JAVA EE 7新特性

杨宇

目录

SERVLETS

- Annotation
- 异步支持
- 非阻塞IO
- Web片段
- 资源打包
- 处理Multipart请求

SERVLETS: ANNOTATION

```
@WebServlet("/account")
public class AccountServlet extends javax.servlet.http.HttpServlet {
   //. . .
}

@WebFilter("/*")
public class LoggingFilter implements javax.servlet.Filter {
   public void doFilter(HttpServletRequest request, HttpServletResponse response) {
      //. . .
   }
}

@WebListener
public class MyContextListener implements ServletContextListener {
   @Override
   public void contextInitialized(ServletContextEvent sce) {
      ServletContext context = sce.getServletContext();
      //. . .
   }

   @Override
   public void contextDestroyed(ServletContextEvent sce) {
      //. . .
   }
}
```

SERVLETS:异步支持

```
class MyAsyncService implements Runnable {
        AsyncContext ac;
        public MyAsyncService(AsyncContext ac) {
                this.ac = ac;
        }
        @Override
        public void run() {
                //. . .
                ac.complete();
        }
}

@WebServlet(urlPatterns="/async", asyncSupported=true)
public class MyAsyncServlet extends HttpServlet {
        @Override
        protected void doGet(HttpServletRequest request, HttpServletResponse response) {
                AsyncContext ac = request.startAsync();
                ac.addListener(new AsyncListener() {
                        public void onComplete(AsyncEvent event) throws IOException {
                                //. . .
                        }
                        public void onTimeout(AsyncEvent event) throws IOException {
                                //. . .
                        }
                        //. . .
                });
                ScheduledThreadPoolExecutor executor = new ScheduledThreadPoolExecutor(10);
                executor.execute(new MyAsyncService(ac));
        }
}
```

SERVLETS:非阻塞IO

```
@WebServlet(urlPatterns="/async", asyncSupported=true)
public class MyAsyncServlet extends HttpServlet {
        protected void doGet(HttpServletRequest request, HttpServletResponse response) {
                AsyncContext context = request.startAsync();
                ServletInputStream input = request.getInputStream();
                input.setReadListener(new MyReadListener(input, context));
                //…
        }

public class MyReadListener implements ReadListener {
        @Override
        public void onDataAvailable() {
                try {
                        StringBuilder sb = new StringBuilder();
                        int len = -1;
                        byte b[] = new byte[1024];
                        while (input.isReady() && (len = input.read(b)) != -1) {
                                String data = new String(b, 0, len);
                        }
                } catch (IOException ex) {
                        //. . .
                }
        }
        //…
}
```

SERVLET:WEB片段

- 在jar的/META-INF/web-fragment.xml中定义web片段。Web容器自动找到它并配置。

```
<web-fragment>
    <filter>
        <filter-name>MyFilter</filter-name>
        <filter-class>org.example.MyFilter</filter-class>
        <init-param>
            <param-name>myInitParam</param-name>
            <param-value>...</param-value>
        </init-param>
    </filter>
    <filter-mapping>
        <filter-name>MyFilter</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
</web-fragment>
```

SERVLET:WEB片段

- Web片段顺序：在web.xml中配置

```
<web-app>
    <name>MyApp</name>
    <absolute-ordering>
        <name>MyServlet</name>
        <name>MyFilter</name>
    </absolute-ordering>
</web-app>
```

- Web片段顺序：在fragment中配置

```
<web-fragment>
    <name>MyFilter</name>
    <ordering>
        <after>MyServlet</after>
    </ordering>
</web-fragment>
```

SERVLET:资源打包

- 将资源文件打包到Jar的/META-INF/resources/目录中

- 可以使用ServletContext.getResource()或getResourceAsStream()从/.加载。

SERVLET:处理MULTIPART请求

```
@WebServlet(urlPatterns = {"/FileUploadServlet"})
@MultipartConfig(location="/tmp")
public class FileUploadServlet extends HttpServlet {
    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        for (Part part : request.getParts()) {
            part.write("myFile");
        }
    }
}
```

