Hibernate Validator

JSR 303 的参考实现

使用指南

5.0.0.CR2

由 Hardy Ferentschik和Gunnar Morling

and thanks to Shaozhuang Liu

序言 v

1. 开始入门 1
  - 1.1. Project set up 1
  - 1.2. 添加约束 2
  - 1.3. 校验约束 3
  - 1.4. 更进一步 4

2. Validation step by step 5
  - 2.1. 定义约束 5
    - 2.1.1. 字段级(field level) 约束 5
    - 2.1.2. 属性级别约束 6
    - 2.1.3. 类级别约束 7
    - 2.1.4. 约束继承 8
    - 2.1.5. 对象图 9
  - 2.2. 校验约束 11
    - 2.2.1. 获取一个Validator的实例 11
    - 2.2.2. Validator中的方法 11
    - 2.2.3. ConstraintViolation 中的方法 13
    - 2.2.4. 验证失败提示信息解析 13
  - 2.3. 校验组 14
    - 2.3.1. 校验组序列 17
    - 2.3.2. 对一个类重定义其默认校验组 18
  - 2.4. 内置的约束条件 21
    - 2.4.1. Bean Validation constraints 21
    - 2.4.2. Additional constraints 24

3. 创建自己的约束规则 31
  - 3.1. 创建一个简单的约束条件 31
    - 3.1.1. 约束标注 31
    - 3.1.2. 约束校验器 33
    - 3.1.3. 校验错误信息 36
    - 3.1.4. 应用约束条件 36
  - 3.2. 约束条件组合 38

4. XML configuration 41
  - 4.1. validation.xml 41
  - 4.2. 映射约束 42

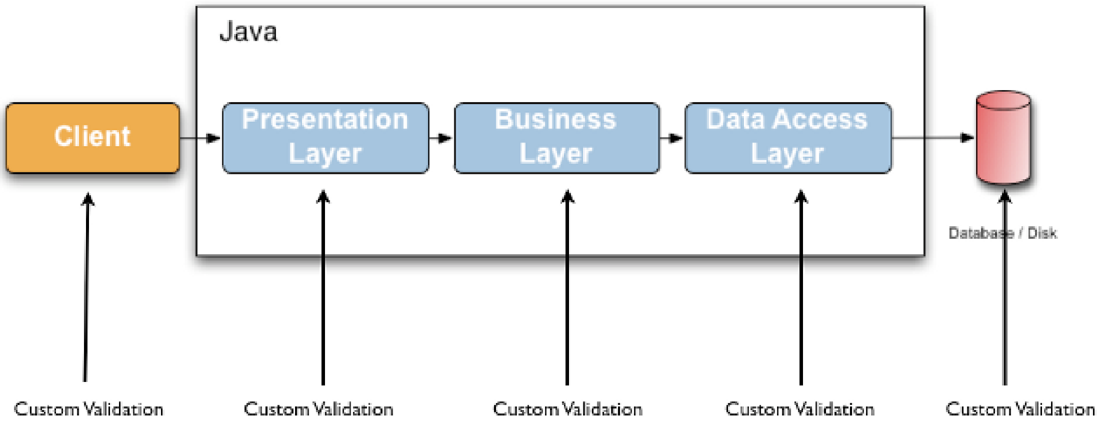
5. Bootstrapping 47
  - 5.1. Configuration 和 ValidatorFactory 47
  - 5.2. ValidationProviderResolver 48
  - 5.3. MessageInterpolator 49
    - 5.3.1. ResourceBundleLocator 49
  - 5.4. TraversableResolver 50
  - 5.5. ConstraintValidatorFactory 51

6. Metadata API 53
  - 6.1. BeanDescriptor 53
  - 6.2. PropertyDescriptor 53

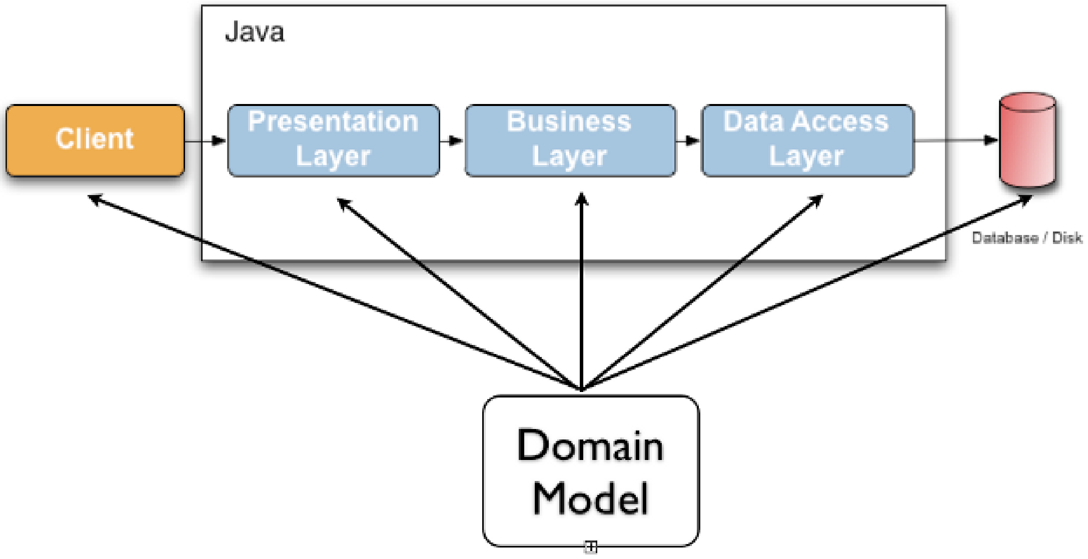
  - 6.3. ElementDescriptor 53
  - 6.4. ConstraintDescriptor 54
- 7. 与其他框架集成 55
  - 7.1. OSGi 55
  - 7.2. 与数据库集成校验 56
  - 7.3. ORM集成 56
    - 7.3.1. 基于Hibernate事件模型的校验 56
    - 7.3.2. JPA 57
  - 7.4. 展示层校验 58
- 8. Hibernate Validator Specifics 59
  - 8.1. Public API 59
  - 8.2. Fail fast mode 61
  - 8.3. Method validation 62
    - 8.3.1. Defining method-level constraints 63
    - 8.3.2. Evaluating method-level constraints 65
    - 8.3.3. Retrieving method-level constraint meta data 66
  - 8.4. Programmatic constraint definition 67
  - 8.5. Boolean composition for constraint composition 70
- 9. Annotation Processor 71
  - 9.1. 前提条件 71
  - 9.2. 特性 71
  - 9.3. 配置项 72
  - 9.4. 使用标注处理器 72
    - 9.4.1. 命令行编译 72
    - 9.4.2. IDE集成 74
  - 9.5. 已知问题 77
- 10. 进一步阅读 79

序言

数据校验是任何一个应用程序都会用到的功能,无论是显示层还是持久层. 通常,相同的校验逻辑会分散在各个层中, 这样,不仅浪费了时间还会导致错误的发生(译注: 重复代码). 为了避免重复, 开发人员经常会把这些校验逻辑直接写在领域模型里面, 但是这样又把领域模型代码和校验代码混杂在了一起, 而这些校验逻辑更应该是描述领域模型的元数据.


Java
Client
Presentation Layer
Business Layer
Data Access Layer
Database / Disk
Custom Validation
Custom Validation
Custom Validation
Custom Validation
Custom Validation

JSR 303 - Bean Validation - 为实体验证定义了元数据模型和API. 默认的元数据模型是通过Annotations来描述的,但是也可以使用XML来重载或者扩展. Bean Validation API 并不局限于应用程序的某一层或者哪种编程模型, 例如,如图所示, Bean Validation 可以被用在任何一层, 或者是像类似Swing的富客户端程序中.


Java
Client
Presentation Layer
Business Layer
Data Access Layer
Database / Disk
Domain Model

Hibernate Validator is the reference implementation of this JSR. The implementation itself as well as the Bean Validation API and TCK are all provided and distributed under the Apache Software License 2.0 [http://www.apache.org/licenses/LICENSE-2.0].

开始入门

本章将会告诉你如何使用Hibernate Validator，在开始之前,你需要准备好下面的环境:

- A JDK >= 6
- Apache Maven [http://maven.apache.org/]
- 网络连接（ Maven需要通过互联网下载所需的类库）
- A properly configured remote repository. Add the following to your settings.xml:

例 1.1. Configuring the JBoss Maven repository

```
<repositories>
    <repository>
        <id>jboss-public-repository-group</id>
        <url>https://repository.jboss.org/nexus/content/groups/public-jboss</url>
        <releases>
          <enabled>true</enabled>
        </releases>
        <snapshots>
          <enabled>true</enabled>
        </snapshots>
     </repository>
</repositories>
```

More information about settings.xml can be found in the Maven Local Settings Model [http://maven.apache.org/ref/2.0.8/maven-settings/settings.html].

1.1. Project set up

In order to use Hibernate Validator within an existing Maven project, simply add the following dependency to your pom.xml:

例 1.2. Hibernate Validator Maven dependency

```
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-validator</artifactId>
    <version>5.0.0.CR2</version>
</dependency>
```

提示

For the purposes of logging, Hibernate Validator uses the JBoss Logging API. This is an abstraction layer which supports several known logging solutions (e.g. log4j or the logging framework provided by the JDK) as implementation. Just add your preferred logging library to the classpath and all log requests from Hibernate Validator will automatically be delegated to that logging provider.

Alternatively, you can explicitely specify a provider using the system property `org.jboss.logging.provider`. Supported values currently are `jboss`, `jdk`, `log4j` and `slf4j`.

1.2. 添加约束

Lets dive directly into an example to see how to apply constraints.

例 1.3. 带约束性标注(annotated with constraints)的Car 类

```java
package com.mycompany;

import javax.validation.constraints.Min;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;

public class Car {

    @NotNull
    private String manufacturer;

    @NotNull
    @Size(min = 2, max = 14)
    private String licensePlate;

    @Min(2)
    private int seatCount;

    public Car(String manufacturer, String licencePlate, int seatCount) {
        this.manufacturer = manufacturer;
        this.licensePlate = licencePlate;
        this.seatCount = seatCount;
    }

    //getters and setters ...
}
```

`@NotNull`, `@Size` and `@Min`就是上面所属的约束性标注(constraint annotations), 我们就是使用它们来声明约束, 例如在`Car`的字段中我们可以看到:

- manufacturer永远不能为null
- licensePlate永远不能为null,并且它的值字符串的长度要在2到14之间
- seatCount的值要不能小于2

1.3. 校验约束

To perform a validation of these constraints, we use a `Validator` instance. Let's have a look at a unit test for `Car`:

例 1.4. 在CarTest中使用校验

```
package com.mycompany;

import static org.junit.Assert.*;

import java.util.Set;

import javax.validation.ConstraintViolation;
import javax.validation.Validation;
import javax.validation.Validator;
import javax.validation.ValidatorFactory;

import org.junit.BeforeClass;
import org.junit.Test;

public class CarTest {

    private static Validator validator;

    @BeforeClass
    public static void setUp() {
        ValidatorFactory factory = Validation.buildDefaultValidatorFactory();
        validator = factory.getValidator();
    }

    @Test
    public void manufacturerIsNull() {
        Car car = new Car(null, "DD-AB-123", 4);

        Set<ConstraintViolation<Car>> constraintViolations =
            validator.validate(car);

        assertEquals(1, constraintViolations.size());
        assertEquals("may not be null", constraintViolations.iterator().next().getMessage());
    }

    @Test
    public void licensePlateTooShort() {
        Car car = new Car("Morris", "D", 4);

        Set<ConstraintViolation<Car>> constraintViolations =
            validator.validate(car);
        assertEquals(1, constraintViolations.size());
                                    assertEquals("size must be between 2 and
 14", constraintViolations.iterator().next().getMessage());
    }

    @Test
    public void seatCountTooLow() {
        Car car = new Car("Morris", "DD-AB-123", 1);

        Set<ConstraintViolation<Car>> constraintViolations =
            validator.validate(car);

        assertEquals(1, constraintViolations.size());
                               assertEquals("must be greater than or equal to
 2", constraintViolations.iterator().next().getMessage());
    }

    @Test
    public void carIsValid() {
        Car car = new Car("Morris", "DD-AB-123", 2);

        Set<ConstraintViolation<Car>> constraintViolations =
            validator.validate(car);

        assertEquals(0, constraintViolations.size());
    }
}
```

在setUp()方法中,我们通过ValidatorFactory得到了一个Validator的实例. Validator是线程安全的, 并且可以重复使用, 所以我们把它保存成一个类变量. 现在我们可以在test方法中使用这个validator的实例来校验不同的car实例了.

validate()方法会返回一个set的ConstraintViolation的实例的集合, 我们可以通过遍历它来查看有哪些验证错误. 前面三个测试用例显示了一些预期的校验约束:

- 在manufacturerIsNull()中可以看到manufacturer违反了@NotNull约束
- licensePlateTooShort()中的licensePlate违反了@Size约束
- 而seatCountTooLow()中则导致seatCount违反了@Min约束

如果一个对象没有校验出问题的话,那么validate() 会返回一个空的set对象.

注意,我们只使用了Bean Validation API中的package javax.validation中的类, 并没有直接调用参考实现中的任何类,所以, 没有任何问题如果切换到其他的实现.

1.4. 更进一步

That concludes our 5 minute tour through the world of Hibernate Validator. Continue exploring the code examples or look at further examples referenced in 第 10 章 进一步阅读. To deepen your understanding of Hibernate Validator just continue reading 第 2 章 Validation step by step. In case your application has specific validation requirements have a look at 第 3 章 创建自己的约束规则.

Validation step by step

在本章中,我们会详细的介绍如何使用Hibernate Validator 来对一个给定的实体模型进行验证. 还会介绍Bean Validation规范提供了哪些默认的约束条件和Hibernate Validator提供了哪些额外的. 让我们先从如何给一个实体添加约束开始.

2.1. 定义约束

Bean Validation 的约束是通过Java 注解(annotations)来标注的. 在本节中,我们会介绍如何使用这些注解(annotations)来标注一个实体模型. 并且,我们会区分三种不通的注解(annotations)类型.

> **注意**
>
> 不是所有的约束都能够被用在所有的类结构上. 事实上, 没有任何定义在Bean Validation规范中的约束可以被用在class上. 约束定义中的java.lang.annotation.Target属性定义了这个约束能够被使用在哪个层次结构上. 详细信息请参考第 3 章 创建自己的约束规则.

2.1.1. 字段级(field level) 约束

约束条件能够被标注在类的字段上面，请参考示例例 2.1 “字段级(field level) 约束”

例 2.1. 字段级(field level) 约束

```java
package com.mycompany;

import javax.validation.constraints.NotNull;

public class Car {

    @NotNull
    private String manufacturer;

    @AssertTrue
    private boolean isRegistered;

    public Car(String manufacturer, boolean isRegistered) {
        super();
        this.manufacturer = manufacturer;
        this.isRegistered = isRegistered;
    }
}
```

当约束被定义在字段上的时候， 这个字段的值是通过字段访问策略来获取并验证的． 也就是说Bean Validation的实现者会直接访问这个实例变量而不会调用属性的访问器(getter) 即使这个方法存在．

注意

这个字段的访问级别(private, protected 或者 public) 对此没有影响．

注意

静态字段或者属性是不会被校验的．

提示

When validating byte code enhanced objects property level constraints should be used, because the byte code enhancing library won't be able to determine a field access via reflection.

2.1.2. 属性级别约束

如果你的模型遵循JavaBeans [http://java.sun.com/javase/technologies/desktop/javabeans/index.jsp]规范的话，你还可以把约束标注在属性上． 例 2.2 “属性级约束”和例 2.1 “字段级(field level) 约束”的唯一不同就是它的约束是定义在属性级别上的．

注意

如果要定义约束在属性级别上的话,那么只能定义在访问器(getter)上面,不能定义在修改器(setter)上．

例 2.2. 属性级约束

```
package com.mycompany;

import javax.validation.constraints.AssertTrue;
import javax.validation.constraints.NotNull;

public class Car {

    private String manufacturer;

    private boolean isRegistered;
    public Car(String manufacturer, boolean isRegistered) {
        super();
        this.manufacturer = manufacturer;
        this.isRegistered = isRegistered;
    }

    @NotNull
    public String getManufacturer() {
        return manufacturer;
    }

    public void setManufacturer(String manufacturer) {
        this.manufacturer = manufacturer;
    }

    @AssertTrue
    public boolean isRegistered() {
        return isRegistered;
    }

    public void setRegistered(boolean isRegistered) {
        this.isRegistered = isRegistered;
    }
}
```

When using property level constraints property access strategy is used to access the value to be validated. This means the bean validation provider accesses the state via the property accessor method. One advantage of annotating properties instead of fields is that the constraints become part of the constrained type's API that way and users are aware of the existing constraints without having to examine the type's implementation.

提示

It is recommended to stick either to field or property annotations within one class. It is not recommended to annotate a field and the accompanying getter method as this would cause the field to be validated twice.

2.1.3. 类级别约束

Last but not least, a constraint can also be placed on class level. When a constraint annotation is placed on this level the class instance itself passed to the ConstraintValidator. Class level constraints are useful if it is necessary to inspect more than a single property of the class to validate it or if a correlation between different state variables has to be evaluated. In 例 2.3 “类级别约束” we add the property passengers to the class Car. We also add the constraint PassengerCount on the class level. We will later see how we can actually create this custom constraint (see 第 3 章 创建自己的约束规则). For now it is enough to know that PassengerCount will ensure that there cannot be more passengers in a car than there are seats.

