**关于Java你可能不知道的10件事**

呃，你是不是写Java已经有些年头了？还依稀记得这些吧： 那些年，它还叫做Oak；那些年，OO还是个热门话题；那些年，C++同学们觉得Java是没有出路的；那些年，Applet还风头正劲……

但我打赌下面的这些事中至少有一半你还不知道。这周我们来聊聊这些会让你有些惊讶的Java内部的那些事儿吧。

**1. 其实没有受检异常（checked exception）**

是的！JVM才不知道这类事情，只有Java语言才会知道。

今天，大家都赞同受检异常是个设计失误，一个Java语言中的设计失误。正如 Bruce Eckel [在布拉格的GeeCON会议上演示的总结](http://www.geecon.cz/speakers/?id=2)中说的， Java之后的其它语言都没有再涉及受检异常了，甚至Java 8的新式流API（Streams API）都不再拥抱受检异常 （[以lambda的方式使用IO和JDBC，这个API用起来还是有些痛苦的](http://blog.jooq.org/2014/05/23/java-8-friday-better-exceptions/)。）

想证明JVM不理会受检异常？试试下面的这段代码：

1. **public** **class** Test {
3. // 方法没有声明throws
4. **public** **static** **void** main(String[] args) {
5. doThrow(**new** SQLException());
6. }
8. **static** **void** doThrow(Exception e) {
9. Test.<RuntimeException> doThrow0(e);
10. }
12. @SuppressWarnings("unchecked")
13. **static** <E **extends** Exception>
14. **void** doThrow0(Exception e) **throws** E {
15. **throw** (E) e;
16. }
17. }

不仅可以编译通过，并且也抛出了SQLException，你甚至都不需要用上Lombok的[@SneakyThrows](http://projectlombok.org/features/SneakyThrows.html" \t "_blank)。

更多细节，可以再看看[这篇文章](http://blog.jooq.org/2012/09/14/throw-checked-exceptions-like-runtime-exceptions-in-java/" \t "_blank)，或Stack Overflow上的[这个问题](http://stackoverflow.com/q/12580598/521799" \t "_blank)。

**2. 可以有只是返回类型不同的重载方法**

下面的代码不能编译，是吧？

1. **class** Test {
2. Object x() { **return** "abc"; }
3. String x() { **return** "123"; }
4. }

是的！Java语言不允许一个类里有2个方法是『重载一致』的，而不会关心这2个方法的throws子句或返回类型实际是不同的。

但是等一下！来看看[Class.getMethod(String, Class...)](http://docs.oracle.com/javase/8/docs/api/java/lang/Class.html" \l "getMethod-java.lang.String-java.lang.Class...-" \t "_blank)方法的Javadoc：

注意，可能在一个类中会有多个匹配的方法，因为尽管Java语言禁止在一个类中多个方法签名相同只是返回类型不同，但是JVM并不禁止。 这让JVM可以更灵活地去实现各种语言特性。比如，可以用桥方法(bridge method)来实现方法的协变返回类型；桥方法和被重载的方法可以有相同的方法签名，但返回类型不同。

嗯，这个说的通。实际上，当写了下面的代码时，就发生了这样的情况：

1. **abstract** **class** Parent<T> {
2. **abstract** T x();
3. }
5. **class** Child **extends** Parent<String> {
6. @Override
7. String x() { **return** "abc"; }
8. }

查看一下Child类所生成的字节码：

1. // Method descriptor #15 ()Ljava/lang/String;
2. // Stack: 1, Locals: 1
3. java.lang.String x();
4. 0  ldc <String "abc"> [16]
5. 2  areturn
6. Line numbers:
7. [pc: 0, line: 7]
8. Local variable table:
9. [pc: 0, pc: 3] local: **this** index: 0 type: Child
11. // Method descriptor #18 ()Ljava/lang/Object;
12. // Stack: 1, Locals: 1
13. bridge synthetic java.lang.Object x();
14. 0  aload\_0 [**this**]
15. 1  invokevirtual Child.x() : java.lang.String [19]
16. 4  areturn
17. Line numbers:
18. [pc: 0, line: 1]

在字节码中，T实际上就是Object类型。这很好理解。

合成的桥方法实际上是由编译器生成的，因为在一些调用场景下，Parent.x()方法签名的返回类型期望是Object。 添加泛型而不生成这个桥方法，不可能做到二进制兼容。 所以，让JVM允许这个特性，可以愉快解决这个问题（实际上可以允许协变重载的方法包含有副作用的逻辑）。 聪明不？呵呵～

