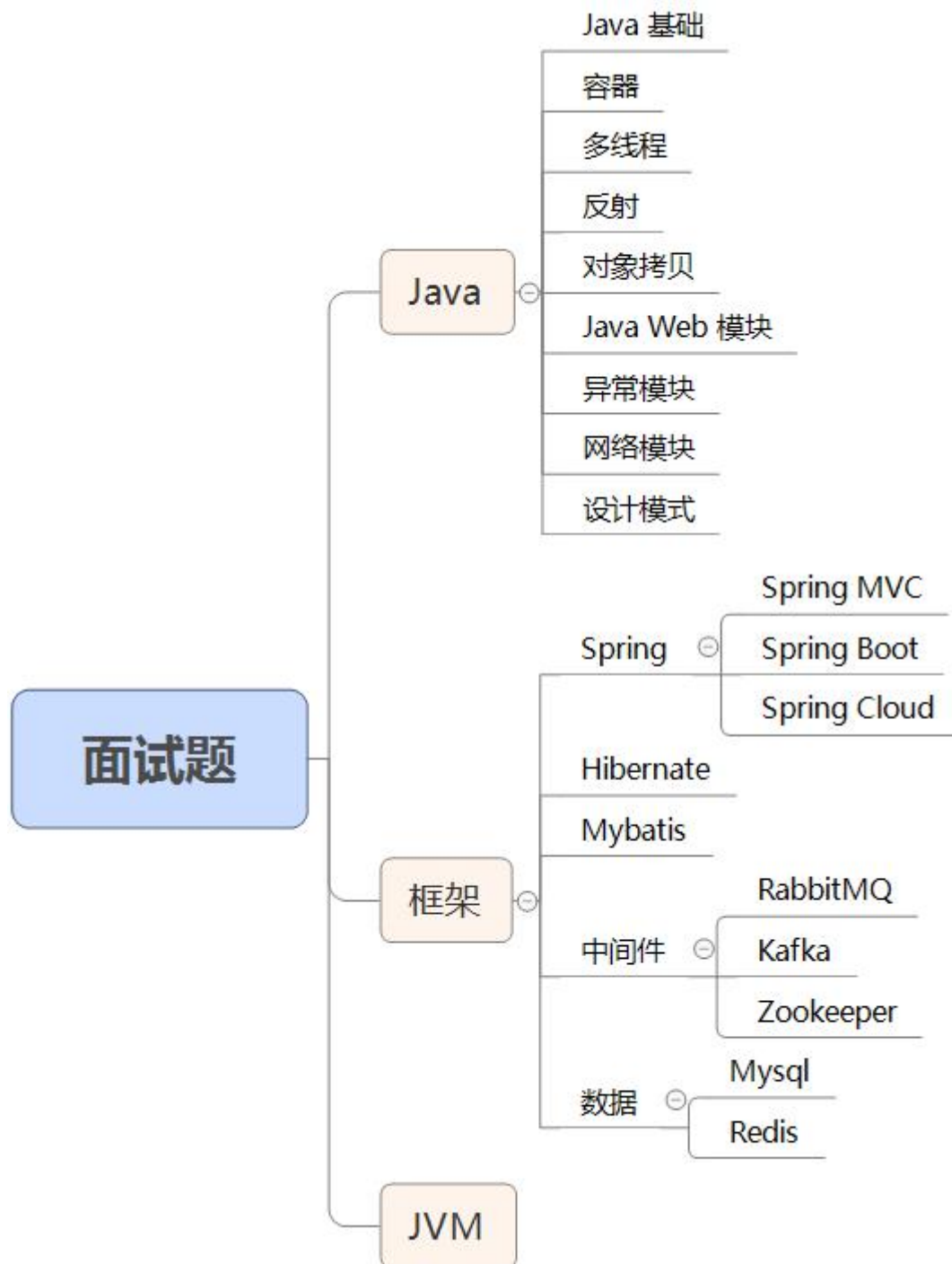


面 试 题 模 块 汇 总

面试题包括以下十九个模块：**Java 基础**、**容器**、**多线程**、**反射**、**对象拷贝**、**Java Web 模块**、**异常**、**网络**、**设计模式**、**Spring/Spring MVC**、**Spring Boot/Spring Cloud**、**Hibernate**、**Mybatis**、**RabbitMQ**、**Kafka**、**Zookeeper**、**MySql**、**Redis**、**JVM**。



一、Java 基础

1. JDK 和 JRE 有什么区别? -----	14
2. == 和 equals 的区别是什么? -----	14
3. 两个对象的 hashCode() 相同, 则 equals() 也一定为 true, 对吗? -----	17
4. final 在 java 中有什么作用? -----	18
5. java 中的 Math.round(-1.5) 等于多少? -----	18
6. String 属于基础的数据类型吗? -----	18
7. java 中操作字符串都有哪些类? 它们之间有什么区别? -----	18
8. String str="i" 与 String str=new String("i") 一样吗? -----	19
9. 如何将字符串反转? -----	19
10. String 类的常用方法都有那些? -----	19
11. 抽象类必须要有抽象方法吗? -----	20
12. 普通类和抽象类有哪些区别? -----	20
13. 抽象类能使用 final 修饰吗? -----	20
14. 接口和抽象类有什么区别? -----	21
15. java 中 IO 流分为几种? -----	21
16. BIO、NIO、AIO 有什么区别? -----	21
17. Files 的常用方法都有哪些? -----	21

二、容器

18. java 容器都有哪些? -----	22
19. Collection 和 Collections 有什么区别? -----	22
20. List、Set、Map 之间的区别是什么? -----	23

21. HashMap 和 Hashtable 有什么区别? -----	23
22. 如何决定使用 HashMap 还是 TreeMap? -----	23
23. 说一下 HashMap 的实现原理? -----	24
24. 说一下 HashSet 的实现原理? -----	24
25. ArrayList 和 LinkedList 的区别是什么? -----	24
26. 如何实现数组和 List 之间的转换? -----	24
27. ArrayList 和 Vector 的区别是什么? -----	25
28. Array 和 ArrayList 有何区别? -----	25
29. 在 Queue 中 poll()和 remove()有什么区别? -----	25
30. 哪些集合类是线程安全的? -----	25
31. 迭代器 Iterator 是什么? -----	25
32. Iterator 怎么使用? 有什么特点? -----	26
33. Iterator 和 ListIterator 有什么区别? -----	26
34. 怎么确保一个集合不能被修改? -----	26
 三、多线程	
35. 并行和并发有什么区别? -----	26
36. 线程和进程的区别? -----	27
37. 守护线程是什么? -----	27
38. 创建线程有哪几种方式? -----	27
39. 说一下 runnable 和 callable 有什么区别? -----	28
40. 线程有哪些状态? -----	28
41. sleep() 和 wait() 有什么区别? -----	28

42. <code>notify()</code> 和 <code>notifyAll()</code> 有什么区别? -----	29
43. 线程的 <code>run()</code> 和 <code>start()</code> 有什么区别? -----	29
44. 创建线程池有哪几种方式? -----	30
45. 线程池都有哪些状态? -----	30
46. 线程池中 <code>submit()</code> 和 <code>execute()</code> 方法有什么区别? -----	31
47. 在 java 程序中怎么保证多线程的运行安全? -----	31
48. 多线程锁的升级原理是什么? -----	31
49. 什么是死锁? -----	32
50. 怎么防止死锁? -----	32
51. <code>ThreadLocal</code> 是什么? 有哪些使用场景? -----	32
52. 说一下 <code>synchronized</code> 底层实现原理? -----	33
53. <code>synchronized</code> 和 <code>volatile</code> 的区别是什么? -----	33
54. <code>synchronized</code> 和 <code>Lock</code> 有什么区别? -----	33
55. <code>synchronized</code> 和 <code>ReentrantLock</code> 区别是什么? -----	34
56. 说一下 <code>atomic</code> 的原理? -----	34

四、反射

57. 什么是反射? -----	35
58. 什么是 java 序列化? 什么情况下需要序列化? -----	35
59. 动态代理是什么? 有哪些应用? -----	35
60. 怎么实现动态代理? -----	36

五、对象拷贝

61. 为什么要使用克隆？	36
62. 如何实现对象克隆？	36
63. 深拷贝和浅拷贝区别是什么？	42

六、Java Web

64. jsp 和 servlet 有什么区别？	42
65. jsp 有哪些内置对象？作用分别是什么？	42
66. 说一下 jsp 的 4 种作用域？	43
67. session 和 cookie 有什么区别？	43
68. 说一下 session 的工作原理？	44
69. 如果客户端禁止 cookie 能实现 session 还能用吗？	44
70. spring mvc 和 struts 的区别是什么？	45
71. 如何避免 sql 注入？	46
72. 什么是 XSS 攻击，如何避免？	46
73. 什么是 CSRF 攻击，如何避免？	46

七、异常

74. throw 和 throws 的区别？	47
75. final、finally、finalize 有什么区别？	48
76. try-catch-finally 中哪个部分可以省略？	48
77. try-catch-finally 中，如果 catch 中 return 了，finally 还会执行吗？	49
78. 常见的异常类有哪些？	52

八、网络

79.http 响应码 301 和 302 代表的是什么？有什么区别？	52
80.forward 和 redirect 的区别？	53
81.简述 tcp 和 udp 的区别？	53
82.tcp 为什么要三次握手，两次不行吗？为什么？	53
83.说一下 tcp 粘包是怎么产生的？	54
84.OSI 的七层模型都有哪些？	55
85.get 和 post 请求有哪些区别？	55
86.如何实现跨域？	56
87.说一下 JSONP 实现原理？	64

九、设计模式

88.说一下你熟悉的设计模式？	64
89.简单工厂和抽象工厂有什么区别？	77

十、Spring/Spring MVC

90.为什么要使用 spring？	79
91.解释一下什么是 aop？	81
92.解释一下什么是 ioc？	81
93.spring 有哪些主要模块？	83
94.spring 常用的注入方式有哪些？	84
95.spring 中的 bean 是线程安全的吗？	84
96.spring 支持几种 bean 的作用域？	84
97.spring 自动装配 bean 有哪些方式？	85

98. spring 事务实现方式有哪些? -----	85
99. 说一下 spring 的事务隔离? -----	86
100. 说一下 spring mvc 运行流程? -----	86
101. spring mvc 有哪些组件? -----	87
102. @RequestMapping 的作用是什么? -----	88
103. @Autowired 的作用是什么? -----	88
 十一、Spring Boot/Spring Cloud	
104. 什么是 spring boot? -----	90
105. 为什么要用 spring boot? -----	90
106. spring boot 核心配置文件是什么? -----	90
107. spring boot 配置文件有哪几种类型? 它们有什么区别? -----	90
108. spring boot 有哪些方式可以实现热部署? -----	91
109. jpa 和 hibernate 有什么区别? -----	93
110. 什么是 spring cloud? -----	94
111. spring cloud 断路器的作用是什么? -----	94
112. spring cloud 的核心组件有哪些? -----	95
 十二、Hibernate	
113. 为什么要使用 hibernate? -----	95
114. 什么是 ORM 框架? -----	96
115. hibernate 中如何在控制台查看打印的 sql 语句? -----	96
116. hibernate 有几种查询方式? -----	97
117. hibernate 实体类可以被定义为 final 吗? -----	98

118. 在 hibernate 中使用 Integer 和 int 做映射有什么区别？	98
119. hibernate 是如何工作的？	99
120. get() 和 load() 的区别？	99
121. 说一下 hibernate 的缓存机制？	99
122. hibernate 对象有哪些状态？	100
123. 在 hibernate 中 getCurrentSession 和 openSession 的区别是什么？	100
124. hibernate 实体类必须要有无参构造函数吗？为什么？	101

十三、Mybatis

125. mybatis 中 #{} 和 \${} 的区别是什么？	101
126. mybatis 有几种分页方式？	101
127. RowBounds 是一次性查询全部结果吗？为什么？	101
128. mybatis 逻辑分页和物理分页的区别是什么？	101
129. mybatis 是否支持延迟加载？延迟加载的原理是什么？	102
130. 说一下 mybatis 的一级缓存和二级缓存？	102
131. mybatis 和 hibernate 的区别有哪些？	102
132. mybatis 有哪些执行器 (Executor)？	103
133. mybatis 分页插件的实现原理是什么？	103
134. mybatis 如何编写一个自定义插件？	103

十四、RabbitMQ

135. rabbitmq 的使用场景有哪些？	105
136. rabbitmq 有哪些重要的角色？	105
137. rabbitmq 有哪些重要的组件？	106

138. rabbitmq 中 vhost 的作用是什么？	106
139. rabbitmq 的消息是怎么发送的？	106
140. rabbitmq 怎么保证消息的稳定性？	106
141. rabbitmq 怎么避免消息丢失？	106
142. 要保证消息持久化成功的条件有哪些？	106
143. rabbitmq 持久化有什么缺点？	107
144. rabbitmq 有几种广播类型？	107
145. rabbitmq 怎么实现延迟消息队列？	107
146. rabbitmq 集群有什么用？	107
147. rabbitmq 节点的类型有哪些？	107
148. rabbitmq 集群搭建需要注意哪些问题？	107
149. rabbitmq 每个节点是其他节点的完整拷贝吗？为什么？	107
150. rabbitmq 集群中唯一一个磁盘节点崩溃了会发生什么情况？	108
151. rabbitmq 对集群节点停止顺序有要求吗？	108

十五、Kafka

152. kafka 可以脱离 zookeeper 单独使用吗？为什么？	108
153. kafka 有几种数据保留的策略？	108
154. kafka 同时设置了 7 天和 10G 清除数据，到第五天的时候消息达到了 10G，这个时候 kafka 将如何处理？	108
155. 什么情况会导致 kafka 运行变慢？	108
156. 使用 kafka 集群需要注意什么？	109

十六、Zookeeper

157. zookeeper 是什么？	109
158. zookeeper 都有哪些功能？	109
159. zookeeper 有几种部署模式？	109
160. zookeeper 怎么保证主从节点的状态同步？	110
161. 集群中为什么要有主节点？	110
162. 集群中有 3 台服务器，其中一个节点宕机，这个时候 zookeeper 还可以使用吗？	110
163. 说一下 zookeeper 的通知机制？	110

十七、MySQL

164. 数据库的三范式是什么？	110
165. 一张自增表里面总共有 7 条数据，删除了最后 2 条数据，重启 mysql 数据库，又插入了一条数据，此时 id 是几？	110
166. 如何获取当前数据库版本？	111
167. 说一下 ACID 是什么？	111
168. char 和 varchar 的区别是什么？	111
169. float 和 double 的区别是什么？	111
170. mysql 的内连接、左连接、右连接有什么区别？	112
171. mysql 索引是怎么实现的？	112
172. 怎么验证 mysql 的索引是否满足需求？	112
173. 说一下数据库的事务隔离？	112
174. 说一下 mysql 常用的引擎？	113
175. 说一下 mysql 的行锁和表锁？	113

176. 说一下乐观锁和悲观锁？ -----	114
177. mysql 问题排查都有哪些手段？ -----	114
178. 如何做 mysql 的性能优化？ -----	114

十八、Redis

179. redis 是什么？ 都有哪些使用场景？ -----	114
180. redis 有哪些功能？ -----	115
181. redis 和 memecache 有什么区别？ -----	115
182. redis 为什么是单线程的？ -----	115
183. 什么是缓存穿透？ 怎么解决？ -----	115
184. redis 支持的数据类型有哪些？ -----	115
185. redis 支持的 java 客户端都有哪些？ -----	116
186. jedis 和 redisson 有哪些区别？ -----	116
187. 怎么保证缓存和数据库数据的一致性？ -----	116
188. redis 持久化有几种方式？ -----	116
189. redis 怎么实现分布式锁？ -----	116
190. redis 分布式锁有什么缺陷？ -----	116
191. redis 如何做内存优化？ -----	117
192. redis 淘汰策略有哪些？ -----	117
193. redis 常见的性能问题有哪些？ 该如何解决？ -----	117

十九、JVM

194. 说一下 jvm 的主要组成部分？ 及其作用？ -----	117
195. 说一下 jvm 运行时数据区？ -----	118

196. 说一下堆栈的区别？ -----	118
197. 队列和栈是什么？ 有什么区别？ -----	118
198. 什么是双亲委派模型？ -----	118
199. 说一下类加载的执行过程？ -----	119
200. 怎么判断对象是否可以被回收？ -----	120
201. java 中都有哪些引用类型？ -----	120
202. 说一下 jvm 有哪些垃圾回收算法？ -----	120
203. 说一下 jvm 有哪些垃圾回收器？ -----	120
204. 详细介绍一下 CMS 垃圾回收器？ -----	121
205. 新生代垃圾回收器和老生代垃圾回收器都有哪些？ 有什么区别？ -----	121
206. 简述分代垃圾回收器是怎么工作的？ -----	121
207. 说一下 jvm 调优的工具？ -----	122
208. 常用的 jvm 调优的参数都有哪些？ -----	122

基础模块（一）

1. JDK 和 JRE 有什么区别？

JDK: Java Development Kit 的简称, java 开发工具包, 提供了 java 的开发环境和运行环境。

JRE: Java Runtime Environment 的简称, java 运行环境, 为 java 的运行提供了所需环境。

具体来说 JDK 其实包含了 JRE, 同时还包含了编译 java 源码的编译器 javac, 还包含了很多 java 程序调试和分析的工具。简单来说: 如果你需要运行 java 程序, 只需安装 JRE 就可以了, 如果你需要编写 java 程序, 需要安装 JDK。

2. == 和 equals 的区别是什么? == 解读

对于基本类型和引用类型 == 的作用效果是不同的, 如下所示:

- 基本类型: 比较的是值是否相同;
- 引用类型: 比较的是引用是否相同;

代码示例:

```
String x = "string";

String y = "string";

String z = new String("string");

System.out.println(x==y); // true

System.out.println(x==z); // false

System.out.println(x.equals(y)); // true

System.out.println(x.equals(z)); // true
```

代码解读: 因为 x 和 y 指向的是同一个引用, 所以 == 也是 true, 而 new String() 方法则重写开辟了内存空间, 所以 == 结果为 false, 而 equals 比较的一直是值, 所以结果都为 true。

equals 解读

equals 本质上就是 `==`，只不过 `String` 和 `Integer` 等重写了 `equals` 方法，把它变成了值比较。看下面的代码就明白了。

首先来看默认情况下 `equals` 比较一个有相同值的对象，代码如下：

```
class Cat {

    public Cat(String name) {

        this.name = name;

    }

    private String name;

    public String getName() {

        return name;

    }

    public void setName(String name) {

        this.name = name;

    }}

Cat c1 = new Cat("王磊");

Cat c2 = new Cat("王磊");

System.out.println(c1.equals(c2)); // false
```

输出结果出乎我们的意料，竟然是 false？这是怎么回事，看了 equals 源码就知道了，源码如下：

```
public boolean equals(Object obj) {  
  
    return (this == obj);}
```

原来 equals 本质上就是 ==。

那问题来了，两个相同值的 String 对象，为什么返回的是 true？代码如下：

```
String s1 = new String("老王");  
  
String s2 = new String("老王");  
  
System.out.println(s1.equals(s2)); // true
```

同样的，当我们进入 String 的 equals 方法，找到了答案，代码如下：

```
public boolean equals(Object anObject) {  
  
    if (this == anObject) {  
  
        return true;  
  
    }  
  
    if (anObject instanceof String) {  
  
        String anotherString = (String)anObject;  
  
        int n = value.length;  
  
        if (n == anotherString.value.length) {  
  
            char v1[] = value;
```



```

        char v2[] = anotherString.value;

        int i = 0;

        while (n-- != 0) {

            if (v1[i] != v2[i])

                return false;

            i++;

        }

        return true;

    }

}

return false;}

```

原来是 String 重写了 Object 的 equals 方法，把引用比较改成了值比较。

总结 : == 对于基本类型来说是值比较，对于引用类型来说是比较的是引用；而 equals 默认情况下是引用比较，只是很多类重写了 equals 方法，比如 String、Integer 等把它变成了值比较，所以一般情况下 equals 比较的是值是否相等。

3. 两个对象的 hashCode() 相同，则 equals() 也一定为 true，对吗？

不对，两个对象的 hashCode() 相同，equals() 不一定 true。

代码示例：

```

String str1 = "通话";

String str2 = "重地";

```

```
System.out.println(String.format("str1: %d | str2: %d",
str1.hashCode(), str2.hashCode()));

System.out.println(str1.equals(str2));
```

执行的结果：

```
str1: 1179395 | str2: 1179395
```

```
false
```

代码解读：很显然“通话”和“重地”的 `hashCode()` 相同，然而 `equals()` 则为 `false`，因为在散列表中，`hashCode()` 相等即两个键值对的哈希值相等，然而哈希值相等，并不一定能得出键值对相等。

4. `final` 在 java 中有什么作用？

- `final` 修饰的类叫最终类，该类不能被继承。
- `final` 修饰的方法不能被重写。
- `final` 修饰的变量叫常量，常量必须初始化，初始化之后值就不能被修改。

5. java 中的 `Math.round(-1.5)` 等于多少？

等于 `-1`。

6. `String` 属于基础的数据类型吗？

`String` 不属于基础类型，基础类型有 8 种：`byte`、`boolean`、`char`、`short`、`int`、`float`、`long`、`double`，而 `String` 属于对象。

7. java 中操作字符串都有哪些类？它们之间有什么区别？

操作字符串的类有：`String`、`StringBuffer`、`StringBuilder`。

`String` 和 `StringBuffer`、`StringBuilder` 的区别在于 `String` 声明的是不可变的对象，每次操作都会生成新的 `String` 对象，然后将指针指向新的 `String` 对象，而 `StringBuffer`、`StringBuilder` 可以在原有对象的基础上进行操作，所以在经常改变字符串内容的情况下最好不要使用 `String`。

StringBuffer 和 StringBuilder 最大的区别在于，StringBuffer 是线程安全的，而 StringBuilder 是非线程安全的，但 StringBuilder 的性能却高于 StringBuffer，所以在单线程环境下推荐使用 StringBuilder，多线程环境下推荐使用 StringBuffer。

8.String str="i"与 String str=new String("i")一样吗？

不一样，因为内存的分配方式不一样。String str="i"的方式，java 虚拟机会将其分配到常量池中；而 String str=new String("i") 则会被分到堆内存中。

9. 如何将字符串反转？

使用 StringBuilder 或者 stringBuffer 的 reverse() 方法。

示例代码：

```
// StringBuffer reverse

StringBuffer stringBuffer = new StringBuffer();

stringBuffer.append("abcdefg");

System.out.println(stringBuffer.reverse()); // gfedcba// StringBuilder reverse

StringBuilder stringBuilder = new StringBuilder();

stringBuilder.append("abcdefg");

System.out.println(stringBuilder.reverse()); // gfedcba
```

10.String 类的常用方法都有哪些？

- indexOf()：返回指定字符的索引。
- charAt()：返回指定索引处的字符。
- replace()：字符串替换。
- trim()：去除字符串两端空白。
- split()：分割字符串，返回一个分割后的字符串数组。
- getBytes()：返回字符串的 byte 类型数组。

- `length()`: 返回字符串长度。
- `toLowerCase()`: 将字符串转成小写字母。
- `toUpperCase()`: 将字符串转成大写字符。
- `substring()`: 截取字符串。
- `equals()`: 字符串比较。

11. 抽象类必须要有抽象方法吗?

不需要，抽象类不一定非要有抽象方法。

示例代码：

```
abstract class Cat {

    public static void sayHi() {

        System.out.println("hi~");

    }}

```

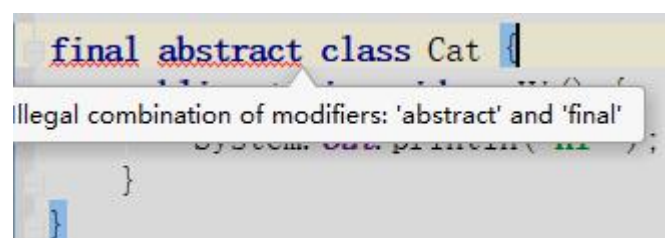
上面代码，抽象类并没有抽象方法但完全可以正常运行。

12. 普通类和抽象类有哪些区别?

- 普通类不能包含抽象方法，抽象类可以包含抽象方法。
- 抽象类不能直接实例化，普通类可以直接实例化。

13. 抽象类能使用 `final` 修饰吗?

