**JAVA笔记**

**1. 基本数据类型**

|  |  |  |  |
| --- | --- | --- | --- |
| **类型** | **占用存储空间(字节)** | **表示范围** | **包装类型** |
| **byte** | **1** | **-128~127** | **Byte** |
| **short** | **2** | **-215~215-1** | **Short** |
| **int** | **4** | **-231~231-1** | **Integer** |
| **long** | **8** | **-263~263-1** | **Long** |
| **float** | **4** | **-2128~2128**  **, 约等于**  **-3.403E38~3.403E38** | **Float** |
| **double** | **8** | **-21024~21024**  **, 约等于**  **-1.798E308~1.798E308** | **Double** |
| **char** | **2** |  | **Character** |
| **boolean** | **1** | **true, flase** | **Boolean** |

**Java基本数据类型有固定的表示范围和字段长度，且不受操作系统影响，因此具有跨平台性**

**1.1. 整数类型**

**默认的整数类型是int**

**计算机实际以补码形式存储整数，进行运算时也是以补码形式进行运算**

**byte一般作为数组使用byte[]**

**整数声明为long时，数字后需加l或L，如long num = 10l**

**1字节=8位，整数由符号位和数值部分组成，符号位0为正1为负**

**byte占1字节，最高位为符号位，可表示256个数，0~127和-0~-127，这里用-0，即1000 0000表示-128，short\int\long同样用-0表示最小负数**

**整数十进制：进位为10，10**

**整数八进制：以数字0开头，进位为8，012**

**整数十六进制：以数字0加字母x开头，A~F分别表示10~15，进位为16，0xA**

**1.2. 浮点类型**

**默认的浮点类型是double**

**浮点数声明为float时，数字后面需加f或F，如float num = 3.14f**

**浮点数由符号位，指数位，尾数位组成**

**float：1bit符号位+8bits指数位+23bits尾数位=4字节组成**

**double：1bit符号位+11bits指数位+52bits尾数位=8字节组成**

**浮点数中指数位的底数为2，指数位的表示范围为-127~128，指数位的表示方法为实际指数值+偏移量127，即若实际指数值为0，则指数位应写入127(0111 1111)**

**浮点数中尾数位为小数点后各位数，默认省略小数点前的一个1**

**浮点数虽然表示数的范围扩大了，但精度有限，因此，需要精确计算时应使用java.math.BigDecimal类对浮点数进行封装**

**如进行精确的浮点数除法，其中scale参数为计算的精确，后面的数字将被四舍五入**

**public static double div(double v1, double v2, int scale) {**

**if (scale < 0) {**

**throw new IllegalArgumentException("The scale must be a positive integer or zero");**

**}**

**BigDecimal b1 = new BigDecimal(Double.toString(v1));**

**BigDecimal b2 = new BigDecimal(Double.toString(v2));**

**return b1.divide(b2, scale, BigDecimal.ROUND\_HALF\_UP).doubleValue();**

**}**

**1.3. 字符类型**

**采用Unicode编码**

**允许使用转义字符，如char a = ‘\n’;表示换行**

**1.4. 布尔类型**

**取值仅有两种：true,false**

**1.5. 基本数据类型的转换**

**各种数据类型容量大小：byte,short,char<int<long<float<long**

**byte,short,char三种类型间不会相互转换，必要时它们会首先转换为int类型**

**多种数据类型混合计算的时候，系统自动将各个数转换为容量最大的数，再继续进行计算**

**2. 表达式**

**aaa**

**3. 语句**

**aaa**

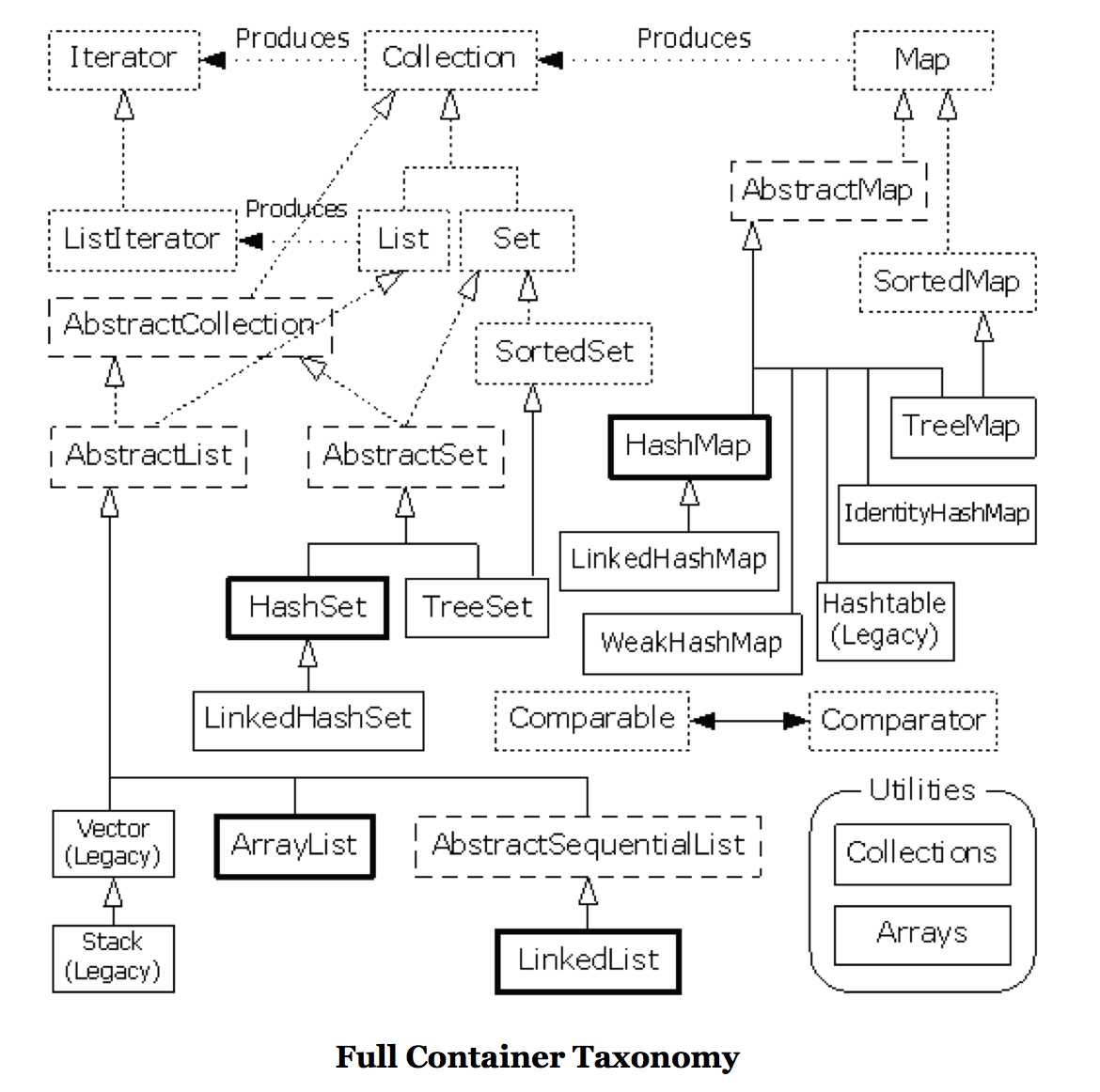
