Spring Boot 多数据源动态配置

Spring Framework 为 SQL 数据库提供了广泛的支持。从直接使用 JdbcTemplate 进行 JDBC 访问到完全的对象关系映射（object relational mapping）技术，比如 Hibernate。Spring Data 提供了更多级别的功能，直接从接口创建的 Repository 实现，并使用了约定从方法名生成查询。

一、默认数据源HikariDataSource

1、创建项目，导入依赖

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jdbc</artifactId>
</dependency>
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <scope>runtime</scope>
</dependency>
```

2、建库脚本

```
-- 创建ds1数据库
CREATE DATABASE `ds1` CHARACTER SET utf8 COLLATE utf8_general_ci;

-- ----------------------------
-- Table structure for user
-- ----------------------------
DROP TABLE IF EXISTS `user`;
CREATE TABLE `user`  (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `name` varchar(255) CHARACTER SET utf8 COLLATE utf8_general_ci NULL DEFAULT
NULL,
  PRIMARY KEY (`id`) USING BTREE
) ENGINE = InnoDB AUTO_INCREMENT = 2 CHARACTER SET = utf8 COLLATE =
utf8_general_ci ROW_FORMAT = Dynamic;

-- ----------------------------
-- Records of user
-- ----------------------------
INSERT INTO `user` VALUES (1, 'zhangsan');

------------------------------------------------------------------------------
---------


-- 创建ds2数据库
CREATE DATABASE `ds2` CHARACTER SET utf8 COLLATE utf8_general_ci;

-- ----------------------------
-- Table structure for user
```

```
-- ----------------------------
DROP TABLE IF EXISTS `user`;
CREATE TABLE `user`  (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `name` varchar(255) CHARACTER SET utf8 COLLATE utf8_general_ci NULL DEFAULT
NULL,
  PRIMARY KEY (`id`) USING BTREE
) ENGINE = InnoDB AUTO_INCREMENT = 2 CHARACTER SET = utf8 COLLATE =
utf8_general_ci ROW_FORMAT = Dynamic;

-- ----------------------------
-- Records of user
-- ----------------------------
INSERT INTO `user` VALUES (2, 'lisi');
```

3、配置数据源

```
spring.datasource.username=root
spring.datasource.password=123456
spring.datasource.url=jdbc:mysql://127.0.0.1:40004/ds1?
serverTimezone=UTC&useUnicode=true@characterEncoding=utf-8
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
```

4、测试类代码

```
package com.yuandengta;

import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;

import javax.sql.DataSource;
import java.sql.Connection;
import java.sql.SQLException;

/**
 * 测试默认数据源
 */
@SpringBootTest
class DefaultDataSourceTest {

    @Autowired
    DataSource dataSource;

    @Test
    void contextLoads() throws SQLException {
        System.out.println(dataSource.getClass());
        Connection connection = dataSource.getConnection();
        System.out.println("默认数据源：" + connection);
        connection.close();
    }
```

```
}
```

5、执行crud操作

1、有了数据源(com.zaxxer.hikari.HikariDataSource)，然后可以拿到数据库连接(java.sql.Connection)，有了连接，就可以使用连接和原生的 JDBC 语句来操作数据库

2、即使不使用第三方第数据库操作框架，如 MyBatis等，Spring 本身也对原生的JDBC 做了轻量级的封装，即 org.springframework.jdbc.core.JdbcTemplate。

3、数据库操作的所有 CRUD 方法都在 JdbcTemplate 中。

4、Spring Boot 不仅提供了默认的数据源，同时默认已经配置好了 JdbcTemplate 放在了容器中，程序员只需自己注入即可使用

5、JdbcTemplate 的自动配置原理是依赖 org.springframework.boot.autoconfigure.jdbc 包下的 org.springframework.boot.autoconfigure.jdbc.JdbcTemplateAutoConfiguration 类

```
package com.yuandengta.controller;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RestController;
import java.util.List;
import java.util.Map;
import java.util.Random;

@RestController
public class UserController {

    @Autowired
    JdbcTemplate jdbcTemplate;

    @GetMapping("/userlist")
    public List<Map<String, Object>> userList() {
        String sql = "select * from user";
        List<Map<String, Object>> maps = jdbcTemplate.queryForList(sql);
        return maps;
    }

