

今回は Django マイグレーションのまとめ記事を書くと言ってしまったので泣きながら書きました ☹️

Djangoマイグレーションのまとめ記事を書かないといけないのかもしれない

— ころ (@crohaco) [October 31, 2018](#)

いつものように応用事例まで書いていたら スクロールバーが米粒くらいの大きさになってしまったので 記事を分割します。

え、今もそうだって？ ノリノリ、まさか...

マイグレーションってよく「あー、もうめちゃくちゃだよ」てなるので 苦手意識を持っている方も多いと思いますが この記事を読めば、きっとそれも解消できると信じています ☺️

備考

- 応用事例は [ケーススタディ編](#) に書きました
 - よくわからない方は 先にこの記事を読むことをおすすめします
- 次の実行環境で検証しています
 - Python 3.7
 - Django 2.1
- 今回の記事を手元で試したい方は [apps.zip](#) をダウンロードしてください。
 - requirements.txt に従って依存ライブラリをインストール すればOKです。(venv推奨)
 - DB は sqlite3 を使うため特殊なミドルウェアのインストールは不要です。
 - Mac や Linux では動くと思いますが、Windows で動くかどうかは不明です

▶ 目次

マイグレーションについて

マイグレーションは Django 経由で データベースに変更を加えるための仕組みです。

Django 1.6 までは [django-south](#) というサードパーティライブラリがデファクトスタンダードだったんですが、1.7 からは Django から 公式に提供されるようになりました。

備考

今回お話するのは South ではなく Django そのものの マイグレーションです。

Southから 移行する場合はこちらを参考にしてください。

[Upgrading from South](#)

構造

まず最初に、Djangoマイグレーションの構造を理解しましょう。

Migration file (マイグレーションファイル)

マイグレーションファイルとは 実行されるマイグレーションの内容が記述された Python モジュールで、マイグレーション(migrate)の最小実行単位です。

この記事でマイグレーションという場合、このマイグレーションファイルの単位を指します。

通常は Model の状態をもとに自動生成しますが、意図したようなマイグレーションが作られないようなときは手動で作成することもできます。

いずれの場合であっても、必ずアプリごとの `migrations/` ディレクトリ配下で管理されます。

あなたが定義したアプリ以外に Django 自身が持つアプリのマイグレーションもあります。すべて把握する必要はありませんが、一応そういうものもあると認識だけはしておいてください。

警告

`migrations/` ディレクトリ配下に置かれているモジュール(`.py` 拡張子)はすべてマイグレーションファイルと判断されるため、関係ないモジュールがあるとエラーになります。

```
django.db.migrations.exceptions.BadMigrationError: Migration aaaa in app sales has no Migration class
```

モジュール分割を行う場合、配置するディレクトリには気をつけてください。

Migration class

マイグレーションファイルには `django.db.migrations.Migration` を継承した `Migration` クラスが必要です。

クラスは以下の属性を持ちます。

`initial`

アプリ内の最初のマイグレーションに指定するフラグ値です。

(現状では) `migrate` コマンドの `--fake-initial` オプションのためだけに存在します。オプションの詳細は後述します。

省略時の挙動は

- 先頭のマイグレーション(同一アプリ内の他のマイグレーションへの依存がない)では `True`
- それ以外のマイグレーションでは `False`
 - 先頭以外のマイグレーションで `initial = True` を手動で指定した場合も、有効に働きます。

