Java 实体映射工具 MapStruct

<https://blog.csdn.net/zhige_me/article/details/80699784>

DO（业务实体对象），DTO（数据传输对象）。

在一个成熟的工程中，尤其是现在的分布式系统中，应用与应用之间，还有单独的应用细分模块之后，DO 一般不会让外部依赖，这时候需要在提供对外接口的模块里放 DTO 用于对象传输，也即是 DO 对象对内，DTO对象对外，DTO 可以根据业务需要变更，并不需要映射 DO 的全部属性。

这种 对象与对象之间的互相转换，就需要有一个专门用来解决转换问题的工具，毕竟每一个字段都 get/set 会很麻烦。

MapStruct 就是这样的一个属性映射工具，只需要定义一个 Mapper 接口，MapStruct 就会自动实现这个映射接口，避免了复杂繁琐的映射实现。MapStruct官网地址

**基本映射**

我们先从一些基本的映射开始。我们会创建一个Doctor对象和一个DoctorDto。为了方便起见，它们的属性字段都使用相同的名称：

public class Doctor {

private int id;

private String name;

// getters and setters or builder

}

public class DoctorDto {

private int id;

private String name;

// getters and setters or builder

}

现在，为了在这两者之间进行映射，我们要创建一个DoctorMapper接口。对该接口使用@Mapper注解，MapStruct就会知道这是两个类之间的映射器。

@Mapper

public interface DoctorMapper {

DoctorMapper INSTANCE = Mappers.getMapper(DoctorMapper.class);

DoctorDto toDto(Doctor doctor);

}

这段代码中创建了一个DoctorMapper类型的实例INSTANCE，在生成对应的实现代码后，这就是我们调用的“入口”。

我们在接口中定义了toDto()方法，该方法接收一个Doctor实例为参数，并返回一个DoctorDto实例。这足以让MapStruct知道我们想把一个Doctor实例映射到一个DoctorDto实例。

当我们构建/编译应用程序时，MapStruct注解处理器插件会识别出DoctorMapper接口并为其生成一个实现类。

public class DoctorMapperImpl implements DoctorMapper {

@Override

public DoctorDto toDto(Doctor doctor) {

if ( doctor == null ) {

return null;

}

DoctorDtoBuilder doctorDto = DoctorDto.builder();

doctorDto.id(doctor.getId());

doctorDto.name(doctor.getName());

return doctorDto.build();

}

}

DoctorMapperImpl类中包含一个toDto()方法，将我们的Doctor属性值映射到DoctorDto的属性字段中。如果要将Doctor实例映射到一个DoctorDto实例，可以这样写：

DoctorDto doctorDto = DoctorMapper.INSTANCE.toDto(doctor);

*注意：你可能也注意到了上面实现代码中的DoctorDtoBuilder。因为builder代码往往比较长，为了简洁起见，这里省略了builder模式的实现代码。如果你的类中包含Builder，MapStruct会尝试使用它来创建实例；如果没有的话，MapStruct将通过new关键字进行实例化。*

**不同字段间映射**

通常，模型和DTO的字段名不会完全相同。由于团队成员各自指定命名，以及针对不同的调用服务，开发者对返回信息的打包方式选择不同，名称可能会有轻微的变化。

MapStruct通过@Mapping注解对这类情况提供了支持。

**1 不同属性名称**

我们先更新Doctor类，添加一个属性specialty：

public class Doctor {

private int id;

private String name;

private String specialty;

// getters and setters or builder

}

在DoctorDto类中添加一个specialization属性：

public class DoctorDto {

private int id;

private String name;

private String specialization;

// getters and setters or builder

}

现在，我们需要让 DoctorMapper 知道这里的不一致。我们可以使用 @Mapping 注解，并设置其内部的 source 和 target 标记分别指向不一致的两个字段。

@Mapper

public interface DoctorMapper {

DoctorMapper INSTANCE = Mappers.getMapper(DoctorMapper.class);

**@Mapping(source = "doctor.specialty", target = "specialization")**

DoctorDto toDto(Doctor doctor);

}

这个注解代码的含义是：Doctor中的specialty字段对应于DoctorDto类的 specialization 。

编译之后，会生成如下实现代码：

public class DoctorMapperImpl implements DoctorMapper {

@Override

public DoctorDto toDto(Doctor doctor) {

if (doctor == null) {

return null;

}

DoctorDtoBuilder doctorDto = DoctorDto.builder();

doctorDto.specialization(doctor.getSpecialty());

doctorDto.id(doctor.getId());

doctorDto.name(doctor.getName());

return doctorDto.build();

}

}

**2 多个源类**

有时，单个类不足以构建DTO，我们可能希望将多个类中的值聚合为一个DTO，供终端用户使用。这也可以通过在@Mapping注解中设置适当的标志来完成。

我们先新建另一个对象 Education:

public class Education {

private String degreeName;

private String institute;

private Integer yearOfPassing;

// getters and setters or builder

}

然后向 DoctorDto中添加一个新的字段：

public class DoctorDto {

private int id;

private String name;

**private String degree;**

private String specialization;

// getters and setters or builder

}

接下来，将 DoctorMapper 接口更新为如下代码：

@Mapper

public interface DoctorMapper {

DoctorMapper INSTANCE = Mappers.getMapper(DoctorMapper.class);

