РУКОВОДСТВО ПРИКЛАДНОГО РАЗРАБОТЧИКА

компонента Интеграционные плагины Hibernate (код: HBRN)

продукта Platform V Persistence (код: HBR)

Содержание

Термины и определения

Термин/сокращение	Расшифровка	Определение
CCIX Сервис межкластерной индексации	-	Сервис межкластерной индексации (CCIX) продукта Platform V Application Sharding (ASD)
JVM	Java virtual machine	Виртуальная Java машина
API	Application Programming Interface	Программный интерфейс приложения
ORM	Object relation mapping	Технология связывания прикладной модели данных с реляционной моделью хранения
REST API	Representational state transfer Application Programming Interface	Стиль взаимодействия компонентов распределенного приложения в сети
Плагин	-	Библиотека или класс, расширяющий функционал библиотеки Hibernate ORM
Конфигуратор	-	Класс или объект класса, предоставляющий возможность настройки плагина
Потребитель сообщений	-	Модуль или сервис, подписанный на сообщения Прикладного Журнала
Клиент APLJ Прикладного Журнала Platform V Backend (#BD).	-	API клиентской библиотеки APLJ Прикладного Журнала Platform V Backend (#BD).
БД	База данных	Совокупность данных, организованных в соответствии с концептуальной структурой, описывающей характеристики этих данных и взаимоотношения между ними
Bean	-	Объект под управлением Spring framework
StandIn	-	Состояние после переключения в резервный контур
Spring boot	-	Проект, упрощающий создание приложений на основе Spring

Spring	-	Универсальный фреймворк с открытым исходным кодом для Java-платформы
HQL	Hibernate Query Language	Объекто-ориентированный язык запросов, который работает с сущностями (классами) и их полями (аттрибутами класса)
JPQL	Jakarta Persistence Query Language	Платформенно-независимый объектно-ориентированный язык запросов, является частью спецификации Java Persistence API (JPA)
JPA	Java Persistence API (JPA)	Спецификация API Java Enterprise Edition
URL	Uniform Resource Locator	Система унифицированных адресов электронных ресурсов

Системные требования

Системные требования, необходимые для настройки, контроля и функционирования продукта Platform V Persistence, приведены в документе «Руководство по установке» в разделе «Системные требования».

Подключение и конфигурирование

Для подключения плагинов к проекту необходимо добавить библиотеку с реализацией плагина в зависимости проекта. Каждый из плагинов реализует определенную интеграцию и может быть подключен независимо от других.

Инициализация плагинов происходит при помощи вызова методов классов, реализованных в библиотеках этих плагинов. Для инициализации и настройки плагина нужно получить экземпляр конфигуратора. В аргументы функции получения конфигуратора передается экземпляр EntityManagerFactory, используемый в проекте пользователя. Для установки конкретных настроек нужно вызвать соответствующие функции конфигуратора и после этого вызвать функцию конфигуратора Configurator#configure.

Получение значения параметров инициализации – зона ответственности приложения, использующего библиотеку.

Описание конкретных настроек алгоритмов и рекомендации по их выбору приведены в разделе Использование программного компонента.

Библиотека не реализует внешних API и не делает журнальных записей, поэтому определить версию библиотеки, использованную при сборке конкретного приложения, можно только путем изучения исходных кодов и скриптов сборки (зависимостей) приложения.

Подключение плагина интеграции с сервисом Прикладной Журнал (APLJ) Platform V Backend (#BD)

Зависимость:

```
<dependency>
    <groupId>sbp.integration.orm</groupId>
    <artifactId>sbp-hibernate-standin</artifactId>
    <version>4.3.0</version>
</dependency>
```

Для получения экземпляра конфигуратора нужно вызвать функцию StandinPlugin.configurator(entityManagerFactory).

Подключение плагина интеграции с Сервис межкластерной индексации (CCIX) Platform V Backend (#BD)

```
<dependency>
    <groupId>sbp.integration.orm</groupId>
    <artifactId>sbp-hibernate-cci</artifactId>
    <version>4.3.0</version>
</dependency>
```

Для получения экземпляра конфигуратора нужно вызвать функцию CciPlugin.configurator(entityManagerFactory).

Подключение плагина интеграции с Генерация ID (Sberflake ID) (SNOW) Platform V Backend (#BD)

```
<dependency>
    <groupId>sbp.integration.orm</groupId>
    <artifactId>sbp-hibernate-sberflake</artifactId>
    <version>4.3.0</version>
</dependency>
```

Для получения экземпляра конфигуратора нужно вызвать функцию SberflakePlugin.configurator().

Миграция на текущую версию

Для миграции на текущую версию достаточно изменить версии зависимостей на актуальные.

Быстрый старт

Каждый плагин подключается независимо от других.

