Engineer

Anslutningspool

en uppsättning öppna databasanslutningar för återanvändning

Vad det är

En anslutningspool är en uppsättning öppna databasanslutningar som en applikation håller kvar och lånar ut igen, i stället för att öppna en ny anslutning för varje arbetsuppgift och stänga den efteråt. Koden lånar en anslutning ur poolen, kör sina satser på den och lämnar tillbaka den, och nästa anropare som behöver en får samma anslutning, redan öppen. SQLAlchemys dokumentation beskriver pooling som en standardteknik för att hålla långlivade anslutningar i minnet så att de kan återanvändas effektivt. Samma sida ger poolen ett uppdrag till: den styr hur många anslutningar applikationen får använda samtidigt, så att en plötslig trafiktopp väntar på en ledig anslutning i stället för att öppna fler.

En pool kan ligga på två ställen. Inne i applikationen är den en del av databasbiblioteket: varje `Engine` i SQLAlchemy har en, och Django kan hålla en i varje process för PostgreSQL. Utanför applikationen är den ett eget program som står mellan klienterna och databasservern. PgBouncer är en sådan pooler för PostgreSQL: applikationerna ansluter till den som om den vore databasen, och den lånar ut serveranslutningar ur sina egna pooler. De två går att kombinera, och Djangos databasreferens har ett avsnitt om att köra bakom en transaktionspooler som PgBouncer.

Varför en anslutning är värd att behålla syns på serversidan. PostgreSQL ger varje klient en egen process: en överordnad process, postmaster, lyssnar efter inkommande anslutningar och startar en ny serverprocess för varje begäran om anslutning, och varje klient är ansluten till exakt en av de processerna. Att öppna en anslutning innebär alltså att starta en process på databasservern, och en pool betalar det priset en gång och återanvänder sedan resultatet.

Använd begreppet i ett beslut

Använd ”Vad är en anslutningspool?” i ett konkret beslut. Vi kan knyta begreppet till systemet ni bygger eller köper. Hittar databasfrågorna och de saknade indexen bakom ett system som har blivit tyngre för varje år, och åtgärdar dem som spelar roll.

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. 30 dagars eldata på 180 ms — 180 ms för att aggregera en månads mätaravläsningar

30 dagars eldata på 180 ms Hör av er →

Varför det spelar roll

Det första skälet är kostnaden för att ansluta. Djangos dokumentation säger det rakt ut: en beständig anslutning sparar arbetet med att upprätta en ny i varje HTTP-förfrågan. Standardinställningen gör ändå det motsatta. `CONN_MAX_AGE` är 0, vilket stänger anslutningen i slutet av varje förfrågan; ett positivt antal sekunder håller den öppen för återanvändning, och `None` håller den öppen utan gräns. Det är den enklaste formen av återanvändning, en anslutning per tråd som lever längre än förfrågan, och en pool är den allmänna formen.

Det andra skälet är gränsen på servern. PostgreSQLs `max_connections` är vanligtvis 100 som standard, kan vara lägre om inställningarna i systemkärnan inte klarar det, och kan bara ändras vid serverstart. Tre av platserna är som standard reserverade för superanvändare. Att höja talet är inte heller gratis: PostgreSQL dimensionerar vissa resurser, bland annat delat minne, direkt efter `max_connections`. Utan en pool håller varje arbetsprocess sin egen anslutning, och Djangos dokumentation säger vad det leder till: eftersom varje tråd har sin egen anslutning måste databasen kunna ta emot minst lika många samtidiga anslutningar som det finns arbetstrådar. En pool gör det till ett tal som teamet själv väljer och kan summera.

