JAVA1.7新特性

1.对集合类的语言支持

2.自动资源管理

3.改进的通用实力创建类型判断

4.数字字面量下划线支持

5.swtich支持string类型

6.二进制字面量

7.简化可变参数方法调用

1. how can we inject denpendencies by setter?

使用配置的方式，注入controller，并将需要注入的属性作为property写在controller标签中

## Temp：

sql性能检测

索引的原理

自动化测试selenium

批量执行sql，事务是如何提交的

RxJava。webFlux

1. 如何保证单例：

格式: [秒] [分] [小时] [日] [月] [周] [年] 如果定时任务涉及到了那一年，则可以写，如果没有涉及到可以不写  
1 秒 是 0-59 , - \* /   
2 分 是 0-59 , - \* /   
3 小时 是 0-23 , - \* /   
4 日 是 1-31 , - \* ? / L W   
5 月 是 1-12 or JAN-DEC , - \* /   
6 周 是 1-7 or SUN-SAT , - \* ? / L #   
7 年 否 empty 或 1970-2099 , - \* /   
  
通配符说明:   
\* 表示所有值. 例如:在分的字段上设置 "\*",表示每一分钟都会触发。   
? 表示不指定值。使用的场景为不需要关心当前设置这个字段的值。例如:要在每月的10号触发一个操作，但不关心是周几，所以需要周位置的那个字段设置为"?" 具体设置为 0 0 0 10 \* ?   
- 表示区间。例如 在小时上设置 "10-12",表示 10,11,12点都会触发。   
, 表示指定多个值，例如在周字段上设置 "MON,WED,FRI" 表示周一，周三和周五触发   
/ 用于递增触发。如在秒上面设置"5/15" 表示从5秒开始，每增15秒触发(5,20,35,50)。在月字段上设置'1/3'所示每月1号开始，每隔三天触发一次。   
L 表示最后的意思。在日字段设置上，表示当月的最后一天(依据当前月份，如果是二月还会依据是否是润年[leap]), 在周字段上表示星期六，相当于"7"或"SAT"。如果在"L"前加上数字，则表示该数据的最后一个。例如在周字段上设置"6L"这样的格式,则表示“本月最后一个星期五"   
W 表示离指定日期的最近那个工作日(周一至周五). 例如在日字段上设置"15W"，表示离每月15号最近的那个工作日触发。如果15号正好是周六，则找最近的周五(14号)触发, 如果15号是周未，则找最近的下周一(16号)触发.如果15号正好在工作日(周一至周五)，则就在该天触发。如果指定格式为 "1W",它则表示每月1号往后最近的工作日触发。如果1号正是周六，则将在3号下周一触发。(注，"W"前只能设置具体的数字,不允许区间"-").   
  
小提示   
'L'和 'W'可以一组合使用。如果在日字段上设置"LW",则表示在本月的最后一个工作日触发(一般指发工资 )   
  
# 序号(表示每月的第几个周几)，例如在周字段上设置"6#3"表示在每月的第三个周六.注意如果指定"#5",正好第五周没有周六，则不会触发该配置(用在母亲节和父亲节再合适不过了)   
  
小提示   
周字段的设置，若使用英文字母是不区分大小写的 MON 与mon相同.   
   
常用示例:   
0 0 12 \* \* ? 每天12点触发   
0 15 10 ? \* \* 每天10点15分触发   
0 15 10 \* \* ? 每天10点15分触发   
0 15 10 \* \* ? \* 每天10点15分触发   
0 15 10 \* \* ? 2005 2005年每天10点15分触发   
0 \* 14 \* \* ? 每天下午的 2点到2点59分每分触发   
0 0/5 14 \* \* ? 每天下午的 2点到2点59分(整点开始，每隔5分触发)   
0 0/5 14,18 \* \* ? 每天下午的 2点到2点59分(整点开始，每隔5分触发)   
每天下午的 18点到18点59分(整点开始，每隔5分触发)   
  
0 0-5 14 \* \* ? 每天下午的 2点到2点05分每分触发   
0 10,44 14 ? 3 WED 3月分每周三下午的 2点10分和2点44分触发 （特殊情况，在一个时间设置里，执行两次或 两次以上的情况）   
0 59 2 ? \* FRI 每周5凌晨2点59分触发；   
0 15 10 ? \* MON-FRI 从周一到周五每天上午的10点15分触发   
0 15 10 15 \* ? 每月15号上午10点15分触发   
0 15 10 L \* ? 每月最后一天的10点15分触发   
0 15 10 ? \* 6L 每月最后一周的星期五的10点15分触发   
0 15 10 ? \* 6L 2002-2005 从2002年到2005年每月最后一周的星期五的10点15分触发   
0 15 10 ? \* 6#3 每月的第三周的星期五开始触发   
0 0 12 1/5 \* ? 每月的第一个中午开始每隔5天触发一次   
0 11 11 11 11 ? 每年的11月11号 11点11分触发(光棍节)

线程的终结钩子怎么获得？

keytool -list -rfc -keystore pcccmall.jks -storepass 123456

/\*private static String private\_key\_path = "";//私钥路径  
private static String public\_key\_path = "";//公钥路径  
private static String store\_password = "";//验证完整性密码  
private static String alias = "";//私钥别名  
private static String key\_password = "";//私钥密码  
private static String signatureAlgorithm = "MD5withRSA";//签名算法  
static{  
 Security.addProvider(new BouncyCastleProvider());  
  
 final String SYS\_CONFIG = "/configure.properties";  
 InputStream \_in = MyOrderController.class.getResourceAsStream(SYS\_CONFIG);  
 Properties properties = new Properties();  
 try{  
 properties.load(\_in);  
 private\_key\_path = properties.getProperty("cafile");  
 public\_key\_path = properties.getProperty("public\_key\_path");  
 store\_password = properties.getProperty("store\_password");  
 alias = properties.getProperty("alias\_key");  
 key\_password = properties.getProperty("key\_password");  
 }catch (Exception e){  
 //*todo*  
 e.printStackTrace();  
 }finally {  
 try{  
 \_in.close();  
 }catch (Exception e){  
 //*todo*  
 e.printStackTrace();  
 }  
 }  
}  
  
public static void main(String[] args) throws Exception {  
 \*//\*String orgData = "123";//待验数据  
 Signature dsa = Signature.getInstance(signatureAlgorithm);//算法实现类  
  
 InputStream privateKeyInputStream = new FileInputStream(private\_key\_path);  
 KeyStore keyStore = KeyStore.getInstance("JKS");  
 keyStore.load(privateKeyInputStream, store\_password.toCharArray());  
  
 PrivateKey privateKey = (PrivateKey)keyStore.getKey(alias,key\_password.toCharArray());  
 dsa.initSign(privateKey);  
 dsa.update(orgData.getBytes());  
 byte[] \_sig = dsa.sign();//生成的数字签名  
 String sign = ByteArrayUtil.toHexString(\_sig);//byte数组转String  
  
 //通过输入流获得Certificate，进而获得publicKey  
 InputStream publicKeyInputStream = new FileInputStream(public\_key\_path);  
 CertificateFactory cf = CertificateFactory.getInstance("X.509");  
 Certificate certificate = cf.generateCertificate(publicKeyInputStream);  
  
 PublicKey publicKey = certificate.getPublicKey();  
 dsa.initVerify(publicKey);  
 dsa.update(orgData.getBytes());  
  
 byte sig[] =ByteArrayUtil.hexStringToByteArray(sign);//String转btye数组  
 System.out.println(dsa.verify(sig));//验证数字签名\*//\*  
  
 String data = "{\"icorderid\":\"180412000005\",\"premium\":\"81600\",\"icno\":\"ECP\",\"tpi\":\"30117826\",\"icvendorid\":\"301001900003781\",\"prodid\":\"A088\",\"prodname\":\"安联臻爱医疗保险06\",\"enddat\":\"20190412\",\"insuamt\":\"4200000\",\"payprd\":\"0000\",\"payfrq\":\"00\",\"effdat\":\"20180413\",\"planlevel\":\"1\",\"jumpflag\":\"00\",\"ogiorderid\":\"\",\"memberId\":\"MB20003839\",\"holdername\":\"\",\"holderidno\":\"\"}";  
 String salt = "bxsdallianz@17st";  
 Md5PasswordEncoder m = new Md5PasswordEncoder();  
 String sign = m.encodePassword(data+salt,null);  
 System.out.println(System.getProperties());  
 System.out.println(sign);  
}\*/

RuntimeException

Error

匿名内部类和内部类编译上有什么区别

Bean scope

@ContextConfiguration(locations = "file:/src/main/webapp/WEB-INF/applicationContext.xml")

## 基础

### 日期处理：

Date date = new Date();  
String now = FastDateFormat.*getInstance*("yyyy-MM-dd").format(date);  
System.*out*.println(now);  
return now;

### 中文乱码：

前台传后台中文处理

后台

String inputVocabularies = request.getParameter("vocabularies");

String vocabularies="";

try {

vocabularies= URLDecoder.decode(inputVocabularies,"UTF-8");

前台：

data:"vocabularies="+encodeURI(encodeURI(strVocabulary)),

或者data:{"commentId":commentId,"commentReply":encodeURI(commentReply)},

继承的作用：

对父类的具体化和拓展

重定向和转发的区别

<http://www.cnblogs.com/shenliang123/archive/2011/10/27/2226892.html>

（1）RequestDispatcher.forward 方法只能将请求转发给同一个WEB应用中的组件；而HttpServletResponse.sendRedirect 方法不仅可以重定向到当前应用程序中的其他资源，还可以重定向到同一个站点上的其他应用程序中的资源，甚至是使用绝对URL重定向到其他站点的资源。如果 传递给HttpServletResponse.sendRedirect 方法的相对URL以“/”开头，它是相对于整个WEB站点的根目录；如果创建RequestDispatcher 对象时指定的相对URL以“/”开头，它是相对于当前WEB应用程序的根目录。  
（2）调用HttpServletResponse.sendRedirect 方法重定向的访问过程结束后，浏览器地址栏中显示的URL会发生改变，由初始的URL地址变成重定向的目标URL；而调用 RequestDispatcher.forward 方法的请求转发过程结束后，浏览器地址栏保持初始的URL地址不变。  
（3）HttpServletResponse.sendRedirect 方法对浏览器的请求直接作出响应，响应的结果就是告诉浏览器去重新发出对另外一个URL的访问请求。  
举个例子：重定向过程好比有个绰号叫“浏览器”的人写信找张三借钱，张三回信说没有钱，让“浏览器”去找李四借，并将李四现在的通信地址告诉给了“浏览器 ”。于是，“浏览器”又按张三提供通信地址给李四写信借钱，李四收到信后就把钱汇给了“浏览器”。可见，“浏览器”一共发出了两封信和收到了两次回复，“ 浏览器”也知道他借到的钱出自李四之手。 RequestDispatcher.forward 方法在服务器端内部将请求转发给另外一个资源，浏览器只知道发出了请求并得到了响应结果，并不知道在服务器程序内部发生了转发行为。这个过程好比绰号叫“ 浏览器”的人写信找张三借钱，张三没有钱，于是张三找李四借了一些钱，甚至还可以加上自己的一些钱，然后再将这些钱汇给了“浏览器”。可见，“浏览器”只 发出了一封信和收到了一次回复，他只知道从张三那里借到了钱，并不知道有一部分钱出自李四之手。  
（4）RequestDispatcher.forward 方法的调用者与被调用者之间共享相同的request 对象和response 对象，它们属于同一个访问请求和响应过程；而HttpServletResponse.sendRedirect 方法调用者与被调用者使用各自的request 对象和response 对象，它们属于两个独立的访问请求和响应过程。  
对于同一个WEB应用程序的内部资源之间的跳转，特别是跳转之前要对请求进行一些前期预处理，并要使用 HttpServletRequest.setAttribute 方法传递预处理结果，那就应该使用RequestDispatcher.forward 方法。  
不同WEB应用程序之间的重定向，特别是要重定向到另外一个WEB站点上的资源的情况，都应该使HttpServletResponse.sendRedirect 方法。  
（5）无论是RequestDispatcher.forward 方法，还是HttpServletResponse.sendRedirect 方法，在调用它们之前，都不能有内容已经被实际输出到了客户端。如果缓冲区中已经有了一些内容，这些内容将被从缓冲区中清除。

serialVersionUID为什么要static final

防止这个变了，日后反序列化出错

---------------------------------------------------------------------------------------------------

## JVM

JAVA运行时数据区

方法区

线程共享

用来存放已被虚拟机加载的类信息，常量，静态变量

运行时常量池：

也是方法区的一部分，存放编译器生成的字面量和符号。运行时常量池并非是在编译器就固定死了，可以动态向池中添加数据，如String类的intern方法。当运行时常量池内存超出了方法区内存时，则会抛出OutOfMemory

