Engineer

ORM

бібліотека, що відображає таблиці бази даних на об'єкти коду

Що це таке

ORM, скорочення від object-relational mapper, — це бібліотека, яка пов’язує таблиці реляційної бази даних із класами в коді застосунку, а рядки цих таблиць — з об’єктами. Код просить автора й отримує об’єкт `Author`, змінює один з його атрибутів і зберігає його, а ORM пише інструкції `SELECT`, `INSERT` і `UPDATE`, які виконують цю роботу. Моделі Django та ORM-частина SQLAlchemy, обидві написані на Python, — два відомі приклади, і саме на них спирається ця стаття.

Відображення потрібне тому, що дві сторони організовують дані по-різному. Мартін Фаулер прямо каже про це в описі патерну Data Mapper: об’єкти й реляційні бази даних структурують дані кожен по-своєму, а частини об’єктної моделі, як-от колекції та успадкування, не мають відповідника в реляційній базі. Перенесення даних між ними стає окремою задачею, і ORM — готова відповідь на неї.

Складники цієї відповіді отримали назви в книзі Фаулера Patterns of Enterprise Application Architecture 2003 року, а каталог патернів із неї вільно доступний онлайн. Два патерни описують загальну форму. Active Record — це об’єкт, який обгортає один рядок таблиці, сам відповідає за доступ до бази й несе доменну логіку для цих даних; за філософією дизайну Django, моделі там наслідують саме цей патерн. Data Mapper — окремий шар, який переносить дані між об’єктами та базою і тримає їх незалежними одне від одного, тож об’єктам навіть не треба знати, що база взагалі існує. Навколо них є кілька менших патернів. Identity Map стежить, щоб кожен об’єкт завантажувався лише один раз. Unit of Work веде список об’єктів, які змінила певна частина роботи, і записує зміни разом. Lazy Load — це об’єкт, що містить лише частину даних, потрібних коду, але знає, як отримати решту. Документація SQLAlchemy прямо називає два з них: `Session` тримає

завантажені об’єкти в identity map, де на кожен первинний ключ припадає лише один об’єкт, а спосіб, яким вона збирає зміни та скидає їх у базу через flush, — це патерн unit of work.

Застосуйте поняття до рішення

Застосуйте «Що таке ORM?» до конкретного рішення. Ми можемо пов'язати поняття із системою, яку ви створюєте або купуєте. Та частина, якої користувач не бачить: де зберігаються дані, які діють правила і як інші системи можуть із нею взаємодіяти.

Переглянути наші послуги Зв'язатися з нами →

Перевірте припущення за терміном

Використайте пояснення та пов'язані роботи, щоб сформулювати запитання для проєкту. Ми можемо перевірити припущення разом до вибору інструмента. Спроєктувала й підтримувала 6 пайплайнів ETL/ELT — 6 пайплайнів ETL на трьох рушіях баз даних

Спроєктувала й підтримувала 6 пайплайнів ETL/ELT Зв'язатися з нами →

Чому це важливо

Перша причина — скільки коду ORM заощаджує. У есеї ORM Hate 2012 року Фаулер називає задачу об’єктно-реляційного відображення складною: йдеться про те, щоб тримати синхронними два дуже різні представлення тих самих даних, і зміну з одного боку треба перенести на інший. За його оцінкою, ORM розв’язує приблизно від 80 до 90% задач відображення, і він вважає, що фреймворк, який позбавляє команду 80% цієї роботи, вартий того, навіть якщо на цьому він і зупиняється. Кожна таблиця, яку команді не доводиться відображати вручну, — це на одне місце менше, де стовпець може потрапити не в той атрибут.

Друга причина — узгодженість усередині запущеної програми. З identity map два запити, які знаходять той самий рядок, повертають той самий об’єкт, тож зміна, зроблена через одне посилання, видна через інше; документація SQLAlchemy показує це, двічі завантажуючи той самий первинний ключ і отримуючи той самий об’єкт. З unit of work ORM збирає вставлення, оновлення й видалення, що очікують, і, за списком можливостей SQLAlchemy, скидає їх у базу одним пакетом, тож коду застосунку не треба самому пам’ятати, яких об’єктів він торкався. Django свідомо робить протилежний вибір: його філософія дизайну вимагає явного виклику `save()` замість того, щоб фреймворк зберігав дані непомітно, і так надсилає якомога менше SQL-інструкцій.

