

06 Java属性映射的正确姿势

更新时间：2020-06-03 13:48:06



“生活永远不像我们想像的那样好，但也不会像我们想像的那样糟。——莫泊桑”

1. 前言

前一节讲到项目为了更容易维护，易于拓展等原因会使用各种分层领域模型。在多层应用中，常需要对各种不同的分层对象进行转换，这就会存在一个非常棘手的问题即：编写不同的模型之间相互转换的代码非常麻烦。其中最常见和最简单的方式是编写对象属性转换函数，即普通的 **Getter/Setter** 方法。除此之外各种各种属性映射工具。

- 那么常见的 **Java** 属性映射工具有哪些？
- 它们的原理以及对其性能怎样？
- 实际开发中该如何选择？

本节将给出解答。

2. 常见的 **Java** 属性映射的工具及其原理

2.1 常见的 **Java** 属性映射工具

常见的 **Java** 属性映射工具有以下几种：

1. `org.apache.commons.beanutils.BeanUtils#copyProperties`
2. `org.springframework.beans.BeanUtils#copyProperties(java.lang.Object, java.lang.Object)`
3. `org.dozer.Mapper#map(java.lang.Object, java.lang.Class<T>)`
4. `net.sf.cglib.beans.BeanCopier#copy`
5. `ma.glasnost.orka.MapperFacade#map(S, D)`

6. mapstruct

2.2 原理

1、Getter/Setter 方式使用原生的语法，虽然简单但是手动编写非常耗时；

2、通过 [dozer](#) 的 [maven 依赖](#) 可以看出，[dozer](#) 并没有使用字节码增强技术，因为并没有引用任何字节码增强技术的 jar 包；

我们再从其核心类 [org.dozer.MappingProcessor](#) 中寻找线索：

```
import java.lang.reflect.Array;
import java.lang.reflect.Modifier;
import java.lang.reflect.InvocationTargetException;
...
```

我们可以断定，[dozer](#) 使用的是反射机制。

3、同样的 [commons](#) 和 [Spring](#) 的 [BeanUtil](#) 工具类也采用的是反射方式。优点是两个是非常常用的类库，不需要引用更多复杂的包；

4、[cglib](#) 的 [BeanCopier](#) 的原理是不是也是反射机制呢？

我们可以通过 [cglib](#) 的 [maven 库](#) 的编译依赖中找到线索：

Compile Dependencies (2)

Category/License	Group / Artifact	Version	Updates
Build Tool Apache 2.0 	org.apache.ant » ant (optional)	1.10.3	1.10.6
Bytecode BSD 	org.ow2.asm » asm	7.1	✓

发现该库依赖了 [asm](#)，我们去 [asm](#) 官网可以看到它的介绍：

[asm](#) 库是一个 [Java](#) 字节码操作和分析框架，它可以用来修改已经存在的字节码或者直接二进制形式动态生成 class 文件。[asm](#) 的特点是小且快。

、同样的，我们可以通过 [orika](#) 的 [maven 库](#) 得到其实现依赖的核心技术：

Compile Dependencies (6)

Category/License	Group / Artifact	Version	Updates
Object Size Apache 2.0	 com.carrotsearch » java-sizeof	0.0.5	✓
Core Utils Apache 2.0	 com.google.guava » guava (optional)	24.0-jre	28.0-jre
Reflection BSD	 com.thoughtworks.paranamer » paranamer	2.8	✓
JVM Languages BSD	 org.codehaus.janino » janino	3.0.8	3.0.14
Bytecode Apache 2.0 LGPL 2.1 MPL 1.1	 org.javassist » javassist	3.24.0-GA	3.25.0-GA
Logging MIT	 org.slf4j » slf4j-api	1.7.25	1.7.26

其中 **javassist** 我们知道它是一个字节码操作工具。

我们去它的官网看下介绍：

javassist 让操作字节码非常容易。**javassist** 允许 **java** 程序运行时定义一个新的类，也可以实现在 **JVM** 加载类文件时修改它。**javassist** 提供两种级别的 **API**，一种是源码级别；一种是字节码级别。使用源码级别的 **API**，无需对 **java** 字节码特定知识有深入的了解就可以轻松修改类文件。字节码级别的 **API** 则允许用户直接修改类文件。

6、通过 **MapStruct** 的官网的介绍我们可以看出，**mapstruct** 采用原生的方法调用，因此更快速，更安全也更容易理解。根据官网的介绍我们知道，使用时只需要使用它的注解，定义好转换接口，转换函数，编译时会自动生成转换工具的实现类、调用属性赋值和取值函数实现转换。**mapstruct** 还支持通过注解形式定义不同属性名的映射关系等，功能很强大。

