

## Оглавление

Аспектно-ориентированное программирование.....	1
Перехватчики (Interceptors).....	2
Методы перехвата (Intercepting Methods).....	3
Interceptor Class.....	4
Интерфейс javax.interceptor.InvocationContext.....	5
Применение перехватчиков.....	6
Аннотирование перехватываемых методов.....	6
Использование для перехвата XML.....	6
Перехватчики по-умолчанию (default interceptors).....	7
Очередность применения множества перехватчиков.....	7
Отмена перехвата.....	8
Перехватчики и внедрение (Interceptors & Injection).....	9
Перехват событий жизненного цикла.....	10
Самопальная аннотация внедрения (Custom Injection Annotation).....	10
Обработка исключений (Exception Handling).....	13
Отмена вызова метода.....	13
Перехват и перевыброс исключений.....	14
Жизненный цикл перехватчика.....	15
@AroundInvoke методы в классе EJB.....	16
Перехватчики CDI.....	16
Связывание с перехватчиком.....	16
Приоритеты.....	17
Ругательства.....	18
Литература.....	19

## Аспектно-ориентированное программирование

АОП — методика программирования, основанная на отделении от специфической прикладной бизнес-логики общесистемной функциональности. Для такой общей неспецифичной функциональности применяют термин «сквозная функциональность» (crosscutting concerns), т.е. функциональность, насквозь пронизывающая бизнес-логику. Блоки кода, внедряемые в приложение и реализующие такую сквозную функциональность называют аспектами.

Если ООП предполагает проектирование системы пляской от данных, оборачивая их необходимой функциональностью в классы, то АОП предполагает вычленение и группировку общей функциональности этих данных в отдельные общесистемные аспекты. В качестве примера можно привести подсчет денег: можно брать из кучи монету и прибавлять ее номинал в общую сумму, а можно сначала разложить монеты по номиналу на кучки, посчитать размеры этих кучек и просуммировать их, умножив на номинал. В первом случае имеет место интегрирование по Риману, близкое к ООП, во втором — интегрирование по Лебегу, сходное с АОП.

Примерами сквозной функциональности являются журналирование обращений к системе, учетный контроль, профилирование программного кода, сбор статистики и т.п.

Примерами аспектов также могут выступать предопределенные (часто именованные) свойства фреймворков, вроде аспектов в JSF(?), влияющие на всю работу приложения, использующего эти фреймворки.

Сквозная функциональность пронизывает прикладную логику через «сквозные» фрагменты кода. В EJB это границы методов, участки кода, определяемые событиями жизненного цикла. Реализация этого пронизывания делается при помощи перехвата вызовов методов и событий жизненного цикла EJB.

АОП использует следующие понятия:

- задача/функциональность (concern) — та задача, которую надо выполнить или та функциональность, которую надо реализовать. Например, задача по занесению в журнал всех вызовов или реализация функциональности по округления всех входящих чисел до целых.
- Совет (advice) — блок кода, который решает задачу или реализует функциональность.
- Точка сопряжения (pointcut) — место в программе, по достижению которого должен выполняться совет.
- Аспект (aspect) — совокупность точки сопряжения и совета. Т.е. внедряемый в программу блок кода, решающий требуемую задачу или реализующий заданную функциональность.

В этих терминах АОП представляет из себя вплетение в точки сопряжения приложения советов при помощи соответствий, заданных аспектами. EJB Interceptors реализуют наиболее общее АОП. Перехватчик представляет из себя совет, окружающий точку сопряжения — перед и после вызова метода, точки возникновения событий жизненного цикла, точки перехвата исключений. Существует более сложный и функциональный фреймворк AspectJ.

## **Перехватчики (Interceptors)**

В EJB-архитектуре бизнес-логика доступна из сессионных бинов и MDB. Существительными являются entity-объекты, а службы вроде транзакций или безопасности обслуживаются приложением как настраиваемые аспекты. Однако существуют пользовательские задачи с уникальными требованиями, для обслуживания которых мало стандартной спецификации. Для таких задач и придуманы перехватчики, представленные спецификацией фреймворка Interceptors 1.1 в EJB 3.1 (JEE6).

Перехватчик — объект, который может быть внедрен в вызовы методов и события ж.ц. сессионных бинов и MDB. Перехватчик инкапсулирует некое общее поведение, которое может пронизывать множество частей приложения. Это поведение в явном виде загрязняло бы бизнес-логику приложения.

