

精通 WebSphere MQ

-- Mastering WebSphere MQ

目录

目录.....	2
内容提要.....	13
前言.....	13
从系统集成到系统整合.....	13
消息驱动和消息触发.....	15
记号约定.....	15
第 1 章 概念与原理.....	16
1.1 简介.....	16
1.1.1 消息中间件.....	16
1.1.2 WebSphere MQ.....	17
1.1.3 WebSphere MQ 产品.....	17
1.2 概念与对象.....	19
1.2.1 消息 (Message).....	19
1.2.2 队列 (Queue).....	20
1.2.3 队列管理器 (Queue Manager).....	23
1.2.4 通道 (Channel).....	24
1.2.5 名称列表 (Name List).....	26
1.2.6 分发列表 (Distribution List).....	26
1.2.7 进程定义 (Process).....	27
1.2.8 认证信息 (Auth Info).....	27
1.2.9 客户端和服务端 (Client & Server).....	27
1.2.10 操作界面 (MQ Interface).....	27
1.2.11 应用程序 (MQ Application).....	28
1.3 工作原理.....	28
1.3.1 PUT 和 GET.....	28
1.3.2 协同工作.....	29
1.3.3 互连通信.....	29
第 2 章 安装.....	31
2.1 安装环境.....	32
2.1.1 硬件.....	32
2.1.2 操作系统.....	32
2.1.3 通信协议.....	33
2.2 安装介质.....	33
2.2.1 正版.....	33
2.2.2 试用版.....	33
2.3 安装过程.....	33
2.4 缺省配置.....	36
2.4.1 准备 WebSphere MQ 向导.....	36
2.4.2 远程管理向导.....	37

2.4.3	缺省配置向导.....	38
2.5	安装补丁.....	40
2.6	其它平台.....	40
2.6.1	AIX	40
2.6.2	HP-UX	41
2.6.3	Solaris	43
2.6.4	Linux	45
2.7	安装目录.....	46
2.7.1	Windows	47
2.7.2	AIX	47
2.8	安装文档.....	48
第 3 章	控制与管理.....	48
3.1	MQ 控制命令	49
3.1.1	MQ 队列管理器控制.....	49
3.1.2	MQ 命令服务器控制.....	51
3.1.3	MQ 监听器控制.....	52
3.1.4	MQ 触发监控器控制.....	54
3.1.5	小结.....	55
3.2	MQ 对象管理	55
3.2.1	队列管理器管理	57
3.2.2	队列管理.....	57
3.2.3	通道管理.....	58
3.2.4	进程定义管理.....	60
3.2.5	名称列表管理.....	60
3.2.6	认证信息管理.....	61
3.2.7	小结.....	61
3.3	基本队列操作.....	62
3.4	MQ 配置信息	63
3.4.1	UNIX 配置文件	63
3.4.2	Windows 注册表.....	64
3.4.3	Windows 中 MQ 运行环境配置.....	65
3.4.4	Windows 中 MQ 队列管理器配置.....	66
3.5	MQ 管理方式	67
3.5.1	本地管理.....	68
3.5.2	远程管理.....	69
3.6	日志 (Log).....	72
3.6.1	队列管理器日志	72
3.6.2	检查点 (Checkpoint).....	74
3.6.3	记录和复原 (Record & Recover).....	75
3.6.4	备份和恢复 (Backup & Restore).....	76
3.6.5	导出日志 (Dump Log).....	77
第 4 章	通信与配置.....	77
4.1	消息路由.....	77
4.4.1	消息路由过程.....	78

4.4.2	缺省传输队列.....	78
4.4.3	队列管理器别名.....	78
4.4.4	多级跳.....	79
4.4.5	传输中的消息.....	79
4.2	通道配置.....	81
4.2.1	Sender (QM1) -- Receiver (QM2).....	81
4.2.2	Server (QM1) -- Receiver (QM2).....	82
4.2.3	Server (QM1) -- Requester (QM2).....	83
4.2.4	Sender (QM1) -- Requester (QM2).....	84
4.2.5	通道启动命令.....	85
4.2.6	通道监控程序.....	85
4.3	通道的属性.....	86
4.3.1	通道会话.....	87
4.3.2	通道协议.....	89
4.4	通道的状态.....	91
4.4.1	公共状态 (Common Status).....	91
4.4.2	当前状态 (Current-Only Status).....	92
4.4.3	通道状态分析.....	93
4.5	互连配置举例.....	95
4.5.1	单向传送.....	95
4.5.2	双向传送.....	96
4.5.3	队列与队列管理器别名.....	97
4.5.4	三级跳 (Multi-hopping).....	98
4.5.5	四级跳 (Multi-hopping).....	100
第 5 章	应用设计.....	102
5.1	架构设计.....	102
5.1.1	两点间通信.....	102
5.1.2	多点间通信.....	103
5.1.3	同步和异步.....	103
5.1.4	Client/Server.....	104
5.1.5	Internet 通信.....	105
5.2	通信方式设计.....	106
5.2.1	进程间会话模式.....	106
5.2.2	系统间通信方式.....	107
5.3	并发设计.....	108
5.3.1	多读多写.....	108
5.3.2	共享与独占.....	108
5.3.3	对象绑定.....	109
5.3.4	队列管理器关闭.....	109
5.3.5	分发列表 (Distribution List).....	109
5.4	消息设计.....	110
5.4.1	消息大小 (Message Size).....	110
5.4.2	消息持久性 (Persistence).....	111
5.4.3	消息优先级 (Priority).....	112

5.4.4	消息超时 (Expiry).....	113
5.5	发送设计.....	113
5.5.1	消息标识.....	113
5.5.2	消息类型.....	114
5.5.3	消息格式.....	114
5.5.4	应答队列.....	115
5.5.5	动态队列.....	116
5.5.6	用户替换.....	118
5.6	读取设计.....	118
5.6.1	等待读取 (Wait & NoWait).....	118
5.6.2	信号中断 (Signal).....	118
5.6.3	截断消息 (Truncated Message)	119
5.6.4	浏览消息 (Browse).....	119
5.6.5	格式转换 (Convert)	120
5.6.6	消息匹配 (Match).....	120
5.6.7	回滚计数 (Backout Count).....	121
5.6.8	固化回滚计数 (Harden Backout).....	121
5.7	容错设计.....	122
5.7.1	出错处理.....	122
5.7.2	报告消息.....	122
5.7.3	死信消息.....	122
5.8	小结.....	123
第 6 章	消息处理	123
6.1	交易 (Transaction).....	123
6.1.1	概述.....	124
6.1.2	本地交易 (Local LUW).....	124
6.1.3	全局交易 (Global LUW)	125
6.2	触发 (Trigger).....	131
6.2.1	原理.....	131
6.2.2	触发方式.....	132
6.2.3	配置.....	133
6.2.4	触发过程.....	134
6.2.5	并发.....	135
6.2.6	通道触发.....	135
6.2.7	触发 CICS 交易.....	137
6.3	报告 (Report).....	138
6.3.1	原理.....	138
6.3.2	选项.....	139
6.3.3	说明.....	140
6.4	分组与分段 (Group & Segment).....	141
6.4.1	消息组的发送.....	142
6.4.2	消息组的接收.....	143
6.5	消息上下文 (Message Context).....	144
6.5.1	消息上下文的内容.....	145

6.5.2	消息上下文的编程.....	146
6.6	死信处理 (DLQ Handler).....	147
6.6.1	死信消息.....	147
6.6.2	死信队列处理器.....	148
6.7	数据转换 (Data Convert).....	151
6.7.1	转换方式.....	154
6.7.2	数据转换表 (Convert Table).....	157
第 7 章	广播通信.....	159
7.1	分发列表 (Distribution List).....	159
7.1.1	概念.....	159
7.1.2	配置举例.....	161
7.1.3	编程.....	162
7.2	发布和订阅 (Pub & Sub).....	168
7.2.1	概念.....	168
7.2.2	安装.....	169
7.2.3	Broker 控制命令.....	169
7.2.4	Broker 网络.....	171
7.2.5	编程设计.....	173
7.2.6	发布/订阅命令.....	175
7.2.7	常见的问题.....	178
第 8 章	客户端.....	179
8.1	配置.....	179
8.1.1	Server 端配置.....	180
8.1.2	Client 端配置.....	180
8.2	用户出口.....	185
8.2.1	用户出口路径.....	186
8.2.2	排错.....	186
8.3	安全检查.....	186
8.4	触发 (Trigger).....	188
8.5	跟踪 (Trace).....	188
8.5.1	Windows.....	189
8.5.2	AIX.....	189
第 9 章	群集.....	189
9.1	相关概念.....	190
9.1.1	配置库 (Repository).....	190
9.1.2	配置库队列管理器 (Repository Queue Manager).....	190
9.1.3	群集通道 (Cluster Channel).....	191
9.1.4	群集队列 (Cluster Queue).....	191
9.1.5	群集传输队列 (Cluster transmission queue).....	191
9.2	群集管理.....	191
9.2.1	对象属性.....	191
9.2.2	管理命令.....	192
9.2.3	管理任务举例.....	193
9.3	群集配置举例.....	194

9.3.1	例 1	194
9.3.2	例 2	196
9.3.3	例 3	198
9.3.4	例 4	201
9.3.5	例 5	203
9.3.6	例 6	205
9.4	多群集队列实例与共享队列组	208
9.5	群集负载用户出口 (Cluster Workload User Exit).....	210
第 10 章	监控与性能	210
10.1	事件 (Event)	210
10.1.1	概念.....	210
10.1.2	队列管理器事件 (Queue Manager Event).....	211
10.1.3	通道事件 (Channel Event).....	214
10.1.4	性能事件 (Performance Event).....	215
10.1.5	配置事件 (Configuration Event).....	218
10.1.6	事件消息.....	219
10.1.7	事件监控.....	219
10.1.9	实验一：Queue Depth	220
10.1.10	实验二：Queue Service Interval	221
10.2	性能设计 (Performance).....	222
10.2.1	队列管理器性能比较	222
10.2.2	数据传递的性能比较	230
10.2.3	性能优化.....	232
10.2.4	小结.....	234
第 11 章	安全协议.....	234
11.1	安全通信.....	235
11.1.1	数据加密.....	235
11.1.2	报文摘要.....	236
11.1.3	数字签名.....	236
11.1.4	SSL.....	237
11.2	数字证书.....	239
11.2.1	概念.....	239
11.2.2	格式.....	239
11.2.3	根签证书与自签证书	241
11.3	WebSphere MQ 配置 SSL.....	241
11.3.1	Server/Server 消息通道	241
11.3.2	Client/Server MQI 通道	243
11.3.3	SSL 相关的对象属性	245
11.3.4	Client 端程序.....	246
11.3.5	证书部署.....	246
11.4	实例 1 根签证书.....	247
11.4.1	准备证书.....	247
11.4.2	配置队列管理器	249
11.4.3	配置通道.....	251

11.5	实例 2 自签证书.....	251
11.5.1	准备证书.....	251
11.5.2	配置队列管理器	254
11.5.3	配置通道.....	254
第 12 章	用户出口.....	255
12.1	概述.....	255
12.2	Channel Exit	256
12.2.1	Channel Exit 函数	258
12.2.2	Security Exit	259
12.2.3	Message Exit	263
12.2.4	Send Exit	264
12.2.5	Receive Exit	267
12.2.6	Message Retry Exit	268
12.2.7	Channel Auto-Definition Exit	270
12.2.7	Transport-Retry Exit	271
12.3	Data Conversion Exit	272
12.4	Cluster Workload Exit	274
12.5	Pub/Sub Routing Exit	276
12.6	MQ API Exit	277
12.6.1	设置.....	278
12.6.2	举例.....	279
12.6.3	编程设计.....	279
第 13 章	MQI 编程.....	280
13.1	编程入门.....	280
13.1.1	数据类型.....	280
13.1.2	数据结构.....	280
13.1.3	程序流程.....	281
13.1.4	例程.....	283
13.2	头文件	284
13.3	库文件	284
13.4	编程参考.....	285
13.4.1	MQCONN	285
13.4.2	MQCONNEX.....	285
13.4.3	MQDISC.....	285
13.4.4	MQOPEN.....	286
13.4.5	MQCLOSE.....	286
13.4.6	MQPUT	287
13.4.7	MQPUT1.....	287
13.4.8	MQGET	287
13.4.9	MQINQ.....	288
13.4.10	MQSET.....	288
13.4.11	MQBEGIN	289
13.4.12	MQCMIT	289
13.4.13	MQBACK	289

第 14 章 Java 编程.....	290
14.1 安装.....	290
14.2 编程设计.....	291
14.2.1 例程.....	292
14.3 连接模式.....	292
14.4 用户出口.....	293
14.5 多线程.....	294
14.6 连接池.....	295
14.6.1 例 1：线程之间串行建立连接.....	296
14.6.2 例 2：线程之间并行建立连接.....	297
14.7 交易保护.....	298
14.7.1 本地交易 (Local LUW).....	298
14.7.2 全局交易 (Global LUW).....	298
14.8 Trace.....	299
第 15 章 JMS 编程.....	299
15.1 JMS 对象.....	299
15.1.1 Context.....	301
15.1.2 ConnectionFactory.....	301
15.1.3 Connection.....	302
15.1.4 Session.....	303
15.1.5 MessageConsumer.....	303
15.1.6 MessageProducer.....	304
15.1.7 MessageListener.....	305
15.1.8 Message.....	306
15.2 编程设计.....	308
15.2.1 Persistence.....	308
15.2.2 Priority.....	308
15.2.3 Expiry.....	308
15.2.4 Transaction.....	309
15.2.5 Acknowledgment.....	310
15.2.6 Message Seletor.....	311
15.2.7 Temporary Destination.....	312
15.2.8 Durable Subscriber.....	312
15.3 MQ JMS 运行环境.....	313
15.3.1 JMS Interface 与 MQ JMS Object.....	313
15.3.2 JNDI.....	313
15.3.3 Client.....	317
15.3.4 CCSID & Encoding.....	318
15.4 ASF.....	319
第 16 章 ActiveX 编程.....	320
16.1 MQAX.....	320
16.1.1 程序设计.....	320
16.1.2 编程参考.....	323
16.1.3 跟踪信息 (Trace).....	328

16.2	MQAI	328
16.3	ADSI	328
第 17 章	AMI 编程.....	329
17.1	安装.....	330
17.1.1	Windows	330
17.1.2	AIX	330
17.2	概念与配置	330
17.2.1	概念.....	330
17.2.2	配置.....	331
17.3	C 编程.....	332
17.3.1	Object Level	332
17.3.2	High Level.....	338
17.4	Java 编程.....	343
第 18 章	PCF & AI 编程.....	344
18.1	PCF 编程.....	344
18.1.1	消息流程.....	344
18.1.2	消息格式.....	345
18.1.3	格式举例.....	347
18.2	AI 编程.....	349
18.2.1	消息流程.....	349
18.2.2	包的组成.....	350
18.2.3	编程.....	351
附录	WebSphere MQ 进程一览表.....	358
	Windows 平台.....	358
	UNIX 平台	358
	进程树	359
附录	WebSphere MQ 命令一览表.....	359
	队列管理器 (Queue Manager).....	359
	crtmqm 创建队列管理器 (Create Queue Manager)	359
	dlmqm 删除队列管理器 (Delete Queue Manager)	360
	strmqm 启动队列管理器 (Start Queue Manager)	360
	endmqm 停止队列管理器 (End Queue Manager).....	361
	dspmqr 显示队列管理器 (Display Queue Manager)	361
	命令服务器 (Command Server)	362
	strmqcvs 启动命令服务器 (Start Command Server).....	362
	endmqcvs 停止命令服务器 (End Command Server).....	362
	dspmqcvs 显示命令服务器 (Display Command Server).....	362
	Listener (监听器)	362
	runmqtsr 运行监听器 (Run Listener).....	362
	endmqtsr 停止监听器 (End Listener).....	363
	触发监控器 (Trigger Monitor)	363
	runmqtrmc 启动 Client 端触发监控器 (Run Trigger Monitor for Client).....	363
	runmqtrm 启动 Server 端触发监控器 (Run Trigger Monitor for Server).....	363
	Trace.....	364

strmqtrc	启动 Trace (Start Trace , Windows 平台).....	364
strmqtrc	启动 Trace (Start Trace , HP-UX , Solaris , Linux 平台)	364
endmqtrc	停止 Trace (End Trace , Windows 平台).....	365
endmqtrc	停止 Trace (End Trace , HP-UX , Solaris , Linux 平台).....	365
dspmqtrc	显示 Trace (Display Trace , HP-UX , Solaris , Linux 平台)	365
介质恢复 (Media Recover).....		365
rcdmqimg	记录对象映像 (Record Object Image).....	365
rcrmqobj	重建对象 (Recreate Object).....	366
日志 (Log).....		367
dmpmqlog	输出格式化日志.....	367
容量单元 (Capacity).....		367
dspmqcap	显示容量单元 (Display Capacity).....	367
setmqcap	设置容量单元 (Set Capacity)	367
权限信息 (Authority).....		368
dmpmqaut	输出权限信息 (Dump Authority)	368
dspmqaut	显示权限信息 (Display Authority)	368
setmqaut	设置权限信息 (Set Authority).....	369
amqoamd	输出授权信息 (OAM Dump)	371
运行环境 (Environment).....		371
mqver	显示版本 (WebSphere MQ Version).....	371
setmqprd	设置生产环境 (Set Production)	371
amqicsdn	安装补丁 (Install CSD)	372
高可用性 (High-Avalability , Windows 平台).....		372
hadltmqm	删除队列管理器 (HA Delete Queue Manager)	372
hamvmqm	移动队列管理器 (HA Move Queue Manager).....	372
haregtyp	注册队列管理器 (HA Register Type)	372
amqmsysn	检查模块版本信息 (System Check).....	373
高可用性 (High-Avalability , 其它平台)		373
疑问交易 (In-Doubt Transaction)		373
dspmqtrn	显示疑问交易	373
rsvmqtrn	解决疑问交易.....	373
消息 (Message)		374
amqsput	往队列中放消息 (Server 程序).....	374
amqsputc	往队列中放消息 (Client 程序).....	374
amqsget	从队列中取消息 (Server 程序).....	374
amqsgetc	从队列中取消息 (Client 程序).....	374
工具 (Utility)		375
runmqsc	脚本命令服务器 (Run MQSC).....	375
mqrc	原因码查询 (MQ Reason Code)	375
amqfirst	MQ 第一步, 仅 Window 平台.....	376
amqapi	API 试验程序, 仅 Windows 平台.....	376
amqpcard	MQI 明信片程序, 仅 Windows 平台	376
amqmtbrn	MQ Task Bar , 仅 Windows 平台.....	376
amqmjpse	准备 MQ 向导, 仅 Windows 平台	377

amqmgse	MQ 缺省配置	377
amqinfon	MQ 信息中心文档 (MQ Info Center)	377
crtmqcvx	创建数据转换程序框架 (Create Conversion)	377
runmqdlq	运行死信队列处理器 (Run Dead-Letter Queue Handler)	378
runmqchi	运行通道初始化程序 (Run Channel Initiator)	379
runmqchl	运行通道 (Run Channel)	379
dspmqls	显示对象对应的文件名 (Display Files)	379
setmqscp	设置服务连接点 (Set Service Connection Point, 仅 Windows 平台) ...	380
setmqcrl	设置无效论证列表 (Set Certificate Revocation List (CRL) LDAP Server Definitions, 仅 Windows 平台)	380
amqmcert	Client 证书配置工具 (Utility for Certification)	380
ffstsummary	FFST 文件摘要 (FFST Summary)	381
mqaxlev	显示 Code Level	382
amqrfdm	查询 MQ Cluster Repository	382
amquregn	Registry 值列表工具	382
amqmdain	MQ 服务控制命令, 仅 Windows 平台	383
amqmsrvn	COM 服务器, 仅 Windows 平台	384
附录 MQSC 命令一览表		385
RUNMQSC		385
执行脚本		385
抑制回显		385
检验脚本		385
远程管理		385
批处理		385
MQSC 命令		386
结构图		386
DEFINE		389
DELETE		395
ALTER		397
DISPLAY		399
CLEAR		406
START		407
STOP		407
RESOLVE		408
PING		408
RESET		408
REFRESH		409
SUSPEND		409
RESUME		410
参考书目		410

内容提要

IBM WebSphere MQ 是一个优秀的消息中间件,它被广泛地应用于各种企业应用系统之间的互连,已经逐渐成为这方面的标准。本书从原理到实践全面系统地阐述了 IBM WebSphere MQ 产品的安装、配置、管理、设计、编程等各个方面,同时也介绍了产品的扩展功能和一些高级使用技巧。本书从功能上重点介绍了日志管理、死信处理、客户端、群集、交易、触发、报告、事件、分段与分组、分发列表、发布订阅、数据转换、用户出口、安全套接字、性能等等。

全书覆盖了 WebSphere MQ 产品的所有相关知识,全文共分 18 章。第 1-2 章为基础部分,介绍了 WebSphere MQ 产品的原理和简单的安装过程。第 3-4 章为管理部分,介绍产品的控制、管理及配置。第 5-12 章为设计部分,介绍了应用设计中可能用到的各种产品高级功能和使用技巧。第 13-18 章为编程部分,讲解了各种编程模式和方法。

对于 WebSphere MQ 的初学者和使用者,可以从本书的第一和第二部分入手,通过深入的原理剖析和详细的管理操作,相信能够帮助这部分读者入门与提高。即使对与 MQ 无关的人员,也能够在通读本书后对这类软件的设计思路和工作原理有一定的了解和启发。第三部分和第四部分是本书的精华,介绍了大量的高级功能与技巧,内含了作者多年的经验积累和实例模型,对于 WebSphere MQ 设计和编程人员会有相当的吸引力,可以作为有一定经验者的高级读物,也是相关开发人员必不可少的参考书。

本书注重实践,附有大量例程,帮助读者在实践中加深理解,也为相关设计和开发人员提供了丰富的参考样例。所有例程都在 WebSphere MQ 5.3 环境下经过测试,供读者参考。全书语言生动并附有很多插图,易于理解。在专业相关的文字叙述上力求简捷,在内容与过程的安排上则力争翔实,使得读者能够容易地自己动手实践。相信能帮助读者的 WebSphere MQ 水平有所提高,从入门到精通。

由于编者水平所限,不足之处在所难免,恳请广大的读者不吝指正。

作者

前言

从系统集成到系统整合

当今的 I/T 世界瞬息万变,各种应用技术层出不穷。企业的 I/T 系统在经过较长时间的建设后,往往会出现系统复杂、数据或功能重复、结构臃肿、难以互联的问题。企业的发展

需要创新出不断的业务，新业务可以通过新建系统来满足，也可以通过整合旧系统来满足。很多企业在经过一段时间的发展后会出现不少应用系统，各个系统之间往往互不相连或联系很少。每一次互连都需要单独设立一个项目，将双方的应用系统改造一番。有些应用系统本身又不断地采用新技术进行改造，如果业务本身没有发生变化，那只能是“新瓶装旧酒”。

我们在每次改造的时候，往往会加入一些新技术，与此同时也增加了一些复杂度。有很多的企业新项目并不完全是为了增加新功能，在很大程度上是原有系统功能上的覆盖，这样一来，原来投资就白白浪费了。众多的企业项目往往是年年要改造，系统在诞生之初就面临着一年一小改，两年一大改，三年一替换的窘境。系统的稳定运行受到干扰，设计效益很难达到。这在很大程度上是因为系统建设的思路存在问题。人们往往倾向于追求一种新技术来创建一套近乎完美的能解决所有问题的复杂系统，这种系统在结构上具有高度的聚合集成能力。

新技术在出现之后往往表现为两种截然不同的方式：一种方式是标新立异，与之前的所有现存技术完全不同，甚至格格不入。技术的簇拥者有时会构想出全方位或大一统的理想境界。但实际上，接受它的同时也意味着对老技术的否定，且往往意味着较大的风险与投资，经过一段时间的锤炼后，人们发现所谓“一统天下”的状态只是短暂的，很快又有无法解决的问题不断涌现，为了解决新的问题，系统又很快会出现很多例外。这就好像在宫殿边上进行违章搭建，让人感觉不快。随着搭建的增多，宫殿的原貌被破坏殆尽。

另一种方式是改良更新，脱胎于旧技术又不同旧技术，过去的概念和方法仍然有所体现，这种技术往往比较容易被接受，技术的推广者往往勾勒出只要不断升级就能始终领先的画面。不幸的是，频繁地升级、迁移、更新所带来的不便与困扰一点也不比前一种少，而且通常经过了一段时间后它的发展会被原先的技术框架所限制，到了不得不“伤筋动骨”转型或面临淘汰的地步。

无论是哪一种方式都是应用需求更新领先于技术能力更新的表现。其实，在当今世界的各种复杂 I/T 系统中，永远是集中与分散并存、各种新老技术并存的局面。就像是一个有机的生物体，有能力进行自身的新陈代谢，自我更新，各个子系统各自分工明确，之间又具备有机的联系。

我们身处的世界就是这样多样化且不断变化更新的，要适应复杂多变的环境，系统整合是一个很好的办法。我们对系统的整合不应该关注于用一种技术将其全盘替换或更新，而是应该试图寻找一种灵活的办法，将它们有机地联系起来。换句话说，不是要改造各个系统，而是要加强各个系统之间的联系，从而更好地使用已有的子系统。

IBM WebSphere MQ 家族就是一个优秀的用于应用系统间联系的软件，它的原理其实很简单也很容易理解，就是消息传递。应用系统就像一群有机生物一样，它们之间可以各自活动，也可以相互交流。通过消息传递，将它们有效地联系起来，每一个应用系统都可以对外提供自身的功能，也可以要求其它应用系统作为它的下一处理环节，消息是应用系统之间请求、应答和中间结果的载体。这样一来，不断流动的消息将松耦合关系的应用系统串起来，它们之间的关系就变成了功能叠加。

消息驱动和消息触发

我们在构建一个应用系统的时候，往往会将其划分成多个模块，各个模块之间需要约定接口规范。对于消息驱动模块之间需要约定的是消息的报文格式、通信模式、功能定义。报文格式也就是双方模块都能理解的消息语言，比如 XML。对于跨网络、跨平台的消息，报文格式应该能够屏蔽双方信息编码上的差异（比如 ASCII 编码或 EBCDIC 编码），屏蔽双方信息表达上的差异（比如整数的高低字节，浮点数的表示，32 位/64 位的整数长短等等）。通信模式也就是双方通话的方式，比如是双向的一问一答方式，还是单向的汇报方式，是点对点方式，还是一对多广播方式。对于跨网络、跨平台的消息，通信模式还应该约定双方的网络层通信协议和应用层通信协议。功能定义也就是说模块在收到一条消息后应该做的相应动作。有了这些约定，模块之间就可以通过消息流转将各个模块的功能发挥出来，形成对外的业务功能。一旦要增加模块，原有的模块可以不需要改动。一旦要改变业务功能或业务流程，可能需要改变的只是消息流转的次序和方式。

消息驱动结构的系统中几乎所有的模块都在等待消息，在消息到来后进行相应的处理，处理结束后又回到这个点等待下一条消息。消息源可以是一个文件，也可以是系统队列、数据库、网络连接等等，可谓五花八门。监听消息的程序通常被称为监听器（Listener），如果每个模块有各自不同的监听器，则在系统空闲的时候，这些模块虽然不在工作，但也一直占用着系统资源。如果模块共同系统提供的监听器，则在监听器上可以设立触发机制（Trigger），可消息到达的时候来启动相应的模块进行处理，这样一来，在系统空闲的时候，只需要开启系统监听器即可，所有的工作模块都可以休息了。

IBM WebSphere MQ 家族就是基于消息驱动和消息触发原理设计的。底层是面向消息的中间件 WebSphere MQ，上层有面向消息整合的 WebSphere MQ Integrator 和面向消息流程的 WebSphere MQ Workflow 两个产品。本书只涉及 WebSphere MQ 产品。

记号约定

A.B

本书中用 A.B 表示对象 A 的属性 B，或者结构 A 中的域 B。如：Queue.CLUSNL 表示队列的 CLUSNL 属性，MQMD.Format 表示消息头 MQMD 结构中的 Format 域。

Config or Code

本书采用阴影来表示代码、数据结构、配置脚本等内容。

QM1:

DEFINE	QREMOTE	(QR_QM2)	+
	RNAME	(QL_QM2)	+
	RQMNAME	(QM2)	+
	XMITQ	(QLX_QM2)	+
	REPLACE		

<MQ_HKEY>

Windows 平台上的 WebSphere MQ 的所有属性设置都存放在注册表中，具体说来，在 `\\HKEY_LOCAL_MACHINE\SOFTWARE\IBM\MQSeries\CurrentVersion\` 路径之下。为了表达简便，我们将其称为 `<MQ_HKEY>`。

例如：`<MQ_HKEY>\Configuration\AllQueueManagers\` 表示

```
\\HKEY_LOCAL_MACHINE\SOFTWARE\IBM\MQSeries\CurrentVersion\Configuration\AllQueueManagers\
```

`<MQ_HKEY_Service>` 和 `<MQ_HKEY_QM>`

有时因为注册表路径太深，用 `<MQ_HKEY>` 仍觉得不方便，我们使用进一步的缩写。

`<MQ_HKEY_Service>` 表示 `<MQ_HKEY>\Configuration\Services\<QMgr>`

`<MQ_HKEY_QM>` 表示 `<MQ_HKEY>\Configuration\QueueManager\<QMgr>`

这里 `<QMgr>` 表示队列管理器名。

第 1 章 概念与原理

1.1 简介

计算机软件发展到今天，很多具有独立功能的应用模块都被逐渐隔离出来形成软件产品，这些软件往往是针对某一种应用需求，在相关的领域中具有很强的通用性。它们通常界于操作系统和应用程序之间，为应用程序提供一些标准的服务，我们称这一类软件为中间件。中间件根据其应用领域也分成多种，比如交易中间件、消息中间件、Web 中间件等等。

WebSphere MQ 本质上是一种消息中间件，用于保证异构应用之间的消息传递。应用程序通过 MQ 接口进行互连通信，可以不必关心网络上的通信细节，从而将更多的注意力集中于应用本身。MQ 在所有的平台上有统一的操作界面，这使得应用程序可以很方便地移植到各种操作系统中。

1.1.1 消息中间件

在早先的时候，人们往往使用两个程序之间直接通信的方式。这种办法最大的问题就是通信协议相关性，也就是说与通信协议相关的代码充斥在应用程序之中，而且可能出现在程序的任何地方，甚至影响程序的设计与结构。这种办法的另一个问题就是应用程序不容易写出可靠强壮的代码。应用程序的通信部分会因为工作方式的灵活性、网络协议的通用性，以及为实现一些实用功能变得非常庞大，往往超过应用程序本身的逻辑代码，变得本末倒置，且代码很难写好，也很难维护。

人们开始慢慢意识到应该把通信代码放到外面，变成独立的工作进程或工作模块，不同的工作进程可以适合同样的通信协议，而应用程序与通信程序之间使用通用的本地通信方式。这样一来，应用程序与通信程序的代码完全分开，各自的逻辑清晰自然，易于管理与维护，在通信方式上前进了一大步。但是，这种方式对通信程序的编程要求比较高，如果考虑到平台的广泛适用性，通信程序可能要写一大堆，要设计一个通用高效的本地通信接口也并非易事，再考虑到通信上的一些附加功能，其实现对普通编程人员是有一定困难的。人们很自

然地想到，这种工作最好交由专业的通信软件来完成。

于是，市场逐渐出现了专门负责消息通信的软件。它通常是一个独立运行的通信环境，有统一的编程调用接口，可以跨平台，跨协议。不同结点之间的软件可以通过配置相互连通，搭起统一的通信平台，从而出现更强大的功能。这种通信软件往往安全可靠、配置灵活，其地位在操作系统之上，在应用程序之下，所以被称为消息中间件（MOM）。WebSphere MQ 就是其中的一款。

1.1.2 WebSphere MQ

WebSphere MQ 是 IBM 公司于 2003 年 2 月推出的面向消息传递的中间件产品，是 IBM MQSeries 产品线的最新力作。目前的版本是 5.3，它的前身是 MQSeries 5.1 和 MQSeries 5.2。到了 5.3 版，产品的功能更丰富，更集成。与 IBM 公司的电子商务应用中间件 WebSphere 产品有很高的集成度和亲合性，从产品更名为 WebSphere MQ 这一点就可见一斑。事实上，WebSphere Application Server 5.0 版就可以内置安装一个嵌入式的 WebSphere MQ。

WebSphere MQ 在面向消息传递的中间件（Message Oriented Middleware, MOM）市场中赫赫有名，是用来连接异构平台之间企业应用的专业产品。通过 WebSphere MQ 可以屏蔽不同的通信协议之间的差别，可以最大程度地简化网络编程的复杂性。通过 MQ 的配置，通信双方的程序可以以松耦合的方式独自运行，并不关心对方所在的位置和状态，通过消息驱动或消息触发的方式来相互联系。它支持多种平台，对消息支持交易式的提交与回滚。

WebSphere MQ 产品主页为 <http://www.ibm.com/software/ts/mqseries>，相关的 WebSphere MQ 产品家族主页为：<http://www-3.ibm.com/software/integration/mqfamily/>。在相关网站上可以下载有效期为 90 天的 WebSphere MQ 试用版。

WebSphere MQ 的补丁被称为 CSD (Corrective Service Diskette)，比如 CSD01 就是一号补丁，后一号的补丁其内容完全覆盖前一号的补丁。WebSphere MQ 补丁可以在产品首页中找到，其下载网址为 <http://www-3.ibm.com/software/integration/mqfamily/support/summary/>。

WebSphere MQ 还有大量的免费资料可以下载，其中包括 Client 端软件、例程代码、相关工具、测评文档等等，内容十分丰富。IBM 公司称之为 MQ SupportPac，下载网址为：<http://www-3.ibm.com/software/integration/support/supportpacs/>。

1.1.3 WebSphere MQ 产品

IBM 公司将 WebSphere MQ 分成三个不同版本的产品：WebSphere MQ Express，WebSphere MQ，WebSphere MQ Extended Security Edition。其中，WebSphere MQ 是核心产品，本书也只针对这个产品进行介绍。

基本上，WebSphere MQ 包含了 WebSphere MQ Express 的全部功能，而 WebSphere MQ Extended Security Edition 又包含了 WebSphere MQ 的全部功能。它们的功能关系如图 1-1。

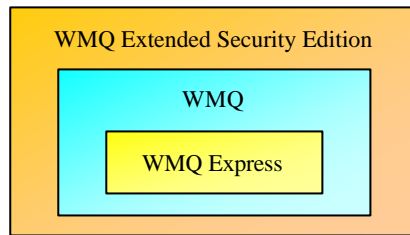


图 1-1 WebSphere MQ 产品

1.1.3.1 WebSphere MQ Express

WebSphere MQ Express 可以认为是 WebSphere MQ 的简化版，或称为体验版。它由 WebSphere MQ Client 和 WebSphere MQ Server 组成，其中 Client 部分可以在 Windows、AIX、HP-UX 和 SUN Solaris 平台上运行，每一个 UNIX 平台又可以有 SSL 附加功能。但 Server 部分仅限于 Windows 和 Linux for Intel 平台，在使用上也有一定的限制。一个队列管理器的通道连接最多只能有 10 个，Client 端通道连接最多也只能有 10 个，消息大小不能超过 4MB（当然，可以用消息分组的办法突破这一限制）等等。WebSphere MQ Express 也不能为远程资源管理器加入全局交易中。

另外，WebSphere MQ Express 产品提供了一些独有的程序。比如：Express 产品漫游，Express 文件传送，Express 信息中心等等。但是，在功能上并没有什么创新，仍然被 WebSphere MQ 所包含。

WebSphere MQ Express 适合于那些中小型企业应用，在并发度上要求不高，传递数据的量也不大，投资较少。WebSphere MQ Express 也适用于树状结构中下一层的结点，这些结点通常对连接数要求不高。

1.1.3.2 WebSphere MQ

WebSphere MQ 是核心产品，它包含了 WebSphere MQ Express 的全部，也突破了连接上的限制。完整的版本由 Client 和 Server 及相应的文档组成。Server 部分支持 Windows、AIX、HP-UX、Solaris、Linux for Intel、Linux for zSeries 和 iSeries 平台。Client 部分除了支持所有 Server 平台之外还支持很多其它平台，可以在 SupportPac 网站上免费下载。

Client 分为普通 Client 和 Extended Transactional Client。后者可以支持 Server 为远程资源管理器加入全局交易中。当然，这种环境需要有全局资源管理器，比如 WebSphere、CICS、Tuxedo 等等，具体参见“交易”一章。对于 UNIX 平台，普通 Client 又可以选用支持 SSL 协议，用于配置基于 SSL 的 Client/Server MQI 安全通道。

WebSphere MQ 是构建完整的异构应用互连平台所必需的，由于对连接数量没有限制，可以根据需要架构各种复杂的网状或树状结构，通常对于大中型企业应用。

1.1.3.3 WebSphere MQ Extended Security Edition

WebSphere MQ Extended Security Edition 由 WebSphere MQ 的全部加上 Tivoli Access Manager for Business Integration (TAMBI) 组成。它除了具有 WebSphere MQ 的全部功能，可以保证数据传送之外，还有 TAMBI 的功能，提供一个集成的 MQ 安全环境，可以对消息加密传送，可以生成数字签名等等。大型企业应用和敏感数据传送应用都可以选择这个产品。

1.2 概念与对象

WebSphere MQ 运行环境中较多的概念，其中有一部分是可以作为实体进行的操作的，称为 MQ 对象。每一个对象都有各自的属性，不同的属性决定了对象的特性和工作方式。消息、队列、队列管理器、通道是 MQ 中最重要的概念和对象。

1.2.1 消息 (Message)

消息是 WebSphere MQ 中最小的概念，本质上就是一段数据，它被一个或多个应用程序所理解，是应用程序之间传递的信息载体。消息可以大致分成两部分：应用数据体和消息数据头。(图 1-2)

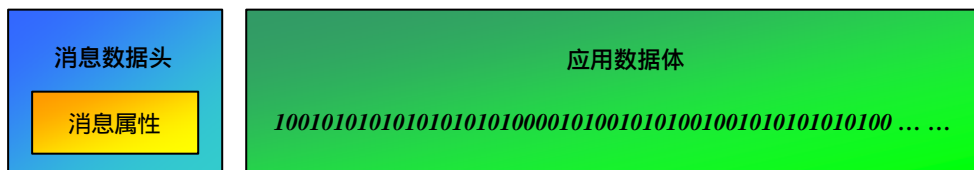


图 1-2 消息的结构

- 消息数据头是对消息属性的描述，这段信息往往被队列管理器用来确定对消息的处理。消息数据头可以由应用程序或系统的消息服务程序共同产生，它包含了消息在传送中的必要信息，如目标队列管理器的名字，目标队列的名字，以及消息的其它一些属性。
- 应用数据体是应用间传送的实质的数据消息，它可以是字串、数据结构甚至二进制数据。包含的内容可以是文本、文件、声音、图像等任何数据，这些数据只对特定的应用具有特定的含义。所以，应用数据体的结构和内容由应用程序定义，通信双方需要事先约定报文格式。

消息可以分成持久 (Persistent) 消息和非持久 (Non-Persistent) 消息。所谓“持久”的意思，就是在 WebSphere MQ 队列管理器重新启动后，消息是否仍然能保持。

在 WebSphere MQ 中，消息可以是有限长度的任何一段信息。所谓有限长度，在开放平台 (Windows NT/2000/XP, AIX, HP-UX, Solaris, Linux) 上缺省是 4MB，但这个上限可以调整到 100MB。也就是说，每段信息应该控制在这个有限长度中。当然，这并不限制 WebSphere MQ 用来传递更大的信息，比如文件。相关内容参见“分组与分段”章节。

1.2.2 队列 (Queue)

我们可以简单地把队列看成一个容器，用于存放消息。队列按其定义可成本地队列、远程队列定义、别名队列定义、模型队列定义。图 1-3。其中只有本地队列是真正意义上的队列实体，可以存放消息。远程队列定义和别名队列定义只是一个队列定义，指向另一个队列实体。远程队列定义指向的是其它队列管理器中的队列，而别名队列指向的是本地队列管理器中的队列。模型队列有一点特殊，它本身只是一个队列定义，描述了模型的属性，但是当打开模型队列的时候，队列管理器会以这个定义为模型，创建一个本地队列，被称为动态队列。

一个队列管理器下辖很多个消息队列，但每个队列却只能属于一个队列管理器。队列在它所属的管理器中只能有一个唯一的名字，不能与同一个管理器的其它队列重名。当消息被添加到队列中，它缺省将被加到最后，与之相反，删除消息缺省却是从头开始。

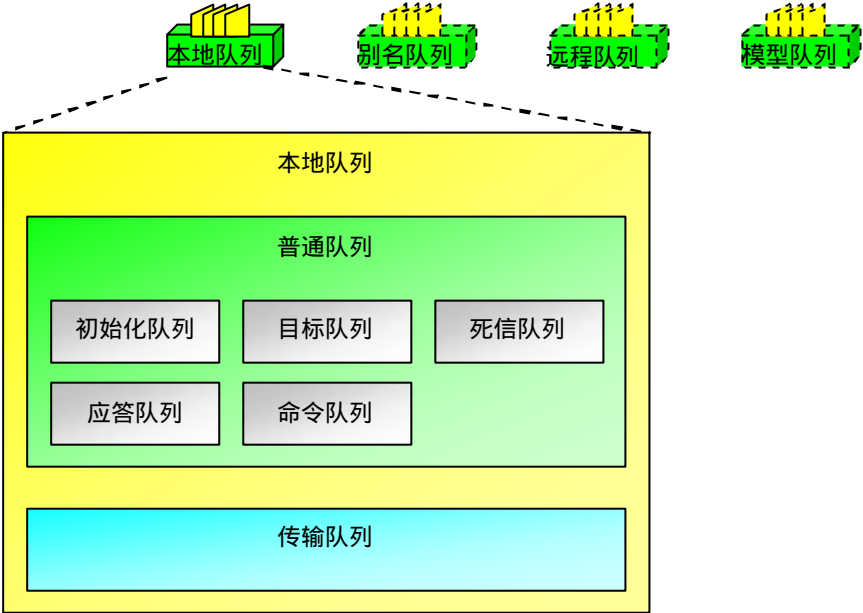


图 1-3 队列的分类

1.2.2.1 本地队列

本地队列按功能又可分成初始化队列，传输队列，目标队列和死信队列。初始化队列用做消息触发功能。传输队列只是暂存待传的消息，在条件许可的情况下，通过管道将消息传送其它的队列管理器。目标队列是消息的目的地，可以长期存放消息。如果消息不能送达目标队列，也不能再路由出去，则被自动放入死信队列保存。(图 1-4)

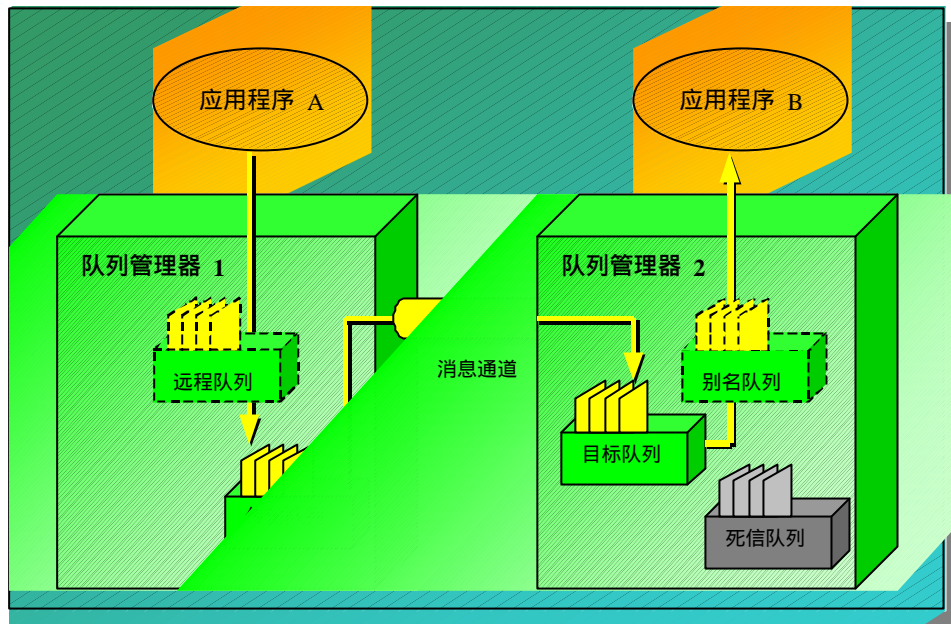


图 1-4 各种队列在消息传送时的作用

1.2.2.1.1 普通队列

能够真正长期存放消息的本地队列，我们称之为普通队列。一般说来，应用程序只对其做简单的 MQGET 和 MQPUT 以收发消息，这也是系统中用得最多的消息队列。通常在不致引起混淆的情况下，我们也将普通本地队列简称为本地队列。

1.2.2.1.2 传输队列

要送往远地的消息将放入传输队列。在适当的时候，消息会被从传输队列中取出并送往远地，最终放入远端的本地队列。所以，从本地系统的立场来看，传输队列是用来暂时存放输出消息的。

传输队列本身是一个本地队列，它与普通队列的差别是传输队列具有的属性 USAGE (XMITQ)。

1.2.2.1.3 初始化队列

初始化队列是配合消息触发用的，如果队列上配置有消息触发功能，则需要指定另一个相关队列以存放触发消息，这个队列就是初始化队列。初始化队列本质上就是一个普通本地队列。

1.2.2.1.4 目标队列

在消息通信的时候，消息最终的目的地称为目标队列。如果消息是通过传输队列转发，WebSphere MQ 会自动为消息体添加一个传输消息头，其数据结构为 MQXQH。其中的 RemoteQName 和 RemoteQMgrName 两个域指明了目标队列和目标队列管理器。如果消息被

放入死信队列，则 WebSphere MQ 会自动为消息体添加一个死信消息头，其数据结构为 MQDLH，其中 DestQName 和 DestQMGrName 两个域指明了原消息的目标队列和目标队列管理器。

1.2.2.1.5 死信队列

死信队列本质上是普通的本地队列，由于队列管理器的 DEADQ 属性指定的该队列为死信队列，所以队列管理器认为无法投递的消息都被自动送去该队列。由于无法投递的消息很像信件投递中的死信，故而得名。

队列管理器在将消息放入死信队列的时候，会自动为消息体添加一个死信消息头，其数据结构为 MQDLH。其中 Reason 域指明了消息无法投递的原因。

1.2.2.1.6 应答队列

由于消息在发送后需要有对方的回应。这种回应可以是系统自动产生的报告消息，也可以是对方应用生成的应答消息。就应用而言，这些回应消息的目标队列就是应答队列。应答队列通常设置在消息头 (MQMD) 的 ReplyToQ 域中，它也总是与消息头中的另一个域 ReplyToQMGr 一起使用。

1.2.2.1.7 命令队列

指的是 WebSphere MQ 队列管理器中预定义的 SYSTEM.ADMIN.COMMAND.QUEUE。任何 MQSC 命令都可以送往该队列，并被队列管理器的命令服务器 (Command Server) 接收处理。

1.2.2.2 别名队列

别名队列只是一种队列定义，它自身只是队列的逻辑名字，并不是队列实体本身。别名队列的 TARGQ 属性指明了其代表的目标队列名称，目标队列通常是本地队列。其实别名队列只是提供了队列名称之间的映射关系，可以将别名队列看作是指针，指向其目标队列。注意，这里的目标队列自身又可以是一个别名队列，指向下一个目标队列。然而，在对别名队列打开 (MQOPEN) 操作时，只允许别名队列指向的目标队列是一个本地普通队列，即一层映射关系。如果存在上述的多层映射关系，则报错 MQRC_ALIAS_BASE_Q_TYPE_ERROR。

通过别名定义，WebSphere MQ 也可以动态改变消息流向。比如：某程序在代码中指定消息写入队列 A，但是您希望在不变动代码的情况下将消息写入队列 B。这时只要定义别名 B，使之指向 A 即可。再如：某队列 C 是可读可写的本地队列，管理员希望它对于一类程序只可读，对另一类程序只可写，这时可以对 C 定义两个别名 D 和 E，D 只允许读，E 只允许写。也就是说，本地队列 C 的属性为 GET(ENABLED) 且 PUT(ENABLED)，将 D 的属性设置为 GET(ENABLED) 且 PUT(DISABLED)，E 的属性为 GET(DISABLED) 且 PUT(ENABLED)。将 D 和 E 分别提供给上述两类程序使用，这样可以完全避免应用程序的误操作。

1.2.2.3 远程队列

远程队列与别名队列类似，也只是一个队列定义，用来指定远端队列管理器中的队列。使用了远程队列定义，程序就不需要知道目标队列的位置（所在的队列管理器）了。

远程队列定义包括目标队列管理器名和目标队列名，而这种队列定义对于访问地的应用程序是透明的。这种技术不但使应用程序只需要对一个简单的队列名操作，而且可以在线地通过修改远程队列定义，而动态改变路由。

1.2.2.4 模型队列

模型队列定义了一套本地队列的属性集合，一旦打开模型队列，队列管理器会按这些属性动态地创建一个本地队列。模型队列的 DEFTYPE 属性可以取值 PERMDYN 和 TEMPDYN，分别代表永久动态队列和临时动态队列。

1.2.2.4.1 永久动态队列

永久动态队列由模型队列动态创建，并可以永久存在。在调用 MQOPEN 时创建，以后就和普通的本地队列一样工作。在调用 MQCLOSE 时，缺省情况下会保留消息和队列，当然也可以通过设置关闭选项 (Close Option) 的来清除消息甚至删除永久动态队列。

1.2.2.4.2 临时动态队列

临时动态队列也是由模型队列动态创建，但只在会话 (Session) 中临时存在。在调用 MQOPEN 时创建，在同一个线程中 MQCLOSE 时关闭并自动删除。MQCLOSE 时无所谓关闭选项 (Close Option) 的取值。

1.2.3 队列管理器 (Queue Manager)

队列管理器构建了独立的 WebSphere MQ 的运行环境，它是消息队列的管理者，用来维护和管理消息队列。一台机器上可以创建一个或多个队列管理器，每个队列管理器有各自的名字，通常情况下，它不能与网络中的其它队列管理器重名。如果我们把队列管理器比作是数据库，那么队列就是其中的一张表，消息就是表中的一条记录。

队列管理器是负责向应用程序提供消息服务的机构。在 WebSphere MQ 中，队列管理器集成了对象的定义、配置、管理、调度以及提供各种服务的功能于一身。WebSphere MQ 的系统管理工具提供了对系统部件配置与管理的功能，应用程序必须首先连接到队列管理器，然后在队列管理器的控制下对各种对象进行操作。

WebSphere MQ 中的队列管理器可以含有很多个队列，但一个队列只能属于一个队列管理器。一个操作系统平台可以创建一个队列管理器，也可以创建多个队列管理器。队列管理器、队列、通道等等都是 WebSphere MQ 的对象，所有的对象都有各自的属性，有些属性

必须在对象创建的时候指定，有些可以在创建以后更改。

1.2.4 通道 (Channel)

通道是两个队列管理器之间的一种单向的点对点的通信连接，消息在通道中只能单向流动。如果需要双向交流，可以建立一对通道，一来一去。站在队列管理器的角度，这一对通道可以按消息的流向分成输入通道和输出通道。通过配置，对于放入本地传输队列中的消息，队列管理器会自动将其通过输出通道发出，送入对方的远程目标队列。

两个队列管理器之间可以有多条通道负责传输不同的内容，这样设计往往是为了将不同优先级的消息错开，运行于不同速率的网络连接上。或者即便是所有通道都运行于相同的网络物理连接上，也可以将不同大小的消息传送分开，以免小数据传送被大文件所堵塞。如果多条通道共享一条网络物理连接，通道的速率之和受限于网络速度。这样可以增加传送的并发度但并不能增加整体的传送速度。

在通道上可以配置不同的通信协议，这样就使得编程接口与通信协议无关。通道两端的配置必须匹配，且名字相同，否则无法连通。队列管理器之间的通信是通过配置通道来实现的，通道两侧的队列管理器对这个通道的相关参数应该能对应起来，一个通道只能用一种通信协议，但不同的通道可以有不同的通信协议。可见，通道是架设在通信协议之上的对象，架设在不同通信协议上的通道在应用层看来都大同小异。

1.2.4.1 通道类型 (Channel Type)

WebSphere MQ 用通道类型属性 (CHLTYPE) 约定了通信双方在连接握手协议中的主动方和被动方以及应用消息的流向。可选以下这些类型：

- SDR Sender。握手协议的主动方，消息的发送方
- RCVR Receiver。握手协议的被动方，消息的接收方
- SVR Server。在握手协议中可以是主动方也可以是被动方，消息的发送方
- RQSTR Requester。在握手协议中可以是主动方也可以是被动方，消息的接收方
- CLNTCONN Client Connection。在 Client-Server 连接时，定义客户端连接定义表 (Client Channel Definition Table) 时使用。握手协议的主动方，消息的发送方
- SVRCONN Server Connection。在 Client-Server 连接时，定义服务器端连接时使用。握手协议的被动方，消息的接收方
- CLUSSDR Cluster Sender。在群集中发送配置信息和应用消息。握手协议的主动方，消息的发送方
- CLUSRCVR Cluster Receiver。在群集中接收配置信息和应用消息。握手协议的被动方，消息的接收方

通信双方的通道类型配对并不是可以随意排列组合的，共有六种。如图 1-5：

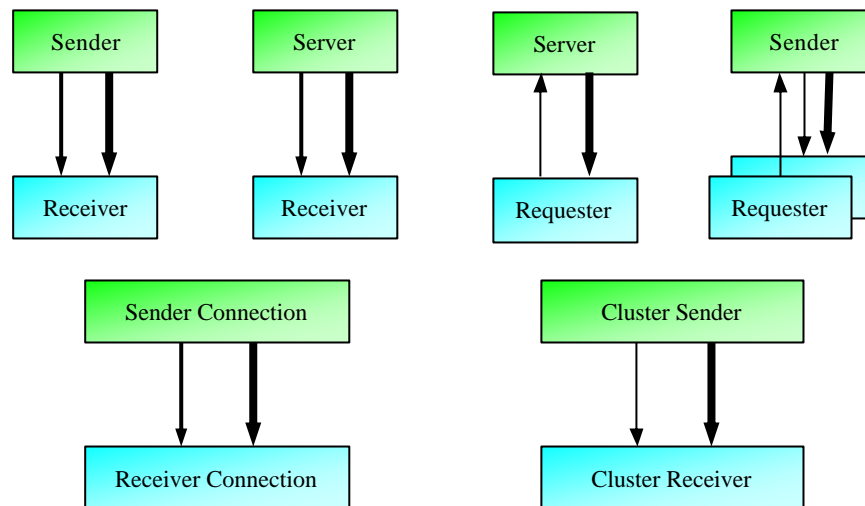


图 1-5 通道类型的配对

图中细线箭标表示握手协议中的主动连接，粗线箭标表示应用消息流向。消息在所有的通道上都是单向传送的。

- Sender/Receiver 是所有连接中最简单、最常用的一种。Sender 是通道主动方，也是消息发送方。
- Requester/Server 也是常用的一种连接方式。Requester 是通道主动方，但通道连接后，它作为消息接收方，Server 是消息发送方。
- Server/Receiver 与 Sender/Receiver 类似，Server 是消息的发送方，也是连接的主动方。与 Sender 定义类似，Server 定义中必须指定 CONNAME 参数。
- Sender/Requester 的连接过程稍微复杂一些，Requester 首先与 Sender 连接，在通知对方连接参数后连接断开。Sender 进行反向连接，消息也是反向传送的。这种反向连接的方式，称为 Callback Connection。
- Sender Connection/Receiver Connection 与 Sender/Receiver 方式相同。用于 Client/Server 之间的 MQI 通道。
- Cluster Sender/Cluster Receiver 与 Sender/Receiver 方式相同。用于群集中队列管理器之间的连接。

由于 Sender/Receiver、Server/Receiver 的连接主动方和消息发送方相同，所以可以在发送端设定通道触发 (Channel Trigger)。

由于 Sender/Receiver、Server/Receiver、Requester/Server 的连接被动方事先不需要知道主动方的连接参数，所以可以用于连接主动端是动态地址的应用场合。

由于 Sender/Requester 有反向建立连接的功能，所以常常用于双向安全认证。

1.2.4.2 消息通道协议 (MCP)

消息通道协议是 WebSphere MQ 用来传递消息时使用的通信协议。MCP (Message Channel Protocol) 可使用多种底层通信协议传递消息 (LU6.2, DECNet ...)。消息通道协议使得消息的传送独立于通信协议，应用程序通过统一的接口与 MQ 打交道，而不再需要关心通信层使用的是 TCP/IP 还是 SNA。目前 MCP 支持的通信协议有 LU6.2、DECNet 和 TCP/IP。

不同操作系统的 WebSphere MQ 支持的通信协议的数量和种类可能稍有不同，详情请

参见本操作系统的 《WebSphere MQ 用户手册》

1.2.4.3 消息通道代理 (MCA)

消息通道代理 (MCA, Message Channel Agent) 本质上是一个通信程序, 它用来在队列管理器之间传递消息。通道可以以进程的方式工作, 即独立的 MCA 进程。也可以以线程的方式嵌入系统 MCA 进程中工作。对于前者, 根据不同的 MCP, 发送端进程和接收端进程的 MCA 名通常是不同的。

1.2.5 名称列表 (Name List)

名称列表是 WebSphere MQ 的一种对象, 它实质上是多个其它 WebSphere MQ 对象的名称集合。其内容由多个字串组成, 中间用逗号隔开, 每个字串就是一个对象名称。

名称列表本身无法代表它所含的对象, 例如无法对名称列表进行 MQPUT 或 MQGET 操作, 类似的操作应该由分发列表 (Distribution List) 完成。定义名称列表只是定义了一个集合, 往往是为了方便应用访问多个对象。比如应用程序动态地从名称列表中读出操作对象并依次进行操作, 如果操作的对象有所增减, 只需要修改名称列表即可。名称列表使管理人员可以在不修改应用的前提下, 通过动态地增减名称列表中的内容来进行管理。

名称列表多用于群集 (Cluster) 环境中指定一个队列管理器同时属于多个群集的情况, 这时名称列表的内容就是多个群集的名称集合。名称列表可以用于以下一些对象属性:

- QMgr.REPOSNL
- QMgr.SSLCRLNL
- Queue.CLUSNL
- Channel.CLUSNL

WebSphere MQ 中每个对象都有各自的属性, 它们中的大多数是可以创建后修改的。这里, 我们采用“对象.属性”的记号方式表示对象的属性, 例 Queue.CLUSNL 表示队列的 CLUSNL 属性。以下同。

1.2.6 分发列表 (Distribution List)

分发列表可以使 WebSphere MQ 应用程序一次将一条消息同时发送到多个队列上。这里的一次发送指调用一次 MQPUT 或 MQPUT1, 多个目标队列可以是本地队列也可以远程队列, 如果多个远程队列的目标队列管理器相同, 则在网络上只需要传送一次即可, 节省了网络开销。当消息到达目标队列管理器后, 再自动分发到各个目标队列中, 当然这要求源队列管理器和目标队列管理器都支持分发列表功能。

分发列表的操作是可以在一个交易中完成的, 也就是说, 多个队列的发送是可以一起提交或回滚的。

1.2.7 进程定义 (Process)

进程定义对象用于 WebSphere MQ 的触发机制中,用来描述触发程序的对象。这个程序可以是一个操作系统程序,可以是一个 MQ 应用,也可以是一个 CICS 交易,在进程定义的属性中需要设定触发程序的路径、名称、参数等信息。

在消息触发环境中,一旦触发条件满足即可引起触发,队列管理器在生成触发消息的时候会参考进程定义,将定义中某些属性被抄入触发消息头 (MQTM 结构) 中,形成触发消息。该触发消息被触发监控器读走并处理,监控器可以根据 MQTM 触发消息头中的信息启动相应的进程。

1.2.8 认证信息 (Auth Info)

认证信息 (Authentication Information) 定义了 SSL 认证所需要的证书吊销列表 (CRL, Certificate Revocation List) 所在的 LDAP 服务器,同时定义了连入该 LDAP 服务器所需的用户名和口令。

1.2.9 客户端和服务端 (Client & Server)

WebSphere MQ 分成客户端和服务端,只有服务端有对象的概念,所以只有服务端的应用程序可以对本地对象进行直接操作。客户端通过 MQI 通道与服务端相连接,客户端应用程序发出的所有操作指令都通过该通道传送到服务器,在服务器端执行后结果返回客户端。通常情况下,客户端的应用程序代码与服务端相同,在程序编译时连接的库文件不同。

1.2.10 操作界面 (MQ Interface)

应用程序通过操作界面与 WebSphere MQ 打交道,这里的操作界面就是消息队列接口 (Message Queue Interface, MQI)。MQI 实际上是一套编程接口,负责处理应用程序向 WebSphere MQ 提交的各种操作请求,应用程序完全不需要关心 WebSphere MQ 的内部结构与具体实现,如消息队列,传输队列等等。

当应用程序通过 MQI 送出一条消息到远程队列,队列管理器会在它的消息数据头中加上路由信息,消息被转入传输队列,等待送出。MQI 的操作非常简单直观,如:MQOPEN、MQCLOSE、MQGET、MQPUT 等等。

由于 MQ 的互连通信是通过存储转发机制完成的,所以操作与传输是异步的。这意味着应用程序通过操作界面将消息发送出去时,消息首先存储在本地,当通信畅通时再被转发。应用程序可以继续处理自己的逻辑,而不必等待消息传达对方。

1.2.11 应用程序 (MQ Application)

应用程序可以是商业的或用户自行开发的含有对 WebSphere MQ 操作的程序。MQI 提供了支持的有平台的通用编程接口，如 VSE/ESA 上的 COBOL，Tandem Guardian 上的 TAL 和 C，其它平台上的 C。应用程序只要能够调用相应的库函数，它就可以操作 WebSphere MQ。

这一节介绍了 WebSphere MQ 中的基本概念和对象，其中最核心的部分是消息、队列、队列管理器和通道。对于程序设计人员，通常更关心消息和队列，对于维护管理人员，通常会更关心队列管理器和通道。下面让我们来看一看这些对象的工作原理。

1.3 工作原理

WebSphere MQ 的工作原理的核心就是存储转发。在单个队列管理器的环境中，队列可以用于存储应用间传递的消息，从而使应用程序在各自环节上进行处理，并通过队列形成环环相扣的处理流程。在多个队列管理器的环境中，消息可以跨平台进行流动，从而使整个处理流程在分布式计算环境中完成。

1.3.1 PUT 和 GET

WebSphere MQ 的应用程序可以通过 MQ 界面 (MQI ,MQ Interface) 进行操作。实际上，MQI 提供了有限的 API，其中最本质的两个动作是 PUT 和 GET。PUT 指应用程序放一条消息放入到队列中，GET 则相反，应用程序将一条消息从队列中取出。WebSphere MQ 通过队列机制来完成消息排队和传递的工作，从而使应用程序之间实现松耦合的联系。如下图，应用程序 A 产生消息，通过 PUT 调用放入队列中，应用程序 B 将消息取出并进行相应的处理，消息的报文格式及内容决定了应用程序 B 处理的具体工作。这样就实现了应用程序 A 到 B 之间的单向消息传递，如果需要双向传递消息则必须再类似地约定反向队列。图 1-6。

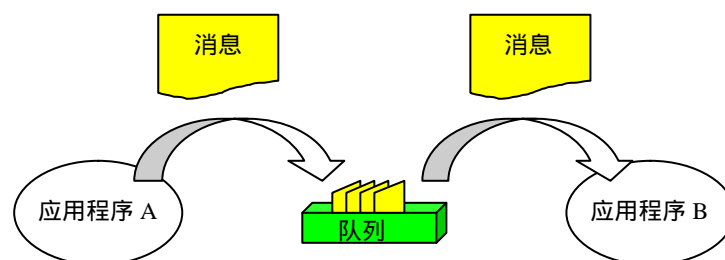


图 1-6 应用通过队列传递消息

应用程序设计的时候必须约定双方的报文格式，如果用通用格式（如 XML）则需考虑由此带来的灵活性和信息冗余，在两者之间平衡选择。在运行环境中，还需要考虑 PUT 和 GET 的频率与速度，以免消息有在队列中堆积起来。

WebSphere MQ 提供的远程队列机制可以将目标队列设定到另外一个队列管理器中，这样应用程序 A 和 B 就可以在两台机器上运行而不改动任何代码，应用程序 A 仍然做着相同的 PUT 操作将消息放入队列中，该消息会自动路由到另一个队列管理器中的队列中，应用程序 B 从该队列中 GET 消息，与原先一样地处理。也就是说，这种配置结构上的改变对应

用程序是完全透明的。WebSphere MQ 的这种特性使得其应用的扩展性极佳，任何应用在设计之初并不需要考虑太多的性能及扩展性问题，在需要时可以很方便地将应用中任何一部分拆到其它的机器上，实现多机计算。图 1-7。

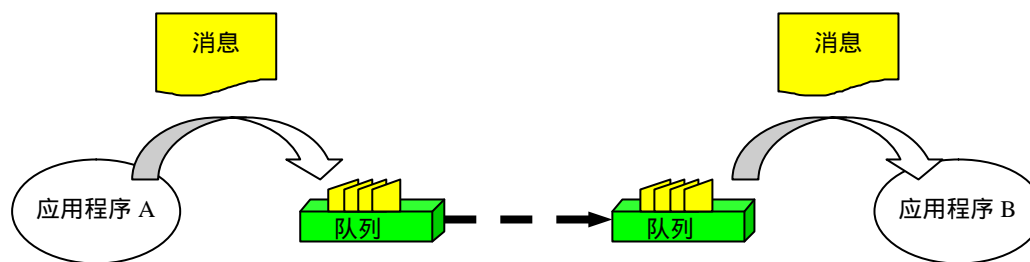


图 1-7 应用通过队列跨网络传递消息

1.3.2 协同工作

通常说来，一个应用系统会由多个应用模块组成，一个处理流程也会由多个处理步骤组成。它们之间可能是串行的关系，也可能是并行的关系。在 WebSphere MQ 应用设计中，我们可以自然地将多个模块或多个步骤设计成不同的应用程序，而它们之间的中间数据则通过消息的方式传递，用队列暂存。图 1-8。这样一来，应用系统会有以下好处：

1. 结构清晰，容易并行开发和调试。
2. 扩展性极好，能够很容易地部署到跨平台环境中。
3. 灵活性好，一旦流程改变了，可以较容易地进行修改。

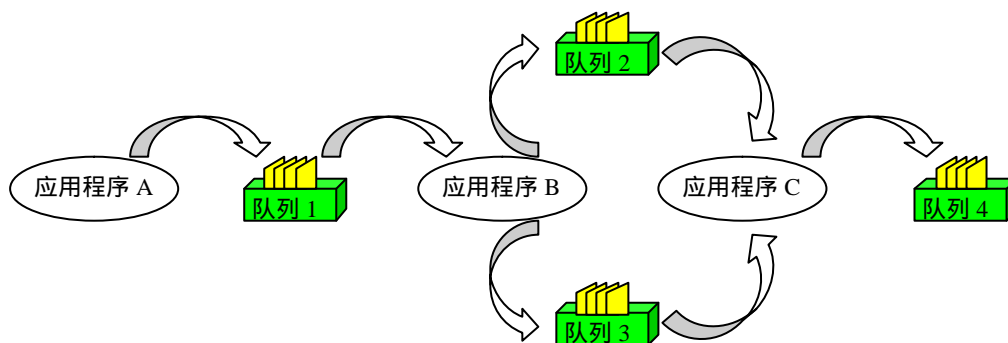


图 1-8 协同工作

这样，在应用系统设计的时候只需要考虑它的逻辑架构，而无需担心它的物理部署。各个处理环节可以通过 WebSphere MQ 贯穿起来，协同工作。

1.3.3 互连通信

1.3.3.1 消息通道 (Message Channel)

WebSphere MQ 跨平台的互连通信是依靠队列管理器之间的消息通道实现的，消息通道就是消息传递的管道，架设 in 队列管理器之间，消息从一头流入从另一头流出，消息的内容和次序完全不变。

配置通道时需要注意通道在两个队列管理器中同名，且类型要配对，见“概念与对象”

章节。WebSphere MQ 中通道中消息是单向传递的,但通道上的协议控制信息可以双向传递。如果需要双向传送消息,则需要创建双向的通道定义。

1.3.3.2 消息路由 (Message Routing)

我们首先拿现实生活中寄信做例子来类比 WebSphere MQ 中的一些基本概念,从而理解 WebSphere MQ 的工作原理。现实生活中家家户户都可能有一个通信地址,对应着一个存放到达信件的信箱。每一封寄出的信件总是先到本地邮局,通过邮局之间的信件交换,到达对方所在的邮局,最后到达对方的信箱里。中间的邮路越复杂,时间就越长。这里的邮局相当于 WebSphere MQ 中的队列管理器,信箱相当于队列,信件相当于消息。我们的每一封信件都有信封和信瓤两部分,信封上面往往有收信人通信地址和发信人通信地址,信瓤里是真正的内容(图 1-9)。WebSphere MQ 中的消息也一样,它分成消息头和消息体两部分。消息头是消息的属性集合,含有目标队列管理器名和目标队列名,WebSphere MQ 就是利用这段消息来找到目标队列的。消息体是消息的内容,可以是任意的一段内存信息。

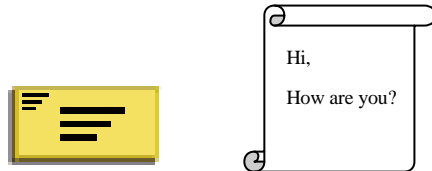


图 1-9 信封和信瓤

WebSphere MQ 依靠每条消息头上所含的路由信息,将消息准确地送达目的地。路由信息中的远程队列管理器名指的是远端系统的名字,远程队列名指的是远端系统中的目标队列名。一个完整的队列名应该包含两部分:队列管理器名和队列名,格式为 `queue_name@queue_manager_name`。两部分名字的长度上限都是 48 字节,这两部分名字构成了消息路由的最基本的信息,算法其实很简单:

如果队列管理器名未标明,则缺省加上本地队列管理器的名字。队列名会缺省地匹配要地普通队列,而队列管理器名会缺省地匹配本地队列管理器名或传输队列名。一般说来,建议传输队列名与远程队列管理器同名。消息传送过程如下:

1. 应用程序通过 MQI 发送消息
2. WebSphere MQ 查看 `queue_manager_name` 是否是本地队列管理器
 - a) 如果是的,将消息放入本地的名为 `queue_name` 的消息队列中
 - b) 如果不是,将消息放入名为 `queue_manager_name` 的传输队列中
3. MCP 会把传输队列中的消息送达远端
4. 远端的消息队列将消息放入远端系统中名为 `queue_name` 的消息队列中
5. 远端的应用程序从远端的本地队列中取得消息

这种办法提供了简单的路由功能,但有一个明显的缺点,直接使用全名会要求应用程序了解软件的网络结构分布,哪些队列在哪里。这有悖于 WebSphere MQ 对应用程序隐藏网络细节的设计初衷。所以,在跨队列管理器的应用中,通常使用别名队列和远程队列来指定对方队列的名字,从而将队列的分布信息保留在配置中。

1.3.3.3 消息传送

我们在实现消息的跨队列管理器之间的传送时,通常会在本地队列管理器上配置远程队列和传输队列,在远端的队列管理器上配置本地队列,并通过通道将两者连接起来。图 1-10。

这里的远程队列只是一个定义并无队列实体,也就是说远程队列不能存放消息。应用程序一旦将消息通过 MQPUT 送出,则立刻放入传输队列中。传输队列暂时存放待由通道发送的消息,一旦通道连通且条件允许,系统通信程序 MCA 会立即将消息送出。消息到达对方目标队列管理器后,由对方的通信程序 MCA 接收下来并放入相应的目标队列。这里的目标队列就是目标队列管理器上的本地队列。整个过程的效果就好像应用程序直接将消息送入目标队列一样。

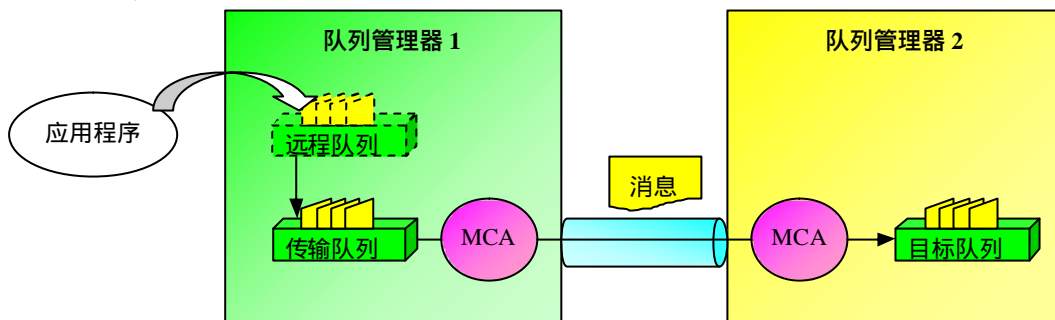


图 1-10 消息的传递过程

远程队列的定义实际上就是指定的目标队列的位置及传输路径。其中,目标队列的位置通过设定目标队列名和目标队列管理器名来确定,消息在路由过程中寻找该目标地址。传输路径就是传输队列名,消息会通过该传输队列送出。不同的远程队列可以共用一个传输队列。

传输队列本质上是一个本地队列,只是由系统通信进程 MCA 监护。应用程序可以人为地通过 MQPUT 放一条消息到传输队列上,但如果该消息没有传输头 (MQXQH),则不会被发送,消息按以下方式处理:

1. 如果队列管理器设置了缺省死信队列,则消息放入该死信队列,死信消息原因码为 MQFB_XMIT_Q_MSG_ERROR。
2. 如果队列管理器未设置缺省死信队列,则对于非持久性消息,消息被仍掉。对于持久性消息,消息会留在传输队列中无处可去,这时有可能会堵住后继的消息,造成通道无法发送。

消息在经过远程队列放入传输队列时会由队列管理器自动添加一个传输头 (MQXQH),且传输头中的内容会根据远程队列定义自动填写。如果说,原先的消息是 MQMD + Body,则放入传输队列的消息为 MQMD + MQXQH + Body。

第 2 章 安装

我们在前面介绍了 WebSphere MQ 的相关概念及其工作原理,从这一章开始介绍动手操作和管理,让我们从产品安装开始。

WebSphere MQ 的安装过程中有不少选择分支，从而可以安装出不同的效果。事实上，安装中的选择大部分可以在以后的管理和配置中再进行调整。由于初学者对这一过程不熟悉比较容易混淆，从而给以后的操作带来不便。这里以 Windows 平台上的 WebSphere MQ 5.3 为例，详细描述产品安装的全过程。同时也简单介绍了其它 UNIX 平台的安装过程。

2.1 安装环境

在安装前要检查机器的硬件配置及操作系统是否达到相关要求，否则可能引起安装失败。WebSphere MQ 对于机器的环境要求并不高，普通的 PC 机都可以安装。如果有网络通信，则需要有相应的网卡并进行通信配置，比如 IP 地址。对于单机环境，可以用自环方式配置通信，比如配置虚拟 Loopback 网卡。具体环境需要如下：

2.1.1 硬件

- PC 机，Intel 32 位兼容芯片。
- 屏幕至少支持 800 x 600 分辨率。
- 如果需要通信，应该有网卡等硬件设备，支持 TCP/IP、SNA LU6.2、NETBIOS、SPX 等通信协议。
- 至少 85MB 硬盘空间用于安装，20MB 空间用于工作，30MB 系统临时空间用于暂存数据。
- 建议有 128MB 以上内存。

2.1.2 操作系统

- Windows NT
 - Service Pack 6a
 - Microsoft Internet Explorer 4.0.1+
 - Microsoft HTML Help 1.2 (产品 CD 中含)
 - Microsoft Management Console (MMC) 1.1 (产品 CD 中含)
 - Microsoft Installer (MSI) 2.0+ (产品 CD 中含)
 - Microsoft Active Directory Client Extensions (ADCE) for Windows NT (如果有 ADCE 支持)
 - Java Runtime Environment Version (JRE) 1.3+ (如果有 JAVA 编程支持)
 - Option Pack 4 for Microsoft Windows NT (如果有 Microsoft Transaction Server -- MTS 支持)
- Windows 2000 Professional, Server, Advanced Server
 - Service Pack 2+
 - Microsoft Installer (MSI) 2.0+
 - Java Runtime Environment Version (JRE) 1.3+ (如果有 JAVA 编程支持)
- Windows XP Professional
 - Java Runtime Environment Version (JRE) 1.3+ (如果有 JAVA 编程支持)

2.1.3 通信协议

对于 TCP/IP、NETBIOS、SPX 操作系统都预置支持。

对于 SNA 协议，则至少需要安装以下软件中的一个来支持

- IBM Communications Server for Windows NT, Version 5.0 and Version 6.1.1.
- Attachmate Extra! Personal Client, Version 6.7.
- Attachmate Extra! Enterprise 2000.
- Microsoft SNA Server, Version 4.0.
- Microsoft Host Integrated Server 2000.

2.2 安装介质

2.2.1 正版

联系 IBM 公司，得到 IBM WebSphere MQ 介质，共 2CD。一张 CD 是相关平台的产品，一张 CD 是文档。产品 CD 中含安装所需的全部软件，包括 HTML Help 1.2，MMC 1.1，MSI 2.0，JDK，ADSI 等等。文档 CD 中含各种语言的文档，且有 PDF，HTML，HTMLHelp 三种格式。需要说明的是，简体中文的文档并不完整，如果要深入学习 WebSphere MQ，建议使用英文文档。

从 <http://www-3.ibm.com/software/integration/mqfamily/support/summary/> 下载最新的补丁。

2.2.2 试用版

试用版产品是可以免费下载的，从 <http://www-3.ibm.com/software/integration/mqfamily/> 页面上下载到有效期为 90 天的 WebSphere MQ 5.3 试用版，目前只有 Windows 平台和 Linux 平台上的试用版产品，且皆为英文版。

从 <http://www-3.ibm.com/software/integration/mqfamily/library/manualsa/> 页面下载相关的 pdf 格式的文档。

从 <http://www-3.ibm.com/software/integration/mqfamily/support/summary/> 下载最新的补丁。

2.3 安装过程

在安装介质中的目录中双击 Setup.exe。在一段动画之后（可以用 ESC 键跳过动画），出现安装启动板。按照左边一档 1、2、3 的步骤顺序来检查系统是否满足安装的先决条件。

- 步骤 1，检查软件安装环境的先决条件。查看右边所需的软件是否全部打勾，打勾表示已经成功安装，打叉表示尚未安装。需要将右边所有项全都打勾，才开始步骤 2。
- 步骤 2，检查网络先决条件。如果不需要特别安装域模式，可以选择“否”。如果需要安装后自动进行缺省配置，则安装前机器最好安装有网卡，且至少拥有一个 IP 地址。
- 步骤 3，检查安装前状态（图 2-1），确保能通过状态检查。开始安装。



图 2-1 WebSphere MQ 安装启动板

按“启动 WebSphere MQ Installer”→ 选择“我接受该许可证协议中的条款”→ 可以选择“定制”安装，并选择程序文件夹安装目录、数据文件夹安装目录、选择日志文件夹安装目录 → 选择全部部件（图 2-2），开始安装。当问及许可证时，回答“是，已经购买了足够的许可证单元”。→ 最后确认完成。

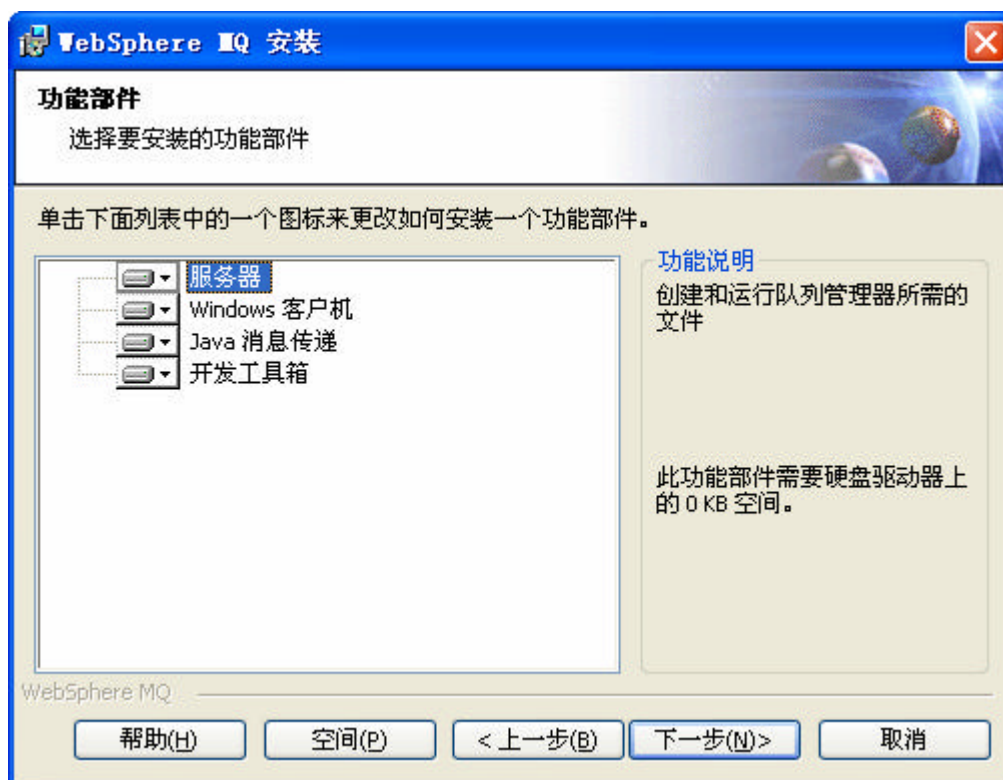


图 2-2 选择安装组件

这里要说明两点：


1. 程序文件夹指的是安装目录，即 WebSphere MQ 产品本身所在的位置。这一部分是不会随着配置的变化而改变的，相对稳定。数据文件夹指的是工作目录，即队列管理器、队列等 WebSphere MQ 对象所在的位置，通常一个对象会对应一个文件，所以，配置的变化会带来目录内容的改变。日志文件夹指的是日志目录，WebSphere MQ 的日常动作，消息出入队列等会被自动记录在日志中。日志对于保证消息安全起着至关重要的作用，在生产环境中，为了高可靠性，经常将日志安装在另一个硬盘上。具体内容，参见日志管理一章。
2. WebSphere MQ 5.3 中的许可证单元延续了过去版本中的容量单位 (Capacity Unit) 的概念。基本上容量单位是机器 CPU 计算能力的一种体现，现在的许可证单元就是 CPU 数量。比如一台双 CPU 的 PC Server，在采购 WebSphere MQ 的时候，就应该购买 2 个许可证单元。在安装完 WebSphere MQ 后可以用 `setmqcap` 命令设置所购买的许可证单元数量。另外，`dspmcap` 命令可以用来查看 CPU 数量与许可证单元数量是否相符。

```
C:\>dspmcap
```

```
购买的处理器定量为 1
```



```
此机器中的处理器数为 1
```

接着安装程序自动进入“准备 WebSphere MQ 向导”，这时可以先取消，等安装完后再配置。这时的 WebSphere MQ 实际上已经可以用了，如果您是第一次安装 WebSphere MQ 5.3，则安装结束。如果以前曾经安装过低版本的 MQ，或者留有上次创建的队列管理器对象，且这些对象需要升级或迁移，则需要继续下一步：缺省配置。

一旦安装成功，会在 Windows 2000 或 XP 的屏幕右下角出现图标 。这是 WebSphere MQ Task Bar for Windows。同时，在“启动”文件夹中出现“WebSphere MQ 任务栏”。事

实上，“WebSphere MQ 任务栏”指向的就是 WebSphere MQ Task Bar 可执行文件，也可以用命令行方式启动：

```
C:\>amqmtbrn.exe -startup。
```

如果将启动组中的“WebSphere MQ 任务栏”删除，则操作系统重启后就不会自动运行 Task Bar 程序，右下角也就不会出现那个图标。这时，如果在命令行中打入 amqmtbrn，则右下角会出现图标：。同时用任务管理器可以观察到 amqmtbrn 进程。按此右键点击“启动 WebSphere MQ”，右下角图标由红转绿。同时用任务管理器可以观察到 amqsvc 和 amqmsrvn 进程的出现。按此图标（绿色），右键点击“停止 WebSphere MQ”，经过一段时间，在此期间，右下角图标变成：，最后右下角图标由绿转红，同时用任务管理器可以观察到 amqsvc 和 amqmsrvn 进程消失。按此图标（红色），右键点击“隐藏”，发现图标不见了，同时用任务管理器可以观察到 amqmtbrn 进程也消失了。所以，对 Task Bar 程序的操作实际上就是对整个 WebSphere MQ 运行环境的操作，效果上就是启动或停止相关的进程。

至此，产品已经安装完毕。但是，WebSphere MQ 还需要有一个配置的过程，才能真正地使用。缺省的安装过程会紧接着启动缺省配置，而缺省配置又分多个步骤，可以经过相应的配置向导向 MQ 配置成不同的运行模式。我们也可以在这时直接结束安装，将配置工作留待以后实施。

2.4 缺省配置

如果您过去安装过低版本的 MQ 软件，在卸载后留有上次创建的队列管理器对象或者配置文件，在新安装 WebSphere MQ 5.3 以后，您希望将其升级或迁移，从而溶入新的软件环境中，而不必重新手工创建且配置一遍。另一种可能是机器迁移，队列管理器数据文件夹是从其它机器上拷贝过来的，通过这一步可以将其放入现在的环境中。

2.4.1 准备 WebSphere MQ 向导

在安装的“IBM WebSphere MQ”程序组中选择“准备 WebSphere MQ 向导”即可出现缺省向导（图 2-3）。或者在命令行窗口中打入 amqmipse 命令，也有同样的效果。



图 2-3 准备 WebSphere MQ 向导

如果您的 Windows 安装运行于单机模式，而非 Windows 域模式，则在回答“网络中是否有域控制器时”选择“否”。→ 如果您愿意将现有的队列管理器配置为允许远程管理，则可点击进入“远程管理向导”，并转去 2.4.2。→ 如果希望简单地设置缺省配置，可点击进入“缺省配置向导”，并转去 2.4.3。→ 最后确认完成。当然“远程管理向导”和“缺省配置向导”都可以跳过，可以事后再修改设置。

2.4.2 远程管理向导

在“准备 WebSphere MQ 向导”中点击“允许对现有队列管理器进行远程管理”即可进入远程管理向导。(图 2-4)



图 2-4 远程管理向导

在向导中选中允许远程管理的队列管理器 → 输入侦听器的端口号，缺省为 1414，设置 → 最后确认完成。

远程管理向导实际上是为队列管理器创建了一个侦听器和命令服务器配置，并且在队列管理器启动的时候将这两个部件一起启动，接受来自远端的网络连接和管理命令。将队列管理器配置成远程管理是为了实现集中式远程控制，便于使用人员的管理工作。

2.4.3 缺省配置向导

在“准备 WebSphere MQ 向导”中点击“设置缺省配置”或在命令行窗口打入命令：
amqmgse，则可进入缺省配置向导。(图 2-5)



图 2-5 缺省配置向导

配置向导会试图创建缺省队列管理器，并将且配置在群集环境中。在向导中选择配置选项，可以保留缺省设置，即允许远程管理且加入缺省群集。→ 选择队列管理器在群集中的地位。可以选择“是”，将它作为该群集的资源库。→ 最后确认完成。等待配置完成后，关闭配置向导。

缺省配置向导会试图在机器上创建一个缺省队列管理器，名为 `QM_HostName`，其中 `HostName` 为机器名。然后创建缺省群集，名为 `DEFAULT_CLUSTER`。将缺省队列管理器配置加入该群集且设为该群集的资源库。

这里有三点需要说明：

1. 缺省配置会创建缺省队列管理器，名为 `QM_HostName`。例如机器名叫 `cyxt21`，则创建的缺省队列管理器为 `QM_cyxt21`。所谓缺省队列管理器，即在应用时如果不提供队列管理器名称的情况下，系统会自动用默认的队列管理器。可以在 Windows 注册表中找出当前的缺省队列管理器名：`<MQ_HKEY>\Configuration\DefaultQueueManager`。
2. 缺省的群集为 `DEFAULT_CLUSTER`，且缺省配置会将队列管理器配入这个群集环境中。在初次安装时，可能会不明白 WebSphere MQ 中群集的概念，这没关系。对于普通的应用来说，队列管理器是否在群集中并不影响。要深入了解群集，可以参见“群集”一章。
3. 对于初次安装，您也许根本没有必要进入准备 WebSphere MQ 向导、远程管理向导、缺省配置向导，将环境搞复杂了。在安装结束后直接取消退出即可。这些复杂的配置安全可以留待以后进行。

2.5 安装补丁

复杂的系统软件都难免会有纰漏 (Bug), 所以各种商业软件都会有补丁, WebSphere MQ 也不例外, 相关补丁可以从 IBM 网站下载。WebSphere MQ 的补丁称为 CSD, 后面紧跟的数字是补丁号, 比如 CSD06 表示 6 号补丁, 后一号的补丁在内容上完全覆盖前一号的补丁。

在安装补丁之前, 首先要停止所有的队列管理器及 WebSphere MQ 相关进程, 在 Windows 任务管理器中看不见所有 amq 打头的进程。然后, 将从网站下载的 WebSphere MQ 5.3 补丁展开, 选择展开目录, 这时安装程序开始自动安装补丁, 当然也可以在展开目录中运行 amqicsdn.exe, 手工开始安装。安装过程十分简单: 选择备份文件夹, 开始安装 → 完成。

在安装完补丁后, 在 WebSphere MQ 安装目录下会出现补丁目录, 例如 CSD06, 它实际上是备份文件夹, 内容是补丁安装前的相应文件。用 mqver 命令, 可以显示目前的版本号 and 补丁号。

```
C:\>mqver
Name:      WebSphere MQ
Version:    530.6 CSD06
CMVC level: p530-06-L040211
BuildType: IKAP - (Production)
```

安装补丁与配置没有必然的先后关系, 这里因为 Windows 平台的“准备 WebSphere MQ 向导”在产品安装之后会自动启动, 为了显得连贯, 在内容上将缺省配置安排在前。事实上, 通常建议补丁安装紧接着产品安装, 然后再配置, 在 UNIX 环境下尤其如此。

2.6 其它平台

WebSphere MQ 不仅在 Windows 平台上有众多的应用, 在 UNIX 平台中的应用也十分普遍。由于各种 UNIX 平台都大同小异, 软件的安装过程也比较类似。所以在这一节中, 读者应该注意安装环境的要求在各个平台上的差异及安装过程中的一些差别。

WebSphere MQ 在 UNIX 或 Linux 上的安装过程中会自动创建 mqm 用户和 mqm 组, 且 mqm 用户属于 mqm 组。当然, 这一点也可以在安装之前用手工完成。以后所有的 WebSphere MQ 管理命令, 如 crtmqm, dltmqm, strmqm, endmqm, dspmq 等等, 都只有 mqm 组中的用户才能使用。

2.6.1 AIX

AIX 是 IBM RS/6000 机器采用的操作系统, 目前正在使用的主要有 4.3.3 和 5L 两个版本。该操作系统上的软件大多可以通过系统工具 smit 来进行安装和管理。

2.6.1.1 操作系统

WebSphere MQ 可以安装在 AIX 4.3.3 或 5L 操作系统上。具体要求见表 2-1。

表 2-1 对 AIX 操作系统的要求

操作系统版本	补丁	说明
AIX 4.3.3	U472177, Y2K fixes	32 位
AIX 5.1	U477366, U477367, U477368	32 位或 64 位
AIX 5.2		32 位或 64 位

2.6.1.2 安装过程

1. 插入产品 CD，作为 root 用户登录。
2. 用 smit 命令安装
Software Installation and Maintenance
Install and Update Software
Install and Update from LATEST Available Software
3. 选择安装介质 /dev/cd0 (CD-ROM Drive)，在 SOFTWARE to install 档中选择需要安装的部件。可以安装的部件有：
 - Runtime
 - Base Kit
 - Server
 - Client for AIX
 - Sample programs
 - DCE support
 - DCE samples
 - Java messaging
 - Message catalogs
 - Man pages
 - IBM Global Security Kit V6
 - IBM Key Management tool (iKeyman)
4. Include corresponding LANGUAGE filesets? 档中选择 YES。
5. 如果是 AIX 4.3.3，选择 OK 开始安装。如果是 AIX 5.1 或 5.2，对 Preview new LICENSE agreements? 可以选择 YES。对 ACCEPT new license agreements? 必须选择 YES。
6. 安装完毕后，用 setmqcap 命令指定许可证单元数量，如下。注意，如果 WebSphere MQ 在运行期间发现 CPU 数量与许可证单元数量不符，会不断地报错。

```
setmqcap 4
```

2.6.2 HP-UX

HP-UX 是 HP 公司的小型机采用的操作系统，目前使用了主要有 11 和 11i 两个版本。WebSphere MQ 可以通过介质中的安装脚本进行安装。注意，在安装之前需要调整一些系统

参数。

2.6.2.1 操作系统

WebSphere MQ for HP-UX 支持 HP-UX 11 和 11i 两个版本的操作系统。(表 2-2)

表 2-2 对 HP-UX 操作系统的要求		
操作系统版本	补丁	说明
HP-UX 11.0	PHSS_24627, PHSS_22543	32 位
HP-UX 11i (11.11)		32 位

2.6.2.2 安装过程

1. 缺省情况下，HP-UX 操作系统中通信参数对 WebSphere MQ 来说都偏小，需要调整。以 root 身份登录，更改系统参数。表 2-3 为最小推荐值。考虑到不同队列管理器的独立性，shmmni, semmni, semmns, semmnu 的值会与队列管理器数量有关。如果使用环型日志，msgmap 和 msgmax 参数是不必要的。

表 2-3 对 HP-UX 操作系统参数要求	
系统参数	最小推荐值
shmmax	536870912
shmseg	1024
shmmni	1024
shmem	1
sema	1
semaem	16384
semvmx	32767
semmns	16384
semmni	1024 (semmni < semmns)
semmap	1026 (semmni +2)
semmnu	2048
semume	256
msgmni	50
msgtql	256
msgmap	258 (msgtql +2)
msgmax	4096
msgmnb	4096
msgssz	8
msgseg	1024
maxusers	32
max_thread_proc	66
maxfiles	1024
nfile	10000

2. 以 root 身份登录，插入产品 CD。

3. Mount CDROM

```
cd /usr/sbin  
pfs_mountd &  
pfsd 4 &  
pfs_mount --o xlat=unix /<path to CD-ROM device>/<localdir>
```

4. 接受安装许可

```
cd <mount point> 例：cd /cdrom  
./mqlicense.sh
```

-text_only 表示文本方式显示，例如：mqlicense.sh -text_only。

-accept 表示缺省接受

5. 用 swinstall 工具安装

```
swinstall -s /cdrom<localdir>/hpux11/<drivername>.v11
```

选择 MQSeries ,在 Action 菜单中选择 Open item ,选择所有需要安装的部件，
可选的部件有：

- Runtime
- Base
- Server
- Client
- Sample programs
- DCE support
- DCE samples
- Java messaging
- Message catalogs
- Man pages
- IBM Global Security Kit V6
- IBM Key Management tool (iKeyman)

6. 在 Action 菜单中选择 Mark for install。到上一层菜单中，在 Action 菜单中选择 Install (analysis)，OK 开始安装。

7. 安装完毕后，用 setmqcap 命令指定许可证单元数量。注意，如果 WebSphere MQ 在运行期间发现 CPU 数量与许可证单元数量不符，会不断地报错。Purchased license units not set (use setmqcap)。setmqcap 命令用法例：

```
setmqcap 4
```

2.6.3 Solaris

Solaris 是 SUN 公司的小型机采用的操作系统，WebSphere MQ 可以通过介质中的安装脚本进行安装。在安装之间要注意操作系统的补丁及系统参数。

2.6.3.1 操作系统

WebSphere MQ for Solaris 支持 7 和 8 两个版本的操作系统。(表 2-4)

表 2-4 对 Solaris 操作系统的要求

操作系统版本	补丁	说明
Sun Solaris 7	107171-02	32 位
	107544-03	
	106950-16	
	106327-11	
	106300-10	
	106541-18	
	106980-17	
Sun Solaris 8	● 基础补丁 同上	32 位
	● 附加补丁	
	108827-12	
	111177-06	
	● SSL 补丁	
	108434-02	
	111327-02	
	108991 108528	

2.6.3.2 安装过程

1. 以 root 身份登录，查看系统参数。

```
sysdef -i
```

2. 更改系统参数，修改 /etc/system 文件。下表为推荐值：

```
set shmsys:shminfo_shmmax = 4294967295
set shmsys:shminfo_shmseg = 1024
set shmmin:shminfo_shmmin = 1
set shmsys:shminfo_shmmni = 1024
set semsys:seminfo_semmni = 1024
set semsys:seminfo_semaem = 16384
set semsys:seminfo_sevmnx = 32767
set semsys:seminfo_semmap = 1026
set semsys:seminfo_semmns = 16384
set semsys:seminfo_semmnl = 100
set semsys:seminfo_semopm = 100
set semsys:seminfo_semmnu = 2048
set semsys:seminfo_semume = 256
set msgsys:msginfo_msgmni = 50
set msgsys:msginfo_msgmap = 1026
set msgsys:msginfo_msgmax = 4096
set msgsys:msginfo_msgmnb = 4096
```

不要修改 shmmin 值。考虑到不同队列管理器的独立性，shmmni, semmni, semmns, semmnu 的值会与队列管理器数量有关。

如果不使用环型日志，msgmap 和 msgmax 参数是不必要的。

3. 以 root 身份登录，插入产品 CD。
4. Mount CDRom，不妨假设 mount point 为 /cdrom
5. 接受安装许可

```
/cdrom/mqlicense.sh
```

-text_only 文本方式显示，例如：mqlicense.sh -text_only。

6. 用 pkgadd 工具安装

```
pkgadd -d /cdrom
```

选择所有需要安装的部件，可选择的部件有：

- Server
- Client
- Sample programs
- DCE support
- DCE samples
- Java messaging
- Message catalogs
- Man pages
- IBM Global Security Kit V6
- IBM Key Management tool (iKeyman)

7. 安装完毕后，用 setmqcap 命令指定许可证单元数量。注意，如果 WebSphere MQ 在运行期间发现 CPU 数量与许可证单元数量不符，会不断地报错。Purchased license units not set (use setmqcap)。setmqcap 命令用法例：

```
setmqcap 4
```

2.6.4 Linux

Linux 是目前流行的操作系统，由于它开放源代码和全球性组织开发的特点，Linux 成为发展最快的操作系统。WebSphere MQ 在 Linux 上的安装相对简单。

2.6.4.1 操作系统

WebSphere MQ for Linux 可以安装于两类 Linux：Linux for Intel 和 Linux for zSeries。

- Linux for Intel
 - Red Hat Linux V7.2
 - Caldera OpenLinux V3.1
 - SuSE Linux Enterprise Server V7
 - Turbolinux V7.0
- Linux for zSeries
 - Red Hat Linux for S/390(R)
 - SuSE Linux Enterprise Server V7

2.6.4.2 安装过程

1. 以 root 身份登录，查看系统参数。按照需要调整相当的参数值，参数文件为：

```
/proc/sys/kernel/shmmax
```

```
/proc/sys/kernel/shmmni
```

```
/proc/sys/kernel/shmall
```

```
/proc/sys/kernel/sem
```

相应地，调整最大打开文件数和最大进程数。

2. 以 root 身份登录，插入产品 CD。
3. Mount CDROM，不妨假设 mount point 为 /cdrom
4. 接受安装许可

```
/cdrom/mqlicense.sh
```

-text_only 文本方式显示，例如：mqlicense.sh -text_only。

5. 用 rpm 工具安装

- a) 如果是 Linux for Intel，顺序执行

```
rpm -i MQSeriesRuntime-5.3.0-1.i386.rpm
```

```
rpm -i MQSeriesSDK-5.3.0-1.i386.rpm
```

```
rpm -i MQSeriesServer-5.3.0-1.i386.rpm
```

- b) 如果是 Linux for zSeries，顺序执行

```
rpm -i MQSeriesRuntime-5.3.0-1.s390.rpm
```

```
rpm -i MQSeriesSDK-5.3.0-1.s390.rpm
```

```
rpm -i MQSeriesServer-5.3.0-1.s390.rpm
```

6. 安装完毕后，用 setmqcap 命令指定许可证单元数量。注意，如果 WebSphere MQ 在运行期间发现 CPU 数量与许可证单元数量不符，会不断地报错。Purchased license units not set (use setmqcap)。setmqcap 命令用法例：

```
setmqcap 4
```

2.7 安装目录

安装目录总共有四个：产品目录，数据目录，日志目录和出错目录。产品目录是 WebSphere MQ 产品本身，一旦安装完毕后，不会因为运行而改变。数据目录是与运行环境相关，通常一个对象对应一个文件或目录，对象属性或内容的改变会反映到文件或目录的改变。日志目录中存放的是 WebSphere MQ 运行日志文件，这些文件的组织会根据日志类型（环型日志或线型日志）而不同，其间记录了 MQ 在运行中的各种操作，在 MQ 出现异常时可用于恢复。出错目录用于存放错误日志信息，通常随着时间的推移，错误文件会越积越多，需要定期清理。

对于 Windows 平台，可以在安装时将这些目录指定到不同的硬盘上。对于 UNIX 平台，安装后生成的目录如下，也可以事先将这些目录指定到不同的文件系统，而这些文件系统可以创建在不同的硬盘上。将数据和日志分开可以提高性能，如果是线型日志，还可以做介质恢复，保证数据安全。

- 安装目录为 /opt/mqm
- 数据目录为 /var/mqm

- 日志目录为 /var/mqm/log
- 出错目录为 /var/mqm/errors

WebSphere MQ 的管理命令都以可执行文件的方式存放在 bin 目录下，编程的头文件放在 include 或 inc 目录，库文件放在 lib 目录下。对于每个队列管理器，会以队列管理器名在 qmgrs 和 log 目录下各生成一个子目录，用以存放所有该队列管理器中的对象和日志，其中 qmgrs 下的子目录是队列管理器的工作目录，其下还会生成一个 error 子目录，用于记录错误日志，WebSphere MQ 最基本的问题诊断方法就是查看这些错误日志。

下面以 Windows 和 AIX 平台为例，查看缺省情况下的安装目录。注意，两者在目录安排和目录取名上有所不同。

2.7.1 Windows

C:\Program Files\IBM\WebSphere MQ	产品目录
+-log	日志目录
+-bin	
+-Errors	
+-Qmgrs	数据目录
+-Config	
+-Java	
+-Tools	
+-c	
+-Samples	
+-include	
+-Lib	
+-Java	

2.7.2 AIX

/usr/lpp/mqm -> /usr/mqm	
/usr/mqm	产品目录
+-samp	
+-inc	
+-java	
+-lib	
+-bin	
+-ssl	
+-tivoli	
/var/mqm	数据目录
+-log	日志目录
+-qmgrs	

2.8 安装文档

WebSphere MQ 各种平台的详细安装文档可以参见 IBM WebSphere MQ 文档。

- Windows Quick Beginnings
- AIX Quick Beginnings
- HP-UX Quick Beginnings
- Solaris Quick Beginnings
- Linux Quick Beginnings
- iSeries Quick Beginnings
- z/OS Concepts and Planning Guide

第 3 章 控制与管理

在成功安装了 WebSphere MQ 产品之后,我们可以进行使用和配置,而这些都是通过控制与管理命令完成的。在这一章中我们会详细介绍各种控制命令的使用方法及管理方式。

WebSphere MQ 中的控制针对的是 MQ 部件,通常使用命令方式完成。管理针对的是 WebSphere MQ 对象,可以用 MQSC 脚本命令或图形界面工具完成。在对 WebSphere MQ 的维护中两者需要结合使用。对于 WebSphere MQ for UNIX,控制与管理都只能通过命令行界面完成。对于 WebSphere MQ for Windows,除了命令行方式之外,系统还提供了“WebSphere MQ 资源管理器”和“WebSphere MQ 服务”两个图形界面管理工具,大多数控制与管理工

作都可以通过点击右键选择对应功能来完成。

对于初次使用的用户可以按以下的步骤来体验一下 WebSphere MQ 的操作。

1. 创建队列管理器,我们不妨假定队列管理器名为 QM

```
C:\> crtmqm QM
```

2. 启动队列管理器

```
C:\> strmqm QM
```

3. 创建队列,假定队列名为 Q

首先,用命令行交互界面管理工具 RUNMQSC 连接队列管理器

```
C:\> runmqsc QM
```

```
5724-B41 (C) Copyright IBM Corp. 1994, 2002. ALL RIGHTS RESERVED.
```

```
启动队列管理器 QM 的 MQSC。
```

接着,在交互管理工具中使用脚本命令 define qlocal 创建出本地队列对象,最后用 end 命令退出交互管理工具。

```
define qlocal (Q)
```

```
end
```

4. 将消息放入队列

用 amqsput 命令将消息逐行放入队列,每行输入代表一条消息,输入空行结束。

```
C:\> amqsput Q QM
```

5. 从队列中取出消息

用 amqsget 命令可以将放入的消息全部取出。


```
C:\> amqsget Q QM
```

通过以上的操作，我们可以大致了解 WebSphere MQ 的管理、配置和使用的过程。其中第 1、2 两步是通过控制命令完成的，第 3 步是对队列管理器的配置，通过配置工具完成，第 4、5 两步实际上是运行了 WebSphere MQ 的应用程序。

如果目前对控制命令尚不熟悉也没关系，下面会做详细介绍。在阅读时也可以参照附录“ WebSphere MQ 命令一览表”，其中对所有的控制命令语法有详细的描述。

3.1 MQ 控制命令

3.1.1 MQ 队列管理器控制

队列管理器是构建 WebSphere MQ 运行环境的基础，用户需要首先创建并启动队列管理器才能进行以后的操作。队列管理器的控制命令可以创建、删除、启动、停止队列管理器，也可以显示系统中所有的队列管理器及其当前运行状态。

3.1.1.1 创建队列管理器

- 格式

```
crtmqm [选项] QMgrName
```

- 功能

创建队列管理器

- 说明

QMGrName 指的是待建的队列管理器名。crtmqm ? 显示命令语法，可以列出所有选项，更详细的语法说明可以参见附录“ WebSphere MQ 命令一览表”。常用的选项有 [-q]、[-d DefXmitQ]、[-u DeadQ]等，它们可以组合使用。选项如果取-q，表示创建缺省队列管理器，一台机器最多只能有一个缺省队列管理器。如果取-d，表示指明队列管理器的缺省传输队列。如果取-u，表示指明队列管理器的死信队列。

创建时的选项指定了队列管理器的属性，这些属性有些可以在创建后修改，有些则不可以。未用选项指明的属性就使用缺省值，创建命令可以不带选项（如 crtmqm QM），则所有属性皆使用缺省值。

- 举例

```
crtmqm -q QM
```

3.1.1.2 删除队列管理器

- 格式

```
dltmqm [-z] QMgrName
```

- 功能

删除队列管理器

- 说明

该命令执行的前提是队列管理器的相关进程已经全部停止了。该命令只有一个选项 `-z`，表示抑制命令执行时发出的信息。

- 举例

```
dltmqm -z QM
```

3.1.1.3 启动队列管理器

- 格式

```
strmqm [-z | -c] QMgrName
```

- 功能

`strmqm` 用来启动队列管理器，也可以用来用缺省对象重建队列管理器。

- 说明

如果在命令中没有选项，则简单地启动队列管理器。如果用 `-z`，表示抑制命令执行时发出的信息。`-c` 选项比较特殊，表示重置队列管理器。命令的执行过程为：先启动队列管理器，再覆盖重建所有的缺省系统对象，最后自动停止该队列管理器。

如果上一次队列管理器未能正常停止，则启动后可能会回滚一些未完成的交易，恢复一些消息和对象。如果因为某些异常不能重启队列管理器，可以在出错日志（errors 目录）中寻找原因。

- 举例

```
strmqm QM
```

3.1.1.4 停止队列管理器

- 格式

```
endmqm [-z] [-c | -w | -i | -p] QMgrName
```

- 功能

`endmqm` 用来停止队列管理器。

- 说明

通过不同的选项，可以设置不同的停止方式。比如，选项取 `-c`，表示受控（Controlled）方式停止，即等到连接在该队列管理器上的所有应用全部主动断开连接后，才停止队列管理器。不过，该命令是立即返回的，显示命令已经提交。如果选项取 `-w`，则同样是受控方式停止，只是命令不是立即返回的，在指定的时限（秒）内等待（Wait）命令执行完毕后或超时返回。若选项取 `-i`，即立即（Immediate）停止，即使其它连接在该队列管理器上的应用的所有后继 MQ API 全部失败，以督促它们退出。命令在队列管理器停止后返回。若选项取 `-p`，即强行（Preemptive）停止。不会等待其它应用程序释放资源或断开连接，而直接将队列管理器进程退出，有可能会造成异常。`-z` 选项表示抑制命令执行时发出的信息，可以与其它选项组合使用。

- 举例

```
endmqm QM
```

3.1.1.5 显示队列管理器

- 格式

dspmq [-m QMgrName]

- 功能

dspmq 用来显示本地的队列管理器的运行状态。

- 说明

如果用 -m 选项，表示显示某个具体的队列管理器运行状态，否则，表示显示所有的队列管理器状态。

- 举例

```
dspmq
```

QMNAME (QM)	STATUS (正在运行)
QMNAME (QM1)	STATUS (正常结束)
QMNAME (QM2)	STATUS (正常结束)

3.1.2 MQ 命令服务器控制

WebSphere MQ 命令服务器是队列管理器的一个组件，用来对外来的命令消息进行解释和执行。在远程管理和编程管理的应用中，需要启动命令服务器。一个队列管理器最多只有一个命令服务器，缺省情况下在创建队列管理器时由系统一并创建。

3.1.2.1 启动命令服务器

- 格式

strmqcsv [QMgrName]

- 功能

strmqcsv 用来启动命令服务器。

- 说明

QMgrName 指的是命令服务器所在的队列管理器，缺省为系统缺省队列管理器。命令服务器是队列管理器的附加部件，用来执行管理命令。

- 举例

```
strmqm QM
```

3.1.2.2 停止命令服务器

- 格式

endmqcsv [-c | -i] QMgrName

- 功能

endmqcsv 用来停止命令服务器

- 说明

通过不同的选项，可以选择不同的停止方式。-c 表示受控方式 (Controlled) 停止，允许已经开始的命令执行完成，然后停止该命令服务器。-i 表示立即 (Immediately) 停止，中止正在执行的命令并立即停止命令服务器。

- 举例

```
endmqm -c QM
```

3.1.2.3 显示命令服务器

- 格式
dspmqlcsv [QMgrName]
- 功能
dspmqlcsv 用来显示命令服务器的状态
- 说明
这里 QMgrName 指的是命令服务器所在的队列管理器，缺省为系统缺省队列管理器。
- 举例

```
dspmqlcsv QM
```

3.1.3 MQ 监听器控制

WebSphere MQ 中监听器也是队列管理器的一个组件，用来监听外来的连接请求并相应地做出反应。监听器通常需要先配置，然后才能运行，配置参数与监听器选择的通信协议有关。当然，也可以在第一次启动监听器时将配置参数传入，隐式地进行配置。一个队列管理器可以有多个监听器，分别应用于不同的通信协议或同一协议的不同参数。比如 TCP/IP 的不同端口。

3.1.3.1 启动监听器

- 格式
runmqclsr [-m QMgrName] -t (TCP | LU62 | NETBIOS | SPX) [参数]
- 功能
runmqclsr 用来启动监听器
- 说明
监听器是通道连接的被动方用来监听网络连接的程序。命令格式中 QMgrName 指的是命令服务器所在的队列管理器，缺省为系统缺省队列管理器。-t 选项指定通信协议，参数与监听器选择的通信协议有关，具体参见“附录 WebSphere MQ 命令一览表”。
- 举例

```
runmqclsr runmqclsr -t tcp -p 1414 -m QM
```

3.1.3.2 停止监听器

- 格式
endmqclsr [-w] [-m QMgrName]
- 功能
endmqclsr 用来停止监听器，实质上就是停止网络监听程序。
- 说明
QMgrName 指的是命令服务器所在的队列管理器，缺省为系统缺省队列管理器。选项为 -w 时表示等待方式 (Wait) 停止，命令执行会等到监听器停止完成后才返回，否则立即返回。

- 举例

```
endmqlsr -m QM
```

3.1.3.3 配置 TCP/IP 监听器

TCP/IP 协议是目前使用最广泛的通信协议。WebSphere MQ 队列管理器的 TCP/IP 监听器可以配置成三种形式，即用带参数的 runmqlsr 进程进行监听，UNIX 环境下可以用操作系统 TCP/IP Daemon 进程 inetd 进行监听，Windows 环境下也可以配置成 WebSphere MQ 服务的形式。一个队列管理器可以有多个监听器，分别监听多个端口。

- 配置 runmqlsr

这是配置 MQ 监听器的标准方式，格式为：runmqlsr -t tcp [-m <QMgr>] [-p <Port>]。如果缺省 -m 选项，表示使用缺省队列管理器，如果缺省 -p 选项，表示使用缺省端口 1414。

在 Windows 下可以用 `start runmqlsr -t tcp [-m <QMgr>] [-p <Port>]` 将命令放在另一个窗口中执行。

在 UNIX 下则可以用 `runmqlsr -t tcp [-m <QMgr>] [-p <Port>] &` 将命令放在后台执行。

- 配置 inetd

在 UNIX 平台中，可以通过配置 inetd 使 MQ 监听器植入系统监听服务中，相当于每次开机监听器就自动启动了。具体配置分三步：

1. 在 /etc/services 中添加：

```
MQSeries1 1414/tcp
MQSeries2 1415/tcp
```

其中，MQSeries1 和 MQSeries2 分别是为监听端口起的别名，可以是任意字符串。

2. 在 inetd.conf 中添加：

```
MQSeries1 stream tcp nowait mqm /usr/lpp/mqm/bin/amqcrsta amqcrsta [-m QMgr1]
MQSeries2 stream tcp nowait mqm /usr/lpp/mqm/bin/amqcrsta amqcrsta [-m QMgr2]
```

其中，MQSeries1 和 MQSeries2 需要与 /etc/services 文件中的定义配合起来。

3. 让修改的配置生效：

```
refresh -s inetd // AIX
kill -1 <inetd daemon 进程的 pid> // 其它 UNIX
```

- 配置 WebSphere MQ 服务

在 WebSphere MQ for Windows 产品中，可以通过“WebSphere MQ 服务”工具来进行配置。在配置界面中右键点击启动了的队列管理器，新建→侦听器，设置通信协议和相关参数后，会在注册表中配置相应的条目，如下。服务可以配置成“自动启动”方式或“手动启动”方式，自动启动方式即在队列管理器启动的同时自动启动侦听器程序。在执行启动失败后，也可以指定恢复的方式和参数。

<MQ_HKEY>\Configuration\Services\<QMName>\侦听器

在“WebSphere MQ 服务”配置工具中也可以配置其它的服务，如队列管理器、命令服务器、通道启动程序、触发器监控器、通道启动等等。其中的启动和恢复的方式和参数都与

此相似，以下不再赘述。

3.1.4 MQ 触发监控器控制

触发监控器 (Trigger Monitor) 是 WebSphere MQ 的组件之一，用于监控消息触发初始化队列并启动消息处理程序。触发监控器根据其运行的位置分为 MQ Client 端触发监控器和 MQ Server 端触发监控器，分别用于启动各自系统中的消息处理进程。

一个队列管理器可以有多个触发监控器，分别监控不同的初始化队列。程序员也可以根据规范自己编写触发监控器。

3.1.4.1 启动 Client 端触发监控器

- 格式

```
runmqtmc [-m QMgrName] [-q InitiationQName]
```

- 功能

runmqtmc 用来启动 Client 端触发监控器

- 说明

QMGrName 指的是命令服务器所在的队列管理器，缺省为系统缺省队列管理器。

InitiationQName 指的是触发监控器所监控的初始化队列，缺省为 SYSTEM.DEFAULT.INITIATION.QUEUE。

runmqtmc 运行于 Client 端，一旦触发条件满足即会有触发消息自动放入初始化队列中并被触发监控器读走。触发监控器根据触发消息的内容启动相关触发进程，该进程也运行于 Client 端。

- 举例

```
runmqtmc -m QM -q SYSTEM.DEFAULT.INITIATION.QUEUE
```

3.1.4.2 启动 Server 端触发监控器

- 格式

```
runmqtrm [-m QMgrName] [-q InitiationQName]
```

- 功能

runmqtrm 用来启动 Server 端触发监控器。

- 说明

与 runmqtmc 相同，QMGrName 指的是命令服务器所在的队列管理器，缺省为系统缺省队列管理器。InitiationQName 指的是触发监控器所监控的初始化队列，缺省为 SYSTEM.DEFAULT.INITIATION.QUEUE。

runmqtrm 运行于 Server 端，一旦触发条件满足即会有触发消息自动放入初始化队列中并被触发监控器读走。触发监控器根据触发消息的内容启动相关触发进程，该进程也运行于 Server 端。

Client 端与 Server 端的触发监控器原理是相同的，只是运行的位置不同，触发后在各自的运行平台上启动应用程序。

- 举例

```
runmqtrm -m QM -q SYSTEM.DEFAULT.INITIATION.QUEUE
```

3.1.4.3 停止触发监控器

WebSphere MQ 提供的 Client 端或 Server 端触发监控程序都无法优雅地停止 (Gracefully Stop)。所以要停止这些缺省的触发监控程序，可以将触发监控器进程找出来并用操作系统命令将其停止，比如 kill。对于程序员自行开发的触发监控器，要注意留有停止接口。

3.1.4.4 配置触发监控器服务

在 WebSphere MQ for Windows 产品中，可以通过“WebSphere MQ 服务”工具来进行配置。在配置界面中右键点击启动了的管理队列管理器，新建→触发器监控器→指定队列名，会在注册表中配置相应的条目：

```
<MQ_HKEY>\Configuration\Services\<QMName>\触发器监控器
```

3.1.5 小结

这一节列举了队列管理器及其部件的控制命令，表 3-1。我们通常可以使用这些命令对 MQ 的各个部件实现启动、停止、显示等功能，所有的控制命令都可以在操作界面下以命令行方式实现。

表 3-1 WebSphere MQ 控制命令

命令	格式	功能说明
队列管理器控制命令		
crtmqm	crtmqm [选项] QMgrName	创建队列管理器
dltmqm	dltmqm [选项] QMgrName	删除队列管理器
strmqm	strmqm [选项] QMgrName	启动队列管理器
endmqm	endmqm [选项] QMgrName	停止队列管理器
dspmq	dspmq [-m QMgrName]	显示队列管理器
命令服务器控制命令		
strmqcsv	strmqcsv [QMgrName]	启动命令服务器
endmqcsv	endmqcsv [选项] QMgrName	停止命令服务器
dspmqcsv	dspmqcsv [QMgrName]	显示命令服务器
监听器控制命令		
runmqlsr	runmqlsr [选项] [-m QMgrName]	启动监听器
endmqlsr	endmqlsr [选项] [-m QMgrName]	停止监听器
触发监控器控制命令		
runmqmtmc	runmqmtmc [-m QMgrName] [-q InitQName]	启动客户端触发监控器
runmqtrm	runmqtrm [-m QMgrName] [-q InitQName]	启动服务端触发监控器

3.2MQ 对象管理

WebSphere MQ，为了要兼顾各种平台的兼容性，提供了一个通用的字符交互界面管理工具 RUNMQSC，它的用法在所有平台上都一样，功能略有差别。RUNMQSC 运行的命令集称为 MQSC (MQ Script Command)，它是一种交互式的脚本命令，在 RUNMQSC 环境下

解释执行。从标准输入中读取一条 MQSC 命令（可能占多行），解释执行并在标准输出打印结果，然后再读取下一条 MQSC 命令，如此循环。通过将标准输入输出重定向到文件，可以将命令存放在脚本文件中，将输出存放在结果文件中，脚本文件习惯上以 `*.tst` 为后缀。

用户可以在命令行方式下运行 RUNMQSC，进入交互界面。RUNMQSC 的运行格式为：`runmqsc [QMgrName]`。其中，`QMgrName` 为连接的队列管理器名，缺省为系统缺省的队列管理器。一旦连接上队列管理器，就可以通过 MQSC 脚本命令创建、删除、显示队列管理器上的 MQ 对象了，而大多数对象及其属性都是一旦设置，立即生效的。所以，RUNMQSC 非常适合于动态配置。

在介绍对象管理的 MQSC 脚本命令之前，让我们来了解一下 RUNMQSC 的使用方法：

1. 在命令行中输入 `runmqsc <QMgrName>`，其中 `QMgrName` 是队列管理器名，不妨假定为 QM。

```
C:\> runmqsc QM
```

