

# Django & Celery in production

DjangoCongress JP 2021

2021/07/03 | Masataka Arai

# お前誰よ？



- Masataka Arai @massa142
- SQUEEZE Inc.
- Pythonもくもく会 主催
- Python Boot Camp 講師

**宣伝！**



Masashi Shibata

@c\_bata\_

翔泳社(@shoeisha\_books)さんから『実践Django Pythonによる本格Webアプリケーション開発』を出版します。サンプルコードや目次はGitHubで公開しました。

中級者以上の方にも参考になる踏み込んだ解説も盛り込んであるのでぜひ読んでみてください！

## c-bata/practical-django-book



7月19日発売「実践Django Pythonによる本格Webアプリケーション開発 (翔泳社)」サンプルコード。

Amazon予約受け付け中。



1

Contributor



0

Issues



33

Stars



0

Forks



c-bata/practical-django-book

7月19日発売「実践Django Pythonによる本格Webアプリケーション開発 (翔泳社)」サンプルコード。Amazon予約受け付け中。 - c-bata/practical-django-...

[github.com](https://github.com/c-bata/practical-django-book)



- 『実践Django Pythonによる本格Webアプリケーション』 2021/7/19 発売 🎉
  - 自分はちょっとレビューに参加しました
- 個人的なおすすめポイント
  - Djangoの話にとどまらないWebアプリケーション開発に必要なDBの話が丁寧
  - Mixinとか認証認可のところなど、チュートリアルにはない設計についての知見が得られる

