**1) Зачем нужен Spring?**

Это платформа которая ускоряет и упрощает разработку приложения. Суть заключается в следующем - мы управление, создание между зависимостями объектов передаем фреймворку. Нам не нужно следить за жизненным циклом объектов их созданием, удалением, внедрением.

**2) Откуда Spring берет зависимости?**

Берет из контейнера зависимостей. Он работает на основе метаданных, которые мы отдаем спрингу в виде xml или javaconfig или повешенных аннотаций, либо смесь аннотаций и xml, так же можно отдать groovy скрипт

**2.1) Какие типы бинов сущесмтвуют**

**2.2) Потокобезопасен ли singleton?**

Нет, Spring Singelton Bean не является потокобезопасным. Вот пример

public class Emmloyee {

private int id;

private String name;

public int getId() {

return id;

}

public void setId(int id) {

this.id = id;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

}

И вотapplicationContext.xml

<bean id="emp" class="com.manikant.Emmloyee" p:id="26" p:name="Manikant Gautam">

А вот тестовый класс

public class Test {

public static void main(String[] args) {

ApplicationContext ctx=new ClassPathXmlApplicationContext("com/manikant/config.xml");

Emmloyee emp=(Emmloyee)ctx.getBean("emp");

System.out.println("User "+emp.getName() + " is of age "+emp.getId());

emp.setName("Changed value");

Emmloyee emp1=(Emmloyee)ctx.getBean("emp");

System.out.println("User "+emp1.getName() + " is of age "+emp1.getId());

}

}

Вот вывод

User \*\*Manikant Gautam\*\* is of age 26

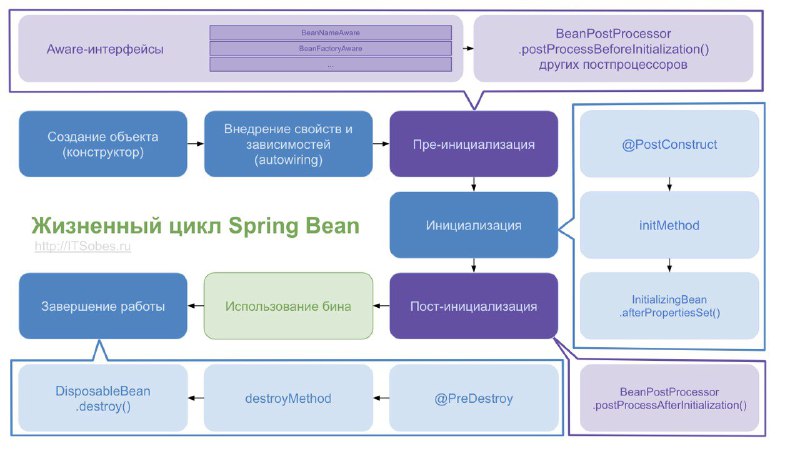
User \*\*Changed value\*\* is of age 26

изменение также valueотражается emp.setName("Changed value");на разных bean emp1.

**2.3) Жизненый цикл бина**

Beans – центральный объект заботы Spring Framework. За кулисами фреймворка с ними происходит множество процессов. Во многие из них можно вмешаться, добавив собственную логику в разные этапы жизненного цикла. Через следующие этапы проходит каждый *отдельно взятый* бин:  
  
**а. Инстанцирование объекта**. Техническое начало жизни бина, работа конструктора его класса;  
  
**б. Установка свойств** из конфигурации бина, [внедрение зависимостей](https://itsobes.ru/JavaSobes/kak-spring-framework-realizuet-pattern-dependency-injection);  
  
**в. Нотификация aware-интерфейсов**. BeanNameAware, BeanFactoryAware и другие. Мы уже писали о таких интерфейсах [ранее](https://itsobes.ru/JavaSobes/kogda-ispolzuiut-aware-interfeisy-v-spring). Технически, выполняется системными подтипами BeanPostProcessor, и совпадает с шагом 4;  
  
**г. Пре-инициализация** – метод postProcessBeforeInitialization() интерфейса BeanPostProcessor;  
  
**д. Инициализация.** Разные способы применяются в таком порядке:  
**•** Метод бина с аннотацией @PostConstruct из стандарта JSR-250 (рекомендуемый способ);  
**•** Метод afterPropertiesSet() бина под интерфейсом InitializingBean;  
**•** Init-метод. Для отдельного бина его имя устанавливается в параметре определения initMethod. В xml-конфигурации можно установить для всех бинов сразу, с помощью default-init-method;  
  
**е. Пост-инициализация** – метод postProcessAfterInitialization() интерфейса BeanPostProcessor.

Когда IoC-контейнер завершает свою работу, мы можем кастомизировать этап штатного уничтожения бина. Как со всеми способами [финализации](https://itsobes.ru/JavaSobes/finalize) в Java, при жестком выключении (kill -9) гарантии вызова этого этапа нет. Три альтернативных способа «деинициализации» вызываются в том же порядке, что симметричные им методы инициализации:  
  
**е.1.** Метод с аннотацией @PreDestroy;  
**е.2.** Метод с именем, которое указано в свойстве destroyMethod определния бина (или в глобальном default-destroy-method);  
**е.3.** Метод destroy() интерфейса DisposableBean.  
  