2. 在交互界面中输入 `?`，即可显示 MQSC 的所有一级命令

```
?
```

3. 在交互界面中输入一级命令，后面紧跟 `?`，则可显示相关的二级命令

```
DISPLAY ?
```

4. 在交互界面中输入二级命令，后面紧跟 `?`，则可显示相关的语法

```
DISPLAY QLOCAL ?
```

在一级命令中的 DEFINE 用于创建 WebSphere MQ 对象，DELETE 用于删除对象。每一个对象在创建后都会有各自的属性，这些属性大多数既可以在创建时指定，也可以在创建后修改。如果在创建时不特别指定，则属性的设置会参照系统缺省对象的设置。这些缺省对象是在创建队列管理器时自动创建的，通常以 SYSTEM 开头命名。一级命令中的 DISPLAY 用于显示对象的属性，ALTER 用于修改属性。注意，有些对象的属性在创建后是无法修改的，比如对象的创建日期和时间或者通道的类型等等。

所有的 MQSC 命令开始的语法格式都是由动作 (Action) + 对象 (Object) + 参数 (Parameter) 组成。动作就是一级命令，对象就是二级命令，每个对象都有唯一的名字，参数通常包括多个属性名与值。比如：

```
DISPLAY QLOCAL (Q)
```

```
ALTER QMGR CCSID (1381)
```

```
DEFINE CHANNEL (C) CHLTYPE (SDR) CONNAME ('127.0.0.1 (1414)') XMITQ (XQ)
```

```
DELETE PROCESS (P)
```

在 RUNMQSC 中大小写无关，所有的命令会先转换成全大写再提交执行。所以如果要表示大小相关的字符串，比如对象名，则用引号将字符串包住。

```
RUNMQSC QM
```

```
DEFINE QLOCAL ('LocalQ')
```

```
END
```

```
amqspout LocalQ QM
```

通常情况下，我们会把要执行的命令先放入脚本文件中，用重定向的方法将其输入，输出也可以定向到另一个结果文件中。执行的时候，RUNMQSC 会首先回显命令，然后显示

执行结果。所以，结果文件中既有命令的执行结果，也有命令本身。如果用-e 选项抑制回显命令，这样命令输出中就只有执行结果了。

```
runmqsc -e QM < test.tst > test.out
```

MQSC 的每条命令可以有各种选项从而完成不同的功能。为了保持通用性，建议每条命令不超过 72 个字符。命令如果太长可以分行，将命令行的最后个非空字符置为 ‘ + ’ 或 ‘ - ’，表示需要折行。其中，‘ + ’ 表示后续内容从下一行的第一个非空字符算起，‘ - ’ 表示后续内容从下一行的第一个字符算起。如：

```
DEFINE CHANNEL          (C_QM1.QM2) +
      CHLTYPE           (SDR) +
      TRPTYPE           (TCP) +
      CONNAME           ('127.0.0.1 (1415)') +
      XMITQ             (QM2) +
      REPLACE
```

3.2.1 队列管理器管理

MQSC 中用 ALTER QMGR 命令来修改队列管理器属性，用 DISPLAY QMGR 命令来显示队列管理器属性。

例：

```
// 将队列管理器设置为中文字符集
```

```
ALTER QMGR CCSID (1381)
```

```
// 将队列管理器的死信队列设为 Q1，缺省传输队列强行设置为 Q2
```

```
ALTER QMGR DEADQ (Q1) DEFXMITQ (Q2) FORCE
```

```
// 显示队列管理器的全部属性
```

```
DISPLAY QMGR
```

```
// 显示队列管理的字符集
```

```
DISPLAY QMGR CCSID
```

3.2.2 队列管理

MQSC 中用 DEFINE 来创建队列，DELETE 来删除队列。这里的队列可以是本地队列、远程队列、别名队列或模型队列。ALTER 命令用来修改队列属性，DISPLAY 用来显示队列属性。

例：

```
// 创建本地队列 Q
```

```
DEFINE QLOCAL (Q)
```

```
// 将本地队列 Q 的最大深度属性设置为 5
```

```
ALTER QLOCAL (Q) MAXDEPTH (5)
```

```
// 重新创建本地队列 Q。如果 Q 已经存在，则将其属性全部重置为缺省属性。
// 注意，这时队列中原有的消息仍然保留。
DEFINE QLOCAL (Q) REPLACE

// 删除本地队列 Q
DELETE QLOCAL (Q)
```

3.2.3 通道管理

MQSC 中用 DEFINE 来创建通道,DELETE 来删除通道。用 DISPLAY 来显示通道属性,用 ALTER 来修改属性。注意,通道只有在停止状态下才可以被删除或修改。

例：

```
// 创建接收端通道 c
DEFINE CHANNEL (C) CHLTYPE (RCVR)

// 创建发送方通道 c,连接对方的 IP 地址为 127.0.0.1,端口为 1414,通道连接传输队列 xQ
DEFINE CHANNEL (C) CHLTYPE (SDR) CONNAME ('127.0.0.1 (1414)') XMITQ (XQ)

// 删除通道 c
DELETE CHANNEL (C)

// 修改通道 c 的批次消息数量为 50
ALTER CHANNEL (C) CHLTYPE (SDR) BATCHSZ (50)

// 显示通道 c 的心跳间隔
DISPLAY CHANNEL (C) HBINT
```

由于通道是一种特殊的 MQ 对象,它的某些状态会随着通信环境而变化,比如通道状态、通道流量、通道消息序号等等,我们称之为通道的动态信息。MQSC 也提供了一些命令用来动态管理通道。

3.2.3.1 启动通道

启动通道有两种方式:MQSC 命令和控制命令,启动通道只有在连接通道的主动方发起才有作用。

1. 用 MQSC 命令 START CHANNEL 可以启动通道。格式为：

```
START CHANNEL (ChannelName)
```

2. 也可以用 WebSphere MQ 控制命令 runmqchl 来启动通道。格式为：

```
runmqchl [-m QMgrName] -c ChannelName
```

3.2.3.2 停止通道

用 MQSC 命令 STOP CHANNEL 可以停止通道,停止通道也只有在连接通道的主动方

发起才有作用。格式为：

```
STOP CHANNEL (ChannelName)
```

3.2.3.3 重置消息序号

通道为上面传送的每一条消息分配了一个序列号,它会自动累计增值,每传送一条消息,双方的消息序号都会自动加一。这个消息序号在通道中用 SEQNUM 属性表示,在双方连接通道的时候会约定一个起始值,以后被传递一条消息各自加一。通道的相关属性 SEQWRAP 表示序号的最大值,缺省最大值为 999,999,999。序列号越界后自动归零,从头开始。通道利用消息序号来标识传送和确认的消息。

通常情况下,通道双方的消息序号计数应该是相同的。然而在某些异常情况下,会出现双序号不一致的情况,这通常是因为通信故障后,双方对前面的某一条(或一批)消息是否发送成功理解不一致。在解决了不确定(In-doubt)的消息后,可以用 MQSC 命令通过重置消息序号将双方调整到一致。一旦连接断开后,通道重连时双方 MCA 会将消息序号同步。如果通信异常造成序号不一致,可以在通道发送端用 MQSC 命令 *RESET CHANNEL SEQNUM* 手工将两者同步。

在连接通道的主动方重置消息序号会将双方一起调整,在被动方重置则只设置一端。

```
RESET CHANNEL (ChannelName) [SEQNUM (number)]
```

3.2.3.4 解决不确定消息

发送方和接收方的通道状态中除了 SEQNUM(通道消息序号)参数控制消息传递外,还有 LUWID 参数。LUWID 指的消息批次交易号,对于每一批消息发送方都需要收到接收方的确认信息才认为消息完整无误地送达对方,接着产生下一个 LUWID 并开始下一批消息传送。如果没有收到确认而与接收方失去联系,这时发送方认为这批消息为不确定(In-doubt)状态。

大多数时候,WebSphere MQ 会在通道重连时自动解决不确定状态的问题。当然,我们也可以手工解决。事实上,通道的 LUWID 分成 CURLUWID 和 LSTLUWID 两个参数属性,具体工作过程如下:

1. 发送方产生一个批次交易号,设置在 CURLUWID 并通知接收方
2. 接收方将其设置在 CURLUWID
3. 发送方向接收方一条接一条地传送整批消息
4. 接收方在完整地收到消息后,将交易号设置在 LSTLUWID,提交整批消息并回送确认信息
5. 发送方在接收确认信息后,将交易号设置在 LSTLUWID,提交整批消息
6. 重复转到 1

所以,在发送方出现不确定状态时,只需要比较一个发送方的 CURLUWID 和接收方的 LSTLUWID,就可以知道该批消息在接收端是否已经提交,从而在发送方做出相应的动作即可。具体步骤如下:

- 1) 比较双方的 LUWID

- 对于不确定 (In-doubt) 状态的发送端：

```
DISPLAY CHSTATUS(name) SAVED CURLUWID
```

- 对于接收端：

```
DISPLAY CHSTATUS(name) SAVED LSTLUWID
```

- 2) 如果两者相同，说明该批消息在接收端已经完整地收到并提交。在发送端执行：

```
RESOLVE CHANNEL(name) ACTION(COMMIT)
```

- 3) 如果两者不同，说明该批消息在接收端未能完整地收到并提交。在发送端执行：

```
RESOLVE CHANNEL(name) ACTION(BACKOUT)
```

3.2.3.5 测试通道

类似于 TCP/IP 中的 PING 命令，MQSC 命令中也有对通道的 PING。格式如下。其中，DATALEN 表示 PING 数据包的大小，可以用 16 字节到 32,768 字节。

```
PING CHANNEL(channel_name) [DATALEN(16 | integer)]
```

PING 命令可以检查对方的队列管理器或端口监听器是否启动，也可以检查对方的通道定义是否正确。但不检查通道的通性状态。换句话说，PING CHANNEL 只检查通道能否连连通，而不检查目前是否连通。

3.2.4 进程定义管理

与队列管理命令类似，MQSC 中用 DEFINE PROCESS 来创建进程定义，用 DELETE PROCESS 删除进程定义。用 ALTER PROCESS 来修改进程定义属性，用 DISPLAY PROCESS 来显示进程定义属性。表 3-7，具体格式参见附录。

例：

```
// 创建进程定义 P，对应程序是记事本 (Notepad)
```

```
DEFINE PROCESS (P) APPLICID ('C:/WINNT/system32/notepad.exe') REPLACE
```

```
// 修改进程定义 P 的属性，设置用户数据 abc
```

```
ALTER PROCESS (P) USERDATA (abc) REPLACE
```

```
// 删除进程定义 P
```

```
DELETE PROCESS (P)
```

3.2.5 名称列表管理

MQSC 用 DEFINE NAMELIST 来创建名称列表，DELETE NAMELIST 来删除名称列表。ALTER NAMELIST 来修改名称列表属性，DISPLAY NAMELIST 来显示名称列表属性。

例：

```
// 创建名称列表 QNL，含两个名称 Q1 和 Q2
```

```
DEFINE NAMELIST (QNL) NAMES (Q1, Q2) REPLACE
```

```
// 修改名称列表 QNL，含三个名称 Q1、Q2 和 Q3
```

```
ALTER NAMELIST (QNL) NAMES (Q1, Q2, Q3) REPLACE
```

```
// 删除名称列表 QNL
```

```
DELETE NAMELIST (QNL)
```

3.2.6 认证信息管理

MQSC 用 DEFINE AUTHINFO 来创建认证信息 ,DELETE AUTHINFO 来删除认证信息。
ALTER AUTHINFO 来修改认证信息属性 ,DISPLAY AUTHINFO 来显示认证信息属性。

例：

```
// 创建认证信息定义 AI，指向 LDAP Server 的地址与端口
```

```
DEFINE AUTHINFO (AI) AUTHTYPE (CRLLDAP) CONNAME ('127.0.0.1 (4000)')
```

```
// 显示认证信息定义 AI 的属性
```

```
DISPLAY AUTHINFO (AI)
```

```
// 修改认证信息定义 AI，指向另一个 LDAP Server 的地址与端口，且设定了用户名与密码
```

```
ALTER AUTHINFO (AI) CONNAME ('127.0.0.1 (5000)') LDAPUSER (user) LDAPPWD (pwd)
```

```
// 删除认证信息定义 AI
```

```
DELETE AUTHINFO (AI)
```

3.2.7 小结

这一节列举了 WebSphere MQ 对象管理中常用的 MQSC 命令 ,表 3-2。我们通常可以使用这些命令对各种 MQ 对象实现创建、删除、修改属性、显示属性等功能，所有的 MQSC 命令都可以通过脚本文件的方式提交执行。对于更详细的语法说明可以参见附录“MQSC 命令一览表”。

表 3-2 MQSC 队列管理命令

命令	说明
队列管理器管理	
ALTER QMGR	修改队列属性
DISPLAY QMGR	显示队列属性
队列管理	
DEFINE QLOCAL/QREMOTE/QALIAS/QMODEL	创建队列
DELETE QLOCAL/QREMOTE/QALIAS/QMODEL	删除队列
ALTER QLOCAL/QREMOTE/QALIAS/QMODEL	修改队列属性
DISPLAY QLOCAL/QREMOTE/QALIAS/QMODEL/QUEUE	显示队列属性
通道管理	
DEFINE CHANNEL	创建通道
DELETE CHANNEL	删除通道
ALTER CHANNEL	修改通道属性

DISPLAY CHANNEL	显示通道属性
START CHANNEL	启动通道
STOP CHANNEL	停止通道
RESET CHANNEL	重置通道
RESOLVE CHANNEL	解决通道争议
PING CHANNEL	测试通道
进程定义管理	
DEFINE PROCESS	创建进程定义
DELETE PROCESS	删除进程定义
ALTER PROCESS	修改进程定义属性
DISPLAY PROCESS	显示进程定义属性
名称列表管理	
DEFINE NAMELIST	创建名称列表
DELETE NAMELIST	删除名称列表
ALTER NAMELIST	修改名称列表属性
DISPLAY NAMELIST	显示名称列表属性
认证信息管理	
DEFINE AUTHINFO	创建认证信息
DELETE AUTHINFO	删除认证信息
ALTER AUTHINFO	修改认证信息属性
DISPLAY AUTHINFO	显示认证信息属性

3.3 基本队列操作

我们在了解了 MQ 控制命令和管理命令后，就可以自己构建 MQ 运行环境了。先创建并启动队列管理器，再创建 MQ 队列对象，然后通过队列操作对消息进行存取。如果您要配置 MQ 的多机运行环境，可转去“4.1 通道配置”。

WebSphere MQ 提供了一些缺省的基本队列操作命令，如表 3-10。通过它们可以对队列进行最基本的读写操作，这些命令实际上都是 WebSphere MQ 应用程序，在安装目录中可以找到它们的源代码。

表 3-3 MQ 基本队列操作命令

命令	格式	说明
amqsput	amqsput QueueName [QueueManagerName]	从 Server 端将消息放入队列
amqsputc	amqsputc QueueName [QueueManagerName]	从 Client 端将消息放入队列
amqsget	amqsget QueueName [QueueManagerName]	从 Server 端将消息取出队列
amqsgetc	amqsgetc QueueName [QueueManagerName]	从 Client 端将消息取出队列
amqsbcg	amqsbcg QueueName [QueueManagerName]	从 Server 端查看消息
amqsbcgc	amqsbcgc QueueName [QueueManagerName]	从 Client 端查看消息

这些基本队列操作命令分成两类，一类在 MQ Server 端运行，它们是 amqsput、amqsget、amqsbcg。另一类在 MQ Client 端运行，它们是 amqsputc、amqsgetc、amqsbcgc。命令执行的语法都类似，以 amqsput 为例，格式为：*amqsput QueueName [QueueManagerName]*，其中 QueueName 为队列名，QueueManagerName 为队列管理器，如果不指明则为缺省队列管

理器。例：

```
C:\> amqspu t Q QM
C:\> amqsge t Q QM
C:\> amqsbcg Q QM
```

amqspu t 和 amqsputc 可以将消息放入队列中，程序把之后的每一行标准输入作为一条独立的消息，读到 EOF 或空行时退出。注意，UNIX 上的 EOF 为 Ctrl+D，Windows 上的 EOF 为 Ctrl+Z。可以将标准输入重定向到文件。队列中每放入一条消息，队列深度增加一。

amqsge t 和 amqsgetc 可以将消息从队列中全部读出并显示。读空后再等待 15 秒，在这段时间内如果有新的消息到达会一并读出。如果强行中断该程序，比如用 Ctrl+C 强行退出，这时等待着的 MQGET 读操作尚未完成，用 MQSC 命令 *DISPLAY QSTATUS(Q) TYPE(HANDLE) OPENTYPE(OUTPUT) ALL* 也可以观察到。MQGET 读操作会在一段时间后自动撤消，在这段时间内如果有新的消息到达，则第一条消息会被隐式地读走而丢失。amqsge t 和 amqsgetc 执行后队列应该为空，即队列深度为零。

amqsbcg 和 amqsbcbg 可以将详细查阅队列中现有的消息属性及内容而不将其取出。它与 amqsge t 和 amqsgetc 唯一的差别就是查阅后消息仍然保留在队列中，队列深度不变。

3.4 MQ 配置信息

WebSphere MQ 的配置信息反映了当前 MQ 运行环境的设置。在 UNIX 中这些信息记录在配置文件中，在 Windows 中记录在系统注册表中。通常情况下，MQ 提供了管理命令或工具来修改这些设置，建议用户不要轻易地进行手工修改。

3.4.1 UNIX 配置文件

UNIX 平台上的 WebSphere MQ 配置文件有两种。一种为 mqs.ini，它是针对整个 MQ 运行环境的，一台机器只有一个这样的文件。另一种为 mq.ini，它是针对某个队列管理器配置的，每个队列管理器有一个这样的文件。管理员可以编辑这些文件以修改配置。

配置文件是分段的，每一段 (stanza) 有一个标题，以冒号 (:) 结尾。每一段中有多条属性设置 (Entry)，它们都是“属性名=属性值” (Name=Value) 的格式。这里，缺省情况下配置文件中的段和设置都是比较精简的，不需要的部分可以不出现。在实际运行环境下，段和设置都是可以根据需要增减的。配置文件中可以有哪一些段，各段中可以有哪一些设置，在 WebSphere MQ 中都有规定。段不分先后次序，段中的设置也不分先后次序。

下面以 WebSphere MQ for AIX 为例，让我们看一看缺省的 mqs.ini 和 qm.ini (队列管理器名为 QM)。

3.4.1.1 mqs.ini

```
AllQueueManagers:
```

```

DefaultPrefix=/var/mqm
ClientExitPath:
  ExitsDefaultPath=/var/mqm/exits
LogDefaults:
  LogPrimaryFiles=3
  LogSecondaryFiles=2
  LogFilePages=1024
  LogType=CIRCULAR
  LogBufferPages=0
  LogDefaultPath=/var/mqm/log
QueueManager:
  Name=QM
  Prefix=/var/mqm
  Directory=QM

```

在 mqs.ini 配置文件中规定了整个 WebSphere MQ 的工作目录为 /var/mqm ,队列管理器缺省有 3 个主 Log 文件 ,2 个备用 Log 文件。每个 Log 文件大小为 1024 页 (一页为 4KB) , Log 文件缺省采用环形模式组织 ,且在 /var/mqm/log 目录下。

当前有一个队列管理器 ,名为 QM。它的工作目录为 /var/mqm 下的 QM ,即/var/mqm/QM 目录。

3.4.1.2 qm.ini

```

ExitPath:
  ExitsDefaultPath=/var/mqm/exits/
Log:
  LogPrimaryFiles=3
  LogSecondaryFiles=2
  LogFilePages=1024
  LogType=CIRCULAR
  LogBufferPages=0
  LogPath=/var/mqm/log/QM/
  LogWriteIntegrity=TripleWrite

```

在 qm.ini 配置文件中规定了队列管理器 QM 的一些参数。它有 3 个主 Log 文件 ,2 个备用 Log 文件。每个 Log 文件大小为 1024 页 (一页为 4KB) ,Log 文件缺省采用环形模式组织 ,且在 /var/mqm/log/QM 目录下 ,Log 用 TripleWrite 方式保持日志完整性。注意 ,qm.ini 中可能有一些参数名与 mqs.ini 相同 ,但设置值不同。则对于该队列管理器而言 ,以 qm.ini 为准。

3.4.2 Windows 注册表

在 Windows 注册表中 ,路径中的一个目录结点称为一个项 (Item) ,一个项中可以含多

个其它项或设置，每个设置都是“名=值”的格式。这里最下一级的项就相当于 UNIX 配置文件中的段，项中的设置相当于 UNIX 配置文件中的设置。在配置运行环境时，可以相互参考。

与 UNIX 类似，WebSphere MQ for Windows 的属性设置也分两类，一类是针对整个 MQ 运行环境的，另一类是针对队列管理器的。它们可以通过编辑注册表来修改，也可以通过“WebSphere MQ 资源管理器”来设置或修改。以下介绍后一种方法的操作及相应的注册表项。

无论是 UNIX 的配置文件或是 Windows 的注册表，对于日志文件的属性在运行环境和队列管理器配置中各有一份设置。其中运行环境中的设置犹如一份模板，在队列管理器创建时如果不特别指明日志参数则将其拷贝到队列管理器中配置中，以后就只有队列管理器配置有作用了。

3.4.3 Windows 中 MQ 运行环境配置

在“WebSphere MQ 服务”中右键点击“WebSphere MQ 服务 (本地)”，然后选择“属性”，即可以配置整个 MQ 运行环境的属性(图 3-1)。在 Windows 中对这些属性的配置都会映射到注册表中。



图 3-1 WebSphere MQ 服务

所有队列管理器	<MQ_HKEY>\Configuration\AllQueueManagers\
缺省前缀	DefaultPrefix
转换 EBCDIC 新行	ConvEBCDICNewline，可以取值 NT_TO_LF、TABLE、ISO
客户机出口路径	<MQ_HKEY>\Configuration\ClientExitPath
缺省日志设置	<MQ_HKEY>\Configuration\LogDefaults\
日志类型	LogType，可以取值 CIRCULAR(循环)或 LINEAR(线性)

日志路径	LogPath
日志文件页面数 (每页 4KB)	LogFilePages
主日志文件数	LogPrimaryFiles
次日志文件数	LogSecondaryFiles
日志缓冲区页面数	LogBufferPages
ACPI 设置	<MQ_HKEY>\Configuration\ACPI\
API 出口	<MQ_HKEY>\Configuration\
公共出口	ApiExitCommon
模块出口	ApiExitTemplate

Win2000 支持 Advanced Configuration and Power Interface (ACPI) 标准，这使得操作系统进入挂起 (Suspend) 模式或者从挂起模式恢复的时候有机会保护正在运行的通道。

3.4.4 Windows 中 MQ 队列管理器配置

在 “WebSphere MQ 服务” 中右键点击队列管理器，然后选择 “属性”，即可以配置队列管理器的各种属性。在 Windows 中对这些属性的配置都会映射到注册表中。



图 3-2 队列管理器属性配置

常规

启动类型	<MQ_HKEY_Services>\队列管理器\Startup 取值为 1 表示将队列管理器配置为自动启动
缺省队列管理器	<MQ_HKEY>\Configuration\DefaultQueueManager\Name

	一台机器只能有一个缺省队列管理器
恢复	<MQ_HKEY_Services>\队列管理器
	在失败时可以设置重新启动方式和重试次数
日志	<MQ_HKEY_QM>\Log\
日志类型	LogType, 可以取值 CIRCULAR(循环)或 LINEAR(线性)
日志路径	LogPath
日志文件页面数 (每页 4KB)	LogFilePages
主日志文件数	LogPrimaryFiles
次日志文件数	LogSecondaryFiles
日志缓冲区页面数	LogBufferPages
写日志完整性	LogWriteIntegrity, 可以取值 SingleWrite、DoubleWrite、TripleWrite 来保证消息完整性
资源	<MQ_HKEY_QM>\XAResourceManager
交换文件	SwitchFile
XA 打开字符串	XAOpenString
XA 关闭字符串	XACloseString
线程控制	ThreadOfControl, 可以取值 PROCESS 或 THREAD
服务	<MQ_HKEY_QM>\Service\
通道	<MQ_HKEY_QM>\Channels\
通道最大数	MaxChannels
活动通道最大数	MaxActiveChannels
启动程序最大数	MaxInitiators
出口	<MQ_HKEY_QM>\<QMGr>\
TCP	<MQ_HKEY_QM>\TCP\
Netbios	<MQ_HKEY_QM>\Netbios
SPX	<MQ_HKEY_QM>\SPX
LU62	<MQ_HKEY_QM>\LU62

MQ 队列管理器的参数配置可以设定队列管理器的运行方式和参数。比如,通过配置资源,可以使队列管理器连接数据库,并在 MQ 交易中含数据库操作。通过配置通道,可以设定队列管理器的最大通道数、最大活动的通道等限制。通过配置出口,可以设定出口程序路径等等。

3.5MQ 管理方式

WebSphere MQ 的管理方式可以分成本地管理和远程管理器两种。无论是哪一种方式,都可以通过命令行或图形界面工具来完成。对于 Windows 平台有“ WebSphere MQ 服务 ”工具进行配置,对于编程方式,可以发送 PCF 命令来实现管理。

本地管理指通过管理工具对本地的队列管理器实施管理,即管理工具与被管理的队列管理器在同一台机器上。远程管理则是通过一定的设置,用相同的管理工具管理另一台机器上的队列管理器。

3.5.1 本地管理

3.5.1.1 用 RUNMQSC 实现本地管理

RUNMQSC 是一个通用的 MQ 对象管理工具，使用 MQSC 命令集可以对 MQ 对象进行全方位的管理，也是各种管理方式最直接、最全面的一种。

在队列管理器所有机器的命令行中执行 `RUNMQSC <QMgrName>` 即可进入 RUNMQSC 的交互界面，其中 `<QMgrName>` 是本地的队列管理器名。

3.5.1.2 用 MQ 资源管理器实现本地管理

Windows 平台的 WebSphere MQ 提供了资源管理器这样一个图形界面管理工具（图 3-3），它是借助 Microsoft Management Console (MMC) 环境进行管理的，使用十分方便。其中列出了所有被管理的队列管理器，缺省情况下它们都是本地队列管理器，当然也可以通过远程管理将远程队列管理器纳入其中。



图 3-3 WebSphere MQ 资源管理器

在队列管理器下列出了所有的 WebSphere MQ 对象，用户可以点击右键，在菜单中选择各种操作，简单易懂。使用 WebSphere MQ 资源管理器来管理 MQ 对象之前必须启动相应队列管理器的命令服务器。

3.5.1.3 用 MQ 服务实现本地管理

除了对队列管理器中的对象进行管理之外，我们还需要对队列管理器自身以及整个 MQ 运行环境进行管理。为此，WebSphere MQ for Windows 提供了另一个管理工具：WebSphere MQ 服务。它也是借助 Microsoft Management Console (MMC) 环境进行管理的，

使用同样方便。图 3-5

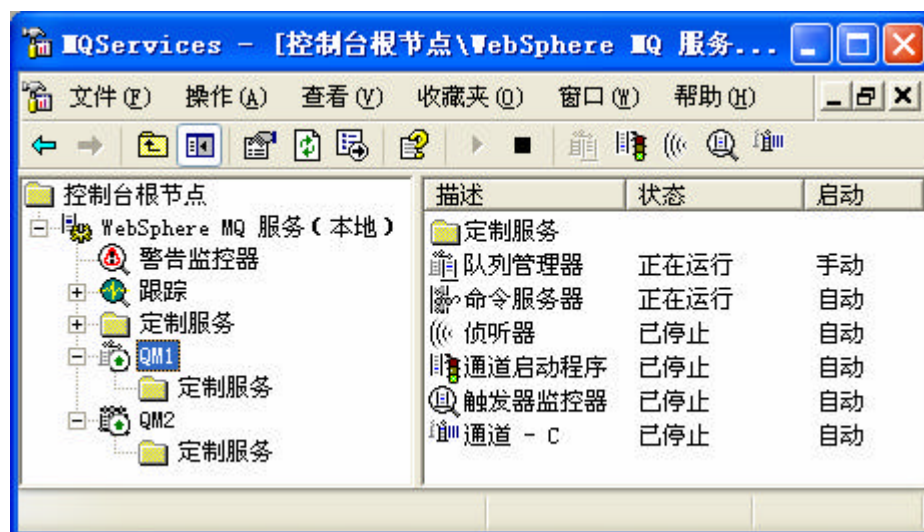


图 3-5 WebSphere MQ 服务

在界面中可以为队列管理器配置各种服务，缺省有“队列管理器”和“命令服务器”，此外还有“侦听器”、“通道启动程序”、“触发器监控器”和“通道”。其中，前两种每个队列管理器只能各配置一个，后四种每个队列管理器可以配置多个。此外，通过配置定制服务，可以配置在队列管理器启动前或后自动执行的命令，它可以是调用一条命令，也可以是以进程方式独立运行的程序。

正如我们前面提到的，WebSphere MQ 中有一些属性是针对队列管理器的，另有一些属性是针对整个 MQ 运行环境的。在 UNIX 中它们分别设置在 `qm.ini` 和 `mqs.ini` 文件中，在 Windows 中它们设置在注册表里，这些属性也可以通过 WebSphere MQ 资源管理器设置或修改。

3.5.1.4 用 PCF 编程实现本地管理

WebSphere MQ 资源管理器实际上把管理指令以 PCF 消息的格式放入被管理的队列管理器的 `SYSTEM.ADMIN.COMMAND.QUEUE` 中，由该队列管理器上的命令服务器执行并返回结果，再在图形界面上表现出来。WebSphere MQ 远程管理实际上也是这个原理。

PCF 消息遵循一定的格式，如果通过编程人为地构造一条 PCF 消息并用 `MQPUT` 放入 `SYSTEM.ADMIN.COMMAND.QUEUE` 中，就可以编程向队列管理器发送一条管理指令了。消息中设置 `ReplyToQ` 指定结果的返回队列，命令服务器在执行完毕后会返回一条结果消息，该消息也遵循同样的格式，也是一条 PCF 消息。基于这个原理，我们可以通过 PCF 编程在程序中随心所欲控制并管理队列管理器。如果管理指令来自远端，则可以很方便地实现远程管理。

3.5.2 远程管理

WebSphere MQ 的远程管理的原理是将管理命令包装成 PCF 消息并传送到远程队列管理器的命令队列 `SYSTEM.ADMIN.COMMAND.QUEUE` 中，命令服务器将管理消息读走并执行再

将执行结果也包装成 PCF 格式的消息并返回。图 3-6。返回队列由管理消息的 ReplyToQ 域指定。另外，要实现远程管理，被管理的队列管理器的命令服务器必须首先启动。

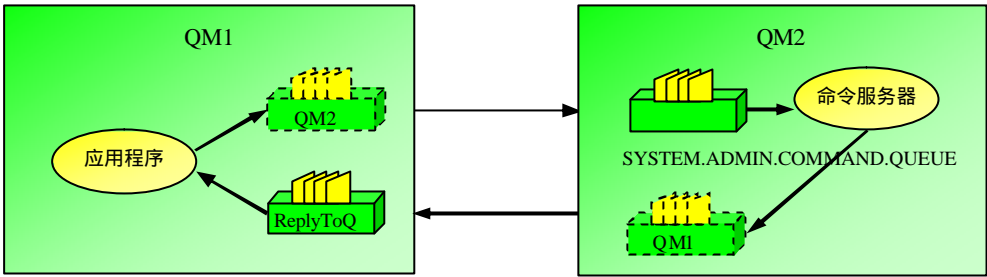


图 3-6 MQ 远程管理的原理

下面让我们仔细了解一下用 RUNMQSC 和用 MQ 资源管理器两种方式各自是如何进行远程管理的。

3.5.2.1 用 RUNMQSC 实现远程管理

我们可以用通用的 RUNMQSC 命令行管理工具对远程队列管理器实施管理。这里假定 QM1 为本地缺省队列管理器，QM2 为受管理的远程队列管理器。步骤如下：

1. 确认队列管理器上有 SYSTEM.ADMIN.COMMAND.QUEUE 队列，并启动远程的队列管理器的命令服务器进程

```
strmqcsv QM2
```

2. 在本地建立缺省队列管理器，并与远程队列管理器建立双向通道

QM1:

```
DEFINE CHANNEL (C_QM1.QM2) +
CHLTYPE (SDR) +
TRPTYPE (TCP) +
CONNAME ('10.10.10.21 (1416)') +
XMITQ (QM2) +
REPLACE
DEFINE CHANNEL (C_QM2.QM1) +
CHLTYPE (RCVR) +
TRPTYPE (TCP) +
REPLACE
DEFINE QLOCAL (QM2) +
USAGE (XMITQ) +
REPLACE
```

QM2:

```
DEFINE CHANNEL (C_QM1.QM2) +
CHLTYPE (RCVR) +
TRPTYPE (TCP) +
REPLACE
```

DEFINE	CHANNEL	(C_QM2.QM1)	+
	CHLTYPE	(SDR)	+
	TRPTYPE	(TCP)	+
	CONNAME	('10.10.10.11 (1415)')	+
	XMITQ	(QM1)	+
	REPLACE		
DEFINE	QLOCAL	(QM1)	+
	USAGE	(XMITQ)	+
	REPLACE		

3. 本地和远程队列管理器都要有相应的 TCP/IP Listener。

```
runmqclsr -m QM1 -t tcp -p 1415
```

```
runmqclsr -m QM2 -t tcp -p 1416
```

4. 通过 RUNMQSC 进行管理，这时 RUNMQSC 工具首先连接本地缺省队列管理器，再通过事先配置并连接好的通道，把命令发往指定的队列管理器，命令执行结果原路返回。由于涉及队列管理器之间的通信，可以用 -w 选项设定超时时间（秒），如果在规定的时间内命令结果没有返回，则执行失败。

```
runmqsc -w 100 QM2 < mqsc.tst > result.out
```

3.5.2.2 用 MQ 资源管理器实现远程管理

我们可以用 Windows 下的 WebSphere MQ 资源管理器进行远程监控，从而利用该管理工具友好的图形界面进行远程管理。步骤如下：

1. 确认队列管理器上有 SYSTEM.ADMIN.COMMAND.QUEUE 队列，并启动远程的队列管理器的命令服务器进程

```
strmqcsv <QMgr>
```

2. 远程的队列管理器至少有一个 TCP/IP Listener，可以是 runmqclsr，也可以是 inetd。参见“配置”章节。

```
runmqclsr -m <QMgr> -t tcp -p <port>
```

3. 在远程的队列管理器上配置名为 SYSTEM.ADMIN.SVRCONN 的 Server-Connection 通道

DEFINE	CHANNEL	(SYSTEM.ADMIN.SVRCONN)	+
	CHLTYPE	(SVRCONN)	+
	TRPTYPE	(TCP)	+
	MCAUSER	('mqm')	+
	REPLACE		

4. 在 Windows 平台上的 WebSphere MQ 资源管理器中点击“队列管理器”，右键选择“显示队列管理器”。在弹出菜单中选中“显示远程队列管理器”，给出队列管理器名称和连接名称，其中连接名称要与通道的 CONNAME 属性相符，需要含有远程队列管理器的地址和监听端口，可以是 IP (Port) 格式。见图 3-6

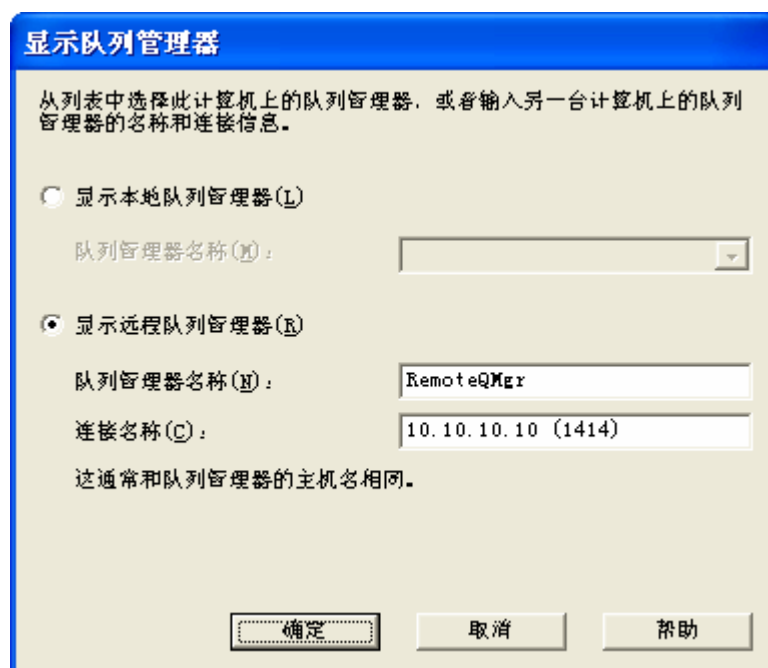


图 3-6 用 WebSphere MQ 资源管理器远程管理

3.6 日志 (Log)

WebSphere MQ 中日志为系统的稳定运行和消息的可靠传递提供了保障，在 WebSphere MQ 中扮演着不可缺少的角色。基本上，MQ 的对象操作及对持久消息的操作都会记入日志中，在系统故障时日志可以帮助 MQ 队列管理器在重新启动后恢复到原来的状态，所有的持久消息仍然保留不丢失。日志也可以用来将整个 MQ 对象整体记录，以便日后进行对象整体恢复。因为日志极其重要，所以对日志文件的定期备份就显得十分必要。

3.6.1 队列管理器日志

队列管理器的日志是随着队列管理器一起创建的，在创建队列管理器时可以指定日志属性，其中一部分属性也可以在队列管理器创建后再修改调整。对于创建时未指定的参数，WebSphere MQ 使用缺省参数值。

日志中记录了队列管理器的各种活动，包括创建和删除对象（除通道外）、持久性消息、交易状态、对象状态的更改、通道的活动等等。所以，在 WebSphere MQ 系统出了问题的时候，日志可以用来做恢复。

- 重启恢复 (Restart Recovery) 在重新启动 WebSphere MQ 后保证所有的持久性消息都还在，所有的交易回滚到发生前的状态，删除未提交的消息。由于非持久消息是不记录日志的，所以恢复只针对持久消息。
- 损坏恢复 (Crash Recovery)，WebSphere MQ 由于内部错误非正常停止，重新启动后恢复到检查点 (CheckPoint)，即队列管理器与日志一致点。
- 介质恢复 (Media Recovery)，在介质损毁修复后，根据日志修复出对象。

3.6.1.1 日志文件

WebSphere MQ 日志分成两部分：多个日志数据文件和一个日志控制文件。

MQ 的日志数据文件在 Windows 中缺省在 `<InstallDir>\log\<QMgrName>\active` 目录下，每个文件缺省大小为 1MB。在 UNIX 中，缺省在 `/var/mqm/log/QmName/active` 目录下，每个文件缺省大小 4MB。

数据文件有主数据文件 (Primary) 和次数据文件 (Secondary) 之分，它们在文件结构和大小上都是相同的。队列管理器初始启动的时候只会创建指定数量的主数据文件，当需要的时候（比如含有众多消息操作的长交易）会一个一个创建出次数据文件，加入到日志数据文件中。当不再需要时，次数据文件又被会动态删除。如果所有的次数据文件也都用完了，仍不能满足系统记录日志的需要，则系统对任何日志操作都将返回 MQRC_RESOURCE_PROBLEM。主次数据文件的数量由创建队列管理器时的 LogPrimaryFiles 和 LogSecondaryFiles 指定。

MQ 的日志控制文件在 Windows 中为 `<InstallDir>\log\<QMgrName>\amqhlctl.lfh`，在 UNIX 中，为 `/var/mqm/log/<QMgrName>\amqhlctl.lfh`。日志控制文件中记录了目前数据文件的主次分配、活动状态等信息。

日志的空间是有限的，理论上最大占用空间为：(主日志数量 + 次日志数量) x 日志文件大小。如果有长交易占用了太多的日志空间，系统会将其强行回滚，以释放日志空间。该交易所在的应用无法在同一个交易中再使用 MQPUT 或 MQGET，MQGET 会返回 MQRC_BACKED_OUT。应用可以用 MQCMIT（返回 MQRC_BACKED_OUT）或 MQBACK 结束交易，从而创建新的交易。

3.6.1.2 环形和线性 (Circular & Linear)

WebSphere MQ 中的日志分为两种：环形日志 (Circular Logging) 和 线性日志 (Linear Logging)

环形日志可以用来做重启恢复 (Restart Recovery)。它记录了当前所有的交易状态和交易中的持久消息操作，维护了所有重启相关需要的数据。

线性日志不仅可以用来做重启恢复 (Restart Recovery)，还可以用来做介质恢复 (Media Recovery)。它不但记录了线性日志的所有内容，还因为日志文件集合是线性增长而不会循环覆盖，队列管理器创建以来所有持久消息的活动都记录在案。所以，如果队列文件坏了，系统可以凭借完整的日志记录重建对象和消息。

3.6.1.3 日志参数

LogPrimaryFiles=3 | 2-62

主日志数据文件数量。缺省为 3，最小 2，最大 62。主次日志文件之和不大于 63，不小于 3

LogSecondaryFiles=2 1-61	次日志数据文件数量。缺省为 2，最小 1，最大 61。主次日志文件之和不大于 63，不小于 3
LogFilePages=number	日志文件大小，以 4KB 为单位。 Windows 平台，缺省为 256，即 1MB。最小 32，最大 16,384 UNIX 平台，缺省为 1024，即 4MB。最小 64，最大 16,384
LogType=CIRCULAR LINEAR	日志类型。缺省为环形日志，CIRCULAR
LogBufferPages=0 0-512	日志缓冲区大小，以 4KB 为单位。最小 18，最大 512。缺省为 0。如果值为 0-17，WebSphere MQ 实际使用 18，即 72KB。
LogPath=dir	日志文件目录
LogWriteIntegrity=SingleWrite DoubleWrite TripleWrite	日志完整性
● SingleWrite	只写一次日志，在高可靠环境下，需要硬件保证，比如 SSA Cache
● DoubleWrite	可能需要写两次
● TripleWrite	可能需要写三次，缺省值。安全性高，效率较低

这些日志参数中，有些在队列管理器创建后就不能改动了。比如 LogType，LogPath，LogFilePages 等等。有些则可以调整，在队列管理器重启后生效。比如 LogPrimaryFiles，LogSecondaryFiles，LogBufferPages 等等。

一般来说，日志文件较大而文件数量较少与文件较小而数量众多相比，更有效率。LogBufferPage 的值较大意味着吞吐量较大，适合于大消息和大交易环境。LogWriteIntegrity 的选取在大容量、高压力的环境中在很大程度上会影响运行性能。

在生产系统中，为了在必要的时候进行介质恢复，在创建队列管理器时通过指定 Log Path 可以把日志文件指定到其它硬盘上去，与队列管理器对象工作目录不在同一块硬盘上。这样，如果有一块硬盘损坏了，系统可以修复。表 3-15 统计了日常操作与日志长度的关系，可以据此估算系统所需的日志文件大小和数量。

表 3-15 WebSphere MQ 操作与日志中记录长度的关系

WebSphere MQ 操作	日志中记录长度
Put (持久消息)	750 字节 + 消息长度
Get	260 字节
Commit 提交	750 字节
Rollback 回滚	1000 字节 + 每个 Get/Put 需要 12 字节
创建对象	1500 字节
删除对象	300 字节
改变属性	1024 字节
Record media image	800 字节 + Image 大小
Checkpoint	750 字节 + 每个活动的交易需要 200 字节

3.6.2 检查点 (Checkpoint)

WebSphere MQ 的检查点是一组记录，含有重启队列管理器必需的信息，检查点标志着队列 (queue) 与日志 (log) 在这一时刻是一致的，检查点之前已经提交的交易被认为在队列

中体现了，所以检查点之前的日志文件在做重启恢复 (Restart Recovery) 时不需要。WebSphere MQ 中，我们把日志文件中做重启恢复所必需的部分称为活动日志 (Active Log)，通常它们是最近一次检查点以来的所有日志，我们把其它部分称为非活动日志，所以最近一次检查点之前的日志为非活动日志。如果是环形日志，非活动日志可以被覆盖重用。如果是线性日志，非活动日志可以被存档 (Archive)，保存在其它介质中。

WebSphere MQ 对于队列上的任何一个持久消息，都是先记日志再入队列，检查点不会插在这两个动作之间。所以，检查点意味着日志与队列在这个时间点上是一致的。WebSphere MQ 在满足以下任何一个条件的时候自动产生检查点：

- 队列管理器启动
- 队列管理器停止
- 日志空间不足，需要主动释放
- 每 10,000 次日志操作 (在 z/OS 平台上，这个数字可以由 LOGLOAD 参数控制)

在出现非活动日志文件后，系统会自动生成出相应数量的新日志数据文件，补齐活动日志文件的数量。对于非活动日志文件，管理员可以将其另行存放，也可以在对 WebSphere MQ 做备份后删除它们。因为介质恢复 (Media Recovery) 可以由备份文件及备份点之后的日志文件完成。

3.6.3 记录和复原 (Record & Recover)

记录和复原功能只对线性日志有效，可以用于介质恢复。记录映像 (Record Image) 就好像给 WebSphere MQ 对象拍照，将内容记入日志中，拍照之前的日志内容对于该对象来说就可以不必保留了。复原映像则是一个逆过程，从日志中找到最近的照片，并按其后的日志内容进行相应的修补，将对象复原到最近的样子。

rcdmqimg (Record Image) 命令和 rcrmobj (Recreate Object) 命令分别用来记录和复原 WebSphere MQ 对象的映像快照 (Image)。rcdmqimg 将快照记入日志文件，rcrmobj 将最近的一次快照从日志文件中取出，并且按以后日志文件中的内容将对象恢复。快照在恢复时可能有两种情况：

- 1) 如果做过快照，则按最近的一次快照所在的日志文件及其以后的所有日志文件来恢复
- 2) 如果没有做过快照，则按队列管理器创建以来的所有日志文件来恢复

WebSphere MQ 在启动时如果有对象损坏，会试图按照快照自动恢复。如果启动时未发现对象损坏，队列管理器启动成功，接着在运行时发现了，这时只有靠手工 rcrmobj 恢复。注意：rcdmqimg 和 rcrmobj 都要在队列管理器运行时操作。

系统会定期自动打印 AMQ7467 和 AMQ7468 信息，记录到错误日志中。使用 rcdmqimg 命令也可以主动生成这些信息，如下，这些信息的内容会指明哪些较早的日志文件都没有在线保存的必要了。对于启动恢复所需的最早日志文件，其中含有最近的一次检查点。对于介质恢复所需的最早日志文件，其中含有最近的一次 rcdmqimg。例：

AMQ7467: 启动队列管理器 QM 所需的最早的日志文件是 S0000000.LOG .

AMQ7468: 执行队列管理器 QM 的介质恢复所需的最早的日志文件是 S0000001.LOG .

3.6.3.1 记录 (Record Image)

- 格式

```
rcdmqimg [-z] [-l] [-m QMgrName] -t ObjType [ObjName]
```

- 功能

rcdmqimg 命令用于记录对象的快照。

- 说明

选项 `-z` 表示抑制命令执行过程中的输出。`-l` 表示列出重启队列管理器和介质恢复所需的最早的日志文件名。`-m QMgrName` 指明对象所在的队列管理器。ObjType 是对象类型可以是 `qmgr`、`queue`、`process`、`namelist`、`authinfo` 等等，也可以用 `all` 或 `*` 表示选择所有的对象。ObjName 表示具体需要记录的对象名。

- 例：

```
rcdmqimg -m QM -t queue Q
```

3.6.3.2 复原 (Recreate Object)

- 格式

```
rcrmqobj [-z] [-m QMgrName] -t ObjType [ObjName]
```

- 功能

通过在日志中查取最近的快照并据此复原对象

- 说明

选项 `-z` 表示抑制命令执行过程中的输出。`-m QMgrName` 指明对象所在的队列管理器。ObjType 是对象类型可以是 `qmgr`、`queue`、`process`、`namelist`、`authinfo` 等等，也可以用 `all` 或 `*` 表示选择所有的对象。ObjName 表示具体需要记录的对象名。

- 例：

```
rcrmqobj -m QM -t queue Q
```

3.6.4 备份和恢复 (Backup & Restore)

备份和恢复是通过文件拷贝备份的手段完成的，对环形日志和线性日志都有效。如果是线性日志，在以下情况下，可以只恢复队列管理器的目录，而沿用原来的日志目录：

- 1) 备份时队列管理器处于检查点状态，即备份的对象与日志在备份点上是一致的
- 2) 日志目录中留有备份点以来的所有日志文件，即有备份点以后的完整变化记录

对于线性日志，WebSphere MQ 在备份后非活动日志可以认为是不必要了，所以在备份点之前的日志可以被删除或永久归档。

3.6.4.1 备份 (Backup)

- 1) 首先，队列管理器处于停止状态

- 2) 拷贝备份相关的目录及文件，包括：

- `<InstallDir>/QMgers/<QMgrName>` (包括所有的目录和文件)
- `<InstallDir>/log` (包括数据文件和控制文件)

3.6.4.2 恢复 (Restore)

- 1) 首先，队列管理器处于停止状态
- 2) 删除原来的相关目录及文件，再拷贝恢复相关的目录及文件，包括：
 - <InstallDir>/QMgers/<QMGrName> (包括所有的目录和文件)
 - <InstallDir>/log (包括数据文件和控制文件)

3.6.5 导出日志 (Dump Log)

用 `dmpmqlog` 命令可以将队列管理器的日志内容输出成文本。`dmpmqlog` 只有在队列管理器停止的时候可以执行，缺省输出上一个检查点以来的内容。由于在队列管理器停止的时候，会写入检查点，所以 `dmpmqlog` 通常只会输出较少的日志内容，当然也可以将指定一段日志的所有内容都输出。下面是 `dmpmqlog` 命令的使用方法。

- 格式
`dmpmqlog [-b | -s StartLSN] [-e EndLSN] [-f LogFilePath] [-m QMGrName]`
- 功能
将指定的日志文件导出并输出成文件
- 说明
-b 表示从第一个日志开始，StartLSN 表示开始的日志号，EndLSN 表示结束的日志号，它们都用来指定一段日志文件。LogFilePath 指定日志文件所在的目录。QMGrName 为队列管理器名。
- 例：

```
dmpmqlog -m QM > QMLog.dmp
```

第 4 章 通信与配置

WebSphere MQ 的互连通信就是将多个队列管理器通过通道配置连接在一起，使消息能够自动地传递到目标队列，形成协同工作的能力。通常情况下，互连配置的复杂度与消息传递路径中经过的队列管理器数量有关。对于最简单的情况，就是只有两个队列管理器，它们之间用通道直接连接。对于多个队列管理器，可以形成链式、树型或网状结构，通过配置技巧形成消息的自动路由。

4.1 消息路由

消息路由就是消息根据其目标地址（目标队列管理器和目标队列）转发传递的过程，如果事先做了恰当的配置，这一过程可以由 WebSphere MQ 自动完成。为了解释清楚，让我们先来了解一下消息的路由过程。

4.4.1 消息路由过程

消息的路由过程大致如下：

1. 消息在源队列管理器中被放入远程队列时，消息的头结构（MQMD）中并未包含消息的目的地，这时消息的结构为 MQMD + Body，且不含目标地址信息。
2. 由于远程队列只是虚的队列定义，指向的队列实体是传输队列。消息在放入传输队列的时候，会自动加上传输消息头，消息结构变为 MQMD + MQXQH + Body。其中，MQXQH 中有 RemoteQName 和 RemoteQMGrName 两个域，分别会自动填入远程队列的目标队列和目标队列管理器属性。这时消息含有目标地址信息。
3. 如果有通道对应与该传输队列，且通道启动，就意味着有 MCA 通信进程在读该传输队列并可以发送，这时消息就被读出并由该通道传送到下一个队列管理器。
4. 消息到达下一个队列管理器后，首先会验证是否到达目标队列管理器。如果队列管理器名与 RemoteQMGrName 一致，则说明到达了目的地，转而寻找目标队列。
5. 如果队列管理器中有名为 RemoteQMGrName 的远程队列定义，指向另一个远程队列管理器为 AnotherRemoteQMGrName，且该定义的远程队列属性为空，这说明目标队列管理器在该队列管理器环境中被重定义了，我们称之为队列管理器别名，真正的目标队列管理器为 AnotherRemoteQMGrName。消息替换目标地址后回到 4 重新验证。
6. 如果 4、5 都不满足，说明该队列管理器只是消息路径中的一个节点，消息需要被继续路由。如果在队列管理器上有名为 RemoteQMGrName 的传输队列，则消息被自动放入该传输队列，并经由相应的通道路由去下一站。如果队列管理器设有缺省传输队列，则消息被放入缺省传输队列中，并经由相应的通道路由去下一站。
7. 转到 4。
8. 消息在到达目标队列管理器后，会试图放入目标队列中。如果没有名为 RemoteQName 的目标队列，但有同名的别名队列，则会放入别名队列指向的目标队列中。
9. 如果在放入时发生问题，比如目标队列满、禁放、目标队列非本地队列，或根本不存在目标队列等等，消息会试图就近放入死信队列。当然，这与消息的持久性属性有关。

4.4.2 缺省传输队列

缺省传输队列的作用就是当队列管理器发现消息需要进一步路由，但又找不到预先定义的合适的传输队列时，为队列管理器指明缺省的路由路径。这时消息将被统统放入缺省传输队列，并由相应的通道路由去下一站。缺省传输队列由队列管理器的 DEFXMLTQ 属性定义。

设置缺省传输队列的想法很简单：凡是上一站来的消息，如果其目的地不是本站，则应该去下一站。缺省传输队列适合于单向的链式结构，对于无法传递的消息，可能会堆积在末端队列管理器的死信队列中。

4.4.3 队列管理器别名

如果远程队列定义中远程队列管理器名（RQMNAME）不为空，但远程队列名（RQNAME）为空，我们称之为队列管理器别名。设置队列管理器别名意味着目标队列管理器在新的环境中被重定义了，所以消息的目的地也就随之变化了。队列管理器别名用于在中途更改映射消息的目标队列管理器。

4.4.4 多级跳

无论是缺省传输队列还是队列管理器别名,都可以在消息传输路径上干预消息的传递和路由。通过两者的结合,可以使消息从源目标管理器经过多次路由中转到达目标队列管理器,我们称之为多级跳 (Multi-Hopping)。源和目标队列管理器之间没有直接连接,消息在路由的过程中只经过传输队列和通道的中转而不经中间程序的干预。

4.4.5 传输中的消息

消息在经过远程队列放入传输队列时会由队列管理器自动添加一个传输头 (MQXQH),且传输头中的内容会根据远程队列定义自动填写。如果说,原先的消息是 MQMD + Body,则放入传输队列的消息为 MQMD + MQXQH + Body,如下。

```
MQMD
  StrucId = [MD ]
  Version = [MQMD_VERSION_1]
  Report  = [MQMO_NONE]
  MsgType = [MQMT_DATAGRAM]
  Expiry  = [-1]
  Feedback = [MQFB_NONE]
  Encoding = [MQENC_NATIVE]
  CodedCharSetId = [1381]
  Format    = [MQXMIT ]
  Priority  = [0]
  Persistence = [MQPER_NOT_PERSISTENT]
  MsgId     = [...]
  CorrelId   = [...]
  BackoutCount = [0]
  ReplyToQ    = [ ]
  ReplyToQMgr = [QM ]
  UserIdentifier = [chenyux ]
  AccountingToken:
  Length = [22]
  Content = [...]
  Type = [MQACTT_NT_SECURITY_ID]
  ApplIdentityData = [ ]
  PutApplType = [MQAT_QMGR]
  PutApplName = [QM ]
  PutDate = [20031028]
  PutTime = [13091072]
  ApplOriginData = [ ]
  GroupId = [...]
  MsgSeqNumber = [1]
```

```

Offset = [0]
MsgFlags = [MQMF_NONE]
OriginalLength = [-1]
MQXQH
StrucId = [XQH ]
Version = [MQXQH_VERSION_1]
RemoteQName = [Q1 ]
RemoteQMgrName = [QM1 ]
StrucId = [MD ]
Version = [4295468]
Report = [4371216]
MsgType = [MQMT_DATAGRAM]
Expiry = [-1]
Feedback = [MQFB_NONE]
Encoding = [MQENC_NATIVE]
CodedCharSetId = [1381]
Format = [MQSTR ]
Priority = [0]
Persistence = [MQPER_NOT_PERSISTENT]
MsgId = [...]
CorrelId = [...]
BackoutCount = [0]
ReplyToQ = [ ]
ReplyToQMgr = [QM ]
UserIdentifier = [chenyux ]
AccountingToken:
Length = [22]
Content = [...]
Type = [MQACTT_NT_SECURITY_ID]
ApplIdentityData = [ ]
PutApplType = [MQAT_WINDOWS_NT]
PutApplName = [D:\WMQ\bin\amqsput.exe ]
PutDate = [20031028]
PutTime = [13091072]
ApplOriginData = [ ]
Body (Length = 3)
0000: 61 62 63 abc

```

这里有几点需要说明：

- 1) 队列管理器同时为消息生成相同的 MQMD.CodedCharSetId 和 MQXQH.CodedCharSetId ,通常会根据 MQPUT 时远程队列的 MQMD.CodedCharSetId 生成。
- 2) 队列管理器会生成 MQMD.Format = MQXMIT, 而 MQXQH.Format 会根据 MQPUT 时远程队列的 MQMD.Format 生成。

- 3) 队列管理器同时为消息生成 MQMD.MsgId、MQMD.CorrelId 和 MQXQH.MsgId，且为了表示 MQMD 与 MQXQH 之间的关系，MQMD.CorrelId = MQXQH.MsgId。
- 4) MQMD.PutApplType = MQAT_QMGR，MQMD.PutApplName = 本地队列管理器名。表示 MQMD 是由本地队列管理器 MCA 生成。MQXQH.PutApplType = MQAT_WINDOWS_NT，MQXQH.PutApplName = 应用程序名。表示 MQXQH 消息是由应用程序产生。
- 5) 队列管理器同时生成 MQMD.PutDate，MQMD.PutTime，MQXQH.PutDate，MQXQH.PutTime。

由于 1) 2) 是可以由程序设定的，所以可以模仿。3) 4) 5) 是队列管理器设定的，几乎无法模仿。如果直接将带有传输头的消息写入传输队列，也可以将消息内容传递出去，但对于系统设定消息属性往往是一片空白。参见本书例程 MQPutTransmissionQueue.c。

传输队列缺省是只写不读的，即使手工将队列属性改为 GET(ENABLED) 也不行，队列管理器过一段时间会将传输队列属性改回 GET(DISABLED)。

4.2 通道配置

正如我们前面提到的，WebSphere MQ 中通信双方的通道类型配对共有六种，我们在这里对其中基本的 Sender/Receiver、Server/Receiver、Server/Requester 和 Sender/Requester 四种方式给出配置脚本。另外两种类型 Sender-Connection/Receiver-Connection 可以参见“客户端”章节，Cluster-Sender/Cluster-Receiver 可以参见“群集”章节。

4.2.1 Sender (QM1) -- Receiver (QM2)

Sender/Receiver 通道是最常见的通道配置方式，Sender 作为通道的发送方也是通道连接的主动发起方，Receiver 作为通道的接收方也是通道连接的被动监听方。在 Receiver 端要配置并运行相应的监听器。

在以下的配置脚本中，通道连接两个队列管理器 QM1 和 QM2。其中，QM1 为 Sender，QM2 为 Receiver。在 QM1 上配置了远程队列 QR 和传输队列 QX，其中 QR 指向队列管理器 QM2 上的本地队列 QL，且 QR 与 QX 对应，即凡是要放入 QR 队列的消息，在加上传输消息头后直接放入 QX 中等待发送。QM1 上配置 Sender 通道需要指定对方的通信参数 (IP 地址和端口)，而这些参数必须与 QM2 上的监听器设置对应。Sender 通道与传输队列 QX 对应，表示凡是在 QX 中等待发送的消息最终都可以由该通道送出。双方通道必须同名。

在连接通道的时候，我们只需在 QM1 端启动通道 *start channel (C)*。

QM1:

DEFINE	QREMOTE	(QR)	+
	RNAME	(QL)	+
	RQMNAME	(QM2)	+

	XMITQ	(QX)	+
	REPLACE		
DEFINE	QLOCAL	(QX)	+
	USAGE	(XMITQ)	+
	REPLACE		
DEFINE	CHANNEL	(C)	+
	CHLTYPE	(SDR)	+
	TRPTYPE	(TCP)	+
	CONNNAME	('127.0.0.1 (1416)')	+
	XMITQ	(QX)	+
	REPLACE		

QM2:

DEFINE	QLOCAL	(QL)	+
	REPLACE		
DEFINE	CHANNEL	(C)	+
	CHLTYPE	(RCVR)	+
	TRPTYPE	(TCP)	+
	REPLACE		

```
start runmqslr -m QM2 -t tcp -p 1416
runmqsc QM1
    start channel (C)
amqspout QR QM1
```

4.2.2 Server (QM1) -- Receiver (QM2)

Server/Receiver 与 Sender/Receiver 类似。Server 端是消息的发送方，也是连接的发起方。在配置的时候，必须指定对方的通信参数，由 CONNNAME 设定。下例中，QM1 是消息的发送方，QM2 是消息的接收方。

QM1:

DEFINE	QREMOTE	(QR)	+
	RNAME	(QL)	+
	RQMNAME	(QM2)	+
	XMITQ	(QX)	+
	REPLACE		
DEFINE	QLOCAL	(QX)	+
	USAGE	(XMITQ)	+
	REPLACE		
DEFINE	CHANNEL	(C)	+
	CHLTYPE	(SVR)	+
	TRPTYPE	(TCP)	+

	CONNNAME	('127.0.0.1 (1416)')	+
	XMITQ	(QX)	+
	REPLACE		

QM2:

DEFINE	QLOCAL	(QL)	+
	REPLACE		
DEFINE	CHANNEL	(C)	+
	CHLTYPE	(RCVR)	+
	TRPTYPE	(TCP)	+
	REPLACE		

```
start runmqslsr -m QM2 -t tcp -p 1416
```

```
runmqsc QM1
```

```
start channel (C)
```

```
amqspout QR QM1
```

4.2.3 Server (QM1) -- Requester (QM2)

Server/Requester 通道也是一种较常见的通道配置方式，从消息流向来看，Server 作为消息的发送方，Requester 作为消息的接收方。但是从连接方式来看，Requester 却是连接的主动方，Server 是被动方。这种模式常用于动态 IP 地址的环境中，Server 是静态 IP 地址的服务器，Requester 的机器上网后自动分配到一个 IP 地址，所以是动态的，由 Requester 发起连接后接收数据。在本例中，由 QM2 (Requester) 启动通道 *start channel (C)*。

QM1:

DEFINE	QREMOTE	(QR)	+
	RNAME	(QL)	+
	RQMNAME	(QM2)	+
	XMITQ	(QX)	+
	REPLACE		
DEFINE	QLOCAL	(QX)	+
	USAGE	(XMITQ)	+
	REPLACE		
DEFINE	CHANNEL	(C)	+
	CHLTYPE	(SVR)	+
	TRPTYPE	(TCP)	+
	XMITQ	(QX)	+
	REPLACE		

QM2:

DEFINE	QLOCAL	(QL)	+
	REPLACE		

```

DEFINE      CHANNEL      (C)                +
            CHLTYPE      (RQSTR)            +
            TRPTYPE      (TCP)              +
            CONNAME      ('127.0.0.1 (1415)') +
            REPLACE

```

```
start runmqslsr -m QM1 -t tcp -p 1415
```

```
runmqsc QM2
```

```
    start channel (C)
```

```
amqspout QR QM1
```

4.2.4 Sender (QM1) -- Requester (QM2)

Sender/Requester 的连接过程稍微复杂一些，Requester 首先与 Sender 连接，在通知对方连接参数后连接断开。Sender 进行反向连接，消息也是反向传送的，即由 Sender 传给 Requester。这种反向连接的方式，称为回调连接 (Callback Connection)。这种模式也常用于做双向验证，即双方必须都要知道对方的通信参数才能完成回调连接。

QM1:

```

DEFINE      QREMOTE      (QR)                +
            RNAME        (QL)                +
            RQMNAME      (QM2)              +
            XMITQ        (QX)               +
            REPLACE
DEFINE      QLOCAL       (QX)                +
            USAGE        (XMITQ)            +
            REPLACE
DEFINE      CHANNEL      (C)                +
            CHLTYPE      (SDR)              +
            TRPTYPE      (TCP)              +
            CONNAME      ('127.0.0.1 (1416)') +
            XMITQ        (QX)               +
            REPLACE

```

QM2:

```

DEFINE      QLOCAL       (QL)                +
            REPLACE
DEFINE      CHANNEL      (C)                +
            CHLTYPE      (RQSTR)            +
            TRPTYPE      (TCP)              +
            CONNAME      ('127.0.0.1 (1415)') +
            REPLACE

```

```
start runmqtsr -m QM1 -t tcp -p 1415
start runmqtsr -m QM2 -t tcp -p 1416
runmqsc QM2
    start channel (C)
amqspout QR QM1
```

4.2.5 通道启动命令

通道的启动可以通过 MQSC 的 start channel 命令完成，也可以在命令行通过 runmqchl 控制命令完成，两者效果相同，在 Windows 中还可以用 MQ 服务配置成自动启动方式。

4.2.5.1 runmqchl

runmqchl 在通道连接的主动方使用，使用时需要指定队列管理器名和通道名。

- 格式
runmqchl [-m QMgrName] -c ChannelName
- 功能
启动通道
- 说明
选项 -m QMgrName 表示队列管理器名，缺省为缺省队列管理器。-c ChannelName 表示通道名。
- 例：

```
runmqchl -m QM -c Channel
```

4.2.5.2 用 MQ 服务配置通道启动

在 WebSphere MQ for Windows 产品中，可以通过 WebSphere MQ 服务（本地）工具来进行配置。在配置界面中右键点击启动了的管理队列管理器，新建→通道→指定启动通道名，会在注册表中配置相应的条目，如下。一个队列管理器可以配置多个通道启动命令，甚至为每一个通道配置一个启动命令，并全部配置成“自动启动”方式，这样队列管理器在启动后，所有的通道自动连通。

<MQ_HEKY>\Configuration\Services\<QMName>\通道 - <ChannelName>

4.2.6 通道监控程序

无论是用 MQSC 中 start channel 脚本命令还是用 runmqchl 的控制命令来启动通道，都需要输入命令完成。通道的启动也可以用通道监控程序自动触发完成，通道监控程序实际上是触发程序的一种特例，与普通队列触发程序的区别在于通道启动程序是监听传输队列的，一旦传输队列中有消息满足传输条件则自动激活通道并进行消息传送。

4.2.6.1 runmqchi

一个队列管理器可以配置多个通道启动程序，以监控不同的通道。可以用控制命令 runmqchi 启动通道监控，启动时需要指定队列管理器和初始化队列名并在传输通道上配置触发功能 (Trigger)。

- 格式

```
runmqchi [-m QMgrName] [-q InitQueueName]
```

- 功能

启动通道监控程序

- 说明

选项 -m QMgrName 表示队列管理器名,缺省为缺省队列管理器。-q InitQueueName 表示初始化队列名，缺省为 SYSTEM.DEFAULT.INITIATION.QUEUE。

- 例：

```
runmqchi -m QM -q InitQueue
```

4.2.6.2 用 MQ 服务配置通道监控程序

在 WebSphere MQ for Windows 产品中,可以通过 WebSphere MQ 服务 (本地) 工具来进行配置。在配置界面中右键点击启动了的管理队列管理器，新建→通道启动程序→指定启动队列名，会在注册表中配置相应的条目：

<MQ_HKEY>\Configuration\Services\<QMName>\启动通道程序

4.3 通道的属性

通道本质上是通信双方建立起来的通信连接，双方各有一个通信的工作进程 (MCA)，一对 MCA 之间可以配置通信协议及相应的参数。通道将双方的通信消息打包后传送，同时通过约定的机制来确保消息的传递的正确性，这种机制称为 MQ 消息协议 (MQ Message Protocol)。通道中消息的传递是单向的，但消息协议却是双向的，通信双方会对消息的发送和接收进行确认。

通道会对每一条传递的消息编号，可以进行批量发送，同时会监测对方是否反应正常。在通信连接断开后会自动恢复现场并试图重连。通道的所有这些功能都是通过通道属性的设置来完成的，通信双方在连接的时候会交换各自通道的属性并约定双方都能接受的值。通常情况下我们在改动双方通道属性设置后需要重新连接，以使设置生效。

通道的各种功能是可以通道的属性来设置和调节的，下面我们会详细介绍通道的功能与属性之间的关系。

4.3.1 通道会话

4.3.1.1 消息序号 (Message Sequence)

通道为每一条消息的传送分配了一个序列号，它会自动累计增值。通道的相关属性 SEQWRAP 表示序号的最大值，缺省为 999,999,999。序列号越界后自动归零，从头开始。

通道两端的消息序号应该是相同的，一旦连接断开后，通道重连时双方 MCA 会将消息序号同步。如果通信异常造成序号不一致，可以在通道发送端用 MQSC 命令 RESET CHANNEL SEQNUM 手工将两者同步。

4.3.1.2 批量 (Batch)

在两个队列管理器 A 和 B 之间的通道上传送消息时，在消息从 A 送达 B 后，会有应答数据从 B 返回 A，表示收到确认。WebSphere MQ 把这个过程叫做一次同步 (SyncPoint)。如果我们用网络工具侦听通道连接，就可以观察到同一个连接上的这个返回同步数据包。

在 WebSphere MQ 中消息可以配置成批量传送，这时的一个同步点可以对一批消息的到达进行确认，以提高传输效率。为了控制两个同步点之间的时间和传送的消息数量，WebSphere MQ 引入相应的属性进行控制 (表 4-1)。其中，批量间隔 (Batch Interval) 表示从本批次第一个消息到下一个同步点之间的最长的时间间隔，批量大小 (Batch Size) 表示一个批次，即两个同步点之间最大的消息数量。两个条件只要满足任意一个就会引发一个同步点，即一个批次结束。

表 4-1 批量相关的通道属性

属性描述	属性名	缺省值	单位	有效范围	说明
Batch Interval	BATCHINT	0	毫秒	0 - 999,999,999	0 表示不使用 Batch，即消息送出后立即提交
Batch Size	BATCHSZ	50	条	1 - 9,999	累计到该值后批量传送

批量并不会影响消息传送，消息总是在一旦准备好就送到远端，不会有任何故意的耽搁，批量只会影响同步的时间。一个大的批量设置 (Batch Interval 和 Batch Size 都比较大) 会增加吞吐量，减少需要同步的数据包。但是，这也会增加响应时间，因为消息到达对方后处于未同步提交 (uncommitted) 的状态时间加长了，所以程序等待消息的时间加长了。在网络出错时，回滚的数据量也相对加大。

Batch Size 是通道建立连接的时候取双方通道属性 Batch Size 较小的那一个，也有些平台的 MQ 取双方通道的 Batch Size 以及队列管理器的属性 MAXUMSGS，取四个值中最小的那一个。实际的批量大小因为有了 Batch Interval 的参与，会小于等于这个值。

在稳定的连接上频繁地传送消息，则适当增大 Batch Size 会增加性能。如果消息的传送请求是断断续续的，则 Batch Size = 1 比较合适。通道属性中有 NPMSPEED 可以设置为 NORMAL 或 FAST。当 NPMSPEED=FAST 时，我们称之为快速通道 (FastPath Channel)。

在快速通道上的非持久消息 (Non-Persistent) 的发送是不受批量控制的，它本身也不会引发确认信息包，即没有同步过程，但它们在批量 (Batch) 计数的时候却是计算其中的。

4.3.1.3 心跳 (HeartBeat)

由于在 WebSphere MQ 中通道是单向传递数据的，消息永远从发送端到接收端。发送端能够反向接到的是一些控制信息。接收端打开通道和相应的队列可能会占用一定的资源，如果长时间没有收到任何数据，则有效的办法是让接收端收到一个信号，表示可以暂关闭通道，释放资源。这个信号就是发送端发出的心跳信号。

另外，发送端在长时间空闲的情况下，也可以选择保持连接或自动断连。WebSphere MQ 通道有相应的属性可以设置。表 4-2。

表 4-2 心跳相关的通道属性					
属性描述	属性名	缺省值	单位	有效范围	说明
Batch HeartBeat Interval	BATCHHB	0	毫秒	0 - 999,999	0 表示不使用
HeartBeat Interval	HBINT	300	秒	0 - 999,999	0 表示不使用，即不发送心跳信号
Disconnect Interval	DISCINT	6000	秒	0 - 999,999 0 - 9,999	除 z/OS with CICS 外的所有平台 z/OS with CICS
KeepAlive Interval	KAINT	AUTO	秒	0 - 99,999	0 表示不使用，仅对 z/OS 平台有效

Batch HeartBeat Interval 允许通道发送端对发送的消息批次进行提交之前检测接收端是否还活着。如果接收端不活，发送端回滚。如果在本批次内 (Batch Interval) 发送端收到过来自接收端的确认，则认为接收端还活着。否则，发送端主动送心跳信号 (HeartBeat) 给接收端并要求心跳回应。此属性只对 Sender、Server、Cluster-Sender 和 Cluster-Receiver 通道有效。

HeartBeat Interval 表示两次心跳之间的间隔时间。在发送端如果一定时间内没有新的消息可以传送，则发送端主动发送一个心跳信号，接收端在接收这个信号之后，可以关闭通道，而不必死等新的消息或等到 Disconnect Interval 时间超时。HeartBeat Interval 通常应该比 Disconnect Interval 小得多，接收端在收到心跳信号后可以释放为大消息准备的内存，也可以关闭那些仍打开着的队列。

Disconnect Interval 是 Sender、Cluster-Sender、Server 和 Cluster-Receiver 通道的属性。如果一个批次结束后 (Batch Interval 或 Batch Size 条件满足) 在一定的时间内没有新的消息可以传送，即两个批次之间的最长时间间隔，那么通道会自动关闭。这个时间就是 Disconnect Interval。在通道关闭的时候，发送端和接收端会交换控制数据指明关闭原因，以保证下一次的成功打开。Disconnect Interval 设置得太长，则对方的资源 (比如：打开的队列) 会长时间挂住。设置得太短，则通道多频繁地被打开关闭。通常可以设置另一个属性 HeartBeat Interval，这个属性值应该比 Disconnect Interval 小得多，通道发送端在一定间隔时间发送心跳数据，接收端收到 HeartBeat Interval 后就释放资源。

KeepAlive Interval (KAINT) 表示定时发送的保持连接信号的间隔时间。如果在 MQ 配置文件 (mq.ini) 中的 TCP stanza 段中设置有 KEEPALIVE=YES，尽管 KeepAlive Interval (KAINT) 的值为 0，TCP 链路上的 KeepAlive 信号仍然会自动发送。如果 KAJNT =

AUTO, 则 KAJNT 的值与 HBINT 有关:

- 如果通道连接时约定的 HBINT > 0, 则 KAJNT = HBINT + 60 秒
- 如果通道连接时约定的 HBINT = 0, 则 KAJNT = 0

4.3.1.4 连接重试 (Connection Retry)

通道 (Channel) 在暂时无法接通时会试图重试, 重试方式分为短等待和长等待两种方式, 每一种方式都会以一定的时间间隔重试多数。属性如表 4-3, 指明了发送方与接收方试图建立连接 (Session) 的重试次数与间隔。

表 4-3 连接重试相关的通道属性

属性描述	属性名	缺省值	单位	有效范围	说明
Short Retry Count	SHORTRTY	10	次	0 - 999,999,999	短等待次数
Short Retry Interval	SHORTTMR	60	秒	0 - 999,999,999	短等待间隔
Long Retry Count	LONGRTY	999,999,999	次	0 - 999,999,999	长等待次数
Long Retry Interval	LONGTMR	1200	秒	0 - 999,999,999	长等待间隔

通道首先用短等待方式进行连接重试, 每次等待 SHORTTMR 时长, 共 SHORTRTY 次。在短等待皆告失败后, 用长等待方式进行连接重试, 每次等待 LONGTMR 时长, 共 LONGRTY 次。所以, 重试 (Retry) 的总时长为: (SHORTRTY * SHORTTMR) + (LONGRTY * LONGTMR)。在重试期间, 通道处于重试 (Retrying) 状态, 在所有的重试工作皆告失败后, 通道处于关闭 (Closed) 状态。

4.3.2 通道协议

4.3.2.1 MCA

通道协议决定了消息通道代理 (Message Channel Agent -- MCA) 的工作方式。我们可以把 MCA 简单地理解为通信程序, 它可以以独立的进程方式运行, 也可以以线程的方式嵌入其它进程中运行。与 MCA 相关的通道属性如表 4-4。

表 4-4 MCA 相关的通道属性

属性描述	属性名	缺省值	可取值	说明
MCA Name	MCANAME	空	字串	保留
MCA Type	MCATYPE	PROCESS	PROCESS THREAD	
MCA User	MCAUSER	空	字串	

MCANAME 表示消息通道代理名, 是一个保留属性, 在实际环境中并不起作用。

MCATYPE 表示通道代理类型, 可以取值 PROCESS 或 THREAD, 分别表示发送端通信程序 MCA 以独立的 RUNMQCHL 进程方式执行还是在 RUNMQCHI 中以线程方式执行。对于接收端的通道代理类型是由启动监听的方式决定的, 即是用 inetd 还是 RUNMQLSR。

MCAUSER 表示 MCA 访问资源时的用户名,如果为空,则使用缺省用户。在 Windows 中,可以用 user@domain 的方式指定域用户。

通道上的通信协议可以是 TCP/IP、SNA/LU62、DCENET、NETBIOS、SPX 和 UDP 等等。以下选取使用得比较普遍的 TCP/IP 和 SNA/LU62 两种通信协议进行说明。

4.3.2.2 TCP/IP

通道定义中与 TCP/IP 相关的属性有三个 TRPTYPE、CONNAME 和 LOCLADDR (表 4-5)。

表 4-5 TCP/IP 相关的通道属性		
属性描述	属性名	说明
Transport Type	TRPTYPE	设置为 TCP
Local Address	LOCLADDR	本地的 IP+Port
Connection Name	CONNAME	对方的 IP+Port

TRPTYPE 表示通信协议,设置为 TCP。LOCLADDR 表示本地通信参数,格式为 LOCLADDR([ip-addr]([low-port[,high-port]])),长度为 48 字节。其中 ip-addr 是本地的 IP 地址,low-port 和 high-port 划定了一段 port 的有效范围。在建立连接时,WebSphere MQ 会根据实际情况选择一个合适的 Port。这种方式对于有防火墙的网络特别合适。

CONNAME 表示对方的通信参数,格式与 LOCLADDR 相同,常见的设置为 CONNAME(IP[(Port)]),其中 IP 地址也可以由机器名代替。该属性指定远端的 IP 地址和端口。如果没有指定端口,则根据远端的同名通道定义来决定,缺省端口值为 1414。

4.3.2.3 SNA/LU62

通道定义中与 SNA 和 LU62 相关的属性有:USERID、PASSWORD、TRPTYPE、MODENAME、TPNAME、CONNAME。(表 4-6)

表 4-6 SNA/LU62 相关的通道属性		
属性描述	属性名	说明
Transport Type	TRPTYPE	设置为 LU62
User ID	USERID	MCA 使用的启动 SNA Session 时的用户名
Password	PASSWORD	MCA 使用的启动 SNA Session 时的口令
LU62 Mode Name	MODENAME	LU62 连接时的附加信息
LU62 Transaction Program Name	TPNAME	LU62 连接时调用的交易名
Connection Name	CONNAME	LU62 连接对象名

对于 SNA,通信协议属性 TRPTYPE 应该设置为 LU62。由 MODENAME 指明 LU62 连接时的附加信息。如果用作提供 SNA 连接侧面信息,则 MODENAME 应该在 CPI-C 或 APPC 环境中定义,该属性值为空。TPNAME 指定远端的 MCA 交易名。CONNAME 指定远端连接的对象名,如果 TPNAME 和 MODENAME 不为空,则该属性指定 LU 全名。否则,如果 TPNAME 和 MODENAME 为空,则该属性为 CPI-C 对象名。

另外，SNA 协议在连接时需要提供连接的用户名和口令，分别由 USERID 和这 PASSWORD 来设定。

4.4通道的状态

通道状态分成三类状态属性：SAVED、CURRENT 和 SHORT。其中，SHORT 只适用于 z/OS 平台，这里不再讨论。SAVED 是 CURRENT 的子集，这些属性在建立连接时设置，此后不变，一直到重建连接时被重新设置。CURRENT 中的状态属性则在连接使用的过程中时时刻刻发生着变化，反应当前的通道状态。通常来说，我们只对 CURRENT 感兴趣。CURRENT 又分两部分：公共状态 (Common Status) 和当前状态 (Current-Only Status)。

运行中的通道状态可以用 MQSC 命令 `DISPLAY CHSTATUS () ALL` 来显示，也可以缩写为 `DIS CHS () ALL`。

下面让我们先来了解一下通道的通常状态和当前状态分别包含哪些属性。

4.4.1 公共状态 (Common Status)

公共状态中包含了通道中变化最快的一些状态属性，通常以消息或批次为单位实时更新，能反应通道目前的工作状态 (表 4-7)。我们可以通过观察公共状态得知通道当前的批次号、消息的序列号、传递的消息数等等。

表 4-7 通道 Common 状态属性

属性	说明
CURLUWID	当前 Batch 的 LUW ID 如果是发送端处于 In-Doubt 状态，则它是 In-Doubt Batch 的 LUW ID
CURMSG	对于发送端，它是当前 Batch 中已发送的消息数，每发送一条消息自动加一。如果是发送端处于 In-Doubt 状态，则它是 In-Doubt Batch 中已发送的消息数 对于接收端，它是当前 Batch 中已接收的消息数，每接收一条消息自动加一 无论是发送端或是接收端，Batch 提交时自动归零
CURSEQNO	对于发送端，它是上一条已发送消息的序列号，每发送一条消息自动加一。如果是发送端处于 In-Doubt 状态，则它是 In-Doubt Batch 中上一条已发送消息的序列号 对于接收端，它是上一条已接收消息的序列号
INDOUBT	指定通道当前是否处于 In-Doubt 状态。 对于发送端，只有在 MCA 已成功发送消息且等待应答 (Acknowledgment) 的时候，为 YES。其它时候皆为 NO 对于接收端，永远是 NO
LSTLUWID	上一个提交 Batch 的 LUW ID
LSTSEQNO	上一个提交 Batch 中最后一条消息的序列号。由于 NPMSPEED (FAST) 通道对于非持久性消息是不需要提交的，所以这种情况不会使 LSTSEQNO 自动累计
STATUS	通道的当前状态，可能是： <ul style="list-style-type: none">● STARTING

- BINDING
- INITIALIZING
- RUNNING
- STOPPING
- RETRYING
- PAUSED
- STOPPED
- REQUESTING

注意：对于 Inactive Channel, CURMSGs、CURSEQNO、CURLUWID 只有在通道处于不确定 (In-Doubt) 状态时值才有意义。

4.4.2 当前状态 (Current-Only Status)

当前状态包含了通道本次连接时约定的通信参数及连接以来的一些统计信息。(表 4-8)。观察当前状态可以得知通道的数据流量，已经完成传递的批次数、消息数、字节数等等。

表 4-8 通道 Current-Only 状态属性

属性	说明
BATCHES	在这个 Session 中 (通道启动以来) 完成的 Batch 数量
BATCHSZ	在这个 Session 中 (通道启动时商定的) Batch 大小
BUFSRCVD	在这个 Session 中 (通道启动以来) 收到的消息包数量, 包括控制信息
BUFSSENT	在这个 Session 中 (通道启动以来) 发出的消息包数量, 包括控制信息
BYTSRCVD	在这个 Session 中 (通道启动以来) 收到的字节总量, 包括控制信息
BYTSENT	在这个 Session 中 (通道启动以来) 发出的字节总量, 包括控制信息
CHSTADA	通道启动日期 (yyyy-mm-dd).
CHSTATI	通道启动时间 (hh.mm.ss).
HBINT	在这个 Session 中 (通道启动时商定的) Heartbeat Interval 大小
JOBNAME	当前通道进程的作业名, 通常是十六进制. <ul style="list-style-type: none"> ● Compaq OpenVMS PID (HEX) ● OS/2, OS/400, UNIX, Windows PID + TID (HEX) ● NSK CPU ID + PID (HEX)
LOCLADDR	本地地址, IP 和 Port
LONGRTS	Long Retry 还剩下的次数
SHORTRTS	Short Retry 还剩下的次数
LSTMSGDA	上一条消息发送, 或者上一个 MQI call 处理的日期
LSTMSGTI	上一条消息发送, 或者上一个 MQI call 处理的时间 <ul style="list-style-type: none"> ● 对于 Sender 或 Server, 这是上一条消息 (如果自动分裂, 指的是最后一部分) 发送的时间 ● 对于 Requester 或 Receiver, 这是上一条消息完整放入目标队列的时间. ● 对于 Server-Connection Channel, 这是上一个 MQI call 处理的时间
MAXMSGL	在这个 Session 中 (通道启动时商定的) 最大消息长度 (z/OS only)
MCASTAT	MCA 当前的状态。可以是“running”或“not running”。对于一个通道, 它的状态 (STATUS 域) 是 STOPPED, 然而 MCA 程序却可能依然活着。
MSGS	在这个 Session 中 (通道启动以来) 发送或接收的消息数量。对于 Server-Connection

	Channel, 这是 MQI call 的数量
NPMSPEED	在这个 Session 中 (通道启动时商定的) NPMSPEED 方式
RQMNAME	远程队列管理器名
SSLPEER	通道另一端的 Peer Queue Manager 或 Client 的 Distinguished Name。最长 256 字节
STOPREQ	是否有突出的用户停止请求。值可以是 YES or NO.

4.4.3 通道状态分析

由于通道状态属性实时地反映了通道的工作状态,在实际使用中经常被用来检验消息是否完整地送达、通道是否畅通及帮助定位并诊断问题。以下我们用两个队列管理器 QM1 和 QM2 之间的一条单向通道上的通道状态为例,分析通道流量与状态的关系。为了方便比较,我们把这一过程中通道两端的属性变化记录在表 4-9 中。

表 4-9 流量与通道的状态变化

QM1	QM2
1. 开始, 用 Start Channel 命令建立连接	
CURSEQNO(0)	CURSEQNO(0)
CURLUWID(2D92B43E10000301)	CURLUWID(0000000000000000)
MSGS(0)	MSGS(0)
BATCHES(0)	BATCHES(0)
BYTSSSENT(132)	BYTSSSENT(132)
BYTSRCVD(132)	BYTSRCVD(132)
BUFSSSENT(1)	BUFSSSENT(1)
BUFSRCVD(1)	BUFSRCVD(1)
JOBNAME(00000E9800000428)	JOBNAME(00000BDC00000FE4)
LSTMSGTI()	LSTMSGTI()
LSTMSGDA()	LSTMSGDA()
CHSTATI(15.55.38)	CHSTATI(15.55.38)
CHSTADA(2003-05-04)	CHSTADA(2003-05-04)
LSTSEQNO(0)	LSTSEQNO(0)
LSTLUWID(0000000000000000)	LSTLUWID(0000000000000000)
CURMSGGS(0)	CURMSGGS(0)
2. 接着, 用 amqsput 命令发送一条消息 (1 Bytes)	
CURSEQNO(1)	CURSEQNO(1)
CURLUWID(2D92B43E10000302)	CURLUWID(2D92B43E10000301)
MSGS(1)	MSGS(1)
BATCHES(1)	BATCHES(1)
BYTSSSENT(637)	BYTSSSENT(160)
BYTSRCVD(160)	BYTSRCVD(637)
BUFSSSENT(3)	BUFSSSENT(2)
BUFSRCVD(2)	BUFSRCVD(3)
JOBNAME(00000E9800000428)	JOBNAME(00000BDC00000FE4)
LSTMSGTI(15.55.56)	LSTMSGTI(15.55.56)
LSTMSGDA(2003-05-04)	LSTMSGDA(2003-05-04)

CHSTATI(15.55.38)	CHSTATI(15.55.38)
CHSTADA(2003-05-04)	CHSTADA(2003-05-04)
LSTSEQNO(0)	LSTSEQNO(0)
LSTLUWID(0000000000000000)	LSTLUWID(0000000000000000)
CURMSGs(0)	CURMSGs(0)
3. 然后，用 amqspout 命令再发送一条消息 (1 Bytes)	
CURSEQNO(2)	CURSEQNO(2)
CURLUWID(2D92B43E10000303)	CURLUWID(2D92B43E10000302)
MSGs(2)	MSGs(2)
BATCHES(2)	BATCHES(2)
BYTSENT(1142)	BYTSENT(188)
BYTSRCVD(188)	BYTSRCVD(1142)
BUFSSENT(5)	BUFSSENT(3)
BUFSRCVD(3)	BUFSRCVD(5)
JOBNAME(00000E9800000428)	JOBNAME(00000BDC00000FE4)
LSTMSGTI(15.56.03)	LSTMSGTI(15.56.03)
LSTMSGDA(2003-05-04)	LSTMSGDA(2003-05-04)
CHSTATI(15.55.38)	CHSTATI(15.55.38)
CHSTADA(2003-05-04)	CHSTADA(2003-05-04)
LSTSEQNO(0)	LSTSEQNO(0)
LSTLUWID(0000000000000000)	LSTLUWID(0000000000000000)
CURMSGs(0)	CURMSGs(0)

从以上通道的状态变化，我们可观察到，队列管理器 QM1 在两次消息之间共发送了 505 个字节 (BYTSENT 的差值)，是分两个数据包发送的 (BUFSSENT 的差值)。其中一个消息是 MQ 通道定时自动发送的同步信息包 (28 字节)，另一个消息包是数据包本身，它含有 MQTSH (48 字节) + MQXQH (428 字节) + Message Data (1 字节)，结构如图 4-1，共 477 字节。两者相加，恰好 505 字节。

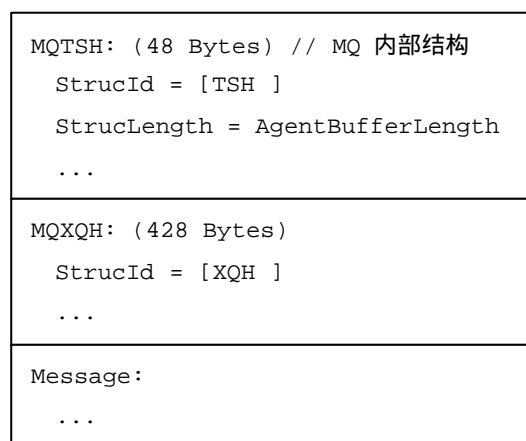


图 4-1 网络上传送的消息结构

QM2 是消息接收方，但从通道的状态变化中观察到它也有数据包发出，每次 28 字节，实际上，这就是同步信息包，用于对批次消息的确认。

另外，对于 Windows 和 UNIX，JOBNAME 是由十六进制的 PID (Process ID) + TID (Thread ID) 合成的。从上面的数据中可以观察到：(Windows)

- QM1 发送端
JOBNAME(00000E9800000428) PID = 3736，TID = 1064 Process = runmqchl
- QM2 接收端
JOBNAME(00000BDC00000FE4) PID = 3036，TID = 4068 Process = amqrmppa

4.5 互连配置举例

在实际工作中，我们往往需要构建复杂的互连结构，如果将其分解成简单的基本结构后，会发现这种基本结构也是有限的。下面我们举例介绍一些常用常用的基本结构。

4.5.1 单向传送

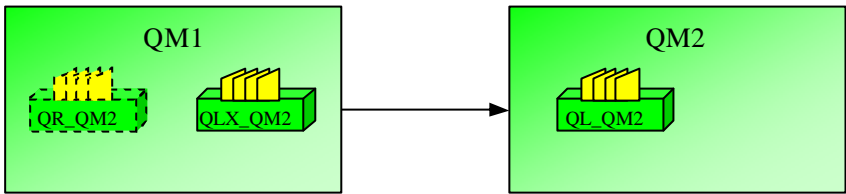


图 4-2 单向传递

QM1 要向 QM2 发送消息，双方采用 Sender/Receiver 方式。其中，QM1 上要定义远程队列 QR_QM2 和传输队列 QLX_QM2，QM2 上要定义本地队列 QL_QM2。图 4-2。

QM1 上的传输队列可以是对方队列管理器的名字 QM2，在远程队列 QR_QM2 的定义中如果不指定 XMITQ 参数，则系统会自动用 RQMNAME 参数去寻找同名传输队列，即 QM2。如果再找不到，会试图根据 QM1 的缺省传输队列参数 DEFXMITQ 再次寻找传输队列。所以，在 WebSphere MQ 设置时，通常建议设置队列管理器的缺省传输队列 DEFXMITQ，同时建议传输队列与对方队列管理器同名。

QM1:

DEFINE	QREMOTE	(QR_QM2)	+
	RNAME	(QL_QM2)	+
	RQMNAME	(QM2)	+
	XMITQ	(QLX_QM2)	+
	REPLACE		
DEFINE	QLOCAL	(QLX_QM2)	+
	USAGE	(XMITQ)	+
	REPLACE		
DEFINE	CHANNEL	(C_QM1.QM2)	+
	CHLTYPE	(SDR)	+
	TRPTYPE	(TCP)	+
	CONNNAME	('127.0.0.1 (1416)')	+

	XMITQ	(QLX_QM2)	+
	REPLACE		
QM2:			

DEFINE	QLOCAL	(QL_QM2)	+
	REPLACE		
DEFINE	CHANNEL	(C_QM1.QM2)	+
	CHLTYPE	(RCVR)	+
	TRPTYPE	(TCP)	+
	REPLACE		

4.5.2 双向传送

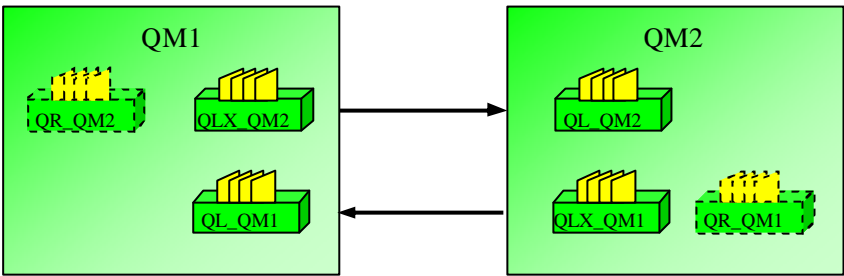


图 4-3 双向传递

QM1 和 QM2 要互相发送消息，双方采用 Sender/Receiver 方式。与上例类似，只是需要反向定义一套配置。图 4-3。

QM1:			

DEFINE	CHANNEL	(C_QM1.QM2)	+
	CHLTYPE	(SDR)	+
	TRPTYPE	(TCP)	+
	CONNAME	('127.0.0.1 (1415)')	+
	XMITQ	(QM2)	+
	REPLACE		
DEFINE	CHANNEL	(C_QM2.QM1)	+
	CHLTYPE	(RCVR)	+
	TRPTYPE	(TCP)	+
	REPLACE		
DEFINE	QLOCAL	(QM2)	+
	USAGE	(XMITQ)	+
	REPLACE		
QM2:			

DEFINE	CHANNEL	(C_QM1.QM2)	+

	CHLTYPE	(RCVR)	+
	TRPTYPE	(TCP)	+
	REPLACE		
DEFINE	CHANNEL	(C_QM2.QM1)	+
	CHLTYPE	(SDR)	+
	TRPTYPE	(TCP)	+
	CONNNAME	('127.0.0.1 (1416)')	+
	XMITQ	(QM1)	+
	REPLACE		
DEFINE	QLOCAL	(QM1)	+
	USAGE	(XMITQ)	+
	REPLACE		

4.5.3 队列与队列管理器别名

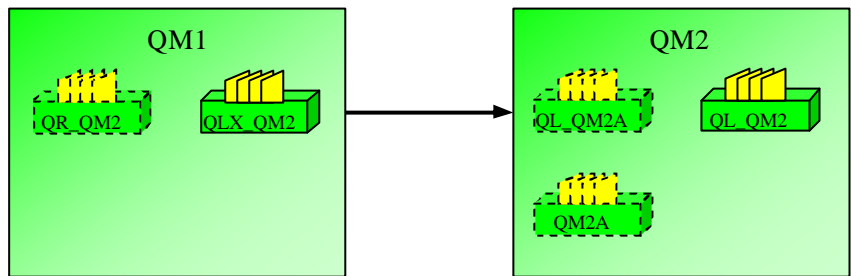


图 4-4 队列与队列管理器别名

QM1 要向 QM2 发送消息，双方采用 Sender/Receiver 方式。其中，QM1 上定义了远程队列 QR_QM2 指向 QM2A 队列管理器上的队列 QL_QM2A ,远程队列指定 QLX_QM2 传输队列。图 4-4。

放入远程队列 QR_QM2 的消息，由于远程队列本身指定了传输路由 (QLX_QM2)，所以消息自动地转送到 QM2 上。到了 QM2 后，消息的目标队列管理器为 QM2A，目标队列为 QL_QM2A，由于在 QM2 上定义了队列管理器别名 (QM2A → QM2)，也定义了别名队列 (QL_QM2A → QL_QM2)。最终，消息放入正确的目标队列。如果在 QM2 上找不到目标队列，消息会最试图进一步路由。

队列管理器别名本质上是 RQNAME 为空的远程队列定义。

QM1:

DEFINE	QREMOTE	(QR_QM2)	+
	RNAME	(QL_QM2A)	+
	RQMNAME	(QM2A)	+
	XMITQ	(QX_QM2)	+
	REPLACE		
DEFINE	QLOCAL	(QX_QM2)	+

	USAGE	(XMITQ)	+
	REPLACE		
DEFINE	CHANNEL	(C_QM1.QM2)	+
	CHLTYPE	(SDR)	+
	TRPTYPE	(TCP)	+
	CONNNAME	('127.0.0.1 (1415)')	+
	XMITQ	(QX_QM2)	+
	REPLACE		

QM2:

DEFINE	QLOCAL	(QL_QM2)	+
	REPLACE		
DEFINE	QALIAS	(QL_QM2A)	+
	TARGQ	(QL_QM2)	+
	REPLACE		
DEFINE	QREMOTE	(QM2A)	+
	RQMNAME	(QM2)	+
	REPLACE		
DEFINE	CHANNEL	(C_QM1.QM2)	+
	CHLTYPE	(RCVR)	+
	TRPTYPE	(TCP)	+
	REPLACE		

4.5.4 三级跳 (Multi-hopping)

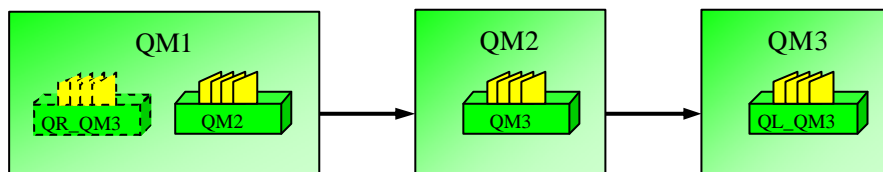


图 4-5 三级跳

QM1 要发送消息去 QM3，消息在 QM2 被自动转发，而不被应用消息处理，即通常所说的消息在 QM2 不落地。图 4-5。

QM1 上的远程队列 QR_QM3 定义了目标队列管理器为 QM3，目标队列为 QL_QM3。消息通过指定的传输队列 QM2 路由到 QM2 上后，试图寻找 QM3，结果找到名为 QM3 的传输队列，消息进一步路由到队列管理器 QM3 上，最终放入正确的目标队列。

如果 QM2 上的传输队列名与消息的目标队列管理器名不同，即不为 QM3。则可以通过设置 QM2 上的 DEFXMITQ 属性达到同样的效果。

QM1:

DEFINE	QREMOTE	(QR_QM3)	+
	RNAME	(QL_QM3)	+
	RQMNAME	(QM3)	+
	XMITQ	(QM2)	+
	REPLACE		
DEFINE	QLOCAL	(QM2)	+
	USAGE	(XMITQ)	+
	REPLACE		
DEFINE	CHANNEL	(C_QM1.QM2)	+
	CHLTYPE	(SDR)	+
	TRPTYPE	(TCP)	+
	CONNAME	('127.0.0.1 (1415)')	+
	XMITQ	(QM2)	+
	REPLACE		

QM2:

DEFINE	QLOCAL	(QM3)	+
	USAGE	(XMITQ)	+
	REPLACE		
DEFINE	CHANNEL	(C_QM1.QM2)	+
	CHLTYPE	(RCVR)	+
	TRPTYPE	(TCP)	+
	REPLACE		
DEFINE	CHANNEL	(C_QM2.QM3)	+
	CHLTYPE	(SDR)	+
	TRPTYPE	(TCP)	+
	CONNAME	('127.0.0.1 (1416)')	+
	XMITQ	(QM3)	+
	REPLACE		

QM3:

DEFINE	QLOCAL	(QL_QM3)	+
	REPLACE		
DEFINE	CHANNEL	(C_QM2.QM3)	+
	CHLTYPE	(RCVR)	+
	TRPTYPE	(TCP)	+
	REPLACE		

4.5.5 四级跳 (Multi-hopping)

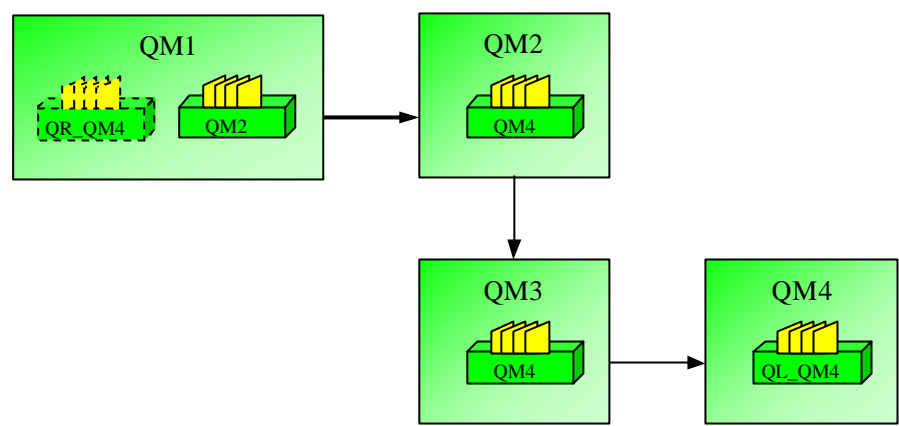


图 4-6 四级跳

QM1 要发送消息去 QM4 ,消息在 QM2 和 QM3 上被自动转发。本例可以看作上例的扩展。图 4-6。消息在传递路径上不落地。

QM1 上的远程队列 QR_QM4 定义了目标队列管理器为 QM4 ,目标队列为 QL_QM4。消息放入 QR_QM4 后,经过指定的传输队列 QM2 路由到 QM2。在 QM2 上消息发现这不是目的地 QM4 ,于是就找到名为 QM4 的传输队列继续漫游。类似地,到了 QM3 后,消息又发现 QM3 仍然不是目的地,再次通过名为 QM4 的传输队列漫游。消息最终到达 QM4 ,且成功放入目标队列 QL_QM4 中。

如果 QM2 和 QM3 上的传输队列名与消息的目标队列管理器名不同,即不为 QM4。则可以通过设置 QM2 和 QM3 上的 DEFXMLTQ 属性达到同样的效果。

```
QM1:
----
DEFINE      QREMOTE      (QR_QM4)                +
            RNAME        (QL_QM4)                +
            RQMNAME      (QM4)                   +
            XMITQ        (QM2)                   +
            REPLACE
DEFINE      QLOCAL      (QM2)                    +
            USAGE        (XMITQ)                  +
            REPLACE
DEFINE      CHANNEL      (C_QM1.QM2)              +
            CHLTYPE      (SDR)                    +
            TRPTYPE      (TCP)                    +
            CONNAME      ('127.0.0.1 (1415)')      +
            XMITQ        (QM2)                    +
            REPLACE
```

QM2:

```

----
DEFINE      QLOCAL      (QM4)                +
            USAGE      (XMITQ)                +
            REPLACE
DEFINE      CHANNEL     (C_QM1.QM2)           +
            CHLTYPE     (RCVR)                +
            TRPTYPE     (TCP)                 +
            REPLACE
DEFINE      CHANNEL     (C_QM2.QM3)           +
            CHLTYPE     (SDR)                 +
            TRPTYPE     (TCP)                 +
            CONNAME     ('127.0.0.1 (1416)')   +
            XMITQ       (QM4)                 +
            REPLACE

QM3:
----
DEFINE      QLOCAL      (QM4)                +
            USAGE      (XMITQ)                +
            REPLACE
DEFINE      CHANNEL     (C_QM2.QM3)           +
            CHLTYPE     (RCVR)                +
            TRPTYPE     (TCP)                 +
            REPLACE
DEFINE      CHANNEL     (C_QM3.QM4)           +
            CHLTYPE     (SDR)                 +
            TRPTYPE     (TCP)                 +
            CONNAME     ('127.0.0.1 (1417)')   +
            XMITQ       (QM4)                 +
            REPLACE

QM4:
----
DEFINE      QLOCAL      (QL_QM4)              +
            REPLACE
DEFINE      CHANNEL     (C_QM3.QM4)           +
            CHLTYPE     (RCVR)                +
            TRPTYPE     (TCP)                 +
            REPLACE

```

第 5 章 应用设计

我们在前面的介绍中已经了解了 WebSphere MQ 的概念、原理、安装、管理、配置等等内容，对 MQ 已经有有了一个初步的了解，可以说在操作上已经没有问题了，但如果要基于 MQ 设计一个应用系统，则可能仍然觉得无所适从。下面我们要进一步介绍 MQ 应用的设计、实现的功能及相关技巧。

5.1 架构设计

5.1.1 两点间通信

WebSphere MQ 的基本功能是用来连接异构平台之间的应用，通过 MQ 的互连，使两个应用系统实现互连通信。图 5-1。所以，对于已有的应用系统，如果没有通信接口则通常需要进行改造，嵌入 MQI 调用使消息进得来、出得去。

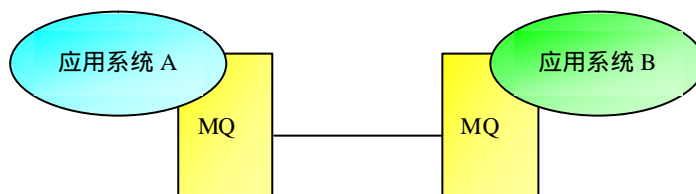


图 5-1 应用系统通过 MQ 实现互连

MQ 的通信方式是异步非实时的，也就是说应用系统 A 调用发送消息的 API，如果返回正常表示发送消息已经被 MQ 接受，但并不表示消息已经传达应用系统 B。首先，在传输的路途中如果发生各种异常情况，应用系统 B 有可能收不到该消息。其次，由于网络或其它的原因，应用系统 B 有可能不能马上收到该消息。在这种情况下，应用系统 A 无法知道消息是否已经成功地到达应用系统 B。所以，在需要有反馈的应用系统中，通常会设计反向应答机制。

应用系统 A 发出请求消息，然后等待对该请求消息的应答反馈，即请求消息。通常情况下，请求消息和应答消息应该放在不同的队列中，以免混淆。每一条 MQ 消息的头结构 (MQMD) 中包含了 MsgId 和 CorrelId 两个域，分别用于标识消息本身和相关消息。对于应答消息，它有不同于请求消息的 MsgId，但它的 CorrelId 应该等于请求消息的 MsgId，表示针对该请求消息的应答。在应用系统 A 等待应答消息的时候，可以用请求消息的 MsgId 作为匹配条件来明确地等待针对该请求消息的应答。

应用系统 B 在收到请求消息后，根据消息内容做出相应的动作，同时在构建应答消息的时候，应该将请求消息的 MsgId 填入应答消息的 CorrelId 域中。

当然，应用系统 A 也可以利用 MQ 消息的一些属性来设定一些特殊功能。比如有些消息是有时效性的，超过时间后即使送达对方也变得没有意义了，这时可以设置消息超时 (MQMD 的 Expiry 域)。有些消息的寄送是需要回执的，在消息送达目标队列或消息被对方取出后由 MQ 自动返回一条报告消息。具体细节参见“报告”章节。

5.1.2 多点间通信

当应用场景中涉及需要互连的系统较多时，如果简单地将所有需要互连的系统之间用 MQ 直接连接起来，就会形成网状结构，如图 5-2 左。网状结构的好处在于从单个应用考虑逻辑简单，直截了当。且为每两个结点之间配置了独立的连接，是通信速度最快最直接的一种结构。由于各条连接的忙闲程度不同，需要传送的数据量不同，数据的重要性也不同，网状结构能最大程度地保证各种通信各行其道、互不干扰。然而，随着结点的增多，网状结构会不可避免地出现接口众多而难以管理的局面。太多的连接会给配置、管理、编程都带来不小的麻烦。

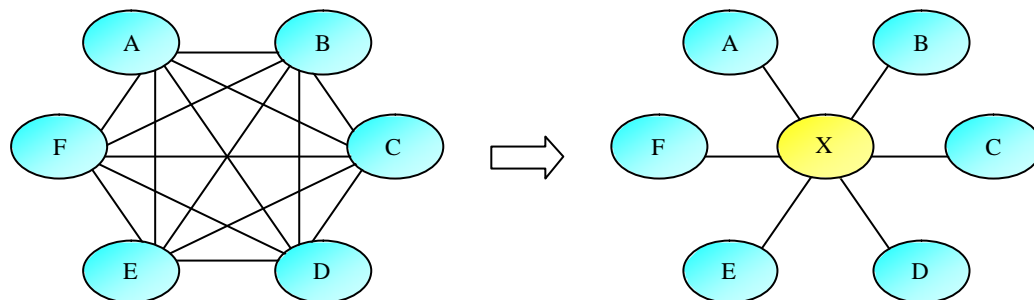


图 5-2 从网状结构到星状结构

人们很自然地会想到用星状结构来解决这一问题，图 5-2 右。星状结构中有一个结点处于中心位置，扮演着数据交换中心的角色。所有的其它结点都与之相连，消息通过它转发到目标队列中。星状结构中又称中心辐射结构 (Hub and Spoke)，在众多系统互连时可以有效地降低拓扑结构的复杂度。星状结构有明显的优点，在中心简单地配置后，消息可以通过多级跳自动路由到目标结点中，我们称这种方式为“消息不落地”。如果中心有存储转发的程序，即采用“消息落地”的方式，则中心可以利用自己的有利位置对所有的消息进行统一管理，比如监控干预流量，检查消息内容，修改消息格式，以及由此衍生出来的各种功能。事实上，已有一些产品提供类似的功能，比如 WebSphere Business Integration Message Broker 等。星状结构也有明显的缺点，即中心结点很容易成为消息通信和处理的瓶颈，且一旦中心结点成为出现故障，整个结构就无法互连通信了。为了避免中心结点成为单点故障，通常需要一些补偿措施，比如采用双机热备技术。

实现的应用场景中，很难简单地说明哪一种结构更好，需要视具体情况而定。

5.1.3 同步和异步

经常有人问，既然 MQ 处理消息在本质上是异步的，那么基于 MQ 设计的应用如何能做到同步呢？这里先让我们来澄清一下应用的同步和异步的概念。传统的应用程序调用后台服务通常采用的是请求/应答模式，应用程序在发出请求后希望在合理的时间内收到相应的应答，从应答中可以方便地判断出业务是成功不是失败，如果超过时间未收到应答则认为服务超时，这时应用程序需要有能力和等待中返回。如图 5-3 左。应用程序通常只关心请求与应答，而不会关心后台是如何将请求变成应答的。对于应用程序而言，请求的内容、应答的内容、期望的时间三者是这种调用模式的三个要素，也就是调用接口。如果这种调用过程可以一次完成（最好在一个 API 中完成），则我们称之为同步调用。如果需要一次以上完成，则称为异步调用。

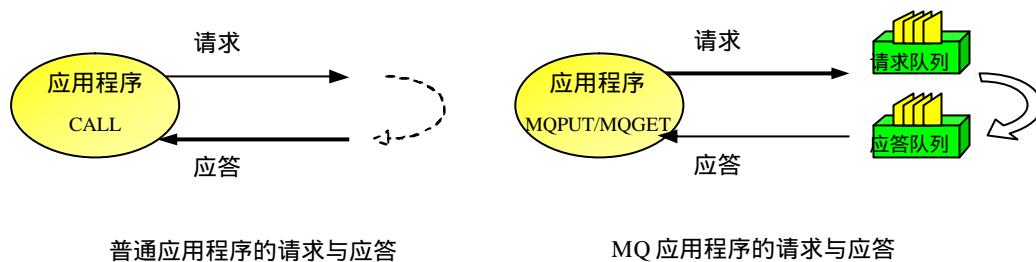


图 5-3 请求与应答

在 WebSphere MQ 中,应用程序将请求消息通常被放入请求队列,接着在应答队列中等待读取相应的应答消息,如果在期望的时间中没有读到应答消息,则认为服务超时,应用程序会从读等待中返回,如图 5-3 右。这一过程是分多个动作完成的,所以从原子操作上看是一个异步调用的过程,但是如果我们把这一过程封装在一个调用中,则又可以看作是同步调用。所以,异步和同步都不是绝对的,在 MQ 设计中是可以转换的。

同步和异步的另一个区别在于同步调用超时后,会有某种机制保证服务无法正常完成。而对于异步调用超时后,由于没有这样的机制,往往会出现滞后响应 (Later Response)。在 MQ 应用中,如果需要达到同步调用同样的效果,则需要在设计上动一些脑筋。首先,请求消息放入请求队列后有可能迟迟得不到服务而躺在队列中睡觉,这时我们可以为消息设计有效寿命,超过寿命后消息不再具有业务上的时效性,可以不处理。通常这时候,应用程序已经超时了。其次,如果处理中碰到某个环节耽搁了很长时间超过了消息的寿命或应用程序可以忍受的时间,在处理中应用可以发现这一问题并放弃进一步处理。最后,消息在传递的过程中可能会遇到一些麻烦,这时可以通过设定自动报告的方式使应用程序及时得知发生的异常情况。

5.1.4 Client/Server

WebSphere MQ 产品分成客户端和服务端两部分,对于客户端而言没有 MQ 对象的概念,它通过配置的 MQI 通道将操作命令送到服务器端执行,结果再原路返回到客户端,所以客户端无法独立于服务器端而自行工作。服务器之间的通道也就是队列管理器之间的通道,称为消息通道。客户端和服务端之间的通道,称为 MQI 通道。那么,在设计时什么时候该采用客户端,什么时候又该采用服务器呢?采用何种方式,完全取决于应用环境中能否发挥产品的特性,不可一概而论。由于客户端的限制,我们在设计时需要进行一些特殊的考虑。

首先,客户端永远只能作为消息发起的主动方而无法作为被动方。由于自身无法存放消息,所以待处理的消息只能暂存于服务器端。由客户端程序读服务器队列或设置客户端触发器来接收消息,但无论是哪一种方式客户端都无法独立工作。

其次,客户端只是将操作命令封装后送到服务器端执行,严格地说 MQI 通道传送的不是消息。客户端没有规范的通信进程 (MCA),也没有保证消息通信的 MQ 协议。所以,客户端与服务器端之间的 MQI 通道实际上是不能保证消息可靠性的。比如,客户端调用 MQGET 将队列中消息取出,命令在服务器端被成功执行,消息会随 API 的执行参数中返回。正当命令返回过程中,网络发生故障,这时客户端只取到消息的前半部分,返回出错。而在服务端该消息已经从队列中取出并删除了。也就是说,客户端再也无法取到消息的后半部分。

正是由于 MQ 客户端自身的这些限制，使客户端与服务器端之间的连接不适合应用于广域网 (WAN)。通常情况下，客户端与服务器端之间的连接应该在局域网内，利用局域网的可靠性和稳定性来弥补客户端的不足。一个典型的客户端与服务器端的应用场景图 5-3，两套应用系统在各自的局域网上运行，通过 MQ 服务器在广域网上互连。这时 MQ 服务器扮演前置机或通信机的角色，应用系统内的机器作为 MQ 客户端连接该通信机，通过操作 MQ 服务器与对方系统系统互连通信。

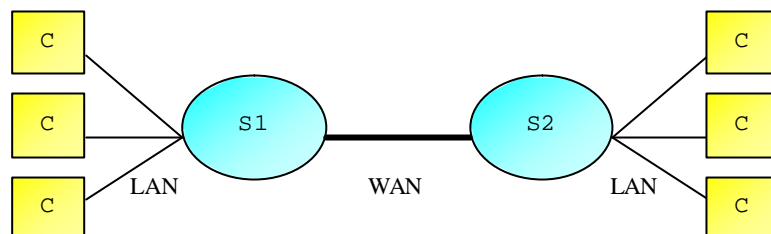


图 5-3 客户端与服务器端

但是，客户端也有它独特的功能，比如客户端程序可以配置连接多个队列管理器，在一个连接不通的情况下自动连接另一个，这使得客户端应用程序有一定程度上有容错功能。如果多个克隆的客户端应用程序读同一个服务器队列，则可以形成负载平衡的效果。另外，由于 Client/Server 不是对等连接，无需在服务端做一对一的配置，客户端的连接配置又十分方便，所以对于结构多变的通信环境中有着相当的用途。

5.1.5 Internet 通信

如果应用系统部署到互联网上，则首要考虑的是安全问题。外网接入层通常有防火墙，以阻隔来自外网的随意连接，有时还会有两道防火墙，并采用代理或隧道技术将内网的网络信息彻底对外屏蔽。在这种情况下，WebSphere MQ 队列管理器无法对外直连，应用结构的设计可能需要做这方面的考虑。通常，我们会用 SupportPac MS81 – WebSphere MQ Internet PassThru (MQIPT) 来解决这方面的问题。

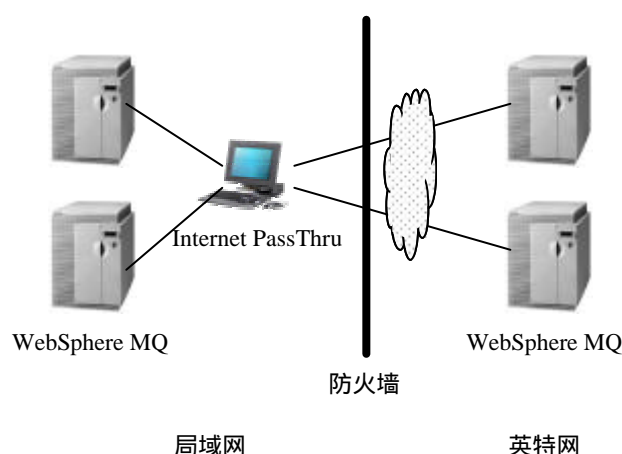


图 5-4 MQIPT 作为通道集中器实例

图 5-4 展示了使用 MQIPT 作为通道集中器的结构，外网的多台 WebSphere MQ 服务器需要与内网的 WebSphere MQ 服务器互连，这样需要在防火墙上做相应的设置，允许对相应 IP 地址和端口的连接通过，俗称在防火墙上钻洞。如果这样的连接需求很多，则需要钻很

多的洞，会引起管理上的不方便。如果采用 MQIPT，可以将所有的外来连接都集中于一台机器，通信数据由 MQIPT 转交 MQ 服务器，这样在防火墙的设置和管理上都带来方便。

现实中的应用系统往往用两道防火墙将内网和外网相隔，两道防火墙的中间部分称为非保护区(DMZ)，用于安全缓冲。如图 5-5。从外网进入的第一道防火墙往往只留很少的通道，允许一些常规的端口对外服务，比如 HTTP，其它连接一概被拒绝。第二道防火墙将安全的内网与不太安全的非保护区隔开，将内部的网络结构和信息进一步保护起来。如果我们将应用的连接和数据封装成为 HTTP 协议进入，这种技术称为 HTTP 隧道。来自 Internet 的 WebSphere MQ 连接可以通过 MQIPT 将消息包装后进入 HTTP 隧道，在通过隧道后再由 MQIPT 将 HTTP 协议外壳脱去，还消息的本来面目。另外，Internet 上的 SSL 加密认证也是常用的安全手段，它可以配置在两端的 MQIPT 上，使通信更安全可靠。

