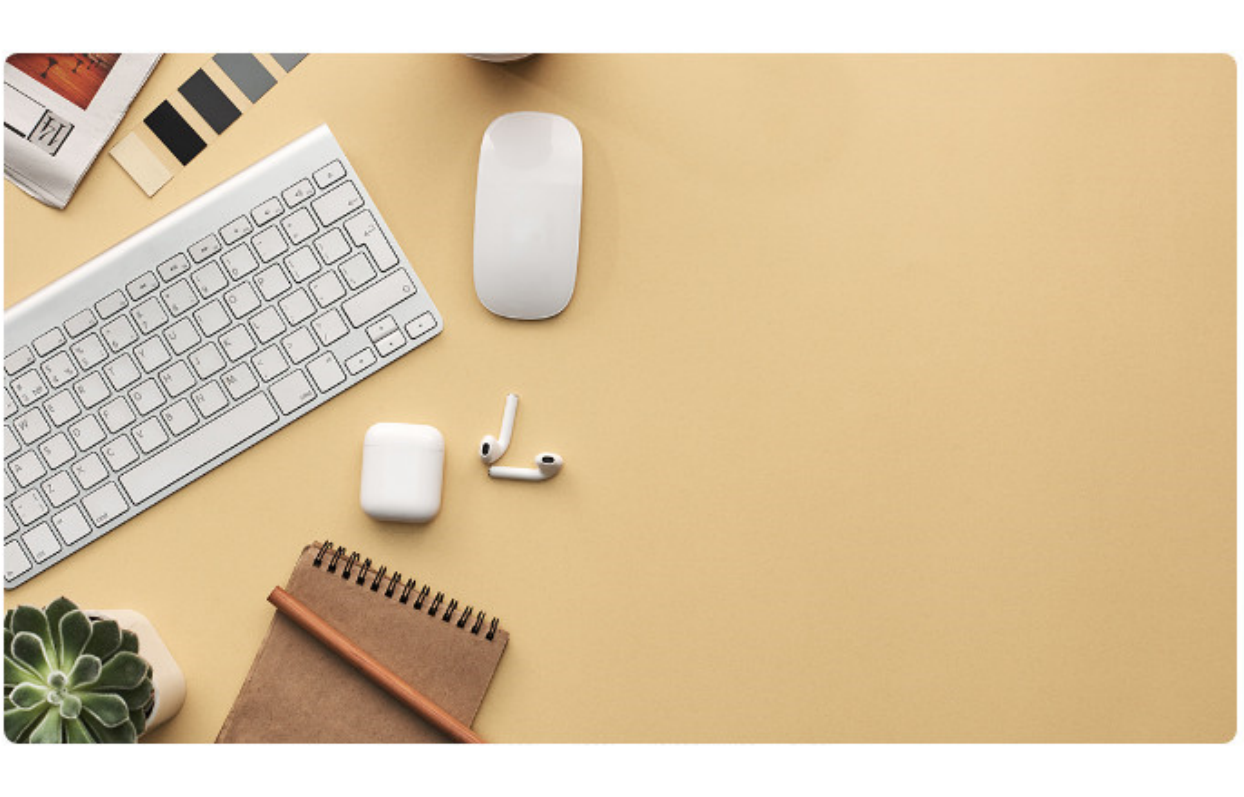
12 | Spring Web 参数验证常见错误

傅健 2021-05-19



讲述：傅健 大小：10.87M 时长：11:51

你好，我是傅健，这节课我们来聊聊 Spring Web 开发中的参数检验（Validation）。

参数检验是我们在 Web 编程时经常使用的技术之一，它帮助我们完成请求的合法性校验，可以有效拦截无效请求，从而达到节省系统资源、保护系统的目的。

相比较其他 Spring 技术，Spring 提供的参数检验功能具有独立性强、使用难度不高的特点。但是在实践中，我们仍然会犯一些常见的错误，这些错误虽然不会导致致命的后果，但是会影响我们的使用体验，例如非法操作要在业务处理时才被拒绝且返回的响应码不够清晰友好。而且这些错误不经测试很难发现，接下来我们就具体分析下这些常见错误案例及背后的原理。

