闲来无事看代码，无意间看到一段同事写的源码，在Spring Boot项目中用Mybatis访问多个数据源的数据。颇感性趣，感觉可以学到很多知识，而且代码有进一步改进的空间。

# 思路

1. 核心，Spring Boot中下面这个抽象类，提供了重要支撑

org.springframework.jdbc.datasource.lookup.AbstractRoutingDataSource

功能：

1. 实现了DataSource接口
2. 可设置默认数据源

public void setDefaultTargetDataSource(Object defaultTargetDataSource)

1. 可设置多个目标数据源

public void setTargetDataSources(Map<Object, Object> targetDataSources)

1. 根据key切换数据源，实现时，key是一个与数据源对应的数据源名字串

protected Object determineCurrentLookupKey()

1. 围绕这个核心

继承AbstractRoutingDataSource类，自定义数据源路由类RoutingDataSource。只需在需要特定数据源的时候，通过determineCurrentLookupKey()提供特定的数据源名字就能自动切换到对应的TargetDataSource上，如果没有映射到的数据源名字会使用默认的数据源DefaultTargetDataSource

1. DataSource Bean的自动注册

Spring Boot的ImportBeanDefinitionRegistrar提供了自动注册回调接口

void registerBeanDefinitions(AnnotationMetadata importingClassMetadata, BeanDefinitionRegistry registry)

实现该接口CustomDataSourceBeanRegister，完成对RoutingDataSource默认数据源和多数据源的设置；完成Bean的注册功能

在Spring Boot启动类中使用@Import(CustomDataSourceBeanRegister.class)或@Import({CustomDataSourceBeanRegister.class})触发Spring Boot自动回调注册接口，完成多DataSource Bean的注册

1. 为程序提供切换数据源的编码方式

引用切换数据源注解@CustomDataSource(name=”自定义数据源名”)，Service中只需使用Mapper接口的方法上使用@CustomDataSource即可完成数据源切换

注解实现通过AOP实现，在调用该注解的方法前改变目标数据源名字，进而关联RoutingDataSource的determineCurrentLookupKey方法返回改变后的数据源名字

离开方法后，恢复原数据源名字

有了思路，接下来开整

# 开始编码

## 核心

“DataSource路由”，RoutingDataSource继承AbstractRoutingDataSource，重写protected Object determineCurrentLookupKey()方法，提供切换的数据源映射名字即可。

## 那么切换的数据源映射名字，从何而来？

从自定义注解CustomDataSource来，开发者可以通过自定义注解的name属性指定要切换的数据源名字。

## 注解的名字如何传递给RoutingDataSource

一个保存当前注解名字的容器DataSourceContextHolder，需要具备如下能力

* 通过静态方法获取当前切换的数据源名字
* 通过静态方法设置当前要切换的数据源名字
* 数据源名字的切换要线程安全

可以通过同步方法，但是更简单的是使用ThreadLocal，每个线程都各自保存各自的变量，没有同步开销，以空间换时间，还好多数据源的线程应该不多。好了，代码片段如下：

|  |
| --- |
| public class DataSourceContextHolder {  /\* 当前目标数据源名字 \*/  private static final ThreadLocal<String> contextHolder = new ThreadLocal<String>();  public static void setTargetDataSourceName(String name) {  contextHolder.set(name);  }  public static String getTargetDataSourceName() {  return contextHolder.get();  }  public static void clearTargetDataSourceName() {  contextHolder.remove();  }  } |

## 注解的名字如何传给容器DataSourceContextHolder

Spring Boot AOP闪亮登场，在使用注解的方法处，截获注解，获取要切换的数据源名字，类的名字是...CustomDataSourceAop！

Maven依赖

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

代码片段

|  |
| --- |
| @Component  @Aspect  // 保证该AOP在@Transactional之前执行  @Order(-1)  public class CustomDataSourceAop {  /\*\*  \* 切换目标数据源  \* 在进入方法之前,更改目标数据源  \* @param point 切点,即用@CustomDataSource注解的目标方法  \* @param ds 目标方法上的注解对象  \*/  @Before("@annotation(ds)")  public void changeTargetDataSource(JoinPoint point, CustomDataSource ds) throws Throwable {  String dsId = ds.name();  DataSourceContextHolder.setTargetDataSourceName(ds.name());  }  /\*\*  \* 清除目标数据源  \* 方法执行后,清除本次切换的目标数据源  \*/  @After("@annotation(ds)")  public void restoreTargetDataSource(JoinPoint point, CustomDataSource ds) {  DataSourceContextHolder.clearTargetDataSourceName();  }  } |

至此，RoutingDataSource的代码片段

|  |
| --- |
| /\*\*  \* SpringBoot Bean注册中使用的自定义多数据源选择类  \* 一方面实现了DataSource接口,一方面可以根据目标数据源名字选择对应DataSource的功能  \*/  public class RoutingDataSource extends AbstractRoutingDataSource {  @Override  protected Object determineCurrentLookupKey() {  return DataSourceContextHolder.getTargetDataSourceName();  }  } |

## 如何将RoutingDataSource注册到SpringBoot中

创建CustomDataSourceBeanRegister实现ImportBeanDefinitionRegistrar接口，在

void registerBeanDefinitions(AnnotationMetadata importingClassMetadata, BeanDefinitionRegistry registry)