Det tredje skälet är hygien. En pool är det enda ställe som varje anslutning passerar mellan två lånare, så det är här en anslutning städas och testas. När en anslutning kommer tillbaka anropar SQLAlchemys pool `rollback()` på den, vilket släpper allt transaktionstillstånd, lås inräknade, så att nästa lånare inte ärver en halvfärdig transaktion. I PgBouncers standardläge med sessionspoolning körs dess `server_reset_query`, som standard `DISCARD ALL`, när en serveranslutning släpps och innan en annan klient får den. Och en pool kan testa en anslutning innan den lånas ut, vilket spelar roll för anslutningar som har stått oanvända en tid.

Så ser det ut i praktiken

I SQLAlchemy 2.1 får varje `Engine` en `QueuePool` som standard, med SQLites databas i minnet som undantag. Dess `pool_size` är 5 och dess `max_overflow` är 10, så en engine kan ha upp till 15 anslutningar samtidigt, och en anropare som finner alla upptagna väntar `pool_timeout`, som standard 30 sekunder, innan den ger upp. Inställningarna är nyckelordsargument till `create_engine()`:

```
from sqlalchemy import create_engine

# Högst tio anslutningar för den här enginen: tio hålls öppna, inga utöver dem.
engine = create_engine(
    "postgresql+psycopg2://me@localhost/mydb",
    pool_size=10,
    max_overflow=0,
    pool_timeout=10,  # vänta tio sekunder på en ledig anslutning, ge sedan upp
    pool_pre_ping=True,  # testa varje anslutning när den lånas ut
    pool_recycle=3600,  # byt ut anslutningar som är över en timme gamla
)
```

Med `pool_pre_ping` kontrollerar poolen vid utlåning att anslutningen fortfarande går att använda, via drivrutinens egen ping eller en sats som motsvarar `SELECT 1`. `pool_recycle` är avsett för servrar som stänger anslutningar som har stått oanvända en viss tid: med 3600 byts varje anslutning som har varit öppen i mer än en timme ut vid nästa utlåning.

Django 6.1 har en egen pool för PostgreSQL. Den fungerar bara med drivrutinen psycopg och kräver paketet `psycopg[pool]`, och den slås på med `"OPTIONS": {"pool": True}` i databasinställningarna eller med en ordbok med inställningar för psycopgs `ConnectionPool`. Poolen återanvänder anslutningar mellan trådar, och varje process har en separat pool för varje databasalias. `CONN_MAX_AGE` måste stå kvar på 0 när pooling är påslaget; livslängden för en anslutning i poolen sätts i stället med poolens inställning `max_lifetime`. Under ASGI råder Djangos dokumentation till att stänga av beständiga anslutningar och använda den här inbyggda poolen i stället.

PgBouncers `pool_mode` avgör när en serveranslutning kan gå vidare till en annan klient. I sessionsläge, som är standard, behåller klienten den tills den kopplar ner. I transaktionsläge har en klient en serveranslutning bara medan en transaktion pågår, så några dussin serveranslutningar kan betjäna långt fler klienter. Satsläge släpper anslutningen efter varje fråga och tillåter inte transaktioner som sträcker

sig över flera satser. Varje pool rymmer upp till `default_pool_size` serveranslutningar per par av användare och databas, som standard 20, och PgBouncer tar emot upp till `max_client_conn` klientanslutningar, som standard 100.

Företagets egna anteckningar placerar poolen tidigt i ordningen. Optimera databasen innan du shardar den låter läsaren räkna de aktiva sessionerna i `pg_stat_activity` mot antalet CPU-kärnor: när många gånger fler är aktiva än det finns kärnor ägnar servern sin tid åt att växla mellan dem, och en anslutningspool löser det innan någon hårdvara gör det. Att sälja den sista biljetten exakt en gång använder samma tanke på en blixtrea: hundratusen köpare ska bli några dussin anslutningar genom PgBouncer, inte tusentals serverprocesser som väntar på ett enda lås.