今日はDjangoで非同期処理をするときに、よく使われているCeleryについて話していきます。

Celery 使ったことない人 🙋

Celery 使ってるけど、雰囲気ですらってる人 🙋



今日はCeleryの

- 基本的なトピック
- 本番運用で知っておくとよいこと

を話していきます。

今回扱わないこと

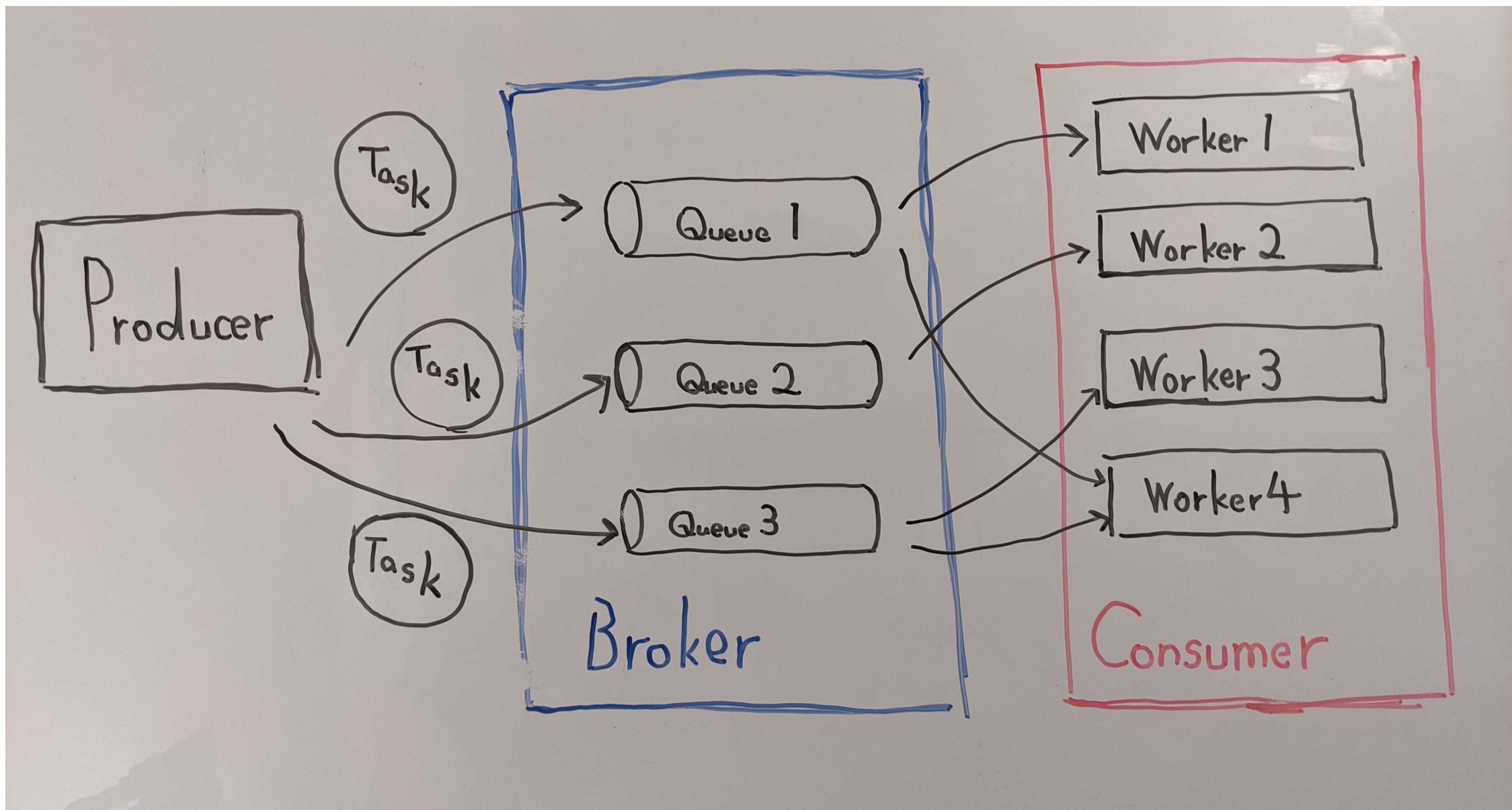
- Django x Celeryの初期設定
- インフラ構成

# 目次

- タスクキューとは
- Celeryの基本Tips
- ワークフローデザイン
- リトライ設計
- ログ保存
- 監視
- デプロイ戦略

タスクキューとは

## 概念图



## 用語集

- **Task:** 非同期で実行させる処理のまとまり
- **Queue:** Taskを格納する入れもの (FIFO: First-In-First-Out)
- **Producer:** Taskを作成してBrokerに渡す (Celeryだと「Celery Client」がこの役割)
- **Broker:** 作成されたTaskをQueueに登録したり、Queueに登録されているTaskをConsumerに渡す
- **Consumer:** BrokerによってQueueから取り出されたTaskを実際に処理する (Celeryだと「Celery Worker」がこの役割)

## メリット

- 時間がかかる処理などを非同期に逃がせる
  - HTTPリクエストをより多く処理できるように
- 処理を分散できるので、スケールがしやすい
- エラーが発生してもリトライできるので、復元力の高いシステムに

## デメリット

- アーキテクチャが複雑になる
- 監視・ログなどの考えごとが増える
- 処理の遅延が大きくなりうる

=> 用法・用量を守って使っていこう 

## 使い所

- メール送信
- 外部からWebhookを受信する場合レスポンスはすべて200で返して、内部処理をタスクキューに流す
  - アプリケーションロジックのバグで、Webhook受信が失敗しないように
- 同期性が必要なく、ユーザーにはやくレスポンスを返したい
  - レスポンスが返ってくるまではローディングを表示するけど、長すぎるとみんなブラウザ再読み込みしたりしちゃうよね



## 他の選択肢

- RQ
  - 「Simple job queues for Python」 と謳っているとおりシンプルで扱いやすい
  - 対応しているBrokerはRedisのみ
- FaaS
  - マイクロサービスとしてアプリケーションを分離できるなら、 AWS Lambda x SQSのような構成も

## Celeryの対応Broker

Name	Status	Monitoring	Remote Control
RabbitMQ	Stable	Yes	Yes
Redis	Stable	Yes	Yes
Amazon SQS	Stable	No	No
Zookeeper	Experimental	No	No

参照: [Backends and Brokers — Celery 5.1.2 documentation](#)

# Celeryの基本Tips

## Taskの処理は短く

- 理解しやすいように、責務をシンプルに
  - アーキテクチャが複雑になっているので、処理自体は簡潔に
- リトライしやすいように、ステートレスでアトミックな処理
  - リトライについては、あとで詳しく話します 🙏

