Разработка серверного ПО. Лекция 3. Spring

Spring — не фреймворк, а целая экосистема для разработки программного обеспечения, которая приобрела популярность для разработки серверного ПО, но также можно использовать для разработки мобильных и десктопных приложений.

Список проектов Spring

Spring Core — это та часть фреймворка Spring, которая обеспечивает фундаментальные механизмы его интеграции в приложение. Spring работает по принципу инверсии управления (inversion of control, IoC): вместо того чтобы приложение само контролировало свое выполнение, управление передается некоторому другому программному обеспечению — в данном случае фреймворку Spring. Посредством системы настроек мы предоставляем фреймворку инструкции о том, как распоряжаться написанным нами кодом, что и определяет логику работы приложения. Именно это и подразумевается под «инверсией» в аббревиатуре IoC: мы не позволяем приложению управлять собственным выполнением посредством его же кода или использовать зависимости. Вместо этого мы передаем фреймворку (зависимости) управление приложением и его кодом.

Создание бинов и добавление в контекст Spring

Первое, что нужно освоить в Spring, — это добавление в контекст Spring экземпляров объектов (называемых бинами). Контекст Spring можно представить как корзина, в которое помещают экземпляры, чтобы затем Spring мог ими управлять. Spring «видит» только те экземпляры, которые были добавлены в контекст.

Есть три способа добавления бинов в контекст Spring: с помощью аннотации `@Bean` и стереотипных аннотаций, а также программно.

- Посредством аннотации `@Bean` в контекст Spring в качестве бина можно добавить экземпляр (или даже несколько экземпляров) объекта любого типа. Такой подход является более гибким, чем стереотипные аннотации. Однако он требует написания большего кода, поскольку необходимо создавать в классе конфигурации отдельный метод для каждого независимого экземпляра, включаемого в контекст. Пример в `com.sibsutis.study.Basics.BeansAnnotation`.

- С помощью стереотипных аннотаций можно создавать бины только для классов, определенных в приложении. (Для этого применяются специфические аннотации, такие как `@Component`.) При подобном подходе требуется писать меньше кода, благодаря чему конфигурацию легче читать. Данный способ предпочтительнее, чем `@Bean`, для тех классов, которые вы создали сами и можете снабдить аннотациями. Пример в `com.sibsutis.study.Basics.StereotypeAnnotation`.

- Метод registerBean() позволяет реализовать собственную логику добавления бинов в контекст Spring. Пример в `com.sibsutis.study.Basics.RegisterBean`.

При реализации приложения часто приходится ссылаться из одного объекта на другой. Таким образом объект при выполнении своих функций может делегировать некоторые действия. Для реализации этого поведения необходимо установить связи между бинами в контексте Spring. Есть три способа установки связей между бинами:

- прямая ссылка на метод с аннотацией @Bean (который создает первый бин) из метода, который создает второй бин. Spring понимает, что вы ссылаетесь на бин, размещенный в контексте, и, если такой бин уже существует, не вызывает метод создания бина еще раз, а возвращает ссылку на тот, который уже есть;
- определение параметра для метода с аннотацией @Bean. Когда Spring встречает метод с аннотацией @Bean и у этого метода есть параметр, то фреймворк ищет в контексте бин идентичного с параметром типа и подставляет в качестве значения параметра данный бин;
- использование аннотации @Autowired следующими способами:
    - добавлением аннотации @Autowired к полю класса, чтобы Spring внедрил в это поле бин из контекста. Данный вариант часто встречается в примерах и демонстрациях концепции;
    - добавлением аннотации @Autowired к конструктору, который Spring будет вызывать при создании бина. Тогда Spring внедрит другие бины из контекста в виде параметров конструктора. Такой способ чаще всего применяется в реальных приложениях;
    - добавлением аннотации @Autowired к сеттеру атрибута, в который Spring должен внедрить бин из контекста. Этот способ редко используется на практике.

Всякий раз, когда Spring предоставляет значение или ссылку через атрибут класса, метод или параметр конструктора, это делается посредством DI построенной по принципу IoC.

