项目描述：

1：采用Spring、springMVC、mybatis搭建网站后台框架；

2：使用mysql数据库存储数据，

添加购物车

添加购物车的时候，在商品页面点击添加购物车的时候，会把商品信息包括 商品id 名字 型号 数量等信息 发送给后台控制器，控制器把这些数据传给service层，然后 service传给dao层，dao层操作mysql数据库增加购物车信息。

购物车展示

Dao层采用逆向工程生成Dao成对应的mapper.Xml文件，以及数据库对应的java类，然后service层调用对应的mapper对象，根据当前登录用户的Id查询相应Id下面的所有购物车信息，然后封装成一个商品信息类，里面包含商品的基本信息（id name miaoshu 单价 等）然后还有购物数量。然后控制层直接调用service层的方法，传入用户id返回对应的购物车信息的集合。然后这个集合一json数据格式返回给jsp页面展示。

订单模块 新建订单、取消订单

在购物车界面点击提交订单的时候，先校验订单是否有效，包括生成订单的商品是否存在而且有效，库存是否充足等，然后校验用户地址信息和手机信息是否是没有问题，一切没有问题后，调用支付模块，进行支付，如果支付完成，会返回一个信息给订单模块，这时候，我这边就会把订单相应的信息传给后台存入数据库。如果用户在支付阶段出现错误或者没有成功，订单取消，如果用户在支付之后取消的话，还是调用订单模块，把相应订单的状态改为待取消状态，然后调用支付模块进行退款操作，根据返回结果进行订单取消操作。

订单完成

订单完成需要等待物流系统返回的数据，如果返回订单成功，那么订单模块就把相应的订单置为已完成，然后调用支付模块进行最后的支付。

订单展示

用户打开订单界面的时候，先进行用户校验，成功后，显示用户所有的订单信息。

单例模式

私有化构造函数，声明一个自己类的对象，对外提供一个静态方法获取本类对象，里面就是创建对象，这是懒汉式，如果是饿汉式，就是在声明本类的时候，直接创建对象，然后

这样的话，就存在线程安全的问题，所以 可以把 这个对外提供的获取本类对象的方法使用同步锁锁起来，这样就能保证线程安全了， 但是这样的话，又会存在一个效率的问题 因为在方法上面加的同步锁，所以如果是已经创建了对象的时候，再来请求这个对象的时候，还是要等。

这时候就有了第三种方式，双重检查锁定：

在对象提供创建对象的方法上面不加同步锁，而是先进行判断，如果已经创建了对象，那么就直接返回对象，这样就不需要进入同步锁，如果对象还没有创建，那就进入同步锁，创建对象。这样比前面的就少了后面等待锁资源的时间消耗

客户注册和员工注册不同，客户注册使用单独的注册页面，员工注册需要具有注册员工账户的账号来建立，没有管理账户权限的账户不能看到建立员工用户的页面。

普通用户访问本网站或使用移动端访问本系统的时候，不需要登录，当需要进行交易的时候，就需要登录了，如果没有账号，需要先注册，注册的时候页面先进性校验，用户名是否合法，密码是否合法等，然后传注册信息到后台业务层后，再进行一次校验。没有问题就把用户信息加入到数据库，登录的时候根据用户名和密码到数据库查询相应用户，如果存在登录成功，然后把相应的用户信息存入到session中，返回一个sessionid给web端，这样用户访问别的页面的时候就不用再次登录了。

#### springcloud的理解，你用什么组件？

#### 概念

SpringCloud是基于SpringBoot的一套实现微服务的框架。它提供了微服务开发所需的配置管理、服务发现、断路器、智能路由、微代理、控制总线、全局锁、决策竞选、分布式会话和集群状态管理等组件。最重要的是，跟SpringBoot框架一起使用的话，会让你开发微服务架构的云服务非常方便。

#### 五大组件

**SpringCloud五大核心组件**

服务注册发现-Netflix Eureka

配置中心 - spring cloud config

负载均衡-Netflix Ribbon

断路器 - Netflix Hystrix

路由(网关) - Netflix Zuul

#### 2.mybatis中$和#的区别，原理是什么

1．#{}传入的参数会在动态解析的时候加上双引号，当成字符串处理，例如一个查询语句：select \* from A where id= #{id} 那么就相当于是select \* from A where id= “id” 而${}里面的参数在动态解析的时候会直接把里面的内容直接显示在sql语句中。

2．#方式能够很大程度上防止sql注入，而$无法防止sql的注入，

3．5.$方式一般用于传入数据库对象，例如传入表名.

4.6.一般能用#的就别用$.

5．MyBatis排序时使用order by 动态参数时需要注意，用$而不是。

**${ } 变量的替换阶段是在动态 SQL 解析阶段，而 #{ }变量的替换是在 DBMS 中**

3.redis缓存，常用命令

4.了解java中的锁吗

Druid框架

分布式锁

公平锁和非公平锁：

公平锁是指多个线程按照申请锁的顺序来获取锁。

非公平锁是指多个线程获取锁的顺序并不是按照申请锁的顺序，有可能后申请的线程比先申请的线程优先获取锁。有可能，会造成优先级反转或者饥饿现象。

对于Java ReentrantLock而言，通过构造函数指定该锁是否是公平锁，默认是非公平锁。非公平锁的优点在于吞吐量比公平锁大。

对于Synchronized而言，也是一种非公平锁。

可重入锁：

可重入锁又名递归锁，是指在同一个线程在外层方法获取锁的时候，在进入内层方法会自动获取锁

对于Synchronized而言,也是一个可重入锁。可重入锁的一个好处是可一定程度避免死锁。

互斥锁和读写锁

互斥锁就是独享锁，一次只能给一个线程所持有，

读写锁，读锁是共享锁，可以给多条线程所持有，写锁是独享锁，只能给一条线程所持有

分段锁：是一种锁的设计，并不是一种具体的锁，我们以ConcurrentHashMap来说一下分段锁的含义以及设计思想，ConcurrentHashMap中的分段锁称为Segment，它即类似于HashMap(JDK7与JDK8中HashMap的实现)的结构，即内部拥有一个Entry数组，数组中的每个元素又是一个链表;同时又是一个ReentrantLock(Segment继承了ReentrantLock)。

当需要put元素的时候，并不是对整个hashmap进行加锁，而是先通过hashcode来知道他要放在那一个分段中，然后对这个分段进行加锁，所以当多线程put的时候，只要不是放在一个分段中，就实现了真正的并行的插入。

