Please visit website: http://cxyroad.com

开发者视角：如何保证RabbitMQ的高可用和可靠性

高可用

高可用HA（High Availability）通常是系统架构设计及程序开发过程中必须考虑的因素之一，它通常是指通过设计减少系统不能提供服务的时间，例如服务重启、服务升级、网络原因等等。通常会使用以下可用性级别去衡量系统，那么大多数公司的高可用目标是达到4个9。

可用性级别	系统可用性（%）	宕机时间/年	计算方法
极高可用	99.999%	5.26分	(1-99.999%)* 365* 24* 60 =5.26分钟
高可用	99.99%	52.6分	(1-99.99%)* 365* 24 =52.6分钟
较高可用	99.9%	8.76时	(1-99.9%)* 365* 24 =8.76时
基本可用	99%	3.65天	(1-99%)* 365 =3.65天
不可用	90%	36.5天	(1-90%)* 365 =36.5天

可用性级别只是我们对系统的一个简单参考，举个例子假设电商系统在一年中的绝大多数时间都可以保证可用性，当我在节假日活动期间有几分钟的服务故障，即使它也达到了高可用的级别但从客观角度来说对系统的影响无疑是巨大的。反之亦然，我可以在大流量或白天的极大多数时间保证可用性，在夜晚或凌晨宕机时间累积比较长，这种情况用户或许没有那么强烈的感知。

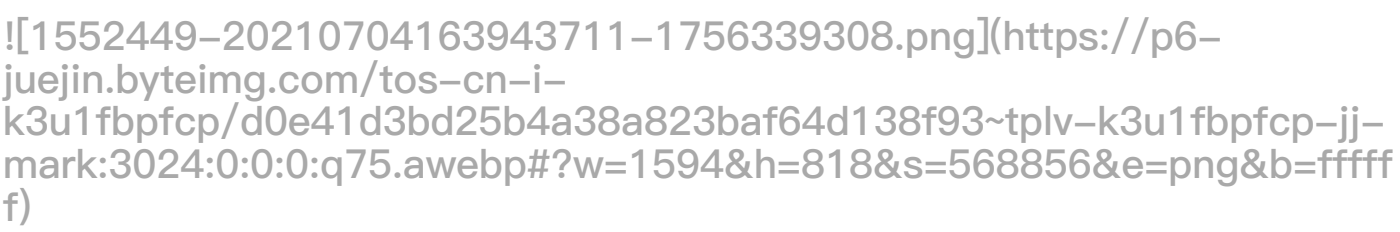
高可用常见方案

1. **集群部署：** 可以参考该文章对集群的介绍
 # 简述RabbitMQ常用集群模式
2. **负载均衡：** 使用负载均衡器如HAProxy可以在多个RabbitMQ节点间分发客户端的连接请求，从而提高吞吐量并提供故障转移能力。
3. **持久化：** 配置消息持久化可以确保在RabbitMQ节点宕机时不会丢失消息。持久化可以将队列中的消息存储到磁盘上，这样即使节点发生故障，重启后仍然可以从磁盘恢复这些消息。
4. **监控和告警：** 实施监控和告警机制，以便在RabbitMQ节点出现问题时及时发现并采取措施。可以使用RabbitMQ自带的管理界面、插件或第三方监控工具来监控集群状态、队列长度、节点健康等。

5. **定期维护：** 定期对RabbitMQ集群进行维护，包括软件更新、性能调优、硬件检查等，以确保系统的稳定性和可靠性。
6. **故障转移和恢复策略：** 制定详细的故障转移和恢复策略，确保在发生故障时能够迅速切换到备用节点，并在故障节点恢复后重新加入集群。
7. **网络分区：** 在网络分区（网络断开导致集群节点间无法通信）的情况下，RabbitMQ集群可以配置为继续运行在可用的节点上，而不是整个集群宕机。
8. **配置文件和.erlang.cookie：** 确保所有节点的配置文件和`.erlang.cookie`文件一致，这是Erlang节点之间进行通信认证的关键。

可靠性
===

在中高级开发的面试中基本上也是必考的问题，如果想要回答好这个问题还是要从RabbitMQ的架构谈起，下图就是一个简单的MQ架构。



其实我们使用MQ也是只有三个步骤，所以我们只需要保证在以下三个步骤中消息不出现丢失既可以保证可靠性：

1. 生产者（Producer）生产消息创建连接（Connection）后通过信道（Channel）发送到RabbitMQ的服务节点Broker。
2. 消息首先传递到交换机（Exchange），根据指定规则分发到不同队列（Queue）
3. 消费者（Consumer）监听消息队列，并消费消息

rabbitMQ配置：

```
@Configuration
public class RabbitConfig {
    @Bean
    public Queue orderQueue() {
        return new Queue(MQConstants.ORDER_DIRECT_QUEUE);
    }
```

```
    @Bean
    TopicExchange exchange() {
        return new
TopicExchange(MQConstants.ORDER_DIRECT_EXCHANGE);
    }

    @Bean
    Binding bindingExchangeMessage() {
        return
BindingBuilder.bind(orderQueue()).to(exchange()).with(MQConstants.ORD
ER_DIRECT_ROUTING);
    }
}
```

