

前言

在生产环境中，我们也会将计算后的数据存储在 Redis 中，以供第三方的应用去 Redis 查找对应的数据。至于 Redis 的特性我不会在本节做过多的讲解。

安装 Redis

下载安装

先在 <https://redis.io/download> 下载到 Redis。

```
1 | wget http://download.redis.io/releases/redis-5.0.4.tar.gz
2 | tar xzf redis-5.0.4.tar.gz
3 | cd redis-5.0.4
4 | make
```

通过 HomeBrew 安装

```
1 | brew install redis
```

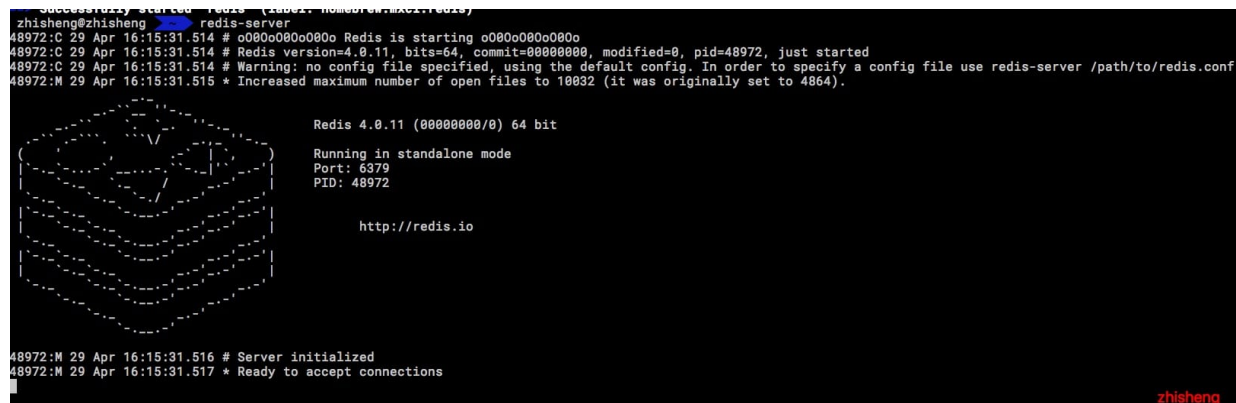
如果需要后台运行 Redis 服务，使用命令：

```
1 | brew services start redis
```

要运行命令，可以直接到 /usr/local/bin 目录下，有：

```
1 | redis-server
2 | redis-cli
```

两个命令，执行 `redis-server` 可以打开服务端：



```
Successfully started Redis (label: homebrew.mxcl.redis)
zhisheng@zhisheng:~$ redis-server
48972:C 29 Apr 16:15:31.514 # oO0oOoOoOoOo Redis is starting oO0oOoOoOoOo
48972:C 29 Apr 16:15:31.514 # Redis version=4.0.11, bits=64, commit=00000000, modified=0, pid=48972, just started
48972:C 29 Apr 16:15:31.514 # Warning: no config file specified, using the default config. In order to specify a config file use redis-server /path/to/redis.conf
48972:M 29 Apr 16:15:31.515 * Increased maximum number of open files to 10032 (it was originally set to 4864).

Redis 4.0.11 (00000000/0) 64 bit
Running in standalone mode
Port: 6379
PID: 48972

http://redis.io

48972:M 29 Apr 16:15:31.516 # Server initialized
48972:M 29 Apr 16:15:31.517 * Ready to accept connections
```

然后另外开一个终端，运行 `redis-cli` 命令可以运行客户端：

```
✖ zhisheng@zhisheng /usr/local redis-cli
127.0.0.1:6379> ping
PONG
127.0.0.1:6379> echo zhisheng
"zhisheng"
127.0.0.1:6379> █
```

zhisheng

Kafka 准备数据

这里我打算将从 Kafka 读取到所有到商品的信息，然后将商品信息中的 **商品ID** 和 **商品价格** 提取出来，然后写入到 Redis 中，供第三方服务根据商品 ID 查询到其对应的商品价格。

