

## 36 | 继承有什么安全缺陷？

2019-03-27 范学雷



有时候，为了解决一个问题，我们需要一个解决办法。可是，这个办法本身还会带来更多的问題。新问题的解决带来更新的问题，就这样周而复始，绵延不绝。

比如[上一篇文章中](#)，我们说到的敏感信息通过异常信息泄露的问题，就是面向对象设计和实现给我们带来的小困扰。再比如[前面还有一个案例](#)，说到了共享内存或者缓存技术带来的潜在危害和挑战，这些都是成熟技术发展背后需要做出的小妥协。只是有时候，这些小小的妥协如果没有被安排好和处理好，可能就会带来不成比例的代价。

### 评审案例

我们一起来看看一段节选的`java.io.FilePermission`类的定义。你知道为什么`FilePermission`被定义为`final`类吗？

```
package java.io;

// <snipped>

/**
 * This class represents access to a file or directory. A
 * FilePermission consists of a pathname and a set of actions
 * valid for that pathname.
 * <snipped>
```

```

*/

public final class FilePermission
    extends Permission implements Serializable {
/**
 * Creates a new FilePermission object with the specified actions.
 * <i>path</i> is the pathname of a file or directory, and
 * <i>actions</i> contains a comma-separated list of the desired
 * actions granted on the file or directory. Possible actions are
 * "read", "write", "execute", "delete", and "readlink".
 * <snipped>
 */
    public FilePermission(String path, String actions);

/**
 * Returns the "canonical string representation" of the actions.
 * That is, this method always returns present actions in the
 * following order: read, write, execute, delete, readlink.
 * <snipped>
 */
    @Override
    public String getActions();

/**
 * Checks if this FilePermission object "implies" the
 * specified permission.
 * <snipped>
 * @param p the permission to check against.
 *
 * @return <code>true</code> if the specified permission
 *         is not <code>null</code> and is implied by this
 *         object, <code>false</code> otherwise.
 */
    @Override
    public boolean implies(Permission p);

    // <snipped>
}

```

`FilePermission`被声明为`final`，也就意味着该类不能被继承，不能被扩展了。我们都知道，在面向对象的设计中，是否具备可扩展性是一个衡量设计优劣的好指标。如果允许扩展的话，那么想要增加一个“link”的操作就会方便很多，只要扩展`FilePermission`类就可以了。但是对于`FilePermission`这个类，OpenJDK为什么放弃了可扩展性？

## 案例分析

如果我们保留`FilePermission`的可扩展性，你来评审一下下面的代码，可以看出这段代码的问题吗？

```
package com.example;

public final class MyFilePermission extends FilePermission {

    @Override
    public String getActions() {
        return "read";
    }

    @Override
    public boolean implies(Permission p) {
        return true;
    }
}
```

如果你还没有找出这个问题，可能是因为我还遗漏了对`FilePermission`常见使用场景的介绍。在Java的安全管理模式，一个用户通常可能会被授予有限的权限。比如用户“xuelel”可以读取用户“duke”的文件，但不能更改用户“duke”的文件。

授权的策略可能看起来像下面的描述：

```
grant Principal com.sun.security.auth.UnixPrincipal "xuelel" {
    permission com.example.MyFilePermission "/home/duke", "read";
};
```

这项策略要想起作用，上面的描述就要转换成一个`MyFilePermission`的实例。然后调用该实例的`implies()`方法类判断是否可以授权一项操作。

```
Permission myPermission = ... // read "/home/duke"

public void checkRead() {
    if (myPermission.implies(New FilePermission(file, "read"))) {
        // read is allowed.
    } else {
        // throw exception, read is not allowed.
    }
}

public void checkWrite() {
    if (myPermission.implies(New FilePermission(file, "write"))) {
        // write is allowed.
    } else {
        // throw exception, write is not allowed.
    }
}
```

这里请注意，`MyFilePermission.implies()`总是返回“true”，所以上述的`checkRead()`和`checkWrite()`方法总是成功的，不管用户被明确指示授予了什么权限，实际上暗地里他已经被授予了所有权限。这就成功地绕过了Java的安全管理。

能够绕过Java安全机制的主要原因，在于我们允许了`FilePermission`的扩展。而扩展类的实现，有可能有意或者无意地改变了`FilePermission`的规范和运行，从而带来不可预料的行为。

如果你关注OpenJDK安全组的代码评审邮件组，你可能会注意到，对于面向对象的可扩展性这一便利和诱惑，很多工程师能够保持住克制。

保持克制，可能会遗漏一两颗看似近在眼前的甜甜的糖果，但可以减轻你对未来长期的担忧。

一个类或者方法如果使用了`final`关键字，我们可以稍微放宽心。如果没有使用`final`关键字，我们可能需要反复揣摩好长时间，仔细权衡可扩展性可能会带来的弊端。