    @GetMapping("/addUser")
    public String addUser() {
        Random random = new Random();
        int no = random.nextInt(99999);
        String sql = "insert into user(name) values('test" + no + "')";
        jdbcTemplate.update(sql);
        return "success";
    }

    @GetMapping("/updateUser/{id}")
    public String updateUser(@PathVariable("id") Integer id) {
        String sql = "update user set name=? where id = " + id;
        String name = "list";
        jdbcTemplate.update(sql, name);
        return "update success";
    }
```

```
    @GetMapping("/deleteUser/{id}")
    public String deleteUser(@PathVariable("id") Integer id) {
        String sql = "delete from user where id = " + id;
        jdbcTemplate.update(sql);
        return "delete success";
    }
}
```

二、自定义数据源DruidDataSource

通过源码查看DataSourceAutoConfiguration.java

```
@Configuration(proxyBeanMethods = false)
@ConditionalOnClass({ DataSource.class, EmbeddedDatabaseType.class })
@EnableConfigurationProperties(DataSourceProperties.class)
@Import({ DataSourcePoolMetadataProvidersConfiguration.class,
DataSourceInitializationConfiguration.class })
public class DataSourceAutoConfiguration {

    @Configuration(proxyBeanMethods = false)
    @Conditional(EmbeddedDatabaseCondition.class)
    @ConditionalOnMissingBean({ DataSource.class, XADataSource.class })
    @Import(EmbeddedDataSourceConfiguration.class)
    protected static class EmbeddedDatabaseConfiguration {

    }

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

    }

    /**
     * {@link AnyNestedCondition} that checks that either {@code
spring.datasource.type}
     * is set or {@link PooledDataSourceAvailableCondition} applies.
     */
    static class PooledDataSourceCondition extends AnyNestedCondition {

        PooledDataSourceCondition() {
            super(ConfigurationPhase.PARSE_CONFIGURATION);
        }

        @ConditionalOnProperty(prefix = "spring.datasource", name = "type")
        static class ExplicitType {

        }
```

```
        @Conditional(PooledDataSourceAvailableCondition.class)
        static class PooledDataSourceAvailable {

        }

    }
```

1、添加druid的maven配置

```
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>druid</artifactId>
    <version>1.1.12</version>
</dependency>
```

2、添加数据源的配置

```
spring.datasource.username=root
spring.datasource.password=123456
spring.datasource.url=jdbc:mysql://127.0.0.1:40004/ds1?
serverTimezone=UTC&useUnicode=true@characterEncoding=utf-8
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
spring.datasource.type=com.alibaba.druid.pool.DruidDataSource
```

3、测试发现数据源已经更改

4、druid是数据库连接池，可以添加druid的独有配置

```
spring:
  datasource:
    username: root
    password: 123456
    url: jdbc:mysql://192.168.85.111:3306/demo?
serverTimezone=UTC&useUnicode=true@characterEncoding=utf-8
    driver-class-name: com.mysql.jdbc.Driver
    type: com.alibaba.druid.pool.DruidDataSource
    #Spring Boot 默认是不注入这些属性值的，需要自己绑定
    #druid 数据源专有配置
    initialSize: 5
    minIdle: 5
    maxActive: 20
    maxWait: 60000
    timeBetweenEvictionRunsMillis: 60000
    minEvictableIdleTimeMillis: 300000
    validationQuery: SELECT 1 FROM DUAL
    testWhileIdle: true
    testOnBorrow: false
    testOnReturn: false
    poolPreparedStatements: true

    #配置监控统计拦截的filters，stat:监控统计、log4j：日志记录、wall：防御sql注入
    #如果允许时报错  java.lang.ClassNotFoundException: org.apache.log4j.Priority
```

```
    #则导入 log4j 依赖即可，Maven 地址:
https://mvnrepository.com/artifact/log4j/log4j
    filters: stat,wall,log4j
    maxPoolPreparedStatementPerConnectionSize: 20
    useGlobalDataSourceStat: true
    connectionProperties: druid.stat.mergeSql=true;druid.stat.slowSqlMillis=500
```

测试类，发现配置的参数没有生效

```
package com.yuandengta;

import com.alibaba.druid.pool.DruidDataSource;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;

import javax.sql.DataSource;
import java.sql.Connection;
import java.sql.SQLException;