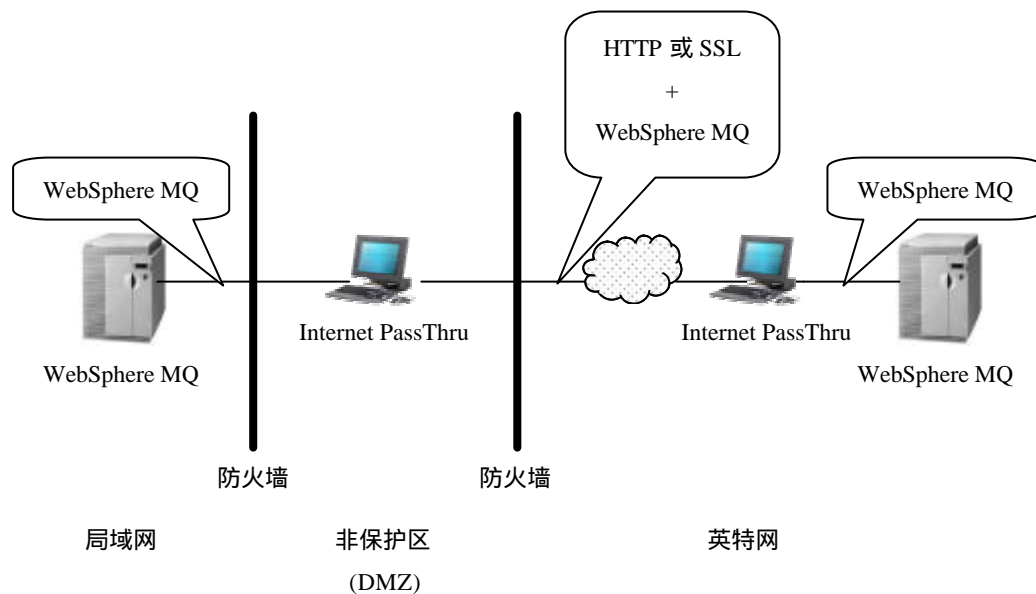


图 5-5 采用 HTTP 隧道的技术的两道防火墙实例

5.2 通信方式设计

5.2.1 进程间会话模式

进程间会话模式实际上是程序与程序之间通信时的一种消息通话机制，它是程序间数据沟通的联系模式。WebSphere MQ 提供了下列灵活的基本通信方式：

- 串行通信

程序 A 发出数据给程序 B，等到回应后再发出下一次数据。这有点类似于打电话的方式。

- 并行通信

程序 A 发出数据给程序 B，无需等待 B 的回应，A 继续处理自己的工作。程序 B 可以在任何方便的时候取到消息并做相应的处理。这就需要有消息队列作为中间人将 A 发出的消息保留直被 B 取到。这有点类似于电话留言的方式。并行通信中 A 可以连接不断地发出数据给 B，然后在合适的时候等待 B 统一的回应。

- 无连接通信

程序 A 与程序 B 之间没有私有的或公共的信道，它们仅靠队列机制提供的端对端的服务来通信，而这对于程序 A，B 来说都是完全透明的。

- 触发机制

程序 A 可以将触发消息发送到带有触发机制的队列上，由触发机制触发程序 B 启动执行。

- 并行处理

程序 A 可以将一个复杂任务分成多个子任务，通过发送消息的办法交给多个程序处理，再由程序 A 收集处理结果。由于多个程序间可以没有执行的先后次序，所以多个子任务可以是并行处理的。

- 客户机/服务器

客户程序 A 发送请求给服务程序 B，服务程序 B 处理完相应的工作后将应答自动封装成另一条消息返回给客户程序 A。

- 分布式处理

由以上所有通信方式组合而成的复杂方式。

在 MQ 的应用设计时，程序之间选择何种会话模式是最初要考虑的部分，也是设计的重点之一。程序间的消息流向是单向的还是双向的，请求和应答程序是串行通信的还是并行通信，是否采用触发机制等等，这都是需要在设计最初确定下来的。

5.2.2 系统间通信方式

WebSphere MQ 的基本功能是用来传递消息的，通过配置通道来实现系统之间的互连通信。通信方式分成点对点的方式和订阅/发布的方式，其中点对点的方式可以实现两个单点系统之间的直接互连，也可以实现多点系统链式的间接互连 (Multi-hopping)。点对点的另一种扩展方式为分发列表，类似于广播。订阅/发布是一种较高级的互连通信方式，发布者和订阅者首先约定一个消息主题 (Topic)，之后发布者在 MQ 网络上发布该主题的消息会经过 MQ 网络自动地送达订阅者。图 5-6

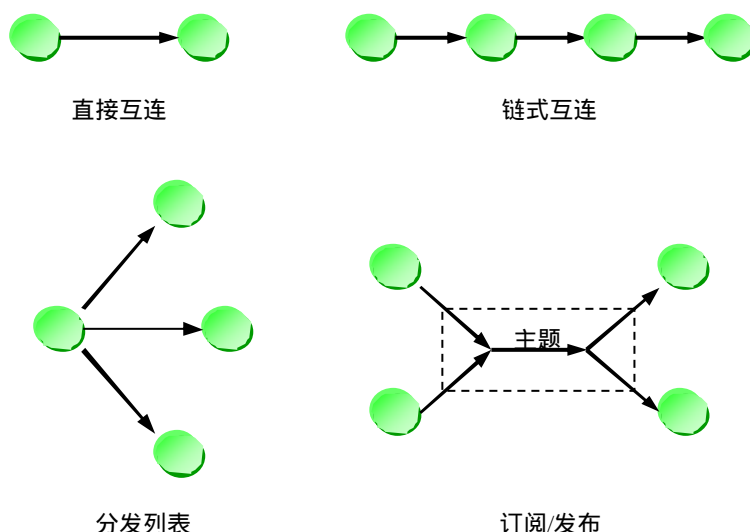


图 5-6 WebSphere MQ 互连通信的方式

点对点（也称为一对一）的通信方式是 WebSphere MQ 最拿手的，也是传统设计方法考

虑得最多的，它通过定义远程队列、传输队列和通道的方式将队列管理器联系起来。订阅/发布（也称为多对多）的通信方式需要有额外的部件支持，通常会采用 SupportPac MA0C 或者 Message Broker。应用设计采取哪一种通信方式取决于应用中各程序之间的关系，如果消息内容是针对某一个体，则采用点对点的方式比较妥当，如果消息内容是针对一个群体，则采用订阅/发布进行广播更合适一些。

5.3 并发设计

5.3.1 多读多写

WebSphere MQ 编程环境是支持多读多写的。无论是读或写，都需要应用程序要在打开对象的时候指定功能选项，比如 MQOO_INPUT 和 MQOO_OUTPUT，这里的 INPUT 和 OUTPUT 都是站在应用程序的立场而言的。

在应用设计时，如果哪一个环节需要增加处理能力，可以考虑多实例（Multi-Instance）并行工作的方式。由于队列等对象支持多读多写，所以多实例的工作效率可能是单实例的数倍。

5.3.2 共享与独占

对于读消息，WebSphere MQ 支持共享和独占两用方式。对于写消息，WebSphere MQ 只支持共享方式。在 MQOPEN 时的参数选项可取值：

- MQOO_INPUT_AS_Q_DEF 根据队列的 DEFSOPT 属性来决定
- MQOO_INPUT_SHARED 共享式读取消息
- MQOO_INPUT_EXCLUSIVE 独占式读取消息
- MQOO_OUTPUT 共享式发送消息

然而，消息所在的队列有自己的属性可以控制打开方式是共享还是独占。如表 5-1。

表 5-1 队列与共享相关的属性

队列属性	说明
SHARE 或 NOSHARE	是否允许多个应用程序并发地 MQGET 操作该队列
DEFSOPT ()	指定 INPUT 方式打开（即需要 MQGET）时的缺省共享模式
● EXCL	● 只能有一个应用程序以 INPUT 方式打开
● SHARED	● 可以有多个应用程序以 INPUT 方式打开

队列属性与打开方式 (MQOO_*) 必须配合起来使用。如果应用程序在 MQOPEN 时希望由队列属性来决定共享或独占打开方式，则用 MQOO_INPUT_AS_Q_DEF，打开方式会参考 DEFSOPT 属性。反之，应用程序 MQOPEN 显式地设置 MQOO_INPUT_EXCLUSIVE 或者 MQOO_INPUT_SHARED 即可。

5.3.3 对象绑定

WebSphere MQ 应用程序在打开对象的时候可以指定是否对象，一旦绑定后，操作对象就固定下来了。反之，如果不绑定对象则每次在发送的时候寻找目标对象，这在群集多实例队列环境中有用。在 MQOPEN 时的参数选项可取值：

- MQOO_BIND_ON_OPEN 打开时绑定
- MQOO_BIND_NOT_FIXED 不绑定
- MQOO_BIND_AS_Q_DEF 根据队列的 DEFBIND 属性来决定

而 DEFBIND 可取值 MQBND_BIND_ON_OPEN 和 MQBND_BIND_NOT_FIXED。它们之间的关系表 5-2。

表 5-2 MQOPEN 参数与队列 DEFBIND 属性共同决定队列绑定方式

MQOPEN 参数选项	队列 DEFBIND 属性			
	MQBND_BIND_ON_OPEN		MQBND_BIND_NOT_FIXED	
MQOO_BIND_ON_OPEN	BIND	绑定	BIND	绑定
MQOO_BIND_NOT_FIXED	NOT_FIXED	不绑定	NOT_FIXED	不绑定
MQOO_BIND_AS_Q_DEF	BIND	绑定	NOT_FIXED	不绑定

5.3.4 队列管理器关闭

由于队列管理器是支持并发操作的，所以一个应用程序在操作时无法预料其它应用程序的操作。比如，一个在队列上做永久等待读操作的应用程序可能会因为队列管理器的突然关闭而僵死，所以在操作时使用 FAIL_IF QUIESCING 是好习惯。队列管理器在关闭的时候，可以选择 endmqm -c 或 -w 选项，使队列管理器进入待关闭 (Quiescing) 状态。这时，队列管理器不再接受新的应用连接，而等待原来的应用运行完毕后彻底关闭队列管理器。在这时，原来的应用应该与之配合尽快退出。有了 FAIL_IF QUIESCING 选项，就可以在 API 调用中及时地发现并做出相应的动作，比如结束对消息的永久等待并回滚交易。

相关的操作选项有：

- MQOO_FAIL_IF QUIESCING
- MQPMO_FAIL_IF QUIESCING
- MQGMO_FAIL_IF QUIESCING

5.3.5 分发列表 (Distribution List)

WebSphere MQ 中一次 MQPUT 或 MQPUT1 可以同时多个队列进行操作，这需要将多个队列纳入分发列表中，以后对该分发列表的操作就如同对列表中所有队列的操作，这有点像群发邮件的功能。这些操作可以在交易环境中进行，使多个队列的消息分发同时提交或回滚。如果多个队列是远程队列且目标队列管理器相同，则在网络上只需要传送一次，在达到对方后由目标队列管理器进行二次分发。

并非所有的 WebSphere MQ 平台都支持分发列表，所以分发列表正常工作相关平台支持，如果需要前文中的二次分发功能，则目标队列管理器也要支持分发列表才可以。

5.4消息设计

5.4.1 消息大小 (Message Size)

WebSphere MQ 中消息可以分成物理消息和逻辑消息，其中逻辑消息是由多条物理消息组成的。物理消息缺省可以达到 4MB，这对大多数应用来说是够用了，所以通常情况下逻辑消息与物理消息是统一的。在某种特定的情况下，消息需要分组或分段，这时逻辑消息可能会含多个物理消息。物理消息是 MQ 对消息计数的单位，通常与 MQGET 或 MQPUT 相对应。物理消息的大小是有限的，受限于队列管理器、队列或通道对象的相关属性。表 6-1 记录了各对象与消息大小相关的属性。

表 6-1 WebSphere MQ 对象中与消息大小相关的属性

对象	属性	缺省值	范围	说明
队列管理器	MAXMSGL	4M	32K - 100M	
队列 (Local/Model)	MAXMSGL	4M	0 - QM.MAXMSGL	0 表示用 QM.MAXMSGL
通道	MAXMSGL	4M	0 - QM.MAXMSGL	0 表示用 QM.MAXMSGL

队列管理器的 MAXMSGL 表示队列管理器中可以容纳的最大消息长度，缺省值为 4,194,403，调整范围在 32,768 到 104,857,600 之间。队列管理器的 MAXMSGL 的值应该始终大于等于队列的 MAXMSGL 值，如果这个值调小了，应该检查队列管理器中所有的本地队列和模型队列的 MAXMSGL 属性。

队列的 MAXMSGL 仅对本地队列和模型队列有效，表示队列中可以容纳的最大消息长度，这个属性的调整范围在各个平台上的上限是不一样的。

- 对于 AIX、Compaq OpenVMS、HP-UX、Linux、OS/2 Warp、OS/400、Solaris 和 Windows 平台，队列的 MAXMSGL 值可以从 0 到队列管理器的 MAXMSGL (上限为 100 MB)。
- 如果 z/OS 平台，大小为 0 - 104,857,600 Bytes。
- 对于其它平台，大小为 0 - 4,194,304 Bytes。

对于传输队列，消息的最大长度也包括消息传输头的大小。如果这个值调小了，已经在队列中的超长消息不受影响。应用程序可以用这个值来决定需要申请的内存，所以调大该值有可能会影响应用程序的正常运行。

通道的 MAXMSGL，表示通道可以一次传送的最大消息长度。

- 对于 AIX、iSeries、HP-UX、Linux、Solaris 和 Windows，通道的 MAXMSGL 的大小应该大于等于 0，小于等于队列管理器的 MAXMSGL。
- 对于 z/OS 平台，大小为 0 - 104,857,600 Bytes。
- 对于其它平台，大小为 0 - 4,194,304 Bytes。

通道在建立的时候会有一个握手过程，双方会交换各自通道定义上的 MAXMSGL，最后协商出通道使用的最大消息长度，一般会取双方定义中较小的那一个。

如果通道的两端支持消息切分，则一旦实际传送的消息长度比通道的可以传送的最大消息长度 (MAXMSGL) 更大也不要紧，MQ 会自动将其切分后送出，在远端自动拼接起来。

只要小于队列管理器的 MAXMSGL 和队列的 MAXMSGL，使本地队列能够放得下这条消息即可。

如果通道的两端都不支持消息切分，且消息实际传送的消息长度比通道的 MAXMSGL 大，则出错。

所以，鉴于各平台对队列和通道上的最大消息长度定义不一，在设计的时候需要将这种差别考虑在内。比如，考虑 SCO OpenServer 与 AIX 之间的数据传输，消息的大小设计应该小于 4MB。

5.4.2 消息持久性 (Persistence)

消息可以分成持久性消息和非持久性消息，前者在被放入队列或从队列中取出时除了操作队列外还会记录日志，所以在队列管理器重启后消息会自动恢复。后者由于没有日志操作，消息会丢失。另外，持久性消息通常表示消息是重要的，不可丢失的，在传送发生困难时，MQ 会尽可能保护消息不丢失。非持久消息在无法送递时往往被丢弃。消息的持久性与存放消息的队列及传递消息的通道有一定的关系，表 5-2 列举了 MQ 对象中与消息持久性相关的属性。

表 6-2 WebSphere MQ 对象中与消息持久性相关的属性

对象	属性	缺省值	可取值	说明
消息 (MQMD)	DefPersistence	Q_DEF	<ul style="list-style-type: none">● MQPER_PERSISTENCE_AS_Q_DEF● MQPER_PERSISTENT● MQPER_NOT_PERSISTENT	
队列	DEFPSIST	NO	<ul style="list-style-type: none">● YES● NO	z/OS 中用 Y 或 N
通道	NPMSPEED	FAST	<ul style="list-style-type: none">● NORMAL● FAST	

消息头 MQMD 属性 DefPersistence 可以有三种取值，但实际上只有两种有效值，即持久或非持久。对于 MQPER_PERSISTENCE_AS_Q_DEF，消息在 MQPUT/MQPUT1 时会设置队列的缺省属性。持久的消息可以在队列管理器重启时恢复，非持久的消息则不可以。持久的消息写入或读出队列的同时会在 Log 中记录，所以性能上比非持久消息差不少。

队列属性 DEFPSIST 表示放入其中的消息缺省是持久的 (Persistence) 还是非持久的 (Non-Persistence)。持久消息不可以放入临时动态队列 (Temporary Dynamic Queue)。因为考虑到临时动态队列可以在 MQOPEN 时创建，在 MQCLOSE 时删除，在队列管理器重启后也会自动清除。这与持久消息不会丢失的性质不相容，所以 WebSphere MQ 禁止持久性消息放入临时动态队列，在程序运行时报错：MQRC_PERSISTENT_NOT_ALLOWED。

通道的属性 NPMSPEED (Nonpersistent Message Speed) 有两种取值 NORMAL 或 FAST。缺省设置为 NORMAL，但如果设置成 FAST，则非持久消息的传送可以不参加交易。交易中的非持久消息一旦到达传输队列会立即送出，这可以使非持久消息的传送速度大大加快。缺点是消息可能丢失，可能泄露到交易之外。丢失指消息传送失败或通道关闭后，消息不会被回滚到传输队列中。泄露指虽然消息在发送方在交易中进行操作，但因为消息是非持久消

息，一旦放入传输队列就立即送出到达对方，无法回滚。

几乎所有的设计人员都会认为系统中出现的每一条消息都是非常重要的，所以都应该是持久消息。但是在有些性能优先的系统中，适当地采用非持久消息从而保证传递速度也是一种常用的办法，通常系统会再设计某种机制在出错时进行纠正或业务上的补偿。

5.4.3 消息优先级 (Priority)

消息可以按其重要程度划分成不同的优先级，MQ 中消息的优先级可以取从 0 到 9 的任意一个值，0 表示最低，9 表示最高。高优先级的消息在传递和读取时都具有较高的优先权，会被优先处理。消息的优先级与队列管理器和队列的相关属性有一定的内在关系，如表 6-3。

表 6-3 WebSphere MQ 对象中与消息优先级相关的属性

对象	属性	缺省值	可取值	说明
队列管理器	MAXPRTY	9	9	这个值无法修改
队列 (Local/Model)	DEFPRTY	0	0 – 9	
队列 (Local/Model)	MSGDLVSQ	PRIORITY	● PRIORITY ● FIFO	
消息 (MQMD)	Priority	Q. DEFPRTY		

队列管理器的 MAXPRTY 属性表示队列管理器中允许的最大消息优先级，缺省为 9。消息的 Priority 表示消息的优先级，可以取 0 到 QM.MAXPRTY 中的任何一个值。

本地队列或模型队列的 DEFPRTY 属性表示队列上消息的缺省优先级。在 MQPUT/MQPUT1 时，如果 MQMD.Priority = MQPRI_PRIORITY_AS_Q_DEF (-1)，则消息取值为队列的 DEFPRTY 属性。

本地队列或模型队列的 MSGDLVSQ 属性可以设置消息传递次序 (Message Delivery Sequence) 的方式，即 MQGET 时消息选取的次序。它可以有两种取值：PRIORITY 或 FIFO，前者表示消息按优先级从高到低排序，同优先级按先进先出排序，后者表示只按先进先出排序。一旦指定了 FIFO，则设置 MQMD.Priority 不再生效，因为消息只按先后次序排序，优先级这时变得没有意义。FIFO 方式排序时，新放入队列的消息优先级为队列的缺省优先级属性 DEFPRTY。事实上，PRIORITY 和 FIFO 的本质是相同的，都是以优先级为第一索引，以时间为第二索引。只是 FIFO 方式中优先级都一样，等于 DEFPRTY。

如果队列的 MSGDLVSQ 属性从 PRIORITY 变成 FIFO，那么队列中可能存有一些 Priority 低于 DEFPRTY 的消息，新到的消息优先级为 DEFPRTY（不管是否指定 MQMD.Priority），所以新到的消息可能比原先的低优先级消息更早地被处理。反过来，如果 MSGDLVSQ 从 FIFO 变成 PRIORITY，队列中原有的消息优先级应该是 DEFPRTY。

5.4.4 消息超时 (Expiry)

在消息头 MQMD 中的 Expiry 域为消息的超时时间，即消息的有效生命周期，单位是 1/10 秒。这个值表示消息从放入队列开始计算，应该在多长的时间内被取出处理。缺省情况下，MQMD.Expiry = MQEI_UNLIMITED (-1)，表示消息永超时。

Expiry 的计时与超时原理如下：

1. 从消息开始放入队列开始计时。不允许 MQMD.Expiry = 0，否则返回 MQRC_EXPIRY_ERROR。这时 MQMD.PutDate 和 MQMD.PutTime 会记录下 MQPUT/MQPUT1 的时间。
2. 消息在路由的过程中可能会被队列管理器多次放入传输队列并取出传送，所有这些在路上耽搁的时间都会计算在内。事实上，这些内部的 MQPUT/MQGET 动作会使 Expiry 时间衰减。每次消息进入传输队列，队列管理器会在消息头上加上一个 MQXQH，其中也有 Expiry，它会标记消息在这段传输路程上的耗时，在脱去 MQXQH 时会反应到 MQMD.Expiry。
3. 消息最终到达目标本地队列，在那里进入排队。在开放平台的队列管理器中并没有超时监控程序始终监视队列中的每一条消息是否超时。事实上，WebSphere MQ 直到消息被 MQGET 扫描到才会被发现超时。
4. 队列可能是 FIFO 方式，也可能是 Priority 方式，并且 MQGET 还可能匹配条件。这需要队列管理器对队列中的消息从头开始扫描，扫描到的消息都会根据消息创建时间 (MQMD.PutDate, MQMD.PutTime)，消息超时 (MQMD.Expiry) 及当时时间计算出消息是否超时。如果超时，则被删除。如果消息带有超时报告标志 (MQRO_EXPIRATION_*), 则同时发送超时报告。

由于开放平台上 WebSphere MQ 消息超时的发现依赖于 MQGET 对队列的扫描，所以消息的有效生命周期是可以保证的，但超时点可能不准确。换句话说，如果消息躺在队列中一百年，尽管早已超时，只要始终未被 MQGET 扫描到，就一直不会收到超时报告。

对 FIFO 队列而言，不含匹配条件的 MQGET 会从第一条消息一直扫描到第一个未超时的消息。对 Priority 队列而言，同样的 MQGET 会扫描所有更高优先级的消息一直扫描到等高优先级中的第一个未超时的消息。扫描过的部分可能有多条消息被发现超时。

z/OS 上的 WebSphere MQ 有独立的消息自动监控机制。队列管理器有 ExpiryInterval 属性表示每隔多长时间队列管理器自动扫描所有的消息，检查是否有消息超时，并做出相应的处理。ExpiryInterval 单位是秒，取值 1- 99,999,999。也可以取值 MQEXPI_OFF，关闭此功能。

5.5 发送设计

5.5.1 消息标识

WebSphere MQ 中每一条消息都有三个标识：MsgId，CorrelId 和 GroupId。其中，MsgId 是消息自身的标识，CorrelId 是消息的相关标识，GroupId 是消息所有组的标识。这三个标

识都是 MQMD 中的域，长度为 24 字节。

在 MQPUT 或 MQPUT1 时，应用程序可以设置 MQMD.MsgId 来设定消息标识。通常来说，应用系统中不应该出现两条标识相同的消息。所以由应用程序自动设定消息标识的办法并不稳妥，这需要程序自己来保证消息标识的唯一性。

如果在发送消息的时候，MQMD.MsgId 为空，或者设置了发送选项 MQPMO.Options = MQPMO_NEW_MSG_ID，则队列管理器会自动生成一个唯一标识来标记这条消息，该标识由消息标识头“AMQ”、队列管理器名简称、时戳等信息构成，理论上队列管理器不会生成两条 MsgId 完全相同的消息。这样就自动地保证了消息标识的唯一性，不致于在匹配消息时引起混乱。然而，自动生成的消息标识可能含有非打印字符，在表达上稍显困难。

在使用分发列表 (Distribution List) 时，一条消息可能在传送的过程中自动复制成多条消息，为了使 WebSphere MQ 自动为这些消息生成不同的 MsgId，必须设置 MQMD.MsgId 为空，或使用 MQPMO_NEW_MSG_ID 选项。

CorrelId 表示相关的消息标识，比如应答消息用 CorrelId 指明相关的请求消息，报告 (Report) 消息指明相关的原消息等等。在 WebSphere MQ 的应用中，可以有多条消息与某一条原消息相关。也就是说，可以有多条 MsgId 不同的消息，它们的 CorrelId 是相同的。应用程序可以自行设定 MQMD.CorrelId 来设定相关标识。

类似于 MsgId，如果设置发送选项 MQPMO.Options = MQPMO_NEW_CORREL_ID，则消息会自动生成 CorrelId，规则同 MsgId。

GroupId 表示消息所在的组标识，相同的组标识表示这些消息属于同一个组，在匹配时可以整组匹配。详见，“分组与分段”章节。

5.5.2 消息类型

MQMD 的 MsgType 域标志着消息类型。它可以表示消息是请求 (Request) 还是相应的应答 (Reply)，是应用消息 (Datagram) 还是相应的报告消息 (Report)。在发送时设定消息类型，主要是使接收程序在收到该消息后便于区别，从而做出正确的反应。

通常编程时，MQMD.MsgType 可以取值：

- MQMT_REQUEST
- MQMT_REPLY
- MQMT_DATAGRAM
- MQMT_REPORT

5.5.3 消息格式

消息内容有一定的组织格式，比如说是一个结构。结构中的域有些是整数类型的，也有些是字串类型的。由于 WebSphere MQ 所处的操作系统平台不同，从而会引起整数型数据在高低字节编码上的不同，也会引起字串型数据所使用的字符集不同。MQMD 中三个域 Format、Encoding 和 CodedCharSetId 分别可以指明消息的格式、编码、字符集。

Format 可以取值：

- MQFMT_NONE

- MQFMT_ADMIN
- MQFMT_CHANNEL_COMPLETED
- MQFMT_CICS
- MQFMT_COMMAND_1
- MQFMT_COMMAND_2
- MQFMT_DEAD_LETTER_HEADER
- MQFMT_DIST_HEADER
- MQFMT_EVENT
- MQFMT_IMS
- MQFMT_IMS_VAR_STRING
- MQFMT_MD_EXTENSION
- MQFMT_PCF
- MQFMT_REF_MSG_HEADER
- MQFMT_RF_HEADER
- MQFMT_RF_HEADER_2
- MQFMT_STRING
- MQFMT_TRIGGER
- MQFMT_WORK_INFO_HEADER
- MQFMT_XMIT_Q_HEADER

消息分为消息头 (MQMD) 和消息体两部分。MQMD.Format 表示的是消息体的数据结构，Format 可以认为是消息体的数据结构名。不过也有例外，MQFMT_STRING 表示消息体是字串，并非一个结构。如果不知道消息体的结构，则可以用 MQFMT_NONE 来表示。

编码指的是各种类型数据的表示方式，比如高低字节安排等，字符集指的是消息中字符所属的集合，比如中文字符集。消息的发送和接收平台可以不一样，所以当发送和接收出现编码和字符集不一致的情况时，就会发生消息格式转换，即从一种格式自动转向另一种格式。详见“格式转换”章节。

5.5.4 应答队列

应用程序通过 WebSphere MQ 进行请求/应答方式通信的时候，请求方发送消息后通常希望应答消息自动放入指定的队列，该队列就是应答队列。在请求消息的 MQMD 结构中的 ReplyToQMgr 和 ReplyToQ 域指明了应答队列的位置。应答方收到这样的请求消息后，经过消息处理，产生相应的应答消息，会“心领神会”地将应答消息放入应答队列中（图 5-7）。当然，设置应答队列，对于应答方应用程序而言，只有“指导意义”而无“强制意义”。

应答队列的设置，可以使相同的应答程序与不同的请求程序形成多个请求/应答消息环路。请求方可以指定各自的应答队列，互不干扰。另一种做法是多个请求方共用一个应答队列，发送请求消息的时候保证消息的 MsgID 不重复，取应答消息的时候匹配各自消息的 CorrelID。这要求应答方要请求消息的 MsgID 拷贝到应答消息的 CorrelID，以表示是针对哪一条请求消息的应答。

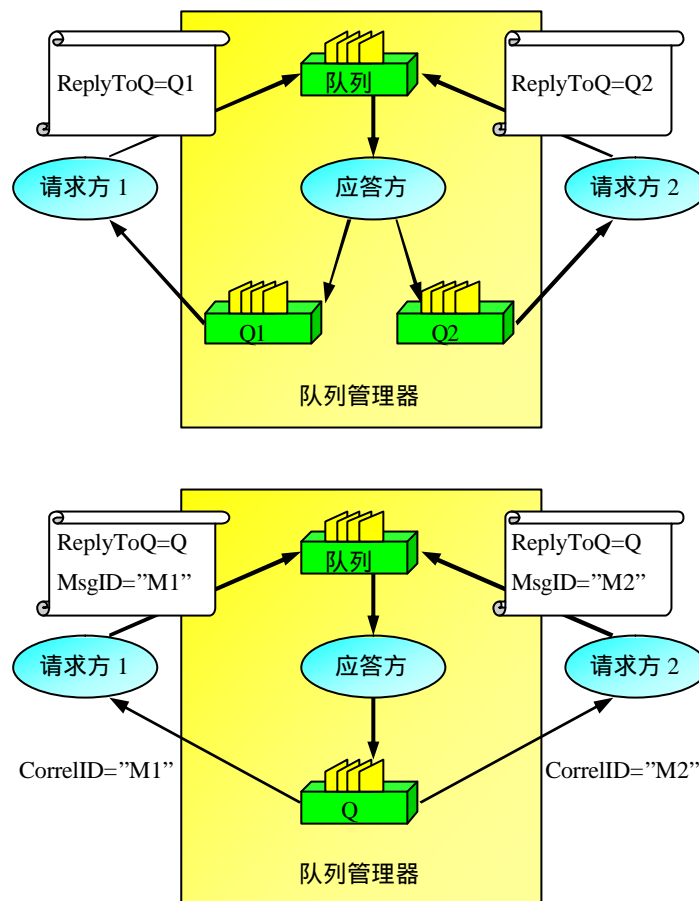


图 5-7 应答队列的设置

通常情况下，请求队列与应答队列是两个不同的队列，但在不致引起混淆的情况下请求队列和应答队列也可以共享同一个队列。

5.5.5 动态队列

打开模型队列时，WebSphere MQ 会按照它的属性自动动态地生成出一个本地队列，称为动态队列。普通本地队列的 DEFTYPE 属性值为 PREDEFINED，动态队列按模型队列的 DEFTYPE 属性可以取值 PERMDYN 和 TEMPDYN，分别表示永久动态队列和临时动态队列。其中永久动态队列指动态队列一旦生成，如果不是显式是将其删除，它将永久地保留在队列管理器中。临时动态队列指队列在关闭时缺省地自动删除。

- 普通本地队列 (Predefined Queue) 队列属性 DEFTYPE = PREDEFINED
- 动态队列 (Dynamic Queue)
 - 临时动态队列 (Temporary) 队列属性 DEFTYPE = TEMPDYN
 - 永久动态队列 (Permanent) 队列属性 DEFTYPE = PERMDYN

动态队列在打开时自动生成，在关闭时自动删除，这种特性使得它能满足很多临时存储的需求。比如偶尔需要生成消息的应答队列来存放应答消息，使用过后就不再需要了。再如一个应用程序在启动时生成一套私用的工作队列，在停止时将现场清理干净。

5.5.5.1 生成动态队列

在 MQOPEN 时,指定对象描述参数 MQOD 的 ObjectType 域为 MQOT_Q,ObjectName 域为模型队列名,DynamicQName 域为动态队列名。这里 DynamicQName 可以取以下值:

- *<Name>* 手工生成,填满队列的名字 (48 个字符)。动态队列为 *<Name>*,如果队列管理器中有重名队列则出错。这里 *<Name>* 为符合队列命名规则的任何一个字串。
- *<Prefix>** 半自动生成,填队列名字的前缀 (少于 33 个字符),最后一个非空字符填 “*”。动态队列名的前缀为 *<Prefix>*,以后的部分自动生成,生成的时候会自动避开已存在的队列名。这里 *<Prefix>* 为符合队列命名规则的任何一个字串。
- * 全自动生成,填 “*”。动态队列名完全由队列管理器自动生成,会自动避开已存在的队列名,不会重复。

MQOPEN 后,生成的动态队列名记录在 MQOD 的 ResolvedQName 域中。动态队列一旦生成了,其 DefinitionType (DEFTYPE) 属性就无法改变了。所以,在一开始选择正确的模型队列 (Model Queue) 很重要。

5.5.5.2 关闭动态队列

永久动态队列是可以永久存在的。在调用 MQCLOSE 时,可以通过关闭选项 (Close Option) 的设置来指定永久动态队列不同的关闭行为。

- MQCO_NONE 永久动态队列会保留下来
- MQCO_DELETE 如果永久动态队列中没有消息,也没有未提交的消息,则删除队列
- MQCO_DELETE_PURGE 如果永久动态队列中没有未提交的消息,则清除现有的所有消息后删除队列

临时动态队列是在此会话 (Session) 中临时存在的,即在线程中 MQOPEN 时队列被自动创建,在同一个线程中 MQCLOSE 时队列被自动删除,下一次 MQOPEN 会重新创建一个临时动态队列。MQCLOSE 时无所谓关闭选项 (Close Option) 的取值。

5.5.5.3 动态队列和持久性消息

永久动态队列由于是可以永久存在的,也就是说,程序退出后甚至队列管理器重新启动后,队列仍然存在。这样,队列一旦生成就和普通的本地队列是一样的。所以,动态队列可以存放持久性消息,也可以存放非持久性消息。

临时动态队列由于其本身是不持久的,程序一旦退出队列自动删除,这样的特性与持久性是相悖的,所以临时动态队列只能存放非持久性消息,不可存放持久性消息。

5.5.5.4 动态队列的使用

动态队列大多数都是为了自动生成,临时使用。常见的使用方式可以是为了生成临时的

应答队列用来临时存放应答消息或报告消息。在 MQOPEN 时临时生成这个应答队列，在 MQPUT 时用消息的 MQMD.ReplyToQ 域来指定该队列，当接收到应答消息后这个队列就没用了，可以用 MQCLOSE 将其删除。

5.5.6 用户替换

消息在从应用程序进入队列的时候，消息的发送者会被填入 MQMD 的 UserIdentifier 域，缺省情况下这就是运行应用程序的用户。

如果在 MQOPEN 时设定 MQOD.AlternateUserId 为指定用户，并选择 MQOO_ALTERNATE_USER_AUTHORITY 选项，则 WebSphere MQ 会用 AlternateUserId 用户身份来打开该队列，实现用户替换。以后的 MQPUT 会将该用户填入 MQMD.UserIdentifier。

类似地，MQPUT1 中含 MQOPEN，可以指定 MQOD.AlternateUserId，并选择 MQPMO_ALTERNATE_USER_AUTHORITY 选项，则效果与 MQOPEN 相同。

在用户替换的过程中，如果 MQOD.AlternateSecurityId 为空，则使用 MQOD.AlternateUserId 来进行权限检查。否则，可以使用 MQOD.AlternateSecurityId。比如，在 Windows 中不同的机器可以有相同的用户名 (UserId)，但某个用户名在域中有唯一的安全标识 (SecurityId)，可以用这样的 SID 放入 MQOD.AlternateSecurityId 进行权限认证。

5.6 读取设计

5.6.1 等待读取 (Wait & NoWait)

MQGET 可以等待读取消息，即先设定消息的匹配条件，然后在队列上等待消息的到来。当然也可以不设条件，等待第一条消息的到来。这时，设置 MQGMO.Options = MQGMO_WAIT，且设置 MQGMO.WaitInterval 为等待时间，单位毫秒。在设定的时间内有匹配的消息到来，则 MQGET 读到消息，返回 MQRC_NONE。否则，MQGET 超时，返回 MQRC_NO_MSG_AVAILABLE。如果 MQGMO.WaitInterval = MQWI_UNLIMITED (-1)，即超时设定为无穷大，则 MQGET 会永远等待在队列上，直到匹配的消息到来。

MQGET 也可以非等待方式读取。设置 MQGMO.Options = MQGMO_NO_WAIT，MQGET 时如果队列中没有匹配的消息，则立即返回。

5.6.2 信号中断 (Signal)

对于 WebSphere MQ for z/OS、Compaq NonStop Kernel Win95/98 环境。可以用信号将消息读取中断。在 MQGET 时设置 MQGMO_SET_SIGNAL 选项，并在 MQGMO 结构的 Signal1 和 Signal2 中设定信号值，则在特定的时候会有信号中断出现。应用程序可以通过截取信号中断的方法，实现并发同时读多个队列。

```
MQLONG  Signal1;  
MQLONG  Signal2;
```

对于 z/OS ,信号值设在 Signal1 中 ,队列管理器自动送出的 ECB (Event Control Block) 中含有信号完成码：

- MQEC_MSG_ARRIVED
- MQEC_WAIT_INTERVAL_EXPIRED
- MQEC_WAIT_CANCELED
- MQEC_Q_MGR QUIESCING
- MQEC_CONNECTION QUIESCING

对于 Compaq NonStop Kernel ,信号值设在 Signal1 中 ,中断消息会发送到进程的 \$RECEIVE 队列。

对于 Windows ,信号值设在 Signal2 中 ,中断消息以 Windows 系统消息的方式送给该进程。

5.6.3 截断消息 (Truncated Message)

MQGET 的时候如果队列中的消息长度大于准备接收的 Buffer 长度 ,则可能发生消息截断 ,即消息被取出 ,但只有前一部分放入 Buffer 中 ,后一部分数据丢失。

有时应用上的确需要这种截断功能 ,比如只需要读取消息的前 100 个字节。这时 ,设置 MQGMO.Options = MQGMO_ACCEPT_TRUNCATED_MSG。在 MQGET 时 ,如果消息长度大于 BufferLength ,则 DataLength 表示消息的实际长度 ,消息被取出 ,但只读到前一部分 ,返回 MQRC_TRUNCATED_MSG_ACCEPTED。

有时应用上需要读取完整的消息 ,避免截断情况的发生。这时 ,MQGMO.Options 上不要设定 MQGMO_ACCEPT_TRUNCATED_MSG 选项。在 MQGET 时 ,如果消息长度大于 BufferLength ,则 DataLength 表示消息的实际长度 ,消息仍然保留在队列中 ,返回 MQRC_TRUNCATED_MSG_FAILED。通常在这种情况下 ,应用程序应该按 DataLength 分配实际需要的内存 ,再进行一次 MQGET 将消息读出。

在 MQGET 时如果发生数据转换 ,则消息的长度可能会有所变化。在 MQGET 返回后 ,如果 MQGMO.ReturnedLength 不为 MQRL_UNDEFINED (-1) ,则表示返回的字节数 ,通常情况下与 DataLength 相同。如果为 MQRL_UNDEFINED ,则以 DataLength 为准。

5.6.4 浏览消息 (Browse)

浏览消息本质上是读消息的一种特殊方式 ,只是消息队列中并没有被自动删除。在打开队列时用 MQOO_BROWSE 指明浏览方式打开并按匹配条件依次读取消息内容。以浏览方式打开队列时会创建一个游标 ,其工作原理类似于数据库游标 ,它会记住当前浏览的位置 ,以后调用 MQGET (MQGMO_BROWSE_FIRST) 可以游标定于匹配消息集合的第一条上 ,MQGET (MQGMO_BROWSE_NEXT) 可以依次浏览。

MQGET (MQGMO_BROWSE_MSG_UNDER_CURSOR) 可以再次浏览游标指向的消息

息，游标不动。MQGMO_BROWSE_* 可以和 MQGMO_LOCK 一起使用，将这条消息锁住，此消息对其它并发应用变为不可见。接着可以用非浏览方式的 MQGET (MQGMO_MSG_UNDER_CURSOR) 将该消息读走。

下面是一段伪代码，用于浏览队列中的消息并取走所需的消息：

```
MQOPEN (MQOO_BROWSE)
MQGET (MQGMO_BROWSE_FIRST + MQGMO_LOCK)
while (not found)
{
    MQGET (MQGMO_BROWSE_NEXT + MQGMO_LOCK)
    if (found)
    {
        MQGET (MQGMO_MSG_UNDER_CURSOR)
        break;
    }
}
MQCLOSE
```

5.6.5 格式转换 (Convert)

WebSphere MQ 中的数据格式可以根据需要自动地转换成目标字符集和编码方式。所谓字符集 (CodedCharSetId) 就是指消息的字符所属的文字集，如单字节英语码，双字节汉语码等等。编码方式 (Encoding) 指的是整数的高低字节安排、浮点数的精度和幂的安排等等，通常应该符合 IEEE 标准。

WebSphere MQ 中的格式转换可以出现在三个地方：

1. 在 MQGET 的时候，用 MQGMO_CONVERT 选项，这时 MCA 在接收消息的时候，会根据消息中的 CodedCharSetId 和 Encoding 域与应用程序的环境进行比较，如果需要则自动进行转换。
2. 在发送方的通道设置属性 CONVERT=YES，这时 MCA 在发送消息的时候，会根据消息中的 CodedCharSetId 和 Encoding 域与对应队列管理器的环境进行比较，如果需要则自动进行转换。
3. 在发送方或接收方设置数据转换用户出口，在情况 1 或 2 中需要数据转换时，WebSphere MQ 会根据消息类型 (Type 域) 来调用同名的用户出口程序。

5.6.6 消息匹配 (Match)

在读消息的时候，可以从队列中的一大堆消息中匹配出所需的部分。消息匹配有三种方式，可以根据消息 MsgId、CorrelId 或 Message Token 来进行匹配。相应地，MQGMO 的匹配选项应该设置为：MQMO_MATCH_MSG_ID、MQMO_MATCH_CORREL_ID 或 MQMO_MATCH_MSG_TOKEN。如果 MQGMO.Options = MQMO_NONE，则不进行消息匹配，WebSphere MQ 会按优先级 (Priority) 或先进先出的原则选取第一条消息。

例：

队列中有两条消息：

消息 1：MQMD.MsgId = “M1”，MQMD.CorrelId = “C1”

消息 2：MQMD.MsgId = “M2”，MQMD.CorrelId = “C2”

在 MQGET 的时候，如果参数 MQMD 的 MsgId 域设为 “M1”，且 MQGMO.Options = MQMO_MATCH_MSG_ID，则取出消息 1。如果参数 MQMD 的 CorrelId 域设为 “C2”，且 MQGMO.Options = MQMO_MATCH_CORREL_ID，则取出消息 2。

5.6.7 回滚计数 (Backout Count)

在交易中，MQGET (MQGMO_SYNCPOINT) 从队列中读取的每一条消息在交易失败时都会回滚放回队列中。回滚后的消息可以参与下一个交易，然而，下一个交易也可能因为相同的原因不成功，造成消息再次被回滚。为了防止这种反复回滚现象，WebSphere MQ 设计了回滚计数 (Backout Count) 机制。初始时回滚计数器为零，消息回滚后计数器会自动加一，在下次 MQGET 时 MQMD.BackoutCount 可以取得计数器的值。当计数器的值达到阈值时，应用程序应该将其做特殊处理，而不是简单地再次回滚。

反复回滚的消息，我们称之为“有毒消息”(Poison Message)。因为这种消息往往会使应用陷入反复无效操作，对性能有害。一般来说，发现“有毒消息”应将其转入特殊处理流程，比如放入指定队列中隔离，同时应及时报警，这往往意味着某些资源工作不正常，或者应用设计有漏洞。

在本地队列或模型队列上有两个相关属性 BOTHRESH 和 BOQNAME，分别指明 BackoutCount 的阈值和超过阈值后应该放入的队列名。但是，这两个属性除了设定一个建议值以外，WebSphere MQ 本身并不会做相应的隔离动作。换句话说，WebSphere MQ 自身不会发现并处理“有毒消息”，这时应用应该做的。目前，建立在 WebSphere MQ 之上的一些产品，比如 WebSphere Business Integration Message Broker，WebSphere MQ Workflow 等等都已经具有这个功能。

5.6.8 固化回滚计数 (Harden Backout)

既然反复回滚可能是由某些单元工作不正常引起，那么如果队列管理器自身遇到这样的问题，最简单的办法就是重启队列管理器。回滚计数器是可以跨越队列管理器重启后继续使用的，但是如果队列管理器自身有问题，这一点就无法保证。

解决办法是固化计数 (Harden Backout)，在每次回滚时，对每条消息及时地将 BackoutCount 记录写到硬盘上去。本地队列或模型队列有一个属性 HARDENBO (也可以将其设置成 NOHARDENBO)，这样，这个计数器就可以在内存和硬盘上各有一分拷贝。对于 Compaq OpenVMS Alpha, OS/2, OS/400, Compaq NSK, UNIX, Windows，BackoutCount 只能是固化的，对 HARDENBO 修改无效。

5.7 容错设计

5.7.1 出错处理

WebSphere MQ 的每一个 API 调用都有可能出错，当错误发生时，调用操作返回的完成码 (Completion Code) 可以指明是警告还是错误，原因码 (Reason Code) 可以指明错误原因。一般来说，每一个 API 调用都需要判断这两个返回码，以决定程序是继续执行还是转去相应地出错处理。如果要中止程序的执行，应该注意关闭所有打开的对象句柄，最后 MQDISC 断开与队列管理器的连接。

对于 Java 程序，可以通过处理各种异常 (Exception) 来简化出错处理的逻辑，使代码更加清晰。通常，一个健壮的程序会有各种考虑到各种出错情况并有相应的出错处理代码，越早发现错误并处理对应用系统的代价越小。所以，在监控每一个 API 调用是否成功并及时处理是一个好方法。

5.7.2 报告消息

对于网络多队列管理器环境中，消息的传递成功与否不由发送方控制。换句话说，发送方发送消息的操作本身是成功的，这只能说明 WebSphere MQ 受理了该消息的传递，可是发送方并不知道消息是否真的能够到达对方并被对方处理。在应用设计时如果发送方需要知道这一点，可以要求得到一个“回执”，这就是报告消息。

报告消息常用的有以下几种：

- 异常 (EXCEPTION) 消息在传递过程中出错异常情况，不可送达。
- 超时 (EXPIRATION) 消息在传递过程中超时，未被对方 MQGET 取走。
- 成功到达 (COA) 消息成功到达目标队列。
- 成功阅读 (COD) 消息成功到达目标队列，并被对方 MQGET 取走。
- 处理成功 (PAN) 消息处理成功
- 处理失败 (NAN) 消息处理失败

报告消息中前 4 种只是描述传递是否成功，由队列管理器发出回执消息。后 2 种描述处理是否成功，由处理程序显式发出回执消息。发送方要求的报告消息选项可以叠加，如 COA+COD，表示消息发出后有可能收到两个相关回执。

5.7.3 死信消息

当消息在无法送达目标队列时会成为死信消息，这时往往意味着应用系统中出现了问题使消息无法正常投递，而这个问题是发送方和接收方都无法感知的。所以，及时地发现并处理死信消息对保证系统正常运作大有好处。

通过配置队列管理器的缺省死信队列使死信消息能够就近存放，通过监控死信队列中的死信消息可以得知死信产生的原因。应用设计者可以选择自行设计并编写死信队列管理程序

或使用 WebSphere MQ 缺省的死信队列管理器 runmqdlq。

对死信消息的处理通常有三种方式：重做、隔离、报告。重做就是将死信消息不成功的操作重做一遍，这对于解决系统临时故障是有效的。比如，消息因为目标队列暂时满了而无法放入，隔一段时间重放一遍就可能成功。再如，消息因为网络原因暂无法发送，隔一段时间重发就可能成功。但是，对消息重做一遍很可能会因为相同的原因不成功而再次成为死信消息，形成死循环，我们在设计上一定要避免这种情况的发生。隔离就是将死信消息放入特定的队列，以便做事后的统计和分析，积累到一定程度可以报警以进行人工干预。报告就是在死信消息隔离同时产生报告消息，而报告消息的目标队列与死信消息的 ReplyToQ 域或 DestQName 域相同，这样消息的发送方或接收方可以感知到死信消息的产生而做出正确的反应。

5.8 小结

本章我们详细介绍了 WebSphere MQ 应用设计时要考虑的各个要素，这些要素都是设计之初需要确定下来的。通常设计人员应该先从应用结构、通信方式、实现功能处入手，设计的方案应该尽可能贴近于应用需求而又保持相当的灵活性，这时尚不需要涉及具体的布署和配置、运行平台、采用何种编程语言等实现细节。

以下我们会对 WebSphere MQ 的各种功能做更加详细的介绍，因为大多数功能需要动手配置或编程实现，请读者注意书中内容与书后例程的结合。

第 6 章 消息处理

这一章介绍 WebSphere MQ 消息处理的相关功能，包括消息的交易控制、消息触发、消息分组与分段，以及消息上下文设计。它们都是 WebSphere MQ 中针对消息处理的一些基本功能，如果掌握了这些内容并灵活使用，可以使您的应用在消息处理上变得随心所欲。

6.1 交易 (Transaction)

WebSphere MQ 应用程序通常会有多次读写队列操作，这些操作缺省是相互独立的，也就是说前一次操作的失败或失败与后一次操作的结果是无关的。但有时候在应用设计时会碰到这样的需求，即一系列的队列操作在业务上实际上是一个原子操作，要么全部成功，要么全部失败，不能出现一部分成功而另一部分失败的“中间状态”。这时，我们就要借助于 WebSphere MQ 的交易功能了。

所谓交易，就是将一系列连续的操作纳入一个事务，最终同时提交或回滚，从而保证只有成功或失败两种状态。这里的操作可以是队列读写操作，也可以是其它资源的操作，比如数据库操作。

6.1.1 概述

在交易中通常有两种角色，一种是交易管理器，用于发起并控制交易，最终负责交易的提交或回滚，另一种是资源管理器，用于参与交易并做具体的操作，比如读、写、增、删、改等等。

WebSphere MQ 的队列管理器既可以作为交易管理器发起并控制交易，也可以作为资源管理器参与交易。无论哪一种方式，WebSphere MQ 所关心的是对队列中消息的操作可以在交易中提交或回滚。

对于 MQGET，是否参与交易是由 MQGMO.Options 来指定的。

- MQGMO_SYNCPOINT MQGET 参与交易
- MQGMO_SYNCPOINT_IF_PERSISTENT 如果消息是 PERSISTENT，则 MQGET 参与交易
- MQGMO_NO_SYNCPOINT MQGET 不参与交易
- MQGMO_MARK_SKIP_BACKOUT MQGET 参与交易，但交易回滚时该消息不能回滚

这里 MQGMO_NO_SYNCPOINT 和 MQGMO_MARK_SKIP_BACKOUT 效果上是相同的，但实现上不同。前者因为不参与交易，MQGET 后消息从队列中被立即删除。后者 MQGET 后消息仍保留在队列中，队列深度不变，只是对其它应用不可见，无论交易是提交或回滚，到那一刻消息才被删除。

对于 MQPUT/MQPUT1，是否参与交易是由 MQPMO.Options 来指定的。

- MQPMO_SYNCPOINT MQPUT/MQPUT1 参与交易
- MQPMO_NO_SYNCPOINT MQPUT/MQPUT1 不参与交易

队列管理器上有一个属性 MaxUncommittedMsgs 表示交易中涉及的最大消息数量，可以在创建队列管理器 crtmqm 时用 -x 指定，也可以事后调整。它的取值范围为 1 - 999,999,999，缺省值为 10,000。这里涉及的消息包括：MQPUT/MQPUT1 (MQPMO_SYNCPOINT) 及其引发的 COA 报告消息，MQGET (MQGMO_SYNCPOINT) 及其引发的 COD 报告消息。当交易中涉及的消息数量达到这个容量阈值时，MQPUT/MQPUT1 或 MQGET 会出错，返回 MQRC_SYNCPOINT_LIMIT_REACHED。对于要处理大交易的系统，最好事先将 MaxUncommittedMsgs 属性调大。

6.1.2 本地交易 (Local LUW)

本地交易也就是说只有队列操作参与交易，这也是最简单的交易模式，WebSphere MQ 使用一阶段提交技术将连接句柄上的所有队列操作一次提交或回滚。交易的开始点为 MQCONN 或上一次 MQCMIT，不需要 MQBEGIN 来标记交易开始。

WebSphere MQ 中涉及消息操作的 API 一共只有三个：MQGET，MQPUT，MQPUT1。在编程时将相应的 MQGMO.Options 和 MQPMO.Options 设置 SYNCPOINT，如下：

```
MQGMO.Options += MQGMO_SYNCPOINT
MQPMO.Options += MQPMO_SYNCPOINT
```

在程序中可以用 MQCMIT 或 MQBACK 显式地提交或回滚交易，也可以隐式地进行。如果程序没有调用 MQCMIT，而直接 MQDISC，则隐含 MQCMIT 提交交易。如果程序没有调用 MQCMIT，也没有调用 MQDISC，而直接结束，则隐含 MQBACK 提交交易。例：

```
MQGET (MQGMO_SYNCPOINT)
MQPUT (MQPMO_SYNCPOINT)
MQCMIT
```

6.1.3 全局交易 (Global LUW)

全局交易也就是说参与交易的除了队列管理器之外还有其它资源，比如数据库。由于牵涉到多个数据源，一般采用两阶段提交技术，对所有的资源操作一起提交或回滚，这要求交易管理器与资源管理器之间符合 XA 接口标准。

WebSphere MQ 的队列管理器既可以作为交易管理器发起并控制交易，也可以作为资源管理器参与交易。前者，交易由 WebSphere MQ 发起并控制，也称为内部交易。后者，交易由第三方交易中间件（比如 CICS，Tuxedo）发起并控制，也称为外部交易。

6.1.3.1 内部交易

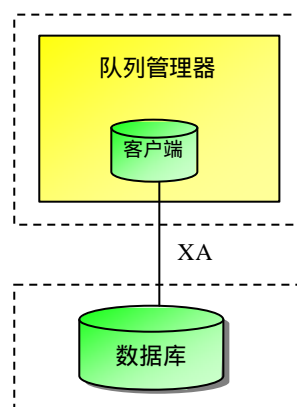


图 6-1 内部交易

内部交易中 WebSphere MQ 作为交易管理器。数据库可以是多个。队列管理器可以通过数据库客户端与数据库服务器连接，所以队列管理器可以与数据库服务器分在两台机器上。图 6-1。

配置内部交易环境需要在队列管理器和数据库两侧配置 XA 接口，分以下两个步骤：

1. 配置数据库

以 DB2 为例，要将 DB2 纳入 MQ 交易管理。

```
db2 update dbm cfg using TP_MON_NAME MQ
```

2. 配置队列管理器

将指定数据库的交换文件进行编译，源文件在 xatm 目录下。以 DB2 为例，需要将 db2swit.c 编译为 db2swit.dll

- 在 Windows 中，运行 WebSphere MQ 服务，选择相应的队列管理器，右键选择“属性”。在资源一档中可以新建一个 XA 设置，交换文件 (Switch File) 中指定编译后动态可执行文件的全路径名，XA 打开字符串为 `database,username,password`，线

程控制可以选择 “PROCESS” 或 “THREAD”，PROCESS 表示 MQ 以进程为单位与数据库建立一条连接，这意味着进程中的多个线程只能串行使用这条连接，THREAD 表示 MQ 以线程为单位与数据库连接，这样，多线程与数据库有多条连接，并行运行。

注意 检查 <MQ_HKEY>\Configuration\Services\队列管理器\XAResourceManager 应该是英文 PROCESS 或 THREAD，而不是中文“进程”或“线程”。

- 在 UNIX 平台中，编辑 qm.ini 中的 XAResourceManager 一段，类似地可以添加一个 XA 设置。

在编程时需要以 MQBEGIN 标志交易开始，MQGET，MQPUT，MQPUT1 时将相应的 MQGMO.Options 和 MQPMO.Options 设置 SYNCPOINT。程序中不需要与数据库的连接与断连语句，可以用嵌入式 SQL 直接操作数据库。例：

```
MQBEGIN
MQGET (MQGMO_SYNCPOINT)
MQPUT (MQPMO_SYNCPOINT)
EXEC SQL ...
MQCMIT
```

在程序中可以用 MQCMIT 或 MQBACK 显式地提交或回滚交易，也可以隐式地进行。如果程序没有调用 MQCMIT，而直接 MQDISC，则隐含 MQCMIT 提交交易。如果程序没有调用 MQCMIT，也没有调用 MQDISC，而直接结束，则隐含 MQBACK 提交交易。

6.1.3.2 外部交易

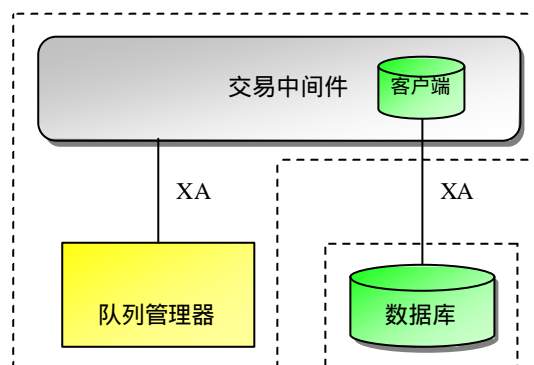


图 6-2 外部交易

外部交易中 WebSphere MQ 作为资源管理器，需要有交易中间件作为交易管理器。这里队列管理器可以有多个，数据库也可以有多个，但交易中间件与队列管理器必须在同一台机器上。图 6-2。

同样，配置外部交易环境要在交易中间件和数据库上进行设置，分以下两个步骤：

1. 配置中间件

以 CICS 为例，将交换文件进行编译，源文件在 <InstallDir>\Tools\c\Samples\amqzscin.c，将其编译为 amqzscin.dll。

```
cicsadd -c xad -r CICSRRGN MQXA SwitchLoadFile=amqzscin.dll XAOpen="QM"
cicsadd -c xad -r CICSRRGN DBXA SwitchLoadFile=cicsxadb2.dll \
```

```
XAOpen="sample,db2admin,db2admin"
```

2. 配置数据库

以 DB2 为例，要将 DB2 纳入 CICS 交易管理。

```
db2 update dbm cfg using TP_MON_NAME CICS
```

在编程时按 CICS 编程模式程序从开始执行即为交易开始，不需要 MQBEGIN 来标记交易开始。MQGET，MQPUT，MQPUT1 时将相应的 MQGMO.Options 和 MQPMO.Options 设置 SYNCPOINT。程序中不需要与数据库的连接与断连语句，可以用嵌入式 SQL 直接操作数据库。不用 MQCMIT 或 MQBACK 来提交或回滚交易，用 EXEC CICS SYNCPOINT 或 EXEC CICS SYNCPOINT ROLLBACK 取而代之。例：

```
MQGET (MQGMO_SYNCPOINT)
MQPUT (MQPMO_SYNCPOINT)
EXEC SQL ...
EXEC CICS SYNCPOINT
```

6.1.3.3 扩展交易客户端 (Extended Transactional Client)

WebSphere MQ Extended Transactional Client (MQETC) 是 WebSphere MQ Client 的一个延伸。需要 WebSphere MQ 5.3 Client + CSD03 以上的安装环境。交易中间件可以通过 MQETC 与队列管理器连接，通过数据库客户端与数据库服务器连接。这样，交易中间件、队列管理器、数据库服务器可以分别在三台不同的机器上。图 6-3。

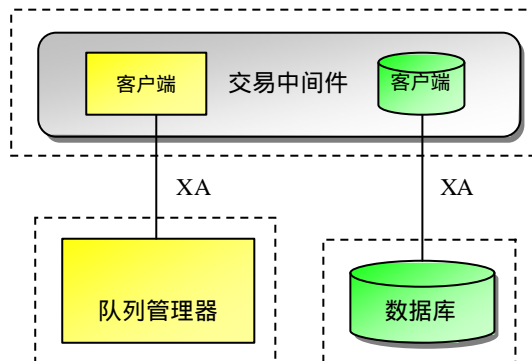


图 6-3 扩展交易客户端

在 WebSphere MQ Client 的应用程序中，MQ Client 与 MQ Server 之间打交道是通过 MQI 通道传递 MQ API 参数的。MQI 通道在 MQCONN 或 MQCONNX 时连通，在 MQDISC 时断开。WebSphere MQ Client 也可以参与 MQ 交易，比如它可以用 MQCMIT 或 MQBACK 来控制应用程序中对多个队列的操作处于一个交易中。但是，普通的 WebSphere MQ Client 无法在同一个交易中对其它资源管理器（如 DB2 数据库）操作，且把这些操作纳入它的交易中。这是因为：

1. 如果用 MQ Server 来做交易管理器，则对数据库的操作应该在 Server 端进行。MQ Client 端自身没有与资源管理器（数据库）的 XA 接口，所以它不能以交易管理器的身份参与全局内部交易。
2. 如果用第三方的交易管理器（比如 TXSeries, Tuxedo, WebSphere Application Server, MTS ...）。普通 MQ Client 端自身没有与交易管理器的 XA 接口，所以它不能以资源

管理器的身份参与交易全局外部交易。

如果要做到这一点，就需要 MQETC。实际上，MQETC 比普通的 MQ Client 就多了一点点参与交易的功能。它提供了与第三方交易管理器的 XA 接口，所以它能以资源管理器的身份参与交易。

MQETC 的平台支持情况如下：

- MQETC 支持的平台有：AIX、HP-UX、Linux for Intel、Linux for zSeries、Solaris、WinNT/2000/XP。
- MQETC 不支持的平台有：OS/400、Win98、z/OS
- MQETC 支持的第三方交易管理器有：TXSeries、Tuxedo、WebSphere Application Server
- MQETC 可连接的 MQ Server 平台有：AIX、HP-UX、iSeries、Linux for Intel、Linux for zSeries、Solaris、Windows

6.1.3.3.1 配置

与全局外部交易相同，交易中间件也需要配置 XA 接口设置。其中最关键的是交换文件 (Switch Load File) 和接口字串 (XA Open String)。

XA Open String 的格式为 *parm_name1=parm_value1,parm_name2=parm_value2, ...* 其中 *parm_name* 和 *parm_value* 的取值有些是大小写无关的。表 6-1。

表 6-1 XA 接口字串中的参数

parm_name	parm_value	解释
CHANNEL	通道名	MQI Channel 的名字
TRPTYPE	<ul style="list-style-type: none">● LU62● NETBIOS● SPX● TCP	缺省为 TCP 与 CHANNEL 一起设置
CONNAME	相应的连接参数	与 CHANNEL 一起设置
QMNAME	队列管理器名	必须提供。必须明确写明，不能为 * 或 * 打头
TPM	<ul style="list-style-type: none">● CICS● ENCINA● TUXEDO	交易管理器
AXLIB	库文件名	可以指定支持动态注册的 XA 库，包含 ax_reg 和 ax_unreg 函数

例：*channel=MARS.SVR,trptype=tcp,conname=MARS(1415),qmname=MARS,tpm=cics*

XA Open String 中的 CHANNEL、TRPTYPE、CONNAME：

- 如果 XA Open String 中定义了 CHANNEL、TRPTYPE、CONNAME，则 MQETC 使用它们来连接 MQ Server。
- 如果 XA Open String 没有定义 CHANNEL、TRPTYPE、CONNAME，则 MQETC 使用 MQSERVER 环境变量来连接 MQ Server
- 如果 MQSERVER 环境变量也没有定义，则 MQETC 使用 Channel Definition

- Table，队列管理器名由 XA Open String 中的 QMNAME 指定
- 以上任何一种方式, MQETC 都会检查 XA Open String 中的 QMNAME (必须项) 与 MQ Server 端的 Queue Manager 是否同名，如果不同名 XA Open 失败。

XA Open String 中的 TPM、AXLIB 关系如表 6-2。

表 6-2 XA 接口字符串中的 TPM 和 AXLIB 的关系		
TPM	平台	缺省的 AXLIB
CICS	AIX	/usr/lpp/encina/lib/libEncServer.a(EncServer_shr.o)
CICS	HP-UX	/opt/encina/lib/libEncServer.sl
CICS	Solaris	/opt/encina/lib/libEncServer.so
CICS	Windows	libEncServer
Encina	AIX	/usr/lpp/encina/lib/libEncServer.a(EncServer_shr.o)
Encina	HP-UX	/opt/encina/lib/libEncServer.sl
Encina	Solaris	/opt/encina/lib/libEncServer.so
Encina	Windows	libEncServer
Tuxedo	AIX	/usr/lpp/tuxedo/lib/libtux.a(libtux.so.60)
Tuxedo	HP-UX	/opt/tuxedo/lib/libtux.sl
Tuxedo	Solaris	/opt/tuxedo/lib/libtux.so.60
Tuxedo	Windows	libtux

如果 XA Open String 中含有 AXLIB，则缺省值被覆盖。

XA Open 与 MQCONN 的次序：

- 如果 XA Open 在先，MQCONN/MQCONNXX 在后，MQETC 检查 MQCONN/MQCONNXX 中的队列管理器与已有的 MQI Channel 连接的队列管理器是否一致。如果一致，返回已有的 MQI Channel 的连接句柄。如果不一致，MQCONN/MQCONNXX 失败 (MQRC_ANOTHER_Q_MGR_CONNECTED)。如果 MQCONN/MQCONNXX 中的队列管理器为 *，MQCONN/MQCONNXX 还是失败 (MQRC_Q_MGR_NAME_ERROR)。
- 如果 MQCONN/MQCONNXX 在先, XA Open 在后，MQETC 检查 XA Open String 中的队列管理器与 MQI Channel 连接的队列管理器是否一致。如果一致，复用已有的连接，XA Open 成功。如果不一致，XA Open 失败。在这种情况下，MQCONN/MQCONNXX 如果使用 *，则真正的 MQI Channel 连接可能是 Channel Definition Table 匹配到的任何一条，这时只有用户来保证后来的 XA Open 与先前的 MQCONN/MQCONNXX 的一致性。
- 无论哪一种方式，两者会复用一条 MQI Channel，而不会在同一线程中各自建立不同连接

xa switch structure 是交易中间件用来接受队列管理器注册时的数据结构。MQETC 支持两种结构：

- MQRMIXASwitch 静态注册
- MQRMIXASwitchDynamic 动态注册

表 6-3 xa switch structure 库文件	
平台	含有 xa switch structure 的库文件
AIX	/usr/mqm/lib/libmqcxa

HP-UX、Linux、Solaris	/opt/mqm/lib/libmqcxa
Solaris	C:\Program Files\IBM\WebSphere MQ\bin\mqcxa.dll

在 switch structure 中代表 WebSphere MQ 资源管理器的名字是 MQSeries_XA_RMI，多个队列管理器可以共享同一个 switch structure。如果使用 CICS 可以在 cicstail 中看见以下字样：

```
ERZ080090I/0801 2003-8-26 10:05:32 CICS RGN : 对于服务器 102 连接到 'MQSeries
_XA_RMI', XA RECOVER 已落实
ERZ080090I/0801 2003-8-26 10:05:32 CICS RGN : 对于服务器 102 连接到 'DB2 for
WINDOWS', XA RECOVER 已落实
```

如果配置的 AXLIB 使用静态注册，则交易中间件每次交易能会与队列管理器打交道，调用 xa_start、xa_end、xa_prepare，尽管队列管理器可能不参与某些交易。

如果配置的 AXLIB 使用动态注册，则交易中间件缺省认为队列管理器不参与交易，不会调用 xa_start，只有到操作到队列管理器的时候才动态注册，调用 ax_reg 将其纳入交易中。动态注册是对 xa 的一种优化，因为它减少了对 xa_* 函数的调用次数。

配置举例：

- MQ Server 上的配置

```
DEFINE QLOCAL (Q) +
REPLACE
DEFINE CHANNEL (C_C.S) +
CHLTYPE (SVRCONN) +
TRPTYPE (TCP) +
REPLACE
```

- 交易中间件 CICS 上的配置

```
cicsadd -c xad -r $(REGION_NAME) -P MQXA SwitchLoadFile=amqzscin.dll
XAOpen="CHANNEL=C_C.S,TRPTYPE=TCP,CONNAME=127.0.0.1(1414),QMNAME=QM,TPM=C
ICS,AXLIB=libEncServer"
```

6.1.3.3.2 编程

MQETC 编程时的要求与外部交易相同，要遵守第三方交易管理器的编程规范，不可以使用 MQBEGIN、MQCMIT、MQBACK。

MQ Client 中一个线程可以同时连接多个队列管理器，MQETC 则不能。在 MQETC 中，一个线程只可以同时连接一个队列管理器，但不同的线程可以连接不同的队列管理器，连接句柄不同被多个线程共享。

6.2 触发 (Trigger)

6.2.1 原理

WebSphere MQ 中的本地队列或模型队列上可以设置消息触发器 (Trigger), 到达的消息在优先级、数量和到达方式上如果满足预先设定的条件, 则会有触发消息自动生成, 放入指定的初始化队列, 再由触发监控器 (Trigger Monitor) 将触发消息读出, 并根据触发消息的内容启动相应的应用程序。一般说来, 该程序应该首先从本地队列中将应用消息读出并进行应用处理。图 6-4。

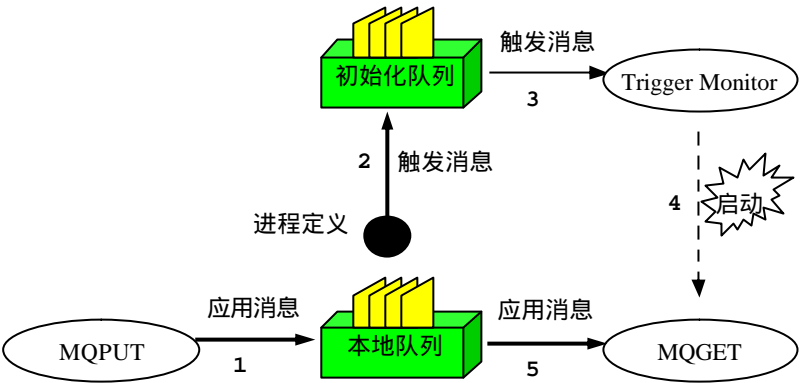


图 6-4 消息触发的原理

其中, 本地队列、初始化队列、进程定义是事先设置好的 WebSphere MQ 对象, 触发监控器也是处于运行状态的, 即在初始化队列上等待触发消息, 一旦读到触发消息, 则可以立即启动 MQGET 应用程序, 所以 MQGET 应用程序是被触发启动的, 通常情况下是触发监控器的子进程。所有这些都只是设定了“连锁反应”中的各个环节, 而触动这个机关的是应用消息的到达。

WebSphere MQ 中的队列管理器、队列、进程定义都有一些属性与消息触发有关, 表 6-4 列举了 WebSphere MQ 对象中与消息触发相关的属性。通过配置这些对象属性可以设置消息触发方式、触发间隔以及被触发的应用程序。

表 6-4 WebSphere MQ 对象中与消息触发相关的属性				
对象	属性	缺省值	可取值	说明
队列管理器	TRIGINT	999,999,999	0 - 999,999,999	TriggerInterval, 单位毫秒, 只对 FIRST 方式有效
队列 (Local/Model)	TRIGGER	NOTRIGGER	● TRIGGER	TriggerControl
			● NOTRIGGER	
队列 (Local/Model)	TRIGTYPE	FIRST	● FIRST	TriggerType
			● EVERY	
			● DEPTH	
队列 (Local/Model)	TRIGDPTH	1		TriggerDepth
队列	TRIGMPRI	0		TriggerMsgPriority

(Local/Model)				
队列	TRIGDATA	空	最多 64 字节	触发数据字符串，可以用来指明触发通道名
(Local/Model)				
进程	APPLICID	空		应用程序名
进程	APPLTYPE	空		应用程序平台
进程	USERDATA	空		用户数据，也可以用来指明触发通道名
进程	ENVRDATA	空		环境数据

队列上的 TRIGGER 选项有两种取值：TRIGGER 或 NOTRIGGER，它相当于触发器的开关。如果设置成 NOTRIGGER，则不会有触发发生，所有与触发相关的设置全部失效。

队列上共有三种触发方式：FIRST、EVERY、DEPTH，由队列属性 TRIGTYPE 指定。无论哪一种方式，能够满足触发条件的消息优先级必须大于等于队列触发优先级属性 TRIGMPRI。

队列管理器上的 TRIGINT 属性表示 FIRST 方式时触发的复位时间。仅对 FIRST 方式有效。队列上触发深度属性 TRIGDPTH 表示 DEPTH 方式时满足触发条件的队列深度，仅对 DEPTH 方式有效。

6.2.2 触发方式

WebSphere MQ 支持三种触发方式：FIRST、EVERY 和 DEPTH，由本地队列或模型队列的 TRIGTYPE 属性控制。

6.2.2.1 FIRST

顾名思义，即“第一条”消息会引起触发，以后紧跟的消息则不会。WebSphere MQ 规定，如果队列深度从 0 到 1 的时候，发生 FIRST 触发。以后一段时间内后继到达的消息只要不是将队列深度从 0 变成 1，则不会再引起 FIRST 触发。在这段时间过后，FIRST 触发复位，下一个到达的消息会再次引起 FIRST 触发。图 6-5。

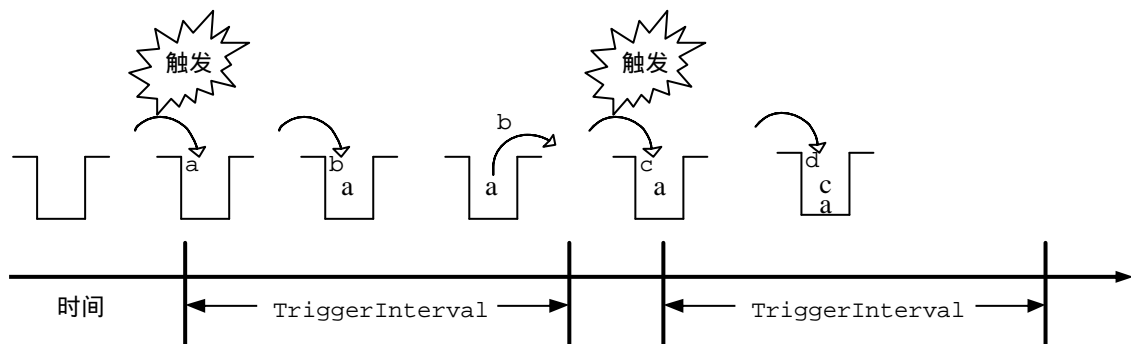


图 6-5 FIRST 触发方式

所以，准确地说，FIRST 不是指队列中到达的第一条消息，而是一段时间内到达的第一条消息。这里所谓的“一段时间”是由队列管理器属性 TRIGINT (TriggerInterval) 指定的，TriggerInterval 单位毫秒，缺省值为 999,999,999，是一段很长的时间，大约 11.5 天。如图，从触发开始计时，在 TriggerInterval 期间，FIRST 触发机制被抑制，只有在这段时间过后，FIRST 触发机制才重新起作用。

在应用设计与布署的时候，要适当地调整 TriggerInterval 时间。一批消息到达的时候，以第一条消息触发起应用程序，一般来说，触发起来的应用程序会连续处理队列中的消息直到队列被取空。后到的消息会在队列中等待一段时间，到 FIRST 触发机制复位后，由以后的第一条消息再次触发起来的应用程序进行处理。如果消息到达队列是均匀的，间隔时间为 T 则消息在队列中的最大等待时间为 TriggerInterval + T。如果 TriggerInterval = 0 则 FIRST 与 EVERY 的效果相同。

6.2.2.2 EVERY

EVERY 方式即每次消息的到达都会引起消息触发。但由于消息到达、触发消息生成、应用进程启动、消息处理四个步骤是异步流水线方式完成的，所以，如果消息以高频率方式到达，在消息处理的时候，有可能发现队列中不止一条消息，通常会连续处理。反过来，也可能发现没有消息，因为被上一次触发起来的应用程序处理完了。这个时候，就要求应用程序有一定的灵活性，具有触发一次处理多条消息的能力，在触发后未发现相应消息也无需报错。

如果队列中每一条消息的对应处理程序是相同的，为了追求效率，一个通用的做法如下：用自己编写的触发监控器从初始化队列中连续读触发消息直到空，然后启动 Trigger 程序。Trigger 程序可以用 Wait 方式读取消息并进行连续处理，直到在一段时间内读不到新来的消息。

6.2.2.3 DEPTH

在队列深度达到一定值 (TRIGDPTH) 后触发，每次触发后队列管理器会自动关闭队列的触发机制，即将队列的 TRIGGER (TriggerControl) 属性变成 NOTRIGGER。所以，在应用程序中用 MQSET 将其复位或通过 MQSC alter queue 命令将触发开关重新打开。在计算队列深度的时候，消息的优先级必须大于等于 TRIGMPRI。TriggerDepth = 1 时，DEPTH 与 EVERY 效果相同。

6.2.3 配置

以 EVERY 触发方式为例，每到达一条消息，启动 Win2000 记事本程序。

QM:

DEFINE	QLOCAL	(QL_QM1)	+
	TRIGGER		+
	TRIGTYPE	(EVERY)	+
	PROCESS	(P_NOTEPAD)	+
	INITQ	(SYSTEM.DEFAULT.INITIATION.QUEUE)	+
	REPLACE		
DEFINE	PROCESS	(P_NOTEPAD)	+
	APPLICID	('C:/WINNT/system32/notepad.exe')	+

MQLONG	ApplType;	应用程序平台，即进程的 APPLTYPE 属性
MQCHAR256	ApplId;	应用程序名，即进程的 APPLICID 属性
MQCHAR128	EnvData;	环境数据，即进程的 ENVRDATA 属性
MQCHAR128	UserData;	用户数据，即进程的 USERDATA 属性
MQCHAR48	QMgrName;	队列管理器名

```
};
```

缺省的触发监控器 (runmqtrm) 会启动触发程序 (由 MQTM2.ApplId 指定)，并将 MQTMC2 结构以命令行参数的方式传给它，传入命令行参数 argv[1]。对于触发程序而言，从命令行参数收到 MQTMC2 结构后，可以按照约定读取并处理应用消息。触发过程结束。

6.2.5 并发

对于 FIRST 和 DEPTH 触发方式，触发程序一般会以 INPUT 方式打开本地队列，读取并处理应用消息。在触发程序 MQCLOSE 之前，可能会有新的消息到达，这些消息无法生成新的触发消息。也就是说，在 MQOPEN (MQOO_INPUT_*) 到 MQCLOSE 之间，队列上的 FIRST 和 DEPTH 触发机制是暂时失效的。所以，对于 FIRST 和 DEPTH 这两种触发方式，是不可能触发起多个应用程序来并行工作的，这也就是为什么触发起来的应用程序通常要连续工作直到将队列取空。

一个初始化队列上可以有多个 Trigger Monitor 在读触发消息，即对该初始化队列以读方式打开的句柄数 (OpenInputCount) 可以大于 1。这时，这些 Trigger Monitor 将会轮流读到触发消息。但是，通常来说，使用触发的应用消息到达的频度应该是比较低的，一个 Trigger Monitor 是够用的，多个 Trigger Monitor 也不会提高性能。

如果大消息在放入队列时被自动拆分成多个段 (Segment)，则在 MQPUT 时只会触发一次。但是一旦消息路由到远端目标队列，它们将被视为多条物理消息而在远端触发多次。

在 WMQ for z/OS 中，Non-Shared Local Queue 在 DEPTH 方式触发计算消息数量的时候，无论消息是否提交都被计算在内。所以被触发起来的程序有可能没有消息可读，因为有消息尚未提交。这种情况下，建议在 MQGET 的时候使用 WaitInterval 选项，使得读消息等一段时间。Shared Local Queue 在计算消息数量的时候，只计算提交后的消息。

6.2.6 通道触发

通道触发是触发应用的一种高级使用技巧，我们在传输队列上配置触发功能，使消息到达后靠触发过程将通道自动连接，将消息传送到对方。如果一段时间内没有消息需要传送，通道连接又会自动断开。在配置通道触发时，需要注意以下几点：

1. 可以用进程定义的 USERDATA 也可以用队列的 TRIGDATA 属性指定要触发的通道
2. 通常用 FIRST 方式触发，一段时间内的第一消息自动触发连接
3. 在缺省情况下，初始化队列为 SYSTEM.CHANNEL.INITQ。当然也可以使用其它的初始化队列，那样就需要手工 runmqchi。
4. 触发应该设置在通道主动方，如 SENDER、REQUESTER。可以设定 DISCINT 在一断时间后自动断连，重新处于待触发状态。

下面是具体的配置脚本实例：(每次第一条消息触发通道，隔 10 秒钟通道自动断开)

QM1:

```
DEFINE QREMOTE      (QR_QM2)          +
      RNAME         (QL_QM2)          +
      RQMNAME       (QM2)             +
      XMITQ         (QX_QM2)          +
      REPLACE
DEFINE QLOCAL       (QX_QM2)          +
      USAGE         (XMITQ)           +
      TRIGGER       +
      TRIGTYPE      (FIRST)           +
      PROCESS       (P_QM1.QM2)       +
      INITQ         (SYSTEM.CHANNEL.INITQ) +
      REPLACE
DEFINE PROCESS      (P_QM1.QM2)       +
      USERDATA      (C_QM1.QM2)       +
      REPLACE
DEFINE CHANNEL      (C_QM1.QM2)       +
      CHLTYPE       (SDR)             +
      TRPTYPE       (TCP)             +
      CONNAME       ('127.0.0.1 (1415)') +
      XMITQ         (QX_QM2)          +
      DISCINT       (10)              +
      REPLACE
```

QM2:

```
DEFINE QLOCAL       (QL_QM2)          +
      REPLACE
DEFINE CHANNEL      (C_QM1.QM2)       +
      CHLTYPE       (RCVR)            +
      TRPTYPE       (TCP)             +
      REPLACE
```

也可以是：

QM1:

```
DEFINE QREMOTE      (QR_QM2)          +
      RNAME         (QL_QM2)          +
      RQMNAME       (QM2)             +
      XMITQ         (QX_QM2)          +
      REPLACE
```



```

DEFINE QLOCAL      (QX_QM2)          +
      USAGE        (XMITQ)           +
      TRIGGER       +                 +
      TRIGTYPE      (FIRST)           +
      TRIGDATA      (C_QM1.QM2)       +
      INITQ         (SYSTEM.CHANNEL.INITQ) +
      REPLACE
DEFINE CHANNEL      (C_QM1.QM2)       +
      CHLTYPE       (SDR)             +
      TRPTYPE       (TCP)             +
      CONNAME       ('127.0.0.1 (1415)') +
      XMITQ         (QX_QM2)          +
      DISCINT       (10)              +
      REPLACE

```

QM2:

同上

触发通道是靠通道初始程序 (Channel Initiator) 完成的，它相当于一个 Trigger Monitor，监听相应的初始化队列，一旦有消息在传输队列上等待传送，该程序会从初始化队列中读到触发消息，自动启动通道。届时，发送端就会启动 runmqchl 进程。事实上，这样的程序用户也可以自己开发。系统提供了缺省的 Channel Initiator — runmqchi，它缺省监听在初始化队列 SYSTEM.CHANNEL.INITQ 上，在 strmqm 启动队列管理器时自动跟随启动。如果要自行指定触发通道的初始化队列，则需手工启动 runmqchi。

```
runmqchi [-q InitQueue] [-m QMgrName]
```

6.2.7 触发 CICS 交易

CICS 是 IBM 的交易中间件，它可以与消息中间件 WebSphere MQ 联合使用，通过配置使 MQ 消息触发 CICS 交易。触发 CICS 交易也是一种高级的触发使用技巧，本质上是将触发监控器嵌入到 CICS 运行环境中，当满足触发条件后，从触发监控器启动 CICS 交易。在配置时，需要注意以下几点：

1. 使用缺省初始化队列：SYSTEM.DEFAULT.INITIATION.QUEUE
2. THLO 是 CICS 交易名
3. 要将 /usr/lpp/mqm/bin/amqltmc0 作为一个 CICS 交易 (如 TMQM) 加入 CICS，再用 EXEC CICS START TRANSID (TMQM) 将其常驻 cicsas
4. amqltmc0 只监视缺省队列管理器的 SYSTEM.DEFAULT.INITIATION.QUEUE，如果是自己写的 Trigger Monitor 则可以自行指定监视的队列管理器和队列。
5. 对于 UNIX，要将 cics 用户加入 mqm 组中，使其具有相应的权限。

QM:

```

DEFINE QLOCAL      (Q)              +

```

```

TRIGGER                                     +
TRIGTYPE      (EVERY)                      +
PROCESS       (P)                          +
INITQ         (SYSTEM.CICS.INITIATION.QUEUE) +
REPLACE
DEFINE PROCESS      (P)                    +
APPLTYPE         (CICS)                   +
APPLICID         (THLO)                   +
REPLACE

```

CICS:

```

cicsadd -c pd -r $(REGION_NAME) -P PMQM \
      PathName=/usr/lpp/mqm/bin/amqltmc0 \
      ProgType=program RSLKey=public
cicsadd -c td -r $(REGION_NAME) -P TMQM ProgName=PMQM

```

6.3 报告 (Report)

WebSphere MQ 的另一个特色，就是自动报告 (Report) 功能。在消息头 MQMD 的 Report 域中设置不同的报告选项，在一定的条件满足时，WebSphere MQ 会自动产生报告消息，放入消息的应答队列中，消息应答队列由消息的 MQMD.ReplyToQMgr 和 MQMD.ReplyToQ 指定。

6.3.1 原理

为了更好地理解报告消息，让我们先来了解一下报告消息产生的过程，如图 6-6。

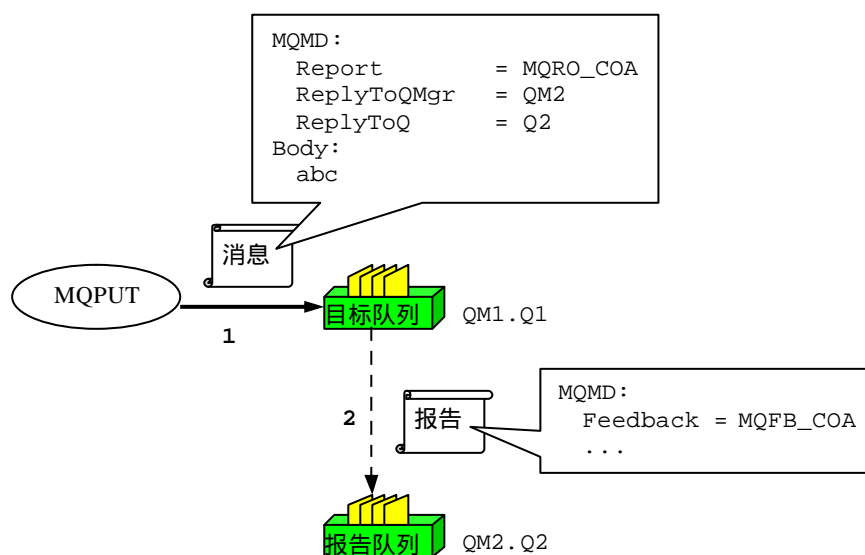


图 6-6 消息与报告

应用程序在发送消息 (MQPUT/MQPUT1) 时可以在 MQMD 的 Report 域设置报告选项，

即在怎样的条件下自动产生报告消息，在 MQMD 的 ReplyToQMGr 和 ReplyToQ 域中设置报告队列的位置，即报告消息的应该送去哪里。当发送的消息满足报告选项指定的条件时，WebSphere MQ 会自动产生报告消息并送去报告队列。报告消息的 Feedback 域与原消息的 Report 域对应。

由 ReplyToQMGr 和 ReplyToQ 指定的应答队列也可以是远程队列，这样可以将报告消息传送到另一个队列管理器中，实现远程监控。如果网络中所有的报告消息都指向同一个目标队列，则方便地实现了集中式远程监控。

6.3.2 选项

报告消息可以只有一个消息头 MQMD，也可以在消息头后含原消息内容的前 100 个字节，还可以在消息头后含原消息的全部内容。报告选项如下：

MQRO_EXCEPTION	原消息异常，报告只含 MQMD
MQRO_EXCEPTION_WITH_DATA	原消息异常，报告含原消息内容的前 100 个字节
MQRO_EXCEPTION_WITH_FULL_DATA	原消息异常，报告含原消息全部内容
MQRO_EXPIRATION	原消息超时，报告只含 MQMD
MQRO_EXPIRATION_WITH_DATA	原消息超时，报告含原消息内容的前 100 个字节
MQRO_EXPIRATION_WITH_FULL_DATA	原消息超时，报告含原消息全部内容
MQRO_COA	原消息到达目标队列，报告只含 MQMD
MQRO_COA_WITH_DATA	原消息到达目标队列，报告含原消息内容的前 100 个字节
MQRO_COA_WITH_FULL_DATA	原消息到达目标队列，报告含原消息全部内容
MQRO_COD	原消息被取走，报告只含 MQMD
MQRO_COD_WITH_DATA	原消息被取走，报告含原消息内容的前 100 个字节
MQRO_COD_WITH_FULL_DATA	原消息被取走，报告含原消息全部内容
MQRO_PAN	原消息处理成功应答，报告只含 MQMD
MQRO_NAN	原消息处理失败应答，报告只含 MQMD
MQRO_NEW_MSG_ID	报告消息可以只产生 Message ID
MQRO_PASS_MSG_ID	报告消息沿用原消息的 Message ID
MQRO_COPY_MSG_ID_TO_CORREL_ID	报告消息用原消息的 Message ID 作为 Correl ID
MQRO_PASS_CORREL_ID	报告消息用原消息的 Correl ID 作为 Correl ID
MQRO_DEAD_LETTER_Q	如果原消息无法到达目标队列，则将其放入死信队列
MQRO_DISCARD_MSG	如果原消息无法到达目标队列，则将其丢弃

在报告消息的 MQMD 中的 Feedback 域表示报告消息产生的原因，取值如下：

MQFB_EXPIRATION	因为原消息超时
MQFB_COA	因为原消息到达目标队列
MQFB_COD	因为原消息被取走
MQFB_PAN	因为原消息处理成功
MQFB_NAN	因为原消息处理失败

通过为消息设置报告选项 (MQMD.Report)，我们可以指定的条件发生时得到报告消息。通过分析报告消息的回执 (MQMD.Feedback)，我们可以知道报告消息产生的原因，由此程序可以做出正确的反应。

6.3.3 说明

6.3.3.1 消息异常 (Exception)

所谓异常，指的是消息因为某种原因未能到达目标队列，这可能是消息自身的问题，也可能是 WebSphere MQ 运行环境的问题，如队列满、通道断等等。消息在 MQPUT/MQPUT1 时指定 MQMD.Report = MQRO_EXCEPTION_*

在消息经过通道传送时，如果是持久 (Persistent) 消息，或者是非持久 (Non-Persistent) 消息通过普通 (Normal) 通道传送的，则只有按照指定动作妥善安置消息后才会发出 Exception Report。这里的指定动作有两种：

MQRO_DISCARD_MSG	消息被直接扔掉
MQRO_DEAD_LETTER_Q	消息被安置在目标队列管理器的死信队列里，当然目标队列管理器要设定死信队列。缺省动作。

如果非持久 (Non-Persistent) 消息通过高速 (Fast-Path) 通道传送，则无论指定动作是否完成，队列管理器都会发出 Exception Report。通道的缺省属性是高速的。

在消息放入目标队列时，如果发现队列不存在或队列满等原因无法落地，则根据报告选项产生相应的报告消息。MQRO_DISCARD_MSG 通常与 MQRO_EXCEPTION_FULL_DATA 合用，从报告消息中可以得知被丢弃的原消息内容。

6.3.3.2 消息超时 (Expiration)

消息在传输途中或到达目标队列后超过消息自身的生命周期，被内部或外部的 MQGET 发现，这时如果消息属性 MQMD.Report = MQRO_EXPIRATION_*, 则自动产生消息超时报告。

在大多数应用设计中，超时意味着消息失效，通常不再继续处理。超时报告往往是用来通知消息发送方的，以便得知是哪一笔业务在处理过程中发生超时，并做一些补偿处理。如果消息发送方等待应答消息，则通常在 MQGET 中也会设等待超时，这时等待超时应该小于消息超时。

通常，消息超时意味着系统异常，比如网络断开、数据库操作互锁、消息处理挂起、应用进程非正常退出等等。也可能是性能问题，比如处理执行时间太长、队列中消息堆积严重、机器负载过重、系统资源耗尽等等。所以，对于消息超时需要重视并及时处理。

6.3.3.3 送达确认 (Confirm-On-Arrival, COA)

消息成功到达目标队列并放入其中，这时如果消息属性 MQMD.Report = MQRO_COA_*, 则自动产生消息送达确认报告。注意，送达确认意味着消息在队列中等待处理，但并不一定会被处理。

COA 报告是消息传达的确认回执，表示消息成功地传达对方。通常设置这个选项是对网络传送或路由过程有所担心，送达确认 (COA) 可以和消息超时 (Expiration) 一起使用，回执可能是两者中的任何一个。当然如果回路不通，则可能收不到任何回执。

6.3.3.4 交付确认 (Confirm-On-Delivery, COD)

消息被 MQGET 成功地取走，这时如果消息属性 MQMD.Report = MQRO_COD_*, 则自动产生交付确认报告。注意这里的 MQGET 不能是 BROWSE 方式，也就是说，如果是浏览消息内容，则不会产生交付确认报告，只有消息被取走时才会产生。

COD 报告是消息被处理的确认回执，表示消息已经开始处理。通常设置这个选项是对消息接收方的应用处理不放心，担心对方宣称未收到或未处理。

6.3.3.5 PAN 和 NAN (Positive/Negative Action Notification)

PAN 和 NAN 表示消息处理是否成功。与其它报告选项不同，报告消息不由 WebSphere 它 MQ 自动生成，而是由应用程序主动生成并发送的。设置 PAN 或 NAN，只是希望读取原消息的应用程序能主动生成通知消息，报告消息处理是否顺利，但严格地说，对应用程序并没有约束力。

另外，通过报告选项的设置，可以指定报告消息的 MQMD.MsgId 和 MQMD.CorrelId 与原消息的关系，进而可以匹配地读取相应的报告消息。对于由队列管理器或 MCA 自动生成的消息 (Exception、Expiration、COA、COD)，这种设定是有效的。对于应用程序生成的消息 (PAN、NAN)，这种设定只是希望应用这么做。对于应用程序，则应该接受这种指导意见。

报告 (Report) 消息是针对物理消息而言的，如果原消息是分组或分段的，则报告消息也是分组或分段的，并结构与原消息一致。报告消息的 MQMD.OriginalLength 指明了原物理消息的长度。

6.4 分组与分段 (Group & Segment)

在 WebSphere MQ 中，如果应用中确实有大量数据的发送的需求，从而使单条消息可能过大，这时可以考虑消息分组与分段。消息分组即将一批消息放入同一消息组，用 MQMD.GroupId 来唯一标识这组 (Group) 消息，而组中的每一部分称为逻辑消息 (Logical Message)，用 MQMD.MsgSeqNumber 来表示逻辑消息在组中的排序，从 1 开始记。消息分段则比分组更低一层，每个逻辑消息可以再分成多个段 (Segment)，用 MQMD.Offset 来表示各段在逻辑消息中相对起始位置的偏移量。最终形成的每一个消息单位称为物理消息 (Physical Message)。图 6-7 显示了分组与分段中逻辑消息和特殊消息的关系。

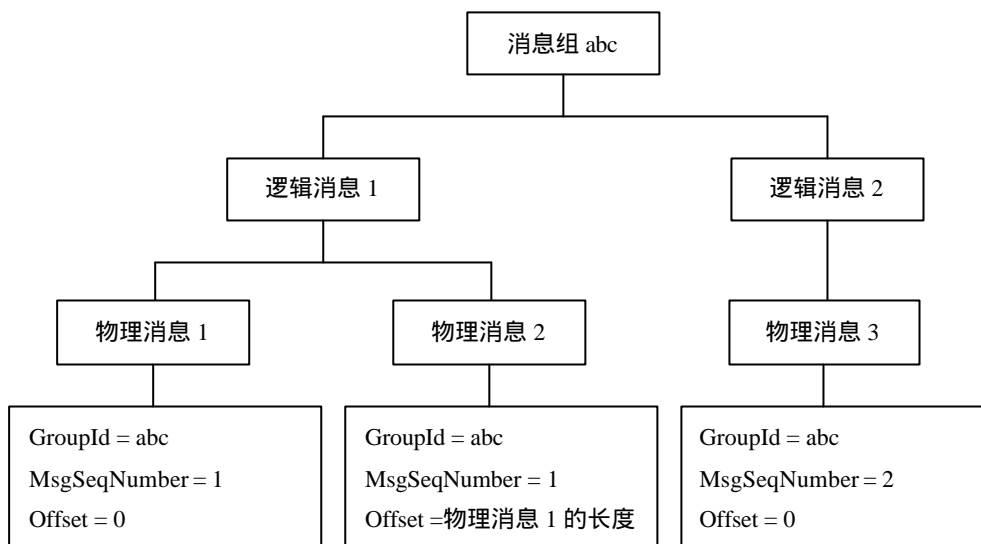


图 6-7 逻辑消息与物理消息

组中的每一条物理消息与普通的独立消息是一样的，有消息头 MQMD 和消息体，可以有自己的 MsgId 和 CorrelId，在消息的传送和处理过程中完全等同于一条独立的消息。事实上，独立消息可以看作是组消息的一种退化方式，即只含一个逻辑消息和一个物理消息。消息可以分组不分段，也可以分段不分组，还可以既分组又分段。具体采用何种方式应该视应用需求而定。WebSphere MQ for z/OS 不支持分段。

6.4.1 消息组的发送

消息组中物理消息在发送前除了普通的消息属性设置外还需要设置以下内容：

- MQMD.Version 消息分组和分段是新版的 WebSphere MQ 所支持的功能，该值应该不小于 MQMD_VERSION_2
- MQMD.MsgFlags
 - MQMF_SEGMENTATION_INHIBITED 不允许分段。如果消息本身已经是分段消息，则不允许消息被分成更小的段。
 - MQMF_SEGMENTATION_ALLOWED 允许分段。队列管理器在传送消息的途中可以根据需要将其分成更小的段。
 - MQMF_MSG_IN_GROUP 消息是组的一个成员
 - MQMF_LAST_MSG_IN_GROUP 消息是组中最后一个逻辑消息
 - MQMF_SEGMENT 消息是逻辑消息的一段
 - MQMF_LAST_SEGMENT 消息是逻辑消息的最后一段
 - MQMF_NONE 不允许分组，也不允许分段
- MQMD.GroupId 同一组消息应该由唯一的共同的 GroupId 来标识
- MQMD.MsgSeqNumber 从 1 开始记，逻辑消息在消息组中的位置
- MQMD.Offset 从 0 开始记，段在逻辑消息中的偏移量
- MQPMO.Options
 - MQPMO_LOGICAL_ORDER 按 MQPUT 的次序自动记数
 - MQPMO_SYNCPOINT 消息组参与交易

这里，分组消息的 MsgSeqNumber 和 Offset 可以手工设置，也可以自动设置。手工设置时务必将 MsgFlags 配合设置成相应的值，例图 6-7 中的消息设置可以是：

1. MsgFlags = MQMF_MSG_IN_GROUP + MQMF_SEGMENT
GroupId = abc, MsgSeqNumber = 1, Offset = 0
2. MsgFlags = MQMF_MSG_IN_GROUP + MQMF_SEGMENT + MQMF_LAST_SEGMENT
GroupId = abc, MsgSeqNumber = 1, Offset = 100 (假定上一条物理消息的长度为 100)
3. MsgFlags = MQMF_MSG_IN_GROUP + MQMF_LAST_MSG_IN_GROUP + MQMF_SEGMENT + MQMF_LAST_SEGMENT
GroupId = abc, MsgSeqNumber = 2, Offset = 0

也可以是：

1. MsgFlags = MQMF_MSG_IN_GROUP + MQMF_SEGMENT
GroupId = abc, MsgSeqNumber = 1, Offset = 0
2. MsgFlags = MQMF_MSG_IN_GROUP + MQMF_SEGMENT + MQMF_LAST_SEGMENT
GroupId = abc, MsgSeqNumber = 1, Offset = 100 (假定上一条物理消息的长度为 100)
3. MsgFlags = MQMF_MSG_IN_GROUP + MQMF_LAST_MSG_IN_GROUP
GroupId = abc, MsgSeqNumber = 2

手工设置可以有最大的灵活性，逻辑消息和物理消息的发送次序可以是乱的，只要最终形成完整的消息组即可。适合某些不记较次序的批量发送应用，比如多线程拆包发送。

自动设置比较简单，将 MQPMO.Options 设置成 MQPMO_LOGICAL_ORDER，以后每次 MQPUT 会根据 MQMD.MsgFlags 来自动记数，并在返回时更新 MQMD 参数中相关域，应用可以复用 MQMD 进行下一次 MQPUT。比如 MsgFlags = MQMF_MSG_IN_GROUP + MQMF_SEGMENT，则 MsgSeqNumber 不变，Offset 自动计算出相对偏移量。MsgFlags = MQMF_MSG_IN_GROUP + MQMF_LAST_SEGMENT，则意味着一个逻辑消息的结束，MsgSeqNumber 自动加一，Offset 清零。自动设置使应用程序不需要再关心 GroupId, MsgSeqNumber, Offset 的设置，但消息的发送次序必须按照“逻辑次序”，即逻辑消息是按顺序的，其中的段也是按顺序的。

6.4.2 消息组的接收

通常情况下，一次 MQGET 可以接收消息组中一条物理消息。在返回时消息自身的 MQMD 会表明消息在组中的位置，MQGMO 参数也可以表明这一点：

- MQGMO.GroupStatus
 - MQGS_NO_IN_GROUP 消息不属于一个组，独立消息
 - MQGS_MSG_IN_GROUP 消息是组的一个成员
 - MQGS_LAST_MSG_IN_GROUP 消息是组中的最后一个逻辑消息
- MQGMO.SegmentStatus
 - MQSS_NO_A_SEGMENT 消息不是一个分段消息
 - MQSS_SEGMENT 消息是逻辑消息的一段
 - MQSS_LAST_SEGMENT 消息是逻辑消息的最后一段
- MQGMO.Segmentation
 - MQSEG_INHIBITED 消息不允许分段
 - MQSEG_ALLOWED 消息允许分段

应用程序也可以在 MQGET 时指定接收特定的消息：

- MQGMO.MatchOptions
 - MQMO_MATCH_GROUP_ID 接收时匹配 GroupId，需要设定 MQMD.GroupId
 - MQMO_MATCH_MSG_SEQ_NUMBER 接收时匹配 MsgSeqNumber，需要设定 MQMD.MsgSeqNumber
 - MQMO_MATCH_OFFSET 接收时匹配 Offset，需要设定 MQMD.Offset

应用程序还可以有其它功能设定：

- MQGMO.Options
 - MQGMO_LOGICAL_ORDER MQGET 会按照“逻辑次序”取消息
 - MQGMO_COMPLETE_MSG 一次取出逻辑消息中所有的段
 - MQGMO_ALL_MSGS_AVAILABLE 要求消息组完整地到达后，才能被 MQGET 取出消息
 - MQGMO_ALL_SEGMENTS_AVAILABLE 要求逻辑消息中所有段完整地到达后，才能被 MQGET 取出
 - MQGMO_SYNCPOINT 消息组参与交易

这里，MQGMO_COMPLETE_MSG 可以将多个段一次取出并拼成一条完整的消息。每一段物理消息在队列中有各自的消息头 MQMD 和消息体，拼成的消息使用第一段的 MQMD 作为消息头，消息体为所有段消息体的内容顺序拼接。比如：三段物理消息内容分别为 a、b、c。一次取出后内容为 abc。

批量消息在经过复杂路由的长途跋涉后，有可能出现后发先至的情况，这时用 MQGMO_LOGICAL_ORDER 可以在 MQGET 消息的时候按“逻辑次序”读取并处理消息。而 MQGMO_ALL_MSG_AVAILABLE 和 MQGMO_ALL_SEGMENTS_AVAILABLE 则是为了保证消息在完整到达后才被处理。

另外，消息的触发 (Trigger) 和报告 (Report) 机制都是针对物理消息的，所以，对于批量分组消息 WebSphere MQ 可能会对应地产生批量触发消息和报告消息。这些消息本身也是分组和分段的，其结构与送出的消息相同，且 GroupId 与原来的消息相同。所以，在 MQGET 接收这些消息的时候只要指定 MQGMO_COMPLETE_MSG + MQGMO_ALL_MSGS_AVAILABLE + MQGMO_ALL_SEGMENTS_AVAILABLE 就可以对整个消息组进行一次性事件触发或完整地确认。当然，对于触发而言，需要自己编写触发监控器 (Trigger Monitor)，对于报告而言，在接收报告时需要指定上述的参数选项。

6.5消息上下文 (Message Context)

一般说来，任意两条消息之间的属性是独立无关的，每条消息都有各自的属性。比如，在缺省情况下，队列中的一条消息被 MQGET 取出来，然后又原封不动地 MQPUT 放回去。虽然在内容上没有任何地改动，然而在一进一出之后，就变成了两条消息。严格地说，放回去的消息已经不是原先在队列中的那条消息了，它们可以有各自的属性，从后一条消息中是看不出前一条消息的痕迹的。

缺省情况下，消息在经历了 MQGET 和 MQPUT 这么一个环节之后就“脱胎换骨”成了另一条消息。对于由多环节组成的应用，下一个环节只能知道消息的来源是上一个环节，

而无法知道消息是由上一个环节产生,还是由上一个环节转发,更无法知道上一个环节之前的发送者。(如图 6-8) 这样消息链中的每一个环节只能对上一环节对安全检查和权限认证,从而决定是否让消息通过,缺乏能在消息链全程中发挥作用的安全机制。消息链的末端也只能看见最后一个环节,根本无法知晓消息的始作俑者,从而给末端授权或末端计费的应用带来一定的困难。

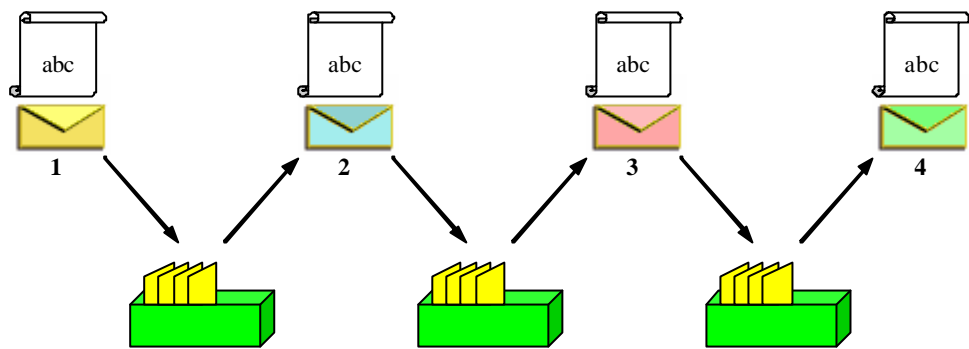


图 6-8 消息在传递的过程中未保持上下文

有时候,在实际的应用场景中需要消息的某些属性能够经历多次传递而不变。这就意味着某些消息属性能够跨越一个环节,被自动复制到新的相关消息中去,从而被一环一环地传递下去,直到消息链的终点。(如图 6-9) 终点的接收程序可以通过这些属性知道消息链起始端的发送程序想要传递的某些信息,这些可以被自动地复制传递而贯穿始终的消息属性就是所谓的消息上下文 (Message Text)。

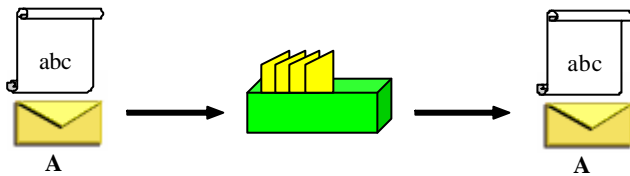


图 6-9 消息在传递的过程中保持了上下文

消息上下文指的是消息描述符 (MQMD) 中的 8 个字段,分两类:身份上下文 (Identity Context) 和 起源上下文 (Origin Context)。其中身份上下文指的是 MQPUT 或 MQPUT1 时发送程序的用户信息,通常存放消息链中最初的发送程序的用户信息;起源上下文指的是 MQPUT 或 MQPUT1 时发送程序自身的信息,通常存放上一个环节中 MQPUT 程序的信息。可见,如果传递身份上下文,则可以传递用户信息,如果传递起源上下文,则可以传递程序信息。

6.5.1 消息上下文的内容

事实上,WebSphere MQ 允许两种方式传递上下文:一种方式是只传递身份上下文,另一种方式是传递全部上下文,包括身份上下文和起源上下文。表 6-5。

表 6-5 消息中的上下文域

消息字段	说明
身份上下文 Identity Context	

<i>UserIdentifier</i>	MQPUT 程序的用户名。根据平台环境的不同而有不同的解释。
<i>AccountingToken</i>	应用程序标识或数字。通常用来对最初的发送程序计费。对于，Windows 平台，存放的是用户的 SID (Security Identifier)，这可以作为 <i>UserIdentifier</i> 的一个补充。
<i>ApplIdentityData</i>	可以用来传递辅助的用户信息，比如加了密的口令等等。

起源上下文 Origin Context

<i>PutApplType</i>	应用程序的类型，例：CICS 交易
<i>PutApplName</i>	应用程序的名字，例：交易名或作业名
<i>PutDate</i>	消息生成日期，GMT 格林威治时间
<i>PutTime</i>	消息生成时间，GMT 格林威治时间
<i>ApplOriginData</i>	起源消息数据

由于消息上下文具有贯串消息链的功能，所以通常用来：

- 对消息源的发送者进行权限检查
- 对消息源的发送者进行服务计费
- 对消息链全程记录

6.5.2 消息上下文的编程

要设置消息上下文是需要有相应的权限的。在使用时，如果应用程序是消息源，则应该设置 (SET) 身份上下文和起源上下文。如果应用程序是中间环节，未改变消息内容，则应该传递 (PASS) 身份上下文，也可以传递起源上下文。如果应用程序是中间环节，且改变了消息内容，则应该设置新的起源上下文。当然，这只是对编程的指导意见，编程本身并不限制对各种上下文域的设置和传递，你甚至可以用拷贝上下文内容的办法来设置上下文，效果看上去就像是传递上下文一样。应用程序对消息上下文的操作是通过 MQPUT 或 MQPUT1 中的 Options 域来完成的。如果不设置此域，则队列管理器会自动设置所有的消息上下文，如同使用 MQPMO_NO_CONTEXT。

在编程时可以用 MQPMO.Options 来设置或传递相应的上下文：

- MQPMO_SET_IDENTITY_CONTEXT
- MQPMO_SET_ALL_CONTEXT
- MQPMO_PASS_IDENTITY_CONTEXT
- MQPMO_PASS_ALL_CONTEXT

如果不设定合适的 Options，则相应的上下文域由队列管理器自动设置。这时，这些域对于 MQPUT 或 MQPUT1 来说，就只是输出域。

6.5.2.1 设置身份上下文 (Set Identity Context)

产生消息的第一个环节可以设置身份上下文，这些信息可以被后续环节传递下去。设置身份上下文时需要注意：

- 当 MQOPEN 队列打算写消息的时候，用 MQOO_SET_IDENTITY_CONTEXT 选项
- 当 MQPUT 时，MQPMO.Options 中设置 MQPMO_SET_IDENTITY_CONTEXT
- 根据需要，设置 MQMD 中的 UserIdentifier, AccountingToken, ApplIdentityData 域

6.7.2.2 设置所有上下文 (Set All Context)

产生消息的第一个环节也可以设置所有的上下文，这些信息可以被后续环节传递下去。设置所有上下文时需要注意：

- 当 MQOPEN 队列打算写消息的时候，用 MQOO_SET_ALL_CONTEXT 选项
- 当 MQPUT 时，MQPMO.Options 中设置 MQPMO_SET_ALL_CONTEXT
- 根据需要，设置 MQMD 中的 UserIdentifier, AccountingToken, ApplIdentityData 域以及 PutApplType, PutApplName, PutDate, PutTime, ApplOriginData 域

6.5.2.3 传递身份上下文 (Pass Identity Context)

消息传递的中间环节可以将身份上下文保持不变地传递下去，在具体实现时需要注意：

- 在 MQOPEN 队列打算读消息的时候，用 MQOO_SAVE_ALL_CONTEXT 选项，以后读到的消息会保存它的 Context (注意，对浏览程序，不可以用这一选项，也就是说，MQOO_SAVE_ALL_CONTEXT 和 MQOO_BROWSE 不能共用)
- 当 MQOPEN 队列打算写消息的时候，用 MQOO_PASS_IDENTITY_CONTEXT 选项
- 当 MQPUT 时，在 MQMD.Context 中设置读队列的句柄 (保存了上下文的那一个队列)，MQPMO.Options 中设置 MQPMO_PASS_IDENTITY_CONTEXT

6.5.2.4 传递所有上下文 (Pass All Context)

消息传递的中间环节也可以将所有上下文保持不变地传递下去，在具体实现时需要注意：

- 在 MQOPEN 队列打算读消息的时候，用 MQOO_SAVE_ALL_CONTEXT 选项，以后读到的消息会保存它的 Context (注意，对浏览程序，不可以用这一选项，也就是说，MQOO_SAVE_ALL_CONTEXT 和 MQOO_BROWSE 不能共用)
- 当 MQOPEN 队列打算写消息的时候，用 MQOO_PASS_ALL_CONTEXT 选项
- 当 MQPUT 时，在 MQMD.Context 中设置读队列的句柄 (保存了上下文的那一个队列)，MQPMO.Options 中设置 MQPMO_PASS_ALL_CONTEXT

6.6 死信处理 (DLQ Handler)

6.6.1 死信消息

持久性消息在无法送达目标队列时会放入就近的死信队列。在创建队列管理器时，系统的缺省对象中有一个死信队列 SYSTEM.DEAD.LETTER.QUEUE，但队列管理器的缺省设置 DEADQ 为空。也就是说，缺省情况下并没有启用这个死信队列，所以在应用设计时应该用 MQSC 的 ALTER QMGR 命令为每个队列管理器设置缺省死信队列。

死信队列本质上是一个本地队列，可以用 MQPUT 或 MQGET 对其操作。由队列管理器转发到死信队列上的消息会自动加上一个死信头 (MQDLH)。如果说普通消息由 MQMD +

Data 组成，那么死信消息则由 MQMD + MQDLH + Data 组成。

死信头结构如下：

```
struct tagMQDLH
{
    MQCHAR4      StrucId;          // “DLH”
    MQLONG       Version;         // MQDLH_VERSION_1
    MQLONG       Reason;          // 消息被放入死信队列的原因码
    MQCHAR48     DestQName;       // 目标队列
    MQCHAR48     DestQMGrName;    // 目标队列管理器
    MQLONG       Encoding;        // 死信头后继消息内容的编码方式
    MQLONG       CodedCharSetId;  // 死信头后继消息内容的字符集
    MQCHAR8      Format;          // 死信头后继消息内容的格式
    MQLONG       PutApplType;     // 将消息放入死信队列的应用程序类型
    MQCHAR28     PutApplName;     // 将消息放入死信队列的应用程序名。
                                // 可能是 WebSphere MQ 的系统程序，如 AMQRMPPA.EXE
    MQCHAR8      PutDate;         // 消息被放入死信队列的日期
    MQCHAR8      PutTime;        // 消息被放入死信队列的时间
};
```

6.6.2 死信队列处理器

死信消息的出现本身就是一种异常，说明有消息无法投递，作为一个健壮的应用系统必须对这种消息进行处理，及时地发现并解决问题。对死信消息进行处理的程序会管理死信队列，称为死信队列处理器 (Dead Letter Queue Handler)，它本质上就是一个 WebSphere MQ 应用程序。程序员可以自行设计并编写死信队列处理器，通过死信头的 Reason 域可以知道死信产生的原因，为诊断问题提供线索。通过 DestQName 和 DestQMGrName 域可以知道消息的目的地，可根据需要重发或转发。

WebSphere MQ 提供了一个缺省的死信队列处理器 (runmqdlq)，它是一个应用程序，监听指定队列管理器上的指定队列。从标准输入 (stdin) 上读取处理规则，读到 EOF 结束 (UNIX 中 EOF 为 Ctrl + D，Windows 中 EOF 为 Ctrl + Z)，结果输出到标准输出 (stdout) 上。通过对标准输入输出的重定向，它可以接受来自规则文件的输入。如：

```
runmqdlq [DLQName [QMGrName]]
runmqdlq SYSTEM.DEAD.LETTER.QUEUE QM < DLQ.rule
```

6.6.2.1 规则文件

死信队列处理器的规则文件由控制数据 (Control Data) 和规则 (Rules) 两部分组成。文件的第一个非空行 (Entry) 为 Control Data，以后的非空行 (Entry) 为 Rules。如同 MQSC 规则，每个 Entry 可以由多个输入行组成，用 + 或 - 作为前一输入行的最后一个非空字符，表示后继输入行与前一输入行同属一个 Entry。其中，+ 表示后继行从第一个非空字符开始。- 表示后继行从行首第一个字符开始。为了保持可移植性，建议每个输入行不要超过 72 个字符。

6.6.2.1.1 控制数据 (Control Data)

控制数据由多个参数和值组成 (表 6-6)

例：

```
INPUTQ (ABC1.DEAD.LETTER.QUEUE) INPUTQM (ABC1.QUEUE.MANAGER)
```

表 6-6 控制数据参数说明

参数	缺省值	可取值	说明
INPUTQ	空	空 QueueName	1. 如果在 runmqdlq 中指定了 DLQName 参数, 则覆盖 INPUTQ 值 2. 如果在 runmqdlq 中未指定 DLQName 参数, INPUTQ 值非空, 则取 INPUTQ 值 3. 如果在 runmqdlq 中未指定 DLQName 参数, INPUTQ 值为空, 且 runmqdlq 中指定了 QMgrName, 则取 QMgrName 上的死信队列 4. 如果在 runmqdlq 中未指定 DLQName 参数, INPUTQ 值为空, 且 runmqdlq 中未指定 QMgrName, 则取 INPUTQM 上的死信队列. 如果 INPUTQM 为空, 表示缺省队列管理器.
INPUTQM	空	空 QueueMgrName	1. 如果在 runmqdlq 中指定了 QMgrName, 则覆盖 INPUTQM 值 2. 如果在 runmqdlq 中未指定 QMgrName, 则使用 INPUTQM 值 3. INPUTQM 值为空表示使用缺省队列管理器
RETRYINT	60	Interval (秒)	隔一段时间, runmqdlq 会重试一次
WAIT	YES	YES NO Interval (秒)	runmqdlq 会一直等待新的消息到来, 即使死信队列为空 runmqdlq 在死信队列为空或没有可以处理的消息时自动退出 runmqdlq 在一段时间内没有消息可以处理, 则自动退出

6.6.2.1.2 规则 (Rules)

规则由多个匹配模式 (Pattern) 和处理动作 (Action) 组成。其中, Pattern 表示匹配条件, Action 表示处理。Pattern 和 Action 也都是由参数和值组成, 如：

```
PERSIST (MQPER_PERSISTENT) REASON (MQRC_PUT_INHIBITED) +  
ACTION (RETRY) RETRY (3)
```

● Pattern (表 6-7)

表 6-7 Pattern 参数说明

参数	缺省值	可取值	说明
APPLIDAT	*	*或 ApplIdentityData	匹配消息的 MQMD.ApplIdentityData
APPLNAME	*	*或 PutApplName	匹配消息的 MQMD.PutApplName, 即 MQPUT 或 MQPUT1 的应用名字
APPLTYPE	*	*或 PutApplType	匹配消息的 MQMD.PutApplType

DESTQ	*	*或 QueueName	匹配消息中的目标队列, MQDLH.DestQName
DESTQM	*	*或 QMgrName	匹配消息中的目标队列管理器, MQDLH.DestQMgrName
FEEDBACK	*	*或 Feedback	当消息的 MQMD.MsgType == MQFB_REPORT 时, Feedback 指定报告的方式, 如 Feedback == MQFB_COA
FORMAT	*	*或 Format	匹配消息 MQMD.Format, 即消息发送时的消息类型
MSGTYPE	*	*或 MsgType	指定消息类型, 如 MsgType == MQMT_REQUEST
PERSIST	*	*或 Persistence	指定消息的 Persistence 属性, 如 Persistence == MQPER_PERSISTENT
REASON	*	*或 ReasonCode	指定消息放入 DLQ 的原因, 如 ReasonCode == MQRC_Q_FULL
REPLYQ	*	*或 QueueName	匹配消息的 MQMD.ReplyToQ
REPLYQM	*	*或 QMgrName	匹配消息的 MQMD.ReplyToQMgr
USERID	*	*或 UserIdentifier	匹配消息的 MQMD.UserIdentifier

● Action (表 6-8)

表 6-8 Action 参数说明

参数	缺省值	可取值	说明
ACTION	无	DISCARD	将消息从 DLQ 中直接删除
		IGNORE	将消息留在 DLQ 中
		RETRY	如果对消息的处理不成功, 重试。后继 RETRY 指定次数, Data Control 中的 RETRYINT 指定时间间隔
		FWD	将消息移到其它队列上, 队列名由 FWDQ 指定
FWDQ	无	QueueName	指定转移的队列名
		&DESTQ	MQDLH.DestQName
		&REPLYQ	MQMD.ReplyToQ
			为了防止匹配到 MQMD.ReplyToQ 为空的消息 (这样的消息无法做转送处理), 可以在 Pattern 中指定 REPLYQ (?*), 则至少可以匹配到一个非空字符
FWDQM	空	空	指定 DLQ 所属的队列管理器
		QM grName	指定转移的队列管理器名
		&DESTQM	MQDLH.DestQMgrName
		&REPLYQM	MQMD.ReplyToQMgr
HEADER	YES	YES	针对 ACTION (FWD), 转移后的消息带着 MQDLH
		NO	针对 ACTION (FWD), 转移后的消息不带 MQDLH
PUTAUT	DEF	DEF	MQPUT 时用 DLQ Handler 做身份验证
		CTX	MQPUT 时用消息上下文中的 UserId 做身份验证
RETRY	1	RetryCount	可以取值 1 - 999,999,999, 重试的最大次数, 与 Control Data 中的 RETRYINT 配合使用。如果在重试期间 DLQ Handler 重启, 则重试计数归零。