堆

线程共享。唯一的作用就是存放对象实例

堆，从内存回收的角度来看可以分为：新生代，老年代

只要求内存空间逻辑上连续即可。当java堆中没有足够的内存完成实例的分配，并且也无法再拓展时，则会抛出OutOfMemory异常

虚拟机栈 JVM Stack

线程私有，生命周期与线程相同。Java在执行方法时，会创建一个栈帧，用来存放局部变量表，操作数栈，动态链接，方法出口等信息。每一个方法的执行，就对应着该栈帧在虚拟机栈中从入栈到出栈的过程。

其中局部变量表就包括编译期可知的基本数据类型和对象引用类型。

若线程请求的栈深度大于虚拟机栈允许的深度，则会抛出stackOverFlow异常。如果虚拟机栈在扩展时无法申请到足够的内存，则会抛出OutOfMemory异常。

本地方法栈

与虚拟机栈相同，只不过执行的是native方法

程序计数器

线程私有，用来指向当前线程正在执行的字节码行号。如果当前线程正在执行的是java方法，则存的是正在执行的虚拟机字节码指令的地址；如果正在执行的是native方法，则存的是空

## 集合

List原有10条数据，在第4个位置插入

ArrayList根据坐标插入10w条数据：1.3秒

LinkedList：0.01-0.03

List原有10000条数据，在第5000个位置插入

LINKEDlist :1.3秒

ArrayList：1.3秒

List原有10000条数据，在第9500个位置插入

LINKEDlist :2.3秒

ArrayList：1.2秒

常见数据结构的初始大小以及是如何扩容的

ArrayList 默认初始容量为10 加载因子为1 扩容增量：原容量的 0.5倍+1

HashMap：默认初始容量为16 加载因子为0.75

HashSet：默认初始容量为16 加载因子为0.75

### Collection:

Iterable接口

提供了遍历的功能： Iterator<T> iterator();

Collection接口继承Iterable接口

jdk不提供直接实现，而是提供更具体的子接口；

非特殊目的实现了Collection（包括子接口）的类，需要实现两个构造器，一个空构造器用来创造一个空 集合，一个包含一个同类型集合参数的构造器，用来实现复制。当然这不是强制的，但java内置集合都有 此功能；

各个实现都应该有自己的同步手段

int size();

boolean isEmpty();

boolean contains(Object o);

Object[] toArray();

boolean add(E e);

boolean remove(Object o);

boolean containsAll(Collection<?> c);

boolean addAll(Collection<? extends E> c);

boolean removeAll(Collection<?> c);

boolean retainAll(Collection<?> c);

void clear();

boolean equals(Object o);

int hashCode();

AbstractCollection抽象类实现Collection接口

提供了Collection接口的骨架实现

定义一个不可变集合，只需继承该抽象类，然后实现Iterator和Size接口

定义一个可变集合，还需要额外实现add

### Set:

Set接口继承Collection接口

不存在两个元素e1,e2使得e1.equals(e2);

最多存在一个null值；

是数学上集合的抽象；

AbstractSet实现了set接口，提供了一些骨架实现

HashSet：继承AbstractSet，实现set

依赖于hash table;

本质上是一个HashMap的实例;

遍历的顺序不一定;

允许null值;

add remove contains size操作消耗的时间不变;

遍历时间与set长度+其背后的hashmap的桶的数量成比例。所以如果遍历的性能很关键的话，初始容量不能太高，负载因子也不能太低。（那么多少合适呢？）;

非线程安全的;

在并发访问的情况下，如果没有额外的同步手段，则可以使用Set s = Collections.synchronizedSet(new HashSet(...));

在iterator创建后，在并发修改的情况下，可能会抛出异常--fail-fast特性，这个特点可以用作侦测bug，但不能依赖这个特性编写代码

其实就是把set的value作为hashmap的key来存储

TreeSet

实现依赖于TreeMap

SortedSet接口继承Set接口

三种实现：HashSet LinkedHashSet TreeSet

* ****HashSet****: is the best-performing implementation and is a widely-used Set implementation. It represents the core characteristics of sets: no duplication and unordered.
* ****LinkedHashSet****: This implementation orders its elements based on insertion order. So consider using a LinkedHashSet when you want to store unique elements in order.
* ****TreeSet****: This implementation orders its elements based on their values, either by their natural ordering, or by a Comparator provided at creation time.

利用Set去重：

Set<String> names = **new** HashSet<>();

names.add("Tom");

names.add("Mary");

**if** (names.add("Peter")) {

    System.out.println("Peter is added to the set");

}

**if** (!names.add("Tom")) {

    System.out.println("Tom is already added to the set");

}

利用Set找交集：

Set<Integer> s1 = **new** HashSet<>(Arrays.asList(1, 2, 3, 4, 5, 7, 9));

Set<Integer> s2 = **new** HashSet<>(Arrays.asList(2, 4, 6, 8));

System.out.println("s1 before intersection: " + s1);

s1.retainAll(s2);

System.out.println("s1 after intersection: " + s1);

利用Set找差集：

Set<Integer> s1 = **new** HashSet<>(Arrays.asList(1, 2, 3, 4, 5, 7, 9));

Set<Integer> s2 = **new** HashSet<>(Arrays.asList(2, 4, 6, 8));

System.out.println("s1 before difference: " + s1);

s1.removeAll(s2);

System.out.println("s1 after difference: " + s1);

同步：

Set set = Collections.synchronizedSet(new HashSet())

Synchronize(set){

Iterator i = set.iteraotr();

While(i.hasNext()){

String value = i.next()

}

}

### List

<http://www.codejava.net/java-core/collections/java-list-collection-tutorial-and-examples>

有序（插入顺序）

允许null和重复值

插入其中的原始类型会自动转换为包装类型

* ****ArrayList****: An implementation that stores elements in a backing array. The array’s size will be automatically expanded if there isn’t enough room when adding new elements into the list. It’s possible to set the default size by specifying an initial capacity when creating a new ArrayList. Basically, an ArrayList offers constant time for the following operations: size, isEmpty, get, set, iterator, and listIterator; amortized constant time for the add operation; and linear time for other operations. Therefore, this implementation can be considered if we want fast, random access of the elements.
* Get()时间复杂度为O(1).add().remove()时间复杂度为O（n）
* 个体****LinkedList****: An implementation that stores elements in a doubly-linked list data structure. It offers constant time for adding and removing elements at the end of the list; and linear time for operations at other positions in the list. Therefore, we can consider using a LinkedList if fast adding and removing elements at the end of the list is required.
* Get()时间复杂度为O(n).add().remove()时间复杂度为O（1）

同步：

* List list = new ArrayList();
* List safeList = Collections.concurrentList(list);
* Iterator i = safeList.iterator();
* Synchronize(safList ){
* While(i.hasNext()){
* String value = i.next();
* }

}

Vector：线程安全，弃用

### Map

http://www.codejava.net/java-core/collections/java-map-collection-tutorial-and-examples

特点： 一种关系型容器

是数学上函数的抽象

存储键值对；

key不允许重复；

一个key最多对应一个value；

分支：

支持null key和null value：HashMap and LinkedHashMap

不支持null key和null value：TreeMap

遍历： 提供3种：keyset，values，entryset

Map<String, String> mapCountryCodes = **new** HashMap<>();

mapCountryCodes.put("1", "USA");

mapCountryCodes.put("44", "United Kingdom");

mapCountryCodes.put("33", "France");

mapCountryCodes.put("81", "Japan");

Set<String> setCodes = mapCountryCodes.keySet();

Iterator<String> iterator = setCodes.iterator();

**while** (iterator.hasNext()) {

    String code = iterator.next();

    String country = mapCountryCodes.get(code);

    System.out.println(code + " => " + country);

}

Collection<String> countries = mapCountryCodes.values();

**for** (String country : countries) {

    System.out.println(country);

}

Set<Map.Entry<String, String>> entries = mapCountryCodes.entrySet();

**for** (Map.Entry<String, String> entry : entries) {

    String code = entry.getKey();

    String country = entry.getValue();

    System.out.println(code + " => " + country);

}

备注： map的顺序就是iterators返回元素的顺序

什么时候用？： key,value间存在某种联系，希望通过key来查找或者更新value

****HashMap****: this implementation uses a hash table as the underlying data structure. It implements all of the Mapoperations and allows null values and one null key. This class is roughly equivalent to Hashtable - a legacy data structure before Java Collections Framework, but it is not synchronized and permits nulls. HashMap does not guarantee the order of its key-value elements. Therefore, consider to use a HashMap when order does not matter and nulls are acceptable.

****LinkedHashMap****: this implementation uses a hash table and a linked list as the underlying data structures, thus the order of a LinkedHashMap is predictable, with insertion-order as the default order. This implementation also allows nulls like HashMap. So consider using a LinkedHashMap when you want a Map with its key-value pairs are sorted by their insertion order.

****TreeMap****: this implementation uses a red-black tree as the underlying data structure. A TreeMap is sorted according to the natural ordering of its keys, or by a Comparator provided at creation time. This implementation does not allow nulls. So consider using a TreeMap when you want a Map sorts its key-value pairs by the natural order of the keys (e.g. alphabetic order or numeric order), or by a custom order you specify.

HashTable:线程安全，弃用

同步：

Map map = new ConcurrentHashMap()

Or

Map map = Collections.synchronizedMap(new HashMap())（给每个方法添加锁）

Set<Integer> keySet = map.keySet();

Synchronized(map){

Iterator<Integer> iterator = keySet.iterator();

While(iterator.hasNext()){

Integer key = iterator.next();

String value = map.get(key)

}

}

Difference:

ConcurrentHashMap:

You should use ConcurrentHashMap when you need very high concurrency in your project.

It is thread safe without synchronizing the whole map.

Reads can happen very fast while write is done with a lock.

There is no locking at the object level.

The [locking](https://crunchify.com/how-to-create-your-own-non-blocking-queue-in-java-same-as-evictingqueue/" \t "https://crunchify.com/hashmap-vs-concurrenthashmap-vs-synchronizedmap-how-a-hashmap-can-be-synchronized-in-java/_blank) is at a much finer granularity at a hashmap bucket level.

ConcurrentHashMap doesn’t throw a ConcurrentModificationException if one thread tries to modify it while another is iterating over it.

ConcurrentHashMap uses multitude of locks.

先通过两段散列函数来定位segment数组，然后获取对应的segement的锁，才能方位segement中的hashEntry对象

SynchronizedHashMap:

Synchronization at [Object level](https://crunchify.com/java-hashmap-containskeyobject-key-and-containsvalueobject-value-check-if-key-exists-in-map/" \t "https://crunchify.com/hashmap-vs-concurrenthashmap-vs-synchronizedmap-how-a-hashmap-can-be-synchronized-in-java/_blank).

Every read/write operation needs to acquire lock.

Locking the entire collection is a performance overhead.

This essentially gives access to only one thread to the entire map & blocks all the other threads.

It may cause contention.

SynchronizedHashMap returns Iterator, which fails-fast on concurrent modification.

HashCode方法即哈希表的散列函数，是一个native方法，用来通过对象的哈希值来定位在table数组的某一位置，以此减少equals的调用次数，提高存储性能

HashMap有一个叫做Entry的内部类，它用来存储key-value对。

上面的Entry对象是存储在一个叫做table的Entry数组中。

table的索引在逻辑上叫做“桶”(bucket)，它存储了链表的第一个元素。

key的hashcode()方法用来找到Entry对象所在的桶。

如果两个key有相同的hash值，他们会被放在table数组的同一个桶里面。

key的equals()方法用来确保key的唯一性。

value对象的equals()和hashcode()方法根本一点用也没有。

### Queue

定义：Queue means “waiting line”.

