> 本文由 简悦 SimpRead 转码， 原文地址 www.imooc.com

最近有一个朋友询问一个《Effective Java》 中的 “诡异问题”，表示看不懂里面的讲解。

本节将以该问题为素材，灵活运用本专栏所介绍的各种方法来研究这个问题。

《Effective Java》 第二版，第 16 条： 复合优于继承小节提到：

> 为了调优该程序的性能，需要查询 `HashSet` ，看看自从被创建以来添加了多少个元素。为了提供这种功能，我们得用一个 `addCount` 变量记录插入的元素数量，并提供一个获取该变量数值的方法。
>
> HashSet 类包含两个可以增加元素的方法： `add` 和 `addAll`，因此两个方法都要覆盖。

文中给出了下面的示例代码：

```
import java.util.Collection;
import java.util.HashSet;

public class InstrumentedHashSet<E> extends HashSet<E> {

    private int addCount = 0;

    public InstrumentedHashSet() {
    }

    public InstrumentedHashSet(int initCap, float loadFactor) {
        super(initCap, loadFactor);
    }

    @Override
    public boolean add(E e) {
        addCount++;
        return super.add(e);
    }

    @Override
    public boolean addAll(Collection<? extends E> c) {
        addCount += c.size();
        return super.addAll(c);
    }

    public int getAddCount() {
        return addCount;
    }
}
```


这段代码看似没啥问题，添加一个元素时 `addCount` 加 1 ，添加多个元素时 `addAll` 函数中会加上参数集合的长度。

接下来我们编写单元测试代码，来验证功能是否正确：

```
public class InstrumentedHashSetTest {

    @Test
    public void testAddCount() {
        List<String> stringList = Arrays.asList("Snap", "Crackle", "Pop");
        InstrumentedHashSet<String> s = new InstrumentedHashSet<>();
        s.addAll(stringList);
        Assert.assertEquals(stringList.size(), s.getAddCount());
    }
}
```

我们运行该程序，发现断言失败，显示期待的值是 3 ，而实际值却是 6 :

> java.lang.AssertionError:
> Expected :3
> Actual :6

此时多人会奇怪，为啥结果会是 6 呢?

希望大家在读下一部分之前先停下来想想为什么会这样，然后再去和后面的解读核对，这样印象会更深刻一些。

否则就像看着答案做题一样，看的时候觉得啥都会，其实没有真正掌握。

3.1 猜想验证

遇到问题时尽量要：先猜想，后验证。

遇到问题根据已有知识去 “猜想”，然后去验证和实际答案对比，才能发现自己理解不到位的地方，可以纠正自己理解。

这样学习才更有效，印象也会更加深刻。

> 很多人很羡慕别人能够快速准确地定位问题，但是平时自己却不能够通过简单的问题锻炼自己的猜想验证能力，直接看结论很难发现自己知识的欠缺，关键时自然无法快速准确地分析问题。

根据这种表现我们作出两个推测：

1、`addAll` 函数被调用了 2 次;