Для подключения к внешним сервисам может понадобиться отдельная настройка подключения к этим сервисам, которая не описана детально в настоящем руководстве. Для получения информации о настройках внешних сервисов нужно обратиться к их документации.

Для получения параметров подключения нужно обратиться к администраторам сервисов.

Ниже приведены примеры подключения и настройки интеграций с рекомендуемыми параметрами для быстрого старта. Эти параметры могут быть неоптимальными для промышленной эксплуатации.

Пререквизиты

Перед подключением плагина нужно иметь доступ к Прикладной Журнал (APLJ) Platform V Backend (#BD) (далее ПЖ), а также настроенную зону ПЖ для проекта и

плагины ПЖ для выбранного вами типа данных. Эти настройки производятся администраторами ПЖ (см. документ «Руководство по системному администрированию» ПЖ, раздел «Настройка типов данных»). Для настройки плагина интеграции понадобится IP адрес Kafka, название зоны ПЖ и название типа данных.

Шаг 1. Подключение зависимости

Добавьте зависимость на библиотеку плагина в проект, как описано в разделе «Подключение плагина интеграции с APLJ Прикладной Журнал (APLJ) Platform V Backend (#BD)».

Шаг 2. Настройка плагина

Добавьте в конфигурацию проекта Bean с настройкой плагина:

```
@Bean
    public StandinPlugin standinPlugin(
        @Qualifier("masterDataSource") DataSource masterDataSource,
        @Qualifier("standinDataSource") DataSource standinDataSource,
        EntityManagerFactory entityManagerFactory,
        JournalCreatorClientApi journalClient
    ) {
        Configurator configurator = StandinPlugin.configurator(entityManagerFactory);
        configurator.setMasterDataSource(masterDataSource);
        configurator.setStandinDataSource(standinDataSource);
        configurator.setJournalClient(journalClient);
         // Идентификатор модуля
        configurator.setModuleIdProvider(() -> "test-module-id");
        // Функция партиционирования
        configurator.setJournalHashKeyResolver(new StaticHashKeyResolver("global-key"));
        // Стратегия репликации - с блокировками или без
```

```
        configurator.setReplicationStrategy(ReplicationStrategy.SIMPLE);

        // Тип сериализатора

        configurator.setSerializerType(SerializerType.JSON_GSON);

        // Режим блокировки корневой сущности при использовании прикладных блокировок

        configurator.setPartitionLockMode(PartitionLockMode.PESSIMISTIC_WRITE);

        // Стратегия контроля порядка применения векторов


configurator.setOrderingControlStrategy(OrderingControlStrategy.OPTIMISTIC_LOCK_VERSION_CONTROL);

        // Стратегия работы с несколькими hashKey в одной транзакции

        configurator.setPartitionMultiplyingMode(PartitionMultiplyingMode.ALLOW);
// Плагин не будет активен, пока не вызван этот метод

        return configurator.configure();

    }
```

Шаг 3. Добавление Bean подписки на события Прикладной Журнал (APLJ)

Добавьте в конфигурацию проекта Bean с настройкой подписки на события Прикладного Журнала (APLJ):

```
@Bean

    public SubscriptionService subscriptionService() {

        return new JournalSubscriptionImpl("<zone-id>");

    }
```

, где zone-id - зона Прикладного Журнала (APLJ) для вашего проекта.

Шаг 4. Добавление служебного Bean Прикладного Журнала (APLJ)

Добавьте в конфигурацию проекта следующий служебный Bean:

```
@Bean
```

```
@Primary
public StandinResourceHelper<String> standinResourceHelper() {
    return new StandinResourceHelper<>("main", "standin");
}
```

Шаг 5. Настройка подключения к Прикладному Журналу (APLJ)

Добавьте в папку с ресурсами в файл application.properties следующие настройки:

```
standin.cloud.client.zoneId=<зона Прикладнго Журнала (APLJ)>
standin.cloud.client.stub=false
standin.cloud.client.sicl_refresh_state_period=100
standin.cloud.client.kafka.bootstrapServers=<IP Kafka>
```

При необходимости так же дополните настройки SSL.

Шаг 6. Проверка работоспособности

Запустите проект и выполните операции по сохранению сущностей с помощью Hibernate ORM. Сообщения об изменениях должны отобразиться в графическом интерфейсе Прикладного Журнала (APLJ) и окраситься в зеленый цвет. Сохраненные сущности должны появиться в резервной БД.

Использование программного компонента

Прикладная репликация

Принцип работы

Плагин интеграции с прикладным журналом предоставляет функционал, позволяющий реализовать распространение сообщений об изменениях данных посредством интеграции с Прикладным Журналом (APLJ) и прикладную репликацию путем воспроизведения операций с API EntityManager на резервной БД. Основные функции плагина:

- Сбор изменений, произведенных в транзакции, и формирование вектора изменений.

