**生产者消费者模式之Redis实现的消息队列代码原理**

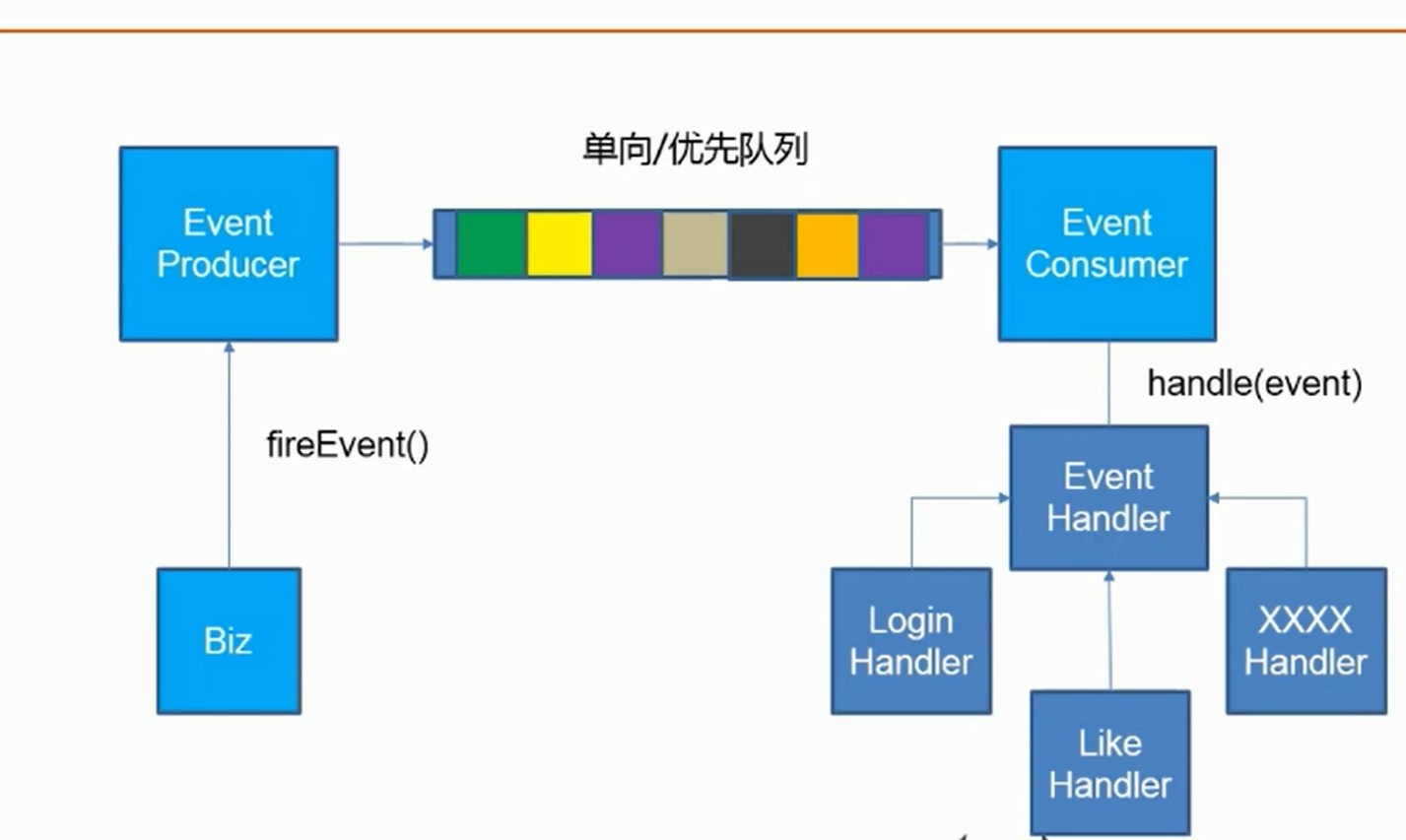
Redis提供了两种方式来作消息队列。   
一个是使用**生产者消费模式模式**，   
另一个就是**发布订阅者模式**。   
前者会让一个或者多个客户端监听消息队列，一旦消息到达，消费者马上消费，谁先抢到算谁的，如果队列里没有消息，则消费者继续监听。   
后者也是一个或多个客户端订阅消息频道，只要发布者发布消息，所有订阅者都能收到消息，订阅者都是平等的。

