自定义Spring Boot Starter：使用注解+AOP实现业务日志处理

目录

正文

引言

Spring Boot 的自定义 Starter 允许我们将通用功能模块封装起来，以便在多个项目中复用。这不仅减少了重复代码，还提高了项目的一致性和可维护性。根据本人过往开发经历，本文将介绍如何实现一个简单的自定义 Starter，使用自定义注解+AOP切面实现一个监控业务方法执行流程的日志管理，并展示在项目中如何使用它。

自定义 Starter 的使用场景

在实际开发中，有很多场景需要自定义 Starter，比如：

* 统一的日志处理
* 复杂的第三方库集成
* 公司内部的标准化开发框架
* 统一配置和管理

本文介绍的属于第一种，主要作用是为了让业务流程日志的监管与实际业务解耦和代码多次复用的目的，并且支持自动配置，功能扩展；

简单介绍一下业务背景，我们有一个业务中台需要收集各个业务系统用户在各类业务上操作的日志信息做数据分析使用，而我负责的项目，需要提供50多个业务节点的流程监管数据。我们系统是一个微服务项目，业务分布在各个服务，总不能在每个业务节点触发的地方都触发一些流程监管吧；考虑到代码复用、使用方便、统一管理数据发送的方面，决定写一个starter组件。

创建自定义 Starter 项目

以下代码实例中，重点在于如何自定义starter、一个日志切面处理流程思路、以及对于可以扩展的功能思考。所以有些实现细节并未展示，只提供了大致的结构代码，感兴趣可以在此基础上自行扩展。

项目结构

首先，创建一个新的 Maven 项目，项目结构如下：

```
my-starter
├── src
│   └── main
│       ├── java
│       │   └── com
│       │       └── pujiho
│       │           ├── annotations
│       │           │   └── ProcessWatch.java
│       │           ├── aspects
│       │           │   └── ProcessWatchAspect.java
│       │           ├── config
│       │           │   ├── ProcessWatchAutoConfiguration.java
│       │           │   └── ProcessWatchProperties.java
│       │           ├── entity
│       │           │   └── ProcessWatchEntity.java
│       │           ├── service
│       │           │   ├── impl
│       │           │   │   ├── DefaultBusinessDataBuilderImpl.java
│       │           │   │   └── SendDataService.java
│       │           │   ├── IBusinessDataBuilder.java
```

```
│       │               └── IOperationDataService.java
│       │               └── ProcessWatchContext.java
│       └── resources
│           └── META-INF
│               └── spring.factories
└── pom.xml
```

添加必要依赖

在 `pom.xml` 中添加必要的依赖 ,其中`spring-boot-configuration-processor`的作用是在配置文件中添加配置时，可以得到提示，类似于我们使用开源组件进行配置时会有自动补全提示的效果。

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <parent>
        <artifactId>example-components</artifactId>
        <groupId>com.pujiho.example</groupId>
        <version>1.0-SNAPSHOT</version>
    </parent>
    <modelVersion>4.0.0</modelVersion>

    <artifactId>processWatch-spring-boot-starter</artifactId>
    <packaging>jar</packaging>
    <version>1.0-SNAPSHOT</version>

    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-aop</artifactId>
        </dependency>

        <dependency>
            <groupId>com.alibaba</groupId>
            <artifactId>fastjson</artifactId>
            <version>1.2.75</version>
        </dependency>

        <dependency>
```

```
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
            <optional>true</optional>
        </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-configuration-processor</artifactId>
            <optional>true</optional>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <configuration>
                    <source>1.8</source>
                    <target>1.8</target>
                </configuration>
            </plugin>
        </plugins>
    </build>

</project>
```

实现自动配置类

创建配置属性类

首先，我们创建一个配置属性类 `ProcessWatchProperties`，主要设置了一些对接业务中台监管服务的地址和失败重试相关配置，使用 `@ConfigurationProperties`注解可以将配置文件中的属性值注入到Java类中。

```
package com.pujiho.config;

import lombok.Getter;
import lombok.Setter;
import
```

```
org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.context.annotation.Configuration;

@Setter
@Getter
@Configuration
@ConfigurationProperties(”processwatch”)
public class ProcessWatchProperties {