JSON处理

- 流式API
- 对象模型API

JSON: 流式API

- String json = "[{\"apple\":\"red\"},{\"banana\":\"yellow\"}]";
- JsonParser parser = Json.createParser(new StringReader(json));
- assertEquals(JsonParser.Event.START_ARRAY, parser.next());
- assertEquals(JsonParser.Event.START_OBJECT, parser.next());
- assertEquals(JsonParser.Event.KEY_NAME, parser.next());
- assertEquals(JsonParser.Event.VALUE_STRING, parser.next());
- assertEquals(JsonParser.Event.END_OBJECT, parser.next());
- assertEquals(JsonParser.Event.START_OBJECT, parser.next());
- assertEquals(JsonParser.Event.KEY_NAME, parser.next());
- assertEquals(JsonParser.Event.VALUE_STRING, parser.next());
- assertEquals(JsonParser.Event.END_OBJECT, parser.next());
- assertEquals(JsonParser.Event.END_ARRAY, parser.next());

JSON: 流式API

```
@Test
public void testSimpleObject() throws JSONException {
    JsonGeneratorFactory factory = Json.createGeneratorFactory(null);
    StringWriter w = new StringWriter();
    JsonGenerator gen = factory.createGenerator(w);
    gen
        .writeStartObject()
        .write("apple", "red")
        .write("banana", "yellow")
        .writeEnd();
    gen.flush();
    JSONAssert.assertEquals("{\"apple\" : \"red\", \"banana\" : \"yellow\" }", w.toString(), JSONCompareMode.STRICT);
}
```

JSON：DOM式

```
jsonReader = Json.createReader(new StringReader("{"
      + " \”apple\”:\”red\","
      + " \"banana\":\"yellow\""
      + "}"));
JsonObject json = jsonReader.readObject();
json.getString("apple");
json.getString(“banana");

jsonReader = Json.createReader(new StringReader("["
      + " { \"apple\":\"red\" },"
      + " { \"banana\":\"yellow\" }"
      + "]"));
JsonArray jsonArray = jsonReader.readArray();
```

JSON：DOM式

```
@Test
public void testSimpleObject() throws JSONException {
    JsonObject jsonObject = Json.createObjectBuilder()
            .add("apple", "red")
            .add("banana", "yellow")
            .build();
        StringWriter w = new StringWriter();
        try (JsonWriter writer = Json.createWriter(w)) {
            writer.write(jsonObject);
        }
        JSONAssert.assertEquals("{\"apple\" : \"red\", \"banana\" : \"yellow\" }",
w.toString(), JSONCompareMode.STRICT);
    }
```

WEB SOCKET

- 定义服务端点
- 定义客户端点
- Javascript客户端
- 编码器和解码器

WEBSOCKET: 定义服务端点

```
@ServerEndpoint("/chat")
public class ChatServer {
	@OnMessage
	public String receiveMessage(String message) {
		//. . .
	}

	@OnMessage
	public void receiveBigText(String message, boolean last) {
		//. . .
	}

	public void receiveMessage(ByteBuffer b) {
		//. . .
	}
}
```

WEBSOCKET: 定义服务端点

```
@ServerEndpoint("/chat/{room}")
public class MyEndpoint {

    @Inject User user;

    @OnMessage
    public void receiveMessage(String message, @PathParam("room")String room,
        Session session) {
        for (Session peer : session.getOpenSessions()) {
            peer.getBasicRemote().sendObject(message);
        }
        //. . .
    }
}
```

WEBSOCKET: 定义客户端点

```
@ClientEndpoint
public class MyClientEndpoint {
        @OnOpen
        public void open(Session s) {
                //. . .
        }
        @OnClose
        public void close(CloseReason c) {
                //. . .
        }
        @OnError
        public void error(Throwable t) {
                //. . .
        }
        @OnMessage
        public void processMessage(String message, Session session) {
                //. . .
        }
}
```

```
WebSocketContainer container = ContainerProvider.getWebSocketContainer();
String uri = "ws://localhost:8080/myApp/websocket";
Session session = container.connectToServer(MyClientEndpoint.class, URI.create(uri));
```

WEBSOCKET: JAVASCRIPT客户端点

```
var websocket = new WebSocket(“ws://localhost:8080/myapp/chat");
websocket.onopen = function() {
      //. . .
}
websocket.onerror = function(evt) {
      //. . .
}
websocket.onclose = function() {
      //. . .
}
websocket.onmessage = function(evt) {
      console.log("message received: " + evt.data);
}

websocket.send(myField.value);
```

