原文：<http://cr.openjdk.java.net/~briangoetz/lambda/lambda-state-final.html>

翻译：<http://zh.lucida.me/blog/java-8-lambdas-insideout-language-features/>

# 关于

本文介绍了Java SE 8中新引入的lambda语言特性以及这些特性背后的设计思想。这些特性包括：

* + lambda表达式（又被称为“闭包”或“匿名方法”）
  + 方法引用和构造方法引用
  + 扩展的目标类型和类型推导
  + 接口中的默认方法和静态方法

# 背景

Java是一门面向对象编程语言。面向对象编程语言和函数是编程语言中的基本元素（Basic Values）都可以动态封装程序行为：面向对象编程语言使用带有方法的对象封装行为，函数是编程语言使用函数封装对象行为。但这个相同点并不明显，因为Java对象往往比较“重量级”：实例化一个类型往往会涉及到不同的类，并需要初始化类里的字段和方法。

不过有些Java对象只是对单个函数的封装。例如下面这个典型用例：Java API中定义了一个接口（一般被称为回调接口），用户通过提供这个接口的实例来传入指定行为，例如：

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

这里并不需要专门定义一个类来实现ActionListener，因为它只会在调用处被调用一次。用户一般会使用匿名类型把行为内联（inline）：

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

很多库都依赖于上面的模式。对于并行API更是如此，因为我们需要把待执行的代码提供给并行API，并行编程是一个非常值得研究的领域，因为在这里摩尔定律得到了重生：尽管我们没有更快的CPU核心，但是我们有更多的CPU核心。而串行API就只能使用有限的计算能力。

随着回调模式和函数是编程风格的日益流行，我们需要在Java中提供一种尽可能轻量级的将代码模块封装为数据（Model code as data）的方法。匿名内部类并不是一个好的选择，因为：

1.语法过于冗余

2.匿名类中的this和变量名容易使用产生误解

3.类型载入和实例创建的语义不够灵活

4.无法捕获非final的局部变量

5.无法对控制流进行抽象

上面的多数问题均在Java SE 8中得以解决：

* 通过提供更简洁的语法和局部作用域规则，Java SE 8彻底解决了问题1和问题2
* 通过提供更加灵活而且便于优化的表达式语义，Java SE 8绕开了问题3
* 通过允许编译器推断变量的“常量性”（finality），Java SE 8减轻了问题4带来的困扰

不过，不过Java SE 8的目标并非解决所有上述问题，因此，捕获可变变量（问题4）和非局部控制流（问题5）并不在Java SE 8的范畴之内。

# 函数式接口

尽管匿名内部类有着种种限制和问题，但是它有一个良好的特性，它和Java类型系统结合的十分紧密：每一个函数对象都对应一个接口类型。之所以说这个特性是良好的，是因为：

* 接口是Java类型系统中固有的部分
* 接口天然就拥有其运行时表示（runtime representation）
* 接口可以通过Javadoc注释来表达一些非正式的协定（contract），例如，通过注释说明该操作应可交换（commutative）

上面提到的ActionListener接口只有一个方法，大多数回调接口都拥有这个特征：比如Runnable接口和Comparator接口。我们把这些只拥有一个方法的接口称为函数式接口。（之前它们被称为SAM类型，即单抽象方法类型（Single Abstract Method））

我们并不需要额外的工作来声明一个接口是函数式接口：编译器会根据接口的结构自行判断（判断过程并非简单的对接口方法计数：一个接口可能冗余的定义了一个Object已经提供的方法，比如toString()，或者定义了静态方法或默认方法，这些都不属于函数式接口方法的范畴）。不过API作者们可以通过@FunctionalInterface注解来显式指定一个接口是函数式接口（以避免无意声明了一个符合函数式标准的接口），加上这个注解之后，编译器就会验证该接口是否满足函数式接口的要求。

实现函数式类型的另一种（或补充）方式是引入一个全新的结构化函数类型，也被称为“箭头”类型。例如，一个接收String和Object并返回int的函数类型可以被表示为(String, Object) -> int。这种方法最终被放弃了（至少现在是这样），理由如下：