方法中，完成注册工作

## Bean的注册需要哪些步骤

创建Bean定义对象GenericBeanDefinition

调用setBeanClass设置Bean类型

调用setSynthetic设置为true标识这个Bean是手动发起合成的，如果不关心可不设置

调用getPropertyValues得到属性设置对象

通过属性设置对象，为Bean注入必要的属性值，属性名是Bean中公共set方法后面的名字，第一个字母小写

通过入参BeanDefinitionRegistry对象的registerBeanDefinition方法，将Bean注册到SpringBoot中

最后通过@Import注解导入ImportBeanDefinitionRegistrar接口实现类，触发Spring Boot调用注册接口

## DataSource的注册需要哪些步骤

创建GenericBeanDefinition对象beanDefinition

beanDefinition.setBeanClass设置Bean类型为RoutingDataSource.class

beanDefinition. setSynthetic(true)标识这个Bean是手工合成的，区别于Spring Boot自动注册的Bean

beanDefinition.getPropertyValues()得到RoutingDataSource属性注入对象mpv

通过mpv.addPropertyValue("defaultTargetDataSource", defaultDataSource)注入默认数据源对象

通过mpv.addPropertyValue("targetDataSources", targetDataSources)注入名值对形式的多数据源映射

ImportBeanDefinitionRegistrar接口registerBeanDefinitionsBean方法的入参DefinitionRegistry对象值为registry

通过registry.registerBeanDefinition("dataSource", beanDefinition)完成Spring Boot上下文注册

最后@Import({CustomDataSourceBeanRegister.class})，一般在Spring Boot入口类上使用

几点说明：

上述属性注入的名字为什么是defaultTargetDataSource、targetDataSource

之前说过，注入的属性名字就是Bean字段值设置方法set之后的名，查看AbstractRoutingDataSource的源码，可看到如下内容

|  |
| --- |
| public void setTargetDataSources(Map<Object, Object> targetDataSources) {  this.targetDataSources = targetDataSources;  }  public void setDefaultTargetDataSource(Object defaultTargetDataSource) {  this.defaultTargetDataSource = defaultTargetDataSource;  } |

通过上面代码，确定了这两个属性的名字

为什么Bean的名字是dataSource

这个是约定俗成，DataSource对象自然是dataSource，当然也可以注册为其它名字。只是如果通过名字getBean()时，要提供正确的名字。

上面步骤中，提到了注入默认数据源对象

## 这个默认数据源对象如何获取

默认数据源配置

利用Spring Boot的spring.datasource来设置默认数据源的选项

## 如何将配置绑定到DataSource对象上

## SpringBoot中DataSource自动注册代码分析

首先想到的是Spring Boot内部如何自动完成了DataSource的注册，一番查找，总结如下，比较无聊，就是按线索一条一条跟下去

入口点org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration完成数据源的自动注册

关注如下这段代码

|  |
| --- |
| @Configuration(  proxyBeanMethods = false  )  @Conditional({DataSourceAutoConfiguration.PooledDataSourceCondition.class})  @ConditionalOnMissingBean({DataSource.class, XADataSource.class})  @Import({Hikari.class, Tomcat.class, Dbcp2.class, Generic.class, DataSourceJmxConfiguration.class})  protected static class PooledDataSourceConfiguration {  protected PooledDataSourceConfiguration() {  }  } |

来看条件类DataSourceAutoConfiguration.PooledDataSourceCondition.class

|  |
| --- |
| static class PooledDataSourceCondition extends AnyNestedCondition {  PooledDataSourceCondition() {  super(ConfigurationPhase.PARSE\_CONFIGURATION);  }  @Conditional({DataSourceAutoConfiguration.PooledDataSourceAvailableCondition.class})  static class PooledDataSourceAvailable {  PooledDataSourceAvailable() {  }  }  } |

再看DataSourceAutoConfiguration.PooledDataSourceAvailableCondition.class

|  |
| --- |
| static class PooledDataSourceAvailableCondition extends SpringBootCondition {  PooledDataSourceAvailableCondition() {  }  public ConditionOutcome getMatchOutcome(ConditionContext context, AnnotatedTypeMetadata metadata) {  Builder message = ConditionMessage.forCondition("PooledDataSource", new Object[0]);  return DataSourceBuilder.findType(context.getClassLoader()) != null ? ConditionOutcome.match(message.foundExactly("supported DataSource")) : ConditionOutcome.noMatch(message.didNotFind("supported DataSource").atAll());  }  } |

DataSourceBuilder.findType返回非空条件成立

|  |
| --- |
| private static final String[] DATA\_SOURCE\_TYPE\_NAMES = new String[] { "com.zaxxer.hikari.HikariDataSource","org.apache.tomcat.jdbc.pool.DataSource", "org.apache.commons.dbcp2.BasicDataSource" };  public static Class<? extends DataSource> findType(ClassLoader classLoader) {  for (String name : DATA\_SOURCE\_TYPE\_NAMES) {  try {  return (Class<? extends DataSource>) ClassUtils.forName(name, classLoader);  }  catch (Exception ex) {  // Swallow and continue  }  }  return null;  } |