@Mapping(source = "doctor.specialty", target = "specialization")

@Mapping(source = "education.degreeName", target = "degree")

**DoctorDto toDto(Doctor doctor, Education education);**

}

我们添加了另一个@Mapping注解，并将其source设置为Education类的degreeName，将target设置为DoctorDto类的degree字段。

如果 Education 类和 Doctor 类包含同名的字段，我们必须让映射器知道使用哪一个，否则它会抛出一个异常。举例来说，如果两个模型都包含一个id字段，我们就要选择将哪个类中的id映射到DTO属性中。

**3 子对象映射**

多数情况下，POJO中不会只包含基本数据类型，其中往往会包含其它类。比如说，一个Doctor类中会有多个患者类：

public class Patient {

private int id;

private String name;

// getters and setters or builder

}

在Doctor中添加一个患者列表List：

public class Doctor {

private int id;

private String name;

private String specialty;

**private List<Patient> patientList**;

// getters and setters or builder

}

因为Patient需要转换，为其创建一个对应的DTO：

public class PatientDto {

private int id;

private String name;

// getters and setters or builder

}

最后，在 DoctorDto 中新增一个存储 PatientDto的列表：

public class DoctorDto {

private int id;

private String name;

private String degree;

private String specialization;

**private List<PatientDto> patientDtoList;**

// getters and setters or builder

}

在修改 DoctorMapper之前，我们先创建一个支持 Patient 和 PatientDto 转换的映射器接口：

@Mapper

public interface PatientMapper {

PatientMapper INSTANCE = Mappers.getMapper(PatientMapper.class);

PatientDto toDto(Patient patient);

}

这是一个基本映射器，只会处理几个基本数据类型。

然后，我们再来修改 DoctorMapper 处理一下患者列表：

@Mapper(**uses = {PatientMapper.class}**)

public interface DoctorMapper {

DoctorMapper INSTANCE = Mappers.getMapper(DoctorMapper.class);

**@Mapping(source = "doctor.patientList", target = "patientDtoList")**

@Mapping(source = "doctor.specialty", target = "specialization")

DoctorDto toDto(Doctor doctor);

}

因为我们要处理另一个需要映射的类，所以这里设置了@Mapper注解的uses标志，这样现在的 @Mapper 就可以使用另一个 @Mapper映射器。我们这里只加了一个，但你想在这里添加多少class/mapper都可以。

我们已经添加了uses标志，所以在为DoctorMapper接口生成映射器实现时，MapStruct 也会把 Patient 模型转换成 PatientDto ——因为我们已经为这个任务注册了 PatientMapper。

编译查看最新想实现代码：

public class DoctorMapperImpl implements DoctorMapper {

private final PatientMapper patientMapper = Mappers.getMapper( PatientMapper.class );

@Override

public DoctorDto toDto(Doctor doctor) {

if ( doctor == null ) {

return null;

}

DoctorDtoBuilder doctorDto = DoctorDto.builder();

doctorDto.patientDtoList( patientListToPatientDtoList(doctor.getPatientList()));

doctorDto.specialization( doctor.getSpecialty() );

doctorDto.id( doctor.getId() );

doctorDto.name( doctor.getName() );

return doctorDto.build();

}

protected List<PatientDto> patientListToPatientDtoList(List<Patient> list) {

if ( list == null ) { return null; }

List<PatientDto> list1 = new ArrayList<PatientDto>( list.size() );

for ( Patient patient : list ) {

list1.add( patientMapper.toDto( patient ) );

}

return list1;

}

}

显然，除了toDto()映射方法外，最终实现中还添加了一个新的映射方法——patientListToPatientDtoList()。这个方法是在没有显式定义的情况下添加的，只是因为我们把PatientMapper添加到了DoctorMapper中。

该方法会遍历一个Patient列表，将每个元素转换为PatientDto，并将转换后的对象添加到DoctorDto对象内中的列表中。

**4 更新现有实例**

有时，我们希望用DTO的最新值更新一个模型中的属性，对目标对象(我们的例子中是DoctorDto)使用@MappingTarget注解，就可以更新现有的实例.

@Mapper(uses = {PatientMapper.class})

public interface DoctorMapper {

DoctorMapper INSTANCE = Mappers.getMapper(DoctorMapper.class);

@Mapping(source = "doctorDto.patientDtoList", target = "patientList")

@Mapping(source = "doctorDto.specialization", target = "specialty")

void updateModel(DoctorDto doctorDto, @MappingTarget Doctor doctor);

}

重新生成实现代码，就可以得到updateModel()方法：

public class DoctorMapperImpl implements DoctorMapper {

@Override

public void updateModel(DoctorDto doctorDto, Doctor doctor) {

if (doctorDto == null) {

return;

}

if (doctor.getPatientList() != null) {

List<Patient> list =

patientDtoListToPatientList(doctorDto.getPatientDtoList());

if (list != null) {

doctor.getPatientList().clear();

doctor.getPatientList().addAll(list);

}

else {

doctor.setPatientList(null);

}

}

else {

List<Patient> list = patientDtoListToPatientList(doctorDto.getPatientDtoList());

if (list != null) {

doctor.setPatientList(list);

