Lambda 的状态

2013年九月

Java SE 8 版

这是对JSR 335指定并在 OpenJDK Lambda 项目中实现的 Java 编程语言增强功能的非正式概述。它改进了2011 年 12 月发布的上一次迭代。一些语言更改的正式描述可以在 JSR的早期草案规范中找到；OpenJDK开发者预览版 也可用。其他历史设计文档可以在 OpenJDK 项目页面上找到。还有一个配套文档，State of the Lambda, Libraries Edition，描述了作为 JSR 335 的一部分添加的库增强功能。

Project Lambda 的高级目标是使需要将代码建模为数据的编程模式在 Java 中变得方便和惯用。主要的新语言功能包括：

- Lambda 表达式（非正式地，"闭包"或"匿名方法"）
- 方法和构造函数引用
- 扩展的目标类型和类型推断
- 接口中的默认方法和静态方法

这些在下面进行描述和说明。

1. 背景

Java 主要是一种面向对象的编程语言。在面向对象和函数式语言中，基本值都可以动态封装程序行为：面向对象语言有对象和方法，函数式语言有函数。然而，这种相似性可能并不明显，因为 Java 对象往往是相对重量级的：包装少数字段和许多方法的单独声明的类的实例化。

然而，对于某些对象来说，本质上只编码一个函数是很常见的。在典型的用例中，Java API 定义了一个接口，有时被描述为"回调接口"，期望用户在调用 API 时提供接口的实例。例如：

```
public interface ActionListener {
    void actionPerformed(ActionEvent e);
}
```

用户通常会匿名地内联实例化实现类，而不是声明一个实现`ActionListener`的唯一目的是在调用站点上分配一次的类：

```
button.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    ui.dazzle(e.getModifiers());
  }
});
```

许多有用的库都依赖于这种模式。这对于并行 API 尤其重要，其中要执行的代码必须独立于运行它的线程来表达。并行编程领域特别令人感兴趣，因为摩尔定律继续为我们提供更多内核但不是更快的内核，串行 API 仅限于可用处理能力的缩小部分。

鉴于回调和其他函数式风格习语的相关性越来越高，在 Java 中将代码建模为数据尽可能轻量是很重要的。在这方面，匿名内部类是不完善的，原因有很多，主要是：

1. 庞大的语法
2. 围绕名称和含义的混淆`this`
3. 不灵活的类加载和实例创建语义
4. 无法捕获非最终局部变量
5. 无法抽象控制流

这个项目解决了许多这些问题。它通过引入具有本地范围规则的新的、更简洁的表达式形式来消除 (1) 和 (2)，通过以更灵活、优化友好的方式定义新表达式的语义来回避 (3) 并改进 (4)通过允许编译器推断最终性（允许捕获 *有效的最终*局部变量）。

然而，解决内部类的所有问题*并不是*这个项目的目标。可变变量的任意捕获 (4) 和非局部控制流 (5) 都不在该项目的范围内（尽管这些功能可能会在该语言的未来迭代中重新访问。）

2. 功能接口

匿名内部类方法尽管有其局限性，但具有非常适合 Java 类型系统的良好特性：具有接口类型的函数值。这很方便，原因有很多：接口已经是类型系统的固有部分；它们自然具有运行时表示；它们带有由 Javadoc 注释表达的非正式契约，例如一个操作是可交换的断言。

上面使用的接口`ActionListener`只有一种方法。很多常见的回调接口都有这个属性，比如`Runnable`和 `Comparator`。我们将为所有只有一种方法的接口命名：*函数式接口*。（这些以前称为 *SAM 类型*，代表"单一抽象方法"。）

不需要做任何特别的事情来将接口声明为功能性的；编译器根据其结构对其进行识别。（这个识别过程不仅仅是计算方法声明；接口可能会冗余声明类自动提供的方法`Object`，例如 `toString()`，或者可能声明静态或默认方法，这些都不计入单一方法的限制.）然而，API 作者可能会额外捕获带有注释的接口功能性（而不是偶然只有一个方法）的*设计意图*`@FunctionalInterface`，在这种情况下，编译器将验证接口是否满足结构性要求成为功能性接口.