特点：队列在处理中时也是可以动态改变的

备注：Queue是顶层接口，主要有四个子接口：Deque，BlockingDeque，BlockingQueue，TtransferQueue，其中Deque是在java.util包，其它三个都在java.util.concurrent包中，主要应用在多线程编程中。Deque是现实中队列的抽象，LinkedList是典型的实现。

实现：

Queue继承了Collection接口，所以Queue的实现都包含了collection的功能：add,remove,contain,clear,isEmpty.其中基于队列头节点的操作是迅速的add(),offer().基于队中节点的操作是缓慢的(contains(e),remove(e))

本地队列：

LinkedList:双端队列，队列两端的增删操作速度快，提供额外的（低效率的）通过索引访问数据功能

PriorityQueue:内部数据具有优先级（默认是自然排序），

ArrayDeque:LinkedList的简单实现，不提供高效增删的特性，有随机访问功能

多线程队列：

ArrayBlockingQueue:简单的blockingQueue实现，有界队列

PriorityBlockingQueue：PriorityQueue和BlockingQueue的结合。

DelayQueue：插入其中的元素必须实现Delay接口

构造：

Queue<String> namesQueue = **new** LinkedList<>();

Deque<Integer> numbersDeque = **new** ArrayDeque<>();

BlockingQueue<String> waitingCustomers = **new** ArrayBlockingQueue<>(100);

遍历：

Queue<String> queueNames = **new** LinkedList<>();

queueNames.add("Dale");

queueNames.add("Bob");

queueNames.add("Frank");

queueNames.add("Alice");

queueNames.add("Eric");

queueNames.add("Cole");

queueNames.add("John");

**for** (String name : queueNames) {

    System.out.println(name);

}

### Arrays

------------------------------------------------------------------------------------------------------------

## IO

File f= new File("E:/IOTest.txt");

f.mkdirs();//创建文件夹,如果路径有多重的，前面不存在则会依次建立

f.mkdir();//不会依次建立

f.createNewFile();//文件夹也算是file如果存在同名文件夹，此方法也不会生效

while(streamReader.read(b)!=-1){每次读取一个字节

count++;//英文和数字1字节，汉字三个字节（GBK2字节，UTF83字节）

}

f.delete()

streamReader.read(b)此方法读取的字节数目等于字节数组的长度

a character contains 2 bytes, a bytes contains 8-bit;

complicated extensive underlying binary primitive

InputStream.read()返回的是什么?

read()这个方法完成的事情就是从数据源中读取8个二进制位，并将这8个0或1转换成十进制的整数，然后将其返回。

Input and output through data streams, serialization and the file system.

Charsets, decoders, and encoders, for translating between bytes and Unicode characters.

Access to file, file attributes and file systems.

APIs for building scalable servers using asynchronous or multiplexed, non-blocking I/O.

Byte Streams：

1.input and output of 8-bit bytes

2.All byte stream classes are descended from InputStream and OutputStream.

3.codeExample:

public class CopyBytes {

public static void main(String[] args) throws IOException {

FileInputStream in = null;

FileOutputStream out = null;

try {

in = new FileInputStream("xanadu.txt");

out = new FileOutputStream("outagain.txt");

int c;

while ((c = in.read()) != -1) {

out.write(c);

}

} finally {

if (in != null) {

in.close();

}

if (out != null) {

out.close();

}

}

}

}

btw:Byte streams should only be used for the most primitive I/O.So why talk about byte streams? Because all other stream types are built on byte streams.

Character Streams:

1.Character Streams will automatically translate the Unicode(JAVA stores charactor uses this) to local charset

2. Reader/Writer,可以在构造器中添加参数来改变编码

FileReader/FileWriter,可以在构造器中添加参数来改变编码

InputStreamReader/OutputStreamWriter:byte-to-character bridge streams;

3.BufferedReader/PrintWriter:line-oriented buffered streams;可以在构造器中添加参数来改变缓冲器的大小

数据，从哪里来，做了些什么，到哪里去

从哪里来：

文件

管道

网络连接

内存缓存

System.in, System.out, System.error(注：Java标准输入、输出、错误输出)

做了些什么：

文件访问：通过FILE类

网络访问

内存缓存访问

线程内部通信(管道)

缓冲

过滤

解析

读写文本 (Readers / Writers)

字符流，FileReader/

读写基本类型数据 (long, int etc.)

读写对象

FileInputStream/out,FileReader/writer都是一个字节字符的读，RandomAccessFile可以随机访问？？？

------------------------------------------------------------------------------------------------------------

## 多线程

### 基础：

Join:

一个线程里调用另一个线程的join方法，会让另一个线程执行完，才执行该线程。

Wait/Notify:

一条线程调用一个锁的wait，会让该线程在该锁上等待。需要等其它线程执行了该锁的notify方法时，并且其它线程的线程体执行完，等待在该锁上的线程才会继续执行。

Interrupt：

线程中断的方式

采用状态位来控制，在线程执行体中设置一个状态位，将状态位设为结束来结束线程体的执行。这种方式相比interrupt来说，可以释放资源

资源的限制：

硬件的资源限制：通过集群解决

软件的资源限制：通过资源复用

多线程无非需要掌握两个点，一个是利用多线程操作提高程序执行效率，一个是在多线程环境下，写出线程安全的程序

不可变变量（final）不需要加锁

单线程操作不需要加锁

1.复合操作的同步手段只有加锁吗？

2.伸缩性是什么意思

3.JVM规范

4.一个类在jvm里是如何存储的？

同步是什么？

程序中用于控制不同线程间操作发生的相对顺序的机制

线程之间如何同步

在共享内存模型中，通过程序显式的指定个方法或者某段代码在程序之间互斥进行

在消息传递模型中，消息传递要在消息接收之前，是隐式同步

线程之间如何通信

共享内存：

线程之间通过对共享状态的“写读”操作进行隐式通信。JAVA通过操作主存与各个线程"本地内存"(抽象概念，包括缓存，写缓冲区，寄存器等)的数据交换来完成线程间的通信

消息传递：

线程之间没有共享状态。通过发送消息进行通信

管道：

原子常量类原理，同步包装集合的原理，并发集合的原理，锁的原理，volatile原理

常用操作：

1.读改写，如i++

2.计数器

3.

Lock指令：

1.将缓存行的数据写回到内存

2.使其它cpu中缓存的该内存无效

写缓冲区：

用于临时保存向主存中写入的数据

上下文切换：

线程状态从保存到加载的过程，就是一次上下文切换

减少上下文切换的方式：

1无锁并发编程：根据数据ID的hash值取模分段，不同线程处理不同段的数据

2 CAS算法：Atomic包更新数据采用的cas算法，不需要加锁

3使用合理数量的线程

4协程

死锁：

常用避免死锁的方式

1避免一个线程争取多个锁

2一个锁只包含一个资源

3尝试使用定时锁，lock.tryLock(timeout)

4对于数据库锁，加锁和解锁在同一个链接里

### Lock:

于JAVA1.5出现，相较于synchronized，后者由于锁的获取和释放都是隐式的，较为简便。前者锁的获取和释放都是显示的，手动的。虽说麻烦一点，但是在涉及多个锁的时候更为清晰明了。并且lock接口也提供更多特性：非阻塞获取锁，响应中断获取锁，超时获取锁。

锁的API：

Void lock()

Void lockInterruptibly() throws InterruptedException

响应中断获取锁：在锁的获取中可以中断该线程

Boolean tryLock()

非阻塞获取锁：方法立即返回，能获取则返回true，不能则返回false

Boolean tryLock(long time,TimeUnit timeUnit) throws InterruptedException

响应中断，响应超时获取锁，在指定时间内获得锁或者在指定时间内线程中断或者超时之后，三种情况下会返回。

Void unlock()

Condition newCondition()

获得等待通知组件，该组件和该锁绑定。线程获得该锁后，才能调用该组件的wait方法，调用wait方法后会释放锁

可重入锁：

获得某个锁的线程可以再次获得该锁，并状态+1.释放的时候状态减一。状态为零是才是代表释放了

synchronized也是可重入的，在使用synchronized进行递归时，可以重复获得同一锁

而Lock接口的可重入实现ReentrantLock，虽然不能隐式重复获得锁，但是可以反复调用lock（）方法来加锁。该实现可以通过构造器来控制是否公平

示例：

读写锁

一般的锁，如上述的可重入锁都是排他锁，即同一时间同一资源只能被一条线程所占有

有某个线程持有读锁的情况下，写锁不能被获取

有某个线程持有写锁的情况下，读锁和写锁都不能被获取

读写锁的性能大多数情况下是高于排他锁的，因为读的情况往往是比写多的。试想一个案例，程序中有一个用作缓存的数据结构。对这个数据结构进行访问时，在没有读写锁以前，就需要用等待通知机制，写操作进行完后，通知其它线程读。而有了读写锁以后，事情就变得简单了。读的时候获取读锁，写的时候获取写锁。

Java并发包提供实现：ReentrantReadWriteLock

示例：

公平锁与非公平锁：

公平锁可以保证线程同步状态的获取保持顺序（在等待队列中依次来）非公平锁没有这个保证，所以同一个线程刚释放锁时，再去请求锁，有极大可能性还是这个线程获得锁。这样的坏处是让其它等待的线程等待更久，好处是极大的减少线程切换的开销，反而提高了吞吐量。所以非公平锁是默认选择。

共享锁和排他锁：

### 线程池：

应用场景：执行异步任务，或者执行并发任务。

工作时序：

1. 任务进入
2. 判断核心线程池是否已满，如果没满，则创建一条工作线程。如果满了则判断3.
3. 判断工作队列是否已满，如果没满，则将任务存储在工作队列中。如果满了则判断4.
4. 判断线程池是否已饱和，如果没饱和，则创建一条工作线程。如果饱和了，则执行饱和策略

备注：创建工作线程需要获取全局锁，效率堪忧

饱和策略：

1. AbotrtPolicy直接抛出异常
2. CallerRunsPolicy只用调用者所在线程来运行任务
3. DiscardOldestPolicy丢弃掉队列里的一个任务来执行该任务
4. DiscardPolicy不处理，丢弃该任务

执行任务：

两种方法：execute()和submit()。

执行的任务需要实现Runnable接口，

Execute()执行无返回值的 任务。Submit()执行有返回值的任务。该方法可以返回一个Future对象，可以调用future.get()来获得线程的返回值。

线程池关闭：

Shutdown() 和shutDownNow().前者是将线程置为SHUTDOWN状态，然后中断没有正在执行任务的线程。ShowDownNow方法是将线程置为STOP状态，然后尝试停止所有正在执行或者暂停执行的线程

线程池配置：

要配置线程池需先分析任务：

1. 任务的性质：CPU密集型，IO密集型

IO密集型需要分配较多的线程，CPU密集型需要分配较少的

1. 任务的优先级：高，中，低

优先级问题可以通过PriorityBlockingQueue来处理

1. 任务的执行时间：长，中，短

任务时间长短不同可以交由不同规模的线程池处理。时间长的任务交由规模大的线程池。也可以 通过优先级队列，让优先级高的任务先处理

1. 任务的依赖性：是否依赖其它资源，如数据库链接

若任务需要和数据库交互，由于提交SQL后会阻塞，等待数据库返回，所以执行时间较长，所以 需要配置较多的线程，这样可以更好的利用CPU

### Executor

基础功能：用来将线程任务和执行机制分离

组成：

任务：实现了Runnable或者Callable接口的类。Runnable无返回值，Callable有返回值

执行机制：

核心接口Executor以及继承了Executor的ExecutorService接口。ExecutorService接口有两个典型实现类：ThreadPoolExecutor和ScheduledThreadPoolExecutor。

ThreadPoolExecutor：线程池的核心实现类，用来执行被提交的任务。主要有三种实现类： FixedThreadPool：线程池线程数固定，corePoolSize和maxmumPoolSize数量相同.工作 队列为LikedBlockingQueue

SingleThreadExecutor：corePoolSize和maxmumPoolSize都为1,工作队列也是LinkedBlockingQ

CachedThreadPool：corePoolSize为0。maxmumPoolSize（）为最大值。线程默认存活时间为60秒。工作队列为SynchronousQueue

ScheduledThreadPoolExecutor：在一段延时后执行任务，或者定时执行任务。该实现类比Timer更加灵活强大

异步计算结果：核心接口Future，典型实现FutureTask

### 生产者消费者模式

该模式可以解决大部分并发问题

原理是通过一个阻塞队列充当第三者，来解耦。生产者和消费者不直接通信，而是生产者将数据放入阻塞队列中，然后消费者丛阻塞中获取数据。

纵观设计模式，有许多都是通过找一个第三者来进行解耦。如工厂模式的第三者是工厂类，模板模式的第三者是模板类，在学习这些模板模式的时候要善于取了解这些第三者

happens-before:

happens-before关系是面向程序员的，编译器和处理器可以对其进行重排序

happens-before保证多线程环境下，在正确同步的情况，执行结果不会发生变化

as-if-serial保证单线程情况下，执行结果不会发生变化

1.程序顺序规则：一个线程中，每个操作都happens-before后面的操作

2.监视器锁(内部锁)规则:一个锁的解锁happens-before随后对这个锁的加锁

3.volatile原则：一个volatile变量的写happens-before这个变量的读

4.传递性

5.start()原则:线程A调用线程B的start方法，这个start方法happens-before线程B中的任意操作

6.join()原则:线程A调用线程B的join方法并成功返回，则B中的方法happens-before join方法

延迟初始化：

两种方案：1.volatile 2.基于类的初始化

Volatile原理:

用法：

修饰一些简单的值，例如volatile flag=true或者 volatile int a =1;

原理：

添加LOCK指令，利用的缓存一致性的机制完成变量的修改

特点：

1.轻量级的同步手段，保证修饰变量的线程可见性

2.性能比synchronized好，不会引起上下文切换

3.复合操作不能使用volatile修饰，需要用对象锁或者lock api

synchonize原理：

修饰普通方法时，锁的是方法所属的实例对象

修饰静态方法时，锁的是该class

修饰代码块时，锁的是括号中的对象

Jvm中采用monitorenter和monitorexit两个指令完成synchronize操作，但并没有详细说明

执行同步代码块之前会先去申请对应的锁。持有锁才能访问对应同步代码块。为了减少竞争锁的消耗，将锁分为了偏向锁，轻量级锁，重量级锁（互斥锁）。锁竞争就会导致锁升级。

三种锁的生命周期：

偏向锁：由于在大多数情况下，锁往往是由同一线程重复获得，所以设计出了偏向锁这个东西。当一个线程访问同步代码块并获取锁的时候，会在对象头（锁对象的头）和栈帧的锁记录中存储锁偏向的线程的ID。以后该线程在进入和退出该同步块时，不会再通过CAS操作来获取和释放锁，而是看对象头中是否存储着指向当前线程的偏向锁。如果有，则认为获取了锁。如果无，则再看对象头中偏向锁标识是否为1，是则通过CAS尝试将该偏向锁指向当前线程，尝试失败撤销原来持有的线程的偏向锁：暂停持有该偏向锁的线程，执行线程体，重新设置栈帧中的锁记录和对象头中的偏向锁标识，再恢复该线程；否，则通过CAS操作竞争该锁；

轻量级锁：

线程在执行同步块前，会在栈帧中创建锁记录，并将锁对象头中的锁标识替换为指向该锁记录的指针，若成功则获取锁；若失败则表示该锁存在竞争，则自旋获取该锁，若失败则锁升级为重量级锁，该线程阻塞。

原子操作的原理：

CPU实现原子操作有两种方式：一种是通过锁住其它CPU与总线的通信来达到内存的独占，称之为总线锁。

另一种是通过基于缓存一致性的缓存锁

JAVA实现CAS操作：

CAS操作的问题：

1. ABA问题：即对象由A变成B再变成A，看上去没变，实际变了。解决方法：添加版本号
2. CAS循环时间长：无解
3. CAS操作只能实现一个变量的原子更新，无法实现多个变量的原子更新。解决方法：将多个变量合并为一个变量。

Lock：

1.可以非阻塞的获取锁（tryLock），成功则持有，（失败呢？）

2.获取到锁的线程可以响应中断，中断则释放锁。（synchronized呢？可以中断吗）

3.可以设置超时

操作原子性原理：

处理器的角度：

总线锁：

处理器提供一个LOCK信号，当CPU向总线发送LOCK信号时，其它CPU的请求会被阻塞，该处理器独占共享内存

缓存锁：

频繁使用的内存会被缓存在L1,L2,L3高速缓存中。高速缓存中数据的原子操作就不用向总线发起LOCK指令，而是修改内存地址，利用缓存一致性的机制来实现院子操作

JAVA的角度：

1.使用循环CAS操作

2.使用锁

缓存一致性：MESI协议：缓存行有四个状态

M：被修改的。只有本CPU中有缓存数据，状态被修改了，与内存不一致

处于M态的缓存行会监听所有试图读取该缓存行对应的主存地址的操作，会在操作之前，把缓存行中的数据写回CPU

E：独占的。只有本CPU中有缓存数据，状态没被修改，与内存一致

处于E态的缓存行，若读该行的主存地址，则该行变为S

S：共享的。多个CPU都有该缓存，且与内存一致

若监听到使该行无效或独占的请求，则转换为I态

I：无效的。该CPU中该缓存已经无效

CPU读数据时，若缓存行为I态，则从内存读，并变为S态。如果不是I态，则从缓存读（若该数据在其它CPU有该数据的缓存的M态，则需要等到M态处理完才能执行）

CPU写数据时，只有该缓存行是M态或者E态时才能执行，否则会发出RFO总线事务，开销较大。

等待/通知，消费者/生成者模型：

1.等待者获取资源的独占权（锁）

2.等待者对条件进行判断，若满足条件则执行任务，若不满足条件，则释放独占权，进入等待状态

3.通知者获取到等待者释放的独占权，对条件进行变更后，通知（唤醒）等待者

4.等待通知者任务执行完毕，对应的等待者重新执行第二步

等待/通知 超时模式模型：

long future = System.currentMills() + mills;

long remaining = mills;

while(result==null && remaining>0){

waiting(remaining);

remaining = future - System.currentMills();

}

return result;

}

线程通信:

1.IO管道 pipedInputStream/pipedOutputStream可以在同一进程的不同线程中传递字节；

进程通信:

同步代码块？

分岔口:

http://docs.oracle.com/javase/tutorial/essential/concurrency/threads.html

创建线程有两种策略：

每次都实例化一个Thread、用Executor

------------------------------------------------------------------------------------------------------------

## 泛型

<? super T>包括T在内的任何父类,<? extends T>包括T在内的任何子类

------------------------------------------------------------------------------------------------------------

## http

400：参数有问题

405：

URI ：Uniform Resource Identifier，统一资源标识符；

URL：Uniform Resource Locator，统一资源定位符；

URN：Uniform Resource Name，统一资源名称。

HTTP URL (URL是一种特殊类型的URI，包含了用于查找某个资源的足够的信息)的格式如下：

http://host[":"port][abs\_path]

网页的缓存是由HTTP消息头中的“Cache-control”来控制的，常见的取值有private、no-cache、max-age、must-revalidate等，默认为private。其作用根据不同的重新浏览方式分为以下几种情况：http://www.cnblogs.com/xuxiang/p/3407539.html

if (generatesDownloadContent()) {

response.setHeader("Pragma", "private");

response.setHeader("Cache-Control", "private, must-revalidate");

}

----------------------------------------------------------------------------------------------------------

## TOMCAT

TOMCAT性能相关

<http://tomcat.apache.org/articles/performance.pdf>

付：以下为国内著名电商某服务器的内核优化参数：

#Controls the default maxmimum size of a mesage queue

kernel.msgmnb = 65536

# Controls the maximum size of a message, in bytes

kernel.msgmax = 65536

# Controls the maximum shared segment size, in bytes

kernel.shmmax = 68719476736

# Controls the maximum number of shared memory segments, in pages

kernel.shmall = 4294967296

net.core.somaxconn = 32768

net.core.wmem\_default = 8388608

net.core.rmem\_default = 8388608

net.core.rmem\_max = 16777216

net.core.wmem\_max = 16777216

net.ipv4.tcp\_timestamps = 0

net.ipv4.tcp\_synack\_retries = 1

net.ipv4.tcp\_syn\_retries = 0

net.ipv4.tcp\_tw\_recycle = 1

net.ipv4.tcp\_tw\_reuse = 1

net.ipv4.tcp\_mem = 94500000 915000000 927000000

net.ipv4.tcp\_max\_orphans = 3276800

net.ipv4.ip\_local\_port\_range = 1024 65535

net.ipv4.tcp\_fin\_timeout = 10

net.ipv4.tcp\_keepalive\_time = 100

net.ipv4.tcp\_syncookies = 1

net.ipv4.tcp\_max\_syn\_backlog = 8192

net.ipv4.tcp\_max\_tw\_buckets = 20000

----------------------------------------------------------------------------------------------------------

## 路径

均指向根目录：

<script type="text/javascript" src="./js/test.js"></script>

<script type="text/javascript" src="js/test.js"></script>

指向：https://localhost:8888/js/test.js

<script type="text/javascript" src="/js/test.js"></script>

路径里面如果不带classpath则是根据文件的所在路径来定位，若带classpath：，则是根据classpath来相对定位。第一个斜杠有没有效果其实是一样的，所以不用带一个斜杠

---------------------------------------------------------------------------------------------------------

## 数据库相关

数据库存储的东西修改的越少越好，例如存birthday不存age

批量添加的时候，若开启自动事务，则会在最后一步提交事务，若此时添加的东西太多了，则有可能造成内存溢出，所以这种情况下，应该关闭自动事务，换成手动提交事务，每1000条提交一次

### DDL

### SQL

普通查询多拼一个字段

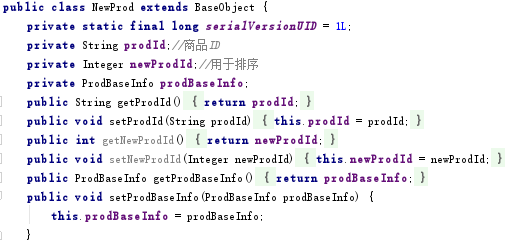
**SELECT** \* **FROM** (**select count from** TBL\_RUSH\_ACTIVITY\_QUEUE\_INFO **WHERE** ACTIVITY\_ID = '10000312'),TBL\_RUSH\_ACTIVITY\_QUEUE\_INFO **WHERE** TBL\_RUSH\_ACTIVITY\_QUEUE\_INFO.ACTIVITY\_ID = '10000312' **AND** TBL\_RUSH\_ACTIVITY\_QUEUE\_INFO.MEMBER\_ID = 'MB20003830';

### Ibatis

sqlmap分号检查（不能加）

参数##中不要加空格

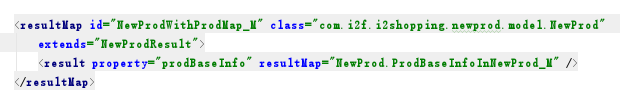
实体类：

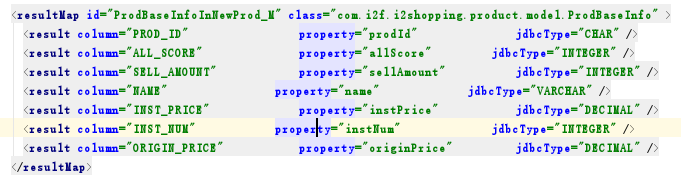


分页查询NewProd类，附加类中产品信息ProdBaseInfo, 查询语句传入参数为NewProd实例newProd

newProd.setProperties(**"catId"**,catId);







上述加载prodBaseInfo属性的sql只会执行一次（分页查一次，就附带这些产品信息），为全量加载。

下述方法为懒加载，分页时，如果查10条数据，则这个查附带信息的sql会被执行10次，加上分页的一次即为N+1次。

<**select id="selectNewProdWithProdPaged\_MNG\_LAZY" parameterClass="com.i2f.i2shopping.newprod.model.NewProd" resultMap="NewProdWithProdMap\_MNG\_LAZY"**>  
 select *\** from (  
 select a.*\** ,rownumber() over() as ROW\_NUM from (  
 select *\** FROM TBL\_NEW\_PROD  
 ORDER BY NEW\_PROD\_ID ASC  
 ) a)  
 <**include refid="Common.Paging\_Clause"** />  
</**select**>

<**resultMap id="NewProdWithProdMap\_MNG\_LAZY" class="com.i2f.i2shopping.newprod.model.NewProd" extends="NewProdResult"**>  
 <**result property="prodBaseInfo" column="PROD\_ID" select="NewProd.selectProd"**/>*<!—要加命名空间-->*  
</**resultMap**>  
<**select id="selectProd" parameterClass="java.lang.String" resultMap="ProdBaseInfoInNewProd\_MNG"**>  
 select *\** from PROD\_BASE\_INFO where PROD\_ID = #value#  
</**select**>

<**resultMap id="ProdBaseInfoInNewProd\_MNG" class="com.i2f.i2shopping.product.model.ProdBaseInfo"** >  
 <**result column="NAME" property="name" jdbcType="VARCHAR"** />  
 <**result column="IS\_PUTAWAY" property="isPutaway" jdbcType="CHAR"** />  
 <**result column="IS\_GROUPON\_PROD" property="isGrouponProd" jdbcType="CHAR"** />  
 <**result column="ALL\_SCORE" property="allScore" jdbcType="INTEGER"** />  
 <**result column="IS\_ONLY\_SCORE" property="isOnlyScore" jdbcType="CHAR"** />  
</**resultMap**>

DEBUG [http-8888-5] ResultSet.debug(27) | {rset-100118} Header: [NAME, IS\_PUTAWAY, IS\_GROUPON\_PROD, ALL\_SCORE, IS\_ONLY\_SCORE]

DEBUG [http-8888-5] ResultSet.debug(27) | {rset-100118} Result: [零售商品测试888, 1, 0, 0, 0]