}

}

doctor.setSpecialty(doctorDto.getSpecialization());

doctor.setId(doctorDto.getId());

doctor.setName(doctorDto.getName());

}

}

值得注意的是，由于患者列表是该模型中的子实体，因此患者列表也会进行更新。

**数据类型转换**

**1 数据类型映射**

MapStruct支持source和target属性之间的数据类型转换。它还提供了基本类型及其相应的包装类之间的自动转换。

自动类型转换适用于：

基本类型及其对应的包装类之间。比如， int 和 Integer， float 和 Float， long 和 Long，boolean 和 Boolean 等。

任意基本类型与任意包装类之间。如 int 和 long， byte 和 Integer 等。

所有基本类型及包装类与String之间。如 boolean 和 String， Integer 和 String， float 和 String 等。

枚举和String之间。

Java大数类型(java.math.BigInteger， java.math.BigDecimal) 和Java基本类型(包括其包装类)与String之间。

其它情况详见MapStruct官方文档。

因此，在生成映射器代码的过程中，如果源字段和目标字段之间属于上述任何一种情况，则MapStrcut会自行处理类型转换。

我们修改 PatientDto ，新增一个 dateofBirth字段：

public class PatientDto {

private int id;

private String name;

**private LocalDate dateOfBirth**;

// getters and setters or builder

}

另一方面，加入 Patient 对象中有一个String 类型的 dateOfBirth ：

public class Patient {

private int id;

private String name;

**private String dateOfBirth;**

// getters and setters or builder

}

在两者之间创建一个映射器：

@Mapper

public interface PatientMapper {

**@Mapping(source = "dateOfBirth", target = "dateOfBirth", dateFormat = "dd/MMM/yyyy")**

Patient toModel(PatientDto patientDto);

}

当对日期进行转换时，我们也可以使用 dateFormat 设置格式声明。生成的实现代码形式大致如下：

public class PatientMapperImpl implements PatientMapper {

@Override

public Patient toModel(PatientDto patientDto) {

if (patientDto == null) { return null; }

PatientBuilder patient = Patient.builder();

if (patientDto.getDateOfBirth() != null) {

patient.dateOfBirth(DateTimeFormatter.ofPattern("dd/MMM/yyyy")

.format(patientDto.getDateOfBirth()));

}

patient.id(patientDto.getId());

patient.name(patientDto.getName());

return patient.build();

}

}

可以看到，这里使用了 dateFormat 声明的日期格式。如果我们没有声明格式的话，MapStruct会使用 LocalDate的默认格式，大致如下：

if (patientDto.getDateOfBirth() != null) {

patient.dateOfBirth(DateTimeFormatter.ISO\_LOCAL\_DATE

.format(patientDto.getDateOfBirth()));

}

**2 数字格式转换**

上面的例子中可以看到，在进行日期转换的时候，可以通过dateFormat标志指定日期的格式。

除此之外，对于数字的转换，也可以使用numberFormat指定显示格式：

// 数字格式转换示例

@Mapping(source = "price", target = "price", numberFormat = "$#.00")

**3 枚举映射**

枚举映射的工作方式与字段映射相同。MapStruct会对具有相同名称的枚举进行映射，这一点没有问题。但是，对于具有不同名称的枚举项，我们需要使用@ValueMapping注解。同样，这与普通类型的@Mapping注解也相似。

我们先创建两个枚举。第一个是 PaymentType:

public enum PaymentType {

CASH,

CHEQUE,

CARD\_VISA,

CARD\_MASTER,

CARD\_CREDIT

}

比如说，这是一个应用内可用的支付方式，现在我们要根据这些选项创建一个更一般、有限的识图：

public enum PaymentTypeView {

CASH,

CHEQUE,

CARD

}

现在，我们创建这两个enum之间的映射器接口：

@Mapper

public interface PaymentTypeMapper {

PaymentTypeMapper INSTANCE = Mappers.getMapper(PaymentTypeMapper.class);

@ValueMappings({

@ValueMapping(source = "CARD\_VISA", target = "CARD"),

@ValueMapping(source = "CARD\_MASTER", target = "CARD"),

@ValueMapping(source = "CARD\_CREDIT", target = "CARD")

})

PaymentTypeView paymentTypeToPaymentTypeView(PaymentType paymentType);

}

这个例子中，我们设置了一般性的CARD值，和更具体的 CARD\_VISA, CARD\_MASTER 和 CARD\_CREDIT 。两个枚举间的枚举项数量不匹配—— PaymentType 有5个值，而 PaymentTypeView 只有3个。

为了在这些枚举项之间建立桥梁，我们可以使用@ValueMappings注解，该注解中可以包含多个@ValueMapping注解。这里，我们将source设置为三个具体枚举项之一，并将target设置为CARD。

MapStruct自然会处理这些情况：

public class PaymentTypeMapperImpl implements PaymentTypeMapper {

@Override

public PaymentTypeView paymentTypeToPaymentTypeView(PaymentType paymentType) {

if (paymentType == null) {

return null;

}

PaymentTypeView paymentTypeView;

switch (paymentType) {

case CARD\_VISA: paymentTypeView = PaymentTypeView.CARD;

break;

case CARD\_MASTER: paymentTypeView = PaymentTypeView.CARD;

break;

case CARD\_CREDIT: paymentTypeView = PaymentTypeView.CARD;

break;

case CASH: paymentTypeView = PaymentTypeView.CASH;

break;

case CHEQUE: paymentTypeView = PaymentTypeView.CHEQUE;

break;

default: throw new IllegalArgumentException( "Unexpected enum constant: " + paymentType );

