**netty**

* 1. **NIO服务端开发流程**

1.创建serverSocketChannel,配置为非阻塞；

2，绑定监听，配置TCP参数；

3，创建一个独立的io线程，用于轮询多路复用器selector；

4，将之前创建的serverSocketChannel注册到selector上，监听SelectionKey.ACCEPT;

5,启动io线程，再循环中执行selector.select()方法，轮询就绪的Channel;

6,轮询到了就绪状态的Channel的时候，对其进行判断，如果是OP\_ACCEPT说明是新客户端接入，调用ServerSocketChannel.accept()方法接受新客户端；

7，设置新客户端链路SocketChannel为非阻塞模式，并注册到selector，监听OP\_READ;

8,轮询就绪的Channel为OP\_READ,这说明SocketChannel中有就绪的数据包需要读取，则构造ByteBuffer来读取数据；

9，轮询就绪的Channel为OP\_WRITE,说明还有数据没有发送完成，需要继续发送数据；

* 1. **netty服务端开发流程**

1.创建两个NioEventLoopGroup线程组，一个用于服务端接受客户端的连接，一个用于进行SocketChannel的网络读写；

2，创建启动NIO服务的辅助启动类ServerBootstrap；

3，将创建的两个NioEventLoopGroup线程组作为参数传到ServerBootstrap；

4，创建Channel并配置TCP参数；

5，绑定io事件处理类ChildChannelHander；

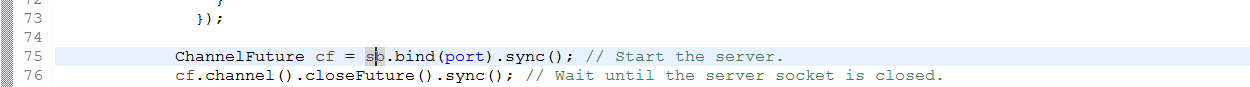
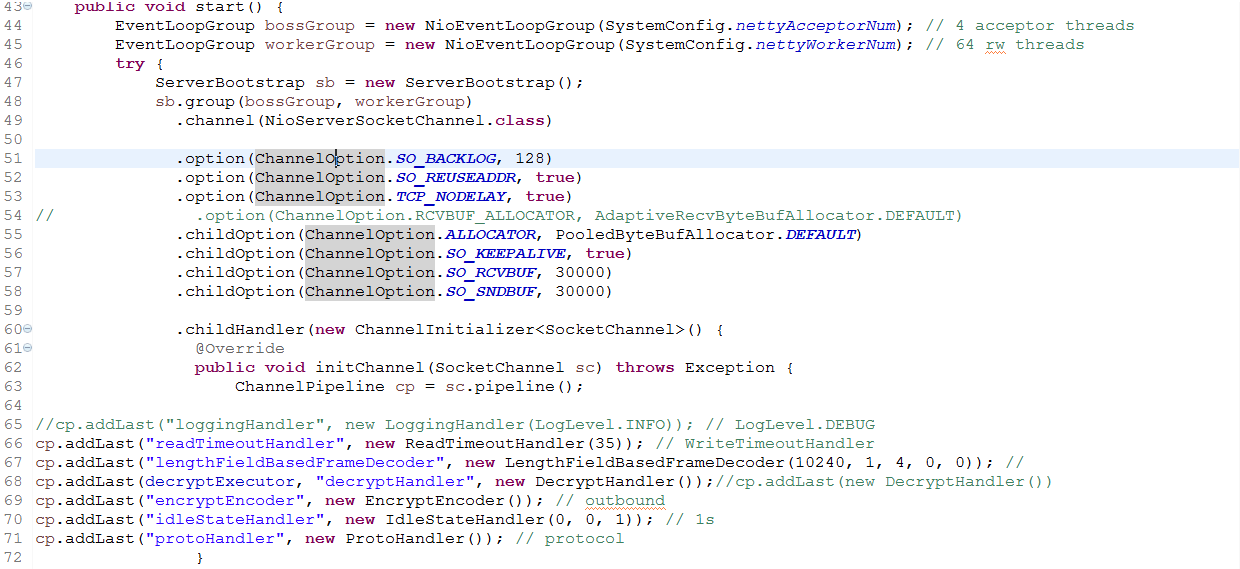
6，所有配置完成后，bind方法来绑定监听端口，调用同步阻塞方法sync();

注意：ChildChannelHander继承ChannelHandlerAdapter，当有数据读取的时候调用channelRead()方法；调用ByteBuf的readableBytes方法可以获取到缓冲区可读的数据；

当数据读取完成调用channelReadComplete()方法，使用ChannelHandlerContext.flush()方法将发送缓冲区中的消息写入到SocketChannel;

当出现报错时调用exceptionCaught()方法；

* 1. **代码示例**



**netty客户端开发流程同服务端**，只是配置完启动辅助类的配置之后，调用connect方法发起异步连接，然后调用同步方法sync()等待连接成功。

**kafka**

**2.1 kafka介绍**

Kafka是一个分布式的流平台，

kafka作为一个集群运行在一个或多个服务器上，

kafka集群存储的消息是以topic为类别记录的。

每个消息（也叫记录record）是由一个key，一个value和时间戳构成。

**Topic**

Kafka将消息分门别类，每一类的消息称之为一个主题(Topic).

