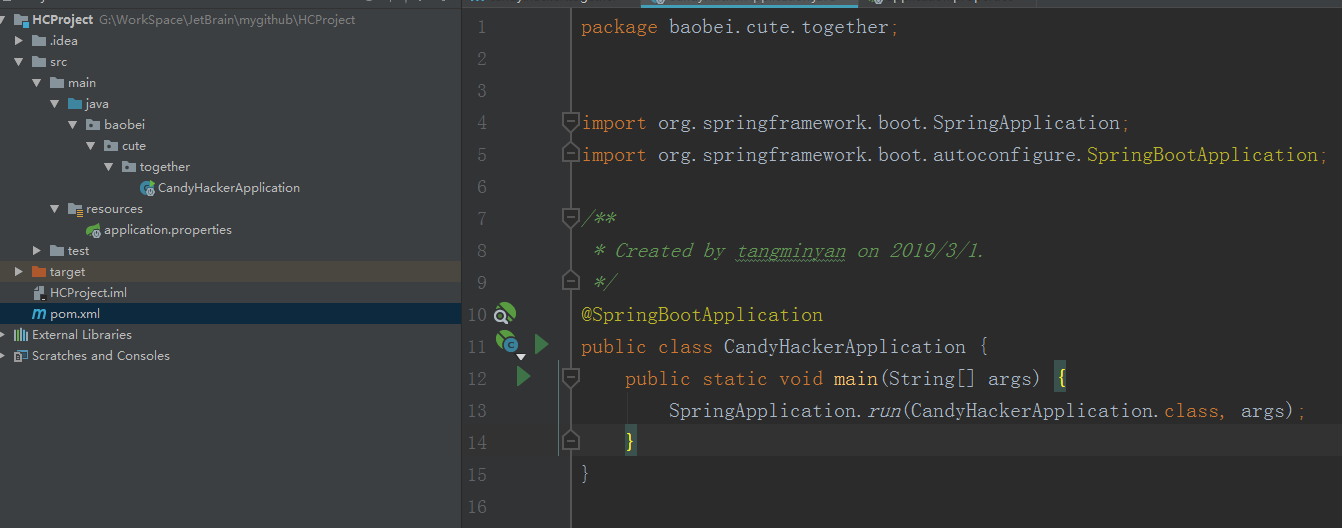
# 基础搭建

1. 新建Maven项目，导入springboot启动依赖和web依赖

<parent>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-parent</artifactId>  
 <version>1.5.2.RELEASE</version>  
</parent>  
  
<dependencies>  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-web</artifactId>  
 </dependency>

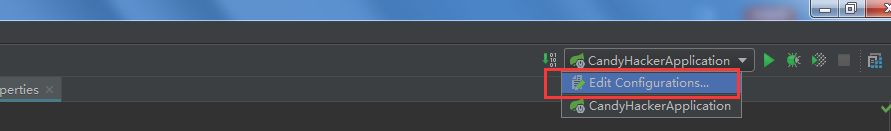
1. 新建启动函数

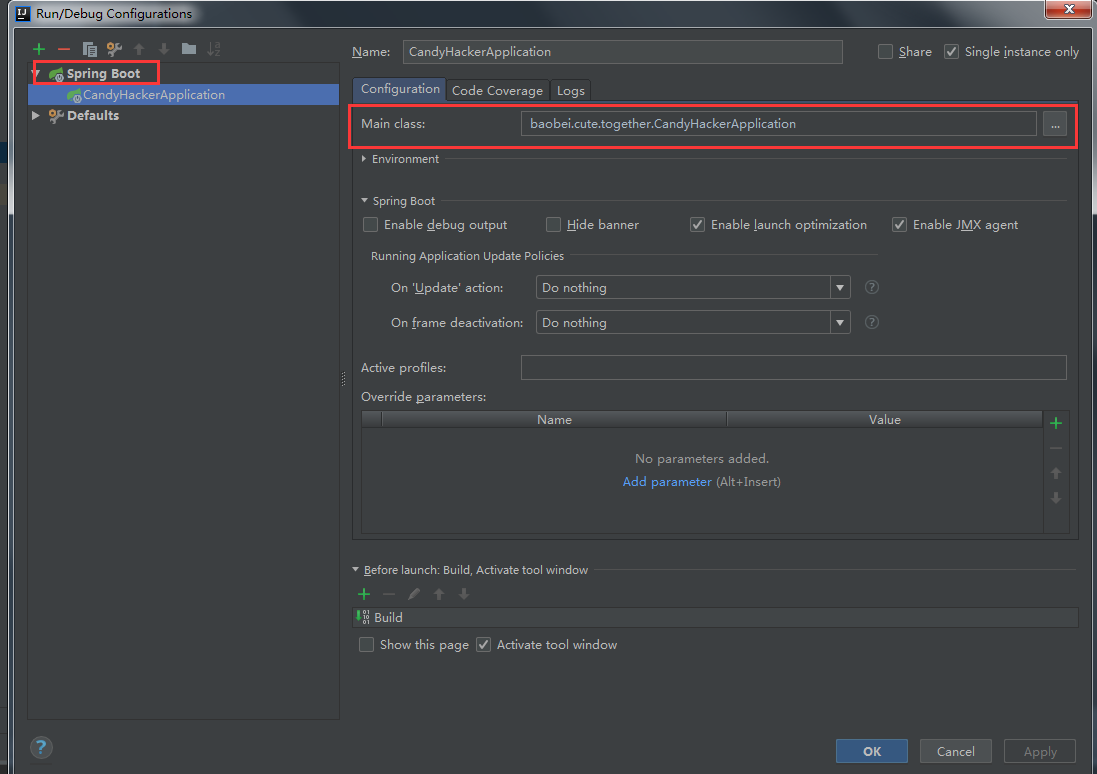


1. resources下新建application.properties配置文件，更改端口号(可不改)



1. 配置启动项，添加springboot





1. 启动

# 连数据库

1）导入依赖：

<dependency>  
 <groupId>mysql</groupId>  
 <artifactId>mysql-connector-java</artifactId>  
</dependency>

1. 设置配置文件

spring.datasource.url=jdbc:mysql://127.0.0.1:3306/better-us  
spring.datasource.username=root  
spring.datasource.password=123

注：

设置hibernate自动建表规则

spring.jpa.hibernate.ddl-auto=update

1. 测试是否成功自动建表，创建测试PO类，为此先引入部分注解的依赖，及在启动函数上添加读注解的标签
2. 导入JPA操作数据库的依赖

<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-data-jpa</artifactId>  
</dependency>

1. 导入lombok依赖

<dependency>  
 <groupId>org.projectlombok</groupId>  
 <artifactId>lombok</artifactId>  
 <version>1.18.2</version>  
 <scope>provided</scope>  
</dependency>

3>

@EntityScan("baobei.cute")

# 状态机基础（statemachinedemo包下）

1. pom文件导入依赖

<dependency>  
 <groupId>org.springframework.statemachine</groupId>  
 <artifactId>spring-statemachine-core</artifactId>  
 <version>2.0.2.RELEASE</version>  
</dependency>

1. 新建基本PO类，DAO类
2. 新建enum， 订单状态类 和 操作类

public enum OrderStatus {  
 // 待支付，待发货，待收货，订单结束  
 *WAIT\_PAYMENT*, *WAIT\_DELIVER*, *WAIT\_RECEIVE*, *FINISH*;  
}

public enum OrderStetusChangeEvent {  
 // 支付，发货，确认收货  
 *PAYED*, *DELIVER*, *RECEIVED*}

4）注入状态机的状态，事件的配置。起主要涉及到以下两个类：

1> StateMachineStateConfigurer < S, E> 配置状态集合以及初始状态，泛型参数S代表状态，E代表事件。

2> StateMachineTransitionConfigurer 配置状态流的转移，可以定义状态转换接受的事件。

@Configuration  
@EnableStateMachineFactory  
public class OrderStateMachineConfig extends StateMachineConfigurerAdapter<OrderStatus, OrderStetusChangeEvent> {  
  
