Пояснительная записка

Данис Тазеев

Оглавление

Сборка и запуск

Я собирал Maven-ом 3.3.3. Точно не скажу, но на всякий случай потребую от Вас использовать Maven 3+. В командной строке набираем просто `mvn`. Баз параметров:

```
$ mvn
```

По умолчанию выполняются фазы `clean package`.

Результат сборки — `target/trial-0.1.0.jar`. Это исполняемый JAR. Рекомендую запустить его из корня проекта, а не из `target`:

```
$ java -jar target/trial-0.1.0.jar
```

Программа автоматически создаст базу данных и лог-файл. База данных — каталог `database` в working directory; лог-файл — `app.log` в той же working directory.

Есть специальный режим, в котором имитируются задержки выполнения запросов к БД. Подробнее см.Имитация задержек.

Требуется JRE 8.

СУБД

Евгений, Вы говорили, что используете PostgreSQL, Miscrosoft SQL Server и Oracle RDBMS. С PostgreSQL я не знаком, а Microsoft SQL Server бесплатно работает только 6 месяцев. Для задания мне бы, конечно, хватило, но через 6 месяцев всё равно пришлось бы от него отказаться, да и MVCC он не поддерживает. Остался Oracle Database 11g Express Edition (Oracle XE). Это полноценный сервер БД, только бесплатный (при условии, что его

использование не приносит дохода) и имеющий несколько ограничений: например, размер tablespace system ограничен 1 GiB. И других tablespace создать нельзя. Кажется :). Ну, наверное, какие-нибудь ещё ограничения.

Потом я посмотрел на скрипты создания схемы БД:

- нужно придумать имя пользователя (схемы), которое бы не конфликтовало с уже существующими именами в вашей БД;
- нужно дать квоту в правильном tablespace, и тут запросто можно не угадать: конфигурация вашего сервера Oracle может не позволять давать unlimited квоту, может не позволять давать квоту на tablespace system, значит, нужно представлять, каков допустимый размер квоты на tablespace, и представлять, каково могло бы быть имя этого tablespace;
- нужно дать разрешения пользователю (grant permissions), а для этого нужно представлять, как сконфигурирован ваш Oracle: каким пользователем нужно подключиться вначале, чтобы создать другого и дать созданному права и всё остальное.

Это всё означает, что для развёртывания схемы в БД придётся просить Вас вручную выполнить скрипт. Может быть, вам пришлось бы исправить предварительно имя пользователя (схемы), имя tablespace и размер квоты. И бог знает, что ещё может пойти не так на вашей инсталляции Oracle. (Не потому, что вы плохие или Oracle у вас плохой, а потому что слишком много условий.)

Потом подумал, Oracle — это отдельный процесс, следовательно — сетевое взаимодействие, следовательно, алгоритмы должны переживать потерю соединения и уметь его восстанавливать. Вспомнил, что толковой документации по параметрам конфигурирования JDBC дравера для Oracle нету. Значит, нужно поэкспериментировать, имитируя потерю и восстановление соединения. На это ушло бы минимум дня два. Минимум! Я уже делал это для Deutsche Bank. Знание, конечно же, осталось у них.

И тут меня озарило: а почему бы не воспользоваться легковесной СУБД? А почему бы не запустить СУБД прямо внутри процесса приложения? Почему бы не использовать такую СУБД, которая эксплуатируется вообще без каких либо усилий? Особенно по сравнению с Oracle. Кстати, у меня на компьютере Oracle XE установлен и работоспособен, если что :).

Словом, я выбрал H2. Решил, что не буду запускать отдельным процессом, а запущу прямо внутри приложения. Никаких имён пользователя, конфликтующих с уже существующими, никаких квот, никаких tablespace, никакого «выполните, пожалуйста, вот этот скрипт». Никакого конфигурирования JDBC драйвера, никаких потерь и восстановлений соединения, никаких grant permissions. По сравнению с Oracle, эксплуатация H2 не стоит *ничего*. Вот так я упростил себе задачу.

