目录

功能介绍

介绍

该框架主要是集成于springboot项目，用于开发插件式应用的集成框架。

核心功能

1. 插件配置式插拔于springboot项目。
2. 在springboot上可以进行插件式开发，扩展性极强，可以针对不同项目开发不同插件，进行不同插件jar包的部署。
3. 可通过配置文件指定要启用或者禁用插件。
4. 支持上传插件和插件配置文件到服务器，并且无需重启主程序，动态部署插件、更新插件。
5. 支持查看插件运行状态，查看插件安装位置。
6. 无需重启主程序，动态的安装插件、卸载插件、启用插件、停止插件、备份插件、删除插件。
7. 在插件应用模块上可以使用Spring注解定义组件，进行依赖注入。
8. 支持在插件中开发Rest接口。
9. 支持在插件中单独定义持久层访问等需求。
10. 可以遵循主程序提供的插件接口开发任意扩展功能。
11. 插件可以自定义配置文件。目前只支持yml文件。
12. 支持自定义扩展开发接口，使用者可以在预留接口上扩展额外功能。
13. 利用扩展机制，定制了SpringBoot-Mybatis扩展包。使用该扩展包，使用者可以在插件中自定义Mapper接口、Mapper xml 以及对应的实体bean。并且支持集成Mybatis-Plus。
14. 支持插件之间的通信。
15. 支持插件中使用事务注解。
16. 支持Swagger。(仅支持首次启动初始化的插件)

运行环境

1. jdk1.8+
2. apache maven 3.6

maven 仓库地址

https://mvnrepository.com/artifact/com.gitee.starblues/springboot-plugin-framework

快速入门

新建项目。

Maven目录结构下所示

```
-example
   - example-runner
      - pom.xml
   - example-main
      - pom.xml
   - example-plugin-parent
      - pom.xml
   - plugins
      - example-plugin1
         - pom.xml
         - plugin.properties
      - example-plugin2
         - pom.xml
         - plugin.properties
      - pom.xml
   - pom.xml
```

结构说明:

1. pom.xml 代表maven的pom.xml
2. plugin.properties 为开发环境下, 插件的元信息配置文件, 配置内容详见 插件包集成步骤。
3. example 为项目的总Maven目录。
4. example-runner 在运行环境下启动的模块。主要依赖example-main模块和插件中使用到的依赖包, 并且解决开发环境下无法找到插件依赖包的问题。可自行选择是否需要。(可选)
5. example-main 该模块为项目的主程序模块。
6. example-plugin-parent 该模块为插件的父级maven pom 模块, 主要定义插件中公共用到的依赖, 以及插件的打包配置。
7. plugins 该文件夹下主要存储插件模块。上述模块中主要包括example-plugin1、example-plugin2 两个插件。
8. example-plugin1、example-plugin2 分别为两个插件Maven包。

主程序集成步骤

主程序为上述目录结构中的 example-main 模块。

1. 在主程序中新增maven依赖包

```
<dependency>
   <groupId>com.gitee.starblues</groupId>
   <artifactId>springboot-plugin-framework</artifactId>
   <version>${springboot-plugin-framework.version}</version>
</dependency>
```

2. 实现并定义配置

实现 **com.plugin.development.integration.IntegrationConfiguration** 接口。

```
import com.gitee.starblues.integration.DefaultIntegrationConfiguration;
import org.pf4j.RuntimeMode;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.stereotype.Component;

@Component
@ConfigurationProperties(prefix = "plugin")
public class PluginConfiguration extends DefaultIntegrationConfiguration {

    /**
     * 运行模式
     *  开发环境: development、dev
```

```
     *  生产/部署 环境: deployment、prod
     */
    @Value("${runMode:dev}")
    private String runMode;

    /**
     * 插件的路径
     */
    @Value("${pluginPath:plugins}")
    private String pluginPath;

    /**
     * 插件文件的路径
     */
    @Value("${pluginConfigFilePath:pluginConfigs}")
    private String pluginConfigFilePath;


    @Override
    public RuntimeMode environment() {
        return RuntimeMode.byName(runMode);
    }

    @Override
    public String pluginPath() {
        return pluginPath;
    }

    @Override
    public String pluginConfigFilePath() {
        return pluginConfigFilePath;
    }

    /**
     * 重写上传插件包的临时存储路径。只适用于生产环境
     * @return String
     */
    @Override
    public String uploadTempPath() {
        return "temp";
    }

    /**
     * 重写插件备份路径。只适用于生产环境
     * @return String
     */
    @Override
    public String backupPath() {
        return "backupPlugin";
    }

    /**
     * 重写插件RestController请求的路径前缀
     * @return String
     */
    @Override
    public String pluginRestControllerPathPrefix() {
        return "/api/plugins";
    }

    /**
     * 重写是否启用插件id作为RestController请求的路径前缀。
     * 启动则插件id会作为二级路径前缀。即: /api/plugins/pluginId/**
     * @return String
     */
    @Override
    public boolean enablePluginIdRestControllerPathPrefix() {
        return true;
    }

    public String getRunMode() {
```

```
        return runMode;
    }

    public void setRunMode(String runMode) {
        this.runMode = runMode;
    }


    public String getPluginPath() {
        return pluginPath;
    }

    public void setPluginPath(String pluginPath) {
        this.pluginPath = pluginPath;
    }

    public String getPluginConfigFilePath() {
        return pluginConfigFilePath;
    }

    public void setPluginConfigFilePath(String pluginConfigFilePath) {
        this.pluginConfigFilePath = pluginConfigFilePath;
    }

    @Override
    public String toString() {
        return "PluginArgConfiguration{" +
                "runMode='" + runMode + '\'' +
                ", pluginPath='" + pluginPath + '\'' +
                ", pluginConfigFilePath='" + pluginConfigFilePath + '\'' +
                '}';
    }
}
```

配置说明:

runMode：运行项目时的模式。分为开发环境(dev)、生产环境(prod)

pluginPath: 插件的路径。开发环境建议直接配置为插件模块的父级目录。例如: plugins。如果启动主程序时, 插件为加载, 请检查该配置是否正确。

pluginConfigFilePath: 在生产环境下, 插件的配置文件路径。在生产环境下 ， 请将所有插件使用到的配置文件统一放到该路径下管理。如果启动主程序时, 报插件的配置文件加载错误, 有可能是该该配置不合适导致的。

uploadTempPath: 上传插件包时使用。上传插件包存储的临时路径。默认 temp(相对于主程序jar路径)

backupPath: 备份插件包时使用。备份插件包的路径。默认: backupPlugin(相对于主程序jar路径)

pluginRestControllerPathPrefix: 插件RestController请求的路径前缀

enablePluginIdRestControllerPathPrefix: 是否启用插件id作为RestController请求的路径前缀。启动则插件id会作为二级路径前缀。即: /api/plugins/pluginId/**

3. 配置PluginApplication

```
import com.gitee.starblues.integration.*;
import com.gitee.starblues.integration.initialize.AutoPluginInitializer;
import com.gitee.starblues.integration.initialize.PluginInitializer;
import org.pf4j.PluginException;
import org.pf4j.PluginManager;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class PluginBeanConfig {

    @Bean
    public PluginApplication pluginApplication(){
        // 实例化自动初始化插件的PluginApplication
        PluginApplication pluginApplication = new AutoPluginApplication();
        return pluginApplication;
```

```
    }

}
```

