学习题目

1、 **JDK 和 JRE 有什么区别?**
2、 **== 和 equals 的区别是什么?**
3、 **两个对象的 hashCode()相同，则 equals()也一定为 true，对吗?**
6、String 属于基础的数据类型吗?
7、**java 中操作字符串都有哪些类? 它们之间有什么区别?**
8、 **String str="i"与 String str=new String("i")一样吗?**
9、如何将字符串翻转
10、 **String 类的常用方法都有那些?**
11、抽象类必须要有抽象方法吗?
12、 **普通类和抽象类有哪些区别?**
13、 抽象类能使用 final 修饰吗?
14、 **接口和抽象类有什么区别?**
15、 **java 中 IO 流分为几种?**
16、 **BIO、NIO、AIO 有什么区别?**
17、 **Files的常用方法都有哪些?**
18、 **java 容器都有哪些?**
19、 **Collection 和 Collections 有什么区别?**
20、 **List、Set、Map 之间的区别是什么?**
21、 **HashMap 和 Hashtable 有什么区别?**
22、 **如何决定使用 HashMap 还是 TreeMap?**
23、 说一下 HashMap 的实现原理?
24、 说一下 HashSet 的实现原理?
25、 ArrayList 和 LinkedList 的区别是什么?
26、如何实现数组和 List 之间的转换?
27、ArrayList 和 Vector 的区别是什么?
28、 Array 和 ArrayList 有何区别?
29、 在 Queue 中 poll()和 remove()有什么区别?
30、 哪些集合类是线程安全的?
31、迭代器 Iterator 是什么?
32、 Iterator 怎么使用? 有什么特点?
33、 Iterator 和 ListIterator 有什么区别?

（二）多线程

34、 什么是线程
35、并行和并发有什么区别?
36、线程和进程的区别?
37、守护线程是什么?
38、创建线程有哪几种方式?
39、说一下 runnable 和 callable 有什么区别?
40、 线程有哪些状态?
41、sleep() 和 wait() 有什么区别?
42、 notify()和 notifyAll()有什么区别?
43、 线程的 run()和 start()有什么区别?
44、 创建线程池有哪几种方式?
45、线程池都有哪些状态?
46、线程池中的submit()和execute()方法有什么区别?
47、在Java程序中怎么保证多线程的运行安全?
48、多线程锁的升级原理是什么?
49、什么是死锁?
50、怎么防止死锁?
51、ThreadLocal是什么? 有哪些使用场景?
52、说一下synchronized 底层实现原理?
53、synchronized 和volatile 的区别是什么?
54、synchronized 和Lock有什么区别?
55、synchronized和reentrantLock区别是什么?

111、Spring Cloud断路器的作用是什么?
112、Spring Could的核心组件有哪些?

(十二)Mybatis

113、mybatis中#{}和${}的区别是什么?
113、mybatis有几种分页方式?
128、mybatis逻辑分页和物理分页的区别是什么?
129、mybatis是否支持延迟加载?延迟加载的原理是什么?
130、说一下mybatis的一级缓存和二级缓存?
131、mybatis有哪些执行器(Executor)?
133、mybatis分页插件的实现原理是什么?
134、mybatis如何编写一个自定义插件?

(十四)RabbitMQ

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

(十五)Kafka

152、kafka可以脱离zookeeper单独使用吗?为什么?
153、kafka有几种数据的保留策略?
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会如何处理?
155、什么情况会导致kafka运行变慢?
156、使用kafka集群需要注意什么?
157、zookeeper是什么?
158、zookeeper有哪些功能?
159、zookeeper有几种部署模式?
160、zookeeper怎么保证主从节点的状态同步?
161、集群中为什么要有主节点?
162、集群中有3台服务器,其中一个节点宕机,这个时候zookeeper还可以继续使用吗?
163、说一下zookeeper的通知机制?

十七、MySql

164、数据库的三范式是什么?
165、一张自增表里面共有7条数据,删除了最后两条数据,重启mysql数据库,又插入了一条数据,此时id是几?
166、如何获取当前数据库版本?
167、说一下ACID是什么?
168、char和varchar的区别是什么?
169、float和double的区别是什么?
170、mysql的内连接、左连接、右连接有什么区别?
171、mysql的索引是怎么实现的?
172、怎么验证mysql的索引是否满足需求?
173、说一下数据库的事务隔离?
174、说一下mysql常用的引擎?
175、说一下mysql的行锁和表锁?
176、说一下乐观锁和悲观锁?

学习题目

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

- 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

```
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
```

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
```

同样的我们进入到String 的equles方法，找到了如下答案，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，只是很多类重新了 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，因为在数轴上取值时，中间值（0.5）向右取整，所以正 0.5 是往上取整，负 0.5 是直接舍弃。

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

- 文本型 char
- 逻辑型 boolean
- 整数型 byte、short、int、long
- 浮点型 float、double

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、如何将字符串翻转

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 类的常用方法都有那些?

- 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~");
    }
}
```

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。

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，客户端和服务器端通过 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。

18、 java 容器都有哪些?

常用的容器目录：

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?

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，另外一个是模拟指针（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,根绝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(logn)

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
- HashSet的value统一为PRESENT

25、 ArrayList 和 LinkedList 的区别是什么?

最明显的区别是 ArrrayList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转换成为数组：调用ArrayList的toArray方法。

数组转换成为List：调用Arrays的asList方法。

27、ArrayList 和 Vector 的区别是什么?

