10个实用的但偏执的Java编程技术

2015-08-31 萌码

相信不少程序猿在在沉浸于编码一段时间以后，会渐渐对这些东西习以为常。

然而事实是，任何事情有可能出错，因此，在编程的时候，会采用“防御性编程”，即一些偏执习惯的原因。

接下来萌小妹为大家总结了10个最有用但偏执的Java编程技术：

1.将String字符串放在最前面

为了防止偶发性的NullPointerException 异常，我们通常将String放置在equals()函数的左边来实现字符串比较，如下代码：

```
// Bad
if (variable.equals("literal")) { ... }
// Good
if ("literal".equals(variable)) { ... }
```

这是随便用脑子想想就可以做的事，从Bad版本的代码改写表达式到Good版本的代码，这中间并不会丢失任何东西。

2.不要相信早期的JDK API

在Java早期，编程是一件非常痛苦的事情。那些API仍然很不成熟，也许你已经碰到过下面的代码块：

```
String[] files = file.list();
// Watch out
if (files != null) {
```

```
if (files != null) {
for (int i = 0; i < files.length; i++) {
...
}
}
```

如果这个虚拟路径不表示一个文件夹目录，则此方法返回null。否则将会返回一个字符串数组，每一个字符串表示目录中的文件或文件夹。

对，没错。我们可以添加一些校验：

```
if (file.isDirectory()) {
String[] files = file.list();
// Watch out
if (files != null) {
for (int i = 0; i < files.length; i++) {
...
}
}
}
```

3.不要相信“-1”

我知道这是偏执的，但Javadoc中对 String.indexOf()方法明确指出：对象内第一次出现指定字符的位置索引，如果为-1则表示该字符不在字符序列中。

所以使用-1是理所当然的，对吗？我说不对，请看以下代码：

```
// Bad
if (string.indexOf(character) != -1) { ... }
// Good
if (string.indexOf(character) >= 0) { ... }
```

谁知道呢。也许到时候他们改变了编码方式，对字符串并不区分大小写，也许更好的方式是返回-2？谁知道呢。

4.避免意外赋值

是的。这种事情也许经常会发生。

```
// Ooops
if (variable = 5) { ... }
// Better (because causes an error)
if (5 = variable) { ... }
// Intent (remember. Paranoid JavaScript: ===)
if (5 === variable) { ... }
```

所以你可以将比较常量放置在左侧，这样就不会发生意外赋值的错误了。

5.检查Null和Length

无论如何，只要你有一个集合、数组等，请确保它存在，并且不为空。

```
// Bad
if (array.length > 0) { ... }
// Good
if (array != null && array.length > 0) { ... }
```

你并不知道这些数组从哪里来，也许是来自早期版本的JDK API，谁知道呢。

6.所有的方法都是final的

你也许会告诉我你的开/闭原则，但这都是胡说八道。我不相信你（正确继承我这个父类的所有子类），我也不相信我自己（不小心继承我这个父类的所有子类）。所以对于那些意义明确的方法要严格用final标识。

```
// Bad
public void boom() { ... }
// Good. Don't touch.
public final void dontTouch() { ... }
```

7.所有变量和参数都是final

```
// Bad
```

```
void input(String importantMessage) {
String answer = "...";
answer = importantMessage = "LOL accident";
}
// Good
final void input(final String importantMessage) {
final String answer = "...";
}
```

8.重载时不要相信泛型

是，它可以发生。你相信你写的超级好看的API，它很直观，随之而来的，一些用户谁只是将原始类型转换成Object类型，直到那该死的编译器停止发牢骚，并且突然他们会链接错误的方法，以为这是你的错误。
看下面的代码：

```
// Bad
<T> void bad(T value) {
bad(Collections.singletonList(value));
}
<T> void bad(List<T> values) {
...
}
// Good
final <T> void good(final T value) {
if (value instanceof List)
good((List<?>) value);
else
good(Collections.singletonList(value));
}
final <T> void good(final List<T> values) {
...

}
```

因为，你知道……你的用户，他们就像：

```
// This library sucks
```

```
@SuppressWarnings("all")
Object t = (Object) (List) Arrays.asList("abc");
bad(t);
```

9.总是在Switch语句的Default中抛出异常

Switch语句……它们其中一个可笑的语句我不知道该对它敬畏还是哭泣，但无论如何，既然我们坚持用switch，那我们不妨将它用得完美，看下面的代码：

```
// Bad
switch (value) {
case 1: foo(); break;
case 2: bar(); break;
}
// Good
switch (value) {
case 1: foo(); break;
case 2: bar(); break;
default:
throw new ThreadDeath("That'll teach them");
}
```

当value == 3时，将会出现无法找到的提示，而不会让人不知所谓。

10.Switch语句带花括号

事实上，switch是最邪恶的语句，像是一些喝醉了或者赌输了的人在写代码一样，看下面的例子：

```
// Bad, doesn't compile
switch (value) {
case 1: int j = 1; break;
case 2: int j = 2; break;
}
// Good
switch (value) {
case 1: {
final int j = 1;
break;
```

```
}
case 2: {
final int j = 2;
break;
}
// Remember:
default:
throw new ThreadDeath("That'll teach them");
}
```

在switch语句中，每一个case语句的范围只有一行语句，事实上，这些case语句甚至不是真正的语句，他们就像goto语句中的跳转标记一样。