java基础

Java基本类型哪些，所占字节

- byte： 1个字节
- char： 2个字节
- int： 4个字节
- long： 8个字节
- float： 4个字节
- double： 8个字节

java集合以及底层原理

Java集合框架的根接口有Collection和Map。Collection根接口包含List和Set二个子接口。

List接口

它的特点是：元素有序、且可重复，主要包含三个实现类：ArrayList，vector，LinkedList

ArrayList的特点：底层是数组，线程不安全，查找快，增删慢（数组的特点）

vector的特点：古老的实现类,底层是数组,线程安全的,JDK1.0就有了,Vector总是比ArrayList慢,所以尽量避免使用。

LinkedList的特点：底层是使用双向链表。增删快，查找慢。

ArrayList的底层实现原理：通过ArrrayList空参构造器创建对象。
底层创建一个长度为10的数组，当我们向数组中添加11个元素时，底层会进行扩容，扩容为原来的1.5倍
(创建一个新的数组，长度为原数组长度的1.5倍，将原数组复制到新数组中)。

Set接口

它的特点：

无序性：通过HashCode方法算出的值来决定在数组中存放的位置；

不可重复性：进行equals方法比较，结果为true则两个数据相同，若为false则不同.

主要包含三个实现类：HashSet，LinkedHashSet，TreeSet

HashSet特点 线程不安全，集合元素可以为null，不能保证元素的排列顺序

HashSet的底层实现原理：

当向HashSet添加数据时，首先调用HashCode方法决定数据存放在数组中的位置，该位置上没有其他元
素，则将数据直接存放，若该位置上有其他元素，调用equals方法进行比较。若返回true则认为两个数据相

同，若返回false，则以链表的形式将该数据存在该位置上，(jdk1.8)如果数量达到8则将链表换成红黑树。
HashSet的底层就是一个HashMap,向HashSet中添加的数据实际上添加到了HashMap中的key里。所以HashMap的key可以看成是Set的集合。

LinkedHashSet特点

继承了HashSet，底层实现原理和HashSet一样,可以安照元素添加的顺序进行遍历，根据元素的hashCode值来决定元素的存储位置，它维护了一张链表该链表记录了元素添加的顺序。底层就是一个LinkedHashMap。

TreeSet特点:

底层为红黑树；可以安照指定的元素进行排序；TreeSet中的元素类型必须保持一致，底层就是TreeMap。TreeSet必须（自然排序）实现Comparable接口，重写compareTo()方法，按照某个属性进行排序，相结合添加元素或（定制排序）创建一个Comparator实现类的对象，并传入到TreeSet的构造器中，按照某个属性进行排序，向集合添加元素。定制排序比自然排序灵活。 如果即有自然排序又有定制
排序谁起作用？ 定制排序

Map接口

Map的特点:

Map存储的是键值对(key,value)，Map中的key是无序的且不可重复的，所有的key可以看成是一个set集合。Map中的key如果是自定义类的对象必须重写hashCode和equals方法，Map中的value是无序的可重复的，所有的value可以看成是Collection集合，Map中的value如果是自定义类的对象必须重equals方法，Map中的键值对可以看成是一个一个的Entry.Entry所存放的位置是由key来决定的。Entry是无序的不可重复的。

主要的实现类：HashMap，LinkedHashMap，TreeMap，HashTable

HashMap特点

1. 底层是一个数组 + 链表 + 红黑树(jdk1.8)
2. 数组的类型是一个Node类型
3. Node中有key和value的属性
4. 根据key的hashCode方法来决定Node存放的位置
5. 线程不安全的 ,可以存放null

HashMap的底层实现原理:

当我们向HashMap中存放一个元素(k1,v1),先根据k1的hashCode方法来决定在数组中存放的位置。如果该位置没有其它元素则将(k1,v1)直接放入数组中,如果该位置已经有其它元素(k2,v2),调用k1的equals方法和k2进行比较。如果结果为true则用v1替换v2,如果返回值为false则以链表的形式将(k1,v1)存放,当元素达到8时和数组大于64则会将链表替换成红黑树以提高查找效率。

HashMap的构造器:

new HashMap() :创建一个容量为16的数组，加载因子为0.75。

当我们添加的数据超过12时底层会进行扩容，扩容为原来的2倍。

LinkedHashMap

继承了HashMap底层实现和HashMap一样.

可以按照元素添加的顺序进行遍历底层维护了一张链表用来记录元素添加的顺序。

TreeMap特点：可以对Key中的元素安照指定的顺序进行排序 （ 不能对value进行排序）

HashTable特点：线程安全的 ,不可以存放null，map中的key不能重复，如果有重复的，后者的value覆盖
前者的value

equals与==的区别?

==：如果==两边是基本数据类型，那么比较的是具体的值。如果==两边是引用数据类型，那么比较的是地
址值。(两个对象是否指向同一块内存)

equals：如果没有重写equals方法那么调用的是Object中的equals方法，比较的是地址值。如果重写了euqlas方法(比属性内容)那么就比较的是对象中属性的内容。

抽象类和接口及普通类的区别?

1. 抽象类和接口都不能直接实例化，如果要实例化，抽象类变量必须指向实现所有抽象方法的子类对象，接口变量必须指向实现所有接口方法的类对象。
2. 抽象类要被子类继承，接口要被类实现。
3. 接口只能做方法申明，抽象类中可以做方法申明，也可以做方法实现
4. 接口里定义的变量只能是公共的静态的常量，抽象类中的变量是普通变量。
5. 抽象类里的抽象方法必须全部被子类所实现，如果子类不能全部实现父类抽象方法，那么该子类只能是抽象类。
6. 抽象方法只能申明，不能实现，接口是设计的结果 ， 抽象类是重构的结果
7. 抽象类里可以没有抽象方法
8. 如果一个类里有抽象方法，那么这个类只能是抽象类
9. 抽象方法要被实现，所以不能是静态的，也不能是私有的。
10. 接口可继承接口，并可多继承接口，但类只能单根继承。

堆和栈的区别

一.堆栈空间分配区别：

1. 栈（操作系统）：由操作系统自动分配释放 ，存放函数的参数值，局部变量的值等。其操作方式类似于数据结构中的栈；
2. 堆（操作系统）： 一般由程序员分配释放， 若程序员不释放，程序结束时可能由OS回收，分配方式倒是类似于链表。

二.堆栈缓存方式区别：

1.栈使用的是一级缓存， 他们通常都是被调用时处于存储空间中，调用完毕立即释放；

2.堆是存放在二级缓存中，生命周期由虚拟机的垃圾回收算法来决定（并不是一旦成为孤儿对象就能被回
收）。

所以调用这些对象的速度要相对来得低一些。

三.堆栈数据结构区别：

堆（数据结构）：堆可以被看成是一棵树，如：堆排序；

栈（数据结构）：一种先进后出的数据结构。

值传递和引用传递的区别?

值传递：会创建副本，函数中无法改变原始对象

引用传递：不会创建副本，函数中可以改变原始对象

值传递：方法调用时，实际参数把它的值传递给对应的形式参数，方法执行中形式参数值的改变不影响实际参数的值。

引用传递：也称为传地址。方法调用时，实际参数的引用(地址，而不是参数的值)被传递给方法中相对应的形式参数，

在方法执行中，对形式参数的操作实际上就是对实际参数的操作，方法执行中形式参数值的改变将会影响实际参数的值。

4种访问控制符区别?

访问权限	类	包	子类	其他包
public	v	v	v	v
protect	v	v	v	×
default	v	v	×	×
private	v	×	×	×

阻塞和非阻塞以及同步和异步的区别?

1. 同步，就是我调用一个功能，该功能没有结束前，我死等结果。
2. 异步，就是我调用一个功能，不需要知道该功能结果，该功能有结果后通知我（回调通知）
3. 阻塞，就是调用我（函数），我（函数）没有接收完数据或者没有得到结果之前，我不会返回。
4. 非阻塞，就是调用我（函数），我（函数）立即返回，通过select通知调用者

同步IO和异步IO的区别就在于：数据拷贝的时候进程是否阻塞

阻塞IO和非阻塞IO的区别就在于：应用程序的调用是否立即返回

线程的sleep和wait区别?

1. wait()可以用notify()直接唤起.
2. sleep()不释放同步锁,wait()释放同步锁.
3. 这两个方法来自不同的类分别是Thread和Object
4. sleep可以用时间指定来使他自动醒过来,如果时间不到你只能调用interreput()来强行打断;
5. 最主要是sleep方法没有释放锁，而wait方法释放了锁，使得其他线程可以使用同步控制块或者方法。
6. wait，notify和notifyAll只能在同步控制方法或者同步控制块里面使用，而sleep可以在任何地方使用

线程的状态

创建、就绪、运行、阻塞、终止

1. 新建状态(New)：新创建了一个线程对象。
2. 就绪状态(Runnable)：线程对象创建后，其他线程调用了该对象的start()方法。该状态的线程位于“可运行线程池”中，变得可运行，只等待获取CPU的使用权。即在就绪状态的进程除CPU之外，其它的运行所需资源都已全部获得。

3. 运行状态(Running)：就绪状态的线程获取了CPU，执行程序代码。
4. 阻塞状态(Blocked)：阻塞状态是线程因为某种原因放弃CPU使用权，暂时停止运行。直到线程进入就绪状态，才有机会转到运行状态。

三种遍历方式

第一种遍历方法和输出结果。

```
for(int i=1,i<list.size(),i++){
    System.out.println(list.get(i));
}
```

第二种用foreach循环。加强型for循环。推荐方式

```
for(String string:list){
    System.out.println(string);
}
```

第三种迭代器

```
List<String> list=new ArrayList<>();
list.add("abc");
list.add("ghi");
for(Iterator<String> it=list.iterator();it.hasNext();){
      System.out.println(it.next());
}
```

讲讲线程的创建及实现线程几种方式之间的区别

1. 继承Therad类
2. 实现Runnable接口
3. 实现Callable接口
4. 使用线程池

继承Thread类，并重写里面的run方法

```
class A extends Thread{
    public void run(){
        for(int i=1;i<=100;i++){
            System.out.println("------------------"+i);
        }
    }
}
A a = new A();
a.start();
```

实现Runnable接口，并实现里面的run方法

```
class B implements Runnable{
    public void run(){
        for(int i=1;i<=100;i++){
            System.out.println("-----------------"+i);
        }
    }
}
B b = new B();
Thread t = new Thread(b);
t.start();
```

实现Callable

```
class A implements Callable<String>{
    public String call() throws Exception{
        //...
    }
}
FutureTask<String> ft = new FutureTask<>(new A());
new Thread(ft).start();
```

线程池

```
ExcutorService es = Executors.newFixedThreadPool(10);
es.submit(new Runnable(){//任务});
es.submit(new Runnable(){//任务});
...
es.shutdown();
```

实现Runnable和实现Callable的区别?

实现Callable接口，任务可以有返回值，Runnable没有。

实现Callable接口，可以指定泛型，Runnable没有

实现Callable接口，可以在call方法中声明异常，Runnable没有

Runnable和Thread二者的区别

实现Runnable接口的方式，更适合处理有共享资源的情况。

实现Runnable接口的方式，避免了单继承的局限性。

Java自定义类加载器与双亲委派模型

启动类加载器（Bootstrap）C++

扩展类加载器（Extension）Java

应用程序类加载器（AppClassLoader）Java

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

为了安全，防止用户自己创建系统类，比如String

讲讲什么是死锁，怎么解决死锁，表级锁和行级锁，悲观锁与乐观锁以及线程同步锁区别

死锁：打个比方，你去面试，面试官问你，你告诉我什么是死锁我就让你进公司。你回答说你让我进公司我就告诉你什么是死锁

讲讲JUC的辅助类

ReentrantReadWriteLock：读写锁

CountDownLatch：减少计数

CyclicBarrier：循环栅栏

Semaphore：信号灯

SSM框架面试问题

讲下springmvc框架的工作流程

1. 用户向服务器发送请求，请求被SpringMVC的前端控制器DispatcherServlet截获。
2. 、DispatcherServlet对请求的URL（统一资源定位符）进行解析，得到URI(请求资源 标识符)，然后根据该URI，调用HandlerMapping获得该Handler配置的所有相关的对 象，包括Handler对象以及Handler对象对应的拦截器，这些对象都会被封装到一个 HandlerExecutionChain对象当中返回。
3. DispatcherServlet根据获得的Handler，选择一个合适的HandlerAdapter。 HandlerAdapter的设计符合面向对象中的单一职责原则，代码结构清晰，便于维护，最 为重要的是，代码的可复制性高。HandlerAdapter会被用于处理多种Handler，调用 Handler实际处理请求的方法。
4. 提取请求中的模型数据，开始执行Handler(Controller)。在填充Handler的入参过程 中，根据配置，spring将帮助做一些额外的工作

   消息转换：将请求的消息，如json、xml等数据转换成一个对象，将对象转换为指定的响应信息。数据转换：对请求消息进行数据转换，如String转换成Integer、Double等。 数据格式化：对请求的消息进行数据格式化，如将字符串转换为格式化数字或格式化日 期等。 数据验证：验证数据的有效性如长度、格式等，验证结果存储到BindingResult或Error 中。
5. Handler执行完成后，向DispatcherServlet返回一个ModelAndView对象， ModelAndView对象中应该包含视图名或视图模型。
6. 根据返回的ModelAndView对象，选择一个合适的ViewResolver(视图解析器)返回给DispatcherServlet。
7. ViewResolver结合Model和View来渲染视图。
8. 将视图渲染结果返回给客户端。

以上8个步骤，DispatcherServlet、HandlerMapping、HandlerAdapter和 ViewResolver等对象协同工作，完成SpringMVC请求—>响应的整个工作流程，这些对 象完成的工作对于开发者来说都是不可见的，开发者并不需要关心这些对象是如何工作 的，开发者，只需要在Handler(Controller)当中完成对请求的业务处理。

图片怎么上传

前端实现异步上传，后端使用springmvc的MultipartFile类型来接收，放到分布式图片 服务器中，服务器返回图片路径把路径返回页面回显图片，开发或者测试环境可以使用 FastDFS

spring框架AOP执行原理简单说下？还有就是AOP在事务管理方面 是怎么实现的?

Spring AOP使用的动态代理，所谓的动态代理就是说AOP框架不会去修改字节码，而 是在内存中临时为方法生成一个AOP对象，这个AOP对象包含了目标对象的全部方法， 并且在特定的切点做了增强处理，并回调原对象的方法。 Spring AOP中的动态代理主要有两种方式，JDK动态代理和CGLIB动态代理。JDK动态 代理通过反射来接收被代理的类，并且要求被代理的类必须实现一个接口。JDK动态代 理的核心是InvocationHandler接口和Proxy类。如果目标类没有实现接口，那么Spring AOP会选择使用CGLIB来动态代理目标类。CGLIB（Code Generation Library），是 一个代码生成的类库，可以在运行时动态的生成某个类的子类，注意，CGLIB是通过继 承的方式做的动态代理，因此如果某个类被标记为final，那么它是无法使用CGLIB做动 态代理的。 AOP在事务管理方面，Spring使用AOP来完成声明式的事务管理有annotation和xml两 种形式。开发中，方便代码编写，很多时候都是在spring配置文件中配置事务管理器并 开启事务控制注解。在业务类或业务类方法中添加@Transactional实现事务控制。

Spring 分布式事务如何处理的?

第一种方案：可靠消息最终一致性，需要业务系统结合MQ消息中间件实现，在实现过 程中需要保证消息的成功发送及成功消费。即需要通过业务系统控制MQ的消息状态 第二种方案：TCC补偿性，分为三个阶段TRYING-CONFIRMING-CANCELING。每个 阶段做不同的处理。

TRYING阶段主要是对业务系统进行检测及资源预留

CONFIRMING阶段是做业务提交，通过TRYING阶段执行成功后，再执行该阶段。默认 如果TRYING阶段执行成功，CONFIRMING就一定能成功。

CANCELING阶段是回对业务做回滚，在TRYING阶段中，如果存在分支事务TRYING失 败，则需要调用CANCELING将已预留的资源进行释放。

Springboot用过没，跟我说说，他的特点?

Springboot是从无数企业实战开发中总结出来的一个更加精炼的框架，使得开发更加简单，能使用寥寥数行代码，完成一系列任务。

Springboot解决那些问题

1. 编码更简单
    1. Spring框架由于超重量级的XML，annotation配置，使得系统变得很笨重，难 以维护
    2. Springboot采用约定大于配置的方法，直接引入依赖，即可实现代码的开发
2. 配置更简单
    1. Xml文件使用javaConfig代替，XML中bean的创建，使用@bean代替后可以直接注入。 配置文件变少很多，就是application.yml
3. 部署更简单
4. 监控更简单

Spring-boot-start-actuator:

可以查看属性配置

线程工作状态

环境变量

JVM性能监控

支付接口是怎么做的?

微信支付

调用微信的支付接口，参考微信提供的api 使用了微信的统一下单接口和查询支付状态接口 每个接口需要的参数放入到map中使用微信提供的sdk转成XML字符串，httpClient远程 提交参数和接收结果。

什么是 Spring Boot?

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。

为什么要用 Spring Boot?

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

Spring Boot 的核心配置文件有哪几个？它们的区别是什么?

Spring Boot 的核心配置文件是 application 和 bootstrap 配置文件。

application 配置文件这个容易理解，主要用于 Spring Boot 项目的自动化配置。

bootstrap 配置文件有以下几个应用场景。

· 使用 Spring Cloud Config 配置中心时，这时需要在 bootstrap 配置文件中添加连接到 配置中心的配置属性来加载外部配置中心的配置信息； · 一些固定的不能被覆盖的属性； · 一些加密/解密的场景

Spring Boot 的配置文件有哪几种格式？它们有什么区别?

.properties 和 .yml，它们的区别主要是书写格式不同。

Spring Boot 的核心注解是哪个？它主要由哪几个注解组成的?

启动类上面的注解是@SpringBootApplication，它也是 Spring Boot 的核心注解，主 要组合包含了以下 3 个注解:

@SpringBootConfiguration：组合了 @Configuration 注解，实现配置文件的功能。

@EnableAutoConfiguration：打开自动配置的功能，也可以关闭某个自动配置的选 项，如关闭数据源自动配置功能： @SpringBootApplication(exclude = { DataSourceAutoConfiguration.class })。
@ComponentScan：Spring组件扫描。

开启 Spring Boot 特性有哪几种方式?

1. 继承spring-boot-starter-parent项目
2. 导入spring-boot-dependencies项目依赖

Spring Boot 需要独立的容器运行吗?

可以不需要，内置了 Tomcat/ Jetty 等容器.

运行 Spring Boot 有哪几种方式?

1. 打包用命令或者放到容器中运行
2. 用 Maven/ Gradle 插件运行
3. 直接执行 main 方法运行

Spring Boot 自动配置原理是什么?

注解 @EnableAutoConfiguration, @Configuration, @ConditionalOnClass 就是自动 配置的核心，首先它得是一个配置文件，其次根据类路径下是否有这个类去自动配置。

你如何理解 Spring Boot 中的 Starters?

Starters可以理解为启动器，它包含了一系列可以集成到应用里面的依赖包，你可以一 站式集成 Spring 及其他技术，而不需要到处找示例代码和依赖包。如你想使用 Spring JPA 访问数据库，只要加入 spring-boot-starter-data-jpa 启动器依赖就能使用了。 Starters包含了许多项目中需要用到的依赖，它们能快速持续的运行，都是一系列得到 支持的管理传递性依赖.

如何在 Spring Boot 启动的时候运行一些特定的代码?

可以实现接口 ApplicationRunner 或者 CommandLineRunner，这两个接口实现方式一 样，它们都只提供了一个 run 方法

Spring Boot 有哪几种读取配置的方式?

Spring Boot 可以通过 @PropertySource,@Value,@Environment,

@ConfigurationProperties 来绑定变量

哪些情况用到activeMq?

商品上架后更新ES索引库、更新静态页、发送短信

Spring 特性

Spring的核心特性就是IOC和AOP，IOC（Inversion of Control），即“控制反转”；AOP（Aspect-OrientedProgramming），即“面向切面编程”。IOC：IOC，另外一种说法叫DI（Dependency Injection），即依赖注入。它并不是一种技术实现，而是一种设计思想。在任何一个有实际开发意义的程序项目中，我们会使用很多类来描述它们特有的功能，并且通过类与类之间的相互协作来完成特定的业务逻辑这个时候，每个类都需要负责管理与自己有交互的类的引用和依赖，代码将会变的异常难以维

护和极度的高耦合。而IOC的出现正是用来解决这个问题，我们通过IOC将这些相互依赖对象的创建、协调工作交给Spring容器去处理，每个对象只需要关注其自身的业务逻辑关系就可以了。在这样的角度上来看，获得依赖的对象的方式，进行了反转，变成了由spring容器控制对象如何获取外部资源（包括其他对象和文件资料等等）。

AOP：面向切面编程，往往被定义为促使软件系统实现关注点的分离的技术。系统是由许多不同的组件所组成的，每一个组件各负责一块特定功能。除了实现自身核心功能之外，这些组件还经常承担着额外的职责。例如日志、事务管理和安全这样的核心服务经常融入到自身具有核心业务逻辑的组件中去。这些系统服务经常被称为横切关注点，因为它们会跨越系统的多个组件。

JDBC的理解

JDBC（Java DataBase Connectivity,java数据库连接）是一种用于执行SQL语句的Java API，可以为多种关系数据库提供统一访问，它由一组用Java语言编写的类和接口组成。JDBC提供了一种基准，据此可以构建更高级的工具和接口，使数据库开发人员能够编写数据库应用程序

有了JDBC，向各种关系数据发送SQL语句就是一件很容易的事。换言之，有了JDBCAPI，就不必为访问Sybase数据库专门写一个程序，为访问Oracle数据库又专门写一个程序，或为访问Informix数据库又编写另一个程序等等，程序员只需用JDBC API写一个程序就够了，它可向相应数据库发送SQL调用。

Ajax异步和同步

同步是指：发送方发出数据后，等接收方发回响应以后才发下一个数据包的通讯方式。

异步是指：发送方发出数据后，不等接收方发回响应，接着发送下个数据包的通讯方式。

同步通信方式要求通信双方以相同的时钟频率进行，而且准确协调，通过共享一个单个时钟或定时脉冲源保证发送方和接收方的准确同步，效率较高；

异步通信方式不要求双方同步，收发方可采用各自的时钟源，双方遵循异步的通信协议，以字符为数据传输单位，发送方传送字符的时间间隔不确定，发送效率比同步传送效率低

使用者可以同步或异步实现服务调用。从使用者的观点来看，这两种方式的不同之处在于：

同步——使用者通过单个线程调用服务；该线程发送请求，在服务运行时阻塞，并且等待响应。
异步——使用者通过两个线程调用服务；一个线程发送请求，而另一个单独的线程接收响应

JVM

讲讲jvm的组成与调优，内存模型，tomcat调优

JVM的内部体系结构分为三部分

1. 类装载器（ClassLoader）子系统。作用: 用来装载.class文件
2. 执行引擎。作用:执行字节码，或者执行本地方法
3. 运行时数据区：方法区，堆，java栈，PC寄存器，本地方法栈

JVM将整个类加载过程划分为了三个步骤：

1. 装载：装载过程负责找到二进制字节码并加载至JVM中。JVM通过类名、类所在的包名通过ClassLoader来完成类的加载，同样，也采用以上三个元素来标识一个被加载了的类：类名+包名+ClassLoader实例ID。
2. 链接：链接过程负责对二进制字节码的格式进行校验、初始化装载类中的静态变量以及解析类中调用的接口、类。在完成了校验后，JVM初始化类中的静态变量，并将其值赋为默认值。最后一步为对类中的所有属性、方法进行验证，以确保其需要调用的属性、方法存在，以及具备应的权限（例如public、private域权限等），会造成NoSuchMethodError、NoSuchFieldError等错误信息。

3. 初始化：初始化过程即为执行类中的静态初始化代码、构造器代码以及静态属性的初始化。在四种情况下初始化过程会被触发执行：调用了new；反射调用了类中的方法；子类调用了初始化；JVM启动过程中指定的初始化类。

tomcat调优

1. 增加JVM堆内存大小
2. 修复JRE内存泄漏
3. 线程池设置
4. 压缩
5. 数据库性能调优
6. Tomcat本地库

GC机制

垃圾收集器一般必须完成两件事：检测出垃圾；回收垃圾。怎么检测出垃圾？一般有以下几种方法：

引用计数法

给一个对象添加引用计数器，每当有个地方引用它，计数器就加1；引用失效就减1。好了，问题来了，如果我有两个对象A和B，互相引用，除此之外，没有其他任何对象引用它们，实际上这两个对象已经无法访问，即是我们说的垃圾对象。但是互相引用，计数不为0，导致无法回收，所以还有另一种方法

可达性分析算法

以根集对象为起始点进行搜索，如果有对象不可达的话，即是垃圾对象。这里的根集一般包括java栈中引用的对象、方法区常量池中引用的对象、本地方法中引用的对象等。

总之，JVM在做垃圾回收的时候，会检查堆中的所有对象是否会被这些根集对象引用，
不能够被引用的对象就会被垃圾收集器回收。

一般回收算法也有如下几种

按照基本回收策略分

（1）标记-清除（Mark-sweep）

算法和名字一样，分为两个阶段：标记和清除。标记所有需要回收的对象，然后统一回收。这是最基础的算法，后续的收集算法都是基于这个算法扩展的。不足：效率低；标记清除之后会产生大量碎片。效果图如下：

（2）复制（Copying）

此算法把内存空间划为两个相等的区域，每次只使用其中一个区域。垃圾回收时，遍历当前使用区域，把正在使用中的对象复制到另外一个区域中。此算法每次只处理正在使用中的对象，因此复制成本比较小，同时复制过去以后还能进行相应的内存整理，不会出现“碎片”问题。当然，此算法的缺点也是很明显的，就是需要两倍内存空间。效果图如下：

（3）标记-整理（Mark-Compact）

此算法结合了“标记-清除”和“复制”两个算法的优点。也是分两阶段，第一阶段从根节点开始标记所有被引用对象，第二阶段遍历整个堆，把清除未标记对象并且把存活对象“压缩”到堆的其中一块，按顺序排放。此算法避免了“标记-清除”的碎片问题，同时也避免了“复制”算法的空间问题。效果图如下：

按分区对待的方式分

（1）增量收集（Incremental Collecting）

实时垃圾回收算法，即：在应用进行的同时进行垃圾回收。不知道什么原因JDK5.0中的收集器没有使用这种算法的。

（2）分代收集（Generational Collecting）：

基于对对象生命周期分析后得出的垃圾回收算
法。把对象分为年青代、年老代、持久代，对不同生命周期的对象使用不同的算法（上述方式中的一个）进行回收。现在的垃圾回收器（从J2SE1.2开始）都是使用此算法的。

按系统线程分（回收器类型）

（1）串行收集：串行收集使用单线程处理所有垃圾回收工作，因为无需多线程交互，实现容易，而且效率比较高。但是，其局限性也比较明显，即无法使用多处理器的优势，所以此收集适合单处理器机器。当然，此收集器也可以用在小数据量（100M左右）情况下的多处理器机器上。默认使用串行收集器。

（2）并行收集:

并行收集使用多线程处理垃圾回收工作，因而速度快，效率高。而且理论
上CPU数目越多，越能体现出并行收集器的优势。适合对吞吐量优先（科学技术，后台应用），无过多交互的应用。吞吐量=业务处理时间/（业务处理时间+垃圾回收时间）。

-XX:+UseParallelGC 设置年轻代为并行收集器

-XX:ParallelGCThreads=20设置并行收集器线程数,一般设置为与CPU数相同。

-XX:+UseParallelOldGC设置年老代为并行收集器。

-XX:+UseAdaptiveSizePolicy设置此选项后，并行收集器会自动选择年轻代区大小和相应的Survivor区比例，以达到目标系统规定的最低相应时间或者收集频率等，此值建议使用并行收集器时，一直打开。

（3）并发收集:

相对于串行收集和并行收集而言，前面两个在进行垃圾回收工作时，需要
暂停整个运行环境，而只有垃圾回收程序在运行，因此，系统在垃圾回收时会有明显的暂停，而且暂停时间会因为堆越大而越长。并发收集器不会暂停应用，适合响应时间优先的应用。保证系统的响应时间，减少垃圾收集时的停顿时间。适用于应用服务器、电信领域等。

-XX:+UseConcMarkSweepGC：设置年老代为并发收集。测试中配置这个以后，-XX:NewRatio=4的配置失效了，原因不明。所以，此时年轻代大小最好用-Xmn设置。

常见配置汇总

1. 堆设置
    1. -Xms:初始堆大小
    2. -Xmx:最大堆大小
    3. -XX:NewSize=n:设置年轻代大小
    4. -XX:NewRatio=n:设置年轻代和年老代的比值。默认为2.，表示年轻代与年老代比值为1：2，年轻代占整个年轻代年老代和的1/3
    5. -XX:SurvivorRatio=n:年轻代中Eden区与两个Survivor区的比值。注意Survivor区有两个。默认为8，表示Eden：Survivor=8：2，一个Survivor区占整个年轻代的1/10
    6. -XX:MaxPermSize=n:设置持久代大小
2. 收集器设置
    1. -XX:+UseSerialGC:设置串行收集器
    2. -XX:+UseParallelGC:设置并行收集器
    3. -XX:+UseParalledlOldGC:设置并行年老代收集器
    4. -XX:+UseConcMarkSweepGC:设置并发收集器
3. 垃圾回收统计信息
    1. -XX:+PrintGC

2. -XX:+PrintGCDetails
3. -XX:+PrintGCTimeStamps
4. -Xloggc:filename

4. 并行收集器设置
    1. -XX:ParallelGCThreads=n:设置并行收集器收集时使用的CPU数。并行收集线程数。
    2. -XX:MaxGCPauseMillis=n:设置并行收集最大暂停时间
    3. -XX:GCTimeRatio=n:设置垃圾回收时间占程序运行时间的百分比。公式为1/(1+n)
5. 并发收集器设置
    1. -XX:+CMSIncrementalMode:设置为增量模式。适用于单CPU情况。
    2. -XX:ParallelGCThreads=n:设置并发收集器年轻代收集方式为并行收集时，使用的CPU数。并行收集线程数。

JVM调优总结

1. 年轻代大小选择
    1. **响应时间优先的应用：尽可能设大，直到接近系统的最低响应时间限制**（根据实际情况选择）。在此种情况下，年轻代收集发生的频率也是最小的。同时，减少到达年老代的对象。
    2. **吞吐量优先的应用：**尽可能的设置大，可能到达Gbit的程度。因为对响应时间没有要求，垃圾收集可以并行进行，一般适合8CPU以上的应用。
2. **年老代大小选择**
    1. **响应时间优先的应用：**年老代使用并发收集器，所以其大小需要小心设置，一般要考虑**并发会话率**和**会话持续时间**等一些参数。如果堆设置小了，可以会造成内存碎片、高回收频率以及应用暂停而使用传统的标记清除方式；如果堆大了，则需要较长的收集时间。最优化的方案，一般需要参考以下数据获得：
        1. 并发垃圾收集信息
        2. 持久代并发收集次数
        3. 传统GC信息
        4. 花在年轻代和年老代回收上的时间比例

        减少年轻代和年老代花费的时间，一般会提高应用的效率
    2. **吞吐量优先的应用：**、一般吞吐量优先的应用都有一个很大的年轻代和一个较小的年老代。原因是，这样可以尽可能回收掉大部分短期对象，减少中期的对象，而年老代尽存放长期存活对象。

JVM调优：

-Xms – 指定初始化时化的堆内存，默认为物理内存的1/64

Xmx – 指定最大的内存,默认为物理内存的1/4

-XX:+PrintGCDetails：输出详细的GC处理日志在重启你的Tomcat服务器之后，这些配置的更改才会有效。

调优和高并发

讲讲高可用的数据与服务怎么实现，负载均衡策略以及区别，分布式（及事物），集群，高并发以及遇到的问题和解决方案

分布式：

分布式架构：把系统按照模块拆分成多个子系统，多个子系统分布在不同的网络计算机上相互协作完成业务流程，系统之间需要进行通信。

优点：

把模块拆分，使用接口通信，降低模块之间的耦合度。
把项目拆分成若干个子项目，不同的团队负责不同的子项目。
增加功能时只需要再增加一个子项目，调用其他系统的接口就可以。
可以灵活的进行分布式部署。

缺点：

1. 系统之间交互需要使用远程通信，接口开发增加工作量。
2. 各个模块有一些通用的业务逻辑无法共用。

基于soa的架构

SOA：面向服务的架构。也就是把工程拆分成服务层、表现层两个工程。服务层中包含业务逻辑，只需要对外提供服务即可。表现层只需要处理和页面的交互，业务逻辑都是调用服务层的服务来实现。

分布式架构和soa架构有什么区别?

