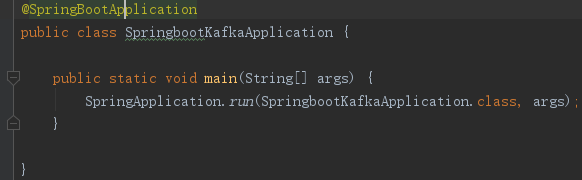
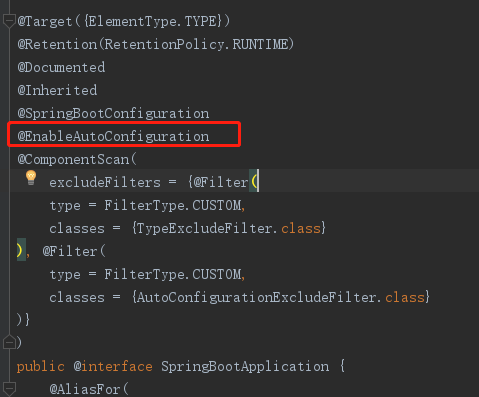
# Springboot 自动配置原理

<https://blog.csdn.net/u014745069/article/details/83820511>

首先看Springboot的启动类，该启动类上有个@SpringBootApplication注解

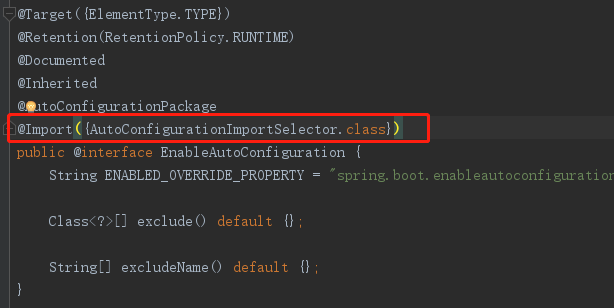
@SpringBootApplication是一个复合注解，在他上面还有一个注解@EnableAutoConfiguration 开启自动配置



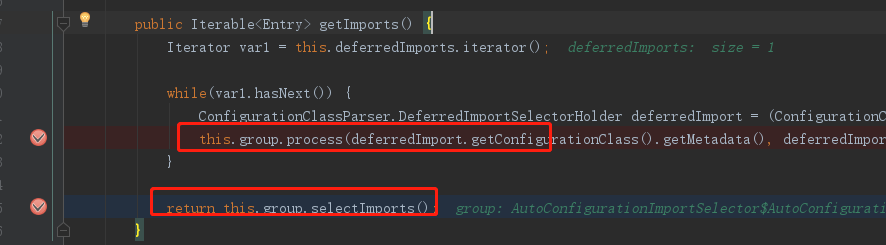
EnableAutoConfiguratin中有个注解@Import(XXXSeleor.class)

由Spring源码可知，该注解会在

invokeBeanFactoryPostProcessors(beanFactory)；根据配置类将使用注解、JAVA配置的类注入IOC容器中。【如果bean上有@Import(XXXSeleor.class)注解，在parser()中会实例化XXXSelector，将XXXSeletor绑定在该bean的Seletor属性上，并最后调用XXXSeletor的SeletorImports()】。



既然，该类的作用时机是在parse()中，我们可以看下parse(),下图的group就是AutoConfigurationImportSelect

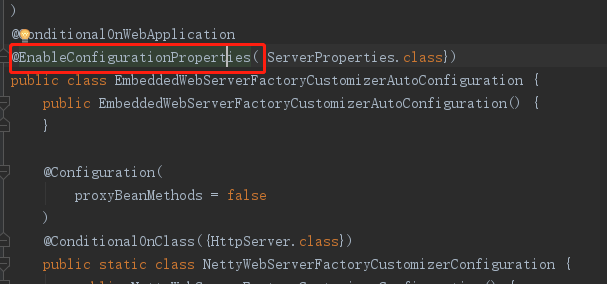


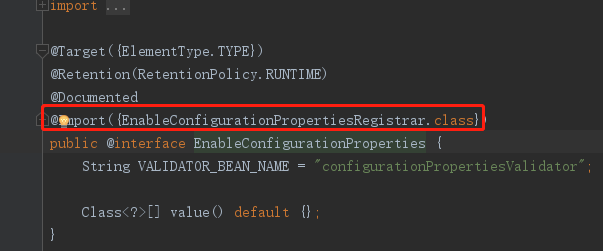
这样，我们看AutoConfigurationImportSelect中做了什么？