- 他们都是有序的，两个类都实现了List接口（List接口继承了Collection接口）
- Vector是线程安全的，也就是说是它的方法之间是线程同步的，而ArrayList是线程序不安全的，它

  的方法之间是线程不同步的。如果只有一个线程会访问到集合，那最好是使用ArrayList，因为它不考虑线程

安全，效率会高些；如果有多个线程会访问到集合，那最好是使用Vector，因为不需要我们自己再去考虑和编

写线程安全的代码。

- 增长性 ，即Vector增长原来的一倍，ArrayList增加原来的0.5倍
- 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是指定大小的，而ArrayList大小是固定的。
- 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。

（二）多线程

34、 什么是线程

线程是操作系统能够进行运算调度的最小单位，它被包含在进程之中，是进程中的实际运作单位。程序员可以通过它进行多处理器编程，你可以使用多线程对 运算密集型任务提速。比如，如果一个线程完成一个任务要100毫秒，那么用十个线程完成改任务只需10毫秒。Java在语言层面对多线程提供了卓越的支 持，它也是一个很好的卖点

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，这是它的作用——而其他的线程只有一种，那就是用户线程。所以java里线程分2种，

1、守护线程，比如垃圾回收线程，就是最典型的守护线程。

2、用户线程，就是应用程序里的自定义线程。

守护线程

1、守护线程，专门用于服务其他的线程，如果其他的线程（即用户自定义线程）都执行完毕，连main线程也执行完毕，那么jvm就会退出（即停止运行）——此时，连jvm都停止运行了，守护线程当然也就停止执行了。

2、再换一种说法，如果有用户自定义线程存在的话，jvm就不会退出——此时，守护线程也不能退出，也就是它还要运行，干嘛呢，就是为了执行垃圾回收的任务啊。

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。

```
构造子类的一个对象：
public class TestExtendThread extends Thread{
    public void run(){
        System.out.println("ggg");
    }
    public static void main(String[] args) {
        //创建thread子类的示例，
        TestExtendThread t = new TestExtendThread();
        t.start();//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。

```
package com.yixiangyang.java.thread;

public class TransferRunnable implements Runnable {

    private Bank bank;
    private double maxAmount;
    private int fromAccount;
    private int DELAY = 10;

    public TransferRunnable(Bank b,int from, double max) {
        bank = b;
        fromAccount = from;
        maxAmount = max;
    }

    @Override
    public void run() {
        try {
            while(true){
                int toAccount = (int)(bank.size() * Math.random());
                double amount = maxAmount * Math.random();
                bank.transfer(fromAccount, toAccount, amount);
                Thread.sleep((int)(DELAY * Math.random()));
            }
        } catch (Exception e) {
        }
    }

}

public class UnsynchBankTest {
    public static final int NACCOUNTS = 100;
    public static final double INITIAL_BALANCE = 1000;
    public static void main(String[] args) {
        Bank b = new Bank(NACCOUNTS, INITIAL_BALANCE);
```

```
        int i;
        for(i = 0;i<NACCOUNTS;i++){
            TransferRunnable r = new TransferRunnable(b, i,INITIAL_BALANCE);
            Thread t = new Thread(r);
            t.start();
        }
    }
}
```

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。

```
public class CallableThreadTest implements Callable<Integer>{
    public static void main(String[] args) {
         //有返回值的线程：CallableThreadTest()
        //CallableThreadTest类似于Runnable()
        CallableThreadTest ctt = new CallableThreadTest();
        FutureTask<Integer> ft = new FutureTask<>(ctt);
        //开启ft线程
        for(int i = 0;i < 21;i++)
        {
            System.out.println(Thread.currentThread().getName()+" 的循环变量i的
值"+i);
            if(i==20)//i为20的时候创建ft线程
            {
                new Thread(ft,"有返回值的线程FutureTask").start();
            }
        }
        //ft线程结束时，获取返回值
        try
        {
            System.out.println("子线程的返回值："+ft.get());//get()方法会阻塞，直到子
线程执行结束才返回
        } catch (InterruptedException e)
        {
            e.printStackTrace();
        } catch (ExecutionException e)
        {
            e.printStackTrace();
        }
    }

    @Override
    public Integer call() throws Exception {
        int i = 0;
        for(;i<10;i++)
        {
            System.out.println(Thread.currentThread().getName()+" "+i);
        }
        return i;
    }
```

```
}
//执行方法后的返回
main 的循环变量i的值0
main 的循环变量i的值1
main 的循环变量i的值2
main 的循环变量i的值3
main 的循环变量i的值4
main 的循环变量i的值5
main 的循环变量i的值6
main 的循环变量i的值7
main 的循环变量i的值8
main 的循环变量i的值9
main 的循环变量i的值10
main 的循环变量i的值11
main 的循环变量i的值12
main 的循环变量i的值13
main 的循环变量i的值14
main 的循环变量i的值15
main 的循环变量i的值16
main 的循环变量i的值17
main 的循环变量i的值18
main 的循环变量i的值19
main 的循环变量i的值20
有返回值的线程FutureTask 0
有返回值的线程FutureTask 1
有返回值的线程FutureTask 2
有返回值的线程FutureTask 3
有返回值的线程FutureTask 4
有返回值的线程FutureTask 5
有返回值的线程FutureTask 6
有返回值的线程FutureTask 7
有返回值的线程FutureTask 8
有返回值的线程FutureTask 9
子线程的返回值：10
```

39、说一下 runnable 和 callable 有什么区别?

- 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。

 Alt文字

- 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。

```
public class NotifyTest {
    private String flag[] = { "true" };

    class NotifyThread extends Thread {
        public NotifyThread(String name) {
            super(name);
        }
        public void run() {
            try {
                sleep(30000);// 推迟30秒钟通知
            } catch (InterruptedException e) {
                e.printStackTrace();
            }

            synchronized (flag) {
                flag[0] = "false";
                flag.notifyAll();
            }
        }
    };

    class WaitThread extends Thread {
        public WaitThread(String name) {
            super(name);
        }

        public void run() {
            System.out.println(getName() + "  flag:" + flag[0]);
            synchronized (flag) {
                System.out.println(getName() + "  flag:" + flag[0]);
                while (!flag[0].equals("false")) {
```

```
                    System.out.println(getName() + " begin waiting!");
                    long waitTime = System.currentTimeMillis();
                    try {
                        flag.wait();
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                    waitTime = System.currentTimeMillis() - waitTime;
                    System.out.println("wait time :" + waitTime);
                }
                System.out.println(getName() + " end waiting!");
            }
            System.out.println(getName() + " end waiting!");