一些早期提议建议的函数类型的替代（或补充）方法是引入一种新的 *结构*函数类型，有时称为*箭头类型*。像"从 a`String`和 an`Object`到a 的函数"这样的类型`int`可以表示为`(String,Object)->int`. 由于以下几个缺点，这个想法至少在目前被考虑并拒绝了：

- 它会增加类型系统的复杂性，并进一步混合结构类型和名义类型（Java 几乎完全是名义类型）。
- 这将导致库风格的分歧——一些库将继续使用回调接口，而其他库将使用结构函数类型。
- 语法可能很笨拙，尤其是在包含检查异常时。
- 每个不同的函数类型不太可能有运行时表示，这意味着开发人员将进一步暴露于擦除并受到擦除的限制。例如，不可能（也许令人惊讶）重载方法`m(T->U)`和`m(X->Y)`.

因此，我们转而遵循"使用你所知道的"的路径——由于现有的库广泛使用函数式接口，我们编纂并利用了这种模式。这使得 *现有的*库可以与 lambda 表达式一起使用。

为了说明，这里是 Java SE 7 中已经存在的一些功能接口的示例，这些接口非常适合与新语言特性一起使用；下面的例子说明了其中一些的使用。

- `java.lang.Runnable`
- `java.util.concurrent.Callable`
- `java.security.PrivilegedAction`
- `java.util.Comparator`
- `java.io.FileFilter`
- `java.beans.PropertyChangeListener`

此外，Java SE 8 增加了一个新的包，`java.util.function`，其中包含预期常用的功能接口，例如：

- `Predicate<T>`-- 对象的布尔值属性
- `Consumer<T>`-- 对对象执行的操作
- `Function<T,R>`-- 将 T 转换为 R 的函数
- `Supplier<T>`-- 提供一个 T 的实例（例如一个工厂）
- `UnaryOperator<T>`-- 从 T 到 T 的函数
- `BinaryOperator<T>`-- 从 (T, T) 到 T 的函数

除了这些基本的"形状"之外，还有诸如`IntSupplier`or之类的原始特化`LongBinaryOperator`。（我们没有提供原始特化的完整补充，而是仅提供`int`、`long`和`double`的特化；其他原始类型可以通过转换来适应。）类似地，也有一些针对多个元的特化，例如`BiFunction<T,U,R>`，它表示从`(T,U)`到 的函数`R`.

3. Lambda 表达式

匿名类的最大痛点是笨重。它们有我们可以称之为"垂直问题"的东西：`ActionListener`第 1 节中的实例使用五行源代码来封装行为的一个方面。

Lambda 表达式是匿名方法，旨在通过用更轻量级的机制替换匿名内部类的机制来解决"垂直问题"。

以下是 lambda 表达式的一些示例：

```
(int x, int y) -> x + y

() -> 42

(String s) -> { System.out.println(s); }
```

第一个表达式接受两个整数参数，名为`x`和`y`，并返回它们的总和。第二个*不*带参数并返回整数`42`。第三个接受一个字符串并将其打印到控制台，不返回任何内容。

一般语法由参数列表、箭头标记`->`和正文组成。主体可以是单个表达式，也可以是语句块。在表达式形式中，主体被简单地计算并返回。在块形式中，主体像方法主体一样被评估——一条`return`语句将控制权返回给匿名方法的调用者；`break`并且`continue`在顶层是非法的，但在循环内当然是允许的；如果主体产生结果，则每个控制路径都必须返回某些内容或抛出异常。

语法针对 lambda 表达式非常小的常见情况进行了优化，如上所示。例如，表达式主体形式消除了对`return`关键字的需要，否则它可能代表相对于表达式大小的大量语法开销。

还期望 lambda 表达式经常出现在嵌套上下文中，例如方法调用的参数或*另一个*lambda 表达式的结果。为了在这些情况下尽量减少噪音，避免使用不必要的分隔符。但是，对于需要将整个表达式分开的情况，它可以用括号括起来，就像任何其他表达式一样。

以下是出现在语句中的 lambda 表达式的一些示例：

```
FileFilter java = (File f) -> f.getName().endsWith(".java");

String user = doPrivileged(() -> System.getProperty("user.name"));

new Thread(() -> {
  connectToService();
  sendNotification();
}).start();
```

4.目标打字

请注意，函数接口的名称*不是*lambda 表达式语法的一部分。那么 lambda 表达式代表什么样的对象呢？它的类型是从周围的上下文中推断出来的。例如，以下 lambda 表达式是`ActionListener`：

```
ActionListener l = (ActionEvent e) -> ui.dazzle(e.getModifiers());
```

