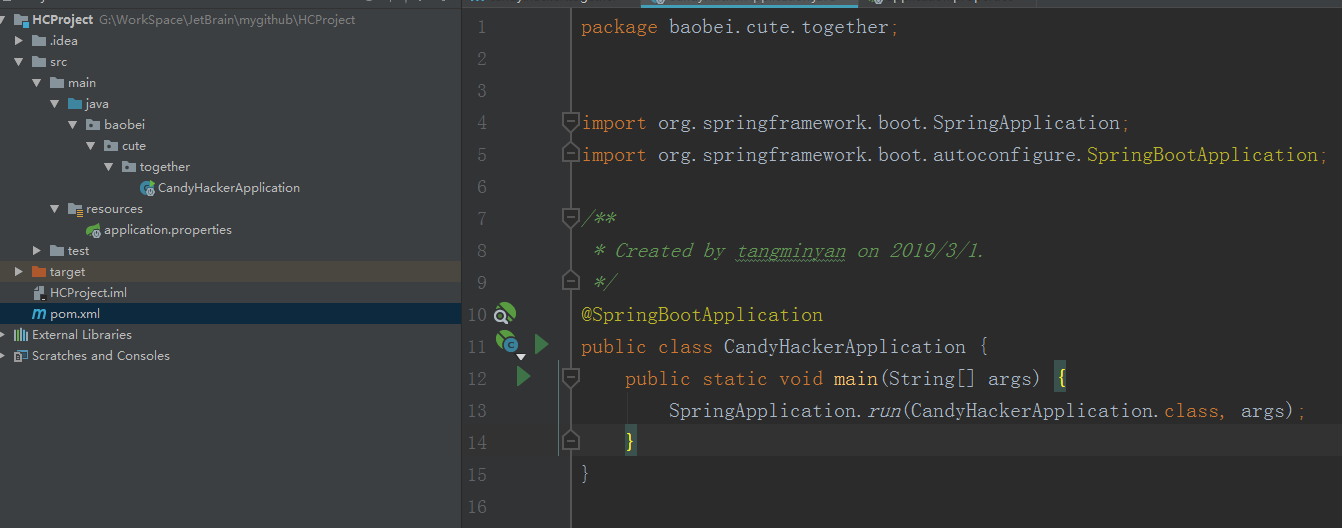
# 基础搭建

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>

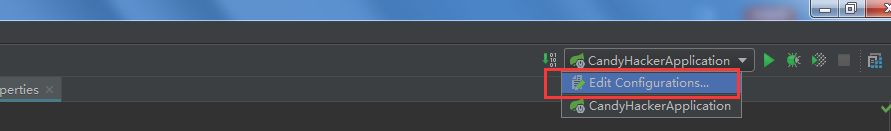
1. 新建启动函数

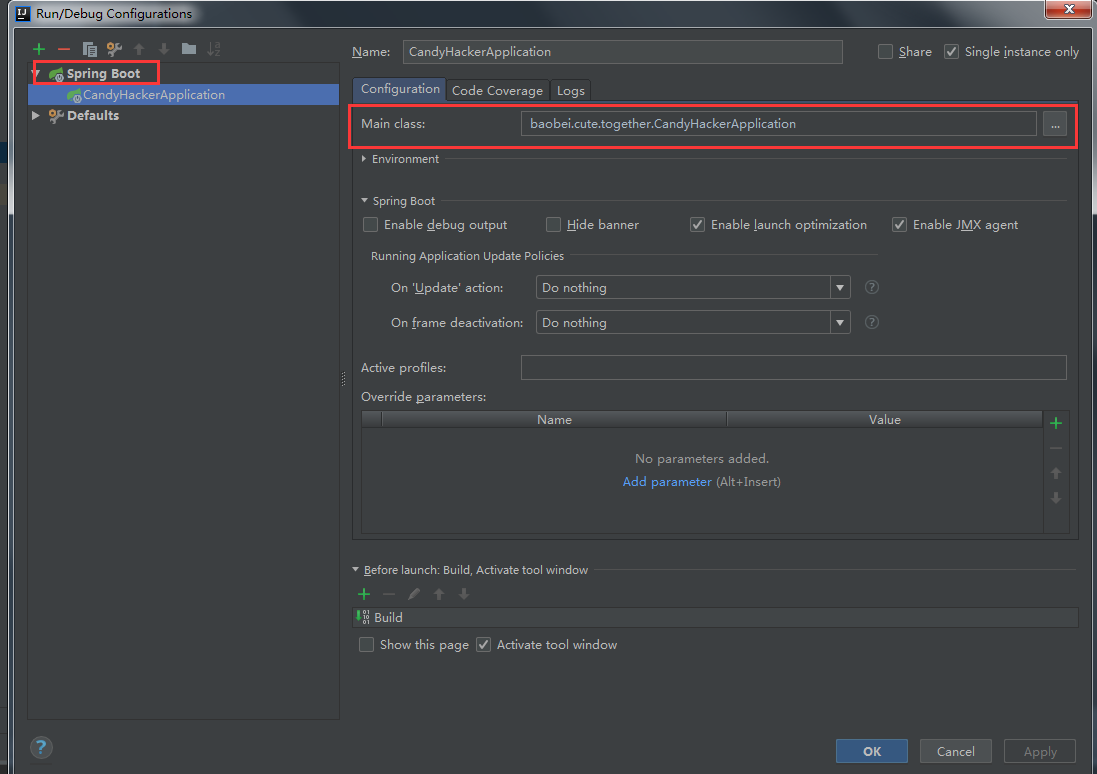


1. resources下新建application.properties配置文件，更改端口号(可不改)



