到的请求还是来自访问负载均衡器的客户端的真实用户；而反向代理就不一样了，反向代理服务器在接收访问用户请求后，会代理用户 重新发起请求代理下的节点服务器， 最后把数据返回给客户端用户。在节点服务器看来，访问的节点服务器的客户端用户就是反向代理服务器，而非真实的网站访问用户。

35.什么是正向代理?

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

36.什么是反向代理?

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

37.什么是负载均衡?

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

38.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。

39.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_upstraem 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。

40.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 个以上话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

41.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。

42.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

43.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。

44.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。

45.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。

46.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

47.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

48.如何保证消息的幂等性？

1.生成者不重复发送消息到 MQ

mq 内部可以为每条消息生成一个全局唯一、与业务无关的消息 id，当 mq 接收到消息时，会先根据该 id 判断消息是否重复发送，mq 再决定是否接收该消息。

2.消费者不重复消费

消费者怎么保证不重复消费的关键在于消费者端做控制，因为 MQ 不能保证不重复发送消息，所以应该在消费者端控制：即使 MQ 重复发送了消息，消费者拿到了消息之后，要判断是否已经消费过，如果已经消费，直接丢弃。所以根据实际业务情况，有下面几种方式：

（1）、如果从 MQ 拿到数据是要存到数据库，那么可以根据数据创建唯一约束，这样的话，同样的数据从 MQ 发送过来之后，当插入数据库的时候，会报违反唯一约束，不会插入成功的。（或者可以先查一次，是否在数据库中已经保存了，如果能查到，那就直接丢弃就好了）。

（2）、让生产者发送消息时，每条消息加一个全局的唯一 id，然后消费时，将该 id 保存到 redis 里面。消费时先去 redis 里面查一下有么有，没有再消费。（其实原理跟第一点差不多）。

（3）、如果拿到的数据是直接放到 redis 的 set 中的话，那就不用考虑了，因为 set 集合就是自动有去重的。

49.什么是 RabbitMQ 的死信队列？

先从概念解释上搞清楚这个定义，死信，顾名思义就是无法被消费的消息，字面意思可以这样理解，一般来说，producer 将消息投递到 broker 或者直接到 queue 里了，consumer 从 queue 取出消息进行消费，但某些时候由于特定的原因导致 queue 中的某些消息无法被消费，这样的消息如果没有后续的处理，就变成了死信，有死信，自然就有了死信队列；

产生死信的原因

对 rabbitmq 来说，产生死信的来源大致有如下几种：

消息被拒绝（basic.reject 或 basic.nack）并且 requeue=false.

消息 TTL 过期

队列达到最大长度（队列满了，无法再添加数据到 mq 中）

50.什么是延迟队列,延迟队列的实现方式

延时队列，首先，它是一种队列，队列意味着内部的元素是有序的，元素出队和入队是有方向性的，元素从一端进入，从另一端取出。

其次，延时队列，最重要的特性就体现在它的延时属性上，跟普通的队列不一样的是，普通队列中的元素总是等着希望被早点取出处理，而延时队列中的元素则是希望被在指定时间得到取出和处理，所以延时队列中的元素是都是带时间属性的，通常来说是需要被处理的消息或者任务。

简单来说，延时队列就是用来存放需要在指定时间被处理的元素的队列。

1，利用 TTL(过期时间)+死信队列

生产者生产一条延时消息，根据需要延时时间的不同，利用不同的 routingkey 将消息路由到不同的延时队列，每个队列都设置了不同的 TTL 属性，并绑定在同一个死信交换机中，消息过期后，根据 routingkey 的不同，又会被路由到不同的死信队列中，消费者只需要监听对应的死信队列进行处理即可。

2，利用 RabbitMQ 插件实现

安装一个插件即可：https:/https://www.rabbitmq.com/community-plugins.html ，下载 rabbitmq_delayed_message_exchange 插件，然后解压放置到 RabbitMQ 的插件目录。

把下载的插件 放到 容器内的 /plugins 目录内

51.如何保证消息不丢失？

丢失的原因可能有多种，每种都有对应的解决方案

第一种：生产者没能成功将消息发送到 MQ;

a、 丢失的原因：**因为网络传输的不稳定性，当生产者在向 MQ 发送消息的过程中，MQ 没有成功接收到消息，但是生产者却以为 MQ 成功接收到了消息，不会再次重复发送该消息，从而导致消息的丢失。

b、 解决办法： 有两个解决办法：事务机制和 confirm 机制，最常用的是 confirm 机制。

第二种：MQ 在接收到消息后弄丢了消息

a、 丢失的原因：RabbitMQ 接收到生产者发送过来的消息，是存在内存中的，如果没有被消费完，此时 RabbitMQ 宕机了，那么再次启动的时候，原来内存中的那些消息都丢失了。

b、 解决办法：开启 RabbitMQ 的持久化。当生产者把消息成功写入 RabbitMQ 之后，RabbitMQ 就把消息持久化到磁盘。结合上面的说到的 confirm 机制，只有当消息成功持久化磁盘之后，才会回调生产者的接口返回 ack 消息，否则都算失败，生产者会重新发送。存入磁盘的消息不会丢失，就算 RabbitMQ 挂掉了，重启之后，他会读取磁盘中的消息，不会导致消息的丢失。

第三种：消费者弄丢了消息

a、丢失的原因：如果 RabbitMQ 成功的把消息发送给了消费者，那么 RabbitMQ 的 ack 机制会自动的返回成功，表明发送消息成功，下次就不会发送这个消息。但如果就在此时，消费者还没处理完该消息，然后宕机了，那么这个消息就丢失了。

b、解决的办法：简单来说，就是必须关闭 RabbitMQ 的自动 `ack`，可以通过一个 api 来调用就行，然后每次在自己代码里确保处理完的时候，再在程序里 `ack` 一把。这样的话，如果你还没处理完，不就没有 `ack`了？那 RabbitMQ 就认为你还没处理完，这个时候 RabbitMQ 会把这个消费分配给别的 consumer 去处理，消息是不会丢的。

52.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。

53.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

54.四种类型的数据节点 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。

55.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
3.1Watcher 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器之间通过 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。

56.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。

57.Zookeeper 是如何保证事务的顺序一直性的？

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。

58.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)

59.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。

60.Zookeeper 角色

Zookeeper 集群是一个基于主从复制的高可用集群，每个服务器承担如下三种角色中的一种

1.Leader

1.一个 Zookeeper 集群同一时间只会有一个实际工作的 Leader，它会发起并维护与各 Follwer 及 Observer 间的心跳。

2.所有的写操作必须要通过 Leader 完成再由 Leader 将写操作广播给其它服务器。只要有超过半数节点（不包括 observeer 节点）写入成功，该写请求就会被提交（类 2PC 协议）。

2.Follower

1.一个 Zookeeper 集群可能同时存在多个 Follower，它会响应 Leader 的心跳，

2.Follower 可直接处理并返回客户端的读请求，同时会将写请求转发给 Leader 处理，

3.并且负责在 Leader 处理写请求时对请求进行投票。

3.Observer

角色与 Follower 类似，但是无投票权。Zookeeper 需保证高可用和强一致性，为了支持更多的客户端，需要增加更多 Server；Server 增多，投票阶段延迟增大，影响性能；引入 Observer，Observer 不参与投票； Observers 接受客户端的连接，并将写请求转发给 leader 节点； 加入更多 Observer 节点，提高伸缩性，同时不影响吞吐率。

61.Zookeeper 工作原理（原子广播）？

1.Zookeeper 的核心是原子广播，这个机制保证了各个 server 之间的同步。实现这个机制的协议叫做 Zab 协议。Zab 协议有两种模式，它们分别是恢复模式和广播模式。

2.当服务启动或者在领导者崩溃后，Zab 就进入了恢复模式，当领导者被选举出来，且大多数 server 的完成了和 leader 的状态同步以后，恢复模式就结束了。

3.状态同步保证了 leader 和 server 具有相同的系统状态

4.一旦 leader 已经和多数的 follower 进行了状态同步后，他就可以开始广播消息了，即进入广播状态。这时候当一个 server 加入 zookeeper 服务中，它会在恢复模式下启动，发现 leader，并和 leader 进行状态同步。待到同步结束，它也参与消息广播。Zookeeper 服务一直维持在 Broadcast 状态，直到 leader 崩溃了或者 leader 失去了大部分的 followers 支持。

5.广播模式需要保证 proposal 被按顺序处理，因此 zk 采用了递增的事务 id 号(zxid)来保证。所有的提议(proposal)都在被提出的时候加上了 zxid。

6.实现中 zxid 是一个 64 为的数字，它高 32 位是 epoch 用来标识 leader 关系是否改变，每次一个 leader 被选出来，它都会有一个新的 epoch。低 32 位是个递增计数。

7.当 leader 崩溃或者 leader 失去大多数的 follower，这时候 zk 进入恢复模式，恢复模式

需要重新选举出一个新的 leader，让所有的 server 都恢复到一个正确的状态。

62.什么是 Dubbo？

Dubbo 是阿里巴巴开源的基于 Java 的高性能 RPC 分布式服务框架，现已成为 Apache 基金会孵化项目。

63.为什么要用 Dubbo？

因为是阿里开源项目，国内很多互联网公司都在用，已经经过很多线上考验。内部使用了 Netty、Zookeeper，保证了高性能高可用性。
使用 Dubbo 可以将核心业务抽取出来，作为独立的服务，逐渐形成稳定的服务中心，可用于提高业务复用灵活扩展，使前端应用能更快速的响应多变的市场需求。

64.Dubbo 和 Spring Cloud 有什么区别？

两个没关联，如果硬要说区别，有以下几点。

1）通信方式不同
Dubbo 使用的是 RPC 通信，而 Spring Cloud 使用的是 HTTP RESTFul 方式。
2）组成部分不同
组件 | Dubbo | Spring Cloud ---|---|--- 服务注册中心 | Zookeeper | Spring Cloud Netflix Eureka 服务监控 | Dubbo-monitor | Spring Boot Admin 断路器 | 不完善 | Spring Cloud Netflix Hystrix 服务网关 | 无 | Spring Cloud Netflix Gateway 分布式配置 | 无 | Spring Cloud Config 服务跟踪 | 无 | Spring Cloud Sleuth 消息总线 | 无 | Spring Cloud Bus 数据流 | 无 | Spring Cloud Stream 批量任务 | 无 | Spring Cloud Task ... | ... | ...

