介绍：

在没有lombok之前，我们写一个实体类，除了定义基本的属性之外，其他如无参构造方法、有参构造方法、setter/getter、toString、甚至equals、hashCode方法等，以及如果要操作Builder模式的话还需要自己手动编写大篇幅的代码去实现，枯燥，编码量大，还容易出现拼写错误。有了lombok之后，通过在类或属性上添加几个注解，就可以让编辑器在代码编译时帮我们自动生成相应的setter/getter/构造方法等，即提高了开发效率，又提升了代码的可读性（这里我们不和record做比较）。

一句话总结：lombok是一个java库，通过引入lombok，添加相应的注解，可以简化java代码的编写，提高工作效率。

使用：（以IntelliJ IDEA中的使用为例）

环境：java+maven+Intellij IDEA

一、在pom.xml文件中添加lombok依赖：

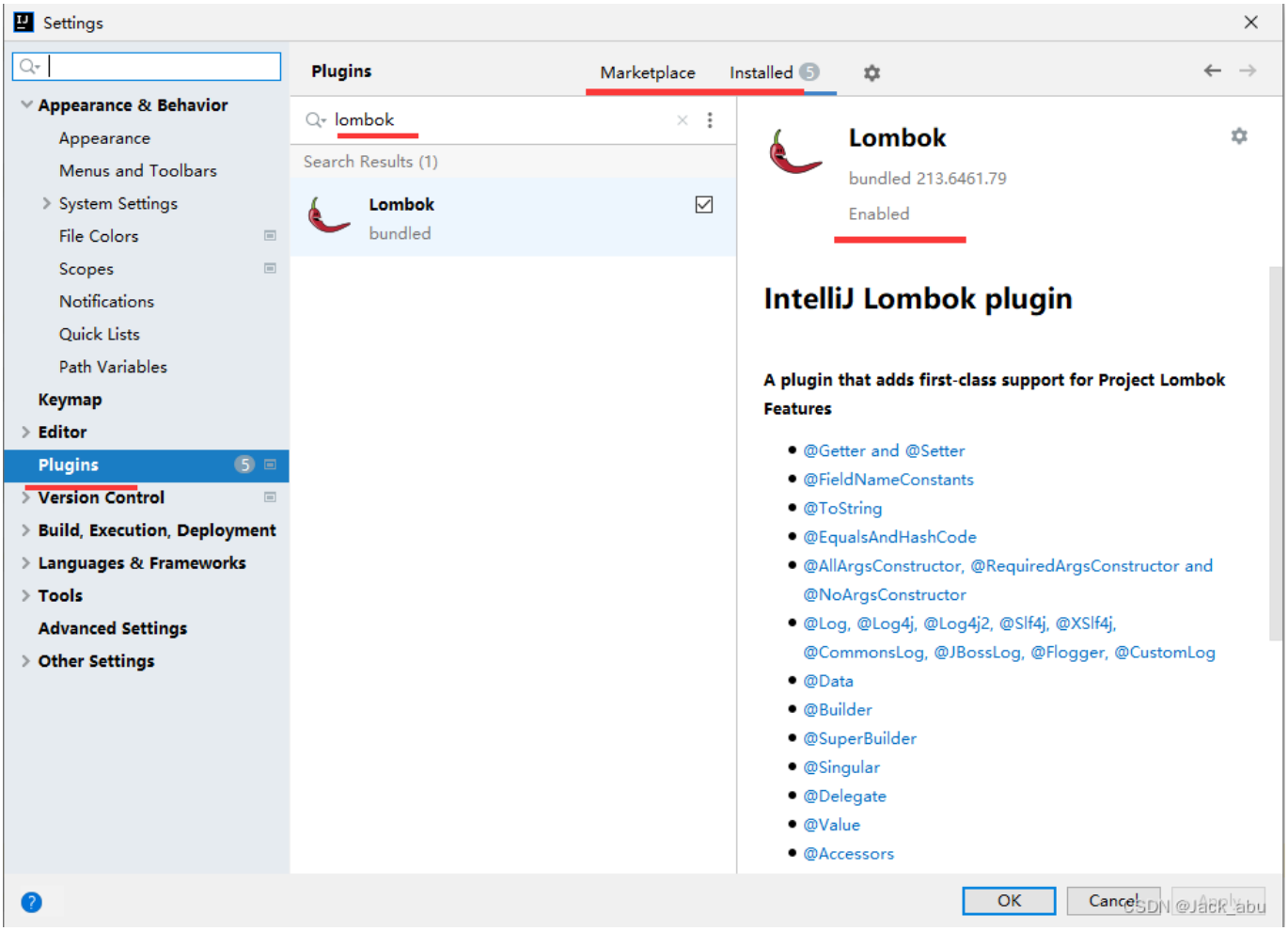
```
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <version>${lombok.version}</version>
    <scope>provided</scope>
</dependency>
```

