

# 6.4 Spring 数据库连接

---

- ORM 与 JPA
- JPA 的使用
- JPA 功能补充



# 目录

1

ORM 与 JPA

2

JPA 的使用

3

JPA 功能补充

# 本节目标

- 在 Spring Boot 中实现登录、注册界面和数据库的连接。

## Register

Please fill in the form to create an account.

**Username**

**Password**

☒ By creating an account you agree to our Terms & Privacy.

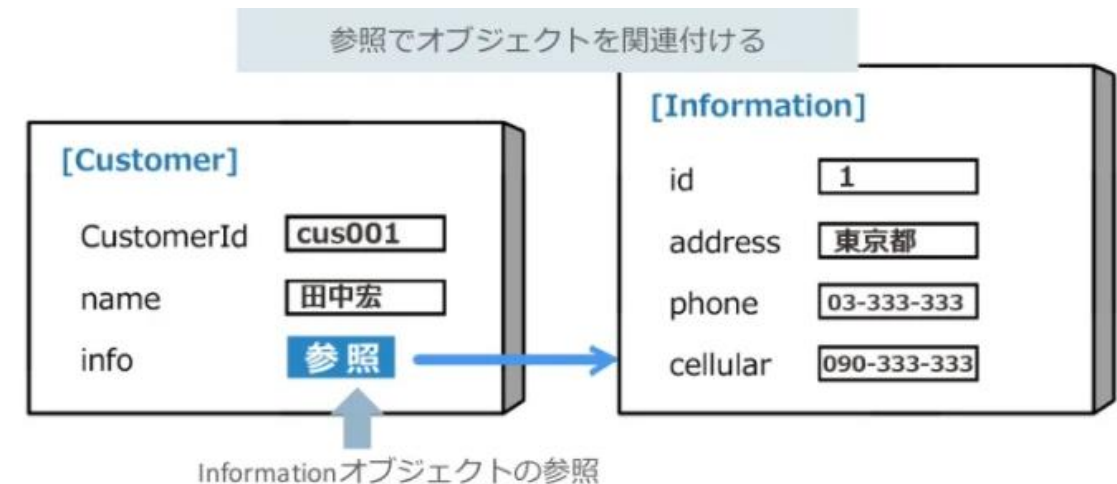
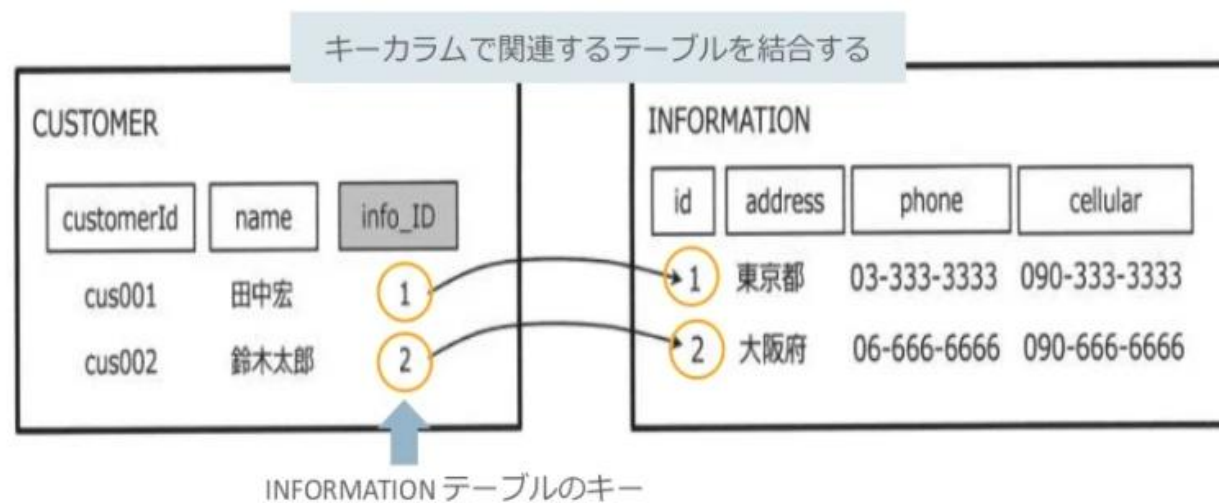
Register

Already have an account? [Sign in.](#)

jpa\_test=# SELECT \* FROM account;  
id | password | username  
-----  
1 | ABC87654 | Alice  
(1 行)

