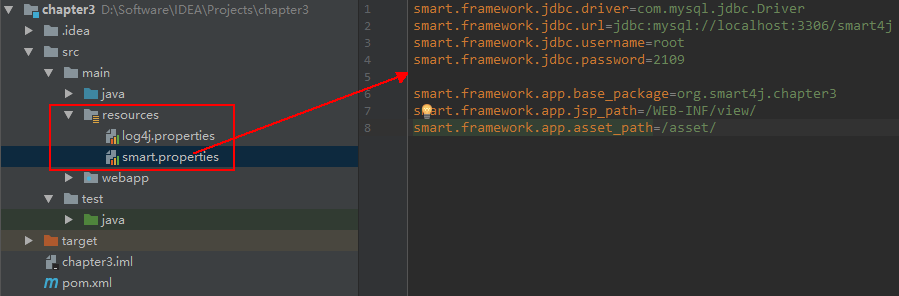
1. 定义smart-framework的pom依赖

<?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>  
  
 <groupId>org.smart4j</groupId>  
 <artifactId>smartframework</artifactId>  
 <version>1.0.0</version>  
  
 <dependencies>  
 <!--Servlet-->  
 <dependency>  
 <groupId>javax.servlet</groupId>  
 <artifactId>javax.servlet-api</artifactId>  
 <version>3.1.0</version>  
 <scope>provided</scope>  
 </dependency>  
  
 <!--JSP-->  
 <dependency>  
 <groupId>javax.servlet.jsp</groupId>  
 <artifactId>jsp-api</artifactId>  
 <version>2.2</version>  
 <scope>provided</scope>  
 </dependency>  
  
 <!--JSTL-->  
 <dependency>  
 <groupId>javax.servlet</groupId>  
 <artifactId>jstl</artifactId>  
 <version>1.2</version>  
 <scope>runtime</scope>  
 </dependency>  
  
 <!--日志-->  
 <dependency>  
 <groupId>org.slf4j</groupId>  
 <artifactId>slf4j-log4j12</artifactId>  
 <version>1.7.2</version>  
 </dependency>  
  
 <!--MySQL-->  
 <dependency>  
 <groupId>mysql</groupId>  
 <artifactId>mysql-connector-java</artifactId>  
 <version>5.1.32</version>  
 <scope>runtime</scope>  
 </dependency>  
  
 <!--Jackson-->  
 <dependency>  
 <groupId>com.fasterxml.jackson.core</groupId>  
 <artifactId>jackson-databind</artifactId>  
 <version>2.4.2</version>  
 </dependency>  
  
 <!--常用工具类-->  
 <dependency>  
 <groupId>org.apache.commons</groupId>  
 <artifactId>commons-lang3</artifactId>  
 <version>3.3.2</version>  
 </dependency>  
  
 <dependency>  
 <groupId>org.apache.commons</groupId>  
 <artifactId>commons-collections4</artifactId>  
 <version>4.0</version>  
 </dependency>  
  
 <!--DBUtils-->  
 <dependency>  
 <groupId>commons-dbutils</groupId>  
 <artifactId>commons-dbutils</artifactId>  
 <version>1.6</version>  
 </dependency>  
  
 <!--数据库连接池-->  
 <dependency>  
 <groupId>org.apache.commons</groupId>  
 <artifactId>commons-dbcp2</artifactId>  
 <version>2.0.1</version>  
 </dependency>  
  
 <!--动态代理-->  
 <dependency>  
 <groupId>cglib</groupId>  
 <artifactId>cglib</artifactId>  
 <version>3.1</version>  
 </dependency>  
  
 <!--文件上传-->  
 <dependency>  
 <groupId>commons-fileupload</groupId>  
 <artifactId>commons-fileupload</artifactId>  
 <version>1.3.1</version>  
 </dependency>  
  
 <!--Shiro, 依赖SLFJ日志框架,上面已提供-->  
 <dependency>  
 <groupId>org.apache.shiro</groupId>  
 <artifactId>shiro-core</artifactId>  
 <version>1.2.3</version>  
 </dependency>  
 </dependencies>  
  
  
</project>

1. 加载配置项

比如我chapter3项目要使用到smart-framework，则建立如下配置文件，其中smart.properties是关键配置文件。



在smart-framework中读取上述配置文件

*/\*\*  
 \* 属性文件工具类  
 \*/*public final class PropsUtil {  
  
 private static final Logger *LOGGER* = LoggerFactory.*getLogger*(PropsUtil.class);  
  