maven仓库传送门：查看mvnrepository中的lombok

二、在IDEA中安装lombok插件

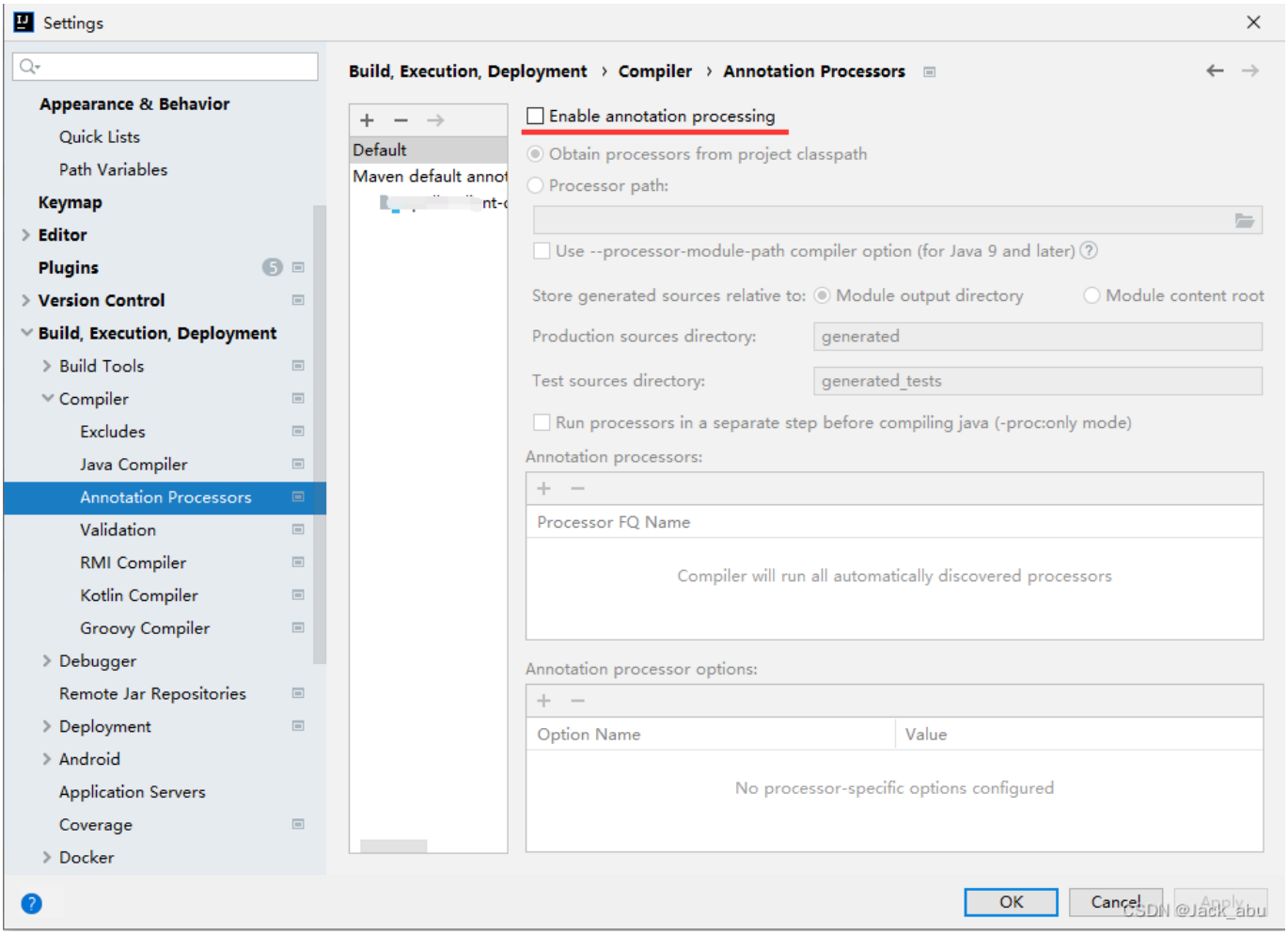
在IDE中使用lombok使用还需要安装相应的插件并启用。

打开file-settings在左侧找到plugins，未安装情况下在MarketPlace中搜索lombok进行安装，安装后再次打开可以Installed标签中找到它并启用。



如遇编译时出错，大概率为没有将注解处理器设置为enable状态，打开file-settings中，依次点击：

Build,Execution,Deployment-Compiler-Annotation Processors，勾选Enable annotation processing