插件包集成步骤

1. 插件包pom.xml配置说明

以 `<scope>provided</scope>` 方式引入springboot-plugin-framework包

```
<dependency>
    <groupId>com.gitee.starblues</groupId>
    <artifactId>springboot-plugin-framework</artifactId>
    <version>${springboot-plugin-framework.version}</version>
    <scope>provided</scope>
</dependency>
```

定义打包配置.主要用途是将 `Plugin-Id、Plugin-Version、Plugin-Provider、Plugin-Class、Plugin-Dependencies` 的配置值定义到 `META-INF\MANIFEST.MF` 文件中

```
<properties>
    <plugin.id>example-plugin1</plugin.id>
    <plugin.class>com.plugin.example.plugin1.DefinePlugin</plugin.class>
    <plugin.version>${project.version}</plugin.version>
    <plugin.provider>StarBlues</plugin.provider>
    <plugin.dependencies></plugin.dependencies>

    <java.version>1.8</java.version>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>

    <maven-compiler-plugin.version>3.7.0</maven-compiler-plugin.version>
    <maven-assembly-plugin.version>3.1.1</maven-assembly-plugin.version>
</properties>
<build>
    <plugins>

        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>${maven-compiler-plugin.version}</version>
            <configuration>
                <source>${java.version}</source>
                <target>${java.version}</target>
                <encoding>${project.build.sourceEncoding}</encoding>
            </configuration>
        </plugin>

        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-assembly-plugin</artifactId>
            <version>${maven-assembly-plugin.version}</version>
            <configuration>
                <descriptorRefs>
                    <descriptorRef>jar-with-dependencies</descriptorRef>
                </descriptorRefs>
                <archive>
                    <manifest>
                        <addDefaultImplementationEntries>true</addDefaultImplementationEntries>
                        <addDefaultSpecificationEntries>true</addDefaultSpecificationEntries>
                    </manifest>
                    <manifestEntries>
                        <Plugin-Id>${plugin.id}</Plugin-Id>
                        <Plugin-Version>${plugin.version}</Plugin-Version>
                        <Plugin-Provider>${plugin.provider}</Plugin-Provider>
                        <Plugin-Class>${plugin.class}</Plugin-Class>
                    </manifestEntries>
```

```
            </archive>
        </configuration>
        <executions>
            <execution>
                <id>make-assembly</id>
                <phase>package</phase>
                <goals>
                    <goal>single</goal>
                </goals>
            </execution>
        </executions>
    </plugin>
</plugins>
</build>
```

2. 在插件包的一级目录下新建plugin.properties文件(用于开发环境) 新增如下内容(属性值同步骤1中pom.xml定义的 `manifestEntries` 属性一致):

```
plugin.id=example-plugin1
plugin.class=com.plugin.example.plugin1.DefinePlugin
plugin.version=2.0-SNAPSHOT
plugin.provider=StarBlues
```

配置说明:

```
plugin.id: 插件id
plugin.class: 插件实现类。见步骤3说明
plugin.version: 插件版本
plugin.provider: 插件作者
```

3. 继承 `com.gitee.starblues.realize.BasePlugin` 包

```
import com.gitee.starblues.realize.BasePlugin;
import org.pf4j.PluginWrapper;

public class DefinePlugin extends BasePlugin {
    public DefinePlugin(PluginWrapper wrapper) {
        super(wrapper);
    }

    @Override
    protected void startEvent() {

    }

    @Override
    protected void deleteEvent() {

    }

    @Override
    protected void stopEvent() {

    }
}
```

并且将该类的包路径(com.plugin.example.plugin1.DefinePlugin)配置在步骤1和2的plugin.class属性中。

4. 新增HelloPlugin1 controller

此步骤主要验证环境是否加载插件成功。

```

@RestController
@RequestMapping(path = "plugin1")
public class HelloPlugin1 {
```

```
    @GetMapping()
    public String getConfig(){
        return "hello plugin1 example";
    }

}
```

运行配置

1. 配置模块 example-runner 的pom.xml

- 将主程序的依赖新增到pom.xml 下
- 将插件中的依赖以 `<scope>provided</scope>` 方式引入到 pom.xml 下

如下所示:

```
<?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">

    <modelVersion>4.0.0</modelVersion>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.0.3.RELEASE</version>
        <relativePath/>
    </parent>

    <groupId>com.gitee.starblues</groupId>
    <artifactId>plugin-example-runner</artifactId>
    <version>2.0-RELEASE</version>
    <packaging>pom</packaging>

    <properties>
        <gson.version>2.8.2</gson.version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>com.gitee.starblues</groupId>
            <artifactId>plugin-example-start</artifactId>
            <version>${project.version}</version>
        </dependency>

        <!-- 此处依赖用于解决在开发环境下, 插件包找不到对应依赖包 -->
        <!--
        <dependency>
            <groupId>com.google.code.gson</groupId>
            <artifactId>gson</artifactId>
            <version>${gson.version}</version>
            <scope>provided</scope>
        </dependency>
        -->

    </dependencies>

</project>
```

2. 设置idea的启动

Working directory : D:\xx\xx\plugin-example

Use classpath of module: plugin-exampe-runner

勾选: Include dependencies with "Provided" scope

3. 启动2步骤的配置。

注意: 启动前一定要确保插件模块的target生成了class类，否则会加载不到插件或者报错，常见问题参考 `注意事项和常见错误`

观察日志出现如下说明加载插件成功。

```
Plugin 'example-plugin1@2.0-RELEASE' resolved
Start plugin 'example-plugin1@2.0-RELEASE'
Init Plugins <example-plugin1> Success
```

