一，celery是什么

1.celery简介

> celery是python开发的基于分布式消息传递的异步任务队列系统

2.特点

- 高可用

  工作者和客户端可以在链接丢失或者失败时自动进行重试

  单个 Celery 进程可以在亚毫秒级的延迟下，每分钟处理数百万个任务（基于 librabbitmq 库使用 RabbitMQ，并且优化过配置）

- 高速

  Celery 几乎每一个部分都可以单独使用或进行扩展，

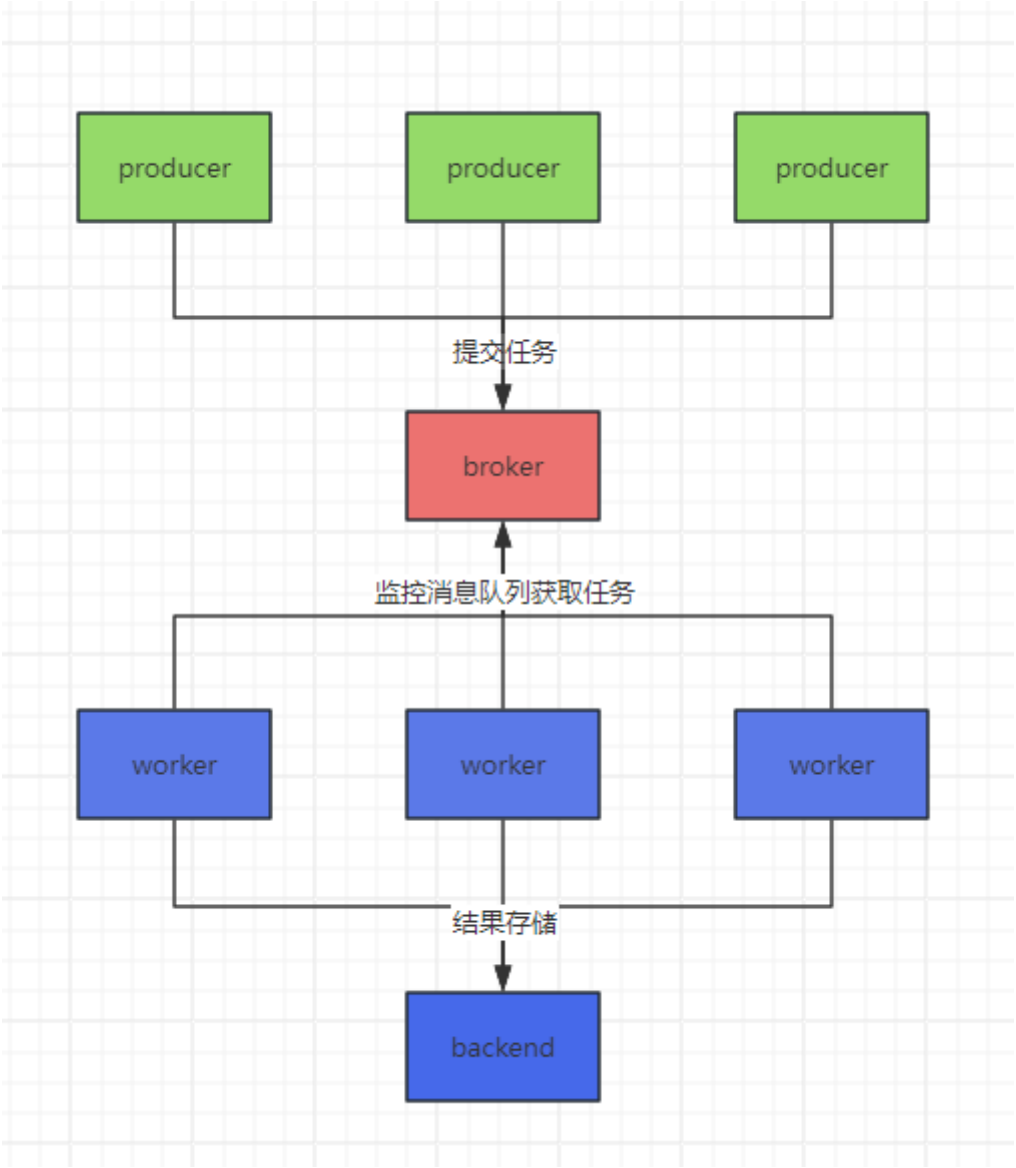
  例如自定义池、序列化器、压缩方案、日志、调度器、消费者、生产者、传输代理等等的实现。

- 灵活的

  Celery 几乎每一个部分都可以单独使用或进行扩展，

  例如自定义池、序列化器、压缩方案、日志、调度器、消费者、生产者、传输代理等等的实现。

3.详细架构



各个部分详解:

- **producer**: 任务生产单元

  任务调用者，使用代码调用

- **broker**：消息中间件

  提交的任务函数都放在这里，celery本身不提供消息中间件

  常用的消息中间件：redis，rabbitmq

- **worker**：任务执行单元

  监控消息中间件，从里面获取任务函数，进行执行

- **backend**：任务存储器

  负责存储任务，常用的存储器：redis，rabbitmq，sqlalchemy，Django ORM

4.使用场景

- **异步任务**：用来处理耗时操作，异步执行防止堵塞应用，比如：web应用中的上传文件，计算任务进行分布式处理等待
- **定时任务**：Celery 支持周期性任务，如每天固定时间备份数据库、定期清理过期数据或发送定时邮件等

- **延迟任务**：支持延迟操作，让某个任务在未来的特定时间点执行，例如在用户注册后的一段时间内发送欢迎邮件，或者是在某个事件发生后的延迟时间内执行某些操作

二、功能点使用

1.基本使用

- **安装celery 和flower 监控**

```
pip install celery
pip install flower
pip install eventlet # 并发模型，启动的时候需要使用
```

- **配置参数**

```
from datetime import timedelta
from kombu import Queue, Exchange

class BaseCeleryConfig:
    # 消息代理（Broker）和结果后端
    broker_url = 'redis://:123456@192.168.137.137:6379/0'  # 指定 Broker 地址
    result_backend = 'redis://:123456@192.168.137.137:6379/1'  # 指定结果存储地址

    # 任务序列化和反序列化使用的格式
    task_serializer = 'json'
    result_serializer = 'json'

    # 可接受的内容类型
    accept_content = ['json']

    # 设置时区
    timezone = 'Asia/Shanghai'

    # 是否启用 UTC 时间
    enable_utc = True

    # 并发 worker 数量
    worker_concurrency = 4

    # 每个 worker 最多执行的任务数
    worker_max_tasks_per_child = 100

    # 任务失败后是否重新排队
    task_acks_late = False

    # 在每个 worker 启动时预取的任务数量
    worker_prefetch_multiplier = 4

    # 设置任务超时时间（秒）
    task_time_limit = 30 * 60  # 30 minutes

    # 设置结果过期时间（秒），默认一天
    result_expires = 24 * 60 * 60  # 1 day
```

```
    # 任务限流
    task_annotations = {'tasks.add': {'rate_limit': '10/s'}}

    # 定义多个队列
    task_queues = (
        Queue('priority_low', Exchange('priority_low'),
routing_key='priority_low'),
        Queue('priority_high', Exchange('priority_high'),
routing_key='priority_high'),
        Queue('default', Exchange('default'), routing_key='default'),
    )

    # 定义路由规则，将不同的任务分配到不同的队列中
    task_routes = {
        'task.priority_high': {'queue': 'priority_high', 'routing_key':
'priority_high'},
        'task.priority_low': {'queue': 'priority_low', 'routing_key':
'priority_low'},
        'task.add': {'queue': 'default', 'routing_key': 'default'},
    }

    # 定时任务调度器的配置
    beat_schedule = {
        'add-every-30-seconds': {
            'task': 'task.add',
            'schedule': 30.0,
            'args': (16, 16)
        },
    }
```

- **定义任务**

```
import time
from celery import Celery
from config import BaseCeleryConfig

app = Celery('task')

app.config_from_object(BaseCeleryConfig)

@app.task
def add(x, y):
    time.sleep(5)
    return x + y

@app.task
def priority_high(messgae):
    time.sleep(5)
    print(messgae)
    return "priority_high"

@app.task
def priority_low(messgae):
    time.sleep(5)
    print(messgae)
    return "priority_low"
```

- **添加任务**

```
from task import add
# 使用任务
result = add.delay(1, 2)
print(result.get())
```

- **启动celery和flower**

```
celery -A task beat --loglevel=info #启动周期性定时任务提交方
celery -A task worker --loglevel=info --pool=eventlet # 启动celery
celery -A task flower --port=5555   # 启动flower 监控
```

  - `-A` 参数指定了包含 Celery 应用实例的模块或包名
  - `-l` 或 `--loglevel`：设置日志输出的详细程度，默认为 `WARNING`
  - `-c` 或 `--concurrency`：设置并发数，默认是 CPU 核心数
  - `-P` 或 `--pool`：指定 worker 使用的池类型（如 prefork, eventlet, gevent, threads）
  - `--queues` 或 `-Q`：指定要监听的任务队列名称
  - `--hostname` 或 `-n`：给 worker 指定一个自定义的名字
  - `-B` 或 `--beat`：启动时同时运行 beat 调度器（不推荐用于生产环境，应分别启动 Beat 和 Worker）

2.delay和`apply_async`

2.1 delay

- **定义**：delay是apply_async的简化版本，用于快速提交任务
- **语法**：

```
task.delay(arg1, arg2, keyward=value)
```

