链滴

springfox-swagger 参数是对象无限递归解决方案。

springfox第二大坑：Controller类的参数，注意防止出现无限递归的情况。

Spring mvc有强大的参数绑定机制，可以自动把请求参数绑定为一个自定义的命令对像。所以，很
开发人员在写Controller时，为了偷懒，直接把一个实体对像作为Controller方法的一个参数。比如
面这个示例代码：

@RequestMapping(value = "update")

public String update(MenuVomenuVo, Model model){

}

这是大部分程序员喜欢在Controller中写的修改某个实体的代码。在跟swagger集成的时候，这里有
个大坑。如果MenuVo这个类中所有的属性都是基本类型，那还好，不会出什么问题。但如果这个类
面有一些其它的自定义类型的属性，而且这个属性又直接或间接的存在它自身类型的属性，那就会出
题。例如：假如MenuVo这个类是菜单类，在这个类时又含有MenuVo类型的一个属性parent代表它
父级菜单。这样的话，系统启动时swagger模块就因无法加载这个api而直接报错。报错的原因就是
在加载这个方法的过程中会解析这个update方法的参数，发现参数MenuVo不是简单类型，则会自
以递归的方式解释它所有的类属性。这样就很容易陷入无限递归的死循环。

为了解决这个问题，我目前只是自己写了一个OperationParameterReader插件实现类以及它依赖的
odelAttributeParameterExpander工具类，通过配置的方式替换掉到srpingfox原来的那两个类，偷
换柱般的把参数解析这个逻辑替换掉，并避开无限递归。当然，这相当于是一种修改源码级别的方式
我目前还没有找到解决这个问题的更完美的方法，所以，只能建议大家在用spring-fox Swagger的时
尽量避免这种无限递归的情况。毕竟，这不符合springmvc命令对像的规范，springmvc参数的命令
像中最好只含有简单的基本类型属性。

原文地地址：https://blog.csdn.net/w4hechuan2009/article/details/68892718

这篇文章给出了解决方案但并未给我代码。我尝试的写出了修改代码。

```
        <dependency>
            <groupId>io.springfox</groupId>
            <artifactId>springfox-swagger2</artifactId>
        </dependency>
        <dependency>
            <groupId>io.springfox</groupId>
            <artifactId>springfox-swagger-ui</artifactId>
        </dependency>
```

version：2.8.0

代码如下：

```
/*
 *
 *  Copyright 2015-2016 the original author or authors.
 *
 *  Licensed under the Apache License, Version 2.0 (the "License");
 *  you may not use this file except in compliance with the License.
 *  You may obtain a copy of the License at
 *
 *         http://www.apache.org/licenses/LICENSE-2.0
 *
 *  Unless required by applicable law or agreed to in writing, software
 *  distributed under the License is distributed on an "AS IS" BASIS,
 *  WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 *  See the License for the specific language governing permissions and
 *  limitations under the License.
 *
 *
 */

package springfox.documentation.spring.web.readers.operation;

import com.fasterxml.classmate.ResolvedType;
import com.google.common.base.Predicate;
import com.google.common.collect.FluentIterable;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.core.Ordered;
import org.springframework.core.annotation.Order;
import org.springframework.stereotype.Component;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RequestPart;
import springfox.documentation.builders.ParameterBuilder;
import springfox.documentation.service.Parameter;
import springfox.documentation.service.ResolvedMethodParameter;
import springfox.documentation.spi.DocumentationType;
import springfox.documentation.spi.schema.EnumTypeDeterminer;
import springfox.documentation.spi.service.OperationBuilderPlugin;
import springfox.documentation.spi.service.contexts.OperationContext;
import springfox.documentation.spi.service.contexts.ParameterContext;
import springfox.documentation.spring.web.plugins.DocumentationPluginsManager;
import springfox.documentation.spring.web.readers.parameter.ExpansionContext;
import springfox.documentation.spring.web.readers.parameter.ModelAttributeParameterExp
nder;

import java.lang.annotation.Annotation;
import java.util.List;
import java.util.Set;

import static com.google.common.base.Predicates.*;
import static com.google.common.collect.Lists.*;
import static springfox.documentation.schema.Collections.*;
import static springfox.documentation.schema.Maps.*;
import static springfox.documentation.schema.Types.*;

@Component
@Order(Ordered.HIGHEST_PRECEDENCE)
public class OperationParameterReader implements OperationBuilderPlugin {
    private final ModelAttributeParameterExpander expander;
    private final EnumTypeDeterminer enumTypeDeterminer;

    @Autowired
    private DocumentationPluginsManager pluginsManager;
    @Autowired
    public OperationParameterReader(
            ModelAttributeParameterExpander expander,
            EnumTypeDeterminer enumTypeDeterminer) {
        this.expander = expander;
        this.enumTypeDeterminer = enumTypeDeterminer;
    }

    @Override
    public void apply(OperationContext context) {
        context.operationBuilder().parameters(context.getGlobalOperationParameters());
        context.operationBuilder().parameters(readParameters(context));
    }

    @Override
    public boolean supports(DocumentationType delimiter) {
        return true;
    }

    private List<Parameter> readParameters(final OperationContext context) {

        List<ResolvedMethodParameter> methodParameters = context.getParameters();

        List<Parameter> parameters = newArrayList();

        for (ResolvedMethodParameter methodParameter : methodParameters) {
            ResolvedType alternate = context.alternateFor(methodParameter.getParameterType());
            if (!shouldIgnore(methodParameter, alternate, context.getIgnorableParameterTypes()))
 {

                ParameterContext parameterContext = new ParameterContext(methodParameter,
                        new ParameterBuilder(),
                        context.getDocumentationContext(),
                        context.getGenericsNamingStrategy(),
                        context);

                if (shouldExpand(methodParameter, alternate)) {
//                    parameters.addAll(
//                            expander.expand(
//                                    new ExpansionContext("", alternate, context.getDocumentationConte
t())));
                } else {
                    parameters.add(pluginsManager.parameter(parameterContext));
                }
            }
        }
        return FluentIterable.from(parameters).filter(not(hiddenParams())).toList();
    }

    private Predicate<Parameter> hiddenParams() {
        return new Predicate<Parameter>() {
            @Override
            public boolean apply(Parameter input) {
                return input.isHidden();
```