Третя причина — те, де проходить межа. Фаулер описує добру ORM як таку, що бере на себе рутину, але залишає люк, через який можна спуститися до SQL, коли це потрібно, і прямо каже, що кожен, хто будує на реляційній базі

даних, мусить знати, як така база працює. Філософія дизайну Django каже те саме з боку фреймворку: API для роботи з базою має розуміти, що воно лише скорочений шлях, а не обов'язково вся відповідь, і фреймворк має робити власний SQL простим у написанні — чи то цілі інструкції, чи лише власна умова `WHERE`. ORM заслуговує на своє місце в рутинній частині доступу до даних; решта — те, на що команда витрачає своє знання SQL.

Як це виглядає на практиці

Ціна, яку ORM приховує найчастіше, — кількість запитів, які вона надсилає. У SQLAlchemy 2.1 зв'язки за замовчуванням завантажуються ліниво: пов'язані об'єкти отримуються тоді, коли код уперше звертається до атрибута. Документація прямо називає наслідок: для N завантажених об'єктів звернення до їхніх ліниво завантажуваних атрибутів дає N+1 інструкцій `SELECT`. Приклад нижче припускає, що класи `Author` і `Book` уже відображено зі зв'язком один-до-багатьох `Author.books`, і проходить авторами трьома способами.

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

# За замовчуванням: один SELECT для авторів, а потім ще по одному
# для кожного автора, чиї книжки зачіпає цикл, разом N+1 інструкцій.
authors = session.scalars(select(Author)).all()
for author in authors:
    print(author.name, len(author.books))

# select IN: другий SELECT отримує книжки всіх авторів одразу,
# з їхніми первинними ключами в умові IN.
stmt = select(Author).options(selectinload(Author.books))
for author in session.scalars(stmt).all():
    print(author.name, len(author.books))