**4. 函数**

**aaa**

**5. 类**

**aaa**

**6. 容器**



**JAVA集合只能存放引用类型的的数据，不能存放基本数据类型**

**6.1. ArrayList**

**内部通过数组实现，在存储空间不够时，默认扩展50%+1的空间。**

**理论上LinkedList插入速度快于ArrayList，但当存储元素达到上万条以后，元素的插入速度和查找速度，ArrayList都远优于LinkedList，主要原因是，当数据量很大时，LinkedList大量时间消耗在插入之前插入位置的寻址上面，而ArrayList通过下标寻址速度非常快。**

**引入包**

**import java.util.ArrayList;**

**初始化**

**List<String> list = Lists.newArrayList(); //推荐，需要引入 import com.google.common.collect.Lists;这个包**

**List<String> list = Lists.newArrayList(“element1”, “element2”, “element3”); //推荐**

**List<String> list = new ArrayList<String>(); //创建对象，并指名存储元素类型**

**List list1 = new ArrayList(); //创建对象，存储Object**

**ArrayList list2 = new ArrayList(); //创建对象，存储Object**

**List<String> list3 = Arrays.asList(“element1”, “element2”, “element3”); //推荐，列表初始化，Arrays类的静态方法asList()用于将数组作为列表**

**List<String> list4 = new ArrayList<String>() { {**

**add(“element1”);**

**add(“element2”);**

**}}**

**添加元素**

**list.add(“element1”); //向尾部添加元素**

**list.add(0, “element2”); //向指定位置插入元素**

**list1.addAll(list); //向尾部添加一组元素**

**list2.addAll(0, list); //向指定位置插入一组元素**

**删除元素**

**list.remove(0); //删除指定位置元素**

**list.remove(“element1”); //若列表中有“element1”存在，则删除首次出现的该元素**

**list.removeRange(0, 2); //删除列表中下标从0到1的元素**

**list.removeAll(); //删除列表中所有元素**

**list.clear(); //删除列表中所有元素**

**查找元素**

**String val = list.get(0); //获取指定位置处元素，list1/list2里存储的为泛型Object，需要强制类型转换，list则不需要**

**boolean result = list.contains(“element1”); //判断列表是否包含指定元素**

**int pos1 = list.indexOf(“element1”); //返回元素第一次出现在列表中的下标，若没有该元素返回-1**

**int pos2 = list.lastindexOf(“element1”); //返回元素最后一次出现在列表中的下标，若没有该元素返回-1**

**修改元素**

**list.set(0, “element4”); //修改指定位置元素**

**获得元素个数**

**list.size();**

**判断列表是否为空**

**list.isEmpty(); //若空则返回true，否则返回false**

**遍历**

**1) Using iterator**

Iterator<String> iter = list.iterator();

while (iter.hasNext()) {

    String value = iter.next();