        }
    }

    public static void main(String[] args) throws InterruptedException {
        System.out.println("Main Thread Run!");
        NotifyTest test = new NotifyTest();
        NotifyThread notifyThread = test.new NotifyThread("notify01");
        WaitThread waitThread01 = test.new WaitThread("waiter01");
        WaitThread waitThread02 = test.new WaitThread("waiter02");
        WaitThread waitThread03 = test.new WaitThread("waiter03");
        notifyThread.start();
        waitThread01.start();
        waitThread02.start();
        waitThread03.start();
    }
}

Main Thread Run!
waiter01  flag:true
waiter03  flag:true
waiter02  flag:true
waiter01  flag:true
waiter01 begin waiting!
waiter02  flag:true
waiter02 begin waiting!
waiter03  flag:true
waiter03 begin waiting!
wait time :30001
waiter03 end waiting!
waiter03 end waiting!
wait time :30001
waiter02 end waiting!
waiter02 end waiting!
wait time :30001
waiter01 end waiting!
waiter01 end waiting!
```

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(),其实就相当于是调用了一个普通函数而已，直接待用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?

共有五种 jdk1.8 新增了一种

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。

```
public class TestThreadPool {
    //定长线程池，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，这时线程数量不再变化，当线程发生错误结束时，线程池会补充一个新的线程
        static ExecutorService fixedExecutor = Executors.newFixedThreadPool(3);
        public static void main(String[] args) {
            testFixedExecutor();
        }
        //测试定长线程池，线程池的容量为3，提交6个任务，根据打印结果可以看出先执行前3个任务，3个任务结束后再执行后面的任务
        private static void testFixedExecutor() {
            for (int i = 0; i < 6; i++) {
                final int index = i;
                fixedExecutor.execute(new Runnable() {
                    public void run() {
                        try {
                            Thread.sleep(3000);
                        } catch (InterruptedException e) {
                            e.printStackTrace();
                        }
                        System.out.println(Thread.currentThread().getName() + " index:" + index);
                    }
                });
            }
            try {
                Thread.sleep(4000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println("4秒后...");
            fixedExecutor.shutdown();
        }

}
//运行结果
pool-1-thread-1 index:0
pool-1-thread-3 index:2
pool-1-thread-2 index:1
4秒后...
```

```
pool-1-thread-1 index:3
pool-1-thread-3 index:4
pool-1-thread-2 index:5
```

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。

```
public class TestNewCachedThreadPool {
    //可缓存的线程池，如果线程池的容量超过了任务数，自动回收空闲线程，任务增加时可以自动添加新线程，线程池的容量不限制
    static ExecutorService cachedExecutor = Executors.newCachedThreadPool();
    public static void main(String[] args) {
        testCachedExecutor();
    }
    // 测试可缓存线程池
    private static void testCachedExecutor() {
        for (int i = 0; i < 20; i++) {
            final int index = i;
            cachedExecutor.execute(new Runnable() {
                public void run() {
                    try {
                        Thread.sleep(4000);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                    System.out.println(Thread.currentThread().getName() + "
index:" + index);
                }
            });
        }
        try {
            Thread.sleep(7000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("7秒后...");
        cachedExecutor.shutdown();
    }
}
//打印结果
pool-1-thread-1 index:0
pool-1-thread-2 index:1
pool-1-thread-6 index:5
pool-1-thread-5 index:4
pool-1-thread-3 index:2
pool-1-thread-4 index:3
pool-1-thread-7 index:6
pool-1-thread-8 index:7
pool-1-thread-9 index:8
pool-1-thread-10 index:9
pool-1-thread-11 index:10
pool-1-thread-12 index:11
pool-1-thread-13 index:12
pool-1-thread-14 index:13
pool-1-thread-15 index:14
```

```
pool-1-thread-16 index:15
pool-1-thread-17 index:16
pool-1-thread-18 index:17
pool-1-thread-19 index:18
pool-1-thread-20 index:19
7秒后...
```

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。

```
public class TestSingleThreadExecutor {
    static ExecutorService singleExecutor = Executors.newSingleThreadExecutor();

    public static void main(String[] args) {
        testSingleExecutor();
    }

    // 测试单线程的线程池
    private static void testSingleExecutor() {
        for (int i = 0; i < 6; i++) {
            final int index = i;
            singleExecutor.execute(new Runnable() {
                public void run() {
                    try {
                        Thread.sleep(3000);
                        System.out.println("睡眠三秒"+ index);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                    System.out.println(Thread.currentThread().getName() + "
index:" + index);
                }
            });
        }
        try {
            Thread.sleep(4000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("4秒后...");
        singleExecutor.shutdown();
    }
}
//执行结果
睡眠三秒0
pool-1-thread-1 index:0
4秒后...
睡眠三秒1
pool-1-thread-1 index:1
睡眠三秒2
pool-1-thread-1 index:2
睡眠三秒3
pool-1-thread-1 index:3
睡眠三秒4
```

```
pool-1-thread-1 index:4
```

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。

```
public class TestNewScheduledThreadPool {
    // 定长线程池，可执行周期性的任务
    static ScheduledExecutorService scheduledExecutor =
Executors.newScheduledThreadPool(3);

    public static void main(String[] args) {
        testScheduledExecutor();
    }

    // 测试定长、可周期执行的线程池
    private static void testScheduledExecutor() {
        for (int i = 0; i < 3; i++) {
            final int index = i;
            // scheduleWithFixedDelay 固定的延迟时间执行任务； scheduleAtFixedRate
固定的频率执行任务
            scheduledExecutor.scheduleWithFixedDelay(new Runnable() {
                public void run() {
                    System.out.println(Thread.currentThread().getName() + "
index:" + index);
                }
            }, 0, 2, TimeUnit.SECONDS);//每两秒去执行一次
        }

        try {
            Thread.sleep(4000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("6秒后...");
        scheduledExecutor.shutdown();
    }