4. 访问插件中的Controller 验证。

浏览器输入：http://ip:port/api/plugins/example-plugin1/plugin1

响应并显示: hello plugin1 example

说明集成成功！

集成框架配置说明

主程序集成配置说明

1. 首先主程序需要继承 `com.gitee.starblues.integration.DefaultIntegrationConfiguration` 该类。在初始化插件时需要传入。该类主要时插件的核心配置。具体说明如下:

```
public class Config extends DefaultIntegrationConfiguration {

    /**
     * 必填。 运行环境。运行项目时的模式。分为开发环境(DEVELOPMENT)、生产环境(DEPLOYMENT)
     * @return RuntimeMode.DEVELOPMENT、RuntimeMode.DEPLOYMENT
     */
    @Override
    public RuntimeMode environment() {
        return RuntimeMode.DEVELOPMENT;
    }

    /**
     * 必填。插件的路径。开发环境建议直接配置为插件模块的父级目录。
       例如: plugins。如果启动主程序时, 插件为加载, 请检查该配置是否正确。
     * @return 插件的路径
     */
    @Override
    public String pluginPath() {
        return null;
    }


    /**
     * 必填。插件文件的配置路径。在生产环境下, 插件的配置文件路径。
     * 在生产环境下，请将所有插件使用到的配置文件统一放到该路径下管理。
     * 在开发环境下，配置为空串。程序会自动从 resources 获取配置文件，请确保编译后的target下存在该配置文件
     * @return 插件文件的配置路径
     */
    @Override
    public String pluginConfigFilePath() {
        return null;
    }

    /**
     * 非必填。上传插件包存储的临时路径。
     * 默认 temp(相对于主程序jar路径)。
     * @return 上传插件的临时保存路径。
     */
    @Override
    public String uploadTempPath() {
        return super.uploadTempPath();
    }

    /**
     * 非必填。插件备份路径。
     * 默认 backupPlugin  (相对于主程序jar路径)。
     * @return 插件备份路径。
     */
    @Override
    public String backupPath() {
        return super.backupPath();
    }

    /**
     * 非必填。插件 RestController 统一请求的路径前缀
     * @return path
     */
    @Override
    public String pluginRestControllerPathPrefix() {
        return super.pluginRestControllerPathPrefix();
    }
```

```
    /**
     * 非必填。启用插件id作为RestController的路径前缀。默认启用
     * 如果启用。则路径前缀为 pluginRestControllerPathPrefix() 返回的路径拼接插件id,
     * 即为: /pathPrefix/pluginId/**
     * @return boolean
     */
    @Override
    public boolean enablePluginIdRestControllerPathPrefix() {
        return super.enablePluginIdRestControllerPathPrefix();
    }


}
```

您也可以使用Builder来构建配置。例如:

```
@Component
@ConfigurationProperties(prefix = "plugin")
public class PluginBeanConfig {

    /**
     * 运行模式
     *  开发环境: development、dev
     *  生产/部署 环境: deployment、prod
     */
    @Value("${runMode:dev}")
    private String runMode;

    /**
     * 插件的路径
     */
    @Value("${pluginPath:plugins}")
    private String pluginPath;

    /**
     * 插件文件的路径
     */
    @Value("${pluginConfigFilePath:pluginConfigs}")
    private String pluginConfigFilePath;



    @Bean
    public IntegrationConfiguration configuration(){
        return ConfigurationBuilder.toBuilder()
                .runtimeMode(RuntimeMode.byName(runMode))
                .pluginPath(pluginPath)
                .pluginConfigFilePath(pluginConfigFilePath)
                .uploadTempPath("temp")
                .backupPath("backupPlugin")
                .pluginRestControllerPathPrefix("/api/plugin")
                .enablePluginIdRestControllerPathPrefix(true)
                .build();
    }


    public void setRunMode(String runMode) {
        this.runMode = runMode;
    }

    public void setPluginPath(String pluginPath) {
        this.pluginPath = pluginPath;
    }

    public void setPluginConfigFilePath(String pluginConfigFilePath) {
        this.pluginConfigFilePath = pluginConfigFilePath;
    }
```

```
}
```

2. 配置PluginApplication

- 自动初始化配置方式(AutoPluginApplication): 该配置方式可以将启动插件的初始化工作交由Spring容器初始化完成后自动调用。具体配置如下:

```
@Bean
public PluginApplication pluginApplication(){
    // 实例化自动初始化插件的PluginApplication
    PluginApplication pluginApplication = new AutoPluginApplication();
    return pluginApplication;
}
```

- 手动(默认)初始化配置（PluginApplication）：该配置方式可在必要的时候手动初始化插件。具体配置如下：

```
@Configuration
public class PluginConfig {

    @Autowired
    private ApplicationContext applicationContext;

    @Bean
    public PluginApplication pluginApplication(){
        // 实例化自动初始化插件的PluginApplication
        PluginApplication pluginApplication = new DefaultPluginApplication();
        return pluginApplication;
    }

    @PostConstruct
    public void init(){
        pluginApplication.initialize(applicationContext, new PluginInitializerListener() {
            @Override
            public void before() {

            }

            @Override
            public void complete() {

            }

            @Override
            public void failure(Throwable throwable) {

            }
        });
    }

}
```

插件之间数据交互功能

插件之间的数据交互功能, 是在同一JVM运行环境下, 基于代理、反射机制完成方法调用。使用说明如下：

1. 被调用类需要使用注解 @Supplier(""), 注解值为被调用者的唯一key, (全局key不能重复) 供调用者使用。例如:

```
@Supplier("SupplierService")
public class SupplierService {

    public Integer add(Integer a1, Integer a2){
        return a1 + a2;
    }

}
```

2. 另一个插件中要调用1步骤中定义的调用类时, 需要定义一个接口,新增注解@Caller(""), 值为1步骤中被调用者定义的全局key。其中方法名、参数个数和类型、返回类型需要和被调用者中定义的方法名、参数个数和类型一致。例如:

```
@Caller("SupplierService")
public interface CallerService {

    Integer add(Integer a1, Integer a2);

}
```

3.被调用者和调用者也可以使用注解定义被调用的方法。例如:

被调用者:

```
@Supplier("SupplierService")
public class SupplierService {

    @Supplier.Method("call")
    public String call(CallerInfo callerInfo, String key){
        System.out.println(callerInfo);
        return key;
    }

}
```

调用者:

```
@Caller("SupplierService")
public interface CallerService {

    @Caller.Method("call")
    String test(CallerInfo callerInfo, String key);

}
```

该场景主要用于参数类型不在同一个地方定义时使用。比如 被调用者的参数类: CallerInfo 定义在被调用者的插件中, 调用者的参数类: CallerInfo 定义在调用者的插件中。就必须配合 @Supplier.Method("")、@Caller.Method("") 注解使用, 否则会导致NotFoundClass 异常。

如果调用者没有使用注解 @Caller.Method("") 则默认使用方法和参数类型来调用。

4.对于3步骤中问题的建议

可以将被调用者和调用者的公用参数和返回值定义在主程序中、或者单独提出一个api maven包, 然后两者都依赖该包。

5.案例位置

basic-example：

com.basic.example.plugin1.service.SupplierService com.basic.example.plugin2.service.CallerService com.basic.example.plugin2.rest.ProxyController

插件自定义配置文件

1. 在插件包的 resources 目录下定义配置文件 plugin1.yml

```
name: plugin1
plugin: examplePlugin1
setString:
  - set1
  - set2
listInteger:
  - 1
  - 2
  - 3
subConfig:
  subName: subConfigName
```

2. 在代码中定义对应的bean

```
import com.gitee.starblues.annotation.ConfigDefinition;
import java.util.List;
import java.util.Set;

@ConfigDefinition("plugin1.yml")
public class PluginConfig1 {

    private String name;
    private String plugin;
    private Set<String> setString;
    private List<Integer> listInteger;
    private String defaultValue = "defaultValue";
    private SubConfig subConfig;

    // 自行提供get set 方法

}


public class SubConfig {

    private String subName;
    public String getSubName() {
        return subName;
    }

    // 自行提供get set 方法
}
```