WEBSOCKET: 配置

```
public class MyConfigurator extends ServerEndpointConfig.Configurator {
    @Override
    public void modifyHandshake(ServerEndpointConfig sec,
        HandshakeRequest request,
        HandshakeResponse response) {
        //. . .
    }
}

@ServerEndpoint(value="/websocket", configurator = MyConfigurator.class)
public class MyEndpoint {
    @OnMessage
    public void receiveMessage(String name) {
        //. . .
    }
}
```

WEBSOCKET:ENCODER & DECODER

```
public class MyMessageDecoder implements Decoder.Text<MyMessage> {
    @Override
    public MyMessage decode(String string) throws DecodeException {
        MyMessage myMessage = new MyMessage(
            Json.createReader(
                new StringReader(string)).readObject()
            );
        return myMessage;
    }

    @Override
    public boolean willDecode(String string) {
        return true;
    }
    //. . .
}
```

WEBSOCKET:ENCODER & DECODER

```
public class MyMessageEncoder implements Encoder.Text<MyMessage> {

    @Override
    public String encode(MyMessage myMessage) throws EncodeException {
        return myMessage.getJsonObject().toString();
    }
    //. . .
}
```

WEBSOCKET:ENCODER & DECODER

```
@ServerEndpoint(value = "/encoder",
    encoders = {MyMessageEncoder.class},
    decoders = {MyMessageDecoder.class})
public class MyEndpoint {
    @OnMessage
    public MyMessage messageReceived(MyMessage message) {
        System.out.println("messageReceived: " + message);

        return message;
    }
     //. . .
}
```

CDI

- 定义Beans
- 使用Beans
- Qualifier & Alternative
- Producer & Disposer
- Interceptors
- Decorators
- Scopes & Contexts
- Stereotypes
- 事件
- 可移植的扩展点
- 内建的Bean
- 生命周期回调

CDI: 定义BEAN

- Bean存在于Bean Archive中。
- 包含含有Bean相关Annotation的类或会话Bean的Archive是隐含的Bean Archive。
- 包含beans.xml文件的Archive是显式的Bean Archive。
  - 对于Web应用，beans.xml位于WEB-INF或WEB-INF/classes/META-INF目录。
  - 对于EJB模块或jar：beans.xml位于META-INF目录。

CDI: 定义BEAN

- Beans.xml

```
<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
       http://xmlns.jcp.org/xml/ns/javaee/beans_1_1.xsd"
       bean-discovery-mode="all">
</beans>
```

- Bean discovery mode有三种设定：
  - all——存档中的所有类型都是bean
  - annotated——存档中只有标注了相关annotation的类才是Bean
  - none——关闭CDI

CDI: 定义BEAN

- 排除某些类不能作为Bean
  - Beans.xml

```
<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
       http://xmlns.jcp.org/xml/ns/javaee/beans_1_1.xsd"
       bean-discovery-mode=“all">
  <scan>
    <exclude name="org.javaee7.cdi.exclude.filter.beans.*">
    </exclude>
  </scan>

</beans>
```

  - Vetoed

```
@Vetoed
public class SimpleGreeting implements Greeting {
...
}

@Vetoed
package org.sample.beans;
```

CDI: 使用BEAN

- 方式1：依赖注入@Inject

```
public interface Greeting {
    public String greet(String name);
}
```

```
public class SimpleGreeting implements Greeting {
    public String greet(String name) {
        return "Hello" + name;
    }
}

@Stateless
public class GreetingService {
    @Inject Greeting greeting;
    public String greet(String name) {
        return greeting.greet(name);
    }
}
```

CDI: 使用BEAN

- 字段注入

```
@Stateless
public class GreetingService {
        @Inject Greeting greeting;
        public String greet(String name) {
                return greeting.greet(name);
        }
}
```

- 方法注入

```
@Inject
public setGreeting(Greeting greeting) {
        this.greeting = greeting;
}
```

- 构造函数注入

```
@Inject
public SimpleGreeting(Greeting greeting) {
        this.greeting = greeting;
}
```

CDI: 使用BEAN

- 方式2： BeanManager

- BeanManager bm = CDI.current().getBeanManager();
  Set<Bean<?>> beans = bm.getBeans(Greeting.class);

- 或：

- @Inject BeanManager bm;
  public void doXXX() {
  Set<Bean<?>> beans = bm.getBeans(Greeting.class);
  }

- 或：

- BeanManager bm = (BeanManager)context.lookup("java:comp/BeanManager");

