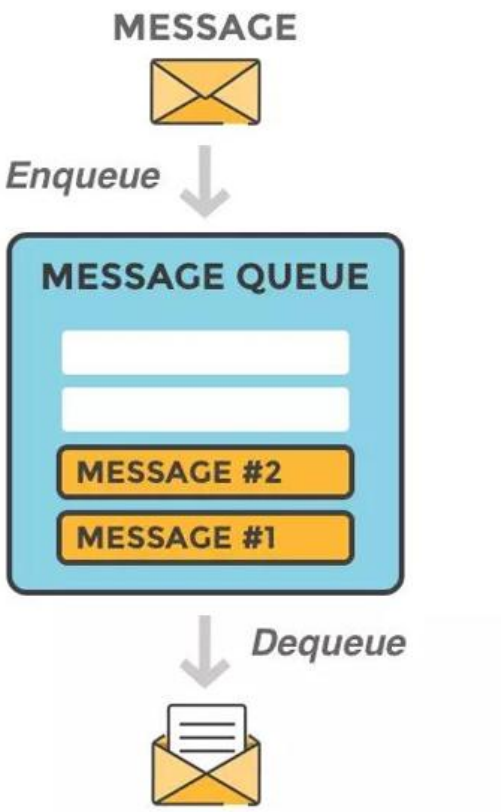
深入了解 ActiveMQ

作者：jianzh5
原文链接：https://ld246.com/article/1599194307526
来源网站：链滴

认识MQ(Message Queue)

什么是消息队列



首先我们先从以下几个维度来认识一下消息队列：

- 消息队列：一般我们会简称它为MQ(MessageQueue)
- 消息(Message):传输的数据。
- 队列(Queue):队列是一种先进先出的数据结构。
- 消息队列从字面的含义来看就是一个存放消息的容器。
- 消息队列可以简单理解为：把要传输的数据放在队列中。
- 把数据放到消息队列叫做生产者。
- 从消息队列里边取数据叫做消费者。

为什么需要消息队列

使用消息队列主要是基于以下三个主要场景：

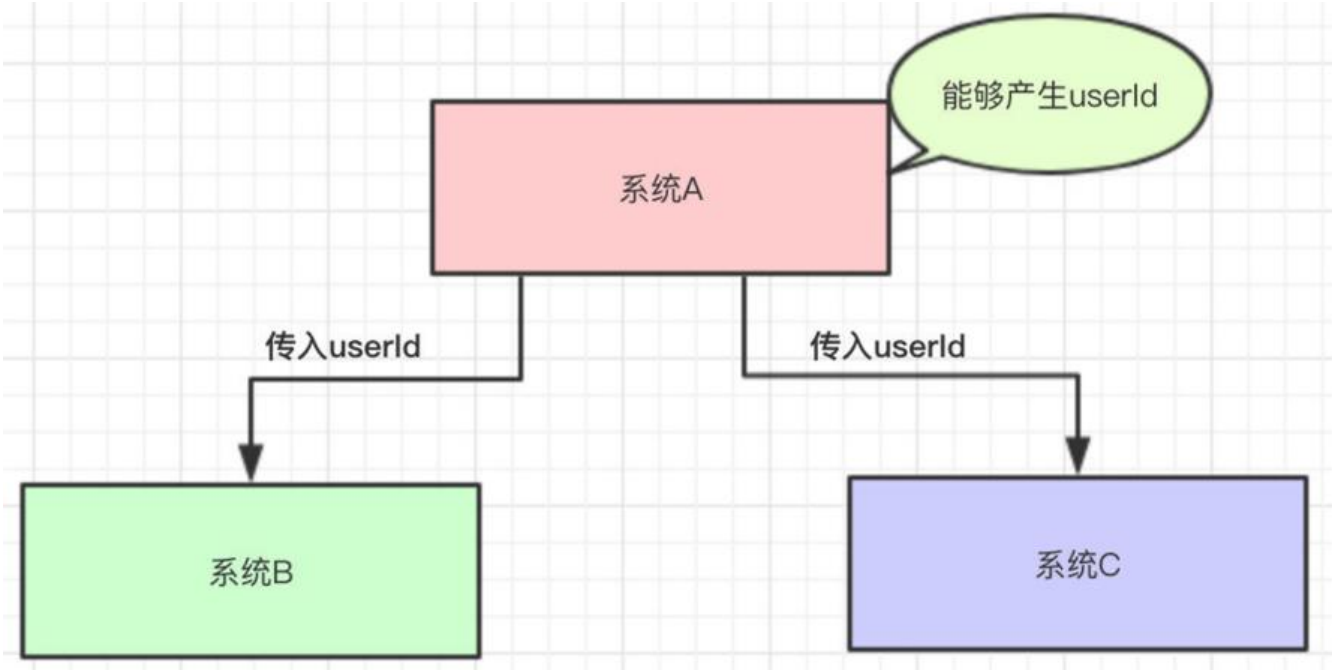
- 解耦
- 异步
- 削峰/限流

下面我们分场景来描述下使用消息队列带来的好处

解耦

假设我们有一个用户系统A，用户系统A可以产生一个userId。

然后，现在有系统B和系统C都需要这个userId去做相关的操作。



伪码大致如下：

```
java
public class SystemA {
    // 系统B和系统C的依赖
    SystemB systemB = new SystemB();
    SystemC systemC = new SystemC();
    // 系统A独有的数据userId
    private String userId = "activeMq-1234567890";
    public void doSomething() {
        // 系统B和系统C都需要拿着系统A的userId去操作其他的事
        systemB.SystemBNeed2do(userId);
        systemC.SystemCNeed2do(userId);
    }
}
```

这样类似的业务场景大家是不是很熟悉，大家是不是这样写很合情合理，也很简单。

某一天，系统B的负责人告诉系统A的负责人，现在系统B的SystemBNeed2do(String userId)这个接不再使用了，让系统A别去调它了。于是，系统A的负责人说"好的，那我就不调用你了。"，于是就把用系统B接口的代码给删掉了。代码变成这样了：

