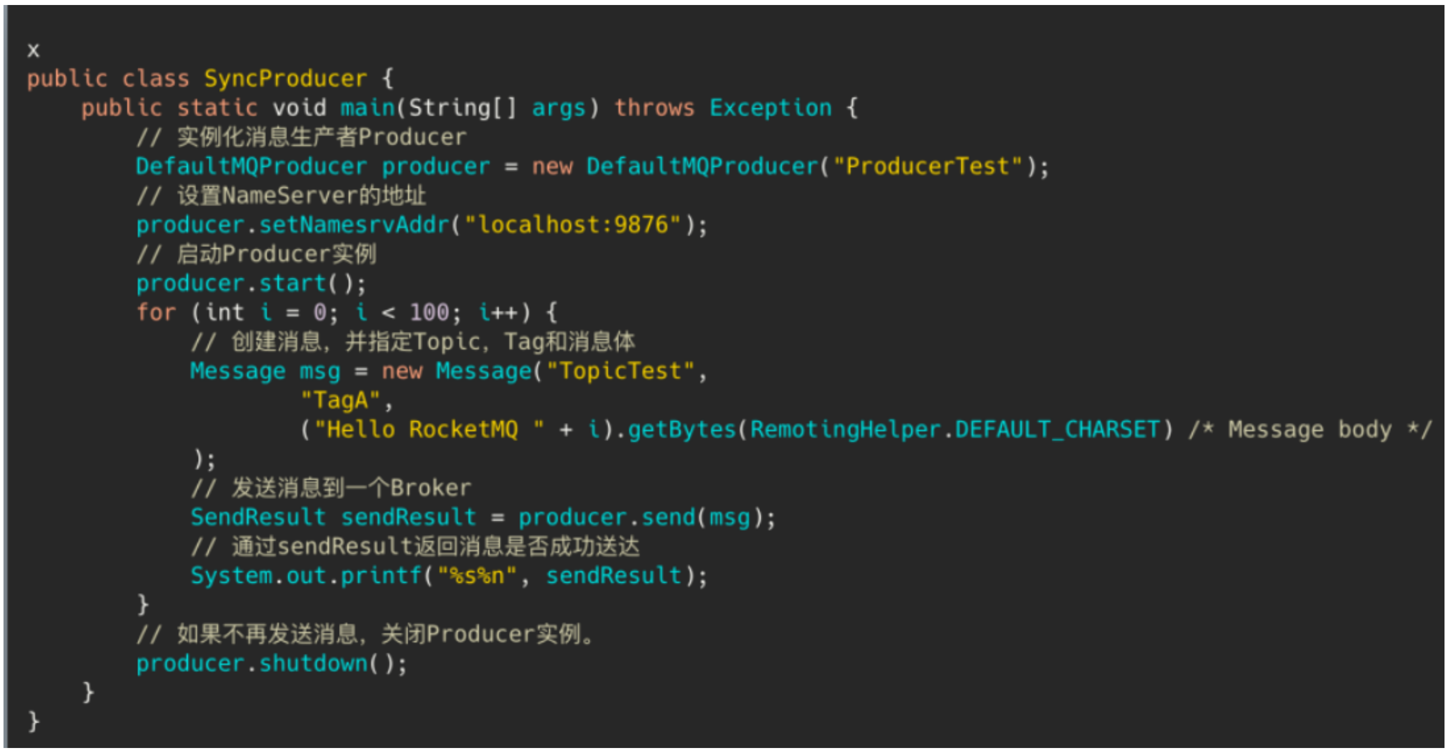
3.24课后思考题

1. 为什么Demo里面没有创建Topic，却可以使用

因为RocketMQ的自动创建Topic机制。

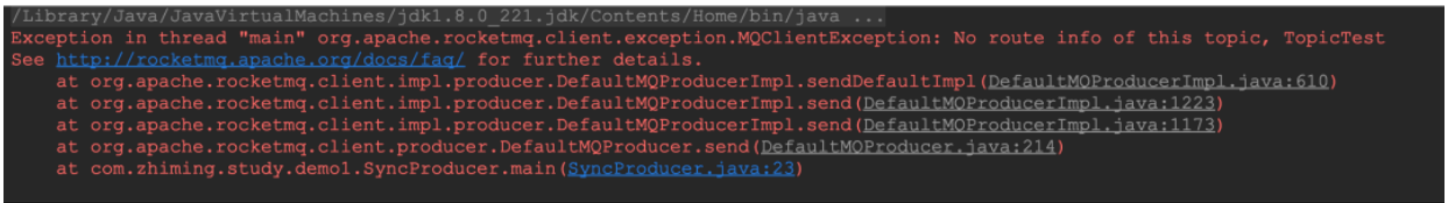
下载了 RocketMQ 源码，在本地 IDE 中正常启动了 Broker 服务 和 NameServer 服务。

生产者发送消息的 demo 如下：



```
x
public class SyncProducer {
    public static void main(String[] args) throws Exception {
        // 实例化消息生产者Producer
        DefaultMQProducer producer = new DefaultMQProducer("ProducerTest");
        // 设置NameServer的地址
        producer.setNamesrvAddr("localhost:9876");
        // 启动Producer实例
        producer.start();
        for (int i = 0; i < 100; i++) {
            // 创建消息，并指定Topic，Tag和消息体
            Message msg = new Message("TopicTest",
                    "TagA",
                    ("Hello RocketMQ " + i).getBytes(RemotingHelper.DEFAULT_CHARSET) /* Message body */
            );
            // 发送消息到一个Broker
            SendResult sendResult = producer.send(msg);
            // 通过sendResult返回消息是否成功送达
            System.out.printf("%s%n", sendResult);
        }
        // 如果不再发送消息，关闭Producer实例。
        producer.shutdown();
    }
}
```

