[RabbitMQ与AMQP协议详解](http://www.cnblogs.com/frankyou/p/5283539.html)

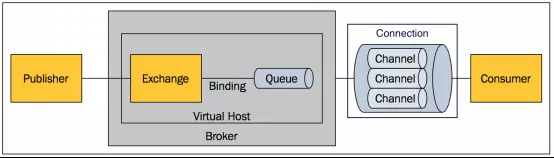
**1. 消息队列的历史**   
了解一件事情的来龙去脉，将不会对它感到神秘。让我们来看看消息队列（Message Queue）这项技术的发展历史。

Message Queue的需求由来已久，80年代最早在金融交易中，高盛等公司采用Teknekron公司的产品，当时的Message queuing软件叫做：the information bus（TIB）。 TIB被电信和通讯公司采用，路透社收购了Teknekron公司。之后，IBM开发了MQSeries，微软开发了Microsoft Message Queue（MSMQ）。这些商业MQ供应商的问题是厂商锁定，价格高昂。2001年，Java Message queuing试图解决锁定和交互性的问题，但对应用来说反而更加麻烦了。

于是2004年，摩根大通和iMatrix开始着手Advanced Message Queuing Protocol （AMQP）开放标准的开发。2006年，AMQP规范发布。2007年，Rabbit技术公司基于AMQP标准开发的RabbitMQ 1.0 发布。

目前RabbitMQ的最新版本为3.5.7，基于AMQP 0-9-1。

RabbitMQ采用Erlang语言开发。Erlang语言由Ericson设计，专门为开发concurrent和distribution系统的一种语言，在电信领域使用广泛。OTP（Open Telecom Platform）作为Erlang语言的一部分，包含了很多基于Erlang开发的中间件／库／工具，如mnesia／SASL，极大方便了Erlang应用的开发。OTP就类似于Python语言中众多的module，用户借助这些module可以很方便的开发应用。

2. AMQP messaging 中的基本概念   


Broker: 接收和分发消息的应用，RabbitMQ Server就是Message Broker。(broker 中间人的意思，类似与服务器的功能)

Virtual host: 出于多租户和安全因素设计的，把AMQP的基本组件划分到一个虚拟的分组中，类似于网络中的namespace概念。当多个不同的用户使用同一个RabbitMQ server提供的服务时，可以划分出多个vhost，每个用户在自己的vhost创建exchange／queue等。(类似与虚拟服务器的概念，每一个用户有一台功能独立的虚拟服务器)

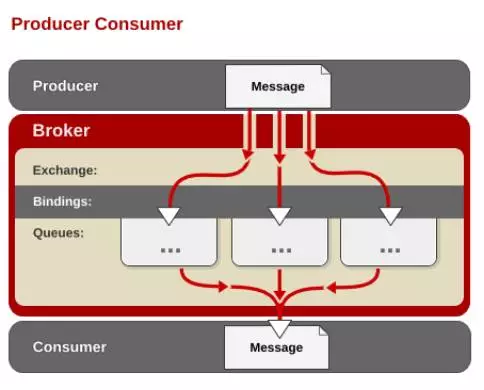
Connection: publisher／consumer和broker之间的TCP连接。断开连接的操作只会在client端进行，Broker不会断开连接，除非出现网络故障或broker服务出现问题。(生产者和消费者对于broker来说，都是客户端，broker不会主动把连接断开，一般都是客户端)

Channel: 如果每一次访问RabbitMQ都建立一个Connection，在消息量大的时候建立TCP Connection的开销将是巨大的，效率也较低。Channel是在connection内部建立的逻辑连接，如果应用程序支持多线程，通常每个thread创建单独的channel进行通讯，AMQP method包含了channel id帮助客户端和message broker识别channel，所以channel之间是完全隔离的。Channel作为轻量级的Connection极大减少了操作系统建立TCP connection的开销。(channel是connection的内部逻辑，为多个连接而创建)

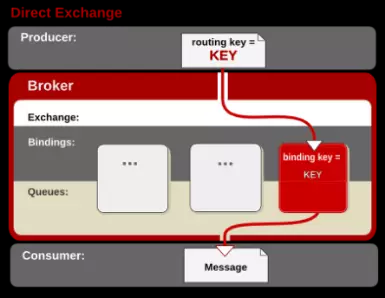
Exchange: message到达broker的第一站，根据分发规则，匹配查询表中的routing key，分发消息到queue中去。常用的类型有：direct (point-to-point)点对点, topic (publish-subscribe) 订阅发布and fanout (multicast)组播。