不能，定义抽象类就是让其他类继承的，如果定义为 `final` 该类就不能被继承，这样彼此就会产生矛盾，所以 `final` 不能修饰抽象类，如下图所示，编辑器也会提示错误信息：



14. 接口和抽象类有什么区别？

- 实现：抽象类的子类使用 `extends` 来继承；接口必须使用 `implements` 来实现接口。
- 构造函数：抽象类可以有构造函数；接口不能有。
- `main` 方法：抽象类可以有 `main` 方法，并且我们能运行它；接口不能有 `main` 方法。
- 实现数量：类可以实现很多个接口；但是只能继承一个抽象类。
- 访问修饰符：接口中的方法默认使用 `public` 修饰；抽象类中的方法可以是任意访问修饰符。

15. java 中 IO 流分为几种？

按功能来分：输入流（input）、输出流（output）。

按类型来分：字节流和字符流。

字节流和字符流的区别是：字节流按 8 位传输以字节为单位输入输出数据，字符流按 16 位传输以字符为单位输入输出数据。

16. BIO、NIO、AIO 有什么区别？

- BIO：Block IO 同步阻塞式 IO，就是我们平常使用的传统 IO，它的特点是模式简单使用方便，并发处理能力低。
- NIO：New IO 同步非阻塞 IO，是传统 IO 的升级，客户端和服务端通过 Channel（通道）通讯，实现了多路复用。
- AIO：Asynchronous IO 是 NIO 的升级，也叫 NIO2，实现了异步非堵塞 IO，异步 IO 的操作基于事件和回调机制。

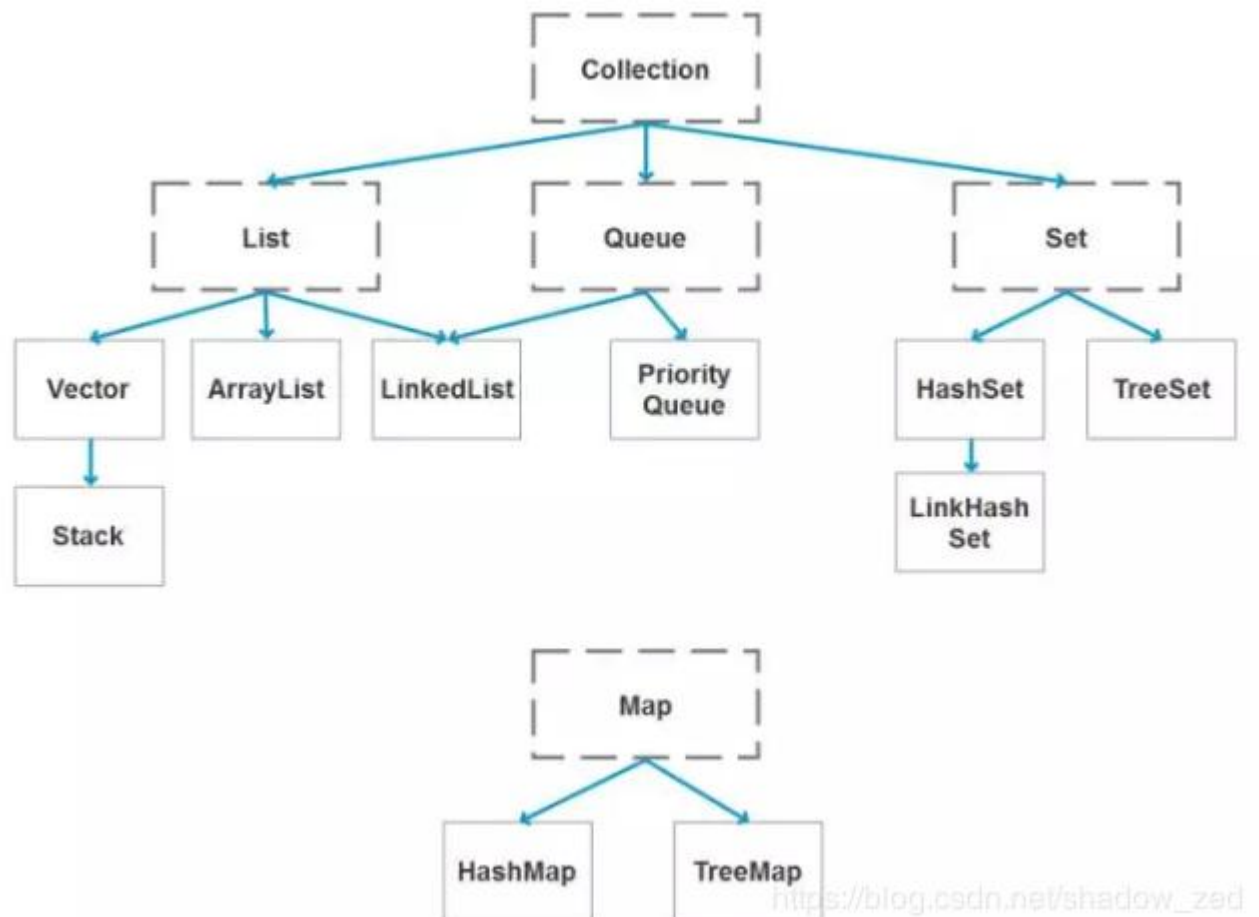
17. Files 的常用方法都有哪些？

- `Files.exists()`：检测文件路径是否存在。
- `Files.createFile()`：创建文件。
- `Files.createDirectory()`：创建文件夹。
- `Files.delete()`：删除一个文件或目录。
- `Files.copy()`：复制文件。
- `Files.move()`：移动文件。
- `Files.size()`：查看文件个数。
- `Files.read()`：读取文件。
- `Files.write()`：写入文件。

容器（二）

18. java 容器都有哪些？

常用容器的图录：



19. Collection 和 Collections 有什么区别？

`java.util.Collection` 是一个集合接口（集合类的一个顶级接口）。它提供了对集合对象进行基本操作的通用接口方法。`Collection` 接口在 Java 类库中有很多具体的实现。

`Collection` 接口的意义是为各种具体的集合提供了最大化的统一操作方式，其直接继承接口有 `List` 与 `Set`。

`Collections` 则是集合类的一个工具类/帮助类，其中提供了一系列静态方法，用于对集合中元素进行排序、搜索以及线程安全等各种操作。

20. List、Set、Map 之间的区别是什么？

比较	List	Set	Map
继承接口	Collection	Collection	
常见实现类	AbstractList(其常用子类有 ArrayList、LinkedList、Vector)	AbstractSet(其常用子类有 HashSet、LinkedHashSet、TreeSet)	HashMap、HashTable
常见方法	add(), remove(), clear(), get(), contains(), size()	add(), remove(), clear(), contains(), size()	put(), get(), remove(), clear(), containsKey(), containsValue(), keySet(), values(), size()
元素	可重复	不可重复(用 equals() 判断)	不可重复
顺序	有序	无序(实际上由 hashCode 决定)	
线程安全	Vector 线程安全		Hashtable 线程安全

https://blog.csdn.net/shadow_zs

21. HashMap 和 Hashtable 有什么区别？

HashMap 去掉了 Hashtable 的 contains 方法,但是加上了 containsValue()和 containsKey() 方法。

Hashtable 同步的,而 HashMap 是非同步的,效率上比 Hashtable 要高。

HashMap 允许空键值,而 Hashtable 不允许。

22. 如何决定使用 HashMap 还是 TreeMap？

对于在 Map 中插入、删除和定位元素这类操作,HashMap 是最好的选择。然而,假如你需要对一个有序的 key 集合进行遍历,TreeMap 是更好的选择。基于你的 collection 的大小,也许向 HashMap 中添加元素会更快,将 map 换为 TreeMap 进行有序 key 的遍历。

23. 说一下 HashMap 的实现原理？

HashMap 概述：HashMap 是基于哈希表的 Map 接口的非同步实现。此实现提供所有可选的映射操作，并允许使用 null 值和 null 键。此类不保证映射的顺序，特别是它不保证该顺序恒久不变。

HashMap 的数据结构：在 java 编程语言中，最基本的结构就是两种，一个是数组，另外一个模拟指针（引用），所有的数据结构都可以用这两个基本结构来构造的，HashMap 也不例外。HashMap 实际上是一个“链表散列”的数据结构，即数组和链表的结合体。

当我们往 Hashmap 中 put 元素时，首先根据 key 的 hashCode 重新计算 hash 值，根据 hash 值得到这个元素在数组中的位置（下标），如果该数组在该位置上已经存放了其他元素，那么在这个位置上的元素将以链表的形式存放，新加入的放在链头，最先加入的放入链尾。如果数组中该位置没有元素，就直接将该元素放到数组的该位置上。

需要注意 Jdk 1.8 中对 HashMap 的实现做了优化，当链表中的节点数据超过八个之后，该链表会转为红黑树来提高查询效率，从原来的 $O(n)$ 到 $O(\log n)$

24. 说一下 HashSet 的实现原理？

HashSet 底层由 HashMap 实现

HashSet 的值存放于 HashMap 的 key 上

HashMap 的 value 统一为 PRESENT

25. ArrayList 和 LinkedList 的区别是什么？

最明显的区别是 ArrayList 底层的数据结构是数组，支持随机访问，而 LinkedList 的底层数据结构是双向循环链表，不支持随机访问。使用下标访问一个元素，ArrayList 的时间复杂度是 $O(1)$ ，而 LinkedList 是 $O(n)$ 。

26. 如何实现数组和 List 之间的转换？

List 转换为数组：调用 ArrayList 的 toArray 方法。

数组转换为 List：调用 Arrays 的 asList 方法。

27. ArrayList 和 Vector 的区别是什么？

Vector 是同步的，而 ArrayList 不是。然而，如果你寻求在迭代的时候对列表进行改变，你应该使用 CopyOnWriteArrayList。

ArrayList 比 Vector 快，它因为有同步，不会过载。

ArrayList 更加通用，因为我们可以使用 Collections 工具类轻易地获取同步列表和只读列表。

28. Array 和 ArrayList 有何区别？

Array 可以容纳基本类型和对象，而 ArrayList 只能容纳对象。

Array 是指定大小后不可变的，而 ArrayList 大小是可变的。

Array 没有提供 ArrayList 那么多功能，比如 addAll、removeAll 和 iterator 等。

29. 在 Queue 中 poll() 和 remove() 有什么区别？

poll() 和 remove() 都是从队列中取出一个元素，但是 poll() 在获取元素失败的时候会返回空，但是 remove() 失败的时候会抛出异常。

30. 哪些集合类是线程安全的？

vector: 就比 arraylist 多了个同步化机制（线程安全），因为效率较低，现在已经不太建议使用。在 web 应用中，特别是前台页面，往往效率（页面响应速度）是优先考虑的。

statck: 堆栈类，先进后出。

hashtable: 就比 hashmap 多了个线程安全。

enumeration: 枚举，相当于迭代器。

31. 迭代器 Iterator 是什么？

迭代器是一种设计模式，它是一个对象，它可以遍历并选择序列中的对象，而开发人员不需要了解该序列的底层结构。迭代器通常被称为“轻量级”对象，因为创建它的代价小。

32. Iterator 怎么使用？有什么特点？

Java 中的 Iterator 功能比较简单，并且只能单向移动：

(1) 使用方法 `iterator()` 要求容器返回一个 Iterator。第一次调用 Iterator 的 `next()` 方法时，它返回序列的第一个元素。注意：`iterator()` 方法是 `java.lang.Iterable` 接口，被 `Collection` 继承。

(2) 使用 `next()` 获得序列中的下一个元素。

(3) 使用 `hasNext()` 检查序列中是否还有元素。

(4) 使用 `remove()` 将迭代器新返回的元素删除。

Iterator 是 Java 迭代器最简单的实现，为 List 设计的 `ListIterator` 具有更多的功能，它可以从两个方向遍历 List，也可以从 List 中插入和删除元素。

33. Iterator 和 ListIterator 有什么区别？

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。

多线程（三）

35. 并行和并发有什么区别？

并行是指两个或者多个事件在同一时刻发生；而并发是指两个或多个事件在同一时间间隔发生。

并行是在不同实体上的多个事件，并发是在同一实体上的多个事件。

在一台处理器上“同时”处理多个任务，在多台处理器上同时处理多个任务。如 hadoop 分布式集群。

所以并发编程的目标是充分的利用处理器的每一个核，以达到最高的处理性能。

36. 线程和进程的区别？

简而言之，进程是程序运行和资源分配的基本单位，一个程序至少有一个进程，一个进程至少有一个线程。进程在执行过程中拥有独立的内存单元，而多个线程共享内存资源，减少切换次数，从而效率更高。线程是进程的一个实体，是 cpu 调度和分派的基本单位，是比程序更小的能独立运行的基本单位。同一进程中的多个线程之间可以并发执行。

37. 守护线程是什么？

守护线程（即 daemon thread），是个服务线程，准确地来说就是服务其他的线程。

38. 创建线程有哪几种方式？

①. 继承 Thread 类创建线程类

定义 Thread 类的子类，并重写该类的 run 方法，该 run 方法的方法体就代表了线程要完成的任务。因此把 run() 方法称为执行体。

创建 Thread 子类的实例，即创建了线程对象。

调用线程对象的 start() 方法来启动该线程。

②. 通过 Runnable 接口创建线程类

定义 runnable 接口的实现类，并重写该接口的 run() 方法，该 run() 方法的方法体同样是该线程的线程执行体。

创建 Runnable 实现类的实例，并依此实例作为 Thread 的 target 来创建 Thread 对象，该 Thread 对象才是真正的线程对象。

调用线程对象的 start() 方法来启动该线程。

③. 通过 Callable 和 Future 创建线程

创建 Callable 接口的实现类，并实现 call() 方法，该 call() 方法将作为线程执行体，并且有返回值。

创建 Callable 实现类的实例，使用 FutureTask 类来包装 Callable 对象，该 FutureTask 对象封装了该 Callable 对象的 call() 方法的返回值。

使用 FutureTask 对象作为 Thread 对象的 target 创建并启动新线程。

调用 `FutureTask` 对象的 `get()` 方法来获得子线程执行结束后的返回值。

39. 说一下 `Runnable` 和 `Callable` 有什么区别？

有点深的问题了，也看出一个 Java 程序员学习知识的广度。

`Runnable` 接口中的 `run()` 方法的返回值是 `void`，它做的事情只是纯粹地去执行 `run()` 方法中的代码而已；

`Callable` 接口中的 `call()` 方法是有返回值的，是一个泛型，和 `Future`、`FutureTask` 配合可以用来获取异步执行的结果。

40. 线程有哪些状态？

线程通常都有五种状态，创建、就绪、运行、阻塞和死亡。

创建状态。在生成线程对象，并没有调用该对象的 `start` 方法，这是线程处于创建状态。

就绪状态。当调用了线程对象的 `start` 方法之后，该线程就进入了就绪状态，但是此时线程调度程序还没有把该线程设置为当前线程，此时处于就绪状态。在线程运行之后，从等待或者睡眠中回来之后，也会处于就绪状态。

运行状态。线程调度程序将处于就绪状态的线程设置为当前线程，此时线程就进入了运行状态，开始运行 `run` 函数当中的代码。

阻塞状态。线程正在运行的时候，被暂停，通常是为了等待某个时间的发生(比如说某项资源就绪)之后再继续运行。`sleep`, `suspend`, `wait` 等方法都可以导致线程阻塞。

死亡状态。如果一个线程的 `run` 方法执行结束或者调用 `stop` 方法后，该线程就会死亡。对于已经死亡的线程，无法再使用 `start` 方法令其进入就绪

41. `sleep()` 和 `wait()` 有什么区别？

`sleep()`：方法是线程类 (`Thread`) 的静态方法，让调用线程进入睡眠状态，让出执行机会给其他线程，等到休眠时间结束后，线程进入就绪状态和其他线程一起竞争 `cpu` 的执行时间。因为 `sleep()` 是 `static` 静态的方法，他不能改变对象的机锁，当一个 `synchronized` 块中调用了 `sleep()` 方法，线程虽然进入休眠，但是对象的机锁没有被释放，其他线程依然无法访问这个对象。

`wait()`: `wait()` 是 `Object` 类的方法, 当一个线程执行到 `wait` 方法时, 它就进入到一个和该对象相关的等待池, 同时释放对象的机锁, 使得其他线程能够访问, 可以通过 `notify`, `notifyAll` 方法来唤醒等待的线程

42. `notify()` 和 `notifyAll()` 有什么区别?

如果线程调用了对象的 `wait()` 方法, 那么线程便会处于该对象的等待池中, 等待池中的线程不会去竞争该对象的锁。

当有线程调用了对象的 `notifyAll()` 方法 (唤醒所有 `wait` 线程) 或 `notify()` 方法 (只随机唤醒一个 `wait` 线程), 被唤醒的线程便会进入该对象的锁池中, 锁池中的线程会去竞争该对象锁。也就是说, 调用了 `notify` 后只要一个线程会由等待池进入锁池, 而 `notifyAll` 会将该对象等待池内的所有线程移动到锁池中, 等待锁竞争。

优先级高的线程竞争到对象锁的概率大, 假若某线程没有竞争到该对象锁, 它还会留在锁池中, 唯有线程再次调用 `wait()` 方法, 它才会重新回到等待池中。而竞争到对象锁的线程则继续往下执行, 直到执行完了 `synchronized` 代码块, 它会释放掉该对象锁, 这时锁池中的线程会继续竞争该对象锁。

43. 线程的 `run()` 和 `start()` 有什么区别?

每个线程都是通过某个特定 `Thread` 对象所对应的方法 `run()` 来完成其操作的, 方法 `run()` 称为线程体。通过调用 `Thread` 类的 `start()` 方法来启动一个线程。

`start()` 方法来启动一个线程, 真正实现了多线程运行。这时无需等待 `run` 方法体代码执行完毕, 可以直接继续执行下面的代码; 这时此线程是处于就绪状态, 并没有运行。然后通过此 `Thread` 类调用方法 `run()` 来完成其运行状态, 这里方法 `run()` 称为线程体, 它包含了要执行的这个线程的内容, `Run` 方法运行结束, 此线程终止。然后 CPU 再调度其它线程。

`run()` 方法是在本线程里的, 只是线程里的一个函数, 而不是多线程的。如果直接调用 `run()`, 其实就相当于调用了一个普通函数而已, 直接调用 `run()` 方法必须等待 `run()` 方法执行完毕才能执行下面的代码, 所以执行路径还是只有一条, 根本就没有线程的特征, 所以在多线程执行时要使用 `start()` 方法而不是 `run()` 方法。

44. 创建线程池有哪几种方式？

①. `newFixedThreadPool(int nThreads)`

创建一个固定长度的线程池，每当提交一个任务就创建一个线程，直到达到线程池的最大数量，这时线程规模将不再变化，当线程发生未预期的错误而结束时，线程池会补充一个新的线程。

②. `newCachedThreadPool()`

创建一个可缓存的线程池，如果线程池的规模超过了处理需求，将自动回收空闲线程，而当需求增加时，则可以自动添加新线程，线程池的规模不存在任何限制。

③. `newSingleThreadExecutor()`

这是一个单线程的 Executor，它创建单个工作线程来执行任务，如果这个线程异常结束，会创建一个新的来替代它；它的特点是能确保依照任务在队列中的顺序来串行执行。

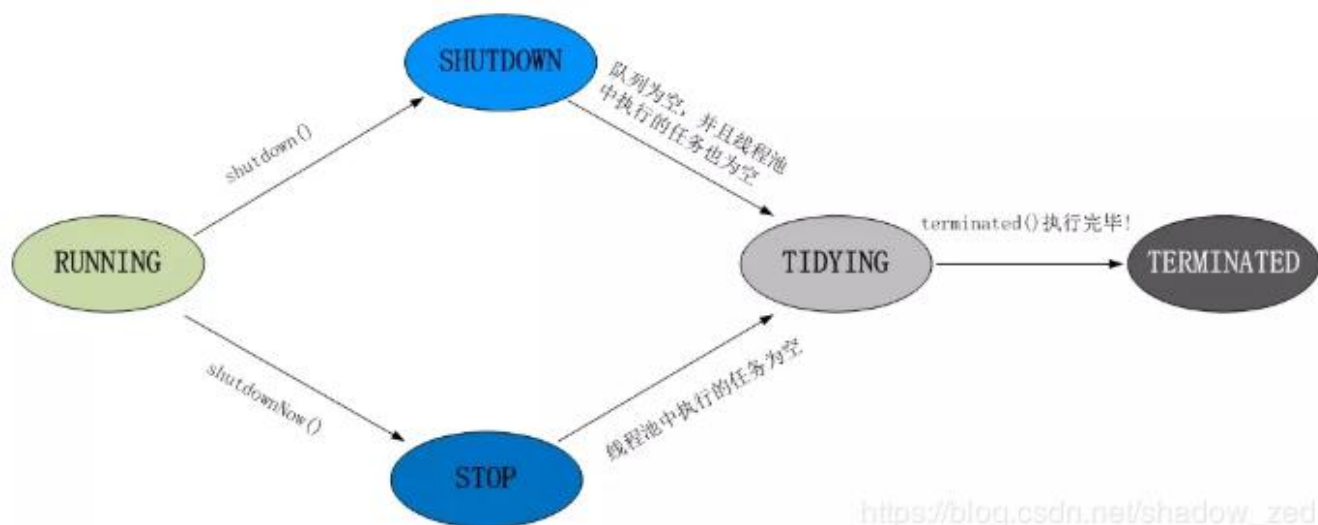
④. `newScheduledThreadPool(int corePoolSize)`

创建了一个固定长度的线程池，而且以延迟或定时的方式来执行任务，类似于 Timer。

45. 线程池都有哪些状态？

线程池有 5 种状态：Running、ShutDown、Stop、Tidying、Terminated。

线程池各个状态切换框架图：



https://blog.csdn.net/shadow_zed

46. 线程池中 submit() 和 execute() 方法有什么区别?

接收的参数不一样

submit 有返回值, 而 execute 没有

submit 方便 Exception 处理

47. 在 java 程序中怎么保证多线程的运行安全?

线程安全在三个方面体现:

原子性: 提供互斥访问, 同一时刻只能有一个线程对数据进行操作, (atomic, synchronized);

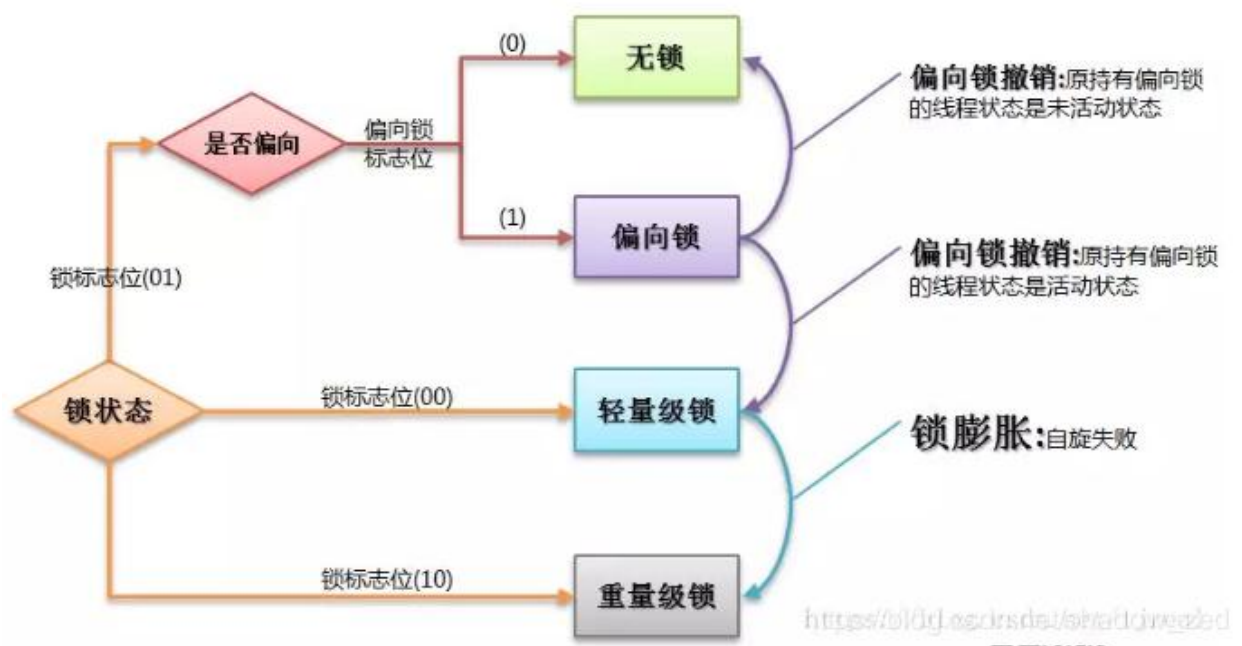
可见性: 一个线程对主内存的修改可以及时地被其他线程看到, (synchronized, volatile);

有序性: 一个线程观察其他线程中的指令执行顺序, 由于指令重排序, 该观察结果一般杂乱无序, (happens-before 原则)。

48. 多线程锁的升级原理是什么?