/**
 * Druid数据源测试
 */
@SpringBootTest
class DruidDataSourceTest {

    @Autowired
    DataSource dataSource;

    @Test
    void contextLoads() throws SQLException {
        System.out.println(dataSource.getClass());
        Connection connection = dataSource.getConnection();
        System.out.println(connection);

        DruidDataSource druidDataSource = (DruidDataSource) dataSource;
        System.out.println("数据源类名：" + druidDataSource.getClass().getName());
        System.out.println(druidDataSource.getMaxActive());
        System.out.println(druidDataSource.getInitialSize());
        connection.close();
    }
}
```

需要定义druidDatasource的配置类，绑定参数

```
package com.yuandengta.config;

import com.alibaba.druid.pool.DruidDataSource;
import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

import javax.sql.DataSource;

@Configuration
```

```
public class DruidConfig {
    @ConfigurationProperties(prefix = "spring.datasource")
    @Bean
    public DataSource druidDataSource() {
        return new DruidDataSource();
    }
}
```

Druid数据源还具有监控的功能，并提供了一个web界面方便用户进行查看。

加入log4j的日志依赖

```
<dependency>
    <groupId>log4j</groupId>
    <artifactId>log4j</artifactId>
    <version>1.2.17</version>
</dependency>
```

向DruidConfig中添加代码，配置druid监控管理台的servlet

```
package com.yuandengta.config;

import com.alibaba.druid.pool.DruidDataSource;
import com.alibaba.druid.support.http.StatViewServlet;
import com.alibaba.druid.support.http.WebStatFilter;
import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.boot.web.servlet.FilterRegistrationBean;
import org.springframework.boot.web.servlet.ServletRegistrationBean;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import javax.servlet.Servlet;
import javax.sql.DataSource;
import java.util.Arrays;
import java.util.HashMap;
import java.util.Map;

@Configuration
public class DruidConfig {
    @ConfigurationProperties(prefix = "spring.datasource")
    @Bean
    public DataSource druidDataSource() {
        return new DruidDataSource();
    }

    @Bean
    public ServletRegistrationBean druidServletRegistrationBean() {
        ServletRegistrationBean<Servlet> servletRegistrationBean = new
ServletRegistrationBean<>(new StatViewServlet(), "/druid/*");
        Map<String, String> initParams = new HashMap<>();
        initParams.put("loginUsername", "admin");
        initParams.put("loginPassword", "123456");
        //后台允许谁可以访问
        //initParams.put("allow", "localhost")：表示只有本机可以访问
        //initParams.put("allow", "")：为空或者为null时，表示允许所有访问
        initParams.put("allow", "");
        //deny：Druid 后台拒绝谁访问
```

```
        //initParams.put("roc", "127.0.0.1");表示禁止此ip访问

        servletRegistrationBean.setInitParameters(initParams);
        return servletRegistrationBean;
    }

    //配置 Druid 监控 之  web 监控的 filter
    //WebStatFilter：用于配置Web和Druid数据源之间的管理关联监控统计
    @Bean
    public FilterRegistrationBean webStatFilter() {
        FilterRegistrationBean bean = new FilterRegistrationBean();
        bean.setFilter(new WebStatFilter());

        //exclusions：设置哪些请求进行过滤排除掉，从而不进行统计
        Map<String, String> initParams = new HashMap<>();
        initParams.put("exclusions", "*.js,*.css,/druid/*");
        bean.setInitParameters(initParams);

        //"/*" 表示过滤所有请求
        bean.setUrlPatterns(Arrays.asList("/*"));
        return bean;
    }
}
```

三、springboot配置多数据源并动态切换

DataSource是和线程绑定的，动态数据源的配置主要是通过继承AbstractRoutingDataSource类实现的，实现在AbstractRoutingDataSource类中的 protected Object determineCurrentLookupKey()方法来获取数据源，所以我们需要先创建一个多线程线程数据隔离的类来存放DataSource，然后在determineCurrentLookupKey()方法中通过这个类获取当前线程的DataSource，在AbstractRoutingDataSource类中，DataSource是通过Key-value的方式保存的，我们可以通过ThreadLocal来保存Key，从而实现数据源的动态切换。

1、修改配置文件类

application-mult.properties