 */\*\*  
 \* 加载属性文件  
 \*/* public static Properties loadProps(String fileName){  
 Properties props = null;  
 InputStream is = null;  
  
 try {  
 //加载项目中的配置文件,如chapter3中classpath下的smart.properties  
 is = Thread.*currentThread*().getContextClassLoader().getResourceAsStream(fileName);  
 if(is == null){  
 throw new FileNotFoundException(fileName + " file is not found");  
 }  
  
 props = new Properties();  
 props.load(is);  
 } catch (IOException e) {  
 *LOGGER*.error("Load properties file failure", e);  
 } finally {  
 if(is != null){  
 try{  
 is.close();  
 } catch (IOException e){  
 *LOGGER*.error("close input stream failure", e);  
 }  
 }  
 }  
  
 return props;  
 }  
  
 */\*\*  
 \* 获取字符型属性(默认为空字符串)  
 \*/* public static String getString(Properties props, String key){  
 return *getString*(props, key, "");  
 }  
  
 */\*\*  
 \* 获取字符型属性(可指定默认值)  
 \*/* public static String getString(Properties props, String key, String defaultValue){  
 String value = defaultValue;  
 if(props.containsKey(key)){  
 value = props.getProperty(key);  
 }  
 return value;  
 }  
  
 */\*\*  
 \* 获取数值型属性(可指定默认值)  
 \*/* public static int getInt(Properties props, String key, int defaultValue){  
 int value = defaultValue;  
 if(props.containsKey(key)){  
 value = CastUtil.*castInt*(props.getProperty(key));  
 }  
 return value;  
 }  
  
 */\*\*  
 \* 获取布尔类型属性值(默认值为false)  
 \*/* public static boolean getBoolean(Properties props, String key){  
 return *getBoolean*(props, key, false);  
 }  
  
 */\*\*  
 \* 获取布尔类型值,可指定默认值  
 \*/* public static boolean getBoolean(Properties props, String key, Boolean defalutValue){  
 boolean value = defalutValue;  
 if(props.containsKey(key)){  
 value = CastUtil.*castBoolean*(props.getProperty(key));  
 }  
 return value;  
 }  
}

*/\*\*  
 \* 提供相关配置项常量  
 \*/*public interface ConfigCostant {  
  
 String *CONFIG\_FILE* = "smart.properties";  
  
 */\*\*  
 \* 数据库连接信息  
 \*/* String *JDBC\_DRIVER* = "smart.framework.jdbc.driver";  
 String *JDBC\_URL* = "smart.framework.jdbc.url";  
 String *JDBC\_USERNAME* = "smart.framework.jdbc.username";  
 String *JDBC\_PASSWORD* = "smart.framework.jdbc.password";  
  
 */\*\*  
 \* 类、资源文件路径信息  
 \*/* String *APP\_BASE\_PACKAGE* = "smart.framework.app.base\_package";  
 String *APP\_JSP\_PATH* = "smart.framework.app.jsp\_path";  
 String *APP\_ASSET\_PATH* = "smart.framework.app.asset\_path";  
  
 */\*\*  
 \* 文件上传限制  
 \*/* String *APP\_UPLOAD\_LIMIT* = "smart.framework.app.upload\_limit";  
}

*/\*\*  
 \* 属性文件助手类  
 \*/*public final class ConfigHelper {  
  
 private static final Properties *CONFIG\_PROPS* = PropsUtil.*loadProps*(ConfigCostant.*CONFIG\_FILE*);  
  
 */\*\*  
 \* 获取JDBC驱动  
 \*/* public static String getJdbcDriver(){  
 return PropsUtil.*getString*(*CONFIG\_PROPS*, ConfigCostant.*JDBC\_DRIVER*);  
 }  
  
 */\*\*  
 \* 获取JDBC URL  
 \*/* public static String getJdbcUrl(){  
 return PropsUtil.*getString*(*CONFIG\_PROPS*, ConfigCostant.*JDBC\_URL*);  
 }  
  
 */\*\*  
 \* 获取JDBC 用户名  
 \*/* public static String getJdbcUserName(){  
 return PropsUtil.*getString*(*CONFIG\_PROPS*, ConfigCostant.*JDBC\_USERNAME*);  
 }  
  
 */\*\*  
 \* 获取JDBC 密码  
 \*/* public static String getJdbcPassword(){  
 return PropsUtil.*getString*(*CONFIG\_PROPS*, ConfigCostant.*JDBC\_PASSWORD*);  
 }  
  
 */\*\*  
 \* 获取应用基础包名  
 \*/* public static String getAppBasePackage(){  
 return PropsUtil.*getString*(*CONFIG\_PROPS*, ConfigCostant.*APP\_BASE\_PACKAGE*);  
 }  
  
 */\*\*  
 \* 获取应用JSP路径  
 \*/* public static String getAppJspPath(){  
 return PropsUtil.*getString*(*CONFIG\_PROPS*, ConfigCostant.*APP\_JSP\_PATH*, "/WEB-INF/view/");  
 }  
  
 */\*\*  
 \* 获取应用静态资源路径  
 \*/* public static String getAppAssetPath(){  
 return PropsUtil.*getString*(*CONFIG\_PROPS*, ConfigCostant.*APP\_ASSET\_PATH*, "/asset/");  
 }  
  