**本文采用的是生产者消费者模式。**

基于Redis的消息队列实现的异步操作原理图如下：

把很多模块的某些非主业务的操作都进行异步处理。

再比如订单模块中用户下单的过程，下单成功后，发送短信等非主要业务。直接把这些操作写在一个handle里。然后先返回直接返回结果，然后再通过异步处理给用户发送下单成功的信息。



EventProducer将事件推送到消息队列中，   
EventConsumer监听队列，只要监测到有事件到达，就将事件取出，交给对应的Handler进行处理。

异步队列总结：

1. 使用redis的列表作为队列，lpush、rpop
2. EventProducer：生产者,负责把事件模型添加到队列中。如点赞操作时，在点赞操作接口生成生产者类，在点赞操作时，调用类中fireEvent（EventModel model）方法将点赞事件添加到Redis队列中。
3. EventModel：事件模型，里面包含事件的类型和事件对应的数据，序列化存储在队列中
4. EventType：事件类型，Enum，点赞、评论、邮件、登陆分别对应不同的枚举值
5. EventHandler：接口，提供handler（）、getSupportedTypes（）两个方法（）。
6. LikeHandler：继承EventHandler接口，提供处理方法和支持多种类型的事件
7. EventConsumer：消费者，设计是一个事件模型传过来，获取它的事件类型，根据事件的。

--》先将所有的Handler注册了，在消费者类初始化时Map<String, EventHandler> beans = **applicationContext**.getBeansOfType(EventHandler.**class**)找出所有Handler。

--》设置一个事件处理Map，key为事件类型，value为该事件类型需要的Handler。

--》遍历Handler，将对应的key、value放入事件处理Map中。

--》再在初始化时，开启一个线程，该线程一直从（死循环）Redis队列中取出事件模型，根据事件类型找到对应的Handler进行处理。

## 代码实现

### ****1. Redis数据库的底层操作：****

**将事件序列化后存入数据库；从数据库获取事件：**

package com.wgs.mailSender.util;

import com.alibaba.fastjson.JSON;

import com.alibaba.fastjson.JSONObject;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import org.springframework.beans.factory.InitializingBean;

import org.springframework.stereotype.Service;

import redis.clients.jedis.Jedis;

import redis.clients.jedis.JedisPool;

import java.util.List;

/\*\*

\* Created by wanggenshen\_sx on 2017/5/9.

\*/

Service

public class JedisAdapter implements InitializingBean{

private static final Logger logger = LoggerFactory.getLogger(JedisAdapter.class);

private Jedis jedis = null;

private JedisPool jedisPool = null;

/\*\*

\* 初始化

\* @throws Exception

\*/

@Override

public void afterPropertiesSet() throws Exception {

jedisPool = new JedisPool("localhost", 6379);

}

/\*\*

\* 从JedisPool获取一个Jedis连接

\* @return

\*/

private Jedis getJedis(){

try {

jedis = jedisPool.getResource();

return jedis;

}catch (Exception e){

logger.error("获取Jedis 异常 ：" + e.getMessage());

return null;

}finally {

if (jedis != null){

try {

jedis.close();

}catch (Exception e){

logger.error(e.getMessage());

}

}

}

}

/\*\*

\* 存入List集合中

\* @param key

\* @param value

\* @return

\*/

public long lpush(String key, String value){

Jedis jedis = null;

try {

jedis = jedisPool.getResource();

long result = jedis.lpush(key, value);

return result;

}catch (Exception e){

logger.error("Jedis lpush 异常 ：" + e.getMessage());

return 0;

}finally {

if (jedis != null){

try {

jedis.close();

}catch (Exception e){

logger.error(e.getMessage());

}

}

}

}

/\*\*

\* 获取指定值

\* @param timeout

\* @param key

\* @return

\*/

public List<String> brpop(int timeout, String key){

Jedis jedis = null;

try {

jedis = jedisPool.getResource();

return jedis.brpop(timeout, key);

}catch (Exception e){

logger.error("Jedis brpop 异常 ：" + e.getMessage());

return null;

}finally {

if (jedis != null){

try {

jedis.close();