SOA，主要还是从服务的角度，将工程拆分成服务层、表现层两个工程。
分布式，主要还是从部署的角度，将应用按照访问压力进行归类，主要目标是充分利用服务器的资源，避免资源分配不均.

集群:

一个集群系统是一群松散结合的服务器组，形成一个虚拟的服务器，为客户端用户提供统一的服务。对于这个客户端来说，通常在访问集群系统时不会意识到它的服务是由具体的哪一台服务器提供。集群的目的，是为实
现负载均衡、容错和灾难恢复。以达到系统可用性和可伸缩性的要求。集群系统一般应具高可用性、可伸缩
性、负载均衡、故障恢复和可维护性等特殊性能。一般同一个工程会部署到多台服务器上。常见的tomcat集群，Redis集群，Zookeeper集群，数据库集群.

分布式与集群的区别:

分布式是指将不同的业务分布在不同的地方。 而集群指的是将几台服务器集中在一起，实现同一业务。一句话：分布式是并联工作的，集群是串联工作的。

分布式中的每一个节点，都可以做集群。 而集群并不一定就是分布式的。

举例：就比如新浪网，访问的人多了，他可以做一个群集，前面放一个响应服务器，后面几台服务器完成同一业务，如果有业务访问的时候，响应服务器看哪台服务器的负载不是很重，就将给哪一台去完成。

而分布式，从窄意上理解，也跟集群差不多， 但是它的组织比较松散，不像集群，有一个组织性，一台服务器垮了，其它的服务器可以顶上来。分布式的每一个节点，都完成不同的业务，一个节点垮了，哪这个业务就不可访问了。

分布式是以缩短单个任务的执行时间来提升效率的，而集群则是通过提高单位时间内执行的任务数来提升效率。

举例：如果一个任务由10个子任务组成，每个子任务单独执行需1小时，则在一台服务器上执行该任务需10小时。采用分布式方案，提供10台服务器，每台服务器只负责处理一个子任务，不考虑子任务间的依赖关系，执行完，这个任务只需一个小时。(这种工作模式的一个典型代表就是Hadoop的Map/Reduce分布式计算模型）而采用集群方案，

同样提供10台服务器，每台服务器都能独立处理这个任务。假设有10个任务同时到达，
10个服务器将同时工作，1小时后，10个任务同时完成，这样，整身来看，还是1小时内完成一个任务!

高并发

处理高并发常见的方法有哪些?

1. 数据层
    1. 数据库集群和库表散列
    2. 分表分库
    3. 开启缓存
    4. 表设计优化
    5. Sql语句优化
    6. 缓存服务器（提高查询效率，减轻数据库压力）
    7. 搜索服务器（提高查询效率，减轻数据库压力）
    8. 图片服务器分离
2. 项目层
    1. 采用面向服务分布式架构（分担服务器压力，提高并发能力）
    2. 采用并发访问较高的详情系统采用静态页面，HTML静态化 freemaker使用页面缓存
    3. 用ActiveMQ使得业务进一步进行解耦，提高业务处理能力使用分布式文件系统存储海量文件
    4. 应用层
        1. Nginx服务器来做负载均衡
        2. Lvs做二层负载
        3. 镜像

高可用:

目的：保证服务器硬件故障服务依然可用，数据依然保存并能够被访问。

高可用的服务

1. 分级管理：核心应用和服务具有更高的优先级，比如用户及时付款比能否评价商品更重要；
2. 超时设置：设置服务调用的超时时间,一旦超时,通信框架抛出异常,应用程序则根据服务调度策略选择重试or请求转移到其他服务器上
3. 异步调用：通过消息队列等异步方式完成，避免一个服务失败导致整个应用请求失败的情况。不是所有服务都可以异步调用，对于获取用户信息这类调用，采用异步方式会延长响应时间，得不偿失。对于那些必须确认服务调用成功后才能继续进行下一步的操作的应用也不适合异步调用。
4. 服务降级：网站访问高峰期间，为了保证核心应用的正常运行，需要对服务降级。

    降级有两种手段：

    1. 一是拒绝服务，拒绝较低优先级的应用的调用，减少服务调用并发数，确保核心应用的正常运行；
    2. 二是关闭功能，关闭部分不重要的服务，或者服务内部关闭部分不重要的功能，以节约系统开销，为核心应用服务让出资源；
5. 幂等性设计：保证服务重复调用和调用一次产生的结果相同；

高可用的数据

保证数据高可用的主要手段有两种：一是数据备份，二是失效转移机制；

1. 数据备份：又分为冷备份和热备份，冷备份是定期复制，不能保证数据可用性。热备份又分为异步热备和同步热备，异步热备是指多份数据副本的写入操作异步完成，而同步方式则是指多份数据副本的写入操作同时完成。
2. 失效转移：若数据服务器集群中任何一台服务器宕机，那么应用程序针对这台服务器的所有读写操作都要重新路由到其他服务器，保证数据访问不会失败。

网站运行监控

不允许没有监控的系统上线

1. 监控数据采集

1. 用户行为日志收集：服务器端的日志收集和客户端的日志收集；目前许多网站逐步开发基于实时计算框架Storm的日志统计与分析工具
2. 服务器性能监控：收集服务器性能指标，如系统Load、内存占用、磁盘IO等，及时判断，防患于未然；
3. 运行数据报告：采集并报告，汇总后统一显示，应用程序需要在代码中处理运行数据采集的逻辑；

2. 监控管理

1. 系统报警：配置报警阀值和值守人员联系方式，系统发生报警时，即使工程师在千里之外，也可以被及时通知；
2. 失效转移：监控系统在发现故障时，主动通知应用进行失效转移；
3. 自动优雅降级：为了应付网站访问高峰，主动关闭部分功能，释放部分系统资源，保 证核心应用服务的正 常运行；—>网站柔性架构的理想状态

负载均衡:

什么是负载均衡?

当一台服务器的性能达到极限时，我们可以使用服务器集群来提高网站的整体性能。 那么，在服务器集群 中，需要有一台服务器充当调度者的角色，用户的所有请求都会首先由它接收，调度者 再根据每台服务器的负 载情况将请求分配给某一台后端服务器去处理。

（1）HTTP重定向负载均衡。

原理：当用户向服务器发起请求时，请求首先被集群调度者截获；调度者根据某种分配 策略，选择一台服务 器，并将选中的服务器的IP地址封装在HTTP响应消息头部的Location字段中，并将响 应消息的状态码设为 302，最后将这个响应消息返回给浏览器。当浏览器收到响应消息后，解析Location字 段，并向该URL发起请 求，然后指定的服务器处理该用户的请求，最后将结果返回给用户。

优点：比较简单

缺点：调度服务器只在客户端第一次向网站发起请求的时候起作用。当调度服务器向浏 览器返回响应信息后， 客户端此后的操作都基于新的URL进行的(也就是后端服务器)，此后浏览器就不会与调度 服务器产生关系，浏 览器需要每次请求两次服务器才能拿完成一次访问，性能较差。而且调度服务器在调度 时，无法知道当前用户 将会对服务器造成多大的压力，只不过是把请求次数平均分配给每台服务器罢了，浏览 器会与后端服务器直接交互

（2）DNS域名解析负载均衡

原理：为了方便用户记忆，我们使用域名来访问网站。通过域名访问网站之前，首先需 要将域名解析成IP地 址，这个工作是由DNS域名服务器完成的。我们提交的请求不会直接发送给想要访问的 网站，而是首先发给域名 服务器，它会帮我们把域名解析成IP地址并返回给我们。我们收到IP之后才会向该IP发起请求。一个域名指向 多个IP地址，每次进行域名解析时，DNS只要选一个IP返回给用户，就能够实现服务器 集群的负载均衡。 调度策略：一般DNS提供商会提供一些调度策略供我们选择，如随机分配、轮询、根据 请求者的地域分配离他最近的服务器。

随机分配策略：

当调度服务器收到用户请求后，可以随机决定使用哪台后端服务器，然后将该服务器的 IP封装在HTTP响应消息 的Location属性中，返回给浏览器即可。

轮询策略(RR)：

调度服务器需要维护一个值，用于记录上次分配的后端服务器的IP。那么当新的请求到 来时，调度者将请求依 次分配给下一台服务器

优点：配置简单，将负载均衡工作交给DNS，省略掉了网络管理的麻烦；

缺点：集群调度权交给了DNS服务器，从而我们没办法随心所欲地控制调度者，没办法 定制调度策略，没办法了 解每台服务器的负载情况，只不过把所有请求平均分配给后端服务器罢了。某一台后端 服务器发生故障时，即 使我们立即将该服务器从域名解析中去除，但由于DNS服务器会有缓存，该IP仍然会在DNS中保留一段时间，那 么就会导致一部分用户无法正常访问网站。不过动态DNS能够让我们通过程序动态修改 DNS服务器中的域名解 析。从而当我们的监控程序发现某台服务器挂了之后，能立即通知DNS将其删掉。

（3）反向代理负载均衡。

原理：反向代理服务器是一个位于实际服务器之前的服务器，所有向我们网站发来的请 求都首先要经过反向代 理服务器，服务器根据用户的请求要么直接将结果返回给用户，要么将请求交给后端服 务器处理，再返回给用 户。反向代理服务器就可以充当服务器集群的调度者，它可以根据当前后端服务器的负载情况，将请求转发给 一台合适的服务器，并将处理结果返回给用户。

优点：

1. 部署简单
2. 隐藏后端服务器：与HTTP重定向相比，反向代理能够隐藏后端服务器，所有浏览器都 不会与后端服务器直接 交互，从而能够确保调度者的控制权，提升集群的整体性能。
3. .故障转移 ： 与DNS负载均衡相比，反向代理能够更快速地移除故障结点。当监控程序 发现某一后端服务器出 现故障时，能够及时通知反向代理服务器，并立即将其删除。
4. 4.合理分配任务 ： HTTP重定向和DNS负载均衡都无法实现真正意义上的负载均衡，也 就是调度服务器无法根据 后端服务器的实际负载情况分配任务。但反向代理服务器支持手动设定每台后端服务器 的权重。我们可以根据 服务器的配置设置不同的权重，权重的不同会导致被调度者选中的概率的不同。

缺点：

1. 调度者压力过大 ： 由于所有的请求都先由反向代理服务器处理，那么当请求量超过调 度服务器的最大负载 时，调度服务器的吞吐率降低会直接降低集群的整体性能。
2. 制约扩展 ： 当后端服务器也无法满足巨大的吞吐量时，就需要增加后端服务器的数 量，可没办法无限量地增 加，因为会受到调度服务器的最大吞吐量的制约。
3. 粘滞会话：反向代理服务器会引起一个问题。若某台后端服务器处理了用户的请求， 并保存了该用户的 session或存储了缓存，那么当该用户再次发送请求时，无法保证该请求仍然由保存了 其Session或缓存的服 务器处理，若由其他服务器处理，先前的Session或缓存就找不到了。
    1. 解决办法1： 可以修改反向代理服务器的任务分配策略，以用户IP作为标识较为合适。 相同的用户IP会交由同 一台后端服务器处理，从而就避免了粘滞会话的问题
    2. 解决办法2： 可以在Cookie中标注请求的服务器ID，当再次提交请求时，调度者将该请 求分配给Cookie中标 注的服务器处理即可。
4. IP负载均衡。
    1. 通过NAT实现负载均衡：响应报文一般比较大，每一次都需要NAT转换的话，大流量 的时候，会导致调度器成为一个瓶颈。
    2. 通过直接路由实现负载均衡
    3. VS/TUN 实现虚拟服务器
        1. 优点：IP负载均衡在内核进程完成数据分发，较反向代理均衡有更好的处理性能。
        2. 缺点：负载均衡的网卡带宽成为系统的瓶颈，场景：某个服务器跑的应用非高峰期 间都能达到500M以 上，晚高峰一般能够超过1G，主流服务器的网卡都是千兆的，超过1G的流量明显会导 致丢包的问题，此时又不 能停止业务对网卡进行更换。
5. 数据链路层负载均衡。

    对于linux系统来说，数据链路层的解决方案就是实现多个网卡绑定联合提供服务，把多 张网卡捆绑做成一个 逻辑网卡。避免负载均衡服务器网卡带宽成为瓶颈，是目前大型网站所使用的最广的一种负载均衡手段。

linux bonding的七种模式，mod=0~6：平衡抡循环策略，主-备份策略，平衡策略，广 播策略，动态链接聚 合，适配器传输负载均衡，适配器适应性负载均衡

网络

http和https的区别?

1. https协议需要到ca申请证书，一般免费证书较少，因而需要一定费用。
2. http是超文本传输协议，信息是明文传输，https则是具有安全性的ssl加密传输协议。
3. http和https使用的是完全不同的连接方式，用的端口也不一样，前者是80，后者是443。
4. http的连接很简单，是无状态的；HTTPS协议是由SSL+HTTP协议构建的可进行加密传输、身份认证的网络协议，比http协议安全。

常见的运行时异常?

NullPointerException - 空指针引用异常
ClassCastException - 类型强制转换异常。
IllegalArgumentException - 传递非法参数异常。
ArithmeticException - 算术运算异常
ArrayStoreException - 向数组中存放与声明类型不兼容对象异常
IndexOutOfBoundsException - 下标越界异常
NegativeArraySizeException - 创建一个大小为负数的数组错误异常
NumberFormatException - 数字格式异常
SecurityException - 安全异常
UnsupportedOperationException - 不支持的操作异常

BIO和NIO的区别

互联网 强调的是信息/数据在网络之间的流通，

BIO：堵塞式IO,相当于轮船运输

NIO：非堵塞式IO：面向缓冲区（buffer），基于通道(chanel)的io操作,相当于火车运输，效率高
文件->双向通道（（缓冲区））->程序

java中关于多线程的了解你有多少？线程池有涉及吗?

同一类线程共享代码和数据空间，每个线程有独立的运行栈和程序计数器(PC)，线程切换开销小。线程分为五个阶段：创建、就绪、运行、阻塞、终止。Java线程有五种基本状态

新建状态（New）：当线程对象对创建后，即进入了新建状态，如：Thread t = new MyThread();

就绪状态（Runnable）：当调用线程对象的start()方法（t.start();），线程即进入就绪状态。处于就绪状态的线程，只是说明此线程已经做好了准备，随时等待CPU调度执行，并不是说执行了t.start()此线程立即就会执行；

运行状态（Running）：当CPU开始调度处于就绪状态的线程时，此时线程才得以真正执行，即进入到运行状态。注：就 绪状态是进入到运行状态的唯一入口，也就是说，线程要想进入运行状态执行，首先必须处于就绪状态中；

阻塞状态（Blocked）：处于运行状态中的线程由于某种原因，暂时放弃对CPU的使用权，停止执行，此时进入阻塞状态，直到其进入到就绪状态，才 有机会再次被CPU调用以进入到运行状态。根据阻塞产生的原因不同，阻塞状态又可以分为三种：
1.等待阻塞：运行状态中的线程执行wait()方法，使本线程进入到等待阻塞状态；
2.同步阻塞 -- 线程在获取synchronized同步锁失败(因为锁被其它线程所占用)，它会进入同步阻塞状态；
3.其他阻塞 -- 通过调用线程的sleep()或join()或发出了I/O请求时，线程会进入到阻塞状态。当sleep()状态超时、join()等待线程终止或者超时、或者I/O处理完毕时，线程重新转入就绪状态。

死亡状态（Dead）：线程执行完了或者因异常退出了run()方法，该线程结束生命周期。

Java中线程的创建常见有如三种基本形式

1.继承Thread类，重写该类的run()方法

2.实现Runnable接口，并重写该接口的run()方法

该run()方法同样是线程执行体，创建Runnable实现类的实例，并以此实例作为Thread类的target来创建Thread对象，该Thread对象才是真正的线程对象。

3.使用Callable和Future接口创建线程。

具体是创建Callable接口的实现类，并实现clall()方法。并使用FutureTask类来包装Callable实现类的对象，且以此FutureTask对象作为Thread对象的target来创建线程。线程池：线程池是一种多线程处理形式，处理过程中将任务添加到队列，然后在创建线程后自动启动这些任务。线程池线程都是后台线程。每个线程都使用默认的堆栈大小，以默认的优先级运行，并处于多线程单元中。如果某个线程在托管代码中空闲（如正在等待某个事件）,则线程池将插入另一个辅助线程来使所有处理器保持繁忙。如果所有线程池线程都始终保持繁忙，但队列中包含挂起的工作，则线程池将在一段时间后创建另一个辅助线程但线程的数目永远不会超过最大值。超过最大值的线程可以排队，但他们要等到其他线程完成后才启动。

如何实现线程的同步?

为何要使用同步?

java允许多线程并发控制，当多个线程同时操作一个可共享的资源变量时（如数据的增删改查），将会导致数据不准确，相互之间产生冲突，因此加入同步锁以避免在该线程没有完成操作之前，被其他线程的调用，从而保证了该变量的唯一性和准确性。
线程同步（5种同步方式）
1.同步方法 2.同步代码块 3.使用特殊域变量(volatile)实现线程同步 4.使用重入锁实现线程同步 5.使用局部变量实现线程同步

遍历hashmap有几种方式?

Map的四种遍历方式

1. for each map.entrySet()
2. 显示调用map.entrySet()的集合迭代器
3. for each map.keySet()，再调用get获取
4. for each map.entrySet()，用临时变量保存map.entrySet()

简单介绍一下Es全文检索在整个系统中的应用，在更新索引库的同时会产生索引碎片，这个碎片是如何处理的?

根据商品的名称，分类，品牌等属性来创建索引进行商品搜索。更新索引库时会先删除索引，然后再重建。而对于删除聚集索引，则会导致对应的非聚集索引重建两次(删除时重建，建立时再重建).

直接删除碎片。

java并发包下有哪些并发组件?

分为两层组成

外层框架主要有Lock(ReentrantLock、ReadWriteLock等)、同步器（semaphores等）、阻塞队列（BlockingQueue等）、Executor（线程池）、并发容器（ConcurrentHashMap等）、还有Fork/Join框架；内层有AQS（AbstractQueuedSynchronizer类，锁功能都由他实现）、非阻塞数据结构、原子变量类(AtomicInteger等无锁线程安全类)三种。

讲一下jvm调优。

1. 堆大小设置
2. 回收器选择
3. 辅助信息
4. JVM提供了大量命令行参数，打印信息，供调试使用；

讲一下jvm的组成。

JVM 由类加载器子系统、运行时数据区、执行引擎以及本地方法接口组成

讲一下ThreadLocal类。

ThreadLocal，很多地方叫做线程本地变量，也有些地方叫做线程本地存储，其实意思差不多。可能很多朋友都知道ThreadLocal为变量在每个线程中都创建了一个副本，那么每个线程可以访问自己内部的副本变量；

ThreadLocal在每个线程中对该变量会创建一个副本，即每个线程内部都会有一个该变量，且在线程内部任何地方都可以使用，线程之间互不影响，这样一来就不存在线程安全问题，也不会严重影响程序执行性能。

但是要注意，虽然ThreadLocal能够解决上面说的问题，但是由于在每个线程中都创建了副本，所以要考虑它对资源的消耗，比如内存的占用会比不使用ThreadLocal要大；

怎么确保session共享?

在分布式项目中实现session共享必须做以下准备工作：

1. Cookie中共享ticket
2. Redis存储session

分布式系统共享用户身份信息session，必须先获取ticket票据，然后再根据票据信息获取redis中用户身份信息。实现以上2点即可实现session共享。

目前项目中使用的springsecurity + cas 来实现的单点登录，cas自动产生ticket票据信息，每次获取用户信息，cas将会携带ticket信息获取用户身份信息。

项目中哪块涉及了线程问题，怎么处理的?

项目的高并发访问就是一个多线程问题。

项目中普通的业务开发基本没有涉及多线程问题，不过你可以谈谈你使用的框架中使用的多线程技术：

因为我们项目使用的框架进行开发的，因此多线程处理多让框架非我们处理结束了。

1. 高并发就是多线程，这里的多线程让servlet服务器给处理了谈谈Tomcat多线程配置;
2. 配置线程池，扩大并发能力
3. 开启NIO能力等等
4. 框架多线程：mybatis 框架底层使用的连接池

MYSQL

事务的ACID和事务的隔离性?

1. 原子性(Atomic)：事务中各项操作，要么全做要么全不做，任何一项操作的失败都会导致整个事务的失
   败;
2. 一致性(Consistent)：事务结束后系统状态是一致的;
3. 隔离性(Isolated)：并发执行的事务彼此无法看到对方的中间状态;
4. 持久性(Durable)：事务完成后所做的改动都会被持久化，即使发生灾难性的失败。通过日志和同步备份可以在故障发生后重建数据。

如果不考虑事务的隔离性，会发生的几种问题：

1. 脏读：事务A读到了事务B未提交的数据。
2. 可重复读:事务A第一次查询得到一行记录row1,事务B提交修改后,事务A第二次查询得到row1,但列内容发生了变化,侧重于次数，侧重于update
3. 幻读：事务A第一次查询得到一行记录row1，事务B提交修改后，事务A第二次查询得到两行记录row1和row2，侧重于内容，侧重于insert

现在来看看MySQL数据库为我们提供的四种隔离级别：

1. Serializable (串行化)：可避免脏读、不可重复读、幻读的发生。
2. Repeatable read (可重复读)：可避免脏读、不可重复读的发生。
3. Read committed (读已提交)：可避免脏读的发生。
4. Read uncommitted (读未提交)：最低级别，任何情况都无法保证。

以上四种隔离级别最高的是Serializable级别，最低的是Read uncommitted级别，当然级别越高，执行效率就越
低。像Serializable这样的级别，就是以锁表的方式(类似于Java多线程中的锁)使得其他的线程只能在锁外等待，所以
平时选用何种隔离级别应该根据实际情况。在MySQL数据库中默认的隔离级别为Repeatable read (可重复读)。

在MySQL数据库中，支持上面四种隔离级别，默认的为Repeatable read (可重复读)；而在Oracle数据库中，只支
持Serializable (串行化)级别和Read committed (读已提交)这两种级别，其中默认的为Read committed级别。

在MySQL数据库中查看当前事务的隔离级别：

select @@tx_isolation;

在MySQL数据库中设置事务的隔离 级别：

set [glogal | session] transaction isolation level 隔离级别名称;

set tx_isolation='隔离级别名称;'

讲讲你是怎么优化数据库（sql，表的设计）以及索引的使用有哪些 限制条件（索引失效）

1. 选取最适用的字段：在创建表的时候，为了获得更好的性能，我们可以将表中字段的 宽度设得尽可能小。另外一 个提高效率的方法是在可能的情况下，应该尽量把字段设置为NOTNULL，
2. 使用连接（JOIN）来代替子查询(Sub-Queries)
3. 使用联合(UNION)来代替手动创建的临时表
4. 事物：
    1. 要么语句块中每条语句都操作成功，要么都失败。换句话说，就是可以保持数据库 中数据的一致性和完整 性。事物以BEGIN关键字开始，COMMIT关键字结束。在这之间的一条SQL操作失败， 那么,ROLLBACK命令就可以 把数据库恢复到BEGIN开始之前的状态。
    2. 是当多个用户同时使用相同的数据源时，它可以利用锁定数据库的方法来为用户提 供一种安全的访问方 式，这样可以保证用户的操作不被其它的用户所干扰。
5. ,减少表关联，加入冗余字段
6. 使用外键：锁定表的方法可以维护数据的完整性，但是它却不能保证数据的关联性。 这个时候我们就可以使用外键。
7. 使用索引
8. 优化的查询语句
9. 集群
10. 读写分离
11. 主从复制
12. 分表
13. 分库
14. 适当的时候可以使用存储过程

限制:：尽量用全职索引，最左前缀：查询从索引的最左前列开始并且不跳过索引中的 列；索引列上不操作，范围之 后全失效； 不等空值还有OR，索引影响要注意；like以通配符%开头索引失效会变成全 表扫描的操作，字符串不加单引号索引失效.

讲讲Redis缓存，它的数据类型，和其他缓存之间的区别，及持久化，缓存穿透与雪崩它的解决方案

redis是内存中的数据结构存储系统，一个key-value类型的非关系型数据库，可持久化的数据库，相对于关系型数据库（数据主要存在硬盘中），性能高，因此我们一般用redis来做缓存使用；并且redis支持丰富的数据类型，比较容易解决各种问题，因此redis可以用来作为注册中心， 数据库、缓存和消息中间件。Redis的 Value支持5种数据类型，string、hash、list、set、zset（sorted set）；

1. String类型：一个key对应一个value
2. Hash类型：它的key是string类型，value又是一个map（key-value），适合存储对 象。
3. List类型：按照插入顺序的字符串链表（双向链表），主要命令是LPUSH和RPUSH，能 够支持反向查找和遍历
4. Set类型：用哈希表类型的字符串序列，没有顺序，集合成员是唯一的，没有重复数据，底层主要是由一个value永远为null的hashmap来实现的。
5. 和set类型基本一致，不过它会给每个元素关联一个double类型的分数 （score），这样就可以为成员排序，并且插入是有序的。

Memcache和redis的区别：

数据支持的类型：redis不仅仅支持简单的k/v类型的数据，同时还支持list、set、 zset、hash等数据结构 的存储；memcache只支持简单的k/v类型的数据，key和value都是string类型 可靠性：memcache不支持数据持久化，断电或重启后数据消失，但其稳定性是有保证 的；redis支持数据持久 化和数据恢复，允许单点故障，但是同时也会付出性能的代价 性能上：对于存储大数据，memcache的性能要高于redis

应用场景：

Memcache：适合多读少写，大数据量的情况（一些官网的文章信息等）

Redis：适用于对读写效率要求高、数据处理业务复杂、安全性要求较高的系统

案例：分布式系统，存在session之间的共享问题，因此在做单点登录的时候，我们利 用redis来模拟了session的共享，来存储用户的信息，实现不同系统的session共享；

redis的持久化方式有两种：

RDB（半持久化方式）：

按照配置不定期的通过异步的方式、快照的形式直接把内存中的 数据持久化到磁盘的一个dump.rdb文件（二进制的临时文件）中，redis默认的持久化方式，它在配置文件 （redis.conf）中。

优点：只包含一个文件，将一个单独的文件转移到其他存储媒介上，对于文件备份、灾 难恢复而言，比较实用。

缺点：系统一旦在持久化策略之前出现宕机现象，此前没有来得及持久化的数据将会产 生丢失

AOF（全持久化的方式）：

把每一次数据变化都通过write()函数将你所执行的命令追加到 一个appendonly.aof文件里面，Redis默认是不支持这种全持久化方式的，需要在配置文件（redis.conf）中将 appendonly no改成appendonly yes。

优点：数据安全性高，对日志文件的写入操作采用的是append模式，因此在写入过程中 即使出现宕机问题，不会破坏日志文件中已经存在的内容；

缺点：对于数量相同的数据集来说，aof文件通常要比rdb文件大，因此rdb在恢复大数 据集时的速度大于AOF；

AOF持久化配置:

在Redis的配置文件中存在三种同步方式，它们分别是： appendf sync always #每次有数据修改发生时都会都调用f sync刷新到aof文件，非 常慢，但是安全；

appendf sync everysec #每秒钟都调用f sync刷新到aof文件中，很快，但是可能丢失 一秒内的数据，推荐使用，兼顾了速度和安全；

appendf sync no #不会自动同步到磁盘上，需要依靠OS（操作系统）进行刷新， 效率快，但是安全性就比较差；

二种持久化方式区别：

AOF在运行效率上往往慢于RDB，每秒同步策略的效率是比较高的，同步禁用策略的效 率和RDB一样高效；

如果缓存数据安全性要求比较高的话，用aof这种持久化方式（比如项目中的购物车）；

如果对于大数据集要求效率高的话，就可以使用默认的。而且这两种持久化方式可以同时使用。

redis-cluster集群，这种方式采用的是无中心结构，每个节点保存数据和整个集群的状态，每个节点都和其他所有节点连接。如果使用的话就用redis-cluster集群。集群这块是公司运维搭建的，具体怎么搭建不是太了解。
我们项目中redis集群主要搭建了6台，3主（为了保证redis的投票机制）3从（高可用），每个主服务器都有一个从服务器，作为备份机。所有的节点都通过PING-PONG机制彼此互相连接；客户端与redis集群连接，只需要连接集群中的任何一个节点即可；Redis-cluster中内置了16384个哈希槽，Redis-cluster把所有的物理节点映射到【0-16383】slot上，负责维护。

Redis是有事务的，redis中的事务是一组命令的集合，这组命令要么都执行，要不都不执行，保证一个事务中的命令依次执行而不被其他命令插入。redis的事务是不支持回滚操作的。redis事务的实现，需要用到MULTI（事务的开始）和EXEC（事务的结束）命令；

缓存穿透：缓存查询一般都是通过key去查找value，如果不存在对应的value，就要去数
据库中查找。如果这
个key对应的value在数据库中也不存在，并且对该key并发请求很大，就会对数据库产生很大的压力，这就叫缓存穿透

解决方案：

1. 对所有可能查询的参数以hash形式存储，在控制层先进行校验，不符合则丢弃。
2. 将所有可能存在的数据哈希到一个足够大的bitmap中，一个一定不存在的数据会被这个bitmap拦截掉，从而避免了对底层存储系统的查询压力。
3. 如果一个查询返回的数据为空（不管是数 据不存在，还是系统故障），我们仍然把这个空结果进行缓存，但它的过期时间会很短，最长不超过五分钟

缓存雪崩：

缓存服务器重启或者大量缓存集中在一段时间内失效，发生大量的缓存穿
透，这样在失效的瞬间对数据库的访问压力就比较大，所有的查询都落在数据库上，造成了缓存雪崩。这个没有完美解决办法，但可以分析用户行为，尽量让失效时间点均匀分布。大多数系统设计者考虑用加锁或者队列的方式保证缓存的单线程（进程）写，从而避免失效时大量的并发请求落到底层存储系统上。

解决方案：

1. 在缓存失效后，通过加锁或者队列来控制读数据库写缓存的线程数量。比如对某个key只允许一个线程查询数据和写缓存，其他线程等待。
2. .可以通过缓存reload机制，预先去更新缓存，再即将发生大并发访问前手动触发加载缓存
3. 不同的key，设置不同的过期时间，让缓存失效的时间点尽量均匀
4. 做二级缓存，或者双缓存策略。A1为原始缓存，A2为拷贝缓存，A1失效时，可以访问A2，A1缓存失效时间设置为短期，A2设置为长期。

redis的安全机制（你们公司redis的安全这方面怎么考虑的？）

漏洞介绍：redis默认情况下，会绑定在bind 0.0.0.0:6379，这样就会将redis的服务暴露到公网上，如果在没有开启认证的情况下，可以导致任意用户在访问目标服务器的情况下，未授权就可访问redis以及读取redis的数据，攻击者就可以在未授权访问redis的情况下可以利用redis的相关方法，成功在redis服务器上
写入公钥，进而可以直接使用私钥进行直接登录目标主机；

解决方案：

1. 禁止一些高危命令。修改redis.conf文件，用来禁止远程修改DB文件地址
2. 以低权限运行redis服务。为redis服务创建单独的用户和根目录，并且配置禁止登录；
3. 为redis添加密码验证。修改redis.conf文件，添加requirepass mypassword；
4. .禁止外网访问redis。修改redis.conf文件，添加或修改 bind 127.0.0.1，使得redis服务只在当前主机使用；
5. 做log监控，及时发现攻击；

redis的哨兵机制（redis2.6以后出现的）

哨兵机制：

监控：监控主数据库和从数据库是否正常运行；

提醒：当被监控的某个redis出现问题的时候，哨兵可以通过API向管理员或者其他应用程序发送通知；

自动故障迁移：主数据库出现故障时，可以自动将从数据库转化为主数据库，实现自动切换；

如果master主服务器设置了密码，记得在哨兵的配置文件（sentinel.conf）里面配置访问密码

redis中对于生存时间的应用

Redis中可以使用expire命令设置一个键的生存时间，到时间后redis会自动删除；

redis中对于生存时间的应用

Redis中可以使用expire命令设置一个键的生存时间，到时间后redis会自动删除；

应用场景：

1. 设置限制的优惠活动的信息；
2. 一些及时需要更新的数据，积分排行榜；
3. 手机验证码的时间；
4. 限制网站访客访问频率；

你们项目中使用到的数据库是什么？你有涉及到关于数据库到建库建表操作吗？数据库创建表的时候会有哪些考虑呢？

项目中使用的是MySQL数据库，

数据库创建表时要考虑

1. 大数据字段最好剥离出单独的表，以便影响性能
2. 使用varchar，代替char，这是因为varchar会动态分配长度，char指定为20，即时你存储字符“1”，它依然是20的长度
3. 给表建立主键，看到好多表没主键，这在查询和索引定义上将有一定的影响
4. 避免表字段运行为null，如果不知道添加什么值，建议设置默认值，特别int类型，比如默认值为0，在索引查询上，效率立显
5. 建立索引，聚集索引则意味着数据的物理存储顺序，最好在唯一的，非空的字段上建立，其它索引也不是越多越好，索引在查询上优势显著，在频繁更新数据的字段上建立聚集索引，后果很严重，插入更新相当忙。
6. 组合索引和单索引的建立，要考虑查询实际和具体模式

mysql中哪些情况下可以使用索引，哪些情况不能使用索引？mysql索引失效的情形有哪些?