 */\*\*  
 \* 获取应用文件上传限制  
 \*/* public static int getAppUploadLimit(){  
 return PropsUtil.*getInt*(*CONFIG\_PROPS*, ConfigCostant.*APP\_UPLOAD\_LIMIT*, 10);  
 }  
}

这样，后面项目中要用到的配置项可以使用ConfigHelper来读取。

1. 开发类加载器

*/\*\*  
 \* 类操作工具类  
 \*/*public final class ClassUtil {  
  
 private static final Logger *LOGGER* = LoggerFactory.*getLogger*(ClassUtil.class);  
  
 */\*\*  
 \* 获取类加载器  
 \*/* public static ClassLoader getClassLoader(){  
 //获取当前线程中的ClassLoader  
 return Thread.*currentThread*().getContextClassLoader();  
 }  
  
 */\*\*  
 \* 加载类,isInitialized是否执行类的静态代码块  
 \*/* public static Class<?> loadClass(String className, boolean isInitialized){  
 Class<?> cls;  
 try {  
 cls = Class.*forName*(className, isInitialized, *getClassLoader*());  
 } catch (ClassNotFoundException e) {  
 *LOGGER*.error("load class failure", e);  
 throw new RuntimeException(e);  
 }  
 return cls;  
 }  
  
 */\*\*  
 \* 加载类,isInitialized为false,可以提高类的加载性能,  
 \* 此处建议为true,否则后面使用时可能有些功能未正常加载  
 \*/* public static Class<?> loadClass(String className){  
 return *loadClass*(className, true);  
 }  
  
 */\*\*  
 \* 获取指定包名下的所有类,根据包名并将其转换为文件路径,读取class文件或者jar包,获取指定的类名取加载  
 \*/* public static Set<Class<?>> getClassSet(String packageName){  
 Set<Class<?>> classSet = new HashSet<>();  
 try {  
 //packageName org.smart4j.framework.bean  
 String newPackageName = packageName.replace(".", "/");  
 //newPackageName org/smart4j/framework/bean  
 Enumeration<URL> urls = *getClassLoader*().getResources(newPackageName);  
 while(urls.hasMoreElements()){  
//url file:/D:/Software/IDEA/Projects/smartframework/target/classes/org/smart4j/framework/bean  
 URL url = urls.nextElement();  
 if(url != null){  
 //protocol file  
 String protocol = url.getProtocol();  
 if(protocol.endsWith("file")){  
//packagePath /D:/Software/IDEA/Projects/smartframework/target/classes/org/smart4j/framework/bean  
 String packagePath = url.getPath().replaceAll("%20","");  
 *addClass*(classSet, packagePath, packageName);  
 }else if(protocol.endsWith("jar")){  
 JarURLConnection jarURLConnection = (JarURLConnection)url.openConnection();  
 if(jarURLConnection != null){  
 JarFile jarFile = jarURLConnection.getJarFile();  
 if(jarFile != null){  
 Enumeration<JarEntry> jarEntries = jarFile.entries();  
 while(jarEntries.hasMoreElements()){  
 JarEntry jarEntry = jarEntries.nextElement();  
 String jarEntryName = jarEntry.getName();  
 if(jarEntryName.endsWith(".class")){  
 String className = jarEntryName.substring(0, jarEntryName.lastIndexOf(".")).replaceAll("/", ".");  
 *doAddClass*(classSet, className);  
 }  
 }  
 }  
 }  
 }  
 }  
 }  
 } catch (Exception e) {  
 *LOGGER*.error("get class set failure", e);  
 throw new RuntimeException(e);  
 }  
 return classSet;  
 }  
  
 private static void addClass(Set<Class<?>> classSet, String packagePath, String packageName){  
 //只接受.class文件或者目录  
 File[] files = new File(packagePath).listFiles(new FileFilter() {  
 @Override  
 public boolean accept(File file) {  
 return (file.isFile() && file.getName().endsWith(".class")) || file.isDirectory();  
 }  
 });  
  
 for (File file : files){  
 //fileName 如Data.class  
 String fileName = file.getName();  
 if(file.isFile()){  
 //className Data  
 String className = fileName.substring(0, fileName.lastIndexOf("."));  
 if(StringUtil.*isNotEmpty*(packageName)){  
 //className org.smart4j.framework.bean.Data  
 className = packageName + "." + className;  
 }  
 *doAddClass*(classSet, className);  
 }else {  
 String subPackagePath = fileName;  
 if(StringUtil.*isNotEmpty*(packagePath)){  
 subPackagePath = packagePath + "/" + subPackagePath;  
 }  
 String subPackageName = fileName;  
 if(StringUtil.*isNotEmpty*(packageName)){  
 subPackageName = packageName + "." + subPackageName;  
 }  
 *addClass*(classSet, subPackagePath, subPackageName);  
 }  
 }  
 }  
  
 private static void doAddClass(Set<Class<?>> classSet, String className){  
 Class<?> cls = *loadClass*(className, false);  
 classSet.add(cls);  
 }  
}