- Сериализация и десериализация вектора изменений.
- Формирование служебных сообщений о блокировках.
- Вызов API клиентской библиотеки Прикладного Журнала (APLJ) для отправки сообщений.
- Обработка сообщений из Прикладного Журнала (APLJ) и воспроизведение изменений на резервной БД.
- Установка и снятие прикладных блокировок.
- Переключение активного источника данных по команде из Прикладного Журнала (APLJ).

Плагин подписывается на события, генерируемые библиотекой Hibernate ORM. В момент сброса изменений в БД плагин формирует вектор изменений - структуру данных, содержащую в себе сведения об измененных сущностях. Перед фиксацией транзакции вектор изменений сериализуется в транспортный формат, определяет ключ партиционирования и синхронно вызывает API клиентской библиотеки Прикладного Журнала (APLJ) для отправки сообщения. В зависимости от настроек алгоритмов также могут быть сформированы и отправлены служебные сообщения о блокировках.

Пользователь может сконфигурировать Bean подписки на события Прикладного Журнала (APLJ). В этом случае приложение будет читать сообщения об изменениях данных из Прикладного Журнала (APLJ) и воспроизводить их на резервной БД.

При указании в настройках источника данных резервной БД плагин настроит фабрику сессий таким образом, чтобы она работала с активным источником данных. При этом активный источник данных будет выбираться по команде из Прикладного Журнала (APLJ). В случае переключения в StandIn плагин получит сообщение о переключении и автоматически заменит основной источник данных на резервный.

Требования и ограничения

- **Наличие поля @Version**. Каждая сущность прикладной модели данных должна содержать в себе целочисленное поле, помеченное аннотацией @Version. Предпочтительный тип поля - long.

- **Запрет на использование sequence и автоинкрементных полей**. В связи с возможностью переключения контуров до окончания репликации sequence генераторы и автоинкрементные колонки не поддерживаются. Вместо них нужно использовать генераторы уникальных идентификаторов.
- **Ограничение использования прямых запросов**. Любые HQL, JPQL или native запросы не реплицируются. Как следствие с такими запросами безопасно можно использовать только выборку.

Обязательные настройки

Для работы плагина нужно установить следующие обязательные параметры:

journalClient - реализация API клиентской библиотеки Прикладного Журнала (APLJ). При использовании spring boot эта реализация автоматически появится в контексте приложения.

```
configurator.setJournalClient(journalClient);
```

masterDataSource - основной источник данных:

```
configurator.setMasterDataSource(masterDataSource);
```

standinDataSource - резервный источник данных. Указывается при настройке прикладной репликации. Если плагин используется только для отправки сообщений, этот параметр можно опустить.

```
configurator.setStandinDataSource(standinDataSource);
```

moduleIdProvider - предоставляет идентификатор пользовательского модуля. Подробнее в разделе Идентификация источника данных.

```
configurator.setModuleIdProvider(() -> "module-id"));
```

journalHashKeyResolver - настройка функции партиционирования. Подробнее в разделе Партиционирование.

```
configurator.setJournalHashKeyResolver(hashKeyResolver);
```

Идентификация источника данных

Для идентификации источника данных используется идентификатор, который задается настройкой setModuleIdProvider. Этот идентификатор будет записан в заголовок сообщения, отправляемого в Прикладной Журнал (APLJ). При обработке сообщений Прикладного Журнала (APLJ) идентификатор из заголовка сравнивается с идентификатором, переданным в настройке. Если они не совпадут, то сообщение будет отправлено в Прикладной Журнал (APLJ) как ошибочное. Это позволяет избежать обработки сообщений из других модулей, запущенных в той же зоне с тем же типом данных.

Идентификатор источника данных не должен меняться без крайней необходимости. При изменении этого идентификатора сообщения со старым идентификатором, которые были накоплены в прикладном журнале, не смогут обработаться. При возникновении такой ситуации требуется переинициализация резервной БД.

Партиционирование

Партиционирование сущностей необходимо для распараллеливания потока репликации и работы прикладных блокировок. Партиционирование настраивается путем передачи реализации интерфейса JournalHashKeyResolver в настройку setJournalHashKeyResolver.

Интерфейс JournalHashKeyResolver:

```
public interface JournalHashKeyResolver {
    Set<String> getHashKeys(Collection<?> entities);
}
```

Функция JournalHashKeyResolver#getHashKeys вызывается перед фиксацией транзакции, и в качестве аргумента функции передается коллекция сущностей, измененных в транзакции. Реализация должна сопоставить каждой сущности ее ключ партиционирования (строковое значение, определяющее к какой партиции относится та или иная сущность) и вернуть результирующий набор. После этого один из этих

ключей будет передан в качестве ключа при отправке в Kafka и с помощью него будет выбрана партиция, а также по каждому из этих ключей может устанавливаться служебная блокировка в зависимости от выбранного метода блокировок.