转换代码：

```
@Mapper
public interface UserMapper {
    UserMapper INSTANCE = (UserMapper)Mappers.getMapper(UserMapper.class);

    UserDTO userDo2Dto(UserDO var1);
}
```

编译后生成自动的转换接口的实现类：

```

public class UserMapperImpl implements UserMapper {
    public UserMapperImpl() {
    }

    public UserDTO userDo2Dto(UserDO userDO) {
        if (userDO == null) {
            return null;
        } else {
            UserDTO userDTO = new UserDTO();
            userDTO.setName(userDO.getName());
            userDTO.setAge(userDO.getAge());
            userDTO.setNickName(userDO.getNickName());
            userDTO.setBirthDay(userDO.getBirthDay());
            return userDTO;
        }
    }
}

```

大大简化了代码。

官方还提供了非常详细的[参考文档](#) 和使用范例，提供了很多高级用法。

2.3 性能

接下来按照惯例，我们对比一下它们的性能。

我们在 `com.imooc.basic.converter.UserConverterTest` 类中对上面的常见对象转换方式进行单测 `UserDO` 对象：

```

@Data
public class UserDO {
    private Long id;
    private String name;
    private Integer age;
    private String nickName;
    private Date birthDay;
}

```

目标对象：

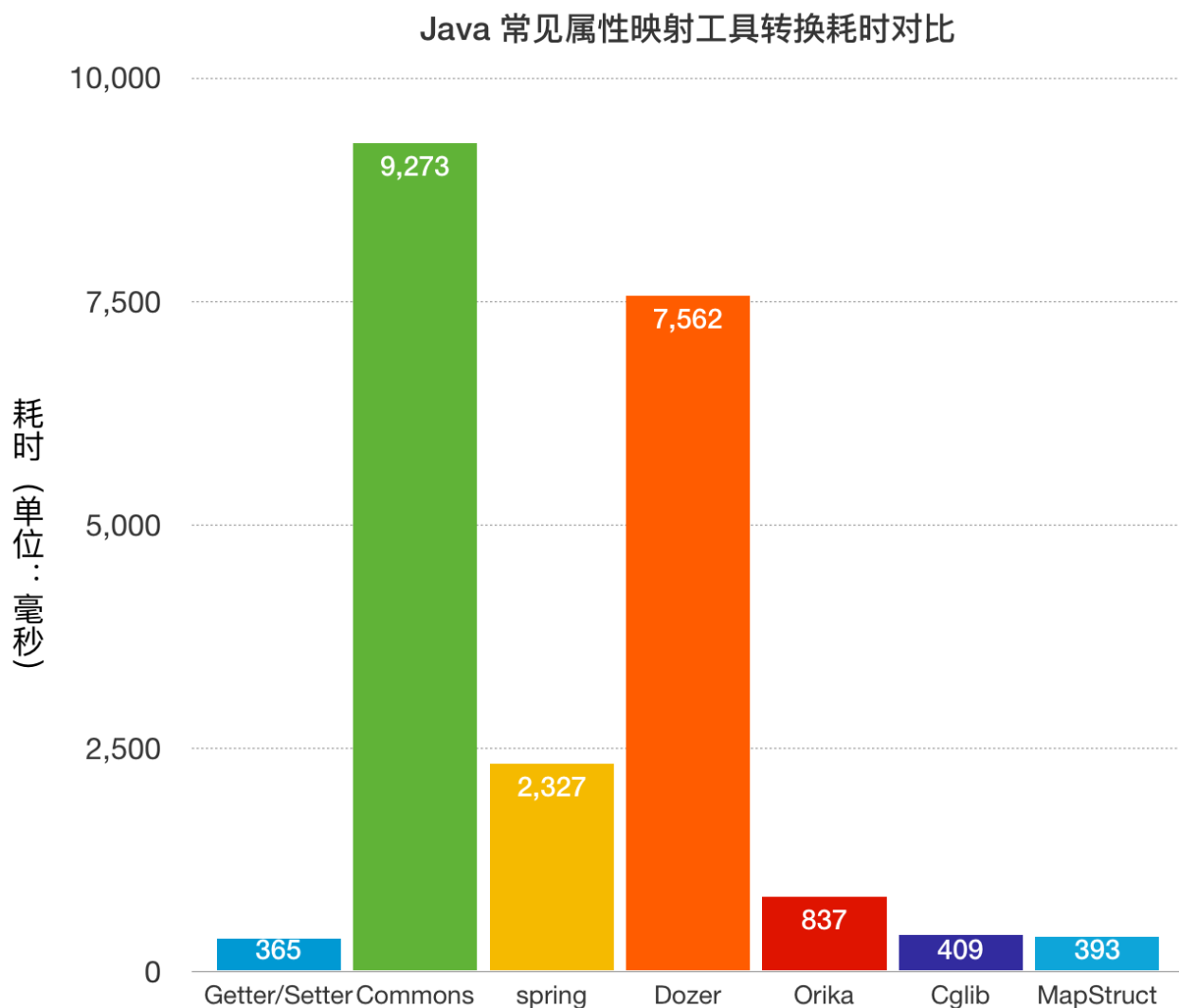
```

@Data
public class UserDTO {
    private String name;
    private Integer age;
    private String nickName;
    private Date birthDay;
}

