0个精妙的Java编码最佳实践

2014/03/31 | 分类： 基础技术 | 1 条评论
分享到： 0

本文由 ImportNew - liken 翻译自 jooq。欢迎加入 翻译小组。转载请见文末要求。

这是一个比Josh Bloch的Effective Java规则更精妙的10条Java编码实践的列表。和Josh Bloch的列表容易学习并且关注日常情况相比，这个列表将包含涉及API/SPI设计中不常见的情况，可能有很大影响。

我在编写和维护jOOQ(Java中内部DSL建模的SQL)时遇到过这些。作为一个内部DSL，jOOQ最大限度的挑战了Java的编译器和泛型，把泛型，可变参数和重载结合在一起，Josh Bloch可能不会推荐的这种太宽泛的API。

让我与你分享10个微妙的Java编码最佳实践：

1. 牢记C++的析构函数

记得C++的析构函数？不记得了？那么你真的很幸运，因为你不必去调试那些由于对象删除后分配的内存没有被释放而导致内存泄露的代码。感谢Sun/Oracle实现的垃圾回收机制吧！

尽管如此，析构函数仍提供了一个有趣的特征。它理解逆分配顺序释放内存。记住在Java中也是这样的，当你操作类析构函数语法：

- 使用JUnit的@Before和@After注释
- 分配，释放JDBC资源
- 调用super方法

还有其他各种用例。这里有一个具体的例子，说明如何实现一些事件侦听器的SPI：

```
@Override
public
void
beforeEvent(EventContext e) {
super
.beforeEvent(e);
// Super code before my code
}
@Override
public
void
afterEvent(EventContext e) {
// Super code after my code
super
.afterEvent(e);
}
interface
EventListener {

// Bad

void
message(String message);
}
```

臭名昭著的哲学家就餐问题是另一个说明它为什么重要的好例子。 关于哲学家用餐的问题，请查看链接：http://adit.io/posts/2013-05-11-The-Dining-Philosophers-Problem-With-Ron-Swanson.html

规则：无论何时使用before/after, allocate/free, take/return语义实现逻辑时，考虑是否逆序执行after/free/return操作。

2. 不要相信你早期的SPI演进判断

向客户提供SPI可以使他们轻松的向你的库/代码中注入自定义行为的方法。当心你的SPI演进判断可能会让你失望，使你认为你*(不)打算需要附加参数*。 当然，不应当过早增加功能。但一旦你发布了你的SPI，一旦你决定遵循语义版本控制，当你意识到在某种情况下你可能需要另外一个参数时，你会真的后悔在SPI中增加一个愚蠢的单参数的方法：

```
// Bad

void
message(String message);
}
```

如果你也需要消息ID和消息源，怎么办？API演进将会阻止你向上面的类型添加参数。当然，有了Java8，你可以添加一个defender方法，“防御”你早期糟糕的设计决策：

```
interface
EventListener {

// Bad

default
void
message(String message) {

message(message,
null
,
null
);
}

// Better?
interface
MessageContext {
void
message(
String message();
String message,
Integer id();
Integer id,
MessageSource source();
}
MessageSource source

);
interface
}
EventListener {

// Awesome!

void
message(MessageContext context);
}
```

注意，不幸的是，defender方法不能使用final修饰符。
但是比起使用许多方法污染你的SPI，使用上下文对象(或者参数对象)会好很多。

比起EventListner SPI你可以更容易演进MessageContext API，因为很少用户会实现它。

规则: 无论何时指定SPI时，考虑使用上下文/参数对象，而不是写带有固定参数的方法。

备注: 通过专用的MessageResult类型交换结果也是一个好主意，该类型可以使用建设者API构造它。这样将大大增加SPI进化的灵活性。

3. 避免返回匿名，本地或者内部类

Swing程序员通常只要按几下快捷键即可生成成百上千的匿名类。在多数情况下，只要遵循接口、不违反SPI子类型的生命周期(SPI subtype lifecycle)，这样

做也无妨。 但是不要因为一个简单的原因——它们会保存对外部类的引用，就频繁的使用匿名、局部或者内部类。因为无论它们走到哪，外部类就得跟到哪。例如，在局部类的域外操作不当的话，那么整个对象图就会发生微妙的变化从而可能引起内存泄露。
规则：在编写匿名、局部或内部类前请三思能否将它转化为静态的或普通的顶级类，从而避免方法将它们的对象返回到更外层的域中。
注意：使用双层花括号来初始化简单对象：

```
new HashMap<String, String>() {{

put("1", "a");

put("2", "b");
}}
```

这个方法利用了 JLS §8.6规范里描述的实例初始化方法(initializer)。表面上看起来不错，但实际上不提倡这种做法。因为要是使用完全独立的HashMap对象，那么实例就不会一直保存着外部对象的引用。此外，这也会让类加载器管理更多的类。

4. 现在就开始编写SAM!