}catch (Exception e){

logger.error(e.getMessage());

}

}

}

}

/\*\*

\* 给Redis中Set集合中某个key值设值

\* @param key

\* @param value

\*/

public void set(String key, String value){

Jedis jedis = null;

try {

jedis = jedisPool.getResource();

jedis.set(key, value);

}catch (Exception e){

logger.error("Jedis set 异常" + e.getMessage());

}finally {

if(jedis != null){

jedis.close();

}

}

}

/\*\*

\* 从Redis中Set集合中获取key对应value值

\* @param key

\*/

public String get(String key){

Jedis jedis = null;

try {

jedis = jedisPool.getResource();

return jedis.get(key);

}catch (Exception e){

logger.error("Jedis get 异常" + e.getMessage());

return null;

}finally {

if(jedis != null){

jedis.close();

}

}

}

/\*\*

\* 序列化

\* @param key

\* @param object

\*/

public void setObject(String key, Object object){

set(key, JSONObject.toJSONString(object));

}

public <T>T getObject(String key, Class<T> clazz){

String value = get(key);

if (value != null){

return JSON.parseObject(value, clazz);

}

return null;

}

}

**存入Redis中时要存到key对应的集合中，所以要写个产生key的工具类：**   
**RedisKeyUtil.java：**

package com.wgs.mailSender.util;

/\*\*

\* Created by wanggenshen\_sx on 2017/5/9.

\*/

public class RedisKeyUtil {

private static String EVENT\_KEY = "ASYNC\_EVENT\_KEY";

public static String getEventQueueKey(){

return EVENT\_KEY;

}

}

### ****2. 定义事件的类型****

/\*\*

\* Created by wanggenshen\_sx on 2017/5/9.

\*/

public enum EventType {

CAlCULATE(0),

COPYFILE(1),

MAIL(2);

private int value;

public int getValue(){

return value;

}

EventType(int value){

this.value = value;

}

}

### ****3.定义事件模型****

package com.wgs.mailSender.async;

import java.util.HashMap;

import java.util.Map;

/\*\*

\* Created by wanggenshen\_sx on 2017/5/9.

\*/

public class EventModel {

private EventType eventType;

private int actorId;

private int entityId;

private int entityType;

private int entityOwnerId;

Map<String, String> exts = new HashMap<>();

public EventModel(EventType eventType){

this.eventType = eventType;

}

public String getExt(String key) {

return exts.get(key);

}

public EventModel setExt(String key, String value) {

exts.put(key, value);

return this;

}

public EventModel(){

}

public EventType getEventType() {

return eventType;

}

public void setEventType(EventType eventType) {

this.eventType = eventType;

}

public int getActorId() {

return actorId;

}

public EventModel setActorId(int actorId) {

this.actorId = actorId;

return this;

}

public int getEntityId() {

return entityId;

}

public EventModel setEntityId(int entityId) {

this.entityId = entityId;

return this;

}

public int getEntityType() {

return entityType;

}

public EventModel setEntityType(int entityType) {

this.entityType = entityType;

return this;

}

public int getEntityOwnerId() {

return entityOwnerId;

}

public EventModel setEntityOwnerId(int entityOwnerId) {

this.entityOwnerId = entityOwnerId;

return this;

}

public Map<String, String> getExts() {

return exts;

}

public void setExts(Map<String, String> exts) {

this.exts = exts;

}

}

### ****4.EventProducer.java: 将事件发送到工作队列中****

package com.wgs.mailSender.async;

import com.alibaba.fastjson.JSONObject;

import com.wgs.mailSender.util.JedisAdapter;

import com.wgs.mailSender.util.RedisKeyUtil;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Service;

/\*\*

\* Created by wanggenshen\_sx on 2017/5/9.

