第5章

分布式爬虫系统的设计


视频讲解

尽管有诸如 Scrapy 等强大的爬虫框架，也有如 scrapy-redis 的分布式爬虫框架，但是，它们并不能应用于所有的爬虫业务中。一些业务对于时效性和准确性有着严格的要求，这个时就不得不考虑针对特定业务设计专用的框架。本章首先基于 Redis 介绍常用的分布式消息分发模型：发布-订阅和消息队列，再介绍经典的消息中间件 RabbitMQ 和 Kafka，最后介绍 Python 的分布式框架 Celery。消息系统作为分布式系统的核心，它们解决了分布式系统的消息传递问题，熟练使用消息系统就能快速构建自己的分布式爬虫应用。成熟的消息系统中间件，可以保证消息的完整性，这些中间件在失误率要求较高的业务场景中被广泛应用。

本章以分布式爬虫系统为主题，主要阐述分布式爬虫系统的设计核心——消息系统。从基础的底层消息传递模式，到成熟的中间件 RabbitMQ 和 Kafka 的原理和运用，再到分布式的框架 Celery 的使用。

本章要点如下。

(1) 基于 Redis 如何实现发布-订阅的消息模式。
(2) 基于 Redis 如何实现消息队列的消息模式。
(3) 消息中间件 RabbitMQ 的设计原理。
(4) RabbitMQ 的工作流程和工作模式。
(5) 在 Python 中如何使用 RabbitMQ 中间件。
(6) Kafka 的设计原理和模式。
(7) Kafka 的工作流程。
(8) 在 Python 中如何使用 Kafka。
(9) 分布式框架 Celery 的架构。
(10) 如何使用 Celery 分布式框架。

5.1 消息系统的消息传递模式

消息系统主要处理数据从一个应用程序到另一个应用程序的传递问题，使应用程序可以专注于事务逻辑的处理，而不用过多地考虑如何将消息共享出去，这也是分布式爬虫系统设计的核心。

消息系统有两种消息传递模式：一种是点对点模式即通信队列方式；另一种是基于发布-订阅的消息系统。本章讨论基于 Redis 的这两种方式的实现，分别是 Redis 发布-订阅框

架和分布式通信队列，其中 1.7.1 节中对基于 Redis 的分布式通信队列做了深入的介绍，scrapy-redis 是通过队列实现分布式的经典框架，本章重点介绍相关概念及基于 Redis 订阅-发布功能的分布式框架实现。

基于 Redis 的分布式系统通常是利用公网上的一台 Redis 服务器来实现各个主机之间的通信。通过 Redis 列表数据格式，可以实现先进先出或先进后出的通信队列，scrapy-redis 框架就是基于此功能实现多种通信方式的。

5.1.1　发布-订阅模式

发布-订阅是一种消息范式，消息的发送者（发布者）不会将消息直接发送给特定的客户端（订阅者），同样，订阅者可以接收一个或多个频道的消息，无须了解发布者。发布-订阅是消息队列范式的“兄弟”，通常是更大的面向消息中间件系统的组成部分。大多数消息系统在 API 中同时支持消息队列模型和发布-订阅模型。

发布-订阅模型的优势在于松耦合和高扩展性。松耦合不仅是在位置上解绑，更是在时间上解绑，发布者和订阅者无须顾及彼此的状态，区别于传统的客户端-服务器端模式。在扩展性方面，通过并行操作、消息缓存，发布-订阅在分布式爬虫架构中提供了比传统的客户端-服务器更好的容错性，发布者和订阅者只需要遵循消息发送和接收的规则，不需要彼此适应。

一般很少直接基于 Redis 来实现发布-订阅模式，因为在发布-订阅中，如何保证可靠的消息消费是复杂的工作，往往是直接使用成熟的消息中间件来实现发布-订阅的消息模式，例如 Kafka、RabbitMQ。

5.1.2　点对点模式

点对点模式常用的实现方式是通信队列，通过公共的 Redis 服务可以实现不同主机在同一个 Redis 的队列里面通信。消息队列提供了异步的通信协议，消息的发送者和接收者不需要同时与消息队列交互，消息会保存在队列中，直到接收者取回它。

消息队列本身是异步的，它允许接收者在消息发送很长时间后再取回消息，这和大多数通信协议是不同的，它的最大缺点是接收者必须轮询消息队列，才能接收到最近的消息，增加了接收者的资源开销。

scrapy-redis 是典型的基于 Redis 消息队列的框架，使用了起始任务队列和爬虫任务队列。通过操作 Redis 的不同的数据格式，实现了 FifoQueue（先进先出队列）、LifoQueue（先进后出队列）、PriorityQueue（优先级队列），前两者都是使用 Redis 的列表数据类型，后者使用的是 Redis 的有序数据类型。

一般 Redis 实现的消息队列使用在轻量级、高并发、延迟敏感、错误率要求低的爬虫系统中。要在任务丢失零容忍的场景中使用通信队列，就不得不考虑成熟的中间件，例如 RabbitMQ 就是典型的高可靠消息队列中间件。

5.1.3　Redis 发布-订阅框架

发布-订阅模型将发布的消息分为不同的类别，订阅者可以只接收感兴趣的消息，无须了解发布者的存在。在发布-订阅模型中，订阅者通常接收所有发布的消息的一个子集。选择接收和处理消息的过程被称作过滤，有两种常用的过滤形式：基于主题的过滤和基于内容的过滤。

在基于主题的过滤系统中，消息被发布到主题的命名通道上。订阅者将接收到其订阅的主题上的所有消息，并且所有订阅同一主题的订阅者将接收到同样的消息，发布者负责定义消息类别。在基于内容的过滤系统中，订阅者订阅其感兴趣的消息的条件，只有当消息的属性或内容满足订阅者定义的条件时，消息才会被投递到该订阅者，订阅者需要负责对消息进行分类。一些系统支持两者的混合：发布者发布消息到主题上，而订阅者将基于内容的订阅注册到一个或多个主题上。

发布者与订阅者松耦合，甚至不需要知道两者的存在。由于主题才是关注的焦点，发布者和订阅者可以对系统拓扑结构保持一无所知的状态，各自继续正常操作而无须顾及对方。在传统的紧耦合的客户端-服务器模式中，当服务器进程不运行时，客户端将无法发送消息给服务器，服务器也无法在客户端不运行时接收消息。许多发布-订阅系统不但将发布者和订阅者从位置上解耦，还从时间上解耦他们。

通过并行操作、消息缓存、基于树或基于网络的路由等技术，发布-订阅提供了比传统的客户端-服务器更好的可扩展性。另外，在企业环境之外，发布-订阅模型已经证明了它的可扩展性远超过一个单一的数据中心，通过网络聚合协议，如 RSS 和 Atom 提供互联网范围内分发的消息。

使用 Redis 发布订阅，可以设定对某一个 key 值进行消息发布及消息订阅。当一个 key 值上进行了消息发布后，所有订阅它的客户端都会收到相应的消息。这样的功能也可以用来开发实时的分布式系统，适用于对延迟要求较高的场景中。

Redis 数据库内置发布-订阅模式，相关的操作方法如表 5-1 所示。

表 5-1　Redis 订阅-发布支持的相关操作方法

命　　令	作　　用
PSUBSCRIBE pattern [pattern…]	订阅一个或多个符合给定模式的频道
PUBSUB subcommand [argument [argument…]]	查看订阅与发布系统状态
PUBLISH channel message	将信息发送到指定的频道
PUNSUBSCRIBE [pattern [pattern…]]	退订所有给定模式的频道
SUBSCRIBE channel [channel…]	监听频道发布的消息
UNSUBSCRIBE [channel [channel…]]	停止频道监听

下面将用 Python 设计 Redis 订阅-发布的一个框架，该框架的核心功能有两个，一个是保持系统活性的心跳功能，一个是绑定频道后收到对应消息自动回调的功能，源码如下。

```
import redis
import time
import threading


class RedisMsg:

    def __init__(self, url = 'redis://127.0.0.1:6377/0', timeout = 1):
        pool = redis.ConnectionPool.from_url(url)
        self.beat = None          #心跳线程属性
        self.timeout = timeout
        self._client = redis.Redis(connection_pool = pool)
        self._sub = None
```

```
    def putmsg(self, channel, msg):
        """将信息发送到指定频道"""
        return self._client.publish(channel, msg)

    def creatpub(self):
        """创建订阅者"""
        if self._sub is None:
            self._sub = self._client.pubsub()

    def startsub(self, channel, *channels, **function):
        """
        启动订阅者并监听多个频道,启动心跳线程
        :param timeout: 心跳线程的跳动间隔时间
        :param channel: 监听频道 channel
        :param channels:更多频道 channel1, channel2, channel3
        :param function:给频道指定回调函数 channel = function
        :return:
        """
        task = threading.Thread(target = self._runsub, args = (channel, function, *channels))
        task.start()
        if self.beat is None or not self.beat.is_alive():
            self.beat = threading.Thread(target = self._heartbeat, args = ())
            self.beat.start()

    def addchannel(self, channel, *channels, **function):
        """添加监听频道"""
        if self._sub is None:
            self.startsub(channel, *channels, **function)
        if self.beat.is_alive():
            self.startsub(channel, *channels, **function)
        else:
            self._sub.subscribe(channel, *channels, **function)

    def delchannel(self, channel, *channels):
        """取消监听指定的频道"""
        self._sub.unsubscribe(channel, *channels)

    def _runsub(self, channel, function, *channels):
        """监听订阅信道的子线程"""
        if self._sub is None:
            self.creatpub()
        self._sub.subscribe(channel, *channels, **function)
        for message in self._sub.listen():
            print(message)

    def _heartbeat(self):
        """心跳子线程函数"""
        while True:
            time.sleep(self.timeout)
            channels = self._sub.channels
            if not channels:
                break
            self._sub.ping()


def channel1(msg):
    """处理 channel1 的自动回调函数"""
    print(f"这是频道 1 回调处理程序: {msg}")
```

```
def channel2(msg):
    """处理 channel1 的自动回调函数"""
    print(f"这是频道 2 回调处理程序: {msg}")


if __name__ == '__main__':
    test = RedisMsg()
    test.startsub("channel1", "channel2", channel1 = channel1, channel2 = channel2)
    time.sleep(5)
    test.putmsg("channel1", "channel1 的消息")
    time.sleep(2)
    test.putmsg("channel2", "channel2 的消息")
    time.sleep(5)
    test.addchannel("channel3")
    time.sleep(5)
    test.putmsg("channel3", "channel3 的消息")
    test.delchannel("channel1", 'channel2')
```

