Engineer

pool de conexiones

conexiones abiertas que se prestan, reutilizan y limitan

Qué es

Un pool de conexiones es un conjunto de conexiones abiertas a una base de datos que una aplicación mantiene y vuelve a prestar, en lugar de abrir una conexión nueva para cada trabajo y cerrarla después. El código toma una conexión del pool, ejecuta sus sentencias y la devuelve; la siguiente llamada que necesita una recibe esa misma conexión, ya abierta. La documentación de SQLAlchemy describe esta práctica como «una técnica estándar utilizada para mantener conexiones de larga duración en memoria y reutilizarlas con eficiencia». La misma página asigna al pool otra tarea: gestionar cuántas conexiones puede utilizar la aplicación al mismo tiempo, para que una ráfaga de tráfico espere una conexión libre en lugar de abrir más.

Un pool puede estar en dos lugares. Dentro de la aplicación forma parte de la biblioteca de base de datos: cada `Engine` de SQLAlchemy incluye uno, y Django puede mantener uno en cada proceso para PostgreSQL. Fuera de la aplicación es un programa independiente situado entre los clientes y el servidor de base de datos. PgBouncer cumple esa función para PostgreSQL: las aplicaciones se conectan a él como si fuera la base de datos y les presta conexiones de servidor de sus propios pools. Las dos opciones pueden combinarse, y la referencia de bases de datos de Django tiene una sección sobre la ejecución detrás de un gestor de pools por transacción como PgBouncer.

El motivo para conservar una conexión se aprecia en el servidor. PostgreSQL asigna a cada cliente un proceso propio: un proceso supervisor llamado postmaster escucha las conexiones entrantes, inicia un proceso de servidor nuevo por cada solicitud de conexión y cada cliente se conecta exactamente a uno de esos procesos. Abrir una conexión exige, por tanto, iniciar un proceso en el servidor de base de datos, mientras que un pool paga ese coste una vez y reutiliza el resultado.

Utilice el concepto en una decisión

Aplique «¿Qué es un pool de conexiones?» a una decisión real. Podemos ayudarle a relacionar el concepto con el sistema que está construyendo o comprando. Detectar las consultas y los índices que faltan en un sistema cada año más pesado y corregir lo que realmente importa.

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. 30 días de datos eléctricos en 180 ms — 180 ms para agregar un mes de lecturas de contadores

30 días de datos eléctricos en 180 ms Póngase en contacto →

Por qué importa

El primer motivo es el coste de conectarse. La documentación de Django explica con claridad por qué mantener las conexiones: una conexión persistente evita el coste de establecer una nueva en cada petición HTTP. Sin embargo, su valor predeterminado hace lo contrario. `CONN_MAX_AGE` vale 0, lo que cierra la conexión al final de cada petición; un número positivo de segundos la mantiene abierta para reutilizarla, y `None` la mantiene sin límite. Esa es la forma más simple de reutilización, una conexión por hilo que sobrevive a la petición, y el pool es su forma general.

El segundo motivo es el límite del servidor. `max_connections` de PostgreSQL suele valer 100 de forma predeterminada, puede ser menor si la configuración del kernel no lo permite y solo puede cambiarse al arrancar el servidor. De forma predeterminada, tres de esas plazas se reservan a superusuarios. Aumentar el número tampoco es gratuito: PostgreSQL dimensiona ciertos recursos, entre ellos la memoria compartida, directamente a partir de `max_connections`. Sin un pool, cada worker mantiene su propia conexión, y la documentación de Django explica la consecuencia: como cada hilo conserva la suya, la base de datos debe admitir al menos tantas conexiones simultáneas como hilos de trabajo haya. Un pool convierte esa cantidad en una cifra que el equipo elige y puede sumar.