你是不是想要扎入语言规范和内核看看？可以在[这里](http://stackoverflow.com/q/442026/521799" \t "_blank)找到更多有意思的细节。

**3. 所有这些写法都是二维数组！**

1. **class** Test {
2. **int**[][] a()  { **return** **new** **int**[0][]; }
3. **int**[] b() [] { **return** **new** **int**[0][]; }
4. **int** c() [][] { **return** **new** **int**[0][]; }
5. }

是的，这是真的。尽管你的人肉解析器不能马上理解上面这些方法的返回类型，但都是一样的！下面的代码也类似：

1. **class** Test {
2. **int**[][] a = {{}};
3. **int**[] b[] = {{}};
4. **int** c[][] = {{}};
5. }

是不是觉得这个很2B？想象一下在上面的代码中使用[JSR-308/Java 8的类型注解](https://jcp.org/en/jsr/detail?id=308" \t "_blank)。 语法糖的数目要爆炸了吧！

1. @Target(ElementType.TYPE\_USE)
2. **@interface** Crazy {}
4. **class** Test {
5. @Crazy **int**[][]  a1 = {{}};
6. **int** @Crazy [][] a2 = {{}};
7. **int**[] @Crazy [] a3 = {{}};
9. @Crazy **int**[] b1[]  = {{}};
10. **int** @Crazy [] b2[] = {{}};
11. **int**[] b3 @Crazy [] = {{}};
13. @Crazy **int** c1[][]  = {{}};
14. **int** c2 @Crazy [][] = {{}};
15. **int** c3[] @Crazy [] = {{}};
16. }

类型注解。这个设计引入的诡异在程度上仅仅被它解决问题的能力超过。

或换句话说：

在我4周休假前的最后一个提交里，我写了这样的代码，然后。。。

【译注：然后，亲爱的同事你，就有得火救啦，哼，哼哼，哦哈哈哈哈～】

请找出上面用法合适的使用场景，还是留给你作为一个练习吧。

**4. 你没有掌握条件表达式**

呃，你认为自己知道什么时候该使用条件表达式？面对现实吧，你还不知道。大部分人会下面的2个代码段是等价的：

1. Object o1 = **true** ? **new** Integer(1) : **new** Double(2.0);

等同于：

1. Object o2;
3. **if** (**true**)
4. o2 = **new** Integer(1);
5. **else**
6. o2 = **new** Double(2.0);

让你失望了。来做个简单的测试吧：

1. System.out.println(o1);
2. System.out.println(o2);

打印结果是：

1. 1.0
2. 1

哦！如果『需要』，条件运算符会做数值类型的类型提升，这个『需要』有非常非常非常强的引号。因为，你觉得下面的程序会抛出NullPointerException吗？

1. Integer i = **new** Integer(1);
2. **if** (i.equals(1))
3. i = **null**;
4. Double d = **new** Double(2.0);
5. Object o = **true** ? i : d; // NullPointerException!
6. System.out.println(o);

关于这一条的更多的信息可以在[这里](http://blog.jooq.org/2013/10/08/java-auto-unboxing-gotcha-beware/" \t "_blank)找到。

**5. 你没有掌握复合赋值运算符**

是不是觉得不服？来看看下面的2行代码：

1. i += j;
2. ii = i + j;

直觉上认为，2行代码是等价的，对吧？但结果即不是！JLS（Java语言规范）指出：

复合赋值运算符表达式 E1 op= E2 等价于 E1 = (T)((E1) op (E2)) 其中T是E1的类型，但E1只会被求值一次。

这个做法太漂亮了，请允许我引用[Peter Lawrey](https://twitter.com/PeterLawrey" \t "_blank)在Stack Overflow上的[回答](http://stackoverflow.com/a/8710747/521799" \t "_blank)：

使用\*=或/=作为例子可以方便说明其中的转型问题：

1. **byte** b = 10;
2. b \*= 5.7;
3. System.out.println(b); // prints 57
5. **byte** b = 100;
6. b /= 2.5;
7. System.out.println(b); // prints 40
9. **char** ch = '0';
10. ch \*= 1.1;
11. System.out.println(ch); // prints '4'
13. **char** ch = 'A';
14. ch \*= 1.5;
15. System.out.println(ch); // prints 'a'