输出结果如下。

```
{'type': 'subscribe', 'pattern': None, 'channel': b'channel1', 'data': 1}
…
{'type': 'pong', 'pattern': None, 'channel': None, 'data': b''}
这是频道 1 回调处理程序: {'type': 'message', 'pattern': None, 'channel': b'channel1', 'data': b
'channel1\xe7\x9a\x84\xe6\xb6\x88\xe6\x81\xaf'}
{'type': 'pong', 'pattern': None, 'channel': None, 'data': b''}
{'type': 'pong', 'pattern': None, 'channel': None, 'data': b''}
这是频道 2 回调处理程序: {'type': 'message', 'pattern': None, 'channel': b'channel2', 'data': b
'channel2\xe7\x9a\x84\xe6\xb6\x88\xe6\x81\xaf'}
{'type': 'pong', 'pattern': None, 'channel': None, 'data': b''}
…
{'type': 'pong', 'pattern': None, 'channel': None, 'data': b''}
```

如上框架基于 Redis 数据库的发布-订阅机制，具备发布-订阅模式的核心机制，包括心跳机制、订阅回调机制、发布消息机制、频道操作机制，下面将对该框架具体功能做解释。

为什么要设置心跳机制？设置心跳机制一方面是为了对 Redis 服务器及客户端之间的网络进行检测，如果出现网络问题可以及时重试；另一方面，Redis 在一定空闲时间后会释放连接，当订阅者与发布者之间并不活跃，并且超过 Redis 检测时间时则认为空闲，将断开连接。源码中是通过一个单独的线程定时发送空消息，以保证订阅者和 Redis 服务器按照一定的频率通信，心跳线程将根据监听频道情况选择结束心跳。运行结果中的{'type': 'pong', 'pattern': None, 'channel': None, 'data': b''}数据就是订阅者收到的心跳测试数据，发送频率是一秒一次，这个频率可以通过修改 timeout 属性来控制，默认是 1 秒。

设置心跳线程在很多场景中会经常用到，基于 Redis 的订阅-发布还提供了一些自动完成心跳效果的设置。redis-py 库可以在命令发出之前定期地运行状况检查，以检测连接的活跃性，可以将 health_check_interval = N 传递给 Redis 或 ConnectionPool 类，或作为 RedisURL 中的查询参数。health_check_interval 的值必须是整数，默认值为 0，禁用运行状况检查。任何正整数将启用运行状况检查，如果基础连接空闲时间超过 health_check_interval 秒，则在执行命令之前执行运行状况检查。例如，health_check_interval = 30 将确保在该连接上执行命令之前，对空闲 30 秒或更长时间的所有连接运行状态检查。如果应用程序在 30 秒后断开空闲连接，则应将 health_check_interval 选项设置为小于 30 的值。

启用 health_check_interval 选项也适用于创建的所有 PubSub 连接。PubSub 用户需要确保 listen()的调用比 health_check_interval 时间更频繁。如果 PubSub 实例不经常调用 listen(),则应定期显式调用 pubsub.check_health()以保持连接的活跃性。

这两行信息{'type': 'subscribe', 'pattern': None, 'channel': b'channel1', 'data': 1}{'type': 'subscribe', 'pattern': None, 'channel': b'channel2', 'data': 2},是订阅者监听成功的反馈信息,首次使用 listen()监听成功将打印出来,分别是客户端执行的命令类型 type、匹配模式 pattern、频道 channel、发布者的连接顺序。每当设置一个频道监听都会打印出该条信息。

这是频道 1 回调处理程序输出的信息{'type': 'message', 'pattern': None, 'channel': b'channel1', 'data': b'channel1\xe7\x9a\x84\xe6\xb6\x88\xe6\x81\xaf'},该条信息是发布者发布信息后,订阅者收到了信息,并且订阅者在该频道设置了回调函数,这是 channel1()函数自动回调打印出来的 msg。对于每个频道消息结构都是一样的,是一个 dict 类型的数据,包含消息类型字段 type、频道 channel、消息数据 data、匹配模式 pattern。这些信息默认以回调函数的唯一参数形式传入,只需要实现特定的回调信息和信息的处理函数,然后在监听的时候绑定即可。

使用频道绑定函数,支持同时绑定多个频道,并且支持频道名和回调函数名以键值对形式传入,在监听端口后会自动调用绑定的回调函数。同时,可以使用 sub.channels 获取所有监听信道与对应回调函数的属性字典。上述源码中的心跳函数,在每次心跳时会检测一下监听字典是否为空,如果空就不再执行心跳。

startsub()函数作为主要功能的实现,一方面是在设置监听频道后创建一个子线程单独监听,另一方面是在没有心跳线程的时候创建心跳线程。self._runsub()函数的主要作用是添加监听频道、绑定频道与回调函数、启动监听,分别是通过 self._sub.subscribe(channel, * channels, ** function)、self._sub.listen()实现的。

5.2　基于 RabbitMQ 中间件的设计

RabbitMQ 是实现了高级消息队列协议(Advanced Message Queuing Protocol, AMQP)的开源消息代理软件,也称为面向消息的中间件。RabbitMQ 服务器是用 Erlang 语言编写的,而聚类和故障转移是构建在开放电信平台框架上的。

Rabbit 科技有限公司开发了 RabbitMQ,并提供支持。起初 Rabbit 科技是 LSHIFT 和 CohesiveFT 在 2007 年成立的合资企业,2010 年 4 月被 VMware 旗下的 SpringSource 收购,RabbitMQ 在 2013 年 5 月成为 GoPivotal 的一部分。

有这样的场景:爬虫任务数不多但是每条任务信息的价值极大,如何保证完成每条任务被执行而不出差错?这个时候可以考虑使用 RabbitMQ。RabbitMQ 是常用的消息中间件,有多种消息交换模式,如发布-订阅、路由、通配符等,是典型的生产者-消费者机制,并且具有消息确认机制,在分布式主机相互配合、多任务交叉的场景中,能够保证每条任务的运行,同时 RabbitMQ 也可以用于设计分布式爬虫系统。

高级消息队列协议是一种二进制应用层协议,用于应对广泛的面向消息应用程序的支持。协议提供了消息流控制,保证一个消息对象的传递过程,如至多一次、保证多次、仅有一

次等，和基于 SASL 和 TLS 的身份验证和消息加密。

RabbitMQ 的特点如下。

(1) 异步消息，支持多种消息传递协议、消息排队、传递确认、到队列的灵活路由、多种交换类型。

(2) 跨语言支持，支持主流的开发语言，如 Java、. NET、PHP、Python、JavaScript、Ruby、Go 等。

(3) 分布式部署支持，部署为集群以实现高可用性和吞吐量，跨多个可用区域和区域联合。

(4) 易于部署，可插拔的身份验证、授权，支持 TLS 和 LDAP。

(5) 工具和插件支持，可使用的工具和插件的种类繁多，支持持续集成及与其他企业系统的集成。

(6) 管理与监控，提供了命令行工具和用于管理与监视 RabbitMQ 的 UI 界面。

5.2.1 RabbitMQ 基础

RabbitMQ 是消息代理，它接收生产者发布的消息并发送给对应的消息队列，等待消费者从消息队列里消费消息，它的作用是消息的集散中心。RabbitMQ 业务逻辑由四个重要部分组成：生产者(Producer)、消费者(Consumer)、交换机(Exchange)、队列(Queue)，即 RabbitMQ 的工作流程示意图如图 5-1 所示。

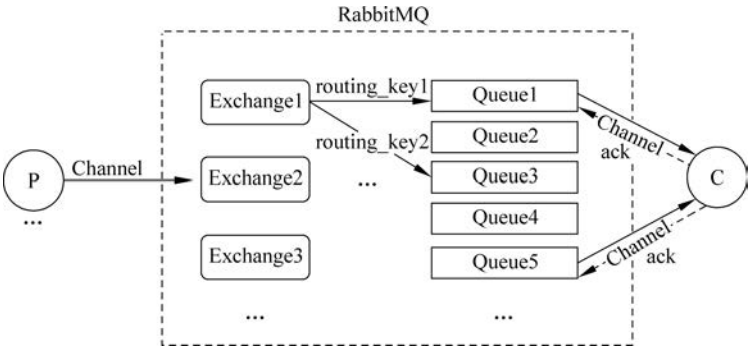

图 5-1 RabbitMQ 工作流程示意图

生产者和消费者通过应用服务器创建的 TCP 连接(Connection)上的虚拟通道(Channel)发送和接收消息。生产者向 RabbitMQ 服务器推送消息时，首先在连接上获取一个通道，通过通道声明一个交换机，然后在向 RabbitMQ 服务器推动消息的时候指定处理消息的交换机和路由键(routing_key)。消费者获取 RabbitMQ 服务器推送的消息时，首先在连接上获取一个通道，通过通道将交换机、路由键、队列绑定(Binding)，交换机与队列的绑定关系可以是多对多，然后通过消费方法指定要消费的队列、消息回调函数、消息是否需要 ack 确认、消息是否持久化等属性。

上述流程中涉及 RabbitMQ 中的几个重要概念，其定义或作用如下。

(1) 生产者(Producer)：发送消息的客户端。

(2) 消费者(Consumer)：接收消息的客户端。

(3) 队列(Queue)：存储消息的队列，可以设置为持久化、临时或者自动删除。

(4) 消息(Message)：由生产者通过 RabbitMQ 发送给消费者的信息。

（5）连接（Connection）：连接 RabbitMQ 和应用服务器的 TCP 连接。

（6）通道（Channel）：连接里的一个虚拟通道，当生产者和消费者发送或接收消息时，这些操作都是通过通道进行的。

（7）交换机（Exchange）：交换机负责从生产者那里接收消息，并根据交换类型和路由键分发到对应的消息列队里。要实现消息的接收，一个队列必须绑定一个交换机。

（8）绑定（Binding）：绑定是队列和交换机的一个关联连接。

（9）路由键（Routing Key）：绑定交换机和队列的关键字，交换机根据工作模式和路由键，决定如何分发消息到列队。

（10）虚拟主机（Virtual Host，Vhosts）：每个 Vhost 都是一个 RabbitMQ 服务，分别管理各自的 Exchange、Binding、Queue，用于对不同用户的权限管理。

交换机有四种类型，分别是 direct（默认）、fanout、topic 和 headers，不同类型的 Exchange 转发消息的策略有所不同。direct 是直接交换机，将消息转发到完全匹配路由键绑定的消息队列上；fanout 是扇出交换机，不管消息的路由键设置是什么，消息都会转发给所有与之绑定的队列；topic 是主题交换机，按照通配符的方式，将消息转发给所有匹配路由键的消息队列；headers 是标头交换机，声明标头交换机，在绑定一个队列的时候，定义一个字典数据结构的消息头；消息发送的时候，会携带一组字典数据结构的信息；当字典的内容匹配上的时候，消息就会被写入队列。绑定交换机和队列的时候，字典结构中要求携带 x-match 键，其值可以是 any 或者 all，代表消息携带的字典是需要全部匹配（all），还是仅需要匹配一个键（any）就可以。相比直连交换机，标头交换机的优势是匹配的规则不被限定为字符串。