Process()中,会去调用SpringFactoriesLoader.loadFactoryNames(),将spring-boot-autoconfigure-2.2.9.RELEASE.jar/META-INF/spring.factories下的配置类经过过滤后保存，最后再selcetorImports()中把这些配置类注入到IOC容器中，这些配置类的作用时机需要看具体的类。本次以EmbeddedWebServerFactoryCustomizerAutoConfiguration举例。



EmbeddedWebServerFactoryCustomizerAutoConfiguration





@Import在扫描加载bean时又把EnableConfigurationPropertiesRegistrar实例化了。

根据spring源码可知，

ImportBeanDefinitionRegistrar也是在invokeBeanFactoryPostProcessors(beanFactory)；调用在之后会调用registerBeanDefinaitions()，将ServerProperties注入到Bean中。

总结：

关键：@EnableAutoConfiguration

@EnableConfigurationProperties(XXXProperties.class)

@ConfigurationProperties

Spring Boot启动的时候会通过@EnableAutoConfiguration注解找到META-INF/spring.factories配置文件中的所有自动配置类，并对其进行加载，而这些自动配置类都是以AutoConfiguration结尾来命名的，它实际上就是一个JavaConfig形式的Spring容器配置类，它能通过以Properties结尾命名的类中取得在全局配置文件中配置的属性如：server.port，而XxxxProperties类是通过@ConfigurationProperties注解与全局配置文件中对应的属性进行绑定的。

# 二、Springboot-stu

## 1. ApplicationEvent和监听器

1.创建监听器实现ApplicationListener<ApplicationEvent>

@Order(Ordered.HIGHEST\_PRECEDENCE)   
public class HelloWorldApplicationListener implements ApplicationListener<ApplicationEvent>{  
 @Override  
 public void onApplicationEvent(ApplicationEvent event) {  
 System.out.println("helloWorld first" + event.getClass().getName());  
 }  
}

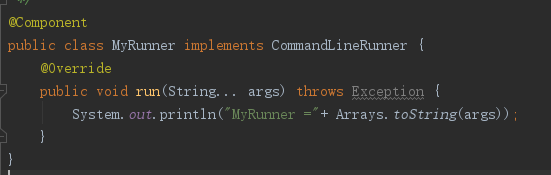
2.在main中向容器添加该监听器

springApplication.addListeners(new HelloWorldApplicationListener());

## 2. ApplicationRunner和CommandLineRunner

在容器启动完成前运行一些特定的代码，需要实现上面两个接口的任一个。具体运行时机是在ApplicationStartedEvent和ApplicationReadyEvent之间。

以CommandLineRunner为例



## 3. 自定义HttpMessageConvert

1. 实现自定义转换器SettingConvert继承AbstractHttpMessageConvert
2. 将自定义转换器加入[HttpMessageConverter](#_1._SettingConvert)中



## 4.@ControllerAdvice

**@ControllerAdvice**(basePackageClasses = AcmeController.class)

**public** **class** **AcmeControllerAdvice** **extends** **ResponseEntityExceptionHandler** {

**@ExceptionHandler**(YourException.class)

**@ResponseBody**

ResponseEntity<?> handleControllerException(HttpServletRequest request, Throwable ex) {

HttpStatus status = getStatus(request);

**return** **new** ResponseEntity<>(**new** CustomErrorType(status.value(), ex.getMessage()), status);

}

**private** HttpStatus **getStatus**(HttpServletRequest request) {

Integer statusCode = (Integer) request.getAttribute("javax.servlet.error.status\_code");

**if** (statusCode == **null**) {

**return** HttpStatus.INTERNAL\_SERVER\_ERROR;

}

**return** HttpStatus.valueOf(statusCode);

}

}

## 5. Springboot MVC（Servlet容器）

会使用就行，暂不追究其原理

### 1）扩展SpringMVC功能

继承WebMvcConfigurerAdapter可以扩展SpringMVC的功能，如跨域

@Configuration  
@EnableAsync  
@ComponentScan  
@EnableConfigurationProperties(AuditConfig.class)  
public class MediaBaseConfiguration extends WebMvcConfigurerAdapter {  
  
