

Spring开发

什么是Spring?

Spring是一个支持快速开发Java EE应用程序的框架。它提供了一系列底层容器和基础设施，并可以和大量常用的开源框架无缝集成，可以说是开发Java EE应用程序的必备。



Spring最早是由Rod Johnson这哥们在他的《[Expert One-on-One J2EE Development without EJB](#)》一书中提出的用来取代EJB的轻量级框架。随后这哥们又开始专心开发这个基础框架，并起名为Spring Framework。

随着Spring越来越受欢迎，在Spring Framework基础上，又诞生了Spring Boot、Spring Cloud、Spring Data、Spring Security等一系列基于Spring Framework的项目。本章我们只介绍Spring Framework，即最核心的Spring框架。后续章节我们还会涉及Spring Boot、Spring Cloud等其他框架。

Spring Framework

Spring Framework主要包括几个模块：

- 支持IoC和AOP的容器；
- 支持JDBC和ORM的数据访问模块；
- 支持声明式事务的模块；
- 支持基于Servlet的MVC开发；
- 支持基于Reactive的Web开发；
- 以及集成JMS、JavaMail、JMX、缓存等其他模块。

我们会依次介绍Spring Framework的主要功能。

IoC容器

在学习Spring框架时，我们遇到的第一个也是最核心的概念就是容器。

什么是容器？容器是一种为某种特定组件的运行提供必要支持的一个软件环境。例如，Tomcat就是一个Servlet容器，它可以为Servlet的运行提供运行环境。类似Docker这样的软件也是一个容器，它提供了必要的Linux环境以便运行一个特定的Linux进程。

通常来说，使用容器运行组件，除了提供一个组件运行环境之外，容器还提供了许多底层服务。例如，Servlet容器底层实现了TCP连接，解析HTTP协议等非常复杂的服务，如果没有容器来提供这些服务，我们就无法编写像Servlet这样代码简单，功能强大的组件。早期的JavaEE服务器提供的EJB容器最重要的功能就是通过声明式事务服务，使得EJB组件的开发人员不必自己编写冗长的事务处理代码，所以极大地简化了事务处理。

Spring的核心就是提供了一个IoC容器，它可以管理所有轻量级的JavaBean组件，提供的底层服务包括组件的生命周期管理、配置和组装服务、AOP支持，以及建立在AOP基础上的声明式事务服务等。

本章我们讨论的IoC容器，主要介绍Spring容器如何对组件进行生命周期管理和配置组装服务。

IoC原理

Spring提供的容器又称为IoC容器，什么是IoC？

IoC全称Inversion of Control，直译为控制反转。那么何谓IoC？在理解IoC之前，我们先看看通常的Java组件是如何协作的。

我们假定一个在线书店，通过 `BookService` 获取书籍：

```
public class BookService {
    private HikariConfig config = new HikariConfig();
    private DataSource dataSource = new HikariDataSource(config);

    public Book getBook(long bookId) {
        try (Connection conn = dataSource.getConnection()) {
            ...
            return book;
        }
    }
}
```

为了从数据库查询书籍，`BookService` 持有一个 `DataSource`。为了实例化一个 `HikariDataSource`，又不得不实例化一个 `HikariConfig`。

现在，我们继续编写 `UserService` 获取用户：

```
public class UserService {
    private HikariConfig config = new HikariConfig();
    private DataSource dataSource = new HikariDataSource(config);

    public User getUser(long userId) {
        try (Connection conn = dataSource.getConnection()) {
            ...
            return user;
        }
    }
}
```

因为 `UserService` 也需要访问数据库，因此，我们不得不也实例化一个 `HikariDataSource`。

在处理用户购买的 `CartServlet` 中，我们需要实例化 `UserService` 和 `BookService`：

```

public class CartServlet extends HttpServlet {
    private BookService bookService = new BookService();
    private UserService userService = new UserService();

    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
        long currentUserId = getFromCookie(req);
        User currentUser = userService.getUser(currentUserId);
        Book book = bookService.getBook(req.getParameter("bookId"));
        cartService.addToCart(currentUser, book);
        ...
    }
}

```

类似的，在购买历史 `HistoryServlet` 中，也需要实例化 `UserService` 和 `BookService`：

```

public class HistoryServlet extends HttpServlet {
    private BookService bookService = new BookService();
    private UserService userService = new UserService();
}

```

上述每个组件都采用了一种简单的通过 `new` 创建实例并持有的方式。仔细观察，会发现以下缺点：

1. 实例化一个组件其实很难，例如，`BookService` 和 `UserService` 要创建 `HikariDataSource`，实际上需要读取配置，才能先实例化 `HikariConfig`，再实例化 `HikariDataSource`。
2. 没有必要让 `BookService` 和 `UserService` 分别创建 `DataSource` 实例，完全可以共享同一个 `DataSource`，但谁负责创建 `DataSource`，谁负责获取其他组件已经创建的 `DataSource`，不好处理。类似的，`CartServlet` 和 `HistoryServlet` 也应当共享 `BookService` 实例和 `UserService` 实例，但也不好处理。
3. 很多组件需要销毁以便释放资源，例如 `DataSource`，但如果该组件被多个组件共享，如何确保它的使用方都已经全部被销毁？
4. 随着更多的组件被引入，例如，书籍评论，需要共享的组件写起来会更困难，这些组件的依赖关系会越来越复杂。
5. 测试某个组件，例如 `BookService`，是复杂的，因为必须要在真实的数据库环境下执行。

从上面的例子可以看出，如果一个系统有大量的组件，其生命周期和相互之间的依赖关系如果由组件自身来维护，不但大大增加了系统的复杂度，而且会导致组件之间极为紧密的耦合，继而给测试和维护带来了极大的困难。

因此，核心问题是：

1. 谁负责创建组件？
2. 谁负责根据依赖关系组装组件？
3. 销毁时，如何按依赖顺序正确销毁？

解决这一问题的核心方案就是IoC。

传统的应用程序中，控制权在程序本身，程序的控制流程完全由开发者控制，例如：

`CartServlet` 创建了 `BookService`，在创建 `BookService` 的过程中，又创建了 `DataSource` 组件。这种模式的缺点是，一个组件如果要使用另一个组件，必须先知道如何正确地创建它。

在IoC模式下，控制权发生了反转，即从应用程序转移到了IoC容器，所有组件不再由应用程序自己创建和配置，而是由IoC容器负责，这样，应用程序只需要直接使用已经创建好并且配置好的组件。为了能让组件在IoC容器中被“装配”出来，需要某种“注入”机制，例如，`BookService` 自己并不会创建 `DataSource`，而是等待外部通过 `setDataSource()` 方法来注入一个 `DataSource`：

```
public class BookService {
    private DataSource dataSource;

    public void setDataSource(DataSource dataSource) {
        this.dataSource = dataSource;
    }
}
```

不直接 new 一个 DataSource，而是注入一个 DataSource，这个小小的改动虽然简单，却带来了一系列好处：

1. BookService 不再关心如何创建 DataSource，因此，不必编写读取数据库配置之类的代码；
2. DataSource 实例被注入到 BookService，同样也可以注入到 UserService，因此，共享一个组件非常简单；
3. 测试 BookService 更容易，因为注入的是 DataSource，可以使用内存数据库，而不是真实的 MySQL 配置。

因此，IoC 又称为依赖注入（DI：Dependency Injection），它解决了一个最主要的问题：将组件的创建+配置与组件的使用相分离，并且，由 IoC 容器负责管理组件的生命周期。

因为 IoC 容器要负责实例化所有的组件，因此，有必要告诉容器如何创建组件，以及各组件的依赖关系。一种最简单的配置是通过 XML 文件来实现，例如：

```
<beans>
    <bean id="dataSource" class="HikariDataSource" />
    <bean id="bookService" class="BookService">
        <property name="dataSource" ref="dataSource" />
    </bean>
    <bean id="userService" class="UserService">
        <property name="dataSource" ref="dataSource" />
    </bean>
</beans>
```

上述 XML 配置文件指示 IoC 容器创建 3 个 JavaBean 组件，并把 id 为 dataSource 的组件通过属性 dataSource（即调用 setDataSource() 方法）注入到另外两个组件中。

在 Spring 的 IoC 容器中，我们把所有组件统称为 JavaBean，即配置一个组件就是配置一个 Bean。

依赖注入方式

我们从上面的代码可以看到，依赖注入可以通过 set() 方法实现。但依赖注入也可以通过构造方法实现。

很多 Java 类都具有带参数的构造方法，如果我们将 BookService 改造为通过构造方法注入，那么实现代码如下：

```
public class BookService {
    private DataSource dataSource;

    public BookService(DataSource dataSource) {
        this.dataSource = dataSource;
    }
}
```

Spring 的 IoC 容器同时支持属性注入和构造方法注入，并允许混合使用。

无侵入容器

在设计上，Spring的IoC容器是一个高度可扩展的无侵入容器。所谓无侵入，是指应用程序的组件无需实现Spring的特定接口，或者说，组件根本不知道自己在Spring的容器中运行。这种无侵入的设计有以下好处：

1. 应用程序组件既可以在Spring的IoC容器中运行，也可以自己编写代码自行组装配置；
2. 测试的时候并不依赖Spring容器，可单独进行测试，大大提高了开发效率。

装配Bean

我们前面讨论了为什么要使用Spring的IoC容器，因为让容器来为我们创建并装配Bean能获得很大的好处，那么到底如何使用IoC容器？装配好的Bean又如何使用？

我们来看一个具体的用户注册登录的例子。整个工程的结构如下：

```
spring-ioc-appcontext
├─ pom.xml
├─ src
│   └─ main
│       ├── java
│       │   └─ com
│       │       └─ itranswarp
│       │           └─ learnjava
│       │               ├── Main.java
│       │               └─ service
│       │                   ├── MailService.java
│       │                   ├── User.java
│       │                   └─ UserService.java
│       └─ resources
│           └─ application.xml
```

首先，我们用Maven创建工程并引入 `spring-context` 依赖：

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.itranswarp.learnjava</groupId>
  <artifactId>spring-ioc-appcontext</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <project.reporting.outputEncoding>UTF-
8</project.reporting.outputEncoding>
    <maven.compiler.source>11</maven.compiler.source>
    <maven.compiler.target>11</maven.compiler.target>
    <java.version>11</java.version>

    <spring.version>5.2.3.RELEASE</spring.version>
  </properties>

  <dependencies>
```

```

        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>${spring.version}</version>
        </dependency>
    </dependencies>
</project>

```

我们先编写一个 `MailService`，用于在用户登录和注册成功后发送邮件通知：

```

public class MailService {
    private ZoneId zoneId = ZoneId.systemDefault();

    public void setZoneId(ZoneId zoneId) {
        this.zoneId = zoneId;
    }

    public String getTime() {
        return
ZonedDateTime.now(this.zoneId).format(DateTimeFormatter.ISO_ZONED_DATE_TIME);
    }

    public void sendLoginMail(User user) {
        System.err.println(String.format("Hi, %s! You are logged in at %s",
user.getName(), getTime()));
    }

    public void sendRegistrationMail(User user) {
        System.err.println(String.format("Welcome, %s!", user.getName()));
    }
}

```

再编写一个 `UserService`，实现用户注册和登录：

```

public class UserService {
    private MailService mailService;

    public void setMailService(MailService mailService) {
        this.mailService = mailService;
    }

    private List<User> users = new ArrayList<>(List.of( // users:
        new User(1, "bob@example.com", "password", "Bob"), // bob
        new User(2, "alice@example.com", "password", "Alice"), // alice
        new User(3, "tom@example.com", "password", "Tom"))); // tom

    public User login(String email, String password) {
        for (User user : users) {
            if (user.getEmail().equalsIgnoreCase(email) &&
user.getPassword().equals(password)) {
                mailService.sendLoginMail(user);
                return user;
            }
        }
        throw new RuntimeException("login failed.");
    }
}

```

```

    }

    public User getUser(long id) {
        return this.users.stream().filter(user -> user.getId() ==
id).findFirst().orElseThrow();
    }

    public User register(String email, String password, String name) {
        users.forEach((user) -> {
            if (user.getEmail().equalsIgnoreCase(email)) {
                throw new RuntimeException("email exist.");
            }
        });
        User user = new User(users.stream().mapToLong(u ->
u.getId()).max().getAsLong() + 1, email, password, name);
        users.add(user);
        mailService.sendRegistrationMail(user);
        return user;
    }
}

```

注意到 `UserService` 通过 `setMailService()` 注入了一个 `MailService`。

然后，我们需要编写一个特定的 `application.xml` 配置文件，告诉Spring的IoC容器应该如何创建并组装Bean：

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="userService" class="com.itranswarp.learnjava.service.UserService">
        <property name="mailService" ref="mailService" />
    </bean>

    <bean id="mailService" class="com.itranswarp.learnjava.service.MailService"
/>
</beans>

```

注意观察上述配置文件，其中与XML Schema相关的部分格式是固定的，我们只关注两个 `<bean ...>` 的配置：

- 每个 `<bean ...>` 都有一个 `id` 标识，相当于Bean的唯一ID；
- 在 `userService` Bean中，通过 `<property name="..." ref="..." />` 注入了另一个Bean；
- Bean的顺序不重要，Spring根据依赖关系会自动正确初始化。

把上述XML配置文件用Java代码写出来，就像这样：

```

UserService userService = new UserService();
MailService mailService = new MailService();
userService.setMailService(mailService);

```

只不过Spring容器是通过读取XML文件后使用反射完成的。

如果注入的不是Bean，而是 `boolean`、`int`、`String` 这样的数据类型，则通过 `value` 注入，例如，创建一个 `HikariDataSource`：

```
<bean id="dataSource" class="com.zaxxer.hikari.HikariDataSource">
  <property name="jdbcUrl" value="jdbc:mysql://localhost:3306/test" />
  <property name="username" value="root" />
  <property name="password" value="password" />
  <property name="maximumPoolSize" value="10" />
  <property name="autoCommit" value="true" />
</bean>
```

最后一步，我们需要创建一个Spring的IoC容器实例，然后加载配置文件，让Spring容器为我们创建并装配好配置文件中指定的所有Bean，这只需要一行代码：

```
ApplicationContext context = new
ClassPathXmlApplicationContext("application.xml");
```

接下来，我们就可以从Spring容器中“取出”装配好的Bean然后使用它：

```
// 获取Bean:
UserService userService = context.getBean(UserService.class);
// 正常调用:
User user = userService.login("bob@example.com", "password");
```

完整的 `main()` 方法如下：

```
public class Main {
    public static void main(String[] args) {
        ApplicationContext context = new
ClassPathXmlApplicationContext("application.xml");
        UserService userService = context.getBean(UserService.class);
        User user = userService.login("bob@example.com", "password");
        System.out.println(user.getName());
    }
}
```

ApplicationContext

我们从创建Spring容器的代码：

```
ApplicationContext context = new
ClassPathXmlApplicationContext("application.xml");
```

可以看到，Spring容器就是 `ApplicationContext`，它是一个接口，有很多实现类，这里我们选择 `ClassPathXmlApplicationContext`，表示它会自动从classpath中查找指定的XML配置文件。

获得了 `ApplicationContext` 的实例，就获得了IoC容器的引用。从 `ApplicationContext` 中我们可以根据Bean的ID获取Bean，但更多的时候我们根据Bean的类型获取Bean的引用：

```
UserService userService = context.getBean(UserService.class);
```

Spring还提供另一种IoC容器叫 `BeanFactory`，使用方式和 `ApplicationContext` 类似：


```
BeanFactory factory = new XmlBeanFactory(new
ClassPathResource("application.xml"));
MailService mailService = factory.getBean(MailService.class);
```

BeanFactory 和 ApplicationContext 的区别在于，BeanFactory 的实现是按需创建，即第一次获取 Bean 时才创建这个 Bean，而 ApplicationContext 会一次性创建所有的 Bean。实际上，ApplicationContext 接口是从 BeanFactory 接口继承而来的，并且，ApplicationContext 提供了一些额外的功能，包括国际化支持、事件和通知机制等。通常情况下，我们总是使用 ApplicationContext，很少会考虑使用 BeanFactory。

练习

在上述示例的基础上，继续给 UserService 注入 DataSource，并把注册和登录功能通过数据库实现。

从  **gitee** 下载练习：[使用ApplicationContext](#)（推荐使用[IDE练习插件](#)快速下载）

小结

Spring 的 IoC 容器接口是 ApplicationContext，并提供了多种实现类；

通过 XML 配置文件创建 IoC 容器时，使用 ClassPathXmlApplicationContext；

持有 IoC 容器后，通过 getBean() 方法获取 Bean 的引用。

使用Annotation配置

使用 Spring 的 IoC 容器，实际上就是通过类似 XML 这样的配置文件，把我们自己的 Bean 的依赖关系描述出来，然后让容器来创建并装配 Bean。一旦容器初始化完毕，我们就直接从容器中获取 Bean 使用它们。

使用 XML 配置的优点是所有的 Bean 都能一目了然地列出来，并通过配置注入能直观地看到每个 Bean 的依赖。它的缺点是写起来非常繁琐，每增加一个组件，就必须把新的 Bean 配置到 XML 中。

有没有其他更简单的配置方式呢？

有！我们可以使用 Annotation 配置，可以完全不需要 XML，让 Spring 自动扫描 Bean 并组装它们。

我们把上一节的示例改造一下，先删除 XML 配置文件，然后，给 UserService 和 MailService 添加几个注解。

首先，我们给 MailService 添加一个 @Component 注解：

```
@Component
public class MailService {
    ...
}
```

这个 @Component 注解就相当于定义了一个 Bean，它有一个可选的名称，默认是 mailService，即小写开头的类名。

然后，我们给 UserService 添加一个 @Component 注解和一个 @Autowired 注解：

```

@Component
public class UserService {
    @Autowired
    MailService mailService;

    ...
}

```

使用 `@Autowired` 就相当于把指定类型的Bean注入到指定的字段中。和XML配置相比，`@Autowired` 大幅简化了注入，因为它不但可以写在 `set()` 方法上，还可以直接写在字段上，甚至可以写在构造方法中：

```

@Component
public class UserService {
    MailService mailService;

    public UserService(@Autowired MailService mailService) {
        this.mailService = mailService;
    }

    ...
}

```

我们一般把 `@Autowired` 写在字段上，通常使用 `package` 权限的字段，便于测试。

最后，编写一个 `AppConfig` 类启动容器：

```

@Configuration
@ComponentScan
public class AppConfig {
    public static void main(String[] args) {
        ApplicationContext context = new
        AnnotationConfigApplicationContext(AppConfig.class);
        UserService userService = context.getBean(UserService.class);
        User user = userService.login("bob@example.com", "password");
        System.out.println(user.getName());
    }
}

```

除了 `main()` 方法外，`AppConfig` 标注了 `@Configuration`，表示它是一个配置类，因为我们创建 `ApplicationContext` 时：

```

ApplicationContext context = new
AnnotationConfigApplicationContext(AppConfig.class);

```

使用的实现类是 `AnnotationConfigApplicationContext`，必须传入一个标注了 `@Configuration` 的类名。

此外，`AppConfig` 还标注了 `@ComponentScan`，它告诉容器，自动搜索当前类所在的包以及子包，把所有标注为 `@Component` 的Bean自动创建出来，并根据 `@Autowired` 进行装配。

整个工程结构如下：

```
spring-ioc-annoconfig
├── pom.xml
├── src
│   ├── main
│   │   ├── java
│   │   │   ├── com
│   │   │   │   ├── itranswarp
│   │   │   │   │   ├── learnjava
│   │   │   │   │   │   ├── AppConfig.java
│   │   │   │   │   │   └── service
│   │   │   │   │   │       ├── MailService.java
│   │   │   │   │   │       ├── User.java
│   │   │   │   │   │       └── UserService.java
```

使用Annotation配合自动扫描能大幅简化Spring的配置，我们只需要保证：

- 每个Bean被标注为 `@Component` 并正确使用 `@Autowired` 注入；
- 配置类被标注为 `@Configuration` 和 `@ComponentScan`；
- 所有Bean均在指定包以及子包内。

使用 `@ComponentScan` 非常方便，但是，我们也要特别注意包的层次结构。通常来说，启动配置 `AppConfig` 位于自定义的顶层包（例如 `com.itranswarp.learnjava`），其他Bean按类别放入子包。

思考

如果我们想给 `UserService` 注入 `HikariDataSource`，但是这个类位于 `com.zaxxer.hikari` 包中，并且 `HikariDataSource` 也不可能有 `@Component` 注解，如何告诉IoC容器创建并配置 `HikariDataSource`？或者换个说法，如何创建并配置一个第三方Bean？

练习

从  **gitee** 下载练习：[使用Annotation配置IoC容器](#)（推荐使用[IDE练习插件](#)快速下载）

小结

使用Annotation可以大幅简化配置，每个Bean通过 `@Component` 和 `@Autowired` 注入；

必须合理设计包的层次结构，才能发挥 `@ComponentScan` 的威力。

定制Bean

Scope

对于Spring容器来说，当我们把一个Bean标记为 `@Component` 后，它就会自动为我们创建一个单例（Singleton），即容器初始化时创建Bean，容器关闭前销毁Bean。在容器运行期间，我们调用 `getBean(Class)` 获取到的Bean总是同一个实例。

还有一种Bean，我们每次调用 `getBean(Class)`，容器都返回一个新的实例，这种Bean称为Prototype（原型），它的生命周期显然和Singleton不同。声明一个Prototype的Bean时，需要添加一个额外的 `@Scope` 注解：

```
@Component
@Scope(ConfigurableBeanFactory.SCOPE_PROTOTYPE) // @Scope("prototype")
public class MailSession {
    ...
}
```

注入List

有些时候，我们会有一系列接口相同，不同实现类的Bean。例如，注册用户时，我们要对email、password和name这3个变量进行验证。为了便于扩展，我们先定义验证接口：

```
public interface Validator {  
    void validate(String email, String password, String name);  
}
```

然后，分别使用3个 `Validator` 对用户参数进行验证：

```
@Component  
public class EmailValidator implements Validator {  
    public void validate(String email, String password, String name) {  
        if (!email.matches("^[a-z0-9]+\\@[a-z0-9]+\\. [a-z]{2,10}$")) {  
            throw new IllegalArgumentException("invalid email: " + email);  
        }  
    }  
}  
  
@Component  
public class PasswordValidator implements Validator {  
    public void validate(String email, String password, String name) {  
        if (!password.matches("^[0-9a-zA-Z]{6,20}$")) {  
            throw new IllegalArgumentException("invalid password");  
        }  
    }  
}  
  
@Component  
public class NameValidator implements Validator {  
    public void validate(String email, String password, String name) {  
        if (name == null || name.isBlank() || name.length() > 20) {  
            throw new IllegalArgumentException("invalid name: " + name);  
        }  
    }  
}
```

最后，我们通过一个 `Validators` 作为入口进行验证：

```
@Component  
public class Validators {  
    @Autowired  
    List<Validator> validators;  
  
    public void validate(String email, String password, String name) {  
        for (var validator : this.validators) {  
            validator.validate(email, password, name);  
        }  
    }  
}
```

注意到 `Validators` 被注入了一个 `List<Validator>`，Spring会自动把所有类型为 `Validator` 的Bean装配为一个 `List` 注入进来，这样一来，我们每新增一个 `Validator` 类型，就自动被Spring装配到 `Validators` 中了，非常方便。

因为Spring是通过扫描classpath获取到所有的Bean，而List是有序的，要指定List中Bean的顺序，可以加上@Order注解：

```
@Component
@Order(1)
public class EmailValidator implements Validator {
    ...
}

@Component
@Order(2)
public class PasswordValidator implements Validator {
    ...
}

@Component
@Order(3)
public class NameValidator implements Validator {
    ...
}
```

可选注入

默认情况下，当我们标记了一个@Autowired后，Spring如果没有找到对应类型的Bean，它会抛出NoSuchBeanDefinitionException异常。

可以给@Autowired增加一个required = false的参数：

```
@Component
public class MailService {
    @Autowired(required = false)
    ZoneId zoneId = ZoneId.systemDefault();
    ...
}
```

这个参数告诉Spring容器，如果找到一个类型为ZoneId的Bean，就注入，如果找不到，就忽略。

这种方式非常适合有定义就使用定义，没有就使用默认值的情况。

创建第三方Bean

如果一个Bean不在我们自己的package管理之内，例如ZoneId，如何创建它？

答案是我们自己在@Configuration类中编写一个Java方法创建并返回它，注意给方法标记一个@Bean注解：

```
@Configuration
@ComponentScan
public class AppConfig {
    // 创建一个Bean：
    @Bean
    ZoneId createZoneId() {
        return ZoneId.of("Z");
    }
}
```

Spring对标记为 `@Bean` 的方法只调用一次，因此返回的Bean仍然是单例。

初始化和销毁

有些时候，一个Bean在注入必要的依赖后，需要进行初始化（监听消息等）。在容器关闭时，有时候还需要清理资源（关闭连接池等）。我们通常会定义一个 `init()` 方法进行初始化，定义一个 `shutdown()` 方法进行清理，然后，引入JSR-250定义的Annotation：

```
<dependency>
  <groupId>javax.annotation</groupId>
  <artifactId>javax.annotation-api</artifactId>
  <version>1.3.2</version>
</dependency>
```

在Bean的初始化和清理方法上标记 `@PostConstruct` 和 `@PreDestroy`：

```
@Component
public class MailService {
    @Autowired(required = false)
    ZoneId zoneId = ZoneId.systemDefault();

    @PostConstruct
    public void init() {
        System.out.println("Init mail service with zoneId = " + this.zoneId);
    }

    @PreDestroy
    public void shutdown() {
        System.out.println("Shutdown mail service");
    }
}
```

Spring容器会对上述Bean做如下初始化流程：

- 调用构造方法创建 `MailService` 实例；
- 根据 `@Autowired` 进行注入；
- 调用标记有 `@PostConstruct` 的 `init()` 方法进行初始化。

而销毁时，容器会首先调用标记有 `@PreDestroy` 的 `shutdown()` 方法。

Spring只根据Annotation查找无参数方法，对方法名不作要求。

使用别名

默认情况下，对一种类型的Bean，容器只创建一个实例。但有些时候，我们需要对一种类型的Bean创建多个实例。例如，同时连接多个数据库，就必须创建多个 `DataSource` 实例。

如果我们在 `@Configuration` 类中创建了多个同类型的Bean：

```

@Configuration
@ComponentScan
public class AppConfig {
    @Bean
    ZoneId createZoneOfZ() {
        return ZoneId.of("Z");
    }

    @Bean
    ZoneId createZoneOfUTC8() {
        return ZoneId.of("UTC+08:00");
    }
}

```

Spring会报 `NoUniqueBeanDefinitionException` 异常，意思是出现了重复的Bean定义。

这个时候，需要给每个Bean添加不同的名字：

```

@Configuration
@ComponentScan
public class AppConfig {
    @Bean("z")
    ZoneId createZoneOfZ() {
        return ZoneId.of("Z");
    }

    @Bean
    @Qualifier("utc8")
    ZoneId createZoneOfUTC8() {
        return ZoneId.of("UTC+08:00");
    }
}

```

可以用 `@Bean("name")` 指定别名，也可以用 `@Bean + @Qualifier("name")` 指定别名。

存在多个同类型的Bean时，注入 `ZoneId` 又会报错：

```

NoUniqueBeanDefinitionException: No qualifying bean of type 'java.time.ZoneId'
available: expected single matching bean but found 2

```

意思是期待找到唯一的 `ZoneId` 类型Bean，但是找到两。因此，注入时，要指定Bean的名称：

```

@Component
public class MailService {
    @Autowired(required = false)
    @Qualifier("z") // 指定注入名称为"z"的ZoneId
    ZoneId zoneId = ZoneId.systemDefault();
    ...
}

```

还有一种方法是把其中某个Bean指定为 `@Primary`：

```

@Configuration
@ComponentScan
public class AppConfig {

```

```

@Bean
@Primary // 指定为主要Bean
@Qualifier("z")
ZoneId createZoneOfZ() {
    return ZoneId.of("Z");
}

@Bean
@Qualifier("utc8")
ZoneId createZoneOfUTC8() {
    return ZoneId.of("UTC+08:00");
}
}

```

这样，在注入时，如果没有指出Bean的名字，Spring会注入标记有 `@Primary` 的Bean。这种方式也很常用。例如，对于主从两个数据源，通常将主数据源定义为 `@Primary`：

```

@Configuration
@ComponentScan
public class AppConfig {
    @Bean
    @Primary
    DataSource createMasterDataSource() {
        ...
    }

    @Bean
    @Qualifier("slave")
    DataSource createSlaveDataSource() {
        ...
    }
}

```

其他Bean默认注入的就是主数据源。如果要注入从数据源，那么只需要指定名称即可。

使用FactoryBean

我们在设计模式的[工厂方法](#)中讲到，很多时候，可以通过工厂模式创建对象。Spring也提供了工厂模式，允许定义一个工厂，然后由工厂创建真正的Bean。

用工厂模式创建Bean需要实现 `FactoryBean` 接口。我们观察下面的代码：

```

@Component
public class ZoneIdFactoryBean implements FactoryBean<ZoneId> {

    String zone = "Z";

    @Override
    public ZoneId getObject() throws Exception {
        return ZoneId.of(zone);
    }

    @Override
    public Class<?> getObjectType() {
        return ZoneId.class;
    }
}

```



```
}
```

当一个Bean实现了 `FactoryBean` 接口后，Spring会先实例化这个工厂，然后调用 `getObject()` 创建真正的Bean。`getObjectType()` 可以指定创建的Bean的类型，因为指定类型不一定与实际类型一致，可以是接口或抽象类。

因此，如果定义了一个 `FactoryBean`，要注意Spring创建的Bean实际上是这个 `FactoryBean` 的 `getObject()` 方法返回的Bean。为了和普通Bean区分，我们通常都以 `xxxFactoryBean` 命名。

练习

从  **gitee** 下载练习：[定制Bean](#)（推荐使用[IDE练习插件](#)快速下载）

小结

Spring默认使用Singleton创建Bean，也可指定Scope为Prototype；

可将相同类型的Bean注入 `List`；

可用 `@Autowired(required=false)` 允许可选注入；

可用带 `@Bean` 标注的方法创建Bean；

可使用 `@PostConstruct` 和 `@PreDestroy` 对Bean进行初始化和清理；

相同类型的Bean只能有一个指定为 `@Primary`，其他必须用 `@Qualifier("beanName")` 指定别名；

注入时，可通过别名 `@Qualifier("beanName")` 指定某个Bean；

可以定义 `FactoryBean` 来使用工厂模式创建Bean。

使用Resource

在Java程序中，我们经常会读取配置文件、资源文件等。使用Spring容器时，我们也可以把“文件”注入进来，方便程序读取。

例如，AppService需要读取 `logo.txt` 这个文件，通常情况下，我们需要写很多繁琐的代码，主要是为了定位文件，打开InputStream。

Spring提供了一个 `org.springframework.core.io.Resource`（注意不是 `javax.annotation.Resource`），它可以像 `String`、`int` 一样使用 `@value` 注入：

```
@Component
public class AppService {
    @Value("classpath:/logo.txt")
    private Resource resource;

    private String logo;

    @PostConstruct
    public void init() throws IOException {
        try (var reader = new BufferedReader(
            new InputStreamReader(resource.getInputStream(),
                StandardCharsets.UTF_8))) {
            this.logo = reader.lines().collect(Collectors.joining("\n"));
        }
    }
}
```

注入 `Resource` 最常用的方式是通过 `classpath`，即类似 `classpath:/logo.txt` 表示在 `classpath` 中搜索 `logo.txt` 文件，然后，我们直接调用 `Resource.getInputStream()` 就可以获取到输入流，避免了自己搜索文件的代码。

也可以直接指定文件的路径，例如：

```
@value("file:/path/to/logo.txt")
private Resource resource;
```

但使用 `classpath` 是最简单的方式。上述工程结构如下：

```
spring-ioc-resource
├─ pom.xml
├─ src
│   └─ main
│       ├── java
│       │   └─ com
│       │       └─ itranswarp
│       │           └─ learnjava
│       │               ├── AppConfig.java
│       │               └─ AppService.java
│       └─ resources
│           └─ logo.txt
```

使用Maven的标准目录结构，所有资源文件放入 `src/main/resources` 即可。

练习

使用Spring的 `Resource` 注入 `app.properties` 文件，然后读取该配置文件。

从  **gitee** 下载练习：[使用Resource](#)（推荐使用[IDE练习插件](#)快速下载）

小结

Spring提供了 `Resource` 类便于注入资源文件。

最常用的注入是通过 `classpath` 以 `classpath:/path/to/file` 的形式注入。

注入配置

在开发应用程序时，经常需要读取配置文件。最常用的配置方法是以 `key=value` 的形式写在 `.properties` 文件中。

例如，`MailService` 根据配置的 `app.zone=Asia/Shanghai` 来决定使用哪个时区。要读取配置文件，我们可以使用上一节讲到的 `Resource` 来读取位于 `classpath` 下的一个 `app.properties` 文件。但是，这样仍然比较繁琐。

Spring容器还提供了一个更简单的 `@PropertySource` 来自动读取配置文件。我们只需要在 `@Configuration` 配置类上再添加一个注解：

```

@Configuration
@ComponentScan
@PropertySource("app.properties") // 表示读取classpath的app.properties
public class AppConfig {
    @Value("${app.zone:Z}")
    String zoneId;

    @Bean
    ZoneId createZoneId() {
        return ZoneId.of(zoneId);
    }
}

```

Spring容器看到 `@PropertySource("app.properties")` 注解后，自动读取这个配置文件，然后，我们使用 `@Value` 正常注入：

```

@Value("${app.zone:Z}")
String zoneId;

```

注意注入的字符串语法，它的格式如下：

- `"${app.zone}"` 表示读取key为 `app.zone` 的value，如果key不存在，启动将报错；
- `"${app.zone:Z}"` 表示读取key为 `app.zone` 的value，但如果key不存在，就使用默认值 `Z`。

这样一来，我们就可以根据 `app.zone` 的配置来创建 `zoneId`。

还可以把注入的注解写到方法参数中：

```

@Bean
ZoneId createZoneId(@Value("${app.zone:Z}") String zoneId) {
    return ZoneId.of(zoneId);
}

```

可见，先使用 `@PropertySource` 读取配置文件，然后通过 `@Value` 以 `${key:defaultValue}` 的形式注入，可以极大地简化读取配置的麻烦。

另一种注入配置的方式是先通过一个简单的JavaBean持有所有的配置，例如，一个 `SmtplibConfig`：

```

@Component
public class SmtplibConfig {
    @Value("${smtp.host}")
    private String host;

    @Value("${smtp.port:25}")
    private int port;

    public String getHost() {
        return host;
    }

    public int getPort() {
        return port;
    }
}

```

然后，在需要读取的地方，使用 `#{smtpConfig.host}` 注入：

```
@Component
public class MailService {
    @Value("#{smtpConfig.host}")
    private String smtpHost;

    @Value("#{smtpConfig.port}")
    private int smtpPort;
}
```

注意观察 `#{}` 这种注入语法，它和 `${key}` 不同的是，`#{}` 表示从JavaBean读取属性。`"#{smtpConfig.host}"` 的意思是，从名称为 `smtpConfig` 的Bean读取 `host` 属性，即调用 `getHost()` 方法。一个Class名为 `SmtpConfig` 的Bean，它在Spring容器中的默认名称就是 `smtpConfig`，除非用 `@Qualifier` 指定了名称。

使用一个独立的JavaBean持有所有属性，然后在其他Bean中以 `#{bean.property}` 注入的好处是，多个Bean都可以引用同一个Bean的某个属性。例如，如果 `SmtpConfig` 决定从数据库中读取相关配置项，那么 `MailService` 注入的 `@Value("#{smtpConfig.host}")` 仍然可以不修改正常运行。

练习

从  **gitee** 下载练习：[注入SMTP配置](#)（推荐使用[IDE练习插件](#)快速下载）

小结

Spring容器可以通过 `@PropertySource` 自动读取配置，并以 `@Value("${key}")` 的形式注入；

可以通过 `${key:defaultValue}` 指定默认值；

以 `#{bean.property}` 形式注入时，Spring容器自动把指定Bean的指定属性值注入。

使用条件装配

开发应用程序时，我们会使用开发环境，例如，使用内存数据库以便快速启动。而运行在生产环境时，我们会使用生产环境，例如，使用MySQL数据库。如果应用程序可以根据自身的环境做一些适配，无疑会更加灵活。

