# 复习

## 静态方法与成员方法

我们曾经学习过两种方法的定义，一种是**静态方法**，格式为：

**public static 返回值类型 方法名称(参数类型 参数名称) {**

**// 方法体**

**}**

另一种为**成员方法**，也就是去掉了static关键字，格式为：

**public返回值类型 方法名称(参数类型 参数名称) {**

**// 方法体**

**}**

这两种方法的区别是什么？现阶段我们主要了解方法的调用方式不同：

* 成员方法的调用必须要创建一个对象才可以，格式为：  
  类名称 对象名 = new 类名称();  
  对象名.成员方法名(参数);
* 而静态方法的调用不需要借助对象，可以直接通过类名称来调用，格式为：  
  类名称.静态方法名(参数);

# 方法引用

## 冗余的Lambda

在某些场景之下，Lambda表达式要做的事情，其实在另外一个地方已经写过了，那么此时如果通过Lambda表达式重复编写相同的代码，就是浪费。例如已经有了一个厨师类：

public class Cook {

// 这是一个静态方法，可以通过类名称进行调用

public static void makeFood(String food) {

System.*out*.println("将" + food + "做成可口的食物。");

}

}

还有一个函数式接口：

// 保姆接口

@FunctionalInterface

**public** **interface** Sitter {

// 保姆的工作，就是生米煮成熟饭

**void** work(String food);

}

但是在使用函数式接口的时候，Lambda的内容其实和厨师Cook类的方法体重复了：

**public** **class** Demo03Lambda {

**public** **static** **void** main(String[] args) {

*hireSitter*(food -> System.***out***.println("将"+food+"做成可口的食物。"));

}

// 雇佣一个保姆，并且让他去做饭

**public** **static** **void** hireSitter(Sitter sitter) {

sitter.work("白菜");

}

}

## 体验方法引用

此时Lambda所做的事情，其实在Cook类的方法当中已经做过了，那么就可以使用方法引用。例如：

**public** **class** Demo03Lambda {

**public** **static** **void** main(String[] args) {

*hireSitter*( Cook::*makeFood* );

}

// 雇佣一个保姆，并且让他去做饭

**public** **static** **void** hireSitter(Sitter sitter) {

sitter.work("白菜");

}

}

## 类名称引用静态方法

通过类名称引用静态方法的格式为：

**类名称::静态方法名**

这将可以作为Lambda表达式的替代者，更加简单。

## 对象名引用成员方法

另一种通过对象名称来引用成员方法的格式为：

**对象名::成员方法名**

例如Cook类当中有一个成员方法（并非静态方法）：

public class Cook {

// 这是一个成员方法，必须要有对象才能调用

public void makeFood(String food) {

System.*out*.println("将" + food + "做成可口的饭菜！");

}

}

那么引用其中的成员方法时，必须要有一个对象才行。例如：

**public** **class** Demo02MethodRef {

**public** **static** **void** main(String[] args) {

Cook cook = **new** Cook(); // 创建了一个厨师对象

// 引用了cook对象当中的成员方法makeFood

*method*( cook::makeFood );

}

**public** **static** **void** method(Sitter sitter) {

sitter.work("土豆");

}

}

# Stream流

## 冗余的集合for遍历

在使用集合进行某些操作时，一次又一次的for循环遍历虽然简单，但是步骤冗余：

**public** **class** Demo01ArrayList {

**public** **static** **void** main(String[] args) {

// 首先创建一个集合，然后放入指定格式的字符串

ArrayList<String> recordList = **new** ArrayList<>();

recordList.add("赵丽颖,98");

recordList.add("鹿晗,95");

recordList.add("宋小宝,87");

// 应该拆分一下每个字符串，只要逗号后面的

ArrayList<String> scoreList = **new** ArrayList<>(); // 保存的多个字符串是："98"、"95"、"87"

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

String record = recordList.get(i); // 当前字符串："赵丽颖,98"

String[] array = record.split(",");

String score = array[1]; // "98"

scoreList.add(score);

}

// 将字符串"98"转换成为int数字98

ArrayList<Integer> numList = **new** ArrayList<>(); // 保存的多个数字是：98、95、87

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

String score = scoreList.get(i); // "98"

**int** num = Integer.*parseInt*(score);

numList.add(num);

}

// 过滤一下，筛选，只要大于90的，小于等于90的不要

ArrayList<Integer> resultList = **new** ArrayList<>(); // 最终结果集合

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

**int** num = numList.get(i); // 98

**if** (num > 90) {

resultList.add(num);

