Engineer

ORM

biblioteca que mapea tablas de bases de datos a objetos

Qué es

Un ORM, abreviatura de object-relational mapper, o mapeador objeto-relacional, es una biblioteca que asigna las tablas de una base de datos relacional a clases del código de aplicación y sus filas a objetos. El código pide un autor y recibe un objeto `Author`, cambia uno de sus atributos y lo guarda, y el ORM escribe las sentencias `SELECT`, `INSERT` y `UPDATE` que realizan el trabajo. Los modelos de Django y el ORM de SQLAlchemy, ambos escritos en Python, son dos ejemplos conocidos y los dos en los que se apoya este artículo.

Esa correspondencia es necesaria porque ambos lados organizan los datos de forma distinta. Martin Fowler lo explica claramente en su descripción del patrón Data Mapper: los objetos y las bases de datos relacionales tienen formas diferentes de estructurar datos, y algunas partes de un modelo de objetos, como las colecciones y la herencia, no tienen equivalente en una base de datos relacional. Mover datos entre ambos se convierte en un problema propio, y un ORM ofrece una solución lista para utilizar.

Las partes de esa respuesta recibieron sus nombres en el libro de Fowler Patterns of Enterprise Application Architecture, publicado en 2003, cuyo catálogo de patrones está disponible en línea. Dos patrones describen la forma general. Active Record es un objeto que envuelve una fila de una tabla, gestiona su propio acceso a la base de datos y contiene la lógica de dominio de esos datos; la filosofía de diseño de Django dice que sus modelos siguen ese patrón. Data Mapper es una capa separada que mueve datos entre los objetos y la base de datos manteniéndolos independientes, de modo que los objetos no necesitan saber que existe una base de datos. Alrededor de ellos hay patrones menores. Identity Map se asegura de que cada objeto se cargue una sola vez. Unit of Work mantiene una lista de los objetos que ha cambiado una tarea y escribe los cambios juntos. Lazy Load es un objeto que solo contiene parte de los datos que necesita el código, pero sabe obtener el resto. La documentación de

SQLAlchemy nombra directamente dos de ellos: su `Session` mantiene los objetos cargados en un mapa de identidad, con un solo objeto por clave primaria, y su forma de reunir cambios y volcarlos a la base de datos es el patrón de unidad de trabajo.

Utilice el concepto en una decisión

Aplique «¿Qué es un ORM?» a una decisión real. Podemos ayudarle a relacionar el concepto con el sistema que está construyendo o comprando. La parte que el usuario no ve: dónde se guardan los datos, qué reglas se aplican y cómo pueden comunicarse otros sistemas con ella.

Explore nuestros servicios Póngase en contacto →

Compruebe los supuestos detrás del término

Utilice la explicación y los trabajos enlazados para formular preguntas sobre su proyecto. Podemos examinar los supuestos con usted antes de elegir una herramienta. Diseño y operación de 6 flujos ETL/ELT — 6 flujos ETL en tres motores de bases de datos

Diseño y operación de 6 flujos ETL/ELT Póngase en contacto →

Por qué importa

La primera razón es la cantidad de código que ahorra. En su ensayo ORM Hate de 2012, Fowler considera difícil el problema del mapeo objeto-relacional: implica mantener sincronizadas dos representaciones muy distintas de los mismos datos, y cualquier cambio en un lado debe trasladarse al otro. Su estimación es que un ORM resuelve aproximadamente entre el 80 y el 90 % de los problemas de mapeo, y sostiene que un marco que ahorre a un equipo el 80 % de ese trabajo merece la pena aunque no vaya más allá. Cada tabla que un equipo no mapea a mano es un lugar menos donde leer una columna en el atributo equivocado.

La segunda razón es la coherencia dentro de un programa en ejecución. Con un mapa de identidad, dos consultas que encuentran la misma fila devuelven el mismo objeto, así que un cambio mediante una referencia resulta visible mediante la otra; la documentación de SQLAlchemy lo demuestra cargando dos veces la misma clave primaria y obteniendo el mismo objeto. Con una unidad de trabajo, el ORM reúne las inserciones, actualizaciones y eliminaciones pendientes y, en palabras de la lista de características de SQLAlchemy, las vuelca todas en un lote, de modo que el código de aplicación no tenga que recordar qué objetos tocó. Django elige deliberadamente lo contrario: su filosofía de diseño exige un `save()` explícito en vez de guardar de forma implícita, para que el marco envíe el menor número posible de sentencias SQL.