Spring为应用程序准备了Profile这一概念，用来表示不同的环境。例如，我们分别定义开发、测试和生产这3个环境：

- native
- test
- production

创建某个Bean时，Spring容器可以根据注解 `@Profile` 来决定是否创建。例如，以下配置：

```
@Configuration
@ComponentScan
public class AppConfig {
    @Bean
    @Profile("!test")
    ZoneId createZoneId() {
        return ZoneId.systemDefault();
    }

    @Bean
```

```

@Profile("test")
ZoneId createZoneIdForTest() {
    return ZoneId.of("America/New_York");
}
}

```

如果当前的Profile设置为 `test`，则Spring容器会调用 `createZoneIdForTest()` 创建 `ZoneId`，否则，调用 `createZoneId()` 创建 `ZoneId`。注意到 `@Profile("!test")` 表示非test环境。

在运行程序时，加上JVM参数 `-Dspring.profiles.active=test` 就可以指定以 `test` 环境启动。

实际上，Spring允许指定多个Profile，例如：

```
-Dspring.profiles.active=test, master
```

可以表示 `test` 环境，并使用 `master` 分支代码。

要满足多个Profile条件，可以这样写：

```

@Bean
@Profile({ "test", "master" }) // 同时满足test和master
ZoneId createZoneId() {
    ...
}

```

使用Conditional

除了根据 `@Profile` 条件来决定是否创建某个Bean外，Spring还可以根据 `@Conditional` 决定是否创建某个Bean。

例如，我们对 `SmtplibService` 添加如下注解：

```

@Component
@Conditional(OnSmtplibEnvCondition.class)
public class SmtplibService implements MailService {
    ...
}

```

它的意思是，如果满足 `OnSmtplibEnvCondition` 的条件，才会创建 `SmtplibService` 这个Bean。

`OnSmtplibEnvCondition` 的条件是什么呢？我们看一下代码：

```

public class OnSmtplibEnvCondition implements Condition {
    public boolean matches(ConditionContext context, AnnotatedTypeMetadata metadata) {
        return "true".equalsIgnoreCase(System.getenv("smtp"));
    }
}

```

因此，`OnSmtplibEnvCondition` 的条件是存在环境变量 `smtp`，值为 `true`。这样，我们就可以通过环境变量来控制是否创建 `SmtplibService`。

Spring只提供了 `@Conditional` 注解，具体判断逻辑还需要我们自己实现。Spring Boot提供了更多使用起来更简单的条件注解，例如，如果配置文件中存在 `app.smtp=true`，则创建 `MailService`：

```
@Component
@ConditionalOnProperty(name="app.smtp", havingValue="true")
public class MailService {
    ...
}
```

如果当前classpath中存在类 `javax.mail.Transport`，则创建 `MailService`：

```
@Component
@ConditionalOnClass(name = "javax.mail.Transport")
public class MailService {
    ...
}
```

后续我们会介绍Spring Boot的条件装配。我们以文件存储为例，假设我们需要保存用户上传的头像，并返回存储路径，在本地开发运行时，我们总是存储到文件：

```
@Component
@ConditionalOnProperty(name = "app.storage", havingValue = "file", matchIfMissing = true)
public class FileUploader implements Uploader {
    ...
}
```

在生产环境运行时，我们会把文件存储到类似AWS S3上：

```
@Component
@ConditionalOnProperty(name = "app.storage", havingValue = "s3")
public class S3Uploader implements Uploader {
    ...
}
```

其他需要存储的服务则注入 `Uploader`：

```
@Component
public class UserService {
    @Autowired
    Uploader uploader;
}
```

当应用程序检测到配置文件存在 `app.storage=s3` 时，自动使用 `S3Uploader`，如果存在配置 `app.storage=file`，或者配置 `app.storage` 不存在，则使用 `FileUploader`。

可见，使用条件注解，能更灵活地装配Bean。

练习

从  **gitee** 下载练习：[使用@Profile进行条件装配](#)（推荐使用[IDE练习插件](#)快速下载）

小结

Spring允许通过 `@Profile` 配置不同的Bean；

Spring还提供了 `@Conditional` 来进行条件装配，Spring Boot在此基础上进一步提供了基于配置、Class、Bean等条件进行装配。

使用AOP

AOP是Aspect Oriented Programming，即面向切面编程。

那什么是AOP？

我们先回顾一下OOP：Object Oriented Programming，OOP作为面向对象编程的模式，获得了巨大的成功，OOP的主要功能是数据封装、继承和多态。

而AOP是一种新的编程方式，它和OOP不同，OOP把系统看作多个对象的交互，AOP把系统分解为不同的关注点，或者称之为切面（Aspect）。

要理解AOP的概念，我们先用OOP举例，比如一个业务组件 `BookService`，它有几个业务方法：

- `createBook`：添加新的Book；
- `updateBook`：修改Book；
- `deleteBook`：删除Book。

对每个业务方法，例如，`createBook()`，除了业务逻辑，还需要安全检查、日志记录和事务处理，它的代码像这样：

```
public class BookService {
    public void createBook(Book book) {
        securityCheck();
        Transaction tx = startTransaction();
        try {
            // 核心业务逻辑
            tx.commit();
        } catch (RuntimeException e) {
            tx.rollback();
            throw e;
        }
        log("created book: " + book);
    }
}
```

继续编写 `updateBook()`，代码如下：

```
public class BookService {
    public void updateBook(Book book) {
        securityCheck();
        Transaction tx = startTransaction();
        try {
            // 核心业务逻辑
            tx.commit();
        } catch (RuntimeException e) {
            tx.rollback();
            throw e;
        }
        log("updated book: " + book);
    }
}
```

```
}
```

对于安全检查、日志、事务等代码，它们会重复出现在每个业务方法中。使用OOP，我们很难将这些四处分散的代码模块化。

考察业务模型可以发现，`BookService` 关系的是自身的核心逻辑，但整个系统还要求关注安全检查、日志、事务等功能，这些功能实际上“横跨”多个业务方法，为了实现这些功能，不得不在每个业务方法上重复编写代码。

一种可行的方式是使用[Proxy模式](#)，将某个功能，例如，权限检查，放入Proxy中：

```
public class SecurityCheckBookService implements BookService {
    private final BookService target;

    public SecurityCheckBookService(BookService target) {
        this.target = target;
    }

    public void createBook(Book book) {
        securityCheck();
        target.createBook(book);
    }

    public void updateBook(Book book) {
        securityCheck();
        target.updateBook(book);
    }

    public void deleteBook(Book book) {
        securityCheck();
        target.deleteBook(book);
    }

    private void securityCheck() {
        ...
    }
}
```

这种方式的缺点是比较麻烦，必须先抽取接口，然后，针对每个方法实现Proxy。

另一种方法是，既然 `SecurityCheckBookService` 的代码都是标准的Proxy样板代码，不如把权限检查视作一种切面（Aspect），把日志、事务也视为切面，然后，以某种自动化的方式，把切面织入到核心逻辑中，实现Proxy模式。

如果我们以AOP的视角来编写上述业务，可以依次实现：

1. 核心逻辑，即BookService；
2. 切面逻辑，即：
3. 权限检查的Aspect；
4. 日志的Aspect；
5. 事务的Aspect。

然后，以某种方式，让框架来把上述3个Aspect以Proxy的方式“织入”到 `BookService` 中，这样一来，就不必编写复杂而冗长的Proxy模式。

AOP原理

如何把切面织入到核心逻辑中？这正是AOP需要解决的问题。换句话说，如果客户端获得了 `BookService` 的引用，当调用 `bookService.createBook()` 时，如何对调用方法进行拦截，并在拦截前后进行安全检查、日志、事务等处理，就相当于完成了所有业务功能。

在Java平台上，对于AOP的织入，有3种方式：

1. 编译期：在编译时，由编译器把切面调用编译进字节码，这种方式需要定义新的关键字并扩展编译器，AspectJ就扩展了Java编译器，使用关键字aspect来实现织入；
2. 类加载器：在目标类被装载到JVM时，通过一个特殊的类加载器，对目标类的字节码重新“增强”；
3. 运行期：目标对象和切面都是普通Java类，通过JVM的动态代理功能或者第三方库实现运行期动态织入。

最简单的方式是第三种，Spring的AOP实现就是基于JVM的动态代理。由于JVM的动态代理要求必须实现接口，如果一个普通类没有业务接口，就需要通过CGLIB或者Javassist这些第三方库实现。

AOP技术看上去比较神秘，但实际上，它本质就是一个动态代理，让我们把一些常用功能如权限检查、日志、事务等，从每个业务方法中剥离出来。

需要特别指出的是，AOP对于解决特定问题，例如事务管理非常有用，这是因为分散在各处的事务代码几乎是完全相同的，并且它们需要的参数（JDBC的Connection）也是固定的。另一些特定问题，如日志，就不那么容易实现，因为日志虽然简单，但打印日志的时候，经常需要捕获局部变量，如果使用AOP实现日志，我们只能输出固定格式的日志，因此，使用AOP时，必须适合特定的场景。

装配AOP

在AOP编程中，我们经常会遇到下面的概念：

- Aspect：切面，即一个横跨多个核心逻辑的功能，或者称之为系统关注点；
- Joinpoint：连接点，即定义在应用程序流程的何处插入切面的执行；
- Pointcut：切入点，即一组连接点的集合；
- Advice：增强，指特定连接点上执行的动作；
- Introduction：引介，指为一个已有的Java对象动态地增加新的接口；
- Weaving：织入，指将切面整合到程序的执行流程中；
- Interceptor：拦截器，是一种实现增强的方式；
- Target Object：目标对象，即真正执行业务的核心逻辑对象；
- AOP Proxy：AOP代理，是客户端持有的增强后的对象引用。

看完上述术语，是不是感觉对AOP有了进一步的困惑？其实，我们不用关心AOP创造的“术语”，只需要理解AOP本质上只是一种代理模式的实现方式，在Spring的容器是实现AOP特别方便。

我们以 `UserService` 和 `MailService` 为例，这两个属于核心业务逻辑，现在，我们准备给 `UserService` 的每个业务方法执行前添加日志，给 `MailService` 的每个业务方法执行前后添加日志，在Spring中，需要以下步骤：

首先，我们通过Maven引入Spring对AOP的支持：

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-aspects</artifactId>
  <version>${spring.version}</version>
</dependency>
```

上述依赖会自动引入AspectJ，使用AspectJ实现AOP比较方便，因为它的定义比较简单。

然后，我们定义一个 `LoggingAspect`：

```

@Aspect
@Component
public class LoggingAspect {
    // 在执行UserService的每个方法前执行：
    @Before("execution(public * com.itranswarp.learnjava.service.UserService.*(..))")
    public void doAccessCheck() {
        System.err.println("[Before] do access check...");
    }

    // 在执行MailService的每个方法前后执行：
    @Around("execution(public * com.itranswarp.learnjava.service.MailService.*(..))")
    public Object doLogging(ProceedingJoinPoint pjp) throws Throwable {
        System.err.println("[Around] start " + pjp.getSignature());
        Object retVal = pjp.proceed();
        System.err.println("[Around] done " + pjp.getSignature());
        return retVal;
    }
}

```

观察 `doAccessCheck()` 方法，我们定义了一个 `@Before` 注解，后面的字符串是告诉AspectJ应该在何处执行该方法，这里写的意思是：执行 `UserService` 的每个 `public` 方法前执行 `doAccessCheck()` 代码。

再观察 `doLogging()` 方法，我们定义了一个 `@Around` 注解，它和 `@Before` 不同，`@Around` 可以决定是否执行目标方法，因此，我们在 `doLogging()` 内部先打印日志，再调用方法，最后打印日志后返回结果。

在 `LoggingAspect` 类的声明处，除了用 `@Component` 表示它本身也是一个Bean外，我们再加上 `@Aspect` 注解，表示它的 `@Before` 标注的方法需要注入到 `UserService` 的每个 `public` 方法执行前，`@Around` 标注的方法需要注入到 `MailService` 的每个 `public` 方法执行前后。

紧接着，我们需要给 `@Configuration` 类加上一个 `@EnableAspectJAutoProxy` 注解：

```

@Configuration
@ComponentScan
@EnableAspectJAutoProxy
public class AppConfig {
    ...
}

```

Spring的IoC容器看到这个注解，就会自动查找带有 `@Aspect` 的Bean，然后根据每个方法的 `@Before`、`@Around` 等注解把AOP注入到特定的Bean中。执行代码，我们可以看到以下输出：

```
[Before] do access check...
[Around] start void
com.itranswarp.learnjava.service.MailService.sendRegistrationMail(User)
welcome, test!
[Around] done void
com.itranswarp.learnjava.service.MailService.sendRegistrationMail(User)
[Before] do access check...
[Around] start void
com.itranswarp.learnjava.service.MailService.sendLoginMail(User)
Hi, Bob! You are logged in at 2020-02-14T23:13:52.167996+08:00[Asia/Shanghai]
[Around] done void
com.itranswarp.learnjava.service.MailService.sendLoginMail(User)
```

这说明执行业务逻辑前后，确实执行了我们定义的Aspect（即 `LoggingAspect` 的方法）。

有些童鞋会问，`LoggingAspect` 定义的方法，是如何注入到其他Bean的呢？

其实AOP的原理非常简单。我们以 `LoggingAspect.doAccessCheck()` 为例，要把它注入到 `UserService` 的每个 `public` 方法中，最简单的方法是编写一个子类，并持有原始实例的引用：

```
public UserServiceAopProxy extends UserService {
    private UserService target;
    private LoggingAspect aspect;

    public UserServiceAopProxy(UserService target, LoggingAspect aspect) {
        this.target = target;
        this.aspect = aspect;
    }

    public User login(String email, String password) {
        // 先执行Aspect的代码：
        aspect.doAccessCheck();
        // 再执行UserService的逻辑：
        return target.login(email, password);
    }

    public User register(String email, String password, String name) {
        aspect.doAccessCheck();
        return target.register(email, password, name);
    }

    ...
}
```

这些都是Spring容器启动时为我们自动创建的注入了Aspect的子类，它取代了原始的 `UserService`（原始的 `UserService` 实例作为内部变量隐藏在 `UserServiceAopProxy` 中）。如果我们打印从Spring容器获取的 `UserService` 实例类型，它类似 `UserService$$EnhancerBySpringCGLIB$$1f44e01c`，实际上是Spring使用CGLIB动态创建的子类，但对于调用方来说，感觉不到任何区别。

Spring对接口类型使用JDK动态代理，对普通类使用CGLIB创建子类。如果一个Bean的class是final，Spring将无法为其创建子类。

可见，虽然Spring容器内部实现AOP的逻辑比较复杂（需要使用AspectJ解析注解，并通过CGLIB实现代理类），但我们使用AOP非常简单，一共需要三步：

1. 定义执行方法，并在方法上通过AspectJ的注解告诉Spring应该在何处调用此方法；

2. 标记 `@Component` 和 `@Aspect` ;
3. 在 `@Configuration` 类上标注 `@EnableAspectJAutoProxy` 。

至于AspectJ的注入语法则比较复杂, 请参考[Spring文档](#)。

Spring也提供其他方法来装配AOP, 但都没有使用AspectJ注解的方式来得简洁明了, 所以我们不再作介绍。

拦截器类型

顾名思义, 拦截器有以下类型:

- `@Before`: 这种拦截器先执行拦截代码, 再执行目标代码。如果拦截器抛异常, 那么目标代码就不执行了;
- `@After`: 这种拦截器先执行目标代码, 再执行拦截器代码。无论目标代码是否抛异常, 拦截器代码都会执行;
- `@AfterReturning`: 和`@After`不同的是, 只有当目标代码正常返回时, 才执行拦截器代码;
- `@AfterThrowing`: 和`@After`不同的是, 只有当目标代码抛出了异常时, 才执行拦截器代码;
- `@Around`: 能完全控制目标代码是否执行, 并可以在执行前后、抛异常后执行任意拦截代码, 可以说是包含了上面所有功能。

练习

从  **gitee** 下载练习: [使用AOP实现日志](#) (推荐使用[IDE练习插件](#)快速下载)

小结

在Spring容器中使用AOP非常简单, 只需要定义执行方法, 并用AspectJ的注解标注应该在何处触发并执行。

Spring通过CGLIB动态创建子类等方式来实现AOP代理模式, 大大简化了代码。

使用注解装配AOP

上一节我们讲解了使用AspectJ的注解, 并配合一个复杂的 `execution(* xxx.Xyz.*(..))` 语法来定义应该如何装配AOP。

在实际项目中, 这种写法其实很少使用。假设你写了一个 `SecurityAspect` :

```
@Aspect
@Component
public class SecurityAspect {
    @Before("execution(public * com.itranswarp.learnjava.service.*.*(..))")
    public void check() {
        if (SecurityContext.getCurrentUser() == null) {
            throw new RuntimeException("check failed");
        }
    }
}
```

基本能实现无差别全覆盖, 即某个包下面的所有Bean的所有方法都会被这个 `check()` 方法拦截。

还有的童鞋喜欢用方法名前缀进行拦截:

```

@Around("execution(public * update*(..))")
public Object doLogging(ProceedingJoinPoint pjp) throws Throwable {
    // 对update开头的方法切换数据源:
    String old = setCurrentDataSource("master");
    Object retVal = pjp.proceed();
    restoreCurrentDataSource(old);
    return retVal;
}

```

这种非精准打击误伤面更大，因为从方法前缀区分是否是数据库操作是非常不可取的。

我们在使用AOP时，要注意到虽然Spring容器可以把指定的方法通过AOP规则装配到指定的Bean的指定方法前后，但是，如果自动装配时，因为不恰当的范围，容易导致意想不到的结果，即很多不需要AOP代理的Bean也被自动代理了，并且，后续新增的Bean，如果不清楚现有的AOP装配规则，容易被强迫装配。

使用AOP时，被装配的Bean最好自己能清清楚楚地知道自己被安排了。例如，Spring提供的 `@Transactional` 就是一个非常好的例子。如果我们自己写的Bean希望在一个数据库事务中被调用，就标注上 `@Transactional`：

```

@Component
public class UserService {
    // 有事务:
    @Transactional
    public User createUser(String name) {
        ...
    }

    // 无事务:
    public boolean isValidName(String name) {
        ...
    }

    // 有事务:
    @Transactional
    public void updateUser(User user) {
        ...
    }
}

```

或者直接在class级别注解，表示“所有public方法都被安排了”：

```

@Component
@Transactional
public class UserService {
    ...
}

```

通过 `@Transactional`，某个方法是否启用了事务就一清二楚了。因此，装配AOP的时候，使用注解是最好的方式。

我们以一个实际例子演示如何使用注解实现AOP装配。为了监控应用程序的性能，我们定义一个性能监控的注解：

```

@Target(METHOD)
@Retention(RUNTIME)
public @interface MetricTime {
    String value();
}

```

在需要被监控的关键方法上标注该注解：

```

@Component
public class UserService {
    // 监控register()方法性能:
    @MetricTime("register")
    public User register(String email, String password, String name) {
        ...
    }
    ...
}

```

然后，我们定义 `MetricAspect`：

```

@Aspect
@Component
public class MetricAspect {
    @Around("@annotation(metricTime)")
    public Object metric(ProceedingJoinPoint joinPoint, MetricTime metricTime)
        throws Throwable {
        String name = metricTime.value();
        long start = System.currentTimeMillis();
        try {
            return joinPoint.proceed();
        } finally {
            long t = System.currentTimeMillis() - start;
            // 写入日志或发送至JMX:
            System.err.println("[Metrics] " + name + ": " + t + "ms");
        }
    }
}

```

注意 `metric()` 方法标注了 `@Around("@annotation(metricTime)")`，它的意思是，符合条件的目标方法是带有 `@MetricTime` 注解的方法，因为 `metric()` 方法参数类型是 `MetricTime`（注意参数名是 `metricTime` 不是 `MetricTime`），我们通过它获取性能监控的名称。

有了 `@MetricTime` 注解，再配合 `MetricAspect`，任何Bean，只要方法标注了 `@MetricTime` 注解，就可以自动实现性能监控。运行代码，输出结果如下：

```

welcome, Bob!
[Metrics] register: 16ms

```

练习

从  **gitee** 下载练习：[使用注解+AOP实现性能监控](#)（推荐使用[IDE练习插件](#)快速下载）

小结

使用注解实现AOP需要先定义注解，然后使用 `@Around("@annotation(name)")` 实现装配；

使用注解既简单，又能明确标识AOP装配，是使用AOP推荐的方式。

AOP避坑指南

无论是使用AspectJ语法，还是配合Annotation，使用AOP，实际上就是让Spring自动为我们创建一个Proxy，使得调用方能无感知地调用指定方法，但运行期却动态“织入”了其他逻辑，因此，AOP本质上就是一个[代理模式](#)。

因为Spring使用了CGLIB来实现运行期动态创建Proxy，如果我们没能深入理解其运行原理和实现机制，就极有可能遇到各种诡异的问题。

我们来看一个实际的例子。

假设我们定义了一个 `UserService` 的Bean：

```
@Component
public class UserService {
    // 成员变量：
    public final ZoneId zoneId = ZoneId.systemDefault();

    // 构造方法：
    public UserService() {
        System.out.println("UserService(): init...");
        System.out.println("UserService(): zoneId = " + this.zoneId);
    }

    // public方法：
    public ZoneId getZoneId() {
        return zoneId;
    }

    // public final方法：
    public final ZoneId getFinalZoneId() {
        return zoneId;
    }
}
```

再写个 `MailService`，并注入 `UserService`：

```
@Component
public class MailService {
    @Autowired
    UserService userService;

    public String sendMail() {
        ZoneId zoneId = userService.zoneId;
        String dt = ZonedDateTime.now(zoneId).toString();
        return "Hello, it is " + dt;
    }
}
```

最后用 `main()` 方法测试一下：

```

@Configuration
@ComponentScan
public class AppConfig {
    public static void main(String[] args) {
        ApplicationContext context = new
        AnnotationConfigApplicationContext(AppConfig.class);
        MailService mailService = context.getBean(MailService.class);
        System.out.println(mailService.sendMail());
    }
}

```

查看输出，一切正常：

```

UserService(): init...
UserService(): zoneId = Asia/Shanghai
Hello, it is 2020-04-12T10:23:22.917721+08:00[Asia/Shanghai]

```

下一步，我们给 `UserService` 加上AOP支持，就添加一个最简单的 `LoggingAspect`：

```

@Aspect
@Component
public class LoggingAspect {
    @Before("execution(public * com..*.UserService.*(..))")
    public void doAccessCheck() {
        System.err.println("[Before] do access check...");
    }
}

```

别忘了在 `AppConfig` 上加上 `@EnableAspectJAutoProxy`。再次运行，不出意外的话，会得到一个 `NullPointerException`：

```

Exception in thread "main" java.lang.NullPointerException: zone
    at java.base/java.util.Objects.requireNonNull(Objects.java:246)
    at java.base/java.time.Clock.system(Clock.java:203)
    at java.base/java.time.ZonedDateTime.now(ZonedDateTime.java:216)
    at
com.itranswarp.learnjava.service.MailService.sendMail(MailService.java:19)
    at com.itranswarp.learnjava.AppConfig.main(AppConfig.java:21)

```

仔细跟踪代码，会发现 `null` 值出现在 `MailService.sendMail()` 内部的这一行代码：

```

@Component
public class MailService {
    @Autowired
    UserService userService;

    public String sendMail() {
        ZoneId zoneId = userService.zoneId;
        System.out.println(zoneId); // null
        ...
    }
}

```

我们还故意在 `UserService` 中特意用 `final` 修饰了一下成员变量：


```
@Component
public class UserService {
    public final ZoneId zoneId = ZoneId.systemDefault();
    ...
}
```

用 `final` 标注的成员变量为 `null`？逗我呢？

怎么肥四？

为什么加了AOP就报NPE，去了AOP就一切正常？`final` 字段不执行，难道JVM有问题？为了解答这个诡异的问题，我们需要深入理解Spring使用CGLIB生成Proxy的原理：

第一步，正常创建一个 `UserService` 的原始实例，这是通过反射调用构造方法实现的，它的行为和我们预期的完全一致；

第二步，通过CGLIB创建一个 `UserService` 的子类，并引用了原始实例和 `LoggingAspect`：

```
public UserService$$EnhancerBySpringCGLIB extends UserService {
    UserService target;
    LoggingAspect aspect;

    public UserService$$EnhancerBySpringCGLIB() {
    }

    public ZoneId getZoneId() {
        aspect.doAccessCheck();
        return target.getZoneId();
    }
}
```

如果我们观察Spring创建的AOP代理，它的类名总是类似

`UserService$$EnhancerBySpringCGLIB$$1c76af9d`（你没看错，Java的类名实际上允许 `$` 字符）。为了让调用方获得 `UserService` 的引用，它必须继承自 `UserService`。然后，该代理类会覆写所有 `public` 和 `protected` 方法，并在内部将调用委托给原始的 `UserService` 实例。

这里出现了两个 `UserService` 实例：

一个是我们代码中定义的 *原始实例*，它的成员变量已经按照我们预期的方式被初始化完成：

```
UserService original = new UserService();
```

第二个 `UserService` 实例实际上类型是 `UserService$$EnhancerBySpringCGLIB`，它引用了原始的 `UserService` 实例：

```
UserService$$EnhancerBySpringCGLIB proxy = new
UserService$$EnhancerBySpringCGLIB();
proxy.target = original;
proxy.aspect = ...
```

注意到这种情况仅出现在启用了AOP的情况，此刻，从 `ApplicationContext` 中获取的 `UserService` 实例是 `proxy`，注入到 `MailService` 中的 `UserService` 实例也是 `proxy`。

那么最终的问题来了：`proxy`实例的成员变量，也就是从 `UserService` 继承的 `zoneId`，它的值是 `null`。

原因在于，UserService 成员变量的初始化：

```
public class UserService {
    public final ZoneId zoneId = ZoneId.systemDefault();
    ...
}
```

在 UserService\$\$EnhancerBySpringCGLIB 中，并未执行。原因是，没必要初始化 proxy 的成员变量，因为 proxy 的目的是代理方法。

实际上，成员变量的初始化是在构造方法中完成的。这是我们要看到的代码：

```
public class UserService {
    public final ZoneId zoneId = ZoneId.systemDefault();
    public UserService() {
    }
}
```

这是编译器实际编译的代码：

```
public class UserService {
    public final ZoneId zoneId;
    public UserService() {
        super(); // 构造方法的第一行代码总是调用super()
        zoneId = ZoneId.systemDefault(); // 继续初始化成员变量
    }
}
```

然而，对于 Spring 通过 CGLIB 动态创建的 UserService\$\$EnhancerBySpringCGLIB 代理类，它的构造方法中，并未调用 super()，因此，从父类继承的成员变量，包括 final 类型的成员变量，统统都没有初始化。

有的童鞋会问：Java 语言规定，任何类的构造方法，第一行必须调用 super()，如果没有，编译器会自动加上，怎么 Spring 的 CGLIB 就可以搞特殊？

这是因为自动加 super() 的功能是 Java 编译器实现的，它发现你没加，就自动给加上，发现你加错了，就报编译错误。但实际上，如果直接构造字节码，一个类的构造方法中，不一定非要调用 super()。Spring 使用 CGLIB 构造的 Proxy 类，是直接生成字节码，并没有源码-编译-字节码这个步骤，因此：

Spring 通过 CGLIB 创建的代理类，不会初始化代理类自身继承的任何成员变量，包括 final 类型的成员变量！

再考察 MailService 的代码：

```
@Component
public class MailService {
    @Autowired
    UserService userService;

    public String sendMail() {
        ZoneId zoneId = userService.zoneId;
        System.out.println(zoneId); // null
        ...
    }
}
```

如果没有启用AOP，注入的是原始的 `UserService` 实例，那么一切正常，因为 `UserService` 实例的 `zoneId` 字段已经被正确初始化了。

如果启动了AOP，注入的是代理后的 `UserService$$EnhancerBySpringCGLIB` 实例，那么问题大了：获取的 `UserService$$EnhancerBySpringCGLIB` 实例的 `zoneId` 字段，永远为 `null`。

那么问题来了：启用了AOP，如何修复？

修复很简单，只需要把直接访问字段的代码，改为通过方法访问：

```
@Component
public class MailService {
    @Autowired
    UserService userService;

    public String sendMail() {
        // 不要直接访问UserService的字段：
        ZoneId zoneId = userService.getZoneId();
        ...
    }
}
```

无论注入的 `UserService` 是原始实例还是代理实例，`getZoneId()` 都能正常工作，因为代理类会覆写 `getZoneId()` 方法，并将其委托给原始实例：

```
public UserService$$EnhancerBySpringCGLIB extends UserService {
    UserService target = ...
    ...

    public ZoneId getZoneId() {
        return target.getZoneId();
    }
}
```

注意到我们还给 `UserService` 添加了一个 `public + final` 的方法：

```
@Component
public class UserService {
    ...
    public final ZoneId getFinalZoneId() {
        return zoneId;
    }
}
```

如果在 `MailService` 中，调用的不是 `getZoneId()`，而是 `getFinalZoneId()`，又会出现 `NullPointerException`，这是因为，代理类无法覆写 `final` 方法（这一点绕不过JVM的ClassLoader检查），该方法返回的是代理类的 `zoneId` 字段，即 `null`。

实际上，如果我们加上日志，Spring在启动时会打印一个警告：

```
10:43:09.929 [main] DEBUG org.springframework.aop.framework.CglibAopProxy - Final
method [public final java.time.ZoneId xxx.UserService.getFinalZoneId()] cannot
get proxied via CGLIB: Calls to this method will NOT be routed to the target
instance and might lead to NPEs against uninitialized fields in the proxy
instance.
```

上面的日志大意就是，因为被代理的 `UserService` 有一个 `final` 方法 `getFinalZoneId()`，这会导致其他Bean如果调用此方法，无法将其代理到真正的原始实例，从而可能发生NPE异常。

因此，正确使用AOP，我们需要一个避坑指南：

1. 访问被注入的Bean时，总是调用方法而非直接访问字段；
2. 编写Bean时，如果可能会被代理，就不要编写 `public final` 方法。

这样才能保证有没有AOP，代码都能正常工作。

思考

为什么Spring刻意不初始化Proxy继承的字段？

如果一个Bean不允许任何AOP代理，应该怎么做来“保护”自己在运行期不会被代理？

练习

从  **gitee** 下载练习：[修复启用AOP导致的NPE](#)（推荐使用[IDE练习插件](#)快速下载）

小结

由于Spring通过CGLIB实现代理类，我们要避免直接访问Bean的字段，以及由 `final` 方法带来的“未代理”问题。

遇到CglibAopProxy的相关日志，务必要仔细检查，防止因为AOP出现NPE异常。

访问数据库

数据库基本上是现代应用程序的标准存储，绝大多数程序都把自己的业务数据存储存储在关系数据库中，可见，访问数据库几乎是所有应用程序必备能力。

我们在前面已经介绍了Java程序访问数据库的标准接口JDBC，它的实现方式非常简洁，即：Java标准库定义接口，各数据库厂商以“驱动”的形式实现接口。应用程序要使用哪个数据库，就把该数据库厂商的驱动以jar包形式引入进来，同时自身仅使用JDBC接口，编译期并不需要特定厂商的驱动。

使用JDBC虽然简单，但代码比较繁琐。Spring为了简化数据库访问，主要做了以下几点工作：

- 提供了简化的访问JDBC的模板类，不必手动释放资源；
- 提供了一个统一的DAO类以实现Data Access Object模式；
- 把 `SQLException` 封装为 `DataAccessException`，这个异常是一个 `RuntimeException`，并且让我们能区分SQL异常的原因，例如，`DuplicateKeyException` 表示违反了一个唯一约束；
- 能方便地集成Hibernate、JPA和MyBatis这些数据库访问框架。

本章我们将详细讲解在Spring中访问数据库的最佳实践。

使用JDBC

我们在前面介绍[JDBC编程](#)时已经讲过，Java程序使用JDBC接口访问关系数据库的时候，需要以下几步：

- 创建全局 `DataSource` 实例，表示数据库连接池；
- 在需要读写数据库的方法内部，按如下步骤访问数据库：
 - 从全局 `DataSource` 实例获取 `Connection` 实例；
 - 通过 `Connection` 实例创建 `PreparedStatement` 实例；
 - 执行SQL语句，如果是查询，则通过 `ResultSet` 读取结果集，如果是修改，则获得 `int` 结果。

正确编写JDBC代码的关键是使用 `try ... finally` 释放资源，涉及到事务的代码需要正确提交或回滚事务。

在Spring使用JDBC，首先我们通过IoC容器创建并管理一个 `DataSource` 实例，然后，Spring提供了一个 `JdbcTemplate`，可以方便地让我们操作JDBC，因此，通常情况下，我们会实例化一个 `JdbcTemplate`。顾名思义，这个类主要使用了[Template模式](#)。

编写示例代码或者测试代码时，我们强烈推荐使用[HSQLDB](#)这个数据库，它是一个用Java编写的关系数据库，可以以内存模式或者文件模式运行，本身只有一个jar包，非常适合演示代码或者测试代码。

我们以实际工程为例，先创建Maven工程 `spring-data-jdbc`，然后引入以下依赖：

```
<dependencies>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>5.2.0.RELEASE</version>
  </dependency>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-jdbc</artifactId>
    <version>5.2.0.RELEASE</version>
  </dependency>
  <dependency>
    <groupId>javax.annotation</groupId>
    <artifactId>javax.annotation-api</artifactId>
    <version>1.3.2</version>
  </dependency>
  <dependency>
    <groupId>com.zaxxer</groupId>
    <artifactId>HikariCP</artifactId>
    <version>3.4.2</version>
  </dependency>
  <dependency>
    <groupId>org.hsqldb</groupId>
    <artifactId>hsqldb</artifactId>
    <version>2.5.0</version>
  </dependency>
</dependencies>
```

在AppConfig中，我们需要创建以下几个必须的Bean：

```
@Configuration
@ComponentScan
@PropertySource("jdbc.properties")
public class AppConfig {

    @Value("${jdbc.url}")
    String jdbcUrl;

    @Value("${jdbc.username}")
    String jdbcUsername;

    @Value("${jdbc.password}")
    String jdbcPassword;

    @Bean
```

```

DataSource createDataSource() {
    HikariConfig config = new HikariConfig();
    config.setJdbcUrl(jdbcUrl);
    config.setUsername(jdbcUsername);
    config.setPassword(jdbcPassword);
    config.addDataSourceProperty("autoCommit", "true");
    config.addDataSourceProperty("connectionTimeout", "5");
    config.addDataSourceProperty("idleTimeout", "60");
    return new HikariDataSource(config);
}

@Bean
JdbcTemplate createJdbcTemplate(@Autowired DataSource dataSource) {
    return new JdbcTemplate(dataSource);
}
}

```

在上述配置中：

1. 通过 `@PropertySource("jdbc.properties")` 读取数据库配置文件；
2. 通过 `@value("${jdbc.url}")` 注入配置文件的相关配置；
3. 创建一个 `DataSource` 实例，它的实际类型是 `HikariDataSource`，创建时需要用到注入的配置；
4. 创建一个 `JdbcTemplate` 实例，它需要注入 `DataSource`，这是通过方法参数完成注入的。

最后，针对HSQLDB写一个配置文件 `jdbc.properties`：

```

# 数据库文件名为testdb:
jdbc.url=jdbc:hsqldb:file:testdb

# Hsqldb默认的用户名是sa，口令是空字符串:
jdbc.username=sa
jdbc.password=

```

可以通过HSQLDB自带的工具来初始化数据库表，这里我们写一个Bean，在Spring容器启动时自动创建一个 `users` 表：

```

@Component
public class DatabaseInitializer {
    @Autowired
    JdbcTemplate jdbcTemplate;

    @PostConstruct
    public void init() {
        jdbcTemplate.update("CREATE TABLE IF NOT EXISTS users (" //
            + "id BIGINT IDENTITY NOT NULL PRIMARY KEY, " //
            + "email VARCHAR(100) NOT NULL, " //
            + "password VARCHAR(100) NOT NULL, " //
            + "name VARCHAR(100) NOT NULL, " //
            + "UNIQUE (email))");
    }
}

```

现在，所有准备工作都已完毕。我们只需要在需要访问数据库的Bean中，注入 `JdbcTemplate` 即可：

```

@Component
public class UserService {
    @Autowired
    JdbcTemplate jdbcTemplate;
    ...
}

```

JdbcTemplate用法

Spring提供的 `JdbcTemplate` 采用Template模式，提供了一系列以回调为特点的工具方法，目的是避免繁琐的 `try...catch` 语句。

我们以具体的示例来说明JdbcTemplate的用法。

首先我们看 `execute(ConnectionCallback<T> action)` 方法，它提供了Jdbc的 `Connection` 供我们使用：