分段锁的设计目的是细化锁的粒度，当操作不需要更新整个数组的时候，就仅仅针对数组中的一项进行加锁操作。

5.多线程做过吗？   
6.spring中的事务  
7.数据库的主从复制

mysql主从复制用途

* 实时灾备，用于故障切换
* 读写分离，提供查询服务
* 备份，避免影响业务

主从部署必要条件：

* 主库开启binlog日志（设置log-bin参数）
* 主从server-id不同
* 从库服务器能连通主库

原理

从库生成两个线程，一个I/O线程，一个SQL线程；

i/o线程去请求主库 的二进制日志binlog，并将得到的binlog日志写到relay log（中继日志） 文件中；

主库会生成一个 log dump 线程，用来给从库 i/o线程传binlog；

SQL 线程，会读取relay log文件中的日志，并解析成具体操作，来实现主从的操作一致，而最终数据一致；

8.nio了解吗，你说一下

NIO的全称是New I/O，与之相对应的是Java中传统的I/O

而NIO包提供的IO是同步非阻塞IO，非阻塞体现在用户线程发起IO请求后，会直接得到返回结果，即便在数据未就绪的情况下，也能马上得到失败信息。而同步体现在用户线程需要主动去轮询直到发现数据就绪，再主动将数据从内核拷贝到用户线程。服务器实现模式为多个连接一个线程（IO多路复用），即客户端发送的连接请求都会注册到多路复用器上，多路复用器轮询到连接有I/O请求时才启动一个线程进行处理。

* 1. Channel：Java NIO中的所有I/O操作都基于Channel对象，一个Channel（通道）代表和某一实体的连接，这个实体可以是文件、网络套接字等。也就是说，通道是Java NIO提供的一座桥梁，用于我们的程序和操作系统底层I/O服务进行交互。
  2. SocketChannel/ServerSocketChannel：套接字通道，通过静态方法获取实例，使用场景在最后会给出demo。

（2）Buffer：NIO中所使用的缓冲区不是一个简单的byte数组，而是封装过的Buffer类，通过它提供的API，我们可以灵活的操纵数据。与Java基本类型相对应，NIO提供了多种 Buffer 类型，如ByteBuffer、CharBuffer、IntBuffer等，区别就是读写缓冲区时的单位长度不一样（以对应类型的变量为单位进行读写）。Buffer中有3个很重要的变量，它们是理解Buffer工作机制的关键，分别是capacity （总容量）、position （指针当前位置）、limit （读/写边界位置）。

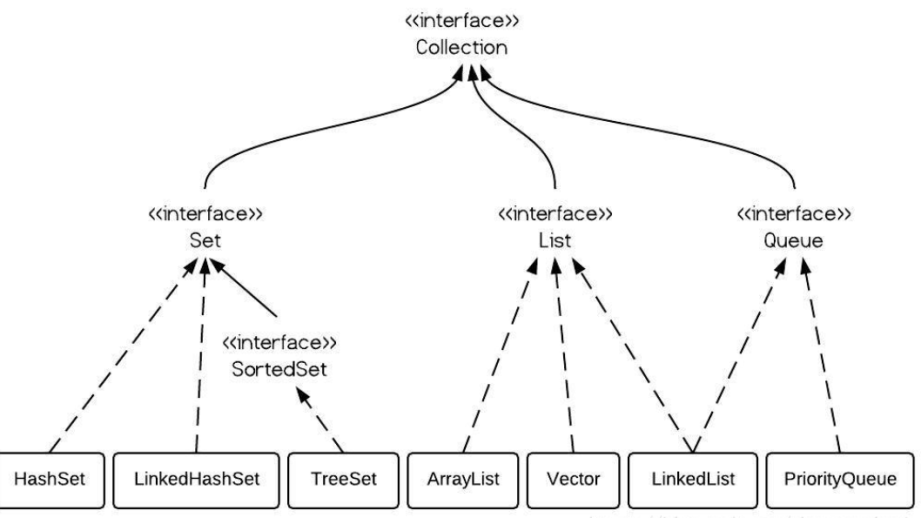
9.Linux的常用命令

ls cd mkdir rmdir 创建（touch）删除（rm）文件 mv cp 查看命令 cat more pwd

vim编辑命令

yum 安装卸载软件和组件命令

#### List ,Set,Map 的区别



1. List，Set的顶层父类为Collection，List的特点是内容有序可重复，Set是无序，不可重复
2. List的实现类有ArrayList，LinkedList，Vector
   1. ArrayList底层是数组，查询快，增删慢，线程不安全的，效率高；
   2. Vector底层是数组，查询快，增删慢，是线程安全的，效率低；
   3. LinkedList底层是链表，查询慢，增删快，是线程不安全的，效率高；
3. Set的实现类，HashSet、LinkedHashSet、TreeSet
   1. HashSet底层是hash表，特点是无序，唯一，通过hash()和equals()方法保证元素唯一
   2. LinkedHashSet底层是由hash表和链表组成，通过链表保证元素有序，通过hash表表征元素唯一。
   3. TreeSet底层是红黑树，特点是唯一且有序，通过自然排序和比较器排序保证元素有序，通过比较的返回值是否为0，来决定元素是否唯一。
4. Set的三个实现都不是线程安全的，如果要保证线程安全，可以使用Collections.synchronizedSet([Set](mk:@MSITStore:C:\Users\Administrator\Desktop\庄致科20180815\私房菜\API\JDK_API_1_6_zh_CN.CHM::/java/util/Set.html)<T> s)

#### Map的底层

1. Map有三个主要实现，HashMap，HashTable，TreeMap
2. TreeMap底层是红黑树，所以存储的内容是有序的；
3. HashMap和HashTable是无序的；
4. HashMap和TreeMap都是线程不安全的，HashTable是线程安全的，所以效率低
5. HashMap底层是数组加链表，在JDK1.8之后采用数组加链表加红黑树的结构；
6. HashMap存储数据原理：默认大小是16，默认加载因子0.75（即在map 储存12个的时候默认扩容2倍），存储的时候，根据key计算key的hash值，然后根据hash值在数组中找到指定位置，然后查看指定位置是否有数据，如果有数据，逐个通过equals方法比较key，如果都不相同，就把这个新加入的key-value对放在链表头，如果有相同的key，就用新加入的key对应的value，覆盖之前的value，如果通过hash值找到的位置没有数据，就直接放到链表头。
7. Jdk1.8以后，如果某一个链表中存储的key-value键值对超过8，就把链表转化为红黑树，每一个红黑树节点存放一个键值对。

