# 一 》集合

关键点：

一图

一类

六接口

# 1 Collection

## 1.1 一图

接口Collection

接口List

接口Map

接口Set

ArrayList

LinkedList

HashSet

HashMap

TreeMap

## 1.2 一类 Collections工具类

## 1.3 六接口

Collection List Set Map / Iterable Comparater

## 1.4 部分底层实现

### 1.4.1 ArrayList底层 ： 数组

package com.example.demo;  
  
import java.util.Arrays;  
  
public class MyList {  
  
 //默认容积  
 private static final int *DEFAULT\_CAPACITY* = 10;  
  
 //空数组  
 private static final Object[] *EMPTY\_ELEMENTDATA* = {};  
  
 //真正操作的数组  
 public Object[] elementData;  
  
 //表示容器中当前成员数  
 int size;  
  
  
 public MyList(){  
 this.elementData = *EMPTY\_ELEMENTDATA*;  
 }  
  
 */\*\*  
 \* 新增数据  
 \** ***@param*** *o  
 \** ***@return*** *\*/* public boolean add(Object o){  
 //判断是容积大小是否还在范围内  
 ensureCapaCityInter(size+1);  
 //size加一，并存入新数据o  
 elementData[size++] = o;  
 return true;  
 }  
  
  
 */\*\*  
 \* 判断容积是否在范围内  
 \** ***@param*** *minCapacity  
 \*/* public void ensureCapaCityInter(int minCapacity){  
 ensureCapaCity(calculateCapacity(elementData,minCapacity));  
 }  
  
 */\*\*  
 \* 预估数组的容积：  
 \* 若为空则返回较大值  
 \* 若不为空，则返回最小新容积  
 \** ***@param*** *obj  
 \** ***@param*** *calculateSize  
 \** ***@return*** *\*/* public int calculateCapacity(Object[] obj, int calculateSize){  
 if(obj == *EMPTY\_ELEMENTDATA*){  
 return Math.*max*(*DEFAULT\_CAPACITY*,calculateSize);  
 }  
 return calculateSize;  
 }  
  
 */\*\*  
 \* 如果，需要的最小容积 大于 现有容积，则需要进行扩容  
 \** ***@param*** *minCapacity  
 \*/* public void ensureCapaCity(int minCapacity){  
  
 if(minCapacity - elementData.length > 0){  
 grow(minCapacity);  
 }  
  
 }  
  
 */\*\*  
 \* 数组扩容  
 \** ***@param*** *minCapacity  
 \*/* public void grow(int minCapacity){  
 //获取旧的容积  
 int oldCapa = elementData.length;  
 //新的容积= 旧容积的1.5 倍  
 int newCapa = oldCapa + (oldCapa >> 1);  
 //在新容积和最小容积中取最大值  
 if(newCapa < minCapacity){  
 newCapa = minCapacity;  
 }  
 //在创建一个新的数组，并copy旧的数据内容，并赋予新的容积  
 elementData = Arrays.*copyOf*(elementData, newCapa);  
 }  
  
*/\*\*--------------------------------------------------------------\*/  
 /\*\*  
 \* 返回存储数据个数  
 \** ***@return*** *\*/* public int size(){  
 return size;  
 }  
*/\*\*--------------------------------------------------------------\*/* public Object get(int index){  
 return elementData[index];  
 }  
  
*/\*\*--------------------------------------------------------------\*/* public Object remove(int index){  
  
 //需要判断index是否溢出 //*TODO* Object oldValue = elementData[index];  
  
 int numMoved = size - index - 1;  
 if (numMoved > 0)  
 System.*arraycopy*(elementData, index+1, elementData, index, numMoved);  
 elementData[--size] = null; // clear to let GC do its work  
 return oldValue;  
 }  
  
*/\*\*--------------------------------------------------------------\*/* public Object[] toArray() {  
 return Arrays.*copyOf*(elementData, size);  
 }  
*/\*\*--------------------------------------------------------------\*/* public Object set(int index, Object element) {  
   
 Object oldValue = elementData[index];  
 elementData[index] = element;  
 return oldValue;  
 }  
}

### 1.4.2 HashSet底层 ：HashMap

package com.example.demo;  
  
import java.util.HashMap;  
import java.util.Iterator;  
  
public class MySet {  
  
 private static final Object *PRESENT* = new Object();  
   
  
  
 HashMap<Object,Object> map = new HashMap<>();  
  
 public void add(Object obj){  
 System.*out*.println(*PRESENT*);  
 map.put(obj,*PRESENT*);  
 }  
  
 public Iterator<Object> iterator() {  
 return map.keySet().iterator();  
 }  
  
 public boolean remove(Object o) {  
 System.*out*.println(*PRESENT*);  
 return map.remove(o)==*PRESENT*;  
 }  
  
  
  
  
  
  
 public static void main(String[] args) {  
 MySet set = new MySet();  
 set.add("张三");  
 set.add("李四");  
 set.add("王五");  
 set.add("赵六");  
  
 Iterator<Object> iter = set.iterator();  
 while (iter.hasNext()){  
 System.*out*.println(iter.next());  
 }  
 System.*out*.println("===============");  
 set.remove("张三");  
 Iterator<Object> iter2 = set.iterator();  
 while (iter2.hasNext()){  
 System.*out*.println(iter2.next());  
 }  
  
 }  
  
  
  
  
  
}

### 1.4.3 HashMap底层 ：Node+数组

package com.example.demo.map;  
  
import java.util.Map;  
import java.util.Objects;  
  
public class MyMap {  
  
  
 public static void main(String[] args) {  
  
 MyMap map = new MyMap();  
 map.put("name","张三");  
 map.put("age",18);  
 map.put("addr","北极");  
 System.*out*.println(map.get("name"));  
 System.*out*.println(map.get("age"));  
 System.*out*.println(map.get("addr"));  
 }  
  
 int capacity;  
 int size = 0;  
  
 Node<Object,Object>[] nodes = (Node<Object,Object>[])new Node[837370700];;  
  
 public void put(Object key,Object value){  
 int hash = *hashCode*(key);  
 Node<Object,Object> node = new Node<>(key,value);  
 nodes[hash] = node;  
 }  
  
  
 public static int hashCode(Object key){  
 return Objects.*hashCode*(key);  
 }  
  
 public Object get(Object key){  
 int hash = Objects.*hashCode*(key);  
 if(nodes[hash] != null){  
 return nodes[hash].value;  
 }  
 return null;  
 }  
  
  
 static class Node<K,V> implements Map.Entry{  
  
 Node<K,V> node;  
 Object key;  
 Object value;  
  
  
 Node(Object key,Object value){  
 this.key = key;  
 this.value = value;  
 }  
  
 public Object getKey() {  
 return this.key;  
 }  
  
 public Object getValue() {  
 return this.value;  
 }  
  
 public Object setValue(Object value) {  
 Object oldValue = value;  
 this.value = value;  
 return oldValue;  
 }  
 }  
  
  
  
}

## 1.5 map的遍历方法：

### 方法一：

Map<String, String> map = jedisClient.hgetAll(TT\_CART\_REDIS\_PRE\_KEY+":"+userId);

// 判断是否为空

**if** (map == **null** || map.size()==0) {

**return** **null**;

}

//如果不为空

Set<Entry<String, String>> entrySet = map.entrySet();

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

**for**(Entry<String, String> temp:entrySet) {

TbItem item = JsonUtils.*jsonToPojo*(temp.getValue(), TbItem.**class**);

}

}

## 1.6 list集合保存数据的移除问题：

|  |
| --- |
| **package** com.taotao.cart.controller;  **import** java.util.ArrayList;  **import** java.util.Iterator;  **import** java.util.List;  **import** com.taotao.manager.pojo.TbItem;  /\*\*  \* List 集合中保存数据的修改和移除  \* -----------------测试一-------------------  \* \* 测试使用增强for循环删除和修改List保存对象  \* 1 移除list内的对象：  \* 1.1 循环删除多个对象时：报错java.util.ConcurrentModificationException  \* 1.2 只删除一个对象，然后使用break停止循环，则不报错  \* 2 修改内容不报错  \* -----------------测试二-------------------  \* \* 测试使用普通for循环删除和修改List保存对象  \* 1 移除list内的对象：  \* 1.1 升序循环  \* 1.1.1 循环删除多个对象时：不报错，但是数据有较大误差  \* 1.1.2 只删除一个对象，然后使用break停止循环:则不报错，且数据正确  \* 1.2 降序循环  \* 删除多个或一个都不报错，且数据准确  \* 2 修改内容不报错  \*  \* -----------------测试三-------------------  \* \* 测试使用Iterator循环删除和修改List保存对象  \* 1 使用list移除其内的对象：  \* 1.1 移除多个 :报错java.util.ConcurrentModificationException  \* 1.2 移除一个 :不报错，数据正常  \* 2 使用Iterator移除list中的对象：  \* 2.1 移除多个 :正确不报错  \* 2.2 移除一个:正确不报错  \* 3 修改内容不报错: 正确不报错  \* **@Description**  \* **@ClassName** Tes  \* **@Author** [浅醉]  \* **@Date** 2019年2月27日 下午2:12:07  \*/  **public** **class** Tes {  **public** **static** **void** main(String[] args) {  //test1();  //test2();  *test3*();  }  /\*\*  \* 测试使用增强for循环删除和修改List保存对象  \* 1 移除list内的对象：  \* 1.1 循环删除多个对象时：报错java.util.ConcurrentModificationException  \* 1.2 只删除一个对象，然后使用break停止循环，则不报错  \* 2 修改内容不报错  \*  \*/  **public** **static** **void** test1(){  List<Person> list = **new** ArrayList();  Person p1 = **new** Person("张三", 1);  Person p2 = **new** Person("李四", 2);  Person p3 = **new** Person("王二", 3);    list.add(p1);  list.add(p2);  list.add(p3);  System.***out***.println(list.size());  **for**(Person temp : list){      /\*-----------1.1 循环删除多个数据报错------------\*/  //list.remove(temp); 循环删除多个数据报错    /\*-----------1.2 删除一个数据，然后停止循环，则不报错------------\*/  **if**(temp.getName().equals("李四")){  list.remove(temp);  **break**;  }  /\*-----------1.3 修改数不报错------------\*/  //temp.setName("liili");  }  **for**(Person temp : list){  System.***out***.println(temp.getName());    }  System.***out***.println(list.size());  }  /\*\*  \* 测试使用普通for循环删除和修改List保存对象  \* 1 移除list内的对象：  \* 1.1 升序循环  \* 1.1.1 循环删除多个对象时：不报错，但是数据有较大误差  \* 1.1.2 只删除一个对象，然后使用break停止循环:则不报错，且数据正确  \* 1.2 降序循环  \* 删除多个或一个都不报错，且数据准确  \* 2 修改内容不报错  \*  \*/  **public** **static** **void** test2(){  List<Person> list = **new** ArrayList();  Person p1 = **new** Person("张三", 1);  Person p2 = **new** Person("李四", 2);  Person p3 = **new** Person("王二", 3);    list.add(p1);  list.add(p2);  list.add(p3);  System.***err***.println("操作前list集合大小："+list.size());    /\*----------1.1.1 循环删除多个对象时：不报错，但是数据有较大误差------------\*/    /\*for(int i = 0;i<list.size();i++){  System.out.println("list的实时大小："+list.size());  System.out.println("被删除的数据："+list.get(i).getName());  list.remove(i);  }\*/  /\* 结果： 并没有全部删除，仍剩下一个数据 李四。  \* 是因为在循环删除时list集合大小变化了，而且数据的下标也变化了:  \* 张三的下标是0，当删除后李四的下标变成了0；王二的变成了1 ，故删除了王二，留下了李四。这是有错误的。  3  被删除的数据：张三  被删除的数据：王二  1  \*/    /\*----------1.1.2 只删除一个对象，然后使用break停止循环，则不报错，且数据正确------------\*/    /\*for(int i = 0;i<list.size();i++){  if(list.get(i).getName().equals("李四")){  System.out.println("被删除的数据："+list.get(i).getName());  list.remove(i);  break;  }  }\*/    /\*----------1.2.1 降序循环 :循环删除多个数据，不报错，数据准确------------\*/  /\*for(int i = list.size()-1;i>-1;i--){  System.out.println("list的实时大小："+list.size());  System.out.println("被删除的数据："+list.get(i).getName());  list.remove(i);  }  \*/    /\*----------1.2.2 降序循环 :删除一个数据，break。不报错，数据准确------------\*/  **for**(**int** i = list.size()-1;i>-1;i--){  **if**(list.get(i).getName().equals("李四")){  System.***out***.println("被删除的数据："+list.get(i).getName());  list.remove(i);  **break**;  }  }        System.***err***.println("操作后list集合大小："+list.size());  }  /\*\*  \* 测试使用Iterator循环删除和修改List保存对象  \* 1 使用list移除其内的对象：  \* 1.1 移除多个 :报错java.util.ConcurrentModificationException  \* 1.2 移除一个 :不报错，数据正常  \* 2 使用Iterator移除list中的对象：  \* 2.1 移除多个 :正确不报错  \* 2.2 移除一个:正确不报错  \* 3 修改内容不报错: 正确不报错  \*  \*/  **public** **static** **void** test3(){  List<Person> list = **new** ArrayList();  Person p1 = **new** Person("张三", 1);  Person p2 = **new** Person("李四", 2);  Person p3 = **new** Person("王二", 3);    list.add(p1);  list.add(p2);  list.add(p3);  System.***err***.println("操作前list集合大小："+list.size());    /\*----------1.1 使用list移除其内多个对象： java.util.ConcurrentModificationException------------\*/  /\*Iterator<Person> iterator = list.iterator();  int i = 0;  while(iterator.hasNext()){  Person person = iterator.next();  list.remove(person);  }\*/      /\*----------1.2 使用list移除其内一个对象：不报错，数据正常------------\*/  /\*Iterator<Person> iterator = list.iterator();  while(iterator.hasNext()){  Person person = iterator.next();  if(person.getName().equals("李四")){  list.remove(person);  break;  }  }\*/      /\*----------2.1 使用iterator移除其内多个对象：不报错，数据正常 ------------\*/  /\*Iterator<Person> iterator = list.iterator();  while(iterator.hasNext()){  System.out.println("list实施大小："+list.size());  Person person = iterator.next();  iterator.remove();  }\*/    /\*----------2.2 使用iterator移除其内一个对象：不报错，数据正常 ------------\*/  /\* Iterator<Person> iterator = list.iterator();  while(iterator.hasNext()){  Person person = iterator.next();  if(person.getName().equals("李四")){  iterator.remove();  break;  }  }\*/      /\*----------3 使用iterator循环是时，修改数据: 正确不报错------------\*/  Iterator<Person> iterator = list.iterator();  **while**(iterator.hasNext()){  Person person = iterator.next();  person.setName("lili");  }    Iterator<Person> iterator2 = list.iterator();  **while**(iterator2.hasNext()){  Person person2 = iterator2.next();  System.***out***.println(person2.getName());  }    System.***err***.println("操作后list集合大小："+list.size());  }  }  **class** Person{  **private** String name;  **private** Integer id;  **public** String getName() {  **return** name;  }  **public** **void** setName(String name) {  **this**.name = name;  }  **public** Integer getId() {  **return** id;  }  **public** **void** setId(Integer id) {  **this**.id = id;  }  **public** Person(String name, Integer id) {  **super**();  **this**.name = name;  **this**.id = id;  }  } |

## 1.7 list集合移除数据Java ConcurrentModificationException异常原因和解决方法

Java ConcurrentModificationException异常原因和解决方法

　　在前面一篇文章中提到，对Vector、ArrayList在迭代的时候如果同时对其进行修改就会抛出java.util.ConcurrentModificationException异常。下面我们就来讨论以下这个异常出现的原因以及解决办法。

### 一.ConcurrentModificationException异常出现的原因

　　先看下面这段代码：

|  |
| --- |
| public class Test {      public static void main(String[] args)  {          ArrayList<Integer> list = new ArrayList<Integer>();          list.add(2);          Iterator<Integer> iterator = list.iterator();          while(iterator.hasNext()){              Integer integer = iterator.next();              if(integer==2)                  list.remove(integer);          }      }  } |

 　　运行结果：



　　从异常信息可以发现，异常出现在checkForComodification()方法中。

　　我们不忙看checkForComodification()方法的具体实现，我们先根据程序的代码一步一步看ArrayList源码的实现：

　　首先看ArrayList的iterator()方法的具体实现，查看源码发现在ArrayList的源码中并没有iterator()这个方法，那么很显然这个方法应该是其父类或者实现的接口中的方法，我们在其父类AbstractList中找到了iterator()方法的具体实现，下面是其实现代码：

|  |
| --- |
| public Iterator<E> iterator() {      return new Itr();  } |

 　　从这段代码可以看出返回的是一个指向Itr类型对象的引用，我们接着看Itr的具体实现，在AbstractList类中找到了Itr类的具体实现，它是AbstractList的一个成员内部类，下面这段代码是Itr类的所有实现：

|  |
| --- |
| private class Itr implements Iterator<E> {      int cursor = 0;      int lastRet = -1;      int expectedModCount = modCount;      public boolean hasNext() {             return cursor != size();      }      public E next() {             checkForComodification();          try {          E next = get(cursor);          lastRet = cursor++;          return next;          } catch (IndexOutOfBoundsException e) {          checkForComodification();          throw new NoSuchElementException();          }      }      public void remove() {          if (lastRet == -1)          throw new IllegalStateException();             checkForComodification();            try {          AbstractList.this.remove(lastRet);          if (lastRet < cursor)              cursor--;          lastRet = -1;          expectedModCount = modCount;          } catch (IndexOutOfBoundsException e) {          throw new ConcurrentModificationException();          }      }        final void checkForComodification() {          if (modCount != expectedModCount)          throw new ConcurrentModificationException();      }  } |

 　　首先我们看一下它的几个成员变量：

　　cursor：表示下一个要访问的元素的索引，从next()方法的具体实现就可看出

　　lastRet：表示上一个访问的元素的索引

　　expectedModCount：表示对ArrayList修改次数的期望值，它的初始值为modCount。

　　modCount是AbstractList类中的一个成员变量

|  |
| --- |
| protected transient int modCount = 0; |

 　　该值表示对List的修改次数，查看ArrayList的add()和remove()方法就可以发现，每次调用add()方法或者remove()方法就会对modCount进行加1操作。

　　好了，到这里我们再看看上面的程序：

　　当调用list.iterator()返回一个Iterator之后，通过Iterator的hashNext()方法判断是否还有元素未被访问，我们看一下hasNext()方法，hashNext()方法的实现很简单：

|  |
| --- |
| public boolean hasNext() {      return cursor != size();  } |

 　　如果下一个访问的元素下标不等于ArrayList的大小，就表示有元素需要访问，这个很容易理解，如果下一个访问元素的下标等于ArrayList的大小，则肯定到达末尾了。

　　然后通过Iterator的next()方法获取到下标为0的元素，我们看一下next()方法的具体实现：

|  |
| --- |
| public E next() {      checkForComodification();   try {      E next = get(cursor);      lastRet = cursor++;      return next;   } catch (IndexOutOfBoundsException e) {      checkForComodification();      throw new NoSuchElementException();   }  } |

 　　这里是非常关键的地方：首先在next()方法中会调用checkForComodification()方法，然后根据cursor的值获取到元素，接着将cursor的值赋给lastRet，并对cursor的值进行加1操作。初始时，cursor为0，lastRet为-1，那么调用一次之后，cursor的值为1，lastRet的值为0。注意此时，modCount为0，expectedModCount也为0。

　　接着往下看，程序中判断当前元素的值是否为2，若为2，则调用list.remove()方法来删除该元素。

　　我们看一下在ArrayList中的remove()方法做了什么：

|  |
| --- |
| public boolean remove(Object o) {      if (o == null) {          for (int index = 0; index < size; index++)              if (elementData[index] == null) {                  fastRemove(index);                  return true;              }      } else {          for (int index = 0; index < size; index++)              if (o.equals(elementData[index])) {                  fastRemove(index);                  return true;              }      }      return false;  }      private void fastRemove(int index) {      modCount++;      int numMoved = size - index - 1;      if (numMoved > 0)          System.arraycopy(elementData, index+1, elementData, index,                  numMoved);      elementData[--size] = null; // Let gc do its work  } |

 　　通过remove方法删除元素最终是调用的fastRemove()方法，在fastRemove()方法中，首先对modCount进行加1操作（因为对集合修改了一次），然后接下来就是删除元素的操作，最后将size进行减1操作，并将引用置为null以方便垃圾收集器进行回收工作。

　　那么注意此时各个变量的值：对于iterator，其expectedModCount为0，cursor的值为1，lastRet的值为0。

　　对于list，其modCount为1，size为0。

　　接着看程序代码，执行完删除操作后，继续while循环，调用hasNext方法()判断，由于此时cursor为1，而size为0，那么返回true，所以继续执行while循环，然后继续调用iterator的next()方法：

　　注意，此时要注意next()方法中的第一句：checkForComodification()。

　　在checkForComodification方法中进行的操作是：

|  |
| --- |
| final void checkForComodification() {      if (modCount != expectedModCount)      throw new ConcurrentModificationException();  } |

 　　如果modCount不等于expectedModCount，则抛出ConcurrentModificationException异常。

　　很显然，此时modCount为1，而expectedModCount为0，因此程序就抛出了ConcurrentModificationException异常。

　　到这里，想必大家应该明白为何上述代码会抛出ConcurrentModificationException异常了。

　　关键点就在于：调用list.remove()方法导致modCount和expectedModCount的值不一致。

　　注意，像使用for-each进行迭代实际上也会出现这种问题。

### 二.在单线程环境下的解决办法

　　既然知道原因了，那么如何解决呢？

　　其实很简单，细心的朋友可能发现在Itr类中也给出了一个remove()方法：

|  |
| --- |
| public void remove() {      if (lastRet == -1)      throw new IllegalStateException();         checkForComodification();        try {      AbstractList.this.remove(lastRet);      if (lastRet < cursor)          cursor--;      lastRet = -1;      expectedModCount = modCount;      } catch (IndexOutOfBoundsException e) {      throw new ConcurrentModificationException();      }  } |

 　　在这个方法中，删除元素实际上调用的就是list.remove()方法，但是它多了一个操作：

|  |
| --- |
| expectedModCount = modCount; |

 　　因此，在迭代器中如果要删除元素的话，需要调用Itr类的remove方法。

　　将上述代码改为下面这样就不会报错了：

|  |
| --- |
| public class Test {      public static void main(String[] args)  {          ArrayList<Integer> list = new ArrayList<Integer>();          list.add(2);          Iterator<Integer> iterator = list.iterator();          while(iterator.hasNext()){              Integer integer = iterator.next();              if(integer==2)                  iterator.remove();   //注意这个地方          }      }  } |

### 三.在多线程环境下的解决方法

　　上面的解决办法在单线程环境下适用，但是在多线程下适用吗？看下面一个例子：

|  |
| --- |
| public class Test {      static ArrayList<Integer> list = new ArrayList<Integer>();      public static void main(String[] args)  {          list.add(1);          list.add(2);          list.add(3);          list.add(4);          list.add(5);          Thread thread1 = new Thread(){              public void run() {                  Iterator<Integer> iterator = list.iterator();                  while(iterator.hasNext()){                      Integer integer = iterator.next();                      System.out.println(integer);                      try {                          Thread.sleep(100);                      } catch (InterruptedException e) {                          e.printStackTrace();                      }                  }              };          };          Thread thread2 = new Thread(){              public void run() {                  Iterator<Integer> iterator = list.iterator();                  while(iterator.hasNext()){                      Integer integer = iterator.next();                      if(integer==2)                          iterator.remove();                  }              };          };          thread1.start();          thread2.start();      }  } |

 　　运行结果：



　　有可能有朋友说ArrayList是非线程安全的容器，换成Vector就没问题了，实际上换成Vector还是会出现这种错误。

　　原因在于，虽然Vector的方法采用了synchronized进行了同步，但是实际上通过Iterator访问的情况下，每个线程里面返回的是不同的iterator，也即是说expectedModCount是每个线程私有。假若此时有2个线程，线程1在进行遍历，线程2在进行修改，那么很有可能导致线程2修改后导致Vector中的modCount自增了，线程2的expectedModCount也自增了，但是线程1的expectedModCount没有自增，此时线程1遍历时就会出现expectedModCount不等于modCount的情况了。

　　因此一般有2种解决办法：

　　1）在使用iterator迭代的时候使用synchronized或者Lock进行同步；

　　2）使用并发容器CopyOnWriteArrayList代替ArrayList和Vector。

# 二 》异常

# 1异常

三大重点：

异常分类 ：Throwable Error Exception (RuntimeException \ 编译时异常 )

异常处理方法：1 自己处理（try catch finally | throws throw）

2 交个JVM处理

自定义异常 ：继承Exception 继承RuntimeException

## 2.1 异常分类

编译时异常—运行时异常（RuntimeException）

Throwable

Exception

Error

运行时异常

编译时异常

崩溃异常

必须

显示处理

不强制

显示处理

Throwable类主要有两个子类：Error类、Exception类

　　Error异常类是系统异常，比如：虚拟机错误(VirtualMachineError)、线程死锁(ThreadDeath)，内存溢出，等，Error类异常一旦发生，程序将会崩溃

　　Exception是开发中我们最常见的一般异常，这种异常原因可能是程序代码编写错误，环境问题，用户输入错误等异常

　　Exception异常一般分为：运行时异常(RuntimeException)和编译时异常，也称为非检查异常、检查异常；

　　非检查异常常见的有：输出空指针时的异常，数组下标越界异常，类型转换异常，算术异常(比如0作为分母)等，运行时异常会由Java虚拟机自动捕获，自动抛出，一般是我们写的代码本身有问题，需要改进我们的代码来解决

　　检查异常的原因有可能是：文件异常(不存在或者权限)、数据库连接异常、网络连接异常等，这种异常系统不会自动捕获，需要我们手动添加捕获处理的语句

异常超级接口：Throwable

getMessage() ：获取异常信息，返回字符串

toString() ：获取异常类名和异常信息，返回字符串

printStackTrace() ：打印异常类名和异常信息，以及异常出现在程序中的位置。（JVM默认就使用这种方式处理异常。）

### 2.1.1 区别：

所有的RuntimeException类及其子类的实例被称为运行时异常，其他的异常就是编译时异常

编译时异常：java程序必须显示处理，否则程序就会发生错误，无法通过编译。

运行时异常：无需显示处理，也可以和编译时异常一样处理。

编译时异常：比如涉及到外部资源时，会让处理异常。不处理编译不通过。

（未雨绸缪异常：在做事时，先做好出现坏结果时的处理方法）

如：FileInputStream fs = new FileInputStream(“my.file”); -🡪会让进行异常处理

运行时异常：就是程序员所犯的错误，需要修改代码来解决。

## 2.2 JVM 默认是如何处理异常的

main函数收到这个问题时，有两种处理方式：

a. 自己将该问题处理，然后继续运行

b. 自己没有针对的处理方式，只有交给调用main的JVM来处理

JVM有一个默认的异常处理机制，就是将该异常进行处理。并将该异常的名称、信息、位置打印在控制台上，同时将程序停止运行。

## 2.3 自主处理异常的两种方式：

A : try… catch …finally ：处理异常后，程序继续向下执行

try{

try 用来检测异常

}catch(NullPointException e){

可以有多个catch

}catch(Exception e){

catch 用来捕获异常

}finally{

finally用来释放资源

如：关闭数据库、流

}

B : throws

定义功能方法时，需要把出现的问题暴露出来让调用者去处理。

那么就通过throws在方法上标识。

编译时异常：必须抛出

运行时异常：也可以不处理，不抛出

## 2.4 注意：

### 2.4.1 try…catch…

1)安卓，客户端开发：如何处理异常？ try{ }catch(Exception e) { }

2)ee, 服务端开发：一般都是底层开发，从底层向上抛。

3)try后跟多个catch时，要把小的异常放前边，大的异常放后边。如果大在前，将会接收所有小异常。

4）JDK 1.7 多个catch处理新特性

catch(NullPointExceptin | IndexOutOfRangeException e){ } 可以多个异常在一个里面

### 2.4.2 throws

#### 1） 编译时异常抛出

因为抛出的时Exception包含了编译时异常，所以要求必须显示的抛出处理。

public class ExceptionTest {  
 private int age;  
 public int getAge() {  
 return age;  
 }  
  
 public void setAge(int age) throws Exception {  
 if(age > 0 && age <= 150){  
 this.age = age;  
 }else{  
 //将异常向上抛出给调用该方法的的main方法处理  
 throw new Exception("年龄非法");  
 }  
 }  
  
 //throws 又将该异常继续上抛出给调用这个main方法的JVM处理

//throws 标识，标识提醒JVM此处可能有异常  
 public static void main(String[] args) throws Exception {  
 ExceptionTest test = new ExceptionTest();  
 test.setAge(100);  
 }  
}

#### 2） 运行时异常抛出

下例是抛出的运行时异常，所以不要求进行处理。可以抛出也可以不抛出。

public class ExceptionTest {  
 private int age;  
 public int getAge() {  
 return age;  
 }  
  
 public void setAge(int age){  
 if(age > 0 && age <= 150){  
 this.age = age;  
 }else{  
 //因为抛出的是 运行时异常 不要求必须处理，所以可以抛出也可以不抛出处理。  
 throw new RuntimeException("年龄非法");  
 }  
 }  
  
 public static void main(String[] args){  
 ExceptionTest test = new ExceptionTest();  
 test.setAge(100);  
 }  
}

#### 3） throw 和 throws的区别

throws ： 定义功能方法时，需要把出现的问题暴露出来让调用者去处理。

throw 的概述：

在功能方法内部出现某种情况，程序不能继续进行。需要进行跳转时，就用throw把异常对象抛出。

区别：

throws ：

用在方法声明后面，跟的是异常类名

可以跟多个异常类名，并用逗号隔开（无先后顺序）

表示抛出异常，由该方法的调用者来处理

throw：

用在方法内部，跟的是异常对象。

只能抛出一个异常对象

表示抛出异常，有方法内部的语句处理。

### 2.4.3 finally

#### 1） finally的特点

==被finally控制的语句体一定会执行

==特殊情况：在执行到finally之前，JVM退出了（比如：System.exit(0)）

#### 2) finally的作用：

==用于释放资源，在IO流操作中和数据库操作中常见

#### 3） finally 和 final / finalize的区别

== final修饰类 、方法、变量

final修饰的类不能被继承，方法不能被重写，变量只能赋值一次。

== finally 是try语句中的一个语句体，不能单独使用，用于释放资源

== finalize 是一个方法。当垃圾回收器确定不存在对某对象的更多应用时，由对象回收 器调用该对象的该方法，回收该对象。

#### 4） 如果catch里面有return语句，

请问finally的代码还会执行吗？

如果会，请问是在return前，还是后？

== 会执行。

== 在return 中间执行，return先开始操作，并没有彻底返回执行，而是在有finally的时候，执行完finally在进行彻底return。

1当： finally中没有return时 ，try或catch中有return？

在return中间，显示按照顺序执行，try或catch中的return一次，但是并没有真正的返回结果结束程序，而是已经把结果给了主函数（此后的操作不会对结果有影响），然后继续进行逻辑执行，即try-catch-finally的逻辑。最后会再一次执行try或catch中的return，返回上次给的结果，结束程序。

2 当：finall中有return时，无论try-catch什么情况？

如果catch中有return，则是catch先发起return，并将结果给到main方法，但并没有彻底结束。此时，再执行finally中的return，算是发起一次新的return，它的结果将覆盖旧的return结果。所以最终返回的时finally中的结果。

但是正常情况下，finally语句中是绝对不能写return返回语句的，因为它会覆盖try和catch中的结果。使他们毫无意义，就是犯罪啊。 他的真正作用是用来释放资源的。

3 另外，try语句块中能够抛出异常，然后执行catch语句；

结论：finally中的代码是在return 和break语句中间执行的。

## 2.5 自定义异常

### 2.5.1 为什么需要自定义异常？

==通过名字与其他的异常区分开来，便于异常问题的排查和处理。

### 2.5.2 如何自定义异常？

== 继承 Exception

== 继承 RuntimeException

重写他们的构造方法。

## 2.6 异常的注意事项及如何使用异常处理

### 1） 继承中的异常：抛出

== 子类重写父类方法时，子类的方法必须抛出相同的异常或父类异常的子类（父亲坏了，儿子不能比父亲更坏）

== 如果父类抛出了多个异常，子类重写父类时，只能抛出相同的异常或是他的子类，子类不能抛出父类没有的异常。

== 如果被重写的方法没有异常抛出，那么子类绝不可以抛出异常，如果子类方法内由异常发生，那么子类只能try，不能throws

### 2） 如何使用异常处理

== 原则：如果该功能内部可以将问题处理，用try；处理不了，交由调用者处理，，用throws

== 区别：后续程序需要继续运行就try

后续程序不需要继续进行就throws （ throw会将异常抛给调用者并停止程序运行）

== 如果JDK没有提供对应的异常，需要自定义异常

# 三 》 IO流

关键点：

流：

字节流

字符流

转换流

其他流

注意事项：

输出流清空文件内容

缓存输出流缓存问题

字节流：

InputStream OutputStream

FileInputStream FileOutputStream

BufferedInputStream BufferedOutputStream

读写：

读：读的是字节

int i = in.read();

byte[] by = new byte[];