4、定义注解

我们的目标是在控制器类上使用Controller注解，在控制器类方法上使用Action注解，在控制器类中可以使用Inject注解将服务类住进来，在服务类上使用Service注解，因此首先定义4个注解

*/\*\*  
 \* 控制器注解  
 \*/*@Target(ElementType.*TYPE*)  
@Retention(RetentionPolicy.*RUNTIME*)  
public @interface Controller {  
  
}

*/\*\*  
 \* Action方法注解  
 \*/*@Target(ElementType.*METHOD*)  
@Retention(RetentionPolicy.*RUNTIME*)  
public @interface Action {  
  
 */\*\*  
 \* 请求类型与路径  
 \*/* String value();  
}

@Target(ElementType.*FIELD*)  
@Retention(RetentionPolicy.*RUNTIME*)  
public @interface Inject {  
}

@Target(ElementType.*TYPE*)  
@Retention(RetentionPolicy.*RUNTIME*)  
public @interface Service {  
}

5、提供一个ClassHelper助手类，可以获取应用包名下的所有类、所有Controller类、Service类等，此处应用包名对应到chapter3即为org.smart4j.chapter3，使用ClassHelper封装了ClassUtil

smart.framework.app.base\_package=org.smart4j.chapter3

*/\*\*  
 \* 类操作助手类  
 \*/*public final class ClassHelper {  
  
 */\*\*  
 \* 定义类集合,用于存放所加载的类  
 \*/* private static final Set<Class<?>> *CLASS\_SET*;  
  
 static{  
 String basePackage = ConfigHelper.*getAppBasePackage*();  
 *CLASS\_SET* = ClassUtil.*getClassSet*(basePackage);  
 }  
  
 */\*\*  
 \* 获取应用包名下的所有类  
 \*/* public static Set<Class<?>> getClassSet(){  
 return *CLASS\_SET*;  
 }  
  
 */\*\*  
 \* 获取应用包名下所有Service类  
 \*/* public static Set<Class<?>> getServiceClassSet(){  
 Set<Class<?>> classSet = new HashSet<>();  
 for (Class<?> cls : *CLASS\_SET*){  
 if(cls.isAnnotationPresent(Service.class)){  
 classSet.add(cls);  
 }  
 }  
 return classSet;  
 }  
  
 */\*\*  
 \* 获取应用包下的所有Controller类  
 \*/* public static Set<Class<?>> getControllerClassSet(){  
 Set<Class<?>> classSet = new HashSet<>();  
 for(Class<?> cls : *CLASS\_SET*){  
 if(cls.isAnnotationPresent(Controller.class)){  
 classSet.add(cls);  
 }  
 }  
 return classSet;  
 }  
  
 */\*\*  
 \* 获取应用包下的所有Bean类(包括Service、Controller类等)  
 \*/* public static Set<Class<?>> getBeanClassSet(){  
 Set<Class<?>> beanClassSet = new HashSet<>();  
 beanClassSet.addAll(*getServiceClassSet*());  
 beanClassSet.addAll(*getControllerClassSet*());  
 return beanClassSet;  
 }  
  
 */\*\*  
 \* 获取应用包下名下某父类(或接口)的所有子类(或实现类)  
 \*/* public static Set<Class<?>> getClassSetBySuper(Class<?> superClass){  
 Set<Class<?>> classSet = new HashSet<>();  
 for(Class<?> cls : *CLASS\_SET*){  
 if(superClass.isAssignableFrom(cls) && !superClass.equals(cls)){  
 classSet.add(cls);  
 }  
 }  
 return classSet;  
 }  
  
 */\*\*  
 \* 获取应用包名下带有某个注解的所有类  
 \*/* public static Set<Class<?>> getClassSetByAnnotation(Class<? extends Annotation> annotationClass){  
 Set<Class<?>> classSet = new HashSet<>();  
 for(Class<?> cls : *CLASS\_SET*){  
 if(cls.isAnnotationPresent(annotationClass)){  
 classSet.add(cls);  
 }  
 }  
 return classSet;  
 }  
}

6、实现Bean容器

使用ClassHelper类可以获取所加载的类，然后我们定义一个ReflectionUtil来封装反射相关的API，对外提供更好的工具方法。

*/\*\*  
 \* 反射工具类  
 \*/*public final class ReflectionUtil {  
  
 private static final Logger *LOGGER* = LoggerFactory.*getLogger*(ReflectionUtil.class);  
  
