消息中间件

- Celery（4.2）：分布式任务队列
  - Celery的定义
    - 分布式系统，可用于处理大量消息
    - 消息队列工具，处理实时数据以及任务调度
  - Celery的作用
    - 实现异步任务和周期任务
    - 控制每秒/分钟/小时执行任务的次数
    - 高可用：失连接或连接失败，重试
    - 快速：每分钟可以处理数以百万的任务
  - Celery的架构
    - Task：任务模块

      包含异步任务和定时任务。 异步任务通常在业务逻辑中被触发并发往任务队列 定时任务由 Celery Beat 进程周期性地将任务发往任务队列
    - Broker：消息中间件

      Broker：即为任务调度队列，接收任务生产者发来的消息（即任务），将任务存入队列。 Celery 本身不提供队列服务，官方推荐使用 RabbitMQ 和 Redis 等
    - Worker：任务执行单元

      Worker 是执行任务的处理单元，它实时监控消息队列，获取队列中调度的任务，并执行它
    - Backend：任务结果存储

      Backend 用于存储任务的执行结果，以供查询。同消息中间件一样，存储也可使用 RabbitMQ, Redis 和 MongoDB 等
  - Celery的并发方式
    - 进程池并发（默认）
    - 协程并发（eventlet）
  - Celery的监控工具
    - Flower
  - Question
    - Celery丢失任务
      - 意外退出，设置task被放回到队列中
      - 只有当worker完成了task时，任务才被标记为ack状态
    - Celery重复执行

      系统负载较高，消息队列因堵了太多东西的情况下，Celery将禁止现重复执行一个Task，甚至不止一次的情况。 Message超过1小时未被消费的情况下，Celery会重新发一个一模一样的（task_id相同）
      - Celery Once：去重

        Celery Once 是利用 Redis 加锁来实现 该类提供了任务去重的功能
      - 增大未消费时间
- APScheduler定时任务框架
  - triggers（触发器）

    触发器就是根据你指定的触发方式，比如是按照时间间隔，还是按照 crontab触发，触发条件是什么等。 触发器包含调度逻辑，每一个作业有它自己的触发器。
    - date：固定日期触发器
    - interval： 时间间隔触发器
    - cron：固定时间点执行任务
  - job stores（作业存储）

    可以存储任务的地方，默认情况下任务保存在内存中，也可以保存到各种数据库中支持MongoDB、Redis、SQLAlchemy存储方式。 当对作业任务进行持久化存储的时候，作业的数据将被序列化。重新读取作业时在反序列化。
    - MemoryJobStore（默认）
    - Sqlalchemy
    - MongoDB
    - Redis
  - executors（执行器）

    安排任务到线程池或者进程池中运行的 执行器用来执行定时任务，只是将需要执行的任务放在新的线程或者线程池中运行。 当作业任务完成时，执行器将会通知调度器。 对于执行器，默认情况下选择ThreadPoolExecutor就可以了，但是如果涉及到一下特殊任务或比较消耗CPU的任务则可以选择ProcessPoolExecutor， 当然根据根据实际需求可以同时使用两种执行器。
    - 线程池（默认）
    - 进程池
  - schedulers（调度器）

    任务调度器是属于整个调度的总指挥官。合理安排作业存储器、执行器、触发器进行工作。开进行添加和删除任务等。 一般在应用程序中只有一个调度器，应用开发者不会直接操作触发器、任务存储以及执行器， 相反调度器提供了处理的接口。通过调度器完成任务的存储以及执行器的配置操作。如可以添加、修改、移除任务作业。
    - BlockingScheduler：运行单个任务
    - BackgroundScheduler：后台运行
    - AsyncIOScheduler：Async异步框架
    - GeventScheduler：Gevent框架
    - TornadoScheduler：Tornado框架
    - TwistedScheduler: Twisted框架
    - QtScheduler: 适合QT
- 消息队列
  - 消息队列
    - 消息队列的概念

      应用间的通信方式，消息发送后可以立即返回，由消息系统来确保消息的可靠传递。
      - 消息系统来确保消息的可靠传递
    - 消息队列的特点
      - 采用异步处理模式

        发送者可以发送一个消息而无须等待响应，接收者无需对消息发送者做出回步回应
      - 发送者和接收者不必同时在线
      - 发送者和接收者不必了解对方，只需要确认消息
    - 消息队列的应用
      - 支付完成，统修改订单支付状态
      - 短信通知、终端状态推送、App 推送、用户注册
  - 常用的消息中间件
    - ActiveMQ：曝光率最高，但是可能会丢消息
    - ZeroMQ：不支持消息持久化和崩溃恢复
    - Kafka：定位是日志消息队列, 吞吐量最大
    - RabbitMQ：可靠。支持持久化，发送确认，接收确认
- RabbitMQ（3.7）

  是一个开源的AMQP（高级消息队列协议）实现，服务器端用Erlang语言编写
  - RabbitMQ的组成
    - Producer：消息生产者
    - Consumer：消息消费者
    - Exchange：消息队列交换机
      - fanout 发布订阅

        把消息路由到与该交换机绑定的所有队列上。用于发布订阅
      - direct：关键字发送

        把消息路由到跟绑定binding key与routing key完全匹配的Queue中。用于关键字发送
      - topic：模糊匹配

        把消息路由到跟绑定binding key与routing key按规则完全匹配的Queue中。用于模糊匹配
      - headers：消息中属性匹配

        根据发送的消息内容中的headers属性进行匹配 有两种模式： 全部匹配 部分匹配
    - Queue：消息队列
  - RabbitMQ分发（消息）模式
    - 简单模式