例 2.3. 类级别约束

```
package com.mycompany;

import javax.validation.constraints.Min;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;

@PassengerCount
public class Car {

    @NotNull
    private String manufacturer;

    @NotNull
    @Size(min = 2, max = 14)
    private String licensePlate;

    @Min(2)
    private int seatCount;

    private List<Person> passengers;

    public Car(String manufacturer, String licencePlate, int seatCount) {
        this.manufacturer = manufacturer;
        this.licensePlate = licencePlate;
        this.seatCount = seatCount;
    }

    //getters and setters ...
}
```

2.1.4. 约束继承

如果要验证的对象继承于某个父类或者实现了某个接口,那么定义在父类或者接口中的约束会在验证这个对象的时候被自动加载,如同这些约束定义在这个对象所在的类中一样. 让我们来看看下面的示例:

例 2.4. 约束继承

```
package com.mycompany;

import javax.validation.constraints.NotNull;

public class RentalCar extends Car {

    private String rentalStation;

    public RentalCar(String manufacturer, String rentalStation) {
        super(manufacturer);
        this.rentalStation = rentalStation;
    }
    @NotNull
    public String getRentalStation() {
        return rentalStation;
    }

    public void setRentalStation(String rentalStation) {
        this.rentalStation = rentalStation;
    }
}
```

我们有了一个新的RentalCar类继承自前面我们已经见到的Car， 这个子类中增加了一个rentalStation属性. 如果校验一个RentalCar的实例对象, 那么不仅会验证属性rentalStation上的 @NotNull约束是否合法,还会校验父类中的manufacturer属性.

如果类Car是一个接口类型的话也是一样的效果.

如果类RentalCar 重写了父类Car的getManufacturer()方法, 那么定义在父类的这个方法上的约束和子类这个方法上定义的约束都会被校验.

2.1.5. 对象图

Bean Validation API不仅能够用来校验单个的实例对象,还能够用来校验完整的对象图.要使用这个功能,只需要在一个有关联关系的字段或者属性上标注@Valid. 这样,如果一个对象被校验,那么它的所有的标注了@Valid的关联对象都会被校验. 请看例 2.6 “Adding a driver to the car”.

例 2.5. Class Person

```
package com.mycompany;

import javax.validation.constraints.NotNull;

public class Person {

    @NotNull
    private String name;

    public Person(String name) {
        super();
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

例 2.6. Adding a driver to the car

```
package com.mycompany;

import javax.validation.Valid;
import javax.validation.constraints.NotNull;

public class Car {

    @NotNull
    @Valid
    private Person driver;

    public Car(Person driver) {
        this.driver = driver;
    }

    //getters and setters ...
}
```

如果校验Car的实例对象的话,因为它的driver属性标注了@Valid， 那么关联的Person也会被校验. 所以,如果对象Person的name属性如果是null的话,那么校验会失败.

关联校验也适用于集合类型的字段，也就是说,任何下列的类型:

- 数组
- 实现了java.lang.Iterable接口(例如Collection, List 和 Set)
- 实现了java.util.Map接口

如果标注了@Valid，那么当主对象被校验的时候,这些集合对象中的元素都会被校验.

例 2.7. Car with a list of passengers

```
package com.mycompany;

import java.util.ArrayList;
import java.util.List;

import javax.validation.Valid;
import javax.validation.constraints.NotNull;

public class Car {

    @NotNull
    @Valid
    private List<Person> passengers = new ArrayList<Person>();

    public Car(List<Person> passengers) {
        this.passengers = passengers;
    }
    //getters and setters ...
}
```

当校验一个Car的实例的时候,如果passengers list中包含的任何一个Person对象没有名字的话,都会导致校验失败(a ConstraintValidation will be created).

> 注意
>
> 对象图校验的时候是会被忽略null值的.

2.2. 校验约束

Validator 是Bean Validation中最主要的接口, 我们会在第 5.1 节 “Configuration 和 ValidatorFactory” 中详细介绍如何获取一个Validator的实例, 现在先让我们来看看如何使用Validator接口中的各个方法.

2.2.1. 获取一个Validator的实例

对一个实体对象验证之前首先需要有个Validator对象, 而这个对象是需要通过Validation 类和ValidatorFactory来创建的. 最简单的方法是调用Validation.buildDefaultValidatorFactory() 这个静态方法.

例 2.8. Validation.buildDefaultValidatorFactory()

```
ValidatorFactory factory = Validation.buildDefaultValidatorFactory();
Validator validator = factory.getValidator();
```

第 5 章 Bootstrapping介绍了其他的获取Validator实例的方法. 现在我们的目标是学习如何使用Validator 来校验实体对象.

2.2.2. Validator中的方法

Validator中有三个方法能够被用来校验整个实体对象或者实体对象中的属性.

这三个方法都会返回一个Set<ConstraintViolation>对象, 如果整个验证过程没有发现问题的话,那么这个set是空的, 否则, 每个违反约束的地方都会被包装成一个ConstraintViolation的实例然后添加到set当中.

All the validation methods have a var-args parameter which can be used to specify, which validation groups shall be considered when performing the validation. If the parameter is not specified the default validation group (javax.validation.groups.Default) will be used. We will go into more detail on the topic of validation groups in 第 2.3 节 “校验组”

2.2.2.1. validate

使用validate()方法对一个给定的实体对象中定义的所有约束条件进行校验 （例 2.9 “Validator.validate() 使用方法” ）.

例 2.9. Validator.validate() 使用方法

```
ValidatorFactory factory = Validation.buildDefaultValidatorFactory();
Validator validator = factory.getValidator();

Car car = new Car(null);

Set<ConstraintViolation<Car>> constraintViolations = validator.validate(car);

assertEquals(1, constraintViolations.size());
assertEquals("may not be null", constraintViolations.iterator().next().getMessage());
```

2.2.2.2. validateProperty

通过validateProperty()可以对一个给定实体对象的单个属性进行校验. 其中属性名称需要符合JavaBean规范中定义的属性名称.

例 2.10. Validator.validateProperty()使用方法

```
Validator validator = Validation.buildDefaultValidatorFactory().getValidator();

Car car = new Car(null);

Set<ConstraintViolation<Car>> constraintViolations = validator.validateProperty(car, "manufacturer");

assertEquals(1, constraintViolations.size());
assertEquals("may not be null", constraintViolations.iterator().next().getMessage());
```

例如，Validator.validateProperty可以被用在把Bean Validation集成进JSF 2中的时候使用（请参考 第 7.4 节 “展示层校验”）.

2.2.2.3. validateValue

通过validateValue() 方法,你能够校验如果把一个特定的值赋给一个类的某一个属性的话,是否会违反此类中定义的约束条件.

例 2.11. Validator.validateValue() 使用方法

```
Validator validator = Validation.buildDefaultValidatorFactory().getValidator();

Set<ConstraintViolation<Car>> constraintViolations = validator.validateValue(Car.class, "manufacturer", null);
```

```
assertEquals(1, constraintViolations.size());
assertEquals("may not be null", constraintViolations.iterator().next().getMessage());
```

> **注意**
>
> validateProperty() 和 validateValue() 会忽略被验证属性上定义的@Valid.

2.2.3. ConstraintViolation 中的方法

现在是时候看看究竟ConstraintViolation是什么了. ConstraintViolation中包含了很多方法能够帮你快速定位究竟是什么导致了校验失败.表 2.1 “ConstraintViolation 中的方法” 列出了这些方法:

表 2.1. ConstraintViolation 中的方法

方法名	作用	示例 (请参考例 2.9 “Validator.validate() 使用方法”)
getMessage()	获取(经过翻译的)校验错误信息	may not be null
getMessageTemplate()	获取错误信息模版	{javax.validation.constraints.NotNull.message
getRootBean()	获取被校验的根实体对象	car
getRootBeanClass()	获取被校验的根实体类.	Car.class
getLeafBean()	如果约束是添加在一个bean(实体对象)上的,那么则返回这个bean的实例, 如果是约束是定义在一个属性上的,则返回这个属性所属的bean的实例对象.	car
getPropertyPath()	从被验证的根对象到被验证的属性的路径.	
getInvalidValue()	倒是校验失败的值.	passengers
getConstraintDescriptor()	导致校验失败的约束定义.	

2.2.4. 验证失败提示信息解析

在第 3 章 创建自己的约束规则中,我们会看到,每个约束定义中都包含有一个用于提示验证结果的消息模版,并且在声明一个约束条件的时候,你可以通过这个约束中的message属性来重写默认的消息模版, 可以参考例 2.13 “Driver” . 如果在校验的时候,这个约束条件没有通过,那么你配置的MessageInterpolator会被用来当成解析器来解析这个约束中定义的消息模版, 从而得到最终的验证失败提示信息. 这个解析器会尝试解析模版中的占位符(大括号括起来的字符串). 其中, Hibernate Validator中默认的解析器 (MessageInterpolator) 会先在类路径下找名称

为ValidationMessages.properties的ResourceBundle， 然后将占位符和这个文件中定义的resource进行匹配,如果匹配不成功的话,那么它会继续匹配Hibernate Validator自带的位于/org/hibernate/validator/ValidationMessages.properties的ResourceBundle， 依次类推,递归的匹配所有的占位符.

因为大括号{ 在这里是特殊字符,所以,你可以通过使用反斜线来对其进行转义， 例如:

- \{ 被转义成 {
- \} 被转义成 }
- \\ 被转义成 \

如果默认的消息解析器不能够满足你的需求,那么你也可以在创建ValidatorFactory的时候， 将其替换为一个你自定义的MessageInterpolator， 具体请参考 第 5 章 Bootstrapping.

2.3. 校验组

校验组能够让你在验证的时候选择应用哪些约束条件. 这样在某些情况下(例如向导) 就可以对每一步进行校验的时候， 选取对应这步的那些约束条件进行验证了. 校验组是通过可变参数传递给validate， validateProperty 和 validateValue的. 让我们来看个例子， 这个实例扩展了上面的Car类,又为其添加了一个新的Driver. 首先， 类Person (例 2.12 “Person”) 的name属性上定义了一个@NotNull 的约束条件. 因为没有明确指定这个约束条件属于哪个组,所以它被归类到默认组 (javax.validation.groups.Default).

注意

如果某个约束条件属于多个组,那么各个组在校验时候的顺序是不可预知的. 如果一个约束条件没有被指明属于哪个组,那么它就会被归类到默认组(javax.validation.groups.Default).

例 2.12. Person

```
public class Person {
    @NotNull
    private String name;

    public Person(String name) {
        this.name = name;
    }
    // getters and setters ...
}
```

接下来， 我们让类Driver (例 2.13 “Driver”) 继承自类Person. 然后添加两个属性,分别是age 和 hasDrivingLicense. 对于一个司机来说， 要开车的话， 必须满足两个条件， 年满18周岁(@Min(18)) 和你的有驾照(@AssertTrue). 这两个约束条件分别定义在那两个属性上， 并且把他们

都归于DriverChecks组. 通过例 2.14 “Group interfaces”, 你可以看到, "DriverChecks组" 就是一个简单的标记接口. 使用接口(而不是字符串) 可以做到类型安全,并且接口比字符串更加对重构友好, 另外, 接口还意味着一个组可以继承别的组.

例 2.13. Driver

```
public class Driver extends Person {
    @Min(value = 18, message = "You have to be 18 to drive a car", groups = DriverChecks.class)
    public int age;

   @AssertTrue(message = "You first have to pass the driving test", groups = DriverChecks.class)
    public boolean hasDrivingLicense;

    public Driver(String name) {
        super( name );
    }

    public void passedDrivingTest(boolean b) {
        hasDrivingLicense = b;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }
}
```

例 2.14. Group interfaces

```
public interface DriverChecks {
}

public interface CarChecks {
}
```

最后, 我们给Car class (例 2.15 “Car”) 添加一个passedVehicleInspection的属性,来表示这个车是否通过了上路检查.

例 2.15. Car

```
public class Car {
    @NotNull
    private String manufacturer;

    @NotNull
    @Size(min = 2, max = 14)
    private String licensePlate;
    @Min(2)
    private int seatCount;

            @AssertTrue(message = "The car has to pass the vehicle inspection
 first", groups = CarChecks.class)
    private boolean passedVehicleInspection;

    @Valid
    private Driver driver;

    public Car(String manufacturer, String licencePlate, int seatCount) {
        this.manufacturer = manufacturer;
        this.licensePlate = licencePlate;
        this.seatCount = seatCount;
    }
}
```

现在，在我们的例子中有三个不同的校验组，Person.name，Car.manufacturer，Car.licensePlate和Car.seatCount都属于默认(Default)组，Driver.age和Driver.hasDrivingLicense从属于DriverChecks组，而Car.passedVehicleInspection在CarChecks组中．例 2.16 “Drive away”演示了如何让Validator.validate验证不同的组来得到不同的校验结果．

例 2.16. Drive away

```
public class GroupTest {

    private static Validator validator;

    @BeforeClass
    public static void setUp() {
        ValidatorFactory factory = Validation.buildDefaultValidatorFactory();
        validator = factory.getValidator();
    }

    @Test
    public void driveAway() {
        // create a car and check that everything is ok with it.
        Car car = new Car( "Morris", "DD-AB-123", 2 );
        Set<ConstraintViolation<Car>> constraintViolations = validator.validate( car );
        assertEquals( 0, constraintViolations.size() );

        // but has it passed the vehicle inspection?
        constraintViolations = validator.validate( car, CarChecks.class );
        assertEquals( 1, constraintViolations.size() );
                        assertEquals("The car has to pass the vehicle inspection
 first", constraintViolations.iterator().next().getMessage());

        // let's go to the vehicle inspection
        car.setPassedVehicleInspection( true );
        assertEquals( 0, validator.validate( car ).size() );

        // now let's add a driver. He is 18, but has not passed the driving test yet
        Driver john = new Driver( "John Doe" );
        john.setAge( 18 );
        car.setDriver( john );
        constraintViolations = validator.validate( car, DriverChecks.class );
        assertEquals( 1, constraintViolations.size() );
                                assertEquals(    "You    first    have    to    pass    the    driving
 test", constraintViolations.iterator().next().getMessage() );

        // ok, John passes the test
        john.passedDrivingTest( true );
        assertEquals( 0, validator.validate( car, DriverChecks.class ).size() );

        // just checking that everything is in order now
        assertEquals( 0, validator.validate( car, Default.class, CarChecks.class, DriverChecks.class ).size() );
    }
}
```

首先我们创建一辆汽车然后在没有明确指定使用哪个校验组的情况下校验它， 可以看到即使passedVehicleInspection的默认值是false也不会校验出错误来． 因为定义在这个属性上的约束条件并不属于默认的校验组， 接下来,我们来校验CarChecks这个组， 这样就会发现car违反了约束条件，必须让这个车先通过检测．接下来,我们给这个车增加一个司机，然后在基于DriverChecks来校验， 会发现因为这个司机因为还没有通过驾照考试， 所以又一次得到了校验错误， 如果我们设置passedDrivingTest属性为true之后，DriverChecks组的校验就通过了．

最后，让我们再来校验所有的组中定义的约束条件,可以看到所有的约束条件都通过了验证．

2.3.1. 校验组序列

By default, constraints are evaluated in no particular order, regardless of which groups they belong to. In some situations, however, it is useful to control the order constraints are evaluated. In our example from 第 2.3 节 “校验组” we could for example require that first all default car constraints are passing before we check the road worthiness of the car. Finally before we drive away we check the actual driver constraints. In order to implement such an order one would define a new interface and annotate it with @GroupSequence defining the order in which the groups have to be validated.

注意

If at least one constraint fails in a sequenced group none of the constraints of the following groups in the sequence get validated.

例 2.17. 标注了@GroupSequence的接口

```
@GroupSequence({Default.class, CarChecks.class, DriverChecks.class})
public interface OrderedChecks {
}
```

警告

一个校验组序列中包含的校验组和这个校验组序列不能造成直接或者间接的循环引用. 包括校验组继承. 如果造成了循环引用的话, 会导致GroupDefinitionException异常.

例 2.18 “校验组序列的用法”展示了校验组序列的用法.

例 2.18. 校验组序列的用法

```
@Test
public void testOrderedChecks() {
    Car car = new Car( "Morris", "DD-AB-123", 2 );
    car.setPassedVehicleInspection( true );

    Driver john = new Driver( "John Doe" );
    john.setAge( 18 );
    john.passedDrivingTest( true );
    car.setDriver( john );

    assertEquals( 0, validator.validate( car, OrderedChecks.class ).size() );
}
```

2.3.2. 对一个类重定义其默认校验组

2.3.2.1. @GroupSequence

The `@GroupSequence` annotation also fulfills a second purpose. It allows you to redefine what the `Default` group means for a given class. To redefine `Default` for a given class, add a `@GroupSequence` annotation to the class. The defined groups in the annotation express the sequence of groups that substitute `Default` for this class. 例 2.19 “RentalCar with @GroupSequence” introduces a new class `RentalCar` with a redefined default group. With this definition you can evaluate the constraints belonging to `RentalChecks`, `CarChecks` and `RentalCar` by just requesting the `Default` group as seen in 例 2.20 “RentalCar with redefined default group”.

例 2.19. RentalCar with @GroupSequence

```
@GroupSequence({ RentalChecks.class, CarChecks.class, RentalCar.class })
public class RentalCar extends Car {
    @AssertFalse(message = "The car is currently rented out", groups = RentalChecks.class)
    private boolean rented;

    public RentalCar(String manufacturer, String licencePlate, int seatCount) {
        super( manufacturer, licencePlate, seatCount );
    }
    public boolean isRented() {
        return rented;
    }

    public void setRented(boolean rented) {
        this.rented = rented;
    }
}
```

例 2.20. RentalCar with redefined default group

