# 基本概念

Lombok 是一种Java实用工具，可用来帮助开发人员消除 Java 的冗长，尤其是对于简单的 Java 对象（POJO），它通过注释实现这一目的。通过在开发环境中实现Lombok，开发人员可以节省构建诸如hashCode()和equals()、getter / setter 这样的方法以及以往用来分类各种 accessor 和 mutator 的大量时间。

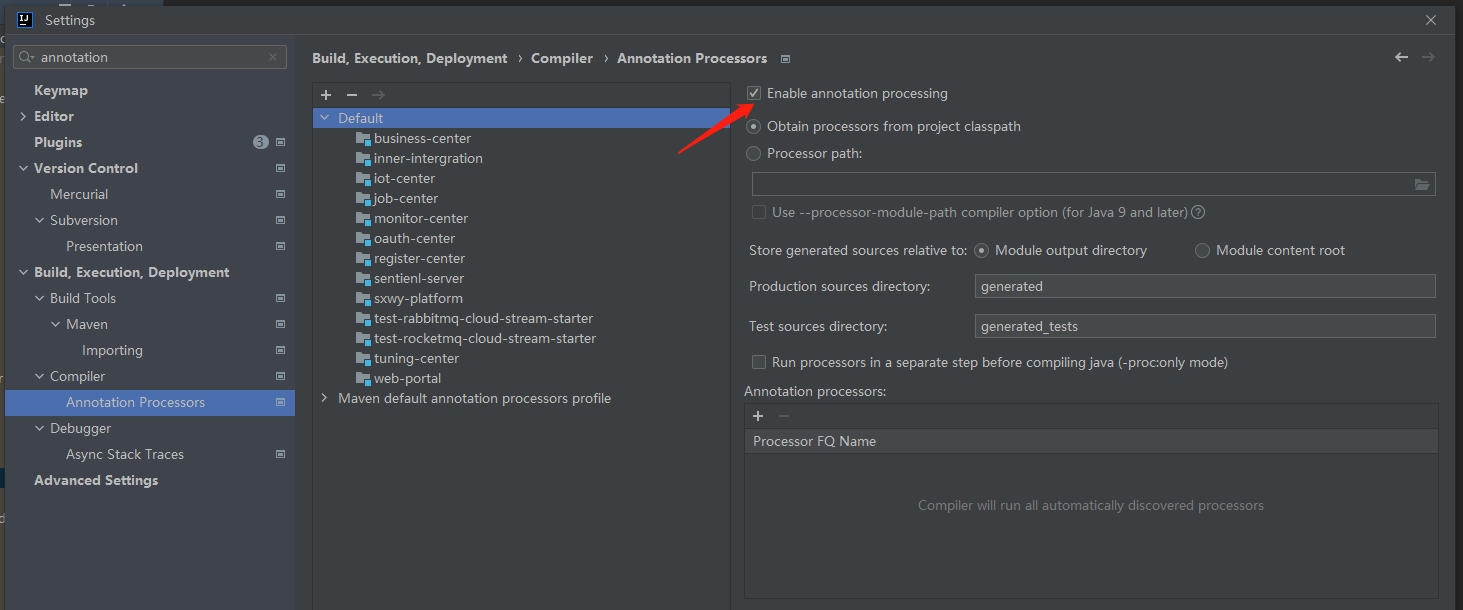
# 安装

使 IntelliJ IDEA 支持 Lombok 方式如下：

## 1、Intellij 设置支持注解处理

点击 File > Settings > Build > Annotation Processors

勾选 Enable annotation processing



## 2、安装插件

点击 Settings > Plugins > Browse repositories

查找 Lombok Plugin 并进行安装

重启 IntelliJ IDEA

## 3、将 lombok 添加到 pom 文件



# 注解

## @Data

### 基本概念

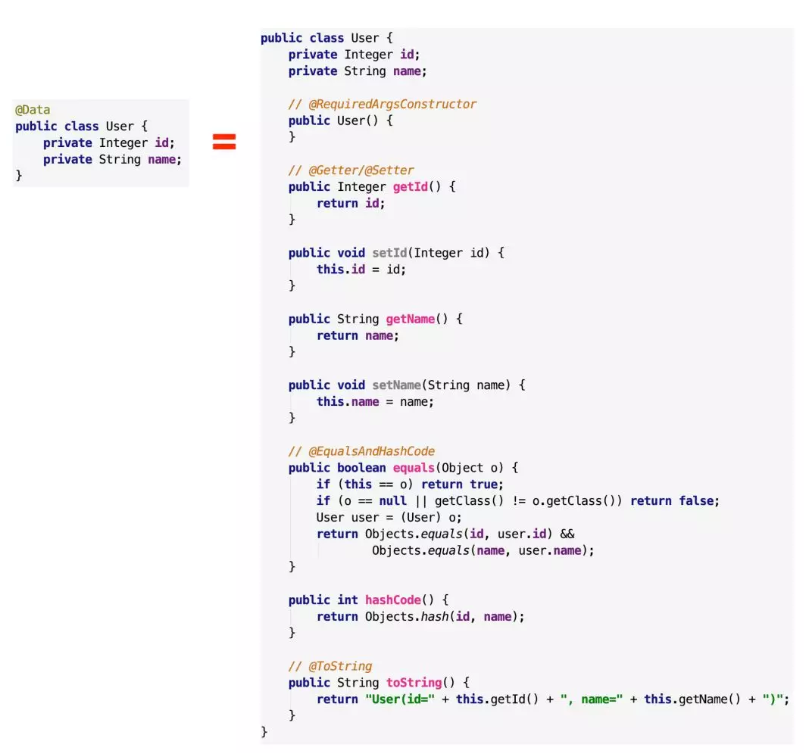
整合包，只要加了 @Data 这个注解，等于同时加了以下注解

@Getter/@Setter

@ToString

@EqualsAndHashCode

@RequiredArgsConstructor



### 说明

通常@Data注解会加在一个值可以被更新的对象上，像是日常使用的 DTO 们、或是 JPA 里的 Entity 们，就很适合加上@Data 注解，也就是@Data for mutable class。