该bean必须加上 @ConfigDefinition("plugin1.yml") 注解。其中值为插件文件的名称。

3. 其他地方使用时, 可以通过注入方式使用。

例如：

```
@Component("plugin2HelloService")
public class HelloService {

    private final PluginConfig1 pluginConfig1;
    private final Service2 service2;

    @Autowired
    public HelloService(PluginConfig1 pluginConfig1, Service2 service2) {
        this.pluginConfig1 = pluginConfig1;
        this.service2 = service2;
    }

    public PluginConfig1 getPluginConfig1(){
        return pluginConfig1;
```

```
    }


    public String sayService2(){
        return service2.getName();
    }

}
```

PluginUser 接口说明

介绍

该接口用于在主程序操作Spring管理的插件bean. 主要用途是: 在主程序定义接口, 插件中实现该接口做扩展, 主程序通过接口class可以获取到插件中的实现类。

接口位置

`com.gitee.starblues.integration.user.PluginUser`

使用

通过 PluginApplication 获取 PluginUser。

```
private final PluginUser pluginUser;

@Autowired
public PluginResource(PluginApplication pluginApplication) {
    this.pluginUser = pluginApplication.getPluginUser();
}
```

接口说明

```

    /**
     * 通过bean名称得到bean。（Spring管理的bean）
     * @param name bean的名称。spring体系中的bean名称。可以通过注解定义，也可以自定义生成。具体可百度
     * @param <T> bean的类型
     * @return T
     */
    <T> T getBean(String name);

    /**
     * 通过aClass得到bean。（Spring管理的bean）
     * @param aClass class
     * @param <T> bean的类型
     * @return T
     */
    <T> T getBean(Class<T> aClass);

    /**
     * 通过bean名称得到插件中的bean。（Spring管理的bean）
     * @param name 插件中bean的名称。spring体系中的bean名称。可以通过注解定义，也可以自定义生成。具体可百度
     * @param <T> bean的类型
     * @return T
     */
    <T> T getPluginBean(String name);

    /**
     * 在主程序中定义的接口。
     * 插件或者主程序实现该接口。可以该方法获取到实现该接口的所有实现类。（Spring管理的bean）
     * 使用场景:
     *  1. 在主程序定义接口
     *  2. 在主程序和插件包中都存在实现该接口, 并使用Spring的组件注解(@Component、@Service)
     *  3. 使用该方法可以获取到所以实现该接口的实现类(主程序和插件中)。
     * @param aClass 接口的类
     * @param <T> bean的类型
     * @return List
     */
    <T> List<T> getBeans(Class<T> aClass);

    /**
     * 得到主函数中定义的类。
     * 使用场景:
     *  1. 在主程序定义接口
```

```
     *  2. 在主程序和插件包中都存在实现该接口, 并使用Spring的组件注解(@Component、@Service)
     *  3. 使用该方法可以获取到主程序实现该接口的实现类。
     * @param aClass 类/接口的类
     * @param <T> bean 的类型
     * @return List
     */
    <T> List<T> getMainBeans(Class<T> aClass);


    /**
     * 在主程序中定义的接口。获取插件中实现该接口的实现类。（Spring管理的bean）
     * 使用场景:
     *  1. 在主程序定义接口
     *  2. 插件包中实现该接口, 并使用Spring的组件注解(@Component、@Service)
     *  3. 使用该方法可以获取到插件中实现该接口的实现类(不包括主程序)。
     * @param aClass 接口的类
     * @param <T> bean的类型
     * @return 实现 aClass 接口的实现类的集合
     */
    <T> List<T> getPluginBeans(Class<T> aClass);

    /**
     * 在主程序中定义的接口。获取指定插件中实现该接口的实现类。（Spring管理的bean）
     * 使用场景:
     *  1. 在主程序定义接口
     *  2. 插件包中实现该接口, 并使用Spring的组件注解(@Component、@Service)
     *  3. 使用该方法可以获取到指定插件中实现该接口的实现类。
     * @param pluginId 插件id
     * @param aClass 接口的类
     * @param <T> bean的类型
     * @return 实现 aClass 接口的实现类的集合
     */
    <T> List<T> getPluginBeans(String pluginId, Class<T> aClass);

    /**
     * 使用场景:
     * 1. 在主程序定义接口(该接口需要继承 ExtensionPoint 接口)。
     * 2. 插件包中实现该接口
     * 3. 在主程序可以使用该方法获取到实现该接口的实现类。（实现类可以配合 @Extension 控制顺序）
     * 注意: 该场景用于非Spring管理的bean, 使用Spring注解无效
     * @param tClass bean的类型
     * @param <T> bean的类型
     * @return List
     */
    <T> List<T> getPluginExtensions(Class<T> tClass);
```

具体使用参考 `basic-example` 案例

插件动态部署

介绍

- 可通过配置文件指定要启用或者禁用插件。
- 支持上传插件和插件配置文件到服务器, 并且无需重启主程序, 动态部署插件、更新插件。
- 支持查看插件运行状态, 查看插件安装位置。
- 无需重启主程序, 动态的安装插件、卸载插件、启用插件、停止插件、备份插件、删除插件。

操作方式

主要调用如下类进行插件的动态部署功能

`com.gitee.starblues.integration.operator.PluginOperator`

通过 PluginApplication 获取。

```
private final PluginOperator pluginOperator;

@Autowired
public PluginResource(PluginApplication pluginApplication) {
    this.pluginOperator = pluginApplication.getPluginOperator();
}
```