```

使用 `easyrandom`（后面的单元测试环节会重点介绍）构造 10 万个 `UserDO` 随机对象进行性能对比。`spring` 版本为 5.1.8.RELEASE，`dozer` 版本为 5.5.1，`orika-core` 版本为 1.5.4，`cglib` 版本为 3.2.12，`commons-lang3` 包版本为 3.9，10 次运行取平均值，最终结果如下：

1. 普通 Getter/Setter 耗时 365ms;
2. `org.apache.commons.beanutils.BeanUtils#copyProperties` 耗时 9s273ms;
3. `org.springframework.beans.BeanUtils#copyProperties(java.lang.Object, java.lang.Object)` 耗时 2s327ms;
4. `org.dozer.Mapper#map(java.lang.Object, java.lang.Class<T>)` 耗时 9s271ms;
5. `ma.glasnost.orka.MapperFacade#map(S, D)` 耗时 837ms;
6. `net.sf.cglib.beans.BeanCopier#copy` 耗时 409ms;



由于机器的性能不同结果会有偏差，本实验并没有将转换框架的功能发挥到到极致，也没有使用更复杂的对象进行对比，因此本实验的结果仅作为一个大致的参考。

我们仍然可以大致得出结论：采用字节码增强技术的 **Java** 属性转换工具和普通的 **Getter/Setter** 方法性能相差不多，甚至比 **Getter/Setter** 效率还高，反射的性能相对较差。

因此从性能来讲首推 **Getter/Setter** 方式（含 **MapStruct**），其次是 **cglib**。

3. 用哪个？为什么？怎么用？

3.1 用什么？为什么？

通过以上的分析，我们对 **Java** 属性转换有了一个基本的了解。

选择太多往往会比较纠结，实际开发中我们用哪种更好呢？

我在业务代码中见到同事用的转换工具主要有 **Getter/Setter** 方式、**orika** 和 **commons/spring** 的属性拷贝工具。

属性转换工具的优势：用起来方便，往往一行行代码就实现多属性的转换，而且属性不对应可以通过注解或者修改配置方式自动适配，功能非常强大。

属性转换工具的缺点：

1. 多次对象映射（从 A 映射到 B，再从 B 映射到 C）如果属性不完全一致容易出错；
2. 有些转换工具，属性类型不一致自动转换容易出现意想不到的 BUG；
3. 基于反射和字节码增强技术的映射工具实现的映射，对一个类属性的修改不容易感知到对其它转换类的影响。

我们可以想想这样一个场景：

一个 `UserDO` 如果属性超多，转换到 `UserDTO` 再被转换成 `UserVO`。如果你修改 `UserDTO` 的一个属性命名，其它类待映射的类新增的对应属性有一个字母写错了，编译期间不容易发现问题，造成 BUG。

如果使用原始的 `Getter/Setter` 方式转换，修改了 `UserDO` 的属性，那么转换代码就会报错，编译都不通过，这样就可以逆向提醒我们注意到属性的变动的影响。

因此强烈建议使用定义转换类和转换函数，使用插件实现转换，不需要引入其它库，降低了复杂性，可以支持更灵活的映射。

大家可以想想这种场景：

如果一个 A 映射到 B，B 有两个属性来自 C，一个属性来自于传参或者计算等。

此时自定义转换函数更方便。

如果使用属性映射工具推荐使用 **MapStruct**，更安全一些，转换效率也很高。

3.2 怎么用？

每种对象属性映射工具的具体用法，大家可以参考官网文档或源码中的测试类，这里主要讲映射的工具类该如何定义。

为了避免转换函数散落到多个业务类中，不容易复用，我们可以在工具包或者对象包下定义一个专门的转换包（`converter` 或者 `mapper` 包），在转换的包下编写转换工具类。