**Producer**

发布消息的对象称之为主题生产者(Kafka topic producer)

**Consumer**

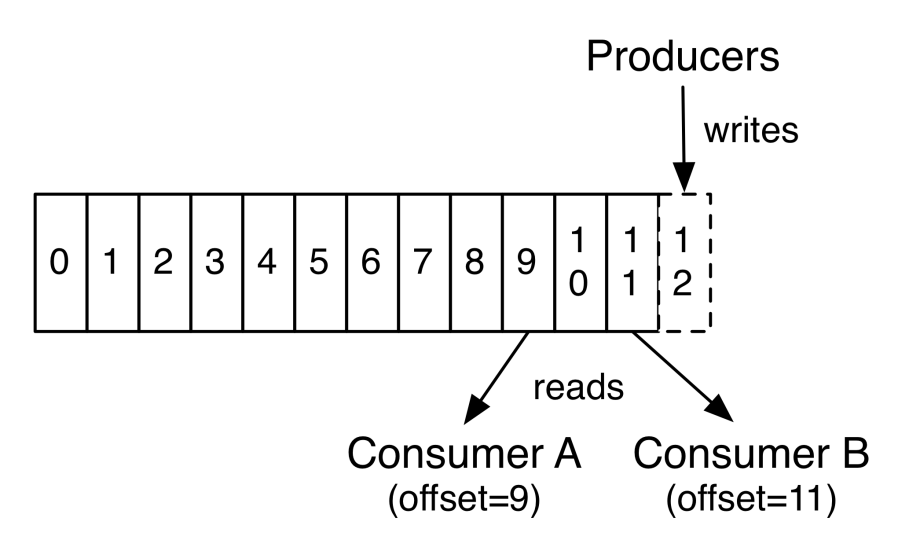
订阅消息并处理发布的消息的对象称之为主题消费者(consumers)

**Broker**

已发布的消息保存在一组服务器中，称之为Kafka集群。集群中的每一个服务器都是一个代理(Broker). 消费者可以订阅一个或多个主题（topic），并从Broker拉数据，从而消费这些已发布的消息。

**2.2 Topic图解**





**2.3 Kafka架构图解**



**2.4 kafka作为一个消息系统与传统企业消息系统（rabbitmq）相比**

传统的消息有两种模式：队列和发布订阅。 在队列模式中，消费者池从服务器读取消息（每个消息只被其中一个读取）; 发布订阅模式：消息广播给所有的消费者。这两种模式都有优缺点，队列的优点是允许多个消费者瓜分处理数据，这样可以扩展处理。但是，队列不像多个订阅者，一旦消息者进程读取后故障了，那么消息就丢了。而发布和订阅允许你广播数据到多个消费者，由于每个订阅者都订阅了消息，所以没办法缩放处理。

kafka中消费者组有两个概念：队列：消费者组（consumer group）允许同名的消费者组成员瓜分处理。发布订阅：允许你广播消息给多个消费者组（不同名）。

**2.5 kafka作为一个存储系统:**

Kafka比别的系统的优势是它是一个非常高性能的存储系统。

**2.6 kafka的流处理:**  
在kafka中，流处理持续获取输入topic的数据，进行处理加工，然后写入输出topic。  
可以直接使用producer和consumer API进行简单的处理。对于复杂的转换，Kafka提供了更强大的Streams API。可构建聚合计算或连接流到一起的复杂应用程序。

**消息传递，存储和流处理的组合,对于Kafka作为流式处理平台的作用至关重要**

像HDFS这样的分布式文件系统允许存储静态文件来进行批处理。这样系统可以有效地存储和处理来自过去的历史数据。

传统企业的消息系统允许在你订阅之后处理未来的消息：在未来数据到达时处理它。

Kafka结合了这两种能力，这种组合对于kafka作为流处理应用和流数据管道平台是至关重要的。

**2.7 Kafka命令**

启动Kafka服务

kafka-server-start.bat .\config\server.properties

创建主题

kafka-topics.bat --create --zookeeper localhost:2181 --replication-factor 1 --partitions 1 --topic wangwb

创建生产者

kafka-console-producer.bat --broker-list localhost:9092 --topic wangwb

创建consumer

kafka-console-consumer.bat --zookeeper localhost:2181 --topic wangwb

列出主题

kafka-topics.bat -list -zookeeper localhost:2181

描述主题

kafka-topics.bat -describe -zookeeper localhost:2181 -topic wangwb

从头读取消息

kafka-console-consumer.bat -zookeeper localhost:2181 -topic wangwb -from-beginning

删除主题

kafka-run-class.bat kafka.admin.TopicCommand -delete -topic wangwb -zookeeper localhost:2181

