(/)

Asignación de listas con ModelMapper

Última modificación: 11 de mayo de 2020

por baeldung (https://www.baeldung.com/author/baeldung/)
(https://www.baeldung.com/author/baeldung/)

Datos (https://www.baeldung.com/category/data/)

Colecciones Java (https://www.baeldung.com/category/java/java-collections/)

1. Información general

En este tutorial, explicaremos cómo asignar listas de diferentes tipos de elementos utilizando el marco ModelMapper (http://modelmapper.org/getting-started/) . **Esto implica el uso de tipos genéricos en Java como una solución para convertir diferentes tipos de datos de una lista a otra** .

2. Mapeador de modelos

La función principal de ModelMapper es asignar objetos determinando cómo se asigna un modelo de objeto a otro llamado Objeto de transformación de datos (DTO).

Para usar ModelMapper (https://search.maven.org/artifact/org.modelmapper/modelmapper) , comenzamos agregando la dependencia a nuestro *pom.xml* :

```
<dependency>
    <groupId>org.modelmapper</groupId>
    <artifactId>modelmapper</artifactId>
    <version>2.3.7</version>
</dependency>
```


2.1. Configuración

ModelMapper proporciona una variedad de configuraciones para simplificar el proceso de mapeo. Personalizamos la configuración habilitando o deshabilitando las propiedades apropiadas en la configuración . **Es una práctica común establecer la propiedad *fieldMatchingEnabled* en *true* y permitir la coincidencia de campos privados** :

```
modelMapper.getConfiguration()
  .setFieldMatchingEnabled(true)
  .setFieldAccessLevel(Configuration.AccessLevel.PRIVATE);
```

Al hacerlo, ModelMapper puede comparar campos privados en las clases de asignación (objetos). En esta configuración, no es estrictamente necesario que todos los campos con los mismos nombres existan en ambas clases. Se permiten varias estrategias de coincidencia (http://modelmapper.org/user-manual/configuration/) . **De forma predeterminada, una estrategia de coincidencia estándar requiere que todas las propiedades de origen y destino deben coincidir en cualquier orden. Esto es ideal para nuestro escenario** .

2.2. Token de tipo

ModelMapper usa TypeToken (http://modelmapper.org/javadoc/org/modelmapper/TypeToken.html) para asignar tipos genéricos. Para ver por qué esto es necesario, veamos qué sucede cuando *asignamos* una lista de *enteros* a una lista de *caracteres* :

```
List<Integer> integers = new ArrayList<Integer>();
integers.add(1);
integers.add(2);
integers.add(3);

List<Character> characters = new ArrayList<Character>();
modelMapper.map(integers, characters);
```

Además, si imprimimos los elementos de la lista de *caracteres* , veríamos una lista vacía. Esto se debe a la ocurrencia de borrado de tipo durante la ejecución del tiempo de ejecución.

Sin embargo, si cambiamos nuestra llamada al *mapa* para usar *TypeToken* , podemos crear un tipo literal para *List <Character>* :

```
List<Character> characters
    = modelMapper.map(integers, new TypeToken<List<Character>>() {}.getType());
```

En tiempo de compilación, el caso interno anónimo *TokenType* conserva el tipo de parámetro ***List <Character>*** , y esta vez nuestra conversión es exitosa.

3. Uso de mapeo de tipo personalizado

Las listas en Java pueden asignarse utilizando tipos de elementos personalizados.

Por ejemplo, supongamos que queremos asignar una lista de entidades de *Usuario* a una lista de *UserDTO* . **Para lograr esto, llamaremos a *map* para cada elemento:**

```
List<UserDTO> dtos = users
  .stream()
  .map(user -> modelMapper.map(user, UserDTO.class))
  .collect(Collectors.toList());
```

Por supuesto, con un poco más de trabajo, podríamos hacer un método parametrizado de propósito general:

```
<S, T> List<T> mapList(List<S> source, Class<T> targetClass) {
    return source
      .stream()
      .map(element -> modelMapper.map(element, targetClass))
      .collect(Collectors.toList());
}
```

Entonces, podríamos hacer :

```
List<UserDTO> userDtoList = mapList(users, UserDTO.class);
```