 */\*\*  
 \* 创建实例  
 \*/* public static Object newInstance(Class<?> cls){  
 Object instance;  
  
 try {  
 instance = cls.newInstance();  
 } catch (Exception e) {  
 *LOGGER*.error("new instance failure", e);  
 throw new RuntimeException(e);  
 }  
 return instance;  
 }  
  
 */\*\*  
 \* 调用方法  
 \*/* public static Object invokeMethod(Object obj, Method method, Object... params){  
 Object result;  
 try {  
 method.setAccessible(true);  
 result = method.invoke(obj, params);  
 } catch (Exception e) {  
 *LOGGER*.error("invoke method failure", e);  
 throw new RuntimeException(e);  
 }  
 return result;  
 }  
  
 */\*\*  
 \* 设置成员变量值  
 \*/* public static void setField(Object obj, Field field, Object value){  
 try {  
 field.setAccessible(true);  
 field.set(obj, value);  
 } catch (IllegalAccessException e) {  
 *LOGGER*.error("set field failure", e);  
 throw new RuntimeException(e);  
 }  
 }  
}

7、定义BeanHelper

我们需要获取所有被Smart框架所管理的Bean类，此时需要调用ClassHelper类的getBeanClassSet方法，然后调用ReflectionUtil的newInstance方法，根据类来实例化对象，将创建的对象放在一个静态的Map<Class<?>,Object>中，可以使用Map的key(类名)取获取Bean对象。

*/\*\*  
 \* Bean助手类  
 \*/*public final class BeanHelper {  
  
 */\*\*  
 \* 定义Bean映射  
 \*/* private static final Map<Class<?>, Object> *BEAN\_MAP* = new HashMap<>();  
  
 static{  
 Set<Class<?>> beanClassSet = ClassHelper.*getBeanClassSet*();  
 for(Class<?> beanClass : beanClassSet){  
 Object obj = ReflectionUtil.*newInstance*(beanClass);  
 *BEAN\_MAP*.put(beanClass, obj);  
 }  
 }  
  
 */\*\*  
 \* 获取Bean映射  
 \*/* public static Map<Class<?>, Object> getBeanMap(){  
 return *BEAN\_MAP*;  
 }  
  
 */\*\*  
 \* 获取Bean实例  
 \*/* @SuppressWarnings("unchecked")  
 public static <T> T getBean(Class<T> cls){  
 if(!*BEAN\_MAP*.containsKey(cls)){  
 throw new RuntimeException("can not get bean by class: " + cls);  
 }  
 return (T)*BEAN\_MAP*.get(cls);  
 }  
  
 */\*\*  
 \* 设置Bean实例  
 \*/* public static void setBean(Class<?> cls, Object obj){  
 *BEAN\_MAP*.put(cls, obj);  
 }  
}

现在，BeanHelper就相当于一个Bean容器了，在Bean Map中存放了Bean类与Bean实例的映射关系。

8、实现依赖注入功能，定义IOCHelper

我们在Controller类中要使用Service成员变量，如何实例化Service?我们定义了@Inject注解，首先通过BeanHelper获取所有Bean Map，遍历这个Map，分别取出Bean类与Bean实例，通过反射获取类中所有成员变量，遍历这些成员变量，判断成员变量是否带有@Inject注解，若带有该注解，从Bean Map中获取成员变量的Bane实例，通过ReflectionUtil的setField方法设置属性值。

*/\*\*  
 \* 依赖注入助手类  
 \*/*public final class IocHelper {  
  
 //此处创建的bean的都是单例的  
 static {  
 //获取所有的bean类与bean实例之间的映射关系(简称Bean Map)  
 Map<Class<?>, Object> beanMap = BeanHelper.*getBeanMap*();  
 if(CollectionUtil.*isNotEmpty*(beanMap)){  
 //遍历BeanMap  
 for(Map.Entry<Class<?>, Object> beanEntry : beanMap.entrySet()){  
 //从BeanMap中取出Bean类与Bean实例  
 Class<?> beanClass = beanEntry.getKey();  
 Object beanInstance = beanEntry.getValue();  
 //获取bean类定义的所有成员变量(简称Bean field)  
 Field[] beanFields = beanClass.getDeclaredFields();  
 if(ArrayUtil.*isNotEmpty*(beanFields)){  
 //遍历BeanField  
 for(Field beanField : beanFields){  
 //判断当前bean Field是否带有Inject注解  
 if(beanField.isAnnotationPresent(Inject.class)){  
 //在beanMap中获取Bean field对应的实例  
 Class<?> beanFieldClass = beanField.getType();  
 Object beanFieldInstance = beanMap.get(beanFieldClass);  
 if(beanFieldInstance != null){  
 //通过反射初始化BeanField的值  
 ReflectionUtil.*setField*(beanInstance, beanField, beanFieldInstance);  
 }  
 }  
 }  
 }  
 }  
 }  
 }  
}