使用示例：

不使用lombok的情况：

```java
public class Student implements java.io.Serializable {
    private static final long serialVersionUID = 1L;

    private String name;
    private Integer age;
    private double score;

    public Student() {
    }

    public Student(String name, Integer age, double score) {
        this.name = name;
        this.age = age;
        this.score = score;
    }

    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public Integer getAge() {
        return age;
    }
    public void setAge(Integer age) {
        this.age = age;
    }
    public double getScore() {
        return score;
    }
    public void setScore(double score) {
        this.score = score;
    }

    @Override
    public String toString() {
        return "Student{" +
                "name='" + name + '\'' +
                ", age=" + age +
                ", score=" + score +
                '}';
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Student student = (Student) o;
        return Double.compare(student.score, score) == 0 && Objects.equals(name,
```

```
student.name) && Objects.equals(age, student.age);
    }

    @Override
    public int hashCode() {
        return Objects.hash(name, age, score);
    }
}
```

使用lombok的情况：

```
@Data
public class Student implements java.io.Serializable {
    private static final long serialVersionUID = 1L;

    private String name;
    private Integer age;
    private double score;

}
```

可以看到代码明显简洁了不少，大大提升开发效率。

@Data注解说明：

```
/**
 * Generates getters for all fields, a useful toString method, and hashCode and equals
implementations that check
 * all non-transient fields. Will also generate setters for all non-final fields, as
well as a constructor.
 * <p>
 * Equivalent to {@code @Getter @Setter @RequiredArgsConstructor @ToString
@EqualsAndHashCode}.
 * <p>
 * Complete documentation is found at <a
href="https://projectlombok.org/features/Data">the project lombok features page for
&#64;Data</a>.
 *
 * @see Getter
 * @see Setter
 * @see RequiredArgsConstructor
 * @see ToString
 * @see EqualsAndHashCode
 * @see lombok.Value
 */
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.SOURCE)
public @interface Data {
    /**
     * If you specify a static constructor name, then the generated constructor will
be private, and
     * instead a static factory method is created that other classes can use to create
instances.
     * We suggest the name: "of", like so:
     *
     * <pre>
     *     public @Data(staticConstructor = "of") class Point { final int x, y; }
     * </pre>
     *
     * Default: No static constructor, instead the normal constructor is public.
     *
     * @return Name of static 'constructor' method to generate (blank = generate a
normal constructor).
     */
    String staticConstructor() default "";
}
```

从注释javadoc中可以看出，它会为所有字段生成getter,setter方法，一个有用的toString方法，hashCode方法，equals方法，还有构造方法。

通过添加

```
@Accessors(chain = true)
```

可以实现setXXX方法的链式调用

```
A container for settings for the generation of getters and setters.

Complete documentation is found at the project lombok features page for @Accessors ↗ .

Using this annotation does nothing by itself; an annotation that makes lombok generate getters and setters, such as lombok.Setter or lombok.Data is also required.

@Target({ElementType.TYPE, ElementType.FIELD})
@Retention(RetentionPolicy.SOURCE)
public @interface Accessors {

    If true, accessors will be named after the field and not include a get or set prefix. If true and chain is omitted, chain defaults to true. default: false

    Returns: Whether or not to make fluent methods (named fieldName(), not for example setFieldName).

    boolean fluent() default false;

    If true, setters return this instead of void. default: false, unless fluent=true, then default: true

    Returns: Whether or not setters should return themselves (chaining) or void (no chaining).

    boolean chain() default false;

    If present, only fields with any of the stated prefixes are given the getter/setter treatment. Note that a prefix only counts if the next character is NOT a lowercase character or the last letter of the prefix is not a letter (for instance an underscore). If multiple fields all turn into the same name when the prefix is stripped, an error will be generated.

    Returns: If you are in the habit of prefixing your fields (for example, you name them fFieldName, specify such prefixes here).

    String[] prefix() default {};
}
```


从注解中可以看出，当chain的值设置为true时，setters的返回值会用this替代void。

常用注解：（按字面意思理解即可）

@NoArgsConstrutor：无参构造方法；

@RequiredArgsConstructor：有参构造方法，使用了@NonNull约束的属性；

@AllArgsContructor：全参构造方法；

@Data：setter/getter,tostring,hashcode,equals,requiredArgsConstructor

@Setter,@Getter:setter/getter

@Accessors(chain=true)：settter的链式调用

@Builder：将类转变为建造者模式

@EqualsAndHashCode：生成equals和hashCode方法

@Slf4j：生成一个log变量，生private static final修饰，配合日志框架使用；

工作原理分析：

知道lombok是什么了，也知道它怎么用，那它究竟是怎么实现的呢？来分析分析它的工作原理！

lombok的使用是注解，那么它的实现也离不开注解。

jdk1.5在引入注解时，也提供了两种注解解析方式：运行时解析 和 编译时解析 。

运行时解析：@Retention(RetentionPolicy.RUNTIME)