Перехватчики стали абсолютно новым направлением в EJB 3, определившим

современный подход к кластеризации (модульности) приложения и даже в расширении функциональности EJB-контейнера.

## Методы перехвата (Intercepting Methods)

Рассмотрим пример общеиспользуемого вспомогательного кода приложения, моделирующего работу online tv-станции. TunerEJB является ее частью, обрабатывающей клиентские запросы, его метод `getChannel` возвращает в ответ на запрос канала ссылку на входной поток с видеоконтентом.

```
Public interface TunerLocal {  
    InputStream getChannel(int channel){...}  
    ...  
}
```

Босс (заказчик) может потребовать регистрацию всех запросов на период запуска продукта. В итоге придется вставлять в бизнес-логику посторонние фрагменты кода.

```
@Override  
public inputStream getChannel(int channel){  
    ...  
    final Method method;  
    method = TunerLocal.class.getMethod("getChannel",int.class);  
    final Object[] params = new Object[] {channel};  
    cache.put(myInvocationDo(method,params));  
    ...  
}
```

В вышеприведенном примере продемонстрирована реализация регистрационного учета клиентских запросов. Фрагмент кода при помощи рефлексии пишет в некий общий кэш информацию об окружении вызова — методе и его параметрах. Для многократного использования фрагмент можно оформить в виде отдельной процедуры — утилиты, но при этом сохраняется необходимость явного вызова этой утилиты из всех методов, где нужно такое поведение, что засоряет бизнес-логику, особенно в случае, когда требуется менять связанные с этим поведением настройки. (В данном примере это может быть запрет доступа к некоторым методам некоторых клиентов в некоторые периоды и т.п.).

В целом пример содержит несколько изъянов проектирования:

- метод `getChannel` загрязнен кодом, не имеющим отношения к бизнес-логике приложения. Это не только заставляет писать этот код, но и делает всю программу трудночитаемой.
- трудно включать/выключать регистрационный учет, т.к. учетный код надо убирать из приложения, после чего перекомпилировать программу.
- учетная бизнес-логика является шаблонной и может быть задействована во многих частях приложения, ее модификация может привести к необходимости переделки и пересборки множества классов.

Перехватчики определяют более чистое разделение бизнес-логики, сквозное для множества модулей системы. Они обеспечивают механизм инкапсуляции специфической логики и легкий путь для ее применения в нужных методах без написания трудно-читаемого кода. Перехватчики обеспечивают структуру для специфического поведения так, что оно может быть легко расширено и выполнено в рамках одного класса. В итоге перехватчики обеспечивают простой, настраиваемый механизм, для применения своего поведения в любом месте.

## Interceptor Class

В качестве перехватчика метода может выступать любой простой java-класс с одним и только одним методом, имеющим следующую сигнатуру:

```
@javax.interceptor.AroundInvoke  
Object exMethod(javax.interceptor.InvocationContext invocation) throws Exception
```

*Rem:* *@AroundInvoke метод не может иметь модификаторы static и final (?)*

Этот метод оборачивает перехватываемый (целевой) метод и вызывается в одном стеке java-вызовов с ним, т.е. в той же транзакции, в едином контексте безопасности, с одним и тем же ENC JNDI. Параметр InvocationContext – обобщенное представление вызываемого бизнес-метода. Из него можно получить такую информацию, как ссылку на экземпляр целевого бина, ссылку на перехватываемый метод этого бина в обобщенном виде java.lang.reflect.Method, набор параметров запроса в виде массива Object.

Пример учетчика-перехватчика, переделка предыдущего примера.

```
public class RecordingAuditor {  
    ...  
    @AroundInvoke  
    public Object audit(final InvocationContext ctx) throws Exception {  
        ...  
        invocations.add(ctx);  
        ...  
        Object result = ctx.proceed();  
        // Код обработки результата java-вызова  
        ...  
        // Возврат текущего результата java-вызова  
        return result;  
    }  
    ...  
}
```

В отличие от прошлого примера, обобщенный код учета тут полностью изолирован от бизнес-логики (будет связан с ней только при помощи метаданных). В дополнение к этому, автоматически передаваемый контейнером в перехватчик параметр ctx сразу дает

информацию о запросе, что избавляет от написания самопальных вставок кода, как в `getChannel()` из прошлого примера.