2、`addAll` 函数的执行，最终又调用了 3 次 `add` 函数。

根据源码可知：第一种猜测，显然不成立。

使用排除法，结论就显而易见了，只有第二种可能性。

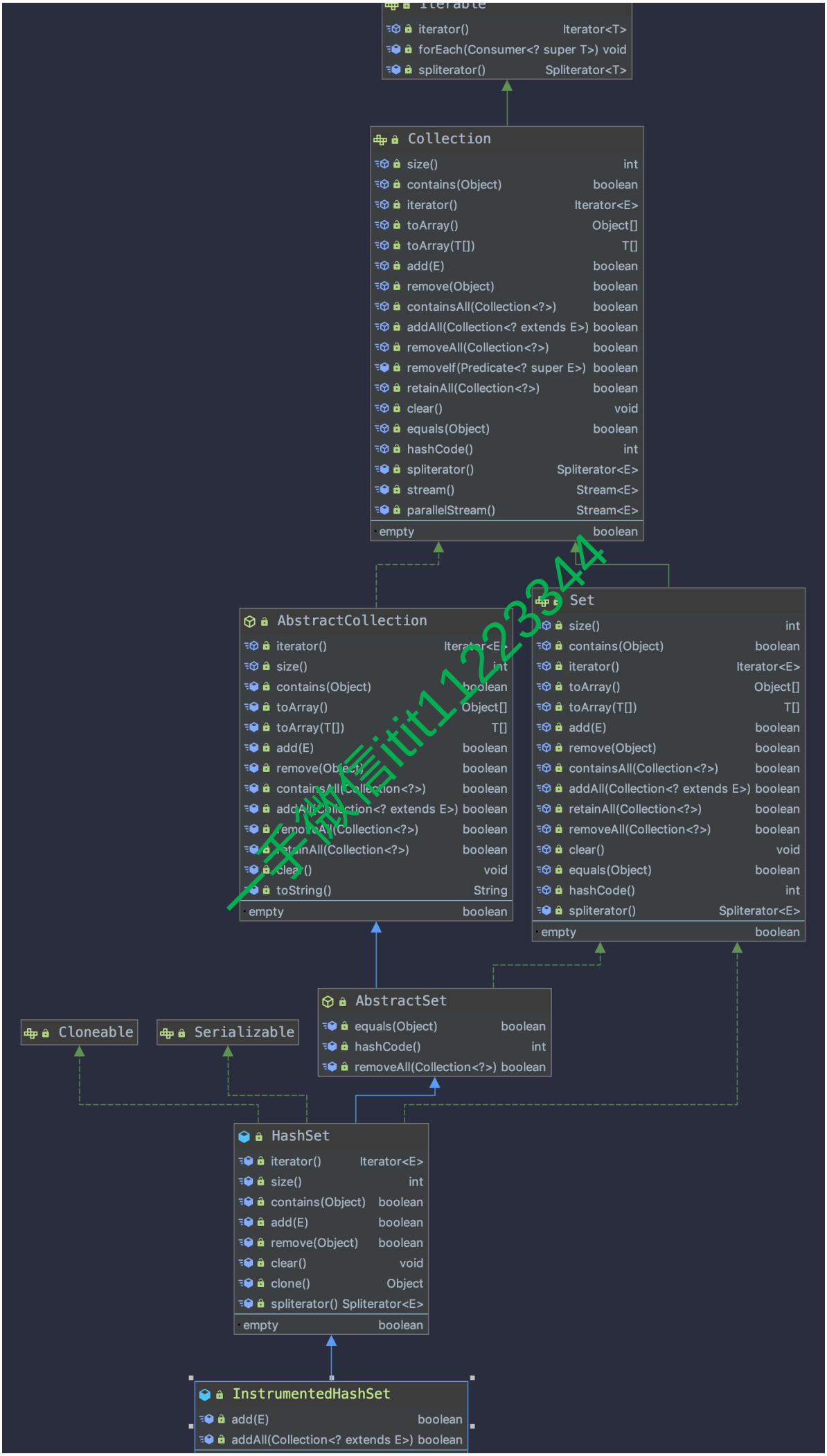
可是为什么 `add` 函数会被调用 3 次呢?

考试可以用排除法，但是研究不可取，我们还需要通过各种方法去验证。

> 很多人学习不够深入的原因之一就是看到某个结论就 “记住”，从来不去验证。
>
> 如果看到的博客、图书讲解有误，自己将会被带偏；如果讲解层次比较浅显，自己也将停留在这个层次。