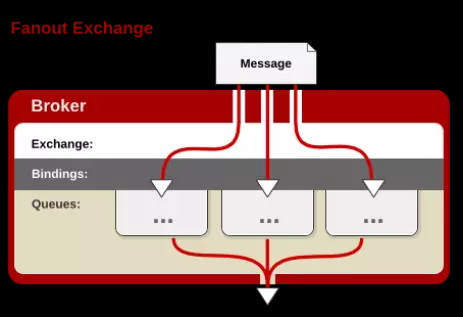
Queue: 消息最终被送到这里等待consumer取走。一个message可以被同时拷贝到多个queue中。

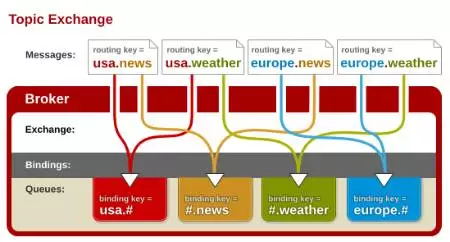
Binding: exchange和queue之间的虚拟连接，binding中可以包含routing key。Binding信息被保存到exchange中的查询表中，用于message的分发依据。

3. 典型的“生产／消费”消息模型   
  
生产者发送消息到broker server（RabbitMQ）。在Broker内部，用户创建Exchange／Queue，通过Binding规则将两者联系在一起。Exchange分发消息，根据类型／binding的不同分发策略有区别。消息最后来到Queue中，等待消费者取走。

4. Exchange类型   
Exchange有多种类型，最常用的是Direct／Fanout／Topic三种类型。

* Direct   
    
  Message中的“routing key”如果和Binding中的“binding key”一致， Direct exchange则将message发到对应的queue中。

Fanout   
  
每个发到Fanout类型Exchange的message都会分到所有绑定的queue上去。

Topic （这种模式是把queue分成了不同的组，每一个组处理不同的信息，同时呢，一个queue可以属于多个组，组里面的queue都得到一个份相同的信息）  
  
根据routing key，及通配规则，Topic exchange将分发到目标queue中。

Routing key中可以包含两种通配符，类似于正则表达式：

“#”通配任何零个或多个word

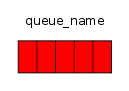
“\*”通配任何单个word

========================

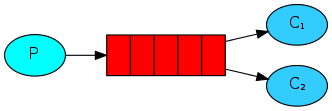
ConnectionFactory、Connection、Channel

ConnectionFactory、Connection、Channel都是RabbitMQ对外提供的API中最基本的对象。Connection是RabbitMQ的socket链接，它封装了socket协议相关部分逻辑。ConnectionFactory为Connection的制造工厂。  
Channel是我们与RabbitMQ打交道的最重要的一个接口，我们大部分的业务操作是在Channel这个接口中完成的，包括定义Queue、定义Exchange、绑定Queue与Exchange、发布消息等。

Queue

Queue（队列）是RabbitMQ的内部对象，用于存储消息，用下图表示。  
[](http://ostest.qiniudn.com/wordpress/wp-content/uploads/2014/02/queue.png)

RabbitMQ中的消息都只能存储在Queue中，生产者（下图中的P）生产消息并最终投递到Queue中，消费者（下图中的C）可以从Queue中获取消息并消费。  
[qq](http://ostest.qiniudn.com/wordpress/wp-content/uploads/2014/02/qq.png)

多个消费者可以订阅同一个Queue，这时Queue中的消息会被平均分摊给多个消费者进行处理，而不是每个消费者都收到所有的消息并处理。（直观的感觉是消息队列queue，只是存储消息用的，而消息的具体内容它是不知道的）  
[](http://ostest.qiniudn.com/wordpress/wp-content/uploads/2014/02/2014-2-21-9-46-43.png)

