

非同期タスクキューを使って業務効率化した話

みんなのPython勉強会#100

2024-01-25

@hrs_sano645

みんなのPython勉強会#100 おめでとうございます

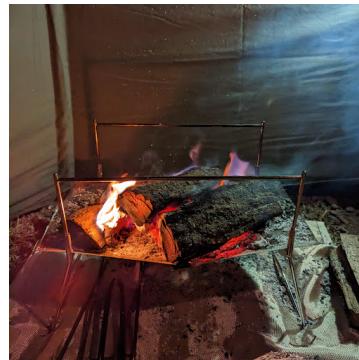
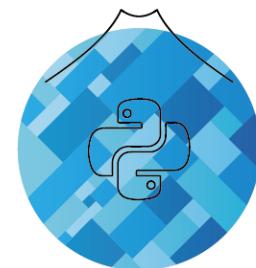
お前誰よ / Self Introduction

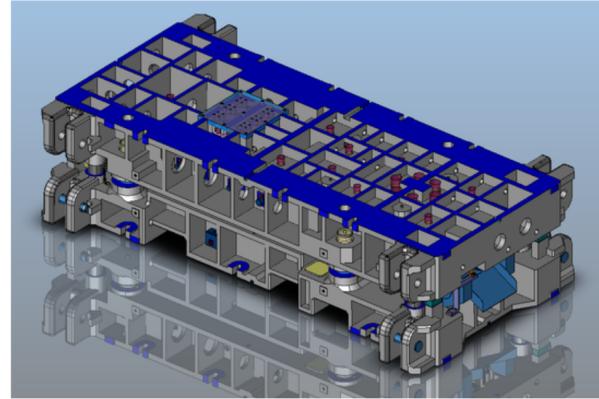
佐野浩士 (Hiroshi Sano) [@hrs_sano645](#)

- 🌍: 静岡県富士市🗻
- 🏢: 株式会社佐野設計事務所 代表取締役
- 💬🤝
 - 🐍: PyCon mini Shizuoka Stuff / Shizuoka.py / Unagi.py / Python駿河
 - CivicTech, Startup Weekend Organizer
- Hobby: Camp🏕, DIY🛠️, IoT💡



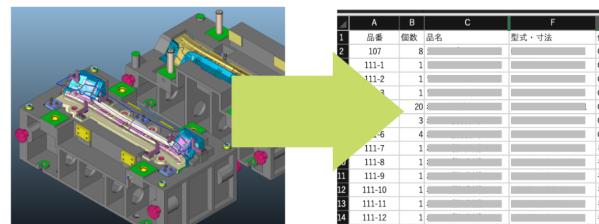
**PyCon
mini
SHIZUOKA**





3D機械設計 / 3Dモデリング

自動車向けプレス金型の3D設計
金型設計関連の3Dモデリング
製品データの3Dモデリング



製造業DX支援



設計データをデジタル化
製造業DX取り組みサポート

2023年の社内個人的目標: 業務効率化を限界まで進める

効率化する理由

- とある依頼ベースの案件業務
- 今までではそれほど多くなかったが、**年を跨いで急激に増える**
 - 人力でやっていては**追いつかん**そう。やばい
- 人間が必要な部分以外、**あらゆる手作業を止める！** → 成功した!



どんなことを効率化？

- **手作業していたこと自動化**

- 依頼受注（メール）→ボイラープレートツールで作業プロジェクトフォルダーを生成
- Googleスプレッドシート連携して案件管理
- 会計サービスと連携して見積書/請求書生成（書類作成）
- 依頼企業側のシステム連携: WEBスクレイピング