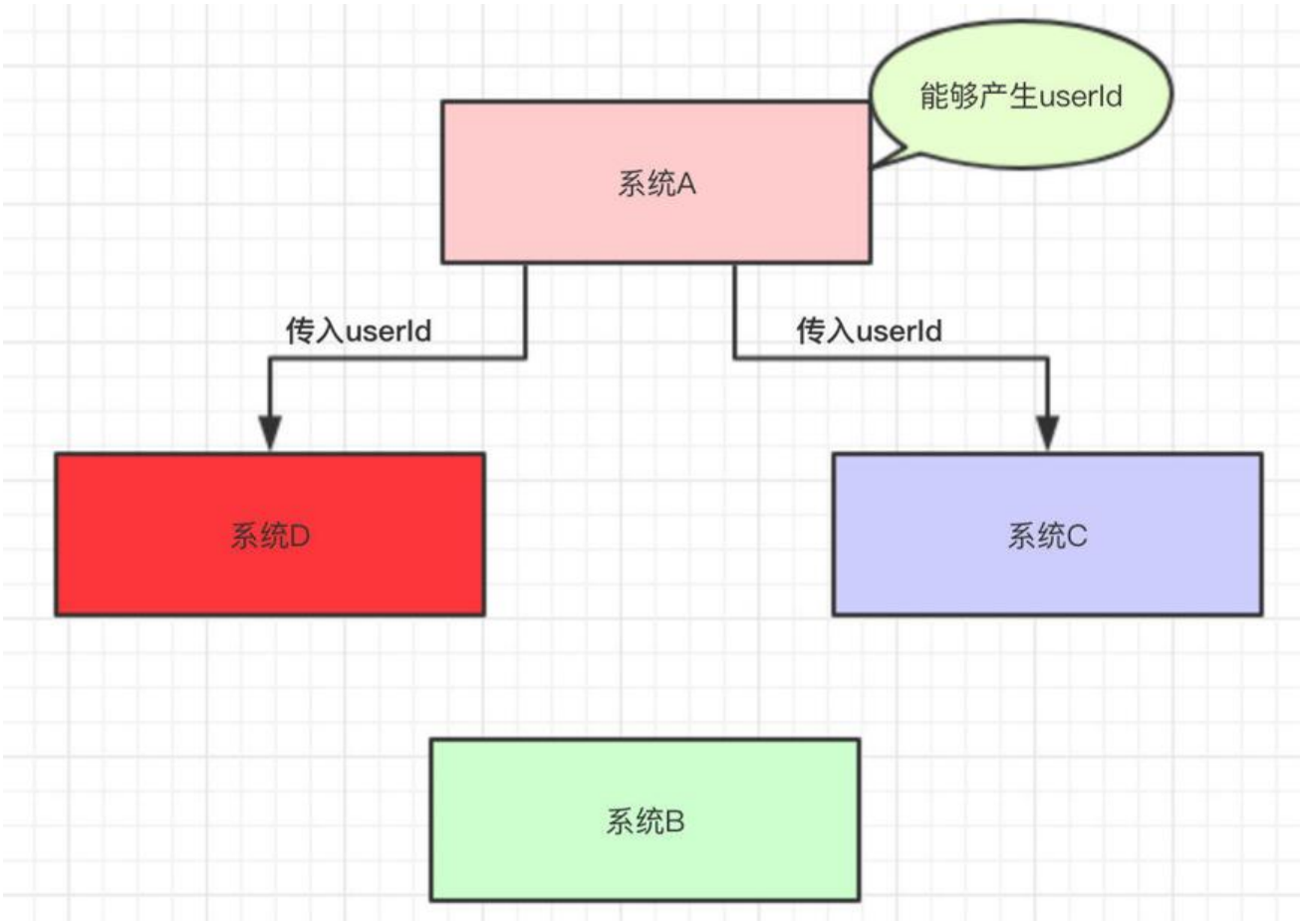
```
java
public void doSomething() {
  // 系统A不再调用系统B的接口了
  //systemB.SystemBNeed2do(userId);
  systemC.SystemCNeed2do(userId);
}
```

由于业务需要，系统D说也需要用到系统A的userId，于是代码改成了这样：

```
java
public void doSomething() {
        // 已经不再需要系统B的依赖了
        //systemB.SystemBNeed2do(userId);
        // 系统C和系统D都需要拿着系统A的userId去操作其他的事
        systemC.SystemCNeed2do(userId);
        systemD.SystemDNeed2do(userId);

}
```