\*/

Service

public class EventProducer {

private static final Logger logger = LoggerFactory.getLogger(EventProducer.class);

@Autowired

JedisAdapter jedisAdapter;

public boolean fireEvent(EventModel eventModel){

try {

//序列化

String json = JSONObject.toJSONString(eventModel);

//产生key

String key = RedisKeyUtil.getEventQueueKey();

//放入工作队列中

jedisAdapter.lpush(key, json);

return true;

}catch (Exception e){

logger.error("EventProducer fireEvent 异常 ：" + e.getMessage());

return false;

}

}

}

### ****5. EventConsumer.java : 从工作队列中取出事件进行处理****

package com.wgs.mailSender.async;

import com.alibaba.fastjson.JSON;

import com.alibaba.fastjson.JSONObject;

import com.wgs.mailSender.util.JedisAdapter;

import com.wgs.mailSender.util.RedisKeyUtil;

import com.wgs.mailSender.util.ThreadTaskUtil;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import org.springframework.beans.BeansException;

import org.springframework.beans.factory.InitializingBean;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.context.ApplicationContext;

import org.springframework.context.ApplicationContextAware;

import org.springframework.stereotype.Service;

import java.util.ArrayList;

import java.util.HashMap;

import java.util.List;

import java.util.Map;

/\*\*

\* Created by wanggenshen\_sx on 2017/5/9.

\*/

Service

public class EventConsumer implements InitializingBean, ApplicationContextAware{

private static final Logger logger = LoggerFactory.getLogger(EventConsumer.class);

@Autowired

JedisAdapter jedisAdapter;

private ApplicationContext applicationContext;

//存储各种type事件对应的Handler处理类

private Map<EventType, List<EventHandler>> config = new HashMap<>();

@Override

public void afterPropertiesSet() throws Exception {

//获取所有实现EventHandler的类

Map<String, EventHandler> eventHandlerMap = applicationContext.getBeansOfType(EventHandler.class);

if (eventHandlerMap != null){

for (Map.Entry<String, EventHandler> entry : eventHandlerMap.entrySet()){

//遍历每个EventHandler，将这个EventHandler放到集合对应的type中

EventHandler eventHandler = entry.getValue();

//获取每个EventHandler所关注的事件类型

List<EventType> eventTypes = eventHandler.getSupportEventType();

for (EventType type : eventTypes){

//初始化的时候config为空

if (!config.containsKey(type)){

config.put(type, new ArrayList<EventHandler>());

}

//将Handler放入指定type中

config.get(type).add(eventHandler);

}

}

}

//启动线程从工作队列中取出事件进行处理

Thread thread = new Thread(new Runnable() {

@Override

public void run() {

while (true){

String key = RedisKeyUtil.getEventQueueKey();

//获取存储的(经过序列化的)事件event

List<String> events = jedisAdapter.brpop(0, key);

for (String jsonEvent : events){

if (jsonEvent.equals(key)){

continue;

}

EventModel eventModel = JSON.parseObject(jsonEvent, EventModel.class);

if (!config.containsKey(eventModel.getEventType())){

logger.error("不能识别的事件！");

continue;

}

//获取关注过该事件eventModel的handler,进行处理

for (EventHandler handler : config.get(eventModel.getEventType())){

handler.doHandler(eventModel);

}

}

}

}

});

thread.start();

}

@Override

public void setApplicationContext(ApplicationContext applicationContext) throws BeansException {

this.applicationContext = applicationContext;

}

}

### ****6. EventHandler.java: 处理事件的接口****

package com.wgs.mailSender.async;

import java.util.List;

/\*\*

\* Created by wanggenshen\_sx on 2017/5/9.

\*/

public interface EventHandler {

//处理事件

void doHandler(EventModel eventModel);

//获取关注事件的类型

List<EventType> getSupportEventType();

}

### ****7. FileCopyHandler.java : 事件处理的具体实现类：****

package com.wgs.mailSender.async.handler;

import com.wgs.mailSender.async.EventHandler;

import com.wgs.mailSender.async.EventModel;

import com.wgs.mailSender.async.EventType;

import com.wgs.mailSender.util.FileCopyUtil;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import org.springframework.stereotype.Component;

import java.util.Arrays;

import java.util.List;

/\*\*

\* Created by wanggenshen\_sx on 2017/5/10.