第一种方式：可以实现 `org.springframework.core.convert.converter.Converter` 接口。

代码如下：

```
import org.springframework.core.convert.converter.Converter;

public class UserDO2DTOConverter implements Converter<UserDO, UserDTO> {

    @Override
    public UserDTO convert(UserDO source) {
        UserDTO userDTO = new UserDTO();
        userDTO.setName(source.getName());
        userDTO.setAge(source.getAge());
        userDTO.setNickName(source.getNickName());
        userDTO.setBirthDay(source.getBirthDay());
        return userDTO;
    }
}
```

上述只能实现单向转换，我们如果想双向转换该怎么做呢？

这时候我们可以采用第二种方式，可以继承 `com.google.common.base.Converter` 接口实现双向转换。

```
import com.imooc.basic.converter.entity.UserDO;
import com.imooc.basic.converter.entity.UserDTO;
import com.google.common.base.Converter;

public class UserDO2DTOConverter extends Converter<UserDO, UserDTO> {

    @Override
    protected UserDTO doForward(UserDO userDO) {
        UserDTO userDTO = new UserDTO();
        userDTO.setName(userDO.getName());
        userDTO.setAge(userDO.getAge());
        userDTO.setNickName(userDO.getNickName());
        userDTO.setBirthDay(userDO.getBirthDay());
        return userDTO;
    }

    @Override
    protected UserDO doBackward(UserDTO userDTO) {
        UserDO userDO = new UserDO();
        userDO.setName(userDTO.getName());
        userDO.setAge(userDTO.getAge());
        userDO.setNickName(userDTO.getNickName());
        userDO.setBirthDay(userDTO.getBirthDay());
        return userDO;
    }
}
```

我更建议采用以下这种方式，因为上述方式只能实现单向或者双向转换，如果更多种对象类型的转换就无能为力。

此时可以自定义接口或者抽象类，支持更多种对象的转换。

更推荐大家直接定义某个对象的转换器类，在其内部编写该对象各层对象的转换函数：

```

public class UserConverter {

    public static UserDTO convertToDTO(UserDO source) {
        UserDTO userDTO = new UserDTO();
        userDTO.setName(source.getName());
        userDTO.setAge(source.getAge());
        userDTO.setNickName(source.getNickName());
        userDTO.setBirthDay(source.getBirthDay());
        return userDTO;
    }

    public static UserDO convertToDO(UserDO source) {
        UserDO userDO = new UserDO();
        userDO.setId(source.getId());
        userDO.setName(source.getName());
        userDO.setAge(source.getAge());
        userDO.setNickName(source.getNickName());
        userDO.setBirthDay(source.getBirthDay());
        return userDO;
    }

    // 转换成UserVO等
}

```

有些同学可能会抱怨，**Getter/Setter** 方式转换函数编写非常耗时而且容易漏，怎么办？

这里推荐一个 IDEA 插件：**GenerateAllSetter** 或者 **GenerateO2O**。

定义好转换函数之后，鼠标放在 **convertToDTO** 上使用快捷键，选择“generate setter getter converter”即可实现根据目标对象的属性名适配同名源对象自动填充，注意如果有个别属性不对应，需手动转换。

另外推荐使用 **mapstruct** 实现对象属性映射：

```

@Mapper
public interface UserMapper {
    UserMapper INSTANCE = Mappers.getMapper(UserMapper.class);

    UserDTO userDo2Dto(UserDO userDO);
}

```

使用时一行代码即可搞定：

```

UserDTO userDTO = UserMapper.INSTANCE.userDo2Dto(userDO);

```

相当于把 IDE 插件自动生成的这部分任务改为了使用注解，通过插件编译时自动生成。

4. 总结

本节主要介绍了 **Java** 属性映射的各种方式，介绍了每种方式背后的原理，并简单对比了各种属性映射方式的耗时。本小节还给出了属性转换工具的推荐定义方式。希望大家在实际的开发中，除了考虑性能外，兼顾考虑安全性和可维护性。

下节将介绍过期代码的正确处理方式。

5. 课后练习

自定义一个 **OrderDO** 和 **OrderDTO** 两个类，自定义属性，使用 **StructMap** 实现属性映射。

}



05 分层领域模型使用解读

07 过期类、属性、接口的正确处理姿势

