宝兰德消息中间件软件
V2.0.2.2.0 快速开始手册

北京宝兰德软件股份有限公司

前 言

本文档为需要快速了解 BES MQ 的用户提供了概览，通过阅读本文档，用户可以轻松掌握 BES MQ 的基本操作和使用。关于 BES MQ 更为详细的介绍请参阅 BES MQ 其他相关文档。

本文档适合的对象

本文档主要适用于想快速了解 BES MQ 基本操作和使用的管理人员，以及基于 BES MQ 进行消息中间件应用开发的开发人员。

本文档假定您已经具备如下技能：

1. 面向对象的编程技术；
2. Java 语言；
3. Java EE API；
4. 在终端窗口中发布命令；
5. 数据库的操作。

约定

BES MQ 定义了一些变量来表示 BES MQ 目录等信息，本文档中涉及到的有：

变量	说明
${com.bes.mq.installRoot}	BES MQ 安装目录
${com.bes.mq.brokerRoot}	BES MQ 代理实例目录

产品文档集

BES MQ 提供的文档集包括：

1. 快速开始：提供 BES MQ 的概览，介绍 BES MQ 的基本操作和使用。
2. 安装手册：详细介绍如何在各个操作系统上安装 BES MQ，以及产品的注册过程。
3. 用户手册：详细介绍 BES MQ 的配置和管理。
4. 开发手册：详细介绍基于 BES MQ 的消息中间件应用的开发过程。

技术支持

BES MQ 提供全方位的技术支持，获得技术支持的方式有：

网址：www.bessystem.com

Support Email：support@bessystem.com

Support Tel：400 650 1976

在取得技术支持时，请提供如下信息：

1. 姓名
2. 公司信息及联系方式
3. 操作系统及其版本
4. BES MQ 版本
5. 日志等错误的详细信息

目 录

第1章 产品介绍

1.1 关于 BES MQ

BES MQ 是一款构建于 BES 微内核体系之上，基于消息传递机制进行平台无关的数据交换的消息中间件产品。通过 BES MQ 可以进行快速、高效、可靠的消息传递，从而实现异步调用及系统解耦，为企业级应用和服务提供坚实的底层架构支撑。

BES MQ 支持消息传递功能，并增加了很多针对企业级应用的特性，同时提供了功能一致的 Java、C、C++、C#、Python 及 Go 客户端接口。利用 BES MQ 强大而灵活的集群模型，可以轻易地部署不同拓扑结构的集群来应对复杂场景的要求，从而快速地构建稳定、高效、安全、健壮、易扩展、跨平台的消息传递应用和企业级应用。

1.2 产品特性

BES MQ 具有如下特性：

1. 遵循最新的行 JMS 2.0、MQTT 3.1、STOMP 1.2、JCA 1.5、JNDI1.2 等；
2. 支持点对点（PTP）及发布/订阅（Pub/Sub）模型；
3. 支持基本队列/主题，以及具有扩展功能的组合队列/主题、镜像队列、虚拟主题；
4. 支持多种通信协议，如 TCP/SSL、UDP、HTTP/HTTPS、AMQP 以及国密 SSL 等；
5. 支持多种消息存储方式，如文件系统、数据库系统等；
6. 支持不同层面的安全设置，传输层支持 SSL，应用层支持 JAAS，支持国密算法
7. 支持不同粒度的访问权限控制，可以精确控制对队列的读、写、管理；
8. 支持强大的集群模型，以满足不同场合的需要；
9. 支持文件传输以及断点续传；
10. 支持主流的操作系统，如 Kylin、UOS、NeoKylin、RedHat、Windows 等；
11. 支持多种客户端编程语言，如 Java、C、C++、C#、Python、Go 等，并提供了 REST 服务接口；
12. 支持通过 CLI 和 Web 浏览器来进行管理和监控；
13. 支持 Spring、Spring Boot、Spring Cloud Stream。

1.3 支持的平台环境

BES MQ 产品认证的平台环境有：

操作系统	Windows：Windows 7、Windows 10

	RedHat AS：6.1、7.1 SUSE：10 Kylin：10 NeoKylin：10 UOS：20
Java 环境	JDK1.8.0
C/C++编译器	RedHat AS 6.1：gcc 7.3.0 SUSE：gcc 4.1.2 NeoKylin：gcc 7.3.0 Kylin：gcc 7.3.0 UOS：gcc 7.3.0 Windows：Visual Studio 2017 Microsoft Visual Studio 2019
C#编译器	Microsoft Visual Studio 2017 Microsoft Visual Studio 2019
浏览器	Firefox：61.0.1+ Chrome：73.0+

1.4 支持的规范

BES MQ 产品支持的规范/API 具体有：

规范/API	版本	规范/API	版本
JMS	2.0	JNDI	1.2
EJB	3.0	JCA	1.5
JTA	1.1	JDBC	3.0
MQTT	3.1	STOMP	1.2

以上规范/API 都可以向后兼容。

第2章 产品安装

本文档以 Windows 平台上的完全安装为例，安装的具体细节请参考《BES MQ 安装手册》。

2.1 安装前准备

请您联系 BES MQ 的销售代表，获取适用于您操作系统的安装包，预先安装好合适版本的 JDK，并确认您的系统至少满足如下需求：

组件	要求		
	最低配置	推荐配置	备注
物理内存	1G	2G 及以上	无
磁盘空间	500M	1G 及以上	安装 BES MQ 时需确保操作系统临时目录所在的磁盘上有 500M 以上的空闲空间。
Java 环境	JDK 1.8.0	JDK1.8.0_25	无
网络	百兆网卡	千兆网卡	BES MQ 安装时不需要使用操作系统的网络服务，但运行时需要使用网络服务。所以建议系统管理员使用 BES MQ 时确保 TCP/IP 协议、UDP 协议、TCP 多播路由等网络服务可用。

2.2 安装步骤

1. 启动安装过程，进入“语言选择”页面，安装程序支持简体中文和英文两种语言的选择。选择“简体中文”演示。

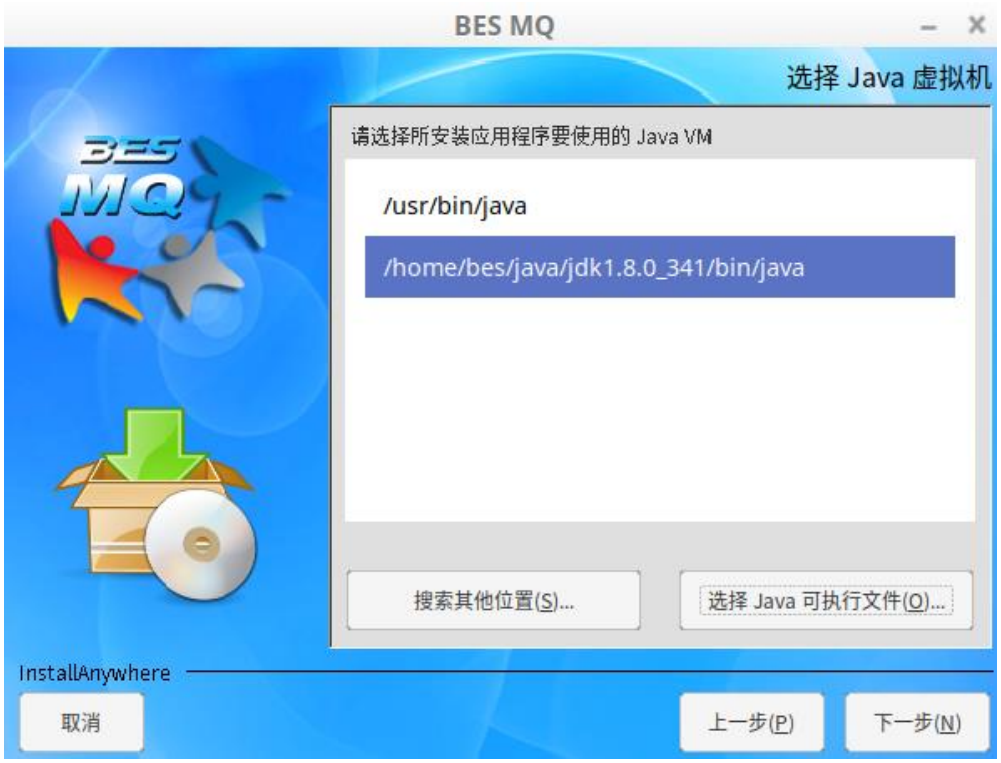
语言选择界面

2. 进入“**简介**”页面，根据向导提示，点击“**下一步**”继续执行安装过程：

3. 点击“**下一步**”，进入“**许可协议**”页面：

许可协议界面

4. 请仔细阅读许可协议的内容，在确认接受许可协议的情况下，选择“**我接受许可协议条款(A)**”，然后点击“**下一步**”，进入“**选择 Java 虚拟机**”页面：