El tercer motivo es la limpieza. El pool es el lugar por el que pasa cada conexión entre dos usos, así que ahí se limpia y comprueba. Cuando vuelve una conexión, el pool de SQLAlchemy llama a `rollback()`, que libera todo estado transaccional, incluidos los bloqueos, para que el siguiente uso no herede una transacción a medio terminar. En el modo predeterminado de agrupación por sesión de PgBouncer, su `server_reset_query`, cuyo valor predeterminado es `DISCARD ALL`, se ejecuta al liberar una conexión de servidor y antes de que otro cliente la reciba. Además, un pool puede probar una conexión antes de prestarla, algo importante cuando lleva un tiempo inactiva.

Cómo se aplica en la práctica

En SQLAlchemy 2.1, cada `Engine` recibe un `QueuePool` de forma predeterminada, con la excepción de la base de datos en memoria de SQLite. Su `pool_size` es 5 y su `max_overflow` es 10, así que un motor puede mantener hasta 15 conexiones a la vez y una llamada que las encuentre todas ocupadas espera `pool_timeout`, cuyo valor predeterminado es 30 segundos, antes de abandonar. Los ajustes son argumentos con nombre de `create_engine()`:

```
from sqlalchemy import create_engine

# At most ten connections for this engine: ten kept open, none beyond.
engine = create_engine(
    "postgresql+psycopg2://me@localhost/mydb",
    pool_size=10,
    max_overflow=0,
    pool_timeout=10,  # wait ten seconds for a free connection, then give up
    pool_pre_ping=True,  # test each connection as it is checked out
    pool_recycle=3600,  # replace connections that are over an hour old
)
```

Con `pool_pre_ping`, el pool comprueba al entregar una conexión que siga siendo utilizable, mediante el ping del propio controlador o una sentencia equivalente a `SELECT 1`. `pool_recycle` está pensado para servidores que cierran las conexiones que llevan cierto tiempo sin usarse: con 3600, toda conexión abierta hace más de una hora se sustituye la próxima vez que se solicita.

Django 6.1 tiene su propio pool para PostgreSQL. Solo funciona con el controlador psycopg y necesita el paquete `psycopg[pool]`; se activa con `"OPTIONS": {"pool": True}` en los ajustes de la base de datos o con un diccionario de opciones para `ConnectionPool` de psycopg. El pool reutiliza las conexiones entre hilos y cada proceso mantiene un pool separado por cada alias de base de datos. `CONN_MAX_AGE` debe permanecer en 0 cuando el pool está activo; el tiempo de vida de una conexión del pool se establece en su lugar con la opción `max_lifetime`. Con ASGI, la documentación de Django aconseja desactivar las conexiones persistentes y utilizar este pool integrado.

`pool_mode` de PgBouncer decide cuándo puede pasar una conexión de servidor a otro cliente. En el modo de sesión, el predeterminado, el cliente la conserva hasta desconectarse. En el modo de transacción, solo tiene una conexión de servidor mientras dura la transacción, por lo que unas pocas decenas de

conexiones de servidor pueden atender a muchos más clientes. El modo de sentencia libera la conexión después de cada consulta y no permite transacciones de varias sentencias. Cada pool mantiene hasta `default_pool_size` conexiones de servidor por pareja de usuario y base de datos, 20 de forma predeterminada, y PgBouncer acepta hasta `max_client_conn` conexiones de cliente, 100 de forma predeterminada.

Las notas de la empresa sitúan el pool al principio del orden de trabajo. Optimizar la base de datos antes de fragmentarla propone contar las sesiones activas en `pg_stat_activity` y compararlas con los núcleos de CPU: cuando hay muchas veces más sesiones activas que núcleos, el servidor dedica el tiempo a cambiar entre ellas, y un pool de conexiones lo soluciona antes que cualquier hardware. Vender la última entrada exactamente una vez aplica la misma idea a una venta relámpago: cien mil compradores deberían convertirse en unas pocas decenas de conexiones a través de PgBouncer y no en miles de procesos de servidor esperando un solo bloqueo.