9、加载ControllerHelper

通过ClassHelper，我们可以获取所有定义了Controller注解的类，可以通过反射获取该类中所有带@Action注解的方法，获取Action注解中的请求表达式，进而获取请求方法与请求路径，封装成一个请求对象Request和处理对象Handler，最后将Request和Handler建立一个映射关系，放入一个Action Map，并提供一个根据请求方法和请求路径获取处理对象的方法。

定义一个Request类

*/\*\*  
 \* 封装请求信息  
 \*/*public class Request {  
  
 */\*\*  
 \* 请求方法  
 \*/* private String requestMethod;  
  
 */\*\*  
 \* 请求路径  
 \*/* private String requestPath;  
  
 public Request(String requestMethod, String requestPath) {  
 this.requestMethod = requestMethod;  
 this.requestPath = requestPath;  
 }  
  
 public String getRequestMethod() {  
 return requestMethod;  
 }  
  
 public String getRequestPath() {  
 return requestPath;  
 }  
  
 @Override  
 public boolean equals(Object obj) {  
 return EqualsBuilder.*reflectionEquals*(this, obj);  
 }  
  
 @Override  
 public int hashCode() {  
 return HashCodeBuilder.*reflectionHashCode*(this);  
 }  
}

定义一个Handler类

*/\*\*  
 \* 封装Action信息  
 \*/*public class Handler {  
  
 */\*\*  
 \* Controller类  
 \*/* private Class<?> controllerClass;  
  
 */\*\*  
 \* Action方法  
 \*/* private Method actionMethod;  
  
 public Handler(Class<?> controllerClass, Method actionMethod) {  
 this.controllerClass = controllerClass;  
 this.actionMethod = actionMethod;  
 }  
  
 public Class<?> getControllerClass() {  
 return controllerClass;  
 }  
  
 public Method getActionMethod() {  
 return actionMethod;  
 }  
}

最后定义ControllerHelper

*/\*\*  
 \* 控制器助手类  
 \*/*public final class ControllerHelper {  
  
 */\*\*  
 \* 用于存放请求与处理器的映射关系  
 \*/* private static final Map<Request, Handler> *ACTION\_MAP* = new HashMap<>();  
  
 static{  
 //获取所有的Controller类  
 Set<Class<?>> controllerClassSet = ClassHelper.*getControllerClassSet*();  
 if(CollectionUtil.*isNotEmpty*(controllerClassSet)){  
 //遍历这些Controller类  
 for(Class<?> controllerClass : controllerClassSet){  
 //获取Controller类中的定义的方法  
 Method[] methods = controllerClass.getDeclaredMethods();  
 if(ArrayUtil.*isNotEmpty*(methods)){  
 //遍历这些Controller类中的方法  
 for(Method method : methods){  
 //判断当前方法是否带有Action注解  
 if(method.isAnnotationPresent(Action.class)){  
 //从Action注解中获取URL映射规则  
 Action action = method.getAnnotation(Action.class);  
 String mapping = action.value();  
 //验证URL映射规则  
 if(mapping.matches("\\w+:/\\w\*")){  
 String[] array = mapping.split(":");  
 if(ArrayUtil.*isNotEmpty*(array) && array.length == 2){  
 //获取请求方法与路径  
 String requestMethod = array[0];  
 String requestPath = array[1];  
 Request request = new Request(requestMethod, requestPath);  
 Handler handler = new Handler(controllerClass, method);  
 //初始化ActionMap  
 *ACTION\_MAP*.put(request, handler);  
 }  
 }  
 }  
 }  
 }  
 }  
 }  
 }  
  
 */\*\*  
 \* 获取Handler  
 \*/* public static Handler getHandler(String requestMethod, String requestPath){  
 Request request = new Request(requestMethod, requestPath);  
 return *ACTION\_MAP*.get(request);  
 }  
  
 public static Map<Request, Handler> getActionMap() {  
 return *ACTION\_MAP*;  
 }  
}

10、初始化框架

通过上面的过程，我们创建了ClassHelper，BeanHelper，IOCHelper，ControllerHelper，这四个Helper类需要通过一个入口程序来加载它们，实际上是加载它们的静态代码块，加载程序时HelperLoader。

*/\*\*  
 \* 加载相应的Helper类  
 \*/*public final class HelperLoader {  
  
 */\*\*  
 \* AopHelper需要在IocHelper之前加载,因为首先需要通过AocHelper获取代理对象,  
 \* 然后才能通过IOCHelper进行依赖注入  
 \*/* public static void init(){  
 Class<?>[] classList = {ClassHelper.class, BeanHelper.class, AopHelper.class,  
 IocHelper.class, ControllerHelper.class};  
  
 for(Class<?> cls : classList){  
 ClassUtil.*loadClass*(cls.getName());  
 }  
 }  
}