```

public User getUserById(long id) {
    // 注意传入的是ConnectionCallback:
    return jdbcTemplate.execute((Connection conn) -> {
        // 可以直接使用conn实例，不要释放它，回调结束后JdbcTemplate自动释放：
        // 在内部手动创建的PreparedStatement、ResultSet必须用try(...)释放：
        try (var ps = conn.prepareStatement("SELECT * FROM users WHERE id = ?"))
        {
            ps.setObject(1, id);
            try (var rs = ps.executeQuery()) {
                if (rs.next()) {
                    return new User( // new User object:
                        rs.getLong("id"), // id
                        rs.getString("email"), // email
                        rs.getString("password"), // password
                        rs.getString("name")); // name
                }
                throw new RuntimeException("user not found by id.");
            }
        }
    });
}

```

也就是说，上述回调方法允许获取Connection，然后做任何基于Connection的操作。

我们再看 `execute(String sql, PreparedStatementCallback<T> action)` 的用法：

```

public User getUserByName(String name) {
    // 需要传入SQL语句，以及PreparedStatementCallback:
    return jdbcTemplate.execute("SELECT * FROM users WHERE name = ?",
        (PreparedStatement ps) -> {
            // PreparedStatement实例已经由JdbcTemplate创建，并在回调后自动释放：
            ps.setObject(1, name);
            try (var rs = ps.executeQuery()) {
                if (rs.next()) {
                    return new User( // new User object:
                        rs.getLong("id"), // id
                        rs.getString("email"), // email
                        rs.getString("password"), // password
                        rs.getString("name")); // name
                }
            }
        });
}

```

```

        }
        throw new RuntimeException("user not found by id.");
    }
}
});
}

```

最后，我们看 `T queryForObject(String sql, Object[] args, RowMapper<T> rowMapper)` 方法：

```

public User getUserByEmail(String email) {
    // 传入SQL，参数和RowMapper实例：
    return jdbcTemplate.queryForObject("SELECT * FROM users WHERE email = ?", new
    Object[] { email },
        (ResultSet rs, int rowNum) -> {
            // 将ResultSet的当前行映射为一个JavaBean：
            return new User( // new User object:
                rs.getLong("id"), // id
                rs.getString("email"), // email
                rs.getString("password"), // password
                rs.getString("name")); // name
        });
}

```

在 `queryForObject()` 方法中，传入SQL以及SQL参数后，`JdbcTemplate` 会自动创建 `PreparedStatement`，自动执行查询并返回 `ResultSet`，我们提供的 `RowMapper` 需要做的事情就是把 `ResultSet` 的当前行映射成一个JavaBean并返回。整个过程中，使用 `Connection`、`PreparedStatement` 和 `ResultSet` 都不需要我们手动管理。

`RowMapper` 不一定返回JavaBean，实际上它可以返回任何Java对象。例如，使用 `SELECT COUNT(*)` 查询时，可以返回 `Long`：

```

public long getUsers() {
    return jdbcTemplate.queryForObject("SELECT COUNT(*) FROM users", null,
    (ResultSet rs, int rowNum) -> {
        // SELECT COUNT(*)查询只有一列，取第一列数据：
        return rs.getLong(1);
    });
}

```

如果我们期望返回多行记录，而不是一行，可以用 `query()` 方法：

```

public List<User> getUsers(int pageIndex) {
    int limit = 100;
    int offset = limit * (pageIndex - 1);
    return jdbcTemplate.query("SELECT * FROM users LIMIT ? OFFSET ?", new
    Object[] { limit, offset },
        new BeanPropertyRowMapper<>(User.class));
}

```

上述 `query()` 方法传入的参数仍然是SQL、SQL参数以及 `RowMapper` 实例。这里我们直接使用Spring提供的 `BeanPropertyRowMapper`。如果数据库表的结构恰好和JavaBean的属性名称一致，那么 `BeanPropertyRowMapper` 就可以直接把一行记录按列名转换为JavaBean。

如果我们执行的不是查询，而是插入、更新和删除操作，那么需要使用 `update()` 方法：


```

public void updateUser(User user) {
    // 传入SQL, SQL参数, 返回更新的行数:
    if (1 != jdbcTemplate.update("UPDATE user SET name = ? WHERE id=?",
        user.getName(), user.getId())) {
        throw new RuntimeException("User not found by id");
    }
}

```

只有一种 `INSERT` 操作比较特殊, 那就是如果某一列是自增列 (例如自增主键), 通常, 我们需要获取插入后的自增值。 `JdbcTemplate` 提供了一个 `KeyHolder` 来简化这一操作:

```

public User register(String email, String password, String name) {
    // 创建一个KeyHolder:
    KeyHolder holder = new GeneratedKeyHolder();
    if (1 != jdbcTemplate.update(
        // 参数1:PreparedStatementCreator
        (conn) -> {
            // 创建PreparedStatement时, 必须指定RETURN_GENERATED_KEYS:
            var ps = conn.prepareStatement("INSERT INTO
users(email,password,name) VALUES(?,?,?)",
                Statement.RETURN_GENERATED_KEYS);
            ps.setObject(1, email);
            ps.setObject(2, password);
            ps.setObject(3, name);
            return ps;
        },
        // 参数2:KeyHolder
        holder)
    ) {
        throw new RuntimeException("Insert failed.");
    }
    // 从KeyHolder中获取返回的自增值:
    return new User(holder.getKey().longValue(), email, password, name);
}

```

`JdbcTemplate` 还有许多重载方法, 这里我们不——介绍。需要强调的是, `JdbcTemplate` 只是对JDBC操作的一个简单封装, 它的目的是尽量减少手动编写 `try(resource) {...}` 的代码, 对于查询, 主要通过 `RowMapper` 实现了JDBC结果集到Java对象的转换。

我们总结一下 `JdbcTemplate` 的用法, 那就是:

- 针对简单查询, 优选 `query()` 和 `queryForObject()`, 因为只需提供SQL语句、参数和 `RowMapper`;
- 针对更新操作, 优选 `update()`, 因为只需提供SQL语句和参数;
- 任何复杂的操作, 最终也可以通过 `execute(ConnectionCallback)` 实现, 因为拿到 `Connection` 就可以做任何JDBC操作。

实际上我们使用最多的仍然是各种查询。如果在设计表结构的时候, 能够和JavaBean的属性——对应, 那么直接使用 `BeanPropertyRowMapper` 就很方便。如果表结构和JavaBean不一致怎么办? 那就需要稍微改写一下查询, 使结果集的结构和JavaBean保持一致。

例如, 表的列名是 `office_address`, 而JavaBean属性是 `workAddress`, 就需要指定别名, 改写查询如下:

```

SELECT id, email, office_address AS workAddress, name FROM users WHERE email = ?

```

练习

从  **gitee** 下载练习: [使用JdbcTemplate](#) (推荐使用[IDE练习插件](#)快速下载)

小结

Spring提供了 `JdbcTemplate` 来简化JDBC操作;

使用 `JdbcTemplate` 时, 根据需要优先选择高级方法;

任何JDBC操作都可以使用保底的 `execute(ConnectionCallback)` 方法。

使用声明式事务

使用Spring操作JDBC虽然方便, 但是我们在前面讨论JDBC的时候, 讲到过[JDBC事务](#), 如果要在Spring中操作事务, 没必要手写JDBC事务, 可以使用Spring提供的高级接口来操作事务。

Spring提供了一个 `PlatformTransactionManager` 来表示事务管理器, 所有的事务都由它负责管理。而事务由 `TransactionStatus` 表示。如果手写事务代码, 使用 `try...catch` 如下:

```
TransactionStatus tx = null;
try {
    // 开启事务:
    tx = txManager.getTransaction(new DefaultTransactionDefinition());
    // 相关JDBC操作:
    jdbcTemplate.update("...");
    jdbcTemplate.update("...");
    // 提交事务:
    txManager.commit(tx);
} catch (RuntimeException e) {
    // 回滚事务:
    txManager.rollback(tx);
    throw e;
}
```

Spring为啥要抽象出 `PlatformTransactionManager` 和 `TransactionStatus`? 原因是JavaEE除了提供JDBC事务外, 它还支持分布式事务JTA (Java Transaction API)。分布式事务是指多个数据源 (比如多个数据库, 多个消息系统) 要在分布式环境下实现事务的时候, 应该怎么实现。分布式事务实现起来非常复杂, 简单地说就是通过一个分布式事务管理器实现两阶段提交, 但本身数据库事务就不快, 基于数据库事务实现的分布式事务就慢得难以忍受, 所以使用率不高。

Spring为了同时支持JDBC和JTA两种事务模型, 就抽象出 `PlatformTransactionManager`。因为我们的代码只需要JDBC事务, 因此, 在 `AppConfig` 中, 需要再定义一个 `PlatformTransactionManager` 对应的Bean, 它的实际类型是 `DataSourceTransactionManager`:

```
@Configuration
@ComponentScan
@PropertySource("jdbc.properties")
public class AppConfig {
    ...
    @Bean
    PlatformTransactionManager createTxManager(@Autowired DataSource dataSource)
    {
        return new DataSourceTransactionManager(dataSource);
    }
}
```

使用编程的方式使用Spring事务仍然比较繁琐，更好的方式是通过声明式事务来实现。使用声明式事务非常简单，除了在AppConfig中追加一个上述定义的PlatformTransactionManager外，再加一个@EnableTransactionManagement就可以启用声明式事务：

```
@Configuration
@ComponentScan
@EnableTransactionManagement // 启用声明式
@PropertySource("jdbc.properties")
public class AppConfig {
    ...
}
```

然后，对需要事务支持的方法，加一个@Transactional注解：

```
@Component
public class UserService {
    // 此public方法自动具有事务支持：
    @Transactional
    public User register(String email, String password, String name) {
        ...
    }
}
```

或者更简单一点，直接在Bean的class处加上，表示所有public方法都具有事务支持：

```
@Component
@Transactional
public class UserService {
    ...
}
```

Spring对一个声明式事务的方法，如何开启事务支持？原理仍然是AOP代理，即通过自动创建Bean的Proxy实现：

```
public class UserService$$EnhancerBySpringCGLIB extends UserService {
    UserService target = ...
    PlatformTransactionManager txManager = ...

    public User register(String email, String password, String name) {
        TransactionStatus tx = null;
        try {
            tx = txManager.getTransaction(new DefaultTransactionDefinition());
            target.register(email, password, name);
            txManager.commit(tx);
        } catch (RuntimeException e) {
            txManager.rollback(tx);
            throw e;
        }
    }
    ...
}
```

注意：声明了@EnableTransactionManagement后，不必额外添加@EnableAspectJAutoProxy。

回滚事务

默认情况下，如果发生了 `RuntimeException`，Spring 的声明式事务将自动回滚。在一个事务方法中，如果程序判断需要回滚事务，只需抛出 `RuntimeException`，例如：

```
@Transactional
public buyProducts(long productId, int num) {
    ...
    if (store < num) {
        // 库存不够，购买失败：
        throw new IllegalArgumentException("No enough products");
    }
    ...
}
```

如果要针对 Checked Exception 回滚事务，需要在 `@Transactional` 注解中写出来：

```
@Transactional(rollbackFor = {RuntimeException.class, IOException.class})
public buyProducts(long productId, int num) throws IOException {
    ...
}
```

上述代码表示在抛出 `RuntimeException` 或 `IOException` 时，事务将回滚。

为了简化代码，我们强烈建议业务异常体系从 `RuntimeException` 派生，这样就不必声明任何特殊异常即可让 Spring 的声明式事务正常工作：

```
public class BusinessException extends RuntimeException {
    ...
}

public class LoginException extends BusinessException {
    ...
}

public class PaymentException extends BusinessException {
    ...
}
```

事务边界

在使用事务的时候，明确事务边界非常重要。对于声明式事务，例如，下面的 `register()` 方法：

```
@Component
public class UserService {
    @Transactional
    public User register(String email, String password, String name) { // 事务开始
        ...
    } // 事务结束
}
```

它的事务边界就是 `register()` 方法开始和结束。

类似的，一个负责给用户增加积分的 `addBonus()` 方法：

```

@Component
public class BonusService {
    @Transactional
    public void addBonus(long userId, int bonus) { // 事务开始
        ...
    } // 事务结束
}

```

它的事务边界就是 `addBonus()` 方法开始和结束。

在现实世界中，问题总是要复杂一点点。用户注册后，能自动获得100积分，因此，实际代码如下：

```

@Component
public class UserService {
    @Autowired
    BonusService bonusService;

    @Transactional
    public User register(String email, String password, String name) {
        // 插入用户记录：
        User user = jdbcTemplate.insert("...");
        // 增加100积分：
        bonusService.addBonus(user.id, 100);
    }
}

```

现在问题来了：调用方（比如 `RegisterController`）调用 `UserService.register()` 这个事务方法，它在内部又调用了 `BonusService.addBonus()` 这个事务方法，一共有几个事务？如果 `addBonus()` 抛出了异常需要回滚事务，`register()` 方法的事务是否也要回滚？

问题的复杂度是不是一下子提高了10倍？

事务传播

要解决上面的问题，我们首先要定义事务的传播模型。

假设用户注册的入口是 `RegisterController`，它本身没有事务，仅仅是调用 `UserService.register()` 这个事务方法：

```

@Controller
public class RegisterController {
    @Autowired
    UserService userService;

    @PostMapping("/register")
    public ModelAndView doRegister(HttpServletRequest req) {
        String email = req.getParameter("email");
        String password = req.getParameter("password");
        String name = req.getParameter("name");
        User user = userService.register(email, password, name);
        return ...
    }
}

```

因此，`UserService.register()` 这个事务方法的起始和结束，就是事务的范围。

我们需要关心的是，在 `UserService.register()` 这个事务方法内，调用 `BonusService.addBonus()`，我们期待的事务行为是什么：

```
@Transactional
public User register(String email, String password, String name) {
    // 事务已开启：
    User user = jdbcTemplate.insert("...");
    // ???：
    bonusService.addBonus(user.id, 100);
} // 事务结束
```

对于大多数业务来说，我们期待 `BonusService.addBonus()` 的调用，和 `UserService.register()` 应当融合在一起，它的行为应该如下：

`UserService.register()` 已经开启了一个事务，那么在内部调用 `BonusService.addBonus()` 时，`BonusService.addBonus()` 方法就没必要再开启一个新事务，直接加入到 `BonusService.register()` 的事务里就好了。

其实就相当于：

1. `UserService.register()` 先执行了一条INSERT语句： `INSERT INTO users ...`
2. `BonusService.addBonus()` 再执行一条INSERT语句： `INSERT INTO bonus ...`

因此，Spring的声明式事务为事务传播定义了几个级别，默认传播级别就是REQUIRED，它的意思是，如果当前没有事务，就创建一个新事务，如果当前有事务，就加入到当前事务中执行。

我们观察 `UserService.register()` 方法，它在 `RegisterController` 中执行，因为 `RegisterController` 没有事务，因此，`UserService.register()` 方法会自动创建一个新事务。

在 `UserService.register()` 方法内部，调用 `BonusService.addBonus()` 方法时，因为 `BonusService.addBonus()` 检测到当前已经有事务了，因此，它会加入到当前事务中执行。

因此，整个业务流程的事务边界就清晰了：它只有一个事务，并且范围就是 `UserService.register()` 方法。

有的童鞋会问：把 `BonusService.addBonus()` 方法的 `@Transactional` 去掉，变成一个普通方法，那不就规避了复杂的传播模型吗？

去掉 `BonusService.addBonus()` 方法的 `@Transactional`，会引来另一个问题，即其他地方如果调用 `BonusService.addBonus()` 方法，那就没法保证事务了。例如，规定用户登录时积分+5：

```
@Controller
public class LoginController {
    @Autowired
    BonusService bonusService;

    @PostMapping("/login")
    public ModelAndView doLogin(HttpServletRequest req) {
        User user = ...
        bonusService.addBonus(user.id, 5);
    }
}
```

可见，`BonusService.addBonus()` 方法必须要有 `@Transactional`，否则，登录后积分就无法添加了。

默认的事务传播级别是 REQUIRED，它满足绝大部分的需求。还有一些其他的传播级别：

SUPPORTS：表示如果有事务，就加入到当前事务，如果没有，那也不开启事务执行。这种传播级别可用于查询方法，因为SELECT语句既可以在事务内执行，也可以不需要事务；

MANDATORY：表示必须要存在当前事务并加入执行，否则将抛出异常。这种传播级别可用于核心更新逻辑，比如用户余额变更，它总是被其他事务方法调用，不能直接由非事务方法调用；

REQUIRES_NEW：表示不管当前有没有事务，都必须开启一个新的事务执行。如果当前已经有事务，那么当前事务会挂起，等新事务完成后，再恢复执行；

NOT_SUPPORTED：表示不支持事务，如果当前有事务，那么当前事务会挂起，等这个方法执行完成后，再恢复执行；

NEVER：和 **NOT_SUPPORTED** 相比，它不但不支持事务，而且在监测到当前有事务时，会抛出异常拒绝执行；

NESTED：表示如果当前有事务，则开启一个嵌套级别事务，如果当前没有事务，则开启一个新事务。

上面这么多事务的传播级别，其实默认的 **REQUIRED** 已经满足绝大部分需求，**SUPPORTS** 和 **REQUIRES_NEW** 在少数情况下会用到，其他基本不会用到，因为把事务搞得越复杂，不仅逻辑跟着复杂，而且速度也会越慢。

定义事务的传播级别也是写在 `@Transactional` 注解里的：

```
@Transactional(propagation = Propagation.REQUIRES_NEW)
public Product createProduct() {
    ...
}
```

现在只剩最后一个问题了：Spring是如何传播事务的？

我们在[JDBC中使用事务](#)的时候，是这么个写法：

```
Connection conn = openConnection();
try {
    // 关闭自动提交：
    conn.setAutoCommit(false);
    // 执行多条SQL语句：
    insert(); update(); delete();
    // 提交事务：
    conn.commit();
} catch (SQLException e) {
    // 回滚事务：
    conn.rollback();
} finally {
    conn.setAutoCommit(true);
    conn.close();
}
```

Spring使用声明式事务，最终也是通过执行JDBC事务来实现功能的，那么，一个事务方法，如何获知当前是否存在事务？

答案是[使用ThreadLocal](#)。Spring总是把JDBC相关的 `Connection` 和 `TransactionStatus` 实例绑定到 `ThreadLocal`。如果一个事务方法从 `ThreadLocal` 未取到事务，那么它会打开一个新的JDBC连接，同时开启一个新的事务，否则，它就直接使用从 `ThreadLocal` 获取的JDBC连接以及 `TransactionStatus`。

因此，事务能正确传播的前提是，方法调用是在一个线程内才行。如果像下面这样写：

```

@Transactional
public User register(String email, String password, String name) { // BEGIN TX-A
    User user = jdbcTemplate.insert("...");
    new Thread(() -> {
        // BEGIN TX-B:
        bonusService.addBonus(user.id, 100);
        // END TX-B
    }).start();
} // END TX-A

```

在另一个线程中调用 `BonusService.addBonus()`，它根本获取不到当前事务，因此，`UserService.register()` 和 `BonusService.addBonus()` 两个方法，将分别开启两个完全独立的事务。

换句话说，事务只能在当前线程传播，无法跨线程传播。

那如果我们想实现跨线程传播事务呢？原理很简单，就是要想办法把当前线程绑定到 `ThreadLocal` 的 `Connection` 和 `TransactionStatus` 实例传递给新线程，但实现起来非常复杂，根据异常回滚更加复杂，不推荐自己去实现。

练习

从  **gitee** 下载练习：[使用声明式事务](#)（推荐使用[IDE练习插件](#)快速下载）

小结

Spring提供的声明式事务极大地方便了在数据库中使用事务，正确使用声明式事务的关键在于确定好事务边界，理解事务传播级别。

使用DAO

在传统的多层应用程序中，通常是Web层调用业务层，业务层调用数据访问层。业务层负责处理各种业务逻辑，而数据访问层只负责对数据进行增删改查。因此，实现数据访问层就是用 `JdbcTemplate` 实现对数据库的操作。

编写数据访问层的时候，可以使用DAO模式。DAO即Data Access Object的缩写，它没有什么神秘之处，实现起来基本如下：

```

public class UserDao {

    @Autowired
    JdbcTemplate jdbcTemplate;

    User getById(long id) {
        ...
    }

    List<User> getUsers(int page) {
        ...
    }

    User createUser(User user) {
        ...
    }

    User updateUser(User user) {

```



```

        ...
    }

    void deleteUser(User user) {
        ...
    }
}

```

Spring提供了一个 `JdbcDaoSupport` 类，用于简化DAO的实现。这个 `JdbcDaoSupport` 没什么复杂的，核心代码就是持有一个 `JdbcTemplate`：

```

public abstract class JdbcDaoSupport extends DaoSupport {

    private JdbcTemplate jdbcTemplate;

    public final void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
        initTemplateConfig();
    }

    public final JdbcTemplate getJdbcTemplate() {
        return this.jdbcTemplate;
    }

    ...
}

```

它的意图是子类直接从 `JdbcDaoSupport` 继承后，可以随时调用 `getJdbcTemplate()` 获得 `JdbcTemplate` 的实例。那么问题来了：因为 `JdbcDaoSupport` 的 `jdbcTemplate` 字段没有标记 `@Autowired`，所以，子类想要注入 `JdbcTemplate`，还得自己想个办法：

```

@Component
@Transactional
public class UserDao extends JdbcDaoSupport {
    @Autowired
    JdbcTemplate jdbcTemplate;

    @PostConstruct
    public void init() {
        super.setJdbcTemplate(jdbcTemplate);
    }
}

```

有的童鞋可能看出来了：既然 `UserDao` 都已经注入了 `JdbcTemplate`，那再把它放到父类里，通过 `getJdbcTemplate()` 访问岂不是多此一举？

如果使用传统的XML配置，并不需要编写 `@Autowired JdbcTemplate jdbcTemplate`，但是考虑到现在基本上是使用注解的方式，我们可以编写一个 `AbstractDao`，专门负责注入 `JdbcTemplate`：

```

public abstract class AbstractDao extends JdbcDaoSupport {
    @Autowired
    private JdbcTemplate jdbcTemplate;

    @PostConstruct
    public void init() {
        super.setJdbcTemplate(jdbcTemplate);
    }
}

```

这样，子类的代码就非常干净，可以直接调用 `getJdbcTemplate()`：

```

@Component
@Transactional
public class UserDao extends AbstractDao {
    public User getById(long id) {
        return getJdbcTemplate().queryForObject(
            "SELECT * FROM users WHERE id = ?",
            new BeanPropertyRowMapper<>(User.class),
            id
        );
    }
    ...
}

```

倘若肯再多写一点样板代码，就可以把 `AbstractDao` 改成泛型，并实现 `getById()`，`getAll()`，`deleteById()` 这样的通用方法：

```

public abstract class AbstractDao<T> extends JdbcDaoSupport {
    private String table;
    private Class<T> entityClass;
    private RowMapper<T> rowMapper;

    public AbstractDao() {
        // 获取当前类型的泛型类型：
        this.entityClass = getParameterizedType();
        this.table = this.entityClass.getSimpleName().toLowerCase() + "s";
        this.rowMapper = new BeanPropertyRowMapper<>(entityClass);
    }

    public T getById(long id) {
        return getJdbcTemplate().queryForObject("SELECT * FROM " + table + "
WHERE id = ?", this.rowMapper, id);
    }

    public List<T> getAll(int pageIndex) {
        int limit = 100;
        int offset = limit * (pageIndex - 1);
        return getJdbcTemplate().query("SELECT * FROM " + table + " LIMIT ?
OFFSET ?",
            new Object[] { limit, offset },
            this.rowMapper);
    }

    public void deleteById(long id) {
        getJdbcTemplate().update("DELETE FROM " + table + " WHERE id = ?", id);
    }
}

```

```
}  
...  
}
```

这样，每个子类就自动获得了这些通用方法：

```
@Component  
@Transactional  
public class UserDao extends AbstractDao<User> {  
    // 已经有了：  
    // User getById(long)  
    // List<User> getAll(int)  
    // void deleteById(long)  
}  
  
@Component  
@Transactional  
public class BookDao extends AbstractDao<Book> {  
    // 已经有了：  
    // Book getById(long)  
    // List<Book> getAll(int)  
    // void deleteById(long)  
}
```

可见，DAO模式就是一个简单的数据访问模式，是否使用DAO，根据实际情况决定，因为很多时候，直接在Service层操作数据库也是完全没有问题的。

练习

从  **gitee** 下载练习：[使用DAO模式](#)（推荐使用[IDE练习插件](#)快速下载）

小结

Spring提供了 `JdbcDaoSupport` 来便于我们实现DAO模式；

可以基于泛型实现更通用、更简洁的DAO模式。

集成Hibernate

使用 `JdbcTemplate` 的时候，我们用得最多的方法就是 `List<T> query(String sql, Object[] args, RowMapper rowMapper)`。这个 `RowMapper` 的作用就是把 `ResultSet` 的一行记录映射为Java Bean。

这种把关系数据库的表记录映射为Java对象的过程就是ORM：Object-Relational Mapping。ORM既可以把记录转换成Java对象，也可以把Java对象转换为行记录。

使用 `JdbcTemplate` 配合 `RowMapper` 可以看作是最原始的ORM。如果要想实现更自动化的ORM，可以选择成熟的ORM框架，例如[Hibernate](#)。

我们来看看如何在Spring中集成Hibernate。

Hibernate作为ORM框架，它可以替代 `JdbcTemplate`，但Hibernate仍然需要JDBC驱动，所以，我们需要引入JDBC驱动、连接池，以及Hibernate本身。在Maven中，我们加入以下依赖项：

```
<!-- JDBC驱动，这里使用HSQLDB -->  
<dependency>  
    <groupId>org.hsqldb</groupId>
```

```

        <artifactId>hsqldb</artifactId>
        <version>2.5.0</version>
    </dependency>

    <!-- JDBC连接池 -->
    <dependency>
        <groupId>com.zaxxer</groupId>
        <artifactId>HikariCP</artifactId>
        <version>3.4.2</version>
    </dependency>

    <!-- Hibernate -->
    <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-core</artifactId>
        <version>5.4.2.Final</version>
    </dependency>

    <!-- Spring Context和Spring ORM -->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>5.2.0.RELEASE</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-orm</artifactId>
        <version>5.2.0.RELEASE</version>
    </dependency>

```

在AppConfig中，我们仍然需要创建DataSource、引入JDBC配置文件，以及启用声明式事务：

```

@Configuration
@ComponentScan
@EnableTransactionManagement
@PropertySource("jdbc.properties")
public class AppConfig {
    @Bean
    DataSource createDataSource() {
        ...
    }
}

```

为了启用Hibernate，我们需要创建一个LocalSessionFactoryBean：

```

public class AppConfig {
    @Bean
    LocalSessionFactoryBean createSessionFactory(@Autowired DataSource
dataSource) {
        var props = new Properties();
        props.setProperty("hibernate.hbm2ddl.auto", "update"); // 生产环境不要使用
        props.setProperty("hibernate.dialect",
"org.hibernate.dialect.HSQLDialect");
        props.setProperty("hibernate.show_sql", "true");
        var sessionFactoryBean = new LocalSessionFactoryBean();
        sessionFactoryBean.setDataSource(dataSource);
    }
}

```

```

// 扫描指定的package获取所有entity class:
sessionFactoryBean.setPackagesToScan("com.itranswarp.learnjava.entity");
sessionFactoryBean.setHibernateProperties(props);
return sessionFactoryBean;
}
}

```

注意我们在[定制Bean](#)中讲到过 `FactoryBean`，`LocalSessionFactoryBean` 是一个 `FactoryBean`，它会自动创建一个 `SessionFactory`，在Hibernate中，`Session` 是封装了一个JDBC `Connection` 的实例，而 `SessionFactory` 是封装了JDBC `DataSource` 的实例，即 `SessionFactory` 持有连接池，每次需要操作数据库的时候，`SessionFactory` 创建一个新的 `Session`，相当于从连接池获取到一个新的 `Connection`。`SessionFactory` 就是Hibernate提供的最核心的一个对象，但 `LocalSessionFactoryBean` 是Spring提供的为了让我们方便创建 `SessionFactory` 的类。

注意到上面创建 `LocalSessionFactoryBean` 的代码，首先用 `Properties` 持有Hibernate初始化 `SessionFactory` 时用到的所有设置，常用的设置请参考[Hibernate文档](#)，这里我们只定义了3个设置：

- `hibernate.hbm2ddl.auto=update`：表示自动创建数据库的表结构，注意不要在生产环境中启用；
- `hibernate.dialect=org.hibernate.dialect.HSQLDialect`：指示Hibernate使用的数据库是HSQLDB。Hibernate使用一种HQL的查询语句，它和SQL类似，但真正在“翻译”成SQL时，会根据设定的数据库“方言”来生成针对数据库优化的SQL；
- `hibernate.show_sql=true`：让Hibernate打印执行的SQL，这对于调试非常有用，我们可以方便地看到Hibernate生成的SQL语句是否符合我们的预期。

除了设置 `DataSource` 和 `Properties` 之外，注意到 `setPackagesToScan()` 我们传入了一个 `package` 名称，它指示Hibernate扫描这个包下面的所有Java类，自动找出能映射为数据库表记录的JavaBean。后面我们会仔细讨论如何编写符合Hibernate要求的JavaBean。

紧接着，我们还需要创建 `HibernateTemplate` 以及 `HibernateTransactionManager`：

```

public class AppConfig {
    @Bean
    HibernateTemplate createHibernateTemplate(@Autowired SessionFactory
    sessionFactory) {
        return new HibernateTemplate(sessionFactory);
    }

    @Bean
    PlatformTransactionManager createTxManager(@Autowired SessionFactory
    sessionFactory) {
        return new HibernateTransactionManager(sessionFactory);
    }
}

```

这两个Bean的创建都十分简单。`HibernateTransactionManager` 是配合Hibernate使用声明式事务所必须的，而 `HibernateTemplate` 则是Spring为了便于我们使用Hibernate提供的工具类，不是非用不可，但推荐使用以简化代码。

到此为止，所有的配置都定义完毕，我们来看看如何将数据库表结构映射为Java对象。

考察如下的数据库表：

```
CREATE TABLE user
  id BIGINT NOT NULL AUTO_INCREMENT,
  email VARCHAR(100) NOT NULL,
  password VARCHAR(100) NOT NULL,
  name VARCHAR(100) NOT NULL,
  createdAt BIGINT NOT NULL,
  PRIMARY KEY (`id`),
  UNIQUE KEY `email` (`email`)
);
```

其中，`id` 是自增主键，`email`、`password`、`name` 是 `VARCHAR` 类型，`email` 带唯一索引以确保唯一性，`createdAt` 存储整型类型的时间戳。用 `JavaBean` 表示如下：

```
public class User {
    private Long id;
    private String email;
    private String password;
    private String name;
    private Long createdAt;

    // getters and setters
    ...
}
```

这种映射关系十分易懂，但我们需要添加一些注解来告诉 `Hibernate` 如何把 `User` 类映射到表记录：

```
@Entity
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(nullable = false, updatable = false)
    public Long getId() { ... }

    @Column(nullable = false, unique = true, length = 100)
    public String getEmail() { ... }

    @Column(nullable = false, length = 100)
    public String getPassword() { ... }

    @Column(nullable = false, length = 100)
    public String getName() { ... }

    @Column(nullable = false, updatable = false)
    public Long getCreatedAt() { ... }
}
```

如果一个 `JavaBean` 被用于映射，我们就标记一个 `@Entity`。默认情况下，映射的表名是 `user`，如果实际的表名不同，例如实际表名是 `users`，可以追加一个 `@Table(name="users")` 表示：

```
@Entity
@Table(name="users")
public class User {
    ...
}
```

每个属性到数据库列的映射用 `@Column()` 标识, `nullable` 指示列是否允许为 `NULL`, `updatable` 指示该列是否允许被用在 `UPDATE` 语句, `length` 指示 `String` 类型的列的长度 (如果没有指定, 默认是 `255`)。

对于主键, 还需要用 `@Id` 标识, 自增主键再追加一个 `@GeneratedValue`, 以便Hibernate能读取到自增主键的值。

细心的童鞋可能还注意到, 主键 `id` 定义的类型不是 `long`, 而是 `Long`。这是因为Hibernate如果检测到主键为 `null`, 就不会在 `INSERT` 语句中指定主键的值, 而是返回由数据库生成的自增值, 否则, Hibernate认为我们的程序指定了主键的值, 会在 `INSERT` 语句中直接列出。 `long` 型字段总是具有默认值 `0`, 因此, 每次插入的主键值总是 `0`, 导致除第一次外后续插入都将失败。

`createdAt` 虽然是整型, 但我们并没有使用 `long`, 而是 `Long`, 这是因为使用基本类型会导致某种查询会添加意外的条件, 后面我们会详细讨论, 这里只需牢记, 作为映射使用的JavaBean, 所有属性都使用包装类型而不是基本类型。

使用Hibernate时, 不要使用基本类型的属性, 总是使用包装类型, 如 `Long` 或 `Integer`。

类似的, 我们再定义一个 `Book` 类:

```
@Entity
public class Book {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(nullable = false, updatable = false)
    public Long getId() { ... }

    @Column(nullable = false, length = 100)
    public String getTitle() { ... }

    @Column(nullable = false, updatable = false)
    public Long getCreatedAt() { ... }
}
```

如果仔细观察 `User` 和 `Book`, 会发现它们定义的 `id`、`createdAt` 属性是一样的, 这在数据库表结构的设计中很常见: 对于每个表, 通常会统一使用一种主键生成机制, 并添加 `createdAt` 表示创建时间, `updatedAt` 表示修改时间等通用字段。

不必在 `User` 和 `Book` 中重复定义这些通用字段, 我们可以把它们提到一个抽象类中:

```
@MappedSuperclass
public abstract class AbstractEntity {

    private Long id;
    private Long createdAt;

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(nullable = false, updatable = false)
    public Long getId() { ... }

    @Column(nullable = false, updatable = false)
    public Long getCreatedAt() { ... }

    @Transient
    public ZonedDateTime getCreatedDateTime() {
```

```

        return
Instant.ofEpochMilli(this.createdAt).atZone(ZoneId.systemDefault());
    }

    @PrePersist
    public void preInsert() {
        setCreatedAt(System.currentTimeMillis());
    }
}

```

对于 `AbstractEntity` 来说，我们要标注一个 `@MappedSuperclass` 表示它用于继承。此外，注意到我们定义了一个 `@Transient` 方法，它返回一个“虚拟”的属性。因为 `getCreatedDateTime()` 是计算得出的属性，而不是从数据库表读出的值，因此必须要标注 `@Transient`，否则Hibernate会尝试从数据库读取名为 `createdDateTime` 这个不存在的字段从而出错。

再注意到 `@PrePersist` 标识的方法，它表示在我们将一个JavaBean持久化到数据库之前（即执行INSERT语句），Hibernate会先执行该方法，这样我们就可以自动设置好 `createdAt` 属性。

有了 `AbstractEntity`，我们就可以大幅简化 `User` 和 `Book`：

```

@Entity
public class User extends AbstractEntity {

    @Column(nullable = false, unique = true, length = 100)
    public String getEmail() { ... }

    @Column(nullable = false, length = 100)
    public String getPassword() { ... }

    @Column(nullable = false, length = 100)
    public String getName() { ... }
}

```

注意到使用的所有注解均来自 `javax.persistence`，它是JPA规范的一部分。这里我们只介绍使用注解的方式配置Hibernate映射关系，不再介绍传统的比较繁琐的XML配置。通过Spring集成Hibernate时，也不再需要 `hibernate.cfg.xml` 配置文件，用一句话总结：

使用Spring集成Hibernate，配合JPA注解，无需任何额外的XML配置。

类似 `User`、`Book` 这样的用于ORM的Java Bean，我们通常称之为Entity Bean。

最后，我们来看看如果对 `user` 表进行增删改查。因为使用了Hibernate，因此，我们要做的，实际上是对 `user` 这个JavaBean进行“增删改查”。我们编写一个 `UserService`，注入 `HibernateTemplate` 以便简化代码：

```

@Component
@Transactional
public class UserService {
    @Autowired
    HibernateTemplate hibernateTemplate;
}

```


Insert操作

要持久化一个 `User` 实例，我们只需调用 `save()` 方法。以 `register()` 方法为例，代码如下：

```
public User register(String email, String password, String name) {
    // 创建一个User对象：
    User user = new User();
    // 设置好各个属性：
    user.setEmail(email);
    user.setPassword(password);
    user.setName(name);
    // 不要设置id，因为使用了自增主键
    // 保存到数据库：
    hibernateTemplate.save(user);
    // 现在已经自动获得了id：
    System.out.println(user.getId());
    return user;
}
```

Delete操作

删除一个 `User` 相当于从表中删除对应的记录。注意Hibernate总是用 `id` 来删除记录，因此，要正确设置 `User` 的 `id` 属性才能正常删除记录：

