## **Capítulo 55. Executando tarefas assíncronas**

Guilherme Arthur de Carvalho

doc de correções solicitadas por Juliana Guamá:  
<https://docs.google.com/document/d/1b28WgwDseBbNQhN80HrZwqSOqHXyg7j9-Zi6pb-lqs0/edit?usp=sharing>

Neste Capítulo iremos aprender como escrever aplicações que executem tarefas de forma assíncrona utilizando Celery.

### **Processos síncrono e assíncrono**

Basicamente podemos dividir nossas tarefas em duas categorias: tarefas síncronas e assíncronas. Dizemos que uma tarefa é síncrona quando os processos acontecem de forma sequencial e assíncronas quando um ou mais processos acontecem ao mesmo tempo.

Vamos imaginar duas tarefas, estudar matemática e escutar música. No cenário síncrono primeiramente preciso iniciar a tarefa de estudo de matemática e após finalizar essa tarefa início a tarefa de escutar música. Já no cenário assíncrono eu posso iniciar a tarefa de estudar matemática e enquanto estudo posso iniciar a tarefa de escutar música.

### **Quando utilizar tarefas síncronas ou assíncronas**

O processamento síncrono deve ser utilizado quando precisamos da resposta para prosseguir o fluxo de execução da aplicação, exemplo: Tomar banho e vestir o uniforme escolar. Não posso vestir o uniforme enquanto tomo banho.

Já o processamento assíncrono é uma boa ideia quando conseguimos otimizar os intervalos de tempo que iremos ficar ociosos entre uma tarefa e outra, exemplo: Tomar café da manhã e ler as notícias.

Toda tarefa assíncrona pode ser realizada de forma síncrona, porém o inverso nem sempre é verdade. Tente analisar o seu cenário e se perguntar: *A tarefa X precisa esperar a tarefa Y terminar de executar?*, se a resposta for **SIM** então provavelmente essa tarefa tem que ser **síncrona**.

### **CPU bound e IO bound**

O termo *bound* para ambos os casos auxilia na identificação de processos ou sistemas que são limitados pela CPU ou IO (input/output ou entrada/saída). Uma alusão é imaginar que você tenha uma grande quantidade de areia, mas tenha uma ampulheta. A parte mais fina da ampulheta limita o tempo de execução da passagem da areia de um lado para o outro. Essa parte fina seria o que faz o “bound” da areia. Outros exemplos:

* Uma tarefa sistema que calcule números primos entre 0 e 1000 será tão rápido quanto o CPU puder processar. Dessa forma, é chamado de **CPU bound**
* Um leitor de arquivo “.csv” será tão rápido quanto o sistema de IO da máquina. Portanto é um processo **IO bound.**

### **Conhecendo o Celery**

O Celery é um projeto open source amplamente adotado pela comunidade Python. Ele é um gerenciador de tarefas assíncronas com base na passagem de mensagens distribuídas. O foco é na operação em tempo real mas também suporta agendamentos.

Cada unidade de execução é chamada de tarefa, que são executadas simultaneamente em um ou mais servidores de trabalho (workers). As tarefas podem ser executadas de forma assíncrona (em segundo plano) ou síncrona (aguarde até que esteja pronto).

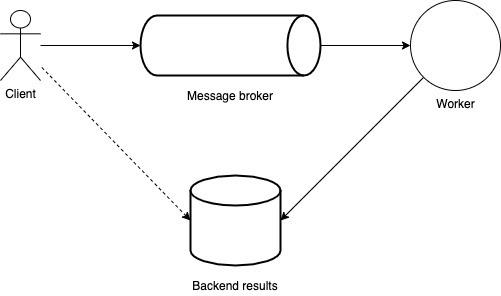


Figura 1.55. Fluxo de funcionamento do Celery.

1. O cliente (aplicação web ou script) insere uma mensagem no broker.
2. O broker armazena as mensagens e despacha para os workers. Na maioria das vezes o **RabbitMQ** ou **Redis** são utilizados como brokers.
3. O worker executa a tarefa e armazena o resultado da execução (salvar o resultado é opcional).
4. Cliente pode analisar os resultados da execução das tarefas, caso elas tenham sido salvas.

### **Instalando o nosso broker**

Como vimos na imagem acima o Celery precisa de uma solução para enviar e receber mensagens, que são os **brokers**. Existem várias opções disponíveis, e para nosso laboratório iremos utilizar o **RabbitMQ**.