```
/**
 * Validating the default group leads to validation on the default group sequence of {@code RentalCar}.
 */
@Test
public void carIsRented() {
  RentalCar rentalCar = new RentalCar( "Morris", "DD-AB-123", 2 );
  rentalCar.setPassedVehicleInspection( true );
  rentalCar.setRented( true );

  Set<ConstraintViolation<RentalCar>> constraintViolations = validator.validate( rentalCar );

  assertEquals( 1, constraintViolations.size() );
  assertEquals(
    "Wrong message",
    "The car is currently rented out",
    constraintViolations.iterator().next().getMessage()
  );

  rentalCar.setRented( false );
  constraintViolations = validator.validate( rentalCar );

  assertEquals( 0, constraintViolations.size() );
}
```

注意

Due to the fact that there cannot be a cyclic dependency in the group and group sequence definitions one cannot just add `Default` to the sequence redefining `Default` for a class. Instead the class itself has to be added!

注意

The `Default` group sequence overriding is local to the class it is defined on and is not propagated to the associated objects. This means in particular that adding `DriverChecks` to the default group sequence of `RentalCar` would not have any effects. Only the group `Default` will be propagated to the driver association when validation a rental car instance.

2.3.2.2. @GroupSequenceProvider

The `@javax.validation.GroupSequence` annotation is a standardized Bean Validation annotation. As seen in the previous section it allows you to statically redefine the default group sequence for a class. Hibernate Validator also offers a custom, non standardized annotation - `org.hibernate.validator.group.GroupSequenceProvider` - which allows for dynamic redefinition of the default group sequence. Using the rental car scenario again, one could dynamically add the `CarChecks` as seen in 例 2.21 "RentalCar with @GroupSequenceProvider" and 例 "DefaultGroupSequenceProvider implementation".

例 2.21. RentalCar with @GroupSequenceProvider

```
@GroupSequenceProvider(RentalCarGroupSequenceProvider.class)
public class RentalCar extends Car {
    @AssertFalse(message = "The car is currently rented out", groups = RentalChecks.class)
    private boolean rented;

    public RentalCar(String manufacturer, String licencePlate, int seatCount) {
        super( manufacturer, licencePlate, seatCount );
    }

    public boolean isRented() {
        return rented;
    }

    public void setRented(boolean rented) {
        this.rented = rented;
    }
}
```

例 . DefaultGroupSequenceProvider implementation

```
public class RentalCarGroupSequenceProvider implements DefaultGroupSequenceProvider<RentalCar> {
    public List<Class<?>> getValidationGroups(RentalCar car) {
        List<Class<?>> defaultGroupSequence = new ArrayList<Class<?>>();
        defaultGroupSequence.add( RentalCar.class );

        if ( car != null && !car.isRented() ) {
            defaultGroupSequence.add( CarChecks.class );
        }

        return defaultGroupSequence;
    }
}
```

2.4. 内置的约束条件

Hibernate Validator comprises a basic set of commonly used constraints. These are foremost the constraints defined by the Bean Validation specification (see 表 2.2 “Bean Validation constraints”). Additionally, Hibernate Validator provides useful custom constraints (see 表 2.3 “Custom constraints” and 表 2.4 “Custom country specific constraints”).

2.4.1. Bean Validation constraints

表 2.2 “Bean Validation constraints” shows purpose and supported data types of all constraints specified in the Bean Validation API. All these constraints apply to the field/property level, there are no class-level constraints defined in the Bean Validation specification. If you are using the Hibernate object-relational mapper, some of the constraints are taken into account when creating the DDL for your model (see column "Hibernate metadata impact").

> **注意**
>
> Hibernate Validator allows some constraints to be applied to more data types than required by the Bean Validation specification (e.g. @Max can be applied to Strings). Relying on this feature can impact portability of your application between Bean Validation providers.

表 2.2. Bean Validation constraints

Annotation	Supported data types	作用	Hibernate metadata impact
@AssertFalse	Boolean, boolean	Checks that the annotated element is false.	没有
@AssertTrue	Boolean, boolean	Checks that the annotated element is true.	没有
@DecimalMax	BigDecimal, BigInteger, String, byte, short, int, long and the respective wrappers of the primitive types. Additionally supported by HV: any sub-type	被标注的值必须不大于约束中指定的最大值. 这个约束的参数是一个通过BigDecimal定义的最大值的字符串表示.	没有

Annotation	Supported data types	作用	Hibernate metadata impact
	of Number and CharSequence.		
@DecimalMin	BigDecimal, BigInteger, String, byte, short, int, long and the respective wrappers of the primitive types. Additionally supported by HV: any sub-type of Number and CharSequence.	被标注的值必须不小于约束中指定的最小值. 这个约束的参数是一个通过BigDecimal定义的最小值的字符串表示.	没有
@Digits(integer=, fraction=)	BigDecimal, BigInteger, String, byte, short, int, long and the respective wrappers of the primitive types. Additionally supported by HV: any sub-type of Number and CharSequence.	Checks whether the annoted value is a number having up to integer digits and fraction fractional digits.	对应的数据库表字段会被设置精度(precision)和准度(scale).
@Future	java.util.Date, java.util.Calendar; Additionally supported by HV, if the Joda Time [http://joda-time.sourceforge.net/] date/time API is on the class path: any implementations of ReadablePartial and ReadableInstant.	检查给定的日期是否比现在晚.	没有
@Max	BigDecimal, BigInteger, byte, short, int, long and the respective wrappers of the	检查该值是否小于或等于约束条件中指定的最大值.	会给对应的数据库表字段添加一个check的约束条件.

Annotation	Supported data types	作用	Hibernate metadata impact
	primitive types. Additionally supported by HV: any sub-type ofCharSequence (the numeric value represented by the character sequence is evaluated), any sub-type of Number.		
@Min	BigDecimal, BigInteger, byte, short, int, long and the respective wrappers of the primitive types. Additionally supported by HV: any sub-type of CharSequence (the numeric value represented by the char sequence is evaluated), any sub-type of Number.	检查该值是否大于或等于约束条件中规定的最小值.	会给对应的数据库表字段添加一个check的约束条件.
@NotNull	Any type	Checks that the annotated value is not null.	对应的表字段不允许为null.
@Null	Any type	Checks that the annotated value is null.	没有
@Past	java.util.Date, java.util.Calendar; Additionally supported by HV, if the Joda Time [http://joda-time.sourceforge.net/] date/time API is on the class path: any implementations	检查标注对象中的值表示的日期比当前早.	没有

Annotation	Supported data types	作用	Hibernate metadata impact
	of ReadablePartial and ReadableInstant.		
@Pattern(regex=, flag=)	String. Additionally supported by HV: any sub-type of CharSequence.	检查该字符串是否能够在match指定的情况下被regex定义的正则表达式匹配.	没有
@Size(min=, max=)	String, Collection, Map and arrays. Additionally supported by HV: any sub-type of CharSequence.	Checks if the annotated element's size is between min and max (inclusive).	对应的数据库表字段的长度会被设置成约束中定义的最大值.
@Valid	Any non-primitive type	递归的对关联对象进行校验, 如果关联对象是个集合或者数组, 那么对其中的元素进行递归校验,如果是一个map,则对其中的值部分进行校验.	没有

注意

On top of the parameters indicated in 表 2.2 "Bean Validation constraints" each constraint supports the parameters message, groups and payload. This is a requirement of the Bean Validation specification.

2.4.2. Additional constraints

In addition to the constraints defined by the Bean Validation API Hibernate Validator provides several useful custom constraints which are listed in 表 2.3 "Custom constraints" . With one exception also these constraints apply to the field/property level, only @ScriptAssert is a class-level constraint.

表 2.3. Custom constraints

Annotation	Supported data types	作用	Hibernate metadata impact
@CreditCardNumber	CharSequence	Checks that the annotated character sequence passes the Luhn checksum	没有

Annotation	Supported data types	作用	Hibernate metadata impact
		test. Note, this validation aims to check for user mistakes, not credit card validity! See also Anatomy of Credit Card Numbers [http://www.merriampark.com/anatomycc.htm].	
@Email	CharSequence	Checks whether the specified character sequence is a valid email address. The optional parameters regexp and flags allow to specify an additional regular expression (including regular expression flags) which the email must match.	没有
@Length(min=, max=)	CharSequence	Validates that the annotated character sequence is between min and max included.	对应的数据库表字段的长度会被设置成约束中定义的最大值.
@ModCheck(modType=, multiplier=, startIndex=, endIndex=, checkDigitPosition=, ignoreNonDigitCharacters=)	CharSequence	Checks that the digits within the annotated character sequence pass the mod 10 or mod 11 checksum algorithm. modType is used to select the modulo type and the multiplier determines the algorithm specific	没有

Annotation	Supported data types	作用	Hibernate metadata impact
		multiplier (see also Luhn algorithm [http://en.wikipedia.org/wiki/Luhn_algorithm]). startIndex and endIndex allow to only run the modulo algorithm on the specified sub-string. checkDigitPosition allows to use an arbitrary digit within the character sequence to be the check digit. If not specified it is assumed that the check digit is part of the specified range. Last but not least, ignoreNonDigitCharacters allows to ignore non digit characters.	
@NotBlank	CharSequence	Checks that the annotated character sequence is not null and the trimmed length is greater than 0. The difference to @NotEmpty is that this constraint can only be applied on strings and that trailing	没有

Annotation	Supported data types	作用	Hibernate metadata impact
		whitespaces are ignored.	
@NotEmpty	CharSequence, Collection, Map and arrays	Checks whether the annotated element is not null nor empty.	没有
@Range(min=, max=)	BigDecimal, BigInteger, CharSequence, byte, short, int, long and the respective wrappers of the primitive types	Checks whether the annotated value lies between (inclusive) the specified minimum and maximum.	没有
@SafeHtml(whitelistType=, additionalTags=)	CharSequence	Checks whether the annotated value contains potentially malicious fragments such as <script/>. In order to use this constraint, the jsoup [http://jsoup.org/] library must be part of the class path. With the whitelistType attribute predefined whitelist types can be chosen. You can also specify additional html tags for the whitelist with the additionalTags attribute.	没有
@ScriptAssert(lang=, script=, alias=)	Any type	Checks whether the given script can successfully be evaluated against the annotated	没有

Annotation	Supported data types	作用	Hibernate metadata impact
		element. In order to use this constraint, an implementation of the Java Scripting API as defined by JSR 223 ("Scripting for the Java^{TM} Platform") must part of the class path. The expressions to be evaluated can be written in any scripting or expression language, for which a JSR 223 compatible engine can be found in the class path.	
@URL(protocol=, host=, port=, regexp=, flags=)	CharSequence	Checks if the annotated character sequence is a valid URL according to RFC2396. If any of the optional parameters protocol, host or port are specified, the corresponding URL fragments must match the specified values. The optional parameters regexp and flags allow to specify an additional regular expression (including regular expression flags)	没有

Annotation	Supported data types	作用	Hibernate metadata impact
		which the URL must match.	

2.4.2.1. Country specific constraints

Hibernate Validator offers also some country specific constraints, e.g. for the validation of social security numbers.

> **注意**
>
> If you have to implement a country specific constraint, consider making it a contribution to Hibernate Validator!

表 2.4. Custom country specific constraints

Annotation	Supported data types	作用	Country	Hibernate metadata impact
@CNPJ	CharSequence	Checks that the annotated character sequence represents a Brazilian corporate tax payer registry number (Cadastro de Pessoa Juríeddica)	Brazil	没有
@CPF	CharSequence	Checks that the annotated character sequence represents a Brazilian individual taxpayer registry number (Cadastro de Pessoa Fídsica).	Brazil	没有

Annotation	Supported data types	作用	Country	Hibernate metadata impact
@TituloEleitoral	CharSequence	Checks that the annotated character sequence represents a Brazilian voter ID card number (Título Eleitoral [http://ghiorzi.org/cgcancpf.htm]).	Brazil	没有

提示

In some cases neither the Bean Validation constraints nor the custom constraints provided by Hibernate Validator will fulfill your requirements. In this case you can easily write your own constraint. We will discuss this in 第 3 章 创建自己的约束规则.

创建自己的约束规则

尽管Bean Validation API定义了一大堆标准的约束条件，但是肯定还是有这些约束不能满足我们需求的时候，在这种情况下，你可以根据你的特定的校验需求来创建自己的约束条件.

3.1. 创建一个简单的约束条件

按照以下三个步骤来创建一个自定义的约束条件

- 创建约束标注
- 实现一个验证器
- 定义默认的验证错误信息

3.1.1. 约束标注

让我们来创建一个新的用来判断一个给定字符串是否全是大写或者小写字符的约束标注. 我们将稍后把它用在第 1 章 开始入门中的类Car的licensePlate字段上来确保这个字段的内容一直都是大写字母.

首先,我们需要一种方法来表示这两种模式(译注: 大写或小写)，我们可以使用String常量，但是在Java 5中，枚举类型是个更好的选择:

例 3.1. 枚举类型CaseMode，来表示大写或小写模式.

```
package com.mycompany;

public enum CaseMode {
    UPPER,
    LOWER;
}
```

现在我们可以来定义真正的约束标注了. 如果你以前没有创建过标注(annotation)的话,那么这个可能看起来有点吓人，可是其实没有那么难的 :)

例 3.2. 定义一个CheckCase的约束标注

```
package com.mycompany;

import static java.lang.annotation.ElementType.*;
import static java.lang.annotation.RetentionPolicy.*;

import java.lang.annotation.Documented;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;
import javax.validation.Constraint;
import javax.validation.Payload;

@Target( { METHOD, FIELD, ANNOTATION_TYPE })
@Retention(RUNTIME)
@Constraint(validatedBy = CheckCaseValidator.class)
@Documented
public @interface CheckCase {

    String message() default "{com.mycompany.constraints.checkcase}";

    Class<?>[] groups() default {};

    Class<? extends Payload>[] payload() default {};

    CaseMode value();

}
```

一个标注(annotation) 是通过`@interface`关键字来定义的. 这个标注中的属性是声明成类似方法的样式的. 根据Bean Validation API 规范的要求

- message属性, 这个属性被用来定义默认得消息模版, 当这个约束条件被验证失败的时候,通过此属性来输出错误信息.
- groups 属性, 用于指定这个约束条件属于哪(些)个校验组(请参考第 2.3 节 “校验组”). 这个的默认值必须是`Class<?>`类型到空到数组.
- payload 属性, Bean Validation API 的使用者可以通过此属性来给约束条件指定严重级别. 这个属性并不被API自身所使用.

提示

通过payload属性来指定默认错误严重级别的示例

```
public class Severity {
    public static class Info extends Payload {};
    public static class Error extends Payload {};
}

public class ContactDetails {
    @NotNull(message="Name is mandatory", payload=Severity.Error.class)
    private String name;

        @NotNull(message="Phone  number  not  specified,  but  not  mandatory",
 payload=Severity.Info.class)
    private String phoneNumber;

    // ...
}
```

这样，在校验完一个ContactDetails的示例之后，你就可以通过调用ConstraintViolation.getConstraintDescriptor().getPayload()来得到之前指定到错误级别了,并且可以根据这个信息来决定接下来到行为.

除了这三个强制性要求的属性(message, groups 和 payload) 之外, 我们还添加了一个属性用来指定所要求到字符串模式. 此属性的名称value在annotation的定义中比较特殊, 如果只有这个属性被赋值了的话, 那么, 在使用此annotation到时候可以忽略此属性名称, 即@CheckCase(CaseMode.UPPER).

另外, 我们还给这个annotation标注了一些(所谓的) 元标注(译注: 或"元模型信息"?, "meta annotatioins"):

- @Target({ METHOD, FIELD, ANNOTATION_TYPE }): 表示@CheckCase 可以被用在方法, 字段或者annotation声明上.
- @Retention(RUNTIME): 表示这个标注信息是在运行期通过反射被读取的.
- @Constraint(validatedBy = CheckCaseValidator.class): 指明使用那个校验器(类) 去校验使用了此标注的元素.
- @Documented: 表示在对使用了@CheckCase的类进行javadoc操作到时候, 这个标注会被添加到javadoc当中.

提示

Hibernate Validator provides support for the validation of method parameters using constraint annotations (see 第 8.3 节 “Method validation”).

In order to use a custom constraint for parameter validation the ElementType.PARAMETER must be specified within the @Target annotation. This is already the case for all constraints defined by the Bean Validation API and also the custom constraints provided by Hibernate Validator.

3.1.2. 约束校验器

Next, we need to implement a constraint validator, that's able to validate elements with a @CheckCase annotation. To do so, we implement the interface ConstraintValidator as shown below:

例 3.3. 约束条件CheckCase的验证器

```
package com.mycompany;
```

```
import javax.validation.ConstraintValidator;
import javax.validation.ConstraintValidatorContext;

public class CheckCaseValidator implements ConstraintValidator<CheckCase, String> {

    private CaseMode caseMode;

    public void initialize(CheckCase constraintAnnotation) {
        this.caseMode = constraintAnnotation.value();
    }

    public boolean isValid(String object, ConstraintValidatorContext constraintContext) {

        if (object == null)
            return true;

        if (caseMode == CaseMode.UPPER)
            return object.equals(object.toUpperCase());
        else
            return object.equals(object.toLowerCase());
    }