- **示例**：

```
from task import add
result = add.delay(4, 6)
```

2.2 apply_async

- **定义**：`apply_async()` 是 Celery 提供的更灵活的任务调度方法，支持更多高级选项
- **常用参数详解**：

参数名	类型	说明	示例
`args`	`tuple`	任务的位置参数。	`args=(4, 6)`
`kwargs`	`dict`	任务的关键字参数。	`kwargs={'x': 4, 'y': 6}`
`countdown`	`int`	任务延迟执行的秒数。	`countdown=10`（10秒后执行）
`eta`	`datetime`	任务预计执行时间（UTC 时间）。	`eta=datetime.utcnow() + timedelta(seconds=10)`
`expires`	`int`/`datetime`	任务过期时间（秒或 UTC 时间），过期后任务会被丢弃。	`expires=60`（60秒后过期）
`task_id`	`str`	为任务分配唯一 ID（默认由 Celery 生成 UUID）。	`task_id='my_custom_task_id'`
`retry`	`bool`	任务失败后是否自动重试（默认 `True`）。	`retry=True`
`retry_policy`	`dict`	重试策略配置（如最大重试次数、重试间隔等）。	`retry_policy={'max_retries': 3, 'interval_start': 0, ...}`
`queue`	`str`	指定任务发送到的队列。	`queue='high_priority'`
`exchange`	`str`	指定任务发送到的交换机（Exchange）。	`exchange='default_exchange'`
`routing_key`	`str`	自定义路由键。	`routing_key='custom.key'`
`priority`	`int`	任务优先级（0-255，数值越小优先级越高）。	`priority=0`（最高优先级）
`serializer`	`str`	任务序列化格式（如 `json`、`yaml`）。	`serializer='json'`
`compression`	`str`	数据压缩方案（如 `zlib`、`bzip2`）。	`compression='zlib'`
`link`	`list`	任务成功后的回调任务（链式任务）。	`link=[signature('task2')]`
`link_error`	`list`	任务失败后的回调任务。	`link_error=[signature('error_handler')]`
`shadow`	`str`	任务在日志中显示的名称（覆盖默认名称）。	`shadow='custom_task_name'`

- **使用示例**

```
from task import add
result = add.apply_async(args=[1, 2])
```

3.AsyncResult 类

3.1 概念

> 在 Celery 中，`AsyncResult` 是一个用于获取异步任务执行结果的类。当通过 `delay()` 或 `apply_async()` 调用任务时，会返回一个 `AsyncResult` 实例。通过它，可以查询任务状态、获取结果、处理异常等。

3.2 作用

- **任务状态查询**：查看任务是否完成、失败或正在重试。
- **结果获取**：阻塞或非阻塞地获取任务结果。
- **异常处理**：捕获任务执行过程中的异常。

3.3 常用属性

属性名	类型	说明
`id`	`str`	任务的唯一标识符（UUID）。
`state`	`str`	任务当前状态（如 `PENDING`、`SUCCESS`、`FAILURE` 等）。
`result`	`any`	任务返回的结果（成功时为结果，失败时为异常对象）。
`scoreboard`	`dict`	内部状态记录（调试用）。

3.4 常用方法

- **get(timeout=None, propagate=True)**
  - 作用：阻塞当前线程，直到任务完成并返回结果
  - 参数：
    - timeout：等待超时时间，超时后会报错
    - propagate：是否将任务重的异常抛出
  - 示例：

```
from task import add
result = add.delay(4, 6)
```

- **ready**
  - **作用**：检查任务是否已经执行完成 。完成(True)，未完成（False）
  - **示例**:

```
from task import add
result = add.delay(4, 6)
if result.ready():
    print("已完成")
```

- **successful()**
  - 作用：检查任务是否成功执行完成。完成(True)，未完成（False）
  - 示例：

```python
from task import add
result = add.delay(4, 6)
if result.successful():
    print("已完成")
```

- **failed()**
  - 作用：检查任务是否失败，`True`（失败）或 `False`（成功/未完成
  - 示例：

```python
from task import add
result = add.delay(4, 6)
if result.failed():
    print("任务失败")
```

- **revoke(terminate=False, signal='SIGTERM')**
  - 作用：尝试取消任务（需 Broker 支持，如 Redis/AMQP）
  - 参数：
    - terminate：是否强制终止任务（True 立即终止）
    - signal：发送的型号
  - 示例：

```python
from task import add
result = add.delay(4, 6)
result.revoke(terminate=True)
```

- **forget()**
  - 作用：从结果后端中删除任务结果，释放空间
  - 示例：

```python
from task import add
result = add.delay(4, 6)
result.forget()
```

3.5 任务状态（`state`）

状态	含义
`PENDING`	任务尚未被处理（等待分配给 Worker）。
`STARTED`	任务已被 Worker 接收并开始执行。
`SUCCESS`	任务成功执行，结果已存储。
`FAILURE`	任务执行失败（抛出异常）。
`RETRY`	任务因异常被标记为重试（需配置重试策略）。
`REVOKED`	任务被手动取消（通过 `revoke()`）。