使用索引：

1. 为了快速查找匹配WHERE条件中涉及到列
2. 如果表有一个multiple-column索引，任何一个索引的最左前缀可以通过使用优化器来查找行
3. 当运行joins时，为了从其他表检索行。MySql可以更有效的使用索引在多列上如果他们声明的类型和大小是一样的话。在这个环境下，VARCHAR和CHAR是一样的如果他们声明的大小是一样的
4. 为了找到 MIN() or MAX()的值对于一个指定索引的列key_col.总之，就是经常用到的列就最好创建索引。

不能使用引用：

1. 数据唯一性差（一个字段的取值只有几种时）的字段不要使用索引比如性别，只有两种可能数据。意味着索引的二叉树级别少，多是平级。这样的二叉树查找无异于全表扫描
2. 频繁更新的字段不要使用索引 比如logincount登录次数，频繁变化导致索引也频繁变化，增大数据库工作量，降低效
   率
3. 字段不在where语句出现时不要添加索引,如果where后含IS NULL /IS NOT NULL/like ‘%输入符%’等条件，不建议使用索引只有在where语句出现，mysql才会去使用索引
4. where 子句里对索引列使用不等于（<>），使用索引效果一般

索引失效：

1. 如果条件中有or，即使其中有条件带索引也不会使用(这也是为什么尽量少用or的原因)
   注意：要想使用or，又想让索引生效，只能将or条件中的每个列都加上索引
2. 对于多列索引，不是使用的第一部分，则不会使用索引
3. like查询是以%开头
4. 如果列类型是字符串，那一定要在条件中将数据使用引号引用起来,否则不使用索引
5. 如果mysql估计使用全表扫描要比使用索引快,则不使用索引

你有了解mysql的隔离级别吗？mysql默认的隔离级别是什么?

数据库事务的隔离级别有四种，隔离级别高的数据库的可靠性高，但并发量低，而隔离级别低的数据库可靠性低，但并发量高，系统开销小。

1. READ UNCIMMITTED（未提交读）
2. READ COMMITTED（提交读）
3. REPEATABLE READ（可重复读）
4. SERIALIZABLE（可串行化）

mysql默认的事务处理级别是'REPEATABLE-READ',也就是可重复读。

sql语句中关于查询语句的优化你们是怎么做的

1. 应尽量避免在 where 子句中使用!=或<>操作符，否则将引擎放弃使用索引而进行全表扫描。
2. 对查询进行优化，应尽量避免全表扫描，首先应考虑在 where 及 order by 涉及的列上建立索引。
3. 应尽量避免在 where 子句中对字段进行 null 值判断，否则将导致引擎放弃使用索引而进行全表扫描
4. 尽量避免在 where 子句中使用 or 来连接条件，否则将导致引擎放弃使用索引而进行全表扫描
5. in 和 not in 也要慎用，否则会导致全表扫描

6. 应尽量避免在 where 子句中对字段进行表达式操作，这将导致引擎放弃使用索引而进行全表扫描。
7. 应尽量避免在where子句中对字段进行函数操作，这将导致引擎放弃使用索引而进行全表扫描
8. 不要在 where 子句中的“=”左边进行函数、算术运算或其他表达式运算，否则系统将可能无法正确使用索引。
9. 在使用索引字段作为条件时，如果该索引是复合索引，那么必须使用到该索引中的第一个字段作为条件时才能保证系统使用该索引，否则该索引将不会被使 用，并且应尽可能的让字段顺序与索引顺序相一致。
10. 索引并不是越多越好，索引固然可以提高相应的 select 的效率，但同时也降低了insert 及 update 的效率，因为 insert 或 update 时有可能会重建索引，所以怎样建索引需要慎重考虑，视具体情况而定。
11. 尽可能的使用 varchar/nvarchar 代替 char/nchar ，因为首先变长字段存储空间小，可以节省存储空间，其次对于查询来说，在一个相对较小的字段内搜索效率显然要高些。
12. 任何地方都不要使用 select * from ，用具体的字段列表代替“*”，不要返回用不到的任何字段。

mysql索引失效的场景有哪些？like做模糊查询的时候会失效吗？

1. WHERE字句的查询条件里有不等于号（WHERE column!=...），MYSQL将无法使用索引
2. 类似地，如果WHERE字句的查询条件里使用了函数（如：WHEREDAY(column)=...），MYSQL将无法使用索引
3. 在JOIN操作中（需要从多个数据表提取数据时），MYSQL只有在主键和外键的数据类型相同时才能使用索引，否则即使建立了索引也不会使用
4. 如果WHERE子句的查询条件里使用了比较操作符LIKE和REGEXP，MYSQL只有在搜索模板的第一个字符不是通配符的情况下才能使用索引。比如说，如果查询条件是LIKE'abc%',MYSQL将使用索引；如果条件是LIKE '�c'，MYSQL将不使用索引。
5. .在ORDER BY操作中，MYSQL只有在排序条件不是一个查询条件表达式的情况下才使用索引。尽管如此，在涉及多个数据表的查询里，即使有索引可用，那些索引在加快ORDER BY操作方面也没什么作用
6. 如果某个数据列里包含着许多重复的值，就算为它建立了索引也不会有很好的效果。比如说，如果某个数据列里包含了净是些诸如“0/1”或“Y/N”等值，就没有必要为它创建一个索引。
7. 索引有用的情况下就太多了。基本只要建立了索引，除了上面提到的索引不会使用的情况下之外，其他情况只要是使用在WHERE条件里，ORDER BY 字段，联表字段，一般都是有效的。 建立索引要的就是有效果。 不然还用它干吗？ 如果不能确定在某个字段上建立的索引是否有效果，只要实际进行测试下比较下执行时间就知道。
8. 如果条件中有or(并且其中有or的条件是不带索引的)，即使其中有条件带索引也不会使用(这也是为什么尽量少用or的原因)。注意：要想使用or，又想让索引生效，只能将or条件中的每个列都加上索引
9. 如果列类型是字符串，那一定要在条件中将数据使用引号引用起来,否则不使用索引
10. 如果mysql估计使用全表扫描要比使用索引快,则不使用索引
11. 问题二：Like模糊查询，建立索引会失效

项目中关于表结构拆分，你们是业务层面的拆分还是表结构层面的拆分？

表结构层面的拆分。通过mycat数据库中间件完成数据库分表操作。

业务层面也有拆分，比如商品模块拆分成8张表来实现存储

有了解过大数据层面的分库分表吗？以及mysql的执行计划吗？

分库：通过Mycat结点来管理不同服务器上的数据库，每个表最多存500万条记录

分表：重直切割，水平切割

MySql提供了EXPLAIN语法用来进行查询分析，在SQL语句前加一个"EXPLAIN"即可。
mysql中的explain语法可以帮助我们改写查询，优化表的结构和索引的设置，从而最大地提高查询效率。

有了解过数据库中的表级锁和行级锁吗？乐观锁和悲观锁你有哪些了解？

MySQL的锁机制比较简单，其最显著的特点是不同的存储引擎支持不同的锁机制。比如，MyISAM和MEMORY存储引擎采用的是表级锁（table-level locking）；InnoDB存储引擎既支持行级锁（row-level locking），也支持表级锁，但默认情况下是采用行级
锁。MySQL主要的两种锁的特性可大致归纳如下:表级锁： 开销小，加锁快；不会出现死锁(因为MyISAM会一次性获得SQL所需的全部
锁)；锁定粒度大，发生锁冲突的概率最高,并发度最低。行级锁： 开销大，加锁慢；会出现死锁；锁定粒度最小，发生锁冲突的概率最低,并发度也最高。

乐观锁：通过version版本字段来实现

悲观锁：通过for update来实现

Mysql优化有没有工具

三个MySQL性能测试工具：The MySQL Benchmark Suite、MySQL super-smack、MyBench。除了第一个为MySQL性能测试工具，其他两个都为压力测试工具。

你们项目中使用到的数据库是什么？你有涉及到关于数据库到建库建表操作吗？数据库创建表的时候会有哪些考虑呢？

项目中使用的是MySQL数据库

数据库创建表时要考虑

1. 大数据字段最好剥离出单独的表，以便影响性能
2. 使用varchar，代替char，这是因为varchar会动态分配长度，char指定为20，即时你存储字符“1”，它依然是20的长度
3. 给表建立主键，看到好多表没主键，这在查询和索引定义上将有一定的影响
4. 避免表字段运行为null，如果不知道添加什么值，建议设置默认值，特别int类型，比如默认值为0，在索引查询上，效率立显
5. 建立索引，聚集索引则意味着数据的物理存储顺序，最好在唯一的，非空的字段上建立，其它索引也不是越多越好，索引在查询上优势显著，在频繁更新数据的字段上建立聚集索引，后果很严重，插入更新相当忙。
6. 组合索引和单索引的建立，要考虑查询实际和具体模式

怎样进行数据库性能调优

应用程序优化

1. 把数据库当作奢侈的资源看待，在确保功能的同时，尽可能少地动用数据库资源。
2. 不要直接执行完整的SQL 语法，尽量通过存储过程实现数据库操作
3. 客户与服务器连接时，建立连接池，让连接尽量得以重用，以避免时间与资源的损耗。
4. 非到不得已，不要使用游标结构，确实使用时，注意各种游标的特性。

基本表设计优化

1. 表设计遵循第三范式。在基于表驱动的信息管理系统中，基本表的设计规范是第三范式。
2. 分割表。分割表可分为水平分割表和垂直分割表两种：水平分割按照行将一个表分割为多个表
3. 引入中间表

数据库索引优化

> 索引是建立在表上的一种数据组织，它能提高访问表中一条或多条记录的特定查询效率

聚集索引

一种索引，该索引中键值的逻辑顺序决定了表中相应行的物理顺序。

聚集索引确定表中数据的物理顺序。

非聚集索引

一种索引，该索引中索引的逻辑顺序与磁盘上行的物理存储顺序不同

怎样进行数据库优化?

1. 选取最适用的字段

   在创建表的时候，为了获得更好的性能，我们可以将表中字段的宽度设得尽可能小。另外一个提高效率的方法是在可能的情况下，应该尽量把字段设置为NOTNULL。

2. 使用连接（JOIN）来代替子查询(Sub-Queries)

3. 使用联合(UNION)来代替手动创建的临时表

4. 事物：

   1. 要么语句块中每条语句都操作成功，要么都失败。换句话说，就是可以保持数据库中数据的一致性和完整性。事物以BEGIN关键字开始，COMMIT关键字结束。在这之间的一条SQL操作失败，那么，ROLLBACK命令就可以把数据库恢复到BEGIN开始之前的状态。
   2. 当多个用户同时使用相同的数据源时，它可以利用锁定数据库的方法来为用户提供一种安全的访问方式，这样可以保证用户的操作不被其它的用户所干扰。
   3. 锁定表

4. 使用外键 锁定表的方法可以维护数据的完整性，但是它却不能保证数据的关联性。这个时候我们就可以使用外键。
5. 使用索引
6. 优化的查询语句

ActiveMq

activemq的原理

原理：生产者生产消息， 把消息发送给activemq。 Activemq 接收到消息， 然后查看 有多少个消费者， 然后把消息转发给消费者， 此过程中生产者无需参与。 消费者接收到消息后做相应的 处理和生产者没有任何关系

对比RabbitMQ

RabbitMQ的协议是AMQP，而ActiveMQ使用的是JMS协议。顾名思义JMS是针对 Java体系的传输协议，队列两 端必须有JVM，所以如果开发环境都是java的话推荐使用ActiveMQ，可以用Java的一 些对象进行传递比如 Map、Blob（二进制大数据）、Stream等。而AMQP通用行较强，非java环境经常使用，传输内容就是标准字 符串。RabbitMQ安装比较麻烦。ActiveMQ解压即可用不用任何安装。

对比KafKa

Kafka性能超过ActiveMQ等传统MQ工具，集群扩展性好。

弊端是：

1. 在传输过程中可能会出现消息重复的情况，
2. 不保证发送顺序
3. 一些传统MQ的功能没有，比如消息的事务功能。所以通常用Kafka处理大数据日志。

对比Redis

其实Redis本身利用List可以实现消息队列的功能，但是功能很少，而且队列体积较大时性能会急剧下降。对于数据量不大、业务简单的场景可以使用。 如何解决消息重复问题 所谓消息重复,就是消费者接收到了重复的消息,一般来说我们对于这个问题的处理要把握

下面几点

1. 消息不丢失(上面已经处理了)
2. 消息不重复执行

   一般来说我们可以在业务段加一张表,用来存放消息是否执行成功,每次业务事物commit 之后,告知服务端,已经 处理过该消息,这样即使你消息重发了,也不会导致重复处理 大致流程如下:业务端的表记录已经处理消息的id,每次一个消息进来之前先判断该消息是 否执行过,如果执行 过就放弃,如果没有执行就开始执行消息,消息执行完成之后存入这个消息的id

Elasticsearch

简单介绍一个Elasticsearch

ElasticSearch是一个基于Lucene的搜索服务器。通过HTTP使用JSON进行数据索引，用于分布式全文检索，解决人们对于搜索的众多要求。

lucene与elasticsearch（solr）有什么区别?

lucene只是一个提供全文搜索功能类库的核心工具包，而真正使用它还需要一个完善的服务框架搭建起来的应用。好比lucene是类似于jdk，而搜索引擎软件就是tomcat 的。elasticsearch和solr,这两款都是基于lucene的搭建的，可以独立部署启动的搜索引擎服务软件。

基本概念:

cluster集群	**整个elasticsearch 默认就是集群状态，整个集群是一份完整、互备的数据。**
node节点	集群中的一个节点，一般只一个进程就是一个node
shard分片	分片，即使是一个节点中的数据也会通过hash算法，分成多个片存放，默认是5片。
index逻辑数据库	相当于rdbms的database, 对于用户来说是一个逻辑数据库，虽然物理上会被分多个shard存放，也可能存放在多个node中。
type	类似于rdbms的table，但是与其说像table，其实更像面向对象中的class , 同一Json的格式的数据集合。
document	类似于rdbms的 row、面向对象里的object
field	相当于字段、属性

与MySQL对比

利用kibana学习 elasticsearch restful api (DSL)

Kibana 是一个开源分析和可视化平台，可视化操作 Elasticsearch 。Kibana可以用来搜索，查看和与存储在 Elasticsearch 索引中的数据进行交互。可以轻松地进行高级数据分析，并可在各种图表，表格和地图中显示数据。
ES提供了基于JSON的query DSL查询语言

```
public class  Movie {
 String id;
     String name;
     Double doubanScore;
     List<Actor> actorList;
}

public class Actor{
String id;
```

```
String name;
}
{
  “id”:”1”,
  “name”:”operation red sea”,
  “doubanScore”:”8.5”,
  “actorList”:[
{“id”:”1”,”name”:”zhangyi”},
{“id”:”2”,”name”:”haiqing”},
{“id”:”3”,”name”:”zhanghanyu”}
]
}
```

这两个对象如果放在关系型数据库保存，会被拆成2张表，但是elasticsearch是用一个json来表示一个document。所以它保存到es中应该是：

es 的java 客户端的选择

目前市面上有两类客户端一类是TransportClient 为代表的ES原生客户端，不能执行原生dsl语句必须使用它的
Java api方法。另外一种是以Rest Api为主的missing client，最典型的就是jest。 这种客户端可以直接使用dsl语句拼成的字符串，直接传给服务端，然后返回json字符串再解析。两种方式各有优劣，但是最近elasticsearch官网，宣布计划在7.0以后的版本中废除TransportClient。以RestClient为主。在官方的RestClient 基础上，进行了简单包装的Jest客户端，就成了首选，而且该客户端也与springboot完美集成。

中文分词

elasticsearch本身自带的中文分词，就是单纯把中文一个字一个字的分开，根本没有词汇的概念。

问题：

1. es大量的写操作会影响es 性能，因为es需要更新索引，而且es不是内存数据库，会做相应的io操作
2. 而且修改某一个值，在高并发情况下会有冲突，造成更新丢失，需要加锁，而es的乐观锁会恶化性能问题。

解决思路：

1. 用redis做精确计数器，redis是内存数据库读写性能都非常快，利用redis的原子性的自增可以解决并发写操作。
2. redis每计100次数（可以被100整除）我们就更新一次es ，这样写操作就被稀释了100倍，这个倍数可以根据业务情况灵活设定。

增量同步索引库

推荐使用MQ（RabbitMQ）

原理：使用MQ做增量同步，即当修改数据之后就将此数据发送至MQ，由MQ将此数据同步到ES上。

Nginx

请解释一下什么是Nginx?

nginx本是一个web服务器和反向代理服务器，但由于丰富的负载均衡策略，常常被用于客户端可真实的服务器之间，作为负载均衡的实现。用于HTTP、HTTPS、SMTP、POP3和IMAP协议。

请列举nginx的一些特性

1. 反向代理/L7负载均衡器
2. 嵌入式Perl解释器
3. 动态二进制升级
4. 可用于重新编写URL，具有非常好的PCRE支持

nginx和apache的区别?

1. 轻量级，同样起web 服务，比apache 占用更少的内存及资源
2. 抗并发，nginx 处理请求是异步非阻塞的，而apache 则是阻塞型的，在高并发下nginx 能保持低资源低消耗高性能
3. 高度模块化的设计，编写模块相对简单
4. 最核心的区别在于apache是同步多进程模型，一个连接对应一个进程；nginx是异步的，多个连接（万级别）可以对应一个进程

nginx是如何实现高并发的?

一个主进程，多个工作进程，每个工作进程可以处理多个请求，每进来一个request，会有一个worker进程去处理。但不是全程的处理，处理到可能发生阻塞的地方，比如向上游（后端）服务器转发request，并等待请求返回。那么，这个处理的worker继续处理其他请求，而一旦上游服务器返回了，就会触发这个事件，worker才会来接手，这个request才会接着往下走。由于web server的工作性质决定了每个request的大部份生命都是在网络传输中，实际上花费在server机器上的时间片不多。这是几个进程就解决高并发的秘密所在。即@skoo所说的webserver刚好属于网络io密集型应用，不算是计算密集型。

请解释Nginx如何处理HTTP请求?

Nginx使用反应器模式。主事件循环等待操作系统发出准备事件的信号，这样数据就可以从套接字读取，在该实例中读取到缓冲区并进行处理。单个线程可以提供数万个并发连接。

在Nginx中，如何使用未定义的服务器名称来阻止处理请求?

只需将请求删除的服务器就可以定义为：Server {listen 80; server_name " " ;return 444;}这里，服务器名被保留为一个空字符串，它将在没有"主机"头字段的情况下匹配请求，而一个特殊的Nginx的非标准代码444被返回，从而终止连接。

7、 使用"反向代理服务器"的优点是什么?

答：反向代理服务器可以隐藏源服务器的存在和特征。它充当互联网云和web服务器之间的中间层。这对于安全方面来说是很好的，特别是当您使用web托管服务时。

8、请列举Nginx服务器的最佳用途?

Nginx服务器的最佳用法是在网络上部署动态HTTP内容，使用SCGI、WSGI应用程序服务器、用于脚本的FastCGI处理程序。它还可以作为负载均衡器。

9.请解释Nginx服务器上的Master和Worker进程分别是什么?

Master进程：读取及评估配置和维持Worker进程：处理请求

10、请解释你如何通过不同于80的端口开启Nginx?

为了通过一个不同的端口开启Nginx，你必须进入/etc/Nginx/sites-enabled/，如果这是默认文件，那么你必须打开名为“default”的文件。编辑文件，并放置在你想要的端口：Like server {listen 81;}

11.请解释是否有可能将Nginx的错误替换为502错误、503

502 =错误网关 503 =服务器超载 有可能，但是您可以确保fastcgi_intercept_errors被设置为ON，并使用错误页面指令。Location / { fastcgi_pass 127.0.01:9001; fastcgi_intercept_errors on; error_page 502 =503/error_page.html; #... }

12.在Nginx中，解释如何在URL中保留双斜线

要在URL中保留双斜线，就必须使用merge_slashes_off;语法:merge_slashes [on/off]默认值: merge_slashes on环境: http，server

13.请解释ngx_http_upstream_module的作用是什么?

ngx_http_upstream_module用于定义可通过fastcgi传递、proxy传递、uwsgi传递、memcached传递和scgi传递指令来引用的服务器组。

14.请解释什么是C10K问题?C10K问题是指无法同时处理大量客户端(10,000)的网络套接字。

15.请陈述stub_status和sub_filter指令的作用是什么?

1. Stub_status指令：该指令用于了解Nginx当前状态的当前状态，如当前的活动连接，接受和处理当前读/写/等待连接的总数
2. Sub_filter指令：它用于搜索和替换响应中的内容，并快速修复陈旧的数据

16、解释Nginx是否支持将请求压缩到上游?

您可以使用Nginx模块gunzip将请求压缩到上游。gunzip模块是一个过滤器，它可以对不支持“gzip”编码方法的客户机或服务器使用“内容编码:gzip”来解压缩响应。

17.解释如何在Nginx中获得当前的时间?

要获得Nginx的当前时间，必须使用SSI模块、$date_gmt和$date_local的变量。Proxy_set_header THE-TIME $date_gmt;

18.用Nginx服务器解释-s的目的是什么?

用于运行Nginx -s参数的可执行文件。

19.解释如何在Nginx服务器上添加模块?

在编译过程中，必须选择Nginx模块，因为Nginx不支持模块的运行时间选择。

20、什么是反向代理和正向代理?

正向代理：被代理的是客户端，比如通过XX代理访问国外的某些网站，实际上客户端没有权限访问国外的网站，客户端请求XX代理服务器，XX代理服务器访问国外网站，将国外网站返回的内容传给真正的用户。用户对于服务器是隐藏的，服务器并不知道真实的用户。

反向代理：被代理的是服务器，也就是客户端访问了一个所谓的服务器，服务器会将请求转发给后台真实的服务器，真实的服务器做出响应，通过代理服务器将结果返给客户端。服务器对于用户来说是隐藏的，用户不知道真实的服务器是哪个。

关于正向代理和反向代理，听起来比较绕，仔细理解，体会也不难明白到底是什么意思。

用nginx做实现服务的高可用，nginx本身可能成为单点，遇见的两种解决方案，一种是公司搭建自己的DNS，将请求解析到不同的NGINX，另一只是配合keepalive实现服务的存活检测。

Fastdfs:

简单介绍一下FastDFS?

1. 开源的分布式文件系统，主要对文件进行存储、同步、上传、下载，有自己的容灾备份、负载均衡、线性扩容机制；
2. FastDFS架构主要包含Tracker（跟踪） server和Storage（组，卷） server。客户端请求Tracker server进行文件上传、下载的时候，通过Tracker server调度最终由Storage server完成文件上传和下载。
3. Tracker server：跟踪器或者调度器，主要起负载均衡和调度作用。通过Tracker server在文件上传时可以根据一些策略找到Storage server提供文件上传服务。Storage server：存储服务器，作用主要是文件存储，完成文件管理的所有功能。客户端上传的文件主要保存在Storage server上，Storage server没有实现自己的文件系统而是利用操作系统的文件系统去管理文件。存储服务器采用了分组/分卷的组织方式。整个系统由一个组或者多个组组成；组与组之间的文件是相互独立的；所有组的文件容量累加就是整个存储系统的文件容量；一个组可以由多台存储服务器组成，一个组下的存储服务器中的文件都是相同的，组中的多台存储服务器起到了冗余备份和负载均衡的作用；在组内增加服务器时，如果需要同步数据，则由系统本身完成，同步完成之后，系统自动将新增的服务器切换到线上提供使用；当存储空间不足或者耗尽时，可以动态的添加组。只需要增加一台服务器，并为他们配置一个新的组，即扩大了存储系统的容量。我们在项目中主要使用fastdfs来存储整个项目的图片。

为什么要使用FastDFS作为你们的图片服务器?

首先基于fastDFS的特点：存储空间可扩展、提供了统一的访问方式、访问效率高、容灾性好 等特点，再结合我们项目中图片的容量大、并发大等特点，因此我们选择了FastDFS作为我们的图片服务器；

Nginx也可以作为一台图片服务器来使用，因为nginx可以作为一台http服务器来使用，作为网页静态服务器，通过location标签配置；在公司中有的时候也用ftp作为图片服务器来使用。

FastDFS中文件上传下载的具体流程?

客户端上传文件后生成一个file_id，返回给客户端，客户端利用这个file_id结合ip地址，生成一个完成图片的url，保存在数据库中。生成的那个file_id用于以后访问该文件的索引信息。FastDFS文件下载的流程

ActiveMQ

什么是activemq

activeMQ是一种开源的，面向消息的中间件，用来系统之间进行通信的

activemq的原理

原理就是生产者生产消息， 把消息发送给activemq。 Activemq 接收到消息， 然后查看有多少个消费者， 然后把消息转发给消费者， 此过程中生产者无需参与。 消费者接收到消息后做相应的处理和生产者没有任何关系

对比RabbitMQ

RabbitMQ的协议是AMQP，而ActiveMQ使用的是JMS协议。顾名思义JMS是针对Java体系的传输协议，队列两端必须有JVM，所以如果开发环境都是java的话推荐使用ActiveMQ，可以用Java的一些对象进行传递比如Map、Blob（二进制大数据）、Stream等。而AMQP通用行较强，非java环境经常使用，传输内容就是标准字符串。另外一点就是RabbitMQ用Erlang开发，安装前要装Erlang环境，比较麻烦。ActiveMQ解压即可用不用任何安装。

对比KafKa

Kafka性能超过ActiveMQ等传统MQ工具，集群扩展性好。
弊端是：
在传输过程中可能会出现消息重复的情况，
不保证发送顺序
一些传统MQ的功能没有，比如消息的事务功能。
所以通常用Kafka处理大数据日志。

对比Redis

其实Redis本身利用List可以实现消息队列的功能，但是功能很少，而且队列体积较大时性能会急剧下降。对于数据量不大、业务简单的场景可以使用。

如何解决消息重复问题

所谓消息重复,就是消费者接收到了重复的消息,一般来说我们对于这个问题的处理要把握下面几点

1. 消息不丢失(上面已经处理了)
2. 消息不重复执行

一般来说我们可以在业务段加一张表,用来存放消息是否执行成功,每次业务事物commit之后,告知服务端,已经处理过该消息,这样即使你消息重发了,也不会导致重复处理.

大致流程如下:

业务端的表记录已经处理消息的id,每次一个消息进来之前先判断该消息是否执行过,如果执行过就放弃,如果没有执行就开始执行消息,消息执行完成之后存入这个消息的id

获取session链接的时候 设置参数 默认不开启

持久化与非持久化

通过producer.setDeliveryMode(DeliveryMode.PERSISTENT) 进行设置
持久化的好处就是当activemq宕机的话，消息队列中的消息不会丢失。非持久化会丢失。但是会消耗一定的性能。

讲讲消息队列和消息被重复消费怎么处理，消费者接收不到消息怎么办

什么是消息队列?

就是消息的传输过程中保存消息的容器。

消息队列都解决了什么问题?

异步，并行，解耦，排队

消息模式?

订阅，点对点

一、重复消费：Queue支持存在多个消费者，但是对一个消息而言，只会有一个消费者 可以消费。

二、丢消息：

1. 用持久化消息
2. 非持久化消息及时处理不要堆积
3. 启动事务，启动事务后，commit()方法会负责等待服务器的返回，也就不会关闭连接 导致消息丢失。

三、消息重发：

消息被重新传递给客户端：

1. 使用事务会话，并调用滚退（）。
2. 在调用commit()之前关闭事务会话。
3. 会话使用CLIENT_ACKNOWLEDGE签收模式，并Session .recover()重发被调用。
4. 客户端连接超时（也许正在执行的代码要比配置的超时周期更长）。

四、不消费：去ActiveMQ.DLQ里找找

什么是ActiveMQ.DLQ?

1. .一旦消息的重发尝试超过了为重发策略配置的最大重发次数，一个“Poison ACK”被 发送回the broker，让 他知道消息被认为是毒丸。the broker然后接收消息并将其发送到死信队列，以便以后可以进行分析。
2. 在activemq中死信队列叫做ActiveMQ.DLQ。所有无法传递的消息将被发送到这个队列，这很难管理。
3. 因此，您可以在Activemq.xml配置文件的目标策略映射中设置个体死信策略，它允许 您为队列或主题指定 特定的死信队列前缀。

Mq消费者接受不到消息存在2中情况：

1. 处理失败 指的是MessageListener的onMessage方法里抛出RuntimeException。
2. Message头里有两个相关字段：Redelivered默认为false，redeliveryCounter默认为 0。
3. 消息先由broker发送给consumer，consumer调用listener，如果处理失败，本地 redeliveryCounter++，给broker一个特定应答，broker端的message里 redeliveryCounter++，延迟一点 时间继续调用，默认1s。超过6次，则给broker另一个特定应答，broker就直接发送消息到DLQ。
4. 如果失败2次，consumer重启，则broker再推过来的消息里， redeliveryCounter=2，本地只能再重试4次 即会进入DLQ。
5. 重试的特定应答发送到broker，broker即会在内存将消息的redelivered设置为 true，redeliveryCounter++，但是这两个字段都没有持久化，即没有修改存储中的消息 记录。所以broker重 启时这两个字段会被重置为默认值。

讲讲SOA与分布式的区别，zookeeper或者activeMQ服务挂了怎么办

SOA和分布式的区别?

SOA，将工程拆分成服务层、表现层两个工程，服务层中包含业务逻辑，只需要对外提 供服务即可。表现层只需要 处理和页面的交互，业务逻辑都是调用服务层的服务来实现。

分布式，主要还是从部署的角度，将应用按照访问压力进行归类，主要目标是充分利用 服务器的资源，避免资源分配不均

如果activeMQ的服务挂了，怎么办?