5.2.2　Docker 部署 RabbitMQ

1. Windows 下使用 Docker 部署 RabbitMQ

1）创建映射文件、启动 RabbitMQ 容器

在 D 盘下创建 rabbitmq/data 路径的文件，并将该文件设置为 Docker 共享文件夹，然后使用如下命令创建容器，并安装管理器插件 rabbitmq_management，安装后如图 5-2 所示。

```
docker run -- restart = always -- hostname rabbit -- name rabbit - v D:\rabbitmq\data:/var/
lib/rabbitmq/mnesia - e RABBITMQ_DEFAULT_USER = root - e RABBITMQ_DEFAULT_PASS = 123456 - p
15672:15672 - p 5672:5672 - d rabbitmq:latest

# 加载 Web 端管理器插件
docker exec rabbit rabbitmq - plugins enable rabbitmq_management
```

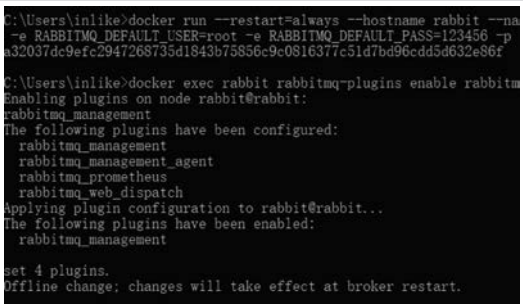

图 5-2　创建 RabbitMQ 容器并安装插件

上述源码命令的解释如下。

```
docker run
--restart=always                                    # 自动重启
--hostname rabbit                                   # 节点名称
--name rabbit                                       # 实例化容器名称
-v D:\rabbitmq\data:/var/lib/rabbitmq/mnesia        # 映射容器内数据文件夹
-e RABBITMQ_DEFAULT_USER=root                       # 设置默认环境变量及 RabbitMQ 用户名
-e RABBITMQ_DEFAULT_PASS=123456                     # 设置默认环境变量,RabbitMQ 的密码
-p 15672:15672                                      # 映射 Web 控制台端口
-p 5672:5672                                        # 映射应用访问端口
-d rabbitmq:latest                                  # 使用最新镜像
```

2）验证服务

打开本地浏览器，输入 http://localhost:15672/#/，正常情况下会出现 Web 管理界面的登录窗口，如图 5-3 所示。然后输入设置的账号 root 和密码 123456，单击 Login 按钮，可看到如图 5-4 所示的登录后界面，则代表 RabbitMQ 服务安装成功，后面将用 RabbitMQ 实现分布式爬虫的设计。

图 5-3　RabbitMQ 管理界面登录窗口

图 5-4　RabbitMQ 登录后界面

2. Linux 下使用 Docker 部署 RabbitMQ

如果直接安装 RabbitMQ 带管理界面的版本，需要后缀带有 mangement 的镜像，可以从 Docker Hub 官方镜像源查询那些带有 mangement 后缀的版本。打开官网镜像源地址 https://hub.docker.com/，在上方搜索框输入 RabbitMQ，单击 Serch 按钮获取结果列表，在 Tags 标签中查找带有 mangement 后缀的版本，如图 5-5 所示。

图 5-5　带有 mangement 后缀的版本

这里找到镜像标签是 3.8.3-beta.2-management-alpine 的版本，使用 docker pull rabbitmq：3.8.3-beta.2-management-alpine 命令拉取镜像。

1）创建映射文件、启动 RabbitMQ 容器

在/home 路径下创建/home/rabbitmq/data 路径文件，编辑容器启动命令。

```
docker run --restart=always --hostname rabbit --name rabbit -v /home/rabbitmq/data:/
var/lib/rabbitmq -e RABBITMQ_DEFAULT_USER=root -e RABBITMQ_DEFAULT_PASS=123456 -p
15672:15672 -p 5672:5672 -d rabbitmq:3.8.3-beta.2-management-alpine
```

上述命令的解释如下。

```
docker run
--restart=always                                    # 自动重启
--hostname rabbit                                   # 节点名称
--name rabbit                                       # 实例化容器名称
-v /home/rabbitmq/data:/var/lib/rabbitmq            # 映射容器内数据文件夹
-e RABBITMQ_DEFAULT_USER=root                       # 设置默认环境变量及 RabbitMQ 的用户名
-e RABBITMQ_DEFAULT_PASS=123456 # 设置默认环境变量及 RabbitMQ 的密码
-p 15672:15672                                      # 映射 Web 控制台端口
-p 5672:5672                                        # 映射应用访问端口
-d rabbitmq:3.8.3-beta.2-management-alpine          # 后台模式启动
```

成功执行命令后将启动 RabbitMQ，如图 5-6 所示。

```
[root@VM_0_12_centos rabbitmq]# docker run --restart=always --hostname rabbit --name rabbit -v /home/
ITMQ_DEFAULT_PASS=123456 -p 15673:15672 -p 5673:5672 -d rabbitmq:management
9cf61693d63ae9c851458c01f94d54c90781666a65bab2f8f0855fce490a5a7e
[root@VM_0_12_centos rabbitmq]# docker ps
CONTAINER ID        IMAGE                  COMMAND                  CREATED             STATUS
                    NAMES
9cf61693d63a        rabbitmq:management    "docker-entrypoint.s…"   6 seconds ago       Up 4 seconds
p, 0.0.0.0:15673->15672/tcp   rabbit
[root@VM_0_12_centos rabbitmq]#
```

图 5-6　执行命令启动 RabbitMQ

2）验证服务

打开本地浏览器，按照 ip:prot 格式输入服务的主机 IP 地址和端口，例如上述案例的地址是 118.24.52.111:15672，正常情况下会出现 Web 管理页面的登录窗口。

5.2.3　RabbitMQ 可视化管理

首先需要正确部署 RabbitMQ 环境，打开 RabbitMQ 的管理界面，如图 5-6 所示。

图 5-7　RabbitMQ 的管理界面

Overview 选项卡中是 RabbitMQ 的总览信息，其下的 Totals 面板是消息统计图表；Connections 选项卡中是生产者和消费者的连接情况，它是 TCP 的连接信息；Channels 选项卡是建立在连接上的通道信息，每个消费者或者生产者工作时都需要一条通道；Exchange 选项卡中是交换机的信息及配置选项，通过该菜单可以对交换机进行手动的管理；Queues 选项卡中是队列情况，包括消息的状态统计；Admin 选项卡中是用户管理信息，主要是用户的增、删、改、查以及虚拟主机和规则等的配置。

主要关注 Exchange 选项卡和 Queues 选项卡，这是两个常用的菜单，用来对交换机和队列进行操作。单击 Exchange 选项卡中的 Add a new exchange 选项可以增加一个交换机，如图 5-8 所示。配置选项有 Name(交换机名字)、Type(交换机类型)、Durability(持久化)、Auto delete(是否自动删除)、Internal(是否内部使用)、Arguments(分发器的其他选项)，一般情况下只需要对 Name、Type、Durability 配置即可。创建一个 name 值是 hello，其他配置项默认的交换机。

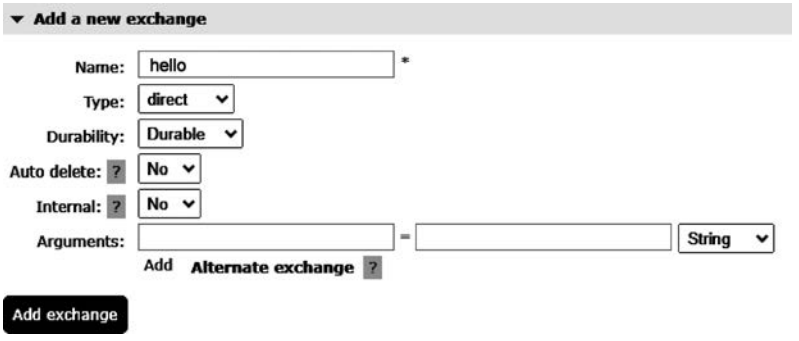

图 5-8　增加一个交换机

单击 Queues 选项卡中的 Add a new queue 选项，可以增加一个队列，如图 5-9 所示。

配置选项有 Name(队列名)、Durabilit(持久化)、Auto delete(自动删除)、Arguments(属性参数)，其中 Name 是必填参数。创建一个队列后，需要从 All queue 选项下的队列列表中进入队列详情页，单击 Bindings 选项设置队列与关键词、交换机的绑定，如图 5-10 所示。

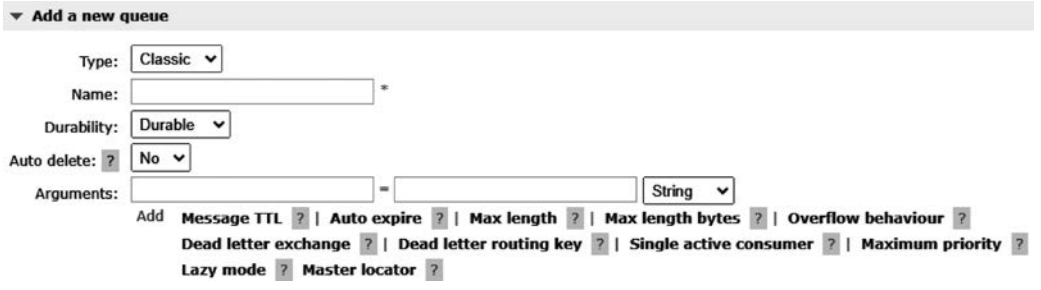

图 5-9　增加一个队列

这里创建一个名为 world 的队列，绑定到交换机 hello，路由键是 test。首先在 Add a new queue 选项卡下创建一个名为 world 的队列，然后进入该队列的详情页，在详情页的 Bindings 选项中，依次在 From exchange 文本框中输入 hello，Routing key 文本框中输入 test，单击 Bind 按钮即可完成绑定。

