AMQP协议与RabbitMQ

- 讲师：pansir

简介

RabbitMQ是一个消息代理(消息系统的媒介)，它是一个通用的消息平台，提供消息发送和接收功能，并保证消息传输安全。

作用

1. 软件、应用相互连接和扩展，组成更大的应用
2. 用户设备和数据连接
3. 消息系统将**发送**和**接收**分离，来实现应用程序的**异步**和**解耦**

应用场景

1. 数据投递，非阻塞操作，推送通知
2. 实现发布/订阅
3. 异步处理
4. 工作队列

技术亮点

- 可靠性：持久化，投递确认(ack回执)，发布者证实，高可用机制
- 灵活路由：
    1. 消息在到达队列前，通过**交换机**路由
    2. RabbitMQ提供多种内置交换机类型：direct/fanout/topic等
    3. 多种交换机可组合使用
    4. 甚至可以自定义交换机类型
- 集群：支持多节点，聚合在一起作为一个独立逻辑代理
- 联合：提供联合模型，因为服务器比集群需要更多的松散和非可靠链接
- 高可用队列：同个集群，队列可被镜像到多个机器，down机消息仍安全
- 多协议：支持多种协议
- 广泛客户端：所有编程语言都适配
- 可视化管理工具：http://10.211.55.5:15672
- 追踪：可追踪消息系统的异常行为
- 插件系统：提供多种插件，也可以自己写插件

AMQP协议

AMQP(高级消息队列协议)，属于一种网络协议。支持客户端(application)和消息中间件代理(broker)之间通信

角色

- 消息代理：负责接收消息，根据路由规则，再发送消息
- 生产者(发布者)：生产消息
- 消费者：消费消息

由于AMQP是网络协议，所以生产者，消费者，代理可在不同设备上

模型

生产者(发消息)=>交换机(路由寻址分发)=>队列=>代理(投递)=>消费者(读取)

- 生产者发布消息时，可以指定各种**消息属性**，部分可被代理使用，其他则完全不透明，只能被消费者使用
- 因为网络不可靠且消费者也可能处理失败，所以引入**消息确认**(message **ack**nowledgements)，即：消费者给消息代理回执
- ack回执可以自动，也可以由消费者执行
- 启用ack时，代理收到回执后才会删除队列中的消息
- 消息路由失败时，生产者可配置参数来处理这些特殊情况

可编程协议

- 队列，交换机和绑定统称为AMQP实体
- AMQP的实体和路由规则是由应用本身定义的，而不是由消息代理定义
- 例如：声明队列和交换机，定义他们之间的绑定，订阅队列等等关于协议本身的操作

交换机

交换机类型(指定路由算法) + 绑定(bindings) => 交给队列

- 交换机类型：
    1. 直连交换机(direct)：amq.direct
    2. 扇形交换机(fanout)：amq.fanout
    3. 主题交换机(topic)：amq.topic
    4. 头交换机(headers)：amq.match
- 重要属性：
    1. name：
    2. durability：消息代理重启后，交换机是否还存在
    3. auto-delete：当所有与之绑定的消息队列都完成了对此交换机的使用后，删除它
    4. arguments：依赖代理本身
- 交换机状态：
    1. 持久(durable)：消息代理重启后依然存在
    2. 暂存(transient)：消息代理重启后需要重新声明

        补充：并不是所有队列都需要持久化队列

默认交换机

本质是直连交换机，每个新建的队列(**queue**)自动绑定到默认交换机，绑定的路由健(**routing key**)和队列同名

例如：声明一个"hello"队列，AMQP代理自动将它绑定到默认交换机，绑定的路由健名称也是"hello"。所以，当携带"hello"路由健的消息被发送到默认交换机时，会被路由到"hello"队列。效果就是：默认交换机可以直接将消息投给队列。