Метод `InvocationContext.proceed()` вызывает следующий перехватчик, принадлежащий тому же стеку вызова. Если таких перехватчиков больше нет, то EJB-контейнер вызывает сам перехваченный метод бина. Код перехватчика после `InvocationContext.proceed()` может обрабатывать текущий результат вызова, возвращаемый целевым методом и уже обработанный более близкими к нему перехватчиками из стека вызова.



*Rem:* если не вызвать `ctx.proceed()` или метод перехвата выбросит раннее исключение, то стек вызова прервется и перехваченный целевой метод также не будет вызван.

В спецификации `Interceptors 1.2` был добавлен перехватчик конструктора:

```
@javax.interceptor.AroundConstruct
public Object exMethod(InvocationContext ctx) throws Exception
```

*Rem:* Также существуют перехватчики методов задержки `@javax.interceptor.AroundTimeout`, которые вмешивающиеся в работу методов временной задержки под управлением службы времени EJB. И перехватчики событий жизненного цикла.

## Интерфейс `javax.interceptor.InvocationContext`

- `Object getTarget()` дает ссылку на экземпляр вызываемого бина
- `Method getMethod()` дает ссылку на обобщенное представление вызова перехваченного метода бина
- `Object[] getParameters` дает массив параметров перехваченного запроса
- `Map<String, Object> getContextData` возвращает именованные данные, которыми могут обмениваться перехватчики единого стека вызова до его завершения
- `void setParameters(Object[] newArgs)` метод позволяет модифицировать параметры вызываемого метода
- `Object proceed() throws Exception` - передает обработку вызова другим перехватчикам или же EJB-контейнеру для последующего вызова перехваченного метода, возвращает текущий результат java-вызова (результат выполнения)

целевого метода, возможно уже обработанный более близкими к нему перехватчиками из этого стека).

- Object getTimer()
- getConstructor() дает ссылку на конструктор целевого класса, введен в Interceptors 1.2

## Применение перехватчиков

Один или более перехватчиков можно применить ко всем классам ejb-приложения (из одного deployment), ко всем методам отдельного ejb-класса или же к отдельному его методу. Применение перехватчиков описывается через @ или через xml-дескриптор развертывания.

### Аннотирование перехватываемых методов

Аннотация javax.interceptor.Interceptors может применяться как к отдельным методам, так и к целому бину сразу, т.е. ко всем его методам. Аннотация @Interceptors имеет только одно свойство Class[] value, в котором перечисляются все классы перехватчиков, которые должны перехватить вызов.

Ex: Применение перехватчика RecordingAuditor к реализации TunerEJB

```
@Stateless
@Interceptors(RecordingAuditor.class)
@Local(TunerLocal.class)
public class TunerBean implements TunerLocal {...}
```

В дополнение к общеклассовому перехвату из примера можно, например, ограничить доступ к методу getChannel при помощи другого перехватчика Channel2Restrict

```
...
@Override
@Interceptors(Channel2Restrict.class)
InputStream getChannel(int no) {...}
...
```

### Использование для перехвата XML

Недостатком аннотирования является необходимость правки и перекомпиляции исходных классов (если вообще есть доступ к исходникам) после каждого перепрофилирования поведения любого перехватываемого метода.

За исключением тех случаев, когда перехватчики являются неотъемлемой частью бизнес-логики, аннотации проигрывают описанию во внешнем xml с т.з. портируемости и гибкости настройки.

Спецификация ejb строго определяет разметку xml-дескриптора, поэтому настраивать перехват через xml достаточно просто.

Ex: перехватчики в дескрипторе развертывания

```
<ejb-jar ...>
...
<assembly-descriptor>
<interceptor-binding>
<ejb-name>TunerBean</ejb-name>
<interceptor-class>... .RecordingAuditor</interceptor-class>
<method-name>getChannel</method-name>
<method-params>
<method-param>int</method-param>
</method-params>
</interceptor-binding>
</assembly-descriptor>
</ejb-jar>
```

В вышеприведенном примере дескриптора указано, что мы желаем перехватывать метод TunerBean.getChannel(int) и обрабатывать его вызов классом-перехватчиком RecordingAuditor.

*Rem:* в данном примере описывать сигнатуру метода элементом `<method-params>` не обязательно, т.к. метод `getChannel` не перегружен.

*Rem:* если надо перехватить весь бин, то элементы `<method-name>` и `<method-params>` не нужны.

*Rem:* XML-описание перехвата имеет приоритет над `@` (?)