如果限定死了resultMap，就可以直接select \* 了，不会查出所有字段的

### DB2

DB2添加主键自增长

CREATE TABLE

TBL\_NEW\_PROD

(

NEW\_PROD\_ID INTEGER NOT NULL GENERATED ALWAYS AS IDENTITY (START WITH 1, INCREMENT BY 1 ),

PROD\_ID VARCHAR(40) NOT NULL,

PRIMARY KEY (NEW\_PROD\_ID)

);

序列

CREATE SEQUENCE RUSHCAPTCHA\_SERIAL AS INT START WITH 1 INCREMENT BY 1 MINVALUE 1 NO MAXVALUE NO CYCLE NO CACHE ORDER;

SELECT NEXTVAL FOR RUSHCAPTCHA\_SERIAL FROM SYSIBM.SYSDUMMY1;

数据库数据生成序号：

**SELECT** *row\_number*() **over**() **as** ROW\_NUM,A.\* **FROM** (**SELECT** \* **FROM** TBL\_RUSH\_CAPTCHA ) **AS** A;

---------------------------------------------------------------------------------------------------------

## 前端

Window属性

Location:用来跳转资源，一般通过location.href来进行跳转，需要输入完整url，如下：

<button onclick="window.location.href='https://www.baidu.com'" ></button>

若不输入完整url，只输入baidu.com或者www.baidu.com则是修改相对路径，即最后一个“/”之后的内容，若输入/baidu.com或者/www.baidu.com则是修改host之后的内容,即修改后的资源路径为:http://localhost:8080/www.baidu.com

style="cursor: pointer;"光标为一个小手指

siblings获得兄弟节点

$("span[name='vendortotalPrice00001016'] span")前面的是父元素，后面的是子元素。可以翻译为$(span[name='vendortotalPrice00001016']).find('span');

$('.allLimit').each(function(index,item){

var bonus = BonusUtils.bonusHTMLtoObj($(item).next()[0])

if(!BonusUtils.isUsable(bonus)) {

$(item).addClass('disable')

}

})

offset()

获得元素的偏移量

x=$("p").offset();x=p元素的外边距加上内边距

x.top,x.left等可以获得上和左的偏移量（只能获得上和左）

id可以重复。

---------------------------------------------------------------------------------------------------------

## Redis

打开一个cmd，切换到redis目录，运行

redis-server.exe redis.windows.conf

再打开一个cmd，切换到redis目录，运行

redis-cli.exe -h 127.0.0.1 -p 6379

数据结构：5种：String，Hash，List，Sort，SortedSort。其中String和Hash使用较多

String键值对

get, set, getset, incr, decr, incrby, decrby, append

赋值：set myKey abc

取值：get myKey

删除：del mykey

Incr：incr num若num存在，而且是整形，则会加1然后返回。若num不存在，则会初始化为0，再+1。若num不是数字类型，则会返回错误信息。

Append：字符串追加，若key不存在，则等同于赋值。返回是字符串追加后的长度

HASH 用来存储对象

hset, hmset, hget, hmget, hgetall, hdel, del, hincrby, hexists, hlen, hkeys, hvals

HMSET:

HMSET hashtest username abg password 123456 age 18 sex 1

HGET:

HGET hashtest username

HGETALL:

HGETALL hashtest

LIST列表--按照插入的顺序排序

LPUSH listtest value1

LPUSH listtest value2

LRANGE listtest

SET集合，无序不重复

SADD settest 1

SADD settest 2

SADD settest 3

SMEMBERS settest;

Redis事务，redis作为非关系型数据库，其事务秉承了隔离性，原子性。但是值得注意的是redis事务并不支持回滚。用法：

list: 链表实现。

两端添加:l/rpush

查看列表:lrange name start end

两段弹出:l/rpop

sorted set：

有序字符串集合，常用于排序

zadd,zscore,zcard,zrem,zrange,zrange \*\* 0 10 withscores,

keys通用操作：

keys \*,

keys my?查看以my开头的所有key，

del keyname，

exist keyname，

rename keyname, newKeyname

expire keyname time设置过期时间

ttl keyname查看过期时间

type keyname查看类型

redis特性

redis具有多数据源特性，一个客户端最多16个，默认选择0号

move keyname 1，将当前数据库里的KEY移到1号

事务：multi开启事务，exec提交，discard回滚

持久化：RDB，AOF

RDB： 每隔多少秒写入一次

性能较AOF好，只有一个文件，备份容易。

缺点：可能有少量的数据丢失

配置：redis.config中的

save 900 1 代表每900秒至少发生1个key变化

save 300 10

save 60 20000

dbfilename dump.rdb

dir ./ redis根目录

AOF： 已日志的形式记录操作，REDIS启动时会读取该文件

性能较RDB差，每次KEY修改都会记录操作

redis -check -aof

配置：appendonly no默认不使用

appendfiname "\*\*\*.aof"

appendfsync always/everysec前者是每次修改都写，后者是每秒写一次

---------------------------------------------------------------------------------------------------

## 静态工具类

StringUtils:位于commons-lang包

IndexOf

Equals

IsBlank,isNotBlank

Trim

Split

Reverse：颠倒字符串

Capitalize：首字母大写

Isalpha：只由字母组成。IsAlphaSpace：字母空格。IsAlphaNumeric：字母数字。IsAlphaNumericSpace：字母数字空格。IsNumeric：数字。IsNumericSpace：数字空格。

随机排序一个LIST

public static List<String> getRandomChargeUrl(List<String> urllist) {

if (urllist==null) {

return null;

}

List<String> urlRandomList = new ArrayList<String>();

for (int x = urllist.size(); x > 0; x--) {

int i = (int) (Math.random() \* x);

urlRandomList.add(urllist.get(i));

urllist.remove(i);

}

return urlRandomList;

}

创建100W条数据

CREATE TABLE user100w (

id INT NOT NULL AUTO\_INCREMENT,

first\_name VARCHAR(10) NOT NULL,

last\_name VARCHAR(10) NOT NULL,

sex VARCHAR(5) NOT NULL,

score INT NOT NULL,

copy\_id INT NOT NULL,

PRIMARY KEY (`id`)

);

BEGIN

DECLARE rowid INT DEFAULT 0;

DECLARE num INT;

DECLARE firstname CHAR(1);

DECLARE name1 CHAR(1);

DECLARE name2 CHAR(1);

DECLARE lastname VARCHAR(3) DEFAULT '';

DECLARE sex CHAR(1);

DECLARE score CHAR(2);

SET num=1000000;

SET @exedata = "";

WHILE rowid < num DO

SET firstname = SUBSTRING('赵钱孙李周吴郑王林杨柳刘孙陈江阮侯邹高彭徐',FLOOR(1+21\*RAND()),1);

SET name1 = SUBSTRING('一二三四五六七八九十甲乙丙丁静景京晶名明铭敏闵民军君俊骏天田甜兲恬益依成城诚立莉力黎励',ROUND(1+43\*RAND()),1);

SET name2 = SUBSTRING('一二三四五六七八九十甲乙丙丁静景京晶名明铭敏闵民军君俊骏天田甜兲恬益依成城诚立莉力黎励',ROUND(1+43\*RAND()),1);

SET sex=FLOOR(0 + (RAND() \* 2));

SET score= FLOOR(40 + (RAND() \*60));

SET rowid = rowid + 1;

IF ROUND(RAND())=0 THEN

SET lastname =name1;

END IF;

IF ROUND(RAND())=1 THEN

SET lastname = CONCAT(name1,name2);

END IF;

IF length(@exedata)>0 THEN

SET @exedata = CONCAT(@exedata,',');

END IF;

SET @exedata=concat(@exedata,"('",firstname,"','",lastname,"','",sex,"','",score,"','",rowid,"')");

IF rowid%1000=0

THEN

SET @exesql =concat("insert into user100w(first\_name,last\_name,sex,score,copy\_id) values ", @exedata);

prepare stmt from @exesql;

execute stmt;

DEALLOCATE prepare stmt;

SET @exedata = "";

END IF;

END WHILE;

IF length(@exedata)>0

THEN

SET @exesql =concat("insert into user100w(first\_name,last\_name,sex,score,copy\_id) values ", @exedata);

prepare stmt from @exesql;

execute stmt;

DEALLOCATE prepare stmt;

END IF;

END

## LINUX

Gnome-system-monitor

SSH走22端口

HTTP走80端口

## SERVLET

Servlet

1. Servlet是独立运行于web服务器的程序，这项技术用来取代CGI程序。
2. 没有特殊要求，创建一个类继承HttpServlet就可以了，有特殊要求的话，再自己写Servlet
3. Servlet的生命周期钩子函数：init(),service(),destroy();
4. 一般情况下，服务器启动时，会加载servlet（创建实例）（如果在webxml中loadonstartup中部配置，则是容器自己判断什么时候加载该servlet），此时会调用servlet的init（）方法（仅此调用一次）。之后客户端每次通过URL访问该servlet，会通过service方法，将请求分发给doPost，doGet等方法处理，怎么写doGet和doPost方法？



1. 如何在Spring环境下，运用自己写的servlet类？

Web.xml



小tips:url-pattern的匹配原则：1.精确定位>2最长匹配>3统配>4\*.do，另外通配只能通配后面的不能通配前面的，如可以匹配/a/\*,但不能匹配/\*/a

1. 如何脱离Spring环境部署Servlet？

## 前端模板

## 日志

一般选用slf4j+logback。Slf4j和common.logging的不同是，后者是运行时发现日志，前者是编译时发现。前者的性能更好，而logback是slf4j的原生实现，二者搭配口味更佳。

在Springboot下，配置是这样的：

## Spring文档阅读

### 章节I:spring概览

a lightweight solution a potential one-stop-shop modular

轻量级 一站式 模块化

declarative transaction management

声明式的事务管理

remote access to your logic through RMI or web services

通过RMI或者WEB SERVICE来远程调用服务

various options for persisting your data

多种方式来存储你的数据

enables you to integrate AOP transparently into your software

使您能够将AOP透明地集成到您的软件中。

Spring enables you to build applications from "plain old Java objects" (POJOs) and to apply enterprise services non-invasively to POJOs. This capability applies to the Java SE programming model and to full and partial Java EE.

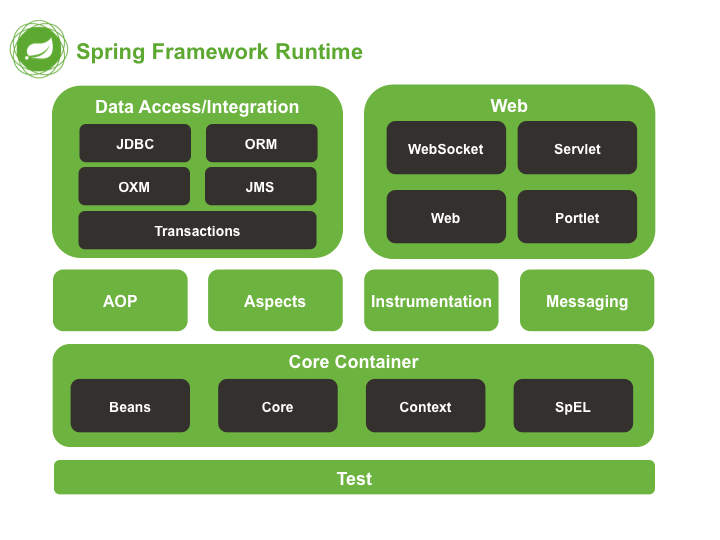
SPRING让你创建POJO，并对这些POJO加上处理逻辑

Make a Java method execute in a database transaction without having to deal with transaction APIS.

Make a local Java method an HTTP endpoint without having to deal with the Servlet API.

Make a local Java method a message handler without having to deal with the JMS API.

