Engineer

ORM

ett bibliotek som kopplar databastabeller till objekt i kod

Vad det är

En ORM, kort för object-relational mapper, är ett bibliotek som kopplar tabellerna i en relationsdatabas till klasser i applikationskoden, och raderna i dem till objekt. Koden ber om en författare och får tillbaka ett `Author` -objekt, ändrar ett av dess attribut och sparar det, och ORM-biblioteket skriver de `SELECT` -, `INSERT` - och `UPDATE` -satser som gör jobbet. Modellerna i Django och ORM-delen av SQLAlchemy, båda skrivna i Python, är två välkända exempel, och det är dem den här artikeln bygger på.

Mappningen behövs eftersom de två sidorna organiserar data på olika sätt. Martin Fowler säger det rakt ut i sin beskrivning av mönstret Data Mapper: objekt och relationsdatabaser strukturerar data på var sitt sätt, och delar av en objektmodell, som samlingar och arv, har ingen motsvarighet i en relationsdatabas. Att flytta data mellan de två blir ett problem i sig, och en ORM är ett färdigt svar på det.

Delarna i det svaret fick namn i Fowlers bok Patterns of Enterprise Application Architecture från 2003, och bokens mönsterkatalog finns fritt tillgänglig på nätet. Två mönster beskriver den övergripande formen. Active Record är ett objekt som kapslar in en rad i en tabell, själv sköter åtkomsten till databasen och bär domänlogiken för de data det håller; enligt designfilosofin i Django följer modellerna där just det mönstret. Data Mapper är ett separat lager som flyttar data mellan objekten och databasen och håller dem oberoende av varandra, så att objekten inte behöver veta att det finns en databas alls. Runt dem finns flera mindre mönster. Identity Map ser till att varje objekt bara läses in en gång. Unit of Work håller en lista över de objekt som ett arbetsmoment har ändrat och skriver ut ändringarna tillsammans. Lazy Load är ett objekt som bara innehåller en del av de data koden behöver men vet hur resten hämtas. Dokumentationen till SQLAlchemy nämner två av dem direkt: `Session` håller inlästa objekt

i en identity map, med bara ett objekt per primärnyckel, och sättet den samlar ändringar på och skickar ut dem till databasen med en flush är unit of work-mönstret.

Använd begreppet i ett beslut

Använd ”Vad är en ORM?” i ett konkret beslut. Vi kan knyta begreppet till systemet ni bygger eller köper. Den del användarna aldrig ser: där data lagras, vilka reglerna är och hur andra system får kommunicera med den.

Se våra tjänster Hör av er →

Granska antagandena bakom ordet

Använd förklaringen och det länkade arbetet för att formulera frågor till projektet. Vi kan undersöka antagandena tillsammans innan ett verktyg väljs. Utformade och drev 6 ETL/ELT-pipelines — 6 ETL-pipelines över tre databasmotorer

Utformade och drev 6 ETL/ELT-pipelines Hör av er →

Varför det spelar roll

Det första skälet är hur mycket kod en ORM sparar. I essän ORM Hate från 2012 kallar Fowler problemet med objektrelationell mappning svårt: det handlar om att hålla två mycket olika representationer av samma data i takt, och en ändring på den ena sidan måste föras över till den andra. Han uppskattar att en ORM hanterar ungefär 80 till 90 % av mappningsproblemen och menar att ett ramverk som besparar ett team 80 % av det arbetet är värt att ha, även om det stannar där. Varje tabell som teamet slipper mappa för hand är ett ställe mindre där en kolumn kan läsas in i fel attribut.

Det andra skälet är konsekvens inuti ett körande program. Med en identity map returnerar två frågor som hittar samma rad samma objekt, så att en ändring som görs via den ena referensen syns via den andra; dokumentationen till SQLAlchemy visar det genom att hämta samma primärnyckel två gånger och få exakt samma objekt tillbaka. Med en unit of work samlar ORM-biblioteket väntande infogningar, uppdateringar och borttagningar och skriver enligt funktionslistan för SQLAlchemy ut dem i en enda omgång, så att applikationskoden inte själv behöver hålla reda på vilka objekt den har rört. Django gör medvetet det motsatta valet: designfilosofin kräver ett uttryckligt anrop till `save()` i stället för att ramverket sparar i det tysta, så att det skickas så få SQL-satser som möjligt.