65.dubbo 都支持什么协议，推荐用哪种？

dubbo://（推荐）
rmi://
hessian://
http://
webservice://
thrift://
memcached://
redis://
rest://

66.Dubbo 需要 Web 容器吗？

不需要，如果硬要用 Web 容器，只会增加复杂性，也浪费资源。

67.Dubbo 内置了哪几种容器？

Spring Container
Jetty Container
Log4j Container
Dubbo 的服务容器只是一个简单的 Main 方法,并加载一个简单的 Spring 容器,用于暴露服务。

68.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

69.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

70.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：一直性 Hash，相同参数的请求总是发到同一提供者

71.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。

72.Dubbo 默认使用什么注册中心，还有别的选择吗？

默认使用 Zookeeper 作为注册中心或者是 Nacos 注册中心，还有 Redis、Multicast、Simple 注册中心，但不推荐。

73.Dubbo 有哪几种配置方式？

1）Spring 配置方式 2）Java API 配置方式

74.Dubbo 核心的配置有哪些？

配置 | 配置说明 ---|--- dubbo:service | 服务配置 dubbo:reference | 引用配置 dubbo:protocol | 协议配置 dubbo:application | 应用配置 dubbo:module | 模块配置 dubbo:registry | 注册中心配置 dubbo:monitor | 监控中心配置 dubbo:provider | 提供方配置 dubbo:consumer | 消费方配置 dubbo:method | 方法配置 dubbo:argument | 参数配置

75.Dubbo 推荐使用什么序列化框架，你知道的还有哪些？

推荐使用 Hessian 序列化，还有 Duddo、FastJson、Java 自带序列化。

76.Dubbo 默认使用的是什么通信框架，还有别的选择吗？

Dubbo 默认使用 Netty 框架，也是推荐的选择，另外内容还集成有 Mina、Grizzly。

77.Dubbo 支持服务多协议吗？

Dubbo 允许配置多协议，在不同服务上支持不同协议或者同一服务上同时支持多种协议。

78.什么是 Spring Boot？

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。

79.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 = {enviornment}。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。

80.什么是 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。

81.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。

82.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。

83.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。

84.什么是 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。

85.什么是 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。

86.什么是 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。

87.什么是 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。

88.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、负载平衡 --负载平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。

89.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。 Nacos 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 Nacos 服务器上注册并通过调用 Nacos 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。