Не следует путать жизненный цикл отдельного бина с жизненным циклом контекста и этапами подготовки фабрик бинов. О них мы поговорим в будущих публикациях.



**3) Что произойдет если спринг не найдёт нужные зависимости?**

Упадет приложение с ошибкой- данная зависимость не найдена

Чтобы приложение не упала если бин не был найден – можно исп @Autowired аннотацию, в которую передать параметр required = false, И если спринг не найдёт этот бин, то он передаст туда просто null. Пометив таким образом что бин не обзательный, и его можно не инжектить

*Способ не особо*- повесить аннотацию @Lazy, но если мы обратимся к бину, его там не окажется и все равно упадет приложение. Либо исп Conditional установив условия использования

**4) А случай кода нашлось несколько бинов?**

Будет ошибка определения какой бин внедрять (несколько кандидатов. Это решается аннотацией **@Primary** говоря о том что какой то бин первичный. Либо повесить **@Qualifire** где указать имя бина

**5) Какие типы внедрение зависимостей**

- Конструктор

- Сеттер

- в Поля которые приватные непосредственно(использует рефлексию)

**6) Что использовать если пишем приложение на спринге с использованием БД**

- Можно исп JDBC – либа для работ с бд

- Hibernate (преобразуем объекты бд(строки) в классы джава)

**7) Проблемы виде когда мы объект мапим на реляционную модель БД (проблемы при представлении данных в объектном)**

- В объектах есть наследование, а в реляционных таблицах этого нет и мы не можем спроецировать наследников(у которых добавляться поля) от базового класса, мы не сможем сложить нормально в одну таблицу в бд. Нам придаться сделать свалку данных и 1го объекта и его наследника.

Решение- делать таблицу для каждого объекта, но будет дублирование данных, потому что родитель имеет поле, а его наследники дополняют данные. И тогда не будет общей таблицы с полями.

- Нельзя проецировать коллекции на реляционную модель. В бд нет понятия коллекции, мы оперируем таблицами там

Решение – делать доп таблицу и вспомогательные ключи и отображать связи

**8) Проблемы Hibernate и как решаються**

N+1 – избыточные запросы

Проблема N + 1 возникает, когда фреймворк доступа к данным выполняет N дополнительных SQL-запросов для получения тех же данных, которые можно получить при выполнении одного SQL-запроса.

Она встречается, когда мы достаем список сущностей со связями (которые нам необходимо подтянуть) И вместо того чтобы сразу сделать джоин двух таблиц, хибер делает селект из 1 таблицы и потом для каждой найденной строки будет делать дополнительный селект чтобы во второй таблице найти связанную строку. Проблема не только у хибера наблюдаеться

Решение: делать джоин и не делаем n+1 – сообщаем хиберу путем указания нужного джоина

Либо же используем аннотацию @EntityGraph для указания связанной сущности

можно использовать JOIN FETCH, чтобы избежать проблемы с N + 1:

List<PostComment> comments = entityManager**.createQuery**("""  
 select pc  
 from PostComment pc  
 join fetch pc.post p  
 """, PostComment**.class**)  
**.getResultList**();

@Lazy – загрузка сущностей (связей) только в момент обращения. Даже если вы явно перейдете на использование FetchType.LAZY для всех ассоциаций, то вы все равно можете столкнуться с проблемой N + 1.

@Eager -загрузка жадная- сразу же

https://habr.com/ru/companies/otus/articles/529692/

https://habr.com/ru/articles/714704/

**9) Какие оптимизации запросов в БД существуют?**

Вместо кучи insert делать один большой на все строки.

Использование индексов при поиске по таблицам.

Индекс – сущность в бд, работает как указатель (вешаем на колонку). Индекс ускоряет процесс запроса, предоставляя быстрый доступ к строкам данных в таблице, аналогично тому, как указатель в книге помогает вам быстро найти необходимую информацию.

<https://www.youtube.com/watch?v=LpEwssOYRKA>

В общем виде, индекс в базах данных — это файл с последовательностью пар ключей и указателей.

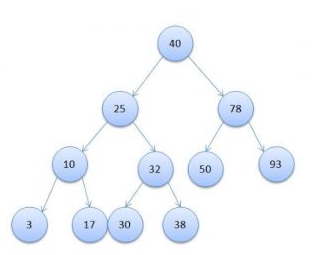
Какие типы индексов бывают

Б индекс, (B-Tree индек), хеш-индекс

https://habr.com/ru/companies/quadcode/articles/696498/

Bitmap-индексы - построение двухмерного массив

Дерево (древовидный индекс) — это специального вида структура, у которой есть корневая вершина и у каждого узла может быть несколько дочерних узлов. При этом каждый узел встречается только один раз и может иметь всего один родительский узел. Выглядит это так:



Хеш индекс – считает хеш для значений столбца и сопоставляет ему колонку

Джин индекс – который исп для индексации сложны структру и полнотекстового поиска

Составные индексы — строятся по нескольким полям, при этом расположение полей является важным.