6.6.2.2 规则文件举例

* runmqdlq 监视队列管理器 QM 中的队列 Q，即 Q 为死信队列

```
INPUTQ (Q) INPUTQM (QM)
```

* 凡是因为目标队列满被放入死信队列的消息，可以隔一段时间重试 (RETRYINT 参数，缺省为 60 秒)，共重试 5 次

```
REASON (MQRC_Q_FULL) ACTION (RETRY) RETRY (5)
```

* 凡是因为目标队列不允许 PUT 而被放入死信队列的消息，可以隔一段时间重试 (RETRYINT 参数，缺省为 60 秒)，共重试 5 次

```
REASON (MQRC_PUT_INHIBITED) ACTION (RETRY) RETRY (5)
```

* 凡是因为无法投递到队列管理器 QM1 而被放入死信队列的消息都被自动送往队列管理器 QM2，队列名与原来相同

```
DESTQM (QM1) ACTION (FWD) FWDQ (&DESTQ) FWDQM (QM2)
```

* 最后，无法匹配的消息都送往 REALLY.DEAD.QUEUE

```
ACTION (FWD) FWDQ (REALLY.DEAD.QUEUE) HEADER (YES)
```

6.6.2.3 运行模式

死信队列 (DLQ) 本身就是一个本地队列，应用程序可以用 MQPUT 放入任意的消息，该消息可以是一条普通消息，也可以是人为地模拟带死信头 (MQDLH) 的消息。但死信队列处理器只匹配带 MQDLH 的消息，对于不带 MQDLH 的消息，只能永远留在 DLQ 中。

死信队列处理器 (Handler) 以 MQOO_INPUT_AS_Q_DEF 方式打开死信队列 DLQ。一个 DLQ 可以有多个 Handler，死信消息被谁读走就由谁处理。为了不至混乱，所有的 Handler 应该使用相同的规则文件。但是，一般说来，DLQ 和 Handler 之间是一对一的关系。在死信队列处理器运行过程中修改规则文件对 Handler 无效，只有重启后改动才生效。

Handler 的工作过程是为每条消息从前到后扫描规则文件，如果发现匹配规则，则执行相应的处理动作。如果处理失败，则累计一次，处理下一条消息。过一段时间会重试这一条消息的处理。队列深度会影响 Handler 工作效率，Handler 会记录每一条在 DLQ 的消息，所以那些留在 DLQ 中又无法处理的消息，只会增加 Handler 的负担。另外，这样的消息越堆越多，DLQ 会被撑满。为了保证 Handler 的有效运行，应该把匹配率高的规则靠前，最后一条规则是无条件地将消息移出 DLQ，以减轻负担。例：

```
ACTION (FWD) FWDQ (REALLY.DEAD.QUEUE) HEADER (YES)
```

6.7 数据转换 (Data Convert)

WebSphere MQ 是可以跨平台跨网络协议进行数据传输的消息中间件，由于不同平台的数据表示是有差异的，比如整型数据的字长和高低字节安排不同，有些平台是 32 位，有些

是 64 位，有些平台中的整数表示是高字节在前，有些是低字节在前。再如浮点数的表示也可能不同，科学记数法的位数分配与精度都可能不同。另外，字符编码也可能不一样，就单字节而言，有的是 ASCII 码，有的是 EBCDIC 码。如果是多字节，情况则更加复杂，不同的字符集之间有些是可以互相转换的，有些则不行。由于网络传输的是比特 (bit) 流，所以在传输一侧的数据到了另一侧就可能无法读懂，这样就需要数据转换。

如果通信双方使用的编程语言本身可以屏蔽这种平台差异，比如 Java 编程，字符串用 Unicode 表示。由于 Java 程序在 Java 虚拟机中运行，语言本身对高低字节、整型字长等都做了严格定义，保证在所有的平台上运行是相同的。所以，在应用层面上就无须考虑这些问题了。如果编程语言（比如 C/C++）本身是可以看得到操作系统细节的，则运行代码可能会受到不同平台的影响。为了屏蔽这种平台差异，WebSphere MQ 对数据的类型进行了重新定义。比如用 MQLONG 表示整型，这就避免了 32 位或 64 位操作系统对 int 字长约定的差异。再用 MQCHAR 或 MQBYTE 表示单字节，MQCHAR48、MQBYTE32 等表示定长字符串。不用 float、double 等可能在不同平台表示上引起两义性的数据类型，所有的结构定义都是定长的。经过这样的定义封装，基本数据类型就只剩下 MQLONG、MQCHAR、MQBYTE 三种，而这在所有的平台上的字长都是相同的。

然而，相同的字长下仍然可能有不同的数据表示。为此，WebSphere MQ 引入了编码 (Encoding) 和字符集 (CodedCharSetId，又称 CCSID) 的概念。在每一条消息的消息头 (MQMD) 中都有 Encoding 和 CodedCharSetId 这两个域，分别表示后继的消息体的编码和字符集。

WebSphere MQ 中的消息通常由各种结构组成，每个结构都有 StrucId 和 Version 两个域，分别表示结构名和版本号。通常这两个域在结构的最前面，在应用程序读到这两个域后，前者可以帮助知道结构的长度和每个域的细节，后者可以知道结构中哪些域是起作用的。另外，每个消息头结构中都有 Format 域，表示后继的消息内容的类型，比如 MQFMT_STRING、MQFMT_EVENT 等等。如下：

```
MQMD
{
    MQCHAR4    StrucId;
    MQLONG     Version;
    MQCHAR8    Format;
    MQLONG     Encoding;
    MQLONG     CodedCharSetId;
}
```

关于 Encoding、CodedCharSetId 和 Format 三个域的设置说明如下：

- Encoding (消息内容编码方式)

MQENC_INTEGER_NORMAL	整型表示，高字节在低地址上
MQENC_INTEGER_REVERSED	整型表示，高字节在高地址上
MQENC_DECIMAL_NORMAL	压缩十进制表示，高字节在低地址上
MQENC_DECIMAL_REVERSED	压缩十进制表示，高字节在高地址上
MQENC_FLOAT_IEEE_NORMAL	浮点型表示，高字节在低地址上
MQENC_FLOAT_IEEE_REVERSED	浮点型表示，高字节在高地址上
MQENC_FLOAT_S390	浮点型表示，S390 专用方式

- CodedCharSetId (消息内容字符集)

MQCCSI_Q_MGR	MQPUT/MQPUT1 时 MQ 自动设置成队列管理器的 CCSID 值,当然手工设置成队列管理器的 CCSID 值也可以
MQCCSI_INHERIT	MQPUT/MQPUT1 时自动设置成与外层的消息头具有相同的 CCSID 值
MQCCSI_EMBEDDED	消息内容本身指定消息的字符集,不需要在 MQGET 时数据转换。用于多字符集混合的 PCF 消息

- Format (消息内容结构类型, WebSphere MQ 支持的内置格式)

MQFMT_ADMIN
MQFMT_CHANNEL_COMPLETED
MQFMT_CICS
MQFMT_COMMAND_1
MQFMT_COMMAND_2
MQFMT_DEAD_LETTER_HEADER
MQFMT_DIST_HEADER
MQFMT_EVENT
MQFMT_IMS
MQFMT_IMS_VAR_STRING
MQFMT_MD_EXTENSION
MQFMT_PCF
MQFMT_REF_MSG_HEADER
MQFMT_RF_HEADER
MQFMT_RF_HEADER_2
MQFMT_STRING
MQFMT_TRIGGER
MQFMT_WORK_INFO_HEADER
MQFMT_XMIT_Q_HEADER

有时候,消息多层嵌套后会出现多个消息头,这时消息头中的域只对自己下一层内容有效。如图 6-10, MQMD 表示以后的消息内容是 MQHRF 消息,使用平台编码方式,字符集与队列管理器相同。MQHRF 消息又含有 RFH 消息头和消息内容,消息内容是字串,同样使用平台编码方式,字符集继承于 MQMD 设定,即与队列管理器相同。

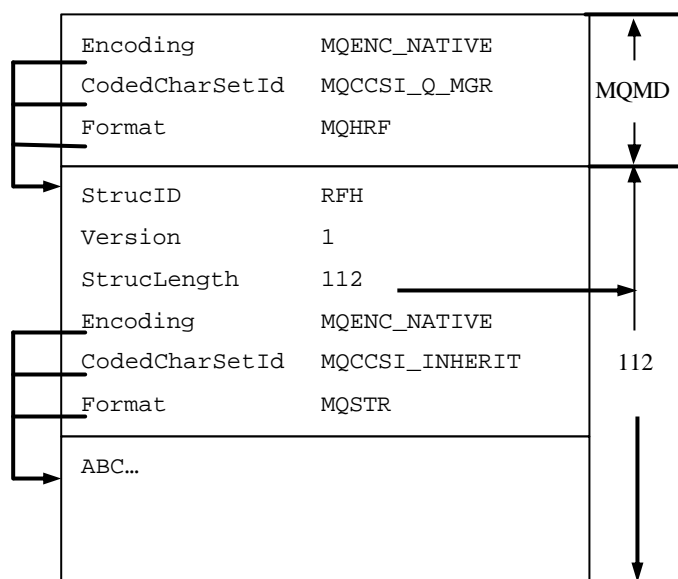


图 6-10 多消息头消息

6.7.1 转换方式

如果发送消息时的编码方式与字符集与接收消息时不同，则需要进行数据转换。

WebSphere MQ 的数据转换有三种方式：

1. 在接收端在 MQGET 时设置 MQGMO_CONVERT 选项
2. 在发送端的 MCA 的通道属性上设置 CONVERT (YES)
3. 使用 Data-Conversion Exit 用户出口

一般说来，首先推荐使用在接收端进行数据转换的方式，对于接收端不支持数据转换功能的平台，才考虑在发送端进行数据转换。如果有特殊的转换要求，比如应用自定的结构或代码集，才考虑使用 Data-Conversion Exit 用户出口。

6.7.1.1 接收端转换

带 MQGMO_CONVERT 选项的 MQGET 在以下所有的条件都满足的时候会自动进行数据转换：

1. 队列中消息的 MQMD.CodedCharSetId 或者 MQMD.Encoding 与 MQGET 参数中的 MQMD.CodedCharSetId 或者 MQMD.Encoding 不一致
2. 队列中消息的 MQMD.Format 不能是 MQFMT_NONE
3. MQGET 参数中的 BufferLength 不能是 0
4. 队列中消息的长度不能是 0
5. 队列管理器支持两个 CodedCharSetId 和 Encoding 之间的转换

6.7.1.2 发送端转换

WebSphere MQ 在建立通道连接的时候会交换双方队列管理器的 Encoding 和 CCSID 参数，如果不一致，则意味着需要在双方之间进行数据转换。若发送端通道属性设置为

CONVERT (YES) ,则发出端 MCA 传输消息的时候会按接收端的 Encoding 和 CCSID 进行数据转换,接收端收到的消息是已经转换好了的,可以直接使用。

6.7.1.3 用户出口转换

用户出口转换可以在发送端也可以在接收端。

在消息发送的时候,如果发送端队列属性设置为 CONVERT (YES),且消息的 MQMD.Format 是 WebSphere MQ 支持的内置格式,则调用发送端转换。如果是非内置格式,则调用用户出口转换。若不成功,消息放入死信队列,若再不成功,则待在传输队列中,通道关闭。

在消息接收的时候,如果接收端 MQGET 有 MQGMO_CONVERT 选项,且消息的 MQMD.Format 是 WebSphere MQ 支持的内置格式,则调用接收端转换。如果是非内置格式,则调用用户出口转换。若不成功,消息留在队列中,MQGET 报错 MQRC_NOT_CONVERTED。

在使用用户出口进行数据转换时,消息的 MQMD.Format 应该置为双方约定的字符串来标识消息内容的结构名,该字符串不超过 8 个字节,比如“MYFORMAT”。在数据转换时,如果发现这不是内置格式,则调用相应的出口程序。Windows 中该程序为 <InstallDir>\Exits\MYFORMAT.dll,UNIX 中为 /var/mqm/exits/MYFORMAT。

事实上,在编写这样的用户出口的时候,会首先定义通信双方使用的数据结构:

```
// MYFORMAT.h
typedef struct MyFormat
{
    MQLONG      l;
    MQCHAR      c;
    MQBYTE      b;
    MQLONG      la [3];      // MQLONG Array
    MQCHAR      ca [5];      // MQCHAR Array
    MQBYTE      ba [6];      // MQBYTE Array
} MYFORMAT;
```

经过 WebSphere MQ 数据转换用户出口辅助生成工具:crtmqcvx,生成对应的转换代码函数:

```
crtmqcvx MyFormat.h MyFormat.c

// MYFORMAT.c
MQLONG ConvertMyFormat(
    PMQBYTE *in_cursor,
    PMQBYTE *out_cursor,
    PMQBYTE in_lastbyte,
    PMQBYTE out_lastbyte,
    MQHCONN hConn,
    MQLONG  opts,
```

```

        MQLONG  MsgEncoding,
        MQLONG  ReqEncoding,
        MQLONG  MsgCCSID,
        MQLONG  ReqCCSID,
        MQLONG  CompCode,
        MQLONG  Reason)
{
    MQLONG ReturnCode = MQRC_NONE;
    ConvertLong(1); /* l */
    ConvertChar(1); /* c */
    ConvertByte(1); /* b */
    AlignLong();
    ConvertLong(3); /* la */
    ConvertChar(5); /* ca */
    ConvertByte(6); /* ba */
Fail:
    return(ReturnCode);
}

```

在出口程序中调用这个转换代码函数：

```

#include "mq_public.h"
#include "MyFormat.c"

#define MQFMT_MYFORMAT    "MYFORMAT"

void MQENTRY MQStart (
    PMQDXP  pDataConvExitParms,
    PMQMD   pMsgDesc,
    MQLONG  InBufferLength,
    PMQVOID pInBuffer,
    MQLONG  OutBufferLength,
    PMQVOID pOutBuffer)
{
    . . .
    RC = ConvertMyFormat (
        & pmqbInCursor,                // PMQBYTE * in_cursor
        & pmqbOutCursor,               // PMQBYTE * out_cursor
        (PMQBYTE) pInBuffer + InBufferLength - 1,    // PMQBYTE in_lastbyte
        (PMQBYTE) pOutBuffer + OutBufferLength - 1,  // PMQBYTE out_lastbyte
        pDataConvExitParms -> Hconn,        // MQHCONN hConn
        pDataConvExitParms -> AppOptions,    // MQLONG  opts
        pMsgDesc -> Encoding,               // MQLONG  MsgEncoding
        pDataConvExitParms -> Encoding,      // MQLONG  ReqEncoding
        pMsgDesc->CodedCharSetId,           // MQLONG  MsgCCSID
    );
}

```

```

        pDataConvExitParms -> CodedCharSetId,          // MQLONG ReqCCSID
        pDataConvExitParms -> CompCode,                // MQLONG CompCode
        pDataConvExitParms -> Reason);                // MQLONG Reason
    if (RC == MQRC_NONE || RC == MQRC_TRUNCATED_MSG_ACCEPTED)
        pDataConvExitParms -> ExitResponse = MQXDR_OK;
    else
        pDataConvExitParms -> ExitResponse = MQXDR_CONVERSION_FAILED;

    if ((RC == MQRC_NONE || RC == MQRC_TRUNCATED_MSG_ACCEPTED)
        && pMsgDesc -> MsgFlags | MQMF_SEGMENT == 0)
        pDataConvExitParms -> DataLength = pmqbOutCursor - pOutBuffer;

    if (RC == MQRC_NONE) // Normally is MQRC_NOT_CONVERTED, should be MQRC_NONE
    {
        pDataConvExitParms -> CompCode = MQCC_OK;
        pDataConvExitParms -> Reason = mqlReasonCode;
    }
    else if (RC != MQRC_TRUNCATED_MSG_ACCEPTED)
    {
        pDataConvExitParms -> CompCode = MQCC_WARNING;
        pDataConvExitParms -> Reason = mqlReasonCode;
    }
}

```

关于用户出口设置，可以参见“用户出口”相关章节，程序代码请参考书后例程。

6.7.2 数据转换表 (Convert Table)

每个队列管理器有自己的 CCSID 属性，表示能够处理的消息字符集。在涉及多语言环境和项目中，可能会在不同 CCSID 的队列管理器之间建立通道进行通信。这时，如果消息内容是 WebSphere MQ 内置格式，则这种转换会依赖于数据转换表。Windows 中，转换表在 <InstallDir>\conv\table 目录，UNIX 中，在 /var/mqm/conv/table 目录。

数据转换表通常以进行转换代码集编码冠名，比如 05650567.tbl，表示从 1381 转换到 1383。前者是 Window 中文平台缺省的队列管理器 CCSID，后者是 AIX 中文平台缺省的队列管理器 CCSID。0565, 0567 是 1381, 1383 的十六进制表示。通常数据转换表会成对出现，以支持双向转换，比如 05650567.tbl 和 05670565.tbl。

另外，在数据转换目录中有一个文件 ccsid.tbl，里面记录了平台支持的所有字符集，包括这些它们在双字节码 (DBCS)、单字节码 (SBCS)、编码方式等等参数。在不同 CCSID 之间建立通道，往往会因为 CCSID 的问题不成功，比如 819 与 1381。可以选取 ccsid.tbl 中相近的字符集。比如：

```
// ccsid.tbl
```

CCSID	BaseCCSID	DBCS	SBCS	Type	Enc	ACRI	Codeset
-------	-----------	------	------	------	-----	------	---------

1381	1381	1380	1115	3	2	6	IBM-1381
5477	1381	1380	1115	3	2	6	IBM-1381
1386	1386	1385	1114	3	2	5	IBM-1386
5482	1386	1385	1114	3	2	5	IBM-1386
936	936	928	903	3	2	3	IBM-936

Type: 1=SBCS 2=DBCS 3=Mixed

Encoding: 1=EBCDIC 2=ASCII 3=ISO 4=UCS-2 5=UTF-8 6=euc

在上表中不妨可以选取以下一些组合来设定队列管理器的 CCSID。

- 936 和 1381
- 1386 和 1381
- 5477 和 1381
- 5482 和 1381

在 ccsid.tbl 文件中可以约定缺省的转换字符集，比如：

default	0	500	1	1	0
default	0	850	1	2	0

表示单字节 EBCDIC 码用 500 字符集，单字节 ASCII 码用 850 字符集。

改动 ccsid.tbl，需要在通信双方都进行相同的改动，重启双方的队列管理器才生效。

如果消息在传输的过程中不需要队列管理器做任何数据转换工作，则 CCSID 与消息内容可以不一致。比如，传输双方的队列管理器 CCSID 都为 819 (英文)，消息的 MQMD.Format = MQFMT_STRING，MQMD.Encoding = MQENC_NATIVE，MQMD.CodedCharSetId = MQCCSI_Q_MGR (消息放入后即 819)。这时并不妨碍消息内容是 1381 中文字串。所以，在开放平台的项目实施时，如果消息内容只是中英文字符串，不妨将所有的队列管理器 CCSID 设置成 819。

改动队列管理器的 CCSID 对已经建立的通道连接无效，对已经连接在队列管理器上的应用也无效。所以，修改队列管理器的 CCSID 后，通常需要重启队列管理器，以保证改动对所有的部件生效。另外，在 Windows 中，WebSphere MQ 资源管理器在连接到队列管理器后会记住其 CCSID，所以改动 CCSID 后，刷新连接会发生问题。需要重启相应的命令服务器和资源管理器。

WebSphere MQ 中产生的报告消息 (Report) 中的 MQMD.Format 与原消息相同。如果原消息是通过用户出口转换的，反向的报告消息会需要反向转换。这一点在应用设计时要特别注意。

6.7.2.1 ASCII 码与 EBCDIC 码的转换设定

通过队列管理器代码集及转换方式的设定，WebSphere MQ 会自动在不同代码集之间进行转换。涉及 ASCII 码与 EBCDIC 码之间的转换时，对于大多数字符是没问题的，但由于 EBCDIC 码中独有的 NL (New Line) 字符是 ASCII 码字符集中没有的，所以需要特别指定。

- 在 UNIX 中，可以在 qms.ini 中的 AllQueueManagers 一段中指定：

```
ConvEBCDICNewline = NL_TO_LF | TABLE | ISO
```

- 在 Windows 中，可以修改操作系统系统注册表 `<MQ_HKEY>\Configuration\AllQueueManagers\ConvEBCDICNewLine`，也可以在“WebSphere MQ 服务”中点击右键选择属性，选择“所有队列管理器”，再选择“转换 EBCDIC 新行”。

<code>ConvEBCDICNewline = NL_TO_LF</code>	缺省值 将 EBCDIC NL 字符 (0x15) 转换成 ASCII LF (0x0a)。
<code>ConvEBCDICNewline = TABLE</code>	根据字符转换表决定 NL 字符的转换，字符转换表取决于各自操作系统的 CCSID
<code>ConvEBCDICNewLine = ISO</code>	对于 ISO CCSIDs，使用 TABLE 方式转换，对于其它的 CCSIDs，使用 NL_TO_CF 方式转换

7.1.2.2 常用中文字符集

WebSphere MQ 中的常用中文字符集与平台有关，如表 6-9。

表 6-9 常用中中文字符集			
平台	中文字符集	说明	
z/OS	935, 1388		
OS/400	935		
OS/2	1381, 1386		
AIX	1383, 1386		
HP-UX	1381	常用的 prc15 和 hp15CN 都属于 1381	
Windows	1381, 1386	操作系统用的 936 代码页属于 1386	
Solaris	1383		
Compaq-OVMS, DEC-OVMS, NSK, Tru64	1383		

第 7 章 广播通信

WebSphere MQ 擅长一对一的方式进行单点通信，如果要实现一对多的广播通信则需要考虑使用高级功能。通常我们有两种选择，分发列表和订阅/发布。本章我们将分别介绍采用这两种方式进行广播通信的概念、设计和实现。

7.1 分发列表 (Distribution List)

7.1.1 概念

通常情况下，MQPUT 或 MQPUT1 的操作对象是 WebSphere MQ 中的一个目标队列。然而，有的时候操作对象也可以是多个目标队列。在具体操作的时候，这些目标队列放在一个数组列表中，MQPUT/MQPUT1 操作针对这个列表一次完成，且可以在一个交易中进行。这里的列表就是分发列表 (Distribution List)。图 7-1。

分发列表中的队列可以是本地队列，也可以是远程队列。如果多个远程队列的目标队列管理器相同且支持分发列表，则消息在网络上只需要传送一次，等消息到达对方队列管理器后，自行分发至多个目标队列中。当然，队列管理器之间的通信环境（比如通道、监听器等）需要事先建立。

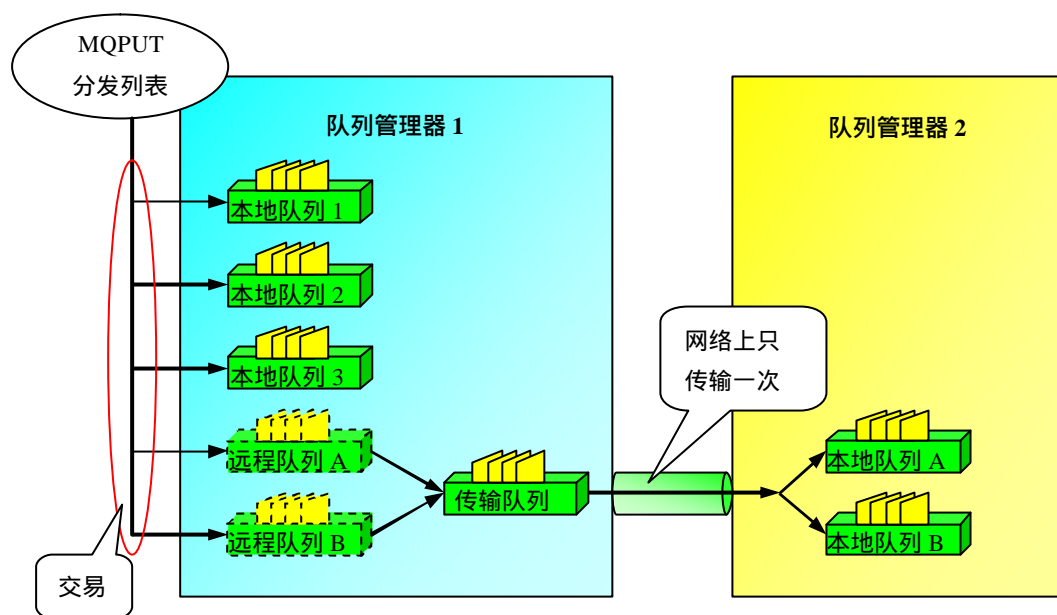


图 7-1 分发列表

由于分发列表功能并未被 WebSphere MQ 的所有平台产品所支持，例如 WebSphere MQ for z/OS 就不支持这一功能。所以，使用分发列表功能之前必须确认应用所在的队列管理器支持这种功能。这里的应用程序与队列管理器的连接可以是 Client 方式，也可以是 Binding 方式。另外，如果针对相同的远程队列管理器上的多个远程队列操作，而需要消息在网络上只传输一次，这需要目标队列管理器也支持分发列表功能才可以。支持分发列表功能的平台有：

- AIX
- HP-UX
- Solaris
- Windows
- Linux
- Compaq OpenVMS
- OS/2
- OS/400

队列管理器有 DISTL 属性，可以取值 YES 或 NO，表示是否支持分发列表功能。对于支持分发列表功能的队列管理器，该属性取值为 YES。该属性在队列管理器创建时就定下了，与生俱来，无法修改。

本地队列和模型队列也有 DISTL 属性，可以取值 YES 或 NO，对于普通本地队列，缺省值为 NO，对于传输队列，缺省值为 YES。这个属性是针对传输队列而言的，在消息通道建立后，MQA 会设定传输队列的 DISTL 属性，标志对方队列管理器是否支持分发列表功能。如表 7-1，如果该属性值为 YES，则在多个远程队列的目标队列管理器相同时，消息在网络

上只需要传送一次，到达对方后由目标队列管理器分发。如果该属性值为 NO，则发送消息的队列管理器会将消息复制并传送多次。

表 7-1 WebSphere MQ 对象中与分发列表相关的属性

对象	属性	可取值	说明
队列管理器	DISTL	● YES	静态属性，无法修改
		● NO	
队列 (Local/Model)	DISTL	● YES	针对传输队列，由 MCA 设定
		● NO	

队列管理器的 DISTL 属性是静态的，而传输队列的 DISTL 属性是 MCA 动态设定的。在程序中需要得到该属性值通常可以用 MQINQ 函数通过 MQIA_DIST_LISTS 选项进行查询，得到的值可能是 MQDL_SUPPORTED 或 MQDL_NOT_SUPPORTED，分别表示支持或不支持分发列表功能。

分发列表可以在交易环境中工作，消息会在多个队列中等待直到提交或回滚。这时如果有远程队列，则消息也会在相应的传输队列上等待提交或回滚指令，这时不会堵住传输队列其它的正常传送工作。分发列表也可以不在交易环境中工作，一旦调用 MQPUT 或 MQPUT1，消息立即发送到所有的队列中。消息在传送的过程中，某一路上的失败（比如目标队列满）不会影响其它路的传送。

7.1.2 配置举例

我们可以把本地队列管理器 QM 上的本地队列 Q 和远程队列管理器 QM1 和 QM2 上的队列 Q1、Q2、Q3、Q4 放入一个分发列表中一起操作。当然，事先需要配置通信环境，如下：

QM:

```
DEFINE QLOCAL      (Q)                +
      REPLACE
DEFINE QLOCAL      (QM1)               +
      USAGE        (XMITQ)             +
      REPLACE
DEFINE QLOCAL      (QM2)               +
      USAGE        (XMITQ)             +
      REPLACE
DEFINE CHANNEL     (C1)                +
      CHLTYPE      (SDR)               +
      TRPTYPE      (TCP)               +
      CONNAME      ('127.0.0.1 (1415)') +
      XMITQ        (QM1)               +
      REPLACE
DEFINE CHANNEL     (C2)                +
      CHLTYPE      (SDR)               +
      TRPTYPE      (TCP)               +
```

```

CONNAME      ('127.0.0.1 (1416)')      +
XMITQ        (QM2)                      +
REPLACE

QM1:
----
DEFINE QLOCAL      (Q1)                  +
REPLACE
DEFINE QLOCAL      (Q2)                  +
REPLACE
DEFINE CHANNEL     (C1)                  +
CHLTYPE        (RCVR)                   +
TRPTYPE        (TCP)                    +
REPLACE

QM2:
----
DEFINE QLOCAL      (Q3)                  +
REPLACE
DEFINE QLOCAL      (Q4)                  +
REPLACE
DEFINE CHANNEL     (C2)                  +
CHLTYPE        (RCVR)                   +
TRPTYPE        (TCP)                    +
REPLACE

start runmqslsr -m QM1 -t tcp -p 1415
start runmqslsr -m QM2 -t tcp -p 1416
runmqsc QM
    start channel (C1)
    start channel (C2)

```

我们可以运行书后例程 MQPutDistList 来感觉一下用分发列表来广播消息的过程。

```
MQPutDistList QM Q QM1 Q1 QM1 Q2 QM2 Q3 QM2 Q4
```

7.1.3 编程

分发列表编程的过程与普通编程是一样的，只是在 MQOPEN 时需要在对象描述符 (MQOD) 结构中指定分发列表，以后 MQPUT 放入远程队列的消息在传输队列中会由 WebSphere MQ 自动加上分发列表信息，并在传输中根据需要自动复制。下面我们将详细介绍分发列表编程中涉及的数据结构和编程方法。

7.1.3.1 数据结构

7.1.3.1.1 MQDH

当多个远程队列的目标队列管理器相同时，WebSphere MQ 会自动为传输队列中的消息加上一个分发列表头 MQDH，图 7-2。例：按上一节中的配置，把本地队列管理器 QM 上的本地队列 Q 和远程队列管理器 QM1 和 QM2 上的队列 Q1、Q2、Q3、Q4 放入一个分发列表中一起操作。这时放入传输队列 QM1 和 QM2 中的消息格式为 :MQXQH(传输头) + MQDH(分发头) + MQOR 数组(可选) + MQRR 数组(可选) + 消息体。其中传输队列 QM1 中的消息的 MQDH 结构中的 RecsPresent 为 2，MQOR 数据为 { {“ Q1 ”，“ QM1 ”}，{“ Q2 ”，“ QM1 ”} }，如下图。传输队列 QM2 中的消息的 MQDH 结构中的 RecsPresent 为 2，MQOR 数据为 { {“ Q3 ”，“ QM2 ”}，{“ Q4 ”，“ QM2 ”} }。

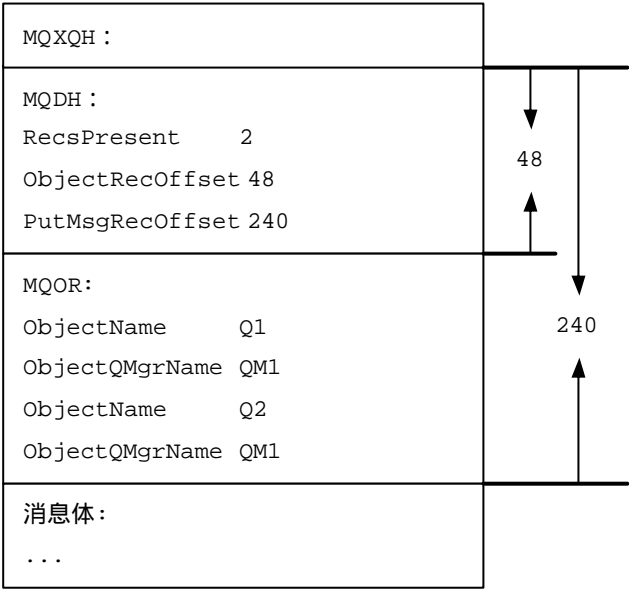


图 7-2 分发列表消息的消息头

MQDH 结构如下：

```
struct tagMQDH
{
    MQCHAR4 StrucId;           // “ DH ”
    MQLONG Version;           // MQDH_VERSION_1
    MQLONG StrucLength;        // MQDH 及后继的 MQOR 和 MQPMR 记录总长度
    MQLONG Encoding;           // MQOR 和 MQPMR 记录之后的数据编码方式
    MQLONG CodedCharSetId;     // MQOR 和 MQPMR 记录之后的数据字符集
    MQCHAR8 Format;            // MQOR 和 MQPMR 记录之后的数据格式
    MQLONG Flags;              // 标志
    MQLONG PutMsgRecFields;     // MQPMR 中所含的域
    MQLONG RecsPresent;        // MQOR 记录数
    MQLONG ObjectRecOffset;     // MQOR 记录的偏移量，从 MQDH 头开始计
    MQLONG PutMsgRecOffset;     // MQPMR 记录的偏移量，从 MQDH 头开始计
};
```

其中，Flags 可以是 MQDHF_NEW_MSG_IDS 或 MQDHF_NONE。前者表示队列管理器会为每条物理消息自动生成消息标识 (MsgId)，后者则表示无相应的功能。队列管理器在加上分发列表头 MQDH 时如果遇到以下情况，会将 Flags 域设置为 MQDHF_NEW_MSG_IDS：

- MQPUT 调用时，MQPMO 参数结构中不带 MQPMR 数组，或是虽然带有 MQPMR 但 MQPMR 结构中无 MsgId 域。
- MQPUT 调用时，MQMD 结构中 MsgId 域为 MQMI_NONE (即 24 个空格) 或者 MQPMO 结构中的 Options 域含有 MQPMO_NEW_MSG_ID 选项。

7.1.3.1.2 MQOR

MQOPEN 和 MQPUT1 中的 MQOD 参数有 ObjectRecPtr 指针域，指向 MQRR 数组。其中，每一个 MQRR 结构指明了目标队列及目标队列管理器。如下：

```
struct tagMQOR
{
    MQCHAR48          ObjectName; // 队列名
    MQCHAR48          ObjectQMgrName; // 队列管理器名
};
```

7.1.3.1.3 MQRR

MQOPEN 和 MQPUT1 中的 MQOD 参数有 ResponseRecPtr 指针域，MQPUT 时 MQPMO 参数也有 ResponseRecPtr 指针域，它们都指向 MQRR 数组。其中，每一个 MQRR 结构指明了目标队列操作的完成码和原因码。如下：

```
struct tagMQRR
{
    MQLONG  CompCode; // 完成码
    MQLONG  Reason;   // 原因码
};
```

7.1.3.1.4 MQPMR

MQPUT 时 MQPMO 参数有 PutMsgRecPtr 指针域，指向 MQPMR 结构。MQPMR 结构很特殊，它并不是一个固定格式的数据结构，所以在 C 的头文件或 COBOL 的 COPY 文件中都找不到它的定义。该结构可以按次序含有以下一个或多个域：

MQPMR:

```
MQBYTE24  MsgId
MQBYTE24  CorrelId
MQBYTE24  GroupId
MQLONG    Feedback
MQBYTE32  AccountingToken
```

为了指明 MQPMO 中到底有哪些域，通常 MQPMR 需要有 MQPMRF 域配合使用，后

者在 MQPMO 中为 PutMsgRecFields 域。它可以是以下多个标记位的叠加：

Put Message Record Fields：

```
#define MQPMRF_MSG_ID 0x00000001
#define MQPMRF_CORREL_ID 0x00000002
#define MQPMRF_GROUP_ID 0x00000004
#define MQPMRF_FEEDBACK 0x00000008
#define MQPMRF_ACCOUNTING_TOKEN 0x00000010
#define MQPMRF_NONE 0x00000000
```

7.1.3.2 MQOPEN

MQOPEN 函数有 MQOD 结构参数，该参数结构中有 Version、ObjectType、RecsPresent、ObjectRecPtr 和 ResponseRecPtr 等域。使用分发列表时，在 MQOPEN 前 Version 域必须设置为 MQOD_VERSION_2 (或以上)，以使后继的 RecsPresent、ObjectRecPtr 和 ResponseRecPtr 等域生效。ObjectType 必须设置为 MQOT_Q，表示操作对象为队列。RecsPresent 为打开的队列数，应该是一个大于零的数字。ObjectRecPtr 指向 MQOR 数组，其中详细记录了要打开的队列名和所在的队列管理器名。ResponseRecPtr 指向 MQRR 数组，其中详细记录了每个队列的操作返回码。如图 7-3。

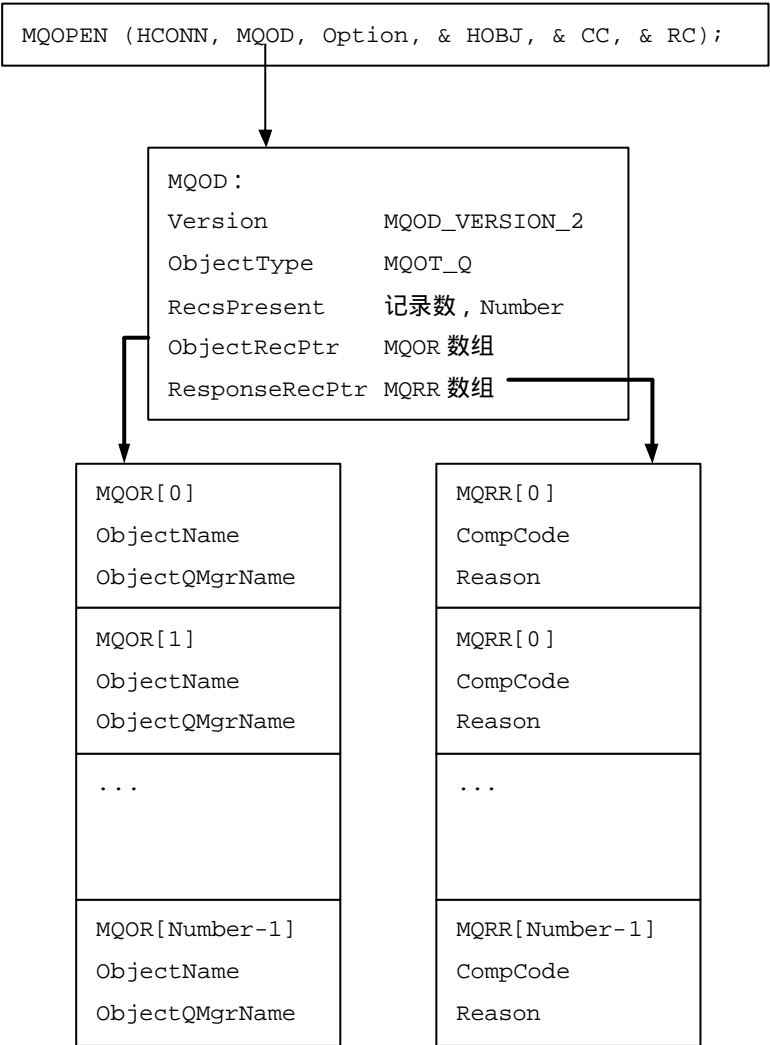


图 7-3 分发列表中 MQOPEN 的参数及相关数据结构

MQOR 数组和 MQRR 数组可以是动态分配的内存，分别由 ObjectRecPtr 和 ResponseRecPtr 指向它们的地址，在编程时要注意内存回收的工作。其中，MQOR 数组是输入参数，在 MQOPEN 调用前要设置好队列名和队列管理器名。MQRR 数组是输出参数，在 MQOPEN 调用前可以不做初始化工作，在调用后返回对每个队列操作的返回码。

MQOPEN 调用返回 HOBJ 为整个分发列表的句柄，以后对该句柄 MQPUT 或 MQPUT1 即对列表中所有队列同时操作。一般来说，通过 MQOPEN 返回的 CC (CompletionCode, 完成码) 和 RC (ReasonCode, 原因码) 可以得知 MQOPEN 操作是否成功，但要知道具体是哪一个或哪几个队列操作出错，则需要借助于 MQRR 数组。比如，原因码为 MQRC_MULTIPLE_REASONS，说明可能有多个队列操作出错，通过 MQRR 数组可以将它们找出来。

7.1.3.3 MQPUT

在 MQPUT 之前，HOBJ 参数必须设为 MQOPEN 时打开的分布列表句柄。同时，MQPUT 函数有 MQPMO 参数，该参数结构中的 Version 域必须置为 MQPMO_VERSION_2 (或以上)，以使后继的 RecsPresent、PutMsgRecFields、PutMsgRecPtr、ResponseRecPtr 等等域生效。RecsPresent 应该设置为 MQPMR 或 MQRR 的记录数，两者应该相等。PutMsgRecFields 和 PutMsgRecPtr 配合使用，指明一条消息为每个不同的目标队列设定的 MQPMR，其中 PutMsgRecFields 指明 MQPMR 中所含的域，PutMsgRecPtr 指针域指向 MQPMR 数组。ResponseRecPtr 指针域则指向 MQRR 数组，准备接收每个队列的操作返回码。如图 7-4：

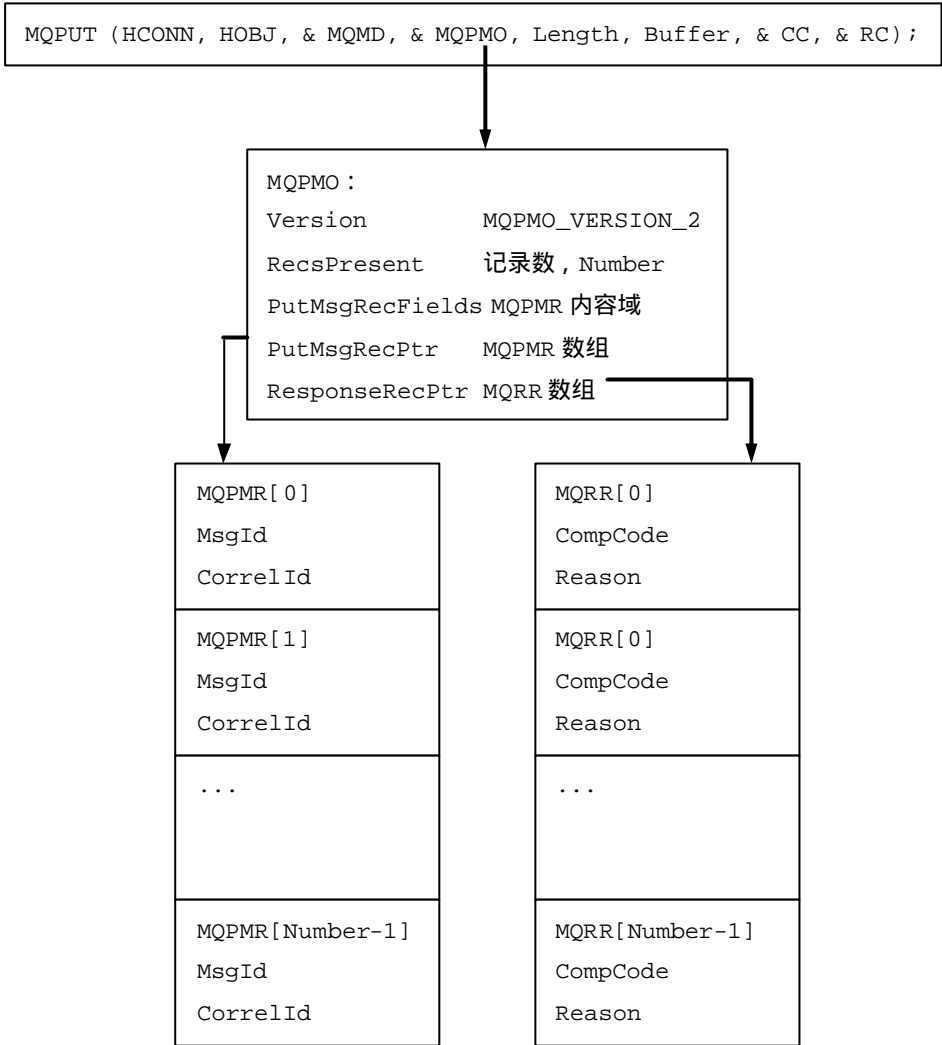


图 7-4 分发列表中 MQPUT 的参数及相关数据结构

与 MQOPEN 类似，MQPMO 数组和 MQRR 数组可以是动态分配的内存，分别由 PutMsgRecPtr 和 ResponseRecPtr 指针域指向它们的地址，在编程时要注意内存的回收工作。其中，PutMsgRecPtr 是输入参数，ResponseRecPtr 是输出参数。MQPMR 是一个可以伸缩的数据结构，它需要 PutMsgRecFields 配合使用指明其中所含的域。

在使用分发列表时，每条 MQPUT 的消息实际上裂变成多条消息，这些消息虽然内容相同，但本质上是不同的消息体。在实际使用中，我们通常有以下几种用法：

1. 我们需要区分这些不同的消息并由队列管理器自动为每条消息设定不同的 MsgId。可以将 MQMD 结构中 MsgId 域设置为 MQMI_NONE (即 24 个空格)，或者 MQPMO 结构中的 Options 域设置含有 MQPMO_NEW_MSG_ID 选项。
2. 我们需要区分这些不同的消息并由应用程序为每条物理消息设定不同的 MsgId，甚至是 CorrelId、GroupId 及 Feedback、AccountingToken。可以在 MQPMO 参数结构中用 MQPMR 数组，用 RecsPresent 指明数组长度。
3. 我们不需要区分这些不同的消息，换句话说，这些消息可以是完全一样的。可以设置 MQMD 结构中的 MsgId 和 CorrelId 域，MQPMO 结构中的 Options 域不含 MQPMO_NEW_MSG_ID 选项，不设置 MQPMR 数组，即 PutMsgRecPtr 为空指针。

4. 我们需要知道分发列表中每个队列的操作返回码。可以设置 ResponseRecPtr 指针，指向 MQRR 数组，用 RecsPresent 指明数组长度。
5. 我们不需要知道分发列表中每个队列的操作返回码。可以不设置 MQRR 数组，即 ResponseRecPtr 为空指针。如果既不设置 MQPMR 数组，也不设置 MQRR 数组，则 RecsPresent 设为零。

注意，MQPUT 中的 RecsPresent 指的是数组长度，也就是分发列表中的队列个数，它与 MQOPEN 时的 RecsPresent 应该始终是一致的。

7.1.3.4 MQPUT1

MQPUT1 可以看作是 MQOPEN + MQPUT + MQCLOSE，在使用时参见 MQOPEN 和 MQPUT。如果要获得分发列表中每个队列的操作返回码，可以在 MQOD 中设置 MQRR 数组，方法与 MQOPEN 相同。MQPMO 结构中的 ResponseRecPtr 和 ResponseRecOffset 必须设为零。

7.2 发布和订阅 (Pub & Sub)

7.2.1 概念

WebSphere MQ 传统上是为了解决点到点的之间数据的可靠传输，这种应用环境中每次的消息发送都只有一个发送者和一个接收者，消息的发送者明确地知道消息的目的地。但在有的应用环境中，消息的发送者在发送消息的时候并不知道有多少个接收者，更不知道消息的目的地在哪里。在这种情况下，就需要发送者用类似于广播的方式将消息发布出去。但如果所有的消息都在网络上广播，则网络负担太大，也不必要。如果所有的接收者收听所有的消息，则需要过滤众多无关的消息，影响效率。

实际上，在现实生活有一个很好的模式可以解决这个问题，那就是发布/订阅模式。在这种环境下，需要有设立一个中间机构，类似于邮局。消息的发布者就好像报社或杂志社，它在发布之前需要一次性地与邮局注册需要发行的刊物代码。消息订阅者就好像读者，需要到邮局确认需要订阅的刊物代码。邮局只负责递送业务，一旦有新的刊物出版，就会送到所有订阅过的读者手中。这种方式可以使得消息的订阅者只接收自己感兴趣的消息。

在 WebSphere MQ 中这个中间机构叫做 Broker，消息发布程序叫做 Publisher，订阅程序叫做 Subscriber，订阅代码叫做主题 (Topic)，Topic 是一个字符串，可以含通配符，“*”表示零个或多个字符，“?”表示一个字符。Topic 可以用“/”进行分级，比如：Sport/Soccer/Reports。发布者和订阅者之间的只需约定一个 Topic 即可。类似于邮局和邮局可以互连，从而使本地读者可以订阅国外刊物。Broker 之间也可以级联，使发布的消息可以传递到 MQ 网络的任何地方。发布者发布的消息的目标叫做流 (Stream)，它由 Broker 控制，会自动将其中的内容为每一个订阅者拷贝一份，即复制一条消息放入订阅者在订阅时指定的目标队列。

MQ 发布和订阅中的 Broker 是建立在队列管理器之上的，与队列管理器一一对应。

Broker 本质上是运行在队列管理器上的一个 MQ 应用，它本身没有名字，所以我们也经常借用队列管理器名称来称呼 Broker。Broker 在创建之后会在队列管理器上建立自己的队列，Broker 之间的通信，也是借助于队列管理器之间的通道配置完成的。事实上，Broker 本质上就是 WebSphere MQ 上的应用。

目前 WebSphere MQ 的发布和订阅功能是通过 SupportPac MA0C 来实现。相关的 SupportPac 如下：

- MA0C MQSeries – Publish/Subscribe
- MA0D Getting started with MQSeries Publish/Subscribe
- MP03 MQSeries Publish/Subscribe – Performance report

7.2.2 安装

从 IBM 网站上下载 WebSphere MQ SupportPac MA0C，根据网页上的安装指导进行安装，也可根据<<MQSeries Publish/Subscribe User's Guide>> (SupportPac MA0C 网页上可以下载)，进行安装。由于安装过程十分简单，这里不再赘述。

7.2.3 Broker 控制命令

Broker 的控制命令很像队列管理器的控制命令，可以实现对 Broker 的启动、停止、删除、显示状态等功能。其中第一启动 Broker 隐含了对它的创建工作。

7.2.3.1 strmqbrk 启动 Broker

- 格式

strmqbrk [-p ParentQMGrName] [-m QMGrName]

- 功能

启动 Broker

- 说明

第一次启动 Broker 的同时创建 Broker，可以用 -p 选项指定上一级 Broker 所在的队列管理器的名字，用 -m 选项指定 Broker 所在的队列管理器的名字，缺省为缺省队列管理器

在第一次启动的时候，系统会创建：

- | | |
|--------------------------------|---------------------|
| ■ SYSTEM.BROKER.CONTROL.QUEUE | Broker 控制命令队列 |
| ■ SYSTEM.BROKER.DEFAULT.STREAM | 缺省 Stream |
| ■ SYSTEM.BROKER.ADMIN.STREAM | 用来发布 Broker 自己的配置信息 |
| ■ SYSTEM.BROKER.MODEL.STREAM | 用来创建动态 Stream 队列 |
| ■ SYSTEM.BROKER.* | 其它内部队列 |

- 例：

strmqbrk -m QM

MQSeries Publish/Subscribe 代理为队列管理器 QM 启动。

7.2.3.2 endmqbrk 停止 Broker

- 格式
endmqbrk [-c | -i] [-m QMgrName]
- 功能
停止 Broker
- 说明
停止 Broker 时可以选择停止方式，如果选择 -c，表示受控方式停止 (Control shutdown)，这是缺省值，Broker 会等目前正在进行的操作做完后才停止工作。如果选择 -i，表示立即停止 (Immediate shutdown)，Broker 不再做任何消息处理，将未完成的
消息回滚，立即结束。通常 endmqbrk 需要指定 Broker 所在的队列管理器，如果不指定，
则缺省认为是缺省队列管理器。
- 例：
endmqbrk -m QM
队列管理器 QM 的 MQSeries Publish/Subscribe 代理结束。

7.2.3.3 dltmqbrk 删除 Broker

- 格式
dltmqbrk -m QMgrName
- 功能
删除 Broker
- 说明
删除 Broker 时必须指定 Broker 所在的队列管理器，在经过整理后系统会删除第一次启动时创建的以 SYSTEM.BROKER 打头的对象。
待删除的 Broker 必须满足两个条件：
 1. 是停止的，否则无法删除
 2. 必须没有下级 Broker，否则会造成订阅链断开或订阅信息缺失。
- 例：
dltmqbrk -m QM
MQSeries Publish/Subscribe 代理 (QM) 已删除。

7.2.3.4 dspmqbrk 显示 Broker 状态

- 格式
dspmqbrk [-m QMgrName]
- 功能
显示 Broker 运行状态
- 说明
可以用 -m 选项指明 Broker 所在的队列管理器的名字，缺省为缺省队列管理器。
- 例：
dspmqbrk -m QM
队列管理器 QM 的 MQSeries Publish/Subscribe 代理正在运行。

7.2.3.5 clrmqbrk 清除 Broker 的上下级关系

- 格式

clrmqbrk [-p | -c ChildQMgrName] -m QMgrName

- 功能

清除 Broker 的上下级关系

- 说明

多个 Broker 可以通过上下级关系组成树状结构,每个 Broker 只有一个上级 Broker,或称父结点,但可以有多个下级 Broker,或称子结点。在执行 clrmqbrk 命令时可以指定清除某一个上下级关系,用 -p 指定清除与上级 (Parent) 之间的关系,用 -c ChildQMgrName 指定清除与某个下级 (Child) 的关系,其中 ChildQMgrName 是下级 Broker 所在的队列管理器的名字。

执行此条命令时 Broker 必须处于停止状态,执行这条命令后将清除的该 Broker 对自己上下级关系的记忆。但它原来的上下级 Broker 仍然记得与这个 Broker 之间的关系。所以,原来的上下级 Broker 的相应关系也应该被重新设置,才能真正地将该 Broker 从网络中隔离出去。

- 例:

clrmqbrk -c SubBroker -m QM

clrmqbrk -p -m QM

7.2.4 Broker 网络

多个 Broker 可以通过配置形成树状结构,发布和订阅信息会通过 Broker 网络自动传送到其它结点中。

7.2.4.1 层次结构

Broker 可以有父结点和子结点,组成层次结构。层次结构中的结点在接受订阅时,如果发现自己有相同主题的发布者则回应订阅成功。如果发现接受过相同主题的订阅,则说明该主题的消息会通过网络由相邻结点送过来。否则,向相邻结点广播这份订阅。

为了解释清楚 Broker 网络中的订阅传播过程,让我们来看一个例子。

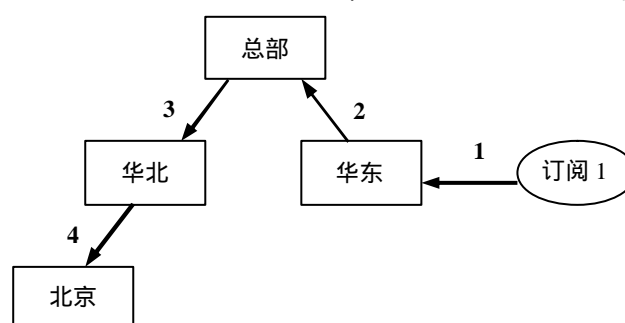


图 7-5 订阅 1 的传播路径

订阅 1 连接到华东 Broker 上订阅消息,华东 Broker 发现自己没有这个 Topic,于是向总部 Broker 订阅。总部收到来自华东的订阅,又向华北 Broker 订阅,后者再向北京 Broker

订阅，图 7-5。这样一来，整个 Broker 网络上的所有结点都知道有对于订阅 1 的请求，如果出现相同主题的发布消息，结点会根据订阅路径将消息逆向返回到订阅者。

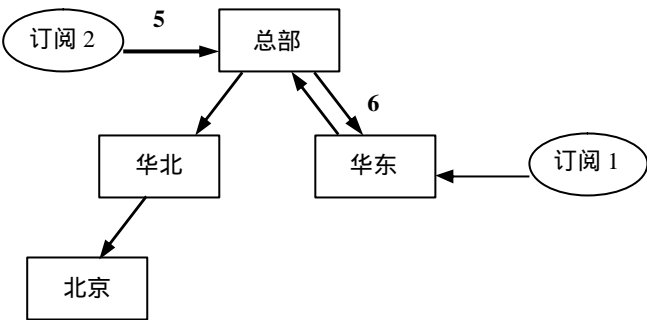


图 7-6 订阅 2 的传播路径

订阅 2 连接到总部 Broker 订阅相同 Topic 的消息，总部下辖两个结点：华东和华北，总部已经向华北订阅过了，于是向华东 Broker 订阅，图 7-6。这时，总部和华东相互订阅，任何一个 Broker 发现有符合主题的消息，都会通知对方。

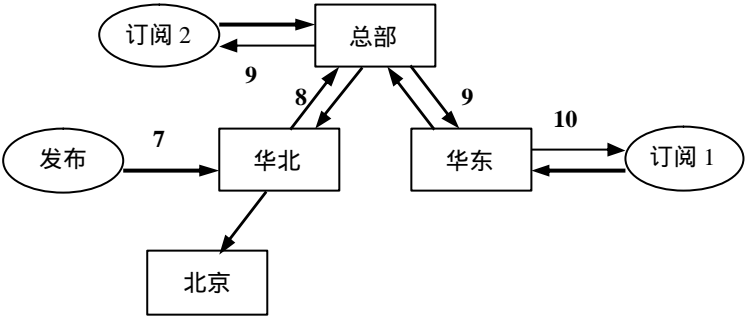


图 7-7 发布消息的传播路径

在华北出现了一个发布者，由于总部曾经向华北订阅过，华北将消息发布到总部，总部同时将消息传递给本地订阅者 2 和华东，再经由华东将消息传达订阅者 1，整个过程如图 7-7。

7.2.4.2 加入一个 Broker

首先，创建 Broker 所在的队列管理器与上下级 Broker 所在的队列管理器之间的双向通道，通常传输队列取名相同于对方的队列管理器名，不然，用队列管理器别名也要可以。然后，用 `strmqbrk -p` 命令把 Broker 作为子结点挂到父结点下面。

7.2.4.3 删除一个 Broker

首先要停止这个 Broker，确保通道畅通，结束相关应用程序。如果接受通道上有队列打开，将通道重启以关闭队列。用 `dlmqbrk` 将其删除。

7.2.4.4 移走一个 Broker

可以用 `clrmqbrk` 命令打破 Broker 上下级之间的关系。例如：我们要从网络中的三层 Broker 结构中移走中间一个 Broker，变成两层结构，如图 7-8。

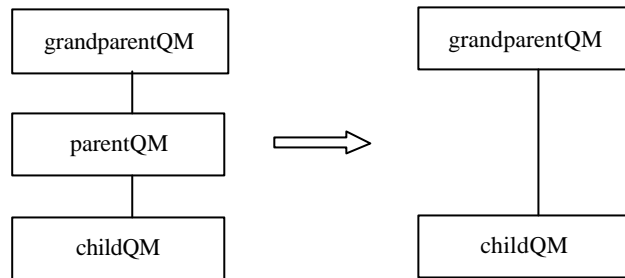


图 7-8 移走一个 Broker

我们可以用以下命令将中间的 Broker 移走

1. `clrmqbrk -m grandparentQM -c parentQM`
2. `clrmqbrk -m parentQM -p`
3. `clrmqbrk -m parentQM -c childQM`
4. `clrmqbrk -m childQM -p`

7.2.5 编程设计

对 MQ 发布和订阅编程，其实是将各种发布/订阅命令 (Publish 除外) 以特定的消息格式发送给 Broker 的控制队列，Broker 在处理后会将执行结果返回应答队列。消息分成 MQMD、MQRFH、UserData 三部分组成，其中，MQMD.ReplyToQueue 和 MQMD.ReplyToQueueManager 指定应答队列。对于 Publish 命令，发送的对象不是控制队列而是 Stream 队列。

对于 Publisher，要事先注册发布者，指定 Topic 和 Stream，表示有这样一个 Topic 需要发布，会发布到 Stream 队列上。将命令消息发到 Control 队列 (SYSTEM.BROKER.CONTROL.QUEUE)，从 Reply 队列中取执行结果。以后每次发布该 Topic 的消息，将消息送到 Stream 队列，再从 Reply 队列中取执行结果，表示发布是否成功。

对于 Subscriber，要事先注册订阅者，指定 Topic、Stream 和 Queue，表示有这样一个 Topic 需要订阅，如果消息出现在 Stream 队列上，Broker 复制一份到 Queue 队列上。类似于 Publisher，将命令消息发到 Control 队列，从 Reply 队列中取执行结果。以后的 Receiver 只要等在 Queue 上读消息即可。消息从 Stream 到 Queue 的复制是 Broker 自动完成的。基本过程如图 7-9：

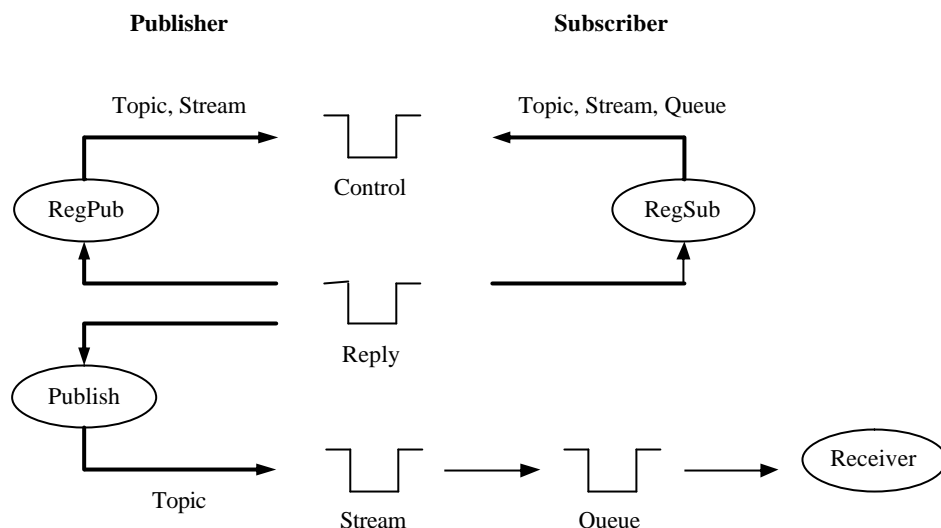


图 7-9 发布/订阅的过程

对于 DeregPub 和 DeregSub，几乎与 RegSub 和 RegPub 一样。此外，在 MQ Pub/Sub 中还有一个概念，叫做 Retained Publication。通常情况下，Broker 在处理完 Publisher 发布的消息后，就会删除这条消息，对于新的 Subscriber，永远只能等待下一次 Publisher 发布新的消息。在有的应用环境下，Publisher 发布的是某种状态信息，新的 Subscriber 启动后需要立即收集这些状态信息，而不是陷入漫长的等待，这就需要消息在发布时用 Retained 方式，由 Broker 记住。如果 Publisher 在 RegPub 的时候使用 MQPS_RETAIN_PUBLICATION 选项，则之后这个 Topic 发布的消息最新一条消息会被 Broker 记住。以后，新的 Subscriber 可以通过请求更新命令，获得 Broker 记住的最新状态信息。

发布/订阅的消息格式大同小异，都是由 MQMD、MQRFH 和 UserData 三部分组成。以 Publish 消息为例，格式如图 7-10。MQMD 中的编码、字符集、格式指的是后继 MQRFH 中的内容，而 RFH 中的编码、字符集、格式指的是后继的订阅/发布命令及以后的 UserData 内容。订阅/发布命令需要遵循一定的格式和规则，UserData 可以是相关的任何内容，例图 7-10 是一条发布命令，发布的消息主题为“IBM Stock Price”，消息内容为“\$112.85”。

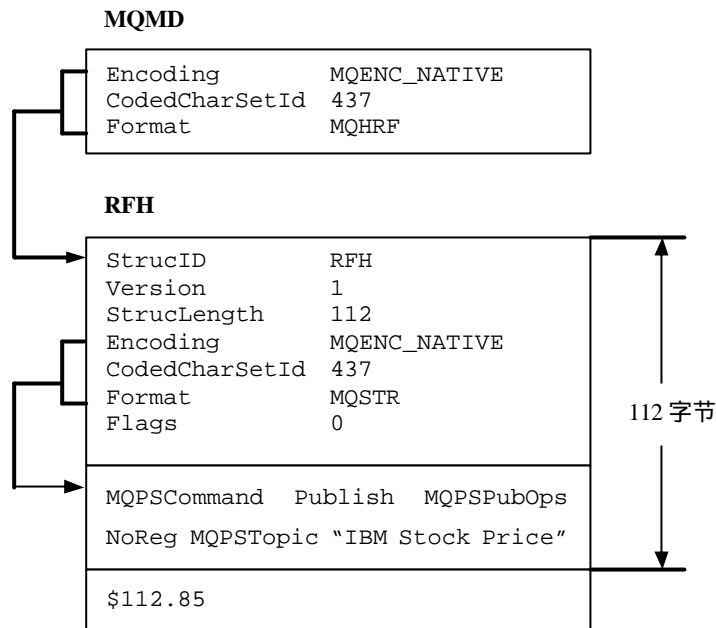


图 7-10 发布/订阅的消息结构

7.2.6 发布/订阅命令

发布/订阅命令包含注册订阅者、注销订阅者、注册发布者、注销发布者、删除发布、发布、请求更新等命令，每条命令由一个命令项和多个参数项组成，参数项中有些是必要的，有些则是可选的。命令项和参数项都是由名/值对组成，它们之间由空格隔开，名和值之间也用空格隔开。比如：*MQPSCCommand Publish MQPSTopic T1*。

7.2.6.1 注册订阅者 (Register Subscriber)

- 格式

MQPSCCommand RegSub MQPSTopic <Topic> ...

- 功能

注册订阅者命令告诉 Broker 有这样一个订阅者：它对 Stream 队列中主题为 Topic 的消息感兴趣，请 Broker 将这样的消息复制一份到队列 Queue 中。

- 说明

注册订阅者的命令项为 *MQPSCCommand RegSub*，必要参数为 Topic，可选参数为 RegistrationOptions、StreamName、QMgrName、QName、SubName、SubIdentity 和 SubUserData。其中 StreamName 指明消息的来源，即流队列，缺省为 SYSTEM.BROKER.DEFAULT.STREAM。QMgrName 和 QName 指明拷贝消息的目标队列。

- 例：

如果在 MyStreamQ 中出现主题为 T1 的消息，则拷贝一份并放入目标队列 Q1。

MQPSCCommand RegSub MQPSTopic T1 MQPSSStreamName MyStreamQ MQPSSQName Q1

7.2.6.2 注销订阅者 (Deregister Subscriber)

- 格式
MQPSCCommand DeregSub ...
- 功能
注销订阅者命令通知 Broker 将某一主题的订阅者注销，即不再需要进行消息复制了。
- 说明
注销订阅的命令项为 MQPSCCommand DeregSub，可选参数为 RegistrationOptions、StreamName、Topic、QMgrName、QName、SubName 和 SubIdentity。其中 Topic 指明消息主题，StreamName 指明消息的来源，即流队列。QMgrName 和 QName 指明拷贝消息的目标队列。
- 例：
注销对主题 T1 的订阅。
MQPSCCommand DeregSub MQPSTopic T1

7.2.6.3 注册发布者 (Register Publisher)

- 格式
MQPSCCommand RegPub MQPSTopic <Topic> ...
- 功能
注册发布者命令告诉 Broker 有这样一个发布者：它将向 Stream 队列发布主题为 Topic 的消息。
- 说明
注册发布的命令项为 MQPSCCommand RegPub，必要参数为 Topic，可选参数为 RegistrationOptions、StreamName、QMgrName 和 QName。其中 StreamName 指定流队列，缺省为 SYSTEM.BROKER.DEFAULT.STREAM。
- 例：
注册发布主题为 T1，消息发布放入流队列 MyStreamQ
MQPSCCommand RegPub MQPSTopic T1 MQPSSStreamName MyStreamQ

7.2.6.4 注销发布者 (Deregister Publisher)

- 格式
MQPSCCommand DeregPub ...
- 功能
注销发布者命令通知 Broker 将某一主题的发布者注销，即不再会有这样的发布者了。
- 说明
注销发布的命令项为 MQPSCCommand DeregPub，可选参数为 RegistrationOptions、StreamName、Topic、QMgrName 和 QName，其中 StreamName 指定流队列，缺省为 SYSTEM.BROKER.DEFAULT.STREAM。
- 例：

注销主题为 T1 的发布者

MQPSCCommand DeregPub MQPSTopic T1

7.2.6.5 删除发布信息 (Delete Publication)

- 格式

MQPSCCommand DeletePub MQPSTopic <Topic>...

- 功能

删除发布信息命令将 Broker 中指定的 *Topic* 主题的信息删除

- 说明

删除发布信息的命令项为 MQPSCCommand DeletePub，必要参数为 Topic，可选参数为 DeleteOptions 和 StreamName。其中 StreamName 指明消息的来源，即流队列。

- 例：

删除主题为 T1 的发布信息

MQPSCCommand DeletePub MQPSTopic T1

7.2.6.6 发布信息 (Publish)

- 格式

MQPSCCommand Publish MQPSTopic <Topic>

- 功能

向流队列中发布信息，主题为 *Topic*

- 说明

发布信息命令项为 MQPSCCommand Publish，必要参数为 Topic，可选参数为 RegistrationOptions、PublicationOptions、StreamName、QMgrName、QName、PublishTimestamp、SequenceNumber、StringData、IntegerData。其中 Topic 指明消息主题，StreamName 指明流队列。

- 例：

向 MyStreamQ 中发布主题为 T1 的信息

MQPSCCommand Publish MQPSTopic T1 MQPStreamName MyStreamQ

7.2.6.7 请求更新 (Request Update)

- 格式

MQPSCCommand ReqUpdate MQPSTopic <Topic>

- 功能

请求主题为 *Topic* 的最新的发布信息

- 说明

请求更新命令项为 MQPSCCommand ReqUpdate，必要参数为 Topic，可选参数为 RegistrationOptions、StreamName、QMgrName、QName 和 SubName。其中 Topic 指明消息主题，StreamName 指明流队列。

- 例：

请求主题为 T1 的最新的发布信息

7.2.7 常见的问题

WebSphere MQ 订阅/发布对应用编程的要求比较高，如果应用程序在订阅/发布的方法或时序上处理不当，有时会导致 Broker 无法正常工作，以下总结了一些常见的问题及解决办法。

- 症状：dltmqbrk 时报错 AMQ5842 (5805)

```
dltmqbrk -m QM
```

```
AMQ5842: 无法删除 MQSeries Publish/Subscribe 代理 (QM), 因为 '5805:
(MQSeries Publish/Subscribe 代理活动)'。
```

但是 endmqbrk -m QM 会无响应

原因：

Broker 工作不正常

解决：

```
endmqm -p QM
```

```
strmqm QM
```

```
dltmqbrk -m QM
```

- 症状：dltmqbrk 时报错 AMQ5842 (5837)

```
dltmqbrk -m QM
```

```
AMQ5842: 无法删除 MQSeries Publish/Subscribe 代理 (QM), 因为 '5837: (一个或多个队列无法停顿。)'。
```

原因：

SYSTEM.BROKER.CONTROL.QUEUE 或 SYSTEM.BROKER.DEFAULT.STREAM 不为空

解决：

清空队列中的内容

```
runmqsc QM
```

```
clear ql (SYSTEM.BROKER.CONTROL.QUEUE)
```

```
clear ql (SYSTEM.BROKER.DEFAULT.STREAM)
```

```
dltmqbrk -m QM
```

- 症状：amqfcxba.exe - 应用程序错误。

...内存不能为 "read"。要终止程序，请单击"确定"，要调试程序，请单击"取消"。

原因：

往 SYSTEM.BROKER.CONTROL.QUEUE 上发送的消息格式有误，造成 Broker 在取消息体时内存偏移量指针出错。检查：

MQMD.Encoding

MQMD.CodedCharSetId

消息体的格式

解决：

这时，amqfcxba 失败退出，Broker 程序变得不完整。尽管 dspmqbrk 可能返回：队列管理器 QM 的 MQSeries Publish/Subscribe 代理正在运行。

重启 QM, 删除并重建 Broker:

```
endmqm -p QM  
strmqm QM  
dltmqbrk -m QM  
strmqbrk -m QM
```

修改程序, 重试。

第 8 章 客户端

WebSphere MQ 产品分客户端 (Client) 和服务器端 (Server) 两部分。客户端只是提供一个 WebSphere MQ 的接入环境, 本身没有各种对象, 即没有队列管理器、队列、名称列表等等, 需要连接到服务器上才能工作。

客户端与服务器端的连接是通过一条特殊通道实现的, 称为 MQI 通道 (图 5-4)。在该通道上传送的是 MQI 调用, 消息是作为 MQI 参数连同调用一起传送到服务器端, WebSphere MQ 会在服务器端执行该调用, 结果仍作为参数原路返回客户端。在客户端的通道类型为 CLNTCONN, 在服务器端为 SVRCONN。事实上, 大多数情况下客户端无需指定通道, 设定环境变量即可。

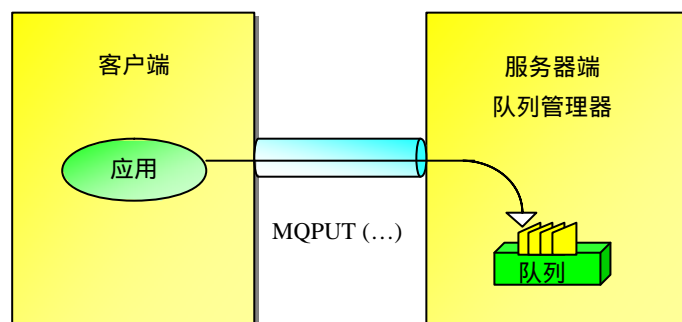


图 5-4 客户端工作原理

安装客户端产品过程与服务器端类似, 可以在服务器介质安装时选择客户端, 也可以使用单独的客户端安装介质, 效果是相同的。

客户端的编程时调用的 API 与服务器端是完全相同的, 在编译的时候服务器端连接 mqm 库, 而客户端连接 mqic 库, 仅此而已。

8.1 配置

创建客户端的运行环境, 服务器端与客户端都需要进行配置。服务器端的配置相对比较简单, 客户端的配置有设定环境变量、使用通道定义表、编程指定通道参数三种方式, 下面将逐一介绍并给出配置样例。

8.1.1 Server 端配置

Server 端需要配置服务器通道，或者打开队列管理器的服务器通道自动定义开关（缺省时光开关是关闭的），在客户端连接的时候自动生成服务器通道。

```
// 可以选用简单定义，不带用户出口。
// 这里 Guest 是 Server 端的用户，作为 Client 端连接后的权限认证用户名。
DEFINE CHANNEL (C_C.S) +
        CHLTYPE (SVRCONN) +
        TRPTYPE (TCP) +
        MCAUSER ('Guest') +
        REPLACE
// 也可以选用复杂定义，带用户出口。这些出口在 Server 端执行。
// 这里 chenyux 是 Server 端的用户，作为 Client 端连接后的权限认证用户名。
DEFINE CHANNEL (C_C.S) +
        CHLTYPE (SVRCONN) +
        TRPTYPE (TCP) +
        SCYEXIT ('E:\MQChannelSecurityExit.dll (MQChannelSecurityExit)') +
        SCYDATA ('E:\MQChannelSecurityExit.log') +
        SENDEXIT ('E:\MQChannelSendExit.dll (MQChannelSendExit)') +
        SENDDATA ('E:\MQChannelSendExit.log') +
        RCVEXIT ('E:\MQChannelReceiveExit.dll (MQChannelReceiveExit)') +
        RCVDATA ('E:\MQChannelReceiveExit.log') +
        MCAUSER ('chenyux') +
        REPLACE

// 打开服务器通道自动定义开关
ALTER QMGR +
        CHAD (ENABLED)

start runmqslsr -m QM -t tcp -p 1414
```

8.1.2 Client 端配置

Client 端应用程序可以有三种方式指定连接 Server 端的通道

- 在 Client 端设置环境变量 MQSERVER，指定通道。
- 在 Client 端使用通道定义表 (Client Channel Definition Table)，指定通道。
- 在 Client 端编程中指定通道。

8.1.2.1 环境变量

设置环境变量 `MQSERVER=ChannelName/TransportType/ConnectionName`，其中 ChannelName 为服务器端的服务器通道名，TransportType 是通信协议可以是 LU62、TCP、

NETBIOS、SPX、DECNET，ConnectionName 可以相关的通信参数。例：

```
set MQSERVER=ChannelName/TCP/server-address(port) // Windows
export MQSERVER=ChannelName/TCP/'server-address(port)' // UNIX
```

在 Windows 中环境变量的大小写无关，例以下两种设置方法的效果是相同的：

```
set MQSERVER=C_C.S/TCP/127.0.0.1(1414)
set mqserver=C_C.S/tcp/127.0.0.1 (1414)
```

可以用 Client 端测试例程 amqsputc 和 amqsgetc 检查是否配置成功，这些例程与 Server 端的 amqsput 和 amqsget 功能相同。

```
amqsputc <Q> [<QMgr>]
amqsgetc <Q> [<QMgr>]
```

由于用户可以在不同的运行窗口为 Client 应用程序设置不同的环境变量，从而连接到不同的 Server 队列管理器上。这是一种最为灵活简便的方式。

实际上，服务器端如果不定义 SVRCONN 类型的通道也可以，只要打开队列管理器的通道自动定义（Channel Auto-Definition）开关即可。WebSphere MQ 会根据客户端的 MQSERVER 环境变量自动创建对应的通道，这些通道在队列管理器重启后仍然保留。

8.1.2.2 客户端通道定义表

通常情况下，在服务器端定义的通道会记录在队列管理器的 @ipcc 目录下的 AMQCLCHL.TAB 文件中，该文件就是通道定义文件。用户可以先在 Server 端定义 CLNTCONN 和 SVRCONN 通道，再将通道定义文件拷贝到 Client 端，用环境变量指明该文件。具体步骤如下：

1. 在 Server 端定义 CLNTCONN 通道（Client-Connection Channel）和 SVRCONN 通道（Server-Connection Channel），两者要同名。例：

```
DEFINE CHANNEL (C_C.S) +
        CHLTYPE (SVRCONN) +
        TRPTYPE (TCP) +
        MCAUSER ('chenyux') +
        REPLACE
// 这里的用户出口在 Client 端执行，所以对应的是 Client 环境中的目录。
// 出口程序缺省放在 <InstallDir>/Exits 目录下
DEFINE CHANNEL (C_C.S) +
        CHLTYPE (CLNTCONN) +
        TRPTYPE (TCP) +
        CONNAME ('169.254.163.161 (1414)') +
        QMNAME (QM) +
        SCYEXIT ('D:\ChannelSecurityExit.dll (ChannelSecurityExit)') +
        SCYDATA ('D:\ChannelSecurityExit.log') +
        SENDEXIT ('D:\ChannelSendExit.dll (ChannelSendExit)') +
        SENDDATA ('D:\ChannelSendExit.log') +
```

```
RCVEXIT ('D:\ChannelReceiveExit.dll (ChannelReceiveExit)') +
RCVDATA ('D:\ChannelReceiveExit.log') +
REPLACE
```

2. 将 Server 端生成的通道定义文件 (Channel Definition Table) 拷贝到 Client 端，缺省情况下可以拷贝到客户端<InstallDir>目录下。当然也可以将其放在其它位置，只要用环境变量指明该文件且用户拥有访问权限即可。

Server 端：

```
<InstallDir>/Qmgrs/<QMgrName>/@ipcc/AMQCLCHL.TAB
```

Client 端：

```
<InstallDir>/AMQCLCHL.TAB
```

3. 在 Client 端用 MQCHLLIB 和 MQCHLTAB 指明客户端通道定义表 (Client Channel Definition Table) 的目录名与文件名。例：

```
set MQCHLLIB=D:\Program Files\IBM\WebSphere MQ
```

```
set MQCHLTAB=AMQCLCHL.TAB
```

事实上，如果不设置这两个环境变量，Client 缺省会试图找 <InstallDir>/AMQCLCHL.TAB 文件。所以只要将客户端通道定义表文件放对地方，不设环境变量也可。

使用通道定义表可以对客户端的连接进行配置上的限制，由于定义表文件本身是服务器端生成的二进制文件，在客户端无法编辑。这样客户端就无法随意地调整连接参数。

使用通道定义表的另一个好处是可以将多个通道定义放入定义表，Client 端在 MQCONN/MQCONNx 时可以用*打头，以匹配多条连接定义。一旦前面的通道无法连接，Client 会试图用后面的通道定义进行连接，从而形成主备式多通道，增加了程序的可靠性。

以下实例说明在这种环境下该如何配置：有两个队列管理器 QM1 和 QM2，上面有同名的本地队列 Q，且有相同的两套应用分别处理各自队列 Q 上的消息。这时，Client 程序如果发现主 QM1 连接不通，会稍后自动连接 QM2。具体步骤如下：

1. 在 Server 端 QM1，QM2 上定义

QM1

```
DEFINE QLOCAL (Q) +
REPLACE
DEFINE CHANNEL (C_C.S1) +
CHLTYPE (SVRCONN) +
TRPTYPE (TCP) +
MCAUSER ('chenyux') +
REPLACE
DEFINE CHANNEL (C_C.S1) +
CHLTYPE (CLNTCONN) +
TRPTYPE (TCP) +
CONNAME ('169.254.163.161 (1415)') +
QMNAME (QM) +
REPLACE
```

```

DEFINE CHANNEL (C_C.S2) +
CHLTYPE (CLNTCONN) +
TRPTYPE (TCP) +
CONNAME ('169.254.163.161 (1416)') +
QMNAME (QM) +
REPLACE

```

QM2

```

DEFINE QLOCAL (Q) +
REPLACE
DEFINE CHANNEL (C_C.S2) +
CHLTYPE (SVRCONN) +
TRPTYPE (TCP) +
MCAUSER ('chenyux') +
REPLACE

```

```
start runmqslsr -m QM1 -t tcp -p 1415
```

```
start runmqslsr -m QM2 -t tcp -p 1416
```

2. 将 QM1 中生成的通道定义表文件从 Server 端拷贝到 Client 端

Server 端：

```
<InstallDir>/Qmgrs/QM1/@ipcc/AMQCLCHL.TAB
```

Client 端：

```
<InstallDir>/AMQCLCHL.TAB
```

3. 在 Client 端用 MQCONN (*QM) 测试，用 WebSphere MQ 例程 amqsputc 或本书的例程 MQCONNClient.c

```
amqsputc Q *QM
```

```
MQCONNClient Q *QM
```

匹配过程如下：

- a) WebSphere MQ Client 在通道定义表文件中寻找符合 QMNAME (*QM) 的项，“*”表示可能匹配多项，QM 表示队列管理器名以 QM 打头。
- b) 如果找到，则使用该项中的定义，并要求：
 - i. Server 端的监听器 (Listener) 要在该项中指定的端口上启动
 - ii. Server 端的监听器 (Listener) 所在的队列管理器名与 QMNAME 指定的名字必须一致 (如果由 “*” 匹配，这一条可以不要求)
- c) 如果不能成功连接，则匹配以后的项，转去 b)

测试结果如下：

```
MQCONNClient Q QM
```

尽管 QM1 和 QM2 都能连通，操作仍失败，因为 QM 与 QM1/QM2 不匹配

```
MQCONNClient Q *QM
```

如果 QM1 能连通，操作 QM1 成功。否则稍后尝试 QM2，如果 QM2 能连通，操

作 QM2 成功。

8.1.2.3 编程指定通道参数

在连接队列管理器时使用 MQCONN 调用，其中的参数 MQCNO 指向连接选项结构 (Connection Options)，该结构的 ClientConnPtr 域指向 MQCD 通道定义结构 (Channel Definition)，利用 MQCD 结构中的 ChannelName、TransportType、ConnectionName 就可以指定通道连接所需的三个基本参数了 (图 5-5)。

此外，MQCD 可以指定 Client 端用户出口，也可以指定 SSL 相关参数。当然，使用 SSL 的同时需要在 MQCNO 结构的 SSL 配置选项结构 (SSL Config) 中指明证书仓库。关于证书仓库的配置可以参见 SSL 相关章节。

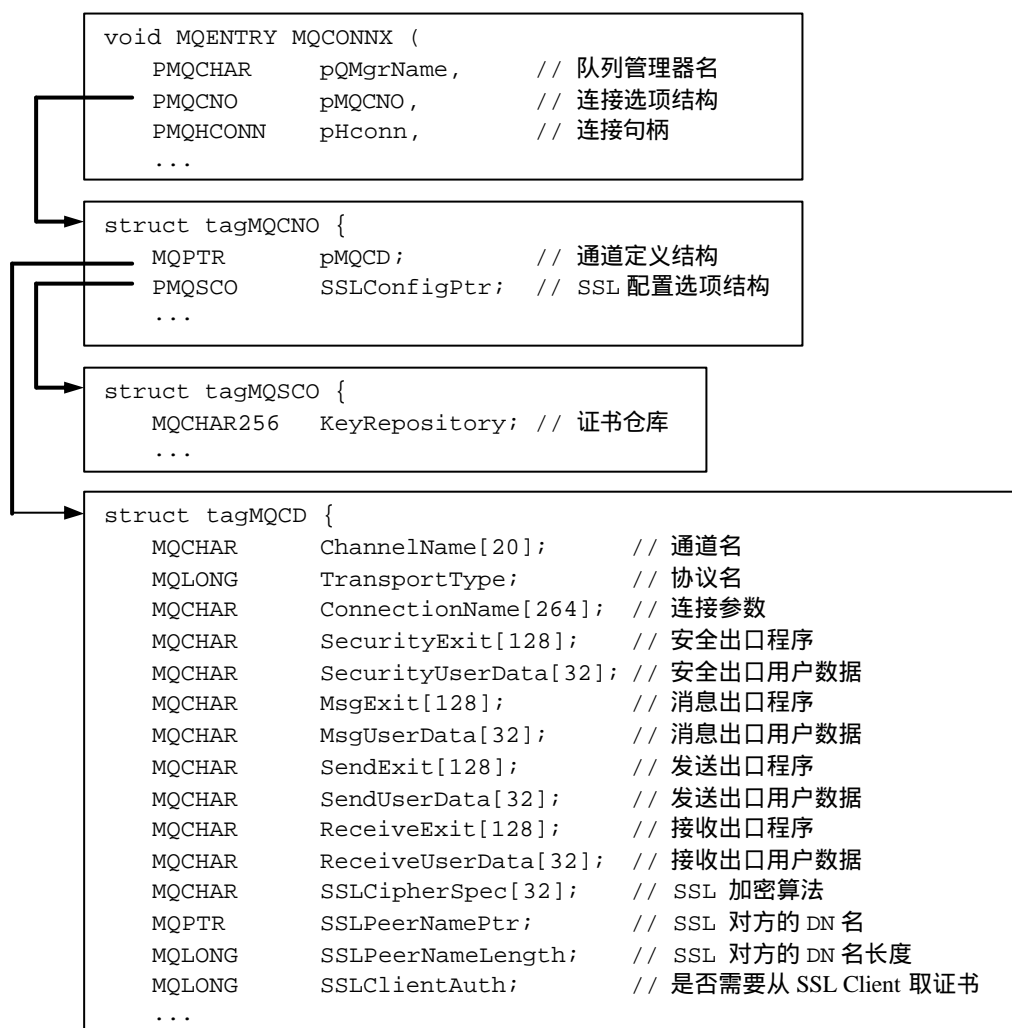


图 5-5 MQCONN 的相关参数与数据结构

具体实现，可参见例程 MQCONNClient.c 和 MQCONNSSLClient.c。

8.1.2.4 优先级

MQ Client 与 MQ Server 连接时需要最基本的参数：ChannelName、TransportType、ConnectionName，按优先级从高到低用以下方式取得：

1. 如果 Client 用 CONNX , 到 MQCNO.ClientConnOffset 或 MQCNO.ClientConnPtr 指向的 MQCD 结构中获取
2. 否则 , 如果 Client 有 MQSERVER 环境变量 , 则从 MQSERVER 中获取 (MQSERVER 的格式为 ChannelName/TransportType/ConnectionName)
3. 否则 , 如果 Client 有 MQCHLLIB 和 MQCHLTAB 环境变量 , 则从它们指定客户端通道定义表文件中获取
4. 如果以上皆不成立 , 则 Client 从配置项 DefaultPrefix 读取缺省的客户端通道定义表文件 :
 - UNIX
DefaultPrefix 缺省为 /var/mqm , 所以缺省定义表为 /var/mqm/AMQCLCHL.TAB
 - Windows
<MQ_HKEY>\Configuration\AllQueueManagers\DefaultPrefix 缺省为安装目录 <InstallDir> , 所以缺省定义表为 <InstallDir>\AMQCLCHL.TAB , 例 : D:\Program Files\IBM\WebSphere MQ\AMQCLCHL.TAB

8.2 用户出口

MQI 通道支持的用户出口种类有发送出口、接收出口、消息出口和安全出口。虽然支持的种类也不少 , 但 MQI 通道上的消息出口不能监视到流过的数据包内容 , 只能在客户端应用程序 MQCONN/MQCONN 调用时自动调用 MQXR_INIT 函数 , 在 MQDISC 时自动调用 MQXR_TERM 函数 , 可以进行一些跟踪记录 , 不如消息通道上的消息出口那样有意义。

MQI 通道上用户出口的使用在 Client 端与 Server 端不必对称。也就是说 , 可以仅在 Client 使用 , 也可以仅在 Server 端使用。

Client 端指定用户出口有两种方式 :

1. 在编程时指定通道参数。

```
MQCD.ChannelName      // 通道名字, 例: C_C.S
MQCD.TransportType    // 传输协议, 例: TCP
MQCD.ConnectionName   // 连接名字, 例: 169.254.163.161(1414)
MQCD.SecurityExit      // 例: MQChannelSecurityExit(MQChannelSecurityExit)
MQCD.SecurityUserData // 例: MQIClientSecurityExit.log
MQCD.SendExit          // 例: MQChannelSendExit(MQChannelSendExit)");
MQCD.SendUserData      // 例: MQIClientSendExit.log");
MQCD.ReceiveExit       // 例: MQChannelReceiveExit(MQChannelReceiveExit)");
MQCD.ReceiveUserData   // 例: MQIClientReceiveExit.log");
MQCD.MessageExit       // 例: MQChannelMessageExit(MQChannelMessageExit)");
MQCD.MessageUserData   // 例: MQIClientMessageExit.log");
```

2. 与 Server 端的使用方式类似 , 在 DEFINE CHANNEL 时指定 SCYEXIT, SCYDATA, SENDEXIT, SENDDATA, RCVEXIT, RCVDATA , 但是要将生成的通道定义表 (Client Channel Definition Table) 文件拷贝到 Client 端。

8.2.1 用户出口路径

在 UNIX 平台的 WebSphere MQ Client 中,用户出口程序的目录由 mqm.ini 配置文件中的 ClientExitPath 段指定。在 Windows 平台中,用户出口程序的目录由系统注册表指定。

- UNIX

AllQueueManagers:

DefaultPrefix=<DefaultPrefix>

ClientExitPath:

ExitsDefaultPath=<DefaultPrefix>/exits

- Windows

<MQ_HKEY>\Configuration\AllQueueManagers\DefaultPrefix

<MQ_HKEY>\Configuration\ClientExitPath\ExitsDefaultPath

出口程序编译时连接的库文件为 mqci32.lib (Windows) 或 libmqic_r.a (AIX), 编译所得的出口程序必须拷贝到 ExitsDefaultPath 中指定的目录下, 或者改变 ExitsDefaultPath 设置。

8.2.2 排错

如果用户出口在运行中出错, 一般来说, 其出错记录会出现在 WebSphere MQ 的系统记录中。对于 Client, 可以参考<InstallDir>/errors/AMQERRnn.LOG。对于 Server, 可以参考<InstallDir>/qmgrs/<QMgrName>/errors/AMQERRnn.LOG

8.3 安全检查

如同所有的通道一样, Client 与 Server 的 MQI 通道在连接的时候也有一个协商的过程 (MQXR_INIT_SEC), 也会通过数据包交换双方的参数。在安全出口中可以得到这个 MQCD 数据包。MQ Client 会根据以下的次序更新 MQCD 数据包, 后一步的结果覆盖前一步的结果。

1. 如果 Client 端存在通道定义表, 则表中的通道定义会被用来设置 MQCD
2. 如果 Client 端设置了 MQSERVER 环境变量, 则 MQSERVER 会覆盖 MQCD 中的 ChannelName、TransportType、ConnectionName 域
3. 如果 Client 端用 MQCONN, 对于 DOS, OS/2 Warp, Win3.1, Win95/98 平台, MQCD.RemoteUserIdentifier 和 MQCD.RemotePassword 会被环境变量 MQ_USER_ID, MQ_PASSWORD 覆盖。对于 UNIX, Win NT/2000/XP 平台, MQCD.MCAUserIdentifier 和 MQCD.RemoteUserIdentifier 取值相同, 都会被 Client 登录的用户名 (UNIX 或 Windows 平台下会用小写) 覆盖。
4. 如果 Client 端用 MQCONN, 则参数 MQCNO.ClientConnPtr 指向的 MQCD 结构会完全覆盖 MQCD 数据包。
5. 如果 Client 端用 MQCONN, 且指定了安全出口, 则该安全出口有机会在数据包发出前进一步更改 MQCD 的值
6. 数据包到了 Server 端。如果 Server 端通道定义中设置了 MCAUSER, 则 MQCD.MCAUserIdentifier 会被 MCAUSER 覆盖, 而不管 Client 端传来的字串。
7. 如果 Server 端通道定义中设置了安全出口, 则 Server 的安全出口可以随意更改

MQCD 的值，且对是否通过安全检查有最终决定权。

例：

Client 端 Win2000 (169.254.53.147) 用 amqspc (MQCONN)发送消息到 Server 端，不设安全出口，Client 端登录的用户为 Guest。

Server 端 WinXP (169.254.163.161(1414))，不设安全出口，通道定义为 DEFINE CHANNEL ... MCAUSER ('chenyux')。其中 chenyux 是 Administrators 组成员。

通过安全出口截获 MQCD 如下：

```
ChannelName      = [C_C.S]
TransportType    = [QXPT_TCP]
ShortConnectionName = [169.254.53.147]
ConnectionName   = [169.254.53.147]
QMGrName        = [QM]
LocalAddress     = [169.254.163.161(1414)]
MCAUserIdentifier = [chenyux]
RemoteUserIdentifier = [guest]
```

由于 Server 端的 chenyux 是 Administrators 组成员，所以可以通过安全检查。对于 UNIX 平台，通过检查的用户必须是 mqm 组成员，对于 Windows 平台，则必须是 mqm 组或 Administrators 组成员。

在 DOS、OS/2 Warp、Win3.1、Win95/98 平台上，如果不用安全出口，则 MQCD 中用户与口令的内容来自环境变量 MQ_USER_ID 和 MQ_PASSWORD。如果使用了安全出口，则可以任意设置 MQCD.RemoteUserIdentifier 和 MQCD.RemotePassword。

```
SET MQ_USER_ID=MYUSERID
SET MQ_PASSWORD=MYPASSWORD
```

在 UNIX, Win NT/2000/XP 中，不支持这种用环境变量指定用户名和口令的方式。Client 端登录时的用户会自动送入 MQCD.MCAUserIdentifier 中。

如果有 Server 端设置安全出口，也可以截获类似的信息，Server 端可以知道 Client 端的用户名、IP 地址等信息，从而在应用层决定是否通过安全检查。

两个比较简单的通过安全检查方式是：

1. Client 用 MQCONN 而不设置任何用户信息。这时 Client 端的 MQCD.MCAUserIdentifier 为空，到了 Server 端后用启动 Server MCA 的用户（比如 mqm）进行身份检查。通常情况下，能够启动 Server 端 MCA 的用户（即启动队列管理器的用户）应该可以通过安全检查。
2. Server 通道定义使用 MCAUSER 选项。这时，无论在 Client 端 MQCD.MCAUserIdentifier 用户信息被设置成什么，到了 Server 端后都会被通道的 MCAUSER 属性指定的用户名覆盖。由于该用户是 Server 用户，WebSphere MQ 会试图在用户中检查。UNIX 中需要是 mqm 组成员，Windows 中需要是 mqm 组或 Administrators 组成员。如果在 Windows 本地用户中查不到，会试图去主域服务器 (Primary Domain Server) 上进一步查找。如果用户名格式为 user@domain，则会去域中查找。

8.4 触发 (Trigger)

在“控制与管理”章节中我们介绍过客户端的监控器为 runmqtmc，在 Client 端运行。如下。当然，运行的环境中也需要设置 MQSERVER 环境变量或客户端通道定义表。

```
// 启动 Client 端 Trigger Monitor
runmqtmc [-m QMgrName] [-q InitQ]
// 例：
runmqtmc -m QM -q SYSTEM.DEFAULT.INITIATION.QUEUE
```

编写 Client 端的 Trigger Monitor 编程类似于普通的应用程序，也要连接 mqic 库，而不是 Server 端的 mqm 库。Client 端触发器的配置仍然在 Server 端进行，方式也与配置 Server 端触发相同。只是要注意在 Server 端设置 PROCESS 时 APPLICID 和 APPLTYPE 应该对应于 Client 端的触发应用和平台。例：

```
QM:
----
DEFINE QLOCAL          (Q)                                +
      TRIGGER                                +
      TRIGTYPE          (EVERY)                    +
      PROCESS           (P_NOTEPAD)                  +
      INITQ             (SYSTEM.DEFAULT.INITIATION.QUEUE) +
      REPLACE
DEFINE PROCESS         (P_NOTEPAD)                    +
      APPLICID          ('C:/WINNT/system32/notepad.exe') +
      APPLTYPE          (WINDOWSNT)                  +
      USERDATA          ('C:/WINNT/win.ini')          +
      ENVRDATA          ('C:/WINNT/win.ini')          +
      REPLACE
// 其中，APPLTYPE 的值：
// WINDOWS             表示 Windows 3.1 Client 或 Win16 应用
// WINDOWSNT            表示 Windows Client 或 Win32 应用
// UNIX                 表示 UNIX Client
```

Client 端触发的原理和 Server 端是一样的，在消息满足触发条件的时候会自动产生一条触发消息放入初始化队列中，Trigger Monitor 从初始化队列中将触发消息读出，启动相应的程序。只是 Server 端的 Trigger Monitor 在 Server 端运行，读到消息后启动的是 Server 端程序，而 Client 端的 Trigger Monitor 在 Client 端运行，启动的是 Client 端程序。如果我们同时启动 Client 端和 Server 端的 Trigger Monitor (runmqtmc 和 runmqtrm)，则只有其中一个可以读到这条触发消息，并各自启动它们理解中的应用程序。

8.5 跟踪 (Trace)

WebSphere MQ Client 可以通过打开跟踪 (Trace) 开关，获得大量的跟踪数据，从而获得 Client 应用程序甚至是 Client 运行环境的执行记录。对于不同的平台，打开跟踪开关的方

法是不同的。

8.5.1 Windows

对于 Win95/98/NT/2000/XP 或 OS/2 平台，在 strmqtrc 和 endmqtrc 两条命令之间的所以 Client 操作都会被记录下来，并写入 <InstallDir>\Errors\AMQppppp.TRC 文件中，ppppp 为 Client 应用进程的进程号。该文件是文本文件，可以打开浏览。

```
strmqtrc [-t TraceType]          // Start Trace
endmqtrc                          // End Trace
```

8.5.2 AIX

AIX 操作系统本身提供的系统的 trace 工具，WebSphere MQ 提供了两个 Trace Hook ID：

- X'30D' // 出入函数记录
- X'30E' // 网络上收发的数据

可以利用系统提供的 trace 工具，将 MQ Client 纳入。具体说来，有同步和异步两种方式：

1. 同步打开 trace

```
trace -j30D,30E -o trace.trc      // 进入 trace 环境
!amqsputc TESTQ                  // ! 表示后面的命令需要记入 trace
    Hello, world!                 // amqsputc 的输入
Quit                             // 退出 trace 环境
```

2. 异步打开 trace

```
trace -a -j30D,j30E -o trace.trc  // 启动 trace, 以后所有命令都会被 trace
amqsputc TESTQ                    // amqsputc 命令
    Hello, world!                 // amqsputc 的输入
Trstop                           // 停止 trace
```

生成了 trace 二进制文件后（假定为 trace.trc），需要将其转换成文本文件（假定为 trace.fmt），才能进行浏览。

```
trcrpt -t /usr/mqm/lib/amqtrc.fmt trace.trc > trace.fmt
```

第 9 章 群集

很多操作系统与平台软件都提供群集（Cluster）功能，虽然不同软件对“群集”一词的理解各不相同，但大致意思是：由多机系统共同担负计算任务，如果其中一个结点坏了，剩余的部分会接管其工作。从本质上讲，群集要解决的是系统容错与负载均衡这两个问题。WebSphere MQ Cluster 也是为解决这些问题而设计的，具体说来，使用 WebSphere MQ 群集有以下的好处：

1. 跨平台多结点计算

由于 WebSphere MQ 本身是跨平台的中间件,它可以将不同操作系统上的队列管理器组织在一个群集中共同计算,最大限度地利用现有计算资源

2. 系统容错

在 WebSphere MQ 中如果有一个结点坏了,即该队列管理器不工作了,则其它队列管理器与它的通道就可能断开。一旦发现这种问题,就不会再有新的消息路由去该结点了,群集会自动将其隔离,它的任务由其它结点分摊。

3. 负载均衡

缺省情况下多结点以轮循的方式分配消息。也可以编写用户出口,植入自己的分配算法。

4. 大大简化复杂系统中的通道配置

以 N 个队列管理器全连通结构为例:如果不用群集,共需要定义 $(N-1) * N$ 个远程队列、传输队列、发送通道、接收通道。如果使用群集,共需要定义 N 个群集发送通道、群集接收通道,如果队列管理器自身又是完全配置库 (Full Repository),则它的群集发送通道定义也可以省了。

9.1 相关概念

群集中可能含有多个队列管理器,是一个分布式的系统,但群集的结构及对象配置信息则集中地存放在群集中的某个地方,这个地方就称配置库,配置库所在的队列管理器称为配置库队列管理器。如果配置库信息是完整的,则称为完整配置库,否则称为部分配置库。相应地,配置库队列管理器也是完整和部分之分。在群集队列管理器中共享的对象称为群集对象,比如群集通道、群集队列等等。

9.1.1 配置库 (Repository)

配置库 (Repository) 是群集的配置信息集合,包括群集中的队列管理器名,它们之间的拓扑结构、通道、队列其相关配置。这些信息存于群集队列管理器的 `SYSTEM.CLUSTER.REPOSITORY.QUEUE` 中。配置库又分成完全配置库 (Full Repository) 和部分配置库 (Partial Repository) 两种。前者拥有整个群集的完整配置,后者只有部分配置,需要从前者处获得最新的更新信息。

9.1.2 配置库队列管理器 (Repository Queue Manager)

在配置群集的时候,可以将队列管理器配置成配置库队列管理器 (Repository Queue Manager)。该队列管理器拥有整个群集的完整配置信息库 (Full Repository),在群集中处于中心地位。其它参与群集的队列管理器可以与其通过群集通道保持联系,自动获得更新信息。发送和查询这些信息是通过 `SYSTEM.CLUSTER.COMMAND.QUEUE` 完成的。

一个群集中可以有多个配置库队列管理器,它们之间通过密切的联系以保持同步。为了控制网络开销同时具有较高的可用性,一般可以配置两个配置库队列管理器,互为备份。

9.1.3 群集通道 (Cluster Channel)

群集通道分成群集发送通道和群集接收通道，用于群集信息其应用消息的传送。缺省配置可以参考 SYSTEM.DEF.CLUSSDR 和 SYSTEM.DEF.CLUSRCVR。

- 群集发送通道 (Cluster-Sender)
用来向群集中其它队列管理器发送应用消息和有关群集配置更新的消息。
- 群集接收通道 (Cluster-Receiver)
用来从群集中其它队列管理器接收应用消息和有关群集配置更新的消息。

事实上，群集接收通道在与对方的群集发送通道相连后，会自动建立反向的通道，只要对方也建立了相应的群集接收通道即可连上。所以，配置库队列管理器可以只定义群集接收通道，以备连接。普通队列管理器需要定义群集发送通道连接配置库，同时需要定义群集接收通道，以备反向连接。

9.1.4 群集队列 (Cluster Queue)

群集队列的概念有点像 Windows 中的共享文件夹。队列属于宿主队列管理器，一旦变成群集队列后，在整个群集中可见。其它群集成员可以向其发送消息，而无需远程队列定义。

不同的群集队列管理器可以定义同名的群集队列，这样在群集中可以看到多个同名群集队列实体，这就形成了多实体 (Multi-Instance) 环境。送往这些同名实体的消息可以轮流地落入这些队列中，从而达到负载均衡的效果。但它的应用前提是，消息路径上没有同名的本地队列，否则该本地队列有优先权，消息皆会落入其中而停止进一步路由。

9.1.5 群集传输队列 (Cluster transmission queue)

群集中的每个队列管理器都有一个名为 SYSTEM.CLUSTER.TRANSMIT.QUEUE 的传输队列，用来传送应用消息和群集配置信息。

9.2 群集管理

9.2.1 对象属性

WebSphere MQ 对象中与群集相关的属性并不是太多，集中在队列管理器、队列、通道这三个最基本也是最重要的对象上。设置群集相关属性基本上设置对象的群集归属，比如对象所属的群集名称。对象也可以同时属于多个群集，这样就需要首先将多个群集名放入某个名称列表，再设置该对象的群集列表属性。以下是对象中与群集相关的属性：

- 队列管理器
 - REPOS 队列管理器所属的群集名字
 - REPOSNL 如果队列管理器同时身处多个群集之中，则需要名称列表 (Name List) 来指明这些群集
- 队列

- CLUSTER 队列共享的群集名字
- CLUSNL 如果队列同时在多个群集中共享，则需要名称列表 (Name List)来指明这些群集
- DEFBIND 队列的缺省绑定方式，可取 OPEN 或 NOTFIXED。当 MQOPEN (MQOO_BIND_AS_Q_DEF) 时，绑定方式取 DEFBIND 值。
- 通道
 - CLUSTER 通道共享的群集名字
 - CLUSNL 如果通道同时在多个群集中共享，则需要名称列表 (Name List)来指明这些群集
 - NETPRTY 网络连接优先级 (Network-Connection Priority)，可以取值 0-9。如果存在多条路径到达目标队列的情况下，系统会选取 NETPRTY 高的通道传送。

9.2.2 管理命令

群集由队列管理器组成，相关的信息存放在群集配置库中，群集的管理命令就是对群集中的队列管理器和配置库进行管理。可以挂起或恢复其中的某个队列管理器，刷新配置信息等等。

● 显示群集中队列管理器信息

DISPLAY CLUSQMGR (QMgrName) [CLUSTER (ClusterName)] [CHANNEL (ChannelName)]
如果 QMgrName 为 “*”，则显示群集所含的队列管理器名

● 挂起群集中队列管理器

SUSPEND QMGR CLUSTER (ClusterName) [MODE (QUIESCE | FORCE)]
SUSPEND QMGR CLUSNL (NamelistName) [MODE (QUIESCE | FORCE)]

连接群集中某个队列管理器，用 SUSPEND 命令通知群集或群集列表中的其它队列管理器该结点被挂起，即暂时不可用。这时该队列管理器仍属于群集，要用 RESET 才能将其移出群集。QUIESCE 模式表示通知其它队列管理器不再送来消息，FORCE 表示断开其它队列管理器之间到该结点的通道连接。

● 恢复挂起的队列管理器

SUSPEND QMGR CLUSTER (ClusterName)
SUSPEND QMGR CLUSNL (NamelistName)

这是 SUSPEND 的逆过程。通知群集或群集列表中的其它队列管理器该结点被重新启用。

● 刷新群集信息

REFRESH CLUSTER (ClusterName) [REPOS (NO | YES)]

该命令都首先删除所有本地已有的群集信息，然后通过与附近的配置库队列管理器交换信息来重建本地的群集信息。

REPOS (NO) 表示保留本地群集信息中的配置库队列管理器名，如果该结点自身就是配置库队列管理器，则会记住群集中的其它配置库队列管理器，其余的信息都被删除重建。通

常用于队列管理器上的群集信息发生混乱时手工更新。

REPOS (YES) 表示在本地群集信息中连群集中配置库队列管理器名也被删除重建。用于非配置库队列管理器。如果该结点自身就是配置库队列管理器，则不能用 REPOS (YES)，应该先将结点改成非配置库，刷新后，再改回来。通常用于群集中有配置库队列管理器的地位发生变化，需要手工更新。

- 从群集中删除队列管理器

`RESET CLUSTER (ClusterName)] ACTION (FORCEREMOVE) QMNAME (string) | QMID (string) [QUEUES (NO | YES)]`

用于配置库队列管理器，将指定的队列管理器从群集配置信息中删除。QUEUES (NO) 表示在群集配置信息中保留被删除队列管理器上的共享队列，QUEUES (YES) 表示从群集配置信息中删除这些共享队列。

9.2.3 管理任务举例

通过群集管理命令的组合使用我们可以进行一些复杂的群集配置工作，下面我们介绍几个常用的管理任务。

- 向群集中增加队列管理器

1. 创建队列管理器与配置库队列管理器之间的 CLUSSDR 和 CLUSRCVR 通道
2. 启动队列管理器的监听器 (Listener)，稍等片刻通道会自动连上，或者手工启动 CLUSSDR 通道

- 从群集中删除队列管理器

1. 将队列管理器在群集中挂起
`SUSPEND QMGR CLUSTER (ClusterName)`
2. 将群集接收通道隔离出群集，以防有新的群集连接
`ALTER CHANNEL (CLUSRCVRChannelName) CHLTYPE (CLUSRCVR) CLUSTER ('')`
3. 停止并删除群集接收通道，可以用以下命令：
`STOP CHANNEL (CLUSRCVRChannelName)`
`DELETE CHANNEL (CLUSRCVRChannelName)`
4. 停止并删除群集发送通道，可以用以下命令：
`STOP CHANNEL (CLUSSDRChannelName)`
`DELETE CHANNEL (CLUSSDRChannelName)`

- 从群集中删除群集共享队列

1. 将队列隔离出群集
`ALTER QLOCAL (QueueName) CLUSTER ('')`
2. 不允许消息继续进入
用 `ALTER QLOCAL (QueueName) PUT (DISABLED)`
3. 等待队列中消息被取空，等待不再有打开该队列的应用程序，等待其它队列管理器上指向该队列管理器的 CLUSSDR 通道不存在尚未提交的消息
`DISPLAY QLOCAL (QueueName) IPPROCS OPPROCS CURDEPTH`

```
DISPLAY CHSTATUS (CLUSSDRChannelName) INDOUBT
```

4. 将队列删除

```
DELETE QLOCAL (QueueName)
```

9.3 群集配置举例

9.3.1 例 1

Cluster 名为 SHINE，其中 QM1 为 Repository Queue Manager。图中方向箭标表示群集发送通道。Cluster 中的每一个队列管理器（包括普通的 QM 和 Repository QM）至少有一个群集接收通道 (Cluster Receiver Channel)。普通的队列管理器至少有一个群集发送通道 (Cluster Sender Channel) 指向某个配置库 (Repository Queue Manager)。图 9-1。

连接后，QM1 会自动建立反向通道到 QM2 和 QM3 上，届时用 display chstatus (*) 命令可以观察到。

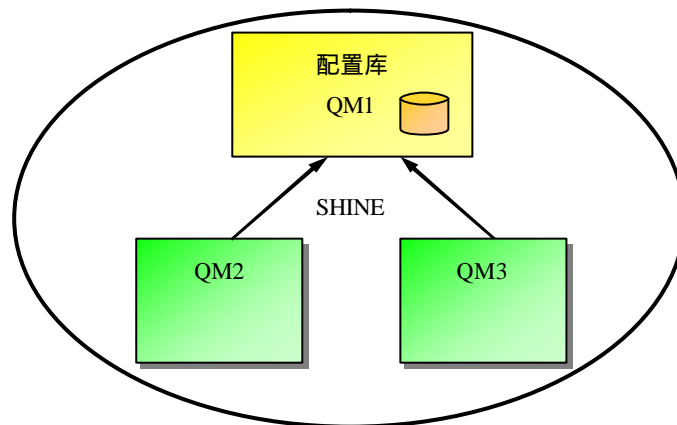


图 9-1 单配置库群集

QM1:

```
ALTER    QMGR REPOS (SHINE)
DEFINE    QLOCAL      (QL_QM1)                +
          CLUSTER     (SHINE)                  +
          REPLACE
DEFINE    CHANNEL      (C_QM1)                 +
          CHLTYPE      (CLUSRCVR)              +
          TRPTYPE      (TCP)                   +
          CONNAME      ('127.0.0.1 (1414)')     +
          CLUSTER      (SHINE)                  +
          REPLACE
```

//

// 以下可选，只要队列管理器定义了 Cluster Sender Channel 指向配置库队列管理器，

// WebSphere MQ 会自动为 Repository QM 建立反向的 Cluster Sender Channel.

```
// DEFINE CHANNEL      (C_QM2)                 +
```

```
//          CHLTYPE      (CLUSSDR)              +
```

```

//      TRPTYPE      (TCP)                +
//      CONNAME      ('127.0.0.1 (1415)')  +
//      CLUSTER      (SHINE)              +
//      REPLACE
// DEFINE CHANNEL    (C_QM3)              +
//      CHLTYPE      (CLUSSDR)            +
//      TRPTYPE      (TCP)                +
//      CONNAME      ('127.0.0.1 (1416)')  +
//      CLUSTER      (SHINE)              +
//      REPLACE

```

QM2:

```

DEFINE  QLOCAL      (QL_QM2)              +
        CLUSTER     (SHINE)              +
        REPLACE
DEFINE  CHANNEL      (C_QM2)              +
        CHLTYPE     (CLUSRCVR)           +
        TRPTYPE     (TCP)                +
        CONNAME     ('127.0.0.1 (1415)')  +
        CLUSTER     (SHINE)              +
        REPLACE
DEFINE  CHANNEL      (C_QM1)              +
        CHLTYPE     (CLUSSDR)            +
        TRPTYPE     (TCP)                +
        CONNAME     ('127.0.0.1 (1414)')  +
        CLUSTER     (SHINE)              +
        REPLACE

```

QM3:

```

DEFINE  QLOCAL      (QL_QM3)              +
        CLUSTER     (SHINE)              +
        REPLACE
DEFINE  CHANNEL      (C_QM3)              +
        CHLTYPE     (CLUSRCVR)           +
        TRPTYPE     (TCP)                +
        CONNAME     ('127.0.0.1 (1416)')  +
        CLUSTER     (SHINE)              +
        REPLACE
DEFINE  CHANNEL      (C_QM1)              +
        CHLTYPE     (CLUSSDR)            +
        TRPTYPE     (TCP)                +
        CONNAME     ('127.0.0.1 (1414)')  +

```

```
CLUSTER      (SHINE)
REPLACE
```

9.3.2 例 2

Cluster 名为 SHINE，含两个配置库队列管理器 QM1 和 QM2。普通队列管理器 QM3 与 QM1 相连，QM4 与 QM2 相连。(图 9-2)。连接后，配置库会首先建立反向通道：QM1 → QM3，QM2 → QM4。接着，两个配置库会交换信息，QM3 和 QM4 也会得到更新信息，得知还有其它的队列管理器存在，会自动进一步建立通道 QM1 ↔ QM4，QM2 ↔ QM3。最终形成全连通结构。届时，可以通过 display chstatus (*) 命令可以观察到。

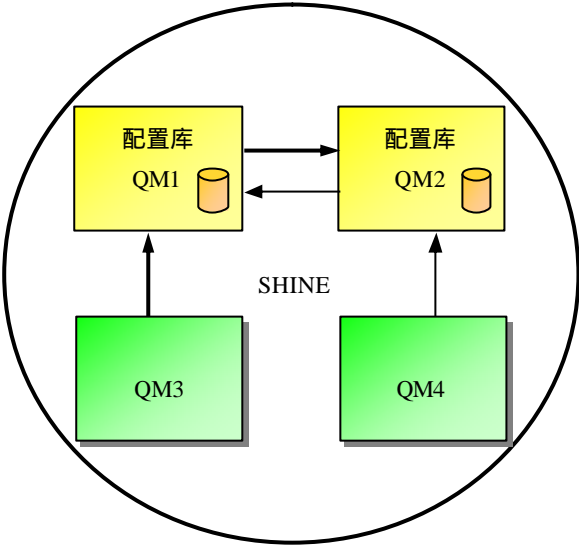


图 9-2 双配置库群集

```
QM1:
----
ALTER    QMGR REPOS (SHINE)
DEFINE   QLOCAL      (QL_QM1)
CLUSTER  (SHINE)
REPLACE

DEFINE   CHANNEL      (C_QM1)
CHLTYPE  (CLUSRCVR)
TRPTYPE  (TCP)
CONNAME  ('127.0.0.1 (1414)')
CLUSTER  (SHINE)
REPLACE

DEFINE   CHANNEL      (C_QM2)
CHLTYPE  (CLUSSDR)
TRPTYPE  (TCP)
CONNAME  ('127.0.0.1 (1415)')
CLUSTER  (SHINE)
REPLACE
```

QM2:

ALTER	QMGR REPOS	(SHINE)	
DEFINE	QLOCAL	(QL_QM2)	+
	CLUSTER	(SHINE)	+
	REPLACE		
DEFINE	CHANNEL	(C_QM2)	+
	CHLTYPE	(CLUSRCVR)	+
	TRPTYPE	(TCP)	+
	CONNAME	('127.0.0.1 (1415)')	+
	CLUSTER	(SHINE)	+
	REPLACE		
DEFINE	CHANNEL	(C_QM1)	+
	CHLTYPE	(CLUSSDR)	+
	TRPTYPE	(TCP)	+
	CONNAME	('127.0.0.1 (1414)')	+
	CLUSTER	(SHINE)	+
	REPLACE		

QM3:

DEFINE	QLOCAL	(QL_QM3)	+
	CLUSTER	(SHINE)	+
	REPLACE		
DEFINE	CHANNEL	(C_QM3)	+
	CHLTYPE	(CLUSRCVR)	+
	TRPTYPE	(TCP)	+
	CONNAME	('127.0.0.1 (1416)')	+
	CLUSTER	(SHINE)	+
	REPLACE		
DEFINE	CHANNEL	(C_QM1)	+
	CHLTYPE	(CLUSSDR)	+
	TRPTYPE	(TCP)	+
	CONNAME	('127.0.0.1 (1414)')	+
	CLUSTER	(SHINE)	+
	REPLACE		

QM4:

DEFINE	QLOCAL	(QL_QM4)	+
	CLUSTER	(SHINE)	+
	REPLACE		
DEFINE	CHANNEL	(C_QM4)	+
	CHLTYPE	(CLUSRCVR)	+

	TRPTYPE	(TCP)	+
	CONNAME	('127.0.0.1 (1417)')	+
	CLUSTER	(SHINE)	+
	REPLACE		
DEFINE	CHANNEL	(C_QM2)	+
	CHLTYPE	(CLUSSDR)	+
	TRPTYPE	(TCP)	+
	CONNAME	('127.0.0.1 (1415)')	+
	CLUSTER	(SHINE)	+
	REPLACE		

9.3.3 例 3

Cluster 名为 SHINE，包括 QM1 和 QM2。其中 QM1 为配置库队列管理器。Cluster 外的 QM3 由于平台不支持 Cluster，只能通过传统的方式 (Sender/Receiver 通道) 接入，QM2 扮演 Gateway 的角色 (图 9-3)。使得群集中的 QM1 和 QM2 可以送消息到 QM3，反过来，QM3 也可以送消息到 QM1 和 QM2。这时，作为群集网关的 QM2 可以是 Cluster 中的配置库队列管理器，也可以是普通队列管理器。

另外 ,为了满足消息的自动应答功能 ,比如报告消息 (Report) ,需要在 QM3 上创建 QM1 和 QM2 的队列管理器别名，以便使 QM3 上的回复消息知道进入群集的路由。同样，需要在 QM2 上创建 QM3 的队列管理器别名，并将其在群集中共享，以便使群集中的回复消息知道出去的路由。

可以用 MQPutCOA 例程测试：

```
MQPutCOA QR_QM3 QM1 QL_QM1 QM1
MQPutCOA QR_QM3 QM2 QL_QM2 QM2
MQPutCOA QR_QM1 QM3 QL_QM3 QM3
MQPutCOA QR_QM2 QM3 QL_QM3 QM3
```

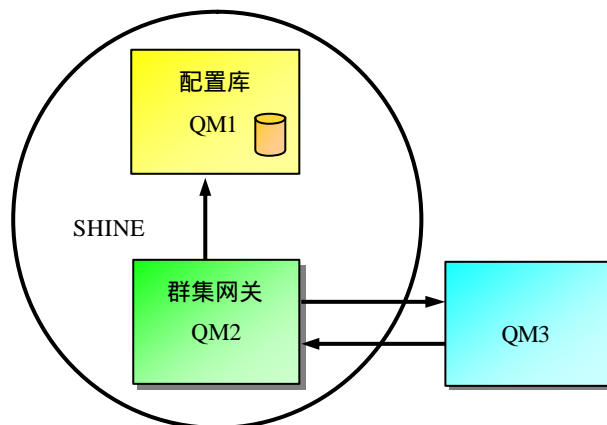


图 9-3 带网关的群集

QM1:

ALTER	QMGR REPOS	(SHINE)	
DEFINE	QLOCAL	(QL_QM1)	+
	CLUSTER	(SHINE)	+

	REPLACE		
DEFINE	CHANNEL	(C_QM1)	+
	CHLTYPE	(CLUSRCVR)	+
	TRPTYPE	(TCP)	+
	CONNAME	('127.0.0.1 (1414)')	+
	CLUSTER	(SHINE)	+
	REPLACE		
QM2:			