4. Mapa de tipo y mapeo de propiedades

Se pueden agregar propiedades específicas como listas o conjuntos al *TypeMap* *(http://modelmapper.org/javadoc/org/modelmapper/TypeMap.html)* proporciona un método para definir explícitamente la asignación de estas propiedades. El objeto *TypeMap* almacena información de mapeo de

tipos específicos (clases):

```
TypeMap<UserList, UserListDTO> typeMap = modelMapper.createTypeMap(UserList.class, UserListDTO.class);
```

La clase *UserList* contiene una colección de *usuarios* . **Aquí, queremos asignar la lista de nombres de usuario de esta colección a la lista de propiedades de la clase *UserListDTO*** . Para lograr esto, crearemos la primera clase *UsersListConverter* y le pasaremos *List <User>* y *List <String>* como tipos de parámetros para la conversión:

```
public class UsersListConverter extends AbstractConverter<List<User>, List<String>> {

    @Override
    protected List<String> convert(List<User> users) {

        return users
          .stream()
          .map(User::getUsername)
          .collect(Collectors.toList());
    }
}
```

Desde el objeto *TypeMap* creado, *agregamos* explícitamente la asignación de propiedades (http://modelmapper.org/user-manual/property-mapping/) invocando una instancia de la clase *UsersListConverter* :

```
typeMap.addMappings(mapper -> mapper.using(new UsersListConverter())
  .map(UserList::getUsers, UserListDTO::setUsernames));
```

Dentro del método *addMappings* , un mapeo de expresiones nos permite definir las propiedades de origen a destino con expresiones lambda. Finalmente, convierte la lista de usuarios en la lista resultante de nombres de usuario.

5. Conclusión

En este tutorial, explicamos cómo se mapean las listas manipulando tipos genéricos en ModelMapper . Podemos utilizar *TypeToken, la* asignación de tipos genéricos y la asignación de propiedades para crear tipos de listas de objetos y realizar asignaciones complejas.

El código fuente completo de este artículo está disponible en GitHub (https://github.com/eugenp/tutorials/tree/master/java-collections-conversions-2) .

Deja una respuesta

Comience la discusión ...

CATEGORIAS

PRIMAVERA (HTTPS://WWW.BAELDUNG.COM/CATEGORY/SPRING/)
DESCANSO (HTTPS://WWW.BAELDUNG.COM/CATEGORY/REST/)
JAVA (HTTPS://WWW.BAELDUNG.COM/CATEGORY/JAVA/)
SEGURIDAD (HTTPS://WWW.BAELDUNG.COM/CATEGORY/SECURITY-2/)
PERSISTENCIA (HTTPS://WWW.BAELDUNG.COM/CATEGORY/PERSISTENCE/)
JACKSON (HTTPS://WWW.BAELDUNG.COM/CATEGORY/JSON/JACKSON/)
HTTP DEL LADO DEL CLIENTE (HTTPS://WWW.BAELDUNG.COM/CATEGORY/HTTP/)
KOTLIN (HTTPS://WWW.BAELDUNG.COM/CATEGORY/KOTLIN/)

SERIE

TUTORIAL DE JAVA "VOLVER A LO BÁSICO" (/JAVA-TUTORIAL)
JACKSON JSON TUTORIAL (/JACKSON)
HTTPCLIENT 4 TUTORIAL (/HTTPCLIENT-GUIDE)
RESTO CON SPRING TUTORIAL (/REST-WITH-SPRING-SERIES)
TUTORIAL SPRING PERSISTENCE (/PERSISTENCE-WITH-SPRING-SERIES)
SEGURIDAD CON PRIMAVERA (/SECURITY-SPRING)

ACERCA DE

SOBRE BAELDUNG (/ABOUT)
LOS CURSOS (HTTPS://COURSES.BAELDUNG.COM)
TRABAJOS (/TAG/ACTIVE-JOB/)
EL ARCHIVO COMPLETO (/FULL_ARCHIVE)
ESCRIBIR PARA BAELDUNG (/CONTRIBUTION-GUIDELINES)
EDITORES (/EDITORS)
NUESTROS COMPAÑEROS (/PARTNERS)
ANUNCIE EN BAELDUNG (/ADVERTISE)

TÉRMINOS DE SERVICIO (/TERMS-OF-SERVICE)
POLÍTICA DE PRIVACIDAD (/PRIVACY-POLICY)
INFORMACIÓN DE LA COMPAÑÍA (/BAELDUNG-COMPANY-INFO)
CONTACTO (/CONTACT)