选择 Java 虚拟机界面

5. 选择对应的 Java 虚拟机，点击“**下一步**”按钮，进入“**选择安装文件夹**”页面：

选择安装文件夹界面

6. 使用默认安装路径，点击“**下一步**”按钮，进入“**选择安装集**”页面：

选择安装集界面

7. 选择“**完全安装**”，点击“**下一步**”按钮，进入“**选择安装类型**”页面：

选择安装类型界面

8. 选择“**典型安装**”，点击“**下一步**”按钮，进入“**选择链接文件夹**”页面：

快捷方式配置界面

9. 使用默认选项，点击“**下一步**”按钮，进入“**预安装摘要**”页面：

预安装摘要界面

10. 点击“**安装**”按钮，直到完成全部的安装操作，提示“**安装成功**”：

至此，BES MQ 已经成功安装至路径/home/bes/BESMQ 下，并缺省创建了消息代理实例 broker1。broker1 位于/home/bes/BESMQ/var/broker/broker1 下，其他重要的默认信息为：

管理端口：3100

管理用户：admin

管用密码：B#2008_2108#es

本文档后面的内容都基于 broker1，约定的变量${com.bes.mq.installRoot}是/home/bes/BESMQ,${com.bes.mq.brokerRoot}为/home/bes/BESMQ/var/broker/broker1。

第3章 产品使用

3.1 启动消息代理实例

首先要启动消息代理 broker1。在${com.bes.mq.installRoot}/bin 目录下运行以下命令：

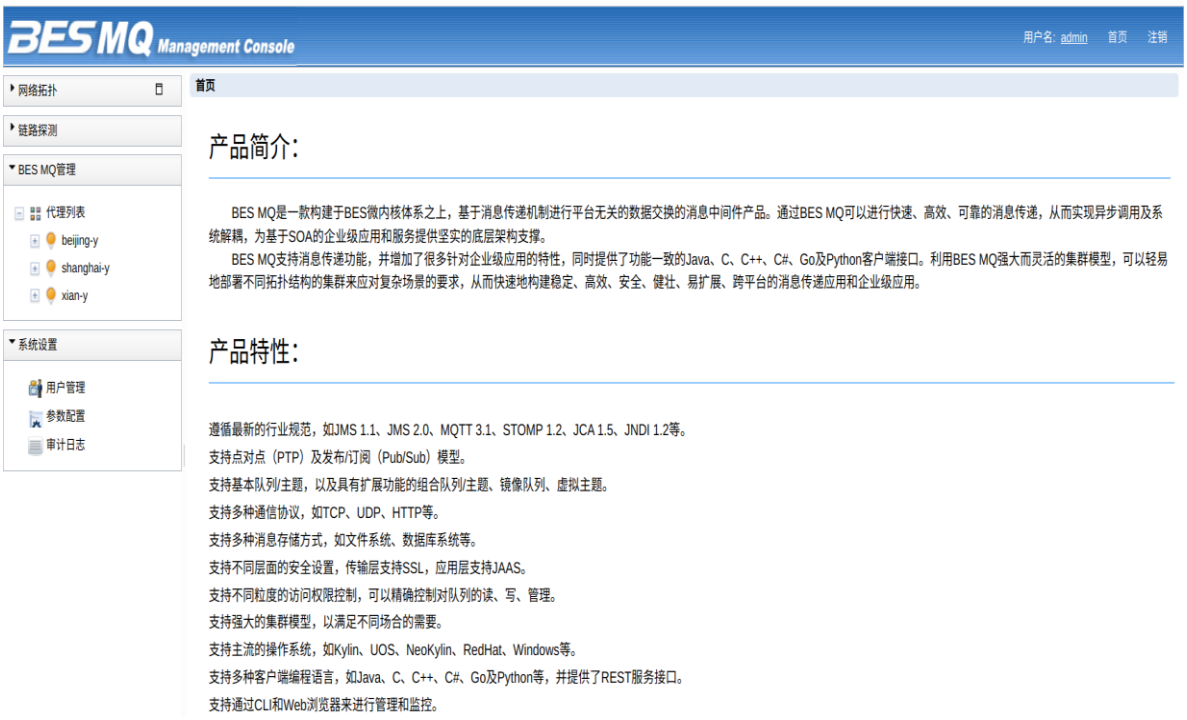
```
mqadmin start --broker --passport B#2008_2108#es broker
```

3.2 管理控制台的使用

3.2.1 启动管理控制台

在${com.bes.mq.installRoot}/bin 下运行脚本 startconsole.bat，然后在浏览器地址栏输入 http://ip:port/mqconsole 即可进入管理控制台登录界面（默认端口为 8490）：

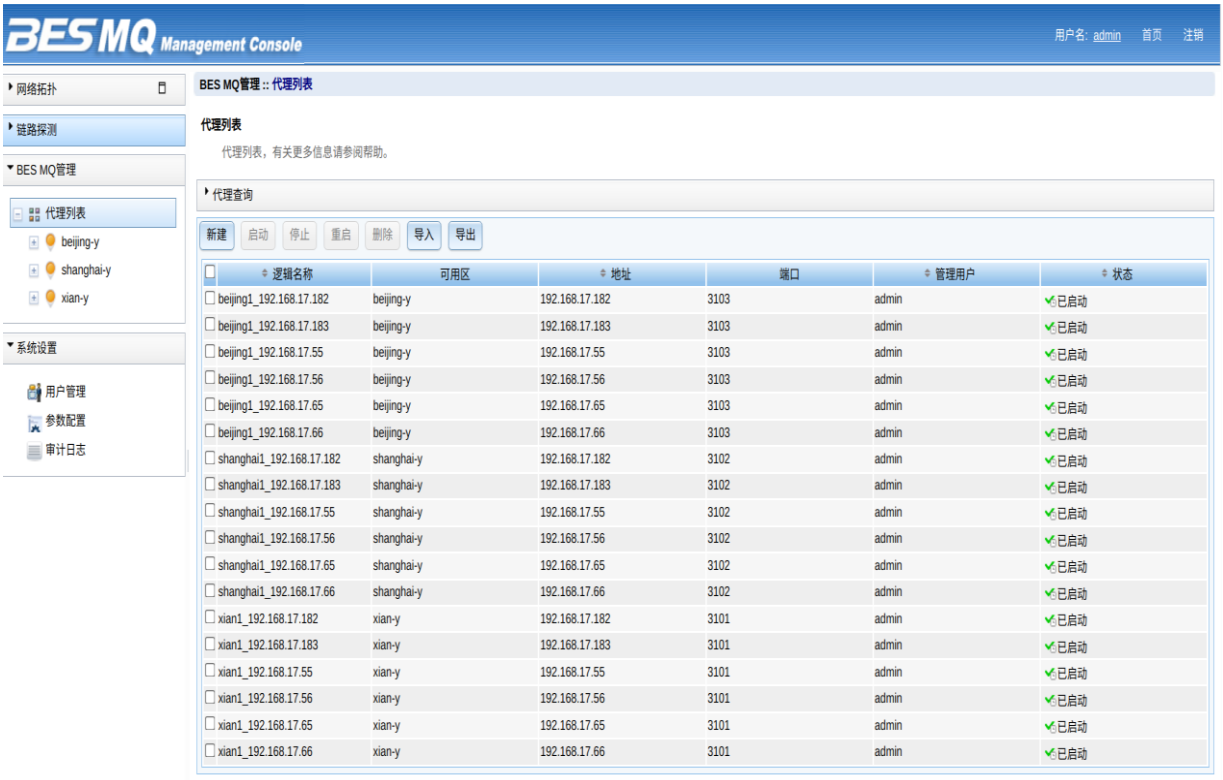
输入用户名和密码（默认的用户名与密码为 admin/B#2008_2108#es），点击**“确定”**进入 BES MQ 管理控制台首页：