```
public boolean deleteUser(Long id) {
    User user = hibernateTemplate.get(User.class, id);
    if (user != null) {
        hibernateTemplate.delete(user);
        return true;
    }
    return false;
}
```

通过主键删除记录时，一个常见的用法是先根据主键加载该记录，再删除。`load()` 和 `get()` 都可以根据主键加载记录，它们的区别在于，当记录不存在时，`get()` 返回 `null`，而 `load()` 抛出异常。

Update操作

更新记录相当于先更新 `User` 的指定属性，然后调用 `update()` 方法：

```
public void updateUser(Long id, String name) {
    User user = hibernateTemplate.load(User.class, id);
    user.setName(name);
    hibernateTemplate.update(user);
}
```

前面我们在定义 `User` 时，对有的属性标注了 `@Column(updatable=false)`。Hibernate在更新记录时，它只会把 `@Column(updatable=true)` 的属性加入到 `UPDATE` 语句中，这样可以提供一层额外的安全性，即如果不小心修改了 `User` 的 `email`、`createdAt` 等属性，执行 `update()` 时并不会更新对应的数据库列。但也必须牢记：这个功能是Hibernate提供的，如果绕过Hibernate直接通过JDBC执行 `UPDATE` 语句仍然可以更新数据库的任意列的值。

最后，我们编写的大部分方法都是各种各样的查询。根据 `id` 查询我们可以直接调用 `load()` 或 `get()`，如果要使用条件查询，有3种方法。

假设我们想执行以下查询：

```
SELECT * FROM user WHERE email = ? AND password = ?
```

我们来看看可以使用什么查询。

使用Example查询

第一种方法是使用 `findByExample()`，给出一个 `User` 实例，Hibernate把该实例所有非 `null` 的属性拼成 `WHERE` 条件：

```
public User login(String email, String password) {
    User example = new User();
    example.setEmail(email);
    example.setPassword(password);
    List<User> list = hibernateTemplate.findByExample(example);
    return list.isEmpty() ? null : list.get(0);
}
```

因为 `example` 实例只有 `email` 和 `password` 两个属性为非 `null`，所以最终生成的 `WHERE` 语句就是 `WHERE email = ? AND password = ?`。

如果我们将 `User` 的 `createdAt` 的类型从 `Long` 改为 `long`，`findByExample()` 的查询将出问题，原因在于 `example` 实例的 `long` 类型字段有了默认值0，导致Hibernate最终生成的 `WHERE` 语句意外变成了 `WHERE email = ? AND password = ? AND createdAt = 0`。显然，额外的查询条件将导致错误的查询结果。

使用`findByExample()`时，注意基本类型字段总是会加入到WHERE条件！

使用Criteria查询

第二种查询方法是使用Criteria查询，可以实现如下：

```
public User login(String email, String password) {
    DetachedCriteria criteria = DetachedCriteria.forClass(User.class);
    criteria.add(Restrictions.eq("email", email))
        .add(Restrictions.eq("password", password));
    List<User> list = (List<User>) hibernateTemplate.findByCriteria(criteria);
    return list.isEmpty() ? null : list.get(0);
}
```

`DetachedCriteria` 使用链式语句来添加多个 `AND` 条件。和 `findByExample()` 相比，`findByCriteria()` 可以组装出更灵活的 `WHERE` 条件，例如：

```
SELECT * FROM user WHERE (email = ? OR name = ?) AND password = ?
```

上述查询没法用 `findByExample()` 实现，但用Criteria查询可以实现如下：

```
DetachedCriteria criteria = DetachedCriteria.forClass(User.class);
criteria.add(
    Restrictions.and(
        Restrictions.or(
            Restrictions.eq("email", email),
            Restrictions.eq("name", email)
        ),
        Restrictions.eq("password", password)
    )
);
```

只要组织好 `Restrictions` 的嵌套关系，Criteria查询可以实现任意复杂的查询。

使用HQL查询

最后一种常用的查询是直接编写Hibernate内置的HQL查询：

```
List<User> list = (List<User>) hibernateTemplate.find("FROM User WHERE email=?
AND password=?", email, password);
```

和SQL相比，HQL使用类名和属性名，由Hibernate自动转换为实际的表名和列名。详细的HQL语法可以参考[Hibernate文档](#)。

除了可以直接传入HQL字符串外，Hibernate还可以使用一种 `NamedQuery`，它给查询起个名字，然后保存在注解中。使用 `NamedQuery` 时，我们要先在 `User` 类标注：

```
@NamedQueries(
    @NamedQuery(
        // 查询名称：
        name = "login",
        // 查询语句：
        query = "SELECT u FROM User u WHERE u.email=?0 AND u.password=?1"
    )
)
@Entity
public class User extends AbstractEntity {
    ...
}
```

注意到引入的 `NamedQuery` 是 `javax.persistence.NamedQuery`，它和直接传入HQL有点不同的是，占位符使用 `?0`、`?1`，并且索引是从0开始的（真乱）。

使用 `NamedQuery` 只需要引入查询名和参数：

```
public User login(String email, String password) {
    List<User> list = (List<User>) hibernateTemplate.findByNameNamedQuery("login",
    email, password);
    return list.isEmpty() ? null : list.get(0);
}
```

直接写HQL和使用 `NamedQuery` 各有优劣。前者可以在代码中直观地看到查询语句，后者可以在 `User` 类统一管理所有相关查询。

使用Hibernate原生接口

如果要使用Hibernate原生接口，但不知道怎么写，可以参考 `HibernateTemplate` 的源码。使用Hibernate的原生接口实际上总是从 `SessionFactory` 出发，它通常用全局变量存储，在 `HibernateTemplate` 中以成员变量被注入。有了 `SessionFactory`，使用Hibernate用法如下：

```
void operation() {
    Session session = null;
    boolean isNew = false;
    // 获取当前Session或者打开新的Session:
    try {
        session = this.sessionFactory.getCurrentSession();
    } catch (HibernateException e) {
        session = this.sessionFactory.openSession();
        isNew = true;
    }
    // 操作Session:
    try {
        User user = session.load(User.class, 123L);
    }
    finally {
        // 关闭新打开的Session:
        if (isNew) {
            session.close();
        }
    }
}
```

练习

从  **gitee** 下载练习：[集成Hibernate](#)（推荐使用[IDE练习插件](#)快速下载）

小结

在Spring中集成Hibernate需要配置的Bean如下：

- `DataSource`;
- `LocalSessionFactory`;
- `HibernateTransactionManager`;
- `HibernateTemplate`（推荐）。

推荐使用Annotation配置所有的Entity Bean。

集成JPA

上一节我们讲了在Spring中集成Hibernate。Hibernate是第一个被广泛使用的ORM框架，但是很多小伙伴还听说过JPA：Java Persistence API，这又是啥？

在讨论JPA之前，我们要注意到JavaEE早在1999年就发布了，并且有Servlet、JMS等诸多标准。和其他平台不同，Java世界早期非常热衷于标准先行，各家跟进：大家先坐下来把接口定了，然后，各自回家干活去实现接口，这样，用户就可以在不同的厂家提供的产品进行选择，还可以随意切换，因为用户编写代码的时候只需要引用接口，并不需要引用具体的底层实现（想想JDBC）。

JPA就是JavaEE的一个ORM标准，它的实现其实和Hibernate没啥本质区别，但是用户如果使用JPA，那么引用的就是 `javax.persistence` 这个“标准”包，而不是 `org.hibernate` 这样的第三方包。因为JPA只是接口，所以，还需要选择一个实现产品，跟JDBC接口和MySQL驱动一个道理。

我们使用JPA时也完全可以选择Hibernate作为底层实现，但也可以选择其它的JPA提供方，比如[EclipseLink](#)。Spring内置了JPA的集成，并支持选择Hibernate或EclipseLink作为实现。这里我们仍然以主流的Hibernate作为JPA实现为例子，演示JPA的基本用法。

和使用Hibernate一样，我们只需要引入如下依赖：

- org.springframework:spring-context:5.2.0.RELEASE
- org.springframework:spring-orm:5.2.0.RELEASE
- javax.annotation:javax.annotation-api:1.3.2
- org.hibernate:hibernate-core:5.4.2.Final
- com.zaxxer:HikariCP:3.4.2
- org.hsqldb:hsqldb:2.5.0

然后，在 AppConfig 中启用声明式事务管理，创建 DataSource：

```
@Configuration
@ComponentScan
@EnableTransactionManagement
@PropertySource("jdbc.properties")
public class AppConfig {
    @Bean
    DataSource createDataSource() { ... }
}
```

使用Hibernate时，我们需要创建一个 `LocalSessionFactoryBean`，并让它再自动创建一个 `SessionFactory`。使用JPA也是类似的，我们需要创建一个 `LocalContainerEntityManagerFactoryBean`，并让它再自动创建一个 `EntityManagerFactory`：

```
@Bean
LocalContainerEntityManagerFactoryBean createEntityManagerFactory(@Autowired
DataSource dataSource) {
    var entityManagerFactoryBean = new LocalContainerEntityManagerFactoryBean();
    // 设置DataSource:
    entityManagerFactoryBean.setDataSource(dataSource);
    // 扫描指定的package获取所有entity class:

    entityManagerFactoryBean.setPackagesToScan("com.itranswarp.learnjava.entity");
    // 指定JPA的提供商是Hibernate:
    JpaVendorAdapter vendorAdapter = new HibernateJpaVendorAdapter();
    entityManagerFactoryBean.setJpaVendorAdapter(vendorAdapter);
    // 设定特定提供商自己的配置:
    var props = new Properties();
    props.setProperty("hibernate.hbm2ddl.auto", "update");
    props.setProperty("hibernate.dialect", "org.hibernate.dialect.HSQLDialect");
    props.setProperty("hibernate.show_sql", "true");
    entityManagerFactoryBean.setJpaProperties(props);
    return entityManagerFactoryBean;
}
```

观察上述代码，除了需要注入 `DataSource` 和设定自动扫描的 `package` 外，还需要指定JPA的提供商，这里使用Spring提供的一个 `HibernateJpaVendorAdapter`，最后，针对Hibernate自己需要的配置，以 `Properties` 的形式注入。

最后，我们还需要实例化一个 `JpaTransactionManager`，以实现声明式事务：

```
@Bean
PlatformTransactionManager createTxManager(@Autowired EntityManagerFactory
entityManagerFactory) {
    return new JpaTransactionManager(entityManagerFactory);
}
```

这样，我们就完成了JPA的全部初始化工作。有些童鞋可能从网上搜索得知JPA需要 `persistence.xml` 配置文件，以及复杂的 `orm.xml` 文件。这里我们负责地告诉大家，使用Spring+Hibernate作为JPA实现，无需任何配置文件。

所有Entity Bean的配置和上一节完全相同，全部采用Annotation标注。我们现在只需关心具体的业务类如何通过JPA接口操作数据库。

还是以 `UserService` 为例，除了标注 `@Component` 和 `@Transactional` 外，我们需要注入一个 `EntityManager`，但是不要使用 `Autowired`，而是 `@PersistenceContext`：

```
@Component
@Transactional
public class UserService {
    @PersistenceContext
    EntityManager em;
}
```

我们回顾一下JDBC、Hibernate和JPA提供的接口，实际上，它们的关系如下：

JDBC	Hibernate	JPA
DataSource	SessionFactory	EntityManagerFactory
Connection	Session	EntityManager

`SessionFactory` 和 `EntityManagerFactory` 相当于 `DataSource`，`Session` 和 `EntityManager` 相当于 `Connection`。每次需要访问数据库的时候，需要获取新的 `Session` 和 `EntityManager`，用完后再关闭。

但是，注意到 `UserService` 注入的不是 `EntityManagerFactory`，而是 `EntityManager`，并且标注了 `@PersistenceContext`。难道使用JPA可以允许多线程操作同一个 `EntityManager`？

实际上这里注入的并不是真正的 `EntityManager`，而是一个 `EntityManager` 的代理类，相当于：

```
public class EntityManagerProxy implements EntityManager {
    private EntityManagerFactory emf;
}
```

Spring遇到标注了 `@PersistenceContext` 的 `EntityManager` 会自动注入代理，该代理会在必要的时候自动打开 `EntityManager`。换句话说，多线程引用的 `EntityManager` 虽然是同一个代理类，但该代理类内部针对不同线程会创建不同的 `EntityManager` 实例。

简单总结一下，标注了 `@PersistenceContext` 的 `EntityManager` 可以被多线程安全地共享。

因此，在 `UserService` 的每个业务方法里，直接使用 `EntityManager` 就很方便。以主键查询为例：

```

public User getUserById(long id) {
    User user = this.em.find(User.class, id);
    if (user == null) {
        throw new RuntimeException("User not found by id: " + id);
    }
    return user;
}

```

JPA同样支持Criteria查询，比如我们需要的查询如下：

```

SELECT * FROM user WHERE email = ?

```

使用Criteria查询的代码如下：

```

public User fetchUserByEmail(String email) {
    // CriteriaBuilder:
    var cb = em.getCriteriaBuilder();
    CriteriaQuery<User> q = cb.createQuery(User.class);
    Root<User> r = q.from(User.class);
    q.where(cb.equal(r.get("email"), cb.parameter(String.class, "e")));
    TypedQuery<User> query = em.createQuery(q);
    // 绑定参数:
    query.setParameter("e", email);
    // 执行查询:
    List<User> list = query.getResultList();
    return list.isEmpty() ? null : list.get(0);
}

```

一个简单的查询用Criteria写出来就像上面那样复杂，太恐怖了，如果条件多加几个，这种写法谁读得懂？

所以，正常人还是建议写JPQL查询，它的语法和HQL基本差不多：

```

public User getUserByEmail(String email) {
    // JPQL查询:
    TypedQuery<User> query = em.createQuery("SELECT u FROM User u WHERE u.email = :e", User.class);
    query.setParameter("e", email);
    List<User> list = query.getResultList();
    if (list.isEmpty()) {
        throw new RuntimeException("User not found by email.");
    }
    return list.get(0);
}

```

同样的，JPA也支持NamedQuery，即先给查询起个名字，再按名字创建查询：

```

public User login(String email, String password) {
    TypedQuery<User> query = em.createNamedQuery("login", User.class);
    query.setParameter("e", email);
    query.setParameter("p", password);
    List<User> list = query.getResultList();
    return list.isEmpty() ? null : list.get(0);
}

```

NamedQuery通过注解标注在 `User` 类上，它的定义和上一节的 `User` 类一样：

```
@NamedQueries(  
    @NamedQuery(  
        name = "login",  
        query = "SELECT u FROM User u WHERE u.email=:e AND u.password=:p"  
    )  
)  
@Entity  
public class User {  
    ...  
}
```

对数据库进行增删改的操作，可以分别使用 `persist()`、`remove()` 和 `merge()` 方法，参数均为Entity Bean本身，使用非常简单，这里不再多述。

练习

从  **gitee** 下载练习：[使用JPA](#)（推荐使用[IDE练习插件](#)快速下载）

小结

在Spring中集成JPA要选择一个实现，可以选择Hibernate或EclipseLink；

使用JPA与Hibernate类似，但注入的核心资源是带有 `@PersistenceContext` 注解的 `EntityManager` 代理类。

集成MyBatis

使用Hibernate或JPA操作数据库时，这类ORM干的主要工作就是把ResultSet的每一行变成Java Bean，或者把Java Bean自动转换到INSERT或UPDATE语句的参数中，从而实现ORM。

而ORM框架之所以知道如何把行数据映射到Java Bean，是因为我们在Java Bean的属性上给了足够的注解作为元数据，ORM框架获取Java Bean的注解后，就知道如何进行双向映射。

那么，ORM框架是如何跟踪Java Bean的修改，以便在 `update()` 操作中更新必要的属性？

答案是使用[Proxy模式](#)，从ORM框架读取的User实例实际上并不是User类，而是代理类，代理类继承自User类，但针对每个setter方法做了覆写：

```
public class UserProxy extends User {  
    boolean _isNameChanged;  
  
    public void setName(String name) {  
        super.setName(name);  
        _isNameChanged = true;  
    }  
}
```

这样，代理类可以跟踪到每个属性的变化。

针对一对多或多对一关系时，代理类可以直接通过getter方法查询数据库：

```
public class UserProxy extends User {  
    Session _session;  
    boolean _isNameChanged;
```



```

    public void setName(String name) {
        super.setName(name);
        _isNameChanged = true;
    }

    /**
     * 获取User对象关联的Address对象:
     */
    public Address getAddress() {
        Query q = _session.createQuery("from Address where userId = :userId");
        q.setParameter("userId", this.getId());
        List<Address> list = query.list();
        return list.isEmpty() ? null : list(0);
    }
}

```

为了实现这样的查询，UserProxy必须保存Hibernate的当前Session。但是，当事务提交后，Session自动关闭，此时再获取 getAddress() 将无法访问数据库，或者获取的不是事务一致的数据。因此，ORM框架总是引入了Attached/Detached状态，表示当前此Java Bean到底是在Session的范围内，还是脱离了Session变成了一个“游离”对象。很多初学者无法正确理解状态变化和事务边界，就会造成大量的 PersistentObjectException 异常。这种隐式状态使得普通Java Bean的生命周期变得复杂。

此外，Hibernate和JPA为了实现兼容多种数据库，它使用HQL或JPQL查询，经过一道转换，变成特定数据库的SQL，理论上这样可以做到无缝切换数据库，但这一层自动转换除了少许的性能开销外，给SQL级别的优化带来了麻烦。

最后，ORM框架通常提供了缓存，并且还分为一级缓存和二级缓存。一级缓存是指在一个Session范围内的缓存，常见的情景是根据主键查询时，两次查询可以返回同一实例：

```

User user1 = session.load(User.class, 123);
User user2 = session.load(User.class, 123);

```

二级缓存是指跨Session的缓存，一般默认关闭，需要手动配置。二级缓存极大的增加了数据的不一致性，原因在于SQL非常灵活，常常会导致意外的更新。例如：

```

// 线程1读取：
User user1 = session1.load(User.class, 123);
...
// 一段时间后，线程2读取：
User user2 = session2.load(User.class, 123);

```

当二级缓存生效的时候，两个线程读取的User实例是一样的，但是，数据库对应的行记录完全可能被修改，例如：

```

-- 给老用户增加100积分：
UPDATE users SET bonus = bonus + 100 WHERE createdAt <= ?

```

ORM无法判断 id=123 的用户是否受该 UPDATE 语句影响。考虑到数据库通常会支持多个应用程序，此 UPDATE语句可能由其他进程执行，ORM框架就更知道了。

我们把这种ORM框架称之为全自动ORM框架。

对比Spring提供的JdbcTemplate，它和ORM框架相比，主要有几点差别：

1. 查询后需要手动提供Mapper实例以便把ResultSet的每一行变为Java对象；

2. 增删改操作所需的参数列表，需要手动传入，即把User实例变为[user.id, user.name, user.email]这样的列表，比较麻烦。

但是JdbcTemplate的优势在于它的确性：即每次读取操作一定是数据库操作而不是缓存，所执行的SQL是完全确定的，缺点就是代码比较繁琐，构造 `INSERT INTO users VALUES (?, ?, ?)` 更是复杂。

所以，介于全自动ORM如Hibernate和手写全部如JdbcTemplate之间，还有一种半自动的ORM，它只负责把ResultSet自动映射到Java Bean，或者自动填充Java Bean参数，但仍需自己写出SQL。[MyBatis](#)就是这样一种半自动化ORM框架。

我们来看看如何在Spring中集成MyBatis。

首先，我们要引入MyBatis本身，其次，由于Spring并没有像Hibernate那样内置对MyBatis的集成，所以，我们需要再引入MyBatis官方自己开发的一个与Spring集成的库：

- org.mybatis:mybatis:3.5.4
- org.mybatis:mybatis-spring:2.0.4

和前面一样，先创建 DataSource 是必不可少的：

```
@Configuration
@ComponentScan
@EnableTransactionManagement
@PropertySource("jdbc.properties")
public class AppConfig {
    @Bean
    DataSource createDataSource() { ... }
}
```

再回顾一下Hibernate和JPA的 `SessionFactory` 与 `EntityManagerFactory`，MyBatis与之对应的是 `SqlSessionFactory` 和 `SqlSession`：

JDBC	Hibernate	JPA	MyBatis
DataSource	SessionFactory	EntityManagerFactory	SqlSessionFactory
Connection	Session	EntityManager	SqlSession

可见，ORM的设计套路都是类似的。使用MyBatis的核心就是创建 `SqlSessionFactory`，这里我们需要创建的是 `SqlSessionFactoryBean`：

```
@Bean
SqlSessionFactoryBean createSqlSessionFactoryBean(@Autowired DataSource
dataSource) {
    var sqlSessionFactoryBean = new SqlSessionFactoryBean();
    sqlSessionFactoryBean.setDataSource(dataSource);
    return sqlSessionFactoryBean;
}
```

因为MyBatis可以直接使用Spring管理的声明式事务，因此，创建事务管理器和使用JDBC是一样的：

```
@Bean
PlatformTransactionManager createTxManager(@Autowired DataSource dataSource) {
    return new DataSourceTransactionManager(dataSource);
}
```

和Hibernate不同的是，MyBatis使用Mapper来实现映射，而且Mapper必须是接口。我们以User类为例，在User类和users表之间映射的UserMapper编写如下：

```
public interface UserMapper {
    @Select("SELECT * FROM users WHERE id = #{id}")
    User getById(@Param("id") long id);
}
```

注意：这里的Mapper不是JdbcTemplate的RowMapper的概念，它是定义访问users表的接口方法。比如我们定义了一个 `User getById(long)` 的主键查询方法，不仅要定义接口方法本身，还要明确写出查询的SQL，这里用注解 `@Select` 标记。SQL语句的任何参数，都与方法参数按名称对应。例如，方法参数 `id` 的名字通过注解 `@Param()` 标记为 `id`，则SQL语句里将来替换的占位符就是 `#{id}`。

如果有多个参数，那么每个参数命名后直接在SQL中写出对应的占位符即可：

```
@Select("SELECT * FROM users LIMIT #{offset}, #{maxResults}")
List<User> getAll(@Param("offset") int offset, @Param("maxResults") int
maxResults);
```

注意：MyBatis执行查询后，将根据方法的返回类型自动把ResultSet的每一行转换为User实例，转换规则当然是按列名和属性名对应。如果列名和属性名不同，最简单的方式是编写SELECT语句的别名：

```
-- 列名是created_time，属性名是createdAt:
SELECT id, name, email, created_time AS createdAt FROM users
```

执行INSERT语句就稍微麻烦点，因为我们希望传入User实例，因此，定义的方法接口与 `@Insert` 注解如下：

```
@Insert("INSERT INTO users (email, password, name, createdAt) VALUES (#
{user.email}, #{user.password}, #{user.name}, #{user.createdAt})")
void insert(@Param("user") User user);
```

上述方法传入的参数名称是 `user`，参数类型是User类，在SQL中引用的时候，以 `#{obj.property}` 的方式写占位符。和Hibernate这样的全自动化ORM相比，MyBatis必须写出完整的INSERT语句。

如果 `users` 表的 `id` 是自增主键，那么，我们在SQL中不传入 `id`，但希望获取插入后的主键，需要再加一个 `@Options` 注解：

```
@Options(useGeneratedKeys = true, keyProperty = "id", keyColumn = "id")
@Insert("INSERT INTO users (email, password, name, createdAt) VALUES (#
{user.email}, #{user.password}, #{user.name}, #{user.createdAt})")
void insert(@Param("user") User user);
```

`keyProperty` 和 `keyColumn` 分别指出JavaBean的属性和数据库的主键列名。

执行UPDATE和DELETE语句相对比较简单，我们定义方法如下：

```
@Update("UPDATE users SET name = #{user.name}, createdAt = #{user.createdAt}
WHERE id = #{user.id}")
void update(@Param("user") User user);

@Delete("DELETE FROM users WHERE id = #{id}")
void deleteById(@Param("id") long id);
```

有了 `UserMapper` 接口，还需要对应的实现类才能真正执行这些数据库操作的方法。虽然可以自己写实现类，但我们除了编写 `UserMapper` 接口外，还有 `BookMapper`、`BonusMapper` 一个一个写太麻烦，因此，MyBatis 提供了一个 `MapperFactoryBean` 来自动创建所有 Mapper 的实现类。可以用一个简单的注解来启用它：

```
@MapperScan("com.itranswarp.learnjava.mapper")
...其他注解...
public class AppConfig {
    ...
}
```

有了 `@MapperScan`，就可以让 MyBatis 自动扫描指定包的所有 Mapper 并创建实现类。在真正的业务逻辑中，我们可以直接注入：

```
@Component
@Transactional
public class UserService {
    // 注入UserMapper:
    @Autowired
    UserMapper userMapper;

    public User getUserById(long id) {
        // 调用Mapper方法:
        User user = userMapper.getById(id);
        if (user == null) {
            throw new RuntimeException("User not found by id.");
        }
        return user;
    }
}
```

可见，业务逻辑主要就是通过 `xxxMapper` 定义的数据库方法来访问数据库。

XML配置

上述在 Spring 中集成 MyBatis 的方式，我们只需要用到注解，并没有任何 XML 配置文件。MyBatis 也允许使用 XML 配置映射关系和 SQL 语句，例如，更新 `user` 时根据属性值构造动态 SQL：

```
<update id="updateUser">
    UPDATE users SET
    <set>
        <if test="user.name != null"> name = #{user.name} </if>
        <if test="user.hobby != null"> hobby = #{user.hobby} </if>
        <if test="user.summary != null"> summary = #{user.summary} </if>
    </set>
    WHERE id = #{user.id}
</update>
```

编写 XML 配置的优点是可以组装出动态 SQL，并且把所有 SQL 操作集中在一起。缺点是配置起来太繁琐，调用方法时如果想查看 SQL 还需要定位到 XML 配置中。这里我们不介绍 XML 的配置方式，需要了解的内容请自行阅读[官方文档](#)。

使用MyBatis最大的问题是所有SQL都需要全部手写，优点是执行的SQL就是我们自己写的SQL，对SQL进行优化非常简单，也可以编写任意复杂的SQL，或者使用数据库的特定语法，但切换数据库可能就不太容易。好消息是大部分项目并没有切换数据库的需求，完全可以针对某个数据库编写尽可能优化的SQL。

练习

从  **gitee** 下载练习：[集成MyBatis](#)（推荐使用[IDE练习插件](#)快速下载）

小结

MyBatis是一个半自动化的ORM框架，需要手写SQL语句，没有自动加载一对多或多对一关系的功能。

设计ORM

我们从前几节可以看到，所谓ORM，也是建立在JDBC的基础上，通过ResultSet到JavaBean的映射，实现各种查询。有自动跟踪Entity修改的全自动化ORM如Hibernate和JPA，需要为每个Entity创建代理，也有完全自己映射，连INSERT和UPDATE语句都需要手动编写的MyBatis，但没有任何透明的Proxy。

而查询是涉及到数据库使用最广泛的操作，需要最大的灵活性。各种ORM解决方案各不相同，Hibernate和JPA自己实现了HQL和JPQL查询语法，用以生成最终的SQL，而MyBatis则完全手写，每增加一个查询都需要先编写SQL并增加接口方法。

还有一种Hibernate和JPA支持的Criteria查询，用Hibernate写出来类似：

```
DetachedCriteria criteria = DetachedCriteria.forClass(User.class);
criteria.add(Restrictions.eq("email", email))
    .add(Restrictions.eq("password", password));
List<User> list = (List<User>) hibernateTemplate.findByCriteria(criteria);
```

上述Criteria查询写法复杂，但和JPA相比，还是小巫见大巫了：

```
var cb = em.getCriteriaBuilder();
CriteriaQuery<User> q = cb.createQuery(User.class);
Root<User> r = q.from(User.class);
q.where(cb.equal(r.get("email"), cb.parameter(String.class, "e")));
TypedQuery<User> query = em.createQuery(q);
query.setParameter("e", email);
List<User> list = query.getResultList();
```

此外，是否支持自动读取一对多和多对一关系也是全自动化ORM框架的一个重要功能。

如果我们自己来设计并实现一个ORM，应该吸取这些ORM的哪些特色，然后高效实现呢？

设计ORM接口

任何设计，都必须明确设计目标。这里我们准备实现的ORM并不想要全自动ORM那种自动读取一对多和多对一关系的功能，也不想给Entity加上复杂的状态，因此，对于Entity来说，它就是纯粹的JavaBean，没有任何Proxy。

此外，ORM要兼顾易用性和适用性。易用性是指能覆盖95%的应用场景，但总有一些复杂的SQL，很难用ORM去自动生成，因此，也要给出原生的JDBC接口，能支持5%的特殊需求。

最后，我们希望设计的接口要易于编写，并使用流式API便于阅读。为了配合编译器检查，还应该支持泛型，避免强制转型。

以User类为例，我们设计的查询接口如下：

```
// 按主键查询: SELECT * FROM users WHERE id = ?
User u = db.get(User.class, 123);

// 条件查询唯一记录: SELECT * FROM users WHERE email = ? AND password = ?
User u = db.from(User.class)
    .where("email=? AND password=?", "bob@example.com", "bob123")
    .unique();

// 条件查询多条记录: SELECT * FROM users WHERE id < ? ORDER BY email LIMIT ?, ?
List<User> us = db.from(User.class)
    .where("id < ?", 1000)
    .orderBy("email")
    .limit(0, 10)
    .list();

// 查询特定列: SELECT id, name FROM users WHERE email = ?
User u = db.select("id", "name")
    .from(User.class)
    .where("email = ?", "bob@example.com")
    .unique();
```

这样的流式API便于阅读，也非常容易推导出最终生成的SQL。

对于插入、更新和删除操作，就相对比较简单：

```
// 插入User:
db.insert(user);

// 按主键更新更新User:
db.update(user);

// 按主键删除User:
db.delete(User.class, 123);
```

对于Entity来说，通常一个表对应一个。手动列出所有Entity是非常麻烦的，一定要传入package自动扫描。

最后，ORM总是需要元数据才能知道如何映射。我们不想编写复杂的XML配置，也没必要自己去定义一套规则，直接使用JPA的注解就行。

实现ORM

我们并不需要从JDBC底层开始编写，并且，还要考虑到事务，最好能直接使用Spring的声明式事务。实际上，我们可以设计一个全局 `DbTemplate`，它注入了Spring的 `JdbcTemplate`，涉及到数据库操作时，全部通过 `JdbcTemplate` 完成，自然天生支持Spring的声明式事务，因为这个ORM只是在 `JdbcTemplate` 的基础上做了一层封装。

在 `AppConfig` 中，我们初始化所有Bean如下：

```
@Configuration
@ComponentScan
@EnableTransactionManagement
@PropertySource("jdbc.properties")
public class AppConfig {
    @Bean
    DataSource createDataSource() { ... }
```

```

@Bean
JdbcTemplate createJdbcTemplate(@Autowired DataSource dataSource) {
    return new JdbcTemplate(dataSource);
}

@Bean
DbTemplate createDbTemplate(@Autowired JdbcTemplate jdbcTemplate) {
    return new DbTemplate(jdbcTemplate, "com.itranswarp.learnjava.entity");
}

@Bean
PlatformTransactionManager createTxManager(@Autowired DataSource dataSource)
{
    return new DataSourceTransactionManager(dataSource);
}
}

```

以上就是我们所需的所有配置。

编写业务逻辑，例如 `UserService`，写出来像这样：

```

@Component
@Transactional
public class UserService {
    @Autowired
    DbTemplate db;

    public User getUserById(long id) {
        return db.get(User.class, id);
    }

    public User getUserByEmail(String email) {
        return db.from(User.class)
            .where("email = ?", email)
            .unique();
    }

    public List<User> getUsers(int pageIndex) {
        int pageSize = 100;
        return db.from(User.class)
            .orderBy("id")
            .limit((pageIndex - 1) * pageSize, pageSize)
            .list();
    }

    public User register(String email, String password, String name) {
        User user = new User();
        user.setEmail(email);
        user.setPassword(password);
        user.setName(name);
        user.setCreatedAt(System.currentTimeMillis());
        db.insert(user);
        return user;
    }
    ...
}

```

上述代码给出了ORM的接口，以及如何在业务逻辑中使用ORM。下一步，就是如何实现这个 `DbTemplate`。这里我们只给出框架代码，有兴趣的童鞋可以自己实现核心代码：

```
public class DbTemplate {
    private JdbcTemplate jdbcTemplate;

    // 保存Entity Class到Mapper的映射:
    private Map<Class<?>, Mapper<?>> classMapping;

    public <T> T fetch(Class<T> clazz, Object id) {
        Mapper<T> mapper = getMapper(clazz);
        List<T> list = (List<T>) jdbcTemplate.query(mapper.selectSQL, new
Object[] { id }, mapper.rowMapper);
        if (list.isEmpty()) {
            return null;
        }
        return list.get(0);
    }

    public <T> T get(Class<T> clazz, Object id) {
        ...
    }

    public <T> void insert(T bean) {
        ...
    }

    public <T> void update(T bean) {
        ...
    }

    public <T> void delete(Class<T> clazz, Object id) {
        ...
    }
}
```

实现链式API的核心代码是第一步从 `DbTemplate` 调用 `select()` 或 `from()` 时实例化一个 `CriteriaQuery` 实例，并在后续的链式调用中设置它的字段：

```
public class DbTemplate {
    ...
    public Select select(String... selectFields) {
        return new Select(new Criteria(this), selectFields);
    }

    public <T> From<T> from(Class<T> entityClass) {
        Mapper<T> mapper = getMapper(entityClass);
        return new From<>(new Criteria<>(this), mapper);
    }
}
```

然后以此定义 `Select`、`From`、`Where`、`OrderBy`、`Limit` 等。在 `From` 中可以设置Class类型、表名等：


```

public final class From<T> extends CriteriaQuery<T> {
    From(Criteria<T> criteria, Mapper<T> mapper) {
        super(criteria);
        // from可以设置class、tableName:
        this.criteria.mapper = mapper;
        this.criteria.clazz = mapper.entityClass;
        this.criteria.table = mapper.tableName;
    }

    public where<T> where(String clause, Object... args) {
        return new Where<>(this.criteria, clause, args);
    }
}

```

在 `where` 中可以设置条件参数:

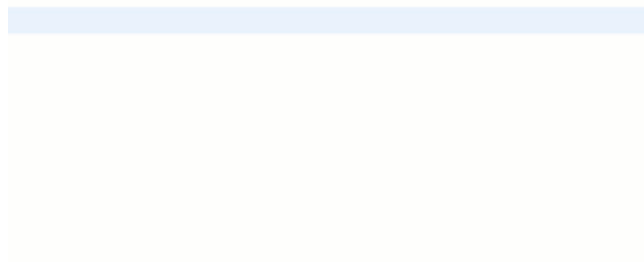
```

public final class Where<T> extends CriteriaQuery<T> {
    Where(Criteria<T> criteria, String clause, Object... params) {
        super(criteria);
        this.criteria.where = clause;
        this.criteria.whereParams = new ArrayList<>();
        // add:
        for (Object param : params) {
            this.criteria.whereParams.add(param);
        }
    }
}

```

最后，链式调用的尽头是调用 `list()` 返回一组结果，调用 `unique()` 返回唯一结果，调用 `first()` 返回首个结果。

在IDE中，可以非常方便地实现链式调用：



需要复杂查询的时候，总是可以使用 `JdbcTemplate` 执行任意复杂的SQL。

练习

从  **gitee** 下载练习：[设计并实现一个微型ORM](#)（推荐使用[IDE练习插件](#)快速下载）

小结

ORM框架就是自动映射数据库表结构到JavaBean的工具，设计并实现一个简单高效的ORM框架并不困难。

开发Web应用

在[Web开发](#)一章中，我们已经详细介绍了JavaEE中Web开发的基础：Servlet。具体地说，有以下几点：

1. Servlet规范定义了几种标准组件：Servlet、JSP、Filter和Listener；
2. Servlet的标准组件总是运行在Servlet容器中，如Tomcat、Jetty、WebLogic等。

直接使用Servlet进行Web开发好比直接在JDBC上操作数据库，比较繁琐，更好的方法是在Servlet基础上封装MVC框架，基于MVC开发Web应用，大部分时候，不需要接触Servlet API，开发省时省力。

我们在[MVC开发](#)和[MVC高级开发](#)已经由浅入深地介绍了如何编写MVC框架。当然，自己写的MVC主要是理解原理，要实现一个功能全面的MVC需要大量的工作以及广泛的测试。

因此，开发Web应用，首先要选择一个优秀的MVC框架。常用的MVC框架有：

- [Struts](#)：最古老的一个MVC框架，目前版本是2，和1.x有很大的区别；
- WebWork：一个比Struts设计更优秀的MVC框架，但不知道出于什么原因，从2.0开始把自己的代码全部塞给Struts 2了；
- [Turbine](#)：一个重度使用Velocity，强调布局的MVC框架；
- 其他100+MVC框架.....（略）

Spring虽然都可以集成任何Web框架，但是，Spring本身也开发了一个MVC框架，就叫[Spring MVC](#)。这个MVC框架设计得足够优秀以至于我们已经不想再费劲去集成类似Struts这样的框架了。

本章我们会详细介绍如何基于Spring MVC开发Web应用。

使用Spring MVC

我们在前面介绍[Web开发](#)时已经讲过了Java Web的基础：Servlet容器，以及标准的Servlet组件：

- Servlet：能处理HTTP请求并将HTTP响应返回；
- JSP：一种嵌套Java代码的HTML，将被编译为Servlet；
- Filter：能过滤指定的URL以实现拦截功能；
- Listener：监听指定的事件，如ServletContext、HttpSession的创建和销毁。

此外，Servlet容器为每个Web应用程序自动创建一个唯一的 `ServletContext` 实例，这个实例就代表了Web应用程序本身。

在[MVC高级开发](#)中，我们手撸了一个MVC框架，接口和Spring MVC类似。如果直接使用Spring MVC，我们写出来的代码类似：

