jdk1.8新特性知识点：

- **Lambda表达式**
- **函数式接口**
- **方法引用和构造器调用**
- **Stream API**
- **接口中的默认方法和静态方法**
- **新时间日期API**

```
在jdk1.8中对hashMap等map集合的数据结构优化。hashMap数据结构的优化
原来的hashMap采用的数据结构是哈希表（数组+链表），hashMap默认大小是16，一个0-15索引的数组，如何往里面存储元素，首先调用元素的hashcode
方法，计算出哈希码值，经过哈希算法算成数组的索引值，如果对应的索引处没有元素，直接存放，如果有对象在，那么比较它们的equals方法比较内容
如果内容一样，后一个value会将前一个value的值覆盖，如果不一样，在1.7的时候，后加的放在前面，形成一个链表，形成了碰撞，在某些情况下如果链表
无限下去，那么效率极低，碰撞是避免不了的
加载因子：0.75，数组扩容，达到总容量的75%，就进行扩容，但是无法避免碰撞的情况发生
在1.8之后，在数组+链表+红黑树来实现hashmap，当碰撞的元素个数大于8时 & 总容量大于64，会有红黑树的引入
除了添加之后，效率都比链表高，1.8之后链表新进元素加到末尾
ConcurrentHashMap（锁分段机制），concurrentLevel,jdk1.8采用CAS算法(无锁算法，不再使用锁分段)，数组+链表中也引入了红黑树的使用
```

一、Lambda表达式

1.1 函数式编程

函数编程非常关键的几个特性如下：

（1）闭包与高阶函数

函数编程支持函数作为第一类对象，有时称为 闭包或者 仿函数（functor）对象。实质上，闭包是起函数的作用并可以像对象一样操作的对象。

与此类似，FP 语言支持 高阶函数。高阶函数可以用另一个函数（间接地，用一个表达式）作为其输入参数，在某些情况下，它甚至返回一个函数作为其输出参数。这两种结构结合在一起使得可以用优雅的方式进行模块化编程，这是使用 FP 的最大好处。

（2）惰性计算

在惰性计算中，表达式不是在绑定到变量时立即计算，而是在求值程序需要产生表达式的值时进行计算。延迟的计算使您可以编写可能潜在地生成无穷输出的函数。因为不会计算多于程序的其余部分所需要的值，所以不需要担心由无穷计算所导致的 out-of-memory 错误。

（3）没有“副作用”

所谓"副作用"（side effect），指的是函数内部与外部互动（最典型的情况，就是修改全局变量的值），产生运算以外的其他结果。函数式编程强调没有"副作用"，意味着函数要保持独立，所有功能就是返回一个新的值，没有其他行为，尤其是不得修改外部变量的值。

综上所述，函数式编程可以简言之是： 使用不可变值和函数， 函数对一个值进行处理， 映射成另一个值。这个值在面向对象语言中可以理解为对象，另外这个值还可以作为函数的输入。

1.2 Lambda表达式

```
lambda表达式本质上是一段匿名内部类，也可以是一段可以传递的代码
```

1.2.1 语法

完整的Lambda表达式由三部分组成：参数列表、箭头、声明语句；

```
(Type1 param1, Type2 param2, ..., TypeN paramN) -> { statment1; statment2; //.............. return statmentM;}
```

1. 绝大多数情况，编译器都可以从上下文环境中推断出lambda表达式的参数类型，所以参数可以省略：

```
(param1,param2, ..., paramN) -> { statment1; statment2; //.............. return statmentM;}
```

2、 当lambda表达式的参数个数只有一个，可以省略小括号：

```
param1 -> { statment1; statment2; //.............. return statmentM;}
```

3、 当lambda表达式只包含一条语句时，可以省略大括号、return和语句结尾的分号：

```
param1 -> statment
```

1.2.2 函数接口

函数接口是只有一个抽象方法的接口， 用作 Lambda 表达式的返回类型。

接口包路径为java.util.function，然后接口类上面都有@FunctionalInterface这个注解。

这些函数接口在使用Lambda表达式时做为返回类型，JDK定义了很多现在的函数接口，实际自己也可以定义接口去做为表达式的返回，只是大多数情况下JDK定义的直接拿来就可以用了。而且这些接口在JDK8集合类使用流操作时大量被使用