Обычные индексы — состоят из одного поля. Здесь, вероятно, все понятно. Обычный каталог страничек.

**10) Отличие Dependency inversion от Dependency Injection**

https://techrocks.ru/2021/02/15/dependencies-inversion-and-injection/

**Микросервисы**

1) Плюсы и минусы

Удобство поддержки для разработчика

Простота разработки

Минусы-нужна развитая инструктора и поддержка со стороны девопсов

- подключение балансировщика если сервис откажет

Частично решает проблему докер - слоями ПО

Проблема распределенного состояния и отсутствие консистентности в микро-сервисной архитектуре

Когда приходит запрос – по системе он продвигается постепенно и какие то сохранили результаты а какие то нет.

Проблема обнаружения ошибок- нужно постоянно анализировать и отлавливать логи какой микросервис выкидывает ошибку. Проблема децентрализации логов сервиса. Здесь приходит на помощь кибана и эластик серч

**Монолиты**

Традиционный подход к созданию приложений путем объединения всех функций в единую сущность, объединяющую интерфейс и серверную часть, по-прежнему актуален. Дело в том, что не всегда есть смысл в усложнении архитектуры. Если требования к сервису (например, к веб-сайту) включают только скорость, доступность и возможность обслуживания статического контента, который находится в свободном доступе, монолитная архитектура для таких целей — вполне естественный выбор. А всё остальное было бы чрезмерным усложнением.

**Плюсы монолитов**

* Скорость, простота и низкая стоимость создания. Приложения, основанные на монолитной архитектуре, как правило, быстрее, проще и дешевле создавать, поскольку они не требуют синхронизации и объединения сложных частей. Поэтому, если монолитное приложение будет делать то, что от него требуется, без ущерба для продвижения, то можно использовать монолитную архитектуру.
* Удобное тестирование. Протестировать монолитное приложение гораздо проще, чем микросервисы или бессерверное. Можно запустить и протестировать приложение на сервере разработчика или в промежуточной среде, а также применить стандартный процесс развертывания для проверки изменений перед запуском приложения в продакшн.
* Простая инфраструктура. Монолитные приложения используют один сервер для внешнего интерфейса, серверной части и базы данных, что упрощает требования к инфраструктуре. А ​​для повышения скорости, масштабируемости, доступности и безопасности монолитных веб-приложений можно добавить сеть доставки контента (например, Cloudflare).
* Снижение затрат на поддержку. При монолите вам нужно поддерживать только один репозиторий. Вам также понадобится только один конвейер тестирования и развертывания, что может значительно снизить затраты, поскольку создание, настройка и обслуживание нескольких конвейеров выйдет дороже, ведь нужно будет обеспечить согласованность между ними. Все данные, используемые приложением, также могут храниться в единой базе данных.
* Мониторинг. Простая инфраструктура обеспечивает дополнительное преимущество в виде упрощения мониторинга.
* Легкое управление транзакциями и поддержка целостности данных. Тот факт, что монолитные приложения обычно используют одну базу данных, означает, что здесь проще управлять транзакциями и поддерживать целостность данных.

**Минусы монолитов**

* Большие массивы кода. Монолитная архитектура подразумевает единую кодовую базу. Если объемы кода растут, его становится труднее читать и понимать, что затрудняет интеграцию новых разработчиков в проект.
* Могут быть проблемы с обновлениями. Обновление монолитного приложения означает перестройку и повторное развертывание всего кода.
* Высокий уровень зависимостей. Единый код означает, что компоненты тесно связаны, и изменения в логике одного модуля или службы сопряжены с гораздо более высоким риском влияния на код и работу других модулей. Порой трудно предсказать последствия для приложения даже после небольших изменений в коде. Поэтому каждое обновление вынуждает повторно тестировать весь продукт.
* Отсутствие гибкости. Монолитная архитектура одновременно ограничивает диапазон технологий, которые вы можете использовать, и означает, что технологический стек должен быть строго согласован в глобальном плане. Поэтому вы не можете использовать другую, возможно, более подходящую технологию для каких-то отдельных частей приложения.
* Проблемы с масштабированием. Монолитные приложения плохо масштабируются. Поэтому, если вы создаете монолитное приложение, которое неожиданно становится чрезвычайно популярным и должно справляться с гораздо большими нагрузками, а также расширяться, вам придется перенести его на другой шаблон. Либо на микросервисы, либо на бессерверную архитектуру.