当前系统A、B、C、D系统的交互是这样子的。

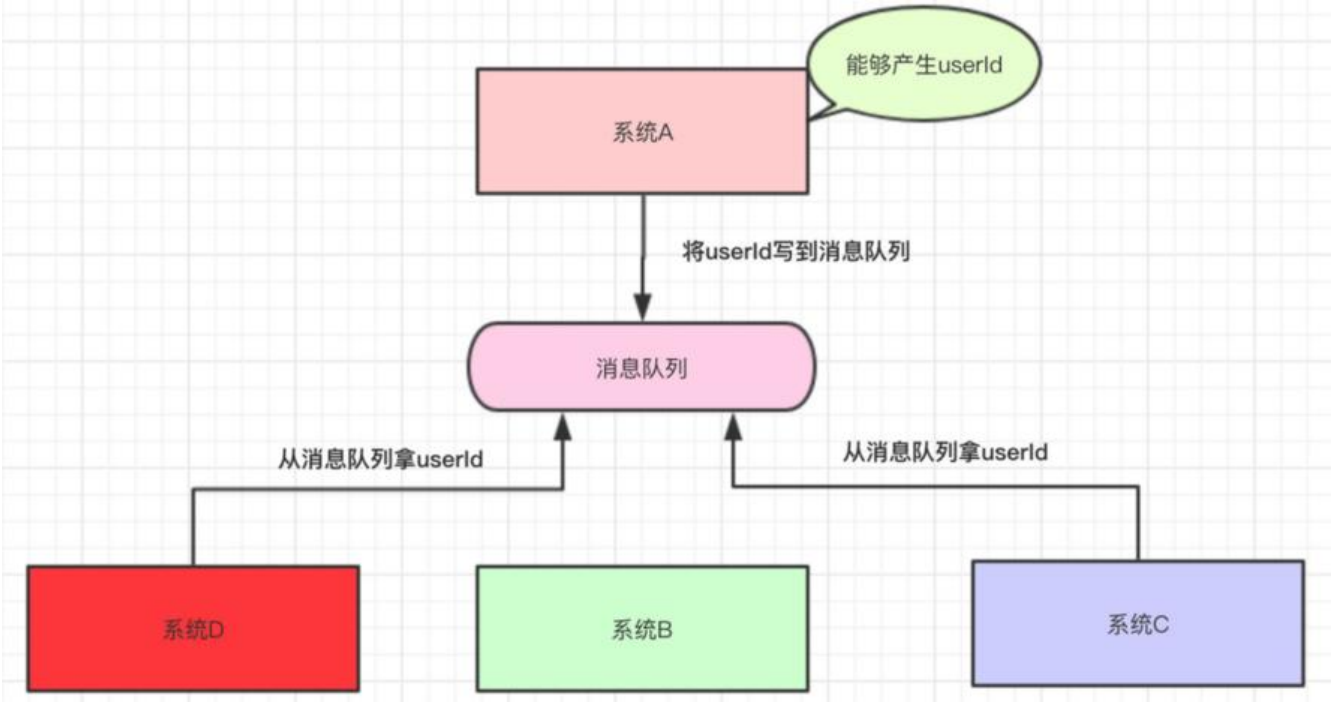


随着业务需求的变化，代码也要一遍一遍的修改。

还会存在另外一个问题，调用系统C的时候，如果系统C挂了，系统A还要想办法处理。如果调用系统时，由于网络延迟，请求超时了，那系统A是反馈fail还是重试?

那么怎么去解决这样的现状呢，如何从频繁的修改代码中解脱呢?

这时候我们就引入一层消息队列中间件，交互图如下：



将系统A产生的userId写到消息队列中，系统C和系统D从消息队列中拿数据。

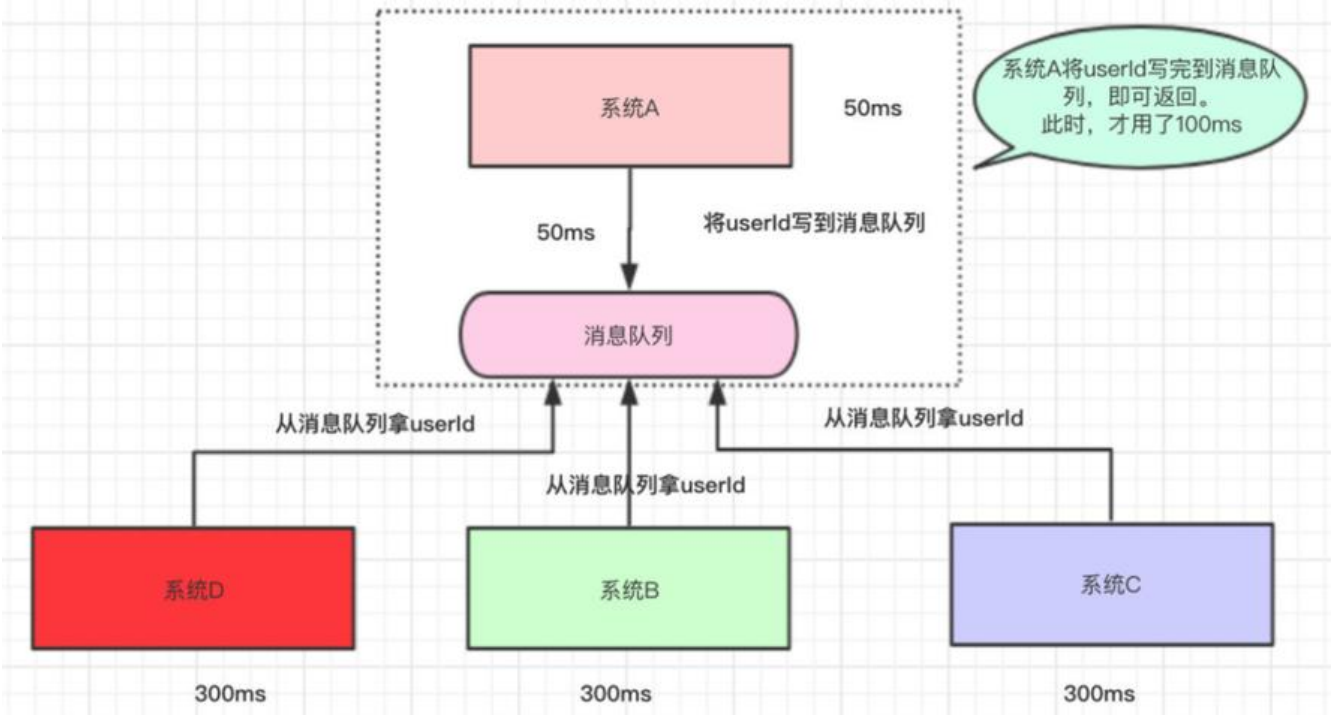
这样有什么好处?

- 系统A只负责把数据写到队列中，谁想要或不想要这个数据(消息)，系统A一点都不关心。
- 即便现在系统D不想要userId这个数据了，系统B又突然想要userId这个数据了，都跟系统A无关，统A一点代码都不用改。
- 系统D拿userId不再经过系统A，而是从消息队列里边拿。系统D即便挂了或者请求超时，都跟系统无关，