1.2.3 类型检查、类型推断

Java编译器根据 Lambda 表达式上下文信息就能推断出参数的正确类型。 程序依然要经过类型检查来保证运行的安全性， 但不用再显式声明类型罢了。 这就是所谓的类型推断。Lambda 表达式中的类型推断， 实际上是 Java 7 就引入的目标类型推断的扩展

有时候显式写出类型更易读，有时候去掉它们更易读。没有什么法则说哪种更好；对于如何让代码更易读，你必须做出自己的选择

```
Comparator<Apple> c =
    (Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight());   <— 没有类型推断

Comparator<Apple> c =
    (a1, a2) -> a1.getWeight().compareTo(a2.getWeight());       <—— 有类型推断
```

1.2.4 局部变量限制

Lambda表达式也允许使用自由变量（不是参数，而是在外层作用域中定义的变量），就像匿名类一样。 它们被称作捕获Lambda。 Lambda可以没有限制地捕获（也就是在其主体中引用）实例变量和静态变量。但局部变量必须显式声明为final，或事实上是final。

为什么局部变量有这些限制?

（1）实例变量和局部变量背后的实现有一个关键不同。实例变量都存储在堆中，而局部变量则保存在栈上。如果Lambda可以直接访问局部变量，而且Lambda是在一个线程中使用的，则使用Lambda的线程，可能会在分配该变量的线程将这个变量收回之后，去访问该变量。因此， Java在访问自由局部变量时，实际上是在访问它的副本，而不是访问原始变量。如果局部变量仅仅赋值一次那就没有什么区别了——因此就有了这个限制。

（2）这一限制不鼓励你使用改变外部变量的典型命令式编程模式。

二、流

2.1 流介绍

流是Java API的新成员，它允许你以声明性方式处理数据集合（通过查询语句来表达，而不是临时编写一个实现）。就现在来说，你可以把它们看成遍历数据集的高级迭代器。此外，流还可以透明地并行处理，你无需写任何多线程代码了!

2.2 使用流

类别	方法名	方法签名	作用
筛选切片	filter	Stream<T> filter(Predicate<? super T> predicate)	过滤操作，根据Predicate判断结果保留为真的数据，返回结果仍然是流
	distinct	Stream<T> distinct()	去重操作，筛选出不重复的结果，返回结果仍然是流
	limit	Stream<T> limit(long maxSize)	截取限制操作，只取前 maxSize条数据，返回结果仍然是流
	skip	Stream<T> skip(long n)	跳过操作，跳过n条数据，取后面的数据，返回结果仍然是流
映射	map	<R> Stream<R> map(Function<? super T, ? extends R> mapper)	转化操作，根据参数T，转化成R类型，返回结果仍然是流

	flatMap	<R> Stream<R> flatMap(Function<? super T, ? extends Stream<? extends R>> mapper)	转化操作，根据参数T，转化成R类型流，这里会生成多个R类型流，返回结果仍然是流
匹配	anyMatch	boolean anyMatch(Predicate<? super T> predicate)	判断是否有一条匹配，根据Predicate判断结果中是否有一条匹配成功
	allMatch	boolean allMatch(Predicate<? super T> predicate)	判断是否全都匹配，根据Predicate判断结果中是否全部匹配成功
	noneMatch	boolean noneMatch(Predicate<? super T> predicate)	判断是否一条都不匹配，根据Predicate判断结果中是否所有的都不匹配
查找	findAny	Optional<T> findAny()	查找操作， 查询当前流中的任意元素并返回Optional
	findFirst	Optional<T> findFirst()	查找操作， 查询当前流中的第一个元素并返回Optional

归约	reduce	T reduce(T identity, BinaryOperator<T> accumulator);	归约操作，同样两个类型的数据进行操作后返回相同类型的结果。比如两个整数相加、相乘等。
	max	Optional<T> max(Comparator<? super T> comparator)	求最大值，根据Comparator计算的比较结果得到最大值
	min	Optional<T> min(Comparator<? super T> comparator)	求最小值，根据Comparator计算的比较结果得到最小值
汇总统计	collect	<R, A> R collect(Collector<? super T, A, R> collector)	汇总操作，汇总对应的处理结果。这里经常与
	count	long count()	统计流中数据数量
遍历	foreach	void forEach(Consumer<? super T> action)	遍历操作，遍历执行Consumer 对应的操作