90.什么是 Netflix Feign？它的优点是什么？

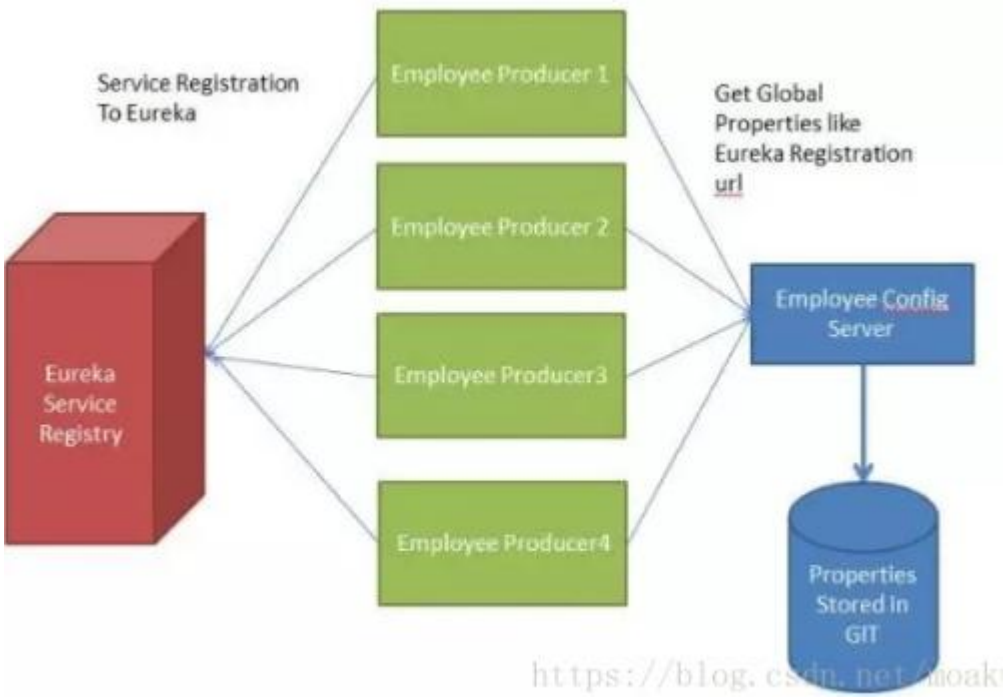
Feign 是受到 Retrofit，JAXRS-2.0 和 WebSocket 启发的 java 客户端联编程序。Feign 的第一个目标是将约束分母的复杂性统一到 http apis，而不考虑其稳定性。在 employee-consumer 的例子中，我们使用了 employee-producer 使用 REST 模板公开的 REST 服务。
但是我们必须编写大量代码才能执行以下步骤
使用功能区进行负载平衡。
获取服务实例，然后获取基本 URL。
利用 REST 模板来使用服务。 前面的代码如下

```
@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 NullPointer 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 默认也会负责负载平衡。

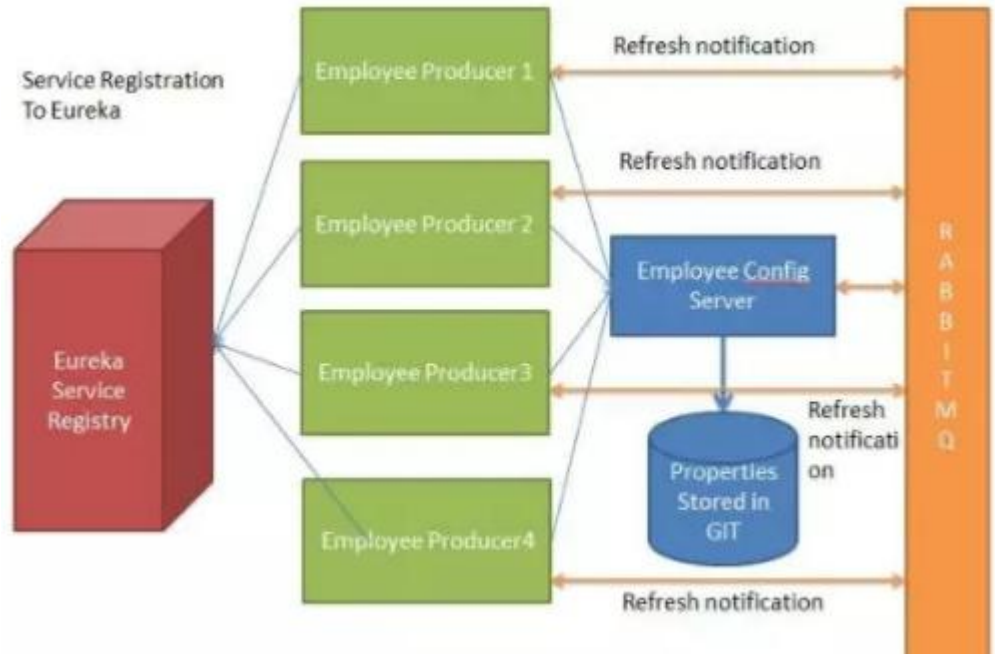
91.什么是 Spring Cloud Bus？我们需要它吗？

考虑以下情况：我们有多个应用程序使用 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。

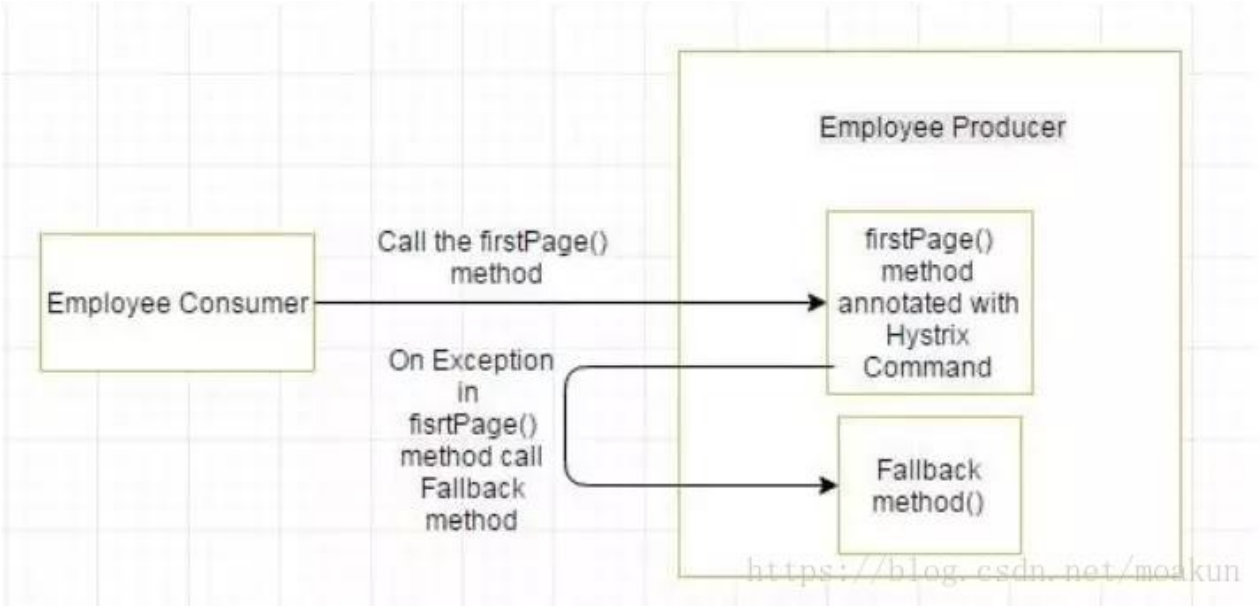


Spring Cloud Bus 提供了跨多个实例刷新配置的功能。因此，在上面的示例中，如果我们刷新 Employee Producer1，则会自动刷新所有其他必需的模块。如果我们有多个微服务启动并运行，

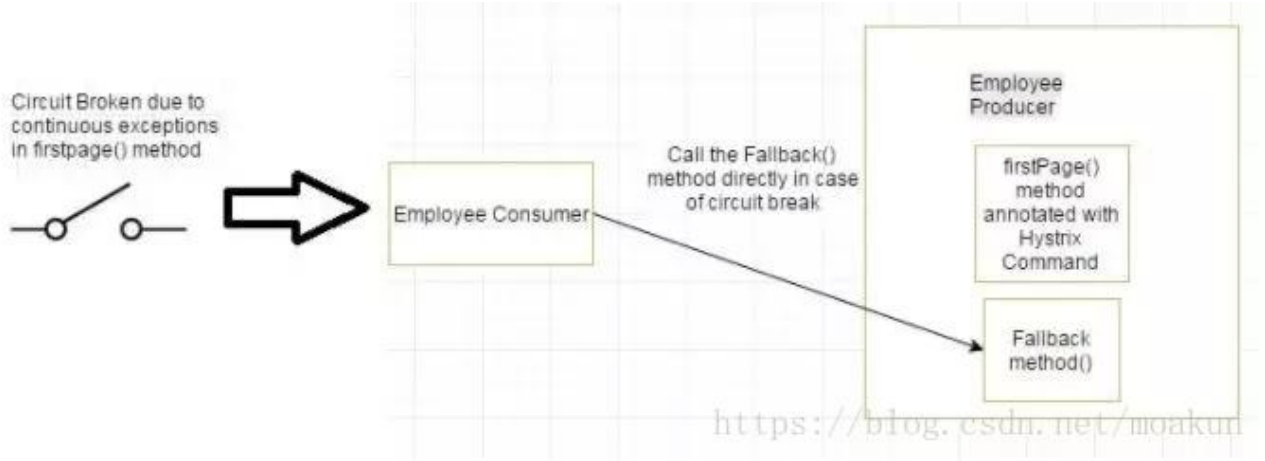
这特别有用。这是通过将所有微服务连接到单个消息代理来实现的。无论何时刷新实例，此事件都会订阅到侦听此代理的所有微服务，并且它们也会刷新。可以通过使用端点/总线/刷新来实现对任何单个实例的刷新。

92.什么是 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 firtsPage 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 。



93.什么是 ELK?

ELK 实际上是三个工具的集合，Elasticsearch+ Logstash + Kibana，这三个工具组合形成了一

套实用、易用的监控架构，很多公司利用它来搭建可视化的海量日志分析平台
Elasticsearch = 搜索日志，Logstash = 管理日志和事件的工具，你可以用它去收集日志、转换日志、解析日志,Kibana = 前端日志展示框架

94.你对微服务的认知?

1.微服务是一种架构风格
2.一组基于业务划分的服务
3.独立的进程
4.每一个模块都是需要相互通讯的。 Http，RPC，MQ。
5.独立技术栈
6.独立研发、部署
7.无集中式管理

95.SpringCloud 的常用组件?

Nacos 服务的注册与发现 (老版本使用 Eureka)
Gateway 为微服务架构系统提供高性能,且简单易用的 api 路由管理方式(老版本使用 Zuul)
Sentinel 流量控制、熔断降级、系统负载保护等多个维度来保障服务之间的稳定性。(老版本使用 Hystrix)
Seata 为微服务架构提供高性能和简单易用的分布式事务框架。
OpenFeign：是一种声明式、模板化的 HTTP 客户端
Sleuth+Zipkin 实现服务的链路跟踪
Nacos-config 实现配置中心

96.服务是如何被注册到 Nacos 的

在配置文件中配置 spring.cloud.nacos.discovery.server-addr

97.Springboot 支持松绑定么？什么是松绑定？

SpringBoot 中的松绑定适用于配置属性的类型安全绑定。使用松绑定，环境属性的键不需要与属性名完全匹配。这样就可以用驼峰式、短横线式、蛇形式或者下划线分割来命名。

98.什么是 CAP?

CAP 原则又称 CAP 定理，指的是在一个分布式系统中， Consistency（一致性）、 Availability（可用性）、Partition tolerance（分区容错性），三者不可得兼。
一致性（C）：
1. 在分布式系统中的所有数据备份，在同一时刻是否同样的值。（等同于所有节点访问同一份 最新的数据副本）
可用性（A）：
2. 在集群中一部分节点故障后，集群整体是否还能响应客户端的读写请求。（对数据更新具备 高可用性）

分区容忍性（P）：

3. 以实际效果而言，分区相当于对通信的时限要求。系统如果不能在时限内达成数据一致性，就意味着发生了分区的情况，必须就当前操作在 C 和 A 之间做出选择。

99.@ComponentScan 注解的作用？

1、@ComponentScan 这个注解在 Spring 中很重要，它对应 XML 配置中的元素，@ComponentScan 的功能其实就是自动扫描并加载符合条件的组件（比如@Component 和@Repository 等）或者 bean 定义，最终将这些 bean 定义加载到 IoC 容器中。

我们可以通过 basePackages 等属性来细粒度的定制@ComponentScan 自动扫描的范围，如果不指定，则默认 Spring 框架实现会从声明@ComponentScan 所在类的 package 进行扫描。

注：所以 SpringBoot 的启动类最好是放在 root package 下，因为默认不指定 basePackages。

2、@ComponentScan 告诉 Spring 哪个 packages 的用注解标识的类 会被 spring 自动扫描并且装入 bean 容器。

例如，如果你有个类用@Controller 注解标识了，那么，如果不加上@ComponentScan，自动扫描该 controller，那么该 Controller 就不会被 spring 扫描到，更不会装入 spring 容器中，因此你配置的这个 Controller 也没有意义。

100.@SpringBootApplication 注解的作用？

@SpringBootApplication 注解的组成

通过查看@SpringBootApplication 这个注解类的源码，我们可以看到，这个注解其实包括了三个注解——

@SpringBootConfiguration

@ComponentScan

@EnableAutoConfiguration

而其中 @SpringBootConfiguration 又是只有 @Configuration 注解而已，所以 @SpringBootApplication 其实相当于以下三个注解——

@Configuration

@EnableAutoConfiguration

@ComponentScan

springboot 项目为了方便开发者使用，才使用了 SpringBootApplication 注解来代替这三个注解，其实在启动类上，也可以只使用这三个注解。

101.@EnableAutoConfiguration(自动装配)注解的作用？

@EnableAutoConfiguration 作用：

从 classpath 中搜索所有 META-INF/spring.factories 配置文件然后，将其中 org.springframework.boot.autoconfigure.EnableAutoConfiguration key 对应的配置项加载到 spring 容器

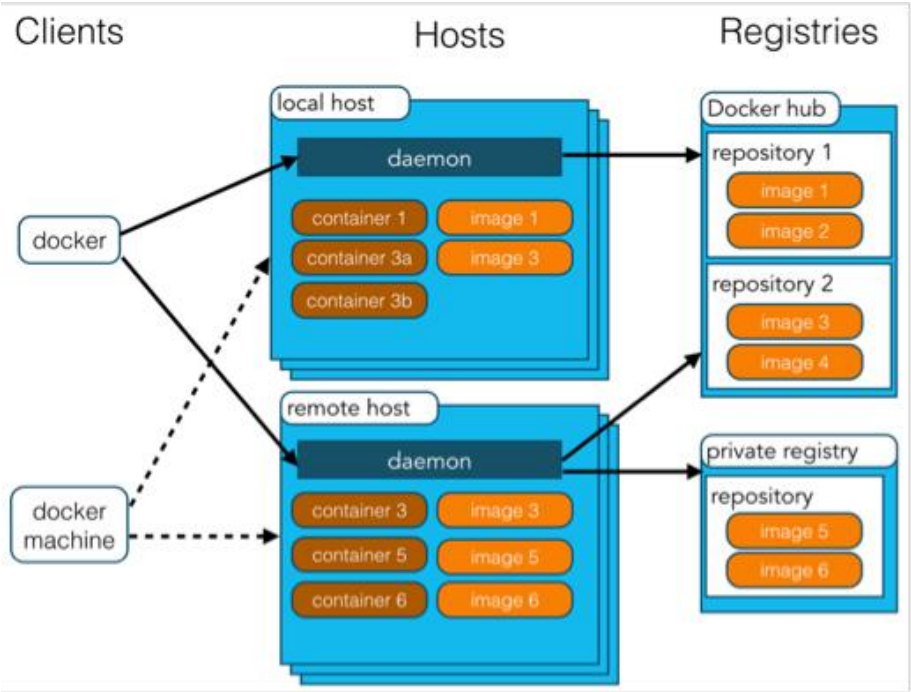
只有 spring.boot.enableautoconfiguration 为 true（默认为 true）的时候，才启用自动配置

@EnableAutoConfiguration 还可以进行排除，排除方式有 2 中，一是根据 class 来排除（exclude），二是根据 class name（excludeName）来排除

其内部实现的关键点有：
ImportSelector 该接口的方法的返回值都会被纳入到 spring 容器管理中
SpringFactoriesLoader 该类可以从 classpath 中搜索所有 META-INF/spring.factories 配置文件，并读取配置

102. Docker 是什么？

Docker 镜像(Images)：Docker 镜像是用于创建 Docker 容器的模板。
Docker 容器(Container) ：容器是独立运行的一个或一组应用。
Docker 客户端(Client)：Docker 客户端通过命令行或者其他工具使用 Docker API 与 Docker 的守护进程通信。
Docker 主机(Host) ：一个物理或者虚拟的机器用于执行 Docker 守护进程和容器。
Docker 仓库(Registry)：Docker 仓库用来保存镜像，可以理解为代码控制中的代码仓库。
Docker Hub：提供了庞大的镜像集合供使用。
DockerMachine：Docker Machine 是一个简化 Docker 安装的命令行工具，通过一个简单的命令行即可在相应的平台上安装 Docker，比如 VirtualBox、 Digital Ocean、Microsoft Azure。