3.2.2 添加代理实例

管理控制台默认未添加任何代理实例，需要手动添加需要管理的代理实例。接下来演示如何将安装时缺省创建的消息代理实例 broker1 添加到代理列表中。

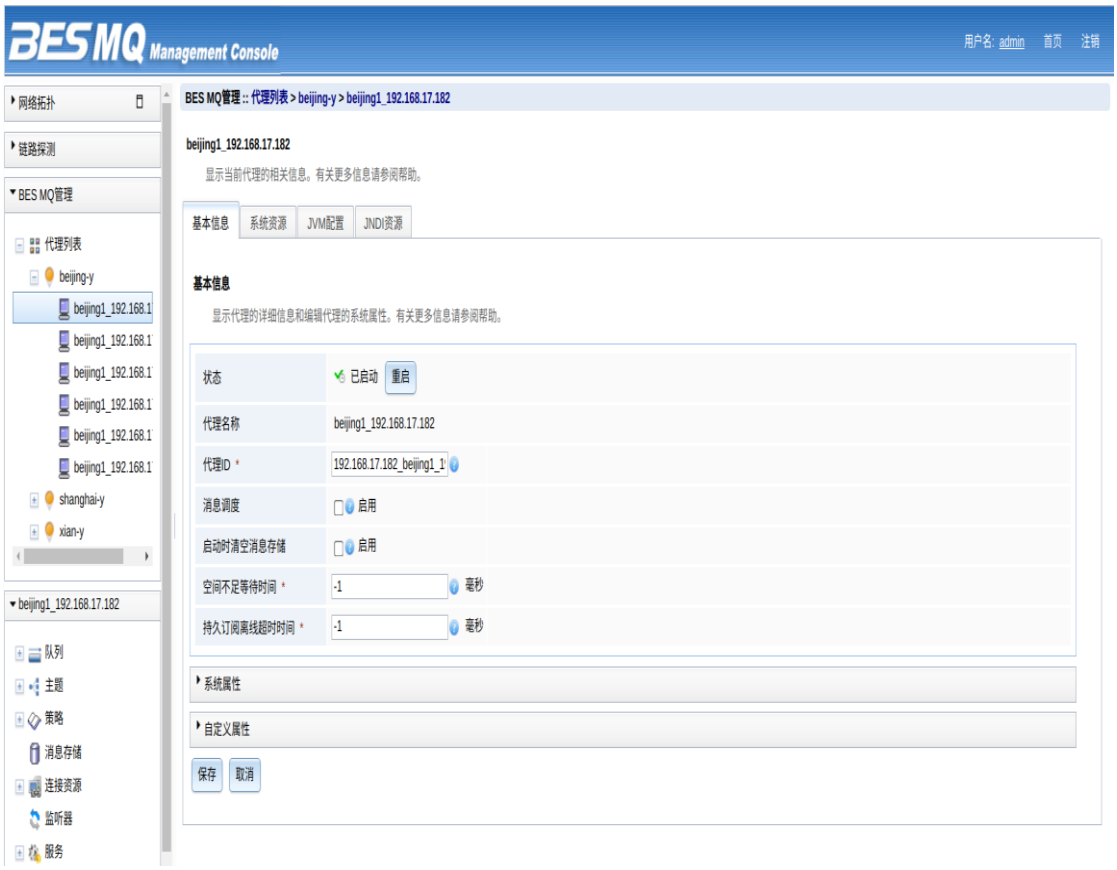
点击左侧导航树中的**“代理列表”**，出现代理列表展示页面：



点击“**新建**”按钮，输入代理 broker1 所在机器的地址、管理端口、管理用户名及管理密码：

点击“**保存**”按钮，回到代理列表展现页面：

点击“**代理列表**”-->“**broker1**”，就可以查看代理的信息了：



3.2.3 创建基本队列

在导航树上点击 broker1 的**“队列”**-->**“基本队列”**，在右侧的基本队列列表页面点击**“新建”**按钮，输入队列的名称及 JNDI 名称，点击**“保存”**，即可完成基本队列的创建：

3.2.4 创建连接工厂

在导航树上点击 broker1 的**“连接工厂”**，在右侧的连接工厂列表页面点击**“新建”**按钮，输入连接工厂的名称、URI（比如 tcp://host:port，其中 host 为代理监听器监听的 IP 或主机名，port 为代理监听器监听的端口，默认监听端口为 3200）及 JNDI 名称，点击**“保存”**，即可完成连接工厂的创建：

3.2.5 停止管理控制台

在${com.bes.mq.installRoot}/bin 下运行脚本 stopconsole.bat 即可停止管理控制台。

3.3 命令行工具的使用

BES MQ 提供了管理消息代理的命令行工具 mqadmin，mqadmin 工具的命令带有一系列的参数，包括可选参数和必选参数。

使用 mqadmin 工具执行命令有两种方式：

1. **命令行执行**：在命令行上直接输入命令，格式如下：

```
mqadmin 命令名 [可选参数 1] [可选参数 2] 必选参数 1
```

2. **交互式模式执行**：在命令行直接输入 mqadmin 进入交互模式，然后在交互模式下输

入命令和参数，格式如下：

```
mqadmin>命令名 [可选参数 1] [可选参数 2] 必选参数 1
```

使用“help”命令可以查看 BES MQ 支持的所有命令。

使用“help 命令名”可以查看指定命令的简介和参数列表。

下面简单介绍 BES MQ 常用命令的使用，关于 BES MQ 命令的详细使用请参阅用户手册相关章节。

3.3.1 管理代理实例

安装产品时默认创建了一个代理,通过命令行可以完成同样的创建任务。创建完成之后，可以启动、停止、重启、删除代理，对应的命令分别示例如下：

1. **创建代理**

```
mqadmin create --broker --passport B#2008_2108#es --admin-port 3100 --tcp-port 3200 --jndi-port 3300 broker1
```

2. **启动代理**

```
mqadmin start --broker --passport B#2008_2108#es broker1
```

3. **停止代理**

```
mqadmin stop --broker --passport B#2008_2108#es broker1
```

4. **重启代理**

```
mqadmin restart --broker --passport B#2008_2108#es broker1
```

5. **删除代理**

删除代理前，首先需要停止代理，然后执行以下命令删除代理：

```
mqadmin delete --broker --passport B#2008_2108#es broker1
```

3.3.2 注册代理实例

当存在多个 broker 时，在管理控制台逐个添加代理必然造成不便，因此 BESMQ 提供了自动注册代理实例功能，用户可以在启动 broker 的时候将代理实例注册到管理控制台。

（1） 启动 broker 时，通过如下命令进行注册 broker

```
mqadmin start --broker --passport B#2008_2108#es --register-address 127.0.0.1:8490 --logic-name broker1 broker1
```

参数说明：

--register-address 127.0.0.1:8490 表示控制台的 IP 和端口，控制台可以和 broker 不在同一台机器。

--logic-name broker1 表示 broker 在控制台中的逻辑名称，如果逻辑名称已经存在则注册失败。

注意：

如果控制台没有启动，则会提示注册失败，但不影响 broker 正常启动。

（2)注册 broker 的时候,采用默认的控制台登录用户名和密码 admin/B#2008_2108#es，可以通过如下参数进行控制

--register-user 控制台登录用户名

--register-password 控制台登录用密码

（3）注册成功后，登录管理控制台，查看 broker 是否添加到了控制台。

3.3.3 简单管理操作

管理控制台上的绝大部分操作都可以通过命令行完成,接下来我们用命令行完成前一节中通过管理控制台创建的基本队列和连接工厂。

1. 基本队列操作

1) **创建基本队列**

```
mqadmin create --queue --passport B#2008_2108#es --jndi-name queue/queue1 queue1
```

2) **查看所有基本队列**

```
mqadmin list --queues --passport B#2008_2108#es
```

3) **删除基本队列**

```
mqadmin delete --queue --passport B#2008_2108#es queue1
```

2. 连接工厂操作

1) **创建连接工厂**

```
mqadmin create --connection-factory --passport B#2008_2108#es --jndi-name connectionFactory/cf1 --uri tcp://192.168.0.26:3200 cf1
```

2) **查看所有连接工厂**

```
mqadmin list --connection-factories --passport B#2008_2108#es
```

3) **删除连接工厂**

```
mqadmin delete --connection-factory --passport B#2008_2108#es cf1
```

第4章 应用开发

BES MQ 提供了基于 Java、C、C++、C#、Python 和 Go 六种编程语言的客户端应用开发 API，并且在 API 的设计上尽可能地保持了接口的一致性，以便应用开发者能轻松地从一种编程语言转移到另一种编程语言。

Java API 完全支持 Java 消息服务（JMS）规范 2.0 版本，并在此基础上增加了大量适用于企业应用的特性。C++ API 提供了类似于 Java API 的一套接口，并且支持绝大多数 Java API 支持的功能。

下面分别对 Java API 和 RedHat AS 5.2（64 位）平台下的 C++、C 及 WINDOWS32 平台下 C# API 客户端应用开发进行简要介绍。基于 BES MQ 进行应用开发的详细信息请参考《BES MQ 开发手册》。