查看topic的详细信息

kafka-topics.bat -zookeeper localhost:2181 -describe -topic wangwb

为topic增加partition

kafka-topics.bat -zookeeper localhost:2181 -alter -partitions 20 -topic wangwb

**2.8 KafkaProducer代码解释**

package kafkaClient;

import java.util.Properties;

import org.apache.kafka.clients.producer.KafkaProducer;

import org.apache.kafka.clients.producer.Producer;

import org.apache.kafka.clients.producer.ProducerRecord;

public class SimpleProducer {

public static void main(String[] args) {

System.out.println("生产者");

Properties props = new Properties();

props.put("bootstrap.servers", "127.0.0.1:9092");

props.put("acks", "0");//

props.put("retries", 0);

props.put("batch.size", 16384);

props.put("linger.ms", 1);

props.put("buffer.memory", 33554432);

props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");

props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");

Producer<String, String> producer = new KafkaProducer<>(props);

for (int i = 0; i < 100000; i++){

System.out.println("生产 "+i);

producer.send(new ProducerRecord<String, String>("wangwb", "101", "202"));

// producer.send(new ProducerRecord<String, String>("wangwb", Integer.toString(i)+"00000", Integer.toString(i)+"00000"));

}

producer.close();

}

}

**bootstrap.servers**,为服务器的ip,port；

**acks**是判别请求是否为完整的条件（就是是判断是不是成功发送了）;

**retries**，如果请求失败，生产者会自动重试，我们指定是0次，如果启用重试，则会有重复消息的可能性。

**batch.size**,producer(生产者)缓存每个分区未发送消息。缓存的大小是通过 batch.size 配置指定的

**linger.ms**，等于0缓冲可立即发送，即便缓冲空间还没有满，大于0则等待N ms之后发送;

**buffer.memory** ，控制生产者可用的缓存总量，如果消息发送速度比其传输到服务器的快，将会耗尽这个缓存空间。当缓存空间耗尽，其他发送调用将被阻塞，阻塞时间的阈值通过max.block.ms设定，之后它将抛出一个TimeoutException。  
 **key.serializer**和**value.serializer**示例，将用户提供的key和value对象ProducerRecord转换成字节

**RabbitMQ**

**RabbitMQ**是一个消息中间件，它的工作就是发送和接收消息。主要使用场景有：异步处理、应用解耦、流量消峰及消息通讯等。RabbitMQ采用Erlang语言，基于AMQP(Advanced Message Queue Protocol 高级消息队列协议)的开源实现。

AMQP是一个提供统一消息服务的应用层标准协议，基于此协议的客户端与消息中间件可传递消息，不受开发语言、MQ产品的限制。

**3.1 基本概念**

**Producer:**发送消息至消息队列。

**Consumer:**从消息队列接收消息。

**Queue：**一个先进先出FIFO的消息存储区域。消息按照顺序发送、接收，一旦消息被消费处理，该消息将从队列中删除。

**Topic：**支持消息多个订阅者的机制。

**Connection:**连接，位于客户端和Broker之间的TCP连接

**Channel:**信道，仅仅创建客户端和Broker之间的连接后，客户端还不能发送消息。需要为每一个Connection创建Channel，AMQP协议只有通过 Channel才能执行AMQP命令。一个Connection可以包含多个Channel。

**Exchange:**接收生产者发送的消息，并根据Binding规则将消息路由给服务器中的队列。Exchange类型有direct、Fanout和Topic、headers，不同类型的Exchange路由的行为不同。其中发布订阅模型设定的Exchange Type即为Topic，点对点的队列Queue的Exchange Type为direct。

**Broker：**MQ节点，接受客户端连接，实现AMQP消息队列和路由功能的进程。

**Binding:**联系Exchange与Message Queue。Exchange在与多个Message Queue发生Binding后会生成一张路由表，路由表中存储着Message Queue所需消息的限制条件即RoutingKey。当Exchange收到Message时会解析其Header得到Routing Key，Exchange根据Routing Key与Exchange Type将Message路由到Message Queue。Binding Key由Consumer在绑定Exchange与Message Queue时指定，而Routing Key由Producer发送Message时指定，两者的匹配方式由Exchange Type决定。

**消息确认：**为了保证消息不丢失，RabbitMQ提供了消息确认Acknowledge机制，即ACK机制，RabbitMQ提供两种确认机制：自动确认和手工确认。

**消息拒绝:**在消费端接收到消息后，如果想明确拒绝当前的消息而不是确认，通过basicReject(long deliverTag,bool requeue) 方法显式的拒绝该消息执行。如果requeue参数设置为false，则RabbitMQ会把消息从队列中移除；如果requeue参数设置为true，则会重新将这条消息存入队列。