DEFINE	QREMOTE	(QM3)	+
	RNAME	('')	+
	RQMNAME	(QM3)	+
	XMITQ	(QX_QM3)	+
	CLUSTER	(SHINE)	+
	REPLACE		
DEFINE	QREMOTE	(QR_QM3)	+
	RNAME	(QL_QM3)	+
	RQMNAME	(QM3)	+
	XMITQ	(QX_QM3)	+
	CLUSTER	(SHINE)	+
	REPLACE		
DEFINE	QLOCAL	(QL_QM2)	+
	CLUSTER	(SHINE)	+
	REPLACE		
DEFINE	CHANNEL	(C_QM2)	+
	CHLTYPE	(CLUSRCVR)	+
	TRPTYPE	(TCP)	+
	CONNAME	('127.0.0.1 (1415)')	+
	CLUSTER	(SHINE)	+
	REPLACE		
DEFINE	CHANNEL	(C_QM1)	+
	CHLTYPE	(CLUSSDR)	+
	TRPTYPE	(TCP)	+
	CONNAME	('127.0.0.1 (1414)')	+
	CLUSTER	(SHINE)	+
	REPLACE		
DEFINE	CHANNEL	(C_QM2.QM3)	+
	CHLTYPE	(SDR)	+
	TRPTYPE	(TCP)	+
	CONNAME	('127.0.0.1 (1416)')	+
	XMITQ	(QX_QM3)	+
	REPLACE		
DEFINE	CHANNEL	(C_QM3.QM2)	+

	CHLTYPE	(RCVR)	+
	TRPTYPE	(TCP)	+
	REPLACE		
DEFINE	QLOCAL	(QX_QM3)	+
	USAGE	(XMITQ)	+
	REPLACE		
QM3:			

DEFINE	QREMOTE	(QM1)	+
	RNAME	('')	+
	RQMNAME	(QM1)	+
	XMITQ	(QX_QM2)	+
	REPLACE		
DEFINE	QREMOTE	(QM2)	+
	RNAME	('')	+
	RQMNAME	(QM2)	+
	XMITQ	(QX_QM2)	+
	REPLACE		
DEFINE	QREMOTE	(QR_QM1)	+
	RNAME	(QL_QM1)	+
	RQMNAME	(QM1)	+
	XMITQ	(QX_QM2)	+
	REPLACE		
DEFINE	QREMOTE	(QR_QM2)	+
	RNAME	(QL_QM2)	+
	RQMNAME	(QM2)	+
	XMITQ	(QX_QM2)	+
	REPLACE		
DEFINE	QLOCAL	(QL_QM3)	+
	REPLACE		
DEFINE	CHANNEL	(C_QM2.QM3)	+
	CHLTYPE	(RCVR)	+
	TRPTYPE	(TCP)	+
	REPLACE		
DEFINE	CHANNEL	(C_QM3.QM2)	+
	CHLTYPE	(SDR)	+
	TRPTYPE	(TCP)	+
	CONNAME	('127.0.0.1 (1415)')	+
	XMITQ	(QX_QM2)	+
	REPLACE		
DEFINE	QLOCAL	(QX_QM2)	+
	USAGE	(XMITQ)	+
	REPLACE		

9.3.4 例 4

QM1 同时属于两个群集 CLUSTER1 和 CLUSTER2 ,两个群集中的队列管理器需要互通消息。QM1 扮演群集网桥 (Bridge) 的角色 (图 9-4)。由于 QM1 地位特殊，如果在其上定义远程队列，指向 QM2 和 QM3 中的本地队列，再将这些远程队列在各自的群集中共享，则 CLUSTER1 中的 QM1 和 QM2 都能发送消息去 QM3。反过来，也一样，CLUSTER2 中的 QM1 和 QM3 都能发送消息去 QM2。这里 QM1 是不是配置库，并不重要。

如果在 QM1 (Bridge) 上定义队列管理器别名，则它们在对方的群集中可见。这样可以满足消息的自动应答功能，比如报告消息 (Report)。

可以用 MQPutCOA 例程测试：

```
MQPutCOA QR_QM2 QM3 QL_QM3 QM3
MQPutCOA QR_QM3 QM2 QL_QM2 QM2
```

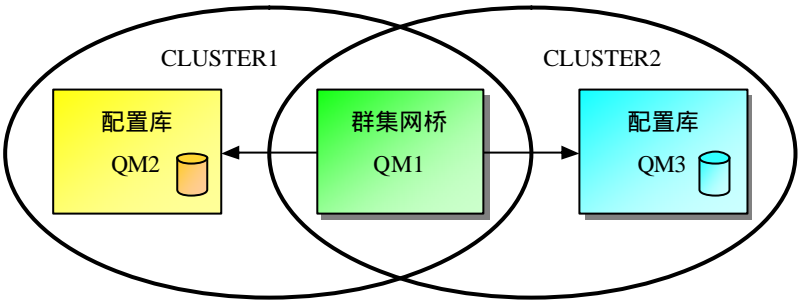


图 9-4 用群集网桥连接两个群集

QM1:

```
DEFINE  NAMELIST      (CLUSTER_NL)                +
        NAMES        (CLUSTER1, CLUSTER2)         +
        REPLACE
DEFINE  QLOCAL        (QL_QM1)                    +
        CLUSNL       (CLUSTER_NL)                 +
        REPLACE
DEFINE  QREMOTE       (QR_QM2)                    +
        RNAME        (QL_QM2)                    +
        RQMNAME      (QM2)                      +
        CLUSTER      (CLUSTER2)                 +
        REPLACE
DEFINE  QREMOTE       (QR_QM3)                    +
        RNAME        (QL_QM3)                    +
        RQMNAME      (QM3)                      +
        CLUSTER      (CLUSTER1)                 +
        REPLACE
DEFINE  CHANNEL       (C_QM1)                    +
        CHLTYPE      (CLUSRCVR)                  +
        TRPTYPE      (TCP)                      +
        CONNAME      ('127.0.0.1 (1414)')         +
        CLUSNL       (CLUSTER_NL)                +
```

```

REPLACE
DEFINE CHANNEL (C_QM2) +
        CHLTYPE (CLUSSDR) +
        TRPTYPE (TCP) +
        CONNAME ('127.0.0.1 (1415)') +
        CLUSTER (CLUSTER1) +
REPLACE
DEFINE CHANNEL (C_QM3) +
        CHLTYPE (CLUSSDR) +
        TRPTYPE (TCP) +
        CONNAME ('127.0.0.1 (1416)') +
        CLUSTER (CLUSTER2) +
REPLACE
DEFINE QREMOTE (QM2) +
        RNAME (') +
        RQMNAME (QM2) +
        CLUSTER (CLUSTER2) +
REPLACE
DEFINE QREMOTE (QM3) +
        RNAME (') +
        RQMNAME (QM3) +
        CLUSTER (CLUSTER1) +
REPLACE

QM2:
----
ALTER QMGR REPOS (CLUSTER1)
DEFINE QLOCAL (QL_QM2) +
        CLUSTER (CLUSTER1) +
REPLACE
DEFINE CHANNEL (C_QM2) +
        CHLTYPE (CLUSRCVR) +
        TRPTYPE (TCP) +
        CONNAME ('127.0.0.1 (1415)') +
        CLUSTER (CLUSTER1) +
REPLACE

QM3:
----
ALTER QMGR REPOS (CLUSTER2)
DEFINE QLOCAL (QL_QM3) +
        CLUSTER (CLUSTER2) +
REPLACE
DEFINE CHANNEL (C_QM3) +

```

CHLTYPE	(CLUSRCVR)	+
TRPTYPE	(TCP)	+
CONNNAME	('127.0.0.1 (1416)')	+
CLUSTER	(CLUSTER2)	+
REPLACE		

9.3.5 例 5

群集 SHINE 中有两个同名群集队列 QL ,我们称这为多队列实例 (Multi Queue Instance)。这时群集中的其它队列管理器 (比如 QM1) 会看见两个 QL (图 9-5), 连接其上的应用程序在 MQOPEN 时 如果 选项 为 MQOO_BIND_NOT_FIXED , 或者 选项 MQOO_BIND_AS_Q_DEF 且 QL 的属性 DEFBIND (NOTFIXED) ,那么打开队列时并不会指定任何一个 QL。在 MQPUT 的时候消息会轮流路由去两个 QL ,这样就达到了负载均衡的效果。另外, 多个应用的首发消息也会轮流路由去两个 QL。这样, 进一步避免了大量并发轮流轰击两个 QL 的局面, 即所有应用的第一条消息去 QM2, 然后, 第二条消息去 QM3... 此外, 如果有一个 QL 所在的队列管理器停了, 则以后的消息会自动全部路由去另一个 QL ,这样在某种程度上达到了系统容错的效果。

群集中 QM1、QM2、QM3 中哪一个是配置库并不重要, 效果是相同的。但是, 对于发送消息的队列管理器有一个要求, 其上不能有同名的本地队列。也就是说, QM1 上不能再有 QL, 否则消息缺省都会直接放入本地的 QL 上。扩展开去, 如果消息来群集外且经历多次转发, 则消息路径上不能有同名的本地队列, 否则消息在途经中间某一环节的时候, 会直接放入本地队列。

多队列实例的环境中, 如果应用程序希望连接发送的消息都路由去同一个队列 QL , 这种需求我们称为消息亲合 (Message Affinity), 那么就需要在 MQOPEN 时选项设置为 MQOO_BIND_ON_OPEN。这会使队列打开时选择一个实例进行绑定, 以后所有的 MQPUT 消息都会路由去那个队列, 从而使连续的相关消息去同一个目标队列。

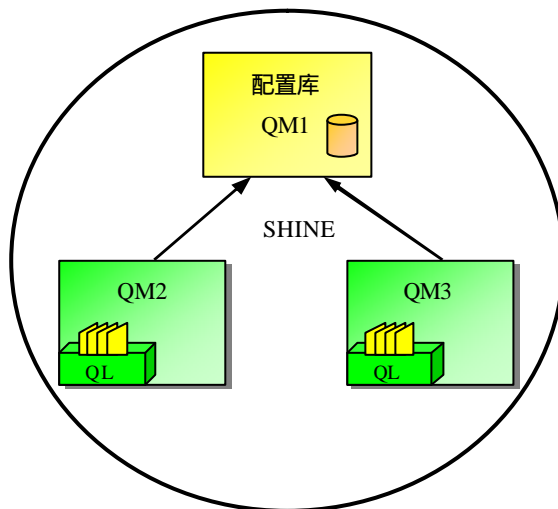


图 9-5 利用群集实现负载均衡

QM1:

ALTER	QMGR REPOS (SHINE)	
DEFINE	CHANNEL (C_QM1)	+

CHLTYPE	(CLUSRCVR)	+
TRPTYPE	(TCP)	+
CONNAME	('127.0.0.1 (1414)')	+
CLUSTER	(SHINE)	+
REPLACE		

QM2:

DEFINE	QLOCAL	(QL)	+
	CLUSTER	(SHINE)	+
	REPLACE		
DEFINE	CHANNEL	(C_QM2)	+
	CHLTYPE	(CLUSRCVR)	+
	TRPTYPE	(TCP)	+
	CONNAME	('127.0.0.1 (1415)')	+
	CLUSTER	(SHINE)	+
	REPLACE		
DEFINE	CHANNEL	(C_QM1)	+
	CHLTYPE	(CLUSSDR)	+
	TRPTYPE	(TCP)	+
	CONNAME	('127.0.0.1 (1414)')	+
	CLUSTER	(SHINE)	+
	REPLACE		

QM3:

DEFINE	QLOCAL	(QL)	+
	CLUSTER	(SHINE)	+
	REPLACE		
DEFINE	CHANNEL	(C_QM3)	+
	CHLTYPE	(CLUSRCVR)	+
	TRPTYPE	(TCP)	+
	CONNAME	('127.0.0.1 (1416)')	+
	CLUSTER	(SHINE)	+
	REPLACE		
DEFINE	CHANNEL	(C_QM1)	+
	CHLTYPE	(CLUSSDR)	+
	TRPTYPE	(TCP)	+
	CONNAME	('127.0.0.1 (1414)')	+
	CLUSTER	(SHINE)	+
	REPLACE		

9.3.6 例 6

例 6 可以看作是例 3 和例 5 的组合，图 9-6。QM1、QM2、QM3 组成群集，这里哪一个为配置库并不重要，假定 QM1 是配置库队列管理器。群集中 QM2 和 QM3 是消息处理单元，上面有同名群集队列 QL。QM1 扮演群集网关（Gateway）的角色，连接 QM4。要求 QM4 上发出的消息会自动平衡地路由去 QM2 和 QM3 上的本地队列 QL，而从 QM2 或 QM3 上的应答消息可以到达 QM4。

另外，为了满足消息的自动应答功能，比如报告消息（Report）需要在 QM4 上创建 QM1、QM2 和 QM3 的队列管理器别名，以便使 QM4 上的回复消息知道进入群集的路由。同样，需要在 QM1 上创建 QM4 的队列管理器别名，并将其在群集中共享，以便使群集中的回复消息知道出去的路由。

可以用 MQPutCOA 例程测试：

```
MQPutCOA QR_QM4 QL_QM4 QM4
MQPutCOA QR_QM4 QM1 QL_QM2
MQPutCOA QR_QM4 QM2 QL_QM2
MQPutCOA QR_QM4 QM3 QL_QM3
MQPutCOA QR_QM4 QM3 QR_QM4 // 自动应答消息也会轮流路由去两个 QL
```

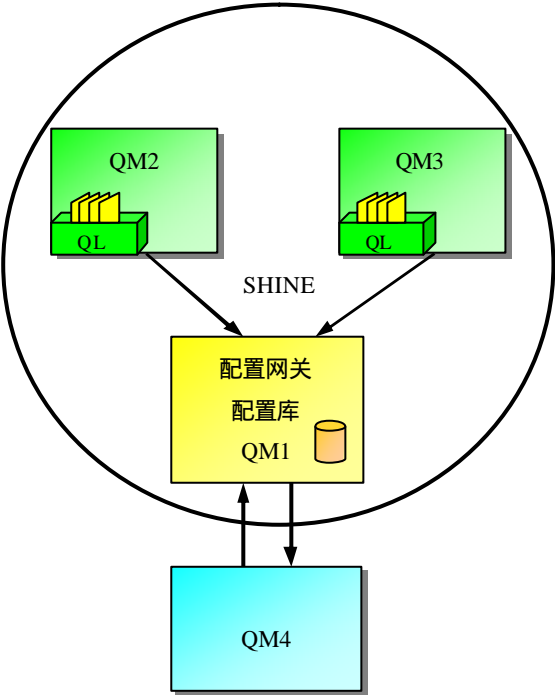


图 9-6 在带网关的群集中实现负载平衡

```
QM1:
----
ALTER QMGR REPOS (SHINE)
DEFINE CHANNEL (C_QM1) +
        CHLTYPE (CLUSRCVR) +
        TRPTYPE (TCP) +
        CONNAME ('127.0.0.1 (1414)') +
        CLUSTER (SHINE) +
        REPLACE
```

DEFINE	QREMOTE	(QM4)	+
	RNAME	('')	+
	RQMNAME	(QM4)	+
	XMITQ	(QX_QM4)	+
	CLUSTER	(SHINE)	+
	REPLACE		
DEFINE	QREMOTE	(SHINE.CLUSTER)	+
	RNAME	('')	+
	RQMNAME	('')	+
	REPLACE		
DEFINE	QREMOTE	(QR_QM4)	+
	RNAME	(QL_QM4)	+
	RQMNAME	(QM4)	+
	XMITQ	(QX_QM4)	+
	CLUSTER	(SHINE)	+
	REPLACE		
DEFINE	CHANNEL	(C_QM1.QM4)	+
	CHLTYPE	(SDR)	+
	TRPTYPE	(TCP)	+
	CONNAME	('127.0.0.1 (1417)')	+
	XMITQ	(QX_QM4)	+
	REPLACE		
DEFINE	CHANNEL	(C_QM4.QM1)	+
	CHLTYPE	(RCVR)	+
	TRPTYPE	(TCP)	+
	REPLACE		
DEFINE	QLOCAL	(QX_QM4)	+
	USAGE	(XMITQ)	+
	REPLACE		
QM2:			

DEFINE	QLOCAL	(QL)	+
	CLUSTER	(SHINE)	+
	REPLACE		
DEFINE	CHANNEL	(C_QM2)	+
	CHLTYPE	(CLUSRCVR)	+
	TRPTYPE	(TCP)	+
	CONNAME	('127.0.0.1 (1415)')	+
	CLUSTER	(SHINE)	+
	REPLACE		
DEFINE	CHANNEL	(C_QM1)	+
	CHLTYPE	(CLUSSDR)	+
	TRPTYPE	(TCP)	+

	CONNAME	('127.0.0.1 (1414)')	+
	CLUSTER	(SHINE)	+
	REPLACE		
QM3:			

DEFINE	QLOCAL	(QL)	+
	CLUSTER	(SHINE)	+
	REPLACE		
DEFINE	CHANNEL	(C_QM3)	+
	CHLTYPE	(CLUSRCVR)	+
	TRPTYPE	(TCP)	+
	CONNAME	('127.0.0.1 (1416)')	+
	CLUSTER	(SHINE)	+
	REPLACE		
DEFINE	CHANNEL	(C_QM1)	+
	CHLTYPE	(CLUSSDR)	+
	TRPTYPE	(TCP)	+
	CONNAME	('127.0.0.1 (1414)')	+
	CLUSTER	(SHINE)	+
	REPLACE		
QM4:			

DEFINE	QREMOTE	(QM1)	+
	RNAME	('')	+
	RQMNAME	(QM1)	+
	XMITQ	(QX_QM1)	+
	REPLACE		
DEFINE	QREMOTE	(QM2)	+
	RNAME	('')	+
	RQMNAME	(QM2)	+
	XMITQ	(QX_QM1)	+
	REPLACE		
DEFINE	QREMOTE	(QM3)	+
	RNAME	('')	+
	RQMNAME	(QM3)	+
	XMITQ	(QX_QM1)	+
	REPLACE		
DEFINE	QREMOTE	(QR)	+
	RNAME	(QL)	+
	RQMNAME	(SHINE.CLUSTER)	+
	XMITQ	(QX_QM1)	+
	REPLACE		

DEFINE	QLOCAL	(QL_QM4)	+
	REPLACE		
DEFINE	CHANNEL	(C_QM1.QM4)	+
	CHLTYPE	(RCVR)	+
	TRPTYPE	(TCP)	+
	REPLACE		
DEFINE	CHANNEL	(C_QM4.QM1)	+
	CHLTYPE	(SDR)	+
	TRPTYPE	(TCP)	+
	CONNNAME	('127.0.0.1 (1414)')	+
	XMITQ	(QX_QM1)	+
	REPLACE		
DEFINE	QLOCAL	(QX_QM1)	+
	USAGE	(XMITQ)	+
	REPLACE		

9.4 多群集队列实例与共享队列组

对于 AIX、HPUX、Solaris、Linux、Windows、OpenVMS、OS/2、OS/400 和 z/OS 多个队列管理器可以组成一个群集，其间用群集发送通道和群集接收通道连接起来。一个群集至少有一个配置库 (Repository)，存放着整个群集的共享信息。群集中凡是登记在配置库上的对象在整个群集上可见。通常群集上可以有多个同名本地队列，用来将负载分担到群集中不同的队列管理器上去。其实，每个队列管理器各自维护着一个本地队列，整个群集中存在多个本地队列的同名实例，这就是多群集队列实例，通常用来负载均衡。

对于 z/OS，除了多群集队列实例外还有另一种类似的共享机制：共享队列组 (Queue-Sharing Group, QSG)。多个队列管理器可以组成一个共享队列组，队列管理器之间不需要用通道连接，本地队列可以在系统耦合器 (Couple Facility, CF) 上创建并共享出来，称为共享队列 (Shared Queue)，在整个共享队列组中可见。应用程序可以连接上 QSG 中任何一个队列管理器，便可操作 QSG 中共享的本地队列。类似于群集，共享队列组也有共享配置库 (Shared Repository)，记录共享配置信息。本质上讲，群集 (Cluster) 中的多个共享队列是多个实例，而共享队列组 (QSG) 中的共享队列是只有一个实例 (图 9-7)。

共享队列组中不同队列管理器之间传递消息叫做 Intra-Group Queuing (IGQ)。

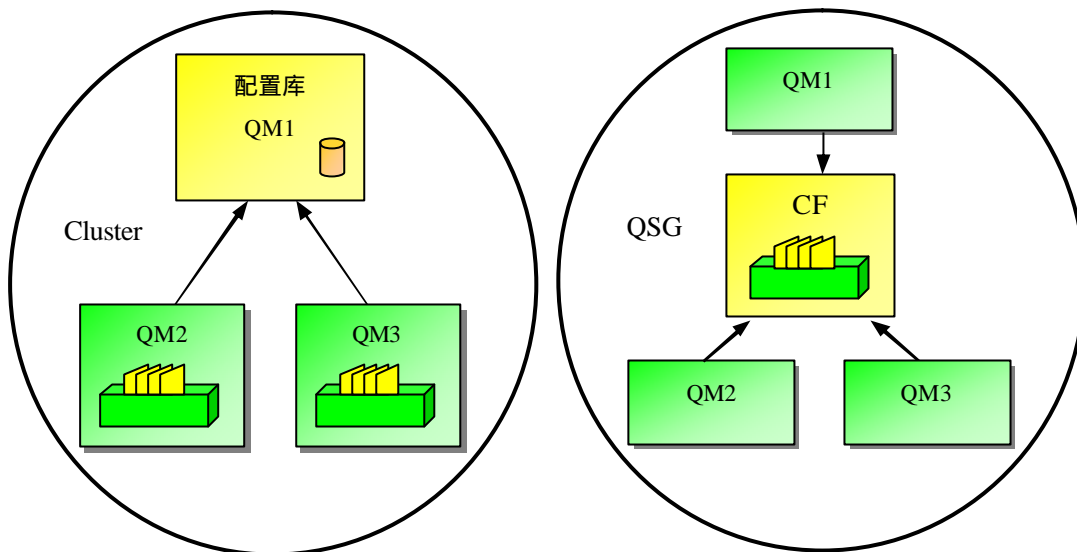


图 9-7 群集(Cluster)与共享队列(QSG)

队列管理器与 QSG 相关的属性：

- QSGNAME QSG 的名字, 即队列管理器所属的那个 QSG 的名字
- IGQ 队列管理器是否支持 IGQ。(Enable 或 Disable)
- IGQAUT IGQ 身分检查方式
- IGQUSER IGQ 身分检查的用户名

本地队列与 QSG 相关的属性：

- CFSTRUCT 在 QSG 中, 消息是放在 Coupling Facility (CF) 中的, 这里指的是 CF 的名字
- QSGDISP 队列共享方式: COPY | GROUP | PRIVATE | SHARED | QMGR
 - QMGR 队列只在队列管理器中有效, 不同的队列管理器可以有同名的队列, 但它们互不相干
 - COPY 队列是共享配置库中的主定义的一份本地拷贝。在一开始, 所有队列管理器的拷贝中的内容应该是完全一样的, 但是队列管理器可以自由修改这份拷贝定义, 从而变得各不相同。但是当共享配置库中的主定义发生修改时, 所有的拷贝定义都会被同步。
 - SHARED 队列在共享配置库中定义, 为 QSG 中所有的队列管理器所知, 在 QSG 只有一个实例。队列的消息存放在 Coupling Facility (CF) 里。
 - GROUP 队列在共享配置库中定义

由于 QSG 中, 用户可以连接到任何一个队列管理器上对 QSG 中的任何一个其它的队列管理器上共享出来的队列进行操作和修改属性, 所以 z/OS 上 MQSC 命令大多数都有 CMDSCOPE 选项。

- .. 只对 MQSC 连接的队列管理器有效
- QMgrName 通过 MQSC 连接的队列管理器, 只对指定的队列管理器有效
- * 对 QSG 上所有的队列管理器有效

在 z/OS 上可以用 display group 或 display dqm 来显示 QSG 的设置。

群集 (Cluster) 与 共享队列组 (QSG) 分别用于不同的场合，群集中消息是分布式的，共享队列组中消息是集中式的。两者可以共存，也就是说，一个队列管理器可以是群集中的成员，同时又是共享队列组中的成员。

9.5 群集负载用户出口 (Cluster Workload User Exit)

在群集多共享实例的环境中，消息会路由去多个实例。缺省情况下，系统会采用轮循算法，即多个实例的负载是相同的。也可以在路由点的队列管理器上通过群集负载用户出口 (Cluster Workload User Exit)，自行设计负载均衡算法。

详见“用户出口”相关章节。

第 10 章 监控与性能

WebSphere MQ 为应用提供后台的消息服务。由于身处后台，往往很难对系统正在发生的事情实时掌握。随着系统复杂度的提高，对系统监控与性能要求就变得非常重要。WebSphere MQ 提供了一种强有力的监控功能——事件消息，系统在满足某些事先设定的条件后，自动产生事件消息，通过捕捉事件消息可以实时地了解系统中发生的问题。事件就像是遍布系统的监视器，一旦发生异常情况就会发送事件消息，自动报警。

事件消息中最重要的是性能事件，它往往意味着系统出现处理瓶颈或异常，系统的内压超时警戒值，这时需要及时干预以防系统过载。事实上，我们在应用设计的时候就可以把性能问题考虑进去。本章对多种消息会话模式、消息设计和编程语言进行了性能上的比较并给出了相应的调优建议。

10.1 事件 (Event)

10.1.1 概念

WebSphere MQ 对某些异常情况做了事先的定义，称为事件 (Event)。一旦这种系统对于异常情况的定义条件满足了，也就是事件发生了。WebSphere MQ 会在事件发生时自动产生一条对应的事件消息 Event Message，放入相应的系统事件队列中。如果我们能够实时监控这些事件队列，就可以动态地得知系统发生了什么，从而做出相应的反应。

WebSphere MQ 的事件分四种，分别对应四个不同的系统队列。如表 8-1

表 8-1 WebSphere MQ 的事件及其队列

事件	事件队列	举例
Queue Manager Events	SYSTEM.ADMIN.QMGR.EVENT	消息的目标队列不存在
Channel Events	SYSTEM.ADMIN.CHANNEL.EVENT	通道停止
Performance Events	SYSTEM.ADMIN.PERFM.EVENT	队列满
Configuration Events	SYSTEM.ADMIN.CONFIG.EVENT	对象有创建，修改，删除 (仅 z/OS 平台)

事实上，事件队列可以被删除重建。可以是本地队列，别名队列甚至是远程队列。如果把多个队列管理器的事件队列都定义成远程队列，并且指定到同一个目标队列上，则可以实现集中式监控管理。如果远程的队列管理器无法投递，事件消息会路由去死信队列。事件队列 (Event Queue) 不可以是传输队列，因为事件消息没有传输头。如果事件队列是本地队列，则事件消息在无法投递时被自动丢弃。如果在事件队列上设置了触发器 (Trigger)，则可以自动触发程序处理这些事件消息，从而做出及时正确的反应。由此可见，对事件消息的灵活运用可以方便地对系统运行情况进行全面掌握。

10.1.2 队列管理器事件 (Queue Manager Event)

Queue Manager Event 大都由队列管理器自身的运行引起，表示队列管理器的运行状态。产生原因可能是：

- MQI 调用。无论是外部或内部的 MQI 调用都可以产生该 Event。Event Message 的 ReasonCode 与 MQI 返回的 ReasonCode 相同。
- 命令的调用。比如启停 Queue Manager，而用户无权调用该命令。

我们可以把队列管理器事件根据原因进一步分类，分成权限 (Authority) 事件、禁止 (Inhibit) 事件、本地 (Local) 事件、远程 (Remote) 事件、启停 (Start & Stop) 事件等等，下面我们就逐一进行介绍。

10.1.2.1 权限事件(Authority)

权限事件产生的原因是因为 MQI 调用或命令执行没有访问权限，共分四种类型。对于 NSK 平台，只支持类型 1，Windows，UNIX，OS/400，OpenVMS 支持所有四个类型。

- Not Authorized (类型 1) MQCONN 时用户无权连接 Queue Manager
- Not Authorized (类型 2) MQOPEN or MQPUT1 时用户无权打开 Queue
- Not Authorized (类型 3) MQCLOSE 时用户无权删除 Permanent Dynamic Queue
- Not Authorized (类型 4) 用户无权访问命令中涉及的对象

10.1.2.2 禁止事件(Inhibit)

禁止事件产生的原因是因为 MQGET 或 MQPUT 时，队列禁止该操作。

- Get Inhibited MQGET 时队列禁止该操作
- Put Inhibited MQPUT 时队列禁止该操作

10.1.2.3 本地事件(Local)

本地事件产生的原因是因为应用程序或队列管理器无法访问本地对象，比如本地对象没有定义。

- Alias Base Queue Type Error
MQOPEN or MQPUT1 打开一个别名队列，但是它定义中的 BaseQName 却不是
一个本地队列或远程队列

- Unknown Alias Base Queue
MQOPEN or MQPUT1 打开一个别名队列，但是它定义中的 BaseQName 却不是
一个队列名
- Unknown Object Name
MQOPEN or MQPUT1 时 MQOD.ObjectQMgrName 指向本地队列管理器，但是
MQOD.ObjectName 却不是一个本地对象

10.1.2.4 远程事件(Remote)

远程事件产生的原因是因为应用程序或队列管理器无法访问远程对象,比如传输队列没有正确定义。

- Default Transmission Queue Type Error
MQOPEN 或 MQPUT1 打开的远程队列没有指定 XmitQName ,所以 MQ 试图寻找合适的传输队列，在没有找到与远程队列管理器同名的传输队列的情况下，MQ 试图寻找缺省传输队列，可是队列管理器的 DefXmitQName 指定的对象却不是一个本地队列。
- Default Transmission Queue Usage Error
MQOPEN 或 MQPUT1 打开的远程队列没有指定 XmitQName ,所以 MQ 试图寻找合适的传输队列，在没有找到与远程队列管理器同名的传输队列的情况下，MQ 试图寻找缺省传输队列，可是队列管理器的 DefXmitQName 指定的对象却没有指定 Usage = MQUS_TRANSMISSION
- Queue Type Error
MQOPEN 时 MQOD.ObjectQMgrName 指定的是远程队列名 (QueueManager Alias)，这个远程队列的 RemoteQMgrName 就是本地队列管理器名。可是 MQOD.ObjectName 指向本地队列管理器上的模型队列。
- Remote Queue Name Error
MQOPEN 或 MQPUT1 时遇到两种情况中任何一种：
 1. 远程队列定义的 RemoteQName 为空
 2. MQOD.ObjectQMgrName 非空且不为本本地队列管理器，可是 ObjectName 为空。
这两种情况都注定找不到远程队列。
- Transmission Queue Type Error
MQOPEN 或 MQPUT1 时 MQOD.ObjectName 和 MQOD.ObjectQMgrName 指定了远程队列，可是这个远程队列定义中遇到了以下两种情况中的任何一种：
 1. 远程队列定义中的 XmitQName 为空
 2. 远程队列定义中的 XmitQName 不为空，但是 RemoteQMgrName 却不是一个本地队列。
- Transmission Queue Usage Error
MQOPEN 或 MQPUT1 时遇到以下情况的任何一种：
 1. MQOD.ObjectQMgrName 指定的是本地队列，可是这个本地队列没有指定 Usage = MQUS_TRANSMISSION。
 2. MQOD.ObjectName 和 MQOD.ObjectQMgrName 指定的是远程队列，可是这个远程队列的 XmitQName 不为空且指向的队列没有指定 Usage = MQUS_TRANSMISSION，或者这个远程队列的 XmitQName 为空且 RemoteQMgrName 指向的队列没有指定 Usage = MQUS_TRANSMISSION。

3. 通过 DCE Cell 方式确定的本地队列没有指定 Usage = MQUS_TRANSMISSION。
- Unknown Default Transmission Queue
MQOPEN 或 MQPUT1 打开的远程队列没有指定 XmitQName ,所以 MQ 试图寻找合适的传输队列, 在没有找到与远程队列管理器同名的传输队列的情况下, MQ 试图寻找缺省传输队列, 可是 Queue Manager 的 DefXmitQName 却不是本地定义的队列。
 - Unknown Remote Queue Manager
MQOPEN 或 MQPUT1 时遇到以下情况的任何一种：
 1. ObjectQMgrName 为空或本地队列管理器名, ObjectName 为远程队列, 且远程队列的 XmitQName 为空。但是, 找不到与 RemoteQMgrName 同名的传输队列, 且队列管理器的缺省传输队列 DefXmitQName 为空。
 2. ObjectQMgrName 是队列管理器别名(与远程队列同名), 远程队列的 XmitQName 为空。但是, 找不到与 RemoteQMgrName 同名的传输队列, 且队列管理器的缺省传输队列 DefXmitQName 为空。
 3. ObjectQMgrName 不为空, 不为本地队列管理器, 不为本地队列, 不为队列管理器别名, 且队列管理器的缺省传输队列 DefXmitQName 为空。
 4. ObjectQMgrName 为空或本地队列管理器名, ObjectName 为远程队列, RemoteQMgrName 为空或本地队列管理器名。
 5. ObjectQMgrName 为远程队列名, 这时应该对应队列管理器别名, 可是这个远程队列的 RemoteQName 却不为空。
 6. ObjectQMgrName 为模型队列名。
 7. 通过 DCE Cell 方式确定了队列的名字, 可是在 Cell 中找不到同名的远程队列管理器, 且队列管理器的缺省传输队列 DefXmitQName 为空。
 - Unknown Transmission Queue
MQOPEN 或 MQPUT1 时 MQOD.ObjectName 和 MQOD.ObjectQMgrName 指定了远程队列, 可是这个远程队列的 XmitQName 不为空且不是本地定义的队列。

10.1.2.5 启停事件(Start & Stop)

启停事件产生的原因是因为启停队列管理器操作。Stop 事件通常不被记录, 除非 SYSTEM.ADMIN.QMGR.EVENT 的缺省消息属性为 Persistent, z/OS 仅支持 start。

- Queue Manager Active Start
- Queue Manager Not Active Stop or Quiesce

10.1.2.6 启用和禁用

队列管理器事件都与队列管理器相关, 启用和禁用的开关都为队列管理器属性。在 MQSC 中用 alter qmgr <attribute> (enable) 或 alter qmgr <attribute> (disable) 来控制。如表 8-2。

表 8-2 与队列管理器事件相关的属性

Event 类型	队列管理器属性
Authority	AUTHOREV
Inhibit	INHIBTEV

Local	LOCALEV
Remote	REMOTEEV
Start & Stop	STRSTPEV

10.1.3 通道事件 (Channel Event)

通道事件通常由通道的运行和配置引起，表示通道的运行状态。产生原因可能为：

- 执行启停通道命令
- 通道实例的启停
- 通道接收消息时格式转换出错 (Conversion Error)
- 自动创建通道 (无论是否成功)

注意 执行命令启停通道和通道实例实现启停是两个事件，所以如果通过命令启停通道，你会得到两个事件消息。但如果通过 listener, runmqchl, 或者 trigger 来启停通道，你只会得到后一个事件消息。另外，通道成功启动，通道两端的队列管理器都会产生事件消息，放入 SYSTEM.ADMIN.CHANNEL.EVEN，如果出错，则会引发相应的其它事件。

通道事件可以根据原因进一步分成 Channel、IMS Bridge 和 SSL 三类，我们在下面逐一进行介绍。

10.1.3.1 Channel

这类事件产生的原因是因为通道的启停和创建。

- Channel Activated Channel 曾经发出过 Not Activated Event，现在变成 Active 状态。一般来说，是因为有其它 Channel 释放了名额。(Active Slot)
- Channel Auto-definition Error Channel 实现 Auto-Definition 时出错。这可能是自动定义时有什么问题，也可能是 Auto-Definition Exit 禁止这样做。具体原因可以参考 Event 消息
- Channel Auto-definition OK Channel 实现 Auto-Definition 成功
- Channel Conversion Error 在应用程序带 MQGMO_CONVERT 参数 MQGET 时，因为数据转换出错。出错原因可以参考 Event Data 中的 ConversionReasonCode
- Channel Not Activated Channel 在启动或重建连接的时候，因为名额 (Active Slot) 有限，无法变成 Active 状态。这个限制对于 OS/2, AIX, HP/UX, Solaris, 是 qm.ini 中的 MaxActiveChannels。对于 Windows NT, 为 Registry 中的 MaxActiveChannels。对于 z/OS, 是 CSQXPARM 中的 ACTCHL.
- Channel Started Channel 成功启动 (或者执行了 Start Channel 命令，或者建立了 Channel 实例)，当 Initial Data Negotiation 结束，必要的同步消息交换过之后。
- Channel Stopped Channel 曾经发出过 Start Event，现在 Channel 实例停止了

- Channel Stopped By User Channel 因为执行了用户的 STOP CHL 命令，具体原因可以参考 Event Data 中的 ReasonQualifier

10.1.3.2 IMS Bridge

这类事件产生的原因是因为 IMS Bridge 的启停，仅支持 z/OS 平台。

- Start IMS Bridge 启动
- Stop IMS Bridge 停止

10.1.3.3 SSL

这类事件产生的原因是因为 Secure Sockets Layer (SSL) 使用出错。

- SSL Error 创建 SSL 连接时出错

10.1.3.4 启用和禁用

大多数通道事件都是自动控制的，无法通过命令启用或禁用。只有 Channel Auto-Definition，可以通过队列管理器的属性 ChannelAutoDefEvent (CHADEV) 来控制。表 8-3。

表 8-3 与通道事件相关的属性

Event 类型	队列管理器属性
Channel Auto-Definition	CHADEV

如果要禁止所有的通道事件消息，可以删除 SYSTEM.ADMIN.CHANNEL.EVENT 队列或者将其设置为 Put-Inhibited，这样，通道事件就无法产生了。

10.1.4 性能事件 (Performance Event)

性能事件是与应用程序性能相关的事件，通常在队列管理器和队列上设置开关。四个 Event Queue 自身不会产生性能事件，事务中的 MQGET 和 MQPUT 不管是否提交都会引发性能事件。

性能事件根据其产生的原因可以进一步细分成队列深度 (Queue Depth) 事件和队列服务间隔 (Queue Service Interval) 事件两类，下面我们将逐一进行介绍。

10.1.4.1 队列深度事件(Queue Depth)

队列深度事件表示队列中的消息达到了一定的数量，通常是超过或跌破某一警戒值。比如队列满，队列空。

- Queue Depth High
MQPUT 或 MQPUT1 时发现队列中的消息数量达到或超过队列的 QDepthHighLimit (QDEPTHHI) 属性。队列中原先的消息数量至少为 $MAXDEPTH * QDEPTHHI \% - 1$ ，新的消息放入后达到或超过 $MAXDEPTH * QDEPTHHI \%$ ，

只有在消息数量上升并突破界线时才会引发 High Event。

- Queue Depth Low

MQGET 时发现队列中的消息数量达到或低于队列的 QDepthLowLimit (QDEPTHLO) 属性。队列中原先的消息数量至多为 $MAXDEPTH * QDEPTHLO \% + 1$ ，消息取走后数量达到或低于 $MAXDEPTH * QDEPTHLO \%$ ，只有是消息数量下降并突破界线时才会引发 Low Event。

- Queue Depth Full

MQPUT 或 MQPUT1 时发现队列中的消息数量达到队列的 MaxQDepth (MAXDEPTH) 属性。队列中原先的消息数量已经达到 MAXDEPTH，新的消息无法放入。

WebSphere MQ 队列的 Full，High，Low Event 开关之间也会相互自动转换设置 (Enable/Disable)，队列到达一个状态后会自动关闭自己的状态开关并打开下一个或多个状态开关，以便跟踪监测。图 8-1。

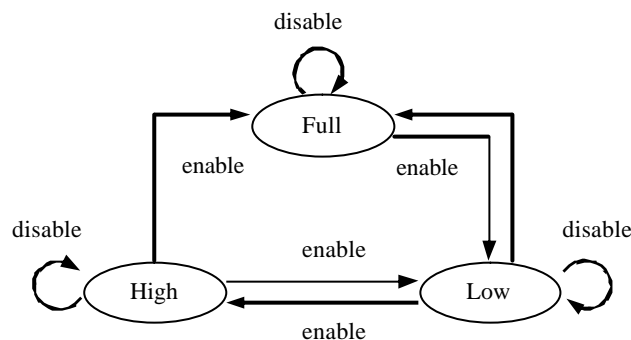


图 8-1 Queue Depth 开关之间的自动转换

10.1.4.2 队列服务间隔事件(Queue Service Interval)

队列服务间隔事件表示在用户规定的时限中服务处理消息的情况，具体如下：

- Queue Service Interval High

在队列 QServiceInterval (QSVICINT) 属性规定的时限内未成功调用 MQPUT 或 MQGET 操作。即上一次 MQPUT 或者 MQGET (队列未取空) 以来的规定时限内未成功调用 MQGET 或 MQPUT

- Queue Service Interval OK

在队列 QServiceInterval (QSVICINT) 属性规定的时限内成功调用了 MQGET 操作。即上一次 MQPUT 或者 MQGET (队列未取空) 以来的规定时限内成功调用了 MQGET

队列服务间隔事件是为了考察对消息处理的服务 (Service) 是否还活着。一般来说，服务活着就意味着不断取消息，所以每隔一定的服务时间应该会有消息被 MQGET 取走。所以考察 MQGET 与上一个 MQGET/MQPUT 之间的时间间距可以在一定程度上反应服务程序的运行健康状况。计时工作是由后台的队列服务计时器 (Queue Service Timer) 完成的，当然，每次超时后计时器 (Timer) 应该归零重置。

队列的 QSVCI EV 缺省值为 NONE，应该将其设置为 HIGH 或 OK。队列 QSVCI NT 缺省值为 999,999,999 以毫秒为单位。

队列服务计时器不是一个时刻监测的超时机制，它只有在下一次 MQGET 或 MQPUT 时才计算出于上一次的时间差，从而发出 High Event 或 OK Event。类似于消息的超时机制 (MQMD.Exipry)。

具体算法：

1. 如果队列取空，队列服务计时器状态为 OFF
2. 如果 MQPUT 一个空的队列，使之有第一条消息，或者 MQGET 一个队列未取空。则队列服务计时器归零重置 (Reset)，状态设置为 ON，记下这个时间点 T1。
3. 下一次 MQPUT 或 MQGET 记下时间点 T2。
4. 如果 $T2 - T1 > Interval$ ，则产生 High Event。如果 $T2 - T1 \leq Interval$ ，且 T2 由 MQGET 记下的，则产生 OK Event。
5. 一旦产生 Event，则在自动切换另一个 Event Enable 状态
6. 回到 1

High Event 和 OK Event 的 Enable/Disable 设置是排它的，但又会自动相互切换。也就是说，如果设置了其中一个 Enable，另一个自动设置成 Disable。一旦产生 High Event，WebSphere MQ 会自动设置 High Event Disable 且 OK Event Enable，反之，一旦产生 OK Event，WebSphere MQ 会自动设置 OK Event Disable 且 High Event Enable。图 8-2。

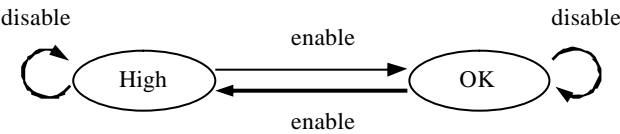


图 8-2 High Event 和 OK Event 之间的互相转换

10.1.4.3 启用和禁用

性能事件通常会与队列管理器和队列的设置都有关系。其实，队列管理器上只有一个属性 PERFMEV，在启用时要首先将其设置为 enable。队列上的相关的阈值可以设置，系统会根据当前的数据与阈值比较，从而产生 Event 消息。

表 8-4 与性能事件相关的属性

Event 类型	队列管理器属性	队列属性	队列阈值
Queue Depth High	PERFMEV	QDPHIEV	QDEPTHHI
Queue Depth Low	PERFMEV	QDPLOEV	QDEPTHLO
Queue Depth Full	PERFMEV	QDPMAXEV	MAXDEPTH
Queue Service Interval High	PERFMEV	QSVCI EV (HIGH)	QSVCI NT
Queue Service Interval OK	PERFMEV	QSVCI EV (OK)	QSVCI NT

对于 Queue Depth，首先打开队列管理器的性能事件开关（设置 PERFMEV 为 enable）。接着，打开队列的性能事件开关（设置相应属性 QDPHIEV, QDPLOEV, QDPMAXEV 为 enable）。最后，设定相关的阈值 (QDEPTHHI, QDEPTHLO, MAXDEPTH)。

对于 Queue Service Interval , 首先打开队列管理器的性能事件开关 (设置 PERFMEV 为 enable)。接着 , 设定队列 QSVCIEV 属性为 HIGH 或 OK。最后 , 设置队列的 QServiceInterval 属性 (QSVCINT)。

10.1.5 配置事件 (Configuration Event)

配置事件是因为队列管理器的对象配置发生了改变而引起的。仅支持 z/OS 平台。

- Create Object 创建对象
- Change Object 修改对象
- Delete Object 删除对象
- Refresh Object 刷新对象

10.1.5.1 启用和禁用

配置事件比较简单 , 只需要设置队列管理器属性 CONFIGEV 即可。

表 8-5 与配置事件相关的属性

Event 类型	队列管理器属性
Configuration Event	CONFIGEV

对于以下任何一种情况下都可以产生配置事件 :

- DELETE 或 REFRESH 命令
 - DELETE AUTHINFO
 - DELETE CFSTRUCT
 - DELETE CHANNEL
 - DELETE NAMELIST
 - DELETE PROCESS
 - DELETE QMODEL/QALIAS/QREMOTE
 - DELETE STGCLASS
 - REFRESH QMGR
- DEFINE 或 ALTER 命令 , 不管配置是否真的改动
 - DEFINE/ALTER AUTHINFO
 - DEFINE/ALTER CFSTRUCT
 - DEFINE/ALTER CHANNEL
 - DEFINE/ALTER NAMELIST
 - DEFINE/ALTER PROCESS
 - DEFINE/ALTER QMODEL/QALIAS/QREMOTE
 - DEFINE/ALTER STGCLASS
 - DEFINE MAXSMSGS
 - ALTER QMGR 除非 CONFIGEV = DISABLED
- 对于非 Temporary Dynamic 的 Local Queue , 不管配置是否真的改动
 - DELETE QLOCAL
 - DEFINE/ALTER QLOCAL
- 对于非 Temporary Dynamic 的 Local Queue , 不管配置是否真的改动

■ MQSET API (成功调用)

对于以下情况不会引发 Configuration Event

- MQSET 调用出错
- 放入 Configuration Event Queue 时出错，则 Configuration Event 消息直接被仍掉
- 对 Temporary Dynamic Queue 的配置改动
- 对 SYSTEM.ADMIN.CONFIG.EVENT 自身的配置改动
- 内部改动队列的 TRIGGER 属性

10.1.6 事件消息

Event 消息的格式与 PCF 消息格式相同，分成 MQMD、MQCFH 和 Data 三部分。可以用 PCF 消息的结构来构造和解析消息内容。在 Data 中含有一些性能的统计信息：

- TimeSinceReset 上一次 Reset 以来的时间
- HighQDepth 上一次 Reset 以来队列达到的最大深度
- MsgEnqCount 上一次 Reset 以来 MQPUT/MQPUT1 的次数
- MsgDeqCount 上一次 Reset 以来 MQGET 的次数

这些统计信息可以被多种方法归零 (Reset)：

- Performance Event 产生
- Queue Manager 重启
- z/OS 中用命令 RESET QSTATS
- 应用程序发出 PCF 命令 Reset Queue Statistics

10.1.7 事件监控

在生产环境中，有时为了系统安全能及早地发现问题，或者对系统的状态和性能问题提前报警，设计者会打开 Event 开关，并对 Event 队列进行监控。

在通常情况下，对 Event 队列监控需要编写程序。我们写出多线程的程序，每一个线程监控一个队列。在开放平台环境下，需要监控三个队列：SYSTEM.ADMIN.QMGR.EVENT, SYSTEM.ADMIN.CHANNEL.EVENT, SYSTEM.ADMIN.PERFM.EVENT（见例程）。对于 z/OS 平台，需要多监控一个队列 SYSTEM.ADMIN.CONFIG.EVENT。

在 Windows 中，可以用 NT/2000/XP 的性能工具通过添加计数器来监控 MQ 性能指标。在“控制面板”里选“管理工具”，再选“性能”。添加计数器，在“对象”中选择“MQSeries 队列”。可添加的计数器有四种：

- % 队列深度
- 队列深度
- 每秒入队列的消息数量
- 每秒出队列的消息数量

下面让我们来做两个实验，加深理解。

10.1.9 实验一：Queue Depth

环境设置：

创建并启动队列管理器 QM，配置如下：

```
alt qmgr AUTHOREV (enabled) INHIBTEV (enabled) LOCALEV (enabled) REMOTEEV
(enabled) STRSTPEV (enabled) CHADEV (enabled) CHAD (enabled) PERFMEV
(enabled) DEADQ (SYSTEM.DEAD.LETTER.QUEUE)
def ql (Q) QDPHIEV (enabled) QDEPTHHI (80) QDPLOEV (enabled) QDEPTHLO (20)
QDPMAXEV (enabled) MAXDEPTH (10) QSVCIEV (NONE) replace
alt ql (SYSTEM.ADMIN.QMGR.EVENT) DEFPSIST (yes)
```

实验步骤：

1. 打开一个命令行窗口，运行事件监控例程。会得到 Queue Manager Event (MQRC_Q_MGR_ACTIVE)，这就是队列管理器启动时的 Start & Stop Event。
2. 打开另一个命令行窗口，用 amqspout 向队列中逐条放入消息，这时队列 Q 中消息递增，监控窗口没有反应。当放入第 8 条消息后，监控窗口得到 Performance Event (MQRC_Q_DEPTH_HIGH)。消息中有如下信息：

[MQCA_Q_MGR_NAME]	[QM]
[MQCA_BASE_Q_NAME]	[Q]
[MQIA_TIME_SINCE_RESET]	[0]
[MQIA_HIGH_Q_DEPTH]	[8]
[MQIA_MSG_ENQ_COUNT]	[8]
[MQIA_MSG_DEQ_COUNT]	[0]

这实际上就是队列中消息数量增长达到预先设定的阈值 8 (10x80%)。用 display qlocal (Q) 命令查看队列属性，发现 QDPMAXEV(ENABLED), QDPHIEV(DISABLED), QDPLOEV(ENABLED)，这说明 WebSphere MQ 会自动改变开关设置。

3. 继续往队列 Q 中放消息，当放入第 3 条 (总第 11 条) 时，队列已满，程序出错。监控窗口得到 Performance Event (MQRC_Q_FULL)。消息中有如下信息：

[MQCA_Q_MGR_NAME]	[QM]
[MQCA_BASE_Q_NAME]	[Q]
[MQIA_TIME_SINCE_RESET]	[16]
[MQIA_HIGH_Q_DEPTH]	[10]
[MQIA_MSG_ENQ_COUNT]	[2]
[MQIA_MSG_DEQ_COUNT]	[0]

这实际上就是队列中消息数量增长企图超过预先设定的队列深度。用 display qlocal (Q) 命令查看队列属性，发现 QDPMAXEV(DISABLED), QDPHIEV(DISABLED), QDPLOEV(ENABLED)，这说明 WebSphere MQ 会自动改变开关设置。

4. 用我们前面介绍的例程 MQGet 逐条将消息取出，这时队列 Q 中消息递减，监控窗口没有反应。当取出第 8 条消息后，监控窗口得到 Performance Event (MQRC_Q_DEPTH_LOW)。消息中有如下信息：

[MQCA_Q_MGR_NAME]	[QM]
[MQCA_BASE_Q_NAME]	[Q]
[MQIA_TIME_SINCE_RESET]	[98]
[MQIA_HIGH_Q_DEPTH]	[10]
[MQIA_MSG_ENQ_COUNT]	[0]

这实际上就是队列中消息数量减少达到预先设定的阈值 2 (10x20%)。用 `display qlocal (Q)` 命令查看队列属性,发现 QDPMAXEV(ENABLED), QDPHIEV(ENABLED), QDPLOEV(DISABLED), 这说明 WebSphere MQ 会自动改变开关设置。

10.1.10 实验二：Queue Service Interval

环境设置：

创建并启动队列管理器 QM，配置如下：

```
alt qmgr AUTHOREV (enabled) INHIBTEV (enabled) LOCALEV (enabled) REMOTEEV
(enabled) STRSTPEV (enabled) CHADEV (enabled) CHAD (enabled) PERFMEV
(enabled) DEADQ (SYSTEM.DEAD.LETTER.QUEUE)
def ql (Q) QDPHIEV (disabled) QDEPTHHI (80) QDPLOEV (disabled) QDEPTHLO (20)
QDPMAXEV (disabled) MAXDEPTH (10) QSVCIEV (HIGH) QSVCINT (5000) replace
alt ql (SYSTEM.ADMIN.QMGR.EVENT) DEFPSIST (yes)
```

实验步骤：

1. 打开一个窗口，运行事件监控例程 MQEventMonitor。会得到 Queue Manager Event (MQRC_Q_MGR_ACTIVE)，这就是队列管理器启动时的 Start & Stop Event。检查这时队列 Q 的属性为 QSVCIEV(HIGH)。
2. 打开另一个窗口，用 amqspout 向队列中放入一条消息。5 秒后用 amqsget 将其取出，监控窗口得到 Performance Event (MQRC_Q_SERVICE_INTERVAL_HIGH)。这时队列属性变为 QSVCIEV(OK)。
3. 用 amqspout 向队列中放入一条消息。在 5 秒内用 amqsget 将消息取出，监控窗口得到 Performance Event (MQRC_Q_SERVICE_INTERVAL_OK)。这时队列属性又变回 QSVCIEV(HIGH)。
4. 用 amqspout 向队列中再放入一条消息，5 秒内放入第二条消息，监控窗口没有反应，这时队列属性仍为 QSVCIEV(HIGH)。再过 5 秒后放入第三条消息，监控窗口得到 Performance Event (MQRC_Q_SERVICE_INTERVAL_HIGH)。这时队列属性变为 QSVCIEV(OK)。
5. 用例程 MQGET 将消息快速逐条取出，取到第二条时，监控窗口得到 Performance Event (MQRC_Q_SERVICE_INTERVAL_OK)，这时队列属性再次变回 QSVCIEV(HIGH)。

这个实验说明，OK 和 HIGH 开关始终在相互切换，切换到哪里，就期待哪种 Event。具体的 Queue Service Interval Event 和队列属性 Queue Service Interval (QSVCINT) 之间存在一定的算法关系（如前所述），从算法中可见，Queue Service Interval Event 更加关注 MQGET 的频率和时机，因为这标志着服务程序的活动能力。MQGET 可以产生 OK Event，也可以产生 HIGH Event。MQPUT 只能产生 HIGH Event，这标志着在一段时间内有消息连续入队列，却没有消息出队列。

Queue Depth Event 和 Queue Service Interval Event 可以配合起来使用，以确定事件的严重程度。如果再配合队列深度查询，可以更精确地知道事件是否需要人工干预。当然，Event 只能帮助我们发现问题，要具体解决问题还需要额外制定事件规则，编写相应的触发程序。

10.2 性能设计 (Performance)

在大多数情况下,人们对传递消息的中间件主要考察的是功能,比如是否有多样的编程方式,是否支持消息触发,是否能支持各种操作系统平台,是否能与各种现有系统连接等等。对于性能,由于此类中间件是用来连接两套能独立工作或具有独立功能的应用系统的,保持双方松耦合可能是系统的设计思想,而双方之间的通信量和通信频率通常都不会太大。所以对消息传递的性能要求往往不是大多数应用项目所追求的重要目标,到是对应用中消息处理的性能问题会考虑得更多一些。

然而,在个别大容量高性能处理的应用场景中,对消息传递也有相当的要求。在这种项目中,往往消息传递中间件扮演类似于数据总线(Data Bus)的角色,贯串整个应用系统,功能模块之间可以实现松耦合。事实上,WebSphere MQ 作为传递消息的中间件,其性能也是相当优秀的,足以承担大量高频消息传递的需求。

WebSphere MQ 应用程序的运行性能可能受到机器计算能力,吞吐能力,内存等硬件设备的制约,也可能受到网络环境的影响。然而,就应用软件而言,WebSphere MQ 主要受到来自两方面的影响,其一是应用设计,其二是产品配置。

前者往往在性能上起决定作用。例如,设计消息属性为持久性还是非持久性,就可能在设计之初就决定了两者的运行性能相差数倍。选用不同的编程模式也可能在相当大的程度上影响系统性能。对于队列是采用多读多写模式还是单读单写模式,也可能在效果上有很大不同。频繁地对队列管理器连接再断连、将队列打开再关闭等等不合理的使用方式也会使性能大打折扣。

后者对性能的影响稍小,但可以在不改动应用的前提下,根据实际情况,通过改变配置达到性能优化的效果。

由于影响性能的条件很多,所以我们在以下的讨论中都是假定其它条件不变,对某一个条件变化时考虑其对性能的影响。通常情况下,人们往往从响应时间和吞吐量两个指标来考察性能,在我们为实验设计的应用场景中采用了先入先出(FIFO)的模式,通过配置保证消息没有积压现象,所以可以近似地认为响应时间是一个稳定的值。这样,通过考察吞吐量就可以反映 WebSphere MQ 的传输效率。

就 WebSphere MQ 的性能而言,主要分两部分。一部分是队列管理器的工作性能,主要表现在应用与之打交道的 MQGET/MQPUT 的工作效率,为了隔离其它因素的影响,我们在讨论这一部分时在单机上进行,不加入网络传输部分。另一部分是数据传递的性能,主要表现在通道在单位时间内的吞吐量,在讨论这一部分时在两台相同配置的机器上进行,中间使用专用的网络。

10.2.1 队列管理器性能比较

10.2.1.1 实验环境

以下的实验数据皆来自作者的笔记本电脑环境。

- 硬件

- IBM Thinkpad T21 (PIII 处理器, 500MB 内存)
- 操作系统
 - Windows 2000 Professional (中文版)
 - Service Pack 4
- 软件
 - WebSphere MQ for Win2000 5.3
 - CSD05

10.2.1.2 应用场景

10.2.1.2.1 场景一

对于队列中的消息进行多读多写，每个消息 2KB。Put 进程不断地写消息到队列中，Get 进程不断地从队列中取走消息并进行统计。由于 WebSphere MQ 中 MQPUT 的速度往往高于 MQGET 的速度，所以在测试中 Put 进程配置一个，Get 进程配置多个，以均衡数据流量。图 8-3。

对于 JMS 而言，读消息可以是同步方式 (Sync)，也可以是异步方式 (Async)，实验中会分别进行比较。

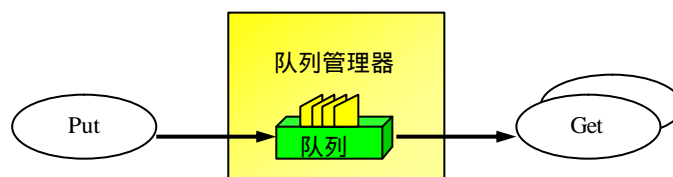


图 8-3 场景一

10.2.1.2.2 场景二

对于请求和应答队列多读多写，每个消息 2KB。Request 进程写消息到请求队列中，然后在应答队列上等待相应的应答消息，请求消息被 Reply 进程取走后生成相应的应答消息放入应答队列，被等候的 Request 取走，一笔环型消息 (Round Message) 处理结束。该 Request 发出下一个请求消息…。由于 Request 和 Reply 进程其实是串行处理的，所以可以适当添加 Request 进程，以保持 Reply 忙。图 8-4。

对于 JMS 而言，读消息可以是同步方式 (Sync)，也可以是异步方式 (Async)，实验中会分别进行比较。

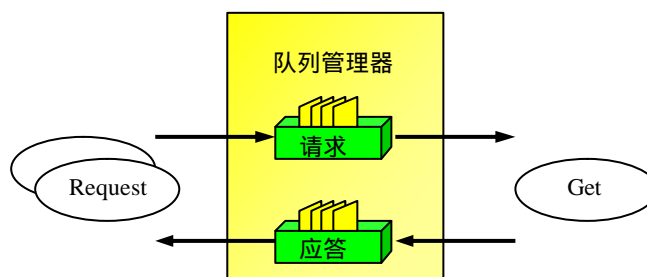


图 8-4 场景二

10.2.1.3 C MQI

10.2.1.3.1 场景一

使用 C MQI 编程的应用在场景一中的实测数据如表 8-5。

表 8-5 C MQI 应用在场景一中实测数据							
配置	进程	消息数	持久消息		消息数	非持久消息	
			总用时	吞吐量		总用时	吞吐量
	(每个)	(笔)	(秒)	(笔/秒)	(笔)	(秒)	(笔/秒)
1 个 Put	Put	100000	330	303	100000	82	1219
1 个 Get	Get	100000	330	303	100000	82	1219
1 个 Put	Put	100000	59	1694	100000	29	3448
2 个 Get	Get	50000	59	847	50000	29	1724
1 个 Put	Put	100000	31	3225	100000	27	3703
3 个 Get	Get	33333	31	1075	33333	27	1234
1 个 Put	Put	100000	28	3571	100000	27	3703
4 个 Get	Get	25000	28	892	25000	27	925

8.5.1.3.2 场景二

使用 C MQI 编程的应用在场景二中的实测数据如表 8-6。

表 8-6 C MQI 应用在场景二中实测数据							
配置	进程	消息数	持久消息		消息数	非持久消息	
			总用时	吞吐量		总用时	吞吐量
	(每路)	(笔)	(秒)	(笔/秒)	(笔)	(秒)	(笔/秒)
1 个 Request	Req/Rep	10000	23	434	10000	5	2000
1 个 Reply							
2 个 Request	Req/Rep	10000	74	135	10000	11	909
1 个 Reply							
3 个 Request	Req/Rep	10000	106	94	10000	19	526
1 个 Reply							
4 个 Request	Req/Rep	10000	142	70	10000	26	384
1 个 Reply							

10.2.1.4 Java Base

10.2.1.4.1 场景一

使用 Java Base 编程的应用在场景一中的实测数据如表 8-7。

表 8-7 Java Base 应用在场景一中实测数据							
配置	进程	消息数	持久消息		消息数	非持久消息	
			总用时	吞吐量		总用时	吞吐量

	(每个)	(笔)	(秒)	(笔/秒)	(笔)	(秒)	(笔/秒)
1 个 Put	Put	100000	333	300	100000	106	943
1 个 Get	Get	100000	334	299	100000	113	884
1 个 Put	Put	100000	200	500	100000	108	925
2 个 Get	Get	50000	200	250	50000	108	462
1 个 Put	Put	100000	153	653	100000	111	900
3 个 Get	Get	33333	153	217	33333	111	300
1 个 Put	Put	100000	126	793	100000	109	917
4 个 Get	Get	25000	126	198	25000	109	229

8.5.1.4.2 场景二

使用 Java Base 编程的应用在场景二中的实测数据如表 8-8。

表 8-8 Java Base 应用在场景二中实测数据

配置	进程 (每路)	持久消息			非持久消息		
		消息数 (笔)	总用时 (秒)	吞 吐 量 (笔/秒)	消 息 数 (笔)	总用时 (秒)	吞 吐 量 (笔/秒)
1 个 Request 1 个 Reply	Req/Rep	10000	54	185	10000	35	285
2 个 Request 1 个 Reply	Req/Rep	10000	110	90	10000	72	138
3 个 Request 1 个 Reply	Req/Rep	10000	163	61	10000	106	94
4 个 Request 1 个 Reply	Req/Rep	10000	219	45	10000	142	70

10.2.1.5 Java JMS (MQ)

以 WebSphere MQ 5.3 作为 JMS Provider。使用 JMS 编程的应用在场景一中的实测数据如表 8-9。

10.2.1.5.1 场景一

表 8-9 Java JMS (MQ) 应用在场景一中实测数据

配置	同步方式 (Sync) 进程 (每个)	持久消息			非持久消息		
		消息数 (笔)	总用时 (秒)	吞 吐 量 (笔/秒)	消 息 数 (笔)	总用时 (秒)	吞 吐 量 (笔/秒)
1 个 Put	Put	10000	46	217	10000	43	232
1 个 Get	Get	10000	46	217	10000	43	232
1 个 Put	Put	10000	41	243	10000	42	238
2 个 Get	Get	5000	41	121	5000	42	119
1 个 Put	Put	10000	41	243	10000	42	238
3 个 Get	Get	3333	41	81	3333	42	79

1 个 Put	Put	10000	42	238	10000	42	238
4 个 Get	Get	2500	42	59	2500	42	59
异步方式 (Async)		持久消息			非持久消息		
配置	进程	消息数	总用时	吞吐量	消息数	总用时	吞吐量
	(每个)	(笔)	(秒)	(笔/秒)	(笔)	(秒)	(笔/秒)
1 个 Put	Put	10000	63	158	10000	41	243
1 个 Get	Get	10000	69	144	10000	41	243
1 个 Put	Put	10000	66	151	10000	42	238
2 个 Get	Get	5000	66	75	5000	42	119
1 个 Put	Put	10000	65	153	10000	42	238
3 个 Get	Get	3333	65	51	3333	42	79
1 个 Put	Put	10000	66	151	10000	42	238
4 个 Get	Get	2500	67	37	2500	42	59

8.5.1.5.2 场景二

以 WebSphere MQ 5.3 作为 JMS Provider。使用 JMS 编程的应用在场景二中的实测数据如表 8-10。

表 8-10 Java JMS (MQ) 应用在场景二中实测数据

同步方式 (Sync)		持久消息			非持久消息		
配置	进程	消息数	总用时	吞吐量	消息数	总用时	吞吐量
	(每路)	(笔)	(秒)	(笔/秒)	(笔)	(秒)	(笔/秒)
1 个 Request	Req/Rep	1000	11	90	1000	12	83
1 个 Reply							
2 个 Request	Req/Rep	1000	25	40	1000	21	47
1 个 Reply							
3 个 Request	Req/Rep	1000	36	27	1000	33	30
1 个 Reply							
4 个 Request	Req/Rep	1000	49	20	1000	43	23
1 个 Reply							
异步方式 (Async)		持久消息			非持久消息		
配置	进程	消息数	总用时	吞吐量	消息数	总用时	吞吐量
	(每路)	(笔)	(秒)	(笔/秒)	(笔)	(秒)	(笔/秒)
1 个 Request	Req/Rep	1000	14	71	1000	13	76
1 个 Reply							
2 个 Request	Req/Rep	1000	25	40	1000	22	45
1 个 Reply							
3 个 Request	Req/Rep	1000	40	25	1000	34	29
1 个 Reply							
4 个 Request	Req/Rep	1000	52	19	1000	45	22
1 个 Reply							

10.2.1.6 Java JMS (SUN J2EE 1.3.1)

为了与 WebSphere MQ 5.3 作为 JMS Provider 的性能进行比较，我们将同样的实验在以 SUN J2EE 1.3.1 作为 JMS Provider 的环境下重新做了一遍，机器环境和应用场景都完全相同。

10.2.1.6.1 场景一

以 SUN J2EE 1.3.1 作为 JMS Provider，使用 JMS 编程的应用在场景一中的实测数据如表 8-11。

表 8-11 Java JMS (SUN) 应用在场景一中实测数据

同步方式 (Sync)		持久消息			非持久消息		
配置	进程	消息数	总用时	吞吐量	消息数	总用时	吞吐量
	(每个)	(笔)	(秒)	(笔/秒)	(笔)	(秒)	(笔/秒)
1 个 Put	Put	1000	89	11	6000	58	103
1 个 Get	Get	1000	206	4	6000	98	61
1 个 Put	Put	1000	108	9	6000	68	88
2 个 Get	Get	500	146	3	3000	95	31
1 个 Put	Put	1000	96	10	6000	77	77
3 个 Get	Get	333	96	3	2000	92	21
1 个 Put	Put	1000	94	10	6000	133	45
4 个 Get	Get	250	94	2	1500	125	12
异步方式 (Async)		持久消息			非持久消息		
配置	进程	消息数	总用时	吞吐量	消息数	总用时	吞吐量
	(每个)	(笔)	(秒)	(笔/秒)	(笔)	(秒)	(笔/秒)
1 个 Put	Put	1000	78	12	6000	84	71
1 个 Get	Get	1000	119	8	6000	183	32
1 个 Put	Put	1000	88	11	6000	118	50
2 个 Get	Get	500	87	5	3000	138	21
1 个 Put	Put	1000	88	11	6000	100	60
3 个 Get	Get	333	87	3	2000	100	20
1 个 Put	Put	1000	89	11	6000	98	61
4 个 Get	Get	333	89	2	1500	98	15

8.5.1.6.2 场景二

以 SUN J2EE 1.3.1 作为 JMS Provider，使用 JMS 编程的应用在场景二中的实测数据如表 8-12。

表 8-12 Java JMS (SUN) 应用在场景二中实测数据

同步方式 (Sync)		持久消息			非持久消息		
配置	进程	消息数	总用时	吞吐量	消息数	总用时	吞吐量
	(每路)	(笔)	(秒)	(笔/秒)	(笔)	(秒)	(笔/秒)
1 个 Request	Req/Rep	100	42	2	100	28	3

1 个 Reply								
2 个 Request	Req/Rep	100	72	1	100	44	2	
1 个 Reply								
3 个 Request	Req/Rep	100	111	0	100	65	1	
1 个 Reply								
4 个 Request	Req/Rep	100	161	0	100	89	1	
1 个 Reply								
异步方式 (Async)		持久消息			非持久消息			
配置	进程	消息数	总用时	吞吐量	消息数	总用时	吞吐量	
	(每路)	(笔)	(秒)	(笔/秒)	(笔)	(秒)	(笔/秒)	
1 个 Request	Req/Rep	100	38	2	100	28	3	
1 个 Reply								
2 个 Request	Req/Rep	100	66	1	100	46	2	
1 个 Reply								
3 个 Request	Req/Rep	100	95	1	100	65	1	
1 个 Reply								
4 个 Request	Req/Rep	100	126	0	100	86	1	
1 个 Reply								

10.2.1.7 结果分析

通过对以上实验数据的观察，我们可以从不同的角度分析出一些结果。

10.2.1.7.1 编程模式的比较

第一，从编程模式上讲，很显然 C 的效率最高，Java Base 其次，MQ JMS 再次。图 8-5。这是因为 C 语言是编译执行的，运行效率非常高。Java 是解释执行的，虽然在设计上相当严谨，但在操作上很容易出现对象的反复申请和释放，使效率大打折扣。MQ JMS 实际上是建立在 MQ Java Base 基础之上的对 JMS Interface 的实现，效率自然会比 MQ Java Base 更差一些。

第二，同一种编程模式中，非持久消息的效率比持久消息高。这是因为非持久消息只需要读写队列，而持久消息还需要更新日志 (Log)，这就多了一倍的 I/O 处理，自然会慢一些。另外，非持久消息会被首先记录在内存中，缺省为 64KB，可以通过 DefaultQBufferSize 参数进行调整。如果内存中放不下了，则更多的部分会写入硬盘，即队列对应的文件中。持久消息则不会记入内存而直接写入硬盘中。所以，如果非持久消息在写入队列后很快被读走，则可能不发生硬盘 I/O，这样大大增加了运行效率。

第三，在相同的编程模式和持久性的情况下，多读多写会增加吞吐能力。保持队列读容量大于等于写容量可以使队列中的消息不至出现积压现象，使每一条消息都得到高效的处理，从而保证消息得到最快的响应时间，这也是 WebSphere MQ 应用设计中的一条基本原则。在 C 语言编程时，这一点表现得尤其明显，但是在 Java 和 JMS 中却不明显，这是因为后者自身的运行效率很慢，MQ 的操作比起应用自身的运行几乎可以忽略，应用无法将压力集中在 MQ 的运行上。另外，当高效的 C 应用进程数量增加时，系统 CPU 计算能力会被大量消

耗，在渐近极限时，系统自身会出现计算能力瓶颈，从而使读写效率的增长纯化。

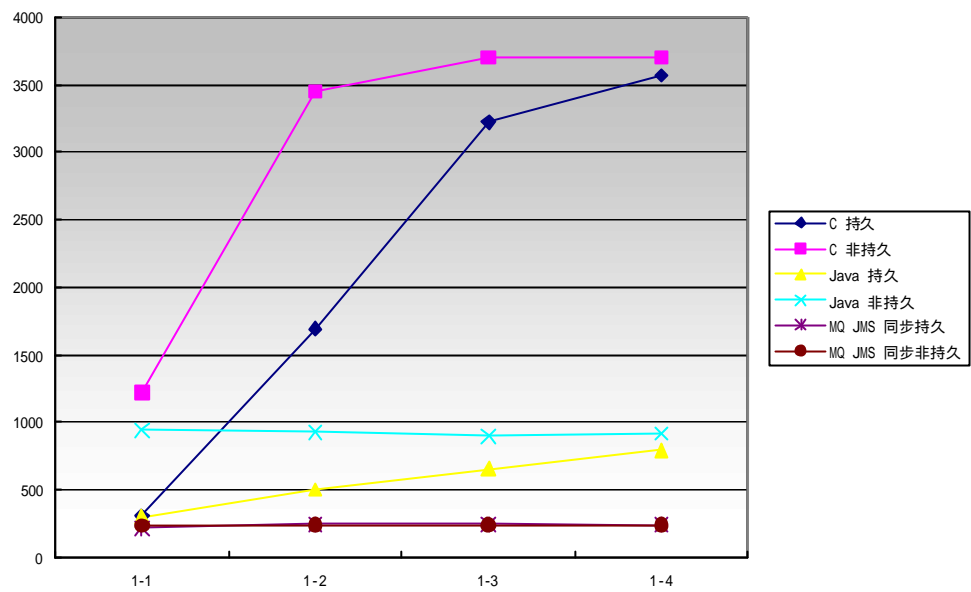


图 8-5 编程模式的比较

10.2.1.7.2 JMS 应用比较

近年来随着 J2EE 的兴起，越来越多的项目使用 JMS 作为消息传递的标准接口。然而，JMS 只是 J2EE 中规定的一套 Java Interface，可以有不同的产品提供其实现方式和运行环境，即 JMS Provider。这里，拿 SUN J2EE 1.3.1 环境中的 JMS 与 WebSphere MQ JMS 比较。图 8-6。

第一，MQ JMS 的效率比 SUN J2EE 1.3.1 中的 JMS 效率高很多。从图中看，几乎以每秒处理 150 笔消息为界，以上部分是 MQ JMS，以下部分是 SUN JMS，泾渭分明。

第二，相同编程模式的比较，非持久消息的读写效率比持久消息高。

第三，JMS 中消息的读取有同步方式和异步方式。同步方式类似于读等待，异步方式类似于消息触发。由于异步方式中消息到达后会引发一系列的后台调用，最终自动调用 MessageListener 接口实现类的 onMessage () 方法，而这一系列的动作会有一定的开销。所以，JMS 同步方式的效率比异步方式高。

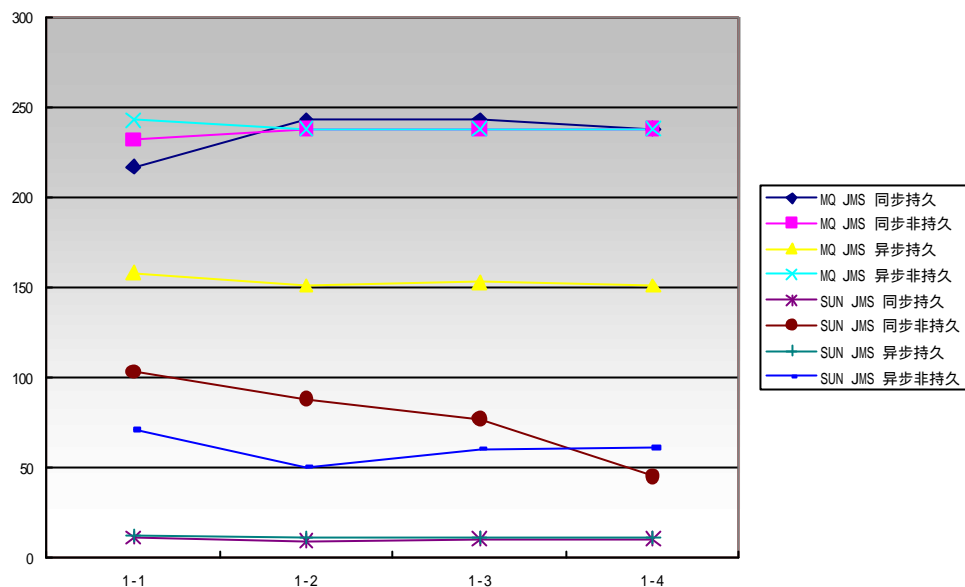


图 8-6 JMS 的比较

10.2.2 数据传递的性能比较

10.2.2.1 实验环境

- 硬件
IBM RS/6000 44p270 小型机 (Power III 375MHz , 2 CPU , 1GB 内存)
- 操作系统
AIX 5.2
Maintenance Level 2
- 软件
WebSphere MQ for Win2000 5.3
CSD05

10.2.2.2 应用场景

两个队列管理器之间建有双向通道，通道上的双向数据传输是独立无关的。应用程序为 C MQI 编程，使用非持久消息，每条消息都是 500 字节。图 8-7。

在高压环境下，消息写入队列 (MQPUT) 的速度可能略高于消息传送的速度，所以消息可能在传输队列中堆积起来，这样会使后面的消息在队列中等待较长的时间，对消息响应时间不利。解决的办法是每次 MQPUT 后休息片刻，用 usleep (微秒) 函数可以调节消息的间距，从而保证响应时间，但是由于压力减小，也会影响整体的吞吐量。通过实验希望找到一个可以兼顾响应时间和吞吐量的平衡点。

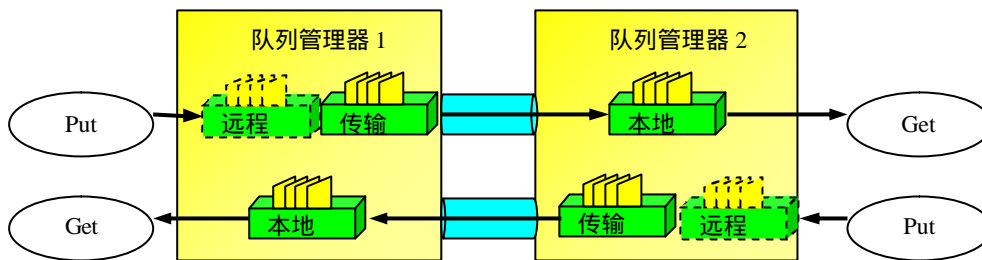


图 8-7 数据跨网络传送应用场景

8.5.2.3 实验数据

我们选取了几种典型的应用场景进行了实验，实测数据如表 8-13。其中单机环境指的是两个队列管理器运行在同一台机器上，通信仍然通过网络连接。双机环境指的是它们运行在两台不同的机器上，用通道通过网络连接。由于 WebSphere MQ 的通道是单向的，所以我们在实验场景中模拟了单向传输和双向传输的情形。对于单向传输也可以采用多条单向通道来增加系统的吞吐能力。

表 8-13 数据跨网络传送应用的实测数据

应用场景描述	消息数(笔)	总用时(秒)	吞吐量(笔/秒)
单机，单队列	1000000	309	3236
单机，单向通道	100000	62	1612
双机，单向通道，usleep 0	100000	38	2631
双机，单向通道，usleep 100	100000	50	2000
双机，单向通道，usleep 200	100000	54	1851
双机，单向通道(两条)，usleep 200	100000	74	1351 x 2
双机，单向通道，usleep 500	100000	83	1204
双机，双向通道，usleep 500	100000	92	1086 x 2
双机，单向通道，群集环境	100000	84	1190

10.2.2.4 结果分析

只有在 usleep 500 的时候，在传输队列中才观察不到消息积压现象，在这个时候的消息吞吐量才是稳定有效的。这说明消息写入与消息传输的速度大约就相差 500 微秒的时间。在实际的应用环境里，由于不可能达到这样的持续压力，在不考虑消息处理的情况下，我们可以姑且将这时的测试值作为系统可能的最大设计吞吐量。

另外，无论双通道是两条同向的还是反向的通道，它们的流量和几乎是单通道的两倍。这说明两条通道的工作是可以互不干扰的。所以，在通道流量成为瓶颈的情况下，不妨考虑增加通道。但是，如果网络带宽已经接近饱和，则开设再多的通道也无济于事。

再者，群集环境下的通道效率也可以达到相当高的水平，如果传输能力达到要求而处理能力达不到，可以考虑配置群集环境，利用多机计算来增加处理能力。

10.2.3 性能优化

10.2.3.1 消息持久性

在应用设计之初就应该考虑消息的持久性属性。如果消息是可以丢失的，或者在出现系统异常时重启队列管理器后不需要保留的，则可以考虑使用非持久消息。然而，几乎没有一个应用设计者会认为自己系统中的消息是“可以丢失的”。但事实上，有很多系统在应用方式上就设计了消息失效的后续矫正处理，比如，在消息应答超时后的自动查询或重发，应用层上的请求确认，消息的发送端和接收端都有数据库记录以便日后校对等等。在这种应用系统中，万一有个别消息丢失也不会造成混乱。这时，如果要追求高性能，可以考虑使用非持久消息。

10.2.3.2 批次

对于稳定批量到达的消息，可以考虑采用批量传送的办法。改变通道上的批量大小和批量间隔参数来调整批次，从而降低网络开销。将其调大，则使多条消息成块传递，提高效率，但往往也意味着延长了响应时间。将其调小，则消息能够更及时地传送，但增加了网络开销。所以，在设定批次的时候，一般会对后续的消息频度和数量有一定的估计。

举一例，白天超市门店系统会自动定时检查货品库存，当发现商品有缺货危险时，会产生消息通知总店尽快配货，由于这种消息的出现是无规律的，需要及时处理，所以批次要尽可能定得小，一旦出现，立即发送。到了夜间，超市门店系统会在下班后将一天的所有交易明细上传到总店，这时会有大批量的数据连续不断地传送，可以考虑适量地加大批次参数，以增加传送效率和速度。

10.2.3.3 快速通道

通道属性中有 NPMSPEED 可以设置为 NORMAL 或 FAST。当 NPMSPEED=FAST 时，我们称之为快速通道。非持久消息在通过快速通道时不受批次控制，这样就不必等到足够多的消息到达或足够长的时间，而直接发送。有利于持久消息和非持久消息混合发送的通道保持高效运作。

10.2.3.4 增加并发

队列是可以被多读多写的，传输队列和通道也是可以被多个远程队列复用的，增加并发无疑可以增加对象的使用效率。

此外，多个 MQ Client 可以连接同一个 MQ Server 进行并发操作，同一个 MQ Client 也可以通过连接池 Connection Pool 与 MQ Server 连接多条连接实现大数据量的并发传送。

多线程也就近年来比较流利的编程方式，它也可以增加并发操作。在 C MQI 编程中，MQ 连接句柄是不可能跨线程的，所以在每个线程中需要分别连接队列管理器并打开对象，这时的多线程工作模式与多进程类似，是可以完全并发操作的。

Java 语言自身有很强的多线程功能,在 Java 编程中多个线程可以共享同一个对象句柄,对该对象的操作会自动互斥,但是这种设计本身就使得多线程中的操作变成了串行操作。所以,Java 程序中是不能利用多线程进行完全的并发操作的。

10.2.3.5 调用方式

在 MQI 所有的 API 中,最慢的是 MQCONN/MQDISC,次慢的是 MQOPEN/MQCLOSE。所以,对于队列管理器频繁地连接断连,对于对象频繁地打开关闭是一种不合理的使用方式。另外,MQPUT1 相当于 MQOPEN、MQPUT、MQCLOSE 三者的连续调用,所以通常只用于低频度的消息发送。

在交易中,MQGET、MQPUT 在提交前会将消息锁住,这会使相应的对象工作受到影响,同时大批量的消息提交或回滚都会引起相当的系统开销。所以,大交易和长交易会在一定程度上影响系统性能,程序中当应用允许时应当尽快通过 MQCMIT、MQBACK 将交易提交或回滚以释放资源。

10.2.3.6 编程模式

不同的编程模式对相同的应用功能的实现效率可能天差地别,在应用设计时应该兼顾到这一点。Java Base 或 JMS 应用往往设计严谨、通常性好,但效率上略逊一畴。

10.2.3.7 触发

应用触发可以使程序在大多数时候处于非活动 (Inactive) 状态,在需要的时候自动被触发调起,处于活动 (Active) 状态,进行消息处理。完毕后,释放资源,又处于非活动状态。这可以使应用尽可能少占用资源,对于低频度的消息处理非常有效。但是,一旦消息大量到达,应用进程被反复触发启动,这时反而显得效率低下。同时,启动进程的额外开销也使消息的响应时间稍慢。

10.2.3.8 设立缓冲区

非持久消息会首先放入队列的缓冲区中,这个缓冲区缺省为 64KB,可以通过 DefaultQBufferSize 参数调整大小。例:

Windows:

```
HKEY_LOCAL_MACHINE\Software\IBM\MQSeries\CurrentVersion\Configuration\QueueManager\<QMgrName>\TuningParameters
DefaultQBufferSize =128000
```

UNIX:

```
TuningParameters
DefaultQBufferSize =128000
```

在缓冲区放满后,消息会放入相应的队列文件中,这样就会产生硬盘 I/O,大大降低了运行效率。所以,将队列缓冲区调大可以增加性能。

持久消息不会使用缓冲区，而将消息直接写入硬盘文件中。此外，持久消息需要记录日志。因此，设置日志的缓冲区大小，日志文件大小，日志文件数量都会对性能有所影响。LogBufferSize 较大，则一次写入硬盘中的消息数量较多，适合于大消息和大交易。一般来说，日志文件较大而文件数量较少与文件较小而数量众多相比，更有效率。此外，日志完整性参数 (LogWriteIntegrity)，在很大程度上也会影响运行性能。

另外，将日志文件和对象文件安装到不同的硬盘上去可以大大增加日志的运行效率。

10.2.3.9 通道 MCA 类型

通道的 MCA 类型通常有两种选择：进程或线程。无论对于发送端或接收端，MCA 都意味着有一个独立的工作单元。如果选择进程，则有一对通信进程来负责通道上的消息传送，一旦进程被杀死，则该通道断开。如果选择线程，则由一对通信线程管理这条通道，所有的通信线程附生在相同的系统通信进程中。

由此可见，进程的独立性好，但开销较大，线程则相反。在有大量通道的应用中，可以考虑线程，降低运行成本。

10.2.3.10 群集

在系统性能出现瓶颈时，应当首先查明瓶颈是发生在消息处理还是消息传送。如果是前者，可以考虑群集方案，将消息散列到不同的队列管理器中进行处理，利用多个计算结点进行分布式计算。如果是后者，可以考虑开启多通道，以增加传输带宽。

另外，利用分布列表 (Distribution List) 也可以降低网络开销。

10.2.4 小结

影响 MQ 性能的因素很多，性能优化也没有一个固定的模式，在很多情况下需要根据实际情况进行设计和配置上的调整。关键是找出瓶颈，对症下药，解决了关键问题，往往能达到神奇的优化效果。

此外，在 IBM WebSphere MQ SupportPac 网站上有大量关于各种平台的性能测评报告及分析，可供参考。

第 11 章 安全协议

近年来，随着 Internet 的普及网络安全越来越受到人们的普遍重视，安全套接字协议层 (Security Socket Layer -- SSL) 协议已逐渐成为一种安全标准被人们广为采用。WebSphere MQ 5.3 开始支持 SSL，这是以前 MQSeries 5.1、5.2 所没有的功能。

11.1 安全通信

安全通信的概念十分广泛，技术也非常复杂。然而，其核心是建立在对称或非对称密码体制上的。

在对称密码体制中，通信双方持有相同的密钥。信息发送方用密钥将信息加密，密文到达接收方后，可以用同一个密钥将其解密。在非对称密码体制中，通信双方持有不同的密钥，发送方用自己的私钥将信息加密，接收方用对应的公钥解密，如果能解开，说明信息的确来自私钥持有者。反过来，发送方用接收方的公钥解密，接收方用自己的私钥解密，如果能解开，说明信息的确是给自己的。公钥和私钥是一对密钥，一起产生的，由公钥加密的信息可以由私钥解密，反过来，由私钥加密的信息也可以由公钥解密。(图 7-1)

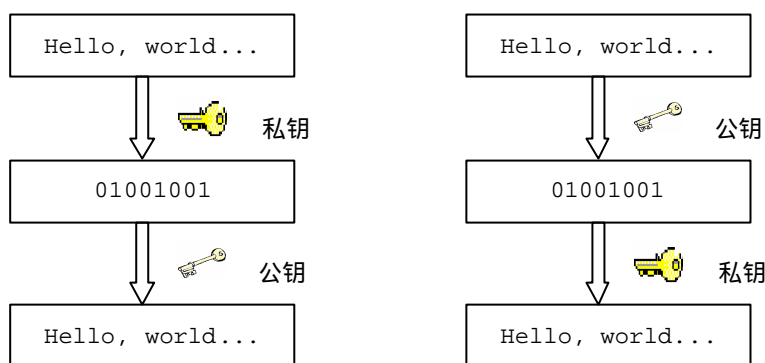


图 7-1 私钥与公钥

11.1.1 数据加密

数字证书采用公钥体制，即利用一对互相匹配的密钥进行加密、解密。每个客户可以设定特定的仅为本人所知的私有密钥（私钥），用它进行数据解密和签名；同时设定一把公共密钥（公钥）并由本人公开，为一组客户所共享，用于数据加密和验证签名。

当发送一份保密文件时，发送方使用接收方的公钥对数据加密，而接收方则使用自己的私钥解密，这样信息就可以安全无误地到达目的地了。

数字加密是一个不可逆过程，即只有使用私有密钥才能解密。在公开密钥密码体制中，常用的是 RSA 体制。其数学原理是将一个大数分解成两个质数的乘积，加密和解密用的是两个不同的密钥。即使已知明文、密文和加密密钥（公开密钥），想要推导出解密密钥（私密密钥），在计算上是不可能的。按现在的计算机技术水平，要破解目前采用的 1024 位 RSA 密钥，需要上千年的计算时间。

公开密钥体系解决了密钥发布的管理问题，客户可以公开其公开密钥，而保留其私有密钥。使用者可以使用接受方的公开密钥对发送的信息进行加密，安全地传送到对方，然后由接受方使用自己的私有密钥进行解密。

在一个发送方同时面对多个接收方需要加密传送数据时，发送方需要持有多个接收方的公钥。(图 7-2) 如果接收方众多，会给发送方的密钥管理带来一定的要求。

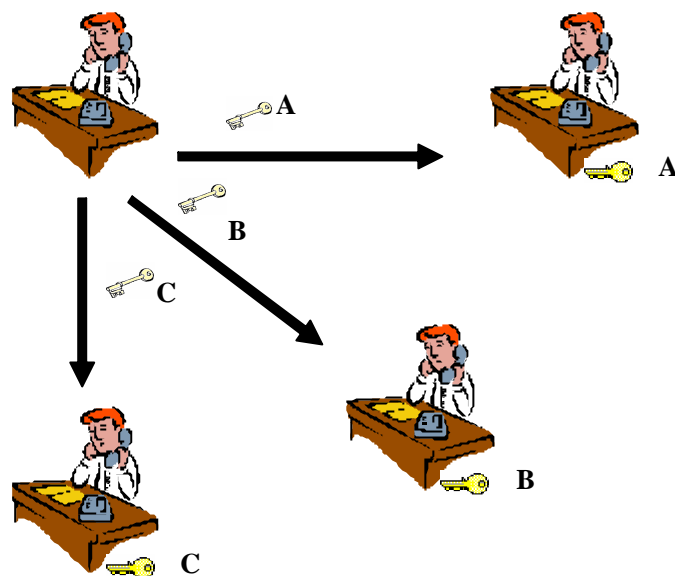


图 7-2 数据加密

11.1.2 报文摘要

报文摘要又称为数字摘要法 (Digital Digest) 或数字指纹法 (Digital Finger Print)。目前主要的算法是 SHA (Secure Hash Algorithm) 和 MD5 (MD Standard For Message Digest) 两种。它们都采用单向 Hash 函数将需加密的明文“摘要”成一串 128bit 的密文，这一串密文亦称为数字指纹 (Finger Print)，它有固定的长度，且不同的明文摘要必定不一致。这样这串摘要便可成为验证明文是否是“真身”的“指纹”了。

报文摘要算法公开，实现方便，在数学上不可逆算，只要改动原文中的任何一个字节，都会造成摘要上的显著不同。所以，如果发送方将生成的摘要用私钥加密后与原文合在一起发送，则即便是明文发送，途中其它人也无法改动消息。图 7-3。

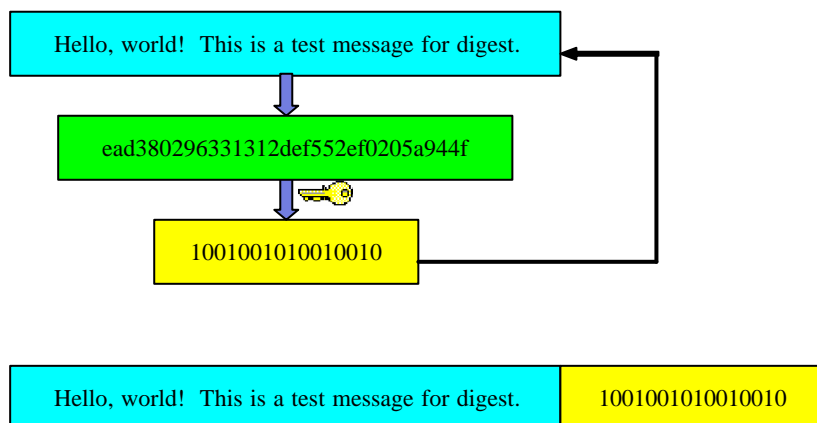


图 7-3 报文加密

11.1.3 数字签名

用户可以采用自己的私钥对信息加以处理，由于密钥仅为本人所有，这样就产生了别人无法生成的信息，也就形成了数字签名。采用数字签名，能够确认以下两点：

1. 保证信息是由签名者自己签名发送的，签名者不能否认或难以否认；
2. 保证信息自签发后到收到为止未曾作过任何修改，签发的文件是真实文件。

数字签名具体做法是：(图 7-4)

1. 将报文按双方约定的 HASH 算法计算得到一个固定位数的报文摘要。在数学上保证，只要改动报文中任何一位，重新计算出的报文摘要值就会与原先的值不相符。这样就保证了报文的不可更改性。
2. 将该报文摘要值用发送者的私人密钥加密，然后连同原报文一起发送给接收者，而产生的报文即称数字签名。
3. 接收方收到数字签名后，用同样的 HASH 算法对报文计算摘要值，然后与用发送者的公开密钥进行解密解开的报文摘要值相比较，如相等则说明报文确实来自所称的发送者。

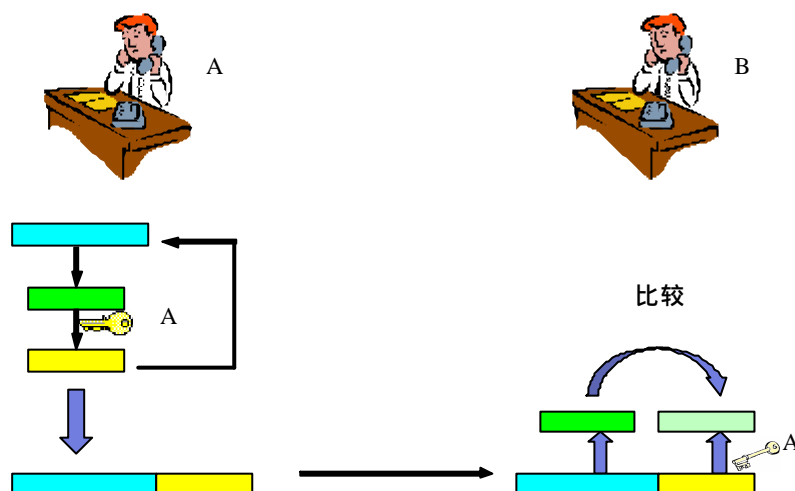


图 7-4 数字签名

近年来，安全通信体系又在数字签名之上发展起来时戳签名、代码签名等等，其原理与数字签名相似。只是签名的对象不同，以达到不可否认的效果。前者对一个时戳进行签名，这在瞬息万变的交易环境十分重要，比如黄金、股票、期货交易，买卖双方都不可否认价格和此价格对应的时间，这样的约束条件才有意义。后者是对一段代码进行签名，对著作维权有用，比如全球性的自主开发项目，众多的开发者合力开发一个项目，在过去不少人喜欢将作者名写入代码中，但这并不能防止代码被抄袭挪作它用，代码签名就可以解决这一问题。只要签名还在，就无法否认作者。

11.1.4 SSL

在安全的通信环境下，既有数字签名，也有数据加密。例如：消息从 A 送往 B。在 A 处，先用 A 的私钥做数字签名，再用 B 的公钥做数据加密。消息送达 B 后，先用 B 的私钥解密，如果能解开，说明消息的确是送给 B 的，再用 A 的公钥验证数字签名，如果通过，说明消息的确来自 A。反过来，消息从 B 送往 A 也类似。(图 7-5)

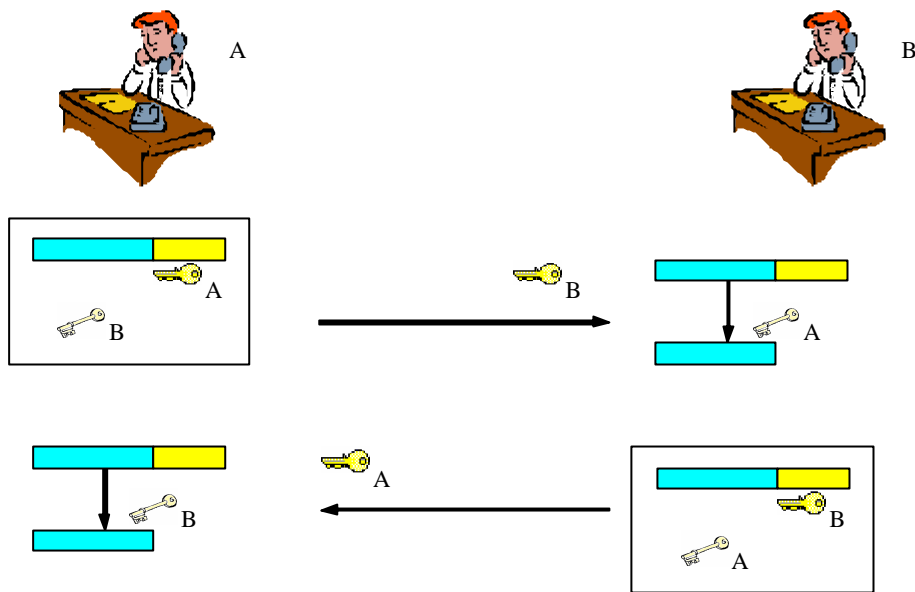


图 7-5 SSL

所以，在通信的任何一侧至少有两个密钥：自己的私钥和对方的公钥。可见在安全通信环境建立之初会有一个互换公钥的过程。

SSL 就是一个安全的通信环境。SSL 安全协议最初是由 Netscape Communication 公司设计开发的，又叫“安全套接层（Secure Sockets Layer）协议”，主要用于提高应用程序之间的数据的安全系数。SSL 协议的整个概念可以被总结为：一个保证任何安装了安全套接字的客户和服务端间通信安全的协议，它涉及所有 TC/IP 应用程序。SSL 协议充分考虑到安全通信中需要的数据加密、报文摘要、数字签名等等。

SSL 安全协议主要提供三方面的服务：

- 用户和服务器的合法性认证

认证用户和服务器的合法性，使得它们能够确信数据将被发送到正确的客户机和服务器上。客户机和服务器都是有各自的识别号，这些识别号由公开密钥进行编号，为了验证用户是否合法，安全套接层协议要求在握手交换数据进行数字认证，以此来确保用户的合法性。
- 加密数据以隐藏被传送的数据

安全套接层协议所采用的加密技术既有对称密钥技术，也有公开密钥技术。在客户机与服务器进行数据交换之前，交换 SSL 初始握手信息，在 SSL 握手信息中采用了各种加密技术对其加密，以保证其机密性和数据的完整性，并且用数字证书进行鉴别。这样就可以防止非法用户进行破译。
- 保护数据的完整性

安全套接层协议采用 Hash 函数和机密共享的方法来提供信息的完整性服务，建立客户机与服务器之间的安全通道，使所有经过安全套接层协议处理的业务在传输过程中能全部完整准确无误地到达目的地。

一旦双方建立了 SSL 连接，两者间的信息传送就会在 SSL 协议框架下加密传送。而 SSL 协议框架保证了信息安全，从而达到防窃取、防篡改、防伪装、防抵赖的目的。

11.2 数字证书

11.2.1 概念

我们现实世界里，经常能看到各种身份证件，比如身份证、驾驶证、学生证、营业执照等等。它们或者是为了证明个人身份，或者是为了证明某种资格。在计算机的虚拟世界中，也有对应的概念，通过数字化的身份证件来验证网上身份。这种数字身份证件就是通常所说的数字证书。

现实世界里的通信如果需要保密，人们往往会采用某种手段将信息包裹起来，比如使用信封或者使用双方约定的暗语等等。在虚拟世界中，这种包裹信息的手段通常称为加密。数字证书由于含有公钥，也可能含有私钥，所以可以参与加密。

现实世界里的信息发送者通常会用某种方法来标识自己，比如签名，接收者可以以此来验证发送者的身份。即便在使用身份证件的时候，接收者也可以检查证件的合法性，当然包括发证机构的合法性。在虚拟世界中，数字签名可以标识信息发送方的身份，数字证书中含有签发机构的数字签名。

使用数字证书，通过运用对称或非对称密码体制等技术建立起的一套严密的身份认证系统，从而保证：信息不被第三方窃取，信息在传输过程中不被篡改，双方能够确认对方的身份，发送方对于自己送出的信息不能抵赖。

11.2.2 格式

在进一步讨论数字证书之前，让我们先来看一个数字证书的内容。(表 7-2)

表 7-2 数字证书的内容

```
-----BEGIN CERTIFICATE-----
MIIDSzCCApugAwIBAgIIIPn9KwWPUprMwDQYJKoZIhvcNAQEFBQAwZzELMAkGA1UE
BhMCQ04xEDAOBgNVBAGTB0JlaWppbmcxEDAOBgNVBACTB0JlaWppbmcxDjAMBGNV
BAoTBUNBMzY1MSQwIgYDVQQDExtDQTM2NSBUZXR0IFJvb3QgQ2VydGlmZWVhdGUw
NjUxJDAlBgNVBAMTG0NBZyY1IFRlc3QgUm9vdCBDZXJ0aWZpY2F0ZTCCASIwDQYJ
BeRFkRLtHml+BF8piiz/89ToczDZBx87TM2KqqUolQ4usKzB+P/r
-----END CERTIFICATE-----
```

这个数字证书是在国内 ca365 网站 (www.ca365.com) 上测试根证书 (rootTest.der)，从内容上看如同天书，根本无法看懂。数字证书一般含有大量二进制信息，为了保证证书文件在各种平台上兼容，可以将其转换成 ASCII 文本方式保存。在 Windows 中可以用系统缺省的 Crypto Shell Extensions 工具打开，缺省情况下，双击证书文件即可。打开后可以看到这个数字证书的详细信息。(图 7-6)



图 7-6 证书信息

我们发现，原来数字证书中还有有效时间、证书所有人的名称、证书发行者的名称、证书发行者的签名等等，这一切与现实世界里的证件属性十分相似。事实上，目前普遍能用的数字证书大多遵循 ITUT X.509 国际标准格式。X.509 数字证书通常包含以下内容：

- 证书的版本信息；
- 证书的序列号，每个证书都有唯一的证书序列号；
- 证书所使用的签名算法；
- 证书的发行机构名称，命名规则一般采用 X.500 格式；
- 证书的有效期，通用的证书一般采用 UTC 时间格式，它的计时范围为 1950-2049；
- 证书所有人的名称，命名规则一般采用 X.500 格式；
- 证书所有人的公开密钥；
- 证书发行者对证书的签名。

其中，在通信安全意义上最重要的因素是签名算法和公开密钥两部分。

SSL 协议在建立连接的过程中需要交换双方的公钥，以便在以后的消息传送和消息验证时使用。每个证书都含有一个公钥，所以一般说来，在建立连接之前，要求将含有公钥的证书安装到对方通信端。

11.2.3 根签证书与自签证书

通常一个证书可以由上一级证书签署的,这个上一级证书是用来验证下一级证书本身是否有效的手段。而上一级证书又可以由再上一级证书签署,证书的这种级联关系也可以在 Windows 中 Crypto Shell Extensions 工具打开证书时,选择证书路径时看到。通常,我们把应用环境中最高一级证书称为根证书,由根证书签署的下一级证书称为根签证书。

根证书一般由第三方权威机构发放,这些机构可以识别自己曾经发放的证书,提供失效的证书清单等等。而这些机构我们通常称之为 CA 认证中心。打开 Windows Internet Explorer (IE) 浏览器,选择“工具”→“Internet 选项”→“内容”→“证书”,可以看见目前比较流行的 CA 中心签署的根证书:GlobalSign, SecureSign, TrustCenter, Thawte, VeriSign 等等。近年来,国内也建设了一些 CA 中心,比如 ca365,银行,邮电,政府网站等等。在这些 CA 网站上,可以下载根证书,同时可以申请由根证书签署的数字证书(即根签证书),一般根据证书的用途和时限,有些是免费的,有些有收费的。

由于根证书可以生成根签证书,根证书也同样可以验证根签证书。如果通信双方安装了相同的根证书,又各自安装了同源的根签证书,则可以检查对方证书是否合法。

签署证书还有另外一种方式,即自签证书。如果签署证书的上级证书就是自身,这种证书称为自签证书。自签证书顾名思义就是自己给自己签署证书,要对方认证就必须把证书(至少是含证书公钥的那一部分)导出并安装到对方的通信端。利用工具,比如 SSL Toolkit 或 IBM GSK(与 WebSphere MQ 同时安装)生成自签证书文件,要生成只含公钥的证书,可以用工具或 IE 导入导出以滤去私钥,也可以由工具,比如 IBM GSK 来抽取。

所以,要进行 SSL 通信,在安装证书时只有两种方式。一种方式是两侧加载相同的根证书以及由这个共同的根证书签署出来的根签证书,另一种方式是两侧加载己方的自签证书以及(必须含私钥)对方的自签证书(可以是只含公钥的部分)。

11.3 WebSphere MQ 配置 SSL

WebSphere MQ 中的 SSL 体现为通道的属性。这里的通道既可以是 MQ Server 之间的消息通道(Message Channel),也可以是 MQ Client 与 MQ Server 之间的调用界面通道(MQI Channel)。由于 MQ 支持多平台,MQ Client 与 MQ Server 端都是既可能在 Windows 平台,也可能在 UNIX 平台。WebSphere SSL 支持跨平台之间的连接。

无论是 Server 端还是 Client 端的配置,大致都要经历准备证书,配置证书(添加和指定证书),配置通道三步。

11.3.1 Server/Server 消息通道

Server/Server 中 SSL 通道配置与普通的通道配置基本相同,只是在配置时需要在通道两端约定 SSL 加密算法。

11.3.1.1 Windows Server 端

- 准备证书

如果用根签证书，可以从专业的 CA 中心网站下载，比如中国数字认证网 (www.ca365.com)。详细经过可以参见实例 1。

如果用自签证书，可以用工具生成。在安装 WebSphere MQ 的同时，会自动安装 IBM GSK，通常安装目录在 C:\Program Files\IBM\GSK\bin，运行 ikmguw.exe，创建 CMS key database 文件 *.kdb，在其中创建自签证书，再将其导出。

- 配置证书

首先，通过设定队列管理器属性 SSLKEYR，指定队列管理器的证书数据库文件。

```
ALTER QMGR SSLKEYR('D:\WMQ\qmgrs\QM1\ssl\key')
```

指定证书数据库文件为 key.kdb

接着，通过 WebSphere MQ 资源管理器来管理 SSL 证书，来对证书数据库文件编辑，添加和指定证书。在 WebSphere MQ 资源管理器中，点中相应的队列管理器，右键选择属性 → SSL → 管理 SSL 证书，将相应的根证书和下级证书加入，将下级证书指定为资源管理器证书。

- 配置通道

与普通通道配置基本相同，但多了三个通道属性。通道双方必须选用相同的算法，即 SSLCIPH 值。接收端可以设置 SSLPEER 来验证发送端的证书属性，接收端可以设置 SSLAUTH 来决定发送端是否必须发送数字证书。

例：

在队列管理器 QM1 上配置：

```
DEFINE CHANNEL (C_QM1.QM2) CHLTYPE (SDR) TRPTYPE (TCP) CONNAME ('127.0.0.1  
(1416)') XMITQ (QLX_QM2) SSLCIPH (TRIPLE_DES_SHA_US)
```

在队列管理器 QM2 上配置：

```
DEFINE CHANNEL (C_QM1.QM2) CHLTYPE (RCVR) TRPTYPE (TCP) SSLCIPH  
(TRIPLE_DES_SHA_US)
```

11.3.1.2 UNIX Server 端

- 准备证书

如果用根签证书，可以从专业的 CA 中心网站下载。办法与 Windows 相同。

如果用自签证书，可以用工具生成，比如 IBM GSK。一种办法是在 Windows 中将证书文件生成后 FTP 到 UNIX 平台上。另一种办法是在 UNIX 平台上直接运行 IBM GSK。注意，在 UNIX 平台上 IBM GSK 为 gsk6ikm，它也是一个 JAVA GUI 工具，执行时要注意以下几点：

1. 设置 JAVA_HOME 环境变量，要包含以下路径 (表 7-3)

表 7-4 JAVA_HOME环境变量	
平台	JAVA_HOME环境变量
AIX	/usr/mqm/ssl/jre
HP-UX	/opt/mqm/ssl
Solaris	/opt/mqm/ssl

Linux /opt/mqm/ssl/jre

2. 执行 GSK 的用户必须属于 mqm 组
3. 生成的密钥数据库文件 (key.kdb) 和密码文件 (key.sth) 属于执行 GSK 的用户, 对此用户可读写, 对同组可读。为了保证 MCA 可读, 要对文件开放读权限。

```
chmod g+r /var/mqm/ssl/key.kdb
```

```
chmod g+r /var/mqm/ssl/key.sth
```

4. 可以用命令行工具 gsk6cmd 完成同样的工作。

```
gsk6cmd -keydb -create -db filename -pw password -type  
cms -expire days -stash
```

-db filename 指定密钥数据库文件名

-pw password 指定密钥数据库密码

-type cms 指定密钥数据库类型

-expire days 指定有效日期

-stash 表示需要生成密码文件

5. 通过生成自签证书, 导入/导出, 抽取证书。将密钥数据库文件编辑好。

- 配置证书

证书数据库文件已经由工具编辑好了, 通过设定队列管理器属性 SSLKEYR, 指定队列管理器的证书数据库文件。

```
ALTER QMGR SSLKEYR(' /var/mqm/qmgrs/QM1/ssl/MyKey')
```

指定证书数据库文件为 MyKey.kdb

- 配置通道

与 Windows 相同。

11.3.2 Client/Server MQI 通道

Client/Server 中 SSL 通道配置与普通通道配置基本相同。可以使用通道定义表文件, 也可以使用 MQCONN 在程序中指定。

11.3.2.1 Server 端

- 准备证书

与 Server/Server 相同

- 配置证书

与 Server/Server 相同

- 配置通道

通道配置与普通 Client/Server 通道基本相同, 例:

```
DEFINE CHANNEL (C_C.S) CHLTYPE (SVRCONN) TRPTYPE (TCP) MCAUSER ('chenyux')  
SSLCIPH (TRIPLE_DES_SHA_US)
```

```
DEFINE CHANNEL (C_C.S) CHLTYPE (CLNTCONN) TRPTYPE (TCP) CONNAME ('127.0.0.1  
(1415)') QMNAME (QM) SSLCIPH (TRIPLE_DES_SHA_US)
```

在 Server 端生成通道定义表 (Channel Definition Table) 文件。

11.3.2.2 Windows Client 端

- 准备证书

与 Server 端 Windows 做法相同。

- 配置证书

将 Server 端的通道定义表 (Channel Definition Table) 文件拷贝到 Client 端,例如 Server 端 D:\Program Files\IBM\WebSphere MQ\Qmgrs\QM\@ipcc\AMQCLCHL.TAB, 拷贝到 Client 端 C:\Program Files\IBM\WebSphere MQ\AMQCLCHL.TAB。在 Client 端用环境变量指定此文件。(方法与普通同 Client 配置相同)

```
set MQCHLLIB=C:\Program Files\IBM\WebSphere MQ
set MQCHLTAB=AMQCLCHL.TAB
```

1. 将 Server 端的密钥数据库 (Key Repository) 文件拷贝到 Client。例如 Server 端 D:\Program Files\IBM\WebSphere MQ\Qmgrs\QM1\ssl\key.sto 拷贝到 Client 端 C:\Program Files\IBM\WebSphere MQ\key.sto。在 Client 端设置环境变量 MQSSLKEYR 指定此文件。注意, 这里不要写文件扩展名 (.sto)

```
set MQSSLKEYR=C:\Program Files\IBM\WebSphere MQ\key
```

2. 在 Client 端对密钥数据库 (Key Repository) 文件做相应的修改。

```
amqmcert -l // 列出文件中的证书
amqmcert -a -p PersonalCertFile -z Password // 添加证书
amqmcert -a -s CertFile // 添加证书
amqmcert -u // 取消指定证书
amqmcert -d handle // 指定证书
```

例:

```
amqmcert -a -p QM1.pfx -z QM1 // 添加证书 QM1
amqmcert -u // 取消指定的证书
amqmcert -d 01470 // 指定证书
```

- 配置通道

使用通道定义表 (Channel Definition Table) 文件

11.3.2.3 UNIX Client 端

- 准备证书

与 Server 端 UNIX 做法相同。

- 配置证书

1. 将 Server 端的 Channel Definition Table 文件拷贝到 Client 端, 例 Server 端 C:\Program Files\IBM\WebSphere MQ\Qmgrs\QM\@ipcc\AMQCLCHL.TAB, 拷贝到 /var/mqm/AMQCLCHL.TAB 在 Client 端用环境变量指定此文件。(方法与普通同 Client 配置相同)

```
export MQCHLLIB=/var/mqm
export MQCHLTAB=AMQCLCHL.TAB
```

2. 在 Server 端利用工具, 如 IBM GSK 生成密钥数据库文件, 拷贝到 Client 端。在 Client 端设置环境变量 MQSSLKEYR, 指定这个文件。注意, 这里不要写文件扩

展名 (.kdb)

```
export MQSSLKEYR=/var/mqm/key
```

- 配置通道

使用通道定义表 (Channel Definition Table) 文件

11.3.3 SSL 相关的对象属性

WebSphere MQ 中与 SSL 相关的对象属性并不多,集中于队列管理器和通道。其中队列管理器的 SSL 相关属性规定了队列管理器运行环境中 SSL 的相关参数,而通道的 SSL 属性则约定了通信双方的 SSL 加密算法、数字认证字串、SSL 认证过程等等。

11.3.3.1 队列管理器属性

WebSphere MQ 中 SSL 有效的证书文件是存放在密钥数据库中的,而过期的证书由认证吊销列表指定,队列管理器用 SSLKEYR 和 SSLCRLNL 属性分别指定密钥数据库和认证吊销列表的位置(表 7-5)。有些特殊的 UNIX 服务器会提供一些加密硬件进行数据加密,加密算法通常固化在硬件中,加密速度非常快。可以用队列管理器的 SSLCRYP 属性对这些加密硬件进行参数配置。

表 7-5 队列管理器中与 SSL 相关的属性

属性	解释	说明
SSLKEYR	SSLKeyRepository	指定 Key Repository 的文件名,缺省为 <QM>/ssl/key.sto
SSLCRLNL	SSLCRLNameList	指定一个 MQ NameList 名称,其含有 SSL CRL 认证吊销列表
SSLCRYP	SSLCryptoHardware	仅为 UNIX 平台有效。对加密硬件配置参数。
SSLTASKS	SSLTasks	仅对 z/OS 平台有效。接到 SSL 调用后启动的任务,可以有多个
SSLKEYRPWD	SSLKeyRepositoryPassword	仅为 OS/400 平台有效。访问 KeyRepository 时的口令。

11.3.3.2 通道属性

通道要使用 SSL 协议则必须约定通信双方的 SSL 加密算法,在通道定义中由 SSLCIPH 属性指定。在 SSL 握手时,发送方会将数字证书的相关信息送出,其中证书的 DN 字串可以在接收端进行验证,所以在接收端通道的 SSLPEER 属性必须设置相同的 DN 字串。SSLCAUTH 表示是否需要将整个数字证书送到对方。表 7-6 列出了通道定义中与 SSL 相关的属性。

表 7-6 通道对象中与 SSL 相关的属性

属性	解释	说明
SSLCIPH	CipherSpec	指定 SSL 加密算法,最长 32 字节 如果为空,表示不用 SSL 协议 通道双方必须选用相同的算法
SSLPEER	Distinguished Name	数字认证签署时的 DN,例如: 'CN=QM1, OU=SWG, O=IBM, L=Shanghai, S=Shanghai, C=CN' 表示在认证证书的时候,要检查对方证书的 DN 是否与设置相同。
SSLCAUTH	SSL Client Authorization	SSL Server 是否要求 SSL Client 送数字认证 ● REQUIRED 表示 SSL Client 必须送出数字证书,缺

- 省。
- OPTIONAL 表示 SSL Client 可以不送数字证书。

11.3.3.3 Client 的相关环境变量

使用 SSL 协议的 Client/Server 连接也要用 MQSERVER 环境变量来与队列管理器相连，如果使用通道定义表文件，则由 MQCHLLIB 和 MQCHLTAB 环境变量来指定目录名和文件名，这些与普通的 Client/Server 连接相同。Client 端的证书存放在证书配置库 (SSL Key Repository) 文件中，由 MQSSLKEYR 环境变量来指定配置库文件。表 7-7 列出了所有相关的 Client 端环境变量。

表 7-7 与 SSL 相关的环境变量	
环境变量	说明
MQSERVER	服务器连接
MQSSLKEYR	SSL Key Repository 文件
MQCHLLIB	通道定义表目录名
MQCHLTAB	通道定义表文件名

11.3.4 Client 端程序

如同普通的 MQ Client 端编程，在 Client 端可以使用 MQCONN 来指定所需要的所有参数，不需要通道定义表文件，不需要设置环境变量。在 MQCONN 中 MQCNO.ClientConnPtr 可以指向 MQCD，其中可以指定 ChannelName, TransportType, ConnectionName。MQCNO.SSLConfigPtr 可以指向 MQSCO，其中可以指定 SSLCipherSpec, KeyRepository。具体程序参见 MQCONNSSLClient.c。

11.3.5 证书部署

数字证书的应用涉及到需要身份认证及数据安全的各个行业，广泛应用于电子商务和电子政务：如支付型和非支付型电子商务活动，包括传统的商业、制造业、流通业的网上交易。各自独立的政府部门之间形成了松散的结构，为了实现跨部门之间的新的业务需求，必须加强部门之间的有机联系。由于业务的多样，参与结点可能多变，造成了环境的复杂性。这也是电子政务要解决的问题。诸如，公共事业、银行保险证券、社保医疗民政、人事劳动用工、工商税务海关、政府行政办公、教育科研单位等部门的网上申报、网上审批、网上办公等应用……

在实际的应用场景中，我们应该根据需要来实施和部署。如果采用了 WebSphere MQ 来实现 SSL 通信，则首先应该考虑的是认证方式，是采用根签证书还是自签证书。在考虑采用 SSL 的同时就应该考虑 CA 认证中心，是在项目中建设一个呢，还是采用所有结点都能接受的第三方 CA 机构。如同没有认证中心来发放统一的根证书，则只能采用自签证书的方式。

根签证书比较规范，实施简单，在每个结点只需要安装一个根证书和一个根签证书即可，但申请过程比较复杂。自签证书申请过程比较简单，可以用工具生成，但是为了安全起见，

每个证书需要有含私钥和不含私钥两种形式,每个结点需要安装自己含私钥的自签证书以及所有其它通信方的不含私钥的自签证书。实施过程比较麻烦,如果结点过多,对众多证书的管理也会成为一个问题。

一般说来,根签证书适合多结点的复杂环境,尤其适合多对多的通信环境。对电子商务和电子政务应用都可以大显身手。自签证书适合少结点的简单环境,尤其适合一对一或少量结点的一对多的通信环境,在企业内网或没有 CA 中心的应用环境下相当实用。

让我们来看一个银行代收公用事业费的例子:一家银行可以代收水、电、煤、话等不同的公用事业费,所以这家银行可能与各家公用事业单位连接。而任何一家公用事业单位都可能与多家银行有类似的代收业务关系。在这样一个多对多的通信环境中,如果都采用 WebSphere MQ 进行通信,则可以屏蔽不同的通信协议,最大程度地减少通信开发工作。在这个例子中,如果通信上采用 SSL 协议,则可以用政府网站的 CA 中心,也可以用金融系统的 CA 中心来管理、签署、发放证书,以及下载最新的证书吊销列表。

11.4 实例 1 根签证书

以下以配置 Windows 上两个队列管理器 QM1 和 QM2 之间的一条 SSL 通道 (C_QM1.QM2) 为例,对如果使用根签证书和自签证书两种方式分别进行举例说明。

11.4.1 准备证书

可以到 CA 认证网站下载根证书并申请由此根证书签发的数字证书,也可以用工具生成自签证书。证书经过转换可以导出为不同的形式。但它们大致分成两类,一类含私钥与公钥,用作已方的发送,此类证书往往有口令保护 (*.p12, *.pfx)。另一类只含公钥不含私钥,用作对方认证,此类证书可以随便分发 (*.cer)。在 Windows 中,含有私钥的证书文件称为个人信息交换文件,不含私钥的证书文件称为安全证书文件,使用时无需口令。两者的图标也是不一样的,前者因为有口令保护,所以多了一个钥匙符号。(图 7-7)

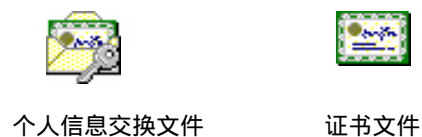


图 7-7 证书文件

登录 CA365 网站 (www.ca365.com), 下载测试根证书 (测试证书区, 根 CA 证书)。申请测试数字证书 (用表格申请证书)。在申请表格中填入相关的资料, 在证书用途一档选择通用证书, 提交。(图 7-8)

The screenshot shows a Microsoft Internet Explorer window with the address bar displaying <http://www.ca365.com/>. The website header features the CA365 logo and the title '中国数字认证网'. A left sidebar contains a navigation menu with links such as 'English', '首页', '用户须知', '用户手册', '技术论坛', '价格付款', '关于我们', '管理员信箱', and 'ca365@sohu.com'. The main content area is titled '申请测试证书' and contains a form for identifying information and certificate options. The form fields are as follows:

识别信息:	
名称:	QM1
公司:	My Company
部门:	My Department
城市:	Shanghai
省:	Shanghai
国家(地区):	CN
电子邮件:	chen_jonathan@163.net
网址:	http://
证书期限:	三十天

Below the identification section, the '证书用途:' (Certificate Purpose) dropdown is set to '通用证书' (General Certificate). The '密钥选项:' (Key Options) section shows '加密服务提供:' (Encryption Service Provider) as 'Microsoft Base Cryptographic Provider v1.0' and '密钥用法:' (Key Usage) with radio buttons for '交换' (Exchange), '签名' (Signature), and '两者' (Both), where '两者' is selected.