La tercera razón es dónde se sitúa el límite. Fowler describe un buen ORM como aquel que se encarga del trabajo rutinario y «ofrece pozos de acceso para poder bajar a trabajar con SQL» cuando hace falta, y afirma sin rodeos que cualquiera

que construya una aplicación sobre una base de datos relacional debería saber cómo funciona una. La filosofía de diseño de Django dice lo mismo desde el lado del marco: la API de base de datos debe concebirse como un atajo y no como la respuesta completa, y debe facilitar escribir SQL propio, ya sea una sentencia entera o una sola cláusula `WHERE`. Un ORM se gana su lugar en los accesos rutinarios, que constituyen la mayoría; el resto es donde un equipo emplea sus conocimientos de SQL.

Cómo se aplica en la práctica

El coste que más suele ocultar un ORM es el número de consultas que envía. En SQLAlchemy 2.1, la carga diferida de las relaciones es la opción predeterminada: los objetos relacionados se obtienen la primera vez que el código accede al atributo. La documentación explica la consecuencia: para N objetos cargados, acceder a sus atributos de carga diferida implica N+1 sentencias `SELECT`. El ejemplo siguiente supone clases `Author` y `Book` ya mapeadas, con una relación uno a muchos `Author.books`, y recorre los autores de tres maneras.

```
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))
```

Para colecciones uno a muchos y muchos a muchos, la documentación considera `selectinload()` generalmente la mejor estrategia. `joinedload()`, que denomina la estrategia de uso más general, añade un `JOIN` a la sentencia original, un `LEFT OUTER JOIN` con la configuración predeterminada, de modo que las filas relacionadas llegan en el mismo conjunto de resultados. `raiseload()` existe para

impedir cargas no deseadas, y su forma con comodín, `raiseload("*")`, extiende la protección también a las demás relaciones, incluidas las de los objetos que la sentencia sí carga.

Django 6.1 da a la misma elección un nombre propio: modos de obtención. Cuando el código accede a un campo que no cargó la consulta original, como una clave foránea, una relación uno a uno o un campo omitido con `defer()` u `only()`, el modo predeterminado, `FETCH_ONE`, lo obtiene para esa única instancia, y la propia documentación llama al resultado el problema de las consultas N+1. `FETCH_PEERS`, nuevo en la versión 6.1, obtiene el campo que falta para esa instancia y todas las demás del mismo `QuerySet`, con el razonamiento de que un código que necesita un campo de una fila suele necesitarlo en todas. `FETCH_RAISE` lanza `FieldFetchBlocked` en su lugar. Se establece un modo por `QuerySet`, como en `Book.objects.fetch_mode(models.FETCH_PEERS)`, o como valor predeterminado de un modelo mediante un gestor personalizado, y Django lo copia a los objetos relacionados que obtiene. Cuando `FETCH_PEERS` no es apropiado o suficientemente eficiente, la guía de optimización remite a `select_related()` y `prefetch_related()`.

Las propias páginas de la empresa sitúan el límite más cerca de la base de datos que la mayoría. Su pila tecnológica trata SQL como un lenguaje de primera clase, en vez de como algo que genera un ORM, porque ahí se concentra la mayor parte del trabajo de rendimiento para el que se recurre a la empresa. En su trabajo bajo NextMariner, la fundadora llevó ese enfoque hasta sus últimas consecuencias con una capa de datos PostgreSQL centrada en funciones: cada operación que puede hacer la aplicación es una de 1.275 funciones almacenadas repartidas entre 34 esquemas, y no hay ningún acceso directo a tablas desde Go. El rol con el que inicia sesión la API tiene `EXECUTE` sobre esas funciones y `USAGE` sobre sus esquemas y nada más, por lo que no puede tocar una tabla directamente. La página es clara sobre el precio: un campo nuevo significa cambiar tanto una tabla como una función, no añadir una línea a una consulta.