这里可以看到按DATA\_SOURCE\_TYPE\_NAMES顺序加载了DataSource实现类，HikariDataSource是首选实现类

接着来看@Import({Hikari.class, Tomcat.class, Dbcp2.class, Generic.class, DataSourceJmxConfiguration.class})，这些Bean都在org.springframework.boot.autoconfigure.jdbc.DataSourceConfiguration中，来看首选的Hikari.class的Bean实现，其它类似。

|  |
| --- |
| @Configuration(  proxyBeanMethods = false  )  @ConditionalOnClass({HikariDataSource.class})  @ConditionalOnMissingBean({DataSource.class})  @ConditionalOnProperty(  name = {"spring.datasource.type"},  havingValue = "com.zaxxer.hikari.HikariDataSource",  matchIfMissing = true  )  static class Hikari {  Hikari() {  }  @Bean  @ConfigurationProperties(  prefix = "spring.datasource.hikari"  )  HikariDataSource dataSource(DataSourceProperties properties) {  HikariDataSource dataSource = (HikariDataSource)DataSourceConfiguration.createDataSource(properties, HikariDataSource.class);  if (StringUtils.hasText(properties.getName())) {  dataSource.setPoolName(properties.getName());  }  return dataSource;  }  } |

通过@Import首先引入Hikari.class

将Hikari作为DataSource的首选Bean

其中Bean的实现方法HikariDataSource dataSource(DataSourceProperties properties)是随后要借鉴和参考的重要代码其中DataSourceConfiguration.createDataSource是关键

|  |
| --- |
| protected static <T> T createDataSource(DataSourceProperties properties, Class<? extends DataSource> type) {  return properties.initializeDataSourceBuilder().type(type).build();  } |

至此，SpringBoot DataSource Bean已经完成自动创建

## 实现自己的DataSource实例创建代码

从上面总结中看到SpringBoot中DataSource Bean创建使用了@ConfiguraionProperties注解将配置数据自动绑定到DataSource对象相应属性上，但是SpringBoot中的这些注解内容都是固定的。

|  |
| --- |
| @ConfigurationProperties(  prefix = "spring.datasource.hikari"  ) |

多数据源DataSource实现，配置是动态的，需要动态修改@ConfigurationProperties的内容。又一番折腾。

## 动态修改@ConfiguraitonProperties内容

Java中注解类是通过代理Proxy生成的，sun.reflect.annotation.AnnotationInvocationHandler.AnnotationInvocationHandler是代理调用句柄，可见范围是包。注解成员名值对，保存在代理调用句柄的memberValues字段中，类型是Map<String,Object>。因此可以通过代理调用对象反射memberValues字段，来修改@ConfigurationProperties的prefix。

动态修改步骤：

通过反射方法对象method.getAnnotation(ConfigurationProperties.class)得到注解对象

Proxy.getInvocationHandler(注解对象)得到调用句柄对象AnnotationInvocationHandler

调用句柄对象反射getDeclaredField得到memberValues字段对象，打开访问权限

取memberValues字段对象值，put(“prefix”,新值)，完成注解修改

还是看看代码吧

|  |
| --- |
| // 取DataSource工厂方法（就是创建DataSource实例的方法）上的注解@ConfigurationProperties  ConfigurationProperties configAnnotation = method.getAnnotation(ConfigurationProperties.class);  // 获取代理实例所持有的 InvocationHandler，Java注解由代理实现，代理处理对象是AnnotationInvocationHandler  InvocationHandler invocationHandler = Proxy.getInvocationHandler(configAnnotation);  // 获取 AnnotationInvocationHandler 的 memberValues 字段  Field memberValuesField = invocationHandler.getClass().getDeclaredField("memberValues");  // 打开访问权限  memberValuesField.setAccessible(true);  // 获取 memberValues (成员和值的映射表，目标注解的信息都在 memberValues 中)  Map<String, Object> memberValues = (Map<String, Object>) memberValuesField.get(invocationHandler);  // 1)、动态获取 TargetAnnotation.targetAnnotationClassField 标注的注解属性值  Object prefixValue = memberValues.get("prefix");  log.info("修改前@ConfigurationProperties prefix:{}", prefixValue);  // 2)、动态修改 TargetAnnotation.targetAnnotationClassField 标注的注解属性值  memberValues.put("prefix", prefix);  // 3)、再次获取修改后的 TargetAnnotation.targetAnnotationClassField 标注的注解属性值  prefixValue = memberValues.get("prefix");  log.info("修改后@ConfigurationProperties prefix:{}", prefixValue); |

## 如何将修改的注解内容绑定到DataSource上

坑，到处是坑，又遇到棘手的问题。实际上前面那些问题都不是一下子解决的，都是看的头大才搞定的，这个又是一个头疼的问题。

先看看前面的SpringBoot DataSource自动注册代码

|  |
| --- |
| properties.initializeDataSourceBuilder().type(type).build(); |

这里的properties是一个DataSourceProperties对象，@ConfigurationProperties注解就是绑定到这个对象上，然后再绑定到DataSource对象上。

Ok，先看注解配置如何绑定到DataSourceProperties

SpringBoot使用Binder将指定配置绑定到对象上，简单来说就是将配置转换成Bean。@ConfigurationProperties注解实际上就是将prefix指定前缀下的配置绑定到DataSourceProperties对象上。