这种方法的一个含义是同一个 lambda 表达式在不同的上下文中可以有不同的类型：

```
Callable<String> c = () -> "done";

PrivilegedAction<String> a = () -> "done";
```

在第一种情况下，lambda 表达式`() -> "done"`表示`Callable`. 在第二种情况下，相同的表达式表示 的一个实例`PrivilegedAction`。

编译器负责推断每个 lambda 表达式的类型。它使用表达式出现的上下文中预期的类型；这种类型称为*目标类型*。lambda 表达式只能出现在目标类型为函数式接口的上下文中。

当然，没有任何 lambda 表达式会兼容*所有* 可能的目标类型。编译器检查 lambda 表达式使用的类型是否与目标类型的方法签名一致。也就是说，`T`如果满足以下所有条件，则可以将 lambda 表达式分配给目标类型：

- `T`是功能接口类型
- lambda 表达式与 ' 方法具有相同数量的参数`T`，并且这些参数的类型相同
- lambda 主体返回的每个表达式都与`T`' 方法的返回类型兼容
- `T`' 方法的 `throws`子句允许 lambda 主体抛出的每个异常

由于函数式接口目标类型已经"知道" lambda 表达式的形式参数应该具有什么类型，因此通常没有必要重复它们。使用目标类型可以推断 lambda 参数的类型：

```
Comparator<String> c = (s1, s2) -> s1.compareToIgnoreCase(s2);
```

在这里，编译器推断 和 的`s1`类型`s2`是`String`。此外，当只有一个参数的类型被推断（一种非常常见的情况）时，围绕单个参数名称的括号是可选的：

```
FileFilter java = f -> f.getName().endsWith(".java");

button.addActionListener(e -> ui.dazzle(e.getModifiers()));
```

这些增强进一步实现了一个理想的设计目标："不要将垂直问题变成水平问题"。我们希望代码的读者在到达 lambda 表达式的"内容"之前必须尽可能少地了解语法。

Lambda 表达式并不是第一个具有上下文相关类型的 Java 表达式：例如，泛型方法调用和"菱形"构造函数调用类似地基于赋值的目标类型进行类型检查。

```
List<String> ls = Collections.emptyList();
List<Integer> li = Collections.emptyList();

Map<String,Integer> m1 = new HashMap<>();
Map<Integer,String> m2 = new HashMap<>();
```

5. 目标类型的上下文

我们之前说过 lambda 表达式只能出现在具有目标类型的上下文中。以下上下文具有目标类型：

- 变量声明
- 作业
- 返回语句
- 数组初始化器
- 方法或构造函数参数
- Lambda 表达式体
- 条件表达式（`?:`）
- 演员表

在前三种情况下，目标类型只是被分配或返回的类型。

```
Comparator<String> c;
c = (String s1, String s2) -> s1.compareToIgnoreCase(s2);

public Runnable toDoLater() {
  return () -> {
    System.out.println("later");
  };
}
```

数组初始值设定项上下文类似于赋值，不同之处在于"变量"是一个数组组件，其类型派生自数组的类型。

```
filterFiles(new FileFilter[] {
               f -> f.exists(), f -> f.canRead(), f -> f.getName().startsWith("q")
            });
```

在方法参数的情况下，事情更复杂：目标类型确定与其他两个语言特性交互，*重载决议*和*类型参数推断*。

重载解决方案涉及为特定方法调用找到最佳方法声明。由于不同的声明具有不同的签名，这可能会影响用作参数的 lambda 表达式的目标类型。编译器将使用它对 lambda 表达式的了解来做出选择。如果 lambda 表达式是*显式类型*的（指定其参数的类型），编译器不仅会知道参数类型，还会知道其主体中所有返回表达式的类型。如果 lambda 是 *隐式类型*的（推断的参数类型），重载决策将忽略 lambda 主体并仅使用 lambda 参数的数量。

如果最佳方法声明的选择不明确，则强制转换或显式 lambda 可以为编译器提供额外的类型信息以消除歧义。如果 lambda 表达式的目标返回类型取决于类型参数推断，则 lambda 主体可能会向编译器提供信息以帮助推断类型参数。

```
List<Person> ps = ...
String<String> names = ps.stream().map(p -> p.getName());
```