原文链接：springfox-swagger 参数是对象无限递归解决方案。

```
            }
        };
    }

    private boolean shouldIgnore(
            final ResolvedMethodParameter parameter,
            ResolvedType resolvedParameterType,
            final Set<Class> ignorableParamTypes) {

        if (ignorableParamTypes.contains(resolvedParameterType.getErasedType())) {
            return true;
        }
        return FluentIterable.from(ignorableParamTypes)
                .filter(isAnnotation())
                .filter(parameterIsAnnotatedWithIt(parameter)).size() > 0;

    }

    private Predicate<Class> parameterIsAnnotatedWithIt(final ResolvedMethodParameter par
meter) {
        return new Predicate<Class>() {
            @Override
            public boolean apply(Class input) {
                return parameter.hasParameterAnnotation(input);
            }
        };
    }

    private Predicate<Class> isAnnotation() {
        return new Predicate<Class>() {
            @Override
            public boolean apply(Class input) {
                return Annotation.class.isAssignableFrom(input);
            }
        };
    }

    private boolean shouldExpand(final ResolvedMethodParameter parameter, ResolvedType r
solvedParamType) {
        return !parameter.hasParameterAnnotation(RequestBody.class)
                && !parameter.hasParameterAnnotation(RequestPart.class)
                && !parameter.hasParameterAnnotation(RequestParam.class)
                && !parameter.hasParameterAnnotation(PathVariable.class)
                && !isBaseType(typeNameFor(resolvedParamType.getErasedType()))
                && !enumTypeDeterminer.isEnum(resolvedParamType.getErasedType())
                && !isContainerType(resolvedParamType)
                && !isMapType(resolvedParamType);

    }

}


/*
```

```
 *
 *  Copyright 2015-2018 the original author or authors.
 *
 *  Licensed under the Apache License, Version 2.0 (the "License");
 *  you may not use this file except in compliance with the License.
 *  You may obtain a copy of the License at
 *
 *         http://www.apache.org/licenses/LICENSE-2.0
 *
 *  Unless required by applicable law or agreed to in writing, software
 *  distributed under the License is distributed on an "AS IS" BASIS,
 *  WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 *  See the License for the specific language governing permissions and
 *  limitations under the License.
 *
 *
 */

package springfox.documentation.spring.web.readers.parameter;

import com.fasterxml.classmate.ResolvedType;
import com.fasterxml.classmate.members.ResolvedField;
import com.google.common.annotations.VisibleForTesting;
import com.google.common.base.Function;
import com.google.common.base.Optional;
import com.google.common.base.Predicate;
import com.google.common.collect.FluentIterable;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;
import org.springframework.util.ClassUtils;
import springfox.documentation.builders.ParameterBuilder;
import springfox.documentation.schema.Maps;
import springfox.documentation.schema.Types;
import springfox.documentation.schema.property.field.FieldProvider;
import springfox.documentation.service.Parameter;
import springfox.documentation.spi.schema.AlternateTypeProvider;
import springfox.documentation.spi.schema.EnumTypeDeterminer;
import springfox.documentation.spi.service.contexts.DocumentationContext;
import springfox.documentation.spi.service.contexts.ParameterExpansionContext;
import springfox.documentation.spring.web.plugins.DocumentationPluginsManager;

import java.beans.BeanInfo;
import java.beans.IntrospectionException;
import java.beans.Introspector;
import java.beans.PropertyDescriptor;
import java.util.HashSet;
import java.util.List;
import java.util.Set;

import static com.google.common.base.Objects.*;
import static com.google.common.base.Predicates.*;
import static com.google.common.base.Strings.*;
import static com.google.common.collect.FluentIterable.*;
import static com.google.common.collect.Lists.*;
import static com.google.common.collect.Sets.*;
import static springfox.documentation.schema.Collections.*;
import static springfox.documentation.schema.Types.*;

@Component
public class ModelAttributeParameterExpander {
    private static final Logger LOG = LoggerFactory.getLogger(ModelAttributeParameterExpan
er.class);
    private final FieldProvider fieldProvider;
    private final EnumTypeDeterminer enumTypeDeterminer;

    @Autowired