 @Override  
 public void configure(StateMachineStateConfigurer<OrderStatus, OrderStetusChangeEvent> states) throws Exception {  
 states  
 .withStates()  
 .initial(OrderStatus.*WAIT\_PAYMENT*)  
 .states(EnumSet.*allOf*(OrderStatus.class));  
 super.configure(states);  
 }  
  
 @Override  
 public void configure(StateMachineTransitionConfigurer<OrderStatus, OrderStetusChangeEvent> transitions) throws Exception {  
 transitions  
 .withExternal()  
 .source(OrderStatus.*WAIT\_PAYMENT*).target(OrderStatus.*WAIT\_DELIVER*)  
 .event(OrderStetusChangeEvent.*PAYED*)  
 .and()  
 .withExternal()  
 .source(OrderStatus.*WAIT\_DELIVER*).target(OrderStatus.*WAIT\_RECEIVE*)  
 .event(OrderStetusChangeEvent.*DELIVER*)  
 .and()  
 .withExternal()  
 .source(OrderStatus.*WAIT\_PAYMENT*).target(OrderStatus.*FINISH*)  
 .event(OrderStetusChangeEvent.*RECEIVED*);  
 }  
}

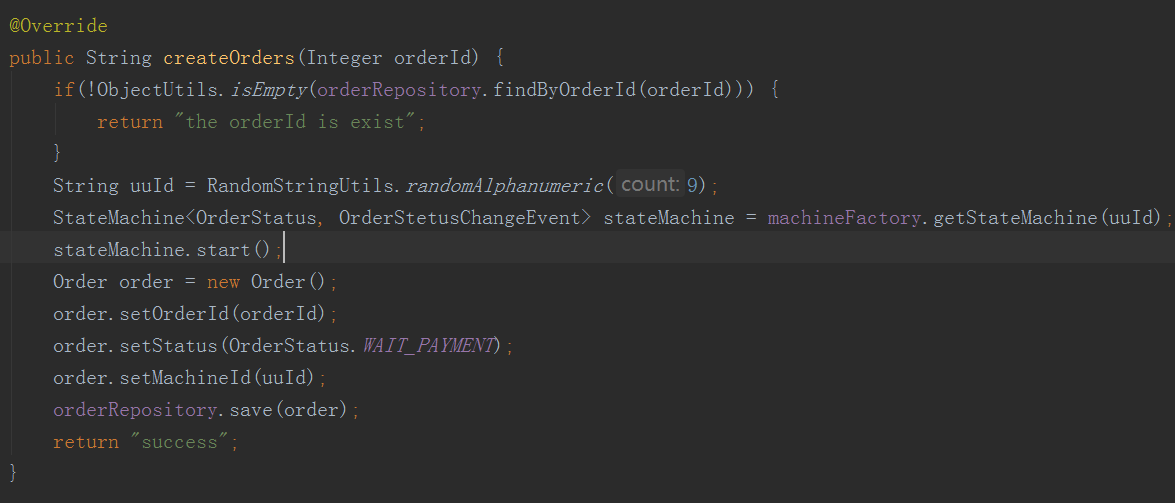
5）设置监听

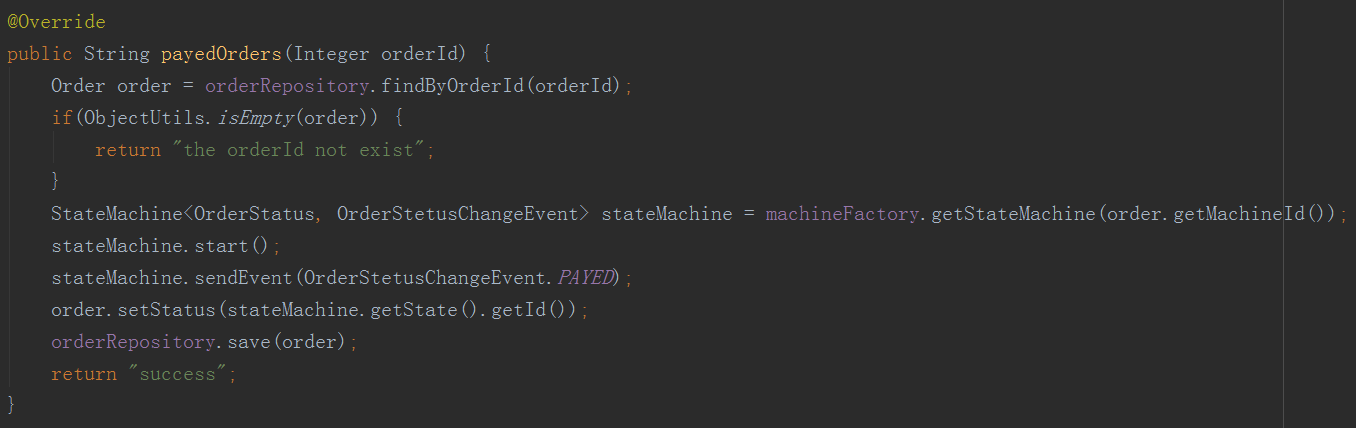
@WithStateMachine  
@Slf4j  
public class OrderEventConfig {  
 @OnTransition(target = "UNPAYED")  
 public void create() {  
 *log*.info("待支付");  
 }  
 @OnTransition(source = "UNPAYED", target = "WAITING\_FOR\_RECEIVE")  
 public void pay() {  
 *log*.info("支付完成，待收货");  
 }  
 @OnTransition(source = "WAITING\_FOR\_RECEIVE", target = "DONE")  
 public void receive() {  
 *log*.info("用户已收货，订单完成");  
 }  
}

1. 测试

注：启动函数上增加读注解的标签：

@ComponentScan(basePackages = {"baobei.cute"}) // controller service注解  
@EnableJpaRepositories("baobei.cute") //jpa





# 4、springboot rabbitMq（一）

参考：

概念：<https://www.cnblogs.com/ityouknow/p/6120544.html>

实例：<http://www.cnblogs.com/ly-radiata/articles/5566504.html>

1）安装rabbitmq，安装完成登录localhos:15672 guest/guest

2）导入Maven依赖

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

3）配置rabbitMq配置

spring.rabbitmq.host=127.0.0.1  
spring.rabbitmq.port=5672  
spring.rabbitmq.username=guest  
spring.rabbitmq.password=guest  
#实现一个监听器用于监听Broker端给我们返回的确认请求  
spring.rabbitmq.publisher-confirms=**true**#virtual host只是起到一个命名空间的作用，'/'是系统默认的，不同的命名空间之间的资源是不能访问的  
spring.rabbitmq.virtual-host=/

1. 新建配置类RabbitMqConfig，添加交换机和key，配置ConnectionFactory

public static final String *EXCHANGE* = "spring-boot-exchange";  
public static final String *ROUTINGKEY* = "spring-boot-routingKey";

@Bean  
public ConnectionFactory connectionFactory() {  
 CachingConnectionFactory connectionFactory = new CachingConnectionFactory();  
 connectionFactory.setAddresses(addresses);  
 connectionFactory.setUsername(username);  
 connectionFactory.setPassword(password);  
 connectionFactory.setPublisherConfirms(publisherConfirm);  
 connectionFactory.setVirtualHost(virtualHost);  
 return connectionFactory;  
}

5）配置RabbitTemplate

@Bean

//@scope默认是单例模式（singleton）,prototype原型模式每次获取Bean的时候会有一个新的实例  
@Scope(ConfigurableBeanFactory.*SCOPE\_PROTOTYPE*)  
public RabbitTemplate rabbitTemplate() {  
 RabbitTemplate template = new RabbitTemplate(connectionFactory());  
 return template;  
}