CDI: QUALIFIER

- Qualifier从Bean类型的多个实现中选择一个

```
@Qualifier
@Retention(RUNTIME)
@Target({METHOD, FIELD, PARAMETER, TYPE})
public @interface Fancy {
}
```

```
@Fancy
public class FancyGreeting implements Greeting {
        public String greet(String name) {
                return "Nice to meet you, hello" + name;
        }
}

@Stateless
public class GreetingService {
        @Inject @Fancy Greeting greeting;
        public String sayHello(String name) {
                return greeting.greet(name);
        }
}
```

CDI: ALTERNATIVE

- Alternative：从多个实现中选择一个

```
@Alternative
public class SimpleGreeting implements Greeting {
	//. . .
}
@Fancy @Alternative
public class FancyGreeting implements Greeting {
	//. . .
}
```

```
<beans
xmlns="http://xmlns.jcp.org/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="
http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/beans_1_1.xsd"
bean-discovery-mode="annotated">
	<alternatives>
		<class>org.sample.FancyGreeting</class>
	</alternatives>
</beans>
```

CDI: PRODUCER

- Producer: Bean工厂

定义:

```
@Produces
public List<String> getGreetings() {
List<String> response = new ArrayList<String>();
//. . .
return response;
}
```

使用:

```
@Inject List<String> list;
```

- 通过这种方式可以:
  - 运行时确定实现子类
  - 被注入的不是一个已定义的Bean
  - Bean实例需要定制的初始化过程

CDI: DISPOSER

- 某些通过@Produce注入的Bean需要析构：

```
void close(@Disposes Connection connection) {
    connection.close();
}
```

CDI: INTERCEPTOR

- 定义拦截器绑定类型
- 定义拦截器
- 在Bean上应用拦截器

CDI: INTERCEPTOR

- 定义拦截器绑定类型

```
@InterceptorBinding
@Retention(RUNTIME)
@Target({METHOD,TYPE})
public @interface Logging {
}
```

CDI: INTERCEPTOR

- 定义拦截器

```
@Interceptor
@Logging
public class LoggingInterceptor {
    @Resource UserTransaction tx;
    @AroundInvoke
    public Object log(InvocationContext context) throws Exception {
        String name = context.getMethod().getName();
        String params = context.getParameters().toString();
        //. . .
        return context.proceed();
    }
}
```

CDI: INTERCEPTOR

- 在Bean上应用拦截器

```
@Logging
public class SimpleGreeting {
    //. . .
}
```

OR:

```
public class SimpleGreeting {
    @Logging
    public String greet(String name) {
        //. . .
    }
}
```

CDI: INTERCEPTOR

- 拦截器处理顺序：Annotation方式

```
@Priority(Interceptor.Priority.APPLICATION+10)
@Interceptor
@Logging
public class LoggingInterceptor {
      //. . .
}
```

拦截器优先级：

Priority value	Constant field value
Interceptor.Priority.PLATFORM_BEFORE	0
Interceptor.Priority.LIBRARY_BEFORE	1000
Interceptor.Priority.APPLICATION	2000
Interceptor.Priority.LIBRARY_AFTER	3000
Interceptor.Priority.PLATFORM_AFTER	4000

CDI: INTERCEPTOR

- 拦截器处理顺序：beans.xml方式

```
<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="
http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/beans_1_1.xsd"
bean-discovery-mode="annotated">
    <interceptors>
        <class>org.sample.LoggingInterceptor</class>
        <class>org.sample.SecurityInterceptor</class>
    </interceptors>
</beans>
```

CDI: DECORATOR

- 定义装饰器

```
@Decorator
@Priority(Interceptor.Priority.APPLICATION+10)
class MyDecorator implements Greeting {
	@Inject @Delegate @Any Greeting greeting;
	public String greet(String name) {
		return greeting.greet(name + " <b>very much!</b>");
	}
}
```

CDI: DECORATOR

- 使能装饰器，以及定义装饰顺序
  - 默认所有装饰器都是disable的
  - 加入@Priority后enable装饰器
  - 或者在beans.xml中定义：

```
<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="
http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/beans_1_1.xsd"
bean-discovery-mode="annotated">
        <decorators>
                <class>org.sample.MyDecorator</class>
        </decorators>
</beans>
```

CDI: 范围和上下文

- 每个Bean都处于某个范围，并关联到一个上下文
- 上下文管理Bean在范围中的生命周期和可见性
- 在范围中bean被创建一次，随后被多次重用。

CDI: 范围和上下文

- 内建的范围:

@RequestScoped
@SessionScoped
@ApplicationScoped
@ConversationScoped：默认状态下Bean是瞬态的。通过Conversation.begin()变成长期的；通过Conversation.end()结束。
@Dependent

除了@Dependent范围之外，上下文引用的都不是Bean实例，而是客户端代理对象。

CDI: 事件

并发工具

- 异步任务
- 排程任务
- 受托管的线程
- 动态上下文对象

并发工具：异步任务

- 中心对象： ManagedExecutorService

```
InitialContext ctx = new InitialContext();
ManagedExecutorService executor =
(ManagedExecutorService)ctx.lookup(“java:comp/
DefaultManagedExecutorService");
```

OR

```
@Resource(lookup="java:comp/DefaultManagedExecutorService")
ManagedExecutorService executor;
```

并发工具：异步任务

- 任务定义：实现Runnable或Callable接口

```
Public class MyTask1 implements Runnable {
        @Override
        public void run() {
                //. . .
        }
}

OR

public class MyTask2 implements Callable<Product> {
        private int id;
        public MyTask2(int id) {
                this.id = id;
        }

        @Override
        public Product call() {
                Product product = new Product(id);
                //. . .
        return product;
        }
}
```

并发工具：异步任务

- 任务执行

```
Future future = executor.submit(new MyTask1());
```

OR

```
Collection<Callable<Product>> tasks = new ArrayList<>();
tasks.add(new MyTask2(1));
tasks.add(new MyTask2(2));
//. . .
List<Future<Product>> results = executor.invokeAll(tasks);
Product result = results.get(0).get();
```

并发工具：排程任务

- 任务执行服务：ManagedScheduledExecutorService

```
InitialContext ctx = new InitialContext();
ManagedScheduledExecutorService executor =
(ManagedScheduledExecutorService)ctx
.lookup(“java:comp/DefaultManagedScheduledExecutorService");
```

OR

```
@Resource(lookup="java:comp/DefaultManagedScheduledExecutorService")
ManagedScheduledExecutorService executor;
```

并发工具：排程任务

- 执行排程任务：

<V> ScheduledFuture<V> schedule(Callable<V>
ScheduledFuture<?> schedule(Runnable command, Trigger trigger)
<V> ScheduledFuture<V> schedule(Callable<V> callable, long delay, TimeUnit unit)
ScheduledFuture<?> scheduleAtFixedRate(Runnable command, long initial Delay, long period, TimeUnit unit)
ScheduledFuture <?> scheduleWithFixedDelay(Runnable command, long in itialDelay, long delay, TimeUnit unit)

并发工具：受托管的线程

- 中心对象： ManagedThreadFactory

```
InitialContext ctx = new InitialContext();
ManagedThreadFactory factory =
(ManagedThreadFactory)ctx.lookup(“java:comp/DefaultManagedThreadFactory");
```

OR

```
@Resource(lookup="java:comp/DefaultManagedThreadFactory")
ManagedThreadFactory factory;
```

- 执行线程

```
Thread thread = factory.newThread(new MyTask());
thread.start();
```

并发工具：动态上下文对象

- 中心对象： ContextService

```
InitialContext ctx = new InitialContext();
ContextService cs =
(ContextService)ctx.lookup(“java:comp/DefaultContextService");

OR

@Resource(lookup="java:comp/DefaultContextService")
ContextService cs;
```

并发工具：动态上下文对象

- 范例

```
public class MyRunnable implements Runnable {
    @Override
    public void run() {
        //. . .
    }
}

@Resource(lookup="DefaultContextService")
ContextService cs;

@Resource(lookup="DefaultManagedThreadFactory")
ThreadFactory factory;

MyRunnable r = new MyRunnable();

Runnable proxy = cs.createContextualProxy(r, Runnable.class);
ExecutorService es = Executors.newFixedThreadPool(10, factory);
Future f = es.submit(proxy);
```

JMS

- 发送消息
- 同步接收消息
- 异步接收消息
- 服务质量
- 临时JMS目标

JMS: 发送消息

- 同步发送

```
@Stateless
@JMSDestinationDefinitions({
      @JMSDestinationDefinition(name = "java:global/jms/myQueue",
interfaceName = "javax.jms.Queue")})
public class MessageSender {
      @Inject
      @JMSConnectionFactory(“java:comp/DefaultJMSConnectionFactory")
      JMSContext context;

      @Resource(mappedName="java:global/jms/myQueue")
      Destination myQueue;

      public void sendMessage(String message) {
            context.createProducer().send(myQueue, message);
      }
}
```