这里，`ps`是 a `List<Person>`，所以`ps.stream()`也是 a `Stream<Person>`。该`map()`方法是泛型的`R`，其中的参数`map()`是a `Function<T,R>`，其中`T`是流元素类型。（此时`T`已知`Person`。）一旦选择了重载并且已知 lambda 的目标类型，我们需要推断`R`；我们通过对 lambda 主体进行类型检查来做到这一点，并发现它的返回类型是`String`，因此`R`是 `String`，因此`map()`表达式的类型是`Stream<String>`。大多数时候，编译器只是把这一切都弄清楚了，但如果它卡住了，我们可以通过显式 lambda 提供额外的类型信息（给出参数`p`显式类型），将 lambda 转换为显式目标类型，例如 ，或为泛型参数( )`Function<Person,String>`提供显式类型见证。`R.<String>map(p -> p.getName())`

Lambda 表达式本身为其主体提供目标类型，在这种情况下，通过从外部目标类型派生该类型。这使得编写返回其他函数的函数变得很方便：

```
Supplier<Runnable> c = () -> () -> { System.out.println("hi"); };
```

类似地，条件表达式可以从周围的上下文中“传递”一个目标类型：

```
Callable<Integer> c = flag ? (() -> 23) : (() -> 42);
```

最后，如果无法从上下文中方便地推断出 lambda 表达式的类型，则强制转换表达式提供了一种显式提供 lambda 表达式类型的机制：

```
// Illegal: Object o = () -> { System.out.println("hi"); };
Object o = (Runnable) () -> { System.out.println("hi"); };
```

当方法声明被不相关的功能接口类型重载时，强制转换也有助于解决歧义。

目标类型在编译器中的扩展作用不仅限于 lambda 表达式：泛型方法调用和“钻石”构造函数调用也可以利用目标类型，只要它们可用。以下声明在 Java SE 7 中是非法的，但在 Java SE 8 中有效：

```
List<String> ls =
  Collections.checkedList(new ArrayList<>(), String.class);

Set<Integer> si = flag ? Collections.singleton(23)
                       : Collections.emptySet();
```

6. 词法作用域

确定`this`内部类中名称（和）的含义比将类限制在顶层时要困难得多且容易出错。继承的成员——包括类的方法——可能会意外地隐藏外部声明，以及始终引用内部类本身的`Object`非限定引用。`this`

Lambda 表达式要简单得多：它们不从超类型继承任何名称，也不引入新级别的作用域。相反，它们是词法范围的，这意味着主体中的名称被解释为就像它们在封闭环境中一样（为 lambda 表达式的形式参数添加新名称）。作为自然扩展，`this`关键字及其成员的引用与它们在 lambda 表达式之外的含义相同。

为了说明，以下程序`"Hello, world!"`向控制台打印两次：

```
public class Hello {
  Runnable r1 = () -> { System.out.println(this); }
  Runnable r2 = () -> { System.out.println(toString()); }

  public String toString() { return "Hello, world!"; }

  public static void main(String... args) {
    new Hello().r1.run();
    new Hello().r2.run();
  }
}
```

使用匿名内部类的等价物可能会打印出类似`Hello$1@5b89a773`and 的东西，这可能会让程序员感到惊讶`Hello$2@537a7706`。

与词法范围方法一致，并遵循其他局部参数化结构（如`for`循环和`catch` 子句）设置的模式，lambda 表达式的参数不得遮蔽封闭上下文中的任何局部变量。

7. 变量捕获

在 Java SE 7中，编译器检查对内部类中封闭上下文的局部变量（*捕获*的变量）的引用非常严格：如果未声明捕获的变量，则会发生错误 `final`。我们放宽了这个限制——对于 lambda 表达式和内部类——还允许捕获 *有效的最终*局部变量。

非正式地，如果一个局部变量的初始值从不改变，那么它实际上是最终的——换句话说，声明它`final`不会导致编译失败。

```
Callable<String> helloCallable(String name) {
  String hello = "Hello";
  return () -> (hello + ", " + name);
}
```

引用`this`——包括通过非限定字段引用或方法调用的隐式引用——本质上是对`final`局部变量的引用。包含此类引用的 Lambda 主体捕获`this`. 在其他情况下，`this`对象不保留对的引用。