}

**2) Using loop**

for (String iter : list) {

    System.out.println(iter);

}

**3) Using loop**

for (int i = 0; i < list.size(); i++) {

    System.out.println(list.get(i));

}

1和2性能接近，3性能稍好

**6.2. LinkedList**

**采用双向链表结构存储数据**

**引入包**

**import java.util.LinkedList;**

**初始化**

**LinkedList<String> list = new LinkedList<String>();**

**添加元素**

**list.push(“element1”); //插入元素到头部，作为栈使用**

**list.offer(“element2”); //插入元素到尾部，作为队列使用**

**list.addFirst(“element3”); //插入元素到头部**

**list.addLast(“element4”); //插入元素到尾部**

**list.add(“element5”); //向尾部添加元素**

**list.add(0, “element6”); //向指定位置插入元素**

**list1.addAll(list); //向尾部添加一组元素**

**list2.addAll(0, list); //向指定位置插入一组元素**

**删除元素**

**list.pop(); //删除头部元素，作为栈使用**

**list.poll(); //删除头部元素，作为队列使用**

**list.removeFirst(); //删除头部元素**

**list.removeLast(); //删除尾部元素**

**list.remove(); //删除头部元素**

**list.remove(0); //删除指定位置元素**

**list.remove(“element1”); //若列表中有“element1”存在，则删除首次出现的该元素**

**list.clear(); //删除列表中所有元素**

**查找元素**

**String val1 = list.peek(); //获取头部元素，作为栈、队列使用**

**String val1 = list.getFirst(); //获取头部元素**

**String val2 = list.getLast(); //获取尾部元素**

**String val3 = list.get(0); //获取指定位置处元素**

**boolean result = list.contains(“element1”); //判断列表是否包含指定元素**

**int pos1 = list.indexOf(“element1”); //返回元素第一次出现在列表中的下标，若没有该元素返回-1**

**int pos2 = list.lastindexOf(“element1”); //返回元素最后一次出现在列表中的下标，若没有该元素返回-1**

**修改元素**

**list.set(0, “element7”); //修改指定位置元素**

**获取元素个数**

**list.size();**

**判断列表是否为空**

**list.isEmpty(); //若空则返回true，否则返回false**

**遍历**

**1) Using iterator**

Iterator<String> iter = list.iterator();

while (iter.hasNext()) {

    String value = iter.next();

}

**2) Using loop**

for (String iter : list) {

    System.out.println(iter);

}

**3) Using loop**

for (int i = 0; i < list.size(); i++) {

    System.out.println(list.get(i));

}

不建议采用随机访问方式遍历，推荐使用方法2

**6.3. HashSet**

**引入包**

**import java.util.HashSet;**

**初始化**

**Set<String> set = Sets.newHashSet(); //推荐，需要引入import com.google.common.collect.Sets;这个包**

**Set<String> set = Sets.newHashSet(“element1”, “element2”, “element3”); //推荐初始化**

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

**添加元素**

**set.add(“element1”); //向集合中添加元素，并保证集合中元素唯一**

**删除元素**

**set.remove(“element1”); //若集合中有“element1”存在，则删除**

**set.clear(); //删除集合中所有元素**

**查找元素**

**boolean result = set.contains(“element1”); //判断集合是否包含指定元素**

**获取元素个数**

**set.size();**

**判断是否为空**

**set.isEmpty(); //若空则返回true，否则返回false**

**遍历**

**1) Using iterator**

Iterator<String> iter = set.iterator();

while (iter.hasNext()) {

    String value = iter.next();

}

**2) Using loop**

for (String iter : set) {

    System.out.println(iter);

}

**6.4. HashMap**

**引入包**

**import java.util.ArrayMap;**

**初始化**

**Map<String, Object> map = Maps.newHashMap(); //推荐，需要引入import com.google.common.collect.Maps;这个包**

**Map<String, Object> map = Maps.newHashMap(“key1”, “value1”, “key2”, “value2”); //推荐初始化，保证键值唯一**

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

**添加元素**

**map.put(key3, value3); //向集合中添加映射，若映射的键值key3存在，则修改该映射的值为新值**

**map1.putAll(map); //向集合添加一组映射**

**删除元素**

**map.remove(“key1”); //删除具有指定键值的映射**

**map.clear(); //删除集合中所有映射**

**查找元素**

**Object val1 = map.get(“key1”); //获取指定键值映射所对应的值**

**boolean result1 = map.containsKey(“key1”); //判断集合是否包含指定键值的映射，若存在该映射只能有一个**

**boolean result2 = map.containsValue(“value1”); //判断集合是否包含指定值的映射，若存在该映射可能有多个**

**Set<String> set = map.keySet(); //返回map中key组成的Set集合**

**Set<Map.Entry<String, Object>> set = map.entrySet(); //返回map中映射组成的Set集合**

**Collection<Object> list = map.values(); //返回map中value组成的列表**

**获取元素个数**

**map.size();**

**判断是否为空**

**map.isEmpty(); //若空则返回true，否则返回false**

**遍历**

**1) Using enrtySet() in for each loop**

for (Map.Entry<Integer, String> entry : map.entrySet()) {

    Integer key = entry.getKey();