 @Override  
 public void addCorsMappings(CorsRegistry registry) {  
 registry.addMapping("/\*\*").allowedOrigins("\*")  
 .allowedMethods("GET", "HEAD", "POST", "PUT", "DELETE", "OPTIONS")  
 .allowCredentials(false).maxAge(3600);  
 }  
  
}

### 2）springboot使用外部servlet



## 6. CROS全局跨域配置

**@Configuration**(proxyBeanMethods = **false**)

**public** **class** **MyConfiguration** {

**@Bean**

**public** WebMvcConfigurer **corsConfigurer**() {

**return** **new** WebMvcConfigurer() {

**@Override**

**public** **void** **addCorsMappings**(CorsRegistry registry) {

registry.addMapping("/api/\*\*");

}

};

}

}

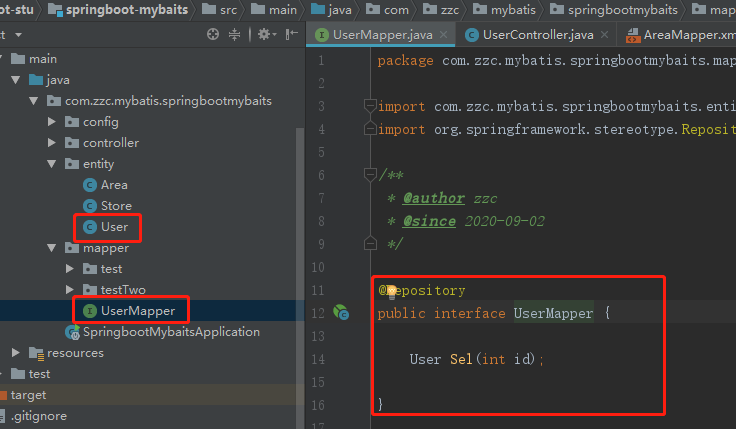
## 7. springboot+mybatis

### 1. 数据库配置

**# 单数据源配置  
spring:  
 datasource:  
 username: root  
 password: 123456  
 url: jdbc:mysql://localhost:3306/book?useUnicode=true&characterEncoding=utf-8&useSSL=true&serverTimezone=UTC  
 driver-class-name: com.mysql.jdbc.Driver**

**mybatis:  
 mapper-locations: classpath:mapping/\*\*.xml  
 type-aliases-package: com.zzc.mybatis.springbootmybaits.entity**

### Entity和Mapper



### Mapper.xml

位置：resource/mapping/UserMapper.xml

**Namespace对应着UserMapper.java的全类名**

**Id对应着其中的方法名**

**<?xml version="1.0" encoding="UTF-8"?>  
 <!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN" "http://mybatis.org/dtd/mybatis-3-mapper.dtd">  
<mapper namespace="com.zzc.mybatis.springbootmybaits.mapper.UserMapper">  
  
<select id="Sel" resultType="com.zzc.mybatis.springbootmybaits.entity.User">  
 select \* from user where uid = #{id}  
 </select>  
  
</mapper>**

### 开启mapper扫描

**@MapperScan("com.zzc.mybatis.springbootmybaits.mapper")**

## 8. springboot+mybaits+多数据源

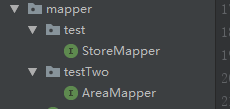
### 1.配置多数据源

**spring:  
 datasource:  
 test1:  
 jdbc-url: jdbc:mysql://localhost:3306/book?serverTimezone=CTT&useUnicode=true&characterEncoding=utf8  
 driverClassName: com.mysql.cj.jdbc.Driver  
 username: root  
 password: 123456  
 test2:  
 jdbc-url: jdbc:mysql://localhost:3306/test?serverTimezone=CTT&useUnicode=true&characterEncoding=utf8  
 driverClassName: com.mysql.cj.jdbc.Driver  
 username: root  
 password: 123456**

**# 扫描xml文件位置  
mybatis:  
 mapper-locations: classpath:mapping/\*\*.xml**

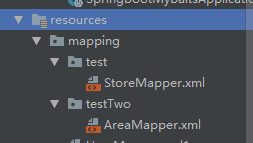
### 2.Entity和Mapper

Mapper目录结构



### Mapper.xml

目录结构



### 数据源配置类

数据源分为主数据源和次数据源

区别就是扫描的包不用，读取的数据库配置不同，主数据源的bean上加@Primary优先加载。

1.指定Dao层扫描

@MapperScan(basePackages = "com.zzc.mybatis.springbootmybaits.mapper.test", sqlSessionFactoryRef = "test1SqlSessionFactory")