При создании двух бинов, зависящих друг от друга, возникает циклическая зависимость. Spring не может создать бины с циклической зависимостью, и при выполнении такого приложения возникает исключение. Работая с бинами, убедитесь, что между ними не возникает циклической зависимости. Если в контексте Spring есть несколько бинов одного типа, фреймворк не может взять из них один для внедрения самостоятельно. Поэтому нужно указать Spring, какой из экземпляров нужно выбрать:

- с помощью аннотации @Primary, благодаря которой при внедрении зависимости один из бинов выбирается по умолчанию;
- присвоив бинам имена и внедряя их по именам с помощью аннотации @Qualifier.

Внедрение зависимостей для абстракций

Репозиторий — объект который взаимодействует с хранилищем данных (repository, data access object, DAO).

Прокси — объект взаимодействующий с внешней по отношению к приложению сущностью.

POJO — это простой объект без зависимостей, описываемый только своими атрибутами и методами.

Предположим, нам нужно создать приложение, с помощью которого команда работников управляет своими обязанностями. Одна из функций продукта — возможность оставлять заметки к задачам. Когда пользователи публикуют комментарий, он где-то сохраняется (например, в базе данных) — и приложение отправляет письмо по электронному адресу, указанному в конфигурации приложения. Пример без ведрения зависимостей в `com.sibsutis.study.DIAbstraction.WithoutDIExample`.

При использовании абстракции с внедрением зависимостей Spring знает, где искать бин, который ее реализует.

Если нужно создать экземпляр класса и поместить его в контекст Spring, то такой класс сопровождается стереотипной аннотацией. Стереотипные аннотации никогда не применяются к интерфейсам.

Для компонентов с обязанностями сервиса вместо стереотипной аннотации @Component можно применить аннотацию @Service. Аналогичным образом для компонентов с обязанностью репозитория вместо @Component можно использовать аннотацию @Repository. Таким образом мы явно обозначаем обязанность компонента, благодаря чему структура классов становится более понятной и удобной для чтения. Пример в `com.sibsutis.study.DIAbstraction.WithDIExample`.

Области видимости и жизненный цикл бинов

В Spring предусмотрено несколько вариантов создания бинов и управления их жизненным циклом — в мире фреймворка эти способы называются областями видимости (scopes).

Одиночная область видимости — это область видимости, которую бин Spring получает по умолчанию. Именно ее мы использовали до сих пор.

Spring создает одиночные бины при загрузке контекста и присваивает им имена (иначе, ID бина). Эта область видимости называется одиночной, поскольку при обращении к конкретному бину мы всегда получаем один и тот же экземпляр. В контексте Spring может существовать несколько экземпляров одного типа, но с разными именами, а само понятие «одиночный» означает уникальность имени, но не уникальность наличия в пределах приложения.

Рассмотрим пример добавления бинов при помощи аннотации `@Bean`:

```
@Configuration
public class ProjectConfig {

    @Bean
    public Cat cat() {
        var cat = new Cat();
        cat.setName("Barsik");
        return cat;
    }

    @Primary
    @Bean
    public Cat cat2() {
        var cat = new Cat();
        cat.setName("Kuzya");
        return cat;
    }

    @Bean
    public Cat cat3() {
        var cat = new Cat();
        cat.setName("Vaska");
        return cat;
```

```
    }
}
```

При получении бина по его имени, будет возвращена ссылка на один и тот же экземпляр.

Поскольку область видимости одиночного бина предполагает, что доступ к экземпляру объекта имеют несколько компонентов приложения, главное, что следует учитывать, — эти бины должны быть неизменяемыми. Как правило, в реальных приложениях (таких как веб-приложения) операции выполняются в нескольких потоках. Следовательно, разные потоки могут иметь доступ к одному экземпляру объекта. Если эти потоки изменят экземпляр, то попадут в состояние гонки (race condition).