案例 1：对象参数校验失效

在构建 Web 服务时，我们一般都会对一个 HTTP 请求的 Body 内容进行校验，例如我们来看这样一个案例及对应代码。

当开发一个学籍管理系统时，我们会提供了一个 API 接口去添加学生的相关信息，其对象定义参考下面的代码：

```
import lombok.Data;
import javax.validation.constraints.Size;
@Data
public class Student {
    @Size(max = 10)
    private String name;
    private short age;
}
```

这里我们使用了 @Size(max = 10) 给学生的姓名做了约束（最大为 10 字节），以拦截姓名过长、不符合“常情”的学生信息的添加。

定义完对象后，我们再定义一个 Controller 去使用它，使用方法如下：

```
import lombok.extern.slf4j.Slf4j;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;
@RestController
@Slf4j
@Validated
public class StudentController {
    @RequestMapping(path = "students", method = RequestMethod.POST)
    public void addStudent(@RequestBody Student student){
        log.info("add new student: {}", student.toString());
        //省略业务代码
    };
}
```

我们提供了一个支持学生信息添加的接口。启动服务后，使用 IDEA 自带的 HTTP Client 工具来发送下面的请求以添加一个学生，当然，这个学生的姓名会远超想象（即 this_is_my_name_which_is_too_long）：

```
POST http://localhost:8080/students
Content-Type: application/json
{
 "name": "this_is_my_name_which_is_too_long",
 "age": 10
}
```

很明显，发送这样的请求（name 超长）是期待 Spring Validation 能拦截它的，我们的预期响应如下（省略部分响应字段）：

```
HTTP/1.1 400
Content-Type: application/json

{
  "timestamp": "2021-01-03T00:47:23.994+0000",
  "status": 400,
  "error": "Bad Request",
  "errors": [
      "defaultMessage": "个数必须在 0 和 10 之间",
      "objectName": "student",
      "field": "name",
      "rejectedValue": "this_is_my_name_which_is_too_long",
      "bindingFailure": false,
      "code": "Size"
    }
  ],
  "message": "Validation failed for object='student'. Error count: 1",
  "path": "/students"
}
```

但是理想与现实往往有差距。实际测试会发现，使用上述代码构建的 Web 服务并没有做任何拦截。

案例解析

要找到这个问题的根源，我们就需要对 Spring Validation 有一定的了解。首先，我们来看下 RequestBody 接受对象校验发生的位置和条件。

假设我们构建 Web 服务使用的是 Spring Boot 技术，我们可以参考下面的时序图了解它的核心执行步骤：

如上图所示，当一个请求来临时，都会进入 DispatcherServlet，执行其 doDispatch()，此方法会根据 Path、Method 等关键信息定位到负责处理的 Controller 层方法（即 addStudent 方法），然后通过反射去执行这个方法，具体反射执行过程参考下面的代码（InvocableHandlerMethod#invokeForRequest）：

