**文档摘要：**[**https://www.cnblogs.com/fengmao/p/7538854.html**](https://www.cnblogs.com/fengmao/p/7538854.html)

**https://blog.csdn.net/husong\_/article/details/80103497**

**多数据源配置**

创建一个Spring配置类，定义两个DataSource用来读取application.properties中的不同配置。如下例子中，主数据源配置为spring.datasource.primary开头的配置，第二数据源配置为spring.datasource.secondary开头的配置。

@Configuration

public class DataSourceConfig {

@Bean(name = "primaryDataSource")

@Qualifier("primaryDataSource")

@ConfigurationProperties(prefix="spring.datasource.primary")

public DataSource primaryDataSource() {

return DataSourceBuilder.create().build();

}

@Bean(name = "secondaryDataSource")

@Qualifier("secondaryDataSource")

@Primary

@ConfigurationProperties(prefix="spring.datasource.secondary")

public DataSource secondaryDataSource() {

return DataSourceBuilder.create().build();

}

}

对应的application.properties配置如下：

spring.datasource.primary.url=jdbc:mysql://localhost:3306/test1

spring.datasource.primary.username=root

spring.datasource.primary.password=root

spring.datasource.primary.driver-class-name=com.mysql.jdbc.Driver

spring.datasource.secondary.url=jdbc:mysql://localhost:3306/test2

spring.datasource.secondary.username=root

spring.datasource.secondary.password=root

spring.datasource.secondary.driver-class-name=com.mysql.jdbc.Driver

**JdbcTemplate支持**

对JdbcTemplate的支持比较简单，只需要为其注入对应的datasource即可，如下例子，在创建JdbcTemplate的时候分别注入名为primaryDataSource和secondaryDataSource的数据源来区分不同的JdbcTemplate。

@Bean(name = "primaryJdbcTemplate")

public JdbcTemplate primaryJdbcTemplate(

@Qualifier("primaryDataSource") DataSource dataSource) {

return new JdbcTemplate(dataSource);

}

@Bean(name = "secondaryJdbcTemplate")

public JdbcTemplate secondaryJdbcTemplate(

@Qualifier("secondaryDataSource") DataSource dataSource) {

return new JdbcTemplate(dataSource);

}

接下来通过测试用例来演示如何使用这两个针对不同数据源的JdbcTemplate。

@RunWith(SpringJUnit4ClassRunner.class)

@SpringApplicationConfiguration(Application.class)

public class ApplicationTests {

@Autowired

@Qualifier("primaryJdbcTemplate")

protected JdbcTemplate jdbcTemplate1;

@Autowired

@Qualifier("secondaryJdbcTemplate")

protected JdbcTemplate jdbcTemplate2;

@Before

public void setUp() {

jdbcTemplate1.update("DELETE FROM USER ");

jdbcTemplate2.update("DELETE FROM USER ");

}

@Test

public void test() throws Exception {

// 往第一个数据源中插入两条数据

jdbcTemplate1.update("insert into user(id,name,age) values(?, ?, ?)", 1, "aaa", 20);

jdbcTemplate1.update("insert into user(id,name,age) values(?, ?, ?)", 2, "bbb", 30);

// 往第二个数据源中插入一条数据，若插入的是第一个数据源，则会主键冲突报错

jdbcTemplate2.update("insert into user(id,name,age) values(?, ?, ?)", 1, "aaa", 20);

// 查一下第一个数据源中是否有两条数据，验证插入是否成功

Assert.assertEquals("2", jdbcTemplate1.queryForObject("select count(1) from user", String.class));

// 查一下第一个数据源中是否有两条数据，验证插入是否成功

Assert.assertEquals("1", jdbcTemplate2.queryForObject("select count(1) from user", String.class));

}

}

[完整示例:Chapter3-2-3](http://git.oschina.net/didispace/SpringBoot-Learning)

**Spring-data-jpa支持**

对于数据源的配置可以沿用上例中DataSourceConfig的实现。

新增对第一数据源的JPA配置，注意两处注释的地方，用于指定数据源对应的Entity实体和Repository定义位置，用@Primary区分主数据源。

@Configuration

@EnableTransactionManagement

@EnableJpaRepositories(

entityManagerFactoryRef="entityManagerFactoryPrimary",

transactionManagerRef="transactionManagerPrimary",

basePackages= { "com.didispace.domain.p" }) //设置Repository所在位置

public class PrimaryConfig {

@Autowired @Qualifier("primaryDataSource")

private DataSource primaryDataSource;