Message acknowledgment

默认情况下，queue中的消息发送给消费者以后，那么该条消息就从queue中移除了。同时在在rabbitmq重启的情况下，queue中的消息也会丢失

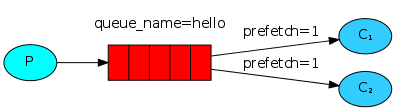
在实际应用中，可能会发生消费者收到Queue中的消息，但没有处理完成就宕机（或出现其他意外）的情况，这种情况下就可能会导致消息丢失。为了避免这种情况发生，我们可以要求消费者在消费完消息后发送一个回执给RabbitMQ，RabbitMQ收到消息回执（Message acknowledgment）后才将该消息从Queue中移除；如果RabbitMQ没有收到回执并检测到消费者的RabbitMQ连接断开，则RabbitMQ会将该消息发送给其他消费者（如果存在多个消费者）进行处理。这里不存在timeout概念，一个消费者处理消息时间再长也不会导致该消息被发送给其他消费者，（隐含的意思是说只要消费的rabbitmq连接没有断开，那么它就一直处于消费状态，不算timeout）除非它的RabbitMQ连接断开。  
这里会产生另外一个问题，如果我们的开发人员在处理完业务逻辑后，忘记发送回执给RabbitMQ，这将会导致严重的bug——Queue中堆积的消息会越来越多；消费者重启后会重复消费这些消息并重复执行业务逻辑…

另外pub message是没有ack的

Message durability（持久化）

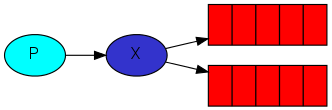
如果我们希望即使在RabbitMQ服务重启的情况下，也不会丢失消息，我们可以将Queue与Message都设置为可持久化的（durable），这样可以保证绝大部分情况下我们的RabbitMQ消息不会丢失。但依然解决不了小概率丢失事件的发生（比如RabbitMQ服务器已经接收到生产者的消息，但还没来得及持久化该消息时RabbitMQ服务器就断电了），如果我们需要对这种小概率事件也要管理起来，那么我们要用到事务。由于这里仅为RabbitMQ的简单介绍，所以这里将不讲解RabbitMQ相关的事务。

Prefetch count预取数目

前面我们讲到如果有多个消费者同时订阅同一个Queue中的消息，Queue中的消息会被平摊给多个消费者。这时如果每个消息的处理时间不同，就有可能会导致某些消费者一直在忙，而另外一些消费者很快就处理完手头工作并一直空闲的情况。我们可以通过设置prefetchCount来限制Queue每次发送给每个消费者的消息数，比如我们设置prefetchCount=1，则Queue每次给每个消费者发送一条消息；消费者处理完这条消息后Queue会再给该消费者发送一条消息。  
[](http://ostest.qiniudn.com/wordpress/wp-content/uploads/2014/02/2014-2-21-9-49-08.png)

（隐含的意思是，默认情况下，queue在平摊消息时，是一块发送多条，给消费者进行处理，而且消费者对消息的处理时间不同，导致有些时间的不一致，有早有晚）

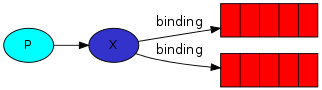
Exchange

在上一节我们看到生产者将消息投递到Queue中，实际上这在RabbitMQ中这种事情永远都不会发生。实际的情况是，生产者将消息发送到Exchange（交换器，下图中的X），由Exchange将消息路由到一个或多个Queue中（或者丢弃）。  
[](http://ostest.qiniudn.com/wordpress/wp-content/uploads/2014/02/2014-2-21-9-51-03.png)  
Exchange是按照什么逻辑将消息路由到Queue的？这个将在Binding一节介绍。  
RabbitMQ中的Exchange有四种类型，不同的类型有着不同的路由策略，这将在Exchange Types一节介绍

routing key

生产者在将消息发送给Exchange的时候，一般会指定一个routing key，来指定这个消息的路由规则，（就是说生产者在发送消息的时候，该消息会携带一个routing key的关键信息，routing key路由关键字，告诉如何消息如何路由）而这个routing key需要与Exchange Type及binding key联合使用才能最终生效。  
在Exchange Type与binding key固定的情况下（在正常使用时一般这些内容都是固定配置好的），我们的生产者就可以在发送消息给Exchange时，通过指定routing key来决定消息流向哪里。  
RabbitMQ为routing key设定的长度限制为255 bytes。