Poderíamos instalar diretamente no sistema operacional o programa, porém vamos utilizar o **Docker** que torna o processo muito mais simples.

|  |
| --- |
| $ docker run --name=livropython\_cap55 -d -p 5672:5672 -p 15672:15672 rabbitmq:3-management |

### **Instalando o Celery**

O Celery pode ser instalado diretamente com o gerenciador de pacotes do Python (PIP).

|  |
| --- |
| $ pip install celery |

### **Calculando números primos**

Vamos desenvolver uma simples aplicação para calcular números primos em um intervalo determinado pelo usuário. Nossa tarefa irá receber dois parâmetros: início e fim da sequência e irá retornar todos os números primos. Crie um arquivo chamado **tasks.py**.

|  |
| --- |
| **from** typing **import** List  **from** celery **import** Celery  app = Celery("tasks", broker="pyamqp://guest@localhost//")   @app.task **def** **prime\_numbers**(start: int, end: int) -> List[int]:  **return** [n **for** n **in** range(start, end + 1)  **if** n != 1 **and** all(n % m != 0 **for** m **in** range(2, n))] |

Na linha 3 informamos qual o endereço do broker para o Celery se conectar. Após isso na linha 4 usamos o decorador **task**, fornecido pelo Celery. Todas as funções anotadas com esse decorador serão tratados como tarefas gerenciadas pelo framework.

Agora que já temos o **broker** configurado precisamos inicializar o **worker**. Para isso vamos fazer uso do utilitário de linha de comando **celery**.

|  |
| --- |
| $ celery -A tasks worker --loglevel=info |

Se tudo correu bem você deve ver uma saída como essa em seu terminal:

|  |
| --- |
| celery@MacBook-Pro-de-Guilherme.local v4.4.2 (cliffs)  macOS-10.15.3-x86\_64-i386-64bit 2020-03-22 15:29:09  [config] .> app: tasks:0x10eec6520 .> transport: amqp://guest:\*\*@localhost:5672// .> results: disabled:// .> concurrency: 4 (prefork) .> task events: OFF (enable -E to monitor tasks in this worker)  [queues] .> celery exchange=celery(direct) key=celery   [tasks]  . tasks.prime\_numbers  [2020-03-22 15:29:09,650: INFO/MainProcess] Connected to amqp://guest:\*\*@127.0.0.1:5672// [2020-03-22 15:29:09,743: INFO/MainProcess] mingle: searching for neighbors [2020-03-22 15:29:11,066: INFO/MainProcess] mingle: all alone [2020-03-22 15:29:11,109: INFO/MainProcess] celery@MacBook-Pro-de-Guilherme.local ready. |

Agora já estamos prontos para enviar uma mensagem ao nosso **broker**, e esperar que o **worker** execute o trabalho.

|  |
| --- |
| **from** tasks **import** prime\_numbers  prime\_numbers.delay(1, 1000) |

Vá até o terminal onde deixamos o **worker** executando, e veja que o Celery imprimiu o retorno da função no log. Porém não conseguimos utilizar esse retorno, já que não informamos qual o **ResultBackend**.

### **Configurando o ResultBackend**

O Celery possui uma configuração padrão que é quase sempre suficiente para a maioria dos cenários. Por esse motivo, para o nosso exemplo precisamos alterar apenas uma linha.

|  |
| --- |
| app = Celery('tasks', backend='rpc://', broker='pyamqp://') |

Pronto! agora podemos recuperar os resultados obtidos na execução da tarefa por meio do método **get**.

|  |
| --- |
| **from** tasks **import** prime\_numbers  result = prime\_numbers.delay(1, 1000) print(result.get()) |

### **Retentativas de tarefas**

Nem sempre as tarefas irão terminar com sucesso na primeira tentativa de execução, por isso é interessante definirmos tratativas de erros. Vamos alterar nossa função para que lance erros quando for executada em minuto par e apenas calcular com sucesso em minutos ímpares.

|  |
| --- |
| @app.task(bind=True, default\_retry\_delay=60) **def** **prime\_numbers**(self, start: int, end: int) -> List[int]:  minute = datetime.utcnow().minute   **if** minute % 2 == 0:  **raise** self.retry(  exc=Exception("Error in prime\_numbers function!"),  countdown=60  )   **return** [n **for** n **in** range(start, end + 1)  **if** n != 1 **and** all(n % m != 0 **for** m **in** range(2, n))] |

Execute a função em um minuto par, por exemplo: 17:04. Vá até o terminal onde o **worker** está executando e analise o log.

|  |
| --- |
| [2020-03-22 17:04:30,994: INFO/ForkPoolWorker-2] Task tasks.prime\_numbers[32ed681e-a49f-4556-b79a-8d1371feb9d7] retry: Retry in 60s: Exception('Error in prime\_numbers function!') [2020-03-22 17:04:31,000: INFO/MainProcess] Received task: tasks.prime\_numbers[32ed681e-a49f-4556-b79a-8d1371feb9d7] ETA:[2020-03-22 20:05:30.992327+00:00] |