}
```

ConstraintValidator定义了两个泛型参数， 第一个是这个校验器所服务到标注类型(在我们的例子中即CheckCase)，第二个这个校验器所支持到被校验元素到类型 (即String).

如果一个约束标注支持多种类型到被校验元素的话， 那么需要为每个所支持的类型定义一个ConstraintValidator,并且注册到约束标注中.

这个验证器的实现就很平常了， initialize() 方法传进来一个所要验证的标注类型的实例， 在本例中，我们通过此实例来获取其value属性的值,并将其保存为CaseMode类型的成员变量供下一步使用.

isValid()是实现真正的校验逻辑的地方， 判断一个给定的String对于@CheckCase这个约束条件来说是否是合法的，同时这还要取决于在initialize()中获得的大小写模式. 根据Bean Validation中所推荐的做法， 我们认为null是合法的值. 如果null对于这个元素来说是不合法的话,那么它应该使用@NotNull来标注.

3.1.2.1. ConstraintValidatorContext

例 3.3 “约束条件CheckCase的验证器” 中的isValid使用了约束条件中定义的错误消息模板，然后返回一个true 或者 false. 通过使用传入的ConstraintValidatorContext对象，我们还可以给约束条件中定义的错误信息模板来添加额外的信息或者完全创建一个新的错误信息模板.

例 3.4. 使用ConstraintValidatorContext来自定义错误信息

```
package com.mycompany;

import javax.validation.ConstraintValidator;
import javax.validation.ConstraintValidatorContext;
```

```
public class CheckCaseValidator implements ConstraintValidator<CheckCase, String> {

    private CaseMode caseMode;

    public void initialize(CheckCase constraintAnnotation) {
        this.caseMode = constraintAnnotation.value();
    }

    public boolean isValid(String object, ConstraintValidatorContext constraintContext) {

        if (object == null)
            return true;

        boolean isValid;
        if (caseMode == CaseMode.UPPER) {
            isValid = object.equals(object.toUpperCase());
        }
        else {
            isValid = object.equals(object.toLowerCase());
        }

        if(!isValid) {
            constraintContext.disableDefaultConstraintViolation();
            constraintContext.buildConstraintViolationWithTemplate( "{com.mycompany.constraints.CheckCase.message
        }
        return result;
    }

}
```

例 3.4 “使用ConstraintValidatorContext来自定义错误信息” 演示了如果创建一个新的错误信息模板来替换掉约束条件中定义的默认的. 在本例中, 实际上通过调用ConstraintValidatorContext达到了一个使用默认消息模板的效果.

提示

在创建新的constraint violation的时候一定要记得调用addConstraintViolation, 只有这样, 这个新的constraint violation才会被真正的创建.

In case you are implementing a ConstraintValidator a class level constraint it is also possible to adjust set the property path for the created constraint violations. This is important for the case where you validate multiple properties of the class or even traverse the object graph. A custom property path creation could look like 例 3.5 “Adding new ConstraintViolation with custom property path” .

例 3.5. Adding new ConstraintViolation with custom property path

```
public boolean isValid(Group group, ConstraintValidatorContext constraintValidatorContext) {
    boolean isValid = false;
    ...
    if(!isValid) {
        constraintValidatorContext
            .buildConstraintViolationWithTemplate( "{my.custom.template}" )
            .addNode( "myProperty" ).addConstraintViolation();
    }
    return isValid;
}
```

3.1.3. 校验错误信息

最后， 我们还需要指定如果@CheckCase这个约束条件验证的时候,没有通过的话的校验错误信息. 我们可以添加下面的内容到我们项目自定义的ValidationMessages.properties (参考 第 2.2.4 节 “验证失败提示信息解析”)文件中.

例 3.6. 为CheckCase约束定义一个错误信息

```
com.mycompany.constraints.CheckCase.message=Case mode must be {value}.
```

如果发现校验错误了的话， 你所使用的Bean Validation的实现会用我们定义在@CheckCase中message属性上的值作为键到这个文件中去查找对应的错误信息.

3.1.4. 应用约束条件

现在我们已经有了一个自定义的约束条件了， 我们可以把它用在第 1 章 开始入门中的Car类上， 来校验此类的licensePlate属性的值是否全都是大写字母.

例 3.7. 应用CheckCase约束条件

```
package com.mycompany;

import javax.validation.constraints.Min;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;

public class Car {

    @NotNull
    private String manufacturer;

    @NotNull
    @Size(min = 2, max = 14)
    @CheckCase(CaseMode.UPPER)
    private String licensePlate;

    @Min(2)
    private int seatCount;

    public Car(String manufacturer, String licencePlate, int seatCount) {

        this.manufacturer = manufacturer;
        this.licensePlate = licencePlate;
        this.seatCount = seatCount;
    }

    //getters and setters ...

}
```

最后,让我们用一个简单的测试来检测@CheckCase约束已经被正确的校验了:

例 3.8. 演示CheckCase的验证过程

```
package com.mycompany;

import static org.junit.Assert.*;

import java.util.Set;

import javax.validation.ConstraintViolation;
import javax.validation.Validation;
import javax.validation.Validator;
import javax.validation.ValidatorFactory;

import org.junit.BeforeClass;
import org.junit.Test;

public class CarTest {

    private static Validator validator;

    @BeforeClass
    public static void setUp() {
        ValidatorFactory factory = Validation.buildDefaultValidatorFactory();
        validator = factory.getValidator();
    }

    @Test
    public void testLicensePlateNotUpperCase() {

        Car car = new Car("Morris", "dd-ab-123", 4);

        Set<ConstraintViolation<Car>> constraintViolations =
            validator.validate(car);
        assertEquals(1, constraintViolations.size());
        assertEquals(
            "Case mode must be UPPER.",
            constraintViolations.iterator().next().getMessage());
    }

    @Test
    public void carIsValid() {

        Car car = new Car("Morris", "DD-AB-123", 4);

        Set<ConstraintViolation<Car>> constraintViolations =
            validator.validate(car);
        assertEquals(0, constraintViolations.size());
    }
}
```

3.2. 约束条件组合

在例 3.7 “应用CheckCase约束条件”中我们可以看到，类Car的licensePlate属性上定义了三个约束条件. 在某些复杂的场景中，可能还会有更多的约束条件被定义到同一个元素上面，这可能会让代码看起来有些复杂，另外，如果在另外的类里面还有一个licensePlate属性，我们可能还要把这些约束条件再拷贝到这个属性上，但是这样做又违反了 DRY 原则.

这个问题可以通过使用组合约束条件来解决. 接下来让我们来创建一个新的约束标注@ValidLicensePlate，它组合了@NotNull，@Size 和 @CheckCase:

例 3.9. 创建一个约束条件组合ValidLicensePlate

```
package com.mycompany;

import static java.lang.annotation.ElementType.*;
import static java.lang.annotation.RetentionPolicy.*;

import java.lang.annotation.Documented;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;

import javax.validation.Constraint;
import javax.validation.Payload;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;

@NotNull
@Size(min = 2, max = 14)
@CheckCase(CaseMode.UPPER)
@Target( { METHOD, FIELD, ANNOTATION_TYPE })
@Retention(RUNTIME)
@Constraint(validatedBy = {})
@Documented
public @interface ValidLicensePlate {

    String message() default "{com.mycompany.constraints.validlicenseplate}";

    Class<?>[] groups() default {};

    Class<? extends Payload>[] payload() default {};

}
```

我们只需要把要组合的约束标注在这个新的类型上加以声明 (注: 这正是我们为什么把annotation types作为了@CheckCase的一个target). 因为这个组合不需要额外的校验器，所以不需要声明validator属性.

现在，在licensePlate属性上使用这个新定义的"约束条件"（其实是个组合）和之前在其上声明那三个约束条件是一样的效果了.

例 3.10. 使用ValidLicensePlate组合约束

```
package com.mycompany;

public class Car {

    @ValidLicensePlate
    private String licensePlate;

    //...

}
```

The set of ConstraintViolations retrieved when validating a Car instance will contain an entry for each violated composing constraint of the @ValidLicensePlate constraint. If you rather prefer a single ConstraintViolation in case any of the composing constraints is violated, the @ReportAsSingleViolation meta constraint can be used as follows:

例 3.11. @ReportAsSingleViolation的用法

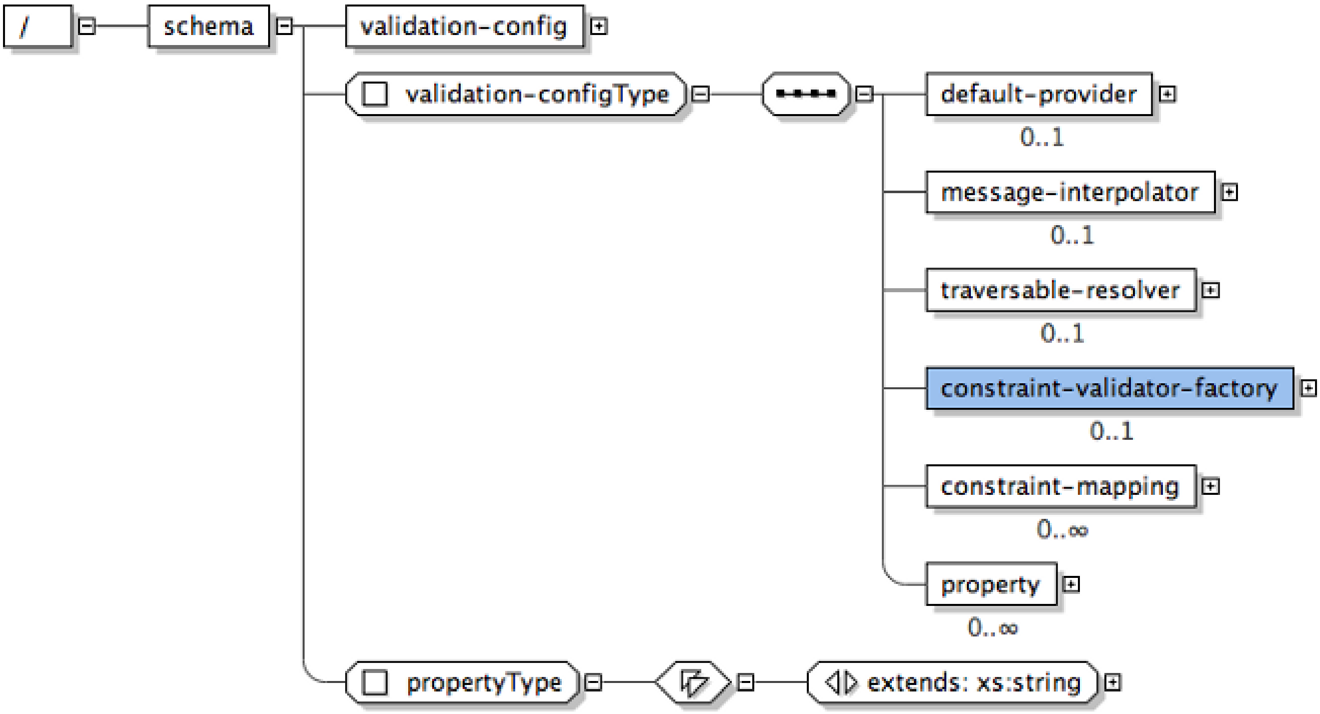
```
//...
@ReportAsSingleViolation
public @interface ValidLicensePlate {

    String message() default "{com.mycompany.constraints.validlicenseplate}";

    Class<?>[] groups() default {};

    Class<? extends Payload>[] payload() default {};

}
```

XML configuration

4.1. validation.xml

The key to enable XML configuration for Hibernate Validator is the file META-INF/validation.xml. If this file exists on the classpath its configuration will be applied when the ValidationFactory gets created. 例 4.1 "validation-configuration-1.0.xsd" shows a model view of the XML schema to which validation.xml has to adhere.

例 4.1. validation-configuration-1.0.xsd


/
schema
validation-config
validation-configType
default-provider
0..1
message-interpolator
0..1
traversable-resolver
0..1
constraint-validator-factory
0..1
constraint-mapping
0..∞
property
0..∞
propertyType
extends: xs:string

例 4.2 "validation.xml" 列出了validation.xml中的一些常用的配置项.

例 4.2. validation.xml

```
<validation-config
    xmlns="http://jboss.org/xml/ns/javax/validation/configuration"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://jboss.org/xml/ns/javax/validation/configuration">

    <default-provider>org.hibernate.validator.HibernateValidator</default-provider>
                                                                         <message-
interpolator>org.hibernate.validator.messageinterpolation.ResourceBundleMessageInterpolator</
message-interpolator>
     <traversable-resolver>org.hibernate.validator.engine.resolver.DefaultTraversableResolver</
traversable-resolver>
```

```
                                                                        <constraint-validator-
factory>org.hibernate.validator.engine.ConstraintValidatorFactoryImpl</constraint-validator-
factory>
    <constraint-mapping>META-INF/validation/constraints-car.xml</constraint-mapping>
    <property name="prop1">value1</property>
    <property name="prop2">value2</property>
</validation-config>
```

警告

There must only be one file named `META-INF/validation.xml` on the classpath. If more than one is found an exception is thrown.

The node `default-provider` allows to choose the Bean Validation provider. This is useful if there is more than one provider on the classpath. `message-interpolator`, `traversable-resolver` and `constraint-validator-factory` allow to customize the used implementations for `javax.validation.MessageInterpolator`, `javax.validation.TraversableResolver` resp. `javax.validation.ConstraintValidatorFactory`.

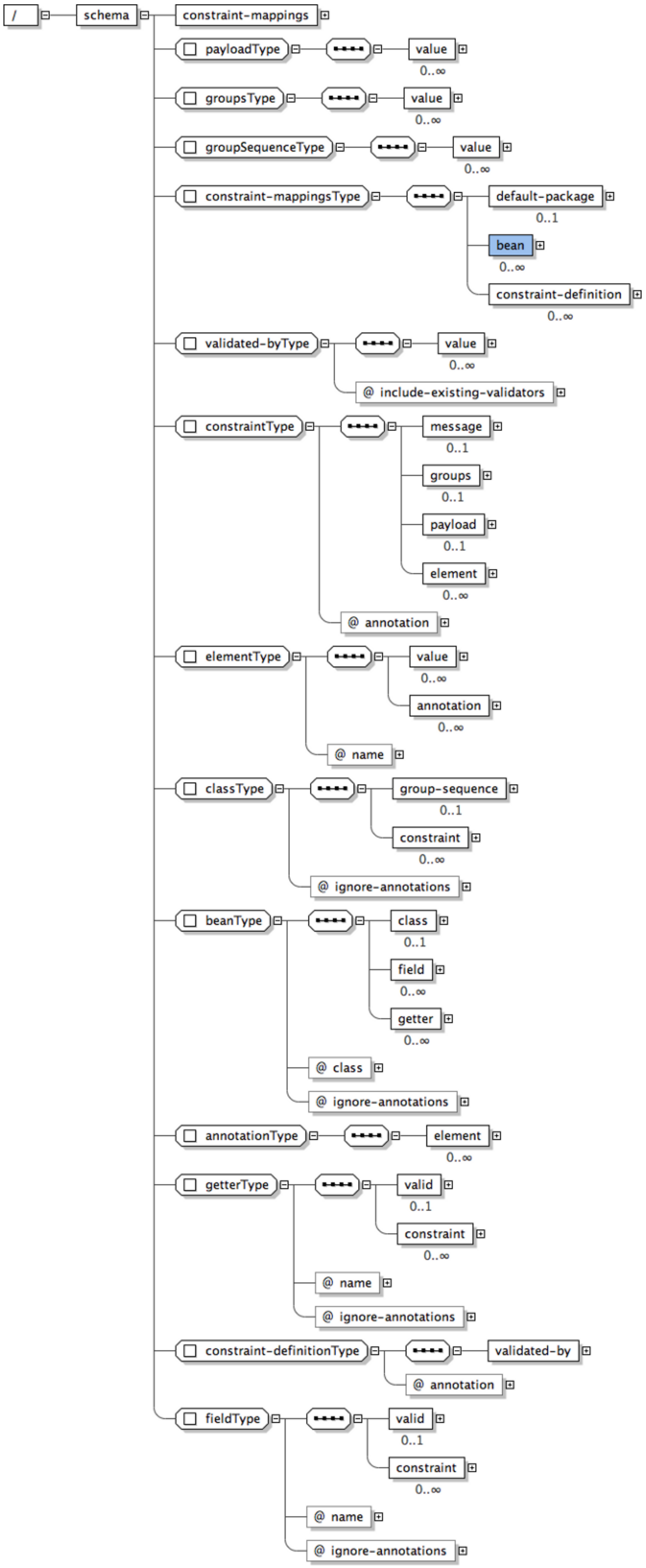
All these settings are optional. 例 4.2 “validation.xml” shows the defaults used within Hibernate Validator. The same configuration options are also available programmatically through `javax.validation.Configuration`. In fact XML configuration will be overridden by values explicitly specified via the API. It is even possible to ignore the XML configuration completely via `Configuration.ignoreXmlConfiguration()`. See also 第 5 章 Bootstrapping.

Via the constraint-mapping element you can list an arbitrary number of additional XML files containing the actual constraint configuration. Mapping file names must be specified using their fully-qualified name on the classpath. Details on writing mapping files can be found in the next section.

Last but not least, you can specify provider specific properties via the property nodes.

4.2. 映射约束

Expressing constraints in XML is possible via files adhering to the schema seen in 例 4.3 “validation-mapping-1.0.xsd” . Note that these mapping files are only processed if listed via constraint-mapping in your `validation.xml`.


/
schema
constraint-mappings
payloadType
value
0..∞
groupsType
value
0..∞
groupSequenceType
value
0..∞
constraint-mappingsType
default-package
0..1
bean
0..∞
constraint-definition
0..∞
validated-byType
value
0..∞
@ include-existing-validators
constraintType
message
0..1
groups
0..1
payload
0..1
element
0..∞
@ annotation
elementType
value
0..∞
annotation
0..∞
@ name
classType
group-sequence
0..1
constraint
0..∞
@ ignore-annotations
beanType
class
0..1
field
0..∞
getter
0..∞
@ class
@ ignore-annotations
annotationType
element
0..∞
getterType
valid
0..1
constraint
0..∞
@ name
@ ignore-annotations
constraint-definitionType
validated-by
@ annotation
fieldType
valid
0..1
constraint
0..∞
@ name
@ ignore-annotations

例 4.4 “constraints-car.xml” shows how our classes Car and RentalCar from 例 2.15 “Car” resp. 例 2.19 “RentalCar with @GroupSequence” could be mapped in XML.

例 4.4. constraints-car.xml