只跟消息队列有关。这样一来，系统A与系统B、C、D都解耦了。

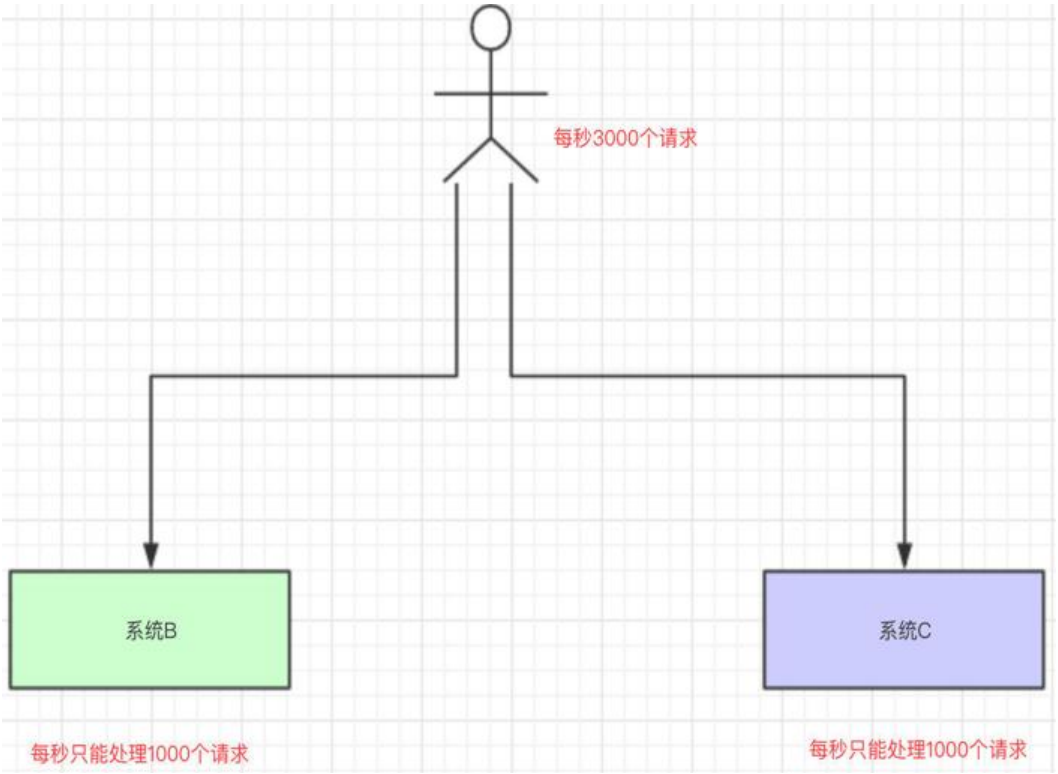
异步

系统A做的是主要的业务，而系统B、C、D是非主要的业务。比如系统A处理的是订单下单，而系统B订单下单成功了，那发送一条短信告诉具体的用户此订单已成功，而系统C和系统D也是处理一些小而已。那么此时，为了提高用户体验和吞吐量，其实可以异步地调用系统B、C、D的接口。

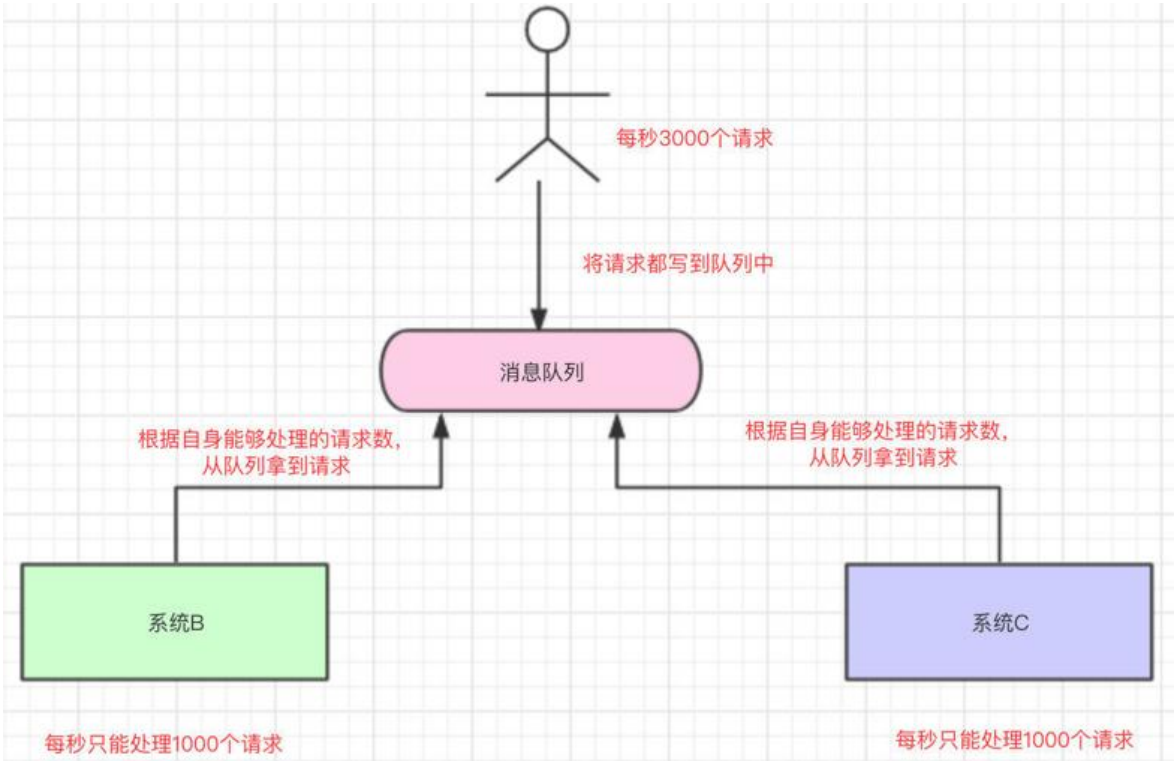


削峰/限流

我们再来一个场景，现在我们每个月要搞一次大促，大促期间的并发可能会很高的，比如每秒3000个求。假设我们现在有两台机器处理请求，并且每台机器只能每次处理1000个请求。



系统B和系统C根据自己的能够处理的请求数去消息队列中拿数据，这样即便有每秒有8000个请求，只是把请求放在消息队列中，去拿消息队列的消息由系统自己去控制，这样就不会把整个系统给搞崩。



什么是JMS MQ

全称：Java MessageService 中文：Java 消息服务。

JMS 是 Java 的一套 API 标准，最初的目的是为了使应用程序能够访问现有的MOM 系 统（MOM 是 MessageOriented Middleware 的英文缩写，指的是利用高效可靠的消息传递机制进行平台无关的据交流，并基于数据通信来进行分布式系统的集成。） 后来被许多现有的 MOM 供应商采用，并实为MOM 系统。

常见 MOM 系统包括 Apache的 ActiveMQ、阿里巴巴的 RocketMQ、IBM 的 MQSeries、Microsoft 的 MSMQ、BEA 的 RabbitMQ 等。 （并非全部的 MOM 系统都遵循JMS 规范）】