}

}

// 最后遍历最终结果集合，打印输出

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

**int** result = resultList.get(i);

System.***out***.println(result);

}

}

}

## 体验流式写法

如果使用Java 8当中新加入的Stream API，代码篇幅将会减少很多：

**public** **class** Demo02ArrayListStream {

**public** **static** **void** main(String[] args) {

// 首先创建一个集合，然后放入指定格式的字符串

ArrayList<String> recordList = **new** ArrayList<>();

recordList.add("赵丽颖,98");

recordList.add("鹿晗,95");

recordList.add("宋小宝,87");

// Stream API更优写法！

recordList.stream().map(s -> s.split(",")[1])

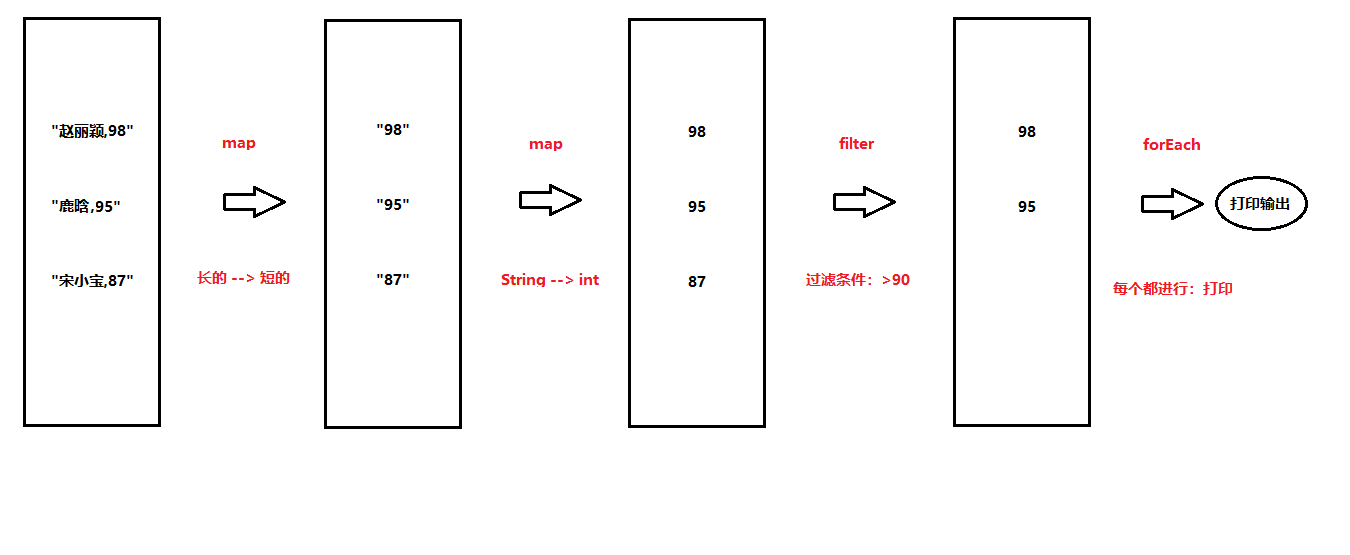
.map(Integer::*parseInt*).filter(n -> n > 90).forEach(System.***out***::println);

}

}

## 流式思想概述

Stream API的思想就是流水线加工，上述代码的流程如下图所示：



## 获取流

一个流，就是一个**java.util.stream.Stream<T>**接口的对象。可以根据集合获取流，格式为：

**集合名称.stream()**

也可以通过一个数组（元素必须为引用类型）来获取一个流，格式为：

**Stream.of(数组名称)**

例如：

**public** **class** Demo03GetStream {

**public** **static** **void** main(String[] args) {

// 1. 根据集合获取流

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

list.add("迪丽热巴");

list.add("古力娜扎");

list.add("玛尔扎哈");

Stream<String> streamA = list.stream();

// 2. 根据数组获取流，数组当中的元素必须是引用类型才行

String[] arrayStr = { "Hello", "World", "Java" };

Stream<String> streamB = Stream.*of*(arrayStr);

Integer[] arrayInteger = { 10, 20, 30 };

Stream<Integer> streamC = Stream.*of*(arrayInteger);

}

}

## 映射map方法

获取流之后，可以使用映射方法：**map(用于转换的Lambda表达式)**。映射：就是将一个对象转换成为另一个对象，把老对象映射到新对象上。

**public** **class** Demo04StreamMap {

**public** **static** **void** main(String[] args) {

// 这个集合当中存放的是字符串类型

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

list.add("100");

list.add("200");

list.add("300");

