```
</plugin>
        </plugins>
    </build>

</project>
```

现在让我们看看我们的`application.yaml`管理我们的财产。

```
server:
  port: 9090
spring:
  data:
    mongodb:
      database: account_deposits
      host: localhost
      port: 27017
logging:
  level:
    org:
      springframework:
        data:
          mongodb:
            core:
              MongoTemplate: DEBUG
redis:
  pubsub:
    topic: deposit_event
```

现在是我们创建实体来将数据持久化到数据库的时候了。

```
package org.vaslabs.deposit.entity;

import lombok.Data;
import lombok.Getter;
import lombok.Setter;
```

```
import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.mapping.Document;

@Data
@Document(collection = ”accounts”)
@Getter
@Setter
public class Deposit {
    @Id
    private String id;
    private String accountNumber;
    private String firstName;
    private String lastName;
    private double amount;
}
```

现在是我们创建`writemodel`的时候了，它将作为我们的命令对象。

```
package org.vaslabs.deposit.writemodel;

import lombok.Data;
import lombok.Getter;
import lombok.Setter;

@Data
@Getter
@Setter
public class DepositCommand {
    private String accountNumber;
    private String firstName;
    private String lastName;
    private double amount;
}
```

让我们从事件处理的角度来看配置，其中服务将能够将事件传播到redis pub/sub。

```
package org.vaslabs.deposit.config;
```

```
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import
org.springframework.data.redis.connection.RedisConnectionFactory;
import org.springframework.data.redis.core.RedisTemplate;
import org.springframework.data.redis.listener.ChannelTopic;
import
org.springframework.data.redis.serializer.GenericJackson2JsonRedisSeri
alizer;

@Configuration
public class EventHandlerConfig {

    @Value("${redis.pubsub.topic}")
    private String topic;

    @Bean
    public RedisTemplate<String, Object>
redisTemplate(RedisConnectionFactory redisConnectionFactory){
        RedisTemplate<String, Object> redisTemplate = new
RedisTemplate<>();
        redisTemplate.setConnectionFactory(redisConnectionFactory);
        redisTemplate.setValueSerializer(new
GenericJackson2JsonRedisSerializer());
        return redisTemplate;
    }

    @Bean
    public ChannelTopic channelTopic(){
        return new ChannelTopic(topic);
    }
}

```

让我们创建一个存储库来处理包装为存储库的数据库逻辑。

```
package org.vaslabs.deposit.respositories;

import org.springframework.data.mongodb.repository.MongoRepository;
import org.vaslabs.deposit.entity.Deposit;

public interface DepositRepository extends MongoRepository<Deposit,
```

String> {
}

```

让我们创建一个控制器来处理来自客户端的请求。

```
package org.vaslabs.deposit.controller;

import org.springframework.http.MediaType;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RestController;
import org.vaslabs.deposit.command.handler.DepositCommandHandler;
import org.vaslabs.deposit.entity.Deposit;
import org.vaslabs.deposit.writemodel.DepositCommand;

@RestController
public class DepositController {

    private final DepositCommandHandler depositCommandHandler;

    public DepositController(DepositCommandHandler
depositCommandHandler) {
        this.depositCommandHandler = depositCommandHandler;
    }

    @PostMapping(value = ”/deposit”, consumes =
MediaType.APPLICATION_JSON_VALUE)
    public ResponseEntity<Deposit> saveDeposit(@RequestBody
DepositCommand depositCommand){
        return
ResponseEntity.ok().body(this.depositCommandHandler.handle(depositC
ommand));
    }

}

```

现在是我们编写处理程序的时候了，它将处理传入的命令。
```

```
package org.vaslabs.deposit.command.handler;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.data.redis.core.RedisTemplate;
import org.springframework.data.re .save(depositEvent);
        } catch (IOException e) {
            LOGGER.error("error while parsing message");
        }
    }
}

```

让我们看看`ViewAccountRepository`，它将与实际的数据库进行交互以进行不同的查询。

```
package org.vaslabs.view_account.respositories;

import org.springframework.data.mongodb.repository.MongoRepository;
import org.springframework.data.mongodb.repository.Query;
import org.vaslabs.view_account.entity.Deposit;

public interface ViewAccountRepository extends
MongoRepository<Deposit, String> {
    @Query("{'accountNumber': ?0}")
    Deposit findByAccountNumber(String accountNumber);
}

```

让我们看看我们的`ViewAccountController`，它将处理传入的请求。

```
package org.vaslabs.view_account.controller;

import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RestController;
import org.vaslabs.view_account.entity.Deposit;
import org.vaslabs.view_account.query.handler.DepositQueryHandler;
```