**3.2 操作命令**

**rabbitmq-service install** ：安装服务

**rabbitmq-service remove**：卸载服务

**rabbitmq-service.bat start**：启动服务

**rabbitmqctl status:**查看服务器状态

**rabbitmqctl list\_queues:**查看队列信息

**rabbitmqctl list\_exchanges:**查看exchange信息

**rabbitmqctl list\_bindings:**查看binding信息

**rabbitmqctl list\_connections:**查看连接信息

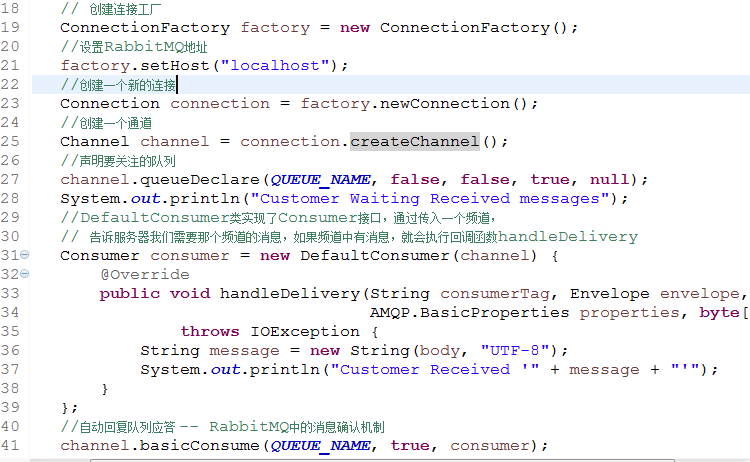
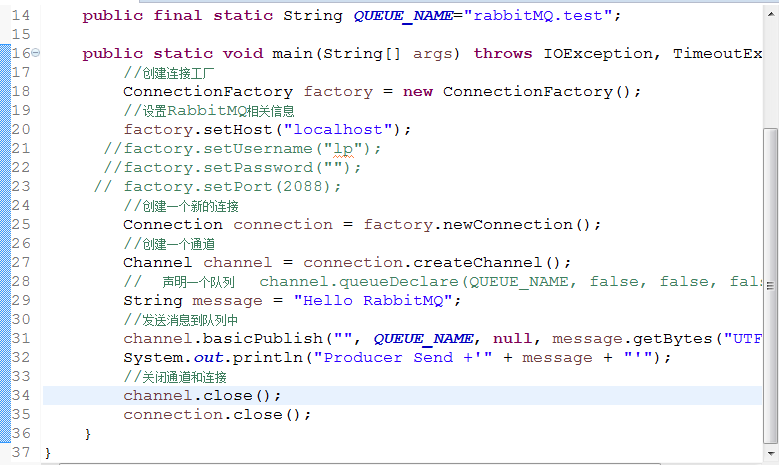
**rabbitmqctl list\_channels** :查看channel信息

**rabbitmqctl add\_user userName password**

**rabbitmqctl change\_password userName newPassword**

**rabbitMQctl join\_cluster rabbitMQ\_Node\_name:**将某个节点添加至集群

**rabbitmqctl forget\_cluster rabbitMQ\_Node\_Name:**将某个节点从集群中移除

**3.3代码示例**

**Redis**

**Redis**是一个key-value存储系统。它支持存储的value类型相对更多，包括string(字符串)、list(链表)、set(集合)、zset(sorted set --有序集合)和hash（哈希类型）。这些数据类型都支持push/pop、add/remove及取交集并集和差集及更丰富的操作，而且这些操作都是原子性的。在此基础上，redis支持各种不同方式的排序。为了保证效率，数据都是缓存在内存中。redis会周期性的把更新的数据写入磁盘或者把修改操作写入追加的记录文件，并且在此基础上实现了master-slave。

**4.1 Redis在windows环境搭建**

采用[redis sentinel（哨兵机制）部署(Windows下实现)](http://www.cnblogs.com/ruiati/p/6374152.html)

采用一主(master)二从(slave)三sentinel的[架构](http://lib.csdn.net/base/architecture" \o "大型网站架构知识库" \t "https://www.cnblogs.com/ruiati/p/_blank)模式来部署

参照资料：<https://www.cnblogs.com/ruiati/p/6374152.html>

启动redis主从服务：

redis-server.exe redis.conf

redis-server.exe redis6380.conf

redis-server.exe redis6381.conf

启动redis哨兵服务：

redis-server.exe sentinel26379.conf --sentinel

redis-server.exe sentinel26479.conf --sentinel

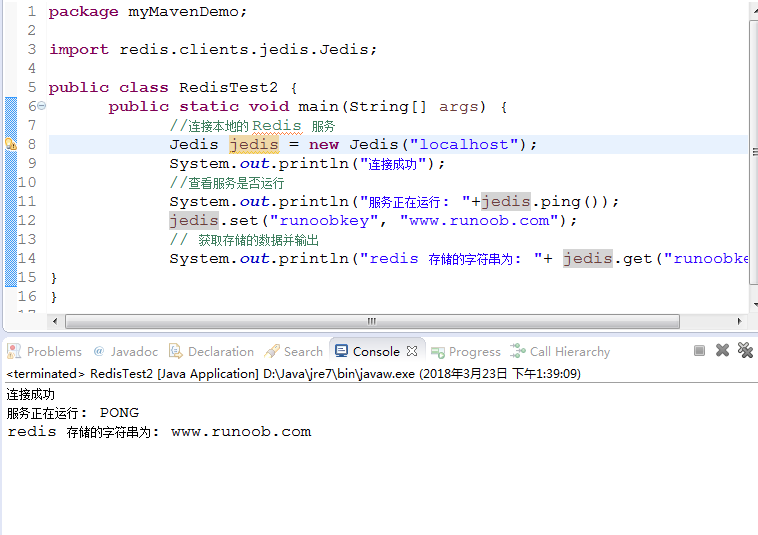
redis-server.exe sentinel26579.conf --sentinel