基于 JMS 实现的 MOM，又被称为JMSProvider。

JMS中的一些概念

Broker

消息服务器，作为server提供消息核心服务

Provider 生产者

消息生产者是由会话创建的一个对象，用于把消息发动到一个目的地

Consumer 消费者

消息消费者是由会话创建的一个对象，它用于接收发送到目的地的消息。消息的消费可以采用以下两方法：

同步消费。通过调用消费者的receive方法从目的地中显式提取消息。receive方法可以一直阻塞到消到达。

原文链接：深入了解 ActiveMQ

异步消费。客户可以为消费者注册一个消息监听器，以定义在消息到达时所采取的动作。

P2P 点对点消息模型

消息生产者生产消息发送到queue 中，然后消息消费者从queue 中取出并且消费消息。

消息被消费以后，queue 中不再有存储，所以消息消费者不可能消费到已经被消费的消息。 Queue 持存在多个消费者，但是对一个消息而言，只会有一个消费者可以消费、其它的则不能消费此消息了当消费者不存在时，消息会一直保存，直到有消费消费。

Pub/Sub 发布订阅消息模型

消息生产者（发布）将消息发布到topic 中，同时有多个消息消费者（订阅）消费该消息。 和点对点式不同，发布到 topic 的消息会被所有订阅者消费。当生产者发布消息，不管是否有消费者。都不会存消息一定要先有消息的消费者，后有消息的生产者。

P2P vs Pub/Sub

	Topic	Queue
	Publish Subscribe messaging 发布 订阅消息	Point-to-Point 点对点
有无状态	topic 数据默认不落地，是无状态的。	Queue 数据默认会在 mq 服 务器上以文件形式保存，比如 Active MQ 一 般 保 存 在 $AMQ_HOME\data\kahadb 下 面。也可以配置成 DB 存储
完整性保障	并不保证 publisher 发布的每条数 据，Subscriber 都能接受到。	Queue 保证每条数据都能被 receiver 接收。消息不超时。
消息是否会丢失	一般来说 publisher 发布消息到某 一个 topic 时，只有正在监听该 topic 地址的 sub 能够接收到消息；如果没 有 sub 在监听，该 topic 就丢失了。	Sender 发 送 消 息 到 目 标 Queue， receiver 可以异步接收这 个 Queue 上的消息。Queue 上的 消息如果暂时没有 receiver 来 取，也不会丢失。前提是消息不 超时。
消息发布接 收策略	一对多的消息发布接收策略，监听同一个 topic地址的多个sub都能收 到 publisher 发送的消息。Sub 接收完通知 mq 服务器	一对一的消息发布接收策 略，一个 sender 发送的消息，只 能有一个 receiver 接收。receiver 接收完后，通知 mq 服务器已接 收，mq 服务器对 queue 里的消息采取删除或其他操作。

Queue

队列存储，常用与点对点消息模型

默认只能由唯一的一个消费者处理。一旦处理消息删除。

Topic

主题存储，用于订阅/发布消息模型

主题中的消息，会发送给所有的消费者同时处理。只有在消息可以重复处理的业务场景中可使用。

ConnectionFactory

连接工厂，jms中用它创建连接

连接工厂是客户用来创建连接的对象，例如ActiveMQ提供的ActiveMQConnectionFactory。

Connection

JMS Connection封装了客户与JMS提供者之间的一个虚拟的连接。

Destination 消息的目的地

目的地是客户用来指定它生产的消息的目标和它消费的消息的来源的对象。

订阅一个主题的消费者只能消费自它订阅之后发布的消息。JMS规范允许客户创建持久订阅，这在一程度上放松了时间上的相关性要求。

持久订阅允许消费者消费它在未处于激活状态时发送的消息。 在点对点消息传递域中，目的地被成为列（queue）；在发布/订阅消息传递域中，目的地被成为主题（topic）。

Session

JMS Session是生产和消费消息的一个单线程上下文。会话用于创建消息生产者（producer）、消息费者（consumer）和消息（message）等。会话提供了一个事务性的上下文，在这个上下文中，一发送和接收被组合到了一个原子操作中。

消息可靠性机制

确认 JMS消息

只有在被确认之后，才认为已经被成功地消费了。消息的成功消费通常包含三个阶段：客户接收消息客户处理消息和消息被确认。

在事务性会话中，当一个事务被提交的时候，确认自动发生。

在非事务性会话中，消息何时被确认取决于创建会话时的应答模式（acknowledgement mode）。参数有以下三个可选值：

Session.AUTO_ACKNOWLEDGE。当客户成功的从receive方法返回的时候，或者从MessageListener.onMessage方法成功返回的时候，会话自动确认客户收到的消息。

Session.CLIENT_ACKNOWLEDGE。客户通过消息的acknowledge方法确认消息。需要注意的是在这种模式中，确认是在会话层上进行：确认一个被消费的消息将自动确认所有已被会话消费的消息例如，如果一个消息消费者消费了10个消息，然后确认第5个消息，那么所有10个消息都被确认。

Session.DUPS_ACKNOWLEDGE。该选择只是会话迟钝的确认消息的提交。如果JMS Provider失，那么可能会导致一些重复的消息。如果是重复的消息，那么JMS Provider必须把消息头的JMSRedelivered字段设置为true。

持久性

JMS 支持以下两种消息提交模式：

PERSISTENT。指示JMSProvider持久保存消息，以保证消息不会因为JMS Provider的失败而丢失。

NON_PERSISTENT。不要求JMS Provider持久保存消息。

优先级

可以使用消息优先级来指示JMS Provider首先提交紧急的消息。优先级分10个级别，从0（最低）到（最高）。如果不指定优先级，默认级别是4。**需要注意的是，JMSProvider并不一定保证按照优先的顺序提交消息。**

消息过期

可以设置消息在一定时间后过期，默认是永不过期

临时目的地

可以通过会话上的createTemporaryQueue方法和createTemporaryTopic方法来创建临时目的地。们的存在时间只限于创建它们的连接所保持的时间。只有创建该临时目的地的连接上的消息消费者才够从临时目的地中提取消息。

持久订阅

首先消息生产者必须使用PERSISTENT提交消息。客户可以通过会话上的createDurableSubscriber法来创建一个持久订阅，该方法的第一个参数必须是一个topic，第二个参数是订阅的名称。

JMS Provider会存储发布到持久订阅对应的topic上的消息。如果最初创建持久订阅的客户或者任何它客户使用相同的连接工厂和连接的客户ID、相同的主题和相同的订阅名再次调用会话上的createDurbleSubscriber方法，那么该持久订阅就会被激活。