Stream<Integer> streamA = list.stream().map((String str) -> {

**int** num = Integer.*parseInt*(str);

**return** num;

});

Stream<Integer> streamB = list.stream().map(str -> {

**int** num = Integer.*parseInt*(str);

**return** num;

});

Stream<Integer> streamC = list.stream().map(str -> {

**return** Integer.*parseInt*(str);

});

Stream<Integer> streamD = list.stream().map(Integer::*parseInt*);

System.***out***.println("========================");

ArrayList<String> list2 = **new** ArrayList<>();

list2.add("赵丽颖,98");

list2.add("鹿晗,95");

list2.add("宋小宝,87");

Stream<String> stream1 = list2.stream().map((String str) -> {

String[] array = str.split(",");

String result = array[1];

**return** result;

});

Stream<String> stream2 = list2.stream().map(s -> {

String[] array = s.split(",");

String result = array[1];

**return** result;

});

Stream<String> stream3 = list2.stream().map(s -> {

String[] array = s.split(",");

**return** array[1];

});

Stream<String> stream4 = list2.stream().map(s -> {

**return** s.split(",")[1];

});

Stream<String> stream5 = list2.stream().map(s -> s.split(",")[1]);

}

}

## 过滤filter方法

如果希望对流当中的元素进行过滤，可以使用过滤方法：

**filter(能产生boolean结果的Lambda)**：如果参数Lambda产生了true，则要元素；如果产生了false，则不要元素。

**public** **class** Demo05StreamFilter {

**public** **static** **void** main(String[] args) {

ArrayList<Integer> list1 = **new** ArrayList<>();

list1.add(90);

list1.add(85);

list1.add(60);

Stream<Integer> stream1 = list1.stream().filter((Integer num) -> {

**boolean** b = num > 80;

**return** b;

});

Stream<Integer> stream2 = list1.stream().filter(num -> {

**boolean** b = num > 80;

**return** b;

});

Stream<Integer> stream3 = list1.stream().filter(num -> {

**return** num > 80;

});

Stream<Integer> stream4 = list1.stream().filter(n -> n > 80);

System.***out***.println("==================");

ArrayList<String> list2 = **new** ArrayList<>();

list2.add("赵丽颖");

list2.add("赵丽颖");

list2.add("宋小宝");

Stream<String> streamA = list2.stream().filter((String str) -> {

// boolean b = "赵丽颖".equals(str);

**boolean** b = str.equals("赵丽颖");

**return** b;

});

Stream<String> streamB = list2.stream().filter(s -> {

**boolean** b = s.equals("赵丽颖");

**return** b;

});

Stream<String> streamC = list2.stream().filter(s -> {

**return** s.equals("赵丽颖");

});

Stream<String> streamD = list2.stream().filter(s -> s.equals("赵丽颖"));

}

}

## 遍历forEach方法

如果希望在流当中进行元素的遍历操作，可以使用forEach方法：

**forEach(Lambda表达式)**：意思是，对流当中的每一个元素都要进行操作。参数Lambda表达式必须是一个能够消费一个参数，而且不产生数据结果的Lambda。

**public** **class** Demo06StreamForEach {

**public** **static** **void** main(String[] args) {

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

list.add("迪丽热巴");

list.add("古力娜扎");

list.add("玛尔扎哈");

Stream<String> streamA = list.stream();

streamA.forEach((String str) -> {

System.***out***.println(str);

});

System.***out***.println("==============");

list.stream().forEach((String str) -> {

System.***out***.println(str);

});

System.***out***.println("==============");

list.stream().forEach(str -> {

System.***out***.println(str);

});

System.***out***.println("==============");

list.stream().forEach(System.***out***::println);

}

}

## 并发流

流当中的元素如果特别多，那么只有一个人在逐一、挨个儿处理，肯定比较慢，费劲。如果对流当中的元素，使用多个人同时处理，这就是“并发”。

获取并发流有两种方式：

1. 直接获取并发流：**.parallelStream()**
2. 已经获取了普通流，然后升级成为并发流：.stream()**.parallel()**

**public** **class** Demo07StreamParallel {

**public** **static** **void** main(String[] args) {

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

**for** (**int** i = 1; i <= 100; i++) {

list.add("Hello-" + i);

}

// 这是只有一个人在做打印输出的操作

// list.stream().forEach(System.out::println);

// 获取一个并发流

list.parallelStream().forEach(System.***out***::println);

list.stream().parallel().forEach(System.***out***::println);

