Engineer

ORM

Bibliothèque liant tables d'une base de données et objets

Ce que c'est

Un ORM, abréviation d'object-relational mapper, est une bibliothèque qui associe les tables d'une base relationnelle à des classes du code applicatif, et leurs lignes à des objets. Le code demande un auteur et reçoit un objet `Author`, modifie un de ses attributs puis l'enregistre ; l'ORM écrit les instructions `SELECT`, `INSERT` et `UPDATE` qui réalisent le travail. Les modèles de Django et l'ORM de SQLAlchemy, tous deux écrits en Python, sont deux exemples connus sur lesquels cet article s'appuie.

Cette correspondance est nécessaire parce que les deux côtés organisent les données différemment. Martin Fowler le dit clairement dans sa description du patron Data Mapper : objets et bases relationnelles structurent les données de façons différentes, et certaines parties d'un modèle objet, comme les collections et l'héritage, n'ont pas d'équivalent dans une base relationnelle. Faire passer les données de l'un à l'autre devient un problème à part entière, auquel un ORM apporte une réponse prête à l'emploi.

Les éléments de cette réponse ont été nommés dans le livre de Fowler Patterns of Enterprise Application Architecture, publié en 2003, dont le catalogue de patrons est en ligne. Deux patrons décrivent la structure générale. Active Record est un objet qui enveloppe une ligne de table, gère ses propres accès à la base et porte la logique métier de ces données ; les principes de conception de Django indiquent que ses modèles suivent ce patron. Data Mapper est une couche séparée qui déplace les données entre objets et base tout en gardant chacun indépendant de l'autre, si bien que les objets n'ont pas besoin de savoir qu'une base existe. Des patrons plus petits les entourent. Identity Map garantit que chaque objet n'est chargé qu'une fois. Unit of Work garde une liste des objets modifiés par une unité de travail et écrit les changements ensemble. Lazy Load est un objet qui ne détient qu'une partie des données nécessaires au code, mais sait récupérer le reste. La documentation de SQLAlchemy en nomme deux

directement : sa `Session` conserve les objets chargés dans une table d'identité, avec un seul objet par clé primaire, et sa façon de rassembler les changements puis de les écrire dans la base suit le patron Unit of Work.

Utilisez le concept pour prendre une décision

Utilisez « Qu'est-ce qu'un ORM ? » pour éclairer une décision réelle. Nous pouvons vous aider à relier ce concept au système que vous construisez ou achetez. La partie invisible pour les utilisateurs : où résident les données, quelles règles s'appliquent et comment les autres systèmes peuvent y accéder.

Découvrir nos prestations Contactez-nous →

Vérifiez les hypothèses derrière le terme

Appuyez-vous sur l'explication et les réalisations liées pour formuler les questions de votre projet. Nous pouvons examiner les hypothèses avec vous avant le choix d'un outil. 6 pipelines ETL/ELT conçus et exploités — 6 pipelines ETL sur trois moteurs de bases de données

6 pipelines ETL/ELT conçus et exploités Contactez-nous →

Pourquoi c'est important

La première raison est la quantité de code économisée. Dans son essai ORM Hate de 2012, Fowler décrit le problème de mise en correspondance entre objets et données relationnelles comme difficile : il faut maintenir en accord deux représentations très différentes des mêmes données, et tout changement d'un côté doit se répercuter de l'autre. Il estime qu'un ORM traite environ 80 à 90 % de ces problèmes et qu'un framework qui épargne 80 % de ce travail à une équipe mérite sa place même s'il s'arrête là. Chaque table que l'équipe n'a pas à associer manuellement à des classes représente une occasion de moins d'affecter la valeur d'une colonne au mauvais attribut.

La deuxième raison est la cohérence dans un programme en cours d'exécution. Avec une table d'identité, deux requêtes qui trouvent la même ligne renvoient le même objet : un changement par une référence est donc visible par l'autre. La documentation de SQLAlchemy le montre en chargeant deux fois la même clé primaire et en retrouvant le même objet. Avec une unité de travail, l'ORM rassemble les insertions, mises à jour et suppressions en attente et, selon la liste des fonctionnalités de SQLAlchemy, les écrit toutes en un lot. Le code applicatif n'a donc pas à se rappeler quels objets il a modifiés. Django fait délibérément le choix inverse : ses principes de conception demandent un `save()` explicite plutôt qu'un enregistrement en coulisses, afin que le framework envoie aussi peu d'instructions SQL que possible.

