EE面试宝典(西安就业部大纲)

> 所有面试题都有对应的代码实现，该代码实现全部属于
> 传智播客西安就业部请各位同仁把自己写的代码实现提
> 交到该仓库提交方式请创建分支提交，传智播客西安就
> 业部管理人员会定期合并分支到主干中如有疑问请联系
> 传智播客西安就业部管理人员，做相应的变动
> 代码仓库地址
> https://github.com/wolvesleader/interview
> 问题反馈和答疑联系13659206656 微信同手机号，添加微信请备注 姓名+多少期 例如 李魁41

Java基础

基础语法概念

1. 面向对象深刻理解

> 小白回答：面向对象就是把问题的处理过程委托类程序，我们只用关注结果，比如把大象装到冰箱的过程....
> 大神回答：面向对象是一种编程思想，不只是在java语言中存在，在其他高级别的语言中也存在，可以把面向对象分为2部分来理解，面向和对象，面向的意思就是正对某一个东西来做一系列的操作对象就是要操作的东西，完整的就是 针对对象来做一系列的操作，在java中对象的创建通常是通过new关键字来实现的。

2. 面向对象的特点

> 面向对象特点有3中
> 1)继承
> java中继承是通过关键字extends来实现的，主要的作用是提高代码的高内聚性，缺点是代码的耦合度会变高
> java中类与类之间是单继承，接口与接口之间是多继承
> 2)封装
> 因为java语言是面向对象的语言，一切的操作都是基于对象来实现的，但是一个对象有很多的属性或者方法
> ，因此我们把对象分装为一个类，暴露给外界能访问的方法，提高了代码的安全性
> 3)多态
> 对态指的是编译时和运行时对象不同的形态例如

```
1   public interface Person{}
2   public class PersonImpl implements Person{}
3   Person p = new PersonImpl()
4   编译时p为person
5   运行时p为PersonImpl
```

3. Java语言和其他语言区别

> 1)一处编译，多处运行

> 2)健全的安全体系
> 3)兼容不同平台
> 4)自带内存管理机制

4. JRE、JDK、JVM区别

> JRE全称java runtime environment，指的是java运行时需要的各种类库，在正式的项目中只需要安装jre即可
> JDK全称java development kit，包括了JRE、java工具、java基础类库，主要是开发人员使用
> JVM全称Java Virtual Machine，java编译好的class文件最终需要在jvm中运行

5. classpath作用

> java系统环境变量，我们编译.java是通过javac命令来实现，运行是通过java来实现
> 这2个命令是在JDK包中存放的，如果我们的.java文件不放在javac命令所在的目录下
> 系统就会报找不到javac,配置classpath是为了在系统的任意目录下都可以运行javac java命令

6. 抽象类不能创建对象为什么提供了构造器

> 调用抽象类的方法时候jvm会自动帮助我们创建抽象类对象
> 有构造器的作用主要是为了让系统帮助我们创建抽象类对象

参考代码
https://github.com/wolvesleader/interview/tree/master/base_java/src/main/java/com/quincy/java/base/abstractconstructor

7. 抽象类

> 抽象类在java中是关键词abstract来修饰的类
> 抽象类就像一个模版，可以帮助我们规范开发，保证创建出来的类具有共同的行为
> 抽象类不能通过new 关键字来创建对象

8. 接口

> 接口在java中通过interface来定义
> 主要是提高代码的可扩展性

9. JDK1.8之前和JDK1.8之后接口不同

> JDK1.8之前接口只能有抽象方法
> JDK1.8之后接口中可以有不同方法，但是方法必须用static或者default来修饰

参考代码
https://github.com/wolvesleader/interview/tree/master/base_java/src/main/java/com/quincy/java/base/jdk8

10. 接口和抽象类区别

> 抽象类有构造器 接口没有构造器
> 抽象类中可以有普通方法 接口中有没有要区分JDK版本
> 抽象类实现关键字用abstract 接口实现关键字是interface
> 抽象类中成员变量自己定义权限修饰 接口成员变量默认为public static final

11. 权限修饰符

访问权限	当前类	同包	子类	其他包
public	ok	ok	ok	ok
protect	ok	ok	ok	no
default	ok	ok	no	no
private	ok	no	no	no

12. 类组成

> 1)属性
> 2)方法
> 3)构造方法
> 4)内部类
> 5)代码块 静态代码块、非静态代码块

13. 类的分类
    普通类 内部类

14. 类加载流程

15. static关键字

> static静态关键字，static方法就是没有this的方法。在static方法内部不能调用非静态方法，反过来是可以的。而且可以在没有创建任何对象的前提下，仅仅通过类本身来调用static方法。这实际上正是static方法的主要用途

16. final关键字

> 修饰类，属性，方法

17. volitate关键字

> 参考java多线程中的volitate讲解

18. native关键字

> native 就是一个java调用非java代码的接口。

19. super关键字怎么使用,在项目中哪里使用了？为什么要使用

> 1)super在构造方法中表示对父类构造方法的调用，要放在第一句。(这是super的主要应用)
> 2)方法，那么在执行的时候就会寻找与super()所对应的构造方法而不会在去寻找父类不带参数的构造方法。.如果子类使用super显示调用父类的某个构造
> 3)super在方法中表示对父类方法的调用，位置没有要求

20. this关键字怎么使用？在项目中哪里使用了？为什么要使用

> this关键字是在类被创建时候，才给这个类给的一个特殊的地址，其实这个this的地址也是指向这个类的
> 在类中使用this关键字主要是为了解决，局部变量影藏成员变量的问题，出现这个问题的原因是因为，JVM在寻找相同类型的变量的时候，他有一个就近原则
> this谁调用他，他就代表的谁

21. final、finally、finalize区别，怎么使用

> final java关键字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，修饰属性表示该属性不能被多次赋值
> finally java中异常处理使用到的关键字，通常和try..catch配合使用，无论结果如何都会执行finally代码块中的任务
> finalize Object类中的方法，表示一个对象可以被垃圾回收器回收了

22. 局部变量和成员变量区别

> 1)位置不同，局部变量在方法中或者在代码块中，成员变量在类中
> 2)局部变量不会有线程安全问题，成员变量会出现线程安全问题
> 3)局部变量必须初始化才能使用，成员变量JVM会默认初始化

23. 值传递和引用传递区别

> 引用也是地址指，所以在java中全部是指传递

24. 创建对象在JVM中的位置

> new 创建出来的对象在堆中
> 常量在方法区

25. ==和equals区别

> 1)基本数据类型==表示比较值，基本数据类型没有equals方法
> 2)引用数据类型==表示比较的地址值，equals比较的是值
> 3)字符串常量"a" == "a" 比较的也是地址值

26. 重载和重写区别

> 重写发生在类与类或者类与接口之间，方法名方法参数列表都要相同
> 重载发生在统一个类中，方法的参数列表不同，方法名必须相同

27. java基本数据类型

> java数据类型分为2中，基本数据类型和引用数据类型

> 基本数据类型分为4类8中
> 整数类型 byte、short、int、long
> 浮点类型 float、double
> 字符类型 char
> 布尔类型 boolean

28. 包装类和基本类区别

> java语言虽然号称 一切都是对象，但是基本数据类型是个例外，基本数据类型只提供了数据的运算逻辑判断
> 功能，儿包装类提供基本数据类型和字符串之间的转换，在JDK1.5中，引入了自动装箱和自动拆箱，java可以
> 根据上下文，自动的进行转换，极大简化了编程复杂性

29. 包装类和基础类怎么转换

> 包装类转基本数据类型 parse
> 基础类装包装类 new

30. 构造器代码块、局部代码块、静态代码块执行顺序和执行次数

> 执行顺序 静态代码块 构造代码块 局部代码块
> 执行次数 静态代码块只执行一次 构造代码块在每次创建都会执行，局部代码块，调用一次执行一次

31. 构造代码块的作用

> 用来创建对象

32. Integer是否可以被继承？为什么？

> 被final修饰的，保证了类基本信息的安全和在并发编程中线程安全

33. Integer缓存区间？什么时候触发缓存区间

> Integer缓存区间是-128--127

```
1 public static Integer valueOf(int i) {
2 if (i >= IntegerCache.low && i <= IntegerCache.high)
3     return IntegerCache.cache[i + (-IntegerCache.low)];
4 return new Integer(i);
5 }
```

我们看到触发valueOf方法的时候会先从缓存区中查找，如果没有在创建对象，valueOf是在自动装箱的时候触发
https://github.com/wolvesleader/interview/tree/master/base_java/src/main/java/com/quincy/java/base/integer

34. "" 和 new String("")区别

> ""创建一个对象
> new String("") 创建2个对象

35. String、StringBuffer、StringBuilder区别，是否线程安全？怎么做到线程安全？

> String字符串类，字符串拼接不建议使用，会生成新的字符串
> StringBuffer线程安全的，主要是在每个方法上都添加了同步锁

> StringBuilder线程不安全的

36. 包装类型、Math类常用的方法有哪些？在项目中怎么使用?

> 包装类常用方法parse 项目中后台接受到的前台传入的分页数据是字符串类型，可以使用parse转换为基本数据类型
> Math常用的方法random，在项目用来生成随机的验证码

37. hashCode和equals必须要重写吗?不重写会有什么问题？在什么情况下会出问题?

> 一般情况下不需要重写，如果有类与类之间的比较，必须要重写这2个方法
> 如果要把对象作为hashmap的健的情况也必须重写这2个方法，因为会对健做hash运算，如果不重写会出现健覆盖

38. &和&& | || 区别是什么

> &逻辑与运算，不会出现短路效果 && 并且关系会出现短路效果
> |逻辑或运算，不会出现短路效果 || 或关系会出现短路效果

39. JDK1.8之后有哪些新特性，在项目中哪里用到了

> 1)lambda表达式，在项目中写匿名类的时候使用的比较多
> 2)引入Optional,判断对象是否为null
> 3)新增Stream流

40. java创建对象方式有哪些？分别说出使用场景

> 1)new 关键字创建 常用的都是这种方法
> 2)调用对象的clone()方法
> 3)利用反射，调用Class类的或者是Constructor类的newInstance()方法 反射方式
> 4)用反序列化，调用ObjectInputStream类的readObject()方法 序列化方式

41. java中深克隆和浅克隆的区别？怎么实现

> 浅克隆：创建一个新对象，新对象的属性和原来对象完全相同，对于非基本类型属性，仍指向原有属性所指向的对象的内存地址。
> 深克隆：创建一个新对象，属性中引用的其他对象也会被克隆，不再指向原有对象地址

参考代码
https://github.com/wolvesleader/interview/tree/master/base_java/src/main/java/com/quincy/java/base/clone

42. Comparable 和Comparator的区别，分别说出使用场景

> Comparable在java.lang包里边
> Comparator在java.util包里边
> Comparable是内在的排序
> Comparator是外在的排序
> 实现comparator需要覆盖compare方法
> compareTo()方法比较是顺序的比较，而不是同等性的比较。
> equals()是等同性的比较。

参考代码
https://github.com/wolvesleader/interview/tree/master/base_java/src/main/java/com/quincy/java/bas

e/comparator

43. 是否看过java源码？分析自己看源码的底层实现
    看过，arrsylist、hashmap源码

44. Object类和范型的区别,说出范型的执行机制，为什么要使用范型

> Object是类，范型是一种约束机制，确保在编译时不会出现类型错误

45. 循环遍历种类分别说出使用场景

> for 明确知道循环次数
> while 不知道循环次数
> do...while 先执行一次，在根据条件循环

46. 枚举类型怎么实现，使用场景

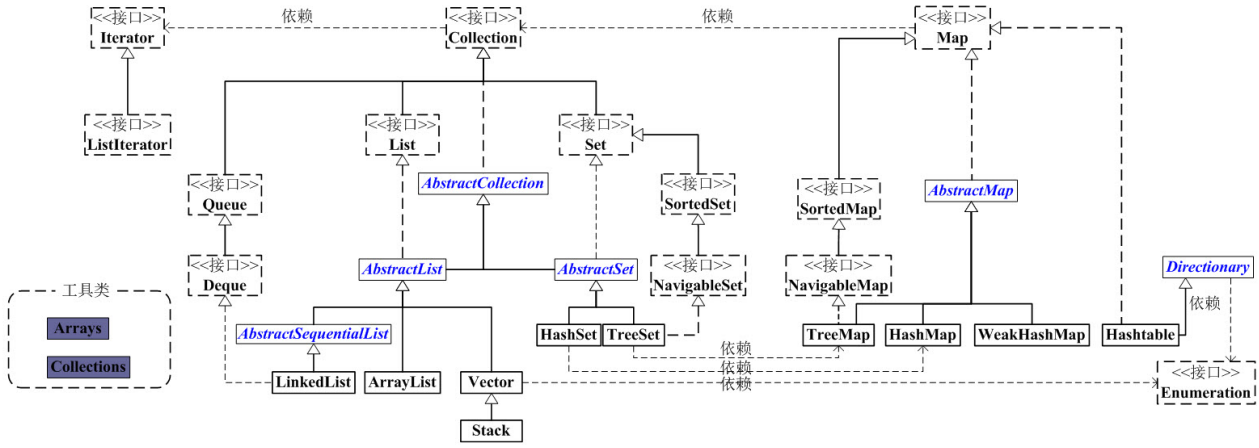
> 枚举是Java1.5引入的新特性，通过关键字enum来定义枚举类。枚举类是一种特殊类，它和普通类一样可以使用构造器、定义成员变量和方法，也能实现一个或多个接口,但枚举类不能继承其他类
> 枚举类的作用是约束程序员使用固定的变量参数
> 参考代码
> https://github.com/wolvesleader/interview/tree/master/base_java/src/main/java/com/quincy/java/base/enumdemo

47. int和Integer哪个占用内存大一些？为什么？

48. 后续.....

Java 集合框架

1. 集合框架 架构



2. 常用集合有哪些

> arraylist、hashmap、concurrenthashmap、队列

3. arraylist集合特点、底层实现原理

> arraylist集合底层是用数组来实现的，它有数组全部的特点，查询块，增加和删除慢

底层的实现原理可以查看源码,我们从add方法开始看起，看它是怎么存储数据的

```
1  public boolean add(E e) {
2      //暂时不重要可以忽略
```

```
    ensureCapacityInternal(size + 1);  // Increments modCount!!
    //可以看到把我们传入的元素放入到elementData数组中
    elementData[size++] = e;
    return true;
}
```

到这里我们应该就明白了，arraylist就是把我们添加的数据存入到一个叫elementData的数组中，相应的get方法的时候就是从这个数组中获取到数据

4. arraylist扩容原理

> 上边我们通过源码开大arraylist把数据存入到elementData数组中，那我们就想，如果数组满了怎么办是不是要扩容

底层的实现原理可以查看源码,我们从add方法开始看起，看它是怎么扩容的

```
public boolean add(E e) {
    //我们要关注的方法
    ensureCapacityInternal(size + 1);  // Increments modCount!!
    //忽略不重要
    elementData[size++] = e;
    return true;
}
private void ensureCapacityInternal(int minCapacity) {
     //很重要
     //判断elementData是否为{}
    if (elementData == DEFAULTCAPACITY_EMPTY_ELEMENTDATA) {
        //计算数组的长度，容量
        //minCapacity是我们在创建数组的时候在构造期中传入的参数
        //如果没有传入参数默认使用的是DEFAULT_CAPACITY长度
        //查看源码可是该长度是10
        minCapacity = Math.max(DEFAULT_CAPACITY, minCapacity);
    }
    //分析上边的条件判断之后我们知道arraylist默认的容量是10
    //如果我们在创建arraylist指定的时候长度就是我们指定的
    ensureExplicitCapacity(minCapacity);
}
private void ensureExplicitCapacity(int minCapacity) {
    //忽略不看
    modCount++;
    // overflow-conscious code
    //很重要需要看
    //使用计算出来的minCapacity容量和数组的长度来比较
    //如果大于0说明容量和长度不匹配，需要扩容
    if (minCapacity - elementData.length > 0)
        grow(minCapacity);
}
private void grow(int minCapacity) {
    // overflow-conscious code
    //愿数组的长度
    int oldCapacity = elementData.length;
    以1.5被扩容之后的数组长度
    int newCapacity = oldCapacity + (oldCapacity >> 1);
    //校验扩容之后数组长度，最后结果就是得到一个扩容之后的数组长度
    if (newCapacity - minCapacity < 0)
        newCapacity = minCapacity;
    if (newCapacity - MAX_ARRAY_SIZE > 0)
```

```
42          newCapacity = hugeCapacity(minCapacity);
43      // minCapacity is usually close to size, so this is a win:
44      //很重要
45      elementData = Arrays.copyOf(elementData, newCapacity);
46  }
47  public static <T> T[] copyOf(T[] original, int newLength) {
48      return (T[]) copyOf(original, newLength, original.getClass());
49  }
50  public static <T,U> T[] copyOf(U[] original, int newLength,
51                                 Class<? extends T[]> newType) {
52      @SuppressWarnings("unchecked")
53      //创建新数组，通过计算出来的容量
54      T[] copy = ((Object)newType == (Object)Object[].class)
55          ? (T[]) new Object[newLength]
56          : (T[]) Array.newInstance(newType.getComponentType(), newLength);
57      //很重要，创建出来的新数组是空的，我们需要把之前数组中的数据copy到新的数组中
58      System.arraycopy(original, 0, copy, 0,
59                       Math.min(original.length, newLength));
60      return copy;
61  }
```

到此扩容原理分析完成
默认数组长度是10，如果指定就使用指定的长度，每次扩容都以原数组长度的1.5倍来扩容，扩容之后通过System.arraycopy方法把原来数组中的数据copy到新的数组中

5. arraylist默认长度

> 通过上边的源码分析我们知道默认的数组长度是10，如果创建数组的时候我们通过构造方法传入长度就是默认长度就是我们指定的长度

6. arraylist为什么能不断的存入数据

> 通过上边的源码分析，我们知道arraylist能不断存入数据的原因是，如果原来的数组满了，会不断的以原数组1.5倍的长度来创建新数组也就是扩容，所以能不断的存入数据

7. arraylist集合是否线程安全？怎么做到线程安全

> arraylist集合是线程不安全的，要让它安全可以使用Collections.synchronizedList方法获取一个线程安全的集合

```
1  ArrayList<String> arrayList = new ArrayList<>();
2  arrayList.add("888");
3  List<String> list = Collections.synchronizedList(arrayList);
```

Collections.synchronizedList是怎么做到线程安全的，是对原集合的方法添加了同步锁

> 另外一种方式是使用concurrent包下提供的线程安全的集合CopyOnWriteArrayList

8. 多线程环境下arraylist存在什么问题？怎么解决

> 多线程环境下遍历arraylist会出现并发修改异常 ConcurrentModificationException
> 解决方案：
> 1)Collections.synchronizedList(new ArrayList<>())
> 2)new CopyOnWriteArrayList<>()

参考代码
https://github.com/wolvesleader/interview/tree/master/base_java/src/main/java/com/quincy/java/bas

e/manythreadarraylist

9. 项目中怎么选择使用那种集合，在项目中的用途

> 这个很重要选择的时候我们一定要根据我们的数据结构，和集合的数据结构来选择

10. 模拟arraylist实现

11. HashMap集合特点、底层实现原理

> 在JDK1.8中HashMap的实现是通过(数组+链表+二叉树)来实现的
> 首先存入数据调用put方法的时候会通过hash方法对键做一次hash操作，最后把原键的hash值和原键hash值无符号右移16位的值按位异或的结果，作为最后返回的hash值，这么做的原因是为了取得的hash值
> 更加的均匀减少hash冲突的次数，在调用putVal存储我们的值，判断Node节点的数组是否创建，如果没有创建，我们也没有设置HashMap的initialCapacity初始化容量和负载因子loadFactor，则创建默认的容量为16，负载因子为0.75的数组，接下来分为2中情况
> 1.通过返回的hash值按位异或我们创建的数组的长度减1的做过作为元素在数组中的坐标，如果该位置没有存放元素，我们直接创建Node节点存储数据
> 2.如果数组索引的位置存在元素又分为三中情况
> (1)键的hash值和我们要存入的数据的键的hash值相同，并且值也相同，说明我们存入的数据重复，这种情况下，会把新的值覆盖原来的值
> (2)判断获取到的元素是否为TreeNode结构，如果是挂载到TreeNode中
> (3)遍历table数组，找出尾节点，把我们的数据挂载到为节点中

12. Collectios和Collection的区别

> Collectios集合的帮助类
> Collection单列集合的一个接口

13. JDK1.7之前、JDK1.7、JDK1.7之后HashMap实现的不同

> JDK1.7之前数据结构数组+链表
> JDK1.7之后数组+链表+红黑树的结构

14. HashMap数据结构

> 数组+链表+红黑树的结构

15. 为什么使用数组+链表+红黑树的结构

> 数组+链表 解决hash冲突
> 红黑树链表长度太长，查询时间较长

16. HashMap为什么要对key多次做hashcode运算

> 取得的hash值更加的均匀减少hash冲突的次数

17. HashMap为什么要以2的幂次方扩容

> 为什么默认长度是16，以及为什么每次扩容是2的幂次方
> (1)好处1:获取元素存储位置的索引是通过(n - 1) & hash来获取的
> 注意最后return的是h&(n-1)。如果n不为2的幂，比如15。那么length-1的2进制就会变成1110。在h为随机数的情况下，和1110做&操作。尾数永远为0。那么0001、1001、1101等尾数为1的位置就永远不可能被entry占用。这样会造成浪费，不随机等问题。 n-1 二进制中为1的位数越多，那么分布就平均

> (2)好处2:扩容之后不需要重新通过hash计算元素的位置,举个例子来说明以下
> 原来的hash值是5，看以下计算过程

```
   0000 0000 0000 0000 0000 0000 0000 1111 //(n-1)=15二进制表示
&  0000 0000 0000 0000 0000 0000 0000 0101 //hash值5二进制表示
_____________________________________________________________________
   0000 0000 0000 0000 0000 0000 0000 0101

 //看扩容之后的计算(扩容到32)
   0000 0000 0000 0000 0000 0000 0001 1111 //(32-1)=31二进制表示
&  0000 0000 0000 0000 0000 0000 0000 0101 //hash值5二进制表示
_____________________________________________________________________
   0000 0000 0000 0000 0000 0000 0001 0101
```

扩容之后的位置=原位置(5) + oldCap(原集合的容量) = 21
原来的hash值是14

```
   0000 0000 0000 0000 0000 0000 0000 1111 //(n-1)=15二进制表示
&  0000 0000 0000 0000 0000 0000 0000 1110 //hash值14二进制表示
_____________________________________________________________________
   0000 0000 0000 0000 0000 0000 0000 1110

 //看扩容之后的计算(扩容到32)
   0000 0000 0000 0000 0000 0000 0001 1111 //(32-1)=31二进制表示
&  0000 0000 0000 0000 0000 0000 0000 1110 //hash值5二进制表示
_____________________________________________________________________
   0000 0000 0000 0000 0000 0000 0000 1110
```

扩容之后的位置=原位置(14)

18. 集合的遍历方式有哪些，各种方式的区别

    > for foreach 迭代器

19. 多线程环境下HashMap存在什么问题，怎么解决

    > 会出现数据丢失，死循环等问题，解决办法是使用线程安全的集合

20. 集合使用优化

    > 创建集合指定集合的长度

21. 线程安全的集合有哪些？为什么是线程安全的?