首先定义我们的商品类（其中 id 和 price 字段是我们最后要提取的）为：

ProductEvent.java

```
1  /**
2   * Desc: 商品
3   * Created by zhisheng on 2019-04-18
4   * blog: http://www.54tianzhisheng.cn/
5   * 微信公众号: zhisheng
6   */
7  @Data
8  @Builder
9  @AllArgsConstructor
10 @NoArgsConstructor
11 public class ProductEvent {
12
13     /**
14      * Product Id
15      */
16     private Long id;
17
18     /**
19      * Product 类目 Id
20      */
21     private Long categoryId;
22
23     /**
24      * Product 编码
25      */
26     private String code;
27
28     /**
29      * Product 店铺 Id
30      */
31     private Long shopId;
32
33     /**
34      * Product 店铺 name
35      */
36 }
```

```

36     private String shopName;
37
38     /**
39      * Product 品牌 Id
40      */
41     private Long brandId;
42
43     /**
44      * Product 品牌 name
45      */
46     private String brandName;
47
48     /**
49      * Product name
50      */
51     private String name;
52
53     /**
54      * Product 图片地址
55      */
56     private String imageUrl;
57
58     /**
59      * Product 状态 (1(上架), -1(下架), -2(冻结), -3(删除))
60      */
61     private int status;
62
63     /**
64      * Product 类型
65      */
66     private int type;
67
68     /**
69      * Product 标签
70      */
71     private List<String> tags;
72
73     /**
74      * Product 价格 (以分为单位)
75      */
76     private Long price;
77 }

```

然后写个工具类不断的模拟商品数据发往 Kafka, 工具类 ProductUtil.java :

```

1  package com.zhisheng.connectors.redis.utils;
2
3  import com.zhisheng.common.model.ProductEvent;
4  import com.zhisheng.common.utils.GsonUtil;
5  import org.apache.kafka.clients.producer.KafkaProducer;
6  import org.apache.kafka.clients.producer.ProducerRecord;
7
8  import java.util.Properties;
9  import java.util.Random;
10
11  /**
12   * blog: http://www.54tianzhisheng.cn/
13   * 微信公众号: zhisheng

```

```

14  */
15  public class ProductUtil {
16      public static final String broker_list = "localhost:9092";
17      public static final String topic = "zhisheng"; //kafka topic 需要和
flink 程序用同一个 topic
18
19      public static final Random random = new Random();
20
21      public static void main(String[] args) {
22          Properties props = new Properties();
23          props.put("bootstrap.servers", broker_list);
24          props.put("key.serializer",
"org.apache.kafka.common.serialization.StringSerializer");
25          props.put("value.serializer",
"org.apache.kafka.common.serialization.StringSerializer");
26          KafkaProducer producer = new KafkaProducer<String, String>(props);
27
28          for (int i = 1; i <= 10000; i++) {
29              ProductEvent product = ProductEvent.builder().id((long) i) //
商品的 id
30                  .name("product" + i) //商品 name
31                  .price(random.nextLong() / 1000000000000000L) //商品价格
(以分为单位)
32                  .code("code" + i).build(); //商品编码
33
34              ProducerRecord record = new ProducerRecord<String, String>
(topic, null, null, GsonUtil.toJson(product));
35              producer.send(record);
36              System.out.println("发送数据: " + GsonUtil.toJson(product));
37          }
38          producer.flush();
39      }
40  }

```

Flink Job 代码

我们需要在 Flink 中消费 Kafka 数据，然后将商品中的两个数据（商品 id 和 price）取出来。先来看下这段 Flink Job 代码：

```

1  public class Main {
2      public static void main(String[] args) throws Exception {
3          final StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
4          ParameterTool parameterTool = ExecutionEnvUtil.PARAMETER_TOOL;
5          Properties props = KafkaConfigUtil.buildKafkaProps(parameterTool);
6
7          SingleOutputStreamOperator<Tuple2<String, String>> product =
env.addSource(new FlinkKafkaConsumer011<>(
8              parameterTool.get(METRICS_TOPIC), //这个 kafka topic 需要
和上面的工具类的 topic 一致
9              new SimpleStringSchema(),
10              props))
11              .map(string -> GsonUtil.fromJson(string,
ProductEvent.class)) //反序列化 JSON

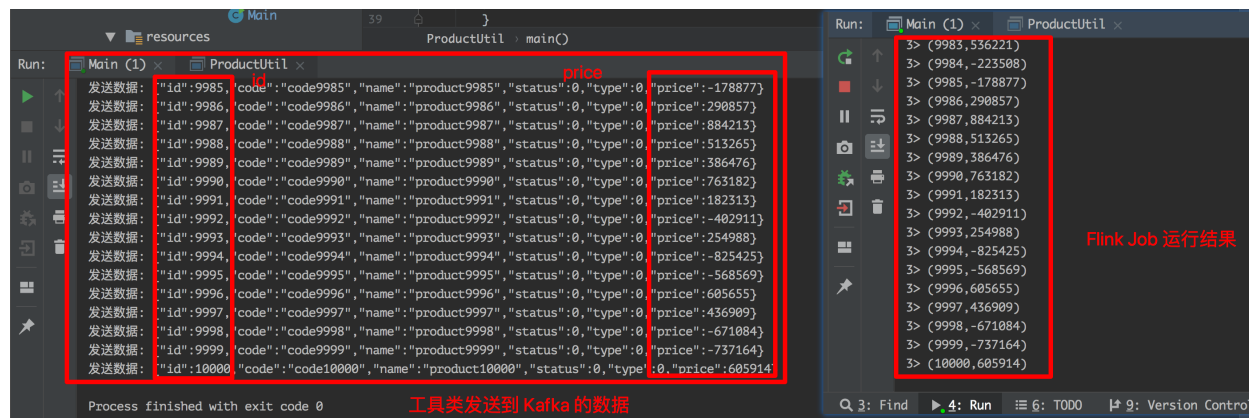
```

```

12         .flatMap(new FlatMapFunction<ProductEvent, Tuple2<String,
String>>>() {
13             @Override
14             public void flatMap(ProductEvent value,
Collector<Tuple2<String, String>> out) throws Exception {
15                 //收集商品 id 和 price 两个属性
16                 out.collect(new Tuple2<>(value.getId().toString(),
value.getPrice().toString()));
17             }
18         });
19         product.print();
20
21         env.execute("flink redis connector");
22     }
23 }