### 取消启动类自动加载数据源配置

**@SpringBootApplication(exclude ={DataSourceAutoConfiguration.class})**

## 9. springboot+jpa

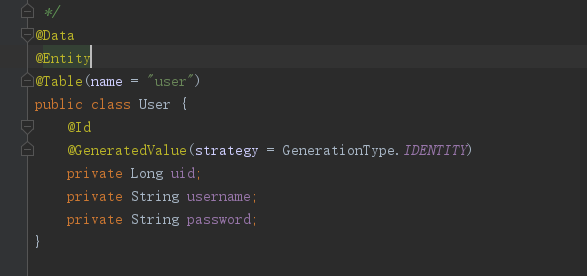
### 1.环境搭建

#### 1）配置数据库和jpa的基础功能

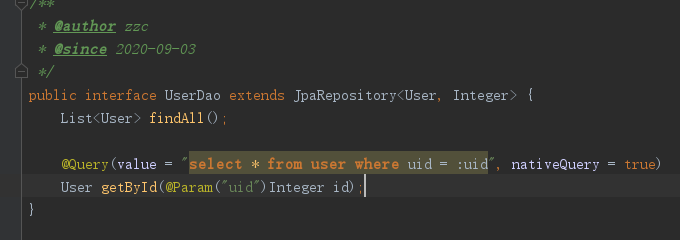
#数据源配置  
spring.datasource.type=com.alibaba.druid.pool.DruidDataSource  
spring.datasource.driverClassName=com.mysql.jdbc.Driver  
spring.datasource.url=jdbc:mysql://localhost:3306/book?useSSL=false&useUnicode=true&characterEncoding=utf-8&autoReconnect=true&serverTimezone=Asia/Shanghai  
spring.datasource.username=root  
spring.datasource.password=123456

#批量插入  
spring.jpa.properties.hibernate.order\_inserts=true  
#批量更新  
spring.jpa.properties.hibernate.order\_updates=true  
# 批量插入每组最大个数  
spring.jpa.properties.hibernate.jdbc.batch\_size=500  
spring.jpa.properties.hibernate.enable\_lazy\_load\_no\_trans=true

#### 2）Entity

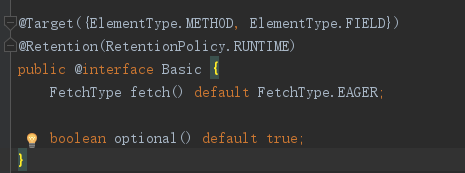


#### 3）Dao



JpaRepository<K,T> K：table对应的java类，T：主键类型

### 2. @Basic



实体类与数据库字段最简单的映射。

Fetch：属性的加载机制

FetchType.EAGER：即时加载（默认）

FetchType.LAZY: 懒加载

Optional用于指定属性是否可空

如果属性上没有@Basic，它会自动加上@Basic，并使用默认值

### 3. @Version

### 4. QueryDSL

## 10. jpa实现xml方式联表查询

原生jpa并没有提供向mybatis通过mapper.xml方式去查询sql。但我们可以通过EntityManager去读取xml执行sql。

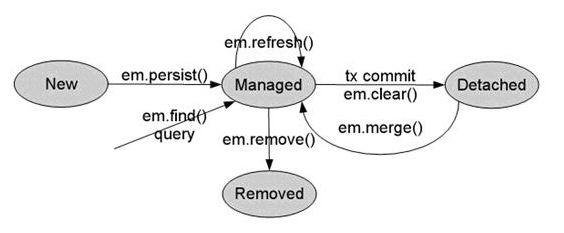
**EntityManager.createNativeQuery(String sql);**

