链滴

《Java8 实战》- 第九章笔记（默认方法）

作者：Not-Found

原文链接：https://ld246.com/article/1541859058250

来源网站：链滴

默认方法

传统上，Java程序的接口是将相关方法按照约定组合到一起的方式。实现接口的类必须为接口中定义每个方法提供一个实现，或者从父类中继承它的实现。但是，一旦类库的设计者需要更新接口，向其加入新的方法，这种方式就会出现问题。现实情况是，现存的实体类往往不在接口设计者的控制范围内，这些实体类为了适配新的接口约定也需要进行修改。由于Java 8的API在现存的接口上引入了非多的新方法，这种变化带来的问题也愈加严重，一个例子就是前几章中使用过的 List 接口上的 sort 法。想象一下其他备选集合框架的维护人员会多么抓狂吧，像Guava和Apache Commons这样的框现在都需要修改实现了 List 接口的所有类，为其添加sort 方法的实现。

且慢，其实你不必惊慌。Java 8为了解决这一问题引入了一种新的机制。Java 8中的接口现在支持在明方法的同时提供实现，这听起来让人惊讶！通过两种方式可以完成这种操作。其一，Java 8允许在口内声明静态方法。其二，Java 8引入了一个新功能，叫默认方法，通过默认方法你可以指定接口方的默认实现。换句话说，接口能提供方法的具体实现。因此，实现接口的类如果不显式地提供该方法具体实现，就会自动继承默认的实现。这种机制可以使你平滑地进行接口的优化和演进。实际上，到前为止你已经使用了多个默认方法。两个例子就是你前面已经见过的 List 接口中的 sort ，以及 Collec ion 接口中的 stream 。

第1章中我们看到的 List 接口中的 sort 方法是Java 8中全新的方法，它的定义如下：

```
default void sort(Comparator<? super E> c){
    Collections.sort(this, c);
}
```

请注意返回类型之前的新 default 修饰符。通过它，我们能够知道一个方法是否为默认方法。这里 sor 方法调用了 Collections.sort 方法进行排序操作。由于有了这个新的方法，我们现在可以直接通过调 sort ，对列表中的元素进行排序。

```
List<Integer> numbers = Arrays.asList(3, 5, 1, 2, 6);
numbers.sort(Comparator.naturalOrder());
```

不过除此之外， 这段代码中还有些其他的新东西。注意到了吗，我们调用了Comparator.naturalOrd r 方法。这是 Comparator 接口的一个全新的静态方法，它返回一个Comparator 对象，并按自然序对其中的元素进行排序（即标准的字母数字方式排序）。

第4章中你看到的 Collection 中的 stream 方法的定义如下：

```
default Stream<E> stream() {
    return StreamSupport.stream(spliterator(), false);
}
```

我们在之前的几章中大量使用了该方法来处理集合，这里 stream 方法中调用了SteamSupport.strea 方法来返回一个流。你注意到 stream 方法的主体是如何调用 spliterator 方法的了吗？它也是 Collec ion 接口的一个默认方法。

喔噢！这些接口现在看起来像抽象类了吧？是，也不是。它们有一些本质的区别，我们在这一章中会对性地进行讨论。但更重要的是，你为什么要在乎默认方法？默认方法的主要目标用户是类库的设计啊。

简而言之，向接口添加方法是诸多问题的罪恶之源；一旦接口发生变化，实现这些接口的类往往也需更新，提供新添方法的实现才能适配接口的变化。如果你对接口以及它所有相关的实现有完全的控制这可能不是个大问题。但是这种情况是极少的。这就是引入默认方法的目的：它让类可以自动地继承口的一个默认实现。

因此，如果你是个类库的设计者，这一章的内容对你而言会十分重要，因为默认方法为接口的演进提了一种平滑的方式，你的改动将不会导致已有代码的修改。此外，正如我们后文会介绍的，默认方法方法的多继承提供了一种更灵活的机制，可以帮助你更好地规划你的代码结构：类可以从多个接口继默认方法。因此，即使你并非类库的设计者，也能在其中发现感兴趣的东西。