3.2 类图大法

我们可以先使用 IDEA 自带的类图工具，来查看我们自定类的继承关系。

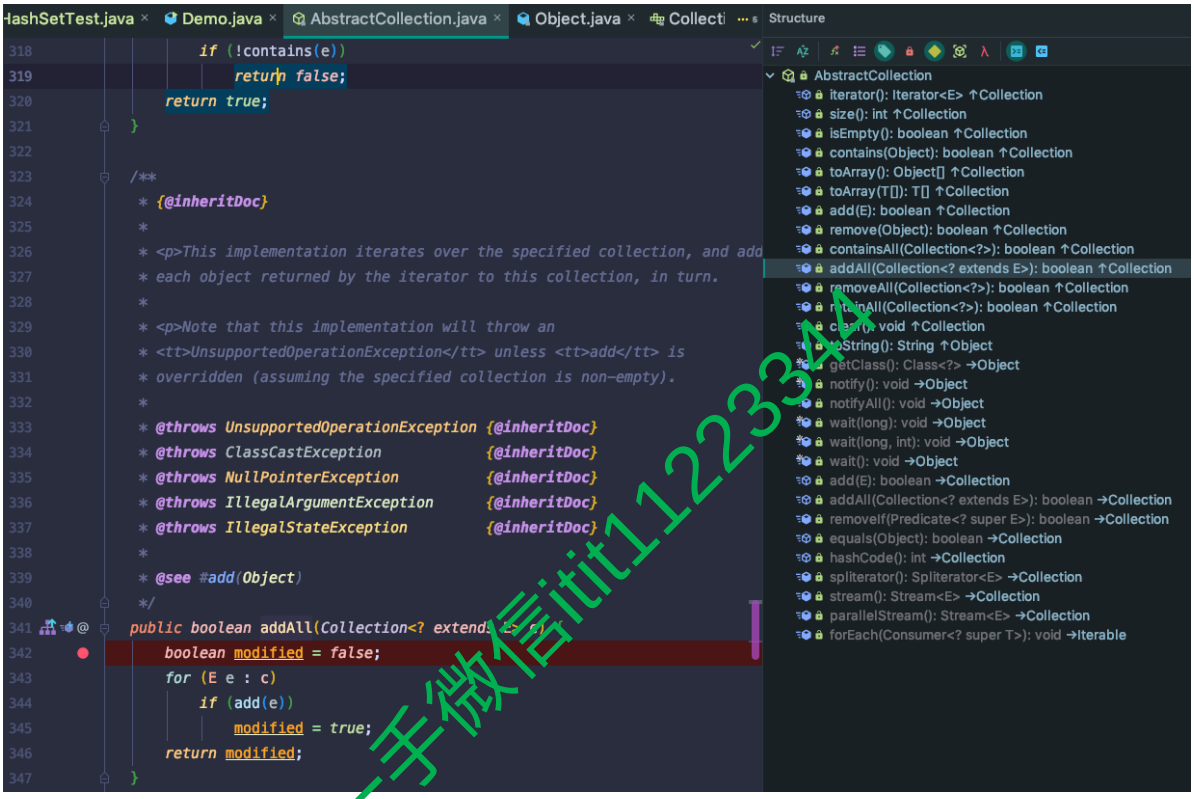


通过上述类图，我们可知：我们自定义的类，通过继承 `HashSet` 实现了 `Cloneable` 接口，即支持克隆；通过继承 `HashSet` 实现了 `Serializable` 即支持 Java 序列化，顶层实现了 `Iterable` 接口，即支持迭代器方式遍历元素。

当我们对一个类不足够熟悉时使用类图，我们可以快速了解它。可以通过观察函数名、参数列表和返回值等，来快速找到我们需要的函数。

我们可以点击源码中的 `super.addAll` 查看调用的函数源码，发现来自 `java.util.AbstractCollection#addAll`。

我们还可以通过函数列表来了解单个类，了解除了常用功能之外还有哪些函数，它们的作用是什么。



在此，建议大家在平时学习和开发时，如果任务并不是特别繁重，可以偶尔进入常用的 JDK 源码和第三方框架类中，查看它们的函数列表，了解该类提供的函数，容易发现一些好用的但是自己不常用的函数，能够了解其底层实现。这样积少成多，会学的越来越深入。

> 举一个常见的现象：
>
> 1、很多人经常使用 guava、commons-lang、commons-collections4 等知名三方工具类，但是永远都是用最常见的那几个工具函数。由于没有养成去源码中看函数列表的习惯，实际开发时会发现自己会重复造轮子，而自己造的轮子不管从健壮性还是代码的优雅程度都不如三方工具类，而且还浪费了不少时间；
>
> 2、很多人没有随手养成看源码的习惯，总是依赖看博客、看书来学习，导致缺少了一个非常好的学习途径。

