Java 语法糖详解


(https://gitbook.cn/gitchat/author/591131ebd7e8c528294a0944)
Hollis (https://gitbook.c…
向作者提问 (https://gitbook.cn/m/mazi/author/591131ebd7e8c528294a0944/question)
HollisChuang's Blog博主，Hollis公众号负责人。Java工程师成神之路系列作者。

查看本场Chat
(https://gitbook.cn/gitchat/activity/5a5d55d36f3da41fa892ef09)

语法糖

语法糖（Syntactic Sugar），也称糖衣语法，是由英国计算机学家 Peter.J.Landin 发明的一个术语，指在计算机语言中添加的某种语法，这种语法对语言的功能并没有影响，但是更方便程序员使用。简而言之，语法糖让程序更加简洁，有更高的可读性。

> 有意思的是，在编程领域，除了语法糖，还有语法盐和语法糖精的说法，篇幅有限这里不做扩展了。

我们所熟知的编程语言中几乎都有语法糖。作者认为，语法糖的多少是评判一个语言够不够牛逼的标准之一。很多人说Java是一个“低糖语言”，其实从Java 7开始Java语言层面上一直在添加各种糖，主要是在“Project Coin”项目下研发。尽管现在Java有人还是认为现在的Java是低糖，未来还会持续向着“高糖”的方向发展。

解语法糖

前面提到过，语法糖的存在主要是方便开发人员使用。但其实，Java虚拟机并不支持这些语法糖，这些语法糖在编译阶段就会被还原成简单的基础语法结构，这个过程就是解语法糖。

说到编译，大家肯定都知道，Java语言中，`javac` 命令可以将后缀名为 `.java` 的源文件编译为后缀名为 `.class` 的可以运行于Java虚拟机的字节码。如果你去看 `com.sun.tools.javac.main.JavaCompiler` 的源码，你会发现在 `compile()` 中有一个步骤就是调用 `desugar()`，这个方法就是负责解语法糖的实现的。

Java 中最常用的语法糖主要有泛型、变长参数、条件编译、自动拆装箱、内部类等。本文主要来分析下这些语法糖背后的原理，一步一步剥去糖衣，看看其本质。

糖块一、 switch 支持 String 与枚举

前面提到过，从Java 7 开始，Java语言中的语法糖在逐渐丰富，其中一个比较重要的就是Java 7中 `switch` 开始支持 `String`。

在开始coding之前先科普下，Java中的 `swith` 自身原本就支持基本类型。比如 `int`、 `char` 等。对于 `int` 类型，直接进行数值的比较。对于 `char` 类型则是比较其ascii码。所以，对于编译器来说， `switch` 中其实只能使用整型，任何类型的比较都要转换成整型。比如 `byte`、 `short`、 `char`（ackii码是整型）以及 `int`。

那么接下来看下 `switch` 对 `String` 得支持，有以下代码：

```
public class switchDemoString {
    public static void main(String[] args) {
        String str = "world";
        switch (str) {
        case "hello":
            System.out.println("hello");
            break;
        case "world":
            System.out.println("world");
            break;
        default:
            break;
        }
    }
}
```

反编译 (http://www.hollischuang.com/archives/58)后内容如下：

```
public class switchDemoString
{
    public switchDemoString()
    {
    }
    public static void main(String args[])
    {
        String str = "world";
        String s;
        switch((s = str).hashCode())
        {
        default:
            break;
        case 99162322:
            if(s.equals("hello"))
                System.out.println("hello");
            break;
        case 113318802:
            if(s.equals("world"))
                System.out.println("world");
            break;
        }
    }
}
```

看到这个代码，你知道原来**字符串的switch是通过 `equals()` 和 `hashCode()` 方法来实现的。**还好 `hashCode()` 方法返回的是 `int` ，而不是 `long` 。

> 仔细看下可以发现，进行 `switch` 的实际是哈希值，然后通过使用 `equals` 方法比较进行安全检查，这个检查是必要的，因为哈希可能会发生碰撞。因此它的性能是不如使用枚举进行switch或者使用纯整数常量，但这也不是很差。