JMS: 发送消息

- 异步发送

```
context.createProducer().setAsync(new CompletionListener() {
        @Override
        public void onCompletion(Message m) {
                //. . .
        }

        @Override
        public void onException(Message msg, Exception e) {
                //. . .
        }
});
```

JMS: 同步接收消息

```
@Inject
private JMSContext context;

@Resource(mappedName="java:global/jms/myQueue")
Destination myQueue;

public void receiveMessage() {
      String result = context.createConsumer(myQueue).receiveBody(String.class, 1000);
      //. . .
}
```

JMS：异步接收消息

```
@MessageDriven(mappedName = "myDestination")
public class MyMessageBean implements MessageListener {

    @Override
    public void onMessage(Message message) {
        try {
            // process the message
        } catch (JMSException ex) {
            //. . .
        }
    }
}
```

JMS: 临时目的地

```
TemporaryQueue tempQueue = context.createTemporaryQueue();
TemporaryTopic tempTopic = context.createTemporaryTopic();
```

JMS: 服务质量

- 默认情况下，JMS消息是非持久化的。

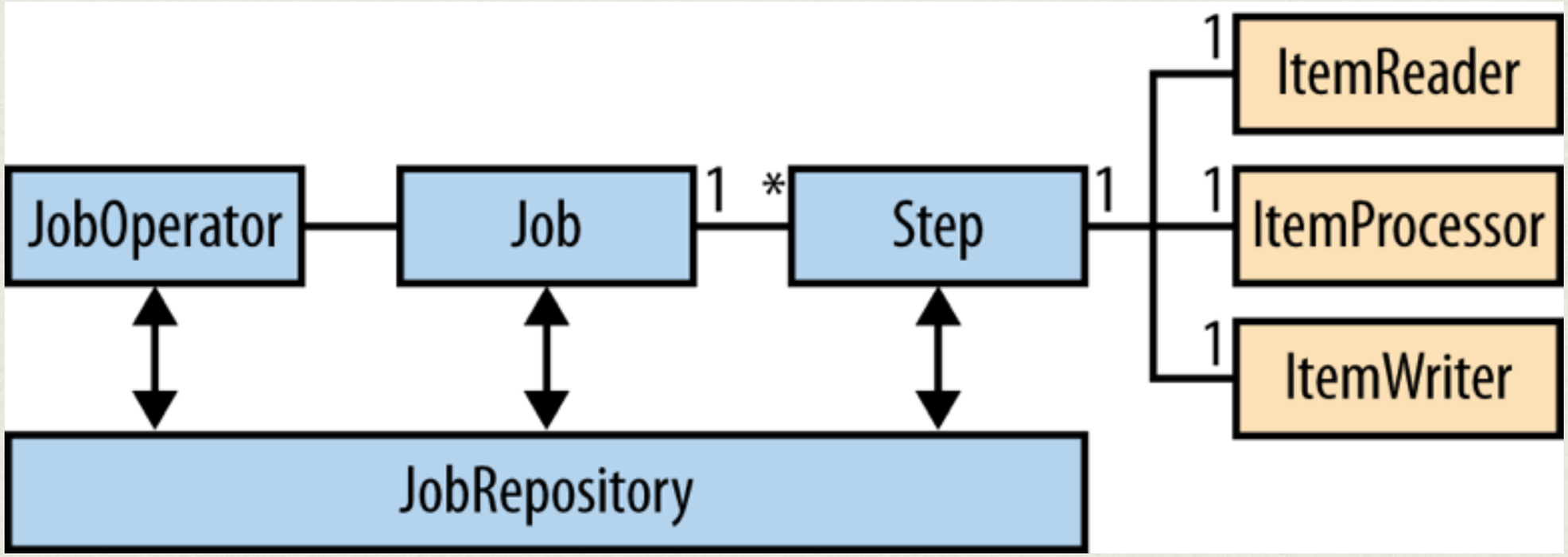
- 设置为持久化：

```
context.createProducer().setDeliveryMode(DeliveryMode.NON_PERSISTENT);
```

批处理

- 面向Chunk的批处理
- 面向Batchlet的批处理
- 监听器
- Job步骤顺序

批处理：关键概念


JobOperator
Job
1
*
Step
1
1
ItemReader
1
ItemProcessor
1
ItemWriter
JobRepository