进入Redis哨兵客户端：

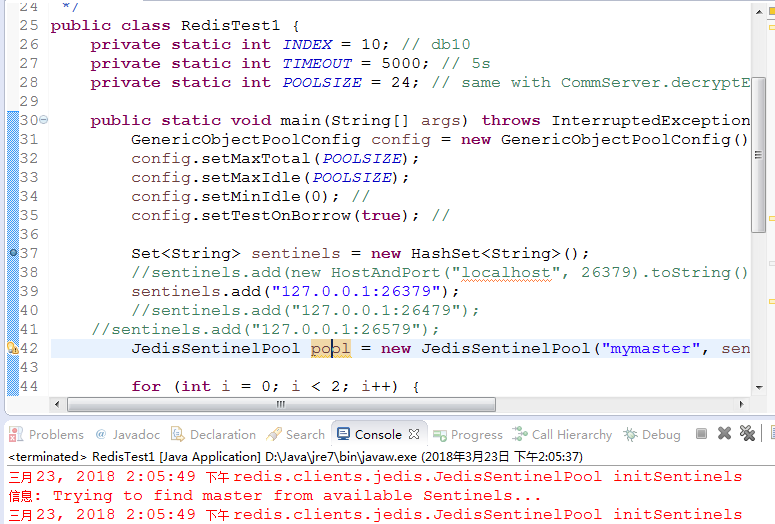
redis-cli -h 127.0.0.1 -p 26379

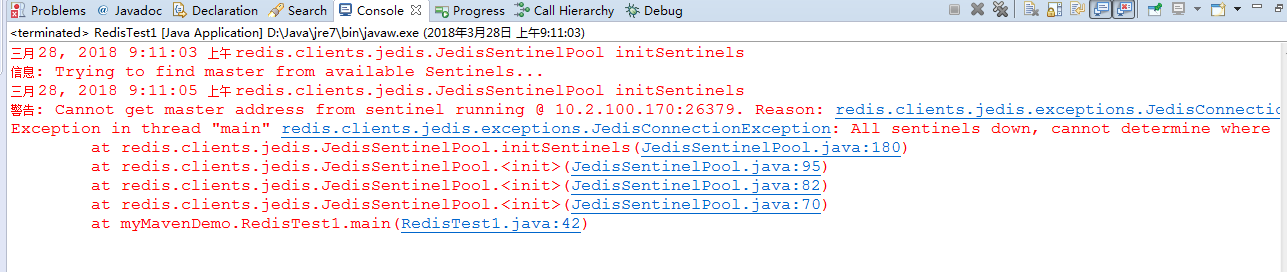
redis-cli -h 127.0.0.1 -p 26479

**4.2 redis基本连接**

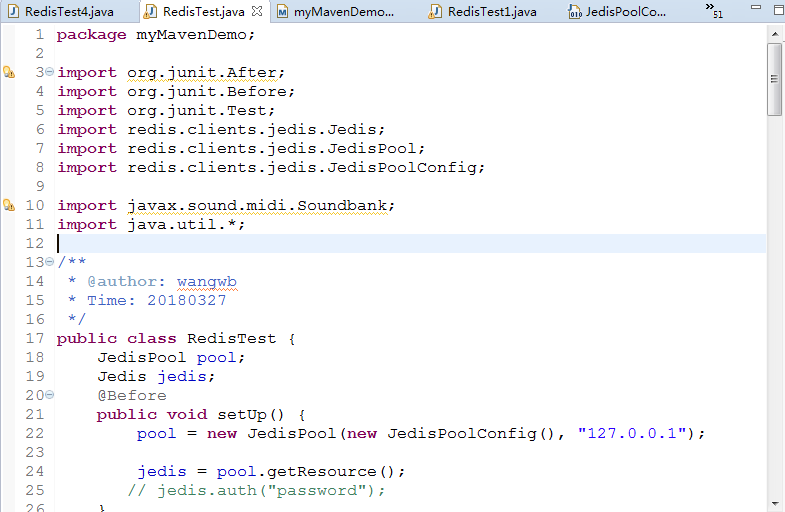


**4.3 高可用redis哨兵线程池连接**



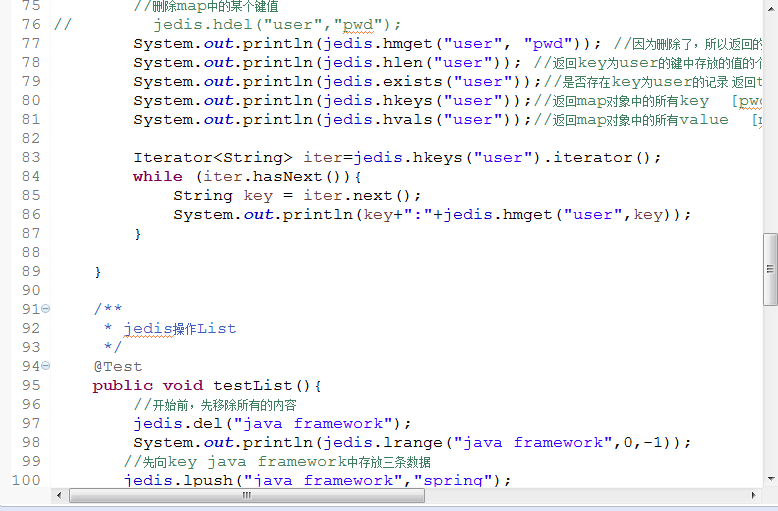


**4.4 Redis操作各种value类型的代码解析**

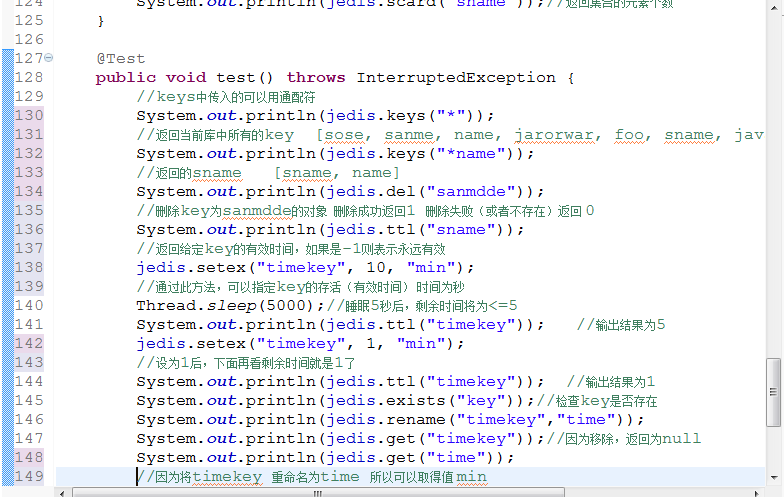


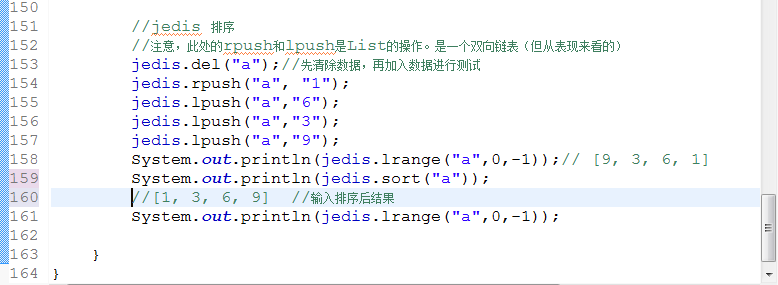


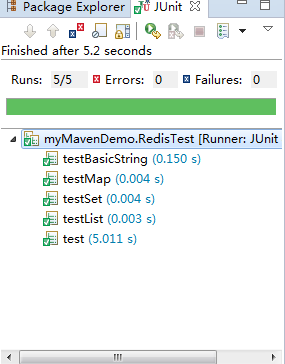












**4.5 补充：Redis开发规范**

****1.冷热数据分离，不要将所有数据全部都放到Redis中****

虽然Redis支持持久化，但是Redis的数据存储全部都是在内存中的，成本昂贵。建议根据业务只将高频热数据存储到Redis中【QPS大于5000】，对于低频冷数据可以使用MySQL/ElasticSearch/MongoDB等基于磁盘的存储方式，不仅节省内存成本，而且数据量小在操作时速度更快、效率更高！  
****2.不同的业务数据要分开存储****  
不要将不相关的业务数据都放到一个Redis实例中，建议新业务申请新的单独实例。因为Redis为单线程处理，独立存储会减少不同业务相互操作的影响，提高请求响应速度；同时也避免单个实例内存数据量膨胀过大，在出现异常情况时可以更快恢复服务！  
****3.存储的Key一定要设置超时时间****  
如果应用将Redis定位为缓存Cache使用，对于存放的Key一定要设置超时时间！因为若不设置，这些Key会一直占用内存不释放，造成极大的浪费，而且随着时间的推移会导致内存占用越来越大，直到达到服务器内存上限！另外Key的超时长短要根据业务综合评估，而不是越长越好！  
****4.对于必须要存储的大文本数据一定要压缩后存储****  
对于大文本【超过500字节】写入到Redis时，一定要压缩后存储！大文本数据存入Redis，除了带来极大的内存占用外，在访问量高时，很容易就会将网卡流量占满，进而造成整个服务器上的所有服务不可用，并引发雪崩效应，造成各个系统瘫痪！  
****5.线上Redis禁止使用Keys正则匹配操作****  
Redis是单线程处理，在线上KEY数量较多时，操作效率极低【时间复杂度为O(N)】，该命令一旦执行会严重阻塞线上其它命令的正常请求，而且在高QPS情况下会直接造成Redis服务崩溃！如果有类似需求，请使用scan命令代替！  