Make a local Java method a management operation without having to deal with the JMX APINaN



### 章节II:spring4新特性

1. 全面支持java8，最低要求java6
2. 支持@restcontroller，在类上面使用这个注解，就不用给每个方法添加@requestbody注解了

### 章节III:CoreTechnologies

7.1介绍了IOC的基础概念

1. org.springframework.beans 和 org.springframework.context 包是 Spring Framework 的 IoC  
   容器的根本。BeanFactory 接口提供了一种更先进的配置机制来管理任意类型的对象。  
   ApplicationContext 是 BeanFactory 的一个子接口。ApplicationContext 使得和 Spring 的  
   AOP 功能集成变得更简单；添加了信息资源处理（国际化中使用） ，事件发布；还添加了应  
   用程序层的特殊上下文 ，如用于 web 应用程序的 WebApplicationContext
2. BeanFactory提供了配置框架和基本功能，而ApplicationContext 添加了更多企业  
   应用功能。ApplicationContext完整扩展了BeanFactory
3. 在Spring中，由Spring IoC容器管理的，构成你的程序骨架的这些对象叫做bean，bean 和 bean 的依赖被容器所使用的配置元数据所反射
4. org.springframework.context.ApplicationContext 代表 Spring IoC 容器，并负责实例化，配置  
   和组装bean 的接口。容器通过读取元数据来获得组装beans的信息。元数据的种类有：xml，注解，java代码。
5. Java-based configuration: Starting with Spring 3.0, many features provided by the Spring JavaConfig project became part of the core Spring Framework. Thus you can define beans external to your  
   application classes by using Java rather than XML files. To use these new features, see the  
   @Configuration, @Bean, @Import and @DependsOn annotations
6. 用java代码的方式：类上用@configuration,方法上用@bean
7. 容器中一般不配置细粒度domain object，因为这些一般由dao或者逻辑代码去创建和加载。但是你可以通过aspectJ来配置容器外创建的类

7.2实例化IOC容器：

III.1

ApplicationContext context =  
**new** ClassPathXmlApplicationContext(**new** String[] {"services.xml", "daos.xml"});

**<bean id**=**"petStore" class**=**"org.springframework.samples.jpetstore.services.PetStoreServiceImpl">  
<property name**=**"accountDao" ref**=**"accountDao"/>  
<property name**=**"itemDao" ref**=**"itemDao"/>***<!-- additional collaborators and configuration for this bean go here -->***</bean>**

Name属性代表这个bean中的一个property（域），ref属性代表容器中另外一个bean的定义。ID和ref之间的关系就代表着这两个类之间的依赖关系。

如III.1中，xml文件可以用过ApplicationContext的构造来加载，同样也可以用import标签进行加载，如III.2

III.2

**<beans>  
<import resource**=**"services.xml"/>  
<import resource**=**"resources/messageSource.xml"/>  
<import resource**=**"/resources/themeSource.xml"/>  
<bean id**=**"bean1" class**=**"..."/>  
<bean id**=**"bean2" class**=**"..."/>  
</beans>**

三个路径都是相对定位。所以service.xml需要在相同目录下，或者是classPath目录下，后面两个则是在当前目录的resouces目录下。第三个省略了第二种的第一个斜杠，两者效果其实一样。两外，最好不要用../

7.3使用容器

ApplicationContext使用高级工厂方法来管理bean和bean的依赖，可以用通过下面的方法来获得bean

*// create and configure beans*ApplicationContext context = **new** ClassPathXmlApplicationContext("services.xml", "daos.xml");  
*// retrieve configured instance*PetStoreService service = context.getBean("petStore", PetStoreService.**class**);  
*// use configured instance*List<String> userList = service.getUsernameList();

或者

GenericApplicationContext context = **new** GenericApplicationContext();  
**new** XmlBeanDefinitionReader(context).loadBeanDefinitions(***"services.xml"***, ***"daos.xml"***);  
context.refresh();

Bean概览

Spring通过beanDefinition来实例化bean，BeanDefinition包括

*A package-qualified class name:* typically the actual implementation class of the bean being defined.  
 Bean behavioral configuration elements, which state how the bean should behave in the container  
(scope, lifecycle callbacks, and so forth).  
  
 References to other beans that are needed for the bean to do its work; these references are also  
called *collaborators* or *dependencies*.  
 Other configuration settings to set in the newly created object, for example, the number of connections  
to use in a bean that manages a connection pool, or the size limit of the pool

也可以用过ApplicationContext来获得beanfactory，调用beanfactory的registerSingletom()和registerDefinition()来引入容器外的类，但是一般没人这么搞。并且最好不要在运行时取修改已经注册的bean。

Bean的命名：id和name用来定义bean的名称，一般用id做为唯一命名，如果非要搞别名，可以通过name属性添加多个别名，用逗号分号或者空格来分隔。但是id和name属性不是必须的，使用inner beans and autowiring collaborators.就可以不提供bean name。指定别名的意义：比如，多个系统共用一个数据源，A系统中叫a，B系统中叫b，C系统中叫c现在要将他们组合到一个父系统中，则可以通过别名来关联他们

**<alias name**=**"fromName" alias**=**"toName"/>**

Bean的实例化：

容器是根据beanDefinition来实例化bean的。在xml中，class属性可以填两种形式，一种是有对应构造器的类，一种是指向该类的含有静态工厂方法的类。

根据构造方法来实例化bean：

这种方法必须包含构造器

**<bean id**=**"exampleBean" class**=**"examples.ExampleBean"/>  
<bean name**=**"anotherExample" class**=**"examples.ExampleBeanTwo"/>**

根据静态工厂方法实例化bean：

**<bean id**=**"clientService"  
class**=**"examples.ClientService"  
factory-method**=**"createInstance"/>  
public class** ClientService {  
**private static** ClientService clientService = **new** ClientService();  
**private** ClientService() {}  
**public static** ClientService createInstance() {  
**return** clientService;  
}  
}

根据实例工厂实例化bean：

和上面的区别是，工厂本身也受Spring管理

*<!-- the factory bean, which contains a method called createInstance() -->***<bean id**=**"serviceLocator" class**=**"examples.DefaultServiceLocator">***<!-- inject any dependencies required by this locator bean -->***</bean>***<!-- the bean to be created via the factory bean -->***<bean id**=**"clientService"  
factory-bean**=**"serviceLocator"  
factory-method**=**"createClientServiceInstance"/>  
public class** DefaultServiceLocator {  
**private static** ClientService clientService = **new** ClientServiceImpl();  
**private** DefaultServiceLocator() {}  
**public** ClientService createClientServiceInstance() {  
**return** clientService;  
}  
}

7.4依赖：

依赖注入：对象在通过构造器或者工厂方法实例化后，通过构造器或者工厂方法的参数或实例的属性来定义依赖的过程。Spring通过这种方法来将bean自己控制依赖转换成Spring将依赖注入到bean当中（通过类的构造或者Service Locator）。好处：代码更简洁，耦合性更低，更易于测试

主要方式有两种：构造器注入，setter注入

构造器注入：

其中constructor-arg的参数有：name,index,type等

**<beans>  
<bean id**=**"foo" class**=**"x.y.Foo">  
<constructor-arg ref**=**"bar"/>  
<constructor-arg ref**=**"baz"/>  
</bean>  
<bean id**=**"bar" class**=**"x.y.Bar"/>  
<bean id**=**"baz" class**=**"x.y.Baz"/>  
</beans>**

setter注入：

**public class** SimpleMovieLister {  
*// the SimpleMovieLister has a dependency on the MovieFinder***private** MovieFinder movieFinder;  
*// a setter method so that the Spring container can inject a MovieFinder***public void** setMovieFinder(MovieFinder movieFinder) {  
**this**.movieFinder = movieFinder;  
}  
*// business logic that actually uses the injected MovieFinder is omitted...*}

两者区别：Spring推荐构造器注入，这样可以保证注入的依赖不为null，并且是不可变对象？不过构造器注入可能导致环形注入（互相依赖，互相等待）

Beans that are singletonscoped and set to be pre-instantiated (the default) are created when the container is created. Otherwise, the bean is created only when it is requested.

下面两种定义依赖的形式，第一种比第二种好，第一种可以在部署期间校验bean是否存在

**<bean id**=**"theTargetBean" class**=**"..."/>  
<bean id**=**"theClientBean" class**=**"...">  
<property name**=**"targetName">  
<idref bean**=**"theTargetBean"/>  
</property>  
</bean>**

**<bean id**=**"theTargetBean" class**=**"..." />  
<bean id**=**"client" class**=**"...">  
<property name**=**"targetName" value**=**"theTargetBean"/>  
</bean>**

还有一种定义一个bean中的依赖就是用<ref bean=’’/>，通过这种方式引用的bean在同一个容器或者父容器中都可以，值可以是ID或者name中的一个

内部Bean

**<bean id**=**"outer" class**=**"...">***<!-- instead of using a reference to a target bean, simply define the target bean inline -->***<property name**=**"target">  
<bean class**=**"com.example.Person">** *<!-- this is the inner bean -->***<property name**=**"name" value**=**"Fiona Apple"/>  
<property name**=**"age" value**=**"25"/>  
</bean>  
</property>  
</bean>**

内部bean不用指定id或者name，指定了也无效

Property指定集合：

**<bean id**=**"moreComplexObject" class**=**"example.ComplexObject">***<!-- results in a setAdminEmails(java.util.Properties) call -->***<property name**=**"adminEmails">  
<props>  
<prop key**=**"administrator">**administrator@example.org**</prop>  
<prop key**=**"support">**support@example.org**</prop>  
<prop key**=**"development">**development@example.org**</prop>  
</props>  
</property>***<!-- results in a setSomeList(java.util.List) call -->***<property name**=**"someList">  
<list>  
<value>**a list element followed by a reference**</value>  
<ref bean**=**"myDataSource" />  
</list>  
</property>***<!-- results in a setSomeMap(java.util.Map) call -->***<property name**=**"someMap">  
<map>  
<entry key**=**"an entry" value**=**"just some string"/>  
<entry key** =**"a ref" value-ref**=**"myDataSource"/>  
</map>  
</property>***<!-- results in a setSomeSet(java.util.Set) call -->***<property name**=**"someSet">  
<set>  
<value>**just some string**</value>  
<ref bean**=**"myDataSource" />  
</set>  
</property>  
</bean>**

Bean中配置的集合页可以有合并操作和父子关系。但是不允许不同类型的集合合并，会抛异常。如果在java类中将某个属性设置为某一类型，如：float等，在配置bean的时候，可以自动将字符串转换为对应的类型。Null和空字符串的注入：

**<bean class**=**"ExampleBean">  
<property name**=**"email">  
<null/>  
</property>  
</bean>**

**<bean class**=**"ExampleBean">  
<property name**=**"email" value**=**""/>  
</bean>**

用p-namespace来引入property。虽然用schema:p可以让xml定义更佳简洁，但是不推荐。：

**<beans xmlns**=**"http://www.springframework.org/schema/beans"  
xmlns:xsi**=**"http://www.w3.org/2001/XMLSchema-instance"  
xmlns:p**=**"http://www.springframework.org/schema/p"  
xsi:schemaLocation**=**"http://www.springframework.org/schema/beans  
http://www.springframework.org/schema/beans/spring-beans.xsd">  
<bean name**=**"classic" class**=**"com.example.ExampleBean">  
<property name**=**"email" value**=**"foo@bar.com"/>  
</bean>  
<bean name**=**"p-namespace" class**=**"com.example.ExampleBean"  
p:email**=**"foo@bar.com"/>  
</beans>**

和

**<beans xmlns**=**"http://www.springframework.org/schema/beans"  
xmlns:xsi**=**"http://www.w3.org/2001/XMLSchema-instance"  
xmlns:p**=**"http://www.springframework.org/schema/p"  
xsi:schemaLocation**=**"http://www.springframework.org/schema/beans  
http://www.springframework.org/schema/beans/spring-beans.xsd">  
<bean name**=**"john-classic" class**=**"com.example.Person">  
<property name**=**"name" value**=**"John Doe"/>  
<property name**=**"spouse" ref**=**"jane"/>  
</bean>  
<bean name**=**"john-modern"  
class**=**"com.example.Person"  
p:name**=**"John Doe"  
p:spouse-ref**=**"jane"/>  
<bean name**=**"jane" class**=**"com.example.Person">  
<property name**=**"name" value**=**"Jane Doe"/>  
</bean>  
</beans>**

可以用连点的形式写，但是路径上所有的reference都不能为null：

**<bean id**=**"foo" class**=**"foo.Bar">  
<property name**=**"fred.bob.sammy" value**=**"123" />  
</bean>**

**<bean id**=**"beanOne" class**=**"ExampleBean" depends-on**=**"manager,accountDao">  
<property name**=**"manager" ref**=**"manager" />  
</bean>  
<bean id**=**"manager" class**=**"ManagerBean" />  
<bean id**=**"accountDao" class**=**"x.y.jdbc.JdbcAccountDao" />**

Depends-on属性可以让某个bean在此bean之前优先加载，同时也优先销毁。

延迟加载：在bean definition中添加对应属性，可以让bean在第一次被使用时才实例化。但是还是采用默认的非eager模式比较好，这样bean在Spring启动时就能发现其中的错误。可以通过在Container中配置默认模式，免得给每个bean都加lazy标签：

