Java API方式调用Kafka各种协议

众所周知，Kafka自己实现了一套二进制协议(binary protocol)用于各种功能的实现，比如发送消息，获取消息，提交位移以及创建topic等。具体协议规范参见：Kafka协议
这套协议的具体使用流程为：

- 客户端创建对应协议的请求
- 客户端发送请求给对应的broker
- broker处理请求，并发送response给客户端

如果想及时了解Spark、Hadoop或者HBase相关的文章，欢迎关注微信公众号：iteblog_hadoop

虽然Kafka提供的大量的脚本工具用于各种功能的实现，但很多时候我们还是希望可以把某些功能以编程的方式嵌入到另一个系统中。这时使用Java
API的方式就显得异常地灵活了。本文我将尝试给出Java API底层框架的一个范例，同时也会针对“创建topic”和“查看位移”这两个主要功能给出对应的例子。 需要提前说明的是，本文给出的范例并没有考虑Kafka集群开启安全的情况。另外Kafka的KIP4应该一直在优化命令行工具以及各种管理操作，有兴趣的读者可以关注这个KIP。

本文中用到的API依赖于kafka-clients，所以如果你使用Maven构建的话，请加上：

```
<dependency>
   <groupId>org.apache.kafka</groupId>
   <artifactId>kafka-clients</artifactId>
```

```
    <version>0.10.2.0</version>
</dependency>
```

准备

```
/**
     * 发送请求主方法
     * @param host          目标broker的主机名
     * @param port          目标broker的端口
     * @param request       请求对象
     * @param apiKey        请求类型
     * @return              序列化后的response
     * @throws IOException
     */
    public ByteBuffer send(String host, int port, AbstractRequest request, ApiKeys apiKey) throw
s IOException {
        Socket socket = connect(host, port);
        try {
            return send(request, apiKey, socket);
        } finally {
            socket.close();
        }
    }

    /**
     * 发送序列化请求并等待response返回
     * @param socket            连向目标broker的socket
     * @param request           序列化后的请求
     * @return                  序列化后的response
     * @throws IOException
     */
    private byte[] issueRequestAndWaitForResponse(Socket socket, byte[] request) throws IOExc
eption {
        sendRequest(socket, request);
        return getResponse(socket);
    }

    /**
     * 发送序列化请求给socket
     * @param socket            连向目标broker的socket
     * @param request           序列化后的请求
     * @throws IOException
     */
```

```
private void sendRequest(Socket socket, byte[] request) throws IOException {
    DataOutputStream dos = new DataOutputStream(socket.getOutputStream());
    dos.writeInt(request.length);
    dos.write(request);
    dos.flush();
}

/**
 * 从给定socket处获取response
 * @param socket          连向目标broker的socket
 * @return               获取到的序列化后的response
 * @throws IOException
 */
private byte[] getResponse(Socket socket) throws IOException {
    DataInputStream dis = null;
    try {
        dis = new DataInputStream(socket.getInputStream());
        byte[] response = new byte[dis.readInt()];
        dis.readFully(response);
        return response;
    } finally {
        if (dis != null) {
            dis.close();
        }
    }
}

/**
 * 创建Socket连接
 * @param hostName        目标broker主机名
 * @param port            目标broker服务端口, 比如9092
 * @return               创建的Socket连接
 * @throws IOException
 */
private Socket connect(String hostName, int port) throws IOException {
    return new Socket(hostName, port);
}

/**
 * 向给定socket发送请求
 * @param request      请求对象
 * @param apiKey       请求类型, 即属于哪种请求
 * @param socket       连向目标broker的socket
 * @return           序列化后的response
 * @throws IOException
 */
```

```
    private ByteBuffer send(AbstractRequest request, ApiKeys apiKey, Socket socket) throws IOE
xception {
        RequestHeader header = new RequestHeader(apiKey.id, request.version(), "client-id", 0);
        ByteBuffer buffer = ByteBuffer.allocate(header.sizeOf() + request.sizeOf());
        header.writeTo(buffer);
        request.writeTo(buffer);
        byte[] serializedRequest = buffer.array();
        byte[] response = issueRequestAndWaitForResponse(socket, serializedRequest);
        ByteBuffer responseBuffer = ByteBuffer.wrap(response);
        ResponseHeader.parse(responseBuffer);
        return responseBuffer;
    }
```

有了这些方法的铺垫，我们就可以创建具体的请求了。

创建topic

```
/**
     * 创建topic
     * 由于只是样例代码，有些东西就硬编码写到程序里面了(比如主机名和端口)，各位看官自行
修改即可
     * @param topicName           topic名
     * @param partitions          分区数
     * @param replicationFactor     副本数
     * @throws IOException
     */
    public void createTopics(String topicName, int partitions, short replicationFactor) throws IO
Exception {
        Map<String, CreateTopicsRequest.TopicDetails> topics = new HashMap<>();
        // 插入多个元素便可同时创建多个topic
        topics.put(topicName, new CreateTopicsRequest.TopicDetails(partitions, replicationFactor
));
        int creationTimeoutMs = 60000;
        CreateTopicsRequest request = new CreateTopicsRequest.Builder(topics, creationTimeout
Ms).build();
        ByteBuffer response = send("localhost", 9092, request, ApiKeys.CREATE_TOPICS);
        CreateTopicsResponse.parse(response, request.version());
    }
```

查看偏移量

```
/**
    * 获取某个consumer group下的某个topic分区的位移
    * @param groupID          group id
    * @param topic           topic名
    * @param parititon        分区号
    * @throws IOException
    */
   public void getOffsetForPartition(String groupID, String topic, int parititon) throws IOExcepti
on {
       TopicPartition tp = new TopicPartition(topic, parititon);
       OffsetFetchRequest request = new OffsetFetchRequest.Builder(groupID, singletonList(tp))
               .setVersion((short)2).build();
       ByteBuffer response = send("localhost", 9092, request, ApiKeys.OFFSET_FETCH);
       OffsetFetchResponse resp = OffsetFetchResponse.parse(response, request.version());
       OffsetFetchResponse.PartitionData partitionData = resp.responseData().get(tp);
       System.out.println(partitionData.offset);
   }

/**
    * 获取某个consumer group下所有topic分区的位移信息
    * @param groupID          group id
    * @return                (topic分区 --> 分区信息)的map
    * @throws IOException
    */
   public Map<TopicPartition, OffsetFetchResponse.PartitionData> getAllOffsetsForGroup(Strin
g groupID) throws IOException {
       OffsetFetchRequest request = new OffsetFetchRequest.Builder(groupID, null).setVersion((
short)2).build();
       ByteBuffer response = send("localhost", 9092, request, ApiKeys.OFFSET_FETCH);
       OffsetFetchResponse resp = OffsetFetchResponse.parse(response, request.version());
       return resp.responseData();
   }
```

其他 API 的调用可以参考上面两个例子，这里不就再介绍了。

本文原文：https://www.cnblogs.com/huxi2b/p/6508274.html