如要移除本例中的字符串长度大于 3 的元素，很多人可能会 “颇有自信” 地用迭代器这么写：

```
List<String> stringList = Arrays.asList("Snap", "Crackle", "Pop");
InstrumentedHashSet<String> s = new InstrumentedHashSet<>();
s.addAll(stringList);

Iterator<String> iterator = s.iterator();
while (iterator.hasNext()){
```

```
    String next = iterator.next();
    if(StringUtils.isNotBlank(next) && next.length() >3){
        iterator.remove();
    }
}
System.out.println(s);
```

但是通过函数列表我们发现有一个 `removeIf` 函数，我们进入函数看源码：

```
default boolean removeIf(Predicate<? super E> filter) {
    Objects.requireNonNull(filter);
    boolean removed = false;
    final Iterator<E> each = iterator();
    while (each.hasNext()) {
        if (filter.test(each.next())) {
            each.remove();
            removed = true;
        }
    }
    return removed;
}
```

我们“惊讶地”发现，和我们要写的代码“惊人地”一致，既然都给我们封装好了，我们可以通过它来简化代码：

```
List<String> stringList = Arrays.asList("Snap", "Crackle", "Pop");
InstrumentedHashSet<String> s = new InstrumentedHashSet<>();
s.addAll(stringList);

s.removeIf(next -> StringUtils.isNotBlank(next) && next.length() > 3);
System.out.println(s);
```


通过这个简单的案例，希望大家可以养成好的学习习惯，这将有助你快速进阶。

接下来我们回归正题，继续研究我们前面提到的问题。

3.3 源码大法

其实看官方解释之前，最好自己动手根据我们已经掌握的方法先去研究这个问题，然后再去验证。

既然出现问题，我们就要分析问题。

既然调用了 `supper.addAll()` ，该函数继承自 java.util.AbstractCollection，我们可以先看下它的实现：

```
public boolean addAll(Collection<? extends E> c) {
    boolean modified = false;
    for (E e : c)
        if (add(e))
            modified = true;
    return modified;
}
```

我们发现，这里会通过 foreach 循环的方式分别调用 add 函数来添加每个元素。

然后我们进入默认的 add 函数:

```
public boolean add(E e) {
    throw new UnsupportedOperationException();
}
```

发现该函数默认会抛出不支持的操作异常（UnsupportedOperationException ）。

我们还发现父类 `HashSet` 重写了 add 函数 （java.util.HashSet#add）：

```
public boolean add(E e) {
    return map.put(e, PRESENT)==null;
}
```

我们自定义的类也重写了 `add` 函数（com.imooc.basic.inheritance.InstrumentedHashSet#add）：

```
@Override
public boolean add(E e) {
    addCount++;
    return super.add(e);
}
```

根据运行结果，我们推测调用到了 `InstrumentedHashSet` 类重写的 `add` 函数。

看到这里**我们得到了一个启示**:

我们在学习和研究问题时，要养成主动去看源码的习惯。

很多人想学得扎实一些，看各种书，但是从来不会主动去看源码，导致记住的东西容易忘记，很多知识知其然而不知其所以然。

看到这里**该同学又产生了一个疑问**:

为啥调用 `supper.addAll` 里面自定义类的 `add` 函数，没调用父类的 `add` 函数呢?