我们要做的是按一定的模板配置xml，使用模板解析器解析成sql语句执行。

### 1）EntityManager和Query

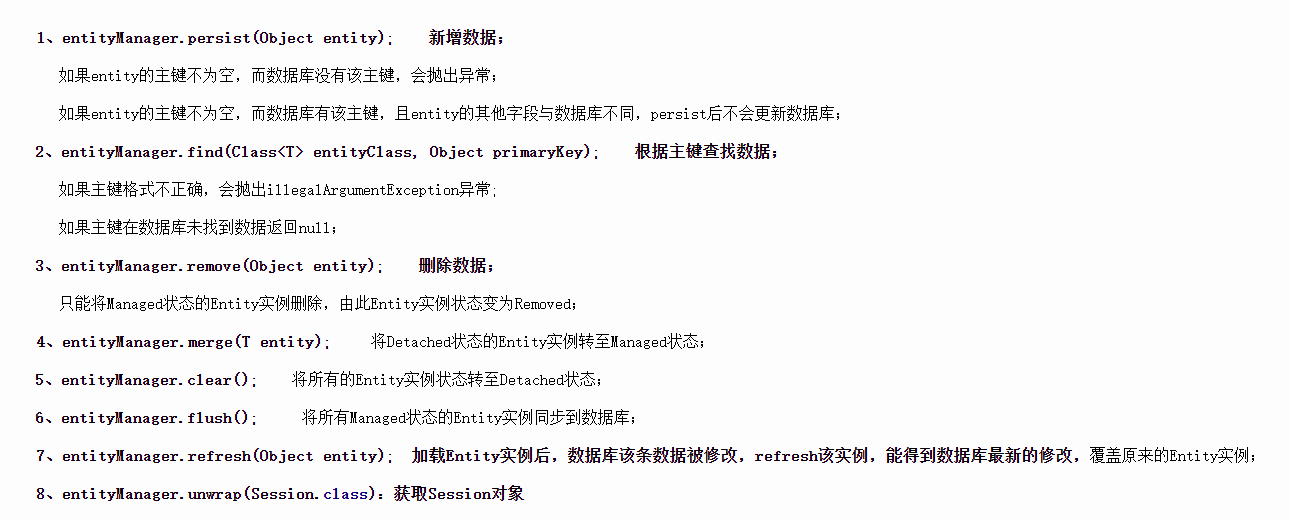
EntityManager是JPA中用于增删改查的接口。是连接内存中的JAVA对象和数据库中数据存储的桥梁。

#### 1.1 Entity声明周期：



* New（瞬时对象）：尚未有id，还未与Presistence Context建立关联的对象
* Managed（持久化受管对象）：有id值，已经和Persistence Context建立了关联的对象
* Detached（游离态离线对象）：有id值，但没有和Persistence Context建立关联的对象
* Removed：删除的对象，有id值，尚且和Persistence Context有关联，但是已经准备好从数据库中删除

#### 1.2 方法



#### 1.3 Query

Query query = entityManager.createNativeQuery("select id, name, age from  t\_user");

List rows = query.getResultList();

rows是Object[]集合;

Query query = entityManager.createNativeQuery("select id, name, age from  t\_user");

query.unwrap(SQLQuery.class).setResultTransformer(Transformers.ALIAS\_TO\_ENTITY\_MAP);

 List rows = query.getResultList();