Да, SQL-скрипт создания схемы БД и первичного её наполнения всё равно есть: `src/main/resources/da/create.sql`. При первом запуске программа автоматически создаёт БД — каталог `database` в рабочем каталоге (working directory), из которого запускается приложение. За эти действия отвечает класс `DbInitializer`.

Транзакции и соединения

Евгений, очень признателен, что задание не требует транзакций, распространяющихся более чем на один запрос. Это позволило использовать режим auto-commit и не управлять транзакциями вручную: сторонними библиотеками Вы всё равно пользоваться запретили.

Кроме того, задание таково, что не требует и вложенных транзакций тоже. Следовательно, можно обойтись единственным соединением с БД, без их пула: запустил программу, подключился к БД одним соединением, и используй его, последовательно передавая от одного экземпляра `DAO` к другому. Именно так и ведёт себя базовый абстрактный класс всех Data Access Objects (DAO) — `da.DAO`.

Говоря «экземпляр `DAO`», я, конечно же, имею в виду экземпляры конкретных наследников `DAO`, так как сам `DAO` абстрактный.

Соединение — это такой объект, который нельзя использовать конкурентно. Из-за того, что текущая транзакция привязана к соединению: не выходит за его рамки и не живёт дольше. Поэтому нельзя запустить ещё один запрос на соединении, пока предыдущий не выполнен. Поэтому, имея единственное соединение на приложение, пользоваться им можно только последовательно: от одной транзакции к другой.

Единственное соединение, пул соединений… Что ещё? Соединение устанавливается в начале каждой транзакции и закрывается при её завершении. Каждый раз, физически устанавливается и закрывается. Похоже на пул, только честно открывается и закрывается. И нагружает СУБД.

А вообще, решение с единственным соединением я реализовал впервые. Точно так же, как впервые написал приложение на Swing. Оглядываясь назад и забегая вперёд, у меня здорово получилось!

Рассуждал так: раз приложение однопользовательское, значит, и доступ к БД строго последовательный — запрос за запросом. Это было интуитивным решением. Раз приложение «настольное», то экземпляров таких приложений столько, сколько «столов» :). Следовательно, количество соединений к СУБД — количество запущенных приложений, умноженное на количество соединений от каждого из них. Честно говоря, не знаю, при каком количестве соединений наступает *практический* предел для СУБД.

Я привык, что серверы приложений используют несколько десятков соединений. Не более. В лучшем случае серверы приложений объединены в кластер из двух-трех серверов. Оценочно, в таких конфигурациях количество соединений к СУБД не превышает двух сотен.

Требования к «дружелюбности»

Использование СУБД in-process *концептуально* не отличается от использования удалённого сервера БД: любой запрос к БД (внутри ли процесса приложения или сетевой) является блокирующей операцией. И эта операция выполняется много дольше, чем обработка какого бы ни было UI-события (`EventObject`) какой бы ни было сложной иерархией UI компонент. Это во-первых. Во-вторых:

1. Swing — однопоточная библиотека, как и все остальные UI библиотеки, полагаю. Даже как JavaScript в браузере. А также event-driven. Следовательно, чтобы приложение

оставалось отзывчивым, блокировать тот самый рабочий поток обработки событий (`EventDispatchThread` — EDT) никак нельзя. И отзывчивым приложение быть *обязано*. Этот принцип реализован во всех компонентах: `CheckInOutPanel`, `DailyReportPanel`, `HistoryPanel`.