来看@ConfigurationProperties的实现，org.springframework.boot.context.properties. ConfigurationPropertiesBindingPostProcessor

|  |
| --- |
| public Object postProcessBeforeInitialization(Object bean, String beanName) throws BeansException {  //这里的bean可以是任意类型，比如DataSource  bind(ConfigurationPropertiesBean.get(this.applicationContext, bean, beanName));  return bean;  } |

其中bind会一直调用到如下代码

|  |
| --- |
| BindResult<?> bind(ConfigurationPropertiesBean propertiesBean) {  Bindable<?> target = propertiesBean.asBindTarget();  ConfigurationProperties annotation = propertiesBean.getAnnotation();  BindHandler bindHandler = getBindHandler(target, annotation);  return getBinder().bind(annotation.prefix(), target, bindHandler);  } |
| private Binder getBinder() {  if (this.binder == null) {  this.binder = new Binder(getConfigurationPropertySources(), getPropertySourcesPlaceholdersResolver(),getConversionService(), getPropertyEditorInitializer(), null,  ConfigurationPropertiesBindConstructorProvider.INSTANCE);  }  return this.binder;  } |

## 可以通过Binder.bind(prefix, Bindable<?> target, BindHandler)来完成绑定

上网查看一下Binder用法，一般都是这种格式

|  |
| --- |
| SomeCalss beans = Binder.get(env).bind(prefix, SomeClass.class).orElse(null); |

因此SpringBoot源码中的new Binder（因为太多没有的参数）可以考虑通过上面代码来替换，而Binder.get(env)源码为

|  |
| --- |
| public static Binder get(Environment environment, BindHandler defaultBindHandler) {  Iterable<ConfigurationPropertySource> sources = ConfigurationPropertySources.get(environment);  PropertySourcesPlaceholdersResolver placeholdersResolver = new PropertySourcesPlaceholdersResolver(environment);  return new Binder(sources, placeholdersResolver, null, null, defaultBindHandler);  } |

再来看ConfigurationPropertiesBean.get(this.applicationContext, bean, beanName)

|  |
| --- |
| public static ConfigurationPropertiesBean get(ApplicationContext applicationContext, Object bean, String beanName) {  //通过Spring上下文找到Bean定义的方法，就是定位创建bean的工厂方法，返回类型为Bean类型的方法就满足工厂方法  Method factoryMethod = findFactoryMethod(applicationContext, beanName);  return create(beanName, bean, bean.getClass(), factoryMethod);  } |
| private static ConfigurationPropertiesBean create(String name, Object instance, Class<?> type, Method factory) {  ConfigurationProperties annotation = findAnnotation(instance, type, factory, ConfigurationProperties.class);  if (annotation == null) {  return null;  }  Validated validated = findAnnotation(instance, type, factory, Validated.class);  Annotation[] annotations = (validated != null) ? new Annotation[] { annotation, validated }: new Annotation[] { annotation };  //bindType如何创建看这里，有工厂方法就可以了  ResolvableType bindType = (factory != null) ? ResolvableType.forMethodReturnType(factory) : ResolvableType.forClass(type);  //bindTarget如何创建看这里，有了bindType，从工厂方法上获取注解ConfigurationProperties，然后用数组传进去，就可以了  Bindable<Object> bindTarget = Bindable.of(bindType).withAnnotations(annotations);  if (instance != null) {  bindTarget = bindTarget.withExistingValue(instance);  }  return new ConfigurationPropertiesBean(name, instance, annotation, bindTarget);  } |
| private ConfigurationPropertiesBean(String name, Object instance, ConfigurationProperties annotation,Bindable<?> bindTarget) {  this.name = name;  this.instance = instance;  this.annotation = annotation;  this.bindTarget = bindTarget;  this.bindMethod = BindMethod.forType(bindTarget.getType().resolve());  } |

## 至此，已经解决如何Bindable<?> target创建

DataSource工厂方法

工厂方法上的@ConfigurationProperties注解

然后就可以创建出Bindable<Object> bindTarget

|  |
| --- |
| // 取DataSource工厂方法（就是创建DataSource实例的方法）上的注解@ConfigurationProperties  ConfigurationProperties configAnnotation = method.getAnnotation(ConfigurationProperties.class);  // 绑定工厂返回类型DataSource类型  ResolvableType bindType = ResolvableType.forMethodReturnType(method);  // DataSource类型绑定注解  Bindable<Object> bindTarget = Bindable.of(bindType).withAnnotations(new Annotation[] { configAnnotation });  // 绑定具体对象  bindTarget = bindTarget.withExistingValue(dataSource); |

再来看BindHandler，最终ConfigurationPropertiesBinder.getBindHandler

|  |
| --- |
| private <T> BindHandler getBindHandler(Bindable<T> target, ConfigurationProperties annotation) {  List<Validator> validators = getValidators(target);  //用这个Handler就可以，其它的都是一些带条件处理的Handler，用不到  BindHandler handler = new IgnoreTopLevelConverterNotFoundBindHandler();  if (annotation.ignoreInvalidFields()) {  handler = new IgnoreErrorsBindHandler(handler);  }  if (!annotation.ignoreUnknownFields()) {  UnboundElementsSourceFilter filter = new UnboundElementsSourceFilter();  handler = new NoUnboundElementsBindHandler(handler, filter);  }  if (!validators.isEmpty()) {  handler = new ValidationBindHandler(handler, validators.toArray(new Validator[0]));  }  for (ConfigurationPropertiesBindHandlerAdvisor advisor : getBindHandlerAdvisors()) {  handler = advisor.apply(handler);  }  return handler;  } |