## @Value

### 基本概念

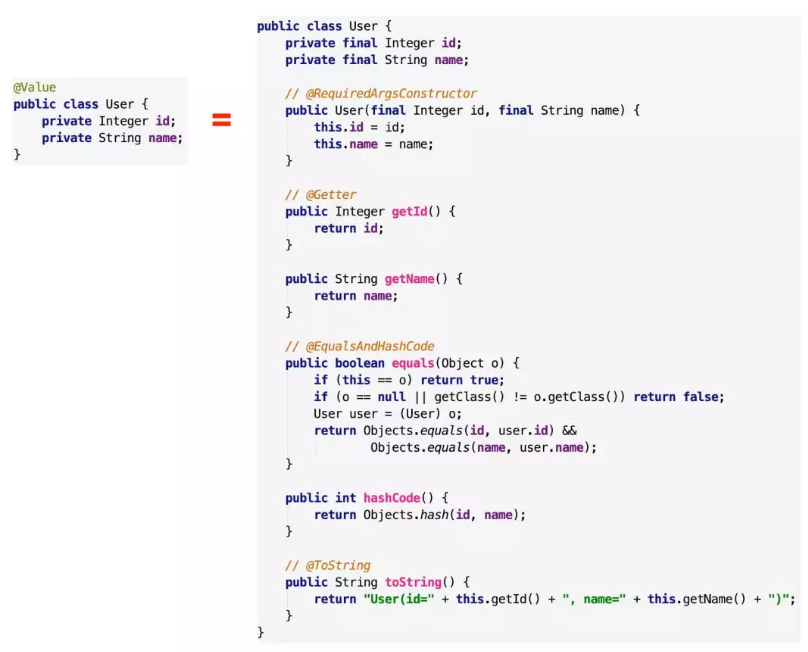
也是整合包，但是他会把所有的变量都设成 final 的，其他的就跟 @Data 一样，等于同时加了以下注解

@Getter (注意没有setter)

@ToString

@EqualsAndHashCode

@RequiredArgsConstructor



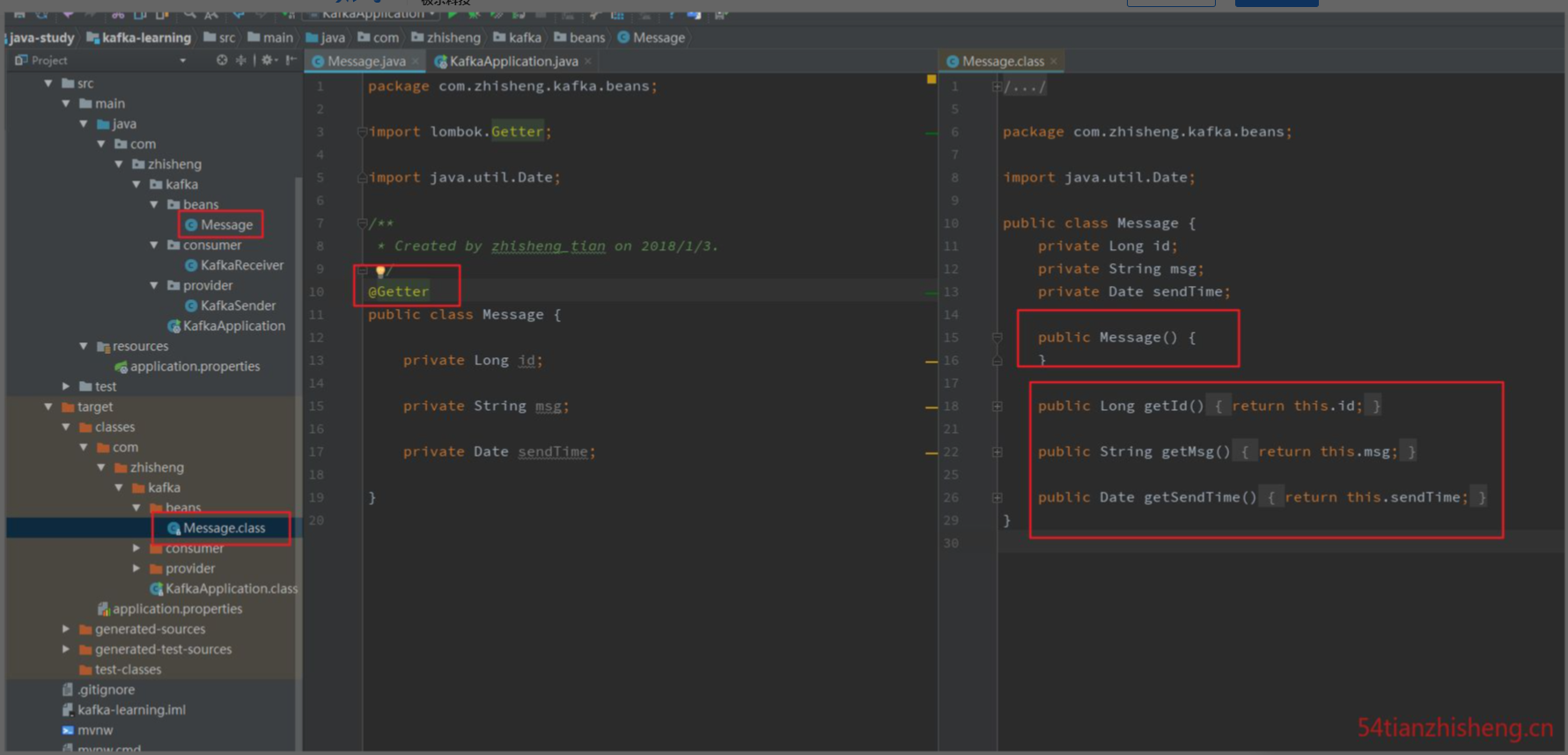
### 说明

这个 @Value 注解，则是适合加在值不希望被改变的类上，像是某个类的值当创建后就不希望被更改，只希望我们读它而已，就适合加上 @Value 注解，也就是 @Value for immutable class。另外注意一下，此 lombok 的注解 @Value 和另一个 Spring 的注解 @Value 撞名，在 import 时不要 import 错了。