在发送消息的过程中报错了，报错信息如下：



```
/Library/Java/JavaVirtualMachines/jdk1.8.0_221.jdk/Contents/Home/bin/java ...
Exception in thread "main" org.apache.rocketmq.client.exception.MQClientException: No route info of this topic, TopicTest
See http://rocketmq.apache.org/docs/faq/ for further details.
    at org.apache.rocketmq.client.impl.producer.DefaultMQProducerImpl.sendDefaultImpl(DefaultMQProducerImpl.java:610)
    at org.apache.rocketmq.client.impl.producer.DefaultMQProducerImpl.send(DefaultMQProducerImpl.java:1223)
    at org.apache.rocketmq.client.impl.producer.DefaultMQProducerImpl.send(DefaultMQProducerImpl.java:1173)
    at org.apache.rocketmq.client.producer.DefaultMQProducer.send(DefaultMQProducer.java:214)
    at com.zhiming.study.demo1.SyncProducer.main(SyncProducer.java:23)
```

从报错信息可以看到：**Topic 对应的路由信息没有正常获取到（Topic 名称为 TopicTest）**。也难怪，Broker 启动后没有人为地去创建 Topic 对应的队列，发送消息自然没办法获取到路由信息。

想到 Broker 有自动创建 Topic 的机制，可以通过在 Broker 的配置文件中添加如下配置：

```
autoCreateTopicEnable = true
```

问题在于即使配置了这个参数，重启 Broker 服务之后，发送消息时还是如上的报错，因此打算仔细看看 RocketMQ 自动创建 Topic 的实现原理。

自动创建Topic机制

首先需要解决两个问题

- W1：Broker 注册路由信息方式
- W2：生产者获取路由信息的策略

路由信息注册（Broker）

- Broker 节点在启动时会先初始化当前 Broker 的路由信息，根据 autoCreateTopicEnable 参数的配置情况决定是否要添加【默认主题】的路由信息
- Broker 节点会定期向 NameServer 注册路由信息

路由信息主要包含的内容

- 当前 Broker 节点存在的 Topic 列表
- 每个 Topic 对应的队列分配情况（读写队列数量）

1. Broker 节点启动时，会开启一个定时任务，定期向 NameServer 注册路由信息

```
x
org.apache.rocketmq.broker.BrokerController

public void start() throws Exception {
    // .....省略代码
    this.scheduledExecutorService.scheduleAtFixedRate(new Runnable() {

        @Override
        public void run() {
            try {
                BrokerController.this.registerBrokerAll(true, false, brokerConfig.isForceRegister());
            } catch (Throwable e) {
                log.error("registerBrokerAll Exception", e);
            }
        }
    }, 1000 * 10, Math.max(10000, Math.min(brokerConfig.getRegisterNameServerPeriod(), 60000)), TimeUnit.

    // ......省略代码

}
```

2. 调用具体的注册方法，topicConfigWrapper 对象中包含本次需要注册的路由信息

```
x
org.apache.rocketmq.broker.BrokerController

public synchronized void registerBrokerAll(final boolean checkOrderConfig, boolean oneway, boolean forceF
  // 构建 topic 路由信息
  TopicConfigSerializeWrapper topicConfigWrapper = this.getTopicConfigManager().buildTopicConfigSerialize

  // ......省略代码
  if (forceRegister || needRegister(this.brokerConfig.getBrokerClusterName(),
                                    this.getBrokerAddr(),
                                    this.brokerConfig.getBrokerName(),
                                    this.brokerConfig.getBrokerId(),
                                    this.brokerConfig.getRegisterBrokerTimeoutMills())) {
    // 注册路由动作
    doRegisterBrokerAll(checkOrderConfig, oneway, topicConfigWrapper);
  }
}
```

3. topicConfigWrapper 对象的构建过程，其实是把 topicConfigTable 对象包装了一层

```
x
org.apache.rocketmq.broker.topic.TopicConfigManager

public TopicConfigSerializeWrapper buildTopicConfigSerializeWrapper() {
        TopicConfigSerializeWrapper topicConfigSerializeWrapper = new TopicConfigSerializeWrapper();
        topicConfigSerializeWrapper.setTopicConfigTable(this.topicConfigTable);
        topicConfigSerializeWrapper.setDataVersion(this.dataVersion);
        return topicConfigSerializeWrapper;
    }
```

4. topicConfigTable 对象类型为 ConcurrentMap，Map 的 Key 为 Topic 名称，Value 为 Topic 对应的路由信息（包含读写队列数量信息）

```
x
org.apache.rocketmq.broker.topic.TopicConfigManager

private final ConcurrentMap<String, TopicConfig> topicConfigTable =
new ConcurrentHashMap<String, TopicConfig>(1024);
```

5. 下面看到 topicConfigTable 对象的初始化过程，主要看到关于【自动创建 Topic】 的逻辑，其它代码先省略（初始化了一系列系统预设的 Topic 路由信息）

- 判断 Broker 是否开启了自动创建 Topic 的开关，即开头讲的 autoCreateTopicEnable = true 的配置
- 如果开启了自动创建 Topic，则会往 topicConfigTable 对象中放入一个【默认主题】的路由信息， 名称为 TBW102，这边将它称为【TBW102默认路由信息】
- 因此最终往 NameServer 中注册的路由信息包含 【TBW102默认路由信息】

```
x
org.apache.rocketmq.broker.topic.TopicConfigManager

public TopicConfigManager(BrokerController brokerController) {
  // ......省略代码, 初始化

  // 判断 Broker 是否开启了自动创建 Broker 的开关
  if (this.brokerController.getBrokerConfig().isAutoCreateTopicEnable()) {
    // 添加默认的 Topic 路由信息,
    String topic = TopicValidator.AUTO_CREATE_TOPIC_KEY_TOPIC;
    TopicConfig topicConfig = new TopicConfig(topic);
    TopicValidator.addSystemTopic(topic);
    topicConfig.setReadQueueNums(this.brokerController.getBrokerConfig()
                                .getDefaultTopicQueueNums());
    topicConfig.setWriteQueueNums(this.brokerController.getBrokerConfig()
                                 .getDefaultTopicQueueNums());
    int perm = PermName.PERM_INHERIT | PermName.PERM_READ | PermName.PERM_WRITE;
    topicConfig.setPerm(perm);
    this.topicConfigTable.put(topicConfig.getTopicName(), topicConfig);
  }

    // ......省略代码, 初始化
}
```

路由信息获取（生产者）

- 生产者发送消息时，会根据当前消息指定的 Topic 查询路由信息，如果本地缓存没有查询到，则尝试从 NameServer 服务查询
- 当没有查询到指定 Topic 的路由信息时，会使用系统【默认主题】的名称再次尝试查询路由，如果查询到【默认主题】的路由信息，则正常发送消息