**Примеры монолитов**

Монолитная архитектура обычно лучше всего подходит для небольших и простых приложений. Мы уже приводили пример веб-сайта. Другим примером может быть быстрая и экономичная разработка MVP или программного обеспечения, предназначенного для выполнения какой-то простой задачи.

**Микросервисы**

С развитием цифровых технологий растет сложность приложений. Как мы уже говорили выше, монолитная архитектура может стать непреодолимой преградой для масштабирования. Кроме того, очень важно стало и распределение рисков путем разделения сервисов таким образом, чтобы сбой одного не приводил к остановке всего приложения. Для этого и был разработан микросервисный подход. Архитектура микросервисов разделяет приложение на отдельные службы или группы.

**Плюсы микросервисов**

* Итеративная разработка. Архитектура программного обеспечения на основе микросервисов хорошо подходит для организации гибкой разработки ПО, основанной на итеративном подходе. Разбиение уже работающего приложения на отдельные, слабо связанные между собой сервисы означает, что новые возможности и функции могут быть проверены и добавлены без риска остановки приложения.
* Гибкость. Микросервисы обеспечивают большую гибкость процесса разработки ПО сразу на нескольких уровнях. Это позволяет разным командам одновременно работать над разными сервисами и упрощает интеграцию в процесс разработки новых членов команды, ведь разделенный на отдельные функциональные части код легче читать и понимать.
* Свобода в выборе технологического стека. В монолитной архитектуре, где сервисы тесно связаны, важно поддерживать согласованность технологий, используемых в приложении. Они также должны быть хорошо совместимы друг с другом, что характерно не для всех языков программирования и средств разработки ПО. Слабо связанные микросервисы обеспечивают гораздо большую свободу в выборе стека технологий. Вплоть до того, что команда разработчиков может даже создавать отдельные сервисы с очень разным набором инструментов разработки.
* Доступность. Одним из самых больших преимуществ архитектуры микросервисов является то, что слабо связанные сервисы обеспечивают большую надежность приложения и более высокую доступность. Если одна служба выходит из строя, то это, как правило, не влияет на остальную часть приложения, которое продолжит функционировать для пользователей.
* Масштабируемость. Еще одним плюсом микросервисов является то, что этот подход хорошо подходит для масштабирования приложений. Модульность микросервисов позволяет легко добавлять новые функции прямо в работающее приложение. Хотя первоначальные затраты на архитектуру микросервисов выше, ее масштабирование может быть значительно дешевле, чем у монолитного приложения, благодаря сочетанию уже упомянутых сильных сторон.
* Производительность. Также за счет разделения служб и нагрузки между несколькими серверами возможно значительно повысить производительность.

**Минусы микросервисов**

* Сложность синхронизации. Распределенная система, такая как микросервисы, неизбежно создает дополнительную сложность, поскольку ее части необходимо синхронизировать таким образом, чтобы они могли работать как единая программная система. И если службы разделены между серверами, вам придется подготовить инфраструктуру для взаимодействия микросервисов.
* Сложность тестирования. С одной стороны, тестировать отдельные микросервисы проще, с другой, необходимость тестировать каждый сервис по отдельности может значительно усложнить приложение по мере его масштабирования. А необходимость тестировать и поддерживать связь между службами создает дополнительную сложность.
* Более высокие первоначальные затраты. Хотя попытки значительно масштабировать монолитное приложение могут оказаться сущим кошмаром, который приводит к резкому росту прямых затрат на разработку, архитектура микросервисов требует более высоких первоначальных затрат. Для каждого микросервиса требуется своя команда разработчиков (хотя одна команда может отвечать и за несколько). Кроме того, придется наладить и процесс автоматического тестирования и развертывания (CI/CD). Всё это приводит к более высоким первоначальным затратам.
* Потребность в DevOps. Распределенная система, такая как микросервисы, требует квалифицированной оркестровки, обычно с использованием Kubernetes и других инструментов и процессов DevOps. Это означает, что вам нанять хотя бы одного инженера DevOps, что также увеличит расходы.

Впрочем, в микросервисной архитектуре повышенные затраты и сложность компенсируются большей гибкостью и значительным улучшением производительности. Таким образом, создание приложения на основе микросервисов увеличивает первоначальные затраты, но взамен предлагает большую независимость и гибкость, что ускоряет циклы выпуска. И для многих современных компаний микросервисный подход стал скорее необходимостью, чем роскошью.

**Примеры микросервисов**

Микросервисная архитектура подходит для крупных и сложных приложений. В качестве примера можно привести приложение электронной коммерции, которое можно разбить на следующие микросервисы:

* фронтенд (пользовательский интерфейс);
* служба поиска, которая позволяет пользователям искать товары в базе данных с помощью поисковых запросов;
* служба связанных товаров, которая рекомендует альтернативные или дополнительные продукты, используя различные алгоритмы;
* сервис корзины покупок;
* сервис оплаты, который занимается обработкой платежей.