Binding

RabbitMQ中通过Binding将Exchange与Queue关联起来，这样RabbitMQ就知道如何正确地将消息路由到指定的Queue了。  
[](http://ostest.qiniudn.com/wordpress/wp-content/uploads/2014/02/2014-2-21-9-52-46.png)

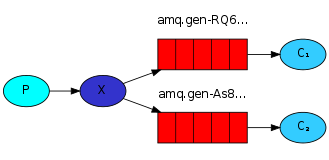
Binding key

在绑定（Binding）Exchange与Queue的同时，一般会指定一个binding key；消费者将消息发送给Exchange时，一般会指定一个routing key；当binding key与routing key相匹配时，消息将会被路由到对应的Queue中。这个将在Exchange Types章节会列举实际的例子加以说明。  
在绑定多个Queue到同一个Exchange的时候，这些Binding允许使用相同的binding key。  
binding key 并不是在所有情况下都生效，它依赖于Exchange Type，比如fanout类型的Exchange就会无视binding key，而是将消息路由到所有绑定到该Exchange的Queue。

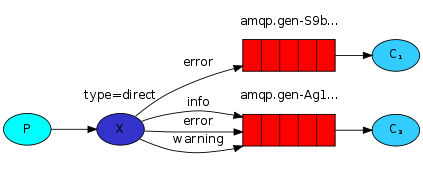
Exchange Types

RabbitMQ常用的Exchange Type有fanout、direct、topic、headers这四种（AMQP规范里还提到两种Exchange Type，分别为system与自定义，这里不予以描述），下面分别进行介绍。

fanout

fanout类型的Exchange路由规则非常简单，它会把所有发送到该Exchange的消息路由到所有与它绑定的Queue中。  
[](http://ostest.qiniudn.com/wordpress/wp-content/uploads/2014/02/2014-2-21-9-54-26.png)  
上图中，生产者（P）发送到Exchange（X）的所有消息都会路由到图中的两个Queue，并最终被两个消费者（C1与C2）消费

direct

direct类型的Exchange路由规则也很简单，它会把消息路由到那些binding key与routing key完全匹配的Queue中。  
[](http://ostest.qiniudn.com/wordpress/wp-content/uploads/2014/02/2014-2-21-9-55-20.png)  
以上图的配置为例，我们以routingKey=”error”发送消息到Exchange，则消息会路由到Queue1（amqp.gen-S9b…，这是由RabbitMQ自动生成的Queue名称）和Queue2（amqp.gen-Agl…）；如果我们以routingKey=”info”或routingKey=”warning”来发送消息，则消息只会路由到Queue2。如果我们以其他routingKey发送消息，则消息不会路由到这两个Queue中。

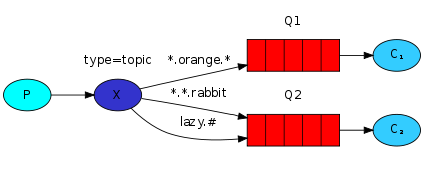
topic

前面讲到direct类型的Exchange路由规则是完全匹配binding key与routing key，但这种严格的匹配方式在很多情况下不能满足实际业务需求。topic类型的Exchange在匹配规则上进行了扩展，它与direct类型的Exchage相似，也是将消息路由到binding key与routing key相匹配的Queue中，但这里的匹配规则有些不同，它约定：

routing key为一个句点号“. ”分隔的字符串（我们将被句点号“. ”分隔开的每一段独立的字符串称为一个单词），如“stock.usd.nyse”、“nyse.vmw”、“quick.orange.rabbit”

binding key与routing key一样也是句点号“. ”分隔的字符串

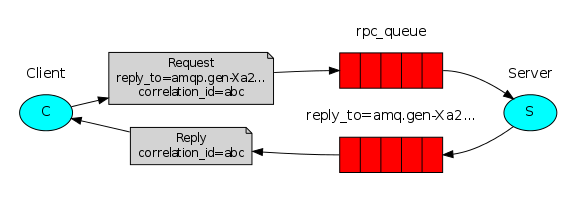
binding key中可以存在两种特殊字符“\*”与“#”，用于做模糊匹配，其中“\*”用于匹配一个单词，“#”用于匹配多个单词（可以是零个）