## @Getter

### 基本概念

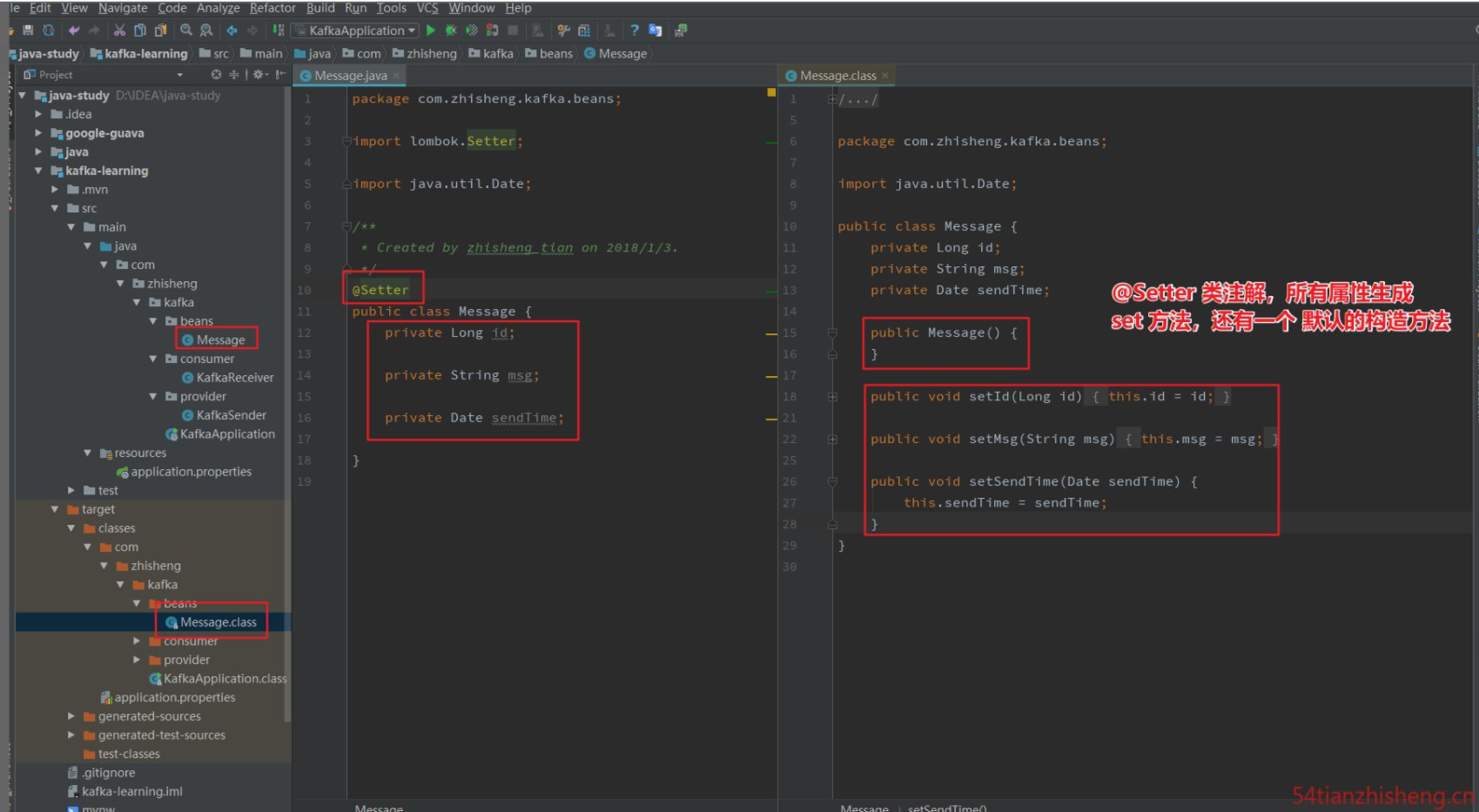
注解在属性上，为单个属性提供 get 方法，注解在类上，为该类所有的属性提供 get 方法，都提供默认构造方法。



## @Setter

### 基本概念

注解在属性上，为单个属性提供 set 方法，注解在类上，为该类所有的属性提供 set 方法， 都提供默认构造方法。



## @RequiredArgsConstructor

### 基本概念

它用于自动生成一个包含必需参数的构造方法。使用 @RequiredArgsConstructor 注解可以简化代码编写，减少不必要的样板代码。当在一个类上使用 @RequiredArgsConstructor 注解时，Lombok 会在编译时自动生成一个构造方法，该构造方法包含标记为 final 或者被 @NonNull 注解修饰的成员变量作为参数。生成的构造方法会初始化这些参数，并按照它们在类中的声明顺序进行排序。