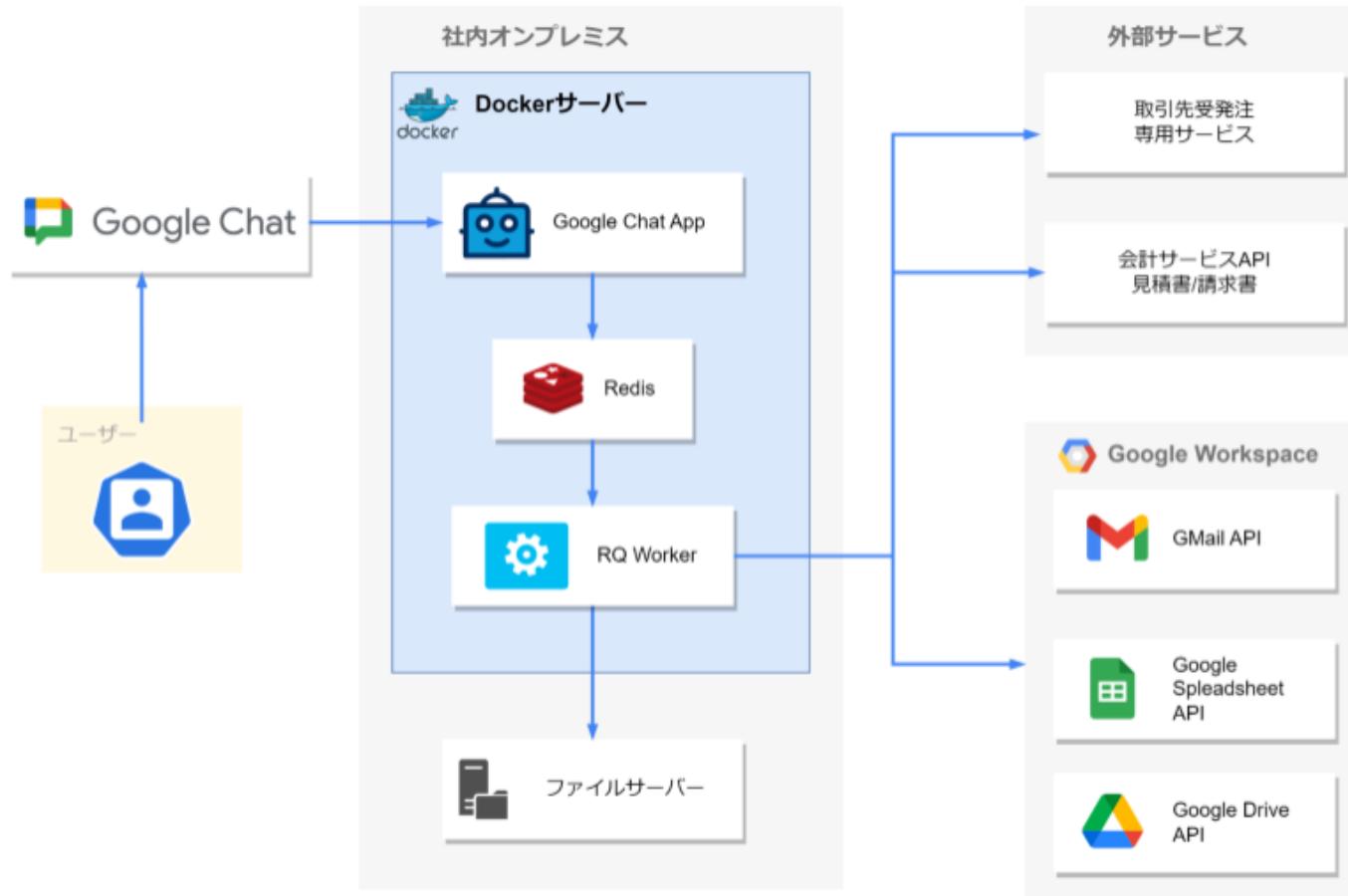
- **自動処理の実行をChatOpsで**

- 処理を動かしていくか確認した上で実行させる
- Google Chatでチャットボット作成
- スマホから実行もできる

究極には指一本で仕事をしたい



業務タスク自動化サービスの構成



「非同期タスクキュー」 を使って作業させています

非同期で動かすため + 処理をまとめたタスクを + キューに入れて実行させる

なんでタスクキューを使う？

- 自動処理→重い処理だった
 - ファイル操作、APIアクセス -> I/Oバウンド処理
 - 組み合わせると**数秒ではなく数十秒～分単位**の処理
 - 結果が返ってくるタイミングはその時次第
- チャットボットの仕様->**素早く応答しないといけない**
 - チャット側がすぐにボット側に応答を求める
 - チャットボットは非同期前提