6）创建生产者（如果不需要在生产者中添加消息消费后的回调，不需要对rabbitTemplate设置ConfirmCallback对象，不用实现RabbitTemplate.ConfirmCallback接口。此处，由于不同的生产者需要对应不同的ConfirmCallback，如果rabbitTemplate设置为单例bean，则所有的rabbitTemplate实际的ConfirmCallback为最后一次申明的ConfirmCallback）

@Component  
public class Send implements RabbitTemplate.ConfirmCallback{  
 private RabbitTemplate rabbitTemplate;  
 @Autowired  
 public Send(RabbitTemplate rabbitTemplate) {  
 this.rabbitTemplate = rabbitTemplate;  
 }  
 public void sendMsg(String content) {  
 CorrelationData correlationData = new CorrelationData(UUID.*randomUUID*().toString());  
 rabbitTemplate.convertAndSend(RabbitMqConfig.*EXCHANGE*, RabbitMqConfig.*ROUTINGKEY*, content, correlationData);  
 }  
 @Override  
 public void confirm(CorrelationData correlationData, boolean ack, String cause) {  
 System.*out*.println("回调id：" + correlationData);  
 if(ack) {  
 System.*out*.println("消息成功消费");  
 } else {  
 System.*out*.println("消息消费失败：" + cause);  
 }  
 }  
}

1. 创建消费者
2. 配置类中设置：

-设置交换机类型

-将队列绑定到交换机

*/\*\*  
 \* 设置交换机类型  
 \** ***@return*** *\*/*@Bean  
public DirectExchange defaultExchange() {  
 return new DirectExchange(*EXCHANGE*);  
}  
*/\*\*  
 \* 队列持久  
 \** ***@return*** *\*/*@Bean  
public Queue queue() {  
 return new Queue("spring-boot-queue", true);  
}  
*/\*\*  
 \* 将队列绑定到交换机  
 \** ***@return*** *\*/*@Bean  
public Binding binding() {  
 return BindingBuilder.*bind*(queue()).to(defaultExchange()).with(*ROUTINGKEY*);  
}

1. 消费消息

@Bean  
public SimpleMessageListenerContainer messageContainer() {  
 SimpleMessageListenerContainer container = new SimpleMessageListenerContainer(connectionFactory());  
 container.setQueues(queue());  
 container.setExposeListenerChannel(true);  
 container.setMaxConcurrentConsumers(1);  
 container.setConcurrentConsumers(1);  
 container.setAcknowledgeMode(AcknowledgeMode.*MANUAL*);  
 container.setMessageListener(new ChannelAwareMessageListener() {  
 @Override  
 public void onMessage(Message message, Channel channel) throws Exception {  
 byte[] body = message.getBody();  
 System.*out*.println("接收消息：" + new String(body));  
 //通知服务端消息已经投递

channel.basicAck(message.getMessageProperties().getDeliveryTag(), false);  
 }  
 });  
 return container;  
}

# 5、springboot redis结合（一）

1）安装redis

2）导入Maven依赖

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

3）设置redis配置项

spring.redis.database=1  
spring.redis.host=127.0.0.1  
spring.redis.port=6379  
spring.redis.password=  
spring.redis.pool.max-active=8  
spring.redis.pool.max-wait=-1  
spring.redis.pool.max-idle=8  
spring.redis.pool.min-idle=0  
spring.redis.timeout=0

4）要启用缓存支持，需要创建一个新的 CacheManager bean。CacheManager 接口有很多实现，和 Redis 的集成，用 RedisCacheManager。Redis 不是应用的共享内存，它只是一个内存服务器，就像 MySql，需要将应用连接到它并使用某种“语言”进行交互，因此还需要一个连接工厂以及一个 Spring 和 Redis 对话要用的 RedisTemplate。

@EnableCaching  
@Configuration  
public class RedisConfig extends CachingConfigurerSupport {  
 @SuppressWarnings("rawtypes") //忽略警告  
 @Bean  
 public CacheManager cacheManager(RedisTemplate redisTemplate) {  
 RedisCacheManager rcm = new RedisCacheManager(redisTemplate);  
 rcm.setDefaultExpiration(30);  
 return rcm;  
 }  
  
 */\*\*  
 \** ***@param*** *factory  
 \** ***@return*** *\*/* @SuppressWarnings({"rawtypes", "unchecked"})  
 @Bean  
 public RedisTemplate<String, String> redisTemplate(RedisConnectionFactory factory) {  
 StringRedisTemplate template = new StringRedisTemplate(factory);  
 //设置序列化工具  
 Jackson2JsonRedisSerializer jackson2JsonRedisSerializer = new Jackson2JsonRedisSerializer(Object.class);  
 ObjectMapper om = new ObjectMapper();  
 //去除getter,setter等的依赖  
 om.setVisibility(PropertyAccessor.*ALL*, JsonAutoDetect.Visibility.*ANY*);  
 //开始使Jackson序列化类型信息  
 //JAVA\_LANG\_OBJECT: 当对象属性类型为Object时生效  
 //OBJECT\_AND\_NON\_CONCRETE: 当对象属性类型为Object或者非具体类型（抽象类和接口）时生效  
 //NON\_CONCRETE\_AND+\_ARRAYS: 同上, 另外所有的数组元素的类型都是非具体类型或者对象类型  
 //NON\_FINAL: 对所有非final类型或者非final类型元素的数组  
 om.enableDefaultTyping(ObjectMapper.DefaultTyping.*NON\_FINAL*);  
 jackson2JsonRedisSerializer.setObjectMapper(om);  
 template.setValueSerializer(jackson2JsonRedisSerializer);  
 //不是注入方法的话，必须调用它。初始化RedisTemplate  
 template.afterPropertiesSet();  
 return template;  
 }  
}

1. redisTemplate(RedisConnectionFactory factory) 方法主要进行了序列化操作，

使用Json方式:

\* 当我们的数据存储到Redis的时候，我们的键（key）和值（value）都是通过Spring提供的Serializer序列化到数据库的。

\* RedisTemplate默认使用的是JdkSerializationRedisSerializer，StringRedisTemplate默认使用的是StringRedisSerializer。

\* Spring Data JPA为我们提供了下面的Serializer：

\* GenericToStringSerializer、Jackson2JsonRedisSerializer、JacksonJsonRedisSerializer、JdkSerializationRedisSerializer、OxmSerializer、StringRedisSerializer。

\* 在此我们将自己配置RedisTemplate并定义Serializer。

Jackson2JsonRedisSerializer： 使用Jackson库将对象序列化为JSON字符串。优点是速度快，序列化后的字符串短小精悍，不需要实现Serializable接口。但缺点也非常致命，那就是此类的构造函数中有一个类型参数，必须提供要序列化对象的类型信息(.class对象)。 通过查看源代码，发现其只在反序列化过程中用到了类型信息。

1. @EnableCachingh和@Configuration 注解不能忘记加
2. 开启redis，直接用controller测试

@RestController  
@RequestMapping("/redis")  
public class RedisController {  
 @Autowired  
 private RedisTemplate redisTemplate;  
 @Autowired  
 private OrderRepository orderRepository;  
 @RequestMapping("/getData")  
 @Cacheable(value = "order\_data", key = "'order\_' + #p0")  
 public Order getData(@Param("id") Integer id) {  
// String name = "order\_" + id;  
 Order order = orderRepository.findByOrderId(id);  
// redisTemplate.opsForValue().set(name, JSONObject.toJSONString(order));  
 System.*out*.println("若下面没出现“无缓存的时候调用”字样且能打印出数据表示测试成功");  
 return order;  
 }  
}

# rabbitmq（二）消息发送确认

参考：<https://www.jianshu.com/p/2c5eebfd0e95>

<https://blog.csdn.net/qq315737546/article/details/54176560>

如果一个 Queue 没被任何消费者订阅，那么这个 Queue 中的消息会被 Cache（缓存），当有消费者订阅时则会立即发送，当 Message 被消费者正确接收时，就会被从 Queue 中移除。

1、通过实现 ConfirmCallback 接口，消息发送到 Broker 后触发回调，确认消息是否到达 Broker 服务器，也就是只确认是否正确到达 Exchange 中。