4.1 准备环境

Java 客户端依赖 jar 包 ${com.bes.mq.installRoot}/lib/client/java/besmq-client.jar。

RedHat AS5.2（64 位）平台下的 C++ 客户端依赖的库在压缩文件 ${com.bes.mq.installRoot}/lib/client /cpp/REDHATAS64-X86-V52.tar.gz 中，请先解压此文件到合适的位置，设定环境变量 BESMQCPP_HOME 指向此目录。

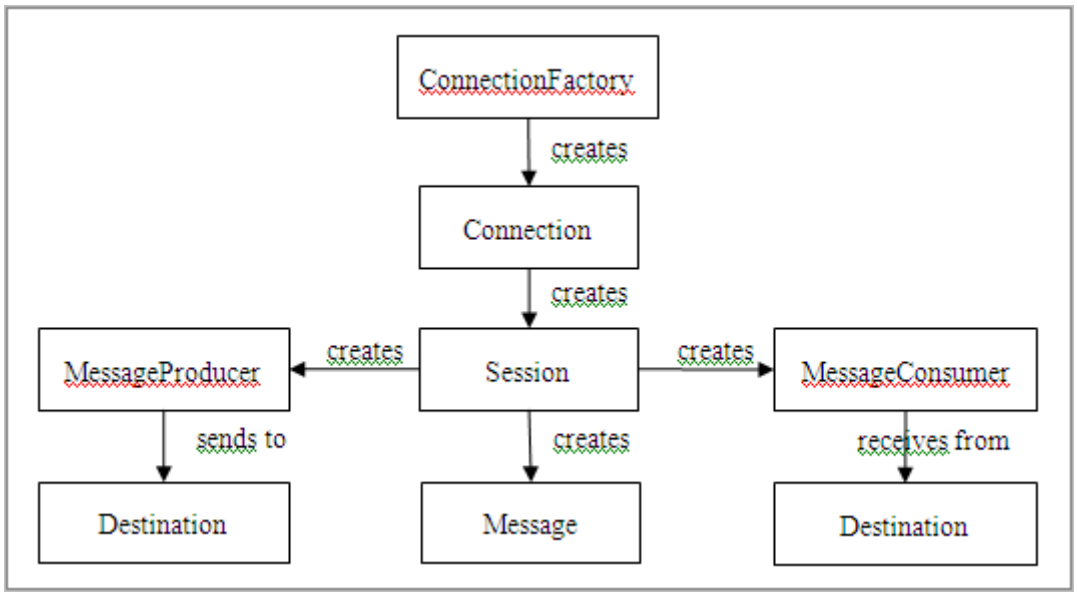
RedHat AS5.2（64 位）平台下的 C 客户端依赖的库在压缩文件 ${com.bes.mq.installRoot}/lib/client /c/REDHATAS64-X86-V52.tar.gz 中，请先解压此文件到合适的位置，设定环境变量 BESMQC_HOME 指向此目录。

在开发应用之前，请先做好如下的准备工作：

1. 正确安装了 JDK（1.8 或者以上版本）；
2. 正确安装了 C/C++及 C#编译器；
3. 正常启动了代理 broker1，请参考代理实例管理；
4. 成功创建了基本队列 queue1，请参考基本队列创建；
5. 成功创建了连接工厂 cf1，请参考连接工厂创建。

4.2 开发 Java 客户端

Java 客户端应用开发中涉及到的编程对象以及它们之间的关系如下图所示：



ConnectionFactory 对象封装了一组创建 Connection 的配置参数。应用程序可以直接实例化 BESMQConnectionFactory 来创建一个 ConnectionFactory 实例；也可以通过 JNDI 来查找已经在代理端创建的连接工厂，jndi.properties 配置如下：

```
java.naming.factory.initial=com.bes.mq.jndi.interfaces.NamingContextFactory
java.naming.factory.url.pkgs=com.bes.mq.jndi.interfaces
java.naming.provider.url=bmq://hostname:port
```

备注：hostname 为代理实例所在主机的 IP 地址或主机名；port 为 JNDI 端口，默认为 3300。

Connection 表示客户端和代理之间的一条物理连接，可以通过 ConnectionFactory 创建 Connection。Session 表示一个用于发送和接收消息的单线程上下文环境，可以通过 Connection 创建 Session。MessageProducer 用于向目的地发送消息，MessageConsumer 用于从目的地接收消息，Message 表示一个具体消息，它们均由 Session 创建。

4.2.1 发送消息

```
public class QueueSenderMain {
  public static void main(String[] args){
    ConnectionFactory connectionFactory = null;
    Connection connection = null;
    Session session = null;
    Queue queue = null;
    MessageProducer producer = null;
    Context context = null;
```

```
try {
    // 初始化 JNDI Context
    context = new InitialContext();
    // 查找连接工厂
    connectionFactory = (ConnectionFactory) context
            .lookup("connectionFactory/cf1");
    // 创建连接
    connection = connectionFactory.createConnection();
    // 打开连接
    connection.start();
    // 创建会话
    session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
    // 查找队列
    queue = (Queue) context.lookup("queue/queue1");
    // 创建生产者
    producer = session.createProducer(queue);
    // 创建 Text 类型消息
    TextMessage msg = session.createTextMessage("test");
    // 发送消息
    producer.send(msg);
} catch (NamingException e) {
    // JNDI 异常处理
    e.printStackTrace();
} catch (JMSException e) {
    // JMS 异常处理
    e.printStackTrace();
} finally {
    // 关闭 Context
    if (context != null) {
        try {
            context.close();
```

```
            } catch (NamingException e) {
                e.printStackTrace();
            }
        }
        // 关闭生产者
        if (producer != null) {
            try {
                producer.close();
            } catch (JMSException e) {
                e.printStackTrace();
            }
        }
        // 关闭会话
        if (session != null) {
            try {
                session.close();
            } catch (JMSException e) {
                e.printStackTrace();
            }
        }
        // 关闭连接
        if (connection != null) {
            try {
                connection.close();
            } catch (JMSException e) {
                e.printStackTrace();
            }
        }
    }
  }
}
```

4.2.2 接收消息

```
public class QueueConsumerMain implements MessageListener {
    public static void main(String[] args) throws Exception {
        Context context = null;
        ConnectionFactory connectionFactory = null;
        Connection connection = null;
        Session session = null;
        Queue queue = null;
        MessageConsumer consumer = null;
        try {
            // 初始化 JNDI Context
            context = new InitialContext();
            // 查找连接工厂
            connectionFactory = (ConnectionFactory) context
                    .lookup("connectionFactory/cf1");
            // 创建连接
            connection = connectionFactory.createConnection();
            // 启动 connection
            connection.start();
            // 创建会话
            session = connection.createSession(false,
                    Session.AUTO_ACKNOWLEDGE);
            // 查找 queue
            queue = (Queue) context.lookup("queue/queue1");
            // 创建消费者
            consumer = session.createConsumer(queue);
            // 设置消息监听器
            consumer.setMessageListener(new QueueConsumerMain ());
            // 等待消息接收完成
            Thread.sleep(60 * 1000);
```

```
            } catch (JMSException e) {
                e.printStackTrace();
            } catch (NamingException e) {
                e.printStackTrace();
            } finally {
                // 关闭 Context
                if (context != null) {
                    try {
                        context.close();
                    } catch (NamingException e) {
                        e.printStackTrace();
                    }
                }
                // 关闭消费者
                if (consumer != null) {
                    try {
                        consumer.close();
                    } catch (JMSException e) {
                        e.printStackTrace();
                    }
                }
                // 关闭会话
                if (session != null) {
                    try {
                        session.close();
                    } catch (JMSException e) {
                        e.printStackTrace();
                    }
                }
                // 关闭连接
                if (connection != null) {
```

```
                    try {
                        connection.close();
                    } catch (JMSException e) {
                        e.printStackTrace();
                    }
                }
            }
        }

        @Override
        public void onMessage(Message message) {
            try {
                // 处理消息
                if (message instanceof TextMessage) {
                    String text = ((TextMessage) message).getText();
                    System.out.println(text);
                }
            } catch (JMSException e) {
                e.printStackTrace();
            }
        }
    }
```

更详细的例子请参考 BES MQ 自带的示例，位于${com.bes.mq.installRoot}/samples/java 下。

4.3 开发 C++客户端

BES MQ 提供的 C++客户端 API 与 Java 客户端 API 类似，Java 客户端应用开发中涉及到的编程对象以及它们之间的关系绝大部分都可以直接映射到 C++客户端编程里，通过下面的例子可以看到，BES MQ 对 Java 客户端和 C++客户端提供了统一的编程模型。