    > Vector,Hashtable,在方法上加锁

22. TreeMap、Hashtable底层实现原理

23. HashSet、TreeSet、LinkedHashSet底层实现原理

24. LinkedList、Stack、Vector底层实现原理

25. 集合中为什么不能存入基本数据类型

26. concurrent包中ConcurrentHashMap、CopyOnWriteArrayList、ArrayBlockingQueue等等特点和底层实现原理

27. 怎么做对象的比较

> Comparable 或者Comparator
> 参考代码
> https://github.com/wolvesleader/interview/blob/master/base_java/src/main/java/com/quincy/java/base/comparator

28. 怎么做对象的比较

> Comparable 或者Comparator
> 参考代码
> https://github.com/wolvesleader/interview/blob/master/base_java/src/main/java/com/quincy/java/base/comparator

29. 并发修改异常是怎么产生的？怎么解决

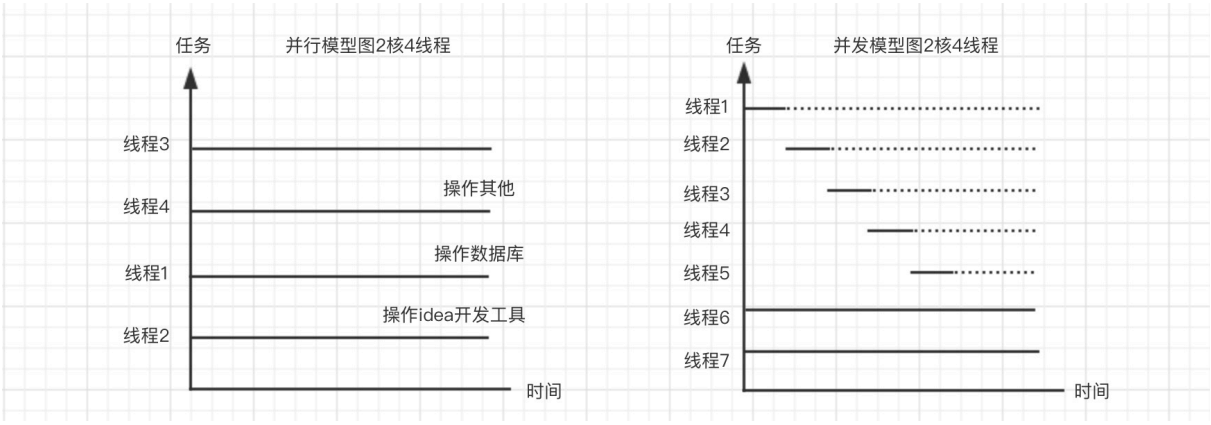
30. Collectios有哪些方法，什么作用，底层怎么实现

Java 线程

1. 线程和进程区别

> 线程是进程中的一条任务
> 进程是是运行在操作系统上的一个应用程序
> 例如：打开微信聊天工具，就是开启一个进程
> 在微信中和你其中的一个朋友聊天就会开启一个线程

2. 并发和并行区别

> 
>
> 并行
> 程序同时所开启的运行中的线程数 <= CPU数量*CPU的核心数*
> *程序同时所开启的运行中的线程数* <= *CPU数量*CPU的线程数量
> 并发
> 程序同时所开启的运行中的线程数 > CPU数量*CPU的核心数*
> *程序同时所开启的运行中的线程数* > *CPU数量*CPU的线程数量
> 建议大家先去看一下cpu4核8线程，2核4线程 到底代表的是什么

3. 线程创建方式有几种,不同方式的区别，开发中使用过那种

> 1)继承Thread类
> 2)实现Runnable接口
> 3)实现Callable接口
> 4)线程池中获取
> 4中方式的具体实现，请参考
> https://github.com/wolvesleader/interview/tree/master/base_java/src/main/java/com/quincy/jav

> a/base/threadcreate
> 提供的代码实现
> 区别如下
> java中是单继承，如果考虑要继承其他类，所以不用继承Thread的方式
> 如果不需要获取到线程执行的结果值，不使用实现Callable接口
> 一般在项目中都是用实现Runnable接口的方式来创建线程
> 好处是松耦合，面向接口编程，扩展性强

4. 线程是否创建越多越好？为什么

> 线程创建我们应该考虑下边的问题
> 线程创建需要占用内存
> 创建之间存在切换，需要耗费时间
> 因此创建线程不能过多，原因就是上边的2点

5. 线程怎么启动,线程启动原理

> 线程的启动是通过start(),启动源码如下所示

```
1  public synchronized void start() {
2      if (threadStatus != 0)
3          throw new IllegalThreadStateException();
4      group.add(this);
5      boolean started = false;
6      try {
7          //线程的启动
8          start0();
9          started = true;
10     } finally {
11         try {
12             if (!started) {
13                 group.threadStartFailed(this);
14             }
15         } catch (Throwable ignore) {
16         }
17     }
18 }
19
```

```
1 private native void start0();
```

native方法在JVM虚拟机中jvm.cpp中的JVM_StartThread中实现

```
1 JVM_ENTRY(void, JVM_StartThread(JNIEnv* env, jobject jthread))
2 JVMWrapper("JVM_StartThread");
3 JavaThread *native_thread = NULL;
4 bool throw_illegal_thread_state = false;
5 {
6      MutexLocker mu(Threads_lock);
7      //判断线程的状态，如果线程状态是启动，会抛出异常
8      if (java_lang_Thread::thread(JNIHandles::resolve_non_null(jthread)) != NULL) {
9        throw_illegal_thread_state = true;
10     } else {
11         jlong size =
12              java_lang_Thread::stackSize(JNIHandles::resolve_non_null(jthread));
13         size_t sz = size > 0 ? (size_t) size : 0;
```

```
        //创建一个线程
        native_thread = new JavaThread(&thread_entry, sz);
        if (native_thread->osthread() != NULL) {
            native_thread->prepare(jthread);
        }
    }
}
//
Thread::start(native_thread);
JVM_END
```

```
JavaThread::JavaThread(ThreadFunction entry_point, size_t stack_sz) :
                    Thread(){
initialize();
_jni_attach_state = _not_attaching_via_jni;
set_entry_point(entry_point);
os::ThreadType thr_type = os::java_thread;
thr_type = entry_point == &compiler_thread_entry ? os::compiler_thread :
                                                   os::java_thread;
//创建线程
os::create_thread(this, thr_type, stack_sz);
}
```

```

bool os::create_thread(Thread* thread, ThreadType thr_type,
                    size_t req_stack_size) {
///////
pthread_t tid;
//创建线程
int ret = pthread_create(&tid, &attr, (void* (*)(void*)) thread_native_entry, thread);
///////
return true;
}

```

```
static void *thread_native_entry(Thread *thread) {
  //关键方法，调用了run方法
  thread->run();
   ///////
  return 0;
}
```

到此线程的启动过程分析完成，总结

1.调用start方法，通过JNI调用到JVM

2.JVM通过pthread_create（）创建一个系统内核线程，并指定内核线程的初始运行地址

3.内核线程在调用java线程中的run方法，开始执行线程

6. 调用run方法会执行吗？会启动线程吗？为什么？

> run方法是Thread类中的一个方法，如果我们创建了线程类，通过类的实例调用方法当然是可以的
> 不会启动线程
> 在上一题中我们分析了线程的启动原理
> 虽然线程最终的启动也是通过调用run方法，但是需要创建系统内核线程
> 在系统内核线程中调用run方法开会开启线程
> 如果直接调用Thread类的run方法，就相当于调用一个普通的方法

7. 在项目中创建多少线程比较合适，为什么？

> 创建多少线程比较合适呢？
> 我们一般需要根据业务场景来做具体分析
> CPU密集型计算
> 对与CPU密集型来说，主要是提高CPU的利用率
> 例如 4核8线程的CPU，理论上创建4个线程就ok了，在多创建线程只会增加线程切换的成本
> 但是我们需要考虑另外一个问题，如果其中的一条线程因为内存页失效或者其他原因阻塞的时候，相当与少了一条线程
> 所以我们在实际开发中
> CPU密集型计算创建的线程数量 = CPU核数 + 1
> 多余创建一个线程，有其他线程阻塞的时候，该线程可以替补上
> I/O密集型计算
> I/O密集型计算创建的线程数量 = 1 + (I/O耗时 / CPU耗时)
> 如果是多核CPU计算公式如下
> I/O密集型计算创建的线程数量 = CPU核数*[1 + (I/O耗时 / CPU耗时)]
> 做完一系列理论设置之后需要用压测工具来检验自己设置的线程数是否合理
> 备注：需要做I/O，CPU耗时测试，可以使用apm工具

8. 线程安全问题产生的本质原因是什么

> 1)缓存导致的可见性问题
> 2)线程切换带来的原子性问题
> 3)编译优化带来的有序性问题

9. 线程安全怎么解决
   主要的手段是加锁

10. volitale能保证单变量的原子性、可见性和有序性吗？为什么

> volitale只能保证单变量线程之间的可见性，不能保证原子性和有序性
> 添加volitale变量之后会在该字段前添加Lock前缀指令
> lock前缀的指令在多核处理器下会引发了两件事情
> 将当前处理器缓存行的数据会写回到系统内存
> 这个写回内存的操作会引起在其他CPU里缓存了该内存地址的数据无效

11. 什么是JMM

> 我们知道导致可见性的原因是缓存，导致有序性的原因是编译优化，那么解决可见性，有序性最直接的办法就是禁用缓存和编译优化
> 这样问题解决了，但是我们程序的性能就会降低
> 合理的方案是按需禁用缓存以及编译优化，怎么如何按需禁用对于并发程序，何时禁用缓存以及编译优化只有程序员知道
> 那么按需禁用就是按照程序员的要求禁用，所以，为了解决可见性和有序性，只需要提供给程序员按需禁用缓存和编译优化的方法即可
> JMM全称是Java内存模型，从程序员的角度考虑，java内存模型规范了JVM如何按需禁用缓存和编译优化的方法
> 这些方法主要包括
> volatile、synchronized、final三个关键字
> 以及happens-before规则

12. 线程怎么优雅的停止

stop(),suspend()方法都被废弃之后我们不能在使用他们停止线程
那我们想想，什么时候线程会停止，就是在run方法中的代码执行完之后
就会停止，那我们如果要人为的控制线程的停止，可以在run方法中做文章
还需要知道，线程的停止有可能是自己停止自己或者被别的线程停止
根据以上的分析我们分为2中情况
1).自己停止自己，我们可以在线程中设置一个boolean变量，人为控制变量

```
1   public void run(){
2     if(boolean){
3       //需要执行的任务
4     }
5   }
```

2).当前线程被其他线程停止
例如在线程A中停止线程B
在线程A中调用线程B的.interrupt();方法
interrupt()方法相当与给线程打上一个标记
该线程可以被停止了,打上标记之后
在run方法中

```
1   public void run(){
2     if(!Thread.currentThread().isInterrupted()){//判断线程是否被停止
3       //需要执行的任务
4     }
5   }
```

综合以上2中情况我们可以这样写

```
1   public void run(){
2     if(boolean && !Thread.currentThread().isInterrupted()){//判断线程是否被停止
3       //需要执行的任务
4     }
5   }
```

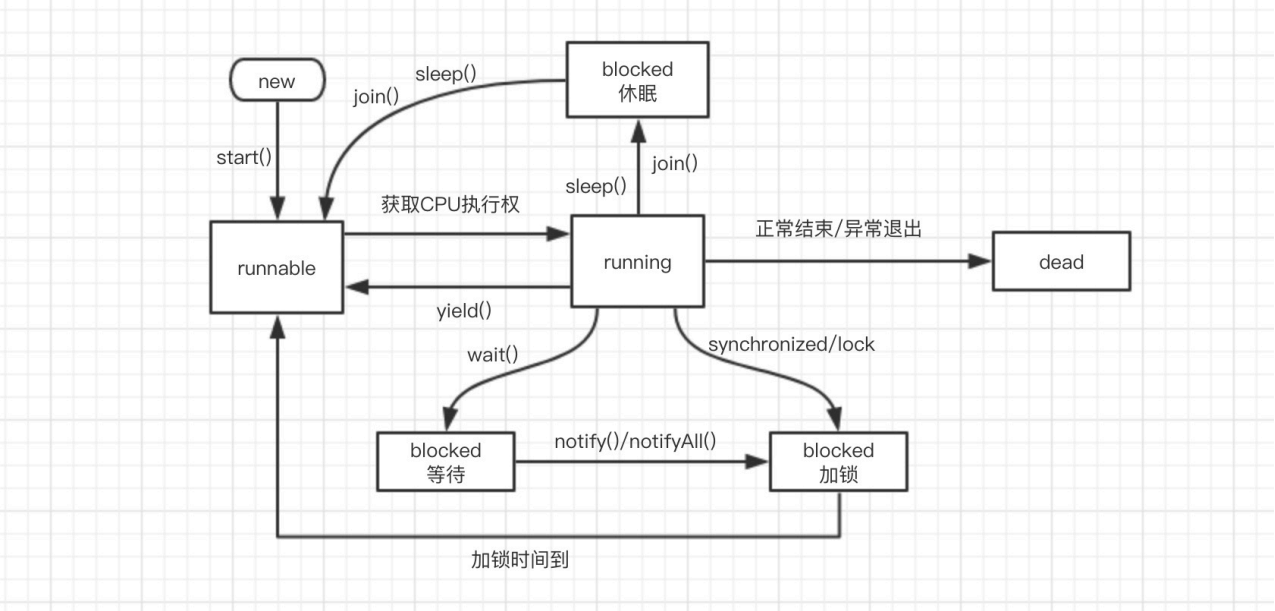
代码实现请参考
https://github.com/wolvesleader/interview/tree/master/base_java/src/main/java/com/quincy/java/base/threadstop

13. stop和suspend方法为什么被废弃，原因是什么

> stop停止是线程不安全的
> 因为它在终止一个线程时会强制中断线程的执行，不管run方法是否执行完了，并且还会释放这个线程所持有的所有的锁对象。这一现象会被其它因为请求锁而阻塞的线程看到，使他们继续向下执行。这就会造成数据的不一致，破坏了原子逻辑
> 运行测试代码可以看到调用stop方法之前线程A和线程B操作的userA和userB数据总和是不变的
> 在调用了stop方法之后数据总和会丢失一部分
> 测试代码
> https://github.com/wolvesleader/interview/tree/master/base_java/src/main/java/com/quincy/java/base/threadstop
> suspend被弃用的原因是因为它会造成死锁。suspend方法和stop方法不一样，它不会破换对象和强制释放锁，相反它会一直保持对锁的占有，一直到其他的线程调用resume方法，它才能继续向下执行。假如有A，B两个线程，A线程在获得某个锁之后被suspend阻塞，这时A不能继续执行，线程B在或者相同的锁之后才能调用resume方法将A唤醒，但是此时的锁被A占有，B不能继续执行，也就不能及时的唤醒A，此时A，B两个线程都不能继续向下执行而形成了死锁。这就是suspend被弃用的原因

14. 线程的生命周期有哪些阶段

线程生命周期图



15. sleep()和wait()方法的区别

> sleep 不会释放锁 会释放CPU的执行权 Thread中的方法
> wait 会释放锁 会释放CPU的执行权 Object类中的方法

16. 线程之间怎么通信

> 通过共享变量
> 通过CountDownLatch
> 通过CyclicBarrier
> 通过Exchanger
> 通过Join
> 通过Lock、Condition
> 通过Wait、Notify

具体代码实现请参考
https://github.com/wolvesleader/interview/tree/master/base_java/src/main/java/com/quincy/java/base/threadcommunication

17. 什么是AQS

> AQS 全程AbstractQueuedSynchronizer,类如其名，抽象队列同步器，AQS定义了一套多线程访问共享资源的同步器框架，许多同步类实现都依赖于它，如常用的ReentrantLock/Semaphore/CountDownLatch，AQS就是一个半成品框架，方便程序员实现锁

18. 结合ReentrantLock分析AQS的使用

非公平锁机制

```
1 public ReentrantLock() {
2     sync = new NonfairSync();
3 }
```

公平锁机制

```
1 public ReentrantLock(boolean fair) {
2 sync = fair ? new FairSync() : new NonfairSync();
```

```
}
```

非公平锁实现源码分析

```
static final class NonfairSync extends Sync {
    //.......
    /**
     * Performs lock.  Try immediate barge, backing up to normal
     * acquire on failure.
     */
    final void lock() {
        if (compareAndSetState(0, 1))
            setExclusiveOwnerThread(Thread.currentThread());
        else
            acquire(1);
    }
    protected final boolean tryAcquire(int acquires) {
        return nonfairTryAcquire(acquires);
    }
}
```

NonfairSync是ReentrantLock的一个内部类继承自Sync，我们在来看Sync的代码

```
abstract static class Sync extends AbstractQueuedSynchronizer {
    private static final long serialVersionUID = -5179523762034025860L;
    /**
     * Performs {@link Lock#lock}. The main reason for subclassing
     * is to allow fast path for nonfair version.
     */
    abstract void lock();
    /**
     * Performs non-fair tryLock.  tryAcquire is implemented in
     * subclasses, but both need nonfair try for trylock method.
     */
    final boolean nonfairTryAcquire(int acquires) {
        final Thread current = Thread.currentThread();
        int c = getState();
        if (c == 0) {
            if (compareAndSetState(0, acquires)) {
                setExclusiveOwnerThread(current);
                return true;
            }
        }
        else if (current == getExclusiveOwnerThread()) {
            int nextc = c + acquires;
            if (nextc < 0) // overflow
                throw new Error("Maximum lock count exceeded");
            setState(nextc);
            return true;
        }
        return false;
    }

    protected final boolean tryRelease(int releases) {
        int c = getState() - releases;
        if (Thread.currentThread() != getExclusiveOwnerThread())
            throw new IllegalMonitorStateException();
```

```
        boolean free = false;
        if (c == 0) {
            free = true;
            setExclusiveOwnerThread(null);
        }
        setState(c);
        return free;
    }

    protected final boolean isHeldExclusively() {
        // While we must in general read state before owner,
        // we don't need to do so to check if current thread is owner
        return getExclusiveOwnerThread() == Thread.currentThread();
    }

    final ConditionObject newCondition() {
        return new ConditionObject();
    }

    // Methods relayed from outer class

    final Thread getOwner() {
        return getState() == 0 ? null : getExclusiveOwnerThread();
    }

    final int getHoldCount() {
        return isHeldExclusively() ? getState() : 0;
    }

    final boolean isLocked() {
        return getState() != 0;
    }

    /**
     * Reconstitutes the instance from a stream (that is, deserializes it).
     */
    private void readObject(java.io.ObjectInputStream s)
        throws java.io.IOException, ClassNotFoundException {
        s.defaultReadObject();
        setState(0); // reset to unlocked state
    }
}
```

Sync是ReentrantLock的一个抽象内部类，继承自AbstractQueuedSynchronizer，到此类与类之间的关系基本清楚，我们分析加锁和释放锁的过程
ReentrantLock中的lock方法如下所示

```
public void lock() {
    sync.lock();
}
```

Sync类中的lock是一个抽象方法，我们看他实现类NonfairSync中的方法，如下所示

```
final void lock() {
    if (compareAndSetState(0, 1))
        setExclusiveOwnerThread(Thread.currentThread());
    else
```

```
5         acquire(1);
6 }
```

先来看看上边的方法compareAndSetState(0, 1)，该方法是通过CAS算法，把锁状态设置为1
我们接着看acquire(1);方法
在java.util.concurrent.locks.AbstractQueuedSynchronizer类中
代码段1

```
1 public final void acquire(int arg) {
2     if (!tryAcquire(arg) &&
3         acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
4         selfInterrupt();
5 }
```

先看看第一个条件判断tryAcquire(arg)
在java.util.concurrent.locks.AbstractQueuedSynchronizer类中

```
1 protected boolean tryAcquire(int arg) {
2     throw new UnsupportedOperationException();
3 }
```

发现只是抛出了异常没有其他逻辑代码，这时候我们可以想想NonfairSync继承Sync,Sync继承自AbstractQueuedSynchronizer，NonfairSync类中提供了tryAcquire(int arg)方法，
相当于对AbstractQueuedSynchronizer类中的方法进行了重写，那我们只需要查看NonfairSync类中的tryAcquire(int arg)方法，代码如下所示

```
1 protected final boolean tryAcquire(int acquires) {
2     return nonfairTryAcquire(acquires);
3 }
```

nonfairTryAcquire(acquires)方法调用了父类Sync中的方法代码如下所示

```
1 final boolean nonfairTryAcquire(int acquires) {
2     final Thread current = Thread.currentThread();
3     int c = getState();
4     if (c == 0) {
5         //不判断是否有等待队列,直接进行占用,如果占用失败也进到等待队列尾
6         //这个也是公平锁和非公平锁的区别所在
7         if (compareAndSetState(0, acquires)) {
8             setExclusiveOwnerThread(current);
9             return true;
10        }
11    }
12    else if (current == getExclusiveOwnerThread()) {
13        int nextc = c + acquires;
14        if (nextc < 0) // overflow
15            throw new Error("Maximum lock count exceeded");
16        setState(nextc);
17        return true;
18    }
19    return false;
20 }
```

终于到核心代码了，我们一行一行来看
final Thread current = Thread.currentThread();
获取当前的线程，目的是实现重入锁

int c = getState();
获取AbstractQueuedSynchronizer中的当前线程的状态
接着看if....if else条件判断
如果当前线程的状态为0，表示该线程第一次获取到锁，compareAndSetState(0, acquires)
通过CAS把锁的状态改变为1，返回true，获取锁成功
else if 表示当前线程之前以及获取过锁，而且没有释放锁，此中情况下把锁的状态值加1，再次设置给state，返回true，表示没有释放锁的线程获取锁成功
如果以上情况都不成立返回false，表示获取锁失败