```
#数据源1
spring.datasource.ds1.jdbc-url=jdbc:mysql://localhost:40004/ds1?
serverTimezone=UTC&useUnicode=true@characterEncoding=utf-8
spring.datasource.ds1.username=root
spring.datasource.ds1.password=123456
spring.datasource.ds1.driver-class-name=com.mysql.jdbc.Driver
#数据源2
spring.datasource.ds2.jdbc-url=jdbc:mysql://localhost:40004/ds2?
serverTimezone=UTC&useUnicode=true@characterEncoding=utf-8
spring.datasource.ds2.username=root
spring.datasource.ds2.password=123456
spring.datasource.ds2.driver-class-name=com.mysql.jdbc.Driver
```

2、创建数据源枚举类

```
package com.yuandengta.mult;

public enum DataSourceType {
    DS1,
    DS2
}
```

3、数据源切换处理

创建一个数据源切换处理类，有对数据源变量的获取、设置和情况的方法，其中threadlocal用于保存某个线程共享变量。

```
package com.yuandengta.mult;

public class DynamicDataSourceContextHolder {

    /**
     * 使用ThreadLocal维护变量，ThreadLocal为每个使用该变量的线程提供独立的变量副本，
     * 所以每一个线程都可以独立地改变自己的副本，而不会影响其它线程所对应的副本。
     */
    private static final ThreadLocal<String> CONTEXT_HOLDER = new ThreadLocal<>
();

    /**
     * 设置数据源变量
     *
     * @param dataSourceType
     */
    public static void setDataSourceType(String dataSourceType) {
        System.out.printf("切换到{%s}数据源\n", dataSourceType);
        CONTEXT_HOLDER.set(dataSourceType);
    }

    /**
     * 获取数据源变量
     *
     * @return
     */
    public static String getDataSourceType() {
        return CONTEXT_HOLDER.get();
    }

    /**
     * 清空数据源变量
     */
    public static void clearDataSourceType() {
        CONTEXT_HOLDER.remove();
    }
}
```

4、继承AbstractRoutingDataSource

动态切换数据源主要依靠AbstractRoutingDataSource。创建一个AbstractRoutingDataSource的子类，重写determineCurrentLookupKey方法，用于决定使用哪一个数据源。这里主要用到AbstractRoutingDataSource的两个属性defaultTargetDataSource和targetDataSources。defaultTargetDataSource默认目标数据源，targetDataSources（map类型）存放用来切换的数据源。

```
package com.yuandengta.mult;

import org.springframework.jdbc.datasource.lookup.AbstractRoutingDataSource;
import javax.sql.DataSource;
import java.util.Map;

public class DynamicDataSource extends AbstractRoutingDataSource {

    public DynamicDataSource(DataSource defaultTargetDataSource, Map<Object,
Object> targetDataSources) {
        super.setDefaultTargetDataSource(defaultTargetDataSource);
        super.setTargetDataSources(targetDataSources);
        // afterPropertiesSet()方法调用时用来将targetDataSources的属性写入
resolvedDataSources中的
        super.afterPropertiesSet();
    }

    /**
     * 根据Key获取数据源的信息
     *
     * @return
     */
    @Override
    protected Object determineCurrentLookupKey() {
        return DynamicDataSourceContextHolder.getDataSourceType();
    }
}
```

5、注入数据源

```
package com.yuandengta.mult;

import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.boot.jdbc.DataSourceBuilder;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Primary;

import javax.sql.DataSource;
import java.util.HashMap;
import java.util.Map;

@Configuration
public class DataSourceConfig {

    @Bean
    @ConfigurationProperties("spring.datasource.ds1")
    public DataSource ds1() {
        return DataSourceBuilder.create().build();
    }
```

```
    @Bean
    @ConfigurationProperties("spring.datasource.ds2")
    public DataSource ds2() {
        return DataSourceBuilder.create().build();
    }

    @Bean(name = "dynamicDataSource")
    @Primary
    public DynamicDataSource dataSource(DataSource ds1, DataSource ds2) {
        Map<Object, Object> targetDataSources = new HashMap<>();
        targetDataSources.put(DataSourceType.DS1.name(), ds1);
        targetDataSources.put(DataSourceType.DS2.name(), ds2);
        return new DynamicDataSource(ds1, targetDataSources);
    }
}
```

6、自定义多数据源切换注解

设置拦截数据源的注解，可以设置在具体的类上，或者在具体的方法上

