Java8实战

参考书籍《Java8实战——中国工信出版集团》

参考代码链接：https://github.com/java8/Java8InAction

背景

希望利用多核计算机或者计算机集群有效处理大数据，需要使用并行处理，Java对此支持不好！，借助其他编程语言思想，比如利用Sql可以利用多核CPU来处理大数据量，所以就萌发了“我为什么不能用SQL的风格来写程序”。所以经过一系列研究，Java8新特性也就出现了。

基础知识

基本概念

- 流

  流是一系列数据项，一次只生产一项；例如（输入流，输出流，常见的io流）
  举例说明：
  linux命令
  cat file1 file2 | tr "[A-Z]" "[a-z]" |sort| tail -3
  生活中的例子：
  比如汽车生产线，工艺加工流程，可以生产线看作一个序列，但是不同的加工站一般是并行的。
  代码层面：
  Stream<T>,就是一系列T类型的数据项

- 行为参数化

  把方法（你的代码）作为参数传递给另一个方法的能力
  即一个方法接受多个不同的行为作为参数，并在内部使用它们，完成不同行为的能力
  举例：
  谓词Predicate，类似于一个函数，该函数接受一个值并返回true或false

- 方法引用

  通过例子引出方法引用：
  比如：需要查找当前目录下所有的隐藏文件，代码层面（略——直接演示即可）
  1、如何创建方法引用：例如File::isHidden,可以作为参数进行传递
  方法引用可以看作仅仅调用特定方法的Lambda的一种快捷写法。

- Lambda——匿名函数

  主题思想：将函数作为值
  (int x) -> x+1，表示“调用时给定参数x，就返回x+1的值”
  为什么需要Lambda表达式呢？
  不需要再次定义只用一次两次的方法，作为创建方法引用的条件，解决代码啰嗦的问题

- 其他

Optional<T>:它是一个容器对象，可以包含，也可以不包含一个值。主要是为了避免出现空指针行为
参数化
isParent():将在Optional包含值的时候返回true，否则返回false
ifParent(Consumer<T> bolck):会在值存在的时候执行给定的代码块。
T get(); 会在值存在是返回值
T orElse(T other)：会在值存在时返回值，否则返回一个默认值；

行为参数化

作用：行为参数化可以让代码更好地适应不断变化的需求，减轻未来的工作量。比如JavaAPI包含很多可以用不同行为进行
参数化的方法，包括排序，线程，GUI处理

实例1（筛选苹果）

```
public class FilteringApples{

    public static void main(String ... args){

        List<Apple> inventory = Arrays.asList(new Apple(80,"green"),
                                              new Apple(155, "green"),
                                              new Apple(120, "red"));
        List<Apple> collect = inventory.stream().filter((Apple a) ->
a.getWeight() > 150).collect(Collectors.toList());

        // [Apple{color='green', weight=80}, Apple{color='green', weight=155}]
        List<Apple> greenApples = filterApples(inventory,
FilteringApples::isGreenApple);
        System.out.println(greenApples);

        // [Apple{color='green', weight=155}]
        List<Apple> heavyApples = filterApples(inventory,
FilteringApples::isHeavyApple);
        System.out.println(heavyApples);

        // [Apple{color='green', weight=80}, Apple{color='green', weight=155}]
        List<Apple> greenApples2 = filterApples(inventory, (Apple a) ->
"green".equals(a.getColor()));
        System.out.println(greenApples2);

        // [Apple{color='green', weight=155}]
        List<Apple> heavyApples2 = filterApples(inventory, (Apple a) ->
a.getWeight() > 150);
        System.out.println(heavyApples2);

        // []
        List<Apple> weirdApples = filterApples(inventory, (Apple a) ->
a.getWeight() < 80 ||

"brown".equals(a.getColor()));
        System.out.println(weirdApples);
    }

    public static List<Apple> filterGreenApples(List<Apple> inventory){
        List<Apple> result = new ArrayList<>();
        for (Apple apple: inventory){
```

```
            if ("green".equals(apple.getColor())) {
                result.add(apple);
            }
        }
        return result;
    }

    public static List<Apple> filterHeavyApples(List<Apple> inventory){
        List<Apple> result = new ArrayList<>();
        for (Apple apple: inventory){
            if (apple.getWeight() > 150) {
                result.add(apple);
            }
        }
        return result;
    }

    public static boolean isGreenApple(Apple apple) {
        return "green".equals(apple.getColor());
    }

    public static boolean isHeavyApple(Apple apple) {
        return apple.getWeight() > 150;
    }

    public static List<Apple> filterApples(List<Apple> inventory,
Predicate<Apple> p){
        List<Apple> result = new ArrayList<>();
        for(Apple apple : inventory){
            if(p.test(apple)){
                result.add(apple);
            }
        }
        return result;
    }

    public static class Apple {
        private int weight = 0;
        private String color = "";

        public Apple(int weight, String color){
            this.weight = weight;
            this.color = color;
        }

        public Integer getWeight() {
            return weight;
        }

        public void setWeight(Integer weight) {
            this.weight = weight;
        }

        public String getColor() {
            return color;
        }

        public void setColor(String color) {
```

```
            this.color = color;
        }

        public String toString() {
            return "Apple{" +
                   "color='" + color + '\'' +
                   ", weight=" + weight +
                   '}';
        }
    }

}
```