Dónde surgen los problemas

Un bucle envía una consulta por fila. El patrón N+1 anterior parece inofensivo en una pequeña base de datos de desarrollo y hace que la página se vuelva más lenta a medida que aumenta el número de filas. Decida la carga para cada ruta importante, con `selectinload()` o `joinedload()` en SQLAlchemy y `FETCH_PEERS`, `select_related()` o `prefetch_related()` en Django. Después haga que el caso

imprevisto falle de forma visible: `raiseload()` en SQLAlchemy o `FETCH_RAISE`, que la documentación de Django presenta como una forma de impedir consultas involuntarias en código donde el rendimiento es crítico.

Nadie lee el SQL que escribe el ORM. Un ORM escribe SQL que nadie del equipo eligió, y ese SQL permanece invisible hasta que una página se vuelve lenta. La guía de optimización de Django empieza aquí: averigüe qué consultas ejecuta una página y cuánto cuestan, utilice `QuerySet.explain()` para ver cómo ejecuta la base de datos un `QuerySet` concreto y lea `django.db.connection.queries` para ver lo que escribe Django en nombre del código. Incluya la lectura de ese SQL en la revisión de cada página o tarea importante y analice las sentencias lentas mediante el plan de ejecución de la propia base de datos.

El trabajo que corresponde a la base de datos se ejecuta en Python. Cargar filas solo para filtrarlas, comparar dos de sus campos o sumarlas lleva cada fila por la conexión para hacer en el código de aplicación lo que la base de datos podría hacer donde ya están los datos. La guía de Django enumera las alternativas: `filter()` y `exclude()` para filtrar en la base de datos, expresiones `F()` para comparar campos del mismo modelo y `annotate()` para agregar en la base de datos. Las escrituras siguen la misma regla, con `bulk_create()` y `bulk_update()` en vez de un `save()` por objeto, después de comprobar las salvedades que la documentación enumera para ambos. Para un conjunto de resultados muy grande, un `QuerySet` conserva todo lo que ha leído y puede consumir mucha memoria, e `iterator()` puede ayudar.

La última parte se fuerza a pasar por el ORM. La propia estimación de Fowler deja entre el 10 y el 20 % de los problemas de mapeo, y esa última parte, en sus palabras, siempre requiere un trabajo cuidadoso. Doblegar el ORM hasta que produzca una sentencia concreta oculta ese trabajo dentro de SQL generado que nadie eligió. Escriba esas sentencias en SQL y revíselas como cualquier otro código. La guía de Django nombra los pasos para bajar de nivel: la expresión `RawSQL` añade SQL explícito a una consulta, menos portable pero más potente, y, más allá, un equipo puede escribir su propio SQL para recuperar datos o rellenar modelos. Mantenga el ORM para los registros rutinarios, donde más ahorra y menos cuesta.

Lecturas adicionales

- Catálogo de patrones de arquitectura de aplicaciones empresariales — resúmenes de Martin Fowler sobre Active Record, Data Mapper, Unit of Work, Identity Map y Lazy Load.
- Odio al ORM — por qué el problema del mapeo objeto-relacional es difícil y cuánto puede asumir un marco.
- Técnicas de carga de relaciones — carga diferida, select IN, por combinación y con lanzamiento de errores en SQLAlchemy 2.1, y cuál elegir.
- Modos de obtención — `FETCH_ONE`, `FETCH_PEERS` y `FETCH_RAISE`, nuevos en Django 6.1.
- Optimización del acceso a bases de datos — análisis de rendimiento, evaluación de `QuerySet`, trabajo dentro de la base de datos, métodos masivos y SQL directo.

Glosario

Elija según sus restricciones reales

Describa el flujo de trabajo, las interfaces existentes y los usuarios que necesitan el cambio. Podemos definir una entrega comprobable y un traspaso que permita a su equipo mantener lo entregado.

Vea cuánto cuesta Póngase en contacto →

Haga práctica la terminología

Pregunte cómo se aplica «¿Qué es un ORM?» a su organización. Basta un ejemplo concreto de su sistema actual para iniciar la conversación.

Explore nuestros servicios

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

Llévese esta página

Tarjeta de contacto

Creado con esmero y cariño por D.

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

Derechos de autor © 2025 – actualidad · Engineer ApS · Todos los derechos reservados.

Impreso desde https://engineer.company/es/glossary/orm/