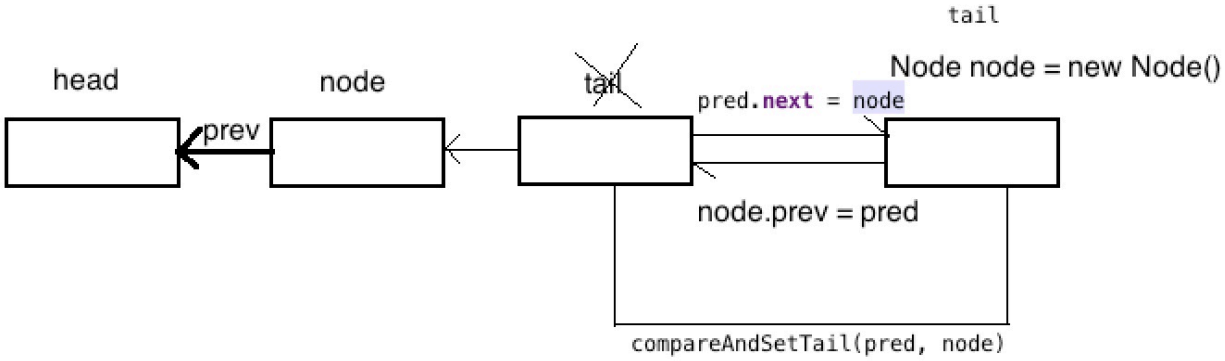
我们接着回到代码段1
!tryAcquire(arg) 经过上边的分析如果tryAcquire(arg)返会true表示获取锁成功
如果返回false表示获取锁失败，!tryAcquire(arg)之后变为了true，我么看获取不到锁之后的处理逻辑
执行了acquireQueued(addWaiter(Node.EXCLUSIVE), arg)代码
先来看addWaiter(Node.EXCLUSIVE)代码

```
1 private Node addWaiter(Node mode) {
2     Node node = new Node(Thread.currentThread(), mode);
3     // Try the fast path of enq; backup to full enq on failure
4     Node pred = tail;
5     if (pred != null) {
6         node.prev = pred;
7         if (compareAndSetTail(pred, node)) {
8             pred.next = node;
9             return node;
10        }
11    }
12    enq(node);
13    return node;
14 }
```

这段代码的主要作用是把没有抢到锁的线程放入到一个队列中，我么来分析一下这个过程
Node node = new Node(Thread.currentThread(), mode);
说明当前的一个节点就代表一个线程，mode默认是独享模式
Node pred = tail;
定义一个node节点把尾节点先保存起来

```
1 if (pred != null) {
2    node.prev = pred;
3    if (compareAndSetTail(pred, node)) {
4        pred.next = node;
5        return node;
6    }
7 }
```

如果尾节点不为空，我们来分析一下插入数据的过程



如果尾节点为空，说明还没有初始化该队列，执行enq()方法，我们来看过程

```
1 private Node enq(final Node node) {
2     for (;;) {
3         Node t = tail;
4         if (t == null) { // Must initialize
5             if (compareAndSetHead(new Node()))
6                 tail = head;
7         } else {
8             node.prev = t;
9             if (compareAndSetTail(t, node)) {
10                t.next = node;
11                return t;
12            }
13        }
14    }
15 }
```

有了上边的分析看这个应该比较简单，在这里就不在做多余的描述
到此获取锁的整个过程完结，我们接下来分析释放所得过程
java.util.concurrent.locks.ReentrantLock中释放锁的代码如下

```
1 public void unlock() {
2     sync.release(1);
3 }
```

在java.util.concurrent.locks.AbstractQueuedSynchronizer中找到了release(1)方法

```
1 public final boolean release(int arg) {
2     if (tryRelease(arg)) {
3         Node h = head;
4         if (h != null && h.waitStatus != 0)
5             unparkSuccessor(h);
6         return true;
7     }
8     return false;
9 }
```

java.util.concurrent.locks.ReentrantLock.Sync中重写了tryRelease(arg)方法

```
1 protected final boolean tryRelease(int releases) {
2     int c = getState() - releases;
3     if (Thread.currentThread() != getExclusiveOwnerThread())
4         throw new IllegalMonitorStateException();
5     boolean free = false;
6     if (c == 0) {
7         free = true;
```

```
        setExclusiveOwnerThread(null);
    }
    setState(c);
    return free;
}
```

这段代码没有什么其他的主要是设置线程的状态
unparkSuccessor(h);//作用是唤醒后续的节点，来看看具体是怎么唤醒的

```
private void unparkSuccessor(Node node) {
    /*
     * If status is negative (i.e., possibly needing signal) try
     * to clear in anticipation of signalling.  It is OK if this
     * fails or if status is changed by waiting thread.
     */
    //当前节点线程的状态,如果小于 0 ，设置为0
    int ws = node.waitStatus;
    if (ws < 0)
        compareAndSetWaitStatus(node, ws, 0);

    /*
     * Thread to unpark is held in successor, which is normally
     * just the next node.  But if cancelled or apparently null,
     * traverse backwards from tail to find the actual
     * non-cancelled successor.
     */
    //当前节点的下一个节点
    Node s = node.next;
    //后继节点为null或者其状态 > 0（超时或者被中断了）
    if (s == null || s.waitStatus > 0) {
        s = null;
        for (Node t = tail; t != null && t != node; t = t.prev)
            if (t.waitStatus <= 0)
                s = t;
    }
    if (s != null)
        LockSupport.unpark(s.thread);
}
```

重要看这段代码

```
for (Node t = tail; t != null && t != node; t = t.prev)
    if (t.waitStatus <= 0)
        s = t;
```

这段代码是采用了回溯法获取到需要唤醒的线程节点
接着看怎么唤醒线程
LockSupport.unpark(s.thread);
java.util.concurrent.locks.LockSupport类中的方法如下

```
public static void unpark(Thread thread) {
    if (thread != null)
        UNSAFE.unpark(thread);
}
```

```
public native void unpark(Object var1);
```

看到了native方法
我们是有去jvm虚拟机的代码中查看
查看网友找到的代码如下所示

```
void Parker::unpark() {
//定义两个变量，staus用于判断是否获取锁
  int s, status ;
  //获取锁
  status = os::Solaris::mutex_lock (_mutex) ;
  //判断是否成功
  assert (status == 0, "invariant") ;
  //存储原先变量_counter
  s = _counter;
  //把_counter设为1
  _counter = 1;
  //释放锁
  status = os::Solaris::mutex_unlock (_mutex) ;
  assert (status == 0, "invariant") ;
  if (s < 1) {
  //如果原先_counter信号量小于1，即为0，则进行signal操作，唤醒操作
    status = os::Solaris::cond_signal (_cond) ;
    assert (status == 0, "invariant") ;
  }
}
```

到此使用AQS实现非公平锁的加锁，解锁过程完毕

公平锁的实现和非公平锁基本相同，只是判断不判断有等待队列存在不相同
调用加锁方法之后非公平锁的实现代码如下所示

```
final boolean nonfairTryAcquire(int acquires) {
    final Thread current = Thread.currentThread();
    int c = getState();
    if (c == 0) {
        if (compareAndSetState(0, acquires)) {
            setExclusiveOwnerThread(current);
            return true;
        }
    }
    else if (current == getExclusiveOwnerThread()) {
        int nextc = c + acquires;
        if (nextc < 0) // overflow
            throw new Error("Maximum lock count exceeded");
        setState(nextc);
        return true;
    }
    return false;
}
```

公平锁的实现如下所示

```
protected final boolean tryAcquire(int acquires) {
    final Thread current = Thread.currentThread();
    int c = getState();
    if (c == 0) {
        if (!hasQueuedPredecessors() &&
            compareAndSetState(0, acquires)) {
```

```
            setExclusiveOwnerThread(current);
            return true;
        }
    }
    else if (current == getExclusiveOwnerThread()) {
        int nextc = c + acquires;
        if (nextc < 0)
            throw new Error("Maximum lock count exceeded");
        setState(nextc);
        return true;
    }
    return false;
}
```

19. java中有哪些锁，怎么使用，区别是什么

> 偏向锁
> 轻量级锁
> 重量级锁
> 读写锁
> 重入锁
> 共享锁
> 独占锁
> 自旋锁
> 公平锁
> 非公平锁
> 死锁
> 活锁

20. java中的死锁是怎么产生的，怎么定位死锁，怎么解决死锁

> 死锁是在2个以上线程中，相互等待对方释放自己运行需要的锁
> 定位死锁可以通过jconsole工具
> 死锁在一定程度上是无法完全解决的，我们只能避免出现死锁
> 常用的方法
> 1).调整加锁顺序
> 2).使用Lock锁的实现ReentrantLock

代码实现请参考
https://github.com/wolvesleader/interview/tree/master/base_java/src/main/java/com/quincy/java/base/deadlock

21. java中的乐观锁和悲观锁分别是哪些

> 1)通过版本控制
> 2)CAS Compare-and-Swap

22. CountDownLatch、CyclicBarrier、Exchanger、Semaphore、ReentrantLock有什么作用，实现原理是什么
CountDownLatch是在java1.5被引入的，跟它一起被引入的并发工具类还有CyclicBarrier、Semaphore、ConcurrentHashMap和BlockingQueue，它们都存在于java.util.concurrent包下。CountDownLatch这个类能够使一个线程等待其他线程完成各自的工作后再执行
CyclicBarrier
Exchanger

Semaphore

23. fork/join怎么使用

> 分而治之的思想，把大任务分解为个个小的任务执行，完成之后在进行汇总，具体参考代码实现

https://github.com/wolvesleader/interview/tree/master/base_java/src/main/java/com/quincy/java/base/forkjoin

24. 实现生产消费模式

25. 项目中支持最大并发是多少
在给面试官讲并发量的时候一定要加一个单位时间，也就是说单位时间内并发量是多少
离开了单位时间来说并发量是没有任何意义的扯淡

> 并发量的影响因素有很多，主要的有以下因素
> 程序执行时间
> CPU的时间片轮换时间
> 程序占用内存大小
> 宽带大小
> 请求数据库的时间 等等
> 给一个理想环境下并发量计算公式
> 不考虑任何客观因素
> 2个物理CPU 每个CPU8核16线程
> 那么1s极限并发 = 1000ms*16(CPU线程数量)*2 / (CPU切片轮换时间假设10ms + 程序执行时间假设10ms) = 1600个并发
> 假设用户容忍的响应的最大时间是3s
> 最大并发量 = 1600 * 3 = 4800个

26. CAS锁实现原理是什么,存在什么问题？怎么解决
实现原理参考代码和图
https://github.com/wolvesleader/interview/tree/master/base_java/src/main/java/com/quincy/java/base/cas

> 主要存在2个问题
> CPU爆满
> ABA问题

27. ThreadLocal是什么，在项目中怎么使用，底层实现原理是什么，怎么保证变量线程安全的

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

参考代码
https://github.com/wolvesleader/interview/tree/master/base_java/src/main/java/com/quincy/java/base/threadlocal

28. 线程池的创建方式有几种

> Executors.newCachedThreadPool();//带有缓存的线程池
> Executors.newFixedThreadPool(3);//固定线程数量的线程池
> Executors.newScheduledThreadPool(5);//定时执行的线程池

> Executors.newSingleThreadExecutor();//单线程的线程池

29. 线程池的实现原理

```
1 public static ExecutorService newSingleThreadExecutor() {
2     return new FinalizableDelegatedExecutorService
3         (new ThreadPoolExecutor(1, 1,
4                                 0L, TimeUnit.MILLISECONDS,
5                                 new LinkedBlockingQueue<Runnable>()));
6 }
```

```
1 public ThreadPoolExecutor(int corePoolSize,
2                           int maximumPoolSize,
3                           long keepAliveTime,
4                           TimeUnit unit,
5                           BlockingQueue<Runnable> workQueue) {
6     this(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue,
7          Executors.defaultThreadFactory(), defaultHandler);
8 }
```

到此我们可以结合线程池的原理图来理解
先创建队列，指定线程池的大小
把任务添加到队列中

30. 自己实现线程池应该怎么做
参考代码
https://github.com/wolvesleader/interview/tree/master/base_java/src/main/java/com/quincy/java/base/threadpool

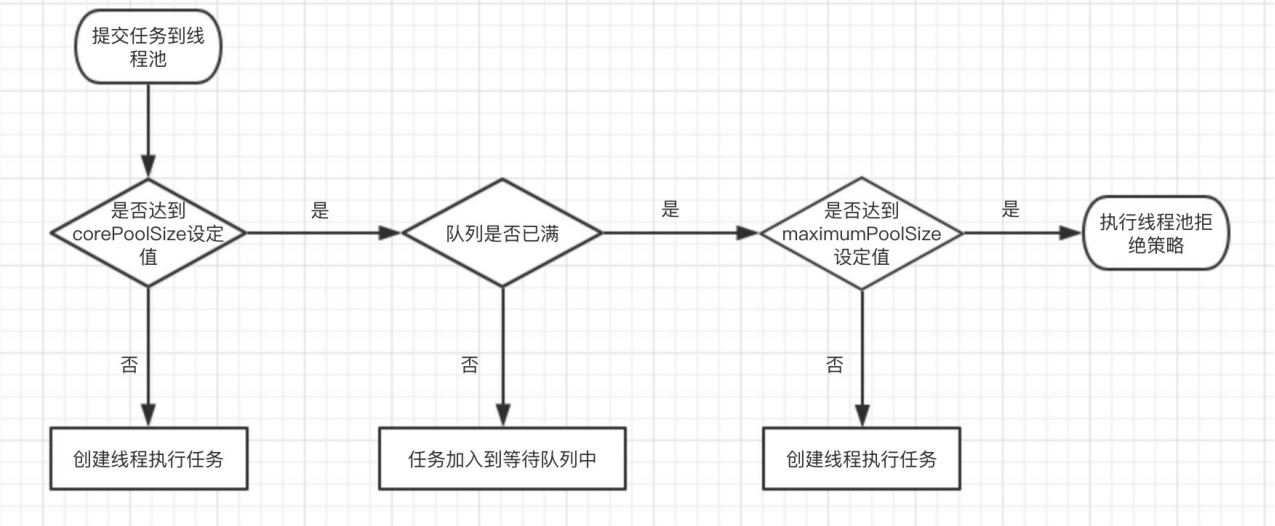
31. 线程池中corePoolSize和maximumPoolSize有什么区别

> corePoolSize线程池的基本大小，即在没有任务需要执行的时候线程池的大小，并且只有在工作队列满了的情况下才会创建超出这个数量的线程
> maximumPoolSize线程池中允许的最大线程数，线程池中的当前线程数目不会超过该值。如果队列中任务已满，并且当前线程个数小于maximumPoolSize，那么会创建新的线程来执行任务

32. 线程池的拒绝策略有哪些，分别在什么情况下触发

> AbortPolicy：直接抛出一个RejectedExecutionException异常（默认使用）
> CallerRunsPolicy：重试策略，如果线程池未关闭，一直重试
> DiscardOldestPolicy：丢弃最旧的未处理任务
> DiscardPolicy：直接丢弃当前任务

33. 线程池的执行原理图



34. Condition是什么怎么使用

> Condition是在java1.5中才出现的，它用来替代传统的Object的wait()、notify()实现线程间的协作，相比使用Object的wait()、notify()，使用Condition的await()、signal()这种方式实现线程间协作更加安全和高效。因此通常来说比较推荐使用Condition，阻塞队列实际上是使用了Condition来模拟线程间协作
> Condition和Object的区别如下
> Condition可以具体的让某一个线程等待，可以唤醒指定的线程，Object只能唤醒当前的线程或者唤醒全部的线程
> Condition必须和Lock锁配合来使用，Object必须配合synchronized使用

35. Immutability怎么解决线程安全问题的
    Immutability模式就是不变性，简单来讲，就是对象一旦被创建之后，状态就不在发生变化，变量一旦被赋值，就是变量一旦被赋值就
    不允许修改了，没有修改操作也就保持了不变性

36. Disruptor是怎么做到线程安全
    Disruptor是一个高性能的异步处理框架，或者可以认为是线程间通信的高效低延时的内存消息组件，它最大特点是高性能
    Disruptor线程安全的原因是采用了CAS锁，这也是其高效的一个原因

37. 在项目中哪里使用到了线程池，怎么使用的，为什么要使用线程池

> 我们在做项目的时候需要导入外部的excel数据，数据比较多，我们开线程来处理的，线程是从线程池中获取到的
> 还有一些第三方框架，例如netty底层也是需要创建线程池来实现的

38. 实现2个线程交替执行
    可以通过线程之间的通信来实现，2个是一种问题
    参考代码
    https://github.com/wolvesleader/interview/tree/master/base_java/src/main/java/com/quincy/java/base/threadcommunication

39. AtomicInteger是怎么做到线程安全的？请分析实现原理

40. run方法中能抛出异常吗？为什么

> run方法不能抛出异常，原因是run方法是继承Thread重写run方法或者是实现Runnable接口
> 实现run方法，在Thread和Runnable接口中run方法没有抛出异常，所以run方法中的异常不能抛出

41. 什么是happen-before模型，主要解决什么问题

> 1)程序顺序规则：一个线程中的每个操作，happens-before于该线程中的任意后续操作
> 2)监视器锁规则：对一个锁的解锁，happens-before于随后对这个锁的加锁
> 3)volatile变量规则：对一个volatile域的写，happens-before于任意后续对这个volatile域的读
> 4)传递性：如果A happens-before B，且B happens-before C，那么A happens-before C
> 5)start()规则：如果线程A执行操作ThreadB.start()（启动线程B），那么A线程的ThreadB.start()操作happens-before于线程B中的任意操作
> 6)join()规则：如果线程A执行操作ThreadB.join()并成功返回，那么线程B中的任意操作happens-before于线程A从ThreadB.join()操作成功返回
> 7)程序中断规则：对线程interrupted()方法的调用先行于被中断线程的代码检测到中断时间的发生
> 8)对象finalize规则：一个对象的初始化完成（构造函数执行结束）先行于发生它的finalize()方法的开始

42. 怎么监控线程状态

> 使用JDK自带的工具
> jstack生成虚拟机当前时刻的线程快照 通过jmap来分析快找信息
> jConsole 查看线程的详细信息
> VisualVM 查看线程的详细信息

43. 能从线程池中获取到一个线程吗？为什么

> 不能，线程池没有提供获取线程的方法
> 我们一般是把一个任务提交给线程池，线程池随机分配一个线程执行任务

44.
45.

46. 待续.....

Java I/O流

1. 流体系架构(请用插图来体现)
2. 流中使用到的设计模式

> 使用到了装饰着模式

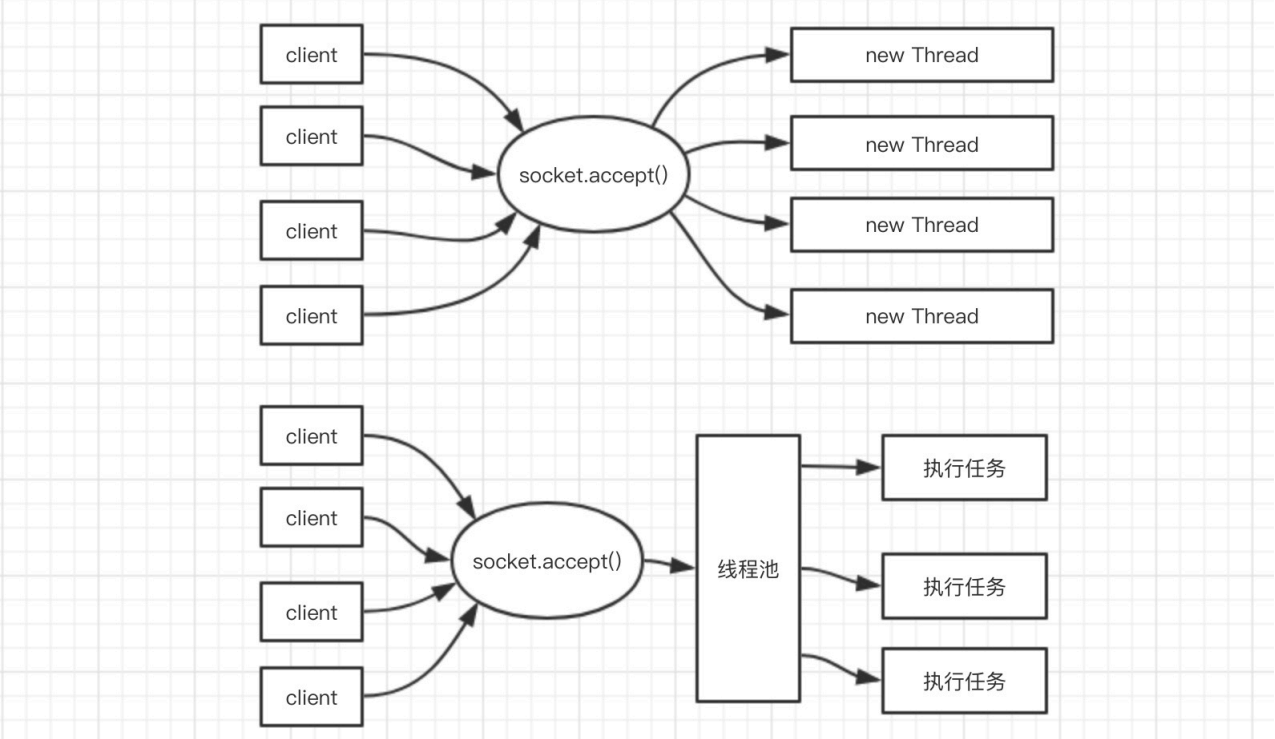
3. 在项目中使用过哪些流，怎么使用的

> 断点续传
> 文件上传下载
> 指定文件读取

4. NIO、BIO、AIO流的区别，以及优化方案

> BIO阻塞式IO流，什么叫阻塞式IO流所有 read() 操作和 write() 操作都会阻当前线程，如果客户端和服务器建立了一个连接
> 而迟迟不发送数据，那么服务端的read操作会处于阻塞状态，当前线程会处于阻塞状态，不能做其他事情，所以使用BIO模型，
> 一般都会为每个socket分配一个独立的线程，这样的问题就是会造成服务端线程数量很多，创建和销毁线程花费大量的时间，
> 可以采用线程池来管理，但是socket和线程之间的数量对应关系并没有发生变化