糖块二、泛型

我们都知道，很多语言都是支持泛型的，但是很多人不知道的是，不同的编译器对于泛型的处理方式是不同的，通常情况下，一个编译器处理泛型有两种方式： `Code specialization` 和 `Code sharing` 。C++和C#是使用 `Code specialization` 的处理机制，而Java使用的是 `Code sharing` 的机制。

> Code sharing方式为每个泛型类型创建唯一的字节码表示，并且将该泛型类型的实例都映射到这个唯一的字节码表示上。将多种泛型类形实例映射到唯一的字节码表示是通过类型擦除（ `type erasue` ）实现的。

也就是说，对于Java虚拟机来说，他根本不认识 `Map<String, String> map` 这样的语法。需要在编译阶段通过类型擦除的方式进行解语法糖。

类型擦除的主要过程如下：

1. 将所有的泛型参数用其最左边界（最顶级的父类型）类型替换。
2. 移除所有的类型参数。

以下代码：

```
Map<String, String> map = new HashMap<String, String>();
map.put("name", "hollis");
map.put("wechat", "Hollis");
map.put("blog", "www.hollischuang.com");
```

解语法糖之后会变成：

```
Map map = new HashMap();
map.put("name", "hollis");
map.put("wechat", "Hollis");
map.put("blog", "www.hollischuang.com");
```

以下代码：

```
public static <A extends Comparable<A>> A max(Collection<A> xs) {
    Iterator<A> xi = xs.iterator();
    A w = xi.next();
    while (xi.hasNext()) {
        A x = xi.next();
        if (w.compareTo(x) < 0)
            w = x;
    }
    return w;
}
```

类型擦除后会变成：

```
 public static Comparable max(Collection xs){
    Iterator xi = xs.iterator();
    Comparable w = (Comparable)xi.next();
    while(xi.hasNext())
    {
        Comparable x = (Comparable)xi.next();
        if(w.compareTo(x) < 0)
            w = x;
    }
    return w;
}
```

虚拟机中没有泛型，只有普通类和普通方法，所有泛型类的类型参数在编译时都会被擦除，泛型类并没有自己独有的 `Class` 类对象。比如并不存在 `List<String>.class` 或是 `List<Integer>.class`，而只有 `List.class`。

糖块三、 自动装箱与拆箱

自动装箱就是Java自动将原始类型值转换成对应的对象，比如将int的变量转换成Integer对象，这个过程叫做装箱，反之将Integer对象转换成int类型值，这个过程叫做拆箱。因为这里的装箱和拆箱是自动进行的非人为转换，所以就称作为自动装箱和拆箱，原始类型byte、short、char、int、long、float、double 和 boolean 对应的封装类为Byte、Short、Character、Integer、Long、Float、Double、Boolean。

先来看个自动装箱的代码：

```
 public static void main(String[] args) {
    int i = 10;
    Integer n = i;
}
```

反编译后代码如下:

```
public static void main(String args[])
{
    int i = 10;
    Integer n = Integer.valueOf(i);
}
```

再来看个自动拆箱的代码：

```
public static void main(String[] args) {

    Integer i = 10;
    int n = i;
}
```

反编译后代码如下：

```
public static void main(String args[])
{
    Integer i = Integer.valueOf(10);
    int n = i.intValue();
}
```

从反编译得到内容可以看出，在装箱的时候自动调用的是 `Integer` 的 `valueOf(int)` 方法。而在拆箱的时候自动调用的是 `Integer` 的 `intValue` 方法。

所以，**装箱过程是通过调用包装器的valueOf方法实现的，而拆箱过程是通过调用包装器的xxxValue方法实现的。**

糖块四 、 方法变长参数

可变参数(`variable arguments`)是在Java 1.5中引入的一个特性，它允许一个方法把任意数量的值作为参数。

看下以下可变参数代码，其中print方法接收可变参数：

```
public static void main(String[] args)
    {
        print("Holis", "公众号:Hollis", "博客：www.hollischuang.com", "QQ：907
607222");
    }

public static void print(String... strs)
{
    for (int i = 0; i < strs.length; i++)
    {
        System.out.println(strs[i]);
    }
}
```