```xml
<constraint-mappings
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:schemaLocation="http://jboss.org/xml/ns/javax/validation/mapping validation-
mapping-1.0.xsd"
    xmlns="http://jboss.org/xml/ns/javax/validation/mapping">

    <default-package>org.hibernate.validator.quickstart</default-package>
    <bean class="Car" ignore-annotations="true">
        <field name="manufacturer">
            <constraint annotation="javax.validation.constraints.NotNull"/>
        </field>
        <field name="licensePlate">
            <constraint annotation="javax.validation.constraints.NotNull"/>
        </field>
        <field name="seatCount">
            <constraint annotation="javax.validation.constraints.Min">
                <element name="value">2</element>
            </constraint>
        </field>
        <field name="driver">
            <valid/>
        </field>
        <getter name="passedVehicleInspection" ignore-annotations="true">
            <constraint annotation="javax.validation.constraints.AssertTrue">
                <message>The car has to pass the vehicle inspection first</message>
                <groups>
                    <value>CarChecks</value>
                </groups>
                <element name="max">10</element>
            </constraint>
        </getter>
    </bean>
    <bean class="RentalCar" ignore-annotations="true">
        <class ignore-annotations="true">
            <group-sequence>
                <value>RentalCar</value>
                <value>CarChecks</value>
            </group-sequence>
        </class>
    </bean>
    <constraint-definition annotation="org.mycompany.CheckCase">
        <validated-by include-existing-validators="false">
            <value>org.mycompany.CheckCaseValidator</value>
        </validated-by>
    </constraint-definition>
</constraint-mappings>
```

The XML configuration is closely mirroring the programmatic API. For this reason it should suffice to just add some comments. default-package is used for all fields where a class name is expected. If the specified class is not fully qualified the configured default package will be used. Every mapping file can then have several bean nodes, each describing the constraints on the entity with the specified class name.

警告

A given entity can only be configured once across all configuration files. If the same class is configured more than once an exception is thrown.

Setting ignore-annotations to true means that constraint annotations placed on the configured bean are ignored. The default for this value is true. ignore-annotations is also available for the nodes class, fields and getter. If not explicitly specified on these levels the configured bean value applies. Otherwise do the nodes class, fields and getter determine on which level the constraints are placed (see 第 2.1 节 “定义约束”). The constraint node is then used to add a constraint on the corresponding level. Each constraint definition must define the class via the annotation attribute. The constraint attributes required by the Bean Validation specification (message, groups and payload) have dedicated nodes. All other constraint specific attributes are configured using the element node.

class节点同样支持通过group-sequence节点来对一个类的默认校验组进行重定义(请参考 第 2.3.2 节 “对一个类重定义其默认校验组”).

Last but not least, the list of ConstraintValidators associated to a given constraint can be altered via the constraint-definition node. The annotation attribute represents the constraint annotation being altered. The validated-by elements represent the (ordered) list of ConstraintValidator implementations associated to the constraint. If include-existing-validator is set to false, validators defined on the constraint annotation are ignored. If set to true, the list of constraint validators described in XML is concatenated to the list of validators specified on the annotation.

Bootstrapping

在第 5.1 节 “Configuration 和 ValidatorFactory”中我们说道过， 最简单的创建一个Validator实例的方法是通过Validation.buildDefaultValidatorFactory. 在本章中我们会继续介绍javax.validation.Validation中的其他方法，以及如何通过这些方法在Bean Validation初始化的时候对其进行配置的.

The different bootstrapping options allow, amongst other things, to bootstrap any Bean Validation implementation on the classpath. Generally, an available provider is discovered by the Java Service Provider [http://docs.oracle.com/javase/6/docs/technotes/guides/jar/jar.html#Service%20Provider] mechanism. A Bean Validation implementation includes the file javax.validation.spi.ValidationProvider in META-INF/services. This file contains the fully qualified classname of the ValidationProvider of the implementation. In the case of Hibernate Validator this is org.hibernate.validator.HibernateValidator.

> **注意**
>
> 如果当前类路径下存在多个Bean Validation的实现, 那么Validation.buildDefaultValidatorFactory()并不能保证具体那个实现会被使用. 如果想指定某一个的话，请使用Validation.byProvider().

5.1. Configuration 和 ValidatorFactory

Validation类提供了三种方法来创建一个Validator的实例, 例 5.1 “Validation.buildDefaultValidatorFactory()” 中显示的是最简单的方法.

例 5.1. Validation.buildDefaultValidatorFactory()

```
ValidatorFactory factory = Validation.buildDefaultValidatorFactory();
Validator validator = factory.getValidator();
```

你也可以通过Validation.byDefaultProvider()现获取一个Configuration对象， 这样可以对要创建的Validator进行配置.

例 5.2. Validation.byDefaultProvider()

```
Configuration<?> config = Validation.byDefaultProvider().configure();
config.messageInterpolator(new MyMessageInterpolator())
    .traversableResolver( new MyTraversableResolver())
    .constraintValidatorFactory(new MyConstraintValidatorFactory());

ValidatorFactory factory = config.buildValidatorFactory();
```

```
Validator validator = factory.getValidator();
```

MessageInterpolator，TraversableResolver 和 ConstraintValidatorFactory会在后面详细介绍.

最后， 你可以指定使用哪个Bean Validation的实现. 如果类路径下存在多个Bean Validation的实现的话,这样就很有必要了. 例如， 如果你想使用Hibernate Validator来作为内部实现来创建Validator的话:

例 5.3. Validation.byProvider(HibernateValidator.class)

```
HibernateValidatorConfiguration config = Validation.byProvider( HibernateValidator.class ).configure();
config.messageInterpolator(new MyMessageInterpolator())
    .traversableResolver( new MyTraversableResolver())
    .constraintValidatorFactory(new MyConstraintValidatorFactory());

ValidatorFactory factory = config.buildValidatorFactory();
Validator validator = factory.getValidator();
```

> **提示**
>
> 创建出来的Validator实例是线程安全的，所以你可以把它缓存起来.

5.2. ValidationProviderResolver

如果 Java Service Provider机制在你的环境中不能够正常工作， 或者你有特别的classloader设置的话， 你也可以提供一个自定义的ValidationProviderResolver.例 5.4 “使用自定义的ValidationProviderResolver” 显示了如何在OSGi环境中插入自定义的provider resolver.

例 5.4. 使用自定义的ValidationProviderResolver

```
Configuration<?> config = Validation.byDefaultProvider()
    .providerResolver( new OSGiServiceDiscoverer() )
    .configure();

ValidatorFactory factory = config.buildValidatorFactory();
Validator validator = factory.getValidator();
```

在这种情况下，你的OSGiServiceDiscoverer类需要实现ValidationProviderResolver接口:

例 5.5. ValidationProviderResolver接口

```
public interface ValidationProviderResolver {
    /**
     * Returns a list of ValidationProviders available in the runtime environment.
     *
     * @return list of validation providers.
     */
    List<ValidationProvider<?>> getValidationProviders();
}
```

5.3. MessageInterpolator

第 2.2.4 节 “验证失败提示信息解析” already discussed the default message interpolation algorithm. If you have special requirements for your message interpolation you can provide a custom interpolator using `Configuration.messageInterpolator()`. This message interpolator will be shared by all validators generated by the `ValidatorFactory` created from this `Configuration`. 例 5.6 “自定义的MessageInterpolator” shows an interpolator (available in Hibernate Validator) which can interpolate the value being validated in the constraint message. To refer to this value in the constraint message you can use:

- ${validatedValue}: this will call `String.valueOf` on the validated value.
- ${validatedValue:<format>}: provide your own format string which will be passed to `String.format` together with the validated value. Refer to the javadoc of `String.format` for more information about the format options.

例 5.6. 自定义的MessageInterpolator

```
Configuration<?> configuration = Validation.byDefaultProvider().configure();
ValidatorFactory factory = configuration
    .messageInterpolator(new ValueFormatterMessageInterpolator(configuration.getDefaultMessageInterpolator()))
    .buildValidatorFactory();

Validator validator = factory.getValidator();
```

提示

It is recommended that `MessageInterpolator` implementations delegate final interpolation to the Bean Validation default `MessageInterpolator` to ensure standard Bean Validation interpolation rules are followed. The default implementation is accessible through `Configuration.getDefaultMessageInterpolator()`.

5.3.1. ResourceBundleLocator

一个普遍的需求是你可能需要为错误消息解析指定你自己的resource bundles. `ResourceBundleMessageInterpolator`是Hibernate Validator中默认的`MessageInterpolator`的实现, 它默认情况下是通过`ResourceBundle.getBundle`来获取resource bundle的. 不过, `ResourceBundleMessageInterpolator`也支持你指定一个自定义的`ResourceBundleLocator`实现来提

供你自己的resource bundle. 例 5.7 “自定义的ResourceBundleLocator”提供了一个示例. 在这个例子中， 先通过 HibernateValidatorConfiguration.getDefaultResourceBundleLocator获取默认的ResourceBundleLocator实现，然后再用你自定义的实现把默认的包装起来，代理模式.

例 5.7. 自定义的ResourceBundleLocator

```java
HibernateValidatorConfiguration configure = Validation.byProvider(HibernateValidator.class).configure();

ResourceBundleLocator defaultResourceBundleLocator = configure.getDefaultResourceBundleLocator();
ResourceBundleLocator myResourceBundleLocator = new MyCustomResourceBundleLocator(defaultResourceBundleLocator);

configure.messageInterpolator(new ResourceBundleMessageInterpolator(myResourceBundleLocator));
```

Hibernate Validator提供了两个ResourceBundleLocator的实现 - PlatformResourceBundleLocator (默认) 和 AggregateResourceBundleLocator. 后者可以定义一系列的resource bundle，然后它会读取这些文件，并且把它们组合成一个. 更多信息请参考此类的javadoc 文档.

5.4. TraversableResolver

到目前位置我们还没有讨论过TraversableResolver接口，它的设计目的是在某些情况下，我们可能不应该去获取一个属性的状态. 最典型的情况就是一个延迟加载的属性或者与JPA中涉及到关联关系的时候. 当验证这两种情况的属性的时候，很可能会触发一次对数据库的查询.Bean Validation 正是通过TraversableResolver接口来控制能否访问某一个属性的 (例 5.8 “TraversableResolver 接口”).

例 5.8. TraversableResolver接口

```java
/**
 * Contract determining if a property can be accessed by the Bean Validation provider
 * This contract is called for each property that is being either validated or cascaded.
 *
 * A traversable resolver implementation must be thread-safe.
 *
 */
public interface TraversableResolver {
    /**
     * Determine if the Bean Validation provider is allowed to reach the property state
     *
     * @param traversableObject object hosting <code>traversableProperty</code> or null
     *                          if validateValue is called
     * @param traversableProperty the traversable property.
     * @param rootBeanType type of the root object passed to the Validator.
     * @param pathToTraversableObject path from the root object to
     *        <code>traversableObject</code>
     *        (using the path specification defined by Bean Validator).
     * @param elementType either <code>FIELD</code> or <code>METHOD</code>.
     *
     * @return <code>true</code> if the Bean Validation provider is allowed to
     *         reach the property state, <code>false</code> otherwise.
     */
    boolean isReachable(Object traversableObject,
                        Path.Node traversableProperty,
                        Class<?> rootBeanType,
                        Path pathToTraversableObject,
                        ElementType elementType);

    /**
     * Determine if the Bean Validation provider is allowed to cascade validation on
     * the bean instance returned by the property value
     * marked as <code>@Valid</code>.
     * Note that this method is called only if isReachable returns true for the same set of
     * arguments and if the property is marked as <code>@Valid</code>
     *
     * @param traversableObject object hosting <code>traversableProperty</code> or null
     *                          if validateValue is called
     * @param traversableProperty the traversable property.
     * @param rootBeanType type of the root object passed to the Validator.
     * @param pathToTraversableObject path from the root object to
     *        <code>traversableObject</code>
     *        (using the path specification defined by Bean Validator).
     * @param elementType either <code>FIELD</code> or <code>METHOD</code>.
     *
     * @return <code>true</code> if the Bean Validation provider is allowed to
     *         cascade validation, <code>false</code> otherwise.
     */
    boolean isCascadable(Object traversableObject,
                         Path.Node traversableProperty,
                         Class<?> rootBeanType,
                         Path pathToTraversableObject,
                         ElementType elementType);
}
```

Hibernate Validator provides two TraversableResolvers out of the box which will be enabled automatically depending on your environment. The first is the DefaultTraversableResolver which will always return true for isReachable() and isTraversable(). The second is the JPATraversableResolver which gets enabled when Hibernate Validator gets used in combination with JPA 2. In case you have to provide your own resolver you can do so again using the Configuration object as seen in 例 5.9 “自定义的TraversableResolver” .

例 5.9. 自定义的TraversableResolver

```
Configuration<?> configuration = Validation.byDefaultProvider().configure();
ValidatorFactory factory = configuration
    .traversableResolver(new MyTraversableResolver())
    .buildValidatorFactory();

Validator validator = factory.getValidator();
```

5.5. ConstraintValidatorFactory

最后，还有个配置项得提一下，那就是ConstraintValidatorFactory类. Hibernate Validator中默认的ConstraintValidatorFactory需要一个无参的构造方法来初始化ConstraintValidator的实例(参

考第 3.1.2 节 “约束校验器”）. 对于自定义的ConstraintValidatorFactory实现来说，例如，你可以让其支持对约束条件的依赖注入等功能. 配置使用这个自定义的ConstraintValidatorFactory的方法还是老样子(例 5.10 “自定义的ConstraintValidatorFactory”）.

例 5.10. 自定义的ConstraintValidatorFactory

```
Configuration<?> configuration = Validation.byDefaultProvider().configure();
ValidatorFactory factory = configuration
    .constraintValidatorFactory(new IOCConstraintValidatorFactory())
    .buildValidatorFactory();

Validator validator = factory.getValidator();
```

你需要实现此接口:

例 5.11. ConstraintValidatorFactory接口

```
public interface ConstraintValidatorFactory {
    /**
     * @param key The class of the constraint validator to instantiate.
     *
     * @return A constraint validator instance of the specified class.
     */
     <T extends ConstraintValidator<?,?>> T getInstance(Class<T> key);
}
```

警告

如果一个约束条件的实现需要依赖ConstraintValidatorFactory的某个特定的行为(例如依赖注入或者没有无参的构造方法等）都可能导致不可移植.

注意

ConstraintValidatorFactory不应该缓存其创建的实例，因为每个实例都可能在其的初始化方法中被修改.

第 6

Metadata API

The Bean Validation specification provides not only a validation engine, but also a metadata repository for all defined constraints. The following paragraphs are discussing this API. All the introduced classes can be found in the `javax.validation.metadata` package.

6.1. BeanDescriptor

The entry into the metadata API is via `Validator.getConstraintsForClass` which returns an instance of the BeanDescriptor [http://docs.jboss.org/hibernate/stable/beanvalidation/api/javax/validation/metadata/BeanDescriptor.html] interface. Using this bean descriptor you can determine whether the specified class hosts any constraints at all via `beanDescriptor.isBeanConstrained`.

提示

If a constraint declaration hosted by the requested class is invalid, a `ValidationException` is thrown.

You can then call `beanDescriptor.getConstraintDescriptors` to get a set of `ConstraintDescriptor`s representing all class level constraints.

If you are interested in property level constraints, you can call `beanDescriptor.getConstraintsForProperty` or `beanDescriptor.getConstrainedProperties` to get a single resp. set of `PropertyDescriptor`s (see 第 6.2 节 “PropertyDescriptor”).

6.2. PropertyDescriptor

The PropertyDescriptor [http://docs.jboss.org/hibernate/stable/beanvalidation/api/javax/validation/metadata/PropertyDescriptor.html] interface extends the `ElementDescriptor` interface and represents constraints on properties of a class. The constraint can be declared on the attribute itself or on the getter of the attribute - provided Java Bean naming conventions are respected. A `PropertyDescriptor` adds `isCascaded` (returning `true` if the property is marked with `@Valid`) and `getPropertyName` to the `ElementDescriptor` functionality.

6.3. ElementDescriptor

The ElementDiscriptor [http://docs.jboss.org/hibernate/stable/beanvalidation/api/javax/validation/metadata/ElementDescriptor.html] interface is the common base class for `BeanDescriptor` and `PropertyDescriptor`. Next to the `hasConstraints` and `getConstraintDescriptors` methods it also offers access to the `ConstraintFinder` API which allows you to query the metadata API in a more fine grained way. For example you can restrict your search to constraints described on fields or on getters or a given set of groups. Given an ElementDescriptor instance you just call findConstraints to retrieve a ConstraintFinder instance.

例 6.1. Usage of ConstraintFinder