## プリミティブな値を渡す

- Taskの引数には、複雑なオブジェクトを渡さずに、intやstrなどのプリミティブな値を渡す
- とくにDjangoのモデルインスタンスは渡しがちなので注意 ⚠
  - シリアライザーのデフォルト設定 json のままだと、`TypeError` 発生
    - シリアライザーにpickleを設定すれば、渡すことはできる
  - 非同期なのでCelery Workerが処理を開始する前にこのレコードが更新されてしまうと、古い状態のままのデータを使ってしまうことに

参照: [95:Celeryのタスクにはプリミティブなデータを渡そう — 自走プログラマー【抜粋版】](#)

## シリアライズに失敗


```
>>> user = User.objects.get(pk=user_id)
>>> sample_task.delay(user=user)
Traceback (most recent call last):
...
TypeError: Object of type 'User' is not JSON serializable
```

id を渡して、Taskのなかで最新のデータを取得するように

```
@shared_task
def smaple_task(user_id):
    if not user_id:
        return

    user = User.objects.get(pk=user_id)
    ...
    user.save()
```

## テストでは `always_eager` を `True` に

- `CELERY_ALWAYS_EAGER = True` を設定すると、Taskは同期的に実行されるので、Celery Workerを起動しなくても 
- テストやローカル開発時には、この設定にしておく と 捗る