    String value = entry.getValue();

}

**2) Using keySet() or values() in for each loop**

如果只需要map中的键或者值，可以通过keySet()或values()来实现遍历，而不是用entrySet。

for (Integer key : map.keySet()) {

  String value = map.get(key);

System.out.println("key="+key);

}

或者

for (String value : map.values()) {

System.out.println("Value="+value);

}

**3) Using enrtySet() and iterator**

Iterator<Map.Entry<Integer, String>> iter1 = map.entrySet().iterator();

while(iter1.hasNext())

{

    Map.Entry<Integer, String> entry = iter1.next();

    Integer key = entry.getKey();

    String value = entry.getValue();

}

**4) Using keySet() and iterator**

Iterator iter2 = map.keySet().iterator();

while(iter2.hasNext())

{

    Integer key = iter2.next();

    String value = map.get(key);

}

**Output of above program (in milliseconds) :**

Using entrySet() in for-each loop :**50**  
Using keySet() in for-each loop :**76**  
Using entrySet() and iterator :**50**  
Using keySet() and iterator :**75**

推荐使用方法1(键值都要)。如果仅需要键(keys)或值(values)使用方法2或4。如果使用的语言版本低于java 5，或是打算在遍历时删除entries，必须使用方法3。

**6.5. HashTable**

**实现一个哈希表，将键映射到值。**

**HashTable中的key和value都不允许为null。**

**HashMap中允许存在一个为null的key和任意个为null的value。**

**引入包**

**import java.util.HashTable;**

**初始化**

**HashTable hashTable1 = new HashTable();**

**HashTable<Integer, Object> hashTable2 = new HashTable<Integer, Object>();**

**添加元素**

**hashTable1.put(key3, value3); //向集合中添加映射，若映射的键值key3存在，则修改该映射的值为新值**

**hashTable1.putAll(map); //向集合添加一组HashMap映射**

**删除元素**

**hashTable.remove(“key1”); //删除具有指定键值的映射**

**hashTable.clear(); //删除集合中所有映射**

**查找元素**

**Object val1 = hashTable.get(“key1”); //获取指定键值映射所对应的值**

**boolean result1 = hashTable.containsKey(“key1”); //判断集合是否包含指定键值的映射，若存在该映射只能有一个**

**boolean result2 = hashTable.containsValue(“value1”); //判断集合是否包含指定值的映射，若存在该映射可能有多个**

**Set<String> set = hashTable.keySet(); //返回map中key组成的Set集合**

**Set<Map.Entry<String, Object>> set = hashTable.entrySet(); //返回map中映射组成的Set集合**

**Collection<Object> list = hashTable.values(); //返回map中value组成的列表**

**获取元素个数**

**hashTable.size();**

**判断是否为空**

**hashTable.isEmpty(); //若空则返回true，否则返回false**

**遍历**

**1) Using enrtySet() in for each loop**

for (Map.Entry<Integer, String> entry : hashTable.entrySet()) {

    Integer key = entry.getKey();

    String value = entry.getValue();

}

**2) Using keySet() or values() in for each loop**

如果只需要map中的键或者值，可以通过keySet()或values()来实现遍历，而不是用entrySet。

for (Integer key : hashTable.keySet()) {

  String value = hashTable.get(key);

System.out.println("key="+key);

}

或者

for (String value : hashTable.values()) {

System.out.println("Value="+value);

}

**3) Using enrtySet() and iterator**

Iterator<Map.Entry<Integer, String>> iterpublic static void main(String[] args) {

SparkSession sparkSession = SparkSession.builder().master("local").appName("CollaborativeFiltering").getOrCreate();

String path = "file:///home/fanyuguang/Documents/recommender-systems/u.data";

JavaRDD<String> rowDataset = sparkSession.read().textFile(path).javaRDD();

JavaRDD<Rating> ratings = rowDataset.map(

new Function<String, Rating>() {

public Rating call(String s) throws Exception {

String[] sarray = s.trim().split("\\s+");

return new Rating(Integer.parseInt(sarray[0]), Integer.parseInt(sarray[1]), Double.parseDouble(sarray[2]));

}

}

);

int i = 1;

for (Rating r : ratings.collect()) {

System.out.println(i++ + " " + r);