* 它会为Java类型系统引入额外的复杂度，进而会带来结构类型（Structural Type）和指名类型（Nominal Type）的混用。（Java几乎全部使用指名类型）
* 它会导致类库风格的分歧——一些类库可能会继续使用回调接口，而另一些类库可能会使用结构化函数类型
* 它的语法会变得十分笨拙，尤其在包含检查型异常时
* 每个函数类型很难拥有其运行时表示，这意味着开发者会受到类型擦除的困扰和局限，比如说，我们无法对m(T->U)和m(X->Y)进行重载

因此，最终Java SE 8中实现函数式类型的方式是：既然现有的类库大量的使用了函数式接口，那就修改并使得现有的类库能够直接使用lambda表达式。例如，Java SE 7中已经存在的一些函数式接口如下：

* java.lang.Runnale
* java.util.concurrent.Callable
* java.security.PrivilegedAction
* java.util.Comparator
* java.io.FileFilter
* java.beans.PropertyChangeListener

除此之外，Java SE 8中还增加了一个新的包：java.util.function，它里面包含了常用的函数式接口，例如：

* Predicate<T>：接收T并返回boolean
* Consumer<T>：接收T，不返回值
* Predicate<T>：接收T并返回 boolean
* Consumer<T>：接收T不返回值
* Function<T, R>：接收T，返回 R
* Supplier<T>：提供T对象（例如工厂），不接收值
* UnaryOperator<T>：接收T对象，返回T
* BinaryOperator<T>：接收两个T，返回T

除了上面的这些基本的函数式接口，Java SE 8还提供了一些针对原始类型的函数式接口，例如IntSupplier和LongBinaryOperator，也提供了一些针对多个参数的函数式接口，例如Bifunction<T, U, R>，它接收T对象和U对象，返回R对象。

# lambda表达式

匿名类型最大的问题就在于其冗余的语法。有人戏称匿名类型导致了“高度问题”（height problem）：比如前面ActionListener的例子里，五行代码中仅有一行在做实际工作。

lambda表达式时匿名方法，它提供了轻量级的语法，从而解决了匿名内部类带来的“高度问题”。下面是一些lambda表达式：

|  |
| --- |
| (int x, int y) -> x + y  () -> 42  (String s) -> { System.out.println(s); } |

第一个lambda表达式接收x和y这两个整形参数并返回它们的和；第二个lambda表达式不接收参数，返回整数‘42’；第三个lambda表达式接收一个字符串并把它打印到控制台，不返回值。

lambda表达式的语法由参数列表、箭头符号->和函数体组成。函数体既可以是一个表达式，也可以是一个语句块：

* 表达式：表达式会被执行然后返回执行结果。
* 语句块：语句块中的语句会被依次执行，就像方法中的语句一样

表达式函数体适合小型lambda表达式，它消除了return关键字，使得语法更加简洁。

lambda表达式也会经常出现在嵌套环境中，比如作为方法参数或者作为另外一个lambda表达式的返回值。为了使lambda表达式在这些场景下尽可能简洁，可以去掉不必要的分隔符。不过在某些情况下，也可以像普通表达式一样，用括号将lambda表达式括起来将其分为多行。下面是一些出现在语句中的lambda表达式：

|  |
| --- |
| FileFilter java = (File f) -> f.getName().endsWith("\*.java");  String user = doPrivileged(() -> System.getProperty("user.name"));  new Thread(() -> {  connectToService();  sendNotification();  }).start(); |

# 目标类型

需要注意的是，函数式接口的名称并不是lambda表达式的一部分。那么问题来了，对于给定的lambda表达式，它的类型是什么？答案是：它的类型是由其上下文推导而来的。例如，下面代码中的lambda表达式类型是ActionListener：

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

这就意味着同样的lambda表达式在不同上下文里可以拥有不同的类型：

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

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

第一个lambda表达式() -> "done"是Callable实例，而第二个lambda表达式则是PrivilegedAction的实例。

编译器负责推导lambda表达式类型，它利用lambda表达式所在上下文所期待的类型进行推导，这个被期待的类型被称为目标类型。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，lambda表达式对目标类型也是有要求的。编译器会检查lambda表达式的类型和目标类型的方法签名是否一致，当且仅当下面所有条件均满足时，lambda表达式才可以被赋给目标类型T：