```
package com.yuandengta.mult;

import java.lang.annotation.*;

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface DataSource {
    /**
     * 切换数据源名称
     */
    DataSourceType value() default DataSourceType.DS1;
}
```

7、AOP拦截类的实现

通过拦截上面的注解，在其执行之前处理设置当前执行SQL的数据源的信息，CONTEXT_HOLDER.set(dataSourceType)这里的数据源信息从我们设置的注解上面获取信息，如果没有设置就是用默认的数据源的信息。

加入aspectj的依赖

```
<dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjweaver</artifactId>
</dependency>
```

```
package com.yuandengta.mult;

import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
```

```
import org.aspectj.lang.annotation.Pointcut;
import org.aspectj.lang.reflect.MethodSignature;
import org.springframework.core.annotation.Order;
import org.springframework.stereotype.Component;
import java.lang.reflect.Method;

@Aspect
@Order(1)
@Component
public class DataSourceAspect {

    @Pointcut("@annotation(com.yuandengta.mult.DataSource)")
    public void dsPointCut() {

    }

    @Around("dsPointCut()")
    public Object around(ProceedingJoinPoint point) throws Throwable {
        MethodSignature signature = (MethodSignature) point.getSignature();
        Method method = signature.getMethod();
        DataSource dataSource = method.getAnnotation(DataSource.class);
        if (dataSource != null) {

 DynamicDataSourceContextHolder.setDataSourceType(dataSource.value().name());
        }
        try {
            return point.proceed();
        } finally {
            // 销毁数据源 在执行方法之后
            DynamicDataSourceContextHolder.clearDataSourceType();
        }
    }
}
```

8、使用切换数据源注解

```
package com.yuandengta.controller;

import com.yuandengta.mult.DataSource;
import com.yuandengta.mult.DataSourceType;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

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

@RestController
public class MultDsController {

    @Autowired
    JdbcTemplate jdbcTemplate;

    @GetMapping("/ds1")
    @DataSource(value = DataSourceType.DS1)
```

```
    public List<Map<String, Object>> ds1() {
        List<Map<String, Object>> maps = jdbcTemplate.queryForList("select *
from user");
        return maps;
    }

    @GetMapping("/ds2")
    @DataSource(value = DataSourceType.DS2)
    public List<Map<String, Object>> ds2() {
        List<Map<String, Object>> maps = jdbcTemplate.queryForList("select *
from user");
        return maps;
    }
}
```

9、在启动项目的过程中会发生循环依赖的问题，直接修改启动类即可

```
package com.yuandengta;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration;

@SpringBootApplication(exclude = DataSourceAutoConfiguration.class)
public class MultDsApplication {

    public static void main(String[] args) {
        SpringApplication.run(MultDsApplication.class, args);
    }
}
```

四、源码剖析AbstractRoutingDataSource

1. AbstractRoutingDataSource类属性说明

```
@Nullable
    private Map<Object, Object> targetDataSources;
    @Nullable
    private Object defaultTargetDataSource;
    private boolean lenientFallback = true;
    private DataSourceLookup dataSourceLookup = new JndiDataSourceLookup();
    @Nullable
    private Map<Object, DataSource> resolvedDataSources;
    @Nullable
    private DataSource resolvedDefaultDataSource;
```

1. **targetDataSources**存储数据源列表
2. **defaultTargetDataSource**是默认数据源
3. **lenientFallback** 表示根据DynamicDataSource返回的lookupKey找不到数据源时是否使用默认的数据源
4. **dataSourceLookup** 数据源查找，使用jndi(java命名和目录接口)的实现

5. **resolvedDataSources** 将targetDataSources的value从Object转换为DataSource类型，存入resolvedDataSources
6. **resolvedDefaultDataSource** 将defaultTargetDataSource转化为DataSource类型，存入resolvedDefaultDataSource

2. afterPropertiesSet方法分析

AbstractRoutingDataSource类的核心方法有2个，分别是**afterPropertiesSet()和determineTargetDataSource()。**

我们来回顾一下在DruidDataSourceConfig中dataSource()方法创建的动态数据源