用int接收字节，是因为当read(）方法当读到-1时就会停止读取，而byte的-1字节可能被读取到，所以不能使用。

写：写的是字节

os.write(97); 结果是 a

字符流：

Reader Wirter

FileReader FileWriter

BufferedReader BufferedWriter

char c = fr.read()

注意问题：

BufferedReader 的readLine()方法，是以读取到\r\n回车符时结束的。如果读取不到就会一直阻塞读取。

BuffferedWrirer 的writer方法，是不写回车符的，所以要自己添加，newLine 或者\r\n

另外：LineNumberReader LineNumberWriter

转换流：

InputStreamReader OutputStreamWriter 使用该方法，可以采用指定的字符编码进行读写文件。

其他流：

数据操作流：DataInputStream DataOutputStream 可以读取或打印相应的类型，而不是转换成字节。

对象操作流：ObjectInputStream ObjectOutputStream 要想使用该流操作对象，则对象类必须实现序列化。

标准输入输出流：System.in System.out 默认是指键盘和控制台

打印流：PrintStream PrintWriter 可以使用println（）方法直接操作字符串。并进行换行。

序列流：SequenceInputStream 用处：串烧。 Eumeration<InputStream> 的获取 :Vector.elements(),

缓存流：ByteArrayOutputStream 将字节读入缓存中，最后获取。 baos.toString() baos.toByteArray();

随机访问流 :RandomAccessFile seek(),设置指针位置。

属性流： Properties load（） store（）

注意事项：

1 输出流问题：FileOutputStream fos = new FileOuputStream("a.txt");

a 创建输出流时，如果存在文件会将文件内容清空.如果将其跟在同一个文件的输入之后，则无法读取到任何消息，因为还没读取就被清空了。

FileInputStream fis = new FileInputStream（"a.txt"）; // 创建a.txt文件的输入流

FileOutputStream fos = new FileOuputStream("a.txt"); // 创建a.txt文件的输出流，此时会清空a.txt内容备用

int i = fis.read() ; 读取不到任何信息

如果不想清空，则可添加参数true：FileOutputStream fos = new FileOuputStream("a.txt",true); //表示追加

2 缓存输出流：

要记得刷新缓存，或关闭缓存，不然缓存中的信息，无法被打印出去。

# 1 IO流-字节流

I/O

InputStream

OutputStream

FileInputStream

FileOutputStream

BufferedInputStream

BufferedOutputStream

Reader

Writer

FileReader

FileWriter

BufferedReader

BufferedWriter

字节流

包装流

包装流

字符流

//1 创建输入流

FileInputStream fis = new FileInpuStream(“a.txt”)

//2 创建输出流

FileOutputStream fos = new FileOutputStream(“copy.txt”)

int byt ;

//3 读：不断循环读取每一个字节

while((byt = fis.read()) != -1 ){

//4 写：不断循环写入每一个字节

fos.write(byt);

}

//5 关闭输入流

fis.close();

//6 关闭输出流

fos.close();

读写六大步

InputStreamReader

OutputStreamWriter

转换流

## 3.1 File

### 3.1.1 File 概述：文件或文件夹的抽象路径

#### 1) file更应该叫做一个路径，而非一个文件

== 文件路径或文件夹的路径

== 路径分为绝对路径和相对路径

== 绝对路径是一个固定的路径，从盘符开始

== 相对路径是相对于某个位置，在eclipse下是指当前项目下，在dos（cmd）下指的是当前路径

#### 2）创建功能

== createNewFile() : 创建文件，如果存在这样的文件，就不创建了

== mkdir() ： 创建文件夹，如果存在这样的文件夹，就不创建了

== mkdirs() ： 创建文件夹，如果父文件夹不存在，会帮你创建出来

== 如果创建文件或文件夹是忘记了写盘符，则默认在项目路径下。

#### 3） 重命名和删除功能

== renameTo(File dest ) : 把文件重命名为指定的文件路径

== delete() ： 删除文件或文件夹

== 注意事项：

☛如果路径名相同，就是改名

☛如果路径名不同，就是改名并剪切

== 删除注意：

☛Java中的删除不走回收站

☛要删除一个文件夹，请注意该文件夹内不能包含文件或文件夹；

可以先遍历文件夹内文件并删除，写一个删除文件夹的方法。

#### 4） 判断功能

== isDirectory() : 是否是文件夹

== isFile() ：是否是文件

== exists() ：是否存在

== canRead() ：是否能读 ☛setReadable(true/false ) : 设置是否能读

（Windos系统认为所有的文件都是可读的，所以设置为false也会可读）

== canWrite() ：是否能写 ☛setWritable(true/false) : 设置是否能写

== isHidden() ：是否隐藏

#### 5） 获取功能

== getAbsolutePath( ) ：获取绝对路径

== getPath() ：获取路径 ☛构造方法中传入的路径（绝对路径/相对路径）

== getName() ：获取名称

== length() ：获取长度，字节数

== lastModified() ：获取最后一次的修改时间，毫秒值

== list() ：获取指定目录下的所有文件或文件夹的名称数组

== listFiles() ：获取指定目录下的所有文件或文件夹的File对象数组

#### 6） FilenameFilter 文件名过滤器

== list(FilenameFilter filter)

== listFiles(FilenameFilter filter)

public static void main(String[] args) throws IOException {  
 //创建要遍历的文件夹路径对象  
 File dir = new File("D:\\");  
 //遍历该文件夹  
 String[] files = dir.list(new FilenameFilter() {  
 @Override  
 public boolean accept(File dir, String name) {  
  
 //参数：dir是要遍历的文件夹的路径  
 //参数：name是遍历的文件名  
 //可以使用这两个参数获取文件夹下的文件的对象  
 File file = new File(dir,name);  
 //判断是否是文件，是否后缀为 .jpg  
 Boolean bool = file.isFile() && file.getName().endsWith(".jpg");  
 //返回该布尔值，是则通过，不是则拦截  
 return bool;  
 }  
 });  
  
 //输出文件名  
 for (String temp: files) {  
 System.*out*.println(temp);  
 }  
}

#### 7)FileFilter 文件过滤器

== listFiles(FileFilter filter)

public static void main(String[] args) throws IOException {  
 //创建要遍历的文件夹路径对象  
 File dir = new File("D:\\");  
  
 File[] files = dir.listFiles(new FileFilter() {  
 @Override  
 public boolean accept(File pathname) {  
 //参数：pathname是文件夹下的路径对象  
 //判断遍历的pathname是否是文件，并且后缀是否是.jpg  
 Boolean bool = pathname.isFile() && pathname.getName().endsWith(".jpg");  
 return bool;  
 }  
 });  
  
 //输出文件名  
 for (File temp: files) {  
 System.*out*.println(temp.getName());  
 }  
}

## 3.2 IO流

### 1）概念

== IO流用来处理设备之间的数据传输

== Java对数据的操作是通过流的方式

== java用于操作流的类都在IO包中

== 流按流向分为两种：

☛输入流

☛输出流

== 流按操作类型分为两种：

☛字节流：字节流可以操作任何数据，因为在计算机中任何数据都是以字节

的形式存储的。

☛字符流：字符流是对字节流的封装，只能操作纯字符数据，比较方便。

### 2）IO流常用的父类

== 字节流的抽象父类：

☛InputStream

☛OuputStream

== 字符流的抽象父类

☛Reader

☛Writer

### 3）IO程序的书写

== 使用前：导入IO包中的类

== 使用时：进行IO异常处理

== 使用后：释放资源

## 3.3 FileInputStream

### 1) 重要的方法解析 read( );

深入理解该方法，将使流的读取十分简单；

FileInputStream fis = new FileInputStream( path/File) ☛参数：文件路径/文件路径对象

== 流的读方法 read（）

int x = fis.read( )

☛该方法会从硬盘上读取一个字节，返回值就是字节数

该字节数就是数据在硬盘上的存储形式；

当内容读取完毕后，会返回 -1

比如：a-97 b-98 c-99

可以使用方法：Character.toChars(int x) 将字节转成字符

public static void test1() throws IOException {  
 //FileInputStream 构造参数：文件路径String 或 路径对象File  
 FileInputStream in = new FileInputStream("a.txt");  
  
 int x = in.read(); //从硬盘上读取一个字节：返回值是硬盘存储的字节数  
 System.*out*.println(x);  
 System.*out*.println(Character.*toChars*(x));

int y = in.read();  
 System.*out*.println(y);  
 System.*out*.println(Character.*toChars*(y));

int z = in.read();  
 System.*out*.println(z);  
 System.*out*.println(Character.*toChars*(z));

in.close(); //关闭流  
  
}

结果：

97

a

98

b

99

c

使用while循环打印整个内容：

public static void test2() throws IOException{  
 //FileInputStream 构造参数：文件路径String 或 路径对象File  
 FileInputStream in = new FileInputStream("a.txt");  
 int i; //声明一个int值，接收返回的 字节数  
 while ((i = in.read()) != -1){ //只要不为 -1 就继续循环输出  
 // System.out.print(i);  
 System.*out*.print(Character.*toChars*(i)); //输出值  
 }  
  
 //关闭流  
 in.close();  
  
}

结果：abc

### 2）为什么read( ) 返回值不是byte类型，而是int类型呢？？

== 因为字节输入流可以操作任意类型的文件，比如图片、音频等，这些文件底层都是

以二进制形式存储的，如果每次读取都返回byte，有可能在读取的时候遇到8个

1 ，那么整个11111111是byte类型的-1，我们的程序是遇到-1 就会停止不读了，

后面的数据就读不到了。所以在读取的时候用int类型接收，如果1111111会在其

前面补上24 个 0凑足4个字节，那么byte类型的-1 就变成了int类型的255，

这样就可以保证整个数据读完，而结束标记的-1 就是int类型。

一个字节是二进制8位

int是四个字节：8的4次方

硬盘数据-》四个字节： 00000001 00000010 11111111 00000000

== 如果是以byte类型接收 ，当读取到11111111 时，会获取x= -1，就停止读取内容

分析：运行时是以字节的补码方式运行:。所以获取 -1 补码

先获取反码，再有反码获取补码。

一个字节：

byte类型 -1 的原码 10000001

获取反码> 11111110 -1 的反码 （高位不变，其他位置取反1 🡪0 0🡪1 ）

获取补码> 11111111 -1 的补码 (末尾位置加1)

此处的补码运行出来是-1 ，也就是byte x = in.read( ) x= -1会直接导致文件读取停止。

== 如果以int类型接收：

int类型是四个字节，32 位。

每当读取一个字节时，都会自动在前面加上24个零进行补位，使之成为int类型

如读取 11111111 时：

会这样：00000000 00000000 00000000 11111111

运算得出结果会是255 ，而不是-1 ，保证继续取下去

== 在保存时，write方法，会自动去掉前面的补位，使之还原

## 3.4 FileOutputStream

FileOutputStream fos = new FileOutputStream(“b.txt”);

在创建对象时：

如果没有这个文件就会帮我们创建出来。

如果有这个文件，就会直接将文件清空。

### 1）重要方法：wirte()方法

== fos.write(97 )

☛: 虽然写出的是一个int数，但是存储到文件上的是一个字节byte，会自动

去除前三个8位。

public static void test1() throws IOException {  
 //参数是： 路径 或 路径对象  
 FileOutputStream fos = new FileOutputStream("b.txt");  
 //FileOutputStream fos2 = new FileOutputStream(new File("b.txt"));  
  
 //写入数据  
 fos.write(97); //虽然写出的是一个int数，但是到文件上的是一个字节，会自动去除前三个8位  
 fos.write(98);  
 fos.write(99);  
 //关闭流  
 fos.close();  
}

结果：b.txt 中 🡪abc

### 1） FileOutputStream 的追加功能

public FileOutputStream(String name, boolean append)

append 追加

FileOutputStream fos = new FileOutputStream(“b.txt”,true);

第二个参数：如果存在文件，则不清空，直接再后边续写内容。

## 3.5 拷贝文件：边读边写

一边读，一边写；

**重要的六行：**

public static void test2() throws IOException {  
 FileInputStream fis = new FileInputStream("a.jpg"); //创建输入流  
 FileOutputStream fos = new FileOutputStream("b.jpg"); //创建输出流  
  
 int byt;  
 while ((byt = fis.read()) != -1){ // 读内容：不断的读每一个字节  
 fos.write(byt); // 写内容：不断的写每一个字节  
 }  
  
 fis.close(); //关闭输入 ：释放资源  
 fos.close(); //关闭输出：释放资源  
}

## 3.6 问题： 效率低。

1KB = 1024B（byte）

== 每次都是读一个字节，写一个字节。 如果一个文件有1000个字节，就要循环1000次，取完成每个字节的读写。

解决：使用字节数组。

== 每次读取内容装到字节数组中，使得一次可以操作多个字节，提高效率。

byte[ ] bytes = new byte[1024];

int len = in.read(bytes);

☛该操作将内容读进byte数组中，一次最大读取1024个字节。返回值len就是实际每次读取的有效字节个数。

☛也就是，byte数组中有效字节的个数。

## 3.7 问题解决操作：byte数组

== 注意事项：

☛byte[ ] arr = new byte[1024\*n] 长度是1024的整数倍

☛len是读进byte中的有效字节的个数

☛fis.read(arr) 一定要写再while中，循环读；

另外，如果忘记读入arr中，len不再是有效字节个数，而是读取字节的码表值

☛fos.write(arr,0,len) 中三个参数一个不可少。

如果不写len，可能会多读出一些无效数据。

abc

程序

a

b

第一次读取

byte[ ] arr = new byte[2]

abc

第二次读取

c

b

如图，第一次读取时，有效字节数时2 ：a和b

第二次读取时，有效字节数时1 ：只有c （重写arr，但是b没有覆盖）

如果在fos.write(arr,0,len)中，不写0和len，会导致第二次写入

时会多一个b，即结果是：abcb

如果参数正确：则结果是：abc

public static void test4() throws IOException {  
 FileInputStream fis = new FileInputStream("a.jpg");  
 FileOutputStream fos = new FileOutputStream("b.jpg");  
  
 byte[] arr = new byte[1024];  
  
 int len;  
 while ((len = fis.read(arr)) != -1){ //一定要将读放在while内，循环读；千万别忘了读进arr数组中  
 fos.write(arr,0,len); // 三个参数必不可少，如果不写后边的参数，可能会多出一些无效数据  
 }  
 fis.close(); //关闭输入  
 fos.close(); //关闭输出  
  
}

## 3.8 字节包装流： 缓存流

BufferedInputStream BufferedOutputStream

### 1） 缓冲思想：

== 字节流一次读写一个数组的速度明显比一次读写一个字节的速度快很多

== 这是加入了数组这样的缓冲区效果，java本身在设计的时候，也考虑到

了这样的设计思想（装饰设计模式），所以提供了字节流缓存区流

### 2）BufferedInputStream

== BufferedInputStream 内置了一个数组（缓冲区）byte[] arr = new byte[8192]

== 从BufferedInputStream中读取一个字节时

== BufferedInputStream会一次性从文件中读8192个字节，存在数组（缓冲区）

中，返回给程序一个。程序再次读取时，就不用找文件了，直接从内存中的

数组（缓冲区）中获取，直到缓冲区中所有的都被使用过，才重新从文件中

读取8192个。

### 3）BufferedOutputStream

== BufferedOutputStream也内置了一个数组（缓冲区）byte[] arr = new byte[8192]

== 程序向流中写出字节时，不会直接写到文件中，先写到缓冲区中

== 当缓冲区写满后，BufferedOutputStream才会把缓冲区中的数据一次性写到

文件中。

### 4）图解

文件

文件

copy

int b = bis.read()

InputStream

OutStream

BufferedInputStream

BufferedOutputStream

BufferedOutputStream 内置byte[] arr = new byte[8192]

一次性向文件中写入8192个字节

BufferedInputStream 内置byte[] arr = new byte[8192]

一次性从文件中读取8192个字节

程序每次从BufferedInputStream中读取一个字节，并将该字节写入BufferedOutputStream，因为操作的内存中的数据，而不是硬盘上的文件，所以效率很快

程序

### 5）copy

public static void test5() throws IOException {  
 FileInputStream fis = new FileInputStream("a.jpg"); //输入流  
 FileOutputStream fos = new FileOutputStream("b.jpg"); //输出流  
  
 BufferedInputStream bis = new BufferedInputStream(fis); //输入缓冲流  
 BufferedOutputStream bos = new BufferedOutputStream(fos); //输出缓冲流  
  
 int byt;  
 while ((byt = bis.read()) != -1){ //不断从 BufferedInputStream的缓冲区读取每一个字节  
 bos.write(byt); //不断向 ufferedOutputStream的缓冲区写入每一个字节  
 }  
  
  
 bis.close(); //关闭输入流，只需关闭包装流  
 bos.close(); //关闭输出流  
  
}

### 6）close 和 flush 的区别

== close()方法具备刷新功能，在关闭流之前，会先刷新一次缓冲区，将缓冲区的有效字节全部刷新到文件上，再关闭流。

== flush也具备刷新功能，刷完之后还能继续写，但是close会刷新并关闭流，

就不能再写了。

## 3.9 字节流读写中文问题

1） 字节流在读取中文时，有可能会读到半个中文，造成乱码

== 一个中文占两个字节，一个逗号占一个字节

public static void test6() throws IOException {  
 FileInputStream fis = new FileInputStream("a.txt"); //输入流  
 byte[] byt = new byte[1024];  
 int len;  
 while ((len = fis.read(byt)) != -1){  
 String str = new String(byt,0,len);  
 System.*out*.println(str);  
 }

fis.close();

}

2） 字节流写出中文的问题：写不会乱码，但必须转为字节数组

== 字节流直接操作的字节，所以写出的中文必须将字符串转换成字节数组

== 写出回车换行 write(“\r\n”.getBytes());

public static void test7() throws IOException {  
 FileOutputStream fos = new FileOutputStream("a.txt"); //输出流  
 fos.write("你好啊".getBytes()); //获取字节数组  
 fos.close(); //关闭流  
  
}

## 3.10 JDK 1.6 之前IO流的标准操作

public static void test8() throws IOException { //throws抛出异常  
  
 FileInputStream fis = null; //输入流声明，扩大作用域  
 FileOutputStream fos = null; //输出流声明，扩大作用域  
  
 try{  
 fis = new FileInputStream("a.jpg"); //输入流  
 fos = new FileOutputStream("b.jpg"); //输出流  
  
 byte[] bytes = new byte[8192]; //创建容器  
 int len;  
 while ((len = fis.read(bytes)) != -1){ //读取内容  
 fos.write(bytes,0,len); //输出内容  
 }  
  
 }finally { //关闭流  
 try{ //双层finally嵌套，是希望两个尽量能关一个是一个;如：fis关闭异常，fos也能正常关  
 if(fis != null){ //判断fis是否赋值成功，若没成功就是null，不需要关闭。  
 fis.close(); //若没成功就是null，直接调用方法会抛空指针异常,所以要判断  
 }  
 }finally {  
 if(fos != null){ //判断fos是否赋值成功，若没成功就是null，不需要关闭。  
 fos.close(); //若没成功就是null，直接调用方法会抛空指针异常,所以要判断  
 }  
 }  
 }

## 3.11 JDＫ1.7 后标准操作

**//该方式，会在{}中的语句执行完毕后，自动调用（）中对象的close（）方法**

public static void test9() throws IOException {  
 try(  
 FileInputStream fis = new FileInputStream("a.jpg"); //创建输入流  
 FileOutputStream fos = new FileOutputStream("b.jpg"); //创建输出流  
 ){  
 byte[] bytes = new byte[8192]; //创建容器  
 int len;  
 while ((len = fis.read(bytes)) != -1){ //读取内容  
 fos.write(bytes,0,len); //输出内容  
 }  
 }  
 //该方式，会在{}中的语句执行完毕后，自动调用（）中对象的close（）方法  
}

分析：看源码

public class FileInputStream extends InputStream{}

public abstract class InputStream implements Closeable {}

public interface Closeable extends AutoCloseable {}

public interface AutoCloseable {

void close() throws Exception;  
}

IO流都实现了一个接口Closeable ,它的顶级接口是AutoCloseable

所有实现了该接口的类，都能拥有上述的特性。

即：**在{}中的语句执行完毕后，自动调用（）中对象的close（）方法**

## 3.12 图片加密：异或

如下：加密后，获取的copy.jpg无法打开；解密后的copy2.jpg能正常打开

加密：

public static void test10() throws IOException {  
 try(  
 BufferedInputStream bis = new BufferedInputStream(new FileInputStream("a.jpg")) ; //创建输入流  
 BufferedOutputStream bos = new BufferedOutputStream(new FileOutputStream("copy.jpg")); //创建输出流  
 ){  
 int byt;  
 while ((byt = bis.read()) != -1){ //读取内容  
 bos.write(byt ^ 123); //输出内容 : 一个数被某个数异或一次，相当于加密；再次被该数异或相当于解密  
 }  
 }  
}

解密：

public static void test10() throws IOException {  
 try(  
 BufferedInputStream bis = new BufferedInputStream(new FileInputStream("copy.jpg")) ; //创建输入流  
 BufferedOutputStream bos = new BufferedOutputStream(new FileOutputStream("cop2.jpg")); //创建输出流  
 ){  
 int byt;  
 while ((byt = bis.read()) != -1){ //读取内容  
 bos.write(byt ^ 123); //输出内容 : 一个数被某个数异或一次，相当于加密；再次被该数异或相当于解密  
 }  
 }  
}

# 2 IO流-字符流

Reader ->InputStreamReader ->FileReader

->BufferedReader

Writer ->OutputStreamWriter ->FileWriter

->BufferedWriter

Reader

InputStreamReader

BufferedReader

FileWriter

Writer

OutputStreamWriter

BufferedWriter

FileReader

转换流

转换流

缓冲流

缓冲流

读文件

写文件

图解：

copy

文件

Writer FileWriter

Reader FileReader

程序

c = br.read(ch)

BufferedWriter

BufferedReader

char[] ch = new char[1024\*8] 因为操作的是字符，所以用字符数组接收，不能再使用字节数组接收

原理图解：

copy

文件

操作

Writer

Reader

读文件时：将字节流自动包装成字符流

写文件时：将字符流自动转成字节流

InputStream

OutputStream

## 4.1 Writer

== Writer类中有一个2k的小缓冲区，如果不关流，就会将内容留在缓冲区中，关流会

先刷新缓存区，在关闭流。

## 4.2 什么情况下使用字符流？

== 字符流也可以拷贝文本文件，但是不推荐使用，因为读取时会把字节转为字符，写

出时，还要把字节转回字符。效率低。所以拷贝推荐使用字节流。

== 只读/只写时使用：程序需要读取一段文本，或者需要写出一段文本的时候，可以使

用字符流。

## 4.3 字符流可否拷贝非纯文本文件？

== 不可以拷贝非纯文本文件。如：图片、音频。

== 因为在读的时候，会将字节转换为字符，在转换过程中，可能找不到对应的字符，

就会用？代替；

写的时候会将字符转换成字节，如果时？，则直接将？写出，导致写出的文件乱码。

## 4.4 字符数组接收

public static void test2() throws IOException {  
   
 FileReader fr = new FileReader("a.txt"); //1 创建输入流  
 FileWriter fw = new FileWriter("b.txt"); //2 创建输出流  
  
 char[] ch = new char[1024]; //定义字符数组（因为操作的是字符，所以不能再使用字节数组）  
 int len;  
 while ((len = fr.read(ch)) != -1 ){ //3 读：循环将字符写入字符数组  
 fw.write(ch,0,len); //4 写：循环将字符读出  
 }  
   
 fr.close(); //5 关闭输入流  
 fw.close(); //6 关闭输出流  
  
}

4.5 字符流的缓冲流

BufferedReader BufferedWriter

public static void test3() throws IOException {  
   
 BufferedReader br = new BufferedReader(new FileReader("a.txt"));  
 BufferedWriter bw = new BufferedWriter(new FileWriter("b.txt"));  
   
 int ch ;  
 while ((ch = br.read()) != -1){  
 bw.write(ch);  
 }  
 br.close();  
 bw.close();  
   
}

## 4.6 缓冲流的 readLine() newline()

字节流的缓冲流没有此方法。

public static void test4() throws IOException {  
  
 BufferedReader br = new BufferedReader(new FileReader("a.txt")); //输入流  
 BufferedWriter bw = new BufferedWriter(new FileWriter("b.txt")); //输出流  
 String line; //接收行  
 while ((line = br.readLine()) != null){ //循环读取每一行  
 bw.write(line); //循环写入每一行  
 bw.newLine(); //开启新的一行(因为readLine中不读取回车换行，所以要加上)： 都支持  
 //bw.write("\r\n"); //这样写可以换行：只支持windows系统  
 }  
 br.close(); //关闭输入流  
 bw.close(); //关闭输出流  
}

## 4.7 BufferedReader的子类LineNumberReader

特殊方法：getLineNumber()

setLineNumber()

public static void test5() throws IOException {  
 LineNumberReader lnr = new LineNumberReader(new FileReader("a.txt")); //输入流  
  
 String line; //行接收  
 lnr.setLineNumber(100); //设置lineNumber初始值  
 while ((line = lnr.readLine()) != null){ //循环读每一行  
 System.*out*.println(lnr.getLineNumber()+":"+line); //每读一次，lineNumber就加1  
 }  
}

结果：

101:你shdi

102:sfds

103:sfds

104:fsa

105:df

106:sdfaf

# 3 转换流：指定码表读写字符

## 5.1 使用转换流InputStreamReader / OutputStreamWriter

1）FileReader是使用默认码表读取文件，如果需要指定的码表读取，要使用InputStreamReader(字节流，编码表)；

2）FileWriter是使用默认码表写出文件，如果需要使用指定的码表写，要使用OutputStreamWriter(字节流，编码表)；

*/\*\*  
 \*   
 \* 指定编码表读写  
 \*/*public static void test6() throws FileNotFoundException, UnsupportedEncodingException {  
 BufferedReader br = new BufferedReader(

new InputStreamReader(new FileInputStream("a.txt"),"UTF-8"));  
 BufferedWriter bw = new BufferedWriter(

new OutputStreamWriter(new FileOutputStream("b.txt"),"GBK"));  
}

## 5.2 转换流图解

示例：将UTF-8的文件，copy到编码为GBK的文件中去。

GBK

UTF-8

FileInputStream

InputStreamReader

BufferedReader

FileoutputStream

OUtputStreamWriter

BufferedWirter

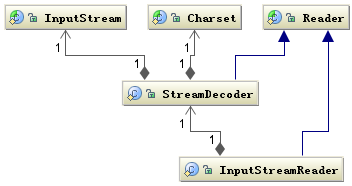
int c = br.read()

OutputStreamWriter(out,”GBK”)按照GBK编码将字符流再转成字节流写出

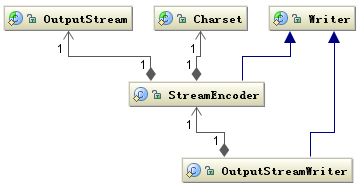
InputStreamReader(in,”UTF-8”)按照utf-8编码将

字节流转换成字符流读取

br.read( ) 将字符提升为int类型，如果像获取char，在读取后再强转char ch = (char) c;



Reader 类是 Java 的 I/O 中读字符的父类，而 InputStream 类是读字节的父类，InputStreamReader 类就是关联字节到字符的桥梁，它负责在 I/O 过程中处理读取字节到字符的转换，而具体字节到字符的解码实现它由 StreamDecoder 去实现，在 StreamDecoder 解码过程中必须由用户指定 Charset 编码格式。值得注意的是如果你没有指定 Charset，将使用本地环境中的默认字符集，例如在中文环境中将使用 GBK 编码。



同样 StreamEncoder 类负责将字符编码成字节，编码格式和默认编码规则与解码是一致的。

# 4 其他流

序列流 ：SequenceInputStream

内存输出流：ByteArrayOutputStream

对象操作流：ObjectOutputStream ObjectInputStream

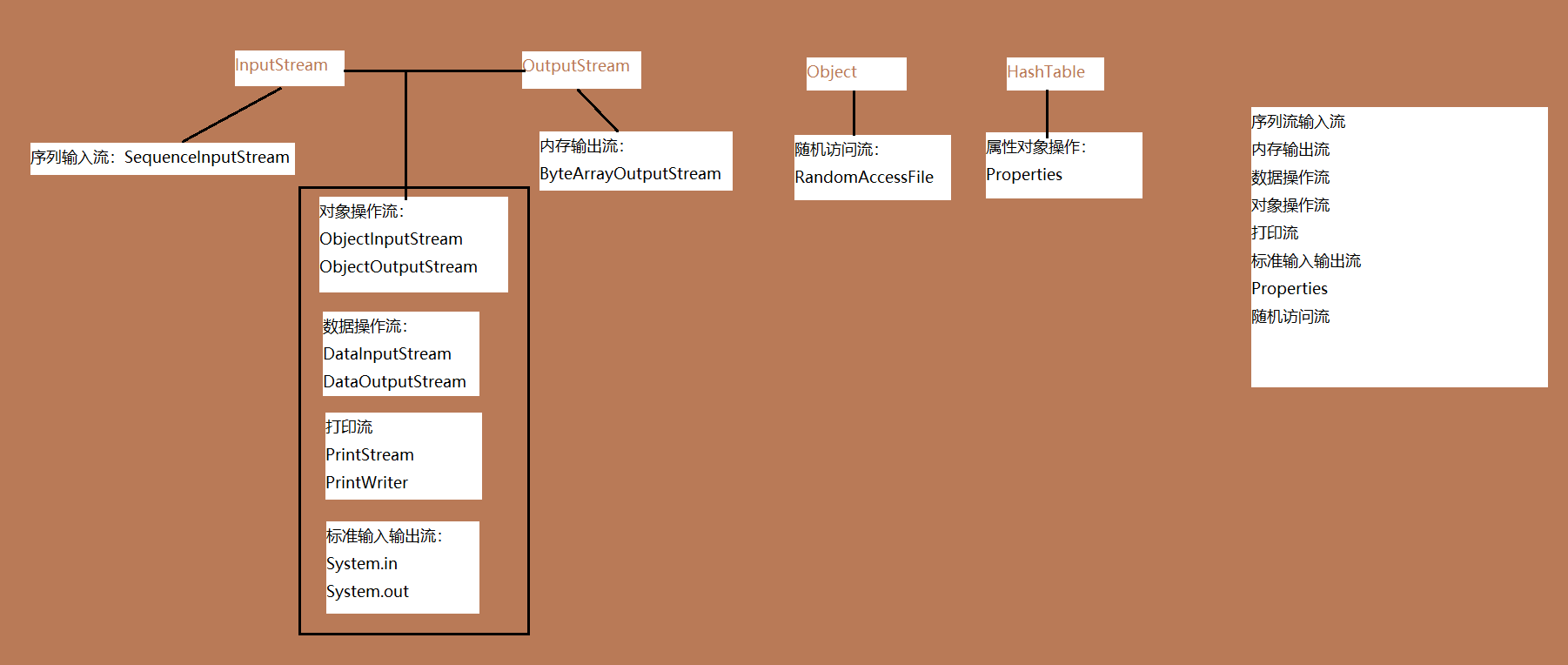
数据流 ：DataInputStream DataOutputStream

打印流 ：PrintStream PrintWriter

标准输入输出流：System.in System.out

随机访问流：RandomAccessFile

属性对象：Properties



记忆法：

序列输入流 --- 内存输出流

对象操作流 --- 数据操作流

打印流 --- 标准输入输出流

随机访问流 --- 属性对象

## 6.1 字节-序列流输入流(队列流)

### 应用：歌曲串烧 视频串烧

### 6.1.1 概述：

序列流可以把多个字节输入流整合成一个，从序列流中读取数据时，将从整合的第一个流开始读，读完一个之后继续读第二个，以此类推。

图解：

FileInputStream2

FileInputStream1

FileOutputStream

FileOutputStream

FileInputStream2

FileInputStream1

第一次写

第二次写

SequenceInputStream

FileOutputStream

只写一次

SequenceInputStream 在关闭时，也会将其传入的参数全部关闭。

### 6.1.2 整合多个输入流

package com.example.demo.io;  
  
  
import java.io.\*;  
import java.util.\*;  
  
public class SequenceIO {  
 public static void main(String[] args) throws IOException {  
 *test4*();  
 }  
  