章的结构如下。首先，我们会跟你一起剖析一个API演化的用例，探讨由此引发的各种问题。紧接着们会解释什么是默认方法，以及它们在这个用例中如何解决相应的问题。之后，我们会展示如何创建己的默认方法，构造Java语言中的多继承。最后，我们会讨论一个类在使用一个签名同时继承多个默方法时，Java编译器是如何解决可能的二义性（模糊性）问题的。

不断演进的 API

为了理解为什么一旦API发布之后，它的演进就变得非常困难，我们假设你是一个流行Java绘图库的计者（为了说明本节的内容，我们做了这样的假想）。你的库中包含了一个 Resizable接口，它定义一个简单的可缩放形状必须支持的很多方法， 比如： setHeight 、 setWidth 、getHeight 、 getWith 以及 setAbsoluteSize 。此外，你还提供了几个额外的实现（out-of-boximplementation），如方形、长方形。由于你的库非常流行，你的一些用户使用 Resizable 接口创建了他们自己感兴趣的实，比如椭圆。

发布API几个月之后，你突然意识到 Resizable 接口遗漏了一些功能。比如，如果接口提供一个 setReativeSize 方法，可以接受参数实现对形状的大小进行调整，那么接口的易用性会更好。你会说这看来很容易啊：为 Resizable 接口添加 setRelativeSize 方法，再更新 Square和 Rectangle 的实现就了。不过，事情并非如此简单！你要考虑已经使用了你接口的用户，他们已经按照自身的需求实现了esizable 接口，他们该如何应对这样的变更呢？非常不幸，你无法访问，也无法改动他们实现了 Resiable 接口的类。这也是Java库的设计者需要改进Java API时面对的问题。让我们以一个具体的实例为，深入探讨修改一个已发布接口的种种后果。

初始版本的 API

Resizable 接口的最初版本提供了下面这些方法：

```
public interface Drawable {
    void draw();
}

public interface Resizable extends Drawable {
    int getWidth();

    void setWidth(int width);

    int getHeight();

    void setHeight(int height);

    void setAbsoluteSize(int width, int height);
}
```

用户实现

你的一位铁杆用户根据自身的需求实现了 Resizable 接口，创建了 Ellipse 类：

```
public class Ellipse implements Resizable {
    ...
}
```

他实现了一个处理各种 Resizable 形状（包括 Ellipse ）的游戏：

```
public class Square implements Resizable {
    ...
}
public class Triangle implements Resizable {
    ...
}
public class Game {
    public static void main(String[] args) {
        List<Resizable> resizableShapes =
                Arrays.asList(new Square(), new Triangle(), new Ellipse());
        Utils.paint(resizableShapes);
    }
}
public class Utils {
    public static void paint(List<Resizable> list) {
        list.forEach(r -> {
            r.setAbsoluteSize(42, 42);
            r.draw();
        });
    }
}
```

第二版 API

库上线使用几个月之后，你收到很多请求，要求你更新 Resizable 的实现，让 Square Triangle 以及他的形状都能支持 setRelativeSize 方法。为了满足这些新的需求，你发布了第二版API。

```
public interface Resizable extends Drawable {
    int getWidth();

    void setWidth(int width);

    int getHeight();

    void setHeight(int height);

    void setAbsoluteSize(int width, int height);

    void setRelativeSize(int wFactor, int hFactor);
}
```

用户面临的窘境

对 Resizable 接口的更新导致了一系列的问题。首先，接口现在要求它所有的实现类添加setRelativeSize 方法的实现。但是用户最初实现的 Ellipse 类并未包含 setRelativeSize方法。向接口添加新方法是进制兼容的，这意味着如果不重新编译该类，即使不实现新的方法，现有类的实现依旧可以运行。不，用户可能修改他的游戏，在他的 Utils.paint 方法中调用setRelativeSize 方法，因为 paint 方法接一个 Resizable 对象列表作为参数。如果传递的是一个 Ellipse 对象，程序就会抛出一个运行时错误因为它并未实现 setRelativeSize 方法：

```
Exception in thread "main" java.lang.AbstractMethodError:lambdasinaction.chap9.Ellipse.setRelativeSize(II)V
```