反编译后代码：

```
 public static void main(String args[])
{
    print(new String[] {
        "Holis", "\u516C\u4F17\u53F7:Hollis", "\u535A\u5BA2\uFF1Awww.hollisc
huang.com", "QQ\uFF1A907607222"
    });
}

public static transient void print(String strs[])
{
    for(int i = 0; i < strs.length; i++)
        System.out.println(strs[i]);

}
```

从反编译后代码可以看出，可变参数在被使用的时候，他首先会创建一个数组，数组的长度就是调用该方法是传递的实参的个数，然后再把参数值全部放到这个数组当中，然后再把这个数组作为参数传递到被调用的方法中。

> PS：反编译后的print方法声明中有一个transient标识，是不是很奇怪？transient不是不可以修饰方法吗？transient不是和序列化有关么？transient在这里的作用是什么？因为这个与本文关系不大，这里不做深入分析了。相了解的同学可以关注我微信公众号或者博客。

糖块五 、 枚举

Java SE5提供了一种新的类型-Java的枚举类型，关键字 enum 可以将一组具名的值的有限集合创建为一种新的类型，而这些具名的值可以作为常规的程序组件使用，这是一种非常有用的功能。

要想看源码，首先得有一个类吧，那么枚举类型到底是什么类呢？是 enum 吗？答案很明显不是， enum 就和 class 一样，只是一个关键字，他并不是一个类，那么枚举是由什么类维护的呢，我们简单的写一个枚举：

```
public enum t {
    SPRING,SUMMER;
}
```

然后我们使用反编译，看看这段代码到底是怎么实现的，反编译后代码内容如下：

```
public final class T extends Enum
{
    private T(String s, int i)
    {
        super(s, i);
    }
    public static T[] values()
    {
        T at[];
        int i;
        T at1[];
        System.arraycopy(at = ENUM$VALUES, 0, at1 = new T[i = at.length], 0,
 i);
        return at1;
    }

    public static T valueOf(String s)
    {
        return (T)Enum.valueOf(demo/T, s);
    }

    public static final T SPRING;
    public static final T SUMMER;
    private static final T ENUM$VALUES[];
    static
    {
        SPRING = new T("SPRING", 0);
        SUMMER = new T("SUMMER", 1);
        ENUM$VALUES = (new T[] {
            SPRING, SUMMER
        });
    }
}
```

通过反编译后代码我们可以看到， public final class T extends Enum ，说明，该类是继承了 Enum 类的，同时 final 关键字告诉我们，这个类也是不能被继承的。**当我们使用 enmu 来定义一个枚举类型的时候，编译器会自动帮我们创建一个 final 类型的类继承 Enum 类，所以枚举类型不能被继承。**

糖块六 、 内部类

内部类又称为嵌套类，可以把内部类理解为外部类的一个普通成员。

内部类之所以也是语法糖，是因为它仅仅是一个编译时的概念， outer.java 里面定义了一个内部类 inner ，一旦编译成功，就会生成两个完全不同的 .class 文件了，分别是 outer.class 和 outer$inner.class 。所以内部类的名字完全可以和它的外部类名字相同。

```
public class OutterClass {
    private String userName;

    public String getUserName() {
        return userName;
    }

    public void setUserName(String userName) {
        this.userName = userName;
    }

    public static void main(String[] args) {

    }

    class InnerClass{
        private String name;

        public String getName() {
            return name;
        }

        public void setName(String name) {
            this.name = name;
        }
    }
}
```

以上代码编译后会生成两个class文件： `OutterClass$InnerClass.class` 、 `OutterClass.class` 。当我们尝试对 `OutterClass.class` 文件进行反编译的时候，命令行会打印以下内容： `Parsing OutterClass.class...Parsing inner class OutterClass$InnerClass.class... Generating OutterClass.jad` 。他会把两个文件全部进行反编译，然后一起生成一个 `OutterClass.jad` 文件。文件内容如下：

```
public class OutterClass
{
    class InnerClass
    {
        public String getName()
        {
            return name;
        }
        public void setName(String name)
        {
            this.name = name;
        }
        private String name;
        final OutterClass this$0;

        InnerClass()
        {
            this.this$0 = OutterClass.this;
            super();
        }
    }

    public OutterClass()
    {
    }
    public String getUserName()
    {
        return userName;
    }
    public void setUserName(String userName){
        this.userName = userName;
    }
    public static void main(String args1[])
    {
    }
    private String userName;
}
```