JMS Provider会象客户发送客户处于非激活状态时所发布的消息。

持久订阅在某个时刻只能有一个激活的订阅者。持久订阅在创建之后会一直保留，直到应用程序调用话上的unsubscribe方法。

本地事务

在一个JMS客户端，可以使用本地事务来组合消息的发送和接收。JMS Session接口提供了commit和ollback方法。事务提交意味着生产的所有消息被发送，消费的所有消息被确认；事务回滚意味着生的所有消息被销毁，消费的所有消息被恢复并重新提交，除非它们已经过期。

事务性的会话总是牵涉到事务处理中，commit或rollback方法一旦被调用，一个事务就结束了，而一个事务被开始。关闭事务性会话将回滚其中的事务。

需要注意的是，如果使用请求/回复机制，即发送一个消息，同时希望在同一个事务中等待接收该消的回复，那么程序将被挂起，因为知道事务提交，发送操作才会真正执行。

需要注意的还有一个，消息的生产和消费不能包含在同一个事务中。

ActiveMQ

存储

ActiveMQ支持很多种存储方式，常见的有 KahaDB存储，AMQ存储，JDBC存储，LevelDB存储，Mmory

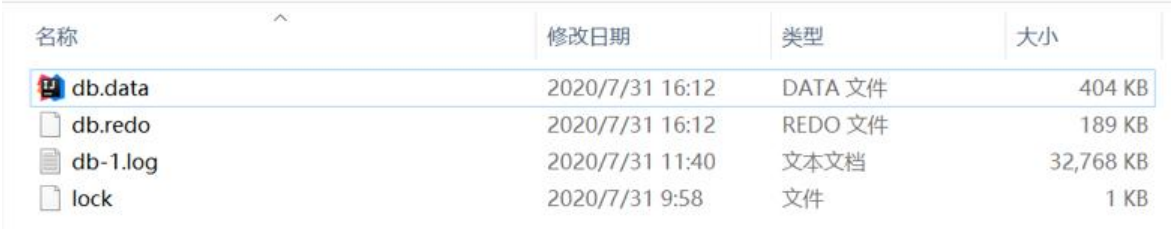
消息存储。 我们重点介绍一下KahaDB和JDBC存储方式。

KahaDB存储

KahaDB是默认的持久化策略，所有消息顺序添加到一个日志文件中，同时另外有一个索引文件记录

向这些日志的存储地址，还有一个事务日志用于消息回复操作。是一个专门针对消息持久化的解决方案它对典型的消息使用模式进行了优化。

在data/kahadb这个目录下，会生成四个文件，来完成消息持久化 db.data 它是消息的索引文件，本上是B-Tree（B树），使用B-Tree作为索引指向db-*.log里面存储的消息 db.redo 用来进行消息恢复 db-.log 存储消息内容。



名称	修改日期	类型	大小
db.data	2020/7/31 16:12	DATA 文件	404 KB
db.redo	2020/7/31 16:12	REDO 文件	189 KB
db-1.log	2020/7/31 11:40	文本文档	32,768 KB
lock	2020/7/31 9:58	文件	1 KB

新的数据以APPEND的方式追加到日志文件末尾。属于顺序写入，因此消息存储是比较 快的。默认是2M，达到阀值会自动递增 lock文件 锁，写入当前获得kahadb读写权限的broker ，用于在集群环境的竞争处理。

KahaDB有如下几个特性：

- 日志形式存储消息；
- 消息索引以 B-Tree 结构存储，可以快速更新；
- 完全支持 JMS 事务；
- 支持多种恢复机制kahadb 可以限制每个数据文件的大小。不代表总计数据容量。

配置方式如下：

```
<persistenceAdapter>
   <kahaDB directory="${activemq.data}/kahadb"/>
</persistenceAdapter>
```

JDBC 存储

支持通过 JDBC 将消息存储到关系数据库，性能上不如文件存储，能通过关系型数据库查询到消息的息。

MQ 支持的数据库：Apache Derby、MySQL、PostgreSQL、Oracle、SQLServer、Sybase、Infor ix、MaxDB。使用JDBC存储需要用到下面三张数据表。

activemq_acks：用于存储订阅关系。如果是持久化Topic，订阅者和服务器的订阅关系在这个表保 。 主要的数据库字段如下：

- container：消息的destination
- sub_dest：如果是使用static集群，这个字段会有集群其他系统的信息
- client_id：每个订阅者都必须有一个唯一的客户端id用以区分
- sub_name：订阅者名称
- selector：选择器，可以选择只消费满足条件的消息。条件可以用自定义属性实现，可支持多属性ad和or操作
- last_acked_id：记录消费过的消息的id。

activemq_lock：在集群环境中才有用，只有一个Broker可以获得消息，称为Master Broker，其他只能作为备份等待Master Broker不可用，才可能成为下一个Master Broker。这个表用于记录哪个Br ker是当前的Master Broker。

activemq_msgs：用于存储消息，Queue和Topic都存储在这个表中。 主要的数据库字段如下

- id：自增的数据库主键
- container：消息的destination
- msgid_prod：消息发送者客户端的主键
- msg_seq：是发送消息的顺序，msgid_prod+msg_seq可以组成jms的messageid
- expiration：消息的过期时间，存储的是从1970-01-01到现在的毫秒数
- msg：消息本体的java序列化对象的二进制数据
- priority：优先级，从0-9，数值越大优先级越高
- xid:topic

配置方式如下：

1. 配置数据源 conf/acticvemq.xml 文件：

```
<bean id="mysql-ds" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="
lose">
  <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
  <property name="url" value="jdbc:mysql://localhost:3306/activemq?relaxAutoCommit=tru
"/>
  <property name="username" value="root"/>
  <property name="password" value="111111"/>
  <property name="maxActive" value="200"/>
  <property name="poolPreparedStatements" value="true"/>
</bean>
```

2. 配置 broke 中的 persistenceAdapter

dataSource 指定持久化数据库的 bean，createTablesOnStartup 是否在启动的时候创建数据表，认值是 true，这样每次启动都会去创建数据表了，一般是第一次启动的时候设置为 true，之后改成 fa se。

```
<persistenceAdapter>
  <jdbcPersistenceAdapter dataSource="#mysql-ds"  createTablesOnStartup="false"/>
</persistenceAdapter>
```