图 7-8 申请证书

申请证书，为 QM1 和 QM2 分别申请一个证书，申请成功后出现确认网页。

将证书下载后，共得到三个证书文件，不妨假定根证书名为 rootTest.der，QM1 和 QM2 的根证书分别为 QM1.der，QM2.der。

导入证书

在 IE 中将证书文件 QM1.der 导入，在菜单中选择“工具”、“Internet 选项”、“内容”“证书”、“导入”→选择证书文件 QM1.der 导入→在导入时，根据证书类型，自动选择证书存储区。完成后可以在证书对话框中看到 QM1 证书。(图 7-9)

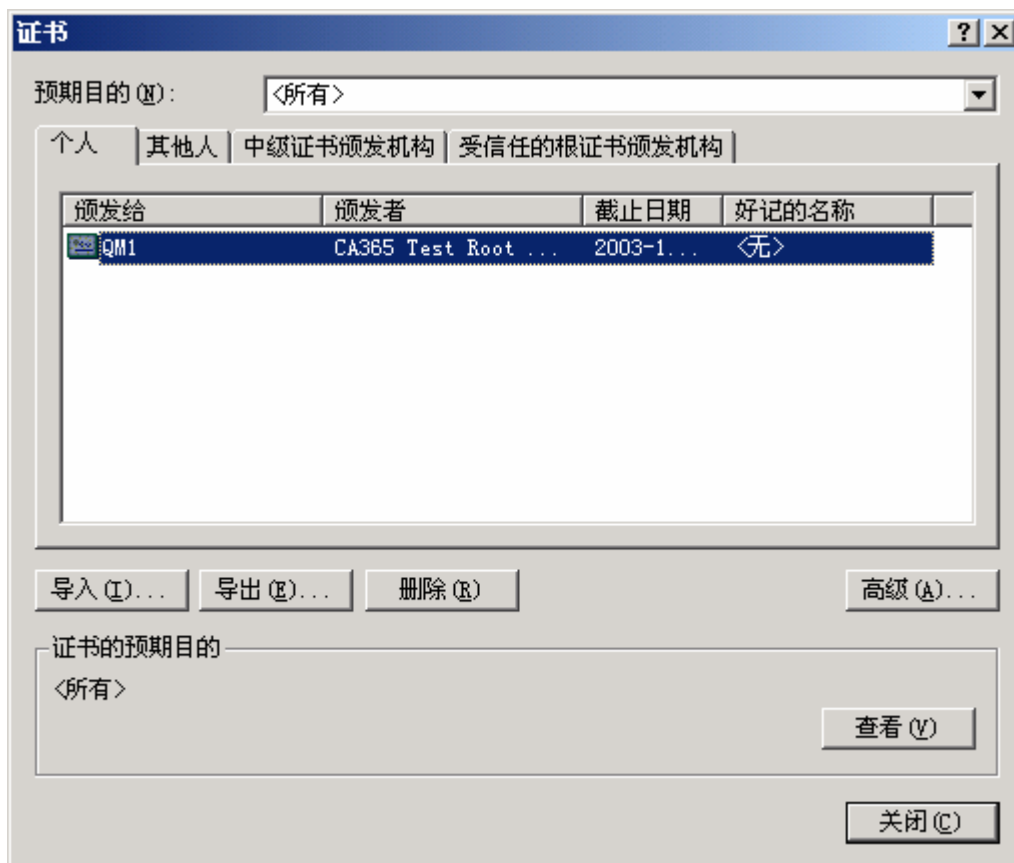


图 7-9 导入后的证书

在证书对话框中将证书导出成为个人信息交换文件。在对话框中选中 QM1 实施导出，在导出过程中选择“私钥与证书一起导出”→ 可以在导出文件格式中选择“私人信息交换”并选择“启动加强保护”→ 设定并确认密码，不妨设为 QM1 → 设置文件名 QM1.pfx → 完成。

这样，就完成了从 QM1.der 到 QM1.pfx 的转换。QM1.pfx 中既含有公钥也含有私钥。一旦生成了个人信息交换文件 (*.pfx) 后，原先的数字证书文件就可以删除了。注意：IE 中删除了证书对话框中的证书以后，就无法用再次用 QM1.der 导入了。

类似地，对 QM2.der 也做类似的操作，得到 QM2.pfx。现在，我们有了一个根证书文 rootTest.der 和两个个人信息交换文件 QM1.pfx 和 QM2.pfx。

11.4.2 配置队列管理器

11.4.2.1 添加 SSL 证书

在 WebSphere MQ 资源管理器中，选中 QM1 → 右键，选择属性 (图 7-10) → SSL → 管理 SSL 证书 → 可以将原来缺省安装的数字证书全部删除，添加 → 从文件导入，选择文件 QM1.pfx，输入刚才设置的口令 QM1 → 添加，将 QM1 证书添加入队列管理器的证书仓库中 → 再添加 CA365 Test Root Certificate → 如果已经在系统的证书仓库中，则选中

后添加，否则，与前面类似，从文件导入。

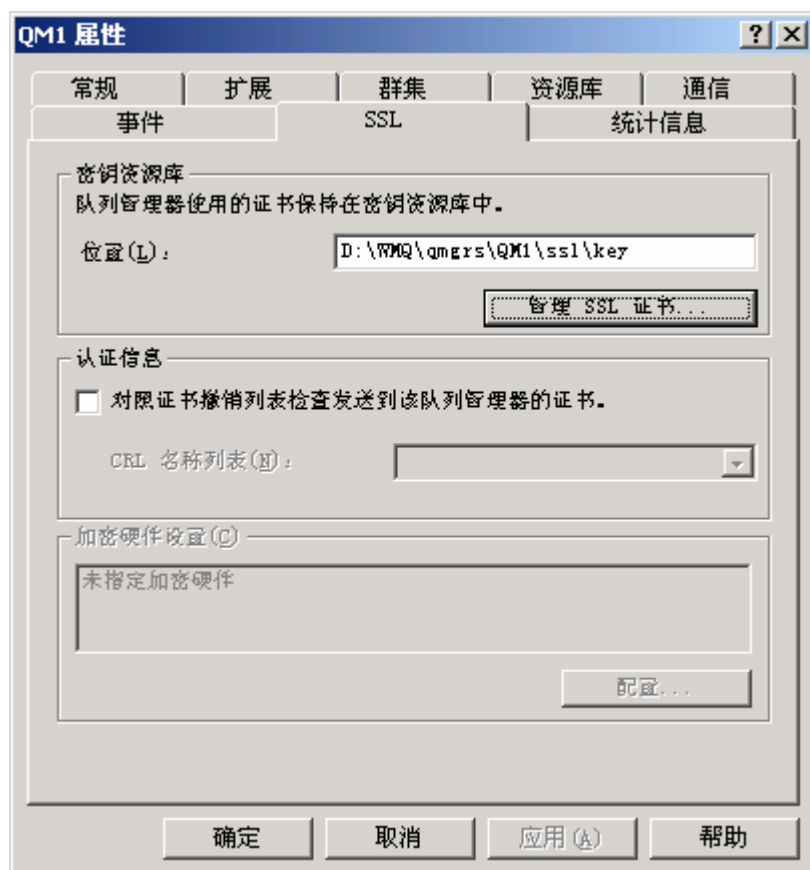


图 7-10 添加 SSL 证书

事实上，在队列管理器创建的时候，系统会把缺省的证书仓库文件 <InstallDir>\!DEFAULT.STO 复制到队列管理器的证书仓库文件 <InstallDir>\Qmgrs\<QM>\ssl\key.sto。所以，在管理 SSL 证书的时候可以看见这些缺省证书。如果事先将 !DEFAULT.STO 文件清空，则新创建的队列管理器缺省不含有任何证书。

11.4.2.2 指定 SSL 证书

在证书仓库对话框中选中的 QM1，指定 → 在指定队列管理器证书中选中的 QM1，指定。
注意：管理 SSL 证书对话框中的被指定的证书图标上会出现绿色的勾。(图 7-11)

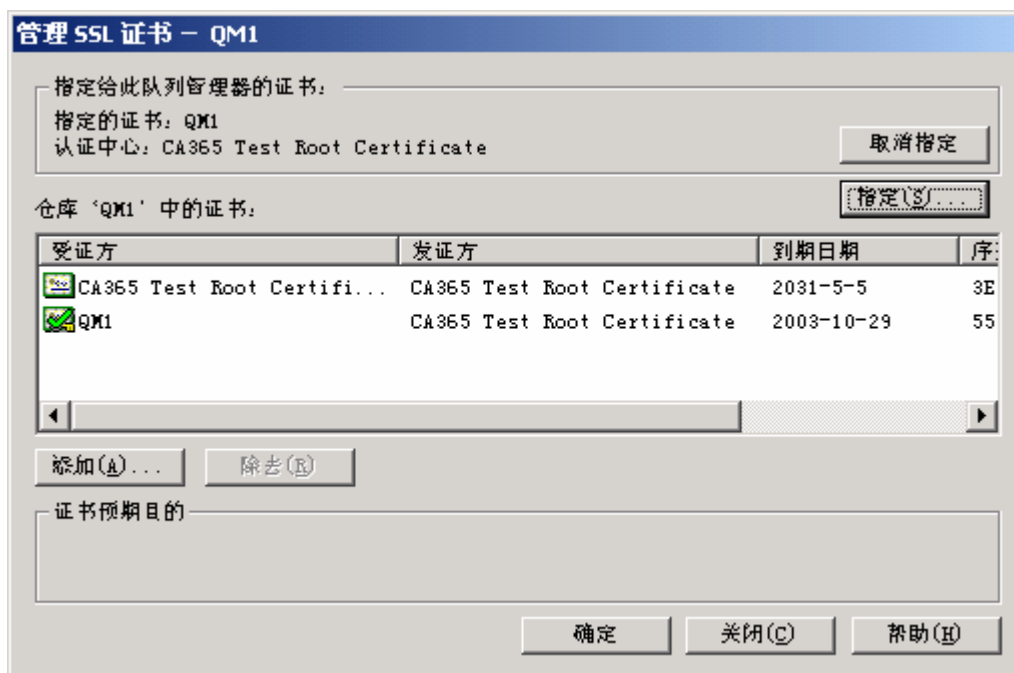


图 7-11 指定 SSL 证书

类似地，对队列管理器 QM2 也做类似的操作。

11.4.3 配置通道

在队列管理器 QM1 上配置：

```
DEFINE CHANNEL (C_QM1.QM2) CHLTYPE (SDR) TRPTYPE (TCP) CONNAME ('127.0.0.1
(1416)') XMITQ (QLX_QM2) SSLCIPH (TRIPLE_DES_SHA_US)
```

在队列管理器 QM2 上配置：

```
DEFINE CHANNEL (C_QM1.QM2) CHLTYPE (RCVR) TRPTYPE (TCP) SSLCIPH
(TRIPLE_DES_SHA_US)
```

注意：SSLCIPH 指定的加密算法在通道两侧必须一致。

11.5 实例 2 自签证书

11.5.1 准备证书

在安装 WebSphere MQ for Windows 的时候会自动装入 IBM GSK 密钥管理工具，通常安装在 C:\Program Files\IBM\GSK\bin 目录下，运行这一工具，可以生成自签证书，也可以将证书导出成为含私钥的个人信息交换文件或不含私钥的证书文件。

在菜单中选择密钥数据库文件 → 新建 → 保留原来的缺省设置，创建 CMS 密钥数据库文件 → 确定 → 输入数据库文件保护口令，不妨用 GSK → 确定。这样就创建了密钥数据库文件，缺省为 key.kdb。

新创建的密钥数据库文件中含有缺省的证书（图 7-12），可以将它们一个一个地删除。

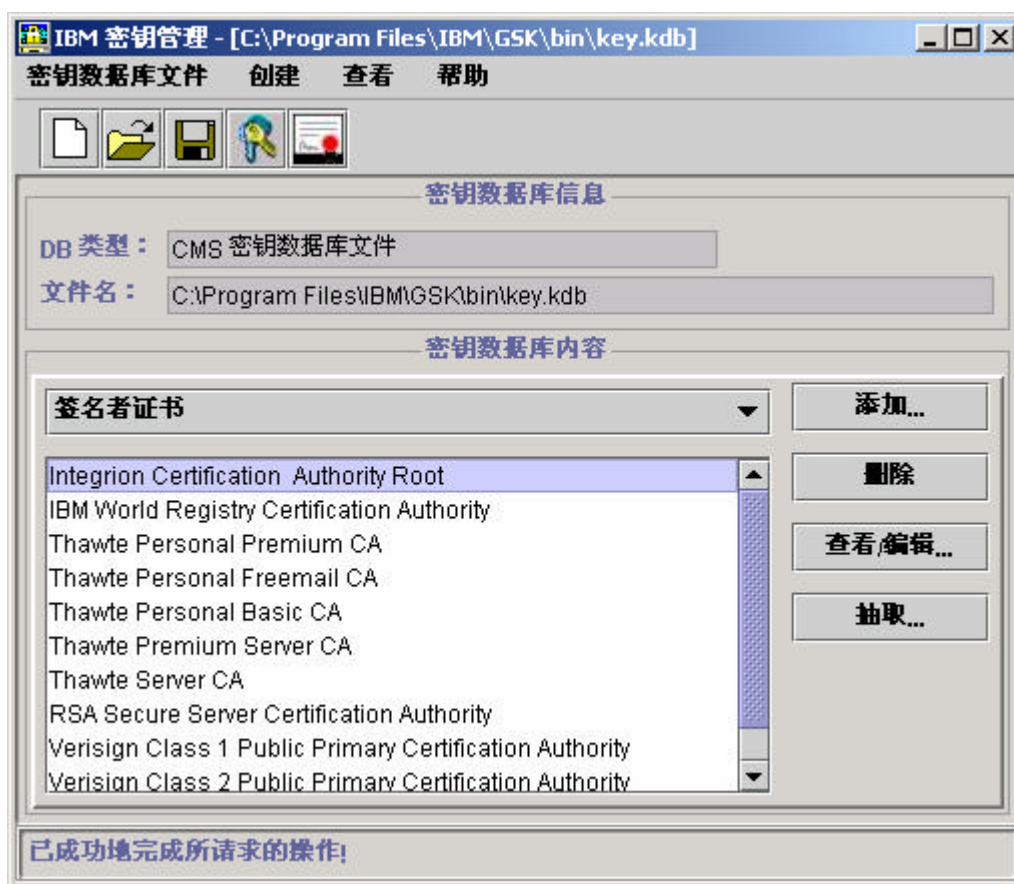


图 7-12 密钥数据库文件中的缺省证书

在菜单上选择创建 → 新建自签证书 → 填入相关的资料 (图 7-13) → 确定。



图 7-13 创建自签证书

选中证书 QM1 ,在界面上选择导入/导出 → 设置导出密钥文件名 ,不妨设为 QM1.p12 (图 7-14) → 确定 → 设置密钥的保护口令 ,不妨设为 QM1 → 确定。

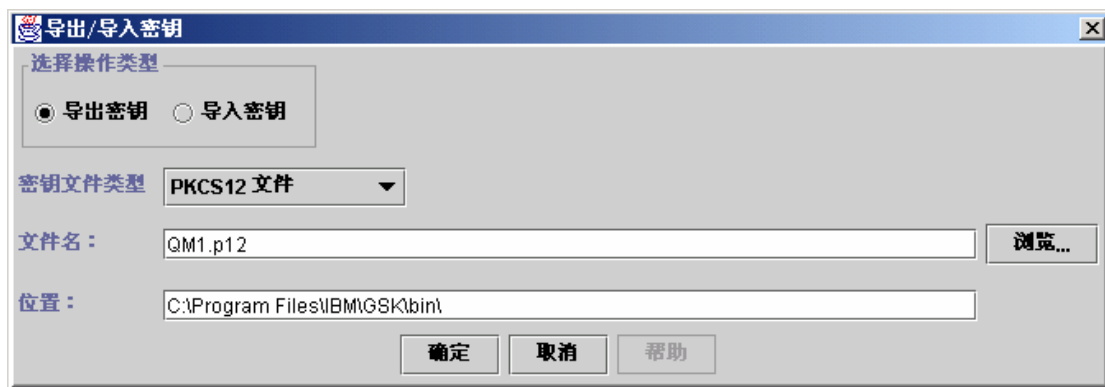


图 7-14 指定自签证书文件名

选中证书 QM1 ,在界面上选择抽取证书 → 在数据类型一档中可以选择 Base64 编码的 ASCII 数据 ,也可以选择二进制 DER 数据 ,建议后者 → 设定证书文件名 → 确定。(图 7-15)

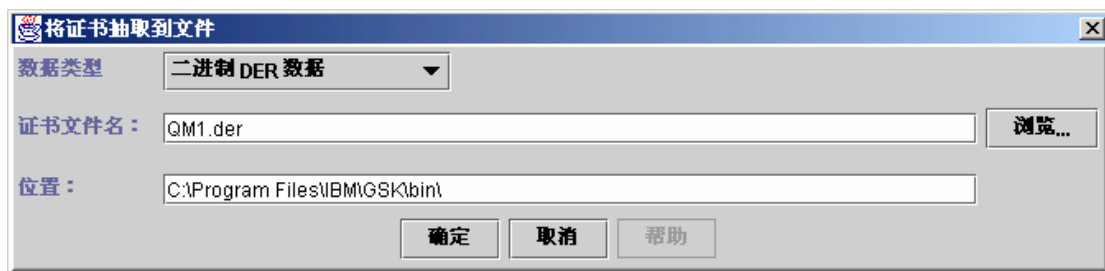


图 7-15 抽取自签证书

类似地，对 QM2 做类似的操作。

至此，我们有了自签的个人信息交换文件 QM1.p12 和 QM2.p12 以及由此抽取的证书 QM1.der 和 QM2.der。其中，*.p12 含私钥，用来安装在己方，作为指定证书；*.der 不含私钥，用来安装在对方，用作对证书的认证。*.arm (Base64 编码的 ASCII 数据) 与 *.der (二进制 DER 数据) 效果是相同的，只是前者是文本文件 (表 7-8)，后者是二进制文件。

表 7-8 抽取证书以文本文件存放

```
-----BEGIN CERTIFICATE-----
MIIBDCCAR6gAwIBAgIIDxzgUmuArZowDQYJKoZIhvcNAQEEBQAwPzELMAkGA1UE
BhMCQ04xGDAWBgNVBAoTD015IE9yZ2FuaXphdGlvbjEWMBQGA1UEAxMNMTkyLjE2
OC4yMjAuMTAeFw0wMzA5MzAwMDI1MzZaFw0wNDA5MzAwMDI1MzZaMD8xCzAJBgNV
BAYTAkNOMRgwFgYDVQQKEw9NeSBPcmdhbml6YXRpb24xFjAUBgNVBAMTDTE5MjE4
NjguMjEwLjEwXjEwXDANBgkqhkiG9w0BAQEFAANLADBIAkEA2+8HemK/BTxJav5VpnM
8d/w3BBn5kno9FlkVihHk5qqP2I4+DhFEKgxv+HSn1lgGf66DezfQOXX5NW5RNs9
xwIDAQABMA0GCSqGSIb3DQEBAUAA0EAYiV1u6Qn50LB9b02gJO3SuSXAgrvzpjD
E9h0TuJN9b+T3WJ1ZQ19fzDjYhCsHh0X+DQXT03Yuz3dtZjrw/eqEg==
-----END CERTIFICATE-----
```

11.5.2 配置队列管理器

11.5.2.1 添加 SSL 证书

与实例 1 类似，在队列管理器 QM1 中添加个人信息交换文件 QM1.p12 和抽取的证书文件 QM2.der。在队列管理器 QM2 中添加个人信息交换文件 QM2.p12 和抽取的证书文件 QM1.der。

11.5.2.2 指定 SSL 证书

与实例 1 类似，在队列管理器 QM1 中指定 QM1 为指定 SSL 证书。在队列管理器 QM2 中指定 QM2 为指定 SSL 证书。证书 QM1 和 QM2 上分别出现绿色的勾标记。

11.5.3 配置通道

在队列管理器 QM1 上配置：

```
DEFINE CHANNEL (C_QM1.QM2) CHLTYPE (SDR) TRPTYPE (TCP) CONNAME ('127.0.0.1
(1416)') XMITQ (QLX_QM2) SSLCIPH (TRIPLE_DES_SHA_US)
```

在队列管理器 QM2 上配置：

```
DEFINE CHANNEL (C_QM1.QM2) CHLTYPE (RCVR) TRPTYPE (TCP) SSLCIPH
(TRIPLE_DES_SHA_US)
```

注意：SSLCIPH 指定的加密算法在通道两侧必须一致。

第 12 章 用户出口

用户出口是 WebSphere MQ 中的高级功能，它可以帮助用户将自己编写的程序嵌入 WebSphere MQ 运行环境中，并在适当的时候被自动调用。由于用户出口程序是内嵌在 MQ 中运行的，如果使用得当可以使应用如虎添翼，发挥相当强大的功能，但如果使用不当也可能画蛇添足，反而增添麻烦。另外，对用户出口的编程是比较难的，各种限制也比较多，对于初学者需要谨慎使用。

12.1 概述

就和 IBM 的很多其它产品一样，WebSphere MQ 也有用户出口 (User Exit) 的设计。用户出口实际上是一个函数接口，在特定的条件下产品会调用这个函数。用户可以根据相关规范自行编写该函数，嵌入产品中，使用用户的程序在一定条件下能参与产品的运行。

WebSphere MQ 的 User Exit 共分成五类，其下又有细分：

- Channel Exit
 - Security Exit
 - Message Exit
 - Send Exit
 - Receive Exit
 - Message Retry Exit
 - Channel Auto-Definition Exit
 - Transport-Retry Exit
- Data Conversion Exit
- Cluster Workload Exit
- Pub/Sub Routing Exit
- MQ API Exit
 - MQI Exit
 - ◆ Before MQI Call Exit
 - ◆ After MQI Call Exit
 - INIT & TERM
 - ◆ Initialization Exit
 - ◆ Termination Exit

无论是 Windows 还是 UNIX 环境，User Exit 都会以动态连接库的形式被加载运行，在编译的时候要指定缺省入口函数，在配置时也要指定入口函数。由于 User Exit 自身不是一个独立的可执行程序，在编程上会有一定的限制。一般来说，在实现时尽可能简单，达到目的即可。User Exit 不允许嵌套调用，但允许在设置中提供多个 User Exit，WebSphere MQ 会顺序地调用它们。另外，编程时要注意打开的句柄要关闭，分配的内存要释放等等。下面我们就各种用户出口逐一进行介绍。

12.2 Channel Exit

Channel Exit 是所有 WebSphere MQ 用户出口中最具价值的一类，共分七种：Security Exit、Message Exit、Send Exit、Receive Exit、Message Retry Exit、Channel Auto-Definition Exit 和 Transport-Retry Exit。每一种都有各自的调用时机，如图 12-1。

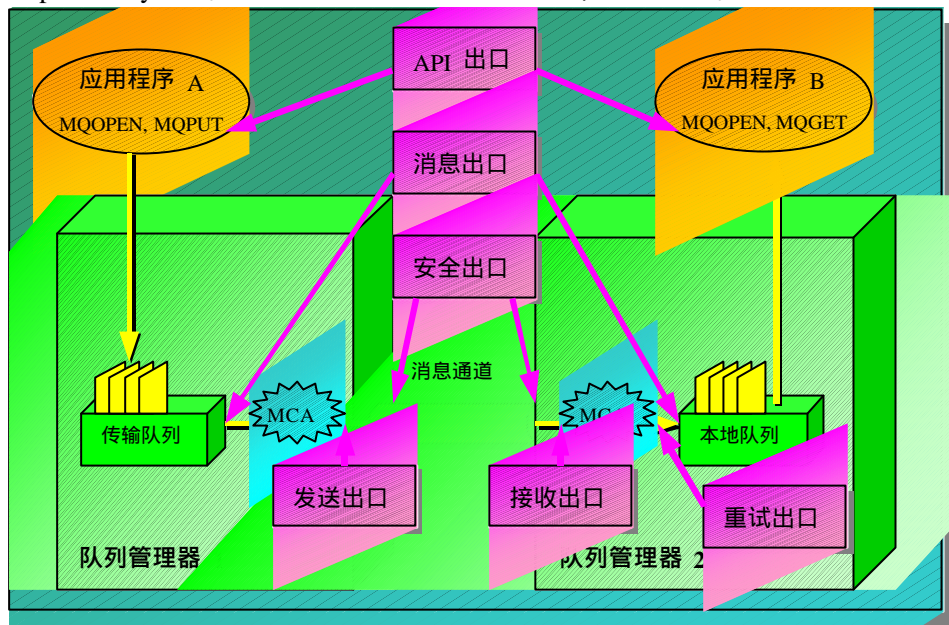


图 12-1 各种通道出口的调用时机

- | | |
|--------------------------------|---------------------------|
| ● Security Exit | 交换安全信息，通常在建立通道时调用 |
| ● Message Exit | 当消息需要完整送出，或已经完整送达时调用 |
| ● Send Exit | 当消息的一部分送出之前调用 |
| ● Receive Exit | 当消息的一部分送达之后调用 |
| ● Message Retry Exit | 在消息到达接收端后无法放入目标队列时调用 |
| ● Channel Auto-Definition Exit | 在发送端启动通道，而接收端未定义同名通道时调用 |
| ● Transport-Retry Exit | 将 UDP 消息发送时调用，可以将消息按住一段时间 |

Security Exit 在建立通道时以某种约定的时序在通道双方各自被调用，Security Exit 之间可以交换自定义的安全信息，进行安全消息通信。通道双方可以以某种方式相互验证，如果有一方出错则通道无法成功建立。

Message Exit 每传送一条消息在发送端和接收端被各自调用一次，在发送端的调用在 MCA 从传输队列中取得消息之后，在接收端则在 MCA 将消息放入目标队列之前。Message Exit 可以访问到 MQMD 和 MQXQH (消息传输头)，也可以访问到消息本身，可以改变消息的长度与内容。一般可以用来加密、解密、压缩、解压，或用来对消息重新打包、解包等等。

Send Exit 的调用在 MCA 将消息发送到网络之前，是在发送端控制下的最后一点。Receive Exit 的调用在 MCA 从网络上接收到消息之后，是在接收端控制下的第一点。WebSphere MQ 通道中消息数据 (Data) 虽然是单向传递的，但控制数据 (Control) 却是通过

通道双向传递的（如表 12-1）。Send Exit 和 Receive Exit 无论在传送控制数据还是消息数据时都会被调用。所以，在同一侧的消息发送中 Send Exit 有可能作为传送数据和控制信息分别被调用，另一侧的 Send Exit 为了送回应答信号也可能被调用。MCA 传送的消息单位是一个消息段 (Transmission Segment)，在消息通道中，如果需要传送的消息太大时，MCA 会自动将其切成多个消息段，而对于每个段，发送端都会调用一次 Send Exit，接收端都会调用一次 Receive Exit。对于 MQI 通道也类似，如果传送的参数太大时，也会引起消息段，Send 和 Receive Exit 也会针对每个段被调用。

表 12-1 通道传送的控制消息和数据消息

	Control	Data
Message Channel	Control Information	Message Data
	双向	单向
MQI Channel	Control Information	MQI Parameters
	双向	双向

Send 和 Recieve Exit 通常可以改变 Message Data、MQI Paramenter 或 Transmission Segment 的长度与内容。但是 Send Exit 不允许修改前 8 个字节，因为前 8 个字节是 MQ 通道协议头 (MQ Channel Protocol Header)，留作 MQ 内部通信用。在通道连接的时候，通道两端会协商消息段的最大长度，在 Sender Exit 修改消息段的内容和长度时不可超过这个限制。

Message Retry Exit 只在接收端通道 (Receiver、Requester 和 Cluster Receiver) 有效，用于在消息放入目标队列不成功时的重试，可以在出口函数中规定重试次数和间隔时间。Channel Auto-Definition Exit 和 Transport-Retry Exit 是针对队列管理器有效的，分别用于通道自动定义和 UDP 消息传输重试。

下面让我们用一个例子说明最常用的四种 Channel Exit 的调用时机，假定发送端和接收端建立通道后发送了两条消息，由于消息太大由通道自动将每条消息分成三个段传送。出口程序的调用次序如表 12-2。

表 12-2 通道出口的调用次序

	发送端	接收端
1	Security Exit	Security Exit
2	Message Exit	
3	Send Exit	Receive Exit
4	Send Exit	Receive Exit
5	Send Exit	Receive Exit
6		Message Exit
7	Message Exit	
8	Send Exit	Receive Exit
9	Send Exit	Receive Exit
10	Send Exit	Receive Exit
11		Message Exit

Channel Exit 都支持 Server/Server 之间的 Message 通道，但并非都支持 Client/Server 之间的 MQI 通道。如表 12-3。

表 12-3 消息通道或 MQI 是否支持通道出口

Channel Exit 名称	支持 Message Channel	支持 MQI Channel
Security Exit	Y	Y
Message Exit	Y	
Send Exit	Y	Y
Receive Exit	Y	Y
Message Retry Exit	Y	
Channel Auto-Definition Exit	Y	Y
Transport -Retry Exit	Y	

表中 Y 表示支持，空格表示不支持

我们知道 ,WebSphere MQ 通道类型较多 ,并不是所有的通道都支持所有的通道出口的 ,表 12-4 详细列出了不同类型的通道对于 Channel Exit 的支持情况。

表 12-4 通道类型对于通道出口的支持

Channel Exit 名称	SDR	SVR	RCVR	RQSTR	CLNTCONN	SVRCONN
Security	Y	Y	Y	Y	Y	Y
Message	Y	Y	Y	Y		
Send	Y	Y	Y	Y	Y	Y
Receive	Y	Y	Y	Y	Y	Y
Message Retry			Y	Y		
Channel Auto-Definition			Y			Y

表中 Y 表示支持，空格表示不支持

12.2.1 Channel Exit 函数

配置通道出口 (Channel Exit) 时需要指明出口的文件名、入口函数、用户参数。在通道建立和运行的时候，WebSphere MQ 会根据握手协议来约定两侧的调用流程，并根据通道的配置在适当的时候加载调用这些函数，在流程中每侧的用户出口函数可能被调用多次。入口参数 ChannelExitParms 结构中的 ExitReason 表示每次被调用时的原因，结构中的 ExitResponse 表示回应码，用户出口程序可以通过设置回应码动态地改变这个流程，同时会影响下一次的调用。

通道中与通道出口相关的属性如下：

- Security Exit Name (SCYEXIT)
- Security Exit User Data (SCYDATA)
- Message Exit Name (MSGEXIT)
- Message Exit User Data (MSGDATA)
- Send Exit Name (SENDEXIT)
- Send Exit User Data (SENDDATA)
- Receive Exit Name (RCVEXIT)
- Receive Exit User Data (RCVDATA)

在设定通道出口属性的时候，一般会同时设置一对 Exit Name 和 Exit User Data。Exit Name 指定出口程序 (可以有多个)，对于每个出口程序，含程序名和入口函数两部分，如：
`SENDEXIT ('C:\SendExit.dll (SendExit)')`。Exit User Data 会作为参数自动传递到函数

入口的 pChannelExitParms -> ExitData 中。

Channel Exit 可以有多个,形成链式结构,系统会顺序地调用它们。在设置 Exit Name 时可以用逗号分开的多个字符串,来指定多个出口程序,如:

```
SENDEXIT ('C:\SendExit1.dll (SendExit)', 'C:\SendExit2.dll (SendExit)')
```

所有的 Channel Exit 具有相同形式的函数入口,如下:

```
void MQENTRY MQChannelExit (  
    PMQVOID pChannelExitParms,      Channel Exit 参数结构, PMQCXP  
    PMQVOID pChannelDefinition,     Channel Definition 结构, PMQCD  
    PMQLONG pDataLength,             AgentBuffer 或 ExitBuffer 中的有效数据长度  
    PMQLONG pAgentBufferLength,      AgentBuffer 长度, 缺省是 32766  
    PMQVOID pAgentBuffer,            Agent Buffer, AgentBuffer 内容  
    PMQLONG pExitBufferLength,       Exit Buffer 长度, 缺省是 0  
    PMQPTR pExitBufferAddr)          Exit Buffer 内容,可以自己 malloc/free。 在  
                                     每次 Channel Exit 调用时共享这块内存。
```

函数参数中指明了两块内存,一块是 AgentBuffer,这是一块预分配的内存,通常用来存放接收到的数据报文。另一块是 ExitBuffer,它可以是用户动态申请的内存,可以被通道同一端的其它 Channel User Exit 共享,有时 ExitBuffer 可以用来存放要送出的额外报文。函数参数中的 AgentBufferLength 和 ExitBufferLength 分别指明两块内存的大小。另有一个参数 DataLength 表示 AgentBuffer 或 UserBuffer 中的有效数据长度。

每个出口程序的 ExitBuffer 是互相独立看不见的。这意味着每个出口程序可以在 MQXR_INIT 时申请自己的 ExitBuffer 空间,这块空间只有在对这个出口程序的多次调用中可见,在 MQXR_TERM 时负责释放 ExitBuffer 空间。

12.2.2 Security Exit

Security Exit 在通道握手时在通道双方各自被调用,根据握手的时序可能在通道一侧被调用多次,每次入口参数都不一样。下面我们对主要的入口参数进行介绍。

入口参数 pChannelDefinition (PMQCD) -> ChannelType 表示所在一侧的通道类型。由于出口程序通常是在通道两端一起使用的,所以在编程实现上经常将两个通道出口程序合在一起,该参数可以在通用出口程序中用来区分通道。

入口参数 pChannelExitParms (PMQCXP) -> ExitReason 表示调用的原因码,不同时机调用的原因码是不同的。比如通道握手时主动方会顺序调用 Security Exit 三次,调用的原因码依次为 MQXR_INIT、MQXR_INIT_SEC、MQXR_SEC_MSG,分别表示通道初始化、安全出口初始化、发送安全报文。安全报文中可以放置任何字符串用以身份验证。

- MQXR_INIT 通道启动时被调用
- MQXR_TERM 通道关闭时被调用
- MQXR_INIT_SEC 初始化 Security 流程时被调用,可以在此安排是否发送 Security 报文及相关流程

- MQXR_SEC_MSG 发送 Security 报文

入口参数 pChannelDefinition (PMQCD) -> ExitResponse 表示对本次调用的回应码，它可以用来改变约定的握手流程，控制会话过程。比如，MQXCC_SEND_SEC_MSG 表示发送安全报文到对方，MQXCC_SEND_AND_REQUEST_SEC_MSG 表示要求对方回送安全报文，即时序上对方安全出口会被自动再调用一次，且 ExitReason=MQXR_SEC_MSG。MQXCC_SUPPRESS_FUNCTION 表示握手失败并关闭通道，通常用于身份验证失败后关闭通道。

- MQXCC_OK 正常，按缺省方式安排以后的流程
- MQXCC_SUPPRESS_FUNCTION 出错，关闭通道
- MQXCC_CLOSE_CHANNEL 同上，关闭通道
- MQXCC_SEND_AND_REQUEST_SEC_MSG 发送 Security 报文，同时要求对方回送 Security 报文
- MQXCC_SEND_SEC_MSG 发送 Security 报文

入口参数 pChannelDefinition (PMQCD) -> ExitResponse2 通常与 ExitResponse 配合使用。如：

- MQXR2_PUT_WITH_DEF_ACTION 仅限 Receiver 通道使用，按缺省方式 PUT，即 PUT 时的用户身份为通道上配置的缺省 MCA 用户或消息上下文中的 MQMD.UserIdentifier
- MQXR2_PUT_WITH_DEF_USERID 仅限 Receiver 通道使用，PUT 时的用户身份为通道上配置的缺省 MCA 用户
- MQXR2_PUT_WITH_MSG_USERID 仅限 Receiver 通道使用，PUT 时的用户身份为消息上下文中的 MQMD.UserIdentifier
- MQXR2_USE_AGENT_BUFFER 使用 AgentBuffer，容量为 AgentBufferLength，有效数据长度为 DataLength
- MQXR2_USE_EXIT_BUFFER 使用 ExitBuffer，容量为 ExitBufferLength，有效数据长度为 DataLength
- MQXR2_CONTINUE_CHAIN 若配置了多个 UserExit，继续下一个 UserExit 调用
- MQXR2_SUPPRESS_CHAIN 若配置了多个 UserExit，跳过后续所有的 UserExit 调用

例：配置脚本

- QM1


```

DEFINE QREMOTE (QR_QM2) +
    RNAME (QL_QM2) +
    RQMNAME (QM2) +
    XMITQ (QLX_QM2)
DEFINE QLOCAL (QLX_QM2) +
    USAGE (XMITQ)
DEFINE CHANNEL (C_QM1.QM2) +
    CHLTYPE (SDR) +
      
```

```

TRPTYPE (TCP) +
CONNNAME ('127.0.0.1 (1416)') +
XMITQ (QLX_QM2) +
SCYEXIT ('C:\SecurityExit.dll (SecurityExit)') +
SCYDATA ('SecurityExit.log')

```

- QM2

```

DEFINE QLOCAL (QL_QM2)
DEFINE CHANNEL (C_QM1.QM2) +
CHLTYPE (RCVR) +
TRPTYPE (TCP) +
SCYEXIT ('C:\SecurityExit.dll (SecurityExit)') +
SCYDATA ('SecurityExit.log')

```

```
start runmqclsr -m QM2 -t tcp -p 1416
```

由于 Security Exit 的会话可以由编程控制，所以通过设置回应码我们可以方便地形成单向认证或双向认证过程。下面将着重介绍这两个过程的时序安排。

12.2.2.1 单向认证

单向认证可以由 Sender 发起，由 Receiver 对 Sender 认证。图 12-2。也可以反过来。流程如图 12-3。

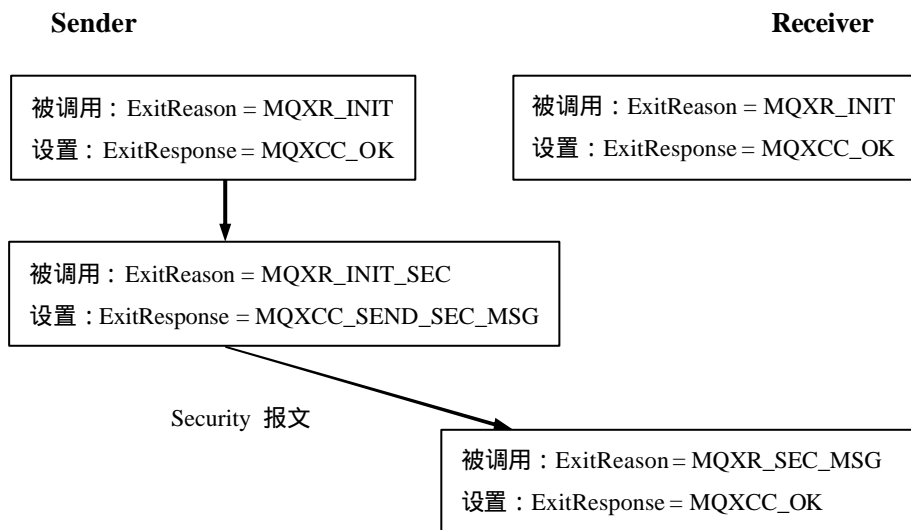


图 12-2 单向认证，由 Sender 发起

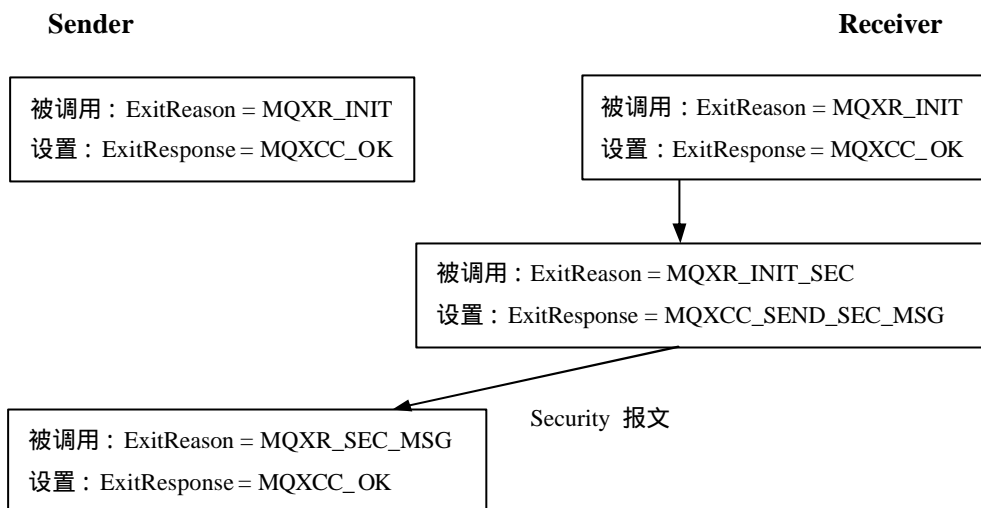


图 12-3 单向认证，由 Receiver 发起

12.2.2.2 双向认证

双向认证可以由 Sender 发起，首先由 Receiver 对 Sender 认证，再由 Sender 对 Receiver 认证。图 12-4。当然，这个次序也可以倒过来。采用何种方式，完全由应用需求而定。图 12-5。

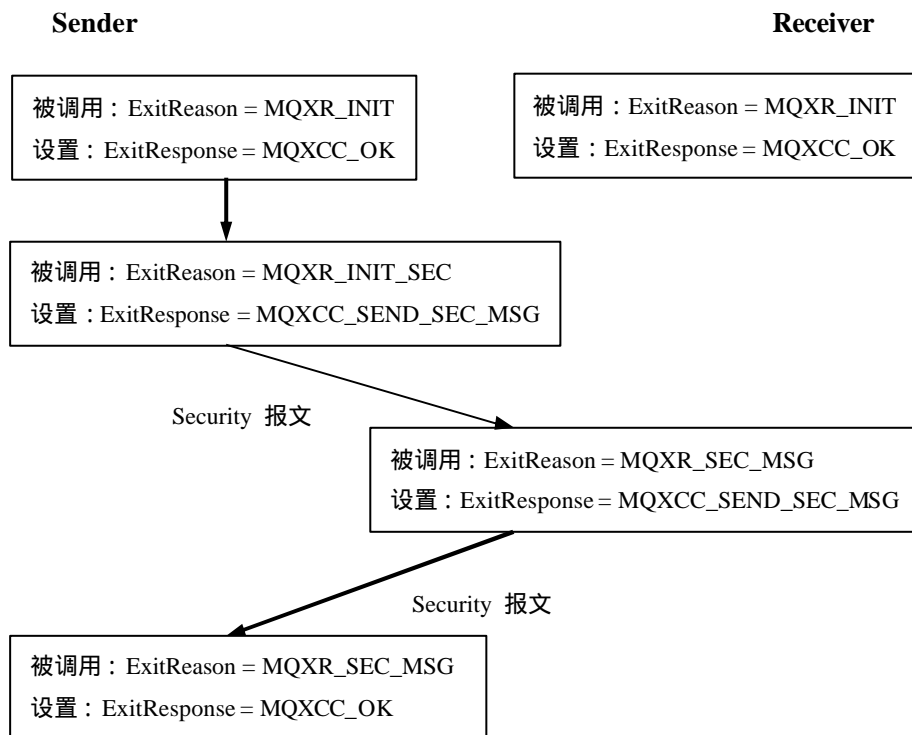


图 12-4 双向认证，由 Sender 发起

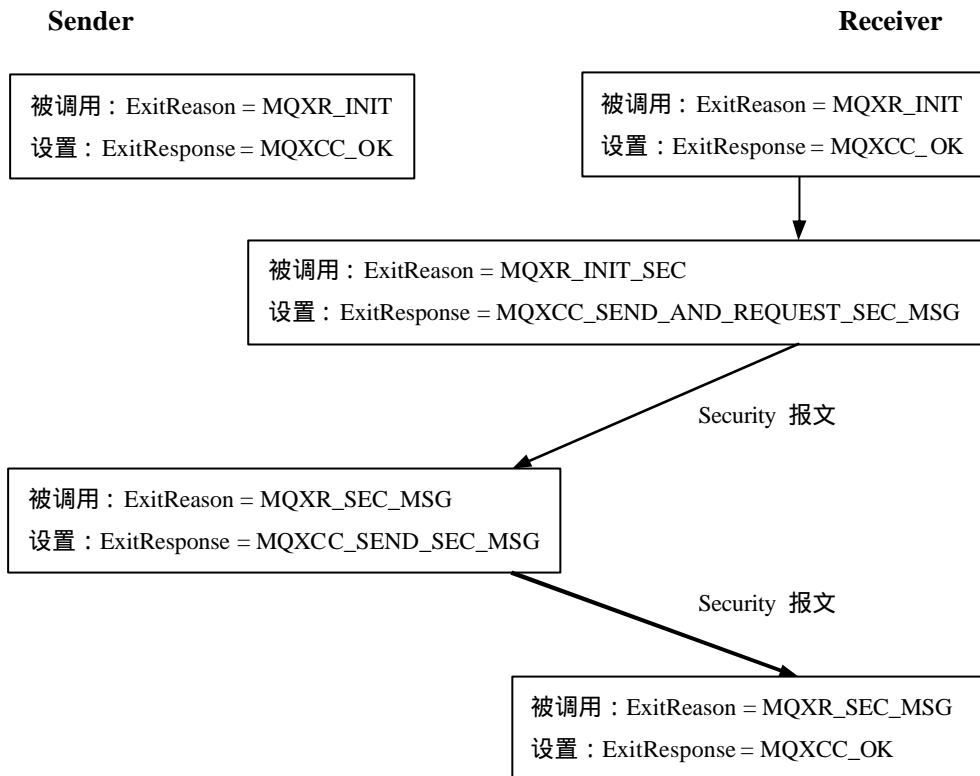


图 12-5 双向认证，由 Receiver 发起

如果 Sender/Receiver 两侧都用 MQXCC_SEND_SEC_MSG，则流程上的次序可以自行安排。换句话说，MQXCC_SEND_SEC_MSG 只是单向地主动发送 Security 报文，对接下来双方的流程安排是不具有约束力和指导意义的。如果发起方用 MQXCC_SEND_AND_REQUEST_SEC_MSG，则约定了接下来的流程安排，即发起方应该首先发送 Security 报文，对方验证后用 MQXCC_SEND_SEC_MSG 回送 Security 报文。在上图中主动方为 Receiver，事实上主动方也可以是 Sender。

12.2.3 Message Exit

Message Exit 在消息被发送时调用，且对于一条消息只会被调用一次，无论这条消息在物理上被拆成多少个段传送。

Message Exit 的入口参数 pChannelExitParms (PMQXP) -> ExitReason 只会取 MQXR_INIT, MQXR_TERM 和 MQXR_MSG 三种值，表示调用的原因码。

- MQXR_INIT 启动通道
- MQXR_TERM 停止通道
- MQXR_MSG 发送消息

Message Exit 的入口参数 pChannelDefinition (PMQCD) -> ChannelType 表示通道类型，通常用于通用程序中区分通道。

例：配置脚本

- QM1

```

DEFINE QREMOTE (QR_QM2) +
    RNAME (QL_QM2) +
    RQMNAME (QM2) +
    XMITQ (QLX_QM2)
DEFINE QLOCAL (QLX_QM2) +
    USAGE (XMITQ)
DEFINE CHANNEL (C_QM1.QM2) +
    CHLTYPE (SDR) +
    TRPTYPE (TCP) +
    CONNAME ('127.0.0.1 (1416)') +
    XMITQ (QLX_QM2) +
    MSGEXIT ('C:\MessageExit.dll (MessageExit)') +
    MSGDATA ('MessageExit.log')

```

- QM2

```

DEFINE QLOCAL (QL_QM2)
DEFINE CHANNEL (C_QM1.QM2) +
    CHLTYPE (RCVR) +
    TRPTYPE (TCP) +
    MSGEXIT ('C:\MessageExit.dll (MessageExit)') +
    MSGDATA ('MessageExit.log')

```

```
start runmqclsr -m QM2 -t tcp -p 1416
```

12.2.4 Send Exit

Send Exit 的入口参数 `pChannelExitParms (PMQCXP) -> ExitReason` 只会取 `MQXR_INIT`, `MQXR_TERM` 和 `MQXR_MSG` 三种值, 表示调用的原因码。

- `MQXR_INIT` 启动通道
- `MQXR_TERM` 停止通道
- `MQXR_MSG` 发送消息

Send Exit 入口参数中的 `AgentBuffer` 是网络上将要传送的数据包, 其结构包括 MQ 内部通信结构 (MQTSH)、消息传输头 (MQXQH) 和消息体三部分, 如图 12-6。其中前两部分是定长的, 共 476 字节, 如果消息体只有一个字节, 则整个数据包长 477 字节, 我们在实际的应用环境中会观察到这个数据包。

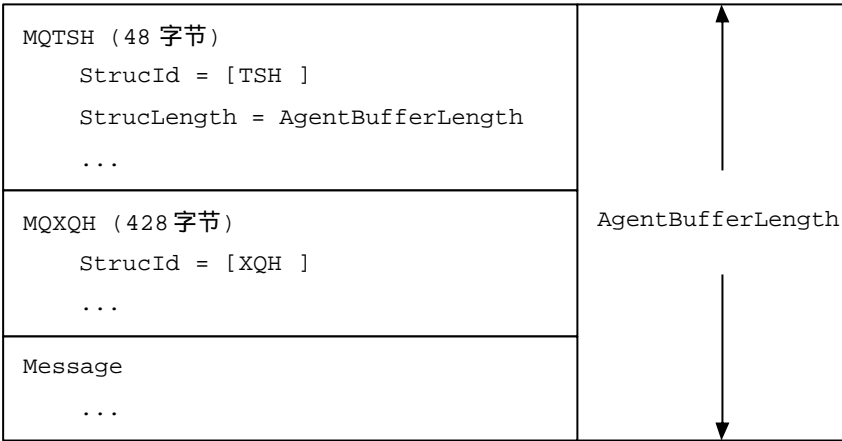


图 12-6 通道出口中的 AgentBuffer 结构

另外，在一段时间内如果没有消息传送，MQ Channel 会定时送出同步信息包（28 字节），在 Send Exit 中可以取到，例：

```
pDataLength          MQLONG: [28]
0000: 54 53 48 20 1C 00 00 00 02 05 08 00 00 00 00 00 TSH .....
0010: 00 00 00 00 22 02 00 00 65 05 00 00           ...."...e...
```

如果单向的 Channel 两端都定义了 Send Exit，在一次单向消息传送时，发送端送出了消息，其上的 Send Exit 很自然会被调用一次。消息到达接收端后，接收端会回送一个同步信息包（28 字节）当作应答信号（Acknowledge Signal），这样接收端的 Send Exit 也会被调用一次。这恰好证明了 MQ 通道的单向性只是针对消息数据而言，对控制信息包而言实际上是双向的，通道本质上是全双工的网络连接。

```
pDataLength          MQLONG: [28]
0000: 54 53 48 20 1C 00 00 00 02 05 00 00 E7 5C 99 3E TSH .....\.>
0010: 10 00 14 01 22 02 00 00 65 05 00 00           ...."...e...
```

下面，让我们来做实验进一步理解 Send Exit 的工作原理。建立 Sender/Receiver 通道后，如果在发送方连续做以下操作：

- 1. RUNMQSC start channel (...)
- 2. amqsput a
- 3. amqsput b
- 4. amqsput c
- 5. RUNMQSC stop channel (...)

在 Send Exit 中可以监控到被多次调用，如表 12-5。

表 12-5 Send Exit 的多次调用

ExitReason	AgentBufferLength	Message
MQXR_INIT	0	
MQXR_XMIT	477	a
MQXR_XMIT	477	b
MQXR_XMIT	477	c
MQXR_XMIT	28	
MQXR_XMIT	28	
MQXR_TERM	0	

将 Send Exit 截获的消息包全部记录下来，写入文件中，其内容摘要如下：

- Start Channel

```
pChannelExitParms    MQCXP:
    ExitReason        - [MQXR_INIT]
    MaxSegmentLength  - [0]
    PartnerName       - [
    ShortConnectionName - [127.0.0.1 (1416) ]
    ConnectionName    - [127.0.0.1 (1416)]
pDataLength          MQLONG: [0]
```

- Stop Channel

```
pChannelExitParms    MQCXP:
    ExitReason        - [MQXR_TERM]
    MaxSegmentLength  - [0]
    PartnerName       - [QM2 ]
    ShortConnectionName - [127.0.0.1 ]
    ConnectionName    - [127.0.0.1]
pDataLength          MQLONG: [0]
```

- MQPUT

```
pChannelExitParms    MQCXP:
    ExitReason        - [MQXR_XMIT]
    MaxSegmentLength  - [32766]
    PartnerName       - [QM2 ]
    ShortConnectionName - [127.0.0.1 ]
    ConnectionName    - [127.0.0.1]
pDataLength          MQLONG: [489]
```

- Interval 每隔一段时间会有一个数据包

```
pChannelExitParms    MQCXP:
    ExitReason        - [MQXR_XMIT]
    MaxSegmentLength  - [32766]
    PartnerName       - [QM2 ]
    ShortConnectionName - [127.0.0.1 ]
    ConnectionName    - [127.0.0.1]
pDataLength          MQLONG: [28]
pAgentBufferLength    MQLONG: [32766]
    0000: 54 53 48 20 1C 00 00 00 02 05 01 00 E7 5C 99 3E TSH .....\.>
    0010: 10 00 09 01 22 02 00 00 65 05 00 00 .....".e...
pAgentBufferLength    MQLONG: [32766]
    0000: 54 53 48 20 1C 00 00 00 02 05 08 00 00 00 00 00 TSH .....
    0010: 00 00 00 00 22 02 00 00 65 05 00 00 .....".e...
```

例：配置脚本

- QM1

```
DEFINE QREMOTE (QR_QM2) +
RNAME (QL_QM2) +
```

```

RQMNAME (QM2) +
XMITQ (QLX_QM2)
DEFINE QLOCAL (QLX_QM2) +
    USAGE (XMITQ)
DEFINE CHANNEL (C_QM1.QM2) +
    CHLTYPE (SDR) +
    TRPTYPE (TCP) +
    CONNNAME ('127.0.0.1 (1416)') +
    XMITQ (QLX_QM2) +
    SENDEXIT ('C:\SendExit.dll (SendExit)') +
    SENDDATA ('SendExit.log')

```

- QM2

```

DEFINE QLOCAL (QL_QM2)
DEFINE CHANNEL (C_QM1.QM2) +
    CHLTYPE (RCVR) +
    TRPTYPE (TCP)

```

```
start runmqclsr -m QM2 -t tcp -p 1416
```

12.2.5 Receive Exit

Receive Exit 与 Send Exit 非常类似，其入口参数 `pChannelExitParms (PMQCXP)` -> `ExitReason` 也只会取 `MQXR_INIT`，`MQXR_TERM` 和 `MQXR_MSG` 三种值，表示调用的原因码。

- `MQXR_INIT` 启动通道
- `MQXR_TERM` 停止通道
- `MQXR_MSG` 接收消息

Receive Exit 接收到的所有信息与对方的 Send Exit 是可以一一对应的，两边记录的 `MQChannelSendExit.log` 和 `MQChannelReceiveExit.log` 应该是完全一样的。如果单向通道两端都定义了 Receive Exit，在一次单向消息传送时，接收端 Receive Exit 被调用一次，接着接收端会回送一个同步信息包 (28 字节) 当作应答信息，导致发送端的 Receive Exit 也被调用一次。这也同样证明了 MQ 通道的单向性只是针对消息数据而言，对控制信息包而言实际上是双向的，通道其实是全双工的网络连接。

例：配置脚本

- QM1

```

DEFINE QREMOTE (QR_QM2) +
    RNAME (QL_QM2) +
    RQMNAME (QM2) +
    XMITQ (QLX_QM2)
DEFINE QLOCAL (QLX_QM2) +
    USAGE (XMITQ)
DEFINE CHANNEL (C_QM1.QM2) +

```

```

CHLTYPE (SDR) +
TRPTYPE (TCP) +
CONNNAME ('127.0.0.1 (1416)') +
XMITQ (QLX_QM2)

```

● QM2

```

DEFINE QLOCAL (QL_QM2)
DEFINE CHANNEL (C_QM1.QM2) +
CHLTYPE (RCVR) +
TRPTYPE (TCP) +
RCVEXIT ('C:\ReceiveExit.dll (ReceiveExit)') +
RCVDATA ('ReceiveExit.log')

```

```
start runmqclsr -m QM2 -t tcp -p 1416
```

12.2.6 Message Retry Exit

通道接收端 MCA 完整地收到消息后就要将其放入到目标队列中，有些会因为某些暂的原因无法放入。所以，在对目标队列访问（大多数是 MQOPEN 和 MQPUT）不成功之后，WebSphere MQ 会调用 Message Retry Exit，以决定是否等待一段间隔时间后重试。

通道中与有 Message Retry Exit 相关的属性如表 12-6。

表 12-6 通道中与 Message Retry Exit 的相关属性

描述	属性	说明
Message-Retry Exit Name	MREXIT	出口名
Message-Retry Exit User Data	MRDATA	出口用户参数
Message-Retry Count	MRRTY	重试次数，0 到 999,999,999，0 表示不重试
Message-Retry Interval	MRTMR	毫秒，0 到 999,999,999，0 表示无间隔时间

注意：这些属性只适用于 Receiver、Requester 和 Cluster Receiver 通道，且在 z/OS 上不支持。

在缺省情况下，通道是不设置 Message-Retry Exit 的，即 MREXIT 和 MRDATA 属性为空。这时，消息放入重试是由 MRRTY 和 MRTMR 两个参数控制，分别表示重试的次数和间隔时间。

- 如果是因为暂时的故障发送失败（比如 MQRC_Q_FULL），MCA 会自动重试
 - 重试次数为 Message-Retry Count
 - 重试间隔为 Message-Retry Interval
- 如果是因为比较永久的原因造成发送失败，则消息会直接写入死信队列。
- 如果重试次数到了，但仍未成功，则消息被写入死信队列，如果队列管理器未设置死信队列，则通道停止。

如果在通道中设置了 Message-Retry Exit，则通道的 Message-Retry Count 和 Message-Retry Interval 属性不再起作用，它们会被映射到 Message-Retry Exit 的入口参数的结构域中，在出口程序中可以将这些域值改变，从而以新的值影响 MQ 的重试工作。因为设置了 Message-Retry Exit，重试的间隔和次数完全被出口程序控制，可以在出口程序中设置等待的时间长短。如果出口程序返回 ExitResponse = MQXCC_OK 则 MCA 会继续重试，

直到 ExitResponse = MQXCC_SUPPRESS_FUNCTION 结束。

ExitResponse 的取值可以有三种，MQXCC_OK、MQXCC_SUPPRESS_FUNCTION 和 MQXCC_SUPPRESS_EXIT ,分别表示隔一段时间继续重试、对该消息不再重试和出口失效。对于 MQXCC_SUPPRESS_EXIT 表示对于这条消息不再重试，对于以后的消息，出口程序也不再起作用。按通道的属性 Message-Retry Count，Message-Retry Interval 来决定重试的次数和间隔。

在调用 Message-Retry Exit 时，入口参数 (PMQCXP) pChannelExitParms 中有三个与 Message Retry 相关的分量，如表 12-7。

表 12-7 通道中与 Message Retry Exit 的相关属性

相关分量	说明	输入/输出
PMQCXP -> MsgRetryCount	从 0 开始，自动加 1 计数	输入输出域
PMQCXP -> MsgRetryInterval	来自 Message-Retry Interval	输入输出域
PMQCXP -> MsgRetryReason	MQRC_*, 上一次重试的返回值	输入域

在第一次调用这个出口程序的时候：

- PMQCXP -> MsgRetryCount = 1 ,表示第一次准备重试 如果当次 ExitResponse = MQXCC_OK, 表示决定等待一段间隔时间后重试，重试不成功后会再次调用 Exit 程序。 以后的调用每次自动加 1。
- PMQCXP -> MsgRetryInterval 为通道的 Message-Retry Interval 属性，如果在出口程序中调整 (0 到 999,999,999), 则 MCA 会等待新的时间间隔，才进行下一次的重试。 如果值超出有效范围，即小于 0 或大于 999,999,999，则设置无效，MCA 会重新使用通道的 Message-Retry Interval 属性。

以后每次调用这个出口程序的时候：

- PMQCXP -> MsgRetryCount 上一次的值加 1
- PMQCXP -> MsgRetryInterval 上一次的值，可以继续修改

例：配置脚本

● QM1

```
DEFINE QREMOTE (QR_QM2) +
  RNAME (QL_QM2) +
  RQMNAME (QM2) +
  XMITQ (QLX_QM2)
DEFINE QLOCAL (QLX_QM2) +
  USAGE (XMITQ)
DEFINE CHANNEL (C_QM1.QM2) +
  CHLTYPE (SDR) +
  TRPTYPE (TCP) +
  CONNAME ('127.0.0.1 (1416)') +
  XMITQ (QLX_QM2)
```

● QM2

```
DEFINE QLOCAL (QL_QM2)
DEFINE CHANNEL (C_QM1.QM2) +
```

```

CHLTYPE (RCVR) +
TRPTYPE (TCP) +
MREXIT ('C:\RetryExit.dll (RetryExit)') +
MRDATA ('RetryExit.log') +
MRRTY (10) +
MRTMR (1234)

```

```
start runmqqlsr -m QM2 -t tcp -p 1416
```

12.2.7 Channel Auto-Definition Exit

在 Receiver、Server 或 Cluster Sender 通道收到对方要求启动己方并没有定义的通道请求时，如果队列管理器的通道自动定义开关打开着，即 CHAD (ENABLED)，则队列管理器会自动定义通道并启动连接。然而，这一过程也可以由调用通道自动定义出口 (Channel Auto-Definition Exit) 来完成。具体过程如下：

在发送方 MQSC start channel 的时候，如果接收方发现没有相应通道定义，且 CHAD (ENABLE)，则试图自动建立通道。

- 如果 CHADEXIT 为空。
没有设定出口程序，根据 SYSTEM.AUTO.RECEIVER 或 SYSTEM.AUTO.SVRCON 的缺省定义自动建立通道。
- 如果 CHADEXIT 不为空。
设定了出口程序，队列管理器调用出口程序。且入口参数为 SYSTEM.AUTO.RECEIVER 或 SYSTEM.AUTO.SVRCON 的缺省定义。在出口程序中可以相应改变，建立符合实际要求的通道。

Channel Auto-Definition Exit 的函数入口为 MQ_CHANNEL_AUTO_DEF_EXIT (ChannelExitParms, ChannelDefinition)。在建立通道的过程中，出口程序会被连续调用三次，ExitReason 码分别为 MQXR_INIT、MQXR_AUTO_RECEIVER 和 MQXR_TERM。通道一旦自动建立，就能持久性地保留下来，以后发送方再次启动该通道就不再导致接收方调用出口程序。也就是说，自动建立的通道是永久性的，一旦建立后，出口程序不会再被调用。

例：配置脚本

- QM1

```

DEFINE QREMOTE (QR_QM2) +
  RNAME (QL_QM2) +
  RQMNAME (QM2) +
  XMITQ (QLX_QM2)
DEFINE QLOCAL (QLX_QM2) +
  USAGE (XMITQ)
DEFINE CHANNEL (C_QM1.QM2) +
  CHLTYPE (SDR) +
  TRPTYPE (TCP) +
  CONNAME ('127.0.0.1 (1416)') +

```

```

XMITQ      (QLX_QM2)
● QM2
ALTER      QMGR      +
CHAD       (ENABLED) +
CHADEXIT   (C:\AutoDefExit.dll (AutoDefExit)')
DEFINE     QLOCAL    (QL_QM2)
DEFINE     CHANNEL   (C_QM1.QM2) +
CHLTYPE    (RCVR)    +
TRPTYPE    (TCP)

```

```
start runmqclsr -m QM2 -t tcp -p 1416
```

12.2.7 Transport-Retry Exit

Transport-Retry Exit 出口程序使你可以将消息在出口程序中按住不发，等待一段时间后，在合适的时机再发。这对于接收方是移动设备的情况下特别有用，比如，接收方是移动设备正进入隧道，不在通信服务区内。

Transport-Retry Exit 只支持在 WebSphere MQ 5.3 for AIX 上基于 UDP 通信协议的通道，即通道的 TRPTYPE 属性为 UDP。UDP 消息必须在 4MB 以内。Transport-Retry Exit 通常在消息发送之前被调用，由 ExitReason 指明调用的原因码。

- MQXR_INIT 启动通道
- MQXR_TERM 停止通道
- MQXR_RETRY 每次 UDP Datagram 消息发送之前
- MQXR_END_BATCH 当一个消息批次结束时

- MQXR_ACK_RECEIVED 当一个消息应答的回执送达时

Transport-Retry Exit 的应答码只有三种：MQXCC_OK、MQXCC_CLOSE_CHANNEL 和 MQXCC_REQUEST_ACK，分别表示继续执行、关闭通道和要求回执。我们可以在每次发送 Datagram 消息时（原因码为 MQXR_RETRY）检查线路，比如对方的 IP 地址在网络中是否存在，如果线路有问题则等一段时间，等到没有问题时设置应答码 MQXCC_OK，立即发送。如果对传送不放心，则设置应答码 MQXCC_REQUEST_ACK。这时 WebSphere MQ 会要求通道对方通过内部通信自动回送应答信号包，在应答收到后，自动再次调用出口，这一次原因码为 MQXR_ACK_RECEIVED。

在 mq.ini 配置文件中的 TRANSPORT 段 (stanza) 中可以指定出口程序名和入口函数名，在 AIX 上格式为 *ExitProg (ExitEntrance)*。其中，*ExitProg* 为出口程序名，*ExitEntrance* 为入口函数名。

例：

UDP:

```
ACKREQ_TIMEOUT = 5
ACKREQ_RETRY = 60
CONNECT_TIMEOUT = 5
CONNECT_RETRY = 60
ACCEPT_TIMEOUT = 5
ACCEPT_RETRY = 60
DROP_PACKETS = 0
BUNCH_SIZE = 8
PACKET_SIZE = 2048
PSEUDO_ACK = YES
```

TRANSPORT:

```
RETRY_EXIT = exitname
```

12.3 Data Conversion Exit

WebSphere MQ 的数据有两个地方可以设置使用，即发送端的通道属性设置 CONVERT (YES)，或者接收端在 MQGET 时设置 MQGMO_CONVERT 选项。一般说来，推荐使用在接收端进行数据转换的方式，对于接收端 MQ 不支持数据转换功能的，才考虑在发送端进行数据转换。无论哪一种方式，WebSphere MQ 会先试图用内置的规则来转换，如果不成功则试图调用 Data-Conversion Exit 用户出口。

Data-Conversion Exit 在满足以下全部条件的时候会被调用

- 带 MQGMO_CONVERT 选项的 MQGET，或者 Sender MCA 的通道属性设置 CONVERT (YES)
- 队列中消息的 MQMD.CodedCharSetId 或者 MQMD.Encoding 与 MQGET 参数中的 MQMD.CodedCharSetId 或者 MQMD.Encoding 不一致。
- 队列中消息的 MQMD.Format 不能是 MQFMT_NONE
- MQGET 参数中的 BufferLength 不能是 0
- 队列中消息的长度不能是 0

- 消息有用户自定义的格式。如果消息有多个部分（多个消息头），每个部分有各自的格式，比如死信消息一定有 MQFMT_DEAD_LETTER_HEADER。 WebSphere MQ 会先为各部分完成内置格式的转换，最后调用用户出口（用户数据只能在最后）。

消息在 MQPUT 的时候，消息头上有三个域与消息格式有关，如表 12-8。在 MQGET 的时候，MQMD 也有这三个域。在消息的 MQMD 与 MQGET 的 MQMD 中的 CodedCharSetId 或 Encoding 不一致的时候，MQ 会试图做格式转换。它取消息的 MQMD.Format 作为格式名，格式名至多 8 个字符，比如“MYFORMAT”，用户自定义格式不可以 MQ 打头。在出口目录中找同名的动态加载模块，调用这个模块的入口函数。比如 MQMD.Format = "MYFORMAT"，相应的 Data-Conversion Exit 用户出口为 MYFORMAT.dll，WebSphere MQ 会在做格式转换的时候调用 MYFORMAT.dll 的入口函数。在 Windows 中出口目录由 <MQ_HKEY>\Configuration\ClientExitPath\ExitsDefaultPath 指定。

表 12-8 消息头中与格式相关的域

消息域	缺省值	解释
MQMD.CodedCharSetId	1381	字符集代码
MQMD.Encoding	MQENC_NATIVE	数字类型的高低字节安排方式
MQMD.Format	无	数据的格式

Data-Conversion Exit 入口参数中 (PMQDXP) pDataConvExitParms 指的是 MQGET 中的值，(PMQMD) pMsgDesc 指的是消息头中的值，两个参数中都有 Encoding 和 CodedCharSetId，在处理时要特别注意。另外，在 (PMQDXP) pDataConvExitParms 有 DataLength, CompCode, Reason 和 ExitResponse 四个域是可以被出口程序改动的（解释如下），对其它域的改动则是无效的。如果需转换的消息是一个消息分段，只是完整消息的一部分，那么 DataLength 不能改变，这是一个例外。

- DataLength 指消息体的长度，返回时要设置转换后消息的长度
- CompCode 返回时要根据转换的结果设置为 MQCC_OK、MQCC_WARNING 或 MQCC_FAILED
- Reason 返回时根据情况设置为 MQRC_NONE、MQRC_TRUNCATED_MSG_ACCEPTED 或者 MQRC_NOT_CONVERTED
- ExitResponse 返回时可以设置为 MQXDR_OK 或者 MQXDR_CONVERSION_FAILED

在使用 Data-Conversion Exit 时，一般来说，先定义一个结构，用 crtmqcvx 生成一段对应的转换代码，嵌入出口程序中调用即可。结构中只支持 MQBYTE, MQCHAR, MQLONG, 不支持 float, double, 指针, bit 型变量，生成的转换代码是含域间距的 (Pad Space)，即结构域是字对齐的。关于这一点，可以在生成的转换代码中体现。我们可以在 MQPUT 或者 MQGET 的程序中对结构加上 #pragma pack，或者修改 amqsvmha.h 中的 AlignShort() 和 AlignLong()，目前它们是空函数。如果要支持特殊域的转换（如 float, double, 或自定义的类型）可以在结构定义中用 MQBYTE 留出空间，且修改生成的转换代码，自己进行类型转换。具体编程请参见书后例程。

12.4 Cluster Workload Exit

在 WebSphere MQ 的群集环境中，可以配置多个同名的本地队列用于负载平衡。例群集中有三个队列管理器 QM1、QM2 和 QM3，其中 QM1 为 Repository Queue Manager。QM2 和 QM3 中有一个同名的本地队列 QL，结构如图 12-7。群集调度算法缺省会在两个 QL 之间轮循，应用程序接入 QM1 后对 QL 的 MQPUT 操作缺省会轮流地分布到两个队列管理器上面。当然，例程 MQClusterPut 可以使用 MQOPEN (MQOO_BIND_ON_OPEN) 来绕过群集调度算法。

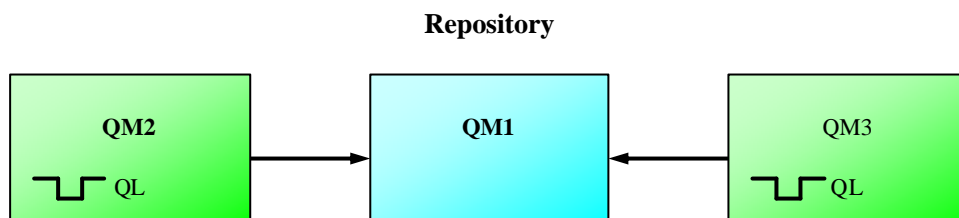


图 12-7 群集环境中的负载平衡

应用程序连接到 QM1 上，通过 QM1 将消息放入到群集队列 QL 中。这里实际上隐含了一个选择路由的过程，消息每次从 QM1 送出时可以有二个选择：QM2 或 QM3。在没有绑定路由的情况下，群集的选择策略是采用轮流法，我们可以通过编写自己的 Cluster Workload Exit 来实现自己的选择算法，比如加权法。如果路由已经绑定，则群集的选择算法不再起作用。下面先让我们通过一个例子来了解什么是绑定。

一般来说，如果 MQOPEN 与队列没有绑定关系，则以后连续的 MQPUT 会轮流分布到 Cluster 的同名队列中。例：

```
amqspout QM1 QL
```

1

2

QM2 上的 QL 含消息 1

QM3 上的 QL 含消息 2

缺省情况下 MQOPEN 本身也会在两个队列管理器之间轮循，它决定了紧接着的 MQ 操作首先针对哪一个队列管理器，MQPUT 会在这个结果的基础上继续轮循。例：

```
amqspout QM1 QL
```

1

2

3

QM2 上有 1, 3，QM3 上有 2

接着

```
amqspout QM1 QL
```

4

5

6

QM2 上有 1,3, 5，QM3 上有 2, 4, 6

应用程序与队列之间是否具有绑定关系是由 MQOPEN 选项与队列属性 (DEFBIND) 共同决定的 (表 12-9)。基本上, MQOPEN 选项可以覆盖队列属性设置。

表 12-9 MQOPEN 选项与队列缺省绑定属性之间的关系

	DEFBIND = OPEN	DEFBIND = NOTFIXED
MQOO_BIND_ON_OPEN	绑定	绑定
MQOO_BIND_NOT_FIXED	不绑定	不绑定
MQOO_BIND_AS_Q_DEF	绑定	不绑定

一旦应用程序与某个本地队列形成绑定关系, Cluster 调度算法就不再被调用。反之, 如果应用程序与 QL 没有绑定关系, Cluster 调度算法就起作用了。例程 MQClusterWorkloadExit.c 可以被编译成动态库 (在 Windows 上为 dll), 在队列管理器启动时被调入, 每次 MQPUT 缺省的算法会在可用的队列管理器之间轮循, 形成入口参数调用 MQClusterWorkloadExit, 此时可以利用代码对缺省算法的结果进行修改。可以通过 QMGR 的属性 CLWLDATA 把 MQPUT 的消息都转入指定的队列管理器中。

MQClusterWorkloadExit 会生成 MQClusterWorkloadExit.log, 记录被调用的入口参数及可选择的通道定义。通过分析 MQClusterWorkloadExit.log 可知, MQClusterWorkloadExit 在队列管理器启动时被调入并调用一次 ExitReason = [MQXR_INIT], 接着每一次 MQPUT 被调用一次 ExitReason = [MQXR_CLWL_PUT], 在队列管理器停止时再被调用一次 ExitReason = [MQXR_TERM]。

例：配置脚本

- QM1

```
ALTER QMGR +
  REPOS (SHINE) +
  CLWLEXIT ('C:\CLWLExit.dll (CLWLExit)') +
  CLWLDATA ('C_QM3')
DEFINE CHANNEL (C_QM1) +
  CHLTYPE (CLUSRCVR) +
  TRPTYPE (TCP) +
  CONNAME ('127.0.0.1 (1414)') +
  CLUSTER (SHINE)
```

- QM2

```
DEFINE QLOCAL (QL) +
  CLUSTER (SHINE) +
  DEFBIND (NOTFIXED)
DEFINE CHANNEL (C_QM2) +
  CHLTYPE (CLUSRCVR) +
  TRPTYPE (TCP) +
  CONNAME ('127.0.0.1 (1415)') +
  CLUSTER (SHINE)
DEFINE CHANNEL (C_QM1) +
  CHLTYPE (CLUSSDR) +
  TRPTYPE (TCP) +
```

```

CONNAME ('127.0.0.1 (1414)') +
CLUSTER (SHINE)
● QM3
DEFINE QLOCAL (QL) +
CLUSTER (SHINE) +
DEFBIND (NOTFIXED)
DEFINE CHANNEL (C_QM3) +
CHLTYPE (CLUSRCVR) +
TRPTYPE (TCP) +
CONNAME ('127.0.0.1 (1416)') +
CLUSTER (SHINE)
DEFINE CHANNEL (C_QM1) +
CHLTYPE (CLUSSDR) +
TRPTYPE (TCP) +
CONNAME ('127.0.0.1 (1414)') +
CLUSTER (SHINE)

```

```

start runmqclsr -m QM1 -t tcp -p 1414
start runmqclsr -m QM2 -t tcp -p 1415
start runmqclsr -m QM3 -t tcp -p 1416

```

在 Queue Manager 配置的 ExitProperties stanza (UNIX) 或是 Exits (Windows) 中可以指定 CLWLMode = SAFE 或 FAST

- CLWLMode = SAFE
缺省设置。Cluster Workload Exit 运行于队列管理器之外的一个独立进程中，比较安全。如果 CLWL 进程 (amqzlw0) 出错，Queue Manager 会重启这个进程，记录出错日志。
- CLWLMode = FAST
Cluster Workload Exit 嵌入 Queue Manager，运行于同一个进程中。只有在确保 Cluster Workload Exit 程序健壮无误的情况下才考虑用 FAST 方式。

12.5 Pub/Sub Routing Exit

在 WebSphere MQ 发布/订阅环境中，当系统决定了对指定 Broker 或 Subscriber 发送订阅消息之后会试图调用发布/订阅用户出口。出口函数可以在这时改变这一决定，比如路由到另一个流，或者使用另一个通道。如果在出口函数中改变 pMQPXP -> DestinationQName 或者是 pMQPXP -> DestinationQMgrName，则会造成订阅消息的重路由。建议不要修改 Broker 之间订阅消息的 MQMD。

配置出口的时候，Broker 必须处于停止状态。

- Windows 平台
<MQ_HKEY>\Configuration\QueueManager\<QM>\Broker 中添加字符串：
RoutingExitPath REG_SZ C:\MQPubSubExit.dll (MQPubSubExit)
其中 C:\MQPubSubExit.dll 是目标文件名，MQPubSubExit 是函数入口名。配置字

串格式为：<路径>/<模块>(<函数>)

- UNIX 平台
类似地，配置 qm.ini 文件针对特定 Queue Manager 的 Broker stanza 段，指定 RoutingExitPath 参数

对 Pub/Sub 出口编程的一些限制：

- 不要使用 MQDISC
- 不要使用 MQCMIT 或者 MQBACK

12.6 MQ API Exit

MQ API Exit 有两类，一类是 MQ API 调用，分别在调用前和调用后引发。另一类是用来管理 MQ API Exit，可以注册或注销 MQ API Exit。

- MQI
 - Before MQI Call Exit
 - After MQI Call Exit
- INIT & TERM
 - Initialization Exit
 - Termination Exit

对于每一个 MQI (比如 MQOPEN、MQPUT、MQGET...) 会有两个相应的 API Exit，一个发生在调用前，一个发生在调用后。对于带 MQGMO_CONVERT 选项的 MQGET 另有一个 API Exit，发生在消息从队列中取出后但在数据转换之前，允许在数据转换之间进行数据加解密或加解压。对于 MQCONN 和 MQCONNX 共用一对 API Exit。

API Exit 可以改变 MQI 调用的参数和结果。比如，MQPUT 之前的 API Exit 可以修改数据的长度和内容，也可以修改 MQMD 甚至 MQPMO，从而随心所欲地控制 MQPUT 的动作。MQI 调用之前的 API Exit 也可以干脆取消 MQI 调用，比如，带 MQGMO_CONVERT 选项的 MQGET 可以干脆不做数据转换。

Initialization Exit 发生在应用程序连接队列管理器的时候，它一般用来登记 Exit 函数，或者检查环境变量，或者申请内存。当应用程序断开与队列管理器连接的时候，登记会被自动删除。当然，不必登记所有 MQI 的 Exit 函数。Termination Exit 正相反，它发生在应用程序断开与队列管理器连接的时候。它一般用来释放内存，清理空间。

API Exit 中本身又可以调用 MQI，WebSphere MQ 保证它们不会引起递归调用。由于 MQI 都需要与队列管理器的连接，所以下列情况的 API Exit 不能调用 MQI。

- Initialization Exit
- Termination Exit
- MQCONN 和 MQCONNX 之前的 API Exit
- MQDISC 之后的 API Exit

MQ Client 也可以用 API Exit，但要注意，Exit 的调用是在 MQ Server 一侧。在 UNIX 中，配置 qms.ini 或者 qm.ini，在 Windows 中配置 MQ Services snap-in。其中需要提供 API Exit 名，API Exit 的可执行程序名和入口函数名，参数，如果有多个 API Exit

应该指定它们的调用次序。 无论哪一种都需要提供 API Exit 的名称，函数，模块，数据，序号信息。

12.6.1 设置

UNIX 中的 WebSphere MQ 有三处可以指定 API Exit。 第一处是 mqs.ini 中的 ApiExitCommon 段,它对整个 MQ 运行环境有效。第二处是 mqs.ini 中的 ApiExitTemplate 它也对整个 MQ 运行环境有效。 第三处是 mq.ini 中的 ApiExitLocal,它对该队列管理器有效。 在队列管理器创建的时候，MQ 会将 mqs.ini 中的 ApiExitTemplate 复制到 mq.ini 中的 ApiExitLocal 上。 所以，你可以在创建 Queue Manager 前修改 ApiExitTemplate 模板，也可以在创建后修改 ApiExitLocal 设置。 在队列管理器启动的时候，WebSphere MQ 会先用 ApiExitCommon 设置，再用 ApiExitLocal 设置，如果发生重复，后者覆盖前者：

例，在 mq.ini 中设置：

```
ApiExitLocal:
Sequence=100
Function=MQInitExit
Module=/var/mqm/MQ_Samples/UserExit/MQAPIExit
Name=SampleApiExit
Data=abc
```

类似地，在 NT 中配置 MQ Services snap-in 也有三处可以指定 API Exit，它也遵循同样的覆盖原则。 在“ WebSphere MQ 服务 ”中点击右键选择属性，再选择 API 出口，可以配置公共设置和模板设置。 在相应的队列管理器上点击右键选择属性，再选择出口，可以配置具体的 API 出口。 实际上，它们分别写了 Windows Registry，如下：

- <MQ_HKEY>\Configuration\ApiExitCommon\<name>
- <MQ_HKEY>\Configuration\ApiExitTemplate\<name>
- <MQ_HKEY>\Configuration\QueueManager\<QM>\ApiExitLocal\<name>

其中，<QM> 是队列管理器的名字， <name> 是 API Exit 的名称，如果有同名则遵照覆盖原则。在注册表中的设置如表 12-10。

表 12-10 MQ API Exit 在注册表中的设置

名称	类型	数据	举例	说明
Data	REG_SZ	<Data>	ABC	函数参数
Function	REG_SZ	<Function>	MQInitExit	函数名
Module	REG_SZ	<Module>	C:\MQAPIExit.dll	对应文件
Name	REG_SZ	<Name>	MQAPIExitCommon	User Exit 名
Sequence	REG_SZ	<Sequence>	200	MQ 缺省设置为 100 ApiExitLocal 200 ApiExitCommon 300 ApiExitTemplate

MQ 会按照 Sequence 从大到小的次序调用出口程序。 比如，你同时定义了 ApiExitLocal (100) 和 ApiExitCommon (200)，MQ 会先调用 ApiExitCommon 再调用 ApiExitLocal。 Sequence 值相同的出口程序，不计调用次序。 比如，你同时定义了两个 ApiExitLocal (100)，它们之间的调用次序 MQ 没有规定。

MQ API Exit 可以用来做访问控制，记录使用 MQ API 的情况，跟踪哪些应用程序在什么时间调用了哪些 MQ API。也可以在 MQ API 访问消息之后但应用访问到消息内容之前，检查数据内容的合法性。还可以用来做点对点加密，防止消息的篡改和侦听。如果用上消息 Report，则可以实现双方的身份确认。总之，出自安全灵活，便于监控等多方面的考虑，可以使用 MQ API Exit。

12.6.2 举例

读者如果有兴趣可以使用 MQ API Exit 来跟踪 WebSphere MQ 程序，比如我们打开所有的 MQ API Exit 来跟踪 amqspout 的运行，表 12-11 列出该应用程序调用的 API 和相关参数，这与例程的源代码是一致的。

表 12-11 用 MQ API Exit 跟踪 amqspout

Exit Entry	Object	Parameter
MQInitExit		
MQConnxExit	QM	
MQOpenExit	Q1	MQOO_OUTPUT + MQOO_FAIL_IF_QUIESCING
MQPutExit	Q1	hello
MQCloseExit	Q1	
MQDiscExit	QM	
MQTermExit	QM	

12.6.3 编程设计

程序在第一次加载时需要一个初始化函数 (MQ_INIT_EXIT) 将 API Exit 函数注册到 WebSphere MQ 中，在这个函数中也可以打开一些文件，保持文件句柄，初始化全局变量等等。相应地，有一个结束函数 (MQ_TERM_EXIT)，可以在这个函数中关闭文件句柄，释放内存等等，MQ_TERM_EXIT 本身也是由 MQ_INIT_EXIT 注册的。

```
typedef void MQENTRY MQ_INIT_EXIT (
    PMQAXP      pExitParms,
    PMQAXC      pExitContext,
    PMQLONG      pCompCode,          输出参数，完成码
    PMQLONG      pReason);          输出参数，原因码

typedef void MQENTRY MQ_TERM_EXIT (
    PMQAXP      pExitParms,
    PMQAXC      pExitContext,
    PMQLONG      pCompCode,          输出参数，完成码
    PMQLONG      pReason);          输出参数，原因码
```

注册函数用来将指定的 MQ API 注册到 WebSphere MQ 中，其格式为：

```
void MQENTRY MQXEP (
    MQHCONFIG    Hconfig,          配置句柄，通常使用 MQ_INIT_EXIT 函数参数中
                                   pExitParms -> Hconfig
    MQLONG       ExitReason,       可以用 MQXR_BEFORE 或 MQXR_AFTER 来表示 UserExit 在
```

		API 执行之前或之后被调用
MQLONG	Function,	用来指定 API , 可以是 MQXF_CONN、MQXF_CONNX、MQXF_DISC、MQXF_OPEN、MQXF_CLOSE、MQXF_GET、MQXF_DATA_CONV_ON_GET、MQXF_PUT、MQXF_PUT1、MQXF_INQ、MQXF_SET、MQXF_BEGIN、MQXF_CMIT、MQXF_BACK MQXF_TERM
PMQFUNC	EntryPoint,	函数指针, 指向具体的 API Exit 函数
MQPTR	Reserved,	保留
PMQLONG	pCompCode,	输出参数, 完成码
PMQLONG	pReason);	输出参数, 原因码

API Exit 函数的参数可能各不相同, 具体可以参见 cmqxc.h 中函数说明, 但大致上都有以下四个参数。

```
MQ_XXX_EXIT MQXXXExit;
void MQENTRY MQXXXExit (
    PMQAXP      pExitParms,
    PMQAXC      pExitContext,
    PMQLONG     pCompCode,
    PMQLONG     pReason)
其中, XXX 表示 MQI, 比如函数名为 MQ_PUT_EXIT。
```

第 13 章 MQI 编程

WebSphere MQ 的编程模式中最简单最标准的就是 MQI 编程。所谓, MQI 就是由 WebSphere MQ 提供的一套标准的 API 函数作为编程界面, 编程语言可以是 C/C++、VB、COBOL 等等。MQI 在这些编程语言中的函数名、参数、功能都是完全一样的。

13.1 编程入门

13.1.1 数据类型

为了屏蔽平台间的数据表示上的差异, MQI 编程模式中对数据的类型进行了重新定义。比如用 MQLONG 表示整型, 用 MQCHAR 或 MQBYTE 表示字符和字节, 从而避免了不同操作系统对字长约定的差异。在 MQI 编程模式中, 基本数据类型就只有 MQLONG、MQCHAR、MQBYTE 三种。

13.1.2 数据结构

MQI 编程模式中的数据结构通常由多个域组成, 每个域都是基本数据类型或嵌套的下一级数据结构。以 C 语言为例, 参考 cmqc.h 头文件。几乎所有的数据结构的开头都有以

下域：

```
MQCHAR4      StrucId;    // 结构标识
MQLONG       Version;    // 版本号
```

其中，四个字节的 StrucId 域为结构标识，任何时候应用程序只要分析这个域就可以知道整个结构的构造及长度。MQLONG 型的 Version 域为版本号，WebSphere MQ 历经多年的发展，其功能在不断地扩充，结构定义本身也可在扩大。有些域在低版本中为保留域，在高版本中被赋予了新的功能。所以，通过版本号的设定可以使后继的某些域生效，所以要用到某些高级功能必须将 Version 域设定为较高的版本号。通常高版本号生效的域完全覆盖低版本号生效的域，而数据定义头文件中一般有 *_CURRENT_VERSION 标识当前环境中的最高版本号。

数据结构根据需要可以拼接，也可以级联。拼接的数据结构在内存中是连接的，上级数据结构中往往有某 Offset 域，它指向后继的下级数据结构，Offset 域中存放的是下级数据结构相对上级数据结构起始点的偏移量。级联的数据结构可以处于相互独立的内存中，上级数据结构中往往有某 Ptr 域，它是一个指针，指向下级数据结构的起始地址。MQI 编程模式本身并不限制使用哪一种方式，事实上，大多数数据结构中通常会同时提供这两种域，程序员可以根据自己的喜好任选一种。但是由于编程习惯的原因，Offset 方式常见于 COBOL 代码，Ptr 方式常见于 C 代码。图 13-1。

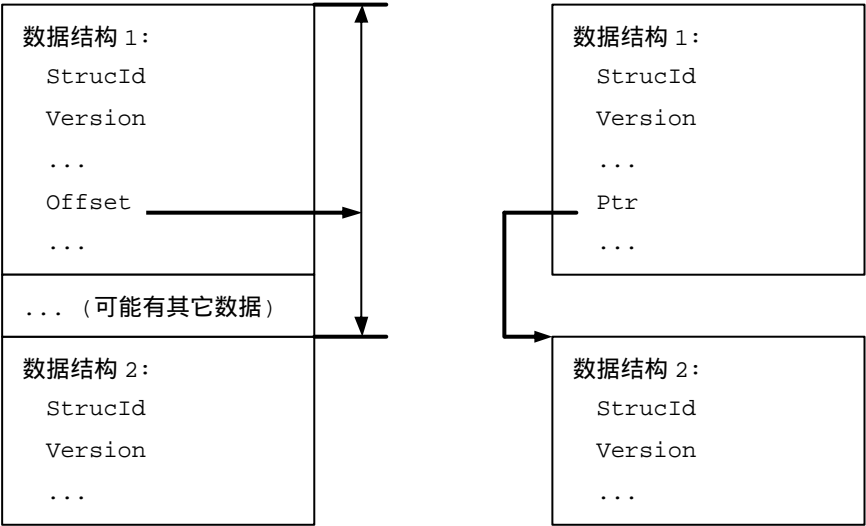


图 13-1 Offset 方式与 Ptr 方式

13.1.3 程序流程

MQI 函数共 13 个，它们之间调用的次序有一定的关系，如图 13-2。

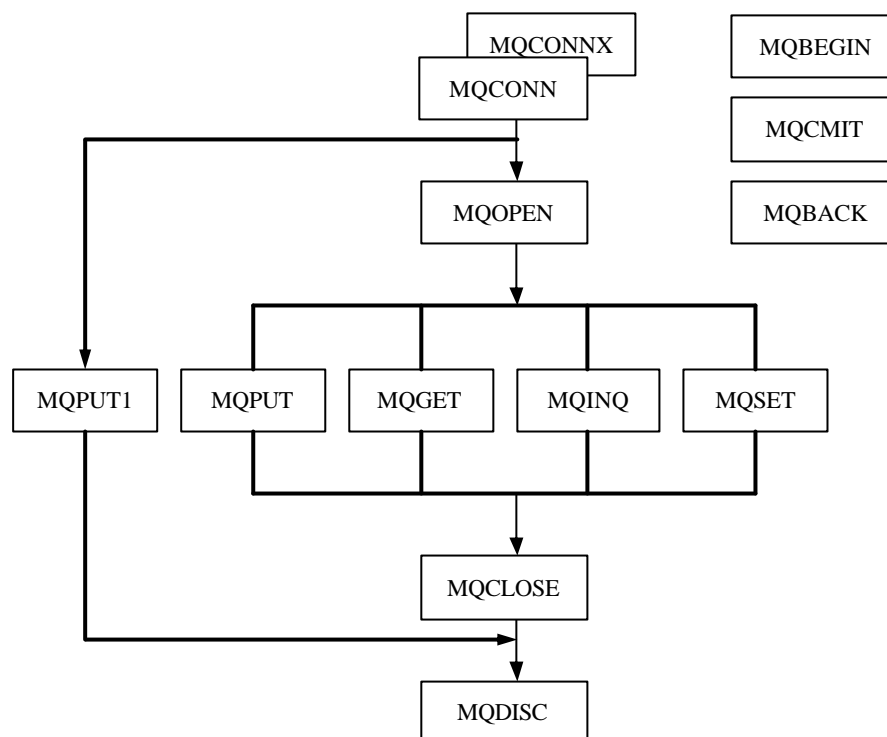


图 13-2 MQI 的调用次序

如前流程图所示，MQI 编程时大致如下：

1. 首先要调用 MQCONN 得到连接句柄，这个连接句柄会贯穿始终，以后所有的 MQI 调用都会以这个句柄为输入参数。WebSphere MQ 的交易也是以连接句柄为基础的，可以对连接上的所有动作提交或回滚。在编程时要注意，连接句柄不可以跨线程。MQCONN 是 MQCONN 的一个扩展形式，具有更强大的功能。
2. 调用 MQOPEN，在参数 MQOD 中指明打开的对象的类型和名字，常用的情况下打开的是一个队列，调用后得到对象的句柄，以后对该对象的所有操作都会使用到这个句柄。
3. 可以调用 MQGET 读取消息，或者调用 MQPUT 发送消息。MQGET/MQPUT 时消息为成消息头 (MQMD) 和消息体 (Buffer) 两部分，作为两个参数可以设置。另外，MQGMO 和 MQPMO 分别是 MQGET/MQPUT 的选项结构，不同的设置可以有不同的功能。
4. 也可以调用 MQINQ 查询对象属性，或者调用 MQSET 设置对象属性。一般可以一次性查询或设置多个对象属性，每个对象属性有一个对应的数值来唯一标记，称为 Selector。在多选时可以指定 Selector 数量和 Selector 数组。具体的属性值分成数值型和字符串型两种，在多选时可以指定多个数值属性和一个字符串和，分别放在数值属性数组和字符串中。
5. 调用 MQCLOSE 关闭对象。
6. 调用 MQDISC 关闭连接。

另有 MQBEGIN 表示交易开始。在本地交易时不需要用 MQBEGIN 标志交易起始点，只有全局交易时才需要这么做，参考交易相关章节。MQCMIT 提交交易，MQBACK 回滚交易。

所有的 API 都有两个输出参数：完成码和原因码。完成码可以取值 MQCC_OK，MQCC_WARNING，MQCC_FAILED，表示 API 调用的结果状态。如果不成功，原因码 (MQRC_*) 表示出错的具体原因。在 MQI 调用成功时，通常完成码为 MQCC_OK，原因码

为 MQRC_NONE。

13.1.4 例程

为了更好地理解 WebSphere MQ 应用程序流程，下面以 MQPUT 将消息放入队列为例，看看 C 语言的实现过程。

首先，要包含 MQI 的相关头文件 cmqc.h。

```
#include <cmqc.h>
```

然后，调用 MQCONN 连接队列管理器。其中第一个参数是输入参数，表示连接的队列管理器名。第二个参数是输出参数，表示连接句柄，该句柄会在以后的操作中贯穿始终。

```
MQCONN      ( "QM",                               // QM
              & mqhConn,                            // Connection Handle
              & mqlCompletionCode,                  // Completion Code
              & mqlReasonCode);                     // Reason Code
```

接着，调用 MQOPEN 打开队列，打开方式为写方式 (MQOO_OUTPUT)。MQOPEN 的第一个参数是输入参数，即前面得到的连接句柄。第二个参数是输入输出参数，表示对象描述符结构，该结构的 ObjectName 域是对象名称。第三个参数是输入参数，表示打开方式。第四个参数是输出参数，表示对象句柄，该句柄会在以后的对象操作中使用。

```
strcpy (mqod.ObjectName, "Q");
mqlOption = MQOO_OUTPUT + MQOO_FAIL_IF QUIESCING + MQOO_BIND_NOT_FIXED;
MQOPEN      (mqhConn,                               // Connection Handle
              & mqod,                               // Object Descriptor for Q
              mqlOption,                             // MQOPEN Options
              & mqhObj,                             // Object Handle
              & mqlCompletionCode,                   // Completion Code
              & mqlReasonCode);                     // Reason Code
```

随后准备一条消息，包括消息头和消息体。调用 MQPUT 将消息放入队列中，其中第一个和第二个参数是输入参数，就是前面得到的连接句柄和对象句柄。第三个参数是输入输出参数，表示消息头结构，在准备消息的时候可以设定消息头的某些域。第四个参数也是输入输出参数，表示发送选项，这里可以使用缺省值。第五和第六个参数是消息体的长度和内容。

```
memcpy (mqmd.MsgId, MQMI_NONE, sizeof (mqmd.MsgId));
memcpy (mqmd.CorrelId, MQCI_NONE, sizeof (mqmd.CorrelId));
memcpy (mqmd.Format, MQFMT_STRING, MQ_FORMAT_LENGTH);
sprintf (mqbMessageBuffer, "Hello, world.");
mqlMessageLength = strlen ((char *) mqbMessageBuffer);
MQPUT      (mqhConn,                               // Connection Handle
              mqhObj,                               // Object Handle
              & mqmd,                               // Message Descriptor
              & mqpmo,                               // Put Message Options
              mqlMessageLength,                      // Message Length
```

```

mqbMessageBuffer,          // Message Buffer
& mqlCompletionCode,       // Completion Code
& mqlReasonCode);         // Reason Code

```

最后，先调用 MQCLOSE 将对象关闭，再调用 MQDISC 断开队列管理器连接。当然，在调用时需要对应的连接句柄和对象句柄。

```

MQCLOSE    (mqhConn,          // Connection Handle
            & mqhObj,         // Object Handle
            0,                // MQCLOSE Options
            & mqlCompletionCode, // Completion Code
            & mqlReasonCode); // Reason Code

MQDISC     (& mqhConn,        // Connection Handle
            & mqlCompletionCode, // Completion Code
            & mqlReasonCode); // Reason Code

```

13.2 头文件

MQI 编程需要包含 cmqc.h 头文件。在 Windows 中，头文件目录为 <InstallDir>\Tools\c\include。在 UNIX 中，头文件目录为 <InstallDir>/inc。头文件目录中会含有多个头文件，它们同于各自不同的编程场合，对于 MQI 编程只需要包含 cmqc.h 即可。例：#include <cmqc.h>

表 13-1 WebSphere MQ 头文件

头文件	解释
amqsvmha.h	Data Conversion 编程
cmqbc.h	AI 编程
cmqc.h	MQI 编程
cmqcfc.h	PCF 和 Event 编程
cmqpssc.h	Pub/Sub 编程
cmqxc.h	User Exit 编程
cmqzc.h	Installable Service 编程

13.3 库文件

MQI 编译时需要的库文件会随着安装过程装入。在 Windows 中，库文件目录为 <InstallDir>\Tools\Lib，库文件名为 xxx.lib。在 UNIX 中，库文件目录为 <InstallDir>/lib，库文件名为 libxxx。WebSphere MQ 为 MQI 提供了多个库文件，支持 Client 端应用和 Server 端应用，支持单线程应用和多线程应用。应用程序要根据自己的特性连接不同的库文件，表 13-2。例：cc mqtest.c -o mqtest -lmqm

表 13-2 WebSphere MQ 库文件

库文件	解释
mqm	Server 端应用，单线程
mqm_r	Server 端应用，多线程
mqic	Client 端应用，单线程

13.4 编程参考

13.4.1 MQCONN

```
void MQENTRY MQCONN (  
    PMQCHAR    pQMgrName,          输入参数，队列管理器名  
    PMQHCONN    pHconn,            输出参数，连接句柄  
    PMQLONG     pCompCode,          输出参数，完成码  
    PMQLONG     pReason);           输出参数，原因码
```

说明：

- 连接队列管理器，得到连接句柄

13.4.2 MQCONNX

```
void MQENTRY MQCONNX (  
    PMQCHAR    pQMgrName,          输入参数，队列管理器名  
    PMQCNO     pConnectOpts,       输入参数，CONNX 选项结构  
    PMQHCONN    pHconn,            输出参数，连接句柄  
    PMQLONG     pCompCode,          输出参数，完成码  
    PMQLONG     pReason);           输出参数，原因码
```

说明：

- MQCONNX 是 MQCONN 的扩充版
- 可以通过 MQCNO 的设置，达到更强大的功能。可以用于 Client/Server 连接，支持 SSL 等等。

13.4.3 MQDISC

```
void MQENTRY MQDISC (  
    PMQHCONN    pHconn,            输入参数，连接句柄  
    PMQLONG     pCompCode,          输出参数，完成码  
    PMQLONG     pReason);           输出参数，原因码
```

说明：

- 断开与队列管理器的连接。
- 缺省情况下隐含对本地交易提交，如果程序未调用 MQDISC 直接结束，则隐含对本地交易回滚。

13.4.4 MQOPEN

```
void MQENTRY MQOPEN (
    MQHCONN  Hconn,           输入参数，连接句柄
    PMQOD    pObjDesc,       输入输出参数，对象描述结构
    MQLONG   Options,        输入参数，打开选项
    PMQHOBJS pHobj,          输出参数，对象句柄
    PMQLONG  pCompCode,       输出参数，完成码
    PMQLONG  pReason);        输出参数，原因码
```

说明：

- 打开对象，获得对象句柄
- MQOD 是对象描述结构，可以设置打开对象的类型、名称等等
- Options 指定对象的打开方式选项
 - MQOO_INPUT_AS_Q_DEF 读 打 开 ， 共 享 或 排 它 与 队 列 属 性 DefaultShareOption (DEFSOPT) 取值相同
 - MQOO_INPUT_SHARED 读方式打开，共享
 - MQOO_INPUT_EXCLUSIVE 读方式打开，排它
 - MQOO_BROWSE 浏览方式打开
 - MQOO_OUTPUT 写方式打开
 - MQOO_INQUIRE 查询方式打开
 - MQOO_SET 设置方式打开
 - MQOO_BIND_ON_OPEN 在打开时绑定对象，只能对指定对象操作
 - MQOO_BIND_NOT_FIXED 在打开时不绑定对象，在群集中可以对多个同名对象轮流操作
 - MQOO_BIND_AS_Q_DEF 在打开时是否绑定对象与队列属性 DefaultBind (DEFBIND) 取值相同
 - MQOO_FAIL_IF_QUIESCING 打开对象时，如果队列管理器正在关闭，操作失败

13.4.5 MQCLOSE

```
void MQENTRY MQCLOSE (
    MQHCONN  Hconn,           输入参数，连接句柄
    PMQHOBJS pHobj,          输入输出参数，对象句柄
    MQLONG   Options,        输入参数，关闭选项
    PMQLONG  pCompCode,       输出参数，完成码
    PMQLONG  pReason);        输出参数，原因码
```

说明：

- 在连接上关闭对象
- Options 指明对象的关闭方式
 - MQCO_NONE 仅关闭
 - MQCO_DELETE 关闭且删除队列，对于动态队列有效

■ MQCO_DELETE_PURGE

关闭且清空消息并删除队列，对于动态队列有效

13.4.6 MQPUT

```
void MQENTRY MQPUT (
    MQHCONN  Hconn,           输入参数，连接句柄
    PMQHOBJ  pHobj,          输入参数，对象句柄
    PMQMD    pMsgDesc,       输入输出参数，消息描述结构
    PMQPMO   pPutMsgOpts,    输入输出参数，PUT 选项结构
    MQLONG   BufferLength,    输入参数，消息长度
    PMQVOID  pBuffer,        输入参数，消息地址
    PMQLONG  pCompCode,      输出参数，完成码
    PMQLONG  pReason);       输出参数，原因码
```

说明：

- 发送消息到指定的队列中，队列对象句柄为 HOBJ
- 消息头放在 MQMD 中，消息体放在 Buffer 中，这里的 BufferLength 指的是消息的有效数据长度
- MQPMO 是一个 PUT 选项结构，通过设置可以使 MQPUT 具体各种功能

13.4.7 MQPUT1

```
void MQENTRY MQPUT1 (
    MQHCONN  Hconn,           输入参数，连接句柄
    PMQOD    pObjDesc,       输入输出参数，对象描述结构
    PMQMD    pMsgDesc,       输入输出参数，消息描述结构
    PMQPMO   pPutMsgOpts,    输入输出参数，PUT 选项结构
    MQLONG   BufferLength,    输入参数，消息长度
    PMQVOID  pBuffer,        输入参数，消息地址
    PMQLONG  pCompCode,      输出参数，完成码
    PMQLONG  pReason);       输出参数，原因码
```

说明：

- 简单地说，MQPUT1 = MQOPEN + MQPUT + MQCLOSE。由于每发送一条消息都需要打开和关闭队列，效率较低。所以，MQPUT1 通常只用于为了编程简洁且偶尔发送消息的场合。
- 相应的参数解释同 MQOPEN 和 MQPUT

13.4.8 MQGET

```
void MQENTRY MQGET (
    MQHCONN  Hconn,           输入参数，连接句柄
    PMQHOBJ  pHobj,          输入参数，对象句柄
```

PMQMD	pMsgDesc,	输入输出参数，消息描述结构
PMQGM0	pGetMsgOpts,	输入输出参数，GET 选项结构
MQLONG	BufferLength,	输入参数，所能容纳的最大消息长度
PMQVOID	pBuffer,	输入参数，输出内容，消息地址
PMQLONG	pDataLength,	输出参数，消息长度
PMQLONG	pCompCode,	输出参数，完成码
PMQLONG	pReason);	输出参数，原因码

说明：

- 从指定的队列中取一条消息，队列句柄为 HOBJ
- 消息头放在 MQMD 中，消息体放在 Buffer 中，这里 BufferLength 指 Buffer 的最大长度，DataLength 为消息的实际长度。
- MQGM0 是一个 GET 选项结构，通过设置可以使 MQGET 具有各种功能

13.4.9 MQINQ

void MQENTRY MQINQ (
MQHCONN	Hconn,	输入参数，连接句柄
PMQH0BJ	pHobj,	输入参数，对象句柄
MQLONG	SelectorCount,	输入参数，选项数量
PMQLONG	pSelectors,	输入参数，选项数组
MQLONG	IntAttrCount,	输入参数，数值属性数量
PMQLONG	pIntAttrs,	输入参数，输出内容，数值属性数组
MQLONG	CharAttrLength,	输入参数，字符串属性长度
PMQCHAR	pCharAttrs,	输入参数，输出内容，字符串属性值
PMQLONG	pCompCode,	输出参数，完成码
PMQLONG	pReason);	输出参数，原因码

- 查询对象属性，可以同时查询多个数值属性和一个字符串属性
- 每个属性由一个 Selector 对应，Selector 数组存放在 Selectors 参数中，SelectorCount 指明属性个数。数值属性的值存放在 IntAttrs 数组中，IntAttrCount 为数组长度。字符串属性值存放在 CharAttrs 中，CharAttrLength 为字符串长度

13.4.10 MQSET

void MQENTRY MQSET (
MQHCONN	Hconn,	输入参数，连接句柄
PMQH0BJ	pHobj,	输入参数，对象句柄
MQLONG	SelectorCount,	输入参数，选项数量
PMQLONG	pSelectors,	输入参数，选项数组
MQLONG	IntAttrCount,	输入参数，数值属性数量
PMQLONG	pIntAttrs,	输入参数，数值属性数组
MQLONG	CharAttrLength,	输入参数，字符串属性长度
PMQCHAR	pCharAttrs,	输入参数，字符串属性值
PMQLONG	pCompCode,	输出参数，完成码


```
PMQLONG pReason);
```

输出参数，原因码

说明：

- 设置对象属性，可以同时设置多个数值属性和一个字串属性
- 每个属性由一个 Selector 对应，Selector 数组存放在 Selectors 参数中，SelectorCount 指明属性个数。数值属性的值存放在 IntAttrs 数组中，IntAttrCount 为数组长度。字串属性值存放在 CharAttrs 中，CharAttrLength 为字串长度

13.4.11 MQBEGIN

```
void MQENTRY MQBEGIN (  
    MQHCONN Hconn,                输入参数，连接句柄  
    PMQBO pBeginOptions,          输入输出参数，BEGIN 选项结构  
    PMQLONG pCompCode,            输出参数，完成码  
    PMQLONG pReason);             输出参数，原因码
```

说明：

- 交易开始

13.4.12 MQCMIT

```
void MQENTRY MQCMIT (  
    MQHCONN Hconn,                输入参数，连接句柄  
    PMQLONG pCompCode,            输出参数，完成码  
    PMQLONG pReason);             输出参数，原因码
```

说明：

- 提交连接句柄上的交易

13.4.13 MQBACK

```
void MQENTRY MQBACK (  
    MQHCONN Hconn,                输入参数，连接句柄  
    PMQLONG pCompCode,            输出参数，完成码  
    PMQLONG pReason);             输出参数，原因码
```

说明：

- 回滚连接句柄上的交易

第 14 章 Java 编程

WebSphere MQ 的 Java 编程模式可以分成 Java Base 和 JMS 两种。对于前者，编程使用 MQ Java API，对于后者，使用 JMS API。这一章中，我们把前者简称为 Java 编程。WebSphere MQ 提供了一组 Java 包，同时提供了 Java 的编程接口。

14.1 安装

在安装时，将“Java 消息传递”模块选入（参见“安装”章节），会在 <InstallDir>/Java/lib 下安装一组 Jar 文件：

- Java Base
 - com.ibm.mq.jar
 - com.ibm.mqbind.jar
- JMS 支持相关标准
 - com.ibm.mqjms.jar
 - connector.jar Version 1.0
 - fscontext.jar Version 1.2 Beta 3
 - jms.jar Version 1.0.2
 - jndi.jar Version 1.2.1 (除 z/OS 和 OS/390)
 - jta.jar Version 1.0.1
 - ldap.jar Version 1.2.2 (除 z/OS 和 OS/390)
 - providerutil.jar Version 1.2

Java 编程会用到其中 com.ibm.mq.jar 和 com.ibm.mqbind.jar。WebSphere MQ 的 Java 程序与队列管理器的连接可以有两种方式：Client 方式或 Binding 方式，分别用到这两个 Jar 文件。

在安装完毕后，需要设置环境变量：

1. 公共环境变量，对所有平台有效
 - MQ_JAVA_INSTALL_PATH
 - MQ_JAVA_DATA_PATH

2. CLASSPATH

平台	CLASSPATH
AIX	CLASSPATH= /usr/mqm/java/lib/com.ibm.mq.jar: /usr/mqm/java/lib/connector.jar : /usr/mqm/samp/java/base:
HP-UX Solaris	和 CLASSPATH= /opt/mqm/java/lib/com.ibm.mq.jar: /opt/mqm/java/lib/connector.jar: /opt/mqm/samp/java/base:

Windows	CLASSPATH=
	mq_root_dir\java\lib\com.ibm.mq.jar;
	mq_root_dir\java\lib\connector.jar;
	mq_root_dir\java\lib\jta.jar;
	mq_root_dir\tools\java\base\;
z/OS 和 OS/390	CLASSPATH=
	install_dir/mqm/java/lib/com.ibm.mq.jar:
	install_dir/mqm/java/lib/connector.jar:
	install_dir/mqm/java/samples/base:
iSeries 和 AS/400	CLASSPATH=
	/QIBM/ProdData/mqm/java/lib/com.ibm.mq.jar:
	/QIBM/ProdData/mqm/java/lib/connector.jar:
	/QIBM/ProdData/mqm/java/samples/base:
Linux	CLASSPATH=
	/opt/mqm/java/lib/com.ibm.mq.jar:
	/opt/mqm/java/lib/connector.jar:
	/opt/mqm/samp/java/base:

3. Library

平台	库环境变量
AIX	LD_LIBRARY_PATH=/usr/mqm/java/lib
HP-UX	SHLIB_PATH=/opt/mqm/java/lib
Solaris	LD_LIBRARY_PATH=/opt/mqm/java/lib
Windows	PATH=install_dir\lib
z/OS 和 OS/390	LIBPATH=install_dir/mqm/java/lib
Linux	LD_LIBRARY_PATH=/opt/mqm/java/lib

环境变量生效之后，可以运行 WebSphere MQ for Java 的安装验证程序，以检验安装是否成功。

```
runmqsc [QM]

def chl (JAVA.CHANNEL) chltype (svrconn) trdtype (tcp)

runmqslsr -t tcp [-m QM] -p 1414

java MQIVP
```

14.2 编程设计

WebSphere MQ Java 编程目前支持 JDK 1.3 和 1.4，如果需要用到完整的 SSL 功能，比如 JSSE，则必须 JDK 1.4。另外，在 JMS 中的 JMSAdmin 对象管理工具可以基于文件系统，也可以基于 LDAP。前者需要 fscontext.jar 和 providerutil.jar，后者需要 ldap.jar 和 providerutil.jar。如果在 JMS 中编写 Pub/Sub 程序，需要有 Pub/Sub Borker 支持，它们可以是：

- SupportPac MA0C
- WebSphere MQ Integrator V2

- WebSphere MQ Event Broker

与 MQI 编程类似,我们也要连接上队列管理器,打开队列,在队列上对消息进行操作。只是具体的操作被 Java 类封装了,调用接口稍有不同。首先,通过创建 MQQueueManager 来获得一个队列管理器的操作对象,在这个对象上调用 accessQueue 来访问到对应的队列,同时获得队列的操作对象,在队列对象上调用 get () 可以读到消息,调用 set () 可以写出消息。

14.2.1 例程

```
qmgr = new MQQueueManager ("QM");
queue = qmgr.accessQueue ("Q", MQC.MQOO_INPUT_AS_Q_DEF +
MQC.MQOO_FAIL_IF_QUIESCING);

gmo = new MQGetMessageOptions ();
gmo.options = MQC.MQGMO_WAIT + MQC.MQGMO_CONVERT;
gmo.waitInterval = 2000; // milli-seconds

msg = new MQMessage ();
msg.messageId = MQC.MQMI_NONE;
msg.correlationId = MQC.MQMI_NONE;
msg.encoding = MQC.MQENC_NATIVE;
msg.characterSet = MQC.MQCCSI_Q_MGR;
msg.clearMessage ();

queue.get (msg, gmo);

str = msg.readLine ();
System.out.println ("Message: [" + str + "]);
```

首先,通过 new MQQueueManager 创建一个队列管理器对象 qmgr,队列管理器名为“QM”,基于该队列管理器可以指定访问队列 queue,队列名为“Q”。然后,通过 new MQGetMessageOptions 创建读消息时的功能选项 gmo,通过 new MQMessage 创建一个空的消息对象。接着,调用 queue.get (msg, gmo)实现在队列上对功能选项读取一条消息并放入 msg 对象中。最后,读取消息内容打印出来。

14.3 连接模式

如前所述,Java 程序与 WebSphere MQ 的连接有 Client 模式和 Binding 模式两种。前者实际上就是 MQ Client 编程,所以需要在 MQ Server 端定义服务通道,在 MQ Client 端用 MQEnvironment 类来指定 Server。后者是 MQ Server 编程。与普通的 C 语言编程不同,Java 编程受到一定的限制:

- Client Mode
 - 只能是 TCP/IP 连接

- 不支持 Connection Tables
- MQ 环境变量都无效
- 原先的通道定义和环境变量中的信息现在存放在 MQEnvironment 类中
- 出错信息写到 MQException 类指定的地方，通常是 Java Console
- 不支持 MQBEGIN 和 Fast Bindings
- 对于 MQEnvironment 类：
 - ◆ 将 MQEnvironment.properties 中 MQC.TRANSPORT_PROPERTY 设置为 MQC.TRANSPORT_MQSERIES
 - ◆ 将 MQEnvironment.hostname 设成 MQ Server 的机器名或 IP 地址
- Binding Mode
 - MQEnvironment 类中的大多数信息被忽略
 - 支持 MQBEGIN 和 Fast Bindings
 - 对于 MQEnvironment 类：
 - ◆ 将 MQEnvironment.hostname 设成 null (缺省)

事实上，程序中连接模式的选择是从 MQ Java Base 的角度看 MQEnvironment.hostname 和 MQEnvironment.properties.MQC.HOST_NAME_PROPERTY 是否为空，如果都为空，则为 Binding Mode，否则为 Client Mode。

MQEnvironment 相当于环境变量，其中的成员变量 properties 和其它 MQ 函数参数之间存在一定的覆盖关系，具体说来它们的优先级如下：

1. MQQueueManager 构造函数参数 properties
2. MQEnvironment.properties
3. 其它 MQEnvironment 成员变量
4. 系统缺省值

14.4 用户出口

Java 编程只支持 Client Mode 的 User Exit，不支持 Binding Mode 的 User Exit，所以 User Exit 的种类与 Client User Exit 相同，只有 Send Exit，Receive Exit 和 Security Exit 三种。用户需要实现 MQSendExit，MQReceiveExit 或者 MQSecurityExit 的 Interface，在 MQEnvironment 做相应的设置即可。

具体做法：

1. 创建自己的类实现 MQSendExit，MQReceiveExit MQSecurityExit 的 Interface，最集约的办法是用一个类同时实现三个 Interface。在 Interface 中的 Method，可以记录或改变传送的数据长度和内容，可以做加解密，加解压等等，最简单的办法是什么也不做，将 agentBuffer 参数原封不动地返回。例：

```
public class MQExit implements MQSendExit, MQReceiveExit, MQSecurityExit
{
    public byte[] sendExit (MQChannelExit channelExitParms,
        MQChannelDefinition channelDefParms, byte agentBuffer [])
    {
        return agentBuffer;
    }
}
```

```

    }

    public byte[] receiveExit (MQChannelExit channelExitParms,
                               MQChannelDefinition channelDefParms, byte agentBuffer [])
    {
        return agentBuffer;
    }

    public byte[] securityExit (MQChannelExit channelExitParms,
                                MQChannelDefinition channelDefParms, byte agentBuffer [])
    {
        return agentBuffer;
    }
}

```

2. 在应用程序中引用这个类即可

```

MQEnvironment.sendExit      = new MQExit ();
MQEnvironment.securityExit  = new MQExit ();
MQEnvironment.receiveExit   = new MQExit ();

```

3. 用户出口的运行环境与 Java Client 模式运行环境相同，需要在 Server 端配置通道。

```

DEFINE CHANNEL          (C_C.S)          +
          CHLTYPE        (SVRCONN)        +
          TRPTYPE         (TCP)           +
          REPLACE

```

```
start runmqlsr -m QM -t tcp -p 1414
```

相关的出错记录，Client 端可参考<InstallDir>/errors/AMQERRxx.LOG，Server 端可参考<InstallDir> qmgrs/QMgrName/errors/AMQERRxx.LOG