}

return paymentTypeView;

}

}

CASH和CHEQUE默认转换为对应值，特殊的 CARD 值通过switch循环处理。

但是，如果你要将很多值转换为一个更一般的值，这种方式就有些不切实际了。其实我们不必手动分配每一个值，只需要让MapStruct将所有剩余的可用枚举项（在目标枚举中找不到相同名称的枚举项），直接转换为对应的另一个枚举项。

可以通过 MappingConstants实现这一点：

@ValueMapping(source = MappingConstants.ANY\_REMAINING, target = "CARD")

PaymentTypeView paymentTypeToPaymentTypeView(PaymentType paymentType);

在这个例子中，完成默认映射之后，所有剩余（未匹配）的枚举项都会映射为CARD：

@Override

public PaymentTypeView paymentTypeToPaymentTypeView(PaymentType paymentType) {

if ( paymentType == null ) {

return null;

}

PaymentTypeView paymentTypeView;

switch ( paymentType ) {

case CASH: paymentTypeView = PaymentTypeView.CASH;

break;

case CHEQUE: paymentTypeView = PaymentTypeView.CHEQUE;

break;

default: paymentTypeView = PaymentTypeView.CARD;

}

return paymentTypeView;

}

还有一种选择是使用ANY UNMAPPED：

@ValueMapping(source = MappingConstants.ANY\_UNMAPPED, target = "CARD")

PaymentTypeView paymentTypeToPaymentTypeView(PaymentType paymentType);

采用这种方式时，MapStruct不会像前面那样先处理默认映射，再将剩余的枚举项映射到target值。而是，直接将所有未通过@ValueMapping注解做显式映射的值都转换为target值。

**集合映射**

简单来说，使用MapStruct处理集合映射的方式与处理简单类型相同。

我们创建一个简单的接口或抽象类并声明映射方法。 MapStruct将根据我们的声明自动生成映射代码。 通常，生成的代码会遍历源集合，将每个元素转换为目标类型，并将每个转换后元素添加到目标集合中。

**1 List映射**

我们先定义一个新的映射方法：

@Mapper

public interface DoctorMapper {

List<DoctorDto> map(List<Doctor> doctor);

}

生成的代码大致如下：

public class DoctorMapperImpl implements DoctorMapper {

@Override

public List<DoctorDto> map(List<Doctor> doctor) {

if ( doctor == null ) {

return null;

}

List<DoctorDto> list = new ArrayList<DoctorDto>( doctor.size() );

for ( Doctor doctor1 : doctor ) {

list.add( doctorToDoctorDto( doctor1 ) );

}

return list;

}

protected DoctorDto doctorToDoctorDto(Doctor doctor) {

if ( doctor == null ) {

return null;

}

DoctorDto doctorDto = new DoctorDto();

doctorDto.setId( doctor.getId() );

doctorDto.setName( doctor.getName() );

doctorDto.setSpecialization( doctor.getSpecialization() );

return doctorDto;

}

}

可以看到，MapStruct为我们自动生成了从Doctor到DoctorDto的映射方法。

但是需要注意，如果我们在DTO中新增一个字段fullName，生成代码时会出现错误：

警告: Unmapped target property: "fullName".

基本上，这意味着MapStruct在当前情况下无法为我们自动生成映射方法。因此，我们需要手动定义Doctor和DoctorDto之间的映射方法。具体参考之前的小节。

**2 Set和Map映射**

Set与Map型数据的处理方式与List相似。按照以下方式修改DoctorMapper：

@Mapper

public interface DoctorMapper {

Set<DoctorDto> setConvert(Set<Doctor> doctor);

Map<String, DoctorDto> mapConvert(Map<String, Doctor> doctor);

}

生成的最终实现代码如下：

public class DoctorMapperImpl implements DoctorMapper {

@Override

public Set<DoctorDto> setConvert(Set<Doctor> doctor) {

if ( doctor == null ) { return null; }

Set<DoctorDto> set =

new HashSet<DoctorDto>( Math.max( (int) ( doctor.size() / .75f ) + 1, 16 ) );

for ( Doctor doctor1 : doctor ) {

set.add( doctorToDoctorDto( doctor1 ) );

}

return set;

}

@Override

public Map<String, DoctorDto> mapConvert(Map<String, Doctor> doctor) {

if ( doctor == null ) { return null; }

Map<String, DoctorDto> map =

new HashMap<String, DoctorDto>( Math.max( (int) ( doctor.size() / .75f ) + 1, 16 ) );

for ( java.util.Map.Entry<String, Doctor> entry : doctor.entrySet() ) {

String key = entry.getKey();

DoctorDto value = doctorToDoctorDto( entry.getValue() );

map.put( key, value );

}

return map;

}

protected DoctorDto doctorToDoctorDto(Doctor doctor) {

if ( doctor == null ) { return null; }

DoctorDto doctorDto = new DoctorDto();

doctorDto.setId( doctor.getId() );

doctorDto.setName( doctor.getName() );

doctorDto.setSpecialization( doctor.getSpecialization() );

return doctorDto;