批处理：关键概念

- Job: 封装一整个批处理的任务
- Step: 封装Job中的一个阶段
- JobOperator: 管理Job处理的命令接口
- JobRepository: 保存正在运行和已经运行完毕的Job
- Reader-Processor-Writer or Batchlet：定义处理行为

批处理： 面向CHUNK

- 实现ItemReader, ItemProcessor和ItemWriter
- 定义job xml（位于META-INF/batch-jobs/）。

```
<job id="myJob" xmlns="http://xmlns.jcp.org/xml/ns/javaee" version="1.0">
      <step id="myStep">
            <chunk item-count="3">
                  <reader ref="myItemReader"/>
                  <processor ref="myItemProcessor"/>
                  <writer ref=“myItemWriter"/>
            </chunk>
      </step>
</job>
```

- 执行Job

```
JobOperator jo = BatchRuntime.getJobOperator();
long jid = jo.start("myJob", new Properties());
```

批处理： BATCHLET

- 实现Batchlet类

```
@Named
public class MyBatchlet extends AbstractBatchlet {
        @Override
        public String process() {
                //. . .
                return "COMPLETED";
        }
}
```

- 定义job xml

```
<job id="myJob" xmlns="http://xmlns.jcp.org/xml/ns/javaee" version="1.0">
        <step id="myStep" >
                <batchlet ref="myBatchlet"/>
        </step>
</job>
```

- 执行批处理

```
JobOperator jo = BatchRuntime.getJobOperator();
long jid = jo.start("myJob", new Properties());
```

批处理： 监听器

- 各种监听器： JobListener, StepListener, ChunkListener, ItemReadListener, ItemProcessListener, ItemWriteListener…

- Job xml配置：

```
<job id="myJob" xmlns="http://xmlns.jcp.org/xml/ns/javaee" version="1.0">
        <listeners>
                <listener ref="myJobListener"/>
        </listeners>
        <step id="myStep" >
                <listeners>
                        <listener ref="myStepListener"/>
                        <listener ref="myChunkListener"/>
                        <listener ref="myItemReadListener"/>
                        <listener ref="myItemProcessorListener"/>
                        <listener ref="myItemWriteListener"/>
                </listeners>
                <chunk>
                        . . .
                </chunk>
        </step>
</job>
```

批处理：JOB顺序

- next属性：指定顺序

```
<job id="myJob" xmlns="http://xmlns.jcp.org/xml/ns/javaee" version="1.0">
    <step id="step1" next="step2">
        <chunk item-count="3">
            <reader ref="myItemReader"></reader>
            <processor ref="myItemProcessor"></processor>
            <writer ref="myItemWriter"></writer>
        </chunk>
    </step>
    <step id="step2" >
        <batchlet ref="myBatchlet"/>
    </step>
</job>
```

批处理：JOB顺序

- Flow: 封装执行Unit

```
<job id="myJob" xmlns="http://xmlns.jcp.org/xml/ns/javaee" version="1.0">
    <flow id="flow1" next="step3">
        <step id="step1" next="step2">
            . . .
        </step>
        <step id="step2" >
            . . .
        </step>
    </flow>
    <step id="step3" >
        . . .
    </step>
</job>
```

批处理：JOB顺序

- Split: 并行执行

```
<job id="myJob" xmlns="http://xmlns.jcp.org/xml/ns/javaee" version="1.0">
      <split id="split1" next="step3">
            <flow id="flow1">
                  <step id="step1">
                        . . .
                  </step>
            </flow>
            <flow id="flow2">
                  <step id="step2">
                        . . .
                  </step>
            </flow>
      </split>
      <step id="step3">
            . . .
      </step>
</job>
```

批处理： JOB顺序

- Decision: 决定执行顺序:next, fail, end, stop

```
<job id="myJob" xmlns="http://xmlns.jcp.org/xml/ns/javaee" version="1.0">
        <step id="step1" next="decider1">
                . . .
        </step>
        <decision id="decider1" ref="myDecider">
                <next on="DATA_LOADED" to="step2"/>
                <end on="NOT_LOADED"/>
        </decision>
        <step id="step2">
                . . .
        </step>
</job>

public class MyDecider implements Decider {
        @Override
        public String decide(StepExecution[] ses) throws Exception {
                //. . .
                if (...)
                        return "NOT_LOADED";
                if (...)
                        return "DATA_LOADED";
        }
}
```