```
    @Primary //当相同类型的实现类存在时，选择该注解标记的类
    @Bean(name = "dynamicDataSource")
    public DynamicDataSource dataSource(DataSource ds1, DataSource ds2) {
        Map<Object, Object> targetDataSources = new HashMap<>();
        targetDataSources.put(DataSourceType.DS1.name(), ds1);
        targetDataSources.put(DataSourceType.DS2.name(), ds2);
        return new DynamicDataSource(ds1, targetDataSources);
    }
```

该方法返回的是DynamicDataSource类型的数据源，不要忘记DynamicDataSource是继承自AbstractRoutingDataSource的。首先实例化了一个DynamicDataSource，设置了默认的数据源，放入defaultTargetDataSource中，然后将数据源列表ds1、ds2以Map的形式放入了targetDataSources中。

afterPropertiesSet()、**determineTargetDataSource()**、这里创建数据源的**dataSource**()方法的调用顺序是：spring启动–>dataSource()–>afterPropertiesSet()；当我们设置数据源DataSourceContextHolder.setDataSource()的时候会调用determineTargetDataSource()

接下来看看**afterPropertiesSet**()做了些什么：

```
@Override
public void afterPropertiesSet() {
    if (this.targetDataSources == null) {
        throw new IllegalArgumentException("Property 'targetDataSources' is
required");
    }
    this.resolvedDataSources = new HashMap<>(this.targetDataSources.size());
    this.targetDataSources.forEach((key, value) -> {
        Object lookupKey = resolveSpecifiedLookupKey(key);
        DataSource dataSource = resolveSpecifiedDataSource(value);
        this.resolvedDataSources.put(lookupKey, dataSource);
    });
    if (this.defaultTargetDataSource != null) {
        this.resolvedDefaultDataSource =
resolveSpecifiedDataSource(this.defaultTargetDataSource);
    }
}
```

targetDataSources为空抛异常，遍历targetDataSources，这里使用了forEach方法，forEach方法内部其实是个foreach的循环，resolveSpecifiedLookupKey(key)其实直接返回了key，resolveSpecifiedDataSource(value)是将targetDataSources的value(Object类型)转换为DataSource类型，放入resolvedDataSources这个Map中。 可以看看resolveSpecifiedDataSource(value)的实现：

```
protected DataSource resolveSpecifiedDataSource(Object dataSource) throws
IllegalArgumentException {
        if (dataSource instanceof DataSource) {
            return (DataSource) dataSource;
        }
        else if (dataSource instanceof String) {
            return this.dataSourceLookup.getDataSource((String) dataSource);
        }
        else {
            throw new IllegalArgumentException(
                    "Illegal data source value - only [javax.sql.DataSource] and
String supported: " + dataSource);
        }
    }
```

如果dataSource是DataSource类型直接返回，如果是String类型，用JndiDataSourceLookup去查找dataSource，否则抛个不合法参数异常。总之就是将Object类型转换为DataSource类型。 至此afterPropertiesSet()执行完毕，resolvedDefaultDataSource存储了默认数据源，resolvedDataSources存放了可用数据源列表。

3. determineTargetDataSource方法分析

当我们设置数据源DataSourceContextHolder.setDataSource()的时候会调用**determineTargetDataSource()**，看看determineTargetDataSource()又做了什么：

```
protected DataSource determineTargetDataSource() {
        Assert.notNull(this.resolvedDataSources, "DataSource router not
initialized");
        Object lookupKey = determineCurrentLookupKey();
        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 + "]");
        }
        return dataSource;
    }
```

可以看到调用了我们的DynamicDataSource重写的determineCurrentLookupKey()方法，返回的正是我们在DataSourceContextHolder设置的dataSource的key，然后根据这个key在resolvedDataSources里查找DataSource ，如果找到直接返回，没找到并且lenientFallback为true(默认为true)或lookupKey为null，就会使用默认的数据源resolvedDefaultDataSource，lenientFallback前面已经说过表示DynamicDataSource返回的lookupKey找不到数据源时是否使用默认的数据源。如果上述条件都不满足，抛异常。 这个determineTargetDataSource()调用完了，数据源也就设置成我们在DataSourceContextHolder中设置的dataSource了。