    /** 是否启用监管服务，默认开启 */
    private boolean enable = true;

    /** 监管服务地址 */
    private String serverAddress;

    /** 监管服务端口 */
    private String serverPort;

    /** 监管服务接口路径 */
    private String serverPath = ”dwjk/api/v1/msg”;

    /** 发送失败是否需要重试 */
    private boolean serverRetryEnabled;

    /** 重试次数 */
    private boolean serverRetryAttempts;

    /** 重试间隔时间 */
    private long serverRetryBackoff = 1000L;

}
```

编写自动配置类

接下来，创建一个自动配置类 `ProcessWatchAutoConfiguration`，在其中对一些复杂的对象进行初始化操作。这里`SendDataServiceImpl`实现`IOperationDataService`接口，是默认的日志数据处理方式，即发送到中台流程监管系统，根据配置信息创建连接；

使用`@ConditionalOnProperty`注解根据配置是否开启，决定是否需要创建对象；

如何未开启配置，则需要实现`IOperationDataService`接口，创建自己的数据

处理实现，例如将业务日志数据存储到数据库。当然关于数据处理，也可以兼容多种实现，所有实现了`IOperationDataService`接口的都会执行，下面切面处理中有代码展示；

```
package com.pujiho.config;

import com.pujiho.service.IOperationDataService;
import com.pujiho.service.impl.SendDataServiceImpl;
import org.springframework.beans.factory.annotation.Autowired;
import
org.springframework.boot.autoconfigure.condition.ConditionalOnProperty
;
import
org.springframework.boot.context.properties.EnableConfigurationPropert
ies;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

@Configuration
@EnableConfigurationProperties({ProcessWatchProperties.class})
//@ConditionalOnProperty(prefix = ”processwatch”, name = ”enable”,
havingValue = ”true”)
public class ProcessWatchAutoConfiguration {


    @Autowired
    private ProcessWatchProperties processWatchProperties;


    @Bean(”defaultOperationService”)
    @ConditionalOnProperty(prefix = ”processwatch”, name = ”enable”,
havingValue = ”true”)
    public IOperationDataService operationDataService() {
        SendDataServiceImpl sendDataService = new
SendDataServiceImpl();
        // todo 初始化连接
        return sendDataService;
    }

}
```

创建自定义注解

创建`ProcessWatch`注解，标注于业务方法使用。提供流程数据获取的一些自定义规则属性，是否获取参数，是否获取返回值，或者排查某个参数。

```
package com.pujiho.annotations;

import java.lang.annotation.*;

@Documented
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface ProcessWatch {

    /**业务类型 */
    String businessType() default ”1.0”;

    /** 是否获取参数*/
    boolean isSaveParamData() default true;

    /** 是否获取返回值 */
    boolean isSaveReturnData() default true;

    /** 排除指定参数 */
    String[] excludeParamNames() default {};
}
```

切面处理逻辑

创建切面处理类

创建`ProcessWatchAspect`类，实现切面逻辑。

* 使用`ProcessWatchEntity`构建基本数据，记录业务方法位置、业务类型；可自行扩展业务操作人信息。
* 获取监管数据，即通过切面获取到的业务方法入参和返回值，获取流程中需要的数据，详情请见下面关于获取监管数据的实现。
* 处理监管数据，默认发送监管系统，可自行实现其他处理方式。

```
package com.pujiho.aspects;

import com.pujiho.annotations.ProcessWatch;
import com.pujiho.entity.ProcessWatchEntity;
import com.pujiho.service.IBusinessDataBuilder;
import com.pujiho.service.IOperationDataService;
import com.pujiho.service.ProcessWatchContext;
import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.AfterReturning;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.reflect.MethodSignature;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.stereotype.Component;

import java.lang.reflect.Method;
import java.util.List;

@Slf4j
@Component
@Aspect
public class ProcessWatchAspect {

    private final List<IOperationDataService> operationDataServiceList;

    private final ProcessWatchContext processWatchContext;

    public ProcessWatchAspect(List<IOperationDataService>
operationDataServiceList, ProcessWatchContext processWatchContext) {
        this.operationDataServiceList = operationDataServiceList;
        this.processWatchContext = processWatchContext;
    }