1. 发送消息前，先尝试获取消息对应 Topic 的路由信息

```
x
org.apache.rocketmq.client.impl.producer.DefaultMQProducerImpl

private SendResult sendDefaultImpl(
  Message msg,
  final CommunicationMode communicationMode,
  final SendCallback sendCallback,
  final long timeout
) throws MQClientException, RemotingException, MQBrokerException, InterruptedException {
    // ...... 省略代码

  // 获取路由信息
  TopicPublishInfo topicPublishInfo = this.tryToFindTopicPublishInfo(msg.getTopic());

  if (topicPublishInfo != null && topicPublishInfo.ok()) {
    // ...... 省略代码
    // 根据获取到的路由信息, 选择合适的 Broker, 发送消息
  }
    // 省略代码
}
```

2. 获取路由信息

```
x
org.apache.rocketmq.client.impl.producer.DefaultMQProducerImpl

private TopicPublishInfo tryToFindTopicPublishInfo(final String topic) {
    // 先尝试从本地缓存获取
    TopicPublishInfo topicPublishInfo = this.topicPublishInfoTable.get(topic);
    if (null == topicPublishInfo || !topicPublishInfo.ok()) {
        // 本地没有获取到, 则尝试从 NameServer 获取
        this.topicPublishInfoTable.putIfAbsent(topic, new TopicPublishInfo());
        this.mQClientFactory.updateTopicRouteInfoFromNameServer(topic);
        topicPublishInfo = this.topicPublishInfoTable.get(topic);
    }

    if (topicPublishInfo.isHaveTopicRouterInfo() || topicPublishInfo.ok(){
        return topicPublishInfo;
    } else {
        // NameServer 不存在指定 Topic 的路由信息, 则尝试获取默认主题的路由信息
        this.mQClientFactory.updateTopicRouteInfoFromNameServer(topic, true, this.defaultMQProducer);
        topicPublishInfo = this.topicPublishInfoTable.get(topic);
        return topicPublishInfo;
    }
}
```

3. 获取默认的路由信息，如果 isDefault = true，则尝试获取【默认主题】的路由信息， Topic 名称为 TBW102（与 Broker 注册的默认主题的路由名称相同）

```
x
org.apache.rocketmq.client.impl.producer.DefaultMQProducerImpl

public boolean updateTopicRouteInfoFromNameServer(final String topic, boolean isDefault,
DefaultMQProducer defaultMQProducer) {
    TopicRouteData topicRouteData;
    // 如果 isDefault = true，则尝试获取默认主题的路由信息，Topic 名称为 TBW102
    if (isDefault && defaultMQProducer != null) {
      topicRouteData = this.mQClientAPIImpl.getDefaultTopicRouteInfoFromNameServer(defaultMQProducer.get(
                                                                                    1000 * 3);

    } else {
      // 反之则通过指定 Topic 获取路由信息
      topicRouteData = this.mQClientAPIImpl.getTopicRouteInfoFromNameServer(topic, 1000 * 3);
    }
}
```

创建路由信息（Broker）

Broker 接收到消息之后，会先检查消息的 Topic 是否存在；如果消息对应的 Topic 不存在，且 Broker 允许自动创建不存在的 Topic，则会自动创建 Topic。

1. 先检查 Topic 的路由信息是否存在，如果不存在，则自动创建 Topic

```
x
org.apache.rocketmq.broker.processor.AbstractSendMessageProcessor

protected RemotingCommand msgCheck(final ChannelHandlerContext ctx,
final SendMessageRequestHeader requestHeader, final RemotingCommand response) {
  // ...... 省略代码

  // 查询 Topic 路由信息
  TopicConfig topicConfig =  this.brokerController.getTopicConfigManager().selectTopicConfig(requestHeade

  if (null == topicConfig) {
    log.warn("the topic {} not exist, producer: {}", requestHeader.getTopic(), ctx.channel().remoteAddres
    // 如果 Topic 不存在，则尝试自动创建 Topic
    topicConfig = this.brokerController.getTopicConfigManager().createTopicInSendMessageMethod(
      requestHeader.getTopic(),
      requestHeader.getDefaultTopic(),
      RemotingHelper.parseChannelRemoteAddr(ctx.channel()),
      requestHeader.getDefaultTopicQueueNums(), topicSysFlag);
  }

  // ...... 省略代码
}
```