糖块七、条件编译

一般情况下，程序中的每一行代码都要参加编译。但有时候出于对程序代码优化的考虑，希望只对其中一部分内容进行编译，此时就需要在程序中加上条件，让编译器只对满足条件的代码进行编译，将不满足条件的代码舍弃，这就是条件编译。

如在C或CPP中，可以通过预处理语句来实现条件编译。其实在Java中也可实现条件编译。我们先来看一段代码：

```
public class ConditionalCompilation {
    public static void main(String[] args) {
        final boolean DEBUG = true;
        if(DEBUG) {
            System.out.println("Hello, DEBUG!");
        }

        final boolean ONLINE = false;

        if(ONLINE){
            System.out.println("Hello, ONLINE!");
        }
    }
}
```

反编译后代码如下：

```
public class ConditionalCompilation
{

    public ConditionalCompilation()
    {
    }

    public static void main(String args[])
    {
        boolean DEBUG = true;
        System.out.println("Hello, DEBUG!");
        boolean ONLINE = false;
    }
}
```

首先，我们发现，在反编译后的代码中没有 `System.out.println("Hello, ONLINE!");`，这其实就是条件编译。当 `if(ONLINE)` 为false的时候，编译器就没有对其内的代码进行编译。

所以，**Java语法的条件编译，是通过判断条件为常量的if语句实现的。其原理也是Java语言的语法糖。根据if判断条件的真假，编译器直接把分支为false的代码块消除。通过该方式实现的条件编译，必须在方法体内实现，而无法在正整个Java类的结构或者类的属性上进行条件编译，这与C/C++的条件编译相比，确实更有局限性。在Java语言设计之初并没有引入条件编译的功能，虽有局限，但是总比没有更强。**

糖块八、断言

在Java中， `assert` 关键字是从 Java SE 1.4 引入的，为了避免和老版本的Java代码中使用了 `assert` 关键字导致错误，Java在执行的时候默认是不启动断言检查的（这个时候，所有的断言语句都将忽略！），如果要开启断言检查，则需要用开关 `-enableassertions` 或 `-ea` 来开启。

看一段包含断言的代码：

```
public class AssertTest {
    public static void main(String args[]) {
        int a = 1;
        int b = 1;
        assert a == b;
        System.out.println("公众号：Hollis");
        assert a != b : "Hollis";
        System.out.println("博客：www.hollischuang.com");
    }
}
```

反编译后代码如下：

```
public class AssertTest {
   public AssertTest()
    {
    }
    public static void main(String args[])
{
    int a = 1;
    int b = 1;
    if(!$assertionsDisabled && a != b)
        throw new AssertionError();
    System.out.println("\u516C\u4F17\u53F7\uFF1AHollis");
    if(!$assertionsDisabled && a == b)
    {
        throw new AssertionError("Hollis");
    } else
    {
        System.out.println("\u535A\u5BA2\uFF1Awww.hollischuang.com");
        return;
    }
}

static final boolean $assertionsDisabled = !com/hollis/suguar/AssertTest.des
iredAssertionStatus();


}
```

很明显，反编译之后的代码要比我们自己的代码复杂的多。所以，使用了assert这个语法糖我们节省了很多代码。**其实断言的底层实现就是if语言，如果断言结果为true，则什么都不做，程序继续执行，如果断言结果为false，则程序抛出AssertError来打断程序的执行。** -enableassertions 会设置 $assertionsDisabled 字段的值。

糖块九 、 数值字面量

在Java 7中，数值字面量，不管是整数还是浮点数，都允许在数字之间插入任意多个下划线。这些下划线不会对字面量的数值产生影响，目的就是方便阅读。

比如：

```
public class Test {
    public static void main(String... args) {
        int i = 10_000;
        System.out.println(i);
    }
}
```

反编译后：

```
public class Test
{
  public static void main(String[] args)
  {
    int i = 10000;
    System.out.println(i);
  }
}
```