4.3.1 发送消息

```
Connection* connection;
```

```
Session* session;
Destination* destination;
MessageProducer* producer;
string brokerURI;
string destName;
try{
    // 创建连接工厂
    auto_ptr<BESMQConnectionFactory> connectionFactory(
        new BESMQConnectionFactory( brokerURI ) );
    // 创建连接
    connection = connectionFactory->createConnection();
    // 启动 Connection
    connection->start();
    // 创建会话
    session = connection->createSession( Session::AUTO_ACKNOWLEDGE );
    // 创建队列
    destination = session->createQueue(destName);
    // 创建生产者
    producer = session->createProducer( destination );
    // 设置 DeliveryMode
    producer->setDeliveryMode( DeliveryMode::PERSISTENT );
    // 发送消息
    auto_ptr<TextMessage> message(session->createTextMessage());
    message->setText("Hello World!");
    cout << "Producer send the message:" << endl;
    cout << message->getText() << endl;
    producer->send(message.get());
}catch ( BMSException& e ) {
    e.printStackTrace();
}
// 关闭连接、释放资源
```

```
try {
    if (NULL != destination) {
        delete destination;
        destination = NULL;
    }
} catch (BMSException& e) {
    e.printStackTrace();
}

try {
    if (NULL != producer) {
        delete producer;
        producer = NULL;
    }
} catch (BMSException& e) {
    e.printStackTrace();
}

try {
    if (NULL != session) {
        session->close();
    }
} catch (BMSException& e) {
    e.printStackTrace();
}

try {
    if (NULL != connection) {
        connection->close();
    }
} catch (BMSException& e) {
```

```
        e.printStackTrace();
    }


    try {
        if (NULL != session) {
            delete session;
            session = NULL;
        }
    } catch (BMSException& e) {
        e.printStackTrace();
    }


    try {
        if (NULL != connection) {
            delete connection;
            connection = NULL;
        }
    } catch (BMSException& e) {
        e.printStackTrace();
    }
```

4.3.2 接收消息

```
    Connection* connection;
    Session* session;
    Destination* destination;
    MessageConsumer* consumer;
    string brokerURI;
    string destName;
    try {
        // 创建连接工厂
        auto_ptr<BESMQConnectionFactory> connectionFactory(
```

```
            new BESMQConnectionFactory( brokerURI ) );
        // 创建连接
        connection = connectionFactory->createConnection();
        // 启动连接
        connection->start();
        // 创建会话
        session = connection->createSession( Session::AUTO_ACKNOWLEDGE );
        // 创建队列
        destination = session->createQueue(destName);
        // 创建消费者
        consumer = session->createConsumer( destination );
        // 接收消息
        const TextMessage* textMessage =
            dynamic_cast<const TextMessage*> (consumer->receive(1000));
        string text =  textMessage->getText();
        cout <<"Received message:"<< text.c_str() << endl;
    } catch (BMSException& e) {
        e.printStackTrace();
    }
    // 关闭连接、释放资源
    try {
        if (NULL != destination) {
            delete destination;
            destination = NULL;
        }
    } catch (BMSException& e) {
        e.printStackTrace();
    }

    try {
        if (NULL != consumer) {
```

```
            delete consumer;
            consumer = NULL;
        }
    } catch (BMSException& e) {
        e.printStackTrace();
    }

    try {
        if (NULL != session) {
            delete session;
            session = NULL;
        }
    } catch (BMSException& e) {
        e.printStackTrace();
    }

    try {
        if (NULL != connection) {
            delete connection;
            connection = NULL;
        }
    } catch (BMSException& e) {
        e.printStackTrace();
    }
```

4.3.3 Makefile

编译使用的 makefile 的内容如下：

```
CXX=g++
CXXFLAGS = -g -O2 -I$(BESMQ_CPP)/include
LIBS= $(BESMQ_CPP)/lib/libbesmq-cpp.so -Wl,-rpath -Wl,$(BESMQ_CPP)/lib
```

```
objects=queue-consumer queue-producer


build:$(objects)


queue-consumer:QueueConsumer.cpp
    $(CXX) $(CXXFLAGS) -o $@ $< $(LIBS)


queue-producer:QueueProducer.cpp
    $(CXX) $(CXXFLAGS) -o $@ $< $(LIBS)


clean:
    rm $(objects)
```

其中 $(BESMQ_CPP) 指向 ${com.bes.mq.installRoot}/lib/client/cpp/RedhatAS64-X86-V52.tar.gz 解压后所在目录的 besmq-cpp-64 子目录。

更详细的例子请参考 BES MQ 自带的示例，位于 ${com.bes.mq.installRoot}/samples/cpp 目录下。

4.4 开发 C 客户端

C 客户端依赖如下目录下的文件：

1. ${com.bes.mq.installRoot}/lib/client/c
2. ${com.bes.mq.installRoot}/lib/client/cpp

请根据客户端操作系统类型选择对应的文件，并满足编译器最低版本要求，两个目录下的文件名一致。具体文件名、对应操作系统最低版本和位数、及编译器相关信息和 C++客户端一致。

以 REDHATAS64-X86-V52.tar.gz 为例，适用于处理器架构为 X86，操作系统为 RedHat 64 位，最低版本支持 5.2。

分别解压 ${com.bes.mq.installRoot}/lib/client/c 和 ${com.bes.mq.installRoot}/lib/client/cpp 下的 REDHATAS64-X86-V52.tar.gz 文件至各自所在目录，并设定环境变量：

1. ${ BESMQCPP_HOME }指向${com.bes.mq.installRoot}/lib/client/cpp
2. ${ BESMQC_HOME }指向${com.bes.mq.installRoot}/lib/client/c

以下两节说明了如何使用 BES MQ C 客户端发送和接收消息。

4.4.1 发送消息

```
int main(int argc, char * argv[]) {
    /* BES MQ C 客户端库资源上下文 */
    bmqc_context_t  ctx;
    /* 错误信息 */
    bmqc_err_t  err;
    /* 连接句柄 */
    bmqc_hconn_t  conn = 0;
    /* 目的地句柄 */
    bmqc_hdest_t  queue = 0;
    /* 消息属性句柄 */
    bmqc_hprops_t  props = 0;
    /* 连接属性 */
    bmqc_conn_opt_t  conn_opt;
    /* 目的地属性 */
    bmqc_dest_opt_t  dest_opt;
    /* 消息属性 */
    bmqc_msg_opt_t  msg_opt;
    /* 消息头 */
    bmqc_msg_info_t  msg_info;
    /* 消息内容 */
    unsigned char *  content = 0;
    /* 消息代理连接串 */
    char *  broker_uri = "tcp://localhost:3200";
    /* 队列名 */
    char *  queue_name = "testQueue";
    /* 消息数 */
    int  msg_num = 10;
    /* 计数器 */
    int  i = 0;
```

```
/* 初始化 C 客户端库资源上下文*/
bmqc_init_context(&ctx);
bmqc_init(&ctx, &err);
if (RET_SUCCESS != err.err_no) {
    // TODO 处理错误信息
    return -1;
}
/* 连接至消息代理 */
bmqc_init_conn_opt(&conn_opt);
bmqc_conn(&conn, broker_uri, &conn_opt, &err);
if (RET_SUCCESS != err.err_no) {
    // TODO 处理错误信息
    return -1;
}
/* 打开目的地 */
bmqc_init_dest_opt(&dest_opt);
dest_opt.dest_type = BMQC_QUEUE;
bmqc_open(conn, &queue, queue_name, &dest_opt, &err);
if (RET_SUCCESS != err.err_no) {
    // TODO 处理错误信息
    return -1;
}
/* 发送消息 */
bmqc_init_msg_info(&msg_info);
bmqc_init_msg_opt(&msg_opt);
msg_info.type = BMQC_BYTES_MESSAGE;
msg_info.conts_len = 8;
content = (unsigned char *)malloc(msg_info.conts_len + 1);
memset(content, 0, msg_info.conts_len + 1);
for (i = 1; i <= msg_num; i++) {
```

```
            memset(content, 'A' + (i - 1), msg_info.conts_len);
            bmqc_put(queue, &msg_info, props, content, &msg_opt, &err);
            if (RET_SUCCESS != err.err_no) {
                // TODO 处理错误信息
                break;
            } else {
                printf("put the %dst message successfully, and content is: %s\n",
i, content);
            }
        }
        free(content);
        /* 关闭目的地 */
        bmqc_close(queue, &err);
        if (RET_SUCCESS != err.err_no) {
            // TODO 处理错误信息
            return -1;
        }
        /* 断开与消息代理之间的连接 */
        bmqc_disconn(conn, &err);
        if (RET_SUCCESS != err.err_no) {
            // TODO 处理错误信息
            return -1;
        }
        /* 销毁 C 客户端库资源上下文 */
        bmqc_destroy(&err);
        if (RET_SUCCESS != err.err_no) {
            // TODO 处理错误信息
            return -1;
        }
        return 0;
    }
```