[](http://ostest.qiniudn.com/wordpress/wp-content/uploads/2014/02/2014-2-21-9-57-37.png)  
以上图中的配置为例，routingKey=”quick.orange.rabbit”的消息会同时路由到Q1与Q2，routingKey=”lazy.orange.fox”的消息会路由到Q1与Q2，routingKey=”lazy.brown.fox”的消息会路由到Q2，routingKey=”lazy.pink.rabbit”的消息会路由到Q2（只会投递给Q2一次，虽然这个routingKey与Q2的两个bindingKey都匹配）；routingKey=”quick.brown.fox”、routingKey=”orange”、routingKey=”quick.orange.male.rabbit”的消息将会被丢弃，因为它们没有匹配任何bindingKey。

**headers**

headers类型的Exchange不依赖于routing key与binding key的匹配规则来路由消息，而是根据发送的消息内容中的headers属性进行匹配。  
在绑定Queue与Exchange时指定一组键值对；当消息发送到Exchange时，RabbitMQ会取到该消息的headers（也是一个键值对的形式），对比其中的键值对是否完全匹配Queue与Exchange绑定时指定的键值对；如果完全匹配则消息会路由到该Queue，否则不会路由到该Queue。  
该类型的Exchange没有用到过（不过也应该很有用武之地），所以不做介绍。

**RPC**

MQ本身是基于异步的消息处理，前面的示例中所有的生产者（P）将消息发送到RabbitMQ后不会知道消费者（C）处理成功或者失败（甚至连有没有消费者来处理这条消息都不知道）。  
但实际的应用场景中，我们很可能需要一些同步处理，需要同步等待服务端将我的消息处理完成后再进行下一步处理。这相当于RPC（Remote Procedure Call，远程过程调用）。在RabbitMQ中也支持RPC。  
[](http://ostest.qiniudn.com/wordpress/wp-content/uploads/2014/02/2014-2-21-9-59-04.png)  
RabbitMQ中实现RPC的机制是：

* 客户端发送请求（消息）时，在消息的属性（MessageProperties，在AMQP协议中定义了14中properties，这些属性会随着消息一起发送）中设置两个值replyTo（一个Queue名称，用于告诉服务器处理完成后将通知我的消息发送到这个Queue中）和correlationId（此次请求的标识号，服务器处理完成后需要将此属性返还，客户端将根据这个id了解哪条请求被成功执行了或执行失败）
* 服务器端收到消息并处理
* 服务器端处理完消息后，将生成一条应答消息到replyTo指定的Queue，同时带上correlationId属性
* 客户端之前已订阅replyTo指定的Queue，从中收到服务器的应答消息后，根据其中的correlationId属性分析哪条请求被执行了，根据执行结果进行后续业务处理

=================================

Kombu和RabbitMQ的关系是什么？

Kombu对RabbitMQ提供的API进行了封装，使得程序更加面向对象化，比如封装出了Exchange, Queue等这些类，使得对RabbitMQ的操作更加简单，同时，功能更加强悍

====================================

客户端的问题

rabbitmq-c是一个用于C语言的，与AMQP server进行交互的client库，AMQP协议为版本0-9-1。rabbitmq-c与server进行交互前需要首先进行login操作，在操作后，可以根据AMQP协议规范，执行一系列操作

kombu是一个python的消息库。

Kombu的目标是通过为AMQP协议提供一个地道的高层次接口，来使python中的消息编程更为简单。同时也为通用的消息问题提供试验和测试的解决方案。

AMQP是先进的消息队列协议，一个提供消息定位，队列，路由，高靠性和安全性的开放的标准协议，其中RabbitMQ消息服务是一个流行的实现

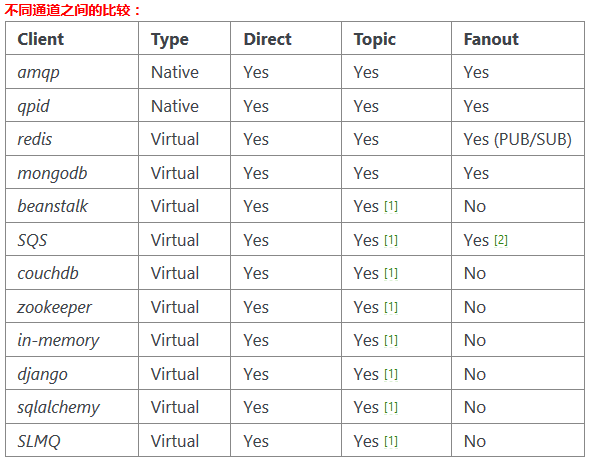
a、AMQP通道使用py-amqp,librabbitmq,或者qpid-python客户端库。