}

}1 = hashTable.entrySet().iterator();

while(iter1.hasNext())

{

    Map.Entry<Integer, String> entry = iter1.next();

    Integer key = entry.getKey();

    String value = entry.getValue();

}

**4) Using keySet() and iterator**

Iterator iter2 = hashTable.keySet().iterator();

while(iter2.hasNext())

{

    Integer key = iter2.next();

    String value = hashTable.get(key);

}

**6.6. Vector**

**内部通过数组实现，在存储空间不够时，默认扩展1倍的空间。Vector为线程安全，要求某一时刻只有一个线程允许写Vector，但因此相对于ArrayList需要更大的系统开销。**

**引入包**

**import java.util.Vector;**

**初始化**

**Vector vector = new Vector(); //默认数组容量为10**

**Vector vector1 = new Vector(20); //初始化容量为20的数组**

**Vector<String> vector2 = new Vector<String>();**

**添加元素**

**vector1.addElement(“element1”); //添加元素**

**vector1.insertElement(“element2”, 1); //在指定位置出入元素，原位置元素依次后移**

**vector2.add(“element1”); //向尾部添加元素**

**vector2.add(0, “element2”); //向指定位置插入元素**

**vector1.addAll(vector2); //向尾部添加一组元素**

**vector1.addAll(0, vector2); //向指定位置插入一组元素**

**删除元素**

**vector.removeElementAt(0); //删除指定位置元素**

**vector.removeElement(“element1”); //若列表中有“element1”存在，则删除首次出现的该元素**

**vector.removeAllElement(); //删除列表中所有元素**

**vector.remove(0); //删除指定位置元素**

**vector.remove(“element1”); //若列表中有“element1”存在，则删除首次出现的该元素**

**vector.removeRange(0, 2); //删除列表中下标从0到1的元素**

**vector.removeAll(); //删除列表中所有元素**

**vector.clear(); //删除列表中所有元素**

**查找元素**

**String val = vector.get(0); //获取指定位置处元素，如存储的为泛型Object，需要强制类型转换**

**boolean result = vector.contains(“element1”); //判断列表是否包含指定元素**

**int pos1 = vector.indexOf(“element1”); //返回元素第一次出现在列表中的下标，若没有该元素返回-1**

**int pos2 = vector.lastindexOf(“element1”); //返回元素最后一次出现在列表中的下标，若没有该元素返回-1**

**修改元素**

**vector.set(0, “element4”); //修改指定位置元素**

**vector.setElementAt(“element2”, 1); //修改指定位置元素**

**获取元素个数**

**vector.size();**

**判断是否为空**

**vector.isEmpty(); //若空则返回true，否则返回false**

**遍历**

**1) Using iterator**

Iterator<String> iter = vector.iterator();

while (iter.hasNext()) {

    String value = iter.next();

}

**2) Using loop**

for (String iter : vector) {

    System.out.println(iter);

}

**3) Using loop**

for (int i = 0; i < vector.size(); i++) {

    System.out.println(vector.get(i));