*Rem:* В спецификации *Interceptors 1.2* введен механизм слабой связанности, приближающий `@` перехвата к XML-описанию по гибкости и портируемости (?).

## Перехватчики по-умолчанию (default interceptors)

Пометить сразу все ejb-классы jar-развертки (deployment) приложения можно только в его дескрипторе развертки ejb-jar.xml. Для этого в `<interceptor-binding>` в качестве значения элемента `<ejb-name>` следует указать «\*» (джокер). Это приведет к тому, что все методы всех ejb-объектов развертки будут перехвачены перехватчиками, перечисленными в элементах `<interceptor-class>` данного `<interceptor-binding>`.

## Очередность применения множества перехватчиков

Первыми применяются перехватчики с наиболее широкой областью действия, т.е. сначала default interceptors, потом перехватчики уровня класса, затем уровня метода. На уровне класса или метода, порядок вызова перехватчиков определяется порядком их перечисления в `@` или в XML.

*Rem: упорядочивание применения перехватчиков введено в Interceptors 1.2*

При этом перехватчики могут обмениваться друг с другом данными при помощи параметров контекста, представляющих из себя именованные объекты Map<String, Object>, которые доступны через InvocationContext.getContextData():

Ex: обмен данными между перехватчиками

```
//Перехватчик 1
...
invocationContext.getContextData().put(«status», «gold»);
...
//Перехватчик 2
...
if(invocationContext.getContextData().get(«status»).equals(«gold»){
...
}
```

## Отмена перехвата

Для отмены перехвата, проведенного на уровне класса или по-умолчанию, можно использовать как @ так и XML.

Для отмены перехвата по-умолчанию можно использовать аннотацию уровня класса @ExcludeDefaultInterceptors. Это аннотация отменит только перехват по-умолчанию, но не затронет другие перехватчики.

Ex: отмена перехвата при помощи аннотаций

```
@Stateless
@ExcludeDefaultInterceptors
@Interceptors(OtherInterceptor.class)
@Local(TunerLocal.class)
public class TunerBean implements ...
```

Это же самое может быть сделано через XML при помощи элемента <exclude-default-interceptors/>:

Ex: отмена перехвата при помощи XML

```
<ejb-jar ...>
...
<assembly-descriptor>
<interceptor-binding>
<ejb-name>*/</ejb-name>
<interceptor-class>... .RecordingAuditor</interceptor-class>
</interceptor-binding>
<interceptor-binding>
<ejb-name>TunerBean</ejb-name>
<exclude-default-interceptors/>
```



```
<interceptor-class>... .OtherInterceptor</interceptor-class>
</interceptor-binding>
</assembly-descriptor>
</ejb-jar>
```

Для исключения перехвата методов, объявленного на уровне класса и по-умолчанию, можно использовать на уровне метода `@ExcludeClassInterceptors` и `@ExcludeDefaultInterceptors`. При этом по-прежнему можно использовать `@Interceptors`, что позволяет создавать для конкретных методов свой собственный стек перехвата. Это же доступно и через XML.

## Перехватчики и внедрение (Interceptors & Injection)

Перехватчики связаны с тем же ENC JNDI, что и EJB, которые они перехватывают. В классах перехватчиков доступны все стандартные для EJB аннотации внедрения `@Resource`, `@EJB`, `@PersistenceContext`. Также можно определить внедрение в класс перехватчика через соответствующие элементы XML дескриптора развертывания.

Ех: модифицированный вариант перехватчика учета, использующий внедренный объект

```
public class RecordingAuditor {
    @Resource
    EJBContext beanCtx;
    ...
    @AroundInvoke
    public Object audit(final InvocationContext ctx) throws Exception {
        ...
        final Principal caller = beanCtx.getCallerPrincipal();
        final AuditedInvocation audit = new AuditInvocation(ctx caller);
        invocations.add(audit);
        ...
    } ...}
}
```

Задачей данного варианта перехватчика является регистрация и сохранение в персистентном (постоянном) виде каждого вызова метода определенного бина, т.е. проведение учетного контроля. В дальнейшем эту информацию можно использовать, например, для анализа уязвимостей. Перехватчик получает информацию о вызывающем клиенте через внедренный в поле `beanCtx` объект `EJBContext`, далее эта информация записывается в объект `caller` и добавляется в список. Можно оформить `audit` как Entity-объект при помощи внедренного РС, тогда информация о вызовах будет сохраняться в РБД.