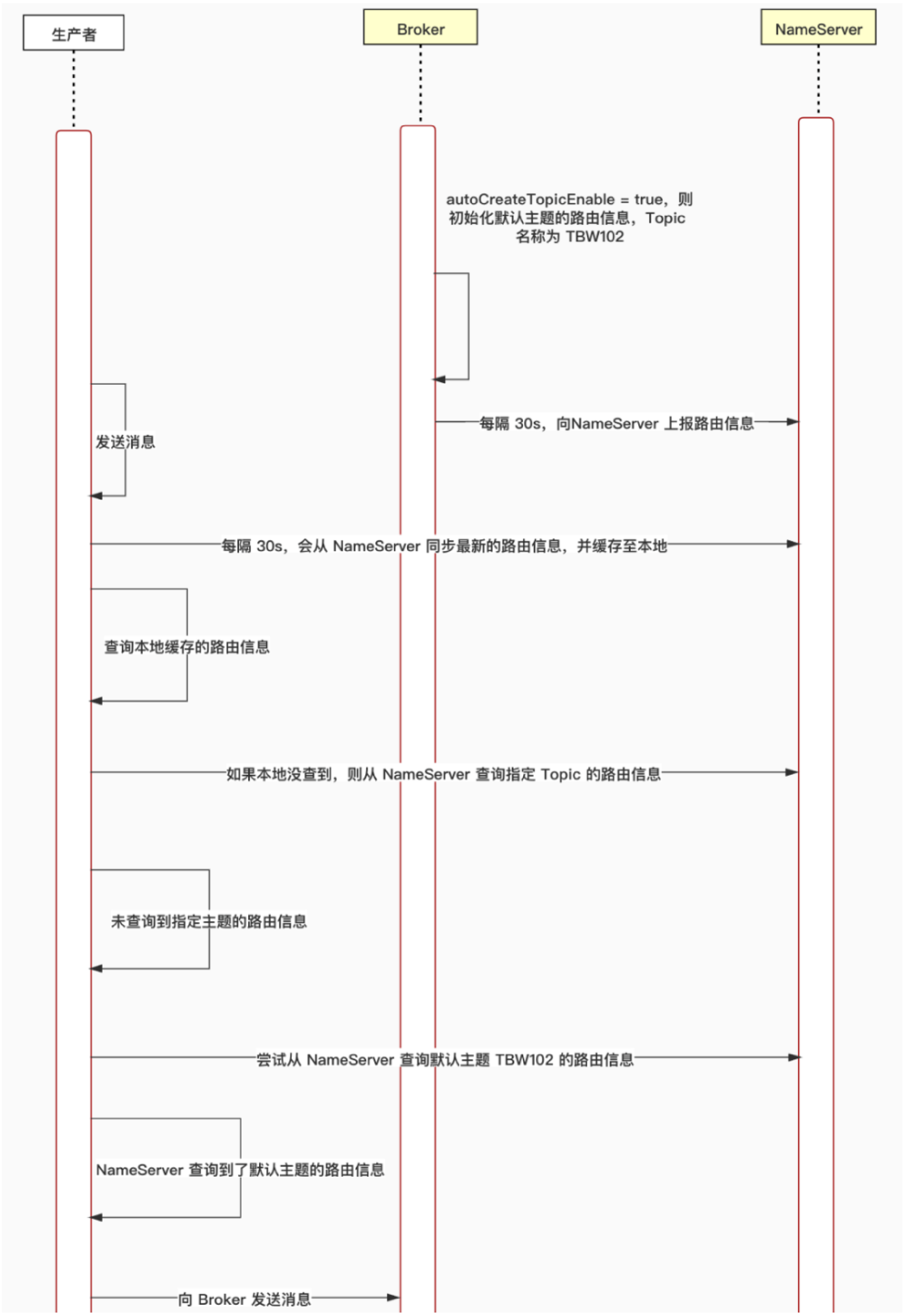
2. 具体自动创建 Topic 的操作，通过检查路由信息中是否包含【默认主题】的路由来判断 Broker 是否开启了自动创建 Topic

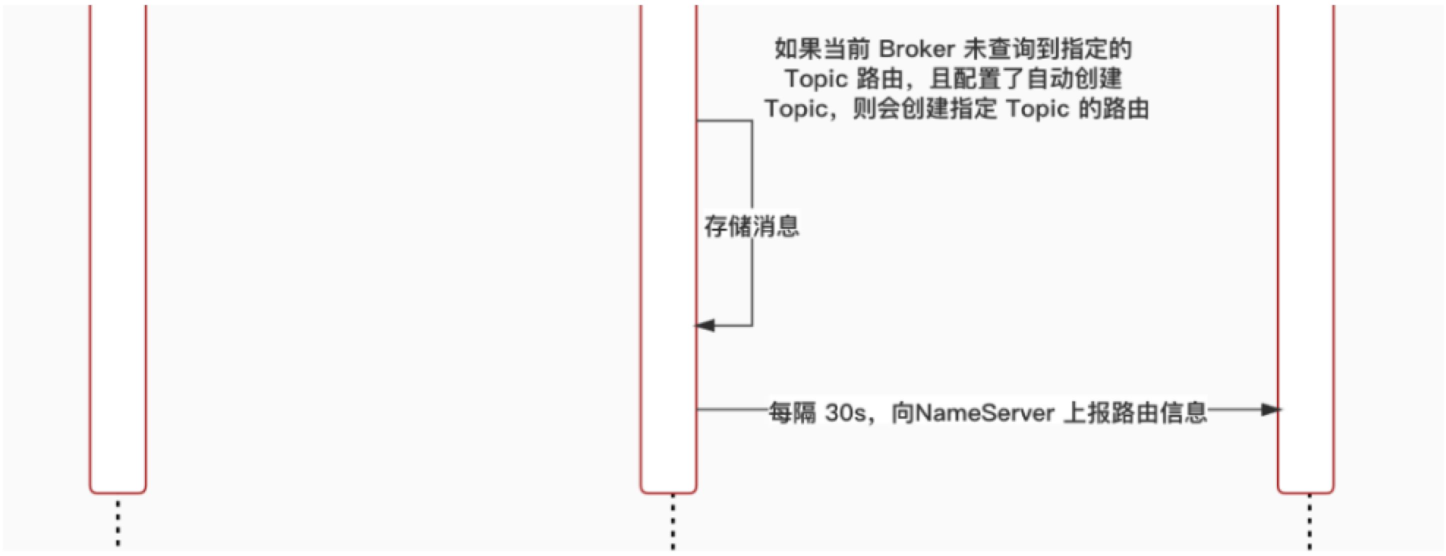
```
x
org.apache.rocketmq.broker.topic.TopicConfigManager

public TopicConfig createTopicInSendMessageMethod(final String topic, final String defaultTopic,
final String remoteAddress,
final int clientDefaultTopicQueueNums, final int topicSysFlag) {
  // 再次尝试获取路由信息，如果获取到则直接返回
  topicConfig = this.topicConfigTable.get(topic);
  if (topicConfig != null)
    return topicConfig;

  // 获取默认路由信息，如果获取到则认为 Broker 开启了自动创建 Topic
  TopicConfig defaultTopicConfig = this.topicConfigTable.get(defaultTopic);
  if (defaultTopicConfig != null) {
    // 创建 Topic 路由信息
  }

  // ...... 省略代码
}
```