* T是一个函数式接口
* lambda表达式的参数和T的方法参数在数量、类型和顺序上一致
* lambda表达式的返回值和T的方法返回值兼容
* lambda表达式内抛出的异常和T的方法异常兼容

由于lambda表达式的参数类型可以从目标类型中推导出，因此没必要显式声明lambda表达式的参数类型：

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

在上面的例子中，编译器可以推导出s1和s2的类型是String。另外，当lambda的参数只有一个而且它的类型可以被推导出来时，则lambda表达式的参数列表外面的括号可以被省略：

|  |
| --- |
| FileFilter java = f -> f.getName().endsWith(".java");  button.addActionListener(e -> ui.dazzle(e.getModifiers())); |

这些改进进一步展示了Java SE 8中lambda的设计目标：不要把高度问题转换为宽度问题，即希望代码的读者能够阅读尽可能少的语法就可以直达lambda表达式的核心。

lambda表达式并不是第一个拥有上下文相关类型的Java表达式：泛型方法调用和“菱形”构造器调用也通过目标类型来进行类型推导：

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

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

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

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

# 目标类型的上下文

之前我们提到lambda表达式只能出现在拥有目标类型的上下文中，下面给出了这些带有目标类型的上下文：

* 变量声明
* 赋值
* 返回语句
* 数组初始化
* 方法和构造方法的参数
* lambda 表达式函数体
* 条件表达式（? :）
* 转型（Cast）表达式

在前三个上下文（变量声明、赋值和返回语句）里，目标类型即是被赋值或返回的类型：

|  |
| --- |
| Comparator<String> c;  c = (String s1, String s2) -> s1.compareToIgnoreCase(s2);  public Runnable toDoLater() {  return () -> {  System.out.println("later");  }  } |

数组初始化和赋值类似，只是这里的“变量”变成了数组元素，而类型则是从数组类型推导得知：

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

方法参数的类型推导要相对复杂些：目标类型的确认会涉及到其他两个语言特性：重载解析（Overload resolution）和参数类型推导（Type argument inference）。

重载解析会为一个给定的方法调用寻找最合适的方法声明。由于不同的方法声明具有不同的方法签名，当lambda表达式作为方法参数时，重载解析就会影响到lambda表达式的目标类型，编译器会通过它所获得的信息来做出决定。如果lambda表达式具有显式类型（参数类型被显式指定），则编译器不仅知道lambda表达式的参数类型，还知道表达式中所有的返回值类型；如果lambda表达式具有隐式类型（参数类型由推导得知），则重载解析会忽略lambda表达式的函数体，而只使用表达式的参数个数。

如果在解析方法声明时存在二义性，类型转换或者显式的lambda可以为编译器提供额外的类型信息来消除二义性。如果lambda表达式的返回类型依赖于类型参数推导，则lambda的函数体可以向编译器提供信息来帮助推导出类型参数。

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

在上面的代码中，ps的类型是List<Person>，因此ps.stream()的返回类型是Stream<Person>。map()方法接收一个类型为Function<T, R>的函数式接口，这里T的类型是Stream元素类型（此时，已经知道T是Person类型），而R的类型未知，需要推导：通过对lambda表达式的函数体进行类型检查，发现函数体返回String，因此R的类型是String，所以map()返回Steam<String>。绝大多数情况下，编译器都能解析出正确的参数类型，但如果碰到无法解析的情况，则需要：

* 使用显式类型的lambda表达式（为参数p提供显式类型）以提供额外的类型信息
* 把lambda表达式强转为Function<Person, String>
* 为泛型参数R提供一个显式类型（.<String>map(p -> p.getName())）

lambda表达式本身也可以为它自己的函数题提供目标类型，这意味着我们可以很方便的编写一个返回函数的函数：

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

类似的，条件表达式可以把目标类型分发给子表达式：

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

最后，类型转换表达式可以显式提供lambda表达式的类型，这个特性在无法确定目标类型时非常有用：

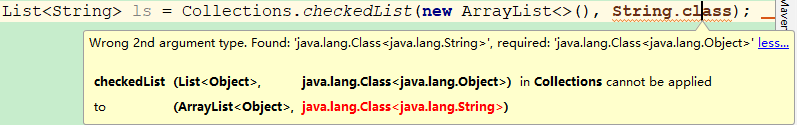
|  |
| --- |
| // Object o = () -> { System.out.println("hi"); }; 这段代码是非法的  Object o = (Runnable) () -> { System.out.println("hi"); }; |