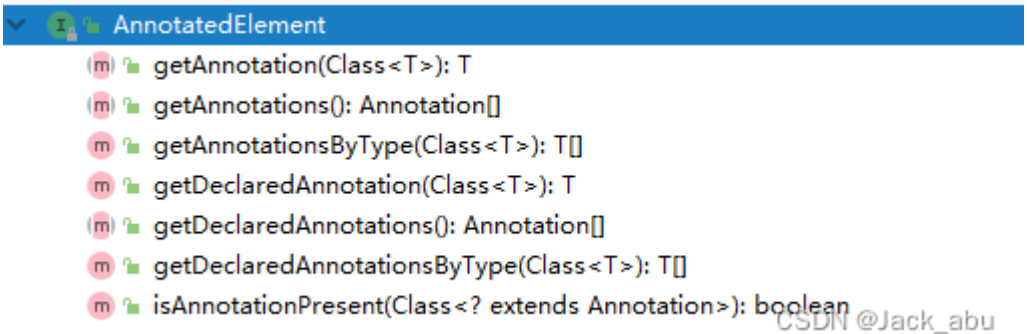
(Retention：保留，RetentionPolicy：保留策略，不同和策略表示注解能保留的时间)

RetentionPolicy.RUNTIME:表示注解将被编译器记录在class文件中，并且在运行时被VM保留，通过反射机制也就可以拿到这个注解了。

> Annotations are to be recorded in the class file by the compiler and retained by the VM at run time, so they may be read reflectively.
> See Also: `reflect.AnnotatedElement`

RUNTIME

java.lang.reflect.AnnotatedElement：一个接口，Class,Field,Method,Constructor,Package等等都实现了这个接口。AnnotatedElement接口中有下面这几个方法获取注解信息：



通过get[Declared]Annotation方法便可获取到对应的注解信息。

编译时解析:

编译时解析有两种机制：APT(Annotation Process Tool)和Pluggable Annotation Processing API，APT由于在jdk7中已被标记为废弃且在jdk8中已删除，所以重点看看Pluggable Annotation Processing API。

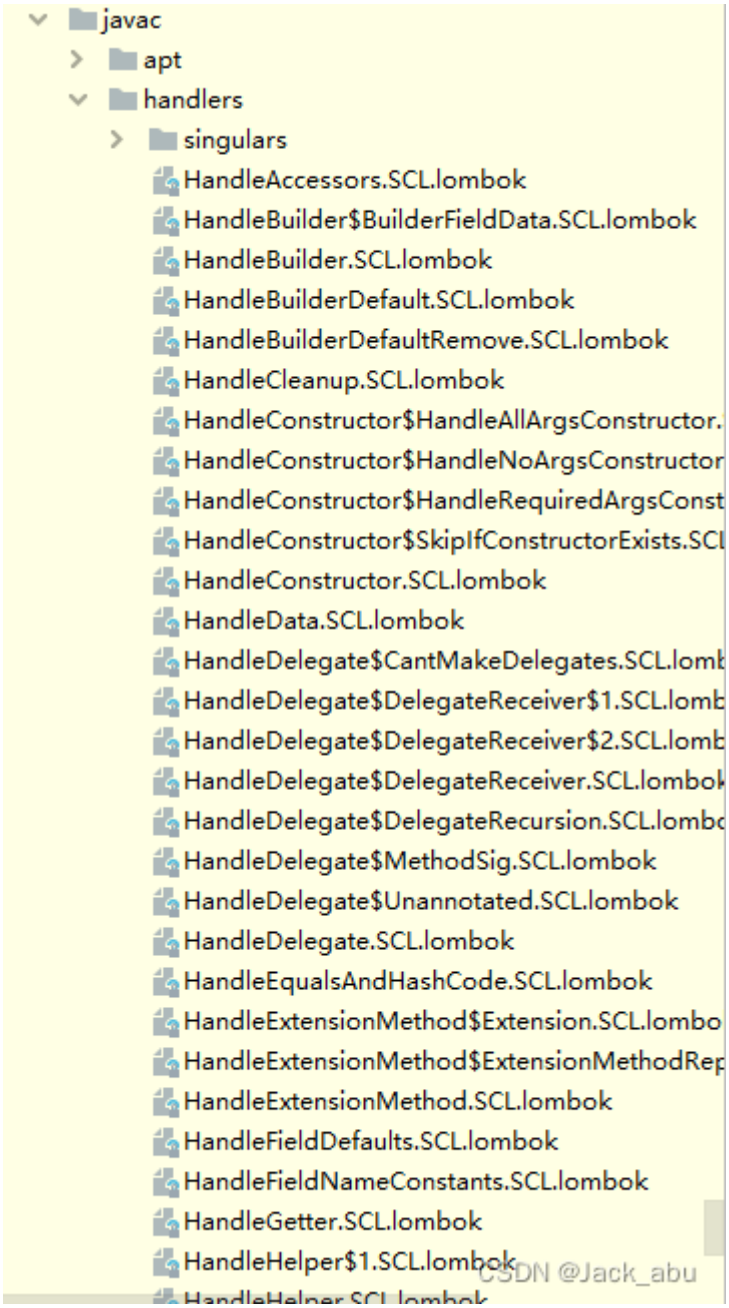
Pluggable Annotation Processing API:

Since JDK1.6，从jdk6开始加入，替代apt，解决javac无法使用apt的问题，javac在执行时会调用实现了此API的程序，我们也就可以在这个过程中对编译器做一些增强处理。

*.java*文件-->*javac(*解析与填充符号表->注解处理->分析与字节码生成->生成二进制*class*文件*)*-->.*class*文件-->*vm*-->*0101*...

> lombok本质上就是一个实现了JSR 269: Pluggable Annotation Processing API (编译期的注解处理器)的程序，在javac的过程中做如下的处理：
>
> 1、通过javac对源码进行分析，生成一棵抽象语法树AST（编程语言源代码的抽象语法结构的树状表示,树的每个节点表示源代码中出现的构造。语法是“抽象的”，因为它不表示源代码的确切文本，而是表示其语法结构）；
>
> 2、运行过程中调用实现了JSR 269: Pluggable Annotation Processing API (编译期的注解处理器)的lombok程序；
>
> 3、lombok程序对AST进行处理，找到由@Data注解所在类对应的语法树，并增加setter,getter等相应的树节点；
>
> 4、javac根据修改的AST生成字节码文件（.class）。

在lombok源码中，则是在HandlerXXX中实现的，如HandlerGetter.handler()



(引入lombok的优点就不用多说了，缺点嘛，感觉就是构造器的重载上支持的不够好)

神奇操作（骚操作）：

（主要是几个注解，也是从别的大佬那里看来的，自己再总结归纳下，巩固巩固，源文链接已贴在文章末尾）

onXXX

如：onConstrutor,onMethod,onParam

使用方式：

jdk7:

```
@RequiredArgsConstructor(onConstructor=@__({@AnnotationsGoHere}))}
```

jdk8:

```
@RequiredArgsConstructor(onConstructor_={@AnnotationsGohere})} // note the underscore
after {@code onConstructor}
```

以onConstructor为例，表示在生成构造方法时，在构造方法上会使用的注解

```
/**
 * Any annotations listed here are put on the generated constructor.
 * The syntax for this feature depends on JDK version (nothing we can do about that;
it's to work around javac bugs).<br>
 * up to JDK7:<br>
 *  {@code @RequiredArgsConstructor(onConstructor=@__({@AnnotationsGoHere}))}<br>
 * from JDK8:<br>
 *  {@code @RequiredArgsConstructor(onConstructor_={@AnnotationsGohere})} // note the
underscore after {@code onConstructor}.
 *
 * @return List of annotations to apply to the generated constructor.
 */
AnyAnnotation[] onConstructor() default {};
```

如：在spring相关的注解中（如@Service,@Controller,@Component等），使用@RequiredArgsConstructor(onContructor=@__(@Autowired))的示例如下：

MyService.java

```
@Service
@RequiredArgsConstructor(onConstructor = @__(@Autowired))
public class MyService {
    private final UserService userService;//想要被lombok注入，这里必须是final修饰
}
```