协议

ActiveMQ支持的client-broker通讯协议有：TCP、NIO、UDP、SSL、Http(s)、VM。

Transmission Control Protocol (TCP)

这是默认的Broker配置，TCP的Client监听端口是61616。

在网络传输数据前，必须要序列化数据，消息是通过一个叫wire protocol的来序列化成字节流。默认况下，ActiveMQ把wire protocol叫做OpenWire，它的目的是促使网络上的效率和数据快速交互。

TCP连接的URI形式：tcp://hostname:port?key=value&key=value

TCP传输的优点： (1)TCP协议传输可靠性高，稳定性强 (2)高效性：字节流方式传递，效率很高 (3)有性、可用性：应用广泛，支持任何平台

New I/O API Protocol（NIO)

NIO协议和TCP协议类似，但NIO更侧重于底层的访问操作。它允许开发人员对同一资源可有更多的clint调用和服务端有更多的负载。

适合使用NIO协议的场景：

(1)可能有大量的Client去链接到Broker上一般情况下，大量的Client去链接Broker是被操作系统的线数所限制的。因此，NIO的实现比TCP需要更少的线程去运行，所以建议使用NIO协议

(2)可能对于Broker有一个很迟钝的网络传输NIO比TCP提供更好的性能

NIO连接的URI形式：nio://hostname:port?key=value

Transport Connector配置示例：

```
<transportConnectors>
    <transportConnector
        name="nio"
        uri="nio://localhost:61618?trace=true" />
</transportConnectors>
```

User Datagram Protocol（UDP)

UDP和TCP的区别

(1)TCP是一个原始流的传递协议，意味着数据包是有保证的，换句话说，数据包是不会被复制和丢失。UDP，另一方面，它是不会保证数据包的传递的

(2)TCP也是一个稳定可靠的数据包传递协议，意味着数据在传递的过程中不会被丢失。这样确保了在送和接收之间能够可靠的传递。相反，UDP仅仅是一个链接协议，所以它没有可靠性之说

从上面可以得出：TCP是被用在稳定可靠的场景中使用的；UDP通常用在快速数据传递和不怕数据丢的场景中，还有ActiveMQ通过防火墙时，只能用UDP

UDP连接的URI形式：udp://hostname:port?key=value

Transport Connector配置示例：

```
<transportConnectors>
  <transportConnector
    name="udp"
    uri="udp://localhost:61618?trace=true" />
</transportConnectors>
```

Active MQ的安全机制

web控制台安全

修改jetty-realm.properties

```
# username: password [,rolename ...]（用户名： 密码 角色）
```

注意：配置需重启ActiveMQ才会生效

消息安全机制

修改activemq.xml 在<broker></broker> 中添加如下代码：

```
<plugins>
    <simpleAuthenticationPlugin>
      <users>
        <authenticationUser username="admin" password="admin" groups="admins,publis
ers,consumers"/>
        <authenticationUser username="publisher" password="publisher"  groups="publish
rs,consumers"/>
        <authenticationUser username="consumer" password="consumer" groups="consu
ers"/>
        <authenticationUser username="guest" password="guest"  groups="guests"/>
      </users>
    </simpleAuthenticationPlugin>
 </plugins>
```

ActiveMQ 使用

在java中使用ActiveMQ只需要引入相关依赖

```
<dependency>
   <groupId>org.apache.activemq</groupId>
   <artifactId>activemq-all</artifactId>
   <version>5.15.11</version>
</dependency>
```

编写生产者

```
public class Sender {
   public static void main(String[] args) throws JMSException {
   // 1. 建立工厂对象，
   ActiveMQConnectionFactory acf = new ActiveMQConnectionFactory(ActiveMQConnection
actory.DEFAULT_USER,ActiveMQConnectionFactory.DEFAULT_PASSWORD,"tcp://localhost:61
18");
   //2 从工厂里拿一个连接
   Connection connection = acf.createConnection();
   connection.start();
   //3 从连接中获取Session(会话)
   Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
   //4 从会话中获取目的地(Destination)消费者会从这个目的地取消息
   Queue queue = session.createQueue("mq.test");
   //5 从会话中创建消息提供者
   MessageProducer producer = session.createProducer(queue);
   //6 从会话中创建文本消息(也可以创建其它类型的消息体)
   TextMessage message = session.createTextMessage("msg: hello world");
   //7 通过消息提供者发送消息到ActiveMQ
   producer.send(message);
   //8 关闭连接
   connection.close();
```

```
    }
}
```

编写消费者

```
public class Receiver {
    public static void main(String[] args) throws JMSException {
    // 1. 建立工厂对象，
    ActiveMQConnectionFactory acf = new ActiveMQConnectionFactory(ActiveMQConnection
actory.DEFAULT_USER,ActiveMQConnectionFactory.DEFAULT_PASSWORD,"tcp://localhost:61
18");
    //2 从工厂里拿一个连接
    Connection connection = acf.createConnection();
    connection.start();
    //3 从连接中获取Session(会话)
    Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
    //4 从会话中获取目的地(Destination)消费者会从这个目的地取消息
    Queue queue = session.createQueue("mq.test");
    //5 从会话中创建消息消费者
    MessageConsumer consumer = session.createConsumer(queue);
    while (true){
        //6 消费者接收消息
        Message msg = consumer.receive();
        TextMessage textMessage = (TextMessage) msg;
        System.out.println("text:"+textMessage.getText());
        }
    }
}
```

常用API及特性

● 事务消息

```
Session session = connection.createSession(true, Session.AUTO_ACKNOWLEDGE);
```

提交事务：session.commit();

回滚事务：session.rollback();

开启事务后，只有事务commit成功，消息才会发送到MQ中

● 持久化

默认持久化是开启的；

开启非持久化示例代码：

```
producer.setDeliveryMode(DeliveryMode.NON_PERSISTENT)
```

● 设置消息优先级

```
producer.setPriority();
```

● 设置消息超时/过期时间

```
producer.setTimeToLive
```

设置了消息超时的消息，消费端在超时后无法在消费到此消息。

● 死信

此类消息会进入到ActiveMQ.DLQ队列且不会自动清除，称为死信，有消息堆积的风险。

● 签收模式

签收代表接收端的session已收到消息的一次确认，反馈给broker

如果session带有事务，并且事务成功提交，则消息被自动签收。如果事务回滚，则消息会被再次传
。

消息事务是在生产者producer到broker或broker到consumer过程中同一个session中发生的，保证条消息在发送过程中的原子性。在支持事务的session中，producer发送message时在message中带transactionID。broker收到message后判断是否有transactionID，如果有就把message保存在transction store中，等待commit或者rollback消息。