 */\*\*  
 \* 序列流整个多个输入流 Enumeration 方式二  
 \* 使用List+Iterator+Enumeration匿名对象  
 \** ***@throws*** *IOException  
 \*/* public static void test4() throws IOException {  
 //创建两个输入流  
 FileInputStream fis1 = new FileInputStream("a.txt");  
 FileInputStream fis2 = new FileInputStream("b.txt");  
 FileInputStream fis3 = new FileInputStream("c.txt");  
  
 //创建集合  
 List list = new ArrayList();  
 list.add(fis1);  
 list.add(fis2);  
 list.add(fis3);  
  
 //获得Iterator对象  
 Iterator<FileInputStream> it = list.iterator();  
  
 //创建序列流  
 SequenceInputStream sis = new SequenceInputStream(new Enumeration<InputStream>() {  
 @Override  
 public boolean hasMoreElements() {  
 return it.hasNext();  
 }  
  
 @Override  
 public InputStream nextElement() {  
 return it.next();  
 }  
 });  
 //记住输出流实例创建时，会清空文件  
 FileOutputStream fos = new FileOutputStream("d.txt");  
  
 int b;  
 while ((b = sis.read())!= -1 ){  
 fos.write(b);  
 }  
  
 sis.close();  
 fos.close();  
  
 }  
  
 */\*\*  
 \* 序列流整个多个输入流 Enumeration 方式一  
 \* 使用Vector获得枚举对象Enumeration  
 \** ***@throws*** *IOException  
 \*/* public static void test3() throws IOException {  
 //创建两个输入流  
 FileInputStream fis1 = new FileInputStream("a.txt");  
 FileInputStream fis2 = new FileInputStream("b.txt");  
 FileInputStream fis3 = new FileInputStream("c.txt");  
  
 //创建容器，存储多个流  
 Vector<FileInputStream> vector = new Vector<>();  
 vector.add(fis1);  
 vector.add(fis2);  
 vector.add(fis3);  
  
 //获得枚举对象  
 Enumeration<FileInputStream> en = vector.elements();  
  
 //创建序列流  
 SequenceInputStream sis = new SequenceInputStream(en);  
 //记住输出流实例创建时，会清空文件  
 FileOutputStream fos = new FileOutputStream("d.txt");  
  
 int b;  
 while ((b = sis.read())!= -1 ){  
 fos.write(b);  
 }  
  
 sis.close();  
 fos.close();  
 }  
 */\*\*  
 \* 使用SequenceInputStream整个两个输入流  
 \** ***@throws*** *IOException  
 \*/* public static void test2() throws IOException {  
 //创建两个输入流  
 FileInputStream fis1 = new FileInputStream("a.txt");  
 FileInputStream fis2 = new FileInputStream("b.txt");  
 //记住输出流实例创建时，会清空文件  
 FileOutputStream fos = new FileOutputStream("c.txt");  
  
 //创建序列流  
 SequenceInputStream sis = new SequenceInputStream(fis1,fis2);  
  
 int b;  
 while ((b = sis.read()) != -1 ){  
 fos.write(b);  
 }  
 //关闭序列流，同时会自动关闭输入的参数流  
 sis.close();  
 //关闭输出流  
 fos.close();  
 }  
 */\*\*  
 \* 普通的多输入流操作  
 \** ***@throws*** *IOException  
 \*/* public static void test1() throws IOException {  
 FileInputStream fis1 = new FileInputStream("a.txt");  
 //记住输出流实例创建时，会清空文件  
 FileOutputStream fos = new FileOutputStream("c.txt");  
  
 int b;  
 while ((b = fis1.read()) != -1 ){  
 fos.write(b);  
 }  
 //关闭输入流1  
 fis1.close();  
  
 FileInputStream fis2 = new FileInputStream("b.txt");  
 while ((b = fis2.read()) != -1){  
 fos.write(b);  
 }  
 //关闭输入流2  
 fis2.close();  
 //关闭输出流  
 fos.close();  
  
  
 }  
  
}

### 6.1.3 [Java 获取Enumeration类型的集合](https://www.cnblogs.com/zenghi-home/p/7351601.html)

学习到java的io流中关于序列流SequenceInputStream使用，其中把3个以上的流串联起来操作，使用的参数是生成运行时类型为 InputStream 对象的 Enumeration 型参数，本人对Enumeration集合

了解较少，便记录一下：

方法一：

　　　　通过Vector集合来获得：

1 InputStream is1 = new FileInputStream("d:\\1.txt");

2 InputStream is2 = new FileInputStream("d:\\2.txt");

3 InputStream is3 = new FileInputStream("d:\\3.txt");

4 OutputStream os = new FileOutputStream("d:\\4.txt");

5

6 //使用Vector集合获取Enumeration

7 Vector<InputStream> v = new Vector<InputStream>();

8 v.add(is1);

9 v.add(is2);

10 v.add(is3);

11

13 Enumeration<InputStream> elem = v.elements();

14

15 SequenceInputStream sis = new SequenceInputStream(elem);

16

17 byte[] buf = new byte[1024];

18

19 int len = 0;

20

21 while((len = sis.read(buf)) != -1)

22 {

23 os.write(buf, 0, len);

24 }

如果要想操作里面的元素，可以使用它的两个方法

1　　　　　　//使用Vector集合获取Enumeration

2 Vector<InputStream> v = new Vector<InputStream>();

3 v.add(is1);

4 v.add(is2);

5 v.add(is3);

6

7

8 Enumeration<InputStream> elem = v.elements();

9

10 while(elem.hasMoreElements())

11 {

12 System.out.println(elem.nextElement());

13 }

方法二：

　　利用List集合+Iterator+匿名内部类方法实现

1 InputStream is1 = new FileInputStream("d:\\1.txt");

2 InputStream is2 = new FileInputStream("d:\\2.txt");

3 InputStream is3 = new FileInputStream("d:\\3.txt");

4 OutputStream os = new FileOutputStream("d:\\4.txt");

5

6 LinkedHashSet<InputStream> set = new LinkedHashSet<InputStream>();

7 set.add(is1);

8 set.add(is2);

9 set.add(is3);

10

11 final Iterator<InputStream> it = set.iterator();

12

13 Enumeration<InputStream> en = new Enumeration<InputStream>()

14 {

15

16 @Override

17 public boolean hasMoreElements()

18 {

19 // TODO Auto-generated method stub

20 return it.hasNext();

21 }

22

23 @Override

24 public InputStream nextElement()

25 {

26 // TODO Auto-generated method stub

27 return it.next();

28 }

29 };

30

31 SequenceInputStream sis = new SequenceInputStream(en);

32

33 　　 byte[] buf = new byte[1024];

int len = 0;

while((len = sis.read(buf)) != -1)

{

os.write(buf, 0, len);

}

如果要想操作里面的元素，也可以使用它的两个方法

1 　　　　　Enumeration<InputStream> en = new Enumeration<InputStream>()

2 {

3

4 @Override

5 public boolean hasMoreElements()

6 {

7 // TODO Auto-generated method stub

8 return it.hasNext();

9 }

10

11 @Override

12 public InputStream nextElement()

13 {

14 // TODO Auto-generated method stub

15 return it.next();

16 }

17 };

18

19 while(en.hasMoreElements())

20 {

21 System.out.println(en.nextElement());

22 }

## 6.2 字节-内存输出流

### 应用：QQ聊天记录读取（从硬盘中读到内存中）

### 6.2.1 概述：

== 该输出流可以向内存中写数据，把内存当作一个缓冲区，写出之后可以一次性获取出所有数据。

public class ByteArrayIO {  
 public static void main(String[] args) throws IOException {  
 *test1*();  
 }  
  
 public static void test1() throws IOException {  
 FileInputStream fis = new FileInputStream("a.txt");  
  
 ByteArrayOutputStream baos = new ByteArrayOutputStream();  
  
 int b;  
 while ((b = fis.read()) != -1){  
 baos.write(b);  
 }  
 fis.close();  
  
 //获取内存输出流数据方式一：该方法使用平台默认编码集  
 String data = baos.toString();  
 System.*out*.println(data);  
  
 //获取内存输出流数据方式二：该方法可以指定编码集  
 byte[] bytes = baos.toByteArray();  
 String data2 = new String(bytes,"UTF-8");  
 System.*out*.println(data2);  
 //因为是内存输出，已经完成从硬盘到内存的输出，此处的关闭无效。内存输出流不需要关闭  
 //baos.close();  
  
 }  
}

### 6.2.2 练习：

定义一个文件输入流，调用read(byte[ ] b)方法，将a.txt文件中的内容打印出来，byte数组大小限制为5

1）分析

==因为调用的是read(byte[ ] b) 方法，可见该输入流是字节输入流FileInputStream

==但是字节输入流读取中文，又可能造成乱码问题。主要是因为读取中文时，字节可能读取不完整造成的。比如：中国人。如果是5个字节，就是读取了中国+人的第一个字节。

==可以使用内存输出流，获取全部字节后，一次性输出。这样就不会缺失字节。

*/\*\*  
 \* 内存输出流练习  
 \** ***@throws*** *IOException  
 \*/*public static void test2() throws IOException {  
 //创建输入流  
 FileInputStream fis = new FileInputStream("a.txt");  
  
 //创建内存输出流  
 ByteArrayOutputStream baos = new ByteArrayOutputStream();  
  
 //创建数组  
 byte[] by = new byte[5];  
 //计数器，记录by的有效字节数  
 int len;  
 while ((len = fis.read(by)) != -1){ // 将内容读取字节数组中  
 baos.write(by,0,len); //将字节中数组中的数据，写入内存中

//此处如果直接使用String str = new String(by,0,len)获取str，

//然后使用System.out.println(str)打印出去的，会乱码

}  
  
 //关闭输入流  
 fis.close();  
 //转换为字符串  
 String data = baos.toString();  
 baos.close(); // 无效关闭  
 System.*out*.println(data);  
}

## 6.3随机访问流-实际不属于流

### 应用：多线程下载

### 6.3.1 概述：

== RandomAccessFile概述

== RandomAccessFile类不属于流，是Object类的子类，但它融合了InputStream和

OutputStream的功能。支持对随机访问文件的读取和写入->既能读，也能写。

== 主要方法：

read() 读

write() 写

seek(int position) 指针设置，在指针位置开始写/读。

package com.example.demo.io.otherIO;  
  
import java.io.FileNotFoundException;  
import java.io.IOException;  
import java.io.RandomAccessFile;  
  
public class MyRandomAccessFile {  
 public static void main(String[] args) throws IOException {  
 *test2*();  
 }  
  
  
 */\*\*  
 \* 设置指针的读写  
 \** ***@throws*** *IOException  
 \*/* public static void test2() throws IOException {  
 //创建随机访问对象  
 RandomAccessFile raf = new RandomAccessFile("a.txt","rw");  
  
 //创建指针：从文件字节索引3开始写  
 raf.seek(12);  
 //写:默认从文件字节索引 0 开始写  
 byte[] by = "kkkkk".getBytes();  
 raf.write(by);  
  
 //读：  
 raf.seek(2);  
 byte[] by2 = new byte[10];  
 int len;  
 while ((len = raf.read(by2) ) != -1 ){  
 String str = new String(by2,0,len);  
 System.*out*.println(str);  
 }  
 long p = raf.getFilePointer();  
 System.*out*.println(p);  
 System.*out*.println("==============");  
  
 //读：  
 raf.seek(6);  
 byte[] by3 = new byte[10];  
 int len2;  
 while ((len2 = raf.read(by3) ) != -1 ){  
 String str = new String(by3,0,len2);  
 System.*out*.println(str);  
 }  
  
  
 //关闭流  
 raf.close();  
 }  
  
 */\*\*  
 \* 随机访问流基本使用  
 \** ***@throws*** *IOException  
 \*/* public static void test1() throws IOException {  
 //创建随机访问对象  
 RandomAccessFile raf = new RandomAccessFile("a.txt","rw");  
 //写:默认从文件字节索引 0 开始写  
 raf.write(99);  
 //读：默认从文件字节索引 0 开始读  
 int b = raf.read();  
 System.*out*.println(b);  
 //关闭流  
 raf.close();  
 }  
}

## 6.4 字节-对象操作流

### 6.4.1 概述

== 该流可以将一个对象写出，或者读取一个对象到程序中，也就是执行了序列化·1和反序列化的操作。

== 只能将支持 java.io.Serializable 接口的对象写入流中。每个 serializable 对象的类都被编码，编码内容包括类名和类签名、对象的字段值和数组值，以及从初始对象中引用的其他所有对象的闭包。

### 6.4.2对象输出流 ObjectOutputStream

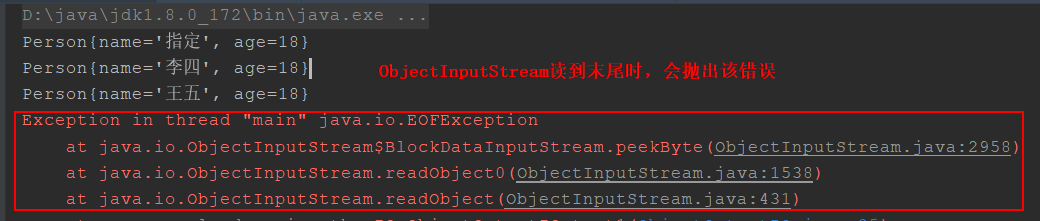
存对象数据：

*/\*\*  
 \* 使用ObjectOutputStream写入对象  
 \** ***@throws*** *IOException  
 \** ***@throws*** *ClassNotFoundException  
 \*/*public static void test1() throws IOException, ClassNotFoundException {  
  
 //创建对象  
 Person p1 = new Person("指定",18);  
 Person p2 = new Person("李四",18);  
 Person p3 = new Person("王五",18);  
  
 //创建对象输出流  
 ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("a.txt"));  
  
 //写对象  
 oos.writeObject(p1);  
 oos.writeObject(p2);  
 oos.writeObject(p3);  
  
 //关闭流  
 oos.close();  
}

### 6.4.3 对象输入流 ObjectInputStream

读数据：

*/\*\*  
 \* 使用ObjectInputStream读取对象  
 \** ***@throws*** *IOException  
 \** ***@throws*** *ClassNotFoundException  
 \*/* public static void test2() throws IOException, ClassNotFoundException {  
 //创建对象输入流  
 ObjectInputStream ois = new ObjectInputStream(new FileInputStream("a.txt"));  
// Person obj = (Person) ois.readObject();  
 Person obj;  
 //读取到文档末尾时报错：java.io.EOFException  
 while ((obj = (Person) ois.readObject()) != null){  
 System.*out*.println(obj);  
 }  
 System.*out*.println(obj);  
  
 //关闭输入流  
 ois.close();  
 }



### 6.4.4 对象流读写优化

将多个对象存入容器中，这样只需要读一次获得容器，在将容器遍历即可。防止EOF异常抛出。

*/\*\*  
 \* 使用ObjectInputStream读取对象优化  
 \** ***@throws*** *IOException  
 \** ***@throws*** *ClassNotFoundException  
 \*/*public static void test4() throws IOException, ClassNotFoundException {  
 //创建对象输入流  
 ObjectInputStream ois = new ObjectInputStream(new FileInputStream("a.txt"));  
  
 //读取容器对象  
 List<Person> list = (List<Person>) ois.readObject();  
  
 //遍历容器  
 for (Person temp: list) {  
 System.*out*.println(temp);  
 }  
 //关闭输入流  
 ois.close();  
}  
*/\*\*  
 \* 使用ObjectOutputStream写入对象优化  
 \** ***@throws*** *IOException  
 \** ***@throws*** *ClassNotFoundException  
 \*/*public static void test3() throws IOException {  
 //创建对象  
 Person p1 = new Person("指定",18);  
 Person p2 = new Person("李四",18);  
 Person p3 = new Person("王五",18);  
  
 //创建容器存储对象  
 List<Person> list = new ArrayList<>();  
 list.add(p1);  
 list.add(p2);  
 list.add(p3);  
  
 //创建对象输出流  
 ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("a.txt"));  
  
 //写入容器  
 oos.writeObject(list);  
  
 //关闭流  
 oos.close();  
}

6.4.5 对象加上id号

当修改对象类，没有进行存储时，直接从文档中读数据，会报错并显示相应的serialVersionUID版本号。更加清晰。不自己加，也会自动生成，就是可读性差。

*/\*\*  
 \* 要实现序列化接口，否则无法操作  
 \*/*class Person implements Serializable {  
 static final Long *serialVersionUID*=1L;

## 6.5 数据输入输出流

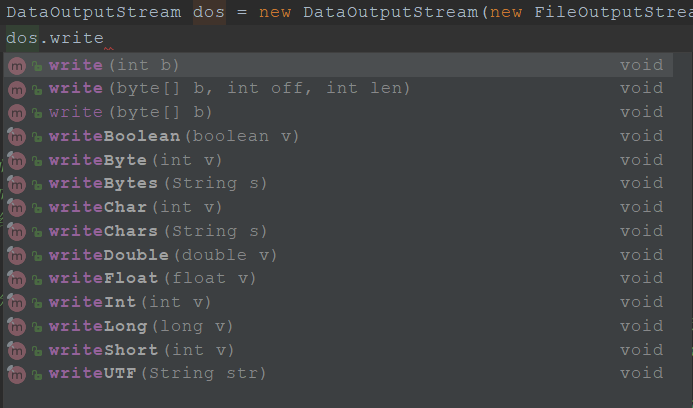
### 6.5.1 问题：

*/\*\*  
 \* 问题测试  
 \* 向文件中输入int类型数据，然后再读取该int数  
 \* 结果：  
 \* 存入的数据是：997 998 999  
 \* 读取的数据是：229 230 231  
 \* 分析：  
 \* write()方法在写入int数据时，会将其前三个字节的24位砍掉，留下后八位作为一个字节存储。  
 \* 如：997 int类型的二进制数据是：00000000 00000000 00000011 11100101  
 \* 调用write（）方法存储时，存的是：11100101  
 \* 读取时读的该二进制的int类型是：00000000 00000000 00000000 11100101  
 \* 二该数转换位十进制int时就是：229  
 \*/*public static void test1() throws IOException {  
 //创建输出流  
 FileOutputStream fos = new FileOutputStream("a.txt");  
  
 //写入数据  
 fos.write(997);  
 fos.write(998);  
 fos.write(999);  
  
 //关闭输出流  
 fos.close();  
  
 //创建输入流  
 FileInputStream fis = new FileInputStream("a.txt");  
 //读取数据  
 int x = fis.read();  
 int y = fis.read();  
 int z = fis.read();  
 //打印数据  
 System.*out*.println(x);  
 System.*out*.println(y);  
 System.*out*.println(z);  
  
 //关闭输入流  
 fis.close();  
  
}

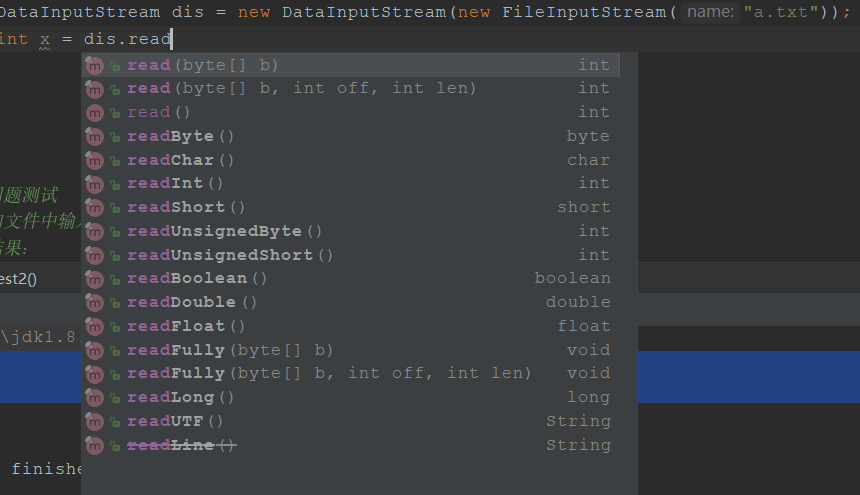
### 6.5.2 专门使用的数据输入输出流：

DataInputStream DataOutputStream

多种专用的写入类型：



多种专用读取类型：



*/\*\*  
 \* 专门用于操作数据的输入输出流  
 \* DataInputStream  
 \* DataOutputStream  
 \* 测试写入：997 998 999  
 \* 读取结果：997 998 999  
 \** ***@throws*** *IOException  
 \*/*public static void test2() throws IOException {  
 //创建数据输入流  
 DataOutputStream dos = new DataOutputStream(new FileOutputStream("a.txt"));  
 //写入int类型数据  
 dos.writeInt(997);  
 dos.writeInt(998);  
 dos.writeInt(999);  
 //关闭输出流  
 dos.close();  
  
 //创建输入流  
 DataInputStream dis = new DataInputStream(new FileInputStream("a.txt"));  
 //读取int类型数据  
 int x = dis.readInt();  
 int y = dis.readInt();  
 int z = dis.readInt();  
 //打印数据  
 System.*out*.println(x);  
 System.*out*.println(y);  
 System.*out*.println(z);  
 //关闭输入流  
 dis.close();  
}

## 6.6 打印流-标准输出流

### 6.6.1 概念

== PrintStream / PrintWriter:分别打印的时字节流和字符流。

只操作数据目的。

== 该流可以很方便的将对象的toString()结果输出，并且自动加上换行；

另外，PrintWriter可以使用自动刷出的模式。（自动刷出只针对println方法）

PrintWriter pw = new PrintWriter(new FileOutputStream(“a.txt”),true);

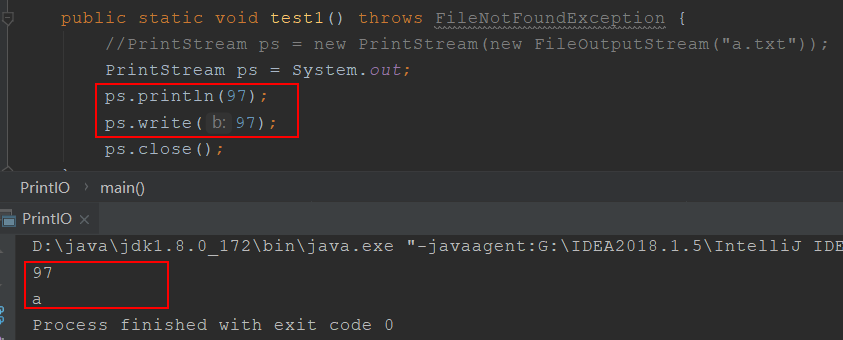
== System.out 就是一个PrintStream,其默认时向控制台输出信息。

即：PrintStream ps = System.out;

== 主要方法：

print() println() 这两个方法，是将参数以字符串形式输出/

write（） 这个方法，是将参数作为字节，并查码表，进行存储。



## 6.7 标准输入输出流

### 6.7.1 什么是标准输入输出流？

== System.in 是InputStream ,标准输入流，默认可以从键盘输入读取字节数据。

== System.out 是PrintStream,标准输出流，默认可以向Console中输出字符和字节数据。

== 两者默认是不需要关闭的，因为没有和外部硬盘文件向关联，而是内存操作。

只有当流和外部文件或网络关联时，需要进行关闭操作。

### 6.7.2 修改标准输入和输出？

== System.setIn(InputStream) : 修改标准输入

== System.setOut(PrintStream)：修改标准输出

public static void test2() throws IOException {  
 System.*setIn*(new FileInputStream("a.txt")); //改变标准输入流  
 System.*setOut*(new PrintStream("b.txt")); //改变标准输出流  
  
 InputStream is = System.*in*; //标准输入流，默认指向键盘输入，改变后指向文件。  
 PrintStream ps = System.*out*; //标准输出流，默认指向控制台，改变后指向文件。  
  
 int b;  
 while ((b = is.read()) != -1){  
 ps.write(b);  
 }  
  
 is.close();  
 ps.close();  
}

## 6.8 两种方式实现键盘录入

== BufferedReader的readLine（）方法

BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

== Scanner

Scanner scanner = new Scanner(System.in)

*/\*\*  
 \* 获取键盘录入的两种方法:方法一  
 \** ***@throws*** *IOException  
 \*/*public static void test3() throws IOException {  
 BufferedReader br = new BufferedReader(new InputStreamReader(System.*in*));  
 String line;  
 while ((line = br.readLine()) != null){  
 System.*out*.println(line);  
 }  
 br.close();  
}  
*/\*\*  
 \* 获取键盘录入的两种方法:方法二  
 \** ***@throws*** *IOException  
 \*/*public static void test4() throws IOException {  
 Scanner scanner = new Scanner(System.*in*);  
 while (scanner.hasNext()){  
 System.*out*.println(scanner.next());  
 }  
}

## 6.9 Properties

### 6.9.1 概述

== 它本身不是流，而是一个双列集合，是HashTable的子类

== Properties类表示了一个持久的属性集

== Properties可保存在流中或从流中加载（保存：store() /加载：load()）

== 属性列表中每一个键及其对应值都是一个字符串

### 6.9.2 特殊功能

== public Object setProperties(String key,String value)

== public String getProperty(String key)

== public Enumeration<String> stringPropertyNames()

### 6.9.3 load() 和store()功能

public static void test() throws IOException {  
 //创建Properties对象  
 Properties prop = new Properties();  
 System.*out*.println(prop);  
  
 //加载流文件对象  
 prop.load(new FileInputStream("config.properties"));  
 System.*out*.println(prop);  
  
 //修改属性  
 prop.setProperty("name","zhangsan");  
 //将修改保存到文件流对象中  
 prop.store(new FileOutputStream("config.properties"),null);  
 System.*out*.println(prop);  
}

# 5 递归

## 递归分析：

递归分析：4\*3\*2\*1

fun(4)=

4\*fun(3)

3\*fun(2)

2\*fun(1)

1

fun(4)=4\*3\*2\*1

代码：

public static int fun(int i){  
 if(i == 1){  
 return 1;  
 }else {  
 return i\**fun*(i-1);  
 }  
}

内存图解分析：

fun(4)=

4\*fun(3)

3\*fun(2) 2\*fun(1)

1

栈内存

main()

fun(4)

4\*fun(3)

3\*fun(2)

2\*fun(1)

fun(1) = 1

main()

fun(4)=24

main()

fun(4) = 24

4\*6

3\*2

2\*1

fun(1) = 1

层层获取结果返回并从栈中弹出

## 6.1 优缺点

好处：不用知道循环的次数

弊端：不能调用过多次，容易造成栈内存溢出。

## 6.2 构造方法能否使用递归？

构造方法不能进行递归。

## 6.3 递归是否必须要有返回值？

不一定。

## 6.4 使用递归遍历文件夹

*/\*\*  
 \* 递归获取文件夹目录  
 \** ***@param*** *file 要操作的文件  
 \** ***@param*** *level 文件级别计数器，打印相应数量的”-“  
 \*/*public static void fun2(File file,int level){  
 level++;  
 //判断是否是文件夹  
 if(file.isDirectory()){  
 //先打印出该文件夹名称  
 for(int i = 0;i<level;i++){  
 System.*out*.print("-");  
 }  
 System.*out*.println("文件夹： "+file.getName());  
  
 //获取并遍历其下的所有文件  
 File[] files = file.listFiles();  
 for (File temp:files) {  
 //再对文件进行递归操作  
 *fun2*(temp,level);  
 }  
 }else {  
 //是文件夹先打印名称，在继续遍历  
 for(int i = 0;i<level;i++){  
 System.*out*.print("-");  
 }  
 //不是文件夹则打印名称  
 System.*out*.println("文件： "+file.getName());  
 }  
}

# 6 装饰设计模式

### 1）好处：

== 耦合性不强，被装饰的类与装饰类的变化无关。

### 2）示例

package com.example.demo.zhuangshizhe;  
  
interface Skills{  
 void skill();  
  
}  
  
*/\*\*  
 \* 装饰者模式  
 \* 好处：被装饰类和装饰类的耦合性不高  
 \* 相对于用继承来实现增强功能，装饰模式更加灵活。 父类变化，子类必须实现该变化，耦合高。  
 \* 被装饰者的变化对于装饰者本身没有影响。 耦合低。  
 \*   
 \*/*

*//被装饰者*public class Person implements Skills{  
  
 public void skill() {  
 System.*out*.println("原有技能：射击");  
 System.*out*.println("原有技能：骑马");  
 }  
  
 public static void main(String[] args) {  
 MakePerson mp = new MakePerson(new Person()); // 使用MakePerson 去装饰（包装） Person  
 mp.skill();  
 }  
}  
*/装饰者*  
class MakePerson implements Skills{  
  
 private Person person ; //1 获取被增强对象引用  
  
 public MakePerson(Person person){ //2 构造方法中注入引用  
 this.person = person;  
 }  
  
 public void skill() { //3 增强方法  
 person.skill();  
 System.*out*.println("新增技能：弹琴");  
 System.*out*.println("新增技能：跳舞");  
 }  
}

# 7 IO流中踩坑：输出流的坑。

## 1 读取的内容为空？

a.txt文件内容：

IO流中的采坑测试

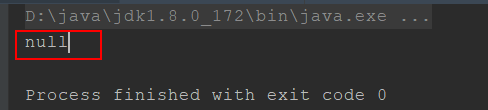
代码：

public static void test3() throws IOException {  
 //输入流  
 BufferedReader br = new BufferedReader(new FileReader("a.txt"));  
 //输出流  
 BufferedWriter bw = new BufferedWriter(new FileWriter("a.txt"));  
 //读取一行  
 String line = br.readLine();  
 //输出  
 System.*out*.println(line);

//关闭流  
br.close();  
bw.close();

}

结果：



什么鬼呢？为什么输出的是null呢？

这个问题主要是输出流导致的：BufferedWriter（其他的输出流也一样）

因为：输出流在创建实例时，会直接将关联文件内容清空，以备后期使用。

上述问题解答：

所以，上述代码中的输出流直接跟在输入流后边，而读取操作在输出流后边。相当于还没有读取文件，文件就被输出流给清空了。

解决：

如果不想输出流清空关联文件，则可以为其添加第二个构造参数，true。表示文件内容追加。

：BufferedWriter bw = new BufferedWriter(new FileWriter("a.txt",true));

## 2 写入的内容为空？

代码：

public static void test4() throws IOException {  
 //输出流  
 BufferedWriter bw = new BufferedWriter(new FileWriter("a.txt",true));  
 //向关联文件中打印一行字符串  
 bw.write("IO输出流采坑测试");  
}

想要写入的内容：

IO输出流采坑测试

结果：

a.txt中为空，没有任何数据

为什么呢？

public abstract class Writer implements Appendable, Closeable, Flushable {  
  
 */\*\*  
 \* Temporary buffer used to hold writes of strings and single characters  
 \*/* private char[] writeBuffer;  
  
 */\*\*  
 \* Size of writeBuffer, must be >= 1  
 \*/* private static final int *WRITE\_BUFFER\_SIZE* = 1024;

FileWriter 上上层父类Writer中有一个缓冲数组。

当向要写入内容是，会先将数据写入该缓冲区中，知道缓冲区满或被flush，才会先向文件中写数据。

上边示例中，数据较少没有使得缓冲区满，所以不会自动向文件中写数据。故此需要手动刷新。使用flush 或close 方法。

所以：

IO流，记得随时关流真的很重要。

//关闭流，不要忘了啊  
bw.close();

# 8 流练习

## 1在控制台输入文件路径名，将该文件copy到项目目录下。

public class ScanTest {  
  
 static String *path*;  
  
 */\*\*  
 \* 获取控制台输入路径，并包装成路径对象  
 \** ***@return*** *路径对象  
 \*/* public static File getFile(){  
 Scanner scanner = new Scanner(System.*in*);  
 System.*out*.println("请输入一个文件地址。。。");  
 File f = null;  
 while (scanner.hasNext()){  
 *path* = scanner.nextLine();  
 System.*out*.println(*path*);  
 f = new File(*path*);  
 return f;  
 }  
 return f;  
 }  
  
 */\*\*  
 \* 对路径对象进行判断，是文件则通过，不是文件则继续  
 \** ***@param*** *file  
 \** ***@throws*** *IOException  
 \*/* public static void check(File file) throws IOException {  
 if(!file.isFile()){  
 System.*out*.println("数据非法：输入的不是一个文件");  
 *getFile*(); //不是文件，则继续调用获取方法  
 }  
   
 try(  
 BufferedInputStream bis = new BufferedInputStream(new FileInputStream(file));  
 BufferedOutputStream bos = new BufferedOutputStream(new FileOutputStream(file.getName()));  
 ){  
 int byt;  
 while ((byt = bis.read()) != -1){  
 bos.write(byt);  
 }  
 }  
 }  
  
 public static void main(String[] args) throws IOException {  
 *check*(*getFile*());  
 }  
  
 //G:\棒.jpg  
}

## 2 将控制台打印的信息copy到文件中

package com.example.demo.io;  
  
import java.io.\*;  
import java.util.Scanner;  
  
public class ScanTest2 {  
  
 boolean flag = true;  
 public void test() throws IOException {  
 Scanner scanner = new Scanner(System.*in*);  
 while (flag){  
 String str = scanner.next();  
 //如果字符串是 停止 则结束  
 if("停止".equals(str)){  
 flag = false;  
 }  
  