归纳起来简单地说，HashMap 在底层将 key-value 当成一个整体进行处理，这个整体就是一个 Entry 对象。HashMap 底层采用一个 Entry[] 数组来保存所有的 key-value 对，当需要存储一个 Entry 对象时，会根据hash算法来决定其在数组中的存储位置，在根据equals方法决定其在该数组位置上的链表中的存储位置；当需要取出一个Entry时，也会根据hash算法找到其在数组中的存储位置，再根据equals方法从该位置上的链表中取出该Entry。

8、考虑到HashMap扩容比较耗费资源，所以如果在知道需要存储的Map数量时，尽量在创建的时候给出容量大小，防止Map扩容。

10.说5个常用的包

Java lang包：String类，包装类，系统类，thread类

Java util包：集合类包括list，set map集合

Java IO包，IO流的一些类，

Java math包 一些用于数学计算的类，比如求平均值等

Java net 网络相关的一些类，socket ，serversocket

11.说下springmvc中常用的注解

@Test

@Conponent，@controller @Service @Pram

@Aspect

@autowired @resourse

@requestMapping

@responsebody

12.git了解吗，你说一下   
13.jekins了解吗，你说一下

了解的不多，知道他是用来持续集成的软件，把他配置到spring容器中，容器启动的时候就自动启动这个jenkens了。

用途主要是持续集成，也就是说给他配置一个代码仓库，把里面的项目编译部署起来后，当仓库的代码发生变化的时候，会自动重新运行代码，这样能起到测试的作用。

* l 持续、自动地构建/测试软件项目。
* l 监控一些定时执行的任务。
* 构建项目完成后，给指定的邮箱发送构建结果邮件

项目中遇到的问题：

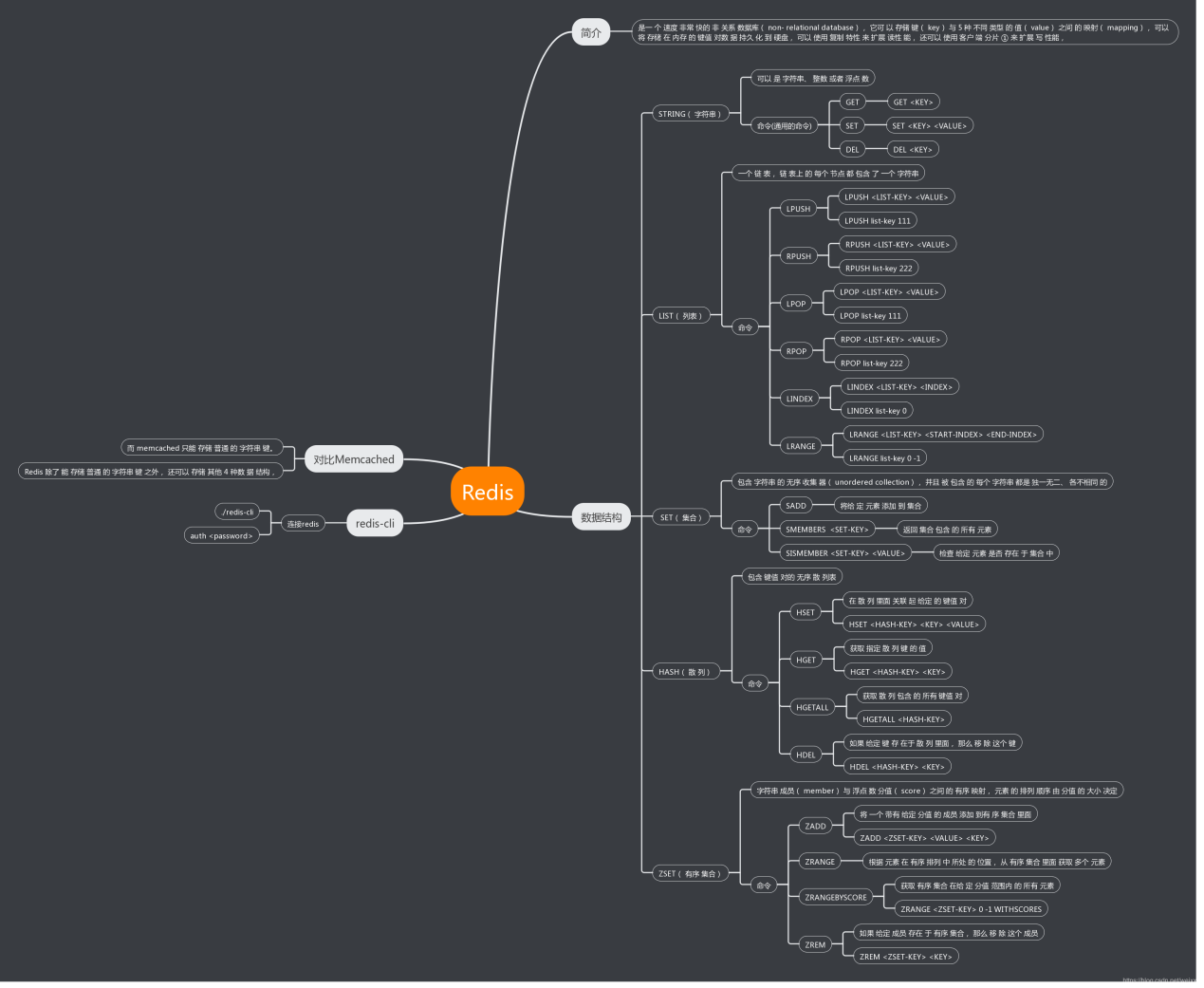
1、在做电商项目的时候，做首页向门户请求数据的时候，采用的是ajax请求，没有参数，请求结果是一个json数据，url是一个.html结尾的url,然后一请求就报406错误，调试了好长时间还是没有解决，在网上找答案说是可能没有引入jackson.jar包，但是我查看项目里面是有jackson包的，而且别的请求也没有什么问题就是这个请求始终报错，当天加班到10点还是没有找出原因，

回去之后在网上找解决的法子，最终找到一个解决方案，试试该url后缀名，说是Html的后缀，不能返回json数据格式的数据，因为当url为HTML的时候，浏览器默认返回的是一个HTML文件，但是返回的是json字符串的时候，解析出现了问题。后来就去问经理看这个问题怎么解决，因为当时是已经做了一段时间，而且配置文件已经做成拦截\*.html 的了，所以只能在web.xml里面在配置一个Servlet-patton，拦截\*.action 的请求。

Redis集群也叫redis cluster

./redis-trib.rb create --replicas 1 192.168.32.128:7001 192.168.32.128:7002 192.168.32.128:7003 192.168.32.128:7004192.168.32.128:7005 192.168.32.128:7006

#### Redis常用命令：



Redis持久化的两种方式：

 Redis 分别提供了 RDB 和 AOF 两种[持久化](http://en.wikipedia.org/wiki/Persistence_(computer_science))模式。