```

然后 IDEA 中启动运行 Job，再运行上面的 ProductUtil 发送 Kafka 数据的工具类，注意：也得提前启动 Kafka。



上图左半部分是工具类发送数据到 Kafka 打印的日志，右半部分是 Job 执行的结果，可以看到它已经将商品的 id 和 price 数据获取到了。

那么接下来我们需要的就是将这种 `Tuple2<Long, Long>` 格式的 KV 数据写入到 Redis 中去。要将数据写入到 Redis 的话是需要先添加依赖的。

Redis Connector 简介

Redis Connector 提供用于向 Redis 发送数据的接口的类。接收器可以使用三种不同的方法与不同类型的 Redis 环境进行通信：

- 单 Redis 服务器
- Redis 集群
- Redis Sentinel

添加依赖

需要添加 Flink Redis Sink 的 Connector，这个 Redis Connector 官方只有老的版本，后面也一直没有更新，所以可以看到网上有些文章都是添加老的版本的依赖：

```
1 <dependency>
2   <groupId>org.apache.flink</groupId>
3   <artifactId>flink-connector-redis_2.10</artifactId>
4   <version>1.1.5</version>
5 </dependency>
```

包括该部分的文档都是很早之前的啦，可以查看 <https://ci.apache.org/projects/flink/flink-docs-release-1.1/apis/streaming/connectors/redis.html>。

另外在 <https://bahir.apache.org/docs/flink/current/flink-streaming-redis/> 也看到一个 Flink Redis Connector 的依赖：

```
1 <dependency>
2   <groupId>org.apache.bahir</groupId>
3   <artifactId>flink-connector-redis_2.11</artifactId>
4   <version>1.0</version>
5 </dependency>
```

两个依赖功能都是一样的，我们还是就用官方的那个 Maven 依赖来进行演示。

Flink 代码

像写入到 Redis，那么肯定要配置 Redis 服务的地址（不管是单机的还是集群）。

单机的 Redis 你可以这样配置：

```
1 FlinkJedisPoolConfig conf = new
   FlinkJedisPoolConfig.Builder().setHost("127.0.0.1").build();
```

这个 FlinkJedisPoolConfig 源码中有四个属性：

```
1 private final String host;    //hostname or IP
2 private final int port;      //端口，默认 6379
3 private final int database;  //database index
4 private final String password; //password
```

另外你还可以通过 FlinkJedisPoolConfig 设置其他的几个属性（因为 FlinkJedisPoolConfig 继承自 FlinkJedisConfigBase，这几个属性在 FlinkJedisConfigBase 抽象类的）：

```
1 protected final int maxTotal;    //池可分配的对象最大数量，默认是 8
2 protected final int maxIdle;    //池中空闲的对象最大数量，默认是 8
3 protected final int minIdle;    //池中空闲的对象最小数量，默认是 0
4 protected final int connectionTimeout; //socket 或者连接超时时间，默认是
   2000ms
```

Redis 集群 你可以这样配置：

```

1 FlinkJedisClusterConfig config = new FlinkJedisClusterConfig.Builder()
2     .setNodes(new HashSet<InetSocketAddress>{
3         Arrays.asList(new InetSocketAddress("redis1",
4             6379))}).build();