}

**6.7. Stack**

**引入包**

**import java.util.Stack;**

**初始化**

**Stack stack = new Stack();**

**添加元素**

**stack.push(“element1”); //压入元素到栈顶部**

**删除元素**

**stack.pop(); //弹出栈顶部元素**

**查找元素**

**String val1 = stack.peek(); //获取栈顶部元素**

**int index = stack.search(“element”);**

**判断是否为空**

**stack.empty();**

**6.8. Queue**

**Java中Queue是一个接口，不能实例化，可用LinkedList实现队列**

**引入包**

**import java.util.Queue;**

**初始化**

**Queue<Integer> queue = new LinkedList<Integer>();**

**添加元素**

**queue.offer(“element1”); 压入元素到队列尾部**

**删除元素**

**queue.poll(); //弹出队列头部元素**

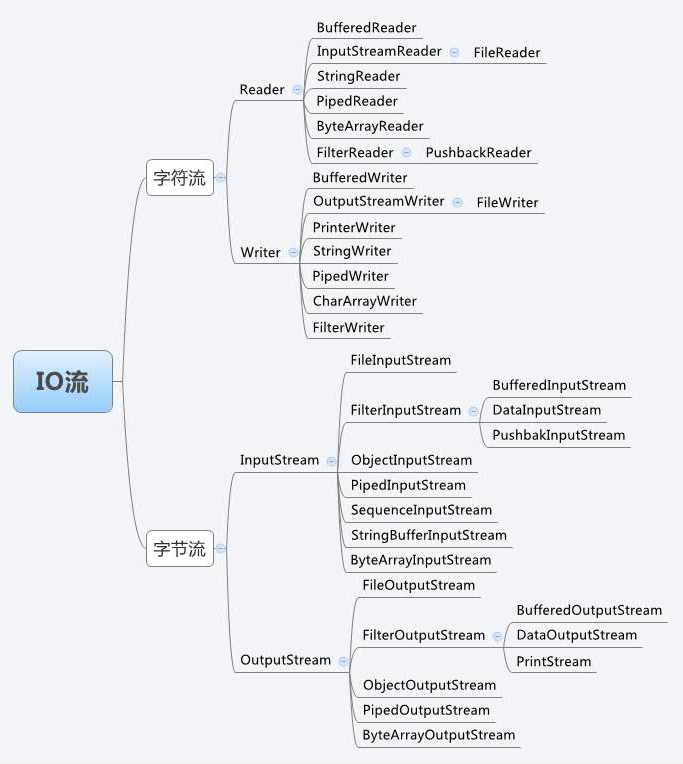
**查找元素**

**String val1 = queue.peek(); //获取队列头部元素**

**判断是否为空**

queue.isEmpty();

**7. IO操作**



**7.1. 字符流**

**FileWriter和FileReader提供文件读写**

**BufferedWriter和BufferedReader提供缓冲区功能**

**文件写入**

**FileWriter fileWriter = null;**

**try {**

**String filePath = “/home/lanling/document.txt”;**

**fileWriter = new FileWriter(filePath); //若new FileWriter(filePath, true)表示追加数据到文件**

**fileWriter.write(“hello world”);**

**fileWriter.flush();**

**} catch (IOException e) {**

**e.printStackTrace();**

**} finally {**

**if (fileWriter != null) {**

**try {**

**fileWriter.close();**

**} catch (IOException e) {**

**e.printStackTrace();**

**}**

**}**

**}**

**文件读取**

**FileReader fileReader = null;**

**try {**

**String filePath = “/home/lanling/document.txt”;**

**fileReader = new FileReader(filePath); //若new FileWriter(filePath, true)表示追加数据到文件**

**/\*\* 逐字符读取 \*\*/**

**int temp = 0;**

**while ((temp = fileReader.read()) != -1) {**

**System.out.print((char)temp);**

**}**

**/\*\* 读入字符数组 \*\*/**

**char[] buf = new char[1024];**

**int temp = 0;**

**while((temp = fileReader.read(buf)) != -1) {**

**System.out.print(new String(buf, 0, temp));**

**}**

**} catch (IOException e) {**

**e.printStackTrace();**

**} finally {**

**if (fileReader != null) {**

**try {**

**fileReader.close();**

**} catch (IOException e) {**

**e.printStackTrace();**

**}**

**}**

**}**

**7.2. 字节流**

**FileOutputStream和FileInputStream提供文件读写**

**BufferedOutputStream和BufferedInputStream提供缓冲区功能**

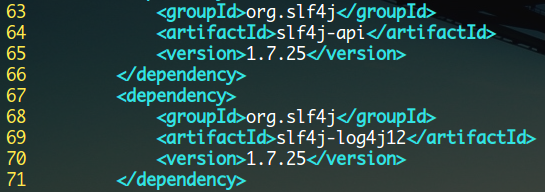
**8. 异常处理**

**aaa**

**9. 日志**

**Log4j是Java下的日志框架，是Apache的一个开源项目**

**Maven项目添加日志，Maven引入slf4j的包**



**log4j日志分为7个等级：ALL< DEBUG < INFO < WARN < ERROR < FATAL < OFF，等级由低到高，等级限制日志输出，只有等于或高于配置文件中设置的等级，日志才会输出**

* **ERROR 严重错误，主要是程序的错误**
* **WARN 一般警告**
* **INFO 一般信息，如用于记录状态**
* **DEBUG 程序调试信息**

**log4j配置文件有两种形式，properties文件和xml文件，两种等效**

**在.../src/main/resources/目录下创建log4j.properties文件，并添加如下内容**

**log4j.rootLogger设置日志等级和日志输出形式：**

* **org.apache.log4j.ConsoleAppender输出到控制台**
* **org.apache.log4j.FileAppender 输出到文件**
* **org.apache.log4j.RollingFileAppender 输出到文件，当文件大小到达指定值时，将会自动滚动，将原来的内容移到ner\_predict\_rolling.log.1中**
* **org.apache.log4j.DailyRollingFileAppender 每天产生一个日志文件**
* **org.apache.log4j.WriterAppender 将日志信息以流格式发送到指定地方**

**log4j.rootLogger设置全局的日志配置，若想对某个类进行单独设置，应将其更改为log4j.logger.com.xiaomi.assistant.NerPredict，其中com.xiaomi.assistant.NerPredict指定类的路径**

**log4j.appender.\*\*\*.Threshold设置日志信息最低输出级别，默认值是DEBUG**

**log4j.appender.\*\*\*.ImmediateFlush设置日志信息是否立即输出，默认值是true**

**log4j.appender.console.Target设置控制台信息输出**

**log4j.appender.\*\*\*.Append设置日志信息以追加或覆盖到指定文件中，默认是true追加**

**log4j.appender.\*\*\*.File设置输出日志文件路径**

**log4j.appender.rollingFile.MaxFileSize设置滚动日志文件到达该大小时，将会自动滚动，并将原内容移到ner\_predict\_rolling.log.1中**

**log4j.appender.rollingFile.MaxBackupIndex设置产生的滚动文件的最大数**

**log4j.appender.dailyFile.DatePattern设置每天产生一个新的日志文件，当前天的日志文件名为ner\_predict\_daily.log，过往的日志文件名为ner\_predict\_daily.log.yyyy-MM-dd**