rows是Map集合，效率比Object[]低，以损失查询性能为代码使代码可读性更高，处理结果更方便。

### 2）使用dom4j读取XML

### 3）VelocityContext

## 11. 日志信息

### 1. 日志框架出现场景

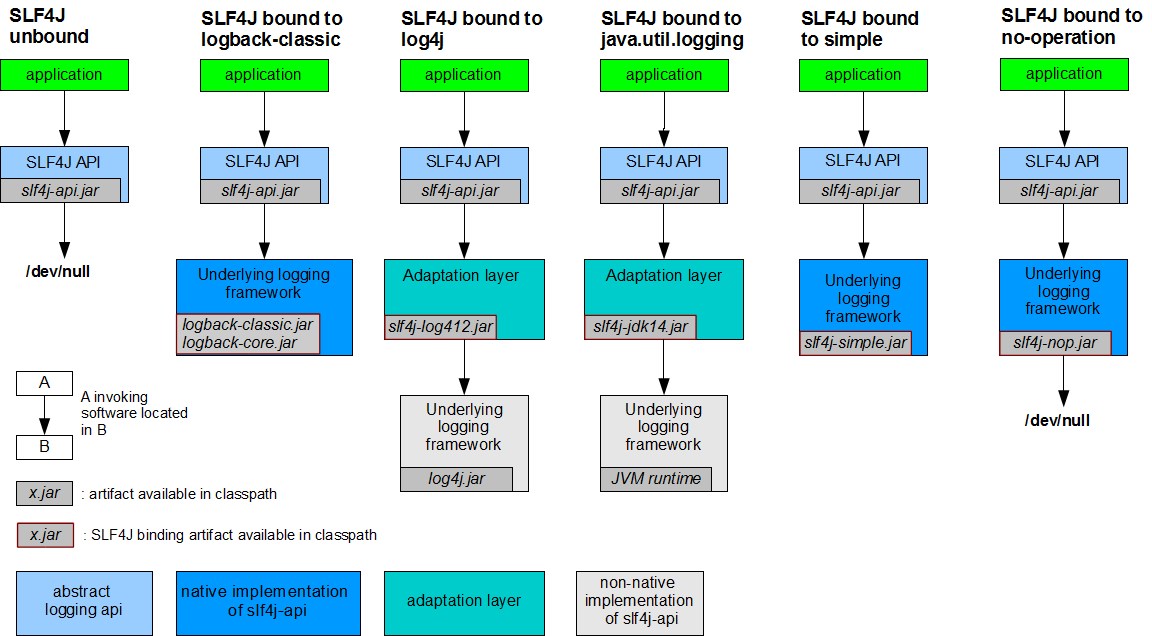
1.张三在系统中为了调式写了大量的System.out，老板要求在上线的时候需要删除，这时就必须一行一行手动注释掉；后面为了调式方便又的加回来。

2.张三开发了一个logger-simple.jar，通过配置日志级别来决定是否打印日志。使用一段时间后，需要优化的地方很多，这是又开发了一个logger-good.jar，但要应用到系统中由于接口很多都被改动了，又需要手动地一个个去改。如果以后在优化一版，又得重复一次大量繁琐无用的工作。

3.张三决定开发一个log-abstract日志抽象层决定日志的接口，log-1.jar实现接口。这样每次只需要改动log实现层的jar包就行了，这样将日志实现和应用解耦了，由接口负责两者的沟通。