Dónde surgen los problemas

Los pools suman más de lo que permite el servidor. Cada pool solo cuenta sus propias conexiones. Con los valores predeterminados de SQLAlchemy, un motor puede llegar a 15, así que doce procesos pueden pedir 180 a un servidor cuyo `max_connections` predeterminado suele ser 100, con tres plazas reservadas a superusuarios, y Django mantiene un pool separado por cada alias de base de datos en cada proceso. Sume el máximo de cada pool que apunte a esa base de datos, en todos los procesos de todas las aplicaciones y todos los trabajos programados, y mantenga la suma por debajo de `max_connections` menos las plazas reservadas. Cuando no quepan, sitúe PgBouncer delante y deje que gestione el número de conexiones de servidor.

Una conexión muere mientras espera en el pool. Una conexión inactiva puede cerrarse en el otro extremo sin que el pool lo sepa y fallar en la primera sentencia del siguiente uso. Active `pool_pre_ping` en SQLAlchemy o `CONN_HEALTH_CHECKS` en Django, que comprueba una conexión reutilizada una vez por petición, y ajuste `pool_recycle` o `max_lifetime` de psycopg por debajo del tiempo tras el que el servidor cierra las conexiones inactivas. Ninguna medida cubre una conexión perdida en mitad de una transacción: la documentación de SQLAlchemy declara expresamente que pre-ping no resuelve ese caso, así que la propia aplicación debe reintentar el trabajo.

Un worker creado mediante fork hereda las conexiones del proceso padre. Un servidor web o un ejecutor de trabajos que crea el motor y después bifurca sus workers deja en cada proceso hijo las mismas conexiones del pool que mantiene el padre. La documentación de SQLAlchemy considera crucial no compartir conexiones de un pool con un proceso creado por fork, porque ambos procesos hablarían por la misma conexión de red. Siga su método recomendado y llame a `engine.dispose(close=False)` en el inicializador de cada hijo, para que empiece con un pool vacío sin tocar las conexiones del padre; las alternativas son `NullPool`, que abre y cierra una conexión en cada uso, o llamar a `engine.dispose()` antes del fork.

El pool por transacción rompe el estado de sesión. En el modo de transacción de PgBouncer, dos transacciones del mismo cliente pueden ejecutarse sobre conexiones de servidor distintas, así que no se garantiza que lo que un cliente configure para su sesión siga disponible en la siguiente transacción. La lista de funciones de PgBouncer marca `SET`, `LISTEN`, los cursores declarados `WITH HOLD`, `PREPARE` a nivel SQL y los bloqueos consultivos de sesión como nunca admitidos en ese modo, y reconoce que, por naturaleza, rompe algunas suposiciones de los clientes sobre el servidor. Mantenga el estado de sesión fuera del código que pasa por un pool por transacción y dirija a un pool en modo de sesión los clientes que lo necesiten. En Django, establezca `DISABLE_SERVER_SIDE_CURSORS` en `True` para una base de datos detrás de un pool por transacción o ejecute esas consultas dentro de `atomic()`.

Lecturas adicionales

- Pools de conexiones — `QueuePool` y sus valores predeterminados, gestión de desconexiones, limpieza al devolver conexiones y pools con fork en SQLAlchemy 2.1.
- Funciones de PgBouncer — los modos de sesión, transacción y sentencia y las funciones de PostgreSQL que admite cada uno.
- Configuración de PgBouncer — `pool_mode`, `default_pool_size`, `max_client_conn` y los tiempos de espera, con sus valores predeterminados.
- Cómo se establecen las conexiones — el modelo de PostgreSQL de un proceso por usuario y el postmaster.
- Conexiones y autenticación — `max_connections` y las plazas de conexión reservadas a superusuarios y otros roles.
- Bases de datos — conexiones persistentes, el pool de conexiones de PostgreSQL y los pools por transacción, en la referencia de Django.

Glosario

Elija según sus restricciones reales

Traiga un conjunto de datos representativo, las preguntas que debe responder y el cuello de botella actual. Podemos acordar cómo medir la calidad, la capacidad de procesamiento y la recuperación.

Vea cuánto cuesta Póngase en contacto →

Haga práctica la terminología

Pregunte cómo se aplica «¿Qué es un pool de conexiones?» 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/connection-pool/