Если нужны изменяемые одиночные бины (атрибуты которых можно модифицировать), необходимо сделать их согласованными (для этого обычно используют синхронизацию потоков). Но одиночные бины не рассчитаны на то, чтобы быть конкурентными. Они, как правило, используются для определения базовой структуры классов и делегирования обязанностей между собой. Технически синхронизация одиночных бинов возможна, но это не является хорошим тоном в программировании. Синхронизация потоков для конкурентности экземпляра резко снижает производительность приложения. Именно поэтому внедрение зависимостей через конструктор предпочтительнее, чем через поле — в этом случае можно пометить поле как `final`.

Как правило, Spring создает одиночные бины при инициализации контекста — таково поведение фреймворка по умолчанию (его также называют немедленным созданием экземпляров), но если добавить к бину аннотацию `@Lazy`, то бин будет создаваться только при попытке получить доступ к бину.

С точки зрения производительности лучше заранее создать все экземпляры, которые должны находиться в контексте (немедленно), иначе при делегировании функций от одного бина другому фреймворк должен будет сделать еще несколько дополнительных проверок. К тому же, если что-либо пойдет не так и фреймворк не сможет создать бин, это будет заметно при запуске приложения. При «ленивом» варианте проблема может быть обнаружена только в уже работающем приложении, когда возникнет необходимость в создании конкретного бина.

Но если пользователю при запуске приложения каждый раз требуется только малая часть его возможностей, то лучше создавать бины по мере необходимости. Но по умолчанию следует использовать немедленное создание.

Каждый раз, когда запрашивается ссылка на прототипный бин, Spring создает новый экземпляр объекта. В случае прототипных бинов Spring создает сам объект и не управляет им. Фреймворк управляет только типом объекта и создает новый экземпляр всякий раз, когда какой-либо объект запрашивает ссылку на этот бин.

Для изменения области видимости бина понадобится новая аннотация — `@Scope` (`@Scope(BeanDefinition.SCOPE_PROTOTYPE)`). Она ставится вместе с `@Bean` перед методом, объявляющим бин. При объявлении бина с помощью стереотипных аннотаций `@Scope` вместе со стереотипными аннотациями ставится перед классом, в котором объявляется бин.

Применение прототипных бинов решает проблемы конкурентности, поскольку каждый поток получает свой экземпляр, так что определение нескольких прототипных бинов не создает проблем.

Аспекты в Spring

Аспект — это просто часть логики, которую фреймворк реализует, вызывая определенные, выбранные методы. При разработке аспекта нужно определить следующее:

- какой код должен выполнять Spring при вызове данных методов. Именно это и называется аспектом;
- когда приложение должно выполнять логику аспекта (например, перед, после или вместо вызова метода). Это называется советом;
- выполнение каких методов должен перехватывать фреймворк и реализовывать аспект. Это называется срезом.

В терминологии аспектов также встречается понятие «точка соединения» — событие, запускающее выполнение аспекта. Но в Spring это событие всегда одно и то же — вызов метода.

При вплетении аспекта Spring предоставляет ссылку не на сам бин, а на прокси-объект, который перехватывает вызовы метода и управляет логикой аспекта.

Чтобы создать аспект, нужно выполнить следующие действия.

1. Активировать механизм аспектов в Spring-приложении, поставив перед классом конфигурации аннотацию @EnableAspectJAutoProxy.
2. Создать новый класс и внедрите перед ним аннотацию @Aspect. Затем, используя либо аннотацию @Bean, либо одну из стереотипных аннотаций, добавить бин этого класса в контекст Spring.
3. Определить метод, в котором будет реализована логика аспекта, и с помощью аннотаций советов сообщить Spring, где и какие методы следует перехватывать.
4. Реализовать логику аспекта.

Документация по AspectJ

Пример в `com.sibsutis.study.AOP.SimpleExample`.

Аспекты позволяют влиять на выполнение перехватываемого метода следующими способами:

- модифицируя значения параметров, передаваемых методу;
- изменяя значение, возвращаемое перехваченным методом и передаваемое вызывающему методу;
- выбрасывая исключение и передавая его вызывающему методу либо перехватывая и обрабатывая исключение, которое выдает перехваченный метод.

Пример с перехватом по аннотации в `com.sibsutis.study.AOP.AOPWithCustomAnnotation`.