Var det går fel

Poolerna kräver tillsammans fler anslutningar än servern tillåter. Varje pool räknar bara sina egna anslutningar. Med SQLAlchemys standardvärden kan en engine nå 15, så tolv processer kan begära 180 från en server där `max_connections` vanligtvis är 100 som standard, med tre av dem reserverade för superanvändare, och Django har en separat pool för varje databasalias i varje process. Summera det högsta antalet för varje pool som pekar mot databasen, över alla processer i alla applikationer och alla schemalagda jobb, och håll summan under `max_connections` minus de reserverade platserna. När det inte går, ställ PgBouncer framför och låt den styra antalet serveranslutningar.

En anslutning dör medan den väntar i poolen. En anslutning som har stått oanvänd kan stängas i andra änden utan att poolen vet om det, och nästa lånare misslyckas då på sin första sats. Slå på `pool_pre_ping` i SQLAlchemy eller `CONN_HEALTH_CHECKS` i Django, som testar en återanvänd anslutning en gång per förfrågan, och sätt `pool_recycle` eller psycopgs `max_lifetime` lägre än den tid efter vilken servern stänger oanvända anslutningar. Ingen av dem täcker en anslutning som bryts mitt i en transaktion: SQLAlchemys dokumentation säger tydligt att pre-ping inte hanterar det fallet, så applikationen måste själv göra om arbetet.

En arbetsprocess som startas med fork ärver förälderns anslutningar. En webbserver eller en jobbkörare som skapar sin engine och sedan startar sina arbetsprocesser med fork lämnar varje barn med samma poolanslutningar som föräldern. SQLAlchemys dokumentation kallar det kritiskt att poolanslutningar

inte delas med en process som startats med fork, eftersom båda processerna då skulle tala över samma nätverksanslutning. Följ det rekommenderade tillvägagångssättet och anropa `engine.dispose(close=False)` i varje barns initializer, så att barnet börjar med en tom pool och lämnar förälderns anslutningar i fred. Alternativen är `NullPool`, som öppnar och stänger en anslutning vid varje användning, eller att anropa `engine.dispose()` före fork-anropet.

Transaktionspooling förstör sessionstillstånd. I PgBouncers transaktionsläge kan två transaktioner från samma klient köras på olika serveranslutningar, så inget av det en klient ställer in för sin session finns garanterat kvar i dess nästa transaktion. PgBouncers funktionslista markerar `SET`, `LISTEN`, markörer deklarerade med `WITH HOLD`, `PREPARE` på SQL-nivå och advisory locks på sessionsnivå som aldrig stödda i det läget, och sticker inte under stol med att läget till sin natur bryter några av de antaganden klienter gör om servern. Håll sessionstillstånd borta från kod som går genom en transaktionspooler, och skicka de klienter som behöver det till en pool i sessionsläge. I Django ska `DISABLE_SERVER_SIDE_CURSORS` sättas till `True` för en databas bakom en transaktionspooler, annars måste frågorna köras inne i `atomic()`.

Läs mer

- Connection Pooling — `QueuePool` och dess standardvärden, hantering av brutna anslutningar, återställning vid retur och pooler vid fork i SQLAlchemy 2.1.
- PgBouncer features — sessions-, transaktions- och satsläge, och vilka PostgreSQL-funktioner vart och ett av dem stöder.
- PgBouncer configuration — `pool_mode`, `default_pool_size`, `max_client_conn` och tidsgränserna med sina standardvärden.
- How Connections Are Established — PostgreSQLs modell med en process per användare och postmaster-processen.
- Connections and Authentication — `max_connections` och de anslutningsplatser som är reserverade för superanvändare och andra roller.
- Databases — beständiga anslutningar, PostgreSQL-poolen och transaktionspooling i Djangos referens.

Ordlista

Välj utifrån era faktiska förutsättningar

Ta med ett representativt dataset, frågorna det ska besvara och dagens flaskhals. Vi kan avtala mått på kvalitet, genomströmning och återställning.

Se vad det kostar Hör av er →

Gör begreppen praktiska

Fråga hur ”Vad är en anslutningspool?” 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/connection-pool/