Алгоритм партиционирования выбирается и реализуется пользователем, при этом пользователь ответственен за качество его реализации и эффекты, возникающие вследствие его корректной или некорректной настройки.

Партиционирование влияет на:

- **Степень параллелизма реплики.** При некорректном распределении ключей сообщения могут попадать преимущественно в одни и те же партиции Kafka, из-за чего будет снижаться количество одновременно обрабатываемых сообщений на стороне применения, и при значительной транзакционной нагрузке реплика может отставать.
- **Качество порядка репликации.** При наличии в одной транзакции сущностей с разными ключами партиционирования возникает неоднозначность определения партиции при отправке сообщений в Прикладной Журнал (APLJ). Это может привести к тому, что зависимые транзакции могут попадать в разные партиции и, как следствие, обрабатываться не в том порядке, в котором они были на основной БД. При отсутствии механизмов компенсации это приведет к ошибке репликации. Механизмы компенсации имеют свои издержки на упорядочивание транзакций, и хоть они и позволяют избавиться от ошибок репликации, качество порядка реплики все равно будет влиять на производительность.
- **Количество записей в служебной таблице.** При использовании прикладных блокировок для каждого переданного ключа партиционирования будет создана запись в служебной таблице. Это стоит учитывать при реализации функции партиционирования, так как большое количество записей может привести к разрастанию таблицы и излишнему расходу ресурсов системы хранения БД.
- **Доступность данных при переходе в StandIn.** При использовании форсированного перевода в StandIn данные, по которым есть отставание реплики, будут заблокированы для использования. Так как данные блокируются по ключу партиционирования, от распределения ключей зависит количество сущностей, которые будут заблокированы одним неотреплицированным сообщением. К

примеру, если для всех сущностей в БД используется один ключ, то любое отстающее сообщение будет блокировать для использования все данные в резервной БД.

Рекомендации для реализации функции:

- После создания сущности ее ключ партиционирования не должен меняться.
- Работа с данными и функция партиционирования должны быть построены таким образом, чтобы в каждой транзакции изменялись сущности по как можно меньшему количеству ключей. Идеальная ситуация - когда в транзакции меняются данные только по одному ключу. Это избавляет от возможных проблем с порядком репликации.

Библиотека содержит несколько встроенных реализаций:

- StaticHashKeyResolver - константная функция партиционирования. При использовании этой реализации репликация будет однопоточной. Эта реализация может быть удобной для первого запуска и тестирования, но в промышленной эксплуатации однопоточная реплика не рекомендуется к использованию.
- InterfaceBasedHashKeyResolver - функция партиционирования на основе интерфейса HashKeyProvider. При использовании этой стратегии все сущности должны реализовывать интерфейс HashKeyProvider и возвращать из функции getHashKey ключ партиционирования.
- PrimitiveFieldHashKeyResolver - функция партиционирования, которая читает значение примитивного поля, указанного в конструкторе, и использует его в качестве ключа партиционирования.
- ReferenceFieldHashKeyResolver - функция партиционирования, которая читает значение ссылочного поля, указанного в конструкторе, и использует его в качестве ключа партиционирования.

Установка функции партиционирования:

```
configurator.setJournalHashKeyResolver(customPartitioner);
```

При работе с данными предпочтителен подход изменения в одной транзакции данных, относящихся только к одному ключу партиционирования. В этом случае

практически отсутствует риск нарушения порядка репликации. Если для пользователя допустим такой подход, то он может явным образом запретить транзакции по данным, относящимся к разным ключам. Сделать это можно с помощью настройки setPartitionMultiplyingMode, которая принимает в качестве значения перечисляемый тип PartitionMultiplyingMode.

Константы перечисляемого типа PartitionMultiplyingMode:

Значение	Описание
FORBIDDEN	Запрещает транзакции по нескольким ключам партиционирования. В случае обнаружения такой транзакции будет выброшено исключение MultiplePartitionsException
WARN	Значение по умолчанию. Разрешает транзакции по нескольким ключам партиционирования. Событие обнаружения такой транзакции будет залогировано
ALLOW	Разрешает транзакции по нескольким ключам партиционирования

Пример:

```
configurator.setPartitionMultiplyingMode(PartitionMultiplyingMode.FORBIDDEN);
```

Центричность

Классический способ реализации функции партиционирования - использование идентификаторов корневых сущностей центричной модели.

Модель данных является центричной, когда для каждой сущности модели данных можно определенным образом сказать, какой родительской сущности она принадлежит.

Для примера клиент и ряд связанных с ним атрибутов, таких как договор, реквизит, адрес и пр. В этом примере клиент будет родительской сущностью, а связанные с ним

атрибуты - дочерними. Если у каждой сущности можно узнать, какому клиенту она принадлежит, то это будет клиентоцентричная модель. Идентификатор клиента в этом случае можно использовать в качестве ключа партиционирования.