其次，如果用户试图重新编译整个应用（包括 Ellipse 类），他会遭遇下面的编译错误：

```
Error:(9, 8) java: xin.coderream.java8.chap9.Ellipse不是抽象的, 并且未覆盖
xin.coderream.java8.chap9.Resizable中的抽象方法setRelativeSize(int,int)
```

最后，更新已发布API会导致后向兼容性问题。这就是为什么对现存API的演进，比如官方发布的Java ollection API，会给用户带来麻烦。当然，还有其他方式能够实现对API的改进，但是都不是明智的择。比如，你可以为你的API创建不同的发布版本，同时维护老版本和新版本，但这是非常费时费力，原因如下。其一，这增加了你作为类库的设计者维护类库的复杂度。其次，类库的用户不得不同时用一套代码的两个版本，而这会增大内存的消耗，延长程序的载入时间，因为这种方式下项目使用的文件数量更多了。

这就是默认方法试图解决的问题。它让类库的设计者放心地改进应用程序接口，无需担忧对遗留代码影响，这是因为实现更新接口的类现在会自动继承一个默认的方法实现。

概述默认方法

经过前述的介绍，我们已经了解了向已发布的API添加方法，对现存代码实现会造成多大的损害。默方法是Java 8中引入的一个新特性，希望能借此以兼容的方式改进API。现在，接口包含的方法签名它的实现类中也可以不提供实现。那么，谁来具体实现这些方法呢？实际上，缺失的方法实现会作为口的一部分由实现类继承（所以命名为默认实现），而无需由实现类提供。

那么，我们该如何辨识哪些是默认方法呢？其实非常简单。默认方法由 default 修饰符修饰，并像类声明的其他方法一样包含方法体。比如，你可以像下面这样在集合库中定义一个名为Sized 的接口，其中定义一个抽象方法 size ，以及一个默认方法 isEmpty ：

```
public interface Sized {
    int size();

    default boolean isEmpty() {
        return size() == 0;
    }
}
```

太棒了！这样任何一个实现了 Sized 接口的类都会自动继承 isEmpty 的实现。因此，向提供了默认现的接口添加方法就不是源码兼容的。

现在，我们回顾一下最初的例子，那个Java画图类库和你的游戏程序。具体来说，为了以兼容的方式进这个库（即使用该库的用户不需要修改他们实现了 Resizable 的类），可以使用默认方法，提供 seRelativeSize 的默认实现：

```
default void setRelativeSize(int wFactor, int hFactor){
    setAbsoluteSize(getWidth() / wFactor, getHeight() / hFactor);
}
```

由于接口现在可以提供带实现的方法，是否这意味着Java已经在某种程度上实现了多继承？如果实现也实现了同样的方法，这时会发生什么情况？默认方法会被覆盖吗？现在暂时无需担心这些，Java 8 已经定义了一些规则和机制来处理这些问题。

你可能已经猜到，默认方法在Java 8的API中已经大量地使用了。本章已经介绍过我们前一章中大量用的 Collection 接口的 stream 方法就是默认方法。 List 接口的 sort 方法也是默认方法。第3章介的很多函数式接口，比如 Predicate 、 Function 以及 Comparator 也引入了新的默认方法，比如 Predicate.and 或者 Function.andThen （记住，函数式接口只包含一个抽象方法，默认方法是种非抽方法）。

默认方法的使用模式

现在你已经了解了默认方法怎样以兼容的方式演进库函数了。除了这种用例，还有其他场景也能利用个新特性吗？当然有，你可以创建自己的接口，并为其提供默认方法。这一节中，我们会介绍使用默方法的两种用例：可选方法和行为的多继承。

可选方法

你很可能也碰到过这种情况，类实现了接口，不过却刻意地将一些方法的实现留白。我们以Iterator口为例来说。 Iterator 接口定义了 hasNext 、 next ，还定义了 remove 方法。Java 8之前，由于户通常不会使用该方法， remove 方法常被忽略。因此，实现 Interator 接口的类通常会为 remove 法放置一个空的实现，这些都是些毫无用处的模板代码。