Docker 的出现一定是因为目前的后端在开发和运维阶段确实需要一种虚拟化技术解决开发环境和生产环境环境一致的问题，通过 Docker 我们可以将程序运行的环境也纳入到版本控制中，排除因为环境造成不同运行结果的可能。但是上述需求虽然推动了虚拟化技术的产生，但是如果没有合 适的底层技术支撑，那么我们仍然得不到一个完美的产品。本文剩下的内容会介绍几种 Docker 使用的核心技术，如果我们了解它们的使用方法和原理，就能清楚 Docker 的实现原理。
Docker 使用客户端-服务器 (C/S) 架构模式，使用远程 API 来管理和创建 Docker 容器。
Docker 容器通过 Docker 镜像来创建。



103. linux 怎么查看系统负载情况？

uptime

load average 后的数字分别表示计算机在 1min、5min、10min 内的平均负载
dmesg | tail
打印内核环形缓存区的内容，通过 dmesg 可以快速判断是否有导致系统性能异常的问题
vmstat
vmstat 打印进程、内存、交换分区、IO 和 CPU 等统计信息。1 表示间隔 1 秒打印一次，3 表示总公打印三次
r：表示正在运行或者等待 CPU 调度的进程数，可以查看 CPU 是否处于饱和状态，如果数字大于 CPU 核数，则表示已经饱和
b：表示等待 IO 的进程数量
swpd：表示虚拟内存使用大小
si：每秒从交换分区写入内存的大小，由磁盘调入内存
so：每秒写入交换分区的内存大小，由内存调入磁盘
这里：如果 swpd 不为 0，但是 si 和 so 为 0，则不影响系统性能
free：表示空闲物理内存大小
buff：buffer cache 使用大小
cache：page cache 使用大小
bi：每秒读取的块数
bo：每秒写入的块数
如果 cache 的值大的时候，说明 cache 处的文件数多，如果频繁访问到的文件都能被 cache 处，那么磁盘的读 IO bi 会非常小

104.什么是 OAuth2？

OAuth（开放授权）是一个开放标准，允许用户授权第三方移动应用访问他们存储在另外的服务提供者上的信息，而不需要将用户名和密码提供给第三方移动应用或分享他们数据的所有内容

105.OAuth2 的四种授权模式

授权码模式（authorization code）
授权码模式（authorization code）是功能最完整、流程最严密的授权模式。
（1）用户访问客户端，后者将前者导向认证服务器，假设用户给予授权，认证服务器将用户导向客户端事先指定的"重定向 URI"（redirection URI），同时附上一个授权码。
（2）客户端收到授权码，附上早先的"重定向 URI"，向认证服务器申请令牌：GET /oauth/token?response_type=code&client_id=test&redirect_uri=重定向页面链接。请求成功返回 code 授权码，一般有效时间是 10 分钟。
（3）认证服务器核对了授权码和重定向 URI，确认无误后，向客户端发送访问令牌（access token）和更新令牌（refresh token）。POST /oauth/token?response_type=authorization_code&code=SplxlOBeZQQYbYS6WxSbIA&redirect_uri=重定向页面链接。
简化模式（implicit）
简化模式（implicit grant type）不通过第三方应用程序的服务器，直接在浏览器中向认证服务器申请令牌，跳过了"授权码"这个步骤，因此得名。所有步骤在浏览器中完成，令牌对访问者是可见的，且客户端不需要认证。
（A）客户端将用户导向认证服务器。

（B）用户决定是否给于客户端授权。

（C）假设用户给予授权，认证服务器将用户导向客户端指定的"重定向 URI"，并在 URI 的 Hash 部分包含了访问令牌。

（D）浏览器向资源服务器发出请求，其中不包括上一步收到的 Hash 值。

（E）资源服务器返回一个网页，其中包含的代码可以获取 Hash 值中的令牌。

（F）浏览器执行上一步获得的脚本，提取出令牌。

（G）浏览器将令牌发给客户端。

密码模式（resource owner password credentials）

密码模式（Resource Owner Password Credentials Grant）中，用户向客户端提供自己的用户名和密码。客户端使用这些信息，向"服务商提供商"索要授权。在这种模式中，用户必须把自己的密码给客户端，但是客户端不得储存密码。这通常用在用户对客户端高度信任的情况下。一般不支持 refresh token。

客户端模式（client credentials）

指客户端以自己的名义，而不是以用户的名义，向"服务提供商"进行认证。严格地说，客户端模式并不属于 OAuth 框架所要解决的问题。在这种模式中，用户直接向客户端注册，客户端以自己的名义要求"服务提供商"提供服务，其实不存在授权问题。

106. RedLock 的实现原理？

Redlock：全名叫做 Redis Distributed Lock;即使用 redis 实现的分布式锁

最低保证分布式锁的有效性及安全性的要求如下：

1.互斥；任何时刻只能有一个 client 获取锁

2.释放死锁；即使锁定资源的服务崩溃或者分区，仍然能释放锁

3.容错性；只要多数 redis 节点（一半以上）在使用，client 就可以获取和释放锁

多节点 redis 实现的分布式锁算法(RedLock):有效防止单点故障

假设有 5 个完全独立的 redis 主服务器

1.获取当前时间戳

2.client 尝试按照顺序使用相同的 key,value 获取所有 redis 服务的锁，在获取锁的过程中的获取时间比锁过期时间短很多，这是为了不要过长时间等待已经关闭的 redis 服务。并且试着获取下一个 redis 实例。

比如：TTL 为 5s,设置获取锁最多用 1s，所以如果一秒内无法获取锁，就放弃获取这个锁，从而尝试获取下个锁

3.client 通过获取所有能获取的锁后的时间减去第一步的时间，这个时间差要小于 TTL 时间并且至少有 3 个 redis 实例成功获取锁，才算真正的获取锁成功

4.如果成功获取锁，则锁的真正有效时间是 TTL 减去第三步的时间差 的时间；比如：TTL 是 5s,获取所有锁用了 2s,则真正锁有效时间为 3s(其实应该再减去时钟漂移);

5.如果客户端由于某些原因获取锁失败，便会开始解锁所有 redis 实例；因为可能已经获取了小于 3 个锁，必须释放，否则影响其他 client 获取锁。

RedLock 算法是否是异步算法：

可以看成是同步算法；因为 即使进程间（多个电脑间）没有同步时钟，但是每个进程时间流速大致相同；并且时钟漂移相对于 TTL 叫小，可以忽略，所以可以看成同步算法；（不够严谨，算法上要算上时钟漂移，因为如果两个电脑在地球两端，则时钟漂移非常大）

RedLock 失败重试：

当 client 不能获取锁时，应该在随机时间后重试获取锁；并且最好在同一时刻并发的把 set 命令

发送给所有 redis 实例；而且对于已经获取锁的 client 在完成任务后要及时释放锁，这是为了节省时间；

RedLock 释放锁：

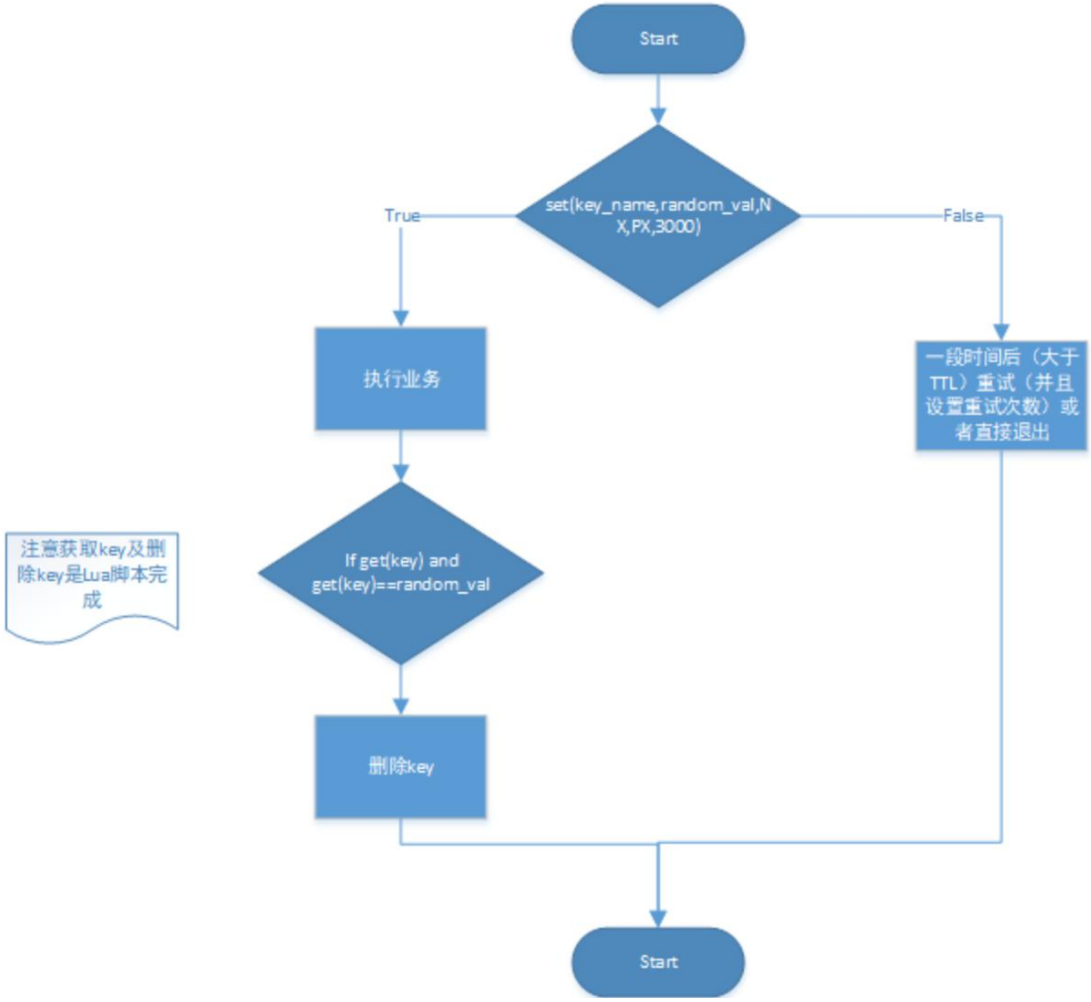
由于释放锁时会判断这个锁的 value 是不是自己设置的，如果是才删除；所以在释放锁时非常简单，只要向所有实例都发出释放锁的命令，不用考虑能否成功释放锁；

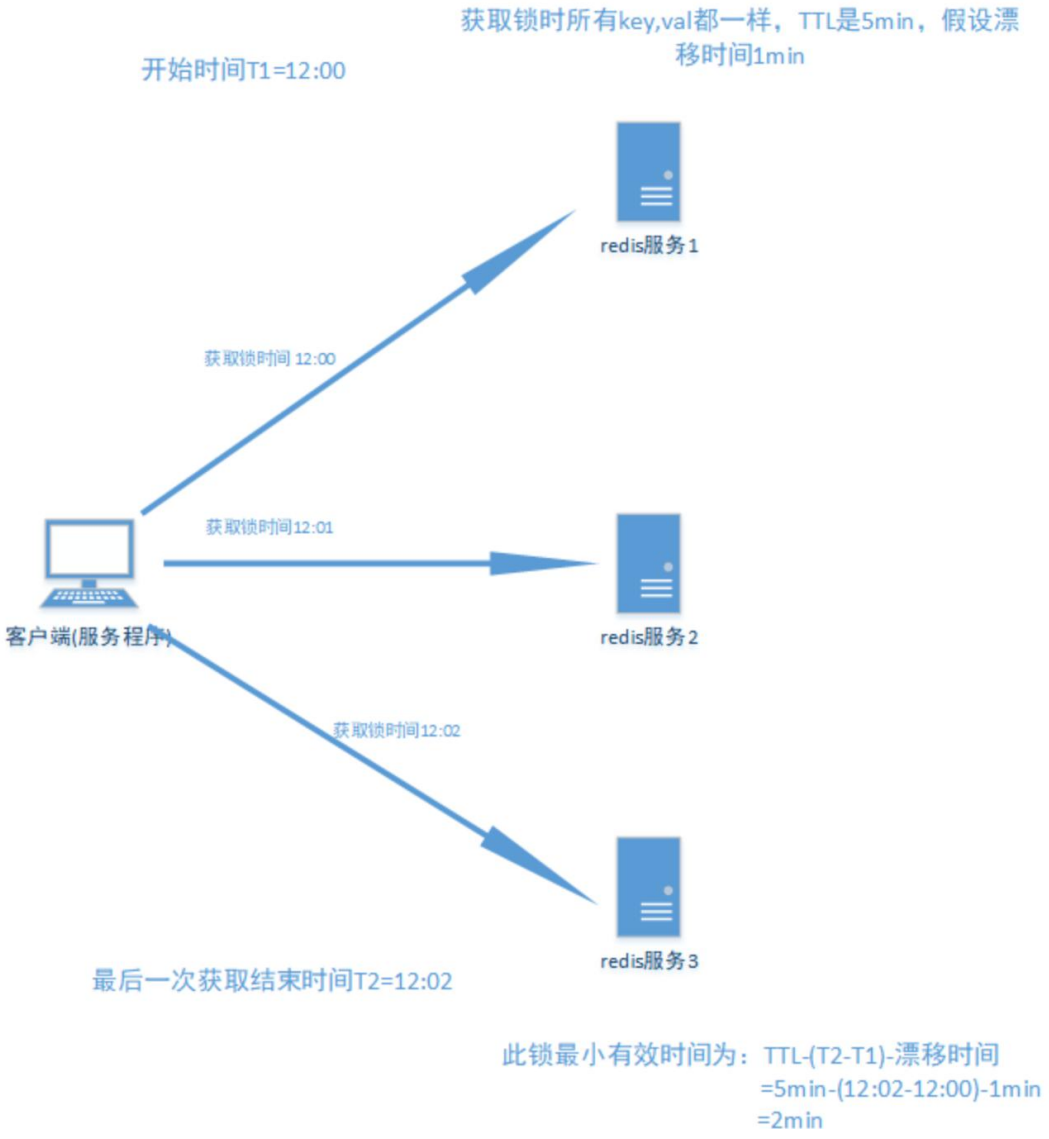
RedLock 注意点（Safety arguments）：

1.先假设 client 获取所有实例，所有实例包含相同的 key 和过期时间(TTL) ,但每个实例 set 命令时间不同导致不能同时过期，第一个 set 命令之前是 T1,最后一个 set 命令后为 T2,则此 client 有效获取锁的最小时间为 TTL-(T2-T1)-时钟漂移;

2.对于以 N/2+ 1(也就是一半以 上)的方式判断获取锁成功，是因为如果小于一半判断为成功的话，有可能出现多个 client 都成功获取锁的情况， 从而使锁失效

3.一个 client 锁定大多数事例耗费的时间大于或接近锁的过期时间，就认为锁无效，并且解锁这个 redis 实例(不执行业务) ;只要在 TTL 时间内成功获取一半以上的锁便是有效锁;否则无效





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
301Moved 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
302found 重定向
400Bad 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&param2=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&param2=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.TCP 为什么连接是三次握手,关闭却是四次握手？

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。故需要四步握手。