**同时这里也是设置日志按每月、每周、每天、每小时、每分钟等来滚动：**

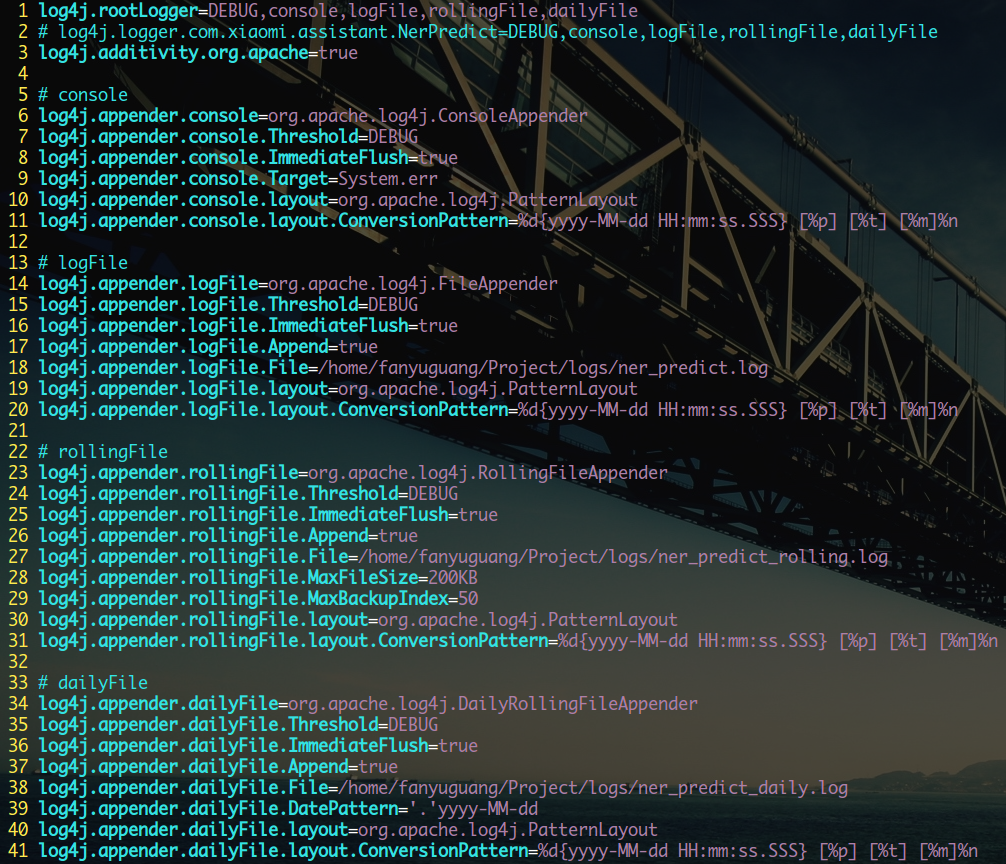
* **‘.’yyyy-MM 每月**
* **‘.’yyyy-WW 每周**
* **‘.’yyyy-MM-dd 每天**
* **‘.’yyyy-MM-dd-a 每天两次**
* **‘.’yyyy-MM-dd-HH 每小时**
* **‘.’yyyy-MM-dd-HH-mm 每分钟**

**log4j.appender.\*\*\*.layout设置日志文件内容及格式，有4中选择：**

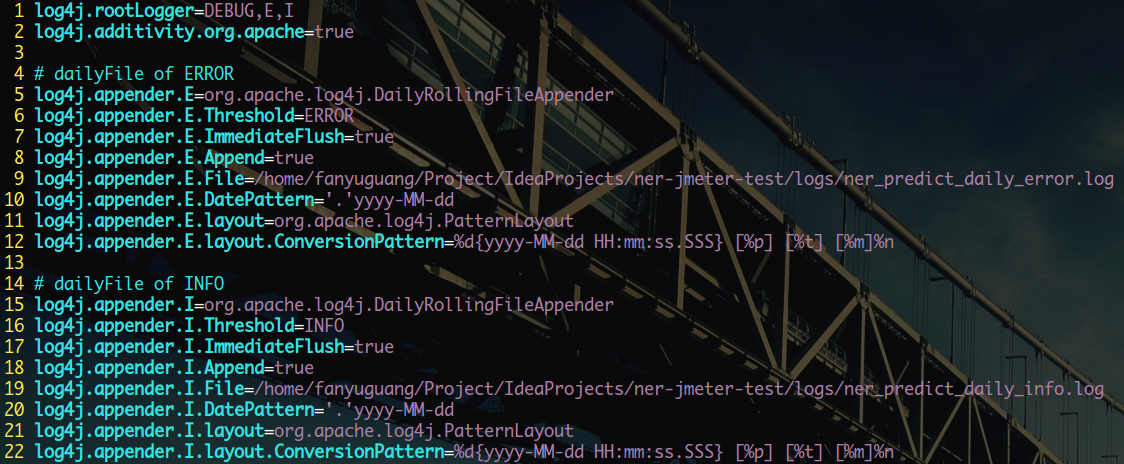
* **org.apache.log4j.HTMLLayout 以HTML表格形式布局**
* **org.apache.log4j.PatternLayout 灵活指定布局模式**
* **org.apache.log4j.SimpleLayout 仅包含日志级别和信息字符串**
* **org.apache.log4j.TTCCLayout 包含日志产生的时间、线程、类别等信息**