至此BindHandler也解决了

## 于是动态替换@ConfigurationProperties内容以及绑定的方法终于新鲜出炉

|  |
| --- |
| /\*\*  \* 指定数据源配置绑定到数据源对象上  \* 动态设置注解@ConfigurationProperties{prefix=...}  \* 利用Proxy代理AnnotationInvocationHandler，修改注解内容  \* 利用SpringBoot Binder，实现配置到数据源的动态绑定  \* @param dataSource 数据源对象  \* @param method 数据源工厂方法  \* @param prefix 指定数据源配置前缀  \*/  protected void bindDataSourceConfiguration(DataSource dataSource, Method method, String prefix)throws NoSuchMethodException, NoSuchFieldException, IllegalAccessException {  // 取DataSource工厂方法（就是创建DataSource实例的方法）上的注解@ConfigurationProperties  ConfigurationProperties configAnnotation = method.getAnnotation(ConfigurationProperties.class);  // 获取代理实例所持有的 InvocationHandler，Java注解由代理实现，代理处理对象是AnnotationInvocationHandler  InvocationHandler invocationHandler = Proxy.getInvocationHandler(configAnnotation);  // 获取 AnnotationInvocationHandler 的 memberValues 字段  Field memberValuesField = invocationHandler.getClass().getDeclaredField("memberValues");  // 打开访问权限  memberValuesField.setAccessible(true);  // 获取 memberValues (成员和值的映射表，目标注解的信息都在 memberValues 中)  Map<String, Object> memberValues = (Map<String, Object>) memberValuesField.get(invocationHandler);  // 1)、动态获取 TargetAnnotation.targetAnnotationClassField 标注的注解属性值  Object prefixValue = memberValues.get("prefix");  log.info("修改前@ConfigurationProperties prefix:{}", prefixValue);  // 2)、动态修改 TargetAnnotation.targetAnnotationClassField 标注的注解属性值  memberValues.put("prefix", prefix);  // 3)、再次获取修改后的 TargetAnnotation.targetAnnotationClassField 标注的注解属性值  prefixValue = memberValues.get("prefix");  log.info("修改后@ConfigurationProperties prefix:{}", prefixValue);  // 绑定工厂返回类型DataSource类型  ResolvableType bindType = ResolvableType.forMethodReturnType(method);  // DataSource类型绑定注解  Bindable<Object> bindTarget = Bindable.of(bindType).withAnnotations(new Annotation[] { configAnnotation });  // 绑定具体对象  bindTarget = bindTarget.withExistingValue(dataSource);  // 绑定事件处理对象  BindHandler handler = new IgnoreTopLevelConverterNotFoundBindHandler();  // 将特定的DataSource配置绑定到数据源上  Binder.get(env).bind(configAnnotation.prefix(), bindTarget, handler);  } |

## 自定义数据源配置规范

|  |
| --- |
| custom:  datasource:  names: cus1,cus2,...  cus1:  #与默认配置spring.datasource格式一样  cus2:  #与默认配置spring.datasource格式一样 |

整个实现中，DataSource配置与SpringBoot自动注册的配置有些差异，就是数据源特异性的配置没有单独放在各自数据源特有的结点下而是都放在了一起。比如，SpringBoot会将HikariDataSource配置放在spring.datasource.hikari结点下，而多数据源配置都放在了spring.datasource下，同理也适用于custom.datasource.<数据源名>结点。

另外生成的数据源对象没有设置数据源名字，因为不同实现的数据源数据源名设置方法不一样，更具体的可以参看DataSourceConfiguration中各DataSource Bean的源码。

## 通过type来区分数据源实现类

多数据源实现中，没有像SpringBoot那样通过各自的结点名，将配置绑定到特定的数据源上。因此，可以通过type以及统一的结点名来将所有类型的DataSource配置绑定到特定的数据源上。

如果不指定type，会使用HikariDataSource

## Bean注册也可通过@Bean注解实现

|  |
| --- |
| @Configuration  @Slf4j  public class CustomDataSourceBean extends CustomDataSourceBeanBase {  @Bean  public DataSource dataSource(){  try {  log.info("初始化customDataSourceBean");  DataSourceFactory.build().setEnvironment(env);  DataSource defaultDataSource = defaultDataSource();  String defaultDataSourceName = "dataSource";  Map<String, DataSource> customDataSourceMap = customDataSource();  if (defaultDataSource == null && customDataSourceMap.size() == 0) {  throw new IllegalAccessError("没有定义任何数据源");  }  // 增加有效数据源名称  DataSourceContextHolder.addValidDataSourceName(defaultDataSourceName);  for (String key : customDataSourceMap.keySet()) {  DataSourceContextHolder.addValidDataSourceName(key);  }  RoutingDataSource dataSource=new RoutingDataSource();    Map<Object, Object> targetDataSources = Maps.newHashMap();  if(defaultDataSource!=null){  targetDataSources.put(defaultDataSourceName,defaultDataSource);  dataSource.setDefaultTargetDataSource(defaultDataSource);  }else{  dataSource.setDefaultTargetDataSource(customDataSourceMap.values().iterator().next());  }  targetDataSources.putAll(customDataSourceMap);  dataSource.setTargetDataSources(targetDataSources);  log.info("初始化customDataSourceBean完毕");  return dataSource;  }catch(Exception e){  log.error("创建自定义数据源Bean失败:{}",e.getMessage(),e);  throw new IllegalAccessError(String.format("创建自定义数据源Bean失败:%s",e));  }  }  } |