    @AfterReturning(pointcut = "@annotation(processWatch)",returning =
"result")
    public void afterReturning(JoinPoint point, ProcessWatch
processWatch, Object result){
        try {
            Method method =
((MethodSignature)point.getSignature()).getMethod();
            String businessType = processWatch.businessType();
            Object[] objs = point.getArgs();
            // 构建基础数据
            ProcessWatchEntity processWatchEntity = new
ProcessWatchEntity();
```

```
            processWatchEntity.setBusinessType(businessType);

 processWatchEntity.setOperationMethod(method.getDeclaringClass()+"
#"+method.getName());
            // 获取监管数据
            IBusinessDataBuilder builder =
processWatchContext.getBuilder(businessType);
            Object watchData = builder.buildData(processWatch, objs,
result);
            processWatchEntity.setWatchData(watchData.toString());
            // 处理数据
            for (IOperationDataService operationDataService:
operationDataServiceList) {
                operationDataService.operationData(processWatchEntity);
            }
        } catch (Exception e) {
            log.error("流程监管切面出现异常", e);
        }
    }
}
```

`ProcessWatchEntity`代码示例：

```
package com.pujiho.entity;

import lombok.Data;

import java.io.Serializable;

@Data
public class ProcessWatchEntity implements Serializable {

    /** 业务类型 */
    private String businessType;

    /** 操作方法  */
    private String operationMethod;

    /** 监管数据 */
    private String watchData;
}
```

获取监管数据

实现`IBusinessDataBuilder`接口，获取需要监管的业务数据，从注解、业务方法入参、业务方法返回值中获取值并构建。提供`DefaultBusinessDataBuilderImpl`作为默认实现，即根据业务方法`ProcessWatch`注解中的属性设置构建数据。

```
javpackage com.pujiho.service;

import com.pujiho.annotations.ProcessWatch;

public interface IBusinessDataBuilder {

    /**
     * 获取业务类型
     * @return 业务类型
     */
    String type();

    /**
     * 构建监管数据
     * @param watch ProcessWatch注解信息
     * @param objects 参数信息
     * @param result 返回值信息
     * @return 监管的结果数据
     */
    Object buildData(ProcessWatch watch, Object[] objects, Object result);
}
```

`IBusinessDataBuilder`默认实现`DefaultBusinessDataBuilderImpl`示例代码：

```
package com.pujiho.service.impl;

import com.pujiho.annotations.ProcessWatch;
import com.pujiho.service.IBusinessDataBuilder;
import org.springframework.stereotype.Service;
```

```
import java.util.HashMap;
import java.util.Map;

@Service
public class DefaultBusinessDataBuilderImpl implements
IBusinessDataBuilder {

    @Override
    public String type() {
        return null;
    }

    @Override
    public Object buildData(ProcessWatch watch, Object[] objects, Object
result) {
        // 根据注解信息判断哪些数据需要作为监管数据
        Map<String, Object> data = new HashMap<>();
        if (watch.isSaveParamData()) {
            data.put(”param”, objects);
        }
        if (watch.isSaveReturnData()) {
            data.put(”result”, result);
        }
        // todo 解析参数, 封装参数
        return data;
    }

}
```

在`ProcessWatchContext`构造器中注入所有的的`IBusinessDataBuilder`实现类，根据实现的`type()`方法返回的业务类型作为key；在上述切面处理类`ProcessWatchAspect`中传入`ProcessWatch`注解中设置的businessType的调用`getBuilder()`方法获取对应业务类型的`IBusinessDataBuilder`实现。

如果对于没有实现`IBusinessDataBuilder`接口自定义监管数据获取规则，则会使用默认实现`DefaultBusinessDataBuilderImpl`获取数据。

```
package com.pujiho.service;

import com.pujiho.service.impl.DefaultBusinessDataBuilderImpl;
import lombok.extern.slf4j.Slf4j;
```

```
import org.springframework.stereotype.Component;

import java.util.HashMap;
import java.util.List;
import java.util.Map;