实例2（用Comparator排序）

```
   class AppleComparator implements Comparator<Apple> {
        public int compare(Apple a1, Apple a2){
            return a1.getWeight().compareTo(a2.getWeight());
        }
    }
   inventory.sort(new AppleComparator());
   改为lambda
  lambda: inventory.sort(Applea a1,Apple a2) ->
a1.getWeight().compareTo(a2.getWeight)
```

实例3（用Runnable执行代码块）

```
Thread t = new Thread(new Runnable(){
    public void run(){
     System.out.println("hello world");
    }
});
lambda:Thread t = new Thread(()-> System.out.println("hello world");)
```

实例4（GUI事件处理）

```
Button button = new Button("Send");
button.setOnAction(new EventHandler<ActionEvent>(){
    pulbic void handle(ActionEvent event){
        lable.setText("Sent!");
    }
})
lambda: button.setOnAction((Actrion event) -> lable.setText("Sent!"))
```

Lambda表达式

作用：匿名类来表示不同的行为并不令人满意，代码十分啰嗦，利用Lambda表达式使代码更简洁，更灵活
使用条件：在函数式接口中使用

Lambda表达式结构

```
inventory.sort((Applea a1,Apple a2) -> a1.getWeight().compareTo(a2.getWeight))
参数列表：(Applea a1,Apple a2)
箭头：->
```

Lambda主体：a1.getWeight().compareTo(a2.getWeight).表达式就是Lambda的返回值，Lambda没有return语句，因为已经隐含了return。
有效的Lambda表达式有哪些?
1、（String s）-> s.length();
2、（Applea a1）-> a1.getWeight() >150
3、（int x,int y）->{
    System.out.println(x);
    System.out.println(y);
}
4、() -> 42：lambda表达式没有参数列表，返回的是一个int型
5、(Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight())：lamda表达式具有两个Apple类型的参数，返回一个int;
测试一下：
6、() ->{}:lambda表达式是一个没有参数，无返回。它类似于public void run();
7、() ->"M":
8、() -> {return “M”;}
9、(Integer i) -> return “M”+i;(✕)
10、(String s) -> {"a";}(✕)

习题练习：
布尔表达式
创建对象
消费对象
从对象中选择/提取
组合两个值
比较两个对象
思考：在哪里使用Lambda表达式

函数式接口

函数式接口就是只定义了一个抽象方法的接口，但是可以拥有很多个默认方法。
例如：
Comparator
Runnable
EventListener
Callable
注解：@FunctionalInterface表示该接口会被设计成一个函数式接口，但不是必须的；
测试一下：
下面哪些是函数式接口：
```
public interface A{
   int add(int x);
}
public interface B extends A{
   int add(double y);
}
public interface C{

}
```
当你使用接口中方法是，需要将对象进行实例化，而lambda表达式是可以为函数式接口的抽象方法提供实现。具体来说，就是通过lambda表达式创建函数式接口的一个具体实现的实例；（匿名内部类也可以做到，但是比较笨拙）
举例说明：
执行任务:
```
public static void process(Runnable r){
    r.run();
}
```