反编译后就是把 _ 删除了。也就是说 **编译器并不认识在数字字面量中的 _ ，需要在编译阶段把他去掉。**

糖块十 、 for-each

增强for循环（ for-each ）相信大家都不陌生，日常开发经常会用到的，他会比for循环要少写很多代码，那么这个语法糖背后是如何实现的呢？

```
public static void main(String... args) {
    String[] strs = {"Hollis", "公众号：Hollis", "博客：www.hollischuang.com"}
;
    for (String s : strs) {
        System.out.println(s);
    }
    List<String> strList = ImmutableList.of("Hollis", "公众号：Hollis", "博客
: www.hollischuang.com");
    for (String s : strList) {
        System.out.println(s);
    }
}
```

反编译后代码如下：

```
public static transient void main(String args[])
{
    String strs[] = {
        "Hollis", "\u516C\u4F17\u53F7\uFF1AHollis", "\u535A\u5BA2\uFF1Awww.h
ollischuang.com"
    };
    String args1[] = strs;
    int i = args1.length;
    for(int j = 0; j < i; j++)
    {
        String s = args1[j];
        System.out.println(s);
    }

    List strList = ImmutableList.of("Hollis", "\u516C\u4F17\u53F7\uFF1AHolli
s", "\u535A\u5BA2\uFF1Awww.hollischuang.com");
    String s;
    for(Iterator iterator = strList.iterator(); iterator.hasNext(); System.o
ut.println(s))
        s = (String)iterator.next();

}
```

代码很简单，**for-each的实现原理其实就是使用了普通的for循环和迭代器。**

糖块十一 、 try-with-resource

Java里，对于文件操作IO流、数据库连接等开销非常昂贵的资源，用完之后必须及时通过close方法将其关闭，否则资源会一直处于打开状态，可能会导致内存泄露等问题。

关闭资源的常用方式就是在 `finally` 块里是释放，即调用 `close` 方法。比如，我们经常会写这样的代码：

```
public static void main(String[] args) {
    BufferedReader br = null;
    try {
        String line;
        br = new BufferedReader(new FileReader("d:\\hollischuang.xml"));
        while ((line = br.readLine()) != null) {
            System.out.println(line);
        }
    } catch (IOException e) {
        // handle exception
    } finally {
        try {
            if (br != null) {
                br.close();
            }
        } catch (IOException ex) {
            // handle exception
        }
    }
}
```

从Java 7开始，jdk提供了一种更好的方式关闭资源，使用 `try-with-resources` 语句，改写一下上面的代码，效果如下：

```
public static void main(String... args) {
    try (BufferedReader br = new BufferedReader(new FileReader("d:\\ hollisc
huang.xml"))) {
        String line;
        while ((line = br.readLine()) != null) {
            System.out.println(line);
        }
    } catch (IOException e) {
        // handle exception
    }
}
```

看，这简直是一大福音啊，虽然我之前一般使用 `IOUtils` 去关闭流，并不会使用在 `finally` 中写很多代码的方式，但是这种新的语法糖看上去好像优雅很多呢。看下他的背后：

```
public static transient void main(String args[])
    {
        BufferedReader br;
        Throwable throwable;
        br = new BufferedReader(new FileReader("d:\\ hollischuang.xml"));
        throwable = null;
        String line;
        try
        {
            while((line = br.readLine()) != null)
                System.out.println(line);
        }
        catch(Throwable throwable2)
        {
            throwable = throwable2;
            throw throwable2;
        }
        if(br != null)
            if(throwable != null)
                try
                {
                    br.close();
                }
                catch(Throwable throwable1)
                {
                    throwable.addSuppressed(throwable1);
                }
            else
                br.close();
            break MISSING_BLOCK_LABEL_113;
            Exception exception;
            exception;
            if(br != null)
                if(throwable != null)
                    try
                    {
                        br.close();
                    }
                    catch(Throwable throwable3)
                      {
                        throwable.addSuppressed(throwable3);
                    }
                else
                    br.close();
        throw exception;
        IOException ioexception;
        ioexception;
    }
}
```

其实背后的原理也很简单，那些我们没有做的关闭资源的操作，编译器都帮我们做了。所以，再次印证了，语法糖的作用就是方便程序员的使用，但最终还是要转成编译器认识的语言。