1）配置文件添加：

spring.rabbitmq.publisher-confirms=**true**

1. 继承RabbitTemplate.ConfirmCallback，init() 函数指定ConfirmCallback，

注解@PostConstruct，修饰的方法会在服务器加载Servlet的时候运行，并且只会被服务器执行一次。

@Component  
public class Send implements RabbitTemplate.ConfirmCallback{  
 private RabbitTemplate rabbitTemplate;  
 @Autowired  
 public Send(RabbitTemplate rabbitTemplate) {  
 this.rabbitTemplate = rabbitTemplate;  
 }  
 public void sendMsg(String content) {  
 CorrelationData correlationData = new CorrelationData(UUID.*randomUUID*().toString());  
 rabbitTemplate.convertAndSend(RabbitMqConfig.*EXCHANGE*, RabbitMqConfig.*ROUTINGKEY*, content, correlationData);  
 }  
 @PostConstruct  
 public void init() {  
 rabbitTemplate.setConfirmCallback(this);  
 }  
 @Override  
 public void confirm(CorrelationData correlationData, boolean ack, String cause) {  
 System.*out*.println("回调id：" + correlationData);  
 if(ack) {  
 System.*out*.println("消息成功消费");  
 } else {  
 System.*out*.println("消息消费失败：" + cause);  
 }  
 }  
}

1. 继承RabbitTemplate.ReturnCallback，通过实现 ReturnCallback 接口，启动消息失败返回，比如路由不到队列时触发回调。
2. 添加配置文件

spring.rabbitmq.publisher-returns=**true**

2）

@Component  
public class Send2 implements RabbitTemplate.ReturnCallback {  
 @Autowired  
 private RabbitTemplate rabbitTemplate;  
 @PostConstruct  
 public void init() {  
 rabbitTemplate.setReturnCallback(this);  
 }  
 @Override  
 public void returnedMessage(Message message, int replyCode, String replyText, String exchange, String routingKey) {  
 System.*out*.println("消息主体 message : "+message);  
 System.*out*.println("消息主体 message : "+replyCode);  
 System.*out*.println("描述："+replyText);  
 System.*out*.println("消息使用的交换器 exchange : "+exchange);  
 System.*out*.println("消息使用的路由键 routing : "+routingKey);  
 }  
 public void sendMsg(String content) {  
 CorrelationData correlationData = new CorrelationData(UUID.*randomUUID*().toString());  
 rabbitTemplate.convertAndSend(RabbitMqConfig.*EXCHANGE*, RabbitMqConfig.*ROUTINGKEY*, content, correlationData);  
 }  
}

3）测试

以下四种情况：

*exchange,queue 都正确；*

*exchange 错误,queue 正确；*

*exchange 正确,queue 错误；*

*exchange 错误,queue 错误；*

在两个Send类中添加函数

public void sendMsg(String exchange, String key, String content) {  
 CorrelationData correlationData = new CorrelationData(UUID.*randomUUID*().toString());  
 rabbitTemplate.convertAndSend(exchange, key, content, correlationData);  
}

由于 EXCHANGE = "spring-boot-exchange";

ROUTINGKEY = "spring-boot-routingKey";

针对以上四种情况，分别调用：

send.sendMsg(*EXCHANGE*,*ROUTINGKEY*,message);

send.sendMsg(*EXCHANGE* + "NO",*ROUTINGKEY*,message);

send.sendMsg(*EXCHANGE*,*ROUTINGKEY* + "NO",message);

send.sendMsg(*EXCHANGE* + "NO",*ROUTINGKEY* + "NO",message);

send2.sendMsg(*EXCHANGE*,*ROUTINGKEY*,message);

send2.sendMsg(*EXCHANGE* + "NO",*ROUTINGKEY*,message);

send2.sendMsg(*EXCHANGE*,*ROUTINGKEY* + "NO",message);

send2.sendMsg(*EXCHANGE* + "NO",*ROUTINGKEY* + "NO",message);

注：实现return的init()函数中需设置：

rabbitTemplate.setMandatory(true);

结论如下：

1. exchange,queue 都正确,confirm被回调, ack=true; return不被回调
2. exchange 错误,queue 正确,confirm被回调,ack=false; return不被回调; 控制台会有[Error]信息
3. exchange 正确,queue 错误 ,confirm被回调, ack=true; return被回调 replyText:NO\_ROUTE
4. 4、exchange 错误,queue 错误,confirm被回调, ack=false; return不被回调; 控制台会有[Error]信息

综上所述：

如果消息没有到exchange,则confirm回调,ack=false

如果消息到达exchange,则confirm回调,ack=true

exchange到queue成功,则不回调return

exchange到queue失败,则回调return(需设置mandatory=true,否则不回回调,消息就丢了)

# 定时任务（未完）

1、启动执行定时任务

@Component  
@EnableScheduling  
public class ScheduledTasks {  
 @Scheduled(cron = "0 \*/1 \* \* \* \* ")  
 public void displayTask() {  
 System.*out*.println("时间为：" + LocalDateTime.*now*());  
 }  
}

每一分钟执行一次，@Scheduled 参数可以接受两种定时的设置，一种是我们常用的cron="\*/6 \* \* \* \* ?",一种是 fixedRate = 6000，两种都表示每隔六秒打印一下内容。

fixedRate 参数：

* @Scheduled(fixedRate = 6000) ：上一次开始执行时间点之后6秒再执行
* @Scheduled(fixedDelay = 6000) ：上一次执行完毕时间点之后6秒再执行
* @Scheduled(initialDelay=1000, fixedRate=6000) ：第一次延迟1秒后执行，之后按fixedRate的规则每6秒执行一次

corn表达式：

CronTrigger配置完整格式为： [秒] [分] [小时] [日] [月] [周] [年]

2、结合quartz

参考：<http://www.wanqhblog.top/2018/02/01/SpringBootTaskSchedule/>

<https://blog.csdn.net/upxiaofeng/article/details/79415108>

1）导入Maven依赖

如果SpringBoot版本是2.0.0以后的：

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-quartz</artifactId>

</dependency>

如果是1.5.9则要使用以下添加依赖：

<dependency>

<groupId>org.quartz-scheduler</groupId>

<artifactId>quartz</artifactId>

<version>2.3.0</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context-support</artifactId>