### 好处

#### 1、简化构造方法的编写

不再需要手动编写包含必需参数的构造方法。

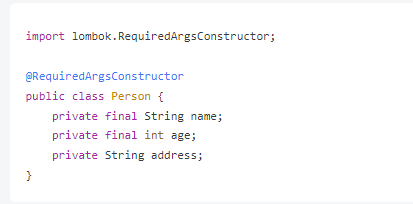
#### 2、提高代码可读性

通过自动生成的构造方法，可以清晰地表达一个类所需的必要参数。

#### 3、强制指定必需参数

通过使用@NonNull注解标记参数，生成的构造方法会对这些参数进行非空检查。

### 示例



在上述示例中，Person 类使用了 @RequiredArgsConstructor 注解。由于name 和 age 成员变量被标记为 final，因此 Lombok 会自动生成一个构造方法，该构造方法接受 name 和 age 作为参数，并进行初始化。address 字段没有被包含在生成的构造方法中，因为它没有被标记为 final。

需要注意的是，@RequiredArgsConstructor 只会生成包含必需参数的构造方法，如果类中已经存在其他构造方法或者某些成员变量没有被标记为 final 或者 @NonNull，则不会生成构造方法。

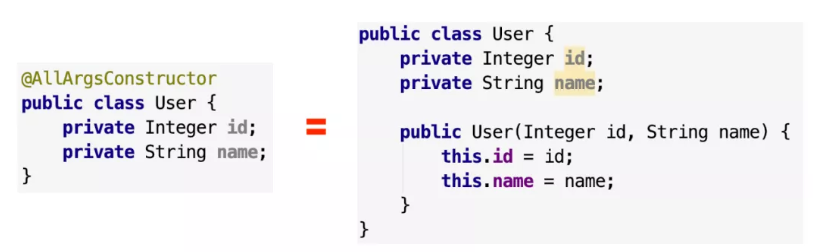
### 总结

总结来说，@RequiredArgsConstructor 注解可以让我们更方便地生成包含必需参数的构造方法，减少冗余的代码，并提高代码的可读性。

## @AllArgsConstructor

### 基本概念

生成一个包含所有参数的构造器。

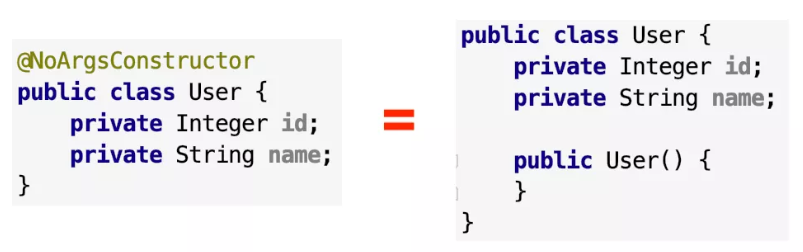


这里注意一个 Java 的小坑，当我们没有指定构造器时，Java 编译器会帮我们自动生成一个没有任何参数的构造器给该类，但是如果我们自己写了构造器之后，Java 就不会自动帮我们补上那个无参数的构造器了。