这对内存管理有一个有益的影响：虽然内部类实例始终持有对其封闭实例的强引用，但不从封闭实例捕获成员的 lambdas*不*持有对它的引用。内部类实例的这一特性通常是内存泄漏的来源。

虽然我们放宽了对捕获值的语法限制，但我们仍然禁止捕获可变局部变量。原因是这样的成语：

```
int sum = 0;
list.forEach(e -> { sum += e.size(); }); // ERROR
```

基本上是连续的；编写像这样没有竞争条件的 lambda 体是相当困难的。除非我们愿意强制执行——最好是在编译时——这样的函数不能逃脱它的捕获线程，否则这个特性可能会造成比它解决的更多的麻烦。Lambda 表达式关闭 *值*，而不是*变量*。

不支持捕获可变变量的另一个原因是，有一种更好的方法可以解决没有突变的累积问题，而是将此问题视为*减少*。该 `java.util.stream`包提供了对集合和其他数据结构的通用和专用（例如 sum、min 和 max）缩减。例如`forEach`，我们可以不使用和突变，而是进行顺序或并行安全的归约：

```
int sum = list.stream()
               .mapToInt(e -> e.size())
               .sum();
```

提供该`sum()`方法是为了方便，但等效于更一般的归约形式：

```
int sum = list.stream()
               .mapToInt(e -> e.size())
               .reduce(0, (x,y) -> x+y);
```

归约接受一个基值（如果输入为空）和一个运算符（这里是加法），并计算以下表达式：

```
0 + list[0] + list[1] + list[2] + ...
```

减少也可以通过其他操作来完成，例如最小值、最大值、乘积等，并且如果运算符是关联的，则可以轻松安全地并行化。因此，我们不支持本质上是顺序的并且容易发生数据竞争（可变累加器）的惯用语，而是选择提供库支持来以更可并行化和不易出错的方式表达累加。

8. 方法参考

Lambda 表达式允许我们定义匿名方法并将其视为功能接口的实例。通常希望对*现有*方法执行相同的操作。

方法引用是与 lambda 表达式具有相同处理方式的表达式（即，它们需要目标类型并编码功能接口实例），但它们不提供方法体，而是通过名称引用现有方法。

例如，考虑一个`Person`可以按名称或年龄排序的类。

```
class Person {
    private final String name;
    private final int age;

    public int getAge() { return age; }
    public String getName() { return name; }
   ...
}

Person[] people = ...
Comparator<Person> byName = Comparator.comparing(p -> p.getName());
Arrays.sort(people, byName);
```

我们可以重写它以使用方法引用来`Person.getName()` 代替：

```
Comparator<Person> byName = Comparator.comparing(Person::getName);
```

在这里，表达式`Person::getName`可以被认为是 lambda 表达式的简写，它简单地调用带有参数的命名方法，并返回结果。虽然方法引用可能（在这种情况下）在语法上不再紧凑，但它更清晰——我们要调用的方法有一个名称，因此我们可以直接通过名称引用它。

因为函数式接口方法的参数类型在隐式方法调用中充当参数，所以引用的方法签名被允许操作参数——通过扩大、装箱、分组为变量数组等——就像方法调用一样.

```
Consumer<Integer> b1 = System::exit;   // void exit(int status)
Consumer<String[]> b2 = Arrays::sort;  // void sort(Object[] a)
Consumer<String> b3 = MyProgram::main; // void main(String... args)
Runnable r = MyProgram::main;          // void main(String... args)
```

9. 方法引用的种类

有几种不同类型的方法引用，每种的语法略有不同：

- 静态方法（`ClassName::methName`）
- 特定对象的实例方法（`instanceRef::methName`）
- `super`特定对象的方法（`super::methName`）
- 特定类型的任意对象的实例方法（`ClassName::methName`）
- 类构造函数引用（`ClassName::new`）
- 数组构造函数引用（`TypeName[]::new`）

对于静态方法引用，方法所属的类位于`::`分隔符之前，例如 in `Integer::sum`。

对于对特定对象的实例方法的引用，对对象引用求值的表达式位于分隔符之前：

```
Set<String> knownNames = ...
Predicate<String> isKnown = knownNames::contains;
```

在这里，隐式 lambda 表达式将捕获`String`引用的对象`knownNames`，并且主体将`Set.contains` 使用该对象作为接收者进行调用。