创建任务任务:
```
Runnable a = new Runnable(){
    public void run(){
     System.out.println("hello world");
    }
}
```
lambda表达式：`Runnable b = () ->  System.out.println("hello world");`
调用方法：
```
process(a);
process(b);
process(() -> System.out.println("hello"));
```

函数描述符

函数式接口的抽象方法叫作函数描述符；
`public void run()`:它的描述符，就是一个什么也不接收什么也不返回的函数；用符号来描述 （）->
`public int compare(Apple a1, Apple a2)`:它的描述符，就是接收两个Apple类型的参数，并返回int型，如何来描述（Apple a1, Apple a2）-> 1;(还可以多种形式表达)
测试一下：（下面这些方法中传递lambda表达式是否正确）
```
1、public void excute（Runnable r）{
     r.run
}
excute(() ->{});
2、public Callable<String> fetch(){
    return () -> "A";
}
3、Predicate<Apple> p = (Apple a) -> a.getWeight();✕
```

使用函数式接口

`Predicate<T>`:定义了一个名叫test抽象方法，接收T类型的对象，并返回boolean
`Consumer<T>`:定义了一个名叫accept的抽象方法，，接收T类型的对象,无返回
`Function<T,R>`:定义了一个apply的方法，它接受一个泛型T的对象，并返回一个泛型R的对象。
`Supplier<T>`:定义了一个抽象方法叫作get，它接受一个泛型T的对象，返回T
`Callable<T>`:定义了一个call方法，它接受一个泛型T的对象，返回T
举例：
```
public interface Function<T,R> {
    R apply(T t);
}
public static <T,R> List<R> map(List<T> list,Function<T,R> f){
    List<R> result = new ArrayList<>();
    for(T s :list){
        result.add(f.apply(s));
    }
    return result;
}
List<Interger> l = map(Arrays.asList("a","ab","abc")),(String s) -> s.length());
```

类型检查

lambda表达式可以为函数式接口生成一个实例，但是表达式本身并不包含它在实现哪个函数式接口的信息。
所以需要了解Lambda的实际类型是什么；而实际类型是怎么来的呢，可以通过Lambda的上下文推断出来；（实际类型-目标类型）
例如：
```
Object o = () -> {System.out.println("");}✕
```

使用局部变量

使用局部变量必须显示声明final。（为什么需要添加这个限制）
思考：局部变量和实例变量的本质区别？

构造函数引用

对于一个现有的构造函数，你可以利用它的名称和关键字new来创建它的一个引用；语法ClassName::new
使用默认构造函数创建对象

```
Supplier<Apple> a = Apple::new;
Apple b = a.get();
```

复合Lambda表达式使用（略）

函数式处理数据

流简介

简短定义：从支持数据处理操作的源生成的元素序列
解释：
1、元素序列：与集合类似，可以访问特定元素类型的一组有序值。但是流的目的在于计算，而不是数据
2、源：流会使用一个提供数据的源，如集合、数组、输入/输出资源
3、数据处理操作：与数据库的操作一致，具有常用函数式编程语言操作，例如：filter、map、reduce、find、match、sort
4、流水线：很多流操作自身也会返回一个流，然后将流链接起来，形成一个大的流水线
5、内部迭代：将流转换成其他形式，如列表
举例说明：

```
    public static List<String> getLowCaloricDishesNamesInJava8(List<Dish>
dishes) {
        return dishes.stream()
                .filter(d -> d.getCalories() < 400)
                .sorted(comparing(Dish::getCalories))
                .map(Dish::getName)
                .limit(2)
                .collect(toList());
    }
```

filter（筛选）：接受Lambda，从流中排出某些元素。d -> d.getCalories() < 400
map（提取）:接收Lamda，将元素转换成其他形式或提取信息，这里提取就是dish的名称
sorted（排序）：接手Lambda，按照卡路里排序
limit（截断）：截断流，取前2个
collect（转换）：将流转换成列表