## @NoArgsConstructor

### 基本概念

生成一个没有参数的构造器。



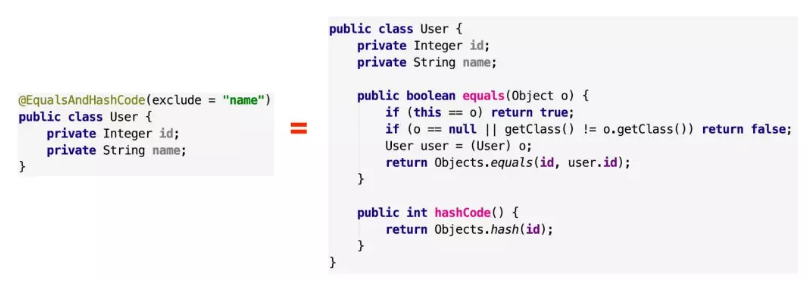
## @EqualsAndHashCode

### 基本概念

自动生成 equals(Object other) 和 hashcode() 方法，包括所有非静态变量和非 transient 的变量。



如果某些变量不想要加进判断，可以透过 exclude 排除，也可以使用 of 指定某些字段。



### 属性

#### exclude

#### of

#### callSuper

## @Log4j

### 基本概念

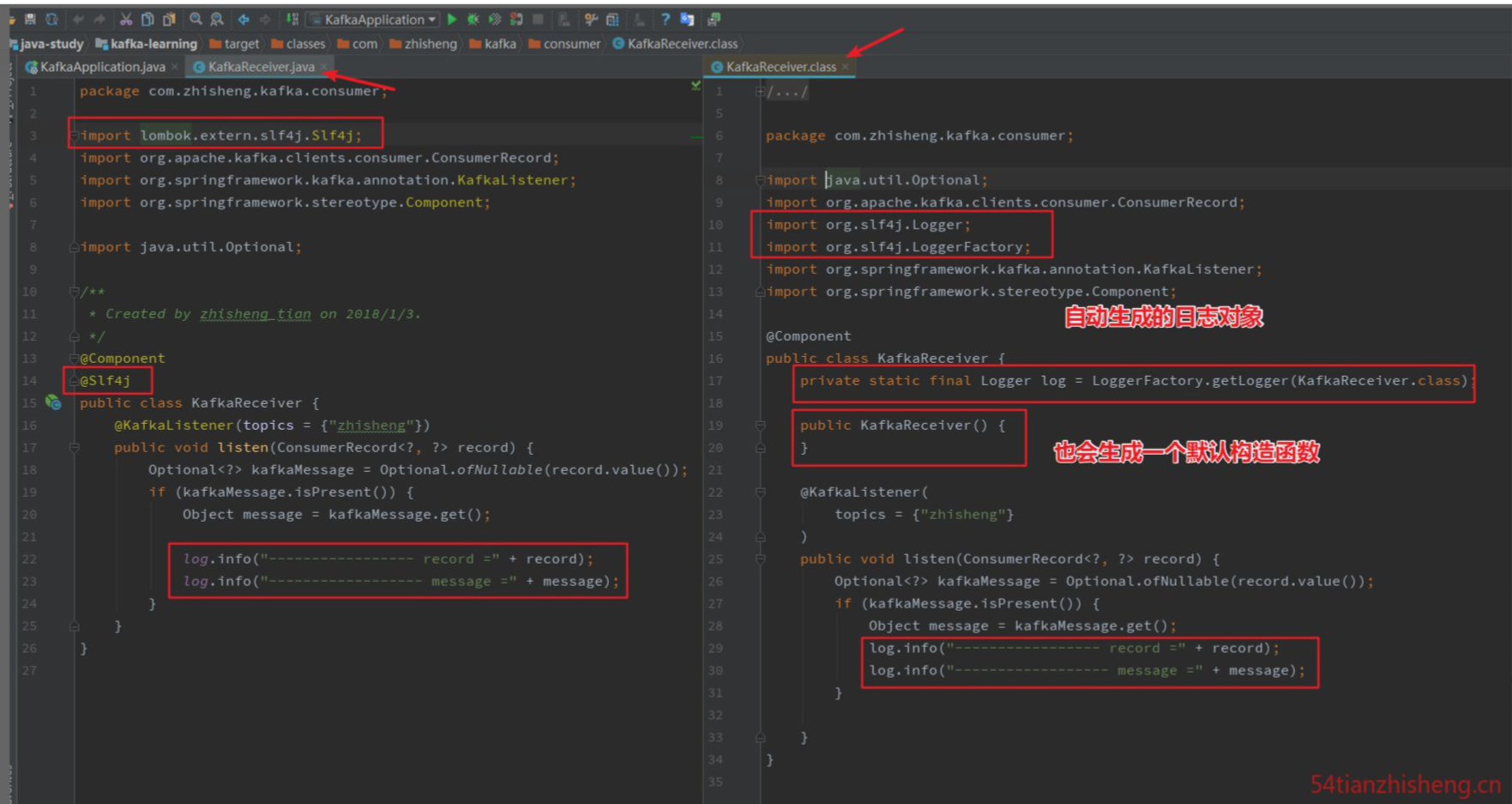
注解在类上，为类提供一个属性名为 log 的 log4j 日志对象，提供默认构造方法。

@Log4j注解生成的是基于Log4j日志框架的日志对象，而@Slf4j注解生成的是基于SLF4J（Simple Logging Facade for Java）的日志对象。如果我们想使用其他的日志框架如Logback或者Java Util Logging，就需要选择@Slf4j注解。

## @Slf4j

### 基本概念

注解在类上，为类提供一个属性名为 log 的 log4j 日志对象，提供默认构造方法。



## @ToString

### 基本概念

自动重写 toString() 方法，会印出所有变量。

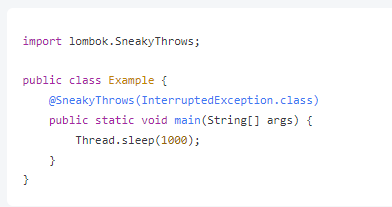


## @SneakyThrows

### 基本概念

用于简化在 Java 中处理受检异常（checked exception）的代码。通过在方法或构造函数上使用 @SneakyThrows 注解，可以使得代码在编译时不需要显式地捕获或声明受检异常，从而减少样板代码的编写。具体来说，@SneakyThrows 注解会自动为被注解的方法或构造函数中的受检异常添加一个隐式的异常处理逻辑，将受检异常转换为非受检异常（unchecked exception），然后抛出。这样在调用该方法或构造函数时，就不需要使用 try-catch 语句来处理受检异常。

### 示例



在上述示例中，main 方法使用了 @SneakyThrows(InterruptedException.class) 注解，以简化对 Thread.sleep() 方法抛出的 InterruptedException 异常的处理。当调用 Thread.sleep(1000) 时，由于 @SneakyThrows 注解的存在，编译器不再要求显式地处理这个受检异常。

在注解处理过程中，Lombok 会通过添加额外的代码实现异常的转换和重新抛出，使得可以避免在代码中编写冗长的 try-catch 块。但需要注意的是，使用 @SneakyThrows 注解并不意味着真正解决了异常问题，而是将受检异常转换为运行时异常，隐藏了异常的检查。

### 注意点

1、只能用于方法或构造函数：@SneakyThrows 注解只能应用于方法或构造函数上。

2、异常类型参数：@SneakyThrows 注解需要指定要抛出的异常类型，可以在注解中指定一个或多个异常类型。

3、自动生成的代码可读性降低：@SneakyThrows 注解自动生成的代码可能会降低代码的可读性，因为异常处理逻辑被隐式地添加，并且异常发生的上下文可能变得不明显。

4、慎用 @SneakyThrows：建议慎用 @SneakyThrows 注解，尽量在真正需要简化异常处理的场景下使用。对于大多数情况，仍然建议显式处理受检异常，以增加代码的可读性和可维护性。

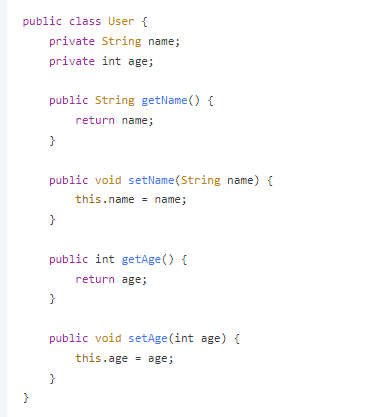
## @Accessors

### 基本概念

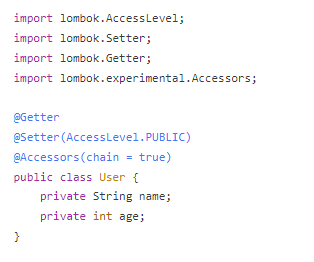
@Accessor(chain = true)是Lombok注解之一，用于简化Java类的Setter方法和Getter方法的定义。其中，chain=true表示生成的Setter方法返回当前对象本身，以实现链式调用。