```
@Controller
public class UserController {
    @GetMapping("/register")
    public ModelAndView register() {
        ...
    }

    @PostMapping("/signin")
    public ModelAndView signin(@RequestParam("email") String email,
    @RequestParam("password") String password) {
        ...
    }
    ...
}
```

但是，Spring提供的是一个IoC容器，所有的Bean，包括Controller，都在Spring IoC容器中被初始化，而Servlet容器由JavaEE服务器提供（如Tomcat），Servlet容器对Spring一无所知，他们之间到底依靠什么进行联系，又是以何种顺序初始化的？

在理解上述问题之前，我们先把基于Spring MVC开发的项目结构搭建起来。首先创建基于Web的Maven工程，引入如下依赖：

- org.springframework:spring-context:5.2.0.RELEASE
- org.springframework:spring-webmvc:5.2.0.RELEASE
- org.springframework:spring-jdbc:5.2.0.RELEASE
- javax.annotation:javax.annotation-api:1.3.2
- io.pebbletemplates:pebble-spring5:3.1.2
- ch.qos.logback:logback-core:1.2.3
- ch.qos.logback:logback-classic:1.2.3
- com.zaxxer:HikariCP:3.4.2
- org.hsqldb:hsqldb:2.5.0

以及 provided 依赖：

- org.apache.tomcat.embed:tomcat-embed-core:9.0.26
- org.apache.tomcat.embed:tomcat-embed-jasper:9.0.26

这个标准的Maven Web工程目录结构如下：

```
spring-web-mvc
├── pom.xml
├── src
│   ├── main
│   │   ├── java
│   │   │   ├── com
│   │   │   │   ├── itranswarp
│   │   │   │   │   ├── learnjava
│   │   │   │   │   │   ├── AppConfig.java
│   │   │   │   │   │   ├── DatabaseInitializer.java
│   │   │   │   │   │   ├── entity
│   │   │   │   │   │   │   ├── User.java
│   │   │   │   │   │   ├── service
│   │   │   │   │   │   │   ├── UserService.java
│   │   │   │   │   │   └── web
│   │   │   │   │   │       └── UserController.java
│   │   │   └── resources
│   │   │       ├── jdbc.properties
│   │   │       └── logback.xml
│   │   └── webapp
│   │       ├── WEB-INF
│   │       │   ├── templates
│   │       │   │   ├── _base.html
│   │       │   │   ├── index.html
│   │       │   │   ├── profile.html
│   │       │   │   ├── register.html
│   │       │   │   └── signin.html
│   │       │   └── web.xml
│   │       └── static
│   │           ├── css
│   │           │   └── bootstrap.css
│   │           └── js
│   │               └── jquery.js
```

其中，`src/main/webapp` 是标准web目录，`WEB-INF` 存放 `web.xml`，编译的class，第三方jar，以及不允许浏览器直接访问的View模版，`static` 目录存放所有静态文件。

在 `src/main/resources` 目录中存放的是Java程序读取的classpath资源文件，除了JDBC的配置文件 `jdbc.properties` 外，我们又新增了一个 `logback.xml`，这是Logback的默认查找的配置文件：

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <appender name="STDOUT"
        class="ch.qos.logback.core.ConsoleAppender">
        <layout class="ch.qos.logback.classic.PatternLayout">
            <Pattern>%d{yyyy-MM-dd HH:mm:ss} %-5level %logger{36} -
%msg%n</Pattern>
        </layout>
    </appender>

    <logger name="com.itranswarp.learnjava" level="info" additivity="false">
        <appender-ref ref="STDOUT" />
    </logger>

    <root level="info">
        <appender-ref ref="STDOUT" />
    </root>
</configuration>
```

上面给出了一个写入到标准输出的Logback配置，可以基于上述配置添加写入到文件的配置。

在 `src/main/java` 中就是我们编写的Java代码了。

配置Spring MVC

和普通Spring配置一样，我们编写正常的 `AppConfig` 后，只需加上 `@EnableWebMvc` 注解，就“激活”了Spring MVC：

```
@Configuration
@ComponentScan
@EnableWebMvc // 启用Spring MVC
@EnableTransactionManagement
@PropertySource("classpath:/jdbc.properties")
public class AppConfig {
    ...
}
```

除了创建 `DataSource`、`JdbcTemplate`、`PlatformTransactionManager` 外，`AppConfig` 需要额外创建几个用于Spring MVC的Bean：

```
@Bean
WebMvcConfigurer createWebMvcConfigurer() {
    return new WebMvcConfigurer() {
        @Override
        public void addResourceHandlers(ResourceHandlerRegistry registry) {

            registry.addResourceHandler("/static/**").addResourceLocations("/static/");
        }
    };
}
```

`WebMvcConfigurer` 并不是必须的，但我们在这里创建一个默认的 `WebMvcConfigurer`，只覆盖 `addResourceHandlers()`，目的是让Spring MVC自动处理静态文件，并且映射路径为 `/static/**`。

另一个必须要创建的Bean是 `ViewResolver`，因为Spring MVC允许集成任何模板引擎，使用哪个模板引擎，就实例化一个对应的 `ViewResolver`：

```

@Bean
viewResolver createViewResolver(@Autowired ServletContext servletContext) {
    PebbleEngine engine = new PebbleEngine.Builder().autoEscaping(true)
        .cacheActive(false)
        .loader(new ServletLoader(servletContext))
        .extension(new SpringExtension())
        .build();
    PebbleViewResolver viewResolver = new PebbleViewResolver();
    viewResolver.setPrefix("/WEB-INF/templates/");
    viewResolver.setSuffix("");
    viewResolver.setPebbleEngine(engine);
    return viewResolver;
}

```

`viewResolver` 通过指定prefix和suffix来确定如何查找View。上述配置使用Pebble引擎，指定模板文件存放在 `/WEB-INF/templates/` 目录下。

剩下的Bean都是普通的 `@Component`，但Controller必须标记为 `@Controller`，例如：

```

// Controller使用@Controller标记而不是@Component:
@Controller
public class UserController {
    // 正常使用@Autowired注入:
    @Autowired
    UserService userService;

    // 处理一个URL映射:
    @GetMapping("/")
    public ModelAndView index() {
        ...
    }
    ...
}

```

如果是普通的Java应用程序，我们通过 `main()` 方法可以很简单地创建一个Spring容器的实例：

```

public static void main(String[] args) {
    ApplicationContext context = new
    AnnotationConfigApplicationContext(AppConfig.class);
}

```

但是问题来了，现在是Web应用程序，而Web应用程序总是由Servlet容器创建，那么，Spring容器应该由谁创建？在什么时候创建？Spring容器中的Controller又是如何通过Servlet调用的？

在Web应用中启动Spring容器有很多种方法，可以通过Listener启动，也可以通过Servlet启动，可以使用XML配置，也可以使用注解配置。这里，我们只介绍一种最简单的启动Spring容器的方式。

第一步，我们在 `web.xml` 中配置Spring MVC提供的 `DispatcherServlet`：

```

<!DOCTYPE web-app PUBLIC
"-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
"http://java.sun.com/dtd/web-app_2_3.dtd" >

<web-app>
    <servlet>

```

```

        <servlet-name>dispatcher</servlet-name>
        <servlet-
class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <init-param>
            <param-name>contextClass</param-name>
            <param-
value>org.springframework.web.context.support.AnnotationConfigWebApplicationCont
ext</param-value>
        </init-param>
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>com.itranswarp.learnjava.AppConfig</param-value>
        </init-param>
        <load-on-startup>0</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>dispatcher</servlet-name>
        <url-pattern>/*</url-pattern>
    </servlet-mapping>
</web-app>

```

初始化参数 `contextClass` 指定使用注解配置的 `AnnotationConfigWebApplicationContext`，配置文件的位置参数 `contextConfigLocation` 指向 `AppConfig` 的完整类名，最后，把这个Servlet映射到 `/*`，即处理所有URL。

上述配置可以看作一个样板配置，有了这个配置，Servlet容器会首先初始化Spring MVC的 `DispatcherServlet`，在 `DispatcherServlet` 启动时，它根据配置 `AppConfig` 创建了一个类型是 `WebApplicationContext`的IoC容器，完成所有Bean的初始化，并将容器绑到 `ServletContext`上。

因为 `DispatcherServlet` 持有IoC容器，能从IoC容器中获取所有 `@Controller` 的Bean，因此，`DispatcherServlet` 接收到所有HTTP请求后，根据Controller方法配置的路径，就可以正确地把请求转发到指定方法，并根据返回的 `ModelAndView` 决定如何渲染页面。

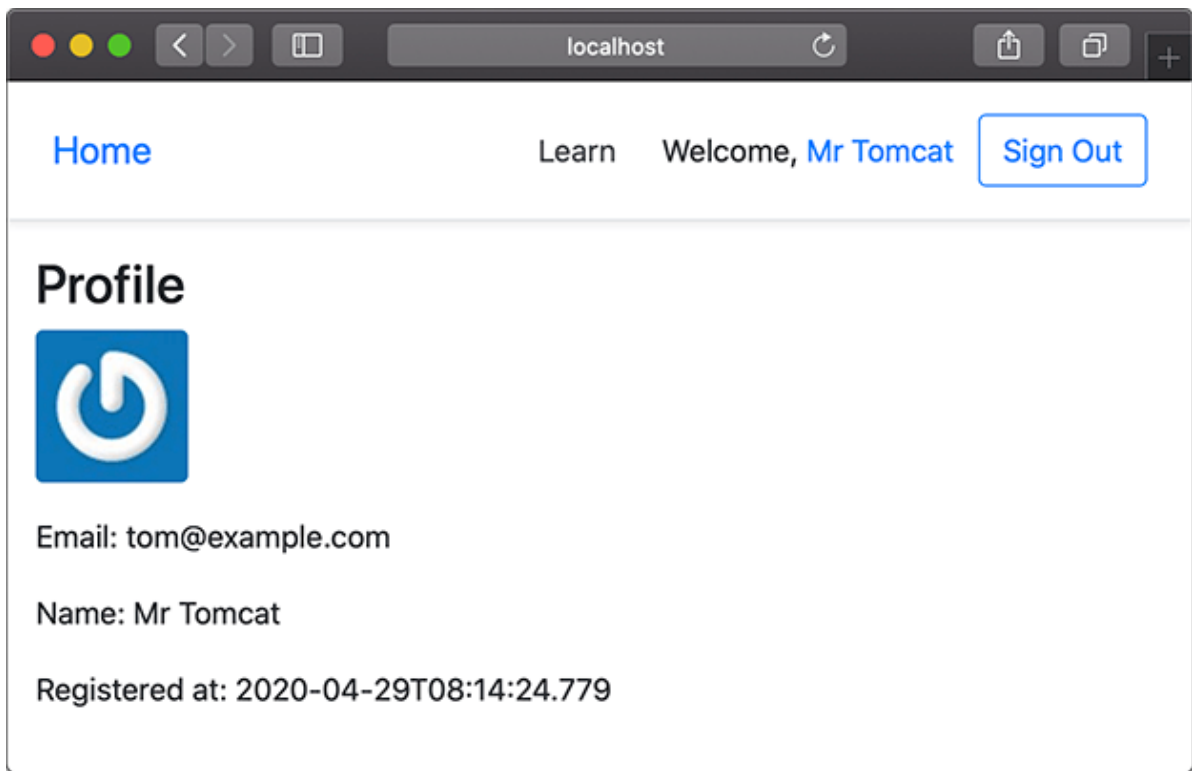
最后，我们在 `AppConfig` 中通过 `main()` 方法启动嵌入式Tomcat：

```

public static void main(String[] args) throws Exception {
    Tomcat tomcat = new Tomcat();
    tomcat.setPort(Integer.getInteger("port", 8080));
    tomcat.getConnector();
    Context ctx = tomcat.addWebapp("", new
File("src/main/webapp").getAbsolutePath());
    WebResourceRoot resources = new StandardRoot(ctx);
    resources.addPreResources(
        new DirResourceSet(resources, "/WEB-INF/classes", new
File("target/classes").getAbsolutePath(), "/"));
    ctx.setResources(resources);
    tomcat.start();
    tomcat.getServer().await();
}

```

上述Web应用程序就是我们使用Spring MVC时的一个最小启动功能集。由于使用了JDBC和数据库，用户的注册、登录信息会被持久化：



编写Controller

有了Web应用程序的最基本的结构，我们的重点就可以放在如何编写Controller上。Spring MVC对Controller没有固定的要求，也不需要实现特定的接口。以UserController为例，编写Controller只需要遵循以下要点：

总是标记 `@Controller` 而不是 `@Component`：

```
@Controller
public class UserController {
    ...
}
```

一个方法对应一个HTTP请求路径，用 `@GetMapping` 或 `@PostMapping` 表示GET或POST请求：

```
@PostMapping("/signin")
public ModelAndView doSignin(
    @RequestParam("email") String email,
    @RequestParam("password") String password,
    HttpSession session) {
    ...
}
```

需要接收的HTTP参数以 `@RequestParam()` 标注，可以设置默认值。如果方法参数需要传入 `HttpServletRequest`、`HttpServletResponse` 或者 `HttpSession`，直接添加这个类型的参数即可，Spring MVC会自动按类型传入。

返回的 `ModelAndView` 通常包含View的路径和一个Map作为Model，但也可以没有Model，例如：

```
return new ModelAndView("signin.html"); // 仅View, 没有Model
```

返回重定向时既可以写 `new ModelAndView("redirect:/signin")`，也可以直接返回String：

```

public String index() {
    if (...) {
        return "redirect:/signin";
    } else {
        return "redirect:/profile";
    }
}

```

如果在方法内部直接操作 `HttpServletResponse` 发送响应，返回 `null` 表示无需进一步处理：

```

public ModelAndView download(HttpServletResponse response) {
    byte[] data = ...
    response.setContentType("application/octet-stream");
    OutputStream output = response.getOutputStream();
    output.write(data);
    output.flush();
    return null;
}

```

对URL进行分组，每组对应一个Controller是一种很好的组织形式，并可以在Controller的class定义出添加URL前缀，例如：

```

@Controller
@RequestMapping("/user")
public class UserController {
    // 注意实际URL映射是/user/profile
    @GetMapping("/profile")
    public ModelAndView profile() {
        ...
    }

    // 注意实际URL映射是/user/changePassword
    @GetMapping("/changePassword")
    public ModelAndView changePassword() {
        ...
    }
}

```

实际方法的URL映射总是前缀+路径，这种形式还可以有效避免不小心导致的重复的URL映射。

可见，Spring MVC允许我们编写既简单又灵活的Controller实现。

练习

在注册、登录等功能的基础上增加一个修改口令的页面。

从  **gitee** 下载练习：[使用Spring MVC](#)（推荐使用[IDE练习插件](#)快速下载）

小结

使用Spring MVC时，整个Web应用程序按如下顺序启动：

1. 启动Tomcat服务器；
2. Tomcat读取web.xml并初始化DispatcherServlet；
3. DispatcherServlet创建IoC容器并自动注册到ServletContext中。

启动后，浏览器发出的HTTP请求全部由DispatcherServlet接收，并根据配置转发到指定Controller的指定方法处理。

使用REST

使用Spring MVC开发Web应用程序的主要工作就是编写Controller逻辑。在Web应用中，除了需要使用MVC给用户显示页面外，还有一类API接口，我们称之为REST，通常输入输出都是JSON，便于第三方调用或者使用页面JavaScript与之交互。

直接在Controller中处理JSON是可以的，因为Spring MVC的 `@GetMapping` 和 `@PostMapping` 都支持指定输入和输出的格式。如果我们想接收JSON，输出JSON，那么可以这样写：

```
@PostMapping(value = "/rest",
              consumes = "application/json;charset=UTF-8",
              produces = "application/json;charset=UTF-8")
@ResponseBody
public String rest(@RequestBody User user) {
    return "{\"restSupport\":true}";
}
```

对应的Maven工程需要加入Jackson这个依赖：`com.fasterxml.jackson.core:jackson-databind:2.11.0`

注意到 `@PostMapping` 使用 `consumes` 声明能接收的类型，使用 `produces` 声明输出的类型，并且额外加了 `@ResponseBody` 表示返回的 `String` 无需额外处理，直接作为输出内容写入 `HttpServletResponse`。输入的JSON则根据注解 `@RequestBody` 直接被Spring反序列化为 `User` 这个 `JavaBean`。

使用curl命令测试一下：

```
$ curl -v -H "Content-Type: application/json" -d '{"email":"bob@example.com"}'
http://localhost:8080/rest
> POST /rest HTTP/1.1
> Host: localhost:8080
> User-Agent: curl/7.64.1
> Accept: */*
> Content-Type: application/json
> Content-Length: 27
>
< HTTP/1.1 200
< Content-Type: application/json;charset=utf-8
< Content-Length: 20
< Date: Sun, 10 May 2020 09:56:01 GMT
<
{"restSupport":true}
```

输出正是我们写入的字符串。

直接用Spring的Controller配合一大堆注解写REST太麻烦了，因此，Spring还额外提供了一个 `@RestController` 注解，使用 `@RestController` 替代 `@Controller` 后，每个方法自动变成API接口方法。我们还是以实际代码举例，编写 `ApiController` 如下：

```
@RestController
@RequestMapping("/api")
public class ApiController {
```

```

@Autowired
UserService userService;

@GetMapping("/users")
public List<User> users() {
    return userService getUsers();
}

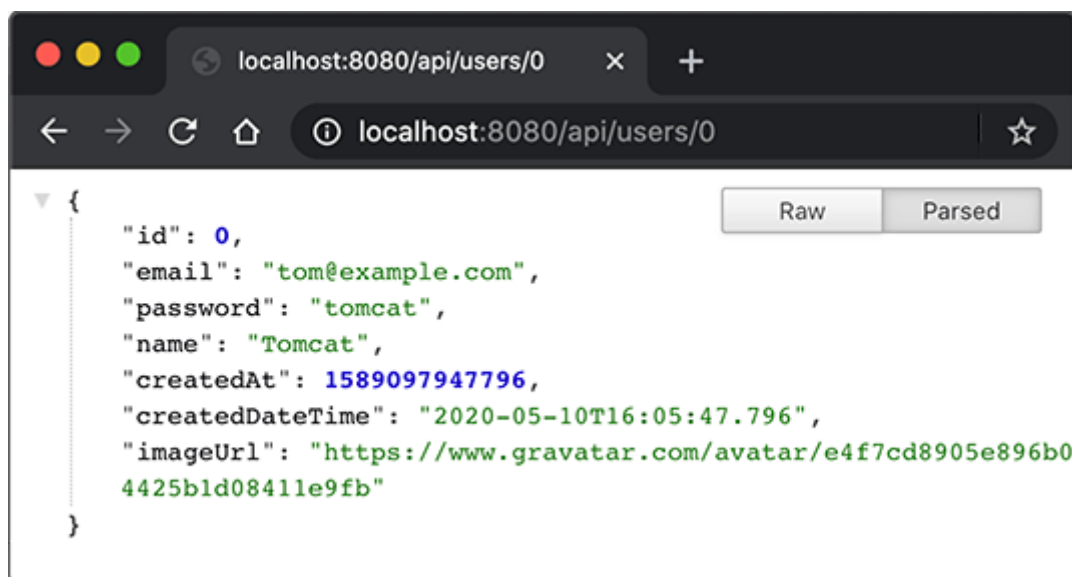
@GetMapping("/users/{id}")
public User user(@PathVariable("id") long id) {
    return userService.getUserById(id);
}

@PostMapping("/signin")
public Map<String, Object> signin(@RequestBody SignInRequest signinRequest)
{
    try {
        User user = userService.signin(signinRequest.email,
signinRequest.password);
        return Map.of("user", user);
    } catch (Exception e) {
        return Map.of("error", "SIGNIN_FAILED", "message", e.getMessage());
    }
}

public static class SignInRequest {
    public String email;
    public String password;
}
}

```

编写REST接口只需要定义 `@RestController`，然后，每个方法都是一个API接口，输入和输出只要能被Jackson序列化或反序列化为JSON就没有问题。我们用浏览器测试GET请求，可直接显示JSON响应：



要测试POST请求，可以用curl命令：

```

$ curl -v -H "Content-Type: application/json" -d
 '{"email":"bob@example.com","password":"bob123"}'
http://localhost:8080/api/signin
> POST /api/signin HTTP/1.1
> Host: localhost:8080

```

```
> User-Agent: curl/7.64.1
> Accept: */*
> Content-Type: application/json
> Content-Length: 47
>
< HTTP/1.1 200
< Content-Type: application/json
< Transfer-Encoding: chunked
< Date: Sun, 10 May 2020 08:14:13 GMT
<
{"user":{"id":1,"email":"bob@example.com","password":"bob123","name":"Bob",...
```

注意观察上述JSON的输出，`User` 能被正确地序列化为JSON，但暴露了 `password` 属性，这是我们不期望的。要避免输出 `password` 属性，可以把 `User` 复制到另一个 `UserBean` 对象，该对象只持有必要的属性，但这样做比较繁琐。另一种简单的方法是直接在 `User` 的 `password` 属性定义处加上 `@JsonIgnore` 表示完全忽略该属性：

```
public class User {
    ...

    @JsonIgnore
    public String getPassword() {
        return password;
    }

    ...
}
```

但是这样一来，如果写一个 `register(User user)` 方法，那么该方法的 `User` 对象也拿不到注册时用户传入的密码了。如果要允许输入 `password`，但不允许输出 `password`，即在JSON序列化和反序列化时，允许写属性，禁用读属性，可以更精细地控制如下：

```
public class User {
    ...

    @JsonProperty(access = Access.WRITE_ONLY)
    public String getPassword() {
        return password;
    }

    ...
}
```

同样的，可以使用 `@JsonProperty(access = Access.READ_ONLY)` 允许输出，不允许输入。

练习

从  **gitee** 下载练习：[使用REST实现API](#)（推荐使用[IDE练习插件](#)快速下载）

小结

使用 `@RestController` 可以方便地编写REST服务，Spring默认使用JSON作为输入和输出。

要控制序列化和反序列化，可以使用Jackson提供的 `@JsonIgnore` 和 `@JsonProperty` 注解。

集成Filter

在Spring MVC中，`DispatcherServlet` 只需要固定配置到 `web.xml` 中，剩下的工作主要是专注于编写 `Controller`。

但是，在Servlet规范中，我们还可以[使用Filter](#)。如果要在Spring MVC中使用 `Filter`，应该怎么做？

有的童鞋在上一节的Web应用中可能发现了，如果注册时输入中文会导致乱码，因为Servlet默认按非UTF-8编码读取参数。为了修复这一问题，我们可以简单地使用一个`EncodingFilter`，在全局范围类给 `HttpServletRequest` 和 `HttpServletResponse` 强制设置为UTF-8编码。

可以自己编写一个`EncodingFilter`，也可以直接使用Spring MVC自带的一个 `CharacterEncodingFilter`。配置Filter时，只需在 `web.xml` 中声明即可：

```
<web-app>
  <filter>
    <filter-name>encodingFilter</filter-name>
    <filter-
class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>
    <init-param>
      <param-name>encoding</param-name>
      <param-value>UTF-8</param-value>
    </init-param>
    <init-param>
      <param-name>forceEncoding</param-name>
      <param-value>true</param-value>
    </init-param>
  </filter>

  <filter-mapping>
    <filter-name>encodingFilter</filter-name>
    <url-pattern>/*</url-pattern>
  </filter-mapping>
  ...
</web-app>
```

因为这种Filter和我们业务关系不大，注意到 `CharacterEncodingFilter` 其实和Spring的IoC容器没有任何关系，两者均互不知晓对方的存在，因此，配置这种Filter十分简单。

我们再考虑这样一个问题：如果允许用户使用Basic模式进行用户验证，即在HTTP请求中添加头 `Authorization: Basic email:password`，这个需求如何实现？

编写一个 `AuthFilter` 是最简单的实现方式：

```
@Component
public class AuthFilter implements Filter {
    @Autowired
    UserService userService;

    public void doFilter(ServletRequest request, ServletResponse response,
        FilterChain chain)
        throws IOException, ServletException {
        HttpServletRequest req = (HttpServletRequest) request;
        // 获取Authorization头:
        String authHeader = req.getHeader("Authorization");
        if (authHeader != null && authHeader.startsWith("Basic ")) {
            // 从Header中提取email和password:
        }
    }
}
```

```

        String email = prefixFrom(authHeader);
        String password = suffixFrom(authHeader);
        // 登录:
        User user = userService.signin(email, password);
        // 放入Session:
        req.getSession().setAttribute(UserController.KEY_USER, user);
    }
    // 继续处理请求:
    chain.doFilter(request, response);
}
}

```

现在问题来了：在Spring中创建的这个 `AuthFilter` 是一个普通Bean，Servlet容器并不知道，所以它不会起作用。

如果我们直接在 `web.xml` 中声明这个 `AuthFilter`，注意到 `AuthFilter` 的实例将由Servlet容器而不是Spring容器初始化，因此，`@Autowired` 根本不生效，用于登录的 `UserService` 成员变量永远是 `null`。

所以，得通过一种方式，让Servlet容器实例化的Filter，间接引用Spring容器实例化的 `AuthFilter`。Spring MVC提供了一个 `DelegatingFilterProxy`，专门来干这个事情：

```

<web-app>
  <filter>
    <filter-name>authFilter</filter-name>
    <filter-
class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>
  </filter>

  <filter-mapping>
    <filter-name>authFilter</filter-name>
    <url-pattern>/*</url-pattern>
  </filter-mapping>
  ...
</web-app>

```

我们来看实现原理：

1. Servlet容器从 `web.xml` 中读取配置，实例化 `DelegatingFilterProxy`，注意命名是 `authFilter`；
2. Spring容器通过扫描 `@Component` 实例化 `AuthFilter`。

当 `DelegatingFilterProxy` 生效后，它会自动查找注册在 `ServletContext` 上的Spring容器，再试图从容器中查找名为 `authFilter` 的Bean，也就是我们用 `@Component` 声明的 `AuthFilter`。

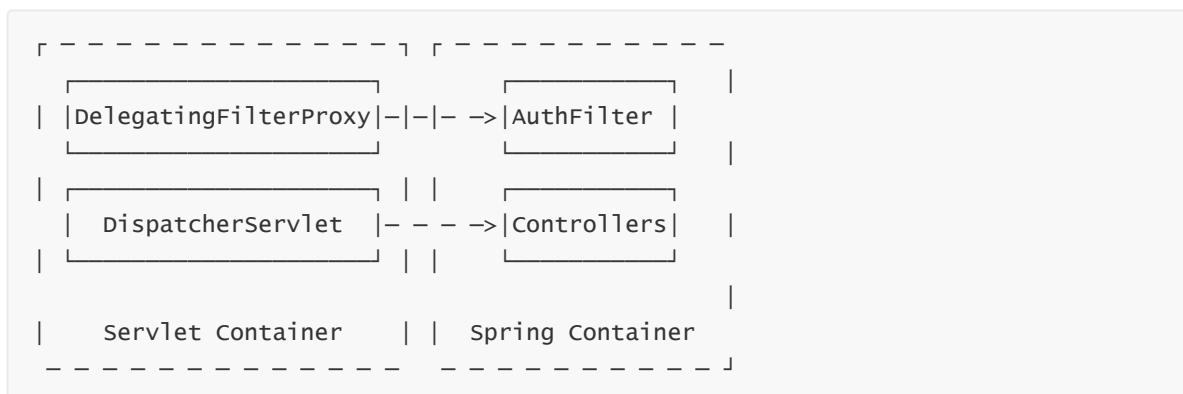
`DelegatingFilterProxy` 将请求代理给 `AuthFilter`，核心代码如下：

```

public class DelegatingFilterProxy implements Filter {
    private Filter delegate;
    public void doFilter(...) throws ... {
        if (delegate == null) {
            delegate = findBeanFromSpringContainer();
        }
        delegate.doFilter(req, resp, chain);
    }
}

```

这就是一个[代理模式](#)的简单应用。我们画个图表示它们之间的引用关系如下：



如果在 `web.xml` 中配置的Filter名字和Spring容器的Bean的名字不一致，那么需要指定Bean的名字：

```
<filter>
  <filter-name>basicAuthFilter</filter-name>
  <filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-
class>
  <!-- 指定Bean的名字 -->
  <init-param>
    <param-name>targetBeanName</param-name>
    <param-value>authFilter</param-value>
  </init-param>
</filter>
```

实际应用时，尽量保持名字一致，以减少不必要的配置。

要使用Basic模式的用户认证，我们可以使用curl命令测试。例如，用户登录名是 `tom@example.com`，口令是 `tomcat`，那么先构造一个使用URL编码的 `用户名:口令` 的字符串：

```
tom%40example.com:tomcat
```

对其进行Base64编码，最终构造出的Header如下：

```
Authorization: Basic dG9tJTQwZXhhbXBsZS5jb206dG9tY2F0
```

使用如下的 `curl` 命令并获得响应如下：


```
$ curl -v -H 'Authorization: Basic dG9tJTQwZXhhbXBsZS5jb206dG9tY2F0'
http://localhost:8080/profile
> GET /profile HTTP/1.1
> Host: localhost:8080
> User-Agent: curl/7.64.1
> Accept: */*
> Authorization: Basic dG9tJTQwZXhhbXBsZS5jb206dG9tY2F0
>
< HTTP/1.1 200
< Set-Cookie: JSESSIONID=CE0F4BFC394816F717443397D4FEABBE; Path=/; HttpOnly
< Content-Type: text/html; charset=UTF-8
< Content-Language: en-CN
< Transfer-Encoding: chunked
< Date: wed, 29 Apr 2020 00:15:50 GMT
<
```

```
<!doctype html>
...HTML输出...
```

上述响应说明 `AuthFilter` 已生效。

注意：Basic认证模式并不安全，本节只用来作为使用Filter的示例。

练习

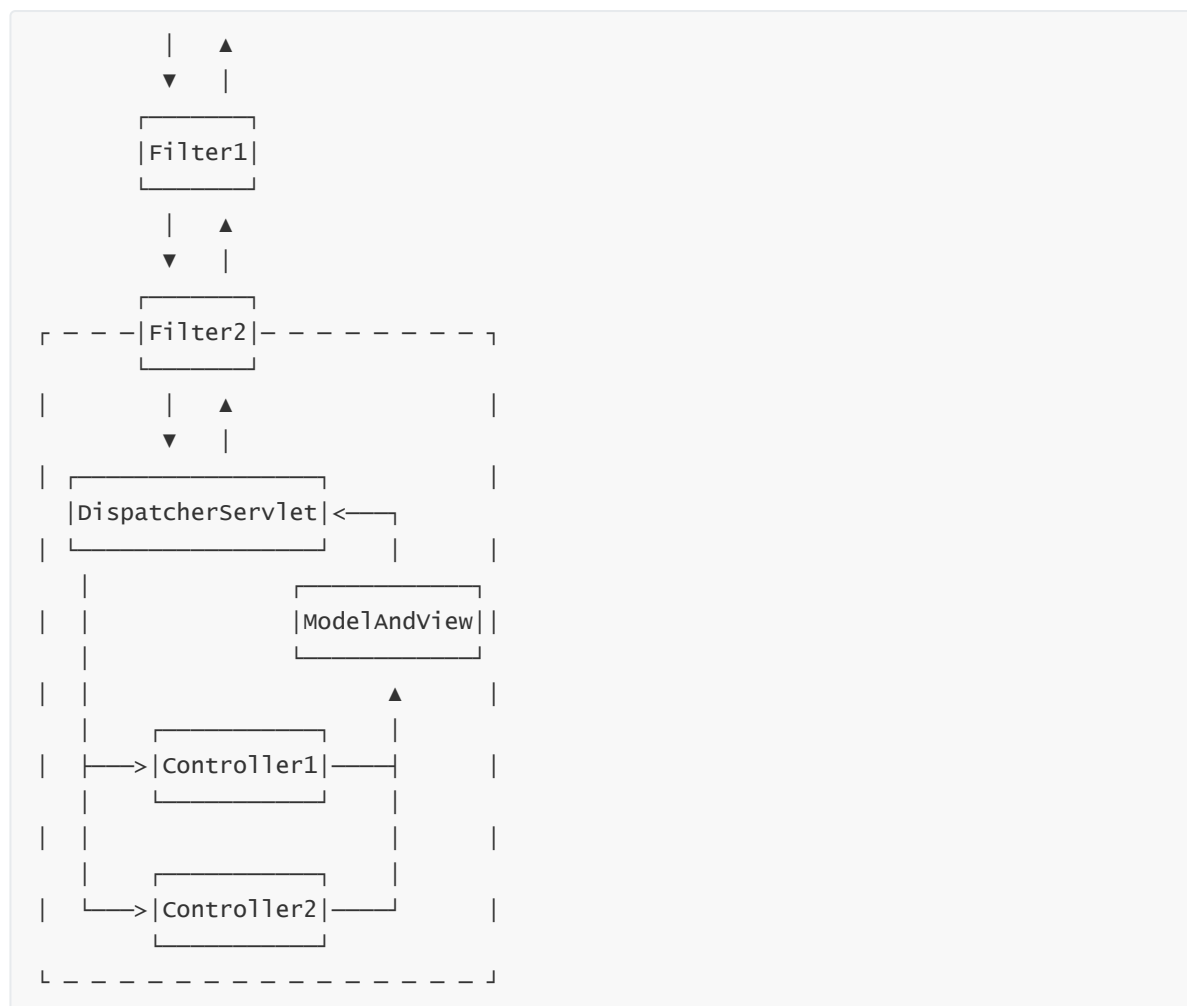
从  **gitee** 下载练习：[使用DelegatingFilterProxy实现AuthFilter](#)（推荐使用[IDE练习插件](#)快速下载）