生成的 MyService.class

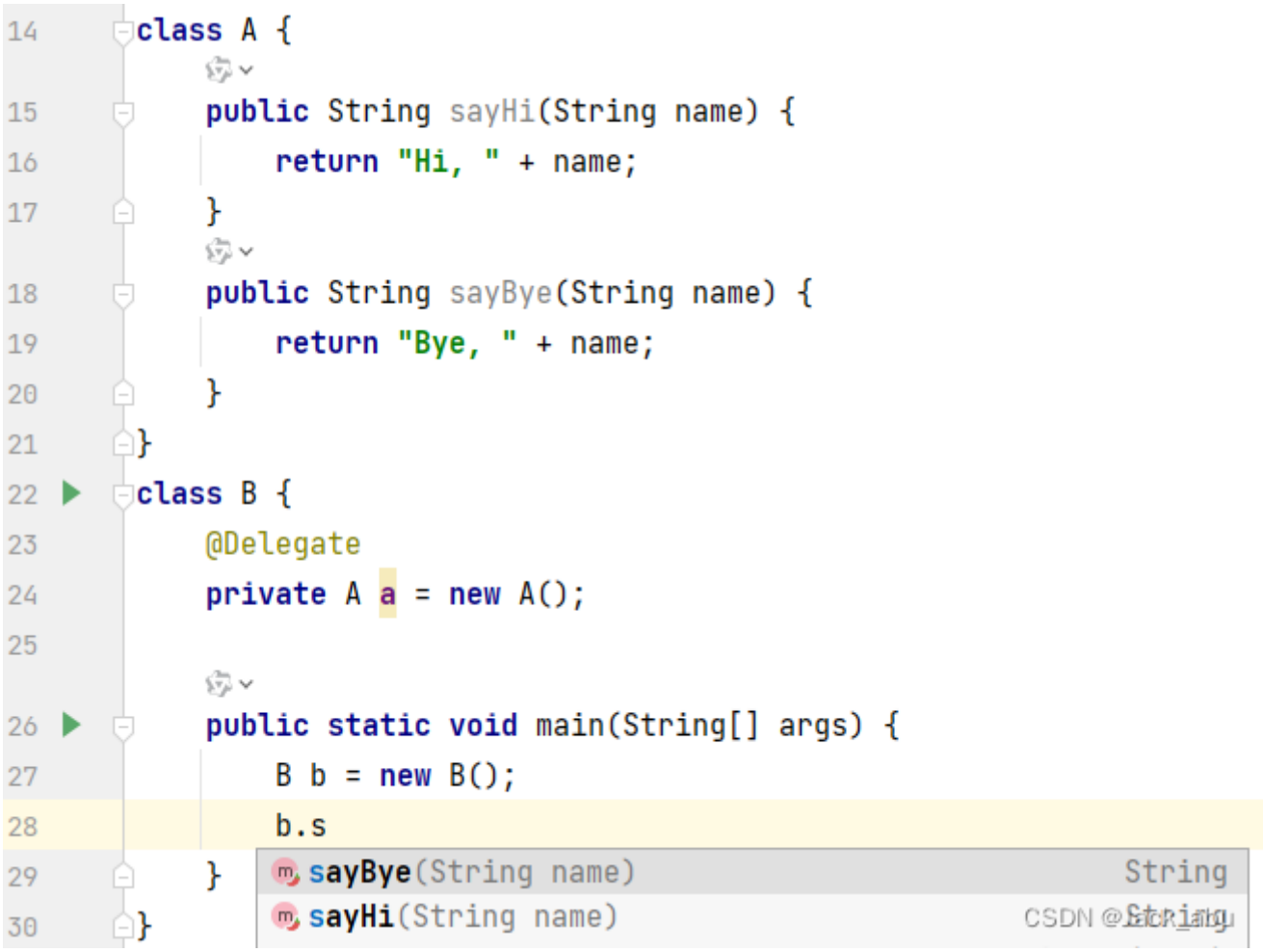
```java
@Service
public class MyService {
  private final UserService userService;

  @Autowired
  public MyService(UserService userService) {
    this.userService = userService;
  }
}
```

@Delegate: 不用写重复代码，直接使用其他类的方法

如：类A有一个方法叫sayHi(String name)和sayBye(String name)，如果想要类B也能用sayHi和sayBye方法，只要让类B有拥有一个类A的属性，并在这个属性上加上@Delegate注解，在类B中就可以直接调用类A中的方法(不是通过a.xxx的方法调用，而是直接调用sayHi和sayHello)。

代码示例：



@Cleanup：自动管理输入输出流等各种需要释放的资源，默认调用close方法

使用方式如：

```
@Cleanup InputStream is = new FileInputStream("some/file.ext");
```

看看Cleanup注解的定义：

```
/**
 * Ensures the variable declaration that you annotate will be cleaned up by calling
its close method, regardless
 * of what happens. Implemented by wrapping all statements following the local
variable declaration to the
 * end of your scope into a try block that, as a finally action, closes the resource.
 * <p>
 * Complete documentation is found at <a
href="https://projectlombok.org/features/Cleanup">the project lombok features page for
&#64;Cleanup</a>.
 * <p>
 * Example:
 * <pre>
 * public void copyFile(String in, String out) throws IOException {
 *     &#64;Cleanup FileInputStream inStream = new FileInputStream(in);
 *     &#64;Cleanup FileOutputStream outStream = new FileOutputStream(out);
 *     byte[] b = new byte[65536];
 *     while (true) {
 *         int r = inStream.read(b);
 *         if (r == -1) break;
 *         outStream.write(b, 0, r);
 *     }
 * }
 * </pre>
 *
 * Will generate:
 * <pre>
 * public void copyFile(String in, String out) throws IOException {
 *     &#64;Cleanup FileInputStream inStream = new FileInputStream(in);
 *     try {
 *         &#64;Cleanup FileOutputStream outStream = new FileOutputStream(out);
 *         try {
 *             byte[] b = new byte[65536];
 *             while (true) {
 *                 int r = inStream.read(b);
 *                 if (r == -1) break;
 *                 outStream.write(b, 0, r);
 *             }
 *         } finally {
 *             if (outStream != null) outStream.close();
 *         }
 *     } finally {
 *         if (inStream != null) inStream.close();
 *     }
 * }
 * </pre>
 */
@Target(ElementType.LOCAL_VARIABLE)
@Retention(RetentionPolicy.SOURCE)
public @interface Cleanup {
```

```
    /** @return The name of the method that cleans up the resource. By default,
'close'. The method must not have any parameters. */
    String value() default "close";
}
```