### 代码示例

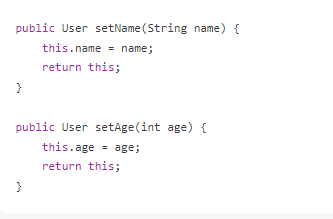
例如，我们有一个类如下：



如果我们使用@Accessor(chain = true)注解，那么上面的代码可以简化为：



这样Lombok会自动帮我们生成以下代码：



这样就可以通过链式调用设置属性，例如：

User user = new User().setName("Tom").setAge(20);

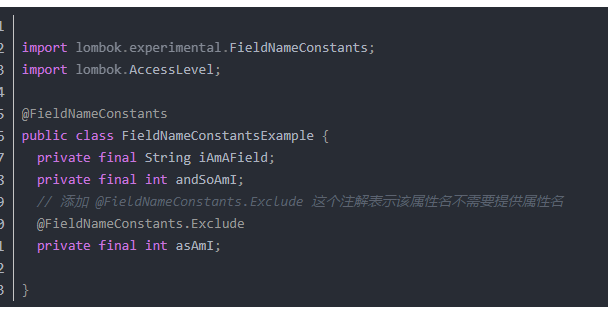
可以看到，使用@Accessor(chain = true)注解，可以快速地生成链式调用的Setter方法，提高开发效率。

## @FieldNameConstants

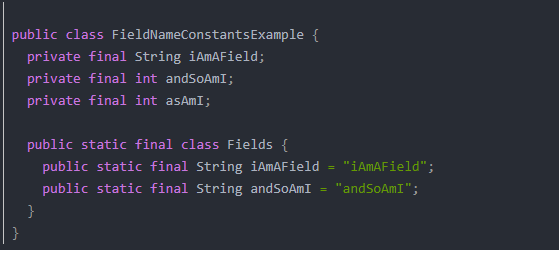
### 基本概念

作用是为类中的字段生成对应的常量。

### 示例



对应生成的代码效果：



用Junit测试一下



可以看到输出的字符串iAmAField跟声明的大小写是完全一致的。

## @Builder

### 基本概念

它用于简化构建器模式（Builder Pattern）的使用。使用 @Builder 注解可以自动生成一个内部静态类，该类充当了一个构建器，用于创建原始类的对象。当在一个类上使用 @Builder 注解时，Lombok 会在编译时自动生成一个具有链式调用风格的构建器方法，以便创建该类的对象。生成的构建器方法包含类中所有非静态的、非 transient 的字段，并自动添加相应的 setter 方法，以便通过链式调用来设置这些字段的值。构建器方法还包括一个 build() 方法，用于构建最终的对象。

注意虽然只要加上 @Builder 注解，我们就能够用流式写法快速设定对象的值，但是 setter 还是必须要写不能省略的，因为 Spring 或是其他框架有很多地方都会用到对象的 getter/setter 对他们取值/赋值，所以通常是 @Data 和 @Builder 会一起用在同个类上，既方便我们流式写代码，也方便框架做事。

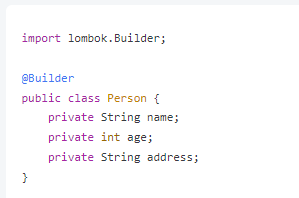
### 好处

1、简化对象的创建：通过生成的构建器方法，可以通过链式调用一次性设置多个字段的值，从而简化了对象的创建过程。

2、可读性强：使用链式调用风格的构建器方法，代码可读性更高，更易于理解和维护。

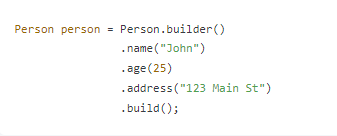
3、灵活性：构建器方法可以根据需要进行自定义扩展，例如添加额外的验证逻辑或默认值设置。

### 示例



在上述示例中，Person 类使用了 @Builder 注解。由于该注解的存在，Lombok 会自动生成一个内部静态类 PersonBuilder，该类充当了一个构建器。通过 Person.builder() 方法可以获取到该构建器，并通过链式调用来设置字段的值。最后调用 build() 方法可以构建出 Person 对象。

下面是使用 @Builder 注解创建对象的示例代码：



以上代码通过链式调用构建器方法，一次性设置了 name、age 和 address 字段的值，并通过 build() 方法构建出 Person 对象。需要注意的是，@Builder 注解只能应用于类上，并且会为该类生成一个包含链式调用风格的构建器方法。如果类中已经存在具有相同名称的方法，则不会生成新的方法。

总结来说，@Builder 注解可以让我们更方便地使用构建器模式创建对象，简化对象的创建过程，提高代码的可读性和灵活性。

## @UtilityClass

### 基本概念

@UtilityClass 是 Lombok 提供的一个注解，用于将类标记为实用工具类。它可以自动生成一些常用的静态方法和常量，并隐藏默认的构造函数，使得工具类更加简洁和易用。

### 特点

1、构造函数私有化：@UtilityClass 会自动为被注解的类生成一个私有的默认构造函数，防止该类被实例化。这样的设计符合工具类的特性，因为工具类通常只包含静态方法和静态常量，而不需要被实例化。

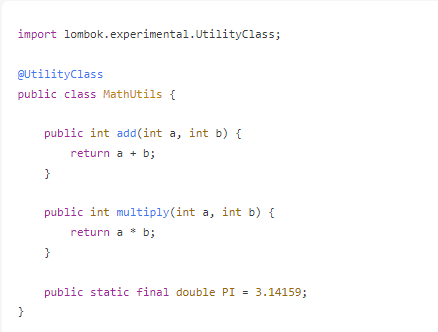
2、静态方法生成：@UtilityClass 会为类中的静态方法生成对应的字节码，可以直接通过类名调用这些静态方法，无需创建类的实例。这样可以方便地在项目中使用这些工具方法，提高代码的可读性和易用性。