小结

当一个Filter作为Spring容器管理的Bean存在时，可以通过 `DelegatingFilterProxy` 间接地引用它并使其生效。

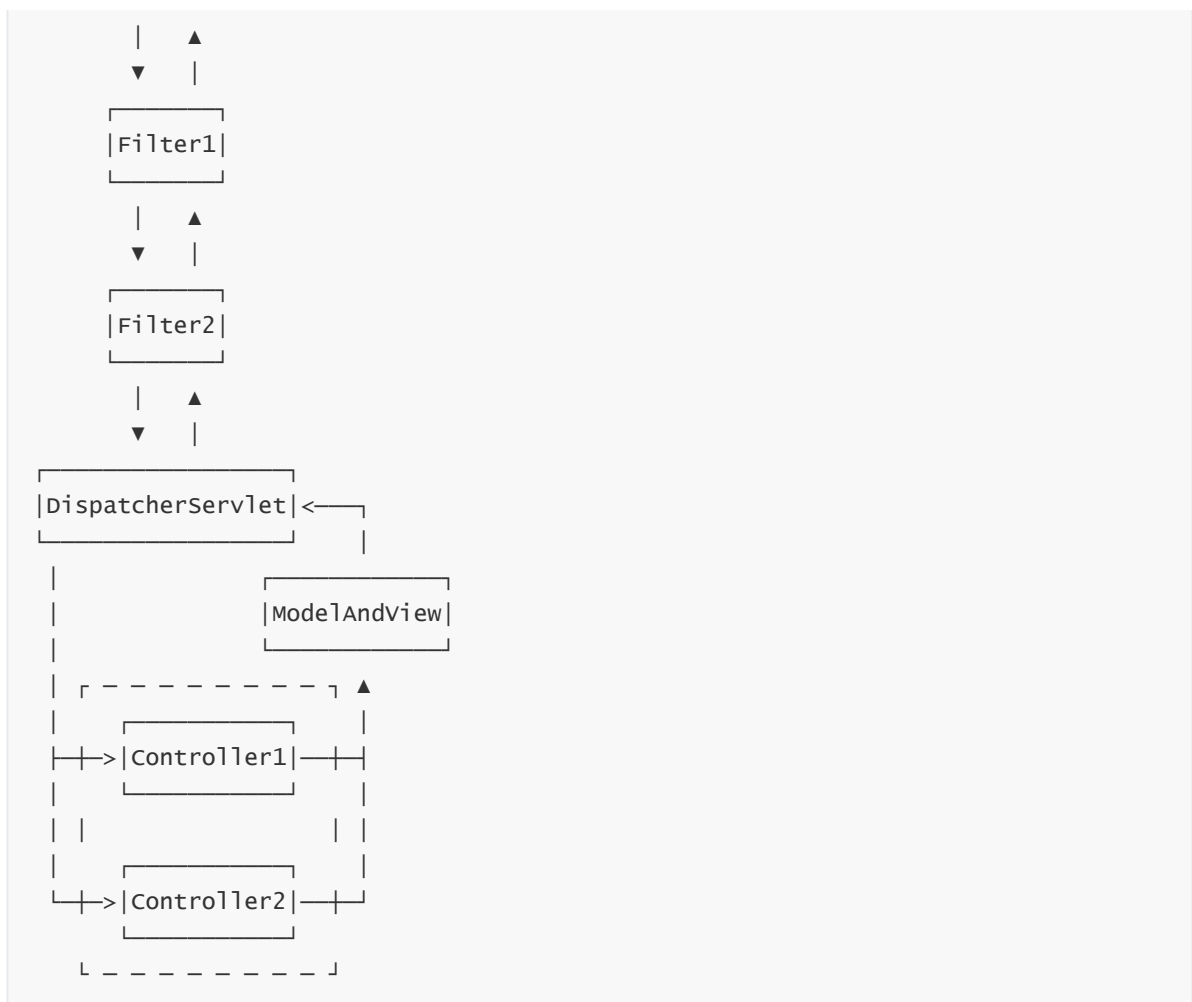
使用Interceptor

在Web程序中，注意到使用Filter的时候，Filter由Servlet容器管理，它在Spring MVC的Web应用程序中作用范围如下：



上图虚线框就是Filter2的拦截范围，Filter组件实际上并不知道后续内部处理是通过Spring MVC提供的 `DispatcherServlet` 还是其他Servlet组件，因为Filter是Servlet规范定义的标准组件，它可以应用在任何基于Servlet的程序中。

如果只基于Spring MVC开发应用程序，还可以使用Spring MVC提供的一种功能类似Filter的拦截器：`Interceptor`。和Filter相比，`Interceptor`拦截范围不是后续整个处理流程，而是仅针对Controller拦截：



上图虚线框就是Interceptor的拦截范围，注意到Controller的处理方法一般都类似这样：

```
@Controller
public class Controller1 {
    @GetMapping("/path/to/hello")
    ModelAndView hello() {
        ...
    }
}
```

所以，Interceptor的拦截范围其实就是Controller方法，它实际上就相当于基于AOP的方法拦截。因为Interceptor只拦截Controller方法，所以要注意，返回 `ModelAndView` 后，后续对View的渲染就脱离了Interceptor的拦截范围。

使用Interceptor的好处是Interceptor本身是Spring管理的Bean，因此注入任意Bean都非常简单。此外，可以应用多个Interceptor，并通过简单的 `@order` 指定顺序。我们先写一个

`LoggerInterceptor`：

```
@Order(1)
@Component
public class LoggerInterceptor implements HandlerInterceptor {

    final Logger logger = LoggerFactory.getLogger(getClass());

    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse
response, Object handler) throws Exception {
        logger.info("preHandle {}...", request.getRequestURI());
    }
}
```



```

        if (request.getParameter("debug") != null) {
            PrintWriter pw = response.getWriter();
            pw.write("<p>DEBUG MODE</p>");
            pw.flush();
            return false;
        }
        return true;
    }

    @Override
    public void postHandle(HttpServletRequest request, HttpServletResponse
response, Object handler, ModelAndView modelAndView) throws Exception {
        logger.info("postHandle {}. ", request.getRequestURI());
        if (modelAndView != null) {
            modelAndView.addObject("__time__", LocalDateTime.now());
        }
    }

    @Override
    public void afterCompletion(HttpServletRequest request, HttpServletResponse
response, Object handler, Exception ex) throws Exception {
        logger.info("afterCompletion {}: exception = {}",
request.getRequestURI(), ex);
    }
}

```

一个Interceptor必须实现 `HandlerInterceptor` 接口，可以选择实现 `preHandle()`、`postHandle()` 和 `afterCompletion()` 方法。`preHandle()` 是Controller方法调用前执行，`postHandle()` 是Controller方法正常返回后执行，而 `afterCompletion()` 无论Controller方法是否抛异常都会执行，参数 `ex` 就是Controller方法抛出的异常（未抛出异常是 `null`）。

在 `preHandle()` 中，也可以直接处理响应，然后返回 `false` 表示无需调用Controller方法继续处理了，通常在认证或者安全检查失败时直接返回错误响应。在 `postHandle()` 中，因为捕获了Controller方法返回的 `ModelAndView`，所以可以继续往 `ModelAndView` 里添加一些通用数据，很多页面需要的全局数据如Copyright信息等都可以放到这里，无需在每个Controller方法中重复添加。

我们再继续添加一个 `AuthInterceptor`，用于替代上一节使用 `AuthFilter` 进行Basic认证的功能：

```

@Order(2)
@Component
public class AuthInterceptor implements HandlerInterceptor {

    final Logger logger = LoggerFactory.getLogger(getClass());

    @Autowired
    UserService userService;

    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse
response, Object handler)
        throws Exception {
        logger.info("pre authenticate {}...", request.getRequestURI());
        try {
            authenticateByHeader(request);
        } catch (RuntimeException e) {
            logger.warn("login by authorization header failed.", e);
        }
    }
}

```

```

    }
    return true;
}

private void authenticateByHeader(HttpServletRequest req) {
    String authHeader = req.getHeader("Authorization");
    if (authHeader != null && authHeader.startsWith("Basic ")) {
        logger.info("try authenticate by authorization header...");
        String up = new
String(Base64.getDecoder().decode(authHeader.substring(6)),
StandardCharsets.UTF_8);
        int pos = up.indexOf(':');
        if (pos > 0) {
            String email = URLDecoder.decode(up.substring(0, pos),
StandardCharsets.UTF_8);
            String password = URLDecoder.decode(up.substring(pos + 1),
StandardCharsets.UTF_8);
            User user = userService.signin(email, password);
            req.getSession().setAttribute(UserController.KEY_USER, user);
            logger.info("user {} login by authorization header ok.", email);
        }
    }
}
}
}
}

```

这个 `AuthInterceptor` 是由Spring容器直接管理的，因此注入 `UserService` 非常方便。

最后，要让拦截器生效，我们在 `WebMvcConfigurer` 中注册所有的Interceptor：

```

@Bean
WebMvcConfigurer createWebMvcConfigurer(@Autowired HandlerInterceptor[]
interceptors) {
    return new WebMvcConfigurer() {
        public void addInterceptors(InterceptorRegistry registry) {
            for (var interceptor : interceptors) {
                registry.addInterceptor(interceptor);
            }
        }
        ...
    };
}

```

如果拦截器没有生效，请检查是否忘了在 `WebMvcConfigurer` 中注册。

处理异常

在Controller中，Spring MVC还允许定义基于 `@ExceptionHandler` 注解的异常处理方法。我们来看具体的示例代码：

```
@Controller
public class UserController {
    @ExceptionHandler(RuntimeException.class)
    public ModelAndView handleUnknownException(Exception ex) {
        return new ModelAndView("500.html", Map.of("error",
            ex.getClass().getSimpleName(), "message", ex.getMessage()));
    }
    ...
}
```

异常处理方法没有固定的方法签名，可以传入 `Exception`、`HttpServletRequest` 等，返回值可以是 `void`，也可以是 `ModelAndView`，上述代码通过 `@ExceptionHandler(RuntimeException.class)` 表示当发生 `RuntimeException` 的时候，就自动调用此方法处理。

注意到我们返回了一个新的 `ModelAndView`，这样在应用程序内部如果发生了预料之外的异常，可以给用户显示一个出错页面，而不是简单的500 Internal Server Error或404 Not Found。例如B站的错误页：



可以编写多个错误处理方法，每个方法针对特定的异常。例如，处理 `LoginException` 使得页面可以自动跳转到登录页。

使用 `ExceptionHandler` 时，要注意它仅作用于当前的Controller，即ControllerA中定义的一个 `ExceptionHandler` 方法对ControllerB不起作用。如果我们有很多Controller，每个Controller都需要处理一些通用的异常，例如 `LoginException`，思考一下应该怎么避免重复代码？

练习

从  **gitee** 下载练习：[使用Interceptor](#)（推荐使用[IDE练习插件](#)快速下载）

小结

Spring MVC提供了Interceptor组件来拦截Controller方法，使用时要注意Interceptor的作用范围。

处理CORS

在开发REST应用时，很多时候，是通过页面的JavaScript和后端的REST API交互。

在JavaScript与REST交互的时候，有很多安全限制。默认情况下，浏览器按同源策略放行JavaScript调用API，即：

- 如果A站在域名 `a.com` 页面的JavaScript调用A站自己的API时，没有问题；

- 如果A站在域名 `a.com` 页面的JavaScript调用B站 `b.com` 的API时，将被浏览器拒绝访问，因为不满足同源策略。

同源要求域名要完全相同（`a.com` 和 `www.a.com` 不同），协议要相同（`http` 和 `https` 不同），端口要相同。

那么，在域名 `a.com` 页面的JavaScript要调用B站 `b.com` 的API时，还有没有办法？

办法是有的，那就是CORS，全称Cross-Origin Resource Sharing，是HTML5规范定义的如何跨域访问资源。如果A站的JavaScript访问B站API的时候，B站能够返回响应头 `Access-Control-Allow-Origin: http://a.com`，那么，浏览器就允许A站的JavaScript访问B站的API。

注意到跨域访问能否成功，取决于B站是否愿意给A站返回一个正确的 `Access-Control-Allow-Origin` 响应头，所以决定权永远在提供API的服务方手中。

关于CORS的详细信息可以参考[MDN文档](#)，这里不再详述。

使用Spring的 `@RestController` 开发REST应用时，同样会面对跨域问题。如果我们允许指定的网站通过页面JavaScript访问这些REST API，就必须正确地设置CORS。

有好几种方法设置CORS，我们来——介绍。

使用@CrossOrigin

第一种方法是使用 `@CrossOrigin` 注解，可以在 `@RestController` 的class级别或方法级别定义一个 `@CrossOrigin`，例如：

```
@CrossOrigin(origins = "http://local.liaoxuefeng.com:8080")
@RestController
@RequestMapping("/api")
public class ApiController {
    ...
}
```

上述定义在 `ApiController` 处的 `@CrossOrigin` 指定了只允许来自 `local.liaoxuefeng.com` 跨域访问，允许多个域访问需要写成数组形式，例如 `origins = {"http://a.com", "https://www.b.com"}`。如果要允许任何域访问，写成 `origins = "*" 即可。`

如果有多个REST Controller都需要使用CORS，那么，每个Controller都必须标注 `@CrossOrigin` 注解。

使用CorsRegistry

第二种方法是在 `WebMvcConfigurer` 中定义一个全局CORS配置，下面是一个示例：

```
@Bean
webMvcConfigurer createWebMvcConfigurer() {
    return new WebMvcConfigurer() {
        @Override
        public void addCorsMappings(CorsRegistry registry) {
            registry.addMapping("/api/**")
                .allowedOrigins("http://local.liaoxuefeng.com:8080")
                .allowedMethods("GET", "POST")
                .maxAge(3600);
            // 可以继续添加其他URL规则：
            // registry.addMapping("/rest/v2/**")...
        }
    };
}
```

这种方式可以创建一个全局CORS配置，如果仔细地设计URL结构，那么可以一目了然地看到各个URL的CORS规则，推荐使用这种方式配置CORS。

使用CorsFilter

第三种方法是使用Spring提供的 `CorsFilter`，我们在[集成Filter](#)中详细介绍了将Spring容器内置的Bean暴露为Servlet容器的Filter的方法，由于这种配置方式需要修改 `web.xml`，也比较繁琐，所以推荐使用第二种方式。

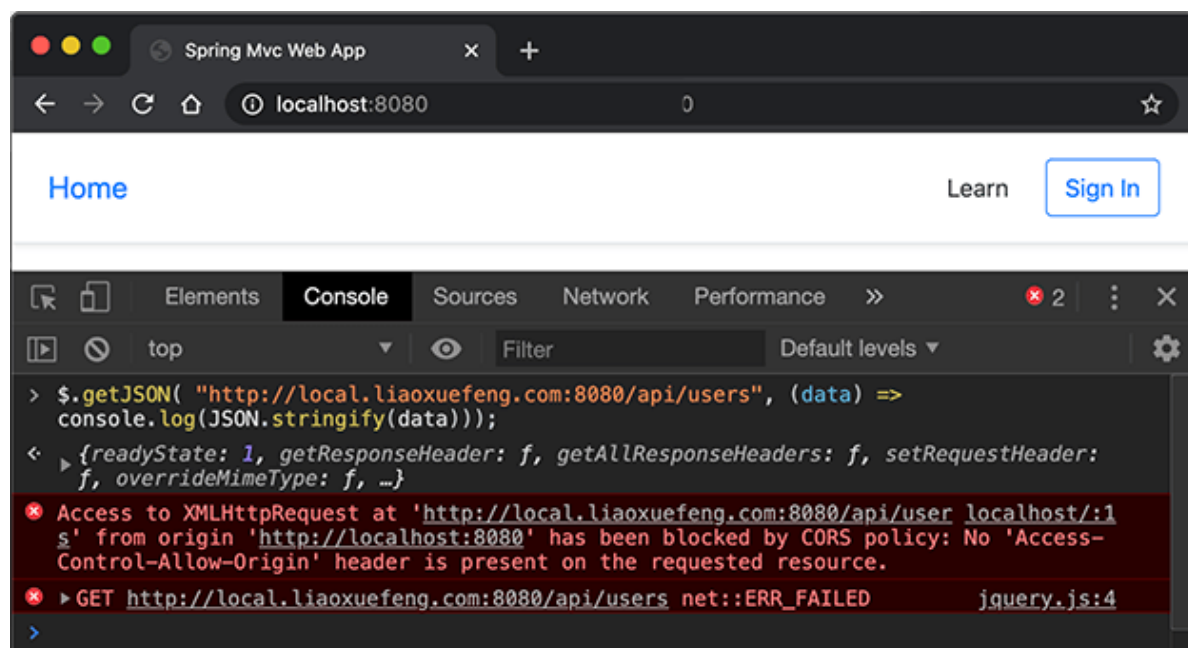
测试

当我们配置好CORS后，可以在浏览器中测试一下规则是否生效。

我们先用 `http://localhost:8080` 在Chrome浏览器中打开首页，然后打开Chrome的开发者工具，切换到Console，输入一个JavaScript语句来跨域访问API：

```
$.getJSON( "http://local.liaoxuefeng.com:8080/api/users", (data) =>
  console.log(JSON.stringify(data)));
```

上述源站的域是 `http://localhost:8080`，跨域访问的是 `http://local.liaoxuefeng.com:8080`，因为配置的CORS不允许 `localhost` 访问，所以不出意外地得到一个错误：

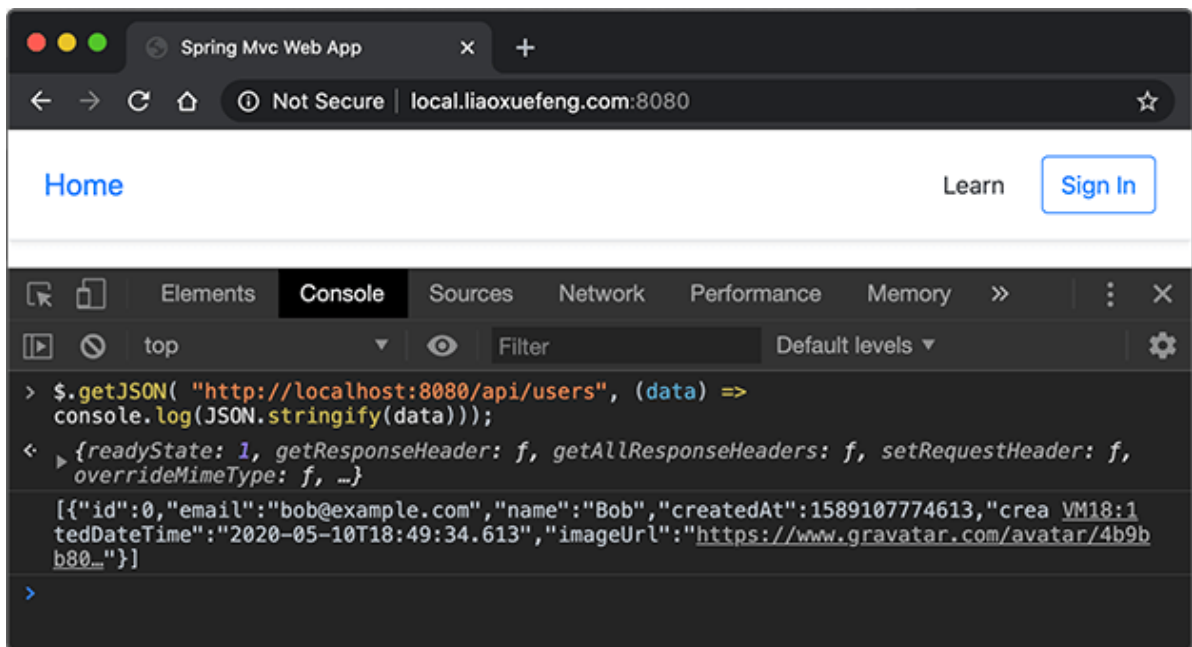


浏览器打印了错误原因就是 `been blocked by CORS policy`。

我们再用 `http://local.liaoxuefeng.com:8080` 在Chrome浏览器中打开首页，在Console中执行JavaScript访问 `localhost`：

```
$.getJSON( "http://localhost:8080/api/users", (data) =>
  console.log(JSON.stringify(data)));
```

因为CORS规则允许来自 `http://local.liaoxuefeng.com:8080` 的访问，因此访问成功，打印出API的返回值：



练习

从  **gitee** 下载练习: [使用CORS控制跨域](#) (推荐使用[IDE练习插件](#)快速下载)

小结

CORS可以控制指定域的页面JavaScript能否访问API。

国际化

在开发应用程序的时候,经常会遇到支持多语言的需求,这种支持多语言的功能称之为国际化,英文是internationalization,缩写为i18n(因为首字母i和末字母n中间有18个字母)。

还有针对特定地区的本地化功能,英文是localization,缩写为L10n,本地化是指根据地区调整类似姓名、日期的显示等。

也有把上面两者合称为全球化,英文是globalization,缩写为g11n。

在Java中,支持多语言和本地化是通过 MessageFormat 配合 Locale 实现的:

```
// MessageFormat
import java.text.MessageFormat;
import java.util.Locale;

public class Time {
    public static void main(String[] args) {
        double price = 123.5;
        int number = 10;
        Object[] arguments = { price, number };
        MessageFormat mfUS = new MessageFormat("Pay {0,number,currency} for {1} books.", Locale.US);
        System.out.println(mfUS.format(arguments));
        MessageFormat mfZH = new MessageFormat("{1}本书一共{0,number,currency}。", Locale.CHINA);
        System.out.println(mfZH.format(arguments));
    }
}
```

Pay \$123.50 for 10 books.
10本书一共¥123.50。

对于Web应用程序，要实现国际化功能，主要是渲染View的时候，要把各种语言的资源文件提出来，这样，不同的用户访问同一个页面时，显示的语言就是不同的。

我们来看看在Spring MVC应用程序中如何实现国际化。

获取Locale

实现国际化的第一步是获取到用户的 `Locale`。在Web应用程序中，HTTP规范规定了浏览器会在请求中携带 `Accept-Language` 头，用来指示用户浏览器设定的语言顺序，如：

```
Accept-Language: zh-CN,zh;q=0.8,en;q=0.2
```

上述HTTP请求头表示优先选择简体中文，其次选择中文，最后选择英文。`q` 表示权重，解析后我们可获得一个根据优先级排序的语言列表，把它转换为Java的 `Locale`，即获得了用户的 `Locale`。大多数框架通常只返回权重最高的 `Locale`。

Spring MVC通过 `LocaleResolver` 来自动从 `HttpServletRequest` 中获取 `Locale`。有多种 `LocaleResolver` 的实现类，其中最常用的是 `CookieLocaleResolver`：

```
@Bean
LocaleResolver createLocaleResolver() {
    var clr = new CookieLocaleResolver();
    clr.setDefaultLocale(Locale.ENGLISH);
    clr.setDefaultTimeZone(TimeZone.getDefault());
    return clr;
}
```

`CookieLocaleResolver` 从 `HttpServletRequest` 中获取 `Locale` 时，首先根据一个特定的Cookie判断是否指定了 `Locale`，如果没有，就从HTTP头获取，如果还没有，就返回默认的 `Locale`。

当用户第一次访问网站时，`CookieLocaleResolver` 只能从HTTP头获取 `Locale`，即使用浏览器的默认语言。通常网站也允许用户自己选择语言，此时，`CookieLocaleResolver` 就会把用户选择的语言存放到Cookie中，下一次访问时，就会返回用户上次选择的语言而不是浏览器默认语言。

提取资源文件

第二步是把写死在模板中的字符串以资源文件的方式存储在外部。对于多语言，主文件名如果命名为 `messages`，那么资源文件必须按如下方式命名并放入classpath中：

- 默认语言，文件名必须为 `messages.properties`；
- 简体中文，Locale是 `zh_CN`，文件名必须为 `messages_zh_CN.properties`；
- 日文，Locale是 `ja_JP`，文件名必须为 `messages_ja_JP.properties`；
- 其它更多语言.....

每个资源文件都有相同的key，例如，默认语言是英文，文件 `messages.properties` 内容如下：

```
language.select=Language
home=Home
signin=Sign In
copyright=Copyright@{0,number,#}
```

文件 `messages_zh_CN.properties` 内容如下：


```
language.select=语言
home=首页
signin=登录
copyright=版权所有©{0,number,#}
```

创建MessageSource

第三步是创建一个Spring提供的 `MessageSource` 实例，它自动读取所有的 `.properties` 文件，并提供一个统一接口来实现“翻译”：

```
// code, arguments, locale:
String text = messageSource.getMessage("signin", null, locale);
```

其中，`signin` 是我们在 `.properties` 文件中定义的key，第二个参数是 `Object[]` 数组作为格式化时传入的参数，最后一个参数就是获取的用户 `Locale` 实例。

创建 `MessageSource` 如下：

```
@Bean("i18n")
MessageSource createMessageSource() {
    var messageSource = new ResourceBundleMessageSource();
    // 指定文件是UTF-8编码:
    messageSource.setDefaultEncoding("UTF-8");
    // 指定主文件名:
    messageSource.setBasename("messages");
    return messageSource;
}
```

注意到 `ResourceBundleMessageSource` 会自动根据主文件名自动把所有相关语言的资源文件都读进来。

再注意到Spring容器会创建不只一个 `MessageSource` 实例，我们自己创建的这个 `MessageSource` 是专门给页面国际化使用的，因此命名为 `i18n`，不会与其它 `MessageSource` 实例冲突。

实现多语言

要在View中使用 `MessageSource` 加上 `Locale` 输出多语言，我们通过编写一个 `MvcInterceptor`，把相关资源注入到 `ModelAndView` 中：

```
@Component
public class MvcInterceptor implements HandlerInterceptor {
    @Autowired
    LocaleResolver localeResolver;

    // 注意注入的MessageSource名称是i18n:
    @Autowired
    @Qualifier("i18n")
    MessageSource messageSource;

    public void postHandle(HttpServletRequest request, HttpServletResponse
response, Object handler, ModelAndView modelAndView) throws Exception {
        if (modelAndView != null) {
            // 解析用户的Locale:
            Locale locale = localeResolver.resolveLocale(request);
            // 放入Model:
```



```

        modelAndView.addObject("__messageSource__", messageSource);
        modelAndView.addObject("__locale__", locale);
    }
}
}

```

不要忘了在 `WebMvcConfigurer` 中注册 `MvcInterceptor`。现在，就可以在 View 中调用 `MessageSource.getMessage()` 方法来实现多语言：

```

<a href="/signin">{{ __messageSource__.getMessage('signin', null, __locale__) }}
</a>

```

上述这种写法虽然可行，但格式太复杂了。使用 View 时，要根据每个特定的 View 引擎定制国际化函数。在 Pebble 中，我们可以封装一个国际化函数，名称就是下划线 `_`，改造一下创建 `ViewResolver` 的代码：

```

@Bean
ViewResolver createViewResolver(@Autowired ServletContext servletContext,
@Nullable @Qualifier("i18n") MessageSource messageSource) {
    PebbleEngine engine = new PebbleEngine.Builder()
        .autoEscaping(true)
        .cacheActive(false)
        .loader(new ServletLoader(servletContext))
        // 添加扩展：
        .extension(createExtension(messageSource))
        .build();
    PebbleViewResolver viewResolver = new PebbleViewResolver();
    viewResolver.setPrefix("/WEB-INF/templates/");
    viewResolver.setSuffix("");
    viewResolver.setPebbleEngine(engine);
    return viewResolver;
}

private Extension createExtension(MessageSource messageSource) {
    return new AbstractExtension() {
        @Override
        public Map<String, Function> getFunctions() {
            return Map.of("_", new Function() {
                public Object execute(Map<String, Object> args, PebbleTemplate
self, EvaluationContext context, int lineNumber) {
                    String key = (String) args.get("0");
                    List<Object> arguments = this.extractArguments(args);
                    Locale locale = (Locale) context.getVariable("__locale__");
                    return messageSource.getMessage(key, arguments.toArray(),
"???" + key + "???", locale);
                }
            });
        }
        private List<Object> extractArguments(Map<String, Object> args)
{
            int i = 1;
            List<Object> arguments = new ArrayList<>();
            while (args.containsKey(String.valueOf(i))) {
                Object param = args.get(String.valueOf(i));
                arguments.add(param);
                i++;
            }
            return arguments;
        }
    };
}

```

```

        }
        public List<String> getArgumentNames() {
            return null;
        }
    });
}
};
}

```

这样，我们可以把多语言页面改写为：

```
<a href="/signin">{{ _('signin') }}</a>
```

如果是带参数的多语言，需要把参数传进去：

```
<h5>{{ _('copyright', 2020) }}</h5>
```

使用其它View引擎时，也应当根据引擎接口实现更方便的语法。

切换Locale

最后，我们需要允许用户手动切换 `Locale`，编写一个 `LocaleController` 来实现该功能：

```

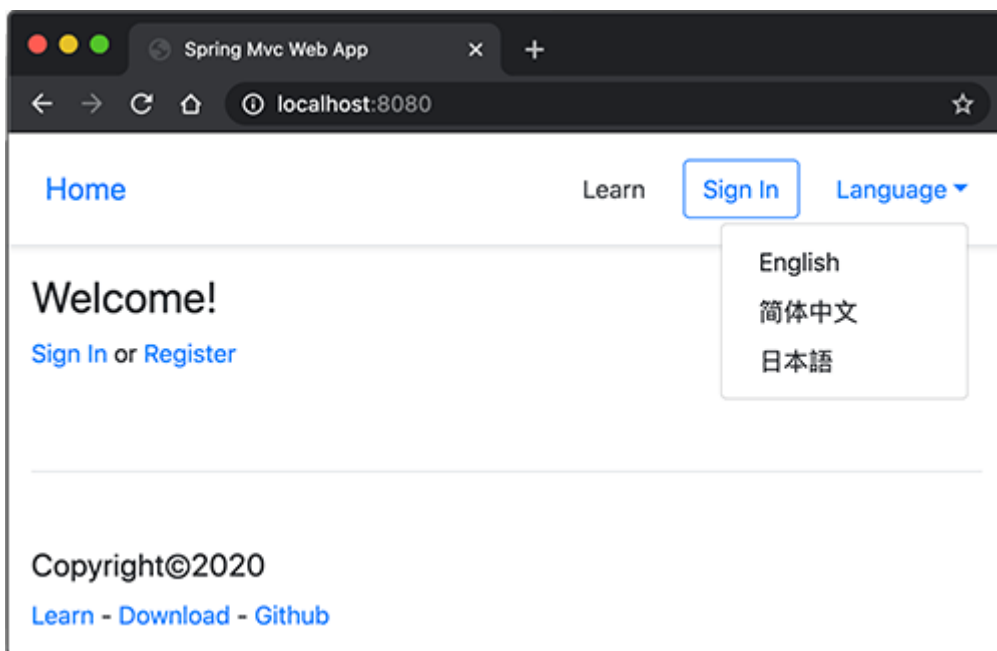
@Controller
public class LocaleController {
    final Logger logger = LoggerFactory.getLogger(getClass());

    @Autowired
    LocaleResolver localeResolver;

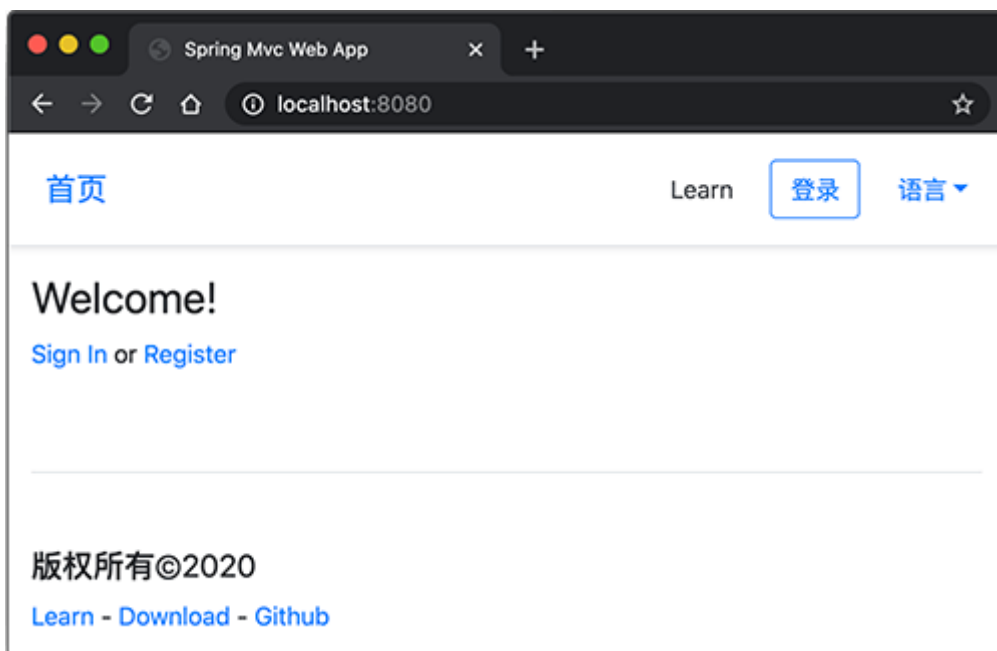
    @GetMapping("/locale/{lo}")
    public String setLocale(@PathVariable("lo") String lo, HttpServletRequest
request, HttpServletResponse response) {
        // 根据传入的lo创建Locale实例：
        Locale locale = null;
        int pos = lo.indexOf('_');
        if (pos > 0) {
            String lang = lo.substring(0, pos);
            String country = lo.substring(pos + 1);
            locale = new Locale(lang, country);
        } else {
            locale = new Locale(lo);
        }
        // 设定此Locale：
        localeResolver.setLocale(request, response, locale);
        logger.info("locale is set to {}. ", locale);
        // 刷新页面：
        String referer = request.getHeader("Referer");
        return "redirect:" + (referer == null ? "/" : referer);
    }
}

```

在页面设计中，通常在右上角给用户提供一个语言选择列表，来看看效果：



切换到中文：



练习

从  **gitee** 下载练习：[在MVC程序中实现国际化](#)（推荐使用[IDE练习插件](#)快速下载）

小结

多语言支持需要从HTTP请求中解析用户的Locale，然后针对不同Locale显示不同的语言；

Spring MVC应用程序通过 `MessageSource` 和 `LocaleResolver`，配合View实现国际化。

异步处理

在Servlet模型中，每个请求都是由某个线程处理，然后，将响应写入IO流，发送给客户端。从开始处理请求，到写入响应完成，都是在同一个线程中处理的。

实现Servlet容器的时候，只要每处理一个请求，就创建一个新线程处理它，就能保证正确实现了Servlet线程模型。在实际产品中，例如Tomcat，总是通过线程池来处理请求，它仍然符合一个请求从头到尾都由某一个线程处理。

这种线程模型非常重要，因为Spring的JDBC事务是基于 `ThreadLocal` 实现的，如果在处理过程中，一会由线程A处理，一会又由线程B处理，那事务就全乱套了。此外，很多安全认证，也是基于 `ThreadLocal` 实现的，可以保证在处理请求的过程中，各个线程互不影响。

但是，如果一个请求处理的时间较长，例如几秒钟甚至更长，那么，这种基于线程池的同步模型很快就会把所有线程耗尽，导致服务器无法响应新的请求。如果把长时间处理的请求改为异步处理，那么线程池的利用率就会大大提高。Servlet从3.0规范开始添加了异步支持，允许对一个请求进行异步处理。

我们先来看看在Spring MVC中如何实现对请求进行异步处理的逻辑。首先建立一个Web工程，然后编辑 `web.xml` 文件如下：

```
<web-app 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/web-app_3_1.xsd"
    version="3.1">
    <display-name>Archetype Created Web Application</display-name>

    <servlet>
        <servlet-name>dispatcher</servlet-name>
        <servlet-
class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <init-param>
            <param-name>contextClass</param-name>
            <param-
value>org.springframework.web.context.support.AnnotationConfigWebApplicationContext</param-value>
        </init-param>
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>com.itranswarp.learnjava.AppConfig</param-value>
        </init-param>
        <load-on-startup>0</load-on-startup>
        <async-supported>true</async-supported>
    </servlet>

    <servlet-mapping>
        <servlet-name>dispatcher</servlet-name>
        <url-pattern>/*</url-pattern>
    </servlet-mapping>
</web-app>
```

和前面普通的MVC程序相比，这个 `web.xml` 主要有几点不同：

- 不能再使用 `<!DOCTYPE ...web-app_2_3.dtd>` 的DTD声明，必须用新的支持Servlet 3.1规范的XSD声明，照抄即可；
- 对 `DispatcherServlet` 的配置多了一个 `<async-supported>`，默认值是 `false`，必须明确写成 `true`，这样Servlet容器才会支持async处理。

下一步就是在Controller中编写async处理逻辑。我们以 `ApiController` 为例，演示如何异步处理请求。

第一种async处理方式是返回一个 `Callable`，Spring MVC自动把返回的 `Callable` 放入线程池执行，等待结果返回后再写入响应：

```

@GetMapping("/users")
public Callable<List<User>> users() {
    return () -> {
        // 模拟3秒耗时:
        try {
            Thread.sleep(3000);
        } catch (InterruptedException e) {
        }
        return userService getUsers();
    };
}