      包含一个生产者、一个消费者和一个队列。生产者向队列里发送消息，消费者从队列中获取消息并消费。
    - 工作模式

      包含一个生产者、两个消费者和一个队列
      - 平均分配

        每个消费者处理相同个数的任务
      - 循环调度

        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。 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。
    - 发布订阅模式

      一个生产者、两个消费者，两个队列和一个交换机。两个消费者同时绑定到不同的队列上去，两个队列绑定到交换机上去，生产者通过发送消息到交换机， 所有消费者接收并消费消息。 交换类型 exchange_type='fanout'
    - 路由模式（关键字）

      包含一个生产者、两个消费者，两个队列和一个交换机。 两个消费者同时绑定到不同的队列上去，两个队列通过路由键绑定到交换机上去，生产者发送消息到交换机。 交换机通过路由键转发到不同队列，队列绑定到消费者接收并消费消息。 交换类型exchange_type='direct'
    - 通配符模式（模糊匹配）

      通过路由键匹配规则转发到不同队列 特殊匹配符号：*：只能匹配一个单词； #：可以匹配零个或多个单词 交换类型exchange_type='topic'
    - RPC双向通道

      场景：客户端发送消息给服务端,服务端处理完后将结果返回.客户端获取到数据 处理：客户端往队列1送完数据后,建个新的队列2,服务端在队列1接收数据后,将处理后将结果放到队列2返回.客户端在队列2处接收 对于RPC请求，客户端发送带有两个属性的消息： reply_to，设置为回调队列， correlation_id，设置为每个请求的唯一值。
      - 作用：消费者能了解到任务处理成功或者失败
      - RPC的实现机制（原理）

        1. 客户端发送请求（消息）时，在消息的属性中设置两个值replyTo（Queue名称，告诉服务器处理消息发送到这个Queue 中）和correlationId（此次请求的标识号）。2. 服务器端收到消息并处理 3. 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 4. 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
  - 消息机制
    - 消息的路由（routing/binding）

      消息创建时设定一个路由键（routing key），通过队列路由键（bind key），把队列绑定到交换器上。 RabbitMQ 会将消息的路由键与队列的路由键进行匹配（针对不同的交换器有不同的路由规则）
    - 消息的传输（Channel信道模式）

      由于 TCP 连接的创建和销毁开销较大，且并发数受系统资源限制，会造成性能瓶颈。 RabbitMQ使用信道的方式来传输数据。信道是建立在真实的TCP连接内的虚拟连接，自每条 TCP 连接上的信道数量没有限制。
    - 保证消息发送至MQ（发送方确认模式）

      1. 将信道设置成 confirm 模式（发送方确认模式），则所有在信道上发布的消息都会被指派一个唯一的 ID。 2. 一旦消息被投递到目的队列后，或者消息被写入磁盘后（可持久化的消息），信道会发送一个确认给生产者（包含消息唯一 ID）。 3. 如果RabbitMQ发生内部错误从而导致消息丢失，会发送一条 nack（notacknowledged，未确认）消息 4. 发送方确认模式是异步的，生产者应用程序在等待确认的同时，可以继续发送消息。当确认消息到达生产者应用程序，生产者应用程序的回调方法就会被触发来处理确认消息
    - 保证消息被消费（接收方确认模式）

      丢失消息的原因 消费者默认采用自动确认模式，消息在发送后立即被认为是发送成功。如果这时处理消息失败，就会丢失该消息： 手动确认： 消费者接收每一条消息后都必须进行确认，只有消费者确认了消息，RabbitMQ 才能安全地把消息从队列中删除。 无超时机制， 只要连接不中断，RabbitMQ 给了 Consumer 足够长的时间来处理消息。 特殊情况： 1.如果消费者接收到消息，在确认之前断开了连接或取消订阅。RabbitMQ会认为消息没有被分发，然后重新分 发给下一个订阅的消费者。 2.如果消费者接收到消息却没有确认消息，连接也未断开，则RabbitMQ认为该消费者繁忙，将不会给该消费者分发更多的消息。
    - 保证消息不被重复消费
      - 生产时消息重复
        - 原因

          生产者发送消息给MQ，在MQ确认的时候出现了网络波动，生产者没有收到确认。实际上MQ已经接收到了消息。这时候生产者就会重新发送一遍这条消息。
        - 解决方案

          MQ内部针对每条生产者发送的消息生成一个inner-msg-id，这个值全局唯一，且由MQ生成。与业务无关
      - 消费时消息重复
        - 原因

          消费者消费成功后，再给MQ确认的时候出现了网络波动，MQ没有接收到确认。为了保证消息被消费，MQ就会继续给消费者投递之前的消息。这时候消费者就接收到了两条一样的消息。
        - 解决方案

          消息体中，必须有一个与业务相关的唯一id（，如支付ID，订单ID，帖子ID等） 由发送方放到消息体中，消费方对ID进行判重，保证重新投递的消息只有1条消息被消费
    - 保证消息的顺序性
      - 乱序的原因

        业务上产生三条消息，分别是对数据的增加、修改、删除操作，如果没有保证顺序消费，执行顺序可能变成删除、修改、增加。这就乱了，多个消费者并发消费消息，获取的消息的速度，执行业务的逻辑的速度快慢，执行异常等等原因都会导致消息的顺序不一致。
      - 解决方案

        将需要保证顺序的数据发到一个队列中，该队列只有一个消费者消费
  - RabbitMQ持久化
    - 交换机的持久化
    - 队列的持久化

      channel.queue_declare（queue = 'hello'，durable = True）
    - 消息的持久化

      channel.basic_publish(exchange='', routing_key="task_queue", body=message, properties=pika.BasicProperties(delivery_mode = 2, # make message persistent))
  - RabbitMQ的可靠传输
    - 生产者丢失消息：发送方确认模式
    - 消息列表丢失消息：队列/消息的持久化
    - 消费者丢失消息：消费者手动确认