4.4.2 接收消息

```
int main(int argc, char * argv[])
{
    /* BES MQ C 客户端库资源上下文 */
    bmqc_context_t  ctx;
    /* 错误信息 */
    bmqc_err_t  err;
    /* 连接句柄 */
    bmqc_hconn_t  conn = 0;
    /* 目的地句柄 */
    bmqc_hdest_t  queue = 0;
    /* 消息属性句柄 */
    bmqc_hprops_t  props = 0;
    /* 连接属性 */
    bmqc_conn_opt_t  conn_opt;
    /* 目的地属性 */
    bmqc_dest_opt_t  dest_opt;
    /* 消息属性 */
    bmqc_msg_opt_t  msg_opt;
    /* 消息头 */
    bmqc_msg_info_t  msg_info;
    /* 消息内容 */
    unsigned char *  content = 0;
    /* 消息代理连接串 */
    char *  broker_uri = "tcp://localhost:3200";
    /* 队列名 */
    char *  queue_name = "testQueue";
    /* 消息数 */
    int  msg_num = 10;
    /* 计数器 */
```

```
int  i = 0;


/* 初始化 C 客户端库资源上下文 */
bmqc_init_context(&ctx);
bmqc_init(&ctx, &err);
if (RET_SUCCESS != err.err_no) {
    // TODO 处理错误信息
    return -1;
}
/*  连接至消息代理 */
bmqc_init_conn_opt(&conn_opt);
bmqc_conn(&conn, broker_uri, &conn_opt, &err);
if (RET_SUCCESS != err.err_no) {
    // TODO 处理错误信息
    return -1;
}
/* 打开目的地 */
bmqc_init_dest_opt(&dest_opt);
dest_opt.dest_type = BMQC_QUEUE;
bmqc_open(conn, &queue, queue_name, &dest_opt, &err);
if (RET_SUCCESS != err.err_no) {
    // TODO 处理错误信息
    return -1;
}
/* 接收消息 */
bmqc_init_msg_opt(&msg_opt);
for (i = 1; i <= msg_num; i++) {
    msg_opt.operate_type = BMQC_GET;
    msg_opt.get_timeout = 5 * 1000;
    bmqc_get(queue, &msg_info, props, &content, &msg_opt, &err);
    if (RET_SUCCESS != err.err_no) {
```

```
                // TODO 处理错误信息
                break;
            }

            if (msg_info.type == BMQC_NONE_MESSAGE) {
                printf("get the %dth message timeout!\n", i);
            }
            else {
                printf("get the %dth message successfully, and content is: %s\n",
i, content);
            }
        }
        /*关闭目的地 */
        bmqc_close(queue, &err);
        if (RET_SUCCESS != err.err_no) {
            // TODO 处理错误信息
            return -1;
        }
        /* 断开与消息代理之间的连接 */
        bmqc_disconn(conn, &err);
        if (RET_SUCCESS != err.err_no) {
            // TODO 处理错误信息
            return -1;
        }
        /* 销毁 C 客户端库资源上下文 */
        bmqc_destroy(&err);
        if (RET_SUCCESS != err.err_no) {
            // TODO 处理错误信息
            return -1;
        }
```

```
        return 0;
 }
```

4.4.3 编译

C 平台编译器众多,这里以典型的 Linux 平台上的 gcc 编译器和 Windows 平台上的 Visual Studio 为例，其余平台请参考${com.bes.mq.installRoot}/samples/c/index.html。

gcc 编译器（适用于 RedHat X86 和 SUSE）

新建 makefile.unix 内容如下，然后运行 make -f makefile.unix 进行编译：

```
    CXX=gcc
    CXXFLAGS=-g -O2 -I$(BESMQC_HOME)/besmq-c-64/include
    LIBS=-L$(BESMQC_HOME)/besmq-c-64/lib    -L$(BESMQCPP_HOME)/besmq-cpp-64/lib -L$(BESMQCPP_HOME)/apr-64/lib    -L$(BESMQCPP_HOME)/openssl-64/lib    -lbesmq-c -lbesmq-cpp -lapr-1 -lssl -lcrypto
    objects=put_msg get_msg
    build:$(objects)
    put_msg:put_msg.c
        $(CXX) $(CXXFLAGS) -o put_msg put_msg.c $(LIBS)
    get_msg:get_msg.c
        $(CXX) $(CXXFLAGS) -o get_msg get_msg.c $(LIBS)
    clean:
        rm -f $(objects) *.o
```

Microsoft Visual Studio 编译器（适用于 Windows）

新建 makefile.win 内容如下，然后进入 Visual Studio 2005 命令行提示工具，运行 nmake -f makefile.win 进行编译：

```
    CXX=cl
    LD=/link
    CXXFLAGS=/DEBUG /O2 /I "$(BESMQC_HOME)\besmq-c-32\include" /EHsc /MD
    LIBS=/LIBPATH:"$(BESMQC_HOME)\besmq-c-32\lib" besmq-c.lib
    objects=put_msg.exe get_msg.exe
    build:$(objects)
    put_msg.exe:
```

```
    $(CXX) put_msg.c $(CXXFLAGS) $(LD) $(LIBS)

get_msg.exe:

    $(CXX) get_msg.c $(CXXFLAGS) $(LD) $(LIBS)

clean:

    del  $(objects)

    del /q *.obj

    del /q *.manifest
```

4.5 开发 C#客户端

C#客户端依赖如下目录下的文件：

1. ${com.bes.mq.installRoot}/lib/client/c
2. ${com.bes.mq.installRoot}/lib/client/cpp

请根据客户端操作系统类型选择对应的文件，并满足编译器最低版本要求，两个目录下的文件名一致。

以 WINDOWS32-X86.zip 为例，适用于处理器架构为 X86，操作系统为 Windows 32 位。

分别解压${com.bes.mq.installRoot}/lib/client/c 和${com.bes.mq.installRoot}/lib/client/cpp 下的 WINDOWS32-X86.zip 文件至各自所在目录，然后将各个子目录中的 bin 目录下的 dll 文件拷贝到和 C#代码编译后的 exe 同一目录中。

3. ${com.bes.mq.installRoot}/lib/client/cs

将此目录下的WINDOWS.zip解压至当前文件夹,然后将解压后的besmq-cs文件夹中所有的文件拷贝到 vs 项目根目录中。

以下两节说明了如何使用 BES MQ C#客户端发送和接收消息。

4.5.1 发送消息

```
static void Main(string[] args)  {

    // 错误信息

    BmqCsErr error = new BmqCsErr();

    try

    {

        // C#客户端上下文信息

        BmqCsContext context = new BmqCsContext();
```

```
            BesMqInterface.BmqCsInitContext(ref context);

            // 初始化资源
            BesMqInterface.BmqCsInit(ref context, ref error);
            // 连接句柄
            IntPtr conn = IntPtr.Zero;

            // 消息代理连接串
            string uri = "tcp://localhost:3200";

            // 连接属性
            BmqCsConnOpt opt = new BmqCsConnOpt();
            BesMqInterface.BmqCsInitConnOpt(ref opt);
            opt.username = "admin";
            opt.password = " B#2008_2108#es";

            // 连接至消息代理
            BesMqInterface.BmqCsConn(ref conn, uri, ref opt, ref error);

            // 目的地句柄
            IntPtr queueDest = IntPtr.Zero;

            // 打开目的地
            BmqCsDestOpt destOpt = new BmqCsDestOpt();
            BesMqInterface.BmqCsInitDestOpt(ref destOpt);
            destOpt.ackMode = BmqCsConst.BMQCS_AUTO_ACKNOWLEDGE;
            destOpt.destType = BmqCsConst.BMQCS_QUEUE;
            BesMqInterface.BmqCsOpen(conn, ref queueDest, "testQueue", ref
destOpt, ref error);

            // 消息属性
```

```
            BmqCsMsgOpt msgOpt = new BmqCsMsgOpt();
            BesMqInterface.BmqCsInitMsgOpt(ref msgOpt);
            msgOpt.operateType = BmqCsConst.BMQCS_PUT;

            // 消息头
            BmqCsMsgInfo bmqMsg = new BmqCsMsgInfo();
            BesMqInterface.BmqCsInitMsgInfo(ref bmqMsg);
            bmqMsg.type = BmqCsConst.BMQCS_TEXT_MESSAGE;
            IntPtr props = IntPtr.Zero;

            // 消息内容
            string content = "Hello world!";
            byte[] byteArray = System.Text.Encoding.Default.GetBytes(content);
            bmqMsg.contsLen = (uint)byteArray.Length;

            // 发送消息
            BesMqInterface.BmqCsPut(queueDest, ref bmqMsg, props, byteArray, ref
msgOpt,        ref error);
        }
        catch (Exception e)
        {
            Console.WriteLine(e.StackTrace);
        }
        finally
        {
            // 关闭资源
            BesMqInterface.BmqCsDestroy(ref error);
        }
    }
```