*Rem:* любые объекты EJB-контейнера, которые могут быть внедрены в сессионные EJB или MDB, также доступны для внедрения в перехватчик.

Как и в случае с EJB-классами, аннотации внедрения в перехватчик создают

дополнительные элементы в ENC JNDI, относящемуся к EJB, к которому привязан данный перехватчик. Это означает, что EJBContext, внедренный в поле beanCtx, будет также доступен в глобальном JNDI по адресу `java:comp/env/org.ejb3book.AuditInterceptor/beanCtx`

Также можно внедрять объекты в перехватчик при помощи XML, для чего в `ejb-jar.xml` надо определить элементы `<interceptor>`.

## **Перехват событий жизненного цикла**

Можно перехватывать не только вызовы методов EJB, но и события их жизненных циклов. Методы класса-перехватчика, перехватывающие эти события, являются методами обратного вызова. Их можно использовать как для инициализации объектов EJB, так и для инициализации собственно объекта перехватчика.

Определение перехватчика событий похоже на определение перехвата вызова метода при помощи `@AroundInvoke`: внутри класса перехватчика нужно определить метод, помеченный одной из аннотаций события жизненного цикла `@PostConstruct`, `@PreDestroy`, `@PrePassivate`, `@PostActivate` и имеющий следующую сигнатуру:  
`@<callback-event-name> void <method-name>(InvocationContext ctx) {...}`

Этот метод не может выбрасывать проверяемые (?) можно системные = непроверяемые + `Remote(?)` исключения (т.к. некому будет их обработать) и возвращает `void`, поскольку все `callback`-методы EJB также ничего не возвращают. Запрещено использовать оператор `throws (?)`.

Подобно `@AroundInvoke`, перехватчик событий жизненного цикла выполняется в едином стеке обработки перехватываемого события. Это означает, что для продолжения обработки события, в его перехватчике должен быть определен вызов `InvocationContext.proceed()`, который передает управление следующему перехватчику или же обработчику события самого EJB, а если таковых нет, то пустому оператору.

*Rem.: callback-методы событий жизненного цикла EJB можно считать внутренними перехватчиками событий, внешние перехватчики событий также не поддерживают транзакции и контекст безопасности EJB, имеют другую сигнатуру.*

*Rem.: смысл выделения внешних перехватчиков событий в том, что с событием жизненного цикла EJB может быть не связано ни одного callback-метода EJB и, следовательно, прицепить перехват метода будет не к чему. Поэтому также `InvocationContext.getMethod()` всегда будет возвращать `null`. Кроме этого, внешний перехватчик событий может обрабатывать события множества различных классов.*

## **Самопальная аннотация внедрения (Custom Injection Annotation)**

Создание собственных `@` внедрения — один из примеров применения перехватчиков событий жизненного цикла EJB.

Вообще EJB-спецификация определяет набор предопределенных аннотаций для внедрения JEE-управляемых объектов (ресурсов, служб, ссылок на EJB и т.д.) в EJB. Однако в дополнение к ним, некоторые сервера приложений, равно как и сами приложения, любят использовать JNDI в качестве глобального реестра настроек и ссылок на посторонние (не JEE) службы.

К сожалению EJB-спецификация не определяет пути для прямого внедрения чего-либо из глобального JNDI прямо в EJB. Вот тут и можно создать собственную аннотацию для внедрения из JNDI, реализацией которого будет заниматься перехватчик. Для этого потребуются: механизм аннотаций, механизм рефлексии, служба JNDI и фреймворк перехвата событий жизненного цикла EJB.

Определим аннотацию внедрения `@JNDIInjected`:

```
import java.lang.annotation.*
@Target({ElementType.METHOD, ElementType.FIELD})
@Retention(RetentionPolicy.RUNTIME)
public Interface JNDIInjected {
    String value();
}
```

Данной аннотацией планируется снабжать свойства EJB (поля и set-методы) для внедрения в них соответствующих объектов. Единственный атрибут по-умолчанию (value) подразумевает имя глобального JNDI.