采用默认方法之后，你可以为这种类型的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显式地提供一个空方法。比如，在Java 8中， Iterator 接口就为 remove 方法提供了一个默认实现，下所示：

```
public interface Iterator<E> {
    ...
    default void remove() {
        throw new UnsupportedOperationException("remove");
    }
    ...
}
```

通过这种方式，你可以减少无效的模板代码。实现 Iterator 接口的每一个类都不需要再声明一个空的emove 方法了，因为它现在已经有一个默认的实现。

行为的多继承

默认方法让之前无法想象的事儿以一种优雅的方式得以实现，即行为的多继承。这是一种让类从多个源重用代码的能力。

Java的类只能继承单一的类，但是一个类可以实现多接口。要确认也很简单，下面是Java API中对 ArrayList 类的定义：

```
public class ArrayList<E> extends AbstractList<E>
        implements List<E>, RandomAccess, Cloneable, java.io.Serializable {
}
```

1. 类型的多继承

这个例子中 ArrayList 继承了一个类，实现了六个接口。因此 ArrayList 实际是七个类型的直接子类分别是： AbstractList 、 List 、 RandomAccess 、 Cloneable 、 Serializable 、Iterable 和 Collecton 。所以，在某种程度上，我们早就有了类型的多继承。

由于Java 8中接口方法可以包含实现，类可以从多个接口中继承它们的行为（即实现的代码）。让我从一个例子入手，看看如何充分利用这种能力来为我们服务。保持接口的精致性和正交性能帮助你在有的代码基上最大程度地实现代码复用和行为组合。

2. 利用正交方法的精简接口

假设你需要为你正在创建的游戏定义多个具有不同特质的形状。有的形状需要调整大小，但是不需要

旋转的功能；有的需要能旋转和移动，但是不需要调整大小。这种情况下，你怎么设计才能尽可能地用代码?

你可以定义一个单独的 Rotatable 接口，并提供两个抽象方法 setRotationAngle 和getRotationAnge ，如下所示：

```
public interface Rotatable {
    int getRotationAngle();

    void setRotationAngle(int angleInDegrees);

    default void rotateBy(int angleInDegrees) {
        setRotationAngle((getRotationAngle() + angleInDegrees) % 360);
    }
}
```

这种方式和模板设计模式有些相似，都是以其他方法需要实现的方法定义好框架算法。

现在，实现了 Rotatable 的所有类都需要提供 setRotationAngle 和 getRotationAngle的实现，但此同时它们也会天然地继承 rotateBy 的默认实现。

类似地，你可以定义之前看到的两个接口 Moveable 和 Resizable 。它们都包含了默认实现。下面是 Moveable 的代码：

```
public interface Moveable {
    int getX();

    void setX(int x);

    int getY();

    void setY(int y);

    default void moveHorizontally(int distance) {
        setX(getX() + distance);
    }

    default void moveVertically(int distance) {
        setY(getY() + distance);
    }
}
```

下面是 Resizable 的代码：

```
public interface Resizable extends Drawable {
    int getWidth();

    void setWidth(int width);

    int getHeight();

    void setHeight(int height);

    void setAbsoluteSize(int width, int height);
```

```
    default void setRelativeSize(int wFactor, int hFactor){
        setAbsoluteSize(getWidth() / wFactor, getHeight() / hFactor);
    }
}
```

3. 组合接口

通过组合这些接口，你现在可以为你的游戏创建不同的实体类。比如， Monster 可以移动、旋转和放。

```
public class Monster implements Rotatable, Moveable, Resizable {
    ...
}
```

Monster 类会自动继承 Rotatable 、 Moveable 和 Resizable 接口的默认方法。这个例子中，Monster 继承了 rotateBy 、 moveHorizontally 、 moveVertically 和 setRelativeSize 的实现。

你现在可以直接调用不同的方法：

```
Monster m = new Monster();
m.rotateBy(180);
m.moveVertically(10);
```