PluginOperator 接口说明

```
/**
 * 初始化插件。该方法只能执行一次。
 * @param pluginInitializerListener 插件初始化监听者
 * @return 成功返回true.不成功抛出异常或者返回false
 * @throws Exception 异常信息
 */
boolean initPlugins(PluginInitializerListener pluginInitializerListener) throws Exception;


/**
 * 通过路径安装插件(会启用), 该插件文件必须存在于服务器 [适用于生产环境]
 * 如果在插件目录存在同名的插件包(大小不为0k), 系统会自动备份该插件包。备份文件命名规则为；[install-backup][时间]_原jar名.jar
 * @param path 插件路径
 * @return 成功返回true.不成功抛出异常或者返回false
 * @throws Exception 异常信息
 */
boolean install(Path path) throws Exception;


/**
 * 卸载插件 [适用于生产环境]
 * 卸载后, 会将该插件备份到备份目录, 备份文件命名规则为；[uninstall][时间]_原jar名.jar
 * 在插件目录, 卸载的插件文件会变成0k. 这是正常的。在下次启动时, 程序会自动清除该文件。
 * @param pluginId 插件id
 * @return 成功返回true.不成功抛出异常或者返回false
 * @throws Exception 异常信息
 */
boolean uninstall(String pluginId) throws Exception;

/**
 * 启用插件 [适用于生产环境、开发环境]
 * @param pluginId 插件id
 * @return 成功返回true.不成功抛出异常或者返回false
 * @throws Exception 异常信息
 */
boolean start(String pluginId) throws Exception;
```

```
/**
 * 停止插件 [适用于生产环境、开发环境]
 * @param pluginId 插件id
 * @return 成功返回true.不成功抛出异常或者返回false
 * @throws Exception 异常信息
 */
boolean stop(String pluginId) throws Exception;


/**
 * 上传插件并启用插件。[适用于生产环境]
 * 如果在插件目录存在同名的插件包(大小不为0k), 系统会自动备份该插件包。备份文件命名规则为：[install-backup][时间]_原jar名.jar
 * @param pluginFile 配置文件
 * @return 成功返回true.不成功返回false, 或者抛出异常
 * @throws Exception 异常信息
 */
boolean uploadPluginAndStart(MultipartFile pluginFile) throws Exception;

/**
 * 通过路径安装插件的配置文件。该文件必须存在于服务器。[适用于生产环境]
 * 如果配置文件目录存在同名的配置文件, 系统会自动备份该配置文件。备份文件命名规则为：[install-config-backup][时间]_原jar名.jar
 * @param path 配置文件路径。
 * @return 成功返回true.不成功返回false, 或者抛出异常
 * @throws Exception
 */
boolean installConfigFile(Path path) throws Exception;


/**
 * 上传配置文件。[适用于生产环境]
 * 如果配置文件目录存在同名的配置文件, 系统会自动备份该配置文件。备份文件命名规则为：[upload-config-backup][时间]_原jar名.jar
 * @param configFile 配置文件
 * @return 成功返回true.不成功返回false, 或者抛出异常
 * @throws Exception 异常信息
 */
boolean uploadConfigFile(MultipartFile configFile) throws Exception;

/**
 * 通过路径备份文件。可备份插件和插件的配置文件。[适用于生产环境]
 * @param path 路径
 * @param sign 备份文件的自定义标识
 * @return 成功返回true.不成功返回false, 或者抛出异常
 * @throws Exception 异常信息
 */
boolean backupPlugin(Path path, String sign) throws Exception;

/**
 * 通过插件id备份插件。[适用于生产环境]
 * @param pluginId 插件id
 * @param sign 备份文件的自定义标识
 * @return 成功返回true.不成功返回false, 或者抛出异常
 * @throws Exception 异常信息
 */
boolean backupPlugin(String pluginId, String sign) throws Exception;

/**
 * 获取插件信息 [适用于生产环境、开发环境]
 * @return 返回插件信息列表
 */
List<PluginInfo> getPluginInfo();


/**
 * 得到插件文件的路径 [适用于生产环境]
 * @return 返回插件路径列表
 * @throws Exception 异常信息
 */
Set<String> getPluginFilePaths() throws Exception;
```

```
    /**
     * 得到插件的包装类 [适用于生产环境、开发环境]
     * @return 返回插件包装类集合
     */
    List<PluginWrapper> getPluginWrapper();
```

案例代码地址

案例 `basic-example` ： `com.basic.example.main.rest.PluginResource`

案例 `integration-mybatis` ： `com.mybatis.main.rest.PluginResource`

生产环境配置禁用启用功能

启用功能

1.在插件目录下新建 `enabled.txt` 文件 2.enabled.txt的内容为:

```
########################################
# - 启用的插件
########################################
example-plugin1
```

将需要启用的插件id配置到文件中。

所有注释行（以 # 字符开头的行）都将被忽略。

启用、禁用功能

1.在插件目录下新建 `disabled.txt` 文件 2.disabled.txt的内容为:

```
########################################
# - 禁用的插件
########################################
example-plugin1
```

将需要启用的插件id配置到文件中。

所有注释行（以 # 字符开头的行）都将被忽略。

监听器

主程序定义的监听器

初始化监听器

1. 实现接口 `com.gitee.starblues.integration.listener.PluginInitializerListener`

```
@Component
public class PluginListener implements PluginInitializerListener {
    @Override
    public void before() {
        System.out.println("初始化之前");
    }

    @Override
    public void complete() {
        System.out.println("初始化完成");
    }

    @Override
    public void failure(Throwable throwable) {
        System.out.println("初始化失败:"+throwable.getMessage());
    }
}
```

2. 注册监听器

- 该监听器只能设置一个
- 自动初始化监听器的设置

```


@Bean
public PluginApplication pluginApplication(PluginListener pluginListener){
    AutoPluginApplication autoPluginApplication = new AutoPluginApplication();
    autoPluginApplication.setPluginInitializerListener(pluginListener);
    return autoPluginApplication;
}
```

- 手动初始化监听器的设置

```
@Configuration
public class PluginBeanConfig {

    /**
     * 定义出 DefaultPluginApplication
     * @return PluginApplication
     */
    @Bean
    public PluginApplication pluginApplication(){
        PluginApplication pluginApplication = new DefaultPluginApplication();
        return pluginApplication;
    }

}


@Component
public class Init {

    @Autowired
    private ApplicationContext applicationContext;

    @Autowired
    private PluginListener pluginListener;
```

```
    @Autowired
    private PluginApplication pluginApplication;


    @PostConstruct
    public void init(){
        // 开始执行初始化
        pluginApplication.initialize(applicationContext, pluginListener);
    }

}
```

插件启动、停止监听

1. 实现接口 `com.gitee.starblues.integration.listener.PluginListener`

```
public class Listener implements PluginListener {
    @Override
    public void registry(String pluginId) {
        // 注册插件
    }

    @Override
    public void unRegistry(String pluginId) {
        // 卸载插件
    }

    @Override
    public void failure(String pluginId, Throwable throwable) {
        // 插件运行错误
    }
}
```

2. 注册监听器

- 该监听器可以注册多个

```
@Bean
public PluginApplication pluginApplication(){
    PluginApplication pluginApplication = new AutoPluginApplication();
    pluginApplication.addListener(Listener.class);
    // 或者使用 pluginApplication.addListener(new Listener());
}
```

3. 添加监听器说明

- pluginApplication.addListener(Listener.class);

该方式新增的监听器，可以使用注入，将Spring容器中的bean注入到 Listener 类中。

- pluginApplication.addListener(new Listener());

该方式新增的监听器，无法使用Spring的注入功能。因为是手动 new 出的对象。

插件中定义的监听器

1. 实现 `com.gitee.starblues.realize.OneselfListener` 接口即可。无需注册

例如：

```
public class Plugin1Listener implements OneselfListener {
```

```
    private final Logger logger = LoggerFactory.getLogger(Plugin1Listener.class);

    private final Plugin1Mapper plugin1Mapper;

    public Plugin1Listener(Plugin1Mapper plugin1Mapper) {
        this.plugin1Mapper = plugin1Mapper;
    }


    @Override
    public OrderPriority order() {
        // 定义监听器执行顺序。用于多个监听器
        return OrderPriority.getMiddlePriority();
    }

    @Override
    public void startEvent(BasePlugin basePlugin) {
        // 启动事件
        logger.info("Plugin1Listener {} start Event", basePlugin.getWrapper().getPluginId());
        logger.info("plugin1Mapper getList : {}", plugin1Mapper.getList());
    }

    @Override
    public void stopEvent(BasePlugin basePlugin) {
        // 停止事件
        logger.info("Plugin1Listener {} stop Event", basePlugin.getWrapper().getPluginId());
        logger.info("plugin1Mapper getList : {}", plugin1Mapper.getList());
    }
}
```