Кроме `@Around`, в Spring есть следующие аннотации советов:

- `@Before` — вызывает метод, определяющий логику аспекта, перед выполнением перехватываемого метода;
- `@AfterReturning` — вызывает метод, определяющий логику аспекта, после успешного завершения перехватываемого метода. При этом значение, возвращаемое перехваченным методом, передается методу аспекта в качестве параметра. Если перехваченный метод выдаст исключение, метод аспекта не вызывается;
- `@AfterThrowing` — вызывает метод, определяющий логику аспекта, если перехваченный метод выдает исключение. Экземпляр исключения передается методу аспекта в качестве параметра;
- `@After` — вызывает метод, определяющий логику аспекта, после выполнения перехватываемого метода, независимо от того, завершится ли перехваченный метод успешно или выдаст исключение.

Данные аннотации советов используются так же, как и `@Around`. Они сопровождаются выражением среза AspectJ, позволяющим вплести логику аспекта в выполнение соответствующих методов. При этом методы аспектов не получают параметр ProceedingJoinPoint, и поэтому с ними нельзя выбирать, в какой момент делегировать управление перехваченному методу. Теперь это событие происходит в соответствии с выбранной аннотацией (например, в случае `@Before` перехватываемый метод всегда выполняется после логики аспекта).

По умолчанию Spring не гарантирует, что каждый раз при запуске приложения цепочка аспектов будет выполняться в одном и том же порядке. Если последовательность неважна, достаточно создать аспекты и позволить фреймворку задействовать их по своему усмотрению. Если же порядок выполнения аспектов имеет значение, то можно воспользоваться аннотацией `@Order`. Эта аннотация получает порядковый номер (число), соответствующий очередности выполнения данного аспекта в цепочке. Чем меньше это число, тем раньше заработает аспект. В случае двух одинаковых чисел очередность выполнения снова не будет определена.

Spring Boot и Spring MVC

Большинство продуктов, разрабатываемых на основе Spring, — это веб-приложения.

Любое веб-приложение состоит из двух частей, таких как.

- клиентская часть — то, с чем непосредственно взаимодействует пользователь. Представлена веб-браузером. Браузер отправляет запросы на веб-сервер, получает от него ответы и предоставляет пользователю способ взаимодействия с приложением. Клиентскую часть приложения еще называют фронтендом;
- серверная часть — получает запросы от клиента и отправляет ему в ответ данные. Серверная часть содержит логику, которая обрабатывает и иногда сохраняет запрашиваемые клиентом данные перед тем, как отправить ему ответ. Серверную часть также называют бэкендом.

Бэкенд обслуживает сразу нескольких клиентов на конкурентной основе. Множество людей могут использовать одно и то же веб-приложение в одно и то же время на разных платформах. Пользователи могут открывать приложение в браузере на компьютере, в телефоне, на планшете и т.п.

Веб-приложения по способу функционирования бывают разные.

1. Приложения, в которых бэкенд в ответ на запрос клиента дает полностью готовое представление. В таких продуктах браузер получает данные от бэкенда и сразу выводит их на экран пользователя.

2. Приложения с разделением обязанностей между фронтендом и бэкендом. В таких продуктах бэкенд предоставляет только первичные данные. Получив ответ от бэкенда, браузер не выводит эти данные сразу на экран, а запускает некое особое клиентское приложение, которое получает ответы от сервера, обрабатывает данные и сообщает браузеру, что именно нужно вывести.

Один из важнейших моментов, который следует учитывать, — это коммуникация между клиентом и сервером. Веб-браузер обменивается данными с сервером по сети через протокол передачи гипертекста — Hypertext Transfer Protocol (HTTP), — который аккуратно регламентирует данный процесс.

Поэтому необходимо что-то, что может понимать HTTP-запросы и HTTP-ответы, принимать и отправлять их. Это «что-то» называется контейнером сервлетов (для которого еще иногда употребляют термин «веб-сервер»). Одним из самых популярных контейнеров сервлетов считается Tomcat (альтернативы Jetty, Jboss, Payara).