Транзакции по данным, принадлежащим одному клиенту, будут попадать в одну и ту же партицию, и между ними будет гарантия порядка отправки - применения. При отставании реплики по отдельным сущностям будут блокироваться все сущности, принадлежащие к той же родительской сущности, что и отстающие.

Распараллеливание потока репликации

При отправке сообщения в Прикладной Журнал (APLJ) оно попадает в определенную партицию топика Kafka, при этом на стороне применения на каждую партицию будет создаваться один потребитель Kafka. Из этого следует, что сообщения одной партиции будут обрабатываться последовательно и параллельно с сообщениями из других партиций. Если записывать сообщения только в одну партицию, то при хоть сколько-нибудь значимой транзакционной нагрузке реплика начнет отставать и сообщения будут копиться в прикладном журнале. Для того, чтобы этого не происходило, сообщения должны быть равномерно распределены по всем партициям топика. Количество партиций на топик настраивается администраторами Прикладного Журнала (APLJ). Пользователь управляет партиционированием посредством реализации функции партиционирования.

Прикладные блокировки

При работе алгоритмов прикладной репликации может возникнуть ситуация, когда после отправки сообщения об изменениях по какой-либо причине не удается зафиксировать транзакцию в основной БД. При этом сообщение отправляется в Прикладной Журнал (APLJ) и реплицируется в резервную БД. В этом случае возникнет неконсистентность баз данных, и после переключения в StandIn прикладной модуль будет работать с неконсистентными данными.

Для защиты от работы с неконсистентными данными используются прикладные блокировки.

При работе с блокировками перед фиксацией транзакции, вместе с данными, будет отправлено сообщение о блокировке по ключу партиционирования. После коммита будет асинхронно отправлено сообщение о разблокировке по этому же ключу. При

этом данные в резервной БД считаются доступными, если по их ключу партиционирования пришло три сообщения: блокировка, данные и разблокировка. Если хоть одно из этих сообщений не пришло, то данные остаются заблокированными и после переключения в StandIn при попытке изменения этих данных будет выброшено исключение CanNotAcquireLockException.

Блокировки устанавливаются путем создания или изменения записи о блокировке в служебных таблицах.

Алгоритм блокировок устанавливается настройкой setReplicationStrategy, которая в качестве аргумента принимает перечисляемый тип ReplicationStrategy:

Значение	Описание
SIMPLE	Алгоритм работы без прикладных блокировок. Не обеспечивает консистентность баз данных
STANDIN_LOCKS	Алгоритм блокировок без поддержки идемпотентности установки блокировок
PARTITION_LOCKS	Улучшенный алгоритм блокировок с поддержкой идемпотентности установки блокировок и подтверждения транзакций на стороне применения

Контроль версионирования и порядок репликации

Для обеспечения консистентности важно, чтобы транзакции реплицировались в том же порядке, в котором они были выполнены на основной БД. Для контроля порядка применения используется версионирование сущностей на основе механизма оптимистичных блокировок Hibernate ORM. Для обеспечения этого прикладная модель должна соответствовать требованию по наличию целочисленного поля с JPA аннотацией *@Version* в каждой сущности. Предпочтительный тип поля - *long*. При работе с данными Hibernate будет автоматически увеличивать значение этого поля при каждом изменении сущности. При формировании вектора изменений в события изменений будут записаны исходные и новые версии сущностей.

При применении вектора изменений будет происходить сравнение исходной версии сущности и текущей версии записи в резервной БД. Если они равны, то транзакция будет сохранена в резервную БД с изменением версии записи на новую версию из события изменения. При различии версий возникнет исключительная ситуация, и, в зависимости от выбранного алгоритма контроля версий, плагин отреагирует на нее соответствующим образом.

Сообщения, которые не могут быть обработаны вследствие ошибки контроля версионирования, будут отправлены в Прикладной Журнал (APLJ) с соответствующей пометкой об ошибке, что отобразится в графическом интерфейсе Прикладного Журнала (APLJ). Сообщения с ошибкой в последствии можно будет отправить на повторное применение с помощью графического интерфейса Прикладного Журнала (APLJ).

Ошибки контроля версий не являются дефектом плагина и возникают вследствие неверной настройки функции партиционирования, работы с несколькими корнями партиционирования в одной транзакции либо инцидентов Прикладного Журнала (APLJ).