14.5 多线程

Java 语言自身有很强的多线程功能。MQ Java Base 所有的类都考虑到 Multi-Thread 访问同一个 MQ Handle 的问题，所以相关函数全部有 synchronized 功能（比如：通过 JAD 反编译看见的 MQQueue.jad：public synchronized void get (...) throws MQException）。所以在 MQ Java Base 没有 C 的问题，即多线程之间可以共享同一个连接句柄。

然而，对于这个句柄的操作是通过 Java 底层函数规定互斥的，实际上进行的是串行操作。这带来了编程上的灵活性，但也可能使多线程中的 MQ 操作相互干扰，比如，一个 Wait 方式读消息的操作就可能将其它线程的 MQ 操作锁住。在 MQ Java Base 中如果要想实现真正的并行操作，只有创建多个 MQQueueManager 对象。

如果线程中多个 MQ 操作具有多很强的连贯性，不希望被其它线程打断。可以使用 Java

的 synchronized 功能，锁住一个共享对象。

14.6 连接池

缺省情况下，每次 new MQQueueManager () 调用就意味着应用程序与队列管理器多了一条连接，在 MQ Client/Server 结构中表现为多了一条 TCP/IP 连接。每次 MQQueueManager.disconnect () 时这条连接断开，在 MQ Client/Server 结构中表现为这条 TCP/IP 连接断开。例如，在同一个线程中，如果反复连接再断开 MQ Connection，在 MQ Client/Server 结构中表现为反复连接再断开 TCP/IP 连接，在系统中用 netstat 命令可以看见一些痕迹，即每次连接断开后的 TIME_WAIT 状态。

例：

```
for (int i = 0; i < 10; i++)
{
    qmgr = new MQQueueManager (SQMName);
    ...
    qmgr.disconnect ();
}
```

```
C:> netstat
```

```
Active Connections
```

Proto	Local Address	Foreign Address	State
TCP	cyxt22:3417	localhost:1414	TIME_WAIT
TCP	cyxt22:3418	localhost:1414	TIME_WAIT
TCP	cyxt22:3419	localhost:1414	TIME_WAIT
TCP	cyxt22:3420	localhost:1414	TIME_WAIT
TCP	cyxt22:3421	localhost:1414	TIME_WAIT
TCP	cyxt22:3422	localhost:1414	TIME_WAIT
TCP	cyxt22:3423	localhost:1414	TIME_WAIT
TCP	cyxt22:3424	localhost:1414	TIME_WAIT
TCP	cyxt22:3425	localhost:1414	TIME_WAIT
TCP	cyxt22:3426	localhost:1414	TIME_WAIT
TCP	cyxt22:3427	localhost:1414	TIME_WAIT
TCP	cyxt22:3428	localhost:1414	TIME_WAIT

如果使用了连接池 (Connection Pool)，在 MQQueueManager.disconnect () 时 MQ 连接并没有真正断开，只是交回连接池，在下次 new MQQueueManager () 时重用。这时我们再用 netstat 命令就只能看到一条连接断开的痕迹，它是最后程序运行结束时回收连接池形成的。

例：

```
MQPoolToken token = MQEnvironment.addConnectionPoolToken ();
for (int i = 0; i < 10; i++)
{
    qmgr = new MQQueueManager (SQMName);
    ...
}
```

```

        qmgr.disconnect ();
    }
    MQEnvironment.removeConnectionPoolToken (token);

C:> netstat
Active Connections

```

Proto	Local Address	Foreign Address	State
TCP	cyxt22:3429	localhost:1414	TIME_WAIT

在并发线程的情况下，如果线程之间的 new MQQueueManager () 是串行的，则连接可以跨线程共享，如果是并发的，则可以为每个线程创建一个连接复用环境，连接在此线程的串行 new MQQueueManager () 操作之间是复用的。

14.6.1 例 1：线程之间串行建立连接

```

        for (int i = 0; i < 3; i++)
        {
            thread = new MQConnectionPoolThread (sQMName);
            thread.start ();
        }

class MQConnectionPoolThread extends Thread
{
    ...
    public void run ()
    {
        token = MQEnvironment.addConnectionPoolToken ();
        for (int i = 0; i < 5; i++)
        {
            try
            {
                synchronized (getClass ())
                {
                    qmgr = new MQQueueManager (sQMName);
                    sleep (1000);
                    qmgr.disconnect ();
                }
            }
        }
        MQEnvironment.removeConnectionPoolToken (token);
    }
}

C:> netstat

```


Active Connections			
Proto	Local Address	Foreign Address	State
TCP	cyxt22:3710	localhost:1414	TIME_WAIT

14.6.2 例 2：线程之间并行建立连接

```

for (int i = 0; i < 3; i++)
{
    thread = new MQConnectionPoolThread (sQMName);
    thread.start ();
}

class MQConnectionPoolThread extends Thread
{
    ...
    public void run ()
    {
        token = MQEnvironment.addConnectionPoolToken ();
        for (int i = 0; i < 5; i++)
        {
            try
            {
                qmgr = new MQQueueManager (sQMName);
                sleep (1000);
                qmgr.disconnect ();
            }
        }
        MQEnvironment.removeConnectionPoolToken (token);
    }
}

```

```
C:> netstat
```

Active Connections			
Proto	Local Address	Foreign Address	State
TCP	cyxt22:3711	localhost:1414	TIME_WAIT
TCP	cyxt22:3712	localhost:1414	TIME_WAIT
TCP	cyxt22:3713	localhost:1414	TIME_WAIT

MQEnvironment 中的缺省 Connection Pool 可以最多保持 10 条无用连接,每条连接最长保持 5 分钟。例如:同时有 15 个并发 new MQQueueManager(),会有 15 条有用连接。其中有 12 个很快就 MQQueueManager.disconnect() 了,这时应用会有 3 条有用连接,10 条无用连接,另有 2 条连接断开。

可以用 MQEnvironment.setDefaultConnectionManager (MQConnectionManager) 将自己

创建的 Connection Pool 设置成缺省的 Connection Pool。

MQConnectionFactory 是一个 private interface，在 MQ Java Base 中只有一个类实现了这个界面：MQSimpleConnectionFactory

MQQueueManager 的构造函数有一款为：public MQQueueManager (String queueManagerName, MQConnectionFactory cxManager) 表示创建的连接加入 cxManager 中的 Connection Pool，由 cxManager 进行管理（最多无用连接、超时等等）。

14.7 交易保护

14.7.1 本地交易 (Local LUW)

LocalLUW (Logic Unit of Work) 表示参与交易的只有一个资源，即 MQ 自身，LocalLUW 用于对同一个 Queue Manager 中的多个 Queue 操作的同步。与 MQI 中的 LocalLUW 类似，应用不需要标记交易的开始点，所以不能用 MQQueueManager.begin ()。在 Put/Get 时设置 options，例：

```
MQPutMessageOptions.options += MQC.MQPMO_SYNCPOINT;
MQGetMessageOptions.options += MQC.MQGMO_SYNCPOINT;
```

最后提交或回滚：

```
MQQueueManager.commit ()
MQQueueManager.backout ()
```

14.7.2 全局交易 (Global LUW)

GlobalLUW (Logic Unit of Work) 表示参与交易的可以有多个资源，即除了 WebSphere MQ 之外，还可以有其它资源，比如数据库。GlobalLUW 用于对多个资源的操作的同步，使用两阶段提交技术。

首先，要配置 WebSphere MQ 的 XA 资源。如果是 Windows 平台，在 WebSphere MQ Service 中配置，如果是 UNIX 平台，配置 QM.ini。这里假定另一个资源是 DB2。

名	值	解释
Name	MQDB2	可以随便起个名字，只要不与其它 XA 设置重复即可
SwitchFile	<InstallDir>\Java\lib\jdbc\jdbcd2.dll	
ThreadOfControl	PROCESS	也可以是 THREAD。 注意，这里必须是英文“PROCESS”或“THREAD”，不可以是中文“进程”或“线程”
XAOpenString	sample,db2admin,db2admin	

接着，配置数据库，如果是 DB2，配置 TP_MON_NAME，指定两阶段提交时的资源

管理器：

```
db2 update dbm cfg using TP_MON_NAME MQ
```

应用程序。使用两阶段提交的应用代码必须保证 MQCONN 所得到的 HCONN 具有 Thread Affinity 属性,即该连接句柄仅属于该线程。这在 C/C++ 编程时是缺省的,但在 Java 编程时不是。所以在 new MQQueueManager 之间设定此属性。否则在 MQBEGIN (C/C++) 或 MQQueueManager.begin (Java) 时会出现 MQException：

```
MQJE001：完成码 2，原因码 2121
```

```
Exception：com.ibm.mq.MQException：MQJE001：完成码 2，原因码 2121
```

设定属性 Thread Affinity 有两种办法，效果相同：

1. 不设 Hashtable

```
MQEnvironment.properties.put (MQC.THREAD_AFFINITY, new Boolean (true));  
qmgr = new MQQueueManager (sQMName);
```

2. 设置 Hashtable

```
Hashtable htProperties = new Hashtable ();  
htProperties.put (MQC.THREAD_AFFINITY, new Boolean (true));  
qmgr = new MQQueueManager (sQMName, htProperties);
```

编译和执行应用程序。用 JDK SE 1.4 编译和执行。注意将相应的 WMQ Java 库和 DB2 Java 库放入 CLASSPATH 中。

14.8 Trace

在 Java 编程时可以打开 Trace，WebSphere MQ 的一些内部操作会记录在 Trace 文件中。缺省情况下，Trace 是关闭的，打开 Trace 时要指定 Trace 级别。例：

```
MQEnvironment.enableTracing (2);           // 打开 Trace，指定 Trace 级别为 2  
...                                         // 中间这些代码会产生 Trace  
MQEnvironment.disableTracing ();          // 关闭 Trace
```

第 15 章 JMS 编程

15.1 JMS 对象

JMS 包含 JMS Provider 和 JMS Application 两部分。其中 JMS Provider 提供对标准 JMS Interface 调用的运行环境，JMS Application 作为应用调用这些 Interface。不同的 JMS Provider 在底层实现上可能不一样，但在接口与功能上都必须遵循 JMS 编程规范（参见 <http://java.sun.com/j2ee>），WebSphere MQ 也可以作为 JMS Provider。

JMS 中有一系列的类：ConnectionFactory，Connection，Session，MessageProducer，

MessageConsumer , Message 它们之间的关系如图 15-1。

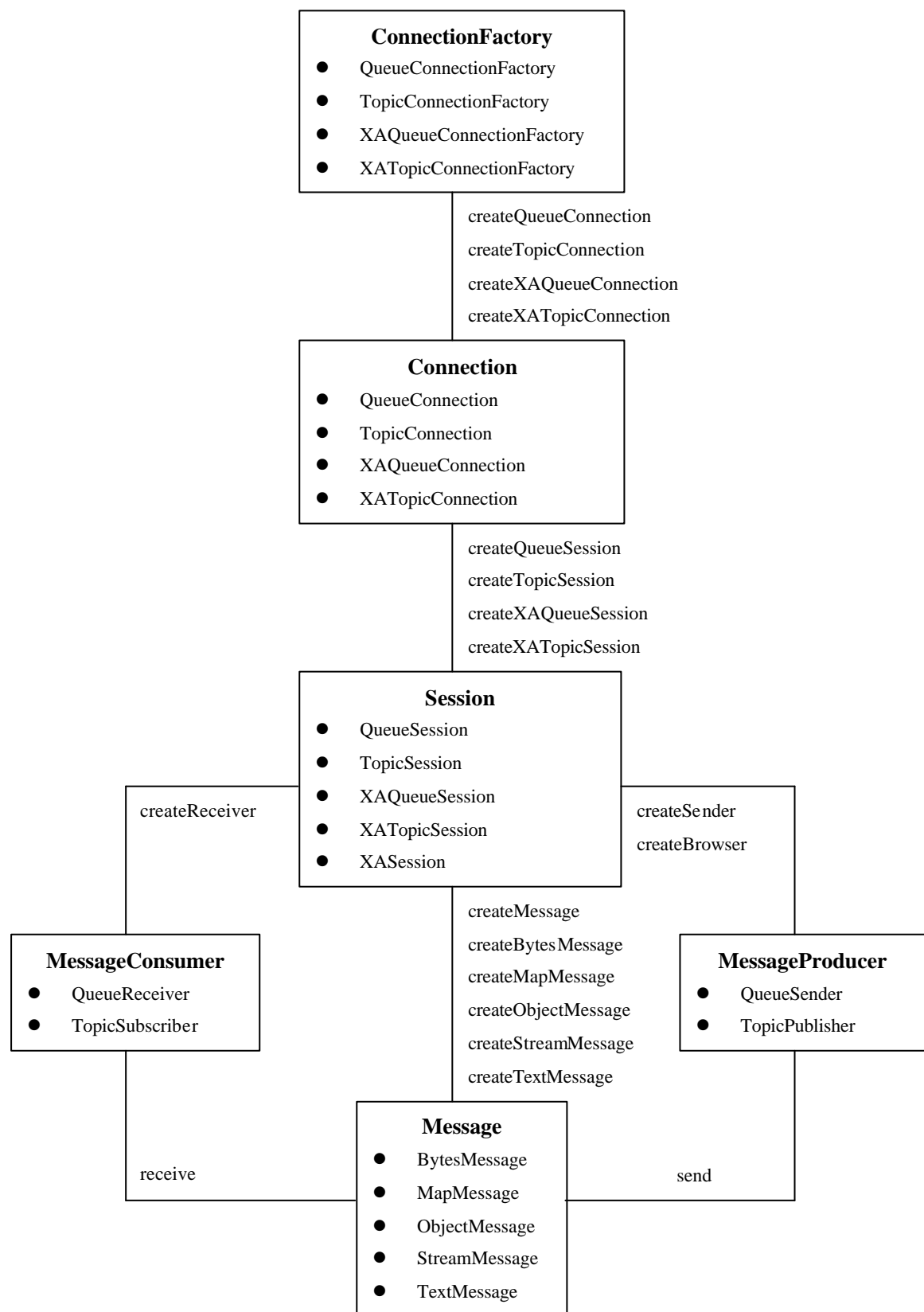


图 15-1 JMS 对象之间的关系

在 JMS 编程时，程序首先要找出 ConnectionFactory，以此创建 Connection，再建立

Session，以后所有的操作都以 Session 为基础。找出 Queue 或 Topic（统称 Destination），以此创建 QueueSender 或 TopicPublisher（统称 MessageProducer），在该对象上发送或发布消息。也可以在 Destination 基础上创建 QueueReceiver 或 TopicSubscriber（统称 MessageConsumer），在该对象上接收或订阅消息。

编程过程大致如此，但编程设计时还需要考虑：

- 是否使用 JNDI。JNDI 需要 JMS Provider 支持的 JNDI 环境，WebSphere MQ 5.3 支持文件、LDAP、IIOP（需要其它 J2EE 运行环境支持）三种方式。
- 应用通信是用点对点（Point-to-Point）方式还是用发布/订阅（Pub/Sub）方式。点对点方式是双方通信，发布/订阅方式可以完成多方通信。
- 同步读取消息还是异步读取消息。同步读取即主动读取方式，异步读取则需要设定 Listener，在消息到达后，自动调用 Listener 的 onMessage() 方法。
- JMS 操作是否需要交易保护。在交易保护下的 JMS 操作是可以提交或回滚的。
- 对于 MessageConsumer，消息在读取后通常需要确认（Acknowledgement），这个确认是自动产生还是程序控制生成。程序控制的方式往往意味着更大的灵活性，但也更麻烦一些。
- 消息是否是持久性的。持久性的消息会记入 Log，在 JMS Provider 重启后仍然保留，就象记入数据库一样。
- 消息是否需要超时设定。缺省情况下，消息都不会超时，有超时设置的消息在超时之后自动消失。
- 消息是否需要优先级设定。高优先级的消息在队列中可以排在靠前的位置。

等等

15.1.1 Context

JMS 编程中需要 Context 来指定 JNDI 上下文。一些预先在管理工具中配置好的对象，如 ConnectionFactory, Queue, Topic，可以通过 context.lookup() 的方法从上下文中直接获得，而无需动态创建。反过来说，如果不用 JNDI，可以不用关心 Context。

WebSphere MQ 中的 Context 需要有一个 Hashtable 指定 InitialContextFactory 和 ProviderURL 等参数。

创建 Context 对象：

- 使用 JNDI

```
Hashtable hashtable = new Hashtable();
hashtable.put (Context.INITIAL_CONTEXT_FACTORY, initialContextFactory);
hashtable.put (Context.PROVIDER_URL, providerURL);
hashtable.put (Context.REFERRAL, "throw");
context = new InitialDirContext (hashtable);
```

- 不用 JNDI

无

15.1.2 ConnectionFactory

ConnectionFactory 对象封装了一组对 Connection 的配置参数，应用程序可以通过

ConnectionFactory 来创建 Connection。如果使用 JNDI，则需要在 JNDI 环境中要事先定义好这对此 ConnectionFactory 的配置，比如队列管理器名。

JMSAdmin：

```
def qcf(jms/QueueConnectionFactory) qmgr(QM)
def tcf(jms/TopicConnectionFactory) qmgr(QM)
```

创建 ConnectionFactory 对象：

- 使用 JNDI

```
queueConnectionFactory = (QueueConnectionFactory) context.lookup
(factoryName);
topicConnectionFactory = (TopicConnectionFactory) context.lookup
(factoryName);
```

- 不用 JNDI

```
queueConnectionFactory = (QueueConnectionFactory) new
MQQueueConnectionFactory ();
((MQQueueConnectionFactory) queueConnectionFactory).setQueueManager
(qmgrName);
topicConnectionFactory = (TopicConnectionFactory) new
MQTopicConnectionFactory ();
((MQTopicConnectionFactory) topicConnectionFactory).setQueueManager
(qmgrName);
```

15.1.3 Connection

Connection 对象表示与 JMS Provider 之间的一条连接，对于 WebSphere MQ Client 方式，Connection 表示 MQ Client 与 MQ Server 之间的一条通道连接，在网络层可以看到一条 TCP 连接。

创建 Connection 对象：

```
queueConnection = queueConnectionFactory.createQueueConnection ();
topicConnection = topicConnectionFactory.createTopicConnection ();
```

关闭对象：

```
queueConnection.close ();
topicConnection.close ();
```

JMS 的资源 and 消息准入在 Connection 上控制

- Connection.close() 会释放所有 Connection 上相关的资源，通常程序结束时调用。
- Connection.start() 允许消息进入
- Connection.stop() 暂停消息进入

对消息的送出，Connection 不做控制，所以一般只有在 QueueReceiver，QueueBrowser，TopicSubscriber 才会用 start() 和 stop()

对消息的进入,用 `Connection.start ()` 设置允许,它必须在 `QueueReceiver.receive ()` 之前设置.但它与 `QueueReceiver` 本身的创建没有必然的先后次序。

可以是：

```
Connection.start ()
QueueSession.createReceiver ()
QueueReceiver.receive ()
```

也可以是：

```
QueueSession.createReceiver ()
Connection.start ()
QueueReceiver.receive ()
```

经过对 IBM WebSphere MQ JMS 的测试,如果程序在 `receive ()` 上等待,被强行退出时 `Connection.close ()` 没有被调用,资源没有被完全释放,则 MQ JMS 环境中有一个读动作会残留一段时间,紧接着的 `send ()` 消息会被吃掉。当这段时间过去之后,资源会被 JMS Provider 回收释放。

15.1.4 Session

一个 Session 通常含有与队列管理器的一个连接句柄 (HCONN)。所以,在 Session 上的后继操作都可以在一个交易内完成,根据 JMS 规范,从 Session 上读取消息需要确认,这个确认可以由系统自动完成,也可以由程序调用完成。创建 Session 的时候,需要指定是否需要交易保护及确认方式。

创建 Session 对象：

```
queueSession = queueConnection.createQueueSession (transacted,
acknowledgeMode);
topicSession = topicConnection.createTopicSession (transacted,
acknowledgeMode);
```

关闭对象：

```
queueSession.close ();
topicSession.close ();
```

JMS 的交易可以调用 `commit` 或 `rollback` 来控制。

- `Session.commit ()`
- `Session.rollback ()`
- `Session.close ()` 会加滚交易,同时也会释放 Session 中的资源
- `Session.recover ()` 会停止消息传送,再将所有未确认 (unacknowledged) 的消息设置为 `redelivered`,对所有 unacknowledged 消息按次序重新传送。

15.1.5 MessageConsumer

`MessageConsumer` 对象封装了一组配置参数,应用程序可以在 Session 上创建这一对象。

如果使用 JNDI，则需要在 JNDI 环境中事先定义好相应的配置，比如队列名、代码页、编码方式等等。

JMSAdmin：

```
def q(jms/Queue) queue(Q) ccsid(1381) encoding(RRR)
def t(jms/Topic) topic(T) ccsid(1381) encoding(RRR)
```

创建 MessageConsumer 对象：

- 使用 JNDI

```
queue = (Queue) context.lookup (queueName);
queueReceiver = queueSession.createReceiver (queue);
topic = (Topic) context.lookup (topicName);
topicSubscriber = topicSession.createSubscriber (topic);
```

- 不用 JNDI

```
queue = queueSession.createQueue (queueName);
queueReceiver = queueSession.createReceiver (queue);
topic = topicSession.createTopic (topicName);
topicSubscriber = topicSession.createSubscriber (topic);
```

接收消息：

```
message = queueReceiver.receive (1000 * 1); // milliseconds
topicSubscriber.setMessageListener (acknowledgeTextListener);
topicConnection.start ();
```

关闭对象：

```
queueReceiver.close ();
topicSubscriber.close ();
```

15.1.6 MessageProducer

类似地，MessageProducer 对象封装了一组配置参数，应用程序可以在 Session 上创建这一对象。如果使用 JNDI，则需要在 JNDI 环境中事先定义好相应的配置，比如队列名、代码页、编码方式等等。

JMSAdmin：

```
def q(jms/Queue) queue(Q) ccsid(1381) encoding(RRR)
def t(jms/Topic) topic(T) ccsid(1381) encoding(RRR)
```

创建 MessageConsumer 对象：

- 使用 JNDI

```
queue = (Queue) context.lookup (queueName);
queueSender = queueSession.createSender (queue);
topic = (Topic) context.lookup (topicName);
topicPublisher = topicSession.createPublisher (topic);
```


- 不用 JNDI

```
queue          = queueSession.createQueue (queueName);
queueSender    = queueSession.createSender (queue);
topic          = topicSession.createTopic (topicName);
topicPublisher = topicSession.createPublisher (topic);
```

发送消息：

```
queueSender.send (textMessage);
topicPublisher.publish (textMessage);
```

关闭对象：

```
queueSender.close ();
topicPublisher.close ();
```

15.1.7 MessageListener

通常 JMS 接收消息有同步和异步之分，同步接收即通过 `receive ()` 调用主动接收，异步接收是通过设置 `setMessageListener (MessageListener)` 来设定消息侦听类，侦听类实例自身类似于一个独立的线程，消息到来的时候会触发它的 `onMessage ()`。消息在 `onMessage ()` 中处理。

JMS 中有以下两个地方可以设置 `MessageListener`。通常选用前者，即对 `MessageConsumer (QueueReceiver, TopicSubscriber)` 设置。后者是高级用法，使用时要相对小心。

- `MessageConsumer.setMessageListener (MessageListener)`
普通用法。同一个队列上有多个侦听线程，其结果有不确定性。JMS 可能会随机选择一个侦听线程得到这条消息。
- `Session.setMessageListener (MessageListener)`
高级用法。一旦 `session` 上设置了侦听线程，这个 `session` 上就不再允许出现其它任何形式的接收动作，比如 `receive ()` 或是在其上的 `MessageConsumer` 设置别的 `MessageListener`。不是普通 JMS Client 使用的，在 SUN J2EE 1.3.1 上普通 JMS Client 调用 `setMessageListener ()` 不起作用。

经过对 IBM WebSphere MQ 的测试，如果有多个进程侦听同一个队列，当有消息到达时，其结果呈轮流分布。

`MessageListener` 用来异步读取消息。在 `MessageConsumer` 上设置 `MessageListener` 后，当消息到来时，应用程序会自动会运行 `MessageListener` 的 `onMessage ()` 方法。例：

```
public class TextListener implements MessageListener
{
    public void onMessage (Message message)
    {
        try
```

```

{
    if (message instanceof TextMessage)
        System.out.println (((TextMessage) message).getText ());
    else
        System.out.println ("Message of wrong type : " + message.getClass
        ().getName ());
}
catch (JMSEException e)
{
    System.err.println ("Error (JMSEException) : " + e.toString ());
}
}
}

textListener          = new TextListener ();
queueReceiver.setMessageListener (textListener);
queueConnection.start ();
topicSubscriber.setMessageListener (textListener);
topicConnection.start ();

```

15.1.8 Message

JMS 中每一条 Message 分成三部分：Header，Properties，Body。其中 Header 由固定的域组成，Properties 是一组名值对，可以由用户自己定义，Body 是消息内容，它可以属于 TextMessage，ObjectMessage，MapMessage，BytesMessage，StreamMessage 中的任何一个。

11.1.8.1 Header

Header 中有些域在发送时是无法通过 Message.setJMSxxx () 来设置的，通常 send 或 publish 时会重新覆盖设置这些值。表 15-1。

表 15-1 Header 中的域被设置的位置

域	在哪里被设置
JMSDestination	send or publish method
JMSDeliveryMode	send or publish method
JMSExpiration	send or publish method
JMSPriority	send or publish method
JMSMessageID	send or publish method
JMSTimestamp	send or publish method
JMSCorrelationID	Client
JMSReplyTo	Client
JMSType	Client
JMSRedelivered	JMS provider

JMSDeliveryMode , JMSExpiration , JMSPriority 可以通过 send 或 publish 的参数完成设置：

```
queueSender.send      (message , deliveryMode , priority , expiration);
topicPublisher.publish (message , deliveryMode , priority , expiration);
```

也可以通过 MessageProducer (QueueSender 和 TopicPublisher 的父类) 的 setXXX () 来完成设置：

```
queueSender.setDeliveryMode (deliveryMode);
queueSender.setPriority      (priority);
queueSender.setTimeToLive   (expiration);
queueSender.send (message);
```

JMSTimestamp 可以通过 setJMSTimestamp 设置 , 然后用 setDisableMessageTimestamp (true) , 使得 send 或 publish 时不设置 JMSTimestamp , 而沿用 Header 中的值。

JMSMessageID 理论上也可以通过 setJMSMessageID 设置 , 然后用 setDisableMessageTimestamp (true) , 使得 send 或 publish 时不设置 JMSMessageID , 而沿用 Header 中的值。但是 , 实际上 , 这个方法通常不起作用 , 消息 Header 中的 JMSMessageID 仍然会自行设置。

JMSDestination 是没有办法在发出前设置的 , 因为 QueueSender 或 TopicPublisher 的创建本身就需要指明 Queue 或 Topic

```
queueSender = queueSession.createSender (queue);
queueSender.send (message);
topicPublisher = topicSession.createPublisher (topic);
topicPublisher.publish (message);
```

Header 中的 JMSCorrelationID , JMSReplyTo , JMSType 是可以通过 Message.setJMSxxx () 来设置的。

Header 中的 JMSRedelivered 是 JMS provider 来管理和设置的。

15.1.8.2 Properties

Properties 是一系列的名值对 , 数量的名称可以由用户自己定义 , 用 setIntProperty () 和 getIntProperty () 来操作。

例：

```
message.setIntProperty ("messageNumber" , 3);
int msgnum = message.getIntProperty ("messageNumber");
```

15.1.8.3 Body

Body 可以是 TextMessage , ObjectMessage , MapMessage , BytesMessage , StreamMessage 中的任何一个。根据需要选用对应的类型 , 里面放入消息的内容。

15.2 编程设计

15.2.1 Persistence

Persistence 可以通过 QueueSender 或 TopicPublisher 的 setDeliveryMode () 来设置

```
public void MessageProducer.setDeliveryMode (int deliveryMode) throws
JMSEException
```

Persistence 也可以通过 QueueSender.send 或 TopicPublisher.publish 中的 DeliveryMode 参数在设定消息是否可以丢失的属性。

```
public void QueueSender.send (Message message ,int deliveryMode, int priority,
long timeToLive) throws JMSEException
```

```
public void TopicPublisher.publish (Message message ,int deliveryMode, int
priority, long timeToLive) throws JMSEException
```

- PERSISTENT 表示 JMS provider 要确保消息不丢失，即便是 JMS provider 失败.
- NON_PERSISTENT 表示 JMS provider 不需要确保消息不丢失

15.2.2 Priority

Priority 可以通过 MessageProducer (QueueSender 和 TopicPublisher 的父类) 的 setPriority () 设置

```
void MessageProducer.setPriority (int defaultPriority)     // 设置
MessageProducer 的缺省 Priority
```

Priority 也可以通过 send () 或 publish () 中的参数对单条消息进行设置

```
public void send (Message message ,int deliveryMode, int priority, long
timeToLive) throws JMSEException
```

```
public void publish (Message message ,int deliveryMode, int priority, long
timeToLive) throws JMSEException
```

Priority 可以取值 0 -- 9，共 10 档。缺省为 4。事实上，JMS API 将 Priority 分成两部分 0-4 表示普通消息 5-9 表示紧急消息。JMS 并不要求 JMS provider 严格地区分这 10 档 Priority，而只要能区分两档 (0-4，5-9) 即可。

15.2.3 Expiry

消息缺省状态下是不会超时的，但是如果设置的超时时间，则消息会超时。Expiry 可以通过 MessageProducer (QueueSender 和 TopicPublisher 的父类) 的 setTimeToLive () 设置

Expiry 也可以通过 send () 或 publish () 中的参数对单条消息进行设置

```
public void send (Message message ,int deliveryMode, int priority, long
```

```
timeToLive) throws JMSEException
public void publish (Message message ,int deliveryMode, int priority, long
timeToLive) throws JMSEException
```

如果 timeToLive 为 0，则不计超时。timeToLive 单位为毫秒。

15.2.4 Transaction

在 Session 上可以控制交易 ,首选 Session 要定义成 transacted ,然后通过调用 commit 或 rollback 来提交或回滚：

```
QueueSession createQueueSession (boolean transacted , int
acknowledgeMode)
TopicSession createTopicSession (boolean transacted , int
acknowledgeMode)

Session.commit ()
Session.rollback ()
```

注意，如果 transacted = true，则 acknowledgeMode 的值便无关紧要。

Transacted session 上的 send () 在没有提交之间，对于其它应用程序是不可见的。receive () 如果成功，其它应用程序也将看不见这条消息，如果提交，则消息被真正取走，如果回滚，则消息重新出现在队列中。这一点与 MQI 编程类似。

一旦提交或回滚，是针对 session 上之前的所有动作。程序在退出时如果没有做 Connection.close ()，则回滚。

让我们来看一个交易环境下的 JMS 应用操作，观察队列中消息的变化。表 15-2。

表 15-2 交易环境下的操作	
应用	队列
初始状态	空
send (A)	空
exit without close connection	空
send (A)	空
send (B)	空
send (C)	空
commit	A B C
receive 返回 A	B C
rollback	A B C
receive 返回 A	B C
receive 返回 B	C
commit	C
receive 返回 C	空
exit without close connection	C

15.2.5 Acknowledgment

消息确认 (Acknowledgment) 是针对 Message Consumer (Receiver ,Subscriber) 而言的, 对 Message Producer (Sender , Publisher) 无效.

JMS 消息只有在得到确认 (Acknowledgment) 后才被认为真正消耗了, 会从 Queue 或 Retained Publication 中消失。通常在以下三种情况下会消耗消息：

- Client 程序取消息
- Client 程序处理消息 (MessageListener 取消息)
- 取消息被确认

如果 Session 是交易型的, 则在交易 Commit 时自动确认, 在交易 Rollback 时回滚.

如果 Session 是非交易型的, 则 createQueueSession 和 createTopicSession 时参数会影响确认的行为：

- Session.AUTO_ACKNOWLEDGE
当客户端程序成功地取出消息或 MessageListener 成功地取出消息时, 自动发确认.
- Session.CLIENT_ACKNOWLEDGE
当客户端程序成功调用 Message.acknowledge () 时, 对 Session 上的动作发确认。比如, 在 Session 上取了 10 条消息, 对第五条 Message.acknowledge () 时, 系统对之前的 10 条消息全部确认.
- Session.DUPS_OK_ACKNOWLEDGE
如果 JMS provider 失败, 则可能出现重复消息。如果应用上可以容忍重复消息 (重复消息 JMS provider 会将消息的 JMSRedelivered 属性设置成 true 相当于调用 Message.setJMSRedelivered (true)), 这个选项可以增加性能, 但需要牺牲可靠性.

对于 Queue, 如果消息被取走但没有被确认, QueueSession 断开以后, 消息实际上还在队列里, 在下次某个客户程序访问队列时, 这些消息会 Redeliver 重现在队列中。

对于 Topic, JMS provider 同样会保留 TopicSession 中未被 TopicSubscriber 确认的消息。

有两种情况会在消息处理之后才发送确认信息 (Acknowledge)

- AUTO_ACKNOWLEDGE session 中的异步 receive (message listener)
自动确认只有在 onMessage () 退出时才会自动发出。也就是说, 在消息处理之后才发送。如果中途退出或回滚, 消息会自动 Redeliver
- CLIENT_ACKNOWLEDGE session 中的同步 receive
确认只有在人工调用 Message.acknowledge () 时才会发出。在人工调用 Session.recover () 时 Redeliver.

注意, 如果是 AUTO_ACKNOWLEDGE session 中的同步 receive, 确认消息会在 receiver () 调用后立即自动发送, 这时消息尚未处理, 如果处理失败, 则无法 Redeliver.

Message.acknowledge () 只对接收 (QueueReceiver 或 TopicReceiver) 消息有效, 对发

送消息无效。

- CLIENT_ACKNOWLEDGE session 中 send ()，消息立即发送，也不需要确认 Message.acknowledge ()
- CLIENT_ACKNOWLEDGE session 中 receive ()，调用成功后消息从队列中消失，用 Receiver 或 Browser 都看不见了。但是如果消息未确认 ,Connection.close () 后消失了消息又会重新出现在队列中。另外，对 CLIENT_ACKNOWLEDGE session 中某一条消息的确认，也会同时会对这个 Session 之前 receive 或 onMessage 的所有消息都确认了。

Topic 的表现可以参见 Queue 的表现。让我们看 Acknowledge 分别在 Send/Receive 模式和 Publish/Subscribe 模式中的行为，注意队列中消息的变化。表 15-3 和 15-4。

表 15-3 Acknowledge 例 1 (Send, Receive)

应用	队列
初始状态	空
send (A)	A
send (B)	A B
send (C)	A B C
acknowledge	A B C 或出错
receive 返回 A	B C
receive 返回 B	C
acknowledge	C
receive 返回 C	空
exit with close connection	C
receive 返回 C	空
exit without close connection	C

表 15-4 Acknowledge 例 2 (Publish, Subscribe)

Publisher	Subscriber	队列
publish (A)	onMessage (A)	空
publish (B)	onMessage (B)	空
	acknowledge	
publish (C)	onMessage (C)	空
	exit without acknowledge	C

15.2.6 Message Selektor

Message Selektor 是可以对 Header 和 Properties 进行检索，以选出匹配的消息。但 Selector 对 Body 中的内容无法检索。

Selector 实际上是一个字串 (String)，语法上符合 SQL92。通常是对 Header 和 Properties 上的字段进行匹配及逻辑组合

例：

```
message.setIntProperty ("NumberOfOrders", 2);
```

Selector : "NumberOfOrders > 1" // OK

```
message.setStringProperty ("NumberOfOrders", "2");
```

Selector : "NumberOfOrders > 1" // ERROR, 因为 String (2) 是无法与数字 (1) 比较大小

Selector : "JMSType = 'car' AND color = 'blue' AND weight > 2500"

Selector : "JMSCorrelationID = 'ID:_cyxt21_1064057051774_10.1.1.2'"

Selector : "JMSDeliveryMode = 'NON_PERSISTENT'"

设定 Message Selector 与应用程序本身的运行环境有关 :

- 如果程序是 JMS 应用, 可以用 createReceiver (queue, messageSelector) 来设定 Message Selector :

```
public QueueReceiver createReceiver (Queue queue, java.lang.String  
messageSelector) throws JMSException
```

- 如果程序是 Message Driven Bean (MDB), 可以在布署的时候指定 Message Selector

15.2.7 Temporary Destination

JMS 程序中可以通过 QueueSession.createTemporaryQueue () 或 TopicSession.createTemporaryTopic () 来创建临时对象 TemporaryQueue 和 TemporaryTopic。这个对象只在创建的这个 Connection 中有效, 一旦 Connection 关闭, 对象连同对象中的消息一起自动消失。

临时对象中的消息只在本次 Connection 中有效, 一旦 Connection 关闭, 消息便自动消失。适合于简单的 Request/Reply 模式, 通常可以创建了临时对象后, 发出的 Request 消息请求中消息头 JMSReplyTo 设置成该临时对象, 则 Reply 回来的消息会自动放入此临时对象, 再由本次 Connection 读出。

15.2.8 Durable Subscriber

Durable Subscriber 即便在不活动的情况下, Broker 也会记录 Publisher 的内容, 当 Durable Subscriber 再次活动的时候, 会收到这些内容, 不会遗漏。

Publisher 的内容可以记在 Queue 中, 也可以记在 Broker 中. 由 SUBSTORE() 调控。

- SUBSTORE(Queue)

订阅信息存放在 SYSTEM.JMS.ADMIN.Queue 和 SYSTEM.JMS.PS.STATUS.Queue 中, 如果订阅失败, 信息留在这两个队列中. JMS 与每个队列管理器有一条连接, 监控失败的订阅. 这是一个长交易, 如果系统很忙, 也可能造成 Log 增长过快, 当然您可以调整 STATREFRESHINT 参数来改变这一点, 使长交易尽快刷新, 释放对 Log 的占用。

- SUBSTORE(Broker)

只有最近的 QMgr 和 Broker 支持这种方式, 订阅信息存放在 Broker 中, 如果订阅失败, 信息会自动删除, 但 Broker 会留一条消息在 SYSTEM.JMS.REPORT.Queue 上, 用

于 Cleanup. 每个应用程序有一个单独的线程用于 Cleanup, 缺省情况下每 10 分钟运行一次, 当然你也可以调整 CLEANUPINT 参数来改变这一周期.

- SUBSTORE(MIGRATE)

如果系统发现可以采用 SUBSTORE(BROKER), 则采用. 否则, 采用 SUBSTORE(QUEUE). 系统采用这种方式后, 原先存放在 Queue 中的订阅信息会移入 Broker 中.

注意, 这种方式需要有 TopicConnectionFactory 有 CLIENTID 属性.

SUBSTORE(QUEUE) 和 SUBSTORE(BROKER) 是各自独立的两个层面的订阅方式, 由 SUBSTORE(MIGRATE) 试图将其统一到 SUBSTORE(BROKER). 在不同的层面中可以有同名的订阅, 但是在同一个层面中, 后来的同名订阅会失败.

15.3 MQ JMS 运行环境

15.3.1 JMS Interface 与 MQ JMS Object

MQ 作为 JMS Provider 提供了 com.ibm.mq.jms 包, 全部实现了 JMS Interface, 且 Object 名字几乎与 JMS Interface 一一对应. 如:

JMS Interface	MQ JMS Object
Queue	MQQueue
Topic	MQTopic
Message	MQMessage

总的来说, MQ JMS 是 JMS Interface 的超集

MQ JMS 也定义了 JMS Interface 中没有的类, 比如 Cleanup, MQQueueEnumeration. 这些类都与 MQ 环境相关.

MQ JMS 还定义了 JMS Interface 中没有的 method, 比如 MQDestination.getCCSID(), MQQueue.setBaseQueueManagerName(String s). 这些方法都与 MQ 本身的特性相关.

MQ JMS 中也有一些 method 与 JMS Interface 说明不相符的, 比如

```
message.setJMSTimestamp (0L);  
queueSender.setDisableMessageTimestamp (resetTimestamp);
```

对 MQ 都是不任何作用的, 这也是由 MQ 特性所限.

所以, 可以说 MQ 作为 JMS Provider 实现了 JMS 所有的 Interface, 基本实现了 JMS 的功能. 但是, 严格地说, MQ 由于受限于自身的特性, 并没有完全实现 JMS 规范 (JMS Specification) 上的功能, 比如 MQ JMS 无法产生出一条不带 Timestamp 或 Timestamp 为零的消息. 因为底层的 MQ 决定任何一条消息都会自动加上 Timestamp, 这是程序无法干预的.

15.3.2 JNDI

建议将队列管理器的 CCSID 改为 819 (中文环境下缺省为 1381. 目前产品对 JMS

Pub/Sub 会有问题)

修改 JMS 配置文件 <InstallDir>\Java\bin\JMSAdmin.config 中的 INITIAL_CONTEXT_FACTORY 和 PROVIDER_URL, 原先缺省的是 LDAP 设置. 这里假定 <InstallDir> 为 D:\WMQ.

```
INITIAL_CONTEXT_FACTORY=com.sun.jndi.fscontext.RefFSContextFactory
PROVIDER_URL=file:/D:/WMQ/Java/bin
```

启动 JMSAdmin.bat, 它会在当前目录下寻找配置文件 JMSAdmin.config. 可以用 dis ctx 命令 看看当前目录下的文件.

```
初始化上下文> dis ctx
```

```
目录 初始化上下文
```

runjms.bat	java.io.File
IVTRun.bat	java.io.File
IVTSetup.bat	java.io.File
IVTTidy.bat	java.io.File
JMSAdmin.bat	java.io.File
MQJMS_PSQ.mqsc	java.io.File
PSIVTRun.bat	java.io.File
PSReportDump.class	java.io.File
formatLog.bat	java.io.File
Cleanup.bat	java.io.File
postcard.bat	java.io.File
JmsPostcardSample.ini	java.io.File
postcard.ini	java.io.File
JMSAdmin.config	java.io.File
.bindings	java.io.File

```
15 对象
```

```
0 上下文
```

```
15 绑定, 0 管理
```

创建 JMS 中的 QueueConnectionFactory 和 Queue, 不妨假设名为 jms/QueueConnectionFactory 和 jms/Queue

```
初始化上下文> def qcf(jms/QueueConnectionFactory) qmgr(QM)
```

```
初始化上下文> def q(jms/Queue) queue(Q) ccsid(1381) encoding(RRR)
```

```
初始化上下文> def tcf(jms/TopicConnectionFactory) qmgr(QM)
```

```
初始化上下文> def t(jms/Topic) topic(T) ccsid(1381) encoding(RRR)
```

注意:

- 如果队列中要存放中文双字节信息, 必须在这里设置 ccsid(1381), 否则每一个中文字都会变成单字节 0x3F。
- 因为 JAVA 中 NATIVE 的数字表达方式与平台无关, 为高字节在前, 与逻辑方式相同, 如 1L 在内存中为 0x00000001, 无论在什么平台都是这样. 但是, C 语言中 NATIVE 数字表达方式与平台有关, 在 Windows 平台上为低字节在前, 如 1L

在内存中为 0x01000000. 在 AIX 平台上为高字节在前, 1L 在内存中为 0x00000001. 这里, 设置 RRR 是为了让 JMS 消息在 QM.CCSID=1381 的情况下能够被 C 应用读取. 如果 QM.CCSID=819, 这里的 RRR 则不是必须的, JMS 消息一样能够被 C 应用读取.

发现当前目录下出现文件 .bindings, 里面记录了配置信息. 用 dis 命令查看创建的对象。

```
初始化上下文> dis qcf(jms/QueueConnectionFactory)
```

```
FAILIFQUIESCE(YES)
```

```
QMANAGER( )
```

```
USECONNPOOLING(YES)
```

```
TEMPMODEL( SYSTEM.DEFAULT.MODEL.QUEUE)
```

```
MSGBATCHSZ(10)
```

```
TRANSPORT(BIND)
```

```
SYNCPOINTALLGETS(NO)
```

```
TEMPQPPREFIX(AMQ.*)
```

```
MSGRETENTION(YES)
```

```
RESCANINT(5000)
```

```
POLLINGINT(5000)
```

```
VERSION(2)
```

```
初始化上下文> dis q(jms/Queue)
```

```
FAILIFQUIESCE(YES)
```

```
QUEUE(Q)
```

```
QMANAGER(QM)
```

```
PERSISTENCE(APP)
```

```
CCSID(1381)
```

```
TARGETCLIENT(JMS)
```

```
ENCODING(RRR)
```

```
PRIORITY(APP)
```

```
EXPIRY(APP)
```

```
VERSION(1)
```

```
初始化上下文> dis tcf(jms/TopicConnectionFactory)
```

```
DIRECTAUTH(BASIC)
```

```
BROKERCCSUBQ( SYSTEM.JMS.ND.CC.SUBSCRIBER.QUEUE)
```

```
BROKERPUBQ( SYSTEM.BROKER.DEFAULT.STREAM)
```

```
QMANAGER(QM)
```

```
SUBSTORE(MIGRATE)
```

```
BROKERVER(V1)
```

```
USECONNPOOLING(YES)
```

```
SPARSESUBS(NO)
```

```
POLLINGINT(5000)
```

```
RESCANINT(5000)
```

```

CLONESUPP(DISABLED)
BROKERQMGR( )
BROKERSUBQ(SYSTEM.JMS.ND.SUBSCRIBER.QUEUE)
STATREFRESHINT(60000)
TRANSPORT(BIND)
SYNCPPOINTALLGETS(NO)
PUBACKINT(25)
PROXYPORT(443)
CLEANUPINT(3600000)
MULTICAST(DISABLED)
MSGSELECTION(CLIENT)
VERSION(2)
MSGBATCHSZ(10)
BROKERCONQ(SYSTEM.BROKER.CONTROL.QUEUE)
FAILIFQUIESCE(YES)
CLEANUP(SAFE)

```

初始化上下文> dis t(jms/Topic)

```

FAILIFQUIESCE(YES)
BROKERDURSUBQ(SYSTEM.JMS.D.SUBSCRIBER.QUEUE)
TOPIC(T)
BROKERVER(V1)
PERSISTENCE(APP)
CCSID(1381)
TARGCLIENT(JMS)
ENCODING(RRR)
BROKERCCDURSUBQ(SYSTEM.JMS.D.CC.SUBSCRIBER.QUEUE)
PRIORITY(APP)
EXPIRY(APP)
VERSION(1)

```

执行 Java 程序，通过 Hashtable 创建 Context，就可以 lookup 找到创建的对象了。

```

Hashtable hashtable = new Hashtable();
hashtable.put (Context.INITIAL_CONTEXT_FACTORY ,
initialContextFactory);
hashtable.put (Context.PROVIDER_URL , providerURL);
context = new InitialDirContext (hashtable);

queueConnectionFactory = (QueueConnectionFactory) context.lookup
("jms/QueueConnectionFactory");
queue = (Queue) context.lookup ("jms/Queue");

```

15.3.3 Client

Server 端的配置

```
RUNMQSC
```

```
ALT QMGR CCSID (819)
```

```
DEFINE CHANNEL (C_C.S) +
```

```
CHLTYPE (SVRCONN) +
```

```
TRPTYPE (TCP) +
```

```
REPLACE
```

```
start runmqslr -m QM -t tcp -p 1414
```

Client 端的配置

```
set MQSERVER=C_C.S/TCP/127.0.0.1(1414)
```

```
set MQCCSID=819
```

JMS Admin

```
def qcf(jms/ClientQCF) transport(client) hostname(127.0.0.1)
port(1414) channel(C_C.S) qmgr(QM)
```

检查 amqspuc 是否可以正常工作

```
amqspuc Q QM
```

在实现中有时会报 2059 (MQRC_Q_MGR_NOT_AVAILABLE) 错,这可能是 Client 端环境与 QMgr 环境不一致,设置 set MQCCSID=819 (与 QM.CCSID 相同)

优先级 (覆盖关系):

程序 > JMS Admin 配置 > 环境变量

程序:

```
((MQQueueConnectionFactory) queueConnectionFactory).setTransportType
(JMSC.MQJMS_TP_CLIENT_MQ_TCPIP);
((MQQueueConnectionFactory) queueConnectionFactory).setQueueManager
("QM");
((MQQueueConnectionFactory) queueConnectionFactory).setHostName
("127.0.0.1");
((MQQueueConnectionFactory) queueConnectionFactory).setPort (1414);
((MQQueueConnectionFactory) queueConnectionFactory).setChannel ("C_C.S");
```

JMS Admin 配置

```
def qcf(jms/ClientQCF) transport(client) hostname(127.0.0.1) port(1414)
channel(C_C.S) qmgr(QM)
```

环境变量

```
set MQSERVER=C_C.S/TCP/127.0.0.1(1414)
```

```
set MQCCSID=819
```

15.3.4 CCSID & Encoding

JMS 缺省情况下 send 出去的消息格式为：MQMD + MQRFH2 + Body。缺省情况下，MQMD.CodedCharSetId = 819，所以在 CCSID 为 1381 的队列管理器上，这条消息被 MQGET 时会出现 MQRC_NOT_CONVERTED。

((com.ibm.mq.jms.MQQueue) queue).setTargetClient (1) 可以假定对方的 MQ 程序非 JMS 应用。send 出去的消息格式为：MQMD + Body，不再有 MQRFH2 了，MQMD.CodedCharSetId = 1208 (UTF8)。再用 ((com.ibm.mq.jms.MQQueue) queue).setCCSID (1381) 即可强行设定 MQMD.CodedCharSetId。

```
((com.ibm.mq.jms.MQQueue) queue).setTargetClient (1);  
((com.ibm.mq.jms.MQQueue) queue).setCCSID (1381);
```

少了 MQRFH2 的消息一样可以被 JMS Receiver 识别并取走。这样，MQ 的另一头，读消息的程序可以是 MQ Java Base 程序、MQ JMS 程序、MQ C 程序。

由于 WebSphere MQ 中 JMS 消息的 CCSID 为 819，可能与现实的队列管理器的 CCSID 不同，由此可能会造成一定的麻烦。如果环境是 Win2000 (中文版) + WMQ (中文版) + CSD04 + MA0C，建议将队列管理器的 CCSID 改为 819。对于 Encoding，可以在 JMSAdmin 工具配置中指定 encoding。

- 对于 Queue

JMSAdmin：

```
def qcf(jms/QueueConnectionFactory) qmgr(QM)  
def q(jms/Queue) queue(Q) ccsid(819)
```

MQMD.Encoding = 273

MQMD.CodedCharSetId = 819

JMS 程序之间可以读写配合，JMS 程序与 C 程序之间也可以读写配合。

JMSAdmin：

```
def qcf(jms/QueueConnectionFactory) qmgr(QM)  
def q(jms/Queue) queue(Q) encoding(RRR)
```

MQMD.Encoding = 546

MQMD.CodedCharSetId = 819

JMS 程序之间可以读写配合，JMS 程序与 C 程序之间也可以读写配合。

- 对于 Topic

JMSAdmin：

```
def tcf(jms/TopicConnectionFactory) qmgr(QM)  
def t(jms/Topic) topic(T) ccsid(819)
```

MQMD.Encoding = 273

MQMD.CodedCharSetId = 819

JMS 程序之间可以读写配合。

JMSAdmin：

```
def tcf(jms/TopicConnectionFactory) qmgr(QM)
def t(jms/Topic) topic(T) ccsid(819) encoding(RRR)
MQMD.Encoding = 546
MQMD.CodedCharSetId = 819
JMS 程序之间可以读写配合.
```

15.4 ASF

Application Server Facilities (ASF) 是 JMS 规范中留给 JMS Provider 的一个高级接口。主要表现为以下一些 Interface:

- Interface
 - ConnectionConsumer
 - ServerSession
 - ServerSessionPool
 - QueueConnection
 - TopicConnection
 - Session
- Method
 - public ConnectionConsumer QueueConnection.createConnectionConsumer (Queue queue, java.lang.String messageSelector, ServerSessionPool serverSessionPool, int maxMessages) throws JMSEException
 - public ConnectionConsumer TopicConnection.createConnectionConsumer (Topic queue, java.lang.String messageSelector, ServerSessionPool serverSessionPool, int maxMessages) throws JMSEException

JMS Provider 可以通过实现 ConnectionConsumer 接口,使得应用服务程序可以实现高效并行处理消息。通常,一个 Connection (比如 QueueConnection 或 TopicConnection) 上可以创建 (createConnectionConsumer) 多个 ConnectionConsumer 及其对应的 MessageSelector 和 ServerSessionPool,而每个 ConnectionConsumer 对应的 ServerSessionPool 通常是一组线程,每条线程含一个 SessionSession,在这个 ServerSession 上有一个 MessageListener。

应用服务程序的并行度可以由 ServerSessionPool 中的 ServerSession 数量控制,当消息到来得比较慢时,每条消息都会触发某一个 ServerSession 的 start () 方法,当消息到来得非常快时,JMS 会累积一批消息触发一次 ServerSession.start (),累积的消息数量不超过 createConnectionConsumer () 时指定的 maxMessages 值。

由于采取了并发和触发的技术,ASF 可以应付大量涌入的消息流,处理效率高。另外,由于一个 Connection 上可以有多个 ConnectionConsumer,可以有各自不同的 MessageSelector,表示对不同的消息感兴趣,可以这样对涌入的消息流分组。注意 MessageSelector 之间的配合,不要在队列中留下无法匹配的消息,这种消息会呆在队列中参与以后的每一次的匹配,积累多了会大大影响性能。

第 16 章 ActiveX 编程

WebSphere MQ for Windows 还提供了一种 Microsoft Windows 应用程序通用的编程模式——ActiveX。Microsoft 在很早就开始倡导 Component Object Model (COM) 的编程模式，在 COM 的框架下，应用程序可以很方便地定位并使用 COM 组件，而无所谓这些组件开发使用的编程语言。

ActiveX 是基于 COM 上的一组技术，含有集成式应用开发技术、组件重用和互连技术等等。它使得用不同的语言开发出来的各种功能的组件可以被共享，并很方便地嵌入应用程序中。在 ActiveX 环境下，各种开发工具和开发语言可以协同工作，封装后的代码可以共享。由于 ActiveX 组件是面向对象的，所以各种编程工具和编程语言对组件的操作几乎相同。

WebSphere MQ 也提供了 ActiveX 的组件：

- IBM MQSeries Automation Classes for ActiveX (MQAX)
- IBM MQSeries MQAI COM Library
- IBM MQSeries ADSI 5.1 Library

其中最为重要的组件为 WebSphere MQ Automation Classes for ActiveX (MQAX)，它提供了 ActiveX 框架下对 WebSphere MQ 的操作接口。对应的动态连接库为 MQAX200.dll，可以用 mqaxlevel 命令查看 Code Level，详见<<附录 WebSphere MQ 命令一览表>>。

由于 ActiveX 的应用极其广泛，可以用各种方式调用，比如 VB, VC, Delphi, PowerBuilder, JavaScript, VBScript, Excel Macro 等等。具体编程见本书例题。

16.1 MQAX

16.1.1 程序设计

16.1.1.1 对象

由于 MQAX 对 MQ API 进行了进一步的封装，编程上与传统的 MQI 有些不同。MQAX 提供了一组对象：

- | | |
|--------------------------|-------------------------|
| ● MQSession | MQ 会话，即连接到队列管理器后的所有操作过程 |
| ● MQQueueManager | 队列管理器 |
| ● MQQueue | 队列 |
| ● MQMessage | 消息 |
| ● MQPutMessageOptions | 放消息选项 |
| ● MQGetMessageOptions | 取消息选项 |
| ● MQDistributionList | 分发列表 |
| ● MQDistributionListItem | 分发列表项，即组成分发列表的最小单位 |

MQAX 对于每一个对象都提供了相应的属性 (Property) 和方法 (Method)。在编程之前,消息的发送者和接收者需要约定报文格式。MQMessage 提供了一组 Read 和 Write 方法来对消息体进行操作,这一点与 MQ Java Base 编程有一点类似。

MQSession 表示与 MQAX 的一次会话,是应用中的根类,应该首先被创建。由 MQSession 的方法可以导出其它的对象 MQQueueManager、MQMessage、MQPutMessageOptions 和 MQGetMessageOptions。其中, MQQueueManager 表示队列管理器对象,可以 Connect, Disconnect, Commit, Backout, 也可以导出 MQQueue 对象。MQQueue 表示队列对象,可以 Open, Close, Put, Get, 其中 Put/Get 需要 MQMessage 作为参数。MQMessage 表示一条消息,由一组 Read 和 Write 的方法对消息体进行操作。MQDistributionList 和 MQDistributionListItem 与 MQI 中的概念相同,可以用来做消息分发。

下面以 VB 6.0 为例,我们来了解一下 ActiveX 的程序:

```
Dim Session As MQSession
Dim QueueManager As MQQueueManager
Dim Queue As MQQueue
Dim Message As MQMessage

On Error GoTo HandleError

Set Session = new MQSession
Set QueueManager = Session.AccessQueueManager("QM")
Set Queue = QueueManager.AccessQueue("Q", MQOO_INPUT_AS_Q_DEF)
Set Message = Session.AccessMessage()

Queue.Get Message
Text1.Text = Message.ReadString(Message.MessageLength)

QueueManager.Disconnect
Exit Sub

HandleError:
Label1.Caption = "CompletionCode = [" + CStr(Session.CompletionCode) + "],
ReasonCode = [" + CStr(Session.ReasonCode) + "], " + CStr(Session.ReasonName)
QueueManager.Disconnect
Exit Sub
```

例程通过调用 new MQSession 创建了一个 MQ 会话,以后所有的 MQ 操作都在该会话中完成。通过该会话再相继访问了队列管理器 QueueManager 和队列 Queue,队列管理器名为“QM”,队列为“Q”,且以读方式打开。接着,准备一条消息 Message,调用 Queue.Get 将队列中的消息读入 Message 对象中。最后,调用 QueueManager.Disconnect 与队列管理器断开连接,整个会话结束。

对 MQMessage 的消息头 MQMD 的操作时, MQAX 对 AccountingToken、CorrelationId、

GroupId 和 MessageId 提供了相应的另一套属性域 ,用于十六进制操作 :AccountingTokenHex、CorrelationIdHex、 GroupIdHex 和 MessageIdHex。事实上 ,对它们操作的效果是相同的。由于这些域中可能含有不可见或不可打印的字符 ,所以推荐使用后者比较保险。

MQAX 编程时使用到的常量基本上与 MQI 相同。对 VB 而言 ,可以参考 <InstallDir>\Tools\VB\include 中的文件。

16.1.1.2 数据转换

如果在 MQAX 中考虑对消息的数据转换 ,则需要对 MQMessage.Encoding 和 MQMessage.CharacterSet 进行设置。与 MQI 编程类似 ,MQMessage.Encoding 可以取值 :

- Encodings for Binary Integers
 - MQENC_INTEGER_UNDEFINED
 - MQENC_INTEGER_NORMAL
 - MQENC_INTEGER_REVERSED
- Encodings for Decimals
 - MQENC_DECIMAL_UNDEFINED
 - MQENC_DECIMAL_NORMAL
 - MQENC_DECIMAL_REVERSED
- Encodings for Floating-Point Numbers
 - MQENC_FLOAT_UNDEFINED
 - MQENC_FLOAT_IEEE_NORMAL
 - MQENC_FLOAT_IEEE_REVERSED
 - MQENC_FLOAT_S390

一旦设置了 Encoding ,则对该对象以后的 Int, Short, Long 等等元素操作时有效。

- | | |
|----------------|-----------------|
| ● ReadDecimal2 | ● WriteDecimal2 |
| ● ReadDecimal4 | ● WriteDecimal4 |
| ● ReadDouble | ● WriteDouble |
| ● ReadDouble4 | ● WriteDouble4 |
| ● ReadFloat | ● WriteFloat |
| ● ReadInt2 | ● WriteInt2 |
| ● ReadInt4 | ● WriteInt4 |
| ● ReadLong | ● WriteLong |
| ● ReadShort | ● WriteShort |
| ● ReadUInt2 | ● WriteUInt2 |

类似地 ,对 MQMessage.CharacterSet 设置可以对字符串操作起作用 :

- ReadString
- ReadNullTerminatedString
- WriteString
- WriteNullTerminatedString

此外，对于 MQMessage.MessageData 属性也有作用。

16.1.1.3 多线程

如同 MQ Java Base 编程一样，MQAX 提供的对象可以在多个线程中共享，而实际上，多个线程对于对象的操作在 MQAX 底层中是串行执行的。所以，对于一个队列的 Wait 方式 Get 操作会锁住其它线程的操作。

一个应用进程中只能有一个 MQSession 对象，这个对象贯穿始终，是这个进程中所有 MQAX 对象的根对象。

16.1.1.4 出错处理

MQAX 中的所有对象都含有 CompletionCode, ReasonCode, ReasonName 三个属性，另含有 ClearErrorCode 方法。CompletionCode 和 ReasonCode 与 MQI 编程中相同，表示 API 执行的完成码和原因码。ReasonName 是与 ReasonCode 对应的字串。

一般来说，对于对象的操作结果会反映到对象的 CompletionCode, ReasonCode, ReasonName 属性中，同时也会反映到 MQSession 的这三个属性。由于对象是可以在多个线程中共享的，所以对象的这三个属性有可能被不同线程的对象操作所覆盖，在使用上要注意。另外，MQSession.ExceptionThreshold（缺省值为 2）属性规定了一个阈值，如果 CompletionCode 大于或等于这个阈值，MQAX 会产生一个异常 (Exception, 32000)，在程序中可以用 On Error 语句指定对异常的处理，通常用 Error 函数可以取得出错的解释字串，格式为：

```
MQAX: CompletionCode=xxx, ReasonCode=xxx, ReasonName=xxx
```

在 MQAX 中修改 MQAX 对象的属性有时也会引起出错，比如修改 CompletionCode, ReasonCode, ReasonName。这时也会引起异常，同样用 On Error 可以截获。对于对象属性的操作：

如果成功，则 CompletionCode, ReasonCode 不改变。

如果出错，且 CompletionCode = Warning，则 ReasonCode 不变

如果出错，且 CompletionCode = Error，则 ReasonCode 为原因在码

16.1.2 编程参考

16.1.2.1 MQSession 对象

MQSession 对象表示与 WebSphere MQ 之间一次会话，它是程序中寿命最长的对象，通常程序以创建 MQSession 对象开始以释放该对象结束。程序中的其它对象都可以通过调用该对象的 AccessXXX 方法衍生出来，比如 AccessQueueManager、AccessMessage 等等。

MQSession 对象中有 CompletionCode 和 ReasonCode，分别表示完成码和原因码，由于应用程序可能是多线程共享 MQSession 对象，所以这两个返回码表示的是所有线程中最近的一次 MQ 操作的结果，ClearErrorCodes 用于将这两个返回码清零。原因码通常在出错判

断分支中起作用，为了提高程序的可读性，MQSession 提供了 ReasonName 属性，它与 ReasonCode 是一致的。另外 ReasonCodeName 方法可以将任意一个原因码转换成字符串。表 16-1 列出了 MQSession 对象的属性和方法。

表 16-1 MQSession 对象的属性和方法

属性		
CompletionCode	ReasonCode	
ExceptionThreshold	ReasonName	
方法		
AccessGetMessageOptions	AccessPutMessageOptions	ClearErrorCodes
AccessMessage	AccessQueueManager	ReasonCodeName

16.1.2.2 MQQueueManager 对象

MQQueueManager 对象对应于 WebSphere MQ 中的队列管理器对象，通过它可以连接到队列管理器或断开连接，可以打开队列，可以控制交易提交或回滚。该对象在应用程序中通常的寿命仅次于 MQSession，所有与队列管理器相关的操作都首先要通过该对象。

MQQueueManager 的 AccessQueue 和 AddDistributionList 使之可以衍生出队列和分发列表对象。Begin、Commit 和 Backout 方法可以用来控制交易，MQQueueManager 也是唯一的可以控制交易的对象。MQQueueManager 有众多的属性，它们与队列管理器的属性几乎一一对应。表 16-2 列出了 MQQueueManager 的对象属性和方法。

表 16-2 MQQueueManager 对象的属性和方法

属性		
AlternateUserId	ConnectionStatus	MaximumPriority
AuthorityEvent	ConnectOptions	MaximumUncommittedMessages
BeginOptions	DeadLetterQueueName	Name
ChannelAutoDefinition	DefaultTransmissionQueueName	ObjectHandle
ChannelAutoDefinitionEvent	Description	PerformanceEvent
ChannelAutoDefinitionExit	DistributionLists	Platform
CharacterSet	InhibitEvent	ReasonCode
CloseOptions	IsConnected	ReasonName
CommandInputQueueName	IsOpen	RemoteEvent
CommandLevel	LocalEvent	StartStopEvent
CompletionCode	MaximumHandles	SyncPointAvailability
ConnectionHandle	MaximumMessageLength	TriggerInterval
方法		
AccessQueue	Begin	Connect
AddDistributionList	ClearErrorCodes	Disconnect
Backout	Commit	

16.1.2.3 MQQueue 对象

MQQueue 对象对应于 WebSphere MQ 中的队列对象，它的特点是方法很少但属性众多。

对于队列而言，只有简单的打开、关闭、放消息、取消息四种操作，分别对应于 MQQueue 对象的 Open、Close、Put、Get 方法，当然 Put 和 Get 需要 MQMessage 对象作为参数。

MQQueue 属性与队列的属性几乎是一一对应的，所以对 MQQueue 属性的操作就如同对队列属性操作一样，非常方便。

表 16-3 MQQueue 对象的属性和方法

属性		
AlternateUserId	Description	QueueType
BackoutRequeueName	DynamicQueueName	ReasonCode
BackoutThreshold	HardenGetBackout	ReasonName
BaseQueueName	InhibitGet	RemoteQueueManagerName
CloseOptions	InhibitPut	RemoteQueueName
CompletionCode	InitiationQueueName	ResolvedQueueManagerName
ConnectionReference	IsOpen	ResolvedQueueName
CreationDateTime	MaximumDepth	RetentionInterval
CurrentDepth	MaximumMessageLength	Scope
DefaultInputOpenOption	MessageDeliverySequence	ServiceInterval
DefaultPersistence	Name	ServiceIntervalEvent
DefaultPriority	ObjectHandle	Shareability
DefinitionType	OpenInputCount	TransmissionQueueName
DepthHighEvent	OpenOptions	TriggerControl
DepthHighLimit	OpenOutputCount	TriggerData
DepthLowEvent	OpenStatus	TriggerDepth
DepthLowLimit	ProcessName	TriggerMessagePriority
DepthMaximumEvent	QueueManagerName	TriggerType
		Usage
方法		
ClearErrorCodes	Get	Put
Close	Open	

16.1.2.4 MQMessage 对象

MQMessage 对象对应于消息，所以它的大多数属性对应于消息的属性，即消息头 MQMD 的域，在编程中经常会在 Put 之前或 Get 之后设置或检查这些域。MQMessage 的方法是对消息内容的操作，分成读 (Read) 和写 (Write) 两类。基本上，这些方法都是将消息体看作数据流，从流的当前位置读写数据单元。流的当前位置是由 DataOffset 属性指定的，表示从消息体开始的字节偏移量，当前位置之后的数据长度由 DataLength 指定，消息的总长度由 MessageLength 指定。它们的关系为 $DataLength = MessageLength - DataOffset$ 。表 16-4 列出了 MQMessage 的对象属性和方法。

表 16-4 MQMessage 对象的属性和方法

属性		
CompletionCode	CorrelationIdHex	Offset
DataLength	Encoding	OriginalLength

DataOffset	Expiry	Persistence
MessageLength	Feedback	Priority
ReasonCode	Format	PutApplicationName
ReasonName	GroupId	PutApplicationType
AccountingToken	GroupIdHex	PutDateTime
AccountingTokenHex	MessageData	ReplyToQueueManagerName
ApplicationIdData	MessageFlags	ReplyToQueueName
ApplicationOriginData	MessageId	Report
BackoutCount	MessageIdHex	TotalMessageLength
CharacterSet	MessageSequenceNumber	UserId
CorrelationId	MessageType	

方法

ClearErrorCodes	ReadNullTerminatedString	WriteDouble4
ClearMessage	ReadShort	WriteFloat
Read	ReadString	WriteInt2
ReadBoolean	ReadUTF	WriteInt4
ReadByte	ReadUInt2	WriteLong
ReadDecimal2	ReadUnsignedByte	WriteNullTerminatedString
ReadDecimal4	ResizeBuffer	WriteShort
ReadDouble	Write	WriteUTF
ReadDouble4	WriteBoolean	WriteString
ReadFloat	WriteByte	WriteUInt2
ReadInt2	WriteDecimal2	WriteUnsignedByte
ReadInt4	WriteDecimal4	
ReadLong	WriteDouble	

16.1.2.5 MQPutMessageOptions 对象

MQPutMessageOptions 对象表示放消息选项，其中最重要的属性是 Options，即选项的组合。RecordFields 用于放消息到分发列表时同时指定多个队列。表 16-5 列出了 MQPutMessageOptions 的对象属性和方法。

表 16-5 MQPutMessageOptions 对象的属性和方法

属性

CompletionCode	ReasonName	ResolvedQueueName
Options	RecordFields	
ReasonCode	ResolvedQueueManagerName	

方法

ClearErrorCodes

16.1.2.6 MQGetMessageOptions 对象

MQGetMessageOptions 对象表示取消息选项，其中最重要的属性是 Options，即选项的组合。WaitInterval 表示取消息的等待时间长度，MatchOptions 表示取消息的匹配消息的选项。这些属性的用法与 MQI 编程模式中是类似的。

表 16-6 MQGetMessageOptions 对象的属性和方法

属性		
CompletionCode	ReasonCode	WaitInterval
MatchOptions	ReasonName	
Options	ResolvedQueueName	
方法		
ClearErrorCodes		

16.1.2.7 MQDistributionList 对象

MQDistributionList 对象与 WebSphere MQ 中的分发列表对象对应，可以把它简单地看作是一组目标队列，把消息放入 MQDistributionList 对象就相当于放入这一组队列中。分发列表是由多个列表项 (Item) 组成的，在 MQDistributionList 中可以用 AddDistributionListItem 将列表项加进来，用 Put 将消息放入列表中。列表项在列表中是链式存放的，由 FirstDistributionListItem 标识第一项。表 16-7 列出了 MQDistributionList 的对象属性和方法。

表 16-7 MQDistributionList 对象的属性和方法

属性		
AlternateUserId	ConnectionReference	OpenOptions
CloseOptions	FirstDistributionListItem	ReasonCode
CompletionCode	IsOpen	ReasonName
方法		
AddDistributionListItem	Close	Put
ClearErrorCodes	Open	

16.1.2.8 MQDistributionListItem 对象

MQDistributionListItem 组成了 MQDistributionList，可以把它简单地看作是一个目标队列对象，所以可以设置各自的队列管理器和队列名。另外，由于消息散列到不同的目标队列后可以具有相同的 ID，也可以具有各自不同的 ID，所以 AccountingToken、MessageId、CorrelationId、GroupId 及其对应的十六进制属性可以用来标识相应的消息。

列表项在列表中是链式存放的，由列表项的双向指针串起来。编程时用 PreviousDistributionListItem 和 NextDistributionListItem 来检索前一项和后一项。

表 16-8 MQDistributionListItem 对象的属性和方法

属性		
AccountingToken	Feedback	PreviousDistributionListItem
AccountingTokenHex	GroupId	QueueManagerName
CompletionCode	GroupIdHex	QueueName
CorrelationId	MessageId	ReasonCode
CorrelationIdHex	MessageIdHex	ReasonName
DistributionList	NextDistributionListItem	
方法		

ClearErrorCodes

16.1.3 跟踪信息 (Trace)

MQAX 的跟踪信息 (Trace) 可以通过环境变量进行控制 ,共有三个环境变量 ,如表 16-9。

表 16-9 MQAX 中 Trace 相关的环境变量

环境变量	解释
OMQ_TRACE	Trace 开关。只要这个变量不为空，表示打开状态
OMQ_TRACE_PATH	指定 Trace 文件的目录，文件名缺省为 OMQnnnnn.trc
OMQ_TRACE_LEVEL	指定 Trace Level，1-9，9 表示 Trace 信息最细致

16.2 MQAI

与 MQI 实现的的 MQAI 编程相同 (参见“ PCF & AI 编程 ”), WebSphere MQ 引入了 Bag 的概念以简化直接对 PCF 消息操作的复杂度。WebSphere MQ for Windows 的第二个 ActiveX 组件 IBM MQSeries MQAI COM Library 中提供了 MQBag 类及相关的操作方法。表 16-10。

表 16-10 MQAI 中 MQBag 对象的属性和方法

属性		
Item	Count	Options
方法		
Add	FromMessage	ToMessage
AddInquiry	ItemType	Truncate
Clear	Remove	
Execute	Selector	

编程方法可参考 MQAI 一章的相关内容。

16.3 ADSI

Windows 操作系统提供了 Active Directory Service Interfaces (ADSI) 框架，使得不同的应用可以使用相同的界面调用指定的服务。任何支持 COM 界面的编程工具开发的应用都可以利用 ADSI 很方便地找到 WebSphere MQ 服务并使用它。当然，相应的队列管理器必须在被使用之前处于运行状态，命令服务器启动，监听器打开。在 ADSI 中，任何服务都有一个命名空间，对 MQ 来说，如图 16-1。

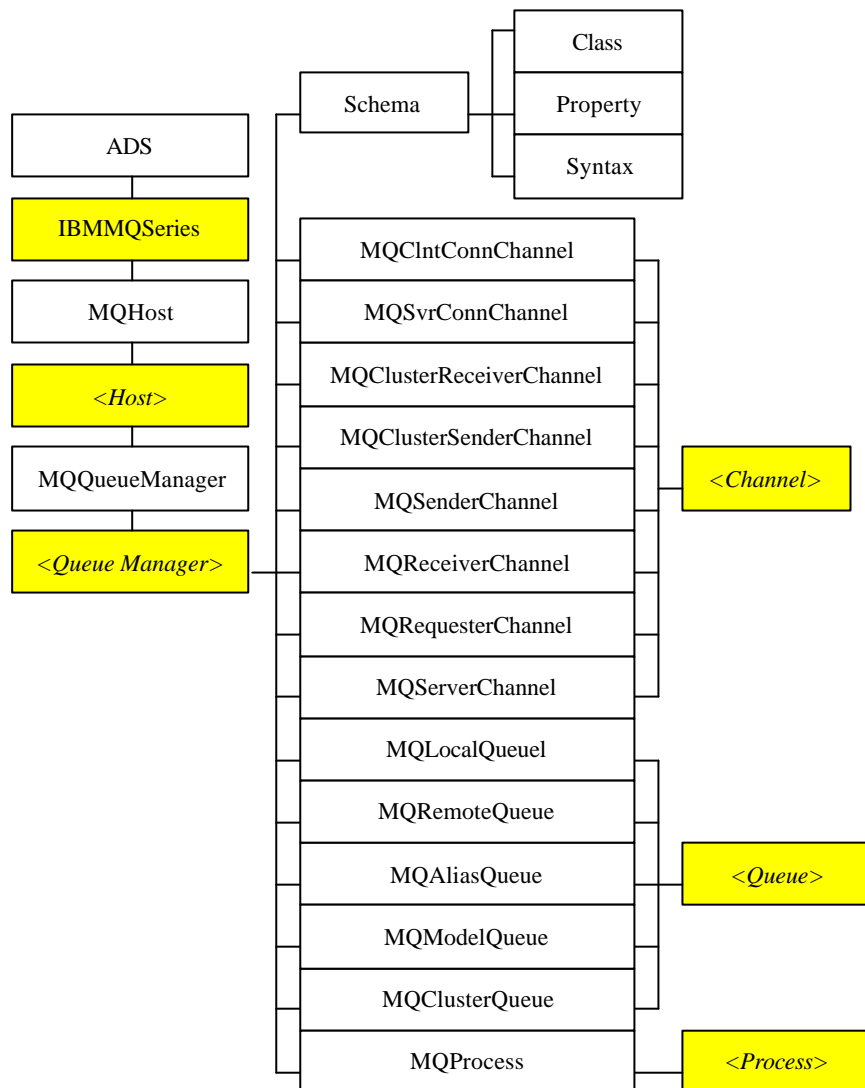


图 16-1 ADSI 中对象的命名空间

对于这样的层次结构，我们也可以用 URL 指定其中的某一个对象，比如：

WebSphere MQ: //MQHost/host1/MQQueueManager/qm1/MQLocalQueue/q1

在应用程序中用 `ADsGetObject ()` 和 `ADsOpenObject ()` 来操作对象。相关内容，可以参考 Microsoft ADSI 编程规范。

第 17 章 AMI 编程

Application Messaging Interface (AMI) 是 WebSphere MQ 的另一种编程模式。与 MQI 不同，AMI 编程不需要知道 WebSphere MQ 底层的对象细节，也不直接与这些对象打交道。取而代之的是 Service 和 Policy，由管理工具将其配置后使用。

AMI 编程环境需要有 WebSphere MQ SupportPac MA0F 的支持，下载网址：

<http://www-3.ibm.com/software/integration/support/supportpacs/>。目前，支持的操作系统平台有 AIX、HP-UX、SUN Solaris、Windows、OS/400、z/OS 等，其中 UNIX 平台上的 AMI 工作原理都类似。AMI 支持的编程语言有 C、C++、COBOL、Java 等。其中面向过程的 C 和 COBOL 可以相互参考，面向对象的 C++ 和 Java 可以相互参考。以下会选取有代表性的 Windows 及 AIX (代表 UNIX) 操作系统介绍安装过程，选取 C 和 Java 介绍编程方式。

17.1 安装

AMI 扩展构件的安装十分简单方便，我们以 Windows 和 AIX 为例介绍安装过程，其它 UNIX 的安装过程与 AIX 类似。

17.1.1 Windows

Windows 中的 AMI 环境安装特别简单。下载 SupportPac MA0F for Windows 文件 ma0f_nt.zip，这是一个压缩文件。将其展开至临时目录下，双击 setup，安装完毕后删除临时目录即可。

17.1.2 AIX

AIX 中的 AMI 环境安装也不难。下载 SupportPac MA0F for AIX 文件 ma0f_ax.tar.Z，这是一个 UNIX compress 压缩文件。以 root 身份登录，将文件放入 /tmp 中。执行以下命令将文件解压缩并展开，完毕后删除原来的压缩文件即可。

```
uncompress -fv /tmp/ma0f_ax.tar.Z
tar -xvf /tmp/ma0f_ax.tar
rm /tmp/ma0f_ax.tar
```

17.2 概念与配置

17.2.1 概念

在 AMI 中有三个基本概念：消息 (Message)、服务 (Service)、策略 (Policy)。在一次应用程序间的数据传递中，消息指的是传递的数据内容，服务指的是传递到哪里，即目的地，而策略指的是消息被如何传递。在 AMI 的应用程序中，这三个概念描述了应用对消息传送的基本要求。

AMI 中的消息包括消息属性和消息数据两部分。与 MQI 类似，属性中有 MessageID、CorrelID、Format、Topic 等等，描述了消息自身的一些特性，这些属性可能在服务中被特定的策略所利用，从而产生特定的功能。消息数据则是一段内存空间，存放着消息的内容。

服务指明了消息的传送方式和目的，共有以下四种类型。

1. 发送方 (Sender) 和接收方 (Receiver) 之间的单向传送
2. 分发列表 (Distribution List) 设定的一方发送多方接收方式

3. 发布方 (Publisher) 含一个发送方 (Sender) 来向转发代理 (Pub/Sub Broker) 登记发布及发送消息
4. 订阅方 (Subscriber) 含一个发送方 (Sender) 来向转发代理 (Pub/Sub Broker) 登记订阅, 另含一个接收方 (Receiver) 来接收来自转发代理的消息

策略规定了消息在传递过程的行为选项。比如, 消息被处理的优先级, 是否参与交易, 发送和接收时的设置, 在发布及订阅时消息是否被保留(Retained) 等等。所有这些最终都演变成消息在传递过程中的功能。

在具体编程的过程中, 还会碰到会话 (Session)、连接 (Connection) 和策略处理 (Policy Handler) 等概念。Session 是一切 AMI 操作的基础, 一般说来, 应用程序应该首先创建 Session, 以后的一切操作都是在 Session 中进行的。Session 可以将一段连续的操作提交或回滚。Connection 实际上就是与底层 WebSphere MQ 环境的连接, 它内置于 Session 中, 通常应用程序不需要关心 Connection。Policy Handler 实际上指明了一个函数库和入口参数, 在适当的时候会自动调用, 其概念有点像用户出口。

17.2.2 配置

服务 (Service) 和策略 (Policy) 都可以通过 AMI 管理工具来进行配置, 配置的结果存放在 xml 文件中。AMI 管理工具是一个 Java 应用程序, 在<InstallDir>\amt\AMITool 目录下, 操作界面如下图。

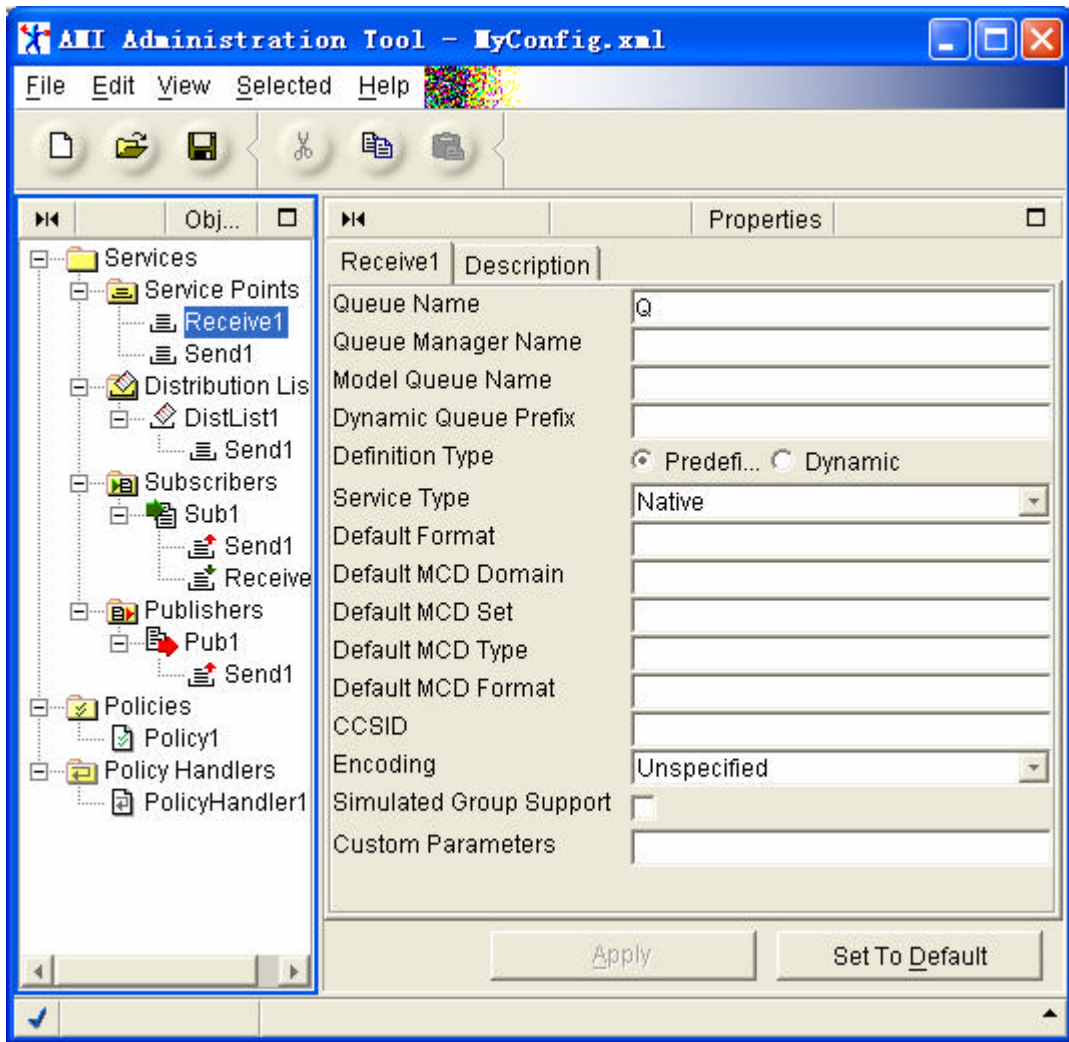


图 17-1 AMI 管理工具

AMI 管理工具配置的 xml 文件可以作为 Repository 供应用程序使用，应用程序可以直接引用 Repository 中已经定义好的对象，这些对象由于在配置时有大量的属性设置，这使得应用程序可以比较简洁，同时也可以将对象配置和对象编程分开，增加灵活性。当然，应用程序也可以不使用 Repository，而在程序中创建这些对象。

17.3 C 编程

在 C 的编程中提供了 Object Level 和 High Level 两个层次的 API。其中，Object Level 层次较低，适合于精通对 WebSphere MQ 编程的人员，它可以提供最大的灵活性，使编程人员管理底层的调用。High Level 层次较高，适合于一般的编程人员，它的编程相对简单，代码简洁明了。在编程时，两种 API 可以混用。

17.3.1 Object Level

Object Level API 不但提供了 High Level API 的全部功能，而且通过大量的底层 API 可以控制 AMI 的细微操作。Object Level API 是针对 AMI 对象设立的，API 名通常为 amXXXXYY，这里 XXX 指的是 AMI 对象缩写，YYY 是针对该对象的操作。AMI 的对象

有：

- Session
- Message
- Sender
- Receiver
- Publisher
- Subscriber
- Distribution List
- Policy

编程的过程大致如下：

1. 创建 Session，用 amSesCreate ()
2. 创建 Policy，用 amSesCreatePolicy ()
3. 创建 Service，根据需要用 amSesCreateXXX ()
4. 创建 Message，用 amSesCreateMessage ()
5. 打开 Session，实际上是将 Session 与 Policy 联系起来。用 amSesOpen ()
6. 打开 Service，实际上是将 Service 与 Policy 联系起来。用 amXXXOpen ()
7. 在 Service 中操作 Message。比如 amSndSend ()，amRcvReceive () 等等
8. 清理环境，关闭打开的对象，删除创建的对象。

具体的 API 使用方法，请参见例程及编程手册。

例程：

```
#include "ami_public.h"

#define SAMPLE_SESSION_NAME      (AMSTR)"AMT.SAMPLE.SESSIO"
#define SAMPLE_POLICY_NAME      (AMSTR)"AMT.SAMPLE.POLICY"
#define SAMPLE_SENDER_NAME      (AMSTR)"AMT.SAMPLE.SERVICE"

void main (void)
{
    AMHSES      amhses;                // Session Handle
    AMHPOL      amhpol;                // Policy Handle
    AMHSND      amhsnd;                // Sender Handle
    AMLONG      amlCompletionCode;      // Completion Code
    AMLONG      amlReasonCode;          // Reason Code
    AMLONG      amlMessageLength;       // Message Length
    AMBYTE      ambMessageBuffer [1024 * 10]; // Message Buffer

    amhses = amSesCreate (
        SAMPLE_SESSION_NAME,            // Session Name
        & amlCompletionCode,              // Completion Code
        & amlReasonCode);                // Reason Code

    amhpol = amSesCreatePolicy (
```

```

        amhses,                                // Session Handle
        SAMPLE_POLICY_NAME,                    // Policy Name
        & amlCompletionCode,                    // Completion Code
        & amlReasonCode);                      // Reason Code

    amhsnd = amSesCreateSender (
        amhses,                                // Session Handle
        SAMPLE_SENDER_NAME,                    // Sender Name
        & amlCompletionCode,                    // Completion Code
        & amlReasonCode);                      // Reason Code

    amSesOpen (
        amhses,                                // Session Handle
        amhpol,                                // Policy Handle
        & amlCompletionCode,                    // Completion Code
        & amlReasonCode);                      // Reason Code

    amSndOpen (
        amhsnd,                                // Sender Handle
        amhpol,                                // Policy Handle
        & amlCompletionCode,                    // Completion Code
        & amlReasonCode);                      // Reason Code

    for ( ; ; )
    {
        if (fgets (ambMessageBuffer, sizeof (ambMessageBuffer), stdin) == NULL)
            break;

        amlMessageLength = strlen (ambMessageBuffer) - 1;
        ambMessageBuffer [amlMessageLength] = '\0';
        if (amlMessageLength == 0)
            break;

        amSndSend (
            amhsnd,                                // Sender Handle
            amhpol,                                // Policy Handle
            AMH_NULL_HANDLE,                      // Response Receiver Handle
            AMH_NULL_HANDLE,                      // Received Message Handle
            amlMessageLength,                      // Message Length
            ambMessageBuffer,                      // Message Buffer
            AMH_NULL_HANDLE,                      // Sender Message Handle
            & amlCompletionCode,                    // Completion Code
            & amlReasonCode);                      // Reason Code
    }

```

```

amSesDelete (
    & amhses,                                // Session Handle
    & amlCompletionCode,                      // Completion Code
    & amlReasonCode);                        // Reason Code
}

```

17.3.1.1 Session 类 API

Session 是程序中“寿命”最长的操作对象，表示与 MQ 环境的一次会话。它必须最早创建，最晚删除，几乎所有的其它对象都需要通过它来创建，如 Message、Sender、Receiver、Publisher、Subscriber、Distribution List 和 Policy 等等。所以，Session 应该是一个需要被最早初始化的对象。此外，会话中的动作是可以作为交易提交或回滚的，Session 也用以支持事务处理。表 17-1 列出了 Session 类 API 及其分类。

表 17-1 Object Level 中 Session 对象的相关 API 及其分类

Session Management			
amSesCreate	amSesOpen	amSesClose	amSesDelete
Create Object			
amSesCreateMessage	amSesCreateSender	amSesCreateReceiver	amSesCreateDistList
amSesCreatePublisher	amSesCreateSubscriber	amSesCreatePolicy	amMsgSetCorrelId
amMsgSetElementCCSID	amMsgSetEncoding	amMsgSetFormat	amMsgSetGroupStatus
amMsgSetReportCode	amMsgSetType	amMsgReset	
Get Object Handle			
amSesGetMessageHandle	amSesGetSenderHandle	amSesGetReceiverHandle	amSesGetDistListHandle
amSesGetPublisherHandle	amSesGetSubscriberHandle	amSesGetPolicyHandle	
Delete Object			
amSesDeleteMessage	amSesDeleteSender	amSesDeleteReceiver	amSesDeleteDistList
amSesDeletePublisher	amSesDeleteSubscriber	amSesDeletePolicy	
Transaction			
amSesBegin	amSesCommit	amSesRollback	
Error Handling			
amSesClearErrorCodes	amSesGetLastError		

17.3.1.2 Message 类 API

Message 对象即消息，包含消息头结构和消息体数据，可以有一系列的 API 对消息头的各个域或者消息内容进行修改和设置。Message 对象是唯一的不在配置界面中事先定义的对象，在编程中具有较高的灵活性，同时也有最丰富的 API。表 7-2 列出了 Message 类 API 及其分类。

表 17-2 Object Level 中 Message 对象的相关 API 及其分类

Get & Set

amMsgGetCCSID	amMsgGetCorrelId	amMsgGetElementCCSID	amMsgGetEncoding
amMsgGetFormat	amMsgGetGroupStatus	amMsgGetMsgId	amMsgGetName
amMsgGetReportCode	amMsgGetType	amMsgSetCCSID	
Read & Write Data			
amMsgGetDataLength	amMsgGetDataOffset	amMsgSetDataOffset	amMsgReadBytes
amMsgWriteBytes			
Pub & Sub			
amMsgAddTopic	amMsgDeleteTopic	amMsgGetTopic	amMsgGetTopicCount
amMsgAddFilter	amMsgDeleteFilter	amMsgGetFilter	amMsgGetFilterCount
amMsgAddElement	amMsgDeleteElement	amMsgGetElement	amMsgGetElementCount
amMsgDeleteNamedElement	amMsgGetNamedElement	amMsgGetNamedElementCount	
Error Handling			
amMsgClearErrorCodes	amMsgGetLastError		

17.3.1.3 Sender 和 Receiver 类 API

Sender 对象代表将消息发送到目的地的服务，而 Receiver 对象代表接收消息的服务。无论是 Sender 还是 Receiver ,都内含了 MQOD 结构 ,即消息对应的队列。Sender 和 Receiver 类 API 可以提供对消息或文件的发送和接收服务。由于 Sender 和 Receiver 对象参数大多数在配置界面中设置，API 只能设置队列名。表 17-3 和表 17-4 列出了 Sender 和 Receiver 类 API 及其分类。

表 17-3 Object Level 中 Sender 对象的相关 API 及其分类

Open & Close			
amSndOpen	amSndClose		
Send			
amSndSend	amSndSendFile		
Get			
amSndGetCCSID	amSndGetEncoding	amSndGetName	
Error Handling			
amSndClearErrorCodes	amSndGetLastError		

表 17-4 Object Level 中 Receiver 对象的相关 API 及其分类

Open & Close			
amRcvOpen	amRcvClose		
Receive & Browse			
amRcvReceive	amRcvReceiveFile	amRcvBrowse	amRcvBrowseSelect
Get & Set			
amRcvGetDefnType	amRcvGetName	amRcvGetQueueName	amRcvSetQueueName
Error Handling			
amRcvClearErrorCodes	amRcvGetLastError		

17.3.1.4 Publisher 和 Subscriber 类 API

Publisher 对象包含 Sender 服务，其中目的地是一个发布/预订代理。而 Subscriber 对象包含 Sender 服务，用来将预订和取消预订消息发送至 Broker，同时也包括一个 Receiver 服务，接收来自 Broker 的发布内容。这两类对象的 API 可以对消息进行订阅或分布，也可以获取消息的某些属性，比如编码及字符集等信息。表 17-5 和表 17-6 分别列出了 Publisher 和 Subscriber 类 API 及其分类。

表 17-5 Object Level 中 Publisher 对象的相关 API 及其分类

Open & Close		
amPubOpen	amPubClose	
Publish		
amPubPublish		
Get		
amPubGetCCSID	amPubGetEncoding	amPubGetName
Error Handling		
amPubClearErrorCodes	amPubGetLastError	

表 17-6 Object Level 中 Subscriber 对象的相关 API 及其分类

Open & Close		
amSubOpen	amSubClose	
Subscribe		
amSubSubscribe	amSubUnsubscribe	amSubReceive
Get & Set		
amSubGetCCSID	amSubGetDefnType	amSubGetEncoding
amSubGetQueueName	amSubSetQueueName	amSubGetName
Error Handling		
amSubClearErrorCodes	amSubGetLastError	

17.3.1.5 Distribution List 类 API

Distribution List 对象本身包含 Sender 服务列表，实际上是目的地的分发列表。对分发列表的操作就如同对多个 Sender 对象同时操作。Distribution List 类 API 就可以提供到消息和文件的分发功能。表 17-7 列出了 Distribution List 类 API 及其分类。

表 17-7 Object Level 中 Distribution List 对象的相关 API 及其分类

Open & Close		
amDstOpen	amDstClose	
Send		
amDstSend	amDstSendFile	
Get		
amDstGetName	amDstGetSenderCount	amDstGetSenderHandle
Error Handling		

amDstClearErrorCodes

amDstGetLastError

17.3.1.6 Policy 类 API

Policy 定义了对消息的特性和处理方式，比如消息发送时设定的优先级、持久性、超时、报告，消息接收时等待的时间，以及其是否应包括在交易中完成等等。由于 Policy 参数大多数在配置界面中设置，Policy 类的 API 只能设置消息接收时等待的时间。表 17-8 列出了 Policy 类 API 及其分类。

表 17-8 Object Level 中 Policy 对象的相关 API 及其分类

Get & Set		
amPolGetName	amPolGetWaitTime	amPolSetWaitTime
Error Handling		
amPolClearErrorCodes	amPolGetLastError	

17.3.2 High Level

High Level 的 API 不多，但往往更简洁实用。High Level 与 Object Level 的 API 是可以混用的，比如 High Level API 返回的 Session Handle 在 Object Level API 中一样使用。

17.3.2.1 amInitialize, amTerminate

首先，应用程序应该用 amInitialize () 创建一个会话 (Session)，以后所有的操作都在 Session 上进行，这些操作可以提交或回滚。在创建的时候可以给会话起一个名字，并指定会话策略。与创建会话相对的是 amTerminate ()，结束该会话。

例：

```
amhses = amInitialize (  
    SAMPLE_SESSION_NAME,          // Session Name  
    SAMPLE_POLICY_NAME,           // Policy Name  
    & amlCompletionCode,           // Completion Code  
    & amlReasonCode);             // Reason Code  
amTerminate (  
    & amhses,                      // Session Handle  
    SAMPLE_POLICY_NAME,           // Policy Name  
    & amlCompletionCode,           // Completion Code  
    & amlReasonCode);             // Reason Code
```

17.3.2.2 amSendMsg, amReceiveMsg, amBrowseMsg

对于点对点的单向通信方式，应用程序可以用 amSendMsg () 和 amReceiveMsg () 对消息进行发送和接收。API 调用中可以指定服务名 (Service Name) 和策略名 (Policy Name)，而它们可以是 Repository 中事先定义好的对象。amBrowseMsg () 是对消息的浏览，并不会将消息取走。

例：

```
amSendMsg (  
    amhses, // Session Handle  
    SAMPLE_SENDER_NAME, // Sender Name  
    SAMPLE_POLICY_NAME, // Policy Name  
    amlMessageLength, // Message Length  
    ambMessageBuffer, // Message Buffer  
    SAMPLE_MESSAGE_NAME, // Message Name  
    & amlCompletionCode, // Completion Code  
    & amlReasonCode); // Reason Code  
amReceiveMsg (  
    amhses, // Session Handle  
    SAMPLE_RESPONSE_NAME, // Receiver Name  
    SAMPLE_POLICY_NAME, // Policy Name  
    SAMPLE_PUB_MESSAGE_NAME, // Selection Name  
    sizeof (ambMessageBuffer), // Buffer Length  
    & amlMessageLength, // Message Length  
    ambMessageBuffer, // Message Buffer  
    NULL, // Message Name  
    & amlCompletionCode, // Completion Code  
    & amlReasonCode); // Reason Code
```

17.3.2.3 amSendFile, amReceiveFile

作为对 amSendMsg () 和 amReceiveMsg () 的功能延伸 ,AMI 提供了对文件的发送和接收：amSendFile () 和 amReceiveFile ()。

例：

```
amSendFile (  
    amhses, // Session Handle  
    SAMPLE_SENDER_NAME, // Sender Name  
    SAMPLE_POLICY_NAME, // Policy Name  
    0, // Options, 保留域，必须为 0  
    0, // Directory Length, 保留域，必须为 0  
    NULL, // Directory, 保留域，必须为 NULL  
    AMLEN_NULL_TERM, // File Name Length  
    pamcFileName, // File Name  
    NULL, // Message Name  
    & amlCompletionCode, // Completion Code  
    & amlReasonCode); // Reason Code  
amReceiveFile (  
    amhses, // Session Handle  
    SAMPLE_RECEIVER_NAME, // Receiver Name
```

```

SAMPLE_POLICY_NAME,          // Policy Name
0,                            // Options,          保留域，必须为 0
NULL,                        // Selection Message Name
0,                            // Directory Length,    保留域，必须为 0
NULL,                        // Directory,           保留域，必须为 NULL
AMLEN_NULL_TERM,            // File Name Length
pamcFileName,                // File Name
NULL,                        // Message Name
& amlCompletionCode,         // Completion Code
& amlReasonCode);           // Reason Code

```

17.3.2.4 amSendRequest, amReceiveRequest, amSendResponse

除了点对点的单向通信方式，AMI 还有一种 Request/Response 的应答式通信方式。这就相当于请求消息中带有 ReplyToQ，应答方可以配合着将应答消息发往该队列。在编程时，请求方用 amSendRequest() 将请求消息送出，用 amReceiveMsg() 接收应答消息。应答方用 amReceiveRequest() 接收请求消息，用 amSendResponse() 将应答消息送出。

例：

```

amSendRequest (
    amhses,                    // Session Handle
    SAMPLE_SENDER_NAME,       // Sender Name
    SAMPLE_POLICY_NAME,       // Policy Name
    SAMPLE_RECEIVER_NAME,     // Receiver Name
    amlMessageLength,         // Message Length
    ambMessageBuffer,         // Message Buffer
    SAMPLE_SEND_MESSAGE_NAME, // Message Name
    & amlCompletionCode,       // Completion Code
    & amlReasonCode);         // Reason Code

amReceiveRequest (
    amhses,                    // Session Handle
    SAMPLE_RECEIVER_NAME,     // Receiver Name
    SAMPLE_POLICY_NAME,       // Policy Name
    sizeof (ambMessageBuffer), // Buffer Length
    & amlMessageLength,        // Message Length
    ambMessageBuffer,         // Message Buffer
    SAMPLE_RECEIVE_MESSAGE_NAME, // Message Name
    SAMPLE_SENDER_NAME,       // Sender Name
    & amlCompletionCode,       // Completion Code
    & amlReasonCode);         // Reason Code

amSendResponse (
    amhses,                    // Session Handle
    SAMPLE_SENDER_NAME,       // Sender Name
    SAMPLE_POLICY_NAME,       // Policy Name

```

```

SAMPLE_RECEIVE_MESSAGE_NAME,    // Receiver Message Name
amlMessageLength,               // Message Length
ambMessageBuffer,               // Message Buffer
SAMPLE_SEND_MESSAGE_NAME,       // Sender Message Name
& amlCompletionCode,            // Completion Code
& amlReasonCode);               // Reason Code

```

17.3.2.5 amPublish, amSubscribe, amUnsubscribe, amReceivePublication

发布及订阅 (Pub/Sub) 又是 AMI 的另一种程序间通信方式。它需要发布方用 amPublish () 将消息发布出来，发布的信息都属于某一个主题 (Topic)，分发代理 (Broker) 可以根据 Topic 进行消息分发。订阅方要首先用 amSubscribe () 进行对该主题 (Topic) 进行订阅，一旦订阅成功，则可以用 amReceivePublication () 源源不断地接收该主题的消息。当订阅不再需要的时候，可以用 amUnsubscribe () 退订。

例：

```

amPublish (
    amhses,                // Session Handle
    SAMPLE_PUBLISHER_NAME,  // Publisher Name
    SAMPLE_POLICY_NAME,     // Policy Name
    SAMPLE_RESPONSE_NAME,   // Response Receiver Name
    strlen (SAMPLE_TOPIC),  // Topic Length
    SAMPLE_TOPIC,           // Topic Buffer
    amlMessageLength,       // Message Length
    ambMessageBuffer,       // Message Buffer
    SAMPLE_PUB_MESSAGE_NAME, // Message Name
    & amlCompletionCode,     // Completion Code
    & amlReasonCode);        // Reason Code
amSubscribe (
    amhses,                // Session Handle
    SAMPLE_SUBSCRIBER_NAME, // Subscriber Name
    SAMPLE_POLICY_NAME,     // Policy Name
    NULL,                  // Response Service
    strlen (SAMPLE_TOPIC),  // Topic Length
    SAMPLE_TOPIC,           // Topic Buffer
    0L,                    // Filter Length
    NULL,                  // Filter Buffer
    SAMPLE_SUB_MESSAGE_NAME, // Message Name
    & amlCompletionCode,     // Completion Code
    & amlReasonCode);        // Reason Code
amReceivePublication (
    amhses,                // Session Handle
    SAMPLE_SUBSCRIBER_NAME, // Subscriber Name

```

```

    SAMPLE_POLICY_NAME,          // Policy Name
    SAMPLE_SUB_MESSAGE_NAME,     // Selection Message Name
    MQ_CORREL_ID_LENGTH,         // Topic Buffer Length
    sizeof (ambMessageBuffer),   // Buffer Length
    NULL,                        // Topic Count
    NULL,                        // First Topic Length
    NULL,                        // First Topic Buffer
    & amlMessageLength,          // Message Length
    ambMessageBuffer,            // Message Buffer
    SAMPLE_PUB_MESSAGE_NAME,     // Message Name
    & amlCompletionCode,          // Completion Code
    & amlReasonCode);            // Reason Code
amUnsubscribe (
    amhses,                      // Session Name
    SAMPLE_SUBSCRIBER_NAME,      // Subscriber Name
    SAMPLE_POLICY_NAME,          // Policy Name
    NULL,                        // Response Service
    strlen (SAMPLE_TOPIC),       // Topic Length
    SAMPLE_TOPIC,                // Topic Buffer
    0L,                          // Filter Length
    NULL,                        // Filter Buffer
    SAMPLE_SUB_MESSAGE_NAME,     // Message Name
    & amlCompletionCode,          // Completion Code
    & amlReasonCode);            // Reason Code

```

17.3.2.6 amBegin, amCommit, amBackout

在 Session 上的一系列动作可以提交或回滚。amBegin () 标志交易的起始点 ,amCommit () 提交交易 , amBackout () 回滚交易。

例：

```

amBegin (
    amhses,                      // Session Name
    SAMPLE_POLICY_NAME,          // Policy Name
    & amlCompletionCode,          // Completion Code
    & amlReasonCode);            // Reason Code
amCommit (
    amhses,                      // Session Name
    SAMPLE_POLICY_NAME,          // Policy Name
    & amlCompletionCode,          // Completion Code
    & amlReasonCode);            // Reason Code
amBackout (
    amhses,                      // Session Name
    SAMPLE_POLICY_NAME,          // Policy Name

```

```
& amlCompletionCode,          // Completion Code
& amlReasonCode);             // Reason Code
```

17.4 Java 编程

AMI Java 编程接口与 C 的 Object Level 类似，也能对 Session、Message、Sender、Receiver、DistributionList、Publisher、Subscriber 和 Policy 等对象进行比较精细的控制。事实上，这些类与方法的名称与 Object Level API 几乎一致，读者很容易找到两者的对应关系。

AMI Java 编程需要引入 Java 包 com.ibm.mq.amt，对应的库文件为 com.ibm.mq.amt.jar。在编程时需要将其引入，例：

```
import com.ibm.mq.amt.*;
```

同时，要确保 JNI 库文件在系统路径中。如表 17-9

表 17-9 JNI 库文件的路径设置

操作系统	JNI 库文件	路径设置
AIX	libamtJava.so	export LIBPATH=\$LIBPATH:/usr/mqm/lib:
HP-UX	libamtJava.sl	export SHLIB_PATH=\$SHLIB_PATH:/opt/mqm/lib:
Solaris	libamtJava.so	export LD_LIBRARY_PATH=\$LD_LIBRARY_PATH:/opt/mqm/lib:
Windows	amtJava.dll	SET PATH=%PATH%;C:\MQSeries\bin;

AMI Java 的对象与 AMI C 的 Object Level 类似，所不同的是 AMI Java 多一个对象类 AmSessionFactory，它表示 WebSphere MQ 运行环境。AMI Java 的对象有：

- AmSessionFactory
- AmSession
- AmMessage
- AmSender
- AmReceiver
- AmPublisher
- AmSubscriber
- AmDistributionList
- AmPolicy

编程的过程大致如下：

1. 创建 SessionFactory，用 new AmSessionFactory ()
2. 创建 Session，用 sessionFactory.createSession ()
3. 创建 Policy，用 session.createPolicy ()
4. 创建 Service，根据需要用 session.createXXX ()
5. 创建 Message，用 session.createMessage ()
6. 打开 Session，实际上是将 Session 与 Policy 联系起来。用 session.open ()
7. 打开 Service，实际上是将 Service 与 Policy 联系起来。用 XXX.open ()
8. 在 Service 中操作 Message。比如 sender.Send ()，receiver.receive () 等等
9. 清理环境，关闭打开的对象，删除创建的对象。

具体的 API 使用方法，请参见例程及编程手册。

第 18 章 PCF & AI 编程

可编程命令格式 (Programmable Command Formats, 简称 PCF) 是对 WebSphere MQ 管理编程的一种基本方式。它规定了管理请求及应答消息的格式和使用方法,通过对队列管理器的命令队列发送管理消息来操纵队列管理器。PCF 是管理编程的基础,但由于 PCF 格式比较复杂,操作稍显困难。

管理界面 (Administration Interface, 简称 AI) 是建立在 PCF 基础之上的一种简化的管理编程方式。它将管理请求应答消息封装成包 (Bag) 的形成,在请求包中可以添加各种对象管理选项,包最后会被自动翻译成 PCF 消息并发送到队列管理器的命令队列中,应答的 PCF 消息又会翻译回包的形式。应用程序通过对包这样一种管理界面的操作,就避免了接触 PCF 的复杂格式。AI 本质上就是 PCF。

18.1 PCF 编程

18.1.1 消息流程

1. 启动队列管理器的命令服务进程 (Command Server)
2. 应用程序组织 PCF 消息,在消息中要指明应答队列。用 MQPUT 将消息放入队列管理器的命令队列中 (SYSTEM.ADMIN.COMMAND.QUEUE),该消息会被命令服务进程读走并执行,执行结果放入应答队列中。
3. 用 MQGET 将应答消息从应答队列中取出,根据 PCF 格式解析应答消息,得知命令是否执行成功及相关结果。图 18-1。

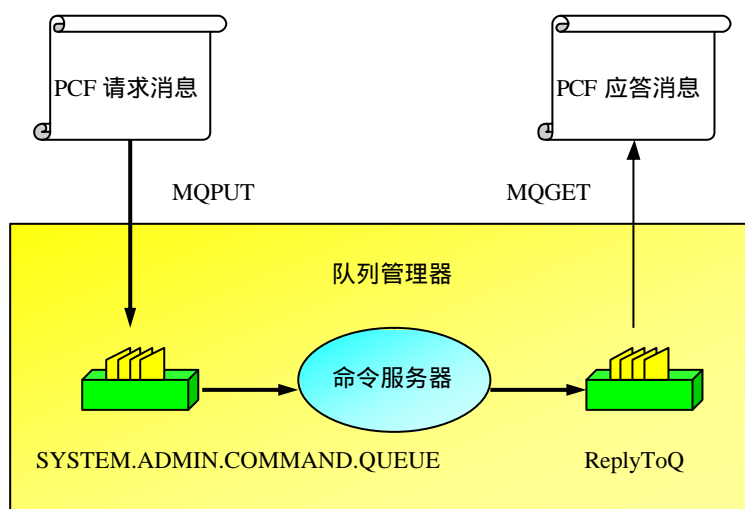


图 18-1 PCF 消息流程

18.1.2 消息格式

类似于普通消息,PCF 消息也是分消息头 (MQMD) 和消息体两部分。其中消息头中的 MsgType 在消息请求时设置 MQMT_REQUEST,在消息应答时会由 WebSphere MQ 设置成 MQMT_REPLY。消息头中的 Format 应该设置为 MQFMT_ADMIN,表示后继的消息体是一条管理消息,即 PCF 格式。编码方式 (Encoding) 和字符集 (CodedCharSetId) 指的都是消息体的内容,通常可以选用缺省值。

消息体的结构相对复杂。它分成 PCF 头结构 (MQCFH) 和相关的参数两部分。PCF 头结构中的 Type 表示该消息目前是请求还是应答。在消息请求时可以设置为 MQCFT_COMMAND,在消息应答时,系统会自动设置成 MQCFT_RESPONSE。头结构中的 Command 表示 PCF 命令,在消息请求时将其填入要执行的 PCF 命令标识 (MQCMD_*)。在一次 PCF 命令执行中,请求消息永远只有一条,但应答消息可能会拆分成多条。所以,消息序号 (MsgSeqNumber) 对请求消息来说永远应该设置成 1,消息控制符 (Control) 应该为 MQCFC_LAST。但对应答消息来说,MsgSeqNumber 可以是消息在组中的序号,而 Control 可以是 MQCFC_NOT_LAST 或 MQCFC_LAST。这两个域可以用来接收成组的应答消息。PCF 头中的 ParameterCount 表示后继的参数选项数量。

参数选项可多可少,具体的选择参见产品相关文档。对于请求消息,参数是命令执行的条件,对于应答消息,参数是命令执行的结果。参数结构可能根据需要有所不同,但任何一个参数结构本质上是一个名值对。

在消息请求时,MQPUT 的中消息长度参数指的是整个消息体的长度,包含 PCF 头和全部的后继参数选项。如图 18-2。

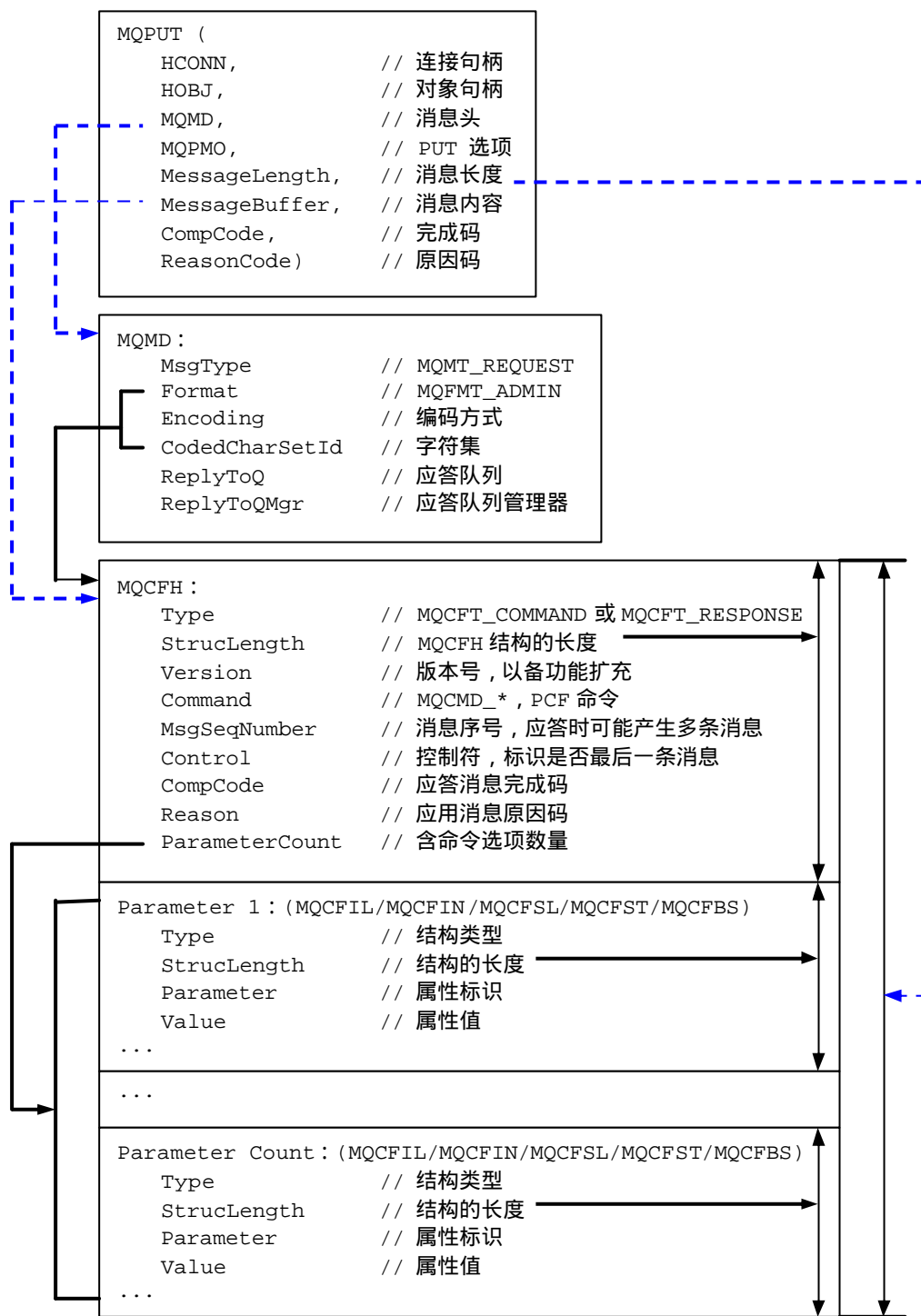


图 18-2 PCF 消息的结构

由于 WebSphere MQ 中的属性有整数型的也有字符串型的，所以 PCF 设计了 5 种参数结构：如图 18-3。

- MQCFIL MQ PCF Integer List 整数数组
- MQCFIN MQ PCF Integer 整数
- MQCFSL MQ PCF String List 字符串数组
- MQCFST MQ PCF String 字符串
- MQCFBS MQ PCF Byte String 二进制字符串

结构中的第一个域 Type 指明是哪一种参数类型，第二个域 StrucLenth 表示结构的长度。

通过这两个域，就可以知道将结构定下来。此外，Parameter 表示属性标识，Value 表示属性的值，如果是数组结构，则由 Count 域表明数组大小。

注意，这里的结构长度 StrucLength 都应该设置为 4 字节的倍数。

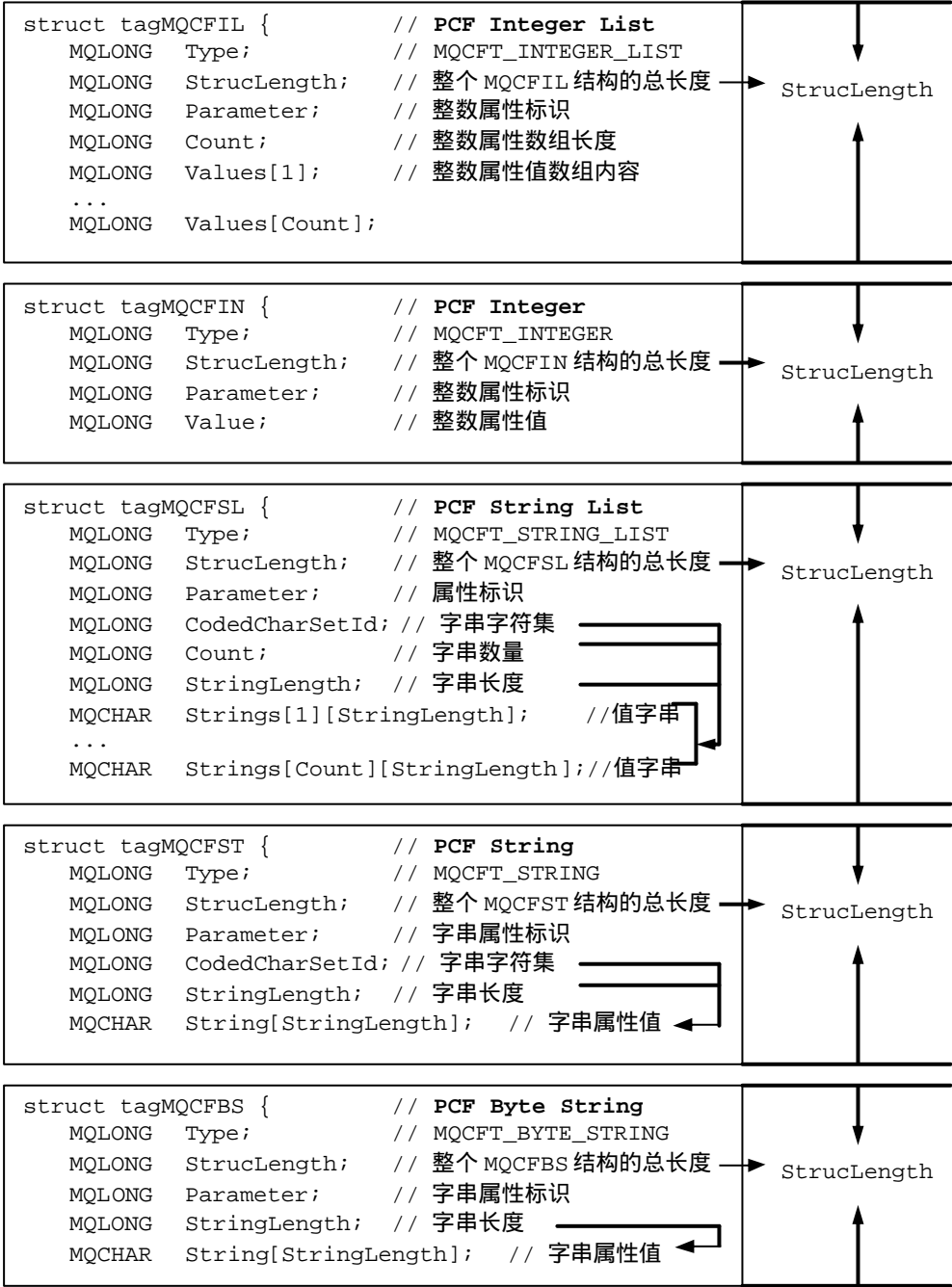


图 18-3 PCF 消息中的 5 种参数结构

18.1.3 格式举例

在 PCF 请求消息中设置 ReplyToQ 和 ReplyToQMgr，指定应答消息返回到 SYSTEM.DEFAULT.LOCAL.QUEUE，同时设置查询对象为本地队列 Q，将请求消息发送至 SYSTEM.ADMIN.COMMAMND.QUEUE。经命令服务器处理后，产生结果，仍以 PCF 格式