`replaces`

自身のマイグレーションが置き換える対象のマイグレーションを `(アプリ名, マイグレーション名)` という形式のタブルのリストで指定します。

詳しくは `squashmigrations` セクションまで読み進めてください。

`dependencies`

依存するマイグレーションを (アプリ名, マイグレーション名) という形式のタプルのリストで指定します。

詳しくは 依存関係 セクションまで読み進めてください。

atomic

True の場合、マイグレーション 単位でトランザクションが張られ、マイグレーションの途中で失敗するとロールバックします。

False の場合、ロールバックしないので失敗する前に実行した操作はそのままDBに残ります。

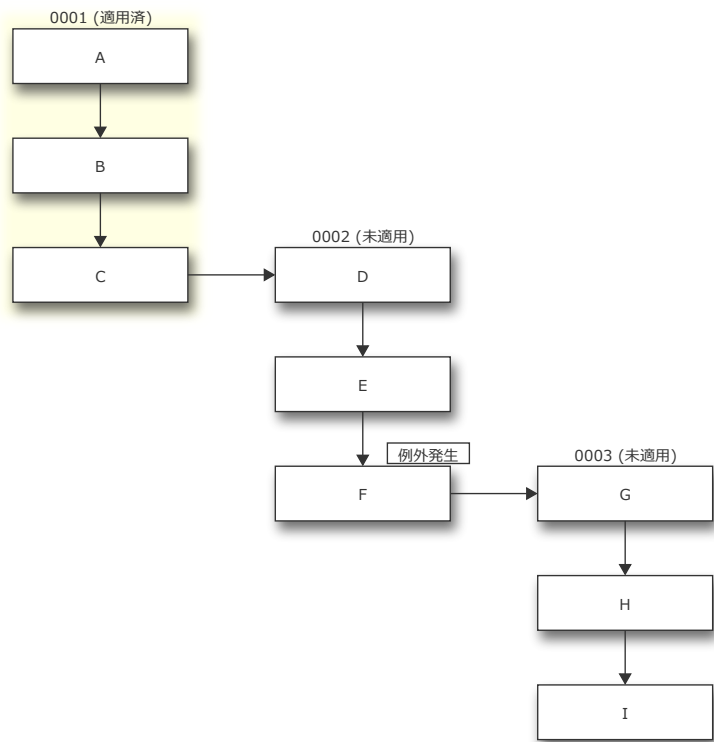
デフォルト値は True です。

このオプションは 常に有効であることが望ましいですが、データがメモリに乗り切らないようなケースでは無効にすることがあるようです。

備考

当該オプションが有効であれ無効であれ、最後まで処理が成功したマイグレーションだけが適用済になります。

例えば 0001, 0002, 0003 というマイグレーションが順に適用され、0002 マイグレーション内の最後の 操作(Operation) で失敗した場合、0001 のみが 適用済としてマークされるという具合です。



<https://docs.djangoproject.com/en/2.1/howto/writing-migrations/#non-atomic-migrations>

operations

マイグレーションの操作内容です。

指定できる操作については後述しますが、例を挙げると以下のように Operation をリスト形式で指定します

```
# 前略
# :
operations = [
    migrations.CreateModel(
        name='Price',
        fields=[
            ('id', models.AutoField(auto_created=True, primary_key=True, serialize=False, verbose_name='ID')),
```

```
        ('price', models.IntegerField()),
        ('effective_date_from', models.DateTimeField(null=True)),
        ('effective_date_to', models.DateTimeField(null=True)),
    ],
),
migrations.AddField(
    model_name='price',
    name='product',
    field=models.ForeignKey(on_delete=django.db.models.deletion.CASCADE, to='products.Product'),
),
]
```

マイグレーションの自動生成で設定される Operation クラスで主要なものは以下です。(ほかにもあります)

CreateModel	DeleteModel
モデルの追加	
モデルの削除	RenameModel
モデルの改名	AddField
フィールドの追加	RemoveField
フィールドの削除	RenameField
フィールドの改名	AlterField
フィールドの属性変更	

組み合わせて任意のマイグレーションを作成することもできます。

しかし、スキーマではなくレコードを追加、変更、削除したいケースもあるでしょう。これをスキーママイグレーションと対比して、データマイグレーションといいます。

データマイグレーションを行う場合、以下の Operation を使ってDBに対して DML を実行するようなマイグレーションを作成します。

RunPython	RunSQL
任意 Pythonスクリプトの実行 (詳しくは ケーススタディ編#runpython)	
任意SQLの実行 (詳しくは ケーススタディ編#runsql)	

マイグレーション適用履歴

マイグレーションの適用履歴は DB の `django_migrations` テーブルで管理されています。

スキーマはこんな感じです。

id	app
ただのサロゲートキー (integer)	
マイグレーションが属するアプリ名	name
マイグレーション名. 例) 0001_initial	applied
適用日時。多分人間が見る用で、この値を元にシステムが何かをするということ はなさそう。	