8.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.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

11.什么是无状态协议?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. 定位：发现需要优化的 SQL 语句

   可以采用一定的技术实现 SQL 语句定位：

   A. Mysql 的慢查询，开启慢查询实现执行慢的 SQL 语句

   B. 采用 Druid 实现，SQL 监控，获取执行慢的 SQL 语句

   C. Java 代码基于 Spring AOP 实现执行慢的 SQL 语句记录

   D. 采用第三方软件实现

2. 分析

   根据 SQL 分析，慢的原因，目前慢的原因主要有下面几种：

   1. 数据量大
   2. 并发量大
   3. Sql 语句业务逻辑

4. 索引失效
5. 多表关系路径问题
6. 互斥锁的问题
7. ……

3. 解决

根据分析的结果，进行方案选择，进行解决

比如数据量大，可以搭建 Mysql 集群，实现数据分片

比如并发量大，可以实现数据库的读写分离

比如 sql 语句有业务逻辑，需要把业务逻辑放到 Java 程序中

比如索引失效，那么尽量保证索引的高效

……

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要完全应用到数据库，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？

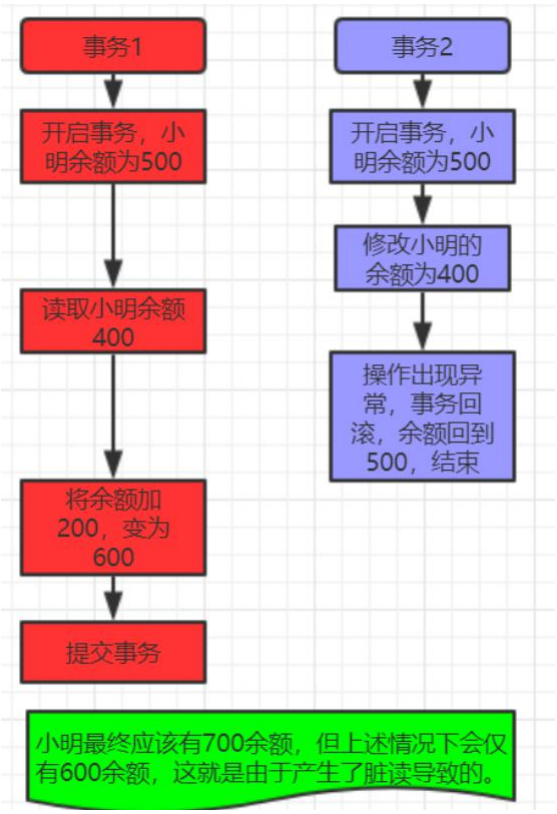
悲观锁（Pessimistic Lock），顾名思义，就是很悲观，每次去拿数据的时候都认为别人会修改,所以每次在拿数据的时候都会上锁,这样别人想拿这个数据就会阻塞挂起直到它拿到锁。传统的关系型数据库里边就用到了很多这种锁机制，比如行锁，表锁，读锁，写锁等，都是在做操作之前先上锁。

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。

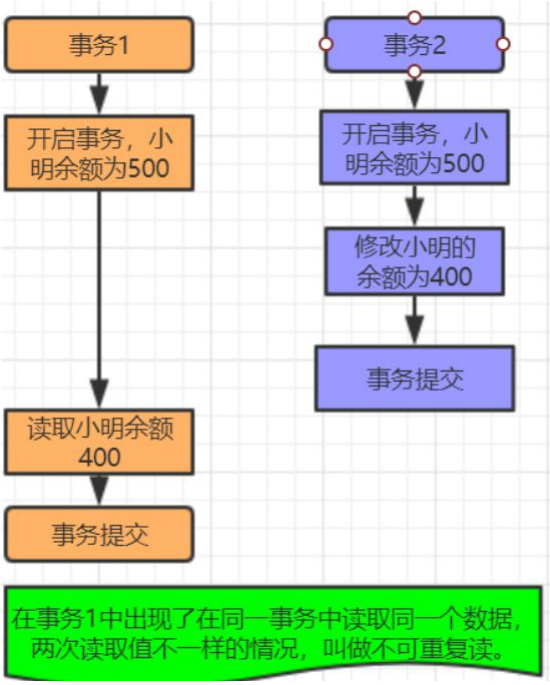
两种锁各有优缺点，乐观锁适用于写比较少的情况下，即冲突真的很少发生的时候，这样就可以省去锁的开销，加大系统的整个吞吐量。但如果经常产生冲突，上层应用会不断的进行 retry，这样反倒是降低了性能，所以这种情况下用悲观锁就比较合适。

10.事务的并发问题？

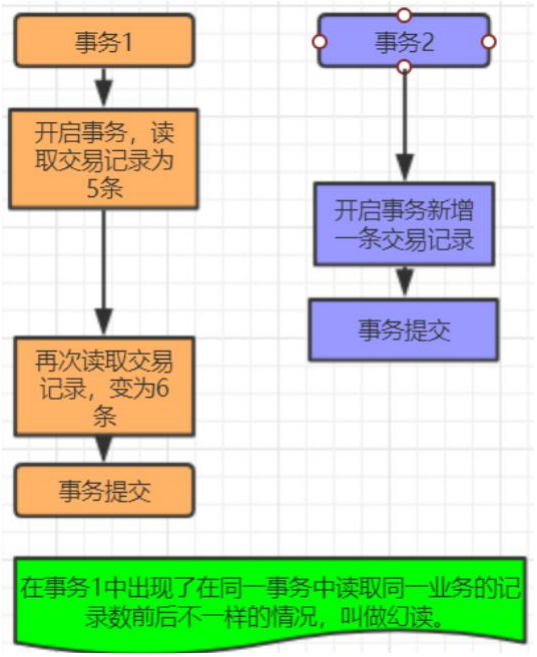
1. 脏读：是两个并发的事务 1 和事务 2 在执行时，事务 1 读取了事务 2 尚未提交的数据，而事务 2 由于出现异常回滚了,事务 1 拿着读到的事务 2 的数据进行后续操作时就出现数据错误了



2. 事务 1 中读取了某个数据，同时事务 2 也在修改这个数据，在事务 2 修改完这个数据并提交事务后，事务 1 又一次读取了这个数据，两次读取的同一数据就会不一致，这种现象叫做不可重复读。



3. 事务 1 中首先读取了某一业务（比如交易记录数）记录数，此时并发的事务 2 新增了一条交易记录，并提交事务，此时事务 1 再次读取交易记录数时，记录数前后就会出现不一致，这种现象称为幻读。



11.MySQL 的 MyISAM 与 InnoDB 存储引擎在事务,锁,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,其执行数度比 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.数据库的五大范式和 BCNF?

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
事务控制语言：TPL
游标操作语言：CCL

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、作为查询条件的字段都可以

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 就可以完全载入 局部性原理与磁盘预读

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 树的非终节点也包含需要查找的有效信息）

25.为什么 B+比 B 树更适合作为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，所以查询效率快，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。

27.事务的隔离级别?

第一种隔离级别：Read uncommitted(读未提交)

如果一个事务已经开始写数据，则另外一个事务不允许同时进行写操作，但允许其他事务读此行数据，该隔离级别可以通过“排他写锁”，但是不排斥读线程实现。这样就避免了更新丢失，却可能出现脏读，也就是说事务 B 读取到了事务 A 未提交的数据

解决了更新丢失，但还是可能会出现脏读

第二种隔离级别：Read committed(读提交)

如果是一个读事务(线程),则允许其他事务读写,如果是写事务将会禁止其他事务访问该行数据，该隔离级别避免了脏读，但是可能出现不可重复读。事务 A 事先读取了数据，事务 B 紧接着更新了数据，并提交了事务，而事务 A 再次读取该数据时，数据已经发生了改变。

解决了更新丢失和脏读问题

第三种隔离级别：Repeatable read(可重复读取)

可重复读取是指在一个事务内，多次读同一个数据，在这个事务还没结束时，其他事务不能访问该数据(包括了读写)，这样就可以在同一个事务内两次读到的数据是一样的，因此称为是可重复读隔离级别，读取数据的事务将会禁止写事务(但允许读事务)，写事务则禁止任何其他事务(包括了读写)，这样避免了不可重复读和脏读，但是有时可能会出现幻读。(读取数据的事务)可以通过“共享读镜”和“排他写锁”实现。

解决了更新丢失、脏读、不可重复读、但是还会出现幻读

第四种隔离级别：Serializable(可序化)

提供严格的事务隔离，它要求事务序列化执行，事务只能一个接着一个地执行，但不能并发执行，

如果仅仅通过“行级锁”是无法实现序列化的，必须通过其他机制保证新插入的数据不会被执行查询操作的事务访问到。序列化是最高的事务隔离级别，同时代价也是最高的，性能很低，一般很少使用，在该级别下，事务顺序执行，不仅可以避免脏读、不可重复读，还避免了幻读

28.索引的分类？

普通索引：仅加速查询
唯一索引：加速查询 + 列值唯一（可以有 null）
主键索引：加速查询 + 列值唯一（不可以有 null）+ 表中只有一个
组合索引：多列值组成一个索引，专门用于组合搜索，其效率大于索引合并
全文索引：对文本的内容进行分词，进行搜索
索引合并，使用多个单列索引组合搜索

29.索引的最佳左匹配？

最左前缀匹配原则：在 MySQL 建立联合索引时会遵守最左前缀匹配原则，即最左优先，在检索数据时从联合索引的最左边开始匹配。

30.什么是幂等性？

就是用户对于同一操作发起的一次请求或者多次请求的结果是一致的,不会因为多次点击而产生了副作用

31.ACID 是什么?可以详细说一下吗?

A=Atomicity
原子性,就是上面说的,要么全部成功,要么全部失败.不可能只执行一部分操作.
C=Consistency
系统(数据库)总是从一个一致性的状态转移到另一个一致性的状态,不会存在中间状态.
I=Isolation
隔离性: 通常来说:一个事务在完全提交之前,对其他事务是不可见的.注意前面的通常来说加了红色,意味着有例外情况.
D=Durability
持久性,一旦事务提交,那么就永远是这样子了,哪怕系统崩溃也不会影响到这个事务的结果.

32.统计过慢查询吗?对慢查询都怎么优化过?

a、错误日志：记录启动、运行或停止 mysqld 时出现的问题。
b、通用日志：记录建立的客户端连接和执行的语句。
c、更新日志：记录更改数据的语句。该日志在 MySQL 5.1 中已不再使用。
d、二进制日志：记录所有更改数据的语句。还用于主从复制。
e、慢查询日志：记录所有执行时间超过 long_query_time 秒的所有查询或不使用索引的查询。
f、Innodb 日志：innodb redo log
MySQL 会记录下查询超过指定时间的语句，我们将超过指定时间的 SQL 语句查询称为慢查询，都记在慢查询日志里.

我们开启后可以查看究竟是哪些语句在慢查询开启慢查询日志.

分析日志 - mysqldumpslow

分析日志，可用 mysql 提供的 mysqldumpslow，使用很简单，参数可 -help 查看

推荐用分析日志工具 - mysqlsla

优化：

1、索引没起作用的情况

使用 LIKE 关键字的查询语句

使用多列索引的查询语句

2、优化数据库结构

将字段很多的表分解成多个表

增加中间表

3、分解关联查询

将一个大的查询分解为多个小查询是很有必要的

4、优化 LIMIT 分页

在系统中需要分页的操作通常会使用 limit 加上偏移量的方法实现，同时加上合适的 order by 子句。如果有对应的索引，通常效率会不错，否则 MySQL 需要做大量的文件排序操作。

5、分析具体的 SQL 语句

十一、算法

1.冒泡排序？