引用特定对象的方法的能力提供了一种在不同功能接口类型之间进行转换的便捷方式：

```
Callable<Path> c = ...
PrivilegedAction<Path> a = c::call;
```

对于任意对象的实例方法的引用，方法所属的类型在分隔符之前，调用的接收者是函数式接口方法的第一个参数：

```
Function<String, String> upperfier = String::toUpperCase;
```

这里，隐式 lambda 表达式有一个参数，即要转换为大写的字符串，它成为`toUpperCase()`方法调用的接收者。

如果实例方法的类是泛型的，则可以在`::`分隔符之前提供其类型参数，或者在大多数情况下，从目标类型推断。

请注意，静态方法引用的语法也可能被解释为对类的实例方法的引用。编译器通过尝试识别每种适用的方法来确定哪个是预期的（注意实例方法少了一个参数）。

对于所有形式的方法引用，方法类型参数根据需要进行推断，或者可以在`::` 分隔符之后显式提供。

通过使用名称，可以像静态方法一样引用构造函数`new`：

```
SocketImplFactory factory = MySocketImpl::new;
```

如果一个类有多个构造函数，则使用目标类型的方法签名来选择最佳匹配，就像解析构造函数调用一样。

对于内部类，没有语法支持在构造函数引用的位置显式提供封闭实例参数。

如果要实例化的类是泛型的，则可以在类名之后提供类型参数，或者将它们推断为"钻石"构造函数调用。

数组的构造函数引用有一种特殊的语法形式，它将数组视为具有接受`int`参数的构造函数。例如：

```
IntFunction<int[]> arrayMaker = int[]::new;
int[] array = arrayMaker.apply(10);  // creates an int[10]
```

10. 默认和静态接口方法

Lambda 表达式和方法引用为 Java 语言增加了很多表现力，但真正实现我们使代码即数据模式变得方便和惯用的目标的关键是通过为利用这些新特性而量身定制的库来补充这些新特性。

在 Java SE 7 中向现有库添加新功能有些困难。特别是，接口一旦发布就基本上是一成不变的。除非可以同时更新接口的所有可能实现，否则向接口添加新方法可能会导致现有实现中断。*默认方法*（以前称为*虚拟扩展方法*或*防御方法*）的目的是使接口在初始发布后能够以兼容的方式进行演进。

为了说明，标准集合 API 显然应该提供新的 lambda 友好操作。例如，该`removeAll`方法可以概括为删除任何集合的元素，该元素具有任意属性，其中该属性表示为功能接口的实例`Predicate`。但是这种新方法会在哪里定义呢？我们不能向 `Collection`接口添加抽象方法——许多现有的实现不知道这个变化。我们可以使它成为`Collections` 实用程序类中的静态方法，但这会将这些新操作降级为一种二等状态。

*默认方法*提供了一种更加面向对象的方式来向接口添加具体行为。这是一种新的方法：接口方法可以是*abstract*或*default*。默认方法的实现由不覆盖它的类继承（有关详细信息，请参见下一节）。功能接口中的默认方法不计入其一个抽象方法的限制。例如，我们可以（虽然没有）向 中添加一个`skip` 方法`Iterator`，如下所示：

```
interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove();

    default void skip(int i) {
        for (; i > 0 && hasNext(); i--) next();
    }
}
```

鉴于上述定义`Iterator`，所有实现的类都 `Iterator`将继承一个`skip`方法。从客户端的角度来看， `skip`只是接口提供的另一种虚拟方法。调用 不提供主体 `skip`的子类的实例具有调用默认实现的效果：调用和最多一定次数。如果一个类想要用更好的实现来覆盖——例如，通过直接推进一个私有游标，或者合并一个原子性保证——它是自由的。 `IteratorskiphasNextnextskip`

当一个接口扩展另一个接口时，它可以为继承的抽象方法添加默认值，为继承的默认方法提供新的默认值，或者通过将方法重新声明为抽象来重新抽象默认方法。

除了允许以默认方法的形式在接口中编写代码之外，Java SE 8 还引入了在接口中放置*静态* 方法的能力。这允许特定于接口的辅助方法与接口一起使用，而不是在侧类中（通常以接口的复数形式命名）。例如，获得了用于制作比较器的静态辅助方法，该方法采用提取排序键并生成的`Comparator`函数：`ComparableComparator`

```
public static <T, U extends Comparable<? super U>>
Comparator<T> comparing(Function<T, U> keyExtractor) {
    return (c1, c2) -> keyExtractor.apply(c1).compareTo(keyExtractor.apply(c2));
}
```