持久化之RDB

定义：在指定的时间间隔内生成数据集的时间点快照（point-in-time snapshot）

RDB 的优点

1.RDB 是一个非常紧凑（compact）的文件，它保存了 Redis 在某个时间点上的数据集。 这种文件非常适合用于进行备份： 比如说，你可以在最近的 24 小时内，每小时备份一次 RDB 文件，并且在每个月的每一天，也备份一个 RDB 文件。 这样的话，即使遇上问题，也可以随时将数据集还原到不同的版本。

2.RDB 非常适用于灾难恢复（disaster recovery）：它只有一个文件，并且内容都非常紧凑，可以（在加密后）将它传送到别的数据中心，或者亚马逊 S3 中。

3.RDB 可以最大化 Redis 的性能：父进程在保存 RDB 文件时唯一要做的就是 fork 出一个子进程，然后这个子进程就会处理接下来的所有保存工作，父进程无须执行任何磁盘 I/O 操作。

4.RDB 在恢复大数据集时的速度比 AOF 的恢复速度要快。

RDB 的缺点

如果你需要尽量避免在服务器故障时丢失数据，那么 RDB 不适合你。 虽然 Redis 允许你设置不同的保存点（save point）来控制保存 RDB 文件的频率， 但是， 因为RDB 文件需要保存整个数据集的状态， 所以它并不是一个轻松的操作。 因此你可能会至少 5 分钟才保存一次 RDB 文件。 在这种情况下， 一旦发生故障停机， 你就可能会丢失好几分钟的数据。

每次保存 RDB 的时候，Redis 都要 fork() 出一个子进程，并由子进程来进行实际的持久化工作。 在数据集比较庞大时， fork() 可能会非常耗时，造成服务器在某某毫秒内停止处理客户端； 如果数据集非常巨大，并且 CPU 时间非常紧张的话，那么这种停止时间甚至可能会长达整整一秒。 虽然 AOF 重写也需要进行 fork() ，但无论 AOF 重写的执行间隔有多长，数据的耐久性都不会有任何损失。

AOF的优缺点

aof的优点

1.数据保证：我们可以设置fsync策略，一般默认是everysec，也可以设置每次写入追加，所以即使服务死掉了，咱们也最多丢失一秒数据

2.自动缩小：当aof文件大小到达一定程度的时候，后台会自动的去执行aof重写，此过程不会影响主进程，重写完成后，新的写入将会写到新的aof中，旧的就会被删除掉。但是此条如果拿出来对比rdb的话还是没有必要算成优点，只是官网显示成优点而已。

缺点呢：和rdb相反嘛，毕竟只有两种。

1.性能相对较差：它的操作模式决定了它会对redis的性能有所损耗

2.体积相对更大：尽管是将aof文件重写了，但是毕竟是操作过程和操作结果仍然有很大的差别，体积也毋庸置疑的更大。

3.恢复速度更慢：

redis有两种持久化方式，aof和rdb，aof相当于日志记录操作命令，rdb相当于数据的快照。安全性来讲由于aof的记录能够精确到秒级追加甚至逐条追加，而rdb只能是全量复制，aof明显高于rdb。但是从性能来讲rdb就略胜一筹，rdb是redis性能最大化的体现，它不用每秒监控是否有数据写入，当达到触发条件后就自动fork一个子进程进行全量更新，速度也很快。容灾回复方面rdb更是能够快速的恢复数据，而aof需要读取再写入，相对慢了很多。

#### Solrj jar包的使用方法

1. 创建solr对象

SolrServer solrServer = new HttpSolrServer(“http:192.168.20.30/solr”);

2 创建文档对象

SolrInputDocument document = new SolrInputDocument()

Document.addField(“id”,”test001”);

Document.addField(“fieldType1”,”xxxx”);

3把文档写入索引库

solrServer.add(document);

4.提交

solrServer.commit();

如果是集群使用solrcloud这个实现类

SolrServer solrServer = new CloudSolrServer(“http….”);

#### 全文检索实现：

创建一个solr的服务模块war工程，先是创建一个获取数据库所有查询数据的接口，获取全部商品数据，然后再写一个根据查询条件在solr库中查询商品数据的接口

为了灵活的进行分布式部署，需要创建搜索的服务工程：

搭建solr服务层，使用portal门户调用搜索服务，实现商品搜索

在java程序中使用solrJ客户端连接solr服务，进行各种操作

首先使用java程序读取mysql数据库中的商品信息，然后创建solr文档对象，把商品信息写入到solr索引库中，以便进行后面的搜索操作。

然后写一个接口：控制层传入参数：搜索内容，页面，每页显示多少条数据，调用业务层方法，返回一个pojo类，里面包含查询到的总记录，总记录条数，总页数和当前页数。

业务层获取到控制层的参数后，调用dao层获取搜索到的数据，封装成pojo类，返回给控制层

Dao层完全就是根据查询条件进行查询，然后返回所有查询到的数据，返回就行

因为我是java开发，所以不怎么关心表设计的优化，我只关心怎么使用sql语句优化，

#### Sql查询优化：

避免全表查询的方法：

1. 建立索引列，让where子句和order by 后面的筛选条件尽量走索引；
2. 避免在where子句后面对字段进行null值判断，这将会导致引擎放弃索引查询而走全表扫描，也就是说不要在存在null值的列上面建立索引。
3. 避免在where 子句后面使用!= 或者<> 这样的字符，会导致引擎不走索引而去全表扫描
4. 避免使用like关键字进行模糊查询，也会导致引擎不走索引而走全表扫描
5. 尽量避免在where子句后面进行表达式操作，比如select id from t where num/2=100
6. 避免在where子句后面对字段进行函数操作，这样不光会进行全表扫描，还需要进行函数计算，速度更慢，比如：select id from t where substring(name,1,3)='abc'

其他优化：

1、查询数据时，尽量查询需要的字段，不返回不必要的字段，降低查询结果的大小，增加传输效率。

2、需要用到外连接查询时，不要使用子查询方式，尽量使用左连接、右连接查询方式，减少查询次数，提高效率。

3、多表查询时（内连接查询），使用inner join方式，不用在from查询多张表的方式。

4、左、右连接查询是以join左边、右边的表为主，所以在查询的时候，尽量吧数据量小的表放在该位置，减少查询量

5、左连接、右连接的on 后面的条件，先于where后面的条件执行，所以尽量提高on后面的筛选条件，减少查询数来的数据量，再进行where筛选的时候，更快

6、避免大事务操作，大事务操作需要等待的时间长，占用的资源多。

建表优化：

不要在包含多组重复数据的列上面建立所以，因为重复数据多了，索引查询效率就低了

Sql查询语句的执行顺序：

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

1、从from子句获取数据，生产虚拟表tb1；

2、应用on子句后面的条件，筛选生成虚拟表tb2；

3、根据join后的表，添加字段到tb2，生成tb3；

4、经过where子句筛选，生成虚拟表tb4；

5、通过group by子句对tb4进行分组，分组后得到的表tb5，只为分组条件列里面每一个不同的值生成一行数据，所以后面的步骤得到的只能是tb5中的列或者聚合函数（count，sum，avg）生成的列；

