# Springboot Reactive MongoDB Quick Start

## 1. 简介

在本篇教程中，会介绍如何通过reactive编程的方式配置并操作mongo DB，会具体介绍在springboot中ReactiveMongoRepository，ReactiveMongoTemplate的基本使用方法。
虽然本文会使用reactive方式编程，但并不会详细介绍reactive，关于reactive programming可以参见之前的教程。
本文基于springboot 2.1.4.RELEASE，spring 5.1.6.RELEASE,请确保所有相关依赖的版本符合要求，否则会出现找不到类的情况。

## 2. Maven 依赖

第一步，先引入spring-boot-dependencies, 可以避免手动引入依赖的jar包以及版本。

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-dependencies</artifactId>
            <version>2.1.4.RELEASE</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

第二步，引入jar包，为了执行测试，我们同时引入了embedded MongoDB。

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-mongodb-reactive</artifactId>
</dependency>

<dependency>
    <groupId>de.flapdoodle.embed</groupId>
    <artifactId>de.flapdoodle.embed.mongo</artifactId>
    <scope>test</scope>
</dependency>
```

## 3. Configuration 文件

为了可以激活reactive的支持，我们需要创建Mongo DB的configuration文件，我们将该文件放到项目的根目录下。

```
@EnableMongoAuditing
@EnableConfigurationProperties(MongoProperties.class)
@SpringBootApplication(exclude =
{MongoReactiveAutoConfiguration.class,
MongoReactiveDataAutoConfiguration.class})
public class MongoReactiveConfiguration extends
AbstractReactiveMongoConfiguration {
 private final MongoProperties mongoProperties;

 public MongoReactiveConfiguration(MongoProperties mongoProperties) {
 this.mongoProperties = mongoProperties;
 }

 @Override
 @Bean
 public MongoClient reactiveMongoClient() {
 return MongoClients.create(mongoProperties.getUri());
 }

 @Override
 protected String getDatabaseName() {
 return mongoProperties.getDatabase();
 }
}
```

@EnableMongoAuditing 注解是为了可以在Document对象中使用Auditing注解，比如@LastModifiedDate  @CreatedDate 注解。

@EnableConfigurationProperties(MongoProperties.class)可以通过约定自动解析application.properties中关于mongo的配置项。

@SpringBootApplication(exclude = {MongoReactiveAutoConfiguration.class, MongoReactiveDataAutoConfiguration.class})可以自动发现并注入我们定义的bean/Service/Repository等，因为Mongo的自动配置会自动连接localhost:27071，而我们需要手动配置mongo的uri，所以需要禁止mongo的自动配置，通过制定exclude = {MongoReactiveAutoConfiguration.class, MongoReactiveDataAutoConfiguration.class}来实现禁止自动配置。

AbstractReactiveMongoConfiguration是mongo的基础配置，我们只需要继承该配置并实现自己的特定需求就完成了对mongo的全部配置。

## 4. 创建 Document

定义数据类并加上@Document注解。

```
@Data
@Document
public class Account {
 @Id
 private String id;
 private String owner;
 private Double value;

 public Account(String id, String owner, double value) {
 this.id = id;
 this.owner = owner;
 this.value = value;
 }

 @CreatedDate
 private DateTime createDate;
 @LastModifiedDate
 private DateTime lastModifiedDate;
}
```

@Id注解标明类型的主键
@CreateDate自动生成数据创建的时间
@LastModifiedDate自动记录数据修改的时间

## 5. 使用ReactiveMongoRepository

```
@Repository
public interface AccountReactiveRepository extends
ReactiveMongoRepository<Account, String> {
 Flux<Account> findAllByValue(double value);

 Mono<Account> findFirstByOwner(Mono<String> owner);
}
```

ReactiveMongoRepository是支持reactive方式的针对mongo封装一系列操作的接口，我们要实现mongo的操作只需要扩展该接口定义自己的操作接口。
springboot会根据接口定义的方法动态生成实现类，我们不需要手动实现。

## 6. Mongo Reactive 事务

Mongo server 4.0之后支持事务。

```
@Autowired
private ReactiveMongoOperations template;

Flux<DeleteResult> result = template.inTransaction()
 .execute(action -> action.remove(query(where("owner").is("Bill")),
Account.class));
```

## 7. 集成测试

第一步，先在application.properties中添加mongo的配置：

```
spring.data.mongodb.port=27017
spring.data.mongodb.database=test
spring.data.mongodb.uri=mongodb://localhost
```

然后编写测试代码：

```
@RunWith(SpringRunner.class)
@SpringBootTest(classes = MongoReactiveConfiguration.class)
public class AccountReactiveRepositoryTest {
 @Autowired
 private AccountReactiveRepository repository;
 @Autowired
 private ReactiveMongoOperations template;

 @Before
 public void setup() {
 repository.deleteAll().block();
 }

 @Test
 public void givenValue_whenFindAllByValue_thenFindAccount() {
 repository.save(new Account(null, "Bill", 12.3)).block();
 Flux<Account> accountFlux = repository.findAllByValue(12.3);

 StepVerifier
 .create(accountFlux)
 .assertNext(account -> {
 assertEquals("Bill", account.getOwner());
 assertEquals(Double.valueOf(12.3), account.getValue());
 assertNotNull(account.getId());
 })
 .expectComplete()
 .verify();
 }

 @Test
 public void givenOwner_whenFindFirstByOwner_thenFindAccount() {
 repository.save(new Account(null, "Bill", 12.3)).block();
 Mono<Account> accountMono = repository
 .findFirstByOwner(Mono.just("Bill"));

 StepVerifier
 .create(accountMono)
 .assertNext(account -> {
 assertEquals("Bill", account.getOwner());
 assertEquals(Double.valueOf(12.3), account.getValue());
 assertNotNull(account.getId());
 })
 .expectComplete()
 .verify();
 }

 @Test
 public void givenAccount_whenSave_thenSaveAccount() {
 Mono<Account> accountMono = repository.save(new Account(null, "Bill",
12.3));

 StepVerifier
 .create(accountMono)
 .assertNext(account -> assertNotNull(account.getId()))
 .expectComplete()
 .verify();
 }

 @Test
 public void reactive_transaction() {
 repository.save(new Account(null, "Bill", 12.3)).block();
 Flux<DeleteResult> result = template.inTransaction()
 .execute(action -> action.remove(query(where("owner").is("Bill")),
Account.class));
 StepVerifier
 .create(result)
 .assertNext(deleteResult ->
assertEquals(deleteResult.getDeletedCount(), 1))
 .expectComplete()
 .verify();
 }
}
```