****6.可靠的消息队列服务****  
Redis List经常被用于消息队列服务。假设消费者程序在从队列中取出消息后立刻崩溃，但由于该消息已经被取出且没有被正常处理，那么可以认为该消息已经丢失，由此可能会导致业务数据丢失，或业务状态不一致等现象发生。为了避免这种情况，Redis提供了RPOPLPUSH命令，消费者程序会原子性的从主消息队列中取出消息并将其插入到备份队列中，直到消费者程序完成正常的处理逻辑后再将该消息从备份队列中删除。同时还可以提供一个守护进程，当发现备份队列中的消息过期时，可以重新将其再放回到主消息队列中，以便其它的消费者程序继续处理。  
****7.谨慎全量操作Hash、Set等集合结构****  
在使用HASH结构存储对象属性时，开始只有有限的十几个field，往往使用HGETALL获取所有成员，效率也很高，但是随着业务发展，会将field扩张到上百个甚至几百个，此时还使用HGETALL会出现效率急剧下降、网卡频繁打满等问题【时间复杂度O(N)】,此时建议根据业务拆分为多个Hash结构；或者如果大部分都是获取所有属性的操作,可以将所有属性序列化为一个STRING类型存储！同样在使用SMEMBERS操作SET结构类型时也是相同的情况！  
****8.根据业务场景合理使用不同的数据结构类型****  
目前Redis支持的数据库结构类型较多：字符串（String），哈希（Hash），列表（List），集合（Set），有序集合（Sorted Set）, Bitmap, HyperLogLog和地理空间索引（geospatial）等,需要根据业务场景选择合适的类型，常见的如：String可以用作普通的K-V、计数类；Hash可以用作对象如商品、经纪人等，包含较多属性的信息；List可以用作消息队列、粉丝/关注列表等；Set可以用于推荐；Sorted Set可以用于排行榜等！  
****9.命名规范****  
redis的key命名尽量简单明确，容易阅读理解，如：系统名+业务名+业务数据+其他  
****10.线上禁止使用monitor命令****  
禁止生产环境使用monitor命令，monitor命令在高并发条件下，会存在内存暴增和影响Redis性能的隐患

****11.禁止大string****

核心集群禁用1mb的string大key(虽然redis支持512MB大小的string)，如果1mb的key每秒重复写入10次，就会导致写入网络IO达10MB;

****12.redis容量****  
单实例的内存大小不建议过大，建议在10~20GB以内。

redis实例包含的键个数建议控制在1kw内，单实例键个数过大，可能导致过期键的回收不及时。

**Flink**

**5.1两种数据集类型**

①无穷数据集：无穷的持续集成的数据集合。