```
**
  *  冒泡法排序<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，最后的元素应该会是最大的数。</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〉

```
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));
    }
}
```

2.实现两个有序数组的合并排序？

```
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();
    }
}
```

3. 实现一个数组的倒序？

```
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？

```java
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？

```java
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);
    }
}
```

7. DFS, BFS 算法？

DFS（深度优先遍历）
BFS（广度优先遍历）

```java
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);
                }
            }
        }
    }
}
```

8. 时间类型转换？

```java
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();
    }
}
```

9. 逆波兰计算器？

```java
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
                int num1 = Integer.parseInt(stack.pop());//弹出数字
                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());
    }
}
```

10. 请实现阶乘？

```java
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;
    }
}
```

11.兔子问题或者斐波那契数列？

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？

```
//这是一个菲波拉契数列问题
public class lianxi01 {
public static void main(String[] args) {
System.out.println("第 1 个月的兔子对数:       1");
System.out.println("第 2 个月的兔子对数:       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 +"个月的兔子对数: "+f2);
                 }
}
}
```

12.判断 101-200 之间素数,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);
}
}
```

13.请实现水仙花数？

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"); }
        }
}
}
```

14.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 <> 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( "请键入一个正整数:          ");
              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++;
                                }
        }
      }
```

15.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，只要除数不等于 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大数，取得的
```

余数作为下一轮循环的较小的数，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。 /*

```
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;
    }
}
```

16. 请实现完数？

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 qianfeng{
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 + "        ");
          }
        }
}
```

17. 请实现完全平方数?

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");
            }
          }
        }
}
}
```

18. 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");
}
}
```

19.请实现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，请编程序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]);
       }
     }
}
}
```

20.求 1+2!+3!+...+20!的和？

求 1+2!+3!+...+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);
}

}
```

21.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);
}
}
```

22.回文数的实现？

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.out.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
import java.util.*;
```

```
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");}
       }
     }
```

23.100 之内的素数？

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 + " ");}
        }
      }
}
```

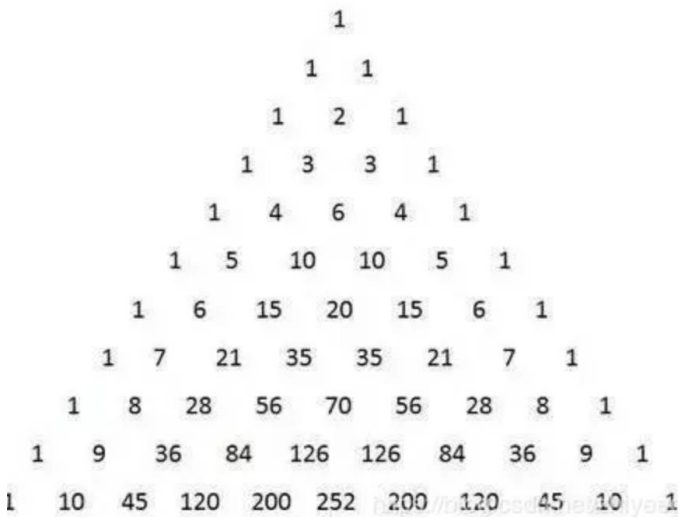
24.矩阵对角线元素之和？

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];
System.out.println("请输入 9 个整数：");
      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);
}
}
```

25.请实现杨辉三角形？

打印出杨辉三角形（要求打印出 10 行如下图）




```
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();
     }
}
}
```

26. 有 n 个人围成一圈顺序排号？

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 ++;
         if(index == n) {
         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。");
       }
     }
  }
}
```

27.海滩上有一堆桃子,5 只猴子来分？

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;}
}
}
}
```

28.一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。

```
//由于用除 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(fun(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
boolean 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) ;
}
}
```

29.实现快速排序？

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>重新排序数列，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（相同的数可以到任一边）。在这个分割之后，
 * 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);
        }
}
```

30. 请实现选择排序？

选择排序

```
/**
 * 选择排序<br/>
 * <li>在未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，直到所有元素均排序完毕。</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;
        }
}
```

31.请实现插入排序？

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>如果该元素（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者等于新元素的位置</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;
        }
}
```

32.请实现二分查找？

又叫折半查找，要求待查找的序列有序。每次取中间位置的值与待查关键字比较，如果中间位置

的值比待查关键字大，则在前半部分循环这个查找的过程，如果中间位置的值比待查关键字小，则在后半部分循环这个查找的过程。直到查找到了为止，否则序列中没有待查的关键字。