直连交换机

直连交换机根据携带的路由健(routing key)将消息投递到对应队列。

- 工作流程：
  1. 将一个队列绑定到某个交换机，同时赋予该"绑定"一个路由健(routing key)
  2. 当一个携带该路由健的消息被发到交换机，交换机会把它路由到同样绑定该路由健的队列
- 工作细节：循环分发任务给多个workers，所以AMQP中，消息的**负载均衡**是发生在**消费者**之间，而不是队列之间

python实现生产者

```
import pika

# 声明鉴权信息
credentials = pika.PlainCredentials("admin", "admin")
# 跟本地机器的代理建立了连接。如果你想连接到其他机器的代理上，需要把代表本地的localhost改为指定的名字或IP地址
connection = pika.BlockingConnection(
    pika.ConnectionParameters(
        host="10.211.55.5",
        port=5672,
        virtual_host="admin",
        credentials=credentials,

    )
)
channel = connection.channel()

# 确认队列是存在，若不存在则创建
# 创建一个名为"queue_dcs"的队列用来将消息投递进去
# 1. 要根据队列来类型来配置参数，如果队列本身
# channel.queue_declare(queue='queue_dcs')
# 2. 如果队列类型支持持久化，必须在此处进行声明
channel.queue_declare(queue='queue_dcs', durable=True)

# 在RabbitMQ中，消息是不能直接发送到队列中的，这个过程需要通过交换机（exchange）来进行
# 使用由空字符串表示的默认交换机即可，空字符串代表默认或者匿名交换机：消息将会根据指定的routing_key分发到指定的队列
```

```
# 默认交换机比较特别，它允许我们指定消息究竟需要投递到哪个具体的队列中，队列名字需要在
routing_key参数中指定
channel.basic_publish(
    # exchange='ex_dcs',
    exchange='',  # 为空也允许
    routing_key='routing_dcs',
    body='Hello World!',
)
print("发送一条消息")
# 安全关闭连接
connection.close()
```

```
 Sent 'Hello World!'
```

python实现消费者

```
import pika

credentials = pika.PlainCredentials("admin", "admin")
connection = pika.BlockingConnection(
    pika.ConnectionParameters(
        host="10.211.55.5",
        port=5672,
        virtual_host="admin",
        credentials=credentials
    )
)
# 建立连接
channel = connection.channel()

channel.queue_declare(queue='queue_dcs', durable=True)


# 为队列定义一个回调（callback）函数
# 当我们获取到消息的时候，Pika库就会调用此回调函数
# 这个回调函数会将接收到的消息内容输出到屏幕上
def callback(ch, method, properties, body):
    print(" [x] Received %r" % body)


# 告诉RabbitMQ这个回调函数将会从名为"queue_dcs"的队列中接收消息
# 这里ack是false，没有回执，所以会一直消费
channel.basic_consume(on_message_callback=callback, queue='queue_dcs',
auto_ack=False)
# 如果ack是true，则会给一个回执，则认定消费成功
channel.basic_consume(on_message_callback=callback, queue='queue_dcs',
auto_ack=True)
```

```
print('等待消息 CTRL+C强制退出')
# 运行一个用来等待消息数据并且在需要的时候运行回调函数的无限循环
channel.start_consuming()
```

扇形交换机

扇形交换机将消息路由给它绑定的所有队列，而不管绑定的路由健。若某个扇形交换机绑定了N个队列，当有消息发到交换机，它会把消息**拷贝分别发送**给这N个队列。即：广播路由

- 使用案例：
    1. 大规模多用户在线游戏，处理排行榜更新等全局事件
    2. 体育新闻网站将比分更新分发给移动客户端
    3. 分发系统广播各种状态和配置更新
    4. 群聊，分发消息给参与群聊的用户
- 先创建好 fanout交换机，然后绑定多个队列，然后通过python往交换机里写数据

```
import pika, random

credentials = pika.PlainCredentials("admin", "admin")
connection = pika.BlockingConnection(
    pika.ConnectionParameters(
        host="10.211.55.5",
        port=5672,
        virtual_host="admin",
        credentials=credentials,

    )
)
channel = connection.channel()
# 声明交换机
channel.exchange_declare(exchange='ex_dcs_fanout', exchange_type='fanout',
durable=True)
# 指定fanout交换机，不用指定routing key，消息会发到多个队列
a = random.randint(1, 99999)
msg = 'Hello World! {}'.format(str(a))
channel.basic_publish(
    exchange='ex_dcs_fanout',
    routing_key='',
    body=msg,
)
print("发送一条消息")
# 安全关闭连接
connection.close()
```