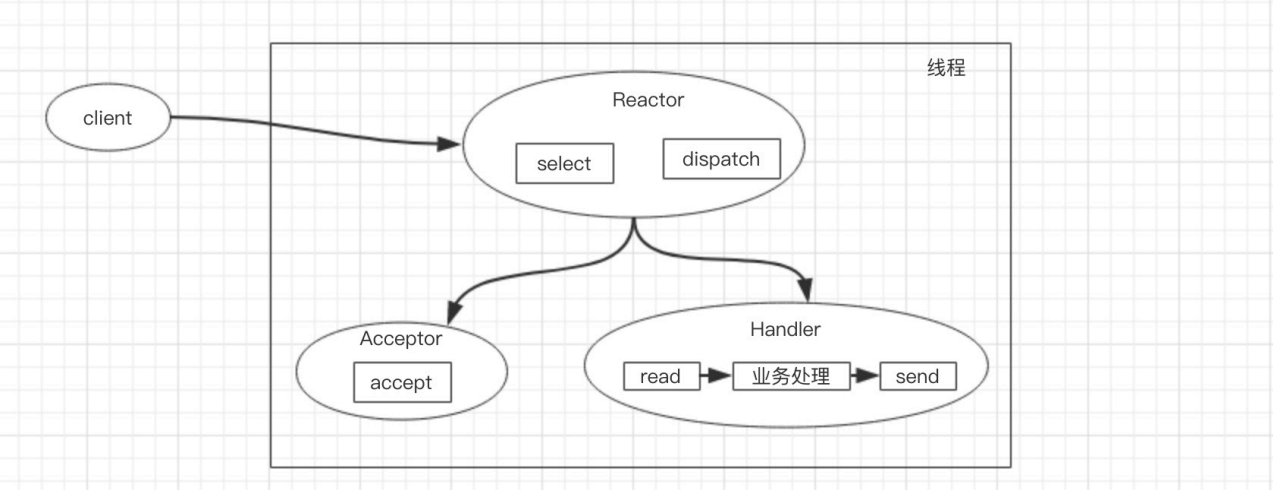
BIO线程模型图



BIO线程模型可以查看tomcat7的源码tomcat使用了线程池的方式

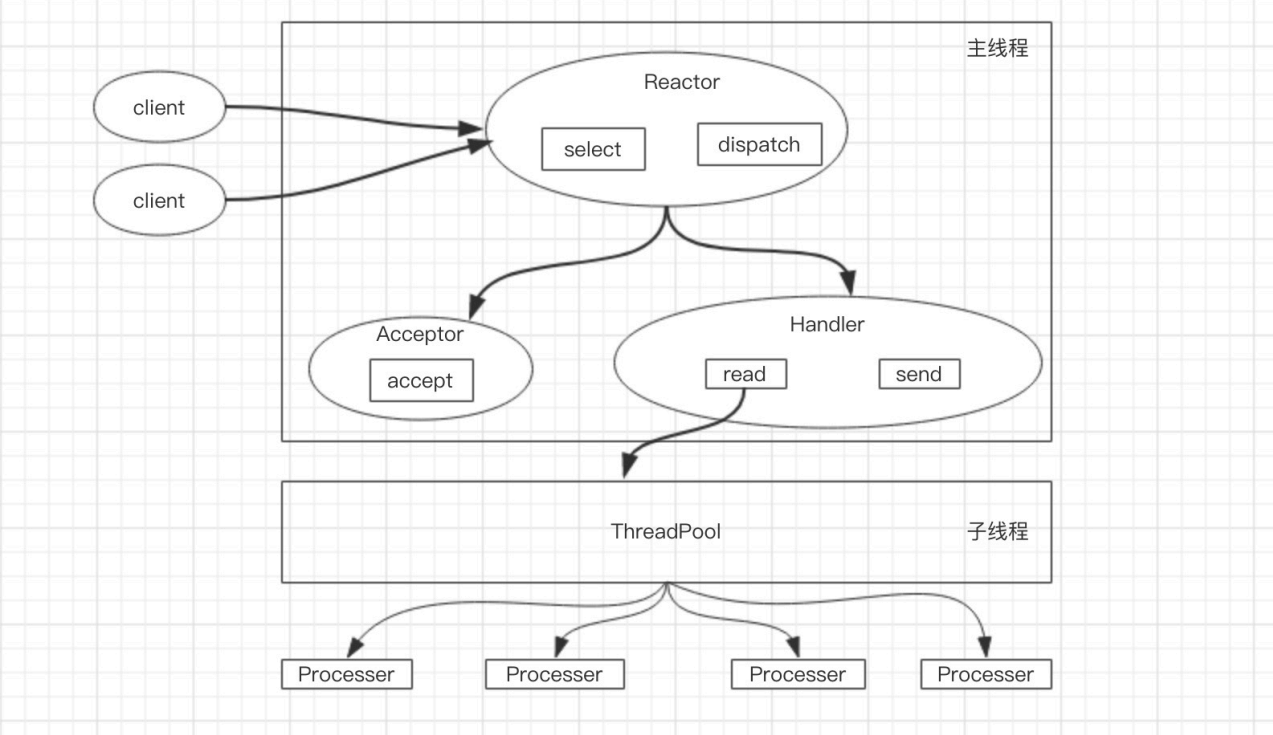
```
org.apache.coyote.http11.Http11Protocol
public Http11Protocol() {
    //处理请求
    endpoint = new JIoEndpoint();
    cHandler = new Http11ConnectionHandler(this);
    ((JIoEndpoint) endpoint).setHandler(cHandler);
    setSoLinger(Constants.DEFAULT_CONNECTION_LINGER);
    setSoTimeout(Constants.DEFAULT_CONNECTION_TIMEOUT);
    setTcpNoDelay(Constants.DEFAULT_TCP_NO_DELAY);
}
org.apache.tomcat.util.net.JIoEndpoint.Acceptor内部类
if (running && !paused && setSocketOptions(socket)) {
    // Hand this socket off to an appropriate processor
    if (!processSocket(socket)) {
        countDownConnection();
        // Close socket right away
        closeSocket(socket);
    }
}
org.apache.tomcat.util.net.JIoEndpoint
//线程池处理
getExecutor().execute(new SocketProcessor(wrapper));
```

NIO+Reactor(反应堆)+单线程 线程模型图



参考代码
https://github.com/wolvesleader/interview/tree/master/base_java/src/main/java/com/quincy/java/base/iomodel/nio_reactor_thread

NIO+Reactor(反应堆)+线程池 线程模型图



https://github.com/wolvesleader/interview/tree/master/base_java/src/main/java/com/quincy/java/base/iomodel/nio_reactor_threads

NIO+多Reactor(反应堆)+线程池 不做详解了，自己学习
参考资料
http://gee.cs.oswego.edu/dl/cpjslides/nio.pdf

看看tomcat7 NIO+线程池的显示代码

```
1 org.apache.tomcat.util.net.AbstractEndpoint#init 初始化NioEndpoint
2 org.apache.tomcat.util.net.NioEndpoint#bind acceptor的线程数为1, poller 的线程数1
3 org.apache.tomcat.util.net.AbstractEndpoint#start 启动过程
4 org.apache.tomcat.util.net.AbstractEndpoint#start
5 org.apache.tomcat.util.net.NioEndpoint#startInternal 启动工作线程池、 poller线程组、
  acceptor线程组 非常重要
6 org.apache.tomcat.util.net.NioEndpoint.Acceptor#run
7 org.apache.tomcat.util.net.NioEndpoint#setSocketOptions
```

```
 8 getPoller0().register(channel);
 9 org.apache.tomcat.util.net.NioEndpoint.Poller#register
10 org.apache.tomcat.util.net.NioEndpoint#processSocket  利用线程池来处理任务
```

5. 使用流过程中注意事项和优化方案

> 使用流一定要记得关闭
> 能使用多线程处理的任务一定要用多线程来处理
> 记得使用有缓存区的流

Java 反射

1. 反射概念

> JAVA反射机制是在运行状态中，对于任意一个类，都能够知道这个类的所有属性和方法；
> 对于任意一个对象，都能够调用它的任意方法和属性；这种动态获取信息以及动态调用
> 对象方法的功能称为java语言的反射机制

2. 反射创建对象方法有几种

> 1.newInstance
> 2.通过构造器newInstance

3. 反射怎么获取私有字段和方法

4. 在项目中哪里用到了反射，为什么要用

> 反射在我们的项目中用到的地方很多，例如
> 我们在做搜索模块的时候，需要自定义model
> 属性字段的注解，定义好注解之后，在扫描
> 类判断是否有注解标识就用到了反射
> 还有一些第三方框架，mybatis等等都用到了注解

5. 后续.....

Java异常

1. 常见的异常有哪些？产生的原因

> NullPointerException 空指针异常 一个对象为null 调用对象的方法或者属性
> ClassCastException 类型转换异常 Person p = (Person) User 相当于把猫类型不能转换为狗类型
> ArrayIndexOutOfBoundsException 下标越界异常
> FileNotFoundException 文件找不到异常
> SQLException SQL语句异常

2. 自定义异常链

3. 在项目中怎么处理异常的

4. 全局异常怎么捕获

5. 异常的处理方式有几种

> 2中处理方式
> try...catch 捕获异常
> throws 抛出异常

6. 异常体系架构图
7. 异常和错误有什么区别

> 异常可以处理
> 错误无法处理

8. throwable是接口还是抽象类

> 不是接口也不是抽象类，就是一个普通类

9. throws和throw区别

> throws 写在方法上 抛出异常标识
> throw 声明一个异常

10. 待续.....

Web基础

1. 什么是servlet，主要作用

> servlet就是一套API，是用来提供给服务器处理请求的API

2. jsp循环遍历

> jstl库foreach标签

3. servlet线程安全吗？怎么解决

> servlet是线程不安全的,解决办法
> 1)在servlet类中不要写成员变量,把变量全部放在方法中

4. servlet域对象有哪些

> request
> session
> servletcontext

5. 待续.....

框架

springmvc

1. springmvc整合web，整和spring

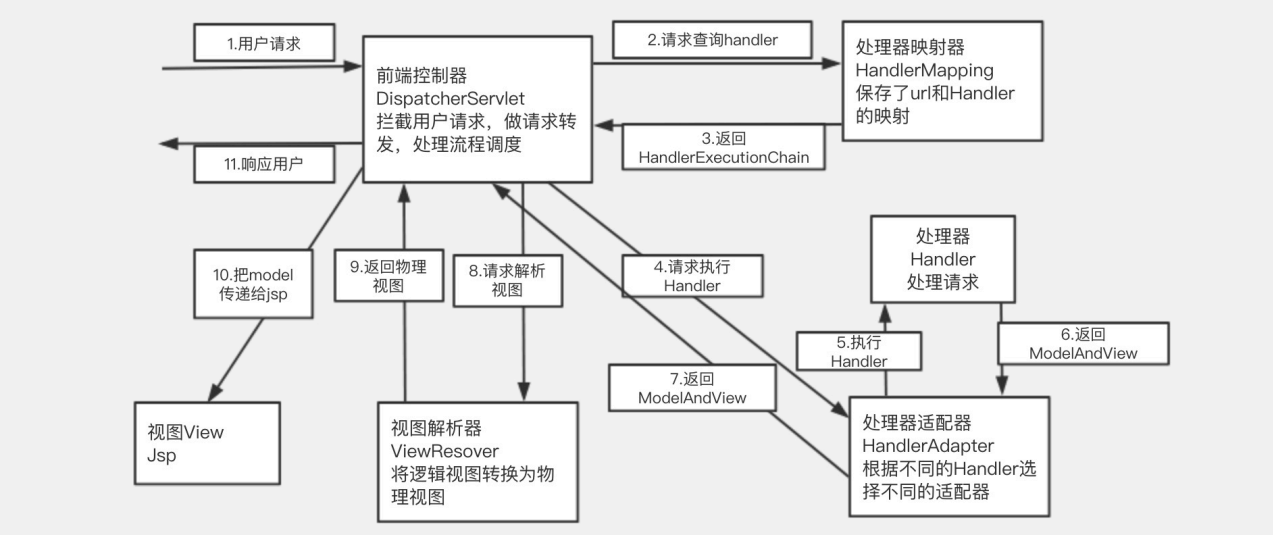
> springmvc整合web是在web.xml文件中配置springmvc的入口Servlet
> springmvc是spring的子框架，直接使用spring的注解就行，不用做其他的整合

2. springmvc常用的注解，每个注解的作用

> 加在类上的注解
> @RequestMapping 配置访问路径
> @ResponseBody 返回json格式数据
> @RequestBody json格式的数据解析为javabean

> @PathVariable 获取路径中的变量
> @RequestParam 表单中的name属性和controller方法上的参数名不一致
> @ModelAttribute("user") 它的作用是将该绑定的命令对象以"user"为名称添加到模型对象中供视图页面展示使用。我们此时可以在视图页面使用${user.username}来获取绑定的命令对象的属性

3. springmvc执行流程



4. springmvc参数绑定原理

```
@RequestMapping(value ="index",method = RequestMethod.GET)
 public String index(Person person){
    System.out.println(person);
    return "helloword";
 }
 以此代码来分析对象的绑定
 org.springframework.web.servlet.mvc.method.annotation.ServletInvocableHandlerMethod#invokeAndHandle
 public void invokeAndHandle(.....) throws Exception {
     Object returnValue = this.invokeForRequest(webRequest, mavContainer, providedArgs);
     //...省略
 }
 org.springframework.web.method.support.InvocableHandlerMethod#invokeForRequest
 public Object invokeForRequest(......) throws Exception {
     Object[] args = this.getMethodArgumentValues(request, mavContainer, providedArgs);
     Object returnValue = this.doInvoke(args);
     return returnValue;
 }
 org.springframework.web.method.support.InvocableHandlerMethod#getMethodArgumentValues
 private Object[] getMethodArgumentValues(......) throws Exception {
     MethodParameter[] parameters = this.getMethodParameters();
     Object[] args = new Object[parameters.length];
     args[i] = this.argumentResolvers.resolveArgument(parameter, mavContainer, request, this.dataBinderFactory);
     return args;
 }
 org.springframework.web.method.support.HandlerMethodArgumentResolverComposite#resolveArgument
 public Object resolveArgument(......) throws Exception {
    return resolver.resolveArgument(parameter, mavContainer, webRequest, binderFactory);
```

```
}
org.springframework.web.method.annotation.ModelAttributeMethodProcessor#resolveArgument
public final Object resolveArgument(......) throws Exception {
    Object attribute = null;
    //根据参数类型创建出对象，在该案例中创建出Person对象
    //创建出来的对象字段没有值
    attribute = this.createAttribute(name, parameter, binderFactory, webRequest);
    WebDataBinder binder = binderFactory.createBinder(webRequest, attribute, name);
    //给创建出来的对象属性设置值
    his.bindRequestParameters(binder, webRequest);
    return attribute;
}
org.springframework.web.servlet.mvc.method.annotation.ServletModelAttributeMethodProcessor#bindRequestParameters
protected void bindRequestParameters(WebDataBinder binder, NativeWebRequest request) {
    ServletRequest servletRequest = (ServletRequest)request.getNativeRequest(ServletRequest.class);
    servletBinder.bind(servletRequest);
}
org.springframework.web.bind.ServletRequestDataBinder#bind
public void bind(ServletRequest request) {
    MutablePropertyValues mpvs = new ServletRequestParameterPropertyValues(request);
    this.addBindValues(mpvs, request);
    this.doBind(mpvs);
}
org.springframework.web.bind.ServletRequestParameterPropertyValues#ServletRequestParameterPropertyValues()
org.springframework.web.util.WebUtils#getParametersStartingWith
public static Map<String, Object> getParametersStartingWith(ServletRequest request, @Nullable String prefix) {
    Assert.notNull(request, "Request must not be null");
    //获取前段提交过来的数据
    Enumeration<String> paramNames = request.getParameterNames();
    Map<String, Object> params = new TreeMap();
    if (prefix == null) {
        prefix = "";
    }

    while(paramNames != null && paramNames.hasMoreElements()) {
        String paramName = (String)paramNames.nextElement();
        if ("".equals(prefix) || paramName.startsWith(prefix)) {
            String unprefixed = paramName.substring(prefix.length());
            String[] values = request.getParameterValues(paramName);
            if (values != null && values.length != 0) {
                //把数据存入到集合中
                if (values.length > 1) {
                    params.put(unprefixed, values);
                } else {
                    params.put(unprefixed, values[0]);
                }
            }
        }
    }
    //返回数据集合
    return params;
```

```
}

org.springframework.web.method.support.InvocableHandlerMethod#invokeForRequest
public Object invokeForRequest(......) throws Exception {
    //对象创建了，页面数据也获取到了，数据也绑定了
    //args handler参数
    Object[] args = this.getMethodArgumentValues(request, mavContainer, providedArgs);
    //执行返回结果
    Object returnValue = this.doInvoke(args);
    return returnValue;
}
```

5. springmvc返回json格式数据

> 使用springmvc默认的json解析器

```
@RequestMapping("/test/get")
@ResponseBody
public String get(){
    return json串;
}
```

> 使用第三方json解析器,在springmvc.xml中配置

```
<mvc:annotation-driven>
    <mvc:message-converters register-defaults="true">
        <!-- 配置Fastjson支持 -->
        <bean class="com.alibaba.fastjson.support.spring.FastJsonHttpMessageConverter">
            <property name="supportedMediaTypes">
                <list>
                    <value>text/html;charset=UTF-8</value>
                    <value>application/json</value>
                </list>
            </property>
            <property name="features">
                <list>
                    <value>WriteMapNullValue</value>
                    <value>QuoteFieldNames</value>
                </list>
            </property>
        </bean>
    </mvc:message-converters>
</mvc:annotation-driven>
```

```
@RequestMapping("/test/get")
@ResponseBody
public String get(){
    return json串;
}
```

6. springmvc文件上传

> 页面form表单配置 method="post" enctype="multipart/form-data"属性

springmvc.xml文件中配置

```
<bean id="multipartResolver"
```

```
  class="org.springframework.web.multipart.commons.CommonsMultipartResolver">
  <property name="maxUploadSize" value="104857600" />
  <property name="maxInMemorySize" value="4096" />
  <property name="defaultEncoding" value="UTF-8"></property>
</bean>
```

controller类方法中配置

```
public String  fileUpload(CommonsMultipartFile file){.....}
```

7. springmvc和spring子父容器关系

> springmvc是spring的子容器,父容器不能访问子容器中的对象，子容器可以访问访问父容器的对象
> 但是必须打开访问开关，在springmvc.xml中配置

```
<bean
  class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerMapping">
 <property name="detectHandlerMethodsInAncestorContexts">
   <value>true</value>
 </property>
</bean>
```

这也是我们在spring的配置文件中配置了包扫描

```
<context:component-scan base-package="com.quincy.controller" />
```

之后，还需要在springmvc.xml中在其配置扫描controller包的原因

8. 为什么要选择springmvc框架

> 1)springmvc框架是spring的子框架，兼容好一些
> 2)springmvc社区比较活跃，版本不断在更新，出现问题也能快速解决
> 3)springmvc支持resetful风格
> 4)springmvc容易上手比较简单
> 5)团队大部分人用过springmvc框架

9. springmvc的controller线程安全吗？怎么解决

> 线程不安全，可以使用spring scop创建多例

10. springmvc支持restful风格吗？有哪些注解

> 支持
> @PathVariable 获取路径中的变量

11. springmvc获得request，response,session的几种方式

```
@RequestMapping("/test")
@ResponseBody
public void saveTest(HttpServletRequest req, HttpServletResponse resp){

}
```

12. 注解方式springmvc使用的是那个requestmapping,adaper

> BeanNameUrlHandlerMapping： 通过对比url和bean的name找到对应的对象
> SimpleUrlHandlerMapping： 也是直接配置url和对应bean,比BeanNameUrlHandlerMapping功能更

> 多
> DefaultAnnotationHandlerMapping : 主要是针对注解配置@RequestMapping的，已过时
> RequestMappingHandlerMapping： 取代了上面的DefaultAnnotationHandlerMapping
> HttpRequestHandlerAdapter： 要求Handler实现HttpRequestHandler接口，该接口的方法为 void handleRequest(HttpServletRequest request, HttpServletResponse response)也就是 handler必须有一个handleRequest方法
> SimpleControllerHandlerAdapter：要求handler实现Controller接口，该接口的方法为ModelAndView handleRequest(HttpServletRequest request, HttpServletResponse response)，也就是本工程采用的
> AnnotationMethodHandlerAdapter： 和上面的DefaultAnnotationHandlerMapping配对使用的，也已过时
> RequestMappingHandlerAdapter : 和上面的RequestMappingHandlerMapping配对使用，针对@RequestMapping

13. springmvc怎么优化

> 建议使用@RequestParam注解，我们分析原因

```
@RequestMapping(value ="index",method = RequestMethod.GET)
public String index(String username,String password){
    System.out.println(username + "--" + password);
    return "helloword";
}
```

```
org.springframework.web.method.annotation.AbstractNamedValueMethodArgumentResolver#resolveArgument
public final Object resolveArgument(......) throws Exception {
    NamedValueInfo namedValueInfo = getNamedValueInfo(parameter);
    MethodParameter nestedParameter = parameter.nestedIfOptional();
    Object resolvedName = resolveStringValue(namedValueInfo.name);
    //根据参数名从request请求中获取到页面传入的数据值
    //现在我们需要的就是获取参数名
    Object arg = resolveName(resolvedName.toString(), nestedParameter, webRequest);
}
org.springframework.web.method.annotation.AbstractNamedValueMethodArgumentResolver#getNamedValueInfo
private NamedValueInfo getNamedValueInfo(MethodParameter parameter) {
    NamedValueInfo namedValueInfo = this.namedValueInfoCache.get(parameter);
    if (namedValueInfo == null) {
        namedValueInfo = createNamedValueInfo(parameter);
        namedValueInfo = updateNamedValueInfo(parameter, namedValueInfo);
        this.namedValueInfoCache.put(parameter, namedValueInfo);
    }
    return namedValueInfo;
}
org.springframework.web.method.annotation.RequestParamMethodArgumentResolver#createNamedValueInfo
protected NamedValueInfo createNamedValueInfo(MethodParameter parameter) {
    RequestParam ann = parameter.getParameterAnnotation(RequestParam.class);
    //如果没有注解RequestParamNamedValueInfo
    return (ann != null ? new RequestParamNamedValueInfo(ann) : new RequestParamNamedValueInfo());
}
org.springframework.web.method.annotation.AbstractNamedValueMethodArgumentResolver#updateNamedValueInfo
private NamedValueInfo updateNamedValueInfo(MethodParameter parameter, NamedValueInfo
```

```
    info) {
28      String name = info.name;
29      //如果使用注解name不为空，不会执行获取参数名的方法
30      if (info.name.isEmpty()) {
31          name = parameter.getParameterName();
32      }
33      String defaultValue = (ValueConstants.DEFAULT_NONE.equals(info.defaultValue) ? null
    : info.defaultValue);
34      return new NamedValueInfo(name, info.required, defaultValue);
35  }
36  org.springframework.core.MethodParameter#getParameterName
37  public String getParameterName() {
38      ParameterNameDiscoverer discoverer = this.parameterNameDiscoverer;
39      if (discoverer != null) {
40          String[] parameterNames = null;
41          if (this.executable instanceof Method) {
42              //通过反射和asm框架获取到所有参数名
43              parameterNames = discoverer.getParameterNames((Method)this.executable);
44          } else if (this.executable instanceof Constructor) {
45              parameterNames =
    discoverer.getParameterNames((Constructor)this.executable);
46          }
47          if (parameterNames != null) {
48              //根据索引获取到参数名
49              this.parameterName = parameterNames[this.parameterIndex];
50          }
51          this.parameterNameDiscoverer = null;
52      }
53      return this.parameterName;
54  }
```