**log4j.appender.\*\*\*.layout.ConversionPattern设置每条日志的输出格式：**

* **%p：输出日志信息的优先级，即ALL, DEBUG, INFO, WARN, ERROR, FATAL, OFF**
* **%d：输出日志时间点的日期或时间，默认格式为ISO8601，可以指定格式如%d{yyyy-MM-dd HH:mm:ss.SSS}**
* **%r：输出自应用程序启动到输出该log信息耗费的毫秒数**
* **%t：输出产生该日志事件的线程名**
* **%l：输出日志事件的发生位置，相当于%c.%M(%F:%L)的组合，包括类全名、方法、文件名以及在代码中的行数**
* **%c：输出日志信息所属的类目，通常就是类全名**
* **%M：输出产生日志信息的方法名**
* **%F：输出日志消息产生时所在的文件名**
* **%L：输出代码中的行号**
* **%m：输出代码中指定的具体日志信息**
* **%n：输出一个回车换行符，Windows平台为"rn"，Unix平台为"n"**
* **%x：输出和当前线程相关联的NDC(嵌套诊断环境)**
* **%%：输出一个"%"字符**



**也可按下述配置将不同级别信息打到不同地方**



**在项目中打印日志，下述分别输出info信息和error信息**

**public class NerPredict {**

**private static final Logger logger = LoggerFactory.getLogger(NerPredict.class);**

**public function() {**

**logger.info("function start ...");**

**try {**

**...**

**catch (Exception e) {**

**logger.error(e.getMessage(), e);**

**}**

**}**

**}**

**10. 内存管理**

**Java在内存分配时会涉及到以下几个区域：**

**10.1. 栈**

**存放基本数据类型的变量和对象的引用变量。**

**Java会在栈内为变量分配内存空间，当变量离开作用域后，Java会自动释放掉该内存空间。**

**这里对象的引用变量为一个特殊变量，该变量的取值等于对象或数组在堆内存中的首地址，所以该变量为对象或数组的引用变量。**

**int a = 3;**

**int b = 3;**

**对于上面两行语句，编译器创建a并将3写入栈，当创建b时编译器会查找栈中是否存在3若存在则将b指向3，上面情况，a和b将指向栈中同一个3，当修改a为4时，编译器会查找栈中是否有4，若存在，则将a指向该位置，若不存在则在栈中创建4，并且这里不会影响b的值。**

**String str = “abc”;**

**String str = new String(“abc”);**

**第一种方法将在栈中创建一个”abc”的引用变量，并将”abc”存于字符串常量池中，若此时常量池中存在”abc”，则令str直接指向该地址。**

**第二种方法将在堆中创建一个对象，并且每new一次都会创建一个全新的对象。**

**测试两个包装类的数值是否相等用equals()方法，测试两个包装类的引用是否指向同一个对象时用==。**

**优势：存取速度较快，仅次于寄存器，栈数据可以共享。**

**缺点：存在栈中数据的大小和生存期必须是确定的。**

**10.2. 堆**

**存放new产生的对象和数组。在堆中分配的内存由Java虚拟机中的自动垃圾回收器管理。**

**当对象或数组没有引用变量指向它时，将定义为垃圾，在随后的一个不确定时间由自动垃圾回收器释放该内存。**

**通过new方法创建的对象，无论堆中是否存在相同值的对象，编译器将每次在堆中生成一个新的对象。**

**优势：可以运行时动态分配内存大小。**

**缺点：存取速度较慢。**

**10.3. 静态域**

**存放对象中用static修饰的静态成员**

**10.4. 常量池**

**存放常量**

**10.5. 寄存器**

**程序中无法控制**

**10.6. 非RAM存储**

**硬盘永久存储空间**

**11. 并发**

**aaa**

**12. 泛型**

**aaa**

**13. 注解**

**aaa**

**14. 编程规范**

子程序一般情况下代码长度不宜超过200行

一、命名规则

* 文件：
* 名字空间：

二、文件操作

打开或创建文本