为什么这个真是太有用了？如果我要在代码中，就地对字符做转型和乘法。然后，你懂的……

**6. 随机Integer**

这条其实是一个迷题，先不要看解答。看看你能不能自己找出解法。运行下面的代码：

1. **for** (**int** i = 0; i < 10; i++) {
2. System.out.println((Integer) i);
3. }

…… 然后要得到类似下面的输出（每次输出是随机结果）：

1. 92
2. 221
3. 45
4. 48
5. 236
6. 183
7. 39
8. 193
9. 33
10. 84

这怎么可能？！

我要剧透了…… 解答走起……

好吧，解答在这里(<http://blog.jooq.org/2013/10/17/add-some-entropy-to-your-jvm/>)， 和用反射覆盖JDK的Integer缓存，然后使用自动打包解包（auto-boxing/auto-unboxing）有关。 同学们请勿模仿！或换句话说，想想会有这样的状况，再说一次：

在我4周休假前的最后一个提交里，我写了这样的代码，然后。。。

【译注：然后，亲爱的同事你，就有得火救啦，哼，哼哼，哦哈哈哈哈～】

**7. GOTO**

这条是我的最爱。Java是有GOTO的！打上这行代码：

1. **int** **goto** = 1;

结果是：

1. Test.java:44: error: <identifier> expected
2. **int** **goto** = 1;
3. ^

这是因为goto是个[还未使用的关键字](http://docs.oracle.com/javase/tutorial/java/nutsandbolts/_keywords.html" \t "_blank)，保留了为以后可以用……

但这不是我要说的让你兴奋的内容。让你兴奋的是，你是可以用break、continue和有标签的代码块来实现goto的：

向前跳：

1. label: {
2. // do stuff
3. **if** (check) **break** label;
4. // do more stuff
5. }

对应的字节码是：

1. 2  iload\_1 [check]
2. 3  ifeq 6          // 向前跳
3. 6  ..

向后跳：

1. label: **do** {
2. // do stuff
3. **if** (check) **continue** label;
4. // do more stuff
5. **break** label;
6. } **while**(**true**);

对应的字节码是：

1. 2  iload\_1 [check]
2. 3  ifeq 9
3. 6  goto 2          // 向后跳
4. 9  ..

**8. Java是有类型别名的**

在别的语言中（比如，[Ceylon](http://blog.jooq.org/2013/12/03/top-10-ceylon-language-features-i-wish-we-had-in-java/" \t "_blank)）， 可以方便地定义类型别名：

1. **interface** People => Set<Person>;

这样定义的People可以和Set<Person>互换地使用：

1. People?      p1 = **null**;
2. Set<Person>? p2 = p1;
3. People?      p3 = p2;

在Java中不能在顶级（top level）定义类型别名。但可以在类级别、或方法级别定义。 如果对Integer、Long这样名字不满意，想更短的名字：I和L。很简单：

1. **class** Test<I **extends** Integer> {
2. <L **extends** Long> **void** x(I i, L l) {
3. System.out.println(
4. i.intValue() + ", " +
5. l.longValue()
6. );
7. }
8. }

上面的代码中，在Test类级别中I是Integer的『别名』，在x方法级别，L是Long的『别名』。可以这样来调用这个方法：

1. **new** Test().x(1, 2L);

当然这个用法不严谨。在例子中，Integer、Long都是final类型，结果I和L 效果上是个别名 （大部分情况下是。赋值兼容性只是单向的）。如果用非final类型（比如，Object），还是要使用原来的泛型参数类型。

玩够了这些恶心的小把戏。现在要上干货了！

**9. 有些类型的关系是不确定的**

好，这条会很稀奇古怪，你先来杯咖啡，再集中精神来看。看看下面的2个类型：

1. // 一个辅助类。也可以直接使用List
2. **interface** Type<T> {}
4. **class** C **implements** Type<Type<? **super** C>> {}
5. **class** D<P> **implements** Type<Type<? **super** D<D<P>>>> {}

类型C和D是啥意思呢？

这2个类型声明中包含了递归，和[java.lang.Enum](http://docs.oracle.com/javase/8/docs/api/java/lang/Enum.html" \t "_blank)的声明类似 （但有微妙的不同）：

1. **public** **abstract** **class** Enum<E **extends** Enum<E>> { ... }

有了上面的类型声明，一个实际的enum实现只是语法糖：

1. // 这样的声明
2. **enum** MyEnum {}
4. // 实际只是下面写法的语法糖：
5. **class** MyEnum **extends** Enum<MyEnum> { ... }