14. springmvc的核心组件有哪些

> 1)HandlerMappings handle对应的是一个方法或者一个类，例如标注了RequestMapping的方法就是一个Handler
> 2)HandlerAdapters 根据handlerMapping传过来的controller与已经注册好了的HandlerAdapter一一匹配，看哪一种HandlerAdapter是支持该controller类型的,如我们常用注解方法找到的是RequestMappingHandlerAdapter,request和response不能和方法的参数做匹配,把request中的参数转换为handler能够接受的参数
> 3)MultipartResolver 上传文件组件
> 4)LocaleResolver 解析浏览器发过来数据的语言环境
> 5)ThemeResolver 用来换皮肤的，实现统一个view用不同的css、图片给页面渲染不同的风格
> 6)HandlerExceptionResolvers handler出现异常为异常配置友好的可视化界面
> 7)RequestToViewNameTranslator 把一个普通的请求展示成一个可以看到的结果
> 8)ViewResolvers 试图解析器
> 9)FlashMapManager

15. springmvc怎样设定重定向和转发
    重定向

```
1  @RequestMapping("/test/get")
2  public String testRedirect(){
3      return "redirect:/index.jsp";
4  }
```

转发，默认就使用转发的方式

```
1 @RequestMapping("/test/get")
2 public String testRedirect(){
3     return "index.jsp";
4 }
```

16. 待续.....

spring

1. 为什么选择spring框架，使用有什么好处

> 1)spring是一个平台提供了对其他框架的整合功能
> 2)spring使用很广社区活跃度很高，出现问题容易解决
> 3)spring帮助我们创建对象和维护对象的生命周期，我们只关注业务逻辑

2. spring启动流程

3. spring 特性ioc,di是什么，实现原理是什么

4. spring常用的注解

> 加在类上的注解
> @Controller
> @Service
> @Repository
> @Component
> 加载属性上的注解
> @Autowired
> @Qualifier
> @Value
> @Bean

5. @autowird和@qualifier,@resourse区别

> @Autowired默认按类型装配,如果接口提供了多中实现,需要使用@Qualifier(“userService”)精确制定注入哪一个,
> 是spring提供的注解
> @resourse默认安装名称进行装配,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,是JSR250提供的注解

6. spring bean生命周期，实现哪些接口可以，自定生命周期行为

7. spring中的循环注入是什么意思，怎么解决

> IOC 容器在读到上面的配置时，会按照顺序，先去实例化beanA。
> 然后发现 beanA 依赖于 beanB，接在又去实例化 beanB。实例化
> beanB 时，发现 beanB 又依赖于beanA。如果容器不处理循环依
> 赖的话，容器会无限执行上面的流程,直到内存溢出，程序崩溃。

> 当然，Spring 是不会让这种情况发生的。在容器再次发现 beanB
> 依赖于 beanA 时，容器会获取 beanA 对象的一个早期的引用(early reference)，
> 并把这个早期引用注入到 beanB 中，让 beanB 先完成实例化。
> beanB 完成实例化,beanA 就可以获取到 beanB 的引用，beanA 随之完成实例化。这里大家可能不知道“早期引用”是什么意思

8. spring 框架中都用到了哪些设计模式

> 工厂模式BeanFactory
> 代理模式

9. spring如何保证高并发下Controller安全

> 可以使用多例模式

10. spring事物配置方式有几种

> 1)声名式事物

```
<!-- 定义事务管理器 -->
<bean id="transactionManager"
class="org.springframework.orm.hibernate3.HibernateTransactionManager">
    <property name="sessionFactory" ref="sessionFactory" />
</bean>
<!-- 定义事务 -->
<tx:advice id="txAdvice" transaction-manager="transactionManager">
    <tx:attributes>
        <tx:method name="*" propagation="REQUIRED" />
    </tx:attributes>
</tx:advice>
<!-- 定义切面 -->
<aop:config>
    <aop:pointcut id="interceptorPointCuts" expression="execution(* com.dao.*.*(..))"
/>
    <aop:advisor advice-ref="txAdvice" pointcut-ref="interceptorPointCuts" />
</aop:config>
```

> 2)注解式事物

```
<tx:annotation-driven transaction-manager="transactionManager"/>
<!-- 定义事务管理器 -->
<bean id="transactionManager"
class="org.springframework.orm.hibernate3.HibernateTransactionManager">
    <property name="sessionFactory" ref="sessionFactory" />
</bean>
```

11. spring注解扫描原理

12. spring核心模块有哪些

> spring-core：Spring中的核心工具类包
> spring-beans：Spring中定义bean的组件
> spring-context：Spring的运行容器
> spring-aop：基于代理的AOP支持
> spring-aspects：集成Aspects的AOP支持
> spring-web：提供web的基础功能

> spring-webmvc：提供springmvc的功能
> spring-jdbc：提供对jdbc连接的封装功能
> spring-tx：提供对事务的支持
> spring-orm：提供对象- 关系映射支持

13. spring BeanFactory和FactoryBean区别

> BeanFactory是spring简单工厂模式的接口类，spring IOC特性核心类，提供从工厂类中获取bean的各种方法，是所有bean的容器
> FactoryBean仍然是一个bean，但不同于普通bean，它的实现类最终也需要注册到BeanFactory中。它也是一种简单工厂模式的接口类，但是生产的是单一类型的对象，与BeanFactory生产多种类型对象不同

14. spring事物传播机制有哪些

> REQUIRED
> 如果当前没有事务，就新建一个事务；如果已经存在一个事务中，加入到这个事务中。这是最常见的选择，也是spring默认的行为
> SUPPORTS
> 支持当前事务；如果当前没有事务，就以非事务方式执行。
> MANDATORY
> 使用当前的事务，如果当前没有事务，就抛出异常。
> REQUIRES_NEW
> 新建事务，如果当前存在事务，把当前事务挂起。内部事物与外部事物互不影响
> NOT_SUPPORTED
> 以非事务方式执行操作，如果当前存在事务，就把当前事务挂起。
> NEVER
> 以非事务方式执行，如果当前存在事务，则抛出异常。
> NESTED
> 如果当前没有事务，创建事物执行；如果当前存在事务，则创建依赖外部事物的事物，即若内部事物回滚，外部事物也回滚，若外部事物回滚，则内部事物也回滚

15. 待续.....

mybatis

1. resultmap是做什么的

> 模型字段和sql字段不匹配手动指定

2. 1对1查询，1对多查询

> association collection

3. 动态sql的常用标签，动态sql的执行原理

> if where trim set foreach choose(when, otherwise)

4. 如何在插入后获得主键

> 1)方式一

```
1 <insert id="insert2" useGeneratedKeys="true" keyProperty="id">
2     insert into
  sys_user(user_name,user_password,user_email,user_info,head_img,create_time) values(
```

```
3        #{userName},#{userPassword},#{userEmail},#{userInfo},#{headImg,jdbcType=BLOB},#{createTime,jdbcType=TIMESTAMP}
4      )
5  </insert>
```

2)方式二

```
1 <insert id="insert">
2     insert into sys_user(user_name, user_password, user_email, user_info, head_img, create_time)
3     values (#{userName}, #{userPassword}, #{userEmail}, #{userInfo}, #{headImg, jdbcType=BLOB}, SYSDATE())
4     <selectKey keyColumn="id" resultType="long" keyProperty="id" order="AFTER">
5         SELECT LAST_INSERT_ID()
6     </selectKey>
7 </insert>
```

5. 如何实现分页，分页插件的原理是什么

> pagehelper

6. ${} 和#{}有什么区别

> 1)#{}占位符，防止SQL注入
> 2)${}sql拼接符号

7. mybatis和hibernate区别

> hibernate是一个orm框架，面向对象开发，不用自己写SQL，但是不好做SQL调优
> mybatis半orm框架，可以自己写sql，能做sql调优

8. 待续.....

springboot

1. 为什么选择springboot

> Spring Boot 是 Spring 开源组织下的子项目，是 Spring 组件一站式解决方案，
> 主要是简化了使用 Spring 的难度，简省了繁重的配置，提供了各种启动器，开
> 发者能快速上手。Spring Boot 优点非常多，如：独立运行、简化配置、自动配置、
> 无代码生成和XML配置、应用监控、上手容易

2. springboot提供了哪些核心功能

> 独立运行Spring项目
> Spring boot?可以以jar包形式独立运行，运行一个Spring Boot项目只需要通过java -jar xx.jar来运行。
> 内嵌servlet容器
> Spring Boot可以选择内嵌Tomcat、jetty或者Undertow,这样我们无须以war包形式部署项目。
> 提供starter简化Maven配置
> spring提供了一系列的start pom来简化Maven的依赖加载，例如，当你使用了spring-boot-starter-web，会自动加入它所依赖的所有依赖包。
> 自动装配Spring?
> SpringBoot会根据在类路径中的jar包，类、为jar包里面的类自动配置Bean，这样会极大地减少我们要使用的配置。当然，SpringBoot只考虑大多数的开发场景，并不是所有的场景，若在实际开发中我们需要配置Bean，而SpringBoot灭有提供支持，则可以自定义自动配置。
> 准生产的应用监控

> SpringBoot提供基于http ssh telnet对运行时的项目进行监控。
> 无代码生产和xml配置
> SpringBoot不是借助与代码生成来实现的，而是通过条件注解来实现的，这是Spring4.x提供的新特性。

3. springboot中的starter 是什么

> starter可以当成是一个maven以来组，引入这个组名就引入了所有的依赖。
> 对于一些starter，比如要使用redis、jpa等等，就不仅仅是引入依赖了，还需要实现一些初始的配置
> @Configuration，这个注解就会在springboot启动时去实例化被其修饰的类，前提是springboot配置了@EnableAutoConfiguration，而springboot启动类默认的@SpringBootApplication中默认包含了该注解，所以不用再显示引入，最后需要在starter项目中META-INF/spring.factories中添加:org.springframework.boot.autoconfigure.EnableAutoConfiguration=com.example.xxx.Xxx(类全路径)
> 这样在springboot启动时，才能正确加载自定义starter的配置。
> 所以说，starter中简单来讲就是引入了一些相关依赖和一些初始化的配置。
> 为什么加了@Configuration注解还是要配置META-INF/spring.factories呢?
> 因为springboot项目默认只会扫描本项目下的带@Configuration注解的类，如果自定义starter，不在本工程中，是无法加载的，所以要配置META-INF/spring.factories配置文件。
> 为什么配置了META-INF/spring.factories配置文件就可以加载?
> 这里才是springboot实现starter的关键点，springboot的这种配置加载方式是一种类SPI（Service Provider Interface）的方式，SPI可以在META-INF/services配置接口扩展的实现类，springboot中原理类似，只是名称换成了spring.factories而已。

4. springboot常用的starter 有哪些

> spring-boot-starter-web 嵌入tomcat和web开发
> spring-boot-starter-data-jpa 数据库支持
> spring-boot-starter-data-redis redis数据库支持
> spring-boot-starter-data-solr solr支持
> mybatis-spring-boot-starter 第三方的mybatis集成starter

5. springboot的配置文件有哪几种格式

> SpringBoot使用一个以application命名的配置文件作为默认的全局配置文件。支持properties后缀结尾的配置文件或者以yml/yaml后缀结尾的YAML的文件配置。
> 同时存在时：properties配置优先级 > YAML配置优先级。（先加载yml的配置，后加载properties配置，后加载的会覆盖先加载的配置）。
> SpringBoot配置文件可以放置在多种路径下，不同路径下的配置优先级有所不同。
> 可放置目录(优先级从高到低)
> 1.file:./config/ (当前项目路径config目录下);
> 2.file:./ (当前项目路径下);
> 3.classpath:/config/ (类路径config目录下);
> 4.classpath:/ (类路径config下).
> 优先级由高到底，高优先级的配置会覆盖低优先级的配置;
> SpringBoot会从这四个位置全部加载配置文件并互补配置;
> 我们可以从ConfigFileApplicationListener这类便可看出，其中DEFAULT_SEARCH_LOCATIONS属性设置了加载的目录：private static final String DEFAULT_SEARCH_LOCATIONS = "classpath:/,classpath:/config/,file:./,file:./config/";
> springboot也提供了加载外部配置文件的方法：
> @PropertySource通常用于属性加载配置文件，注意@PropertySource注解不支持加载yaml文件，支

> 持properties文件。
> @ImportResource通常用于加载Spring的xml配置文件(SpringBoot提出零xml的配置，因此SpringBoot默认情况下时不会识别项目中Spring的xml配置文件。为了能够加载xml的配置文件，SpringBoot提供了@ImportResource注解该注解可以加载Spring的xml配置文件，通常加于启动类上);想加载yaml配置文件，那么可以通过PropertySourcesPlaceholderConfigurer类来加载yaml文件，例如：
> @Bean
> public static PropertySourcesPlaceholderConfigurer loadProperties() {
> PropertySourcesPlaceholderConfigurer configurer = new PropertySourcesPlaceholderConfigurer();
> YamlPropertiesFactoryBean yaml = new YamlPropertiesFactoryBean();
> //yaml.setResources(new FileSystemResource("classpath:config/user.yml"));//File路径引入
> yaml.setResources(new ClassPathResource("config/user.yml"));//class路径引入
> configurer.setProperties(yaml.getObject());
> return configurer;
> }

6. springboot怎么管理配置文件版本

```
1.通过标签<parent>:
    <parent>
       <groupId>org.springframework.boot</groupId>
       <artifactId>spring-boot-starter-parent</artifactId>
       <version>1.4.1.RELEASE</version>
     </parent>
   之后添加具体stater时可以不指定版本；
   以上的这种统一版本的管理是spring boot默认的方式。
2.通过标签<dependencyManagement>:
   <dependencyManagement>
       <dependencies>
           <dependency>
              <groupId>org.springframework.boot</groupId>
              <artifactId>spring-boot-dependencies</artifactId>
              <version>1.4.1.RELEASE</version>
              <type>pom</type>
              <scope>import</scope>
           </dependency>
       </dependencies>
    </dependencyManagement>
```

7. springboot的核心注解是哪个

> SpringBoot的核心注解是@SpringBootApplication由以下3个注解组成：
> @SpringBootConfiguration：它组合了Configuration注解实现了 配置文件的功能。
> @EnableAutoConfiguration：打开自动配置功能，也可以关闭某个指定的自动配置选项如关闭数据源自动配置功能： @SpringBootApplication(exclude = { DataSourceAutoConfiguration.class })。
> @ComponentScan：Spring扫描组件。

8. springboot自动配置原理是什么

> 1.通过各种注解实现了类与类之间的依赖关系，容器在启动的时候Application.run，会调用EnableAutoConfigurationImportSelector.class的selectImports方法（其实是其父类的方法）--这里需要注意，调用这个方法之前发生了什么和是在哪里调用这个方法需要进一步的探讨
> 2.selectImports方法最终会调用SpringFactoriesLoader.loadFactoryNames方法来获取一个全面的常用BeanConfiguration列表

> 3.loadFactoryNames方法会读取FACTORIES_RESOURCE_LOCATION（也就是spring-boot-autoconfigure.jar 下面的spring.factories），获取到所有的Spring相关的Bean的全限定名ClassName，大概120多个
> 4.selectImports方法继续调用filter(configurations, autoConfigurationMetadata);这个时候会根据这些BeanConfiguration里面的条件，来一一筛选，最关键的是@ConditionalOnClass，这个条件注解会去classpath下查找，jar包里面是否有这个条件依赖类，所以必须有了相应的jar包，才有这些依赖类，才会生成IOC环境需要的一些默认配置Bean
> 5.最后把符合条件的BeanConfiguration注入默认的EnableConfigurationPropertie类里面的属性值，并且注入到IOC环境当中

9. springboot有哪几种读取配置的方式?

```
1.@Value注解:
    @Component
    public class Student {

        @Value("${student.name}")
        private String name;

        public String getName() {
            return name;
        }
        public void setName(String name) {
            this.name = name;
        }
    }

2.@ConfigurationProperties注解
    @ConfigurationProperties(prefix = "person")//读取配置文件中前缀为person对应的属性值
    @Component
    public class Person {
        private String lastName;
        public String getLastName() {
            return lastName;
        }
        public void setLastName(String lastName) {
            this.lastName = lastName;
        }
    }

3.@PropertySource配合@Value
    @Component
    @PropertySource(value = {"classpath:config/teacher.properties"})
    public class Teacher {
        @Value("${name}")
        private String name;
        public String getName() {
            return name;
        }
        public void setName(String name) {
            this.name = name;
        }
    }
4.@ConfigurationProperties配合@PropertySource配合@Value
    @Component
```

```
    @ConfigurationProperties(prefix = "woman")
    @PropertySource(value = { "classpath:config/woman.properties" })
    public class Woman {
        @Value("${name}")
        private String name;

        public String getName() {
            return name;
        }

        public void setName(String name) {
            this.name = name;
        }
    }

5.@ConfigurationProperties配合@Value
    @Component
    @ConfigurationProperties(prefix = "man")
    public class Man {
        @Value("${a}")
        private String name;

        public String getName() {
            return name;
        }

        public void setName(String name) {
            this.name = name;
        }
    }
6.注入Environment
    @Autowired
    private Environment env;
    @Test
    public void testEnvironment() {
        System.out.println(env.getProperty("student.name"));
    }

```

10. 待续.....

数据库

MySQL数据库

1. mysql常见的sql语句编写
2. 事物特性

> InnoDB存储引擎的事务特性是ACID也叫事务的酸性
> 原子性(A):构成事务的的所有操作必须是一个逻辑单元，要么全部执行，要么全部不执行
> 一致性(C):一致性是指事务将数据库从一种状态转变为下一种一致的状态，在事务开始之前和事务结束以后，数据库的完整性约束并没有被破坏。例如，表中有一个字段为姓名，为唯一约束，即在表中姓名不能重复。如果一个事务对姓名字段进行了修改，但是在事务提交或事务操作发生回滚后，表的姓名变得非唯一了，这就破坏了事务的一致性要求，即事务将从一种状态变为一种不一致的状态，因此事务是一致性的单位，如果事务中的某个动作失败了，系统可以自动撤销事务---返回初始化状态

> 隔离性(I):隔离性还有其他称呼，如并发控制(concurrency control)、可串行化(serializability)、锁(locking)等。事务的隔离性要求每个读写事务的对象对其他事务的操作对象能互相分离，即该事务提交对其他事务是不可见的，通常这使用锁来实现。当前数据库系统中提供了一种粒度锁(granular lock)的策略，允许事务锁住一个实体对象的自己，以此来提高事务之间的并发度
> 持久性(D):持久性事务一旦提交，其结果是永久性的，即发生宕机托故障，数据库也能将数据恢复

3. 事物隔离级别

事务隔离级别	脏读	不可重复	幻读
读未提交(read uncommitted)	是	是	是
不可重复读(read committed)	否	是	是
可重复读(repeatable read)	否	否	是
串行化(serializable)	否	否	否

> 脏读:一个事务修改了数据，但尚未提交，而另外一个事务中读到这些未被提交的数据。
> 不可重复读:事务T1读取一次数据，但是还没有结束，事务T2也开始操作这条数据，并且修改了这条数据，提交事务，T1事务再次读取这条数据的时候，发现和第一次读取的数据不相同，这样就发生了在同一个事务内两次读取到的数据不相同，这样的效果称之为不可重复读
> 幻度:是指当事务不是独立执行时发生的一种现象，例如在事务T1中插入id为15的一条记录，提交事务，在事务T2中查询并没有发现这条记录，我把同样在T2事务中把id为15的数据插入到数据库，这个时候就会报主键冲突的错误
> 幻读和不可重复读的区别:
> 不可重复读的重点是修改，同样的条件，你读取过的数据，再次读取出来发现值不一样了
> 幻读的重点在于新增或者删除，同样的条件，第 1 次和第 2 次读出来的记录数不一样

4. mysql常用的存储引擎，以及之间的区别

```
mysql> show engines;
+--------------------+---------+----------------------------------------------------------------+--------------+------+------------+
| Engine             | Support | Comment                                                        | Transactions | XA   | Savepoints |
+--------------------+---------+----------------------------------------------------------------+--------------+------+------------+
| InnoDB             | DEFAULT | Supports transactions, row-level locking, and foreign keys     | YES          | YES  | YES        |
| MRG_MYISAM         | YES     | Collection of identical MyISAM tables                          | NO           | NO   | NO         |
| MEMORY             | YES     | Hash based, stored in memory, useful for temporary tables      | NO           | NO   | NO         |
| BLACKHOLE          | YES     | /dev/null storage engine (anything you write to it disappears) | NO           | NO   | NO         |
| MyISAM             | YES     | MyISAM storage engine                                          | NO           | NO   | NO         |
| CSV                | YES     | CSV storage engine                                             | NO           | NO   | NO         |
| ARCHIVE            | YES     | Archive storage engine                                         | NO           | NO   | NO         |
| PERFORMANCE_SCHEMA | YES     | Performance Schema                                             | NO           | NO   | NO         |
| FEDERATED          | NO      | Federated MySQL storage engine                                 | NULL         | NULL | NULL       |
+--------------------+---------+----------------------------------------------------------------+--------------+------+------------+
```