- 同时消费两个队列

```
import pika


def callback(ch, method, properties, body):
    # print(ch, method, properties)
    print("获取到消息 %s" % body)
    # 手动确认，确认方式要匹配
    # ch.basic_ack(delivery_tag=method.delivery_tag)


credentials = pika.PlainCredentials("admin", "admin")
connection = pika.BlockingConnection(
    pika.ConnectionParameters(
        host="10.211.55.5",
        port=5672,
        virtual_host="admin",
        credentials=credentials
    )
)
# 建立连接1
channel = connection.channel()
channel.queue_declare(queue='queue_dcs_fanout', durable=True)
channel.basic_consume(on_message_callback=callback, queue='queue_dcs_fanout',
auto_ack=True)

# 建立连接2
channel2 = connection.channel()
channel2.queue_declare(queue='queue_dcs_fanout2', durable=True)
channel2.basic_consume(on_message_callback=callback,
queue='queue_dcs_fanout2', auto_ack=True)

print('等待消息 CTRL+C强制退出')
# 运行一个用来等待消息数据并且在需要的时候运行回调函数的无限循环
channel.start_consuming()
channel2.start_consuming()
```

```
等待消息 CTRL+C强制退出
```

主题交换机

针对性的选择多个消费者(应用)，即可考虑主题交换机

- 使用案例：
    1. 分发有关于特定地理位置的数据，例如销售点
    2. 由多个工作者（workers）完成的后台任务，每个工作者负责处理某些特定的任务

3. 股票价格更新（以及其他类型的金融数据更新）
4. 涉及到分类或者标签的新闻更新（例如，针对特定的运动项目或者队伍）
5. 云端的不同种类服务的协调
6. 分布式架构/基于系统的软件封装，其中每个构建者仅能处理一个特定的架构或者系统。

topic是多对多，但是生成的时候，必须指定具体的队列，而不能使用模糊匹配

```
import pika, random

credentials = pika.PlainCredentials("admin", "admin")
connection = pika.BlockingConnection(
    pika.ConnectionParameters(
        host="10.211.55.5",
        port=5672,
        virtual_host="admin",
        credentials=credentials,

    )
)
channel = connection.channel()
channel.exchange_declare(exchange="ex_dcs_topic", exchange_type='topic',
durable=True)
# 写第一条
a = random.randint(1, 99999)
msg = 'Hello World! {}'.format(str(a))
channel.basic_publish(
    exchange='ex_dcs_topic',
    # routing_key='',  # 为空，任何队列都收不到
    routing_key='routing.dcs.topic1',
    body=msg,
)
# 写第二条
a = random.randint(1, 99999)
msg = 'Hello World! {}'.format(str(a))
channel.basic_publish(
    exchange='ex_dcs_topic',
    # routing_key='',  # 为空，任何队列都收不到
    routing_key='routing.dcs.topic2',
    body=msg,
)
print("发送2条消息 %s" % msg)
# 安全关闭连接
connection.close()
```