 // 写入信息  
 try(  
 BufferedWriter bw = new BufferedWriter(new OutputStreamWriter(new FileOutputStream("a.txt",true)));

//BufferedWriter bw2 = new BufferedWriter(new FileWriter("a.txt"));

){  
 bw.write(str);  
 }  
 }  
 }  
  
 public static void main(String[] args) throws IOException {  
 ScanTest2 st = new ScanTest2();  
 st.test();  
 }  
}

I/O

InputStream

OutputStream

FileInputStream

FileOutputStream

BufferedInputStream

BufferedOutputStream

Reader

Writer

FileReader

FileWriter

BufferedReader

BufferedWriter

字节流

包装流

包装流

字符流

//1 创建输入流

FileInputStream fis = new FileInpuStream(“a.txt”)

//2 创建输出流

FileOutputStream fos = new FileOutputStream(“copy.txt”)

int byt ;

//3 读：不断循环读取每一个字节

while((byt = fis.read()) != -1 ){

//4 写：不断循环写入每一个字节

fos.write(byt);

}

//5 关闭输入流

fis.close();

//6 关闭输出流

fos.close();

读写六大步

InputStreamReader

OutputStreamWriter

转换流

## 3 截取字符串，但不截取半个汉字

汉字在GBK编码中是使用两个字节来表示的，而且第一个字节肯定是负数。

UTF-8 中用三个字节标识，第一个是负数

题目描述：编写一个截取字符串的函数，

1 输入为一个字符串和字节数，

2 输出为按字节截取的字符串，

3 但要保证汉字不被截取半个，

4 如“我ABC”，4，应该截取“我AB”，输入“我ABC汉DEF”，6，应该输出“我ABC”，而不是“我ABC+汉的半个”—》可见输入的是字节的个数

public static void main(String[] args) throws Exception{  
 String str2 = "我ABC ";  
 String str1 = "我ABC汉DEF ";  
 int num = *trimGBK*(str.getBytes("GBK"),6);  
 System.*out*.println(str.substring(0,num) );  
}  
  
public static int trimGBK(byte[] buf,int n){  
 //打印出byte数组观察汉字的字节数，及正负

for(int j = 0;j<buf.length;j++){  
 System.*out*.println(buf[j]);  
 }

//声明一个计数器  
 int num = 0;

//声明一个标识，是否位汉字的第一个字节（也就是前一半）  
 boolean bChineseFirstHalf = false;  
 for(int i=0;i<n;i++){

//判段如果是前一半则不计数，是后一半则计数器加一，表示截取一个字符串位数  
 if(buf[i]<0 && !bChineseFirstHalf){

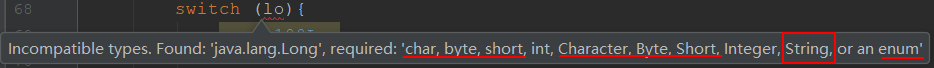
//是前一半，将标识修改位true，使得下一次走计数语句  
 bChineseFirstHalf = true;  
 }else{

//确认是后一半，计数器加一  
 num++;

//计数完成，将标识修改位false，进行下一次的读取判断  
 bChineseFirstHalf = false;  
 }  
 }

//返回要截取的个数  
 return num;  
}

## 4 获取文本上每个字符出现的次数，将结果写在b.txt中



public static void test1() throws IOException {  
 */\*\* 第一大步：读取流 \*/* //创建输入流（先不创建输出流：晚开早关原则）  
 BufferedReader br = new BufferedReader(new FileReader("a.txt"));  
 //创建TreeMap双列集合，来保存字符信息  
 TreeMap<Character,Integer> treeMap = new TreeMap<>();  
 //读取文件  
 int c ;  
 while ((c = br.read()) != -1){ //循环读取每一个字符的int值  
 char ch = (char)c; //将int值强转为char  
 //判断map的key中是否有ch，没有则直接保存，有则值加1  
 if(treeMap.containsKey(ch)){  
 treeMap.put(ch,treeMap.get(ch)+1);  
 }else{  
 treeMap.put(ch,1);  
 }  
 }  
 //关闭输入流  
 br.close();  
  
 */\*\* 第二大步：写出结果流 \*/* //创建输出流  
 BufferedWriter bw = new BufferedWriter(new FileWriter("b.txt"));  
 //写出结果  
 Set<Character> set = treeMap.keySet();  
  
// for(Character key:set){  
// //遍历treeMap并输出结果  
// bw.write(key+": "+treeMap.get(key));  
// bw.newLine();  
// }  
   
 //上面的方法无法输出\r \n，优化如下  
 for(Character key:set) {  
//其中的switch参数，可以是byte-Byte/short-Short/int-Integer char-Character/String enum 但不能是long  
 switch (key) {  
 case '\r':  
 bw.write("\\r" + ": " + treeMap.get(key)); //使用转义字符，使得打印出 \r  
 break; //break是必须的，不然会case穿透  
 case '\n':  
 bw.write("\\n" + ": " + treeMap.get(key));  
 break;  
 case '\t':  
 bw.write("\\t" + ": " + treeMap.get(key));  
 break;  
 default:  
 bw.write(key+": "+treeMap.get(key));  
 break;  
 }  
 bw.newLine(); //每打印一次，进行一次换行  
 }  
 //关闭输出流  
 bw.close();  
 }

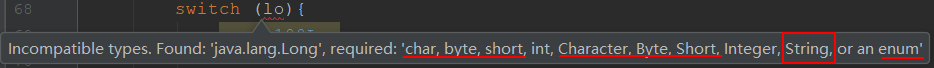
a.txt

你shdi  
sfds  
sfds  
fsa  
df  
sdfaf

结果：

\n: 5  
\r: 5  
a: 2  
d: 5  
f: 6  
h: 1  
i: 1  
s: 7  
你: 1

## 5 switch相关知识点：



switch(condition){

case key :

操作语句；

break;

…

default:

操作语句；

break;

}

switch(condition)可作用参数

常数：byte \ short \ int

字符：char \ String 以及他们的对象包装类Byte\Short\Intger \Character

常量：enum

不支持：double \ float \ long

## 6 定义一个文件输入流，调用read(byte[ ] b)方法，将a.txt文件中的内容打印出来，byte数组大小限制为5

1）分析

==因为调用的是read(byte[ ] b) 方法，可见该输入流是字节输入流FileInputStream

==但是字节输入流读取中文，又可能造成乱码问题。主要是因为读取中文时，字节可能读取不完整造成的。比如：中国人。如果是5个字节，就是读取了中国+人的第一个字节。

==可以使用内存输出流，获取全部字节后，一次性输出。这样就不会缺失字节。

*/\*\*  
 \* 内存输出流练习  
 \** ***@throws*** *IOException  
 \*/*public static void test2() throws IOException {  
 //创建输入流  
 FileInputStream fis = new FileInputStream("a.txt");  
  
 //创建内存输出流  
 ByteArrayOutputStream baos = new ByteArrayOutputStream();  
  
 //创建数组  
 byte[] by = new byte[5];  
 //计数器，记录by的有效字节数  
 int len;  
 while ((len = fis.read(by)) != -1){ // 将内容读取字节数组中  
 baos.write(by,0,len); //将字节中数组中的数据，写入内存中

//此处如果直接使用String str = new String(by,0,len)获取str，

//然后使用System.out.println(str)打印出去的，会乱码

}  
  
 //关闭输入流  
 fis.close();  
 //转换为字符串  
 String data = baos.toString();  
 baos.close(); // 无效关闭  
 System.*out*.println(data);  
}

## 7 流的write（）方法都能写什么？

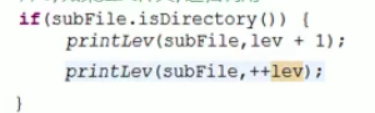
字符流的write()能写： int / char[ ] /String

字节流的write()能写： int / byte[ ] 实际写得都是字节，会自动把int转为byte

## 递归练习-文件夹操作

## 8 遍历文件夹打印其下所有文件以及级别

*/\*\*  
 \* 递归获取文件夹目录  
 \** ***@param*** *file 要操作的文件  
 \** ***@param*** *level 文件级别计数器，打印相应数量的”-“  
 \*/*public static void fun2(File file,int level){  
 level++;  
 //无论是文件还是文件夹，都要打印其名称  
 for(int i = 0;i<level;i++){  
 System.*out*.print("\t");  
 }  
 System.*out*.println(file.getName());  
 //判断如果是文件夹，再对其进行遍历  
 if(file.isDirectory()){  
 //获取并遍历其下的所有文件  
 File[] files = file.listFiles();  
 for (File temp:files) {  
 //再对文件进行递归操作  
 *fun2*(temp,level);  
 }  
 }  
}



此处如果使用fun2(temp,level++) 或 fun2(temp,++level)

将改变原来的level值，导致打印跟预期不同。

所以使用：fun2(temp,level+1);或同示例在方法中操作level

fun2(temp,level++)

异常分析：

fun2(temp,0) 一步

fun2(temp,1) 二步

fun2(temp,2) 三步

模拟目录：

文件夹 一步

文件夹 二步

文件 三步

level++

就是level=level+1

在三步时导致level这个变量被直接修改，而等待返回的

二步中也有变量level，也会被直接修改位2，1->2多一个

打印结果：  
文件夹 一步

文件夹 二步

文件 三步

## 9 统计文件夹大小

package com.example.demo.practice;  
  
import java.io.File;  
import java.nio.file.Files;  
import java.util.Scanner;  
  
*/\*\*  
 \* 统计文件夹大小  
 \*/*public class DirSize {  
  
 //统计大小的变量  
 static long *size*;  
 */\*\*  
 \* 统计文件夹大小:方法一  
 \* 该方法参数可以是文件夹，也可以是文件  
 \** ***@param*** *file  
 \*/* public static void getSize(File file){  
 //判断file是文件夹还是文件  
 if(file.isDirectory()){  
 //获取文件夹下的所有文件  
 File[] files = file.listFiles();  
 //遍历所有文件  
 for (File temp:files) {  
 //再对每一个文件进行 递归判断  
 *getSize*(temp);  
 }  
 }else{  
 System.*out*.println(file.getName());  
 //如果是文件，获取文件大小  
 *size* = *size* + file.length();  
 }  
 }  
  
 */\*\*  
 \* 统计文件夹大小:方法二  
 \* 该方法传入参数必须是文件夹  
 \** ***@param*** *file  
 \*/* public static long getSize2(File file){  
 long size2 = 0;  
 //获取文件夹下所有文件  
 File[] files = file.listFiles();  
 for (File temp:files) {  
 if(temp.isDirectory()){  
 *getSize2*(temp);  
 }else{  
 size2 = size2+temp.length();  
 }  
 }  
 return size2;  
 }  
 */\*\*  
 \* 键盘输入获取路径，返回路径对象  
 \** ***@return*** *\*/* public static File getFile(){  
 //创建输入流对象  
 Scanner scanner = new Scanner(System.*in*);  
 System.*out*.println("请输入文件夹路径。。。");  
 while (true){  
 //获取输入的值  
 String str = scanner.next();  
 File file = new File(str);  
 if(!file.isDirectory()){  
 System.*out*.println("你输入的不是文件夹路径。。。");  
 System.*out*.println("请输入文件夹路径。。。");  
 }  
 else if(!file.exists()){  
 System.*out*.println("你输入的路径不存在。。。");  
 System.*out*.println("请输入文件夹路径。。。");  
 }else {  
 //关闭流  
 scanner.close();  
 return file;  
 }  
 }  
 }  
 public static void main(String[] args) {  
 // File file = new File("G:\\logs");  
 *getSize*(*getFile*());  
 System.*out*.println(*size*);  
 }  
}

## 10 文件夹的删除

package com.example.demo.practice;  
  
import java.io.File;  
import java.util.Scanner;  
  
public class DirDelete {  
  
 public static File getDir(){  
 Scanner scanner = new Scanner(System.*in*);  
 System.*out*.println("输入文件夹路径。。。。");  
 while (true){  
 String line = scanner.nextLine();  
 File file = new File(line);  
 if(!file.isDirectory() || !file.exists()){  
 System.*out*.println("路径不存在或不是文件夹");  
 System.*out*.println("请输入一个文件夹路径。。。");  
 }else {  
 //关闭流  
 scanner.close();  
 return file;  
 }  
 }  
 }  
  
 public static String deleteDir(File dir){  
 //获取文件夹下文件  
 File[] files = dir.listFiles();  
 //遍历文件逐个删除  
 for (File temp:files) {  
 if(temp.isFile()){  
 temp.delete();  
 }else {  
 *deleteDir*(temp);  
 }  
 }  
 //最后再删除空文件夹  
 dir.delete();  
 return "删除成功";  
 }  
  
 public static void main(String[] args) {  
 String str = *deleteDir*(*getDir*());  
 System.*out*.println(str);  
 }  
  
}

## 11 文件夹拷贝

给定两个文件：把一个文件夹，copy到另一个文件里。

方法一：靠的是File构造方法：new File(父级对象，子类名称)。推荐使用。

package com.example.demo.practice;  
  
import java.io.\*;  
import java.util.Scanner;  
  
public class DirCopy2 {  
 */\*\*  
 \* 通过控制台获取要被copy文件路径  
 \* 组装成对象返回  
 \** ***@return*** *\*/* public static File getDir(){  
 Scanner scanner = new Scanner(System.*in*);  
 System.*out*.println("输入文件夹路径。。。。");  
 while (true){  
 String line = scanner.nextLine();  
 File file = new File(line);  
 if(!file.isDirectory() || !file.exists()){  
 System.*out*.println("输入的路径不存在或不是一个文件夹路径。。。");  
 System.*out*.println("请输入文件路径。。。");  
 }else {  
 //关闭流  
 scanner.close();  
 return file;  
 }  
 }  
 }  
  
 public static String copyDir(File src,File dest) throws IOException {  
  
 //这个判断十分重要，不然会无限的进行复制  
 //目标文件夹不能是源文件夹的子文件夹  
 if(src.equals(dest)){  
 System.*out*.println("目标文件夹是源文件夹的子文件夹。。。");  
 return "请重新选择copy";  
 }  
  
 //创建copy生成的文件夹: new File(父路径对象，子类名称)  
 File newDir = new File(dest,src.getName());  
 newDir.mkdirs();  
  
 //获取文件夹下所有文件  
 File[] files = src.listFiles();  
 //遍历所有文件  
 for (File temp:files) {  
 //如果是文件，进行文件copy操作  
 if(temp.isFile()){  
 */\*\*copy操作\*/* //创建输入流  
 BufferedInputStream bis = new BufferedInputStream(new FileInputStream(temp));  
 //创建输出流  
 BufferedOutputStream bos = new BufferedOutputStream(new FileOutputStream(new File(newDir,temp.getName())));  
  
 //声明一个byte数组  
 byte[] bytes = new byte[1024\*8];  
 //声明有效长度  
 int len;  
 //循环读入数组  
 while ((len = bis.read(bytes)) != -1){  
 //循环写出  
 bos.write(bytes,0,len);  
 }  
  
 //关闭流，同时会缓冲区刷出  
 bis.close();  
 bos.close();  
  
 }else {  
 //递归调用处理文件夹  
 *copyDir*(temp,newDir);  
 }  
 }  
 return "文件夹拷贝成功";  
 }  
  
 public static void main(String[] args) throws IOException {  
 File dest = new File("G:\\tempCopy");  
 dest.mkdir();  
 File src = *getDir*();  
 String str = *copyDir*(src, dest);  
 System.*out*.println(str);  
 }  
}

方法二：靠的是构造方法：new File(文件路径)；有点笨这个方法。

package com.example.demo.practice;  
  
import java.io.\*;  
import java.util.Scanner;  
  
public class DirCopy {  
 */\*\*  
 \* 通过控制台获取要被copy文件路径  
 \* 组装成对象返回  
 \** ***@return*** *\*/* public static File getDir(){  
 Scanner scanner = new Scanner(System.*in*);  
 System.*out*.println("输入文件夹路径。。。。");  
 while (true){  
 String line = scanner.nextLine();  
 File file = new File(line);  
 if(!file.isDirectory() || !file.exists()){  
 System.*out*.println("路径不存在或不是文件夹");  
 System.*out*.println("请输入文件路径。。。");  
 }else {  
 //关闭流  
 scanner.close();  
 return file;  
 }  
 }  
 }  
  
 //声明变量，保存被拷贝文件的父路径  
 static String *basePath* ;  
 public static String copyDir(File src,File dest) throws IOException {  
 //获取要被copy文件的父路径: G:\  
 *basePath* = src.getParent();  
 //调用真正的copy操作  
 return *realCopyDir*(src,dest);  
 }  
  
 public static String realCopyDir(File src,File dest) throws IOException {  
  
 //这个判断十分重要，不然会无限的进行复制  
 //目标文件夹不能是源文件夹的子文件夹  
 if(src.equals(dest)){  
 System.*out*.println("目标文件夹是源文件夹的子文件夹。。。");  
 return "请重新选择copy";  
 }  
  
 //获取文件夹下所有文件  
 File[] files = src.listFiles();  
 //遍历所有文件  
 for (File temp:files) {  
  
 */\*\*组装copy生成文件的路径\*/* //获取被copy文件的绝对路径: G:\temp\Dir2\File2.txt  
 String path = temp.getAbsolutePath();  
 //截取路径：获取copy文件的相对路径： temp\Dir2\File2.txt  
 String relatePath = path.substring(*basePath*.length());  
 //组装copy生成的路径： G:\tempCopy \ temp\Dir2\File2.txt  
 String savePath = dest.getPath()+"\\"+relatePath;  
  
 if(temp.isFile()){  
 */\*\*copy操作\*/* //创建输入流  
 BufferedInputStream bis = new BufferedInputStream(new FileInputStream(temp));  
 //创建输出流  
 BufferedOutputStream bos = new BufferedOutputStream(new FileOutputStream(savePath));  
 //声明一个byte数组  
 byte[] bytes = new byte[1024\*8];  
 //声明有效长度  
 int len;  
 //循环读入数组  
 while ((len = bis.read(bytes)) != -1){  
 //循环写出  
 bos.write(bytes,0,len);  
 }  
  
 //关闭流，同时会缓冲区刷出  
 bis.close();  
 bos.close();  
  
 }else {  
 //创建文件夹  
 File file = new File(savePath);  
 file.mkdirs();  
 //继续copy文件  
 *realCopyDir*(temp,dest);  
 }  
 }  
 return "文件夹拷贝成功";  
 }  
  
 public static void main(String[] args) throws IOException {  
 File file = new File("G:\\tempCopy");  
 file.mkdir();  
 String str = *copyDir*(*getDir*(),file);  
 System.*out*.println(str);  
 }  
}

## 12 斐波那契数列

描述：不死神兔

故事从西元1202年说起，一个意大利青年，名叫斐波那契

提出，假设一对刚出生的小兔一个月后就能长成大兔，再过一个月就能生下一对小兔，并且此后每个月都生一对小兔，一年内没有发生死亡。  
 问：一对刚出生的兔子，一年内能繁殖多少对兔子。

一年12个月。

小兔第3个月就能生1对兔。之后每个月生1对。（月初生）

也就上个月新增的兔子，本月都不生。

而上上个月的所又兔子，本月都能生。

本月数=上月数+上上月数

1

1

2

3

5

8

13

21

34

55

89

144

1 2 3

1 2 3 4 5 6 7 8 9 10 11 12

如图(月初生)

6月份的所有兔子包括新增幼崽，到8分至少三个月，都能生。

7月份的所有幼崽，到8月份都是两个月，都不能生，不能为8月献力。

因此，能在8月份产仔的是6月份的所有兔子。

因此，8月份新增数就是6月份的总数。

8 月份 = 7月份+6月份

fun(n) = fun(n-1)+fun(n-2)

// 1 1 2 3 5 8 13 21 34 55 89 144 兔子数  
// 1 2 3 4 5 6 7 8 9 10 11 12 月份  
//本月份 = 上月份+新增的  
//新增的 = 上上月份的总数量  
// 从上月份新增的所有幼崽，一直到本月份，都不满三个月不能生，可以不考虑；

而上上月份的所有兔子都至少满三个月，都产仔，都会加1。

package com.example.demo.practice;  
  
*/\*\*  
 \* 斐波那契数列  
 \*/*public class feibonaqieshu {  
  
 public static int fun(int n){  
 if(n<3){  
 return 1;  
 }else{  
 // 1 1 2 3 5 8 13 21 34 55 89 144  
 // 1 2 3 4 5 6 7 8 9 10 11 12  
 //本月份 = 上月份+新增的  
 //新增的就是，上上月份的总数量  
 //因为上上月份的生所有幼崽，一直到本月份，都不满三个月不能生  
  
 return *fun*(n-1)+*fun*(n-2);  
 }  
 }  
  
 public static void main(String[] args) {  
 System.*out*.println(*fun*(8));  
 }  
}

## 13 约瑟夫环

一个国王叫约瑟夫，他拉出500囚犯围成一圈，从1开始数，数到3的倍数，则杀掉该人，最终剩下一个人。该人最初站在哪个位置。

package com.example.demo.practice;  
  
import java.util.ArrayList;  
  
public class YueSeFuHuan {  
  
 */\*\*  
 \* 方法一：  
 \** ***@param*** *num  
 \*/* public static void test1(int num){  
 //创建容器，并存入数字  
 ArrayList<Integer> list = new ArrayList<>();  
 for(int i = 1;i<=num;i++){  
 list.add(i);  
 }  
  
 int count = 0; //读数计数器，当是3的整数倍时，杀人  
 for(int i = 0;i<list.size();i++){  
 count++;  
  
 //先判断如果只剩一个数，则停止  
 if(list.size() == 1){  
 System.*out*.println(list);  
 break;  
 }  
  
 //如果读数为3的整数倍，则移除当前索引位的数  
 if(count%3 == 0){  
 System.*out*.println("读数： "+count);  
 System.*out*.println("杀死： "+list.get(i));  
 list.remove(i);  
 //因为移除该位后，下一位会自动补到该位置，  
 //所以为了让它也参与读数，强行让 i 减 1，重新读一次该位置  
 i--;  
 }  
  
 if(i == list.size()-1){  
 System.*out*.println(list.size());  
 i = -1; //当i走到最后一位时，要从新从头数，也就是要让i=0；  
 //但是在这个循环节点后，i++会自动加1，导致0索引位无法读取  
 // 所以要先设置i= -1，加1 后刚好为0  
 }  
 System.*out*.println("--------------");  
 }  
 }  
 */\*\*  
 \* 方法二：  
 \** ***@param*** *num  
 \*/* public static void test2(int num){  
 //创建容器，并存入数字  
 ArrayList<Integer> list = new ArrayList<>();  
 for(int i = 1;i<=num;i++){  
 list.add(i);  
 }  
  
 int count = 0; //读数计数器，当是3的整数倍时，杀人  
 for(int i = 0;list.size() != 1;i++){  
 count++; //读一次计数加1  
  
 //如果i等于list.size,重置为0  
 if(i == list.size()){  
 i = 0;  
 }  
  
 //如果读数为3的整数倍，则移除当前索引位的数  
 if(count%3 == 0){  
 list.remove(i);  
 //因为移除该位后，下一位会自动补到该位置，  
 //所以为了让它也参与读数，强行让 i 减 1，重新读一次该位置  
 i--;  
 }  
 }  
 System.*out*.println(list);  
 }  
  
 public static void main(String[] args) {  
 //test1(500);  
 *test2*(8);  
 }  
}

## 14 1000的阶乘所有零和尾部零的个数

StringBuilder 有反转字符串的方法，也可以在本例中使用。

package com.example.demo.practice;  
  
import java.math.BigInteger;  
  
*/\*\*  
 \* 获取1000阶乘的所有0 个数 和 尾部0 的个数  
 \*/*public class BigIntege {  
  
  
 */\*\*  
 \* 获取1000阶乘结果  
 \*/* public static void test1(){  
  
 */\*\*第一步：先获取1000阶乘的结果\*/* //获取 1 的big对象，参数不能是int类型  
 BigInteger big1 = new BigInteger("1");  
  
 for(int i = 1; i<=1000;i++){  
 BigInteger big2 = new BigInteger(i+""); //因为参数不接受int，所以转为字符串  
 big1 = big1.multiply(big2); //相乘  
 }  
  
 System.*out*.println("1000阶乘结果是： "+big1);  
  
 */\*\*第二步：获取其结果中所有零的个数\*/* String str = big1.toString(); //将结果转为字符串  
 int count = 0; //个数计数器  
 for(int i = 0;i<str.length();i++){  
 char c = str.charAt(i); //获取索引i处的字符  
 if(c == '0'){ //如果字符等于0字符，则count加1  
 count++;  
 }  
 }  
 System.*out*.println("结果所有0的个数是： "+count);  
  
 */\*\*第三步：获取其结果末尾所有零的个数\*/* int count2 = 0; //个数计数器  
 for(int i = str.length()-1; i>=0; i--){ //倒着遍历结果  
 char c = str.charAt(i); //获取索引i处的字符  
 if(c == '0'){ //如果字符等于0字符，则count加1  
 count2++;  
 }else {  
 break; //如果字符不等0，则停止  
 }  
 }  
 System.*out*.println("结果末尾0的个数是： "+count2);  
  
  
 }  
  
 public static void main(String[] args) {  
 *test1*();  
 }  
}

## 15 水仙花数

一个数 = 组成这个数每个数位上的数的立方和

package cn.hu.homework;

public class ShuiXianHuaShu {

public static void main(String[] args){

ShuiXianHuaShu.getCount();

}

public static void getCount(){

for (int i = 100; i <= 999; i++) {

//将数字转换为字符串

String str = Integer.toString(i);

//将字符串按照空字符分割

String[] ch = str.split("");

int num = 0;

//循环求和判断

for(int j = 0;j<ch.length;j++){

//使用parseInt（）将字符串装换为int

Integer inte = Integer.parseInt(ch[j]);

//求立方和

num = (int) (num+Math.pow(inte, 3));

}

//判断是否符合水仙花

if(num == i){

System.out.println(i);

}

}

}

}

# 四 》 线程

关键点：

1 线程实现

2 线程设置

3 线程安全（单例实现）

4 线程通讯（对象锁，互斥锁）

5 线程组

6 线程池

7 定时器

== 线程实现

两大类方法：

匿名内部类：

new Thread(){}

new Thread(new Runable(){});

new Thread(new Callable(){});

接口实现类：

继承Thread

实现Runable

实现Callable

相关问题：

== Thread 和 Runable的区别：

继承Thread，可以直接使用Thread的start方法，启动线程，代码简单方便

实现Runable，因为Runable没有start方法，所以需要创建对象，作为Thread的入参，然年使用Thread的start方法，代码复杂。

但是，另一方面，继承如果实现类已经继承了别的类就无法再继承Thread，而Runable接口则不同，实现类可以继承别的类的同事，在实现接口。

== Runable实现的原理：

观察Thread的源码可知，他的构造方法入参可以是Runable类对象，而且有属性target也就是Runnable类型，当调用start方法去调用run方法时，会先判断

target是否为空，如果不为空，则会调用其run方法

== Runnable和Callable的区别

前者run方法无法返回，也不能抛出异常

后者run方法可以有返回值，和抛出异常。

== 线程设置

== 名字设置 setName getName

== 优先级设置 setPrority()

== 线程加入 join

== 线程礼让 yeild

== 线程休眠 sleep

== 线程等待 wait

== 线程守护 daemon

== 线程安全

同步代码块：

同步方法：

== 线程通讯

对象锁：Synchronized（）

互斥锁：

ReentrantLock lock = new ReentranLock();

Condition c1 = lock.newCondition()

Condition c2 = lock.newCondition()

lock.lock(); 获取锁

c1.await(); 指定条件的等待

c2.signal(); 指定条件的唤醒

lock.unlock(); 释放锁

== 线程组

ThreadGroup tg = new ThreadGroup（“group\_name”）

new Thread(tg,"thread\_name");

== 线程池

ExcutorService es = Excutors.newFixedThreadPool(4);

es.submit(new Thread());

== 单例实现：

实现原理：

构造方法私有化

对象引用私有化

获取方法

实现模式：

饿汉模式

懒汉模式/双重锁机制

== 定时器

Timer

TimeTask

timer.schedule(tt);

# 1多线程引入

## 1.1 线程概念和应用场景

### **线程的基本概念、线程的基本状态以及状态之间的关系**

一个程序中可以有多条路径同时执行，一个线程就是程序中的一条执行路径，每个线程上都关联有要执行的代码，即可以有多段程序代码同时运行，每个程序至少都有一个线程，即main方法执行的那个线程。

如果只是一个cpu，它怎么能够同时执行多段程序呢？

这是从宏观上来看的，cpu一会执行a线索，一会执行b线索，切换时间很快，给人的感觉是a,b在同时执行，好比大家在同一个办公室上网，只有一条链接到外部网线，其实，这条网线一会为a传数据，一会为b传数据，由于切换时间很短暂，所以，大家感觉都在同时上网。

就绪 运行 阻塞 等待 挂起 结束

状态：就绪，运行，synchronize阻塞，wait和sleep挂起，结束。wait必须在synchronized内部调用。

调用线程的start方法后线程进入就绪状态，线程调度系统将就绪状态的线程转为运行状态，遇到synchronized语句时，由运行状态转为阻塞，当synchronized获得锁后，由阻塞转为运行，在这种情况可以调用wait方法转为挂起状态，当线程关联的代码执行完后，线程变为结束状态。

1）什么是线程

== 线程就是程序执行的一条路径，一个进程中可以包含多个线程。

== 多线程并发执行，可以提高程序的效率，可以同时完成多个工作。

2）多线程应用场景？

== 红蜘蛛同时共享屏幕给多个电脑

== 迅雷开启多个线程一其下载

== QQ同时和多个人一起视频

== 服务器同时处理多个客户端请求

## 1.2多线程并行和并发的区别

1）并行就是两个任务同时运行，就是甲任务执行的同时，乙任务也在进行。

2）并发是指两个任务都请求允许，而处理器只能接受一个任务，就把这两个任务安排

轮流进行，由于时间间隔较短，使人感觉两个任务都在运行。

3）比如跟两个网友聊天，左右操作一个电脑跟甲聊，同时右手操作另一个电脑跟乙聊，

这就是并行。

4）如果，用一台电脑先给甲发消息，然后再给乙发消息，如此切换，这就叫并发。

## 1.3 Java程序运行原理和JVM的启动是多线程的码？

1）Java程序运行原理？

== java命令会启动java虚拟机，启动JVM，等于启动了一个应用程序，也就是启

动了一个进程，该进程会自动启动一个“主线程”，然后主线程去调用某个类的main

方法。

2） JVM的启动是多线程码？

== JVM启动至少启动了垃圾回收线程和主线程，所以是多线程。

# 2多线程的三实现方式？

[第三种在后边（了解）](#_11_第三种线程实现方法)

## 2.1 \*自定义实现类实现线程的方式：

### 1）继承Thread

== 定义类继承Thread

== 重写run方法

== 把新线程要做的事写再run方法中

== 创建线程对象

== 调用start()方法开启新线程，内部会自动执行run方法

package com.example.demo.thread;  
  
*/\*\*  
 \* 自定义多线程  
 \* 结果：两句话循环打印  
 \*/*public class MyThread {  
  
 public static void main(String[] args) {  
 MyThread2 thread = new MyThread2();// 4 创建自定义线程类对象  
 thread.start(); // 5 调用其start（）方法，如果调用其run方法，则变成普通方法  
  
 for (int i= 0;i<1000;i++){  
 System.*out*.println("主线程for循环");  
 }  
 }  
  
}  
class MyThread2 extends Thread{ // 1 自定义类继承Thread类  
 @Override  
 public void run() { // 2 重写run方法  
 for (int i= 0;i<1000;i++){ // 3 自己的操作语句  
 System.*out*.println("自定义线程。。。");  
 }  
 }  
}

### 2）实现Runnable接口

== 定义类实现Runnable接口

== 实现run方法

== 把新线程要做的事写再run方法中

== 创建自定义的Runnable的子类对象

== 创建Thread对象，传入Runnable

== 调用start()方法开启新线程，内部会自动执行run方法

package com.example.demo.thread;  
  
*/\*\*  
 \* 自定义多线程  
 \* 结果：两句话循环打印  
 \*/*public class MyThread4 {  
  
 public static void main(String[] args) {  
 MyThread5 thread = new MyThread5();// 4 创建自定义线程类对象  
 Thread thread1 = new Thread(thread);// 5 创建Thread对象并传入参数  
 thread1.start(); // 6 调用其start（）方法，如果调用其run方法，则变成普通方法  
  
 for (int i= 0;i<1000;i++){  
 System.*out*.println("主线程for循环");  
 }  
 }  
  
}  
  
class MyThread5 implements Runnable{// 1 实现Runnable接口  
 @Override  
 public void run() { // 2 实现run方法  
 for (int i= 0;i<1000;i++){ // 3 自己的操作语句  
 System.*out*.println("自定义线程。。。");  
 }  
 }  
}