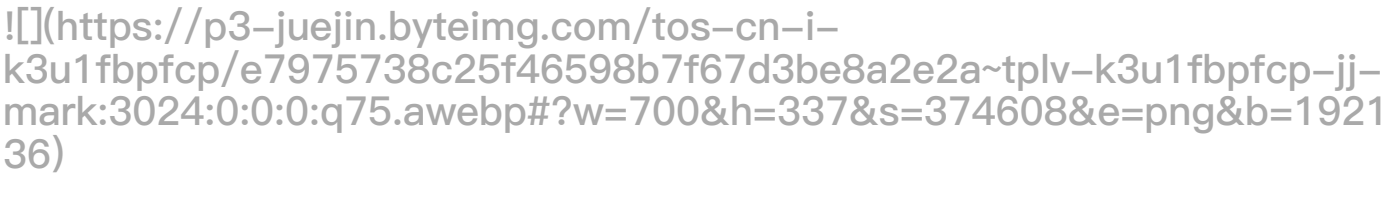
```
import java.util.List;

@RestController
public class ViewAccountController {

    private final DepositQueryHandler depositQueryHandler;

    public ViewAccountController(DepositQueryHandler
depositQueryHandler) {
        this.depositQueryHandler = depositQueryHandler;
    }

    @GetMapping(value = "/deposit/{accountNumber}")
    public ResponseEntity<Deposit>
findbyAccountNumber(@PathVariable("accountNumber") String
accountNumber){
        return
ResponseEntity.ok().body(this.depositQueryHandler.handleByAccountNu
mber(accountNumber));
    }

    @GetMapping(value = "/deposit")
    public ResponseEntity<List<Deposit>> findAll(){
        return
ResponseEntity.ok().body(this.depositQueryHandler.handleAll());
    }

}
```

现在我们来看看`ViewAccountQueryHandler`，它将处理来自用户的查询/项目。

```
package org.vaslabs.view_account.query.handler;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.stereotype.Service;
import org.vaslabs.view_account.entity.Deposit;
import org.vaslabs.view_account.respositories.ViewAccountRepository;

import java.util.List;
```

```
@Service
public class ViewAccountQueryHandler {

    private static final Logger LOGGER =
LoggerFactory.getLogger(ViewAccountQueryHandler.class);

    private final ViewAccountRepository viewAccountRepository;

    public ViewAccountQueryHandler(ViewAccountRepository
viewAccountRepository) {
        this.viewAccountRepository = viewAccountRepository;
    }

    public Deposit handleByAccountNumber(String accountNumber) {
        LOGGER.info("before querying the database, accountNumber: {}",
accountNumber);
        return
this.viewAccountRepository.findByAccountNumber(accountNumber);
    }

    public List<Deposit> handleAll(){
        return this.viewAccountRepository.findAll();
    }
}

```

如果一切顺利，您应该可以看到读取数据库中的数据，因为它是由事件异步填充的。观察下面的输出，它清楚地表明，我们已经取得了API只是通过提供一些readall查询，它也收到了事件，并成功地解析和持久化，在读存储。



qery API的输出，显示事件接收和qery处理。

使用CQRS的优点和缺点：

命令查询责任分离（CQRS）模式也不例外，它有自己的优点和缺点。它是在软件设计中处理数据操作的独特方法，提供了几个优点。首先，它通过分离读取

和写入操作来增强性能，允许每个操作针对其特定用例进行优化。在具有各种读写模式的复杂系统中，这可以显著提高性能。对于读取，数据可以以视图模型的形式进行非规范化，这些视图模型针对用户界面进行了优化，从而减少了昂贵的连接数量，否则如果将单个数据模型用于读取和写入，则可能需要昂贵的连接。对于写入，系统可以数据的事务完整性和一致性。

尽管CQRS有其优点，但也并非没有缺点。随着CQRS的引入，系统的复杂性会显著增加。开发人员必须管理和同步两个独立的模型，这可能会使代码库和架构复杂化。它需要对领域有透彻的理解，并且对于简单的CRUD应用程序来说通常是多余的，因为在这些应用程序中，开销并不能证明其好处是合理的。

CQRS的另一个挑战是数据一致性。使用单独的阅读和写数据模型，存在陈旧读取的风险–其中读取模型尚未更新以反映最新的写入。在需要高一致性的系统中尤其如此。开发人员需要仔细设计系统，以处理最终的一致性，并确保应用程序可以容忍一定程度的数据陈旧。

结论：

命令查询责任隔离（Command Query Responsibility Segregation，CQRS）模式在优化和扩展复杂的软件系统方面具有明显的优势，特别是在微服务架构中。它允许通过分离读和写操作来有针对性地提高性能。然而，它的采用带来了数据一致性方面的复杂性和挑战。当业务逻辑和数据模型的复杂性证明开销合理时，应该考虑CQRS，而不是将其作为简单应用程序的默认方法。明智地实施CQRS可以产生强大的和可维护的系统，这些系统能够很好地适应不断变化的需求。

原文链接: https://juejin.cn/post/7360961068166856745