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*);  
 }  
}

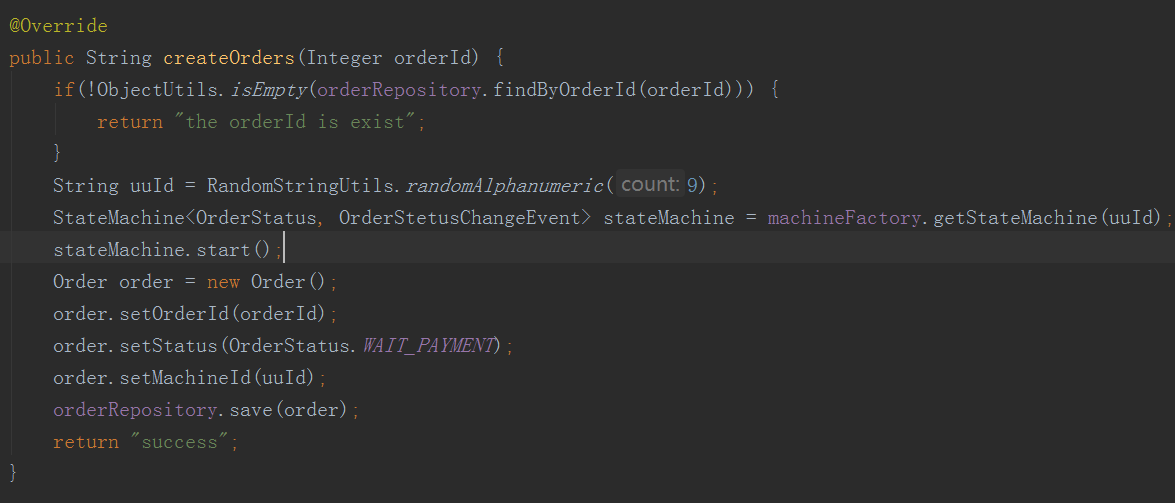
5）设置监听

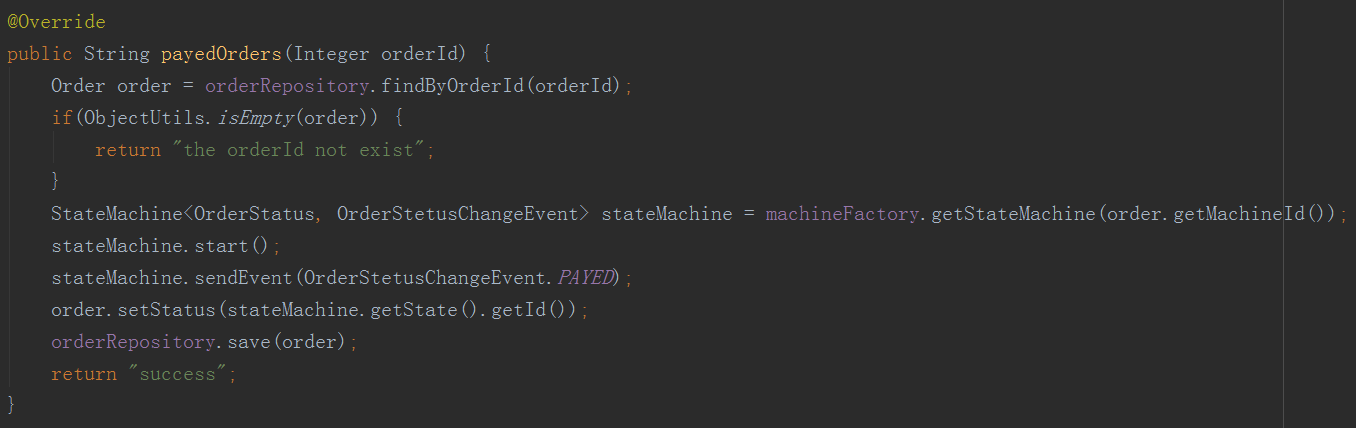
@WithStateMachine  
@Slf4j  
public class OrderEventConfig {  
 @OnTransition(target = "UNPAYED")  
 public void create() {  
 *log*.info("待支付");  
 }  
 @OnTransition(source = "UNPAYED", target = "WAITING\_FOR\_RECEIVE")  
 public void pay() {  
 *log*.info("支付完成，待收货");  
 }  
 @OnTransition(source = "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);  
 //此处可设置redis过期时间  
 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), 20);  
 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;  
 }  
}