## 2.2 \*匿名内部类实现线程的方式

### 1）继承Thread类

*/\*\*匿名内部类实现：方式一 继承Thread类\*/*new Thread(){  
 @Override  
 public void run() {  
 //something...  
 }  
}.start();

### 2）实现Runnable接口

*/\*\*匿名内部类实现：方式二 实现Runnable方法\*/*//JDK 1.8 新特性  
new Thread(()->{  
   
}).start();  
  
new Thread(new Runnable() {  
 @Override  
 public void run() {  
 //something...  
 }  
}).start();

## 2.3 实现Runnable的原理

1）查看源码

== 看Thread类的构造函数，传递了Runnable接口的引用

== 通过init()方法找到传递的target给成员变量target赋值

== 查看run（）方法，发现run方法中有判断，如果target不为null就会调用

Runnable接口子类对象的run方法。

## 2.4 两种实现方式的区别？

Runnable只有run方法：

@FunctionalInterface  
public interface Runnable {  
 public abstract void run();

Thread的run方法方法

*\*/*@Override  
public void run() {  
 if (target != null) {  
 target.run();  
 }  
}

1） 查看源码的区别：

== 继承Thread：由于子类重写了Thread类的run()，当调用start()时，直接找子

类的run（）方法

== 实现Runnable接口：构造函数中传入了Runnable的引用，成员变量记住了

它，start（）调用run（）方法时，内部判断成员变量Runnable的引用是否

为空，不为空编译时看Runnable的run(),运行执行的是子类的run（）方法。

2）继承Thread:

== 好处是：可以直接使用Thread类中的方法，代码简单

== 弊端是：如果已经有了父类，就不能用这种方法。（只能继承一个父类，不能再

继承Thread了）

3）实现Runnable接口

== 好处是：即使自定义的类有了父类，也没关系，因为还可以去实现接口，而且

接口是可以多实现的

== 弊端是：不能直接使用Thread中的方法，需要先获取到线程对象后传参，才

能的到Thread的方法，代码复杂。

# 3线程操作

## 3.1 获取当前线程对象currentThread

|  |  |
| --- | --- |
| static [Thread](mk:@MSITStore:C:\Users\Bang\Desktop\开发手册\JDK_API_1_6_zh_CN.CHM::/java/lang/Thread.html) | [**currentThread**](mk:@MSITStore:C:\Users\Bang\Desktop\开发手册\JDK_API_1_6_zh_CN.CHM::/java/lang/Thread.html#currentThread())()            返回对当前正在执行的线程对象的引用。 |

package com.example.demo.thread;  
  
public class CurrentThread {  
  
 public static void main(String[] args) {  
 new Thread(){  
 @Override  
 public void run() {  
 //打印当前线程的名字  
 System.*out*.println(getName());  
 }  
 }.start();  
  
  
 new Thread(new Runnable() {  
 @Override  
 public void run() {  
 //由于是实现Runnable接口，所以无法直接使用Thread的方法，去设置或获取名字  
 //所以使用静态方法Thread.currentThread()先获取当前线程，再打印当前线程的名字  
 System.*out*.println(Thread.*currentThread*().getName());  
 }  
 }).start();  
  
 //打印主线程的名字  
 System.*out*.println(Thread.*currentThread*().getName());  
 }  
}

## 3.2 休眠线程sleep(毫秒，纳秒)

一般是用毫秒

|  |  |
| --- | --- |
| static void | [**sleep**](mk:@MSITStore:C:\Users\Bang\Desktop\开发手册\JDK_API_1_6_zh_CN.CHM::/java/lang/Thread.html#sleep(long, int))(long millis, int nanos)  在指定的毫秒数加指定的纳秒数内让当前正在执行的线程休眠（暂停执行），此操作受到系统计时器和调度程序精度和准确性的影响。 |

## 3.3 加入线程 join

== join() 当前线程暂停，等待指定线程执行结束后，当前线程继续

== join(long millis) 可以等待指定的毫秒之后继续

|  |  |
| --- | --- |
| void | [**join**](mk:@MSITStore:C:\Users\Bang\Desktop\开发手册\JDK_API_1_6_zh_CN.CHM::/java/lang/Thread.html#join())()            等待该线程终止。 |
| void | [**join**](mk:@MSITStore:C:\Users\Bang\Desktop\开发手册\JDK_API_1_6_zh_CN.CHM::/java/lang/Thread.html#join(long))(long millis)            等待该线程终止的时间最长为 millis 毫秒。 |
| void | [**join**](mk:@MSITStore:C:\Users\Bang\Desktop\开发手册\JDK_API_1_6_zh_CN.CHM::/java/lang/Thread.html#join(long, int))(long millis, int nanos)  等待该线程终止的时间最长为 millis 毫秒 + nanos 纳秒。 |

package com.example.demo.thread;  
  
public class JoinThread {  
 public static void main(String[] args) {  
   
 Thread t1 = new Thread("线程一"){  
 public void run(){  
 for(int i = 0;i<50 ;i++){  
 System.*out*.println(getName());  
  
 }  
 }  
 };  
  
 Thread t2 = new Thread("线程二"){  
 public void run(){  
 for(int i = 0;i<50 ;i++){  
 System.*out*.println(getName());  
 if(i == 9){  
 try {  
 t1.join(500); // 让t1线程插队执行，  
 // 参数是最多等待500毫秒后，两条线程同时执行  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
 }  
 };  
  
 t1.start();  
 t2.start();  
 }  
}

## 3.4 礼让线程 yield

== yield让出cpu

|  |  |
| --- | --- |
| static void | [**yield**](mk:@MSITStore:C:\Users\Bang\Desktop\开发手册\JDK_API_1_6_zh_CN.CHM::/java/lang/Thread.html#yield())()  暂停当前正在执行的线程对象，让出CPU，并执行其他线程。 |

package com.example.demo.thread;  
  
public class YeildThread {  
  
 public static void main(String[] args) {  
 Thread t1 = new Thread(){  
 public void run(){  
 for(int i =0;i<500;i++){  
 System.*out*.println("aaaaaa");  
 }  
  
 }  
  
 };  
   
 Thread t2 = new Thread(){  
 public void run(){  
 for(int i =0;i<500;i++){  
 System.*out*.println(i+"bb");  
 if(i % 5 == 0){  
 Thread.*yield*(); //让出cpu  
 }  
 }  
 }  
 };  
  
 t1.start();  
 t2.start();  
  
 }  
}

## 3.5 守护线程 setDaemon(Boolean on)

== 设置一个线程为守护线程，该线程不会单独执行，当其他非守护线程都执行结束后，

自动退出。（会有一个时间缓冲）

== 应用：QQ主窗口（非守护线程） 和 聊天窗口（守护线程），关闭主界面，聊天界

面自动关闭

|  |  |
| --- | --- |
| void | [**setDaemon**](mk:@MSITStore:C:\Users\Bang\Desktop\开发手册\JDK_API_1_6_zh_CN.CHM::/java/lang/Thread.html#setDaemon(boolean))(boolean on)            将该线程标记为守护线程或用户线程。 |

package com.example.demo.thread;  
  
public class DaemonThread {  
  
 public static void main(String[] args) {  
 Thread t1 = new Thread(){  
 public void run(){  
 for(int i = 0;i<2 ;i++){  
  
 System.*out*.println("非守护线程。。。");  
 }  
 }  
 };  
  
 Thread t2 = new Thread(){  
 public void run(){  
 for(int i = 0;i<50 ;i++){  
 System.*out*.println("守护线程。。。");  
 }  
 }  
 };  
  
 t2.setDaemon(true);   
  
 t1.start();  
 t2.start();  
 }  
}

## 3.6 线程名字设置和获取

3.6.1设置名字：

== 通过够造参数可以传入String类型的名字

== 通过setName(String name)方法可以设置线程对象的名字

3.6.2获取名字

== getName()方法获取线程的名字

package com.example.demo.thread;  
  
public class ThreadName {  
 public static void main(String[] args) {  
 //this.方法必须写在方法中  
 //构造方法设置名字  
 new Thread("我是线程一"){  
 @Override  
 public void run() {  
 //this.getName() 获取名字  
 System.*out*.println("名字： "+this.getName());  
 }  
 }.start();  
 //构造方法设置名字  
 new Thread("我是线程二"){  
 @Override  
 public void run() {  
 //this.getName() 获取名字  
 System.*out*.println("名字： "+this.getName());  
 }  
 }.start();  
  
 new Thread(){  
 @Override  
 public void run() {

//this.方法必须写在方法中，this.setName()不能写在run()外面  
 //setName()方法设置名字  
 this.setName("我是线程三");  
 //this.getName() 获取名字  
 System.*out*.println("名字： "+this.getName());  
 }  
 }.start();  
 }  
}

## 3.7 线程优先级设置

== setPriority( )设置优先级

|  |  |
| --- | --- |
| void | [**setPriority**](mk:@MSITStore:C:\Users\Bang\Desktop\开发手册\JDK_API_1_6_zh_CN.CHM::/java/lang/Thread.html#setPriority(int))(int newPriority)            更改线程的优先级。 |

package com.example.demo.thread;  
  
public class PriorityThread {  
 public static void main(String[] args) {  
 Thread t1 = new Thread(){  
 public void run(){  
 for(int i =0;i<50;i++){  
 System.*out*.println(getName());  
 }  
  
 }  
  
 };  
 Thread t2 = new Thread(){  
 public void run(){  
 for(int i =0;i<50;i++){  
 System.*out*.println(getName());  
 }  
  
 }  
  
 }; Thread t3 = new Thread(){  
 public void run(){  
 for(int i =0;i<50;i++){  
 System.*out*.println(getName());  
 }  
  
 }  
  
 };  
  
 //设置优先级 1-10  
 t3.setPriority(8);  
 t2.setPriority(5);  
 t1.setPriority(2);  
  
 t1.start();  
 t2.start();  
 t3.start();  
 }  
}

# 4 线程同步-同步代码块 – 同步方法

## 4.1 同步代码块 – 同步方法

### 4.1.1 什么情况需要同步？

== 多线程并发，当有多段代码同时执行时，我们希望某一段代码执行的过程中

CPU不要切换到其他线程工作，这是就需要同步。

== 如果两段代码块是同步，那么同一时间只能执行一段，在一段代码没执行结束

之前，不会执行另一段代码。

### 4.1.2 同步代码块？

== 使用synchronized 关键字加上一个锁对象来定义一段代码，这就叫同步代码块。

== 多个同步代码块如果使用相同的锁对象，那么他们就是同步的。

== 锁对象是任意一个对象（的引用），但是不能是匿名对象。因为匿名对象不是

同一个对象。

//静态方法，可以使用类的class对象

synchronized (SynchronizedThread.class){}

synchronized (SynchronizedThread.class){}

//普通方法，也可以使用类的class对象

synchronized (SynchronizedThread.class){}

synchronized (SynchronizedThread.class){}

//普通方法，可以使用this关键字

synchronized (this){}

synchronized (this){}

//同一个对象的引用

Demo d = new Demo();

synchronized (d){}

synchronized (d){}

//不可以使用匿名对象，因为匿名对象不是同一个对象

synchronized (new Demo()){}

synchronized (new Demo()){}

package com.example.demo.thread.saveThread;  
  
*/\*\*  
 \* 静态方法同步块 :自定义的同一对象 类的class对象  
 \* 普通方法同步块 :自定义的同一对象 类的class对象 类的this对象  
 \* 只要锁相同，就能同步  
 \*/*public class SynchronizedThread {  
 static Demo *d* = new Demo();  
 */\*\*  
 \* 静态方法同步块，使用对象应用 d  
 \*/* public static void test1(){  
 synchronized (*d*){  
 System.*out*.print("a");  
 System.*out*.print("b");  
 System.*out*.print("c");  
 System.*out*.print("d");  
 System.*out*.print("e");  
 System.*out*.print("f");  
 System.*out*.print("g");  
 System.*out*.print("h");  
 System.*out*.println("i");  
 }  
 }  
  
 */\*\*  
 \* 静态方法同步块 使用类的class对象  
 \*/* public static void test2(){  
 synchronized (SynchronizedThread.class){  
 System.*out*.print(1);  
 System.*out*.print(2);  
 System.*out*.print(3);  
 System.*out*.print(4);  
 System.*out*.println(6);  
 }  
 }  
  
 */\*\*  
 \* 普通方法同步块 使用同一对象应用 d  
 \*/* public void test3(){  
 synchronized (*d*){  
 System.*out*.print(1);  
 System.*out*.print(2);  
 System.*out*.print(3);  
 System.*out*.print(4);  
 System.*out*.print(5);  
 System.*out*.print(6);  
 System.*out*.print(7);  
 System.*out*.print(8);  
 System.*out*.print(9);  
 System.*out*.println();  
 }  
  
 }  
public static void main(String[] args) {  
 new Thread(){  
 public void run(){  
 for(int i = 0; i<1000;i++){  
 test1();  
 }  
  
 }  
 }.start();

SynchronizedThread s1 = new SynchronizedThread();  
 new Thread(){  
 public void run(){  
 for(int i = 0; i<1000;i++){  
 s1.test3();  
 }  
 }  
 }.start();  
 }  
}

### 4.1.3 同步方法

== 同步方法：

只需要在方法声明上加synchnorized关键字即可。

== 同步方法的锁是什么？

☛普通方法的是：加载该方法的对象 （this）。不同对象调用，锁不同，不同步

☛静态方法的是：所在类的class对象。该类所有对象的class对象（字节码对象）

相同，锁相同，同步。

package com.example.demo.thread.saveThread;  
  
*/\*\*  
 \* 静态方法同步块 :自定义的同一对象 类的class对象  
 \* 普通方法同步块 :自定义的同一对象 类的class对象 类的this对象  
 \* 只要锁相同，就能同步  
 \*/*public class SynchronizedThread {

*/\*\*  
 \* 如下两种方式：不是自己指定的锁  
 \* test4()的锁是调用该方法的类对象*

*\* test5()的锁是类的class对象*

*\* synchronized：  
\* == 修饰不加static的方法，锁是加在调用该方法的单个对象上，不同的对象竞争不同的锁，不能同步；  
 \* （不同对象(this)锁就不同）  
 \* 即： SynchronizedThread s1 = new SynchronizedThread();  
 \* SynchronizedThread s2 = new SynchronizedThread();  
 \* s1.test4(); 和 s2.test4(); 两个锁加的对象不同，不能同步  
 \*  
\* == 修饰加了static的方法，锁是加在类的class对象上，这个类所有的对象竞争一把锁，同步。  
 \* （同一个类的所有对象的class对象相同，锁就相同）  
 \* SynchronizedThread s1 = new SynchronizedThread();  
 \* SynchronizedThread s2 = new SynchronizedThread();  
 \* s1.test4(); 和 s2.test4(); 两个锁加的class对象行同，同步。  
 \*/* //不用static修饰，锁加在调用该方法的对象上  
 public synchronized void test4(){  
 System.*out*.print(1);  
 System.*out*.print(2);  
 System.*out*.print(3);  
 System.*out*.print(4);  
 System.*out*.print(5);  
 System.*out*.print(6);  
 System.*out*.print(7);  
 System.*out*.print(8);  
 System.*out*.println();  
 }  
 //static修饰，锁加在类的class对象上  
 public static synchronized void test5(){  
 System.*out*.print(1);  
 System.*out*.print(2);  
 System.*out*.print(3);  
 System.*out*.print(4);  
 System.*out*.print(5);  
 System.*out*.print(6);  
 System.*out*.print(7);  
 System.*out*.println();  
 }  
  
  
 public static void main(String[] args) {  
 //创建俩个不同的对象

SynchronizedThread s1 = new SynchronizedThread();  
 SynchronizedThread s2 = new SynchronizedThread();  
  
 new Thread(){  
 public void run(){  
 for(int i = 0; i<1000;i++){  
 s1.test4();  
 }  
  
 }  
 }.start();  
  
 new Thread(){  
 public void run(){  
 for(int i = 0; i<1000;i++){  
 s2.test4();  
 }  
 }  
 }.start();  
 }  
}

### 注意区分同步方法和同步代码块：

Synchronized关键字+锁对象+{代码块}

synchronized(同一对象){

操作语句

}

1）锁由自己指定

2）锁对象相同，就构成同步

普通方法：自定义对象/类对象this/类字节码对象

静态方法：自定义对象/类字节码对象

在方法声明上加Synchronized关键字

public synchronized void test(){ }

public static synchronized void test(){ }

1）锁是默认的

2）锁对象相同，就构成同步

普通方法：默认锁是调用该方法的对象this

不同对象调用，不构成同步。

静态方法：默认锁是该类的字节码对象；

该类所有对象调用，都同步。

同步代码块：

同步方法：

## 4.2 线程安全问题 -火车票案例

== 多线程并发操作同一数据时，就有可能出现线程安全问题

== 使用同步技术可以解决这种问题，把操作数据的代码块进行同步，不要多个线程

一起操作。

如下示例：不加同步锁，会出现负数的票，加了同步锁就安全了。

package com.example.demo.thread.saveThread;  
  
  
public class TicketThread extends Thread {  
  
 //创建汽车票变量，静态，供多个对象公用  
 static Integer *num* = 100;  
 boolean flag = true;  
 //操作购票语句  
 public void run(){  
 //同步代码块：为了保证该类的所有对象同步，所以锁是：类的字节码对象  
 synchronized (TicketThread.class){  
 //无限循环  
 while (flag){  
 //当票数《=0 时，停止售票  
 if(*num* <= 0){

//停止程序  
 flag = false;  
 break;  
 }  
 //使得线程暂停10毫秒，模拟有较多的语句在执行，  
 //这让在没有加同步锁的情况下，会让其他线程同时进入操作  
 //进而可以暴露不安全问题  
 try {  
 Thread.*sleep*(10);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 //打印购买的时第几张票  
 System.*out*.println(getName()+":"+"你够买的是第： "+*num*+" 张票");  
 //操作票数减一  
 *num* = *num* - 1;  
 }  
 }  
 }  
  
 public static void main(String[] args) {  
 //创建多个对象，测试不同对象间的同步  
 TicketThread tt1 = new TicketThread();  
 TicketThread tt2 = new TicketThread();  
 TicketThread tt3 = new TicketThread();  
 TicketThread tt4 = new TicketThread();  
  
 //开启线程  
 tt1.start();  
 tt2.start();  
 tt3.start();  
 tt4.start();

//此时我想，使用同一个对象开启多次？是不是就开启了多个线程呢？

//tt1.start();

//tt1.start();  
//tt1.start();

//tt1.start();

运行后会报错，因为start()方法中有说明，一个线程开启多次是非法的。  
 }  
  
}

## 4.3 死锁

== 多线程同步的时候，如果同步代码块嵌套，使用相同锁，就有可能出现死锁。

== 所以，尽量不要出现同步代码块嵌套。

如下示例：两个人用一双筷子吃饭；

synchronized (*left*){} 只有大括号{ }内的语句执行完，才释放锁  
...获得：筷子右 等待：筷子左 --》 两人都无法获得另一个筷子

...获得：筷子左 等待： 筷子右 --》

package com.example.demo.thread.saveThread;  
  
public class DeadThread {  
  
 //创建两把锁  
 static String *left* = "筷子左";  
 static String *right* = "筷子右";  
  
 public static void main(String[] args) {  
  
 new Thread(){  
 public void run(){  
 while (true){  
 //只有大括号{ }内的语句执行完，才释放锁

synchronized (*left*){  
 System.*out*.println("...获得："+*left*+" 等待： "+*right*);  
 synchronized (*right*){  
 System.*out*.println("...获得："+*right*+" 开吃");  
 }  
 }  
 }  
 }  
 }.start();  
 new Thread(){  
 public void run(){  
 while (true){  
 synchronized (*right*){  
 System.*out*.println("...获得："+*right*+" 等待："+*left*);  
 synchronized (*left*){  
 System.*out*.println("...获得："+*left*+" 开吃");  
 }  
 }  
 }  
 }  
 }.start();  
  
  
 }  
  
}

## 4.4 线程安全的类

类：

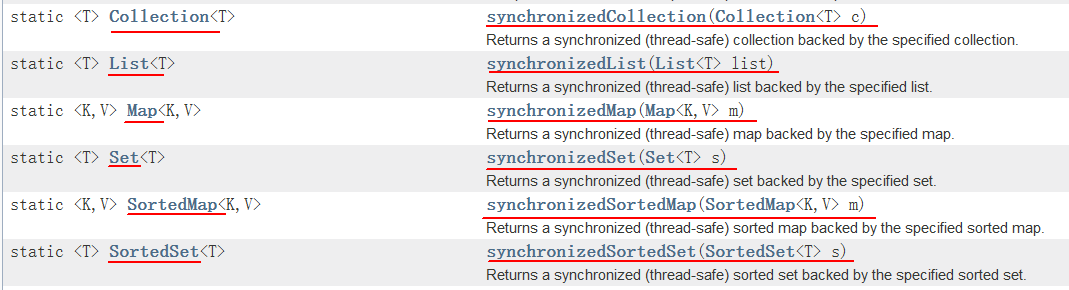
StringBuffer 安全 StringBuilder 不安全

HashTable 安全 HashMap 不安全

Vector 安全 ArrayList 不安全

方法：

Collections.synchronizedXXX(); 可以将不安全的集合变成安全的。



# 静态变量的特殊功能：

== 静态变量：从属于类，只在类第一次被加载时，创建一次，与对象创建与否无关。

而且，被全部该类的对象共有。

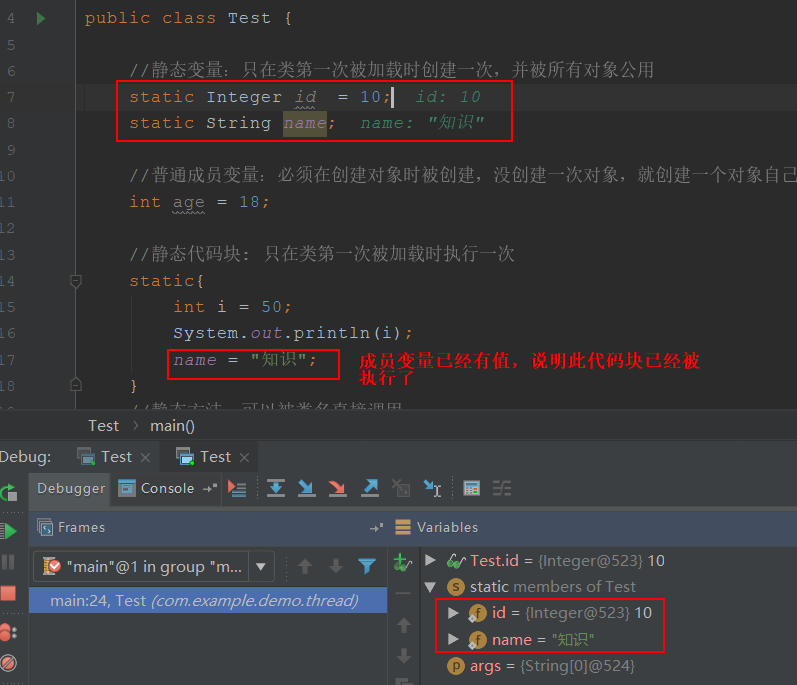
== 静态代码块：只在类第一次被加载时，执行一次，与对象创建与否无关。

== 静态方法：从属于类，可以直接被类名调用。

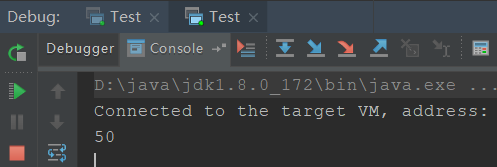
== 静态变量/静态方法都从属于类，可以被类名直接调用，也可以被对象调用。

package com.example.demo.thread;  
  
  
public class Test {  
  
 //静态变量：只在类第一次被加载时创建一次，并被所有对象公用  
 static Integer *id* = 10;  
 static String *name*;  
  
 //普通成员变量：必须在创建对象时被创建，每创建一次对象，就创建一个对象自己的普通变量  
 int age = 18;  
  
 //静态代码块: 只在类第一次被加载时执行一次  
 static{  
 int i = 50;  
 System.*out*.println(i);  
 *name* = "知识";  
 }  
 //静态方法：可以被类名直接调用  
 public static void test(){  
 Integer m = 100;  
 }  
 public static void main(String[] args) {  
 Test t = new Test();  
 Test t2 = new Test();  
 Test t3 = new Test();  
 System.*out*.println("dd");  
 }  
}

Debug调试：断点在对象创建之前，观察。



static代码块操作输出：



# 5 单例设计模式

## 5.1 概述

保证该类在内存中只有一个对象

## 5.2 如何保证类在内存中只有一个对象？

== 控制类的创建，不让别人（其他类）来创建本对象：🡪构造方法私有化

== 在本类中定义一个本类的对象，并私有化。Singleton s；

== 提供公共的访问方法。 public static Singleton getInstance(){return s}

## 5.3 单例的二种写法：

### == 饿汉模式 ： 开发用这种。

上来就吃，上来就new对象

对象不私有化造成的问题：

package com.example.demo.thread.singleton;  
  
public class Singleton {  
  
 //构造方法私有化： 不让别的类调用  
 private Singleton(){}  
 //创建一个public的本类对象  
 public static Singleton *s* = new Singleton();  
}  
class Test{  
 public static void main(String[] args) {  
 */\*\*测试 一\*/* Singleton s1 = Singleton.*s*;  
 Singleton s2 = Singleton.*s*;  
 //让s1 == s2 返回true  
 System.*out*.println(s1 == s2);  
  
 */\*\*测试 二\*/* Singleton s3 = Singleton.*s*;  
 Singleton.*s* = null;  
 Singleton s4 = Singleton.*s*;  
 //让s3 == s4  
 System.*out*.println(s3 == s4);  
  
 //这次返回时false，为什么呢？  
 */\*\*  
 \* 因为s3确实是被赋值为 Singleton.s  
 \* 但是在其之后，我将 Singleton.s 设置为null  
 \* 导致，s4 被赋值为null  
 \* 所以，两者肯定不等了。  
 \* --->由于Singleton没有将这个对象私有化，导致能够通过外部进行修改*

*解决办法：  
 \* ---> 1 ）所以不安全，因此应该将对象私有化，并之对外提供获取方法，不提供设置方法*

*--🡪 2 ） 或者使用final进行修饰，不让外界修改  
 \*/* }  
}

正确的方式：

1） 对象私有化

package com.example.demo.thread.singleton;  
  
public class Singleton2 {  
  
 //构造方法私有化： 不让别的类调用  
 private Singleton2(){}  
 //创建一个private的本类对象  
 private static Singleton2 *s* = new Singleton2();  
  
 //对外部提供获取方法  
 public static Singleton2 getInstance(){  
 return *s*;  
 }  
}  
  
class Test2{  
 public static void main(String[] args) {  
 Singleton2 s1 = Singleton2.*getInstance*();  
 Singleton2 s2 = Singleton2.*getInstance*();  
 System.*out*.println(s1 == s2);  
 }  
}

2） 使用public static final 修饰

外部就不能在使用Singleton.s = null；进行修改了

package com.example.demo.thread.singleton;  
  
public class Singleton {  
  
 //构造方法私有化： 不让别的类调用  
 private Singleton(){}  
 //创建一个public的本类对象  
 public static final Singleton *s* = new Singleton();  
}  
class Test{  
 public static void main(String[] args) {  
 */\*\*测试 一\*/* Singleton s1 = Singleton.*s*;  
 Singleton s2 = Singleton.*s*;  
 //让s1 == s2 返回true  
 System.*out*.println(s1 == s2);  
  
 */\*\*测试 二\*/* Singleton s3 = Singleton.*s*;  
 // Singleton.s = null; 不能再对其进行修改了  
 Singleton s4 = Singleton.*s*;  
 //让s3 == s4  
 System.*out*.println(s3 == s4);  
 }  
}

### == 懒汉模式 ： 面试写这种，多线程的问题？

不打不干活；只声明，不调用方法就不new对象；

package com.example.demo.thread.singleton;  
  
public class Singleton3 {  
  
 //构造方法私有化  
 private Singleton3(){  
 }  
 //声明引用  
 private static Singleton3 *s*;  
 //对外提供获取方法  
 public static Singleton3 getInstance(){  
 if(*s* == null){  
 //判断如果s为空，则为其赋值  
 *s* = new Singleton3();  
 }  
 return *s*;  
 }  
  
 public static void main(String[] args) {  
 Singleton3 s1 = Singleton3.*getInstance*();  
 Singleton3 s2 = Singleton3.*getInstance*();  
 System.*out*.println(s1 == s2);  
 }  
  
}

问题：懒汉式，在多线程访问时有可能创建多个对象

比如：同时三个线程访问，并且同时进入了判断

public static Singleton3 getInstance(){

//线程1 线程2 线程3 都通过了判断并进入 ↓  
 if(s == null){  
 //线程1 线程2 线程3

s = new Singleton3();  
 }  
 return s;  
 }  
如此，当每个线程都会new出一个对象，最终创建了三个对象。

解决：线程同步块

package com.example.demo.thread.singleton;  
  
public class Singleton3 {  
  
 //构造方法私有化  
 private Singleton3(){  
 }  
 //声明引用  
 private static Singleton3 *s*;  
 //对外提供获取方法  
 public static Singleton3 getInstance(){  
 //同步块，同一时间只需一个线程进入判断  
 synchronized (Singleton3.class){  
 if(*s* == null){  
 //判断如果s为空，则为其赋值  
 *s* = new Singleton3();  
 }  
 return *s*;  
 }  
 }  
  
 public static void main(String[] args) {  
 Singleton3 s1 = Singleton3.*getInstance*();  
 Singleton3 s2 = Singleton3.*getInstance*();  
 System.*out*.println(s1 == s2);  
 }  
  
}

问题：这样每个线程都要进入同步块判断，导致效率降低。

解决：双重判断机制。

双重判断机制

package com.example.demo.thread.singleton;  
  
public class Singleton3 {  
  
 //构造方法私有化  
 private Singleton3(){  
 }  
 //声明引用  
 private static Singleton3 *s*;  
 //对外提供获取方法  
 public static Singleton3 getInstance(){  
 //判断为空了在进入同步块，不为空就不需要进同步块判断了  
 if(*s* == null){  
 //同步块，同一时间只需一个线程进入判断  
 synchronized (Singleton3.class){  
 if(*s* == null){  
 //判断如果s为空，则为其赋值  
 *s* = new Singleton3();  
 }  
 }  
 }  
 return *s*;  
 }  
  
 public static void main(String[] args) {  
 Singleton3 s1 = Singleton3.*getInstance*();  
 Singleton3 s2 = Singleton3.*getInstance*();  
 System.*out*.println(s1 == s2);  
 }  
  
}

### 饿汉式和懒汉式区别:

1) 饿汉式：空间换时间；上来就new对象，占用了空间，但是节省了时间。

懒汉式：时间换空间；调用方法判断再new对象，节省了空间，浪费了时间。

2) 在多线程访问时，饿汉式不会创建多个对象，而懒汉式有可能会创建多个对象。

开发时选择：饿汉式。

因为：随着科技发展，时间比空间更宝贵。空间可以通过扩大内存解决，但是时间很难缩短。

## 5.4 单例模式应用场景

== Runtime类

源码：饿汉模式

public class Runtime {

*// 构造方法私有化* private Runtime() {}

*//实例对象私有化* private static Runtime *currentRuntime* = new Runtime();

*// 对外提供的获取方法* public static Runtime getRuntime() {  
 return *currentRuntime*;  
 }

public static void main(String[] args) throws IOException {  
 //Runtime是单例模式，并对外提供了获取对象方法  
 Runtime r = Runtime.*getRuntime*();  
 //exec()方法能都执行DOS命令  
 r.exec("shutdown -s -t 300"); //计划5分钟后关机  
 r.exec("shutdown -a"); //取消关机计划  
}

同一对象来修改，他自己曾经修改的东西。

# 6 Timer（掌握）

## 6.1 概述：

== 定时器（计时器）

== 一种工具，线程用其安排，以后在后台线程中执行的任务。可安排任务执行一次，

或者定期重复执行。

## 6.2 需要掌握的类

### 6.2. 1 == Timer ：定时器

与每个 Timer 对象相对应的是单个后台线程，用于顺序地执行所有计时器任务。计时器任务应该迅速完成。如果完成某个计时器任务的时间太长，那么它会“独占”计时器的任务执行线程。因此，这就可能延迟后续任务的执行，而这些任务就可能“堆在一起”，并且在上述不友好的任务最终完成时才能够被快速连续地执行。