b.如果你安装了librabbitmq,当使用librabbitmq时，C实现的高性能的AMQP通道会自动使能。

c.虚拟的通道使增加对非AMQP通道的支持变得十分容易。现在已经有对 [Redis](http://code.google.com/p/redis/), [Beanstalk](http://kr.github.com/beanstalkd/), [Amazon SQS](http://aws.amazon.com/sqs/), [CouchDB](http://couchdb.apache.org/), [MongoDB](http://www.mongodb.org/), [ZeroMQ](http://zeromq.org/), [ZooKeeper](https://zookeeper.apache.org/), [SoftLayer MQ](http://www.softlayer.com/services/additional/message-queue) 和 [Pyro](http://pythonhosting.org/Pyro)的内置支持。

d.你也可以通过使用SQLAlchemy和Django ORM通道来使用数据库作为中间人。

e.用于单元测试的内存通道。



消息队列的使用过程大概如下：

（1）客户端连接到消息队列服务器，打开一个channel。  
　　（2）客户端声明一个exchange，并设置相关属性。  
　　（3）客户端声明一个queue，并设置相关属性。  
　　（4）客户端使用routing key，在exchange和queue之间建立好绑定关系。  
　　（5）客户端投递消息到exchange。

exchange接收到消息后，就根据消息的key和已经设置的binding，进行消息路由，将消息投递到一个或多个队列里。

exchange也有几个类型，完全根据key进行投递的叫做Direct交换机，例如，绑定时设置了routing key为”abc”，那么客户端提交的消息，只有设置了key为”abc”的才会投递到队列。对key进行模式匹配后进行投递的叫做Topic交换机，符号”#”匹配一个或多个词，符号”\*”匹配正好一个词。例如”abc.#”匹配”abc.def.ghi”，”abc.\*”只匹配”abc.def”。还有一种不需要key的，叫做Fanout交换机，它采取广播模式，一个消息进来时，投递到与该交换机绑定的所有队列。

RabbitMQ支持消息的持久化，也就是数据写在磁盘上，为了数据安全考虑，我想大多数用户都会选择持久化。消息队列持久化包括3个部分：  
　　（1）exchange持久化，在声明时指定durable => 1  
　　（2）queue持久化，在声明时指定durable => 1  
　　（3）消息持久化，在投递时指定delivery\_mode=> 2（1是非持久化）

如果exchange和queue都是持久化的，那么它们之间的binding也是持久化的。如果exchange和queue两者之间有一个持久化，一个非持久化，就不允许建立绑定。

==============

开发接口

Mostly you will be getting a connection from a connection pool, and this connection can be stale, or you could lose the connection in the middle of sending the message. Using retries is a good way to handle these intermittent failures:

大多数情况下，您将从连接池获取连接，并且此连接可能会过时，或者您可能在发送消息时丢失连接。 使用重试是处理这些间歇性故障的好方法：

>>> producer.publish({'hello': 'world', ..., retry=True})

====================

openstack里面的知识点

**OpenStack的rpc源码时，知道rpc是通过消息队列实现的，其中相关的消息队列操作主要是使用的oslo\_messaging库。而这个库进行消息的处理是对kombu开源库的封装。**

**发送和接收信息要依靠Transport和Connections。可供选择的Transport有:amqp, librabbitmq, redis, qpid, in-memory,你甚至还可以自己实现一个。默认的Transport是amqp。**

## AMQP Transports

**对于AMQP来说，有4种transports可用.**

1. **pyamqp 使用纯Python库amqp, 会自动与Kombu一起安装.**
2. **librabbitmq使用C语言编写的高性能transport.这需要安装 librabbitmq Python包, 会自动用c库编译.**
3. **amqp尝试使用librabbitmq 但是因为失败又转而使用pyamqp.**
4. **quid使用纯python库qpid.messaging, 也会自动与Kombu一起安装. Quid库使用AMQP, 但是使用了被Apache Qpid Broker支持的特定扩展.**

**为了达到最高性能，你应该安装librabbitmq包. 为确保使用librabbitmq，你可以在transport URL里显式地指定,或者使用amp作为备选方案.**