# ORM

- **ORM** (Object Relational Mapping, 对象关系映射) 是一种将**对象**和**关系** (即数据库中的表) 对应起来的软件设计技术。
- ORM 把数据库映射成面向对象编程中的元素:
  - 每一个**表**对应一个**类**;
  - 每一个**记录** (表中的一行) 对应一个**对象**;
  - 每一个**字段**对应一个对象的**属性** (成员变量);
  - 关系中的**外键**对应其它对象的**引用**。





# ORM 的例子

- 比如，下面这行 SQL 语句：

```
SELECT id, first_name, last_name, phone  
FROM person  
WHERE id = 10;
```

- 改成 ORM 的写法如下：

```
1 Person person = repository.findById(10);  
2  
3 System.out.println("First Name: " + person.first_name);  
4 // ...
```

- 一比较就可以发现，ORM 使用对象，封装了数据库操作，因此可以不碰 SQL 语言。开发者只使用面向对象编程，与数据对象直接交互，不用关心底层数据库。

# JPA

- **JPA** (Java Persistence API) 是 Java EE 中制定的一个用于实现 ORM 标准的接口。
- Spring 开发了 Spring Data JPA 用于实现 JPA 标准。使用 Spring Data JPA, 我们可以在几乎不书写任何 SQL 代码的前提下实现以下功能:
  - 表的建立;
  - 数据的基本增删改查;
  - 相对复杂的查询操作, 比如根据某些列查询、排序、分页等。
  - 通过外键获得其它表的对象等。

# ORM 的优缺点

## ● 优点：

- 数据模型都在一个地方定义，更容易更新和维护，也利于重用代码。
- ORM 有现成的工具方法，很多功能都可以自动完成，比如数据消毒、预处理、事务等等。
- 它迫使你使用 **MVC 架构**，ORM 就是天然的 Model，使代码更清晰。
- 基于 ORM 的业务代码比较简单，代码量少、语义性好，容易理解。

## ● 缺点：

- ORM 库不是轻量级工具，需要花很多精力学习和设置。
- 对于复杂的查询，ORM 要么是无法表达，要么是性能不如原生的 SQL。
- ORM 抽象掉了数据库层，开发者无法了解底层的数据库操作，也较难定制一些特殊的 SQL。

Q & A

*Question and answer*



# 目录

1

ORM 与 JPA

2

JPA 的使用

3

JPA 功能补充

# 添加 JPA 依赖

- 在创建项目时勾选 JPA 和 Postgres 驱动的依赖。

Available:	Selected:
<input type="text" value="Type to search dependencies"/>	
<div>SQL</div> <div><input type="checkbox"/> JDBC API</div> <div><input checked="" type="checkbox"/> Spring Data JPA</div> <div><input type="checkbox"/> Spring Data JDBC</div> <div><input type="checkbox"/> Spring Data R2DBC</div> <div><input type="checkbox"/> MyBatis Framework</div> <div><input type="checkbox"/> Liquibase Migration</div> <div><input type="checkbox"/> Flyway Migration</div> <div><input type="checkbox"/> JOOQ Access Layer</div> <div><input type="checkbox"/> IBM DB2 Driver</div> <div><input type="checkbox"/> Apache Derby Database</div> <div><input type="checkbox"/> H2 Database</div> <div><input type="checkbox"/> HyperSQL Database</div> <div><input type="checkbox"/> MariaDB Driver</div> <div><input type="checkbox"/> MS SQL Server Driver</div> <div><input type="checkbox"/> MySQL Driver</div> <div><input type="checkbox"/> Oracle Driver</div> <div><input checked="" type="checkbox"/> PostgreSQL Driver</div>	

# 添加 JPA 配置

- 要使用 JPA，我们需要在 `resources/application.properties` 配置文件中添加一些配置。
- 首先，和 JDBC 一样，我们要添加一些用于连接数据库的信息：

```
spring.datasource.url=jdbc:postgresql://localhost:5432/jpa_test
spring.datasource.username=postgres
spring.datasource.password=123456
```

- 接着，我们添加以下配置以根据 Java 代码自动生成表。这些配置暂时不需要更改：

```
spring.jpa.properties.hibernate.jdbc.lob.non_contextual_creation=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.PostgreSQLDialect
spring.jpa.hibernate.ddl-auto=update
```