@Primary

@Bean(name = "entityManagerPrimary")

public EntityManager entityManager(EntityManagerFactoryBuilder builder) {

return entityManagerFactoryPrimary(builder).getObject().createEntityManager();

}

@Primary

@Bean(name = "entityManagerFactoryPrimary")

public LocalContainerEntityManagerFactoryBean entityManagerFactoryPrimary (EntityManagerFactoryBuilder builder) {

return builder

.dataSource(primaryDataSource)

.properties(getVendorProperties(primaryDataSource))

.packages("com.didispace.domain.p") //设置实体类所在位置

.persistenceUnit("primaryPersistenceUnit")

.build();

}

@Autowired

private JpaProperties jpaProperties;

private Map<String, String> getVendorProperties(DataSource dataSource) {

return jpaProperties.getHibernateProperties(dataSource);

}

@Primary

@Bean(name = "transactionManagerPrimary")

public PlatformTransactionManager transactionManagerPrimary(EntityManagerFactoryBuilder builder) {

return new JpaTransactionManager(entityManagerFactoryPrimary(builder).getObject());

}

}

新增对第二数据源的JPA配置，内容与第一数据源类似，具体如下：

@Configuration

@EnableTransactionManagement

@EnableJpaRepositories(

entityManagerFactoryRef="entityManagerFactorySecondary",

transactionManagerRef="transactionManagerSecondary",

basePackages= { "com.didispace.domain.s" }) //设置Repository所在位置

public class SecondaryConfig {

@Autowired @Qualifier("secondaryDataSource")

private DataSource secondaryDataSource;

@Bean(name = "entityManagerSecondary")

public EntityManager entityManager(EntityManagerFactoryBuilder builder) {

return entityManagerFactorySecondary(builder).getObject().createEntityManager();

}

@Bean(name = "entityManagerFactorySecondary")

public LocalContainerEntityManagerFactoryBean entityManagerFactorySecondary (EntityManagerFactoryBuilder builder) {

return builder

.dataSource(secondaryDataSource)

.properties(getVendorProperties(secondaryDataSource))

.packages("com.didispace.domain.s") //设置实体类所在位置

.persistenceUnit("secondaryPersistenceUnit")

.build();

}

@Autowired

private JpaProperties jpaProperties;

private Map<String, String> getVendorProperties(DataSource dataSource) {

return jpaProperties.getHibernateProperties(dataSource);

}

@Bean(name = "transactionManagerSecondary")

PlatformTransactionManager transactionManagerSecondary(EntityManagerFactoryBuilder builder) {

return new JpaTransactionManager(entityManagerFactorySecondary(builder).getObject());

}

}

完成了以上配置之后，主数据源的实体和数据访问对象位于：com.didispace.domain.p，次数据源的实体和数据访问接口位于：com.didispace.domain.s。

分别在这两个package下创建各自的实体和数据访问接口

* 主数据源下，创建User实体和对应的Repository接口

@Entity

public class User {

@Id

@GeneratedValue

private Long id;

@Column(nullable = false)

private String name;

@Column(nullable = false)

private Integer age;

public User(){}

public User(String name, Integer age) {

this.name = name;

this.age = age;

}

// 省略getter、setter

}

public interface UserRepository extends JpaRepository<User, Long> {

}

* 从数据源下，创建Message实体和对应的Repository接口

@Entity

public class Message {

@Id

@GeneratedValue

private Long id;

@Column(nullable = false)

private String name;

@Column(nullable = false)

private String content;

public Message(){}

public Message(String name, String content) {

this.name = name;

this.content = content;

}

// 省略getter、setter

}

public interface MessageRepository extends JpaRepository<Message, Long> {

}

接下来通过测试用例来验证使用这两个针对不同数据源的配置进行数据操作。

@RunWith(SpringJUnit4ClassRunner.class)

@SpringApplicationConfiguration(Application.class)

public class ApplicationTests {

@Autowired

private UserRepository userRepository;

@Autowired

private MessageRepository messageRepository;

@Test

public void test() throws Exception {

userRepository.save(new User("aaa", 10));

userRepository.save(new User("bbb", 20));

userRepository.save(new User("ccc", 30));

userRepository.save(new User("ddd", 40));

userRepository.save(new User("eee", 50));

Assert.assertEquals(5, userRepository.findAll().size());

messageRepository.save(new Message("o1", "aaaaaaaaaa"));

messageRepository.save(new Message("o2", "bbbbbbbbbb"));

messageRepository.save(new Message("o3", "cccccccccc"));

Assert.assertEquals(3, messageRepository.findAll().size());

}

}