```

Redis Sentinels 你可以这样配置：

```

1 FlinkJedisSentinelConfig sentinelConfig = new
FlinkJedisSentinelConfig.Builder()
2     .setMasterName("master")
3     .setSentinels(new HashSet<>(Arrays.asList("sentinel1",
"sentinel2")))
4     .setPassword("")
5     .setDatabase(1).build();

```

另外就是 Redis Sink 了，Redis Sink 核心类是 RedisMapper，它是一个接口，里面有三个方法，使用时我们需要重写这三个方法：

```

1 public interface RedisMapper<T> extends Function, Serializable {
2     //设置使用 Redis 的数据结构类型，和 key 的名词，RedisCommandDescription 中有两
    个属性 RedisCommand、key
3     RedisCommandDescription getCommandDescription();
4     //获取 key 值
5     String getKeyFromData(T var1);
6     //获取 value 值
7     String getValueFromData(T var1);
8 }

```

上面 RedisCommandDescription 中有两个属性 RedisCommand、key。RedisCommand 可以设置 Redis 的数据结果类型，下面是 Redis 数据结构的类型对应着的 Redis Command 的类型：

Data Type	Redis Command [Sink]
HASH	HSET
LIST	R PUSH, L PUSH
SET	SADD
PUBSUB	PUBLISH
STRING	SET
HYPER_LOG_LOG	PFADD
SORTED_SET	ZADD
SORTED_SET	ZREM

zhisheng

其对应的源码如下：

```

1 public enum RedisCommand {
2     LPUSH(RedisDataType.LIST),

```

```

3      RPUSH(RedisDataType.LIST),
4      SADD(RedisDataType.SET),
5      SET(RedisDataType.STRING),
6      PFADD(RedisDataType.HYPER_LOG_LOG),
7      PUBLISH(RedisDataType.PUBSUB),
8      ZADD(RedisDataType.SORTED_SET),
9      HSET(RedisDataType.HASH);
10
11     private RedisDataType redisDataType;
12
13     private RedisCommand(RedisDataType redisDataType) {
14         this.redisDataType = redisDataType;
15     }
16
17     public RedisDataType getRedisDataType() {
18         return this.redisDataType;
19     }
20 }

```

我们实现这个 RedisMapper 接口如下：

```

1 public static class RedisSinkMapper implements RedisMapper<Tuple2<String,
String>> {
2     @Override
3     public RedisCommandDescription getCommandDescription() {
4         //指定 RedisCommand 的类型是 HSET, 对应 Redis 中的数据结构是 HASH, 另外设置 key = zhisheng
5         return new RedisCommandDescription(RedisCommand.HSET, "zhisheng");
6     }
7
8     @Override
9     public String getKeyFromData(Tuple2<String, String> data) {
10         return data.f0;
11     }
12
13     @Override
14     public String getValueFromData(Tuple2<String, String> data) {
15         return data.f1;
16     }
17 }

```

然后在 Flink Job 中加入下面这行，将数据通过 RedisSinkMapper 写入到 Redis 中去：

```

1 product.addSink(new RedisSink<Tuple2<String, String>>(conf, new
RedisSinkMapper()));

```

运行结果

运行 Job 的话，就是把数据已经插入进 Redis 了，那么如何验证我们的结果是否正确呢？

1、我们去终端 Cli 执行命令查看这个 zhisheng 的 key，然后查找某个商品 id (1 ~ 10000) 对应的商品价格，超过这个 id 则为 nil。


```
127.0.0.1:6379> HGET HASH_NAME 9999
"477562"
127.0.0.1:6379> HGET zhsheng 9999
(nil)
127.0.0.1:6379> HGET zhsheng 1
(nil)
127.0.0.1:6379> HSET myhash field1 111
(integer) 0
127.0.0.1:6379> HSET myhash field 111
(integer) 1
127.0.0.1:6379> HGET myhash field
"111"
127.0.0.1:6379> del zhisheng
(integer) 1
127.0.0.1:6379> HGET zhisheng 111
"-556968"
127.0.0.1:6379> HGET zhisheng 2222
"-505976"
127.0.0.1:6379> HGET zhisheng 888
"469088"
127.0.0.1:6379> HGET zhisheng 88899
(nil)
127.0.0.1:6379> █
```