2. 一个插件模块的监听器可定义多个。执行顺序由 OrderPriority 控制。
3. 案例

参考: example/integration-mybatis 模块中定义的监听器。

插件案例

案例说明

- basic-example：插件基础功能案例。
- integration-mybatis: 针对Mybatis集成的案例。
- integration-mybatisplus: 针对Mybatis-Plus集成的案例

基础功能案例演示

- 普通例子运行见：basic-example
- windows环境下运行: package.bat
- linux、mac 环境下运行: package.sh
- 接口地址查看: http://127.0.0.1:8080/doc.html
- 代码位置: example/basic-example

mybatis 案例演示

注意：该案例运行前需要初始化sql, 并且在配置文件配置好相应的mysql数据库连接。

- 例子见：integration-mybatis
- windows环境下运行: package.bat
- linux、mac 环境下运行: package.sh
- sql在 integration-mybatis/sql 文件夹下。
- 接口地址查看: http://127.0.0.1:8081/doc.html
- 案例代码位置: example/integration-mybatis

mybatis-plus 案例演示

注意：该案例运行前需要初始化sql, 并且在配置文件配置好相应的mysql数据库连接。

- 例子见：integration-mybatisplus
- windows环境下运行: package.bat
- linux、mac 环境下运行: package.sh
- sql在 integration-mybatisplus/sql 文件夹下。
- 接口地址查看: http://127.0.0.1:8082/doc.html
- 案例代码位置: example/integration-mybatisplus

案例源码启动说明

和Spring boot 启动方式一样。直接启动主类。如果插件未加载, 请检查 application-dev.yml 的如下配置。重点检查 `pluginPath` 配置， 该配置主要是插件代码或者jar存放的上一级目录。

```
plugin:
  runMode: dev
  pluginPath: ./example/basic-example/plugins
  pluginConfigFilePath:
```

案例常见报错

1. 插件未编译 java.lang.ClassNotFoundException: com.basic.example.plugin1.DefinePlugin java.lang.ClassNotFoundException: com.basic.example.plugin2.DefinePlugin 该类型报错是由于插件源码没有编译成 class 文件; 需要手动编译, 保证在插件目录出现 target 文件

项目目录

生产环境建议目录

```
-main.jar

-main.yml

-plugins
  -plugin1.jar
  -plugin2.jar

-pluginFile
  -plugin1.yml
  -plugin2.yml
```

结构说明:

1. main.jar 为主程序。
2. main.yml 为主程序配置文件。
3. plugins 为插件存放的目录。plugin1.jar、plugin2.jar 分别为两个插件。
4. pluginFile 插件的配置文件存放位置。plugin1.yml、plugin2.yml 分别为 plugin1.jar、plugin2.jar 的配置。
5. 不一定每个插件都需要配置文件，可根据需求来。如果代码中定义了配置文件，则启动时必须将配置文件存放到pluginFile目录。不能使用jar包中的配置文件。

开发环境建议目录

```
-example
    - example-runner
        - pom.xml
    - example-main
        - pom.xml
    - example-plugin-parent
        - pom.xml
    - plugins
        - example-plugin1
            - pom.xml
            - plugin.properties
        - example-plugin2
            - pom.xml
            - plugin.properties
        - pom.xml
    - pom.xml
```

结构说明:

1. pom.xml 代表maven的pom.xml
2. plugin.properties 为开发环境下, 插件的元信息配置文件, 配置内容详见下文。
3. example 为项目的总Maven目录。
4. example-runner 在运行环境下启动的模块。主要依赖example-main模块和插件中使用到的依赖包, 并且解决开发环境下无法找到插件依赖包的问题。
5. example-main 该模块为项目的主程序模块。
6. example-plugin-parent 该模块为插件的父级maven pom 模块, 主要定义插件中公共用到的依赖, 以及插件的打包配置。
7. plugins 该文件夹下主要存储插件模块。上述模块中主要包括example-plugin1、example-plugin2 两个插件。
8. example-plugin1、example-plugin2 分别为两个插件Maven包。

注意事项和常见错误

注意事项

插件配置文件说明

- 在开发环境：配置文件必须放在resources目录下。并且@ConfigDefinition("plugin1.yml")中定义的文件名和resources下配置的文件名一致
- 在生产环境: 该文件存放在 `pluginConfigFilePath` 配置的目录下。生产环境下插件的配置文件必须外置, 不能使用jar包里面的配置文件。

打包插件

- 打包插件时, 必须使用maven打包插件将如下信息打入到文件中。否则启动时无法加载插件。

```
Plugin-Version: 2.2.0-RELEASE
Plugin-Id: integration-mybatis-plugin1
Plugin-Provider: StarBlues
Plugin-Class: com.mybatis.plugin1.ExamplePlugin1
```

- 推荐使用如下maven打包插件:

```
<plugin>
   <groupId>org.apache.maven.plugins</groupId>
   <artifactId>maven-assembly-plugin</artifactId>
   <version>${maven-assembly-plugin.version}</version>
   <configuration>
      <descriptorRefs>
         <descriptorRef>jar-with-dependencies</descriptorRef>
      </descriptorRefs>
      <archive>
         <manifest>
            <addDefaultImplementationEntries>true</addDefaultImplementationEntries>
            <addDefaultSpecificationEntries>true</addDefaultSpecificationEntries>
         </manifest>
         <manifestEntries>
            <Plugin-Id>${plugin.id}</Plugin-Id>
            <Plugin-Version>${plugin.version}</Plugin-Version>
            <Plugin-Provider>${plugin.provider}</Plugin-Provider>
            <Plugin-Class>${plugin.class}</Plugin-Class>
         </manifestEntries>
      </archive>
   </configuration>
   <executions>
      <execution>
         <id>make-assembly</id>
         <phase>package</phase>
         <goals>
            <goal>single</goal>
         </goals>
      </execution>
   </executions>
</plugin>
```

```
## 常见错误
### 插件未编译错误
java.lang.ClassNotFoundException: com.basic.example.plugin1.DefinePlugin
java.lang.ClassNotFoundException: com.basic.example.plugin2.DefinePlugin
该类型报错是由于插件源码没有编译成 class 文件; 需要手动编译, 保证在插件目录出现 target 文件

### 无法加载插件
请检查配置文件中的 pluginPath