}
//执行结果
pool-1-thread-1 index:0
pool-1-thread-1 index:1
pool-1-thread-1 index:2
pool-1-thread-1 index:0
pool-1-thread-1 index:1
pool-1-thread-1 index:2
6秒后...
```

5. newWorkStealingPool，这个是 JDK1.8 版本加入的一种线程池，stealing 翻译为抢断、窃取的意思，它实现的一个线程池和上面4种都不一样，用的是 ForkJoinPool 类

任务窃取线程池，不保证执行顺序，适合任务耗时差异较大。

线程池中有多个线程队列，有的线程队列中有大量的比较耗时的任务堆积，而有的线程队列却是空的，就存在有的线程处于饥饿状态，当一个线程处于饥饿状态时，它就会去其它的线程队列中窃取任务。解决饥饿导致的效率问题。

默认创建的并行 level 是 CPU 的核数。主线程结束，即使线程池有任务也会立即停止。

```
public class TestNewWorkStealingPool {
    // 任务窃取线程池
    static ExecutorService workStealingExecutor =
Executors.newWorkStealingPool();

    public static void main(String[] args) {
        testWorkStealingExecutor();
    }

    // 测试任务窃取线程池
    private static void testWorkStealingExecutor() {
        for (int i = 0; i < 10; i++) {//我这里电脑默认是4核的,这里创建10个任务进行测试
            final int index = i;
            workStealingExecutor.execute(new Runnable() {
                public void run() {
                    try {
                        Thread.sleep(3000);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                    System.out.println(Thread.currentThread().getName() + "
index:" + index);
                }
            });
        }

        try {
            Thread.sleep(4000);// 这里主线程不休眠，不会有打印输出
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("4秒后...");
//          workStealingExecutor.shutdown();
    }

}
//
ForkJoinPool-1-worker-2 index:1
ForkJoinPool-1-worker-1 index:0
ForkJoinPool-1-worker-3 index:2
ForkJoinPool-1-worker-0 index:3
4秒后...
```

45、线程池都有哪些状态?

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

1. Running

   (1)状态说明:线程池在Running状态的时候，能够接受新任务，以及对已添加的任务进行处理。

   (2)状态切换:线程池的初始化状态是Running,也就是说，线程池一旦被创建就处于Running状态，并且线程池中任务数为0

2. ShutDown

   (1)状态说明:线程池出去ShutDown状态时，不接收新任务，但能处理已添加的任务。

   (2)状态切换:调用线程池的shutDownNow()接口时，线程池由(Running or ShutDown) --> stop

3. Stop

   (1)状态说明:线程池处于Stop状态的时候，不接收新任务，不处理已添加的任务，并且会中断正在执行的任务。

   (2)状态切换:调用线程池的shutDownNow()接口时，线程池由(running or shutDown)-->stop

4. Tidying

   (1)状态说明:当所有的任务已终止，ctl记录的任务数量为0，线程池会变为Tidying状态，当线程池变为Tidying状态时会执行钩子函数terminated()。terminated()在ThreadPoolExecutor类中是空的，若用户想在线程池状态变为Tidying时，进行相应的处理，可以通过重载terminated()函数来实现。

   (2)状态切换:当线程池在ShutDown状态下,阻塞队列为空并且线程池中执行的任务也为空时，就会由ShutDown -->Tidying

   当线程池在stop状态下，线程池中执行的任务为空时，就会由Stop --> Tidying

5. Terminated

   (1)状态说明:线程池彻底终止,就会变成Terminated状态.

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

46、线程池中的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?

- 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的被其他线程看到（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，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。

49、什么是死锁?

死锁是指两个活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现象，若无外力，它们将一直无法推进下去。此时称系统处于死锁状态或者系统产生了死锁，这些永远在互相等待的进行称为死锁进行。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正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：进程对锁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，但是该资源不可能被其他进程占有，此时请求阻塞，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，只能在使用完成后释放。
- 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，只要上述条件之一不满足就不会发生死锁。

所以，在系统设计，进程调度等方面如何注意如何不让着四个必要条件成立，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，也要进程处于等待的情况下占用资源，因此对于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ThreadLoad类来支持线程的局部变量，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被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应用就存在内存泄漏的风险。

52、说一下synchronized 底层实现原理?

synchronized 可以保证方法或代码块在运行时，同一时刻只有一个方法进入到临界区，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，这是对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，只有当前线程方位该变量，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。
- volatie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的内置关键字，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需要再finally中手动释放锁（unlock()方法释放锁）,否则容易造成死锁。
- 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，线程就可以不用等待直接结束了。
- 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。
- Lock锁适合大量同步代码的同步问题，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一样的关键字,ReentrantLock是类，这是两者的本质区别。ReentrantLock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可以灵活的实现多路通知

另外，两者的锁机制也是不一样的，ReentrantLock底层调用的是Unsafe的park方法加锁，synchronized操作的应该是对象投中的mark word。

56、说一下atomic的原理?

Atomic包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Sunsafe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++一样指针越界到其他进程的问题。

（四）反射

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，能否知道这个类的有哪些属性和方法?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的功能：

- 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：

- 当你想把内存中的对象状态保存到一个文件和数据库的时候。
- 当你想用套接字在网络上传输对象的时候。
- 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，比如日志，加事务等。可以给这个类创建一个代理，顾名思义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。这个代理类并不是蒂尼好的，是动态生成的。具有解耦意义，灵活扩展性强。

动态代理的应用：

- Spring的AOP
- 加事务

- 加选项
- 加日志

60、怎么实现动态代理?

首先必须定义一个接口，还要一个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，因为调用它的newInstance() 可以产生一个代理对象，其实他只是一个产生代理对象的工具类)。利用到InvocationHander,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。

（五）对象拷贝

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?

1. 实现Cloneable接口并重写Object类中的clone()方法。

```
package com.yixiangyang.java.employee;

import java.util.Date;
import java.util.GregorianCalendar;

public class CloneTest {

    public static void main(String[] args){
        //
        //original.setHireDay(new Date(2017, 12, 11));
        try {
            Employee2 original = new CloneTest().new Employee2("aaaaa", 2222);
            original.setHireDay(2017, 11, 14);
            Employee2 copy = original.clone();
            copy.raiseSalary(100);
            copy.setHireDay(2012, 12, 12);
            System.out.println("original:"+original.toString());
            System.out.println("copy:"+copy.toString());
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
    class Employee2 implements Cloneable{
        /**名字*/
        private String name;
        /**薪水*/
        private double salary;
        /**聘请日期*/
        private Date hireDay;
        public Employee2(String name, double salary) {
            super();
            this.name = name;
            this.salary = salary;
            this.hireDay = new Date();
        }
        public Employee2 clone() throws CloneNotSupportedException{
            Employee2 cloned = (Employee2)super.clone();
```