先删除 zhisheng 这个 key 然后运行 Job

查找商品 id 对应的价钱

不存在这个商品 id，则不存在

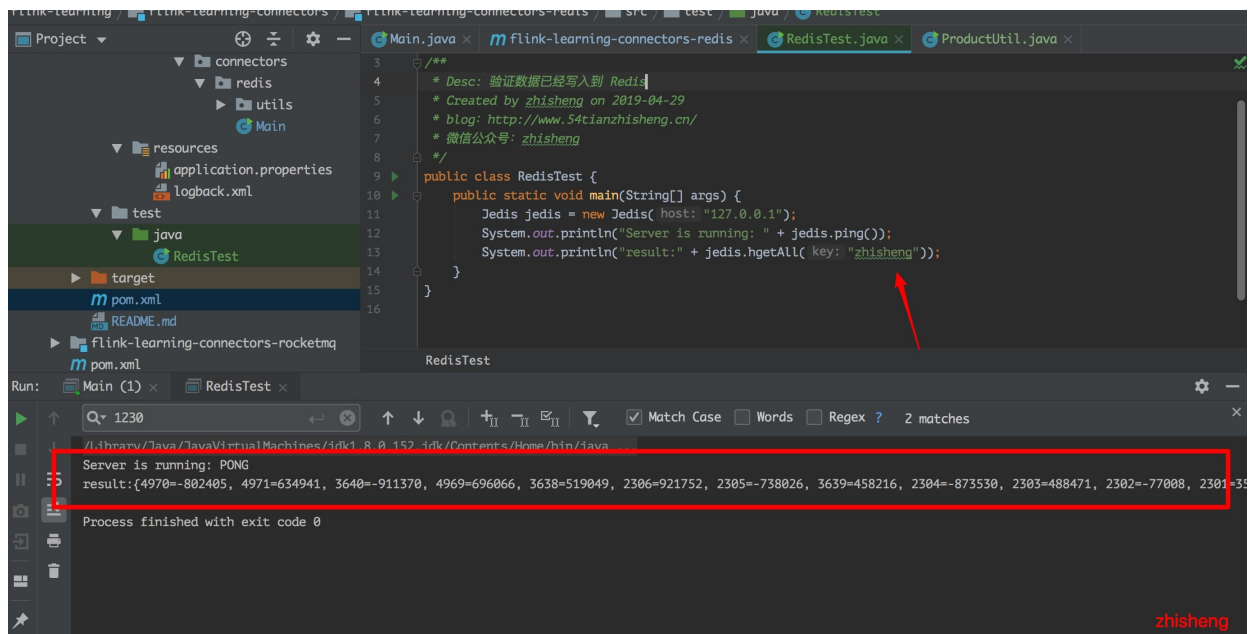
2、另外一种验证的方式就是通过 Java 代码来操作 Redis 查询数据了。

我们先引入 Redis 的依赖：

```
1 <dependency>
2   <groupId>redis.clients</groupId>
3   <artifactId>jedis</artifactId>
4   <version>2.9.0</version>
5 </dependency>
```

连接 Redis 查询数据：

```
1 public class RedisTest {
2     public static void main(String[] args) {
3         Jedis jedis = new Jedis("127.0.0.1");
4         System.out.println("Server is running: " + jedis.ping());
5         System.out.println("result:" + jedis.hgetAll("zhisheng"));
6     }
7 }
```



```
3  /**
4  * Desc: 验证数据已经写入到 Redis
5  * Created by zhisheng on 2019-04-29
6  * blog: http://www.54tianzhisheng.cn/
7  * 微信公众号: zhisheng
8  */
9  public class RedisTest {
10     public static void main(String[] args) {
11         Jedis jedis = new Jedis( host: "127.0.0.1");
12         System.out.println("Server is running: " + jedis.ping());
13         System.out.println("result:" + jedis.hgetAll( key: "zhisheng"));
14     }
15 }
16
```

Run: Main (1) x RedisTest x

1230

Server is running: PONG
result:{4970=-802405, 4971=634941, 3640=-911370, 4969=696066, 3638=519049, 2306=921752, 2305=-738026, 3639=458216, 2304=-873530, 2303=488471, 2302=-77008, 2301=35}

Process finished with exit code 0

zhisheng

这一行把所有数据都打印出来了，所以我们的数据确实成功地插入到 Redis 中去了。

总结

本文先讲解了 Redis 的安装，然后讲了 Flink 如何消费 Kafka 的数据并将数据写入到 Redis 中去。在实战的过程中还分析了 Flink Redis Connector 中的原理，只要我们懂得了这些原理，后面再去做这块的需求就难不倒大家了。

Github 代码仓库

<https://github.com/zhisheng17/flink-learning/tree/master/flink-learning-connectors/flink-learning-connectors-redis>