# 代码模块

## 核心类RoutingDataSource

简单，但非常关键，根据上下文数据源名字，返回相应的数据源实例

通过DataSourceContextHolder获取当前要切换的数据源名字

## 自定义注解@CustomDataSource

拦截使用自定义数据源的方法，方法调用前根据注解内容设定目标数据源名，调用后恢复之前的数据源名。对下为DataSourceContextHolder提供数据源名称

## 数据源名切换上下文类DataSourceContextHolder

对上接收@CustomDataSource注解中的数据源名字，对下为核心类RoutingDataSource提供当前要切换的数据源名字

## 数据源工厂类DataSourceFactory

通过Enviroment根据给定的配置前缀创建DataSource实例

通过修改工厂方法的@ConfigurationProperties的prefix动态绑定数据源配置

为CustomDataSourceBean和CustomDataSourceBeanRegister创建相应的数据源实例提供封装和辅助

## 数据源Bean基类CustomDataSourceBeanBase

为数据源注册方式类CustomDataSourceBeanRegister和数据源Bean方式类CustomDataSourceBean提供公共方法，包括：Spring环境注入、按照默认数据源配置创建默认数据源、按照自定义数据源配置创建自定义数据源

这里，通过DataSourceFactory来创建数据源

数据源注册类和数据源Bean功能一样，都是注入Bean，可二者选一

## 数据源注册类

创建RoutingDataSource对象，并绑定默认数据源和自定义数据源，将RoutingDataSource注册到SpringBoot中

## 配置数据源Bean类

通过SpringBoot配置方式创建Bean——@Configuration与@Bean，创建RoutingDataSource，并绑定默认数据源和自定义数据源

## 关系图

RoutingDataSource

CustomDataSourceBeanBase

CustomDataSourceBeanRegister

CustomDataSourceBean

DataSourceFactory

IDataSourceFactory

DataSourceContextHolder

CustomDataSource

使用DataSource的方法

# 测试

## RoutingDataSource改进

测试JdbcTemplate注入不同数据源时，发现注解中使用不同数据源名返回的数据源对象都一样，经查看AbstractRoutingDataSource代码，有如下代码

|  |
| --- |
| protected DataSource determineTargetDataSource() {  Assert.notNull(this.resolvedDataSources, "DataSource router not initialized");  Object lookupKey = this.determineCurrentLookupKey();  DataSource dataSource = (DataSource)this.resolvedDataSources.get(lookupKey);  if (dataSource == null && (this.lenientFallback || lookupKey == null)) {  dataSource = this.resolvedDefaultDataSource;  }  if (dataSource == null) {  throw new IllegalStateException("Cannot determine target DataSource for lookup key [" + lookupKey + "]");  } else {  return dataSource;  }  } |

这个才是获取真正DataSource的代码，封装一下

|  |
| --- |
| /\*\*真正的数据源，如果不使用该方法，在JdbcTemplate上使用注解会导致始终返回同一数据源对象的问题，  \* 但是Mybatis mapper不存在这个问题 \*/  public DataSource getTargetDataSource(){  return super.determineTargetDataSource();  } |
| if(dataSource instanceof RoutingDataSource){  targetDataSource=((RoutingDataSource)dataSource).getTargetDataSource();  } |

在注入代码中，进行如下转换，不过这种方式明显是一种紧耦合。这个问题在Mybatis的mapper上不存在，mapper取的应该是Connection？查看了一下AbstractRoutingDataSource代码，在获取Connection时用的是determineTargetDataSource。

为了稍稍解耦，通过反射来调用determineTargetDataSource

|  |
| --- |
| public final class DataSourceHelper {  /\*\*多数据源切换，获取实际的数据源\*/  public static DataSource getTargetDataSource(DataSource dataSource) throws NoSuchMethodException,InvocationTargetException,IllegalAccessException {  if(dataSource instanceof AbstractRoutingDataSource){  Class clz = dataSource.getClass();  //因为determineTargetDataSource在AbstractRoutingDataSource类中，需要回溯到这个类上面  while(clz!=AbstractRoutingDataSource.class){  clz=clz.getSuperclass();  }  Method method = clz.getDeclaredMethod("determineTargetDataSource");  method.setAccessible(true);  return (DataSource) method.invoke(dataSource);  }else{  return dataSource;  }  }  } |