```
public static int biSearch(int[] array, int a) {
    int lo = 0;
    int hi = array.length - 1;
    int mid;
    while (lo <= hi) {
        mid = (lo + hi) / 2; //中间位置 if(array[mid]==a){
        return mid + 1;
    } else if (array[mid] < a) {
        //向右查找
        lo = mid + 1;
    } else {
        //向左查找
        hi = mid - 1;
    }
}
return - 1;
}
```

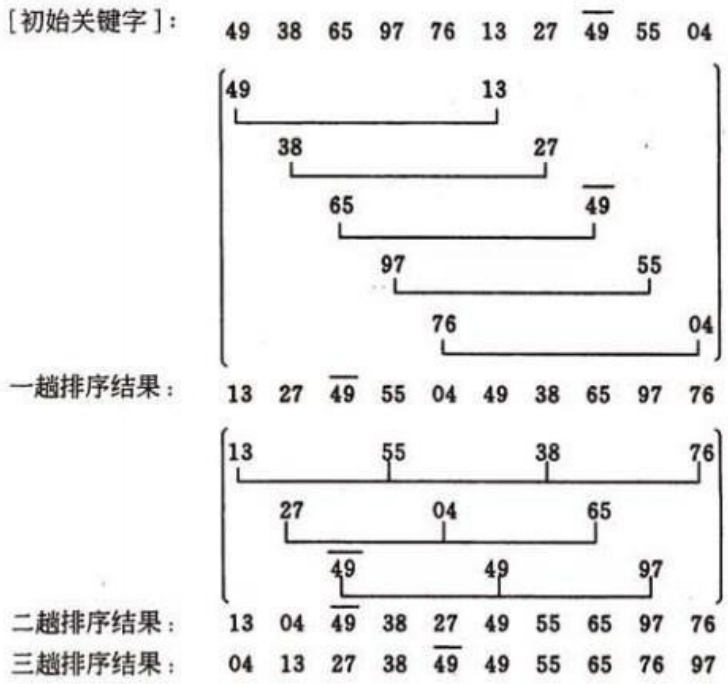
33.请实现希尔排序？

基本思想：先将整个待排序的记录序列分割成为若干子序列分别进行直接插入排序，待整个序列中的记录“基本有序”时，再对全体记录进行依次直接插入排序。

1. 操作方法：

选择一个增量序列 t1，t2，…，tk，其中 ti>tj，tk=1；

2. 按增量序列个数 k，对序列进行 k 趟排序；



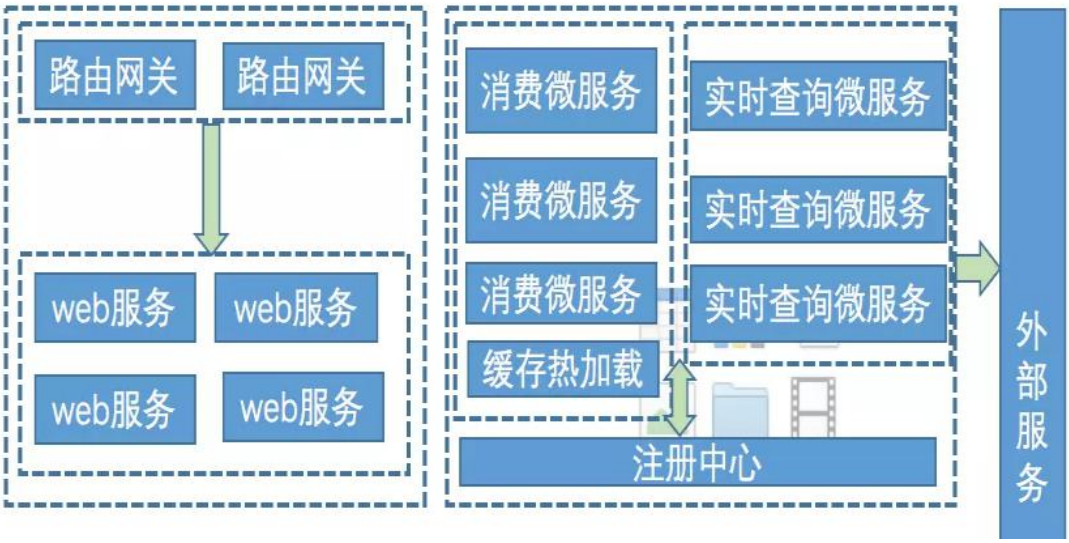
3. 每趟排序，根据对应的增量 ti，将待排序列分割成若干长度为 m 的子序列，分别对各子表进行直接插入排序。仅增量因子为 1 时，整个序列作为一个表来处理，表长度即为整个序列的长度。