# raise: звернення до author.books тепер спричиняє помилку,
# а не непомітно надсилає ще один запит.
stmt = select(Author).options(raiseload(Author.books))
```

Для колекцій один-до-багатьох і багато-до-багатьох документація називає `selectinload()` загалом найкращою стратегією. `joinedload()`, яку вона називає найуніверсальнішою, додає `JOIN` до початкової інструкції, за замовчуванням `LEFT OUTER JOIN`, тож пов'язані рядки приходять в одному наборі результатів. `raiseload()` існує, щоб захищати від небажаних

завантажень, а її форма з символом підстановки, `raiseload("*")`, поширює цей захист і на інші зв'язки, зокрема на зв'язки об'єктів, які інструкція таки завантажує.

Django 6.1 дає тому самому вибору власну назву: fetch modes, режими отримання. Коли код звертається до поля, якого початковий запит не завантажив, наприклад до зовнішнього ключа, зв'язку один-до-одного або поля, пропущеного через `defer()` чи `only()`, режим за замовчуванням `FETCH_ONE` отримує його лише для цього одного екземпляра, і документація сама називає наслідок проблемою N+1 запитів. `FETCH_PEERS`, новий у 6.1, отримує відсутнє поле для цього екземпляра й для всіх інших екземплярів із того самого `QuerySet`, виходячи з того, що коду, якому поле потрібне для одного рядка, зазвичай потрібне воно й для решти. `FETCH_RAISE` натомість спричиняє `FieldFetchBlocked`. Режим задають для окремого `QuerySet`, як у `Book.objects.fetch_mode(models.FETCH_PEERS)`, або як типовий для моделі через власний менеджер, і Django копіює його на пов'язані об'єкти, які отримує. Там, де `FETCH_PEERS` не підходить або недостатньо ефективний, посібник з оптимізації радить `select_related()` і `prefetch_related()`.

Власні сторінки компанії проводять цю межу ближче до бази даних, ніж більшість. Сторінка технологій розглядає SQL як повноцінну мову, а не те, що генерує ORM, бо саме там здебільшого лежить робота з продуктивністю, заради якої компанію запрошують. Під час роботи в NextMariner засновниця довела цю думку до кінця й спроєктувала функційний рівень даних PostgreSQL: кожна операція застосунку — це одна з 1 275 збережених функцій у 34 схемах, і прямого доступу до таблиць із Go немає взагалі. Роль, від імені якої API входить у базу, має привілей `EXECUTE` для цих функцій і `USAGE` для їхніх схем — і більше нічого, тож напряму звернутися до таблиці вона не може. Сторінка відверто називає ціну: нове поле означає зміну і таблиці, і функції, а не ще один рядок у запиті.

Де виникають проблеми

Цикл надсилає один запит на кожен рядок. Описаний вище патерн N+1 здається нешкідливим на невеликій базі розробника, а потім сповільнює сторінку пропорційно до кількості рядків. Визначте спосіб завантаження для кожного важливого шляху в коді: `selectinload()` або `joinedload()` у SQLAlchemy, `FETCH_PEERS`, `select_related()` або `prefetch_related()` у Django. Потім зробіть так, щоб незапланований випадок падав гучно: `raiseload()` у SQLAlchemy або `FETCH_RAISE`, який документація Django описує як спосіб запобігти ненавмисним запитам у критичному до продуктивності коді.

Ніхто не читає SQL, який пише ORM. ORM пише SQL, якого ніхто в команді не обирав, і цей SQL залишається невидимим, доки сторінка не почне гальмувати. Посібник Django з оптимізації починається саме з цього: з'ясуйте, які запити виконує сторінка і скільки вони коштують, скористайтеся `QuerySet.explain()`, щоб побачити, як база виконує певний `QuerySet`, і прочитайте `django.db.connection.queries`, щоб побачити, що Django пише від імені коду. Зробіть читання цього SQL сталою частиною рев'ю кожної важливої сторінки чи задачі, а повільні інструкції проганяйте через план запиту самої бази.

Робота, якій місце в базі, виконується в Python. Завантажувати рядки лише для того, щоб відфільтрувати їх, порівняти два їхні поля чи підсумувати, означає переносити кожен рядок через з'єднання, щоб у коді застосунку зробити те, що база могла б зробити там, де дані вже лежать. Посібник Django перелічує альтернативи: `filter()` і `exclude()` для фільтрування в базі, вирази `F()` для порівняння полів у межах однієї моделі та `annotate()` для агрегування в базі. Записування підпорядковується тому самому правилу: `bulk_create()` і `bulk_update()` замість окремого `save()` для кожного об'єкта, після перевірки застережень, які документація наводить для обох. Для дуже великого набору результатів `QuerySet` тримає все, що прочитав, і може займати багато пам'яті, і тоді може допомогти `iterator()`.

Останню частину проштовхують крізь ORM. Власна оцінка Фаулера залишає від 10 до 20% задач відображення, і ця остання частина, за його словами, завжди потребує ретельної роботи. Якщо гнути ORM, доки вона не видасть певну інструкцію, ця робота ховається в згенерованому SQL, якого ніхто не обирав. Пишіть такі інструкції на SQL і переглядайте їх, як будь-який інший код. Посібник Django називає щаблі вниз: вираз `RawSQL` додає до запиту явний SQL, менш переносний, але потужніший, а далі команда може писати власний SQL, щоб отримувати дані або заповнювати моделі. Залишайте ORM для рутинних записів, де вона заощаджує найбільше й коштує найменше.

Що почитати далі

- Catalog of Patterns of Enterprise Application Architecture — короткі описи Active Record, Data Mapper, Unit of Work, Identity Map і Lazy Load від Мартіна Фаулера.
- ORM Hate — чому об'єктно-реляційне відображення складне і скільки з нього може взяти на себе фреймворк.

- Relationship Loading Techniques — лініве завантаження, select IN, joined і raise у SQLAlchemy 2.1 та вибір між ними.
- Fetch modes — `FETCH_ONE`, `FETCH_PEERS` і `FETCH_RAISE`, нові в Django 6.1.
- Database access optimization — профілювання, обчислення `QuerySet`, робота в базі, пакетні методи й сирий SQL.

Словник

Обирайте за фактичними умовами

Опишіть робочий процес, наявні інтерфейси та користувачів, яким потрібна зміна. Ми можемо визначити перевірюваний результат і спосіб передачі, який дасть вашій команді змогу підтримувати його.

Дивитися, скільки це коштує Зв'язатися з нами →

Зробіть термінологію практичною

Запитайте, як «Що таке ORM?» стосується вашої організації. Конкретного прикладу з поточної системи достатньо для початку розмови.

Переглянути наші послуги

welcome@engineer.company +45 50 10 61 84 Копенгаген, Данія

Візьміть цю сторінку із собою

Візитівка

Дбайливо зроблено з любов'ю — D.

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

Авторське право © 2025 – дотепер · Engineer ApS · Усі права захищено.

Надруковано з https://engineer.company/uk/glossary/orm/