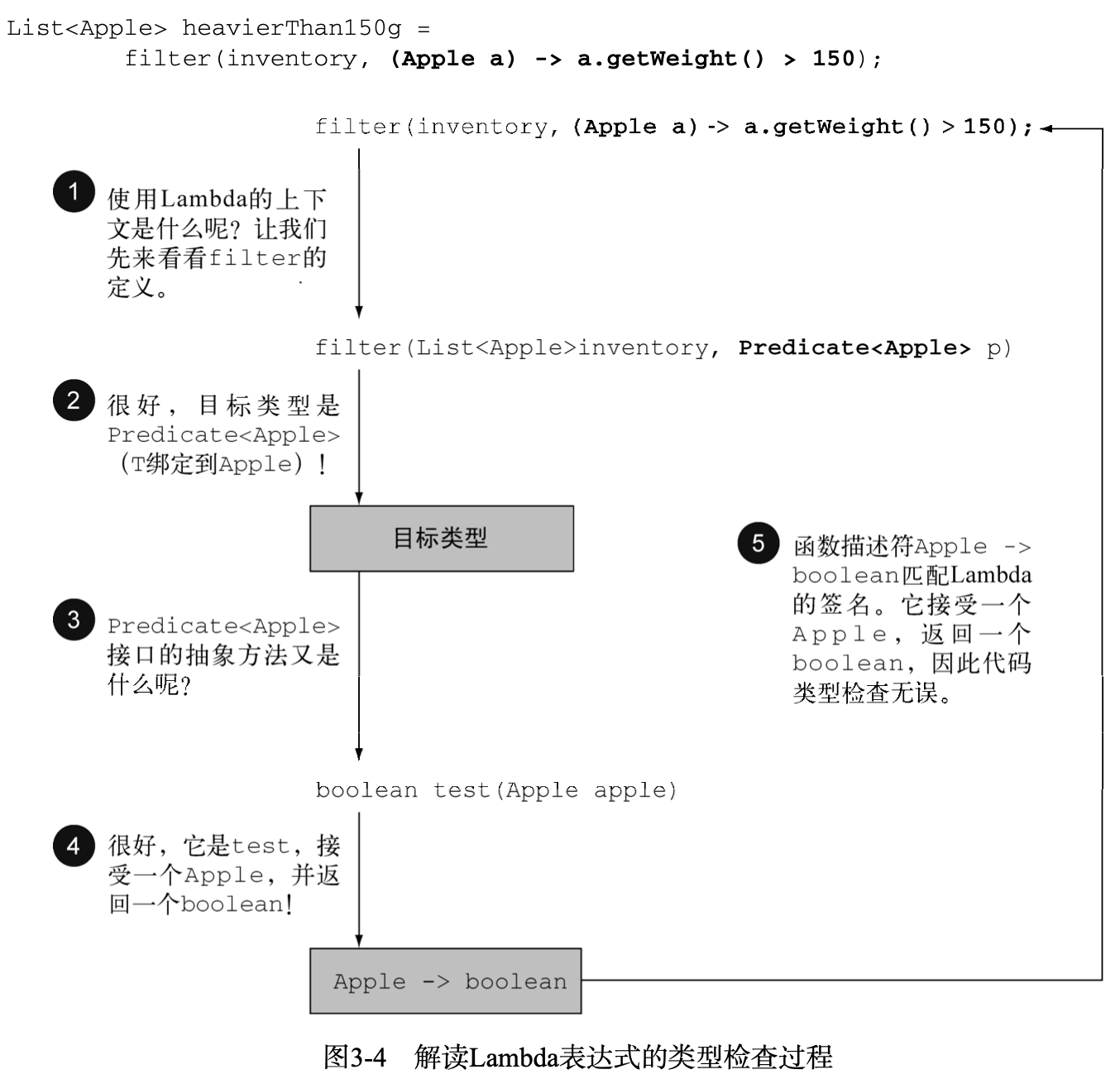
Lambda表达式可以为函数式接口生成一个实例。然而，Lambda 表达式本身并不包含它在实现哪个函数式接口的信息。为了全面了解Lambda表达式，你应该知道Lambda的实际类型是什么。