现在我们可以调用HelperLoader的init方法来加载这些Helper类了。

11、请求转发器

现在我们需要编写一个Servlet，让它来处理所有的请求，从HttPServletRequest对象中获取请求方法与请求路径，通过ControllerHelper的getHandler方法获取Handler对象，随后我们可以从Handler对象中获取Controller的类，进而通过BeanHelper的getBean方法获取Controller类的实例对象。随后可以将HttpServletRequest中获取的请求参数封装为一个Param对象，从Handler中获取要调用的方法，使用ReflectionUtil调用方法，得到result，若result是一个View对象，则返回jsp页面，若是Data对象，则返回一个JSON数据。

定义Param对象

*/\*\*  
 \* 请求参数对象  
 \*/*public class Param {  
   
 private List<FormParam> formParamList;  
  
 private List<FileParam> fileParamList;  
  
 public Param(List<FormParam> formParamList){  
 this.formParamList = formParamList;  
 }  
  
 public Param(List<FormParam> formParamList, List<FileParam> fileParamList) {  
 this.formParamList = formParamList;  
 this.fileParamList = fileParamList;  
 }  
  
 */\*\*  
 \* 获取请求参数映射  
 \*/* public Map<String, Object> getParamMap(){  
 Map<String, Object> paramMap = new HashMap<>();  
 if(CollectionUtil.*isNotEmpty*(formParamList)){  
 for(FormParam formParam : formParamList){  
 String fieldName = formParam.getFieldName();  
 Object fieldValue = formParam.getFieldValue();  
 if(paramMap.containsKey(fieldName)){  
 fieldValue = paramMap.get(fieldName) + StringUtil.*SEPARATOR* + fieldValue;  
 }  
 paramMap.put(fieldName, fieldValue);  
 }  
 }  
 return paramMap;  
 }  
  
 */\*\*  
 \* 获取上传文件映射  
 \*/* public Map<String, List<FileParam>> getFileMap(){  
 Map<String, List<FileParam>> fileMap = new HashMap<>();  
 if(CollectionUtil.*isNotEmpty*(fileParamList)){  
 for(FileParam fileParam : fileParamList){  
 String fieldName = fileParam.getFieldName();  
 List<FileParam> fileParamList;  
 if(fileMap.containsKey(fieldName)){  
 fileParamList = fileMap.get(fieldName);  
 }else {  
 fileParamList = new ArrayList<>();  
 }  
 fileParamList.add(fileParam);  
 fileMap.put(fieldName, fileParamList);  
 }  
 }  
 return fileMap;  
 }  
  
 */\*\*  
 \* 获取所有上传文件  
 \*/* public List<FileParam> getFileList(String fieldName) {  
 return getFileMap().get(fieldName);  
 }  
  
 */\*\*  
 \* 获取唯一上传文件  
 \*/* public FileParam getFile(String fieldName){  
 List<FileParam> fileParamList = getFileList(fieldName);  
 if(CollectionUtil.*isNotEmpty*(fileParamList) && fileParamList.size() == 1){  
 return fileParamList.get(0);  
 }  
 return null;  
 }  
  
 */\*\*  
 \* 验证参数是否为空  
 \*/* public boolean isEmpty(){  
 return CollectionUtil.*isEmpty*(formParamList) && CollectionUtil.*isEmpty*(fileParamList);  
 }  
  
 */\*\*  
 \* 根据参数名获取String型参数值  
 \*/* public String getString(String name){  
 return CastUtil.*castString*(getParamMap().get(name));  
 }  
  
 */\*\*  
 \* 根据参数名获取long型参数值  
 \*/* public long getLong(String name){  
 return CastUtil.*castLong*(getParamMap().get(name));  
 }  
  
 */\*\*  
 \* 根据参数名获取int型参数值  
 \*/* public int getInt(String name){  
 return CastUtil.*castInt*(getParamMap().get(name));  
 }  
  
 */\*\*  
 \* 根据参数名获取double型参数值  
 \*/* public double getDouble(String name){  
 return CastUtil.*castDouble*(getParamMap().get(name));  
 }  
  
 */\*\*  
 \* 根据参数名获取String型参数值  
 \*/* public boolean getBoolean(String name){  
 return CastUtil.*castBoolean*(getParamMap().get(name));  
 }  
}

定义视图对象View

*/\*\*  
 \* 返回视图对象  
 \*/*public class View {  
  
 */\*\*  
 \* 视图路径  
 \*/* private String path;  
  