```

第二种async处理方式是返回一个 `DeferredResult` 对象，然后在另一个线程中，设置此对象的值并写入响应：

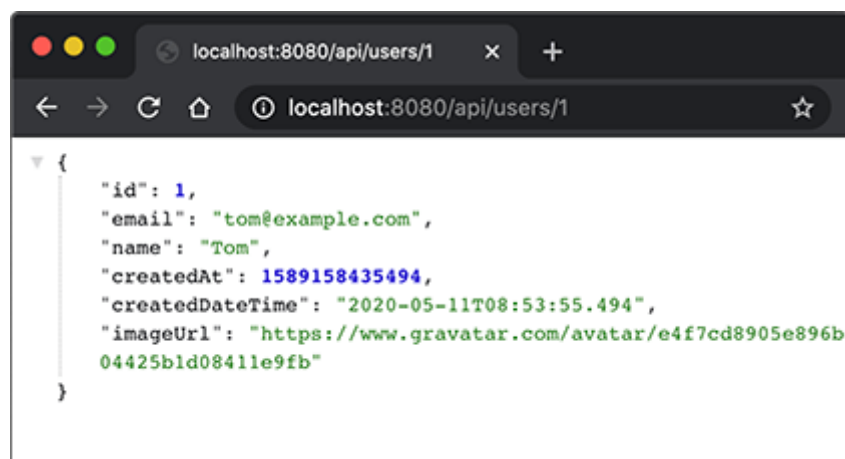
```

@GetMapping("/users/{id}")
public DeferredResult<User> user(@PathVariable("id") long id) {
    DeferredResult<User> result = new DeferredResult<>(3000L); // 3秒超时
    new Thread(() -> {
        // 等待1秒:
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
        }
        try {
            User user = userService.getUserById(id);
            // 设置正常结果并由Spring MVC写入Response:
            result.setResult(user);
        } catch (Exception e) {
            // 设置错误结果并由Spring MVC写入Response:
            result.setErrorResult(Map.of("error", e.getClass().getSimpleName(),
"message", e.getMessage()));
        }
    }).start();
    return result;
}

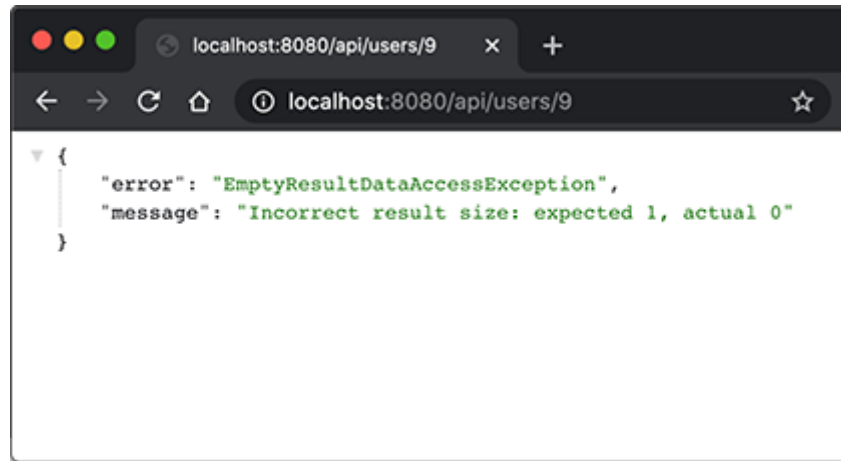
```

使用 `DeferredResult` 时，可以设置超时，超时会自动返回超时错误响应。在另一个线程中，可以调用 `setResult()` 写入结果，也可以调用 `setErrorResult()` 写入一个错误结果。

运行程序，当我们访问 `http://localhost:8080/api/users/1` 时，假定用户存在，则浏览器在1秒后返回结果：



访问一个不存在的User ID，则等待1秒后返回错误结果：



使用Filter

当我们使用async模式处理请求时，原有的Filter也可以工作，但我们必须在 `web.xml` 中添加 `<async-supported>` 并设置为 `true`。我们用两个Filter：SyncFilter和AsyncFilter分别测试：

```
<web-app ...>
  ...
  <filter>
    <filter-name>sync-filter</filter-name>
    <filter-class>com.itranswarp.learnjava.web.SyncFilter</filter-class>
  </filter>

  <filter>
    <filter-name>async-filter</filter-name>
    <filter-class>com.itranswarp.learnjava.web.AsyncFilter</filter-class>
    <async-supported>true</async-supported>
  </filter>

  <filter-mapping>
    <filter-name>sync-filter</filter-name>
    <url-pattern>/api/version</url-pattern>
  </filter-mapping>

  <filter-mapping>
    <filter-name>async-filter</filter-name>
    <url-pattern>/api/*</url-pattern>
  </filter-mapping>
  ...
</web-app>
```

一个声明为支持 `<async-supported>` 的Filter既可以过滤async处理请求，也可以过滤正常的同步处理请求，而未声明 `<async-supported>` 的Filter无法支持async请求，如果一个普通的Filter遇到async请求时，会直接报错，因此，务必注意普通Filter的 `<url-pattern>` 不要匹配async请求路径。

在 `logback.xml` 配置文件中，我们把输出格式加上 `[%thread]`，可以输出当前线程的名称：

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
  <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
    <layout class="ch.qos.logback.classic.PatternLayout">
      <Pattern>%d{yyyy-MM-dd HH:mm:ss} [%thread] %-5level %logger{36} -
%msg%n</Pattern>
    </layout>
  </appender>
  ...
</configuration>
```

对于同步请求，例如 `/api/version`，我们可以看到如下输出：

```
2020-05-16 11:22:40 [http-nio-8080-exec-1] INFO c.i.learnjava.web.SyncFilter -
start SyncFilter...
2020-05-16 11:22:40 [http-nio-8080-exec-1] INFO c.i.learnjava.web.AsyncFilter -
start AsyncFilter...
2020-05-16 11:22:40 [http-nio-8080-exec-1] INFO c.i.learnjava.web.ApiController
- get version...
2020-05-16 11:22:40 [http-nio-8080-exec-1] INFO c.i.learnjava.web.AsyncFilter -
end AsyncFilter.
2020-05-16 11:22:40 [http-nio-8080-exec-1] INFO c.i.learnjava.web.SyncFilter -
end SyncFilter.
```

可见，每个Filter和ApiController都是由同一个线程执行。

对于异步请求，例如 `/api/users`，我们可以看到如下输出：

```
2020-05-16 11:23:49 [http-nio-8080-exec-4] INFO c.i.learnjava.web.AsyncFilter -
start AsyncFilter...
2020-05-16 11:23:49 [http-nio-8080-exec-4] INFO c.i.learnjava.web.ApiController
- get users...
2020-05-16 11:23:49 [http-nio-8080-exec-4] INFO c.i.learnjava.web.AsyncFilter -
end AsyncFilter.
2020-05-16 11:23:52 [MvcAsync1] INFO c.i.learnjava.web.ApiController - return
users...
```

可见，AsyncFilter和ApiController是由同一个线程执行的，但是，返回响应的是另一个线程。

对DeferredResult测试，可以看到如下输出：

```
2020-05-16 11:25:24 [http-nio-8080-exec-8] INFO c.i.learnjava.web.AsyncFilter -
start AsyncFilter...
2020-05-16 11:25:24 [http-nio-8080-exec-8] INFO c.i.learnjava.web.AsyncFilter -
end AsyncFilter.
2020-05-16 11:25:25 [Thread-2] INFO c.i.learnjava.web.ApiController - deferred
result is set.
```

同样，返回响应的是另一个线程。

在实际使用时，经常用到的就是DeferredResult，因为返回DeferredResult时，可以设置超时、正常结果和错误结果，易于编写比较灵活的逻辑。

使用async异步处理响应时，要时刻牢记，在另一个异步线程中的事务和Controller方法中执行的事务不是同一个事务，在Controller中绑定的ThreadLocal信息也无法在异步线程中获取。

此外，Servlet 3.0规范添加的异步支持是针对同步模型打了一个“补丁”，虽然可以异步处理请求，但高并发异步请求时，它的处理效率并不高，因为这种异步模型并没有用到真正的“原生”异步。Java标准库提供了封装操作系统的异步IO包 `java.nio`，是真正的多路复用IO模型，可以用少量线程支持大量并发。使用NIO编程复杂度比同步IO高很多，因此我们很少直接使用NIO。相反，大部分需要高性能异步IO的应用程序会选择[Netty](#)这样的框架，它基于NIO提供了更易于使用的API，方便开发异步应用程序。

练习

从  **gitee** 下载练习：[使用Spring MVC实现异步处理请求](#)（推荐使用[IDE练习插件](#)快速下载）

小结

在Spring MVC中异步处理请求需要正确配置 `web.xml`，并返回 `Callable` 或 `DeferredResult` 对象。

使用WebSocket

WebSocket是一种基于HTTP的长链接技术。传统的HTTP协议是一种请求-响应模型，如果浏览器不发送请求，那么服务器无法主动给浏览器推送数据。如果需要定期给浏览器推送数据，例如股票行情，或者不定期给浏览器推送数据，例如在线聊天，基于HTTP协议实现这类需求，只能依靠浏览器的JavaScript定时轮询，效率很低且实时性不高。

因为HTTP本身是基于TCP连接的，所以，WebSocket在HTTP协议的基础上做了一个简单的升级，即建立TCP连接后，浏览器发送请求时，附带几个头：

```
GET /chat HTTP/1.1
Host: www.example.com
Upgrade: websocket
Connection: Upgrade
```

就表示客户端希望升级连接，变成长连接的WebSocket，服务器返回升级成功的响应：

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
```

收到成功响应后表示WebSocket“握手”成功，这样，代表WebSocket的这个TCP连接将不会被服务器关闭，而是一直保持，服务器可随时向浏览器推送消息，浏览器也可随时向服务器推送消息。双方推送的消息既可以是文本消息，也可以是二进制消息，一般来说，绝大部分应用程序会推送基于JSON的文本消息。

现代浏览器都已经支持WebSocket协议，服务器则需要底层框架支持。Java的Servlet规范从3.1开始支持WebSocket，所以，必须选择支持Servlet 3.1或更高规范的Servlet容器，才能支持WebSocket。最新版本的Tomcat、Jetty等开源服务器均支持WebSocket。

我们以实际代码演示如何在Spring MVC中实现对WebSocket的支持。首先，我们需要在 `pom.xml` 中加入以下依赖：

- `org.apache.tomcat.embed:tomcat-embed-websocket:9.0.26`
- `org.springframework:spring-websocket:5.2.0.RELEASE`

第一项是嵌入式Tomcat支持WebSocket的组件，第二项是Spring封装的支持WebSocket的接口。

接下来，我们需要在AppConfig中加入Spring Web对WebSocket的配置，此处我们需要创建一个 `WebSocketConfigurer` 实例：


```

@Bean
WebSocketConfigurer createWebSocketConfigurer(
    @Autowired ChatHandler chatHandler,
    @Autowired ChatHandshakeInterceptor chatInterceptor)
{
    return new WebSocketConfigurer() {
        public void registerWebSocketHandlers(WebSocketHandlerRegistry registry)
        {
            // 把URL与指定的WebSocketHandler关联，可关联多个：
            registry.addHandler(chatHandler,
                "/chat").addInterceptors(chatInterceptor);
        }
    };
}

```

此实例在内部通过 `WebSocketHandlerRegistry` 注册能处理WebSocket的 `WebSocketHandler`，以及可选的WebSocket拦截器 `HandshakeInterceptor`。我们注入的这两个类都是自己编写的业务逻辑，后面我们详细讨论如何编写它们，这里只需关注浏览器连接到WebSocket的URL是 `/chat`。

处理WebSocket连接

和处理普通HTTP请求不同，没法用一个方法处理一个URL。Spring提供了 `TextWebSocketHandler` 和 `BinaryWebSocketHandler` 分别处理文本消息和二进制消息，这里我们选择文本消息作为聊天室的协议，因此，`ChatHandler` 需要继承自 `TextWebSocketHandler`：

```

@Component
public class ChatHandler extends TextWebSocketHandler {
    ...
}

```

当浏览器请求一个WebSocket连接后，如果成功建立连接，Spring会自动调用 `afterConnectionEstablished()` 方法，任何原因导致WebSocket连接中断时，Spring会自动调用 `afterConnectionClosed` 方法，因此，覆写这两个方法即可处理连接成功和结束后的业务逻辑：

```

@Component
public class ChatHandler extends TextWebSocketHandler {
    // 保存所有Client的WebSocket会话实例：
    private Map<String, WebSocketSession> clients = new ConcurrentHashMap<>();

    @Override
    public void afterConnectionEstablished(WebSocketSession session) throws
    Exception {
        // 新会话根据ID放入Map：
        clients.put(session.getId(), session);
        session.getAttributes().put("name", "Guest1");
    }

    @Override
    public void afterConnectionClosed(WebSocketSession session, CloseStatus
    status) throws Exception {
        clients.remove(session.getId());
    }
}

```

每个WebSocket会话以 `WebSocketSession` 表示，且已分配唯一ID。和WebSocket相关的数据，例如用户名等，均可放入关联的 `getAttributes()` 中。

用实例变量 `clients` 持有当前所有的 `WebSocketSession` 是为了广播，即向所有用户推送同一消息时，可以这么写：

```
String json = ...
TextMessage message = new TextMessage(json);
for (String id : clients.keySet()) {
    WebSocketSession session = clients.get(id);
    session.sendMessage(message);
}
```

我们发送的消息是序列化后的JSON，可以用 `ChatMessage` 表示：

```
public class ChatMessage {
    public long timestamp;
    public String name;
    public String text;
}
```

每收到一个用户的消息后，我们就需要广播给所有用户：

```
@Component
public class ChatHandler extends TextWebSocketHandler {
    ...
    @Override
    protected void handleTextMessage(WebSocketSession session, TextMessage
message) throws Exception {
        String s = message.getPayload();
        String r = ... // 根据输入消息构造待发送消息
        broadcastMessage(r); // 推送给所有用户
    }
}
```

如果要推送给指定的几个用户，那就需要在 `clients` 中根据条件查找出某些 `WebSocketSession`，然后发送消息。

注意到我们在注册WebSocket时还传入了一个 `ChatHandshakeInterceptor`，这个类实际上可以从 `HttpSessionHandshakeInterceptor` 继承，它的主要作用是在WebSocket建立连接后，把 `HttpSession` 的一些属性复制到 `WebSocketSession`，例如，用户的登录信息等：

```
@Component
public class ChatHandshakeInterceptor extends HttpSessionHandshakeInterceptor {
    public ChatHandshakeInterceptor() {
        // 指定从HttpSession复制属性到WebSocketSession:
        super(List.of(UserController.KEY_USER));
    }
}
```

这样，在 `ChatHandler` 中，可以从 `WebSocketSession.getAttributes()` 中获取到复制过来的属性。

客户端开发

在完成了服务器端的开发后，我们还需要在页面编写一点JavaScript逻辑：

```
// 创建WebSocket连接：
var ws = new WebSocket('ws://' + location.host + '/chat');
// 连接成功时：
ws.addEventListener('open', function (event) {
    console.log('websocket connected.');
```

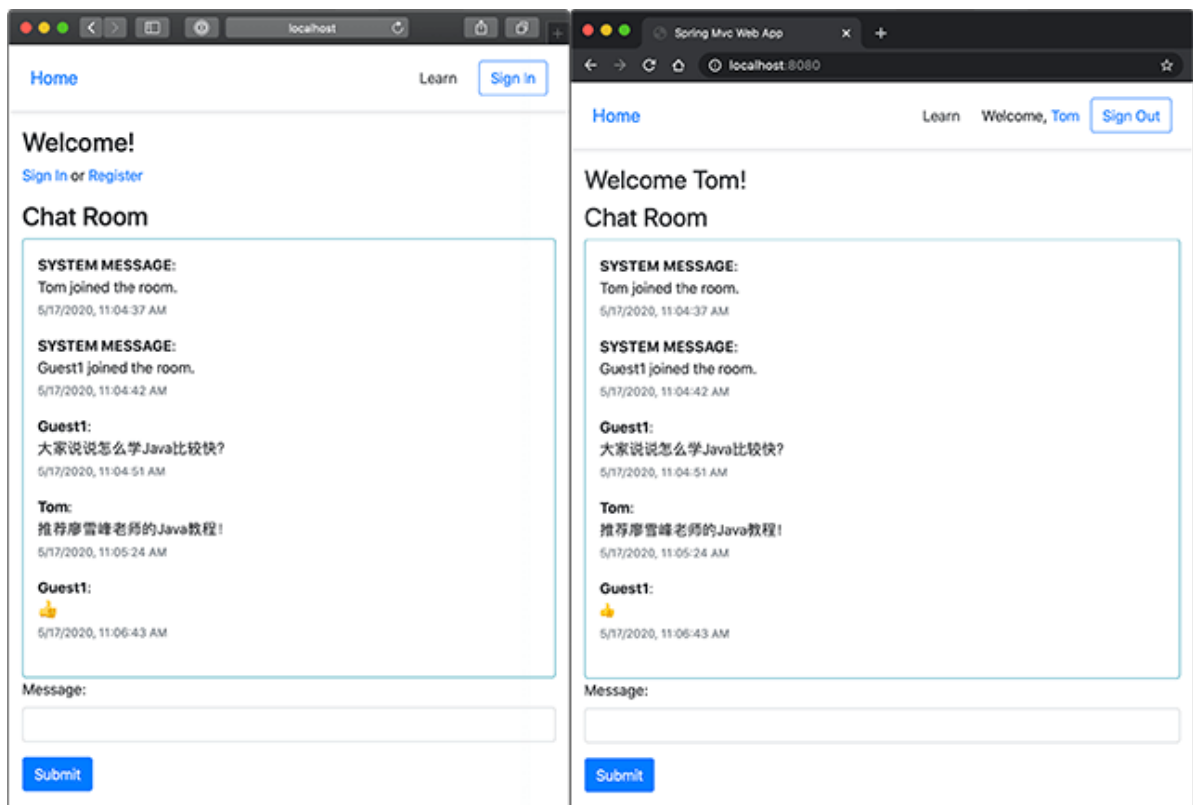
```
});
// 收到消息时：
ws.addEventListener('message', function (event) {
    console.log('message: ' + event.data);
    var msgs = JSON.parse(event.data);
    // TODO:
});
// 连接关闭时：
ws.addEventListener('close', function () {
    console.log('websocket closed.');
```

```
});
// 绑定到全局变量：
window.chatWs = ws;
```

用户可以在连接成功后任何时候给服务器发送消息：

```
var inputText = 'Hello, WebSocket.';
window.chatWs.send(JSON.stringify({text: inputText}));
```

最后，连调浏览器和服务端，如果一切无误，可以开多个不同的浏览器测试WebSocket的推送和广播：



和上一节我们介绍的异步处理类似，Servlet的线程模型并不适合大规模的长链接。基于NIO的Netty等框架更适合处理WebSocket长链接，我们将在后面介绍。

练习

从  **gitee** 下载练习: [使用WebSocket编写一个聊天室](#) (推荐使用[IDE练习插件](#)快速下载)

小结

在Servlet中使用WebSocket需要3.1及以上版本;

通过 `spring-websocket` 可以简化WebSocket的开发。

集成第三方组件

Spring框架不仅提供了标准的IoC容器、AOP支持、数据库访问以及WebMVC等标准功能, 还可以非常方便地集成许多常用的第三方组件:

- 可以集成JavaMail发送邮件;
- 可以集成JMS消息服务;
- 可以集成Quartz实现定时任务;
- 可以集成Redis等服务。

本章我们介绍如何在Spring中简单快捷地集成这些第三方组件。

集成JavaMail

我们在[发送Email](#)和[接收Email](#)中已经介绍了如何通过JavaMail来收发电子邮件。在Spring中, 同样可以集成JavaMail。

因为在服务器端, 主要以发送邮件为主, 例如在注册成功、登录时、购物付款后通知用户, 基本上不会遇到接收用户邮件的情况, 所以本节我们只讨论如何在Spring中发送邮件。

在Spring中, 发送邮件最终也是需要JavaMail, Spring只对JavaMail做了一点简单的封装, 目的是简化代码。为了在Spring中集成JavaMail, 我们在 `pom.xml` 中添加以下依赖:

- `org.springframework:spring-context-support:5.2.0.RELEASE`
- `javax.mail:javax.mail-api:1.6.2`
- `com.sun.mail:javax.mail:1.6.2`

以及其他Web相关依赖。

我们希望用户在注册成功后能收到注册邮件, 为此, 我们先定义一个 `JavaMailSender` 的Bean:

```
@Bean
JavaMailSender createJavaMailSender(
    // smtp.properties:
    @Value("${smtp.host}") String host,
    @Value("${smtp.port}") int port,
    @Value("${smtp.auth}") String auth,
    @Value("${smtp.username}") String username,
    @Value("${smtp.password}") String password,
    @Value("${smtp.debug:true}") String debug
) {
    var mailSender = new JavaMailSenderImpl();
    mailSender.setHost(host);
    mailSender.setPort(port);
    mailSender.setUsername(username);
    mailSender.setPassword(password);
    Properties props = mailSender.getJavaMailProperties();
    props.put("mail.transport.protocol", "smtp");
```

```

        props.put("mail.smtp.auth", auth);
        if (port == 587) {
            props.put("mail.smtp.starttls.enable", "true");
        }
        if (port == 465) {
            props.put("mail.smtp.socketFactory.port", "465");
            props.put("mail.smtp.socketFactory.class",
"javax.net.ssl.SSLSocketFactory");
        }
        props.put("mail.debug", debug);
        return mailSender;
    }
}

```

这个 `JavaMailSender` 接口的实现类是 `JavaMailSenderImpl`，初始化时，传入的参数与 `JavaMail` 是完全一致的。

另外注意到需要注入的属性是从 `smtp.properties` 中读取的，因此，`AppConfig` 导入的不止一个 `.properties` 文件，可以导入多个：

```

@Configuration
@ComponentScan
@EnableWebMvc
@EnableTransactionManagement
@PropertySource({ "classpath:/jdbc.properties", "classpath:/smtp.properties" })
public class AppConfig {
    ...
}

```

下一步是封装一个 `MailService`，并定义 `sendRegistrationMail()` 方法：

```

@Component
public class MailService {
    @Value("${smtp.from}")
    String from;

    @Autowired
    JavaMailSender mailSender;

    public void sendRegistrationMail(User user) {
        try {
            MimeMessage mimeMessage = mailSender.createMimeMessage();
            MimeMessageHelper helper = new MimeMessageHelper(mimeMessage, "utf-8");

            helper.setFrom(from);
            helper.setTo(user.getEmail());
            helper.setSubject("welcome to Java course!");
            String html = String.format("<p>Hi, %s,</p><p>welcome to Java course!</p><p>Sent at %s</p>", user.getName(), LocalDateTime.now());
            helper.setText(html, true);
            mailSender.send(mimeMessage);
        } catch (MessagingException e) {
            throw new RuntimeException(e);
        }
    }
}

```

观察上述代码，`MimeMessage` 是JavaMail的邮件对象，而 `MimeMessageHelper` 是Spring提供的用于简化设置`MimeMessage`的类，比如我们设置HTML邮件就可以直接调用 `setText(String text, boolean html)` 方法，而不必再调用比较繁琐的JavaMail接口方法。

最后一步是调用 `JavaMailSender.send()` 方法把邮件发送出去。

在MVC的某个Controller方法中，当用户注册成功后，我们就启动一个新线程来异步发送邮件：

```
User user = userService.register(email, password, name);
logger.info("user registered: {}", user.getEmail());
// send registration mail:
new Thread(() -> {
    mailService.sendRegistrationMail(user);
}).start();
```

因为发送邮件是一种耗时的任务，从几秒到几分钟不等，因此，异步发送是保证页面能快速显示的必要措施。这里我们直接启动了一个新的线程，但实际上还有更优化的方法，我们在下一节讨论。

练习

从  **gitee** 下载练习：[使用Spring发送邮件](#)（推荐使用[IDE练习插件](#)快速下载）

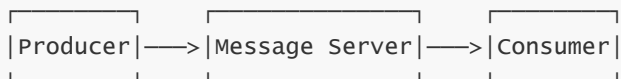
小结

Spring可以集成JavaMail，通过简单的封装，能简化邮件发送代码。其核心是定义一个 `JavaMailSender` 的Bean，然后调用其 `send()` 方法。

集成JMS

JMS即Java Message Service，是JavaEE的消息服务接口。JMS主要有两个版本：1.1和2.0。2.0和1.1相比，主要是简化了收发消息的代码。

所谓消息服务，就是两个进程之间，通过消息服务器传递消息：



使用消息服务，而不是直接调用对方的API，它的好处是：

- 双方各自无需知晓对方的存在，消息可以异步处理，因为消息服务器会在Consumer离线的时候自动缓存消息；
- 如果Producer发送的消息频率高于Consumer的处理能力，消息可以积压在消息服务器，不至于压垮Consumer；
- 通过一个消息服务器，可以连接多个Producer和多个Consumer。

因为消息服务在各类应用程序中非常有用，所以JavaEE专门定义了JMS规范。注意到JMS是一组接口定义，如果我们要使用JMS，还需要选择一个具体的JMS产品。常用的JMS服务器有开源的[ActiveMQ](#)，商业服务器如WebLogic、WebSphere等也内置了JMS支持。这里我们选择开源的ActiveMQ作为JMS服务器，因此，在开发JMS之前我们必须首先安装ActiveMQ。

现在问题来了：从官网下载ActiveMQ时，蹦出一个页面，让我们选择ActiveMQ Classic或者ActiveMQ Artemis，这两个是什么关系，又有什么区别？

实际上ActiveMQ Classic原来就叫ActiveMQ，是Apache开发的基于JMS 1.1的消息服务器，目前稳定版本号是5.x，而ActiveMQ Artemis是由RedHat捐赠的[HornetQ](#)服务器代码的基础上开发的，目前稳定版本号是2.x。和ActiveMQ Classic相比，Artemis版的代码与Classic完全不同，并且，它支持JMS 2.0，使用基于Netty的异步IO，大大提升了性能。此外，Artemis不仅提供了JMS接口，它还提供了AMQP接口，STOMP接口和物联网使用的MQTT接口。选择Artemis，相当于一鱼四吃。

所以，我们这里直接选择ActiveMQ Artemis。从官网[下载](#)最新的2.x版本，解压后设置环境变量 `ARTEMIS_HOME`，指向Artemis根目录，例如 `C:\Apps\artemis`，然后，把 `ARTEMIS_HOME/bin` 加入 `PATH` 环境变量：

- Windows下添加 `%ARTEMIS_HOME%\bin` 到Path路径；
- Mac和Linux下添加 `$ARTEMIS_HOME/bin` 到PATH路径。

Artemis有个很好的设计，就是它把程序和数据完全分离了。我们解压后的 `ARTEMIS_HOME` 目录是程序目录，要启动一个Artemis服务，还需要创建一个数据目录。我们把数据目录直接设定在项目 `spring-integration-jms` 的 `jms-data` 目录下。执行命令 `artemis create jms-data`：

```
$ pwd
/Users/liaoxuefeng/workspace/spring-integration-jms

$ artemis create jms-data
Creating ActiveMQ Artemis instance at: /Users/liaoxuefeng/workspace/spring-integration-jms/jms-data

--user: is a mandatory property!
Please provide the default username:
admin

--password: is mandatory with this configuration:
Please provide the default password:
*****

--allow-anonymous | --require-login: is a mandatory property!
Allow anonymous access?, valid values are Y,N,True,False
N

Auto tuning journal ...
done! Your system can make 0.09 writes per millisecond, your journal-buffer-timeout will be 11392000

You can now start the broker by executing:

"/Users/liaoxuefeng/workspace/spring-integration-jms/jms-data/bin/artemis"
run

Or you can run the broker in the background using:

"/Users/liaoxuefeng/workspace/spring-integration-jms/jms-data/bin/artemis-service" start
```

在创建过程中，会要求输入连接用户和口令，这里我们设定 `admin` 和 `password`，以及是否允许匿名访问（这里选择 `N`）。

此数据目录 `jms-data` 不仅包含消息数据、日志，还自动创建了两个启动服务的命令 `bin/artemis` 和 `bin/artemis-service`，前者在前台启动运行，按Ctrl+C结束，后者会一直在后台运行。

我们把目录切换到 `jms-data/bin`，直接运行 `artemis run` 即可启动Artemis服务：

```
$ ./artemis run
```

```
  _      _      _  
 / \    _ _ | | _ _ _ _ _ ( ) _ _ _  
/_ _ \ | _ \ _ _ |/_ _ \ \ / | |/_ _/  
/_ _ \ | \ / |/_ _ / | \ | | | \ _ _ \  
/_ /   \ \ |   \ _ \ _ _ | | | | |/_ _ /  
Apache ActiveMQ Artemis 2.13.0
```

...

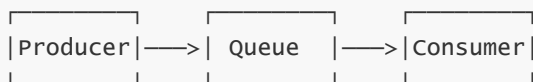
```
2020-06-02 07:50:21,718 INFO [org.apache.activemq.artemis] AMQ241001: HTTP  
Server started at http://localhost:8161
```

```
2020-06-02 07:50:21,718 INFO [org.apache.activemq.artemis] AMQ241002: Artemis  
Jolokia REST API available at http://localhost:8161/console/jolokia
```

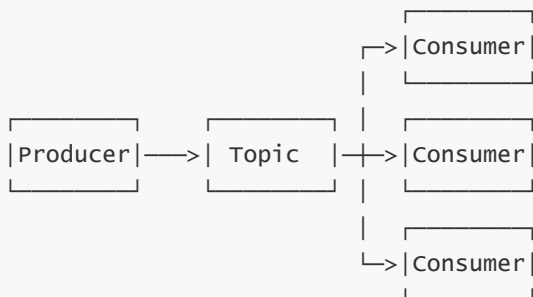
```
2020-06-02 07:50:21,719 INFO [org.apache.activemq.artemis] AMQ241004: Artemis  
Console available at http://localhost:8161/console
```

启动成功后，Artemis提示可以通过URL <http://localhost:8161/console> 访问管理后台。注意不要关闭命令行窗口。

在编写JMS代码之前，我们首先得理解JMS的消息模型。JMS把生产消息的一方称为Producer，处理消息的一方称为Consumer。有两种类型的消息通道，一种是Queue：



一种是Topic：



它们的区别在于，Queue是一种一对一的通道，如果Consumer离线无法处理消息时，Queue会把消息存起来，等Consumer再次连接的时候发给它。设定了持久化机制的Queue不会丢失消息。如果有多个Consumer接入同一个Queue，那么它们等效于以集群方式处理消息，例如，发送方发送的消息是A，B，C，D，E，F，两个Consumer可能分别收到A，C，E和B，D，F，即每个消息只会交给其中一个Consumer处理。

Topic则是一种一对多通道。一个Producer发出的消息，会被多个Consumer同时收到，即每个Consumer都会收到一份完整的消息流。那么问题来了：如果某个Consumer暂时离线，过一段时间后又上线了，那么在它离线期间产生的消息还能不能收到呢？

这取决于消息服务器对Topic类型消息的持久化机制。如果消息服务器不存储Topic消息，那么离线的Consumer会丢失部分离线时期的消息，如果消息服务器存储了Topic消息，那么离线的Consumer可以收到自上次离线时刻开始后产生的所有消息。JMS规范通过Consumer指定一个持久化订阅可以在上线后收取所有离线期间的消息，如果指定的是非持久化订阅，那么离线期间的消息会全部丢失。

细心的童鞋可以看出来，如果一个Topic的消息全部都持久化了，并且只有一个Consumer，那么它和Queue其实是一样的。实际上，很多消息服务器内部都只有Topic类型的消息架构，Queue可以通过Topic“模拟”出来。

无论是Queue还是Topic，对Producer没有什么要求。多个Producer也可以写入同一个Queue或者Topic，此时消息服务器内部会自动排序确保消息总是有序的。

以上是消息服务的基本模型。具体到某个消息服务器时，Producer和Consumer通常是通过TCP连接消息服务器，在编写JMS程序时，又会遇到 `ConnectionFactory`、`Connection`、`Session` 等概念，其实这和JDBC连接是类似的：

- `ConnectionFactory`：代表一个到消息服务器的连接池，类似JDBC的`DataSource`；
- `Connection`：代表一个到消息服务器的连接，类似JDBC的`Connection`；
- `Session`：代表一个经过认证后的连接会话；
- `Message`：代表一个消息对象。

在JMS 1.1中，发送消息的典型代码如下：

```
try {
    Connection connection = null;
    try {
        // 创建连接：
        connection = connectionFactory.createConnection();
        // 创建会话：
        Session session =
connection.createSession(false,Session.AUTO_ACKNOWLEDGE);
        // 创建一个Producer并关联到某个Queue：
        MessageProducer messageProducer = session.createProducer(queue);
        // 创建一个文本消息：
        TextMessage textMessage = session.createTextMessage(text);
        // 发送消息：
        messageProducer.send(textMessage);
    } finally {
        // 关闭连接：
        if (connection != null) {
            connection.close();
        }
    }
} catch (JMSException ex) {
    // 处理JMS异常
}
```

JMS 2.0改进了一些API接口，发送消息变得更简单：

```
try (JMSContext context = connectionFactory.createContext()) {
    context.createProducer().send(queue, text);
}
```

`JMSContext` 实现了 `AutoCloseable` 接口，可以使用 `try(resource)` 语法，代码更简单。

有了以上预备知识，我们就可以开始开发JMS应用了。

首先，我们在 `pom.xml` 中添加如下依赖：

- `org.springframework:spring-jms:5.2.0.RELEASE`
- `javax.jms:javax.jms-api:2.0.1`
- `org.apache.activemq:artemis-jms-client:2.13.0`

- io.netty:netty-handler-proxy:4.1.45.Final

在AppConfig中, 通过 @EnableJms 让Spring自动扫描JMS相关的Bean, 并加载JMS配置文件

jms.properties:

```
@Configuration
@ComponentScan
@EnableWebMvc
@EnableJms // 启用JMS
@EnableTransactionManagement
@PropertySource({ "classpath:/jdbc.properties", "classpath:/jms.properties" })
public class AppConfig {
    ...
}
```

首先要创建的Bean是ConnectionFactory, 即连接消息服务器的连接池:

```
@Bean
ConnectionFactory createJMSConnectionFactory(
    @Value("${jms.uri:tcp://localhost:61616}") String uri,
    @Value("${jms.username:admin}") String username,
    @Value("${jms.password:password}") String password)
{
    return new ActiveMQJMSConnectionFactory(uri, username, password);
}
```

因为我们使用的消息服务器是ActiveMQ Artemis, 所以 ConnectionFactory 的实现类就是消息服务器提供的 ActiveMQJMSConnectionFactory, 它需要的参数均由配置文件读取后传入, 并设置了默认值。

我们再创建一个 JmsTemplate, 它是Spring提供的一个工具类, 和 JdbcTemplate 类似, 可以简化发送消息的代码:

```
@Bean
JmsTemplate createJmsTemplate(@Autowired ConnectionFactory connectionFactory) {
    return new JmsTemplate(connectionFactory);
}
```

下一步要创建的是 JmsListenerContainerFactory,

```
@Bean("jmsListenerContainerFactory")
DefaultJmsListenerContainerFactory createJmsListenerContainerFactory(@Autowired
ConnectionFactory connectionFactory) {
    var factory = new DefaultJmsListenerContainerFactory();
    factory.setConnectionFactory(connectionFactory);
    return factory;
}
```

除了必须指定Bean的名称为 jmsListenerContainerFactory 外, 这个Bean的作用是处理和 Consumer相关的Bean。我们先跳过它的原理, 继续编写 MessagingService 来发送消息:

```
@Component
public class MessagingService {
    @Autowired ObjectMapper objectMapper;
```

```

@Autowired JmsTemplate jmsTemplate;

public void sendMailMessage(MailMessage msg) throws Exception {
    String text = objectMapper.writeValueAsString(msg);
    jmsTemplate.send("jms/queue/mail", new MessageCreator() {
        public Message createMessage(Session session) throws JMSException {
            return session.createTextMessage(text);
        }
    });
}
}

```

JMS的消息类型支持以下几种：

- TextMessage：文本消息；
- BytesMessage：二进制消息；
- MapMessage：包含多个Key-Value对的消息；
- ObjectMessage：直接序列化Java对象的消息；
- StreamMessage：一个包含基本类型序列的消息。

最常用的是发送基于JSON的文本消息，上述代码通过 `JmsTemplate` 创建一个 `TextMessage` 并发送到名称为 `jms/queue/mail` 的Queue。

注意：Artemis消息服务器默认配置下会自动创建Queue，因此不必手动创建一个名为 `jms/queue/mail` 的Queue，但不是所有的消息服务器都会自动创建Queue，生产环境的消息服务器通常会关闭自动创建功能，需要手动创建Queue。

再注意到 `MailMessage` 是我们自己定义的一个JavaBean，真正的JMS消息是创建的 `TextMessage`，它的内容是JSON。

当用户注册成功后，我们就调用 `MessagingService.sendMailMessage()` 发送一条JMS消息，此代码十分简单，这里不再贴出。

下面我们要详细讨论的是如何处理消息，即编写Consumer。从理论上讲，可以创建另一个Java进程来处理消息，但对于我们这个简单的Web程序来说没有必要，直接在同一个Web应用中接收并处理消息即可。