Java8的脚步近了。伴随着Java8带来了lambda表达式，无论你是否喜欢。尽管你的API用户可能会喜欢，但是你最好确保他们可以尽可能经常的使用。因此除非你的API接收简单的“标量”类型，比如int、long、String 、Date，否则让你的API尽可能经常的接收SAM。
什么是SAM? SAM是单一抽象方法[类型]。也称为函数接口，不久会被注释为@FunctionalInterface。这与规则2很配，EventListener实际上就是一个SAM。最好的SAM只有一个参数，因为这将会进一步简化lambda表达式的编写。设想编写

```
listeners.add(c -> System.out.println(c.message()));
```

来替代

```
listeners.add(
new
EventListener() {

@Override

public
void
message(MessageContext c) {

System.out.println(c.message()));
}
});
```

设想以JOOX的方式来处理XML。JOOX就包含很多的SAM：

```
// Find elements with an ID

.find(c -> $(c).id() !=
null
)

// Find their child elements

.children(c -> $(c).tag().equals(
"order"
))

// Print all matches

.each(c -> System.out.println($(c)))
```

规则：对你的API用户好一点儿，从*现在*开始编写SAM/函数接口。

备注：有许多关于Java8 lambda表达式和改善的Collections API的有趣的博客：

- http://blog.informatech.cr/2013/04/10/java-optional-objects/
- http://blog.informatech.cr/2013/03/25/java-streams-api-preview/
- http://blog.informatech.cr/2013/03/24/java-streams-preview-vs-net-linq/
- http://blog.informatech.cr/2013/03/11/java-infinite-streams/

5.避免让方法返回null

我曾写过1、2篇关于java NULLs的文章，也讲解过Java8中引入新的Optional类。从学术或实用的角度来看，这些话题还是比较有趣的。

尽管现阶段Null和NullPointerException依然是Java的硬伤，但是你仍可以设计出不会出现任何问题的API。在设计API时，应当尽可能的避免让方法返回null，因为你的用户可能会链式调用方法：

```
initialise(someArgument).calculate(data).dispatch()
```

从上面代码中可看出，任何一个方法都不应返回null。实际上，在通常情况下使用null会被认为相当的异类。像 jQuery或 jOOX这样的库在可迭代的对象上已完全的摒弃了null。
Null通常用在延迟初始化中。在许多情况下，在不严重影响性能的条件下，延迟初始化也应该被避免。实际上，如果涉及的数据结构过于庞大，那么就要慎用延迟初始化。
规则：无论何时方法都应避免返回null。null仅用来表示“未初始化”或“不存在”的语义。

6.设计API时永远不要返回空(null)数组或List

尽管在一些情况下方法返回值为null是可以的，但是绝不要返回空数组或空集合！请看 java.io.File.list()方法，它是这样设计的：
此方法会返回一个指定目录下所有文件或目录的字符串数组。如果目录为空(empty)那么返回的数组也为空(empty)。如果指定的路径不存在或发生I/O错误，则返回null。
因此，这个方法通常要这样使用：

```
File directory =
// ...
1

2
if
(directory.isDirectory()) {
3
String[] list = directory.list();
4

5
if
6
(list !=
null
7
) {
8
for
(String file : list) {
9
// ...
10
}
11
}
}
```

```
1
```

大家觉得
null
检查有必要吗？大多数I/O操作会产生IOExceptions，但这个方法却
null
是无法存放I/O错误信息的。因此这样的设计，有以下
3
方面的不足：

- Null无助于发现错误
- Null无法表明I/O错误是由File实例所对应的路径不正确引起的
- 每个人都可能会忘记判断null情况

以集合的思维来看待问题的话，那么空的(empty)的数组或集合就是对“不存在”的最佳实现。返回空(null)数组或集合几乎是无任何实际意义的，除非用于延迟初始化。
规则：返回的数组或集合不应为null。