#### 1、类型检查

Lambda的类型是从使用Lambda的上下文推断出来的。上下文（比如，接受它传递的方法的参数，或接受它的值的局部变量）中Lambda表达式需要的类型称为目标类型。让我们通过一个例子，看看当你使用Lambda表达式时背后发生了什么。下图概述了类型检查过程。



类型检查过程可以分解如下所示。

* 首先，你要找出filter方法的声明。
* 第二，要求它是Predicate<Apple>（目标类型）对象的第二个正式参数。
* 第三，Predicate<Apple>是一个函数式接口，定义了一个叫做test的抽象方法。
* 第四，test方法描述了一个函数描述符，它可以接受一个Apple，并返回一个boolean。
* 最后，filter的任何实际参数都必须匹配这个要求。

这段代码是有效的，因为我们所传递的Lambda表达式也同样接受Apple为参数，并返回一个boolean。请注意，如果Lambda表达式抛出一个异常，那么抽象方法所声明的throws与也必须与之匹配。

#### 2、同样的Lambda，不同的函数式接口

有了目标类型的概念，同一个Lambda表达式就可以与不同的函数式接口联系起来，只要它们的抽象方法签名能够兼容。比如，签名提到的Callable和PrivilegedAction，这两个接口都代表着什么也不接受且返回一个泛型T的函数。因此，下面两个赋值是有效的：

Callable<Integer> c = () -> 42;

PrivilegedAction<Integer> p = () -> 42;

这里，第一个赋值的目标类型是Callable<Integer>，第二个赋值的目标类型是PrivilegedAction<Integer>。

同一个Lambda可用于多个不同的函数式接口：

Comparator<Apple> c1 =

(Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight());

ToIntBiFunction<Apple, Apple> c2 =

(Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight());

BiFunction<Apple, Apple, Integer> c3 =

(Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight());

**菱形运算符**

Java 7中已经引入了菱形运算符<>，利用泛型推断从上下文推断类型的思想（这一思想甚至可以追溯到更早的泛型方法）。一个类实例表达式可以出现在两个或更多不同的上下文中，并会像下面这样推断出适当的类型参数：

List<String> listOfStrings = new ArrayList<>();

List<Integer> listOfIntegers = new ArrayList<>();

**特殊的void兼容规则**

如果一个Lambda的主体是一个语句表达式，它就和一个返回void的函数描述符兼容（当然需要参数列表也兼容）。例如，以下两行都是合法的，尽管List的add方法返回了一个boolean，而不是Consumer上下文（T -> void）所要求的void：

// Predicate返回了一个boolean

Predicate<String> p = s -> list.add(s);

// Consumer返回了一个void

Consumer<String> b = s -> list.add(s);

到目前为止，你应该能够很好地理解在什么时候以及在哪里可以使用Lambda表达式了。它们可以从赋值的上下文、方法调用的上下文（参数和返回值），以及类型转换的上下文中获得目标类型。为了检验你的掌握情况：

类型检查——为什么下面的代码不能编译呢？

Object o = () -> (System.out.println(“Tricky example”););

Lambda表达式的上下文是Object（目标类型）。但Object不是一个函数式接口。为了解决这个问题，可以把目标类型改成Runnable，它的函数描述符是

() -> void

Runnable R = () -> (System.out.println(“Tricky example”););

你已经见过如何利用目标类型来检查一个Lambda是否可以用于某个特定的上下文。其实，它也可以用来做一些略有不同的事：推断Lambda参数的类型。

#### 3、类型推断

还可以进一步简化代码。Java编译器会从上下文（目标类型）推断出用什么函数式接口来配合Lambda表达式，这意味着它也可以推断出适合Lambda的签名，因为函数描述符可以通过目标类型来得到。这样做的好处在于，编译器可以了解Lambda表达式的参数类型，这样就可以在Lambda语法中省去标注参数类型。换句话说，Java编译器会像下面这样推断Lambda的参数类型：