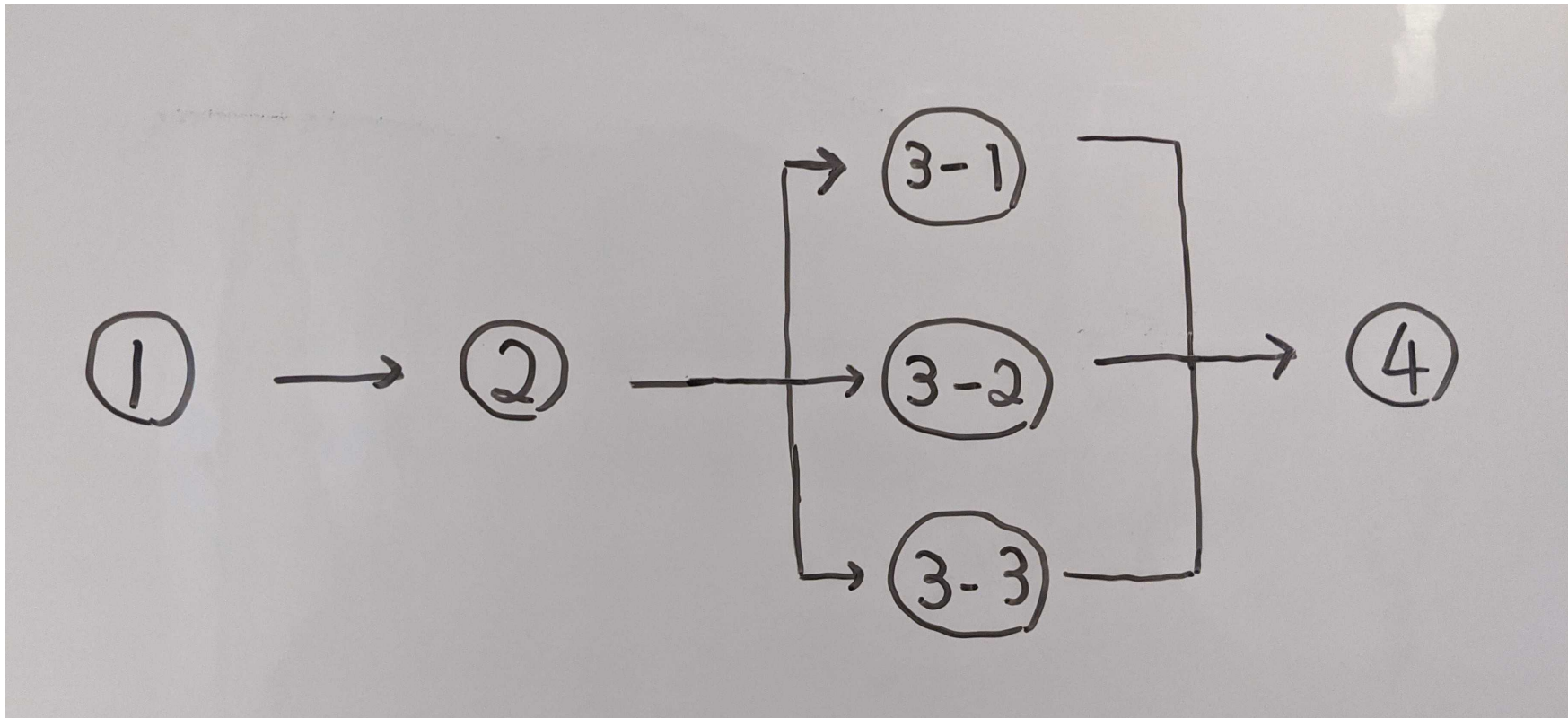
# ワークフローデザイン



## ワークフロー

- Celeryは非同期実行だけでなく、ワークフローを組むこともできる
- 直列・並列・展開を組み合わせて、処理をかけあわしていく

### 簡単なイメージ



## Chain

```
>>> from celery import chain
>>> from proj.tasks import add, mul

>>> # (4 + 4) * 8 * 10
>>> res = chain(add.s(4, 4), mul.s(8), mul.s(10))()
>>> res.get()
640
```

- タスクを直列に（順番に）実行する
- 前のタスクの実行結果が、後ろのタスクの第1引数に渡される

## Signature

- `add.s(4, 4)` は、`add.signature(4, 4)` のショートカット記法
- `signature`を利用するとタスクに渡された引数や実行オプションをラップすることができ、他の関数に渡すことができる

## Immutable signature

- 前のタスクの実行結果が必要ない場合は、immutableなsignatureにできる
- `add.signature((2, 2), immutable=True)` で、ショートカット記法は `add.si(2, 2)`

Immutable signatureを使うと、それぞれ独立したTaskを指定した順番で実行できるようになる 💡

```
>>> res = (add.si(2, 2) | add.si(4, 4) | add.si(8, 8))()  
>>> res.get()  
16
```

Chainの他にも

- 複数のTaskを並列に実行するGroup
- 複数のTaskの実行結果をコールバックに渡すことができるChord

などがある。

参照: [Canvas: Designing Work-flows — Celery 5.1.2 documentation](#)

# リトライ設計

## そもそもの例外設計

- 復帰可能なエラーのみキャッチする
- 復帰可能なエラー以外は、握りつぶさず投げっぱなしにする
  - 最上位層で復帰できない例外として処理する
  - そこでSentryなどの Error Tracking Software に飛ばす
- 運用でカバーする必要があるものはDBに保存しておく

参照: <https://gist.github.com/sunaot/6138546#例外設計の話>

## リトライ設計

- 復帰可能なエラーはリトライし続ける
  - 設定した最大リトライ回数まで
- 処理はステートレスで設計する
- 処理の冪等性を担保しておく
  - 基本的には `transaction.atomic` で
- 1回の大きな処理よりも複数の小さな処理に分割しておく

## リトライ設計

- 障害の回避が目的ではなく、ダウンタイムやデータ損失を回避すべく障害に対応していく
- 復旧が遅くなればなるほど、確認事項が増えて自動化が困難
- 「壊れない」から、「素早くいつでも回復できる」設計へ


参照: [15分で分かる NoOps](#)



## リトライ関連の設定

- autoretry\_for
  - 自動リトライ対象のExceptionクラスを指定
- max\_retries
  - 最大リトライ回数
- retry\_backoff
  - Exponential backoff を有効に
  - リトライ間隔を指数関数的に増加（例: 1秒後、2秒後、4秒後、8秒後、、、
- retry\_backoff\_max
  - retry\_backoff 有効時の最大リトライ間隔
- retry\_jitter
  - retry\_backoff 有効時に、jitter（ゆらぎ）を導入
  - リトライ実行タイミングの衝突を回避するため

## リトライ関連の設定

- `time_limit`
  - ハードタイムリミット
  - このタイムリミットを超えたら、Taskを実行しているWorkerがkillされて、新しいWorkerに置き換わる 
- `soft_time_limit`
  - ソフトタイムリミット
  - このタイムリミットを超えたら、`SoftTimeLimitExceeded` がraiseされる
  - ハードタイムリミットになる前に、アプリケーション側で制御が可能

復帰可能のハンドリングが簡単であれば、 `autoretry_for` の設定でOK

```
@celery_app.task(
    bind=True,
    soft_time_limit=60,
    time_limit=120,
    autoretry_for=(RetryableException,),
    retry_kwargs={'max_retries': 10},
)
def example_task(
    self,
    user_id=None,
    **kwargs,
):
    ...
```