Пример аннотирования :

```
@Stateless
public class ExBean implements ExRemote {
    @JNDIInjected("java:/TransactionManager")
    private javax.transaction.TransactionManager tm;
    ...
}
```

После создания аннотации и определения порядка ее применения, осталось реализовать само внедрение объектов при помощи перехватчика .

```
import java.lang.reflect.*;
import javax.ejb.*;
import javax.annotation.*;
import javax.interceptor.*;
import javax.naming.*;
import JNDIInjected;

public class JNDIInjector {
```

```

@PostConstruct
public void JNDIInject(InvocationContext ctx){
    Object target = ctx.getTarget();
    // Список всех видов полей, объявленных исключительно в классе
    Field[] fields = target.getClass().getDeclaredFields();
    // Список всех видов методов, объявленных исключительно в классе
    Method[] methods = target.getClass().getDeclaredMethods();
    try {
        InitialContext jndi = new InitialContext();
    // Перебор методов
        for(Method m : methods) {
    // Попытка извлечь @JNDIInjected уровня метода (подразумеваем set-методы)
            JNDIInjected inject = m.getAnnotation(JNDIInjected.class);
    // Если нашли аннотированный @JNDIInjected метод
            if(inject!=null) {
    // Получение через JNDI внедряемого объекта по имени из атр. аннотации
                Object obj = jndi.lookup(inject.value());
    // Принудительная установка доступности «отраженного» set-метода
                m.setAccessible(true);
    // Внедрение полученного объекта при помощи set-метода
                m.invoke(target,obj);
            }
        }
    // Перебор полей
        for(Field f : fields) {
    // Попытка извлечь @JNDIInjected уровня поля
            JNDIInjected inject = f.getAnnotation(JNDIInjected.class);
    // Если нашли аннотированное @JNDIInjected поле
            if(inject!=null) {
    // Получение через JNDI внедряемого объекта по имени из атр. аннотации
                Object obj = jndi.lookup(inject.value());
    // Принудительная установка доступности «отраженного» поля
                f.setAccessible(true);
    // Внедрение полученного объекта в поле
                f.set(target,obj);
            }
        }
    // Передача управления следующему перехватчику/ejb-callback/void-оператору
        ctx.proceed();
    }
    catch(Exception ex) {
    // Выброс непроверяемого исключения
        throw new EJBException();
    }
}

```

Перехватчик, связанный метаданными с одним или более классов EJB, сообщает EJB-контейнеру, что он заинтересован в обработке события @PostConstruct жизненного цикла этих EJB в своем методе JNDIInject(). После перехвата события, в этом методе

получаем ссылку на экземпляр перехваченного целевого бина в виде Object, отражаем на этот объект поиск всех полей и методов, аннотированных @JNDIInjected. Извлекаем из аннотаций соответствующие JNDI-имена, по ним получаем ссылки на соответствующие объекты, иницилируем ими свойства бина. После запускаем дальнейшую обработку события ctx.proceed(). Все операции заключены в блок try/catch для того, чтобы заворачивать все проверяемые исключения в EJBException (расширяет RuntimeException, непроверяемое, системное исключение), т.к. выброс проверяемых исключений запрещен.

В этом примере показано в том числе, что можно использовать EJBInterceptors в качестве фреймворка для написания собственных аннотаций и добавления собственного самопального поведения к EJB.

Использование в связке перехватчиков по-умолчанию дает простой способ применения перехватчика ко всему приложению (ejb-jar) и тем самым глобальную реализацию поведения самопальных аннотаций. Наконец это поведение свободно портируемо и вендорно-независимо. Тем самым EJB 3.x не только стало простым, но и легко-расширяемым.

## Обработка исключений (Exception Handling)

Поскольку перехватчики сидят в стеке перехватываемого вызова метода или события жизненного цикла, то можно регулировать продолжение обработки/вызова путем обертки в блок try/catch/finally вызова InvocationContext.proceed(). Можно остановить вызов, выбросив в перехватчике исключение. Можно перехватить исключение, выброшенное в самом перехватываемом бине и перевыбросить другое или подавить его. С помощью @AroundInvoke можно даже перехватить исключение метода бина и вызвать этот метод вновь.

## Отмена вызова метода

При помощи перехвата вызова метода можно вернуть клиенту исключение вместо ответа, если запрос чем-то не устроил.

*Rem: @AroundInvoke метод перехвата выполняется в одном стэке java-вызова, точнее оборачивает своим вызовом последующие перехваты и сам метод как матрешка. При этом InvocationContext.proceed() выдает в финале ответ или перехваченного метода или void при перехвате событий.*

Ех.: проверка доступа к платным TV-каналам и заворот с возбуждением исключения неаутентифицированных запросов.

...