このテーブルに履歴がないマイグレーションは未適用と判断されます。

つまり、適用するかどうかはこのテーブルを見て判断するというわけです。

依存関係

DBを適切な状態に保つため、マイグレーションファイルを適用する順番には特に気をつけないといけません。

各マイグレーションファイルには4桁の連番が付与されていますが、これはただの数字で、(大抵は一致しますが) 実は適用順とは関係ありません。

マイグレーションごとに依存関係があり、これを辿ることで適用順が決定します。具体的には、依存先のマイグレーションが適用されてから自身を適用するという具合です。

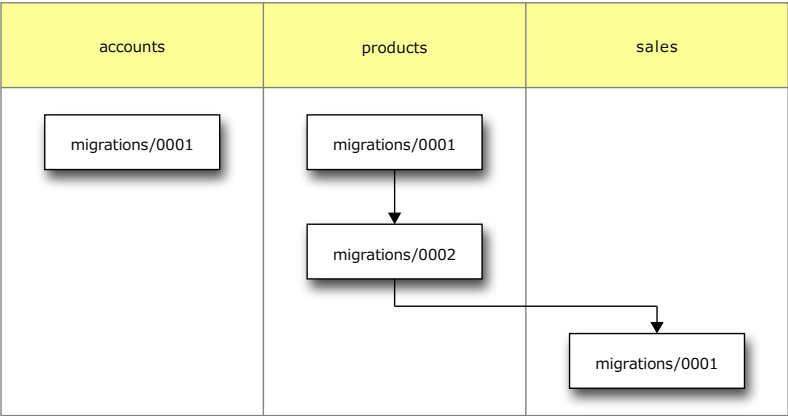
依存 には 以下の 2 パターンがあります。

- アプリごとの依存
 - 同じアプリの、1つ前のマイグレーションを指す
 - 先頭のマイグレーション (0001_initial.py) の場合は なし
 - 同じアプリ内に同じ依存先を持つマイグレーションの存在は望ましくない (適用順が保証されないため)
- 外部アプリへの依存
 - 外部キーがある場合、他のアプリの最後のマイグレーションを指す
 - 最大で依存している外部アプリの数だけできる

例えば下記のような手順を踏むと

1	2
accounts アプリを makemigrations	
products アプリを makemigrations	3
products アプリのモデルを修正して再度 makemigrations	4
(Productを参照している) sales アプリを makemigrations	

次のような実行順のマイグレーションが出来上がります。



sales は products に依存していますが、accounts と products は関係がないので、実行順は保証されません。

上述したように、マイグレーションは依存関係を辿るので一つでも欠けるとうまく動作しません。

```
django.db.migrations.exceptions.NodeNotFoundError: Migration accounts.0004_dummy dependencies reference nonexistent
```

そのため、「今回だけはマイグレーションを使う」とか気まぐれな実行はできません。最初から最後まで「使うか」「使わないか」の2択です。

とはいえ、Djangoのマイグレーションで対応できないケースというのはほとんどないので積極的に使っていきましょう ☺ (この記事を見ている時点で使っている人が大半だとは思いますが..)

備考

マイグレーション番号は実行順を決める要素ではありませんが、 当記事では わかりやすさを重視するため、暗黙的に数字が小さい方が先に実行されるものとして取り扱います。

適用方向

マイグレーションの適用方法については後述しますが、その適用順は2種類あります。

通常の実行では 昇順 (ascendant) に適用されます。簡単に言うと番号の小さい順です。

この次に説明するものと区別するため ここでは 順適用 (forward applying) といいます。(正式な用語ではありません)

そして逆向きの 降順 (descendant) の適用です。適用したマイグレーションを取り消す目的で使用されます。

ここでは 逆適用 (backward applying) といいます。(正式な用語ではありません)

逆適用するには operations に指定された すべての Operation に 逆の操作 (backwards) が定義されている必要があります。

自動的に作成されるような単純な Operation は この逆操作が定義されていますが、 上述した RunSQL や RunPython のように自由度の高い操作の場合、 逆操作は私達が指定してあげないと未定義となります。

未定義だと逆適用は不可能です。エラーになります。

警告

とはいえ、逆適用 はトラブル用の応急処置であり、多用するべきものではありません。

git revert や hg backout のようなバージョンコントロールシステム における取り消しでは 逆差分を適用して「打ち消し」を表現していたのでもとに戻すことができますが、 Django migration における 取り消し は 順適用とは逆の操作を実行しているに過ぎません。

例えば **CreateModel** (create table) の逆操作は **DeleteModel** (drop table) であり、実行されればテーブルとともに 中のデータも消え去ります。