3、实例方法禁用：由于 @UtilityClass 自动生成的类被设计为实用工具类，所以不允许存在实例方法。如果在被注解的类中定义了非静态方法，Lombok 会在编译时抛出错误。

4、静态常量生成：通过 @UtilityClass 注解标记的类可以定义静态常量，Lombok 会自动生成对应的常量字段。这样可以集中管理工具类中使用的常量，并提供给其他代码使用。

通过使用 @UtilityClass 注解，你可以简化实用工具类的编写，减少样板代码，并提高代码可读性和易用性。但需要注意的是，@UtilityClass 注解仅适用于具有静态方法和静态常量的类，对于需要实例化的类或包含非静态方法的类，不要使用该注解。

### 示例代码



在上述示例中，MathUtils 类使用 @UtilityClass 注解标记为实用工具类。Lombok 自动生成了 add() 和 multiply() 两个静态方法，并生成了一个名为 PI 的静态常量。这样，在其他地方就可以直接通过 MathUtils.add() 或 MathUtils.PI 来使用这些方法和常量。

## @NonNull

### 基本概念

@NonNull 是 Lombok 提供的一个注解，用于标记方法参数、构造器参数或字段不能为 null。当 Lombok 在编译时处理这个注解时，它会自动为你生成相应的 null 检查代码。

### 功能

#### 1、自动生成 null 检查

如果参数或字段被标记为 @NonNull，Lombok 会在相应的构造函数或方法中自动添加一个 null 检查。

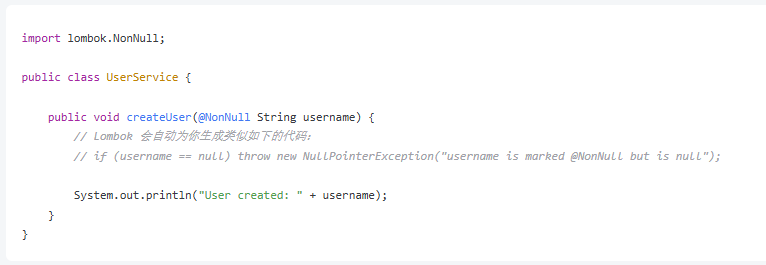
#### 2、抛出 NullPointerException

如果传递了一个 null 值，Lombok 会自动抛出一个 NullPointerException 异常。这样就能保证程序的健壮性。

### 基本用法

#### 1、用于方法参数

当你在方法的参数上使用 @NonNull 注解时，Lombok 会自动生成一个 null 检查代码。如果该参数为 null，它将抛出 NullPointerException。

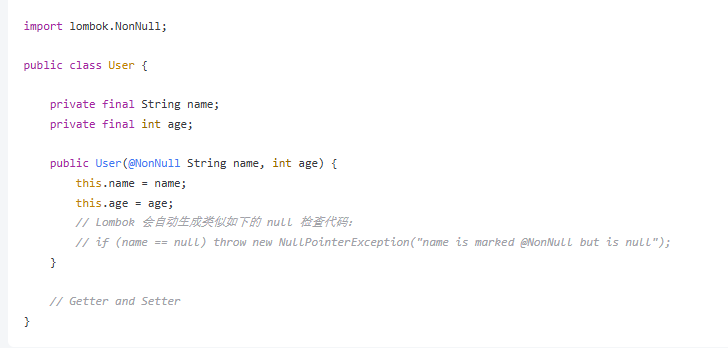


在调用 createUser(null) 时，Lombok 会自动生成 NullPointerException，并指出哪个参数为 null。



#### 2、用于构造器参数

@NonNull 也可以用于构造器的参数，Lombok 会为每个构造器参数生成 null 检查代码。

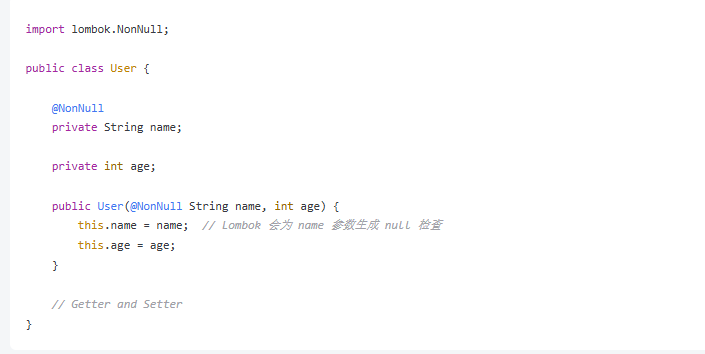


在这个例子中，如果你创建一个 User 对象时传入 null 的 name 参数，Lombok 会抛出 NullPointerException



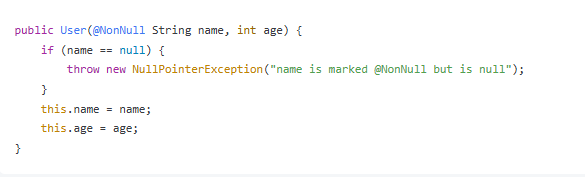
#### 3、用于字段

@NonNull 也可以直接应用于类的字段（但并不是所有情况都适用）。Lombok 会为字段生成一个 null 检查代码，通常在构造器中进行。



### 生成的代码

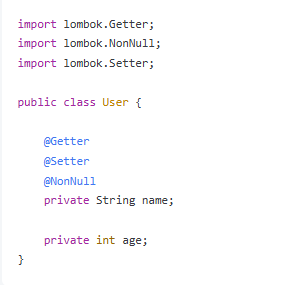
Lombok 在编译时自动生成代码，@NonNull 注解本身不会出现在生成的字节码中。以下是 Lombok 为一个带有 @NonNull 的构造器生成的代码示例。