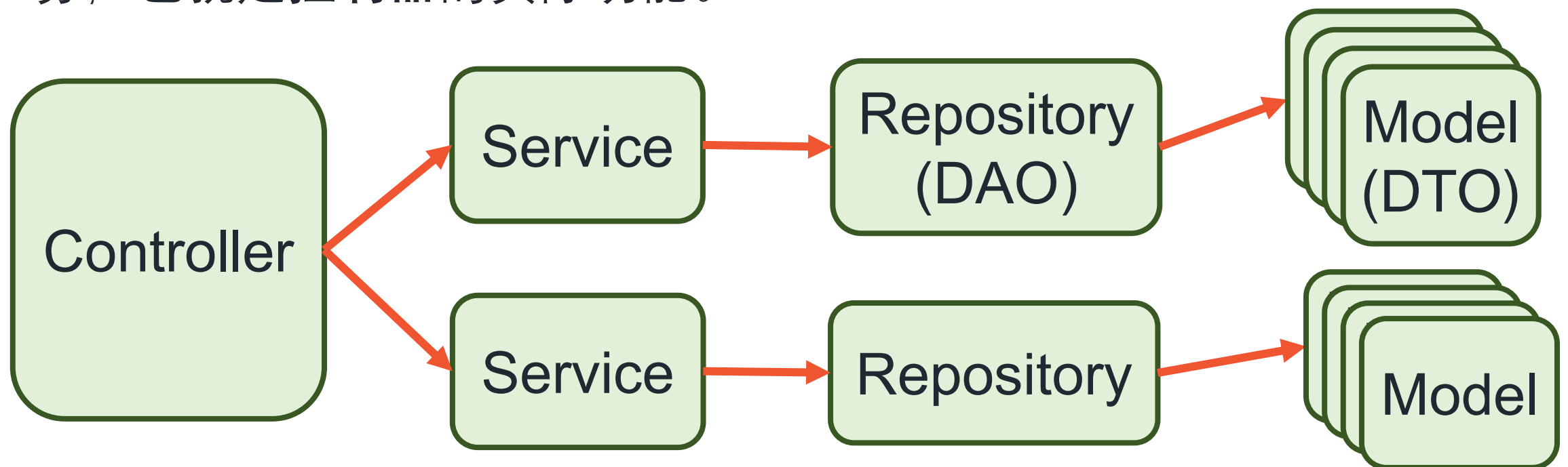


# 数据库设计

- 我们先思考一下用户登录系统需要一个什么样的数据库。
- 功能：
  - 注册新用户。
  - 已注册用户的登录验证。
- 用户数据：
  - 一个用户表 `account`。
  - 表的列：`id`, `username`, `password`。（数据库表中通常会加一个数字类型的 `ID` 作为主键，这是一个好的习惯。`ID` 的值通常设置为自增的。）

# 三层模型

- 在开发连接数据库的项目时，我们通常会创建以下 3 种类，构建三层模型：
  - 永久层（**Repository 层**），或 DAO 层：直接操作数据库，进行增删改查等永久化操作的类。
  - 服务层（**Service 层**）：使用永久层中的方法创建一些基本的服务，为控制器中的主逻辑提供一些功能的类。
  - 表现层（**Controller 层**），使用服务层中提供的方法实现实际的业务，也就是**控制器**的实际功能。



# 代码结构

- 我们可以把不同层的代码整理在不同包内，结构如下：

```
▼ JPA Test [boot]
  ▼ src/main/java
    ▼ net.lighthouseplan.spring.jpa
      ▼ controllers
        > LoginController.java
        > RegisterController.java
      ▼ models
        > Account.java
      ▼ repositories
        > AccountRepository.java
      ▼ services
        > AccountService.java
        > JpaTestApplication.java
```

## Note



这些包需要位于核心包（你的 Application 类在的包）内！



# Model 的定义

- 为了对应数据库中的 account 表，我们需要定义一个 Account 类，添加对应的**属性**和 **Getter、Setter**:

```
1 public class Account {  
2     private Long id;  
3     private String username;  
4     private String password;  
5  
6     public Long getId() { return id; }  
7     public void setId(Long id) { this.id = id; }  
8  
9     public String getUsername() { return username; }  
10    public void setUsername(String username) { this.username = username; }  
11  
12    public String getPassword() { return password; }  
13    public void setPassword(String password) { this.password = password; }  
14 }
```

# 为 Account 类添加注解

- 我们需要在 Account 类前添加 **@Entity** 注解。这将告诉 Spring 该类对应数据库中的一个表：

```
@Entity  
public class Account {
```

- 默认情况下，JPA 会自动帮我们为表取一个对应的名字。对应规则是大写驼峰形 → 小写蛇形。比如 AccountGroup 类会对应数据库中的 account\_group 表。
- 如果想手动设置对应的表名，可以使用 **@Table** 注解：

```
@Entity  
@Table(name = "account_2")  
public class Account {
```

# 属性的注解

- 类中的每一个属性对应一个数据库中的列名（字段）。如果我们没有特别设置，列名将会根据变量名自动设置（小写驼峰形 → 小写蛇形）。
- 你也可以使用 **@Column** 注解手动设置列名：

```
@Column(name = "user")  
private String username;
```