②有界数据集：有限不会改变的数据集合。

**5.2两种数据运算模型**

①流式：只要数据一直在产生，计算就持续地进行

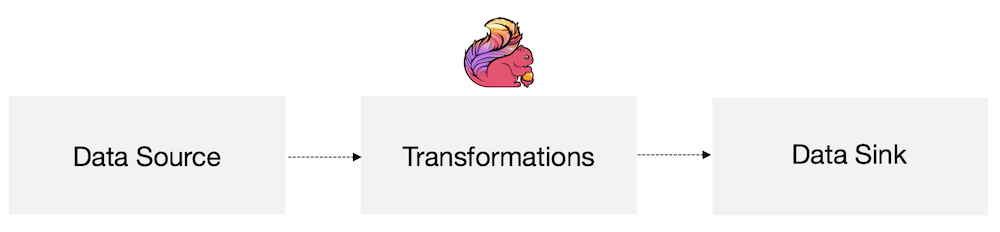
②批处理：在预先定义的时间内运行计算，当完成时释放计算机资源

**5.3 Flink程序组成**

数据源：Flink处理的输入的数据。

转化：Flink对数据进行处理的步骤。

接收器：Flink将处理之后的数据发送的地点



**Redis**

**6.1 jedis操作string**

//向key-->name中放入了value-->wangwb

jedis.set("name","wangwb");

//获取key-->name的value

jedis.get("name");

//将abc append到已经有的value之后

jedis.append("name","abc");

jedis.get("name");//执行结果:wangwbabc

//覆盖原来的数据

jedis.set("name","wangwb\_1");

jedis.get("name");//执行结果:wangwb\_1

//删除key对应的记录

jedis.del("name");

jedis.get("name");//执行结果：null

\* jedis.set("name","wangwb");

\* jedis.set("home","qingdao");

jedis.mset("name","wangwb","home","qingdao");

jedis.mget("name","jarorwar");执行结果:[wangwb, qingdao]

**6.2 jedis操作Map**

Map<String,String> user=**new** HashMap<String,String>();

user.put("name","wangwb");

user.put("pwd","password");

//向key-->user中放入了value-->(map)user

jedis.hmset("user",user);

//第一个参数是存入redis中对象的key，后面是放入map中的key

List<String> rsmap = jedis.hmget("user", "name");//结果是一个泛型的List[wangwb, password]

//删除map中的某个键值

jedis.hdel("user","pwd");

//返回key为user的键中存放的值的个数

jedis.hlen("user");

//是否存在key为user的记录 返回true

jedis.exists("user");