Сервлет — это просто объект Java, который напрямую взаимодействует с контейнером сервлетов. Когда контейнер получает HTTP-запрос, он вызывает метод объекта сервлета, которому передает этот запрос в виде параметра. Сервлет отправляет ответ клиенту, сделавшему запрос, а метод получает параметр, представляющий собой HTTP-ответ.

Контейнер сервлетов (Tomcat) регистрирует несколько экземпляров сервлетов. У каждого из них есть свой путь. Когда клиент отправляет запрос, Tomcat вызывает метод сервлета, связанного с путем, по которому клиент сделал запрос. Сервлет получает значения запроса и формирует ответ, который затем Tomcat возвращает клиенту.

После того как объект сервлета определен в Spring-приложении и зарегистрирован в контейнере сервлетов, и Spring, и контейнер знают о его существовании и могут им управлять. Контейнер вызывает сервлет для любого клиентского запроса, чтобы сервлет управлял этим запросом и формировал ответ.

Список самых важных опций Spring Boot и тех возможностей, которые они открывают.

- Упрощенное создание проектов. Сервис инициализации проекта создает пустую, но полностью сконфигурированную заготовку приложения.

- Диспетчеры зависимостей. Spring Boot группирует зависимости, используемые для тех или иных целей, и объединяет их с помощью диспетчеров. Больше не нужно выяснять ни полный список зависимостей, которые необходимо добавить в проект с определенной целью, ни то, какие их версии нужны для обеспечения совместимости.

- Автоконфигурация на основе зависимостей. На основе зависимостей, добавленных в проект, Spring Boot создает несколько конфигураций по умолчанию. Вместо того чтобы задавать все эти конфигурации самому, теперь достаточно изменить отдельные из них, которые не соответствуют вашим потребностям, причем для этого приходится писать меньше кода (или не приходится его писать вообще).

Операции, которые нужно выполнить, чтобы создать проект Spring Boot с помощью start.spring.io:

1. Откройте в браузере страницу start.spring.io.

2. Укажите свойства проекта (язык, версию, систему сборки и т. п.).
3. Укажите зависимости, которые нужно добавить в проект.
4. Загрузите упакованный проект с помощью кнопки GENERATE.
5. Распакуйте проект и откройте его в IDE.

После того как вы кликните на кнопке GENERATE, браузер загрузит ZIP-архив с проектом Spring Boot.

Spring MVC

Последовательность обработки запроса:

1. Клиент делает HTTP-запрос.
2. Tomcat получает HTTP-запрос клиента и вызывает компонент сервлета, ответственный за его обработку. В случае Spring MVC это сервлет, указанный в конфигурации Spring Boot. Его мы называем сервлетом-диспетчером (dispatcher servlet).
3. Сервлет-диспетчер является точкой входа в веб-приложение Spring. Tomcat вызывает сервлет-диспетчер для всех полученных HTTP-запросов. Его обязанность — управлять запросами в Spring-приложении. Диспетчер должен найти действие контроллера, которое вызывается в ответ на данный запрос, и определить, что следует вернуть клиенту. Этот сервлет еще называют фронтальным контроллером или единой точкой входа (front controller).
4. Сервлет-диспетчер ищет действие контроллера для ответа на запрос. Чтобы определить, какой контроллер нужно вызвать, сервлет делегирует управление компоненту, который называется картой обработчиков (handler mapping). Она находит действие контроллера, связанное с запросом через аннотацию `@RequestMapping`.

5.Когда нужное действие контроллера будет найдено, сервлет-диспетчер вызывает его. Если на карте обработчиков не нашлось ни одного действия, связанного с запросом, приложение выдает клиенту HTTP-статус 404 — Not Found. Если же действие есть, контроллер возвращает сервлету-диспетчеру имя страницы, которую нужно отобразить. Ее принято называть представлением (view).

6.В этот момент сервлет-диспетчер должен найти представление, имя которого было предоставлено контроллером, получить содержимое представления и отправить его клиенту. Сервлет-диспетчер делегирует обязанность получить содержимое компоненту, который называется арбитром представлений (view resolver).

7.Сервлет-диспетчер возвращает сгенерированное представление клиенту в виде HTTP-ответа.