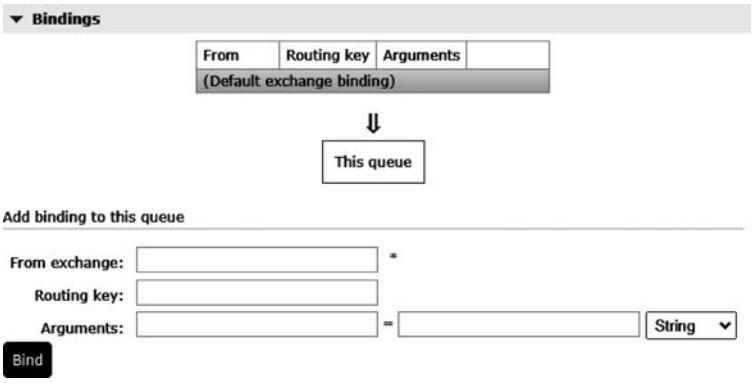

图 5-10　队列与关键词、交换机的绑定

5.2.4　Python 中使用 RabbitMQ

RabbitMQ 提供了多种语言的开发接口，Python 使用的模块是 pika，首先确保运行环境中存在该库。使用 pika 实现 RabbitMQ 的生产者和消费者，其思路是客户端与服务器先创建 TCP 连接，然后从连接中生成通道，消费者和生产者的相关操作都是在通道上完成的。

下面是实现生产者和消费者通信的示例代码，使用的是在前面已经创建的 hello 交换机和 world 队列。

```
import pika


def callback(ch, method, properties, body):
    """
    消费者的回调函数
    :param ch:
    :param method:
    :param properties:
    :param body:
    :return:
    """
    print(" [消费者消息: ] %r" % body)


user_pwd = pika.PlainCredentials('root', '123456')
conn = pika.BlockingConnection(pika.ConnectionParameters(host = 'localhost', port = 5672,
heartbeat = 10,
credentials = user_pwd))
channel = conn.channel()
# 创建交换机
# channel.exchange_declare(exchange = 'hello', durable = True)
channel.basic_publish(exchange = 'hello', routing_key = 'test', body = 'hello world')
# channel.queue_bind(exchange = 'hello', queue = 'world', routing_key = 'test')
# 绑定
channel.basic_consume('world', callback, auto_ack = True)
channel.start_consuming()
```

其中，callback()是用于回调的函数，该函数有固定的形参。pika.PlainCredentials()方法用于创建默认的身份验证方法的凭据对象，如果没有向该方法传递参数，将自动创建一个

都是 guest 的用户名和密码。其中 heartbeat 参数是心跳间隔时间，单位是秒。

使用 pika. ConnectionParameters()包装连接参数，然后通过 pika. BlockingConnection()堵塞式创建连接，该方法能返回可用的连接对象或抛出连接中的异常。conn. channel()的作用是从连接上获取一个通道 channel，后面的消息推送和消费都是在通道上完成的。

注释掉的 channel. exchange_declare(exchange＝'hello', durable＝True)，其作用是创建一个交换机并设置相关属性，这里使用手动创建的 hello 交换机，不需要再另外创建一个交换机。通过 channel. basic_publish()方法向指定的交换机推送消息，并指定该消息的路由键。

注释掉的 channel. queue_bind(exchange＝'hello', queue＝'world', routing_key＝'test')，其作用是绑定交换机、队列、路由键，这里使用的是已经绑定的交换机和队列。通过 channel. basic_consume()方法设置消费的队列和回调函数以及是否确认消息，如果确认消息则回调函数中调用 ch. basic_ack(delivery_tag＝method. delivery_tag)完成确认，不确认的消息会一直保存在队列中直到被其他消费者消费或过期删除。

channel. start_consuming()开始监听队列，程序将堵塞等待来自队列的消息。

下面将基于 RabbitMQ 设计一个用于分布式爬虫通信的框架，主要使用直接交换机。新建一个 rabbitmq. py 文件，写入以下代码。

```
import pika


class RabbitMQBASE:

    def __new__(cls, *args, **kw):
        if not hasattr(cls, '_instance'):
            new = super(RabbitMQBASE, cls)
            cls._instance = new.__new__(cls)
        return cls._instance

    def __init__(self, use, pwd, host, port = 5673, heartbeat = 30):
        user_pwd = pika.PlainCredentials(use, pwd)
        self.s_conn = pika.BlockingConnection(
              pika. ConnectionParameters ( host = host, port = port, heartbeat = heartbeat,
credentials = user_pwd))

    def channel(self):
        """
        获取通道
        :return:
        """
        return self.s_conn.channel()

    def close(self):
        """
        关闭链接
        :return:
        """
        self.s_conn.close()
```

```
class RabbitMQ(RabbitMQBASE):

    def __init__(self, use, pwd, host, port, queue_key, exchange,
                 queue_name, ack = True, exchange_durable = True, queue_durable = True):
        RabbitMQBASE.__init__(self, use, pwd, host, port)
        self.exchange = exchange
        self.queue_name = queue_name
        self.queue_key = queue_key
        self.ack = ack
        self.exchange_durable = exchange_durable
        self.queue_durable = queue_durable

    def rabbit_get(self, callback = None):
        """
        消费者
        :param callback: 传入回调函数
        :return:
        """
        channel = self.channel()
        channel.queue_bind(exchange = self.exchange,      #将交换机、队列、关键字绑定
                           queue = self.queue_name, routing_key = self.queue_key)
        channel.basic_consume(self.queue_name, callback if callback else self.callback, auto_ack =
self.ack)
        channel.start_consuming()                         #堵塞等待消息

    def callback(self, ch, method, properties, body):
        """
        示例消息回调函数
        :param ch:
        :param method:
        :param properties:
        :param body: 消息正文
        :return:
        """
        print(f"[消息: ]{body.decode()}")
        if self.ack is False:
            ch.basic_ack(delivery_tag = method.delivery_tag)

    def rabbit_put(self, msg_key, message = 'hello world'):
        """
        生产者
        :param msg_key: 路由关键字
        :param message: 需要发送的消息
        :return:
        """
        channel = self.channel()
        channel.exchange_declare(exchange = self.exchange, durable = self.exchange_durable)
        channel.basic_publish(exchange = self.exchange, routing_key = msg_key, body =
message)
        channel.close()
```

通过单例模式保证实例化一个 RabbitMQ 对象，避免创建多余的 TCP 链接。在 RabbitMQ 对象中，既实现了消费者方法又实现了生产者方法，但是需要注意消费者运行后，程序将处于堵塞状态。在消费者的回调函数内可以调用生产者方法产生新任务，这样就实现了网络爬虫大致的下载-解析-下载的整体工作路线。

RabbitMQ 对象的初始化参数有 use(连接用户名)、pwd(连接密码)、host(链接地址)、port(连接端口)、queue_key(队列绑定的关键字)、exchange(交换机)、queue_name(队列名)、ack(消息确认,为 False 时需要确认)、exchange_durable(交换机持久化)、queue_durable(队列持久化)。

rabbit_put()是生产者方法,参数 msg_key 是消息的路由键,参数 message 是要发送的消息; rabbit_get()是消费者方法,首先使用 queue_bind()方法绑定消费的队列、交换机和路由键,然后使用 basic_consume()方法设置监听的队列和回调方法,即是否确认消息,最后使用 start_consuming()开始堵塞监听,当收到消息时将自动调用指定的回调方法。

通过 RabbitMQ 的生产者和消费者,可以快速实现分布式爬虫系统的开发。针对不同层级的 URL,可以绑定不同的队列,指定不同的回调函数,通过确认机制实现任务的精准执行。同时借助于 RabbitMQ 自带的任务分发机制,可以很容易地扩展系统,实现任务的高并发。

5.3 基于 Kafka 中间件的设计

Kafka 是一个分布式的发布-订阅消息系统和一个强大的队列,适用于大数据场景中。Kafka 支持离线和在线消息消费,消息在磁盘上持久化保存,并通过集群内复制备份,保障数据安全。Kafka 构建在 Apache ZooKeeper 的同步服务之上,可与 Apache Storm 和 Apache Spark 非常好地集成,便于实时流式数据分析。

ApacheZooKeeper 是 Apache 软件基金会的一个软件项目,它为大型分布式计算提供开源的分布式配置服务、同步服务和命名注册。Storm 是一个分布式计算框架,主要由 Clojure 编程语言编写。Apache Spark 是一个开源集群运算框架,最初是由加州大学伯克利分校 AMPLab 所开发。

Kafka 专为分布式高吞吐量系统而设计。Kafka 与其他传统消息传递系统相比,具有更好的吞吐量、内置分区、复制备份和固有的容错能力等特点,这使它非常适合作为设计大规模消息处理的系统中间件。

通过 Kafka 设计的分布式爬虫系统,可以更好地嵌入大数据分析平台。不管是在任务的分发还是任务结果的存储和传递中,每秒超百万次的读写操作为大型爬虫系统提供了高并发的任务读写服务,为分布式大数据处理系统提供了数据高效分析和处理的基础服务。

5.3.1 Kafka 基础

1. Kafka 集群架构

Kafka 集群服务主要由生产者(Producer)、消费者(Consumer)、代理(Broker)、ZooKeeper 组成,它们之间的关系如图 5-11 所示。Kafka 集群中有很多台 Server,其中每台 Server 都可以存储消息,将每台 Server 称为一个 Kafka 实例,也叫作 Broker。

Zookeeper 是 Broker 和消费者之间的协调接口。Kafka 服务器是无状态的,它们通过 Zookeeper 集群共享信息。Kafka 在 Zookeeper 中存储主题、代理等基本元数据信息。将所有关键信息存储在 Zookeeper 中,并且在其整体上复制此数据,当 Broker 或 Zookeeper 发生故障时,不会影响 Kafka 集群的状态。

在 Broker 中,信息是按照主题分类后保存到相同主题(Topic)下分区(Partition)中的,

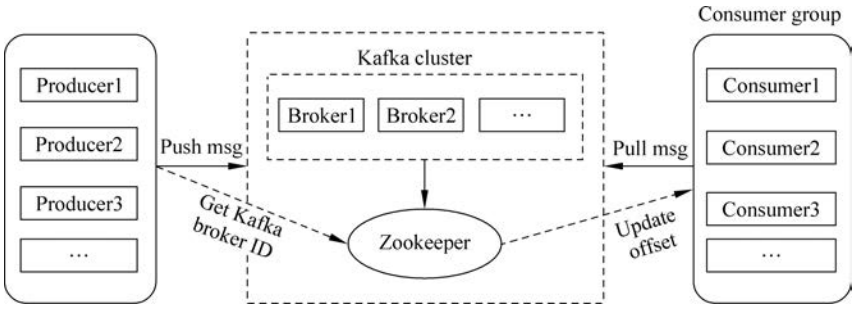

图 5-11　Kafka 集群架构示意图

消费者通过偏移量(Offset)来获取对应分区中的数据。一个主题里保存的是同一类消息，每个生产者将消息发送到 Kafka 中，都需要指明该消息所属的主题。每个主题都可以分成多个分区，每个分区在存储层面是日志文件，任何发布到此分区的消息都会被直接追加到文件的尾部，避免单个数据文件达到磁盘存储上限。在分布式的集群中，同一个主题下的分区可以分布在不同的 Broker 上，每个 Broker 负责存储在自己机器上的文件读写。