上面是Stream API的一些常用操作，按场景结合lambda表达式调用对应方法即可。至于Stream的生成方式，Stream的of方法或者Collection接口实现类的stream方法都可以获得对应的流对象，再进一步根据需要做对应处理。

另外上述方法如果返回是Stream对象时是可以链式调用的，这个时候这个操作只是声明或者配方，不产生新的集合，这种类型的方法是惰性求值方法；有些方法返回结果非Stream类型，则是及早求值方法。

“为什么要区分惰性求值和及早求值？ 只有在对需要什么样的结果和操 作有了更多了解之后， 才能更有效率地进行计算。 例如， 如果要找出大于 10 的第一个数字， 那么并不需要和所有元素去做比较， 只要找出第一个匹配的元素就够了。 这也意味着可以在集合类上级联多种操作， 但迭代只需一次。这也是函数编程中惰性计算的特性，即只在需要产生表达式的值时进行计算。这样代码更加清晰，而且省掉了多余的操作。

这里还对上述列表操作中相关的Optional与Collectors类做下说明。

Optional类是为了解决经常遇到的NullPointerException出现的，这个类是一个可能包含空值的容器类。用Optional替代null可以显示说明结果可能为空或不为空，再使用时使用isPresent方法判断就可以避免直接调用的空指针异常。

Collectors类是一个非常有用的是归约操作工具类，工具类中的方法常与流的collect方法结合使用。比如
groupingBy方法可以用来分组，在转化Map时非常实用；partitioningBy方法可以用来分区（分区可以当做一种特殊的分组，真假值分组），joining方法可以用来连接，这个应用在比如字符串拼接的场景。

2.3 并行流

Collection接口的实现类调用parallelStream方法就可以实现并行流，相应地也获得了并行计算的能力。或者Stream接口的实现调用parallel方法也可以得到并行流。
并行流实现机制是基于fork/join 框架，将问题分解再合并处理。
不过并行计算是否一定比串行快呢？这也不一定。实际影响性能的点包括：
（1）数据大小输入数据的大小会影响并行化处理对性能的提升。 将问题分解之后并行化处理， 再将结果合并会带来额外的开销。 因此只有数据足够大、 每个数据处理管道花费的时间足够多
时， 并行化处理才有意义。
（2） 源数据结构
每个管道的操作都基于一些初始数据源， 通常是集合。 将不同的数据源分割相对容易，这里的开销影响了在管道中并行处理数据时到底能带来多少性能上的提升。
（3） 装箱
处理基本类型比处理装箱类型要快。
（4） 核的数量
极端情况下， 只有一个核， 因此完全没必要并行化。 显然， 拥有的核越多，获得潜在性能提升的幅度就越大。 在实践中， 核的数量不单指你的机器上有多少核， 更是指运行时你的机器能使用多少核。 这也就是说同时运行的其他进程， 或者线程关联性（ 强制线程在某些核或 CPU 上运行） 会影响性能。
（5） 单元处理开销
比如数据大小， 这是一场并行执行花费时间和分解合并操作开销之间的战争。
花在流中

每个元素身上的时间越长， 并行操作带来的性能提升越明显

实际在考虑是否使用并行时需要考虑上面的要素。在讨论流中单独操作每一块的种类时， 可以分成两种不同的操作： 无状态的和有状态的。无状态操作整个过程中不必维护状态， 有状态操作则有维护状态所需的开销和限制。如果能避开有状态， 选用无状态操作， 就能获得更好的并行性能。 无状态操作包括 map、filter 和 flatMap， 有状态操作包括 sorted、 distinct 和 limit。这种理解在理论上是更好的，当然实际使用还是以测试结果最为可靠 。

三、方法引用

方法引用的基本思想是，如果一个Lambda代表的只是“直接调用这个方法”，那最好还是用名称来调用它，而不是去描述如何调用它。事实上，方法引用就是让你根据已有的方法实现来创建Lambda表达式。但是，显式地指明方法的名称，你的代码的可读性会更好。所以方法引用只是在内容中只有一个表达式的简写。