La troisième raison est la position de la frontière. Fowler décrit un bon ORM comme celui qui se charge du travail courant et « ménage des trappes pour descendre travailler en SQL » lorsque c'est nécessaire. Il affirme sans détour que

toute personne qui construit une application sur une base relationnelle doit savoir comment elle fonctionne. Les principes de conception de Django disent la même chose côté framework : l'API de base de données doit être conçue comme un raccourci plutôt que comme une réponse complète et faciliter l'écriture de SQL personnalisé, qu'il s'agisse d'une instruction entière ou d'une seule clause `WHERE`. Un ORM mérite sa place pour la majorité des accès aux données, ceux qui relèvent du travail courant ; c'est sur le reste que l'équipe emploie ses compétences en SQL.

À quoi cela ressemble en pratique

Le coût qu'un ORM masque le plus souvent est le nombre de requêtes envoyées. Dans SQLAlchemy 2.1, les relations se chargent paresseusement par défaut : les objets liés sont récupérés lorsque le code accède pour la première fois à l'attribut. La documentation en précise la conséquence : pour N objets chargés, accéder à leurs attributs chargés paresseusement produit N+1 instructions `SELECT`. L'exemple ci-dessous suppose des classes `Author` et `Book` déjà associées, avec une relation un-à-plusieurs `Author.books`, et parcourt les auteurs de trois façons.

```
from sqlalchemy import select
from sqlalchemy.orm import raiseload, selectinload

# The default: one SELECT for the authors, then one more for each
# author whose books the loop touches, N+1 statements in all.
authors = session.scalars(select(Author)).all()
for author in authors:
    print(author.name, len(author.books))

# select IN loading: a second SELECT fetches the books of every
# author at once, with the authors' primary keys in an IN clause.
stmt = select(Author).options(selectinload(Author.books))
for author in session.scalars(stmt).all():
    print(author.name, len(author.books))

# raise loading: touching author.books now raises an error
# instead of quietly sending another query.
stmt = select(Author).options(raiseload(Author.books))
```

Pour les collections un-à-plusieurs et plusieurs-à-plusieurs, la documentation présente `selectinload()` comme généralement la meilleure stratégie. `joinedload()`, qu'elle décrit comme la stratégie la plus polyvalente, ajoute un `JOIN` à l'instruction initiale, un `LEFT OUTER JOIN` par défaut, pour que les lignes

liées arrivent dans le même jeu de résultats. `raiseload()` protège contre les chargements indésirables. Sa forme avec joker, `raiseload("*")`, étend cette protection aux autres relations, y compris celles des objets effectivement chargés par l'instruction.

Django 6.1 donne son propre nom au même choix : les modes de récupération. Lorsque le code accède à un champ que la requête initiale n'a pas chargé, comme une clé étrangère, une relation un-à-un ou un champ omis avec `defer()` ou `only()`, le mode par défaut, `FETCH_ONE`, le récupère pour cette seule instance. La documentation nomme elle-même le résultat : le problème des requêtes N+1. `FETCH_PEERS`, nouveau en version 6.1, récupère le champ manquant pour cette instance et toutes les autres du même `QuerySet`, selon l'idée que le code qui a besoin d'un champ pour une ligne en a généralement besoin pour toutes. `FETCH_RAISE` lève plutôt `FieldFetchBlocked`. Le mode se définit par `QuerySet`, comme dans `Book.objects.fetch_mode(models.FETCH_PEERS)`, ou comme valeur par défaut du modèle au moyen d'un gestionnaire personnalisé. Django le copie aux objets liés qu'il récupère. Lorsque `FETCH_PEERS` ne convient pas ou n'est pas assez efficace, le guide d'optimisation renvoie à `select_related()` et `prefetch_related()`.

Les pages de l'entreprise elle-même placent la frontière plus loin vers la base de données que la plupart des approches. Sa pile technique traite SQL comme un langage à part entière plutôt que comme du code généré par un ORM, parce que l'essentiel des interventions sur les performances pour lesquelles l'entreprise est sollicitée concerne cette couche. Dans son travail pour NextMariner, la fondatrice a poussé cette approche jusqu'à sa conclusion logique avec une couche de données PostgreSQL fondée sur les fonctions : chaque opération possible dans l'application est l'une des 1 275 fonctions stockées réparties dans 34 schémas, et Go n'accède directement à aucune table. Le rôle utilisé par l'API pour se connecter détient `EXECUTE` sur ces fonctions et `USAGE` sur leurs schémas, sans aucun autre droit : il ne peut donc pas toucher directement une table. La page expose le prix : un nouveau champ oblige à modifier une table et une fonction, pas à ajouter une ligne à une requête.