6、通过having子句进行筛选，得到tb6；

7、通过select子句把需要的列返回出来，生成tb7；

8、通过distinct，筛选出tb7里面相同的行，得到tb8；

9,、应用order by子句排序，排序生成一个游标，而不是虚拟表；

Nginx 反向代理

主要用在请求静态资源的时候使用，比如浏览器在请求一个页面的时候，请求直接到Nginx服务器上面，然后Nginx直接转发到配置的静态资源服务器上，获取到静态资源，然后返回给浏览器

Nginx负载均衡：

当服务器收到的请求太多的时候，服务器忙不过来就会导致单台服务器反应慢，甚至宕机，这时候就需要用到集群了，而nginx就是用来平衡多台服务器的，降低单台服务器的压力，或者在高性能服务器上处理更多的访问，在低性能的服务器上处理烧一些的请求。

订单模块有哪些接口：创建订单，根据ID查询订单，分页查询订单，修改订单状态

单点登录接口：检查数据的方法，注册，登录，根据token查询用户数据，退出。

龙华新区腾龙路淘金地电子商务孵化大厦5楼

**自学项目名称：GoShoopping项目**

**开发环境：**JDK 1.7+Eclipse+Tomcat+Maven +svn

**软件架构**：SSM + Dubbo + MySQL + Redis + Nginx +solr

**项目描述**：互联网电商类项目.

**技术描述：**

1.项目使用前后端分离的方式进行开发，后端采用分布式开发，利用Maven聚合、依赖和继承，形成多个子项目，方便团队分开进行开发、调试和部署。

2.项目分成后台管理模块、门户模块、底层服务模块、搜索服务模块、单点登录服务模块、订单服务模块这些模块，模块之间使用zookeeper配置互相访问数据，nginx进行反向代理和负载均衡。

3.使用Redis分片集群来缓存访问量大的请求数据，比如主页页面信息，商品详情信息，降低数据库压力，配置一主一从服务器，提高可用性。

4.使用solr集群处理搜索服务。

5.使用SSO单点登录服务模块处理用户的注册和登录，利用Cookie和Redis缓存实现一次登录多次使用，解决用户访问前台，购物车、订单等模块时需重复登录问题。

6.使用Redis incr自增长特性生成唯一订单号。

**责任描述**：

1. 负责门户，单点登录 购物车和订单等模块接口开发以及测试。

负责相关接口文档的编写。

#### JVM分为哪些区，每一个区干吗的

1）方法区(method)：被所有的线程共享。方法区包含所有的类信息和静态变量。

2）堆(heap)：被所有的线程共享，存放对象实例以及数组，Java堆是GC的主要区域。

3）栈(stack)：每个线程包含一个栈区，栈中保存一些局部变量等。

4）程序计数器：是当前线程执行的字节码的行指示器。

#### 内存溢出和内存泄漏

内存溢出是申请的内存大于实际存在的内存的时候，就会报out of memory 内存溢出；

内存泄漏是创建对象时，使用后没有释放使用的资源，那么GC也不会去回收这一块使用的资源，那么这一块被占用的内存就相当于泄漏了，如果这样的操作进行的次数多了，就会导致可用的资源越来越少，最后导致内存溢出

分析内存溢出，内存泄漏的原因：

 1、检查对数据库查询中，是否有一次获得全部数据的查询。一般来说，如果一次取十万条记录到内存，就可能引起内存溢出。这个问题比较隐蔽，在上线前，数据库中数据较少，不容易出问题，上线后，数据库中数据多了，一次查询就有可能引起内存溢出。因此对于数据库查询尽量采用分页的方式查询。

2、检查代码中是否有死循环或递归调用。

3、检查是否有大循环重复产生新对象实体。

4、检查List、MAP等集合对象是否有使用完后，未清除的问题。List、MAP等集合对象会始终存有对对象的引用，使得这些对象不能被GC回收。

#### Java中栈和堆的区别

JVM 中堆和栈属于不同的内存区域，使用目的也不同。

栈常用于保存方法或变量的引用和局部变量，通常分别的内存小于堆，不会被多个线程共享

堆是被整个 JVM 的所有线程共享，存放对象实例以及数组，Java堆是GC的主要区域。

#### 同步异步阻塞和非阻塞

#### 实现线程的几种方法

(1)继承Thread类，重写run函数

(2)实现Runnable接口，重写run函数

(3)实现Callable接口，重写call函数

#### 数据库表设计三范式

第一范式：1NF是对属性的原子性约束，要求属性具有原子性，不可再分解；

第二范式：2NF是对记录的惟一性约束，要求记录有惟一标识，即实体的惟一性；

第三范式：3NF是对字段冗余性的约束，即任何字段不能由其他字段派生出来，它要求字段没有冗余。

原子性：比如在一个管理订单的表中：订单号，收件人姓名，电话，收件地址 这样的表中，需要根据地址进行分类管理订单的话，就需要把订单中的地址分开成 省、市、详细地址这样的列，以便于分类管理。

唯一性例子：还是订单的表：订单号，商品信息，收件人，收件人电话，收件人地址，这样的表中，当一个订单里面有多个商品的时候，就会产生数据重复，所以需要把这样的表给分开成多张表分别存储收件人信息：收件人编号，收件人姓名、电话、地址等，然后商品也有一张表，另外建一张表存储订单信息：订单号，商品编号，收件人编号。

冗余性例子：还是比如一张订单表：订单号，商品编号，商品数量，单价，总价这样的表，因为由单价和数量就能计算出总计，这样这个总价的列就没有必要了，这一列的数据就是冗余数据。

#### 数据库存储过程的优缺点

定义：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。

优点：

1. 运行速度快。因为存储过程在创建的时候数据库已经对其进行了解析和优化，并存储在数据库中，一旦执行，就会在内存中保留一份这个存储过程，下次调用，就直接从内存中调用了，所以速度会很快。
2. 增强安全性。提高代码安全，防止SQL注入，但是这个作用代码也能解决

