|  |
| --- |
| ИнтерТраст |
| Подсистема миграции структур и данных |
|  |

|  |
| --- |
| Митавский Д. В. |

Содержание

[Типы миграции 2](#_Toc410068391)

[Автоматические миграции 3](#_Toc410068392)

[Скриптовые миграции 5](#_Toc410068393)

[Инкрементальная миграция 7](#_Toc410068394)

[Алгоритм запуска миграций 7](#_Toc410068395)

[Разработчикам миграционных сценариев 9](#_Toc410068396)

# Типы миграции

Миграции разделяются на несколько категорий:

1. Структуры данных (мета-данных)

В данный тип входят изменение набора полей, ограничений, индексов доменных объектов – состава, имён, типов и т.п.

В свою очередь миграции структуры данных разделяются на

* 1. Автоматические. Выполняются платформой при смене конфигурации.
     1. Безопасные - те, которые могут быть произведены кодом автоматически при смене конфигурации таким образом, что никогда не произойдёт внештатной ситуации.
     2. Небезопасные – те, при выполнении которых могут возникнуть неоднозначности и ошибки. Например, добавление нового уникального ключа породит исключение, если данные в хранилище не уникальны.
  2. Скриптовые. Выполняются на основе конфигурации предопределёнными командами или конкретным кодом, на который команда делегирует.

1. Данных

При этом изменяются данные в хранилище – в силу изменений архитектуры, логики работы, исправления ошибок или, например, при изменении типа данных. Такие являются исключительно скриптовыми

Скриптовые миграции выполняться до и после автоматической (и до, и после - тоже может). Конфигурируемые скрипты могут работать как в понятиях доменных объектов (предпочтительный, но не всё решающий вариант), так и в "родных" понятиях хранилища.

При накладывании новых ограничений на данные (not-null, уникальные ключи) в подавляющем большинстве случаев необходимо корректно обработать данные, уже находящиеся в этой колонке. Рассчитывать на то, что таблица пустая или в ней одна запись, нельзя – распределение данных в разных экземплярах системы может быть разным.

# Автоматические миграции

**Новое требование (требуется для того, чтобы скриптовые миграции могли произвести неподдерживаемые автоматикой изменения перед запуском автоматической миграции):**

При обнаружении любых «невыполнимых» сценариев (смена типа поля, уменьшение размера колонки, добавление not null) нужно проверить соответствие этого сценария содержимому СУБД. Если содержимое колонки, индекса, ограничения в базе соответствует конфигурации, то данное несоответствие игнорируется.

О каждой конкретной атомарной операции (например, добавление колонки) в логе должны быть уведомления (если SQL-Trace это покрывает, то его достаточно).

Поддерживаются следующие сценарии безопасных автоматических миграций структуры доменных объектов (красным отмечены новые требования):

1. Добавление новых типов доменных объектов
2. Добавление полей в тип доменный объект
3. Произвольное изменение индексов
4. Удаление полей из доменных объектов. Колонки при этом остаются в таблицах, ограничения not-null снимаются. Если впоследствии это поле добавится вновь, то, в случае полного совпадения с колонкой автоматическая миграция пройдёт нормально (схоже с «поддержкой «невыполнимых» сценариев»).
5. Удаление типов доменных объектов, если на них не ссылаются другие ДО. Причём если в конфигурации были удалены колонки, указывающие на удалённый тип, и сам тип удалён, то такой сценарий тоже поддерживается. Другими словами, если конфигурация без типа целостна, то тип можно «удалять». Так проще реализовать алгоритмически.
6. Если в конфигурации типа или поля нет, и в базе их уже тоже нет, то ситуацию считать корректной и игнорировать (см. «поддержку «невыполнимых» сценариев»).
7. Удаление not-null ограничения
8. Увеличение размера поля
9. Удаление уникального ключа
10. Добавление полей к уникальному ключу
11. Удаление внешнего ключа (замена типа ссылки на «звёздочку»)
12. Замена типа в ссылке на его супертип
13. Добавление поля, колонка для которого уже есть в базе, в случае полного совпадения типа и ограничений
14. Добавление типа доменного объекта, для которого есть таблица в базе, частично удовлетворяющая данному типу по составу колонок. Таблицу следует «преобразовать» в описание типа объекта и рассматривать его как предыдущий вариант конфигурации. Если из этого типа можно автоматически мигрировать в тип в текущей версии конфигурации – сделать это.