Slf4j（抽象）---logback（实现）

### 2. slf4j使用各种实现框架方式



有的实现框架不是slf4j的实现层，因此需要一个适配器去连接。

Slf4j的每个实现框架都有自己的配置文件，具体应用时，还是需要配置实现框架的配置文件。

### 3. 应用中的每个框架都使用logback日志框架

一个ssm框架的应用中每个层级的日志框架都不相同

Spring: commons-logging

Hibernate: jboss+logging

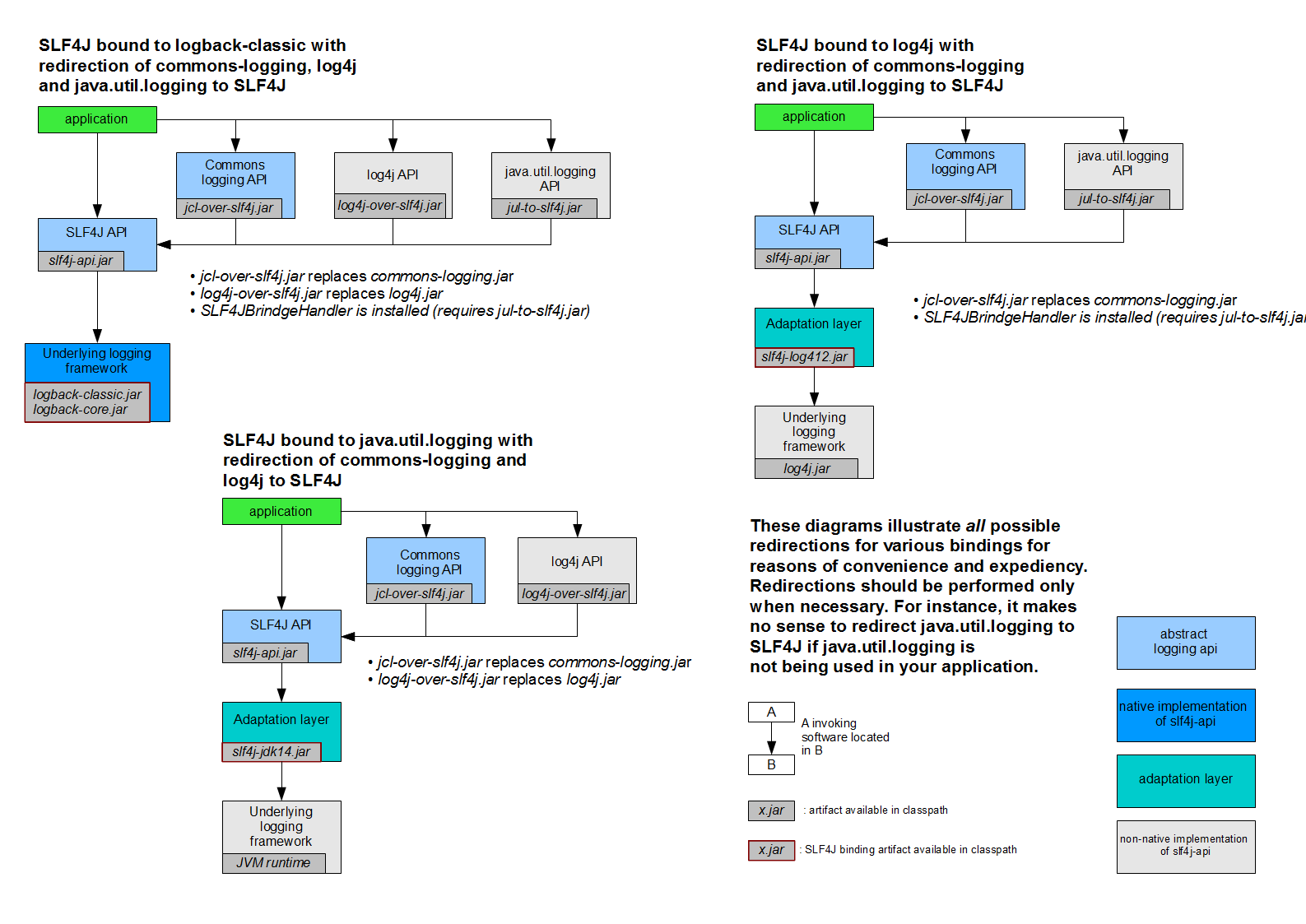
如何让一个应用中的各个框架都是用slf4j+logback

如图1所示：

**想要一个commons-logging使用slf4j+logback，首先需要排除commons-loggin.jar，引入jcl-over-slf4j.jar。**

总结：

1. 将系统中其他日志框架排除
2. 用中间包替换原有的日志框架
3. 导入slf4j的其他实现



### springboot日志配置

logging.file : 可以是盘符路径，也可以是项目根路径

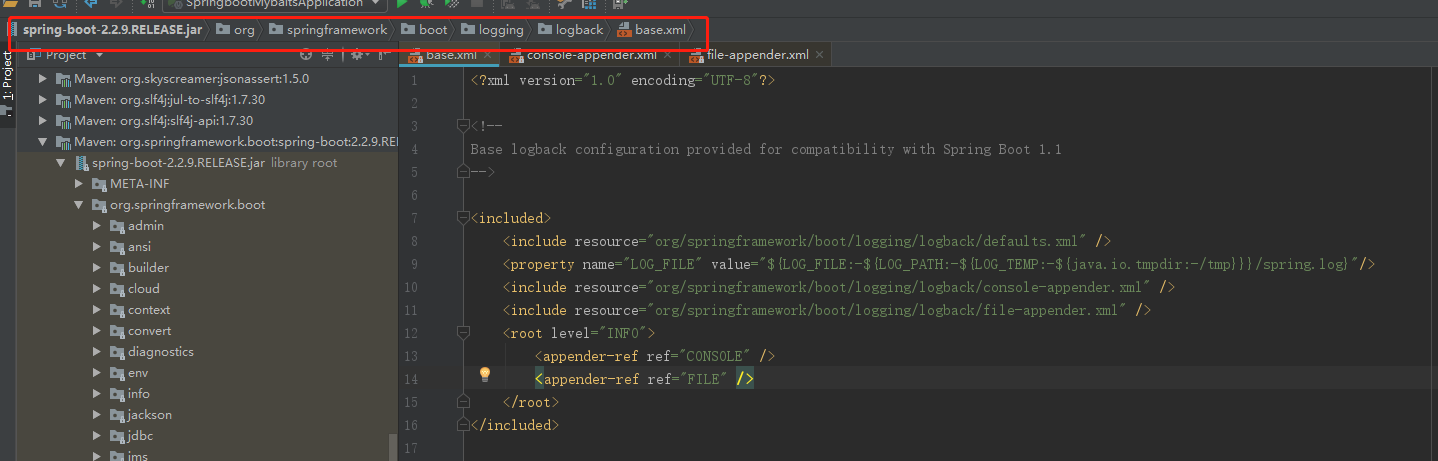
logging.path: /xxx.log 项目所处的盘符路径， 如：D:/xxx.log