```
private void shellSort(int[] a) {
    int dk = a.length / 2;
    while (dk >= 1) {
        ShellInsertSort(a, dk);
        dk = dk / 2;
    }
}
private void ShellInsertSort(int[] a, int dk) {
    //类似插入排序，只是插入排序增量是 1，这里增量是 dk,把 1 换成 dk 就可以了
    for (int i = dk; i < a.length; i++) {
        if (a[i] < a[i - dk]) {
            int j;
            int x = a[i]; //x 为待插入元素
            a[i] = a[i - dk];
            for (j = i - dk; j >= 0 && x < a[j]; j = j - dk) {
                //通过循环，逐个后移一位找到要插入的位置。a[j+dk]=a[j];
            }
            a[j + dk] = x; //插入
        }
    }
}
```

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?

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，众所周知一个良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.高并发下,我们系统是如何支撑请求?

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 的处置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。

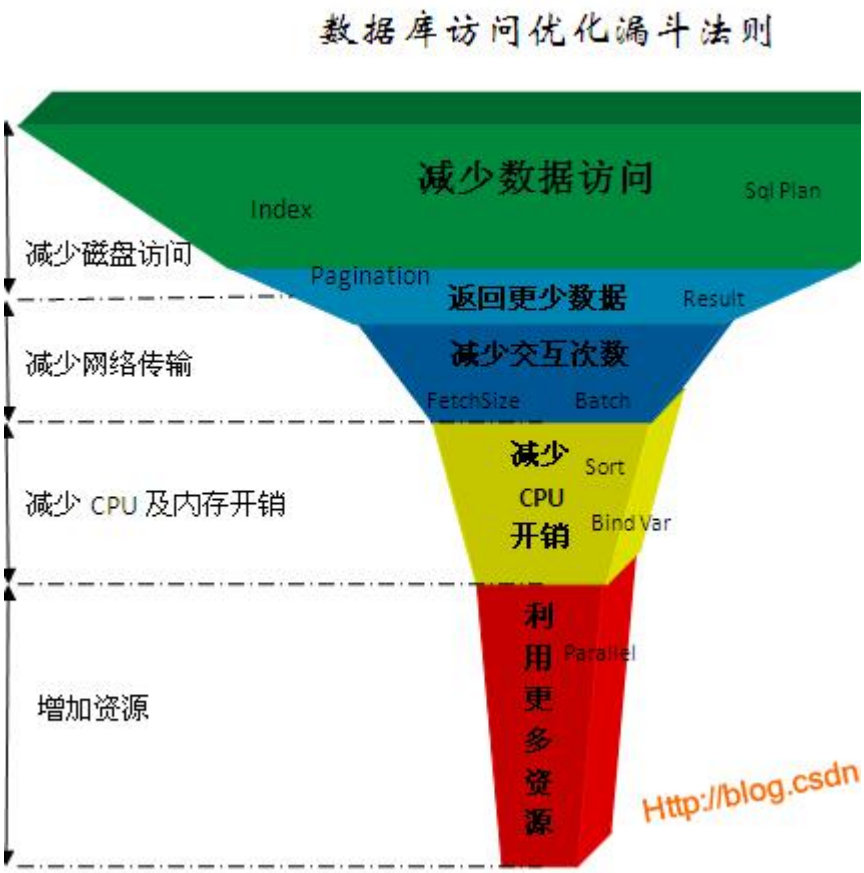
集群主要分为:高可用集群(High Availability Cluster),负载均衡集群(Load Balance Cluster,nginx 即可实现)，科学计算集群(High Performance Computing Cluster)。

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



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

9.秒杀系统的设计与实现？

1，验证时间；
2，输入验证码，这一步主要是时间换取空间，换取后台代码的执行时间；
3，用户点击按钮后，按钮变灰，主要是防止用户重复点击，也只能对‘小白’起作用，其中还有限制 IP，限制账号，分析用户行为等等；
4，验证库存，秒杀成功后库存-1，其中使用的是乐观锁，主要不认为会产生数据的冲突，因为并发量没那么高（如果出现商品超卖，就使用 redis 自旋锁配合数据库的乐观锁）；
5，用户秒杀成功后，但是五分钟内不付款，视为放弃放弃本次优惠，库存+1；

十三、开放性问题

1.工作上有哪些有成就感,又有哪些不能接受的地方？

每当项目交付的时候，成就感是最强的，因为每个产品都像是自己的创造的，看到项目圆满交付。有时也会陷入内卷中，导致项目烂尾什么的，就心理很难受

2.跟产品的关系怎么样？

这家公司的话，跟产品关系还可以，因为我们这个产品原来是做开发的，所以沟通需求很轻松没有太大的压力

3.你关注外包么,那在做外包的时候,最主要关注的点在哪里？

个人提升、团建、项目的参与度、福利待遇、稳定性

4.于你个人而言,你觉得软件开发这个行业是怎样一个行业？

指尖造梦，这个行业我自己很喜欢

5.在工作中有没有遇到过什么比较有趣的问题,最后是怎么解决的？

有时候用到新技术的时候，整个团队都需要去踩坑，有时也会遇到各种各样的问题，一群人一起攻坚战

6.有参与过需求分析会议么？

参加过，一般项目立项之后，会参与需求分析会，跟着老大他们和产品一起分析和梳理初始需求，完成需求池的整理

7.从技术角度来说,你觉得你跟前同事比怎么样？

相差不大，基本上我们干活整体都还可以，每个人的侧重点不一样

8.工作中觉得哪方面欠缺？

有的时候，感觉还是沟通，特别是跨部门的沟通，有时自己把握不准

9.项目出现许多 BUG,你一般会怎么解决？

通过 bug 日志分析，进行复盘分析，总结和提升。一般会：定位、分析、解决。怎么定位 bug 呢，一般我是通过记录日志或者 debug 的方式去定位 bug，然后分析错误的信息进行解决

10.你们公司 Git 的分支？

Git 企业级应用：多分支
master -主支 正式版代码 每一次都是一次版本的更新
test -分支 测试分支 主要是测试工程师用来进行测试的 都是完整代码
dev -分支 研发分支 研发工程师每次上传和下来代码的分支
bug -分支 项目缺陷分支管理 主要是日常项目有 bug 的代码 主要是测试工程师的反馈
error -分支 紧急错误分支 一般都是线上代码出了 bug 需要从 master 上同步代码

十四、IDEA 快捷键大全

Ctrl
快捷键 介绍
Ctrl + F 在当前文件进行文本查找 （必备）
Ctrl + R 在当前文件进行文本替换 （必备）
Ctrl + Z 撤销 （必备）
Ctrl + Y 删除光标所在行 或 删除选中的行 （必备）
Ctrl + X 剪切光标所在行 或 剪切选择内容
Ctrl + C 复制光标所在行 或 复制选择内容
Ctrl + D 复制光标所在行 或 复制选择内容，并把复制内容插入光标位置下面 （必备）
Ctrl + W 递进式选择代码块。可选中光标所在的单词或段落，连续按会在原有选中的基础上再扩展选中范围 （必备）
Ctrl + E 显示最近打开的文件记录列表
Ctrl + N 根据输入的 类名 查找类文件
Ctrl + G 在当前文件跳转到指定行处
Ctrl + J 插入自定义动态代码模板
Ctrl + P 方法参数提示显示

Ctrl + Q 光标所在的变量 / 类名 / 方法名等上面（也可以在提示补充的时候按），显示文档内容
Ctrl + U 前往当前光标所在的方法的父类的方法 / 接口定义
Ctrl + B 进入光标所在的方法/变量的接口或是定义出，等效于 Ctrl + 左键单击
Ctrl + K 版本控制提交项目，需要此项目有加入到版本控制才可用
Ctrl + T 版本控制更新项目，需要此项目有加入到版本控制才可用
Ctrl + H 显示当前类的层次结构
Ctrl + O 选择可重写的方法
Ctrl + I 选择可继承的方法
Ctrl + + 展开代码
Ctrl + - 折叠代码
Ctrl + / 注释光标所在行代码，会根据当前不同文件类型使用不同的注释符号 （必备）
Ctrl + [移动光标到当前所在代码的花括号开始位置
Ctrl +] 移动光标到当前所在代码的花括号结束位置
Ctrl + F1 在光标所在的错误代码出显示错误信息
Ctrl + F3 调转到所选中的词的下一个引用位置
Ctrl + F4 关闭当前编辑文件
Ctrl + F8 在 Debug 模式下，设置光标当前行为断点，如果当前已经是断点则去掉断点
Ctrl + F9 执行 Make Project 操作
Ctrl + F11 选中文件 / 文件夹，使用助记符设定 / 取消书签
Ctrl + F12 弹出当前文件结构层，可以在弹出的层上直接输入，进行筛选
Ctrl + Tab 编辑窗口切换，如果在切换的过程又加按上 delete，则是关闭对应选中的窗口
Ctrl + Enter 智能分隔行
Ctrl + End 跳到文件尾
Ctrl + Home 跳到文件头
Ctrl + Space 基础代码补全，默认在 Windows 系统上被输入法占用，需要进行修改，建议修改为 Ctrl + 逗号 （必备）
Ctrl + Delete 删除光标后面的单词或是中文句
Ctrl + BackSpace 删除光标前面的单词或是中文句
Ctrl + 1,2,3...9 定位到对应数值的书签位置
Ctrl + 左键单击 在打开的文件标题上，弹出该文件路径
Ctrl + 光标定位 按 Ctrl 不要松开，会显示光标所在的类信息摘要
Ctrl + 左方向键 光标跳转到当前单词 / 中文句的左侧开头位置
Ctrl + 右方向键 光标跳转到当前单词 / 中文句的右侧开头位置
Ctrl + 前方向键 等效于鼠标滚轮向前效果
Ctrl + 后方向键 等效于鼠标滚轮向后效果
Alt
快捷键 介绍
Alt + ` 显示版本控制常用操作菜单弹出层
Alt + Q 弹出一个提示，显示当前类的声明 / 上下文信息
Alt + F1 显示当前文件选择目标弹出层，弹出层中有很多目标可以进行选择
Alt + F2 对于前面页面，显示各类浏览器打开目标选择弹出层
Alt + F3 选中文本，逐个往下查找相同文本，并高亮显示
Alt + F7 查找光标所在的方法 / 变量 / 类被调用的地方

Alt + F8 在 Debug 的状态下，选中对象，弹出可输入计算表达式调试框，查看该输入内容的调试结果

Alt + Home 定位 / 显示到当前文件的 Navigation Bar

Alt + Enter IntelliJ IDEA 根据光标所在问题，提供快速修复选择，光标放在的位置不同提示的结果也不同 （必备）

Alt + Insert 代码自动生成，如生成对象的 set / get 方法，构造函数，toString() 等

Alt + 左方向键 按左方向切换当前已打开的文件视图

Alt + 右方向键 按右方向切换当前已打开的文件视图

Alt + 前方向键 当前光标跳转到当前文件的前一个方法名位置

Alt + 后方向键 当前光标跳转到当前文件的后一个方法名位置

Alt + 1,2,3...9 显示对应数值的选项卡，其中 1 是 Project 用得最多

Shift

快捷键 介绍

Shift + F1 如果有外部文档可以连接外部文档

Shift + F2 跳转到上一个高亮错误 或 警告位置

Shift + F3 在查找模式下，查找匹配上一个

Shift + F4 对当前打开的文件，使用新 Windows 窗口打开，旧窗口保留

Shift + F6 对文件 / 文件夹 重命名

Shift + F7 在 Debug 模式下，智能步入。断点所在行上有多个方法调用，会弹出进入哪个方法

Shift + F8 在 Debug 模式下，跳出，表现出来的效果跟 F9 一样

Shift + F9 等效于点击工具栏的 Debug 按钮

Shift + F10 等效于点击工具栏的 Run 按钮

Shift + F11 弹出书签显示层

Shift + Tab 取消缩进

Shift + ESC 隐藏当前 或 最后一个激活的工具窗口

Shift + End 选中光标到当前行尾位置

Shift + Home 选中光标到当前行头位置

Shift + Enter 开始新一行。光标所在行下空出一行，光标定位到新行位置

Shift + 左键单击 在打开的文件名上按此快捷键，可以关闭当前打开文件

Shift + 滚轮前后滚动 当前文件的横向滚动轴滚动

Ctrl + Alt

快捷键 介绍

Ctrl + Alt + L 格式化代码，可以对当前文件和整个包目录使用 （必备）

Ctrl + Alt + O 优化导入的类，可以对当前文件和整个包目录使用 （必备）

Ctrl + Alt + I 光标所在行 或 选中部分进行自动代码缩进，有点类似格式化

Ctrl + Alt + T 对选中的代码弹出环绕选项弹出层

Ctrl + Alt + J 弹出模板选择窗口，讲选定的代码加入动态模板中

Ctrl + Alt + H 调用层次

Ctrl + Alt + B 在某个调用的方法名上使用会跳到具体的实现处，可以跳过接口

Ctrl + Alt + V 快速引进变量

Ctrl + Alt + Y 同步、刷新

Ctrl + Alt + S 打开 IntelliJ IDEA 系统设置

Ctrl + Alt + F7 显示使用的地方。寻找被该类或是变量被调用的地方，用弹出框的方式找出来

Ctrl + Alt + F11 切换全屏模式

Ctrl + Alt + Enter 光标所在行上空出一行，光标定位到新行
Ctrl + Alt + Home 弹出跟当前文件有关联的文件弹出层
Ctrl + Alt + Space 类名自动完成
Ctrl + Alt + 左方向键 退回到上一个操作的地方 （必备）（注意与其他软件快捷键冲突）
Ctrl + Alt + 右方向键 前进到上一个操作的地方 （必备）（注意与其他软件快捷键冲突）
Ctrl + Alt + 前方向键 在查找模式下，跳到上个查找的文件
Ctrl + Alt + 后方向键 在查找模式下，跳到下个查找的文件
Ctrl + Shift
快捷键 介绍
Ctrl + Shift + F 根据输入内容查找整个项目 或 指定目录内文件 （必备）
Ctrl + Shift + R 根据输入内容替换对应内容，范围为整个项目 或 指定目录内文件 （必备）
Ctrl + Shift + J 自动将下一行合并到当前行末尾 （必备）
Ctrl + Shift + Z 取消撤销 （必备）
Ctrl + Shift + W 递进式取消选择代码块。可选中光标所在的单词或段落，连续按会在原有选中的基础上再扩展取消选中范围 （必备）
Ctrl + Shift + N 通过文件名定位 / 打开文件 / 目录，打开目录需要在输入的内容后面多加一个正斜杠 （必备）
Ctrl + Shift + U 对选中的代码进行大 / 小写轮流转换 （必备）
Ctrl + Shift + T 对当前类生成单元测试类，如果已经存在的单元测试类则可以进行选择
Ctrl + Shift + C 复制当前文件磁盘路径到剪贴板
Ctrl + Shift + V 弹出缓存的最近拷贝的内容管理器弹出层
Ctrl + Shift + E 显示最近修改的文件列表的弹出层
Ctrl + Shift + H 显示方法层次结构
Ctrl + Shift + B 跳转到类型声明处
Ctrl + Shift + I 快速查看光标所在的方法 或 类的定义
Ctrl + Shift + A 查找动作 / 设置
Ctrl + Shift + / 代码块注释 （必备）
Ctrl + Shift + [选中从光标所在位置到它的顶部中括号位置
Ctrl + Shift +] 选中从光标所在位置到它的底部中括号位置
Ctrl + Shift + + 展开所有代码
Ctrl + Shift + - 折叠所有代码
Ctrl + Shift + F7 高亮显示所有该选中文本，按 Esc 高亮消失
Ctrl + Shift + F8 在 Debug 模式下，指定断点进入条件
Ctrl + Shift + F9 编译选中的文件 / 包 / Module
Ctrl + Shift + F12 编辑器最大化
Ctrl + Shift + Space 智能代码提示
Ctrl + Shift + Enter 自动结束代码，行末自动添加分号 （必备）
Ctrl + Shift + Backspace 退回到上次修改的地方
Ctrl + Shift + 1,2,3...9 快速添加指定数值的书签
Ctrl + Shift + 左方向键 在代码文件上，光标跳转到当前单词 / 中文句的左侧开头位置，同时选中该单词 / 中文句
Ctrl + Shift + 右方向键 在代码文件上，光标跳转到当前单词 / 中文句的右侧开头位置，同时选中该单词 / 中文句
Ctrl + Shift + 左方向键 在光标焦点是在工具选项卡上，缩小选项卡区域

Ctrl + Shift + 右方向键 在光标焦点是在工具选项卡上，扩大选项卡区域
Ctrl + Shift + 前方向键 光标放在方法名上，将方法移动到上一个方法前面，调整方法排序
Ctrl + Shift + 后方向键 光标放在方法名上，将方法移动到下一个方法前面，调整方法排序
Alt + Shift
快捷键 介绍
Alt + Shift + N 选择 / 添加 task
Alt + Shift + F 显示添加到收藏夹弹出层
Alt + Shift + C 查看最近操作项目的变化情况列表
Alt + Shift + F 添加到收藏夹
Alt + Shift + I 查看项目当前文件
Alt + Shift + F7 在 Debug 模式下，下一步，进入当前方法体内，如果方法体还有方法，则会进入该内嵌的方法中，依此循环进入
Alt + Shift + F9 弹出 Debug 的可选择菜单
Alt + Shift + F10 弹出 Run 的可选择菜单
Alt + Shift + 左键双击 选择被双击的单词 / 中文句，按住不放，可以同时选择其他单词 / 中文句
Alt + Shift + 前方向键 移动光标所在行向上移动
Alt + Shift + 后方向键 移动光标所在行向下移动
Ctrl + Shift + Alt
快捷键 介绍
Ctrl + Shift + Alt + V 无格式黏贴
Ctrl + Shift + Alt + N 前往指定的变量 / 方法
Ctrl + Shift + Alt + S 打开当前项目设置
Ctrl + Shift + Alt + C 复制参考信息
其他
快捷键 介绍
F2 跳转到下一个高亮错误 或 警告位置 （必备）
F3 在查找模式下，定位到下一个匹配处
F4 编辑源
F7 在 Debug 模式下，进入下一步，如果当前行断点是一个方法，则进入当前方法体内，如果该方法体还有方法，则不会进入该内嵌的方法中
F8 在 Debug 模式下，进入下一步，如果当前行断点是一个方法，则不进入当前方法体内
F9 在 Debug 模式下，恢复程序运行，但是如果该断点下面代码还有断点则停在下一个断点上
F11 添加书签
F12 回到前一个工具窗口
Tab 缩进
ESC 从工具窗口进入代码文件窗口
连按两次 Shift 弹出 Search Everywhere 弹出层(全局搜索)