对 Timer 对象最后的引用完成后，*并且* 所有未处理的任务都已执行完成后，计时器的任务执行线程会正常终止（并且成为垃圾回收的对象）。但是这可能要很长时间后才发生。默认情况下，任务执行线程并不作为*守护线程* 来运行，所以它能够阻止应用程序终止。如果调用者想要快速终止计时器的任务执行线程，那么调用者应该调用计时器的 cancel 方法。

如果意外终止了计时器的任务执行线程，例如调用了它的 stop 方法，那么所有以后对该计时器安排任务的尝试都将导致 IllegalStateException，就好像调用了计时器的 cancel 方法一样。

此类是线程安全的：多个线程可以共享单个 Timer 对象而无需进行外部同步。

此类*不* 提供实时保证：它使用 Object.wait(long) 方法来安排任务。

实现注意事项：此类可扩展到大量同时安排的任务（存在数千个都没有问题）。在内部，它使用二进制堆来表示其任务队列，所以安排任务的开销是 O(log n)，其中 n 是同时安排的任务数。

|  |  |
| --- | --- |
| **构造方法摘要** | |
| [**Timer**](mk:@MSITStore:C:\Users\Bang\Desktop\开发手册\JDK_API_1_6_zh_CN.CHM::/java/util/Timer.html#Timer())()            创建一个新计时器。 |  |
| [**Timer**](mk:@MSITStore:C:\Users\Bang\Desktop\开发手册\JDK_API_1_6_zh_CN.CHM::/java/util/Timer.html#Timer(boolean))(boolean isDaemon)            创建一个新计时器，可以指定其相关的线程作为守护程序运行。 |  |
| [**Timer**](mk:@MSITStore:C:\Users\Bang\Desktop\开发手册\JDK_API_1_6_zh_CN.CHM::/java/util/Timer.html#Timer(java.lang.String))([String](mk:@MSITStore:C:\Users\Bang\Desktop\开发手册\JDK_API_1_6_zh_CN.CHM::/java/lang/String.html) name)            创建一个新计时器，其相关的线程具有指定的名称。 |  |
| [**Timer**](mk:@MSITStore:C:\Users\Bang\Desktop\开发手册\JDK_API_1_6_zh_CN.CHM::/java/util/Timer.html#Timer(java.lang.String, boolean))([String](mk:@MSITStore:C:\Users\Bang\Desktop\开发手册\JDK_API_1_6_zh_CN.CHM::/java/lang/String.html) name, boolean isDaemon)            创建一个新计时器，其相关的线程具有指定的名称，并且可以指定作为守护程序运行。 |  |

|  |  |
| --- | --- |
| **方法摘要** | |
| void | [**cancel**](mk:@MSITStore:C:\Users\Bang\Desktop\开发手册\JDK_API_1_6_zh_CN.CHM::/java/util/Timer.html#cancel())()            终止此计时器，丢弃所有当前已安排的任务。 |
| int | [**purge**](mk:@MSITStore:C:\Users\Bang\Desktop\开发手册\JDK_API_1_6_zh_CN.CHM::/java/util/Timer.html#purge())() 清除，净化           从此计时器的任务队列中移除所有已取消的任务。 |
| void | [**schedule**](mk:@MSITStore:C:\Users\Bang\Desktop\开发手册\JDK_API_1_6_zh_CN.CHM::/java/util/Timer.html#schedule(java.util.TimerTask, java.util.Date))([TimerTask](mk:@MSITStore:C:\\Users\\Bang\\Desktop\\开发手册\\JDK_API_1_6_zh_CN.CHM::/java/util/TimerTask.html" \o "java.util 中的类) task, [Date](mk:@MSITStore:C:\Users\Bang\Desktop\开发手册\JDK_API_1_6_zh_CN.CHM::/java/util/Date.html) time) 时刻表           安排在指定的时间执行指定的任务。 |
| void | [**schedule**](mk:@MSITStore:C:\Users\Bang\Desktop\开发手册\JDK_API_1_6_zh_CN.CHM::/java/util/Timer.html#schedule(java.util.TimerTask, java.util.Date, long))([TimerTask](mk:@MSITStore:C:\\Users\\Bang\\Desktop\\开发手册\\JDK_API_1_6_zh_CN.CHM::/java/util/TimerTask.html" \o "java.util 中的类) task, [Date](mk:@MSITStore:C:\Users\Bang\Desktop\开发手册\JDK_API_1_6_zh_CN.CHM::/java/util/Date.html) firstTime, long period)            安排指定的任务在指定的时间开始进行重复的固定延迟执行。 |
| void | [**schedule**](mk:@MSITStore:C:\Users\Bang\Desktop\开发手册\JDK_API_1_6_zh_CN.CHM::/java/util/Timer.html#schedule(java.util.TimerTask, long))([TimerTask](mk:@MSITStore:C:\\Users\\Bang\\Desktop\\开发手册\\JDK_API_1_6_zh_CN.CHM::/java/util/TimerTask.html" \o "java.util 中的类) task, long delay)            安排在指定延迟后执行指定的任务。 |
| void | [**schedule**](mk:@MSITStore:C:\Users\Bang\Desktop\开发手册\JDK_API_1_6_zh_CN.CHM::/java/util/Timer.html#schedule(java.util.TimerTask, long, long))([TimerTask](mk:@MSITStore:C:\\Users\\Bang\\Desktop\\开发手册\\JDK_API_1_6_zh_CN.CHM::/java/util/TimerTask.html" \o "java.util 中的类) task, long delay, long period)            安排指定的任务从指定的延迟后开始进行重复的*固定延迟执行*。 |
| void | [**scheduleAtFixedRate**](mk:@MSITStore:C:\Users\Bang\Desktop\开发手册\JDK_API_1_6_zh_CN.CHM::/java/util/Timer.html#scheduleAtFixedRate(java.util.TimerTask, java.util.Date, long))([TimerTask](mk:@MSITStore:C:\Users\Bang\Desktop\开发手册\JDK_API_1_6_zh_CN.CHM::/java/util/TimerTask.html) task, [Date](mk:@MSITStore:C:\Users\Bang\Desktop\开发手册\JDK_API_1_6_zh_CN.CHM::/java/util/Date.html) firstTime, long period)            安排指定的任务在指定的时间开始进行重复的*固定速率执行*。 |
| void | [**scheduleAtFixedRate**](mk:@MSITStore:C:\Users\Bang\Desktop\开发手册\JDK_API_1_6_zh_CN.CHM::/java/util/Timer.html#scheduleAtFixedRate(java.util.TimerTask, long, long))([TimerTask](mk:@MSITStore:C:\Users\Bang\Desktop\开发手册\JDK_API_1_6_zh_CN.CHM::/java/util/TimerTask.html) task, long delay, long period)            安排指定的任务在指定的延迟后开始进行重复的*固定速率执行*。 |

### 6.2.2 == TimerTask ：定时任务。

由 Timer 安排为一次执行或重复执行的任务。

抽象类，并实现了Runnable接口。（schedule 时刻表）

|  |  |
| --- | --- |
| **构造方法摘要** | |
| protected | [**TimerTask**](mk:@MSITStore:C:\Users\Bang\Desktop\开发手册\JDK_API_1_6_zh_CN.CHM::/java/util/TimerTask.html#TimerTask())()  一个新的计时器任务。 |

|  |  |
| --- | --- |
| **方法摘要** | |
| boolean | [**cancel**](mk:@MSITStore:C:\Users\Bang\Desktop\开发手册\JDK_API_1_6_zh_CN.CHM::/java/util/TimerTask.html#cancel())()            取消此计时器任务。 |
| abstract  void | [**run**](mk:@MSITStore:C:\Users\Bang\Desktop\开发手册\JDK_API_1_6_zh_CN.CHM::/java/util/TimerTask.html#run())()            此计时器任务要执行的操作。 |
| long | [**scheduledExecutionTime**](mk:@MSITStore:C:\Users\Bang\Desktop\开发手册\JDK_API_1_6_zh_CN.CHM::/java/util/TimerTask.html#scheduledExecutionTime())()            返回此任务最近实际 执行的已安排 执行时间。 |

## 6.3 示例

package com.example.demo.thread.timer;  
  
import java.util.Date;  
import java.util.Timer;  
import java.util.TimerTask;  
  
public class timer {  
  
 public static void main(String[] args) throws InterruptedException {  
 //创建定时任务：匿名内部类；也可以自定义一个类来继承TimerTask  
 TimerTask timerTask = new TimerTask() {  
 public void run() {  
 System.*out*.println("定时爆炸。。。");  
 }  
 };  
  
 //创建定时器  
 Timer timer = new Timer();  
 //任务定时  
 */\*\*  
 \* new Date(year,month,date,hrs,min,sec)  
 \* \* @param year the year minus 1900.  
 \* \* @param month the month between 0-11.  
 \* \* @param date the day of the month between 1-31.  
 \* \* @param hrs the hours between 0-23.  
 \* \* @param min the minutes between 0-59.  
 \* \* @param sec the seconds between 0-59.  
 \*/* // timer.schedule(tt,new Date(118,11,5,18,2,0)); //过时了该构造方法  
 //获取程序启动的时间：毫秒  
 long time = new Date().getTime();  
 //任务在10秒后执行，然后每个3秒执行一次  
 timer.schedule(timerTask,new Date(time+10000),3000);  
  
 //每个1秒打印一下，方便看效果  
 while (true){  
 Thread.*sleep*(1000);  
 System.*out*.println("嘀嘀嘀。。。");  
 }  
  
 }  
}

# 7 线程间通信

## 7.1 概述

7.1.1 什么时候需要通信？

== 多个线程并发执行时，在默认情况下CPU是随机切换线程的

== 如果我们希望他们有规律的执行，就可以使用通信，例如每个线程执行10次打印。

7.1.2 怎么通信？

== 如果希望线程等待，就调用wait( )

== 如果希望唤醒等待的线程，就调用notify（）。

== 这俩个方法必须在同步代码块或同步方法中执行，

== 必须用同步锁对象调用方法

## 7.2 两个线程间通信

主要方法：

wait( ) : 当前线程等待（必须被唤醒才能重新进入线程池）

notify( ): 随机唤醒一个等待线程

notifyAll():唤醒所有等待的线程

关键点：设置两线程间的信使-flag

package com.example.demo.thread.thread\_tongxin;  
  
public class TongXin {  
  
 //设置标识，用于判断是否应该让线程等待  
 static boolean *flag* = true;  
  
 */\*\*  
 \* 同步代码块的方法完成通信  
 \** ***@throws*** *InterruptedException  
 \*/* public static void test() throws InterruptedException {  
 synchronized(TongXin.class){  
 //当flag为true时，此线程等待  
 if (*flag*){  
 TongXin.class.wait();  
 }  
  
 for(int i = 0;i<5 ;i++){  
 System.*out*.println("1111");  
 }  
 //最后修改ture,以让当前线程等待  
 *flag* = true;  
 //随机唤醒单个等待的线程

TongXin.class.notify();  
 }  
 }  
 */\*\*  
 \* 同步代码块的方法完成通信  
 \** ***@throws*** *InterruptedException  
 \*/* public static void test2() throws InterruptedException {  
 synchronized(TongXin.class){  
 //当flag为false时，此线程等待  
 if (!*flag*) {  
 TongXin.class.wait();  
 }  
 for(int i = 0; i<5 ;i++){  
 System.*out*.println("2222");  
 }  
 //最后修改false,以让当前线程等待  
 *flag* = false;  
 //随机唤醒单个等待的线程

TongXin.class.notify();

}  
  
 }  
  
 */\*\*  
 \* 同步方法的方式完成通信  
 \** ***@throws*** *InterruptedException  
 \*/* public static synchronized void test3() throws InterruptedException {  
 //当flag为true时，此线程等待  
 if (*flag*){  
 TongXin.class.wait();  
 }  
  
 for(int i = 0;i<5 ;i++){  
 System.*out*.println("11111");  
 }  
 //最后修改ture,以让当前线程等待  
 *flag* = true;  
 //随机唤醒单个等待的线程  
 TongXin.class.notify();  
 }  
  
 */\*\*  
 \* 同步方法的方式完成通信  
 \** ***@throws*** *InterruptedException  
 \*/* public static synchronized void test4() throws InterruptedException {  
 //当flag为false时，此线程等待  
 if (!*flag*) {  
 TongXin.class.wait();  
 }  
 for(int i = 0; i<5 ;i++){  
 System.*out*.println("22222");  
 }  
 //最后修改false,以让当前线程等待  
 *flag* = false;  
 //随机唤醒单个等待的线程  
 TongXin.class.notify();  
 }  
  
 public static void main(String[] args) {  
  
 new Thread(){  
 public void run(){  
 while (true){  
 try {  
 *test3*();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
 }.start();  
 new Thread(){  
 public void run(){  
 while (true){  
 try {  
 *test4*();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
 }.start();  
  
  
  
  
 }  
  
  
}

## 7.3 多个线程间通信

关键点：

1) 信使状态判断：使用while( ) 进行循环判断

// if 判断，在哪里等待，在哪里起来

// 如果是if判断，当符合条件时，会在此等待；  
 // 被唤醒时不会再次进行判断，而是直接再此开始  
 // 所以，当只有两个线程时，没有暴露此问题，但是多个时，就会出现问题

所以使用：while循环进行判断，每次唤醒都会先判断在操作

2）线程唤醒：使用notifyAll()

如果使用notify（），会造成三个线程全部等待的情况。

初级版本：

if(*flag* != 1){ // if 判断，在哪里等待，在哪里起来  
 ThreadTX.class.wait(); // 如果是if判断，当符合条件时，会在此等待；  
} // 被唤醒时不会再次进行判断，而是直接在此开始  
 // 所以，当只有两个线程时，没有暴露此问题，但是多个时，就会出现问题

解决：使用while循环进行判断，每次唤醒都会先判断在操作

package com.example.demo.thread.thread\_tongxin;  
  
*/\*\*  
 \* 按照顺序 1111 2222 3333 进行打印，每次各自打印五次  
 \*/*public class ThreadTX {  
 //设置标识，用于判断是否应该让线程等待  
 static int *flag* = 1;  
  
 */\*\*  
 \* 同步代码块的方法完成通信  
 \** ***@throws*** *InterruptedException  
 \*/* public static void test1() throws InterruptedException {  
 synchronized(ThreadTX.class){  
 //if (flag != 1){ // if 判断，在哪里等待，在哪里起来  
 while (*flag* != 1){ //while循环进行判断，每次唤醒都会先判断在操作  
 ThreadTX.class.wait(); // 如果是if判断，当符合条件时，会在此等待；  
 // 被唤醒时不会再次进行判断，而是直接再此开始  
 // 所以，当只有两个线程时，没有暴露此问题，但是多个时，就会出现问题  
 }  
 for(int i = 0;i<5 ;i++){  
 System.*out*.println(Thread.*currentThread*().getName()+"-11111");  
 }  
 *flag* = 2;  
 ThreadTX.class.notifyAll(); //唤醒所有等待线程，符合条件者运行  
 }  
 }  
 */\*\*  
 \* 同步代码块的方法完成通信  
 \** ***@throws*** *InterruptedException  
 \*/* public static void test2() throws InterruptedException {  
 synchronized(ThreadTX.class){  
 //if (flag != 2) {  
 while (*flag* != 2) {  
 ThreadTX.class.wait();  
 }  
  
 for(int i = 0; i<5 ;i++){  
 System.*out*.println(Thread.*currentThread*().getName()+"-22222");  
 }  
 *flag* = 3;  
 ThreadTX.class.notifyAll();  
 }  
 }  
 */\*\*  
 \* 同步代码块的方法完成通信  
 \** ***@throws*** *InterruptedException  
 \*/* public static void test3() throws InterruptedException {  
 synchronized(ThreadTX.class){  
 //if (flag != 3) {  
 while (*flag* != 3) {  
 ThreadTX.class.wait();  
 }  
  
 for(int i = 0; i<5 ;i++){  
 System.*out*.println(Thread.*currentThread*().getName()+"-33333");  
 }  
 *flag* = 1;  
 ThreadTX.class.notifyAll();  
 }  
 }  
  
 public static void main(String[] args) {  
  
 new Thread(){  
 public void run(){  
 while (true){  
 try {  
 *test1*();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
 }.start();  
  
 new Thread(){  
 public void run(){  
 while (true){  
 try {  
 *test2*();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
 }.start();  
  
 new Thread(){  
 public void run(){  
 while (true){  
 try {  
 *test3*();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
 }.start();  
  
  
  
  
 }  
}

## 7.4 线程通讯注意问题

== 同步代码块中用哪个对象锁，就用那个对象调用wait 和 notify 和notifyall方法

== 为什么wait方法和notify方法定义在Object这个类中？

因为锁对象可以是任意对象，Object是所有类的基类，所以wait方法和notify方

法需要定义在Object这个类中

== sleep方法和wait方法的区别？

☛sleep方法必须传入参数，参数就是时间，时间到了自动醒来。

wait方法可以传入参数，也可以不传入参数，传入参数是在等待指的参数的时间结束后自动唤醒，不传入参数就是立刻等待，必须被别的线程唤醒。

☛sleep方法在同步函数或同步代码块中，不释放锁（睡着了也抱着锁睡）

wait方法在同步函数或同步代码块中，释放锁。

## 7.5 互斥锁（JDK 1.5 新特性）

### 7.5.1 同步

使用ReentrantLock类的lock（） 和unlock（）方法进行同步

### 7.5.2 通信

== 使用ReentrantLock类的newCondition（）方法可以获取Condition对象

== 需要等待的时候使用Condition的await（）方法，唤醒时用signal（）方法

== 不同的线程使用不同的Condition，这样就能区分唤醒的时候找哪个线程了

### 7.5.3 synchronized和Lock（ReentrantLock的父类）的异同 ？

== 主要相同点：Lock能完成synchronized所实现的所有功能

== 主要不同点：

☛Lock有比synchronized更精确的线程语义和更好的性能。

☛synchronized会自动释放锁，而Lock一定要求程序员手工释放

lock.lock() 获得锁 lock.unlock() 释放锁

☛Lock还有更强大的功能，例如，它的tryLock方法可以非阻塞方式去拿锁。

### 7.5.4 Condition状态

== c.await() : 让当前线程进入等待

== c.signal() ： 唤醒指定此状态的线程

== c.notify() : 随机唤醒等待的线程（不能使用这个）

package com.example.demo.thread.reentrantlock;  
  
import java.util.concurrent.locks.Condition;  
import java.util.concurrent.locks.Lock;  
import java.util.concurrent.locks.ReentrantLock;  
  
public class MyLock {  
 //创建锁对象  
 private Lock lock = new ReentrantLock();  
 //创建不同的条件状态  
 Condition c1 = lock.newCondition();  
 Condition c2 = lock.newCondition();  
 Condition c3 = lock.newCondition();  
  
 //线程间的信使  
 private int flag = 1;  
  
 public void test1() throws InterruptedException {  
 //获取锁: 使用ReentrantLock代替了synchronized  
 lock.lock();  
 //flag不等于1 则等待  
 if(flag != 1){ //因为此时是唤醒指定线程，所以不需要循环判断的使用while  
 c1.await(); //在此等待，在此唤醒  
 }  
 //输出一句  
 System.*out*.println(11111);  
 //修改信使状态  
 flag = 2;  
 //唤醒下一个指定状态的线程  
 c2.signal(); //不能使用notify，notify是随机唤醒，不是指定唤醒  
 //最后释放锁  
 lock.unlock();  
 }  
 public void test2() throws InterruptedException {  
 //获取锁: 使用ReentrantLock代替了synchronized  
 lock.lock();  
 //flag不等于1 则等待  
 if(flag != 2){ //因为此时是唤醒指定线程，所以不需要循环判断的使用while  
 c2.await(); //在此等待，在此唤醒  
 }  
 //输出一句  
 System.*out*.println(22222);  
 //修改信使状态  
 flag = 3;  
 //唤醒下一个指定状态的线程  
 c3.signal(); //不能使用notify，notify是随机唤醒，不是指定唤醒  
 //最后释放锁  
 lock.unlock();  
 }  
 public void test3() throws InterruptedException {  
 //获取锁: 使用ReentrantLock代替了synchronized  
 lock.lock();  
 //flag不等于1 则等待  
 if(flag != 3){ //因为此时是唤醒指定线程，所以不需要循环判断的使用while  
 c3.await(); //在此等待，在此唤醒  
 }  
 //输出一句  
 System.*out*.println(33333);  
 //修改信使状态  
 flag = 1;  
 //唤醒下一个指定状态的线程  
 c1.signal(); //不能使用notify，notify是随机唤醒，不是指定唤醒  
 //最后释放锁  
 lock.unlock();  
 }  
  
 public static void main(String[] args) {  
 MyLock myLock = new MyLock();  
  
  
 new Thread(){  
 public void run(){  
 while (true){  
 try {  
 myLock.test1();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
  
 }.start();  
 new Thread(){  
 public void run(){  
 while (true){  
 try {  
 myLock.test2();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
  
 }.start();  
 new Thread(){  
 public void run(){  
 while (true){  
 try {  
 myLock.test3();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
  
 }.start();  
  
 }  
  
  
}

# 生产者-消费者案例

## Food代码

|  |
| --- |
| **package** cn.qianzui.providerAndConsumer;  /\*\*  \* 生产者 - 消费者测试  \* 效果：生产一个食物，就消费一个食物  \* 一共20个循环  \* 需要注意的是:  \* 生产和消费的必须是同一个对象  \* 这样才能体现出线程安全问题  \* 另外，谁拥有属性，就应该将方法写在哪，也即是说生产和消费食物的方法，要写在食物内，有别的方法调用  \*  \* **@author** Bang  \*  \*/  **public** **class** Food {  **private** String name;  **private** String desc;  **private** Boolean flag = **false**;// 为true 的时候能吃不能做，为false时能做不能吃    /\*\*  \* 生产食物  \* **@param** name  \* **@param** desc  \*/  **public** **synchronized** **void** makeFood(String name,String desc) {  // 当为true时，能吃不能做  **if**(flag) {  **try** {  System.***err***.println("cpu切换到此。。。makeFood1");  **this**.wait();  System.***err***.println("cpu切换到此,并被唤醒。。。makeFood2");  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }    **try** {  Thread.*sleep*(500);  } **catch** (InterruptedException e) {  e.printStackTrace();  }    **this**.setName(name);  **this**.setDesc(desc);  System.***out***.println("生产了Food： "+name);    // 修改flag  flag = **true**;  // 随机唤醒线程  **this**.notify();  }  /\*\*  \* 消费事务  \*/  **public** **synchronized** **void** eatFood() {  // 当为true时，能吃不能做  **if**(!flag) {  **try** {  System.***err***.println("cpu切换到此。。。eatFood1");  **this**.wait();  System.***err***.println("cpu切换到此，并被唤醒。。。eatFood2");  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }    **try** {  Thread.*sleep*(500);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  System.***out***.println(**this**.toString());    // 修改flag  flag = **false**;  // 随机唤醒线程  **this**.notify();  }  **public** String getName() {  **return** name;  }  **public** **void** setName(String name) {  **this**.name = name;  }  **public** String getDesc() {  **return** desc;  }  **public** **void** setDesc(String desc) {  **this**.desc = desc;  }  **public** Food() {  **super**();  }  **public** Food(String name, String desc) {  **super**();  **this**.name = name;  **this**.desc = desc;  }  @Override  **public** String toString() {  **return** name + "--->>>" + desc;  }      } |

## 生产者

|  |
| --- |
| **package** cn.qianzui.providerAndConsumer;  **public** **class** Provider **implements** Runnable{  // 传入的食物，和消费的食物是同一个对象  **private** Food food;    **public** Provider(Food food) {  **this**.food = food;  }  @Override  **public** **void** run() {  **for**(**int** i =0;i<20;i++) {  **if**(i%2 == 0) {  food.makeFood("红烧肉","香甜滑嫩");  }**else** {  food.makeFood("西红柿炒鸡蛋","好吃");  }    }  }  } |

## 消费者

|  |
| --- |
| **package** cn.qianzui.providerAndConsumer;  **public** **class** Consumer **implements** Runnable{  **private** Food food;  **public** Consumer(Food food) {  **this**.food = food;  }  @Override  **public** **void** run() {  **for**(**int** i = 0;i<20;i++) {  food.eatFood();  }  }  } |

## 测试

|  |
| --- |
| **package** cn.qianzui.providerAndConsumer;  **public** **class** ProviderAndConsumerTest {  **public** **static** **void** main(String[] args) {  // 创建一个食物对象，但是不进行赋值 : 同时传入生产者和消费者  Food food = **new** Food();  // 创建生产者  Provider p = **new** Provider(food);  // 创建消费者  Consumer c = **new** Consumer(food);    Thread t1 = **new** Thread(p);  Thread t2 = **new** Thread(c);  t1.start();  t2.start();  }  } |

# 8 线程组

线程的部分构造方法回顾

|  |
| --- |
| [**Thread**](mk:@MSITStore:C:\Users\Bang\Desktop\开发手册\JDK_API_1_6_zh_CN.CHM::/java/lang/Thread.html#Thread(java.lang.ThreadGroup, java.lang.Runnable, java.lang.String))([ThreadGroup](mk:@MSITStore:C:\\Users\\Bang\\Desktop\\开发手册\\JDK_API_1_6_zh_CN.CHM::/java/lang/ThreadGroup.html" \o "java.lang 中的类) group, [Runnable](mk:@MSITStore:C:\Users\Bang\Desktop\开发手册\JDK_API_1_6_zh_CN.CHM::/java/lang/Runnable.html) target, [String](mk:@MSITStore:C:\Users\Bang\Desktop\开发手册\JDK_API_1_6_zh_CN.CHM::/java/lang/String.html) name)            分配新的 Thread 对象，以便将 target 作为其运行对象，将指定的 name 作为其名称，并作为 group 所引用的线程组的一员。 |
| [**Thread**](mk:@MSITStore:C:\Users\Bang\Desktop\开发手册\JDK_API_1_6_zh_CN.CHM::/java/lang/Thread.html#Thread(java.lang.ThreadGroup, java.lang.Runnable, java.lang.String, long))([ThreadGroup](mk:@MSITStore:C:\\Users\\Bang\\Desktop\\开发手册\\JDK_API_1_6_zh_CN.CHM::/java/lang/ThreadGroup.html" \o "java.lang 中的类) group, [Runnable](mk:@MSITStore:C:\Users\Bang\Desktop\开发手册\JDK_API_1_6_zh_CN.CHM::/java/lang/Runnable.html) target, [String](mk:@MSITStore:C:\Users\Bang\Desktop\开发手册\JDK_API_1_6_zh_CN.CHM::/java/lang/String.html) name, long stackSize)            分配新的 Thread 对象，以便将 target 作为其运行对象，将指定的 name 作为其名称，作为 group 所引用的线程组的一员，并具有指定的*堆栈大小*。 |

## 8.1 线程组概述

== Java中使用ThreadGroup来表示线程组，它可以对一批线程进行分类管理，java允

许程序直接对线程组进行控制。

== 默认情况下，所有的线程都属于主线程组

☛public final ThreadGroup getThreapGroup() //通过线程对象获取他所属的组

☛public final String getName() //通过线程组对象获取他的组名

== 也可以给线程设置分组

☛ThreadGroup(String name) 创建线程组对象，并命名。

☛创建线程对象

☛Thread（ThreadGroup group，Runnable target，String name ）

☛设置整组的优先级 或 守护线程

package com.example.demo.thread.threadgroup;  
  
public class MyThreadGroup {  
 public static void main(String[] args) {  
  
 //创建线程组  
 ThreadGroup tg = new ThreadGroup("我是一个新的线程组");  
 //创建线程：并写入参数  
 MyThread mt1 = new MyThread(tg,"张三");  
 MyThread mt2 = new MyThread(tg,"李四");  
 MyThread mt3 = new MyThread(tg,"王五");  
 //输出线程组的名字  
 System.*out*.println(mt1.getThreadGroup().getName());  
 System.*out*.println(mt2.getThreadGroup().getName());  
 System.*out*.println(mt3.getThreadGroup().getName());  
  
 //将整个组设置为守护线程  
 tg.setDaemon(true);  
 }  
  
}  
  
*/\*\*  
 \* 自定义一个类继承父类线程  
 \*/*class MyThread extends Thread{  
 //提供无参构造  
 public MyThread() {  
 }  
 //重写父类的构造方法，不然无法添加参数  
 public MyThread(ThreadGroup group, String name) {  
 super(group, name);  
 }  
  
 public void run(){  
 while (true){  
 System.*out*.println(getName()+"...");  
 }  
 }  
}

# 9 线程五种状态

新建 就绪 运行 阻塞 死亡

== 新建：创建线程对象

== 就绪：线程对象已经启动了，但是还没有获取到CPU的执行权

== 运行：获取到了CPU的执行权

== 阻塞：没有CPU的执行权，回到就绪状态

== 死亡：代码运行完毕，线程消亡（其他：自定义标志flag或interrupt）

1 新建

创建线程对象

2 就绪

线程对象已经启动

有执行的资格

但没有执行权

4 阻塞

没有执行资格

没有执行权

3 运行

有执行资格

有执行权

5 死亡

代码运行完毕，

线程对象变成垃圾

start()

唤醒

notify()

signal()

被其他线程抢走CPU执行权

抢到CPU执行权

wait()等待

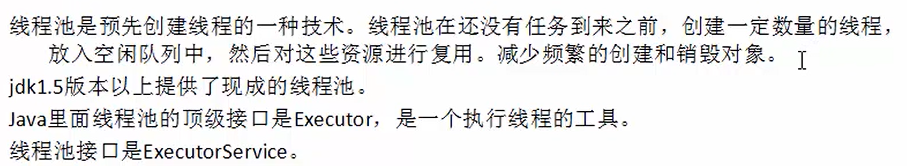
stop()停止

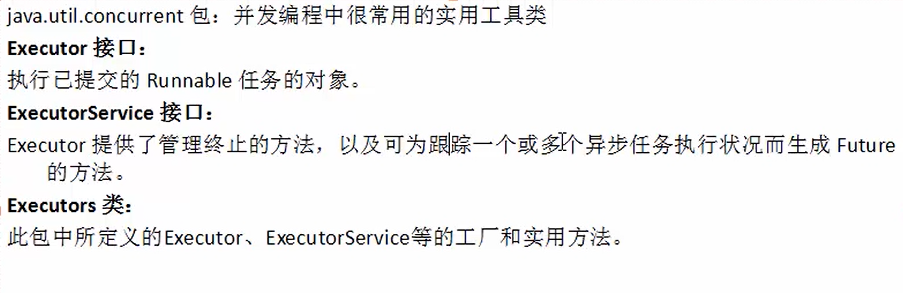
已过时

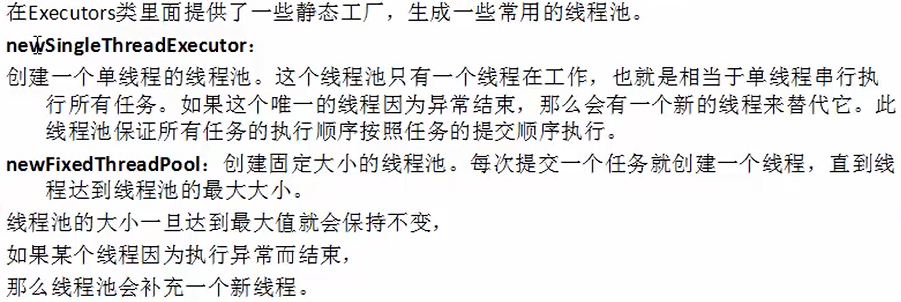
run()结束

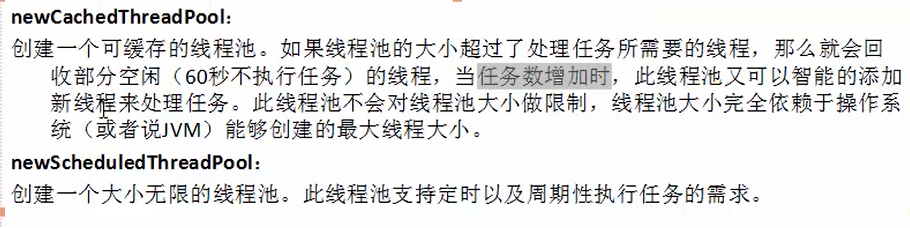
sleep()休眠

# 10 线程池









10.1 线程池概述

== 程序启动一个新线程成本是比较高的，因为它涉及到要与操作系统交互，而使用线程池可以很好的提高性能，尤其是当程序中要创建大量生存期很短的线程时，更应该考虑使用线程池。

private native void start(); 这是Thread的start方法底层调用的方法，native关键字修饰，可见是和操作系统语言进行了交互。

== 线程池里的每一个线程代码结束后，不会死亡，而是再次回到线程池中成为空闲状态，等待下一个对象来使用。

== JDK1.5之前，我们必须手动实现自己的线程池，从JDK1.5开始，Java内置支持线程池。

10.2 内置线程池的使用

== JDK1.5新增了一个Executors工厂类来产生线程池

== 主要方法：

☛public static ExecutorService newFixedThreadPool(int nThreads)