1. 在通常的情况下，非持久化消息是存储在内存中的，持久化消息是存储在文件中 的，它们的最大限制在配置文 件的节点中配置。但是，在非持久化消息堆积到一定程度，内存告急 的时候，ActiveMQ会将内存 中的非持久化消息写入临时文件中，以腾出内存。虽然都保存到了文件里，但它和持久 化消息的区别是，重启后持 久化消息会从文件中恢复，非持久化的临时文件会直接删除。
2. 考虑高可用，实现activemq集群。

如果zookeeper服务挂了怎么办? 注册中心对等集群，任意一台宕掉后，会自动切换到另一台 注册中心全部宕掉，服务提供者和消费者仍可以通过本地缓存通讯 服务提供者无状态，任一台宕机后，不影响使用 服务提供者全部宕机，服务消费者会无法使用，并无限次重连等待服务者恢复

activeMQ存在发出消息太大，造成消息接受不成功。多个线程从activeMQ中取消息，随着业务的扩大，该机器占用的网络带宽越

来越高。

仔细分析发现，mq入队时并没有异常高的网络流量，仅仅在出队时会产生很高的网络流量

最终发现是spring的jmsTemplate与activemq的prefetch机制配合导致的问题。研究源码发现jmsTemplate实现机制是：每次调用receive()时都会创建一个新的consumer对象，用完即销毁。

正常情况下仅仅会浪费重复创建consumer的资源代价，并不至于产生正常情况十倍百倍的网络流量。但是activeMQ有一个提高性能的机制prefetch，此时就会有严重的问题。prefetch机制：每次consumer连接至MQ时，MQ预先存放许多message到消费者（前提是MQ中存在大量消息），预先存放message的数量取决于prefetchSize（默认为1000）。此机制的目的很显然，是想让客户端代码用一个consumer反复进行 receive操作，这样能够大量提高出队性能。此机制与jmsTemplate配合时就会产生严重的问题，每次jmsTemplate.receive()，都会产生1000个消息的网络流量， 但是因为jmsTemplae并不会重用consumer，导致后面999个消息都被废弃。反复jmsTemplate.receive()时，表面上看 不出任何问题，其实网络带宽会造成大量的浪费。

解决方案:

1. 若坚持使用jmsTemplate，需要设置prefetch值为1，相当于禁用了activeMQ的prefetch机制，此时感觉最健壮， 就算多线程，反复调用jmsTemplate.receive()也不会有任何问题。但是会有资源浪费，因为要反复创建consumer并频繁与服务器进 行数据通信，但在性能要求不高的应用中也不算什么问题。
2. 不使用jmsTemplate，手工创建一个consumer，并单线程反复使用它来receive()，此时可以充分利用prefetch机制。配合多线程的方式每个线程拥有自己的一个consumer，此时能够充分发挥MQ在大吞吐量时的速度优势。切记避免多线程使用一个consumer造成的消息混乱。大吞吐量的应用推荐使用方案2，能够充分利用prefetch机制提高系MQ的吞吐性能

分布式

分布式架构：把系统按照模块拆分成多个子系统，多个子系统分布在不同的网络计算机上相互协作完成业务流程，系统之间需要进行通信

优点:

1. 把模块拆分，使用接口通信，降低模块之间的耦合度。
2. 把项目拆分成若干个子项目，不同的团队负责不同的子项目
3. 增加功能时只需要再增加一个子项目，调用其他系统的接口就可以.
4. 可以灵活的进行分布式部署。

缺点:

1. 系统之间交互需要使用远程通信，接口开发增加工作量。
2. 各个模块有一些通用的业务逻辑无法共用。

基于soa的架构

SOA：面向服务的架构。也就是把工程拆分成服务层、表现层两个工程。服务层中包含业务逻辑，只需要对外提供服务即可。表现层只需要处理和页面的交互，业务逻辑都是调用服务层的服务来实现。

分布式架构和soa架构有什么区别?

SOA，主要还是从服务的角度，将工程拆分成服务层、表现层两个工程。分布式，主要还是从部署的角度，将应用按照访问压力进行归类，主要目标是充分利用服务器的资源，避免资源分配不均

讲讲分布式事务的异步通信问题解决方案

问题介绍: 一个消息发送过去了，不管结果如何发送端都不会原地等待接收端。直到接 收端再推送回来回执消息， 发送端才直到结果。但是也有可能发送端消息发送后，石沉大海，杳无音信。这时候就 需要一种机制能够对这种不 确定性进行补充。

解决方案: 你给有很多笔友，平时写信一去一回，但是有时候会遇到迟迟没有回信的情况。那么针 对这种偶尔出现的情况，你 可以选择两种策略。一种方案是你发信的时候用定个闹钟，设定1天以后去问一下对方 收没收到信。另一种方案就 是每天夜里定个时间查看一下所有发过信但是已经一天没收到回复的信。然后挨个打个 电话问一下。 第一种策略就是实现起来就是延迟队列，第二种策略就是定时轮询扫描。 二者的区别是延迟队列更加精准，但是如果周期太长，任务留在延迟队列中时间的就会 非常长，会把队列变得冗 长。比如用户几天后待办提醒，生日提醒。 那么如果遇到这种长周期的事件，而且并不需要精确到分秒级的事件，可以利用定时扫 描来实现，尤其是比较消耗

讲讲怎么解单点登录的访问，分布式session跨域问题

单点登录是相互信任的系统模块登录一个模块后，其他模块不需要重复登录即认证通过。

采用CAS单点登录框架，首先CAS有两大部分：客户端和服务端。 服务端就是一个web工程部署在tomcat中。在服务端完成用户认证操作。每次访问系统 模块时，需要去CAS完成获 取ticket。当验证通过后，访问继续操作。对于CAS服务端来说，我们访问的应用模块 就是CAS客户端。

分布式架构session共享问题，如何在集群里边实现共享

用了CAS，所有应用项目中如果需要登录时在web.xml中配置过滤器做请求转发到cas端工作原理是在cas登录后会给浏览器发送一个票据（ticket），浏览器cookie中会缓存这个ticket，在登录其他项目时会拿着浏览器的ticket转发到cas，到cas后根据票据判断是否登录

项目中如何配置集群?

配置了redis集群，使用redis3.0版本官方推荐的配置方式solr集群使用了solrCloud，使用zookeeper关联solrCloud的配置文件
zookeeper也配置了集群应用层使用Nginx负载均衡

单点登录业务介绍

早期单一服务器，用户认证

缺点：单点性能压力，无法扩展

WEB应用集群，session共享模式

解决了单点性能瓶颈。

问题：

多业务分布式数据独立管理，不适合统一维护一份session数据。

分布式按业务功能切分，用户、认证解耦出来单独统一管理

cookie中使用jsessionId 容易被篡改、盗取。

跨顶级域名无法访问。

NQ

分布式，SSO(single sign on)模式

解决：

用户身份信息独立管理，更好的分布式管理。

可以自己扩展安全策略

跨域不是问题

缺点：

认证服务器访问压力较大

业务流程图

认证中心模块(oauth认证)

数据库表：user_info，并添加一条数据！密码应该是加密的！

在设计密码加密方式时 一般是使用MD5+盐的方式进行加密和解密。

登录功能

业务：

用接受的用户名密码核对后台数据库

将用户信息写入redis，redis中有该用户视为登录状态。

用userId+当前用户登录ip地址+密钥生成token

重定向用户到之前的来源地址，同时把token作为参数附上。

生成token

JWT工具

JWT（Json Web Token） 是为了在网络应用环境间传递声明而执行的一种基于JSON的开放标准。

JWT的声明一般被用来在身份提供者和服务提供者间传递被认证的用户身份信息，以便于从资源服务器获取资源。比如用在用户登录上

JWT 最重要的作用就是对 token信息的防伪作用。

JWT的原理，

一个JWT由三个部分组成：公共部分、私有部分、签名部分。最后由这三者组合进行

base64编码得到JWT。

公共部分

主要是该JWT的相关配置参数，比如签名的加密算法、格式类型、过期时间等等。

私有部分

用户自定义的内容，根据实际需要真正要封装的信息。

签名部分

根据用户信息+盐值+密钥生成的签名。如果想知道JWT是否是真实的只要把JWT的信息取出来，加上盐值和服务器中的密钥就可以验证真伪。所以不管由谁保存JWT，只要没有密钥就无法伪造。

例如:usrInfo+ip=密钥

base64编码，并不是加密，只是把明文信息变成了不可见的字符串。但是其实只要用一些工具就可以吧base64编码解成明文，所以不要在JWT中放入涉及私密的信息，因为实际上JWT并不是加密信息。

验证功能

功能：当业务模块某个页面要检查当前用户是否登录时，提交到认证中心，认证中心进行检查校验，返回登录状态、用户Id和用户名称。

业务：

1. 利用密钥和IP检验token是否正确，并获得里面的userId
2. 用userId检查Redis中是否有用户信息,如果有延长它的过期时间。
3. 登录成功状态返回

业务模块页面登录情况检查

问题：

1. 由认证中心签发的token如何保存?
2. 难道每一个模块都要做一个token的保存功能?
3. 如何区分请求是否一定要登录？ 使用的是拦截器

登录成功后将token写道cookie中

加入拦截器

首先这个验证功能是每个模块都要有的，也就是所有web模块都需要的。在每个controller方法进入前都需要进行检查。可以利用在springmvc中的拦截器功能。因为咱们是多个web模块分布式部署的，所以不能写在某一个web模块中，可以一个公共的web模块，就是gmall-web-util中。

检验方法是否需要验证用户登录状态

为了方便程序员在controller方法上标记，可以借助自定义注解的方式。比如某个controller方法需要验证用户登录，在方法上加入自定义的@LoginRequie。

CAS

CAS（Central Authentication Service），是耶鲁大学开发的单点登录系统（SSO，single sign-on），应用广泛，具有独立于平台的，易于理解，支持代理功能。CAS系统在各个大学如耶鲁大学、加州大学、剑桥大学、香港科技大学等得到应用CAS 的设计目标

(1)为多个Web应用提供单点登录基础设施，同时可以为非Web应用但拥有Web前端的功能服务提供单点登录的功能；

(2)简化应用认证用户身份的流程；

(3)将用户身份认证集中于单一的Web应用，让用户简化他们的密码管理，从而提高安全性；而且，当应用需要修改身份验证的业务逻辑时，不需要到处修改代码。

对分布式，dubbo，zookeeper说的不太清楚

分布式是从项目业务角度考虑划分项目整个架构。可以将项目基于功能模块划分再分别部署。Dubbo是实现分布式项目部署框架。在zookeeper是dubbo分布式框架的注册中心，管理服务的注册和调用。

从前端到后台的实现的过程描述的也不清楚

项目前端采用angularjs框架在controller控制器中完成数据组装和数据展示，在服务层（service）代码完成中后台请求操作。后端基于前端的接口调用，完成数据的增删改查操作。前后端数据交互通过json格式字符串完成。

Dubbo为什么选择Zookeeper，而不选择Redis

引入了ZooKeeper作为存储媒介，也就把ZooKeeper的特性引进来。首先是负载均衡，单注册中心的承载能力是有限的，在流量达到一定程度的时候就需要分流，负载均衡就是为了分流而存在的，一个ZooKeeper群配合相应的Web应用就可以很容易达到负载均衡；资源同步，单单有负载均衡还不够，节点之间的数据和资源需要同步，ZooKeeper集群就天然具备有这样的功能；命名服务，将树状结构用于维护全局的服务地址列表，服务提供者在启动 的时候，向ZK上的指定节点/dubbo/${serviceName}/providers目录下写入自己的URL地址，这个操作就完成了服务的发布。 其他特性还有Mast选举，分布式锁等。

项目中Zookeeper服务器挂了，服务调用可以进行吗

可以的，消费者在启动时，消费者会从zk拉取注册的生产者的地址接口等数据，缓存在本地。每次调用时，按照本地存储的地址进行调用

如何保证dubbo高可用?

zookeeper宕机与dubbo直连

在实际生产中，假如zookeeper注册中心宕掉，一段时间内服务消费方还是能够调用提供方的服务的，实际上它使用的本地缓存进行通讯，这只是dubbo健壮性的一种。
dubbo的健壮性表现：

1. 监控中心宕掉不影响使用，只是丢失部分采样数据
2. 数据库宕掉后，注册中心仍能通过缓存提供服务列表查询，但不能注册新服务
3. 注册中心对等集群，任意一台宕掉后，将自动切换到另一台
4. 注册中心全部宕掉后，服务提供者和服务消费者仍能通过本地缓存通讯
5. 服务提供者无状态，任意一台宕掉后，不影响使用
6. 服务提供者全部宕掉后，服务消费者应用将无法使用，并无限次重连等待服务提供者恢复

注册中心的作用在于保存服务提供者的位置信息，我们可以完全可以绕过注册中心——采用dubbo直连，即在服务消费方配置服务提供方的位置信息。

点对点直连方式，将以服务接口为单位，忽略注册中心的提供者列表，A 接口配置点对点，不影响 B 接口从注册中心获取列表。

xml配置方式

<dubbo:reference id="userService" interface="com.zang.gmall.service.UserService" url="dubbo://localhost:20880"/>

注解上直接添加

@Reference(url = "127.0.0.1:20880")

UserService userService;

<dubbo:service interface="com.zang.gmall.service.UserService"
<dubbo:method name="getUserAddressList" loadbalance="roundrobin">
/dubbo:method
/dubbo:service

权重设置

当不设置负载均衡策略，即采用默认的Random LoadBalance(随机均衡算法)时，默认每个服务的权重相同，我们可以通过设置权重来分配访问的随机性。权重默认为100，可以在暴露服务的同时设置

服务降级

当服务器压力剧增的情况下，根据实际业务情况及流量，对一些服务和页面有策略的不处理或换种简单的方式处理，从而释放服务器资源以保证核心交易正常运作或高效运作。

可以通过服务降级功能临时屏蔽某个出错的非关键服务，并定义降级后的返回策略（不调用服务即返回为空 or 调用失败返回为空）。

向注册中心写入动态配置覆盖规则：
RegistryFactory registryFactory =ExtensionLoader.getExtensionLoader(RegistryFactory.class).getAdaptiveExtension();
Registry registry =registryFactory.getRegistry(URL.valueOf("zookeeper://10.20.153.10:2181"));
registry.register(URL.valueOf("override://0.0.0.0/com.foo.BarService?category=configurators&dynamic=false&application=foo&mock=force:return+null"));

其中：
· mock=force:return+null 表示消费方对该服务的方法调用都直接返回 null 值，不发起远程调用。用来屏蔽不重要服务不可用时对调用方的影响。还可以改为 mock=fail:return+null 表示消费方对该服务的方法调用在失败后，再返回null 值，不抛异常。用来容忍不重要服务不稳定时对调用方的影响。通过操作管理控制台也可以方便的进行服务降级

集群容错

在集群调用失败时，Dubbo 提供了多种容错方案，缺省为 failover 重试。

集群容错模式主要有以下几种：

Failover Cluster

失败自动切换，当出现失败，重试其它服务器。通常用于读操作，但重试会带来更长延迟。可通过 retries="2" 来设置重试次数(不含第一次)。消费方服务级注解添加（不能到方法级）

Failfast Cluster

快速失败，只发起一次调用，失败立即报错。通常用于非幂等性的写操作，比如新增记录。

Failsafe Cluster

失败安全，出现异常时，直接忽略。通常用于写入审计日志等操作

Failback Cluster

失败自动恢复，后台记录失败请求，定时重发。通常用于消息通知操作

Forking Cluster

并行调用多个服务器，只要一个成功即返回。通常用于实时性要求较高的读操作，但需要浪费更多服务资源。可通过 forks="2" 来设置最大并行数。

Broadcast Cluster

广播调用所有提供者，逐个调用，任意一台报错则报错。通常用于通知所有提供者更新缓存或日志等本地资源信息。
集群模式配置方法在服务提供方和消费方配置集群模式

整合hystrix

Hystrix 旨在通过控制那些访问远程系统、服务和第三方库的节点，从而对延迟和故障提供更强大的容错能力。Hystrix具备拥有回退机制和断路器功能的线程和信号隔离，请求缓存和请求打包，以及监控和配置等功能。spring boot官方提供了对hystrix的集成，直接在pom.xml里加入依赖：

```
<dependency>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-starter-netflix-hystrix</artifactId>
<version>1.4.4.RELEASE</version>
</dependency>
```

然后在Application类上增加@EnableHystrix来启用hystrix starter：配置服务提供方：

在Dubbo的Provider上增加@HystrixCommand 配置，这样子调用就会经过Hystrix代理。

配置服务消费方：

对于Consumer端，则可以增加一层method调用，并在method上配置@HystrixCommand 。当调用出错时，会走到 fallbackMethod = "reliable" 的调用里。

@HystrixCommand注解设置的 reliable 调用方法的里的参数要和 method 的参数保持一致。

ActiveMq消息被重复消费，丢失，或者不消费怎么办

重复消费：Queue支持存在多个消费者，但是对一个消息而言，只会有一个消费者可以消费。

丢消息：用持久化消息，或者非持久化消息及时处理不要堆积，或者启动事务，启动事务后，commit() 方法会负责任的等待服务器的返回，也就不会关闭连接导致消息丢失了。

不消费：去ActiveMQ.DLQ里找找

怎样解决activeMQ的消息持久化问题?

持久化为文件

这个你装ActiveMQ时默认就是这种，只要你设置消息为持久化就可以了。涉及到的配置和代码有

```
<persistenceAdapter>
<kahaDB directory="${activemq.base}/data/kahadb"/>
</persistenceAdapter>
```

producer.Send(request, MsgDeliveryMode.Persistent, level, TimeSpan.MinValue);

持久化为MySql

加载驱动jar，为数据中创建三个数据库表，存储activemq的消息信息

如果activeMQ的消息没有发送成功，怎样确保再次发送成功

重新传递消息的情况

ActiveMQ在接收消息的Client有以下几种操作的时候，需要重新传递消息：

1：Client用了transactions（事务），且在session中调用了rollback()

2：Client用了transactions，且在调用commit()之前关闭

3：Client在CLIENT_ACKNOWLEDGE的传递模式下，在session中调用了recover()确保客户端有几种状态，检测状态，只要提交了那就说明客户端成功！

Zookeeper怎样进行服务治理

接受提供者的接口信息和提供者ip地址进行存储，然后管理消费者和提供者之间调用关系！

如果activeMQ的服务挂了，怎么办?

1. 在通常的情况下，非持久化消息是存储在内存中的，持久化消息是存储在文件中的，它们的最大限制在配置文件的节点中配置。但是，在非持久化消息堆积到一定程度，内存告急的时候，ActiveMQ会将内存中的非持久化消息写入临时文件中，以腾出内存。虽然都保存到了文件里，但它和持久化消息的区别是，重启后持久化消息会从文件中恢复，非持久化的临时文件会直接删除。
2. 考虑高可用，实现activemq集群。

如果zookeeper服务挂了怎么办?

注册中心对等集群，任意一台宕掉后，会自动切换到另一台 注册中心全部宕掉，服务提供者和消费者仍可以通过本地缓存通讯
服务提供者无状态，任一台宕机后，不影响使用服务提供者全部宕机，服务消费者会无法使用，并无限次重连等待服务者恢复

mq消费者接收不到消息怎么办

Mq消费者接受不到消息存在2中情况：

1. 处理失败 指的是MessageListener的onMessage方法里抛出RuntimeException。
2. Message头里有两个相关字段：Redelivered默认为false，redeliveryCounter默认为0
3. 消息先由broker发送给consumer，consumer调用listener，如果处理失败，本地redeliveryCounter++，给broker一个特定应答，broker端的message里redeliveryCounter++，延迟一点时间继续调用，默认1s。超过6次，则给broker另一个特定应答，broker就直接发送消息到DLQ。

4. 如果失败2次，consumer重启，则broker再推过来的消息里redeliveryCounter=2，本地只能再重试4次即会进入DLQ。
5. 重试的特定应答发送到broker，broker即会在内存将消息的redelivered设置为true，redeliveryCounter++，但是这两个字段都没有持久化，即没有修改存储中的消息记录。所以broker重启时这两个字段会被重置为默认值。

系统的高并发问题是怎么解决的

并发问题高，这个问题的解决方案是一个系统性的，系统的每一层面都需要做优化：

1. 数据层
2. 集群
3. 分表分库
4. 开启索引
5. 开启缓存
6. 表设计优化
7. Sql语句优化
8. 缓存服务器（提高查询效率，减轻数据库压力）
9. 搜索服务器（提高查询效率，减轻数据库压力）

项目层

1. 采用面向服务分布式架构（分担服务器压力，提高并发能力）
2. 采用并发访问较高的详情系统采用静态页面
3. 使用页面缓存
4. 用ActiveMQ使得业务进一步进行解耦，提高业务处理能力
5. 使用分布式文件系统存储海量文件
6. 应用层
7. Nginx服务器来做负载均衡
8. Lvs做二层负载

并发数多少，项目中怎么解决并发问题?

面试中项目的并发数不宜说的过大，安装目前谷粒商城项目拆分规模，这个项目的并发是在10000+，但是学生面试不能说的这么高。

可以有以下2方面的回答：

1. 项目并发并不清楚（只是底层程序员）
2. 参与核心业务设计，知道并发是多少（测试峰值，上线并发）3000---5000吧

消息发送失败怎么处理，发送数据，数据库已经保存了数据，但是redis中没有同步，怎么办。或者说如何做到消息同步。

消息发送失败，可以进行消息的重新发送，可以配置消息的重发次数

如果消息重发完毕后，消息还没有接受成功，重启服务。

Dubbo的通信原理?

Dubbo底层使用hessain2进行二进制序列化进行远程调用

Dubbo底层使用netty框架进行异步通信。NIO

单点登录的访问或者跨域问题

首先要理解什么是单点登录。单点登录是相互信任的系统模块登录一个模块后，其他模块不需要重复登录即认证通过。项目采用的是CAS单点登录框架完成的。首先CAS有两大部分。客户端和服务端。服务端就是一个web工程部署在tomcat中。在服务端完成用
户认证操作。每次访问系统模块时，需要去CAS完成获取ticket。当验证通过后，访问继续操作。对于CAS服务端来说，我们访问的应用模块就是CAS客户端。

跨域问题，首先明白什么是跨域。什么时候涉及跨域问题。当涉及前端异步请求的时候才涉及跨域。那什么是跨域呢？当异步请求时，访问的请求地址的协议、ip地址、端口号任意一个与当前站点不同时，就会涉及跨域访问。解决方案：1、jQuery提供了jsonp实现2、W3C标准提供了CORS（跨域资源共享）解决方案。

shiro安全认证时如何做的

要明白shiro执行流程以及shiro的核心组件

认证过程：

在application Code应用程序中调用subject的login方法。将页面收集的用户名和密码传给安全管理器securityManager，将用户名传给realm对象。Realm对象可以理解为是安全数据桥，realm中认证方法基于用户名从数据库中查询用户信息。如果用户存在，将数据库查询密码返回给安全管理器securityManager，然后安全管理器判断密码是否正确。

ES的用途

ES在系统中主要完成商品搜索功能，提高搜索性能

分布式锁的问题

针对分布式锁的实现，目前比较常用的有以下几种方案：

1. 基于数据库实现分布式锁
2. .基于缓存（redis，memcached，tair）实现分布式锁
3. 基于zookeeper实现分布式锁

ES索引中使用了IK分词器，你们项目中使用到了分词器的哪种工作模式

IK分词器，基本可分为两种模式，一种为smart模式，一种为非smart模式。

例如：张三说的确实在理

smart模式的下分词结果为：

张三 | 说的 | 确实 | 在理

而非smart模式下的分词结果为：

张三 | 三 | 说的 | 的确 | 的 | 确实 | 实在 | 在理

可见非smart模式所做的就是将能够分出来的词全部输出；smart模式下，IK分词器则会根据内在方法输出一个认为最合理的分词结果，这就涉及到了歧义判断。项目中采用的是smart模块分词的。

什么是跨域?

当异步请求时，访问的请求地址的协议、ip地址、端口号任意一个与当前站点不同时， 就会涉及跨域访问。 什么时候涉及跨域问题？当涉及前端异步请求的时候才涉及跨域。

解决方案：

1. jQuery提供了jsonp实现
2. W3C标准提供了CORS（跨域资源共享）解决方案。用了CAS，所有应用项目中如果需要登录时在web.xml中配置过滤器做请求转发到cas端 工作原理是在cas登录后会 给浏览器发送一个票据（ticket），浏览器cookie中会缓存这个ticket，在登录其他项 目时会拿着浏览器的 ticket转发到cas，到cas后根据票据判断是否登录

Linux

讲讲linux命令awk、cat、sort、cut、grep、uniq、wc、top、 find、sed等作用

awk:相较于sed 常常作用于一整个行的处理，awk 则比较倾向于一行当中分成数个『字 段』来处理。因此，awk相当的适合处理小型的数据数据处理

cat:主要用来查看文件内容，创建文件，文件合并，追加文件内容等功能。

sort:功能：排序文本，默认对整列有效

cut:cut命令可以从一个文本文件或者文本流中提取文本列

grep:是一种强大的文本搜索工具，它能使用正则表达式搜索文本，并把匹配的行打印出来

uniq:功能：去除重复行，只会统计相邻的

wc:功能： 统计文件行数、字节、字符数

top:用来监控Linux的系统状况,比如cpu、内存的使用

find:功能： 搜索文件目录层次结构

sed:sed 是一种在线编辑器，它一次处理一行内容

讲讲什么是死锁，怎么解决死锁，表级锁和行级锁，悲观锁与乐观 锁以及线程同步锁区别

死锁：打个比方，你去面试，面试官问你，你告诉我什么是死锁我就让你进公司。你回 答说你让我进公司我就告诉你什么是死锁

互斥条件：资源不能被共享，只能由一个进程使用。

请求与保持条件：进程已获得了一些资源，但因请求其它资源被阻塞时，对已获得的资 源保持不放。

不可抢占条件：有些系统资源是不可抢占的，当某个进程已获得这种资源后，系统不能 强行收回，只能由进程使用完时自己释放。

循环等待条件：若干个进程形成环形链，每个都占用对方申请的下一个资源

(1) 死锁预防：破坏导致死锁必要条件中的任意一个就可以预防死锁。例如，要求用户 申请资源时一次性申请 所需要的全部资源，这就破坏了保持和等待条件；将资源分层，得到上一层资源后，才 能够申请下一层资源， 它破坏了环路等待条件。预防通常会降低系统的效率。

(2) 死锁避免：避免是指进程在每次申请资源时判断这些操作是否安全，例如，使用银行 家算法。死锁避免算 法的执行会增加系统的开销。

(3) 死锁检测：死锁预防和避免都是事前措施，而死锁的检测则是判断系统是否处于死 锁状态，如果是，则执 行死锁解除策略。

(4) 死锁解除：这是与死锁检测结合使用的，它使用的方式就是剥夺。即将某进程所拥有 的资源强行收回，分 配给其他的进程。

表级锁： 开销小，加锁快；不会出现死锁(因为MyISAM会一次性获得SQL所需的全部 锁)；锁定粒度大，发生锁冲 突的概率最高,并发度最低。

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

悲观锁：总是假设最坏的情况，每次去拿数据的时候都认为别人会修改，所以每次在拿 数据的时候都会上锁，这样 别人想拿这个数据就会阻塞直到它拿到锁。传统的关系型数据库里边就用到了很多这种 锁机制，比如行锁，表锁 等，读锁，写锁等，都是在做操作之前先上锁。再比如Java里面的同步原语 synchronized关键字的实现也是悲观 锁。通过for update来实现

乐观锁：顾名思义，就是很乐观，每次去拿数据的时候都认为别人不会修改，所以不会 上锁，但是在更新的时候会 判断一下在此期间别人有没有去更新这个数据，可以使用版本号等机制。乐观锁适用于 多读的应用类型，这样可以 提高吞吐量，像数据库提供的类似于write_condition机制，其实都是提供的乐观锁。在 Java中 java.util.concurrent.atomic包下面的原子变量类就是使用了乐观锁的一种实现方式 CAS 实现的。通过version版本字段来实现

同步锁： 场景：在开发中,遇到耗时的操作,我们需要把耗时的逻辑放入子线程中执行,防止卡顿。 二个线程分别执行两个 任务，同时执行完成,同时解析文件,获取数据后,同时插入数据库,由于插入的表比较多，这样容易出现插入错 乱的bug

采用synchronized：

声明该方法为同步方法,如果一个方法正在执行,别的方法调用,则处于等待状态。当这个 方法执行完成后,可以 调用解锁方法,wait():释放占有的对象锁，线程进入等待池。

区别：synchronized是在JVM层面实现的,因此系统可以监控锁的释放与否,而ReentrantLock 使用代码实现的,系统 无法自动释放锁,需要在代码中finally子句中显式释放锁lock.unlock(); 在并发量比较小的情况下，使用synchronized是个不错的选择，但是在并发量比较高的 情况下，其性能下降 很严重，此 ReentrantLock是个不错的方案。

讲讲怎么加快访问速度，怎样进行程序性能调优

加快访问：

硬件上加大网络带宽、和服务器内存 代码的处理：静态页面、缓存、优化sql、创建索引等方案

系统性能就是两个事：

Throughput ，吞吐量。也就是每秒钟可以处理的请求数，任务数。 Latency， 系统延迟。也就是系统在处理一个请求或一个任务时的延迟。 那么Latency越好，能支持的Throughput就会越高。因为 Latency短说明处理速度快， 于是就可以处理更多的请求。 提高吞吐量：分布式集群，模块解藕，设计模式 系统延迟：异步通信

activeMQ存在运行时间长了以后，收不到消息的现象。时间长了就会出现，卡死，新的数据不能从队列接听到。只能重启程序。

解决方案：

不要频繁的建立和关闭连接：JMS使用长连接方式，一个程序，只要和JMS服务器保持一个连接就可以了，不要频繁的建立和关闭连接。频繁的建立和关闭连接，对程序的性能影响还是很大的。这一点和jdbc还是不太一样的。

redis

1. 降低后端负载：对于高消耗的SQL：join结果集、分组统计结果；对这些结果进行缓 存。
2. 加速请求响应
3. .大量写合并为批量写：如计数器先redis累加再批量写入DB
4. 超时剔除：例如expire
5. 主动更新：开发控制生命周期（最终一致性，时间间隔比较短）
6. 缓存空对象
7. 布隆过滤器拦截
8. 命令本身的效率：例如sql优化，命令优化
9. 网络次数：减少通信次数
10. 降低接入成本:长连/连接池,NIO等。
11. IO访问合并

目的：要减少缓存重建次数、数据尽可能一致、减少潜在危险。

解决方案：

1. 互斥锁setex,setnx：
    1. 如果 set(nx 和 ex) 结果为 true，说明此时没有其他线程重建缓存，那么当前线程执行 缓存构建逻辑。
    2. 如果 setnx(nx 和 ex) 结果为 false，说明此时已经有其他线程正在执行构建缓存的工作，那么当前线程将休 息指定时间 (例如这里是 50 毫秒，取决于构建缓存的速度) 后，重新执行函数，直到 获取到数据
2. 永远不过期：
    1. 热点key,无非是并发特别大一级重建缓存时间比较长，如果直接设置过期时间，那么时 间到的时候，巨大的访 问量会压迫到数据库上，所以要给热点key的val增加一个逻辑过期时间字段，并发访问 的时候，判断这个逻辑 字段的时间值是否大于当前时间，大于了说明要对缓存进行更新了，那么这个时候，依 然让所有线程访问老的 缓存，因为缓存并没有设置过期，但是另开一个线程对缓存进行重构。等重构成功，即 执行了redis set操作 之后，所有的线程就可以访问到重构后的缓存中的新的内容了 从缓存层面来看，确实没有设置过期时间，所以不会出现热点 key 过期后产生的问题， 也就是“物理”不过期。 从功能层面来看，为每个 value 设置一个逻辑过期时间，当发现超过逻辑过期时间后， 会使用单独的线程去构建缓存。

    一致性问题：

    1. .先删除缓存，然后在更新数据库，如果删除缓存失败，那就不要更新数据库，如果说 删除缓存成功，而更新 数据库失败，那查询的时候只是从数据库里查了旧的数据而已，这样就能保持数据库与 缓存的一致性。
    2. .先去缓存里看下有没有数据，如果没有，可以先去队列里看是否有相同数据在做更 新，发现队列里有一个请 求了，那么就不要放新的操作进去了，用一个while（true）循环去查询缓存，循环个 200MS左右再次发送到 队列里去，然后同步等待缓存更新完成。

讲到redis缓存的时候说不清楚

1. redis中项目中的应用。1.主要应用在门户网站首页广告信息的缓存。因为门户网站访问量较大，将广告缓存到redis中，可以降低数据库访问压力，提高查询性能。2.应用在用户注册验证码缓存。利用redis设置过期时间，当超过指定时间后，redis清理验证码，使过期的验证码无效。3.用在购物车模块，用户登陆系统后，添加的购物车数据需要保存到redis缓存中。
2. 技术角度分析：
    1. 内存如果满了，采用LRU算法进行淘汰
    2. Redis如何实现负载的？采用Hash槽来运算存储值，使用CRC16算法取模运算，来保证负载问题。
    3. Redis缓存穿透问题？将数据查询出来如果没有强制设置空值，并且设置过期时间，减少频繁查询数据库。

能讲下redis的具体使用场景吗？使用redis存储长期不改变的数据完全可以使用也看静态化，那么你们当时是为什么会使用redis?

redis在项目中应用：1.主要应用在门户网站首页广告信息的缓存。因为门户网站访问量较大，将广告缓存到redis中，可以降低数据库访问压力，提高查询性能。2.应用在用户注册验证码缓存。利用redis设置过期时间，当超过指定时间后，redis清理验证码，使过期的验证码无效。3.用在购物车模块，用户登陆系统后，添加的购物车数据需要保存到redis缓存中。
使用redis主要是减少系统数据库访问压力。从缓存中查询数据，也提高了查询性能，挺高用户体验度。

redis中对一个key进行自增或者自减操作，它是原子性的吗?