/**
 * @ClassName ProcessWatchContext
 * @Description TODO
 * @Author pujiho
 * @Date 2024/5/24 16:29
 */
@Slf4j
@Component
public class ProcessWatchContext {

    private static final Map<String, IBusinessDataBuilder>
BUILDER_MAPPING = new HashMap();

    private IBusinessDataBuilder defaultBusinessDataBuilder;

    public ProcessWatchContext( List<IBusinessDataBuilder>
businessDataBuilderList) {
        if (businessDataBuilderList == null || businessDataBuilderList.size()
== 0) {
            log.warn("IBusinessDataBuilder未被实例化");
            return;
        }
        for (IBusinessDataBuilder builder : businessDataBuilderList) {
            if (builder instanceof DefaultBusinessDataBuilderImpl) {
                this.defaultBusinessDataBuilder = builder;
            }
            if (builder.type() != null) {
                BUILDER_MAPPING.put(builder.type(), builder);
            }
        }
    }

    /**
     * 获取数据构建Builder
     * @param type 类型
     * @return IBusinessDataBuilder
     */
    public IBusinessDataBuilder getBuilder(String type) {
        return BUILDER_MAPPING.getOrDefault(type,
defaultBusinessDataBuilder);
    }
}
```

```

##### 监管数据处理

通过实现接口`IOperationDataService`对数据进行处理，我这儿默认的实现类`SendDataServiceImpl`是发送外部系统，也可以根据需要自行调整；

```
package com.pujiho.service;

import com.pujiho.entity.ProcessWatchEntity;

/**
 * @ClassName IOperationDataService
 * @Description TODO
 * @Author pujiho
 * @Date 2024/5/24 16:26
 */
public interface IOperationDataService {

    void operationData(ProcessWatchEntity data);
}
```

`SendDataServiceImpl`伪代码：

```
package com.pujiho.service.impl;

import com.alibaba.fastjson.JSONObject;
import com.pujiho.entity.ProcessWatchEntity;
import com.pujiho.service.IOperationDataService;
import lombok.extern.slf4j.Slf4j;

@Slf4j
public class SendDataServiceImpl implements IOperationDataService {

    @Override
    public void operationData(ProcessWatchEntity data) {
        log.info("使用SendDataServiceImpl实现类开始处理监控数据，发送监管系统");
```
```

```
        log.info("打印监控数据：" + JSONObject.toJSONString(data));
    }
}
```

创建 Spring Factories 文件

在 `src/main/resources/META-INF/` 目录下创建 `spring.factories` 文件，声明自动配置类，其他需要spring管理的对象（例如使用@Component、@Service标注的类）要需要在此配置，以便扫描注入到容器。

```
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
  com.pujiho.config.ProcessWatchAutoConfiguration,\
  com.pujiho.service.impl.DefaultBusinessDataBuilderImpl,\
  com.pujiho.service.ProcessWatchContext,\
  com.pujiho.aspects.ProcessWatchAspect
```

发布自定义 Starter

将自定义 Starter 项目打包并发布到 Maven 仓库，便于在其他项目中使用。我这儿是直接使用idea中的maven插件执行install将组件发布到我的配置的本地仓库中。

在项目中使用自定义 Starter

在另一个 Spring Boot 项目中引入自定义 Starter：

```
<dependency>
    <groupId>com.pujiho.example</groupId>
    <artifactId>processWatch-spring-boot-starter</artifactId>
    <version>1.0-SNAPSHOT</version>
</dependency>
```

在 `application.yml` 中配置 Starter 属性：

```
processwatch:
  # 开启发送监管功能，默认开启，可不配置
  enable: true
  # 监管系统信息
  server-address: xxx
  server-port: xxxx
  server-path: xxxx
  server-retry-attempts: false
```

创建一个业务方法，使用注解， 默认获取返回值为true，我们这儿设置成false；

```
@Service
public class ProcessWatchTest {

    @ProcessWatch(businessType = "1.1", isSaveReturnData = false)
    public String invokeBusiness(User user, BigDecimal money) {
        return "执行成功";
    }
}
```

使用单元测试执行：

```
@SpringBootTest
class SpringBootTest {


    @Autowired
    private ProcessWatchTest processWatchTest;