流与集合

流与集合的区别：
1、概念上讲：集合与流之间的差异就在于什么时候计算：集合中的每个元素都预先计算出来才能添加到集合中，而流则是按需计算，只有在消费者要求时候才开始计算；
2、遍历数据的方式：流只能遍历依次，遍历完之后，这个流已经被消费掉了。如果需要再次遍历，需要重新创建；
举例说明：

```
List<String> names = Arrays.asList("Java8", "Lambdas", "In", "Action");
Stream<String> s = names.stream();
s.forEach(System.out::println);
//s.forEach(System.out::println);
```

流操作

流操作分为：
中间操作：可以连接起来的流操作
filter、map、limit、sorted、distinct
终端操作：关闭流的操作
forEach count collect
为什么需要这么定义和区分呢？
主要解决的问题是，除非流水线上触发一个终端操作，否则中间操作不会执行任何处理，利用了流的延迟性质，循环合并
举例说明：

```
public static List<String> getLowCaloricDishesNamesInJava8(List<Dish> dishes) {
        return dishes.stream()
                .filter(dish -> {
                            System.out.println("filtering"+dish.getName());
                            return dish.getCalories() < 400;
                        }
                        )
                .sorted(comparing(Dish::getCalories))
                .map(dish -> {
                    System.out.println("mapping"+dish.getName());
                    return dish.getName();
                })
                .limit(3)
                .collect(toList());
}
```

总结：
流程的使用一般包括三件事：
一个数据源来执行一个查询
一个中间操作链，形成一条流水线
一个终端操作，执行流水行，并能生产结果

使用流

流具有哪些操作？
筛选，切片，映射，查找，匹配，归约；

筛选和切片

从源码上看，filter接收一个谓词（一个返回boolean的函数）作为参数

```
Stream<T> filter(Predicate<? super T> predicate);
```

筛选还支持distinct的方法，去重的操作
举例说明：筛选出列表中所有的偶数，并确保没有重复

```
List<Integer> numbers = Arrays.asList(1, 2, 1, 3, 3, 2, 4);
numbers.stream()
        .filter(i -> i % 2 == 0)
        .distinct()
        .forEach(System.out::println);
```

切片：
limit：注意源可以为一个set，limit的结果不会以任何顺序排列
skip：跳过，例如跳过卡路里超过300的菜

```
        List<Dish> dishesSkip2 =
            menu.stream()
                .filter(d -> d.getCalories() > 300)
                .skip(2)
```

```
                .collect(toList());
```
测试：如何利用流来筛选前两个荤菜？（描述即可，不用编写）

映射

什么是映射？简单理解，就是从某些对象中选择信息，可以通过map和flatMap
举例说明：
场景：
1、给定一个单词列表，显示每个单词有几个字母
```
List<String> words = Arrays.asList("Hello", "World");
List<Integer> wordLengths = words.stream()
                                 .map(String::length)
                                 .collect(toList());
System.out.println(wordLengths);
```
2、找出每条鱼的名称有多长？
```
List<String> dishNames = menu.stream()
                             .map(Dish::getName)
                             .map(String::length)
                             .collect(toList());
System.out.println(dishNames);
```
流的扁平化：
举例说明：如何返回一张列表，列出里面各不相同的字符呢？
```
List<String> words = Arrays.asList("Hello", "World");
        List<Integer> wordLengths = words.stream()
                                         .map(s->s.split(""))
                                         .distinct()
                                         .collect(toList());
        System.out.println(wordLengths);✕
List<String> words = Arrays.asList("Hello", "World");
        List<Integer> wordLengths = words.stream()
                                         .map(s->s.split(""))
                                         .map(Arrays::stream)    流的列表
                                         .distinct()
                                         .collect(toList());
        System.out.println(wordLengths);✕
List<String> words = Arrays.asList("Hello", "World");
        List<Integer> wordLengths = words.stream()
                                         .map(s->s.split(""))
                                         .flatMap(Arrays::stream) 将流的列表合并，
扁平化一个流
                                         .distinct()
                                         .collect(toList());
        System.out.println(wordLengths);✓
```
测试：
1、例如给定一个数字列表，如何返回一个由每个数的平方构成的列表？
如：【1，2，3，4，5】变成了【1，4，9，16，25】
2、给定两个数字列表，如何返回所有的数对呢？
如【1，2，3】【3，4】【（1，3）】
```
List<Integer> numbers1 = Arrays.asList(1, 2, 3);
List<Integer> numbers2 = Arrays.asList(3, 4);
List<Integer[]> p = numbers1.stream().flatMap(i -> numbers2.stream().map(j ->
new Integer[]{i, j})).collect(toList());
        p.forEach(s -> System.out.println("(" + s[0] + "," + s[1] + ")"));
```
3、扩展总能被3整除的数对？（略，提示使用filter (i+j)%3==0）