消息在分区文件中的位置称为偏移量，它是一个 Long 型数字，用于标记一条消息，因此文件只能顺序地读写，基本不存在对数据的随机读写操作。当消息被消费者消费之后，Kafka 也不会立即删除该消息，而是通过配置文件使系统过一段时间后自动删除该消息，以释放磁盘空间。

Kafka 分布式消息存储的实现过程如下。消息保存在主题中，为了实现大数据的存储，一个主题被划分为多个分区，每个分区对应一个文件，可以分别存储到不同的机器上，以实现分布式的集群存储。同时每个分区可以设置一定的副本，备份到多台机器上，以提高可用性。

Kafka 分布式消息备份的实现是通过配置分区的备份数，每个分区将被备份到多台机器上。Kafka 对同一个文件的多个备份进行管理和调度的方案是：每个分区选举一个 Broker 作为 Leader，由 Leader 负责该分区的所有读写，其他 Broker 作为跟随者，只需要保持简单地与 Leader 同步。如果原来的 Leader 失效，则会从该分区的跟随者中重新选一个新的 Leader。

作为 Leader 的服务器承担了该分区所有的读写请求压力，为了确保整体的负载均衡，Kafka 将 Leader 分散到不同的 Broker 上。因为从整体来看，多少个分区就意味着会有多少个 Leader 需要协调。

2. Kafka 的工作流程

Kafka 消费者的消费过程是这样的：每个消费者都属于一个消费组，一个组可以包含多个消费者。订阅主题是以一个消费组的形式来订阅的，发送到主题的消息只会被该主题下的每个消费组中的一个消费者消费。如果所有的消费者都在同一个组，那么就像是一个点对点的消息系统；如果每个消费者都在不同的组，那么消息会广播给所有的消费者。

一个分区只能被一个消费组下的一个消费者消费，但是又可以同时被多个消费组消费，消费组里的每个消费者都关联到一个分区。同一个消费组的两个消费者不会同时消费一个分区消息，但是在消息队列模式下可以共享消息。分区中的消息不存在状态的控制，也没有复杂的消息确认机制，当消息被消费者接收之后，需要保存 Offset 记录。

Kafka 发布-订阅模式消息的工作流程如下。

(1) 生产者与 Topic 下所有分区 Leader 保持 socket 连接,消息由生产者直接通过 socket 发送到 Broker。其中分区 Leader 的位置注册在 Zookeeper 中,生产者作为 Zookeeper 的客户端,可以监听分区 Leader 的变更事件,因此可以获取当前的 Leader。

(2) 当生产者写入一条记录时,指定四个参数: Topic(必需)、Partition(可选)、Key(可选)和 Value(必需)。对于一条数据记录,先对其进行序列化,然后根据 Topic 和 Partition 放进对应的发送队列中。如果没有 Partition,则在存在 Key 值的情况下先对 Key 进行散列计算,相同 Key 取一个 Partition,否则由 Round-Robin(循环/轮转/轮替,用于多种情况中,通常指将多个某物轮流用于某事)来选 Partition。一般是多条消息批量发送到 Broker,避免频繁的 I/O 操作拖慢整体的响应速度。

(3) 当消费者订阅主题时,Kafka 将向消费者提供主题的当前偏移,并且还将偏移保存在 Zookeeper 系统中。消费者将定期请求 Kafka 的新消息,一旦 Kafka 收到来自生产者的消息,它会将这些消息转发给消费者。

(4) 消费者收到消息并进行处理,一旦消息被处理,消费者将向 Broker 发送确认。

(5) Kafka 收到确认,它将偏移更改为新值,并在 Zookeeper 中更新它(在 0.10 版本后,Offset 保存在一个名叫 consumeroffsetstopic 的 Topic 中)。

Kafka 消息队列模式下,消费过程略有不同。同一个消费组中订阅相同主题的消费者则被认为是单个组,并且消息在它们之间共享,具体消费过程如下。

(1) 单个消费者订阅特定主题,Kafka 以发布-订阅模式相同的方式与消费者交互,直到同一消费组下的新消费者也订阅了这个主题。

(2) 新消费者订阅后,Kafka 将其操作切换到共享模式,将在同一个组下的两个消费者之间共享数据,直到用户数达到为该主题配置的分区数。

(3) 一旦消费者的数量超过该主题下的分区数量,新消费者将不会再接收到任何消息,直到现有消费者取消订阅。出现这种情况是因为 Kafka 中的每个消费者都将分配至少一个分区,并且一旦所有分区被分配给现有消费者,新消费者必须等待。

5.3.2 docker 部署 Kafka 集群

下面通过 docker-compose 编排 Kafka 集群,目标是创建一个 zookeeper 容器,两个 Kafka 实例容器,通过 kafka-manager 来管理该集群。新建一个 docker-compose.yml 文件,向该文件写入以下指令。

```
version: '2'
services:
  zookeeper:
    image: zookeeper
    ports:
      - "2181:2181"

  kafka1:
    image: wurstmeister/kafka
    ports:
      - "9092:9092"
      - "9001:9001"
    environment:
      KAFKA_ADVERTISED_HOST_NAME: 192.168.0.104
```

```
      KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://192.168.0.104:9092
      KAFKA_LISTENERS: PLAINTEXT://0.0.0.0:9092
      KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
      JMX_PORT: 9001


  kafka2:
    image: wurstmeister/kafka
    ports:
      - "9093:9092"
      - "9002:9002"
    environment:
      KAFKA_ADVERTISED_HOST_NAME: 192.168.0.104
      KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://192.168.0.104:9093
      KAFKA_LISTENERS: PLAINTEXT://0.0.0.0:9092
      KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
      JMX_PORT: 9002

  kafka-manager:
    image: sheepkiller/kafka-manager
    ports:
      - 9000:9000
    environment:
      ZK_HOSTS: zookeeper:2181
```

docker_com pose.yml 文件创建了四个容器，分别是 zookeeper、kafka1、kafka2、kafka-manager。zookeeper 用于管理创建的两个 Kafka 实例，kafka-manager 是 Kafka 的可视化管理平台。关于 Kafka 实例中的环境变量的解释如下。

(1) KAFKA_ADVERTISED_HOST_NAME：设置主机名，用于其他组件连接。

(2) KAFKA_ADVERTISED_LISTENERS：发布到 zookeeper 的地址，用于客户端的连接。一般使用宿主主机的 IP 并映射端口让外部可以访问，这里需要改成对应的部署主机的 IP。

(3) KAFKA_LISTENERS：指定 Broker 绑定的端口和协议。

(4) KAFKA_ZOOKEEPER_CONNECT：指定连接至 zookeeper 的地址。

完成配置后，在文件目录下通过 docker-compose up 命令构建镜像并启动容器，启动后在浏览器中访问 http://localhost:9000/，打开 Kafka Manager 管理界面，如图 5-12 所示。

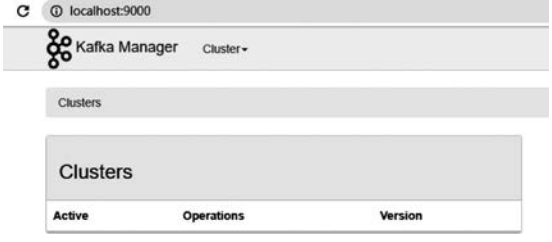

图 5-12　Kafka Manager 管理界面

5.3.3　Kafka 可视化管理

在 Kafka Manager 管理界面中，单击 Clusters 下拉菜单中的 Add Cluster 添加刚才部

署的 zookeeper，Cluster Name 文本框中输入集群名 task，Cluster Zookeeper Hosts 文本框中填入 zookeeper 的地址 zookeeper:2181，并在面板中勾选下列几项。

- Enable JMX Polling (Set JMX_PORT env variable before starting kafka server)
- Enable Logkafka
- Poll consumer information (Not recommended for large # of consumers)
- Enable Active OffsetCache (Not recommended for large # of consumers)

其他配置项使用默认值即可。在页面底部单击 Save 按钮，返回首页即可看见新加的集群在列表中，如图 5-13 所示。

图 5-13　添加 Zookeeper

单击 task 即可进入该集群的管理界面，如图 5-14 所示。在管理菜单中可以直接对主题、分区、备份进行手动管理，同时还有消费者、数据、性能等多项指标的状态监控和管理。

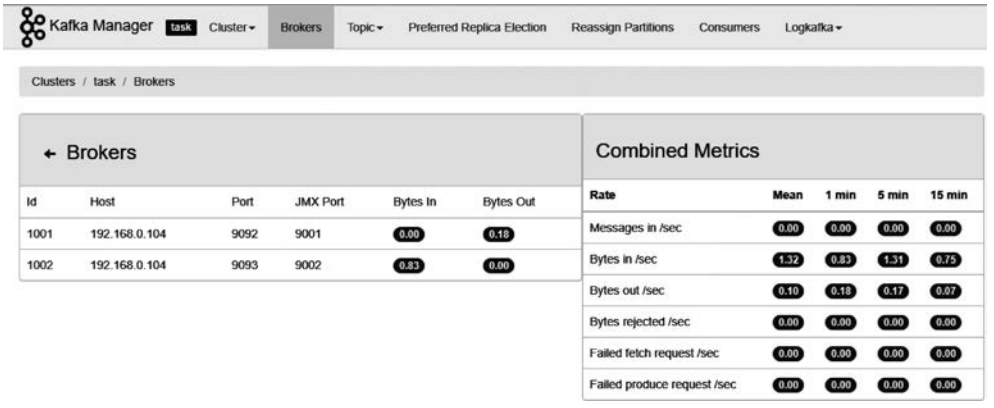

图 5-14　集群管理界面

5.3.4　Python 中使用 Kafka

使用 kafka-python 库操作 Kafka 客户端，首先确保 Python 运行环境中已经正确安装了 kafka-python 库，官方地址是 https://kafka-python.readthedocs.io/en/master/index.html。kafka-python 库实现的客户端类似于官方的 Java 客户端，并带有大量的 pythonic 接口，kafka-python 支持的最低 brokers 版本是 0.8.0。支持消息的压缩发送和数据的自动编码解码，以及封装了 Kafka 提供的高级功能接口。