ActiveMQ支持自动签收与手动签收

Session.AUTO_ACKNOWLEDGE

当客户端从receiver或onMessage成功返回时，Session自动签收客户端的这条消息的收条。

Session.CLIENT_ACKNOWLEDGE

客户端通过调用消息(Message)的acknowledge方法签收消息。在这种情况下，签收发生在Session面：签收一个已经消费的消息会自动地签收这个Session所有已消费的收条。

Session.DUPS_OK_ACKNOWLEDGE

Session不必确保对传送消息的签收，这个模式可能会引起消息的重复，但是降低了Session的开销，以只有客户端能容忍重复的消息，才可使用。

● 独占消费者

```
Queue queue = session.createQueue("xxoo?consumer.exclusive=true");
```

● 发送异步消息

```
ActiveMQConnectionFactory connectionFactory = new ActiveMQConnectionFactory(
                "admin",
                "admin",
                "tcp://localhost:61616"
                );
// 2.获取一个向ActiveMQ的连接
connectionFactory.setUseAsyncSend(true);
ActiveMQConnection connection = (ActiveMQConnection)connectionFactory.createConnecti
n();
connection.setUseAsyncSend(true);
```

● 消息堆积

producer每发送一个消息，统计一下发送的字节数，当字节数达到ProducerWindowSize值时，需等待broker的确认，才能继续发送。

brokerUrl中设置: tcp://localhost:61616?jms.producerWindowSize=1048576

destinationUri中设置: myQueue?producer.windowSize=1048576

● 延迟消息投递

首先在配置文件中开启延迟和调度

```
<broker xmlns="http://activemq.apache.org/schema/core" brokerName="localhost" dataDire
tory="${activemq.data}" schedulerSupport="true">
```

延迟发送示例代码：

```
message.setLongProperty(ScheduledMessage.AMQ_SCHEDULED_DELAY,10*1000);
```

- 创建监听器

```
ActiveMQConnectionFactory acf = new ActiveMQConnectionFactory(ActiveMQConnectionFac
ory.DEFAULT_USER,
        ActiveMQConnectionFactory.DEFAULT_PASSWORD,
        "tcp://localhost:61618");
//2 从工厂里拿一个连接
Connection connection = acf.createConnection();
connection.start();
//3 从连接中获取Session(会话)
Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
//4 从会话中获取目的地(Destination)消费者会从这个目的地取消息
Queue queue = session.createQueue("mq.test");
//5 从会话中创建消息消费者
MessageConsumer consumer = session.createConsumer(queue);
MyListener myListener = new MyListener();
MessageListener listener = myListener::receiveMessage;
consumer.setMessageListener(listener);
```

SpringBoot整合ActiveMQ

- 添加依赖

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-activemq</artifactId>
</dependency>
```

- 配置文件

```
server:
  port: 80
spring:
  activemq:
    broker-url: tcp://localhost:61618
    user: admin
    password: admin
    pool:
      enabled: true
      #连接池最大连接数
      max-connections: 5
      #空闲的连接过期时间，默认为30秒
      idle-timeout: 0
    packages:
      trust-all: true
  jms:
    pub-sub-domain: true
```

- 配置类

```
@Configuration
```

```
@EnableJms
public class ActiveMqConfig {
// topic模式的ListenerContainer
 @Bean
public JmsListenerContainerFactory<?> jmsListenerContainerTopic(ConnectionFactory active
QConnectionFactory) {
        DefaultJmsListenerContainerFactory bean = new DefaultJmsListenerContainerFactory()

        bean.setPubSubDomain(true);
        bean.setConnectionFactory(activeMQConnectionFactory);
        return bean;
    }
// queue模式的ListenerContainer
@Bean
public JmsListenerContainerFactory<?> jmsListenerContainerQueue(ConnectionFactory activ
MQConnectionFactory) {
        DefaultJmsListenerContainerFactory bean = new DefaultJmsListenerContainerFactory()

        bean.setConnectionFactory(activeMQConnectionFactory);
        return bean;
    }
}
```

- 编写生产者

```
@Service
public class MqProducerService {
   @Autowired
   private JmsMessagingTemplate jmsMessagingTemplate;

   public void sendStringQueue(String destination, String msg) {
      System.out.println("send...");
      ActiveMQQueue queue = new ActiveMQQueue(destination);
      jmsMessagingTemplate.afterPropertiesSet();
      ConnectionFactory factory = jmsMessagingTemplate.getConnectionFactory();
      try {
         Connection connection = factory.createConnection();
         connection.start();

         Session session = connection.createSession(true, Session.AUTO_ACKNOWLEDGE);
         Queue queue2 = session.createQueue(destination);

         MessageProducer producer = session.createProducer(queue2);

         TextMessage message = session.createTextMessage("hahaha");


         producer.send(message);
      } catch (JMSException e) {
         // TODO Auto-generated catch block
         e.printStackTrace();
      }
```

```
        jmsMessagingTemplate.convertAndSend(queue, msg);
    }
    public void sendStringQueueList(String destination, String msg) {
        System.out.println("xxooq");
        ArrayList<String> list = new ArrayList<>();
        list.add("1");
        list.add("2");
        jmsMessagingTemplate.convertAndSend(new ActiveMQQueue(destination), list);
    }
}
```

- 编写消费者

```
@JmsListener(destination = "user",containerFactory = "jmsListenerContainerQueue")
public void receiveStringQueue(String msg) {
        System.out.println("接收到消息...." + msg);
    }

@JmsListener(destination = "ooo",containerFactory = "jmsListenerContainerTopic")
public void receiveStringTopic(String msg) {
     System.out.println("接收到消息...." + msg);
 }
```

小结

本文详细介绍了为什么需要引入消息队列，JMS、ActiveMQ的基础概念以及常用API，与原生JAVA合及SpringBoot整合等知识点，可以让大家更好的了解ActiveMQ的使用场景及使用方式。