记住上面的这点后，回到我们的2个类型声明上。下面的代码可以编译通过吗？

1. **class** Test {
2. Type<? **super** C> c = **new** C();
3. Type<? **super** D<Byte>> d = **new** D<Byte>();
4. }

很难的问题，[Ross Tate](http://www.cs.cornell.edu/~ross/" \t "_blank)回答过这个问题。答案实际上是不确定的：

1. C是Type<? **super** C>的子类吗？
3. 步骤 0) C <?: Type<? **super** C>
4. 步骤 1) Type<Type<? **super** C>> <?: Type （继承）
5. 步骤 2) C （检查通配符 ? **super** C）
6. 步骤 . . . （进入死循环）

然后：

1. D是Type<? **super** D<Byte>>的子类吗？
3. 步骤 0) D<Byte> <?: Type<? **super** C<Byte>>
4. 步骤 1) Type<Type<? **super** D<D<Byte>>>> <?: Type<? **super** D<Byte>>
5. 步骤 2) D<Byte> <?: Type<? **super** D<D<Byte>>>
6. 步骤 3) List<List<? **super** C<C>>> <?: List<? **super** C<C>>
7. 步骤 4) D<D<Byte>> <?: Type<? **super** D<D<Byte>>>
8. 步骤 . . . （进入永远的展开中）

试着在你的[Eclipse](http://res.importnew.com/eclipse" \o "Eclipse ImportNew主页" \t "_blank)中编译上面的代码，会Crash！（别担心，我已经提交了一个Bug。）

我们继续深挖下去……

在Java中有些类型的关系是不确定的！

如果你有兴趣知道更多古怪Java行为的细节，可以读一下Ross Tate的论文[『驯服Java类型系统的通配符』](http://www.cs.cornell.edu/~ross/publications/tamewild/tamewild-tate-pldi11.pdf" \t "_blank) （由Ross Tate、Alan Leung和Sorin Lerner合著），或者也可以看看我们在[子类型多态和泛型多态的关联](http://blog.jooq.org/2013/06/28/the-dangers-of-correlating-subtype-polymorphism-with-generic-polymorphism/" \t "_blank)方面的思索。

**10. 类型交集（Type intersections）**

Java有个很古怪的特性叫类型交集。你可以声明一个（泛型）类型，这个类型是2个类型的交集。比如：

1. **class** Test<T **extends** Serializable & Cloneable> {
2. }

绑定到类Test的实例上的泛型类型参数T必须同时实现Serializable和Cloneable。比如，String不能做绑定，但Date可以：

1. // 编译不通过！
2. Test<String> s = **null**;
4. // 编译通过
5. Test<Date> d = **null**;

Java 8保留了这个特性，你可以转型成临时的类型交集。这有什么用？ 几乎没有一点用，但如果你想强转一个lambda表达式成这样的一个类型，就没有其它的方法了。 假定你在方法上有了这个蛋疼的类型限制：

1. <T **extends** Runnable & Serializable> **void** execute(T t) {}

你想一个Runnable同时也是个Serializable，这样你可能在另外的地方执行它并通过网络发送它。lambda和序列化都有点古怪。

lambda是可以序列化的：

如果lambda表达式的目标类型和它捕获的参数（captured arguments）是可以序列化的，则这个lambda表达式是可序列化的。

但即使满足这个条件，lambda表达式并没有自动实现Serializable这个标记接口（marker interface）。 为了强制成为这个类型，就必须使用转型。但如果只转型成Serializable …

1. execute((Serializable) (() -> {}));

… 则这个lambda表达式不再是一个Runnable。

呃……

So……

同时转型成2个类型：

1. execute((Runnable & Serializable) (() -> {}));

**结论**

一般我只对SQL会说这样的话，但是时候用下面的话来结束这篇文章了：

Java中包含的诡异在程度上仅仅被它解决问题的能力超过。

原文链接： [Jooq](http://blog.jooq.org/2014/11/03/10-things-you-didnt-know-about-java/) 翻译： [ImportNew.com](http://www.importnew.com/)- [Jerry Lee](http://www.importnew.com/author/oldratlee)  
译文链接： <http://www.importnew.com/13859.html>

【编辑推荐】

1. [Java 中常用缓存Cache机制的实现](http://developer.51cto.com/art/201411/456219.htm)
2. [Java开发者值得关注的7款新工具](http://developer.51cto.com/art/201411/456857.htm)
3. [5个强大的Java分布式缓存框架推荐](http://developer.51cto.com/art/201411/457423.htm)