整体流程

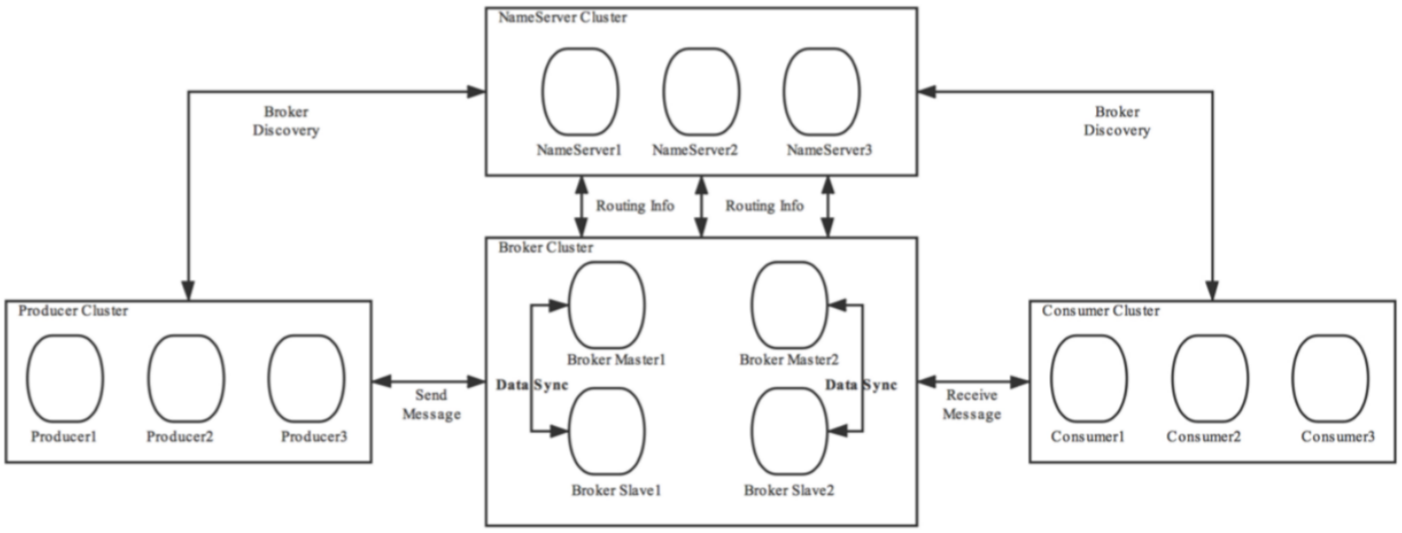
我把自动创建 Topic 的流程概括为【偷梁换柱】，这个活是由 Broker，NameServer，生产者配合完成的

- 生产发送消息时，如果指定的 Topic 不存在，NameServer 会返回一个【默认主题】的路由信息，使得生产者能够正常发生消息
- Broker 收到消息后，发现消息对应 Topic 不存在，且 Broker 允许自动创建 Topic，则会为消息创建 Topic ，并定时把路由信息同步至 NameServer
- 生产者也会定时从 NameServer 同步最新的路由信息，缓存至本地
- 后续生产者发送消息时，就可以从本地的缓存中查询到对应 Topic 的路由信息了


生产者
Broker
NameServer
autoCreateTopicEnable = true，则初始化默认主题的路由信息，Topic 名称为 TBW102
每隔 30s，向NameServer 上报路由信息
发送消息
每隔 30s，会从 NameServer 同步最新的路由信息，并缓存至本地
查询本地缓存的路由信息
如果本地没查到，则从 NameServer 查询指定 Topic 的路由信息
未查询到指定主题的路由信息
尝试从 NameServer 查询默认主题 TBW102 的路由信息
NameServer 查询到了默认主题的路由信息
向 Broker 发送消息


如果当前 Broker 未查询到指定的 Topic 路由，且配置了自动创建 Topic，则会创建指定 Topic 的路由
存储消息
每隔 30s，向NameServer 上报路由信息

2. 元数据的生命周期



我们来看下 **RouteInfoManager** 都管理了哪些元数据：

```
// topic 消息队列路由信息，消息发送时根据路由表进行负载均衡
private final HashMap<String/* topic */, List<QueueData>> topicQueueTable;
// Broker 基础信息，包含 brokerName、所属集群名称、主备 Broker 地址
private final HashMap<String/* brokerName */, BrokerData> brokerAddrTable;
// Broker 集群信息，存储集群中所有 Broker 名称
private final HashMap<String/* clusterName */, Set<String/* brokerName */>> clusterAddrTable;
// Broker 状态信息。NameServer 每次收到心跳包时会替换该信息
private final HashMap<String/* brokerAddr */, BrokerLiveInfo> brokerLiveTable;
// Broker 上的 FilterServer 列表，用于类模式消息过滤
private final HashMap<String/* brokerAddr */, List<String>/* Filter Server */> filterServerTable;
```

一个 topic 拥有多个消息队列，一个 Broker 为每一个 topic 默认创建4个读队列4个写队列。多个 Broker 组成一个集群，多台具有相同 BrokerName 的 Broker 组成 Master-Slave 架构，brokerId 为0代表 Master，大于0表示 Slave。BrokerLiveInfo 中的 lastUpdateTimestamp 存储上次收到 Broker 心跳包的时间。

RocketMQ 路由注册是通过 Broker 与 NameServer 之间的心跳功能实现的。Broker 启动时向集群中所有的 NameServer 发送心跳请求，并且每隔30s向集群中所有 NameServer 发送心跳包，NameServer 收到 Broker 心跳包时会更新 brokerLiveTable 缓存中 BrokerLiveInfo的 lastUpdateTimestamp，然后 NameServer 每隔10s扫描 brokerLiveTable，如果连续120s没有收到心跳包，NameServer 将移除该 Broker 的路由信息同时关闭 Socket 连接。

路由信息注册

在 Broker 的启动流程中，我们看到在 BrokerController#start 方法中启动了定时任务来调用 registerBrokerAll 方法，这个方法实际上就是将 Broker 的信息注册到 NameServer 上成为 NameServer 管理的路由元数据信息：