下面使用 kafka-Python 库结合上面创建的 Kafka 集群实现 Kafka 消费者和生产者的示例，源码如下。

```
from kafka import KafkaProducer, KafkaConsumer, KafkaClient


def kafka_put(topic = "test", msg = "hello world"):
    """
    kafka 生产者,默认消息 hello world
    :param topic:
    :param msg:
    :return:
    """
    producer = KafkaProducer(bootstrap_servers = ["192.168.0.104:9092", "192.168.0.104:
9093"], )
    producer.send(topic, value = msg.encode())
    producer.flush()
    producer.close()


def kafka_get(topic = "test", group_id = "user"):
    """
    kafka 消费者,默认用户组是 user
    :param topic:
    :param group_id:
    :return:
    """
    consumer = KafkaConsumer(topic, bootstrap_servers = ["192.168.0.104:9092", "192.168.
0.104:9093"], group_id = group_id, auto_offset_reset = 'earliest')
    for message in consumer:
        print(f"{message.topic}, {message.partition}, {message.offset}, {message.key},
{message.value}")


if __name__ == '__main__':
    kafka_put()
    kafka_get()#打印 test, 0, 2, None, b'hello world'
```

生产者和消费者中的 bootstrap_servers 参数是客户端用来引导初始集群元数据的链接地址列表,至少需要一个 Broker 的地址来响应元数据的 API 请求,如果未指定服务器则默认为 localhost:9092。

生产者中的 send()方法,向集群的特定主题发送数据,可以通过 value_serializer 参数指定数据的编码方法,一般为匿名方法,例如 lambda m: json.dumps(m).encode()是序列化字典,然后使用 UTF-8 编码,指定后可以直接发送字典数据。flush()方法是不缓冲数据而直接发送,这样订阅的消费者可以立即收到消息。

消费者中的 auto_offset_reset 参数是重置 OffsetOutOfRange 错误的偏移量的策略,只有两个值分别是 earliest 和 latest,前者移至最早的消息,后者移至最新的消息。除此之外,可以通过 value_deserializer()方法指定消息的自动解码如 lambda m: json.loads(m.decode())。

除了上面涉及的一些参数和方法外,kafka-python 还提供了更多的生产者和消费者的实例化参数和 Kafka 服务的接口方法,例如消费者订阅多主题的 subscribe(topics=['topic1', 'topic2'])、生产者绑定消息发送状态的 add_callback()和 add_errback()方法。

基于 Kafka 的分布式爬虫系统的设计思想,是通过 Kafka 消息中间件实现各个分布式主机上的爬虫生产者和消费者之间的通信。一个主题下的爬虫任务将被相同组下的一个消

费者消费，还可以被不同消费组同时消费，基于此特性让同一个网站任务的爬虫在同一个组获取消息，如果要获取不同维度信息的爬虫，可以设置在不同的组，同时接收任务消息。

5.4 基于 Celery 分布式框架的设计

Celery 与之前涉及的 Redis 和消息中间件不同，它是一个成熟的分布式框架，它的消息底层支持 Redis 和 RabbitMQ，是更为高级的分布式 Python 库。Celery 不仅是一款消息队列工具，还是一个任务调度的工具。其特点是简单、灵活、快速、高度可用，它是可靠的分布式框架，可用于调度分布式任务的执行。

5.4.1 Celery 基础

Celery 架构主要由 Task、Broker、Worker、Backend 四部分组成，其组成示意图如图 5-15 所示。Task 是分布执行的任务，可以是异步任务，也可以是定时任务；Broker 是中间人，是任务调度队列，在应用程序和 Worker 之间传递消息；Worker 是任务执行单元，是分布式系统的各个节点，实时监控消息队列并发地运行任务；Backend 是结果后端，用于保存任务结果和状态。

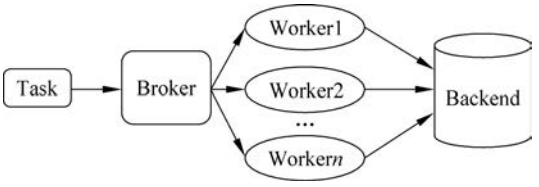

图 5-15 Celery 组成示意图

Celery 通过第三方的消息中间件来实现 Broker。支持的消息中间件包括 RabbitMQ、Redis、Amazon SQS、Zookeeper，Celery 对消息中间件的支持功能如表 5-2 所示。常用 Redis 和 RabbitMQ 作为消息中间件，Celery 对 RabbitMQ 的支持性非常好，推荐使用 RabbitMQ 作为 Broker。

表 5-2 Celery 对消息中间件的支持中间件功能

中 间 件	状 态	是否有监控	是否有远程控制
RabbitMQ	稳定	有	有
Redis	稳定	有	有
Amazon SQS	稳定	无	无
Zookeeper	实验性	无	无

如果要跟踪任务或需要返回值，则 Celery 必须将状态存储或发送到某个地方，以便以后可以检索它们。有几种内置的结果后端可供选择：SQLAlchemy/Django ORM、MongoDB、Memcached、Redis、RPC(RabbitMQ/AMQP)，甚至可以根据开发接口定义自己的后端。

Worker 是包含任务执行程序的应用，它分布运行在各个主机上，与中间人 Broker 通信。当 Broker 收到任务后，会调度给某一个 Worker 执行，如果任务执行时长超过了默认或配置的 visibility_timeout，将指定其他 Worker 重新执行该任务。

Flower 是 Celery 的可视化监控和管理工具，是 Celery 推荐的监视器。Flower 提供了任务的进度和历史信息查询、任务的详情查询(执行参数、开始时间、运行时间、任务结果等)、运行的异常信息查询和追溯，以及聚合的图表和统计信息面板功能。它还具有远程控

制功能，例如查看 Worker 的状态和统计信息，关闭和重启 Worker 实例，查看并修改一个 Worker 实例所指向的任务队列，撤销或终止任务等。

5.4.2　Celery 的使用

1. 创建 Celery

Celery 是由 Python 编写的库，使用前应确保 Python 环境中正确安装了该库，可通过执行 pip install celery 命令完成安装。使用前导入 Celery 的 Celery 实例并初始化，后面的创建任务和管理 Worker 都在 Celery 实例上操作。

创建一个名为 tasks.py 的文件，写入下面的代码，实现一个简单的分布式任务。

```
from celery import Celery

app = Celery('tasks', broker = 'pyamqp://guest@localhost//')

@app.task
def add(x, y):
    return x + y
```

Celery 实例化的第一个参数是当前模块名称，用于启动 Worker 时自动查找模块下定义的任务。对于简单的项目，定义所有内容在一个模块中即可，大型项目应该创建一个文件夹作为专用模块，在模块下定义多个任务文件，启动 Worker 时可以自动发现任务。

Celery 实例化的第二个参数是 Broker 关键字参数，定义消息代理的 URL 链接地址。默认使用 RabbitMQ 作为 Broker，URL 格式为 amqp://host:port，带用户名和密码认证的格式是 amqp://user:password@host:port；使用 Redis 作为 Broker 时的 URL 格式是 redis://hoist:port；如果带密码认证，则 URL 格式是 redis://password@host:port。

除了上面两个常用参数外，常用的还有 backend 参数和 include 参数。backend 参数是指定要使用的结果后端，它用于跟踪任务状态和结果，默认情况下禁用结果。include 参数是 Worker 启动时要导入的模块列表，在列表中添加任务模块，以便 Worker 能找到 task。

2. 注册任务

上面使用@app.task 装饰器注册任务类，注册后的任务在 Worker 启动时可以被远程调用。每个任务类都有一个唯一的名称，并且在消息队列中引用该名称，便于 Worker 找到并执行对应的任务函数。调度的任务在 Worker 确认消息之前，不会从任务队列中删除该任务消息，如果 Worker 被中断(kill)或出现意外情况(例如突然关机)，该 Worker 下的所有的消息会被传递给其他的 Worker。

除了上面的直接创建任务外，还可以通过@task(bind=True)绑定任务，绑定任务通过 task 装饰器的 bind 参数实现，被绑定的任务方法的第一个参数是 self，Python 绑定方法的操作如下。

```
@app.task(bind = True)
def get_id(self):
    return self.request.id
```

3. 启动 Worker

启动 Worker，在 tasks 模块目录下使用如下命令。

```
celery -A tasks worker --loglevel=INFO
```

启动后在控制台会输出如下调试信息，可查看配置(config)、任务队列(queues)、任务函数列表(tasks)。

```
-  ***  ---  *  ---
-  **  ---------- [config]
-  **  ---------- .> app:        tasks:0x2379301ff48
…
---  *****  -----
--------------- [queues]
                .> celery        exchange=celery(direct) key=celery

[tasks]
  . tasks.get_id
```

启动时，默认的并发数为当前计算机的 CPU 数，可以通过设置 celery worker -c 项进行自定义设置并发数。如果要对任务进行速率限制，则通过在启动命令后使用 control 项。如果要指定某个 Worker 运行某个队列的任务，则可以通过-Q 指定监听的任务队列，在没有配置路由任务的情况下默认队列是 celery。Celery 还提供了更多的启动参数设置，运行命令 celery worker --help 可查看支持的启动项参数。例如：

```
celery -A tasks worker --loglevel=INFO -c 2                          #并发数为 2
celery -A tasks worker --loglevel=INFO control rate_limit tasks.add 10/m #限制 add 每分钟
执行 10 次任务
celery -A tasks worker --loglevel=INFO -Q celery                     # 处理默认队列任务
```

如果想在后台启动 Worker 或借助于守护程序(如 Supervisor)，可以使用 celery multi 命令(windows 不支持)在后台启动一个或多个 Worker，启动、重启、停止的命令如下，其中 w1 为启动的 Worker 名。

```
celery multi start w1 -A tasks -l info          #启动
celery multi restart w1 -A tasks -l info        #重启
celery multi stop w1 -A tasks -l info           #停止
```

4. 程序调用

启动 Worker 后，可以在其他应用中导入注册的任务函数，然后在本地运行或者通过远程 Worker 运行任务，代码如下。

```
>>> from tasks import add                                #导入穿件的 tasks.py
>>> add.delay(1,1)                                       #通过 Worker 执行，返回 AsyncResult 对象
<AsyncResult: 00dc9a2a-359a-4f62-ac1f-02aaba65bd49>
>>> add(1, 1)                                            #本地执行，返回执行结果 2
2
```

通过远程 Worker 执行任务，导入任务后调用 delay()方法并传入任务所需要的参数。delay()方法实质上为 apply_async()方法的快捷方法，而 apply_async()可以指定调用时执行的参数，例如运行的时间、使用的任务队列等。例如：

```
>>> add.apply_async((1, 1), queue='lopri', countdown=10)
```

delay()和 apply_async()方法都会返回一个 AsyncResult 实例化对象，同时每个任务被调用时会赋值一个任务 ID(UUIID)。AsyncResult 用于检查任务的状态，等待任务完成或使用 AsyncResult 对象的 get()方法获取任务返回值，或者在任务失败情况下用于获取异常和回溯。