生产者具体实现：

```
@Service
public class MessageServiceImpl implements MessageService {
    @Autowired
    private RabbitTemplate rabbitTemplate;

    @Override
    public boolean sendMessage() {
        MessageProperties messageProperties = new MessageProperties();
messageProperties.setMessageId(String.valueOf(UUID.randomUUID()));
     messageProperties.setContentType("text/plain");
        messageProperties.setContentEncoding("utf-8");
        Message message = new Message("hello,message
idempotent!".getBytes(), messageProperties);

rabbitTemplate.convertAndSend(MQConstants.ORDER_DIRECT_EXCHAN
GE, MQConstants.ORDER_DIRECT_ROUTING, message);
        return true;
    }
}
```

消费者具体实现：

```
```

```
@Slf4j
@Component
public class MessageConsumer {
    @RabbitListener(queues = MQConstants.ORDER_DIRECT_QUEUE)
    public void process(Channel channel, Message message) {
        String messageID =
message.getMessageProperties().getMessageId();
        log.info("messageID:::{}", messageID);

        String body = new String(message.getBody());
        log.info("body:::{}", body);

    }
}
```

生产者保证消息可靠性

生产者保证消息可靠性有两种方案，**只能二选一**：

> 消息确认事务机制（不推荐）

1. 配置类中配置事务管理器

```
@Configuration
public class RabbitConfig {
    /**
     * 配置事务管理器
     */
    @Bean
    public RabbitTransactionManager
transactionManager(ConnectionFactory connectionFactory) {
        return new RabbitTransactionManager(connectionFactory);
    }

    @Bean
    public Queue orderQueue() {
        return new Queue(MQConstants.ORDER_DIRECT_QUEUE);
    }
```

```
    @Bean
    TopicExchange exchange() {
        return new
TopicExchange(MQConstants.ORDER_DIRECT_EXCHANGE);
    }

    @Bean
    Binding bindingExchangeMessage() {
        return
BindingBuilder.bind(orderQueue()).to(exchange()).with(MQConstants.ORD
ER_DIRECT_ROUTING);
    }
}
```

2. 开启事务实现事务机制

```
@Service
public class MessageServiceImpl implements MessageService {
    @Autowired
    private RabbitTemplate rabbitTemplate;

    @Override
    @Transactional
    public boolean sendMessage() {
        MessageProperties messageProperties = new MessageProperties();
messageProperties.setMessageId(String.valueOf(UUID.randomUUID()));
     messageProperties.setContentType("text/plain");
        messageProperties.setContentEncoding("utf-8");
        Message message = new Message("hello,message
idempotent!".getBytes(), messageProperties);

        // 开启事务
        rabbitTemplate.setChannelTransacted(true);
rabbitTemplate.convertAndSend(MQConstants.ORDER_DIRECT_EXCHAN
GE, MQConstants.ORDER_DIRECT_ROUTING, message);
        return true;
    }
}
```

**总结：如果MQ未成功收到消息抛出异常。该方式虽然可以保证生产者消息可

靠性但是增加了系统性能开销，由于事务机制需要等待消息发送的结果确认导致了消息的延迟。同时对事物的处理增加了系统复杂性**

> 消息确认Confirm机制（推荐）

消息确认机制有两个关键点：一个为保证生产者发送消息到 RabbitMQ Server，另一个为保证消息能从交换机路由到指定队列

1. 配置文件设置

```
rabbitmq:
  host: localhost
  port: 5672
  username: admin
  password: admin
  virtual-host: /
  publisher-confirm-type: correlated  # 开启发送方确认机制
  publisher-returns: true    # 开启消息返回
  template:
    mandatory: true      # 消息投递失败返回客户端
```