BrokerController：

```java
public synchronized void registerBrokerAll(final boolean checkOrderConfig, boolean oneway, boolean forceRegister) {
        // Broker 上的 topic 配置信息
    TopicConfigSerializeWrapper topicConfigWrapper = this.getTopicConfigManager().buildTopicConfigSerializeWrapper();
    if (!PermName.isWriteable(this.getBrokerConfig().getBrokerPermission())
        || !PermName.isReadable(this.getBrokerConfig().getBrokerPermission())) {
        ConcurrentHashMap<String, TopicConfig> topicConfigTable = new ConcurrentHashMap<String, TopicConfig>();
        for (TopicConfig topicConfig : topicConfigWrapper.getTopicConfigTable().values()) {
            TopicConfig tmp =
                new TopicConfig(topicConfig.getTopicName(), topicConfig.getReadQueueNums(), topicConfig.getWriteQueueNums(),
                    this.brokerConfig.getBrokerPermission());
            topicConfigTable.put(topicConfig.getTopicName(), tmp);
        }
        topicConfigWrapper.setTopicConfigTable(topicConfigTable);
    }
    /* 调用远程服务判断是否需要发送本次注册请求，这里的 forceRegister 入参默认为 true */
    if (forceRegister || needRegister(this.brokerConfig.getBrokerClusterName(),
        this.getBrokerAddr(),
        this.brokerConfig.getBrokerName(),
        this.brokerConfig.getBrokerId(),
        this.brokerConfig.getRegisterBrokerTimeoutMills())) {
        /* 判断通过，执行注册请求 */
        doRegisterBrokerAll(checkOrderConfig, oneway, topicConfigWrapper);
    }
}
```

BrokerOuterAPI：

```java
public List<Boolean> needRegister(
    final String clusterName,
    final String brokerAddr,
    final String brokerName,
    final long brokerId,
    final TopicConfigSerializeWrapper topicConfigWrapper,
    final int timeoutMills) {
    final List<Boolean> changedList = new CopyOnWriteArrayList<>();
    List<String> nameServerAddressList = this.remotingClient.getNameServerAddressList();
    // 遍历当前 Broker 连接的每个 NameServer 的地址
    if (nameServerAddressList != null && nameServerAddressList.size() > 0) {
            // 构建闭锁，控制每个 NameServer 全部检查完成之后再返回，这里的远程调用是采用异步的方式
        final CountDownLatch countDownLatch = new CountDownLatch(nameServerAddressList.size());
        for (final String namesrvAddr : nameServerAddressList) {
            brokerOuterExecutor.execute(new Runnable() {
                @Override
                public void run() {
                    try {
                        QueryDataVersionRequestHeader requestHeader = new QueryDataVersionRequestHeader();
                        requestHeader.setBrokerAddr(brokerAddr);
                        requestHeader.setBrokerId(brokerId);
                        requestHeader.setBrokerName(brokerName);
                        requestHeader.setClusterName(clusterName);
                        // 构建查询 NameServer 数据版本请求
                        RemotingCommand request = RemotingCommand.createRequestCommand(RequestCode.QUERY_DATA_VERSION, requestHeader);
                        request.setBody(topicConfigWrapper.getDataVersion().encode());
                        // 发送异步请求，查询 Broker 在 NameServer 保存的数据版本
                        RemotingCommand response = remotingClient.invokeSync(namesrvAddr, request, timeoutMills);
                        DataVersion nameServerDataVersion = null;
                        Boolean changed = false;
                        switch (response.getCode()) {
                            case ResponseCode.SUCCESS: {
                                QueryDataVersionResponseHeader queryDataVersionResponseHeader =
                                    (QueryDataVersionResponseHeader) response.decodeCommandCustomHeader(QueryDataVersionResponseHeader.class);
                                changed = queryDataVersionResponseHeader.getChanged();
                                byte[] body = response.getBody();
                                if (body != null) {
                                    nameServerDataVersion = DataVersion.decode(body, DataVersion.class);
                                    // 对比版本号，如果不一致，说明 Broker 信息发生了改变，需要执行本次注册请求
                                    if (!topicConfigWrapper.getDataVersion().equals(nameServerDataVersion)) {
                                        changed = true;
                                    }
                }
                if (changed == null || changed) {
                  changedList.add(Boolean.TRUE);
                }
              }
              default:
                break;
            }
            log.warn("Query data version from name server {} OK,changed {}, br
oker {},name server {}", namesrvAddr, changed, topicConfigWrapper.getDataVersion(), nameSe
rverDataVersion == null ? "" : nameServerDataVersion);
          } catch (Exception e) {
            changedList.add(Boolean.TRUE);
            log.error("Query data version from name server {}  Exception, {}",
namesrvAddr, e);
          } finally {
              // 每完成一个 NameServer 的检查，闭锁减1
            countDownLatch.countDown();
          }
        }
      });

    }
    try {
        // 全部完成后，返回结果
      countDownLatch.await(timeoutMills, TimeUnit.MILLISECONDS);
    } catch (InterruptedException e) {
      log.error("query dataversion from nameserver countDownLatch await Exception",
 e);
    }
  }
  return changedList;
}
```

我们来看 NameServer 对应 RequestCode.QUERY_DATA_VERSION 这个请求码的处理逻辑：

DefaultRequestProcessor：

```java
public RemotingCommand queryBrokerTopicConfig(ChannelHandlerContext ctx,
    RemotingCommand request) throws RemotingCommandException {
    final RemotingCommand response = RemotingCommand.createResponseCommand(QueryDataVersionResponseHeader.class);
    final QueryDataVersionResponseHeader responseHeader = (QueryDataVersionResponseHeader) response.readCustomHeader();
    final QueryDataVersionRequestHeader requestHeader =
        (QueryDataVersionRequestHeader) request.decodeCommandCustomHeader(QueryDataVersionRequestHeader.class);
    DataVersion dataVersion = DataVersion.decode(request.getBody(), DataVersion.class);
    // 判断数据版本是否发生改变，通过前面提到了 brokerLiveTable 中保存的数据与传入的数据进行对比
    Boolean changed = this.namesrvController.getRouteInfoManager().isBrokerTopicConfigChanged(requestHeader.getBrokerAddr(), dataVersion);
    if (!changed) {
            // 没有发生改变更新一下 lastUpdateTimestamp
        this.namesrvController.getRouteInfoManager().updateBrokerInfoUpdateTimestamp(requestHeader.getBrokerAddr());
    }
    // 同样从 brokerLiveTable 中拿到数据版本返回
    DataVersion nameSeverDataVersion = this.namesrvController.getRouteInfoManager().queryBrokerTopicConfig(requestHeader.getBrokerAddr());
    response.setCode(ResponseCode.SUCCESS);
    response.setRemark(null);
    if (nameSeverDataVersion != null) {
        response.setBody(nameSeverDataVersion.encode());
    }
    responseHeader.setChanged(changed);
    return response;
}
```