Алгоритм контроля порядка репликации задается настройкой setReplicationStrategy:

Значение	Описание
DISABLED	Контроль версий отключен
OPTIMISTIC_LOCK_VERSION_CONTROL	Простой контроль версий. При несовпадении версий будет выброшено исключение
BASIC_IDEMPOTENCY_CONTROL	Устаревшее. Контроль версий с поддержкой идемпотентности применения. Можно использовать только если в БД нет никаких констрейнтов, за исключением первичных ключей
SIMPLE_VERSION_CONTROL	Простой контроль версий. При несовпадении версий будет выброшено исключение. Версии сверяются заранее,

	перед применением вектора изменений
IDEMPOTENT_VERSION_CONTROL	Контроль версий с улучшенной поддержкой идемпотентности применения
IDEMPOTENT_ORDERING_CONTROL	Контроль версий с поддержкой идемпотентности применения и упорядочивания транзакций на стороне применения

Переключение в StandIn

Переключение в StandIn инициируется в прикладном журнале. После этого Прикладной Журнал (APLJ) рассылает в экземпляры пользовательских модулей сообщения о начале переключения. В этот момент начинается окно недоступности, и попытка изменения данных в БД с помощью Hibernate будет вызывать исключение ReourceNotAllowedException. Прикладной Журнал (APLJ) будет дожидаться окончания обработки всех накопленных сообщений и после этого разошлет сообщение об окончании переключения. После того, как пользовательский модуль получит сообщение о переключении, плагин сделает активным резервный источник данных, и работа с БД снова станет возможной.

Настройка сериализации

Поддерживаются несколько встроенных сериализаторов для сериализации сообщений об изменениях данных. Сериализатор устанавливается настройкой setSerializerType, принимающей в качестве аргумента перечисляемый тип SerializerType. Значение по умолчанию - BINARY_KRYO.

Значение	Описание
BINARY_KRYO	Бинарная сериализация с использованием сериализатора Kryo
BINARY_KRYO_GZIP	Бинарная сериализация с использованием сериализатора Kryo и gzip сжатием

BINARY_FST	Бинарная сериализация с использованием сериализатора Fst
BINARY_FST_GZIP	Бинарная сериализация с использованием сериализатора Fst и gzip сжатием
JSON_GSON	Json сериализация с использованием сериализатора Gson
BINARY_JAVA	Бинарная сериализация стандартным Java сериализатором

Настройка фильтров

Включить или исключить сущность из процесса репликации можно с помощью аннотации @Standin и установки одного из значений атрибута replication:

- ENABLED - включает репликацию для сущности или поля;
- DISABLED - исключает сущность или поле из репликации.

Пример:

```
@Entity
@Standin(replication = Replication.DISABLED)
public class Product {     ... }
```

По умолчанию репликация включена для всех сущностей и полей. Изменить значение по умолчанию можно с помощью настройки setEnableReplicationByDefault:

```
configurator.setEnableReplicationByDefault(false);  // Отключает репликацию по умолчанию
```

Так же есть возможность настроить вручную белый или черный списки для репликации без использования аннотаций. Если репликация включена по умолчанию, тогда сущности и поля из черного списка будут исключены из репликации. Если репликация выключена по умолчанию, тогда только сущности и поля из белого списка будут участвовать в репликации.

Примеры:

```
configurator.setEnableReplicationByDefault(true); // Включает репликацию по умолчанию для всех сущностей

configurator.getBlacklist().addEntity("org.example.Product1"); // Исключает из репликации сущность Product1

configurator.getBlacklist().addProperty("org.example.Product2", "attribute"); // Исключает из репликации поле attribute сущности Product2
```

```
configurator.setEnableReplicationByDefault(true); // Выключает репликацию по умолчанию для всех сущностей

configurator.getWhitelist().addEntity("org.example.Product1"); // Включает репликацию для сущности Product1

configurator.getWhitelist().addProperty("org.example.Product2", "attribute"); // Включает репликацию для поля attribute сущности Product2
```

Миграция схемы и модели данных

Плагин репликации не обеспечивает явным образом обратной совместимости пользовательской прикладной модели данных при ее обновлении. В прикладном журнале в любой момент времени присутствуют накопленные сообщения об изменениях данных с той моделью, с которой они были сформированы. Несовместимые изменения модели могут привести к ошибкам репликации сообщений, сформированных с предыдущей версией модели. Поэтому пользователь должен обеспечивать обратную совместимость при изменении прикладной модели данных.

Изменения модели нужно производить с учетом следующих возможных последствий:

- Добавление сущности или необязательного поля - обратно совместимое изменение. Не приводит к ошибкам репликации. Накопленные сообщения могут быть обработаны как с предыдущей версией, так и с новой.
- Добавление обязательного поля - несовместимое изменение. Может привести к ошибкам репликации, что потребует переинициализации резервной БД.

- Удаление сущности или поля - несовместимое изменение. Может привести к ошибкам репликации, что потребует переинициализации резервной БД. Можно выполнить в два релиза - в первом релизе исключить поле или сущность из репликации и во втором удалить их.
- Переименование сущности или поля - несовместимое изменение, интерпретируется как удаление старой сущности или поля и добавление новых. Можно выполнить в два релиза - в первом добавить новую сущность или поле и исключить старые из репликации. Во втором релизе - удалить старую сущность или поле.
- Изменение типа поля - несовместимое изменение, может привести к ошибкам репликации, что потребует переинициализации резервной БД.