//返回map对象中的所有key [name, pwd]

jedis.hkeys("user");

//返回map对象中的所有value [wangwb, password]

jedis.hvals("user");

**6.3 jedis操作List**

//开始前，先移除所有的内容

jedis.del("java framework");

//先向key java framework中存放三条数据

jedis.lpush("java framework","spring");

jedis.lpush("java framework","struts");

jedis.lpush("java framework","hibernate");

//第一个是key，第二个是起始位置，第三个是结束位置，jedis.llen获取长度 -1表示取得所有

jedis.lrange("java framework",0,-1);//跟下面结果一样

jedis.lrange("java framework",0,jedis.llen("java framework"));//[hibernate, struts, spring]

**6.4 jedis操作Set**

//添加

jedis.sadd("sname","minxr");

jedis.sadd("sname","jarorwar");

jedis.sadd("sname","wangwb");

jedis.sadd("sanme","noname");

//移除noname

jedis.srem("sname","noname");

//获取所有加入的value

jedis.smembers("sname");

//判断 minxr 是否是sname集合的元素

jedis.sismember("sname", "minxr");

//返回集合中的一个随机元素

jedis.srandmember("sname");

//返回集合的元素个数

jedis.scard("sname");

**6.5补充**

//keys中传入的可以用通配符

jedis.keys("\*");

jedis.keys("\*name");

//删除key为sanmdde的对象 删除成功返回1 删除失败返回 0

jedis.del("sanmdde");

//返回给定key的有效时间，如果是-1则表示永远有效

jedis.ttl("sname");

//通过此方法，可以指定key的存活（有效时间） 时间为秒

jedis.setex("timekey", 10, "min");

//检查key1是否存在

jedis.exists("key1");

//重命名key

jedis.rename("timekey","time");

//排序

jedis.sort("a");

**WATCH**命令可以监控一个或多个键，一旦其中有一个键被修改（或删除），之后的事务就不会执行。监控一直持续到EXEC命令（事务中的命令是在EXEC之后才执行的，所以在MULTI命令后可以修改WATCH监控的键值）。

WATCH mykey

val = GET mykey

val = val + 1

MULTI

SET mykey $val

EXEC

**非pipeline**客户端会请求，然后获取结果，一般情况下这是 一个阻塞的过程，会等待服务器执行并返回结果。

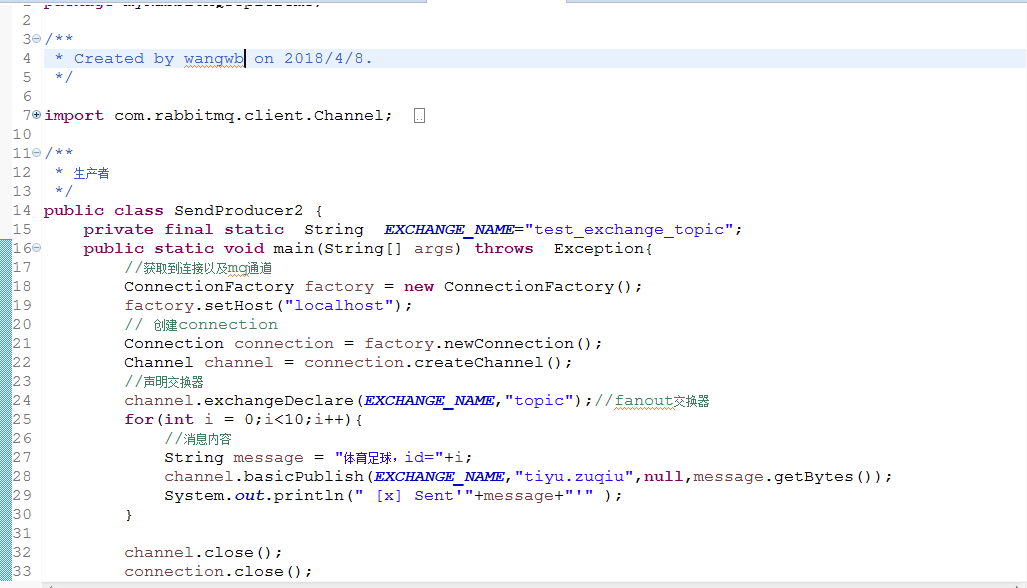
**pipeline**不用等待请求结果，一直发送命令，最后一次性获取。

有些系统可能对可靠性要求很高，每次操作都需要立马知道这次操作是否成功，是否数据已经写进redis了，那这种场景就不适合。

**Pipeline适用场景：**还有的系统，可能是批量的将数据写入redis，允许一定比例的写入失败，那么这种场景就可以使用了，比如10000条一下进入redis，可能失败了2条无所谓，后期有补偿机制就行了，比如短信群发这种场景，如果一下群发10000条，按照第一种模式去实现，那这个请求过来，要很久才能给客户端响应，这个延迟就太长了，如果客户端请求设置了超时时间5秒，那肯定就抛出异常了，而且本身群发短信要求实时性也没那么高，这时候用pipeline最好了。

**RabbitMQ**

**7.1生产者**



**7.2消费者**