NameServer 对应 RequestCode.QUERY_DATA_VERSION 这个请求码的处理逻辑也非常简单，就是从 brokerLiveTable 中拿到对应 Broker 现有保存的数据与传入的数据进行对比。如果返回是需要执行本次的注册请求的话，就会进入下面的 doRegisterBrokerAll 方法：

BrokerController：

```java
private void doRegisterBrokerAll(boolean checkOrderConfig, boolean oneway,
    TopicConfigSerializeWrapper topicConfigWrapper) {
    /* 调用远程服务，向 NameServer 注册当前 Broker 信息 */
    List<RegisterBrokerResult> registerBrokerResultList = this.brokerOuterAPI.registerBrokerAll(
        this.brokerConfig.getBrokerClusterName(),
        this.getBrokerAddr(),
        this.brokerConfig.getBrokerName(),
        this.brokerConfig.getBrokerId(),
        this.getHAServerAddr(),
        topicConfigWrapper,
        this.filterServerManager.buildNewFilterServerList(),
        oneway,
        this.brokerConfig.getRegisterBrokerTimeoutMills(),
        this.brokerConfig.isCompressedRegister());
    if (registerBrokerResultList.size() > 0) {
            // 取出注册结果中的某一个，目的是拿到 master 节点的地址
        RegisterBrokerResult registerBrokerResult = registerBrokerResultList.get(0);
        if (registerBrokerResult != null) {
            if (this.updateMasterHAServerAddrPeriodically && registerBrokerResult.getHaServerAddr() != null) {
                this.messageStore.updateHaMasterAddress(registerBrokerResult.getHaServerAddr());
            }
            this.slaveSynchronize.setMasterAddr(registerBrokerResult.getMasterAddr());
            if (checkOrderConfig) {
                this.getTopicConfigManager().updateOrderTopicConfig(registerBrokerResult.getKvTable());
            }
        }
    }
}
```

BrokerOuterAPI：

```java
public List<RegisterBrokerResult> registerBrokerAll(
    final String clusterName,
    final String brokerAddr,
    final String brokerName,
    final long brokerId,
    final String haServerAddr,
    final TopicConfigSerializeWrapper topicConfigWrapper,
    final List<String> filterServerList,
    final boolean oneway,
    final int timeoutMills,
    final boolean compressed) {
    final List<RegisterBrokerResult> registerBrokerResultList = new CopyOnWriteArrayList<>();
    List<String> nameServerAddressList = this.remotingClient.getNameServerAddressList();
    if (nameServerAddressList != null && nameServerAddressList.size() > 0) {
        final RegisterBrokerRequestHeader requestHeader = new RegisterBrokerRequestHeader();
        requestHeader.setBrokerAddr(brokerAddr);
        requestHeader.setBrokerId(brokerId);
        requestHeader.setBrokerName(brokerName);
        requestHeader.setClusterName(clusterName);
        requestHeader.setHaServerAddr(haServerAddr);
        requestHeader.setCompressed(compressed);
        RegisterBrokerBody requestBody = new RegisterBrokerBody();
        requestBody.setTopicConfigSerializeWrapper(topicConfigWrapper);
        requestBody.setFilterServerList(filterServerList);
        final byte[] body = requestBody.encode(compressed);
        final int bodyCrc32 = UtilAll.crc32(body);
        requestHeader.setBodyCrc32(bodyCrc32);
        // 构建闭锁，控制每个 NameServer 全部注册完成之后再返回，这里的远程调用是采用异步的方式
        final CountDownLatch countDownLatch = new CountDownLatch(nameServerAddressList.size());
        for (final String namesrvAddr : nameServerAddressList) {
            brokerOuterExecutor.execute(new Runnable() {
                @Override
                public void run() {
                    try {
                        /* 调用远程服务注册 Broker */
                        RegisterBrokerResult result = registerBroker(namesrvAddr,oneway, timeoutMills,requestHeader,body);
                        if (result != null) {
                            registerBrokerResultList.add(result);
                        }
                        log.info("register broker[{}]to name server {} OK", brokerId, namesrvAddr);
                    } catch (Exception e) {
                        log.warn("registerBroker Exception, {}", namesrvAddr, e);
                    } finally {
                        countDownLatch.countDown();
                    }
                }
            });
        }
        try {
                // 全部完成后，返回结果
            countDownLatch.await(timeoutMills, TimeUnit.MILLISECONDS);
        } catch (InterruptedException e) {
        }
    }
    return registerBrokerResultList;
}

```

接下来我们来看 NameServer 对 REGISTER_BROKER 命令的处理。

DefaultRequestProcessor：

```java
public RemotingCommand registerBrokerWithFilterServer(ChannelHandlerContext ctx, RemotingCommand request)
    throws RemotingCommandException {
    final RemotingCommand response = RemotingCommand.createResponseCommand(RegisterBrokerResponseHeader.class);
    final RegisterBrokerResponseHeader responseHeader = (RegisterBrokerResponseHeader) response.readCustomHeader();
    final RegisterBrokerRequestHeader requestHeader =
        (RegisterBrokerRequestHeader) request.decodeCommandCustomHeader(RegisterBrokerRequestHeader.class);
    if (!checksum(ctx, request, requestHeader)) {
        response.setCode(ResponseCode.SYSTEM_ERROR);
        response.setRemark("crc32 not match");
        return response;
    }
    RegisterBrokerBody registerBrokerBody = new RegisterBrokerBody();
    if (request.getBody() != null) {
        try {
            registerBrokerBody = RegisterBrokerBody.decode(request.getBody(), requestHeader.isCompressed());
        } catch (Exception e) {
            throw new RemotingCommandException("Failed to decode RegisterBrokerBody", e);
        }
    } else {
        registerBrokerBody.getTopicConfigSerializeWrapper().getDataVersion().setCounter(new AtomicLong(0));
        registerBrokerBody.getTopicConfigSerializeWrapper().getDataVersion().setTimestamp(0);
    }
    /* 注册 Broker 信息到 RouteInfoManager 中 */
    RegisterBrokerResult result = this.namesrvController.getRouteInfoManager().registerBroker(
        requestHeader.getClusterName(),
        requestHeader.getBrokerAddr(),
        requestHeader.getBrokerName(),
        requestHeader.getBrokerId(),
        requestHeader.getHaServerAddr(),
        registerBrokerBody.getTopicConfigSerializeWrapper(),
        registerBrokerBody.getFilterServerList(),
        ctx.channel());
    responseHeader.setHaServerAddr(result.getHaServerAddr());
    responseHeader.setMasterAddr(result.getMasterAddr());
    byte[] jsonValue = this.namesrvController.getKvConfigManager().getKVListByNamespace(NamesrvUtil.NAMESPACE_ORDER_TOPIC_CONFIG);
    response.setBody(jsonValue);
    response.setCode(ResponseCode.SUCCESS);
    response.setRemark(null);
    return response;
}
```