在 Java 中, 锁共有 4 种状态, 级别从低到高依次为: 无状态锁, 偏向锁, 轻量级锁和重量级锁状态, 这几个状态会随着竞争情况逐渐升级。锁可以升级但不能降级。

锁升级的图示过程:



49. 什么是死锁？

死锁是指两个或两个以上的进程在执行过程中，由于竞争资源或者由于彼此通信而造成的一种阻塞的现象，若无外力作用，它们都将无法推进下去。此时称系统处于死锁状态或系统产生了死锁，这些永远在互相等待的进程称为死锁进程。是操作系统层面的一个错误，是进程死锁的简称，最早在 1965 年由 Dijkstra 在研究银行家算法时提出的，它是计算机操作系统乃至整个并发程序设计领域最难处理的问题之一。

50. 怎么防止死锁？

死锁的四个必要条件：

互斥条件：进程对所分配到的资源不允许其他进程进行访问，若其他进程访问该资源，只能等待，直至占有该资源的进程使用完成后释放该资源

请求和保持条件：进程获得一定的资源之后，又对其他资源发出请求，但是该资源可能被其他进程占有，此事请求阻塞，但又对自己获得的资源保持不放

不可剥夺条件：是指进程已获得的资源，在未完成使用之前，不可被剥夺，只能在使用完后自己释放

环路等待条件：是指进程发生死锁后，若干进程之间形成一种头尾相接的循环等待资源关系

这四个条件是死锁的必要条件，只要系统发生死锁，这些条件必然成立，而只要上述条件之一不满足，就不会发生死锁。

理解了死锁的原因，尤其是产生死锁的四个必要条件，就可以最大可能地避免、预防和解除死锁。

所以，在系统设计、进程调度等方面注意如何不让这四个必要条件成立，如何确定资源的合理分配算法，避免进程永久占据系统资源。

此外，也要防止进程在处于等待状态的情况下占用资源。因此，对资源的分配要给予合理的规划。

51. ThreadLocal 是什么？有哪些使用场景？

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。

52. 说一下 synchronized 底层实现原理？

synchronized 可以保证方法或者代码块在运行时，同一时刻只有一个方法可以进入到临界区，同时它还可以保证共享变量的内存可见性。

Java 中每一个对象都可以作为锁，这是 synchronized 实现同步的基础：

普通同步方法，锁是当前实例对象

静态同步方法，锁是当前类的 class 对象

同步方法块，锁是括号里面的对象

53. synchronized 和 volatile 的区别是什么？

volatile 本质是在告诉 jvm 当前变量在寄存器（工作内存）中的值是不确定的，需要从主存中读取；synchronized 则是锁定当前变量，只有当前线程可以访问该变量，其他线程被阻塞住。

volatile 仅能使用在变量级别；synchronized 则可以使用在变量、方法、和类级别的。

volatile 仅能实现变量的修改可见性，不能保证原子性；而 synchronized 则可以保证变量的修改可见性和原子性。

volatile 不会造成线程的阻塞；synchronized 可能会造成线程的阻塞。

volatile 标记的变量不会被编译器优化；synchronized 标记的变量可以被编译器优化。

54. synchronized 和 Lock 有什么区别？

首先 synchronized 是 java 内置关键字，在 jvm 层面，Lock 是个 java 类；

synchronized 无法判断是否获取锁的状态，Lock 可以判断是否获取到锁；

synchronized 会自动释放锁（a 线程执行完同步代码会释放锁；b 线程执行过程中发生异常会释放锁），Lock 需在 finally 中手工释放锁（unlock() 方法释放锁），否则容易造成线程死锁；

用 `synchronized` 关键字的两个线程 1 和线程 2，如果当前线程 1 获得锁，线程 2 线程等待。如果线程 1 阻塞，线程 2 则会一直等待下去，而 `Lock` 锁就不一定会等待下去，如果尝试获取不到锁，线程可以不用一直等待就结束了；

`synchronized` 的锁可重入、不可中断、非公平，而 `Lock` 锁可重入、可判断、可公平（两者皆可）；

`Lock` 锁适合大量同步的代码的同步问题，`synchronized` 锁适合代码少量的同步问题。

55. `synchronized` 和 `ReentrantLock` 区别是什么？

`synchronized` 是和 `if`、`else`、`for`、`while` 一样的关键字，`ReentrantLock` 是类，这是二者的本质区别。既然 `ReentrantLock` 是类，那么它就提供了比 `synchronized` 更多更灵活的特性，可以被继承、可以有方法、可以有各种各样的类变量，`ReentrantLock` 比 `synchronized` 的扩展性体现在几点上：

`ReentrantLock` 可以对获取锁的等待时间进行设置，这样就避免了死锁

`ReentrantLock` 可以获取各种锁的信息

`ReentrantLock` 可以灵活地实现多路通知

另外，二者的锁机制其实也是不一样的：`ReentrantLock` 底层调用的是 `Unsafe` 的 `park` 方法加锁，`synchronized` 操作的应该是对象头中 `mark word`。

56. 说一下 `atomic` 的原理？

`Atomic` 包中的类基本的特性就是在多线程环境下，当有多个线程同时对单个（包括基本类型及引用类型）变量进行操作时，具有排他性，即当多个线程同时对该变量的值进行更新时，仅有一个线程能成功，而未成功的线程可以向自旋锁一样，继续尝试，一直等到执行成功。

`Atomic` 系列的类中的核心方法都会调用 `unsafe` 类中的几个本地方法。我们需要先知道一个东西就是 `Unsafe` 类，全名为：`sun.misc.Unsafe`，这个类包含了大量的对 C 代码的操作，包括很多直接内存分配以及原子操作的调用，而它之所以标记为非安全的，是告诉你这个里面大量的方法调用都会存在安全隐患，需要小心使用，否则会导致严重的后果，例如在通过 `unsafe` 分配内存的时候，如果自己指定某些区域可能会导致一些类似 C++ 一样的指针越界到其他进程的问题。

反射（四）

57. 什么是反射？

反射主要是指程序可以访问、检测和修改它本身状态或行为的一种能力

Java 反射：

在 Java 运行时环境中，对于任意一个类，能否知道这个类有哪些属性和方法？对于任意一个对象，能否调用它的任意一个方法

Java 反射机制主要提供了以下功能：

在运行时判断任意一个对象所属的类。

在运行时构造任意一个类的对象。

在运行时判断任意一个类所具有的成员变量和方法。

在运行时调用任意一个对象的方法。

58. 什么是 java 序列化？什么情况下需要序列化？

简单说就是为了保存在内存中的各种对象的状态（也就是实例变量，不是方法），并且可以把保存的对象状态再读出来。虽然你可以用你自己的各种各样的方法来保存 object states，但是 Java 给你提供一种应该比你自己的好的保存对象状态的机制，那就是序列化。

什么情况下需要序列化：

- a) 当你想把的内存中的对象状态保存到一个文件中或者数据库中时候；
- b) 当你想用套接字在网络上传送对象的时候；
- c) 当你想通过 RMI 传输对象的时候；

59. 动态代理是什么？有哪些应用？

动态代理：

当想要给实现了某个接口的类中的方法，加一些额外的处理。比如说加日志，加事务等。可以给这个类创建一个代理，故名思议就是创建一个新的类，这个类不仅包含原来类方法的功

能，而且还在原来的基础上添加了额外处理的新类。这个代理类并不是定义好的，是动态生成的。具有解耦意义，灵活，扩展性强。

动代理的应用：

Spring 的 AOP

加事务

加权限

加日志

60. 怎么实现动态代理？

首先必须定义一个接口，还要有一个 `InvocationHandler` (将实现接口的类的对象传递给它) 处理类。再有一个工具类 `Proxy` (习惯性将其称为代理类，因为调用他的 `newInstance()` 可以产生代理对象，其实他只是一个产生代理对象的工具类)。利用到 `InvocationHandler`，拼接代理类源码，将其编译生成代理类的二进制码，利用加载器加载，并将其实例化产生代理对象，最后返回。

对象拷贝（五）

61. 为什么要使用克隆？

想对一个对象进行处理，又想保留原有的数据进行接下来的操作，就需要克隆了，Java 语言中克隆针对的是类的实例。

62. 如何实现对象克隆？

有两种方式：

实现 `Cloneable` 接口并重写 `Object` 类中的 `clone()` 方法；

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，代码如下：

```
import java.io.ByteArrayInputStream;import java.io.ByteArrayOutputStream;import
java.io.ObjectInputStream;import java.io.ObjectOutputStream;import
java.io.Serializable;

public class MyUtil {
```

```

private MyUtil() {

    throw new AssertionError();

}

@SuppressWarnings("unchecked")

public static <T extends Serializable> T clone(T obj) throws Exception {

    ByteArrayOutputStream bout = new ByteArrayOutputStream();

    ObjectOutputStream oos = new ObjectOutputStream(bout);

    oos.writeObject(obj);

    ByteArrayInputStream bin = new ByteArrayInputStream(bout.toByteArray());

    ObjectInputStream ois = new ObjectInputStream(bin);

    return (T) ois.readObject();

    // 说明：调用 ByteArrayInputStream 或 ByteArrayOutputStream 对象的 close
    方法没有任何意义

    // 这两个基于内存的流只要垃圾回收器清理对象就能够释放资源，这一点不同于
    对外部资源（如文件流）的释放

}}

```

下面是测试代码：

```

import java.io.Serializable;

//class Person implements Serializable {

    private static final long serialVersionUID = -9102017020286042305L;

```

```
private String name;    // 姓名

private int age;        // 年龄

private Car car;        // 座驾

public Person(String name, int age, Car car) {

    this.name = name;

    this.age = age;

    this.car = car;

}

public String getName() {

    return name;

}

public void setName(String name) {

    this.name = name;

}

public int getAge() {

    return age;

}

public void setAge(int age) {

    this.age = age;

}
```

```

    public Car getCar() {

        return car;

    }

    public void setCar(Car car) {

        this.car = car;

    }

    @Override

    public String toString() {

        return "Person [name=" + name + ", age=" + age + ", car=" + car + "]";

    }

}

/**
 * 小汽车类
 *
 * @author nnngu
 *
 */
class Car implements Serializable {

    private static final long serialVersionUID = -5713945027627603702L;

    private String brand; // 品牌

```

```
private int maxSpeed;        // 最高时速

public Car(String brand, int maxSpeed) {

    this.brand = brand;

    this.maxSpeed = maxSpeed;

}

public String getBrand() {

    return brand;

}

public void setBrand(String brand) {

    this.brand = brand;

}

public int getMaxSpeed() {

    return maxSpeed;

}

public void setMaxSpeed(int maxSpeed) {

    this.maxSpeed = maxSpeed;

}

@Override

public String toString() {
```



```

        return "Car [brand=" + brand + ", maxSpeed=" + maxSpeed + "]";
    }
}

class CloneTest {

    public static void main(String[] args) {

        try {

            Person p1 = new Person("郭靖", 33, new Car("Benz", 300));

            Person p2 = MyUtil.clone(p1);    // 深度克隆

            p2.getCar().setBrand("BYD");

            // 修改克隆的 Person 对象 p2 关联的汽车对象的品牌属性

            // 原来的 Person 对象 p1 关联的汽车不会受到任何影响

            // 因为在克隆 Person 对象时其关联的汽车对象也被克隆了

            System.out.println(p1);

        } catch (Exception e) {

            e.printStackTrace();

        }

    }
}

```

注意：基于序列化和反序列化实现的克隆不仅仅是深度克隆，更重要的是通过泛型限定，可以检查出要克隆的对象是否支持序列化，这项检查是编译器完成的，不是在运行时抛出异常，这种方案明显优于使用 `Object` 类的 `clone` 方法克隆对象。让问题在编译的时候暴露出来总是好过把问题留到运行时。

63. 深拷贝和浅拷贝区别是什么？

- 浅拷贝只是复制了对象的引用地址，两个对象指向同一个内存地址，所以修改其中任意的值，另一个值都会随之变化，这就是浅拷贝（例：`assign()`）
- 深拷贝是将对象及值复制过来，两个对象修改其中任意的值另一个值不会改变，这就是深拷贝（例：`JSON.parse()` 和 `JSON.stringify()`，但是此方法无法复制函数类型）

JavaWeb（六）

64. jsp 和 servlet 有什么区别？

jsp 经编译后就变成了 `Servlet`。（JSP 的本质就是 `Servlet`，JVM 只能识别 `java` 的类，不能识别 JSP 的代码，Web 容器将 JSP 的代码编译成 JVM 能够识别的 `java` 类）

jsp 更擅长表现于页面显示，servlet 更擅长于逻辑控制。

`Servlet` 中没有内置对象，Jsp 中的内置对象都是必须通过 `HttpServletRequest` 对象，`HttpServletResponse` 对象以及 `HttpServlet` 对象得到。

Jsp 是 `Servlet` 的一种简化，使用 Jsp 只需要完成程序员需要输出到客户端的内容，Jsp 中的 `Java` 脚本如何镶嵌到一个类中，由 Jsp 容器完成。而 `Servlet` 则是个完整的 `Java` 类，这个类的 `Service` 方法用于生成对客户端的响应。

65. 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。

66. 说一下 jsp 的 4 种作用域?

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。

67. session 和 cookie 有什么区别?

由于 HTTP 协议是无状态的协议，所以服务端需要记录用户的状态时，就需要用某种机制来识具体的用户，这个机制就是 Session. 典型的场景比如购物车，当你点击下单按钮时，由于 HTTP 协议无状态，所以并不知道是哪个用户操作的，所以服务端要为特定的用户创建了特定的 Session，用于标识这个用户，并且跟踪用户，这样才知道购物车里面有几本书。这个 Session 是保存在服务端的，有一个唯一标识。在服务端保存 Session 的方法很多，内存、数据库、文件都有。集群的时候也要考虑 Session 的转移，在大型的网站，一般会有专门的 Session 服务器集群，用来保存用户会话，这个时候 Session 信息都是放在内存的，使用一些缓存服务比如 Memcached 之类的来放 Session。

思考一下服务端如何识别特定的客户？这个时候 Cookie 就登场了。每次 HTTP 请求的时候，客户端都会发送相应的 Cookie 信息到服务端。实际上大多数的应用都是用 Cookie 来实现 Session 跟踪的，第一次创建 Session 的时候，服务端会在 HTTP 协议中告诉客户端，需要在 Cookie 里面记录一个 Session ID，以后每次请求把这个会话 ID 发送到服务器，我就知道你是谁了。有人问，如果客户端的浏览器禁用了 Cookie 怎么办？一般这种情况下，会使

用一种叫做 URL 重写的技术来进行会话跟踪，即每次 HTTP 交互，URL 后面都会被附加上一个诸如 `sid=xxxxx` 这样的参数，服务端据此来识别用户。

Cookie 其实还可以用在一些方便用户的场景下，设想你某次登陆过一个网站，下次登录的时候不想再次输入账号了，怎么办？这个信息可以写到 Cookie 里面，访问网站的时候，网站页面的脚本可以读取这个信息，就自动帮你把用户名给填了，能够方便一下用户。这也是 Cookie 名称的由来，给用户的一点甜头。所以，总结一下：Session 是在服务端保存的一个数据结构，用来跟踪用户的状态，这个数据可以保存在集群、数据库、文件中；Cookie 是客户端保存用户信息的一种机制，用来记录用户的一些信息，也是实现 Session 的一种方式。

68. 说一下 session 的工作原理？

其实 session 是一个存在服务器上的类似于一个散列表的文件。里面存有我们需要的信息，在我们需要用的时候可以从里面取出来。类似于一个大号的 map 吧，里面的键存储的是用户的 sessionid，用户向服务器发送请求的时候会带上这个 sessionid。这时就可以从中取出对应的值了。

69. 如果客户端禁止 cookie 能实现 session 还能用吗？

Cookie 与 Session，一般认为是两个独立的东西，Session 采用的是在服务器端保持状态的方案，而 Cookie 采用的是在客户端保持状态的方案。但为什么禁用 Cookie 就不能得到 Session 呢？因为 Session 是用 Session ID 来确定当前对话所对应的服务器 Session，而 Session ID 是通过 Cookie 来传递的，禁用 Cookie 相当于失去了 Session ID，也就得不到 Session 了。

假定用户关闭 Cookie 的情况下使用 Session，其实现途径有以下几种：

设置 `php.ini` 配置文件中的“`session.use_trans_sid = 1`”，或者编译时打开打开了“`--enable-trans-sid`”选项，让 PHP 自动跨页传递 Session ID。

手动通过 URL 传值、隐藏表单传递 Session ID。

用文件、数据库等形式保存 Session ID，在跨页过程中手动调用。

70. spring mvc 和 struts 的区别是什么？

拦截机制的不同

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

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

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

底层框架的不同

Struts2 采用 Filter (StrutsPrepareAndExecuteFilter) 实现，SpringMVC

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

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。

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。

71. 如何避免 sql 注入？

PreparedStatement（简单又有效的方法）

使用正则表达式过滤传入的参数

字符串过滤

JSP 中调用该函数检查是否包函非法字符

JSP 页面判断代码

72. 什么是 XSS 攻击，如何避免？

XSS 攻击又称 CSS, 全称 Cross Site Script（跨站脚本攻击），其原理是攻击者向有 XSS 漏洞的网站中输入恶意的 HTML 代码，当用户浏览该网站时，这段 HTML 代码会自动执行，从而达到攻击的目的。XSS 攻击类似于 SQL 注入攻击，SQL 注入攻击中以 SQL 语句作为用户输入，从而达到查询/修改/删除数据的目的，而在 xss 攻击中，通过插入恶意脚本，实现对用户浏览器的控制，获取用户的一些信息。XSS 是 Web 程序中常见的漏洞，XSS 属于被动式且用于客户端的攻击方式。

XSS 防范的总体思路是：对输入(和 URL 参数)进行过滤，对输出进行编码。

73. 什么是 CSRF 攻击，如何避免？

CSRF（Cross-site request forgery）也被称为 one-click attack 或者 session riding，中文全称是叫**跨站请求伪造**。一般来说，攻击者通过伪造用户的浏览器的请求，向访问一个用户自己曾经认证访问过的网站发送出去，使目标网站接收并误以为是用户的真实操作而去执行命令。常用于盗取账号、转账、发送虚假消息等。攻击者利用网站对请求的验证漏洞而实现这样的攻击行为，网站能够确认请求来源于用户的浏览器，却不能验证请求是否源于用户的真实意愿下的操作行为。

如何避免：

1. 验证 HTTP Referer 字段

HTTP 头中的 Referer 字段记录了该 HTTP 请求的来源地址。在通常情况下，访问一个安全受限页面的请求来自于同一个网站，而如果黑客要对其实施 CSRF 攻击，他一般只能在他自己的网站构造请求。因此，可以通过验证 Referer 值来防御 CSRF 攻击。

2. 使用验证码

关键操作页面加上验证码，后台收到请求后通过判断验证码可以防御 CSRF。但这种方法对用户不太友好。

3. 在请求地址中添加 token 并验证

CSRF 攻击之所以能够成功，是因为黑客可以完全伪造用户的请求，该请求中所有的用户验证信息都是存在于 cookie 中，因此黑客可以在不知道这些验证信息的情况下直接利用用户自己的 cookie 来通过安全验证。要抵御 CSRF，关键在于在请求中放入黑客所不能伪造的信息，并且该信息不存在于 cookie 之中。可以在 HTTP 请求中以参数的形式加入一个随机产生的 token，并在服务器端建立一个拦截器来验证这个 token，如果请求中没有 token 或者 token 内容不正确，则认为可能是 CSRF 攻击而拒绝该请求。这种方法要比检查 Referer 要安全一些，token 可以在用户登陆后产生并放于 session 之中，然后在每次请求时把 token 从 session 中拿出，与请求中的 token 进行比对，但这种方法的难点在于如何把 token 以参数的形式加入请求。

对于 GET 请求，token 将附在请求地址之后，这样 URL 就变成 `http://url?csrftoken=tokenvalue`。

而对于 POST 请求来说，要在 form 的最后加上 `<input type="hidden" name="csrftoken" value="tokenvalue"/>`，这样就把 token 以参数的形式加入请求了。

4. 在 HTTP 头中自定义属性并验证

这种方法也是使用 token 并进行验证，和上一种方法不同的是，这里并不是把 token 以参数的形式置于 HTTP 请求之中，而是把它放到 HTTP 头中自定义的属性里。通过 `XMLHttpRequest` 这个类，可以一次性给所有该类请求加上 `csrftoken` 这个 HTTP 头属性，并把 token 值放入其中。这样解决了上种方法在请求中加入 token 的不便，同时，通过 `XMLHttpRequest` 请求的地址不会被记录到浏览器的地址栏，也不用担心 token 会透过 Referer 泄露到其他网站中去。

异常（七）

74. throw 和 throws 的区别？

throws 是用来声明一个方法可能抛出的所有异常信息，throws 是将异常声明但是不处理，而是将异常往上传，谁调用我就交给谁处理。而 throw 则是指抛出的一个具体的异常类型。

75. final、finally、finalize 有什么区别？

final 可以修饰类、变量、方法，修饰类表示该类不能被继承、修饰方法表示该方法不能被重写、修饰变量表示该变量是一个常量不能被重新赋值。

finally 一般作用在 try-catch 代码块中，在处理异常的时候，通常我们将一定要执行的代码方法 finally 代码块中，表示不管是否出现异常，该代码块都会执行，一般用来存放一些关闭资源的代码。

finalize 是一个方法，属于 Object 类的一个方法，而 Object 类是所有类的父类，该方法一般由垃圾回收器来调用，当我们调用 System 的 gc() 方法的时候，由垃圾回收器调用 finalize(), 回收垃圾。

76. try-catch-finally 中哪个部分可以省略？

答：catch 可以省略

原因：

更为严格的说法其实是：try 只适合处理运行时异常，try+catch 适合处理运行时异常+普通异常。也就是说，如果你只用 try 去处理普通异常却不加以 catch 处理，编译是通不过的，因为编译器硬性规定，普通异常如果选择捕获，则必须用 catch 显示声明以便进一步处理。而运行时异常在编译时没有如此规定，所以 catch 可以省略，你加上 catch 编译器也觉得无可厚非。

理论上，编译器看任何代码都不顺眼，都觉得可能有潜在的问题，所以你即使对所有代码加上 try，代码在运行期时也只不过是在正常运行的基础上加一层皮。但是你一旦对一段代码加上 try，就等于显示地承诺编译器，对这段代码可能抛出的异常进行捕获而非向上抛出处理。如果是普通异常，编译器要求必须用 catch 捕获以便进一步处理；如果运行时异常，捕获然后丢弃并且+finally 扫尾处理，或者加上 catch 捕获以便进一步处理。

至于加上 finally，则是在不管有没捕获异常，都要进行的“扫尾”处理。

77. try-catch-finally 中, 如果 catch 中 return 了, finally 还会执行吗?