☛public static ExecutorService newSingleThreadExecutor()

这些方法的返回值是ExecutorService对象，该对象表示一个线程池，可以执行Runnable对象或者Callable对象代表的线程。它的主要方法如下：

☛Future<?> submit(Runnable task)

☛<T> Future<T> submit(Callable<T> task)

== 使用步骤

**☛创建线程池对象**

**☛创建Runnable实例**

**☛提交Runnable实例 executorService.submit(xxx)**

**☛关闭线程池 executorService.shutdown()**

shutdowm() 执行以前提交的任务后并顺序关闭线程池，不再接受新任务

package com.example.demo.thread.threadpool;  
  
import java.util.concurrent.ExecutorService;  
import java.util.concurrent.Executors;  
  
public class MyThreadPool {  
  
 public static void main(String[] args) {  
 //获取执行服务器  
 ExecutorService es = Executors.*newFixedThreadPool*(4);  
  
 //提交线程并执行 : submit（） 方法，提交并执行  
 es.submit(new MyRunnable());  
 es.submit(new MyRunnable());  
 es.submit(new MyRunnable());  
  
 //关闭线程池 ：shutdowm() 执行以前提交的任务后关闭线程池，不再接受新任务  
 //启动一次顺序关闭，执行以前提交的任务，但不接受新任务。  
 //如果已经关闭，则调用没有其他作用  
 es.shutdown();  
 //es.submit(new MyRunnable()); 此处线程不被执行，且抛出异常  
  
 }  
}  
  
  
*/\*\*  
 \* 创建Runnable接口的子类  
 \*/*class MyRunnable implements Runnable{  
 @Override  
 public void run() {  
 for (int i = 0;i<=100;i++) {  
 System.*out*.println(Thread.*currentThread*().getName()+"..."+i);  
 }  
 }  
}

# 11 第三种线程实现方法

## 11.1 实现Callable接口

public interface **Callable<V>**

返回结果并且可能抛出异常的任务。实现者定义了一个不带任何参数的叫做 call 的方法。

Callable 接口类似于 [Runnable](mk:@MSITStore:C:\Users\Bang\Desktop\开发手册\JDK_API_1_6_zh_CN.CHM::/java/lang/Runnable.html)，两者都是为那些其实例可能被另一个线程执行的类设计的。但是 Runnable 不会返回结果，并且无法抛出经过检查的异常。

|  |  |
| --- | --- |
| **方法摘要** | |
| [V](mk:@MSITStore:C:\Users\Bang\Desktop\开发手册\JDK_API_1_6_zh_CN.CHM::/java/util/concurrent/Callable.html) | [**call**](mk:@MSITStore:C:\Users\Bang\Desktop\开发手册\JDK_API_1_6_zh_CN.CHM::/java/util/concurrent/Callable.html#call())()            计算结果，如果无法计算结果，则抛出一个异常。 |

## 11.2 Callable和Runnable区别

相同点：两者都是为那些，其实例可能被另一个线程执行的类设计的。

不同点：

Runnable 的run方法不会返回结果，并且无法抛出经过检查的异常。

Callable的call方法，返回结果并且可能抛出异常的任务。

package com.example.demo.thread.getThread;  
  
import java.util.concurrent.\*;  
  
public class MyCallable implements Callable<Integer> {  
 //定义数  
 private Integer num;  
 //带参构造  
 public MyCallable(Integer num) {  
 this.num = num;  
 }  
  
 int total = 0;  
 //重写call方法  
 @Override  
 public Integer call() throws Exception {  
 for(int i = 1; i<=num;i++){  
 total = total + i;  
 }  
 return total;  
 }  
}  
  
class Test{  
 public static void main(String[] args) throws ExecutionException, InterruptedException {  
 //创建线程池  
 ExecutorService es = Executors.*newFixedThreadPool*(2);  
 //创建并提交Callable实现类  
 Future<Integer> f1 = es.submit(new MyCallable(100));  
 Future<Integer> f2 = es.submit(new MyCallable(50));  
  
 //打印结果  
 System.*out*.println(f1.get());  
 System.*out*.println(f2.get());  
  
 //关闭线程池  
 es.shutdown();  
  
  
 }  
}

# 12 设计模式：

设计模式|菜鸟教程：

http://www.runoob.com/design-pattern/abstract-factory-pattern.html

## 这里的所说的模式和Spring中获取bean的三种方式不一样。

Spring中获取bean的三种方式：

### == 使用类的无参构造创建

一个PIJO：

public class Car{

private String name;

private String price;

private Car(){ }

}

配置文件配置：

<bean name=”car” class=”cn.htb.Car”><bean/>

测试获取实例对象：

ApplictionContext applictionContext =

new ClassPathXmlApplictionContext(“applicationContext.xml”)

Car car = (Car) applictionContext.getBean(“car”);

### == 使用静态工厂创建

创建类POJO：

public class Car{ }

创建静态工厂类：

public class MyBeanFactory{

//静态方法

public static Car getBean(){

return new Car();

}

}

配置文件配置：

<bean name=”car” class=”cn.htb.MyBeanFactory” factory-method=”getBean”><bean/>

测试获取实例：

ApplictionContext applictionContext =

new ClassPathXmlApplictionContext(“applicationContext.xml”)

Car car = (Car) applictionContext.getBean(“car”);

### == 使用实例化工厂创建

创建类POJO：

public class Car{ }

创建实例化工厂类：

public class MyBeanFactory{

//非静态方法

public Car getBean(){

return new Car();

}

}

配置文件配置：

<bean name=”myBeanFactory” class=”cn.htb.MyBeanFactory” ><bean/>

<bean name=”car” factory-bean=” myBeanFactory” factory-method=”getBean”><bean/>

测试获取实例：

ApplictionContext applictionContext =

new ClassPathXmlApplictionContext(“applicationContext.xml”)

Car car = (Car) applictionContext.getBean(“car”);

静态工厂类与非静态工厂类的区别:，

前者不需要创建对象，直接可以调用静态方法创建bean；

后者则要先创建对象，然后再通过对象调用其方法创建bean

## 12.1 简单工厂模式 - 生产一类实例

1） 简单工厂模式

== 又叫静态工厂方法模式，它定义一个具体的工厂类负责创建一些类的实例。

2）优点

== 客户端不需要在负责对象的创建，从而明确了各个类的职责。同时实现了调用者，和创建被调用者的动作分离，降低了耦合性。

3）缺点：

== 这个静态工厂类负责所有对象的创建，如果有新的对象增加，或者某些对象的创建方式不同，就需要不断的修改工厂类，不利于后期的维护。

package com.example.demo.设计模式.简单工厂模式;  
  
*/\*\*  
 \* 创建Cat类  
 \*/*public class Cat {  
}

package com.example.demo.设计模式.简单工厂模式;  
  
*/\*\*  
 \* 创建Dog类  
 \*/*public class Dog {  
}

工厂类

package com.example.demo.设计模式.简单工厂模式;  
  
*/\*\*  
 \* 创建工厂类  
 \*/*public class AnimalFactory {  
  
 //创建获取Dog实例的方法  
 public static Dog getDog(){  
 return new Dog();  
 }  
 //创建获取Cat实例的方法  
 public static Cat getCat(){  
 return new Cat();  
 }  
}

package com.example.demo.设计模式.简单工厂模式;  
  
*/\*\*  
 \* 测试类  
 \*/*public class Test {  
 public static void main(String[] args) {  
 Dog dog = AnimalFactory.*getDog*();  
 Cat cat = AnimalFactory.*getCat*();  
 }  
}

改进》》》》》》》》》》》》》》》》》》》》》》》》》》》》》》》》》》》》》》》》》》》》》》》》》》》》》》》》》》》》》》

创建动物类作为父类：

package com.example.demo.设计模式.简单工厂模式;  
  
public class Animal {  
}

创建Dog类并继承Animal类：

package com.example.demo.设计模式.简单工厂模式;  
  
*/\*\*  
 \* 创建Dog类  
 \*/*public class Dog extends Animal{  
}

创建Cat类并继承Animal类：

package com.example.demo.设计模式.简单工厂模式;  
  
*/\*\*  
 \* 创建Cat类  
 \*/*public class Cat extends Animal{  
}

创建工厂类：

package com.example.demo.设计模式.简单工厂模式;  
  
*/\*\*  
 \* 创建工厂类  
 \*/*public class AnimalFactory {  
  
 */\*\*  
 \* 根据传入的参数判断返回什么  
 \* 缺点：可能返回为null，当调用时空指针异常  
 \** ***@param*** *name  
 \** ***@return*** *\*/* public static Animal getAnimal(String name){  
 */\*\*根据传入的参数，判断返回哪个实例\*/* switch (name){  
 case "Dog":  
 return new Dog();  
 case "Cat":  
 return new Cat();  
 default:  
 return null;  
 }  
 }  
}

测试类：

package com.example.demo.设计模式.简单工厂模式;  
  
*/\*\*  
 \* 测试类  
 \*/*public class Test {  
  
 public static void main(String[] args) {  
 //获取实例并强转  
 Dog dog = (Dog) AnimalFactory.*getAnimal*("Dog");  
 Cat cat = (Cat) AnimalFactory.*getAnimal*("Cat");  
  
 }  
}

## 12.2 工厂方法模式 – 生产一类实例（拥有共同父类）

1）工厂方法模式：抽象工厂定义接口，工厂实现类生产

== 工厂方法模式中抽象工厂类负责定义创建对象的接口，具体对象的创建工作又继承了抽象工厂的具体类实现。

2）优点

== 客户端不需要再负责对的创建，从而明确了各个类的职责，如果有新对象增加，只需要增加一个具体的类和具体的工厂类即可，不影响已有的代码，后期维护容易，增强了系统的扩展性

3）缺点

== 需要额外的编写代码，增加了工作量。

创建动物类作为父类：

package com.example.demo.设计模式.工厂方法模式;  
  
public class Animal {  
}

创建Dog类并继承Animal类：

package com.example.demo.设计模式.工厂方法模式;  
  
*/\*\*  
 \* 创建Dog类  
 \*/*public class Dog extends Animal{  
}

创建Cat类并继承Animal类：

package com.example.demo.设计模式.工厂方法模式;  
  
*/\*\*  
 \* 创建Cat类  
 \*/*public class Cat extends Animal{  
}

工厂接口

package com.example.demo.设计模式.工厂方法模式;  
  
*/\*\*  
 \* 创建抽象工厂接口类  
 \*/*public interface AnimalFactory {  
 Object getInstance();  
}

工厂实现类

package com.example.demo.设计模式.工厂方法模式;  
  
*/\*\*  
 \* 创建工厂实现类CatFactory  
 \*/*public class CatFactory implements AnimalFactory{  
  
 @Override  
 public Object getInstance() {  
 return new Cat();  
 }  
}

package com.example.demo.设计模式.工厂方法模式;  
  
*/\*\*  
 \* 创建工厂实现类DogFactory  
 \*/*public class DogFactory implements AnimalFactory {  
 @Override  
 public Object getInstance() {  
 return new Dog();  
 }  
}

package com.example.demo.设计模式.工厂方法模式;  
  
*/\*\*  
 \* 测试类  
 \*/*public class Test {  
 public static void main(String[] args) {  
 //创建工厂  
 DogFactory dogFactory = new DogFactory();  
 CatFactory catFactory = new CatFactory();  
 //获取实例并强转  
 Dog dog = (Dog) dogFactory.getInstance();  
 Cat cat = (Cat) catFactory.getInstance();  
 }  
}

## 12.3 抽象工厂模式

12.3.1 类准备

package com.example.demo.设计模式.抽象工厂模式;  
  
*/\*\*  
 \* 动物接口  
 \*/*public interface Animal {  
}

package com.example.demo.设计模式.抽象工厂模式;  
  
*/\*\*  
 \* Cat类  
 \*/*public class Cat implements Animal {  
}

package com.example.demo.设计模式.抽象工厂模式;  
  
*/\*\*  
 \* Dog类  
 \*/*public class Dog implements Animal {  
}

package com.example.demo.设计模式.抽象工厂模式;  
  
*/\*\*  
 \* Person接口  
 \*/*public interface Person {  
}

package com.example.demo.设计模式.抽象工厂模式;  
  
*/\*\*  
 \* Man类  
 \*/*public class Man implements Person {  
}

package com.example.demo.设计模式.抽象工厂模式;  
  
*/\*\*  
 \* Woman类  
 \*/*public class Woman implements Person {  
}

12.3.2 工厂创建

package com.example.demo.设计模式.抽象工厂模式;  
  
*/\*\*  
 \* 抽象工厂  
 \* 创建超级工厂：抽象工厂模式  
 \* 能够生产多种类型的产品：动物+人  
 \* 简单工厂和工厂方法模式大多是生产一种类型：动物 或者 人  
 \*/*public abstract class SuperFactory {  
 public abstract Animal getAnimal(String name);  
 public abstract Person getPerson(String name);  
}

package com.example.demo.设计模式.抽象工厂模式;  
  
*/\*\*  
 \* 抽象接口实现类  
 \*/*public class FirstFactory extends SuperFactory {  
 @Override  
 public Animal getAnimal(String name) {  
 switch (name){  
 case "Dog":  
 return new Dog();  
 case "cat":  
 return new Cat();  
 default:  
 return null;  
 }  
 }  
  
 @Override  
 public Person getPerson(String name) {  
 switch (name){  
 case "Man":  
 return new Man();  
 case "Woman":  
 return new Woman();  
 default:  
 return null;  
 }  
 }  
}

package com.example.demo.设计模式.抽象工厂模式;  
  
*/\*\*  
 \* 测试类  
 \*/*public class Test {  
  
 public static void main(String[] args) {  
 //创建超级工厂子类实例  
 FirstFactory ff = new FirstFactory();  
 //获取动物实例  
 Dog dog = (Dog) ff.getAnimal("Dog");  
 Cat cat = (Cat) ff.getAnimal("Cat");  
 //获取人实例  
 Man man = (Man) ff.getPerson("Man");  
 Woman woman = (Woman) ff.getPerson("Woman");  
  
 }  
}

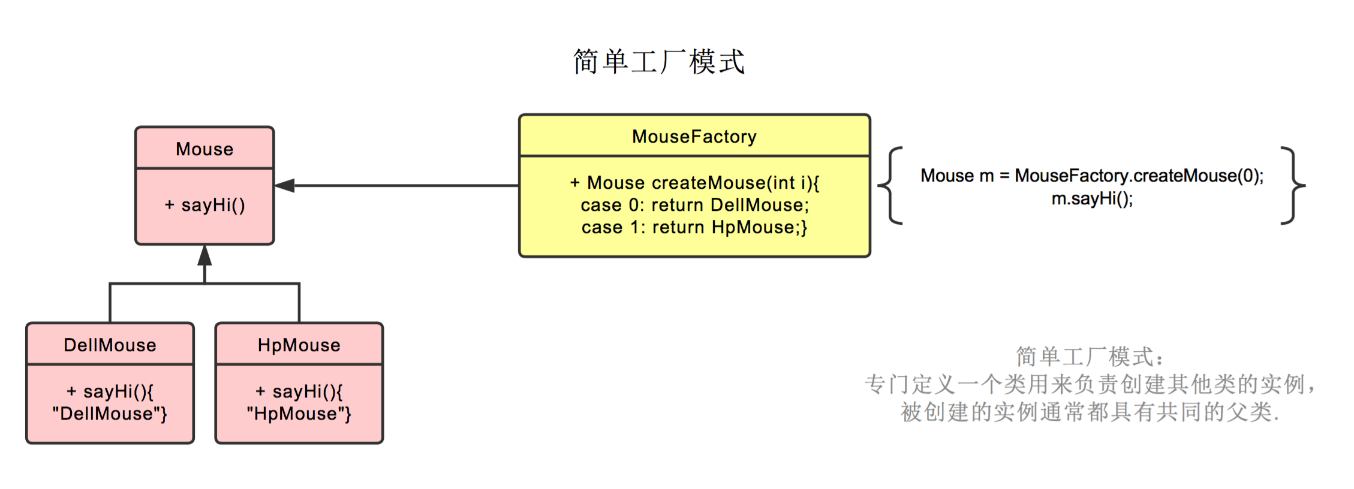
## 三种模式讲解

下面例子中鼠标，键盘，耳麦为产品，惠普，戴尔为工厂。

### 简单工厂模式

简单工厂模式不是 23 种里的一种，简而言之，就是有一个专门生产某个产品的类。

比如下图中的鼠标工厂，专业生产鼠标，给参数 0，生产戴尔鼠标，给参数 1，生产惠普鼠标。



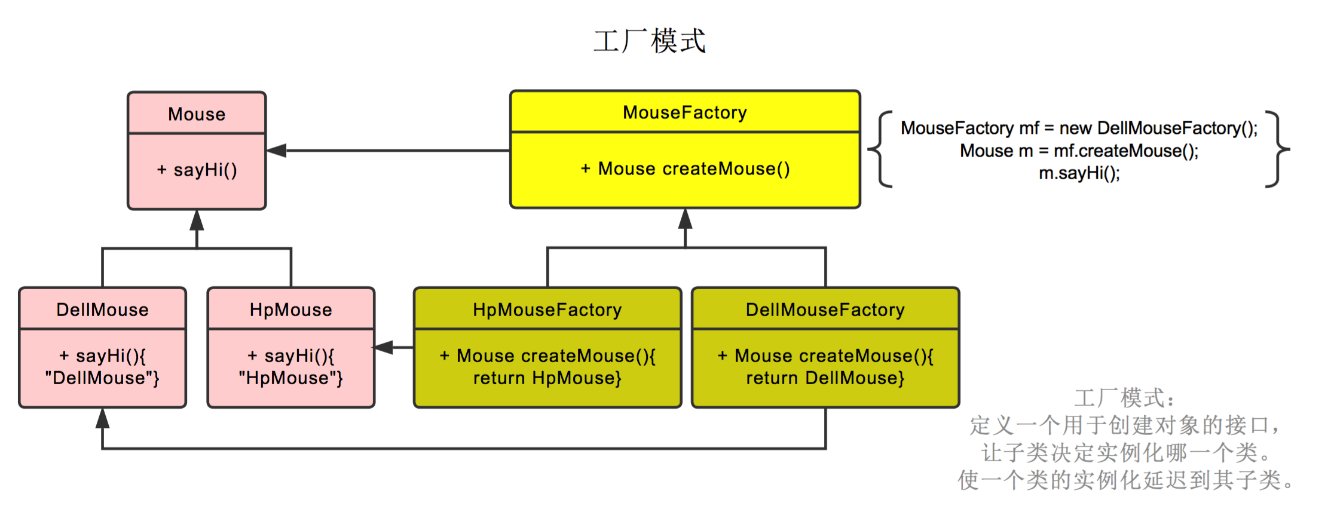
### 工厂模式

工厂模式也就是鼠标工厂是个父类，有生产鼠标这个接口。

戴尔鼠标工厂，惠普鼠标工厂继承它，可以分别生产戴尔鼠标，惠普鼠标。

生产哪种鼠标不再由参数决定，而是创建鼠标工厂时，由戴尔鼠标工厂创建。

后续直接调用**鼠标工厂.生产鼠标()**即可



### 抽象工厂模式

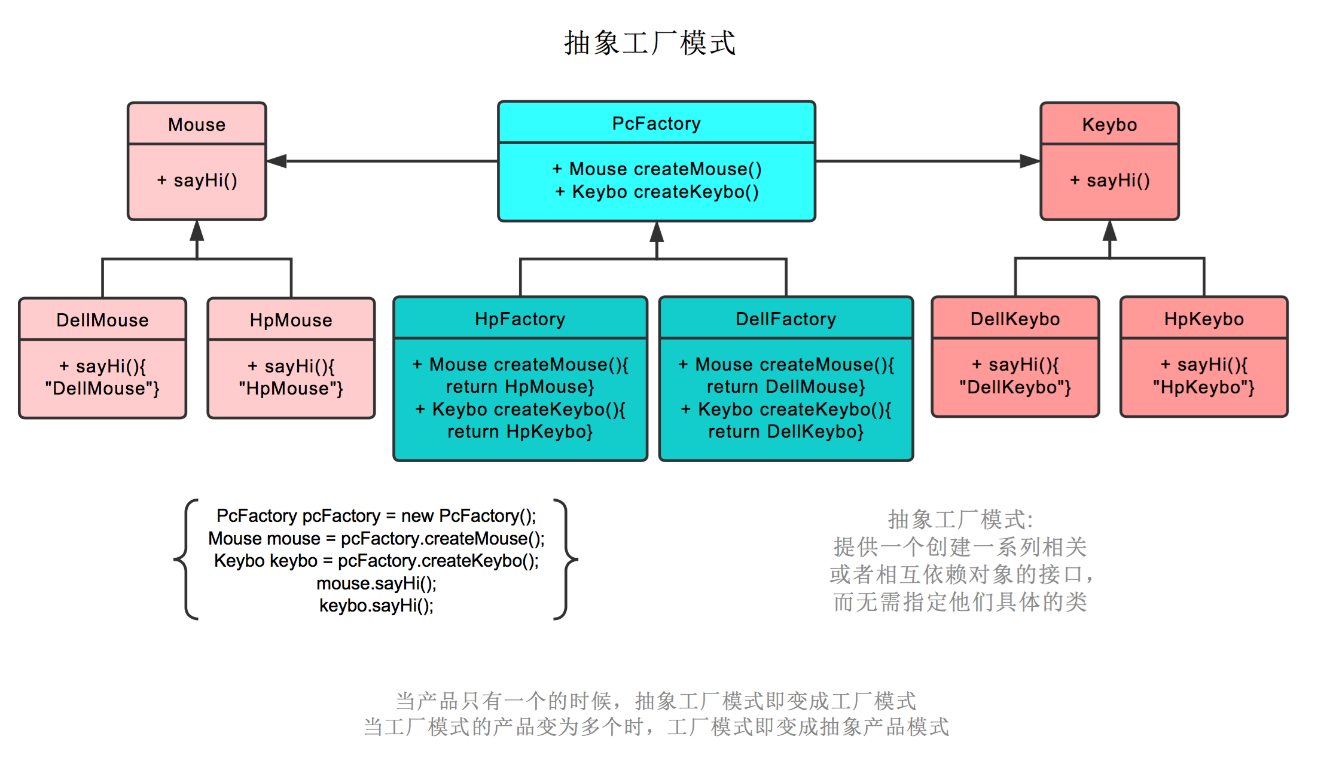
抽象工厂模式也就是不仅生产鼠标，同时生产键盘。

也就是 PC 厂商是个父类，有生产鼠标，生产键盘两个接口。

戴尔工厂，惠普工厂继承它，可以分别生产戴尔鼠标+戴尔键盘，和惠普鼠标+惠普键盘。

创建工厂时，由戴尔工厂创建。

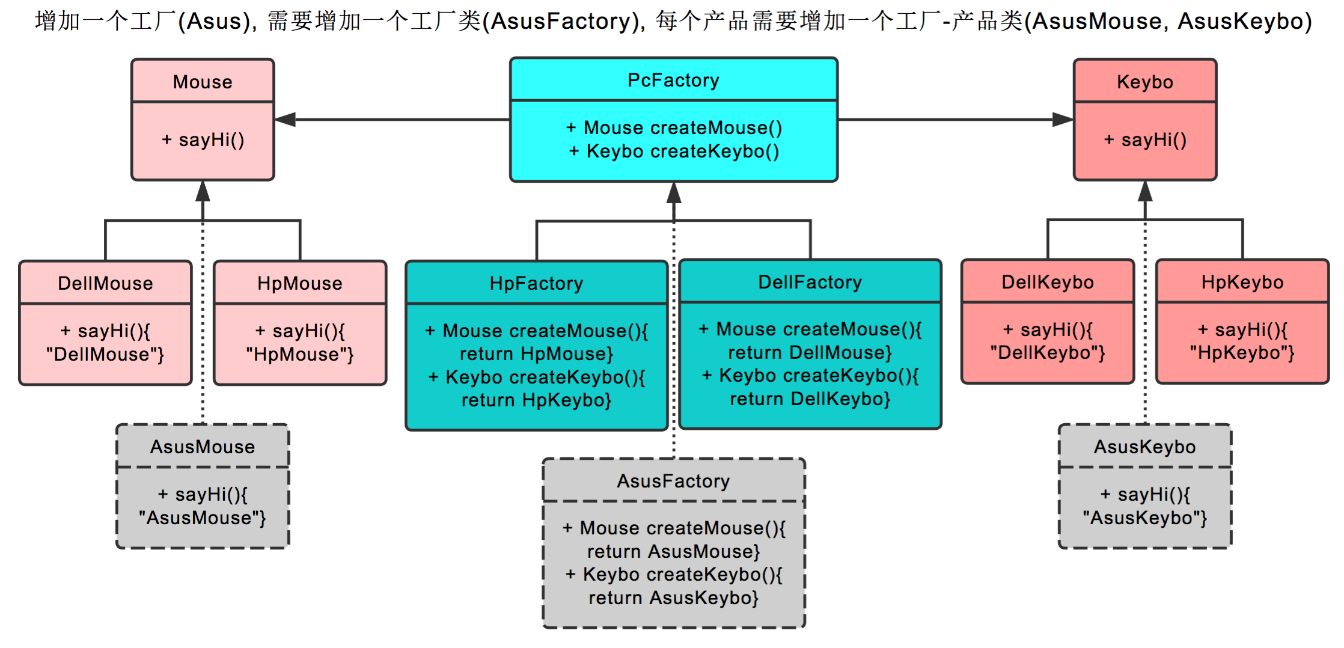
后续**工厂.生产鼠标()**则生产戴尔鼠标，**工厂.生产键盘()**则生产戴尔键盘。



在抽象工厂模式中，假设我们需要增加一个工厂

假设我们增加华硕工厂，则我们需要增加华硕工厂，和戴尔工厂一样，继承 PC 厂商。

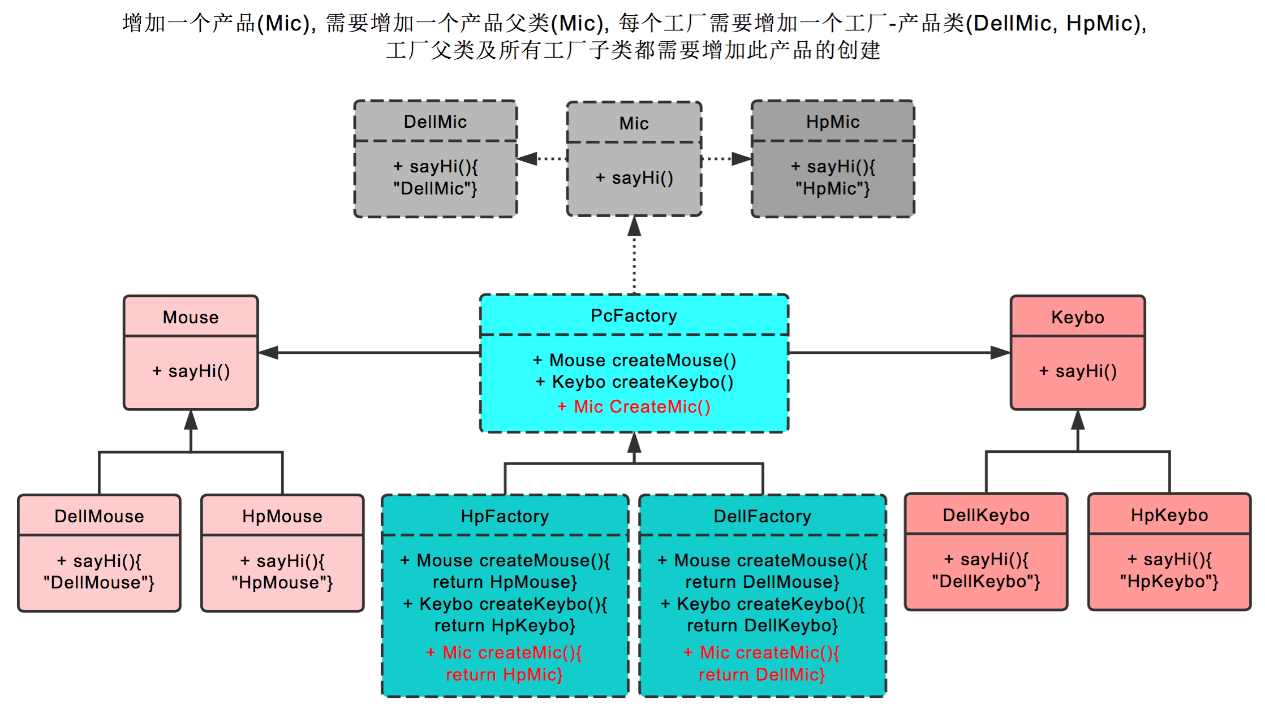
之后创建华硕鼠标，继承鼠标类。创建华硕键盘，继承键盘类即可。



在抽象工厂模式中，假设我们需要增加一个产品

假设我们增加耳麦这个产品，则首先我们需要增加耳麦这个父类，再加上戴尔耳麦，惠普耳麦这两个子类。

之后在PC厂商这个父类中，增加生产耳麦的接口。最后在戴尔工厂，惠普工厂这两个类中，分别实现生产戴尔耳麦，惠普耳麦的功能。 以上。



# 设计模式简介

设计模式（Design pattern）代表了最佳的实践，通常被有经验的面向对象的软件开发人员所采用。设计模式是软件开发人员在软件开发过程中面临的一般问题的解决方案。这些解决方案是众多软件开发人员经过相当长的一段时间的试验和错误总结出来的。

设计模式是一套被反复使用的、多数人知晓的、经过分类编目的、代码设计经验的总结。使用设计模式是为了重用代码、让代码更容易被他人理解、保证代码可靠性。 毫无疑问，设计模式于己于他人于系统都是多赢的，设计模式使代码编制真正工程化，设计模式是软件工程的基石，如同大厦的一块块砖石一样。项目中合理地运用设计模式可以完美地解决很多问题，每种模式在现实中都有相应的原理来与之对应，每种模式都描述了一个在我们周围不断重复发生的问题，以及该问题的核心解决方案，这也是设计模式能被广泛应用的原因。

## 设计模式的使用

设计模式在软件开发中的两个主要用途。

### 开发人员的共同平台

设计模式提供了一个标准的术语系统，且具体到特定的情景。例如，单例设计模式意味着使用单个对象，这样所有熟悉单例设计模式的开发人员都能使用单个对象，并且可以通过这种方式告诉对方，程序使用的是单例模式。

### 最佳的实践

设计模式已经经历了很长一段时间的发展，它们提供了软件开发过程中面临的一般问题的最佳解决方案。学习这些模式有助于经验不足的开发人员通过一种简单快捷的方式来学习软件设计。

## 设计模式的类型

根据设计模式的参考书 **Design Patterns - Elements of Reusable Object-Oriented Software（中文译名：设计模式 - 可复用的面向对象软件元素）** 中所提到的，总共有 23 种设计模式。这些模式可以分为三大类：创建型模式（Creational Patterns）、结构型模式（Structural Patterns）、行为型模式（Behavioral Patterns）。当然，我们还会讨论另一类设计模式：J2EE 设计模式。

|  |  |  |
| --- | --- | --- |
| **序号** | **模式 & 描述** | **包括** |
| 1 | **创建型模式** 这些设计模式提供了一种在创建对象的同时隐藏创建逻辑的方式，而不是使用 new 运算符直接实例化对象。这使得程序在判断针对某个给定实例需要创建哪些对象时更加灵活。 | * 工厂模式（Factory Pattern） * 抽象工厂模式（Abstract Factory Pattern） * 单例模式（Singleton Pattern） * 建造者模式（Builder Pattern） * 原型模式（Prototype Pattern） |
| 2 | **结构型模式** 这些设计模式关注类和对象的组合。继承的概念被用来组合接口和定义组合对象获得新功能的方式。 | * 适配器模式（Adapter Pattern） * 桥接模式（Bridge Pattern） * 过滤器模式（Filter、Criteria Pattern） * 组合模式（Composite Pattern） * 装饰器模式（Decorator Pattern） * 外观模式（Facade Pattern） * 享元模式（Flyweight Pattern） * 代理模式（Proxy Pattern） |
| 3 | **行为型模式** 这些设计模式特别关注对象之间的通信。 | * 责任链模式（Chain of Responsibility Pattern） * 命令模式（Command Pattern） * 解释器模式（Interpreter Pattern） * 迭代器模式（Iterator Pattern） * 中介者模式（Mediator Pattern） * 备忘录模式（Memento Pattern） * 观察者模式（Observer Pattern） * 状态模式（State Pattern） * 空对象模式（Null Object Pattern） * 策略模式（Strategy Pattern） * 模板模式（Template Pattern） * 访问者模式（Visitor Pattern） |
| 4 | **J2EE 模式** 这些设计模式特别关注表示层。这些模式是由 Sun Java Center 鉴定的。 | * MVC 模式（MVC Pattern） * 业务代表模式（Business Delegate Pattern） * 组合实体模式（Composite Entity Pattern） * 数据访问对象模式（Data Access Object Pattern） * 前端控制器模式（Front Controller Pattern） * 拦截过滤器模式（Intercepting Filter Pattern） * 服务定位器模式（Service Locator Pattern） * 传输对象模式（Transfer Object Pattern） |