}

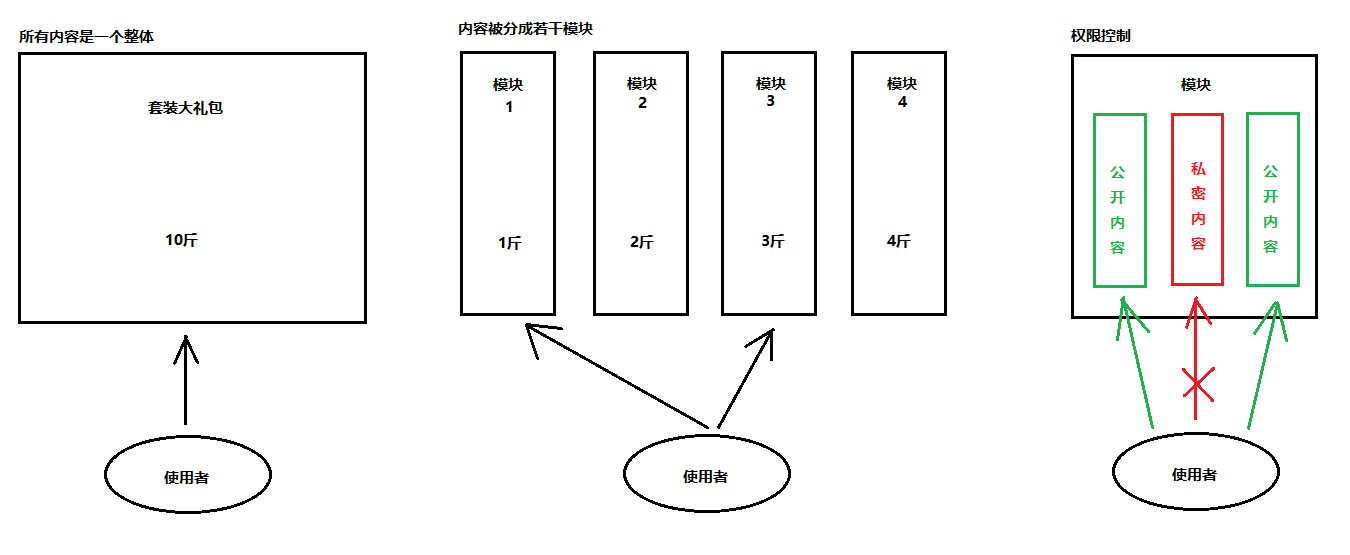
}

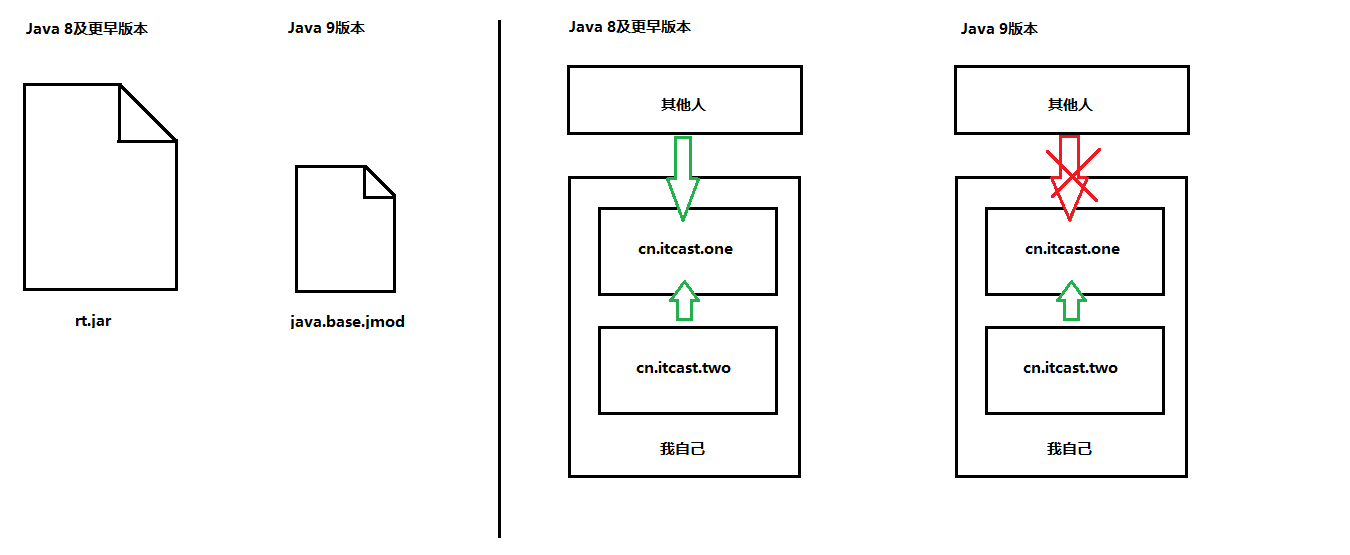
# 模块化

## 思想概述

Java 9中引入的模块化技术可以达到两种效果：

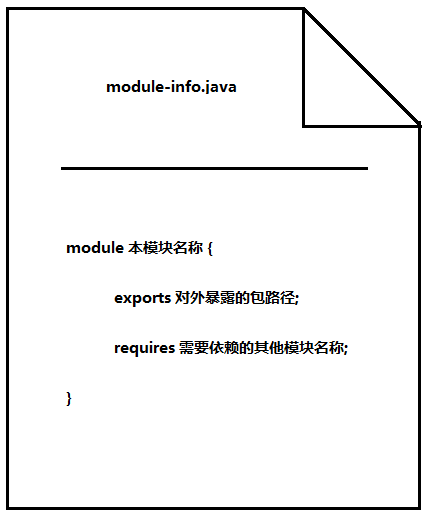
1. 所有资源都在一起，体积臃肿，通常并不需要其中所有的内容。有了模块化，拆分成为若干个小模块，可以只选择需要的。
2. 可以精确控制package包级别的访问权限，控制模块之外的访问情况。



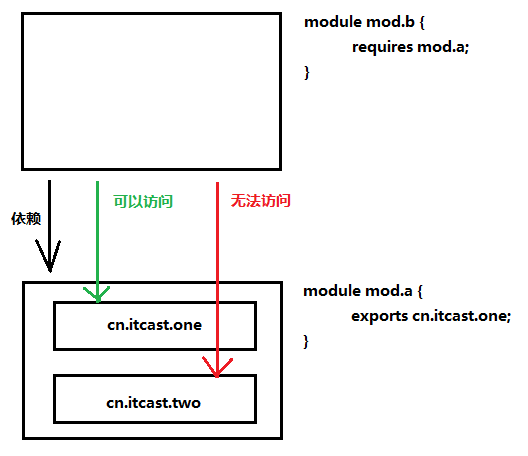


## 认识module-info.java文件

在Java的一个模块中，需要使用**module-info.java**文件描述模块信息。



模块之前的依赖关系以及权限访问控制情况为：



## 模块的基本使用

可以通过下面的操作将Eclipse的Java Project改造成一个Java 9的模块：

1. 点击项目名称，右键选择Configure，点击Create module-info.java选项。
2. 在生成的module-info.java文件中编写导出exports和依赖requires等信息。
3. 点击项目名称，右键选择Build Path，点选最后一项Configure Build Path。在Projects标签中选中Modulepath，然后右侧Add所需要依赖的模块。

# 知识总结

1. 对比一下静态方法与成员方法的区别：
   1. 静态方法与对象无关，可以直接通过类名称调用：  
      类名称.静态方法名(参数值);
   2. 成员方法一定要有对象才能使用：  
      对象名.成员方法名(参数值);