}

}

与List映射类似，MapStruct自动生成了Doctor转换为DoctorDto的映射方法。

**3 集合映射策略**

很多场景中，我们需要对具有父子关系的数据类型进行转换。通常来说，会有一个数据类型（父），其字段是另一个数据类型（子）的集合。

对于这种情况，MapStruct提供了一种方法来选择如何将子类型设置或添加到父类型中。具体来说，就是@Mapper注解中的**collectionMappingStrategy**属性，该属性可以取值为ACCESSOR\_ONLY， SETTER\_PREFERRED， ADDER\_PREFERRED 或TARGET\_IMMUTABLE。

这些值分别表示不同的为子类型集合赋值的方式。默认值是ACCESSOR\_ONLY，这意味着只能使用访问器来设置子集合。

当父类型中的Collection字段setter方法不可用，但我们有一个子类型add方法时，这个选项就派上用场了；另一种有用的情况是父类型中的Collection字段是不可变的。

我们新建一个类：

public class Hospital {

private List<Doctor> doctors;

// getters and setters or builder

}

同时定义一个映射目标DTO类，同时定义子类型集合字段的getter、setter和adder：

public class HospitalDto {

private List<DoctorDto> doctors;

// 子类型集合字段getter

public List<DoctorDto> getDoctors() {

return doctors;

}

// 子类型集合字段setter

public void setDoctors(List<DoctorDto> doctors) {

this.doctors = doctors; }

// 子类型数据adder

public void addDoctor(DoctorDto doctorDTO) {

if (doctors == null) { doctors = new ArrayList<>(); }

doctors.add(doctorDTO);

}

}

创建对应的映射器：

@Mapper(uses = DoctorMapper.class)

public interface HospitalMapper {

HospitalMapper INSTANCE = Mappers.getMapper(HospitalMapper.class);

HospitalDto toDto(Hospital hospital);

}

生成的最终实现代码为：

public class HospitalMapperImpl implements HospitalMapper {

@Override

public HospitalDto toDto(Hospital hospital) {

if ( hospital == null ) {

return null;

}

HospitalDto hospitalDto = new HospitalDto();

hospitalDto.setDoctors( doctorListToDoctorDtoList( hospital.getDoctors() ) );

return hospitalDto;

}

}

可以看到，在默认情况下采用的策略是ACCESSOR\_ONLY，使用setter方法setDoctors()向HospitalDto对象中写入列表数据。

相对的，如果使用 ADDER\_PREFERRED 作为映射策略：

@Mapper(collectionMappingStrategy = CollectionMappingStrategy.ADDER\_PREFERRED,

uses = DoctorMapper.class)

public interface HospitalMapper {

HospitalMapper INSTANCE = Mappers.getMapper(HospitalMapper.class);

HospitalDto toDto(Hospital hospital);

}

此时，会使用adder方法逐个将转换后的子类型DTO对象加入父类型的集合字段中。

public class CompanyMapperAdderPreferredImpl implements CompanyMapperAdderPreferred {

private final EmployeeMapper employeeMapper = Mappers.getMapper( EmployeeMapper.class );

@Override

public CompanyDTO map(Company company) {

if ( company == null ) { return null; }

CompanyDTO companyDTO = new CompanyDTO();

if ( company.getEmployees() != null ) {

for ( Employee employee : company.getEmployees() ) {

companyDTO.addEmployee( employeeMapper.map( employee ) );

}

}

return companyDTO;

}

}

如果目标DTO中既没有setter方法也没有adder方法，会先通过getter方法获取子类型集合，再调用集合的对应接口添加子类型对象。

可以在参考文档中看到不同类型的DTO定义（是否包含setter方法或adder方法），采用不同的映射策略时，所使用的添加子类型到集合中的方式。

**4 目标集合实现类型**

MapStruct支持将集合接口作为映射方法的目标类型。

在这种情况下，在生成的代码中会使用一些集合接口默认实现。 例如，上面的示例中，List的默认实现是ArrayList。

**常见接口及其对应的默认实现如下：**

Interface type Implementation type

你可以在参考文档中找到MapStruct支持的所有接口列表，以及每个接口对应的默认实现类型。

**进阶操作**

**1 依赖注入**

到目前为止，我们一直在通过getMapper()方法访问生成的映射器：

DoctorMapper INSTANCE = Mappers.getMapper(DoctorMapper.class);

但是，如果你使用的是Spring，只需要简单修改映射器配置，就可以像常规依赖项一样注入映射器。

修改 DoctorMapper 以支持Spring框架：

@Mapper(componentModel = "spring")

public interface DoctorMapper {}

在@Mapper注解中添加（componentModel = "spring"），是为了告诉MapStruct，在生成映射器实现类时，我们希望它能支持通过Spring的依赖注入来创建。现在，就不需要在接口中添加 INSTANCE 字段了。

这次生成的 DoctorMapperImpl 会带有 @Component 注解：

@Component

public class DoctorMapperImpl implements DoctorMapper {}

只要被标记为@Component，Spring就可以把它作为一个bean来处理，你就可以在其它类（如控制器）中通过@Autowire注解来使用它：

@Controller

public class DoctorController() {

@Autowired

private DoctorMapper doctorMapper;