\*/

Component

public class FileCopyHandler implements EventHandler {

private static final Logger logger = LoggerFactory.getLogger(CalculateHandler.class);

private static String source = "D:/SQL.zip";

private static String target = "D:/copy/sql1.zip";

@Override

public void doHandler(EventModel eventModel) {

long start1 = System.currentTimeMillis();

//模拟从工作队列中取出的一个事件进行处理

FileCopyUtil.copyFile(source, target);

long end1 = System.currentTimeMillis();

System.out.println("非业务运行完成，运行时间为\*\*：" + (end1 - start1));

}

@Override

public List<EventType> getSupportEventType() {

return Arrays.asList(EventType.COPYFILE);

}

}

## 测试

写了一个简单接口，对其进行测试：

package com.wgs.mailSender.controller;

import com.wgs.mailSender.async.EventModel;

import com.wgs.mailSender.async.EventProducer;

import com.wgs.mailSender.async.EventType;

import com.wgs.mailSender.util.FileCopyUtil;

import com.wgs.mailSender.util.JSONUtil;

import com.wgs.mailSender.util.TaskUtil;

import com.wgs.mailSender.util.ThreadTaskUtil;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Controller;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RequestMethod;

import org.springframework.web.bind.annotation.ResponseBody;

/\*\*

\* Created by wanggenshen\_sx on 2017/5/9.

\*/

@Controller

public class EventController {

private static final Logger logger = LoggerFactory.getLogger(EventController.class);

//主业务

private static String source = "D:/HEXO.zip";

private static String target = "D:/copy/1.zip";

private static String target2 = "D:/copy/2.zip";

//非主业务

private static String sourceGame = "D:/SQL.zip";

private static String targetGame = "D:/copy/SQL1.zip";

@Autowired

EventProducer eventProducer;

@RequestMapping(path = {"/event/async"}, method = {RequestMethod.POST})

@ResponseBody

public String eventAsync(){

try{

long start = System.currentTimeMillis();

//模拟主业务1 ： 复制文件HEXO.zip到指定文件夹target中

long time1 = FileCopyUtil.copyFile(source, target);

System.out.println("业务1运行时间为： " + time1);

//模拟非主业务: 这个业务可能不是那么很紧急要立刻实现的

System.out.println("非业务程序开始运行...");

long t = FileCopyUtil.copyFile(sourceGame, targetGame);

System.out.println("非业务程序运行结束，运行时间为：" + t);

//模拟主业务2 ： 复制文件HEXO.zip到指定文件夹target2中

long time2 = FileCopyUtil.copyFile(source, target);

System.out.println("业务2运行时间为： " + time2);

long end = System.currentTimeMillis();

System.out.println("当前运行总时间为：" + (end - start));

return JSONUtil.getJSONString(0);

}catch (Exception e){

logger.error("eventAsync 异常:" + e.getMessage());

return JSONUtil.getJSONString(1, "failed");

}

}

}

**不使用队列处理处理：**   
主业务1，2是复制一个较小的文件；非主业务是复制一个大小为2G的文件，所以复制时间会比较长。   
运行结果： 

可以看出，主业务1执行结束以后，开始执行非主业务的操作，由于操作过程时间较长，等了很久直到其复制完毕才去执行主业务2。如果这个过程中非主业务在复制过程出错的话，就会导致整个程序抛出异常，无法执行下去，这可是个致命的问题。

试想下：如果你在一个网站进行注册操作的时候，注册完成会发送一封邮件到你的邮箱。如果这个邮件发送过程时间很长，那么你在注册完成后需要等很久才能进入主页面，这会严重影响用户的体验。

**使用Redis队列处理：**   
如果交给Redis实现的工作队列去处理，在主业务1操作结束后，非主业务开始执行，但非主业务其实并没真正执行而是将这个事件发送到工作队列中，EventConsumer会时刻监听队列，一旦有事件到达立刻取出交由对应的Handler类去处理。而主业务2是在主业务1执行完后就去执行，不用等待这个非主业务执行完毕，也就减少等待时间。

将代码中的非主业务的操作改为交由工作队列去处理：

//模拟非主业务: 这个业务可能不是那么很紧急要立刻实现的

System.out.println("非主业务程序开始运行...");

eventProducer.fireEvent(new EventModel(EventType.COPYFILE));

System.out.println("非主业务程序运行结束");

运行结果：   
   
可以看到，主业务1执行结束后，非主业务开始执行，但是并没有真正去立刻执行，而是将事件发送到工作队列中。等待主业务2执行结束后，等了很久非主业务才真正执行结束，这样就实现 了解耦的功能。