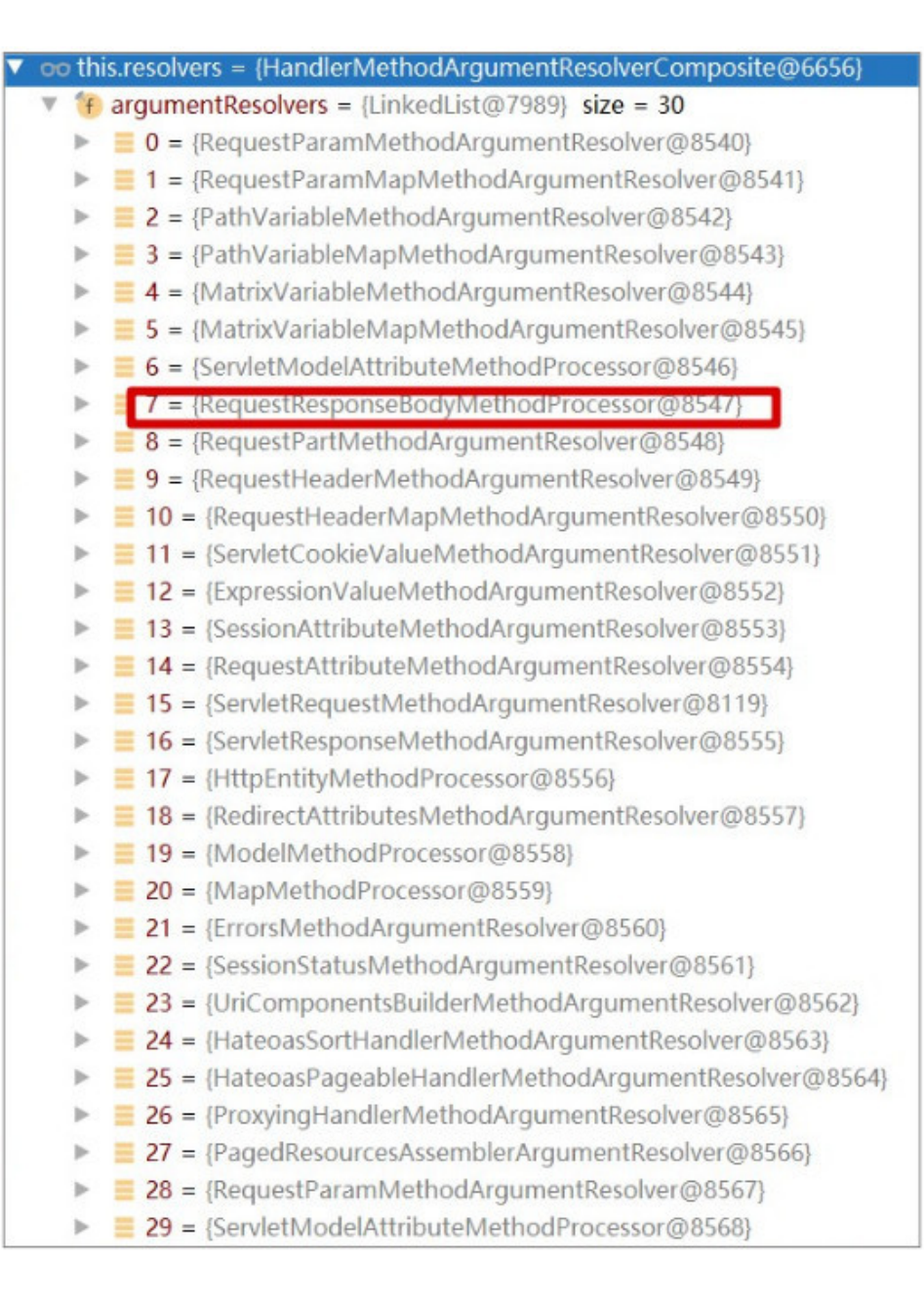
```
public Object invokeForRequest(NativeWebRequest request, @Nullable ModelAndVie
        Object... providedArgs) throws Exception {
    //根据请求内容和方法定义获取方法参数实例
    Object[] args = getMethodArgumentValues(request, mavContainer, providedArgs
    if (logger.isTraceEnabled()) {
        logger.trace("Arguments: " + Arrays.toString(args));
    }
    //携带方法参数实例去“反射”调用方法
    return doInvoke(args);
}
```

要使用 Java 反射去执行一个方法，需要先获取调用的参数，上述代码正好验证了这一点：getMethodArgumentValues() 负责获取方法执行参数，doInvoke() 负责使用这些获取到的参数去执行。

而具体到 getMethodArgumentValues() 如何获取方法调用参数，可以参考 addStudent 的方法定义，我们需要从当前的请求（NativeWebRequest ）中构建出 Student 这个方法参数的实例。

> public void addStudent(@RequestBody Student student)

那么如何构建出这个方法参数实例？Spring 内置了相当多的 HandlerMethodArgumentResolver，参考下图：



当试图构建出一个方法参数时，会遍历所有支持的解析器（Resolver）以找出适合的解析器，查找代码参考 HandlerMethodArgumentResolverComposite#getArgumentResolver：

```
@Nullable
private HandlerMethodArgumentResolver getArgumentResolver(MethodParameter para
   HandlerMethodArgumentResolver result = this.argumentResolverCache.get(param
   if (result == null) {
      //轮询所有的HandlerMethodArgumentResolver
      for (HandlerMethodArgumentResolver resolver : this.argumentResolvers) {
         //判断是否匹配当前HandlerMethodArgumentResolver
         if (resolver.supportsParameter(parameter)) {
            result = resolver;
            this.argumentResolverCache.put(parameter, result);
            break;
         }
      }
   }
   return result;
}
```