復帰可能なエラーのパターンが増えてきたら、リトライ用のdecoratorを自作して育てていくのがよき

```
@celery_app.task(
    bind=True,
    soft_time_limit=60,
    time_limit=120,
    retry_kwargs={'max_retries': 10},
)
@retry_when_retryable(logger=logger)
def example_task(
    self,
    user_id=None,
    **kwargs,
):
    ...
```

```

def retry_when_retryable(logger=None): # noqa C901 That's too complex :(
    if not logger:
        logger = _logger

    def _retry_when_retryable(task):
        @functools.wraps(task)
        def wrapper(app, *args, **kwargs):
            try:
                return task(app, *args, **kwargs)
            except DjangoDBOperationalError as exc:
                # Retryable mysql errors
                if exc.args[0] in [
                    1040, # Too many connections
                    2003, # Can't connect to MySQL server
                    2013, # Lost connection to MySQL server during query
                ]:
                    logger.warning('Database operation occurred: %s', exc)
                    raise app.retry(countdown=60 * 5, exc=exc)
                raise exc
            except (
                DjangoDBInternalError,
                DjangoDBIntegrityError,
            ) as exc:
                # Retryable mysql errors
                if exc.args[0] in [
                    1062, # Duplicate entry (when get_or_create)
                    1205, # Lock wait timeout exceeded; try restarting transaction
                    1206, # The total number of locks exceeds the lock table size
                    1213, # Deadlock found when trying to get lock; try restarting transaction
                    1689, # Wait on a lock was aborted due to a pending exclusive lock
                ]:
                    logger.warning('Database internal occurred: %s', exc)
                    raise app.retry(countdown=25, exc=exc)
                raise exc
            except CelerySoftTimeLimitExceeded as exc:
                logger.warning('Time limit occurred: %s', exc)
                raise app.retry(countdown=60 * 5, exc=exc)
            except RetryableException as exc:
                logger.warning('Retryable error occurred: %s', exc)
                raise app.retry(countdown=exc.countdown, exc=exc)

        return wrapper

    return _retry_when_retryable

```

<https://gist.github.com/massa142/d9256496469c8e95f526d2132fab9426>

ログ保存

## django-celery-results

- Taskの実行結果をDBに保存してくれる
  - キャッシュバックエンドに保存するように設定も可能
- なにかあったときに、あとから運用でカバーできるように

## テーブル構造

```
mysql> desc django_celery_results_taskresult;
```

Field	Type	Null	Key	Default	Extra
id	int(11)	NO	PRI	NULL	auto_increment
task_id	varchar(255)	NO	UNI	NULL	
status	varchar(50)	NO	MUL	NULL	
content_type	varchar(128)	NO		NULL	
content_encoding	varchar(64)	NO		NULL	
result	longtext	YES		NULL	
date_done	datetime(6)	NO	MUL	NULL	
traceback	longtext	YES		NULL	
meta	longtext	YES		NULL	
task_args	longtext	YES		NULL	
task_kwargs	longtext	YES		NULL	
task_name	varchar(255)	YES	MUL	NULL	
worker	varchar(100)	YES	MUL	NULL	
date_created	datetime(6)	NO	MUL	NULL	

```
14 rows in set (0.26 sec)
```

```
mysql>
```



# 運用でカバー

## 具体例

HTTPリクエスト先が500エラーを返しきて、Celery Taskが異常終了していた

## アプローチ

Dango adminから異常終了した(status=FAILED)のTaskを期間指定で絞りこんで、まとめてリトライするようなaction用意しておくとか渉る

- HTTPリクエスト先が復旧した後で、このactionを実行する
- task\_name, task\_args, task\_kwargsが保存されているので、そこから再実行可能

## task protocol

- Celery 4.0 から、task protocolは version 2 がデフォルト
  - [Message Protocol — Celery 5.1.2 documentation](#)
- 普通にTaskを実行する分には問題ないけど、django-celery-resultsに実行結果を保存する場合は要注意
  - task protocol 2だと、`celery.utils.saferepr.saferepr` をかました文字列が `task_args` , `task_kwargs` に保存される
  - `celery.utils.saferepr.saferepr` は、`maxlevels=3` が固定になっていてネストされたデータ構造はmaskされちゃう
    - `"{'a': 1, 'b': {'c': {'d': [...]}}}"`
  - いまのところ `CELERY_TASK_PROTOCOL = 1` に設定すれば解決できる