答: 会执行, 在 return 前执行。

代码示例 1:

```
/*  
  
* java 面试题--如果 catch 里面有 return 语句, finally 里面的代码还会执行吗?  
  
*/public class FinallyDemo2 {  
  
    public static void main(String[] args) {  
  
        System.out.println(getInt());  
  
    }  
  
    public static int getInt() {  
  
        int a = 10;  
  
        try {  
  
            System.out.println(a / 0);  
  
            a = 20;  
  
        } catch (ArithmeticException e) {  
  
            a = 30;  
  
            return a;  
  
        }  
  
        /*  
  
        * return a 在程序执行到这一步的时候, 这里不是 return a 而是 return  
30; 这个返回路径就形成了  
  
        * 但是呢, 它发现后面还有 finally, 所以继续执行 finally 的内容, a=40
```

** 再次回到以前的路径,继续走 return 30, 形成返回路径之后, 这里的 a 就不是 a 变量了, 而是常量 30*

```
        */

    } finally {

        a = 40;

    }

//    return a;

}}
```

执行结果: 30

代码示例 2:

```
package com.java_02;

/*

* java 面试题--如果 catch 里面有 return 语句, finally 里面的代码还会执行吗?

*/public class FinallyDemo2 {

    public static void main(String[] args) {

        System.out.println(getInt());

    }

    public static int getInt() {

        int a = 10;
```

```

try {

    System.out.println(a / 0);

    a = 20;

} catch (ArithmeticException e) {

    a = 30;

    return a;

    /*

        * return a 在程序执行到这一步的时候，这里不是 return a 而是 return
        30；这个返回路径就形成了

        * 但是呢，它发现后面还有 finally，所以继续执行 finally 的内容，a=40

        * 再次回到以前的路径，继续走 return 30，形成返回路径之后，这里的 a 就
        不是 a 变量了，而是常量 30

        */

    } finally {

        a = 40;

        return a; //如果这样，就又重新形成了一条返回路径，由于只能通过 1 个
        return 返回，所以这里直接返回 40

    }

//    return a;

}}

```

执行结果：40

78. 常见的异常类有哪些？

- `NullPointerException`: 当应用程序试图访问空对象时，则抛出该异常。
- `SQLException`: 提供关于数据库访问错误或其他错误信息的异常。
- `IndexOutOfBoundsException`: 指示某排序索引（例如对数组、字符串或向量的排序）超出范围时抛出。
- `NumberFormatException`: 当应用程序试图将字符串转换成一种数值类型，但该字符串不能转换为适当格式时，抛出该异常。
- `FileNotFoundException`: 当试图打开指定路径名表示的文件失败时，抛出此异常。
- `IOException`: 当发生某种 I/O 异常时，抛出此异常。此类是失败或中断的 I/O 操作生成的异常的通用类。
- `ClassCastException`: 当试图将对象强制转换为不是实例的子类时，抛出该异常。
- `ArrayStoreException`: 试图将错误类型的对象存储到一个对象数组时抛出的异常。
- `IllegalArgumentException`: 抛出的异常表明向方法传递了一个不合法或不正确的参数。
- `ArithmeticException`: 当出现异常的运算条件时，抛出此异常。例如，一个整数“除以零”时，抛出此类的一个实例。
- `NegativeArraySizeException`: 如果应用程序试图创建大小为负的数组，则抛出该异常。
- `NoSuchMethodException`: 无法找到某一特定方法时，抛出该异常。
- `SecurityException`: 由安全管理器抛出的异常，指示存在安全侵犯。
- `UnsupportedOperationException`: 当不支持请求的操作时，抛出该异常。
- `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。

网络（八）

79. http 响应码 301 和 302 代表的是什么？有什么区别？

答：301, 302 都是 HTTP 状态的编码，都代表着某个 URL 发生了转移。

区别：

301 redirect: 301 代表永久性转移 (Permanently Moved)。

302 redirect: 302 代表暂时性转移 (Temporarily Moved)。

80. forward 和 redirect 的区别？

Forward 和 Redirect 代表了两种请求转发方式：直接转发和间接转发。

直接转发方式（Forward），客户端和浏览器只发出一次请求，Servlet、HTML、JSP 或其它信息资源，由第二个信息资源响应该请求，在请求对象 request 中，保存的对象对于每个信息资源是共享的。

间接转发方式（Redirect）实际是两次 HTTP 请求，服务器端在响应第一次请求的时候，让浏览器再向另外一个 URL 发出请求，从而达到转发的目的。

举个通俗的例子：

直接转发就相当于：“A 找 B 借钱，B 说没有，B 去找 C 借，借到借不到都会把消息传递给 A”；

间接转发就相当于：“A 找 B 借钱，B 说没有，让 A 去找 C 借”。

81. 简述 tcp 和 udp 的区别？

- TCP 面向连接（如打电话要先拨号建立连接）；UDP 是无连接的，即发送数据之前不需要建立连接。
- TCP 提供可靠的服务。也就是说，通过 TCP 连接传送的数据，无差错，不丢失，不重复，且按序到达；UDP 尽最大努力交付，即不保证可靠交付。
- TCP 通过校验和，重传控制，序号标识，滑动窗口、确认应答实现可靠传输。如丢包时的重发控制，还可以对次序乱掉的分包进行顺序控制。
- UDP 具有较好的实时性，工作效率比 TCP 高，适用于对高速传输和实时性有较高的通信或广播通信。
- 每一条 TCP 连接只能是点到点的；UDP 支持一对一，一对多，多对一和多对多的交互通信。
- TCP 对系统资源要求较多，UDP 对系统资源要求较少。

82. tcp 为什么要三次握手，两次不行吗？为什么？

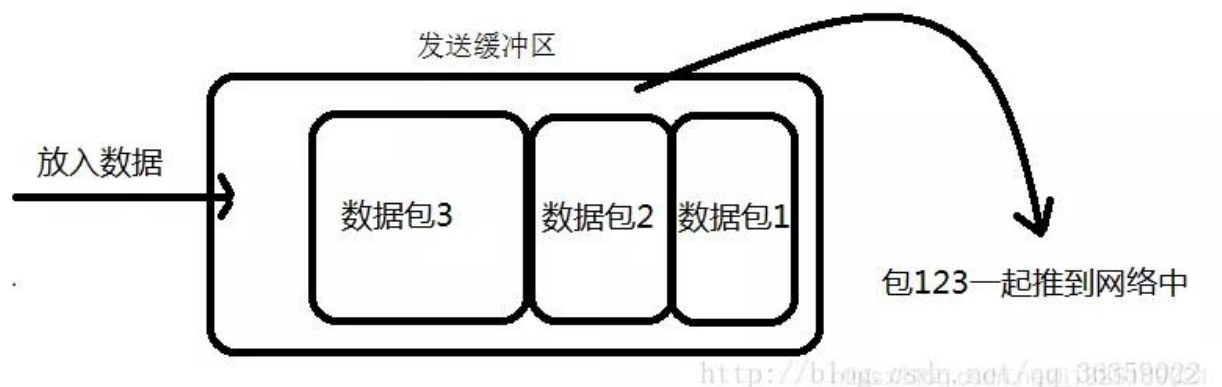
为了实现可靠数据传输，TCP 协议的通信双方，都必须维护一个序列号，以标识发送出去的数据包中，哪些是已经被对方收到的。三次握手的过程即是通信双方相互告知序列号起始值，并确认对方已经收到了序列号起始值的必经步骤。

如果只是两次握手，至多只有连接发起方的起始序列号能被确认，另一方选择的序列号则得不到确认。

83. 说一下 tcp 粘包是怎么产生的？

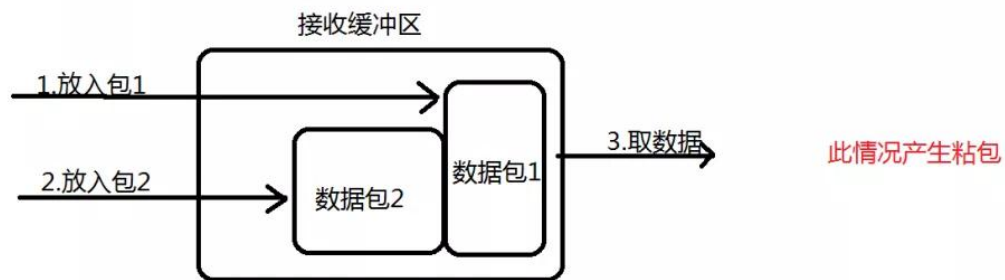
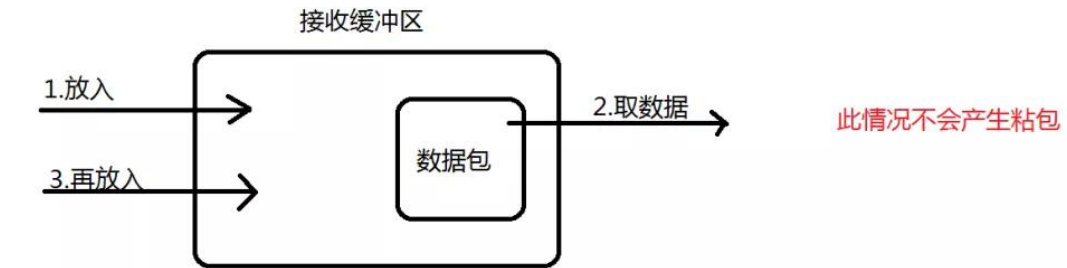
①. 发送方产生粘包

采用 TCP 协议传输数据的客户端与服务器经常是保持一个长连接的状态（一次连接发一次数据不存在粘包），双方在连接不断开的情况下，可以一直传输数据；但当发送的数据包过于小时，那么 TCP 协议默认会启用 Nagle 算法，将这些较小的数据包进行合并发送（缓冲区数据发送是一个堆压的过程）；这个合并过程就是在发送缓冲区中进行的，也就是说数据发送出来它已经是粘包的状态了。



②. 接收方产生粘包

接收方采用 TCP 协议接收数据时的过程是这样的：数据到底接收方，从网络模型的下方传递至传输层，传输层的 TCP 协议处理是将其放置接收缓冲区，然后由应用层来主动获取（C 语言用 `recv`、`read` 等函数）；这时会出现一个问题，就是我们在程序中调用的读取数据函数不能及时的把缓冲区中的数据拿出来，而下一个数据又到来并有一部分放入的缓冲区末尾，等我们读取数据时就是一个粘包。（放数据的速度 > 应用层拿数据速度）



<http://www.it-ebooks.info/files/326356023>

84. OSI 的七层模型都有哪些？

- 应用层：网络服务与最终用户的一个接口。
- 表示层：数据的表示、安全、压缩。
- 会话层：建立、管理、终止会话。
- 传输层：定义传输数据的协议端口号，以及流控和差错校验。
- 网络层：进行逻辑地址寻址，实现不同网络之间的路径选择。
- 数据链路层：建立逻辑连接、进行硬件地址寻址、差错校验等功能。
- 物理层：建立、维护、断开物理连接。

85. get 和 post 请求有哪些区别？

- GET 在浏览器回退时是无害的，而 POST 会再次提交请求。
- GET 产生的 URL 地址可以被 Bookmark，而 POST 不可以。
- GET 请求会被浏览器主动 cache，而 POST 不会，除非手动设置。
- GET 请求只能进行 url 编码，而 POST 支持多种编码方式。
- GET 请求参数会被完整保留在浏览器历史记录里，而 POST 中的参数不会被保留。
- GET 请求在 URL 中传送的参数是有长度限制的，而 POST 没有。
- 参数的数据类型，GET 只接受 ASCII 字符，而 POST 没有限制。
- GET 比 POST 更不安全，因为参数直接暴露在 URL 上，所以不能用来传递敏感信息。
- GET 参数通过 URL 传递，POST 放在 Request body 中。

86. 如何实现跨域?

方式一：图片 ping 或 script 标签跨域

图片 ping 常用于跟踪用户点击页面或动态广告曝光次数。

script 标签可以得到从其他来源数据，这也是 JSONP 依赖的根据。

方式二：JSONP 跨域

JSONP (JSON with Padding) 是数据格式 JSON 的一种“使用模式”，可以让网页从别的网域要数据。根据 XMLHttpRequest 对象受到同源策略的影响，而利用

- 只能使用 Get 请求
- 不能注册 success、error 等事件监听函数，不能很容易的确定 JSONP 请求是否失败
- JSONP 是从其他域中加载代码执行，容易受到跨站请求伪造的攻击，其安全性无法确保

方式三：CORS

Cross-Origin Resource Sharing(CORS)跨域资源共享是一份浏览器技术的规范,提供了 Web 服务从不同域传来沙盒脚本的方法,以避开浏览器的同源策略,确保安全的跨域数据传输。现代浏览器使用 CORS 在 API 容器如 XMLHttpRequest 来减少 HTTP 请求的风险来源。与 JSONP 不同, CORS 除了 GET 要求方法以外也支持其他的 HTTP 要求。服务器一般需要增加如下响应头的一种或几种:

```
Access-Control-Allow-Origin: *  
  
Access-Control-Allow-Methods: POST, GET, OPTIONS  
  
Access-Control-Allow-Headers: X-PINGOTHER, Content-Type  
  
Access-Control-Max-Age: 86400
```

跨域请求默认不会携带 Cookie 信息, 如果需要携带, 请配置下述参数:

```
"Access-Control-Allow-Credentials": true // Ajax 设置 withCredentials": true
```


方式四: window.name+iframe

window.name 通过在 iframe (一般动态创建 i) 中加载跨域 HTML 文件来起作用。然后, HTML 文件将传递给请求者的字符串内容赋值给 window.name。然后, 请求者可以检索 window.name 值作为响应。

- iframe 标签的跨域能力;
- indow.name 属性值在文档刷新后依旧存在的能力 (且最大允许 2M 左右)。

每个 iframe 都有包裹它的 window, 而这个 window 是 top window 的子窗口。contentWindow 属性返回元素的 Window 对象。你可以使用这个 Window 对象来访问 iframe 的文档及其内部 DOM。

```
<!--  
  
  下述用端口  
  
  10000 表示: domainA  
  
  10001 表示: domainB-->  
  
<!-- localhost:10000 --><script>  
  
  var iframe = document.createElement('iframe');  
  
  iframe.style.display = 'none'; // 隐藏  
  
  
  var state = 0; // 防止页面无限刷新  
  
  iframe.onload = function() {  
  
    if(state === 1) {  
  
      console.log(JSON.parse(iframe.contentWindow.name));  
  
      // 清除创建的 iframe  
  
      iframe.contentWindow.document.write('');
```

```

        iframe.contentWindow.close();

        document.body.removeChild(iframe);

    } else if(state === 0) {

        state = 1;

        // 加载完成, 指向当前域, 防止错误(proxy.html 为空白页面)

        // Blocked a frame with origin "http://localhost:10000" from accessing
a cross-origin frame.

        iframe.contentWindow.location = 'http://localhost:10000/proxy.html';

    }

};

iframe.src = 'http://localhost:10001';

document.body.appendChild(iframe);</script>

<!-- localhost:10001 --><!DOCTYPE html>...<script>

window.name = JSON.stringify({a: 1, b: 2});</script></html>

```

方式五: window.postMessage()

HTML5 新特性, 可以用来向其他所有的 window 对象发送消息。需要注意的是我们必须要保证所有的脚本执行完才发送 MessageEvent, 如果在函数执行的过程中调用了它, 就会让后面的函数超时无法执行。

下述代码实现了跨域存储 localStorage

```
<!--
```

下述用端口

10000 表示: domainA

10001 表示: domainB-->

```
<!-- localhost:10000 --><iframe src="http://localhost:10001/msg.html"
name="myPostMessage" style="display:none;"></iframe>
```

```
<script>
```

```
function main() {
```

```
    LSsetItem('test', 'Test: ' + new Date());
```

```
    LSgetItem('test', function(value) {
```

```
        console.log('value: ' + value);
```

```
    });
```

```
    LSremoveItem('test');
```

```
}
```

```
var callbacks = {};
```

```
window.addEventListener('message', function(event) {
```

```
    if (event.source === frames['myPostMessage']) {
```

```
        console.log(event)
```

```
        var data = /^#localStorage#(\d+) (null)?#([\S\s]*)/.exec(event.data);
```

```
        if (data) {
```

```

        if (callbacks[data[1]]) {

            callbacks[data[1]](data[2] === 'null' ? null : data[3]);

        }

        delete callbacks[data[1]];

    }

}

}, false);

var domain = '*';

// 增加

function LSsetItem(key, value) {

    var obj = {

        setItem: key,

        value: value

    };

    frames['myPostMessage'].postMessage(JSON.stringify(obj), domain);

}

// 获取

function LSgetItem(key, callback) {

    var identifier = new Date().getTime();

```

```

    var obj = {

        identifier: identifier,

        getItem: key

    };

    callbacks[identifier] = callback;

    frames['myPostMessage'].postMessage(JSON.stringify(obj), domain);

}

// 删除

function LSremoveItem(key) {

    var obj = {

        removeItem: key

    };

    frames['myPostMessage'].postMessage(JSON.stringify(obj), domain);

}</script>

<!-- localhost:10001 --><script>

window.addEventListener('message', function(event) {

    console.log('Receiver debugging', event);

    if (event.origin == 'http://localhost:10000') {

        var data = JSON.parse(event.data);

        if ('setItem' in data) {

```

```

        localStorage.setItem(data.setItem, data.value);

    } else if ('getItem' in data) {

        var gotItem = localStorage.getItem(data.getItem);

        event.source.postMessage(

            '#localStorage#' + data.identifier +

            (gotItem === null ? 'null#' : '#' + gotItem),

            event.origin

        );

    } else if ('removeItem' in data) {

        localStorage.removeItem(data.removeItem);

    }

}, false);</script>

```

注意 Safari 一下，会报错：

```

Blocked a frame with origin "http://localhost:10001" from
accessing a frame with origin "http://localhost:10000 ".
Protocols, domains, and ports must match.

```

避免该错误，可以在 Safari 浏览器中勾选开发菜单==>停用跨域限制。或者只能使用服务器端转存的方式实现，因为 Safari 浏览器默认只支持 CORS 跨域请求。

方式六：修改 document.domain 跨子域

前提条件：这两个域名必须属于同一个基础域名！而且所用的协议，端口都要一致，否则无法利用 document.domain 进行跨域，所以只能跨子域

在根域范围内，允许把 domain 属性的值设置为它的上一级域。例如，在”aaa.xxx.com”域内，可以把 domain 设置为 “xxx.com” 但不能设置为 “xxx.org” 或者”com”。

现在存在两个域名 aaa.xxx.com 和 bbb.xxx.com。在 aaa 下嵌入 bbb 的页面，

由于其 document.name 不一致，无法在 aaa 下操作 bbb 的 js。

可以在 aaa 和 bbb 下通过 js 将 document.name = 'xxx.com'；

设置一致，来达到互相访问的作用。

方式七：WebSocket

WebSocket protocol 是 HTML5 一种新的协议。它实现了浏览器与服务器全双工通信，同时允许跨域通讯，是 server push 技术的一种很棒的实现。相关文章，请查看：WebSocket、WebSocket-SockJS

需要注意：WebSocket 对象不支持 DOM 2 级事件侦听器，必须使用 DOM 0 级语法分别定义各个事件。

方式八：代理

同源策略是针对浏览器端进行的限制，可以通过服务器端来解决该问题

DomainA 客户端（浏览器） ==> DomainA 服务器 ==> DomainB 服务器 ==> DomainA 客户端（浏览器）

87. 说一下 JSONP 实现原理?

jsonp 即 json+padding, 动态创建 script 标签, 利用 script 标签的 src 属性可以获取任何域下的 js 脚本, 通过这个特性(也可以说漏洞), 服务器端不在返货 json 格式, 而是返回一段调用某个函数的 js 代码, 在 src 中进行了调用, 这样实现了跨域。

设计模式 (九)

88. 说一下你熟悉的设计模式?

单例模式

简单点说, 就是一个应用程序中, 某个类的实例对象只有一个, 你没有办法去 new, 因为构造器是被 private 修饰的, 一般通过 getInstance() 的方法来获取它们的实例。

getInstance() 的返回值是一个对象的引用, 并不是一个新的实例, 所以不要错误的理解成多个对象。单例模式实现起来也很容易, 直接看 demo 吧

```
public class Singleton {  
  
    private static Singleton singleton;  
  
    private Singleton() {  
    }  
  
    public static Singleton getInstance() {  
        if (singleton == null) {  
            singleton = new Singleton();  
        }  
        return singleton;  
    }  
}
```

按照我的习惯, 我恨不得写满注释, 怕你们看不懂, 但是这个代码实在太简单了, 所以我没写任何注释, 如果这几行代码你都看不明白的话, 那你可以洗洗睡了, 等你睡醒了再来看我的博客说不定能看懂。

上面的是最基本的写法, 也叫懒汉写法 (线程不安全) 下面我再公布几种单例模式的写法:

懒汉式写法（线程安全）

```
public class Singleton {  
    private static Singleton instance;  
    private Singleton () {}  
    public static synchronized Singleton getInstance() {  
        if (instance == null) {  
            instance = new Singleton();  
        }  
        return instance;  
    }  
}
```

饿汉式写法

```
public class Singleton {  
    private static Singleton instance = new Singleton();  
    private Singleton () {}  
    public static Singleton getInstance() {  
        return instance;  
    }  
}
```

静态内部类

```
public class Singleton {  
    private static class SingletonHolder {  
        private static final Singleton INSTANCE = new Singleton();  
    }  
    private Singleton () {}  
    public static final Singleton getInstance() {  
        return SingletonHolder.INSTANCE;  
    }  
}
```

枚举

```
public enum Singleton {  
    INSTANCE;  
    public void whateverMethod() {  
    }  
}
```

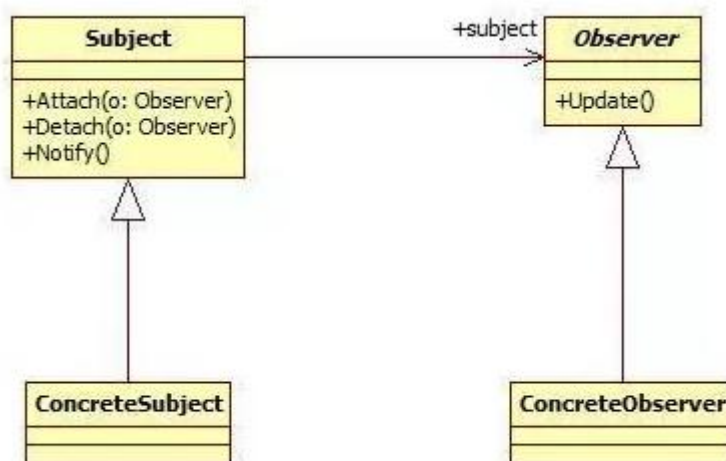
这种方式是 Effective Java 作者 Josh Bloch 提倡的方式，它不仅能避免多线程同步问题，而且还能防止反序列化重新创建新的对象，可谓是很坚强的壁垒啊，不过，个人认为由于 1.5 中才加入 enum 特性，用这种方式写不免让人感觉生疏。

双重校验锁

```
public class Singleton {  
    private volatile static Singleton singleton;  
    private Singleton () {}  
    public static Singleton getSingleton() {  
        if (singleton == null) {  
            synchronized (Singleton.class) {  
                if (singleton == null) {  
                    singleton = new Singleton();  
                }  
            }  
        }  
        return singleton;  
    }  
}
```

观察者模式

对象间一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新。



观察者模式 UML 图

看不懂图的人端着小板凳到这里来，给你举个栗子：假设有三个人，小美（女，22），小王和小李。小美很漂亮，小王和小李是两个程序猿，时刻关注着小美的一举一动。有一天，小美说了一句：“谁来陪我打游戏啊。”这句话被小王和小李听到了，结果乐坏了，蹭蹭蹭，没一会儿，小王就冲到小美家门口了，在这里，小美是被观察者，小王和小李是观察者，被观察者发出一条信息，然后观察者们进行相应的处理，看代码：

```
public interface Person {  
    //小王和小李通过这个接口可以接收到小美发过来的消息  
    void getMessage(String s);  
}
```

这个接口相当于小王和小李的电话号码，小美发送通知的时候就会拨打 `getMessage` 这个电话，拨打电话就是调用接口，看不懂没关系，先往下看

```
public class LaoWang implements Person {  
  
    private String name = "小王";  
  
    public LaoWang() {  
    }  
  
    @Override  
    public void getMessage(String s) {  
        System.out.println(name + "接到了小美打过来的电话，电话内容是：" + s);  
    }  
}  
  
public class LaoLi implements Person {  
  
    private String name = "小李";  
  
    public LaoLi() {  
    }  
  
    @Override  
    public void getMessage(String s) {  
        System.out.println(name + "接到了小美打过来的电话，电话内容是：->" + s);  
    }  
}
```

代码很简单，我们再看看小美的代码：

```

public class XiaoMei {
    List<Person> list = new ArrayList<Person>();
    public XiaoMei() {
    }

    public void addPerson(Person person) {
        list.add(person);
    }

    //遍历 list, 把自己的通知发送给所有暗恋自己的人
    public void notifyPerson() {
        for(Person person:list){
            person.getMessage("你们过来吧, 谁先过来谁就能陪我一起玩儿游
戏!");
        }
    }
}

```

我们写一个测试类来看一下结果对不对

```

public class Test {
    public static void main(String[] args) {

        XiaoMei xiao_mei = new XiaoMei();
        LaoWang lao_wang = new LaoWang();
        LaoLi lao_li = new LaoLi();

        //小王和小李在小美那里都注册了一下
        xiao_mei.addPerson(lao_wang);
        xiao_mei.addPerson(lao_li);

        //小美向小王和小李发送通知
        xiao_mei.notifyPerson();
    }
}

```

装饰者模式

对已有的业务逻辑进一步的封装, 使其增加额外的功能, 如 Java 中的 IO 流就使用了装饰者模式, 用户在使用的时候, 可以任意组装, 达到自己想要的效果。举个栗子, 我想吃三明治, 首先我需要一根大大的香肠, 我喜欢吃奶油, 在香肠上面加一点奶油, 再放一点蔬菜, 最后再用两片面包夹一下, 很丰盛的一顿午饭, 营养又健康。(ps:

不知道上海哪里有卖好吃的三明治的，求推荐～）那我们应该怎么来写代码呢？首先，我们需要写一个 Food 类，让其他所有食物都来继承这个类，看代码：

```
public class Food {  
  
    private String food_name;  
  
    public Food() {  
    }  
  
    public Food(String food_name) {  
        this.food_name = food_name;  
    }  
  
    public String make() {  
        return food_name;  
    };  
}
```

代码很简单，我就不解释了，然后我们写几个子类继承它：

```
//面包类  
public class Bread extends Food {  
  
    private Food basic_food;  
  
    public Bread(Food basic_food) {  
        this.basic_food = basic_food;  
    }  
  
    public String make() {  
        return basic_food.make()+"面包";  
    }  
}
```

```
//奶油类  
public class Cream extends Food {  
  
    private Food basic_food;  
  
    public Cream(Food basic_food) {  
        this.basic_food = basic_food;  
    }  
  
    public String make() {
```

```

        return basic_food.make()+"奶油";
    }
}

//蔬菜类
public class Vegetable extends Food {

    private Food basic_food;

    public Vegetable(Food basic_food) {
        this.basic_food = basic_food;
    }

    public String make() {
        return basic_food.make()+"蔬菜";
    }

}

```

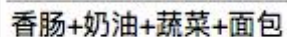
这几个类都是差不多的，构造方法传入一个 Food 类型的参数，然后在 make 方法中加入一些自己的逻辑，如果你还是看不懂为什么这么写，不急，你看看我的 Test 类是怎么写的，一看你就明白了

```

public class Test {
    public static void main(String[] args) {
        Food food = new Bread(new Vegetable(new Cream(new Food("香肠
"))));
        System.out.println(food.make());
    }
}

```

看到没有，一层一层封装，我们从里往外看：最里面我 new 了一个香肠，在香肠的外面我包裹了一层奶油，在奶油的外面我又加了一层蔬菜，最外面我放的是面包，是不是很形象，哈哈~ 这个设计模式简直跟现实生活中一摸一样，看懂了吗？我们看看运行结果吧



运行结果

一个三明治就做好了~

适配器模式

将两种完全不同的事物联系在一起，就像现实生活中的变压器。假设一个手机充电器需要的电压是 20V，但是正常的电压是 220V，这时候就需要一个变压器，将 220V 的电压转换成 20V 的电压，这样，变压器就将 20V 的电压和手机联系起来了。

```
public class Test {
    public static void main(String[] args) {
        Phone phone = new Phone();
        VoltageAdapter adapter = new VoltageAdapter();
        phone.setAdapter(adapter);
        phone.charge();
    }
}

// 手机类
class Phone {

    public static final int V = 220; // 正常电压 220v, 是一个常量

    private VoltageAdapter adapter;

    // 充电
    public void charge() {
        adapter.changeVoltage();
    }

    public void setAdapter(VoltageAdapter adapter) {
        this.adapter = adapter;
    }
}

// 变压器
class VoltageAdapter {
    // 改变电压的功能
    public void changeVoltage() {
        System.out.println("正在充电...");
        System.out.println("原始电压: " + Phone.V + "V");
        System.out.println("经过变压器转换之后的电压: " + (Phone.V - 200)
+ "V");
    }
}
```

工厂模式

简单工厂模式：一个抽象的接口，多个抽象接口的实现类，一个工厂类，用来实例化抽象的接口

```
// 抽象产品类
abstract class Car {
    public void run();

    public void stop();
}

// 具体实现类
class Benz implements Car {
    public void run() {
        System.out.println("Benz 开始启动了。。。。");
    }

    public void stop() {
        System.out.println("Benz 停车了。。。。");
    }
}

class Ford implements Car {
    public void run() {
        System.out.println("Ford 开始启动了。。。");
    }

    public void stop() {
        System.out.println("Ford 停车了。。。。");
    }
}

// 工厂类
class Factory {
    public static Car getCarInstance(String type) {
        Car c = null;
        if ("Benz".equals(type)) {
            c = new Benz();
        }
        if ("Ford".equals(type)) {
            c = new Ford();
        }
    }
}
```



```

        return c;
    }
}

public class Test {

    public static void main(String[] args) {
        Car c = Factory.getCarInstance("Benz");
        if (c != null) {
            c.run();
            c.stop();
        } else {
            System.out.println("造不了这种汽车。。。");
        }
    }

}

```

工厂方法模式：有四个角色，抽象工厂模式，具体工厂模式，抽象产品模式，具体产品模式。不再是由一个工厂类去实例化具体的产品，而是由抽象工厂的子类去实例化产品

```

// 抽象产品角色
public interface Moveable {
    void run();
}

// 具体产品角色
public class Plane implements Moveable {
    @Override
    public void run() {
        System.out.println("plane...");
    }
}

public class Broom implements Moveable {
    @Override
    public void run() {
        System.out.println("broom....");
    }
}

// 抽象工厂
public abstract class VehicleFactory {

```

```

    abstract Moveable create();
}

// 具体工厂
public class PlaneFactory extends VehicleFactory {
    public Moveable create() {
        return new Plane();
    }
}

public class BroomFactory extends VehicleFactory {
    public Moveable create() {
        return new Broom();
    }
}

// 测试类
public class Test {
    public static void main(String[] args) {
        VehicleFactory factory = new BroomFactory();
        Moveable m = factory.create();
        m.run();
    }
}

```

抽象工厂模式：与工厂方法模式不同的是，工厂方法模式中的工厂只生产单一的产品，而抽象工厂模式中的工厂生产多个产品

```

/抽象工厂类
public abstract class AbstractFactory {
    public abstract Vehicle createVehicle();
    public abstract Weapon createWeapon();
    public abstract Food createFood();
}

//具体工厂类，其中Food, Vehicle, Weapon 是抽象类，
public class DefaultFactory extends AbstractFactory{
    @Override
    public Food createFood() {
        return new Apple();
    }
    @Override
    public Vehicle createVehicle() {
        return new Car();
    }
    @Override

```

```

    public Weapon createWeapon() {
        return new AK47();
    }
}

//测试类
public class Test {
    public static void main(String[] args) {
        AbstractFactory f = new DefaultFactory();
        Vehicle v = f.createVehicle();
        v.run();
        Weapon w = f.createWeapon();
        w.shoot();
        Food a = f.createFood();
        a.printName();
    }
}

```

代理模式 (proxy)

有两种，静态代理和动态代理。先说静态代理，很多理论性的东西我不讲，我就算讲了，你们也看不懂。什么真实角色，抽象角色，代理角色，委托角色。。。乱七八糟的，我是看不懂。之前学代理模式的时候，去网上翻一下，资料一大堆，打开链接一看，基本上都是给你分析有什么什么角色，理论一大堆，看起来很费劲，不信的话你们可以去看看，我是看不懂他们在说什么。咱不来虚的，直接用生活中的例子说话。

（注意：我这里并不是否定理论知识，我只是觉得有时候理论知识晦涩难懂，喜欢挑刺的人一边去，你是来学习知识的，不是来挑刺的）

到了一定的年龄，我们就要结婚，结婚是一件很麻烦的事情，（包括那些被父母催婚的）。有钱的家庭可能会找司仪来主持婚礼，显得热闹，洋气～好了，现在婚庆公司的生意来了，我们只需要给钱，婚庆公司就会帮我们安排一整套结婚的流程。整个流程大概是这样的：家里人催婚→男女双方家庭商定结婚的黄道吉日→找一家靠谱的婚庆公司→在约定的时间举行结婚仪式→结婚完毕

婚庆公司打算怎么安排婚礼的节目，在婚礼完毕以后婚庆公司会做什么，我们一概不知。。。别担心，不是黑中介，我们只要把钱给人家，人家会把事情给我们做好。所以，这里的婚庆公司相当于代理角色，现在明白什么是代理角色了吧。

代码实现请看：

```

//代理接口
public interface ProxyInterface {
    //需要代理的是结婚这件事，如果还有其他事情需要代理，比如吃饭睡觉上厕所，也可以写
    void marry();
    //代理吃饭(自己的饭，让别人吃去吧)
    void eat();
}

```

```
//代理拉屎，自己的屎，让别人拉去吧
//void shit();
}
```

文明社会，代理吃饭，代理拉屎什么的我就不写了，有伤社会风化~~~能明白就好

好了，我们看看婚庆公司的代码：

```
public class WeddingCompany implements ProxyInterface {

    private ProxyInterface proxyInterface;

    public WeddingCompany(ProxyInterface proxyInterface) {
        this.proxyInterface = proxyInterface;
    }

    @Override
    public void marry() {
        System.out.println("我们是婚庆公司的");
        System.out.println("我们在做结婚前的准备工作");
        System.out.println("节目彩排...");
        System.out.println("礼物购买...");
        System.out.println("工作人员分工...");
        System.out.println("可以开始结婚了");
        proxyInterface.marry();
        System.out.println("结婚完毕，我们需要做后续处理，你们可以回家了，其余的事情我们公司来做");
    }

}
```

看到没有，婚庆公司需要做的事情很多，我们再看看结婚家庭的代码：

```
public class NormalHome implements ProxyInterface{

    @Override
    public void marry() {
        System.out.println("我们结婚啦~");
    }

}
```

这个已经很明显了，结婚家庭只需要结婚，而婚庆公司要包揽一切，前前后后的事情都是婚庆公司来做，听说现在婚庆公司很赚钱的，这就是原因，干的活多，能不赚钱吗？

来看看测试类代码：

```

public class Test {
    public static void main(String[] args) {
        ProxyInterface proxyInterface = new WeddingCompany(new NormalHome());
        proxyInterface.marry();
    }
}

```

运行结果如下：



```

<terminated> Test (2) [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_102.jdk/Co
我们是婚庆公司的
我们在做结婚前的准备工作
节目彩排...
礼物购买...
工作人员分工...
可以开始结婚了
我们结婚啦~
结婚完毕，我们需要做后续处理，你们可以回家了，其余的事情我们公司来做

```

89. 简单工厂和抽象工厂有什么区别？

简单工厂模式：

这个模式本身很简单而且使用在业务较简单的情况下。一般用于小项目或者具体产品很少扩展的情况（这样工厂类才不用经常更改）。

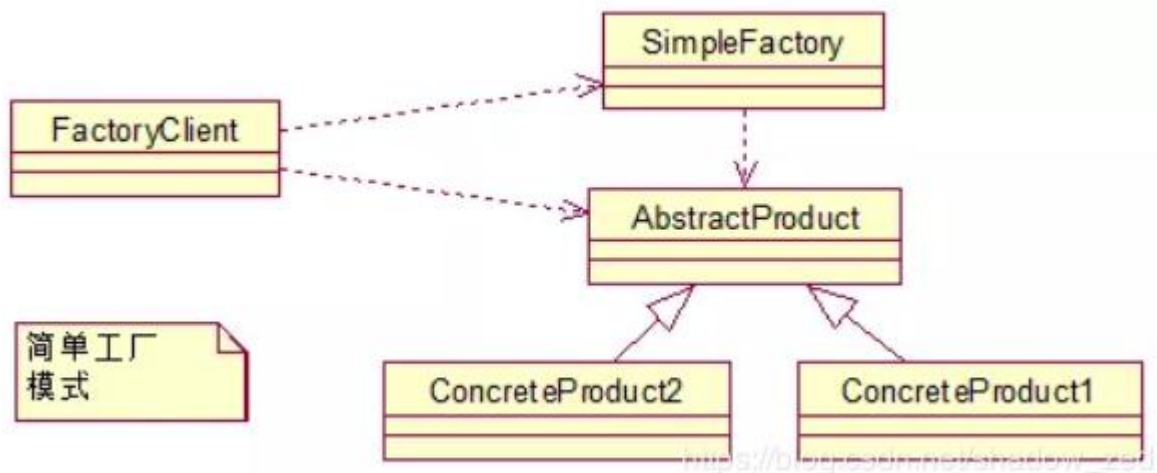
它由三种角色组成：

工厂类角色：这是本模式的核心，含有一定的商业逻辑和判断逻辑，根据逻辑不同，产生具体的工厂产品。如例子中的 Driver 类。

抽象产品角色：它一般是具体产品继承的父类或者实现的接口。由接口或者抽象类来实现。如例中的 Car 接口。

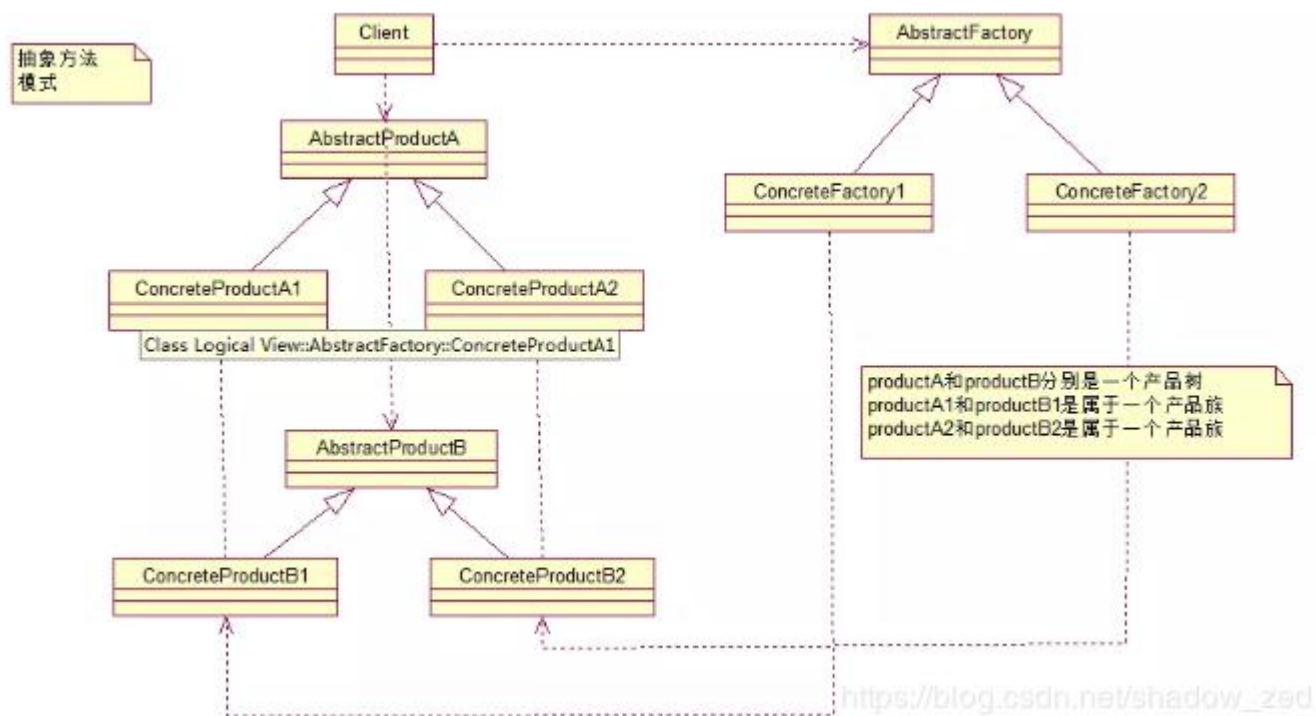
具体产品角色：工厂类所创建的对象就是此角色的实例。在 java 中由一个具体类实现，如例子中的 Benz、Bmw 类。

来用类图来清晰的表示下的它们之间的关系：



抽象工厂模式：

先来认识下什么是产品族： 位于不同产品等级结构中，功能相关联的产品组成的家族。



图中的 BmwCar 和 BenzCar 就是两个产品树（产品层次结构）；而如图所示的 BenzSportsCar 和 BmwSportsCar 就是一个产品族。他们都可以放到跑车家族中，因此功能有所关联。同理 BmwBussinessCar 和 BenzBusinessCar 也是一个产品族。

可以这么说，它和工厂方法模式的区别就在于需要创建对象的复杂程度上。而且抽象工厂模式是三个里面最为抽象、最具一般性的。抽象工厂模式的用意为：给客户端提供一个接口，可以创建多个产品族中的产品对象。

而且使用抽象工厂模式还要满足以下条件：

系统中有多个产品族，而系统一次只可能消费其中一族产品

同属于同一个产品族的产品以其使用。

来看看抽象工厂模式的各个角色（和工厂方法的如出一辙）：

抽象工厂角色：这是工厂方法模式的核心，它与应用程序无关。是具体工厂角色必须实现的接口或者必须继承的父类。在 java 中它由抽象类或者接口来实现。

具体工厂角色：它含有和具体业务逻辑有关的代码。由应用程序调用以创建对应的具体产品的对象。在 java 中它由具体的类来实现。

抽象产品角色：它是具体产品继承的父类或者是实现的接口。在 java 中一般有抽象类或者接口来实现。

具体产品角色：具体工厂角色所创建的对象就是此角色的实例。在 java 中由具体的类来实现。

十、Spring / Spring MVC

90. 为什么要使用 spring?

1. 简介

目的：解决企业应用开发的复杂性

功能：使用基本的 JavaBean 代替 EJB，并提供了更多的企业应用功能

范围：任何 Java 应用

简单来说，Spring 是一个轻量级的控制反转 (IoC) 和面向切面 (AOP) 的容器框架。

2. 轻量

从大小与开销两方面而言 Spring 都是轻量的。完整的 Spring 框架可以在一个大小只有 1MB 多的 JAR 文件里发布。并且 Spring 所需的处理开销也是微不足道的。此外，Spring 是非侵入式的：典型地，Spring 应用中的对象不依赖于 Spring 的特定类。

3. 控制反转

Spring 通过一种称作控制反转（IoC）的技术促进了松耦合。当应用了 IoC，一个对象依赖的其它对象会通过被动的方式传递进来，而不是这个对象自己创建或者查找依赖对象。你可以认为 IoC 与 JNDI 相反——不是对象从容器中查找依赖，而是容器在对象初始化时不等对象请求就主动将依赖传递给它。

4. 面向切面

Spring 提供了面向切面编程的丰富支持，允许通过分离应用的业务逻辑与系统级服务（例如审计（auditing）和事务（transaction）管理）进行内聚性的开发。应用对象只实现它们应该做的——完成业务逻辑——仅此而已。它们并不负责（甚至是意识）其它的系统级关注点，例如日志或事务支持。

5. 容器

Spring 包含并管理应用对象的配置和生命周期，在这个意义上它是一种容器，你可以配置你的每个 bean 如何被创建——基于一个可配置原型（prototype），你的 bean 可以创建一个单独的实例或者每次需要时都生成一个新的实例——以及它们是如何相互关联的。然而，Spring 不应该被混同于传统的重量级的 EJB 容器，它们经常是庞大与笨重的，难以使用。

6. 框架

Spring 可以将简单的组件配置、组合成为复杂的应用。在 Spring 中，应用对象被声明式地组合，典型地是在一个 XML 文件里。Spring 也提供了很多基础功能（事务管理、持久化框架集成等等），将应用逻辑的开发留给了你。

所有 Spring 的这些特征使你能够编写更干净、更可管理、并且更易于测试的代码。它们也为 Spring 中的各种模块提供了基础支持。

91. 解释一下什么是 aop?

AOP (Aspect-Oriented Programming, 面向方面编程), 可以说是 OOP (Object-Oriented Programming, 面向对象编程) 的补充和完善。OOP 引入封装、继承和多态性等概念来建立一种对象层次结构, 用以模拟公共行为的一个集合。当我们需要为分散的对象引入公共行为的时候, OOP 则显得无能为力。也就是说, OOP 允许你定义从上到下的关系, 但并不适合定义从左到右的关系。例如日志功能。日志代码往往水平地散布在所有对象层次中, 而与它所散布到的对象的核心功能毫无关系。对于其他类型的代码, 如安全性、异常处理和透明的持续性也是如此。这种散布在各处的无关的代码被称为横切 (cross-cutting) 代码, 在 OOP 设计中, 它导致了大量代码的重复, 而不利于各个模块的重用。

而 AOP 技术则恰恰相反, 它利用一种称为“横切”的技术, 剖解开封装的对象内部, 并将那些影响了多个类的公共行为封装到一个可重用模块, 并将其名为“Aspect”, 即方面。所谓“方面”, 简单地说, 就是将那些与业务无关, 却为业务模块所共同调用的逻辑或责任封装起来, 便于减少系统的重复代码, 降低模块间的耦合度, 并有利于未来的可操作性和可维护性。AOP 代表的是一个横向的关系, 如果说“对象”是一个空心的圆柱体, 其中封装的是对象的属性和行为; 那么面向方面编程的方法, 就仿佛一把利刃, 将这些空心圆柱体剖开, 以获得其内部的消息。而剖开的切面, 也就是所谓的“方面”了。然后它又以巧夺天工的妙手将这些剖开的切面复原, 不留痕迹。

使用“横切”技术, AOP 把软件系统分为两个部分: 核心关注点和横切关注点。业务处理的主要流程是核心关注点, 与之关系不大的部分是横切关注点。横切关注点的一个特点是, 他们经常发生在核心关注点的多处, 而各处都基本相似。比如权限认证、日志、事务处理。Aop 的作用在于分离系统中的各种关注点, 将核心关注点和横切关注点分离开来。正如 Avanade 公司的高级方案构架师 Adam Magee 所说, AOP 的核心思想就是“将应用程序中的商业逻辑同对其提供支持的通用服务进行分离。”

92. 解释一下什么是 ioc?