对于 student 参数而言，它被标记为 @RequestBody，当遍历到 RequestResponseBodyMethodProcessor 时就会匹配上。匹配代码参考其 RequestResponseBodyMethodProcessor 的 supportsParameter 方法：

```
@Override
public boolean supportsParameter(MethodParameter parameter) {
   return parameter.hasParameterAnnotation(RequestBody.class);
}
```

找到 Resolver 后，就会执行 HandlerMethodArgumentResolver#resolveArgument 方法。它首先会根据当前的请求（NativeWebRequest）组装出 Student 对象并对这个对象进行必要的校验，校验的执行参考 AbstractMessageConverterMethodArgumentResolver#validateIfApplicable：

```
protected void validateIfApplicable(WebDataBinder binder, MethodParameter para
   Annotation[] annotations = parameter.getParameterAnnotations();
   for (Annotation ann : annotations) {
      Validated validatedAnn = AnnotationUtils.getAnnotation(ann, Validated.cl
      //判断是否需要校验
      if (validatedAnn != null || ann.annotationType().getSimpleName().startsW
         Object hints = (validatedAnn != null ? validatedAnn.value() : Annotat
         Object[] validationHints = (hints instanceof Object[] ? (Object[]) hi
         //执行校验
         binder.validate(validationHints);
         break;
      }
   }
}
```

如上述代码所示，要对 student 实例进行校验（执行 binder.validate(validationHints) 方法），必须匹配下面两个条件的其中之一：

1. 标记了 org.springframework.validation.annotation.Validated 注解；
2. 标记了其他类型的注解，且注解名称以 Valid 关键字开头。

因此，结合案例程序，我们知道：student 方法参数并不符合这两个条件，所以即使它的内部成员添加了校验（即 @Size(max = 10)），也不能生效。

问题修正

针对这个案例，有了源码的剖析，我们就可以很快地找到解决方案。即对于 RequestBody 接受的对象参数而言，要启动 Validation，必须将对象参数标记上 @Validated 或者其他以 @Valid 关键字开头的注解，因此，我们可以采用对应的策略去修正问题。

1. 标记 @Validated

修正后关键代码行如下：

> public void addStudent(**@Validated** @RequestBody Student student)

2. 标记 @Valid 关键字开头的注解

这里我们可以直接使用熟识的 javax.validation.Valid 注解，它就是一种以 @Valid 关键字开头的注解，修正后关键代码行如下：

> public void addStudent(**@Valid** @RequestBody Student student)

另外，我们也可以自定义一个以 Valid 关键字开头的注解，定义如下：

```
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
@Retention(RetentionPolicy.RUNTIME)
public @interface ValidCustomized {
}
```

定义完成后，将它标记给 student 参数对象，关键代码行如下：

> public void addStudent(@ValidCustomized @RequestBody Student student)

通过上述 2 种策略、3 种具体修正方法，我们最终让参数校验生效且符合预期，不过需要提醒你的是：当使用第 3 种修正方法时，一定要注意自定义的注解要显式标记 @Retention(RetentionPolicy.RUNTIME)，否则校验仍不生效。这也是另外一个容易疏忽的地方，究其原因，不显式标记 RetentionPolicy 时，默认使用的是 RetentionPolicy.CLASS，而这种类型的注解信息虽然会被保留在字节码文件（.class）中，但在加载进 JVM 时就会丢失了。所以在运行时，依据这个注解来判断是否校验，肯定会失效。

案例 2：嵌套校验失效

前面这个案例虽然比较经典，但是，它只是初学者容易犯的错误。实际上，关于 Validation 最容易忽略的是对嵌套对象的校验，我们沿用上面的案例举这样一个例子。

学生可能还需要一个联系电话信息，所以我们可以定义一个 Phone 对象，然后关联上学生对象，代码如下：

```
public class Student {
    @Size(max = 10)
    private String name;
    private short age;
    private Phone phone;
}
@Data
class Phone {
    @Size(max = 10)
    private String number;
}
```

这里我们也给 Phone 对象做了合法性要求（@Size(max = 10)），当我们使用下面的请求（请求 body 携带一个联系电话信息超过 10 位），测试校验会发现这个约束并不生效。

```
POST http://localhost:8080/students
Content-Type: application/json
{
  "name": "xiaoming",
  "age": 10,
  "phone": {"number":"12306123061230612306"}
}
```