該当ソース

危険な操作であることを理解した上で使うようにしましょう。

実際の取り消し方については **migrate** セクションまで読み進めてください。

使い方

続いてマイグレーション関連コマンドの使い方を見ていきましょう。

備考

このセクションの用語について

- マイグレーション名はマイグレーションファイル名の前方から一意に特定できる部分までで構いません。大抵の場合 **0001** のような 数字部分 だけでよいです。
 - 後述する `squashmigrations` をするとこの番号が重複するので、古い方を消すなりして対応します。
- 位置引数** とは 先頭(左)から順にコマンドに指定する引数です。指定する位置によって異なる用途に使われることが多いです。
- オプション** とは コマンドに指定する 引数のうち **--name** や **-n** のようにハイフンから始まる引数です。これらは位置引数と違い、どのような順番で指定しても構いません。
 - オプションはコマンド毎にたくさんありますが、全部説明するのは正直しんどいので 重要なものに絞って説明します。

makemigrations

マイグレーションファイルを作成するコマンドです。

自動生成する場合、既存マイグレーションとモデルの定義の差分によって 新たなマイグレーションが作られます。

```
$ ./manage.py makemigrations --help
usage: manage.py makemigrations [-h] [--dry-run] [--merge] [--empty]
                                [--noinput] [-n NAME] [--check] [--version]
                                [-v {0,1,2,3}] [--settings SETTINGS]
                                [--pythonpath PYTHONPATH] [--traceback]
                                [--no-color]
                                [app_label [app_label ...]]

Creates new migration(s) for apps.

positional arguments:
  app_label              Specify the app label(s) to create migrations for.
```

- 位置引数は **app_label** (アプリ名) だけです。
- 可変長引数なので複数指定可能で、省略もできます。
 - アプリ名を省略した場合、`INSTALLED_APPS` に登録されているアプリのうち 差分があるものに関してマイグレーションを作成します
 - 最初のマイグレーションファイルが作られていない状態では、マイグレーションは作られないようです。つまり、アプリを新規作成した場合は `app_label` の指定が必須というわけです。

続いて オプションについて説明します。

`--name`

マイグレーションファイルの名前を指定します。

通常、マイグレーションファイル名は変更対象のモデルやフィールドなどから自動生成されますが、 マイグレーションの内容をわかりやすくするためにファイルを名を変えたいという場合に利用します。

作成されるマイグレーションファイルは `{連番}_{name}.py` のような名称になります。

作成後のファイルを普通にリネームしても問題ないので特に覚えてなくても困りません。(ただし、適用済のマイグレーションファイル名を変えるのはだめですよ)

`--merge`

重複したマイグレーションファイルをマージします。

Django は 同じ依存先を先端に持つマイグレーションの存在を許しません。 これを束ねるためのマイグレーションを作成するオプションです。

実例は [ケーススタディ編#migrations-having-the-same-parent](#) を参照ください。

`--empty`

空のマイグレーションファイルを作成できます。

手動でマイグレーションファイルを作成する場合に必要なので覚えてください。

自動生成する差分がない場合は `No changes detected` となります。

<https://docs.djangoproject.com/el/2.1/ref/django-admin/#makemigrations>

実行例

```
$ ./manage.py makemigrations products
Migrations for 'products':
  products/migrations/0001_initial.py
    - Create model Category
    - Create model Price
    - Create model Product
    - Add field product to price
```

migrate

マイグレーションファイルを適用するコマンドです。

```
$ ./manage.py migrate --help
usage: manage.py migrate [-h] [--noinput] [--database DATABASE] [--fake]
                        [--fake-initial] [--run-syncdb] [--version]
                        [-v {0,1,2,3}] [--settings SETTINGS]
                        [--pythonpath PYTHONPATH] [--traceback] [--no-color]
                        [app_label] [migration_name]

Updates database schema. Manages both apps with migrations and those without.

positional arguments:
  app_label              app_label of an application to synchronize the state
  migration_name          the specific migration to apply
```



```
app_label      App label of an application to synchronize the state.
migration_name Database state will be brought to the state after that
               migration. Use the name "zero" to unapply all
               migrations.
```