Google Chatの場合

「同期的に応答するには、Chat アプリが 30 秒以内に応答し、その応答をインタラクションが発生したスペースに投稿する必要があります。それ以外の場合は、Chat アプリは非同期で応答できます。」

<https://developers.google.com/chat/api/guides/message-formats?hl=ja#sync-response>

(Slackの3秒よりも全然緩いけど、非同期前提な様子)

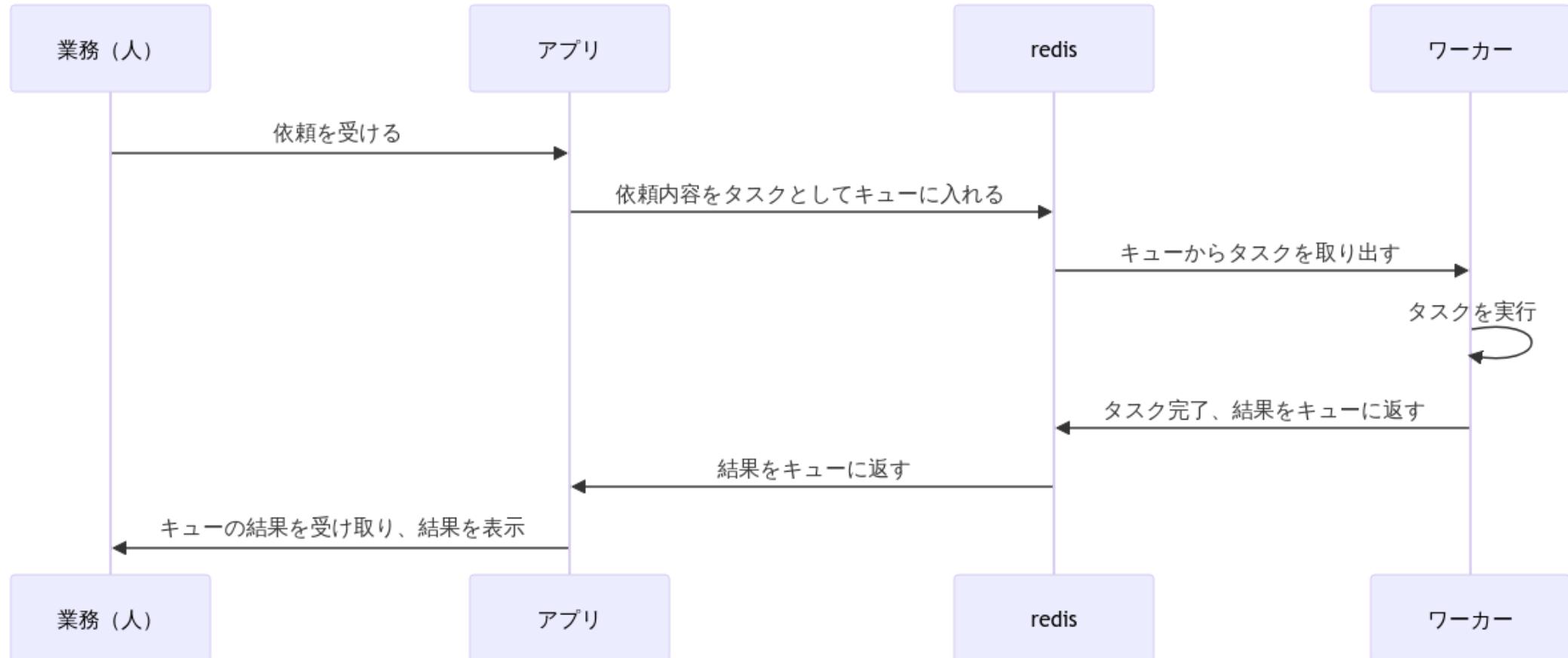
RQ(python-rq)を使いました

python-rq: <https://python-rq.org/>

以下の3つの要素で構成される

- アプリ: タスク発行→キューへ入れる→ワーカーから処理結果を受け取る
- ワーカー: タスクの処理を行う
- redis: アプリとワーカーの間に入りキューとして利用する