```
Validator validator = Validation.buildDefaultValidatorFactory().getValidator();
BeanDescriptor beanDescriptor = validator.getConstraintsForClass(Person.class);
PropertyDescriptor propertyDescriptor = beanDescriptor.getConstraintsForProperty("name");
Set<ConstraintDescriptor<?>> constraints = propertyDescriptor.findConstraints()
                                            .declaredOn(ElementType.METHOD)
                                            .unorderedAndMatchingGroups(Default.class)
                                            .lookingAt(Scope.LOCAL_ELEMENT)
                                            .getConstraintDescriptors();
```

例 6.1 “Usage of ConstraintFinder” shows an example on how to use the ConstraintFinder API. Interesting are especially the restrictions unorderedAndMatchingGroups and lookingAt(Scope.LOCAL_ELEMENT [http://docs.jboss.org/hibernate/stable/beanvalidation/api/javax/validation/metadata/Scope.html]). The former allows to only return ConstraintDescriptors matching a specified set of groups wheras the latter allows to distinguish between constraint directly specified on the element (Scope.LOCAL_ELEMENT) or constraints belonging to the element but hosted anywhere in the class hierarchy (Scope.HIERARCHY).

警告

Order is not respected by unorderedAndMatchingGroups, but group inheritance and inheritance via sequence are.

6.4. ConstraintDescriptor

Last but not least, the ConstraintDescriptor [http://docs.jboss.org/hibernate/stable/beanvalidation/api/javax/validation/metadata/ConstraintDescriptor.html] interface describes a single constraint together with its composing constraints. Via an instance of this interface you get access to the constraint annotation and its parameters, as well as the groups the constraint is supposed to be applied on. It also also you to access the pass-through constraint payload (see 例 3.2 “定义一个CheckCase的约束标注”).

与其他框架集成

Hibernate Validator 的设计初衷是在一个分层的应用程序中，约束信息只需要被定义一次(通过在领域模型上标注)，然后在不同的层中进行数据校验.

7.1. OSGi

The Hibernate Validator jar file is conform to the OSGi specification and can be used within any OSGi container. The following lists represent the packages imported and exported by Hibernate Validator. The classes within the exported packages are considered part of Hibernate Validator public API.

> **提示**
>
> The Java Service Provider mechanism used by Bean Validation to automatically discover validation providers doesn't work in an OSGi environment. To solve this, you have to provide a custom ValidationProviderResolver. See 第 5.2 节 “ValidationProviderResolver”

Exported packages

- org.hibernate.validator
- org.hibernate.validator.cfg.*
- org.hibernate.validator.constraints.*
- org.hibernate.validator.group
- org.hibernate.validator.messageinterpolation
- org.hibernate.validator.resourceloading
- org.hibernate.validator.spi.*

Imported packages

- javax.persistence.*, [2.0.0,3.0.0), optional
- javax.validation.*, [1.0.0,2.0.0)
- javax.xml.*
- javax.el.*, [2.0.0,4.0.0)
- org.xml.sax.*
- org.jboss.logging.*, [3.1.0,4.0.0)

- com.fasterxml.classmate.*, 0.8.0
- org.joda.time.*, [1.6.0,2.0.0), optional
- org.jsoup.*, [1.5.2,2.0.0), optional

7.2. 与数据库集成校验

Hibernate Annotations (即 Hibernate 3.5.x) 会自动的把你定已在实体模型上的约束信息添加到其映射信息中. 例如, 假设你的一个实体类的属性上有@NotNull的约束的话, 那么Hibernate在生成创建此实体对应的表的DDL的时候, 会自动的给那个属性对应的字段添加上not null.

If, for some reason, the feature needs to be disabled, set hibernate.validator.apply_to_ddl to false. See also 表 2.2 “Bean Validation constraints” and 表 2.3 “Custom constraints” .

你也可以限制这个DDL约束自动生成的特性只应用到一部分实体类. 只需要设置org.hibernate.validator.group.ddl属性, 这个属性的值是你想要应用此特性的实体类的全路径名称, 每个以逗号分隔.

7.3. ORM集成

Hibernate Validator不仅能够和Hibernate集成工作, 还能够和任何JPA的实现很好的一起工作.

提示

When lazy loaded associations are supposed to be validated it is recommended to place the constraint on the getter of the association. Hibernate replaces lazy loaded associations with proxy instances which get initialized/loaded when requested via the getter. If, in such a case, the constraint is placed on field level the actual proxy instance is used which will lead to validation errors.

7.3.1. 基于Hibernate事件模型的校验

Hibernate Annotations (即 Hibernate 3.5.x) 中包含了一个的 Hibernate 事件监听器(译注: 请阅读Hibernate Core文档了解Hibernate的事件模型) - org.hibernate.cfg.beanvalidation.BeanValidationEventListener [http://fisheye.jboss.org/browse/Hibernate/core/trunk/annotations/src/main/java/org/hibernate/cfg/beanvalidation/BeanValidationEventListener.java] - 来为Hibernate Validator服务. 当一个PreInsertEvent, PreUpdateEvent 或 PreDeleteEvent事件发生的时候, 这个监听器就可以对该事件所涉及到的实体对象进行校验, 如果校验不通过的话, 则抛出异常. 默认情况下, Hibernate在对每个对象进行保存或者修改操作的时候,都会对其进行校验, 而删除操作则不会. 你可以通过javax.persistence.validation.group.pre-persist, javax.persistence.validation.group.pre-update 和 javax.persistence.validation.group.pre-remove属性来定义对应事件发生的时候, 具体要校验

哪(些)个校验组，这个属性的值是要应用的校验组类的全路径，使用逗号分隔. 例 7.1 “自定义 BeanValidationEvenListener” 显示了这几个属性在Hibernate内部定义的默认值，所以，你不需要在你的应用中再重复定义了.

如果发生了违反约束条件的情况， 该监听器会抛出一个运行时的`ConstraintViolationException`异常，此异常包含了一系列的`ConstraintViolation`对象用于描述每个违反了约束条件的情况.

如果类路径上有Hibernate Validator，则Hibernate Annotations (或 Hibernate EntityManager) 会自动调用它， 如果你想避免这种情况， 可以设置`javax.persistence.validation.mode`属性为`none`.

注意

如果实体模型上没有定义约束条件，则不会有任何性能损耗.

如果你想在Hibernate Core中使用上面提到的事件监听器，则可以在`hibernate.cfg.xml`中定义如下的配置信息:

例 7.1. 自定义BeanValidationEvenListener

```
<hibernate-configuration>
    <session-factory>
       ...
                                    <property     name="javax.persistence.validation.group.pre-
persist">javax.validation.groups.Default</property>
                                    <property     name="javax.persistence.validation.group.pre-
update">javax.validation.groups.Default</property>
       <property name="javax.persistence.validation.group.pre-remove"></property>
       ...
       <event type="pre-update">
         <listener class="org.hibernate.cfg.beanvalidation.BeanValidationEventListener"/>
       </event>
       <event type="pre-insert">
         <listener class="org.hibernate.cfg.beanvalidation.BeanValidationEventListener"/>
       </event>
       <event type="pre-delete">
         <listener class="org.hibernate.cfg.beanvalidation.BeanValidationEventListener"/>
       </event>
    </session-factory>
</hibernate-configuration>
```

7.3.2. JPA

If you are using JPA 2 and Hibernate Validator is in the classpath the JPA2 specification requires that Bean Validation gets enabled. The properties `javax.persistence.validation.group.pre-persist`, `javax.persistence.validation.group.pre-update` and `javax.persistence.validation.group.pre-remove` as described in 第 7.3.1 节 “基

于Hibernate事件模型的校验” can in this case be configured in persistence.xml. persistence.xml also defines a node validation-mode which can be set to AUTO, CALLBACK, NONE. The default is AUTO.

对于JPA1来讲， 你需要自己创建和注册Hibernate Validator. 如果你是使用Hibernate EntityManager， 那么你可以把第 7.3.1 节 “基于Hibernate事件模型的校验”中列出来的BeanValidationEventListener类添加到你的项目中，然后再手工注册它.

7.4. 展示层校验

如果你正在使用JSF2或者JBoss Seam™,并且Hibernate Validator (Bean Validation) 在类路径上的话，那么界面上的字段可以被自动校验. 例 7.2 “在JSF2中使用Bean Validation”显示了一个在JSF页面上使用f:validateBean标签的实例. 更多的信息请参考Seam的文档或者JSF2规范.

例 7.2. 在JSF2中使用Bean Validation

```
<h:form>
  <f:validateBean>
    <h:inputText value=”#{model.property}” />
    <h:selectOneRadio value=”#{model.radioProperty}” > ... </h:selectOneRadio>
    <!-- other input components here -->
  </f:validateBean>
</h:form>
```

提示

The integration between JSF 2 and Bean Validation is described in the "Bean Validation Integration" chapter of JSR-314 [http://jcp.org/en/jsr/detail?id=314]. It is interesting to know that JSF 2 implements a custom MessageInterpolator to ensure ensure proper localization. To encourage the use of the Bean Validation message facility, JSF 2 will per default only display the generated Bean Validation message. This can, however, be configured via the application resource bundle by providing the following configuration ({0} is replaced with the Bean Validation message and {1} is replaced with the JSF component label):

```
javax.faces.validator.BeanValidator.MESSAGE={1}: {0}
```

The default is:

```
javax.faces.validator.BeanValidator.MESSAGE={0}
```

Hibernate Validator Specifics

In the following sections we are having a closer look at some of the Hibernate Validator specific features (features which are not part of the Bean Validation specification). This includes the fail fast mode, the programmatic constraint configuration API and boolean composition of composing constraints.

> 注意
>
> The features described in the following sections are not portable between Bean Validation providers/implementations.

8.1. Public API

Let's start, however, with a look at the public API of Hibernate Validator. 表 8.1 “Hibernate Validator public API” lists all packages belonging to this API and describes their purpose.

Any packages not listed in that table are internal packages of Hibernate Validator and are not intended to be accessed by clients. The contents of these internal packages can change from release to release without notice, thus possibly breaking any client code relying on it.

> 注意
>
> In the following table, when a package is public its not necessarily true for its nested packages.

表 8.1. Hibernate Validator public API

Packages	Description
`org.hibernate.validator`	This package contains the classes used by the Bean Validation bootstrap mechanism (eg. validation provider, configuration class). For more details see 第 5 章 Bootstrapping.
`org.hibernate.validator.cfg`, `org.hibernate.validator.cfg.context`, `org.hibernate.validator.cfg.defs`	With Hibernate Validator you can define constraints via a fluent API. These packages contain all classes needed to use this feature. In the package `org.hibernate.validator.cfg` you will find the `ConstraintMapping` class and in

Packages	Description
	package org.hibernate.validator.cfg.defs all constraint definitions. For more details see 第 8.4 节 “Programmatic constraint definition” .
org.hibernate.validator.constraints, org.hibernate.validator.constraints.br	In addition to Bean Validation constraints, Hibernate Validator provides some useful custom constraints. These packages contain all custom annotation classes. For more details see 第 2.4.2 节 “Additional constraints” .
org.hibernate.validator.group, org.hibernate.validator.spi.group	With Hibernate Validator you can define dynamic default group sequences in function of the validated object state. These packages contain all classes needed to use this feature (GroupSequenceProvider annotation and DefaultGroupSequenceProvider contract). For more details see 第 2.3.2 节 “对一个类重定义其默认校验组” .
org.hibernate.validator.messageinterpolation, org.hibernate.validator.resourceloading, org.hibernate.validator.spi.resourceloading	These packages contain the classes related to constraint message interpolation. The first package contains two implementations of MessageInterpolator. The first one, ValueFormatterMessageInterpolator allows to interpolate the validated value into the constraint message, see 第 5.3 节 “MessageInterpolator” . The second implementation named ResourceBundleMessageInterpolator is the implementation used by default by Hibernate Validator. This implementation relies on a ResourceBundleLocator, see 第 5.3.1 节 “ResourceBundleLocator” . Hibernate Validator provides different ResourceBundleLocator implementations located in the package org.hibernate.validator.resourceloading.
org.hibernate.validator.method, org.hibernate.validator.method.metadata	Hibernate Validator provides support for method-level constraints based on appendix C of the Bean Validation specification. The first package contains the MethodValidator interface allowing you to validate method return values and

Packages	Description
	parameters. The second package contains meta data for constraints hosted on parameters and methods which can be retrieved via the `MethodValidator`.

注意

The public packages of Hibernate Validator fall into two categories: while the actual API parts are intended to be invoked or used by clients (e.g. the API for programmatic constraint declaration or the custom constraints), the SPI (service provider interface) packages contain interfaces which are intended to be implemented by clients (e.g. `ResourceBundleLocator`).

8.2. Fail fast mode

First off, the fail fast mode. Hibernate Validator allows to return from the current validation as soon as the first constraint violation occurs. This is called the fail fast mode and can be useful for validation of large object graphs where one is only interested whether there is a constraint violation or not. 例 8.1 “Enabling failFast via a property” , 例 8.2 “Enabling failFast at the Configuration level” and 例 8.3 “Enabling failFast at the ValidatorFactory level” show multiple ways to enable the fail fast mode.

例 8.1. Enabling `failFast` via a property

```
HibernateValidatorConfiguration                         configuration                         =
 Validation.byProvider( HibernateValidator.class ).configure();
ValidatorFactory   factory   =   configuration.addProperty(   "hibernate.validator.fail_fast",
 "true" ).buildValidatorFactory();
Validator validator = factory.getValidator();

// do some actual fail fast validation
...
```

例 8.2. Enabling `failFast` at the `Configuration` level

```
HibernateValidatorConfiguration                         configuration                         =
 Validation.byProvider( HibernateValidator.class ).configure();
ValidatorFactory factory = configuration.failFast( true ).buildValidatorFactory();
Validator validator = factory.getValidator();

// do some actual fail fast validation
...
```

例 8.3. Enabling `failFast` at the `ValidatorFactory` level

```
HibernateValidatorConfiguration                         configuration                        =
 Validation.byProvider( HibernateValidator.class ).configure();
ValidatorFactory factory = configuration.buildValidatorFactory();

Validator validator = factory.getValidator();

// do some non fail fast validation
...

validator = factory.unwrap( HibernateValidatorFactory.class )
            .usingContext()
            .failFast( true )
            .getValidator();

// do fail fast validation
...
```

8.3. Method validation

The Bean Validation API allows to specify constraints for fields, properties and types. Hibernate Validator goes one step further and allows to place contraint annotations also on method parameters and method return values, thus enabling a programming style known as "Programming by Contract".

More specifically this means that Bean Validation constraints can be used to specify

- the preconditions that must be met before a method invocation (by annotating method parameters with constraints) and
- the postconditions that are guaranteed after a method invocation (by annotating methods)

This approach has several advantages over traditional ways of parameter and return value checking:

- The checks don't have to be performed manually (e.g. by throwing `IllegalArgumentException`s or similar), resulting in less code to write and maintain.
- A method's pre- and postconditions don't have to be expressed again in the method's JavaDoc, since the constraint annotations will automatically be included in the generated JavaDoc. This avoids redundancy and reduces the chance of inconsistencies between implementation and documentation.

注意

Method validation was also considered to be included in the Bean Validation API as defined by JSR 303, but it didn't become part of the 1.0 version.

A basic draft is outlined in appendix C of the specification, and the implementation in Hibernate Validator is largely influenced by this draft. The feature is considered again for inclusion in BV 1.1.

8.3.1. Defining method-level constraints

例 8.4 “Using method-level constraints” demonstrates the definition of method-level constraints.

例 8.4. Using method-level constraints

```
public class RentalStation {

    @NotNull
     public Car rentCar(@NotNull Customer customer, @NotNull @Future Date startDate, @Min(1)
 int durationInDays) {
        //...
    }
}
```

Here the following pre- and postconditions for the `rentCar()` method are declared:

- The renting customer may not be null
- The rental's start date must not be null and must be in the future
- The rental duration must be at least one day
- The returned `Car` instance may not be null

Using the `@Valid` annotation it's also possible to define that a cascaded validation of parameter or return value objects shall be performed. An example can be found in 例 8.5 “Cascaded validation of method-level constraints” .

例 8.5. Cascaded validation of method-level constraints

```
public class RentalStation {

    @Valid
    public Set<Rental> getRentalsByCustomer(@Valid Customer customer) {
        //...
    }
}
```

Here all the constraints declared at the `Customer` type will be evaluated when validating the method parameter and all constraints declared at the returned `Rental` objects will be evaluated when validating the method's return value.

8.3.1.1. Using method constraints in type hierarchies

Special care must be taken when defining parameter constraints in inheritance hierarchies.

When a method is overridden in sub-types method parameter constraints can only be declared at the base type. The reason for this restriction is that the preconditions to be fulfilled by a type's client must not be strengthened in sub-types (which may not even be known to the base type's client). Note that also if the base method doesn't declare any parameter constraints at all, no parameter constraints may be added in overriding methods.

The same restriction applies to interface methods: no parameter constraints may be defined at the implementing method (or the same method declared in sub-interfaces).

If a violation of this rule is detected by the validation engine, a `javax.validation.ConstraintDeclarationException` will be thrown. In 例 8.6 "Illegal parameter constraint declarations" some examples for illegal parameter constraints declarations are shown.

例 8.6. Illegal parameter constraint declarations

```
public class Car {

    public void drive(Person driver) { ... }

}

public class RentalCar extends Car {

    //not allowed, parameter constraint added in overriding method
    public void drive(@NotNull Person driver) { ... }

}

public interface ICar {

    void drive(Person driver);

}

public class CarImpl implements ICar {

    //not allowed, parameter constraint added in implementation of interface method
    public void drive(@NotNull Person driver) { ... }

}
```

This rule only applies to parameter constraints, return value constraints may be added in sub-types without any restrictions as it is alright to strengthen the postconditions guaranteed to a type's client.

8.3.2. Evaluating method-level constraints

To validate method-level constraints Hibernate Validator provides the interface `org.hibernate.validator.method.MethodValidator`.

As shown in 例 8.7 "The MethodValidator interface" this interface defines methods for the evaluation of parameter as well as return value constraints and for retrieving an extended type descriptor providing method constraint related meta data.

例 8.7. The `MethodValidator` interface

```
public interface MethodValidator {

     <T> Set<MethodConstraintViolation<T>> validateParameter(T object, Method method, Object
 parameterValue, int parameterIndex, Class<?>... groups);