位置引数は **app_label** (アプリ名) と **migration_name** (マイグレーション名) となっていますが、いずれも省略可能です。

- 省略した場合は 未適用のマイグレーションをすべて順適用します。
- アプリだけを指定した場合、アプリ内に含まれる未適用のマイグレーションを順適用します。
- マイグレーション名まで指定した場合は 指定したアプリ内のマイグレーションのうち、適用済マイグレーションから指定したマイグレーションまでを適用します。
 - 適用順は 指定したマイグレーション名によって異なります。
 - 最後に適用したマイグレーションより **後** の場合は **昇順** に順適用
 - 最後に適用したマイグレーションより **前** の場合は **降順** に逆適用
 - **zero** を指定すると **先頭より前** のマイグレーションを指すため、すべての適用済マイグレーションが逆適用される
 - 最後に適用したマイグレーションと **同じ** の場合は 何も起こらない (No migrations to apply.)
 - マイグレーションは アプリ名を含めて一意なため、マイグレーション名だけを指定することはできません (そもそも位置引数だからマイグレーションだけ指定するなんて無理な話ですが)

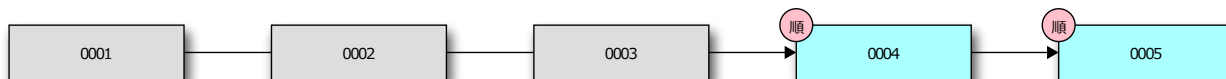
言葉だけではすこしわかりにくいので例を用意します。

accounts アプリに 0001 ~ 0005 までの 計5つのマイグレーションがあり、現在 0003 までが適用されている状態だとします。

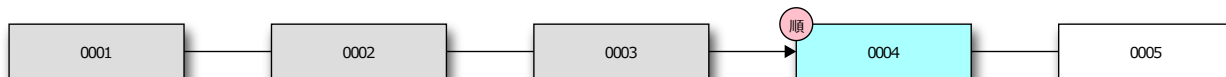
この状態で マイグレーションを適用したらどうなるのかを表したのが以下の図です。

順適用が 薄い青, 逆適用が 薄い黄色, 適用済が グレー, 未適用が 白 です

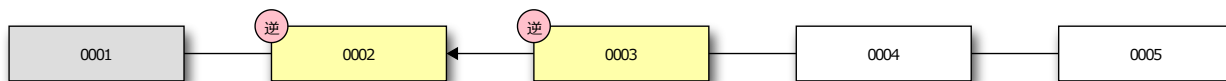
```
./manage.py migrate accounts
```



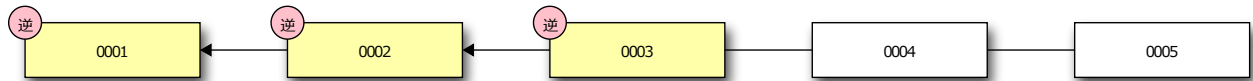
```
./manage.py migrate accounts 0004
```



```
./manage.py migrate accounts 0001
```



```
./manage.py migrate accounts zero
```



イメージとしては先頭のマイグレーションを 左 とした場合、適用対象となるマイグレーションは常に **左開右閉** となります。(始端のマイグレーションは含まれないという意味です)

備考

逆適用対象のマイグレーション に 依存しているマイグレーションも逆適用されます。

つまり `products/migrations/0003` まで適用していて、これに依存している `sales/migrations/0001` がある状態で 逆適用すると以下のような順番になります。

- `sales/migrations/0001`
- `products/migrations/0003`
- `products/migrations/0002`
- `products/migrations/0001`

続いてオプションです。

`--fake`

マイグレーションをDBには実際に適用せず、適用済ということにするためのオプションです。

DBとマイグレーションの間に不整合が生じてマイグレーションが当てられないときに使用します。

とはいえ、現状のDBとマイグレーションの定義に差異がある場合は以降ズレ続けてトラブルの原因となる可能性があるので、手動でスキーマを調整してあげましょう。

`--fake-initial`

いわば移行用の オプションです。

テーブルを新規で作成する 操作(CreateModel) では、同名のテーブルがすでに存在するとエラーになります。

Django 1.7 以降に新規で作成されたテーブルでない限り、「マイグレーションは未適用状態」だが、「テーブルは存在する」という不整合状態となります。

このような場合に、`--fake-initial` を使うと、CreateModel で指定したテーブルが既に存在、かつ `initial = True` が指定されている場合、そのマイグレーション は fake 適用されます。