}

如果你不使用Spring, MapStruct也支持Java CDI：

@Mapper(componentModel = "cdi")

public interface DoctorMapper {}

**2 添加默认值**

@Mapping 注解有两个很实用的标志就是常量 constant 和默认值 defaultValue 。无论source如何取值，都将始终使用常量值； 如果source取值为null，则会使用默认值。

修改一下 DoctorMapper ，添加一个 constant 和一个 defaultValue ：

@Mapper(uses = {PatientMapper.class}, componentModel = "spring")

public interface DoctorMapper {

@Mapping(target = "id", **constant = "-1"**)

@Mapping(source = "doctor.patientList", target = "patientDtoList")

@Mapping(source = "doctor.specialty", target = "specialization", **defaultValue = "Information Not Available"**)

DoctorDto toDto(Doctor doctor);

}

如果specialty不可用，我们会替换为"Information Not Available"字符串，此外，我们将id硬编码为-1。

生成代码如下：

@Component

public class DoctorMapperImpl implements DoctorMapper {

@Autowired

private PatientMapper patientMapper;

@Override

public DoctorDto toDto(Doctor doctor) {

if (doctor == null) { return null; }

DoctorDto doctorDto = new DoctorDto();

if (doctor.getSpecialty() != null) {

doctorDto.setSpecialization(doctor.getSpecialty());

}

else {

doctorDto.setSpecialization("Information Not Available");

}

doctorDto.setPatientDtoList(patientListToPatientDtoList(doctor.getPatientList()));

doctorDto.setName(doctor.getName());

doctorDto.setId(-1);

return doctorDto;

}

}

可以看到，如果 doctor.getSpecialty() 返回值为null，则将specialization设置为我们的默认信息。无论任何情况，都会对 id赋值，因为这是一个constant。

**3 添加表达式**

MapStruct甚至允许在@Mapping注解中输入Java表达式。你可以设置 defaultExpression （ source 取值为 null时生效），或者一个expression（类似常量，永久生效）。

在 Doctor 和 DoctorDto两个类中都加了两个新属性，一个是 String 类型的 externalId ，另一个是LocalDateTime类型的 appointment ，两个类大致如下：

public class Doctor {

private int id;

private String name;

private String externalId;

private String specialty;

private LocalDateTime availability;

private List<Patient> patientList;

// getters and setters or builder

}

public class DoctorDto {

private int id;

private String name;

private String externalId;

private String specialization;

private LocalDateTime availability;

private List<PatientDto> patientDtoList;

// getters and setters or builder

}

修改 DoctorMapper：

@Mapper(uses = {PatientMapper.class}, componentModel = "spring", imports = {LocalDateTime.class, UUID.class})

public interface DoctorMapper {

@Mapping(target = "externalId", **expression = "java(UUID.randomUUID().toString())"**)

@Mapping(source = "doctor.availability", target = "availability",

**defaultExpression = "java(LocalDateTime.now())"**)

@Mapping(source = "doctor.patientList", target = "patientDtoList")

@Mapping(source = "doctor.specialty", target = "specialization")

DoctorDto toDtoWithExpression(Doctor doctor);

}

可以看到，这里将 externalId的值设置为 java(UUID.randomUUID().toString()) ，如果源对象中没有 availability 属性，则会把目标对象中的 availability 设置为一个新的 LocalDateTime对象。

由于表达式只是字符串，我们必须在表达式中指定使用的类。但是这里的表达式并不是最终执行的代码，只是一个字母的文本值。因此，我们要在 @Mapper 中添加 imports = {LocalDateTime.class, UUID.class} 。

**4 添加自定义方法**

到目前为止，我们一直使用的策略是添加一个“占位符”方法，并期望MapStruct能为我们实现它。其实我们还可以向接口中添加自定义的default方法，也可以通过default方法直接实现一个映射。然后我们可以通过实例直接调用该方法，没有任何问题。

为此，我们创建一个 DoctorPatientSummary类，其中包含一个 Doctor 及其 Patient列表的汇总信息：

public class DoctorPatientSummary {

private int doctorId;

private int patientCount;

private String doctorName;

private String specialization;

private String institute;

private List<Integer> patientIds;

// getters and setters or builder

}

接下来，我们在 DoctorMapper中添加一个default方法，该方法会将 Doctor 和 Education 对象转换为一个 DoctorPatientSummary:

@Mapper

public interface DoctorMapper {

default DoctorPatientSummary toDoctorPatientSummary(Doctor doctor, Education education) {

return DoctorPatientSummary.builder()

.doctorId(doctor.getId())

.doctorName(doctor.getName())

.patientCount(doctor.getPatientList().size())

.patientIds(doctor.getPatientList()

.stream()

.map(Patient::getId)

.collect(Collectors.toList()))

.institute(education.getInstitute())

.specialization(education.getDegreeName())

.build();

}

}

这里使用了Builder模式创建DoctorPatientSummary对象。

在MapStruct生成映射器实现类之后，你就可以使用这个实现方法，就像访问任何其它映射器方法一样：

DoctorPatientSummary summary = doctorMapper.toDoctorPatientSummary(dotor, education);

**创建自定义映射器**

前面我们一直是通过接口来设计映射器功能，其实我们也可以通过一个带 @Mapper 的 abstract 类来实现一个映射器。MapStruct也会为这个类创建一个实现，类似于创建一个接口实现。