    protected DocumentationPluginsManager pluginsManager;

    @Autowired
    public ModelAttributeParameterExpander(
            FieldProvider fields,
            EnumTypeDeterminer enumTypeDeterminer) {

        this.fieldProvider = fields;
        this.enumTypeDeterminer = enumTypeDeterminer;
    }

    public List<Parameter> expand(ExpansionContext context) {

        List<Parameter> parameters = newArrayList();
        Set<String> beanPropNames = getBeanPropertyNames(context.getParamType().getEras
dType());
        Iterable<ResolvedField> fields = FluentIterable.from(fieldProvider.in(context.getParamT
pe()))
                .filter(onlyBeanProperties(beanPropNames));
        LOG.debug("Expanding parameter type: {}", context.getParamType());
        AlternateTypeProvider alternateTypeProvider = context.getDocumentationContext().getA
ternateTypeProvider();

        FluentIterable<ModelAttributeField> modelAttributes = from(fields)
                .transform(toModelAttributeField(alternateTypeProvider));

        FluentIterable<ModelAttributeField> expendables = modelAttributes
                .filter(not(simpleType()))
                .filter(not(recursiveType(context)));
        for (ModelAttributeField each : expendables) {
            LOG.debug("Attempting to expand expandable field: {}", each.getField());
//            parameters.addAll(
//                    expand(
//                            context.childContext(
//                                    nestedParentName(context.getParentName(), each.getField()),
//                                    each.getFieldType(),
//                                    context.getDocumentationContext())));
        }

        FluentIterable<ModelAttributeField> collectionTypes = modelAttributes
```

```
            .filter(and(isCollection(), not(recursiveCollectionItemType(context.getParamType()))))

        for (ModelAttributeField each : collectionTypes) {
          LOG.debug("Attempting to expand collection/array field: {}", each.getField());

          ResolvedType itemType = collectionElementType(each.getFieldType());
          if (Types.isBaseType(itemType) || enumTypeDeterminer.isEnum(itemType.getErasedTy
e())) {
            parameters.add(simpleFields(context.getParentName(), context.getDocumentation
ontext(), each));
          } else {
//              parameters.addAll(
//                  expand(
//                      context.childContext(
//                          nestedParentName(context.getParentName(), each.getField()),
//                          itemType,
//                          context.getDocumentationContext())));
          }
        }

        FluentIterable<ModelAttributeField> simpleFields = modelAttributes.filter(simpleType());
        for (ModelAttributeField each : simpleFields) {
          parameters.add(simpleFields(context.getParentName(), context.getDocumentationCon
ext(), each));
        }
        return FluentIterable.from(parameters).filter(not(hiddenParameters())).toList();
    }

    private Predicate<ModelAttributeField> recursiveCollectionItemType(final ResolvedType pa
amType) {
        return new Predicate<ModelAttributeField>() {
          @Override
          public boolean apply(ModelAttributeField input) {
            return equal(collectionElementType(input.getFieldType()), paramType);
          }
        };
    }

    private Predicate<Parameter> hiddenParameters() {
        return new Predicate<Parameter>() {
          @Override
          public boolean apply(Parameter input) {
            return input.isHidden();
          }
        };
    }

    private Parameter simpleFields(
          String parentName,
          DocumentationContext documentationContext,
          ModelAttributeField each) {
        LOG.debug("Attempting to expand field: {}", each);
        String dataTypeName = Optional.fromNullable(typeNameFor(each.getFieldType().getEra
edType()))
```

```
            .or(each.getFieldType().getErasedType().getSimpleName());
        LOG.debug("Building parameter for field: {}, with type: ", each, each.getFieldType());
        ParameterExpansionContext parameterExpansionContext = new ParameterExpansionCon
ext(
            dataTypeName,
            parentName,
            each.getField(),
            documentationContext.getDocumentationType(),
            new ParameterBuilder());
        return pluginsManager.expandParameter(parameterExpansionContext);
    }


    private Predicate<ModelAttributeField> recursiveType(final ExpansionContext context) {