</dependency>

1. 定义Job类

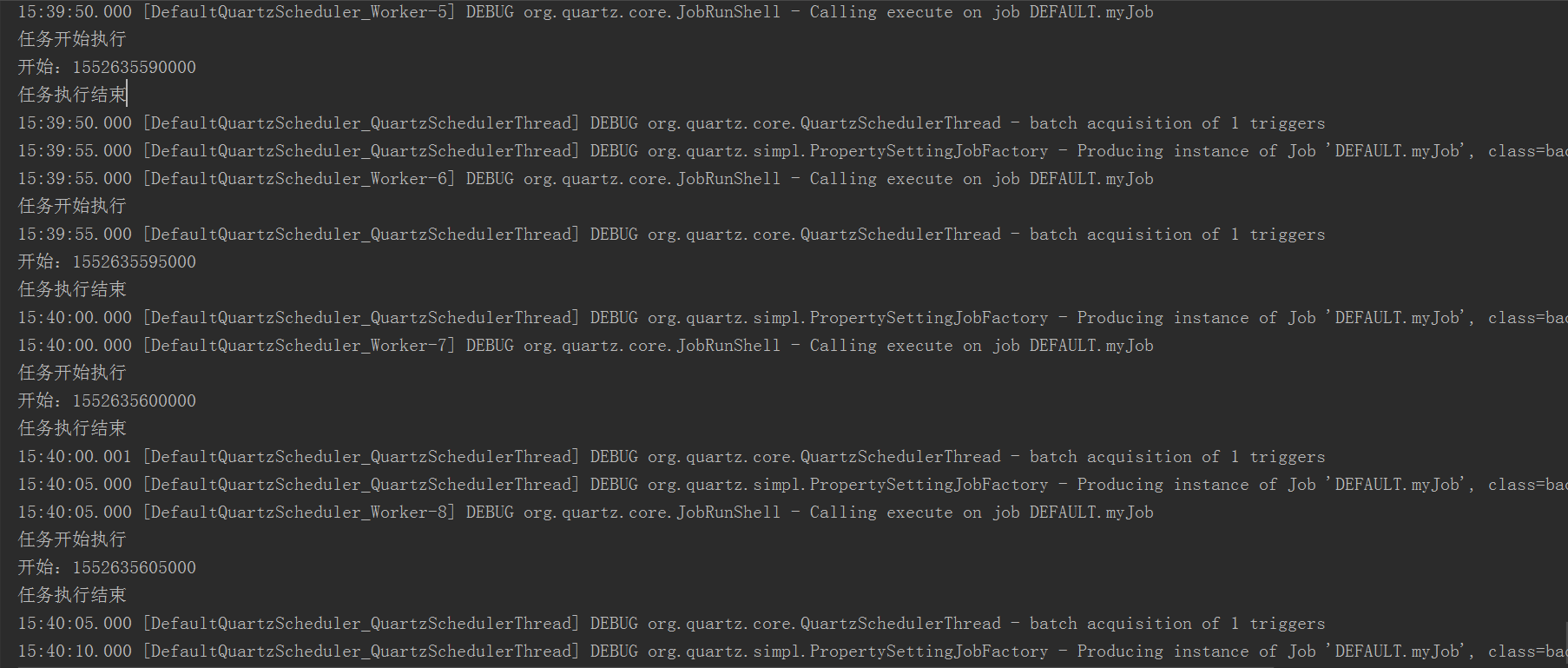
public class SchedulerQuartzJob implements Job {  
 @Override  
 public void execute(JobExecutionContext jobExecutionContext) throws JobExecutionException {  
 before();  
 System.*out*.println("开始：" + System.*currentTimeMillis*());  
 after();  
 }  
 private void before() {  
 System.*out*.println("任务开始执行");  
 }  
 private void after() {  
 System.*out*.println("任务执行结束");  
 }  
}

1. main函数测试

public class QuartzMain {  
 public static void main(String[] args) {  
 try {  
 // 1. 创建一个JodDetail实例 将该实例与Hello job class绑定 (链式写法)  
 JobDetail jobDetail = JobBuilder.*newJob*(SchedulerQuartzJob.class) // 定义Job类为HelloQuartz类，这是真正的执行逻辑所在  
 .withIdentity("myJob") // 定义name/group  
 .build();  
 // 打印当前的时间  
 SimpleDateFormat sf = new SimpleDateFormat("yyyy-MM-dd hh:mm:ss");  
 Date date = new Date();  
 System.*out*.println("current time is :" + sf.format(date));

.  
 CronTrigger trigger = (CronTrigger) TriggerBuilder.*newTrigger*()  
 .withIdentity("myTrigger", "group1")// 定义名字和组  
 .withSchedule( //定义任务调度的时间间隔和次数  
 CronScheduleBuilder  
 .*cronSchedule*("0/5 \* \* \* \* ?"))  
 .build();  
 // 3. 创建scheduler  
 SchedulerFactory sfact = new StdSchedulerFactory();  
 Scheduler scheduler = sfact.getScheduler();  
 // 4. 将trigger和jobdetail加入这个调度  
 scheduler.scheduleJob(jobDetail, trigger);  
 // 5. 启动scheduler  
 scheduler.start();  
 //scheduler执行2s后挂起  
 Thread.*sleep*(2000);  
 scheduler.standby();  
 //scheduler挂起3s后再次启动scheduler  
 Thread.*sleep*(3000);  
 scheduler.start();  
 } catch (Exception e) {  
 e.printStackTrace();  
 }  
 }  
}

控制台如下：



1. 结合quartz，完善代码
2. 创建Job类同上，新建工厂类和配置类

@Component  
public class JobFactory extends AdaptableJobFactory {  
 */\*\*  
 \* AutowireCapableBeanFactory接口是BeanFactory的子类  
 \* 可以连接和填充那些生命周期不被Spring管理的已存在的bean实例  
 \*/* private AutowireCapableBeanFactory factory;  
  
 public JobFactory(AutowireCapableBeanFactory factory) {  
 this.factory = factory;  
 }  
 @Override  
 protected Object createJobInstance(TriggerFiredBundle bundle) throws Exception {  
 Object job = super.createJobInstance(bundle);  
 factory.autowireBean(job);  
 return job;  
 }  
}

@Configuration  
public class QuartzConfig {  
 private JobFactory jobFactory;  
 public QuartzConfig(JobFactory factory) {  
 this.jobFactory = factory;  
 }  
 */\*\*  
 \* 配置SchedulerFactoryBean  
 \* 将一个方法产生为Bean并交给Spring容器管理  
 \*/* @Bean  
 public SchedulerFactoryBean schedulerFactoryBean() {  
 SchedulerFactoryBean factoryBean = new SchedulerFactoryBean();  
 factoryBean.setJobFactory(jobFactory);  
 return factoryBean;  
 }  
 @Bean(name = "scheduler")  
 public Scheduler scheduler() {  
 return schedulerFactoryBean().getScheduler();  
 }  
}

1. Quartz工具类

@Component  
public class SelfQuartzScheduler {  
 private Scheduler scheduler;  
  
 public SelfQuartzScheduler(Scheduler scheduler) {  
 this.scheduler = scheduler;  
 }  
 */\*\*  
 \* 执行所有任务  
 \** ***@throws*** *SchedulerException  
 \*/* public void startJob() throws SchedulerException {  
 startJob1(scheduler);  
 scheduler.start();  
 }  
 */\*\*  
 \* 获取Job信息  
 \** ***@param*** *name  
 \** ***@param*** *group  
 \** ***@return*** *\** ***@throws*** *SchedulerException  
 \*/* public String getJobInfo(String name, String group) throws SchedulerException {  
 TriggerKey triggerKey = new TriggerKey(name, group);  
 CronTrigger cronTrigger = (CronTrigger) scheduler.getTrigger(triggerKey);  
 return String.*format*("时间：%s, 状态：%s", cronTrigger.getCronExpression(), scheduler.getTriggerState(triggerKey).name());  
 }  
 */\*\*  
 \* 修改某个任务的执行时间  
 \** ***@param*** *name  
 \** ***@param*** *group  
 \** ***@param*** *time  
 \** ***@return*** *\** ***@throws*** *SchedulerException  
 \*/* public boolean modifyJob(String name, String group, String time) throws SchedulerException {  
 Date date = null;  
 TriggerKey triggerKey = new TriggerKey(name, group);  
 CronTrigger cronTrigger = (CronTrigger) scheduler.getTrigger(triggerKey);  
 String oldTime = cronTrigger.getCronExpression();  
 if(!oldTime.equals(time)) {  
 CronScheduleBuilder cronScheduleBuilder = CronScheduleBuilder.*cronSchedule*(time);  
 CronTrigger trigger = TriggerBuilder.*newTrigger*().withIdentity(name, group)  
 .withSchedule(cronScheduleBuilder).build();  
 date = scheduler.rescheduleJob(triggerKey, trigger);  
 }  
 return date != null;  
 }  
 */\*\*  
 \* 暂停所有任务  
 \** ***@throws*** *SchedulerException  
 \*/* public void pauseAll() throws SchedulerException {  
 scheduler.pauseAll();  
 }  
 */\*\*  
 \* 暂停某个任务  
 \** ***@param*** *name  
 \** ***@param*** *group  
 \** ***@throws*** *SchedulerException  
 \*/* public void pauseJob(String name, String group) throws SchedulerException {  
 JobKey jobKey = new JobKey(name, group);  
 JobDetail jobDetail = scheduler.getJobDetail(jobKey);  
 if(ObjectUtils.*isEmpty*(jobDetail)) {  
 return;  
 }  
 scheduler.pauseJob(jobKey);  
 }  
 */\*\*  
 \* 恢复所有任务  
 \*  
 \** ***@throws*** *SchedulerException  
 \*/* public void resumeAllJob() throws SchedulerException {  
 scheduler.resumeAll();  
 }  
 */\*\*  
 \* 恢复某个任务  
 \*  
 \** ***@param*** *name  
 \** ***@param*** *group  
 \** ***@throws*** *SchedulerException  
 \*/* public void resumeJob(String name, String group) throws SchedulerException {  
 JobKey jobKey = new JobKey(name, group);  
 JobDetail jobDetail = scheduler.getJobDetail(jobKey);  
 if (jobDetail == null)  
 return;  
 scheduler.resumeJob(jobKey);  
 }  
 */\*\*  
 \* 删除某个任务  
 \*  
 \** ***@param*** *name  
 \** ***@param*** *group  
 \** ***@throws*** *SchedulerException  
 \*/* public void deleteJob(String name, String group) throws SchedulerException {  
 JobKey jobKey = new JobKey(name, group);  
 JobDetail jobDetail = scheduler.getJobDetail(jobKey);  
 if (jobDetail == null)  
 return;  
 scheduler.deleteJob(jobKey);  
 }  
 private void startJob1(Scheduler scheduler) throws SchedulerException {  
 // 创建一个JodDetail实例 将该实例与SchedulerQuartzJob class绑定 (链式写法)  
 JobDetail jobDetail = JobBuilder.*newJob*(SchedulerQuartzJob.class) //定义Job类，真正的执行逻辑所在  
 .withIdentity("job1", "group1") //定义name/group  
 .build();  
 //定义任务调度的时间间隔和次数  
 CronScheduleBuilder cronScheduleBuilder = CronScheduleBuilder.*cronSchedule*("0/5 \* \* \* \* ?");  
 CronTrigger cronTrigger = TriggerBuilder.*newTrigger*().withIdentity("job1", "group1")  
 .withSchedule(cronScheduleBuilder).build();  
 // 将trigger和jobdetail加入这个调度  
 scheduler.scheduleJob(jobDetail, cronTrigger);  
 }  
  