logging.pattern.console: 控制台输出日志的格式

logging.pattern.file: 文件输出日志格式

* %d: 表示日期
* %thread: 表示线程名
* %-5level: 级别从左显示5个字符宽度
* %msg: 日志消息
* %n: 是换行符

### Springboot 日志的配置文件



Springboot在spring-boot-2.2.9.RELEASE.jar中配置了logback的默认配置。

在resource下配置：

logback.xml 直接被logback读取

logback-spring.xml交由spring框架处理

**日志模板：**



# 三、代码

## 1. SettingConvert

public class SettingConverter extends AbstractHttpMessageConverter<Object> {  
 /\*\*  
 \* 定义字符编码，防止乱码  
 \*/  
 private static final Charset DEFAULT\_CHARSET = Charset.forName("UTF-8");  
  
 /\*\*  
 \* 新建自定义的媒体类型  
 \*/  
 public SettingConverter() {  
 super(new MediaType("application", "json", DEFAULT\_CHARSET));  
 }  
  
 /\*\*  
 \* 表明只处理Settings这个类  
 \*/  
 @Override  
 protected boolean supports(Class<?> aClass) {  
 return true;  
 }  
  
 /\*\*  
 \* 重写readInternal方法，处理请求的数据  
 \*/  
 @Override  
 protected Object readInternal(Class<? extends Object> aClass, HttpInputMessage httpInputMessage)

throws IOException, HttpMessageNotReadableException {  
 final String temp = StreamUtils.copyToString(httpInputMessage.getBody(), DEFAULT\_CHARSET);  
 Object object = JSON.parseObject(temp, aClass);  
 return object;  
// if (temp.contains(TRAINING.name())) {  
// return JSONObject.parseObject(temp, new TypeReference<SettingInfoRequest<SettingInfo>>(){});  
// } else if (temp.contains(MODELDUMP.name())) {  
// return JSONObject.parseObject(temp, new TypeReference<SettingInfoRequest<DumpSettings>>(){});  
// }  
// return null;  
 }  
  
 @Override  
 protected void writeInternal(Object user, HttpOutputMessage httpOutputMessage) throws IOException, HttpMessageNotWritableException {  
 httpOutputMessage.getBody().write("11".getBytes());  
 }  
}

## 2. DataSourceConfig1

@Configuration  
@MapperScan(basePackages = "com.zzc.mybatis.springbootmybaits.mapper.test", sqlSessionFactoryRef = "test1SqlSessionFactory")  
public class DataSourceConfig1 {  
  
 // 表示这个数据源是默认数据源  
 @Primary  
 @Bean(name = "test1DataSource")  
 @ConfigurationProperties(prefix = "spring.datasource.test1")  
 public DataSource getDateSource1()  
 {  
 return DataSourceBuilder.create().build();  
 }  
  
 @Bean(name = "test1SqlSessionFactory")  
 @Primary  
 // @Qualifier表示查找Spring容器中名字为test1DataSource的对象  
 public SqlSessionFactory test1SqlSessionFactory(@Qualifier("test1DataSource") DataSource datasource)  
 throws Exception  
 {  
 SqlSessionFactoryBean bean = new SqlSessionFactoryBean();  
 bean.setDataSource(datasource);  
 bean.setMapperLocations(  
 // 设置mybatis的xml所在位置  
 new PathMatchingResourcePatternResolver().getResources("classpath\*:mapping/test/\*.xml"));  
 return bean.getObject();  
 }  
  
 @Bean("test1SqlSessionTemplate")  
 @Primary  
 public SqlSessionTemplate test1SqlSessionTemplate(  
 @Qualifier("test1SqlSessionFactory") SqlSessionFactory sessionFactory)  
 {  
 return new SqlSessionTemplate(sessionFactory);  
 }  
  
}

加载b