2. Пользователь UI-приложения должен иметь возможность не дожидаться завершения этих самых блокирующих операций. Попросил он, скажем, «историю для указанного сотрудника за заданный период», UI показал ему «песочные часы» — так вот, пользователь должен иметь возможность плюнуть и уйти в другое место приложения, выбрасывая результаты своего запроса, пока те даже ещё не созрели (prematurely). Другими словами, операция отказа от своих намерений *обязана* быть реализованной в UI приложении. Этот принцип реализован в компонентах `DailyReportPanel` и `HistoryPanel`.
3. А ещё UI приложение должно изменять свой вид, закрывая или открывая функции по мере поступления данных. Например, пока список сотрудников не подгрузился, приложение не замрёт в ожидании, а выключит combo box, который как раз и должен показать этот самый список. Выключит кнопку «Показать историю для указанного сотрудника за заданный период», потому что пока сотрудники не подгрузились, указывать просто некого. Этот принцип реализован в компонентах `CheckInOutPanel` и `HistoryPanel`.
4. А вот кнопку «Показать отчёт на заданный день» не выключит, так как параметры запроса — «заданный день» — можно сформировать и без подгруженного списка сотрудников. Параметры сформировать можно, а результат без подгруженного списка *показать* нельзя Это тонкий и очень важный момент. UI приложение должно различать такие тонкости. Этот принцип реализован в компоненте `DailyReportPanel`.

Лично я считаю, что работа не выполнена *совсем*, если приложение перестаёт реагировать на действия пользователя. Даже местами. Даже иногда. Если это так — работа не выполнена. И не важно, сколько строк кода написано и сколько времени потрачено.

Это не означает, что результаты запросов должны выскакивать мгновенно. Это означает, что «песочные часы» не являются препятствием для использования приложения.

Имея в виду сказанное, становится понятно, что *концептуально* СУБД, запущенная внутри приложения, ничего не меняет — всё равно запросы блокирующие, а UI останься отзывчивым.

Команды

Должен пояснить, когда я только получил Ваше задание, Евгений, интуитивно у меня были смутные ожидания от будущего приложения. Было понятно, что оно должно быть отзывчивым. Я ощущал себя слепым котёнком, ищущим способ, приём, шаблон, который позволил бы реализовать отзывчивость. И лишь сейчас, в конце, те смутные общие ожидания отзывчивости кристаллизовались в гораздо более осмысленные принципы. А также были найдены шаблоны построения компонент.

В какой-то момент знакомства со Swing, критическая масса набралась, преобразовавшись в качество, и стало понятно, что виды (то, на чём располагаются элементы управления, и которые отображают результаты) нужно проектировать в соответствии со стандартом JavaBeans (если это стандарт). Один вид — один класс по правилам JavaBeans. Вид генерирует события. Виды пользуются моделями, модели тоже компоненты JavaBeans, тоже

генерируют события. Виды подписываются на события моделей. Виды могут протаскивать в свой API операции моделей; могут объявлять в своём API операции, вызывающие операции моделей, обновляя их.

Стало понятно, что можно выделить конечное множество укрупнённых операций, модифицирующих вид UI компоненты. Их можно назвать атомарными. Эти операции естественным образом выделяются в отдельные методы. Можно сказать, что это команды, приказы, которые можно отдать UI компоненте, в ответ на которые компонента меняет свой вид::

- переведи себя в состояние, отражающее ожидание (`setWaitingState()`);
- переведи себя в состояние, отражающее готовность (`setReadyState()`);
- о, размер элемента управления изменился — перерасположи свои компоненты и отцентрируйся на экране (`packAndCenterFrameOf()`);
- и т. д.

Фоновые задания

Есть команды, запускающие фоновый процесс: `launchReportFetcher()`, `launchHistoryFetcher()`… Так как эти операции запускают фоновый процесс, то часть формы выключается — та часть, что как раз ответственна за запуск этого процесса, включаются «песочные часы». Как только фоновый процесс завершается, его результаты отображаются на форме, а выключенная часть формы снова включается, «песочные часы» превращаются в стрелочку. Важно не забыть дать пользователю возможность наплевать на только что отданный запрос и заранее выбросить его результаты, ещё до того, как они готовы (prematurely) — кнопка «Назад» на формах `DailyReportPanel` и `HistoryPanel`. Это отмена *ожидания результатов* запроса.