Небезопасные автоматические миграции, не поддерживаются.

Операции над данными должны быть осознанными и явно описаны скриптами, во избежание их потери. Кроме того, подобный подход позволяет надеяться на возможность удобной реализации инкрементальной миграции (см. далее).

После реального проведения любой скриптовой миграции необходимо проверить полное соответствие конфигурации базе данных (с допущением, что «лишние» колонки и таблицы в базе могут быть). В противном случае, миграция может привести к трудоновыявимым проблемам, которые могут дать о себе знать лишь после перехода заказчика на новую версию. Например, миграционный скрипт может удалить колонку, которая по ошибке разработчика останется в конфигурации. Конфигурация останется прежней и поэтому не проверится, а при попытке сохранения возникнут исключения.

# Скриптовые миграции

Скриптовые миграции определяются в конфигурационном XML.

Пример такого XML:

<**migration-script sequence-number="4"**>  
 <**before-auto-migration**>  
 <**rename-field type="type1"**>  
 <**field name="field1" new-name="field2"**/>  
 <**field name="field2" new-name="field3"**/>  
 </**rename-field**>  
 <**change-field-class type="type1"**>  
 <**field name="field4"**/>  
 <**field name="field5"**/>  
 </**change-field-class**>  
 <**execute component-name="migration\_000001"**/>  
 <**native-command**>  
 <![CDATA[  
 alter table ...  
 ]]>  
 </**native-command**>

<**create-unique-key type="type1"**>  
 <**field name="f1"**/>  
 <**field name="f2"**/>  
 </**create-unique-key**>

<**make-not-null type="type1"**>  
 <**field name="f1"**/>  
 <**field name="f2"**/>  
 </**make-not-null**>

</**before-auto-migration**>  
 <**after-auto-migration**>  
 <**delete-types**>  
 <**type name="type1"**/>  
 <**type name="type2"**/>  
 </**delete-fields**>  
 <**delete-fields type="type"**>  
 <**field name="field1"**/>  
 <**field name="field2"**/>  
 </**delete-fields**>  
 </**after-auto-migration**>  
</**migration-script**>