```text
如果 pluginPath 配置为相当路径，请检查是否是相对于当前工作环境的目录。
```
```

如果 pluginPath 配置为绝对路径，请检查路径是否正确。
```

### Spring包冲突
如果出现Spring包冲突。可以排除Spring包。
例如:
```xml
<dependency>
<groupId>com.gitee.starblues</groupId>
<artifactId>springboot-plugin-framework</artifactId>
<version>${springboot-plugin-framework.version}</version>
<exclusions>
<exclusion>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
</exclusion>
 <exclusion>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
 </exclusion>
</exclusions>
</dependency>
```
自行选择排除冲突的包。

框架依赖的包详见 `springboot-plugin-framework/pom.xml`

插件的单独设置的配置文件无法加载

- 检查 `@ConfigDefinition("plugin1.yml")` 注解中的值是否和配置文件名称一致。
- 检查 主程序配置的 `pluginConfigFilePath` 插件配置文件路径是否正确。开发环境下请将所以插件的配置文件放在统一的目录下，然后将 `pluginConfigFilePath` 配置该

Springboot 热部署问题说明
- 如果依赖并使用了Springboot热部署功能，会导致该框架的功能无法正常运行。例如会出现注入错误、类的类型转换错误等。
- 如果存在 Springboot 热部署依赖，请自动移除。

Springboot 热部署依赖包

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-devtools</artifactId>
</dependency>
```
```

开发小技巧

小技巧

1. idea 启动主程序时, 自动编译插件包的配置 选择 File->Project Structure->Project Settings->Artifacts->点击+号->JAR->From modules whith dependencies->选择对应的插件包->确认OK

启动配置: 在Before launch 下-> 点击小+号 -> Build ->Artifacts -> 选择上一步新增的>Artifacts

版本升级

版本更新

2.2.1 版本 （2019-11-09）

1. 新增mybatis扩展包支持别名(包设置、注解设置、自定义Map映射设置)。
2. 新增可访问插件中的静态资源的扩展。
3. 新增插件模块中可定义自监听器, 用于监听启动、停止事件。
4. 修复插件卸载后, 某些场景无法删除的问题。

2.2.0 版本 （2019-10-29）

1. 修改通过路径安装插件后, 卸载后该插件后, 原插件会变成 0k 的问题。
2. 新增卸载后直接备份插件，然后删除该插件。
3. 修复卸载插件时进程被占用问题。（mybatis扩展中的bug）
4. 优化备份逻辑。
5. 升级pf4j到3.1.0版本
6. 简化插件配置集成的步骤。(针对PluginApplication)

注意：本版本更新后，会存在报错

1. 删除掉不存在的异常。【该版本去除掉多余的异常类】

2. 主程序集成框架的方式改变。具体详见 `集成框架配置说明` ，或者参考最新案例配置。

2.1.4 版本 （2019-10-24）

1. 修复插件中存在定义的事务类的话, 无法重复安装、卸载插件的bug。
2. 修复插件安装、卸载、启动、重启 无法连续操作的bug。
3. 优化了插件的安装、卸载逻辑。
4. 删除插件操作 `PluginOperator` 多余的操作。

2.1.3 版本 （2019-10-15）

在PluginUser接口新增getMainBeans方法, 用于获取Spring管理的主程序接口的实现类。

2.1.2 版本 （2019-09-18）

1. 修复使用多AOP情况, 无法加载插件类(被AOP代理的类)的bug。
2. 新增可以通过插件id获取插件中的bean的实现。详见：PluginUser->getPluginBeans(String pluginId, Class aClass)
3. 新增插件注册监听器可通过Class方式添加。案例详见: basic-eaxmple->com.basic.example.main.config.ExamplePluginListener

2.1.1 版本 （2019-09-01）

1. 插件中支持事务注解。
2. 修复重复启动插件时报错的bug。

2.1.0 版本 （2019-08-24）

1. 修复mybatis案例无法加载mapper.xml的bug。
2. 优化代码逻辑。
3. 新增插件间的通信。详见文档-使用说明->插件之间数据交互功能

2.0.3 版本 （2019-08-15）

1. 修复插件动态重新安装后, 无法访问到插件中的接口的bug。

2.0.2 版本 （2019-07-18）

1. 新增 com.gitee.starblues.integration.user.PluginUser

使用场景: 在主程序中定义了接口, 插件中存在实现了该接口的实现类, 通过PluginUser 的 getPluginBeans(接口Class) 可以获取所有插件中实现该接口的实现类。具体详见源码。

2. 新增插件bean刷新抽象类。继承它可动态获取接口实现类集合。

2.0.1 版本 （2019-07-16）

1. 修复插件的Controller无法定义一级请求路径的bug。

2.0 版本(重大版本更新)（2019-07-02）

1. 重构代码。
2. 新增扩展机制。
3. 简化依赖注入注解, 保持与SpringBoot依赖注入方式一致。
4. 新增插件工厂监听器、新增插件初始化监听器(适用于第一次启动)。
5. 新增插件包Mybatis的集成, 可在插件包中独立定义Mapper接口、Mapper xml、实体bean。

1.1 版本 （2019-06-26）

1. 新增插件注册、卸载监听器。
2. 新增可通过 PluginUser 获取插件中实现主程序中定义的接口的实现类。
3. 新增插件注册、卸载时监听器。

扩展集成

SpringBoot Mybatis 扩展

maven 仓库地址

https://mvnrepository.com/artifact/com.gitee.starblues/springboot-plugin-framework-extension-mybatis

集成步骤

主程序配置

1. 引入依赖

```
<dependency>
    <groupId>com.gitee.starblues</groupId>
    <artifactId>springboot-plugin-framework-extension-mybatis</artifactId>
    <version>${springboot-plugin-framework-extension-mybatis.version}</version>
</dependency>

<!--  自行引入 mybatis-spring-boot-starter 依赖 -->
 <dependency>
    <groupId>org.mybatis.spring.boot</groupId>
    <artifactId>mybatis-spring-boot-starter</artifactId>
    <version>${mybatis-spring-boot-starter.version}</version>
</dependency>
```

2. 集成

定义PluginApplication bean时, 新增该扩展。

```
@Bean
public PluginApplication pluginApplication(){
    PluginApplication pluginApplication = new AutoPluginApplication();
    pluginApplication.addExtension(new SpringBootMybatisExtension());
    return defaultPluginApplication;
}
```

插件程序配置

1. 引入依赖

```
<dependency>
    <groupId>com.gitee.starblues</groupId>
    <artifactId>springboot-plugin-framework-extension-mybatis</artifactId>
    <version>${springboot-plugin-framework-extension-mybatis.version}</version>
</dependency>

<!-- 自行引入 mybatis-spring-boot-starter 依赖。可用于自定义注解Sql。该依赖非必须 -->
<dependency>
    <groupId>org.mybatis.spring.boot</groupId>
    <artifactId>mybatis-spring-boot-starter</artifactId>
    <version>${mybatis-spring-boot-starter.version}</version>
</dependency>
```

2. 继承BasePlugin的类, 实现接口 com.gitee.starblues.extension.mybatis.configuration.SpringBootMybatisConfig

例如:

```
import com.gitee.starblues.extension.mybatis.configuration.SpringBootMybatisConfig;
import com.gitee.starblues.realize.BasePlugin;
import org.pf4j.PluginException;
```

```
import org.pf4j.PluginWrapper;

import java.util.HashSet;
import java.util.Set;

public class PersistenceExamplePlugin1 extends BasePlugin implements SpringBootMybatisConfig {

    private final Set<String> mybatisMapperXmlLocationsMatch = new HashSet<>();

    private final Set<String> typeAliasesPackage = new HashSet<>();
    private final Map<String, Class> typeAliasesClass = new HashMap<>();

    public PersistenceExamplePlugin1(PluginWrapper wrapper) {
        super(wrapper);
        mybatisMapperXmlLocationsMatch.add("classpath:mapper/*Mapper.xml");

        typeAliasesPackage.add("com.mybatis.plugin1.entity");
    }


    @Override
    public Set<String> mybatisMapperXmlLocationsMatch() {
        return mybatisMapperXmlLocationsMatch;
    }