7. 避免状态，使用函数

HTTP的好处是无状态。所有相关的状态在每次请求和响应中转移。这是REST命名的本质：含状态传输(Representational state transfer)。在Java中这样做也很赞。当方法接收状态参数对象的时候从规则2的角度想想这件事。如果状态通过这种对象传输，而不是从外边操作状态，那么事情将会更简单。以JDBC为例。下述例子从一个存储的程序中读取一个光标。

```
1  CallableStatement s =
2  connection.prepareCall(
   "{ ? = ... }"
   );
3
4  // Verbose manipulation of statement state:
   s.registerOutParameter(
5  1
   , cursor);
   s.setString(
6  2
   ,
7  "abc"
   );
8  s.execute();
   ResultSet rs = s.getObject(
9  1
   );
10
11 // Verbose manipulation of result set state:
   rs.next();
12 rs.next();
```

这使得JDBC API如此的古怪。每个对象都是有状态的，难以操作。具体的说，有两个主要的问题：

- 在多线程环境很难正确的处理有状态的API
- 很难让有状态的资源全局可用，因为状态没有被描述

规则：更多的以函数风格实现。通过方法参数转移状态。极少操作对象状态。

8. 短路式 equals()

这是一个比较容易操作的方法。在比较复杂的对象系统中，你可以获得显著的性能提升，只要你在所有对象的equals()方法中首先进行相等判断:

```
@Override
public
boolean
equals(Object other) {

if
(
this
== other)
return
true
;

// 其它相等判断逻辑...
}
```

注意，其它短路式检查可能涉及到null值检查，所以也应当加进去:

```
@Override
public
boolean
equals(Object other) {

if
(
this
== other)
return
true
;

if
(other ==
null
)
return
false
;

// Rest of equality logic...
}
```

规则: 在你所有的equals()方法中使用短路来提升性能。

9. 尽量使方法默认为final

有些人可能不同意这一条，因为使方法默认为final与Java开发者的习惯相违背。但是如果你对代码有完全的掌控，那么使方法默认为final是肯定没错的：

- 如果你确实需要覆盖（override）一个方法（你真的需要？），你仍然可以移除final关键字
- 你将永远不会意外地覆盖（override）任何方法

这特别适用于静态方法，在这种情况下“覆盖”（实际上是遮蔽）几乎不起作用。我最近在Apache Tika中遇到了一个很糟糕的遮蔽静态方法的例子。看一下：

- TaggedInputStream.get(InputStream)
- TikaInputStream.get(InputStream)

TikaInputStream扩展了TaggedInputStream，以一种相对不同的实现遮蔽了它的静态get()方法。

与常规方法不同，静态方法不能互相覆盖，因为调用的地方在编译时就绑定了静态方法调用。如果你不走运，你可能会意外获得错误的方法。

规则：如果你完全掌控你的API，那么使尽可能多的方法默认为final。

10. 避免方法(T...)签名

在特殊场合下使用“accept-all”变量参数方法接收一个Object...参数就没有错的：

```
1 void acceptAll(Object... all);
```

编写这样的方法为Java生态系统带来一点儿JavaScript的感觉。当然你可能想要根据真实的情形限制实际的类型，比如String...。因为你不想要限制太多，你可能会认为用泛型T取代Object是一个好想法：

```
1 void acceptAll(T... all);
```

但是不是。T总是会被推断为Object。实际上你可能仅仅认为上述方法中不能使用泛型。更重要的是你可能认为你可以重载上述方法，但是你不能：

```
1 void acceptAll(T... all);
2 void acceptAll(String message, T... all);
```

这看起来好像你可以可选地传递一个String消息到方法。但是这个调用会发生什么呢？

编译器将T推断为<? extends Serializable & Comparable<?>>，这将会使调用不明确！ 所以无论何时你有一个“accept-all”签名（即使是泛型），你将永远不能类型安全地重载它。;API使用者可能仅仅在走运的时候才会让编译器“偶然地”选择“正确的”方法。但是也可能使用accept-all方法或者无法调用任何方法。	acceptAll("Message" 123 , "abc");

规则: 如果可能，避免“accept-all”签名。如果不能，不要重载这样的方法。

结论

Java是一个野兽。不像其它更理想主义的语言，它慢慢地演进为今天的样子。这可能是一件好事，因为以Java的开发速度就已经有成百上千个警告，而且这些警告只能通过多年的经验去把握。

敬请期待更多关于这个主题的前十名列表！