常用的是InnoDB和MyISAM,看看区别
从上表我们可以看出

> InnoDB支持行锁MyISAM支持表锁
> InnoDB支持外健MyISAM不支持
> InnoDB支持事物MyISAM不支持
> InnoDB支持分布式事物XAMyISAM不支持
> InnoDB支持事物回滚点设置MyISAM不支持

5. 数据库中的锁有哪些
在讲解锁的时候建议给面试官说明自己使用的mysql的版本，事务的隔离级别，使用的存储引擎

> SELECT @@tx_isolation; 查看事务隔离级别
> show table status from xc_course \G 查看xc_course数据库中表使用的存储引擎
> 我们只针对默认隔离级别，存储引擎使用InnoDB

> 锁的分类
> 按照锁的粒度来划分
> 行锁:按照行的粒度对数据进行锁的,因此发生锁冲突概率底,可以实现并发度高，但是对于锁的开销比较大,加锁会比较慢
> 页锁:页锁就是在页的粒度上加锁,锁定的数据资源比行锁要多,开销介于行锁和表锁之间，会出现死锁，并发度一般
> 表锁:对数据表进行锁定
> MySQL中InnoDB支持行锁和表锁
> 从数据库管理的角度来划分
> 共享锁:也叫读锁或S锁，锁定的资源可以被其他用户读取，但是不能修改
> 排他锁:也叫独占锁，写锁或X锁，锁定的数据只允许进行锁定的事务使用，其他事务无法对已经锁定的数据查询或修改
> 意向锁:简单来说就是给更大一级别的空间示意里边是否已经上过锁
> 从程序员的角度划分
> 乐观锁:认为对同一数据的并发操作不会总发生，属于小概率事件，不采用数据库自身的锁机制，而是通过程序来实现，版本号机制，时间戳机制
> 悲观锁:对数据被其他事务的修改持保守态度，因此会通过数据库自身的锁机制来实现，从而保证数据操作的排他性
> 使用场景:乐观锁适合读比较多的场景，悲观锁适合写比较多的场景，需要特别注意悲观锁和乐观锁并不是锁，只是我的一种设计思想

```
1 DROP TABLE IF EXISTS `t`;
2 CREATE TABLE `t` (
3  `id` tinyint(10) unsigned NOT NULL AUTO_INCREMENT,
4  `name` varchar(10) NOT NULL,
5  PRIMARY KEY (`id`)
6 ) ENGINE=InnoDB AUTO_INCREMENT=8 DEFAULT CHARSET=utf8;
```

对于select语句，InnoDB不会加任何锁，对于insert、update、delete语句，InnoDB会自动的给涉及的数据加排他锁
InnoDB通过select * from table_name where ... lock in share mode添加共享锁(S) 为了模拟锁效果我们显示加锁
InnoDB通过select * from table_name where...for update添加排他锁(X) 为了模拟锁效果我们显示加锁
LOCK TABLE table_name READ;给表添加共享锁，表就变为了只读模式
UNLOCK TABLE;对共享锁解锁
LOCK TABLE product_comment WRITE;给表添加排他锁
UNLOCK TABLE;怕他锁解锁

锁效果演示

session1	**session2**
set autocommit = 0 ;	set autocommit = 0 ;
update t set name = 'r' where id = 11;	update t set name = 'r' where id = 11;
	等待中 Lock wait timeout exceeded

show engine innodb status \G 查看锁使用状态

```
---TRANSACTION 150308, ACTIVE 20783 sec
2 lock struct(s), heap size 1136, 1 row lock(s), undo log entries 1
MySQL thread id 63, OS thread handle 123145394417664, query id 439 localhost root cleaning up
Trx read view will not see trx with id >= 150308, sees < 150308
TABLE LOCK table `xc_course`.`t` trx id 150308 lock mode IX
RECORD LOCKS space id 1266 page no 3 n bits 72 index PRIMARY of table `xc_course`.`t` trx id 150308 lock_mode X locks rec but not gap
Record lock, heap no 3 PHYSICAL RECORD: n_fields 4; compact format; info bits 0
 0: len 1; hex 0b; asc  ;;
 1: len 6; hex 000000024b24; asc     K$;;
 2: len 7; hex 220000012d1001; asc "   -  ;;
 3: len 1; hex 63; asc c;;
```

结果图我们可以看出来
使用了table lock IX锁
行使用了 X锁
先解释为什么会使用X锁，如果是update mysql默认会给我们添加X锁
我们说InnoDB默认使用的是行锁为什么还会有table lock IX，这个需要解释一下
举个例子你可以给整个房子设置一个标识，告诉它里边有人，即使你只获取了房子中某一个房间的锁
其他人如果想要获取整个房子的控制权，只需要看这个房子的标识即可，不需要在对房子中的每个房间进行查找
对应我们的mysql来说，假如我们标中有10条数据，我们可以给表上添加一个IX锁，告诉其他的事务有其他事务
正在使用，这是意向锁IX会告诉其他事务已经有事务锁定了表中的某些记录，不能对整个表进行全表扫描
这就是为什么要在表上添加意向锁IX
其他的测试可以自己进行

6. 自增id用完怎么办

7. 自增主键为什么是不连续的

8. mysql支持存储过程吗？怎么写存储过程？jdbc怎么调用存储过程，mybatis怎么调用存储过程

> mysql支持存储过程

无参数存储过程

```
drop procedure if EXISTS productpricing
delimiter ;;
create procedure productpricing()
begin
   select * from t_post;
end
;;
delimiter ;
-- mysql控制台调用存储过程
call productpricing();
```

有参数存储过程

```
 drop PROCEDURE if EXISTS num_from_post
 delimiter &&
 create procedure num_from_post(in userID int,out postNumber int)
 comment '根据用户id，查看该用户的发帖数量'
 begin
   SELECT count(*) into postNumber from t_post where user_ID = userID;
 end &&
 delimiter ;
 -- mysql控制台调用存储过程
 call num_from_post(2,@postNumber)
 -- 输出存储过程结果
 select @postNumber;
```

jdbc调用存储过程

```
CallableStatement cs = conn.prepareCall("{call productpricing()}");
ResultSet rs = cs.executeQuery();
while(rs.hasNext()){
  //......
}
CallableStatement cs = conn.prepareCall("{call num_from_post(?,?)}");
cs.setInt(1, 2);
cs.registerOutParameter(2, Types.INTEGER);
cs.execute();
System.out.println(cs.getInt(2));
```

mybatis调用存储过程

```
<select id="call" statementType="CALLABLE" >
{
  call num_from_post(#{param1,mode=IN},#{param2,mode=OUT,jdbcType=INTEGER})
}
</select>

```

测试数据表

```
 SET NAMES utf8;
 SET FOREIGN_KEY_CHECKS = 0;
 -- ----------------------------
 --  Table structure for `t_post`
 -- ----------------------------
 DROP TABLE IF EXISTS `t_post`;
 CREATE TABLE `t_post` (
   `id` int(11) NOT NULL AUTO_INCREMENT,
   `title` varchar(50) DEFAULT NULL,
   `content` varchar(100) DEFAULT NULL,
   `createTime` datetime DEFAULT NULL,
   `user_ID` int(11) NOT NULL,
   PRIMARY KEY (`id`)
 ) ENGINE=InnoDB AUTO_INCREMENT=3 DEFAULT CHARSET=utf8;

 -- ----------------------------
 --  Records of `t_post`
 -- ----------------------------
 BEGIN;
 INSERT INTO `t_post` VALUES ('1', '杀人蜂', '最近杀人蜂连杀41人，好恐怖啊！~', '2017-05-16 14:40:07', '2'), ('2', '九寨沟井喷', '九寨沟人满为患，滞留旅客上千', '2017-05-16 14:40:07', '4');
 COMMIT;

 SET FOREIGN_KEY_CHECKS = 1;

```

9. mysql 常用的聚合函数

> min(),max(),count(),sum(),avg()

10. count(*)优化(mysql5.7.15,默认隔离级别，存储引擎InnoDB)

> 1)count() *在表很大的情况下会出现查询比较慢，可以把count()*的结果缓存下来

2.1)我们在demo表中查询id>100的数据
SQL语句如下所示
select count(*) from demo where id > 100;

```
+----------+
| count(*) |
+----------+
|  9999900 |
+----------+
1 row in set (15.82 sec)
```

耗费时间15.82
2.2)我们查询全表的数量
select count(*) from demo ;

```
+----------+
| count(*) |
+----------+
| 10000000 |
+----------+
1 row in set (10.58 sec)
```

耗费时间10.58
2.3)我们查询
select count(*) from demo where id <= 100;

```
+----------+
| count(*) |
+----------+
|      100 |
+----------+
1 row in set (0.01 sec)
```

2.4)我们可以把全表的总数查询出来,在把id<=100的数量查询出来
id > 100 = 总数 - id <= 100
2.5)SQL语句如下所示
select (select count() *from demo)* - *(select count()* from demo where id <= 100) as result;

```
+---------+
| result  |
+---------+
| 9999900 |
+---------+
1 row in set (5.35 sec)
```

2.6)我们可以看到结果相同，但是查询时间节省了许多
2.7)接着优化我们发现我们查询全表的时候时间还是比较长,我们可以通过添加索引再次优化,我们给demo表添加一个字段c,不存入数据,默认之为1
通过alter table demo add index (c);为c添加索引
2.8)设置好之后表结构如下所示

```
*********** 1. row ***************
       Table: demo
Create Table: CREATE TABLE `demo` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `name` varchar(100) DEFAULT NULL,
```

```
  `url` varchar(100) DEFAULT NULL,
  `address` varchar(100) DEFAULT NULL,
  `c` tinyint(4) NOT NULL DEFAULT '1',
  PRIMARY KEY (`id`),
  KEY `c` (`c`)
) ENGINE=InnoDB AUTO_INCREMENT=10000001 DEFAULT CHARSET=utf8
1 row in set (0.00 sec)
```

2.9)我们执行
select count(*) from demo where c > 0;

```
+----------+
| count(*) |
+----------+
| 10000000 |
+----------+
1 row in set (1.83 sec)
```

ok 仅仅用时1.83s
2.10)接着执行
select (select count() *from demo where c > 0) - (select count()* from demo where id <= 100) as result;

```
+---------+
| result  |
+---------+
| 9999900 |
+---------+
1 row in set (1.86 sec)
```

ok 结果相同时间更短
总结
主要利用了MySQL count()*对全表缓存查询快的特性,以及索引的特性*
count(字段)<count(主键id)<count(1)~count() 建议使用count()
主要是因为count()有缓存

11. limit优化

> select id,name,url,address from demo limit 0,10;
> 0表示起始页数，10表示每页显示的记录数
> 我们做如下测试起始页分别是下面的数据
> 0，10，100，1000，10000，100000，1000000
> select id,name,url,address from demo limit 0,10;
> select id,name,url,address from demo limit 10,10;
> select id,name,url,address from demo limit 100,10;
> select id,name,url,address from demo limit 1000,10;
> select id,name,url,address from demo limit 10000,10;
> select id,name,url,address from demo limit 100000,10;
> select id,name,url,address from demo limit 1000000,10;
> 我们观察耗费的时间在逐渐的增加
> 出现这种情况的原因是因为，limit是逐行扫描例如现在起始行是10000，需要查询出前10000行，然后从10000行之后在取10条记录出来，这样分页数越大查询越慢
> 1)从业务逻辑上优化,例如我们默认只能做79页分页,其他的数据忽略不计,可以查看百度和谷歌的分页来展示
> 2)我们可以跳过前10000行数据，然后在取10条出来,对应的SQL

> select id,name,url,address from demo where id > 10000 limit 10;
> 这样也是很快,原因是有索引,但是有一个问题,如果之前删除过一些数据,这样会造成分页数据不准确
> 该中方式必须是数据没有进行物理删除过
> 3)假如现在数据存在物理删除，还要不限制分页，可以使用延迟关联策略
> 思路：我们只查询id列
> select id from demo limit 1000000,10;
> 获取到10个id,我们在通过10个id分别查询出10条记录
> select demo.id,name,url,address from demo inner join (select id from demo limit 1000000,10) as tmp on demo.id = tmp.id;
> 或者使用子查询
> select demo.id,name,url,address from demo where id > (select id from demo limit 1000000,1) limit 10;

12. mysql普通索引创建

> alter table 表名 add index 索引名(列名) ;
> create index 索引名 on 表名(列名);

13. mysql索引优化(InnoDB,默认隔离级别)

> 索引的使用场景
> 1)数据库表中的数据比较少，不建议使用，比如数据少于1000条
> 2)数据重复读比较大的情况，不建议使用，比如给性别字段添加索引不一定查询效率会更好
> 测试代码
> SELECT id, name, hp_max, mp_max FROM heros_without_index WHERE name = '刘禅'
> SELECT id, name, hp_max, mp_max FROM heros_with_index WHERE name = '刘禅'
> 查看2条sql语句的运行结果
> SELECT * FROM user_gender WHERE user_gender = 1
> 索引的分类
> 普通索引:是基础的索引,没有任何约束,主要是为了提高查询效率
> 唯一索引:在普通索引的基础上加了unique,一张表中可以有多个
> 主键索引:在唯一索引的基础上添加了非空
> 全文索引:一般不用，用es，solr代替
> 聚集索引:也叫聚簇索引按照主键的纬度来排序存储数据,将数据存储与索引放到了一块,找到索引也就找到了数据，主键索引就是聚集索引一种，一张表中只能有一个聚集索引
> 非聚集索引:也叫非聚簇索引，二级索引，辅助索引，将数据存储于索引分开结构，索引结构的叶子节点指向了数据的对应行，myisam通过key_buffer把索引先缓存到内存中，当需要访问数据时（通过索引访问数据），在内存中直接搜索索引，然后通过索引找到磁盘相应数据，这也就是为什么索引不在key buffer命中时，速度慢的原因
> 单一索引:索引列为一列
> 联合索引:多列组合在一起的索引

14. mysql索引原理,为什么不实用hash

> b+tree
> hash,二叉树，BTree树，在MySQL中使用的是BTree树索引
> 我们来分析一下为什么不使用hash索引，hash值一般是唯一的，所以我们执行以下操作
> select * from demo where id =1000;
> 这样可以一次查找就可以找到需要的数据
> 假如现在要做这样的查询
> select * from demo where id > 1000;

> 因为hash值是不断变化的，所以任然需要全表扫描查找

15. 索引为什么不用二叉树

> 不使用二叉树的原因是二叉树一个节点最多只能有2个叶子节点，这样会造成二叉树深度很大，读取数据是IO次数过多
> 可以结合可视化数据来演示
> https://www.cs.usfca.edu/~galles/visualization/Algorithms.html 查看不同的数据结构图形演示

16. 开发中能不能使用join，join语句怎么优化

> 可以使用,必须要注意驱动表的选择
> 2张表的id和a字段都添加了索引,t1 100行 t2 1000行
> select * from t1 straight_join t2 on (t1.a=t2.a);
> 比如该条SQL的执行流程是这个样子的
> 执行流程
> 1)对驱动表t1做了全表扫描，需要扫描100行
> 2)扫描到的每一行需要根据a字段去t2表中查找，走的是索引树，因此每次都只扫描一次总共100次扫描
> 3)所以，整个流程扫描200次
> 这种根据索引扫描的做法也叫Index Nested-Loop Join算法
> 如果上边例子中a字段不是索引我们计算一下
> 执行流程
> 1)把表t1的数据读入到线程内存join_buffer中
> 2)扫描表t2,把表t2中的每一行取出来，跟join_buffer中的数据做比较，生成结果集
> 由于join_buffer是以无序的方式存储的，所以需要扫描t2表中的每一行数据
> 在内存中总过做了100*1000次扫描，效果不好这就是Block Nested-Loop Join 算法
> 从上边可知，优化方案如下
> 1)选择小表作为驱动表
> 2)on 之后的条件判断需要是索引
> 查看执行计划中查看Extra字段里边有没有出现Block Nested-Loop

17. 子查询怎么优化

> 自查询分类
> existis子查询
> 集合比较自查询 in any all some
> 子查询一定要使用索引列
> 例如 select * from user where id in (索引列)

18. 怎么判断sql语句需要优化

> 1)查看执行计划，根据执行计划的标准判断
> 2)根据业务需求，例如我们业务接受的想要速度是1s
> 3)开启慢查询日志

19. 慢查询日志怎么开启，以及作用

> 在mysql配置文件中添加
> log_output=file
> slow_query_log=on
> slow_query_log_file = /tmp/mysql-slow.log

> log_queries_not_using_indexes=on
> long_query_time = 1

20. 项目中有几台mysql服务器，是怎么搭建集群

21. 数据库表结构设计字段怎么优化

> 1)根据自己字段实际的长度，设置指定的长度
> 2)在字段比较复杂、易变动、不方便统一的情况下，建议使用健值对来存储

22. 表设计原则

> 1)总徇表设计3大范式
> 2)数据量叠加比较快的，需要考虑水平分表和分库，避免但表操作的性能瓶颈
> 3)可以适当冗余一些字段，减少join查询，创建一些中间表较少join查询
> 4)使用逻辑外健，减少锁表情况，和删除时数据校验
> 5)高并发情况下的查询，可以使用缓存替代数据库，提高并发性能

23. sql语句执行的顺序

> (1)from
> (3) join
> (2) on
> (4) where
> (5)group by(可以使用select中的别名，后面的语句中都可以使用)
> (6) avg,sum等等
> (7)having
> (8) select
> (9) distinct
> (10) order by
> (11) limit

24. 怎么查看sql语句的执行计划，执行计划中的每一个字段代表什么意思

> explain select * from .....
> id 1
> table:显示这一行的数据是关于哪一个表的
> 表名
> null 如select 1+1;
> 表别名
> derived
> type:显示连接使用了那种数据类型，从最好到最差的以此为
> const,system,null 常数查找，一般主键或者是唯一索引查找
> eq_reg 范围查找
> ref 连接查找，一个表基于一个索引的范围查找
> range 基于索引范围查找
> index 扫描索引数据查找
> ALL 扫描磁盘数据查找
> select_type:
> SIMPLE:除子查询或者 union 之外的其他查询
> primary:子查询中的最外层查询,注意并不是主键查询
> subquery:子查询内层查询的第一个select,结果不依赖于外部查询结果集

> derived:from型子查询
> union result:union中的合并结果
> union:union语句中第二个select开始的后面所有select,第一个select为primary
> dependent subquery:子查询中内层的第一个select,依赖于外部查询的结果集
> dependent union:子查询中的 union,且为 union 中从第二个 select 开始的后面所有select,同样依赖于外部查询的结果集
> uncacheable subquery:结果集无法缓存的子查询
> possible_keys:显示可能应用到这张表中的索引，如果为空，不可能用到索引
> key:实际使用的索引，如果为NULL，则没有使用索引
> key_len:索引的长度，在不损失精度的情况下，长度越短越好
> ref:显示索引的那一列被使用了，如果可能的话是一个常数
> rows:在表中查询时候必须检查的行数
> extra:一般需要注意的是前2个
> Using filesort:看到这个的时候MySQL需要优化了，MySQL需要进行额外的步骤来发现如何对返回得行排序，它更具连接类型以及存储排序键值和匹配条件的全部行指针来排序全部的行
> Using temporary:看到这个的时候MySQL需要优化了，MySQL需要创建一个临时的表来存储结果，这通常发生在对不同的列集进行order by上，而不是group by上
> range checked for each record:通过MySQL官方手册的描述,当MySQL Query Optimizer没有发现好的可以使用的索引的时候,如果发现如果来自前面的 表的列值已知,可能部分索引可以使用。对前面的表的每个行组合,MySQL 检查是否可以使用 range 或 index_merge 访问方法来索取行
> Using where:如果我们不是读取表的所有数据,或者不是仅仅通过索引就可以获取所有需要的数据,则会出现 Using where 信息
> Using index:所需要的数据只需要在index即可全部获得而不需要再到表中取数据
> Select tables optimized away 已经达到最优化的级别了，从缓存中获取数据，无需在优化

25. mysql配置文件怎么优化，做了哪些属性调整，为什么

> 内存调优设置
> have_query_cache 表示是否支持query cache
> query_cache_limit 表示query cache 存放的单条query的最大结果集，默认1M结果集大小超过最大不会被缓存
> query_cache_min_res_unit 每个结果集存放的最小内存，默认4K
> query_cache_size 系统用来query cache的内存大小
> query_cache_type 表示系统是否打开query cache功能
> InnoDB存储引擎参数设置
> InnoDB_buffer_pool_size 一般设置为服务器内存的80%
> max_connections mysql数据库最大连接数默认151
> thread_cache_size

26. 怎么做分库分表

27. order by的工作原理以及优化

28. 为什么表数据删掉一半了，表文件大小任然不变

> 删除数据是把数据标记为可覆盖，磁盘上的空间仍然存在，所以我们删除表中的数据之后需要执行
> alter table 表名 engine = InnoDB
> optimeze table 表名

29. 在开发中数据怎么备份

> 我们使用的shell脚本定时备份数据库

30. mysql集群数据同步原理

> binarylog 实现数据同步
> 主要是mysql的三个线程的分工合作。
> binlog线程：记录所有涉及到mysql数据变化的sql语句，将sql语句写入binlog中。
> io线程：当我们开启slave之后，slave服务器就监听从master服务器获取binlog内容，将获取的数据放进自己的relay log。
> sql执行线程：执行relay log里的sql语句，完成数据同步。