Perceba que a execução falhou e a mesma será executada novamente dentro de 60 segundos, que foi o período que definimos. Após 60 segundos passados, ainda no terminal do **worker**, podemos ver que a função foi executada com sucesso.

|  |
| --- |
| [2020-03-22 17:04:30,994: INFO/ForkPoolWorker-2] Task tasks.prime\_numbers[32ed681e-a49f-4556-b79a-8d1371feb9d7] retry: Retry in 60s: Exception('Error in prime\_numbers function!') [2020-03-22 17:04:31,000: INFO/MainProcess] Received task: tasks.prime\_numbers[32ed681e-a49f-4556-b79a-8d1371feb9d7] ETA:[2020-03-22 20:05:30.992327+00:00] [2020-03-22 17:05:31,408: INFO/ForkPoolWorker-2] Task tasks.prime\_numbers[32ed681e-a49f-4556-b79a-8d1371feb9d7] succeeded in 0.011209459000042443s: [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103, 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173, 179, 181, 191, 193, 197, 199, 211, 223, 227, 229, 233, 239, 241, 251, 257, 263, 269, 271, 277, 281, 283, 293, 307, 311, 313, 317, 331, 337, 347, 349, 353, 359, 367, 373, 379, 383, 389, 397, 401, 409, 419, 421, 431, 433, 439, 443, 449, 457, 461, 463, 467, 479, 487, 491, 499, 503, 509, 521, 523, 541, 547, 557, 563, 569, 571, 577, 587, 593, 599, 601, 607, 613, 617, 619, 631, 641, 643, 647, 653, 659, 661, 673, 677, 683, 691, 701, 709, 719, 727, 733, 739, 743, 751, 757, 761, 769, 773, 787, 797, 809, 811, 821, 823, 827, 829, 839, 853, 857, 859, 863, 877, 881, 883, 887, 907, 911, 919, 929, 937, 941, 947, 953, 967, 971, 977, 983, 991, 997] |

### **Integrando Celery com Flask**

Agora que já entendemos o funcionamento básico do Celery, vamos criar uma aplicação Web para disparar emails utilizando o framework Flask para fazer a camada Web.

Primeiro vamos criar um novo ambiente virtual utilizando **pyenv**, e instalar as dependências necessárias:

|  |
| --- |
| $ pyenv virtualenv 3.8.2 cap55 $ pyenv activate cap55 $ pip install flask flask-mail celery |

Vamos precisar também de um servidor SMTP, novamente utilizaremos uma imagem **Docker** pronta.

|  |
| --- |
| $ docker run -d -p 1080:1080 -p 1025:1025 --name mailcatcher schickling/mailcatcher |

No navegador entre com o endereço <http://localhost:1080/> para acessar a caixa de entrada do nosso servidor de email.

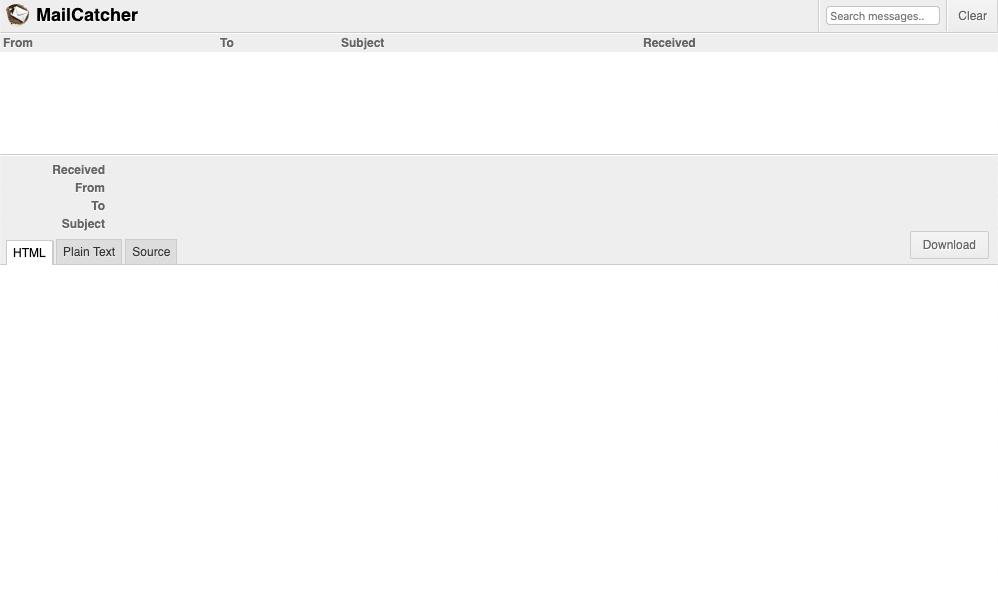


Figura 2.55. Caixa de entrada.