    @Override
    public Map<String, Class> aliasMapping() {
        // 别名自定义映射。
        return null;
    }

    @Override
    public Set<String> typeAliasesPackage() {
        // 要统一定义别名的包集合
        return typeAliasesPackage;
    }

}
```

- 定义插件中的Mapper xml的位置。该位置的定义规则如下:

```
? 匹配一个字符
* 匹配零个或多个字符
** 匹配路径中的零或多个目录

例如:
文件路径-> file: D://xml/*PluginMapper.xml
classpath路径-> classpath: xml/mapper/*PluginMapper.xml
包路径-> package: com.plugin.xml.mapper.*PluginMapper.xml
```

- 定义别名说明

```
1. 根据包名定义实体类别名

重写如下方法。返回实体类包集合

Set<String> typeAliasesPackage(){
    return null;
}

别名规则为: 首字母小写的包名, 只支持当前包下的类, 不支持包递归。

如果该包下的类使用了别名注解@Alias, 则优先使用@Alias中定义的别名。
```

2. 根据注解定义实体类别名

直接在实体类上使用 mybatis 注解 @Alias("plugin1") 定义别名。

3. 自定义映射别名

重写如下方法。返回自定义映射集合

```
Map<String, Class> aliasMapping(){
    return null;
}
```

在Map中自定义别名。key为别名的key. 值为别名对应的实体类。

4. 优先级别

别名生效优先级别: aliasMapping > @Alias('') > typeAliasesPackage

如果三种方式都使用, 对于别名key一样的定义, 则 aliasMapping 中定义的别名最终的生效。

使用别名时建议不要三种方式混合使用，直接选中一种方式使用即可。否则不好维护。

3. 定义的Mapper 接口需要加上注解 @PluginMapper

注解位置: com.gitee.starblues.extension.mybatis.annotation.PluginMapper

例如:

```
import com.gitee.starblues.extension.mybatis.annotation.PluginMapper;
import Plugin1;
import org.apache.ibatis.annotations.Param;

import java.util.List;
@PluginMapper
public interface Plugin1Mapper {


    /**
     * 得到角色列表
     * @return List
     */
    List<Plugin1> getList();

    /**
     * 通过id获取数据
     * @param id id
     * @return Plugin2
     */
    Plugin1 getById(@Param("id") String id);

}
```

具体案例参考: example/integration-mybatis 模块。

集成Mybatis-Plus说明

由于原生 Mybatis-Plus Service 层集成的 ServiceImpl<M, T> 无法在插件中注入 BaseMapper。

因此针对此问题, 该扩展新增 ServiceImpl<M, T> 的包装类 `com.gitee.starblues.extension.mybatis.support.mybatisplus.ServiceImplWrapper` 来解决该问题, 该包装类的功能和 ServiceImpl<M, T> 功能一模一样。

用法如下:

```
@Component
public class PluginDataServiceImpl extends ServiceImplWrapper<PluginDataMapper, PluginData>
        implements PluginDataService{
```

```
    public PluginDataServiceImpl(PluginDataMapper baseMapper) {
        super(baseMapper);
    }

}
```

集成Mybatis-plus案例见 `example/integration-mybatisplus` 模块。集成的 mybatis-plus 版本为: 3.2.0

版本升级

2.2.1 版本

新增别名的定义。支持三种方式定义: 注解、包名、自定义映射。

2.2.0 版本

1. 修复xml加载，未关闭流，导致插件无法卸载的bug
2. 新增资源卸载时的接口实现

2.1.4 版本

跟随 springboot-plugin-framework 版本，未做修改。升级到 2.1.4 版本

2.1.3 版本

跟随 springboot-plugin-framework 版本的部分类修改。升级到 2.1.3 版本

2.1.1 版本

1. 新增支持 Mybatis-Plus ServiceImpl的包装类。ServiceImplWrapper。使用详见 `集成Mybatis-Plus说明`
2. 修复 Mapper.xml 中定义的 resultType 类型无法定义的bug。

2.0.3 版本

1. 修复Mapper无法注入的bug. (由于springboot-plugin-framework 2.0.3 版本升级导致)

插件静态资源访问扩展

maven 仓库地址

https://mvnrepository.com/artifact/com.gitee.starblues/springboot-plugin-framework-extension-resources

集成步骤

主程序配置

1. 引入依赖

```
<dependency>
    <groupId>com.gitee.starblues</groupId>
    <artifactId>springboot-plugin-framework-extension-resources</artifactId>
    <version>${springboot-plugin-framework-extension-resources.version}</version>
</dependency>
```

2. 集成

定义PluginApplication bean时, 新增该扩展。

```
@Bean
public PluginApplication pluginApplication(){
    PluginApplication pluginApplication = new AutoPluginApplication();
    // 新增静态资源扩展
    StaticResourceExtension staticResourceExtension = new StaticResourceExtension();
    // 设置路径访问前缀
    staticResourceExtension.setPathPrefix("static");
    // 设置缓存。可选。
    staticResourceExtension.setCacheControl(CacheControl.maxAge(1, TimeUnit.HOURS).cachePublic());
    pluginApplication.addExtension(staticResourceExtension);
    return pluginApplication;
}
```

参数设置说明

- staticResourceExtension.setPathPrefix("static"); 设置访问插件静态资源前缀。默认为: static-plugin。
- staticResourceExtension.setCacheControl(CacheControl.maxAge(1, TimeUnit.HOURS).cachePublic()); 设置缓存控制。设置为null, 则表示不缓存。

插件程序配置

1. 引入主程序的定义的pom依赖
2. 继承BasePlugin的类, 实现接口 com.gitee.starblues.extension.resources.StaticResourceConfig

例如:

```
package com.mybatis.plugin1;

import com.gitee.starblues.extension.resources.StaticResourceConfig;
import com.gitee.starblues.realize.BasePlugin;
import com.mybatis.plugin1.entity.Plugin1;
import org.pf4j.PluginWrapper;

import java.util.HashSet;
import java.util.Set;


public class ExamplePlugin1 extends BasePlugin implements StaticResourceConfig {
```

```
    private final Set<String> locations = new HashSet<>();


    public ExamplePlugin1(PluginWrapper wrapper) {
        super(wrapper);
        locations.add("classpath:static");
        locations.add("file:D:\\aa");
    }


    @Override
    public Set<String> locations() {
        return locations;
    }

}
```

该步骤主要定义插件中的静态资源的位置。该位置的定义规则如下:

```
文件路径定义方式 -> file: D://static
classpath路径定义方式-> classpath: static

例如:

locations.add("classpath:static");
locations.add("file:D:\\aa");
```

访问静态资源规则

```
http://ip:port/${pathPrefix}/${pluginId}/${staticResource}

pathPrefix: 上述 staticResourceExtension.setPathPrefix("static"); 所设置的值

pluginId: 插件id

staticResource: 所访问文件的相对物理路径。相对于上述 locations 所定义的。
```

具体案例

参考: example/integration-mybatis 模块中访问静态资源的定义。

版本升级

2.2.1 版本(基础版本)

完成初步可通过http接口访问插件中的静态资源功能。