IOC 是 Inversion of Control 的缩写, 多数书籍翻译成“控制反转”。

1996 年, Michael Mattson 在一篇有关探讨面向对象框架的文章中, 首先提出了 IOC 这个概念。对于面向对象设计及编程的基本思想, 前面我们已经讲了很多了, 不再赘述, 简单来说就是把复杂系统分解成相互合作的对象, 这些对象类通过封装以后, 内部实现对外部是透明的, 从而降低了解决问题的复杂度, 而且可以灵活地被重用和扩展。

IOC 理论提出的观点大体是这样的: 借助于“第三方”实现具有依赖关系的对象之间的解耦。如下图:

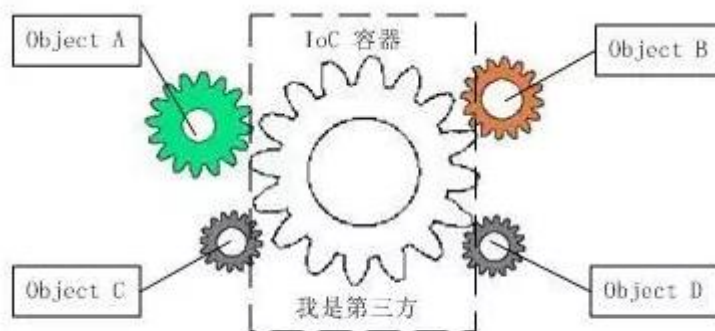


图 IOC解耦过程

图 IOC 解耦过程

大家看到了吧，由于引进了中间位置的“第三方”，也就是 IOC 容器，使得 A、B、C、D 这 4 个对象没有了耦合关系，齿轮之间的传动全部依靠“第三方”了，全部对象的控制权全部上缴给“第三方”IOC 容器，所以，IOC 容器成了整个系统的关键核心，它起到了一种类似“粘合剂”的作用，把系统中的所有对象粘合在一起发挥作用，如果没有这个“粘合剂”，对象与对象之间会彼此失去联系，这就是有人把 IOC 容器比喻成“粘合剂”的由来。

我们再来做个试验：把上图中间的 IOC 容器拿掉，然后再来看看这套系统：

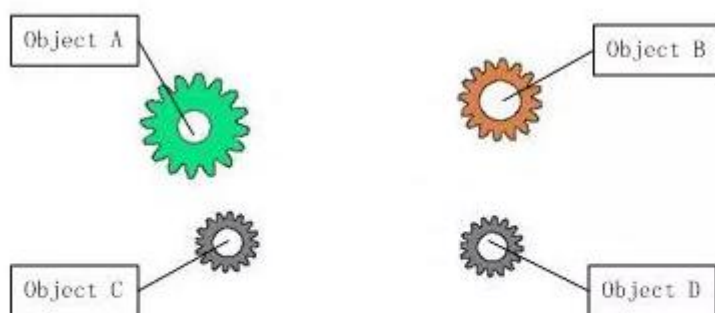


图 拿掉IOC容器后的系统

图 拿掉 IOC 容器后的系统

我们现在看到的画面，就是我们要实现整个系统所需要完成的全部内容。这时候，A、B、C、D 这 4 个对象之间已经没有了耦合关系，彼此毫无联系，这样的话，当你在实现 A 的时候，

根本无须再去考虑 B、C 和 D 了，对象之间的依赖关系已经降低到了最低程度。所以，如果真能实现 IOC 容器，对于系统开发而言，这将是一件多么美好的事情，参与开发的每一成员只要实现自己的类就可以了，跟别人没有任何关系！

我们再来看看，控制反转 (IOC)到底为什么要起这么个名字？我们来对比一下：

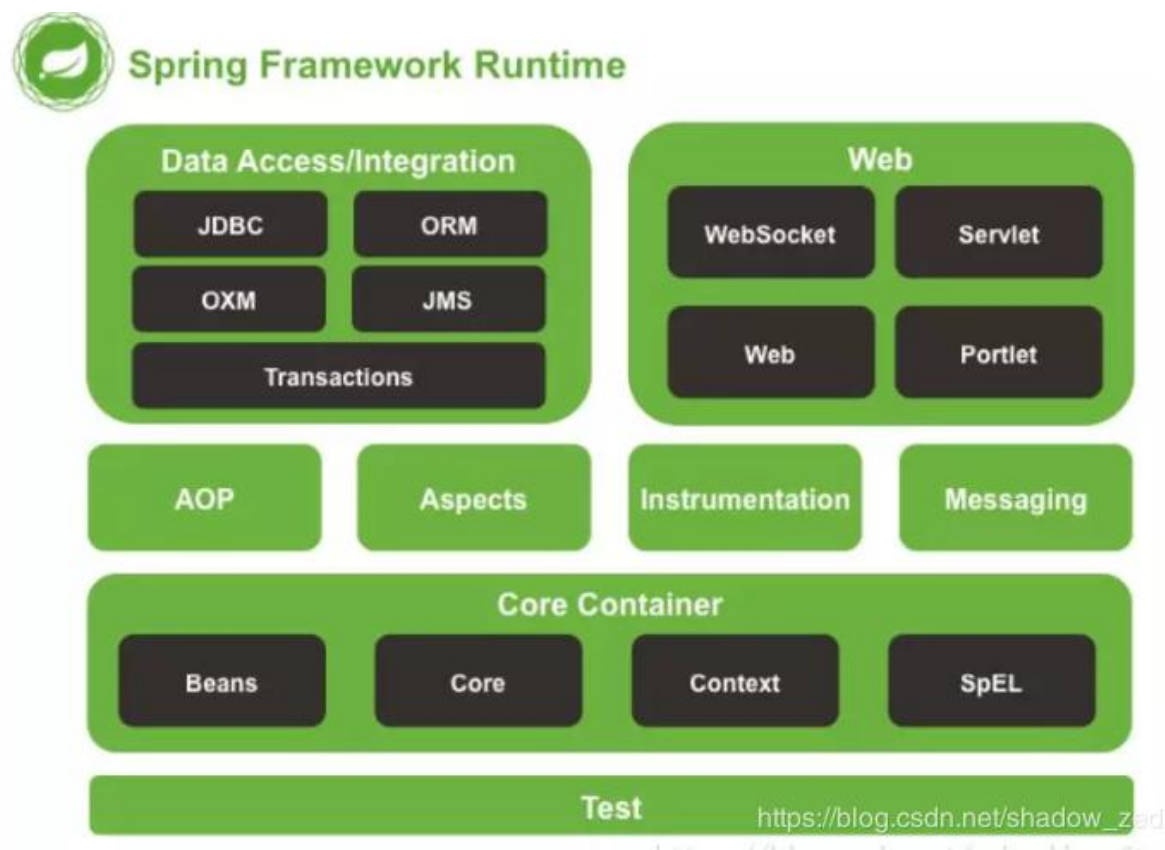
软件系统在没有引入 IOC 容器之前，如图 1 所示，对象 A 依赖于对象 B，那么对象 A 在初始化或者运行到某一点的时候，自己必须主动去创建对象 B 或者使用已经创建的对象 B。无论是创建还是使用对象 B，控制权都在自己手上。

软件系统在引入 IOC 容器之后，这种情形就完全改变了，如图 3 所示，由于 IOC 容器的加入，对象 A 与对象 B 之间失去了直接联系，所以，当对象 A 运行到需要对象 B 的时候，IOC 容器会主动创建一个对象 B 注入到对象 A 需要的地方。

通过前后的对比，我们不难看出来：对象 A 获得依赖对象 B 的过程,由主动行为变为了被动行为，控制权颠倒过来了，这就是“控制反转”这个名称的由来。

93. spring 有哪些主要模块？

Spring 框架至今已集成了 20 多个模块。这些模块主要被分如下图所示的核心容器、数据访问/集成、Web、AOP（面向切面编程）、工具、消息和测试模块。



94. spring 常用的注入方式有哪些？

Spring 通过 DI（依赖注入）实现 IOC（控制反转），常用的注入方式主要有三种：

构造方法注入

setter 注入

基于注解的注入

95. spring 中的 bean 是线程安全的吗？

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。

96. spring 支持几种 bean 的作用域？

当通过 spring 容器创建一个 Bean 实例时，不仅可以完成 Bean 实例的实例化，还可以为 Bean 指定特定的作用域。Spring 支持如下 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：原型模式，每次通过容器的 getBean 方法获取 prototype 定义的 Bean 时，都将产生一个新的 Bean 实例

request：对于每次 HTTP 请求，使用 request 定义的 Bean 都将产生一个新实例，即每次 HTTP 请求将会产生不同的 Bean 实例。只有在 Web 应用中使用 Spring 时，该作用域才有效

session：对于每次 HTTP Session，使用 session 定义的 Bean 都将产生一个新实例。同样只有在 Web 应用中使用 Spring 时，该作用域才有效

globalsession：每个全局的 HTTP Session，使用 session 定义的 Bean 都将产生一个新实例。典型情况下，仅在使用 portlet context 的时候有效。同样只有在 Web 应用中使用 Spring 时，该作用域才有效

其中比较常用的是 singleton 和 prototype 两种作用域。对于 singleton 作用域的 Bean，每次请求该 Bean 都将获得相同的实例。容器负责跟踪 Bean 实例的状态，负责维护 Bean 实例的生命周期行为；如果一个 Bean 被设置成 prototype 作用域，程序每次请求该 id 的 Bean，Spring 都会新建一个 Bean 实例，然后返回给程序。在这种情况下，Spring 容器仅仅使用 new 关键字创建 Bean 实例，一旦创建成功，容器不在跟踪实例，也不会维护 Bean 实例的状态。

如果不指定 Bean 的作用域，Spring 默认使用 singleton 作用域。Java 在创建 Java 实例时，需要进行内存申请；销毁实例时，需要完成垃圾回收，这些工作都会导致系统开销的增加。因此，prototype 作用域 Bean 的创建、销毁代价比较大。而 singleton 作用域的 Bean 实例一旦创建成功，可以重复使用。因此，除非必要，否则尽量避免将 Bean 被设置成 prototype 作用域。

97. spring 自动装配 bean 有哪些方式？

Spring 容器负责创建应用程序中的 bean 同时通过 ID 来协调这些对象之间的关系。作为开发人员，我们需要告诉 Spring 要创建哪些 bean 并且如何将其装配到一起。

spring 中 bean 装配有两种方式：

隐式的 bean 发现机制和自动装配

在 java 代码或者 XML 中进行显示配置

当然这些方式也可以配合使用。

98. spring 事务实现方式有哪些？

编程式事务管理对基于 POJO 的应用来说是唯一选择。我们需要在代码中调用 beginTransaction()、commit()、rollback() 等事务管理相关的方法，这就是编程式事务管理。

基于 TransactionProxyFactoryBean 的声明式事务管理

基于 @Transactional 的声明式事务管理

基于 Aspectj AOP 配置事务

99. 说一下 spring 的事务隔离？

事务隔离级别指的是一个事务对数据的修改与另一个并行的事务的隔离程度，当多个事务同时访问相同数据时，如果没有采取必要的隔离机制，就可能发生以下问题：

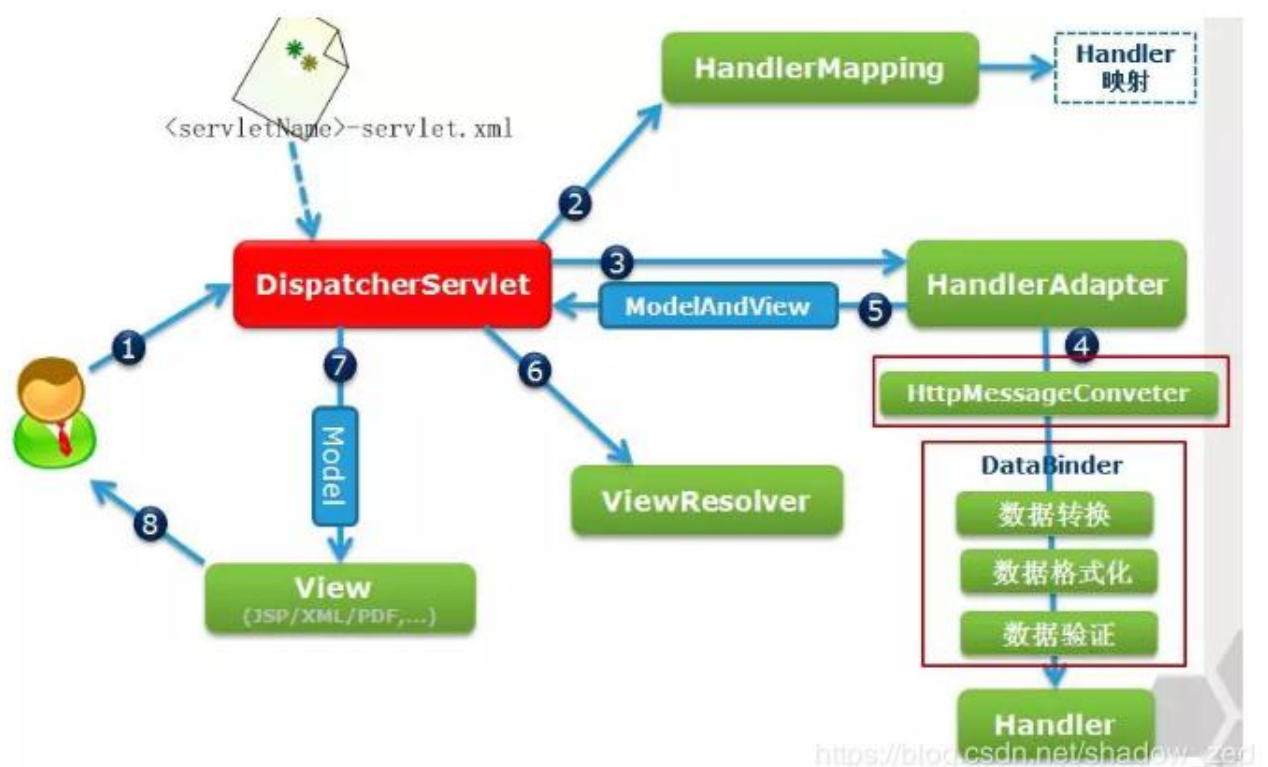
脏读：一个事务读到另一个事务未提交的更新数据。

幻读：例如第一个事务对一个表中的数据进行了修改，比如这种修改涉及到表中的“全部数据行”。同时，第二个事务也修改这个表中的数据，这种修改是向表中插入“一行新数据”。那么，以后就会发生操作第一个事务的用户发现表中还存在没有修改的数据行，就好像发生了幻觉一样。

不可重复读：比方说在同一个事务中先后执行两条一模一样的 select 语句，期间在此次事务中没有执行过任何 DDL 语句，但先后得到的结果不一致，这就是不可重复读。

100. 说一下 spring mvc 运行流程？

Spring MVC 运行流程图：



Spring 运行流程描述:

1. 用户向服务器发送请求，请求被 Spring 前端控制 Servlet DispatcherServlet 捕获；
2. DispatcherServlet 对请求 URL 进行解析，得到请求资源标识符（URI）。然后根据该 URI，调用 HandlerMapping 获得该 Handler 配置的所有相关的对象（包括 Handler 对象以及 Handler 对象对应的拦截器），最后以 HandlerExecutionChain 对象的形式返回；
3. DispatcherServlet 根据获得的 Handler，选择一个合适的 HandlerAdapter；（附注：如果成功获得 HandlerAdapter 后，此时将开始执行拦截器的 preHandler(...)方法）
4. 提取 Request 中的模型数据，填充 Handler 入参，开始执行 Handler（Controller）。在填充 Handler 的入参过程中，根据你的配置，Spring 将帮你做一些额外的工作：

HttpMessageConveter： 将请求消息（如 Json、xml 等数据）转换成一个对象，将对象转换为指定的响应信息

数据转换：对请求消息进行数据转换。如 String 转换成 Integer、Double 等

数据根式化：对请求消息进行数据格式化。 如将字符串转换成格式化数字或格式化日期等

数据验证： 验证数据的有效性（长度、格式等），验证结果存储到 BindingResult 或 Error 中

5. Handler 执行完成后，向 DispatcherServlet 返回一个 ModelAndView 对象；
6. 根据返回的 ModelAndView，选择一个适合的 ViewResolver（必须是已经注册到 Spring 容器中的 ViewResolver）返回给 DispatcherServlet ；
7. ViewResolver 结合 Model 和 View，来渲染视图；
8. 将渲染结果返回给客户端。

101. spring mvc 有哪些组件？

Spring MVC 的核心组件：

DispatcherServlet：中央控制器，把请求转发到具体的控制类

Controller：具体处理请求的控制器

HandlerMapping: 映射处理器, 负责映射中央处理器转发给 controller 时的映射策略

ModelAndView: 服务层返回的数据和视图层的封装类

ViewResolver: 视图解析器, 解析具体的视图

Interceptors : 拦截器, 负责拦截我们定义请求然后做处理工作

102. @RequestMapping 的作用是什么?

RequestMapping 是一个用来处理请求地址映射的注解, 可用于类或方法上。用于类上, 表示类中的所有响应请求的方法都是以该地址作为父路径。

RequestMapping 注解有六个属性, 下面我们把她分成三类进行说明。

value, method:

value: 指定请求的实际地址, 指定的地址可以是 URI Template 模式 (后面将会说明);

method: 指定请求的 method 类型, GET、POST、PUT、DELETE 等;

consumes, produces

consumes: 指定处理请求的提交内容类型 (Content-Type), 例如 application/json, text/html;

produces: 指定返回的内容类型, 仅当 request 请求头中的 (Accept) 类型中包含该指定类型才返回;

params, headers

params: 指定 request 中必须包含某些参数值是, 才让该方法处理。

headers: 指定 request 中必须包含某些指定的 header 值, 才能让该方法处理请求。

103. @Autowired 的作用是什么?

《@Autowired 用法详解》: 首先要知道另一个东西, default-autowire, 它是在 xml 文件中进行配置的, 可以设置为 byName、byType、constructor 和 autodetect; 比如 byName,

不用显式的在 bean 中写出依赖的对象,它会自动的匹配其它 bean 中 id 名与本 bean 的 set** 相同的,并自动装载。

@Autowired 是用在 JavaBean 中的注解,通过 byType 形式,用来给指定的字段或方法注入所需的外部资源。

两者的功能是一样的,就是能减少或者消除属性或构造器参数的设置,只是配置地方不一样而已。

autowire 四种模式的区别:

先看一下 bean 实例化和@Autowired 装配过程:

一切都是从 bean 工厂的 getBean 方法开始的,一旦该方法调用总会返回一个 bean 实例,无论当前是否存在,不存在就实例化一个并装配,否则直接返回。(Spring MVC 是在什么时候开始执行 bean 的实例化过程的呢?其实就在组件扫描完成之后)

实例化和装配过程中会多次递归调用 getBean 方法来解决类之间的依赖。

Spring 几乎考虑了所有可能性,所以方法特别复杂但完整有条理。

@Autowired 最终是根据类型来查找和装配元素的,但是我们设置了<beans

default-autowire="byName"/>后会影响最终的类型匹配查找。因为在前面有根据

BeanDefinition 的 autowire 类型设置 PropertyValue 值得一步,其中会有新实例的创建和注册。就是那个 autowireByName 方法。

Spring Boot / Spring Cloud (十一)

104. 什么是 spring boot?

在 Spring 框架这个大家族中,产生了很多衍生框架,比如 Spring、SpringMvc 框架等, Spring 的核心内容在于控制反转 (IOC) 和依赖注入 (DI), 所谓控制反转并非是一种技术, 而是一种思想, 在操作方面是指在 spring 配置文件中创建, 依赖注入即为由 spring 容器为应用程序的某个对象提供资源, 比如 引用对象、常量数据等。

SpringBoot 是一个框架, 一种全新的编程规范, 他的产生简化了框架的使用, 所谓简化是指简化了 Spring 众多框架中所需的大量且繁琐的配置文件, 所以 SpringBoot 是一个服务于框架的框架, 服务范围是简化配置文件。

105. 为什么要用 spring boot?

Spring Boot 使编码变简单

Spring Boot 使配置变简单

Spring Boot 使部署变简单

Spring Boot 使监控变简单

Spring 的不足

106. spring boot 核心配置文件是什么?

Spring Boot 提供了两种常用的配置文件:

- properties 文件
- yaml 文件
- pom 文件

107. spring boot 配置文件有哪几种类型? 它们有什么区别?

Spring Boot 提供了两种常用的配置文件, 分别是 properties 文件和 yaml 文件。相对于 properties 文件而言, yaml 文件更年轻, 也有很多的坑。可谓成也萧何败萧何, yaml 通过空格来确定层级关系, 使配置文件结构跟清晰, 但也会因为微不足道的空格而破坏了层级关系。

108. spring boot 有哪些方式可以实现热部署?

SpringBoot 热部署实现有两种方式:

①. 使用 spring loaded

在项目中添加如下代码:

```
<build>

    <plugins>

        <plugin>

            <!-- springBoot 编译插件-->

            <groupId>org.springframework.boot</groupId>

            <artifactId>spring-boot-maven-plugin</artifactId>

            <dependencies>

                <!-- spring 热部署 -->

                <!-- 该依赖在此处下载不下来, 可以放置在 build 标签外部下载完
成后再粘贴进 plugin 中 -->

                <dependency>

                    <groupId>org.springframework</groupId>

                    <artifactId>springloaded</artifactId>

                    <version>1.2.6.RELEASE</version>

                </dependency>

            </dependencies>

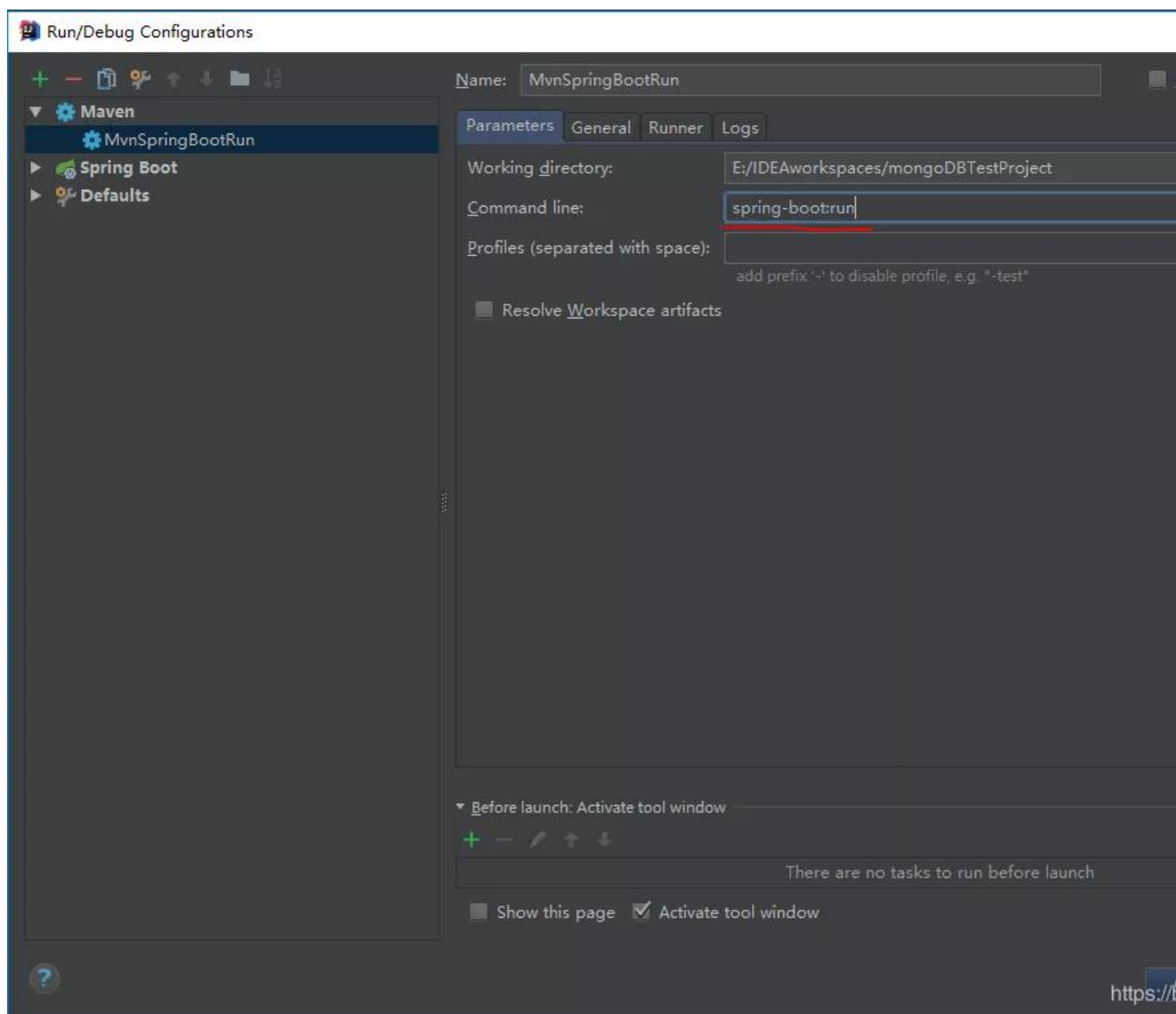
        </plugin>
```

```
</plugins>

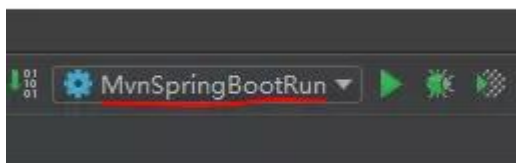
</build>
```

添加完毕后需要使用 mvn 指令运行：

首先找到 IDEA 中的 Edit configurations , 然后进行如下操作：（点击左上角的“+”，然后选择 maven 将出现右侧面板，在红色划线部位输入如图所示指令，你可以为该指令命名（此处命名为 MvnSpringBootRun））



点击保存将会在 IDEA 项目运行部位出现，点击绿色箭头运行即可



②. 使用 spring-boot-devtools

在项目的 pom 文件中添加依赖：

```
<!--热部署 jar-->

<dependency>

    <groupId>org.springframework.boot</groupId>

    <artifactId>spring-boot-devtools</artifactId>

</dependency>
```

然后：使用 shift+ctrl+alt+“/”（IDEA 中的快捷键）选择“Registry” 然后勾选 compiler. automake. allow. when. app. running

109. jpa 和 hibernate 有什么区别？

- JPA Java Persistence API，是 Java EE 5 的标准 ORM 接口，也是 ejb3 规范的一部分。
- Hibernate，当今很流行的 ORM 框架，是 JPA 的一个实现，但是其功能是 JPA 的超集。
- JPA 和 Hibernate 之间的关系，可以简单的理解为 JPA 是标准接口，Hibernate 是实现。那么 Hibernate 是如何实现与 JPA 的这种关系的呢。Hibernate 主要是通过三个组件来实现的，及 hibernate-annotation、hibernate-entitymanager 和 hibernate-core。
- hibernate-annotation 是 Hibernate 支持 annotation 方式配置的基础，它包括了标准的 JPA annotation 以及 Hibernate 自身特殊功能的 annotation。
- hibernate-core 是 Hibernate 的核心实现，提供了 Hibernate 所有的核心功能。
- hibernate-entitymanager 实现了标准的 JPA，可以把它看成 hibernate-core 和 JPA 之间的适配器，它并不直接提供 ORM 的功能，而是对 hibernate-core 进行封装，使得 Hibernate 符合 JPA 的规范。

110. 什么是 spring cloud?

从字面理解, Spring Cloud 就是致力于分布式系统、云服务的框架。

Spring Cloud 是整个 Spring 家族中新的成员, 是最近云服务火爆的必然产物。

Spring Cloud 为开发人员提供了快速构建分布式系统中一些常见模式的工具, 例如:

- 配置管理
- 服务注册与发现
- 断路器
- 智能路由
- 服务间调用
- 负载均衡
- 微代理
- 控制总线
- 一次性令牌
- 全局锁
- 领导选举
- 分布式会话
- 集群状态
- 分布式消息
-

使用 Spring Cloud 开发人员可以开箱即用的实现这些模式的服务和应用程序。这些服务可以任何环境下运行, 包括分布式环境, 也包括开发人员自己的笔记本电脑以及各种托管平台。

111. spring cloud 断路器的作用是什么?