Vamos criar nossa aplicação Web e integrá-la com o Celery. Crie o arquivo **app.py**.

|  |
| --- |
| **from** datetime **import** datetime  **from** celery **import** Celery **from** flask **import** Flask, jsonify **from** flask\_mail **import** Mail, Message  app = Flask(\_\_name\_\_) app.config["MAIL\_PORT"] = 1025 app.config["CELERY\_RESULT\_BACKEND"] = "rpc://" app.config["CELERY\_BROKER\_URL"] = "pyamqp://guest@localhost//"  mail = Mail(app)  celery = Celery(  app.import\_name,  backend=app.config["CELERY\_RESULT\_BACKEND"],  broker=app.config["CELERY\_BROKER\_URL"] ) celery.conf.update(app.config)   **class** **ContextTask**(celery.Task):  **def** **\_\_call\_\_**(self, \*args, \*\*kwargs):  **with** app.app\_context():  **return** self.run(\*args, \*\*kwargs)   celery.Task = ContextTask  users = []   @app.route("/users/<name>") **def** **create\_user**(name: str = "Anonymous") -> **None**:  users.append({"name": name, "created\_at": datetime.utcnow()})   send\_mail.delay(name)   **return** jsonify({"message": "Email successfully sent!"})   @celery.task(bind=True, default\_retry\_delay=60) **def** **send\_mail**(self, name: str) -> **None**:  msg = Message(  subject="Seja bem vindo",  html=f"<p>Olá {name}, <br><br>Seja bem vindo ao livro <b>Jornada Python</b>.</p>",  sender="noreply@jornadapython.com.br",  recipients=["{name}@gmail.com"]  )   **try**:  mail.send(msg)  **except** Exception **as** exc:  **raise** self.retry(exc=exc, countdown=60, max\_retries=5) |

Nossa aplicação tem apenas um endpoint, ele é do tipo GET e pode ser acessado em <http://127.0.0.1:5000/users/Guilherme>. Quando for requisitado a função **create\_user** será executada, e irá inserir o usuário na lista de usuários, chamar a função **send\_mail** e retornar uma mensagem de sucesso.

Antes de requisitar a URL precisamos iniciar o servidor de aplicação do Flask e o **worker** do Celery. Os comandos devem ser executados em terminais separados.

|  |
| --- |
| $ FLASK\_APP=app.py flask run |

|  |
| --- |
| $ celery -A app.celery worker --loglevel=info |

Pronto! agora podemos voltar ao navegador e acionar o endpoint <http://127.0.0.1:5000/users/Guilherme>. Você pode substituir **Guilherme** passando o seu nome. Vamos até a nossa caixa de entrada para conferir se o e-mail foi recebido.

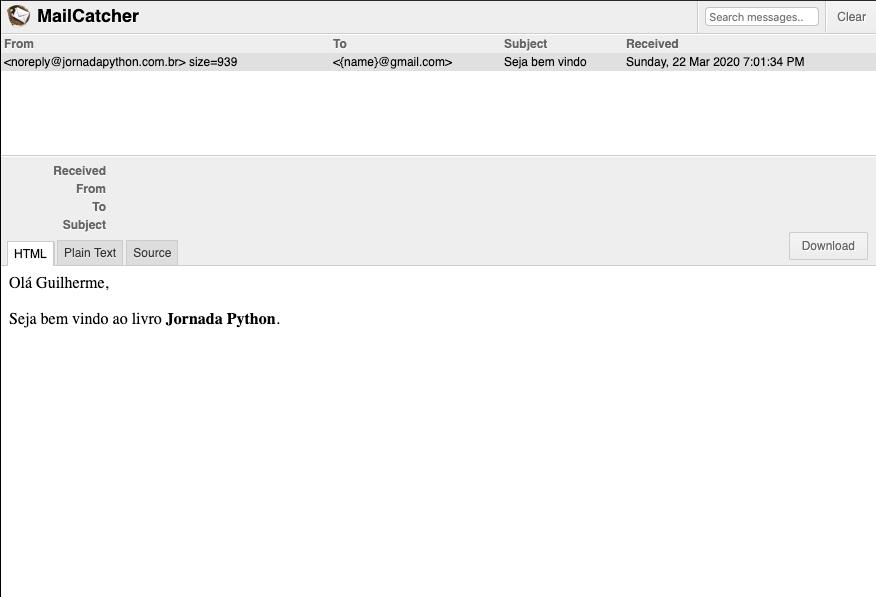


Figura 3.55. Caixa de entrada com e-mail recebido.

Com isso encerramos o nosso Capítulo sobre tarefas assíncronas com Python, para mais informações e tópicos avançados não deixe de conferir a documentação do Celery.

### **Referências**

Distributed Task Queue - Celery Documentation. Site. Disponível em: <<https://docs.celeryproject.org/en/latest/index.html>>