像你的游戏代码那样使用默认实现来定义简单的接口还有另一个好处。假设你需要修改moveVertically的实现，让它更高效地运行。你可以在 Moveable 接口内直接修改它的实现，所有实现该接口的类会动继承新的代码（这里我们假设用户并未定义自己的方法实现）。

通过前面的介绍，你已经了解了默认方法多种强大的使用模式。不过也可能还有一些疑惑：如果一个同时实现了两个接口，这两个接口恰巧又提供了同样的默认方法签名，这时会发生什么情况？类会选使用哪一个方法？这些问题，我们会在接下来的一节进行讨论。

解决冲突的规则

我们知道Java语言中一个类只能继承一个父类，但是一个类可以实现多个接口。随着默认方法在Java中引入，有可能出现一个类继承了多个方法而它们使用的却是同样的函数签名。这种情况下，类会选使用哪一个函数？在实际情况中，像这样的冲突可能极少发生，但是一旦发生这样的状况，必须要有套规则来确定按照什么样的约定处理这些冲突。这一节中，我们会介绍Java编译器如何解决这种潜在冲突。我们试图回答像“接下来的代码中，哪一个 hello 方法是被 C 类调用的”这样的问题。注意，下来的例子主要用于说明容易出问题的场景，并不表示这些场景在实际开发过程中会经常发生。

```
public interface A {
    default void hello() {
        System.out.println("Hello from A");
    }
}
public interface B extends A {
    default void hello() {
        System.out.println("Hello from B");
    }
}
public class C implements A, B {
    public static void main(String[] args) {
        // 猜猜打印的是什么?
        new C().hello();
```

```
    }
}
```

此外，你可能早就对C++语言中著名的菱形继承问题有所了解，菱形继承问题中一个类同时继承了具相同函数签名的两个方法。到底该选择哪一个实现呢？ Java 8也提供了解决这个问题的方案。请接着读下面的内容。

解决问题的三条规则

如果一个类使用相同的函数签名从多个地方（比如另一个类或接口）继承了方法，通过三条规则可以行判断。

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，显式地选择用哪一个默认方法的实现。

是的，就是这三条准则就是你需要知道的全部了！

运行结果

让我们回顾一下开头的例子，这个例子中 C 类同时实现了 B 接口和 A 接口，而这两个接口恰巧又都义了名为 hello 的默认方法。

编译器会使用声明的哪一个 hello 方法呢？其实上面的代码是编译不通过的，按照规则(2)，应该选择是提供了最具体实现的默认方法的接口。但，在C中不知道谁比谁更具体，所以需要显示的指定调用个接口的方法：

```
public class C implements A, B {
    public static void main(String[] args) {
        new C().hello();
    }

    @Override
    public void hello() {
      A.super.hello();
    }

    OR

    @Override
    public void hello() {
      B.super.hello();
    }

    OR

    @Override
    public void hello() {
      System.out.println("Hello from C!");
    }
```

```
}
```

比如：调用 A.super.Hello()，那么打印的是 Hello form A!，调用 B.super.Hello() 那么输出的是 Hell from B!。

如果，你碰到类似的问题，以上的三条准则将可以帮助你解决这个问题！

小结

1. Java 8中的接口可以通过默认方法和静态方法提供方法的代码实现。
2. 默认方法的开头以关键字 default 修饰，方法体与常规的类方法相同。
3. 向发布的接口添加抽象方法不是源码兼容的。
4. 默认方法的出现能帮助库的设计者以后向兼容的方式演进API。
5. 默认方法可以用于创建可选方法和行为的多继承。
6. 我们有办法解决由于一个类从多个接口中继承了拥有相同函数签名的方法而导致的冲突。
7. 类或者父类中声明的方法的优先级高于任何默认方法。如果前一条无法解决冲突，那就选择同函数名的方法中实现得最具体的那个接口的方法。
8. 两个默认方法都同样具体时，你需要在类中覆盖该方法，显式地选择使用哪个接口中提供的默认方法。

代码

Github:chap9

Gitee:chap9