缺点：

1. SQL语句本身是一种结构化的查询语句，擅长的就是查询数据而非处理业务逻辑，业务逻辑交给代码去实现会更好。
2. 更新一些带有输入输出的程序过程，会比较麻烦，调试起来也比通过IDE更麻烦。
3. 没办法应用缓存，虽然数据库有一些临时表可以进行缓存，但是这样也加重的数据库的负担，而且并发严重的时候，还要加锁，效率更加低下。

总结：

1. 适当的使用存储过程能提高查询性能，但是不应该大规模的使用和滥用
2. 随着大量的缓存工具的出现，存储过程的优势不明显，使用Redis和memory cache 把数据缓存到内存，读取速度一样很快。
3. SQL本身是一种结构化查询语言，我们不应该用存储过程处理复杂的业务逻辑

#### 数据库触发器

#### REST ful架构风格

REST：Representational State Transfer，翻译的话就是“表现层状态转化”

REST的名称"表现层状态转化"中，省略了主语。"表现层"其实指的是"资源"（Resources）的"表现层"。

**一、资源（Resources）**

所谓"资源"，就是网络上的一个实体，或者说是网络上的一个具体信息。它可以是一段文本、一张图片、一首歌曲、一种服务，总之就是一个具体的实在。你可以用一个URI（统一资源定位符）指向它，每种资源对应一个特定的URI。要获取这个资源，访问它的URI就可以，因此URI就成了每一个资源的地址或独一无二的识别符。

所谓"上网"，就是与互联网上一系列的"资源"互动，调用它的URI。

**二、表现层（Representation）**

"资源"是一种信息实体，它可以有多种外在表现形式。**我们把"资源"具体呈现出来的形式，叫做它的"表现层"（Representation）。**

比如，文本可以用txt格式表现，也可以用HTML格式、XML格式、JSON格式表现，甚至可以采用二进制格式；图片可以用JPG格式表现，也可以用PNG格式表现。

URI只代表资源的实体，不代表它的形式。严格地说，有些网址最后的".html"后缀名是不必要的，因为这个后缀名表示格式，属于"表现层"范畴，而URI应该只代表"资源"的位置。它的具体表现形式，应该在HTTP请求的头信息中用Accept和Content-Type字段指定，这两个字段才是对"表现层"的描述。

**三、状态转化（State Transfer）**

访问一个网站，就代表了客户端和服务器的一个互动过程。在这个过程中，势必涉及到数据和状态的变化。

互联网通信协议HTTP协议，是一个无状态协议。这意味着，所有的状态都保存在服务器端。因此，**如果客户端想要操作服务器，必须通过某种手段，让服务器端发生"状态转化"（State Transfer）。而这种转化是建立在表现层之上的，所以就是"表现层状态转化"。**

客户端用到的手段，只能是HTTP协议。具体来说，就是HTTP协议里面，四个表示操作方式的动词：GET、POST、PUT、DELETE。它们分别对应四种基本操作：**GET用来获取资源，POST用来新建资源（也可以用于更新资源），PUT用来更新资源，DELETE用来删除资源。**

**六、综述**

综合上面的解释，我们总结一下什么是RESTful架构：

　　（1）每一个URI代表一种资源；

　　（2）客户端和服务器之间，传递这种资源的某种表现层；

　　（3）客户端通过四个HTTP动词，对服务器端资源进行操作，实现"表现层状态转化"。

#### 什么是SpringMVC框架：

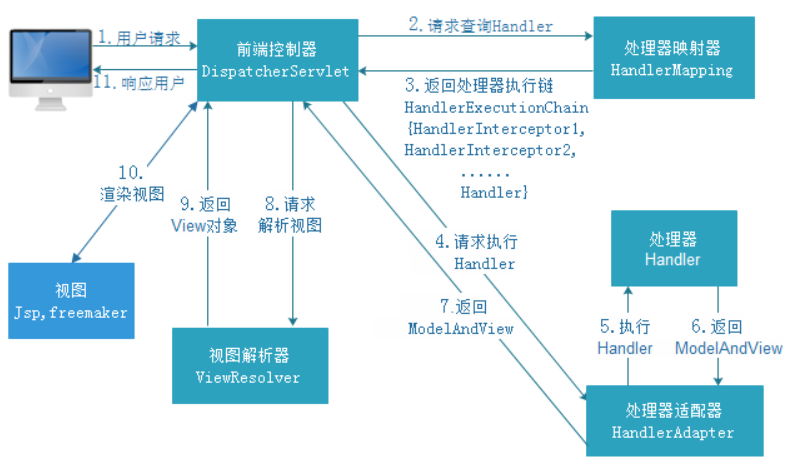
SpringMVC框架是基于java实现的MVC设计模式的请求驱动类型的轻量级WEB框架，通过把Model、View、Controller进行分离，将WEB层进行职责解耦，把复杂的web应用分成逻辑清晰的几部分，简化开发，减少出错。

WEB层的框架

#### Spring MVC 框架的组成（五大组件）：

1. 前端控制器DispatcherServlet:接受请求，响应结果，相当与转发器；
2. 处理器映射器HandlerMapping:根据请求的url来查找对应的handler处理请求；
3. 处理器适配器HandlerAdapter:调用handler的方法去执行具体逻辑；
4. 处理器Handler:执行方法，并返回ModelAndView对象；
5. 视图解析器View Resolver:进行视图解析，根据逻辑视图名解析成真正的视图（JSP）

流程图：



#### SpringMVC怎样设定转发和重定向：

（1）转发：在返回值前面加"forward:"，譬如"forward:user.do?name=method4"

（2）重定向：在返回值前面加"redirect:"，譬如"redirect:http://www.baidu.com"