各アプリの `0001` マイグレーションだけをポチポチ指定して `--fake` すればいいんですが、それはめんどくさいですし、適用漏れが発生する恐れもあります。

逆に `--fake` を全部に適用してしまうと、本来あるべきテーブルが作られない自体も起こりえます。

`--fake-initial` を使うと 初期テーブルの新規作成 Operation が含まれるであろう 先頭のマイグレーションだけを フェイク適用できるんですね。

警告

`--fake-initial` オプションを過信してはいけません。

もし、一つのマイグレーション中に **存在する** テーブルと **存在しない** テーブルが混ざっている状態で `--fake-initial` をすると そのマイグレーションはフェイク適用され、存在しないテーブルは作られないままになってしまいます。

これは `--fake-initial` のリスクと言えるでしょう。

また、フェイク適用されるべきマイグレーションを、知らずに 0002 以降のマイグレーション (`initial = False`) として作ってしまうと逆にフェイク適用されず、適用時にエラーとなります。

適用対象の DB の状態が予めわかっている場合は、存在 **する/しない** でマイグレーションファイルを分けてしまうのが良いと思います。

備考

両者のフェイク適用の条件のみを比較すると以下ようになります

- `--fake`: なし (問答無用でフェイク適用)
- `--fake-initial`: 以下を両方とも満たしたときにフェイク適用
 - `initial = True` が指定されている
 - マイグレーションファイル内の `CreateModel` で作成予定のテーブルがすでに存在する

あくまで私の見解としての使い分けを書きます。

- `--fake`: フェイク対象のマイグレーションが明確にきまっている
 - 例1) 特定のマイグレーションのみを 適用済 にしたい
 - 例2) 現在のモデル定義と DB の状態は 完璧に一致しているが、履歴 (django_migrations) に反映されていないので 適用済 にしておきたい
- `--fake-initial`: 適用対象に フェイクしたいマイグレーションとそうでないものが混在している
 - 例) **0001** 「テーブルの追加用のマイグレーション」、**0002** 「列追加用のマイグレーション」があって
 - 同様のテーブル自体は存在するので 0001 はフェイク適用されてほしい
 - 列追加自体は要件として必須なので 0002 は普通に適用されてほしい

`--fake-initial` のほうが実行条件が狭いので `--fake` よりは安全と言えますが、リスクがないわけではないので 適用して大丈夫な状態かは予め調査すべきです。

同一DB のクローンを建てて、それに対してマイグレーションを適用可否を確認するのがおすすめです。(こういう場合にクラウドだと便利ですね)

<https://docs.djangoproject.com/el/2.1/ref/django-admin/#migrate>

実行例

アプリ名とマイグレーションを指定せずに実行してみます。

```
$ ./manage.py migrate
Operations to perform:
  Apply all migrations: accounts, admin, auth, contenttypes, products, sales, sessions
Running migrations:
  Applying accounts.0001_initial... OK
  Applying contenttypes.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying admin.0002_logentry_remove_auto_add... OK
```

```
Applying admin.0003_logentry_add_action_flag_choices... OK
Applying contenttypes.0002_remove_content_type_name... OK
Applying auth.0001_initial... OK
Applying auth.0002_alter_permission_name_max_length... OK
Applying auth.0003_alter_user_email_max_length... OK
Applying auth.0004_alter_user_username_opts... OK
Applying auth.0005_alter_user_last_login_null... OK
Applying auth.0006_require_contenttypes_0002... OK
Applying auth.0007_alter_validators_add_error_messages... OK
Applying auth.0008_alter_user_username_max_length... OK
Applying auth.0009_alter_user_last_name_max_length... OK
Applying products.0001_initial... OK
Applying products.0002_product_deleted_at... OK
Applying sales.0001_initial... OK
Applying sessions.0001_initial... OK
```

現時点で適用可能なすべてのマイグレーションがアプリを横断して適用されます。

showmigrations

マイグレーション一覧と **適用**、**未適用** 状態をアプリごとに表示します。

```
$ ./manage.py showmigrations --help
usage: manage.py showmigrations [-h] [--database DATABASE] [--list | --plan]
                                [--version] [-v {0,1,2,3}]
                                [--settings SETTINGS]
                                [--pythonpath PYTHONPATH] [--traceback]
                                [--no-color]
                                [app_label [app_label ...]]

Shows all available migrations for the current project

positional arguments:
  app_label              App labels of applications to limit the output to.
```