糖块十二、Lambda表达式

关于lambda表达式，有人可能会有质疑，因为网上有人说他并不是语法糖。其实我想纠正下这个说法。**Labmda表达式不是匿名内部类的语法糖，但是他也是一个语法糖。实现方式其实是依赖了几个JVM底层提供的lambda相关api。**

先来看一个简单的lambda表达式。遍历一个list：

```
public static void main(String... args) {
    List<String> strList = ImmutableList.of("Hollis", "公众号：Hollis", "博客
: www.hollischuang.com");

    strList.forEach( s -> { System.out.println(s); } );
}
```

为啥说他并不是内部类的语法糖呢，前面讲内部类我们说过，内部类在编译之后会有两个class文件，但是，包含lambda表达式的类编译后只有一个文件。

反编译后代码如下:

```
public static /* varargs */ void main(String ... args) {
    ImmutableList strList = ImmutableList.of((Object)"Hollis", (Object)"\u51
6c\u4f17\u53f7\uff1aHollis", (Object)"\u535a\u5ba2\uff1awww.hollischuang.com"
);
    strList.forEach((Consumer<String>)LambdaMetafactory.metafactory(null, nu
ll, null, (Ljava/lang/Object;)V, lambda$main$0(java.lang.String ), (Ljava/la
ng/String;)V)());
}

private static /* synthetic */ void lambda$main$0(String s) {
    System.out.println(s);
}
```

可以看到，在 `forEach` 方法中，其实是调用了 `java.lang.invoke.LambdaMetafactory#metafactory` 方法，该方法的第四个参数implMethod指定了方法实现。可以看到这里其实是调用了一个 `lambda$main$0` 方法进行了输出。

再来看一个稍微复杂一点的，先对List进行过滤，然后再输出：

```
public static void main(String... args) {
    List<String> strList = ImmutableList.of("Hollis", "公众号：Hollis", "博客
: www.hollischuang.com");

    List HollisList = strList.stream().filter(string -> string.contains("Hol
lis")).collect(Collectors.toList());

    HollisList.forEach( s -> { System.out.println(s); } );
}
```

反编译后代码如下：

```
public static /* varargs */ void main(String ... args) {
    ImmutableList strList = ImmutableList.of((Object)"Hollis", (Object)"\u51
6c\u4f17\u53f7\uff1aHollis", (Object)"\u535a\u5ba2\uff1awww.hollischuang.com"
);
    List<Object> HollisList = strList.stream().filter((Predicate<String>)Lam
bdaMetafactory.metafactory(null, null, null, (Ljava/lang/Object;)Z, lambda$m
ain$0(java.lang.String ), (Ljava/lang/String;)Z)()).collect(Collectors.toLis
t());
    HollisList.forEach((Consumer<Object>)LambdaMetafactory.metafactory(null,
null, null, (Ljava/lang/Object;)V, lambda$main$1(java.lang.Object ), (Ljava/
lang/Object;)V)());
}

private static /* synthetic */ void lambda$main$1(Object s) {
    System.out.println(s);
}

private static /* synthetic */ boolean lambda$main$0(String string) {
    return string.contains("Hollis");
}
```

两个lambda表达式分别调用了 `lambda$main$1` 和 `lambda$main$0` 两个方法。

所以，lambda表达式的实现其实是依赖了一些底层的api，在编译阶段，编译器会把lambda表达式进行解糖，转换成调用内部api的方式。

可能遇到的坑

泛型

一、当泛型遇到重载

```
public class GenericTypes {

    public static void method(List<String> list) {
        System.out.println("invoke method(List<String> list)");
    }

    public static void method(List<Integer> list) {
        System.out.println("invoke method(List<Integer> list)");
    }
}
```

上面这段代码，有两个重载的函数，因为他们的参数类型不同，一个是List<String>另一个是List<Integer>，但是，这段代码是编译通不过的。因为我们前面讲过，参数List<Integer>和List<String>编译之后都被擦除了，变成了一样的原生类型List，擦除动作导致这两个方法的特征签名变得一模一样。