@AroundInvoke

public Object checkAccess(InvocationContext ctx) throws Exception {

// Если был запрошен 2-й канал

if(isRequestForChannel2(ctx)) {

```
// Если 2-й канал в настоящее время закрыт
if(!channel2AccessPolicy.isChannel2Permitted()){
// Остановка запроса и выброс вместо ответа исключения
throw chanel2CloseException.INSTANCE;
}
}
// Продолжение обработки запроса, если все в порядке
return ctx.proceed();
}
...
```

Этот пример показывает альтернативу EJB Security в виде собственного разрешительного механизма (rule engine), сопряженного с EJB при помощи перехватчика.

## Перехват и перевыброс исключений

При помощи перехвата можно организовать самопальный фреймворк для обработки исключений в EJB.

Для примера рассмотрим JDBC и исключения SQLException. При возникновении исключений наш код программно не может понять причины, вызвавшей это исключение без просмотра кода ошибки и текста сообщения. Но код ошибки привязан к вендору и эти коды различаются в разных СУБД. Обработка (полная) таких исключений сделает EJB вендорно-зависимым. Для ослабления зависимости можно создать собственные SQL-исключения и вендорно-зависимый анализатор на базе перехватчика, который и будет перепackовывать вендорные исключения в эти самопальные.

Определим 2 класса часто-встречающихся SQL-исключений (прикладных исключений с вызовом отката транзакции).

```
@ApplicationException(rollback=true)
public class DBDeadlockException extends SQLException {
    public DBDeadlockException(Exception ex) {
        super(ex);
    }
}

@ApplicationException(rollback=true)
public class DBCursorNotAvailException extends SQLException {
    public DBCursorNotAvailException(Exception ex) {
        super(ex);
    }
}
```

С такими исключениями мы абстрагируемся от конкретных кодов ошибок и наши EJB становятся портируемыми и вендорно-независимыми.

Но для использования этих самопальных исключений нужно создать вендорно-зависимый анализатор-перехватчик, который и будет заниматься перепакровкой и перевыбросом SQL-исключений.

Ex: Пример анализатора-перехватчика SQL-исключений для СУБД MySQL.

```
public class MySQLExceptionHandler {
    @AroundInvoke
    public Object handleException(InvocationContext ctx) throws Exception {
        try {
            return ctx.proceed();
        }
        catch(SQLException sqlEx) {
            int errNum = sqlEx.getErrorCode();
            switch(errNum) {
                case 32343: {
                    throw new DBDeadlockException(sql);
                }
                case 22211: {
                    throw new DBCursorNotAvailException(sql);
                }
            }
        }
    }
}
```

Теперь эти вендорно-независимые исключения можно применять в клиенте / EJB.

```
...
try {
    exEjb.exDBOperation();
}
catch(DBDeadlockException ex) {
    ...
}
...
```

Код стал портируемым относительно СУБД.

### **Жизненный цикл перехватчика**

Перехватчики обладают теми же событиями жизненного цикла, что и перехватываемые ими EJB. Вообще стоит рассматривать перехватчик как продолжение перехватываемого экземпляра EJB. Они вместе создаются, уничтожаются, пассивируются и активируются. На перехватчик накладываются те же ограничения, что и на перехватываемый EJB, например, в перехватчик нельзя внедрять TSEM, если перехватываемый бин не

является SFSB.

Поскольку перехватчик обладает жизненным циклом и может обрабатывать события жизненного цикла, то он может также сохранять внутреннее состояние и следить за ним. Например, можно организовать обобщенное получение через перехватчик открытых соединений с удаленными системами и закрытие их во время ликвидации экземпляра EJB. Можно организовать деактивацию объектов самопальных аннотаций внедрения после ликвидации экземпляра бина. Можно хранить часть внутреннего состояния в перехватчике и очищать ее при ликвидации EJB.

### **@AroundInvoke методы в классе EJB**

@AroundInvoke метод может быть определен непосредственно в классе бина. В этом случае он будет играть роль «последнего» перехватчика. Такие перехватчики могут использоваться для динамической реализации бина. Также можно использовать такой метод как бин-специфичный перехватчик, адаптирующий общее поведение перехвата под конкретный класс.

Кроме того внутренний перехватчик перехватывает все методы своего класса (?)кроме конструктора(?).

### **Перехватчики CDI**

Помимо основной спецификации Interceptors, перехватчики описаны и в спецификации CDI — службы внедрения контекстов и зависимостей, включенной в JEE6 и выше. Это дает возможность перехватывать не только EJB, но и компоненты, управляемые CDI.