}

1. 直接调用工具类的方法

# OAuth认证（一）

有关OAuth介绍：

参考：<http://www.ruanyifeng.com/blog/2014/05/oauth_2_0.html>

带程序：<https://blog.csdn.net/weixin_42033269/article/details/80086422>

OAuth2为我们提供了四种授权方式：

1、授权码模式（authorization code）   
2、简化模式（implicit）   
3、密码模式（resource owner password credentials）   
4、客户端模式（client credentials）

1. 参考：<https://www.jianshu.com/p/ded9dc32f550>

导入依赖

<dependency>  
 <groupId>org.springframework.security.oauth</groupId>  
 <artifactId>spring-security-oauth2</artifactId>  
</dependency>  
<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-security</artifactId>  
</dependency>

1. 添加PO类及JPA类，并配置两个控制器用来区分我们配置OAuth2是否已经生效：

@RestController  
@RequestMapping("/hello")  
public class HelloWorldController {  
 @RequestMapping(method = RequestMethod.*GET*)  
 public String helloWorld() {  
 return "hello world";  
 }  
}

@RestController  
@RequestMapping("/secure")  
public class SecureController {  
 @RequestMapping(method = RequestMethod.*GET*)  
 public String helloWorld() {  
 return "hello world";  
 }  
}

1. 自定义UserDetailsService，要将UserInfo提供给权限系统，需要实现自定义的UserDetailsService，该类只包含一个方法，实际运行中，系统会通过这个方法获得登录用户的信息

@Service  
public class SelfUserDetailsService implements UserDetailsService {  
 @Autowired  
 private UserRepository userRepository;  
 @Override  
 public UserDetails loadUserByUsername(String login) throws UsernameNotFoundException {  
 String lowerCaseLogin = login.toLowerCase();  
 User userFromDatabase = userRepository.findByUsernameCaseInsensitive(lowerCaseLogin);  
 if(ObjectUtils.*isEmpty*(userFromDatabase)) {  
 throw new UsernameNotFoundException("User " + lowerCaseLogin + " was not found in database");  
 }  
 //获取用户所有权限形成SpringSecurity需要的集合  
 Collection<GrantedAuthority> grantedAuthorities = new ArrayList<>();  
 for(Authority authority : userFromDatabase.getAuthority()) {  
 GrantedAuthority grantedAuthority = new SimpleGrantedAuthority((authority.getName()));  
 grantedAuthorities.add(grantedAuthority);  
 }  
 //返回SpringSecurity需要的集合  
 return new org.springframework.security.core.userdetails.User(  
 userFromDatabase.getUsername(),  
 userFromDatabase.getPassword(),  
 grantedAuthorities);  
 }  
}

1. 开启SpringSecurity配置，注入了上面我们自定义的SelfUserDetailsService 以及用户密码验证规则，我们使用ignoring()方法排除了HelloWorldController内的公开方法，这里可以配置通配符的形式排除。

@Configuration  
@EnableWebSecurity  
public class SecurityConfiguration extends WebSecurityConfigurerAdapter {  
 @Autowired  
 private SelfUserDetailsService selfUserDetailsService;  
 @Bean //配置匹配用户时密码规则  
 public PasswordEncoder passwordEncoder() {  
 return new StandardPasswordEncoder();  
 }  
 @Autowired  
 public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {  
 auth.userDetailsService(selfUserDetailsService).passwordEncoder(passwordEncoder());  
 }  
  
 @Override  
 public void configure(WebSecurity web) throws Exception {  
 web.ignoring().antMatchers("/hello");  
 super.configure(web);  
 }  
  
 @Override  
 @Bean  
 public AuthenticationManager authenticationManagerBean() throws Exception {  
 return super.authenticationManagerBean();  
 }  
 //开启全局方法拦截  
 @EnableGlobalMethodSecurity(prePostEnabled = true, jsr250Enabled = true)  
 public static class GlobalSecurityExpressionHandler extends GlobalMethodSecurityConfiguration {  
 @Override  
 protected MethodSecurityExpressionHandler createExpressionHandler() {  
 return new OAuth2MethodSecurityExpressionHandler();  
 }  
 }  
}

1）StandardPasswordEncoder 加密方法，采用SHA-256算法，迭代1024次，使用一个密钥(site-wide secret)以及8位随机盐对原密码进行加密。

2）Spring Security默认是禁用注解的，要想开启注解，需要在继承WebSecurityConfigurerAdapter的类上加@EnableGlobalMethodSecurity注解，来判断用户对某个控制层的方法是否具有访问权限，部分参数如下：

@EnableGlobalMethodSecurity(securedEnabled=true)

开启@Secured 注解过滤权限

@EnableGlobalMethodSecurity(jsr250Enabled=true)

开启@RolesAllowed 注解过滤权限

@EnableGlobalMethodSecurity(prePostEnabled=true)

使用表达式时间方法级别的安全性 4个注解可用：

@PreAuthorize 在方法调用之前,基于表达式的计算结果来限制对方法的访问

@PostAuthorize 允许方法调用,但是如果表达式计算结果为false,将抛出一个安全性异常

@PostFilter 允许方法调用,但必须按照表达式来过滤方法的结果

@PreFilter 允许方法调用,但必须在进入方法之前过滤输入值

1. 配置安全资源服务器

@Configuration  
public class OAuth2Configuration {  
 @Configuration  
 @EnableResourceServer  
 protected static class ResourceServerConfiguration extends ResourceServerConfigurerAdapter {  
 @Autowired  
 private CustomAuthenticationEntryPoint customAuthenticationEntryPoint;  
 @Autowired  
 private CustomLogoutSuccessHandler customLogoutSuccessHandler;  
 @Override  
 public void configure(HttpSecurity http) throws Exception {  
 http.exceptionHandling()  
 .authenticationEntryPoint(customAuthenticationEntryPoint)  
 .and()  
 .logout()  
 .logoutUrl("oauth/logout")  
 .logoutSuccessHandler(customLogoutSuccessHandler)  
 .and()  
 .authorizeRequests()  
 .antMatchers("/hello/").permitAll()  
 .antMatchers("/secure/\*\*").authenticated();  
 }  
 }  
}