31. 主库出现问题了，从库怎么办

> 可以通过以下shell脚本将从库提升为主库

```
#!/bin/bash
echo -e ""
echo -e "\033[34m *****请输入本地数据库管理员账户***** \033[0m"

echo -e ""
echo -e -n "\033[34m 账户: \033[0m"
read user

echo -e ""
echo -e "\033[34m 密码: \033[0m \c"

while : ;do
    char=`
        stty cbreak -echo
        dd if=/dev/tty bs=1 count=1 2>/dev/null
        stty -cbreak echo
    `
    if [ "$char" = "" ];then
        echo
        break
    fi
    passwd_1="$passwd_1$char"
    echo -n "*"
done
echo -e ""
mysql -u$user -p$passwd_1 -e quit 2>/dev/null
if [ $? -ne 0 ];then
    echo -e "\033[31m 验证错误\033[0m"
    exit 1
else
    echo -e "\033[32m 验证成功\033[0m"
fi
#####从库提升为主库
echo -e ""
echo -e -n "\033[34m my.cnf的绝对路径:  \033[0m"
read myfile
echo -e ""
if [ ! -f $myfile ];then
    echo "文件不存在"
    exit 1
fi
```

```
echo -e ""
datename=$(date +%Y%m%d-%H%M%S)
new_myfile=$myfile.$datename.bak
cp $myfile $new_myfile
echo -e "\033[32m 备份文件为: $new_myfile\033[0m"
echo -e ""
read=`grep '^relay\|^read\|^log_slave_updates' $myfile`
if [ $? -ne 0 ];then
    echo -e -n "\033[31m 请手动检查$myfile只读功能是否关闭，程序继续请输入【y】，其他任意键退出: \033[0m"
    read read_1
    if [ $read_1 != "y" ];then
        exit 1
    fi
fi
read_2=($read)
for i in ${read_2[@]}
do
    sed -i "s/$i/#$i/g" $myfile
    echo -e "\033[32m $i 参数已注释 \033[0m"
done
echo -e ""
logbin=`grep '^log-bin=' $myfile`
if [ ! $logbin ]; then
    sed -i '/\[mysqld\]$/a\log-bin=mysql-bin' $myfile
    echo -e "\033[32m 开启log-bin=mysql-bin参数成功\033[0m"
else
    echo -e "\033[32m $logbin 参数已开启 \033[0m"
fi
echo -e ""
echo -e "\033[34m 正在重启数据库，请稍等... \033[0m"
#systemctl restart mariadb
echo -e ""
service stop mysqld 1>/dev/null 2>&1
if [ $? -ne 0 ];then
    echo -e "\033[31m 数据库关闭失败，程序退出\033[0m"
    exit 1
else
    echo -e "\033[32m 数据库关闭成功\033[0m"
fi
echo -e ""
service start mysqld 1>/dev/null 2>&1
sleep 2s
mysql -u$user -p$passwd_1 -e quit 2>/dev/null
if [ $? -ne 0 ];then
    echo -e "\033[31m 启动失败\033[0m"
    exit 1
else
    echo -e "\033[32m 启动成功\033[0m"
fi
echo -e ""
mysql -u $user -p$passwd_1 -e "stop slave io_thread;" 2>/dev/null
if [ $? -ne 0 ];then
    echo -e "\033[31m IO线程关闭失败\033[0m"
    mysql -u $user -p$passwd_1 -e "stop slave io_thread;"
    exit 1
```

```
97 else
98     echo -e "\033[32m I/O线程关闭成功\033[0m"
99 fi
100 echo -e ""
101 mysql -u $user -p$passwd_1 -e "show processlist\G" |grep "State: Slave has read all
    relay log; waiting for the slave I/O thread to update it"  1>/dev/null 2>&1
102 if [ $? -ne 0 ];then
103     echo -e "\033[31m 请检查中继日志是否更新完毕...\033[0m"
104     mysql -u $user -p$passwd_1 -e "show processlist\G"
105     exit 1
106 else
107     echo -e "\033[32m 中继日志更新完毕\033[0m"
108 fi
109 echo -e ""
110 mysql -u $user -p$passwd_1 -e "stop slave;" 2>/dev/null
111 if [ $? -ne 0 ];then
112     echo -e "\033[31m 主从同步关闭失败...\033[0m"
113     mysql -u $user -p$passwd_1 -e "stop slave;"
114     exit 1
115 else
116     echo -e "\033[32m 主从同步关闭成功\033[0m"
117 fi
118 echo -e ""
119 mysql -u $user -p$passwd_1 -e "reset master;"
120 if [ $? -ne 0 ];then
121     echo -e "\033[31m 切换主库失败...\033[0m"
122     mysql -u $user -p$passwd_1 -e "reset master;"
123     exit 1
124 else
125     echo -e "\033[32m 切换主库成功\033[0m"
126 fi
```

> 步骤
> 1.确保所有的relay log全部更新完毕，在每个从库上执行stop slave io_thread; show processlist;直到看到Has read all relay log,则表示从库更新都执行完毕了
> 2.登陆所有从库，查看master.info文件，对比选择pos最大的作为新的主库，这里我们选择192.168.1.102为新的主库
> 3.登陆192.168.1.102，执行stop slave; 并进入数据库目录，删除master.info和relay-log.info文件, 配置my.cnf文件，开启log-bin,如果有log-slaves-updates和read-only则要注释掉，执行reset master
> 4.创建用于同步的用户并授权slave，同第五大步骤
> 5.登录另外一台从库，执行stop slave停止同步
> 6.根据第七大步骤连接到新的主库
> 7.执行start slave;
> 8.修改新的master数据，测试slave是否同步更新

32. mysql数据量多大的时候才会考虑读写分离，集群，分库分表

> 百万级别的数据量

33. 常用的高性能连接池有哪些

数据库连接池	区别
HikariCP	快 优化并精简字节码 使用FastList替代ArrayList ConcurrentBag：更好的并发集合类

	实现
Druid	可扩展性，良好的监控管理连接方面做的比较好
C3P0	否
DBCP	否

34. date,datetime,timestamp和time类型的区别

名称	显示格式	显示范围	应用场景
Date	YYYY-MM-DD	1601-01-01 到9999-01-01	当业务需求中只需要精确到天时
DateTime	YYYY-MM-DD HH:mm:ss	1601-01-01 00:00:00 到 9999-12-31 23:59:59 与时区无关	当业务需求中需要精确到秒时
TimeStamp	YYYY-MM-DD HH:mm:ss	1970-01-01 00:00:00到2037年 时区转化	当业务需求中需要精确到秒或者毫秒时，或者该系统用于不同时区
Time	HH:mm:ss	00:00:00 到 23:59:59	当业务需求中只需要每天的时间

35. mysql支持数据类型

> int bigint varchar text date datetime等等

36. 待续.....

NoSQL数据库

redis数据库

1. redis数据存在哪里

> 内存中

2. redis是单线程还是多线程的

> 单线程

3. redis是单线程为什么还比较快

> 1)纯内存访问
> 2)非阻塞I/O,redis使用epoll作为I/O多路复用技术的实现，在加上redis自身的事件处理模型将epoll中的连接、读写、关闭都转换为事件，不在I/O上浪费过多的时间
> 3)单线程避免了线程切换和竞态产生的消耗

4. 在项目中哪里使用到了redis，解决什么问题

> 秒杀，存放库存数

5. redis常用的数据类型有哪些

> string list set zset hash

6. redis高级数据类型有哪些

> pipline bitmaps geo hyperloglog

7. redis string类型实现原理

> 首先字符串的长度不能超过512M，字符串对象的编码可以是int，raw或者embstr
> int：保存的是可以用 long 类型表示的整数值,当int编码保存的值不再是整数，或大小超过了long的范围时，自动转化为raw
> raw：保存长度大于44字节的字符串（redis3.2版本之前是39字节，之后是44字节）
> embstr：保存长度小于44字节的字符串（redis3.2版本之前是39字节，之后是44字节），在对embstr对象进行修改时，都会先转化为raw再进行修改，因此，只要是修改embstr对象，修改后的对象一定是raw的，无论是否达到了44个字节
> 注：embstr与raw都使用redisObject和sds保存数据，区别在于，embstr的使用只分配一次内存空间（因此redisObject和sds是连续的），而raw需要分配两次内存空间（分别为redisObject和sds分配空间）。因此与raw相比，embstr的好处在于创建时少分配一次空间，删除时少释放一次空间，以及对象的所有数据连在一起，寻找方便。而embstr的坏处也很明显，如果字符串的长度增加需要重新分配内存时，整个redisObject和sds都需要重新分配空间，因此redis中的embstr实现为只读。

```
//redisObject源码:
typedef struct redisObject{
    //类型
    unsigned type:4;
    //编码
    unsigned encoding:4;
    //指向底层数据结构的指针
    void *ptr;
    //引用计数
    int refcount;
    //记录最后一次被程序访问的时间
    unsigned lru:22;
}robj;
//sds源码:
struct sdshdr {
    // buf 中已占用空间的长度
    int len;
    // buf 中剩余可用空间的长度
    int free;
    // 数据空间
    char buf[];
};
```

8. redis常见的应用

> 缓存；排行榜(redis的zset数据类型)；计数器(redis的incr命令)；分布式锁(Redis的setnx功能)；消息系统(Redis提供了发布/订阅及阻塞队列功能)等；

9. redis怎么做持久化，区别是什么

> 1)RDB持久化是指在指定的时间间隔内将内存中的数据集快照写入磁盘，实际操作过程是fork一个子进程，先将数据集写入临时文件，写入成功后，再替换之前的文件，用二进制压缩存储
> 2)AOF持久化是以日志的形式记录Redis每一个写操作,将Redis执行过的所有写指令记录下来（读操作不记录），只许追加文件不可以改写文件，redis启动之后会读取appendonly.aof文件来实现重新恢复

> 数据，完成恢复数据的工作。默认不开启，需要将redis.conf中的appendonly no改为yes启动Redis
> 3)区别：
> RDB优点与缺点
> 优点：
> 如果要进行大规模数据的恢复，RDB方式要比AOF方式恢复速度要快。
> RDB可以最大化Redis性能，父进程做的就是fork子进程，然后继续接受客户端请求，让子进程负责持久化操作，父进程无需进行IO操作。
> RDB是一个非常紧凑(compact)的文件,它保存了某个时间点的数据集，非常适合用作备份，同时也非常适合用作灾难性恢复，它只有一个文件，内容紧凑，通过备份原文件到本机外的其他主机上，一旦本机发生宕机，就能将备份文件复制到redis安装目录下，通过启用服务就能完成数据的恢复。
> 缺点：
> 1RDB这种持久化方式不太适应对数据完整性要求严格的情况，因为，尽管我们可以用过修改快照实现持久化的频率，但是要持久化的数据是一段时间内的整个数据集的状态，如果在还没有触发快照时，本机就宕机了，那么对数据库所做的写操作就随之而消失了并没有持久化本地dump.rdb文件中。
> 2每次进行RDB时，父进程都会fork一个子进程，由子进程来进行实际的持久化操作，如果数据集庞大，那么fork出子进程的这个过程将是非常耗时的，就会出现服务器暂停客户端请求，将内存中的数据复制一份给子进程，让子进程进行持久化操作。
> AOF优点与缺点
> 优点：
> 1AOF有着多种持久化策略：appendfsync always:每修改同步，每一次发生数据变更都会持久化到磁盘上，性能较差，但数据完整性较好；appendfsync everysec: 每秒同步，每秒内记录操作，异步操作，如果一秒内宕机，有数据丢失；appendfsync no:不同步。
> 2AOF文件是一个只进行追加操作的日志文件，对文件写入不需要进行seek，即使在追加的过程中，写入了不完整的命令（例如：磁盘已满），可以使用redis-check-aof工具可以修复这种问题
> 3Redis可以在AOF文件变得过大时，会自动地在后台对AOF进行重写：重写后的新的AOF文件包含了恢复当前数据集所需的最小命令集合。整个重写操作是绝对安全的，因为Redis在创建AOF文件的过程中，会继续将命令追加到现有的AOF文件中，即使在重写的过程中发生宕机，现有的AOF文件也不会丢失。一旦新AOF文件创建完毕，Redis就会从旧的AOF文件切换到新的AOF文件，并对新的AOF文件进行追加操作。
> 4AOF文件有序地保存了对数据库执行的所有写入操作。这些写入操作一Redis协议的格式保存，易于对文件进行分析；例如，如果不小心执行了FLUSHALL命令，但只要AOF文件未被重写，通过停止服务器，移除AOF文件末尾的FLUSHALL命令，重启服务器就能达到FLUSHALL执行之前的状态
> 缺点：
> 1对于相同的数据集来说，AOF文件要比RDB文件大。
> 2根据所使用的持久化策略来说，AOF的速度要慢与RDB。一般情况下，每秒同步策略效果较好。不使用同步策略的情况下，AOF与RDB速度一样快
> 3选择使用哪种持久化方式?
> 1一般来说，如果想达到足以媲美PostgreSQL的数据安全性，应该同时使用两种持久化方式。
> 2有很多用户都只使用 AOF 持久化， 但我们并不推荐这种方式： 因为定时生成 RDB 快照（snapshot）非常便于进行数据库备份， 并且 RDB 恢复数据集的速度也要比 AOF 恢复的速度要快，除此之外， 使用 RDB 还可以避免之前提到的 AOF 程序的 bug。
> 3如果可以承受输分钟内的数据丢失，可以只使用RDB持久化

10. redis集群的搭建方式

> 3中方式 主从 哨兵 cluster

11. zset为什么要使用跳表来实现，为什么不用红黑树来实现

> 查询效率比红黑树高

12. redis常用命令有哪些

> https://github.com/wolvesleader/interview/tree/master/base_java/src/main/java/com/quincy/java/redis

13. redis优化

> 1)精简键值对大小，键值字面量精简，使用高效二进制序列化工具
> 2)数据优先使用整数，比字符串类型更节省空间
> 3)优化字符串使用，避免预分配造成的内存浪费

14. mangoDB和redis 的区别

15. redis内存回收策略

> volatile-lru：只对设置了过期时间的key进行LRU（默认值）
> allkeys-lru： 是从所有key里 删除 不经常使用的key
> volatile-random：随机删除即将过期key
> allkeys-random：随机删除
> volatile-ttl： 删除即将过期的
> noeviction： 永不过期，返回错误

16. 怎么使用BloomFilter解决缓存穿透

17. java怎么操作redis

> jedis springdataredis

18. 怎么把新的redis加入到集群中

> redis-trib.rb add-node 新节点ip：端口 任意节点ip：端口。 //新增master节点
> redis-trib.rb add-node --slave 新节点ip：端口 任意节点ip：端口。 //新增slave节点

19. 缓存穿透是什么？怎么解决

> 缓存穿透，是指查询一个数据库一定不存在的数据。正常的使用缓存流程大致是，数据查询先进行缓存查询，如果key不存在或者key已经过期，再对数据库进行查询，并把查询到的对象，放进缓存。如果数据库查询对象为空，则不放进缓存；如果我们在高并发的情况下查询一个一定不存在的key，在缓存中没有，在数据库也没有就会产生缓存穿透的问题；
> 解决办法：
> 1、如果查询数据库也为空，直接设置一个默认值存放到缓存，这样第二次到缓冲中获取就有值了，而不会继续访问数据库，这种办法最简单粗暴。
> 2、根据缓存数据Key的规则。在做缓存规划的时候，Key有一定规则的话，如果不符合规则就过滤掉，这样可以过滤一部分查询。这种办法只能缓解一部分的压力，过滤和系统无关的查询，但是无法根治。
> 3、采用布隆过滤器，将所有可能存在的数据哈希到一个足够大的BitSet中，不存在的数据将会被拦截掉，从而避免了对底层存储系统的查询压力。

20. 项目中使用reids是不是服务器内存越大会越好，为什么

> 不是，内存太大，查找的速度会降低

21. 在开发中，使用的redis服务器内存是多少

> 64G 128G 215G

22. redis集群中插槽有多少个？怎么计算

> Redis 集群中内置了 16384 个哈希槽。根据公式HASH_SLOT=CRC16(key) mod 16384，每个key通过CRC16校验后对16384取模来决定放置哪个槽

23. 缓存雪崩是什么？怎么解决

> 缓存雪崩是指缓存由于某些原因（比如 宕机、cache服务挂了或者不响应）整体crash掉了，导致大量请求到达后端数据库，从而导致数据库崩溃，整个系统崩溃；
> 解决办法：
> 1、采用加锁计数，或者使用合理的队列数量来避免缓存失效时对数据库造成太大的压力。这种办法虽然能缓解数据库的压力，但是同时又降低了系统的吞吐量。
> 2、分析用户行为，尽量让失效时间点均匀分布。避免缓存雪崩的出现。
> 3、如果是因为某台缓存服务器宕机，可以考虑做主备，比如：redis主备。

24. 待续.....

JMS中间件

ActiveMQ

1. 消息模式有哪些

> topic模式也叫发布订阅模式，一对多模式
> 队列模式也叫一对一模式

2. 项目中哪里使用到了ActiveMQ，为什么要用，不用不行吗

> 项目中使用到ActiveMQ的地方还比较多
> 电商项目支付系统支付成功后我们会发生消息，告知下单系统进行下单
> 通知库存管理系统对库存跟新，通知短信系统给用户发生短信
> 通知日志系统，记录用户购买记录
> 生成静态页面的时候获取数据和模版之后会通知页面生成系统生成静态
> 利用消息中间件实现分布式事务

3. 消息丢失怎么办

> 生产端，需要捕获消息发送的错误，并重发消息
> 存储阶段，可以通过配置刷盘和复制相关的参数，让消息写入到多个服务器中
> 消费端，需要处理完所有的业务之后在发送确认消息
> 把消息持久化到数据库中
> 关于怎么排查消息是否丢失可以使用链路追踪器相关的插件
> 或者在生产端给消息添加序号，在消费端根据需要来判断是否有消息丢失

4. 消息积压怎么排查

> 消息排查可以根据每个消息中间件自带的管理中心来排查
> 积压的解决方案
> 如果是业务比较着急不能做代码优化，可以考虑加机器
> 消费端慢就给消费端添加，生产端慢就给生产端加,一定要注意同步分区数量
> 如果没有服务器，就做服务降级，关闭掉一些不重要的业务逻辑
> 如果业务紧急可以排查代码
> 是否消息消费端存在大的消费，导致队列中其他消息无法消费

> 是否存在失败消费，不停的在尝试消费，造成队列中其他消息无法消费，可以把不误消费的消息放入到死信队列中，让其他消费继续进行
> 是否存在消费倾斜，大部分的消息都发生到集群中的某一台机器上

5. ActiveMQ中发布订阅如何设置偏移量

> 据我了解ActiveMQ中没有关于偏移量的设置

6. MQ如果要有返回值，如何处理

> 就是消息队列的rpc机制，为每个队列设置一个回调队列，从回调队列中回去消息的返回值
> 一般不建议这么使用，如果比较关注结果可以考虑使用rpc框架

7. 各种消息中间件对比
8. 待续.....

项目(结合具体项目分析)

1. 项目开发规范

> 命名规范
> 文档规范
> 包规范
> 接口规范

2. 项目开发人员

> 这是一个开放性话题
> 大公司15人
> 后台5人，运维1人，测试2人，前端4人，经理1人，美工2人
> 小公司8人
> 后台5人，前端3人

3. 项目架构和技术选型

> 第一套采用微服务架构
> 技术选型 springmvc spring mybatis dubbo zookeeper rabbitmq vue mysql
> 第二套采用微服务架构
> 技术选型 springboot springcloud mysql vue
> 第三套单项目
> 技术选型 springmvc spring mybatis springdatajpa mysql html

4. 项目开发周期

> 根据项目的大小一般是3-12个月必须发第一版，发完之后在做版本迭代

5. 项目迭代周期

> 10天一次这个每一个公司都不一样

6. jar包冲突怎么解决

> 通过maven工具排除

```
1 <exclusions>
```

```
    <exclusion>
        <groupId>org.apache.commons</groupId>
        <artifactId>commons-lang</artifactId>
    </exclusion>
</exclusions>

```

7. 项目介绍

> 1)介绍项目开发背景和客户群体，工期，开发人数，我负责的模块 例如
> 我上个项目是做实时交通规则显示的数据层展示，客户是汽车司机，工期是3个月，有5个人一起做，我在里边做的是页面静态化模块的开发
> 到这里第一部分介绍完毕，不要说太多，思路清晰，语言简洁就ok，让面试官有一个大体的了解
> 2)结合自己项目中实际的业务需求来讲 面试公司关注的技术，比如面试公司招聘信息中有springmvc rabbitmq等技术 例如
> 在一些特殊的场景下，我们的用户响应不能达到我们的要求，我们做了一些优化，把动态页面静态化，我们通过spring提供restful
> 写了一个页面静态化接口，请求接口携带需要静态化的数据id，把id发送到rabbitmq中，在消费端取出id，根据id生成具体的静态
> 页面，用rabbitmq主要是为了解耦合，提供服务端处理请求的数量
> 到此如果你有其他模块按照上边描述的方法说完，切记一定要结合这你的项目业务来讲技术点
> 3)如果项目中有亮点，给面试官讲亮点
> 介绍项目的时候一定要思路清晰，语言简洁

8. 项目负责模块

> 切记不要只说页面跳转，我们是后台开发工程师，页面跳转都做了什么事情需要讲清楚
> 1.注册登录模块
> 我先说用户的注册功能吧，用户进入首页点击注册会弹出一个注册表单，为了减少用户在注册时所填写的信息，在注册表单中只需要填写用户名，
> 密码，手机号码，邮箱等一些最基本的信息，剩下的信息可以在用户注册完成之后慢慢进行完善，用户注册的信息在后端还要进行一些完善，比
> 如设置用户的注册日期、用户的修改日期还有用户的密码在数据库中应当存的加密之后的密码而不是明文，因此我们要对接受过来的密码进行加
> 密，我们当时采用的是MD5加密，（在业务层调用DigestUtils.md5Hex(user.getPassword())这个工具类进行加密），用户在输入密码时有可能会
> 输错，所以在提交的时候在前端得进行校验，判断两次输入的密码是否一致，如果不一致给用户返回两次密码输入不一致的信息。
> 为了给用户一个更好的体验，用户在输入用户名我们采用的是一个异步请求的方式去数据库中查找该用户名是否可用，如果不可用返回一个用户
> 名不可用，如果可用返回一个用户名可用的信息，为了防止恶意注册，我们使用了短信验证，短信验证当时用的是第三方短信平台阿里大于，当时
> 也参考过别的短信平台，选择阿里大于是因为阿里大于收费便宜一条短信不到五分钱，还有一点是阿里大于是阿里云旗下的产品，大公司有保障，
> 使用阿里大于只需要申请一个签名，申请一个模板再创建一个Accesskey秘钥就可以使用了。（从阿里云通信官网下载demo工程将它导入项目），用
> 户输入完手机号点击发送验证码，向后端传递手机号，后端随机生成六位数验证码，在控制层判断从前端传过来的手机号格式是否合法，这个当时做
> 的时候公司有专门封装的工具类，调用这个工具类来判断用户输入的手机号是否合法，如果不合法直接返回信息告诉用户大陆的手机号是11位，香港
> 的手机号是8位，这个工具类不仅封装了可以判断大陆的手机号的方法还封装了判断香港的手机号码。