是原子性的。对于Redis而言，命令的原子性指的是：一个操作的不可以再分，操作要么执行，要么不执行。Redis的操作之所以是原子性的，是因为Redis是单线程的。对Redis来说，执行get、set以及eval等API，都是一个一个的任务，这些任务都会由Redis的线程去负责执行，任务要么执行成功，要么执行失败，这就是Redis的命令是原子性的原因。Redis本身提供的所有API都是原子操作，Redis中的事务其实是要保证批量操作的原子性。

Redis分布式锁理解

实现思想

获取锁的时候，使用setnx加锁，并使用expire命令为锁添加一个超时时间，超过该时间则自动释放锁，锁的value值为一个随机生成的UUID，通过此在释放锁的时候进行判断。

获取锁的时候还设置一个获取的超时时间，若超过这个时间则放弃获取锁。

释放锁的时候，通过UUID判断是不是该锁，若是该锁，则执行delete进行锁释放。

Redis怎么设置过期的？项目过程中，使用了哪一种持久化方式

设置过期：

this.redisTemplate.expire("max",tempTime,TimeUnit.SECONDS);

持久化方式：Redis默认的RDB方式

项目添加Redis缓存后，持久化具体怎么实现的。

RDB：保存存储文件到磁盘；同步时间为15分钟，5分钟，1分钟一次，可能存在数据丢失问题。

AOF：保存命令文件到磁盘；安全性高，修改后立即同步或每秒同步一次。

上述两种方式在我们的项目中都有使用到，在广告轮播的功能中使用了redis缓存，先从redis中获取数据，无数据后从数据库中查询后保存到redis中采用默认的RDB方式，在广告轮播的功能中使用了redis缓存，先从redis中获取数据，无数据就从数据库中查询后再保存到redis中

项目中有用到过redis，访问redis是通过什么访问的？redis能够存储的数据类型有哪几种？

Redis通过SpringDataRedis访问的.

Redis支持五种数据类型：string（字符串），hash（哈希），list（列表），set（集合）及zset(sorted set：有序集合)

简单介绍一个redis?

redis是内存中的数据结构存储系统，一个key-value类型的非关系型数据库，可持久化的数据库，相对于关系型数据库（数据主要存在硬盘中），性能高，因此我们一般用redis来做缓存使用；并且redis支持丰富的数据类型，比较容易解决各种问题，因此redis可以用来作为注册中心，数据库、缓存和消息中间件。Redis的Value支持5种数据类型，string、hash、list、set、zset（sorted set）；

String类型：一个key对应一个value

Hash类型：它的key是string类型，value又是一个map（key-value），适合存储对象。

List类型：按照插入顺序的字符串链表（双向链表），主要命令是LPUSH和RPUSH，能够支持反向查找和遍历

Set类型：用哈希表类型的字符串序列，没有顺序，集合成员是唯一的，没有重复数据，底层主要是由一个value永远为null的hashmap来实现的。

zset类型：和set类型基本一致，不过它会给每个元素关联一个double类型的分数（score），这样就可以为成员排序，并且插入是有序的。

Memcache和redis的区别：

数据支持的类型：redis不仅仅支持简单的k/v类型的数据，同时还支持list、set、zset、hash等数据结构的存储；memcache只支持简单的k/v类型的数据，key和value都是string类型

可靠性：memcache不支持数据持久化，断电或重启后数据消失，但其稳定性是有保证的；redis支持数据持久化和数据恢复，允许单点故障，但是同时也会付出性能的代价
性能上：对于存储大数据，memcache的性能要高于redis

应用场景：

Memcache：适合多读少写，大数据量的情况（一些官网的文章信息等）

Redis：适用于对读写效率要求高、数据处理业务复杂、安全性要求较高的系统

案例：分布式系统，存在session之间的共享问题，因此在做单点登录的时候，我们利用redis来模拟了session的共享，来存储用户的信息，实现不同系统的session共享；

对redis的持久化了解不?

redis的持久化方式有两种：

RDB（半持久化方式）：按照配置不定期的通过异步的方式、快照的形式直接把内存中的数据持久化到磁盘的一个dump.rdb文件（二进制的临时文件）中，redis默认的持久化方式，它在配置文件（redis.conf）中。

优点：只包含一个文件，将一个单独的文件转移到其他存储媒介上，对于文件备份、灾难恢复而言，比较实用。

缺点：系统一旦在持久化策略之前出现宕机现象，此前没有来得及持久化的数据将会产生丢失

RDB持久化配置：

Redis会将数据集的快照dump到dump.rdb文件中。此外，我们也可以通过配置文件来修改Redis服务器dump快照的频率，在打开6379.conf文件之后，我们搜索save，可以看到下面的配置信息：

save 900 1 在15分钟之后，如果1个key发生变化，则dump内存快照

AOF（全持久化的方式）： 把每一次数据变化都通过write()函数将你所执行的命令追加到一个appendonly.aof文件里面，Redis默认是不支持这种全持久化方式的，需要在配置文件（redis.conf）中将appendonly no改成appendonly yes

优点：数据安全性高，对日志文件的写入操作采用的是append模式，因此在写入过程中即使出现宕机问题，也不会破坏日志文件中已经存在的内容；

缺点：对于数量相同的数据集来说，aof文件通常要比rdb文件大，因此rdb在恢复大数据集时的速度大于AOF；

AOF持久化配置:

在Redis的配置文件中存在三种同步方式，它们分别是：

appendfsync always 每次有数据修改就会调用fsync刷新到aof文件，非常慢，但是安全

appendfsync everysec　每秒

appendfsync no　依靠os进行刷新

二种持久化方式区别：

AOF在运行效率上往往慢于RDB，每秒同步策略的效率是比较高的，同步禁用策略的效率和RDB一样高效；

如果缓存数据安全性要求比较高的话，用aof这种持久化方式（比如项目中的购物车）；

如果对于大数据集要求效率高的话，就可以使用默认的。而且这两种持久化方式可以同时使用。

做过redis的集群吗？你们做集群的时候搭建了几台，都是怎么搭建的

Redis的数据是存放在内存中的，不适合存储大数据，大数据存储一般公司常用hadoop中的Hbase或者MogoDB。redis主要用来处理高并发的，用我们的项目来说，电商项目如果并发大的话，一台单独的redis是不能足够支持我们的并发，这就需要我们扩展多台
设备协同合作，即用到集群。

Redis搭建集群的方式有多种，例如：客户端分片、Twemproxy、Codis等，但是redis3.0之后就支持redis-cluster集群，这种方式采用的是无中心结构，每个节点保存数据和整个集群的状态，每个节点都和其他所有节点连接。如果使用的话就用rediscluster集群。集群这块是公司运维搭建的，具体怎么搭建不是太了解。

我们项目中redis集群主要搭建了6台，3主（为了保证redis的投票机制）3从（高可用），每个主服务器都有一个从服务器，作为备份机。所有的节点都通过PING-PONG机制彼此互相连接；客户端与redis集群连接，只需要连接集群中的任何一个节点即可；

Redis-cluster中内置了16384个哈希槽，Redis-cluster把所有的物理节点映射到【0-16383】slot上，负责维护。

redis有事务吗?

Redis是有事务的，redis中的事务是一组命令的集合，这组命令要么都执行，要不都不执行，保证一个事务中的命令依次执行而不被其他命令插入。redis的事务是不支持回滚操作的。redis事务的实现，需要用到MULTI（事务的开始）和EXEC（事务的结束）命
令；

缓存穿透

缓存查询一般都是通过key去查找value，如果不存在对应的value，就要去数据库中查找。如果这个key对应的value在数据库中也不存在，并且对该key并发请求很大，就会对数据库产生很大的压力，这就叫缓存穿透

解决方案：

对所有可能查询的参数以hash形式存储，在控制层先进行校验，不符合则丢弃。

.将所有可能存在的数据哈希到一个足够大的bitmap中，一个一定不存在的数据会被这 个bitmap拦截掉，从而避免了对底层存储系统的查询压力。

如果一个查询返回的数据为空（不管是数 据不存在，还是系统故障），我们仍然把这个空结果进行缓存，但它的过期时间会很短，最长不超过五分钟。

缓存雪崩

当缓存服务器重启或者大量缓存集中在一段时间内失效，发生大量的缓存穿透，这样在失效的瞬间对数据库的访问压力就比较大，所有的查询都落在数据库上，造成了缓存雪崩。 这个没有完美解决办法，但可以分析用户行为，尽量让失效时间点均匀分布。大多数系统设计者考虑用加锁或者队列的方式保证缓存的单线程（进程）写，从而避免失效时大量的并发请求落到底层存储系统上。

解决方案：

在缓存失效后，通过加锁或者队列来控制读数据库写缓存的线程数量。比如对某个key只允许一个线程查询数据和写缓存，其他线程等待。

可以通过缓存reload机制，预先去更新缓存，再即将发生大并发访问前手动触发加载缓存

不同的key，设置不同的过期时间，让缓存失效的时间点尽量均匀

做二级缓存，或者双缓存策略。A1为原始缓存，A2为拷贝缓存，A1失效时，可以访问A2，A1缓存失效时间设置为短期，A2设置为长期。

redis的安全机制（你们公司redis的安全这方面怎么考虑的？）

漏洞介绍：redis默认情况下，会绑定在bind 0.0.0.0:6379，这样就会将redis的服务暴露到公网上，如果在没有开启认证的情况下，可以导致任意用户在访问目标服务器的情况下，未授权就可访问redis以及读取redis的数据，攻击者就可以在未授权访问redis的
情况下可以利用redis的相关方法，成功在redis服务器上写入公钥，进而可以直接使用私钥进行直接登录目标主机；

解决方案：

1.禁止一些高危命令。修改redis.conf文件，用来禁止远程修改DB文件地址，比如rename-command FLUSHALL "" 、rename-command CONFIG"" 、renamecommand EVAL “”等；

2.以低权限运行redis服务。为redis服务创建单独的用户和根目录，并且配置禁止登录；
3.为redis添加密码验证。修改redis.conf文件，添加requirepass mypassword；

4.禁止外网访问redis。修改redis.conf文件，添加或修改 bind 127.0.0.1，使得redis服务只在当前主机使用；

5.做log监控，及时发现攻击；

6.redis的哨兵机制（redis2.6以后出现的）

哨兵机制：

监控：监控主数据库和从数据库是否正常运行；

提醒：当被监控的某个redis出现问题的时候，哨兵可以通过API向管理员或者其他应用程序发送通知；

自动故障迁移：主数据库出现故障时，可以自动将从数据库转化为主数据库，实现自动切换；

具体的配置步骤参考的网上的文档。要注意的是，如果master主服务器设置了密码，记得在哨兵的配置文件（sentinel.conf）里面配置访问密码

redis中对于生存时间的应用

Redis中可以使用expire命令设置一个键的生存时间，到时间后redis会自动删除；应用场景：

1. 设置限制的优惠活动的信息；
2. . 一些及时需要更新的数据，积分排行榜；
3. 手机验证码的时间；
4. 限制网站访客访问频率；

集群

一个集群系统是一群松散结合的服务器组，形成一个虚拟的服务器，为客户端用户提供统一的服务。对于这个客户端来说，通常在访问集群系统时不会意识到它的服务是由具体的哪一台服务器提供。集群的目的，是为实现负载均衡、容错和灾难恢复。以达到系统可用性和可伸缩性的要求。集群系统一般应具高可用性、可伸缩性、负载均衡、故障恢复和可维护性等特殊性能。一般同一个工程会部署到多台服务

器上。常见的tomcat集群，Redis集群，Zookeeper集群，数据库集群

分布式与集群的区别：

分布式是指将不同的业务分布在不同的地方。 而集群指的是将几台服务器集中在一起，
实现同一业务。一句话：分布式是并联工作的，集群是串联工作的。

分布式中的每一个节点，都可以做集群。 而集群并不一定就是分布式的。

高并发

处理高并发常见的方法有哪些?

1. HTML静态化 freemaker

   其实大家都知道，效率最高、消耗最小的就是纯静态化的html页面，所以我们尽可能使我们的网站上的页面采 用静态页面来实现，这个最简单的方法其实也是最有效的方法。但是对于大量内容并且频繁更新的网站，我们无法全部手动去挨个实现，于是出现了我们常见的信息 发布系统CMS，像我们常访问的各个门户站点的新闻频道，甚至他们的其他频道，都是通过信息发布系统来管理和实现的，信息发布系统可以实现最简单的信息录 入自动生成静态页面，还能具备频道管理、权限管理、自动抓取等功能，对于一个大型网站来说，拥有一套高效、可管理的CMS是必不可少的。除了门户和信息发布类型的网站，对于交互性要求很高的社区类型网站来说，尽可能的
   静态化也是提高性能的必要手段，将社区内的帖子、文章进行实时的静态化，有更新的时候再重新静态化也是大量使用的策略，像Mop的大杂烩就是使用了这样的策略，网易社区等也是如此。
   同 时，html静态化也是某些缓存策略使用的手段，对于系统中频繁使用数据库查询但是内容更新很小的应用，可以考虑使用html静态化来实现，比如论坛中论 坛的公用设置信息，这些信息目前的主流论坛都可以进行后台管理并且存储再数据库中，这些信息其实大量被前台程序调用，但是更新频率很小，可以考虑将这部分 内容进行后台更新的时候进行静态化，这样避免了大量的数据库访问请求。

2. 图片服务器分离

   大家知道，对于Web服务器来说，不管 是 Apache、IIS还是其他容器，图片是最消耗资源的，于是我们有必要将图片与页面进行分离，这是基本上大型网站都会采用的策略，他们都有独立的图片服务器，甚至很多台图片服务器。这样的架构可以降低提供页面访问请求的服务器系统压力，并且可以保证系统不会因为图片问题而崩溃，在应用服务器和图片服务器 上，可以进行不同的配置优化，比如apache在配置ContentType的时候可以尽量少支持，尽可能少的LoadModule，保证更高的系统消耗 和执行效率。

3. 数据库集群和库表散列

   大型网站都有复杂的应用，这些应用必须使用数据库，那么在面对大量访问的时候，数据库的瓶颈很快就能显现出来，这时一台数据库将很快无法满足应用，于是我们需要使用数据库集群或者库表散列。
   在数据库集群方面，很多数据库都有自己的解决方案，Oracle、Sybase等都有很好的方案，常用的MySQL提供的Master/Slave也是类似的方案，您使用了什么样的DB，就参考相应的解决方案来实施即可。
   上 面提到的数据库集群由于在架构、成本、扩张性方面都会受到所采用DB类型的限制，于是我们需要从应用程序的角度来考虑改善系统架构，库表散列是常用并且最有效的解决方案。我们在应用程序中安装业务和应用或者功能模块将数据库进行分离，不同的模块对应不同的数据库或者表，再按照一定的策略对某个页面或者功能 进行更小的数据库散列，比如用户表，按照用户ID进行表散列，这样就能够低成本的提升系统的性能并且有很好的扩展性。sohu的论坛就是采用了这样的架构，将论坛的用户、设置、帖子等信息进行数据库分离，然后对帖子、用户按照板块和ID进行散列数据库和表，最终可以在配置文件中进行简单的配置便能让系统 随时增加一台低成本的数据库进来补充系统性能。

4. 缓存

   缓存一词搞技术的都接触过，很多地方用到缓存。网站架构和网站开发中的缓存也是非常重要。这里先讲述最基本的两种缓存。高级和分布式的缓存在后面讲述。 架构方面的缓存，对Apache比较熟悉的人都能知道Apache提供了自己的缓存模块，也可以使用外加的Squid模块进行缓存，这两种方式均可以有效的提高Apache的访问响应能力。
   网 站程序开发方面的缓存，Linux上提供的Memory Cache是常用的缓存接口，可以在web开发中使用，比如用Java开发的时候就可以调用MemoryCache对一些数据进行缓存和通讯共享，一些大型社区使用了这样的架构。另外，在使用web语言开发的时候，各种语言基本都有自己的缓存模块和方法，PHP有Pear的Cache模块，Java就更多 了，.net不是很熟悉，相信也肯定有。

5. 镜像

   镜像是大型网站常采用的提高性能和数据安全性的方式，镜像的技术可以解决不同网络接入商和地域带来的用户访问速度差异，比如ChinaNet和EduNet之间的差异就促使了很多网站在教育网内搭建镜像站点，数据进行定时更新或者实 时更新。在镜像的细节技术方面，这里不阐述太深，有很多专业的现成的解决架构和产品可选。也有廉价的通过软件实现的思路，比如Linux上的rsync等 工具。

6. 负载均衡

   负载均衡将是大型网站解决高负荷访问和大量并发请求采用的终极解决办法。
   负载均衡技术发展了多年，有很多专业的服务提供商和产品可以选择，我个人接触过一些解决方法，其中有两个架构可以给大家做参考。

秒杀的时候，只有最后一件物品，该怎么去抢或者分配?

秒杀商品的库存都会放到redis中，在客户下单时就减库存，减完库存会判断库存是否为 大于 0，如果小于0,表示库存不足，刚才减去的数量再恢复，整个过程使用redis的 watch锁。

你项目对于订单是怎么处理的，假如一个客户在下订单的时候没有 购买怎么办，对于顾客在购买商品的时候你们怎么处理你们的库存?

订单表中设置了一个过期时间，每天会有定时任务来扫描订单表数据，如果到达预订的过期时间没有付款就会取消此订单交易。

关于库存的设计是这样的:

普通商品在发货时才去更新库存，如果库存不足商家会马上补货 秒杀的商品会在客户下单时就减库存，如果在规定时间（半个小时）没有付款，会取消 此订单把库存还原

架构

商品表中的数据是哪里来的

商品表的数据是在商家管理后台中由商家录入的。数据分别录入到商品表、商品描述表 和商品项表

当初设计项目时预计的访问量计划是多少

访问量计划是3000至5000

简单介绍一下你的这个项目以及项目中涉及到的技术框架以及使用 场景以及你主要负责项目中的哪一块?

项目介绍时，先整体介绍是什么项目，项目主要是做啥的，为什么会做这个项目（市场 需求）?例如：XXX电商项目，是一个B2B2C综合电商平台。由三个系统组成，包含： 运营商管理后台、商家管理后台、网站前台。运营商平台主要负责基础数据维护、商家 审核、商品审核等。商家管理后台主要负责商家入驻、商品录入/修改、商品上下架等。 网站前台主要负责商品销售。包含：网站首页、商品搜索、商品详情展示、购物车、订 单、支付、用户中心等模块。 再介绍自己在项目中做的功能模块。例如：运营商管理后台的品牌、规格数据录入，已 经商品管理后台商品录入功能。同时，实现了网站前台系统中的商品搜索、购物车等功 能模块。 然后介绍里面使用的技术：例如：dubbo分布式框架、ssm、es、redis、activeMQ、 支付宝支付等等。最好是结合技术讲解项目功能点如何实现。

秒杀系统中如何防止超售？如何避免脚本进行恶意刷单?

防止超售解决方案：将存库从MySQL前移到Redis中，所有的写操作放到内存中，由于 Redis中不存在锁故不会出现互相等待，并且由于Redis的写性能和读性能都远高于 MySQL，这就解决了高并发下的性能问题。然后通过队列等异步手段，将变化的数据异 步写入到DB中。当达到库存阀值的时候就不在消费队列，并关闭购买功能。 避免脚本恶意刷单：采用IP级别的限流，即针对某一个IP，限制单位时间内发起请求数 量。

如果一个用户的token被其他用户劫持了，怎样解决这个安全问题。

1. 在存储的时候把token进行对称加密存储，用时解开。
2. 将请求URL、时间戳、token三者进行合并加盐签名，服务端校验有效性。
3. HTTPS对URL进行加密

项目部署上线后，运营商管理，商品审核等后台流量问题?

先询问流量是指哪方面？流量分为三种，一种是商家流量，另一种是用户流量，第三种 运营商流量。

解决方案：

这三种流量对系统运行造成很大压力，随着项目上线时间增长，压力会越来越大，因此我们要减轻系统访问压力 ，就需要做一系列优化措施。

具体优化如下：

数据层面的优化：

从数据库层面做优化，比如:索引，缓存，集群，读写分离，主从复制，分表，分库。

从数据库设计层面的优化：比如减少表关联，加入冗余字段

从缓存方面优化：比如redis实现数据缓存，减轻数据库压力

从搜索上进行优化：比如查找索引库

项目层面的优化：

采用面向服务的分布式架构：分担服务器压力，提高项目并发量。 比如

dubbox+zookeeper分布式架构

采用分布式文件系统实现海量文件存储：如采用fastdfs实现海量图片存储，提高文件的访问速度。

采用mq使用服务进一步解藕：同步索引库，同步静态资源，短信发送

服务器层面的优化：

集群思想的使用：tomcat,zookeeper,redis,mysql等

Tomcat异步通信的使用，tomcat连接池配置

秒杀和团购业务实现思路

将商品数量查询出存入到redis中，所有用户下单后，减掉redis中的数量

如果并发量很大时，还要考虑高并发问题，所以可以加入mq消息中间件处理抢单问题，再结合redis实现库存减少操作。高并发方面还可以考虑CDN，Nginx负载均衡等 。

讲一下每台服务器的集群数量：

项目中一共15台项目服务，那么为了每一台高可用一主一备，但首页项目高并发设为四台服务器，则一共32台项目服务器，再加redis集群用了3台，为了每一台高可用一主一备一共6台，fastdfs一个trackerServer一个storageServer搭建集群一共6台，solr集群7台服务器，nginx为了高可用一主一备一共2台，mysql数据库集群3台！activemq消息中间件高可用2台；
共计：58台服务器！

你在项目开发中碰到过哪些重大且棘手的问题

场景一：需求不明确困境

在项目开发中，项目采用迭代开发，开发需求不是很明确，对于项目开发初期来说非常困难，进度非常慢，有时开发的出的产品结果往往不能令老板满意，或者是甲方满意，项目还需要不停的迭代，修改。

比如说：

在开发商城项目的时候，客户定位是一个综合性的商务平台，可以实现在线第三方商家对接，实现商品的销售 。

但是并没有明确的需求，因此开发全凭借电商的项目经验来实现里面的相关的业务，后期慢慢迭代。

场景二： ES高亮不能显示的问题

前台使用angularJS加载搜索结果，但是发现高亮不能展示。

问题原因：

angularJS底层使用ajax，异步加载高亮信息返回给页面后，页面没有刷新，就直接显示返回的数据。此时会把所有的数据作为普通的文本数据进行加载。因此就没有高亮的效果。

解决方案：

使用angularJS过滤器过滤文本数据，此时angularJS过滤器把html文本数据解析为浏览器能识别的html标签。高亮就能展示了。

场景三：Nginx静态页面服务跳转到购物车跨域问题

在Nginx中部署了静态页面，添加购物车时必须从静态页面跳转到购物车系统,实现购物车添加操作。

由于在静态页面中使用angularJS实现的跳转，发现跳转到购物车系统完全没有问题，但是并不能跳转回到购物车系统页面。

问题分析：

从静态详情系统跳转到购物车系统，会存在跨域问题，因此不能进行回调函数的数据传递。所以在回调函数中的页面跳转就不能实现。

解决方案：

使用angularJS跨域调用及springmvc跨域配置，解决问题。

场景四：activeMQ存在运行时间长了以后，收不到消息的现象

时间长了就会出现，卡死，新的数据不能从队列接听到。只能重启程序。

解决方案：

不要频繁的建立和关闭连接

JMS使用长连接方式，一个程序，只要和JMS服务器保持一个连接就可以了，不要频繁的建立和关闭连接。频繁的建立和关闭连接，对程序的性能影响还是很大的。这一点和jdbc还是不太一样的

Connection的start()和stop()方法代价很高

JMS的Connection的start()和stop()方法代价很高，不能经常调用。我们试用的时候，写了个jms的connection pool，每次将connection取出pool时调用start()方法，归还时调用stop()方法，然而后来用jprofiler发现，一般的cpu时间都耗在了这两个方法上。

start()后才能收消息

Connection的start()方法调用后，才能收到jms消息。如果不调用这个方法，能发出消息，但是一直收不到消息。不知道其它的jms服务器也是这样。

显式关闭Session

如果忘记了最后关闭Connection或Session对象，都会导致内存泄漏。这个在我测试的时候也发现了。本来以为关闭了Connection，由这个Connection生成的Session也会被自动关闭，结果并非如此，Session并没有关闭，导致内存泄漏。所以一定要显式的关闭Connection和Session。

对Session做对象池

对Session做对象池，而不是Connection。Session也是昂贵的对象，每次使用都新建和关闭，代价也非常高。而且后来我们发现，原来Connection是线程安全的，而Session不是，所以后来改成了对Session做对象池，而只保留一个Connection。

集群

ActiveMQ有强大而灵活的集群功能，但是使用起来还是会有很多陷阱

场景五：activeMQ存在发出消息太大，造成消息接受不成功

多个线程从activeMQ中取消息，随着业务的扩大，该机器占用的网络带宽越来越高。仔细分析发现，mq入队时并没有异常高的网络流量，仅仅在出队时会产生很高的网络流量。

商品的价格变化后，如何同步redis中数以百万计的购物车数据。

购物车只存储商品id,到购物车结算页面将会从新查询购物车数据，因此就不会涉及购物车商品价格同步的问题。

系统中的钱是如何保证安全的。

在当前互联网系统中钱的安全是头等大事，如何保证钱的安全可以从以下2个方面来思考：

1. 钱计算方面
    1. 在系统中必须是浮点数计算类型存储钱的额度，否则计算机在计算时可能会损失精度。
2. 事务处理方面

1. 在当前环境下，高并发访问，多线程，多核心处理下，很容易出现数据一致性问题，此时必须使用事务进行控制，访问交易出现安全性的问题，那么在分布式系统中，存在分布式事务问题，可以有很多解决方案：
    1. 使用 jpa可以解决
    2. 使用 tcc 框架可以解决等等

做交易或是金融系统安全性需要从哪些方面考虑？没有用什么第三方可以框架

ip黑白名单，访问日志明细记录，防止重复提交，访问频率控制，分布式锁，数据前后端校验，自动对账任务处理，互联网金融项目一般情况下，不建议自动重试，最好结合对账系统，人工进行处理，写好人工处理的接口就好。其他就是控制好数据的一致性了，这个最重要，其次还要保证接口的幂等性，不要重复处理订单。这些是最基本的安全控制了。像这类网站用户的输入数据一般都不会太多，一般敏感词过滤，广告之类的可以忽略，如果有的话还要控制这些。安全框架选shiro 了，在系统中分配好角色就好了，控制好用户的资源访问。其他的用springmvc 就够了

订单中的事物是如何保证一致性的。

使用分布式事务来进行控制，保证数据最终结果的一致性。

当商品库存数量不足时，如何保证不会超卖。

当库存数量不足时，必须保证库存不能被减为负数，如果不加以控制，库存被减为小于等于0的数，那么这就叫做超卖。那么如何防止超卖的现象发生呢?

场景一： 如果系统并发要求不是很高

那么此时库存就可以存储在数据库中，数据库中加锁控制库存的超卖现象。

场景二：系统的并发量很大

如果系统并发量很大，那么就不能再使用数据库来进行减库存操作了，因为数据库加锁操作本身是以损失数据库的性能来进行控制数据库数据的一致性的。

但是当并发量很大的时候，将会导致数据库排队，发生阻塞。因此必须使用一个高效的nosql数据库服务器来进行减库存。
此时可以使用redis服务器来存储库存，redis是一个内存版的数据库，查询效率相当的高，可以使用watch来监控减库存的操作，一旦发现库存被减为0，立马停止售卖操作。

你们系统的商品模块和订单模块的数据库是怎么设计的

商品模块设计：

商品模块一共8张表，整个核心就是模板表。采用模板表为核心的设计方法，来构造商品数据。

订单设计：

订单涉及的表有：

1. 收货人地址
2. 订单信息
3. 订单明细

系统中商家活动策划以及上报相关业务流程。

商城系统中有以下活动：

1. 秒杀活动
    1. 后台设置秒杀商品
    2. 设置秒杀开启时间，定时任务，开启秒杀
    3. 秒杀减库存（秒杀时间结束，库存卖完，活动结束）
2. 促销活动
3. 团购活动
4. 今日推荐

以上活动销售记录，统计，使用图形化报表进行统计，可以查看销售情况。

涉及到积分积累和兑换商品等业务是怎么设计的

积分累计有2大块：

积分累计：

根据用户购买的商品的价格不同，没有购买一定价格的商品，获取一定的积分。

积分商城：

积分商城是用户可以使用积分商品换取商品的区域

介绍下电商项目，你觉得那些是亮点?

这个项目是为xxx开发的b2b2c类型综合购物平台，主要以销售xxx，电子产品为主要的 电子商城网站

项目的亮点是：

1. 项目采用面向服务分布式架构（使用dubbo,zookeeper）
    1. 解耦
    2. 提高项目并发能力
    3. 分担服务器压力
2. 项目中使用activeMQ对项目进一步解耦
    1. 提高项目并发能力
    2. 提高任务处理速度
3. 使用微信支付，支付宝支付（自己总结）
4. 使用阿里大鱼发生短信
5. 使用第三方分布式文件系统存储海量文件
6. Nginx部署静态页面实现动静分离

购物车功能做了吗，实现原理说一下?