为什么会不生效？

案例解析

在解析案例 1 时，我们提及只要给对象参数 student 加上 @Valid（或 @Validated 等注解）就可以开启这个对象的校验。但实际上，关于 student 本身的 Phone 类型成员是否校验是在校验过程中（即案例 1 中的代码行 binder.validate(validationHints)）决定的。

在校验执行时，首先会根据 Student 的类型定义找出所有的校验点，然后对 Student 对象实例执行校验，这个逻辑过程可以参考代码 ValidatorImpl#validate：

```
@Override
public final <T> Set<ConstraintViolation<T>> validate(T object, Class<?>... gr
   //省略部分非关键代码
   Class<T> rootBeanClass = (Class<T>) object.getClass();
   //获取校验对象类型的“信息”（包含“约束”）
   BeanMetaData<T> rootBeanMetaData = beanMetaDataManager.getBeanMetaData( roo

   if ( !rootBeanMetaData.hasConstraints() ) {
      return Collections.emptySet();
   }

   //省略部分非关键代码
   //执行校验
   return validateInContext( validationContext, valueContext, validationOrder
}
```

这里语句"beanMetaDataManager.getBeanMetaData(rootBeanClass)"根据 Student 类型组装出 BeanMetaData，BeanMetaData 即包含了需要做的校验（即 Constraint）。

在组装 BeanMetaData 过程中，会根据成员字段是否标记了 @Valid 来决定（记录）这个字段以后是否做级联校验，参考代码 AnnotationMetaDataProvider#getCascadingMetaData：

```
private CascadingMetaDataBuilder getCascadingMetaData(Type type, AnnotatedElem
      Map<TypeVariable<?>, CascadingMetaDataBuilder> containerElementTypesCasc
   return CascadingMetaDataBuilder.annotatedObject( type, annotatedElement.isA
         getGroupConversions( annotatedElement ) );
}
```

在上述代码中"annotatedElement.isAnnotationPresent(Valid.class)"决定了 CascadingMetaDataBuilder#cascading 是否为 true。如果是，则在后续做具体校验时，做级联校验，而级联校验的过程与宿主对象（即 Student）的校验过程大体相同，即先根据对象类型获取定义再来做校验。

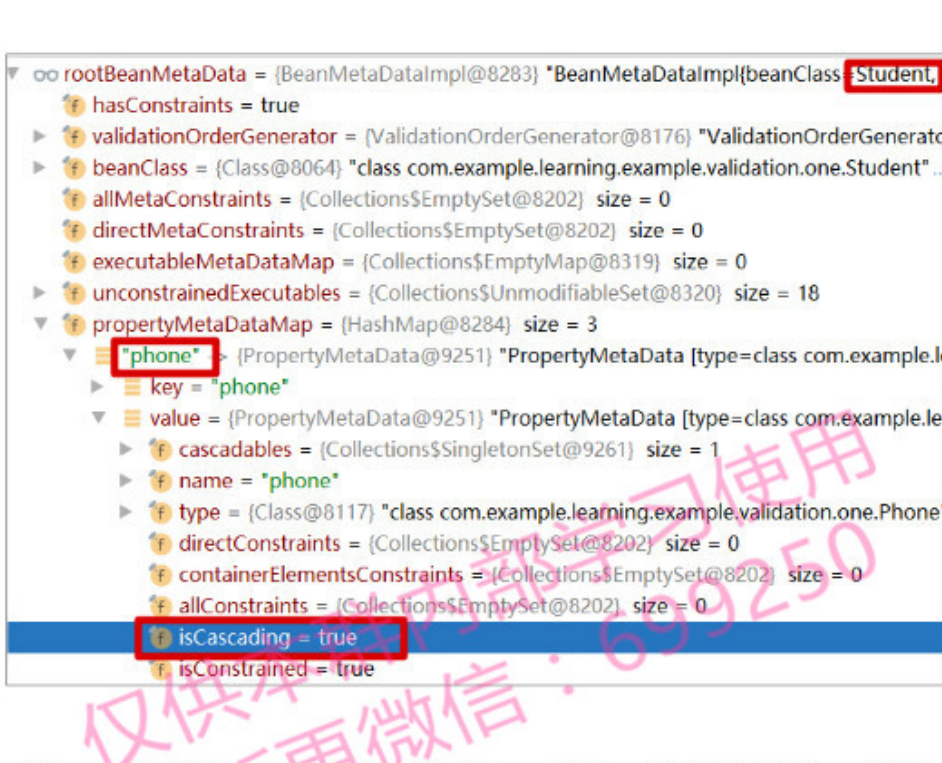
在当前案例代码中，phone 字段并没有被 @Valid 标记，所以关于这个字段信息的 cascading 属性肯定是 false，因此在校验 Student 时并不会级联校验它。