一个任务当前只能有一个状态，如果有多个状态过程，依次状态是 PENDING(挂起)、STARTED(启动)、SUCCESS(成功)/FAILURE(失败)，特殊情况下还有 RETRY(重试)，这些可以通过 AsyncResult. state 查看状态。当决定任务是都重试时，可以通过调用 AsyncResult. failed()来判断是否失败，或通过 AsyncResult. successful()来判断任务是否成功。例如：

```
>>> result = get_id.delay()
>>> result.status
'SUCCESS'
>>> result.state
'SUCCESS'
>>> result.failed()
False
>>> result.successful()
True
```

5. 配置结果后端

默认情况下不启用结果存储，如果要获得任务结果，需要配置 backend 存储任务状态和结果。在上面示例中并没有配置 backend 参数，直接对返回的 AsyncResult 调用 get()方法，将返回错误。如果使用 RabbitMQ 作为结果后端，那么它是特殊的结果后端，因为它并不是存储状态，而是将其作为消息发送，结果只能检索一次，并且只能由发起任务的客户端检索，连接的 URL 以 rpc://开头。修改上面的 Celery 实例，增加 backend 参数，然后重新启动 Worker。代码如下：

```
app = Celery('tasks', broker = 'pyamqp://guest@localhost//', backend = rpc://guest@
localhost//')
```

在 Python 的命令行界面中重新导入任务函数 add()，使用 delay()方法执行并获取 AsyncResult 实例化对象，然后调用 get()方法查询执行结果。代码如下：

```
>>> from tasks import add
>>> result = add.delay(1,1)
>>> result.get()
2
```

6. 日志问题

Celery 使用的是 Python 标准的日志库，运行 Worker 时会自动记录日志信息。手动配置日志记录器可通过 Celery 提供的 get_task_logger()方法在模块顶部为所有任务创建一个共有的日志记录器。例如：

```
logger = get_logger(__name__)


@app.task(bind = True)
```

```
def get_id(self):
    logger.info(f"ID {self.id}")
    return self.request.id
```

7. 独立配置

在复杂的业务中时常需要烦琐的配置，此时需要将配置独立出来，以方便修改。Celery提供了直接读取配置文件和更新当前配置的功能。

支持在代码中更新配置，通过 Celery 下的 update()方法，指定需要更新的参数。例如：

```
from celery import Celery
from celery.utils.log import get_logger

app = Celery('tasks')
app.conf.update(broker = 'pyamqp://guest@localhost//', backend = rpc://guest@localhost//')
```

通过 app.config_from_object()方法可直接从配置文件中读取配置项。在 Celery 4.0 版中引入了小写字母的配置名，同时配置项的前缀命名也有所变化。在新版本中，以前大多数顶级的 CELERY_前缀，都被新的 task_前缀替代，但在 Celery 6.0 版本之前仍然可以读取旧的配置文件，常用新旧配置项参数对照及作用如表 5-3 所示，完整配置项可参考官方文档，地址是 https://docs.celeryproject.org/en/stable/userguide/configuration.html#configuration。

表 5-3 Celery 常用新旧配置项参数对照及作用

Celery 旧版本的配置	Celery 4.0+新版本的配置	参数的作用	配置示例/默认值
CELERY_ACCEPT_CONTENT	accept_content	允许的内容类型、序列化程序的白名单	['json']
CELERY_ENABLE_UTC	enable_utc	如果启用，消息中的日期和时间将转换为 UTC 时区	默认启用
CELERY_IMPORTS	imports	工作程序启动时要导入的一系列模块	["requests"]
CELERY_INCLUDE	include	作用与 imports 相同，设置的模块在 imports 导入之后再导入	["selenium"]
CELERY_TIMEZONE	timezone	指定自定义时区，支持 pytz 库内的时区	默认为 UTC
CELERYBEAT_MAX_LOOP_INTERVAL	beat_max_loop_interval	beat 调度在检查计划之间可以休眠的最大秒数	默认为 0
CELERYBEAT_SCHEDULE	beat_schedule	beat 调度的周期性任务	{}
CELERYBEAT_SCHEDULER	beat_scheduler	默认的调度器类	默认为 celery.beat:PersistentScheduler
CELERYBEAT_SCHEDULE_FILENAME	beat_schedule_filename	存储的周期性任务，最后运行时间的文件的名称	默认为 celerybeat-schedule
CELERYBEAT_SYNC_EVERY	beat_sync_every	在发出另一个数据库同步之前，可以调用的定期任务数	默认为 0
BROKER_URL	broker_url	默认代理 URL	默认为 amqp://

续表

Celery 旧版本的配置	Celery 4.0+ 新版本的配置	参数的作用	配置示例/默认值
BROKER_TRANSPORT_OPTIONS	broker_transport_options	传递给底层传输中间件的附加选项的字典	{'max_retries':5}
BROKER_CONNECTION_TIMEOUT	broker_connection_timeout	放弃与 AMQP 服务器连接之前，默认等待的超时时间	默认为 4 秒
BROKER_CONNECTION_RETRY	broker_connection_retry	如果与 AMQP 消息中间件的连接断开，自动重连	默认启用
BROKER_CONNECTION_MAX_RETRIES	broker_connection_max_retries	放弃与 AMQP 服务器重新建立连接之前的最大重试次数	默认为 100 次
BROKER_LOGIN_METHOD	broker_login_method	设置自定义 amqp 登录方式	默认为 AMQPLAIN
BROKER_POOL_LIMIT	broker_pool_limit	连接池中打开的最大连接数	默认为 10
BROKER_USE_SSL	broker_use_ssl	在消息中间件连接上使用 SSL，支持 pyamqp、redis	默认禁用
CELERY_MONGODB_BACKEND_SETTINGS	mongodb_backend_settings	MongoDB 后端设置，支持 database、taskmeta_collection、max_pool_size、options 字段配置	{'database':'mydb', 'taskmeta_collection': 'my_taskmeta_collection',}
CELERY_EVENT_QUEUE_EXPIRES	event_queue_expires	一个监控客户端事件的队列，被删除前的过期时间	默认为 60 秒
CELERY_EVENT_QUEUE_TTL	event_queue_ttl	一个发送到监控客户端事件队列消息的过期时间	默认为 5 秒
CELERY_EVENT_QUEUE_PREFIX	event_queue_prefix	事件接收队列名称的前缀	默认值为 celeryev
CELERY_EVENT_SERIALIZER	event_serializer	当发送事件消息时使用的消息序列化格式	默认值为 JSON
CELERY_REDIS_DB	redis_db	指定 Redis 使用的库名	"spider"
CELERY_REDIS_HOST	Redis_host	指定 Redis 数据库 host 地址	"118.24.52.111"
CELERY_REDIS_MAX_CONNECTIONS	redis_max_connections	Redis 连接池中用于发送和检索结果的最大连接数	默认无限制
CELERY_REDIS_USERNAME	redis_username	Redis 连接用户名	"Admin"
CELERY_REDIS_PASSWORD	redis_password	Redis 连接密码	"123abc"
CELERY_REDIS_PORT	redis_port	redis 连接端口	6379
CELERY_REDIS_BACKEND_USE_SSL	redis_backend_use_ssl	Redis 后端是否使用 SSL	默认禁用
CELERY_RESULT_BACKEND	result_backend	用来存储结果的后端	result_backend='db+scheme://user:password@host:port/dbname'
CELERY_MAX_CACHED_RESULTS	result_cache_max	启用结果的客户端缓存	默认禁用

续表

Celery 旧版本的配置	Celery 4.0+ 新版本的配置	参数的作用	配置示例/默认值
CELERY_MESSAGE_COMPRESSION	result_compression	结果值的可选压缩方法	默认无压缩
CELERY_RESULT_EXPIRES	result_expires	存储的结果被删除的时间	默认 24 小时后
CELERY_RESULT_PERSISTENT	result_persistent	结果持久化存储	默认禁用
CELERY_ACKS_LATE	task_acks_late	任务消息在任务执行后确认	默认禁用
CELERY_ACKS_ON_FAILURE_OR_TIMEOUT	task_acks_on_failure_or_timeout	task_acks_late 启用时，关于任务的所有消息都将被确认	默认启用
CELERY_ALWAYS_EAGER	task_always_eager	如果为 True 则所有任务将在本地堵塞执行并返回	默认禁用
CELERY_ANNOTATIONS	task_annotations	在配置文件中重写任意任务属性	更改所有任务的 rate_limit 属性：task_annotations={'*':{'rate_limit':'10/s'}}
CELERY_COMPRESSION	task_compression	任务消息的默认压缩算法	默认为 None
CELERY_CREATE_MISSING_QUEUES	task_create_missing_queues	自动创建未声明队列	默认启用
CELERY_DEFAULT_EXCHANGE	task_default_exchange	默认消息交换器	默认为 celery
CELERY_DEFAULT_EXCHANGE_TYPE	task_default_exchange_type	默认消息交换器类型	默认为 direct
CELERY_DEFAULT_QUEUE	task_default_queue	默认消息队列	默认 celery
CELERY_DEFAULT_RATE_LIMIT	task_default_rate_limit	任务的全局默认速率限制，默认无限制	"10/s"
CELERY_DEFAULT_ROUTING_KEY	task_default_routing_key	默认路由键	默认为 celery
CELERY_IGNORE_RESULT	task_ignore_result	是否存储任务返回值	默认禁用
CELERY_PUBLISH_RETRY	task_publish_retry	在连接丢失或其他连接错误的情况下，是否重试发布任务消息	默认启用
CELERY_PUBLISH_RETRY_POLICY	task_publish_retry_policy	定义在连接丢失或其他连接错误的情况下，重试发布任务消息时的默认策略	{'max_retries':3,#最大重试次数 'interval_start':0,#间隔开始 'interval_step':0.2,#间隔步长 'interval_max':0.2,#间隔最大值}