```
            cloned.hireDay =(Date) hireDay.clone();
            return cloned;
        }

        @Override
        public String toString() {
            return "Employee2 [name=" + name + ", salary=" + salary + ",
hireDay=" + hireDay + "]";
        }
        public void setHireDay(int year,int month,int day){
            Date newHireDay = new GregorianCalendar(year, month-1,
day).getTime();
            hireDay.setTime(newHireDay.getTime());
        }
        public void raiseSalary(double byPercent){
            double raise = salary * byPercent / 100;
            salary += raise;
        }
    }
}
//运行结果
original:Employee2 [name=aaaaa, salary=2222.0, hireDay=Tue Nov 14 00:00:00 CST
2017]
copy:Employee2 [name=aaaaa, salary=4444.0, hireDay=Wed Dec 12 00:00:00 CST 2012]
```

2. 实现Serializable，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：

```
/**
 * 实现Serializable，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
 * @author 15138
 *
 */
public class MyCloneUtil {
    @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
    }
}


package com.yixiangyang.java.Serializable;

import java.io.Serializable;
```

```java
public class Car implements Serializable{
    /**
     *
     */
    private static final long serialVersionUID = 9071473629977418542L;
    private String brand;       // 品牌
    private int maxSpeed;       // 最高时速

    public Car(String brand, int maxSpeed) {
        super();
        this.brand = brand;
        this.maxSpeed = maxSpeed;
    }

    public Car() {
        super();
        // TODO Auto-generated constructor stub
    }

    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 + "]";
    }

}

package com.yixiangyang.java.Serializable;

import java.io.Serializable;

public class Person implements Serializable{
    /**
     *
     */
    private static final long serialVersionUID = -7524108921652910420L;
    private String name; // 姓名
    private int age; // 年龄
    private Car car;      // 座驾

    public Person() {
        super();
        // TODO Auto-generated constructor stub
    }
    public Person(String name, int age, Car car) {
        super();
        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 + "]";
    }

}


package com.yixiangyang.java.Serializable;

public class CloneTest {
    public static void main(String[] args) {
        try {
             Person p1 = new Person("郭靖", 33, new Car("Benz", 300));
             Person p2 = MyCloneUtil.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) {
            // TODO: handle exception
        }