`publisher-confirm-type`选项：

* **none**：禁用发布确认模式。这是默认值，表示消息发布后不会触发任何确认回调。
* **correlated**：当消息成功到达Broker（消息队列服务器）后，会触发`ConfirmCallBack`回调。这种模式通常用于确保消息已成功发送到Broker。
* **simple**：在simple模式下，如果消息成功到达Broker，同样会触发`ConfirmCallBack`回调。但与`correlated`模式不同的是，发布消息成功后，可以使用`rabbitTemplate`调用`waitForConfirms`或`waitForConfirmsOrDie`方法等待Broker节点返回发送结果。根据返回结果，可以判定下一步的逻辑。需要注意的是，如果`waitForConfirmsOrDie`方法返回false，则会关闭channel信道，导致接下来无法发送消息到Broker。

`mandatory`：分为 **true 失败后返回客户端** 和 **false 失败后自动删除**两种策略。false策略无法保证消息可靠性。

2. 设置消息发送回调方法，及设置路由失败后的回调方法

```
@Slf4j
@Service
public class MessageServiceImpl implements MessageService {
    @Autowired
    private RabbitTemplate rabbitTemplate;

    @Override
    @Transactional
    public boolean sendMessage() {
        MessageProperties messageProperties = new MessageProperties();
messageProperties.setMessageId(String.valueOf(UUID.randomUUID()));
     messageProperties.setContentType("text/plain");
        messageProperties.setContentEncoding("utf-8");
        Message message = new Message("hello,message
idempotent!".getBytes(), messageProperties);

rabbitTemplate.convertAndSend(MQConstants.ORDER_DIRECT_EXCHAN
GE, MQConstants.ORDER_DIRECT_ROUTING, message);

        // 设置消息确认回调方法
        rabbitTemplate.setConfirmCallback(new
RabbitTemplate.ConfirmCallback() {
            /**
             * 消息确认回掉方法
             * @param correlationData 回掉的相关数据
             * @param ack true 为 ack, false 为 nack。是否成功收到消息
             * @param cause nack可选参数，原因
             **/
            @Override
            public void confirm(CorrelationData correlationData, boolean
ack, String cause) {
                log.info("消息ID：{},是否成功：{}, 失败原因：{}",
JSONObject.toJSONString(correlationData), ack, cause);
            }
        });

        // 设置路由失败回调方法
        rabbitTemplate.setReturnsCallback(new
RabbitTemplate.ReturnsCallback() {
            /**
             * @param returned MQ没有将消息投递给指定的队列回调方法
             * @return void
```

```
             * @author taoxiangqian
             * @since 2024/04/02 11:07:40
             **/
            @Override
            public void returnedMessage(ReturnedMessage returned) {
                log.info("投递失败的消息详细信息：{},回复的状态码：{}, 回复的
文本内容：{},消息发给哪个交换机:{},消息用哪个路邮键:{}",
returned.getMessage(), returned.getReplyCode(), returned.getReplyText(),
returned.getExchange(), returned.getRoutingKey());
            }
        });
        return true;
    }
}
```

总结：消息确认机制可以确保消息的可靠性，通过该机制还可以确定消费者的处理进度，同时支持灵活的配置提供了多种确认模式（如简单模式和批量模式）

MQ自身保证消息可靠性

MQ自身保证消息可靠性同样有两种，而且可以同时存在：

> 集群部署

关于集群部署前文已经详细介绍了在此就不再过多赘述了。

> RabbitMQ持久化

RabbitMQ持久化包含交换机持久化、队列持久化和消息持久化。

交换机持久化：

```
```

```
@Bean
public TopicExchange exchange() {
    /**
     * 交换机持久化
     * @param name 交换机名称
     * @param durable 是否持久化
     * @param autoDelete 自动删除
     **/
    return new
TopicExchange(MQConstants.ORDER_DIRECT_EXCHANGE, true, false);
}
```

队列持久化：

```
@Bean
public Queue orderQueue() {
    /**
     * 队列持久化
     * @param name 队列名称
     * @param durable 是否持久化
     **/
    return new Queue(MQConstants.ORDER_DIRECT_QUEUE, true);
}
```