**<bean id**=**"lazy" class**=**"com.foo.ExpensiveToCreateBean" lazy-init**=**"true"/>**

**<beans default-lazy-init**=**"true">***<!-- no beans will be pre-instantiated... -->***</beans>**

自动装配：

目的是减少配置的繁杂行，减少错误的发生（我就经历过这种错误好几次）

自动装配的主要方式：

ByName:在类中声明的变量名必须要和beanDefinition中的该bean的ID相同

ByType:根据Class来匹配，若容器中有多个相同的class则会报错

方法注入：

方法注入的最初目的是为了解决一个单例中引用一个非单例：

*<!-- a stateful bean deployed as a prototype (non-singleton) -->***<bean id**=**"myCommand" class**=**"fiona.apple.AsyncCommand" scope**=**"prototype">***<!-- inject dependencies here as required -->***</bean>***<!-- commandProcessor uses statefulCommandHelper -->***<bean id**=**"commandManager" class**=**"fiona.apple.CommandManager">  
<lookup-method name**=**"createCommand" bean**=**"myCommand"/>  
</bean>**

或者：

**public abstract class** CommandManager {  
**public** Object process(Object commandState) {  
Command command = createCommand();  
command.setState(commandState);  
**return** command.execute();  
}  
@Lookup("myCommand")  
**protected abstract** Command createCommand();  
}

**<public|protected>** [abstract] **<return-type>** theMethodName(no-arguments);

Bean的作用域：singleton,prototype,request,session,applicaiton,websoket

通常，对于所有有状态的beans使用原型作用域，对于无状态的beans使用单例作用域。

对于非单例bean，需要额外去关注资源的释放，可以尝试使用Spring的bean后处理器：

post-processor。上述作用域中，后面四个都是web-aware的Context才能使用的，需要在web.xml中添加额外的配置：  
**<web-app>**...  
**<listener>  
<listener-class>**org.springframework.web.context.request.RequestContextListener  
**</listener-class>  
</listener>**...  
**</web-app>**

**<web-app>**...  
**<filter>  
<filter-name>**requestContextFilter**</filter-name>  
<filter-class>**org.springframework.web.filter.RequestContextFilter**</filter-class>  
</filter>  
<filter-mapping>  
<filter-name>**requestContextFilter**</filter-name>  
<url-pattern>**/\***</url-pattern>  
</filter-mapping>**...  
**</web-app>**

当向一个声明周期较长的bean中注入一个声明周期较短的bean可以考虑用动态代理（具体怎么用没看懂）

Bean的生命周期回调函数：

3种方式：

1. 在bean的方法上架@PreDestroy或者@PostConstruct
2. Xml中的配置是在bean后面加一个init-method或者destroy-method
3. 实现InitializingBean，DisposableBean接口不过不推荐这样做，会增加代码耦合性

还可以添加容器默认初始化方法：

<beans default-init-method="init"></beans>

上述三种方法可以都使用，如果方法是一样的，只会执行一次。如果方法不一样，则按照如下顺序：

Methods annotated with @PostConstruct

afterPropertiesSet() as defined by the InitializingBean callback interface

A custom configured init() method

Spring提供一系列\*\*aware接口用来赋予bean直接与container联系的能力，比如applicationContextawre和beannameare。但是使用这些aware接口就使代码和Spring API耦合了。

Bean Definition的继承：

**<bean id**=**"inheritedTestBeanWithoutClass" abstract**=**"true">  
<property name**=**"name" value**=**"parent"/>  
<property name**=**"age" value**=**"1"/>  
</bean>  
<bean id**=**"inheritsWithClass" class**=**"org.springframework.beans.DerivedTestBean"  
parent**=**"inheritedTestBeanWithoutClass" init-method**=**"initialize">  
<property name**=**"name" value**=**"override"/>***<!-- age will inherit the value of 1 from the parent bean definition-->***</bean>**

这里父类definition加了一个abstruct=“true”的标记，标识容器并不会实例化这个bean，而是单纯的当作其它bean definition的父类。

### Spring事务管理

Spring事务机制分为声明式事务和编程式事务。编程式事务可以获取事务的执行状态（成功或者失败），编程式事务分为使用transactionTemplate（不好用？），TransactionManager

Typically you need an application server’s JTA capability only if your application needs to handle transactions across multiple resources，which is not a requirement for many applications

事务管理策略无非就是：什么时候commit，什么时候rollback

用hbiernate：You can also use Hibernate local transactions easily, as shown in the following examples. In this case, you need to define a Hibernate LocalSessionFactoryBean, which your application code will use to obtain Hibernate Session instances.

声明式事务的好处：

Most Spring Framework users choose declarative transaction management. This option has the least impact on application code, and hence is most consistent with the ideals of a *non-invasive* lightweight container.

SPRING事务的配置与SPRING事务传播：

[https://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/#transaction-declarative-first-example](https://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/" \l "transaction-declarative-first-example)

Spring form

[https://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/#view-jsp-formtaglib](https://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/" \l "view-jsp-formtaglib)

### SpringAOP

1.AOP概览：

编程范式：1.面向过程编程 2.面向对象编程 3面向函数编程 4事件驱动编程 5面向切面编程

AOP是一种编程范式，用来解决特定问题，来辅助面向对象编程

设计初衷：1.减少代码重复 2.关注点分离：1.水平分离：展示层，服务层，持久层 2.垂直分离：模块划分（订单， 库存等） 3.切面分离：分离功能性需求和 非功能性需求

典型应用：权限控制，缓存控制，事务控制，审计日志，性能监控，分布式追踪，异常处理

2.

4.为什么要引入AOP，AOP的好处和适用场景，AOP的缺点？

5.AOP的使用：

Poincut expression包括

designators 用来描述要匹配什么样的切点

\*\*\*匹配方法：execution()

匹配注解：@args参数级别 @target类级别 @within类级别 @annotation方法级别

匹配包/类型：within（）

匹配对象：this() bean() target()

匹配参数：args()

wildcards \*匹配任意数量的字符 ..任意数的子包或者参数 +匹配指定类及子类

operators && || ! 与或非

excution:{modifier-pattern修饰符(可省略),ret-type-pattern返回值,declaring-type-pattern包名(可省略),name- pattern（param-pattern）方法名（方法参数）,throws-pattern异常(可省略)}

advice:@Befor,@After(finaly),@AfterReturning,@AfterThrowing，@Around

6.AOP部分原理：

织入时机：1.编译期 2类加载时 3运行时。Spring采用的是运行时织入，运用了代理技术

7.代理部分原理：

分为静态代理和动态代理。

静态代理的缺点是，若接口的方法过多，会导致代理类也要搞这多方法，代码重复性高

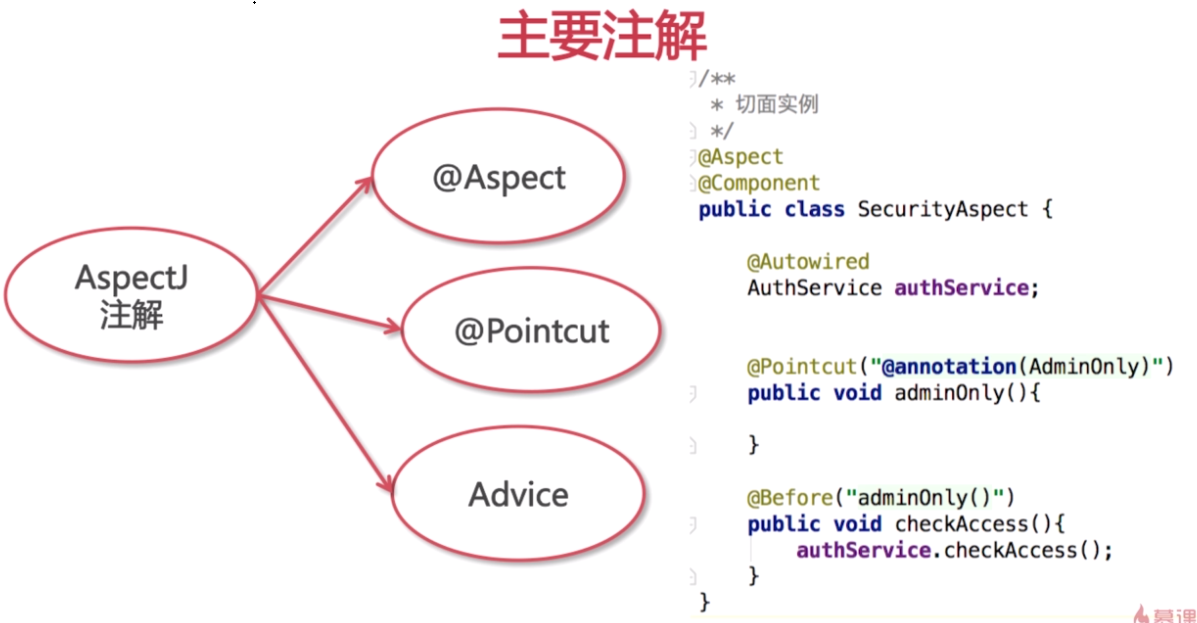
动态代理分为基于接口代理和基于继承代理。二者实现的经典代表就是JDK代理和CGLIB代理。

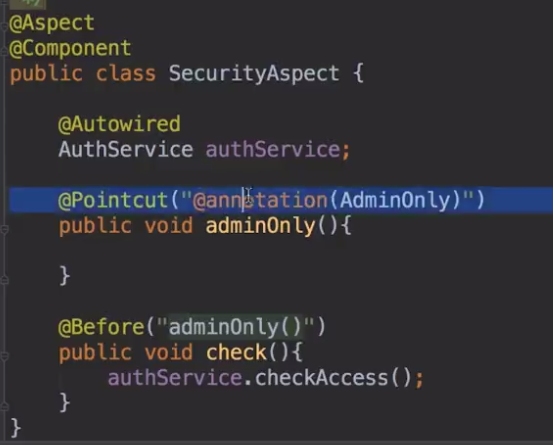
JDK代理只能针对有接口的类的接口方法进行代理。

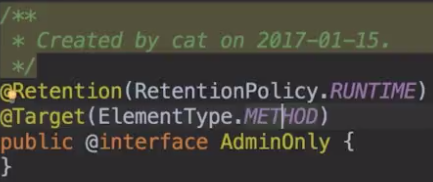
CGLIB由于是基于继承实现的，所以无法对static和final类进行代理。同样也不能对private和static方法进行代理

AOP（面向切面编程）是OOP（面向对象编程）的一种补充。SpringAOP的主要用途有两点：

1. 提供声明式事务管理
2. 可以自定义切面来完善编程







### springMVC

ModelAndView:

用途：将数据与模板绑定在一起

备注：在SpringBoot和Spring中皆可使用

示例：

@RequestMapping("/testModelAndView")  
public ModelAndView testModelAndView(){  
 ModelAndView modelAndView = new ModelAndView("testModelAndViewFTL","name","abg");  
 return modelAndView;  
}

前端：模板采用freemark，文件位于resource/template，模板文件名：testModelAndViewFTL.ftl。利用${name}获得数据

#### 综合

跳页面的方式展现list

url: location.href="${ctx}/exhibitionfloor/test.html?testParam="+'123';

后台：

String testParam = request.getParameter("testParam");

System.out.println(testParam);

ModelAndView mav = new ModelAndView("homepage/floor/test");

List<Object> list = new ArrayList<Object>();

HomePageFloor h1 = new HomePageFloor();

h1.setFloorName("h1");

HomePageFloor h2 = new HomePageFloor();

h2.setFloorName("h2");

list.add(h1);

list.add(h2);

mav.addObject("list", list);

return mav;

前台：

<c:forEach var="l" items="${list}">

${l.floorName }

</c:forEach>

ajax方式:

后台： List<Object> list = new ArrayList<Object>();

HomePageFloor h1 = new HomePageFloor();

h1.setFloorName("h1");

HomePageFloor h2 = new HomePageFloor();

h2.setFloorName("h2");

list.add(h1);

list.add(h2);

Map<String,Object> map = new HashMap<String,Object>();

map.put("list", list);

return new ModelAndView("jsonView",map);

前台： $.ajax({

type:"get",

url:'${ctx}/exhibitionfloor/test.html',

//dataType:"json",

success:function(data){

var list = data.list;

for(var i=0;i<list.length;i++){

console.log(list[i].floorName)

}

}

});

对于无返回值方法：

public boolean voidMethod(){

boolean flag = false;

try{

//process

Flag = true;

}catch(Exception e){

\*\*\*\*\*\*

}

Return flag;