二、当泛型遇到catch

泛型的类型参数不能用在Java异常处理的catch语句中。因为异常处理是由JVM在运行时刻来进行的。由于类型信息被擦除，JVM是无法区分两个异常类型 `MyException<String>` 和 `MyException<Integer>` 的

三、当泛型内包含静态变量

```
public class StaticTest{
    public static void main(String[] args){
        GT<Integer> gti = new GT<Integer>();
        gti.var=1;
        GT<String> gts = new GT<String>();
        gts.var=2;
        System.out.println(gti.var);
    }
}
class GT<T>{
    public static int var=0;
    public void nothing(T x){}
}
```

以上代码输出结果为：2！由于经过类型擦除，所有的泛型类实例都关联到同一份字节码上，泛型类的所有静态变量是共享的。

自动装箱与拆箱

对象相等比较

```
public class BoxingTest {

    public static void main(String[] args) {
        Integer a = 1000;
        Integer b = 1000;
        Integer c = 100;
        Integer d = 100;
        System.out.println("a == b is " + (a == b));
        System.out.println(("c == d is " + (c == d)));
    }
```

输出结果：

```
a == b is false
c == d is true
```

在Java 5中，在Integer的操作上引入了一个新功能来节省内存和提高性能。整型对象通过使用相同的对象引用实现了缓存和重用。

> 适用于整数值区间-128 至 +127。
>
> 只适用于自动装箱。使用构造函数创建对象不适用。

增强for循环

ConcurrentModificationException

```
for (Student stu : students) {
    if (stu.getId() == 2)
        students.remove(stu);
}
```

会抛出 `ConcurrentModificationException` 异常。

Iterator是工作在一个独立的线程中，并且拥有一个 mutex 锁。 Iterator被创建之后会建立一个指向原来对象的单链索引表，当原来的对象数量发生变化时，这个索引表的内容不会同步改变，所以当索引指针往后移动的时候就找不到要迭代的对象，所以按照 fail-fast 原则 Iterator 会马上抛出 `java.util.ConcurrentModificationException` 异常。

所以 `Iterator` 在工作的时候是不允许被迭代的对象被改变的。但你可以使用 `Iterator` 本身的方法 `remove()` 来删除对象， `Iterator.remove()` 方法会在删除当前迭代对象的同时维护索引的一致性。

总结

前面介绍了12种Java中常用的语法糖。所谓语法糖就是提供给开发人员便于开发的一种语法而已。但是这种语法只有开发人员认识。要想被执行，需要进行解糖，即转成JVM认识的语法。当我们把语法糖解糖之后，你就会发现其实我们日常使用的这些方便的语法，其实都是一些其他更简单的语法构成的。

有了这些语法糖，我们在日常开发的时候可以大大提升效率，但是同时也要避免过渡使用。使用之前最好了解下原理，避免掉坑。

参考资料：

- Java的反编译 (http://www.hollischuang.com/archives/58)
- Java中的Switch对整型、字符型、字符串型的具体实现细节 (http://www.hollischuang.com/archives/61)
- 深度分析Java的枚举类型—-枚举的线程安全性及序列化问题 (http://www.hollischuang.com/archives/197)
- Java的枚举类型用法介绍 (http://www.hollischuang.com/archives/195)
- Java中的增强for循环（for each）的实现原理与坑 (http://www.hollischuang.com/archives/1776)
- Java中泛型的理解 (http://www.hollischuang.com/archives/230)
- Java中整型的缓存机制 (http://www.hollischuang.com/archives/1174)
- Java中的可变参数 (http://www.hollischuang.com/archives/1271)

(全文完)

欢迎关注Hollis维系公众号

---

8 (https://gitbook.cn/books/5a61532299d4bd32a9872506/index.html#) 0

(https://gitbook.cn/books/5a61532299d4bd32a9872506/index.html#)

写评论

向作者提问 (https://gitbook.cn/m/mazi/author/591131ebd7e8c528294a

姚小焕

棒棒的，很用心

2月4日 2 0

小哼哼呀、

很详细，这8.8花的值。

2月4日 1 0

IF

受教了

3月6日 0 0

Maps

作者文笔不错，逻辑清晰，表达明确，棒棒哒！

4月27日 0 0