#### 如何解决POST请求和GET请求中文乱码：

POST请求在WEB-XML文件中加一个CharacterEncodingFilter过滤器，并设置值为utf-8；

GET请求，在TOMCAT配置文件添加编码与工程编码一致，修改编码为UTF-8；

#### SpringMVC控制器是否是单例模式，会导致什么问题：

是单例模式，所以当多线程调用的时候，会存在线程安全的问题，所以不要在控制器里面写字段，也不能用同步方法，不然会影响性能。；

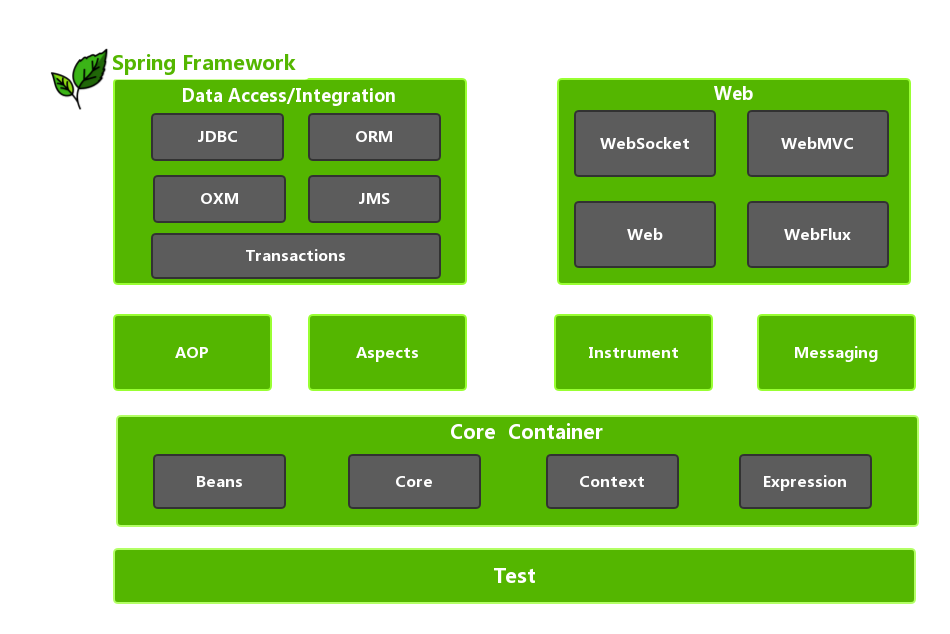
#### SpringMvc里面拦截器是怎么写的：

有两种写法,一种是实现HandlerInterceptor接口，另外一种是继承适配器类，接着在接口方法当中，实现处理逻辑；然后在SpringMvc的配置文件中配置拦截器即可：

|  |
| --- |
| <!-- 配置SpringMvc的拦截器 -->  <mvc:interceptors>  <!-- 配置一个拦截器的Bean就可以了 默认是对所有请求都拦截 -->  <bean id="myInterceptor" class="com.zwp.action.MyHandlerInterceptor"></bean>  <!-- 只针对部分请求拦截 -->  <mvc:interceptor>  <mvc:mapping path="/modelMap.do" />  <bean class="com.zwp.action.MyHandlerInterceptorAdapter" />  </mvc:interceptor>  </mvc:interceptors> |

### Spring框架

Spring框架模块：



#### 什么是spring

Spring是java企业级开发的开源框架，主要用来进行java开发，但是有些扩展是针对构建j2ee平台的web应用。Spring框架的目标是简化java企业级应用开发，核心是IOC（控制反转）和AOP（面向切面）

#### 使用spring框架的好处：

轻量：Spring是轻量的，基本版本大约2M；

控制反转：Spring通过控制反转，把对象创建的工作交给容器来完成，减少开发过程，另外就是实现对象之间的松耦合（举例：以前想要调用某一个类的方法，必须new出来这个类，然后调用，这样这两个类之间就有了耦合，使用了spring之后，创建对象的工作不需要手动做了，需要调用某个类的方法时，直接通过spring注入进来，用就好了，也减轻了两个类的耦合关系。）

面向切面的编程：Spring支持面向切面的编程

容器：Spring 包含并管理应用中对象的生命周期和配置。

MVC框架：Spring的WEB框架是个精心设计的框架，是Web框架的一个很好的替代品。

事务管理：Spring 提供一个持续的事务管理接口，可以扩展到上至本地事务下至全局事务（JTA）。

异常处理：Spring 提供方便的API把具体技术相关的异常（比如由JDBC，Hibernate or JDO抛出的）转化为一致的unchecked 异常。

### Redis

缓存的处理流程一般是请求到达后台，后台先去缓存中获取数据，获取到直接返回，获取不到，就去数据库中查询数据，如果数据库中也没有，就返回空结果。

#### 缓存穿透、缓存击穿和缓存雪崩

缓存穿透就是恶意用户不断请求缓存中没有的数据，这样请求会绕过缓存直接落到数据库上面，这样会给数据库带来过大压力

缓存击穿是指缓存中没有，但是数据库中有的数据(一般是缓存时间到期)，这时由于并发用户请求特别多，同时读取缓存没有读取到，就会同时去访问数据库进行查询引起数据库瞬时压力过大。

缓存雪崩是指缓存中大批量数据在短时间内过期，而查询数据量过大，引起数据路压力过大甚至宕机。

缓存击穿和缓存雪崩的区别是缓存击穿指并发查一条数据，缓存雪崩指的是不同数据同时过期，很多数据查询不到，从而去数据库去查

解决方法：

缓存穿透方法是，当查询到值为空的key是，把这个key缓存到Redis中去，值为null，设置一个很短的过期时间，这样短时间内相同的key再次访问的时候，就不会再去访问数据库了，而是直接返回null，然后等到key过期，删除这个key。

缓存击穿和缓存雪崩的解决方法是设置缓存数据永不过期。

建议书精算系统就是采用的上面两种方式。

### 面试项目描述

#### 精算系统描述

该系统是用来给平安寿险制作建议书展示使用，因为是面向用户的，所以跟正常的投保系统的要求一致，都需要做到很精确，所以里面的所有涉及到计算的算法，都是转化成BigDecimal类型进行计算后，在转化成int类型或者double类型。因为涉及到的业务很多，用到的表也多，所以需要做缓存优化，本系统采用两种缓存方式，一种jvm缓存，一种Redis缓存，jvm缓存缓存一些算法数据，因为这些算法数据不会变动，只有每次版本更新的时候，重启服务的时候才会更新，另外就是Redis缓存，本系统大量采用Redis缓存，包括少量的记录险种和开售信息的表，以及数据量非常大的记录费率和现价的表。

另外为了防止Redis缓存雪崩，专门有一个微服务管理数据的缓存、失效和比对，在服务重启完成后几分钟后，把Redis里面缓存的表全部失效掉，为了方式缓存雪崩，把需要缓存的表记录到枚举里面去，按照枚举顺序乘以10秒作为每个表的失效时间，等失效完成后，开始全量加载数据到Redis中，另外还做有部分加载数据，不过这个功能默认关闭着，不开。另外做了一个对账功能，每天凌晨2点系统不繁忙的时候开始Redis对账，比对Redis和数据库中的数据，如果Redis中有的数据，数据库中没有，就认为这个是无效数据，把这个比对不通过的记录记到专门的数据库的表中，然后发送邮件给负责人，进行处理。

刚开始说了有三个微服务，一个微服务专门用来管理数据的加载和维护，另外两个微服务对接多个渠道，提供接口服务，接口会被注册到网关，然后前段就通过网关调用调用我们的接口。