Det tredje skälet är var gränsen går. Fowler beskriver en bra ORM som en som tar hand om rutinarbetet men ger ett manhål ner till SQL-koden när det behövs, och han säger rakt ut att den som bygger på en relationsdatabas måste veta hur en sådan databas fungerar. Designfilosofin i Django säger samma sak från

ramverkets sida: databas-API:t ska veta att det är en genväg och inte nödvändigtvis hela svaret, och ramverket ska göra det enkelt att skriva egen SQL, vare sig det är hela satser eller bara ett eget `WHERE` -villkor. En ORM förtjänar sin plats i den rutinmässiga delen av dataåtkomsten; resten är där ett team använder sina SQL-kunskaper.

Så ser det ut i praktiken

Den kostnad som en ORM oftast döljer är antalet frågor den skickar. I SQLAlchemy 2.1 laddas relationer lat som standard: de relaterade objekten hämtas första gången koden rör attributet. Dokumentationen säger det rakt ut: för N inlästa objekt ger åtkomst till deras latladdade attribut N+1 `SELECT` -satser. Exemplet nedan förutsätter att klasserna `Author` och `Book` redan är mappade med en en-till-många-relation, `Author.books` , och går igenom författarna på tre sätt.

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

# Standard: en SELECT för författarna och sedan en till för varje
# författare vars böcker loopen rör, N+1 satser sammanlagt.
authors = session.scalars(select(Author)).all()
for author in authors:
    print(author.name, len(author.books))

# select IN-laddning: en andra SELECT hämtar böckerna för alla
# författare på en gång, med deras primärnycklar i ett IN-villkor.
stmt = select(Author).options(selectinload(Author.books))
for author in session.scalars(stmt).all():
    print(author.name, len(author.books))