续表

Celery 旧版本的配置	Celery 4.0+ 新版本的配置	参数的作用	配置示例/默认值
CELERYD_SOFT_ TIME _LIMIT	task _ soft _ time _limit	任务的软时间限制，以秒为单位	默认无限制
CELERY_TASK_ TRACK _STARTED	task_track_started	是否显示详细的任务状态信息	默认禁用
CELERYD_TIME_LIMIT	task_time_limit	任务的硬时间限制，超时将“杀死”工作进程并使用一个新的工作进程替代	默认无限制
CELERYD_CONCURRENCY	worker_ concurrency	执行任务的并发工作单元数量	默认是 CPU 的核心数
CELERY_DISABLE_ RATE _LIMITS	worker _ disable _ rate_limits	禁用所有速率限制，包括任务单独设置的速率限制	默认禁用
CELERY_ENABLE_ REMOTE_CONTROL	worker_enable_ remote_control	工作单元的远程控制是否启用	默认启用
CELERYD_HIJACK_ROOT _LOGGER	worker_hijack_root _logger	移除用户配置的根日志记录器的处理函数	默认启用
CELERYD_LOG_COLOR	worker_log_color	启用/禁用 Celery 应用日志输出的颜色	在中端中默认启用
CELERYD_LOG_FORMAT	worker_log_format	日志消息格式	[%（asctime）s:%（levelname） s/%（processName）s]%（message)s
CELERYD_WORKER_ LOST_WAIT	worker_lost_wait	工作单元意外结束，抛出 WorkerLostError 异常之前的等待时间	默认为 10 秒
CELERYD_MAX_TASKS_ PER_CHILD	worker_ max_ tasks _per_child	一个工作单元进程在被一个新的进程替代之前可以执行的最大任务数	默认无限制
CELERYD_POOL_ RESTARTS	worker_pool_ restarts	如果启用，工作单元池可以使用 pool_ restart 远程控制命令进行重启	默认禁用
CELERYD_PREFETCH_ MULTIPLIER	worker_prefetch_ multiplier	工作单元一次性获取的消息数，是这个设置值乘以并发进程的数量	默认为 4
CELERYD_REDIRECT_ STDOUTS	worker_redirect_ stdouts	如果启用，标准输出和标准错误输出将重定向到当前日志器	默认启用
CELERYD_REDIRECT_ STDOUTS_LEVEL	worker_redirect_ stdouts_level	标准输出和标准错误输出的日志级别	默认 WARNING
CELERY_SEND_EVENTS	worker_send_task_ events	发送任务相关的事件，可以被类似 Flower 的工具监控	默认禁用

续表

Celery 旧版本的配置	Celery 4.0+ 新版本的配置	参数的作用	配置示例/默认值
CELERYD_TASK_LOG_FORMAT	worker_task_log_format	任务中记录的日志，使用的格式	[%(asctime)s:%(levelname)s/%(processName)s][%(task_name)s(%(task_id)s)]%(message)s

新建一个配置对象文件 settings.py，写入下列配置。

```
broker_url = 'pyamqp://'                    #Broker 地址
result_backend = 'rpc://'                   #Backend 地址
task_routes = {                             #指定任务监听队列
    'tasks.add': 'low-priority',
}
task_annotations = {                        #限定任务执行速度
    'tasks.add': {'rate_limit': '10/m'}
}
```

然后修改 tasks.py 文件，读取上面的配置对象：

```
from celery import Celery
from celery.utils.log import get_logger

app = Celery('tasks)
app.config_from_object("settings")
```

如果要读取配置对象中指定前缀的配置项，可通过 config_from_object()方法中的 namespace 参数指定配置项的命名空间。

5.4.3 Celery 可视化管理

使用 Celery 可视化管理工具 Flower 之前，需要先安装 Flower 库。常用的启动 Flower 的方式有两种，分别是从 Celery 项目模块启动、通过 Docker 镜像启动。目前前一种方式只适用于 Celery 4.0 版本系列，不兼容最新版本的 Celery。通过 Docker 启动 Flower，不受 Celery 版本的影响，推荐该方式。例如：

```
celery -A tasks.app flower --port=5555          # 从 Celery 项目目录下启动
docker run --name flower -p 5555:5555 mher/flower:0.9.5 flower --broker=pyamqp://
                                                #从 Docker 启动，指定 broker 地址监听任务
```

启动后访问 5555 端口，打开 Flower 监控面板，如图 5-16 所示。有 Dashnard、Tasks、Broker 等的详细列表信息，进入列表还能对相应项进行管理，例如对任务设置限速和执行时间，查看每项任务的详情和结果，如果出现异常会有异常信息反馈、Broker 消息的消费统计情况等。最后一项 Monitor 是各种性能的监控面板，包括任务成功和失败情况、任务执行时间和任务排队情况。

5.4.4 路由任务与定时任务

如果要给不同的机器分配不同的任务来执行，就需要配置任务的路由队列，将不同的任

图 5-16　Flower 监控面板

务分配到对应的队列中，在 Worker 启动时只监听指定队列的任务。如果要让任务每隔一段时间就执行一次，例如每隔 24 小时就自动刷新一下数据，那么就需要使用定时任务。

1. 路由任务

路由任务分为自动路由任务和手动路由任务。自动路由任务适用于简单配置，只需要指定任务和其对应的队列名即可，支持通配符。简单路由任务隐藏了复杂的 AMPQ 协议实现，因为交换机的名称与队列的名称一致，所以非 AMPQ 的后端组件（如 Redis、SQS）也适用。例如：

```
app = Celery('tasks', broker = 'amqp:// ', backend = 'rpc:// ')
logger = get_logger(__name__)
app.conf.task_routes = {'tasks.add': {'queue': 'add'}, 'tasks.get_ * ': {'queue': 'get_id'}}#
简单路由
app.conf.task_default_queue = 'add'#更改默认监听队列


@app.task(bind = True)
def get_id(self):
    logger.info(f"ID {self.request.id}")
    return self.request.id


@app.task
def add(x, y):
    return x + y
```

通过 task_routes 配置任务的队列，配置后匹配的任务会被发送到指定的队列。但是 Worker 还是监听的默认队列 celery，需要通过 task_default_queue 修改监听的默认队列，或者在启动 Worker 时指定需要监听的队列，这样才会获得相应任务数据并执行。如果要在一台 Worker 上监听上面源码中的 add 和 get_id 队列，启动命令时可以添加 Q 参数，代码如下。

```
celery -A tasks worker --loglevel = INFO -Q add,get_id
```

手动路由是通过创建队列实例指定队列、交换机、路由键，其配置类似于使用 RabbitMQ 的设置，队列实例的定义是通过 Kombu 来实现的，Kombu 为 AMQ 协议通信提供了高级接口，使 Python 中的消息传递尽可能简单。下面是一个定义路由队列的实例。

```
app = Celery('tasks', broker = 'pyamqp:/ ', backend = 'rpc:// ')
logger = get_logger(__name__)
```

```
from kombu import Queue

app.conf.task_queues = (                                    #设置队列
    Queue('add', routing_key = 'add'),
    Queue('get_id', routing_key = 'get_id'),
)
app.conf.update(                                            #设置默认的交换机信息
    task_default_exchange = 'tasks',
    task_default_exchange_type = 'topic',
    task_default_routing_key = 'task.default')
app.conf.task_routes = {                                    #添加路由键
    'tasks.add': {
        'queue': 'add',
        'routing_key': 'add',
        'priority': 10,                                     #优先级
    },
    'tasks.get_id': {
        'queue': 'get_id',
        'routing_key': 'get_id',
        'priority': 5,
    },
}
```

task_queues 是一个包含 Queue 实例的列表，默认运行的 Worker 将监听所有创建的队列，但是可以通过启动参数中的-Q 指定 Worker 需要监听的队列。如果不想修改默认的 exchange 和 exchange_type 的值，将自动设置为 task_default_exchange 和 task_default_exchange_type。

增加 task_routes 的目的是将任务路由到指定的任务队列中。同时在 RabbitMQ 和 Redis 作为 Broker 的 Celery 实例中，可以通过 priority 指定队列的默认优先级。除了使用 task_routes 配置让任务路由到指定队列，还可以通过 Task.apply_async()重载队列和路由键。例如：

```
>>> result = add.apply_async(args = (1,1),queue = "add",routing_key = "add")
>>> result
< AsyncResult: 01028cc7 - 31e9 - 47d1 - 9bd8 - 3ccd11019192 >
>>> result.state
'SUCCESS'
>>> result.get()
2
```

2. 定时任务

Celery Beat 是一个调度程序，它定期启动任务，然后由集群中的 Worker 执行任务。默认情况下会从配置中的 beat_schedule 项中获取条目，但是也可以从 SQL 等数据库中获取。需要注意的是调度程序要在单独的进程中执行，应确保一次只运行一个调度程序，否则最终将导致任务重复。

默认情况下，定期任务计划使用 UTC 时区，但是可以使用时区设置更改使用的时区。例如配置 Asia/Shanghai，代码如下。

```
timezone = 'Asia/Shanghai'#或 app.conf.timezone = 'Asia/Shanghai'
```

beat_schedule 配置项是一个字典，含有计划名和计划字段的字典集合，例如每隔 30 秒执行一次 add，代码如下。

```
app.conf.beat_schedule = {
    'add-every-30-seconds': {
        'task': 'tasks.add',
        'schedule': 30.0,
        'args': (1, 1)
    },
}
app.conf.timezone = 'UTC'
```

配置字段中的 task 是指要执行任务的名称，schedule 是执行周期，单位是秒，args 是任务所需的位置参数，需要传递关键字参数可以通过 kwargs 字段实现。除此之外，还有 options 字段，用于传递 apply_async()方法支持的任何参数（如 exchange、routing_key、expires 等）。

启动 Celery Beat 服务，可以单独启动 Beat，也可以嵌入 Worker 一起启动。启动后，Beat 需要将任务的最后运行时间存储在本地数据库文件（默认情况下命名为 celerybeat-schedule）中。命令如下。

```
celery -A tasks beat              #单独启动
celery -A tasks worker -B         #嵌入 Worker 启动
```

除了上面使用的是默认调度 celery.beat.PersistentScheduler 外，还有 Crontab 调度器和 Solar 调度器。Crontab 调度器可以精确控制任务执行时间，例如在一天或一周的某时某分某秒更新。Solar 调度器可以根据日出、日落、黎明或黄昏等事件来执行任务，因此使用该调度器不仅要指明事件，还要指明经纬度。更多事件类型可以参考地址 https://docs.celeryproject.org/en/stable/userguide/periodic-tasks.html#crontab-schedules 中的信息。

```
from celery.schedules import crontab

app.conf.beat_schedule = {
    #每周一早上七点半执行
    'add-every-monday-morning': {
        'task': 'tasks.add',
        'schedule': crontab(hour=7, minute=30, day_of_week=1),
        'args': (16, 16),
    },
}

from celery.schedules import solar

app.conf.beat_schedule = {
    # 在墨尔本日落时执行
    'add-at-melbourne-sunset': {
        'task': 'tasks.add',
        'schedule': solar('sunset', -37.81753, 144.96715),
        'args': (16, 16),
    },
}
```