敏捷开发过程：一个版本迭代，一般都是两周开发时间，PO一般会把需求按照开发量分成两个迭代开发。当一个新版本过来的时候，都是由组内PO（Product Owner 即熟悉该产品所有业务相关的逻辑、流程、设置等方面事宜的人员，一般可由产品经理担任，也可由熟悉业务的开发人员担任）组织大家开需求梳理会，给大家讲解需求的背景，以及需求实现过程中需要注意的问题，然后根据经验讨论每个需求的开发量，把所有的工作量分成两个迭代进行开发。需求梳理会完成后，开发开始进行第一个迭代的开发，测试开始按照需求设计案例，两者同步进行，当业务开发完成，自测完成，就要开始跟前端进行开发环境联调，联调通过就可以移交代码进行测试了。测试在设计完案例后，由测试负责人组织会议进行案例评审，评审案例设计中，由开发分析可能遗漏没有测试到的地方，由测试修改。当测试人员开始进行测试的时候，开发人员就开始进行下个迭代的开发，并支持测试的上个迭代。

要完成这么紧密的开发和迭代，就需要开发对开发质量进行保证，保证在开发完成后BUG不多，另外就是测试需要实现自动化测试，不然全部手动化测试，会跟不上开发节奏，导致天天加班还赶不上进度。

进入到平安两个多月适应了他们那边的开发进度后，正好赶上平安集团进行微服务改造的浪潮，我们系统也需要进行微服务的改造，经过两个多月的工作，我们组长也认可了我的能力，让我参与到了微服务系统的搭建。我们组里面6个开发，我们三个那段时间主要进行微服务系统框架的搭建，另外的三个开发就进行业务开发。进过我们开会讨论嘛，决定把整个系统分成common模块，service模块，mapper模块，aop模块，微服务模块，后来加进来一个单元测试的模块，manager模块，三个微服务两个专门对外提供接口，一个用来更新数据。这样框架就算这样决定了，开始开发吧，整体框架由我们组长完成，包括需要依赖的jar包什么的，因为是整体的升级，平安那边依赖的jar包都是集团规定版本号。框架搭建起来后连接配置中心，这个工作也是我们组长做的，这样大体框架弄起来后，我们就开始迁移代码，按照之前讨论好的模块开始迁移，我主要迁移的是加载数据的那一块。另外一个同事迁移业务那一块，，组长负责协调分配任务。一些公共的工具包，就直接迁移就玩了，基本没什么变动，加载数据那一块，有加载一些重要的算法数据到mem cache缓存，大量的业务数据存到redis，

另外设计的把Redis清理和Redis全量加载和Redis比对的功能放到manager微服务，这样重启这个微服务的时候，会让数据过期，然后全量加载进去，另外有一个比对的功能，这些弄完了之后就自测下迁移功能没什么问题就行，比对Redis和数据库的数据。另外就是一些pojo的类工具类还有xml文件。

等到这些代码迁移完成，接口前已完成后，整体前已完成后，就开始测试系统的稳定性，

专门做一个工具去把生产上面的调用数据给同步到测试环境，然后分别去调用测试环境新系统和老系统接口，比对返回的结果，把比对不通过的结果记录下来分析。就这样在持续的改进和更新一个月后，比对结果就大部分一致了，偶尔的不同也是字符串的顺序不一致的。然后把我们的接口微服务和接口全部注册到注册中心，到了注册中心后，我们就开始通过网关调用接口。然后再去比对，这其中就发现很多比对不通过的，仔细检查是网关那边设置的等待时间不够，我们有一个接口运行时间差不多都是5s，有事慢的时候就是10多秒，这些都完成了后，就开始上生产进行试运行，在这个过程中我们开发了一个线上比对的模块，就是做一个拦截器，在每一个接口被调用的时候，

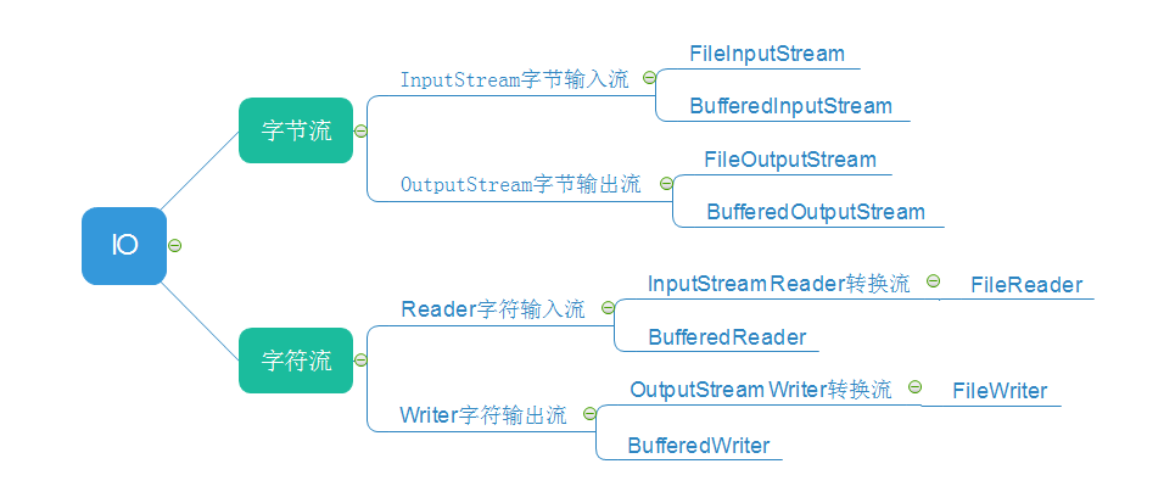
并行开发的时候，新系统的每一个接口都进行拦截，拦截到之后转发到老系统上面去。等到这一套过程稳定后，就开始按照调用渠道去切换到新系统。先那拿一个调用量少的渠道的险种做实验。当然这个切换是有做配置的，支持热刷新，一旦出现问题，马上让运维那边修改配置切换到转发模式。

当然在这个切换的时候，我们做的一个线上比对功能，就是当别的渠道调用接口的时候，我们会进过新系统进行转发，然后拿到入参调用我们新系统，对结果进行比较。比较结果记录到表中。

#### 冒泡排序：

|  |
| --- |
| **private** **static** **void** sort(**char**[] ch) {  **int** len = ch.length;  **char** temp;  **for**(**int** i=0;i<len;i++){  **for**(**int** j=i+1;j<len;j++){  **if**(ch[i]>ch[j]){  temp=ch[i];  ch[i]=ch[j];  ch[j]=temp;  }  }  System.***out***.println(ch);  } |

#### 流操作



锁，

HashMap 和HashSet

HashTable底层