[**https://timeweb.cloud/blog/monolity-mikroservisy-besservernaya-arhitektura**](https://timeweb.cloud/blog/monolity-mikroservisy-besservernaya-arhitektura)

**3) Api GATEWAY**

Проблема когда фронтенду нужно 5 разных микросервисов опросить чтобы получить данные- например статус заказа, баланс на карте, данные карточки. На помощь приходит Api GATEWAY отдельный сервис точка входа для фронтенда. Серввис собирает нужный ответ – по опр урлу отправляется в нужные 5 сервисов. Согрегирует и вернет на фронт. Здесь соблюдается инкапсуляция

**4) Типы взаимодействия между микросервисов**

Рест, соап(xsd схема и xml), графКуЭль. JRPS. Можно исп очередь данных(тибко, ребит, кафка)

Очередь используем когда несколько экземпляров (асинхронное взаимодействие). Либо же когда нам не важно когда обработают сообщения, тоесть асинхрон. Кафка исп -когда нужен гарантии что сообщение будет доставлено.

Рест же – синхронное.

**5) Решение ошибки Hibernate LazyInitializationException в Spring**

Быстрый ответ

Суть решения проблемы LazyInitializationException заключается в обеспечении возможности загрузки коллекции в течение активной сессии Hibernate:

@Transactional: Эта аннотация, применённая к методу сервиса, позволяет поддерживать сессию открытой в течение всего времени выполнения метода, что обеспечивает доступность коллекции.

Java

Скопировать код

@Transactional

public Entity getEntity(Long id) {

Entity entity = repository.findById(id).orElse(null);

return entity; // Сессия активна, коллекция доступна.

}

Жадная загрузка: Возможно применение жадной загрузки коллекций через аннотации @OneToMany(fetch = FetchType.EAGER) или @ManyToMany(fetch = FetchType.EAGER) в классе сущности.

Инициализация Hibernate: Применимо использование Hibernate.initialize(entity.getCollection()) для явной инициализации коллекции во время активной сессии.

Теперь рассмотрим эти методы более детально.

Связь между аннотацией @Transactional и сессией

Правильное понимание того, как обработать LazyInitializationException, неразрывно связано с управлением границами транзакций. К примеру, рассмотрим аннотацию @Transactional. Spring поддерживает сессию открытой до окончания выполнения метода, помеченного этой аннотацией:

Java

Скопировать код

@Transactional

public User authenticate(String username) {

User user = userRepository.findByUsername(username);

// Коллекция ролей доступна к использованию в рамках метода.

return user;

}

Правильное размещение аннотации @Transactional определяет границы активной сессии, что очень важно при использовании ленивой загрузки связей в Hibernate.

Применение JOIN FETCH и DTO

Жадная загрузка иногда бывает неэффективной, поэтому рекомендуется использовать JOIN FETCH в JPQL-запросах для загрузки необходимых сущностей:

Java

Скопировать код

@Query("SELECT u FROM User u JOIN FETCH u.roles WHERE u.id = :id")

User findUserAndRolesById(@Param("id") Long id);

Благодаря проекции DTO можно извлекать только нужные данные, минуя полную инициализацию сущностей. Это уменьшает нагрузку на базу данных и мешает возникновению LazyInitializationException:

Java

Скопировать код

public UserDTO getUserDTO(Long id) {

User user = findUserAndRolesById(id);

// DTO позволяет избежать полной загрузки сущностей.

return new UserDTO(user);

}

Этот метод является более гибким и эффективным, сравнивая с возникающими проблемами при ленивой загрузке.

Оптимизация стратегий получения данных

Такие стратегии, как Open Session in View или использование "hibernate.enable\_lazy\_load\_no\_trans", могут сказаться на производительности приложения. Эти методы могут казаться удобными, но в перспективе они могут негативно влиять на производительность. Более продуманными и эффективными являются решения, основанные на понимании принципов управления сессиями в Hibernate.

Визуализация

Для понимания Hibernate LazyInitializationException можно провести аналогию с путешествием на поезде:

Мы заходим в поезд 🚂, который отправляется в путь с множеством остановок (сессиями). Каждый вагон (сущность) перевозит багажные отделения (ленивые коллекции).

Markdown

Скопировать код

[ 🚂 Двигатель (Сессия) ]

|---📦 Багажное отделение 1 (Ленивая коллекция)

|---📦 Багажное отделение 2 (Ленивая коллекция)

Проблема: Когда поезд останавливается и двигатель выключается, кто-то пытается открыть багажное отделение:

Markdown

Скопировать код

[ 🚂🛑 Двигатель ОСТАНОВЛЕН (Сессия закрыта) ]

|---🔒📦? Доступ к багажному отделению (LazyInitializationException)

Решение: Чтобы открыть отсеки, нужно либо оставить двигатель включенным (сессию активной), либо разблокировать отсеки, пока поезд стоит (заблаговременно инициализировать коллекции):

Markdown

Скопировать код

[ 🚂💡 Двигатель РАБОТАЕТ (Сессия активна) ]

|---🔓📦 Доступ к багажному отделению (Коллекция инициализирована)

Основная мысль: Для устранения LazyInitializationException нужно проинициализировать ленивые коллекции в течение активной сессии или предусмотреть их инициализацию до завершения сессии.