我们重写一下前面的示例，这一次，我们将它修改为一个抽象类：

@Mapper

public abstract class DoctorCustomMapper {

public DoctorPatientSummary toDoctorPatientSummary(Doctor doctor, Education education) {

return DoctorPatientSummary.builder()

.doctorId(doctor.getId())

.doctorName(doctor.getName())

.patientCount(doctor.getPatientList().size())

.patientIds(doctor.getPatientList()

.stream()

.map(Patient::getId)

.collect(Collectors.toList()))

.institute(education.getInstitute())

.specialization(education.getDegreeName())

.build();

}

}

你可以用同样的方式使用这个映射器。由于限制较少，使用抽象类可以在创建自定义实现时给我们更多的控制和选择。另一个好处是可以添加@BeforeMapping和@AfterMapping方法。

**@BeforeMapping 和 @AfterMapping**

为了进一步控制和定制化，我们可以定义 @BeforeMapping 和 @AfterMapping方法。显然，这两个方法是在每次映射之前和之后执行的。也就是说，在最终的实现代码中，会在两个对象真正映射之前和之后添加并执行这两个方法。

可以在 DoctorCustomMapper中添加两个方法：

@Mapper(uses = {PatientMapper.class}, componentModel = "spring")

public abstract class DoctorCustomMapper {

@BeforeMapping

protected void validate(Doctor doctor) {

if(doctor.getPatientList() == null){

doctor.setPatientList(new ArrayList<>());

}

}

@AfterMapping

protected void updateResult(@MappingTarget DoctorDto doctorDto) {

doctorDto.setName(doctorDto.getName().toUpperCase());

doctorDto.setDegree(doctorDto.getDegree().toUpperCase());

doctorDto.setSpecialization(doctorDto.getSpecialization().toUpperCase());

}

@Mapping(source = "doctor.patientList", target = "patientDtoList")

@Mapping(source = "doctor.specialty", target = "specialization")

public abstract DoctorDto toDoctorDto(Doctor doctor);

}

基于该抽象类生成一个映射器实现类：

@Component

public class DoctorCustomMapperImpl extends DoctorCustomMapper {

@Autowired

private PatientMapper patientMapper;

@Override

public DoctorDto toDoctorDto(Doctor doctor) {

validate(doctor);

if (doctor == null) { return null; }

DoctorDto doctorDto = new DoctorDto();

doctorDto.setPatientDtoList(patientListToPatientDtoList(doctor.getPatientList()));

doctorDto.setSpecialization(doctor.getSpecialty());

doctorDto.setId(doctor.getId());

doctorDto.setName(doctor.getName());

updateResult(doctorDto);

return doctorDto;

}

}

可以看到， validate() 方法会在 DoctorDto 对象实例化之前执行，而updateResult()方法会在映射结束之后执行。

**映射异常处理**

异常处理是不可避免的，应用程序随时会产生异常状态。MapStruct提供了对异常处理的支持，可以简化开发者的工作。

考虑这样一个场景，我们想在 Doctor 映射为DoctorDto之前校验一下 Doctor 的数据。我们新建一个独立的 Validator 类进行校验：

public class Validator {

public int validateId(int id) throws ValidationException {

if(id == -1){

throw new ValidationException("Invalid value in ID");

}

return id;

}

}

我们修改一下 DoctorMapper 以使用 Validator 类，无需指定实现。跟之前一样， 在@Mapper使用的类列表中添加该类。我们还需要做的就是告诉MapStruct我们的 toDto() 会抛出 throws ValidationException：

@Mapper(uses = {PatientMapper.class, Validator.class}, componentModel = "spring")

public interface DoctorMapper {

@Mapping(source = "doctor.patientList", target = "patientDtoList")

@Mapping(source = "doctor.specialty", target = "specialization")

DoctorDto toDto(Doctor doctor) throws ValidationException;

}

最终生成的映射器代码如下：

@Component

public class DoctorMapperImpl implements DoctorMapper {

@Autowired

private PatientMapper patientMapper;

@Autowired

private Validator validator;

@Override

public DoctorDto toDto(Doctor doctor) throws ValidationException {

if (doctor == null) { return null; }

DoctorDto doctorDto = new DoctorDto();

doctorDto.setPatientDtoList(patientListToPatientDtoList(doctor

.getPatientList()));

doctorDto.setSpecialization(doctor.getSpecialty());

doctorDto.setId(validator.validateId(doctor.getId()));

doctorDto.setName(doctor.getName());

doctorDto.setExternalId(doctor.getExternalId());

doctorDto.setAvailability(doctor.getAvailability());

return doctorDto;

}

}

MapStruct自动将doctorDto的id设置为Validator实例的方法返回值。它还在该方法签名中添加了一个throws子句。

注意，如果映射前后的一对属性的类型与Validator中的方法出入参类型一致，那该字段映射时就会调用Validator中的方法，所以该方式请谨慎使用。

**映射配置**

MapStruct为编写映射器方法提供了一些非常有用的配置。多数情况下，如果我们已经定义了两个类型之间的映射方法，当我们要添加相同类型之间的另一个映射方法时，我们往往会直接复制已有方法的映射配置。