查找和匹配

查看数据集中的某些元素是否匹配一个给定属性
API：allMatch、anyMatch、noneMatch、findFirst、findAny
anyMatch:流中是否含有一个元素能匹配给定的谓词
allMatch：查看流中元素是否都能匹配给定的谓词
noneMatch:确保流中没有任何元素与给定谓词相匹配
findAny：将返回当前流中任意元素，可以与其他流配合使用；
findFirst:查找第一个元素

归约

将流归约成一个值，术语叫折叠
1、元素求和
2、最大值和最小值
举例说明：

```
 List<Integer> numbers = Arrays.asList(3,4,5,1,2);
        int sum = numbers.stream().reduce(0, (a, b) -> a + b);
        System.out.println(sum);

        int sum2 = numbers.stream().reduce(0, Integer::sum);
        System.out.println(sum2);

        int max = numbers.stream().reduce(0, (a, b) -> Integer.max(a, b));
        System.out.println(max);

        Optional<Integer> min = numbers.stream().reduce(Integer::min);
        min.ifPresent(System.out::println);

        int calories = menu.stream()
                           .map(Dish::getCalories)
                           .reduce(0, Integer::sum);
        System.out.println("Number of calories:" + calories);
```

构建流

由值创建流
Stream.of，通过显示值创建一个流,它可以接受任意数据量的参数

```
Stream<String> stream = Stream.of("Java 8", "Lambdas", "In", "Action");
        stream.map(String::toUpperCase).forEach(System.out::println);
```

Strem.empty()得到一个空流
由数组创建流

```
int[] numbers = {2, 3, 5, 7, 11, 13};
System.out.println(Arrays.stream(numbers).sum());
```

由文件生成流

```
// Files.lines:它会返回一个由指定文件中的各行构成的字符串流
```

场景：数据流中有多少各不相同的单词

```
long uniqueWords = Files.lines(Paths.get("lambdasinaction/chap5/data.txt"),
Charset.defaultCharset())
                .flatMap(line -> Arrays.stream(line.split(" ")))
                .distinct()
                .count();
System.out.println("There are " + uniqueWords + " unique words in data.txt");
```

由函数生成流，创建无限流
举例说明

```
 Stream.iterate(new int[]{0, 1}, t -> new int[]{t[1],t[0] + t[1]})
              .limit(10)
```

```
              .forEach(t -> System.out.println("(" + t[0] + ", " + t[1] + ")"));

        Stream.iterate(new int[]{0, 1}, t -> new int[]{t[1],t[0] + t[1]})
              .limit(10)
              . map(t -> t[0])
              .forEach(System.out::println);

        // random stream of doubles with Stream.generate
        Stream.generate(Math::random)
              .limit(10)
              .forEach(System.out::println);

        // stream of 1s with Stream.generate
        IntStream.generate(() -> 1)
                 .limit(5)
                 .forEach(System.out::println);

        IntStream.generate(new IntSupplier(){
            public int getAsInt(){
                return 2;
            }
        }).limit(5)
          .forEach(System.out::println);
```

用流收集数据（略）

并行数据处理与性能（略）

调试（略）

其他（略）

总结（略）

*择其善者而从之，*其不善者而改之

——《论语》