@EnableResourceServer是Oauth2 资源服务器的便利方法，开启了一个spring security的filter，这个filter通过一个Oauth2的token进行认证请求。使用者应该增加这个注解，并提供一个ResourceServerConfigurer类型的Bean(例如通过ResouceServerConfigurerAdapter)来指定资源(url路径和资源id)的细节。为了利用这个filter，你必须在你的应用中的某些地方EnableWebSecurity，或者使用这个注解的地方，或者其他别的地方。

1. 自定义401错误码内容，因为整合SpringSecurity的缘故，我们需要配置登出时清空对应的access\_token控制以及自定义401错误内容

@Component  
@Slf4j  
public class CustomAuthenticationEntryPoint implements AuthenticationEntryPoint {  
 @Override  
 public void commence(HttpServletRequest httpServletRequest, HttpServletResponse httpServletResponse, AuthenticationException e) throws IOException, ServletException {  
 *log*.info("Pre-authenticated entry point called. Rejecting access");  
 httpServletResponse.sendError(HttpServletResponse.*SC\_UNAUTHORIZED*, "Access Denied");  
 }  
}

7、定义登出控制

@Component  
public class CustomLogoutSuccessHandler extends AbstractAuthenticationTargetUrlRequestHandler implements LogoutSuccessHandler {  
 private static final String *BEARER\_AUTHENTICATION* = "Bearer";  
 private static final String *HEADER\_AUTHENTICATION* = "authorization";  
  
 @Autowired  
 private TokenStore tokenStore;  
  
 @Override  
 public void onLogoutSuccess(HttpServletRequest httpServletRequest, HttpServletResponse httpServletResponse, Authentication authentication) throws IOException, ServletException {  
 String token = httpServletRequest.getHeader(*HEADER\_AUTHENTICATION*);  
 if(!ObjectUtils.*isEmpty*(token) && token.startsWith(*BEARER\_AUTHENTICATION*)) {  
 OAuth2AccessToken oAuth2AccessToken = tokenStore.readAccessToken(token.split(" ")[0]);  
 if(oAuth2AccessToken != null) {  
 tokenStore.removeAccessToken(oAuth2AccessToken);  
 }  
 }  
 }  
}

注意：此处tokenStore注入为失败的，有两种方法可令其成功：

1. 修改@Autowired  
    private TokenStore tokenStore; 为：

@Bean  
public TokenStore tokenStore() {  
 return new InMemoryTokenStore();  
}  
private TokenStore tokenStore = tokenStore();

1. （推荐）配置步骤8的内容

8、开启OAuth2验证服务器，

1>创建AuthorizationConfiguration

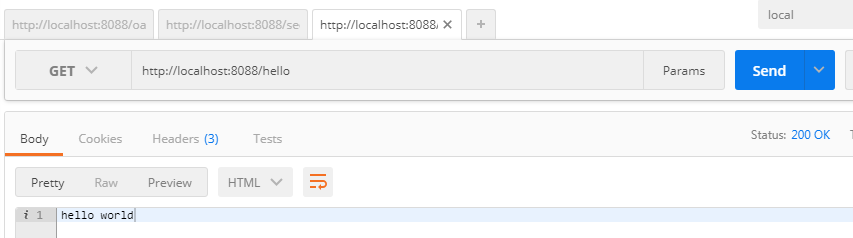
的类继承自AuthorizationServerConfigurerAdapter 并且实现了EnvironmentAware （读取properties文件需要）接口，并使用@EnableAuthorizationServer注解开启了验证服务器。此处使用SpringSecurityOAuth2内定义的JdbcStore来操作数据库中的Token。

@Configuration  
@EnableAuthorizationServer  
public class AuthorizationConfiguration extends AuthorizationServerConfigurerAdapter implements EnvironmentAware {  
 private static final String *ENV\_OAUTH* = "authentication.oauth";  
 private static final String *PROP\_CLIENTID* = "clientid";  
 private static final String *PROP\_SECRET* = "secret";  
 private static final String *PROP\_TOKEN\_VALIDITY\_SECONDS* = "tokenValiditySecends";  
 private RelaxedPropertyResolver propertyResolver;  
 @Autowired  
 private DataSource dataSource;  
 @Bean  
 public TokenStore tokenStore() {  
 return new JdbcTokenStore(dataSource);  
 }  
 @Autowired  
 @Qualifier("authenticationManagerBean")  
 private AuthenticationManager authenticationManager;  
 @Override  
 public void configure(AuthorizationServerEndpointsConfigurer endpoints) throws Exception {  
 endpoints.tokenStore(tokenStore())  
 .authenticationManager(authenticationManager);  
 }  
 @Override  
 public void setEnvironment(Environment environment) {}  
}

2> 可以通过SpringDataJPA自定义Sotre，configure和setEnvironment更改如下：

@Override  
public void configure(ClientDetailsServiceConfigurer clients) throws Exception {  
 clients.inMemory()  
 .withClient(propertyResolver.getProperty(*PROP\_CLIENTID*))  
 .scopes("read", "write")  
 .authorities(AuthorityEnum.*ROLE\_ADMIN*.name(), AuthorityEnum.*ROLE\_USER*.name())  
 .authorizedGrantTypes("password", "refresh\_token")  
 .secret(propertyResolver.getProperty(*PROP\_SECRET*))  
 .accessTokenValiditySeconds(propertyResolver.getProperty(*PROP\_TOKEN\_VALIDITY\_SECONDS*, Integer.class, 1800));  
}  
  
@Override  
public void setEnvironment(Environment environment) {  
 this.propertyResolver = new RelaxedPropertyResolver(environment, *ENV\_OAUTH*);  
}

这时候访问/hello可以访问，/secure路径被拦截



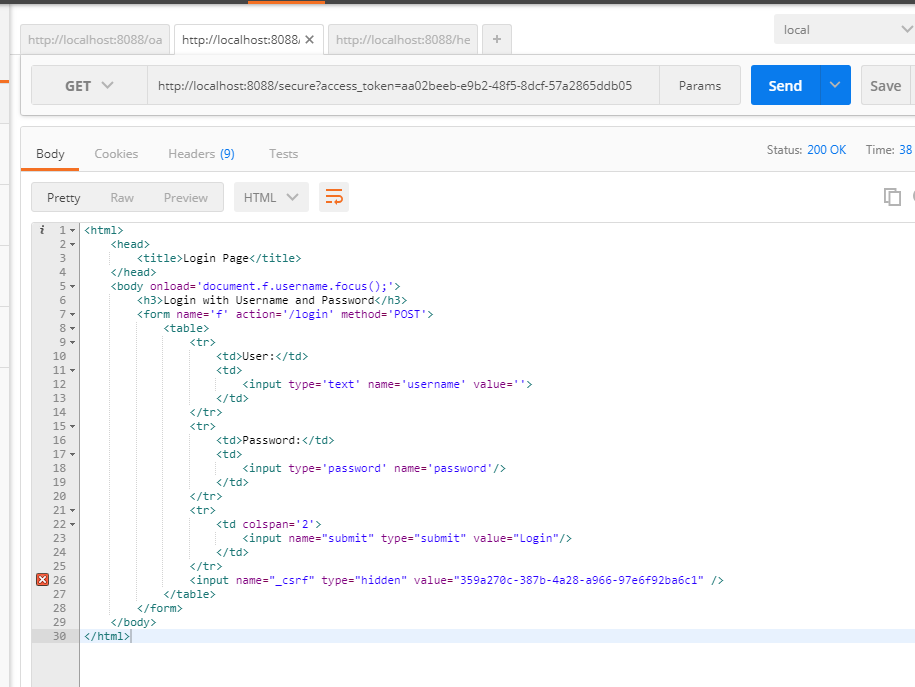
此时没有从数据库中读取而是使用了内存中获取，需要在配置文件中加入内容（此处并没有被用到，估测是那几个常量需要取这里的值）：

authentication.oath.clientid=tangminyan\_home\_pc  
authentication.oath.secret=tangminyan\_secret  
authentication.oath.tokenValidityInSeconds=1800