Погружение в суть вопроса

Производительность можно улучшить, следуя ряду рекомендаций:

Анализ производительности: Тщательный анализ использования коллекций позволяет определить, когда лучше использовать жадную, а когда -- ленивую загрузку.

Тестирование: Проведение тестов подтверждает, что транзакции организованы корректно и не вызывают LazyInitializationException.

Проведение инициализации в рамках транзакции: Инициализация внутри границ транзакции помогает избежать исключений.

Слоистая архитектура: Правильное структурирование архитектуры может уменьшить потребность в ленивой загрузке, делая код более надёжным и удобным для управления.

Баланс компромиссов

Важно понимать последствия выбора стратегии извлечения данных:

Долгосрочные решения: Важно учесть, как выбранная стратегия повлияет на производительность и поддерживаемость кода. Жадная загрузка может быстро решить проблему, но повышает потребление памяти.

Избегание антипаттернов: Понимание рисков, связанных с Open Session in View и подобными методами, помогает предотвращать проблемы с производительностью.

Гибкость JPQL запросов с JOIN FETCH: Выборочная подгрузка связанных сущностей не влияет на все запросы, что помогает избегать накладных расходов при полной загрузке всех сущностей.

Полезные материалы

Рекомендации Влада Михальцева по оптимизации производительности Hibernate — советы от эксперта по настройке производительности в Hibernate.

Как исправить LazyInitializationException – профессиональный подход — подробное объяснение ошибки LazyInitializationException и эффективные способы её решения.

Руководство пользователя Hibernate ORM 5.4.33.Final — описание стратегий извлечения данных в Hibernate, помогающих предотвращать ошибки.

Аннотация @Transactional в Spring Data JPA – DZone Java — обзор работы @Transactional в контексте Spring Data JPA от DZone.

Документация Spring Framework – Управление транзакциями — информация о транзакциях в Spring Framework.

https://sky.pro/wiki/java/reshenie-oshibki-hibernate-lazy-initialization-exception-v-spring/

**6) Понимание @Transactional в Spring Framework**

Понимание @Transactional в Spring Framework

Иногда возникает вопрос о том, что происходит при использовании аннотации @Transactional в Spring Framework. Данная аннотация является ключевой в транзакционном

Иногда возникает вопрос о том, что происходит при использовании аннотации @Transactional в Spring Framework. Данная аннотация является ключевой в транзакционном управлении Spring, и её понимание может помочь в написании более эффективного и безопасного кода.

Прокси-классы в Spring

Сначала стоит разобраться, что такое прокси-классы в Spring. Когда метод помечен аннотацией @Transactional, Spring создает прокси класс для этого метода. Этот прокси класс оборачивает вызов метода в транзакцию.

Прокси класс — это специально созданный класс, который служит промежуточным звеном между вызывающим кодом и целевым методом. Он перехватывает вызовы методов и выполняет дополнительную логику, прежде чем передать вызов целевому методу. В случае @Transactional, эта дополнительная логика заключается в управлении транзакцией.

Однако прямо взглянуть на созданный Spring прокси класс не выйдет, поскольку он создается динамически во время выполнения программы.

Внешние и внутренние вызовы методов

Ещё один интересующий момент — почему аннотация @Transactional работает только для внешних вызовов методов и не работает для внутренних (самовызовов) методов.

Это связано с тем, как работает механизм прокси. Прокси класс перехватывает только внешние вызовы методов, приходящие через прокси. Если метод объекта вызывает другой метод этого же объекта напрямую (т.е. самовызов), то этот вызов обходит прокси, и, следовательно, не перехватывается им.

Иными словами, если внутри транзакционного метода вызывается другой транзакционный метод того же объекта, то этот внутренний вызов не будет обернут в новую транзакцию, а будет выполняться в рамках уже существующей транзакции.

Важно помнить об этих особенностях при работе с @Transactional, чтобы избегать непредвиденного поведения и ошибок в работе с транзакциями.

<https://sky.pro/media/ponimanie-transactional-v-spring-framework/>

Транзакционное Java: что это

Источник: <https://uchet-jkh.ru/i/tranzakcionnoe-java-cto-eto>

Transactional java — это концепция, которая позволяет обеспечить целостность данных в приложениях Java. Она базируется на концепции транзакций, которые представляют собой логические операции, выполняющиеся одним блоком кода. В рамках транзакции выполняется набор операций, и если хотя бы одна из них не удалась, то весь набор операций откатывается и возвращается в исходное состояние.