然后JdbcTemplate Bean这样使用

|  |
| --- |
| @CustomDataSource(name = "cus1")  @Bean  @Qualifier("cus1")  JdbcTemplate cus1JdbcTemplate(DataSource dataSource) throws SQLException{  try{  return new JdbcTemplate(DataSourceHelper.getTargetDataSource(dataSource));  }catch(Exception e){  throw new SQLException(e.getCause());  }  } |

## 配置

在Application.yml进行如下配置，共三个配置，数据库地址、用户都一样，只是默认连接的数据库不一样，分别是def、cus1、cus2。def、cus1、cus2下都创建了cus表，字段只有name和value，分别插入了一条记录，name值都是”name”，value分别为对应的库名。

|  |
| --- |
| #默认数据源  spring:  datasource:  username: test  password: mysql  #MySQL5用的驱动url是com.mysql.jdbc.Driver，MySQL6以后用的是com.mysql.cj.jdbc.Driver  driver-class-name: com.mysql.cj.jdbc.Driver  url: jdbc:mysql://192.168.128.27:3306/def?characterEncoding=UTF-8&useSSL=false&serverTimezone=Asia/Shanghai  maximum-pool-size: 2 #最大连接数，小于等于0会被重置为默认值10；大于零小于1会被重置为minimum-idle的值  connection-timeout: 60000 #连接超时时间:毫秒，小于250毫秒，否则被重置为默认值30秒  minimum-idle: 1 #最小空闲连接，默认值10，小于0或大于maximum-pool-size，都会重置为maximum-pool-size  idle-timeout: 500000 #空闲连接超时时间，默认值600000（10分钟），大于等于max-lifetime且max-lifetime>0，会被重置为0；不等于0且小于10秒，会被重置为10秒。  max-lifetime: 540000 #连接最大存活时间.不等于0且小于30秒，会被重置为默认值30分钟.设置应该比mysql设置的超时时间短  connection-test-query: SELECT 1 #连接测试查询  #自定义数据源  custom:  datasource:  names: cus1,cus2  cus1:  username: test  password: mysql  driver-class-name: com.mysql.cj.jdbc.Driver  url: jdbc:mysql://192.168.128.27:3306/cus1?characterEncoding=UTF-8&useSSL=false&serverTimezone=Asia/Shanghai&allowPublicKeyRetrieval=true  maximum-pool-size: 2  connection-timeout: 60000  minimum-idle: 1  idle-timeout: 500000  max-lifetime: 540000  connection-test-query: SELECT 1  cus2:  username: test  password: mysql  driver-class-name: com.mysql.cj.jdbc.Driver  url: jdbc:mysql://192.168.128.27:3306/cus2?characterEncoding=UTF-8&useSSL=false&serverTimezone=Asia/Shanghai&allowPublicKeyRetrieval=true  maximum-pool-size: 2  connection-timeout: 60000  minimum-idle: 1  idle-timeout: 500000  max-lifetime: 540000  connection-test-query: SELECT 1 |

## JdbcTemplate Bean配置

|  |
| --- |
| @Configuration  public class CustomDataSourceConfig {  @Bean  @Primary  JdbcTemplate defaultJdbcTemplate(DataSource dataSource) throws SQLException {  try{  return new JdbcTemplate(DataSourceHelper.getTargetDataSource(dataSource));  }catch(Exception e){  throw new SQLException(e.getCause());  }  }  @CustomDataSource(name = "cus1")  @Bean  @Qualifier("cus1")  JdbcTemplate cus1JdbcTemplate(DataSource dataSource) throws SQLException{  try{  return new JdbcTemplate(DataSourceHelper.getTargetDataSource(dataSource));  }catch(Exception e){  throw new SQLException(e.getCause());  }  }  @CustomDataSource(name = "cus2")  @Bean  @Qualifier("cus2")  JdbcTemplate cus2JdbcTemplate(DataSource dataSource) throws SQLException {  try{  return new JdbcTemplate(DataSourceHelper.getTargetDataSource(dataSource));  }catch(Exception e){  throw new SQLException(e.getCause());  }  }  } |

## Mybatis mapper配置

bean

|  |
| --- |
| @Data  public class Cus {  private String name;  private String value;  } |

接口

|  |
| --- |
| public interface CusMapper {  List<Cus> allCus();  } |

Mapper sql

|  |
| --- |
| <!-- 此处与接口类地址对应 -->  <mapper namespace="org.xynok.lab.mapper.CusMapper">  <!-- 此处与接口方法名对应 指定参数类型与返回结果类型-->  <select id="allCus" resultType="org.xynok.lab.dao.Cus">  select \* from cus  </select>  </mapper> |

服务组件

|  |
| --- |
| @Service  public class DbService {  @Autowired  private CusMapper cusMapper;  public List<Cus> allCus(){  return cusMapper.allCus();  }  @CustomDataSource(name="cus1")  public List<Cus> allCus1(){  return cusMapper.allCus();  }  @CustomDataSource(name="cus2")  public List<Cus> allCus2(){  return cusMapper.allCus();  }  } |

application.yml配置

|  |
| --- |
| mybatis:  mapper-locations: classpath\*:org/xynok/lab/mapper/\*.xml |