处理消息的核心代码是编写一个Bean，并在处理方法上标注 `@JmsListener`：

```

@Component
public class MailMessageListener {
    final Logger logger = LoggerFactory.getLogger(getClass());

    @Autowired ObjectMapper objectMapper;
    @Autowired MailService mailService;

    @JmsListener(destination = "jms/queue/mail", concurrency = "10")
    public void onMailMessageReceived(Message message) throws Exception {
        logger.info("received message: " + message);
        if (message instanceof TextMessage) {
            String text = ((TextMessage) message).getText();
            MailMessage mm = objectMapper.readValue(text, MailMessage.class);
            mailService.sendRegistrationMail(mm);
        } else {
            logger.error("unable to process non-text message!");
        }
    }
}

```

注意到 `@JmsListener` 指定了 Queue 的名称，因此，凡是发到此 Queue 的消息都会被这个 `onMailMessageReceived()` 方法处理，方法参数是 JMS 的 `Message` 接口，我们通过强制转型为 `TextMessage` 并提取 JSON，反序列化后获得自定义的 `JavaBean`，也就获得了发送邮件所需的所有信息。

下面问题来了：Spring 处理 JMS 消息的流程是什么？

如果我们直接调用 JMS 的 API 来处理消息，那么编写的代码大致如下：

```
// 创建JMS连接：
Connection connection = connectionFactory.createConnection();
// 创建会话：
Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
// 创建一个Consumer：
MessageConsumer consumer = session.createConsumer(queue);
// 为Consumer指定一个消息处理器：
consumer.setMessageListener(new MessageListener() {
    public void onMessage(Message message) {
        // 在此处理消息...
    }
});
// 启动接收消息的循环：
connection.start();
```

我们自己编写的 `MailMessageListener.onMailMessageReceived()` 相当于消息处理器：

```
consumer.setMessageListener(new MessageListener() {
    public void onMessage(Message message) {
        mailMessageListener.onMailMessageReceived(message);
    }
});
```

所以，Spring 根据 `AppConfig` 的注解 `@EnableJms` 自动扫描带有 `@JmsListener` 的 Bean 方法，并为其创建一个 `MessageListener` 把它包装起来。

注意到前面我们还创建了一个 `JmsListenerContainerFactory` 的 Bean，它的作用就是为每个 `MessageListener` 创建 `MessageConsumer` 并启动消息接收循环。

再注意到 `@JmsListener` 还有一个 `concurrency` 参数，10 表示可以最多同时并发处理 10 个消息，5-10 表示并发处理的线程可以在 5~10 之间调整。

因此，Spring 在通过 `MessageListener` 接收到消息后，并不是直接调用 `mailMessageListener.onMailMessageReceived()`，而是用线程池调用，因此，要时刻牢记，`onMailMessageReceived()` 方法可能被多线程并发执行，一定要保证线程安全。

我们总结一下 Spring 接收消息的步骤：

通过 `JmsListenerContainerFactory` 配合 `@EnableJms` 扫描所有 `@JmsListener` 方法，自动创建 `MessageConsumer`、`MessageListener` 以及线程池，启动消息循环接收处理消息，最终由我们自己编写的 `@JmsListener` 方法处理消息，可能会由多线程同时并发处理。

要验证消息发送和处理，我们注册一个新用户，可以看到如下日志输出：

```
2020-06-02 08:04:27 INFO c.i.learnjava.web.UserController - user registered: bob@example.com
2020-06-02 08:04:27 INFO c.i.l.service.MailMessageListener - received message: ActiveMQMessage[ID:9fc5...]:PERSISTENT/ClientMessageImpl[messageID=983, durable=true, address=jms/queue/mail, ...]]
2020-06-02 08:04:27 INFO c.i.learnjava.service.MailService - [send mail] sending registration mail to bob@example.com...
2020-06-02 08:04:30 INFO c.i.learnjava.service.MailService - [send mail] registration mail was sent to bob@example.com.
```

可见，消息被成功发送到Artemis，然后在很短的时间内被接收处理了。

使用消息服务对发送Email进行改造的好处是，发送Email的能力通常是有限的，通过JMS消息服务，如果短时间内需要给大量用户发送Email，可以先把消息堆积在JMS服务器上慢慢发送，对于批量发送邮件、短信等尤其有用。

练习

从  **gitee** 下载练习：[使用JMS](#)（推荐使用[IDE练习插件](#)快速下载）

小结

JMS是Java消息服务，可以通过JMS服务器实现消息的异步处理。

消息服务主要解决Producer和Consumer生产和处理速度不匹配的问题。

使用Scheduler

在很多应用程序中，经常需要执行定时任务。例如，每天或每月给用户发送账户汇总报表，定期检查并发送系统状态报告，等等。

定时任务我们在[使用线程池](#)一节中已经讲到了，Java标准库本身就提供了定时执行任务的功能。在Spring中，使用定时任务更简单，不需要手写线程池相关代码，只需要两个注解即可。

我们还是以实际代码为例，建立工程 `spring-integration-schedule`，无需额外的依赖，我们可以直接在 `AppConfig` 中加上 `@EnableScheduling` 就开启了定时任务的支持：

```
@Configuration
@ComponentScan
@EnableWebMvc
@EnableScheduling
@EnableTransactionManagement
@PropertySource({ "classpath:/jdbc.properties", "classpath:/task.properties" })
public class AppConfig {
    ...
}
```

接下来，我们可以直接在一个Bean中编写一个 `public void` 无参数方法，然后加上 `@scheduled` 注解：

```

@Component
public class TaskService {
    final Logger logger = LoggerFactory.getLogger(getClass());

    @Scheduled(initialDelay = 60_000, fixedRate = 60_000)
    public void checkSystemStatusEveryMinute() {
        logger.info("Start check system status...");
    }
}

```

上述注解指定了启动延迟60秒，并以60秒的间隔执行任务。现在，我们直接运行应用程序，就可以在控制台看到定时任务打印的日志：

```

2020-06-03 18:47:32 INFO [pool-1-thread-1] c.i.learnjava.service.TaskService -
Start check system status...
2020-06-03 18:48:32 INFO [pool-1-thread-1] c.i.learnjava.service.TaskService -
Start check system status...
2020-06-03 18:49:32 INFO [pool-1-thread-1] c.i.learnjava.service.TaskService -
Start check system status...

```

如果没有看到定时任务的日志，需要检查：

- 是否忘记了在 `AppConfig` 中标注 `@EnableScheduling`；
- 是否忘记了在定时任务的方法所在的class标注 `@Component`。

除了可以使用 `fixedRate` 外，还可以使用 `fixedDelay`，两者的区别我们已经在[使用线程池](#)一节中讲过，这里不再重复。

有的童鞋在实际开发中会遇到一个问题，因为Java的注解全部是常量，写死了 `fixedDelay=30000`，如果根据实际情况要改成60秒怎么办，只能重新编译？

我们可以把定时任务的配置放到配置文件中，例如 `task.properties`：

```
task.checkDiskSpace=30000
```

这样就可以随时修改配置文件而无需动代码。但是在代码中，我们需要用 `fixedDelayString` 取代 `fixedDelay`：

```

@Component
public class TaskService {
    ...

    @Scheduled(initialDelay = 30_000, fixedDelayString =
"${task.checkDiskSpace:30000}")
    public void checkDiskSpaceEveryMinute() {
        logger.info("Start check disk space...");
    }
}

```

注意到上述代码的注解参数 `fixedDelayString` 是一个属性占位符，并配有默认值30000，Spring在处理 `@Scheduled` 注解时，如果遇到 `String`，会根据占位符自动用配置项替换，这样就可以灵活地修改定时任务的配置。

此外，`fixedDelayString` 还可以使用更易读的 `Duration`，例如：

```
@Scheduled(initialDelay = 30_000, fixedDelayString =
"${task.checkDiskSpace:PT2M30S}")
```

以字符串 `PT2M30S` 表示的 `Duration` 就是2分30秒，请参考[LocalDateTime](#)一节的Duration相关部分。

多个 `@Scheduled` 方法完全可以放到一个Bean中，这样便于统一管理各类定时任务。

使用Cron任务

还有一类定时任务，它不是简单的重复执行，而是按时间触发，我们把这类任务称为Cron任务，例如：

- 每天凌晨2:15执行报表任务；
- 每个工作日12:00执行特定任务；
-

Cron源自Unix/Linux系统自带的crond守护进程，以一个简洁的表达式定义任务触发时间。在Spring中，也可以使用Cron表达式来执行Cron任务，在Spring中，它的格式是：

秒 分 小时 天 月份 星期 年

年是可以忽略的，通常不写。每天凌晨2:15执行的Cron表达式就是：

```
0 15 2 * * *
```

每个工作日12:00执行的Cron表达式就是：

```
0 0 12 * * MON-FRI
```

每个月1号，2号，3号和10号12:00执行的Cron表达式就是：

```
0 0 12 1-3,10 * *
```

在Spring中，我们定义一个每天凌晨2:15执行的任务：

```
@Component
public class TaskService {
    ...

    @Scheduled(cron = "${task.report:0 15 2 * * *}")
    public void cronDailyReport() {
        logger.info("Start daily report task...");
    }
}
```

Cron任务同样可以使用属性占位符，这样修改起来更加方便。

Cron表达式还可以表达每10分钟执行，例如：

```
0 */10 * * * *
```

这样，在每个小时的0:00，10:00，20:00，30:00，40:00，50:00均会执行任务，实际上它可以取代 `fixedRate` 类型的定时任务。

集成Quartz

在Spring中使用定时任务和Cron任务都十分简单，但是要注意到，这些任务的调度都是在每个JVM进程中的。如果在本机启动两个进程，或者在多台机器上启动应用，这些进程的定时任务和Cron任务都是独立运行的，互不影响。


如果一些定时任务要以集群的方式运行，例如每天23:00执行检查任务，只需要集群中的一台运行即可，这个时候，可以考虑使用[Quartz](#)。

Quartz可以配置一个JDBC数据源，以便存储所有的任务调度计划以及任务执行状态。也可以使用内存来调度任务，但这样配置就和使用Spring的调度没啥区别了，额外集成Quartz的意义就不大。

Quartz的JDBC配置比较复杂，Spring对其也有一定的支持。要详细了解Quartz的集成，请参考[Spring的文档](#)。

思考：如果不使用Quartz的JDBC配置，多个Spring应用同时运行时，如何保证某个任务只在某一台机器执行？

练习

从  **gitee** 下载练习：[使用Scheduler](#)（推荐使用[IDE练习插件](#)快速下载）

小结

Spring内置定时任务和Cron任务的支持，编写调度任务十分方便。

集成JMX

在Spring中，可以方便地集成JMX。

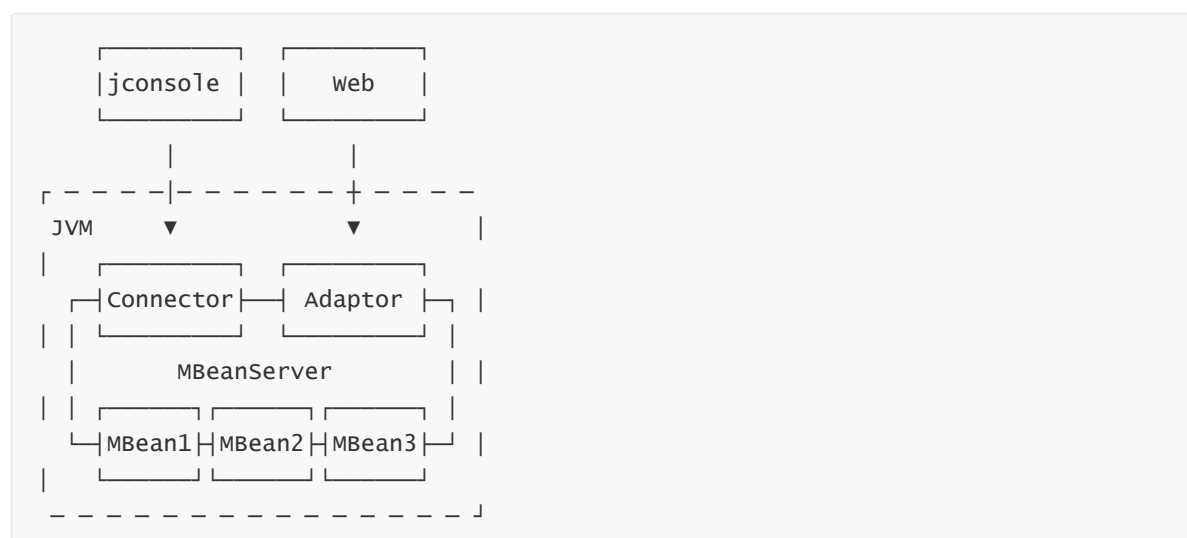
那么第一个问题来了：什么是JMX？

JMX是Java Management Extensions，它是一个Java平台的管理和监控接口。为什么要搞JMX呢？因为在所有的应用程序中，对运行中的程序进行监控都是非常重要的，Java应用程序也不例外。我们肯定希望知道Java应用程序当前的状态，例如，占用了多少内存，分配了多少内存，当前有多少活动线程，有多少休眠线程等等。如何获取这些信息呢？

为了标准化管理和监控，Java平台使用JMX作为管理和监控的标准接口，任何程序，只要按JMX规范访问这个接口，就可以获取所有管理与监控信息。

实际上，常用的运维监控如Zabbix、Nagios等工具对JVM本身的监控都是通过JMX获取的信息。

因为JMX是一个标准接口，不但可以用于管理JVM，还可以管理应用程序自身。下图是JMX的架构：



JMX把所有被管理的资源都称为MBean (Managed Bean)，这些MBean全部由MBeanServer管理，如果要访问MBean，可以通过MBeanServer对外提供的访问接口，例如通过RMI或HTTP访问。

注意到使用JMX不需要安装任何额外组件，也不需要第三方库，因为MBeanServer已经内置在JavaSE标准库中了。JavaSE还提供了一个 `jconsole` 程序，用于通过RMI连接到MBeanServer，这样就可以管理整个Java进程。

除了JVM会把自身的各种资源以MBean注册到JMX中，我们自己的配置、监控信息也可以作为MBean注册到JMX，这样，管理程序就可以直接控制我们暴露的MBean。因此，应用程序使用JMX，只需要两步：

1. 编写MBean提供管理接口和监控数据；
2. 注册MBean。

在Spring应用程序中，使用JMX只需要一步：

1. 编写MBean提供管理接口和监控数据。

第二步注册的过程由Spring自动完成。我们以实际工程为例，首先在 `AppConfig` 中加上 `@EnableMBeanExport` 注解，告诉Spring自动注册MBean：

```
@Configuration
@ComponentScan
@EnableWebMvc
@EnableMBeanExport // 自动注册MBean
@EnableTransactionManagement
@PropertySource({ "classpath:/jdbc.properties" })
public class AppConfig {
    ...
}
```

剩下的全部工作就是编写MBean。我们以实际问题为例，假设我们希望给应用程序添加一个IP黑名单功能，凡是在黑名单中的IP禁止访问，传统的做法是定义一个配置文件，启动的时候读取：

```
# blacklist.txt
1.2.3.4
5.6.7.8
2.2.3.4
...
```

如果要修改黑名单怎么办？修改配置文件，然后重启应用程序。

但是每次都重启应用程序实在是太麻烦了，能不能不重启应用程序？可以自己写一个定时读取配置文件的函数，检测到文件改动时自动重新读取。

上述需求本质上是在应用程序运行期间对参数、配置等进行热更新并要求尽快生效。如果以JMX的方式实现，我们不必自己编写自动重新读取等任何代码，只需要提供一个符合JMX标准的MBean来存储配置即可。

还是以IP黑名单为例，JMX的MBean通常以MBean结尾，因此我们遵循标准命名规范，首先编写一个 `BlacklistMBean`：

```
public class BlacklistMBean {
    private Set<String> ips = new HashSet<>();

    public String[] getBlacklist() {
        return ips.toArray(String[]::new);
    }
}
```

```

    }

    public void addBlacklist(String ip) {
        ips.add(ip);
    }

    public void removeBlacklist(String ip) {
        ips.remove(ip);
    }

    public boolean shouldBlock(String ip) {
        return ips.contains(ip);
    }
}

```

这个MBean没什么特殊的，它的逻辑和普通Java类没有任何区别。

下一步，我们要使用JMX的客户端来实时热更新这个MBean，所以要给它加上一些注解，让Spring能根据注解自动把相关方法注册到MBeanServer中：

```

@Component
@ManagedResource(objectName = "sample:name=blacklist", description = "Blacklist
of IP addresses")
public class BlacklistMBean {
    private Set<String> ips = new HashSet<>();

    @ManagedAttribute(description = "Get IP addresses in blacklist")
    public String[] getBlacklist() {
        return ips.toArray(String[]::new);
    }

    @ManagedOperation
    @ManagedOperationParameter(name = "ip", description = "Target IP address that
will be added to blacklist")
    public void addBlacklist(String ip) {
        ips.add(ip);
    }

    @ManagedOperation
    @ManagedOperationParameter(name = "ip", description = "Target IP address that
will be removed from blacklist")
    public void removeBlacklist(String ip) {
        ips.remove(ip);
    }

    public boolean shouldBlock(String ip) {
        return ips.contains(ip);
    }
}

```

观察上述代码，`BlacklistMBean` 首先是一个标准的Spring管理的Bean，其次，添加了`@ManagedResource` 表示这是一个MBean，将要被注册到JMX。objectName指定了这个MBean的名字，通常以 `company:name=xxx` 来分类MBean。

对于属性，使用 `@ManagedAttribute` 注解标注。上述MBean只有get属性，没有set属性，说明这是一个只读属性。

对于操作，使用 `@ManagedOperation` 注解标准。上述MBean定义了两个操作：`addBlacklist()` 和 `removeBlacklist()`，其他方法如 `shouldBlock()` 不会被暴露给JMX。

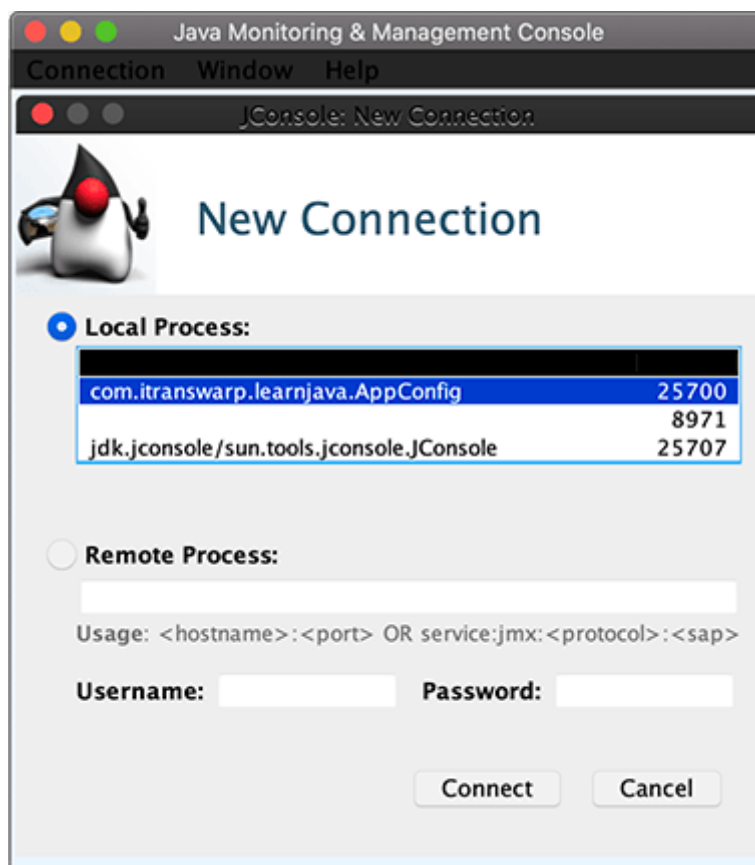
使用MBean和普通Bean是完全一样的。例如，我们在 `BlacklistInterceptor` 对IP进行黑名单拦截：

```
@Order(1)
@Component
public class BlacklistInterceptor implements HandlerInterceptor {
    final Logger logger = LoggerFactory.getLogger(getClass());

    @Autowired
    BlacklistMBean blacklistMBean;

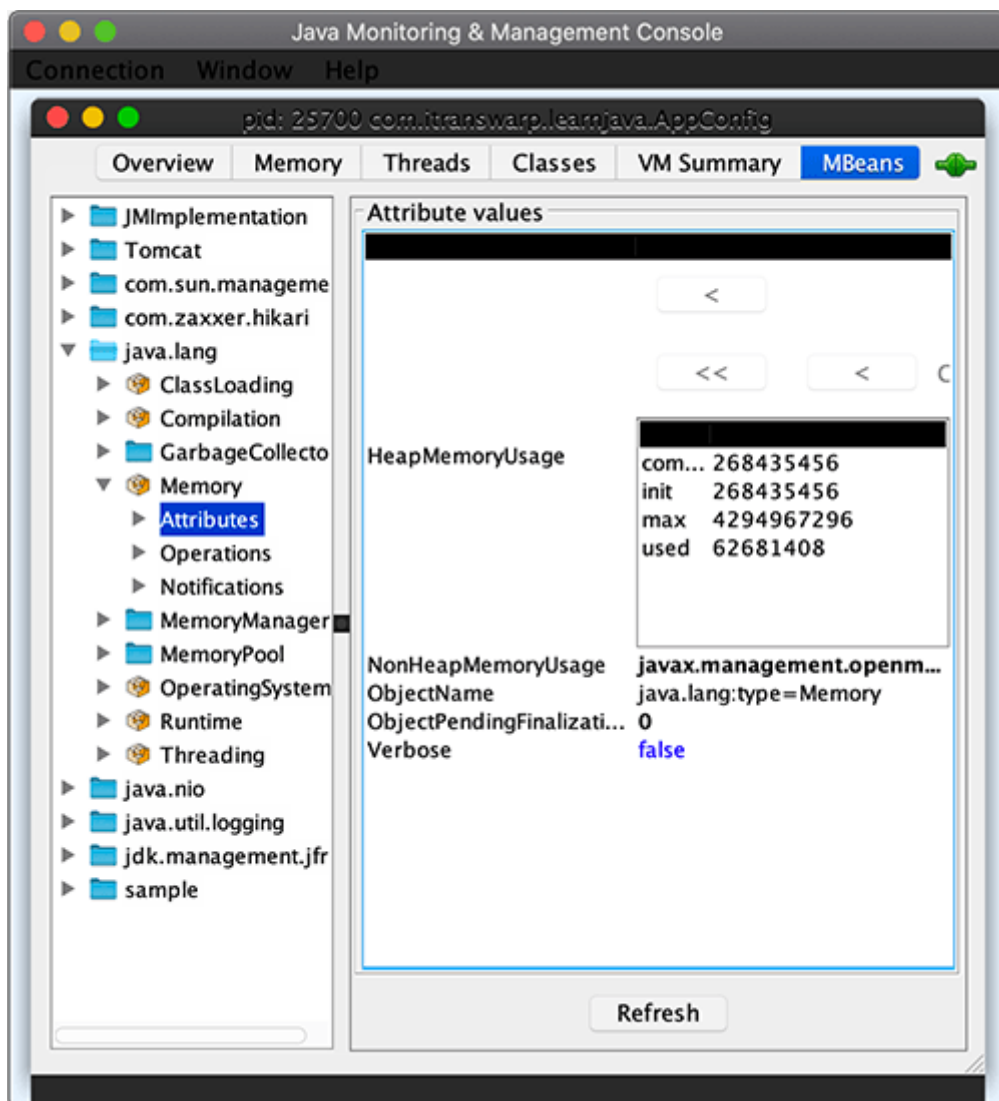
    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse
response, Object handler)
        throws Exception {
        String ip = request.getRemoteAddr();
        logger.info("check ip address {}...", ip);
        // 是否在黑名单中：
        if (blacklistMBean.shouldBlock(ip)) {
            logger.warn("will block ip {} for it is in blacklist.", ip);
            // 发送403错误响应：
            response.sendError(403);
            return false;
        }
        return true;
    }
}
```

下一步就是正常启动Web应用程序，不要关闭它，我们打开另一个命令行窗口，输入 `jconsole` 启动JavaSE自带的一个JMX客户端程序：



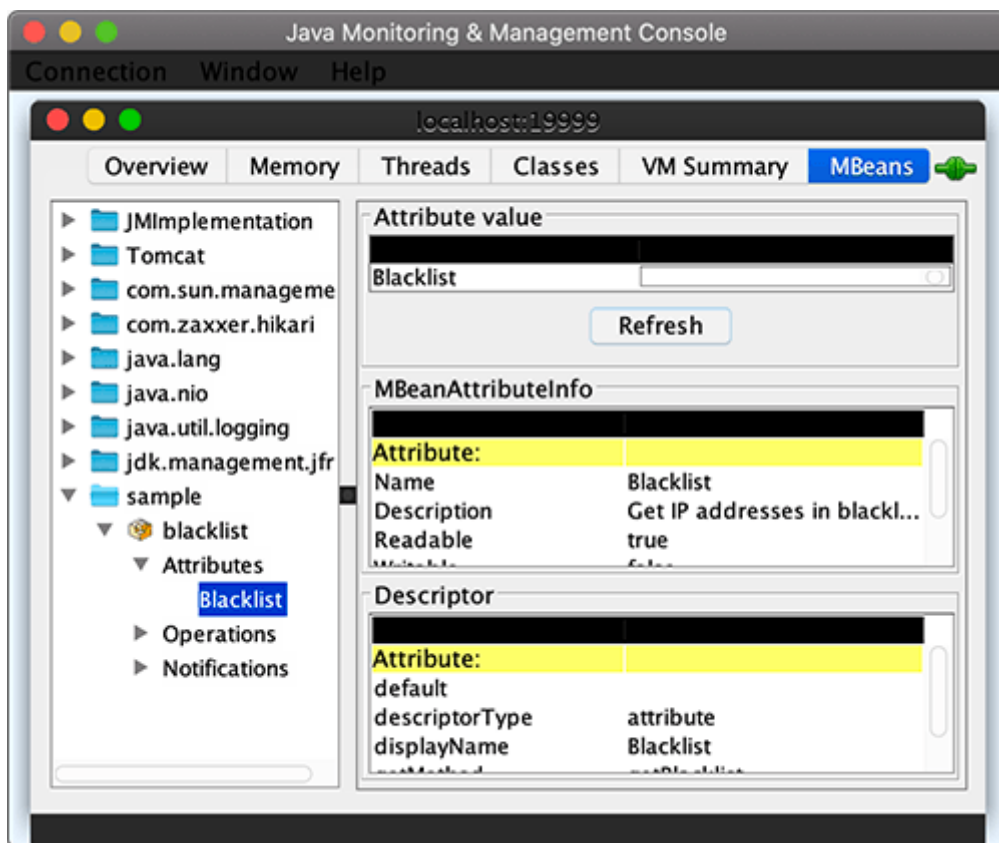
通过jconsole连接到一个Java进程最简单的方法是直接在Local Process中找到正在运行的 AppConfig，点击Connect即可连接到我们当前正在运行的Web应用，在jconsole中可直接看到内存、CPU等资源的监控。

我们点击MBean，左侧按分类列出所有MBean，可以在 java.lang 查看内存等信息：

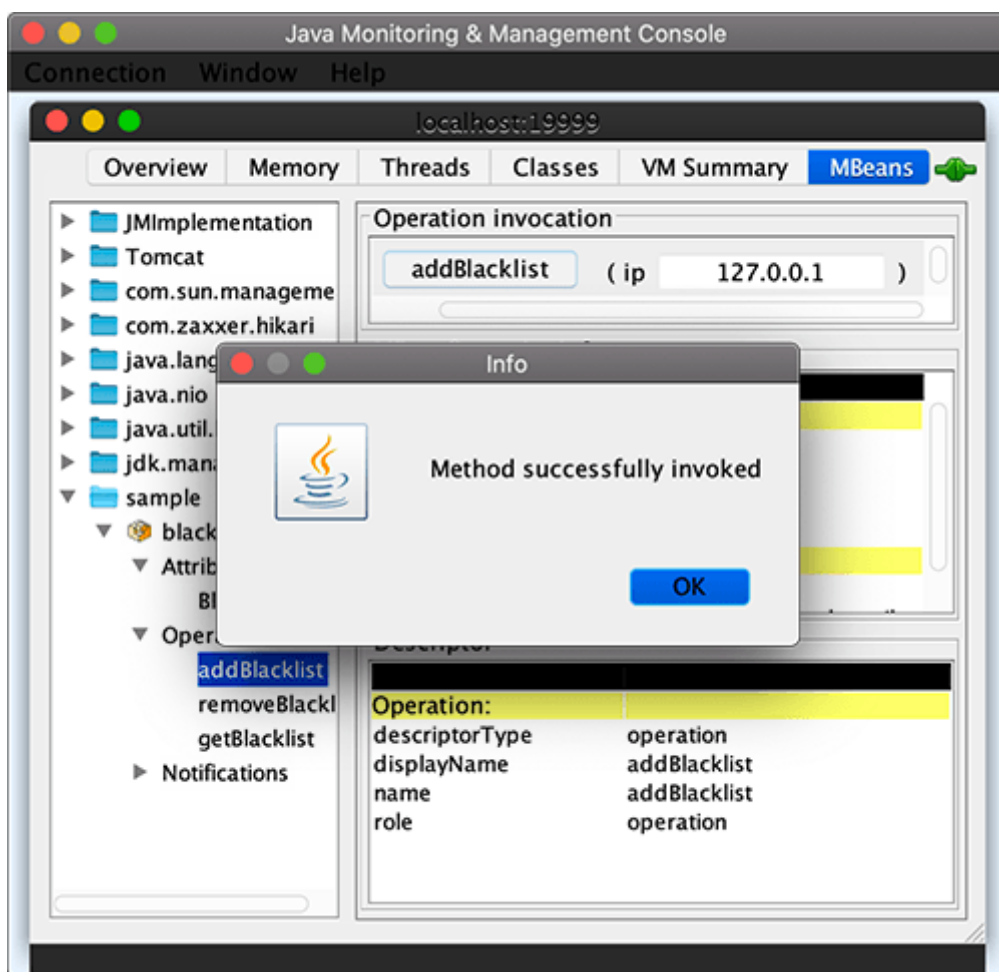


细心的童鞋可以看到HikariCP连接池也是通过JMX监控的。

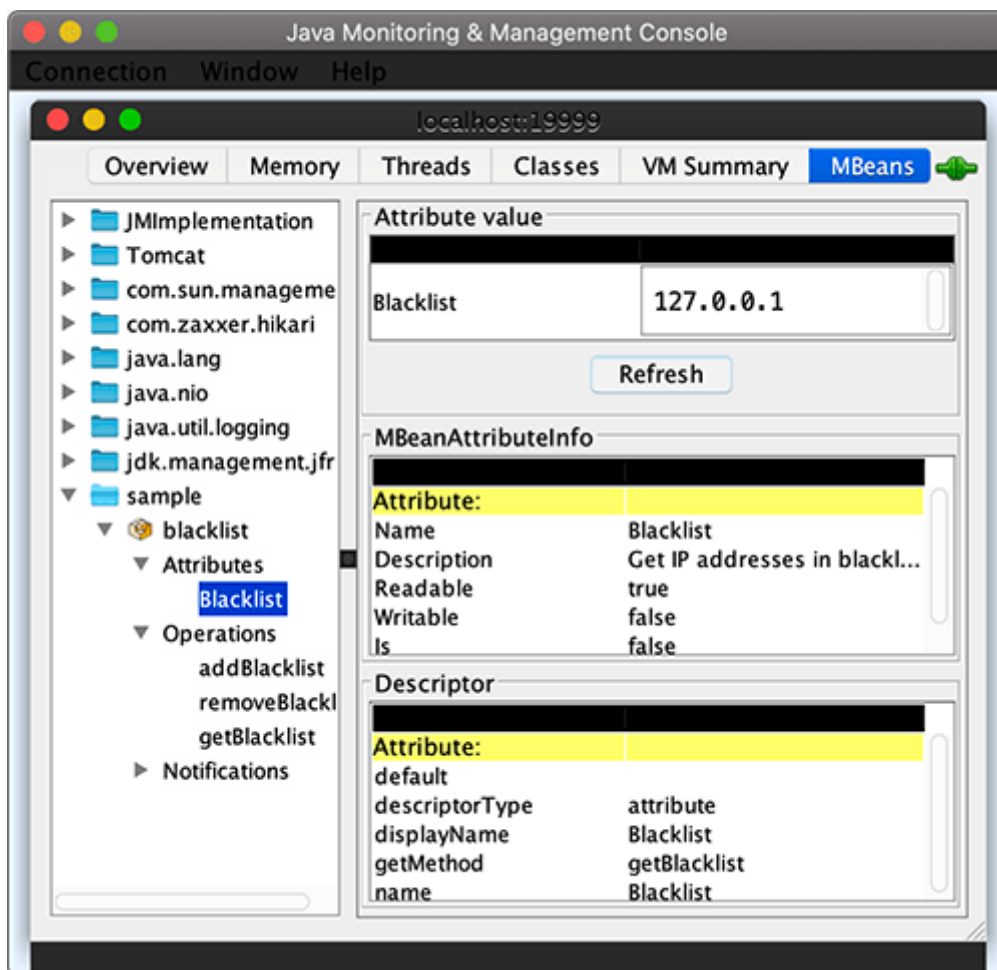
在 sample 中可以看到我们自己的MBean，点击可查看属性 blacklist：



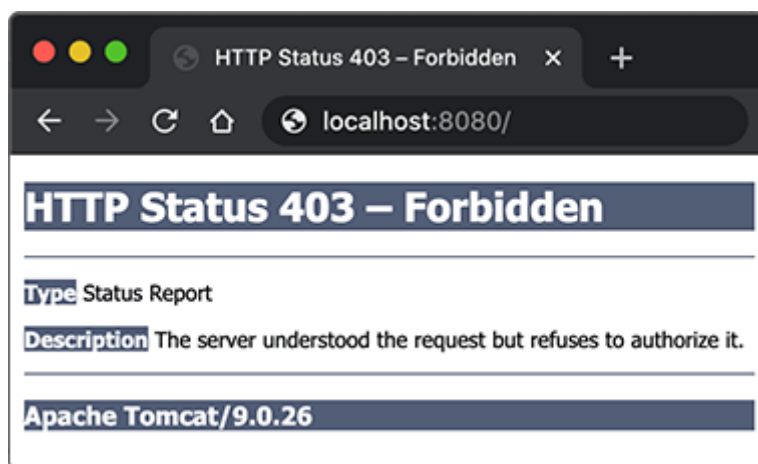
点击 Operations - addBlacklist，可以填入 127.0.0.1 并点击 addBlacklist 按钮，相当于jconsole通过JMX接口，调用了我们自己的 BlacklistMBean 的 addBlacklist() 方法，传入的参数就是填入的 127.0.0.1：



再次查看属性 blacklist，可以看到结果已经更新了：



我们可以在浏览器中测试一下黑名单功能是否已生效：



可见，127.0.0.1 确实被添加到了黑名单，后台日志打印如下：

```
2020-06-06 20:22:12 INFO c.i.l.web.BlacklistInterceptor - check ip address 127.0.0.1...
2020-06-06 20:22:12 WARN c.i.l.web.BlacklistInterceptor - will block ip 127.0.0.1 for it is in blacklist.
```

注意：如果使用IPv6，那么需要把 0:0:0:0:0:0:0:1 这个本机地址加到黑名单。

如果从jconsole中调用 `removeBlacklist` 移除 127.0.0.1，刷新浏览器可以看到又允许访问了。

使用jconsole直接通过Local Process连接JVM有个限制，就是jconsole和正在运行的JVM必须在同一台机器。如果要远程连接，首先要打开JMX端口。我们在启动 AppConfig 时，需要传入以下JVM启动参数：

- `-Dcom.sun.management.jmxremote.port=19999`

- -Dcom.sun.management.jmxremote.authenticate=false
- -Dcom.sun.management.jmxremote.ssl=false

第一个参数表示在19999端口监听JMX连接，第二个和第三个参数表示无需验证，不使用SSL连接，在开发测试阶段比较方便，生产环境必须指定验证方式并启用SSL。详细参数可参考Oracle[官方文档](#)。这样jconsole可以用 `ip:19999` 的远程方式连接JMX。连接后的操作是完全一样的。

许多JavaEE服务器如JBoss的管理后台都是通过JMX提供管理接口，并由Web方式访问，对用户更加友好。

练习

编写一个MBean统计当前注册用户数量，并在jconsole中查看：

从  **gitee** 下载练习：[编写MBean](#)（推荐使用[IDE练习插件](#)快速下载）

小结

在Spring中使用JMX需要：

- 通过 `@EnableMBeanExport` 启用自动注册MBean；
- 编写MBean并实现管理属性和管理操作。