除此之外，当重载的方法都拥有函数式接口时，类型转换可以帮助解析重载解析时出现的二义性。

目标类型这个概念不仅仅适用于lambda表达式，泛型方法调用和“菱形”构造方法调用也可以从目标类型中受益，例如下面的代码在Java SE 7中是非法的，但在Java SE 8中是合法的。

|  |
| --- |
| List<String> ls = Collections.checkedList(new ArrayList<>(), String.class);  Set<Integer> si = flag ? Collections.singleton(23) : Collections.emptySet(); |

以上代码在Java SE 7中会报错：



# 作用域

在内部类中使用变量名（以及this）非常容易出错，内部类中继承得到的成员（包括继承自Object类的方法）可能会把外部类的成员掩盖，另外，未限定的this引用总是指向内部类自己而非指向外部类。

相对于内部类，lambda表达式的语义就十分简单：它不会从超类中继承任何变量名，也不会引入一个新的作用域。lambda表达式基于词法作用域，也就是说lambda表达式函数体中的变量和它外部环境的变量具有相同的语义（也包括lambda表达式的形参）。此外，this关键字及其引用在lambda表达式内部和外部也拥有相同的语义。

为了进一步说明词法作用域的优点，请参考下面的代码，它会把"Hello, world!"打印两遍：

|  |
| --- |
| 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();  }  } |

与之相类似的内部类实现则会打印出类似Hello$1@5b89a773和Hello$2@537a7706之类的字符串，这往往会使开发者大吃一惊。

# 变量捕获

在Java SE 7中，编译器对内部类中引用的外部变量（即捕获的变量）要求非常严格：如果捕获的变量没有被声明为final就会产生一个编译错误。在Java SE 8中放宽了这个限制：对于lambda表达式和内部类，允许在其中捕获有效只读（effectively final）的局部变量。

简单来说，如果一个局部变量在初始化后从未被修改过，那么它就符合有效只读的要求。换句话说，加上final后也不会导致编译错误的局部变量就是有效只读变量。

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

对this的引用，包括隐式引用this，比如引用未限定字段或者调用未限定方法，本质上都是引用一个final局部变量，包含此类引用的lambda表达式捕获了this实例。对于其他情况，lambda对象不会保留任何对this的引用。

这个特性对内存管理来说是一件好事：内部类实例会一直保留一个对其外部类实例的强引用，而那些没有捕获外部类成员的lambda表达式则不会保留对外部类实例的引用。要知道，内部类保留外部类实例的强引用这一特性往往会造成内存泄漏。

尽管放宽了对捕获变量的语法限制，但试图修改捕获变量的行为仍然会被禁止，比如下面这个例子就是非法的：

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

为什么要禁止这种行为呢？因为这样的lambda表达式很容易引起race condition。除非能够强制（最好是在编译时）这样的函数不能离开当前线程，但如果这么做了，可能会导致更多的问题。简而言之，lambda表达式对值封闭，对变量开放，下面的例子更好的说明了这一特性：

|  |
| --- |
| int sum = 0;  list.forEach(e -> { sum += e.size(); }); // Illegal, close over values  List<Integer> aList = new List<>();  list.forEach(e -> { aList.add(e); }); // Legal, open over variables |

lambda表达式不支持修改捕获变量的另一个原因是可以使用更好的方式来实现同样的效果：使用归约（reduction）。java.util.stream包提供了各种通用的和专用的归约操作（例如sum、min和max），就上面的例子而言，可以使用归约操作（在串行和并行下都是安全的）来代替forEach：

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

sum()方法等价于下面的归约操作：

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

归约需要一个初始值（以防输入为空）和一个操作符（在这里是加号），然后用下面的表达式计算结果：

|  |
| --- |
| 0 + list[0] + list[1] + list[2] + ... |

归约也可以完成其他操作，比如求最小值、最大值和乘积等等。如果操作符具有可结合性，那么归约操作就可以容易的被并行化。所以，与其支持一个本质上是串行且容易导致race condition的操作，还不如在库中提供一个更加并行友好且不容易出错的方式来进行累积。