位置引数は **app_label** だけで、**makemigrations** と同様に可変長で省略可能です。

先程マイグレーションの構造を説明しましたが、各アプリの **migrations/** 配下のファイルと、**django_migrations** テーブルの内容を突き合わせた結果を見せているに過ぎません。

トラブルシューティングのときに活躍します。

<https://docs.djangoproject.com/el/2.1/ref/django-admin/#showmigrations>

実行例

```
$ ./manage.py showmigrations
accounts
[X] 0001_initial
[X] 0002_dummy
[X] 0003_dummy
[ ] 0004_dummy
[ ] 0005_dummy
admin
[X] 0001_initial
[X] 0002_logentry_remove_auto_add
[X] 0003_logentry_add_action_flag_choices
auth
[X] 0001_initial
[X] 0002_alter_permission_name_max_length
[X] 0003_alter_user_email_max_length
[X] 0004_alter_user_username_opts
[X] 0005_alter_user_last_login_null
[X] 0006_require_contenttypes_0002
```

```
[X] 0007_alter_validators_add_error_messages
[X] 0008_alter_user_username_max_length
[X] 0009_alter_user_last_name_max_length
contenttypes
[X] 0001_initial
[X] 0002_remove_content_type_name
products
[X] 0001_initial
[X] 0002_product_deleted_at
sales
[X] 0001_initial
```

上記だと accounts アプリの 0004 と 0005 だけが未適用だとわかりますね。

squashmigrations

複数のマイグレーションファイルを一つに統合するコマンドです。

```
usage: manage.py squashmigrations [-h] [--no-optimize] [--noinput]
                                   [--squashed-name SQUASHED_NAME] [--version]
                                   [-v {0,1,2,3}] [--settings SETTINGS]
                                   [--pythonpath PYTHONPATH] [--traceback]
                                   [--no-color]
                                   app_label [start_migration_name]
                                   migration_name

Squashes an existing set of migrations (from first until specified) into a
single new one.

positional arguments:
  app_label              App label of the application to squash migrations for.
  start_migration_name  Migrations will be squashed starting from and
                        including this migration.
  migration_name        Migrations will be squashed until and including this
                        migration.
```

位置引数は **app_label** (アプリ名)、 **start_migration_name** (マイグレーションの始点)、 **migration_name** (マイグレーションの終点) です

このうち 始点となる **start_migration_name** だけが省略可能です。

位置引数なのに省略? と思うかもしれませんが、 **range** ビルトイン関数のようなもので、引数が足りないと始点が先頭からと解釈されるようです。

実行すると始点から終点までのマイグレーションが統合され、始点と同じ番号で出力されます。

続いてオプションです。

--squashed-name

当該指定すると マイグレーションのファイル名(連番より後ろの部分)を変更できます。

makemigrations における **--name** と同じようなものですね。

なぜ違うオプション名にしまったのか。コレガワカラナイ。

squashmigrations で作られたマイグレーションファイルには **replaces** 属性が存在します。この属性(タブルのリスト)に該当するマイグレーションは適用せずに無視します。

統合前の古いファイルは勝手に消えたりはしないので、不要な場合は各自で処分しましょう。 **replaces** 対象のマイグレーションが存在しなくてもエラーにはなりません。

<https://docs.djangoproject.com/el/2.1/ref/django-admin/#squashmigrations>

実行例

```
$ ./manage.py squashmigrations sales 0001 0003 --squashed-name squashed
Will squash the following migrations:
- 0001_initial
- 0002_summary
- 0003_renamed_and_added
Do you wish to proceed? [yN] y
Optimizing...
  Optimized from 5 operations to 2 operations.
Created new squashed migration /Users/crohaco/Projects/cronote/posts/2018/django-migration/apps/sales/migrations/0
  You should commit this migration but leave the old ones in place;
  the new migration will be used for new installs. Once you are sure
  all instances of the codebase have applied the migrations you squashed,
  you can delete them.
```