    }
}

//运行结果
Person [name=郭靖, age=33, car=Car [brand=Benz, maxSpeed=300]]
```

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，这种是方案明显优于使用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、深拷贝和浅拷贝的区别是什么?

- 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，所以修改其中的任意的值，另一个值都会随之变化，这就是浅拷贝。
- 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，（例如JSON.parse()和JSON.staringify(),但是此方法无法复制函数的类型）。

（六）Java Web

64、jsp和servlet有什么区别?

- jsp经过编译后就变成了Servlet，（Jsp的本质就是Servlet,Jvm只能够识别Java的类，不能够识别jsp的代码，web容器将jsp的代码编译成JVM能够识别的java类）。
- jsp更擅长于页面的显示，servlet更擅长于逻辑的控制。
- Servlet中没有内置对象，Jsp中的内置对象都必须通过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，有Jsp容器完成。而Servlet则是完成整个的Java类，这个类的Service方法用于生成对客户端的相应。

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中作用域?

jsp中的四种作用域包括page、request、session、application，具体：

- 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的相关对象和属性。一个请求可能跨越多个页面，涉及多个web组件。需要再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，所有服务端需要记录用户的状态时，就需要用某种机制来识别具体的用户，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，所以并不知道是那个用户操作的，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的方法有很多，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信息都是放在内存中的，使用一些缓存服务器比如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，客户端都会发送响应的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sessionID，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后面都会附加上一个诸如sid=XXX这样的参数，服务端据此来识别用户。

- cookie其实还可以再用一些方便用户的场景下，设想你某次登录过一个网站，下次登录的时候不想再次输入账号了，怎么办？这个信息可以写在cookie里，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写了，能够方便一下用户，这也是cookie名称的由来。

总结：

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?

session是一个存在服务器上类似一个散列表格的文件。里面存有我们需要的信息，在我们需要的时候可以从里面取出来。类似于一个大号的map，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使用sessionId来确定当前对话所对应的服务器session，而sessionId使用cookie来传递的，禁用cookie相当于失去了sessionId，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.user_trans_sid = 1",或者编译时打开了"-enable-trans-sid"选项，让php自动跨页传递sessionId。
- 手动通过URL传值、隐藏表单传递sessionId。
- 用文件、数据库等形式保存sessionId,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，每次请求都会创建一个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把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，只能设置为多例。

Spring MVC是方法级别的拦截，一个方法对应一个request上下文，所以方法基本上是独立的，独享request，response数据。而每个方法同时又和一个URL对应，参数的传递是直接注入到方法中的，是方法所独有的。处理结果通过ModeMap返回给框架。在Spring整合时，Spring MVC的Controller Bean默认单例Singleton，所以没有共享的属性，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，Spring MVC这是用独立的Aop方式，这样导致Struts2的配置文件量比Spring MVC大。

- 底层框架的不同

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

- 性能反面

Struts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Strutes2高。

71、如何避免sql注入?

- PreparedStatement
- 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含非法字符
- JSP页面判断代码

72、什么事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属于被动式且用户客户端的攻击方式。

XSS防范的总体思路是：对输入（和RUL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。

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 ，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。

（七）异常

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是讲异常声明但是不处理，而是将异常往上传，谁掉用我就交给谁处理。而throw则是抛出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放在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类事所有类的父类，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。

75、try-catch-finally中哪个部分可以省略?

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，代码再运行时也只不过是加上了一层皮。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，则是不管有没有捕获异常，都要进行扫尾处理。

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

会执行，并且在return之前执行。

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给适当的格式时，抛出该异常。

- FileNotFondException：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

79、Http响应码301和302代表的是什么？有什么区别?

301、302都是Http状态的编码，都达标者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代表了两种转发请求转发方式：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或其他信息资源，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。

81、简述tcp和udp的区别?

- TCP面向连接（如打电话要先拨号建立间接），UDP是无连接的，即发送数据之前不需要建立间接。
- 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，哪些是被对方接收到的，三次握手的过程即是通信双方相互告诉序列化起始值，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。

TCP的三次握手

第一次握手：建立连接时,客户端发送syn包(syn=j)到服务器,并进入SYN_SEND状态,等待服务器确认；
SYN：同步序列编号(Synchronize Sequence Numbers)。
第二次握手：服务器收到syn包,必须确认客户的SYN（ack=j+1）,同时自己也发送一个SYN包（syn=k），即SYN+ACK包,此时服务器进入SYN_RECV状态。
第三次握手：客户端收到服务器的SYN + ACK包,向服务器发送确认包ACK(ack=k+1),此包发送完毕,客户端和服务器进入ESTABLISHED状态,完成三次握手。

完成三次握手,客户端与服务器开始传送数据

【问题1】为什么连接的时候是三次握手，关闭的时候却是四次握手?

答：因为当Server端收到Client端的SYN连接请求报文后，可以直接发送SYN+ACK报文。其中ACK报文是用来应答的，SYN报文是用来同步的。但是关闭连接时，当Server端收到FIN报文时，很可能并不会立即关闭SOCKET，所以只能先回复一个ACK报文，告诉Client端，"你发的FIN报文我收到了"。只有等到我Server端所有的报文都发送完了，我才能发送FIN报文，因此不能一起发送。故需要四步握手。

【问题2】为什么TIME_WAIT状态需要经过2MSL(最大报文段生存时间)才能返回到CLOSE状态?

答：虽然按道理，四个报文都发送完毕，我们可以直接进入CLOSE状态了，但是我们必须假象网络是不可靠的，有可以最后一个ACK丢失。所以TIME_WAIT状态就是用来重发可能丢失的ACK报文。在Client发送出最后的ACK回复，但该ACK可能丢失。Server如果没有收到ACK，将不断重复发送FIN片段。所以Client不能立即关闭，它必须确认Server接收到了该ACK。Client会在发送出ACK之后进入到TIME_WAIT状态。Client会设置一个计时器，等待2MSL的时间。如果在该时间内再次收到FIN，那么Client会重发ACK并再次等待2MSL。所谓的2MSL是两倍的MSL(Maximum Segment Lifetime)。MSL指一个片段在网络中最大的存活时间，2MSL就是一个发送和一个回复所需的最大时间。如果直到2MSL，Client都没有再次收到FIN，那么Client推断ACK已经被成功接收，则结束TCP连接。

【问题3】为什么不能用两次握手进行连接?

答：3次握手完成两个重要的功能，既要双方做好发送数据的准备工作(双方都知道彼此已准备好)，也要允许双方就初始序列号进行协商，这个序列号在握手过程中被发送和确认。

现在把三次握手改成仅需要两次握手，死锁是可能发生的。作为例子，考虑计算机S和C之间的通信，假定C给S发送一个连接请求分组，S收到了这个分组，并发 送了确认应答分组。按照两次握手的协定，S认为连接已经成功地建立了，可以开始发送数据分组。可是，C在S的应答分组在传输中被丢失的情况下，将不知道S 是否已准备好，不知道S建立什么样的序列号，C甚至怀疑S是否收到自己的连接请求分组。在这种情况下，C认为连接还未建立成功，将忽略S发来的任何数据分 组，只等待连接确认应答分组。而S在发出的分组超时后，重复发送同样的分组。这样就形成了死锁。

【问题4】如果已经建立了连接，但是客户端突然出现故障了怎么办?

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

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，但当发送数据包过于小时，那么TCP协议默认的会启用Nagle算法，将这些较小的数据包进行合并发送（缓冲数据发送是一个堆压的过程），这个合并过程就是在发送穿冲去进行的，这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：数据到达接收方，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这时候会出现一个问题，就是我们再程序中调用的读取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到缓冲区末尾，等我们读取数据时就是一个粘包。（放数据的速度 > 应用层拿数据的速度）

84、OSI的七层模型有哪些?

1. 应用层：网络服务与最终用户的一个接口。
2. 表示层：数据的表示、安全、压缩。
3. 会话层：建立、管理、终止会话。
4. 传输层：定义传输数据的协议端口号，以及流控和差错校验。
5. 网络层：进行逻辑地址寻址，实现不同网络之间的路径选择。
6. 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。
7. 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。
- 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中传送的参数是有限制的，而POST没有。
- 对参数的数据类型，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、Java后端如何实现跨域?

- 使用Java过滤器过滤
- 后台HTTP请求转发
- 后台配置同源CORS
- 使用springCloud网关
- 使用nginx转发