# 方法引用

lambda表达式允许我们定义一个匿名方法，并将视为一个函数式接口来使用。Java SE 8允许我们使用方法引用在已有方法上实现相同的特性。

方法引用和lambda表达式拥有相同的特性，比如它们都需要一个目标类型，并需要被转换为函数式接口的实例，不过我们并不需要为方法引用提供方法体，而是可以直接通过方法名称引用已有方法。

以下面的代码为例，假设我们要按照name或age为Person数组进行排序：

|  |
| --- |
| 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); |

我们可以使用方法引用代替lambda表达式来重写以上代码：

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

这里的Person::getName可以被看作为lambda表达式的简写形式。尽管方法引用不一定会把语法变得更紧凑，但它拥有更明确的语义——如果我们想要调用的方法拥有一个名字，我们就可以通过它的名称直接调用它。

# 方法引用的种类

方法引用有很多种，它们的语法如下：

静态方法引用：ClassName:: methName

对象的实例方法引用：instanceRef::methName

对象的超类方法引用：super::methName

类的任意对象上的实例方法引用：ClassName::methName

构造方法引用：Class::new

数组构造方法引用：TypeName[]::new

* 对于静态方法引用，需要在类名和方法名之间加上::分隔符，例如Integer::sum。
* 对于对象的实例方法引用，需要在对象名和方法名之间加上分隔符：

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

这里，隐式lambda表达式会从knownNames中捕获到String对象，然后表达式的函数体会使用该String对象作为接收者调用Set.contains()方法。

通过实例方法引用，可以很方便的在不同函数式接口之间进行类型转换：

|  |
| --- |
| Callable<Path> c = ...  Privileged<Path> a = c::call; |

* 对于类的任意对象上的实例方法引用，需要在类名和方法名之间加上分隔符：

|  |
| --- |
| Function<String, String> upperfier = String::toUpperCase; |

这里，隐式lambda表达式有一个String参数，该参数会被toUppperCase方法使用。

如果类是泛型的，则需要在分隔符前提供类型参数，或者不提供——大多数情况下，可以由目标类型推导出类型参数。

需要注意的是，静态方法引用和类型的任意对象上的实例方法引用拥有一样的语法，编译器会根据实际情况做出决定。

一般我们不需要指定方法引用中的参数类型，因为编译器通常可以推导出结果，但我们也可以显式的在分隔符之前提供参数类型。

* 对于构造方法引用，需要在类名和new之间加上分隔符：

|  |
| --- |
| SocketImplFactory factory = MySocketImpl::new; |

如果一个类拥有多个构造函数，则目标类型的方法签名会被用来最佳匹配出构造函数，这个过程和调用构造方式的选择过程是一样的。

如果待实例化的类是泛型类，则可以在类名后提供类型参数，否则，编译器会按照“菱形”构造方法调用时的方式进行推导。

* 数组构造方法引用的语法比较特殊，数组会被视为拥有一个接收int类型参数的构造函数，如下：

|  |
| --- |
| IntFunction<int[]> arrayMaker = int[]::new;  int[] array = arrayMaker.apply(10) // 创建数组 int[10] |

# 默认方法和静态接口方法

lambda表达式和方法引用大大提升了Java的表达能力，不过为了使“代码即数据”变得更加容易，需要把这些特性融入到已有库中，以便开发者使用。

Java SE 7中为一个已有的类库增加功能是非常困难的。具体来说，接口在发布之后就已经被定性，除非我们能够一次性更新该接口的实现，否则向接口添加方法就会破坏现有的接口实现。默认方法的目标即是解决这一问题，使得接口在发布之后仍能被逐步演化。

举个例子，标准库中的Collection API显然应该提供新的lambda友好的操作，比如removeAll方法应该被泛化为接收一个函数式接口Predicate，但这个新的方法应该被放到哪里呢？我们无法直接在Collection接口上新增方法，那样会破坏现有的Collection实现；我们倒是可以在Collections工具类中增加对应的静态方法，但这样会把这个方法置于“二等公民”的境地。