Où les problèmes surviennent

Une boucle envoie une requête par ligne. Le motif N+1 ci-dessus paraît inoffensif sur une petite base de développement et ralentit une page à mesure que le nombre de lignes augmente. Décidez du chargement pour chaque chemin important, avec `selectinload()` ou `joinedload()` dans SQLAlchemy, et `FETCH_PEERS`, `select_related()` ou `prefetch_related()` dans Django. Faites ensuite

échouer le cas imprévu avec une erreur explicite : `raiseload()` dans SQLAlchemy ou `FETCH_RAISE`, que la documentation de Django décrit comme un moyen d'empêcher les requêtes involontaires dans le code critique pour les performances.

Personne ne lit le SQL écrit par l'ORM. Un ORM écrit du SQL que personne dans l'équipe n'a choisi, et ce SQL reste invisible jusqu'à ce qu'une page ralentisse. Le guide d'optimisation de Django commence là : déterminez quelles requêtes une page exécute et leur coût, utilisez `QuerySet.explain()` pour voir comment la base exécute un `QuerySet` donné, et lisez `django.db.connection.queries` pour voir ce que Django écrit au nom du code. Intégrez la lecture de ce SQL à la revue de toute page ou tâche importante et examinez le plan d'exécution produit par la base pour les instructions lentes.

Le travail de la base s'exécute en Python. Charger les lignes uniquement pour les filtrer, comparer deux de leurs champs ou les additionner transporte chaque ligne sur la connexion pour faire dans l'application ce que la base pourrait faire là où les données se trouvent déjà. Le guide de Django énumère les alternatives : `filter()` et `exclude()` pour filtrer dans la base, les expressions `F()` pour comparer les champs d'un même modèle, et `annotate()` pour agréger dans la base. Les écritures suivent la même règle, avec `bulk_create()` et `bulk_update()` à la place d'un `save()` par objet, après avoir vérifié les réserves que la documentation énumère pour les deux. Pour un très grand résultat, un `QuerySet` retient tout ce qu'il a lu et peut utiliser beaucoup de mémoire ; `iterator()` peut aider.

On force la dernière partie à passer par l'ORM. L'estimation de Fowler lui-même laisse 10 à 20 % des problèmes de correspondance, et cette dernière partie demande toujours, selon lui, un travail attentif. Tordre l'ORM jusqu'à lui faire produire une instruction précise cache ce travail dans du SQL généré que personne n'a choisi. Écrivez ces instructions en SQL et revoyez-les comme tout autre code. Le guide de Django nomme les étapes pour descendre à ce niveau : l'expression `RawSQL` ajoute du SQL explicite à une requête, moins portable mais plus puissant ; au-delà, l'équipe peut écrire son propre SQL pour récupérer des données ou remplir des modèles. Gardez l'ORM pour les enregistrements courants, là où il économise le plus et coûte le moins.

Pour aller plus loin

- Catalogue des patrons de Patterns of Enterprise Application Architecture — les résumés par Martin Fowler d'Active Record, Data Mapper, Unit of Work, Identity Map et Lazy Load.
- La haine des ORM — pourquoi la mise en correspondance entre objets et données relationnelles est difficile et quelle part un framework peut prendre en charge.
- Techniques de chargement des relations — chargements paresseux, par sélection IN, par jointure et avec levée d'erreur dans SQLAlchemy 2.1, et choix de la méthode.
- Modes de récupération — `FETCH_ONE`, `FETCH_PEERS` et `FETCH_RAISE`, nouveaux dans Django 6.1.
- Optimisation des accès à la base — profilage, évaluation des `QuerySet`, travail dans la base, méthodes par lots et SQL brut.

Glossaire

Choisissez selon vos contraintes réelles

Présentez le processus de travail, les interfaces existantes et les utilisateurs qui ont besoin du changement. Nous pouvons définir une livraison testable et une transmission permettant à votre équipe d'assurer la maintenance des éléments livrés.

Consulter les tarifs Contactez-nous →

Rendez la terminologie concrète

Demandez comment « Qu'est-ce qu'un ORM ? » s'applique à votre organisation. Un exemple concret tiré de votre système actuel suffit pour engager la discussion.

Découvrir nos prestations

welcome@engineer.company +45 50 10 61 84 Copenhague, Danemark

Emportez cette page avec vous

Fiche de contact

Créé avec soin et amour par D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Droit d'auteur © 2025 – aujourd'hui · Engineer ApS · Tous droits réservés.

Imprimé depuis https://engineer.company/fr/glossary/orm/