1、jsonp
利用了 script 不受同源策略的限制
缺点：只能 get 方式，易受到 XSS攻击

2、CORS（Cross-Origin Resource Sharing）,跨域资源共享
当使用XMLHttpRequest发送请求时，如果浏览器发现违反了同源策略就会自动加上一个请求头 origin；
后端在接受到请求后确定响应后会在后端在接受到请求后确定响应后会在 Response Headers 中加入一个属性 Access-Control-Allow-Origin；
浏览器判断响应中的 Access-Control-Allow-Origin 值是否和当前的地址相同，匹配成功后才继续响应处理，否则报错
缺点：忽略 cookie，浏览器版本有一定要求

3、代理跨域请求
前端向发送请求，经过代理，请求需要的服务器资源
缺点：需要额外的代理服务器

4、Html5 postMessage 方法
允许来自不同源的脚本采用异步方式进行有限的通信，可以实现跨文本、多窗口、跨域消息传递
缺点：浏览器版本要求，部分浏览器要配置放开跨域限制

5、修改 document.domain 跨子域
相同主域名下的不同子域名资源，设置 document.domain 为 相同的一级域名
缺点：同一 一级域名；相同协议；相同端口

6、基于 Html5 websocket 协议
websocket 是 Html5 一种新的协议，基于该协议可以做到浏览器与服务器全双工通信，允许跨域请求
缺点：浏览器一定版本要求，服务器需要支持 websocket 协议

7、document.xxx + iframe
通过 iframe 是浏览器非同源标签，加载内容中转，传到当前页面的属性中
缺点：页面的属性值有大小限制

###（九、设计模式）

88、Java的有几种设计模式?

23中设计模式 共分为三大类：

- 创建型模式（5种）：工厂方法模式、抽象工厂模式、单例模式、建造者模式、原型模式。
- 结构型模式（7种）：适配器模式、装饰器模式、代理模式、外观模式、桥接模式、组合模式、享元模式
- 行为型模式（11种）：策略模式、模板方法模式、观察者模式、迭代子模式、责任链模式、命令模式、备忘者模式、解释器模式。

设计模式遵循的六个原则

1. 开闭原则(Open Close Principle)

   对扩展开放，对修改关闭。

2. 里氏代换原则（Liskov Substitution Principle）

   只有当衍生类可以替换掉基类，软件单位的功能不受到影响时，基类才能真正被复用，而衍生类也能够在基类的基础上增加新的行为。

3. 依赖倒转原则（Dependence Inversion Principle）

   这个是开闭原则的基础，对接口编程，依赖于抽象而不依赖于具体。

4. 接口隔离原则（Interface Segregation Principle）

使用多个隔离的接口来降低耦合度。

5. 迪米特法则（最少知道原则）Demeter Principle

   一个实体应当尽量少的与其他实体之间发生相互作用，使得系统功能模块相对独立。

6. 合成复用原则(Composite reuse Principle)

   原则是尽量使用合成/聚合的方式，而不是使用继承。继承实际上破坏了类的封装性，超类的方法可能会被子类修改。

89、简单工厂和抽象工厂有什么区别?

简单工厂模式：

是由一个工厂对象创建产品实例，简单工厂模式的工厂类一般是使用静态方法，通过不同的参数创建不同的对象实例，可以生产结构中的任意产品，不能增加新的产品

抽象工厂模式：

提供一个创建一系列相关或相互依赖对象的接口，而无需指定他们具体的类，生产多个系列产品，生产不同产品族的全部产品，不能新增产品，可以新增产品族

###（十、Spring/Spring MVC）

90、为什么要用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、解释一下什么事aop?

AOP(Aspect-Oriented Programming,面向方面编程)，可以说是OOP（Object-Oriented Programing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，但并不适合从左到右的关系。例如日志功能。日志代码往往水平的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代码称之为横切（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“，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”。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门：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征是，他们经常发生的核心关注点的多处，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?

OC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：

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：

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条件，但是具体还是要结合具体的scope的Bean去研究。

96、spring支持几种bean的作用域?

Spring 容器负责创建应用程序中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

98、Spring事务实现方式有哪些?

- 编程式事务管理对于基于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。
- 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，如果没有采用必要的隔离机制，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了这个表中的数据，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

- 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，负责拦截我们定义的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，标识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。

value,method

- value:指定请求的实际地址，请求地址可以是URI Template模板
- 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，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和@Resource的区别是什么?

@Resource的作用相当于@Autowired，只不过@Autowired按byType自动注入，而@Resource默认按 byName自动注入罢了。@Resource有两个属性是比较重要的，分是name和type，Spring将@Resource注解的name属性解析为bean的名字，而type属性则解析为bean的类型。所以如果使用name属性，则使用byName的自动注入策略，而使用type属性时则使用byType自动注入策略。如果既不指定name也不指定type属性，这时将通过反射机制使用byName自动注入策略。

@Resource装配顺序

1. 如果同时指定了name和type，则从Spring上下文中找到唯一匹配的bean进行装配，找不到则抛出异常
2. 如果指定了name，则从上下文中查找名称（id）匹配的bean进行装配，找不到则抛出异常
3. 如果指定了type，则从上下文中找到类型匹配的唯一bean进行装配，找不到或者找到多个，都会抛出异常
4. 如果既没有指定name，又没有指定type，则自动按照byName方式进行装配；如果没有匹配，则回退为一个原始类型进行匹配，如果匹配则自动装配；

区别：

1、 @Autowired与@Resource都可以用来装配bean. 都可以写在字段上,或写在setter方法上。

2、 @Autowired默认按类型装配（这个注解是属业spring的），默认情况下必须要求依赖对象必须存在，如果要允许null值，可以设置它的required属性为false

3、@Resource（这个注解属于J2EE的），默认按照名称进行装配，名称可以通过name属性进行指定，如果没有指定name属性，当注解写在字段上时，默认取字段名进行安装名称查找，如果注解写在setter方法上默认取属性名进行装配。当找不到与名称匹配的bean时才按照类型进行装配。但是需要注意的是，如果name属性一旦指定，就只会按照名称进行装配。

（十一）Spring Boot/Spring Could

104、什么是Spring Boot?

在Spring 框架的这个大家族中，产生了很多衍生框架，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bean，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SpirngBoot是一个服务与框架的框架，服务范围时简化配置文件。

105、为什么要使用SpringBoot?

- SpringBoot使编码变的简单
- SpringBoot使配置变的简单
- SpringBoot使部署变的简单
- SpringBoot使监控变的简单
- Spring的不足

106、SpringBoot核心配置文件是什么?

SpringBoot提供了两种常用的配置文件：

- properties文件
- yml文件

107、SpringBoot配置文件有哪几种类型？他们有什么区别?

SpringBoot提供了两种常用的配置文件，分别是properties文件和yml文件。相对properties文件来说，yml文件更年轻，yml文件通过空格来确定层级关系，使配置文件结构更清晰，但也会因为微不足道的空格而破坏了层级关系。

108、SpringBoot有哪些方式可以实现热部署?

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JavaEE5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之前的关系，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，二十对hibernate-core进行封装，使得Hibernate符合JPA的规范。

110、什么是SpringCould?

SpringCould就是致力于分布式系统、云服务的框架。

SpringCould是整个Spring家族中新的成员，是最新云服务火爆的必然产物。

SpringCould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SpringCould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。这些服务可以再任何环境下运行，包括分布式环境，也包括开发人员自己的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，即很有可能失败，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时间，该信息可以用来监测由断路器保护系统的部位的健康状况，或以提醒管理员当断路器跳闸，以在打开状态。

112、Spring Could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服务器作为服务注册服务器。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. 客户端的负载均衡--Netflix Ribbon

   Ribbon,主要提供客户侧的软件负责均衡算法。Ribbon客户端组件提供一系列完善的配置选项，比如超时、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Ngingx ,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，需要配置Spring Cloud Bus实现动态的配置更新。

(十二)Mybatis

113、mybatis中#{}和${}的区别是什么?

- #{}是预编译处理，${}是字符串替换。
- Mybatis在处理#{}时，会将sql中的#{}替换为？号，调用PreparedStatement的set方法来赋值。
- Mybatis在处理${}时，就是把${}替换成变量的值。
- 使用#{}可以有效的防止sql注入，提高系统安全性。

113、mybatis有几种分页方式?

- 数组分页
- sql分页
- 拦截器分页
- RowBounds分页

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，然而其它性能上的有点足以弥补这个缺点。

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conllection关联集合对象的延迟加载，association指的就是一对一，conllection指的就是就是一对多查询。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。

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),并可以自定义存储源，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）的进行了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，简而言之，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都添加到批量处理中（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的方法

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：