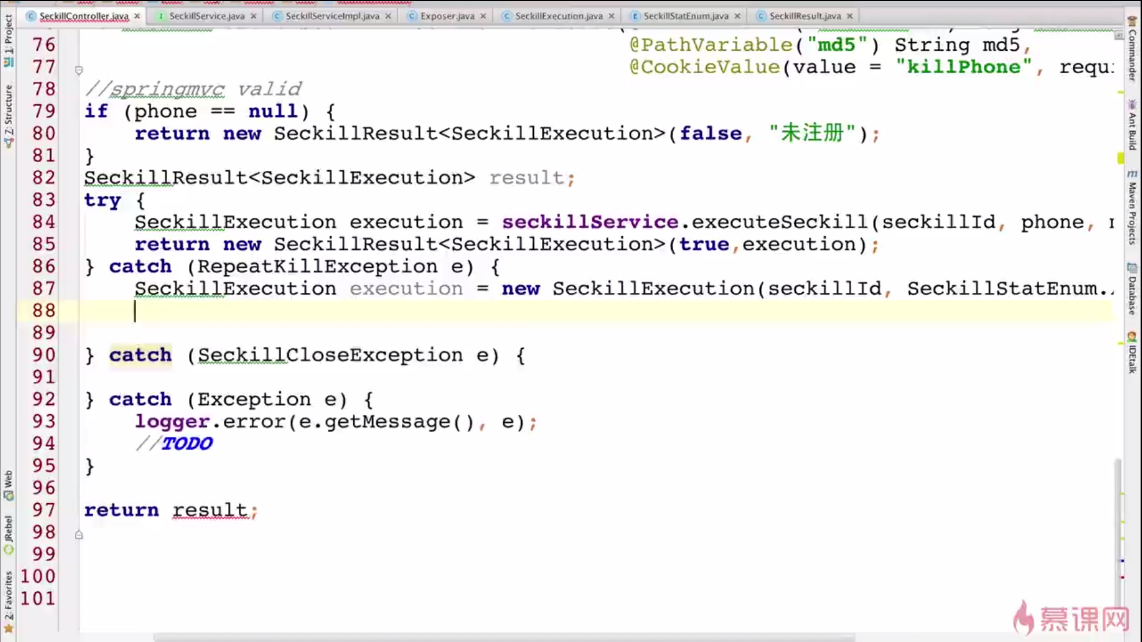
}

#### 视图、视图解析器的介绍

http://elim.iteye.com/blog/1770554

dispatcherServlet：[https://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/#mvc-servlet](https://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/" \l "mvc-servlet)

#### controller







Spring Security

<http>

</http>

httpSessionContextIntegrationFilter

authenticationManager关联一个session-controller名字为currentController(需要注入sessionRegistry)

用来控制session的行为

authenticationProvider关联一个userLoginInfoService和一个密码加密器

用来指定用户信息来源

authenticationProcessingFilter

用来存储表单中存入的账号和密码，并指定跳转页面

concurrentSessionFilter(需要注入SessionRegistry)

authenticationProcessingFilterEntryPoint

authenticationDetailsSource

accessDecisionManager

filterSecurityInterceptor

用户的权限控制都包含在这个过滤器中。

功能一：如果用户尚未登陆，则抛出AuthenticationCredentialsNotFoundException“尚未认证异常”。

功能二：如果用户已登录，但是没有访问当前资源的权限，则抛出AccessDeniedException“拒绝访问异常”。

功能三：如果用户已登录，也具有访问当前资源的权限，则放行。

## 工具

1. 制图软件
2. 压力测试工具

## 代码风格

添加默认的构造方法

restful接口设计：

参数校验应该放在service层进行

## 设计模式

单例模式：

*Factory*, *Abstract Factory*, *Builder*, *Decorator*, and *Service Locator*

BeanFactory是工厂模式的完美实现

Spring jar包整合：Table 2.1. Spring Framework Artifacts

-Xmx5120m -Xms5120m -Xmn1024m -XX:PermSize=512m -XX:MaxPermSize=1024m

## CSS

https://www.w3schools.com/css/css3\_animations.asp

https://daneden.github.io/animate.css/

这个库是很出名的

纯css 动画库

https://css-tricks.com/almanac/properties/a/animation/

这个网站专门搞css的，宇宙第一

刚才这个库的github社区活跃度

https://cssanimation.rocks/

这个实用教程版，有很多例子的

https://cssanimation.rocks/starwars/

https://cssanimation.rocks/portal/

<http://cssanimate.com/>

## IDEA

IDEA创建maven工程：

打开maven auto-import,否则不会自动下载包

idea文件夹下无法创建class

将此文件夹变为source

IDEA创建springboot工程：

这三个地方需要一致：

1，Project Structure里确认两个地方:Project sdk以及project language level

2，Project Structure->Modules里Sources里的Language level

3，Preferences->java Compiler->Per-module bytecode Version

IDEA优化：

关闭乱七八糟的插件

inspection 搜索spell关闭拼写检查

inspection 搜索unused 选择java栏中的unused declaration中的method和parameter丛public改为protect

## Spingboot

Idea创建SpringBoot项目：

spring.datasource.url = jdbc:mysql://localhost:3306/test?characterEncoding=utf8&useSSL=true

spring.datasource.username = root

spring.datasource.password = 111111

spring.datasource.driverClassName = com.mysql.jdbc.Driver

#?characterEncoding=utf8&useSSL=true不写这个会导致 WARN: Establishing SSL connection without server's identity verification is not recommended. According to MySQL 5.5.45+, 5.6.26+ and 5.7.6+ requirements SSL connection must be established by default if explicit opti

关于devtools：

http://blog.csdn.net/isea533/article/details/70495714

## VUE

<https://github.com/Hxvin/vue-home>

--------------------------------------------------------------------------------------------------

数据绑定

<div id="app" v-html='message'>

</div>

<script>

new Vue({

el: '#app',

data: {

message: '<a href="https://www.baidu.com">baidu</a>!'

}

})

</script>

--------------------------------------------------------------------------------------------------

样式绑定

<style>

.class1{

background: #444;

color: #eee;

}

</style>

<div id="app" >

<label for="c">修改颜色</label>

<input type="checkbox" v-model="c" id="c">

<div v-bind:class="{'class1':c}">呵呵</div>

</div>：

<script>

new Vue({

el: '#app',

data: {

c:false

}

})

</script>

其实就是通过checkbox改变c的bool值

--------------------------------------------------------------------------------------------------

<div id="app" >

<p v-if="yesorno">猜猜我是谁</p>

<label for="bucai">不猜,滚</label>

<input type="checkbox" id="bucai" v-model="yesorno">

</div>

<script>

new Vue({

el:'#app',

data:{

yesorno:false

}

})

</script>

--------------------------------------------------------------------------------------------------

<div id="app" >

<p>{{hehe}}</p>

<input type="text" v-model="hehe">

<button v-on:click="reverse()">反转</button>

</div>

<script>

new Vue({

el:'#app',

data:{

hehe:"123456789"

},

methods:{

reverse:function () {

this.hehe = this.hehe.split("").reverse().join("");

}

}

})

</script>

小TIPS

页面数据没显示，首先F12检查DOM，看是否有数据

Enumeration enumeration = request.getParameterNames();  
if(enumeration!=null){  
 String paramName = null;  
 while(enumeration.hasMoreElements()){  
 paramName = (String) enumeration.nextElement();  
 buffer.append(";").append(paramName).append("=").append(request.getParameter(paramName));  
 }  
}

## Context

ServletContext,ApplicationContext,WebApplicationContext

ServletContext:

早期的web应用由许多servlet堆砌而成，故web服务器又称之为servlet容器。当web服务器启动后， 会初始化一个servletContext，被所有servlet共享

我们所使用的java web技术都是基于servlet技术的，所以肯定是有servletContext的

每个应用都会有一个对应的ServletContext，当应用部署在多台机器上时，每台机器都有一个独立的ServletContext实例，缺省情况下ServletContext不是分布式的。

ServletContext的生命周期和Web服务相同。

ServletContext可以访问应用的静态资源：getResource(“/path”)，getResourceAsStream(“/path”)

ServletContext servletContext = webApplicationContext.getServletContext();

ApplicationContext（接口）

ApplicationContext继承BeanFactory,在[国际化支持](#_国际化支持)，[资源访问](#_资源访问)（如URL和文件），[事件传播](#_事件传播)等有良好的支持

ApplicationContext实例化后会对所有单例bean进行装配，使之成为待使用状态，而beanFactory初始化时并不会对bean进行装配

可以通过实现ApplicationContextAware接口来获得ApplicationContext

WebApplicationContext

继承自ApplicationContext，，提供了web应用的configuration，并且可以获得servletContext

WebApplicationContext webApplicationContext = ContextLoader.*getCurrentWebApplicationContext*();

WebApplicationContextUtils提供一个方法getWebApplicationContext(ServletContext sc)可以通过ServletContext实例来获得webApplicationContext实例。

## Session

## Listner

classLoadListener，ServletContextListener

## 代数基础

## 数据结构与算法

^是[异或运算](https://www.baidu.com/s?wd=%E5%BC%82%E6%88%96%E8%BF%90%E7%AE%97&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao" \t "https://zhidao.baidu.com/question/_blank)符（把[数据转换](https://www.baidu.com/s?wd=%E6%95%B0%E6%8D%AE%E8%BD%AC%E6%8D%A2&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao" \t "https://zhidao.baidu.com/question/_blank)成二进制，然后按位进行运算）。

运算规则：0^0 = 0， 1^0 = 1，  0^1 = 1，  1^1 = 0，运算对象相同为0，不同为1.

如：3^5 的运算过程为：

    (1)先将3和5转换成二进制的11和101

    (2)再按对应的位分别进行运算，11位数不足补零

           011

       ^   101

      -----------

           110

     (3)运算结果转换成10进制：6

[异或运算](https://www.baidu.com/s?wd=%E5%BC%82%E6%88%96%E8%BF%90%E7%AE%97&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao" \t "https://zhidao.baidu.com/question/_blank)的三个个特点:

    (1) 0^0=0,   0^1=1   0与任何数异或＝任何数

    (2) 1^0=1,   1^1=0   1与任何数异或 =任何数取反

    (3) 任何数异或自己＝把自己置0

异或运算的常见用途:

    (1) 使某些特定的位翻转

    例如对数10100001的第2位和第3位翻转，其他位不变，则可以将该数与00000110进行按位异或运算。

    10100001^00000110 = 10100111

    (2) 实现两个值的交换，而不必使用临时变量。

    例如交换两个整数a=10100001，b=00000110的值，可通过下列语句实现：

a = a^b； 　　//a=10100111

b = b^a； 　　//b=10100001

a = a^b； 　　//a=00000110

    (3) 在[汇编语言](https://www.baidu.com/s?wd=%E6%B1%87%E7%BC%96%E8%AF%AD%E8%A8%80&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao" \t "https://zhidao.baidu.com/question/_blank)中经常用于将变量置零：

            xor   a，a

    (4) 快速判断两个值是否相等

            判断两个整数a，b是否相等，则可通过下列语句实现：

            return ((a ^ b) == 0)

### 二叉树：

树中的任何节点的值大于它的左子节点，且小于它的右子节点

#### **平衡二叉树：**

二叉树容易失衡，造成只有一边有子树，这样会导致搜索速度下降，因此出现了平衡二叉树。

#### **排序二叉树：**

它是一棵空树或它的左右两个子树的高度差的绝对值不超过1，并且左右两个子树都是一棵平衡二叉树。也就是说该二叉 树的任何一个等等子节点，其左右子树的高度都相近

23树：

在普通的二叉查找树种，插入节点的逻辑是：从根节点开始比较，若比根节点小，则选择左支，然后比较子树的左节点，依次循环。

在2-3树种，分为普通节点（2节点，包含一个元素，两条分支）和包含两个元素，三条分支的节点（3节点）。3节点中，左支最小，中支在两个元素之间，右支最大。这种数据结构可以保证二叉树是总是平衡的。

新增节点逻辑：

1. 如果插入值到一个2节点，则将2节点拓展为3节点
2. 如果插入值到一个3节点：
   1. 3节点无父节点，则先变成一个4节点，即包含三个元素，然后分解为一个二叉树
   2. 3节点有一个2节点的父节点，则先把3节点变为4节点，然后分解4节点为一个二叉树，并将此二叉树的父节点与2节点融合，生成一个新的3节点
   3. 3节点有一个3节点的父节点，则在情况（2）接触上，继续分解，继续向上融合，直到父类是一个2节点。若到根节点了父节点依然是3节点，则直接分解3节点，生成一个新的二叉树，层数+1

红黑树的另一种定义：

1. 所有红链接都是左连接
2. 没有任何一个节点同时和两条红链接相连
3. 该二叉查找树是完美黑色平衡的，即任意空链接到根节点的路径上的黑链接数量相同

红黑树：

性质一：节点是红色或者是黑色；

性质二：根节点是黑色

性质三：每个叶节点（NIL或空节点）是黑色；

性质四：每个红色节点的两个子节点都是黑色的（也就是说不存在两个连续的红色节点）；

性质五：从任一节点到其没个叶节点的所有路径都包含相同数目的黑色节点；

## 功能点

### 国际化支持

### 资源访问

### 事件传播

Questions:

static final是否能初始化为NULL，然后运行时注入

否，final类型需要初始化