其实从这个疑问中可以看出该同学基础并不扎实，对面向对象的多态特性理解不够透彻。

3.4 断点调试大法

有问题没关系， 为了解开困惑，我们继续用我们已经掌握的方法来研究。

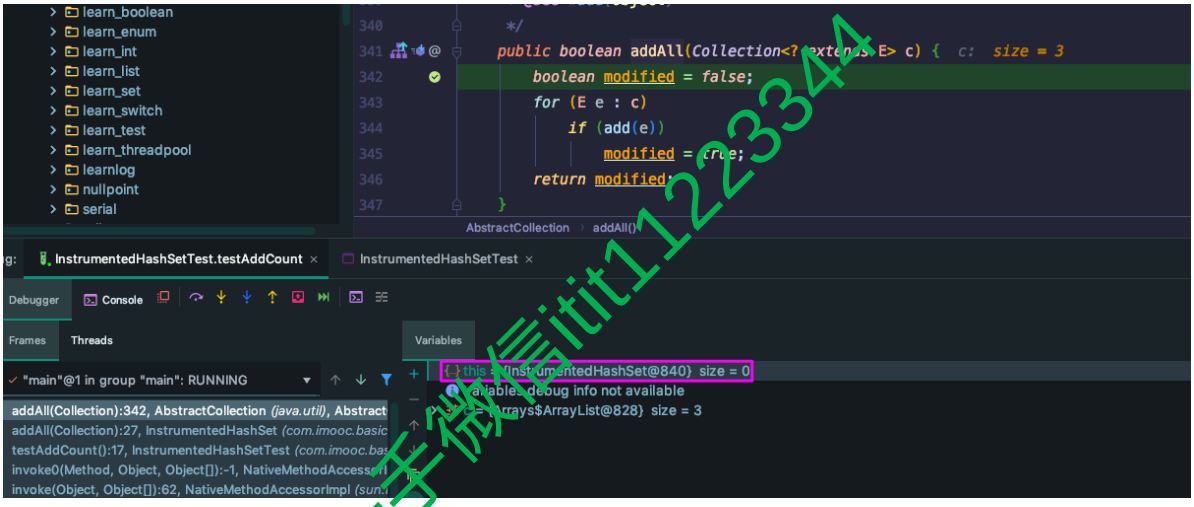
接下来我们用断点调试大法来分析这个问题。

我们先思考一个问题：

`java.util.AbstractCollection#addAll` 并不是静态函数，属于实例函数，也就是说这里的 `add(e)` 等价于 `this.add(e)`

```
public boolean addAll(Collection<? extends E> c) {
    boolean modified = false;
    for (E e : c)
        if (add(e))
            modified = true;
    return modified;
}
```

那么这里的 this 是什么（函数所属的对象是谁）？



单步或者通过条件断点走到这里时我们发现，此时 this 的类型为 `InstrumentedHashSet` 。

因此，这里的 this 应该是调用测试代码中通过 new 关键字构造的对象：

```
InstrumentedHashSet<String> s = new InstrumentedHashSet<>();
```

调用的就应该是 `com.imooc.basic.inheritance.InstrumentedHashSet#add` 函数。

继续单步跟到函数内部，发现的确如此。

自此，通过代码调试我们找到了 addCount 值为 6 的原因。

那么是不是这个问题到这里就结束了呢?

当然不是...

问题来了：

如果我们不借助调试，如何确定它最终调用的是 `com.imooc.basic.inheritance.InstrumentedHashSet#add` 函数，而不是直接调用父类（HashSet）中的 `add` 函数呢？

很多人可能会回答："显而易见"、"博客上这么说的"、"看书上就这么讲的"、"老师就是这么教的"，囧...