### **Связывание с перехватчиком**

При аннотировании @Interceptors методов EJB необходимо явно указывать классы перехватчиков, что задает между ними жесткую связь. Избавиться от этого можно использованием XML-дескриптора развертывания, но это тоже может иметь свои минусы, например, невозможность обработки метаданных в run-time.

В CDI задача ослабления связи решается при помощи связывания с перехватчиками перехватываемого (целевого) объекта. Для этого вводится посредник — самопальная (прикладная) аннотация и связь устанавливается через нее. В итоге у перехватчика и целевых методов имеется связь только с этой аннотацией и их исходные коды при изменениях связи перекомпилировать уже не нужно.

Прикладная аннотация создается при помощи @javax.interceptor.InterceptorBinding:

Ех: прикладная аннотация

@InterceptorBinding



```
@Target({ElementType.METHOD, ElementType.TYPE})
@Retention(RUNTIME)
public @interface Loggable{}
```

В примере выше создана прикладная аннотация `@Loggable`, аннотированная `@InterceptorBinding`. Аннотация `@InterceptorBinding` делает `@Loggable` аннотацией связывания, что дает возможность использовать `@Loggable` для указания методов и классов CDI-объектов, которые надо перехватывать.

Перехватчик полностью аналогичен описанному для EJB, для его связывания с прикладной аннотацией используется аннотация `@Interceptor`:

Ex: связывание прикладной аннотации с перехватчиком

```
@Interceptor
@Loggable
public class LoggingInterceptor {
    ...
    @AroundInvoke
    public Object exMethod(InvocationContext ctx) {...}
    ...
}
```

Перехватываемый метод или класс CDI-объекта просто аннотируется прикладной аннотацией:

Ex: связывание прикладной аннотации с целевым классом CDI-объекта

```
@Transactional
@Loggable
public class ExCDIClass { ... }
```

CDI-перехватчики по-умолчанию отключены, их надо включать в XML-дескрипторе `beans.xml` jar-файла или jee-модуля:

Ex: включение CDI-перехвата в XML-дескрипторе `beans.xml`

```
<beans>
<interceptors>
<class>
    ... .LoginInterceptor
</class>
</interceptors>
</beans>
```

## Приоритеты

Для упорядочивания вызовов перехватчиков используется аннотация `@javax.annotation.Priority`, введенная в CDI 1.1. Единственный ее атрибут – целочисленное значение приоритета, чем оно меньше, тем раньше сработает аннотация в стеке вызова.

Ex: задание приоритета перехвата

```
@Interceptor
@Loggable
@Priority(200)
public class LogInterceptor {...}
```

В `javax.interceptor.Interceptor` определены целые константы, задающие начала диапазонов системных приоритетов:

PLATFORM_BEFORE = 0	Ранние перехватчики jee-платформы
LIBRARY_BEFORE = 1000	Ранние перехватчики библиотек расширений jee-платформы
APPLICATION = 2000	Прикладные перехватчики
LIBRARY_AFTER = 3000	Поздние перехватчики библиотек расширений jee-платформы
PLATFORM_AFTER = 4000	Поздние перехватчики jee-платформы

В качестве примера можно задать приоритет перехватчику с тем, чтобы он запускался после jee-перехватчиков, но перед другими прикладными перехватчиками:

```
Ex:
@Interceptor
@Loggable
@Priority(Interceptor.Priority.LIBRARY_BEFORE + 10)
public class LogInterceptor {...}
```

## Ругательства

- Аспект = именованный параметр, встроенный параметр
- Таблица различий JEE

	JEE5	JEE6	JEE7
EJB	3.0	3.1	3.2
JPA	1.0	2.0	2.1
JTA	1.1	1.1	1.2
Interceptors	1.0	1.1	1.2

- Отражение = Reflection = проверка/клонирование/модификация программой собственного кода. Отраженный объект — наблюдаемая, модифицируемая копия

объекта. Отражение на (отраженный) объект операции — делегирование объекту выполнение этой операции.

## **Литература**

Enterprise JavaBeans 3.1 (6<sup>th</sup> edition), Andrew Lee Rubinger, Bill Burke, стр. nnn

Изучаем Java EE 7, Энтони Гонсалвес, стр. nnn

EJB3 в действии, (2е издание), Дебу Панда, Реза Рахман, Райан Купрак, Майкл Ремижан, стр. nnn