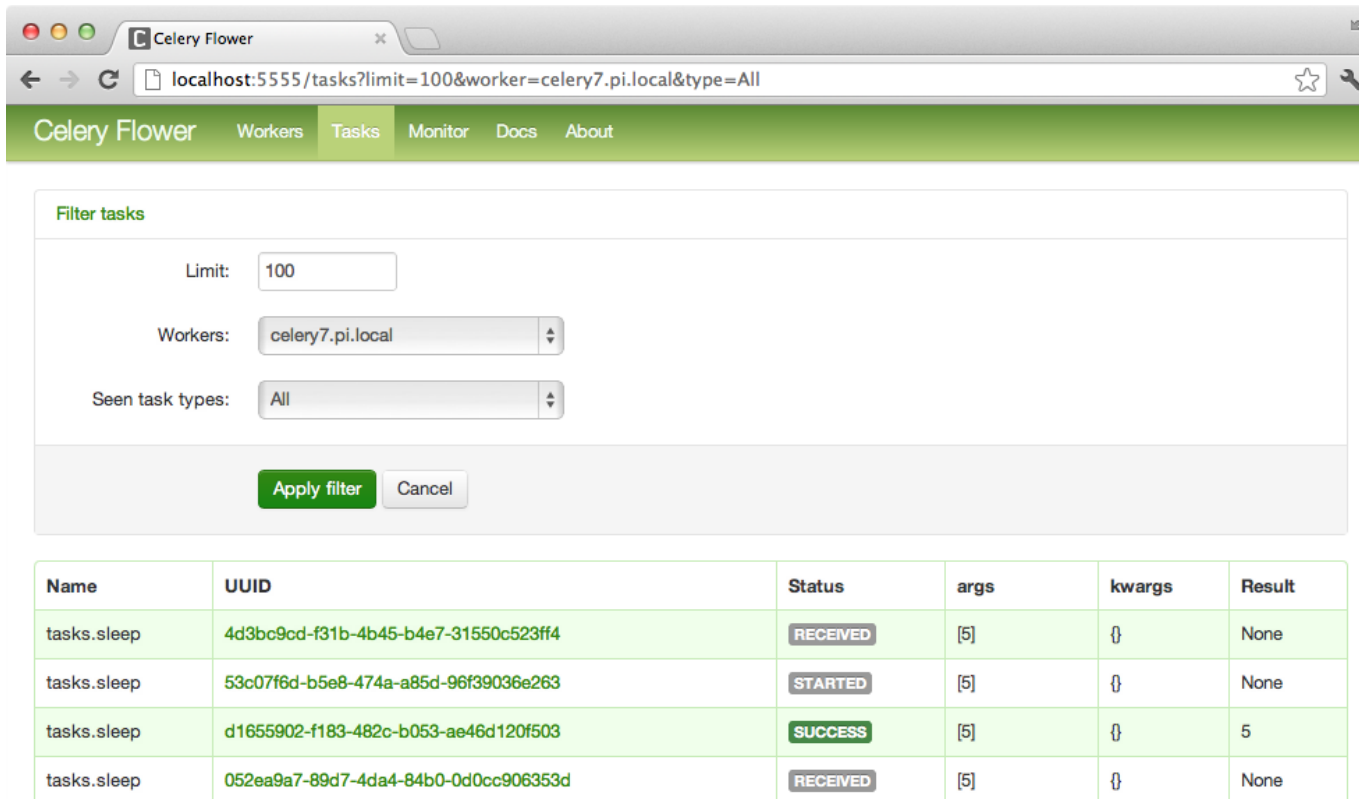
監視

- django-celery-resultsを使えば実行ログを保存できるので、Celery Workerが受け取ったTask状況は確認できるけど、、、
- Queueがどれくらい詰まっているかなどはわからない

=> 監視が必要に 🙄🙄

# Flower

- Task状況が可視化されて便利 🌸
- ローカル開発のときにめっちゃありがたい
- `$ flower -A proj --port=5555`



The screenshot shows the Celery Flower web interface in a browser window. The address bar shows `localhost:5555/tasks?limit=100&worker=celery7.pi.local&type=All`. The interface has a green header with navigation links: Celery Flower, Workers, Tasks (selected), Monitor, Docs, and About. Below the header is a 'Filter tasks' section with three input fields: 'Limit' set to 100, 'Workers' set to 'celery7.pi.local', and 'Seen task types' set to 'All'. There are 'Apply filter' and 'Cancel' buttons. Below the filter section is a table with 6 columns: Name, UUID, Status, args, kwargs, and Result. The table contains 4 rows of task data.