1. 为实现access\_token访问，有以下几个步骤：
2. 配置文件添加如下内容：

security.oauth2.resource.filter-order=3

这个配置的意思时，将源拦截的过滤器运行顺序放到第3个执行，也就是在oauth2的认证服务器后面执行，不然会直接被拦截，如下，采用token访问/secure 还是被拦截

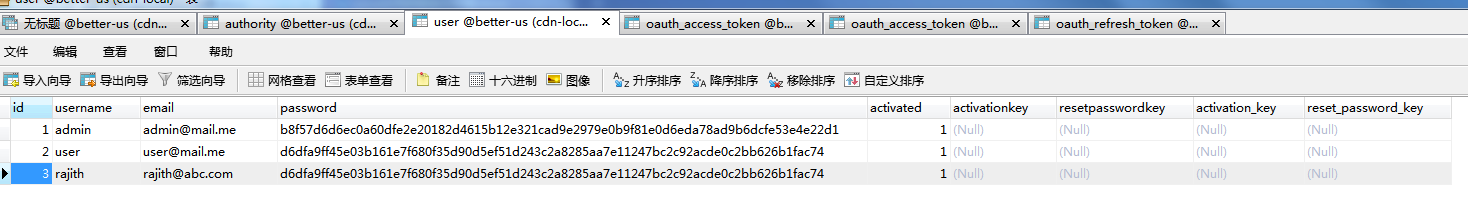


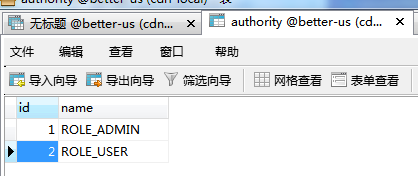
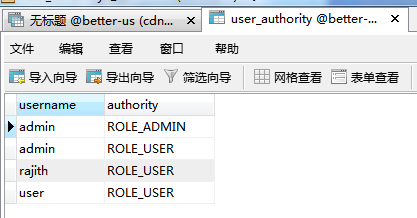
1. 建依赖表

Spring Security OAuth2有一个奇葩的设计，那就是它将与access\_token相关的所有属于都封装到OAuth2AccessToken中，然后保存时会直接将该对象序列化成字节写入数据库。

新建oauth\_access\_token 和 oauth\_refresh\_token，并在User和Authority中插入相应数据，

**DROP TABLE** IF **EXISTS** `oauth\_access\_token`;  
**CREATE TABLE** `oauth\_access\_token` (  
 `token\_id` **varchar**(256) **DEFAULT NULL**,  
 `token` blob,  
 `authentication\_id` **varchar**(256) **DEFAULT NULL**,  
 `user\_name` **varchar**(256) **DEFAULT NULL**,  
 `client\_id` **varchar**(256) **DEFAULT NULL**,  
 `authentication` blob,  
 `refresh\_token` **varchar**(256) **DEFAULT NULL**) ENGINE=MyISAM **DEFAULT** CHARSET=utf8;  
  
**DROP TABLE** IF **EXISTS** `oauth\_refresh\_token`;  
**CREATE TABLE** `oauth\_refresh\_token` (  
 `token\_id` **varchar**(256) **DEFAULT NULL**,  
 `token` blob,  
 `authentication` blob  
) ENGINE=MyISAM **DEFAULT** CHARSET=utf8;



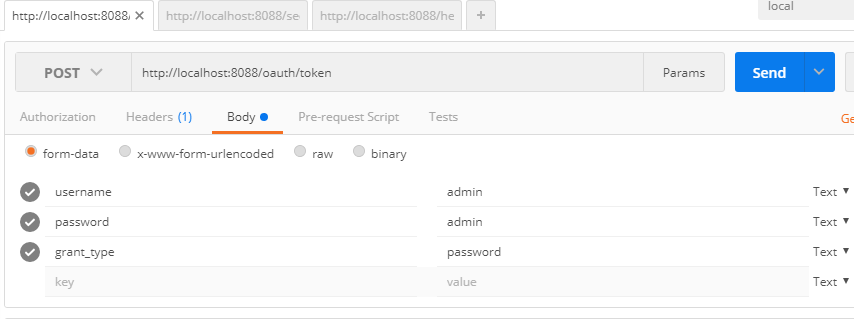
 

1. 修改AuthorizationConfiguration 中的常量值和configure函数：

@Value("${authentication.oath.clientid}")  
private String PROP\_CLIENTID;  
@Value("${authentication.oath.secret}")  
private String PROP\_SECRET;  
@Value("${authentication.oath.tokenValidityInSeconds}")  
private Integer PROP\_TOKEN\_VALIDITY\_SECONDS;

@Override  
public void configure(ClientDetailsServiceConfigurer clients) throws Exception {  
 clients.inMemory()  
 .withClient(PROP\_CLIENTID)  
 .scopes("read", "write")  
 .authorities(AuthorityEnum.*ROLE\_ADMIN*.name(), AuthorityEnum.*ROLE\_USER*.name())  
 .authorizedGrantTypes("password", "refresh\_token")  
 .secret(PROP\_SECRET)  
 .accessTokenValiditySeconds(PROP\_TOKEN\_VALIDITY\_SECONDS);  
}

1. 访问 /oauth/token，postman设置如下：



得出如下结果，其中，

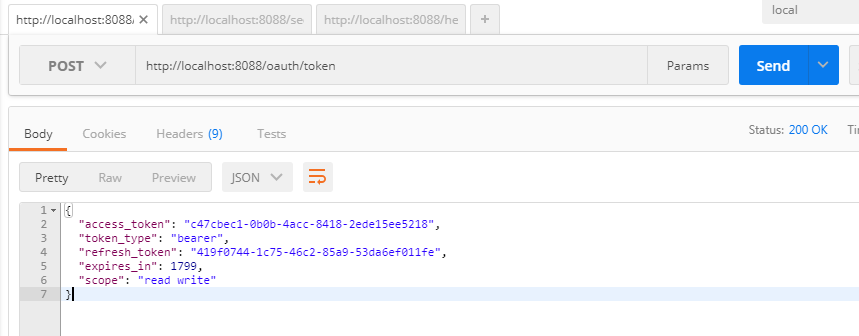
access\_token：本地访问获取到的access\_token，会自动写入到数据库中。

token\_type：获取到的access\_token的授权方式

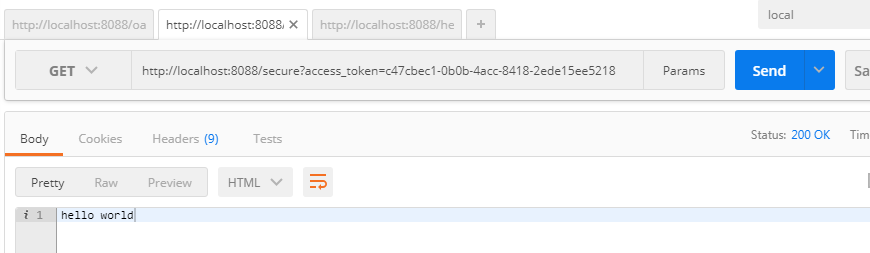
refersh\_token：刷新token时所用到的授权token

expires\_in：有效期（从获取开始计时，值秒后过期）

scope：客户端的接口操作权限（read：读，write：写）



此时可访问/secure



# 9、spring batch

参考：<https://segmentfault.com/a/1190000016278038>