在 Spring Cloud 中使用了 Hystrix 来实现断路器的功能, 断路器可以防止一个应用程序多次试图执行一个操作, 即很可能失败, 允许它继续而不等待故障恢复或者浪费 CPU 周期, 而它确定该故障是持久的。断路器模式也使应用程序能够检测故障是否已经解决, 如果问题似乎已经得到纠正, 应用程序可以尝试调用操作。

断路器增加了稳定性和灵活性, 以一个系统, 提供稳定性, 而系统从故障中恢复, 并尽量减少此故障的对性能的影响。它可以帮助快速地拒绝对一个操作, 即很可能失败, 而不是等待操作超时 (或者不返回) 的请求, 以保持系统的响应时间。如果断路器提高每次改变状态的时间的事件, 该信息可以被用来监测由断路器保护系统的部件的健康状况, 或以提醒管理员当断路器跳闸, 以在打开状态。

112. spring cloud 的核心组件有哪些？

①. 服务发现——Netflix Eureka

一个 RESTful 服务，用来定位运行在 AWS 地区 (Region) 中的中间层服务。由两个组件组成：Eureka 服务器和 Eureka 客户端。Eureka 服务器用作服务注册服务器。Eureka 客户端是一个 java 客户端，用来简化与服务器的交互、作为轮询负载均衡器，并提供服务的故障切换支持。Netflix 在其生产环境中使用的是另外的客户端，它提供基于流量、资源利用率以及出错状态的加权负载均衡。

②. 客户端负载均衡——Netflix Ribbon

Ribbon，主要提供客户侧的软件负载均衡算法。Ribbon 客户端组件提供一系列完善的配置选项，比如连接超时、重试、重试算法等。Ribbon 内置可插拔、可定制的负载均衡组件。

③. 断路器——Netflix Hystrix

断路器可以防止一个应用程序多次试图执行一个操作，即很可能失败，允许它继续而不等待故障恢复或者浪费 CPU 周期，而它确定该故障是持久的。断路器模式也使应用程序能够检测故障是否已经解决。如果问题似乎已经得到纠正，应用程序可以尝试调用操作。

④. 服务网关——Netflix Zuul

类似 nginx，反向代理的功能，不过 netflix 自己增加了一些配合其他组件的特性。

⑤. 分布式配置——Spring Cloud Config

这个还是静态的，得配合 Spring Cloud Bus 实现动态的配置更新。

Hibernate（十二）

113. 为什么要使用 hibernate？

对 JDBC 访问数据库的代码做了封装，大大简化了数据访问层繁琐的重复性代码。

Hibernate 是一个基于 JDBC 的主流持久化框架，是一个优秀的 ORM 实现。他很大程度的简化 DAO 层的编码工作

hibernate 使用 Java 反射机制，而不是字节码增强程序来实现透明性。

hibernate 的性能非常好，因为它是个轻量级框架。映射的灵活性很出色。它支持各种关系数据库，从一对一到多对多的各种复杂关系。

114. 什么是 ORM 框架？

对象-关系映射（Object-Relational Mapping，简称 ORM），面向对象的开发方法是当今企业级应用开发环境中的主流开发方法，关系数据库是企业级应用环境中永久存放数据的主流数据存储系统。对象和关系数据是业务实体的两种表现形式，业务实体在内存中表现为对象，在数据库中表现为关系数据。内存中的对象之间存在关联和继承关系，而在数据库中，关系数据无法直接表达多对多关联和继承关系。因此，对象-关系映射 (ORM) 系统一般以中间件的形式存在，主要实现程序对象到关系数据库数据的映射。

115. hibernate 中如何在控制台查看打印的 sql 语句？

打印 sql 语句到控制台

首先，我使用的是 application.properties 配置文件，使用 yml 也可以达到同样的效果。

在网上查这个问题查了好久，基本上都是 xml 配置，在此不多说；

正确的 properties 配置项应该如下图所示：

在 jpa 下一级不直接是 hibernate，而是 properties。

```
spring.jpa.properties.hibernate.show_sql=true           //控制台是否打印
```

```
spring.jpa.properties.hibernate.format_sql=true         //格式化 sql 语句
```

```
spring.jpa.properties.hibernate.use_sql_comments=true   //指出是什么操作生成了该  
语句
```

打印 sql 语句中的参数值

经过上面的步骤，我们已经可以在控制台打印出格式化之后的 sql 语句，但是大多数情况下，我们还需要具体的 sql 参数值，这个时候我们就需要配置 日志配置文件。

博主使用的是 slf4j 的日志，配置文件用的是 logback.xml，配置方式如下：

```
<logger name="org.hibernate.SQL" level="DEBUG"/>
```

```
<logger name="org.hibernate.engine.QueryParameters" level="DEBUG"/>
```

```
<logger name="org.hibernate.engine.query.HQLQueryPlan" level="DEBUG"/>
```


打印 sql 语句到日志

在上述步骤的基础上，在 logback.xml 中增加两项配置：

```
<logger name="org.hibernate.type.descriptor.sql.BasicBinder" level="TRACE"/>
```

```
<logger name="org.hibernate.type.descriptor.sql.BasicExtractor" level="TRACE"/>
```

116. hibernate 有几种查询方式？

- hql 查询
- sql 查询
- 条件查询

hql 查询，sql 查询，条件查询

HQL: Hibernate Query Language. 面向对象的写法：

```
Query query = session.createQuery("from Customer where name = ?");
```

```
query.setParameter(0, "苍老师");
```

```
Query.list();
```

QBC: Query By Criteria. (条件查询)

```
Criteria criteria = session.createCriteria(Customer.class);
```

```
criteria.add(Restrictions.eq("name", "花姐"));
```

```
List<Customer> list = criteria.list();
```

SQL:

```
SQLQuery query = session.createSQLQuery("select * from customer");
```

```
List<Object[]> list = query.list();
```

```
SQLQuery query = session.createSQLQuery("select * from customer");  
  
query.addEntity(Customer.class);  
  
List<Customer> list = query.list();
```

Hql: 具体分类 1、 属性查询 2、 参数查询、命名参数查询

3、 关联查询 4、 分页查询 5、 统计函数

HQL 和 SQL 的区别

HQL 是面向对象查询操作的, SQL 是结构化查询语言 是面向数据库表结构的

117. hibernate 实体类可以被定义为 final 吗?

可以将 Hibernate 的实体类定义为 final 类,但这种做法并不好。因为 Hibernate 会使用代理模式在延迟关联的情况下提高性能,如果你把实体类定义成 final 类之后,因为 Java 不允许对 final 类进行扩展,所以 Hibernate 就无法再使用代理了,如此一来就限制了使用可以提升性能的手段。不过,如果你的持久化类实现了一个接口而且在该接口中声明了所有定义于实体类中的所有 public 的方法轮到话,你就能够避免出现前面所说的不利后果。

118. 在 hibernate 中使用 Integer 和 int 做映射有什么区别?

在 Hibernate 中,如果将 OID 定义为 Integer 类型,那么 Hibernate 就可以根据其值是否为 null 而判断一个对象是否是临时的,如果将 OID 定义为了 int 类型,还需要在 hbm 映射文件中设置其 unsaved-value 属性为 0。

119. hibernate 是如何工作的？

hibernate 工作原理：

1. 通过Configuration config = new Configuration().configure();//读取并解析hibernate.cfg.xml配置文件
2. 由hibernate.cfg.xml中的<mapping resource="com/xx/User.hbm.xml"/>读取并解析映射信息
3. 通过SessionFactory sf = config.buildSessionFactory();//创建SessionFactory
4. Session session = sf.openSession();//打开Session
5. Transaction tx = session.beginTransaction();//创建并启动事务Transaction
6. persistent operate操作数据，持久化操作
7. tx.commit();//提交事务
8. 关闭Session
9. 关闭SessionFactory

<https://blog.csdn.net/li1325169021>

120. get()和 load()的区别？

- load() 没有使用对象的其他属性的时候，没有 SQL 延迟加载
- get() 没有使用对象的其他属性的时候，也生成了 SQL 立即加载

121. 说一下 hibernate 的缓存机制？

Hibernate 中的缓存分为一级缓存和二级缓存。

一级缓存就是 Session 级别的缓存，在事务范围内有效，内置的不能被卸载。二级缓存是 SessionFactory 级别的缓存，从应用启动到应用结束有效。是可选的，默认没有二级缓存，需要手动开启。保存数据库后，缓存在内存中保存一份，如果更新了数据库就要同步更新。

什么样的数据适合存放到第二级缓存中？

- 很少被修改的数据 帖子的最后回复时间
- 经常被查询的数据 电商的地点
- 不是很重要的数据，允许出现偶尔并发的数据
- 不会被并发访问的数据
- 常量数据

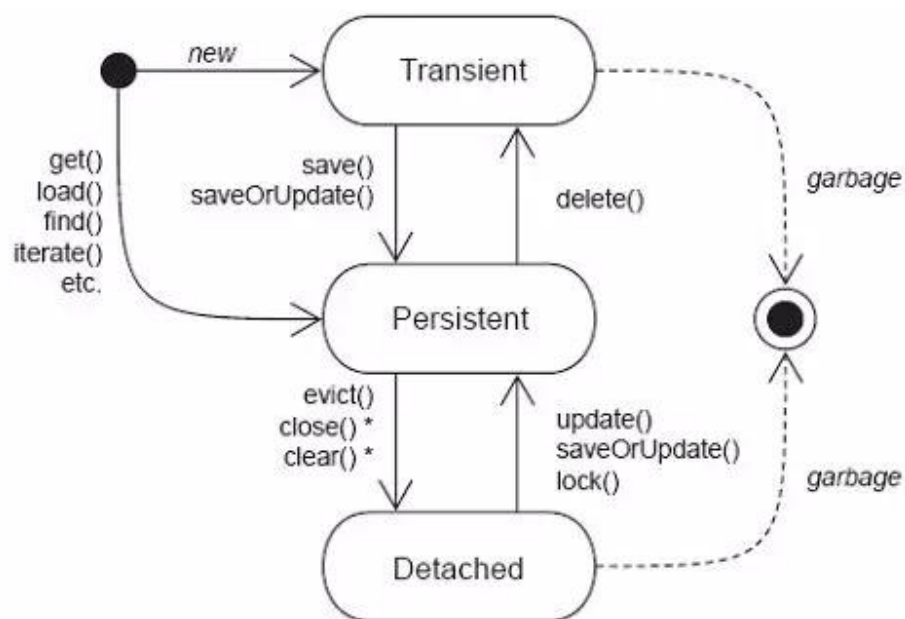
扩展：hibernate 的二级缓存默认是不支持分布式缓存的。使用 memcache, redis 等中央缓存来代替二级缓存。

122. hibernate 对象有哪些状态?

hibernate 里对象有三种状态:

1. Transient (瞬时): 对象刚 new 出来, 还没设 id, 设了其他值。
2. Persistent (持久): 调用了 save()、saveOrUpdate(), 就变成 Persistent, 有 id。
3. Detached (脱管): 当 session close() 完之后, 变成 Detached。

对象状态—完整版



<https://blog.csdn.net/li1325169021>

123. 在 hibernate 中 getCurrentSession 和 openSession 的区别是什么?

`openSession` 从字面上可以看得出来, 是打开一个新的 session 对象, 而且每次使用都是打开一个新的 session, 假如连续使用多次, 则获得的 session 不是同一个对象, 并且使用完需要调用 `close` 方法关闭 session。

`getCurrentSession`, 从字面上可以看得出来, 是获取当前上下文一个 session 对象, 当第一次使用此方法时, 会自动产生一个 session 对象, 并且连续使用多次时, 得到的 session 都是同一个对象, 这就是与 `openSession` 的区别之一, 简单而言, `getCurrentSession` 就是: 如果有已经使用的, 用旧的, 如果没有, 建新的。

注意：在实际开发中，往往使用 `getCurrentSession` 多，因为一般是处理同一个事务（即使用一个数据库的情况），所以在一般情况下比较少使用 `openSession` 或者说 `openSession` 是比较老旧的一套接口了。

124、hibernate 实体类必须要有无参构造函数吗？为什么？

必须，因为 hibernate 框架会调用这个默认构造方法来构造实例对象，即 `Class` 类的 `newInstance` 方法，这个方法就是通过调用默认构造方法来创建实例对象的。

另外再提醒一点，如果你没有提供任何构造方法，虚拟机会自动提供默认构造方法（无参构造器），但是如果你提供了其他有参数的构造方法的话，虚拟机就不再为你提供默认构造方法，这时必须手动把无参构造器写在代码里，否则 `new Xxx()` 是会报错的，所以默认的构造方法不是必须的，只有在有多个构造方法时才是必须的，这里“必须”指的是“必须手动写出来”。

Mybatis（十三）

125、mybatis 中 `#{}` 和 `${}` 的区别是什么？

`#{}` 是预编译处理,KaTeX parse error: Expected 'EOF', got '#' at position 21: ...串替换； Mybatis在处理`# {}` 时，会将sql中的`#{}` 替`…{}` 时，就是把`${}` 替换成变量的值；使用`#{}` 可以有效的防止 SQL 注入，提高系统安全性。

126、mybatis 有几种分页方式？

数组分页

sql 分页

拦截器分页

RowBounds 分页

127、RowBounds 是一次性查询全部结果吗？为什么？

RowBounds 表面是在“所有”数据中检索数据，其实并非是一次性查询出所有数据，因为 MyBatis 是对 jdbc 的封装，在 jdbc 驱动中有一个 `Fetch Size` 的配置，它规定了每次最多从数据库查询多少条数据，假如你要查询更多数据，它会在你执行 `next()` 的时候，去查询更多的数据。就好比你去自动取款机取 10000 元，但取款机每次最多能取 2500 元，所以你要取 4 次才能把钱取完。只是对于 jdbc 来说，当你调用 `next()` 的时候会自动帮你完成查询工作。这样做的好处可以有效的防止内存溢出。

128、mybatis 逻辑分页和物理分页的区别是什么？

物理分页速度上并不一定快于逻辑分页，逻辑分页速度上也并不一定快于物理分页。

物理分页总是优于逻辑分页：没有必要将属于数据库端的压力加诸到应用端来，就算速度上存在优势，然而其它性能上的优点是足以弥补这个缺点。

129、mybatis 是否支持延迟加载？延迟加载的原理是什么？

Mybatis 仅支持 association 关联对象和 collection 关联集合对象的延迟加载，association 指的就是一对一，collection 指的就是一对多查询。在 Mybatis 配置文件中，可以配置是否启用延迟加载 lazyLoadingEnabled=true|false。

它的原理是，使用 CGLIB 创建目标对象的代理对象，当调用目标方法时，进入拦截器方法，比如调用 a.getB().getName()，拦截器 invoke() 方法发现 a.getB() 是 null 值，那么就会单独发送事先保存好的查询关联 B 对象的 sql，把 B 查询上来，然后调用 a.setB(b)，于是 a 的对象 b 属性就有值了，接着完成 a.getB().getName() 方法的调用。这就是延迟加载的基本原理。

当然了，不光是 Mybatis，几乎所有的包括 Hibernate，支持延迟加载的原理都是一样的。

130、说一下 mybatis 的一级缓存和二级缓存？

一级缓存：基于 PerpetualCache 的 HashMap 本地缓存，其存储作用域为 Session，当 Session flush 或 close 之后，该 Session 中的所有 Cache 就将清空，默认打开一级缓存。

二级缓存与一级缓存其机制相同，默认也是采用 PerpetualCache，HashMap 存储，不同在于其存储作用域为 Mapper(Namespace)，并且可自定义存储源，如 Ehcache。默认不打开二级缓存，要开启二级缓存，使用二级缓存属性类需要实现 Serializable 序列化接口(可用来保存对象的状态)，可在它的映射文件中配置；

对于缓存数据更新机制，当某一个作用域(一级缓存 Session/二级缓存 Namespaces)的进行 C/U/D 操作后，默认该作用域下所有 select 中的缓存将被 clear。

131、mybatis 和 hibernate 的区别有哪些？

(1) Mybatis 和 hibernate 不同，它不完全是一个 ORM 框架，因为 MyBatis 需要程序员自己编写 Sql 语句。

(2) Mybatis 直接编写原生态 sql，可以严格控制 sql 执行性能，灵活度高，非常适合对关系数据模型要求不高的软件开发，因为这类软件需求变化频繁，一旦需求变化要求迅速输出成果。但是灵活的前提是 mybatis 无法做到数据库无关性，如果实现支持多种数据库的软件，则需要自定义多套 sql 映射文件，工作量大。

(3) Hibernate 对象/关系映射能力强，数据库无关性好，对于关系模型要求高的软件，如果用 hibernate 开发可以节省很多代码，提高效率。

132、mybatis 有哪些执行器 (Executor) ?

Mybatis 有三种基本的执行器 (Executor) :

SimpleExecutor: 每执行一次 update 或 select, 就开启一个 Statement 对象, 用完立刻关闭 Statement 对象。

ReuseExecutor: 执行 update 或 select, 以 sql 作为 key 查找 Statement 对象, 存在就使用, 不存在就创建, 用完后, 不关闭 Statement 对象, 而是放置于 Map 内, 供下一次使用。简言之, 就是重复使用 Statement 对象。

BatchExecutor: 执行 update (没有 select, JDBC 批处理不支持 select), 将所有 sql 都添加到批处理中 (addBatch()), 等待统一执行 (executeBatch()), 它缓存了多个 Statement 对象, 每个 Statement 对象都是 addBatch() 完毕后, 等待逐一执行 executeBatch() 批处理。与 JDBC 批处理相同。

133、mybatis 分页插件的实现原理是什么?

分页插件的基本原理是使用 Mybatis 提供的插件接口, 实现自定义插件, 在插件的拦截方法内拦截待执行的 sql, 然后重写 sql, 根据 dialect 方言, 添加对应的物理分页语句和物理分页参数。

134、mybatis 如何编写一个自定义插件?

Mybatis 自定义插件针对 Mybatis 四大对象 (Executor、StatementHandler、ParameterHandler、ResultSetHandler) 进行拦截, 具体拦截方式为:

Executor: 拦截执行器的方法 (log 记录)

StatementHandler : 拦截 Sql 语法构建的处理

ParameterHandler : 拦截参数的处理

ResultSetHandler : 拦截结果集的处理

Mybatis 自定义插件必须实现 Interceptor 接口:

```
public interface Interceptor {  
  
    Object intercept(Invocation invocation) throws Throwable;  
  
    Object plugin(Object target);  
  
    void setProperties(Properties properties);}
```

intercept 方法: 拦截器具体处理逻辑方法

plugin 方法: 根据签名 signatureMap 生成动态代理对象

setProperties 方法：设置 Properties 属性

自定义插件 demo：

```
// ExamplePlugin.java@Intercepts({@Signature(  
  
    type= Executor.class,  
  
    method = "update",  
  
    args = {MappedStatement.class, Object.class}})})public class ExamplePlugin  
implements Interceptor {  
  
    public Object intercept(Invocation invocation) throws Throwable {  
  
        Object target = invocation.getTarget(); //被代理对象  
  
        Method method = invocation.getMethod(); //代理方法  
  
        Object[] args = invocation.getArgs(); //方法参数  
  
        // do something ..... 方法拦截前执行代码块  
  
        Object result = invocation.proceed();  
  
        // do something ..... 方法拦截后执行代码块  
  
        return result;  
  
    }  
  
    public Object plugin(Object target) {  
  
        return Plugin.wrap(target, this);  
  
    }  
  
    public void setProperties(Properties properties) {  
  
    }  
}
```


一个@Intercepts 可以配置多个@Signature, @Signature 中的参数定义如下:

type: 表示拦截的类, 这里是 Executor 的实现类;

method: 表示拦截的方法, 这里是拦截 Executor 的 update 方法;

args: 表示方法参数。

RabbitMQ (十四)

135、rabbitmq 的使用场景有哪些?

- ①. 跨系统的异步通信, 所有需要异步交互的地方都可以使用消息队列。就像我们除了打电话 (同步) 以外, 还需要发短信, 发电子邮件 (异步) 的通讯方式。
- ②. 多个应用之间的耦合, 由于消息是平台无关和语言无关的, 而且语义上也不再是函数调用, 因此更适合作为多个应用之间的松耦合的接口。基于消息队列的耦合, 不需要发送方和接收方同时在线。在企业应用集成 (EAI) 中, 文件传输, 共享数据库, 消息队列, 远程过程调用都可以作为集成的方法。
- ③. 应用内的同步变异步, 比如订单处理, 就可以由前端应用将订单信息放到队列, 后端应用从队列里依次获得消息处理, 高峰时的大量订单可以积压在队列里慢慢处理掉。由于同步通常意味着阻塞, 而大量线程的阻塞会降低计算机的性能。
- ④. 消息驱动的架构 (EDA), 系统分解为消息队列, 和消息制造者和消息消费者, 一个处理流程可以根据需要拆成多个阶段 (Stage), 阶段之间用队列连接起来, 前一个阶段处理的结果放入队列, 后一个阶段从队列中获取消息继续处理。
- ⑤. 应用需要更灵活的耦合方式, 如发布订阅, 比如可以指定路由规则。
- ⑥. 跨局域网, 甚至跨城市的通讯 (CDN 行业), 比如北京机房与广州机房的应用程序的通信。

136、rabbitmq 有哪些重要的角色?