1. 加入购物车

    加入购物车插入到库中一条购物记录，同时插入到缓存中，缓存的key是记录的id

    未登录状态

    用户未登录时点击加入购物车，将productId ，skuId，buyNum 转换成json存到cookie中（同一件商品不同的skuId视为两个商品，相同的skuId和productId视为相同商品数量累加），用户登录成功的时候接收用户的消息将cookie中的商品信息保存到数据库中，然后清空cookie数据（京东）不然会出现登录成功后删除购物车商品然后退出，购物车中显示问题

    登录状态

点击加入购物车将long userId,long productId,long skuId,int count 存到库中，相同的productId和skuId 数量累加，不同的skuId新增一条
addToCart(long userId,long skuId,int count); //加入sku到购物车商品

2. 修改商品数量

   未登录状态

   用户未登录时，点击加减数量，根据productId和skuId从cooike中将商品数量进行加减，注意校验cooike中的数量不能小于0，不能大于库存数量

   登录状态

   用户登录状态时，点击加减数量productId和skuId，userId将用户购物车中某个sku的数量增加或减去differ值，注意校验库存数量
   updateAmount(long userId,skuId,int differ,List selectedSkuIds); //将用户购物车中某个sku的数量增加或减去differ值。此方法更新商品后，会根据selectedSkuIds重新计算一遍购物车价格，返回满足条件的优惠券

3. 删除购物车记录

   1. 未登录状态

      用户未登录时，根据productId和skuId删除cookie中的记录
      deleteCart(long userId,long skuId, List selectedSkuIds); //将某个sku从用户购物车移
      除。此接品，在清除后台会重复计算selectedSkuIds价格，并会返回选中的sku列表与未
      选中的sku列表集合。及相应优惠券

   2. 登录状态

      登录状态下，直接根据productId和skuId以及userId删除库中数据

2. 购物车列表展示

   1. 未登录状态

      1. 从cookie中取出productId以及skuId列表展示商品信息

   2. 登录状态

      1. 登录状态下根据用户id查询库中的记录数
         getCart(long userId,list selectedSkuIds); //查询用户购物车。此接口会重新计算selectedSkuIds，并返回选中与未选中sku列表集合，返回相应的满足条件的优惠券信息。

3. 订单提交成功后更新购物车数量以及修改购物车状态
4. 订单提交成功后接收订单成功消息，更新购物车状态和数量删除缓存记录
5. 商品下架后，更新库存状态，显示失效
6. 商品下架后接收消息修改购物车里的商品状态为失效

购物车流程

秒杀商品流程：

1. 商家提交秒杀商品申请，录入秒杀商品数据，主要包括：商品标题、原价、秒杀价、商品图片、介绍等信息
2. 运营商审核秒杀申请
3. 秒杀频道首页列出秒杀商品（进行中的）点击秒杀商品图片跳转到秒杀商品详细页。将秒杀的商品放入缓存减少数据库瞬间的访问压力！

4. 商品详细页显示秒杀商品信息，点击立即抢购实现秒杀下单，下单时扣减库存。当库存为0或不在活动期范围内时无法秒杀。读取商品详细信息时运用缓存，当用户点击抢购时减少redis中的库存数量，当库存数为0时或活动期结束时，同步到数据库。
5. 秒杀下单成功，直接跳转到支付页面（微信扫码），支付成功，跳转到成功页，填写收货地址、电话、收件人等信息，完成订单。
6. 当用户秒杀下单5分钟内未支付，取消预订单，调用微信支付的关闭订单接口，恢复库存。产生的秒杀预订单也不会立刻写到数据库中，而是先写到缓存，当用户付款成功后再写入数据库。

介绍一下自己的项目？

我最近的一个项目是一个电商项目，我主要负责的是后台管理和商品详情的模块，然后也会参与到购物车和订单模块。这个项目是以SpringBoot和mybatis为框架，应为springBoot相对于SSM来说 配置方面，还有操作方面简单很多。然后是采用zookeeper加dubbo分布式架构和RPC远程调用，因为他Dubbo实现了软负载均衡，其特点是成本低，但也会有缺点，就是负载能力会受服务器本身影响，然后为了解决软负载均衡的缺点，我们使用了Nginx进行负载均衡的轮询算法，但Nginx主要在我们项目还是实现反向代理，就是可以防止外网对内网服务器的恶性攻击、缓存以减少服务器的压力和访问安全控制。基础模块就有后台管理，商品详情，订单，支付，物流情况，库存服务。然后SpringBoot整合Thymeleaf模块技术开发项目商品详情模块，easyUI开发后台管理项目。至于我负责的两个模块呢，就是后台管理和商品详情，其中呢使用了sku和spu的数据表结构进行增删改查，spu就好比我们要买一台Mate20，但是我们没有选择它是什么配置，那么关于详细的配置就是sku了，就是我要买一台Mate20，黑色，内存是128G的。商品详情和商品列表模块使用Nginx实现集群，使用Redis解决应用服务器的cpu和内存压力，减少io的读操作，减轻io的压力，使用分布式锁防止Redis缓存击穿。其中Redis的作用我是觉得挺大的，因为他可以防止过多的用户去直接访问我们的数据库，当然，Redis也会在高并发的时候宕机，在使用Redis做缓存的时候，我们使用Redis持久化功能，防止Redis宕机后数据丢失，如果Redis宕机了，用户就会大量的去访问数据库，从而我们数据库也会崩溃吧。这个时候我们就用了一个分布式锁，用户需要获得一个锁才能访问我们的数据库，当然啦，并不只是只有一个锁，而是锁的数量是有限的，当一位用户查完了数据之后，锁就会释放，给下位用户，这也就是服务降降级。没有获得锁的用户，页面就一直刷新直到自己拿到锁为止。redis提供了持久化功能——RDB和AOF。通俗的讲就是将内存中的数据写入硬盘中。在实际应用中，用户如果要查询商品的话呢，首先回到Redis缓存里面找的，如果找不到，就会到数据库里面找，然后缓存到Redis中，那么下一次或者下一个用户需要查找这个数据就不必到数据库中查找了！然后我还参与了购物车和订单模块的开发。购物车模块里面呢，我先和您讲下他的业务逻辑吧。就像你逛网页淘宝一样，在没有登录的时候，把东西放入购物车，它是不会和你的账号里的商品合并的，这个时候，商品就会以cookie的形式，放到你的浏览器里面。这个时候如果你想购买这些商品的时候，你就要登录，这个时候就会使用到单点登录这一个技术。用户跳转到订单页面的时候，我们会用拦截器去进行判断用户是否已经登录。我们是用cookie中是否有token，如果没有token的话就跳转到登录页面，然后生成token，至于token的生成呢，我们是用本地的IP，用户的id，保存在map中，还有一个常量，这个我们通常会以项目名称来命名的。至于为什么要token呢，其实是因为cookie是不太安全的，它很容易被伪造，所以我们就需要token，然后有了token之后，我们用JWT这个盐值生成最后的token。并把它保存到cookie当中。下一次支付的时候我们也还会用到这个token，用一个加密算法再去运算验证一下就可以了！然后就是合并购物车了。这个的话我所知道的就是将客户端的cookie复印一份到缓存中进行修改然后送回客户端进行覆盖，再接着就是数据库的修改了。那这个如果登陆了的就直接从数据库中取得数据跳到订单系统了。然后订单模块里面，简单来说就是从购物车中勾选的商品迁移到订单里面。但是呢订单模块其实是会联系到另外两个模块的，就是库存和支付。如果你点击了提交订单，商品就会在购物车里移除。然后我们提交订单避免他反复的提交同一个订单，就会通过交易码防止订单重复提交。我们会吧tradecode放在缓存里面，以用户id为key商品的交易为value在Redis里面保存这个交易

码。到最后选好收货地址，留言之后，提交订单了，就会用自己的tradecode和在Redis里面通过用户的id去获取tradecode进行对比，如果能跳转到支付页面，那么缓存中的交易码就会删除掉。到最后就是支付功能，这一步的话我是不太清楚其中的技术点了，只知道这个模块调用了支付宝的接口和用了消息队列，异步通知

你们的项目上线了吗？这么大的项目怎么没上线？

项目上线问题回答：

1. 项目没有上线
    1. 如果你没有做过电商的项目，可以说项目没有上线之前，你离职了，这个一个创业型的公司，或者此项目是给甲方做的项目，你没有参与上线。以此来回避这个问题.
    2. 项目上线

        项目已经上线了

        上线环境： Centos7 Mysql Jdk8 Tomcat8

        关于上线，那么面试官一定会问您，上线遇到什么问题没有？

        因此必须把项目中遇到的问题准备2个，以下可以作为参考

        问题一：（用户非正常流程导致的错误）

        用户注册一半就退出来，导致再次注册不成功或者用证件号登陆触发空指针异常。
        解决办法：一旦输入证件号时，检查数据库的表是否有相应的证件号记录，有则把相关记录全部删掉，从而让他成功注册。空指针异常的解决办法，做非空验证的判断。

        问题二：（并发插入，流水号不一致）

        出现大量的主键唯一约束错误，后来想到是产生的预报名号不同步，导致有可能大并发量时产生多个相同的流水号，插入就会出现主键唯一约束错误。

        解决办法：在数据库里写一个insert的触发器。自动替换掉要插入的主键为max（key）+1.

        问题三：（并发删除，索引失效）

        出现某些表的索引失效，后来发现是插入相同主键多次之后导致表失效。

        解决办法:设定oracle任务，让数据库每隔12个小时自动重建所有索引。

订单怎么实现的，你们这个功能怎么这么简单？

订单实现：

从购物车系统跳转到订单页面，选择默认收货地址

选择支付方式

购物清单展示

提交订单

订单业务处理：

一个商家一个订单，不同的仓库发送的货品也是属于不同的订单。因此会产出不同的订单号

订单处理：根据支付的状态进行不同的处理

1. 在线支付
2. 支付未成功—从新发起支付
3. 支付超时---订单关闭
4. 货到付款

你们这个项目有秒杀吗，怎么实现的?

所谓“秒杀”，就是网络卖家发布一些超低价格的商品，
所有买家在同一时间网上抢购的一种销售方式。通俗一点讲就是网络商家为促销等目的组织的网上限时抢购活动。由于商品价格低廉，往往一上架就被抢购一空，有时只用一秒钟。

秒杀商品通常有两种限制：库存限制、时间限制。

需求：

1. 商家提交秒杀商品申请，录入秒杀商品数据，主要包括：商品标题、原价、秒杀价、商品图片、介绍等信息
2. 运营商审核秒杀申请
3. 秒杀频道首页列出秒杀商品（进行中的）点击秒杀商品图片跳转到秒杀商品详细页。
4. 商品详细页显示秒杀商品信息，点击立即抢购实现秒杀下单，下单时扣减库存。当库存为0或不在活动期范围内时无法秒杀
5. 秒杀下单成功，直接跳转到支付页面（微信扫码），支付成功，跳转到成功页，填写收货地址、电话、收件人等信息，完成订单。
6. 当用户秒杀下单5分钟内未支付，取消预订单，调用微信支付的关闭订单接口，恢复库存。

数据库表分析

Tb_seckill_goods 秒杀商品表

Tb_seckill_order 秒杀订单表

秒杀实现思路

秒杀技术实现核心思想是运用缓存减少数据库瞬间的访问压力！读取商品详细信息时运用缓存，当用户点击抢购时减少redis中的库存数量，当库存数为0时或活动期结束时，同步到数据库。 产生的秒杀预订单也不会立刻写到数据库中，而是先写到缓存，当用户付款成功后再写入数据库。

你们这个项目用的什么数据库，数据库有多少张表?

项目使用mysql数据库，总共有103张表，其中商品表共计有8张

单点登录怎么做的，用别人知道原理吗?

在分布式项目中实现session共享，完成分布式系统单点登录

1. Cookie中共享ticket
2. Redis存储session

分布式系统共享用户身份信息session，必须先获取ticket票据，然后再根据票据信息获取redis中用户身份信息。

实现以上2点即可实现session共享

目前项目中使用的springsecurity + cas 来实现的单点登录，cas自动产生ticket票据信息，每次获取用户信息，cas将会携带ticket信息获取用户身份信息

支付做了吗，支付宝还是微信，实现说下?

微信支付：

1. 调用微信支付下单接口
2. 返回支付地址，生成二维码
3. 扫描二维码即可完成支付

问题： 微信支付二维码是我们自己生成的，因此必须时刻监控微信支付二维码的状态，

确保支付成功。

支付宝支付可以参考www.alipay.com

缓存及优化方面的面试问题

怎么提高redis缓存利用率?

1. 怎么提高redis缓存利用率?
2. 可以定时扫描命中率低的数据，可以直接从redis中清除。

怎么实现数据量大、 并发量高的搜索

创建solr索引库，数据量特别大时采用solr分布式集群

MySQL索引使用限制

不要在列上进行运算。

select * from users where YEAR(adddate)<2007; 将在每个行上进行运算，这将导致索引失效而进行全表扫描，因此我们可以改成select * from users where adddate<‘2007-01-01’;

like语句操作

如果使用like。like “%a%” 不会使用索引而like “aaa%”可以使用索引。
select * from users where name like '%a%'不会使用索引

select * from users where name like 'aaa%'可以使用索引

使用短索引

例如，如果有一个CHAR(255)的列，如果在前10个或20个字符内，多数值是惟一的，那么就不要对整个列进行索引。短索引不仅可以提高查询速度而且可以节省磁盘空间和I/O操作

索引不会包含NULL列

复合索引中如果有一列含有NULL值那么这个组合索引都将失效,一般需要给默认值0或者 ' '字符串

最左匹配

不按索引最左列开始查询（多列索引） 例如index(‘c1’, ‘c2’, ‘c3’)，where ‘c2’ = ‘aaa’ 不使用索引,where ‘c2’ = ‘aaa’ and ‘c3’ = ‘sss’ 不能使用索引。where ‘c1’ = ‘aaa’ and ‘c2’ = ‘bbb’ 可以使用索引

多列范围查询

查询中某个列有范围查询，则其右边的所有列都无法使用查询（多列查询）。where c1= ‘xxx’ and c2 like = ‘aa%’ and c3=’sss’ 该查询只会使用索引中的前两列,c3将不能使用到索引,因为like是范围查询。

检索排序

一个查询语句中，既有检索又有排序并且是不同的字段，且这两个列上都有单列索引（独立索引），那么只有其中一个列用到索引，因为查询优化器在做检索和排序中不能同时使用两个不同的索引

索引散列度

通过索引扫描的记录超过了表总行数的30%（估计值），则查询优化器认为全表扫描的效率更高，所以会变成全表扫描查询

隐式转换

隐式转换导致的索引失效。比如，表的字段tu_mdn定义为varchar(20),但在查询时把该字段作为number类型当做where条件,这样会导致索引失效. 错误的例子：select * from test where tu_mdn=13333333333; 正确的例子：select * from test where tu_mdn='13333333333’;

怎么分词

使用第三方的分词器IKAnalyzer，会按照中国人用此习惯自动分词。

seo怎么优化

使用restful，或静态页这样能更好的被搜索引擎收录。

怎么加快访问速度

硬件上加大网络带宽、和服务器内存

代码的处理：静态页面、缓存、优化sql、创建索引等方案

java中的多线程在你们的这个项目当中有哪些体现?

1. 后台任务：如定时向大量(100W以上)的用户发送邮件；定期更新配置文件、任务调度(如quartz)，一些监控用于定期信息采集
2. 自动作业处理：比如定期备份日志、定期备份数据库
3. 异步处理：如发微博、记录日志

哪些情况用到activeMq?

商品上架后更新ES索引库、更新静态页、发送短信 ,提交订单后清除购物车中的数据 ,支付未完成时支付完成后修改订单状态

秒杀的时候，只有最后一件物品，该怎么去抢或者分配?

秒杀商品的库存都会放到redis缓存中，在客户下单时就减库存，我们设置库存库存闸值，用于某些商品数量非单件不可分割，减完库存会判断库存是否为大于库存闸值，如果小于,表示库存不足，刚才减去的数量再恢复，整个过程使用redis的watch锁 。

你项目对于订单是怎么处理的，假如一个客户在下订单的时候没有购买怎么办?

订单表中设置了一个过期时间，每天会有定时任务来扫描订单表数据，如果到达预订的过期时间没有付款就会取消此订单交易。

对于顾客在购买商品的时候你们怎么处理你们的库存?

普通商品只有在发货时才去更新库存，如果库存不足商家会马上补货秒杀的商品会在客户下单时就减库存，如果在规定时间（半个小时）没有付款，会取消此订单把库存还原。

秒杀系统中如何防止超售？如何避免脚本进行恶意刷单?

防止超售解决方案：将存库从MySQL前移到Redis中，所有的写操作放到内存中，由于Redis中不存在锁故不会出现互相等待，并且由于Redis的写性能和读性能都远高于MySQL，这就解决了高并发下的性能问题。然后通过队列等异步手段，将变化的数据异步写入到DB中。当达到库存阀值的时候就不在消费队列，并关闭购买功能。避免脚本恶意刷单：采用IP级别的限流，即针对某一个IP，限制单位时间内发起请求数量。

单点登录你们是自己编写的还是使用通用的CAS?

项目使用通用的CAS框架

什么是CAS?

中央认证服务,企业级单点登录解决方案。CAS Server 需要独立部署，主要负责对用户的认证工作；CAS Client 负责处理对客户端受保护资源的访问请求，需要登录时，重定向到 CAS Server。

如果一个用户的token被其他用户劫持了，怎样解决这个安全问题。

1. 在存储的时候把token进行对称加密存储，用时解开。
2. 将请求URL、时间戳、token三者进行合并加盐签名，服务端校验有效性。
3. HTTPS对URL进行加密

对系统运行造成很大压力，随着项目上线时间增长，压力会越来越大，我们怎么减轻系统访问压力

流量分为三种，一种是商家流量，另一种是用户流量，第三种运营商流量。

解决方案：

这三种流量对系统运行造成很大压力，随着项目上线时间增长，压力会越来越大，因此我们要减轻系统访问压力 ，就需要做一系列优化措施。
具体优化如下：

数据层面的优化：

从数据库层面做优化，比如:索引，缓存集群双缓存，把查询独立出来读写分离，配置数据库集群主从复制，使用Mycat分表，分库。从数据库设计层面的优化：比如减少表关联，加入冗余字段从缓存方面优化：比如redis实现数据缓存，减轻数据库压力从搜索上进行优化：比如查找索引库，使用es或solr全文搜索.

项目层面的优化：

采用面向服务的分布式架构：分担服务器压力，提高项目并发量。比如dubbox+zookeeper的SOA分布式架构采用分布式文件系统实现海量文件存储：如采用fastdfs实现海量图片存储，提高文件的访问速度。采用mq使用服务进一步解藕：同步索引库，同步静态资源，短信发送

服务器层面的优化

集群思想的使用：tomcat,zookeeper,redis,mysql等Tomcat异步通信的使用，tomcat连接池配置

秒杀和团购业务实现思路

将商品数量查询出存入到redis中，所有用户下单后，减掉redis中的数量如果并发量很大时，还要考虑高并发问题，所以可以加入mq消息中间件处理抢单问题，再结合redis实现库存减少操作。高并发方面还可以考虑CDN，Nginx负载均衡等

秒杀活动

秒杀架构设计理念

限流： 鉴于只有少部分用户能够秒杀成功，所以要限制大部分流量，只允许少部分流量进入服务后端

削峰：对于秒杀系统瞬时会有大量用户涌入，所以在抢购一开始会有很高的瞬间峰值。高峰值流量是压垮系统很重要的原因，所以如何把瞬间的高流量变成一段时间平稳的流量也是设计秒杀系统很重要的思路。实现削峰的常用的方法有利用缓存和消息中间件等技术。

异步处理：秒杀系统是一个高并发系统，采用异步处理模式可以极大地提高系统并发量，其实异步处理就是削峰的一种实现方式

内存缓存：秒杀系统最大的瓶颈一般都是数据库读写，由于数据库读写属于磁盘IO，性能很低，如果能够把部分数据或业务逻辑转移到内存缓存，效率会有极大地提升。

可拓展：当然如果我们想支持更多用户，更大的并发，最好就将系统设计成弹性可拓展的，如果流量来了，拓展机器就好了。像淘宝、京东等双十一活动时会增加大量机器应对交易高峰。

前端方案

浏览器端(js)：

页面静态化：将活动页面上的所有可以静态的元素全部静态化，并尽量减少动态元素。通过CDN来抗峰值。

禁止重复提交：用户提交之后按钮置灰，禁止重复提交

用户限流：在某一时间段内只允许用户提交一次请求，比如可以采取IP限流

后端方案

服务端控制器层(网关层)

限制uid（UserID）访问频率：我们上面拦截了浏览器访问的请求，但针对某些恶意攻击或其它插件，在服务端控制层需要针对同一个访问uid，限制访问频率。

服务层

上面只拦截了一部分访问请求，当秒杀的用户量很大时，即使每个用户只有一个请求，到服务层的请求数量还是很大。比如我们有100W用户同时抢100台手机，服务层并发请求压力至少为100W。

采用消息队列缓存请求：既然服务层知道库存只有100台手机，那完全没有必要把100W个请求都传递到数据库啊，那么可以先把这些请求都写到消息队列缓存一下，数据库层订阅消息减库存，减库存成功的请求返回秒杀成功，失败的返回秒杀结束。利用缓存应对读请求：对类似于12306等购票业务，是典型的读多写少业务，大部分请求是查询请求，所以可以利用缓存分担数据库压力。

利用缓存应对写请求：缓存也是可以应对写请求的，比如我们就可以把数据库中的库存数据转移到Redis缓存中，所有减库存操作都在Redis中进行，然后再通过后台进程把Redis中的用户秒杀请求同步到数据库中。

数据库层

数据库层是最脆弱的一层，一般在应用设计时在上游就需要把请求拦截掉，数据库层只承担"能力范围内"的访问请求。所以，上面通过在服务层引入队列和缓存，让最底层的数据库高枕无忧。

案例：利用消息中间件和缓存实现简单的秒杀系统

Redis是一个分布式缓存系统，支持多种数据结构，我们可以利用Redis轻松实现一个强大的秒杀系统。

我们可以采用Redis 最简单的key-value数据结构，用一个原子类型的变量值(AtomicInteger)作为key，把用户id作为value，库存数量便是原子变量的最大值。对于每个用户的秒杀，我们使用 RPUSH key value插入秒杀请求， 当插入的秒杀请求数达到上限时，停止所有后续插入。

然后我们可以在台启动多个工作线程，使用 LPOP key 读取秒杀成功者的用户id，然后再操作数据库做最终的下订单减库存操作。

当然，上面Redis也可以替换成消息中间件如ActiveMQ、RabbitMQ等，也可以将缓存和消息中间件 组合起来，缓存系统负责接收记录用户请求，消息中间件负责将缓存中的请求同步到数据库。

单点登陆如果在另一台电脑上登陆并修改了密码怎么办?

（Single Sign On），简称为 SSO，是目前比较流行的企业业务整合的解决方案之一。SSO的定义是在多个应用系统中，用户只需要登录一次就可以访问所有相互信任的应用系统。

当用户第一次访问应用系统的时候，因为还没有登录，会被引导到认证系统中进行登录；根据用户提供的登录信息，认证系统进行身份校验，如果通过校验，应该返回给用户一个认证的凭据 - - ticket；用户再访问别的应用的时候，就会将这个ticket带上，作为自己认证的凭据，应用系统接受到请求之后会把ticket送到认证系统进行校验，检查ticket的合法性。如果通过校验，用户就可以在不用再次登录的情况下访问应用系统2和应用系统3了。

要实现SSO，需要以下主要的功能：

所有应用系统共享一个身份认证系统。

统一的认证系统是SSO的前提之一。认证系统的主要功能是将用户的登录信息和用户信息库相比较，对用户进行登录认证；认证成功后，认证系统应该生成统一的认证标志（ticket），返还给用户。另外，认证系统还应该对ticket进行效验，判断其有效性。所有应用系统能够识别和提取ticket信息

要实现SSO的功能，让用户只登录一次，就必须让应用系统能够识别已经登录过的用户。应用系统应该能对ticket进行识别和提取，通过与认证系统的通讯，能自动判断当前用户是否登录过，从而完成单点登录的功能。当用户在另一终端登陆并修改密码，则对应的ticket附带的信息会发生改变，导致原有ticket因无法通过校验而失效。因此要求用户使用新的密码重新登陆。在我们的电商项目中，单点登陆使用的验证字符串叫token。这里的ticket是门票的意思，与我们学的token对应相同。

Redis宕机之后，购物车中的数据如何处理？如何缓解mysql压力?

用redis保存的*.rdb文件恢复即可。 另外redis还有AOF功能，启动时可以自动恢复到前一条查询。 这样做在一定程度上减少数据丢失。但重启redis会需要从关系型数据库中读取数据，增大mysql的压力。依据实际情况，如果redis之前有主从复制，则可在其他节点redis上拿到数据。如果公司没钱，则只能暂时限制客户端访问量，优先恢复redis数据。

Zookeeper待机的情况下，dubbo如何工作?

Zookeeper的作用：

zookeeper用来注册服务和进行负载均衡，哪一个服务由哪一个机器来提供必需让调用者知道，简单来说就是ip地址和服务名称的对应关系。当然也可以 通过硬编码的方式把这种对应关系在调用方业务代码中实现，但是如果提供服务的机器挂掉调用者无法知晓，如果不更改代码会继续请求挂掉的机器提供服务。 zookeeper通过心跳机制可以检测挂掉的机器并将挂掉机器的ip和服务对应关系从列表中删除。至于支持高并发，简单来说就是横向扩展，在不更改代码 的情况通过添加机器来提高运算能力。通过添加新的机器向zookeeper注册服务，服务的提供者多了能服务的客户就多了。

dubbo：

是管理中间层的工具，在业务层到数据仓库间有非常多服务的接入和服务提供者需要调度，dubbo提供一个框架解决这个问题。
注意这里的dubbo只是一个框架，至于你架子上放什么是完全取决于你的，就像一个汽车骨架，你需要配你的轮子引擎。这个框架中要完成调度必须要有一个分布式的注册中心，储存所有服务的元数据，你可以用zk，也可以用别的，只是大家都用zk。

zookeeper和dubbo的关系：

Dubbo的将注册中心进行抽象，是得它可以外接不同的存储媒介给注册中心提供服务，有ZooKeeper，Memcached，Redis等。

引入了ZooKeeper作为存储媒介，也就把ZooKeeper的特性引进来。首先是负载均衡，单注册中心的承载能力是有限的，在流量达到一定程度的时 候就需要分流，负载均衡就是为了分流而存在的，一个ZooKeeper群配合相应的Web应用就可以很容易达到负载均衡；资源同步，单单有负载均衡还不 够，节点之间的数据和资源需要同步，ZooKeeper集群就天然具备有这样的功能；命名服务，将树状结构用于维护全局的服务地址列表，服务提供者在启动 的时候，向ZK上的指定节点/dubbo/${serviceName}/providers目录下写入自己的URL地址，这个操作就完成了服务的发布。 其他特性还有Mast选举，分布式锁等。从MQ在完成订单之后，发送消息锁定库存。消息始终失败。

网关是如何实现?

就是定义一个Servlet接收请求。然后经过preFilter(封装请求参数),routeFilter(转发请求)，postFilter(输出内容)。三个过滤器之间，共享request、response以及其他的一些全局变量。

1. 将request,response放入threadlocal中
2. 执行三组过滤器
3. 清除threadlocal中的的环境变量

Redis和mysql数据同步是先删除redis还是先删除mysql?

不管是先写库，再删除缓存；还是先删缓存，再写库，都有可能出现数据不一致的情况因为写和读是并发的，没法保证顺序，如果删了缓存，还没有来得及写库，另一个线程就来读取，发现缓存为空，则去数据库中读取数据写入缓存，此时缓存中为脏数据。如果先写了库，再删除缓存前，写库的线程宕机了，没有删除掉缓存，则也会出现数据不一致情况。 如果是redis集群，或者主从模式，写主读从，由于redis复制存在一定的时间延迟，也有可能导致数据不一致。这时候，考虑先删除数据库内容，再删

redis。因为在库存等实时数据都是直接在数据库中读取，从业务逻辑上来说，我们允许查询时的数据缓存误差，但是不允许结算时的数据存在误差。

Hashmap为什么线程不安全，如何让它线程安全

HashMap在put的时候，插入的元素超过了容量（由负载因子决定）的范围就会触发扩容操作，就是rehash，这个会重新将原数组的内容重新hash到新的扩容数组中，在多线程的环境下，存在同时其他的元素也在进行put操作，如果hash值相同，可能出现同时在同一数组下用链表表示，造成闭环，导致在get时会出现死循环，所以HashMap是线程不安全的。

使用 java.util.Hashtable 类，此类是线程安全的。
使用 java.util.concurrent.ConcurrentHashMap，此类是线程安全的。
使用 java.util.Collections.synchronizedMap() 方法包装 HashMap object，得到线程安全的Map，并在此Map上进行操作。

设计模式在项目中如何体现

1. 模板方法模式

定义一个操作中的算法的骨架，而将一些步骤延迟到子类中，如JdbcTemplate

2. 代理

spring的Proxy模式在aop中有体现

3. 观察者

定义对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新。spring中Observer模式常用的地方是listener的实现。如ApplicationListener。

4. 适配器（Adapter ）

MethodBeforeAdviceAdapter类

5. 策略模式

使用了java的继承和多态

案例1：加减法计算器，定义一个计算类接口，加法和减法类都实现它，加的时候传入加法对象。

案例2：导出excel,pdf,word时，分别创建不同的对象简单理解：执行多个事情时，创建多个对象

6. 单例模式

解决一个全局使用的类频繁的创建与销毁

7. 工厂模式

分为三种：简单工厂，工厂方法，抽象工厂

根据“需求”生产“产品”，解耦“需求”“工厂”和“产品”。简单工厂：通过构造时传入的标识来生产产品，不同产品都在同一个工厂中生产，每新增加一种产品，需要改工厂类，来判断，这种判断会随着产品的增加而增加，给扩展和维护带来麻烦

简单工厂项目案例：根据传入的 不同（比如1对应支付流水，2 对应订单流水），生成不同类型的流水号

工厂方法：（使一个类的使用延迟到子类）

其中的工厂类根据传入的A.class类型，反射出实例

产品接口，产品类A，产品类B，工厂类可以生成不同的产品类对象，如果要随着产品的增加而增加，工厂类不变，只需新增一个产品类C即可。

项目案例：邮件服务器，有三种协议，POP3，IMAP,HTTP,把这三种做完产品类，在定义个工厂方法

抽象工厂：一个工厂生产多个产品，它们是一个产品族，不同的产品族的产品派生于不同的抽象产品

MQ丢包如何解决

transaction机制就是说，发送消息前，开启事物(channel.txSelect())，然后发送消息，如果发送过程中出现什么异常，事物就会回滚(channel.txRollback())，如果发送成功则提交事物(channel.txCommit())。然而缺点就是吞吐量下降了。

所有在该信道上面发布的消息都将会被指派一个唯一的ID(从1开始)，一旦消息被投递到所有匹配的队列之后，rabbitMQ就会发送一个Ack给生产者(包含消息的唯一ID)，这就使得生产者知道消息已经正确到达目的队列了.如果rabiitMQ没能处理该消息，则会发送一个Nack消息给你，你可以进行重试操作。

分布式事务在项目中如何体现

一、两阶段提交（2PC）

两阶段提交这种解决方案属于牺牲了一部分可用性来换取的一致性。在实现方面，在.NET 中，可以借助 TransactionScop 提供的 API 来编程实现分布式系统中的两阶段提交，比如WCF中就有实现这部分功能。不过在多服务器之间，需要依赖于DTC来完成事务一致性

优点： 尽量保证了数据的强一致，适合对数据强一致要求很高的关键领域。（其实也不能100%保证强一致）

缺点： 实现复杂，牺牲了可用性，对性能影响较大，不适合高并发高性能场景，如果分布式系统跨接口调用，目前 .NET 界还没有实现方案

二、补偿事务（TCC）

TCC 其实就是采用的补偿机制，其核心思想是：针对每个操作，都要注册一个与其对应的确认和补偿（撤销）操作。它分为三个阶段：
Try 阶段主要是对业务系统做检测及资源预留Confirm 阶段主要是对业务系统做确认提交，Try阶段执行成功并开始执行 Confirm阶
段时，默认 Confirm阶段是不会出错的。即：只要Try成功，Confirm一定成功。Cancel 阶段主要是在业务执行错误，需要回滚的状态下执行的业务取消，预留资源释放。

优点： 跟2PC比起来，实现以及流程相对简单了一些，但数据的一致性比2PC也要差一些

缺点： 缺点还是比较明显的，在2,3步中都有可能失败。TCC属于应用层的一种补偿方式，所以需要程序员在实现的时候多写很多补偿的代码，在一些场景中，一些业务流程可能用TCC不太好定义及处理

三、本地消息表（异步确保）（使用最多的技术方案）

消息生产方，需要额外建一个消息表，并记录消息发送状态。消息表和业务数据要在一个事务里提交，也就是说他们要在一个数据库里面。然后消息会经过MQ发送到消息的消费方。如果消息发送失败，会进行重试发送。

消息消费方，需要处理这个消息，并完成自己的业务逻辑。此时如果本地事务处理成功，表明已经处理成功了，如果处理失败，那么就会重试执行。如果是业务上面的失败，可以给生产方发送一个业务补偿消息，通知生产方进行回滚等操作。生产方和消费方定时扫描本地消息表，把还没处理完成的消息或者失败的消息再发送一遍。

四、MQ 事务消息

有一些第三方的MQ是支持事务消息的，比如RocketMQ，他们支持事务消息的方式也是类似于采用的二阶段提交，但是市面上一些主流的MQ都是不支持事务消息的，比如RabbitMQ 和 Kafka 都不支持。

以阿里的 RocketMQ 中间件为例，其思路大致为：

第一阶段Prepared消息，会拿到消息的地址.

第二阶段执行本地事务，第三阶段通过第一阶段拿到的地址去访问消息，并修改状态。也就是说在业务方法内要想消息队列提交两次请求，一次发送消息和一次确认消息。如果确认消息发送失败了RocketMQ会定期扫描消息集群中的事务消息，这时候发现了Prepared消息，它会向消息发送者确认，所以生产方需要实现一个check接口，RocketMQ会根据发送端设置的策略来决定是回滚还是继续发送确认消息。这样就保证了消息发送与本地事务同时成功或同时失败。

Spring管理bean的作用域，为什么不会被GC处理?