ちなみにこの場合は 0001 は省略しても同じ結果になります

中身は概ねこんな感じでしょうか

```
from django.db import migrations, models
import django.db.models.deletion
import django.utils.timezone

class Migration(migrations.Migration):

    replaces = [('sales', '0001_initial'), ('sales', '0002_summary'), ('sales', '0003_renamed_and_added')]

    dependencies = [
        ('products', '0002_product_deleted_at'),
    ]

    operations = [
        migrations.CreateModel(
            name='Sales',
            fields=[
                ('id', models.AutoField(auto_created=True, primary_key=True, serialize=False, verbose_name='ID')),
                ('sold_at', models.DateTimeField(default=django.utils.timezone.now)),
                ('product', models.ForeignKey(on_delete=django.db.models.deletion.CASCADE, to='products.Product')),
            ],
        ),
        migrations.CreateModel(
            name='Summary',
            fields=[
                ('id', models.AutoField(auto_created=True, primary_key=True, serialize=False, verbose_name='ID')),
                ('date', models.DateField()),
                ('total_price', models.IntegerField()),
                ('total_sales', models.IntegerField(default=0)),
            ],
        ),
    ]
```

`sales/migrations/0001_squashed.py` が作られたら 統合された マイグレーション (0001, 0002, 0003) を消して再マイグレーションして、同じテーブルが作られることを検証してみてください。

▶ 答え合わせ

sqlmigrate

特定のマイグレーションによって実行されるSQLを表示します。

この コマンドは DBコネクションが必要です。実際の制約名は接続してみないと解決できないですね。

```
$ ./manage.py sqlmigrate --help
usage: manage.py sqlmigrate [-h] [--database DATABASE] [--backwards]
                             [--version] [-v {0.1.2.3}] [--settings SETTINGS]
```

```
[-version] [-v {0,1,2,3}] [-settings SETTINGS]
[--pythonpath PYTHONPATH] [--traceback]
[--no-color]
app_label migration_name
```

Prints the SQL statements for the named migration.

positional arguments:

app_label	App label of the application containing the migration.
migration_name	Migration name to print the SQL for.

位置引数は **app_label** (アプリ名) と **migration_name** (マイグレーション名) でいずれも必須です。

続いてオプション.

backwards

逆適用 する SQL を表示します。

<https://docs.djangoproject.com/en/2.1/ref/django-admin/#sqlmigrate>

実行例


以下が順適用のSQLです。

```
$ ./manage.py sqlmigrate products 0001
BEGIN;
--
-- Create model Category
--
CREATE TABLE "products_category" ("id" integer NOT NULL PRIMARY KEY AUTOINCREMENT, "name" varchar(30) NOT NULL, "c
--
-- Create model Price
--
CREATE TABLE "products_price" ("id" integer NOT NULL PRIMARY KEY AUTOINCREMENT, "price" integer NOT NULL, "effecti
--
-- Create model Product
--
CREATE TABLE "products_product" ("id" integer NOT NULL PRIMARY KEY AUTOINCREMENT, "name" varchar(255) NOT NULL, "c
--
-- Add field product to price
--
ALTER TABLE "products_price" RENAME TO "products_price_old";
CREATE TABLE "products_price" ("id" integer NOT NULL PRIMARY KEY AUTOINCREMENT, "price" integer NOT NULL, "effecti
INSERT INTO "products_price" ("id", "price", "effective_date_start", "effective_date_end", "product_id") SELECT "i
DROP TABLE "products_price_old";
CREATE INDEX "products_product_category_id_9b594869" ON "products_product" ("category_id");
CREATE INDEX "products_price_product_id_8481dedb" ON "products_price" ("product_id");
COMMIT;
```

以下が逆適用のSQLです

```
$ ./manage.py sqlmigrate products 0001 --backwards
BEGIN;
--
-- Add field product to price
--
ALTER TABLE "products_price" RENAME TO "products_price_old";
CREATE TABLE "products_price" ("id" integer NOT NULL PRIMARY KEY AUTOINCREMENT, "price" integer NOT NULL, "effecti
INSERT INTO "products_price" ("id", "price", "effective_date_start", "effective_date_end") SELECT "id", "price", "
DROP TABLE "products_price_old";
--
-- Create model Product
--
DROP TABLE "products_product";
--
```

```
-- Create model Price
--
DROP TABLE "products_price";
--
-- Create model Category
--
DROP TABLE "products_category";
COMMIT;
```

A horizontal scrollbar is located at the bottom of the code editor. It consists of a light gray track with a darker gray slider bar positioned approximately one-third of the way from the left. Small arrowheads are visible at both ends of the track.