Name	UUID	Status	args	kwargs	Result
tasks.sleep	4d3bc9cd-f31b-4b45-b4e7-31550c523ff4	RECEIVED	[5]	{}	None
tasks.sleep	53c07f6d-b5e8-474a-a85d-96f39036e263	STARTED	[5]	{}	None
tasks.sleep	d1655902-f183-482c-b053-ae46d120f503	SUCCESS	[5]	{}	5
tasks.sleep	052ea9a7-89d7-4da4-84b0-0d0cc906353d	RECEIVED	[5]	{}	None

本番環境だとDatadogなどにログを投げて監視するのはもちろんあるけど、`celery inspect` コマンドを使った簡易的な監視Scriptも有益

- `$ celery -A proj inspect reserved`
  - Workerが取得したけど、実行待ちになっているTaskを列挙
- これを使って、ずっと実行待ちになっているTaskがないかをチェック

```
$ celery -A proj inspect reserved --json | check_celery_reserved.py 300
/tmp/celery_reserved_jobs
```

```
def check():
    recs = json.load(sys.stdin)

    prev = {}
    cur = {}

    save = pathlib.Path(args.save)
    if save.exists():
        try:
            prev = json.loads(save.read_text())
        except Exception as e:
            print(e, file=sys.stderr)

    for worker, jobs in recs.items():
        for job in jobs:
            jobid = job.get('id', '')
            if not jobid:
                continue
            cur[jobid] = prev.get(jobid, now)

    try:
        save.write_text(json.dumps(cur))
    except Exception as e:
        print(e, file=sys.stderr)

    for job, t in cur.items():
        d = now - t
        if d >= args.secs:
            # Slack通知

if __name__ == '__main__':
    check()
```

# デプロイ戦略



## 既存Taskを拡張する

Taskは引数変更したものをデプロイした後も、しばらく旧引数で呼び出される可能性あり。  
いつでも引数の増減が可能なように最初から

- 引数はすべてキーワード引数渡しで
- `*kwargs` を追加しておく

と、デプロイタイミングに慎重になることもなく運用がしやすい 😊

## 变更前


```
@app.task
def change_kwargs_task(current=None, **kwargs): # old
    ...
```

## 变更后

```
@app.task
def change_kwargs_task(current=None, new=None, **kwargs): # New
    new = new or some_calc(old)
    if not new:
        return True
    ...
```

## 既存Taskを削除する

さっきのと同様にWorker側から削除したあとでも、デプロイタイミングによってはそのTaskが呼び出される可能性あり。

まずは削除せずdeprecatedにして、そのTaskが呼ばれなくなったことを確認できてから別リリースで削除すれば 

おわりに

- 今日話したことはだいたいCeleryの公式ドキュメントに載ってます 📖
  - Djangoほど日本語の情報・知見が充実していないかなと思うので、この発表がどこかで役に立てばいいな
- Celeryを使うことで、ユーザーへのレスポンスがはやい & なにかエラーになってもリトライ可能な復元力が高い サービスにしていきましょう 100

ご静聴ありがとうございました 😊

時間があまったら

# 重い処理するときの注意点

## 時間がかかる場合

- 時間がかかる処理専用のQueueに分ける
  - そのQueueを購読する専用のWorkerを立ちあげる
- `CELERYD_PREFETCH_MULTIPLIER` を 1 に
  - WorkerがBrokerから一度に取得するTask数
  - 通信コストを抑えるためにデフォルト値は 4
  - WorkerがTaskを取得したあとに詰まらないようにするために 1 をセット

## メモリリークケア

- `CELERY_WORKER_MAX_TASKS_PER_CHILD` を 1 に
  - Workerが同じプロセスで実行する最大Task数
  - 1にセットしたら、Taskごとに新しいプロセスが生成される