当你回答上述答案，而不再深究时，就说明你将在很长一段时间停留在这个层次。

这种 "显而易见"，这种 "书上说的" 将是阻碍我们进一步深入研究的一个很大的障碍，很多人都会存在这种问题，希望大家警惕。

很多 "显而易见" 的知识，并没那么 "理所当然"，请看下面的分析。

3.5 反汇编和 JVMS 大法

我们继续从字节码层面研究这个问题。

我们先对代码进行编译： `javac InstrumentedHashSet.java`

然后对编译后的字节码文件进行反汇编： `javap -c InstrumentedHashSet

得到如下的反汇编代码：

```
public class com.imooc.basic.inheritance.InstrumentedHashSet<E> extends
java.util.HashSet<E> {
  public com.imooc.basic.inheritance.InstrumentedHashSet();
    Code:
       0: aload_0
       1: invokespecial #1
       4: aload_0
       5: iconst_0
       6: putfield      #2
       9: return

  public com.imooc.basic.inheritance.InstrumentedHashSet(int, float);
    Code:
       0: aload_0
       1: iload_1
       2: fload_2
       3: invokespecial #3
       6: aload_0
       7: iconst_0
       8: putfield      #2
      11: return

  public boolean add(E);
    Code:
       0: aload_0
       1: dup
       2: getfield      #2
       5: iconst_1
       6: iadd
       7: putfield      #2
      10: aload_0
      11: aload_1
      12: invokespecial #4
      15: ireturn

  public boolean addAll(java.util.Collection<? extends E>);
```

```
    Code:
       0: aload_0
       1: dup
       2: getfield      #2
       5: aload_1
       6: invokeinterface #5,  1
      11: iadd
      12: putfield      #2
      15: aload_0
      16: aload_1
      17: invokespecial #6
      20: ireturn

  public int getAddCount();
    Code:
       0: aload_0
       1: getfield      #2
       4: ireturn
}
```

从上述代码，我们看到，super 关键字在编译器已经确定要调用的函数。即 `super.add(e);` 调用的是 `java.util.HashSet#add` 函数，`super.addAll(c)` 调用的是 `HashSet.addAll` 实际上该函数是继承自 `AbstractCollection#addAll`。

既然 `super.addAll` 调到了 `AbstractCollection#addAll`，**那么这里的 `addAll` 中调用的 `add` 又是哪个函数呢**?

我们通过 `javap -c AbstractCollection` 反汇编 `AbstractCollection`：

```
public abstract class java.util.AbstractCollection<E> implements
java.util.Collection<E> {
  protected java.util.AbstractCollection();
    Code:
       0: aload_0
       1: invokespecial #2
       4: return


  public boolean add(E);
    Code:
       0: new           #23
       3: dup
       4: invokespecial #24
       7: athrow

  public boolean addAll(java.util.Collection<? extends E>);
    Code:
       0: iconst_0
       1: istore_2
       2: aload_1
       3: invokeinterface #26,  1
       8: astore_3
       9: aload_3
      10: invokeinterface #5,  1
```

```
      15: ifeq          40
      18: aload_3
      19: invokeinterface #6,  1
      24: astore        4
      26: aload_0
      27: aload         4
      29: invokevirtual #28
      32: ifeq          37
      35: iconst_1
      36: istore_2
      37: goto          9
      40: iload_2
      41: ireturn

}
```

理解该问题的关键就是要理解 invokevirtual 方法调用指令。

此时有些同学又要开始百度了！

NO，请打开我们 “随手必备的” JVMS，这一个习惯非常重要，直接决定你是掌握最准确的一手资料还是第 N 手资料。

在 JVMS 中搜索 invokevirtual 去了解即可找到它的含义和示例。

部分描述摘抄如下：

> The Java Virtual Machine gives special treatment to signature polymorphic methods in the *invokevirtual* instruction (§*invokevirtual*) in order to effect invocation of a *method handle*.
>
> JVM 可以通过 invokevirtual 实现多态函数逻辑。
>
> *invokevirtual* invokes an instance method of an object, dispatching on the (virtual) type of the object. This is the normal method dispatch in the Java programming language.
>
> *invokevirtual* 调用对象的实例函数，会根据对象的实际类型进行分派（即：在编译器不能确定最终调用的是子类还是父类的方法）。
>
> The difference between the *invokespecial* instruction and the *invokevirtual* instruction (§*invokevirtual*) is that *invokevirtual* invokes a method based on the class of the object. The *invokespecial* instruction is used to invoke instance initialization methods (§2.9) as well as private methods and methods of a superclass of the current class.
>
> `invokevirtual` 是基于对象的类来调用的方法的，而 `invokespecial` 用于调用实例初始化方法（构造函数），`private` 方法和当前类的父类的方法

另外，我们顺手把相关的 invokeinterface、invokespecial、invokestatic、invokedynamic 也初步了解一下：

- **invokestatic**：用于调用静态方法，即使用 static 关键字修饰的方法。这些方法在编译器就可以确定，运行期不会修改，因此方法调用指令中效率最高，属于静态绑定；
- **invokespecial**：用于调用私有实例方法、构造器，以及使用 super 关键字调用父类的实例方法或构造器，和所实现接口的默认方法。用在类加载时就能确定具体的方法，不需要等到运行时根据实际对象去调用该对象的函数；
- **invokevirtual**：用于调用非私有实例方法；
- **invokeinterface**：用于调用接口方法，在运行时确定一个实现此接口的对象；