4.5.2 接收消息

```
static void Main(string[] args)  {
    // 错误信息
    BmqCsErr error = new BmqCsErr();
    try
    {
        // C#客户端上下文信息
        BmqCsContext context = new BmqCsContext();
        BesMqInterface.BmqCsInitContext(ref context);

        // 初始化资源
        BesMqInterface.BmqCsInit(ref context, ref error);
        // 连接句柄
        IntPtr conn = IntPtr.Zero;

        // 消息代理连接串
        string uri = "tcp://localhost:3200";

        // 连接属性
        BmqCsConnOpt opt = new BmqCsConnOpt();
        BesMqInterface.BmqCsInitConnOpt(ref opt);
        opt.username = "admin";
        opt.password = " B#2008_2108#es";

        // 连接至消息代理
        BesMqInterface.BmqCsConn(ref conn, uri, ref opt, ref error);

        // 目的地句柄
        IntPtr queueDest = IntPtr.Zero;
```

```
            // 打开目的地
            BmqCsDestOpt destOpt = new BmqCsDestOpt();
            BesMqInterface.BmqCsInitDestOpt(ref destOpt);
            destOpt.ackMode = BmqCsConst.BMQCS_AUTO_ACKNOWLEDGE;
            destOpt.destType = BmqCsConst.BMQCS_QUEUE;
            BesMqInterface.BmqCsOpen(conn, ref queueDest, "testQueue", ref
destOpt, ref error);

            // 消息头
            BmqCsMsgOpt msgOpt = new BmqCsMsgOpt();
            BesMqInterface.BmqCsInitMsgOpt(ref msgOpt);
            msgOpt.operateType = BmqCsConst.BMQCS_GET;

            // 消息属性
            BmqCsMsgInfo bmqMsg = new BmqCsMsgInfo();
            BesMqInterface.BmqCsInitMsgInfo(ref bmqMsg);
            IntPtr props = IntPtr.Zero;

            // 消息内容
            byte[] byteArray = null;

            // 接收消息
            BesMqInterface.BmqCsGet(queueDest, ref bmqMsg, props, ref byteArray,
ref msgOpt, ref error);
            string content = System.Text.Encoding.Default.GetString(byteArray, 0,
byteArray.Length);
            Console.WriteLine(content);
        }
        catch (Exception e)
        {
            Console.WriteLine(e.StackTrace);
        }
```

```
        finally
        {
            // 关闭资源
            BesMqInterface.BmqCsDestroy(ref error);
        }
    }
```

4.5.3 编译

Microsoft Visual Studio 编译器

将上面发送消息以及接收消息的代码粘贴到 vs 项目工程中，然后 F5 启动编译即可。

4.6 开发 Go 客户端

Go 客户端依赖如下目录下的文件：

1. ${com.bes.mq.installRoot}/lib/client/go

请根据客户端操作系统类型选择对应的文件，并满足编译器最低版本要求，两个目录下的文件名一致。具体文件名、对应操作系统最低版本和位数、及编译器相关信息和 Go 客户端一致。

以 RHEL64-X86-V7.tar.gz 为例，适用于处理器架构为 X86，操作系统为 RedHat 64 位，最低版本支持 7。

解压${com.bes.mq.installRoot}/lib/client/go 下的对应包并按照使用说明进行操作

以下两节说明了如何使用 BES MQ Go 客户端发送和接收消息。

4.6.1 发送消息

```
package main


import (
    . "besmq"
    "fmt"
)
const (
```

```
        BESMQ_URI string = "tcp://localhost:3200"
        BESMQ_QUEUE_NAME string = "testQueue"
    )
    func sendBytesMsg(hdest *BQMCHdest, err *BMQCErr) {
        var msgInfo *BQMCMsgInfo = &BQMCMsgInfo{}
        var msgOpt  *BQMCMsgOpt = &BQMCMsgOpt{}
        var hprops  *BQMCHprops = &BQMCHprops{}
        var testMsg []byte = []byte {0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08,
0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f}

        msgInfo.BMQCInitMsgInfo()
        msgInfo.BMQCType = BQMCMsgType(BMQC_BYTES_MESSAGE)
        msgInfo.ContsLen = uint32(len(testMsg))

        msgOpt.BMQCInitMsgOpt()

        hdest.BMQCPut(msgInfo, hprops, testMsg, msgOpt, err)
    }

    func main() {
        var ctx *BQMCContext = new(BQMCContext)
        var err *BMQCErr = new(BMQCErr)
        var connOpt *BQMCConnOpt = new(BQMCConnOpt)
        var conn *BQMCHconn = &BQMCHconn {}
        var dest *BQMCHdest = &BQMCHdest{}
        var destOpt *BQMCDestOpt = &BQMCDestOpt{}

        fmt.Println("Begin...")
    /*
     1. Initialize the BES MQ C library resources
    */
```

```
        ctx.BMQCInitContext()
        ctx.BMQCInit(err)
        if err.ErrNum != RET_SUCCESS {
            fmt.Println("BMQCInit failed! ErrNum: ", err.ErrNum, "ErrName: ", err.ErrName, "ErrDesc: ", err.ErrName)
            return
        }

        fmt.Println("Initialize the BES MQ C library finished")
    /*
     2. Open the connection to the broker
    */
        connOpt.BMQCInitConnOpt()
        conn.BQMConn(BESMQ_URI, connOpt, err)
        if err.ErrNum != RET_SUCCESS {
            fmt.Println("BQMConn failed! ErrNum: ", err.ErrNum, "ErrName: ", err.ErrName, "ErrDesc: ", err.ErrName)
            return
        }
        fmt.Println("Open the connection to the broker [ ",  BESMQ_URI, " ] finished")
    /*
     3. Open the queue
    */
        destOpt.BMQCInitDestOpt()
        destOpt.DestType = BQMCDestType(BMQC_QUEUE)
        conn.BMQCOpen(dest, BESMQ_QUEUE_NAME, destOpt, err)
        if err.ErrNum != RET_SUCCESS {
            fmt.Println("BMQCOpen failed! ErrNum: ", err.ErrNum, "ErrName: ", err.ErrName, "ErrDesc: ", err.ErrName)
            return
        }
```

```
        fmt.Println("Open the queue [", BESMQ_QUEUE_NAME, "] finished")
    /*
     4. Puts messages
    */
        sendBytesMsg(dest, err)
        if err.ErrNum != RET_SUCCESS {
            fmt.Println("sendBytesMsg failed! ErrNum: ", err.ErrNum, "ErrName: ",
err.ErrName, "ErrDesc: ", err.ErrName)
            return
        }
        fmt.Println("Send Bytes Message finished")

    /*
     5. Close the queue
    */
        dest.BMQCClose(err)
        if err.ErrNum != RET_SUCCESS {
            fmt.Println("BMQCClose failed! ErrNum: ", err.ErrNum, "ErrName: ",
err.ErrName, "ErrDesc: ", err.ErrName)
            return
        }

    /*
     6. Disconnect broker
    */
        conn.BMQCDisconn(err)
        if err.ErrNum != RET_SUCCESS {
            fmt.Println("BMQCDisconn failed! ErrNum: ", err.ErrNum, "ErrName: ",
err.ErrName, "ErrDesc: ", err.ErrName)
            return
        }
```

```
    /*
     7. Destroy the BES MQ C library resources
    */
        ctx.BMQCDestroy(err)
        if err.ErrNum != RET_SUCCESS {
            fmt.Println("BMQCDestroy failed! ErrNum: ", err.ErrNum, "ErrName: ",
err.ErrName, "ErrDesc: ", err.ErrName)
            return
        }

        fmt.Println("Done")
}
```