 */\*\*  
 \* 模型数据  
 \*/* private Map<String, Object> model;  
  
 public View(String path) {  
 this.path = path;  
 model = new HashMap<>();  
 }  
  
 public View addModel(String key, Object value){  
 model.put(key, value);  
 return this;  
 }  
  
 public String getPath(){  
 return path;  
 }  
  
 public Map<String, Object> getModel() {  
 return model;  
 }  
}

定义Data对象

*/\*\*  
 \* 返回数据对象  
 \*/*public class Data {  
  
 */\*\*  
 \* 模型数据  
 \*/* private Object model;  
  
 public Data(Object model){  
 this.model = model;  
 }  
  
 public Object getModel() {  
 return model;  
 }  
}

MVC框架中最核心的DispatcherServlet

*/\*\*  
 \* 请求转发器  
 \*/*@WebServlet(urlPatterns = "/\*", loadOnStartup = 0)  
public class DispatcherServlet extends HttpServlet{  
  
 @Override  
 public void init(ServletConfig servletConfig) throws ServletException {  
 //初始化相关Helper类  
 HelperLoader.*init*();  
  
 //获取ServletContext对象,用于注册Servlet  
 ServletContext servletContext = servletConfig.getServletContext();  
  
 //注册处理JSP的Servlet  
 ServletRegistration jspServlet = servletContext.getServletRegistration("jsp");  
  
 jspServlet.addMapping(ConfigHelper.*getAppJspPath*() + "\*");  
  
 //注册处理静态资源的默认Servlet  
 ServletRegistration defaultServlet = servletContext.getServletRegistration("default");  
 defaultServlet.addMapping(ConfigHelper.*getAppAssetPath*() + "\*");  
  
 //上传文件初始化  
 UploadHelper.*init*(servletContext);  
 }  
  
 @Override  
 protected void service(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {  
 //初始化  
 //这样就可以在Controller类中随时调用ServletHelper封装的Servlet API了,在Service中也可以使用  
 //因为所有的调用都来自同一个请求线程,DispatcherServlet是请求线程的入口,随后线程会先后来到Controller  
 //与Service中,我们只需要使用ThreadLocal来确保ServletHelper对象中Request和Response对象线程安全即可  
 ServletHelper.*init*(request, response);  
  
 try{  
 //获取请求方法与请求路径  
 String requestMethod = request.getMethod().toLowerCase();  
 String requestPath = request.getPathInfo();  
  
 //跳过/favicon.ico请求,只处理普通请求  
 if(requestPath.endsWith("/favicon.ico")){  
 return;  
 }  
  
 //获取Action处理器  
 Handler handler = ControllerHelper.*getHandler*(requestMethod, requestPath);  
 if(handler != null){  
 //获取Controller类及其bean实例  
 Class<?> controllerClass = handler.getControllerClass();  
 Object controllerBean = BeanHelper.*getBean*(controllerClass);  
  
 //创建请求参数对象  
 Param param;  
 if(UploadHelper.*isMultipart*(request)){  
 param = UploadHelper.*createParam*(request);  
 }else {  
 param = RequestHelper.*createParam*(request);  
 }  
  
 Object result;  
  
 //调用Action方法  
 Method actionMethod = handler.getActionMethod();  
 if(param.isEmpty()){  
 result = ReflectionUtil.*invokeMethod*(controllerBean, actionMethod);  
 }else {  
 result = ReflectionUtil.*invokeMethod*(controllerBean, actionMethod, param);  
 }  
  
 //处理Action返回值  
 if(result instanceof View){  
 //返回jsp页面  
 handleViewResult((View)result, request, response);  
 }else if(result instanceof Data){  
 handleDataResult((Data) result, response);  
 }  
 }  
 } finally {  
 ServletHelper.*destroy*();  
 }  
  
 }  
  
 private void handleViewResult(View view, HttpServletRequest request, HttpServletResponse response) throws IOException, ServletException{  
 String path = view.getPath();  
 if(StringUtil.*isNotEmpty*(path)){  
 if(path.startsWith("/")){  
 response.sendRedirect(request.getContextPath() + path);  
 }else{  
 Map<String, Object> model = view.getModel();  
 for(Map.Entry<String, Object> entry : model.entrySet()){  
 request.setAttribute(entry.getKey(), entry.getValue());  
 }  
 request.getRequestDispatcher(ConfigHelper.*getAppJspPath*() + path).forward(request, response);  
 }  
 }  
 }  
  
 private void handleDataResult(Data data, HttpServletResponse response) throws IOException{  
 //返回JSON数据  
 Object model = data.getModel();  
 if(model != null){  
 response.setContentType("application/json");  
 response.setCharacterEncoding("UTF-8");  
 PrintWriter writer = response.getWriter();  
 String json = JsonUtil.*toJson*(model);  
 writer.write(json);  
 writer.flush();  
 writer.close();  
 }  
 }  
  
}

至此，一个简单的MVC框架就开发完成了，通过这个DispatcherServlet来处理所有的请求，根据请求信息从ControllerHelper类中获取对应的Action方法，然后使用反射技术调用Action方法，同时传入具体的方法参数，最后拿到返回值并判断返回值的类型，进行相应的处理。