当通过spring容器创建一个Bean实例时，不仅可以完成Bean实例的实例化，还可以为Bean指定特定的作用域。Spring支持如下5种作用域：

singleton：单例模式，在整个Spring IoC容器中，使用singleton定义的Bean将只有一个实例

prototype：原型模式，每次通过容器的getBean方法获取prototype定义的Bean时，都将产生一个新的Bean实例

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

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

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

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

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

spring底层使用map来存放bean实体，而map的键值是强引用，所以不会被GC，可以重复使用

上传图片的过程中图片是在前端压缩还是后端压缩

前端，减少服务器压力，从最开始就降低了传输的数据量。

Redis和MySQL如何对接

应用Redis实现数据的读写，同时利用队列处理器定时将数据写入mysql，此种情况存在的问题主要是如何保证mysql与redis的数据同步，二者数据同步的关键在于mysql数据库中主键，方案是在redis启动时去mysql读取所有表键值存入redis中，往redis写数据时，对redis主键自增并进行读取，若mysql更新失败，则需要及时清除缓存及同步redis主键。

Spring aop注解

设置事务隔离级别

多线程问题（原理）如何查看Stop之后的线程

Spring对bean是如何解析

所谓bean的解析就是将我们的xml文件中的bean解析出来，上面的入口看到使用的是ClassPathXmlApplicationContext来获取ApplicationContext，所以，分析的入口也就从ClassPathXmlApplicationContext类中相应的构造函数开始。

getBean() 方法开始创建过程，getBean()有一系列的重载方法，最终都是调用doGetBean() 方法

getSingleton 方法尝试从缓存中获取单例 bean当前 bean 是单例且缓存不存在则通过getSingleton(String beanName,ObjectFactory<?> singletonFactory) 方法创建单例对象

主要包含下下面三个主要方法：

createBeanInstance

populateBean

initializeBean

createBeanInstance 方法用于创建 Bean 实例

populateBean 方法主要给 Bean 进行属性注入

initializeBean 方法主要处理各种回调

为什么InnoDB支持事务而myisam不支持

MyISAM:这个是默认类型,它是基于传统的ISAM类型,ISAM是Indexed SequentialAccess Method (有索引的顺序访问方法) 的缩写,它是存储记录和文件的标准方法.与其他存储引擎比较,MyISAM具有检查和修复表格的大多数工具. MyISAM表格可以被压缩,而且它们支持全文搜索.它们不是事务安全的,而且也不支持外键。如果事物回滚将造成不完全回滚，不具有原子性。如果执行大量的SELECT，MyISAM是更好的选择。InnoDB:这种类型是事务安全的.它与BDB类型具有相同的特性,它们还支持外键.InnoDB表格速度很快.具有比BDB还丰富的特性,因此如果需要一个事务安全的存储引擎,建议使用它.如果你的数据执行大量的INSERT或UPDATE,出于性能方面的考虑，应该使用InnoDB表

幂等性防止订单重复提交

1. token机制，防止页面重复提交
2. 唯一索引，防止新增脏数据
3. 悲观锁乐观锁机制
4. 分布式锁

Mysql除了可以做读写分离集群外还可以做什么集群

Mysql有主从复制集群和读写分离集群

项目中哪些体现了动态代理

Aop面向切面使用动态代理，有jdk和cglig

CurrentHashmap的并发度 Synchronized和writeeverywhere

乐观锁和悲观锁在代码层面和sql层面如何实现

Sql层面：

一、悲观锁

1、排它锁，当事务在操作数据时把这部分数据进行锁定，直到操作完毕后再解锁，其他事务操作才可操作该部分数据。这将防止其他进程读取或修改表中的数据

2、实现：大多数情况下依靠数据库的锁机制实现

一般使用 select ...for update 对所选择的数据进行加锁处理，例如select * from account where name=”Max” for update， 这条sql 语句锁定了account 表中所有符合检索条件（name=”Max”）的记录。本次事务提交之前（事务提交时会释放事务过程中的锁），外界无法修改这些记录。

二、乐观锁

1、如果有人在你之前更新了，你的更新应当是被拒绝的，可以让用户重新操作。

2、实现：大多数基于数据版本（Version）记录机制实现

具体可通过给表加一个版本号或时间戳字段实现，当读取数据时，将version字段的值一同读出，数据每更新一次，对此version值加一。当我们提交更新的时候，判断当前版本信息与第一次取出来的版本值大小，如果数据库表当前版本号与第一次取出来的version值相等，则予以更新，否则认为是过期数据，拒绝更新，让用户重新操作。

代码层面：

悲观锁:一段执行逻辑加上悲观锁,不同线程同时执行时,只能有一个线程执行,其他的线程在入口处等待,直到锁被释放.

乐观锁:一段执行逻辑加上乐观锁,不同线程同时执行时,可以同时进入执行,在最后更新数据的时候要检查这些数据是否被其他线程修改了(版本和执行初是否相同),没有修改则进行更新,否则放弃本次操作。

Jdk1.7和1.8之后的锁有什么不同

MySQL存储过程

SQL语句需要先编译然后执行，而存储过程（Stored Procedure）是一组为了完成特定功能的SQL语句集，经编译后存储在数据库中，用户通过指定存储过程的名字并给定参数（如果该存储过程带有参数）来调用执行它。存储过程是可编程的函数，在数据库中创建并保存，可以由SQL语句和控制结构组成。当想要在不同的应用程序或平台上执行相同的函数，或者封装特定功能时，存储过程是非常有用的。数据库中的存储过程可以看做是对编程中面向对象方法的模拟，它允许控制数据的访问方式。

存储过程的优点：

(1)增强SQL语言的功能和灵活性：存储过程可以用控制语句编写，有很强的灵活性，可以完成复杂的判断和较复杂的运算。

(2)标准组件式编程：存储过程被创建后，可以在程序中被多次调用，而不必重新编写该存储过程的SQL语句。而且数据库专业人员可以随时对存储过程进行修改，对应用程序源代码毫无影响。

(3)较快的执行速度：如果某一操作包含大量的Transaction-SQL代码或分别被多次执行，那么存储过程要比批处理的执行速度快很多。因为存储过程是预编译的。在首次运行一个存储过程时查询，优化器对其进行分析优化，并且给出最终被存储在系统表中的执行计划。而批处理的Transaction-SQL语句在每次运行时都要进行编译和优化，速度相对要慢一些。

(4)减少网络流量：针对同一个数据库对象的操作（如查询、修改），如果这一操作所涉及的Transaction-SQL语句被组织进存储过程，那么当在客户计算机上调用该存储过程时，网络中传送的只是该调用语句，从而大大减少网络流量并降低了网络负载。
(5).作为一种安全机制来充分利用：通过对执行某一存储过程的权限进行限制，能够实现对相应的数据的访问权限的限制，避免了非授权用户对数据的访问，保证了数据的安全。

MySQL存储过程的创建

语法

CREATE PROCEDURE 过程名([[IN|OUT|INOUT] 参数名 数据类型[,[IN|OUT|INOUT]参数名 数据类型...]]) [特性 ...] 过程体

DELIMITER //

```
CREATE PROCEDURE myproc(OUT s int)
BEGIN
SELECT COUNT(*) INTO s FROM students;
END
//
DELIMITER ;
```

分隔符

MySQL默认以";"为分隔符，如果没有声明分割符，则编译器会把存储过程当成SQL语句进行处理，因此编译过程会报错，所以要事先用“DELIMITER //”声明当前段分隔符，让编译器把两个"//"之间的内容当做存储过程的代码，不会执行这些代码；“DELIMITER;”的意为把分隔符还原。

参数

存储过程根据需要可能会有输入、输出、输入输出参数，如果有多个参数用","分割开。MySQL存储过程的参数用在存储过程的定义，共有三种参数类型,IN,OUT,INOUT: IN参数的值必须在调用存储过程时指定，在存储过程中修改该参数的值不能被返回，为默认值

- OUT:该值可在存储过程内部被改变，并可返回
- INOUT:调用时指定，并且可被改变和返回

过程体

过程体的开始与结束使用BEGIN与END进行标识。

Spring boot和spring cloud的区别与联系

Spring boot 是 Spring 的一套快速配置脚手架，可以基于spring boot 快速开发单个微服务，Spring Boot，看名字就知道是Spring的引导，就是用于启动Spring的，使得Spring的学习和使用变得快速无痛。不仅适合替换原有的工程结构，更适合微服务开发。

Spring Cloud基于Spring Boot，为微服务体系开发中的架构问题，提供了一整套的解决方案——服务注册与发现，服务消费，服务保护与熔断，网关，分布式调用追踪，分布式配置管理等。

Spring Cloud是一个基于Spring Boot实现的云应用开发工具；Spring boot专注于快速、方便集成的单个个体，Spring Cloud是关注全局的服务治理框架；spring boot使用了默认大于配置的理念，很多集成方案已经帮你选择好了，能不配置就不配置，Spring Cloud很大的一部分是基于Spring boot来实现。

分布式锁（zookeeper，redis，数据库）如何实现

一、基于数据库实现的分布式锁

基于表实现的分布式锁

```
CREATE TABLE `methodLock` (
`id` int(11) NOT NULL AUTO_INCREMENT COMMENT '主键',
`method_name` varchar(64) NOT NULL DEFAULT '' COMMENT '锁定的方法名',
`desc` varchar(1024) NOT NULL DEFAULT '备注信息',
`update_time` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP ON
UPDATE CURRENT_TIMESTAMP COMMENT '保存数据时间，自动生成',
PRIMARY KEY (`id`),
UNIQUE KEY `uidx_method_name` (`method_name `) USING BTREE )
ENGINE=InnoDB DEFAULT CHARSET=utf8 COMMENT='锁定中的方法';
```

当我们想要锁住某个方法时，执行以下SQL：

insert into methodLock(method_name,desc) values (‘method_name’,‘desc’)

因为我们对method_name做了唯一性约束，这里如果有多个请求同时提交到数据库的话，数据库会保证只有一个操作可以成功，那么我们就可以认为操作成功的那个线程获得了该方法的锁，可以执行方法体内容。

当方法执行完毕之后，想要释放锁的话，需要执行以下Sql:
delete from methodLock where method_name ='method_name'

上面这种简单的实现有以下几个问题：

这把锁强依赖数据库的可用性，数据库是一个单点，一旦数据库挂掉，会导致业务系统不可用。

这把锁没有失效时间，一旦解锁操作失败，就会导致锁记录一直在数据库中，其他线程无法再获得到锁。

这把锁只能是非阻塞的，因为数据的insert操作，一旦插入失败就会直接报错。没有获得锁的线程并不会进入排队队列，要想再次获得锁就要再次触发获得锁操作。

这把锁是非重入的，同一个线程在没有释放锁之前无法再次获得该锁。因为数据中数据已经存在了。

这把锁是非公平锁，所有等待锁的线程凭运气去争夺锁。

当然，我们也可以有其他方式解决上面的问题。

数据库是单点？搞两个数据库，数据之前双向同步。一旦挂掉快速切换到备库上。没有失效时间？只要做一个定时任务，每隔一定时间把数据库中的超时数据清理一遍。

非阻塞的？搞一个while循环，直到insert成功再返回成功。非重入的？在数据库表中加个字段，记录当前获得锁的机器的主机信息和线程信息，那么下次再获取锁的时候先查询数据库，如果当前机器的主机信息和线程信息在数据库可以查到的话，直接把锁分配给他就可以了。

非公平的？再建一张中间表，将等待锁的线程全记录下来，并根据创建时间排序，只有最先创建的允许获取锁

基于排他锁实现的分布式锁

除了可以通过增删操作数据表中的记录以外，其实还可以借助数据中自带的锁来实现分布式的锁。

我们还用刚刚创建的那张数据库表。可以通过数据库的排他锁来实现分布式锁。 基于MySql的InnoDB引擎，可以使用以下方法来实现加锁操作：

```
public boolean lock(){
    connection.setAutoCommit(false);
while(true){
        try{
        result = select * from methodLock where method_name=xxx for update;
        if(result==null){
        return true;
        }
}catch(Exception e){
} sleep(1000);} return false;
}
```

在查询语句后面增加for update，数据库会在查询过程中给数据库表增加排他锁。当某条记录被加上排他锁之后，其他线程无法再在该行记录上增加排他锁。我们可以认为获得排它锁的线程即可获得分布式锁，当获取到锁之后，可以执行方法的业务逻辑，执行完方法之后，再通过以下方法解锁：public void unlock(){ connection.commit(); }通过connection.commit();操作来释放锁。

这种方法可以有效的解决上面提到的无法释放锁和阻塞锁的问题。阻塞锁？ for update语句会在执行成功后立即返回，在执行失败时一直处于阻塞状态，直到成功。锁定之后服务宕机，无法释放？使用这种方式，服务宕机之后数据库会自己把锁释放掉。

但是还是无法直接解决数据库单点、可重入和公平锁的问题。总结一下使用数据库来实现分布式锁的方式，这两种方式都是依赖数据库的一张表，一种是通过表中的记录的存在情况确定当前是否有锁存在，另外一种是通过数据库的排他锁来实现分布式锁。数据库实现分布式锁的优点直接借助数据库，容易理解。数据库实现分布式锁的缺点会有各种各样的问题，在解决问题的过程中会使整个方案变得越来越复杂。

操作数据库需要一定的开销，性能问题需要考虑。

二、基于缓存的分布式锁

相比较于基于数据库实现分布式锁的方案来说，基于缓存来实现在性能方面会表现的更好一点。

目前有很多成熟的缓存产品，包括Redis，memcached等。这里以Redis为例来分析下使用缓存实现分布式锁的方案。基于Redis实现分布式锁在网上有很多相关文章，其中主要的实现方式是使用Jedis.setNX方法来实现。

```
public boolean trylock(String key) {
    ResultCode code = jedis.setNX(key, "This is a Lock.");
    if (ResultCode.SUCCESS.equals(code))
    return true;
else
    return false;
}
public boolean unlock(String key){
    ldbTairManager.invalid(NAMESPACE, key);
}
```

以上实现方式同样存在几个问题：

1、单点问题。

2、这把锁没有失效时间，一旦解锁操作失败，就会导致锁记录一直在redis中，其他线程无法再获得到锁。

3、这把锁只能是非阻塞的，无论成功还是失败都直接返回。

4、这把锁是非重入的，一个线程获得锁之后，在释放锁之前，无法再次获得该锁，因为使用到的key在redis中已经存在。无法再执行setNX操作。

5、这把锁是非公平的，所有等待的线程同时去发起setNX操作，运气好的线程能获取锁。

当然，同样有方式可以解决。

现在主流的缓存服务都支持集群部署，通过集群来解决单点问题。没有失效时间？redis的setExpire方法支持传入失效时间，到达时间之后数据会自动删除。

非阻塞？while重复执行。

非可重入？在一个线程获取到锁之后，把当前主机信息和线程信息保存起来，下次再获取之前先检查自己是不是当前锁的拥有者。

非公平？在线程获取锁之前先把所有等待的线程放入一个队列中，然后按先进先出原则获取锁。

redis集群的同步策略是需要时间的，有可能A线程setNX成功后拿到锁，但是这个值还没有更新到B线程执行setNX的这台服务器，那就会产生并发问题。redis的作者Salvatore Sanfilippo，提出了Redlock算法，该算法实现了比单一节点更安全、可靠的分布式锁管理（DLM）。Redlock算法假设有N个redis节点，这些节点互相独立，一般设置为N=5，这N个节点运行在不同的机器上以保持物理层面的独立。算法的步骤如下：

1、客户端获取当前时间，以毫秒为单位。

2、客户端尝试获取N个节点的锁，（每个节点获取锁的方式和前面说的缓存锁一样），N个节点以相同的key和value获取锁。客户端需要设置接口访问超时，接口超时时间需要远远小于锁超时时间，比如锁自动释放的时间是10s，那么接口超时大概设置5-50ms。这样可以在有redis节点宕机后，访问该节点时能尽快超时，而减小锁的正常使用。

3、客户端计算在获得锁的时候花费了多少时间，方法是用当前时间减去在步骤一获取的时间，只有客户端获得了超过3个节点的锁，而且获取锁的时间小于锁的超时时间，客户端才获得了分布式锁。

4、客户端获取的锁的时间为设置的锁超时时间减去步骤三计算出的获取锁花费时间。

5、如果客户端获取锁失败了，客户端会依次删除所有的锁。 使用Redlock算法，可以保证在挂掉最多2个节点的时候，分布式锁服务仍然能工作，这相比之前的数据库锁和缓存锁大大提高了可用性，由于redis的高效性能，分布式缓存锁性能并不比数据库锁差。

但是，有一位分布式的专家写了一篇文章《How to do distributed locking》，质疑Redlock的正确性。

该专家提到，考虑分布式锁的时候需要考虑两个方面：性能和正确性。如果使用高性能的分布式锁，对正确性要求不高的场景下，那么使用缓存锁就足够了。如果使用可靠性高的分布式锁，那么就需要考虑严格的可靠性问题。而Redlock则不符合正确性。为什么不符合呢？专家列举了几个方面。

现在很多编程语言使用的虚拟机都有GC功能，在Full GC的时候，程序会停下来处理GC，有些时候Full GC耗时很长，甚至程序有几分钟的卡顿，文章列举了HBase的例子，HBase有时候GC几分钟，会导致租约超时。而且Full GC什么时候到来，程序无法掌控，程序的任何时候都可能停下来处理GC，比如下图，客户端1获得了锁，正准备处理共享资源的时候，发生了Full GC直到锁过期。这样，客户端2又获得了锁，开始处理共享资源。在客户端2处理的时候，客户端1 Full GC完成，也开始处理共享资源，这样就出现了2个客户端都在处理共享资源的情况。给锁带上token，token就是version的概念，每次操作锁完成，token都会加1，在处理共享资源的时候带上token，只有指定版本的token能够处理共享资源。使用缓存实现分布式锁的优点性能好。

使用缓存实现分布式锁的缺点

实现过于负责，需要考虑的因素太多。

基于Zookeeper实现的分布式锁

基于zookeeper临时有序节点可以实现的分布式锁。

大致思想即为：每个客户端对某个方法加锁时，在zookeeper上的与该方法对应的指定节点的目录下，生成一个唯一的瞬时有序节点。 判断是否获取锁的方式很简单，只需要判断有序节点中序号最小的一个。 当释放锁的时候，只需将这个瞬时节点删除即可。同时，其可以避免服务宕机导致的锁无法释放，而产生的死锁问题。

来看下Zookeeper能不能解决前面提到的问题。

锁无法释放？使用Zookeeper可以有效的解决锁无法释放的问题，因为在创建锁的时候，客户端会在ZK中创建一个临时节点，一旦客户端获取到锁之后突然挂掉（Session连接断开），那么这个临时节点就会自动删除掉。其他客户端就可以再次获得锁。非阻塞锁？使用Zookeeper可以实现阻塞的锁，客户端可以通过在ZK中创建顺序节点，并且在节点上绑定监听器，一旦节点有变化，Zookeeper会通知客户端，客户端可以检查自己创建的节点是不是当前所有节点中序号最小的，如果是，那么自己就获取到锁，便可以执行业务逻辑了。

不可重入？使用Zookeeper也可以有效的解决不可重入的问题，客户端在创建节点的时候，把当前客户端的主机信息和线程信息直接写入到节点中，下次想要获取锁的时候和当前最小的节点中的数据比对一下就可以了。如果和自己的信息一样，那么自己直接获取到锁，如果不一样就再创建一个临时的顺序节点，参与排队。

单点问题？使用Zookeeper可以有效的解决单点问题，ZK是集群部署的，只要集群中有半数以上的机器存活，就可以对外提供服务。

公平问题？使用Zookeeper可以解决公平锁问题，客户端在ZK中创建的临时节点是有序的，每次锁被释放时，ZK可以通知最小节点来获取锁，保证了公平。问题又来了，我们知道Zookeeper需要集群部署，会不会出现Redis集群那样的数据同步问题呢？

Zookeeper是一个保证了弱一致性即最终一致性的分布式组件。

Zookeeper采用称为Quorum Based Protocol的数据同步协议。假如Zookeeper集群有N台Zookeeper服务器(N通常取奇数，3台能够满足数据可靠性同时有很高读写性能，5台在数据可靠性和读写性能方面平衡最好)，那么用户的一个写操作，首先同步到N/2 +1台服务器上，然后返回给用户，提示用户写成功。基于Quorum Based Protocol的数据同步协议决定了Zookeeper能够支持什么强度的一致性。

在分布式环境下，满足强一致性的数据储存基本不存在，它要求在更新一个节点的数据，需要同步更新所有的节点。这种同步策略出现在主从同步复制的数据库中。但是这种同步策略，对写性能的影响太大而很少见于实践。因为Zookeeper是同步写N/2+1个节点，还有N/2个节点没有同步更新，所以Zookeeper不是强一致性的。

用户的数据更新操作，不保证后续的读操作能够读到更新后的值，但是最终会呈现一致性。牺牲一致性，并不是完全不管数据的一致性，否则数据是混乱的，那么系统可用性再高分布式再好也没有了价值。牺牲一致性，只是不再要求关系型数据库中的强一致性，而是只要系统能达到最终一致性即可。

Zookeeper是否满足因果一致性，需要看客户端的编程方式。

不满足因果一致性的做法

A进程向Zookeeper的/z写入一个数据，成功返回

A进程通知B进程，A已经修改了/z的数据

B读取Zookeeper的/z的数据

由于B连接的Zookeeper的服务器有可能还没有得到A写入数据的更新，那么B将读不到A写入的数据

满足因果一致性的做法

B进程监听Zookeeper上/z的数据变化

A进程向Zookeeper的/z写入一个数据，成功返回前，Zookeeper需要调用注册在/z上的监听器，Leader将数据变化的通知告诉B

B进程的事件响应方法得到响应后，去取变化的数据，那么B一定能够得到变化的值这里的因果一致性提现在Leader和B之间的因果一致性，也就是是Leader通知了数据有变化

第二种事件监听机制也是对Zookeeper进行正确编程应该使用的方法，所以，Zookeeper应该是满足因果一致性的

所以我们在基于Zookeeper实现分布式锁的时候，应该使用满足因果一致性的做法，即等待锁的线程都监听Zookeeper上锁的变化，在锁被释放的时候，Zookeeper会将锁变化的通知告诉满足公平锁条件的等待线程。

可以直接使用zookeeper第三方库客户端，这个客户端中封装了一个可重入的锁服务。

Dubbo原理，ES权重如何实现

Spring boot和dubbo的区别

Ik分词器分出中英文的原因是什么

后台订单倒计时如何对接（同步）前端页面

Spring代理是什么

项目上线后，日志如何处理

输出成文本文档保存记录

如何使用集合分组

ThreadLocal作用和使用场景，数据传递是否同步

ThreadLocal提供一个方便的方式，可以根据不同的线程存放一些不同的特征属性，可以方便的在线程中进行存取。

在Hibernate中是通过使用ThreadLocal来实现的。在getSession方法中，如果ThreadLocal存在session，则返回session，否则创建一个session放入ThreadLocal中。

总结一下就是在ThreadLocal中存放了一个session。实际上ThreadLocal中并没有存放任何的对象或引用，在上面的的代码中ThreadLocal的实例threadSession只相当于一个标记的作用。而存放对象的真正位置是正在运行的Thread线程对象，每个Thread对象中都存放着一个ThreadLocalMap类型threadLocals对象，这是一个映射表map，这个map的键是一个ThreadLocal对象，值就是我们想存的局部对象。

在线程中存放一些就像session的这种特征变量，会针对不同的线程，有不同的值。因此，不同步。

线程池满了怎么办

Dubbo失败策略

Failover Cluster 模式

1.失败自动切换，当出现失败，重试其它服务器。(缺省)
\2. 通常用于读操作，但重试会带来更长延迟。
\3. 可通过retries=”2”来设置重试次数(不含第一次)。

Failfast Cluster

快速失败，只发起一次调用，失败立即报错。
通常用于非幂等性的写操作，比如新增记录。
Failsafe Cluster
失败安全，出现异常时，直接忽略。
通常用于写入审计日志等操作。
.Failback Cluster
失败自动恢复，后台记录失败请求，定时重发。
通常用于消息通知操作。
Forking Cluster
并行调用多个服务器，只要一个成功即返回。
通常用于实时性要求较高的读操作，但需要浪费更多服务资源。
可通过forks="2"来设置最大并行数。
Broadcast Cluster
广播调用所有提供者，逐个调用，任意一台报错则报错。(2.1.0开始支持)
通常用于通知所有提供者更新缓存或日志等本地资源信息。

Redis中watch机制和原理

我们常用redis的watch和multi来处理一些涉及并发的操作，redis的watch+multi实际是一种乐观锁

watch命令描述WATCH命令可以监控一个或多个键，一旦其中有一个键被修改（或删除），之后的事务就不会执行。监控一直持续到EXEC命令（事务中的命令是在EXEC之后才执行的，所以在MULTI命令后可以修改WATCH监控的键值）

Mybatis:

1、什么是mybatis?

mybatis是一个优秀的基于java的持久层框架，它内部封装了jdbc，使开发者只需要关注sql语句本身，而不需要花费精力去处理加载驱动、创建连接、创建statement等繁杂的过程。

mybatis通过xml或注解的方式将要执行的各种statement配置起来，并通过java对象和statement中sql的动态参数进行映射生成最终执行的sql语句，最后由mybatis框架执行sql并将结果映射为java对象并返回。

MyBatis 支持定制化 SQL、存储过程以及高级映射。MyBatis 避免了几乎所有的JDBC 代码和手动设置参数以及获取结果集。MyBatis 可以使用简单的 XML 或注解来配置和映射原生信息，将接口和 Java 的 POJO映射成数据库中的记录。

2、Mybait的优点：

简单易学，容易上手（相比于Hibernate） —- 基于SQL编程；

JDBC相比，减少了50%以上的代码量，消除了JDBC大量冗余的代码，不需要手动开关连接；

很好的与各种数据库兼容（因为MyBatis使用JDBC来连接数据库，所以只要JDBC支持的数据库MyBatis都支持，而JDBC提供了可扩展性，所以只要这个数据库有针对Java的jar包就可以就可以与MyBatis兼容），开发人员不需要考虑数据库的差异性。

提供了很多第三方插件（分页插件 / 逆向工程）；

能够与Spring很好的集成；

MyBatis相当灵活，不会对应用程序或者数据库的现有设计强加任何影响，SQL写在XML里，从程序代码中彻底分离，解除sql与程序代码的耦合，便于统一管理和优化，并可重用。

提供XML标签，支持编写动态SQL语句。

提供映射标签，支持对象与数据库的ORM字段关系映射。

提供对象关系映射标签，支持对象关系组建维护。

MyBatis框架的缺点：

SQL语句的编写工作量较大，尤其是字段多、关联表多时，更是如此，对开发人员编写SQL语句的功底有一定要求。

SQL语句依赖于数据库，导致数据库移植性差，不能随意更换数据库。

MyBatis框架适用场合：

MyBatis专注于SQL本身，是一个足够灵活的DAO层解决方案

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

#{}和${}的区别是什么？

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

Mybatis在处理${}时，就是把${}替换成变量的值。

使用#{}可以有效的防止SQL注入，提高系统安全性。

Dubbo:

简单的介绍一下Dubbo？(Dubbo是什么)

dubbo就是个服务调用的东东。

为什么怎么说呢?

因为Dubbo是由阿里开源的一个RPC分布式框架

那么RPC是什么呢?

就是不同的应用部署到不同的服务器上，应用之间想要调用没有办法直接调用，因为不在一个内存空间，需要通过网络通讯来调用，或者传达调用的数据。而且RPC会将远程调用的细节隐藏起来，让调用远程服务像调用本地服务一样简单。

dubbo有哪些组件?

主要有五个角色/核心组件，分为是Container（容器）、Provider（服务的提供方）、Registry（注册中心）、Consumer（服务的消费方）、Monitor（监控中心）。

容器：主要负责启动、加载、运行服务提供者；

注册中心：注册中心只负责地址的注册和查找

监控中心：监控中心负责统计各服务调用次数、调用时间

Dubbo支持什么协议?

Dubbo协议：缺省协议、采用了单一长连接和NIO异步通讯、使用线程池并发处理请求，能减少握手和加大并发效率

Zookeeper:

Zookeeper的实现原理？（工作原理）

Zookeeper会维护一个类似于标准的文件系统的具有层次关系的数据结构。这个文件系统中每个子目录项都被称为znode节点，这个znode节点也可以有子节点，每个节点都可以存储数据，客户端也可以对这些node节点进行getChildren，getData,exists方法，同时也可以在znode tree路径上设置watch（类似于监听），当watch路径上发生节点create、delete、update的时候，会通知到client。client可以得

到通知后，再获取数据，执行业务逻辑操作。Zookeeper 的作用主要是用来维护和监控存储的node节点上这些数据的状态变化，通过监控这些数据状态的变化，从而达到基于数据的集群管理。

为什么要用zookeeper作为dubbo的注册中心？能选择其他的吗?

Zookeeper的数据模型是由一系列的Znode数据节点组成，和文件系统类似。zookeeper的数据全部存储在内存中，性能高；zookeeper也支持集群，实现了高可用；同时基于zookeeper的特性，也支持事件监听（服务的暴露方发生变化，可以进行推送），所以zookeeper适合作为dubbo的注册中心区使用。redis、Simple也可以作为dubbo的注册中心来使用。

项目中主要用zookeeper做了什么?

作为注册中心用；主要是在服务器上搭建zookeeper，其次在spring管理的dubbo的配置文件中配置（暴露方和消费方都需要配置）

高可用

通常企业级应用系统（特别是政府部门和大企业的应用系统）一般会采用安规的软硬件设备，如IOE（IBM的小型机、Oracle数据、EMC存储设备）系列。而一般互联网公司更多地采用PC级服务器（x86），开源的数据库（MySQL）和操作系统（Linux）组建廉价且高容错（硬件故障是常态）的应用集群。

1. 设计的目的?

   保证服务器硬件故障服务依然可用，数据依然保存并能够被访问

2. 主要的手段?

   数据和服务的①冗余备份以及②失效转移：

   对于服务而言，一旦某个服务器宕机，就将服务切换到其他可用的服务器上；

   对于数据而言，如果某个磁盘损坏，就从备份的磁盘（事先就做好了数据的同步复制）读取数据。

高可用的数据

保证数据高可用的主要手段有两种：一是数据备份，二是失效转移机制；

1. 数据备份：又分为冷备份和热备份，冷备份是定期复制，不能保证数据可用性。热备份又分为异步热备和同步热备，异步热备是指多份数据副本的写入操作异步完成，而同步方式则是指多份数据副本的写入操作同时完成。
2. 失效转移：若数据服务器集群中任何一台服务器宕机，那么应用程序针对这台服务器的所有读写操作都要重新路由到其他服务器，保证数据访问不会失败。

网站运行监控

""不允许没有监控的系统上线"

1. 监控数据采集

   用户行为日志收集：服务器端的日志收集和客户端的日志收集；目前许多网站逐步开 发基于实时计算框架Storm的日志统计与分析工具；

2. 服务器性能监控

   收集服务器性能指标，如系统Load、内存占用、磁盘IO等，及时判 断，防患于未然；

3. 运行数据报告：采集并报告，汇总后统一显示，应用程序需要在代码中处理运行数据 采集的逻辑；

（2）监控管理

1. 系统报警：配置报警阀值和值守人员联系方式，系统发生报警时，即使工程师在千里之外，也可以被及时通知；
2. 失效转移：监控系统在发现故障时，主动通知应用进行失效转移；
3. 自动优雅降级：为了应付网站访问高峰，主动关闭部分功能，释放部分系统资源，保 证核心应用服务的正 常运行；—>网站柔性架构的理想状态

面试真题

1堆栈消息
2如何处理高并发U
3MYSQL优化‘
4ArrayList和linkedlist区别
5．http协议
6.对象的创建过程
7.Spring（aop，ioc）
8.springboot原理
9.redis具体用在项目的哪
缓存怎么做
项目用到多线程了没
处理机制
哨兵模式
主从复制
10.aop项目用到没
11.配置服务器
12.mq消息阻塞怎么
13.dubbo调用失败
14.微服务优点
15jvm
16zookeeper怎么实现分布锁
17.多线程什么情况导致死锁
多线程状态查询
线程间通信’
多线程sychronized和lock区别
18.concurrenthsahmap和hashmap区别
19.同步代码块同步方法的区别
20.重定向和转发能不能调用到同一个web
21.分布式事物在项目里怎么用的
22.消息中间件用了哪些
23.fastdfs上传有什么格式，怎么转码
24单点登录
25.git怎处理冲突
26quartz怎么配置
27订单有几个支付状态
28dubbo之间调用问题
29.Spring SpringMVC Spring Boot 有什么区别?
30.有没有用过 Redis 做异步队列， 你是怎么用的?
31.Dubbo主要的配置项有哪些，作用是什么?
32.如果Dubbo的服务端未启动，消费端能起来吗?