当 你 需 要使用 方 法 引用时， 目 标引用 放 在 分隔符::前 ， 方法 的 名 称放在后 面 ， 即ClassName :: methodName 。例如 ， Apple::getWeight就是引用了Apple类中定义的方法getWeight。请记住，不需要括号，因为你没有实际调用这个方法。方法引用就是Lambda表达式(Apple a) -> a.getWeight()的快捷写法。

这里有种情况需要特殊说明，就是类的构造函数情况，这个时候是通过ClassName::new这种形式创建Class构造函数对应的引用，例如：

```
Supplier<Apple> c1 = Apple::new;
Apple a1 = c1.get();
```

构造函数引用指向默认的Apple()构造函数

调用Supplier的get方法将产生一个新的Apple

这就等价于：

```
Supplier<Apple> c1 = () -> new Apple();
Apple a1 = c1.get();
```

利用默认构造函数创建Apple的Lambda表达式

调用Supplier的get方法将产生一个新的Apple

四、默认方法

4.1 介绍

为了以兼容方式改进API，Java 8中加入了默认方法。主要是为了支持库设计师，让他们能够写出更容易改进的接口。具体写法是在接口中加default关键字修饰。
4.2 使用说明

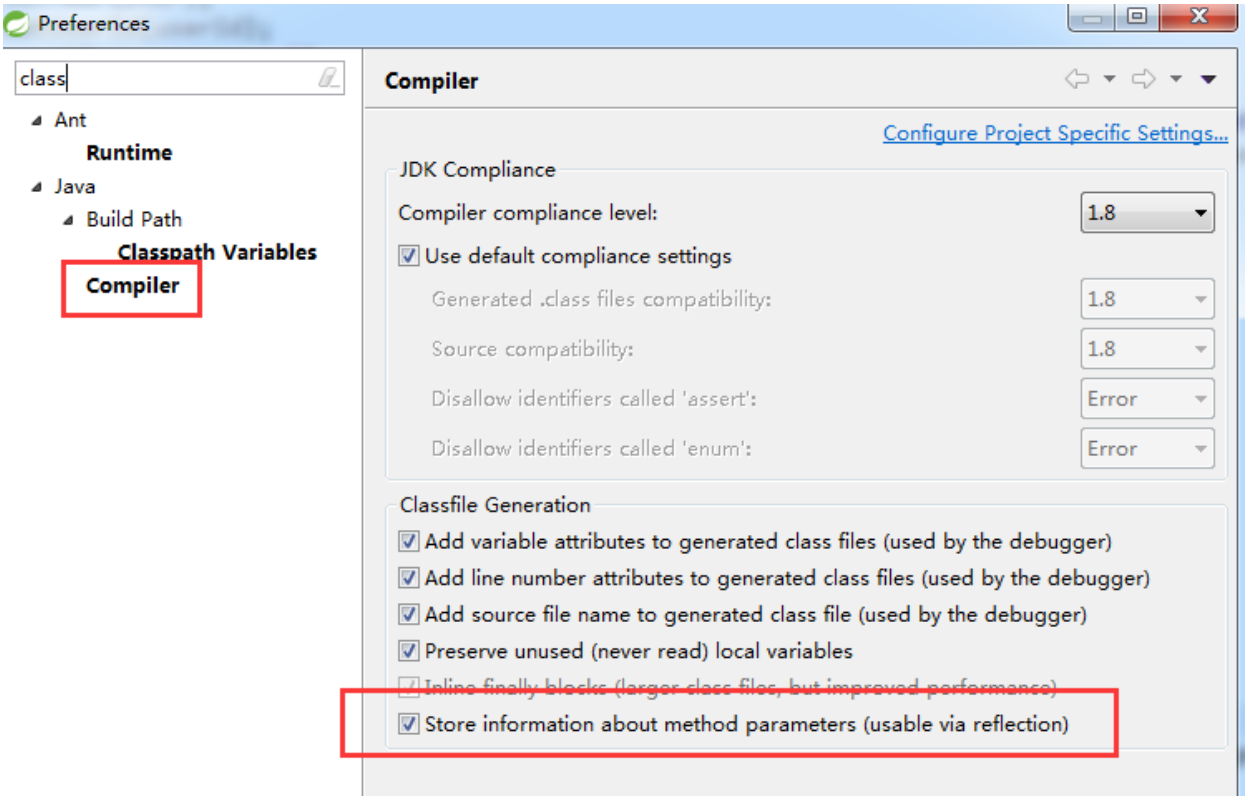
默认方法由于是为了避免兼容方式改进API才引入，所以一般正常开发中不会使用，除非你也想改进API，而不影响老的接口实现。当然在JDK8有大量的地方用到了默认方法，所以对这种写法有一定的了解还是有帮助的。
采用默认方法之后，你可以为这个方法提供一个默认的实现，这样实体类就无需在自己的实现中显式地提供一个空方法，而是默认就有了实现。
4.3 注意事项