Кстати, `DailyReportPanel` отличается от своих собратьев тем, что он умеет ждать результаты сразу двух запросов: списка сотрудников и посещаемости. Лишь получив результаты обоих (в любой последовательности и любым интервалом между), вид считает, что получил один совокупный, и только тогда включает обратно элементы управления и отображает совокупный результат.

Для отмены запроса не всегда требуется кнопка. Например, когда пользователь бегает по ниспадающему списку сотрудников вида `CheckInOutPanel` с помощью клавиатуры (стрелки вверх и вниз), форма для каждого сотрудника должна включить только одну из кнопок «Пришёл» или «Ушёл», а вторую выключить. Чтобы выяснить, какую именно, вид обращается к БД. Пока запрос выполняется, выключены обе. В то же время, пока запрос выполняется, пользователь может прокрутить список дальше. Он не должен ждать *результатов* запроса по каждому из сотрудников, по которым он пробежался в списке. Смена выбора в ниспадающем списке до того, как получены результаты запроса по предыдущему выбору, заранее выбрасывает эти результаты и запускает новый запрос.

В этом механизме есть тонкость. Фоновые запросы — это такие задания, которые попадают в спец. очередь. Рано или поздно какой-нибудь сторонний поток подхватывает задание из очереди и полностью его выполняет. У задания есть операция `cancel()`. Так вот если `cancel()` успела выполниться до того, как задание было подхвачено потоком, то попытки его выполнить даже не будет предпринято. А вот если задание уже находится на выполнении, то

оно будет выполнено. (Отмена уже выполняющихся в СУБД запросов в приложении не реализована.)

Другой поток ожидает результатов — это EDT. Так вот если задание было отменено, то EDT узнаёт об отмене сразу, вне зависимости от того, взято ли уже задание на выполнение и будет выполнено или ещё не взято и выполняться даже и не будет.

Это означает, что когда пользователь быстро бежит по ниспадающему списку, скорость генерации фоновых заданий превышает скорость их выполнения. Но так как каждое предыдущее отменяется, то для большей части отданных фоновых заданий так и не будет предпринята попытка их выполнить. Некоторые всё же будут выполнены, не смотря на то, что результатов их выполнения уже давно никто не ждёт. А «выигрывает» последний — именно его ответ и приведёт к включению нужной кнопки «Пришёл» или «Ушёл».

Имитация задержек

Приложение можно запустить в специальном режиме, когда оно имитирует большие задержки в выполнении запросов. Можно увидеть пустой список сотрудников на форме «Отметиться» или «История», пока он не подгрузится. Можно зайти на форму «Отметиться», отметиться и снова тут же туда зайти. Будет видно, что предыдущий запрос, сохраняющий время отмечания, всё ещё выполняется, и отменить его нельзя.

Чтобы запустить приложение в таком режиме, выполните в корневом каталоге проекта:

```
$ java -jar target/trial-0.1.0.jar --max-delay-secs <d>
```

Параметр `<d>` — неотрицательное `int`, задаёт максимальную задержку в секундах перед каждым SQL-запросом. Конкретная задержка для каждого выполняемого запроса — случайная величина в интервале [0, `<d>`] секунд.

Рекомендую 3 секунды.

Охрана соединения

Каждый SQL-запрос — блокирующая операция. Для каждого создаётся задание, помещается в очередь, откуда их вычёрпывают несколько фоновых потоков. Получается, что потоков, готовых выполнять SQL-запросы, несколько, а соединение с БД одно, и его запрещено использовать конкурентно. Как это обеспечивается?