下面用一个图片来整体描述一下设计模式之间的关系：



## 设计模式的六大原则

**1、开闭原则（Open Close Principle）**

开闭原则的意思是：**对扩展开放，对修改关闭**。在程序需要进行拓展的时候，不能去修改原有的代码，实现一个热插拔的效果。简言之，是为了使程序的扩展性好，易于维护和升级。想要达到这样的效果，我们需要使用接口和抽象类，后面的具体设计中我们会提到这点。

**2、里氏代换原则（Liskov Substitution Principle）**

里氏代换原则是面向对象设计的基本原则之一。 里氏代换原则中说，任何基类可以出现的地方，子类一定可以出现。LSP 是继承复用的基石，只有当派生类可以替换掉基类，且软件单位的功能不受到影响时，基类才能真正被复用，而派生类也能够在基类的基础上增加新的行为。里氏代换原则是对开闭原则的补充。实现开闭原则的关键步骤就是抽象化，而基类与子类的继承关系就是抽象化的具体实现，所以里氏代换原则是对实现抽象化的具体步骤的规范。

**3、依赖倒转原则（Dependence Inversion Principle）**

这个原则是开闭原则的基础，具体内容：针对接口编程，依赖于抽象而不依赖于具体。

**4、接口隔离原则（Interface Segregation Principle）**

这个原则的意思是：使用多个隔离的接口，比使用单个接口要好。它还有另外一个意思是：降低类之间的耦合度。由此可见，其实设计模式就是从大型软件架构出发、便于升级和维护的软件设计思想，它强调降低依赖，降低耦合。

**5、迪米特法则，又称最少知道原则（Demeter Principle）**

最少知道原则是指：一个实体应当尽量少地与其他实体之间发生相互作用，使得系统功能模块相对独立。

**6、合成复用原则（Composite Reuse Principle）**

合成复用原则是指：尽量使用合成/聚合的方式，而不是使用继承。

# 五 》网络编程

关键点：

网络编程三要素：

IP

port

协议

UDP

面向无连接，不安全，速度快，都是客户端

关键类：

DatagramSocket 码头

Datagrampacket 集装箱

关键方法；

ds.send(data.getBytes(),length,ip,port)

ds.receive(dp);

dp.getData()

dp.getLength()

TCP

面向连接，安全，速度略低，区分服务端和客户端

三次握手：请求 响应 传输

Socket s = new Socket(ip,port);

ServerSocket ss = new ServerSocket(port);

ss.accept();

# 1 网络编程概述

1）计算机网络：

== 是指将地理位置不同的具有独立功能的多台计算机及其外部设备，通过通信线路连接起来，在网络操作系统，网络管理软件及网络通信协议的管理和协调下，实现资源共享和信息传递的计算机系统。

2）网络编程

== 就是用来实现网络互连的不同计算机上运行的程序间可以进行数据交换。

# 2 网络编程三要素：IP | 端口 | 协议

## 2.1 网络编程三要素之----IP概述

== 每个设备在网络中的唯一标识

== 每台网络终端在网络中都有一个独立的地址，我们在网络中传输数据就是使用这个

地址。

== （Windows） ipconfig ： 查看本机IP

（Linux）addr

== ping：测试连接（Windows和Linux都能用）

Linux中测试连接：CURL http://ip:port

== 本地会路地址：127.0.0.1 255.255.255 是广播地址

== IPv4 ： 4个字节组成， 4个0-255 大概42亿，30亿都在北美，亚洲4亿，2011

年初已经用尽

== IPv6 ：8组，每组6个16进制数

== 1a2b ： 0000 : aaaa : 0000 : 0000 : 0000 : aabb : 1f2f

== 1a2b : : aaaa : 0000 : 0000 : 0000 : aabb : 1f2f

== 1a2b : 0000 : aaaa : : aabb : 1f2f

== 1a2b : 0000 : aaaa : : 0000 : aabb : 1f2f

== 1a2b : 0000 : aaaa : 0000 : : aabb : 1f2f

## 2.2 网络编程三要素之----端口号

== 每个程序在设备上的唯一标识

== 每个网络程序都需要绑定一个端口号，传输数据的时候除了确定发到哪台机器上

，还要明确发到哪个程序；

== 端口号范围：0-65535

== 编写网络应用就需要绑定一个端口，尽量使用1024以上的，1024一下的基本被

系统程序占用了。

== 常用端口：

mysql 3306

oracle 1521

web 80

tomcat 8080

QQ 4000

## 2.3 网络编程三要素之----协议

== 为计算机在网络中进行数据交换而建立的规则，标准或约定的集合

== UDP

面向无连接，数据不安全，速度快，不区分客户端和服务端

== TCp

面向连接（三次握手），数据安全，速度略低，分为客户端和服务端

三次握手：1 客户端发起请求 2 服务端响应请求 3 传输数据

（约吗 约 开吃）

# 3 Socket通信原理

Socket套接字概述：

== 网络上具有唯一标识IP地址和端口号组合在一起才能构成唯一能识别的标识

符套接字

== 通信的两端都有Socket

== 网络通信其实就是Socket间的通信

== 数据在两个Socket间通过IO传输

== Socket在应用程序中创建，通过一种绑定机制与驱动程序建立关系，告诉自己 所对应的IP和Port

# 4 UDP协议 DatagramSocket | DatagramPacket

UDP : user datagram protocol 用户数据报协议

datagram数据报

socket 插座（套接字）

packet（小包，信息包）- >package

DatagramSocket 数据套接字 --连接管道

DatagramPacket 数据包 --数据载体

发送时：DatagramPacket需要携带对方信息：数据的数组，数组长度，对方IP， 对方端口

接收时：DatagramPacket 只需要准备接收工具就好：数组，数组长度

## 4.1 发送send

== 创建DatagramSocket 随机端口号

== 创建DatagramPacket 指定数据，长度，地址，端口

== 使用DatagramSocket 发送DatagramPacket

== 关闭DatagramSocket

package com.example.demo.网络编程.UDP.TCP.UDP;  
  
import java.io.IOException;  
import java.net.\*;  
  
*/\*\*  
 \* 创建发送数据端  
 \*/*public class MyDatagramSocket {  
 public static void main(String[] args) throws IOException {  
 //1 创建码头：套接字 (可以不指定端口号，随机端口号)  
 DatagramSocket datagramSocket = new DatagramSocket();  
  
 String data = "你是色";  
 //2 创建集装箱：数据包 （参数：数据字节数组 ，字节数组长度 ，ip ，port）  
 DatagramPacket datagrampacket = new DatagramPacket(data.getBytes(),data.getBytes().length,  
 InetAddress.*getLocalHost*(),8888);  
 //3 使用码头发送集装箱  
 datagramSocket.send(datagrampacket);  
 //4 关闭码头  
 datagramSocket.close();  
  
 }  
  
  
}

## 4.2 接收receive

== 创建DatagramSocket 指定端口号

== 创建DatagramPacket 指定数组，长度

== 使用DatagramSocket 接收DatagramPacket

== 关闭DatagramSocket

== 从DatagramPacket中获取数据

package com.example.demo.网络编程.UDP.TCP.UDP;  
  
import java.io.IOException;  
import java.net.DatagramPacket;  
import java.net.DatagramSocket;  
  
*/\*\*  
 \* 创建接收数据端  
 \*/*public class Client {  
  
 public static void main(String[] args) throws IOException {  
  
 // 1 创建码头：套接字（指定接收端口号）  
 DatagramSocket datagramSocket = new DatagramSocket(8888);  
  
 // 2 创建集装箱：数据包（接收数组，数组长度）  
 DatagramPacket datagramPacket = new DatagramPacket(new byte[1024\*8],1024\*8);  
  
 // 3 码头用集装箱接货 : 接收数据  
 datagramSocket.receive(datagramPacket);  
  
 // 4 获取数据包中的数据  
 byte[] arr = datagramPacket.getData();  
 // 5 获取有效字节个数  
 int len = datagramPacket.getLength();  
  
 //将byte数组转为字符串  
 String str = new String(arr,0,len);  
  
 //打印数据  
 System.*out*.println(str);  
  
 // 6 关闭  
 datagramSocket.close();  
 }  
}

## 4.3 接收方获取Ip和端口号

== String ip = packet.getAddress().getHostAddress()

== int port = packet.getPort()

## 4.4 优化 – 多线程

### 创建接收线程：

package com.example.demo.网络编程.UDP.TCP.UDP.优化2;  
  
import java.io.IOException;  
import java.net.DatagramPacket;  
import java.net.DatagramSocket;  
  
public class ReceiveThread extends Thread {  
 private DatagramSocket socket;  
 private DatagramPacket packet;  
 public ReceiveThread(DatagramSocket socket,DatagramPacket packet){  
 this.socket = socket;  
 this.packet = packet;  
 }  
  
 public void run(){  
 while (true){  
 // 3 使用码头接收数据  
 try {  
 socket.receive(packet);  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 // 4 获取数据  
 byte[] arr = packet.getData();  
 int len = packet.getLength();  
  
 String str = new String(arr,0,len);  
 System.*out*.println(packet.getAddress()+"发来数据："+str);  
 System.*out*.println("请输入数据：");  
 }  
 }  
  
  
}

### 创建发送线程：

package com.example.demo.网络编程.UDP.TCP.UDP.优化2;  
  
import java.io.IOException;  
import java.net.DatagramPacket;  
import java.net.DatagramSocket;  
import java.net.InetAddress;  
import java.net.UnknownHostException;  
import java.util.Scanner;  
  
public class SendThread extends Thread{  
  
 //码头：套接字  
 private DatagramSocket socket;  
 //集装箱：数据包  
 private DatagramPacket packet;  
 //端口号  
 private Integer port;  
  
 public SendThread(DatagramSocket socket,Integer port){  
 this.socket = socket;  
 this.port = port;  
 }  
  
 */\*\*  
 \* 获取键盘输入的信息*

*输入send发送数据*

*不是send则添加到StringBuilder中，统一发送  
 \** ***@return*** *\*/* public String getData(){  
 //获取控制台数据  
 Scanner scanner = new Scanner(System.*in*);  
 //创建StrnigBuilder 来收集输入的数据  
 StringBuilder stringBuilder = new StringBuilder();  
  
 //while (scanner.hasNext()){

while (true){  
 // 接收数据赋值给data  
 String data = scanner.nextLine();  
 //判断data等于send发送数据  
 if ("send".equals(data)){  
 return stringBuilder.toString();  
 }  
 //如果data不是send，则直接追加到stringBuilder上  
 stringBuilder.append(data);  
 }  
 return stringBuilder.toString();  
 }  
  
 */\*\*  
 \* 发送数据  
 \*/* public void run(){  
 System.*out*.println("请输入数据：");  
 while (true){  
 String data = getData();  
 // 2 创建集装箱：数据包  
 try {  
 packet = new DatagramPacket(data.getBytes(),data.getBytes().length,  
 InetAddress.*getLocalHost*(),port);  
 } catch (UnknownHostException e) {  
 e.printStackTrace();  
 }  
  
 try {  
 socket.send(packet);  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 System.*out*.println("发送成功。。");  
 System.*out*.println("请输入数据：");  
 }  
  
 }  
}

### 创建客户端1：

package com.example.demo.网络编程.UDP.TCP.UDP.优化2;  
  
import java.io.IOException;  
import java.net.DatagramPacket;  
import java.net.DatagramSocket;  
import java.net.InetAddress;  
import java.net.UnknownHostException;  
import java.util.Scanner;  
  
public class Send {  
  
 public static void main(String[] args) throws IOException {  
  
 // 1 码头：创建套接字  
 DatagramSocket socket = new DatagramSocket(9999);  
 */\*\*发送线程\*/* SendThread st = new SendThread(socket,8888);  
 st.start();  
  
 */\*\* 接收线程 \*/* DatagramPacket packet = new DatagramPacket(new byte[1024],1024);  
 ReceiveThread rt = new ReceiveThread(socket,packet);  
 rt.start();  
  
 }  
}

### 创建客户端2：

package com.example.demo.网络编程.UDP.TCP.UDP.优化2;  
  
import java.io.IOException;  
import java.net.DatagramPacket;  
import java.net.DatagramSocket;  
import java.net.InetAddress;  
import java.net.UnknownHostException;  
import java.util.Scanner;  
  
public class Receive {  
  
 public static void main(String[] args) throws IOException {  
 // 1 创建码头： 套接字  
 DatagramSocket socket = new DatagramSocket(8888);  
  
 */\*\*接收线程\*/* // 2 创建集装箱：数据包  
 DatagramPacket packet = new DatagramPacket(new byte[1024],1024);  
 ReceiveThread rt = new ReceiveThread(socket,packet);  
 rt.start();  
  
 */\*\*发送线程\*/* SendThread st = new SendThread(socket,9999);  
 st.start();  
 }  
  
}

## 4.5 UI界面的聊天室

package com.example.demo.网络编程.UDP.TCP.UDP.UI;  
  
import javax.swing.\*;  
import java.awt.\*;  
import java.awt.event.ActionEvent;  
import java.awt.event.ActionListener;  
import java.awt.event.WindowAdapter;  
import java.awt.event.WindowEvent;  
import java.io.\*;  
import java.net.\*;  
import java.text.SimpleDateFormat;  
import java.util.Date;  
  
public class MyFrame extends JFrame {  
 //南部面板组件  
 JButton send ;  
 JButton log ;  
 JButton clear;  
 JButton shake ;  
 TextField textField ;  
  
 //中间面板组件  
 TextArea sendArea;  
 TextArea viewArea;  
  
 //套接字  
 DatagramSocket socket;  
  
 //聊天记录  
 BufferedWriter bw;  
 BufferedReader br;  
  
 boolean flag = true;  
  
 public MyFrame() throws IOException {  
 super("聊天室");  
 Container c = this.getContentPane();  
 setBounds(300, 300, 500, 600);  
 setDefaultCloseOperation(*DISPOSE\_ON\_CLOSE*);  
  
 // 码头：套接字  
 //socket = new DatagramSocket(8888);  
 socket = new DatagramSocket();  
 //聊天记录  
 bw = new BufferedWriter(new FileWriter("log.txt",true));  
  
 //中部面板  
 centerPanel(c);  
 //南部面板  
 southPanel(c);  
 //监听事件  
 event();  
 //接收信息的线程  
 new Receive().start();  
  
 }  
 */\*\*  
 \* 中间面板  
 \** ***@param*** *c  
 \*/* public void centerPanel( Container c){  
 //两个面板  
 JPanel center = new JPanel();  
 center.setLayout(new BorderLayout());  
 //显示面板  
 viewArea = new TextArea(null,35,20 ,TextArea.*SCROLLBARS\_VERTICAL\_ONLY*);  
 sendArea = new TextArea(null,7,20,TextArea.*SCROLLBARS\_VERTICAL\_ONLY*);  
 //设置文本区属性  
 viewArea.setEditable(false); //不可编辑  
 viewArea.setBackground(Color.*WHITE*);//白色背景  
 viewArea.setFont(new Font("仿宋",Font.*PLAIN*,18)); //设置字体  
  
 sendArea.setFont(new Font("仿宋",Font.*PLAIN*,18)); //设置字体  
  
  
 center.add(viewArea);  
 center.add(sendArea,BorderLayout.*SOUTH*);  
 //面板加进容器  
 c.add(center,BorderLayout.*CENTER*);  
 }  
  
 */\*\*  
 \* 南部面板  
 \** ***@param*** *c  
 \*/* public void southPanel( Container c){  
 //创建面板  
 JPanel pane = new JPanel();  
 //创建组件  
 send = new JButton("发送");  
 log = new JButton("记录");  
 clear = new JButton("清屏");  
 shake = new JButton("震动");  
 textField = new TextField(25);  
 textField.setText("127.0.0.1:9999");  
 //textField.setText("127.0.0.1:8888");  
 //添加组件到面板  
 pane.add(textField);  
 pane.add(send);  
 pane.add(log);  
 pane.add(clear);  
 pane.add(shake);  
 //添加面板到容器  
 c.add(pane,BorderLayout.*SOUTH*);  
 }  
  
 */\*\*  
 \* 发送信息  
 \*/* private void send() throws IOException {  
 String message = sendArea.getText(); //获取发送内容  
  
 String ipAndPort = textField.getText(); //获取ip地址和端口  
 //如果为空，则认为是群发，改为255.255.255.255 （共享IP）  
 String ip = null;  
 String port = null;  
 if(ipAndPort.trim().length() == 0){  
 ip = "255.255.255.255";  
 port = "10000";  
 }else {  
 int index = ipAndPort.indexOf(":");  
 ip = ipAndPort.substring(0, index);  
 port = ipAndPort.substring(index + 1);  
 }

send(message.getBytes(),ip,port);  
  
 //将数据追加到viewArea  
 String time = getCurrentTime();  
 viewArea.append("\r\n["+time+"]\r\n");  
 viewArea.append("我对 "+ipAndPort+"说 ："+message);  
 //将信息写入日志  
 bw.write("\r\n["+time+"]\r\n"+"我对 "+ipAndPort+"说 ："+message);  
 bw.flush();  
  
 //在将发送区域清除  
 //127.0.0.1:9999  
 sendArea.setText("");  
 }  
 */\*\*  
 \* 方法的重载  
 \** ***@param*** *\** ***@throws*** *IOException  
 \*/* private void send(byte[] arr,String ip,String port) throws IOException {  
 // 集装箱：数据包  
 DatagramPacket packet = new DatagramPacket(arr,  
 arr.length,InetAddress.*getByName*(ip),Integer.*parseInt*(port));  
 //发送数据  
 socket.send(packet);  
 //关闭套接字  
 //socket.close(); 不能在此关闭，要在关闭窗口后关闭  
 }  
 */\*\*  
 \* 获取信息的内部类  
 \*/* private class Receive extends Thread{  
 DatagramPacket packet ;  
 DatagramSocket socket;  
 public Receive() throws SocketException {  
 packet = new DatagramPacket(new byte[1024],1024);  
 socket = new DatagramSocket(8888);  
 //socket = new DatagramSocket(9999);  
 }  
 public void run(){  
 while (flag){  
 try {  
 socket.receive(packet);  
 byte[] arr = packet.getData();  
 int len = packet.getLength();  
  
 //判断是否震动  
 //必须是： 第一个字节是-1，（注意使字节 -1）  
 // 该次数据有效长度为 1  
 if(arr[0] == -1 && len == 1){  
 shake();  
 continue; //不执行一下操作，继续循环  
 }  
  
 String str = new String(arr,0,len);  
 String time = getCurrentTime();  
 String ip = packet.getAddress().getHostAddress();  
 viewArea.append("\r\n["+time+"]\r\n");  
 viewArea.append(ip+" 对我说："+str);  
 //接收到后写入日志  
 bw.write("\r\n["+time+"]\r\n"+ip+" 对我说："+str);  
 bw.flush();  
 } catch (SocketException e) {  
 e.printStackTrace();  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 }  
 if(!flag){  
 socket.close();  
 }  
 }  
 }  
  
 */\*\*  
 \* 事件整合  
 \*/* public void event(){  
 // 发送按钮监听  
 send.addActionListener(new ActionListener() {  
 @Override  
 public void actionPerformed(ActionEvent e) {  
 try {  
 send();  
 } catch (IOException e1) {  
 e1.printStackTrace();  
 }  
 }  
 });

//发送快捷键  
send.addKeyListener(new KeyAdapter() {  
 @Override  
 public void keyPressed(KeyEvent e) {  
 //ctr+enter发送  
 //if(e.getKeyCode() == KeyEvent.VK\_ENTER && e.isControlDown()){  
 // 按回车键发送  
 if(e.getKeyCode() == KeyEvent.*VK\_ENTER*){  
 try {  
 send();  
 } catch (IOException e1) {  
 e1.printStackTrace();  
 }  
 }  
 }  
});

//关闭窗口监听  
 addWindowListener(new WindowAdapter() {  
 @Override  
 public void windowClosing(WindowEvent e) {  
 //关闭窗口时，关闭套接字  
 socket.close();  
 //关闭窗口时，关闭输出流  
 flag = false;  
 try {  
 bw.close();  
 } catch (IOException e1) {  
 e1.printStackTrace();  
 }  
 //窗口关闭，退出虚拟机  
 System.*exit*(0);  
 }  
 });  
  
 //聊天记录  
 log.addActionListener(new ActionListener() {  
 @Override  
 public void actionPerformed(ActionEvent e) {  
 logFile();  
 }  
 });  
  
 //清屏  
 clear.addActionListener(new ActionListener() {  
 @Override  
 public void actionPerformed(ActionEvent e) {  
 viewArea.setText("");  
 }  
 });  
  
 //震动  
 shake.addActionListener(new ActionListener() {  
 @Override  
 public void actionPerformed(ActionEvent e) {  
  
 String ipAndPort = textField.getText(); //获取ip地址和端口  
 int index = ipAndPort.indexOf(":");  
 String ip = ipAndPort.substring(0,index);  
 String port = ipAndPort.substring(index+1);  
 // 发送一个字节 -1 （是字节-1 不是int -1）  
 byte[] arr = new byte[]{-1};  
 try {  
 send(arr,ip,port);  
 } catch (IOException e1) {  
 e1.printStackTrace();  
 }  
 }  
 });  
 }  
  
  
 */\*\*  
 \* 展示记录  
 \*/* private void logFile(){  
 try {  
 br = new BufferedReader(new FileReader("log.txt"));  
 String line;  
 while ((line = br.readLine()) != null){  
 viewArea.append("\r\n");  
 viewArea.append(line);  
 }  
 } catch (FileNotFoundException e1) {  
 e1.printStackTrace();  
 } catch (IOException e1) {  
 e1.printStackTrace();  
 }  
 }  
  
 */\*\*  
 \* 震动  
 \*/* private void shake() throws IOException {  
 int x = this.getLocation().x;  
 int y = this.getLocation().y;  
  
 for(int i= 0;i<6;i++){  
 try {  
 this.setLocation(x+10,y+10);  
 Thread.*sleep*(20);  
 this.setLocation(x+10,y-10);  
 Thread.*sleep*(20);  
 this.setLocation(x-10,y-10);  
 Thread.*sleep*(20);  
 this.setLocation(x-10,y+10);  
 Thread.*sleep*(20);  
 this.setLocation(x,y);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
  
 }  
  
  
 */\*\*  
 \* 获取当前时间  
 \** ***@return*** *\*/* public String getCurrentTime(){  
 Date date = new Date();  
 SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");  
 String str = sdf.format(date);  
 return str;  
 }  
  
 public static void main(String[] args) throws IOException {  
 MyFrame mf = new MyFrame();  
 //设置可见  
 mf.setVisible(true);  
 }  
  
}

# 5 TCP协议 ServerSocket | Socket

TCP transfer control protocol 传输控制协议

三次握手：

== 客户端发起请求 ：创建socket

== 服务端相应请求 ：获取socket = serverSocket.accept( )

== 数据交换 : socke获取IO流：getInputStream() getOutputStream()

== 关闭Socket ： 关闭socket时，会自动关闭相关IO流

## 5.1客户端：

== 创建Socket连接服务端（指定ip地址，端口号），通过ip地址找对应服务器

== 调用Socket的getInputStream（）和getOutputStream（）方法获取和服务端

连接的IO流

== 输入流可以读取服务端输出流写出的数据

== 输出流可以写出数据到服务端输入流

## 5.2服务端：

== 创建ServerSocket 需要指定端口号

== 调用ServerSocket 的accept（）方法接收一个客户端请求，得到一个Socket

== 调用 Socket的getInputStream()和getOutputStream()方法获取和客户端相连

的IO流

== 输入流可以读取客户端输出流写出的数据

== 输出流可以写出数据到客户端的输入流

## 5.3 基本使用：

客户端：

package com.example.demo.网络编程.UDP.TCP.TCP.基本使用;  
  
import java.io.IOException;  
import java.io.InputStream;  
import java.io.OutputStream;  
import java.net.InetAddress;  
import java.net.Socket;  
  
public class Client {  
 public static void main(String[] args) throws IOException {  
  
 Socket socket = new Socket("127.0.0.1",10000);  
  
 //获取输出流  
 OutputStream os = socket.getOutputStream();  
 //获取输入流  
 InputStream in = socket.getInputStream();  
  
 //读取服务端信息  
 byte[] bytes = new byte[1024];  
 int len = in.read(bytes);  
 System.*out*.println(new String(bytes,0,len));  
 //向服务器发送  
 os.write("你好。。。".getBytes());  
 len = in.read(bytes);  
 System.*out*.println(new String(bytes,0,len));  
 os.write("再次你好。。。。".getBytes());  
 socket.close();  
  
  
 }  
}

服务端：

package com.example.demo.网络编程.UDP.TCP.TCP.基本使用;  
  
import java.io.IOException;  
import java.io.InputStream;  
import java.io.OutputStream;  
import java.net.ServerSocket;  
import java.net.Socket;  
  
public class Server {  
 public static void main(String[] args) throws IOException {  
 ServerSocket serverSocket = new ServerSocket(10000);  
  
 Socket socket = serverSocket.accept();  
  
 //获取输入流  
 InputStream in = socket.getInputStream();  
 //获取输出流  
 OutputStream os = socket.getOutputStream();  
  
 //向客户端写数据  
 os.write("欢迎光临。。。".getBytes());  
 byte[] bytes = new byte[1024];  
 int len = in.read(bytes);  
 System.*out*.println(new String(bytes,0,len));  
  
 os.write("再次欢迎光临。。。".getBytes());  
 len = in.read(bytes);  
 System.*out*.println(new String(bytes,0,len));  
  
 socket.close();  
 }  
  
}

## 练习

## 5.4 优化 – 多线程 – 字符串反转

服务端：

package com.example.demo.网络编程.UDP.TCP.TCP.优化;  
  
import com.example.demo.网络编程.UDP.TCP.UDP.优化.Receive;  
  
import java.io.\*;  
import java.net.ServerSocket;  
import java.net.Socket;  
  
public class Server {  
  
 static Socket *socket*;  
  
 public static void main(String[] args) throws IOException {  
 ServerSocket serverSocket = new ServerSocket(9999);  
  
 // 多线程 : 让服务器不停的接收 socket  
 while (true){  
 *socket* = serverSocket.accept();  
 //开启多线程，来处理数据交换  
 new Thread(){ //接收一次请求，就开启一条线程  
 public void run(){  
 //获取输入流  
 try {  
  
 BufferedReader br = new BufferedReader(new InputStreamReader(*socket*.getInputStream()));  
 PrintStream ps = new PrintStream(*socket*.getOutputStream());  
  
 // 欢迎语  
 ps.println("欢迎光临。。。");  
 //读取一行信息，readLine方法是以判断\r\n结束的  
 String data ;  
  
 while ((data = br.readLine()) != null){  
 System.*out*.println(data);  
 //反转数据  
 StringBuilder sb = new StringBuilder(data);  
 data = sb.reverse().toString();  
 //将数据再写回去  
 ps.println(data);  
 */\*\*此处读写不需要异步：因为需要先读再写，所以不用分成两个线程\*/* //socket.close();  
 }  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 }  
 }.start();  
 }  
 }  
  
}

客户端：

package com.example.demo.网络编程.UDP.TCP.TCP.优化;  
  
import java.io.\*;  
import java.net.Socket;  
import java.util.Scanner;  
  
public class Client {  
 static Socket *socket*;  
  
  
 public static void main(String[] args) throws IOException {  
 *socket* = new Socket("127.0.0.1",9999);  
 //获取输入流  
 try {  
 BufferedReader br = new BufferedReader(new InputStreamReader(*socket*.getInputStream()));  
 PrintStream ps = new PrintStream(*socket*.getOutputStream());  
  
 //如果要实现读和写的分离（异步），就要将他们置于不同的线程中  
 */\*\*写线程\*/* new Thread(){  
 public void run(){  
 while (true) {  
 Scanner scanner = new Scanner(System.*in*);  
 String str = scanner.nextLine();  
 ps.println(str);  
 }  
 }  
 }.start();  
  
 */\*\*读线程\*/* while (true){  
 //读取一行信息，readLine方法是以判断\r\n结束的  
 String data ;  
 while ((data = br.readLine()) != null){  
 System.*out*.println(data);  
 }  
 }  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 }  
  
}

## 5.5 用BufferedWriter在输出数据时会出问题？

1 BuffererWriter 的writer方法，不会在写完后添加回车符\r\n

而BufferedReader 的readLine()方法，恰恰是以读到回车符\r\n，表示结束的。

2 所以，如果直接只使用BufferedWriter的writer方法，会使得readLine()一直读不完，

而一直阻塞。

3 如果想要使用Bufferedwriter的writer方法，则要自己加上writer(“\r\n”),表示结尾。

## 5.6 上传文件

服务端：

package com.example.demo.网络编程.UDP.TCP.TCP.文件上传;  
  
import java.io.\*;  
import java.net.ServerSocket;  
import java.net.Socket;  
  
public class Server {  
  
 public static void main(String[] args) throws IOException {  
 //创建服务器  
 ServerSocket serverSocket = new ServerSocket(9999);  
 System.*out*.println("服务器启动");  
  
 //多线程服务器  
 while (true){  
 Socket socket = serverSocket.accept();  
 //每接收一次创建一个新线程  
 new Thread(){  
 public void run(){  
 try {  
 //获取输入输出流  
 InputStream is = socket.getInputStream();  
 BufferedReader br = new BufferedReader(new InputStreamReader(is));  
 PrintStream ps = new PrintStream(socket.getOutputStream());  
 //读取信息  
 String name = br.readLine();  
 System.*out*.println(name);  
 File file = new File(name);  
 //判断文件是否存在  
 if(file.exists()){  
 //发送提示文件已存在  
 ps.println("文件已存在");  
 System.*out*.println("文件已存在");  
 socket.close();  
 }else{  
 //发送提示文件不存在  
 ps.println("文件不存在");  
  
 System.*out*.println("文件不存在，开始接收");  
  
 */\*\*准备接收文件\*/* FileOutputStream fos = new FileOutputStream(name);  
 byte[] by = new byte[1024\*8];  
 int len;  
 while ((len = is.read(by)) != -1){  
 fos.write(by,0,len);  
 }  
 //关闭文件输出流  
 fos.close();  
 //关闭socket  
 socket.close();  
 System.*out*.println("接收完毕");  
 }  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 }  
 }.start();  
  
 }  
 }  
}

客户端：

package com.example.demo.网络编程.UDP.TCP.TCP.文件上传;  
  
import java.io.\*;  
import java.net.Socket;  
import java.util.Scanner;  
  
public class Client {  
  
 public static File getFile(){  
 //输入线程  
 Scanner scanner = new Scanner(System.*in*);  
 while (true) {  
 System.*out*.println("请输入文件路径：");  
 //获取文件路径  
 String path = scanner.nextLine();  
 //判断是否存在  
 File file = new File(path);  
  
 if(!file.exists() || !file.isFile()){  
 System.*out*.println("你输入的路径不存在，或者不是一个文件");  
 continue; //跳出重新循环  
 }  
 return file;  
 }  
  
 }  
  
  
 public static void main(String[] args) throws IOException {  
 Socket socket = new Socket("127.0.0.1",9999);  
 //获取输出流  
 OutputStream os = socket.getOutputStream();  
 PrintStream ps = new PrintStream(os);  
 //后去输入流  
 BufferedReader br = new BufferedReader(new InputStreamReader(socket.getInputStream()));  
 // while (true) {  
 */\*\* 1 传输文件名\*/* //获取传输的文件  
 File file = *getFile*();  
 //获取文件名  
 String name = file.getName();  
 //传输名称  
 ps.println(name);  
  
 */\*\* 2 接收响应\*/* String flag = null;  
 flag = br.readLine();  
  
 if ("文件已存在".equals(flag)) {  
 System.*out*.println("文件已存在,请不要重复上传");  
 //存在就关闭连接  
 socket.close();  
 //返回  
 return;  
 } else {  
 */\*\*文件上传\*/* System.*out*.println("上传开始");  
 //创建文件输入流 G:\棒.jpg  
 FileInputStream fis = new FileInputStream(file);  
 byte[] bytes = new byte[1024 \* 8];  
 int len;  
 while ((len = fis.read(bytes)) != -1) { //循环读入数组  
 os.write(bytes, 0, len); //循环写入输出流  
 }  
 //关闭文件输入流  
 fis.close();  
 //上传结束 关闭socket:如果不关闭，服务方会一直读取  
 socket.close();  
 System.*out*.println("上传完毕");  
 }  
 }  
 // }  
}