一个公共类或者方法如果使用了`final`关键字，将来如果需要扩展性，就可以去掉这个关键字。但是，如果最开始没有使用`final`关键字，特别是对于公开的接口来说，将来想要加上就可能是一件非常困难的事。

上面的例子是子类通过改变父类的规范和行为带来的潜在问题。那么父类是不是也可以改变子类的行为呢？这听起来有点怪异，但是父类对子类行为的影响，有时候也的确是一个让人非常头

疼的问题。

## 麻烦的继承

我先总结一下，父类对子类行为的影响大致有三种：

1. 改变未继承方法的实现，或者子类调用的方法的实现（**super**）；
2. 变更父类或者父类方法的规范；
3. 为父类添加新方法。

第一种和第三种相对比较容易理解，第二种稍微复杂一点。我们还是通过一个例子来看看其中的问题。

**Hashtable**是一个古老的，被广泛使用的类，它最先出现在**JDK 1.0**中。其中，**put()**和**remove()**是两个关键的方法。在**JDK 1.2**中，又有更多的方法被添加进来，比如**entrySet()**方法。

```

public class Hashtable<K,V> ... {
    // snipped
    /**
     * Returns a {@link Set} view of the mappings contained in
     * ( this map.
     *
     * The set is backed by the map, so changes to the map are
     * reflected in the set, and vice-versa. If the map is modified
     * while an iteration over the set is in progress (except through
     * the iterator's own {@code remove} operation, or through the
     * {@code setValue} operation on a map entry returned by the
     * iterator) the results of the iteration are undefined. The set
     * supports element removal, which removes the corresponding
     * mapping from the map, via the {@code Iterator.remove},
     * {@code Set.remove}, {@code removeAll}, {@code retainAll} and
     * {@code clear} operations. It does not support the
     * {@code add} or {@code addAll} operations.
     *
     * @since 1.2
     */
    public Set<Map.Entry<K,V>> entrySet() {
        // snipped
    }
    // snipped
}

```

这就引入了一个难以察觉的潜在的安全漏洞。你可能会问，添加一个方法不是很常见吗？这能有什么问题呢？

问题在于继承**Hashtable**的子类。假设有一个子类，它的**Hashtable**里要存放敏感数据，数据的添加和删除都需要授权，在**JDK 1.2**之前，这个子类可以重写**put()**和**remove()**方法，加载权限检查的代码。在**JDK 1.2**中，这个子类可能意识不到**Hashtable**添加了**entrySet()**这个新方法，从而也没有意识到要重写覆盖**entrySet()**方法，然而，通过对**entrySet()**返回值的直接操作，就可以执行数据的添加和删除的操作，成功地绕过了授权。

```

public class MySensitiveData extends Hashtable<Object, Object> {
    // snipped

    @Override
    public synchronized Object put(Object key, Object value) {
        // check permission and then add the key-value
        // snipped
        super.put(key, value)
    }

    @Override
    public synchronized Object remove(Object key) {
        // check permission and then remove the key-value
        // snipped
        return super.remove(key);
    }
    // snipped, no override of entrySet()
}

```

```

MySensitiveData sensitiveData = ... // get the handle of the data
Set<Map.Entry<Object, Object>> sdSet = sensitiveData.entrySet();
sdSet.remove(...); // no permission check
sdSet.add(...);    // no permission check

// the sensitive data get modified, unwarranted.

```

现实中，这种问题非常容易发生。一般来说，我们的代码总是依赖一定的类库，有时候需要扩展某些类。这个类库可能是第三方的产品，也可能是一个独立的内部类库。但遗憾的是，类库并不知道我们需要拓展哪些类，也可能没办法知道我们该如何拓展。

所以，当有一个新方法添加到类库的新版本中时，这个新方法会如何影响扩展类，该类库也没有特别多的想象空间和处理办法。就像`Hashtable`要增加`entrySet()`方法时，让`Hashtable`的维护者意识到有一个特殊的`MySensitiveData`扩展，是非常困难和不现实的。然而`Hashtable`增加`entrySet()`方法，合情又合理，也没有什么值得抱怨的。

然而，当JDK 1.0/1.1升级到JDK 1.2时，`Hashtable`增加了`entrySet()`方法，上述的`MySensitiveData`的实现就存在严重的安全漏洞。要想修复该安全漏洞，`MySensitiveData`需要重

写覆盖`entrySet()`方法，植入权限检查的代码。

可是，我们怎样可能知道**MySensitiveData**需要修改呢！一般来说，如果依赖的类库进行了升级，没有影响应用的正常运营，我们就正常升级了，而不会想到检查依赖类库做了哪些具体的变更，以及评估每个变更潜在的影响。这实在不是软件升级的初衷，也远远超越了大部分组织的能力范围。

而且，如果**MySensitiveData**不是直接继承**Hashtable**，而是经过了中间环节，这个问题就会更加隐晦，更加难以察觉。

```
public class IntermediateOne extends Hashtable<Object, Object>;

public class IntermediateTwo extends IntermediateOne;

public class Intermediate extends IntermediateTwo;

public class MySensitiveData extends Intermediate;
```