直接面试

1、初试

自我介绍一下

介绍一下你的项目

你负责的模块？主要做了什么？用了什么技术？负责模块的业务流程？

redis内存满了，怎么处理？

A1：

1、修改redis.conf中的maxmemory-policy选项

一般把noeviction改为least Recently Used，最近最少使用算法

2、加内存

3、缩短（或设置）数据过期时间，以释放内存

4、redis集群

当然还有其他更好的方式

A2：

Redis 提供了多种缓存淘汰策略，含义在注释中说的很清楚。

LRU：表示最近最少使用； LFU：表示最不常用的。

区别在于：LFU 是一定时间内访问最少的，比如 10 分钟内访问最少的，而 LRU 则是指 服务启动后，访问量最少的内容。 下面按顺序说明下淘汰策略：

volatile-lru 筛选出设置了有效期的，最近最少使用的 key；

allkeys-lru 所有 key 中，筛选出最近最少使用的 key；

volatile-lfu 筛选出设置了有效期的，最不常用的 key；

valatile-random 随机筛选出设置了有效期的 key；

allkeys-random 所有 key 中，随机筛选出 key进行删除；

volatile-ttl 筛选出所有设置有效期的 key 中，有效期最短的 key；

noeviction 拒绝策略，当内存满了之后，服务不做任何处理，直接返回一个错误 Redis默认是拒绝策略，可根据实际情况做出设置。（默认）

A3：

maxmemory-policy策略尽量不要使用allkeys-random，使用该策略基本每天都会爆发该问题，可以使用allkeys-lru代替

尽量按实际需求设置redis的允许最大内存设置，当redis内存接近允许的最大内存设置时，进行redis集群扩容，避免redis频繁清理内存保证宿主机有足够的内存空间，内存占用高峰值最好不要

tcp/ip,udp,区别？

string和stringbuffer的区别？

设计模式？单例模式？懒汉式，线程安全的问题？

http和HTTPS的区别？https怎么保证安全的？

Linux命令。

查看端口，已知一个字符串，查找哪些文件包含字符串?

手写一个排序算法

2、复试

主要问项目有关的，

订单的状态，

项目中业务的流程

项目的并发。

团队配置。。

说说JVM原理？内存泄漏与溢出的区别？何时产生内存泄漏?

JVM原理：

JVM是Java Virtual Machine（Java虚拟机）的缩写，它是整个java实现跨平台的最核心的部分，所有的Java程序会首先被编译为.class的类文件，这种类文件可以在虚拟机上执行，也就是说class并不直接与机器的操作系统相对应，而是经过虚拟机间接与操作系统交互，由虚拟机将程序解释给本地系统执行。JVM是Java平台的基础，和实际的机器一样，它也有自己的指令集，并且在运行时操作不同的内存域。JVM通过抽象操作系统和CPU结构，提供了一种与平台无关的代码执行9方法，即与特殊的实现方法、主机硬件、主机操作系统无关。JVM的主要工作是解释自己的指令集（即字节码）到CPU的指令集或对应的系统调用，保护用户免被恶意程序骚扰。JVM对上层的Java源文件是不关心的，它关注的只是由源文件生成的类文件（.class文件）。

内存泄漏与溢出的区别：

内存泄漏是指分配出去的内存无法回收了

内存溢出是指程序要求的内存，超出了系统所能分配的范围，从而发生溢出。比如用

byte类型的变量存储10000这个数据，就属于内存溢出。

内存溢出是提供的内存不够；内存泄漏是无法再提供内存资源。

何时产生内存泄漏

静态集合类：在使用Set、Vector、HashMap等集合类的时候需要特别注意，有可能会发生内存泄漏。当这些集合被定义成静态的时候，由于它们的生命周期跟应用程序一样长，这时候，就有可能会发生内存泄漏

监听器：在Java中，我们经常会使用到监听器，如对某个控件添加单击监听器addOnClickListener()，但往往释放对象的时候会忘记删除监听器，这就有可能造成存泄漏。好的方法就是，在释放对象的时候，应该记住释放所有监听器，这就能避免了因为监听器而导致的内存泄漏。

各种连接：Java中的连接包括数据库连接、网络连接和io连接，如果没有显式调用其close()方法，是不会自动关闭的，这些连接就不能被GC回收而导致内存泄漏。一般情况6下，在try代码块里创建连接，在finally里释放连接，就能够避免此类内存泄漏

外部模块的引用：调用外部模块的时候，也应该注意防止内存泄漏。如模块A调用了外部模块B的一个方法，如：public void register(Object o)。这个方法有可能就使得A模块持有传入对象的引用，这时候需要查看B模块是否提供了去除引用的方法，如unregister()。这种情况容易忽略，而且发生了内存泄漏的话，比较难察觉，应该在编写代码过程中就应该注意此类问题。

单例模式：使用单例模式的时候也有可能导致内存泄漏。因为单例对象初始化后将在JVM的整个生命周期内存在，如果它持有一个外部对象（生命周期比较短）的引用，那么这个外部对象就不能被回收，而导致内存泄漏。如果这个外部对象还持有其它对象的引用，那么内存泄漏会更严重，因此需要特别注意此类情况。这种情况就需要考虑下单例模式的设计会不会有问题，应该怎样保证不会产生内存泄漏问

题。

GC线程是否为守护线程?

GC线程是守护线程。线程分为守护线程和非守护线程（即用户线程）。只要当前JVM实例中尚存在任何一个非守护线程没有结束，守护线程就全部工作；只有当最后一个非守护线程结束时，守护线程随着JVM一同结束工作。
垃圾回收器（GC）的本原理是什么？垃圾回收器可以马上回收内存吗？如何通知虚拟机进行垃圾回收?

对于GC来说，当程序员创建对象时，GC就开始监控这个对象的地址、大小以及使用情况。通常，GC采用有向图的方式记录和管理堆(heap)中的所有对象。通过这种方式确定哪些对象是"可达的"，哪些对象是"不可达的"。当GC确定一些对象为"不可达"时，GC就有责任回收这些内存空间。

可以。程序员可以手动执行System.gc()，通知GC运行，但是Java语言规范并不保证GC一定会执行。

System.gc();或者Runtime.getRuntime().gc();

解释内存中的栈（stack）、堆(heap)和静态存储区的用法

通常我们定义一个基本数据类型的变量，一个对象的引用，还有就是函数调用的现场保存都使用内存中的栈空间；而通过new关键字和构造器创建的对象放在堆空间；程序中的字面量（literal）如直接书写的100、“hello”和常量都是放在静态存储区中。栈空间操作最快但是也很小，通常大量的对象都是放在堆空间，整个内存包括硬盘上的虚
拟内存都可以被当成堆空间来使用。
String str = new String(“hello”);

解释内存中的栈（stack）、堆(heap)和静态存储区的用法。

答：通常我们定义一个基本数据类型的变量，一个对象的引用，还有就是函数调用的现场保存都使用内存中的栈空间；而通过new关键字和构造器创建的对象放在堆空间；程序中的字面量（literal）如直接书写的100、“hello”和常量都是放在静态存储区中。栈空间操作最快但是也很小，通常大量的对象都是放在堆空间，整个内存包括硬盘上的虚拟内存都可以被当成堆空间来使用。
String str = new String(“hello”);

上面的语句中str放在栈上，用new创建出来的字符串对象放在堆上，而“hello”这个字面量放在静态存储区 。

补充：较新版本的Java中使用了一项叫“逃逸分析”的技术，可以将一些局部对象放在栈上以提升对象的操作性能。

Java 中会存在内存泄漏吗，请简单描述。

答：理论上Java因为有垃圾回收机制（GC）不会存在内存泄露问题（这也是Java被广泛使用于服务器端编程的一个重要原因）；然而在实际开发中，可能会存在无用但可达的对象，这些对象不能被GC回收也会发生内存泄露。一个例子就是hibernate的Session（一级缓存）中的对象属于持久态，垃圾回收器是不会回收这些对象的，然而这些对象中可能存在无用的垃圾对象。下面的例子也展示了Java中发生内存泄露的情况：

```
package com. atguigu;
import java.util.Arrays;
import java.util.EmptyStackException;
public class MyStack<T> {
private T[] elements;
private int size = 0;
private static final int INIT_CAPACITY = 16;
public MyStack() {
elements = (T[]) new Object[INIT_CAPACITY];
}
    public void push(T elem) {
```

```
    ensureCapacity();
    elements[size++] = elem;
}
    public T pop() {
    if(size == 0)
    throw new EmptyStackException();
    return elements[--];
}
    private void ensureCapacity() {
    if(elements.length == size) {
    elements = Arrays.copyOf(elements, 2 * size + 1);
        }
    }
}
```

上面的代码实现了一个栈（先进后出（FILO））结构，乍看之下似乎没有什么明显的问题，它甚至可以通过你编写的各种单元测试。然而其中的pop方法却存在内存泄露的问题，当我们用pop方法弹出栈中的对象时，该对象不会被当作垃圾回收，即使使用栈的程序不再引用这些对象，因为栈内部维护着对这些对象的过期引用（obsolete reference）。在支持垃圾回收的语言中，内存泄露是很隐蔽的，这种内存泄露其实就是无意识的对象保持。如果一个对象引用被无意识的保留起来了，那么垃圾回收器不会处理这个对象，也不会处理该对象引用的其他对象，即使这样的对象只有少数几个，也可能会导致很多的对象被排除在垃圾回收之外，从而对性能造成重大影响，极端情况下
会引发Disk Paging（物理内存与硬盘的虚拟内存交换数据），甚至造成OutOfMemoryError。

GC 是什么？为什么要有GC?

GC是垃圾收集的意思，内存处理是编程人员容易出现问题的地方，忘记或者错误的内存回收会导致程序或系统的不稳定甚至崩溃，Java提供的GC功能可以自动监测对象是否超过作用域从而达到自动回收内存的目的，Java语言没有提供释放已分配内存的显示操作方法。Java程序员不用担心内存管理，因为垃圾收集器会自动进行管理。要请求垃圾收集，可以调用下面的方法之一：System.gc()
Runtime.getRuntime().gc()，但JVM可以屏蔽掉显示的垃圾回收调用。

垃圾回收可以有效的防止内存泄露，有效的使用可以使用的内存。垃圾回收器通常是作为一个单独的低优先级的线程运行，不可预知的情况下对内存堆中已经死亡的或者长时间没有使用的对象进行清除和回收，程序员不能实时的调用垃圾回收器对某个对象或所有对象进行垃圾回收。在Java诞生初期，垃圾回收是Java最大的亮点之一，因为服务器端的编程需要有效的防止内存泄露问题，然而时过境迁，如今Java的垃圾回收机制已经成为被诟病的东西。移动智能终端用户通常觉得iOS的系统比Android系统有/更好的用户体验，其中一个深层次的原因就在于Android系统中垃圾回收的不可预知性。补充：垃圾回收机制有很多种，包括：分代复制垃圾回收、标记垃圾回收、增量垃圾回收等方式。标准的Java进程既有栈又有堆。栈保存了原始型局部变量，堆保存了要创建的对象。Java平台对堆内存回收和再利用的基本算法被称为标记和清除，但是Java对其进行了改进，采用“分代式垃圾收集”。这种方法会跟Java对象的生命周期将堆内存划分
为不同的区域，在垃圾收集过程中，可能会将对象移动到不同区域：

伊甸园（Eden）：这是对象最初诞生的区域，并且对大多数对象来说，这里是它们唯一存在过的区域。

幸存者乐园（Survivor）：从伊甸园幸存下来的对象会被挪到这里。

终身颐养园（Tenured）：这是足够老的幸存对象的归宿。年轻代收集（MinorGC）过程是不会触及这个地方的。当年轻代收集不能把对象放进终身颐养园时，就会触发一次完全收集（Major-GC），这里可能还会牵扯到压缩，以便为大对象腾出足够的空间。

与垃圾回收相关的JVM参数：

-Xms / -Xmx --- 堆的初始大小 / 堆的最大大小

-Xmn --- 堆中年轻代的大小

-XX:-DisableExplicitGC --- 让System.gc()不产生任何作用

-XX:+PrintGCDetail --- 打印GC的细节

-XX:+PrintGCDateStamps --- 打印GC操作的时间戳

10描述一下JI机制?

JVM中类的装载是由ClassLoader和它的子类来实现的,Java ClassLoader 是一个重要的Java运行时系统组件。它负责在运行时查找和装入类文件的类。

11jvm中一次完整的GC流程（从ygc到fgc）是怎样的，重点讲讲对象如何晋升到老年代等答：对象优先在新生代区中分配，若没有足够空间，Minor GC；大对象（需要大量连续内存空间）直接进入老年态；长期存活的对象进入老年态。如果对象在新生代出生并经过第一次MGC后仍然存活，年龄+1，若年龄超过一定限制（15），则被晋升到老年态。

12Eden和Survivr的比例分配等

默认比例8:1。大部分对象都是朝生夕死。复制算法的基本思想就是将内存分为两块，每次只用其中一块，当这一块内存用完，就
将还活着的对象复制到另外一块上面。复制算法不会产生内存碎片。

13JVM垃圾回收机制，何时触发MinorGC等操作

分代垃圾回收机制：不同的对象生命周期不同。把不同生命周期的对象放在不同代上，不同代上采用最合适它的垃圾回收方式进行回收。

JVM中共划分为三个代：年轻代、年老代和持久代，

年轻代：存放所有新生成的对象；

年老代：在年轻代中经历了N次垃圾回收仍然存活的对象，将被放到年老代中，故都是一些生命周期较长的对象；

持久代：用于存放静态文件，如Java类、方法等。

新生代的垃圾收集器命名为“minor gc”，老生代的GC命名为”Full Gc 或者Major GC”.

其中用System.gc()强制执行的是Full Gc

判断对象是否需要回收的方法有两种：

1.引用计数

当某对象的引用数为0时，便可以进行垃圾收集。

2.对象引用遍历

触发GC（Garbage Collector）的条件：

1)GC在优先级最低的线程中运行，一般在应用程序空闲即没有应用线程在运行时被调用。

2)Java堆内存不足时，GC会被调用

14深入分析了Classloader，双亲委派机制

ClassLoader：类加载器（class loader）用来加载 Java 类到 Java 虚拟机中。Java源程序（.java 文件）在经过 Java 编译器编译之后就被转换成 Java 字节代码（.class文件）。类加载器负责读取 Java 字节代码，并转换成 java.lang.Class 类的一个实例。
双亲委派机制：某个特定的类加载器在接到加载类的请求时，首先将加载任务委托给父类加载器，依次递归，如果父类加载器可以完成类加载任务，就成功返回；只有父类加载器无法完成此加载任务时，才自己去加载。

1，单例模式的优缺点

优点
1，减少对象创建，节省系统资源，提高系统性能。
缺点：
1，不利于扩展。

2，jsp的内置对象与作用

JSP中一共预先定义了9个这样的对象，分别为：request、response、session、application、out、pagecontext、config、page、exception
1、request对象
request 对象是 javax.servlet.httpServletRequest类型的对象。 该对象代表了客户端的请求信息，主要用于接受通过HTTP协议传送到服务器的数据。（包括头信息、系统信息、请求方式以及请求参数等）。request对象的作用域为一次请求。
2、response对象
response 代表的是对客户端的响应，主要是将JSP容器处理过的对象传回到客户端。response对象也具有作用域，它只在JSP页面内有效。
3、session对象
session 对象是由服务器自动创建的与用户请求相关的对象。服务器为每个用户都生成一个session对象，用于保存该用户的信息，跟踪用户的操作状态。session对象内部使用Map类来保存数据，因此保存数据的格式为 “Key/value”。 session对象的value可以使复杂的对象类型，而不仅仅局限于字符串类型。
4、application对象
application 对象可将信息保存在服务器中，直到服务器关闭，否则application对象中保存的信息会在整个应用中都有效。与session对象相比，application对象生命周期更长，类似于系统的“全局变量”。
5、out 对象
out 对象用于在Web浏览器内输出信息，并且管理应用服务器上的输出缓冲区。在使用 out 对象输出数据时，可以对数据缓冲区进行操作，及时清除缓冲区中的残余数据，为其他的输出让出缓冲空间。待数据输出完毕后，要及时关闭输出流。
6、pageContext 对象
pageContext 对象的作用是取得任何范围的参数，通过它可以获取 JSP页面的out、request、reponse、session、application 等对象。pageContext对象的创建和初始化都是由容器来完成的，在JSP页面中可以直接使用 pageContext对象。
7、config 对象
config 对象的主要作用是取得服务器的配置信息。通过 pageConext对象的getServletConfig() 方法可以获取一个config对象。当一个Servlet 初始化时，容器把某些信息通过 config对象传递给这个 Servlet。开发者可以在web.xml 文件中为应用程序环境中的Servlet程序和JSP页面提供初始化参数。
8、page 对象
page 对象代表JSP本身，只有在JSP页面内才是合法的。 page隐含对象本质上包含当前 Servlet接口引用的变量，类似于Java编程中的 this 指针。
9、exception 对象
exception 对象的作用是显示异常信息，只有在包含 isErrorPage="true" 的页面中才可以被使用，在一般的JSP页面中使用该对象将无法编译JSP文件。excepation对象和Java的所有对象一样，都具有系统提供的继承结构。exception 对象几乎定义了所
3，Integer a = new Integer(100);
Integer b= new Integer(100);
Integer a= new Integer(200);
Integer b= new Integer(200);
a==b false
a.equals（b）true
哪个是true哪个是false
4，将一组字符，串去重并排序
不可以用数组排序
5，逻辑题

一个试管10ml液体，分别有7ml，3ml两个试管，如何分成5，5
1、cookie伪造问题（我们购物车用的cookie+redis；最初用户信息是保存在cookie中，若有人伪造了一个cookie登录，我们该如何处理？）
答：关于cookie伪造问题，我们可以先使用 密钥密码，生成 3DES密钥；再使用 3DES密钥 解密 Token Cookie。
详情请参考：
https://www.cnblogs.com/zhengyun_ustc/archive/2012/11/17/topic3.html
2、恶意攻击购物车问题（在购物车中恶意攻击不停的往购物车中添加商品，关于这个问题有项目中有没有什么限制？）
答：可以通过IP获取，判断该用户是否是恶意操作，（比如：一个用户一分钟之内，添加商品超过30件）；如果有恶意操作，我们可以短时间内禁止它的访问路径（比如5~10分钟）。
3、商品表中价格调整问题（针对节假日活动，不同日期商品的价格不同，关于价格调整问题一般如何处理的？）
答：针对节假日活动，可能有大量商品需要价格调整，我们一般会设置一个定时器，去商品系统里检索，要价格调整的商品，统一调整（要用到人工智能的一些技术），有特殊的商品我们可能需要手动调整。
（若不好回答，可先说项目初期，我们商品的数量不算太多，我们目前是手动修改的。）
4、针对商品不同颜色和尺寸，数据库是如何存储的（如苹果手机有不同的颜色和尺寸，数
据库是如何存储的）
答：数据库字段存json格式数据，通过使用模板设计理念来实现颜色和尺寸；
针对不同类型商品模板不同，同类产品模板相同。
v
有异常情况。在Java程序中，可以使用try/catch关键字来处理异常情况； 如果在JSP页面中出现没有捕获到的异常，就会生成 exception 对象，并把 exception 对象传送到在page指令中设定的错误页面中，然后在错误页面中处理相应的 exception 对象。

3，Integer a = new Integer(100);
Integer b= new Integer(100);
Integer a= new Integer(200);
Integer b= new Integer(200);
a==b false
a.equals（b）true
哪个是true哪个是false

4，将一组字符，串去重并排序

不可以用数组排序

5，逻辑题

一个试管10ml液体，分别有7ml，3ml两个试管，如何分成5，5
1、cookie伪造问题（我们购物车用的cookie+redis；最初用户信息是保存在cookie中，若有人伪造了一个cookie登录，我们该如何处理？）
答：关于cookie伪造问题，我们可以先使用 密钥密码，生成 3DES密钥；再使用 3DES密钥 解密 Token Cookie。
详情请参考：
https://www.cnblogs.com/zhengyun_ustc/archive/2012/11/17/topic3.html

2、恶意攻击购物车问题（在购物车中恶意攻击不停的往购物车中添加商品，关于这个问题有项目中有没有什么限制？）

答：可以通过IP获取，判断该用户是否是恶意操作，（比如：一个用户一分钟之内，添加商品超过30件）；如果有恶意操作，我们可以短时间内禁止它的访问路径（比如5~10分钟）

商品表中价格调整问题（针对节假日活动，不同日期商品的价格不同，关于价格调整问题一般如何处理的？）

答：针对节假日活动，可能有大量商品需要价格调整，我们一般会设置一个定时器，去商品系统里检索，要价格调整的商品，统一调整（要用到人工智能的一些技术），有特殊的商品我们可能需要手动调整。

（若不好回答，可先说项目初期，我们商品的数量不算太多，我们目前是手动修改的。）

4、针对商品不同颜色和尺寸，数据库是如何存储的（如苹果手机有不同的颜色和尺寸，数据库是如何存储的）

1创建线程的方式

比较常见的一个问题了，一般就是四种：

继承Thread类

实现Runnable接口

实现 Callable 接口：该方式相较于实现 Runnable 接口，方法多了返回值，并且可以抛出异常
FutureTask

线程池

2 start()方法和run()方法的区别

只有调用了start()方法，才会表现出多线程的特性，不同线程的run()方法里面的代码交替执行。如果只是调用run()方法，那么代码还是同步执行的，必须等待一个线程的run()方法里面的代码 全部执行完毕之后，另外一个线程才可以执行其run()方法里面的代码。

3什么是线程安全

如果你的代码在多线程下执行和在单线程下执行永远都能获得一样的结果，那么你的代码就是 线程安全的 。

不可变

像String、Integer、Long这些，都是final类型的类，任何一个线程都改变不了它们的值，要改变除非新创建一个，因此这些不可变对象不需要任何同步手段就可以直接在多线程环境下使用

设置不可变对象的原因:

因为不变对象一旦创建,对象内部的数据就不能修改,这样就减少了由于修改数据导致的错误.此外,由于对象不变,多任务环境下同时读取对象不需要加锁,同时读取数据时不会有任何问题.我们在编写程序时,如果可以设计一个不变对象,那就尽量设计成不变对象.

绝对线程安全

不管运行时环境如何，调用者都不需要额外的同步措施。要做到这一点通常需要付出许多额外的代价，Java中标注自己是线程安全的类，实际上绝大多数都不是线程安全的，不过绝对线程安全的类，Java中也有，比方说CopyOnWriteArrayList、CopyOnWriteArraySet

相对线程安全

相对线程安全也就是我们通常意义上所说的线程安全，像Vector这种，add、remove方法都是原子操作，不会被打断，但也仅限于此，如果有个线程在遍历某个Vector、有个线程同时在add这个Vector，99%的情况下都会出现ConcurrentModificationException，也就是 fail-fast机制

线程非安全

这个就没什么好说的了，ArrayList、LinkedList、HashMap等都是线程非安全的类

Volatile

volatile 当一个共享变量被volatile修饰时，它会保证修改的值会立即被更新到主存，当有其他线程需要读取时，它会去内存中读取新值。

一旦一个共享变量（类的成员变量、类的静态成员变量）被volatile修饰之后，那么就具备了两层语义：
1）保证了不同线程对这个变量进行操作时的可见性，即一个线程修改了某个变量的值，这新值对其他线程来说是立即可见的。

禁止进行指令重排序。

volatile的一个重要作用就是和CAS结合，保证了原子性，

与多线程相比

1、对于多线程，不是一种互斥关系

2、不能保证变量状态的"原子性操作

5 CAS算法

CAS (Compare-And-Swap) 是一种硬件对并发的支持，针对多处理器操作而设计的处理器中的一种特殊指令，用于管理对共享数据的并发访问。

CAS 是一种无锁的非阻塞算法的实现。

CAS 包含了 3 个操作数：

需要读写的内存值 V

进行比较的值 A

拟写入的新值 B

当且仅当 V 的值等于 A 时，CAS 通过原子方式用新值 B 来更新 V 的值，否则不会执行任何操作。

通过版本戳解决aba问题

6 如何在两个线程之间共享数据

通过在线程之间共享对象就可以了，然后通过wait/notify/notifyAll、await/signal/signalAll进行唤起和等待，比方说阻塞队列BlockingQueue就是为线程之间共享数据而设计的

7 在java中wait和sleep方法的不同

最大的不同是在等待时wait会释放锁，而sleep一直持有锁。Wait通常被用于线程间交互，sleep通常被用于暂停执行。

8 ThreadLocal有什么用

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

因为ThreadLocal在每个线程中对该变量会创建一个副本，即每个线程内部都会有一个该变量，且在线程内部任何地方都可以使用，线程之间互不影响，这样一来就不存在线程安全问题，也不会严重影响程序执行性能。

9 为什么wait()方法和notify()/notifyAll()方法要在同步块中被调用

这是JDK强制的，wait()方法和notify()/notifyAll()方法在调用前都必须先获得对象的锁

10 wait()方法和notify()/notifyAll()方法在放弃对象监视器时有什么区别

wait()方法和notify()/notifyAll()方法在放弃对象监视器的时候的区别在于： wait()方法立即释放对象监视器，notify()/notifyAll()方法则会等待线程剩余代码执行完毕才会放弃对象监视器 。

11为什么要使用线程池

避免频繁地创建和销毁线程，达到线程对象的重用。另外，使用线程池还可以根据项目灵活地控制并发的数目。

12 synchronized和ReentrantLock的区别

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

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

（2）ReentrantLock可以获取各种锁的信息

（3）ReentrantLock可以灵活地实现多路通知

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

13ConcurrentHashMap的并发度是什么

ConcurrentHashMap的并发度就是segment的大小，默认为16最多65536 2*16，这意味着最多同时可以有16条线程操作ConcurrentHashMap，这也是ConcurrentHashMap对Hashtable的最大优势，任何情况下，Hashtable能同时有两条线程获取Hashtable中的数据吗?

JDK1.8的实现已经摒弃了Segment的概念，而是直接用Node数组+链表+红黑树的数据结构来实现，并发控制使用Synchronized和CAS来操作，整个看起来就像是优化过线程安全的HashMap

volatile和synchronize的比较：

1.volatile是线程同步的轻量级实现，所以volatile的性能要比synchronize好；volatile只能用于修饰变量，synchronize可以用于修饰方法、代码块。随着jdk技术的发展，synchronize在执行效率上会得到较大提升，所以synchronize在项目过程中还是较为常见的；

2.多线程访问volatile不会发生阻塞；而synchronize会发生阻塞；

3.volatile能保证变量在私有内存和主内存间的同步，但不能保证变量的原子性；synchronize可以保证变量原子性；

4.volatile是变量在多线程之间的可见性；synchronize是多线程之间访问资源的同步性；

对于volatile修饰的变量，可以解决变量读时可见性问题，无法保证原子性。对于多线程访问同一个实例变量还是需要加锁同步。

14 FutureTask是什么

这个其实前面有提到过，接口future ，实现类FutureTask表示一个异步运算的任务。FutureTask里面可以传入一个Callable的具体实现类，可以对这个异步运算的任务的结果进行等待获取、判断是否已经完成、取消任务等操作。当然，由于FutureTask也是Runnable接口的实现类，所以FutureTask也可以放入线程池中。task.get()获得callable接口的返回值

15怎么唤醒一个阻塞的线程

如果线程是因为调用了wait()、sleep()或者join()方法而导致的阻塞，可以中断线程t.interrupt();，并且通过抛出InterruptedException来唤醒它；如果线程遇到了IO阻塞，无能为力，因为IO是操作系统实现的，Java代码并没有办法直接接触到操作系统，比如Socket的I/O，关闭底层套接字，然后抛出异常处理就好了;比如同步I/O，关闭底层Channel然后处理异常。对 于Lock.lock方法，我们可以改造成Lock.lockInterruptibly方法去实现。

16 ReadWriteLock是什么

首先明确一下，不是说ReentrantLock不好，只是ReentrantLock某些时候有局限。如果使用ReentrantLock，可能本身是为了防止线程A在写数据、线程B在读数据造成的数据不一致，但这样，如果线程C在读数据、线程D也在读数据，读数据是不会改变数据的，没有必要加锁，但是还是加锁了，降低了程序的性能。因为这个，才诞生了读写锁ReadWriteLock。ReadWriteLock是一个读写锁接口，ReentrantReadWriteLock是ReadWriteLock接口的一个具体实现，实现了读写的分离， 读锁是共享的，写锁是独占的 ，读和读之间不会互斥，读和写、写和读、写和写之间才会互斥，提升了读写

17 如果你提交任务时，线程池队列已满，这时会发生什么

如果你使用的LinkedBlockingQueue，也就是无界队列的话，没关系，继续添加任务到阻塞队列中等待执行，因为LinkedBlockingQueue可以近乎认为是一个无穷大的队列，可以无限存放任务；如果你使用的是有界队列比方说ArrayBlockingQueue的话，任务首先会被添加到ArrayBlockingQueue中，ArrayBlockingQueue满了，则会使用拒绝策略RejectedExecutionHandler处理满了的任务，默认是AbortPolicy。

18 Java中用到的线程调度算法是什么

抢占式。一个线程用完CPU之后，操作系统会根据线程优先级、线程饥饿情况等数据算出一个总的优先级并分配下一个时间片给某个线程执行。

19 Thread.sleep(0)的作用是什么

这个问题和上面那个问题是相关的，我就连在一起了。由于Java采用抢占式的线程调度算法，因此可能会出现某条线程常常获取到CPU控制权的情况，为了让某些优先级比较低的线程也能获取到CPU控制权，可以使用Thread.sleep(0)手动触发一次操作系统分配时间片的操作，这也是平衡CPU控制权的一种操作

20什么是乐观锁和悲观锁

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

乐观锁(Optimistic Lock), 顾名思义，就是很乐观，每次去拿数据的时候都认为别人不会修改，所以不会上锁，但是在更新的时候会判断一下在此期间别人有没有去更新这个数据，可以使用版本号等机制。乐观锁适用于多读的应用类型，这样可以提高吞吐量，像数据库如果提供类似于write_condition机制的其实都是提供的乐观锁。

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

21 生产者、消费者有很多的实现方法：

- 用wait() / notify()方法 解决了线程没有关闭一直等待的问题，if改while解决多个生产者消费者的问题

- 用Lock的多Condition方法

- BlockingQueue阻塞队列方法

22在Java中CycliBarriar和CountdownLatch有什么区别?

CountDownLatch :java.util.concurrent下，都可以用来表示代码运行到某个点上，二者的区别在于

1）CyclicBarrier的某个线程运行到某个点上之后，该线程即停止运行，直到所有的线程都到达了这个点，所有线程才重新运行；CountDownLatch则不是，某线程运行到某个点上之后，只是给某个数值-1而已，该线程继续运行

2）CyclicBarrier只能唤起一个任务，CountDownLatch可以唤起多个任务

3）CyclicBarrier可重用，CountDownLatch不可重用，计数值为0该CountDownLatch就不可再用了

23总结下CountDownLatch与join的区别：调用thread.join() 方法必须等thread 执行完毕，当前线程才能继续往下执行，而CountDownLatch通过计数器提供了更灵活的控制，只要检测到计数器为0当前线程就可以往下执行而不用管相应的thread是否执行完毕。

23同步方法和同步块，哪个是更好的选择

同步块，这意味着同步块之外的代码是异步执行的，这比同步整个方法更提升代码的效率。请知道一条原则：同步的范围越少越好。借着这一条，我额外提一点，虽说同步的范围越少越好，但是在Java虚拟机中还是存在着一种叫做 锁粗化 的优化方法，这种方法就是把同步范围变大。这是有用的，比方说StringBuffer，它是一个线程安全的类，自然最常用的append()方法是一个同步方法，我们写代码的时候会反复append字符串，这意味着要进行反复的加锁->解锁，这对性能不利，因为这意味着Java虚拟机在这条线程上要反复地在内核态和用户态之间进行切换，因此Java虚拟机会将多次append方法调用的代码进行一个锁粗化的操作，将多次的append的操作扩展到append方法的头尾，变成一个大的同步块，这样就减少了加锁–>解锁的次数，有效地提升了代码执行的效率。

24高并发、任务执行时间短的业务怎样使用线程池？并发不高、任务执行时间长的业务怎样使用线程池？并发高、业务执行时间长的业务怎样使用线程池？

（1）高并发、任务执行时间短的业务，线程池线程数可以设置为CPU核数+1，减少线程上下文的切换

(2）并发不高、任务执行时间长的业务要区分开看：a）假如是业务时间长集中在IO操作上，也就是IO密集型的任务，因为IO操作并不占用CPU，所以不要让所有的CPU闲下来，可以加大线程池中的线程数目，让CPU处理更多的业务

b）假如是业务时间长集中在计算操作上，也就是计算密集型任务，这个就没办法了，和（1）一样吧，线程池中的线程数设置得少一些，减少线程上下文的切换

3）并发高、业务执行时间长，解决这种类型任务的关键不在于线程池而在于整体架构的设计，看看这些业务里面某些数据是否能做缓存是第一步，增加服务器是第二步，至于线程池的设置，设置参（2）。最后，业务执行时间长的问题，也可能需要分析一下，看看能不能使用中间件对任务进行拆分和解耦