11.默认方法的继承

默认方法像其他方法一样被继承；在大多数情况下，行为与预期的一样。但是，当类或接口的超类型提供具有相同签名的多个方法时，继承规则会尝试解决冲突。两个基本原则驱动这些规则：

- 类方法声明优于接口默认值。无论类方法是具体的还是抽象的，都是如此。（因此**default** 关键字：如果类层次结构没有说明任何内容，则默认方法是后备方法。）
- 已被其他候选者覆盖的方法将被忽略。当超类型共享一个共同的祖先时，就会出现这种情况。

作为第二条规则如何发挥作用的示例，假设 **Collection**和**List**接口为 提供了不同的默认值 **removeAll**，并从;**Queue**继承了默认方法 。**Collection**在以下**implements**子句中，**List** 声明将优先于由**Collection**继承的声明**Queue**：

```
class LinkedList<E> implements List<E>, Queue<E> { ... }
```

如果两个独立定义的默认值发生冲突，或者默认方法与抽象方法发生冲突，则为编译错误。在这种情况下，程序员必须显式地重写超类型方法。通常，这相当于选择首选默认值，并声明调用首选默认值的主体。**super**支持调用特定超接口的默认实现的增强语法：

```
interface Robot implements Artist, Gun {
    default void draw() { Artist.super.draw(); }
}
```

前面的名称**super**必须引用定义或继承被调用方法的默认值的直接超接口。这种形式的方法调用不限于简单的消除歧义——它可以像任何其他调用一样在类和接口中使用。

inherits在任何情况下，在or子句中声明接口的顺序 **extends**，或者"首先"或"最近"实现的接口的顺序都不会影响继承。

12. 放在一起

Project Lambda 的语言和库功能旨在协同工作。为了说明，我们将考虑按姓氏对人员列表进行排序的任务。

今天我们写：

```
List<Person> people = ...
Collections.sort(people, new Comparator<Person>() {
    public int compare(Person x, Person y) {
        return x.getLastName().compareTo(y.getLastName());
    }
});
```

这是写"按姓氏排序"的一种非常冗长的方式！

使用 lambda 表达式，我们可以使这个表达式更简洁：

```
Collections.sort(people,
                 (Person x, Person y) -> x.getLastName().compareTo(y.getLastName()));
```

然而，虽然更简洁，但它不再抽象；它仍然使程序员需要进行实际比较（当排序键是原语时更糟）。对库的小改动在这里会有所帮助，例如添加到的静态**comparing**方法**Comparator**：

```
Collections.sort(people, Comparator.comparing((Person p) -> p.getLastName()));
```

这可以通过允许编译器推断 lambda 参数的类型并**comparing**通过静态导入导入方法来缩短：

```
Collections.sort(people, comparing(p -> p.getLastName()));
```

上面表达式中的 lambda 只是现有方法的转发器**getLastName**。我们可以使用方法引用来重用现有方法来代替 lambda 表达式：

```
Collections.sort(people, comparing(Person::getLastName));
```

最后，由于许多原因，使用类似的辅助方法**Collections.sort**是不可取的：它更冗长；它不能专门用于实现的每个数据结构**List**；并且它破坏了**List**接口的价值，因为用户**sort**在检查**List**.

默认方法为这个问题提供了一个更加面向对象的解决方案，我们在其中添加了一个**sort()**方法**List**：

```
people.sort(comparing(Person::getLastName));
```

这也读起来更像是问题陈述首先：**people**按姓氏对列表进行排序。

如果我们向 中添加一个默认方法**reversed()**，**Comparator**它生成**Comparator**使用相同排序键但顺序相反的 a，我们可以很容易地表示降序排序：

```
people.sort(comparing(Person::getLastName).reversed());
```

13. 总结

Java SE 8 添加了数量相对较少的新语言特性——lambda 表达式、方法引用、接口中的默认和静态方法，以及更广泛地使用类型推断。然而，综合起来，它们使程序员能够用更少的样板更清晰、更简洁地表达他们的意图，并能够开发更强大、对并行友好的库。