При изменении модели данных в два релиза нужно убедиться, что все сообщения с предыдущей моделью были обработаны.

Откат модуля к предыдущей версии равнозначен выпуску новой версии с обратными операциями изменения прикладной модели данных.

Таблица обратных операций:

Операция	Обратная операция
Добавление поля	Удаление поля
Переименование поля	Переименование поля
Изменение типа поля	Изменение типа поля
Удаление поля	Добавление поля
Добавление сущности	Удаление сущности
Переименование сущности	Переименование сущности
Изменение суперкласса сущности	Изменение суперкласса сущности
Удаление сущности	Добавление сущности

К примеру, если в версии 2 пользовательского модуля был добавлен новый класс сущности, то откат на версию 1 с точки зрения изменения модели данных будет

равнозначен удалению сущности, что приведет к эффектам, описанным для операции удаления сущностей.

Заполнение заголовков контейнера сообщений

При необходимости ручного заполнения заголовков контейнера сообщения (журнала) пользователь может воспользоваться настройкой setJournalListener, передав в нее реализацию интерфейса com.sbt.pprb.integration.replication.journal.JournalHeaderHandler. Функция интерфейса handle будет вызываться при отправке сообщения в Прикладной Журнал (APLJ), в аргумент будет передан заголовок контейнера, и пользователь сможет заполнить необходимую информацию.

Пример:

```
configurator.setHeaderHandler(header -> header.setSourceInfo("test-module"));
```

Межкластерная индексация

Плагин интеграции с Сервисом межкластерной индексации (CCIX) Platform V Backend (#BD) собирает изменения сущностей, произошедшие в транзакции, и перед коммитом синхронно вызывает API Сервиса межкластерной индексации (CCIX) для записи значений индексов, сконфигурированных для прикладного модуля. В качестве ключа индекса используется значение какого-либо поля измененной сущности, в качестве значения - идентификатор кластера, установленный с помощью настройки setClusterId. В дальнейшем можно получить идентификатор кластера, соответствующий какому-либо ключу. Это можно использовать для определения кластера, в котором была сохранена та или иная сущность.

Настройка подключения

Подключение к сервису настраивается с помощью настройки setCciServer, в аргументы нужно передать хост сервера и порт:

```
configurator.setCciServer("localhost", 80);
```

Установка текущего кластера

Для установки идентификатора кластера, с которым работает экземпляр прикладного модуля, нужно использовать настройку setClusterId:

```
configurator.setClusterId("cluster-id");
```

Это значение будет использовано в качестве значения индекса при записи в Сервис межкластерной индексации (CCIX).

Настройка индексов

Индексы сущностей конфигурируются при помощи аннотации @ClusterIndex:

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Repeatable(ClusterIndexes.class)
public @interface ClusterIndex {
    String name();
    String[] fields();
    boolean required() default true;
}
```

Атрибуты аннотации:

Атрибут	Тип	Назначение
name	String	Имя существующего индекса в сервисе межкластерной индексации
fields	String[]	Массив полей, значения которых будут использованы в качестве ключа индекса

required	boolean	Обязательность заполнения индекса. Если true, то при отсутствии значения будет выброшено исключение IllegalIndexFieldValueException

Аннотация устанавливается над классом сущности. Указывается имя индекса и поля, используемые в качестве ключа. Перед коммитом плагин прочитает значения этих полей и запишет в индекс с указанным именем значение полей и текущий кластер.

Пример:

```
@Entity

@Getter

@Setter

@Table(name = "PRODUCT")

@ClusterIndex(name = "CCI_PRODUCT_SINGLE", fields = { "partitionKey" }, required = false)

@ClusterIndex(name = "CCI_PRODUCT_COMPOSITE", fields = { "partitionKey1", "partitionKey2" }, required = false)

@ClusterIndex(name = "CCI_PRODUCT_OWNER", fields = { "owner.partitionKey" }, required = false)

@ClusterIndex(name = "CCI_PRODUCT_OWNER_ID", fields = { "owner" }, required = false)

@ClusterIndex(name = "CCI_PRODUCT_OWNER_INH", fields = { "owner.basePartitionKey" }, required = false)

@ClusterIndex(name = "CCI_PRODUCT_OWNER_INH_COMPOSITE", fields = { "owner.partitionKey", "owner.basePartitionKey" }, required = false)

public class Product extends BaseEntity {

    @Column(name = "NAME")

   private String name;