RabbitMQ 中重要的角色有: 生产者、消费者和代理:

生产者: 消息的创建者, 负责创建和推送数据到消息服务器;

消费者: 消息的接收方, 用于处理数据和确认消息;

代理: 就是 RabbitMQ 本身, 用于扮演 “快递” 的角色, 本身不生产消息, 只是扮演 “快递” 的角色。

137、rabbitmq 有哪些重要的组件？

ConnectionFactory（连接管理器）：应用程序与 Rabbit 之间建立连接的管理器，程序代码中使用。

Channel（信道）：消息推送使用的通道。

Exchange（交换器）：用于接受、分配消息。

Queue（队列）：用于存储生产者的消息。

RoutingKey（路由键）：用于把生成者的数据分配到交换器上。

BindingKey（绑定键）：用于把交换器的消息绑定到队列上。

138、rabbitmq 中 vhost 的作用是什么？

vhost 可以理解为虚拟 broker，即 mini-RabbitMQ server。其内部均含有独立的 queue、exchange 和 binding 等，但最最重要的是，其拥有独立的权限系统，可以做到 vhost 范围的用户控制。当然，从 RabbitMQ 的全局角度，vhost 可以作为不同权限隔离的手段（一个典型的例子就是不同的应用可以跑在不同的 vhost 中）。

139、rabbitmq 的消息是怎么发送的？

首先客户端必须连接到 RabbitMQ 服务器才能发布和消费消息，客户端和 rabbit server 之间会创建一个 tcp 连接，一旦 tcp 打开并通过了认证（认证就是你发送给 rabbit 服务器的用户名和密码），你的客户端和 RabbitMQ 就创建了一条 amqp 信道（channel），信道是创建在“真实” tcp 上的虚拟连接，amqp 命令都是通过信道发送出去的，每个信道都会有一个唯一的 id，不论是发布消息，订阅队列都是通过这个信道完成的。

140、rabbitmq 怎么保证消息的稳定性？

提供了事务的功能。

通过将 channel 设置为 confirm（确认）模式。

141、rabbitmq 怎么避免消息丢失？

消息持久化

ACK 确认机制

设置集群镜像模式

消息补偿机制

142、要保证消息持久化成功的条件有哪些？

声明队列必须设置持久化 durable 设置为 true。

消息推送投递模式必须设置持久化，deliveryMode 设置为 2（持久）。

消息已经到达持久化交换器。

消息已经到达持久化队列。

以上四个条件都满足才能保证消息持久化成功。

143、rabbitmq 持久化有什么缺点？

持久化的缺点就是降低了服务器的吞吐量，因为使用的是磁盘而非内存存储，从而降低了吞吐量。可尽量使用 ssd 硬盘来缓解吞吐量的问题。

144、rabbitmq 有几种广播类型？

三种广播模式：

fanout：所有 bind 到此 exchange 的 queue 都可以接收消息（纯广播，绑定到 RabbitMQ 的接受者都能收到消息）；

direct：通过 routingKey 和 exchange 决定的那个唯一的 queue 可以接收消息；

topic：所有符合 routingKey（此时可以是一个表达式）的 routingKey 所 bind 的 queue 可以接收消息；

145、rabbitmq 怎么实现延迟消息队列？

通过消息过期后进入死信交换器，再由交换器转发到延迟消费队列，实现延迟功能；

使用 RabbitMQ-delayed-message-exchange 插件实现延迟功能。

146、rabbitmq 集群有什么用？

集群主要有以下两个用途：

高可用：某个服务器出现问题，整个 RabbitMQ 还可以继续使用；

高容量：集群可以承载更多的消息量。

147、rabbitmq 节点的类型有哪些？

磁盘节点：消息会存储到磁盘。

内存节点：消息都存储在内存中，重启服务器消息丢失，性能高于磁盘类型。

148、rabbitmq 集群搭建需要注意哪些问题？

各节点之间使用“-link”连接，此属性不能忽略。

各节点使用的 erlang cookie 值必须相同，此值相当于“密钥”的功能，用于各节点的认证。

整个集群中必须包含一个磁盘节点。

149、rabbitmq 每个节点是其他节点的完整拷贝吗？为什么？

不是，原因有以下两个：

存储空间的考虑：如果每个节点都拥有所有队列的完全拷贝，这样新增节点不但没有新增存储空间，反而增加了更多的冗余数据；

性能的考虑：如果每条消息都需要完整拷贝到每一个集群节点，那新增节点并没有提升处理消息的能力，最多是保持和单节点相同的性能甚至是更糟。

150、rabbitmq 集群中唯一一个磁盘节点崩溃了会发生什么情况？

如果唯一磁盘的磁盘节点崩溃了，不能进行以下操作：

不能创建队列

不能创建交换器

不能创建绑定

不能添加用户

不能更改权限

不能添加和删除集群节点

唯一磁盘节点崩溃了，集群是可以保持运行的，但你不能更改任何东西。

151、rabbitmq 对集群节点停止顺序有要求吗？

RabbitMQ 对集群的停止的顺序是有要求的，应该先关闭内存节点，最后再关闭磁盘节点。

如果顺序恰好相反的话，可能会造成消息的丢失。

Kafka（十五）

152、kafka 可以脱离 zookeeper 单独使用吗？为什么？

kafka 不能脱离 zookeeper 单独使用，因为 kafka 使用 zookeeper 管理和协调 kafka 的节点服务器。

153. kafka 有几种数据保留的策略？

kafka 有两种数据保存策略：按照过期时间保留和按照存储的消息大小保留。

154. kafka 同时设置了 7 天和 10G 清除数据，到第五天的时候消息达到了 10G，这个时候 kafka 将如何处理？

这个时候 kafka 会执行数据清除工作，时间和大小不论那个满足条件，都会清空数据。

155. 什么情况会导致 kafka 运行变慢？

cpu 性能瓶颈

磁盘读写瓶颈

网络瓶颈

156. 使用 kafka 集群需要注意什么？

集群的数量不是越多越好，最好不要超过 7 个，因为节点越多，消息复制需要的时间就越长，整个群组的吞吐量就越低。

集群数量最好是单数，因为超过一半故障集群就不能用了，设置为单数容错率更高。

Zookeeper（十六）

157、zookeeper 是什么？

zookeeper 是一个分布式的，开放源码的分布式应用程序协调服务，是 google chubby 的开源实现，是 hadoop 和 hbase 的重要组件。它是一个为分布式应用提供一致性服务的软件，提供的功能包括：配置维护、域名服务、分布式同步、组服务等。

158. zookeeper 都有哪些功能？

集群管理：监控节点存活状态、运行请求等。

主节点选举：主节点挂掉了之后可以从备用的节点开始新一轮选主，主节点选举说的就是这个选举的过程，使用 zookeeper 可以协助完成这个过程。

分布式锁：zookeeper 提供两种锁：独占锁、共享锁。独占锁即一次只能有一个线程使用资源，共享锁是读锁共享，读写互斥，即可以有多线程同时读同一个资源，如果要使用写锁也只能有一个线程使用。zookeeper 可以对分布式锁进行控制。

命名服务：在分布式系统中，通过使用命名服务，客户端应用能够根据指定名字来获取资源或服务的地址，提供者等信息。

159. zookeeper 有几种部署模式？

zookeeper 有三种部署模式：

单机部署：一台集群上运行；

集群部署：多台集群运行；

伪集群部署：一台集群启动多个 zookeeper 实例运行。

160. zookeeper 怎么保证主从节点的状态同步？

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，分别是恢复模式（选主）和广播模式（同步）。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 完成了和 leader 的状态同步以后，恢复模式就结束了。状态同步保证了 leader 和 server 具有相同的系统状态。

161. 集群中为什么要有主节点？

在分布式环境中，有些业务逻辑只需要集群中的某一台机器进行执行，其他的机器可以共享这个结果，这样可以大大减少重复计算，提高性能，所以需要主节点。

162. 集群中有 3 台服务器，其中一个节点宕机，这个时候 zookeeper 还可以使用吗？

可以继续使用，单数服务器只要没超过一半的服务器宕机就可以继续使用。

163. 说一下 zookeeper 的通知机制？

客户端会对某个 znode 建立一个 watcher 事件，当该 znode 发生变化时，这些客户端会收到 zookeeper 的通知，然后客户端可以根据 znode 变化来做出业务上的改变。

MySql（十七）

164、数据库的三范式是什么？

第一范式：强调的是列的原子性，即数据库表的每一列都是不可分割的原子数据项。

第二范式：要求实体的属性完全依赖于主关键字。所谓完全依赖是指不能存在仅依赖主关键字一部分的属性。

第三范式：任何非主属性不依赖于其它非主属性。

165. 一张自增表里面总共有 7 条数据，删除了最后 2 条数据，重启 mysql 数据库，又插入了一条数据，此时 id 是几？

表类型如果是 MyISAM，那 id 就是 8。

表类型如果是 InnoDB，那 id 就是 6。

InnoDB 表只会把自增主键的最大 id 记录在内存中，所以重启之后会导致最大 id 丢失。

166. 如何获取当前数据库版本？

使用 `select version()` 获取当前 MySQL 数据库版本。

167. 说一下 ACID 是什么？

Atomicity（原子性）：一个事务（transaction）中的所有操作，或者全部完成，或者全部不完成，不会结束在中间某个环节。事务在执行过程中发生错误，会被恢复（Rollback）到事务开始前的状态，就像这个事务从来没有执行过一样。即，事务不可分割、不可约简。

Consistency（一致性）：在事务开始之前和事务结束以后，数据库的完整性没有被破坏。这表示写入的资料必须完全符合所有的预设约束、触发器、级联回滚等。

Isolation（隔离性）：数据库允许多个并发事务同时对其数据进行读写和修改的能力，隔离性可以防止多个事务并发执行时由于交叉执行而导致数据的不一致。事务隔离分为不同级别，包括读未提交（Read uncommitted）、读提交（read committed）、可重复读（repeatable read）和串行化（Serializable）。

Durability（持久性）：事务处理结束后，对数据的修改就是永久的，即便系统故障也不会丢失。

168. char 和 varchar 的区别是什么？

char(n)：固定长度类型，比如订阅 char(10)，当你输入“abc”三个字符的时候，它们占的空间还是 10 个字节，其他 7 个是空字节。

char 优点：效率高；缺点：占用空间；适用场景：存储密码的 md5 值，固定长度的，使用 char 非常合适。

varchar(n)：可变长度，存储的值是每个值占用的字节再加上一个用来记录其长度的字节的长度。

所以，从空间上考虑 varchar 比较合适；从效率上考虑 char 比较合适，二者使用需要权衡。

169. float 和 double 的区别是什么？

float 最多可以存储 8 位的十进制数，并在内存中占 4 字节。

double 最可以存储 16 位的十进制数，并在内存中占 8 字节。

170. mysql 的内连接、左连接、右连接有什么区别？

内连接关键字：inner join；左连接：left join；右连接：right join。

内连接是把匹配的关联数据显示出来；左连接是左边的表全部显示出来，右边的表显示出符合条件的数据；右连接正好相反。

171. mysql 索引是怎么实现的？

索引是满足某种特定查找算法的数据结构，而这些数据结构会以某种方式指向数据，从而实现高效查找数据。

具体来说 MySQL 中的索引，不同的数据引擎实现有所不同，但目前主流的数据库引擎的索引都是 B+ 树实现的，B+ 树的搜索效率，可以到达二分法的性能，找到数据区域之后就找到了完整的数据结构了，所有索引的性能也是更好的。

172. 怎么验证 mysql 的索引是否满足需求？

使用 explain 查看 SQL 是如何执行查询语句的，从而分析你的索引是否满足需求。

explain 语法：explain select * from table where type=1。

173. 说一下数据库的事务隔离？

MySQL 的事务隔离是在 MySQL. ini 配置文件里添加的，在文件的最后添加：

transaction-isolation = REPEATABLE-READ

可用的配置值：READ-UNCOMMITTED、READ-COMMITTED、REPEATABLE-READ、SERIALIZABLE。

READ-UNCOMMITTED：未提交读，最低隔离级别、事务未提交前，就可被其他事务读取（会出现幻读、脏读、不可重复读）。

READ-COMMITTED：提交读，一个事务提交后才能被其他事务读取到（会造成幻读、不可重复读）。

REPEATABLE-READ：可重复读，默认级别，保证多次读取同一个数据时，其值都和事务开始时候的内容是一致的，禁止读取到别的事务未提交的数据（会造成幻读）。

SERIALIZABLE：序列化，代价最高最可靠的隔离级别，该隔离级别能防止脏读、不可重复读、幻读。

脏读：表示一个事务能够读取另一个事务中还未提交的数据。比如，某个事务尝试插入记录 A，此时该事务还未提交，然后另一个事务尝试读取到了记录 A。

不可重复读：是指在一个事务内，多次读同一数据。

幻读：指同一个事务内多次查询返回的结果集不一样。比如同一个事务 A 第一次查询时候有 n 条记录，但是第二次同等条件下查询却有 n+1 条记录，这就好像产生了幻觉。发生幻读的原因也是另外一个事务新增或者删除或者修改了第一个事务结果集里面的数据，同一个记录的数据内容被修改了，所有数据行的记录就变多或者变少了。

174. 说一下 mysql 常用的引擎？

InnoDB 引擎：InnoDB 引擎提供了对数据库 acid 事务的支持，并且还提供了行级锁和外键的约束，它的设计的目标就是处理大数据容量的数据库系统。MySQL 运行的时候，InnoDB 会在内存中建立缓冲池，用于缓冲数据和索引。但是该引擎是不支持全文搜索，同时启动也比较的慢，它是不会保存表的行数的，所以当进行 `select count(*) from table` 指令的时候，需要进行扫描全表。由于锁的粒度小，写操作是不会锁定全表的，所以在并发度较高的场景下使用会提升效率的。

MyIASM 引擎：MySQL 的默认引擎，但不提供事务的支持，也不支持行级锁和外键。因此当执行插入和更新语句时，即执行写操作的时候需要锁定这个表，所以会导致效率会降低。不过和 InnoDB 不同的是，MyIASM 引擎是保存了表的行数，于是当进行 `select count(*) from table` 语句时，可以直接的读取已经保存的值而不需要进行扫描全表。所以，如果表的读操作远远多于写操作时，并且不需要事务的支持的，可以将 MyIASM 作为数据库引擎的首选。

175. 说一下 mysql 的行锁和表锁？

MyISAM 只支持表锁，InnoDB 支持表锁和行锁，默认为行锁。

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，并发量最低。

行级锁：开销大，加锁慢，会出现死锁。锁力度小，发生锁冲突的概率小，并发度最高。

176. 说一下乐观锁和悲观锁？

乐观锁：每次去拿数据的时候都认为别人不会修改，所以不会上锁，但是在提交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。

悲观锁：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止，直到这个锁被释放。

数据库的乐观锁需要自己实现，在表里面添加一个 `version` 字段，每次修改成功值加 1，这样每次修改的时候先对比一下，自己拥有的 `version` 和数据库现在的 `version` 是否一致，如果不一致就不修改，这样就实现了乐观锁。

177. mysql 问题排查都有哪些手段？

使用 `show processlist` 命令查看当前所有连接信息。

使用 `explain` 命令查询 SQL 语句执行计划。

开启慢查询日志，查看慢查询的 SQL。

178. 如何做 mysql 的性能优化？

为搜索字段创建索引。

避免使用 `select *`，列出需要查询的字段。

垂直分割分表。

选择正确的存储引擎。

Redis（十八）

179、redis 是什么？都有哪些使用场景？

Redis 是一个开源的使用 ANSI C 语言编写、支持网络、可基于内存亦可持久化的日志型、Key-Value 数据库，并提供多种语言的 API。

Redis 使用场景：

数据高并发的读写

海量数据的读写

对扩展性要求高的数据

180. redis 有哪些功能？

数据缓存功能

分布式锁的功能

支持数据持久化

支持事务

支持消息队列

181. redis 和 memecache 有什么区别？

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

redis 的速度比 memcached 快很多

redis 可以持久化其数据

182. redis 为什么是单线程的？

因为 cpu 不是 Redis 的瓶颈，Redis 的瓶颈最有可能是机器内存或者网络带宽。既然单线程容易实现，而且 cpu 又不会成为瓶颈，那就顺理成章地采用单线程的方案了。

关于 Redis 的性能，官方网站也有，普通笔记本轻松处理每秒几十万的请求。

而且单线程并不代表就慢 nginx 和 nodejs 也都是高性能单线程的代表。

183. 什么是缓存穿透？怎么解决？

缓存穿透：指查询一个一定不存在的数据，由于缓存是不命中时需要从数据库查询，查不到数据则不写入缓存，这将导致这个不存在的数据每次请求都要到数据库去查询，造成缓存穿透。

解决方案：最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），我们就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。

184. redis 支持的数据类型有哪些？

string、list、hash、set、zset。

185. redis 支持的 java 客户端都有哪些？

Redisson、Jedis、lettuce 等等，官方推荐使用 Redisson。

186. jedis 和 redisson 有哪些区别？

Jedis 是 Redis 的 Java 实现的客户端，其 API 提供了比较全面的 Redis 命令的支持。

Redisson 实现了分布式和可扩展的 Java 数据结构，和 Jedis 相比，功能较为简单，不支持字符串操作，不支持排序、事务、管道、分区等 Redis 特性。Redisson 的宗旨是促进使用者对 Redis 的关注分离，从而让使用者能够将精力更集中地放在处理业务逻辑上。

187. 怎么保证缓存和数据库数据的一致性？

合理设置缓存的过期时间。

新增、更改、删除数据库操作时同步更新 Redis，可以使用事物机制来保证数据的一致性。

188. redis 持久化有几种方式？

Redis 的持久化有两种方式，或者说有两种策略：

RDB (Redis Database)：指定的时间间隔能对你的数据进行快照存储。

AOF (Append Only File)：每一个收到的写命令都通过 write 函数追加到文件中。

189. redis 怎么实现分布式锁？

Redis 分布式锁其实就是在系统里面占一个“坑”，其他程序也要占“坑”的时候，占用成功了就可以继续执行，失败了就只能放弃或稍后重试。

占坑一般使用 setnx(set if not exists)指令，只允许被一个程序占有，使用完调用 del 释放锁。

190. redis 分布式锁有什么缺陷？

Redis 分布式锁不能解决超时的问题，分布式锁有一个超时时间，程序的执行如果超出了锁的超时时间就会出现问题。

191. redis 如何做内存优化？

尽可能使用散列表（hashes），散列表（是说散列表里面存储的数少）使用的内存非常小，所以你应该尽可能的将你的数据模型抽象到一个散列表里面。

比如你的 web 系统中有一个用户对象，不要为这个用户的名称，姓氏，邮箱，密码设置单独的 key，而是应该把这个用户的所有信息存储到一张散列表里面。

192. redis 淘汰策略有哪些？

volatile-lru：从已设置过期时间的数据集（server.db[i].expires）中挑选最近最少使用的数据淘汰。

volatile-ttl：从已设置过期时间的数据集（server.db[i].expires）中挑选将要过期的数据淘汰。

volatile-random：从已设置过期时间的数据集（server.db[i].expires）中任意选择数据淘汰。

allkeys-lru：从数据集（server.db[i].dict）中挑选最近最少使用的数据淘汰。

allkeys-random：从数据集（server.db[i].dict）中任意选择数据淘汰。

no-eviction（驱逐）：禁止驱逐数据。

193. redis 常见的性能问题有哪些？该如何解决？

主服务器写内存快照，会阻塞主线程的工作，当快照比较大时对性能影响是非常大的，会间断性暂停服务，所以主服务器最好不要写内存快照。

Redis 主从复制的性能问题，为了主从复制的速度和连接的稳定性，主从库最好在同一个局域网内。

JVM（十九）

194、说一下 jvm 的主要组成部分？及其作用？

类加载器（ClassLoader）

运行时数据区（Runtime Data Area）

执行引擎（Execution Engine）

本地库接口（Native Interface）

组件的作用：首先通过类加载器（ClassLoader）会把 Java 代码转换成字节码，运行时数据区（Runtime Data Area）再把字节码加载到内存中，而字节码文件只是 JVM 的一套指令集规范，并不能直接交给底层操作系统去执行，因此需要特定的命令解析器执行引擎（Execution Engine），将字节码翻译成底层系统指令，再交由 CPU 去执行，而这个过程需要调用其他语言的本地库接口（Native Interface）来实现整个程序的功能。

195. 说一下 jvm 运行时数据区？

程序计数器

虚拟机栈

本地方法栈

堆

方法区

有的区域随着虚拟机进程的启动而存在，有的区域则依赖用户进程的启动和结束而创建和销毁。

196、说一下堆栈的区别？

栈内存存储的是局部变量而堆内存存储的是实体；

栈内存的更新速度要快于堆内存，因为局部变量的生命周期很短；

栈内存存放的变量生命周期一旦结束就会被释放，而堆内存存放的实体会被垃圾回收机制不定时的回收。

197、队列和栈是什么？有什么区别？

队列和栈都是被用来预存储数据的。

队列允许先进先出检索元素，但也有例外的情况，Deque 接口允许从两端检索元素。

栈和队列很相似，但它运行对元素进行后进先出进行检索。

198. 什么是双亲委派模型？

在介绍双亲委派模型之前先说下类加载器。对于任意一个类，都需要由加载它的类加载器和这个类本身一同确立在 JVM 中的唯一性，每一个类加载器，都有一个独立的类名称空间。

类加载器就是根据指定全限定名称将 class 文件加载到 JVM 内存，然后再转化为 class 对象。

类加载器分类：

启动类加载器(Bootstrap ClassLoader)，是虚拟机自身的一部分，用来加载 Java_HOME/lib/ 目录中的，或者被 -Xbootclasspath 参数所指定的路径中并且被虚拟机识别的类库；

其他类加载器：

扩展类加载器 (Extension ClassLoader)：负责加载<java_home style= “box-sizing: border-box; -webkit-tap-highlight-color: transparent; text-size-adjust: none; -webkit-font-smoothing: antialiased; outline: 0px !important;” >\lib\ext 目录或 Java. ext. dirs 系统变量指定的路径中的所有类库；</java_home>

应用程序类加载器 (Application ClassLoader)。负责加载用户类路径 (classpath) 上的指定类库，我们可以直接使用这个类加载器。一般情况，如果我们没有自定义类加载器默认就是用这个加载器。

双亲委派模型：如果一个类加载器收到了类加载的请求，它首先不会自己去加载这个类，而是把这个请求委派给父类加载器去完成，每一层的类加载器都是如此，这样所有的加载请求都会被传送到顶层的启动类加载器中，只有当父加载无法完成加载请求（它的搜索范围中没找到所需的类）时，子加载器才会尝试去加载类。

199. 说一下类加载的执行过程？

类加载分为以下 5 个步骤：

加载：根据查找路径找到相应的 class 文件然后导入；

检查：检查加载的 class 文件的正确性；

准备：给类中的静态变量分配内存空间；

解析：虚拟机将常量池中的符号引用替换成直接引用的过程。符号引用就理解为一个标示，而在直接引用直接指向内存中的地址；

初始化：对静态变量和静态代码块执行初始化工作。

200. 怎么判断对象是否可以被回收？

一般有两种方法来判断：

引用计数器：为每个对象创建一个引用计数，有对象引用时计数器 +1，引用被释放时计数 -1，当计数器为 0 时就可以被回收。它有一个缺点不能解决循环引用的问题；

可达性分析：从 GC Roots 开始向下搜索，搜索所走过的路径称为引用链。当一个对象到 GC Roots 没有任何引用链相连时，则证明此对象是可以被回收的。

201. java 中都有哪些引用类型？

强引用

软引用

弱引用

虚引用（幽灵引用/幻影引用）

202. 说一下 jvm 有哪些垃圾回收算法？

标记-清除算法

标记-整理算法

复制算法

分代算法

203. 说一下 jvm 有哪些垃圾回收器？

Serial：最早的单线程串行垃圾回收器。

Serial Old：Serial 垃圾回收器的老年版本，同样也是单线程的，可以作为 CMS 垃圾回收器的备选预案。

ParNew：是 Serial 的多线程版本。

Parallel 和 ParNew 收集器类似是多线程的，但 Parallel 是吞吐量优先的收集器，可以牺牲等待时间换取系统的吞吐量。

Parallel Old 是 Parallel 老年代版本，Parallel 使用的是复制的内存回收算法，

Parallel Old 使用的是标记-整理的内存回收算法。

CMS：一种以获得最短停顿时间为目标的收集器，非常适用 B/S 系统。

G1：一种兼顾吞吐量和停顿时间的 GC 实现，是 JDK 9 以后的默认 GC 选项。

204. 详细介绍一下 CMS 垃圾回收器？

CMS 是英文 Concurrent Mark-Sweep 的简称，是以牺牲吞吐量为代价来获得最短回收停顿时间的垃圾回收器。对于要求服务器响应速度的应用上，这种垃圾回收器非常适合。在启动 JVM 的参数加上 “-XX:+UseConcMarkSweepGC” 来指定使用 CMS 垃圾回收器。

CMS 使用的是标记-清除的算法实现的，所以在 gc 的时候会回产生大量的内存碎片，当剩余内存不能满足程序运行要求时，系统将会出现 Concurrent Mode Failure，临时 CMS 会采用 Serial Old 回收器进行垃圾清除，此时的性能将会被降低。

205. 新生代垃圾回收器和老生代垃圾回收器都有哪些？有什么区别？

新生代回收器：Serial、ParNew、Parallel Scavenge

老年代回收器：Serial Old、Parallel Old、CMS

整堆回收器：G1

新生代垃圾回收器一般采用的是复制算法，复制算法的优点是效率高，缺点是内存利用率低；老年代回收器一般采用的是标记-整理的算法进行垃圾回收。

206. 简述分代垃圾回收器是怎么工作的？

分代回收器有两个分区：老生代和新生代，新生代默认的空间占比总空间的 1/3，老生代的默认占比是 2/3。

新生代使用的是复制算法，新生代里有 3 个分区：Eden、To Survivor、From Survivor，它们的默认占比是 8:1:1，它的执行流程如下：

把 Eden + From Survivor 存活的对象放入 To Survivor 区；

清空 Eden 和 From Survivor 分区；

From Survivor 和 To Survivor 分区交换，From Survivor 变 To Survivor，To Survivor 变 From Survivor。

每次在 From Survivor 到 To Survivor 移动时都存活的对象，年龄就 +1，当年龄到达 15（默认配置是 15）时，升级为老生代。大对象也会直接进入老生代。

老生代当空间占用到达某个值之后就会触发全局垃圾回收，一般使用标记整理的执行算法。以上这些循环往复就构成了整个分代垃圾回收的整体执行流程。

207. 说一下 jvm 调优的工具？

JDK 自带了很多监控工具，都位于 JDK 的 bin 目录下，其中最常用的是 jconsole 和 jvisualvm 这两款视图监控工具。

jconsole：用于对 JVM 中的内存、线程和类等进行监控；

jvisualvm：JDK 自带的全能分析工具，可以分析：内存快照、线程快照、程序死锁、监控内存的变化、gc 变化等。

208. 常用的 jvm 调优的参数都有哪些？

-Xms2g：初始化堆大小为 2g；

-Xmx2g：堆最大内存为 2g；

-XX:NewRatio=4：设置年轻的和老年代的内存比例为 1:4；

-XX:SurvivorRatio=8：设置新生代 Eden 和 Survivor 比例为 8:2；

-XX:+UseParNewGC：指定使用 ParNew + Serial Old 垃圾回收器组合；

-XX:+UseParallelOldGC：指定使用 ParNew + ParNew Old 垃圾回收器组合；

-XX:+UseConcMarkSweepGC：指定使用 CMS + Serial Old 垃圾回收器组合；

-XX:+PrintGC：开启打印 gc 信息；

-XX:+PrintGCDetails：打印 gc 详细信息。