```
public interface Interceptor {
    Object intercept(Invocation invocation) throws Throwable;
    Object plugin(Object target);
    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的实现类；
- args：表示方法参数。

（十四）RabbitMQ

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，后端应用从队列中依次获得消息处理，高峰时的大量订单可以积压在队列里慢慢处理掉。由于同步异常意味着阻塞，而大量线程的阻塞会降低计算机的性能。

4. 消息驱动的价格（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）,阶段之间用消息队列连接起来，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讯。

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Rabbitmq之间建立连接的管理器，程序代码中使用。
- Channel(信道)：消息推送使用的管道。
- 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-RaabbitMQ server。其内部均含有独立的queue、exchange和binding等，但最主要的是，其拥有独立的权限系统，可以做到vhost范围的用户控制。当然，荣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的消息是如何发送的?

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queue可以接收消息。
- topic：所有符合routingKey(此时可以使一个表达式)的routingKey所bingd的queue可以接收消息。

145、rabbitmq怎么实现延迟消息队列?

- 通过消息后期进入死信交换器，再由交换器转发到延迟消费队列，实现延迟功能。
- 使用Rabbitmq-delayed-messge-exchange插件实现延迟功能。

146、rabbitmq集群有什么用?

主要有以下两个用途：

- 高可用：某个服务器出现问题，真个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?

- 各个节点之间使用“--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“秘钥”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：如果每个节点都用于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对集群节点的停止顺序有要求吗?

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。

（十五）Kafka

152、kafka可以脱离zookeeper单独使用吗？为什么？

kafaka不可以脱离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有几种数据的保留策略？

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会如何处理？

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

155、什么情况会导致kafka运行变慢？

- cup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使用了，设置为单数容错率更高。

###（十六）Zookeeper

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，即可以有多县县城同时读一个资源，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：多台集群上运行
- 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zeb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，切大多数server完成了和leader的状态同步后，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，所以就需要主节点。

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

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时间，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变化做出业务上的改变。

十七、MySql

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的一部分属性。
- 第三范式：任何非主属性不依赖于其他非主属性。

165、一张自增表里面共有7条数据，删除了最后两条数据，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18
- 表类型如果是InnoDB,那id就是15

  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，不会结束在中间的某个环节。事务在执行过程中发生国务，会被恢复（rollback）到事务开始前的状态，就好像这个事务从来没有被执行过一样。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的三个字符的时候，他们占的空间还是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，存储的值是每格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，两者的使用需要权衡。

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最多可以存储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，右边的表显示符合条件的数据，，右连接是右边的表全部显示出来，左边的表显示符合条件的数据。

171、mysql的索引是怎么实现的?

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殷勤的索引都是B+树实现的，B+树的搜索效率，可以达到二分法的性能，找到数据区域之后就找到了完整的数据结构了，所以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。

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）。
- 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，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界别，该隔离界别能防止脏读、幻读、不可重复读。

脏读：表示一个事务能够读取另一个事务中未提交的数据。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调记录，但是第二次同等条件下查询缺有n+1条记录，这就好像差生了幻觉。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，会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，需要进行扫描权标。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权标。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下在此期间别人有没有更新这个数据。
- 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的时候就会阻止，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version是否一致，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性能优化?

- 为搜索字段创建索引
- 避免使用select * 列出要查询的字段
- 垂直分隔分表
- 选择正确的存储引擎。

十八 Redis

179、Redis是什么？都有那些使用场景?

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?

- memached所有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快的多

181、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。

而且线程并不代表慢，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，由于缓存是不命中需要从数据库中查询，查不到数据则不写入到缓存，这将导致这个不存在的数据每次都要到数据库中查询，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尽力更集中地房子处理业务逻辑上。

187、如何保证缓存和数据库的一致性?

- 合理设置缓存的过期时间。
- 新增、更改、删除数据库操作的同时，同步更新redis，可以使用事务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是通过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，程序的执行如果超过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（hashs），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（servver.db[i].expires）中挑选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-enviction（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

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（Excution Engine）,将字节码翻译成底层系统指令，再交由CPU去执行，而这个过程中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?

1. 栈内存存储的是局部变量而堆内存存储的是实体；
2. 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；

3. 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：

1. 加载：根据查找路径找到相应的 class 文件然后导入；
2. 检查：检查加载的 class 文件的正确性；
3. 准备：给类中的静态变量分配内存空间；
4. 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；
5. 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 老生代版本，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收回，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：初始化推大小为 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。