Transactional java предоставляет множество инструментов и аннотаций, с помощью которых можно определить, какие методы и операции должны выполняться в рамках одной транзакции. Одной из основных аннотаций является @Transactional, которая указывает, что метод должен быть выполнен в рамках одной транзакции.

Используя Transactional java, разработчики могут обеспечить целостность данных в своих приложениях. Это особенно важно в случаях, когда несколько операций должны быть выполнены одновременно и успешно, чтобы гарантировать целостность или консистентность данных. Например, при выполнении банковских операций, где изначальное состояние счета должно быть сохранено до успешного выполнения всех операций.

Содержание

Transactional Java: основные принципы и применение

Принципы Transactional Java

Применение Transactional Java в разработке

Вопрос-ответ

Что такое Transactional Java?

Зачем использовать Transactional Java?

Как использовать Transactional Java в своем Java-приложении?

Transactional Java: основные принципы и применение

Transactional Java — это подход к программированию, который позволяет обеспечить целостность данных в распределенных системах и обработке транзакций в рамках этих систем. Транзакции — это операции, которые должны быть выполнены как одно целое, либо все должны быть отменены в случае сбоя или ошибки.

Основные принципы Transactional Java:

Атомарность: все операции, входящие в транзакцию, должны быть выполнены либо не выполнены вовсе;

Согласованность: после завершения транзакции данные должны оставаться в согласованном состоянии;

Изолированность: выполнение транзакции не должно зависеть от параллельно выполняемых операций;

Надежность: в случае сбоя или отказа системы данные должны быть сохранены и транзакция должна быть восстановлена.

Применение Transactional Java:

Базы данных: использование транзакций позволяет обеспечить надежность и целостность данных при выполнении операций чтения и записи;

Финансовые приложения: транзакционность особенно важна для обработки финансовых операций, таких как переводы средств и платежи;

Сетевые приложения: использование транзакций позволяет обрабатывать запросы и обеспечивать целостность данных в распределенных системах, таких как многопользовательские онлайн-игры;

Электронная коммерция: в онлайн-магазинах и других системах электронной коммерции транзакционность позволяет обеспечить правильную обработку заказов и платежей.

Применение Transactional Java позволяет разработчикам создавать надежные и безопасные приложения, способные обрабатывать транзакции и поддерживать целостность данных. Этот подход особенно полезен для разработки систем с высоким уровнем нагрузки и распределенных систем, где требуется обмен данных между несколькими узлами.

Принципы Transactional Java

Transactional Java — это подход к управлению транзакциями в языке программирования Java. Транзакция – это единица работы, которая должна быть выполнена атомарно и цельно.

Основные принципы Transactional Java включают:

Атомарность: Транзакции в Java должны выполняться как единое целое или не выполняться вообще. Если одна часть транзакции завершается неудачно, то остальные изменения должны быть отменены и возвращены к исходному состоянию.

Согласованность: После выполнения успешной транзакции в Java все данные должны оставаться в последовательном и согласованном состоянии. То есть, если транзакция изменяет несколько элементов данных, то после ее выполнения все измененные элементы должны быть в согласованном состоянии.

Изолированность: Каждая транзакция в Java должна выполняться в изолированном окружении. Это означает, что изменения, внесенные одной транзакцией, не должны быть видимы другим транзакциям до завершения первой транзакции.

Надежность: Транзакции в Java должны быть надежными и не должны приводить к потере данных или нарушению целостности данных. Если транзакция завершается неудачно, она должна быть отменена и возможно восстановлена до состояния до начала транзакции.

Transactional Java обеспечивает надежный и согласованный способ управления транзакциями в приложениях на языке Java. Этот подход позволяет программистам создавать надежные и отказоустойчивые системы, где важно, чтобы операции выполнялись без ошибок и сохраняли целостность данных.

Применение Transactional Java в разработке

Transactional Java является подходом в разработке ПО, который позволяет управлять транзакциями баз данных с использованием Java-кода. Транзакция представляет собой набор операций, которые должны быть выполнены вместе как единое целое и иметь либо полностью выполненными, либо полностью не выполненными.

Применение Transactional Java в разработке приносит следующие преимущества:

Целостность данных: Transactional Java позволяет обеспечить целостность данных, гарантируя, что все операции внутри транзакции будут выполнены либо полностью, либо не выполнены вовсе. Это гарантирует, что база данных находится в согласованном состоянии.

Управление конкурентностью: Transactional Java автоматически управляет конкурентным доступом к данным, предотвращая конфликты и позволяя одновременно выполняться нескольким транзакциям, обеспечивая при этом целостность данных.

Атомарность операции: Все операции внутри транзакции либо выполняются полностью, либо не выполняются вовсе. Если одна из операций не может быть выполнена, все предыдущие операции откатываются, возвращая базу данных в исходное состояние.

Управление исключениями: Transactional Java предоставляет механизмы для обработки исключительных ситуаций, которые могут возникнуть во время выполнения транзакции. Это позволяет разработчикам управлять ошибками и восстанавливать базу данных в случае необходимости.