Описание приводится в следующей таблице.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Тэг | Атрибут/  Тип | Описание | Мн. | По умол-  ча-нию | Обя-  за-  тель-  ность |
| **migration-script** |  | Описание миграционного сценария | Да |  | Нет |
|  | sequence-number /  **integer** | Глобально-уникальный порядковый номер сценария. На уникальность и последовательность (все номера должны идти по порядку, без промежутков) должна проводиться логическая валидация |  |  | Да |
| **before-auto-migration** |  | Описание последовательности сценариев, запускаемых до автоматической миграции | Нет |  | Нет |
| **after-auto-migration** |  | Описание последовательности сценариев, запускаемых после автоматической миграции. Набор вложенных тэгов (непосредственно сценарии) аналогичен тэгу **before-auto-migration** | Нет |  | Нет |
| **rename-field** |  | Сценарий переименования поля (колонки в таблице). Содержит в себе одно или более описание полей, которые будут переименованы. | Да |  | Нет |
|  | type | Тип доменного объекта, в котором происходит переименование |  |  | Да |
| rename-field**.**  **field** |  | Определение переименуемого поля |  |  | Нет |
|  | name | Текущее название поля |  |  | Да |
|  | new-name | Новое название поля |  |  | Да |
| **change-field-class** |  | Сценарий прозрачной трансформации типа поля. Текущий тип (определяется по предыдущей версии конфигурации, или непосредственно по колонке в базе) трансформируется в тот, который определён в текущей версии конфигурации. Поддерживаются следующие прозрачные трансформации:  string 🡨🡪 string c обрезанием,  string 🡨🡪 text с обрезанием,  string 🡨🡪 boolean,  long 🡨🡪 decimal с потерей десятичных долей | Да |  | Нет |
| change-field-class.**field** |  | Определение поля, меняющего тип | Да |  | Да |
|  | name | Имя поля, меняющего тип |  |  |  |
| **execute** |  | Сценарий, осуществляемый именованным компонентом системы. Он должен реализовывать интерфейс Migrator и его метод execute() | Да |  | Нет |
|  | component-name | Название компонента |  |  | Да |
| **native-command** |  | Сценарий, исполняющий скрипт, написанный на языке хранилища (пока только SQL/PLSQL) | Да |  | Нет |
| **delete-types** |  | Сценарий физического удаления типов | Да |  | Нет |
| delete-types.**type** |  | Определение типа удаляемого объекта | Да |  | Да |
|  | name | Название типа |  |  | Да |
| **delete-fields** |  | Сценарий физического удаления колонок | Да |  | Нет |
| delete-fields**.field** |  | Определение типа удаляемого поля | Да |  | Да |
|  | name | Название поля |  |  |  |
| **create-unique-key** |  | Сценарий создания уникального ключа | Да |  | Нет |
| create-unique-key**.field** |  | Описание поля, входящего в уникальный ключ | Да |  | Да |
|  | name | Название поля, входящего в уникальный ключ | Да |  | Да |
| **make-not-null** |  | Сценарий создания ограничения «not null» | Да |  | Нет |
| make-not-null**.field** |  | Описание поля, на которое накладывается ограничение | Да |  | Да |
|  | name | Название поля | Да |  | Да |

# Инкрементальная миграция

Часто случается так, что заказчик переходит на версию системы, версия которой существенно старше текущей, и между ними существуют версии системы, которые требуют неавтоматической миграции (далее в этом параграфе – просто миграции). Скажем, заказчик использует версию 1, переходит на версию 5, а версии 3 и 4 требуют миграции. Казалось бы, можно сразу перейти на версию 5, однако наличие автоматической миграции и отсутствие в текущей сборке системы информации о конфигурациях версии 2, 3, 4 мешает провести миграцию автоматически, так как в процесс вмешивается автоматическая миграция, а скрипты запускаются до и после неё. Интуитивно я догадываюсь, что, запустив по порядку скрипты, выполняющиеся до автоматической миграции, потом автоматическую миграцию (может быть, только безопасную), а потом скрипты «после миграции», мы достигнем нужного результата. Однако, у меня пока не было времени более-менее строго это доказать, подобрать сценарий-опровержение или подобрать ограничения, в которых это гарантировано бы работало. Реализована миграция с sequence=”x” до sequence=”y” должна быть именно таким образом – сначала все «миграции до», потом «автоматика», потом «миграции после» в порядке их следования.

Гарантированный же способ инкрементальной миграции на данный момент – это последовательная установка версий системы, в которых появлялись сценарии миграции. Для вышеупомянутого примера – это установка сначала версии 3, потом версии 4, и потом 5.

# Алгоритм запуска миграций

Для упрощения восприятия, представим себе следующую временную шкалу

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| К1 | К2 | К3 | К4 | К5 | К6 | К7 | К8 | К9 | К10 |
|  | А  М1 | А  М2 | А |  | М3 |  | А  М4 | А  М5 |  |
| С1 | С2 | С3 | С4 | --- | С5 | --- | С6 | С7 | --- |

**К** обозначает версию конфигурации. Явного версионирования в платформе нет, новую версию конфигурация приобретает с любым её изменением.

**C** – версия структуры базы данных. Опять же, явное версионирование отсутствует. Если структура меняется хоть как-то это считается новой версией.

**A** – процесс автоматической миграции

**М** – скриптовая миграция

Например, при переходе K2 с K1 происходит скриптовая и автоматическая миграции, в результате база данных изменяет свою структуру с С1 до С2.