然后在业务层生成六位数验证码，当时我是通过
Math.random*1000000这个方法生成一个六位数验证码的，这个方法产生的是0-1之间的随机数乘以一百万就是一个六位数的验证码，将生成的验证码
存入缓存当中，以手机号为key以验证码为value，这里使用redis存储验证码是因为redis运行效率高，可以设置过期时间，因为验证码的有效时间是
60秒，如果存在数据库当中会存在很多垃圾数据。用户点击注册，后端根据用户提交过来的验证码与redis缓存中的验证码进行校验，如果相同则返回
信息注册成功，如果不同则返回验证码输入有误。注册时还添加了邮箱激活功能，用户注册成功后默认是未激活状态，需要用户去邮箱激活自己的账号。
用户在注册成功以后我们会向用户邮箱里面发送一个带有用户id的链接，并将这个id在用户注册成功之后存入redis缓存当中，以邮箱为key，以用户id
为value，设置过期时间为24小时，当用户点击邮箱中的链接时，后台程序会接受到这个用户id，然后拿这个id和用缓存当中的id值进行比较，如果不同
的话给用户一个激活失败的提示信息，如果相同的话告诉用户激活成功并将数据库中user表中status字段改成1（0为未激活状态，1位激活状态）我们注
册功能当时做的时候把注册功能和发送短信功能分离开来，当需要发送短信的时候只需要把短信的内容传给activemq，发送短信的微服务再从activemq中
取出消息然后再调用阿里大于发送短信，在将短信内容传给activemq时需要设置手机号，模板号，签名和参数等信息，activemq采用的是点对点的消息模式。
2.商品搜索模块
当用户访问购物网站的时候，用户可以根据自己所想的内容输入关键字就可以查询出相关的内容，商品搜索模块我们使用的是solr，solr是基于Luncence的全文搜索服务器，它提供了比Luncence更为丰富的语言，solr是在Tomcat环境下运行的，实现了可配置，可扩展，并对查询性能进行优化，有效降低访问数据库的频率减小对数据库造成的压力。同时还提供了一个完善的功能管理界面。
要使用solr首先得在solrhome中找到schema.xml文件，在schema.xml中配置IK分词器。，因为默认的分词不适合业务需要，举个例子我爱上海，默认的分词效果就是 我 爱 上 海，而IK分词器分词后是我 爱 上海；不光要配置IK分词器，还要根据具体的业务需要配置solr域，它相当于数据库中的表字段，每一种对应一种数据，一般分为基本域，复制域和动态域，使用复制域是因为在商品搜索中有可能既要在标题域中搜索又要在品牌域中搜索，因此将这些需要搜索的域配置到一个域当中；使用动态域是因为规格的值是不确定的，比如我搜手机时它出现的可能是机身内存、网络等，搜电视时它有可能是电视屏幕尺寸大小等，所以要在动态域中实现。
Solr域中常见的字段类型name 域的名称 type 域的类型、index 是否索引 store 是否存储 required 是否必须（相当于数据库中的非空） multivalued 是否多值
Solr能减少对数据库的压力是因为在我们搜索的时候不需要去数据库中查找，solr有索引库，我们只需要去索引库查找。新建一个模块将数据库中的数据导入到索引库，因为一般只使用一次所以不用将它添加到后台程序中，要注意两点，其一，因为规格是一个动态域，所以要在实体类当中添加一个specMap的字段并添加上Field注解，其二是在导入索引库当中的时候要注意添加一个条件，就是把status状态为1的导入进去，没有通过审核的不要导入到索引库当中，索引库中存放的字段id sku的id、title 标题、price 价格、image 图片地址、brand 品牌、seller 商家名称、category 商品分类、goodsid 商品spu的id其中图片地址是不需要搜索的所以index设置为false,还需要在实体类字段上添加Field注解，域名和实体类字段名相对应。
关键字搜索：打开搜索页面，在搜索框输入要搜索的关键字，点击搜索按钮即可进行搜索，展示搜索结果并进行高亮显示，高亮显示通过solrTemplate调用高亮显示特定的API，根据搜索关键字进行分组查询，通过solrTemplate.queryForGroupPage方法把查询到的商品分类信息显示在筛选条件栏上，从用户的体验来讲，如果搜索之后从数据库当中查询的话可能速度比较慢，因此这里使用了redis配合关系型数据库mysql做缓存，当用户进入商品分类页面时，将商品分类数据放入缓存（Hash）。以商品分类名称作为key ,以模板ID作为值；当用户进入模板管理页面时，分别将品牌数据和规格数据放入缓存（Hash）。以模板ID作为key,以品牌列表和规格列表作为值；
redisTemplate.boundHashOps("itemCat").put(itemCat.getName(),itemCat.getTypeId());

> 我们这个商城是仿京东的，为了更好的用户体验我们还做了过滤查询，大概就是点击搜索面板上的分类、品牌和规格，实现查询条件的构建，查询条件以面包屑的形式显示，当面包屑显示分类、品牌和规格时，要同时隐藏搜索面板对应的区域，用户可以点击面包屑上的X 撤销查询条件。撤销后显示搜索面包屑相应的内容；过滤查询中价格区间过滤有点特殊，我当时是把价格区间当成一个字符串从前端页面传到后端，后端然后通过-切割，然后根据切割后的结果分别查询他们的价格所对应的商品列表并将结果返回给前端页面。
> 我们在买东西的时候经常会使用多关键字搜索，多关键字搜索各个关键字是通过或的关系来搜索的，因为从运营的角度来讲希望可以改用户更多的选择，如果采用并的关系进行搜索极有可能查询到极少的记录甚至查询不到记录，多关键字查询还有一个智能的排序策略，就是按照关键字匹配度来进行排序，也就是说如果记录中同时包含了小米和手机，那么这部分数据会排列在前面显示，而只包含小米和只包含手机的记录会显示在后边。在关键字查询的时候还有一个小的细节就是用户有可能在输入关键字的时候中间有空格，这就有可能导致查询不到数据，因此在后端接受到用户查询的关键字时候得进行一些处理，把接受到的关键字中的空格全部替换成空字符串。
> 最后需要注意的是在商品审核后更新到solr索引库，或者在商品删除后删除solr索引库中的相应记录。

9. 项目难点

> 在做秒杀的时候最开始，用户量少的时候还可以，用户量大一些，会出现库存为负数，响应慢等问题
> 解决方案 分布式事务，分布式锁，redis

10. 项目亮点

> 视频上传我们采用了断点续传

11. 项目是否上线

> 上线，如果找的项目是已经上线了的
> 没有

12. 项目中有多少长表

> 1百多张表

13. 表怎么设计的

> 遵循表设计3大范式，为了减少join查询也会做一些字段冗余

14. 表与表之间的关系是什么样子的

> 根据自己的表关系描述

15. 线上项目出现问题，怎么调试，调试思路是怎么样子的

> 查看项目日志，使用工具远程调试项目

16. 你们系统里有大媒体资源都多大

> 一般是几十M

17. 媒体资源服务器带宽是多少

> 1G

18. 媒体资源是存在服务器上?你们服务器有多少个

> 3台

19. 数据库服务器有多少个

> 1台

20. 最大的媒体资源有多大，是以什么单位进行分割的

> 一般是几十M 以M来划分

21. 怎么实现断网后从之前的观看位置继续播放

> 记录播放时间点

22. 微信支付怎么对接的？都有哪些流程

> https://pay.weixin.qq.com/wiki/doc/api/native.php?chapter=6_5 参考微信官方文档

23. 第三方支付用过吗？对接的是什么

> 微信扫码支付，参考微信支付流程
> https://pay.weixin.qq.com/wiki/doc/api/native.php?chapter=6_5

24. 购物车怎么实现的

> 添加购物车解决方案一
> 点击加入购物车，把数据携带到后台
> 判断用户是否登录，没有登录在后台把数据加入到cookie，判断添加数据是否达到最大限制
> 没有添加，跳转到购物车页面，有跳转到登录页面提示用户登录
> 用户已登录，把cookie的数据和redis中的数据合并，跳转到购物车
> 添加购物车解决方案二
> 点击加入购物车，把数据携带到后台
> 判断用户是否登录，没有登录在后台生成随机key，把key和对应的数据写入到redis中，同时把key添加到cookie中
> 没有添加，跳转到购物车页面，有跳转到登录页面提示用户登录
> 用户已登录，把cookie的数据和redis中的数据合并，跳转到购物车

25. JWT中token 的生成和解密

> Json Web Token
> 组成 header 令牌类型 和使用的加密算法
> payload 负载 可以存放过期时间、签发者、等等信息也可以自定义字段
> 签名 防止负载或者内容被篡改 header + payload 通过加密算法
> 创建公钥和私钥根据创建公钥和私钥来生成JWT令牌

26. 权限认证中对称加密和非对称加密

> 主要是用来生成JWT令牌

27. 如何在保证数据库事务的前提下提高并发量

> 使用数据库乐观锁
> 修改数据库配置max_connection

28. token信息存在cookie中是在header还是body

> 存在header中

29. 待续.....

RPC中间件

dubbo

1. 什么是dubbo

> Dubbo是阿里巴巴公司开源的一个高性能优秀的服务框架，使得应用可通过高性能的 RPC 实现服务的输出和输入功能，可以和 Spring框架无缝集成

2. dubbo支持哪些协议

> rmi协议 hessian协议 http协议 webservice协议 thrift协议 redis协议 dubbo协议

3. dubbo怎么做负载均衡

> 内部提供负载均衡算法 RoundRobinLoadBlance RoundRobinLoadBlance LeastActiveLoadBlance ConsistentHashLoadBalance

4. dubbo服务降级怎么实现

> 服务消费端
> 1)容错
> mock=fail:return null
> 2)屏蔽
> mock=force:return null

5. 为什么用zookeeper做dubbo注册中心

> Zookeeper的数据模型很简单，有一系列被称为ZNode的数据节点组成，与传统的磁盘文件系统不同的是，zk将全量数据存储在内存中，可谓是高性能，而且支持集群，可谓高可用，另外支持事件监听。这些特点决定了zk特别适合作为注册中心

6. dubbo消息异步变同步实现原理

> com.alibaba.dubbo.rpc.protocol.dubbo.DubboInvoker#doInvoke

```
protected Result doInvoke(final Invocation invocation) throws Throwable {
     RpcInvocation inv = (RpcInvocation) invocation;
     final String methodName = RpcUtils.getMethodName(invocation);
     inv.setAttachment(Constants.PATH_KEY, getUrl().getPath());
     inv.setAttachment(Constants.VERSION_KEY, version);

     ExchangeClient currentClient;
     if (clients.length == 1) {
         currentClient = clients[0];
     } else {
         currentClient = clients[index.getAndIncrement() % clients.length];
     }
     try {
         boolean isAsync = RpcUtils.isAsync(getUrl(), invocation);
         boolean isOneway = RpcUtils.isOneway(getUrl(), invocation);
         int timeout = getUrl().getMethodParameter(methodName, Constants.TIMEOUT_KEY,
Constants.DEFAULT_TIMEOUT);
         if (isOneway) {//异步无返回值
```

```
18              boolean isSent = getUrl().getMethodParameter(methodName,
    Constants.SENT_KEY, false);
19              currentClient.send(inv, isSent);
20              RpcContext.getContext().setFuture(null);
21              return new RpcResult();
22          } else if (isAsync) {//异步有返回值
23              //获取返回值方法
24              //Future<String> temp = RpcContext.getContext().getFuture()
25              //temp.get();
26              ResponseFuture future = currentClient.request(inv, timeout);
27              RpcContext.getContext().setFuture(new FutureAdapter<Object>(future));
28              return new RpcResult();
29          } else {//异步 变同步，默认的方式
30              RpcContext.getContext().setFuture(null);
31              return (Result) currentClient.request(inv, timeout).get();
32          }
33      }
34  }
35  //com.alibaba.dubbo.remoting.exchange.support.DefaultFuture#get(int)
36  public Object get(int timeout) throws RemotingException {
37      if (timeout <= 0) {
38          timeout = Constants.DEFAULT_TIMEOUT;
39      }
40      if (!isDone()) {
41          long start = System.currentTimeMillis();
42          lock.lock();
43          try {
44              //关键点 判断resopnse是否为null如果为null，处于等待状态
45              while (!isDone()) {
46                  done.await(timeout, TimeUnit.MILLISECONDS);
47                  if (isDone() || System.currentTimeMillis() - start > timeout) {
48                      break;
49                  }
50              }
51          } catch (InterruptedException e) {
52              throw new RuntimeException(e);
53          } finally {
54              lock.unlock();
55          }
56          if (!isDone()) {
57              throw new TimeoutException(sent > 0, channel, getTimeoutMessage(false));
58          }
59      }
60      return returnFromResponse();
61  }
62  public boolean isDone() {
63      return response != null;
64  }
```

7. dubbo网络通信怎么做的

> netty

8. dubbo怎么连接zookeeper的

> zkclient

9. dubbo服务发布原理

> 1).暴露本地服务
> 2).暴露远程服务
> 3).启动netty
> 4).打开zk连接
> 5).到zk注册服务
> 6).监听zk

10. dubbo服务引用原理

> 把注册中心的服务转换为Invoker,再把Invoker转换为客户端的接口

11. dubbo怎么解决消息粘包

> 自定义协议来解决

13. 请求失败dubbo怎么处理，默认尝试连接几次

> 默认尝试3次

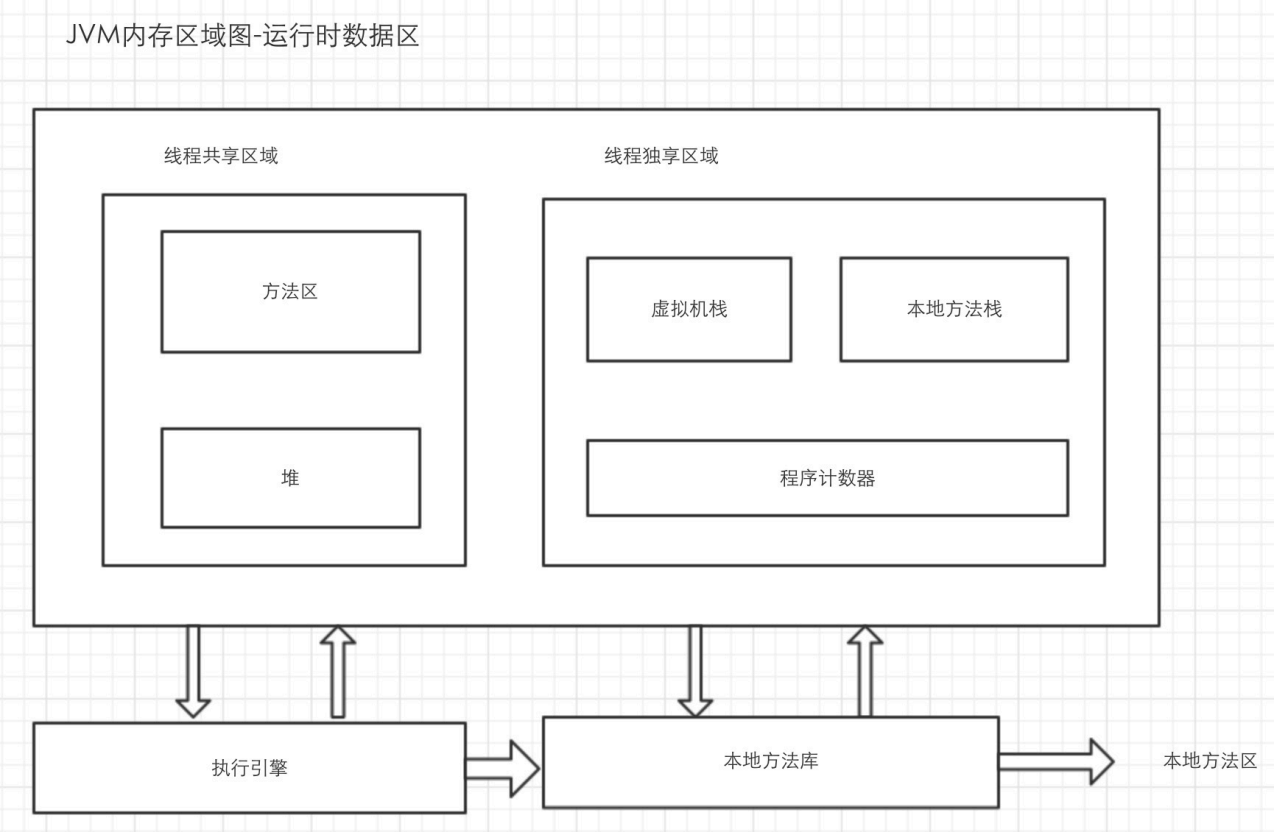
14. 服务调用链怎么监控

> SkyWalking

15. 待续.....

JVM知识

1. 运行时内存分布图



2. 线程安全的是哪些区域

> 看上边的图，线程独享的为安全

3. jmap是什么

> 查看线程栈工具

4. jvm怎么判断一个对象是垃圾

> 1)标记清楚法
> 2)可达性分析法

5. 垃圾回收算法有哪些

> 1)标记-清除算法
> 通过可达性分析法找到那些对象属于可回收对象之后，对他们进行标记，垃圾回收器判断对象是否被打标记，如果有清除。
> 标记清除法的缺点
> 会造成大量的内存碎片
> 2)标记-整理算法
> 标记过程和标记-清除是相同的,后续不是直接删除垃圾对象,而是将存活的对象移动到内存的一端,然后直接清除另外一端
> 整理内存比较耗时
> 3)分代收集算法
> Young区使用复制算法
> old区用标记清除或者标记整理算法
> 4)复制算法
> 将可用的内存根据容量分为大小相等的2块,每次只使用一块当其中的一块用完之后,将还存活这的对象复制到另外一块上，在把使用过的内存空间一次全部清除掉
> 运行高效，空间使用率底

6. 对象在内存中的分配策略

> 1)优先分配到eden区
> 2)大对象直接分配到老年代(什么是大对象呢？一般是指对象所占用的空间比新生代的空间要大)
> 3)长期存活的对象分配到老年代
> 4)动态对象年龄判断
> 5)空间分配担保
> 6)逃逸分析与栈上分配

7. 在项目中你们配置最大内存，最小内存是多少

> 最大和最小都是1024M

8. 类加载器有哪些

> BootstrapClassLoader
> ExtensionsClassLoader
> SystemClassLoader

9. jvm垃圾回收器有哪些，使用场景

> 新生代收集器：Serial、ParNew、Parallel Scavenge
> 老年代收集器：CMS、Serial Old、Parallel Old
> 整堆收集器： G1
> Serial

> 特点：单线程、简单高效（与其他收集器的单线程相比），对于限定单个CPU的环境来说，Serial收集器由于没有线程交互的开销，专心做垃圾收集自然可以获得最高的单线程手机效率。收集器进行垃圾回收时，必须暂停其他所有的工作线程，直到它结束（Stop The World）。
> 应用场景：适用于Client模式下的虚拟机。
> ParNew
> 特点：多线程、ParNew收集器默认开启的收集线程数与CPU的数量相同，在CPU非常多的环境中，可以使用-XX:ParallelGCThreads参数来限制垃圾收集的线程数,和Serial收集器一样存在Stop The World问题
> 应用场景：ParNew收集器是许多运行在Server模式下的虚拟机中首选的新生代收集器，因为它是除了Serial收集器外，唯一一个能与CMS收集器配合工作的。

10. 怎么配置最小内存

> -Xms10240m

11. 怎么配置最大内存

> -Xmx10240m

12. 怎么开启指定的垃圾回收器

> -XX:+UseSerialGC，虚拟机运行在Client模式下的默认值，Serial+Serial Old。
> -XX:+UseParNewGC，ParNew+Serial Old，在JDK1.8被废弃，在JDK1.7还可以使用。
> -XX:+UseConcMarkSweepGC，ParNew+CMS+Serial Old。
> -XX:+UseParallelGC，虚拟机运行在Server模式下的默认值，Parallel Scavenge+Serial Old(PS Mark Sweep)。
> -XX:+UseParallelOldGC，Parallel Scavenge+Parallel Old。
> -XX:+UseG1GC，G1+G1。

13. 待续.....

版本控制工具

git

1. 版本出现冲突怎么办

> 保留正确的版本，删除不需要的版本

2. 回退到之前版本

> git reflog
> git reset --hard id

3. 分支合并

> git merge 分支

4. gitlab和git区别

> gitlab是git的图形化服务

5. git和svn区别

> git分布式 svn不是

6. 创建分支

   > git branch 分支

7. 待续.....

项目构建工具

maven

1. 怎么引入jar包

   > 通过坐标

2. oracle驱动包怎么打入到项目中

   > oracle的jar包在中央仓库没有，所以我们必须手动打入到本地仓库
   > mvn install:install-file -DgroupId=com.oracle -DartifactId=ojbdc6-Dversion=11.2.0.1.0 -Dpackaging=jar -Dfile=D:\repository\ojbdc6.jar

3. 待续.....

linux操作

1. 常用的命令
2. 服务器与服务器之间怎么传输文件

   > scp 源文件位置 root@目标ip:/目标位置

3. 服务器时间怎么同步

   > 设置时间服务器，其他服务器和时间服务器同步时间

4. 查看系统负载

   > top

5. 怎么安装文件

   > yum

6. tar -zxvf zxvf分别是什么意思

   > z tar包文件压缩格式
   > x 从 tar 包中把文件提取出来
   > v 显示进度
   > f 指定被处理的文件是 xxx.tar.gz

7. 查看占用某端口的进程

   > netstat -tunpl | grep 端口号

8. 查看某进程监听的端口

   > netstat -lantp | grep -i listen

9. 待续.....

设计模式

单例模式

1. 实现思路分析
2. 具体实现
3. 在项目中怎么应用
4. 是否线程安全？为什么是线程安全的，为什么不是线程安全的
5. 怎么实现延迟加载线程安全的单例模式

> https://github.com/wolvesleader/interview/tree/master/base_java/src/main/java/com/quincy/java/designpatterns/singleton

动态代理

1. 动态代理实现原理
2. 动态代理在项目中哪里使用了，为什么要用
3. 动态代理回调函数中的proxy对象是什么

> https://github.com/wolvesleader/interview/tree/master/base_java/src/main/java/com/quincy/java/designpatterns/proxy

分布式锁

1. 实现思路分析
2. 具体实现

> https://github.com/wolvesleader/interview/tree/master/zookeeper-demo/src/main/java/com/quincy/curator

分布式ID生成器

1. 实现思路分析
2. 具体实现

> https://github.com/wolvesleader/interview/tree/master/zookeeper-demo/src/main/java/com/quincy/curator

常见算法

1. 常见的排序算法
2. 加权轮训算法
3. 二分查找
4. 模拟队列

5. 模拟栈

> https://github.com/wolvesleader/interview/tree/master/base_java/src/main/java/com/quincy/java/algorithm