通过javadoc中的说明，便一目了然，在会try-finally的finally代码块中执行value指定的方法。value的值默认为"close"，所以，如果释放资源的方法并非close()则通过value值指定即可。

@Singular：让集合类型的字段更容易维护；

@Builder：让类支持链式构造，注意：只生前当前类的字段和参数，对继承的字段不做处理；

@Singular与@Builder的组合：

```
@Data
@Builder
class User {
    private String name;
    private int age;
    @Singular
    private List<String> hobbies;
}

User user = User.builder().name("张三")
                    .age(18)
                    .hobbies(Arrays.asList("打篮球", "看电影"))
                    .clearHobbies()//清空hobbies
                    .hobby("听音乐")
                    .build();
```

@With:创建一个当前对象的副本，更改某些属性的值

With注解定义：

Put on any field to make lombok build a 'with' - a withX method which produces a clone of this object (except for 1 field which gets a new value).

Complete documentation is found at the project lombok features page for @With ↗.

Example:

```
private @With final int foo;
```

will generate:

```
public SELF_TYPE withFoo(int foo) {
    return this.foo == foo ? this : new SELF_TYPE(otherField1, otherField2, foo);
}
```

This annotation can also be applied to a class, in which case it'll be as if all non-static fields that don't already have a With annotation have the annotation.

```
@Target({ElementType.FIELD, ElementType.TYPE})
@Retention(RetentionPolicy.SOURCE)
public @interface With {
```


使用示例：

```
@With
@AllArgsConstructor
@Data
class Person {
    private String name;
    private int age;
}

Person person = new Person("张三",18);
Person person1 = person.withName("李四");//person对象的副本，name属性改成了李四，age=18
System.out.println(person1.getAge());//18
```

工作中遇到的一些坑：（请绕行）

一、使用@Data注解生成的equals方法，默认只判断了当前类的属性是否相等，而忽略了父类的属性，从而使两个明显不相等的对象通过equals尽判断为相等。这个坑，在一些去重的场景中很容易踩到，如：set,map中，都是通过equals进行比较进行去除重复的。

示例如：

```
@Data
class Child {
    private int age;
}

@With
@Builder
@Data
class Person extends Child {
    private String name;
}

Person p1 = new Person("张三");
        p1.setAge(18);
        Person p2 = new Person("张三");
        p2.setAge(20);
        System.out.println(p1.equals(p2));//这里居然输出了true
```

解决方式：

使用@EnqualsAndHashCode并显示指定canSuper=true，这样就会调用父类的equals

```
@EqualsAndHashCode(callSuper = true)
```

不过，这种解决方式，要在所有类中去查找添加，费时费力，容易造成遗漏，可以添加一个lombok.config配置文件，内容写为：

```
lombok.equalsAndHashCode.callSuper=call
```

那么在项目中所有用到@Data注解的类中在生成代码时都会自动加上@EqualsAndHashCode(callSuper=true)，从而在全局上解决问题。

二、lombok对于第一个字母小写第二个字母大写的属性，生成的setter/getter方法，与mybatis获取属性的方法容易造成冲突。如sName属性，生成getSName,setSName，但在mybatis中并不能正确的解析到对应的属性名称为sName，原因见下面的源码中的注释。