RouteInfoManager：

```java
public RegisterBrokerResult registerBroker(
    final String clusterName,
    final String brokerAddr,
    final String brokerName,
    final long brokerId,
    final String haServerAddr,
    final TopicConfigSerializeWrapper topicConfigWrapper,
    final List<String> filterServerList,
    final Channel channel) {
    RegisterBrokerResult result = new RegisterBrokerResult();
    try {
        try {
            this.lock.writeLock().lockInterruptibly();
            Set<String> brokerNames = this.clusterAddrTable.get(clusterName);
            if (null == brokerNames) {
                brokerNames = new HashSet<String>();
                this.clusterAddrTable.put(clusterName, brokerNames);
            }
            // 将本次注册的 Broker 名称添加的集群中，Set 会去重
            brokerNames.add(brokerName);
            boolean registerFirst = false;
            BrokerData brokerData = this.brokerAddrTable.get(brokerName);
            if (null == brokerData) {
                registerFirst = true;
                brokerData = new BrokerData(clusterName, brokerName, new HashMap<Long, String>());
                // 第一次注册，初始化 BrokerData
                this.brokerAddrTable.put(brokerName, brokerData);
            }
            // 相同的 ip:port 对在 brokerAddrsMap 中只会存在一份
            Map<Long, String> brokerAddrsMap = brokerData.getBrokerAddrs();
            Iterator<Entry<Long, String>> it = brokerAddrsMap.entrySet().iterator();
            while (it.hasNext()) {
                Entry<Long, String> item = it.next();
                // 如果发现 ip:port 对一样但是 brokerId 不一样（一般是 Broker 主从切换会发生 brokerId 变化），则移除原来的映射存储从新存储
                if (null != brokerAddr && brokerAddr.equals(item.getValue()) && brokerId != item.getKey()) {
                    it.remove();
                }
            }
            String oldAddr = brokerData.getBrokerAddrs().put(brokerId, brokerAddr);
            registerFirst = registerFirst || (null == oldAddr);
            if (null != topicConfigWrapper
                && MixAll.MASTER_ID == brokerId) {
                // master 并且 topic 信息发生变化，则创建或更新 topic 路由元数据
                if (this.isBrokerTopicConfigChanged(brokerAddr, topicConfigWrapper.getDataVersion())
                    || registerFirst) {
                    ConcurrentMap<String, TopicConfig> tcTable =
                        topicConfigWrapper.getTopicConfigTable();
                    if (tcTable != null) {
                        for (Map.Entry<String, TopicConfig> entry : tcTable.entrySet()) {
                            this.createAndUpdateQueueData(brokerName, entry.getValue());
                        }
                    }
                }
            }
            // 更新 Broker 元数据
            BrokerLiveInfo prevBrokerLiveInfo = this.brokerLiveTable.put(brokerAddr,
                new BrokerLiveInfo(
                    System.currentTimeMillis(),
                    topicConfigWrapper.getDataVersion(),
                    channel,
                    haServerAddr));
            if (null == prevBrokerLiveInfo) {
                log.info("new broker registered, {} HAServer: {}", brokerAddr, haServerAdd
r);
            }
            // 更新 FilterServer 元数据
            if (filterServerList != null) {
                if (filterServerList.isEmpty()) {
                    this.filterServerTable.remove(brokerAddr);
                } else {
                    this.filterServerTable.put(brokerAddr, filterServerList);
                }
            }
            // 如果本次注册的 Broker 是 slave 节点，返回值中添加 master 节点的信息，上面 Broker
 对注册结果的处理逻辑中我们看到了这部分内容的使用
            if (MixAll.MASTER_ID != brokerId) {
                String masterAddr = brokerData.getBrokerAddrs().get(MixAll.MASTER_ID);
                if (masterAddr != null) {
                    BrokerLiveInfo brokerLiveInfo = this.brokerLiveTable.get(masterAddr);
                    if (brokerLiveInfo != null) {
                        result.setHaServerAddr(brokerLiveInfo.getHaServerAddr());
                        result.setMasterAddr(masterAddr);
                    }
                }
            }
        } finally {
            this.lock.writeLock().unlock();
        }
    } catch (Exception e) {
        log.error("registerBroker Exception", e);
    }
    return result;
}
```

路由信息删除

RocktMQ 有两个触发点来触发路由删除，Broker 在正常被关闭的情况下，会发送 UNREGISTER_BROKER 命令到 NameServer 移除元数据信息，另外 NameServer 会每个10s定时扫描 brokerLiveTable 检测上次心跳包与当前系统时间的时间差，如果时间差大于120s，则需要移除该 Broker 信息。

路由发现

RocketMQ 路由发现是非实时的，当 topic 路由出现变化后，NameServer 不主动推送给客户端，而是由客户端定时拉取 topic 最新的路由信息。在客户端启动时，会调用 MQClientInstance#startScheduledTask 启动更新 topic 路由信息的定时任务，这个方法我们在分析 Producer 启动流程时分析过，定时任务会向 NameServer 发送 GET_ROUTEINTO_BY_TOPIC 请求来获取最新的 topic 路由信息，然后更新内部消费者的订阅信息和生产者的发布信息。

总结

路由信息删除和路由发现同路由信息注册类似，这里就不用代码赘述了。获取 topic 路由信息的逻辑很简单，就是从 NameServer 管理的路由元数据中拿到对应 topic 的队列信息和 Broker 信息返回。