4.6.2 接收消息

```
    package main

    import (
        . "besmq"
        "fmt"
    )

    const (
        BESMQ_URI string = "tcp://localhost:3200"
        BESMQ_QUEUE_NAME string = "testQueue"
        TEST_BYTES_RECV_TIMEOUT uint32 = 5 * 1000
    )

    func getBytesMsg(hdest *BQMCHdest, err *BMQCErr) {
        var msgInfo *BQMCMsgInfo = &BQMCMsgInfo{}
        var msgOpt  *BQMCMsgOpt = &BQMCMsgOpt{}
```

```
        var hprops  *BQMCHprops = &BQMCHprops{}
        var content []byte = make([]byte, 0, 0)

        msgOpt.OperateType = BQMCMsgOPType(BMQC_GET)
        msgOpt.GetTimeout = TEST_BYTES_RECV_TIMEOUT
        hdest.BMQCGet(msgInfo, hprops, &content, msgOpt, err)

        fmt.Printf("Receive %d bytes: %#v", msgInfo.ContsLen, content)
        fmt.Println()
    }

    func main() {
        var ctx *BQMCContext = new(BQMCContext)
        var err *BMQCErr = new(BMQCErr)
        var connOpt *BQMCConnOpt = new(BQMCConnOpt)
        var conn *BQMCHconn = &BQMCHconn {}
        var dest *BQMCHdest = &BQMCHdest{}
        var destOpt *BQMCDestOpt = &BQMCDestOpt{}

        fmt.Println("Begin...")
    /*
     1. Initialize the BES MQ C library resources
    */
        ctx.BMQCInitContext()
        ctx.BMQCInit(err)
        if err.ErrNum != RET_SUCCESS {
            fmt.Println("BMQCInit failed! ErrNum: ", err.ErrNum, "ErrName: ",
err.ErrName, "ErrDesc: ", err.ErrName)
            return
        }
```

```
        fmt.Println("Initialize the BES MQ C library finished")
    /*
     2. Open the connection to the broker
    */
        connOpt.BMQCInitConnOpt()
        conn.BQMConn(BESMQ_URI, connOpt, err)
        if err.ErrNum != RET_SUCCESS {
            fmt.Println("BQMConn failed! ErrNum: ", err.ErrNum, "ErrName: ", err.ErrName, "ErrDesc: ", err.ErrName)
            return
        }
        fmt.Println("Open the connection to the broker [ ",  BESMQ_URI, " ] finished")
    /*
     3. Open the queue
    */
        destOpt.BMQCInitDestOpt()
        destOpt.DestType = BQMCDestType(BMQC_QUEUE)
        conn.BMQCOpen(dest, BESMQ_QUEUE_NAME, destOpt, err)
        if err.ErrNum != RET_SUCCESS {
            fmt.Println("BMQCOpen failed! ErrNum: ", err.ErrNum, "ErrName: ", err.ErrName, "ErrDesc: ", err.ErrName)
            return
        }
        fmt.Println("Open the queue [", BESMQ_QUEUE_NAME, "] finished")
    /*
     4. Puts messages
    */
        getBytesMsg(dest, err)
        if err.ErrNum != RET_SUCCESS {
            fmt.Println("sendBytesMsg failed! ErrNum: ", err.ErrNum, "ErrName: ", err.ErrName, "ErrDesc: ", err.ErrName)
```

```
            return
        }
        fmt.Println("Send Bytes Message finished")

    /*
     5. Close the queue
    */
        dest.BMQCClose(err)
        if err.ErrNum != RET_SUCCESS {
            fmt.Println("BMQCClose failed! ErrNum: ", err.ErrNum, "ErrName: ",
err.ErrName, "ErrDesc: ", err.ErrName)
            return
        }

    /*
     6. Disconnect broker
    */
        conn.BMQCDisconn(err)
        if err.ErrNum != RET_SUCCESS {
            fmt.Println("BMQCDisconn failed! ErrNum: ", err.ErrNum, "ErrName: ",
err.ErrName, "ErrDesc: ", err.ErrName)
            return
        }

    /*
     7. Destroy the BES MQ C library resources
    */
        ctx.BMQCDestroy(err)
        if err.ErrNum != RET_SUCCESS {
            fmt.Println("BMQCDestroy failed! ErrNum: ", err.ErrNum, "ErrName: ",
err.ErrName, "ErrDesc: ", err.ErrName)
            return
```

```
        }


        fmt.Println("Done")
}
```

4.6.3 编译

1、 将测试程序放于

${BESMQ_HOME}/lib/client/go/besmq-go-client/besmq-go-client/src 目录下；

2、设置 GOPATH 为${BESMQ_HOME}/lib/client/go/besmq-go-client/besmq-go-client；

3、在 src 目录下执行编译即可：go build -o xxx xxx.go

4.7 开发 Python 客户端

Python 客户端依赖如下目录下的文件：

1. ${com.bes.mq.installRoot}/lib/client/python

2. ${com.bes.mq.installRoot}/lib/client/go

解压${com.bes.mq.installRoot}/lib/client/Go 的对应包并按照使用说明进行操作，并 pip install besmq-client.tar.gz 配置 python 编译环境

以下两节说明了如何使用 BES MQ Python 客户端发送和接收消息。

4.7.1 发送消息

```
    from besmq.client import BesMQClient, bmqc_init_context, bmqc_init, bmqc_destroy
    from besmq.consts import DEST_TYPE, MSG_INFO_TYPE
    from besmq.ffi import BMQC_DEST_OPT, BMQC_MSG_INFO
    try:
        # bmqc 初始化上下文,只能初始化一次
        bmqc_init_context()
        # bmqc 初始化，只能初始化一次
        bmqc_init()
        #bmqc 客户端初始化，必须放在 bmqc_init_context 和 bmqc_init()后面
        client = BesMQClient()
```

```
# bmqc 初始化 connection 参数选项
client.bmqc_init_conn_opt()
# bmqc 连接 besmq 服务端
client.bmqc_conn("tcp://127.0.0.1:3200")

# bmqc 初始化目标（队列）参数选项
client.bmqc_init_dest_opt()
dest_opt = BMQC_DEST_OPT()
dest_opt.dest_type = DEST_TYPE.BMQC_QUEUE
client.set_dest_opt(dest_opt)

# bmqc 打开目标（队列） （先要创建队列）
client.bmqc_open("queue_sample")

# bmqc 初始化消息参数
client.bmqc_init_msg_info()
msg_info = BMQC_MSG_INFO()
msg_info.type = MSG_INFO_TYPE.BMQC_TEXT_MESSAGE
msg_info.conts_len = 20  # 设置消息最大长度
client.set_msg_info(msg_info)

# bmqc 初始化消息参数选项
client.bmqc_init_msg_opt()

# bmqc 发送消息
client.bmqc_put("hello,queue")
# bmqc 关闭目标(队列)
client.bmqc_close()
# bmqc 关闭连接
client.bmqc_disconn()
# bmqc destroy，只能销毁一次
```

```
        bmqc_destroy()
    except Exception as e:
    print(e)
```

4.7.2 接收消息

```
    from besmq.client import BesMQClient, bmqc_init_context, bmqc_init, bmqc_destroy
    from besmq.consts import DEST_TYPE, MSG_OPT_OPERATE_TYPE, MSG_INFO_TYPE
    from besmq.ffi import BMQC_DEST_OPT, BMQC_MSG_OPT


    try:
        # bmqc 初始化上下文,只能初始化一次
        bmqc_init_context()
        # bmqc 初始化，只能初始化一次
        bmqc_init()
        # bmqc 客户端初始化，必须放在 bmqc_init_context 和 bmqc_init()后面
        client = BesMQClient()
        #bmqc 初始化 connection 参数选项
        client.bmqc_init_conn_opt()
        #bmqc 连接 besmq 服务端
        client.bmqc_conn("tcp://127.0.0.1:3200")

        #bmqc 初始化目标（队列）参数选项
        client.bmqc_init_dest_opt()
        dest_opt = BMQC_DEST_OPT()
        dest_opt.dest_type = DEST_TYPE.BMQC_QUEUE
        client.set_dest_opt(dest_opt)

        #bmqc 打开目标（队列） （先要创建队列）
        client.bmqc_open("queue_sample")
```

```
        #bmqc 初始化消息参数选项
        client.bmqc_init_msg_opt()
        msg_opt = BMQC_MSG_OPT()
        msg_opt.operate_type = MSG_OPT_OPERATE_TYPE.BMQC_GET
        msg_opt.get_timeout = 5 * 1000
        client.set_msg_opt(msg_opt)

        #bmqc 获取消息
        msg_info =  client.bmqc_get()

        #如果获取的消息是字节消息，转换成 str 并打印
        if client.get_msg_info().type  == MSG_INFO_TYPE.BMQC_TEXT_MESSAGE:
            msg_opt_a = msg_info.decode('utf-8')
            print(msg_opt_a)

        #bmqc 关闭目标（队列）
        client.bmqc_close()
        #bmqc 关闭连接
        client.bmqc_disconn()
        # bmqc destroy，只能销毁一次
        bmqc_destroy()

    except Exception as e:
    print(e)
```

4.7.3 编译

使用 python/python3 xxx.py 即可