存放在应答队列中。用 MQGET 将结构取出，解析 PCF 消息结构，可以得知查询内容。下面我们看一下 PCF 请求和应答消息格式，队列设置如图 18-4。

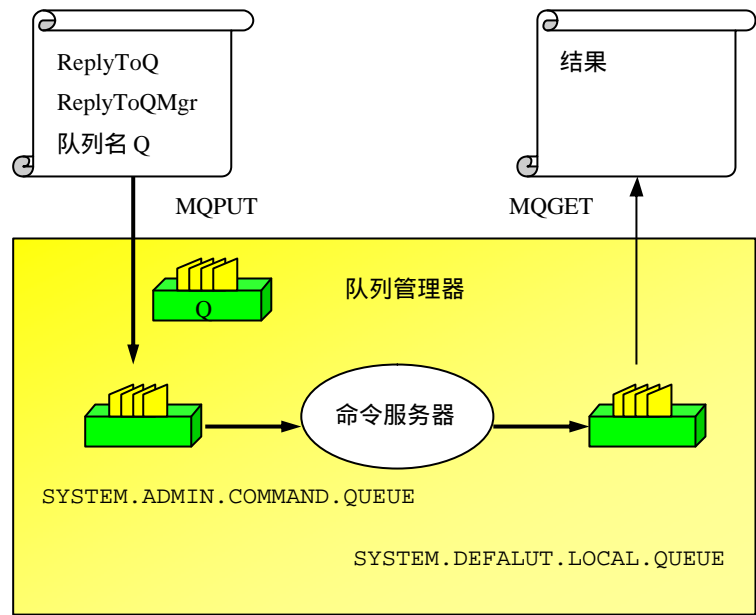


图 18-4 PCF 消息格式举例中的队列设置

18.1.3.1 请求消息格式

MQMD :

MsgType	=	MQMT_REQUEST
Format	=	MQFMT_ADMIN
ReplyToQ	=	SYSTEM.DEFAULT.LOCAL.QUEUE
ReplyToQMGr	=	QM

MQCFH :

Type	=	MQCFT_COMMAND
Command	=	MQCMD_INQUIRE_Q
ParameterCount	=	2

Parameter 1 - MQCFST

StrucLength	=	MQCFST_STRUC_LENGTH_FIXED + MQ_Q_NAME_LENGTH
Parameter	=	MQCA_Q_NAME
StringLength	=	MQ_Q_NAME_LENGTH
String	=	Q

Parameter 2 - MQCFIN

StrucLength	=	MQCFIN_STRUC_LENGTH
Parameter	=	MQIA_Q_TYPE
Value	=	MQQT_LOCAL

18.1.3.2 应答消息格式

MQMD :

MsgType	=	MQMT_REPLY
---------	---	------------

```

MQCFH :
    Type           =   MQCFT_RESPONSE
    ParameterCount  =   44
Parameter 1 - MQCFST
    Parameter      =   MQCA_Q_DESC
    String         =   WebSphere MQ Default Local Queue
    ...
Parameter 44 - MQCFIN
    Parameter      =   MQIA_CURRENT_Q_DEPTH
    String         =   0

```

18.2 AI 编程

18.2.1 消息流程

1. 应用程序用 `mqCreateBag ()` 创建 AI 请求包和应答包 ,在请求包中用 `mqAddXXX ()` 添加各种操作选项
2. 用 `mqExecute ()` 将 AI 请求包提交执行 ,得到执行结果 AI 应答包
3. 用 `mqInquireXXX ()` 从 AI 应答包中将操作结果选项逐一取出分析 ,得到详细的执行结果信息
4. 用 `mqDeleteBag ()` 将 AI 请求包和应用包删除

与 PCF 的消息流程比较 ,发现应用程序只需要与 AI 请求包和应答包打交道 ,且一个 `mqExecute ()` 就可以立即得到执行结果 ,而不需要构造和解析 PCF 消息 ,也不需要通 MQPUT/MQGET 去做队列操作。

事实上 ,WebSphere MQ 会将 AI 包与 PCF 消息之间自动翻译 ,而 PCF 消息的流程同前。如图 18-5。所以说 ,AI 本质上就是对 PCF 的封装。

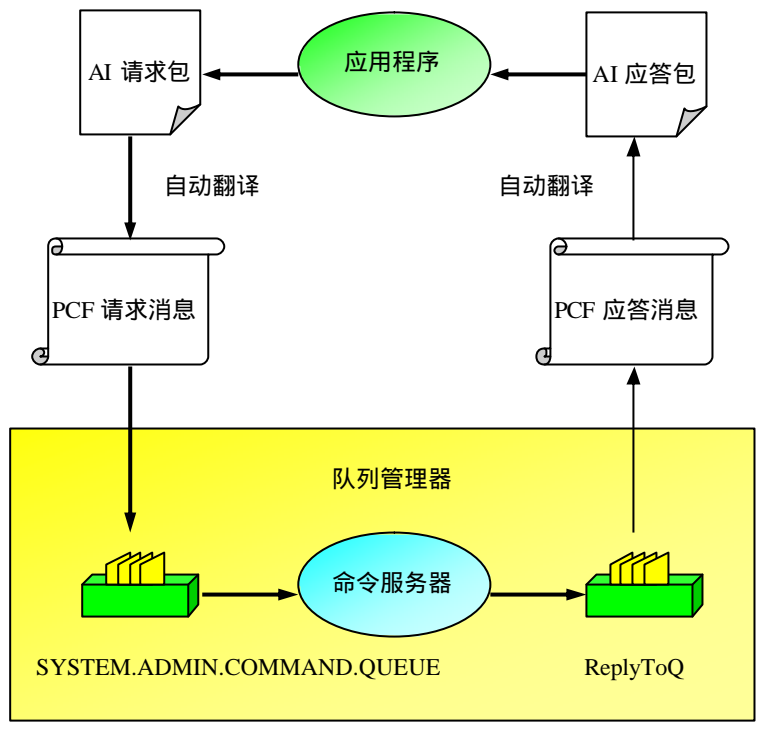


图 18-5 AI 消息流程

18.2.2 包的组成

AI 编程中出现的包 (Bag) 的概念，让我们先来了解一下包的组成，如图 18-6。

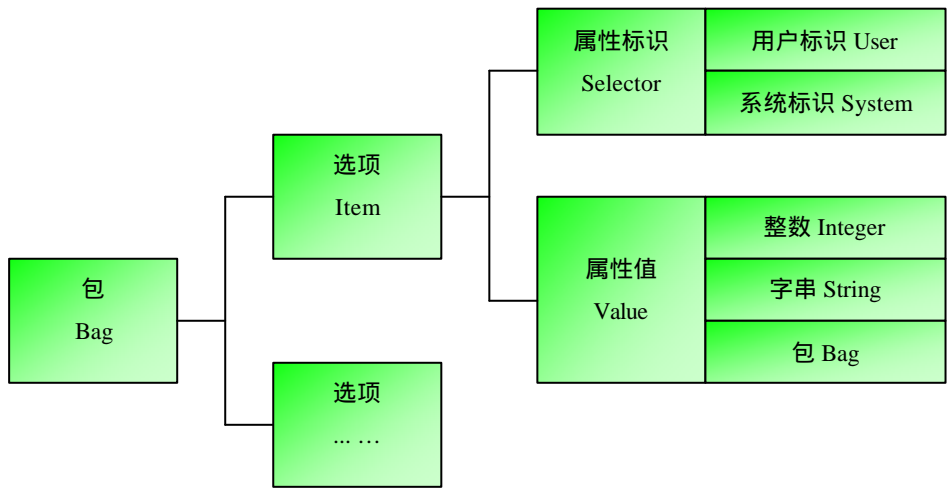


图 18-6 AI 消息中包的组成

包 (Bag) 含若干个选项 (Item)，每个选项含属性标识 (Selector) 和属性值 (Value) 两部分。其中，属性标识分成用户标识和系统标识两种，属性值分成整数、字串和包包柄三种。由此可见，选项 (Item) 也是名值对，其概念几乎与 PCF 中的参数选项 (Parameter) 一样，只是包可以通过选项形成级联关系。

从包的组成不难发现，它几乎是可以与 PCF 消息结构一一映射的。事实上，AI 也提供了 mqBufferToBag () 和 mqBagToBuffer () 来实现两者的相互转换。

18.2.3 编程

18.2.3.1 mqCreateBag, mqDeleteBag, mqClearBag

```
void MQENTRY mqCreateBag (  
    MQLONG      Options,          // 创建选项  
    PMQHBAG     pBag,            // 包句柄  
    PMQLONG     pCompCode,       // 完成码  
    PMQLONG     pReason);        // 原因码  
  
void MQENTRY mqDeleteBag (  
    PMQHBAG     pBag,            // 包句柄  
    PMQLONG     pCompCode,       // 完成码  
    PMQLONG     pReason);        // 原因码  
  
void MQENTRY mqClearBag (  
    PMQHBAG     pBag,            // 包句柄  
    PMQLONG     pCompCode,       // 完成码  
    PMQLONG     pReason);        // 原因码
```

首先，利用 mqCreateBag () 来创建包，以备使用。包刚刚创建的时候，里面是空的，以后可以通过 mqAddXXX () 往里面添加各种操作选项 (Item)。如果不再需要了，可以用 mqClearBag () 来清空内容，以便重用。与 mqCreateBag () 相对的是 mqDeleteBag ()，删除包。

例：

```
mqCreateBag    (MQCBO_ADMIN_BAG,          // Create Bag Options  
                & mqhAdminBag,           // Bag Handle  
                & mqlCompletionCode,     // Completion Code  
                & mqlReasonCode);        // Reason Code  
mqDeleteBag    (& mqhAdminBag,           // Bag Handle  
                & mqlCompletionCode,     // Completion Code  
                & mqlReasonCode);        // Reason Code
```

18.2.3.2 mqAddInteger, mqAddString, mqAddInquiry

```
void MQENTRY mqAddInteger (  
    MQHBAG      Bag,             // 包句柄  
    MQLONG      Selector,        // 选项标识  
    MQLONG      ItemValue,       // 选项值  
    PMQLONG     pCompCode,       // 完成码  
    PMQLONG     pReason);        // 原因码  
  
void MQENTRY mqAddString (  
    MQHBAG      Bag,             // 包句柄  
    MQLONG      Selector,        // 选项标识  
    MQLONG      BufferLength,     // 字符串长度
```

```

    PMQCHAR    pBuffer,          // 字符串值
    PMQLONG    pCompCode,        // 完成码
    PMQLONG    pReason);         // 原因码
void MQENTRY mqAddInquiry (
    MQHBAG     Bag,              // 包句柄
    MQLONG     Selector,         // 选项标识
    PMQLONG    pCompCode,        // 完成码
    PMQLONG    pReason);         // 原因码

```

在已经创建的包中可以添加各种操作选项 (Item)。如果选项属性是整数，则使用 mqAddInteger ()，如果选项属性是字符串，则使用 mqAddString ()。对于这两种调用，都需要在 API 中提供属性值的存放空间，比如 ItemValue 和 Buffer。在自动翻译成 PCF 消息时，WebSphere MQ 会将该空间中的内容也一起拷贝进入消息。如果不指定选项属性的类似，也不提供属性值变量，则可以使用 mqAddInquiry ()。它会在返回的 PCF 消息及包中自动生成相应所需要的空间来存入属性值。

例：

```

mqAddString    (mqhAdminBag,      // Bag Handle
                MQCA_Q_NAME,       // Selector
                MQBL_NULL_TERMINATED, // Buffer Length
                szQName,           // Buffer String
                & mqlCompletionCode, // Completion Code
                & mqlReasonCode);  // Reason Code
mqAddInteger    (mqhAdminBag,      // Bag Handle
                MQIA_Q_TYPE,        // Selector
                MQQT_LOCAL,         // Item Value
                & mqlCompletionCode, // Completion Code
                & mqlReasonCode);  // Reason Code
mqAddInquiry    (mqhAdminBag,      // Bag Handle
                MQIA_MAX_MSG_LENGTH, // Selector
                & mqlCompletionCode, // Completion Code
                & mqlReasonCode);  // Reason Code

```

18.2.3.3 mqExecute, mqPutBag, mqGetBag

```

void MQENTRY mqExecute (
    MQHCONN     Hconn,           // 连接句柄
    MQLONG       Command,         // 命令 MQCMD_*
    MQHBAG       OptionsBag,      // 选项包句柄，可以为空 MQHB_NONE
    MQHBAG       AdminBag,        // 请求包句柄
    MQHBAG       ResponseBag,     // 应答包句柄
    MQHOBJ       AdminQ,          // 请求队列句柄，SYSTEM.ADMIN.COMMAND.QUEUE
    MQHOBJ       ResponseQ,       // 应用队列句柄，ReplyToQ
    PMQLONG      pCompCode,        // 完成码
    PMQLONG      pReason);         // 原因码

```



```

void MQENTRY mqPutBag (
    MQHCONN      Hconn,          // 连接句柄
    MQHOBJ       Hobj,          // 队列句柄
    PMQVOID      pMsgDesc,       // MQMD
    PMQVOID      pPutMsgOpts,    // MQPMO
    MQHBAG       Bag,           // 包句柄
    PMQLONG      pCompCode,      // 完成码
    PMQLONG      pReason);      // 原因码

void MQENTRY mqGetBag (
    MQHCONN      Hconn,          // 连接句柄
    MQHOBJ       Hobj,          // 队列句柄
    PMQVOID      pMsgDesc,       // MQMD
    PMQVOID      pGetMsgOpts,    // MQGMO
    MQHBAG       Bag,           // 包句柄
    PMQLONG      pCompCode,      // 完成码
    PMQLONG      pReason);      // 原因码

```

如果准备好了请求包和应答包，就可以用 mqExecute () 来执行。在一个 API 中完成了 AI 到 PCF 的自动翻译，送出 PCF 请求消息并取回 PCF 应答消息，将结果从 PCF 再自动翻译成 AI 包，API 返回这一系统的动作。为了灵活性，AI 也提供分段式的操作，mqPutBag () 将请求包送出，mqGetBag () 将应答包取回。

例：

```

mqExecute      (mqhConn,          // Connection Handle
                MQCMD_INQUIRE_Q, // Command
                MQHB_NONE,        // Options Bag
                mqhAdminBag,       // Admin Bag Handle
                mqhReplyBag,       // Reply Bag Handle
                MQHO_NONE,         // Admin Queue Handle,
                                // default SYSTEM.ADMIN.COMMAND.QUEUE
                MQHO_NONE,         // Reply Queue Handle,
                                // default Qynamic Queue
                & mqlCompletionCode, // Completion Code
                & mqlReasonCode);   // Reason Code

mqPutBag       (mqhConn,          // Connection Handle
                mqhObj,           // Object Handle
                & mqmd,           // Message Descriptor
                & mqpmo,          // Put Message Options
                mqhBag,           // Bag Handle
                & mqlCompletionCode, // Completion Code
                & mqlReasonCode);   // Reason Code

mqGetBag       (mqhConn,          // Connection Handle
                mqhObj,           // Object Handle
                & mqmd,           // Message Descriptor
                & mqgmo,          // Get Message Options

```

```

mqhBag, // Bag Handle
& mqlCompletionCode, // Completion Code
& mqlReasonCode); // Reason Code

```

18.2.3.4 mqInquireInteger, mqInquireString, mqInquireBag

```

void MQENTRY mqInquireInteger (
    MQHBAG    Bag, // 包句柄
    MQLONG    Selector, // 选项标识
    MQLONG    ItemIndex, // 选项索引, 在选项数组中的下标
    PMQLONG    pItemValue, // 数据选项值, 返回整数属性值
    PMQLONG    pCompCode, // 完成码
    PMQLONG    pReason); // 原因码

void MQENTRY mqInquireString (
    MQHBAG    Bag, // 包句柄
    MQLONG    Selector, // 选项标识
    MQLONG    ItemIndex, // 选项索引, 在选项数组中的下标
    MQLONG    BufferLength, // 字串缓冲区大小
    PMQCHAR    pBuffer, // 字串缓冲区
    PMQLONG    pStringLength, // 字串长度
    PMQLONG    pCodedCharSetId, // 字串字符集
    PMQLONG    pCompCode, // 完成码
    PMQLONG    pReason); // 原因码

void MQENTRY mqInquireBag (
    MQHBAG    Bag, // 包句柄
    MQLONG    Selector, // 选项标识
    MQLONG    ItemIndex, // 选项索引, 在选项数组中的下标
    PMQHBAG    pItemValue, // 数据选项值, 返回子包的句柄
    PMQLONG    pCompCode, // 完成码
    PMQLONG    pReason); // 原因码

```

包 (Bag) 中的内容可以通过 mqInquireXXX () 来查看。如果是整数类型, 可以用 mqInquireInteger () 获得相应选项 (Item) 值, 如果是字串类型, 则可以用 mqInquireString (), 如果选项本身又是一个子包, 则可以用 mqInquireBag () 获得子包后进一步解析。

然而, 在解析应答包的时候, 通常并不知道应答包里会有什么, 有多少个选项, 也不知道每个选项的类型。这时, 就需要 mqCountItems () 和 mqInquireItemInfo () 了。

例:

```

mqInquireInteger (mqhBag, // Bag Handle
    MQSEL_ANY_SELECTOR, // Selector
    i, // Item Index
    & mqlItemValue, // Item Value
    & mqlCompletionCode, // Completion Code
    & mqlReasonCode); // Reason Code

mqInquireString (mqhBag, // Bag Handle

```

```

        MQSEL_ANY_SELECTOR,    // Selector
        i,                    // Item Index
        BUFFER_SIZE,          // Length
        szStringBuffer,       // Buffer
        & mqlStringLength,    // String Length
        & mqlCCSI,            // Code Char Set ID
        & mqlCompletionCode,   // Completion Code
        & mqlReasonCode);     // Reason Code
mqInquireBag (mqhBag,         // Bag Handle
        MQSEL_ANY_SELECTOR,   // Selector
        i,                    // Item Index
        & mqhSubBag,          // Item Value
        & mqlCompletionCode,  // Completion Code
        & mqlReasonCode);    // Reason Code

```

18.2.3.5 mqCountItems, mqInquireItemInfo

```

void MQENTRY mqCountItems (
    MQHBAG    Bag,           // 包句柄
    MQLONG    Selector,      // 选项标识, 可以用 MQSEL_ANY_SELECTOR 任意匹配
    PMQLONG   pItemCount,    // 选项数量
    PMQLONG   pCompCode,     // 完成码
    PMQLONG   pReason);      // 原因码

void MQENTRY mqInquireItemInfo (
    MQHBAG    Bag,           // 包句柄
    MQLONG    Selector,      // 选项标识, 可以用 MQSEL_ANY_SELECTOR 任意匹配
    MQLONG    ItemIndex,     // 选项索引, 在选项数组中的下标
    PMQLONG   pOutSelector,   // 数据选项标识
    PMQLONG   pItemType,     // 数据选项类型
    PMQLONG   pCompCode,     // 完成码
    PMQLONG   pReason);      // 原因码

```

mqCountItems () 用来计算包中选项的数量, mqInquireItemInfo () 则用来查看每一个选项的类型, 以便选用合适的 mqInquireXXX 去取出相应的选项值。对请求包和应答包都可以用这种方法进行内容清点。

例：

```

mqCountItems (mqhBag,        // Bag Handle
        MQSEL_ALL_SELECTORS, // Selector
        & mqlItemCount,     // Item Count
        & mqlCompletionCode, // Completion Code
        & mqlReasonCode);   // Reason Code
mqInquireItemInfo (mqhBag,   // Bag Handle
        MQSEL_ANY_SELECTOR,  // Selector
        i,                   // Item Index

```

```

        & mqlSelector,          // Out Selector
        & mqlItemType,          // Item Type
        & mqlCompletionCode,    // Completion Code
        & mqlReasonCode);      // Reason Code

```

18.2.3.6 mqSetInteger, mqSetString, mqDeleteItem, mqTruncateBag

```

void MQENTRY mqSetInteger (
    MQHBAG      Bag,          // 包句柄
    MQLONG      Selector,      // 选项标识
    MQLONG      ItemIndex,     // 选项索引，在选项数组中的下标
    MQLONG      ItemValue,     // 整数选项值
    PMQLONG     pCompCode,     // 完成码
    PMQLONG     pReason);      // 原因码

void MQENTRY mqSetString (
    MQHBAG      Bag,          // 包句柄
    MQLONG      Selector,      // 选项标识
    MQLONG      ItemIndex,     // 选项索引，在选项数组中的下标
    MQLONG      BufferLength,   // 字符串选项缓冲区长度
    PMQCHAR     pBuffer,       // 字符串选项缓冲区
    PMQLONG     pCompCode,     // 完成码
    PMQLONG     pReason);      // 原因码

void MQENTRY mqDeleteItem (
    MQHBAG      Bag,          // 包句柄
    MQLONG      Selector,      // 选项标识
    MQLONG      ItemIndex,     // 选项索引，在选项数组中的下标
    PMQLONG     pCompCode,     // 完成码
    PMQLONG     pReason);      // 原因码

void MQENTRY mqTruncateBag (
    MQHBAG      Bag,          // 包句柄
    MQLONG      ItemCount,     // 选项数量
    PMQLONG     pCompCode,     // 完成码
    PMQLONG     pReason);      // 原因码

```

在确定了选项之后，可以根据选项类型用 `mqSetInteger()` 或 `mqSetString()` 对其内容进行修改。如果某些选项不再需要了，可以用 `mqDeleteItem()` 将选项从包中删除，但被删除的选项会在包中留下空位，其中的内容不会被翻译进入 PCF 消息。用 `mqTruncateBag()` 将从最近加入的选项开始计算，此后的所有选项删除。

18.2.3.7 mqBagToBuffer, mqBufferToBag

```

void MQENTRY mqBagToBuffer (
    MQHBAG      OptionsBag,    // 参数包句柄
    MQHBAG      DataBag,       // 包句柄
    MQLONG      BufferLength,   // PCF 消息缓冲区大小

```

```

    PMQVOID    pBuffer,          // PCF 消息缓冲区
    PMQLONG    pDataLength,      // PCF 消息缓冲区中消息的长度
    PMQLONG    pCompCode,        // 完成码
    PMQLONG    pReason);         // 原因码
void MQENTRY mqBufferToBag (
    MQHBAG     OptionsBag,       // 参数包句柄
    MQLONG      BufferLength,      // PCF 消息缓冲区大小
    PMQVOID     pBuffer,          // PCF 消息缓冲区
    MQHBAG      DataBag,          // 包句柄
    PMQLONG     pCompCode,        // 完成码
    PMQLONG     pReason);         // 原因码

```

WebSphere MQ 提供了 AI 中的包和 PCF 中的消息之间的翻译函数：mqBagToBuffer 和 mqBufferToBag。

例：

```

mqBagToBuffer (MQHB_NONE,          // Options Bag
               mqhBag,              // Data Bag
               BUFFER_SIZE,         // Buffer Length
               pmqbBuffer,          // Buffer
               pmqlDataLength,      // Data Length
               & mqlCompletionCode, // Completion Code
               & mqlReasonCode);    // Reason Code
mqBufferToBag (MQHB_NONE,          // Options Bag
               mqlBufferLength,     // Buffer Length
               pmqbBuffer,          // Buffer
               mqhBag,              // Data Bag
               & mqlCompletionCode, // Completion Code
               & mqlReasonCode);    // Reason Code

```

18.2.3.8 mqPad, mqTrim

```

void MQENTRY mqPad (
    PMQCHAR     pString,          // 原字符串
    MQLONG      BufferLength,      // 缓冲区长度
    PMQCHAR     pBuffer,          // 缓冲区，函数会将原字符串拷入后补足空格
    PMQLONG     pCompCode,        // 完成码
    PMQLONG     pReason);         // 原因码
void MQENTRY mqTrim (
    MQLONG      BufferLength,      // 缓冲区长度
    PMQCHAR     pBuffer,          // 缓冲区
    PMQCHAR     pString,          // 字符串，函数会将缓冲区内容末尾的空格删除后拷入该字符串
    PMQLONG     pCompCode,        // 完成码
    PMQLONG     pReason);         // 原因码

```

这是两个辅助编程的函数。mqPad () 在字串后加足空格，mqTrim () 则正相反，将字串末尾的空格删除。

例：

```
mqTrim      (mqlStringLength,      // String Length
             szStringBuffer,        // Buffer
             szStringBuffer,        // String
             & mqlCompletionCode,    // Completion Code
             & mqlReasonCode);      // Reason Code
```

以上介绍了 AI 的编程模式，进一步的内容，请参考本书例程和相关编程手册。

附录 WebSphere MQ 进程一览表

Windows 平台

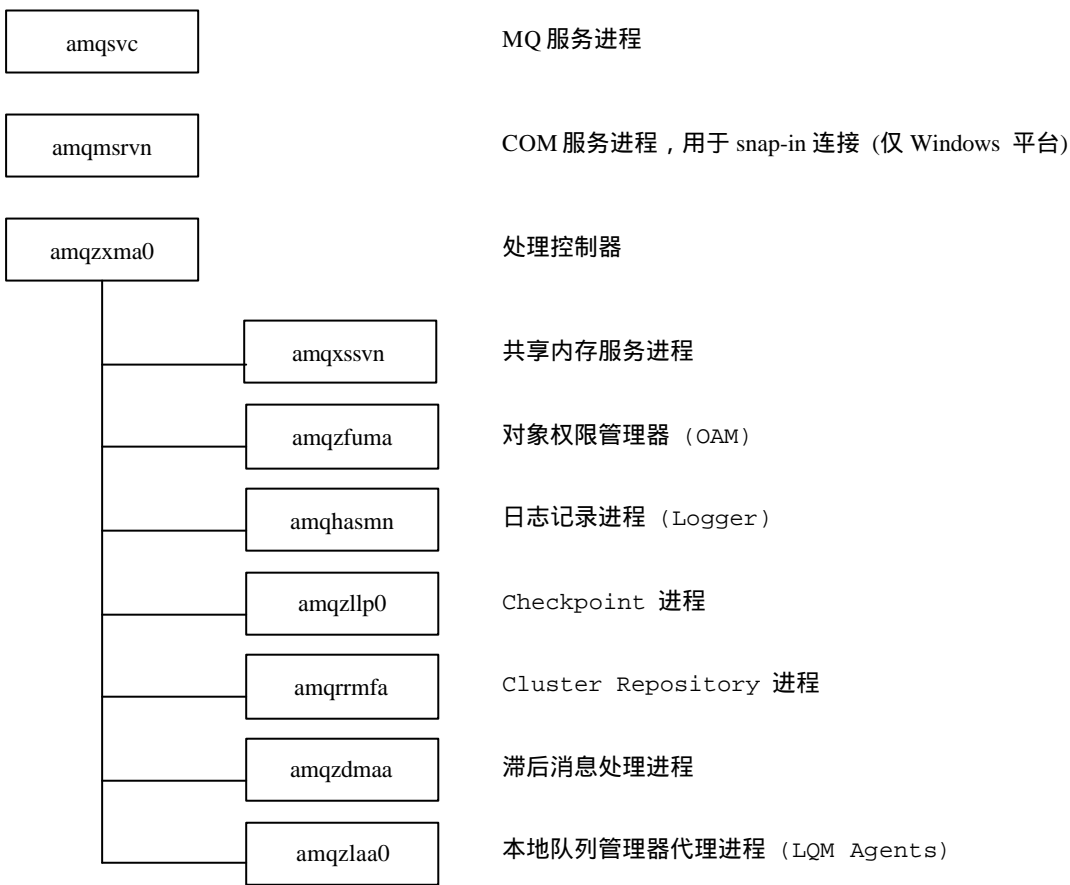
amqpcsea.exe	命令服务器 (Command Server)
amqhasmn.exe	日志记录进程 (Logger)
amqharmn.exe	日志格式化进程 (Log Formatter, 仅支持线型日志)
amqzllp0.exe	Checkpoint 进程
amqzlaa0.exe	本地队列管理器代理进程 (LQM Agents)
amqzfuma.exe	对象权限管理器 (OAM)
amqztrcn.exe	Trace 进程
amqzxma0.exe	处理控制器
amqxssvn.exe	共享内存服务进程
amqrrmfa.exe	Cluster Repository 进程
amqzdmaa	滞后消息处理进程
amqrmppa	通道接收进程 (Channel Receiver)
amqcrs6a	消息接收进程 (Inbound MCA, LU 6.2)
amqcrsta	消息接收进程 (Inbound MCA, TCP)
runmqchl	消息发送进程 (Outbound MCA, Any)
runmqchi	消息发送进程 (Outbound MCA, Any)
amqzlwa0	群集负载平衡 (CLWL) 进程

UNIX 平台

amqpcsea	命令服务器 (Command Server)
amqhasmx	日志记录进程 (Logger)
amqharmx	日志格式化进程 (Log Formatter, 仅支持线型日志)
amqzllp0	Checkpoint 进程
amqzlaa0	本地队列管理器代理进程 (LQM Agents)

amqzfuma	对象权限管理器 (OAM)
amqzxma0	处理控制器
amqrrmfa	Cluster Repository 进程
amqzdmaa	滞后消息处理进程
amqrmppa	通道接收进程 (Channel Receiver)

进程树



附录 WebSphere MQ 命令一览表

队列管理器 (Queue Manager)

crtmqm 创建队列管理器 (Create Queue Manager)

- `[-c Text]` 描述，最多 64 个字符
- `[-d DefaultTransmissionQueue]` 缺省传输队列

<code>[-h MaximumHandleLimit]</code>	一个应用程序可以 MQOPEN 的最大句柄数 min=1 , max=999,999,999 , default=256
<code>[-lc -ll]</code>	Log 类型
<code>-lc</code>	Circular Logging , 环型日志
<code>-ll</code>	Linear Logging , 线型日志
<code>[-ld LogPath]</code>	Log 文件的目录 , mqm 用户必须有访问权限 , 缺省为 : Windows X:\Program Files\IBM\WebSphere MQ\log\qmgr UNIX /var/mqm/log
<code>[-lf LogFileSize]</code>	Log 文件大小 , 4KB 的倍数 . Windows min=32 , max=16,384 , default=256 (1 MB) UNIX min=64 , max=16,384 , default=1024 (4 MB)
<code>[-lp LogPrimaryFiles]</code>	主 Log 文件数量 , min=2 , max=62 , default=3
<code>[-ls LogSecondaryFiles]</code>	备用 Log 文件数量 , min=1 , max=61 , default=2 注意 : LogPrimaryFiles + LogSecondaryFiles <= 63
<code>[-q]</code>	缺省队列管理器
<code>[-g ApplicationGroup]</code>	应用组。Application Group 中的用户可以运行 MQI 应用 , 更新 IPCC 资源 , 改变 Queue Manager 目录中的内容。仅对 WMQ for AIX , Solaris , HPUX , Linux 有效。会反映到 qm.ini 中。mqm 必须是 Application Group 中的用户。 缺省 -g all
<code>[-t IntervalValue]</code>	min=0 , max=999,999,999 , default=999,999,999 , 单位: 毫秒。触发间隔 (Trigger Time Interval)
<code>[-u DeadLetterQueue]</code>	死信队列 (Dead Letter Queue)
<code>[-x MaximumUncommittedMessages]</code>	最大的未提交的消息数量 min=1 , max=999,999,999 , default=10,000 在一个交易中的消息最大数量 , 为 MQPUT + MQGET + LUW 中产生的 Trigger Message 数量之和
<code>[-z]</code>	抑止出错信息
QMgrName	队列管理器名

例:

```
crtmqm -t 5000 -u SYSTEM.DEAD.LETTER.QUEUE -ll QM
```

dltmqm 删除队列管理器 (Delete Queue Manager)

<code>[-z]</code>	抑止出错信息
QMgrName	队列管理器名

例:

```
dltmqm -z QM
```

strmqm 启动队列管理器 (Start Queue Manager)

<code>[-c]</code>	启动队列管理器 , 覆盖重建所有的系统对象 , 再停止该队列管理器
-------------------	-----------------------------------

例：

endmqm 停止队列管理器 (End Queue Manager)

例：

dspmq 显示队列管理器 (Display Queue Manager)

例：

QMNAME (QM) STATUS (正在运行)

命令服务器 (Command Server)

strmqcsv 启动命令服务器 (Start Command Server)

[QMgrName] 缺省为系统的缺省队列管理器

例：

```
strmqcsv QM
```

endmqcsv 停止命令服务器 (End Command Server)

[-c | -i]

-c Controlled, 允许已经开始处理的 Command 消息完成。缺省值

-i Immediately, 中止正在处理的 Command 任务

QMgrName 队列管理器

例：

```
endmqcsv QM
```

```
endmqcsv -i QM
```

dspmqcsv 显示命令服务器 (Display Command Server)

[QMgrName] 缺省为系统的缺省队列管理器

返回的 Command Server 状态可能是：

- Starting
- Running
- Running with SYSTEM.ADMIN.COMMAND.QUEUE not enabled for gets
- Ending
- Stopped

例：

```
dspmqcsv QM
```

Listener (监听器)

runmqlsr 运行监听器 (Run Listener)

-t tcp

[-p Port] Port 口, 缺省为 1414

[-i IPAddr] IP 地址, 如果不指定, Listener 会监听所有的 IP 地址

<code>[-b Backlog]</code>	允许的并发连接请求数
<code>-t lu62</code>	
<code>-n TpName</code>	交易名, 如果不指定, 从 Queue Manager 的配置中取
<code>-t netbios</code>	
<code>-a Adapter</code>	NetBIOS Adapter Number。缺省为 0
<code>-l LocalName</code>	NetBIOS Local Name。缺省从队列管理器的配置中取
<code>-e Names</code>	Listener 可用的 Name 数量。缺省从队列管理器的配置中取
<code>-s Sessions</code>	Listener 可用的 Session 数量。缺省从队列管理器的配置中取
<code>-o Commands</code>	Listener 可用的 Command 数量。缺省从队列管理器的配置中取
<code>-t spx</code>	
<code>-x Socket</code>	缺省为 0x5E86
<code>-b Backlog</code>	允许的并发连接请求数
<code>-t udp</code>	
<code>[-p Port]</code>	Port 口, 缺省为 1414
<code>[-m QMgrName]</code>	缺省为系统的缺省队列管理器

备注:

`-t` 为必选项, 其它为可选项

例:

```
runmqqlsr -t tcp -p 1414 -m QM
```

endmqqlsr 停止监听器 (End Listener)

<code>[-w]</code>	等到命令完成后才返回控制
<code>[-m QMgrName]</code>	缺省为系统的缺省队列管理器

触发监控器 (Trigger Monitor)

runmqtmc 启动 Client 端触发监控器 (Run Trigger Monitor for Client)

<code>[-m QMgrName]</code>	缺省为系统的缺省队列管理器
<code>[-q InitiationQName]</code>	初始化队列名字, 缺省为 SYSTEM.DEFAULT.INITIATION.QUEUE

runmqtrm 启动 Server 端触发监控器 (Run Trigger Monitor for Server)

<code>[-m QMgrName]</code>	缺省为系统的缺省队列管理器
<code>[-q InitiationQName]</code>	初始化队列名字, 缺省为 SYSTEM.DEFAULT.INITIATION.QUEUE

Trace

strmqtrc 启动 Trace (Start Trace , Windows 平台)

`[-t TraceType]`

`[-x TraceType]`

`[-l MaxSize]`

解释同后

strmqtrc 启动 Trace (Start Trace , HP-UX , Solaris , Linux 平台)

`[-m QMgrName]`

缺省为系统的缺省队列管理器

`[-e]`

Early Tracing。可以 Trace 队列管理器的创建与启动。

`[-t TraceType]`

Trace 的部分，可以有多个 `-t` 选项

`[-x TraceType]`

不要 Trace 的部分，可以有多个 `-x` 选项

TraceType 可以是以下的组合：

`all`

全部，是以下全部选项的叠加

`api`

MQI 调用和主要队列管理器部件相关信息

`commentary`

部件中的评注信息

`comms`

网络通信上的数据流

`csdata`

通用服务的内部数据

`csflows`

通用服务的控制过程

`detail`

控制过程的细节数据

`lqmdata`

本地队列管理器的内部数据

`lqmflows`

本地队列管理器的控制过程

`otherdata`

其它部件的内容数据

`otherflows`

其它部件的控制过程

`parms`

激活 trace 控制过程

`remotedata`

通信部件的内部数据

`remoteflows`

通信部件的控制过程

`servicedata`

服务部件的内部数据

`serviceflows`

服务部件的控制过程

`ssl`

用 GSKit 与 SSL Channel 安全数据 ,不支持 WebSphere MQ for Windows

`versiondata`

WebSphere MQ 版本信息

`[-l MaxSize]`

Trace 文件 (AMQnnnnn.TRC) 的上限。以 MB 为单位。如果达到上限 ,Trace 文件会更名为 AMQnnnnn.TRS ,且创建出新的 Trace 文件 AMQnnnnn.TRC。如果原来存在 AMQnnnnn.TRS ,则原来的文件被删除。

例：

`strmqtrc -m QM`

`strmqtrc -m QM -e`

```
strmqtrc -m QM -t all -x ssl
```

endmqtrc 停止 Trace (End Trace , Windows 平台)

解释同后

endmqtrc 停止 Trace (End Trace , HP-UX , Solaris , Linux 平台)

<code>[-m QMgrName]</code>	缺省为系统的缺省队列管理器
<code>[-e]</code>	Early Tracing.
<code>[-a]</code>	All Tracing。必须单独设置

例：

```
endmqtrc -m QM
endmqtrc -m QM -e
```

dspmqtrc 显示 Trace (Display Trace , HP-UX , Solaris , Linux 平台)

<code>[-t FormatTemplate]</code>	格式文件，缺省为 <code>/opt/mqm/lib/amqtrc.fmt</code>
<code>[-h]</code>	从 Report 中删去 Header Information
<code>[-s]</code>	抽取出 Trace Header 输出到 stdout
<code>[-o OutputFilename]</code>	输出文件
<code>InputFileName</code>	Trace 文件。如果只有一个文件，输出可以是 stdout 或是 <code>-o</code> 指定的文件。如果不止一个文件，输出为 <code>AMQXXXXX.FMT (PID)</code>

介质恢复 (Media Recover)

rcdmqimg 记录对象映像 (Record Object Image)

<code>[-m QMgrName]</code>	缺省为系统的缺省队列管理器
<code>[-z]</code>	抑止出错信息
<code>[-l]</code>	输出重启 Queue Manager 和介质恢复所需的最早的日志文件名。

例：

AMQ7467：启动队列管理器 QM 所需的最早的日志文件是 S0000000.LOG。

AMQ7468：执行队列管理器 QM 的介质恢复所需的最早的日志文件是 S0000001.LOG。

输出到出错日志 `../QM/errors/AMQERR01.LOG` 上，同时输出到 stderr 上。如果 `-l -z` 同时使用，则只输出到出错日志上。

<code>-t ObjectType</code>	
<code>nl 或 namelist</code>	名字列表
<code>prcs 或 process</code>	进程

q 或 queue	队列
ql 或 qlocal	本地队列
qa 或 qalias	别名队列
qr 或 qremote	远程队列
qm 或 qmodel	模型队列
qmgr	队列管理器
syncfile	通道同步文件
ctlg 或 catalog	对象目录
authinfo	授权信息, SSL 通道
*或 all	全部
GenericObjName	对象名字

备注

此命令仅对 Linear Logging 的 Queue Manager 有效

例:

```
rcdmqimg -l -m QM -t queue Q
```

记录了对象 Q 类型 queue 的媒体映像。

AMQ7467: 启动队列管理器 QM 所需的最早的日志文件是 S0000000.LOG。

AMQ7468: 执行队列管理器 QM 的介质恢复所需的最早的日志文件是 S0000000.LOG。

rcrmqobj 重建对象 (Recreate Object)

[-m QMgrName]	缺省为系统的缺省队列管理器
[-z]	抑止出错信息
-t ObjectType	
nl 或 namelist	名字列表
prcs 或 process	进程
q 或 queue	队列
ql 或 qlocal	本地队列
qa 或 qalias	别名队列
qr 或 qremote	远程队列
qm 或 qmodel	模型队列
qmgr	队列管理器
syncfile	通道同步文件
ctlg 或 catalog	对象目录
authinfo	授权信息, SSL 通道
*或 all	全部
GenericObjName	对象名字

备注

此命令仅对 Linear Logging 的 Queue Manager 有效

例:

```
rcrmqobj -m QM -t queue Q
```

重新建立类型 queue 的对象 Q 。

日志 (Log)

dmpmqlog 输出格式化日志

```
[ -b | -s StartLSN | -n ExtentNumber ]
```

指定 Dump 开始的 Log Sequence Number (LSN)

-b 从 Base LSN 开始, Base LSN active log 中的第一个

-s StartLSN 从指定的某个 LSN 开始。StartLSN 格式为
nnnn:nnnn:nnnn:nnnn。
如果是 Circular Logging, StartLSN 必须大于等于 Base LSN

-n ExtentNumber 从指定的某 ExtentNumber 开始。取值为 0 - 9,999,999。仅对
Linear Logging 有效

[-e EndLSN] 指定 Dump 结束的 LSN。EndLSN 格式为 nnnn:nnnn:nnnn:nnnn.

[-f LogFilePath] 指定 Log 目录, 缺省为 C:\Program Files\IBM\WebSphere
MQ\log\QMGr

[-m QMgrName] 缺省为系统的缺省队列管理器

例:

```
dmpmqlog -m QM > QMLog.dmp
```

容量单元 (Capacity)

dspmqcap 显示容量单元 (Display Capacity)

例:

```
dspmqcap
```

购买的处理器定量为 1
此机器中的处理器数为 1

setmqcap 设置容量单元 (Set Capacity)

例:

```
setmqcap 2
```

权限信息 (Authority)

dmpmqaut 输出权限信息 (Dump Authority)

<code>[-m QMgrName]</code>	缺省为系统的缺省队列管理器
<code>[-n Profile -l]</code>	
<code>-n Profile</code>	输出 Profile 文件名
<code>-l</code>	仅输出 Profile 名字和类型
<code>[-t ObjectType]</code>	对象类型
<code>nl 或 namelist</code>	名字列表
<code>prcs 或 process</code>	进程
<code>q 或 queue</code>	队列
<code>qmgr</code>	队列管理器
<code>authinfo</code>	授权信息, SSL 通道
<code>[-s ServiceComponent]</code>	如果有 Installable Authorization Services, 通过这个选项可以提供 Authorization Service 的名字。如果不用这个选项, Authorization Inquiry 使用 Service 中的第一个安装部件。
<code>[-p PrincipalName -g GroupName]</code>	
<code>-g GroupName</code>	组名
<code>-p PrincipalName</code>	用户名. 如果是 Windows 系统, 可以用 userid@domain

例:

```
dmpmqaut -m QM -l -t queue
dmpmqaut -m QM -n Q -t queue
```

dspmqaut 显示权限信息 (Display Authority)

<code>[-m QMgrName]</code>	缺省为系统的缺省队列管理器
<code>-n ObjectName</code>	对象名。如果是队列管理器, 则可以不选此项
<code>-t ObjectType</code>	对象类型
<code>nl 或 namelist</code>	名字列表
<code>prcs 或 process</code>	进程
<code>q 或 queue</code>	队列
<code>qmgr</code>	队列管理器
<code>authinfo</code>	授权信息, SSL 通道
<code>[-s ServiceComponent]</code>	如果有 Installable Authorization Services, 通过这个选项可以提供 Authorization Service 的名字。如果不用这个选项, Authorization Inquiry 使用 Service 中的第一个安装部件。
<code>[-p PrincipalName -g GroupName]</code>	
<code>-g GroupName</code>	组名
<code>-p PrincipalName</code>	用户名. 如果是 Windows 系统, 可以用 userid@domain

例：

```
dspmqaut -m QM -n Q -t queue -p chenyux
```

实体 chenyux 对于对象 Q 具有下列权限：

```
get
browse
put
inq
set
crt
dlt
chg
dsp
passid
passall
setid
setall
clr
```

```
dspmqaut -m QM -t qmgr -g mqm
```

实体 mqm 对于对象 QM 具有下列权限：

```
inq
set
connect
altusr
crt
dlt
chg
dsp
setid
setall
```

setmqaut设置权限信息 (Set Authority)

<code>[-m QMgrName]</code>	缺省为系统的缺省队列管理器
<code>-n Profile</code>	Profile 文件名
<code>-t ObjectType</code>	对象类型
<code>nl 或 namelist</code>	名字列表
<code>prcs 或 process</code>	进程
<code>q 或 queue</code>	队列
<code>qmgr</code>	队列管理器
<code>authinfo</code>	授权信息，SSL 通道
<code>[-p PrincipalName -g GroupName]</code>	
<code>-g GroupName</code>	组名
<code>-p PrincipalName</code>	用户名。如果是 Windows 系统，可以用 <code>userid@domain</code>

[-s ServiceComponent] 如果有 Installable Authorization Services , 通过这个选项可以提供 Authorization Service 的名字。如果不用这个选项 , Authorization Inquiry 使用 Service 中的第一个安装部件 .

[-remove] 删除 Profile 文件

%MQI authorizations%

+altusr	添加更换用户身份权限
-altusr	删除更换用户身份权限
+browse	添加浏览队列 MQGET (BROWSE) 权限
-browse	删除浏览队列 MQGET (BROWSE) 权限
+connect	添加连接队列管理器 MQCONN 权限
-connect	删除连接队列管理器 MQCONN 权限
+get	添加读消息 MQGET 权限
-get	删除读消息 MQGET 权限
+inq	添加查询对象参数 MQINQ 权限
-inq	删除查询对象参数 MQINQ 权限
+put	添加写消息 MQPUT 权限
-put	删除写消息 MQPUT 权限
+set	添加设置对象参数 MQSET 权限
-set	删除设置对象参数 MQSET 权限

%Context authorizations%

+passall	添加传递所有上下文权限
-passall	删除传递所有上下文权限
+passid	添加传递身份上下文权限
-passid	删除传递身份上下文权限
+setall	添加设置所有上下文权限
-setall	删除设置所有上下文权限
+setid	添加设置身份上下文权限
-setid	删除设置身份上下文权限

%Administration authorizations%

+chg	添加更改对象权限
-chg	删除更改对象权限
+clr	添加清除对象权限
-clr	删除清除对象权限
+crt	添加创建对象权限
-crt	删除创建对象权限
+dlt	添加删除对象权限
-dlt	删除删除对象权限
+dsp	添加显示对象权限
-dsp	删除显示对象权限

%Generic authorizations%

+all	添加所有权限
-all	删除所有权限

+alladm	添加所有管理操作可达到的权限
-alladm	删除所有管理操作可达到的权限
+allmqi	添加所有 MQI 可达到的权限
-allmqi	删除所有 MQI 可达到的权限
+none	无权限，用来创建不含权限的 Profile

例：

```
setmqaut -m QM -n Q -t queue -g group +inq +alladm
setmqaut -m QM -n Q -t queue -g group -allmqi +alladm
setmqaut -m QM -n a.b.* -t q -p mqm +all
setmqaut -m QM -n a.b.* -t q -p mqm -remove
setmqaut -m QM -n a.b.* -t q -p mqm +none
```

amqoamd 输出授权信息 (OAM Dump)

[-m QMgrName]	缺省为系统的缺省队列管理器
[-t ObjectType]	对象类型
[-n ObjName]	对象名
[-f -s]	
-f	旧的授权文件格式
-s	输出 setmqaut 命令

例；

```
amqoamd
```

运行环境 (Environment)

mqver 显示版本 (WebSphere MQ Version)

例：

```
mqver
Name:           WebSphere MQ
Version:        530.4 CSD04
CMVC level:    p530-04-030617
BuildType:     IKAP - (Production)
```

这里很清楚地表明 WebSphere MQ 的版本为 5.3，目前的补丁是 CSD04。

具体的 PTF 说明可参考：C:\Program Files\IBM\WebSphere MQ\PTF\zh_cn\memo.ptf

setmqprd 设置生产环境 (Set Production)

```
[LicenseFile]
```

例：

```
setmqprd "C:\Program Files\IBM\WebSphere MQ\bin\amqpcert.lic"
```

此 WebSphere MQ 副本正在以 "生产" 方式运行。

备注：

WebSphere MQ 有三种不同的安装介质，试用版 (Trial)、测试版 (Beta) 和生产版 (Production)，分别对应三个不同的许可文件 amqtcert.lic，amqbcert.lic 和 amqpcert.lic。如果系统用生产版介质安装覆盖了原来的试用版或测试版，则用此命令可以加载生产许可证文件，切换到生产方式运行。

amqicsdn 安装补丁 (Install CSD)

例：

```
amqicsdn
```

备注：

通常这条命令在展开的补丁包中。

高可用性 (High-Availability , Windows 平台)

hadltmqm 删除队列管理器 (HA Delete Queue Manager)

```
/m QMgrName                      队列管理器
```

备注：

从一个结点上删除 HA 环境下的队列管理器

hamvmqm 移动队列管理器 (HA Move Queue Manager)

```
/m QMgrName                      队列管理器  
/dd AbsoluteDataDirectory      数据目录的绝对路径  
/ld AbsoluteLogDirectory        日志目录的绝对路径
```

备注：

将队列管理器移至 MSCS 存储空间

haregtyp 注册队列管理器 (HA Register Type)

```
/r                                  注册  
/u                                  注销
```

备注：

在 MSCS (Microsoft Cluster Server) 中注册或注销队列管理器

amqmsysn 检查模块版本信息 (System Check)

例：

```
amqmsysn
```

输出对话框，显示所有模块的版本信息和 MSCS Cluster 的信息。

高可用性 (High-Avalability，其它平台)

其它平台的 HA 方案由 MQ SupportPac 提供。

- MC63 AIX HACMP
- MC66 HP ServiceGuard
- MC68 Compaq TruCluster

疑问交易 (In-Doubt Transaction)

dspmqtrn 显示疑问交易

[-e]	Externally Coordinated，WMQ 作为资源管理器已经应答了交易管理器 2PC 中的 Phase 1 (Prepare to commit) 请求，但是还没有收到 Phase 2 (Commit or Rollback) 请求。
[-i]	Internally Coordinated，WMQ 作为交易管理器已经向资源管理器发出了 2PC 中的 Phase 1 (Prepare to commit) 请求，但是还没有回答。
[-m QMgrName]	缺省为系统的缺省队列管理器

备注：

如果既不指定 -e 也不指定 -i，则 Externally 和 Internally 都输出。由于在测试环境中无法在 Phase 1 和 Phase 2 之间插入代码，很难模拟出 2PC 中断的情况。

rsvmqtrn 解决疑问交易

{-a {-b -c -r RMID} Transaction}	
-a	All。Commit 所有的 Internally Coordinated In-Doubt Transaction。向所有的资源管理器发 Phase 2 Commit 请求
-b	Backout 指定的 Externally Coordinated In-Doubt Transaction。不接受资源管理器的 Phase 2 请求
-c	Commit 指定的 Externally Coordinated In-Doubt Transaction。不接受资源管理器的 Phase 2 请求
-r RMID	Commit Internally Coordinated In-Doubt Transaction。忽略某一路的资源管理器，对其它的资源管理器发 Phase 2 Commit 请求。只有先在 Queue Manager 配置中删去此资源管理器项，才

Transaction	可以用这个方法 交易号
-m QMgrName	队列管理器

备注：

由于在测试环境中无法在 Phase 1 和 Phase 2 之间插入代码，很难模拟出 2PC 中断的情况。

消息 (Message)

amqsput 往队列中放消息 (Server 程序)

QName	队列名
[QMgrName]	缺省为系统的缺省队列管理器

例：

```
amqsput Q QM
```

amqsputc 往队列中放消息 (Client 程序)

QName	队列名
[QMgrName]	缺省为系统的缺省队列管理器

例：

```
amqsputc Q QM
```

amqsget 从队列中取消息 (Server 程序)

QName	队列名
[QMgrName]	缺省为系统的缺省队列管理器

例：

```
amqsget Q QM
```

amqsgetc 从队列中取消息 (Client 程序)

QName	队列名
[QMgrName]	缺省为系统的缺省队列管理器

例：

```
amqsgetc Q QM
```

工具 (Utility)

runmqsc 脚本命令服务器 (Run MQSC)

`[-e | -v | -w WaitTime [-x]]`

`-e` export。MQSC 命令不再拷贝到输出 report 中

`-v` verify。检查命令的合法性,不执行。只有本地有效,不可与 `-w` 或 `-x` 共用。

`-w` 命令在另一个队列管理器上执行,需要定义有相应的通道和传输队列单位: 秒。min=1, max=999,999。
在指定的时间内等待命令的执行结果,每一条命令都会通过缺省队列管理器,以 PCF 的形式通过缺省队列管理器送达对方的 SYSTEM.ADMIN.COMMAND.QUEUE 中,执行结果放入 SYSTEM.MQSC.REPLY.QUEUE 中。如果未指定缺省队列管理器,runmqsc -w 会失败。

`-x` 目标队列管理器是 MQ for z/OS,和 `-w` 共用。MQSC 命令要符合 z/OS 的命令格式。

`[QMgrName]` 目标队列管理器 (不一定是直连的缺省队列管理器)

例:

```
runmqsc -v QM < QM.tst
runmqsc -v QM < QM.in > QM.out
```

mqrc 原因码查询 (MQ Reason Code)

- 通过一个或一段原因码查询

```
mqrc [-a] <retcode>
mqrc -r <retcode> [-a]
mqrc -R [-f <first> -l <last>] [-a]
```
- 通过一个或一段消息代号查询

```
mqrc [-a] AMQ<number>
mqrc -m [AMQ]<number> [-a]
mqrc -M -f <first> -l <last> [-a]
```
- 通过原因码符号查询

```
mqrc <symbol>
mqrc -s <symbol>
```
- 显示帮助界面

```
mqrc -h 显示帮助界面
mqrc -v 显示版本信息
```
- 通用选项

```
-a 在所有的级别中寻找
-f <number> 起始数
-l <number> 终止数
```

备注：

- 原因码如果是从 1 到 9 打头，则认为是十进制数，如果是 0x 打头，则认为是十六进制数。
- 如果一段原因码或消息代号未指明始末，则按最大范围查询。
- 如果一段原因码或消息代号查询中有错误，则在查询结果前会有以下标识。
 - ? 原因码不存在
 - ! 原因码与消息的查询级别不一不致

例：

```
mqrc 2058
      2058 0x0000080a MQRC_Q_MGR_NAME_ERROR
mqrc 0x0000080a
      2058 0x0000080a MQRC_Q_MGR_NAME_ERROR
mqrc MQRC_Q_MGR_NAME_ERROR
      2058 0x0000080a
mqrc AMQ9001
      36865 0x00009001 rrcI_NORMAL_CHANNEL_END
MESSAGE:
通道程序正常结束.
EXPLANATION:
通道程序 '<insert one>' 正常结束.
ACTION:
不必响应.
```

amqfirst MQ 第一步，仅 Window 平台

例：

```
amqfirst
```

amqapi API 试验程序，仅 Windows 平台

例：

```
amqapi
```

amqpccard MQI 明信片程序，仅 Windows 平台

例：

```
amqpccard
```

amqmtbrn MQ Task Bar，仅 Windows 平台

例：

amqmtbrn

amqmpjese 准备 MQ 向导，仅 Windows 平台

-l <file>	创建 Log 记录文件，系统会把“Prepare WebSphere MQ Wizard”的执行过程追加到指定的文件中。如果未指明路径，则使用 WebSphere MQ Data 路径，如果未指明文件名，则使用 AMQMJPSE.LOG
-r	重置 (Reset) MQSeriesService 用户帐号。“Prepare WebSphere MQ Wizard”在第一次被执行的时候会自动创建一个帐户 MUSR_MQADMIN，具有一定的设置与权限，MQSeriesService 部件都用这个帐户运行。在网络发生变化后，也许“准备 WebSphere MQ 向导”会要求重新运行，以 domain 用户替代，所以需要重置用户帐号。
-s	安静 (silent) 安装模式，无输入输出。
-p <file>	从指定文件中读取参数。如果未指明路径，则使用 WebSphere MQ Data 路径，如果未指明文件名，则使用 AMQMJPSE.INI。文件中应该有以下配置段落： [Services] [DefaultConfiguration] [RemoteAdministration] 事实上，安静安装模式中使用的就是 AMQJPSE.INI
-m <file>	产生安装报告文件。产生 Microsoft System Management Server (SMS) 的文件 .MIF，如果没有指明路径，则使用 WebSphere MQ Data 路径，如果没有指明文件名，则使用 AMQMJPSE.MIF。

amqmgse MQ 缺省配置

例：

amqmgse

amqinfon MQ 信息中心文档 (MQ Info Center)

例：

amqinfon

crtmqcvx 创建数据转换程序框架 (Create Conversion)

SourceFile TargetFile

例：

crtmqcvx MyFormat.h MyFormat.c

```

MyFormat.h
-----
typedef struct MyFormat
{
    MQLONG      l;
    MQCHAR      c;
    MQBYTE      b;
    MQLONG      la [3];    // MQLONG  Array
    MQCHAR      ca [5];    // MQCHAR  Array
    MQBYTE      ba [6];    // MQBYTE  Array
} MYFORMAT;

```

```

MyFormat.c
-----
MQLONG ConvertMyFormat (
    PMQBYTE *in_cursor,
    PMQBYTE *out_cursor,
    PMQBYTE in_lastbyte,
    PMQBYTE out_lastbyte,
    MQHCONN hConn,
    MQLONG  opts,
    MQLONG  MsgEncoding,
    MQLONG  ReqEncoding,
    MQLONG  MsgCCSID,
    MQLONG  ReqCCSID,
    MQLONG  CompCode,
    MQLONG  Reason)
{
    MQLONG ReturnCode=MQRC_NONE;
    ConvertLong(1);          /* l */
    ConvertChar(1);          /* c */
    ConvertByte(1);          /* b */
    AlignLong();
    ConvertLong(3);          /* la */
    ConvertChar(5);          /* ca */
    ConvertByte(6);          /* ba */
Fail:
    return(ReturnCode);
}

```

runmqdlq 运行死信队列处理器 (Run Dead-Letter Queue Handler)

[QName [QMgrName]]

QName 死信队列，缺省为队列管理器的缺省死信队列

QMgrName 队列管理器，缺省为系统缺省队列管理器

例：

```
runmqdlq SYSTEM.DEAD.LETTER.QUEUE QM
```

runmqchi 运行通道初始化程序 (Run Channel Initiator)

[-q InitiationQName] 缺省为 SYSTEM.CHANNEL.INITQ

[-m QMgrName] 缺省为系统的缺省队列管理器

例：

```
runmqchi -q SYSTEM.CHANNEL.INITQ QM
```

runmqchl 运行通道 (Run Channel)

-c ChannelName 通道名

[-m QMgrName] 缺省为系统的缺省队列管理器

例：

```
runmqchl -c C_C.S -m QM
```

dspmqls 显示对象对应的文件名 (Display Files)

[-m QMgrName] 缺省为系统的缺省队列管理器

[-t ObjectType]

nl 或 namelist 名字列表

prcs 或 process 进程

q 或 queue 队列

ql 或 qlocal 本地队列

qa 或 qalias 别名队列

qr 或 qremote 远程队列

qm 或 qmodel 模型队列

qmgr 队列管理器

syncfile 通道同步文件

ctlg 或 catalog 对象目录

authinfo 授权信息，SSL 通道

*或 all 全部

GenericObjName 对象名字，可以用通配符

例：

```
dspmqls -m QM -t queue Q
```

WebSphere MQ 显示 MQ 文件

```
QLOCAL                      Q
```


CA	用 IE 中的 Certificate Store CA
ROOT	用 IE 中的 Certificate Store ROOT
MY	用 IE 中的 Certificate Store MY
[-m qmgr]	队列管理器，此选项会根据队列管理器的 SSLKEYR 定位 Certificate Store
[-h]	Host，引用本机的 Certificate Store。Windows 系统允许两套 Certificate Store。分别在 HKEY_CURRENT_USER 和 HKEY_LOCAL_MACHINE。
[-a [handle] [-p filename -z password]]	Add，增加证书 .p12，.pfx
[-s filename]	CA certificate 文件 .der，.pb7，.cer
[-x handle]	Export，导出证书
[-d handle]	Assign，指定工作证书
[-r handle]	Remove，删除证书
[-u]	Unassign，取消工作证书
[-l]	List，列出所有证书

例：

```
amqmcert -l // 用 MQSSLKEYR
amqmcert -l -k C:\mqm\key
amqmcert -l -m QM
amqmcert -l -k ROOT -h
amqmcert -l -k CA -h
amqmcert -l -k MY
amqmcert -m QM -a 102
amqmcert -a 102
amqmcert -a 4 -k MY -m QM
amqmcert -m QM -d 122
amqmcert -d 123
amqmcert -a -p mqper.pfx -z password
amqmcert -a -s mqcacert.cer
```

ffstsummary FFST文件摘要 (FFST Summary)

FFST 文件名格式为 AMQpid.seq.FDC

ffstsummary 会对当前目录下的所有 FFST 文件进行分析，输出摘要

例：

```
ffstsummary
    AMQ03624.0.FDC 2002/11/30 23:00:00 amqzxma0 3624 1 AD004001 adhOpen
arcE_OBJECT_MISSING OK
    AMQ03624.0.FDC 2002/11/30 23:00:00 amqzxma0 3624 1 AQ168001 aqpReadData
arcE_OBJECT_MISSING OK
    AMQ03624.0.FDC 2002/11/30 23:00:00 amqzxma0 3624 1 AQ143008
```

```
acqAccessQHeader arcE_OBJECT_DAMAGED OK
```

mqaxlev 显示 Code Level

有时候在碰上疑难杂症时，IBM Service Team 会要求调查所安装 WebSphere MQ 的 Code Level。找到运行的 mqax200.dll，用 mqaxlev 命令即可查看。

例：

```
cd C:\Program Files\IBM\WebSphere MQ\bin
mqaxlev mqax200.dll > CodeLevel.txt
```

amqrfdm 查询 MQ Cluster Repository

[-m QMgrName] 缺省为系统的缺省队列管理器

例：

```
amqrfdm -m QM
#) Summary
q) Print Cluster Queues
s) Print Subscriptions
m) Print Cluster Queue Managers
o) Output entire cache
a) Print Free Areas
r) Print Registrations
d) Dump Area of repository
t) Browse Transmission Queue
D) Disconnect
f) Set Filters (Off)
+) Set Details level
Enter option character                      Q - Quit
```

amquiregn Registry 值列表工具

fname 文件名，内含需要列出的键值

例：

```
amquiregn -?
ERROR: unable to open file -?, fopen return code was 123
-----
-----
Environment follows....
=::=::\
=C:=C:\
```

```

=D:=C:\Program Files\IBM\WebSphere MQ\bin
=E:=E:\WMQ\MQ_Samples\UserExit
=ExitCode=00000001
ALLUSERSPROFILE=C:\Documents and Settings\All Users
ANT_HOME=C:\jakarta-ant-1.5.1
MQ_JAVA_DATA_PATH=C:\Program Files\IBM\WebSphere MQ
MQ_JAVA_INSTALL_PATH=C:\Program Files\IBM\WebSphere MQ\Java

```

amqmdain MQ 服务控制命令，仅 Windows 平台

用于配置和管理 WebSphere MQ Service。相当于 Windows MQ Services snap-in

start QMgrName	启动队列管理器服务
end QMgrName	停止队列管理器服务
auto QMgrName	将队列管理器服务设置为自动启动
manual QMgrName	将队列管理器服务设置为手工启动
crtlsr QMgrName LsrParams	创建队列管理器中的监听器
LsrParams	监听器参数，与 runmqclsr 命令的参数相同。比如 -t trptype -p port
crttrm QMgrName QueueName	创建触发监控器
crtchi QMgrName InitQName	创建通道初始器
status	如果没有指定参数，则输出 WMQ Service 的状态
QMgrName	输出指定队列管理器 Service 的状态
all	输出所有队列管理器
cstmig filename	导入 Custom Service 定义文件。
	文件格式为 Command Name, Start Command, End Command, Flags, Reserved。
	例：PubSub Broker, strmqbrk -p blue.queue.manager, endmqbrk -i -m blue.queue.manager, SUFFIX ROOT STARTUP SHUTDOWN COMMAND, 1
regsec	确保 Registry 中的 security permissions 正确
spn qmgrname	Set 或 Unset 队列管理器的 Service Principal
set	
unset	
reg [QMgrName *] RegParams	修改 Windows Registry 中 RegParams 指定的配置段落 (stanza) 及 WebSphere MQ 属性参数。
	-c add -s stanza -v attribute=value
	-c remove -s stanza -v [attribute *]
	-c display -s stanza -v [attribute *]
*	配置段落 (stanza) 可以是
	ApiExitCommon\name
	ApiExitTemplate\name
	ACPI

	AllQueueManagers
	Channels
	DefaultQueueManager
	LogDefaults
QMgrName	配置段落 (stanza) 可以是
	XAResourceManager\name
	ApiExitLocal\name
	Channels
	ExitPath
	Log
	QueueManagerStartup
	TCP
	LU62
	SPX
	NetBios

例：

```
amqmdain status
5724-B41 (C) Copyright IBM Corp. 1994, 2002. ALL RIGHTS RESERVED.
MQSeries 服务状态是：就绪

amqmdain status QM
5724-B41 (C) Copyright IBM Corp. 1994, 2002. ALL RIGHTS RESERVED.
MQSeries 服务状态是：就绪
队列管理器 'QM' 状态为：正在运行
0 队列管理器 (正在运行)
1 命令服务器 (已停止)
```

amqmsrvn COM 服务器，仅 Windows 平台

amqmsrvn 由所有使用这个 COM 组件的 client 程序共享。比如：WebSphere MQ Services snap-in, Alert Monitor Task Bar, WebSphere MQ Service。amqmsrvn 供所有的交互的和非交互的应用使用，它必须被一个特殊的用户 (MUSR_MQADMIN) 启动，这个用户的口令是随机产生的。

如果要改变用户名，可以先自己创建一个用户 NEW_NAME，然后用 amqmsrvn 设置，例：

```
amqmsrvn -user <domain\>NEW_NAME -password <password>
```

如果要恢复到 MUSR_MQADMIN，则：

```
amqmjpse -r
```


附录 MQSC 命令一览表

RUNMQSC

WebSphere MQ 为所有的开放平台 (Windows 和 UNIX) 提供了相同的管理界面。通过交互命令行的方式执行 WebSphere MQ Script Commands (MQSC)。这个通用的管理工具称为 RUNMQSC。

执行脚本

可以把要执行的命令先放入脚本文件中,用重定向的方法将其输入,输出也可以定向到另一个结果文件中。脚本文件通常用 `tst` 文件后缀。执行的时候,RUNMQSC 会首先回显命令,然后显示执行结果。所以,结果文件中既有命令的执行结果,也有命令本身。

```
runmqsc QM < test.tst > test.out
```

抑制回显

可以用 `-e` 选项抑制回显命令,这样命令输出中就只有执行结果了。

```
runmqsc -e QM < mqscfile.in > myscfile.out
```

检验脚本

可以用 `-v` 选项对脚本文件中的命令进行语法检查而不执行。在真正执行命令脚本之前用 `-v` 选项检查一遍是一个好习惯。

```
runmqsc -v QM < myprog.in > myprog.out
```

远程管理

RUNMQSC 可以进行管理,这时 RUNMQSC 工具首先连接本地缺省队列管理器,再通过事先配置并连接好的通道,把命令发往指定的队列管理器,命令执行结果原路返回。由于涉及队列管理器之间的通信,可以用 `-w` 选项设定超时时间 (秒),如果在规定的时间内命令结果没有返回,则执行失败。

```
runmqsc -w 100 QM < mqscfile.in > myscfile.out
```

批处理

开放平台都提供操作系统的批处理方式,如 UNIX 中的 SHELL 脚本,Windows 中的 BAT 文件。它与 RUNMQSC 脚本文件结合就可以灵活自如地进行一些自动配置。例:创建并启动队列管理器,添加一个本地队列,放入三条消息。

```
// batch.sh 或 batch.bat
```

```
crtmqm QM
strmqm QM
runmqsc QM < config.tst
amqsput Q QM < data.txt
```

```
// config.tst
DEFINE QLOCAL (Q)
```

```
// data.txt
string 1
string 2
string 3
```

如果在批处理中需要有变量代入，在 UNIX 中可以利用 EOF 将批处理命令和 MQSC 配置命令写在一起。例：

```
export QMgrName=QM
export QName=Q
export Data=string

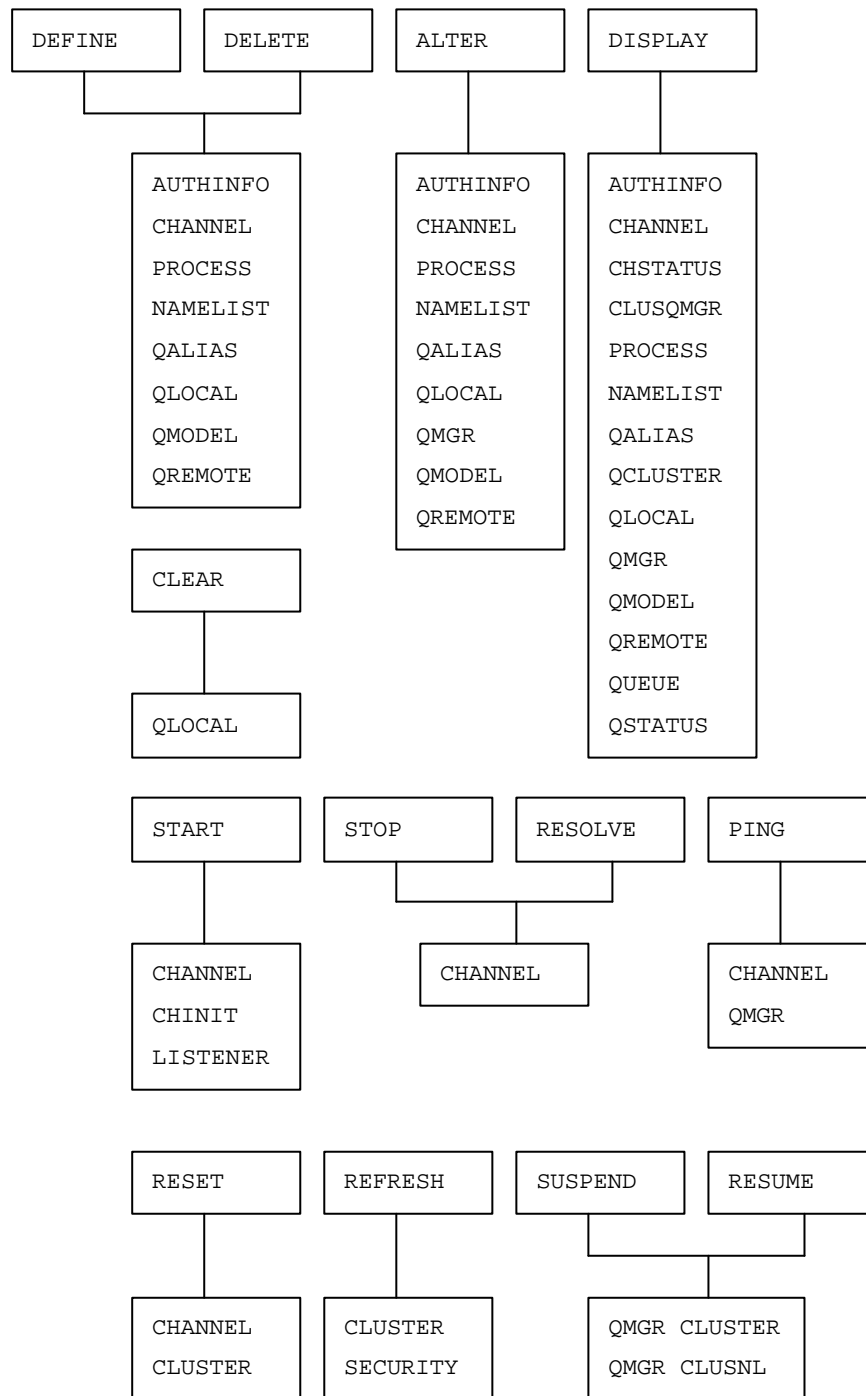
// batch.sh
crtmqm $QMGrName
strmqm $QMGrName
runmqsc $QMGrName << EOF
    DEFINE QLOCAL ($QName)
EOF
amqsput $QName $QMGrName << EOF
    $Data 1
    $Data 2
    $Data 3
EOF
```

对于 RUNMQSC 本身的语法，请参见相关附录 (WebSphere MQ 命令一览表)。

MQSC 命令

结构图

所有的 MQSC 命令开始的语法格式都是动作 (Action) + 对象 (Object) + 参数 (Parameter)。开放平台的 MQSC 命令结构如下：



DEFINE 命令用来定义创建各种 WebSphere MQ 对象，在创建的时候可以指定对象的各
种属性，DELETE 用来删除它们。ALTER 可以修改这些对象的属性，此外，ALTER 可以修
改队列管理器的属性。DISPLAY 用来显示各对象的属性、队列和通道的工作状态、群集中
的队列和队列管理器。

START 用来启动通道及通道相关的对象，比如通道初始化程序、端口监听器。STOP 正
相反，用来停止通道。有时通道会处于不确定 (In-doubt) 状态，表示不知道对方是否收到
该消息，两边对消息是否到达的判断可能不一致。RESOLVE 命令可以在两侧手工地将状态
调整到一致。PING 命令经常用来检查己方的命令服务器是否启动，也可以检查对方的队
列管理器或端口监听器是否启动，还可以检查对方的通道定义是否正确。

RESET 可以重置通道计数，也可以将队列管理器从群集中删除。REFRESH 可以刷新 WebSphere MQ 安全性高速缓存，也可以刷新群集中队列管理器的群集信息。SUSPEND 和 RESUME 都是针对群集的操作，前者将队列管理器隔离出去，后者将其恢复回来。

MQSC 的动作可以有简写方式，如下表：

全称	简称	说明
DEFINE	DEF	创建定义对象
DELETE	DELETE	删除对象
ALTER	ALT	修改对象属性
DISPLAY	DIS	显示对象属性
CLEAR	CLEAR	清除本地队列内容
START	STA	启动通道、通道启动程序、监听器
STOP	STOP	停止通道
RESOLVE	RESOLVE	手工提交或回滚通道传送事务
PING	PING	测试队列管理器或通道是否工作
RESET	RESET	重置通道或群集
REFRESH	REF	刷新群集或安全信息
SUSPEND	SUSPEND	暂挂队列管理器，从群集中隔离出去
RESUME	RESUME	恢复队列管理器，回到群集中去

对象也可以有简写方式，如下表：

全称	简称	说明
QMGR	QMGR	队列管理器
QUEUE	Q	队列
QLOCAL	QL	本地队列
QREMOTE	QR	远程队列
QALIAS	QA	别名队列
QMODEL	QM	模型队列
QSTATUS	QS	队列状态
CHANNEL	CHL	通道
CHSTATUS	CHS	通道状态
CHINIT	CHI	通道初始程序
LISTENER	LSTR	监听器
NAMELIST	NL	名称列表
PROCESS	PRO	进程定义
AUTHINFO	AUTHINFO	认证信息
QCLUSTER	QC	群集队列
CLUSQMGR	CLUSQMGR	群集队列管理器
CLUSTER	CLUSTER	群集
CLUSNL	CLUSNL	群集名称列表
SECURITY	SECURITY	安全信息

DEFINE

DEFINE 命令用来定义创建 WebSphere MQ 对象，这些对象可以是队列 (QLOCAL、QREMOTE、QALIAS、QMODEL)，通道 (SDR、SVR、RCVR、RQSTR、CLNTCONN、SVRCONN、CLUSSDR、CLUSRCVR)，进程定义，名称列表，认证信息。

DEFINE QLOCAL

DEFINE QLOCAL(q_name)

[CLUSTER(cluster_name)]	队列所属的群集名
[CLUSNL(namelist_name)]	队列所属的群集列表名
[SCOPE(QMGR CELL)]	队列属于队列管理器，还是属于 DCE Cell 工作单元
[DESCR(string)]	描述字符串
[LIKE(qlocal_name)]	未指定的属性缺省值从指定队列拷贝。如果不设定 LIKE，系统从 SYSTEM.DEFAULT.LOCAL.QUEUE 中拷贝
[NOREPLACE REPLACE]	未指定的属性是否用缺省值替换
[USAGE(NORMAL XMITQ)]	正常的本地队列，还是传输队列
[DISTL(NO YES)]	Distribution List。对方的队列管理器是否支持分发表，这个属性由 MCA 设定，在传输队列 (USAGE= XMITQ) 中有效
[DEFPRTY(integer)]	Default Priority。消息的缺省优先级
[DEFPERSIST(NO YES)]	Default Persistence。消息的缺省持久性
[DEFBIND(NOTFIXED OPEN)]	Default Binding。缺省绑定方式，对 MQOPEN (MQOO_BIND_AS_Q_DEF)有效
[DEFSOPT(EXCL SHARED)]	Default Share Option。缺省 MQPUT 共享方式，对 MQOPEN (MQOO_INPUT_AS_Q_DEF) 有效
[NOSHARE SHARE]	MQGET 共享方式
[GET(ENABLED DISABLED)]	允许或禁止 MQGET
[PUT(ENABLED DISABLED)]	允许或禁止 MQPUT
[MAXDEPTH(integer)]	Max Depth。队列的最大深度
[MAXMSGL(integer)]	Max Message Length。队列中消息的最大长度
[MSGDLVSQ(PRIORITY FIFO)]	Message Delivery Sequence。消息排序方式
[RETINTVL(integer)]	Retrieval Interval。队列的生命周期，单位：小时。
[BOQNAME(string)]	Backout Queue Name。回滚超量后的目标队列
[BOTHRESH(integer)]	Backout Thresh。回滚计数阈值
[NOHARDENBO HARDENBO]	Harden Backout。是否固化回滚计数
[QDPHIEV(ENABLED DISABLED)]	Queue Depth High Event。设置队列深度高性能事件开关
[QDEPTHHI(integer)]	Queue Depth High。队列高位阈值百分比，在 QDPHIEV= ENABLED 时生效，当队列深度上升至大于等于该值时，出现性能事件
[QDPLOEV(ENABLED DISABLED)]	Queue Depth Low Event。设置队列深度低性能事件开关
[QDEPTHLO(integer)]	Queue Depth Low。队列低位阈值百分比，在 QDPLOEV = ENABLED 时生效，当队列深度下降至小于该值时，出现

	性能事件
[QDPMAXEV(ENABLED DISABLED)]	Queue Depth Max Event。设置队列满性能事件开关
[QSVCI EV(NONE HIGH OK)]	Queue Service Event。设置队列服务间隔开关
[QSVCI NT(integer)]	Queue Service Interval。队列服务间隔定时，单位：毫秒。 在 QSVCI EV= HIGH 或 OK 时生效
[NOTRI GGER TRI GGER]	触发器开关
[TRI GTYPE(FI RST EV ERY DEPTH NONE)]	触发类型 (Trigger Type)
[I NITQ(string)]	触发初始化队列名
[P R O C E S S(string)]	触发进程定义名
[T R I G D P T H(integer)]	触发深度，当 TRI GTYPE = DEPTH 时生效
[T R I G M P R I(integer)]	Trigger Message Priority。触发消息优先级，只有大于等于 该优先级的消息才会引起触发
[T R I G D A T A(string)]	触发数据字符串，会映射到 MQTM 或 MQTMC2 数据结构的 TriggerData 域
创建本地队列。	

DEFINE QREMOTE

DEFINE QREMOTE(q_name)	
[CLUSTER(cluster_name)]	队列所属的群集名
[CLUSNL(namelist_name)]	队列所属的群集列表名
[SCOPE(QMGR CELL)]	队列属于队列管理器，还是属于 DCE Cell 工作单元
[DESCR(string)]	描述字符串
[LIKE(qlocal_name)]	未指定的属性缺省值从指定队列拷贝。如果不设定 LIKE， 系统从 SYSTEM.DEFAULT.REMOTE.QUEUE 中拷贝
[NOREPLACE REPLACE]	未指定的属性是否用缺省值替换
[DEFPRTY(integer)]	消息的缺省优先级
[DEFPSIST(NO YES)]	消息的缺省持久性
[DEFBIND(NOTFIXED OPEN)]	缺省绑定方式，对 MQOPEN MQOO_BIND_AS_Q_DEF) 有效
[PUT(ENABLED DISABLED)]	允许或禁止 MQPUT
[RQMNAME(string)]	目标队列管理器名
[RNAME(string)]	目标队列名
[XMITQ(string)]	传输队列名，如同不指定，则先找 RQMNAME 同名的传 输队列，再找队列管理器的缺省传输队列，如果都找不到， 去死信队列
创建远程队列。	

DEFINE QALIAS

DEFINE QALIAS(q_name)	
[CLUSTER(cluster_name)]	队列所属的群集名
[CLUSNL(namelist_name)]	队列所属的群集列表名
[SCOPE(QMGR CELL)]	队列属于队列管理器，还是属于 DCE Cell 工作单元

[DESCR(string)]	描述字串
[LIKE(qlocal_name)]	未指定的属性缺省值从指定队列拷贝。如果不设定 LIKE , 系统从 SYSTEM.DEFAULT.ALIAS.QUEUE 中拷贝
[NOREPLACE REPLACE]	未指定的属性是否用缺省值替换
[DEFPRTY(integer)]	消息的缺省优先级
[DEFPSIST(NO YES)]	消息的缺省持久性
[DEFBIND(NOTFIXED OPEN)]	缺省绑定方式, 对 MQOPEN (MQOO_BIND_AS_Q_DEF) 有效
[GET(ENABLED DISABLED)]	允许或禁止 MQGET
[PUT(ENABLED DISABLED)]	允许或禁止 MQPUT
[TARGQ(string)]	目标队列名
创建别名队列。	

DEFINE QMODEL

DEFINE QMODEL(q_name)	
[DESCR(string)]	描述字串
[LIKE(qlocal_name)]	未指定的属性缺省值从指定队列拷贝。如果不设定 LIKE , 系统从 SYSTEM.DEFAULT.MODEL.QUEUE 中拷贝
[NOREPLACE REPLACE]	未指定的属性是否用缺省值替换
[USAGE(NORMAL XMITQ)]	正常的本地队列, 还是传输队列
[DISTL(NO YES)]	对方的队列管理器是否支持分发列表, 这个属性由 MCA 设定, 在传输队列 (USAGE= XMITQ) 中有效
[DEFPRTY(integer)]	消息的缺省优先级
[DEFPSIST(NO YES)]	消息的缺省持久性
[DEFTYPE(TEMPDYN PERMDYN)]	缺省动态生成方式, 是生成临时动态队列还是永久动态队列
[DEFSOPT(EXCL SHARED)]	缺省 MQPUT 共享方式, 对 MQOPEN (MQOO_INPUT_AS_Q_DEF) 有效
[NOSHARE SHARE]	MQGET 共享方式
[GET(ENABLED DISABLED)]	允许或禁止 MQGET
[PUT(ENABLED DISABLED)]	允许或禁止 MQPUT
[MAXDEPTH(integer)]	队列的最大深度
[MAXMSGL(integer)]	队列中消息的最大长度
[MSGDLVSQ(PRIORITY FIFO)]	消息排序方式 (Message Delivery Sequence)
[RETINTVL(integer)]	队列的生命周期, 单位: 小时
[BOQNAME(string)]	回滚超量后的目标队列
[BOTHRESH(integer)]	回滚计数阈值
[NOHARDENBO HARDENBO]	是否固化回滚计数
[QDPHIEV(ENABLED DISABLED)]	设置队列深度高性能事件开关
[QDEPTHHI(integer)]	队列高位阈值百分比, 在 QDPHIEV= ENABLED 时生效, 当队列深度上升至大于等于该值时, 出现性能事件
[QDPLOEV(ENABLED DISABLED)]	设置队列深度低性能事件开关
[QDEPTHLO(integer)]	队列低位阈值百分比, 在 QDPLOEV = ENABLED 时生效,

	当队列深度下降至小于该值时，出现性能事件
[QDPMAEV(ENABLED DISABLED)]	设置队列满性能事件开关
[QSVCI EV(NONE HIGH OK)]	设置队列服务间隔开关
[QSVCI NT(integer)]	队列服务间隔定时，单位：毫秒。在 QSVCI EV= HIGH 或 OK 时生效
[NOTRI GGER TRI GGER]	触发器开关
[TRI GTYPE(FI RST E V E R Y DE PTH NONE)]	触发类型 (Trigger Type)
[I NITQ(string)]	触发初始化队列名
[P R O C E S S(string)]	触发进程定义名
[TRI GD PTH(integer)]	触发深度，当 TRI GTYPE = DE PTH 时生效
[TRI GMPRI(integer)]	触发消息优先级，只有大于等于该优先级的消息才会引起触发
[TRI GDATA(string)]	触发数据字符串，会映射到 MQTM 或 MQTMC2 数据结构的 TriggerData 域
创建模型队列。	

DEFINE CHANNEL

DEFINE CHANNEL(channel_name)

CHLTYPE(SDR SVR RCVR RQSTR CLNTCONN SVRCONN CLUSSDR CLUSRCVR)	通道类型，会使后面的参数选项有所不同
CONNAME(string)	连接参数。对于 SDR、RQSTR、CLNTCONN、CLUSSDR、CLUSRCVR，此参数选项是必需的
XMITQ(string)	传输对于名。对于 SDR、SVR，此参数选项是必需的
[CLUSTER(cluster_name)]	队列所属的群集名
[CLUSNL(namelist_name)]	队列所属的群集列表名
[DESCR(string)]	描述字符串
[LIKE(channel_name)]	未指定的属性缺省值从指定通道拷贝。如果不设定 LIKE，系统从 SYSTEM.DEF.*中拷贝
[NOREPLACE REPLACE]	未指定的属性是否用缺省值替换
[TRPTYPE(LU62 TCP NETBIOS SPX)]	通信协议，通道双方设置应该相同
[CONNAME(string)]	对方连接参数，与 TRPTYPE 指定的通信协议有关
[LOCLADDR(string)]	本地连接参数，格式与 CONNAME 类似
[TPNAME(string)]	当 TRPTYPE=LU62 时，指定 TP 名
[USERID(string)]	当 TRPTYPE=LU62 时，指定 SNA Session 用户名
[PASSWORD(string)]	当 TRPTYPE=LU62 时，指定 SNA Session 用户口令
[MODENAME(string)]	当 TRPTYPE=LU62 时，指定通信模式名
[NETPRTY(integer)]	通道优先级，当到达目标队列的路由有多种选择的时候，系统会选取优先级最高的通道。取值 0-9，只对 CLUSRCVR 有效
[NPMSPEED(NORMAL FAST)]	非持久消息的速度，如果设置成 FAST，则该通道称为快速通道 (FastPath) 非持久消息可以不参与等待批量消息而直接发送
[PUTAUT(DEF CTX)]	对于消息通道，表示消息放入目标队列时权限检查所针对

	的用户名。对于 MQI 通道，表示 MQ API 操作时权限检查所针对的用户名。DEF (Default) 表示用缺省用户。CTX (Context) 表示用消息上下文中的 UserIdentifier 域中的用户名
[QMNAME(string)]	只对 CLNTCONN 有效，表示所在的队列管理器名
[SEQWRAP(integer)]	最大序列号，取值 100-999,999,999。通道在传递消息时序列号会不断增加，如果达到这个数字，则重新从 1 开始
[BATCHINT(integer)]	批量间隔，单位毫秒。取值 0-999,999,999
[BATCHSZ(integer)]	批量大小，取值 0-9,999
[BATCHHB(integer)]	批量心跳间隔，单位毫秒。取值 0-999,999。0 表示不发生指不发生指心跳
[HBINT(integer)]	心跳间隔，单位秒。取值 0-999,999。0 表示不发生心跳
[DISCINT(integer)]	自动断连间隔，单位秒。取值 0-999,999。缺省为 6000 秒
[KAJINT(integer)]	保持连接间隔，单位秒。取值 0-99,999。0 表示不使用
[CONVERT(NO YES)]	发送时是否对消息进行格式转换
[MAXMSGL(integer)]	通道能接受的最大消息长度，指的是物理消息长度。0 表示使用队列管理器的最大消息长度设置
[MCANAME(string)]	MCA 名字
[MCATYPE(PROCESS THREAD)]	MCA 类型，可以选进程或线程
[MCAUSER(string)]	MCA 用户名
[RCVDATA(string)]	Receive Exit 用户数据
[RCVEXIT(string)]	Receive Exit 名，由程序名和入口函数名组成，可以有多个，用逗号隔开。例：'C:\Exit1.dll (Func1), C:\Exit2.dll (Func2)'
[SENDDATA(string)]	Send Exit 用户数据
[SENDEXIT(string)]	Send Exit 名
[MSGDATA(string)]	Message Exit 用户数据
[MSGEXIT(string)]	Message Exit 名
[SCYDATA(string)]	Security Exit 用户数据
[SCYEXIT(string)]	Security Exit 名
[MRDATA(string)]	Message Retry Exit 用户数据
[MREXIT(string)]	Message Retry Exit 名
[MRRTY(integer)]	Message Retry 重试次数，取值 0-999,999,999
[MRTMR(integer)]	Message Retry 重试间隔，单位毫秒，取值 0-999,999,999
[SHORTRTY(integer)]	短重试次数，取值 0-999,999,999
[SHORTTMR(integer)]	短重试间隔，单位秒，取值 0-999,999
[LONGRTY(integer)]	长重试次数，取值 0-999,999,999
[LONGTMR(integer)]	长重试间隔，单位秒，取值 0-999,999
[SSLAUTH(REQUIRED OPTIONAL)]	是否需要从 SSL Client 取证书
[SSLCIPH(string)]	SSL 加密算法，如果为空，表示不用 SSL 协议。通道双方必须指定相同的加密算法
[SSLPEER(string)]	数字认证时用的 DN，如：'CN=QM1, OU=SWG, O=IBM, L=Shanghai, S=Shanghai, C=CN'。如果设置了该值，则在 SSL 握手时会检查对方证书的 DN 是否与该值相同

1. Cluster (CLUSTER, CLUSNL) 只对 CLUSSDR、CLUSRCVR 有效

2. Message Retry (MREXIT, MRDATA, MRRTY, MRTMR) 只对 RCVR、RQSTR、CLUSRCVR 有效
 3. Channel Retry (SHORTRTY, SHORTTMR, LONGRTY, LONGTMR) 只对 SDR、SVR、CLUSSDR、CLUSRCVR 有效
 4. Batch (BATCHINT, BATCHSZ) 表示消息的批量发送, BATCHINT 只对 SDR、SVR、CLUSSDR、CLUSRCVR 有效, BATCHSZ 除此之外还可以对 RCVR、RQSTR 有效。
 5. HeartBeat (BATCHEHB, HBINT) 表示通道心跳, BATCHEHB 及与此相关的 DISCINT 只对 SDR、SVR、CLUSSDR、CLUSRCVR 有效, HBINT 除此之外还可以对 RCVR、RQSTR 有效。
 6. SSLCAUTH 只对 RCVR、SVRCONN、CLUSRCVR、SVR、RQSTR 有效
 7. MCA (MCATYPE, MCAUSER, MCANAME) 表示在通道上设定 Message Channel Agent 的缺省参数, 可以对 MCA 的工作方式和安全认证产生影响。MCANAME 只对 SDR、SVR、RQSTR、CLUSSDR 有效, MCATYPE 除此之外还可以对 CLUSRCVR 有效, MCAUSER 的有效范围更大, 还包括 RCVR 和 SVRCONN
- 创建通道。

DEFINE PROCESS

DEFINE PROCESS(process_name)

[APPLICID(string)]	要触发的进程名字, 最长 256 字节。如果 APPLTYPE=CICS, 则 APPLICID 指的是 CICS 交易名。会映射到 MQTM 或 MQTMC2 数据结构的 ApplId 域
[APPLTYPE(CICS DEF DOS OS2 UNIX WINDOWS WINDOWSNT NOTESAGENT OS400 integer)]	要触发的进程类型
[DESCR(string)]	描述字串
[LIKE(process_name)]	未指定的属性缺省值从指定进程定义拷贝。如果不设定 LIKE, 系统从 SYSTEM.DEFAULT.PROCESS 中拷贝
[NOREPLACE REPLACE]	未指定的属性是否用缺省值替换
[ENVADATA(string)]	环境数据, 会映射到 MQTM 或 MQTMC2 数据结构的 EnvData 域
[USERDATA(string)]	用户数据, 会映射到 MQTM 或 MQTMC2 数据结构的 UserData 域

创建进程定义。

DEFINE NAMELIST

DEFINE NAMELIST(namelist_name)

[DESCR(string)]	描述字串
[LIKE(namelist_name)]	未指定的属性缺省值从指定名字列表拷贝。如果不设定 LIKE, 系统从 SYSTEM.DEFAULT.NAMELIST 中拷贝
[NOREPLACE REPLACE]	未指定的属性是否用缺省值替换
[NAMES(string)]	用逗号隔开的名字串, 比如: Name1, Name2, Name3。最长 48 字节

创建名称列表。

DEFINE AUTHINFO

DEFINE AUTHINFO(authinfo_name)	
AUTHTYPE(CRLLDAP)	认证方式，这里只能是 CRLLDAP
CONNNAME (string)	LDAP Server 的工作机器名或 IP 地址及端口号，例： CONNNAME('hostname(nnn)'), nnn 缺省为 389
[DESCR(string)]	描述字符串
[LIKE(authinfo_name)]	未指定的属性缺省值从指定认证信息拷贝。如果不设定 LIKE，系统从 SYSTEM.DEFAULT.AUTHINFO.CRLLDAP 中拷贝
[NOREPLACE REPLACE]	未指定的属性是否用缺省值替换
[LDAPPWD (string)]	LDAP 用户名
[LDAPUSER (string)]	LDAP 用户口令
创建认证信息。	

DELETE

DELETE QLOCAL

DELETE QLOCAL(q_name) [NOPURGE PURGE]
删除本地队列。

- NOPURGE 如果队列中有已提交的消息，停止删除。
- PURGE 不管队列中是否有已提交的消息，删除队列。消息也被删除。

DELETE QREMOTE

DELETE QREMOTE(q_name)
删除远程队列。

DELETE QALIAS

DELETE QALIAS(q_name)
删除别名队列。

DELETE QMODEL

DELETE QMODEL(q_name)
删除模型队列。

DELETE CHANNEL

DELETE CHANNEL(channel_name) [CHLTABLE(QMGRTBL | CLNTTBL)]
删除通道。

- CHLTABLE (QMGRTBL) 从队列管理器的通道定义表中删除，表中不含 CLNTCONN 通道。缺省值
- CHLTABLE (CLNTTBL) 从 CLNTCONN 的通道定义表中删除

DELETE PROCESS

DELETE PROCESS(process_name)
删除进程定义。

DELETE NAMELIST

DELETE NAMELIST(namelist_name)
删除名称列表。

DELETE AUTHINFO

DELETE AUTHINFO(authinfo_name)
删除认证信息。

ALTER

ALTER QMGR

ALTER QMGR

[FORCE]

[DESCR (string)]

[CCSID(integer)]

[CHAD(DISABLED | ENABLED)]

[CHADEXIT(string)]

[CLWLEXIT(string)]

[CLWLDATA(string)]

[CLWLLEN(integer)]

[DEADQ(string)]

[DEFXMITQ(string)]

[MAXHANDS(integer)]

[MAXMSGL(integer)]

[MAXUMSGS(integer)]

[LOCALEV(ENABLED | DISABLED)]

[REMOTEEV(ENABLED | DISABLED)]

[INHIBTEV(ENABLED | DISABLED)]

[AUTHOREV(ENABLED | DISABLED)]

[STRSTPEV(ENABLED | DISABLED)]

[CHADEV(DISABLED | ENABLED)]

[PERFMEV(ENABLED | DISABLED)]

[REPOS(string)]

[REPOSNL(string)]

[SSLCRLNL(string)]

[SSLKEYR(string)]

[TRIGINT(integer)]

修改队列管理器属性。

如果指定了缺省传输队列 DEFXMITQ, 且有应用打开远程队列并使用到 DEFXMITQ。该次修改对此应用立即生效, 则需要 FORCE 选项, 否则出错

描述字串。也可以在 crtqm -c 时指定

Character Code Set Id。字符集

设置通道自动定义开关

通道自动定义出口程序 (Channel Auto Definition User Exit)

群集负载用户出口程序 (Cluster Workload User Exit)

群集负载用户出口用户数据

能通过群集负载用户出口程序路由的最长消息字节数

死信队列名。也可以在 crtqm -u 时指定

缺省传输队列名。也可以在 crtqm -d 时指定

一个任务 (通常指一个队列管理器连接线程) 中最多同时执有的打开对象的句柄数量。也可以在 crtqm -h 时指定

消息的最大长度

最多的非提交 (Uncommitted) 的消息数量。也可以在 crtqm -x 时指定

Local Event。设置本地事件开关

Remote Event。设置远程事件开关

Inhibit Event。设置违禁事件开关

Authorization Event。设置权限事件开关

Start/Stop Event。设置启停事件开关

Channel Auto Definition Event。设置通道自动定义事件开关

Performance Event。设置性能事件开关

群集名, 队列管理器在该群集中担任配置库

群集列表名, 队列管理器该列表的所有群集中担任配置库

SSL 认证吊销列表名

SSL 认证库

Trigger Interval。当队列的触发类型 TRIGTYPE=FIRST 时, TRIGINT 指定时效间隔。也可以在 crtqm -t 时指定

ALTER QLOCAL

ALTER QLOCAL(q_name)

[FORCE]

其它选项同 DEFINE QLOCAL

如果有应用程序打开队列进行操作, 该次修改对此应用立即生效, 则需要 FORCE 选项

修改本地队列属性。

ALTER QREMOTE

ALTER QREMOTE(q_name)

[FORCE]

如果有应用程序打开队列进行操作，该次修改对此应用立即生效，则需要 FORCE 选项

其它选项同 DEFINE QREMOTE
修改远程队列属性。

ALTER QALIAS

ALTER QALIAS(q_name)

[FORCE]

如果有应用程序打开队列进行操作，该次修改对此应用立即生效，则需要 FORCE 选项

其它选项同 DEFINE QREMOTE
修改别名队列属性。

ALTER QMODEL

ALTER QMODEL(q_name)

选项同 DEFINE QMODEL

修改模型队列属性。

ALTER CHANNEL

ALTER CHANNEL(channel_name)

选项同 DEFINE CHANNEL

修改通道属性。

ALTER PROCESS

ALTER PROCESS(process_name)

选项同 DEFINE PROCESS

修改进程定义属性。

ALTER NAMELIST

ALTER NAMELIST(namelist_name)

选项同 DEFINE NAMELIST

修改名称列表属性。

ALTER AUTHINFO

ALTER AUTHINFO(authinfo_name)

选项同 DEFINE AUTHINFO
修改认证信息属性。

DISPLAY

DISPLAY QMGR

DISPLAY QMGR

[ALL]	显示所有属性，缺省值
[CRDATE]	创建日期，DEFINE QMGR 时自动设置
[CRTIME]	创建时间，DEFINE QMGR 时自动设置
[ALTDATE]	上一次修改日期，修改对象时自动更新
[ALTTIME]	上一次修改时间，修改对象时自动更新
[CMDLEVEL]	版本号，WebSphere MQ 5.3 的 CMDLEVEL=530
[COMMANDQ]	命令队列，缺省为 SYSTEM.ADMIN.COMMAND.QUEUE
[MAXPRTY]	消息的最大优先级，缺省为 9
[PLATFORM]	操作系统
[QMID]	队列管理器 ID，由系统内部产生，唯一标识该队列管理器
[QMNAME]	队列管理器名
[SYNCPT]	队列管理器是否支持同步 (SyncPoint)
其它选项同 ALTER QMGR	
显示队列管理器属性。	

DISPLAY QUEUE

DISPLAY QUEUE(generic_q_name)

[TYPE(QALIAS QCLUSTER QLOCAL QMODEL QREMOTE)]	输入参数，队列类型
[QTYPE]	输出参数，队列类型
[ALL]	显示所有属性，缺省值
[CRDATE]	创建日期，DEFINE QMGR 时自动设置
[CRTIME]	创建时间，DEFINE QMGR 时自动设置
[ALTDATE]	上一次修改日期，修改对象时自动更新
[ALTTIME]	上一次修改时间，修改对象时自动更新
[CLUSDATE]	群集队列对于本地队列管理器生效的日期
[CLUSTIME]	群集队列对于本地队列管理器生效的时间
[CLUSQMGR]	群集队列的宿主队列管理器名
[CLUSQT]	群集队列类型
[CLUSINFO]	除了从本地队列管理器中查找队列外，也从该队列管理器的群集信息 (Repository) 中查找。通常欲显示群集队列信

	息需要加上 CLUSINFO 选项
[CURDEPTH]	队列当前深度
[IPPROCS]	以读方式打开的句柄数
[OPPROCS]	以写方式打开的句柄数
[QMID]	队列管理器 ID，由系统内部产生，唯一标识该队列管理器
其它选项参见 DEFINE QLOCAL、QREMOTE、ALIAS、MODEL	
显示队列属性。	

DISPLAY QLOCAL

DISPLAY QLOCAL(generic_q_name)

[ALL]	显示所有属性，缺省值
[CRDATE]	创建日期，DEFINE QMGR 时自动设置
[CRTIME]	创建时间，DEFINE QMGR 时自动设置
[ALTDATE]	上一次修改日期，修改对象时自动更新
[ALTTIME]	上一次修改时间，修改对象时自动更新
[CLUSDATE]	群集队列对于本地队列管理器生效的日期
[CLUSTIME]	群集队列对于本地队列管理器生效的时间
[CLUSQMGR]	群集队列的宿主队列管理器名
[CLUSQT]	群集队列类型
[CLUSINFO]	除了从本地队列管理器中查找队列外，也从该队列管理器的群集信息 (Repository) 中查找。通常欲显示群集队列信息需要加上 CLUSINFO 选项
[CURDEPTH]	队列当前深度
[IPPROCS]	以读方式打开的句柄数
[OPPROCS]	以写方式打开的句柄数
[QMID]	队列管理器 ID，由系统内部产生，唯一标识该队列管理器
[QTYPE]	输出参数，队列类型
[TYPE]	与 QTYPE 相同
其它选项同 DEFINE QLOCAL	
显示本地队列属性。	

DISPLAY QREMOTE

DISPLAY QREMOTE(generic_q_name)

[ALL]	显示所有属性，缺省值
[CRDATE]	创建日期，DEFINE QMGR 时自动设置
[CRTIME]	创建时间，DEFINE QMGR 时自动设置
[ALTDATE]	上一次修改日期，修改对象时自动更新
[ALTTIME]	上一次修改时间，修改对象时自动更新
[CLUSDATE]	群集队列对于本地队列管理器生效的日期
[CLUSTIME]	群集队列对于本地队列管理器生效的时间
[CLUSQMGR]	群集队列的宿主队列管理器名
[CLUSQT]	群集队列类型

[CLUSINFO]	除了从本地队列管理器中查找队列外，也从该队列管理器的群集信息 (Repository) 中查找。通常欲显示群集队列信息需要加上 CLUSINFO 选项
[QMID]	队列管理器 ID，由系统内部产生，唯一标识该队列管理器
[QTYPE]	输出参数，队列类型
[TYPE]	与 QTYPE 相同
其它选项同 DEFINE QREMOTE	
显示远程队列属性。	

DISPLAY QALIAS

DISPLAY QALIAS(generic_q_name)

[ALL]	显示所有属性，缺省值
[CRDATE]	创建日期，DEFINE QMGR 时自动设置
[CRTIME]	创建时间，DEFINE QMGR 时自动设置
[ALTDATE]	上一次修改日期，修改对象时自动更新
[ALTTIME]	上一次修改时间，修改对象时自动更新
[CLUSDATE]	群集队列对于本地队列管理器生效的日期
[CLUSTIME]	群集队列对于本地队列管理器生效的时间
[CLUSQMGR]	群集队列的宿主队列管理器名
[CLUSQT]	群集队列类型
[CLUSINFO]	除了从本地队列管理器中查找队列外，也从该队列管理器的群集信息 (Repository) 中查找。通常欲显示群集队列信息需要加上 CLUSINFO 选项
[QMID]	队列管理器 ID，由系统内部产生，唯一标识该队列管理器
[QTYPE]	输出参数，队列类型
[TYPE]	与 QTYPE 相同
其它选项同 DEFINE QALIAS	
显示别名队列属性。	

DISPLAY QMODEL

DISPLAY QMODEL(generic_q_name)

[ALL]	显示所有属性，缺省值
[CRDATE]	创建日期，DEFINE QMGR 时自动设置
[CRTIME]	创建时间，DEFINE QMGR 时自动设置
[ALTDATE]	上一次修改日期，修改对象时自动更新
[ALTTIME]	上一次修改时间，修改对象时自动更新
[QTYPE]	输出参数，队列类型
[TYPE]	与 QTYPE 相同
其它选项同 DEFINE QMODEL	
显示模型队列属性。	

DISPLAY QSTATUS

DISPLAY QSTATUS(q_name) TYPE(QUEUE)

[ALL]	显示所有状态，缺省值
[CURDEPTH]	队列的当前深度
[IPPROCS]	以读方式打开的句柄数
[OPPROCS]	以写方式打开的句柄数
[UNCOM]	是否有未提交的消息。取值 YES 或 NO

DISPLAY QSTATUS(q_name) TYPE (HANDLE)

[ALL]	显示所有状态
[OPENTYPE (ALL INPUT OUTPUT)]	ALL 显示所有打开的句柄信息 INPUT 只显示读方式打开的句柄信息 OUTPUT 只显示写方式打开的句柄信息
[PID]	进程号，Process ID
[TID]	线程号，Thread ID
[APPLTAG]	应用程序标签。取值 z/OS 批处理作业名，TSO 用户名，CICS 应用名，IMS Region 名，通道初始化作业名，OS/400 作业名，UNIX 或 Windows 进程名
[APPLTYPE]	应用程序类型。取值 BATCH，RRSBATCH，CICS，IMS，CHINIT，SYSTEM，USER。其中 CHINIT 表示通道初始化程序，SYSTEM 表示 MQ 系统程序，USER 表示用户程序
[CHANNEL]	与句柄相关的通道名
[CONNAME]	与句柄相关的连接参数
[USERID]	句柄执有者的用户名
[BROWSE]	句柄是否以浏览方式打开
[INPUT]	句柄是否以读方式打开
[OUTPUT]	句柄是否以写方式打开
[INQUIRE]	句柄是否以查询方式打开
[SET]	句柄是否以设置方式打开
显示队列状态。	

例：

```
dis qs (Q)
```

AMQ8450: 显示队列状态详细信息。

QUEUE(Q)	IPPROCS(1)
OPPROCS(2)	CURDEPTH(5)
UNCOM(NO)	

```
dis qs (Q) type (handle) opentype (output) all
```

AMQ8450: 显示队列状态详细信息。

QUEUE(Q)	PID(3940)
APPLTAG(D:\WMQ\bin\amqsput.exe)	TID(1)
APPLTYPE(USER)	CHANNEL()
CONNAME()	BROWSE(NO)

INPUT(NO)	INQUIRE(NO)
OUTPUT(YES)	SET(NO)
USERID(chenyux@T40XP)	

DISPLAY CHANNEL

DISPLAY CHANNEL(generic_channel_name)

[ALL]	显示所有属性，缺省值
[TYPE(ALL SDR SVR RCVR RQSTR CLNTCONN SVRCONN)]	输入参数，通道类型
[CHLTYPE]	输出参数，通道类型
其它选项同 DEFINE CHANNEL	
显示通道属性。	

DISPLAY CHSTATUS

DISPLAY CHSTATUS(generic_channel_name)

[CURRENT SAVED]	CURRENT 只显示当前通道初始化程序执有的通道状态，通常指连通或准备连通的通道状态，缺省值
	SAVED 显示所有定义的通道状态
[ALL]	显示以下所有的状态选项，缺省值
[CONNAME(connection_name)]	连接参数
[XMITQ(q_name)]	传输队列
[CURMSGs]	当前传送批次中的消息序号，对发送端而言是发送的消息序号，对接收端而言是接收的消息序号。一旦整批消息提交或回滚，该值自动归零。通道两端的值应该相同
[CURLUWID]	当前 LUW 号，当前传送批次的交易号。最终会对整批消息一起提交或回滚
[CURSEQNO]	当前消息序号，随着消息的传送该值会不断增加，直到超过通道定义的最大序列号 SEQWRAP 后自动重新从 1 开始，或者手工 RESET CHANNEL SEQNUM。通道两端的值应该相同
[LSTLUWID]	上一个 LUW 号，通道启动时置为全零。如果有错，则为出错点的 LUW 号
[LSTSEQNO]	上一个序号。如果有错，则为出错点的序号
[INDOUBT]	通道是否处于不确定状态
[STATUS]	状态。取值 STARTING, BINDING, INITIALIZING, RUNNING, STOPPING, RETRYING, PAUSED, STOPPED, REQUESTING。
	STARTING 有启动请求，但尚未开始启动过程
	BINDING 开始启动过程，处于握手协议中
	INITIALIZING 通道初始化程序正在试图启动通道
	RUNNING 运行状态，连接成功
	STOPPING 有停止请求，但尚未开始停止过程

	RETRYING	上一次连接失败 ,MCA 间隔一断时间后会重试连接
	PAUSED	等待消息重试间隔
	STOPPED	已停止，用户停止了通道或重试次数达到限制
[BATCHES]	REQUESTING	本地的 RQSTR 通道与远程 MCA 之间通信自通道启动以来传送的批次数量
[BATCHSZ]		自通道启动时约定的批次大小
[BUF SRCVD]		自通道启动以来收到的传输缓冲区数量，只含控制消息包
[BUF SENT]		自通道启动以来发送的传输缓冲区数量，只含控制消息包
[BYT SRCVD]		自通道启动以来收到的字节数，含应用和控制消息包
[BYT SENT]		自通道启动以来发送的字节数，含应用和控制消息包
[CHSTADA]		通道启动日期
[CHSTATI]		通道启动时间
[HBINT]		自通道启动时约定的心跳间隔
[JOBNAM]		作业名。在 UNIX 和 Windows 中由 PID 和 TID 组成，比如 00000D1400000FC0 表示 PID=3348 (D14)，TID=4032 (FC0)。在发送端对应的进程为 runmqchl，接收端对应的进程为 amqrmppa。
[LOCLADDR]		本地连接参数
[LONGRTS]		还剩下的长重试次数
[SHORTRTS]		还剩下的短重试次数
[LSTMSGDA]		上一个消息日期
[LSTMSGTI]		上一个消息时间
[MCASTAT]		MCA 的运行状态。取值 RUNNING 或 NOT RUNNING
[MSGS]		自通道启动以来传送的消息数量
[NPMSPEED]		是普通通道或是高速通道
[RQMNAME]		远程队列管理器名
[SSLPEER]		SSL 协议用的 DN 名
[STOPREQ]		用户是否有停止通道请求，取值 YES 或 NO
显示通道状态。		

例：

```
dis chstatus (C) all
```

AMQ8417: 显示通道状态细节。

CHANNEL(C)	XMITQ(QX)
CONNAME(127.0.0.1 (1415))	CURRENT
CHLTYPE(SDR)	INDOUBT(NO)
LSTSEQNO(0)	LSTLUWID(0000000000000000)
CURMSG(0)	CURSEQNO(0)
CURLUWID(F29BEB3F10000401)	STATUS(RUNNING)
LSTMSGTI()	LSTMSGDA()
MSGS(0)	BYTSENT(244)
BYT SRCVD(244)	BATCHES(4)
BATCHSZ(50)	HBINT(300)

NPMSPEED (FAST)	CHSTATI (10.58.33)
CHSTADA (2003-12-27)	BUFSSSENT (5)
BUFSRCVD (5)	LONGRTS (999999999)
SHORTRTS (10)	JOBNAME (00000D1400000FC0)
MCASTAT (RUNNING)	STOPREQ (NO)
LOCLADDR (127.0.0.1 (3707))	SSLPEER ()
RQMNAME (QM2)	

DISPLAY PROCESS

DISPLAY PROCESS(generic_process_name)

[ALL]	显示所有属性，缺省值
[ALTDAT]	上一次修改日期，修改对象时自动更新
[ALTTIME]	上一次修改时间，修改对象时自动更新
其它选项同 DEFINE PROCESS	
显示进程定义属性。	

DISPLAY NAMELIST

DISPLAY NAMELIST(generic_namelist_name)

[ALL]	显示所有属性，缺省值
[ALTDAT]	上一次修改日期，修改对象时自动更新
[ALTTIME]	上一次修改时间，修改对象时自动更新
[NAMCOUNT]	名称列表属性 NAMES 串中的名称数量，名称用逗号隔开
其它选项同 DEFINE NAMELIST	
显示名称列表属性。	

DISPLAY AUTHINFO

DISPLAY AUTHINFO(generic_authinfo_name)

[ALL]	显示所有属性，缺省值
[ALTDAT]	上一次修改日期，修改对象时自动更新
[ALTTIME]	上一次修改时间，修改对象时自动更新
其它选项同 DEFINE AUTHINFO	
显示认证信息属性。	

DISPLAY CLUSQMGR

DISPLAY CLUSQMGR(generic_Q_Mgr_name)

[ALL]	显示所有属性，缺省值
[ALTDAT]	上一次修改日期，修改对象时自动更新
[ALTTIME]	上一次修改时间，修改对象时自动更新
[CLUSDATE]	群集队列对于本地队列管理器生效的日期

[CLUSTIME]	群集队列对于本地队列管理器生效的时间
[CLUSTER]	该队列管理器所属的群集名
[QMID]	队列管理器 ID，由系统内部产生，唯一标识该队列管理器
[QMTYPE]	群集队列管理器的类型，取值 REPOS 或 NORMAL，表示配置库队列管理器或普通队列管理器
[CHANNEL]	该队列管理器与配置库之间的通道名
[STATUS]	通道状态
[SUSPEND]	是否被暂挂起，取值 YES 或 NO
其它选项同 DEFINE CHANNEL	
显示群集队列管理器属性。	

DISPLAY QCLUSTER

DISPLAY QCLUSTER(generic_q_name)

[ALL]	显示所有属性，缺省值
[ALTDAT]	上一次修改日期，修改对象时自动更新
[ALTTIME]	上一次修改时间，修改对象时自动更新
[CLUSDATE]	群集队列对于本地队列管理器生效的日期
[CLUSTIME]	群集队列对于本地队列管理器生效的时间
[CLUSTER]	群集队列所属的群集名
[CLUSQMGR]	群集队列的宿主队列管理器名
[CLUSQT]	群集队列类型
[DESCR(string)]	描述字串
[QTYPE]	队列类型
[TYPE]	与 QTYPE 相同
[PUT(ENABLED DISABLED)]	允许或禁止 MQPUT
[DEFPRTY(integer)]	Default Priority。消息的缺省优先级
[DEFPSIST(NO YES)]	Default Persistence。消息的缺省持久性
[DEFBIND(NOTFIXED OPEN)]	Default Binding。缺省绑定方式，对 MQOPEN (MQOO_BIND_AS_Q_DEF)有效

CLEAR

CLEAR QLOCAL

CLEAR QLOCAL(q_name)

清除本地队列中的所有消息。

START

START CHANNEL

START CHANNEL(channel_name)

该命令适用于除 CLNTCONN 之外的所有通道。对握手协议中的主动方，比如 SDR、CLUSDR，使用该命令可以启动通道并进行连接。对握手协议中的被动方，比如 RCVR、SVRCONN、CLUSRCVR，使用该命令只能使通道准备好连接，而不能启动通道连接。

START CHINIT

START CHINIT

[INITQ(string)]

初始化队列名

启动通道初始化进程

START LISTENER

START LISTENER

[TRPTYPE(NETBIOS | LU62 | SPX | TCP)]通信协议

启动监听器

STOP

STOP CHANNEL

STOP CHANNEL(channel_name)

[MODE(QUIESCE | FORCE | TERMINATE)] 停止模式

QUIESCE	允许本批次传输完毕后，关闭通道
---------	-----------------

FORCE 立即结束传输，关闭通道，有可能使正在传输的消息出现不确定 (In doubt) 状态

TERMINATE 功能上等同于 FORCE ,同时可以终止通道的工作进程或线程

[STATUS(STOPPED INACTIVE)]	设定 SDR 或 SVR 通道停止后的状态
-------------------------------	-----------------------

STOPPED 将传输队列属性设为 GET (DISABLED) 及 NOTRIGGER。如果命令中不指定 QMNAME 和 CONNAME, 则 STOPPED 为缺省值

INACTIVE 不改变传输队列属性。如果命令中指定了 QMNAME 和 CONNAME, 则 INACTIVE 为缺省值

[QMNAME(qmname)]

通道所属的队列管理器，该选项可以作为选择通道的匹配条件

[CONNAME(connection name)]

连接参数，该选项可以作为选择通道的匹配条件

停止通道。

RESOLVE

RESOLVE CHANNEL

RESOLVE CHANNEL(channel_name)

ACTION(COMMIT | BACKOUT)

解决通道中处于不确定状态的消息。

手工将通道中处于不确定 (In doubt) 状态的消息提交或回滚。用于 SDR、SVR、CLUSSDR 通道。

PING

PING QMGR

PING QMGR

检查队列管理器的对 SYSTEM.ADMIN.COMMAND.QUEUE 是否有反映 检查命令服务器是否工作正常。

PING CHANNEL

PING CHANNEL(channel_name)

[DATALEN(16 | integer)]

PING 数据包的大小，16-32,768 字节。

检查对方的队列管理器或端口监听器是否启动，也可以检查对方的通道定义是否正确。但不检查通道的通性状态。换句话说，PING CHANNEL 只检查通道能否连连通，而不检查目前是否连通。

RESET

RESET CHANNEL

RESET CHANNEL(channel_name)

[SEQNUM(1 | integer)]

重设通道的消息序号

消息序号，两端的值应该相同

RESET CLUSTER

RESET CLUSTER(cluster_name)]

ACTION(FORCEREMOVE)

QMNAME(string) | QMID(string)

[QUEUES(NO | YES)]

强行将队列管理器从群集中删除

指定队列管理器。QMNAME 为队列管理器名，QMID 为队列管理器 ID

是否被删除的队列管理器中的群集队列也被一并删除

REFRESH

REFRESH CLUSTER

REFRESH CLUSTER(cluster_name)]
[REPOS(NO | YES)]

是否连同配置库信息一并删除重建

NO 保留本地群集信息中的配置库队列管理器名，如果该结点自身就是配置库队列管理器，则会记住群集中的其它配置库队列管理器，其余的信息都被删除重建。通常用于队列管理器上的群集信息发生混乱时手工更新。

YES 在本地群集信息中连群集中配置库队列管理器名也被删除重建。用于非配置库队列管理器。如果该结点自身就是配置库队列管理器，则不能用 REPOS (YES)，应该先将结点改成非配置库，刷新后，再改回来。通常用于群集中有配置库队列管理器的地位发生变化，需要手工更新。

该命令首先删除所有本地已有的群集信息，然后通过与附近的配置库队列管理器交换信息来重建本地的群集信息。通常用于非配置库队列管理器。

REFRESH SECURITY

REFRESH SECURITY
刷新队列管理器中所有的权限配置信息

SUSPEND

SUSPEND QMGR CLUSTER

SUSPEND QMGR CLUSTER (cluster_name)
[MODE(QUIESCE | FORCE)]

QUIESCE 通知群集中的其它队列管理器不要再向该队列管理器发送消息

FORCE 除此之外，断开该队列管理器的所有接收端通道

暂时将队列管理器从群集中挂起。

SUSPEND QMGR CLUSNL

SUSPEND QMGR CLUSNL (namelist_name)
[MODE(QUIESCE | FORCE)]

QUIESCE 通知群集中的其它队列管理器不要再向该队列管理器发送消息

FORCE 除此之外，断开该队列管理器的所有接收

端通道

暂时将队列管理器从群集列表中挂起。

RESUME

RESUME QMGR CLUSTER

RESUME QMGR CLUSTER (cluster_name)

将队列管理器恢复回到群集中。

RESUME QMGR CLUSNL

RESUME QMGR CLUSNL (namelist_name)

将队列管理器恢复回到群集列表中。

参考书目

GC34-6073-01 WebSphere MQ for Windows Quick Beginnings
GC34-6076-01 WebSphere MQ for AIX Quick Beginnings
GC34-6077-01 WebSphere MQ for HP-UX Quick Beginnings
GC34-6075-01 WebSphere MQ for Solaris Quick Beginnings
GC34-6078-00 WebSphere MQ for Linux Quick Beginnings
SC34-6068-01 WebSphere MQ System Administration Guide
SC34-6059-03 WebSphere MQ Intercommunication
SC34-6079-01 WebSphere MQ Security
SC34-6069-02 WebSphere MQ Event Monitoring
SC34-6060-03 WebSphere MQ Programmable Command Formats and Administration Interface
SC34-6061-02 WebSphere MQ Queue Manager Clusters
SC34-6065-00 WebSphere MQ Application Messaging Interface
SC34-6064-03 WebSphere MQ Application Programming Guide
SC34-6062-03 WebSphere MQ Application Programming Reference
SC34-6067-02 WebSphere MQ Using C++
SC34-6066-01 WebSphere MQ Using Java
SC34-6134-00 WebSphere MQ Using the Component Object Model Interface
GC34-6058-01 WebSphere MQ Clients
SC34-6275-00 WebSphere MQ Extended Transactional Clients
GC34-6057-01 WebSphere MQ Messages
SC34-6055-03 WebSphere MQ Script (MQSC) Command Reference

GC34-5269-09 MQSeries Publish/Subscribe User's Guide