- **invokedynamic**：用于调用动态方法，invokedynamic 把如何查找目标方法的决定权从虚拟机下放到了具体的用户代码中，为实现 lambda 表达式，实现动态语言等提供了便利。

> 注：
>
> 1、学习一个知识时，如果能主动学习相关知识，并对比他们的异同，可以学的更全面；
>
> 2、想深入学习更多虚拟机指令，可参考《深入理解 Java 虚拟机》、《Java 虚拟机规范》等资料。

有了这些知识背景，我们再回看我们反编译后的代码就容易理解了：

上述反汇编代码中的下面这行：

表示我们自定义 `add` 函数中的 `super.add()` ，即使用 super 关键字调用父类实例方法：`HashSet#add` 函数

再回看前面提到的关键代码，`AbstractCollection#addAll` 调用的 `add` 函数的反汇编代码：

```
public boolean addAll(java.util.Collection<? extends E>);
   Code:
      0: iconst_0
      1: istore_2
      2: aload_1
      3: invokeinterface #26,  1
      8: astore_3
      9: aload_3
     10: invokeinterface #5,  1
     15: ifeq          40
     18: aload_3
     19: invokeinterface #6,  1
     24: astore        4
     26: aload_0
     27: aload         4
     29: invokevirtual #28
     32: ifeq          37
     35: iconst_1
     36: istore_2
     37: goto          9
     40: iload_2
     41: ireturn
```


根据源码 26 行我们看到对象是 aload_0 加载到操作数栈的 this , 参数是 aload 4 加载的迭代器 next 返回的字符串对象（参见偏移量为 18-24 next 函数调用部分）。

> 注：aload_0 表示将第一个参数加载到操作数栈，非静态函数中第一个参数是 this。
>
> 有些同学可能对虚拟机指令不太熟悉，虚拟机指令最权威的参考资料还是《Java 虚拟机规范》，大家还可以参考周大大的《深入理解 Java 虚拟机》。

此时的 this 就是我们创建的 `InstrumentedHashSet` 类型的对象。

因为我们自定义类重写了 `add` 函数，根据 invokevirtual 的含义，我们就可以确认会调用到我们自定义的 `add` 函数。

至此，该问题就迎刃而解了。

插曲：

可能很多同学反汇编时会使用 -v 选项，来查看附加信息，

`javap -v AbstractCollection`：

会发现这里 **“明明调用的就是 AbstractCollection.add 函数啊”，困惑 ing...**

其实这正是本专栏为什么要一直强调：**“是什么，为什么比怎么做更重要”** 这个思想。

因为通过对 `invokespecial` “是什么” 的学习，我们就知道 ** 多态函数会根据实际调用的对象类型运行时选择方法，在编译器无法确定。

那么，**为啥编译时给出函数签名是 `AbstractCollection.add` 呢**?

其实 #28 这个参数代表的符号引用提供了调用所需的方法名称，参数列表，参数类型和方法返回值等

如果没有这个参数，虚拟机只能找到类型，而不知道到底调用哪个方法。让它多为难?

此时，虚拟机默默地长叹一声，“做虚拟机好难...”

3.6 “官方” 解读

关于第二部分描述的问题，《Effective Java》 给出的解释是：

问题出在 `HashSet` 的内部实现上， `addAll` 方法是基于 `add` 方法来实现的，虽然 HashSet 的文档并没有专门强调这一细节，但是这样做也是非常合理的。

`InstrumentedHashSet` 中的 `addAll` 方法首先给 `addCount` 加 3 ，然后利用 supper.addAll 来调用 HashSet 的 addAll 函数，然后调用被 `InstrumentedHashSet` 覆盖的 `add` 方法，每个元素调用一次，所以又加了 3 此，最终结果是 6。