如：在mybatis(本文为3.5.6)中通过set/get方法获取属性名的源码如下：

```java
/**
 *    Copyright 2009-2019 the original author or authors.
 *
 *    Licensed under the Apache License, Version 2.0 (the "License");
 *    you may not use this file except in compliance with the License.
 *    You may obtain a copy of the License at
 *
 *       http://www.apache.org/licenses/LICENSE-2.0
 *
 *    Unless required by applicable law or agreed to in writing, software
 *    distributed under the License is distributed on an "AS IS" BASIS,
 *    WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 *    See the License for the specific language governing permissions and
 *    limitations under the License.
 */
package org.apache.ibatis.reflection.property;

import java.util.Locale;

import org.apache.ibatis.reflection.ReflectionException;

/**
 * @author Clinton Begin
 */
public final class PropertyNamer {

  private PropertyNamer() {
    // Prevent Instantiation of Static Class
  }

  public static String methodToProperty(String name) {
    if (name.startsWith("is")) {
      name = name.substring(2);
    } else if (name.startsWith("get") || name.startsWith("set")) {
      name = name.substring(3);
    } else {
      throw new ReflectionException("Error parsing property name '" + name + "'.
Didn't start with 'is', 'get' or 'set'.");
    }
    //这里是重点
    //如果属性长度为1，如private int x; public int getX()，name属性就是x
    //如果属性长度大于1，此时还会判断第二个字母是否为大写，如果不是大写的情况下，才会对第一
个字母转换为小写处理，如果我们的属性叫：sName,那会getSName就不会被转把sName这个正确的属性
    if (name.length() == 1 || (name.length() > 1 &&
!Character.isUpperCase(name.charAt(1)))) {
      name = name.substring(0, 1).toLowerCase(Locale.ENGLISH) + name.substring(1);
    }

    return name;
  }
```

```
  public static boolean isProperty(String name) {
    return isGetter(name) || isSetter(name);
  }

  public static boolean isGetter(String name) {
    return (name.startsWith("get") && name.length() > 3) || (name.startsWith("is") &&
name.length() > 2);
  }

  public static boolean isSetter(String name) {
    return name.startsWith("set") && name.length() > 3;
  }

}
```

踩坑的重点见代码中的中文注释部分，规避方式也就是尽量避免这种命名方式吧！！！

三、@Accessors(chain=true)与excel导出工具（如easyexcel）在使用中遇到的坑

首先，说下@Accessors(chain=true)注解，当实体类被此注解标记时，我们在调用这个实体类的setXXX方法就可以使用链式调用了，因为它setXXX当chaine=true返回的是this而非void。

在easyexcel中，是通过cglib做为反射工具包的。

cglib中则使用rt.jar中的Introspector这个类的方法来获取get和set方法进一步获取属性的，Introspector类中的实现，在判断setXXX方法加了一个判断返回值是否为Void的条件，此时，返回this的setXXX方法将会被过滤。。。

解雇方案：不用@Accessors(chain=true)...

四、@Builder默认值的坑

在lombok1.18.4版本之前，字段属性上的默认值，通过build构建出来的对象并没有赋值。

如：

```java
@Data
@Builder
@Accessors(chain = true)
public class Student implements java.io.Serializable {
    private static final long serialVersionUID = 1L;

    private String name = "张三";//默认值设为张三
    private Integer age;
    private double score;

    public static void main(String[] args) {
        Student student = Student.builder().build();
        System.out.println(student);
    }

}
```

输出：

```
"C:\Program Files\Java\jdk1.8.0_311\bin\java.exe" ...
Student(name=null, age=null, score=0.0)

Process finished with exit code 0
```


这与预期的不符啊，name居然不是“张三”

解决方案1：在name属性上添加@Builder.Default注解

```java
@Builder.Default
private String name = "张三";//默认值设为张三
```

输出：

```
"C:\Program Files\Java\jdk1.8.0_311\bin\java.exe" ...
Student(name=张三, age=null, score=0.0)

Process finished with exit code 0
```


解决方案2：在@Builder注解中增加toBuilder=true

```java
@Data
@Builder(toBuilder = true)
@Accessors(chain = true)
public class Student implements java.io.Serializable {
    private static final long serialVersionUID = 1L;

    private String name = "张三";//默认值设为张三
    private Integer age;
    private double score;

    public static void main(String[] args) {
        Student student = Student.builder().build();
        System.out.println(student);
    }

}
```

输出：

```
"C:\Program Files\Java\jdk1.8.0_311\bin\java.exe" ...
Student(name=null, age=null, score=0.0)

Process finished with exit code 0
```


问题并没有解决！！！

调整main方法：

```java
Student student = new Student();
System.out.println(student);

Student student1 = new Student().toBuilder().build();
System.out.println(student1);
```

输出：

```
"C:\Program Files\Java\jdk1.8.0_311\bin\java.exe" ...
Student(name=张三, age=null, score=0.0)
Student(name=张三, age=null, score=0.0)

Process finished with exit code 0
```

这次是正常的，说明我们必须先进行实例化(new Student())才能构造（build()）,否则默认值还是不生效！！！

五、当@Data和@Builder结合使用的时候，要注意，总会有些方法不会自动生成，这个就留着慢慢踩吧，发现要用的时候没有生成再去单独添加相应的注解:(

本文中引参及参考过的一些链接：

lombok官网

盘点Lombok的几个骚操作