   <T> Set<MethodConstraintViolation<T>> validateAllParameters(T object, Method method, Object[]
 parameterValues, Class<?>... groups);

    <T> Set<MethodConstraintViolation<T>> validateReturnValue(T object, Method method, Object
 returnValue, Class<?>... groups);

    TypeDescriptor getConstraintsForType(Class<?> clazz);
}
```

To retrieve a method validator get hold of an instance of HV's `javax.validation.Validator` implementation and unwrap it to `MethodValidator` as shown in 例 8.8 "Retrieving a MethodValidator instance".

例 8.8. Retrieving a `MethodValidator` instance

```
MethodValidator methodValidator = Validation.byProvider( HibernateValidator.class )
    .configure()
    .buildValidatorFactory()
    .getValidator()
    .unwrap( MethodValidator.class );
```

The validation methods defined on `MethodValidator` each return a `Set<MethodConstraintViolation>`. The type `MethodConstraintViolation` (see 例 8.9 "The MethodConstraintViolation type") extends `javax.validation.ConstraintViolation` and provides additional method level validation specific information such as the method and index of the parameter which caused the constraint violation.

例 8.9. The `MethodConstraintViolation` type

```
public interface MethodConstraintViolation<T> extends ConstraintViolation<T> {
    public static enum Kind { PARAMETER, RETURN_VALUE }

    Method getMethod();

    Integer getParameterIndex();

    String getParameterName();

    Kind getKind();
}
```

注意

The method `getParameterName()` currently returns synthetic parameter identifiers such as "arg0", "arg1" etc. In a future version of Hibernate Validator support for specifying parameter identifiers might be added.

Typically the validation of method-level constraints is not invoked manually but automatically upon method invocation by an integration layer using AOP (aspect-oriented programming) or similar method interception facilities such as the JDK's `java.lang.reflect.Proxy` API or CDI ("JSR 299: Contexts and Dependency Injection for the Java^{TM} EE platform").

If a parameter or return value constraint can't be validated sucessfully such an integration layer typically will throw a `MethodConstraintViolationException` which similar to `javax.validation.ConstraintViolationException` contains a set with the occurred constraint violations.

提示

If you are using CDI you might be interested in the Seam Validation [http://seamframework.org/Seam3/ValidationModule] project. This Seam module provides an interceptor which integrates the method validation functionality with CDI.

8.3.3. Retrieving method-level constraint meta data

As outlined in 第 6 章 Metadata API the Bean Validation API provides rich capabilities for retrieving constraint related meta data. Hibernate Validator extends this API and allows to retrieve constraint meta data also for method-level constraints.

例 8.10 "Retrieving meta data for method-level constraints" shows how to use this extended API to retrieve constraint meta data for the `rentCar()` method from the `RentalStation` type.

例 8.10. Retrieving meta data for method-level constraints

```
TypeDescriptor typeDescriptor = methodValidator.getConstraintsForType(RentalStation.class)

//retrieve a descriptor for the rentCar() method
MethodDescriptor     rentCarMethod     =     typeDescriptor.getConstraintsForMethod("rentCar",
 Customer.class, Date.class, int.class);
assertEquals(rentCarMethod.getMethodName(), "rentCar");
assertTrue(rentCarMethod.hasConstraints());
assertFalse(rentCarMethod.isCascaded());

//retrieve constraints from the return value
Set<ConstraintDescriptor<?>>                    returnValueConstraints                    =
 rentCarMethod.findConstraints().getConstraintDescriptors();
assertEquals(returnValueConstraints.size(), 1);
assertEquals(returnValueConstraints.iterator().next().getAnnotation().annotationType(),
 NotNull.class);

List<ParameterDescriptor> allParameters = rentCarMethod.getParameterDescriptors();
assertEquals(allParameters.size(), 3);

//retrieve a descriptor for the startDate parameter
ParameterDescriptor startDateParameter = allParameters.get(1);
assertEquals(startDateParameter.getIndex(), 1);
assertFalse(startDateParameter.isCascaded());
assertEquals(startDateParameter.findConstraints().getConstraintDescriptors().size(), 2);
```

Refer to the JavaDoc [http://docs.jboss.org/hibernate/validator/4.2/api/index.html?org/hibernate/validator/method/metadata/package-summary.html] of the package `org.hibernate.validator.method.metadata` for more details on the extended meta data API.

8.4. Programmatic constraint definition

Another addition to the Bean Validation specification is the ability to configure constraints via a fluent API. This API can be used exclusively or in combination with annotations and xml. If used in combination programmatic constraints are additive to constraints configured via the standard configuration capabilities.

The API is centered around the `ConstraintMapping` class which can be found in the package `org.hibernate.validator.cfg`. Starting with the instantiation of a new `ConstraintMapping`, constraints can be defined in a fluent manner as shown in 例 8.11 “Programmatic constraint definition” .

例 8.11. Programmatic constraint definition

```
ConstraintMapping mapping = new ConstraintMapping();
mapping.type( Car.class )
    .property( "manufacturer", FIELD )
        .constraint( new NotNullDef() )
    .property( "licensePlate", FIELD )
```

```
        .constraint( new NotNullDef() )
        .constraint( new SizeDef().min( 2 ).max( 14 ) )
    .property( "seatCount", FIELD )
        .constraint( new MinDef()value ( 2 ) )
.type( RentalCar.class )
    .property( "rentalStation", METHOD )
        .constraint( new NotNullDef() );
```

As you can see constraints can be configured on multiple classes and properties using method chaining. The constraint definition classes `NotNullDef`, `SizeDef` and `MinDef` are helper classes which allow to configure constraint parameters in a type-safe fashion. Definition classes exist for all built-in constraints in the `org.hibernate.validator.cfg.defs` package.

For custom constraints you can either create your own definition classes extending `ConstraintDef` or you can use `GenericConstraintDef` as seen in 例 8.12 “Programmatic constraint definition using createGeneric()” .

例 8.12. Programmatic constraint definition using `createGeneric()`

```
ConstraintMapping mapping = new ConstraintMapping();
mapping.type( Car.class )
    .property( "licensePlate", FIELD )
      .constraint( new GenericConstraintDef<CheckCase.class>( CheckCase.class ).param( "value",
 CaseMode.UPPER ) );
```

Not only standard class- and property-level constraints but also method constraints can be configured using the API. As shown in 例 8.13 “Programmatic definition of method constraints” methods are identified by their name and their parameters (if there are any). Having selected a method, constraints can be placed on the method's parameters and/or return value.

例 8.13. Programmatic definition of method constraints

```
ConstraintMapping mapping = new ConstraintMapping();
mapping.type( Car.class )
    .method( "drive", String.class, Integer.class )
        .parameter( 0 )
            .constraint( new NotNullDef() )
            .constraint( new MinDef().value ( 1 ) )
        .parameter( 1 )
            .constraint( new NotNullDef() )
        .returnValue()
            .constraint( new NotNullDef() )
    .method( "check" )
        .returnValue()
            .constraint( new NotNullDef() );
```

Using the API it's also possible to mark properties, method parameters and method return values as cascading (equivalent to annotating them with @Valid). An example can be found in 例 8.14 “Marking constraints for cascaded validation” .

例 8.14. Marking constraints for cascaded validation

```
ConstraintMapping mapping = new ConstraintMapping();
mapping.type( Car.class )
    .property( "manufacturer", FIELD )
        .valid()
    .property( "licensePlate", METHOD )
        .valid()
    .method( "drive", String.class, Integer.class )
        .parameter( 0 )
            .valid()
        .parameter( 1 )
            .valid()
        .returnValue()
            .valid()
.type( RentalCar.class )
    .property( "rentalStation", METHOD )
        .valid();
```

Last but not least you can configure the default group sequence or the default group sequence provider of a type as shown in 例 8.15 “Configuration of default group sequence and default group sequence provider” .

例 8.15. Configuration of default group sequence and default group sequence provider

```
ConstraintMapping mapping = new ConstraintMapping();
mapping.type( Car.class )
    .defaultGroupSequence( Car.class, CarChecks.class )
.type( RentalCar.class )
    .defaultGroupSequenceProvider( RentalCarGroupSequenceProvider.class );
```

Once a ConstraintMapping is set up it has to be passed to the configuration. Since the programmatic API is not part of the official Bean Validation specification you need to get hold of a HibernateValidatorConfiguration instance as shown in 例 8.16 “Creating a Hibernate Validator specific configuration” .

例 8.16. Creating a Hibernate Validator specific configuration

```
ConstraintMapping mapping = new ConstraintMapping();
// configure mapping instance

HibernateValidatorConfiguration                            config                            =
 Validation.byProvider( HibernateValidator.class ).configure();
```

```
config.addMapping( mapping );
ValidatorFactory factory = config.buildValidatorFactory();
Validator validator = factory.getValidator();
```

8.5. Boolean composition for constraint composition

As per Bean Validation specification the constraints of a composed constraint (see 第 3.2 节 "约束条件组合") are all combined via a logical AND. This means all of the composing constraints need to return `true` in order for an overall successful validation. Hibernate Validator offers an extension to this logical AND combination which allows you to compose constraints via a logical OR or NOT. To do so you have to use the `ConstraintComposition` annotation and the enum `CompositionType` with its values AND, OR and ALL_FALSE. 例 8.17 "OR composition of constraints" shows how to build a composing constraint where only one of the constraints has to be successful in order to pass the validation. Either the validated string is all lowercased or it is between two and three characters long.

例 8.17. OR composition of constraints

```
@ConstraintComposition(OR)
@Pattern(regexp = "[a-z]")
@Size(min = 2, max = 3)
@ReportAsSingleViolation
@Target({ METHOD, FIELD })
@Retention(RUNTIME)
@Constraint(validatedBy = { })
public @interface PatternOrSize {
   public abstract String message() default "{PatternOrSize.message}";
   public abstract Class<?>[] groups() default { };
   public abstract Class<? extends Payload>[] payload() default { };
}
```

提示

Using ALL_FALSE as composition type implicitly enforces that only a single violation will get reported in case validation of the constraint composition fails.

Annotation Processor

你碰到过下面这些让人抓狂的情况么:

- specifying constraint annotations at unsupported data types (e.g. by annotating a String with @Past)
- 对一个JavaBean的setter方法进行标注(而不是getter)
- 对一个静态的变量或者方法进行约束条件标注(这样是不支持滴)

这样的话, 你就应该看看Hibernate Validator 的约束处理器了. 它会被插入到编译过程中, 然后如果发现如果哪个约束标注用错了的话, 则汇报编译错误.

注意

You can find the Hibernate Validator Annotation Processor as part of the distribution bundle on Sourceforge [http://sourceforge.net/projects/hibernate/files/hibernate-validator] or in the JBoss Maven Repository (see 例 1.1 "Configuring the JBoss Maven repository") under the GAV org.hibernate:hibernate-validator-annotation-processor.

9.1. 前提条件

The Hibernate Validator Annotation Processor is based on the "Pluggable Annotation Processing API" as defined by JSR 269 [http://jcp.org/en/jsr/detail?id=269] which is part of the Java Platform since Java 6.

9.2. 特性

As of Hibernate Validator 5.0.0.CR2 the Hibernate Validator Annotation Processor checks that:

- 应用了约束标注的属性的类型是否被该约束所支持
- only non-static fields or methods are annotated with constraint annotations
- only non-primitive fields or methods are annotated with @Valid
- only such methods are annotated with constraint annotations which are valid JavaBeans getter methods (optionally, see below)
- only such annotation types are annotated with constraint annotations which are constraint annotations themselves
- definition of dynamic default group sequence with @GroupSequenceProvider is valid

9.3. 配置项

Hibernate Validator标注处理器的行为可以通过表 9.1 “Hibernate Validator 标注处理器配置项”中列出的处理器配置项 [http://java.sun.com/javase/6/docs/technotes/tools/windows/javac.html#options]加以控制.

表 9.1. Hibernate Validator 标注处理器配置项

配置项	功能
diagnosticKind	控制编译错误级别. 必须是枚举类型javax.tools.Diagnostic.Kind中的值(字符串形式), 例如WARNING. 如果是ERROR的话, 那么如果API检测到约束信息应用的错误的话, 会让编译过程终止, 默认是ERROR.
methodConstraintsSupported	Controls whether constraints are allowed at methods of any kind. Must be set to true when working with method level constraints as supported by Hibernate Validator. Can be set to false to allow constraints only at JavaBeans getter methods as defined by the Bean Validation API. Defaults to true.
verbose	Controls whether detailed processing information shall be displayed or not, useful for debugging purposes. Must be either true or false. Defaults to false.

9.4. 使用标注处理器

本小节详细介绍了如何把Hibernate Validator标注处理器与命令行编译(javac, Ant, Maven)以及IDE (Eclipse, IntelliJ IDEA, NetBeans)集成.

9.4.1. 命令行编译

9.4.1.1. javac

When compiling on the command line using javac [http://java.sun.com/javase/6/docs/technotes/guides/javac/index.html], specify the JAR hibernate-validator-annotation-processor-5.0.0.CR2.jar using the "processorpath" option as shown in the following listing. The processor will be detected automatically by the compiler and invoked during compilation.

例 9.1. 在javac中使用标注处理器

```
javac src/main/java/org/hibernate/validator/ap/demo/Car.java \
   -cp /path/to/validation-api-1.0.0.GA.jar \
   -processorpath /path/to/hibernate-validator-annotation-processor-5.0.0.CR2.jar
```

9.4.1.2. Apache Ant

和直接使用javac差不多，可以在Apache Ant [http://ant.apache.org/]的javac task [http://ant.apache.org/manual/CoreTasks/javac.html]中添加上面例子中的参数:

例 9.2. 在Ant中使用标注处理器

```
<javac srcdir="src/main"
       destdir="build/classes"
       classpath="/path/to/validation-api-1.0.0.GA.jar">
       <compilerarg value="-processorpath" />
       <compilerarg value="/path/to/hibernate-validator-annotation-processor-5.0.0.CR2.jar"/>
</javac>
```

9.4.1.3. Maven

对于和Apache Maven [http://maven.apache.org/]集成来说我们有很多选择，通常，我们可以把Hibenrate Validator标注处理器作为依赖添加到你的项目当中:

例 9.3. 添加HV 标注处理器为依赖

```
...
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-validator-annotation-processor</artifactId>
    <version>5.0.0.CR2</version>
    <scope>compile</scope>
</dependency>
...
```

这样， 这个处理器就能够自动的被编译器所调用. 虽然基本上能工作,但是还是有一些缺点， 在某些情况下， 标注处理器的输出信息可能不能够被显示出来. (请参考MCOMPILER-66 [http://jira.codehaus.org/browse/MCOMPILER-66]).

另外的一个选择是使用Maven Annotation Plugin [http://code.google.com/p/maven-annotation-plugin/]. 不过在此文档撰写的时候，这个插件还没有被上传到任何一个广泛被使用的仓库中. 所以，你需要自己把这个插件自己的仓库添加到你的settings.xml 或 pom.xml中:

例 9.4. 添加Maven Annotation Plugin的仓库

```
...
<pluginRepositories>
    <pluginRepository>
        <id>maven-annotation-plugin-repo</id>
        <url>http://maven-annotation-plugin.googlecode.com/svn/trunk/mavenrepo</url>
    </pluginRepository>
</pluginRepositories>
...
```

现在， 禁用compiler插件所调用的标准的标注处理过程， 然后再通过定义一个execution来配置annotation plugin的运行，还需要把Hibernate Validator标注处理器作为该插件的依赖添加进去(这样，此标注处理器就不会被当成你的项目的依赖而出现在类路径中了)：

例 9.5. 配置Maven Annotation Plugin