### 与其他 Lombok 注解结合使用

#### 与 @Getter 和 @Setter 配合使用

@Getter 和 @Setter 可以自动生成 getter 和 setter 方法。配合 @NonNull 使用时，@Setter 也会自动对 null 值进行检查。



在这个例子中，name 字段的 setter 会自动生成 null 检查代码，确保不能将 null 值设置到 name 字段。

#### 与 @AllArgsConstructor 和 @NoArgsConstructor 配合使用

如果你使用 Lombok 提供的构造器注解 @AllArgsConstructor 和 @NoArgsConstructor，@NonNull 会确保构造器的参数在构造时进行 null 检查。



在 @AllArgsConstructor 自动生成的构造器中，name 参数会自动进行 null 检查。

### 注意事项

1、NullPointerException 消息：当你使用 @NonNull 注解时，Lombok 会为每个 null 参数抛出一个 NullPointerException，并在异常消息中包含参数名。例如，"name is marked @NonNull but is null"。

2、只能用于字段、方法参数和构造器参数：@NonNull 只对方法参数、构造器参数和字段有效，不能应用于类、方法或其他代码块。

3、空值检查生成代码的开销：虽然 @NonNull 自动生成了空值检查代码，但在性能敏感的代码中，应该小心使用，特别是对于频繁调用的方法。虽然这种空值检查通常不会造成明显的性能问题，但在高性能要求的场景中仍应注意。

4、与 Java 标准注解冲突：@NonNull 与一些其他常见的注解（如 javax.validation.constraints.NotNull）不同，它不是用来进行校验的注解。它仅仅是生成一个 null 检查代码，因此它的用途和其他注解略有不同。

## @Cleanup

### 基本概念

@Cleanup 注解是 Lombok 提供的一个非常实用的注解，主要用于自动资源管理，类似于 Java 7 引入的 try-with-resources 语法。

### 作用

@Cleanup 注解的主要作用是简化资源的关闭过程。当你在一个类的字段上使用 @Cleanup 注解时，Lombok 会在该字段的作用域结束时自动调用该资源的 close() 方法。这通常用于需要在使用完成后进行关闭的资源，如 InputStream、OutputStream、Connection、Reader、Writer 等实现了 AutoCloseable 接口的类。这意味着你不需要手动写 try-finally 代码来关闭这些资源，Lombok 会自动处理。

### 使用方法

字段级别：

你可以将 @Cleanup 注解应用于类的字段。当该字段超出作用域时，Lombok 会自动为该字段调用 close() 方法。



在这个例子中，@Cleanup 注解用于 inputStream 字段。无论方法如何结束（正常结束或抛出异常），Lombok 都会在方法结束时自动调用 inputStream.close() 来关闭流，避免资源泄露。

### 工作原理

Lombok @Cleanup 注解的工作原理基于 Java 的 AutoCloseable 接口以及 Java 编译器的注解处理机制。基本流程如下：

1、字段或变量声明：当你在一个字段或局部变量上使用 @Cleanup 注解时，Lombok 会在编译时生成对应的代码。

2、插入关闭逻辑：编译器会为该字段或变量自动插入 close() 方法调用，这些调用通常会被放置在方法结束之前，确保即使发生异常时也能执行资源关闭。

3、生成 try-finally 结构：对于局部变量，Lombok 会在内部生成类似 try-finally 的结构，确保在方法执行完后自动关闭资源。

### 注意事项

1、仅适用于 AutoCloseable 或 Closeable：Lombok 只能对实现了 AutoCloseable 或 Closeable 接口的资源进行管理。通常，流、数据库连接、文件句柄等都实现了这两个接口，因此它们可以使用 @Cleanup 注解。

2、异常处理：@Cleanup 会处理异常情况，但如果你手动管理流或资源，并希望在关闭时处理特定异常（比如 IOException），你仍然需要自己编写相关的异常处理代码。

3、避免重复关闭：如果你在同一资源上重复使用 @Cleanup 注解，可能会导致 Lombok 在生成的代码中重复调用 close() 方法，造成潜在的错误。应避免重复注解。

### 示例

#### 1、处理文件流



在上述示例中，@Cleanup 自动在方法结束时调用 inputStream.close()，避免了手动编写关闭代码的麻烦。

#### 2、处理数据库连接



在这个例子中，@Cleanup 注解会确保 connection 和 statement 在方法结束时自动关闭，防止资源泄漏。

### 总结

1、@Cleanup 是 Lombok 提供的一个非常有用的注解，能够简化资源的关闭操作。

2、它自动为标记的字段或局部变量调用 close() 方法，帮助你避免手动编写 try-finally 代码。

3、它适用于所有实现了 AutoCloseable 或 Closeable 接口的资源，常见的如文件流、数据库连接等。

4、使用 @Cleanup 可以让代码更加简洁、易读，并减少因忘记关闭资源而导致的资源泄漏问题。

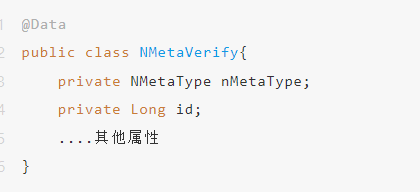
# 问题

## 1、Setter-Getter方法的坑

### 问题描述

我们在项目当中主要使用Lombok的Setter-Getter方法的注解，也就是组合注解@Data，但是在一次使用Mybatis插入数据的过程当中，出现了一个问题，问题描述如下：

我们有个实体类：



当我们使用Mybatis插入数据的时候，发现，其他属性都能正常的插入，但是就是nMetaType属性在数据库一直是null。

### 解决方法

1、修改属性名字，让第二个字母小写，或者说是规定所有的属性的前两个字母必须小写。

2、如果数据库已经设计好，并且前后端接口对接好了，不想修改，那就专门为这种特殊的属性使用idea生成get-set方法复制代码。