由于类可以实现多个接口，也可以继承类，当接口或类中有相同函数签名的方法时，这个时候到底使用哪个类或接口的实现呢?
这里有三个规则可以进行判断:
(1) 类中的方法优先级最高。类或父类中声明的方法的优先级高于任何声明为默认方法的优先级。
(2) 如果无法依据第一条进行判断，那么子接口的优先级更高：函数签名相同时，优先选择拥有最具体实现的默认方法的接口，即如果B继承了A，那么B就比A更加具体。
(3) 最后，如果还是无法判断，继承了多个接口的类必须通过显式覆盖和调用期望的方法，显式地选择使用哪一个默认方法的实现。不然编译都会报错。

五、方法参数反射

JDK8 新增了Method.getParameters方法，可以获取参数信息，包括参数名称。不过为了避免.class文件因为保留参数名而导致.class文件过大或者占用更多的内存，另外也避免有些参数（ secret/password）泄露安全信息，JVM默认编译出的class文件是不会保留参数名这个信息的。

这一选项需由编译开关 javac -parameters 打开，默认是关闭的。在Eclipse（或者基于Eclipse的IDE）中可以如下图勾选保存:



六、日期/时间改进

```
1  * 之前使用的java.util.Date月份从0开始，我们一般会+1使用，很不方便，java.time.LocalDate月份和星期都改成了enum
2  * java.util.Date和SimpleDateFormat都不是线程安全的，而LocalDate和LocalTime和最基本的String一样，是不变类型，不但线程安全，而且不能修改。
3  * java.util.Date是一个“万能接口”，它包含日期、时间，还有毫秒数，更加明确需求取舍
4  * 新接口更好用的原因是考虑到了日期时间的操作，经常发生往前推或往后推几天的情况。用java.util.Date配合Calendar要写好多代码，而且一般的开发人员还不一定能写对。
```

1.8之前JDK自带的日期处理类非常不方便，我们处理的时候经常是使用的第三方工具包，比如commons-lang包等。不过1.8出现之后这个改观了很多，比如日期时间的创建、比较、调整、格式化、时间间隔等。

这些类都在java.time包下。比原来实用了很多。

6.1 LocalDate/LocalTime/LocalDateTime

LocalDate为日期处理类、LocalTime为时间处理类、LocalDateTime为日期时间处理类，方法都类似，具体可以看API文档或源码，选取几个代表性的方法做下介绍。

now相关的方法可以获取当前日期或时间，of方法可以创建对应的日期或时间，parse方法可以解析日期或时间，get方法可以获取日期或时间信息，with方法可以设置日期或时间信息，plus或minus方法可以增减日期或时间信息；

6.2 TemporalAdjusters

这个类在日期调整时非常有用，比如得到当月的第一天、最后一天，当年的第一天、最后一天，下一周或前一周的某天等。

6.3 DateTimeFormatter

以前日期格式化一般用SimpleDateFormat类，但是不怎么好用，现在1.8引入了DateTimeFormatter类，默认定义了很多常量格式（ISO打头的），在使用的时候一般配合LocalDate/LocalTime/LocalDateTime使用，比如想把当前日期格式化成yyyy-MM-dd hh:mm:ss的形式:

```
1 LocalDateTime dt = LocalDateTime.now();
2 DateTimeFormatter dtf = DateTimeFormatter.ofPattern("yyyy-MM-dd hh:mm:ss");
3 System.out.println(dtf.format(dt));
```