# raise-laddning: åtkomst till author.books ger nu ett fel
# i stället för att i det tysta skicka ännu en fråga.
stmt = select(Author).options(raiseload(Author.books))
```

För en-till-många- och många-till-många-samlingar kallar dokumentationen `selectinload()` i allmänhet den bästa strategin. `joinedload()` , som den kallar den mest allsidiga strategin, lägger till en `JOIN` i den ursprungliga satsen, som standard en `LEFT OUTER JOIN` , så att de relaterade raderna kommer i samma resultatmängd. `raiseload()` finns för att skydda mot oönskade laddningar, och jokerteckenformen `raiseload("*")` utvidgar skyddet till de andra relationerna också, inklusive dem på de objekt som satsen faktiskt hämtar.

Django 6.1 ger samma val ett eget namn: fetch modes. När koden rör ett fält som den ursprungliga frågan inte hämtade, till exempel en främmande nyckel, en en-till-en-relation eller ett fält som utelämnats med `defer()` eller `only()`, hämtar standardläget `FETCH_ONE` det för just den instansen, och dokumentationen kallar själv resultatet för N+1-problemet. `FETCH_PEERS`, som är nytt i 6.1, hämtar det saknade fältet för instansen och för alla andra instanser från samma `QuerySet`, utifrån antagandet att kod som behöver ett fält för en rad i regel behöver det för alla. `FETCH_RAISE` utlöser i stället `FieldFetchBlocked`. Ett läge sätts per `QuerySet`, som i `Book.objects.fetch_mode(models.FETCH_PEERS)`, eller som standard för en modell via en egen manager, och Django kopierar det vidare till de relaterade objekt som hämtas. Där `FETCH_PEERS` inte passar eller inte är tillräckligt effektivt hänvisar optimeringsguiden till `select_related()` och `prefetch_related()`.

Företagets egna sidor drar gränsen längre in mot databasen än de flesta. Sidan om teknikstacken behandlar SQL som ett fullvärdigt språk snarare än något en ORM genererar, eftersom det mesta av det prestandaarbete som företaget kallas in för finns där. I arbetet hos NextMariner drev grundaren tanken hela vägen med ett funktionsbaserat skikt i PostgreSQL: varje operation som appen kan utföra är en av 1 275 lagrade funktioner i 34 scheman, och det finns ingen direkt tabellåtkomst från Go över huvud taget. Rollen som API-lagret loggar in som har `EXECUTE` på de funktionerna och `USAGE` på deras scheman och inget annat, så den kan inte röra en tabell direkt. Sidan döljer inte priset: ett nytt fält kräver en ändring i en tabell och dessutom i en funktion, inte bara en ny rad i en fråga.

Var det går fel

En loop skickar en fråga per rad. N+1-mönstret ovan ser ofarligt ut i en liten utvecklingsdatabas och gör en sida långsammare i takt med antalet rader. Bestäm laddningen för varje väg genom koden som spelar roll, med `selectinload()` eller `joinedload()` i SQLAlchemy och `FETCH_PEERS`, `select_related()` eller `prefetch_related()` i Django. Se sedan till att det oplanerade fallet misslyckas högljutt: `raiseload()` i SQLAlchemy, eller `FETCH_RAISE`, som dokumentationen till Django beskriver som ett sätt att förhindra oavsiktliga frågor i prestandakritisk kod.

Ingen läser den SQL som ORM-biblioteket skriver. En ORM skriver SQL som ingen i teamet har valt, och den koden förblir osynlig tills en sida blir långsam. Optimeringsguiden till Django börjar just här: ta reda på vilka frågor en sida kör och vad de kostar, använd `QuerySet.explain()` för att se hur databasen kör ett visst `QuerySet`, och läs `django.db.connection.queries` för att se vad Django skriver

för kodens räkning. Gör läsningen av den SQL-koden till en fast del av granskningen av varje viktig sida eller körning, och kör de långsamma satserna genom databasens egen frågeplan.

Arbete som hör hemma i databasen körs i Python. Att hämta rader bara för att filtrera dem, jämföra två av deras fält eller summera dem flyttar varenda rad över anslutningen för att i applikationskoden göra det som databasen kunde ha gjort där datan redan ligger. Guiden till Django räknar upp alternativen: `filter()` och `exclude()` för att filtrera i databasen, `F()`-uttryck för att jämföra fält i samma modell och `annotate()` för att aggregera i databasen. Skrivningar följer samma regel, med `bulk_create()` och `bulk_update()` i stället för en `save()` per objekt, efter att man har kontrollerat de förbehåll som dokumentationen anger för båda. Vid en mycket stor resultatmängd håller ett `QuerySet` kvar allt det har läst och kan använda mycket minne, och då kan `iterator()` hjälpa.

Den sista delen tvingas genom ORM-biblioteket. Fowlers egen uppskattning lämnar kvar 10 till 20 % av mappningsproblemen, och den sista delen kräver enligt honom alltid noggrant arbete. Att böja en ORM tills den producerar en viss sats gömmer det arbetet i genererad SQL som ingen har valt. Skriv de satserna i SQL och granska dem som all annan kod. Guiden till Django nämner stegen nedåt: uttrycket `RawSQL` lägger till uttrycklig SQL i en fråga, mindre portabelt men kraftfullare, och bortom det kan ett team skriva sin egen SQL för att hämta data eller fylla modeller. Håll ORM-biblioteket till de rutinmässiga posterna, där det sparar mest och kostar minst.

Läs mer

- Catalog of Patterns of Enterprise Application Architecture — Martin Fowlers korta beskrivningar av Active Record, Data Mapper, Unit of Work, Identity Map och Lazy Load.
- ORM Hate — varför objektrelationell mappning är svår, och hur mycket av den ett ramverk kan ta hand om.
- Relationship Loading Techniques — lat laddning, select IN, joined och raise i SQLAlchemy 2.1, och vilken strategi man bör välja.
- Fetch modes — `FETCH_ONE`, `FETCH_PEERS` och `FETCH_RAISE`, som är nya i Django 6.1.
- Database access optimization — profilering, utvärdering av `QuerySet`, arbete i databasen, bulkmetoder och rå SQL.

Ordlista

Välj utifrån era faktiska förutsättningar

Beskriv arbetsflödet, befintliga gränssnitt och användarna som behöver förändringen. Vi kan definiera en testbar leverans och en överlämning som gör att teamet kan underhålla lösningen.

Se vad det kostar Hör av er →

Gör begreppen praktiska

Fråga hur ”Vad är en ORM?” gäller i er organisation. Ett konkret exempel från ert nuvarande system räcker för att börja samtalet.

Se våra tjänster

welcome@engineer.company +45 50 10 61 84 Köpenhamn, Danmark

Ta med dig den här sidan

Visitkort

Omsorgsfullt skapad med kärlek av D.

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

Upphovsrätt © 2025 – nu · Engineer ApS · Alla rättigheter förbehållna.

Utskrivet från https://engineer.company/sv/glossary/orm/