消息持久化：

由于我使用的ampq版本是2.4.7 `MessageProperties`对象内的`deliveryMode`属性默认值就是代表持久化所以不用设置，可根据具体不同版本调整。

```
@Slf4j
@Service
public class MessageServiceImpl implements MessageService {
    @Autowired
    private RabbitTemplate rabbitTemplate;

    @Override
```

```
    @Transactional
    public boolean sendMessage() {
        MessageProperties messageProperties = new MessageProperties();
messageProperties.setMessageId(String.valueOf(UUID.randomUUID()));
     messageProperties.setContentType("text/plain");
        messageProperties.setContentEncoding("utf-8");
        Message message = new Message("hello,message
idempotent!".getBytes(), messageProperties);

rabbitTemplate.convertAndSend(MQConstants.ORDER_DIRECT_EXCHAN
GE, MQConstants.ORDER_DIRECT_ROUTING, message);

        // 设置消息确认回调方法
        rabbitTemplate.setConfirmCallback(new
RabbitTemplate.ConfirmCallback() {
            /**
             * 消息确认回掉方法
             * @param correlationData 回掉的相关数据
             * @param ack true 为 ack, false 为 nack。是否成功收到消息
             * @param cause nack可选参数，原因
             **/
            @Override
            public void confirm(CorrelationData correlationData, boolean
ack, String cause) {
                log.info("消息ID：{},是否成功：{}, 失败原因：{}",
JSONObject.toJSONString(correlationData), ack, cause);
            }
        });

        // 设置路由失败回调方法
        rabbitTemplate.setReturnsCallback(new
RabbitTemplate.ReturnsCallback() {
            /**
             * @param returned MQ没有将消息投递给指定的队列回调方法
             * @return void
             * @author taoxiangqian
             * @since 2024/04/02 11:07:40
             **/
            @Override
            public void returnedMessage(ReturnedMessage returned) {
                log.info("投递失败的消息详细信息：{},回复的状态码：{}, 回复的
文本内容：{},消息发给哪个交换机:{},消息用哪个路邮键:{}",
returned.getMessage(), returned.getReplyCode(), returned.getReplyText(),
returned.getExchange(), returned.getRoutingKey());
            }
        });
        return true;
    }
```

}

```

消费者保证消息可靠性
----------

设计消费者逻辑时，应确保操作是幂等的，这样即使消息被重复消费，也不会产生不良影响。

**消息确认机制**

启用RabbitMQ的自动确认（auto-ack）是默认行为应该关闭自动确认，并在消息被成功处理后手动发送确认（ACK）

1. 增加yml配置

```
spring:
  application:
    name: message
  rabbitmq:
    host: localhost
    port: 5672
    username: admin
    password: admin
    virtual-host: /
    publisher-confirm-type: correlated # 开启发送方确认机制
    publisher-returns: true   # 开启消息返回
    template:
      mandatory: true   # 消息投递失败返回客户端
    listener:
      simple:
        acknowledge-mode: manual # 开启手动确认消费机制
```

2. 消费者增加手动处理

```
    @Slf4j
```
```

```
    @Component
    public class MessageConsumer {
        @RabbitListener(queues = MQConstants.ORDER_DIRECT_QUEUE)
        public void process(Channel channel, Message message) {
            MessageProperties messageProperties =
message.getMessageProperties();
            try {
                // 业务处理
                log.info("messageID:::{}，body:::{}",
messageProperties.getMessageId(), new String(message.getBody()));
                log.info("========:::{}", messageProperties.getDeliveryTag());

                // 业务执行成功则手动确认,deliveryTag:消息
index，multiple：是否批量处理如果为true将ack所有小于deliveryTag的消息
                channel.basicAck(messageProperties.getDeliveryTag(), false);
            } catch (Exception e) {
                // 记录日志
                log.info("出现异常：{}", e.getMessage());
                try {
                    // 手动丢弃信息 deliveryTag:消息index，multiple：是否批量
处理如果为true将ack所有小于deliveryTag的消息，requeue：被拒绝是否重新
入队列（true 添加在队列的末端；false 丢弃）
                    channel.basicNack(messageProperties.getDeliveryTag(),
false, false);
                } catch (IOException ex) {
                    log.info("丢弃消息异常");
                }
            }
        }
    }

```


SpringBoot消息重试机制

SpringBoot提供了消息重试机制，如果消费者抛出异常可以重新发起重试。


```
spring:
  application:
    name: message
  rabbitmq:
    host: localhost
    port: 5672
```

```
    username: admin
    password: admin
    virtual-host: /
    publisher-confirm-type: correlated  # 开启发送方确认机制
    publisher-returns: true   # 开启消息返回
    template:
      mandatory: true      # 消息投递失败返回客户端
    listener:
      simple:
        acknowledge-mode: auto  # 自动确认消费
        retry:
          enabled: true # 开启消费者失败重试
          initial-interval: 10000ms # 初始失败等待时长为10秒
          multiplier: 1  # 失败的等待时长倍数（下次等待时长 = multiplier * 上次等待时间）
          max-attempts: 5 # 最大重试次数
          stateless: true # true无状态；false有状态（如果业务中包含事务，这里设置为false）

```

结论
--

对于消息丢失问题其实无法100%保证，即使现在已有这么多成熟的方案也只能是降低概率而已。

原文链接: https://juejin.cn/post/7353087310109982730