很多人可能不会亲自分析，看到这一小段就会就此止步，认为自己 “真的懂了”。其实这就是很多人读了很多书时对很多知识一知半解，理解不够透彻的重要原因之一。

通过上面几步的分析之后，再看书本的解释就能理解地非常深刻。

本文灵活运用本专栏介绍的几个核心方法和思想来学习和研究《Effective Java》中的一个典型问题。

本小节想向大家传达的核心思想是：

- **“是什么”，“为什么” 有时候比 “怎么做” 更重要。**很多时候搞懂了 “是什么”，“为什么”，你就知道该 “怎么做” 了。很多时候是因为我们只记忆了 “是什么” 和 “怎么做”，没有思考将两者之间联系起来的 “为什么”，才导致我们知其然，而不知其所以然。
- **猜想和验证是非常重要的学习方式**，希望大家在学习和工作中多做这种训练，这样才能更容易地发现自己的问题，让自己错误的理解得到纠正，才能让自己的思考更严谨和深入。
- **方法是通用的**，一个好方法往往能够解决至少一类问题，本专栏所分享的方法绝不仅限于学习《手册》，还用来学习 Java 相关的知识。希望大家帮这些方法当作帮助自己解决问题的习惯，遇到问题时信手拈来。
- ** 解决问题的方法不止有一种，** 而能够用什么方法解决问题取决于你的知识面。通过单纯的看书，通过自己动手写例子，通过源码调试，通过反汇编等不同方法所能够掌握和理解的程度显然是不一样的。这是造成不同人学习能力差距的重要原因之一，然而介绍这些内容的专栏极少，本专栏就是其中之一，然而很多人更喜欢 “买椟还珠”，不愿意在这方面下功夫。
- **每一个疑问背后可能都隐藏着至少一个知识盲区，隐藏着一个彻底搞懂某个知识的机会**，然而很多人总是忽视这种机会。
- **很多人正是因为平时有时间时不想 “浪费时间” 去研究问题，才会在真正需要某个知识时，浪费了更多的时间去解决问题，走更多的弯路。**

很多人学的不好，不光是读书读的少，而是没有 GET 到重点，没有掌握学习的方法。

“知道”和“理解”是两回事，大家都知道“授人以鱼不如授人以渔”的道理，然而现实却是很多人根本不重视“渔”，只重视找“鱼”；大家都知道“磨刀不误砍柴工”的道理，然而现实生活中很多朋友着急“砍柴”，很少“磨刀”，认为“磨刀”是浪费时间。

然而，对同一个问题的认知可以分为好几个层次，很多人会在不同的层次认为自己“懂了”，而不再往下深挖。

希望更多的人，能够从更深的层次来理解知识，而不只是记忆知识，这样才能够知其所以然，真正掌握和运用知识。

希望大家在未来的学习过程中，能够重视方法的价值，能够多一些思考，在未来的学习和工作中少走一些弯路。

如果你觉得本专栏对你有帮助，欢迎推荐给更多朋友，一起交流学习。

1、《Effective Java》该章节给出的自定义 `addAll()` 函数除了书中提到问题，还存在哪些隐患？

2、由于我们无法修改 JDK 源码库，我这里给出了类似的代码范例，请修改注释处下面一行代码，让 `Parent` 类中的 `eatAll` 函数，调用自身的 `eat` 函数，而不是子类的 `eat` 函数。

```
public class Demo {
    public static void main(String[] args) {
        Child child = new Child();
        child.eatAll(Arrays.asList("a", "b"));
    }
}
```



```
import java.util.Collection;
import java.util.Iterator;

public class Parent<E> {

    public void eat(E e) {
        System.out.println("P:eat-->" + e);
    }
    public void eatAll(Collection<? extends E> c) {
        System.out.println("P:eatAll");
        Iterator<? extends E> iterator = c.iterator();
        while (iterator.hasNext()) {

           eat(iterator.next());
        }
    }
}
```

```
import java.util.Collection;

public class Child<E> extends Parent<E> {
```

```
    @Override
    public void eat(E e) {
        System.out.println("C:eat" + e);
    }

    @Override
    public void eatAll(Collection<? extends E> c) {
        System.out.println("C:eatAll");
        super.eatAll(c);
    }
}
```

}