糟糕的是，随着语言变得越来越高级，类库越来越丰富，发现这些潜在问题的难度也是节节攀升。我几乎已经不期待肉眼可以发现并防范这类问题了。

那么，到底有没有办法可以防范此类风险呢？

主要有两个方法。

一方面，当我们变更一个可扩展类时，要极其谨慎小心。一个类如果可以不变更，就尽量不要变更，能在现有框架下解决问题，就尽量不要试图创造新的轮子。有时候，我们的确难以压制想要创造出什么好东西的冲动，这是非常好的品质。只是变更公开类库时，一定要多考虑这么做的潜在影响。你是不是开始思念**final**关键字的好处了？

另一方面，当我们扩展一个类时，如果涉及到敏感信息的授权与保护，可以考虑使用代理的模式，而不是继承的模式。代理模式可以有效地降低可扩展对象的新增方法带来的影响。



```
public class MySensitiveData {  
    private final Hashtable hashtable = ...  
  
    public synchronized Object put(Object key, Object value) {  
        // check permission and then add the key-value  
        hashtable.put(key, value)  
    }  
  
    public synchronized Object remove(Object key) {  
        // check permission and then remove the key-value  
        return hashtable.remove(key);  
    }  
}
```

我们使用了**Java**语言来讨论继承的问题，其实这是一个面向对象机制的普遍的问题，甚至它也不单单是面向对象语言的问题，比如使用**C**语言的设计和实现，也存在类似的问题。

## 小结

通过对这个案例的讨论，我想和你分享下面两点个人看法。

1. 一个可扩展的类，子类和父类可能会相互影响，从而导致不可预知的行为。
2. 涉及敏感信息的类，增加可扩展性不一定是个优先选项，要尽量避免父类或者子类的影响。

学会处理和保护敏感信息，是一个优秀工程师必须迈过的门槛。

## 一起来动手

了解语言和各种固定模式的缺陷，是我们打怪升级的一个很好的办法。有时候，我们偏重于学习语言或者设计经验的优点，忽视了它们背后做出小小的妥协，或者缺陷。如果能利用好优点，处理好缺陷，我们就可以更好地掌握这些经验总结。毕竟世上哪有什么完美的东西呢？不完美的东西，用好了，就是好东西。

我们利用讨论区，来聊聊设计模式这个老掉牙的、备受争议的话题。说起“老掉牙”，科技的进步真是快，设计模式十多年前还是一个时髦的话题，如今已经不太受待见了，虽然我们或多或少，或直接或间接地都受益于设计模式的思想。如果你了解过设计模式，你能够分享某个设计模式的优点和缺陷吗？使用设计模式有没有给你带来实际的困扰呢？

上面的例子中，我们提到了使用代理模式来降低父类对子类的影响。那么你知道代理模式的缺陷

吗？

欢迎你把自己的经验和看法写在留言区，我们一起来学习、思考、精进！

如果你觉得这篇文章有所帮助，欢迎点击“请朋友读”，把它分享给你的朋友或者同事。



# 代码精进之路

你写的每一行代码都是你的名片

**范学雷**  
Oracle 首席软件工程师  
Java SE 安全组成员  
OpenJDK 评审成员



新版升级：点击「 请朋友读」，10位好友免费读，邀请订阅更有**现金**奖励。

## 精选留言



天佑

1

嗯，子类会把父类的安全实现改的不安全

父类扩展新方法时，会给子类添乱，比如子类费劲心机防绕行，父类却悄悄加了个“后门”。。

。

2019-03-27



hua168

0

如果是web程序的话，攻击者是怎么查看我们内部程序？

如果是API接口的话，这些方法我们不是隐藏起来，不公开，它怎么绕过漏洞攻击？

2019-03-27

作者回复

没太明白这两个问题。

无法访问，当然就没问题。公开接口的问题是，公开了，就可以被使用，使用场景什么样的，我们不知道。有没有场景，漏洞无法被攻击者利用，当然有的。一个场景没问题，不代表另一

个场景就没问题。多用户、多应用、多类库的场景，这些用户之间、应用之间、类库之间，彼此可以不信任，可以参考现代编程语言和操作系统的domain的概念，或者虚拟机、云计算、远程调用等的技术架构。

2019-03-28



轻歌赋

👍 0

有个问题，案例中hashtable增加了一个entryset后，攻击者如何直接访问对象的entryset呢？以web程序为例的话，我想不出用户如何传入可以执行的代码，能过直接让权限检查的调用对象直接执行entryset，也看不出对方如何能够重写我服务端的代码或者继承并且被jvm加载老师能给个实际的例子吗

2019-03-27

作者回复

访问不了对象当然不行。你先找找对象可以访问的场景。换种场景你可能就熟悉了，比如说浏览器，访问一个页面的用户名密码，怎么不被正在访问的另一个页面窃取？

2019-03-27