其实我们不必手动复制这些注解，只需要简单的配置就可以创建一个相同/相似的映射方法。

**继承配置**

我们回顾一下“更新现有实例”，在该场景中，我们创建了一个映射器，根据DoctorDto对象的属性更新现有的Doctor对象的属性值：

@Mapper(uses = {PatientMapper.class})

public interface DoctorMapper {

DoctorMapper INSTANCE = Mappers.getMapper(DoctorMapper.class);

@Mapping(source = "doctorDto.patientDtoList", target = "patientList")

@Mapping(source = "doctorDto.specialization", target = "specialty")

void updateModel(DoctorDto doctorDto, @MappingTarget Doctor doctor);

}

假设我们还有另一个映射器，将 DoctorDto转换为 Doctor ：

@Mapper(uses = {PatientMapper.class, Validator.class})

public interface DoctorMapper {

@Mapping(source = "doctorDto.patientDtoList", target = "patientList")

@Mapping(source = "doctorDto.specialization", target = "specialty")

Doctor toModel(DoctorDto doctorDto);

}

这两个映射方法使用了相同的注解配置， source和 target都是相同的。其实我们可以使用@InheritConfiguration注释，从而避免这两个映射器方法的重复配置。

如果对一个方法添加 @InheritConfiguration 注解，MapStruct会检索其它的已配置方法，寻找可用于当前方法的注解配置。一般来说，这个注解都用于mapping方法后面的update方法，如下所示：

@Mapper(uses = {PatientMapper.class, Validator.class}, componentModel = "spring")

public interface DoctorMapper {

@Mapping(source = "doctorDto.specialization", target = "specialty")

@Mapping(source = "doctorDto.patientDtoList", target = "patientList")

Doctor toModel(DoctorDto doctorDto);

@InheritConfiguration

void updateModel(DoctorDto doctorDto, @MappingTarget Doctor doctor);

}

**继承逆向配置**

还有另外一个类似的场景，就是编写映射函数将\*\*\*Model\*\*\* 转为 DTO，以及将 DTO 转为 Model。如下面的代码所示，我们必须在两个函数上添加相同的注释。

@Mapper(componentModel = "spring")

public interface PatientMapper {

@Mapping(source = "dateOfBirth", target = "dateOfBirth", dateFormat = "dd/MMM/yyyy")

Patient toModel(PatientDto patientDto);

@Mapping(source = "dateOfBirth", target = "dateOfBirth", dateFormat = "dd/MMM/yyyy")

PatientDto toDto(Patient patient);

}

两个方法的配置不会是完全相同的，实际上，它们应该是相反的。将\*\*\*Model\*\*\* 转为 DTO，以及将 DTO 转为 Model——映射前后的字段相同，但是源属性字段与目标属性字段是相反的。

我们可以在第二个方法上使用@InheritInverseConfiguration注解，避免写两遍映射配置：

@Mapper(componentModel = "spring")

public interface PatientMapper {

@Mapping(source = "dateOfBirth", target = "dateOfBirth", dateFormat = "dd/MMM/yyyy")

Patient toModel(PatientDto patientDto);

@InheritInverseConfiguration

PatientDto toDto(Patient patient);

}

这两个Mapper生成的代码是相同的。

**总结**

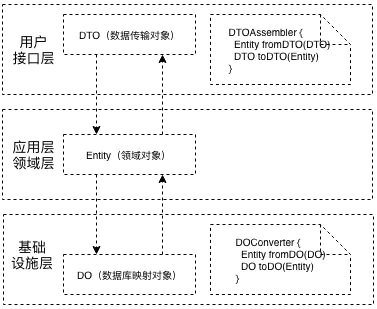
在本文中，我们探讨了MapStruct——一个用于创建映射器类的库。从基本映射到自定义方法和自定义映射器，此外， 我们还介绍了MapStruct提供的一些高级操作选项，包括依赖注入，数据类型映射、枚举映射和表达式使用。

MapStruct提供了一个功能强大的集成插件，可减少开发人员编写模板代码的工作量，使创建映射器的过程变得简单快捷。

如果要探索更多、更详细的使用方式，可以参考MapStruct官方提供的参考指南。

# 背景

在代码开发中，我们通常都会使用分层架构，在分层架构中都会使用模型转换，在不同的层使用不同的模型。以 DDD 分层模型为例，如下：



**模型分类**

### DO

DataObject，数据库映射对象，通常用于基础设施层，与数据库字段完全对应。

### Entity

领域对象，通常用于应用层和领域层（有一些 DDD 代码模型在应用层使用的是 DTO，但是基于应用层是业务编排的职责，可能会直接使用 Entity 的行为进行逻辑编排，那么个人建议应用层应该使用 Entity）。不只是指实体、还包括值对象。通常是充血模型，包括属性和行为。

### DTO

数据传输对象，通常用于用户接口层（用户接口层，通常指的是流量入口，包括web 流量、服务消费者 RPC 调用、消息输入等）。所以 DTO 通常用于Controller中的输入输出参数、打到二方包里的输入输出参数（例如，Dubbo 接口的输入输出参数）以及消息消费者中的消息模型。  
根据实际需要，有时候在 web 中，我们也会使用 vo。

## 转换器

### DTOAssembler

DTO 和 Entity 的转换器

### DOConverter

DO 和 Entity 的转换器