Итак, непосредственно алгоритм

* При установке системы «с нуля» происходит инициализация базы, **никакие** миграции не запускаются. В доменный объект с информацией о запущенных миграциях записывается максимальный порядковый миграции (как будто бы, она уже произошла)
* При повторном запуске системы (той же или следующей версии конфигурации) происходит проверка наличия миграций, которые ещё не были запущены (тех, порядковый номер которых больше сохранённого в хранилище). Если таковые найдены, то они запускаются:

1. Все скрипты «до автоматической миграции» в порядке их номеров миграционных сценариев
2. Автоматическая миграция
3. Все скрипты «до автоматической миграции» в порядке их номеров миграционных сценариев

Примеры.

1. Устанавливается «с нуля» система версии K1.
2. Миграционных сценариев в конфигурации нет, соответственно запускается автоматическая «миграция» (в данном случае инициализация).
3. Система обновляется до версии K2. Система не «новая», запущенных миграций не было (в логе миграций пусто) в конфигурации присутствует новая миграция M1, запускается миграция М1 с автоматической.
4. Далее система обновляется до версии К10. Система не «новая», в конфигурации есть новые миграции M2 – M5 (М1 уже была запущена, об этом есть информация в логе), запускаются M2-ДО 🡪 М3-ДО🡪 М4-ДО🡪 M5-ДО🡪Автоматическая 🡪 M2-ПОСЛЕ 🡪 М3- ПОСЛЕ 🡪 М4- ПОСЛЕ 🡪 M5- ПОСЛЕ
5. Устанавливается «с нуля» система версии K3.
6. В конфигурации есть миграционные сценарии М1 и М2. Но система устанавливается «с нуля», поэтому миграционные сценарии не исполняются, происходит лишь инициализация хранилища и запись в лог миграций о том, что М1 и М2 уже «были запущены».
7. Система обновляется до версии K4. Система не «новая», запущенная последняя миграция была М2 в конфигурации новых миграций нет. Запускается автоматическая миграция.
8. Далее система обновляется до версии К9. Система не «новая», в конфигурации есть новые миграции M3 – M5 (М2 уже запускалась, информация в логе) запускаются М3-ДО🡪 М4-ДО🡪 M5-ДО🡪Автоматическая 🡪 М3- ПОСЛЕ 🡪 М4- ПОСЛЕ 🡪 M5- ПОСЛЕ

# Разработчикам миграционных сценариев

Когда разработчик работает локально на своей машине и разрабатывает миграционные сценарии, его конфигурация и схема базы данных постоянно изменяются. Следует помнить, что разрабатываемый в данный момент сценарий миграции с уникальным порядковым номером будет запущен платформой *только один раз*. Если разработчик обнаружил ошибку, исправил её и хочет проверить исправленный вариант, то ему необходимо восстановить исходный вариант базы данных до миграции, в которой, кроме того, хранится вариант конфигурации до предыдущей миграции.

Есть ещё вариант «инкрементальной разработки». Благодаря описанному алгоритму, разработчик после отладки одной миграции может добавить ещё одну (с более увеличенным номером) и проверить её. Однако нужно помнить, что перед публикацией изменений все миграции нужно слить в одну.

**Настоятельно рекомендуется** (хотя и поддерживается данным подходом) в рамках одной версии продукта разрабатывать не более одной миграции (определяемой порядковым номером).

Менять код миграционного сценария, который существовал в предыдущей версии продукта **нельзя**. Для новой версии продукта сценарий должен иметь уникальный порядковый номер старше предыдущего. Исключение, пожалуй, можно сделать, только если в существующем сценарии грубая ошибка. При этом необходимо гарантировать факт того, что версия с ошибкой **никогда** и ни у кого не будет работать (грубо говоря, после неудачной попытки миграции будет восстановлена предыдущая версия системы и резервная копия базы данных).

Если разработчик меняет код миграционного сценария и при этом изменяет конфигурацию доменных объектов в версии продукта, в которой