Jdbctemplate测试

|  |
| --- |
| @Autowired  JdbcTemplate JdbcTemplateDefault;  @Autowired  @Qualifier("cus1")  JdbcTemplate JdbcTemplateCus1;  @Autowired  @Qualifier("cus2")  JdbcTemplate JdbcTemplateCus2;  @Test  public void jdbcTemplate() {  String sql = "select name,value from cus";    JdbcTemplateDefault.query(sql, new RowCallbackHandler() {  @Override  public void processRow(ResultSet rs) throws SQLException {  Cus cus=new Cus();  cus.setName(rs.getString("name"));  cus.setValue(rs.getString("value"));  System.out.println(String.format("Default jdbcTemplate result:%s",cus));  }  });  JdbcTemplateCus1.query(sql, new RowCallbackHandler() {  @Override  public void processRow(ResultSet rs) throws SQLException {  Cus cus=new Cus();  cus.setName(rs.getString("name"));  cus.setValue(rs.getString("value"));  System.out.println(String.format("Cus1 jdbcTemplate result:%s",cus));  }  });  JdbcTemplateCus2.query(sql, new RowCallbackHandler() {  @Override  public void processRow(ResultSet rs) throws SQLException {  Cus cus=new Cus();  cus.setName(rs.getString("name"));  cus.setValue(rs.getString("value"));  System.out.println(String.format("Cus2 jdbcTemplate result:%s.",cus));  }  });  } |

Mybatis测试

|  |
| --- |
| @Autowired  DbService dbService;  @Test  void multiDs(){  List<Cus> list=dbService.allCus();  list.forEach(item->{  System.out.println(item);  });  list=dbService.allCus1();  list.forEach(item->{  System.out.println(item);  });  list=dbService.allCus2();  list.forEach(item->{  System.out.println(item);  });  } |

# 搭建MySQL环境

Docker容器方便快捷，只需具备Docker环境，便可利用已有的镜像资源来快速搭建所需的环境，实乃快速构建之最佳选择。

## 最小化安装CentOS7.2虚拟机

只有CentOS7.2的iso，网络设置为NAT

## 安装Docker

### 配置IP

1. VMware>编辑>虚拟网络编辑器

选择VMnet8 NAT模式

NAT设置>查看网关>192.168.128.2

记录，随后配置ip时用到

1. 编辑网卡配置文件

*# cd /etc/sysconfig/network-script*

*# vi ifcfg-eno16777736*

BOOTPROTO=static

IPADDR=192.168.128.27

GATEWAY=192.168.128.2

*# service network restart*

*# ip a*

### 安装

SecureCRT ssh连接虚拟机

指令如下

mount /dev/cdrom /mnt

yum makecache fast

yum install -y yum-utils device-mapper-persistent-data lvm2

yum -y update

**yum-config-manager** --add-repo http://mirrors.aliyun.com/docker-ce/linux/centos/docker-ce.repo

yum -y install docker

systemctl enable docker

指令含义

挂接本地光盘

刷新安装源

安装Docker所需的依赖包

更新Linux，如果不升级会导致后续MySQL连接出现10060未知异常

配置Docker国内镜像源

安装Docker

配置系统启动自动运行Docker

## 安装MySQL

### 上传镜像

/usr/local/docker/image-bak/mysql-image.tar

mysql-image.tar，这个是以前使用docker时导出的

恢复镜像

docker load -**i** mysql-image.tar

docker images

### 创建MySQL挂接目录

**cd** /usr/**local**/docker

**mkdir** mysql

**cd** mysql

**mkdir** **conf**

**cd** **conf**

touch mysql.cnf

**mkdir** data

**mkdir** logs

### 运行容器

docker run -p 3306:3306 --name mysql \

-v /usr/local/docker/mysql/conf:/etc/mysql/conf.d \

-v /usr/local/docker/mysql/logs:/var/log/mysql \

-v /usr/local/docker/mysql/data:/var/lib/mysql \

-e MYSQL\_ROOT\_PASSWORD=mysql \

-d mysql

run 运行

-p 3306:3306，主机port与容器port映射

--name mysql，容器名

-v <主机目录>:<容器目录>，容器目录挂接到主机目录

-e MYSQL\_ROOT\_PASSWORD=mysql，配置MySQL root密码

-d mysql，daemon方式运行，mysql是镜像名

### 创建mysql用户

*#进入容器，-it表示终端交互，mysql容器名，bash shell*

docker exec -it mysql bash

*#进入mysql,这里版本时8.0.21*

mysql -uroot -pmysql

Server version: 8.0.21 MySQL Community Server - GPL

*#创建test用户*

**create** **user** **test**@'%' **identified** **by** 'mysql';

*#授权*

**grant** all **privileges** **on** \*.\* **to** **test**@'%' **with** **grant** **option**;

*#刷新权限*

**flush** **privileges**;

### 启动停止mysql

docker stop mysql

docker start mysql

## 其它

1. 解决使用代理后，连接MySQL慢的问题

修改/usr/local/docker/mysql/conf/mysql.cnf文件

[mysqld]

skip-name-resolve

重启mysql

1. 解决使用代理后，ssh连接虚拟机慢的问题

是ssh服务端在连接时会自动检测DNS环境导致

修改文件：/etc/ssh/sshd\_config

*#UseDNS yes*

去掉注释，改为

UseDNS no

重启ssh服务

systemctl restart sshd

# 源码