问题修正

从源码级别了解了嵌套 Validation 失败的原因后，我们会发现，要让嵌套校验生效，解决的方法只有一种，就是加上 @Valid，修正代码如下：

> @Valid
> private Phone phone;

当修正完问题后，我们会发现校验生效了。而如果此时去调试修正后的案例代码，会看到 phone 字段 MetaData 信息中的 cascading 确实为 true 了，参考下图：



另外，假设我们不去解读源码，我们很可能会按照案例 1 所述的其他修正方法去修正这个问题。例如，使用 @Validated 来修正这个问题，但是此时你会发现，不考虑源码是否支持，代码本身也编译不过，这主要在于 @Validated 的定义是不允许修饰一个 Field 的：

```
@Target({ElementType.TYPE, ElementType.METHOD, ElementType.PARAMETER})
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface Validated {
```

通过上述方法修正问题，最终我们让嵌套验证生效了。但是你可能还是会觉得这个错误看起来不容易犯，那么可以试想一下，我们的案例仅仅是嵌套一层，而产品代码往往都是嵌套 n 层，此时我们是否能保证每一级都不会疏忽漏加 @Valid 呢？所以这仍然是一个典型的错误，需要你格外注意。

案例 3：误解校验执行

通过前面两个案例的填坑，我们一般都能让参数校验生效起来，但是校验本身有时候是一个无止境的完善过程，校验本身已经生效，但是否完美匹配我们所有苛刻的要求是另外一个容易疏忽的地方。例如，我们可能在实践中误解一些校验的使用。这里我们可以继续沿用前面的案例，变形一下。

之前我们定义的学生对象的姓名要求是小于 10 字节的（即 @Size(max = 10)）。此时我们可能想完善校验，例如，我们希望姓名不能是空，此时你可能很容易想到去修改关键行代码如下：

```
@Size(min = 1, max = 10)
private String name;
```

然后，我们以下面的 JSON Body 做测试：

```
{
  "name": "",
  "age": 10,
  "phone": {"number":"12306"}
}
```