SQL-запросы выполняются в методах конкретных наследников абстрактного класса `DAO`. Чтобы выполнить метод какого-нибудь DAO с SQL-запросом, нужно создать экземпляр этого DAO. Базовый класс `DAO` является `AutoCloseable`, а значит и все его наследники тоже. Это позволяет применять удобную синтаксическую конструкцию `try`-with-resource:

```
try (EmployeeDAO dao = new EmployeeDAO()) {
     return dao.selectEmployees();
}
```

На выходе из `try` блока у объекта `dao` вызывается операция `close()`, если `dao != null`. Если `dao == null` (из-за исключения, возникшего в конструкторе объекта `dao`), ничего не делается.

Каждый конкретный класс DAO — наследник `DAO`, который хранит внутри себя то самое единственное соединение, которым последовательно пользуются всё однозапросные транзакции приложения (а болеезапросных просто нет). Так как вложенных транзакций в приложении тоже нет, то единственного соединения точно достаточно.

Одного соединения точно не хватало бы, если бы были вложенные транзакции. Причём соединений потребовалось бы столько, каков уровень вложенности транзакций. Из-за того, что любая транзакция привязана ровно к одному соединению. И с каждым соединением связана какая-либо одна транзакция. А раз вложенных транзакций нет, то одного соединения точно достаточно.

Получается, что каждый SQL-запрос, выполняемый любым из методов конкретного DAO, обрамлён созданием экземпляра этого DAO и вызовом `close()` на нём. Следовательно, базовый класс `DAO` может блокировать выполнение своего конструктора, пока есть хоть один экземпляр конкретного наследника, у которого ещё не вызван `close()`. Итак, конструктор `DAO` ждёт (`wait()`), пока не будет вызван `close()` у экземпляра, созданного в прошлый вызов конструктора `DAO`. Метод `close()`, в свою очередь, уведомляет (`notify()`) конструктор о том, что предыдущий экземпляр закрыт и больше не воспользуется соединением. Тогда конструктор перестаёт блокировать свой поток выполнения, захватывает соединение и создаёт очередной экземпляр.

Если конструктор `DAO` основан на `wait()`, почему же тогда он не объявляет `InterruptedException` в своей сигнатуре? Дело в том, что у меня есть мотивированное правило: не пользоваться прерываниями потоков — методом `interrupt()` у экземпляров класса `Thread`. Правило — не значит закон. Технически я могу воспользоваться механизмом прерываний, если встречу задачу, где этот механизм придётся как нельзя кстати.

На данный момент в приложении `interrupt()` никогда не вызывается. Если же он всё же однажды будет вызван, или же он будет вызван изнутри стороннего кода, и будет прерван как раз тот поток, который создаёт экземпляр DAO, пока тот ожидает освобождения соединения, то флаг прерывания сохраняется, конструктор всё равно продолжает блокировать поток, ожидая соединения, а захватив его, вновь взводит флаг прерывания на выполняющем потоке. Это тонкий и error-prone момент.

Да, получается, конструктор игнорирует просьбу какого-то другого потока прервать своё выполнение. В противном случае в сигнатуру пришлось бы вводить checked `InterruptedException`, которое прокралось бы в сигнатуры конструкторов конкретных наследников, и каждый раз, создавая экземпляр DAO, пришлось бы, чертыхаясь, писать код, как-то об этом исключении заботящийся. Но зачем это делать, если я всё равно не пользуюсь прерываниями? И делаю это крайне осознанно и мотивированно? Готов рассказать при встрече.

Если кратко — а зачем? Разве можно с помощью прерывания прервать выполнение SQL-запроса? Или прервать поток, заблокированный в операции `write()` у `OutputStream`, полученного у `Socket`? А разве можно прервать резолюцию DNS-имени по IP-адресу в таком,

казалось бы, невинном методе, как `InetAddress.toString()`? Нет, нет, нет. И это далеко не полный список. А что делать с недобросовестным кодом, который и не прерывается — так же, как и конструктор `DAO`, — да ещё и флажок прерывания проглатывает — в отличие от конструктора `DAO`?

Словом, настоятельно и ответственно рекомендую всем, кто пишет на Java, не пользоваться прерываниями потоков. Никогда.