RQのざっくりイメージ



なぜRQを選んだのか

ドキュメント見ていたら簡単に見えて良かった

- celeryと悩んだ -> celeryを使うほどの規模ではなかった
- asyncioと悩んだ -> RQがシンプルだった

*I/Oバウンズ処理はasyncio, multiprocessingは制限にならないので、
この選択肢がベストとは限らない（速度とか）

ということで、お手軽でdocker composeで試す場合のサンプルです

参考: [Python で分散タスクキュー \(RQ 編\) #Python - Qiita @hoto17296](#)

Dockerfile

```
FROM python:3.11
RUN pip install rq
```

compose.yml

```
version: '3'
services:
  redis:
    image: redis
  worker:
    build: .
    depends_on:
      - redis
    environment:
      RQ_REDIS_URL: redis://redis
    command: rq worker
    volumes:
      - .:/app
    working_dir: /app
  app:
    build: .
    depends_on:
      - redis
      - worker
    environment:
      RQ_REDIS_URL: redis://redis
    command: python app.py
    volumes:
      - .:/app
    working_dir: /app
```

ファイル操作をしてみる

tasks.py

```
from pathlib import Path
import random
import string
import sys

def create_random_string(length):
    """指定された長さのランダムな文字列を生成する関数"""
    letters = string.ascii_letters + string.digits
    return ''.join(random.choice(letters) for i in range(length))

def create_files(num_files, file_size, directory="test_files"):
    """指定された数とサイズのファイルを生成する関数"""
    Path(directory).mkdir(parents=True, exist_ok=True)

    for i in range(num_files):
        savefile = Path(f"{directory}/file_{i}.txt")
        with savefile.open("w") as f:
            f.write(create_random_string(file_size))
```

app.py

```
import os
import redis
from rq import Queue
from tasks import create_files

NUM_FILES = 100
FILE_SIZE = 1048576
NUM_TASKS = 3

q = Queue(connection=redis.from_url(os.environ.get("RQ_REDIS_URL")))

# タスクの実行をキューに投げる
tasks = [
    q.enqueue(create_files, args=(NUM_FILES, FILE_SIZE, f"test_files_{i}"))
    for i in range(NUM_TASKS)
]
```

実行

```
# シングルワーカー  
$ docker compose up  
  
# マルチワーカー: 3つのワーカーを起動  
$ docker compose up --scale worker=3  
## ログは別途ファイルでみせます
```

ワーカーのログ: シングルワーカー

1つのワーカーにタスクが順番に渡されて実行される

```
file_task-worker-1 | 03:20:35 Worker rq:worker:81027d039a944dc3b8a230519243f68e started with PID 1, version 1.15.1
file_task-worker-1 | 03:20:35 Subscribing to channel rq:pubsub:81027d039a944dc3b8a230519243f68e
file_task-worker-1 | 03:20:35 *** Listening on default...
file_task-worker-1 | 03:20:35 default: tasks.create_files(50, 1048576, 'test_files_0') (f18592d2-16ba-4c82-98ef-11da85c44493)
file_task-app-1 exited with code 0
file_task-worker-1 | 03:20:52 default: Job OK (f18592d2-16ba-4c82-98ef-11da85c44493)
file_task-worker-1 | 03:20:52 Result is kept for 500 seconds
file_task-worker-1 | 03:20:52 default: tasks.create_files(50, 1048576, 'test_files_1') (9f4a596f-73af-4973-9007-af03da8f5057)
file_task-worker-1 | 03:21:09 default: Job OK (9f4a596f-73af-4973-9007-af03da8f5057)
file_task-worker-1 | 03:21:09 Result is kept for 500 seconds
file_task-worker-1 | 03:21:09 default: tasks.create_files(50, 1048576, 'test_files_2') (bf5c15ad-3222-45be-ab8a-3f214a57700d)
file_task-worker-1 | 03:21:27 default: Job OK (bf5c15ad-3222-45be-ab8a-3f214a57700d)
file_task-worker-1 | 03:21:27 Result is kept for 500 seconds
```