默认方法提供了一种更加面向对象的方式来向接口中增加新的行为。默认方法是一种新的方法：接口方法可以是抽象的或者默认的。默认方法拥有其默认实现，实现接口的类通过继承得到该默认实现（如果该类没有覆盖该默认实现），此外，默认方法不是抽象方法，所以我们可以放心的向函数式接口中增加默认方法，而不用担心函数式接口的单抽象方法限制。

下面的例子展示了如何向Iterator接口增加默认方法skip：

|  |
| --- |
| interface Iterator<E> {  boolean hasNext();  E next();  void remove();  default void skip(int i) {  for ( ; i > 0 && hasNext(); i -= 1) next();  }  } |

根据上面的Iterator定义，所有实现Iterator的类型都会自动继承skip方法。在使用者眼中，skip不过是接口新增的一个虚拟方法。在没有覆盖skip方法的Iterator子类实例上调用skip方法会执行其默认实现：调用hasNext和next若干次。子类可以覆盖skip来提供更好的实现，比如直接移动游标，或是为操作提供原子性。

除了默认方法，Java SE 8还允许在接口中定义静态方法，这使得我们可以从接口中直接调用和它相关的辅助方法，而不是从其他辅助类中调用（辅助类往往以对应接口的复数命名，比如Collections）。比如，我们一般需要使用静态辅助方法生成实现Comparator的比较器，在Java SE 8中我们可以直接把静态方法定义在Comparator接口中：

|  |
| --- |
| 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));  } |

# 继承默认方法

和其他方法一样，默认方法也可以被继承，大多数情况下这种继承行为和我们所期待的一致。不过，当类或者接口的超类拥有多个具有相同签名的方法时，就需要一套规则来解决冲突：

* 类的方法优先级最高
* 如果无法根据第一条规则进行判断，则子接口的优先级更高，例如B继承了A，则B的优先级比A高
* 如果还是无法判断，则必须显式覆盖方法

**interface** A{  
 **default void** hello(){  
 System. ***out***.println(**"Hello from A"** );  
 }  
}  
**interface** B **extends** A{  
 **default void** hello(){  
 System. ***out***.println(**"Hello from B"** );  
 }  
}  
**class** C **implements** A,B{  
  
}

上面的代码中，接口B继承了接口A，类C实现了接口A和B。因为接口A和B中都有默认方法hello，根据第二条规则，B的优先级更高。

当两个默认方法相冲突时，会产生编译错误，此时就需要显式覆盖超类方法。一般来说，我们会定一个默认方法，然后再其中显式选择超类方法：

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

1. **融会贯通**

在设计lambda时的一个重要目标就是使得新增的语言特性能够和库特性无缝结合，下面通过一个实际例子（按照姓名排序）来演示这一点：

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

上面的代码虽然能够实现功能，但冗余代码太多。有了lambda表达式，我们可以去掉冗余的匿名类：

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

尽管代码简洁了很多，但它的抽象依然很差：开发者仍然需要进行实际的比较操作。我们可以借助Comparator里的comparing方法实现比较操作：

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

在类型推导和静态导入的帮助下，我们可以进一步简化上面的代码：

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

注意到这里的lambda表达式实际上是一个Person::getLastName方法引用，因此可以用方法引用代替它：

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

最后，使用Collections.sort这样的辅助方法并不是一个好主意：它不但使代码变的冗余，也无法为实现List接口的数据结构提供特定的高效实现，而且，由于Collectoins.sort方法不属于List接口，用户在阅读List接口的文档时不会察觉在Collections类中还有一个针对List接口的排序方法。

默认方法可以有效的解决这个问题，Java SE 8中为List接口增加了默认方法sort()，于是我们可以这样调用：

|  |
| --- |
| people.sort(comparing(Person::getLastName)); |

1. **小结**

Java SE 8提供的新语言特性并不算多——lambda表达式，方法引用，默认方法和静态接口方法，以及范围更广的类型推导。但是把它们结合在一起之后，开发者可以编写出更加清晰简洁的代码，类库编写者可以编写更加强大易用的并行类库。