2. 方法引用：如果Lambda表达式想做的事情，在某一个地方某一个方法已经做过了，那么就可以使用方法引用来替代Lambda。
3. 方法引用的方式：
   1. 通过类名称引用静态方法：类名称::静态方法名
   2. 通过对象名引用成员方法：对象名::成员方法名
4. Stream流式思想：简化了普通的集合操作，类似于流水线。Stream流在Java当中就是一个java.util.stream.Stream<T>接口的对象。
   1. 如何获取流：
      1. 通过集合获取：集合名称.stream()
      2. 通过数组获取：Stream.of(数组名称) 注意，数组当中必须是引用类型元素
   2. 映射的方法：map，参数是一个负责转换的Lambda表达式
   3. 过滤的方法：filter，参数是一个可以产生boolean结果的Lambda表达式
   4. 遍历的方法：forEach，参数也是一个Lambda，Lambda对应的抽象方法应该只有一个参数并且不产生返回值。
   5. 如何获取并发流
      1. 直接一次性获取并发流：.parallelStream()
      2. 首先获取普通流，然后继续变成并发流：.stream().parallel()
5. 模块化思想可以为我们带来两点好处：
   1. 整体一整个，文件体积太大。如果只是用到了其中的一部分内容，就会浪费。
   2. 精确控制package包级别的访问控制。只有导出的包，模块之外才可以访问，否则只能模块内部自己访问。
   3. 模块的描述信息module-info.java文件的基本内容：  
      module 本模块的名称 {  
       exports 导出的包名称;  
       requires 需要依赖的其他模块名称;  
      }