```
...
<plugin>
    <artifactId>maven-compiler-plugin</artifactId>
    <configuration>
        <source>1.6</source>
        <target>1.6</target>
        <compilerArgument>-proc:none</compilerArgument>
    </configuration>
</plugin>
<plugin>
    <groupId>org.bsc.maven</groupId>
    <artifactId>maven-processor-plugin</artifactId>
    <version>1.3.4</version>
    <executions>
        <execution>
            <id>process</id>
            <goals>
                <goal>process</goal>
            </goals>
            <phase>process-sources</phase>
        </execution>
    </executions>
    <dependencies>
        <dependency>
            <groupId>org.hibernate</groupId>
            <artifactId>hibernate-validator-annotation-processor</artifactId>
            <version>5.0.0.CR2</version>
            <scope>compile</scope>
        </dependency>
    </dependencies>
</plugin>
...
```

9.4.2. IDE集成

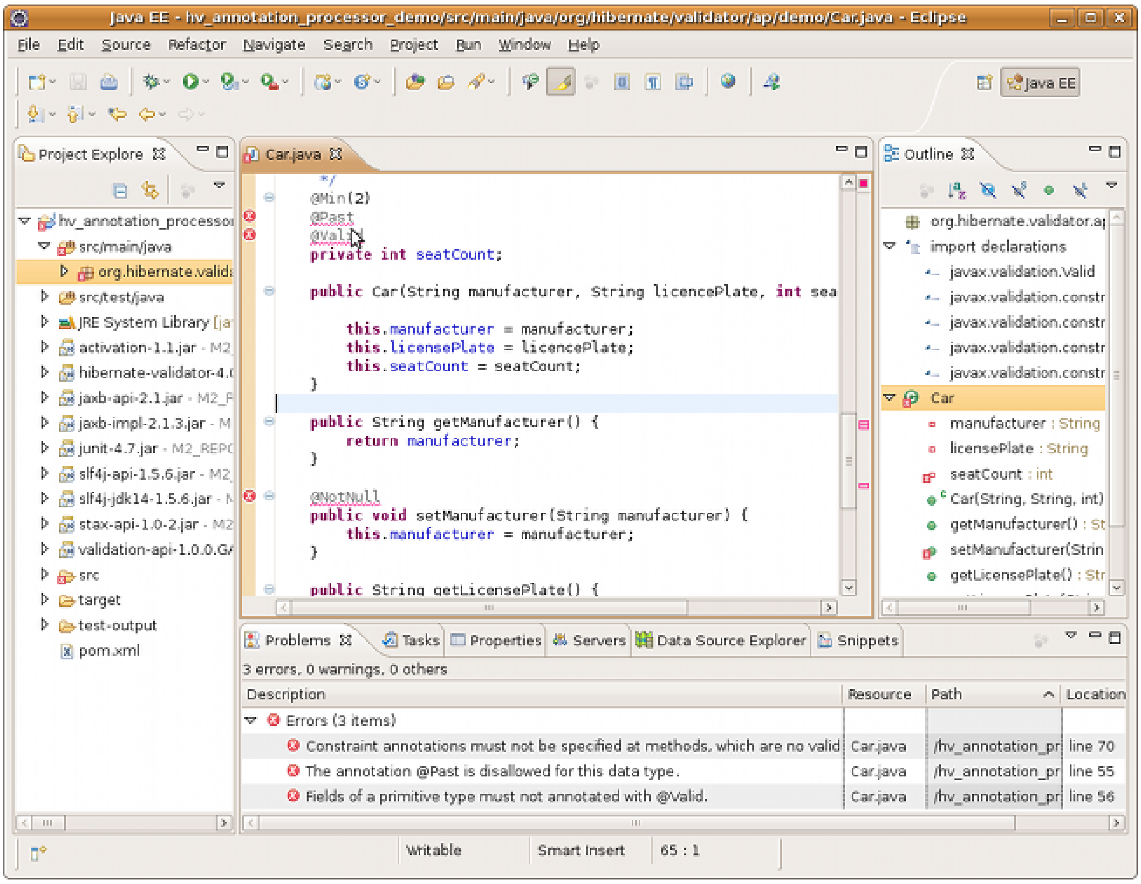
9.4.2.1. Eclipse

请参考以下步骤来在Eclipse [http://www.eclipse.org/]中使用标注处理器:

- 右键点击你的项目，然后选择"属性"

- 在"Java Compiler"页面确认"编译级别"设置的是"1.6". 否则的话是无法使用标注处理器的.
- 到"Java Compiler"下面的"Annotation Processing" 页面，然后选择"启用标注处理"(译注: 我的电脑是英文版的，所以真的不知道中文版的eclipse上，这些翻译成了什么:(
- Go to "Java Compiler - Annotation Processing - Factory Path" and add the JAR hibernate-validator-annotation-processor-5.0.0.CR2.jar
- 确认工作空间重新编译

现在你应该能够看到所有的标注错误都在编辑窗口中显示出了错误标记,也都显示在了"问题"视图:


Java EE - hv_annotation_processor_demo/src/main/java/org/hibernate/validator/ap/demo/Car.java - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help
Java EE
Project Explore
hv_annotation_processo
src/main/java
org.hibernate.valid
src/test/java
JRE System Library [ja
activation-1.1.jar - M2
hibernate-validator-4.
jaxb-api-2.1.jar - M2_R
jaxb-impl-2.1.3.jar - M
junit-4.7.jar - M2_REPC
slf4j-api-1.5.6.jar - M2
slf4j-jdk14-1.5.6.jar - M
stax-api-1.0-2.jar - M2
validation-api-1.0.0.GA
src
target
test-output
pom.xml
Car.java
Outline
org.hibernate.validator.a
import declarations
javax.validation.Valid
javax.validation.constr
javax.validation.constr
javax.validation.constr
javax.validation.constr
Car
manufacturer : String
licensePlate : String
seatCount : int
Car(String, String, int)
getManufacturer() : St
setManufacturer(Strin
getLicensePlate() : Str
Problems
Tasks
Properties
Servers
Data Source Explorer
Snippets
Writable
Smart Insert
65 : 1

```
    */
    @Min(2)
    @Past
    @Valid
    private int seatCount;

    public Car(String manufacturer, String licencePlate, int sea

        this.manufacturer = manufacturer;
        this.licensePlate = licencePlate;
        this.seatCount = seatCount;
    }

    public String getManufacturer() {
        return manufacturer;
    }

    @NotNull
    public void setManufacturer(String manufacturer) {
        this.manufacturer = manufacturer;
    }

    public String getLicensePlate() {
```

3 errors, 0 warnings, 0 others

Description	Resource	Path	Location
Errors (3 items)			
Constraint annotations must not be specified at methods, which are no valid	Car.java	/hv_annotation_pr	line 70
The annotation @Past is disallowed for this data type.	Car.java	/hv_annotation_pr	line 55
Fields of a primitive type must not annotated with @Valid.	Car.java	/hv_annotation_pr	line 56

9.4.2.2. IntelliJ IDEA

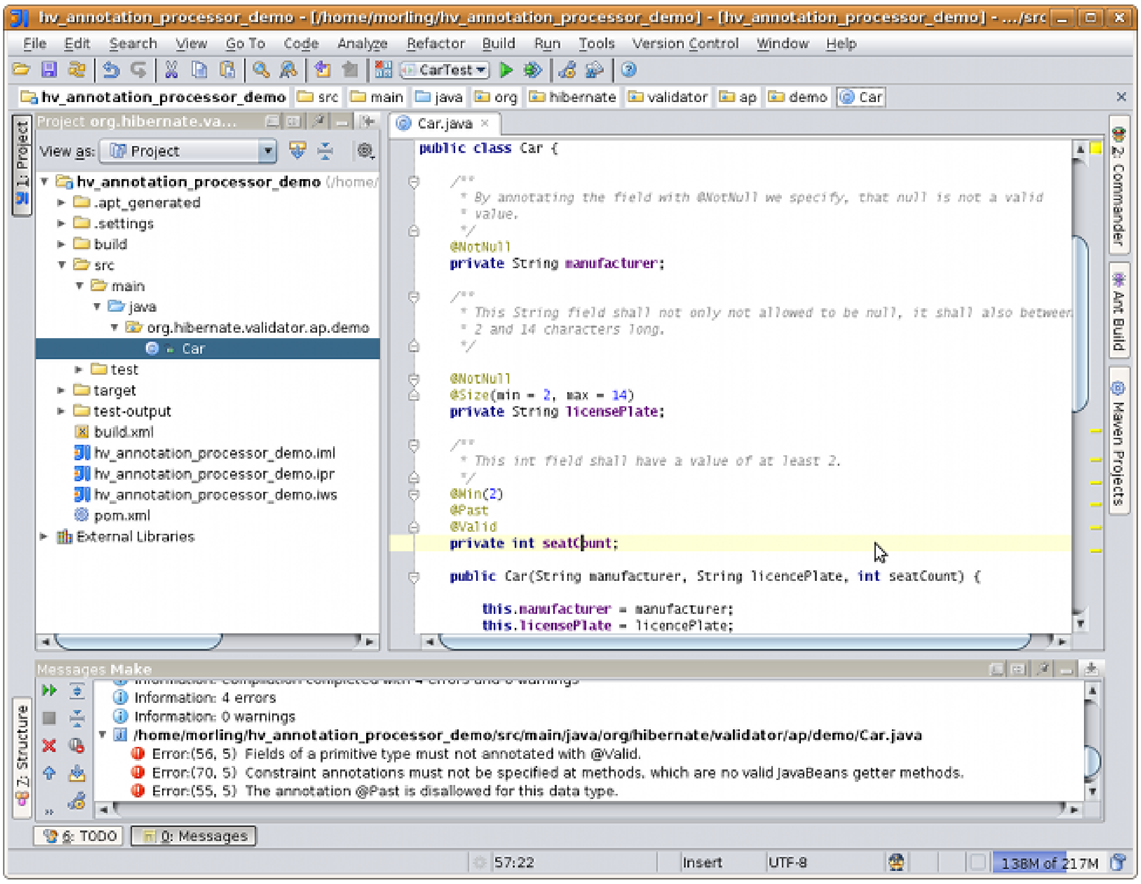
请参考以下步骤来在IntelliJ IDEA [http://www.jetbrains.com/idea/] (9.0及以上):中使用标注处理器:

- 选择 "File", 然后 "Settings",
- 展开"Compiler"节点, 然后点击"Annotation Processors"
- Choose "Enable annotation processing" and enter the following as "Processor path": /path/to/hibernate-validator-annotation-processor-5.0.0.CR2.jar

- 添加处理器的全路径名称org.hibernate.validator.ap.ConstraintValidationProcessor到"Annotation Processors"列表

- 如果需要的话，添加你的模块到"Processed Modules"列表

重新编译你的项目，然后应该能看到关于约束标注的错误信息了：


hv_annotation_processor_demo - [/home/morling/hv_annotation_processor_demo] - [hv_annotation_processor_demo] - .../src
File Edit Search View Go To Code Analyze Refactor Build Run Tools Version Control Window Help
CarTest
hv_annotation_processor_demo src main java org hibernate validator ap demo Car
Project org.hibernate.va...
View as: Project
hv_annotation_processor_demo (/home/
.apt_generated
.settings
build
src
main
java
org.hibernate.validator.ap.demo
Car
test
target
test-output
build.xml
hv_annotation_processor_demo.iml
hv_annotation_processor_demo.ipr
hv_annotation_processor_demo.iws
pom.xml
External Libraries
Car.java
1: Project
2: Commander
Ant Build
Maven Projects
7: Structure
Messages Make
6: TODO
0: Messages
57:22
Insert
UTF-8
138M of 217M

```
public class Car {

    /**
     * By annotating the field with @NotNull we specify, that null is not a valid
     * value.
     */
    @NotNull
    private String manufacturer;

    /**
     * This String field shall not only not allowed to be null, it shall also between
     * 2 and 14 characters long.
     */

    @NotNull
    @Size(min = 2, max = 14)
    private String licensePlate;

    /**
     * This int field shall have a value of at least 2.
     */
    @Min(2)
    @Past
    @Valid
    private int seatCount;

    public Car(String manufacturer, String licencePlate, int seatCount) {

        this.manufacturer = manufacturer;
        this.licensePlate = licencePlate;
```

```
Information: 4 errors
Information: 0 warnings
/home/morling/hv_annotation_processor_demo/src/main/java/org/hibernate/validator/ap/demo/Car.java
    Error:(56, 5)  Fields of a primitive type must not annotated with @Valid.
    Error:(70, 5)  Constraint annotations must not be specified at methods, which are no valid JavaBeans getter methods.
    Error:(55, 5)  The annotation @Past is disallowed for this data type.
```

9.4.2.3. NetBeans

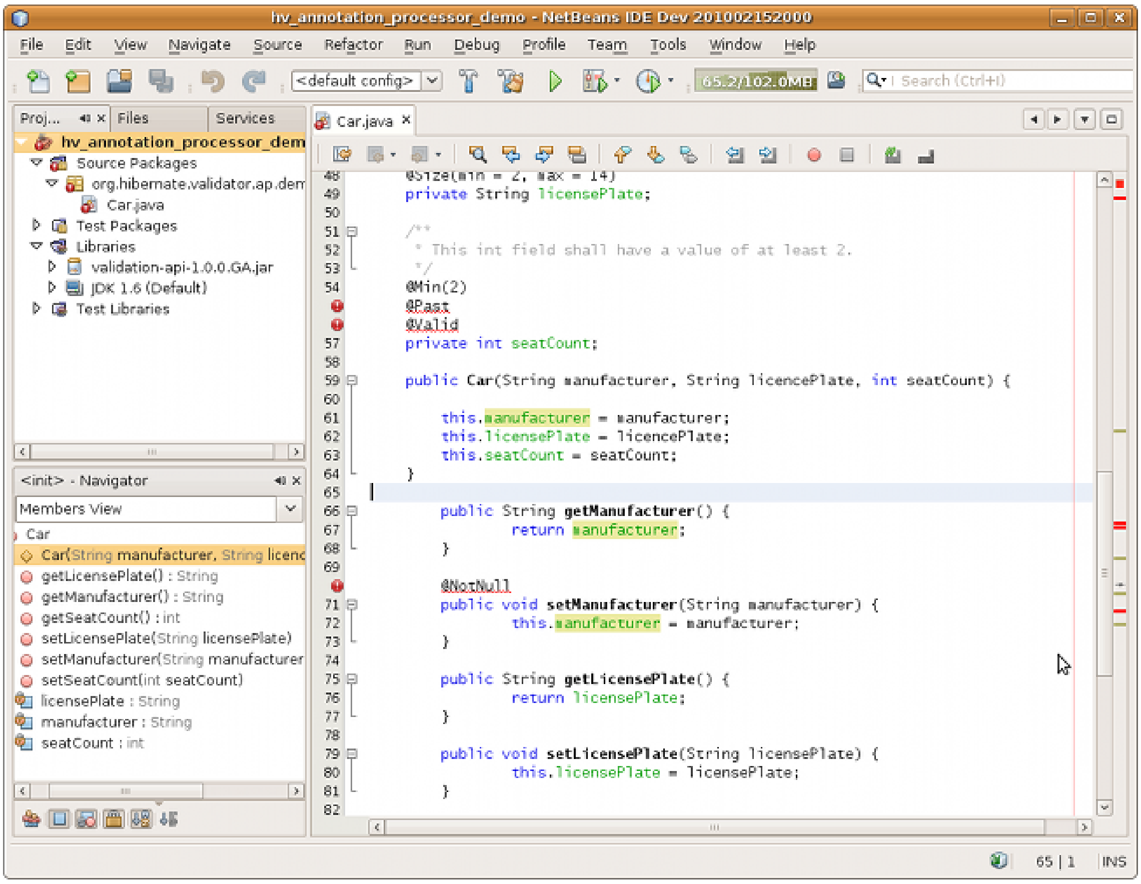
从6.9这个版本开始，NetBeans [http://www.netbeans.org/]也支持标注处理了．可以通过下面的步骤来启用它：

- 右键点击你的项目，然后选择"属性"

- Go to "Libraries", tab "Processor", and add the JAR hibernate-validator-annotation-processor-5.0.0.CR2.jar

- 到"Build - Compiling"页面选中"Enable Annotation Processing" 和 "Enable Annotation Processing in Editor"，并且指定标注处理器的全路径名称org.hibernate.validator.ap.ConstraintValidationProcessor.

所有的约束标注问题应该都会在编辑器里面直接被标记出来了：



hv_annotation_processor_demo - NetBeans IDE Dev 201002152000
File Edit View Navigate Source Refactor Run Debug Profile Team Tools Window Help
<default config>
65.2/102.0MB
Search (Ctrl+I)
Proj... Files Services
hv_annotation_processor_dem
Source Packages
org.hibernate.validator.ap.dem
Car.java
Test Packages
Libraries
validation-api-1.0.0.GA.jar
JDK 1.6 (Default)
Test Libraries
<init> - Navigator
Members View
Car
Car(String manufacturer, String licenc
getLicensePlate() : String
getManufacturer() : String
getSeatCount() : int
setLicensePlate(String licensePlate)
setManufacturer(String manufacturer
setSeatCount(int seatCount)
licensePlate : String
manufacturer : String
seatCount : int
Car.java
65 | 1
INS

```java
    @Size(min = 2, max = 14)
    private String licensePlate;

    /**
     * This int field shall have a value of at least 2.
     */
    @Min(2)
    @Past
    @Valid
    private int seatCount;

    public Car(String manufacturer, String licencePlate, int seatCount) {

        this.manufacturer = manufacturer;
        this.licensePlate = licencePlate;
        this.seatCount = seatCount;
    }

        public String getManufacturer() {
                return manufacturer;
        }

        @NotNull
        public void setManufacturer(String manufacturer) {
                this.manufacturer = manufacturer;
        }

        public String getLicensePlate() {
                return licensePlate;
        }

        public void setLicensePlate(String licensePlate) {
                this.licensePlate = licensePlate;
        }

```

9.5. 已知问题

以下是截止到2010年五月我们发现(但尚未解决)的问题:

- HV-308 [http://opensource.atlassian.com/projects/hibernate/browse/HV-308]: Additional validators registered for a constraint using XML [http://docs.jboss.org/hibernate/stable/validator/reference/en/html_single/#d0e1957] are not evaluated by the annotation processor.
- 有时候, 在eclipse里面自定义的约束条件不能够被正确的检查 [http://opensource.atlassian.com/projects/hibernate/browse/HV-293]. 清理这个项目可能会有帮助. 这可能是因为Eclipse中对 JSR 269 API的实现有问题, 但是还需要进一步的研究.
- When using the processor within Eclipse, the check of dynamic default group sequence definitions doesn't work. After further investigation, it seems to be an issue with the Eclipse JSR 269 API implementation.

进一步阅读

Last but not least, a few pointers to further information.

A great source for examples is the Bean Validation TCK which is available for anonymous access on GitHub [https://github.com/beanvalidation/beanvalidation-tck]. In particular the TCK's tests [https://github.com/beanvalidation/beanvalidation-tck/tree/master/src/main/java/org/hibernate/beanvalidation/tck/tests] might be of interest. The JSR 303 [http://jcp.org/en/jsr/detail?id=303] specification itself is also a great way to deepen your understanding of Bean Validation resp. Hibernate Validator.

如果你还有什么关于Hibernate Validator的问题或者想和我们分享你的使用经验,请使用Hibernate Validator Wiki [http://community.jboss.org/en/hibernate/validator]或者去Hibernate Validator Forum [https://forum.hibernate.org/viewforum.php?f=9]发帖子(译注:但是不要灌水:-D).

如果你发现了Hibernate Validator的bug,请在Hibernate's Jira [http://opensource.atlassian.com/projects/hibernate/browse/HV]中报告，我们非常欢迎您的反馈!