测试结果符合我们的预期，但是假设更进一步，用下面的 JSON Body（去除 name 字段）做测试呢？

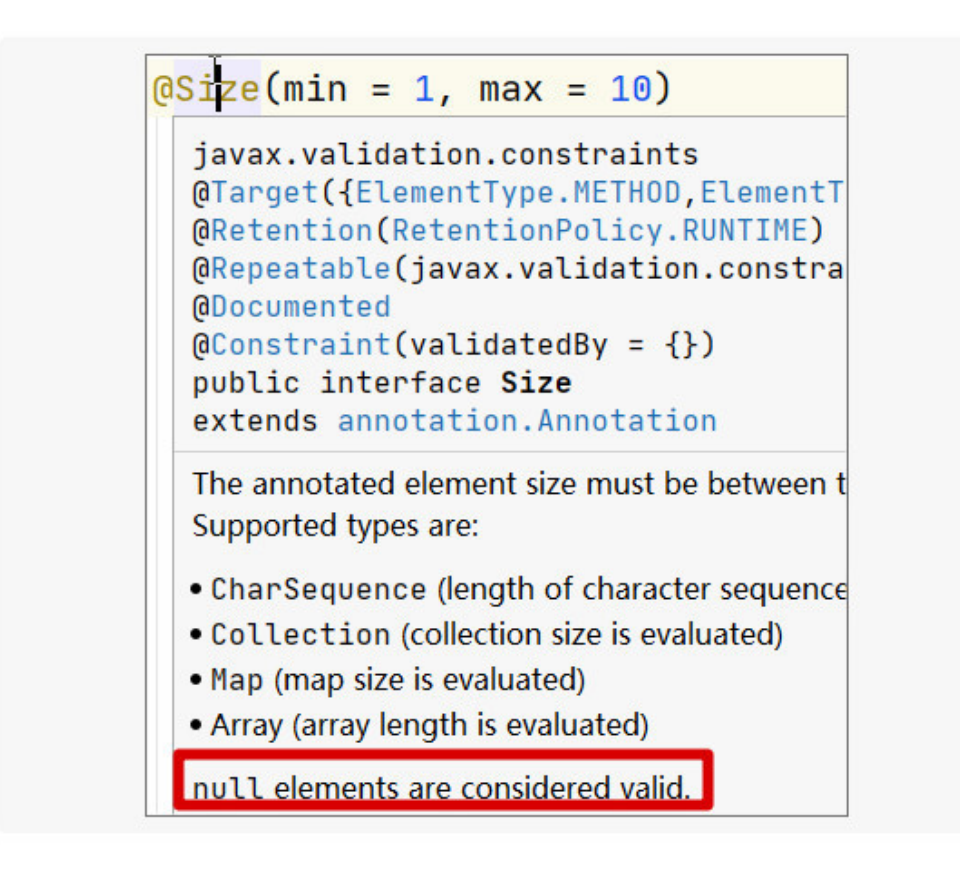
```
{
  "age": 10,
  "phone": {"number":"12306"}
}

```

我们会发现校验失败了。这结果难免让我们有一些惊讶，也倍感困惑：@Size(min = 1, max = 10) 都已经要求最小字节为 1 了，难道还只能约束空字符串（即“”），不能约束 null?

案例解析

如果我们稍微留心点的话，就会发现其实 @Size 的 Javadoc 已经明确了这种情况，参考下图：



如图所示，"null elements are considered valid" 很好地解释了约束不住 null 的原因。当然纸上得来终觉浅，我们还需要从源码级别解读下 @Size 的校验过程。

这里我们找到了完成 @Size 约束的执行方法，参考 SizeValidatorForCharSequence#isValid 方法：

```
   public boolean isValid(CharSequence charSequence, ConstraintValidatorContex
      if ( charSequence == null ) {
         return true;
      }
      int length = charSequence.length();
      return length >= min && length <= max;
   }
```

如代码所示，当字符串为 null 时，直接通过了校验，而不会做任何进一步的约束检查。

问题修正

关于这个问题的修正，其实很简单，我们可以使用其他的注解（@NotNull 或 @NotEmpty）来加强约束，修正代码如下：

```
@NotEmpty
@Size(min = 1, max = 10)
private String name;
```

完成代码修改后，重新测试，你就会发现约束已经完全满足我们的需求了。

重点回顾

看完上面的一些案例，我们会发现，这些错误的直接结果都是校验完全失败或者部分失败，并不会造成严重的后果，但是就像本讲开头所讲的那样，这些错误会影响我们的使用体验，所以我们还是需要去规避这些错误，把校验做强最好！

另外，关于 @Valid 和 @Validation 是我们经常犯迷糊的地方，不知道到底有什么区别。同时我们也经常产生一些困惑，例如能用其中一种时，能不能用另外一种呢？

通过解析，我们会发现，在很多场景下，我们不一定要寄希望于搜索引擎去区别，只需要稍微研读下代码，反而更容易理解。例如，对于案例 1，研读完代码后，我们发现它们不仅可以互换，而且完全可以自定义一个以 @Valid 开头的注解来使用；而对于案例 2，只能用 @Valid 去开启级联校验。

思考题

在上面的学籍管理系统中，我们还存在一个接口，负责根据学生的学号删除他的信息，代码如下：

```
@RequestMapping(path = "students/{id}", method = RequestMethod.DELETE)
public void deleteStudent(@PathVariable("id") @Range(min = 1,max = 10000) Stri
    log.info("delete student: {}",id);
    //省略业务代码
};
```

这个学生的编号是从请求的 Path 中获取的，而且它做了范围约束，必须在 1 到 10000 之间。那么你能找出负责解出 ID 的解析器（HandlerMethodArgumentResolver）是哪一种吗？校验又是如何触发的？

期待你的思考，我们留言区见！