绑定多个routing key后，进行统一消费，支持这种：`routing.dcs.*` 方式消费

```
import pika
```

```
def callback(ch, method, properties, body):
    # print(ch, method, properties)
    print("获取到消息 %s" % body)
    # 手动确认，确认方式要匹配
    # ch.basic_ack(delivery_tag=method.delivery_tag)


credentials = pika.PlainCredentials("admin", "admin")
connection = pika.BlockingConnection(
    pika.ConnectionParameters(
        host="10.211.55.5",
        port=5672,
        virtual_host="admin",
        credentials=credentials
    )
)
# 建立连接1
channel = connection.channel()
# 绑定交换机
channel.exchange_declare(
    exchange='ex_dcs_topic',
    exchange_type='topic',
    durable=True
)
# 获取队列名
result = channel.queue_declare('', exclusive=True)
queue_name = result.method.queue
print(queue_name)
# 循环绑定routing key
# binding_keys = ["routing.dcs.topic1", "routing.dcs.topic2"]
binding_keys = ["routing.dcs.*"]
for binding_key in binding_keys:
    channel.queue_bind(
        exchange='ex_dcs_topic',
        queue=queue_name,
        routing_key=binding_key
    )

channel.basic_consume(on_message_callback=callback, queue=queue_name,
auto_ack=True)

print('等待消息 CTRL+C强制退出')
# 运行一个用来等待消息数据并且在需要的时候运行回调函数的无限循环
channel.start_consuming()
```

小结

- 三类交换机：
    1. fanout ex：不处理路由键，一对多，一台交换机，绑定多个队列，每个都发，可以理解成广播方式
    2. direct ex：处理路由键，一对一，通过：交换机 => 路由键 => 队列 方式，找到队列
    3. topic ex：模糊匹配路由键，例如：`dcs.*` 可以匹配到队列`dcs.send`

头交换机

头信息代替路由健

队列

存储即将被消费的数据。队列在声明(declare)后才能被使用，若不存在，声明时就会创建。若队列存在且属性完全一样，也不会影响原队列；若属性有差异则会报错：406

- 属性：
    1. name：
    2. durable：代理重启后，队列依然窜
    3. exclusive：只被一个连接使用，而且当连接关闭后队列立即删除
    4. auto-delete：当最后一个消费者退订后立即删除
    5. arguments：一些消息代理用来完成类似ttl的某些额外功能

队列名称

1. 应用(application)可以给队列取名
2. 消息代理直接生成队列名，需要指定`name=''`
3. 消息代理会记住最后一次生成当队列名称

队列持久化

1. 持久化队列会存储在磁盘上，当消息代理重启，队列重新加载
2. 暂存队列：重启即丢失

绑定

绑定，是交换机将消息路由给队列所遵循的规则，通过路由健(routing key)来实现

- 比喻：
    1. 队列：上海
    2. 交换机：保安机场
    3. 绑定：机场去上海的路线，可以一条也可以多条
- 路由失败：如果交换机没有绑定到队列，发过来的消息则会销毁或返还给生产者，如何处理由生产者设置消息属性

消费者

消息如果只是存储在队列里是没有任何用处的。被应用消费掉，消息的价值才能够体现。

- 两种消费方式：

    1. 将消息推送给应用：push api
    2. 应用根据需要主动拉消息：pull api
- 术语：
    1. 应用注册了一个消费者
    2. 应用订阅了一个队列
- 一个队列可以注册多个消费者，也可以注册一个独享的消费者(当独享消费者存在，其他队列被排除)

消息确认

2种规范建议：

1. 自动确认模式：消息代理发送给应用后，立即删除
2. 显式确认模式：等应用返回一个ack回执后，再删除消息
    - 收到消息立即发送
    - 将未处理的消息存储后，再发送
    - 等到消息被处理完毕后，再发送

如果一个消费者在尚未发生ack时down机了，那么AMQP代理会将消息重新投递到另外一个消费者；若没有可用消费者，则会**死等**下一个注册到此队列的消费者，再尝试投递。

拒绝消息

消费者可拒绝消息，拒绝消息时需要告诉消息代理如何处理这条消息(销毁还是重新放入队列)。

当只有一个消费者时，拒绝消息=>放入队列，会引起无限循环

预取消息

多个消费者共享队列，可以指定在收到下一个回执前，每个消费者可以接受多少消息

消息属性和荷载

- 消息头：类似http-headers，在消息被发布的时候定义
- 有效荷载(payload)：消息实际携带的数据
    1. 消息代理不检查或修改有效荷载
    2. 消息可以只带属性而不带荷载
    3. 最常用的是json格式
    4. MessagePack将结构化数据 =>(序列化成) json数据
    5. 持久化方式发布，牺牲性能获取健壮性