В разработке Transactional Java используется с использованием различных фреймворков и библиотек. Некоторые из популярных фреймворков для работы с Transactional Java включают Spring, Java Transaction API (JTA) и Hibernate.

Использование Transactional Java в разработке обеспечивает надежное и безопасное выполнение операций с базой данных, гарантирует целостность данных и обеспечивает управление конкурентным доступом. Это делает Transactional Java важным инструментом для разработки надежных и эффективных приложений.

Источник: <https://uchet-jkh.ru/i/tranzakcionnoe-java-cto-eto>

7) По умолчанию FetchType в Hibernate: One-to-One и One-to-Many

Быстрый ответ

Java

Скопировать код

@OneToOne // По умолчанию: fetch = FetchType.EAGER

private RelatedEntity otoEntity;

@ManyToOne // По умолчанию: fetch = FetchType.EAGER

private RelatedEntity mtoEntity;

@OneToMany // По умолчанию: fetch = FetchType.LAZY

private Set<RelatedEntity> otmEntities;

В **Hibernate** для отношений типа **@OneToOne** и **@ManyToOne** используется немедленная загрузка (иначе говоря, применяется стратегия жадной загрузки). Для отношений типа **@OneToMany** вместо этого используется отложенная загрузка (данные запрашиваются по мере необходимости).

**Принципы работы с типами загрузки**

Понимание последствий выбора между жадной (**EAGER**) и отложенной (**LAZY**) стратегиями загрузки крайне важно. Если вам важно как можно скорее получить доступ ко всем связанным сущностям – выбирайте стратегию **EAGER**. Если же вы хотите сэкономить ресурсы при первоначальной загрузке и готовы запрашивать данные по мере их нужды – воспользуйтесь стратегией **LAZY**.

**Переопределение стратегий загрузки**

Hibernate предоставляет возможность переопределить стратегии загрузки, если вас не устраивают установленные по умолчанию:

Java

Скопировать код

@OneToMany(fetch = FetchType.EAGER)

private Set<RelatedEntity> otmEntities;

**Работа с прокси**

При использовании отложенной загрузки Hibernate применяет прокси, представляющие сущности. Это помогает сэкономить ресурсы, в частности, в случае редко используемых отношений One-To-One и Many-To-One.

**Бережное отношение к выбору стратегии загрузки**

Тщательно анализируйте и оценивайте выбранную стратегию загрузки, чтобы максимально повысить производительность вашего приложения. По умолчанию используйте **LAZY загрузку** и переключайтесь на **EAGER** только в случае наличия веской необходимости.

**Настройка стратегий загрузки**

Выбор подходящей стратегии загрузки — ключевой момент оптимизации. Изучите опции fetch=JOIN и fetch=SELECT, чтобы лучше понять, как можно настроить загрузку данных.

**Выбор между 'join' и 'select'**

При использовании fetch='JOIN' возможно принудительное обращение Hibernate к загрузке связанных сущностей даже при значении lazy='true'. С другой стороны, стратегия fetch='SELECT' позволяет обращаться к связанным сущностям отдельно, что подходит для отложенной загрузки.

**Изучение тонкостей работы с Hibernate**

Для глубокого понимания работы Hibernate рекомендуется изучать и пользовательское руководство, и официальную документацию.

**Понимание стандартных стратегий загрузки**

По умолчанию для коллекций используется стратегия загрузки select, для одиночных ассоциаций предпочтительнее join.

**Визуализация**

Отношений в Hibernate можно сравнить с соседями в жилом комплексе:

OneToOne: 🏠 = 🏠

ManyToOne: 🏘️ → 🏠

OneToMany: 🏠 → 🏘️

Стандартные типы загрузки:

| Связь | Загрузка по умолчанию |

| ------------ | --------------------- |

| One-to-One | 🚪 LAZY |

| Many-to-One | 🏃 EAGER |

| One-to-Many | 🚪 LAZY |

Легенда:

🏠 Это сущность

🏘️ Это коллекция сущностей

🚪 Загрузка по требованию (LAZY)

🏃 Немедленная загрузка (EAGER)

Шпаргалка для разработчика

EAGER против LAZY: выбор имеет значение

Выбор между жадной и отложенной загрузкой влияет на производительность приложения. Основывайтесь на требованиях вашего проекта и на их влиянии на производительность при выборе стратегии.

Предотвращение проблем N+1 и LazyInitializationException

При использовании EAGER стоит остерегаться проблемы избыточных запросов, а LAZY может вызвать исключение LazyInitializationException, если сессия Hibernate будет закрыта до загрузки данных.

Все возможности Hibernate в вашем распоряжении

Максимально используйте возможности настройки стратегии загрузки данных путем применения пакетной загрузки и других функций, предоставляемых Hibernate.

Не забывайте обновлять свои знания

Следите за последними обновлениями Hibernate, чтобы оставаться в курсе и обновлять свои навыки.