    @Test
    void processWatch(){
        User user = new User("pujiho", 18);
```

```
        processWatchTest.invokeBusiness(user, BigDecimal.ONE);
    }
}
```

获取数据和处理数据都使用默认实现

从执行日志可以看出，使用了`SendDataServiceImpl`作为默认数据处理实现，从打印的数据结果可以看到数据中只有业务方法的入参，没有返回值，符合预期。

```
2024-06-03 18:04:52.554  INFO 54308 --- [           main] c.p.service.impl.SendDataServiceImpl     : 使用SendDataServiceImpl实现类开始处理监控数据， 发送监管系统
2024-06-03 18:04:52.554  INFO 54308 --- [           main] c.p.service.impl.SendDataServiceImpl     : 打印监控数据：{"businessType":"1.1","operationMethod":"class com.pujiho.practice.spring.starter.ProcessWatchTest#invokeBusiness","watchData":"{"param":[{"age":18,"name":"pujiho"},1]}"}
2024-06-03 18:04:52.570  INFO 54308 --- [extShutdownHook] o.s.s.concurrent.ThreadPoolTaskExecutor  : Shutting down ExecutorService 'applicationTaskExecutor'
```

获取数据使用自定义实现，处理数据添加实现

实现数据获取接口`IBusinessDataBuilder`,我们这儿只返回业务方法的返回值，type方法返回“1.1”，映射上面的业务方法；

```
@Slf4j
@Service
public class MyBusinessBuilder implements IBusinessDataBuilder {

    @Override
    public String type() {
        return "1.1";
    }
```

```
    @Override
    public Object buildData(ProcessWatch processWatch, Object[]
objects, Object o) {
        log.info("使用自定义MyBusinessBuilder获取监控数据");
        return o;
    }
}

```

实现`IOperationDataService`接口，创建另外的数据处理方式。

```
@Slf4j
@Service
public class OtherOperationDataService implements
IOperationDataService {


    @Override
    public void operationData(ProcessWatchEntity processWatchEntity) {
        log.info("添加OtherOperationDataService实现处理数据，进行业务日
志入库处理");
        log.info("打印监管数据："+
JSONObject.toJSONString(processWatchEntity));
    }
}

```

执行业务方法，从执行结果日志可以看出，自定义的数据处理实现`OtherOperationDataService`也成功执行；自定义的数据构建`MyBusinessBuilder`也执行成功，从打印的监管数据看，自定义实现的优先级要高于注解中的属性设置，只获取了业务方法的返回值。

```
2024-06-03 18:14:05.072  INFO 46976 --- [           main]
c.p.p.spring.starter.MyBusinessBuilder   : 使用自定义MyBusinessBuilder获
取监控数据,只获取业务方法返回值
2024-06-03 18:14:05.122  INFO 46976 --- [           main]
c.p.p.s.s.OtherOperationDataService      : 添加
OtherOperationDataService实现处理数据，进行业务日志入库处理
2024-06-03 18:14:05.149  INFO 46976 --- [           main]
c.p.p.s.s.OtherOperationDataService      : 打印监管数据
```

: {"businessType":"1.1","operationMethod":"class com.pujiho.practice.spring.starter.ProcessWatchTest#invokeBusiness","watchData":""执行成功""}
2024-06-03 18:14:05.149 INFO 46976 --- [main] c.p.service.impl.SendDataServiceImpl   : 使用SendDataServiceImpl实现类开始处理监控数据， 发送监管系统
2024-06-03 18:14:05.149 INFO 46976 --- [main] c.p.service.impl.SendDataServiceImpl   : 打印监控数据
: {"businessType":"1.1","operationMethod":"class com.pujiho.practice.spring.starter.ProcessWatchTest#invokeBusiness","watchData":""执行成功""}
2024-06-03 18:14:05.171 INFO 46976 --- [extShutdownHook] o.s.s.concurrent.ThreadPoolTaskExecutor : Shutting down ExecutorService 'applicationTaskExecutor'

```

#### 结论

通过自定义 Starter，我们可以将通用功能模块化，并在多个项目中复用。本文详细介绍了如何实现一个关于业务日志切面处理的自定义 Starter，并在项目中使用它。希望这些步骤能帮助你在实际开发中创建和使用自定义 Starter，提高开发效率和代码一致性。

原文链接: https://juejin.cn/post/7375810931098959935
```