连接

AMQP连接通常是**长连接**。

1. 基于TCP协议
2. 使用认证机制，并提供TSL(SSL)保护
3. 应用需要优雅的释放掉AMQP连接，而不是直接断开TCP

通道

开启过多TCP连接会消耗过多系统资源，并且使防火墙配置困难。所以引入通道，即：共享同个TCP连接的多个轻量化连接。

1. 应用使用多线程/进程连接，每个线程/进程都会开启一个通道，并且通道不能共享
2. 通道的通讯是隔离的，所以AMQP方法需要携带通道号，这样客户端就可以指定此方法是为哪个通道准确的

虚拟主机

虚拟主机(virtual hosts - vhosts)，用来实现多个隔离环境。当连接建立时，AMQP客户端需要指定使用哪个虚拟主机

日常测试与自动化

基于mq的生产与消费，需要根据实际被测对象到底是生产还是消费，来设计测试用例

消费服务测试

对于tsms业务，下游消费者为tsms_consumer的处理逻辑为：消费mq数据，将消息转发给第三方，然后根据第三方的返回结果，决定是否更新状态。测试的起点，不再是接口调用，而是mq的写入

```
class TestTsmsMqProduce(object):
    uuid = "a1e2ff2e-23c5-11ea-a694-acde48001122"

    @retry(stop=stop_after_attempt(5), wait=wait_random_exponential(2, max=5))
    def db_retry(self, uuid):
        """
        异步数据校验，重试查询数据库
        """
        res = td.tsms_select("send", "consume,status,mobile", uuid=uuid)
        assert res["status"] == "success"

    def test_pro_01(self, mq, td):
        data = {
            "uid": self.uuid,
            "phone": "17134198056",
            "content": "【hellokitty】验证码为：123"
        }
        # 改为failed
        td.tsms_update("send", "status", "failed", uuid=self.uuid)
        res = td.tsms_select("send", "status", uuid=self.uuid)
        assert res.get("status") == "failed"
        # 写mq
        mq.push_direct_secure(json.dumps(data))
        # 再查数据库
        self.db_retry(self.uuid)
```

生产服务测试

针对生产者的测试时，重点是关注生产者生产的数据是否符合预期。即：写入mq里的数据是否符合定义。由于写入mq的数据，必须使用消费程序才能消费出来，而消费程序又不能直接和自动化用例进行数据交互(除非消费程序本身提供接口)，所以我们需要一个存储介质，将我们的实际结果保存起来。本次实战，将使用redis存储。注意：测试生产服务的时候，一定要将消费服务关闭

```
import logging, time, json
from tsms_pytest_commons.tsms_rds import TsmsRedis
from tenacity import retry, stop_after_attempt, wait_random_exponential

logging.basicConfig(level=logging.INFO, format='%(asctime)-16s %(levelname)-8s
%(message)s')

rds = TsmsRedis()


class TestTsmsMqConsume(object):

    @retry(stop=stop_after_attempt(5), wait=wait_random_exponential(2, max=5))
    def rds_retry(self, uuid, exp):
        res = rds.get_mq(uuid)
        assert json.loads(res) == exp

    def test_consume_01(self, tb, rds):
        """验证mq消费到的的数据是否符合预期"""
        # 调接口
        data = tb.send_data()
        tb.req_post("message", data)
        # 检查redis数据(消费到的真实数据)
        exp = {
            "uid": tb.json.get("uuid"),
            "phone": data.get("mobiles")[0],
            "content": "【hellokitty】验证码为：123"
        }
        self.rds_retry(tb.json.get("uuid"), exp)
```