    @Column(name = "CLUSTER")
```

```
    private Long partitionKey;
     @Column(name = "CLUSTER1")
    private Long partitionKey1;
     @Column(name = "CLUSTER2")
    private Long partitionKey2;
     @ManyToOne
    @Cascade(CascadeType.ALL)
    private Owner owner;
}
```

Получить значение индекса после записи в него можно следующим образом:

```
String clusterId = cciPlugin.getIndexService(entityManagerFactory).getZone(indexName, indexKey);
```

Генерация идентификаторов

Для работы прикладной репликации идентификаторы сущностей должны быть сгенерированы генератором уникальных идентификаторов. Реализацию одного из таких генераторов предоставляет сервис Генерации ID (Sberflake ID) (SNOW) Platform V Backend (#BD). Подключить интеграцию с ним можно с помощью плагина интеграции с сервисом Генерации ID (Sberflake ID) (SNOW). Данный сервис генерирует целочисленные идентификаторы типа long.

Настройка подключения к сервису Генерации ID (Sberflake ID) (SNOW)

Сервис Генерации ID (Sberflake ID) (SNOW) предоставляет REST API. Для подключения к сервису нужно указать URL сервиса:

```
configurator.setUrl("<url сервиса>");
```

Для идентификации модуля и регистрации плагином экземпляра генератора для прикладного модуля нужно указать идентификатор модуля:

```
configurator.setModuleId("<module-id>");
```

Для того, чтобы использовать генератор по умолчанию для всех сущностей без явного указания генератора, нужно установить следующую настройку:

```
configurator.enableByDefault();
```

При этом важно, чтобы Bean плагина инициализировался раньше фабрики сессий Hibernate.

Использование генератора сервиса Генерации ID (Sberflake ID) (SNOW)

Если установлена настройка enableByDefault, то для автоматически генерируемых идентификаторов без указания конкретного генератора будет использован генератор сервиса Генерации ID (Sberflake ID) (SNOW):

```
    @Id
    @GeneratedValue
    private Long id;
```

Для явного указания генератора нужно объявить поле идентификатора следующим образом:

```
    @Id
    @GeneratedValue(generator = "sberflake")
    @GenericGenerator(name = “sberflake”, strategy = “com.sbt.pprb.integration.hibernate.idgen.plugin.PlatformIdGenerator”)
    private Long id;
```

Часто встречающиеся проблемы и пути их устранения

- Проблема:

В графическом интерфейсе Прикладного Журнала (APLJ) появляются красные записи с сообщением об ошибке «VersionControlException...» или «Unable to find entity...».

- Решение:

  Такие ошибки возникают вследствие нарушения порядка применения векторов изменений и требуют исследования прикладной модели и функции партиционирования для оптимизации распределения сообщений по партициям. Не являются дефектом Platform V Persistence.

  - При единичных проблемных векторах воспользуйтесь функциональностью повторной отправки векторов через графический интерфейс Прикладного Журнала (APLJ).
  - При большом количестве сообщений с подобными ошибками необходимо пересмотреть реализацию функции партиционирования в сторону снижения количества ключей партиционирования на одну транзакцию и уменьшения вероятности попадания зависимых транзакций в разные партиции Kafka.
  - Если указанные выше способы не помогли решить проблему — подключите механизмы компенсации, как описано в инструкции по использованию программного компонента. Важно учитывать, что не все системы-потребители сообщений могут реализовывать механизмы компенсации. Стоит ознакомиться с документацией этих систем, чтобы узнать о наличии возможности обработки неупорядоченных сообщений.

- Проблема:

  После настройки прикладной репликации и запуска модуля в графическом интерфейса Прикладного Журнала (APLJ) не появляются новые записи.

- Решение:

  - Проверьте настройку зоны Прикладного Журнала (APLJ). Она должна быть создана в Прикладном Журнале (APLJ) и указана в настройках плагинов.
  - Проверьте настройку клиента Прикладного Журнала (APLJ) kafka.bootstrapServers. Должен быть указан корректный IP-адрес сервера Kafka.

- Убедитесь, что в контексте приложения создается Bean StandinPlugin, Bean SubscriptionService и импортирована конфигурация клиента Прикладного Журнала (APLJ).

- Проблема:

  В графическом интерфейсе Прикладного Журнала (APLJ) появляются красные записи с сообщением об ошибке «module-id not found», «Unable to set field...» или «Field not found...».

- Решение:

  Такие ошибки возникают при работе нескольких модулей в одной зоне Прикладного Журнала (APLJ) (далее ПЖ) из-за того, что сообщение попало в тот модуль, в котором нет метаданных для применения сообщения.

  Для настройки роутинга сообщений укажите пользовательский тип данных для каждого из модулей (настройка setDataType) и обратитесь к сопровождению ПЖ для добавления этих типов данных (см. документ «Руководство по системному администрированию» ПЖ, раздел «Настройка типов данных»).