ワーカーのログ: マルチワーカー

3つそれが並列に動いて実行される

```
file_task-worker-3 | 03:19:26 Worker rq:worker:a3fae5de17c34f658f597ce4d5543dbc started with PID 1, version 1.15.1
file_task-worker-3 | 03:19:26 Subscribing to channel rq:pubsub:a3fae5de17c34f658f597ce4d5543dbc
file_task-worker-3 | 03:19:26 *** Listening on default...
file_task-worker-3 | 03:19:26 Cleaning registries for queue: default
file_task-worker-3 | StartedJobRegistry cleanup: Moving job to FailedJobRegistry (due to AbandonedJobError)
file_task-worker-3 | StartedJobRegistry cleanup: Moving job to FailedJobRegistry (due to AbandonedJobError)
file_task-worker-3 | StartedJobRegistry cleanup: Moving job to FailedJobRegistry (due to AbandonedJobError)
file_task-worker-2 | 03:19:26 Worker rq:worker:3f33ea404d2040e99961f0a2d5d46b1f started with PID 1, version 1.15.1
file_task-worker-2 | 03:19:26 Subscribing to channel rq:pubsub:3f33ea404d2040e99961f0a2d5d46b1f
file_task-worker-2 | 03:19:26 *** Listening on default...
file_task-worker-1 | 03:19:26 Worker rq:worker:32c4d2c46a944ba3b05dcb295cc522c2 started with PID 1, version 1.15.1
file_task-worker-1 | 03:19:26 Subscribing to channel rq:pubsub:32c4d2c46a944ba3b05dcb295cc522c2
file_task-worker-1 | 03:19:26 *** Listening on default...
file_task-worker-3 | 03:19:27 default: tasks.create_files(50, 1048576, 'test_files_0') (a2767c5e-2caa-47ed-b8f2-204881f671ac)
file_task-worker-2 | 03:19:27 default: tasks.create_files(50, 1048576, 'test_files_1') (288a07b9-80eb-4736-9e1c-56887781babe)
file_task-worker-1 | 03:19:27 default: tasks.create_files(50, 1048576, 'test_files_2') (49f43d8d-b0c2-4edb-a4b3-711852102a9f)
file_task-app-1 exited with code 0
file_task-worker-2 | 03:19:44 default: Job OK (288a07b9-80eb-4736-9e1c-56887781babe)
file_task-worker-2 | 03:19:44 Result is kept for 500 seconds
file_task-worker-1 | 03:19:44 default: Job OK (49f43d8d-b0c2-4edb-a4b3-711852102a9f)
file_task-worker-1 | 03:19:44 Result is kept for 500 seconds
file_task-worker-3 | 03:19:44 default: Job OK (a2767c5e-2caa-47ed-b8f2-204881f671ac)
file_task-worker-3 | 03:19:44 Result is kept for 500 seconds
```

dockerで動かす時

- 手っ取り早い方法として**タスク側もワーカー側も同じ環境=Dockerfileを使う**
- RQはredis（キュー）へタスクを渡すときはpickleを使ってる
 - ワーカー側でもpickleで渡されたオブジェクトが理解できないといけない
-> ワーカー側にも同じライブラリをインストールする必要がある
- タスクとワーカーを同時に動かすならcomposeが便利
- コード参照や利用するボリュームも同じ箇所を参照すると楽

まとめ

- **膨大な手作業は間違えるので自動化しよう**
- 自動化は重い処理をよく扱う -> 非同期前提で考える
 - **外部サービス連携、チャットボット、LLMのAPIとの連携も**
- 非同期タスクキューを使うことで、重い処理を任せられ、自動化の幅や連携方法が広がる（はず）

Google Chatアプリの話はまたどこかで～

参考

- メッセージキュー - [Wikipedia](#)
- [python-rq](#)
- [【Pythonで高速化】I/Oバウンドとか並列処理とかマルチプロセスとかってなんぞや #Python - Qiita](#)
- [docker利用時の参考: Python で分散タスクキュー \(RQ 編\) #Python - Qiita](#)
- [Python3.12で新たにサポートされたsub-interpretersの紹介 | gihyo.jp](#)

サンプルコード

<https://github.com/hrzano645/exam-python-rq-by-docker>