        return new Predicate<ModelAttributeField>() {
            @Override
            public boolean apply(ModelAttributeField input) {
                return context.hasSeenType(input.getFieldType());
            }
        };
    }

    private Predicate<ModelAttributeField> simpleType() {
        return and(not(isCollection()), not(isMap()),
                or(
                        belongsToJavaPackage(),
                        isBaseType(),
                        isEnum()));
    }

    private Predicate<ModelAttributeField> isCollection() {
        return new Predicate<ModelAttributeField>() {
            @Override
            public boolean apply(ModelAttributeField input) {
                return isContainerType(input.getFieldType());
            }
        };
    }

    private Predicate<ModelAttributeField> isMap() {
        return new Predicate<ModelAttributeField>() {
            @Override
            public boolean apply(ModelAttributeField input) {
                return Maps.isMapType(input.getFieldType());
            }
        };
    }

    private Predicate<ModelAttributeField> isEnum() {
        return new Predicate<ModelAttributeField>() {
            @Override
            public boolean apply(ModelAttributeField input) {
                return enumTypeDeterminer.isEnum(input.getFieldType().getErasedType());
            }
        };
    }

    private Predicate<ModelAttributeField> belongsToJavaPackage() {
        return new Predicate<ModelAttributeField>() {
            @Override
            public boolean apply(ModelAttributeField input) {
                return ClassUtils.getPackageName(input.getFieldType().getErasedType()).startsWith(
java.lang");
            }
        };
    }

    private Predicate<ModelAttributeField> isBaseType() {
        return new Predicate<ModelAttributeField>() {
            @Override
            public boolean apply(ModelAttributeField input) {
                return Types.isBaseType(input.getFieldType())
                        || input.getField().getType().isPrimitive();
            }
        };
    }

    private Function<ResolvedField, ModelAttributeField> toModelAttributeField(
            final AlternateTypeProvider
                    alternateTypeProvider) {
        return new Function<ResolvedField, ModelAttributeField>() {
            @Override
            public ModelAttributeField apply(ResolvedField input) {
                return new ModelAttributeField(fieldType(alternateTypeProvider, input), input);
            }
        };
    }

    private Predicate<ResolvedField> onlyBeanProperties(final Set<String> beanPropNames) {
        return new Predicate<ResolvedField>() {
            @Override
            public boolean apply(ResolvedField input) {
                return beanPropNames.contains(input.getName());
            }
        };
    }

    private String nestedParentName(String parentName, ResolvedField field) {
        String name = field.getName();
        ResolvedType fieldType = field.getType();
        if (isContainerType(fieldType) && !Types.isBaseType(collectionElementType(fieldType))) {
            name += "[0]";
        }

        if (isNullOrEmpty(parentName)) {
            return name;
        }
        return String.format("%s.%s", parentName, name);
    }

    private ResolvedType fieldType(AlternateTypeProvider alternateTypeProvider, ResolvedFiel
 field) {
        return alternateTypeProvider.alternateFor(field.getType());
    }

    private Set<String> getBeanPropertyNames(final Class<?> clazz) {

        try {
            Set<String> beanProps = new HashSet<String>();
            PropertyDescriptor[] propDescriptors = getBeanInfo(clazz).getPropertyDescriptors();

            for (PropertyDescriptor propDescriptor : propDescriptors) {

                if (propDescriptor.getReadMethod() != null) {
                    beanProps.add(propDescriptor.getName());
                }
            }

            return beanProps;

        } catch (IntrospectionException e) {
            LOG.warn(String.format("Failed to get bean properties on (%s)", clazz), e);
        }
        return newHashSet();
    }

    @VisibleForTesting
    BeanInfo getBeanInfo(Class<?> clazz) throws IntrospectionException {
        return Introspector.getBeanInfo(clazz);
    }

}
```

这样就能解决swagger 参数递归问题了。

这里统一回复下，这些代码是我网上抄的，费了好大力气。

但是经过测试还是没有完美的解决无限递归问题。

原文链接：springfox-swagger 参数是对象无限递归解决方案。