- 特别的，对于 ID 列，必须使用 **@Id** 注解标出：

```
@Id  
@GeneratedValue(strategy = GenerationType.AUTO)  
private Long id;
```

- 这里，**@GeneratedValue** 注解设置 ID 自动增加。



# DAO 层

- DAO 层处理和数据库有关的操作。
- 这个项目只有一个表 Account，因此 DAO 层只需要一个类：AccountRepository。
- 我们可以使用以下代码定义 AccountRepository 接口：

```
@Repository  
public interface AccountRepository extends JpaRepository<Account, Long> { }
```

# Repository 定义的解读

```
@Repository  
public interface AccountRepository extends JpaRepository<Account, Long> { }
```

- **@Repository** 注解告诉 Spring 这是一个持久层的接口，提供一些和数据库操作有关的方法。
- 这里我们定义的是一个 interface（接口）而不是 class（类），是因为我们只需要说明需要实现的方法，而具体的实现可以由 JPA 帮我们自动完成。
- 为此，我们需要继承 **JpaRepository<T, ID>** 接口。其中，T 是表里数据的类型（这里是 Account），ID 是数据 ID 的类型（这里是 Long）。

# 定义需要实现的方法

- 由于 JPA 可以帮我们自动实现，我们只需要声明需要的方法：

```
1 @Repository
2 public interface AccountRepository extends JpaRepository<Account, Long> {
3     Account findByUsername(String username);
4
5     Account save(Account account);
6 }
```

- 这里，第一个方法 **findByUsername()** 接受用户名，返回用户名对应的用户信息，相当于：

```
SELECT * FROM account WHERE username = [传入的参数]
```

- **save()** 方法将会向数据库中插入一个新的用户信息，并返回最终被插入表中的信息。（由于 **save** 方法的定义在 **JpaRepository** 中已经有了，此处也可以省略。）



# Service 层

- Service 层使用 DAO 层的 Repository 实现一些控制器可以使用的功能（服务）。
- 这里我们需要两个服务：**validateAccount()** 用于判断某个用户名和密码是否已被存在数据库中，**createAccount()** 使用用户名和密码创建一个新的用户。
- 创建一个 AccountService 类，并用 **@Service** 注解告诉 Spring 这是一个服务层的类：

```
1 @Service
2 public class AccountService { }
```

# Service 层

- AccountService 中的方法需要访问 account 表。我们已经创建了访问 account 表的 AccountRepository 接口。在 AccountService 中声明一个它的变量：

```
1 @Service
2 public class AccountService {
3     @Autowired
4     AccountRepository repository;
5 }
```

- 这里的 **@Autowired** 注解将会让 Spring 自动实现该接口并创建实例。我们可以直接在 Service 的方法中使用它。

# validateAccount() 方法

- validateAccount() 方法接收两个参数 username 和 password, 判断对应的数据是否已存在数据库中。如果存在, 则返回 true; 否则, 返回 false:

```
1 public boolean validateAccount(String username, String password) {  
2     Account account = repository.findByUsername(username);  
3     if (account == null || !account.getPassword().equals(password)) {  
4         return false;  
5     } else {  
6         return true;  
7     }  
8 }
```



## createAccount() 方法

- createAccount() 方法接收两个参数 username 和 password，并尝试创建一个新的用户账号。如果用户名已在数据库内，则插入失败，返回 false；否则，返回 true：

```
1 public boolean createAccount(String username, String password) {  
2     if (repository.findByUsername(username) == null) {  
3         repository.save(new Account(username, password));  
4         return true;  
5     } else {  
6         return false;  
7     }  
8 }
```

# 编写 Controller

- login.html 页面和 LoginController 的代码与上节中的代码类似，只是这次我们可以使用 Service 提供的方法。

```
1 public class LoginController {
2     @Autowired
3     AccountService accountService;
4
5     @PostMapping("/login")
6     public ModelAndView login(@RequestParam String username,
7                               @RequestParam String password, ModelAndView mav) {
8         if (accountService.validateAccount(username, password)) {
9             mav.addObject("name", username);
10            mav.setViewName("hello.html");
11        } else {
12            mav.setViewName("login.html");
13        }
14        return mav;
15    }
16    // ...
```

接次页 



```
@Autowired  
AccountService accountService;
```

- 和 Repository 一样，Spring 会自动帮我们实例化 Service 对象，我们直接使用即可。
- 完成后，我们可以启动服务器，并手动向数据库中添加一些用户信息。使用你的浏览器访问请求地址并查看登录效果。



Q & A

*Question and answer*

# 练习：编写 RegisterController

- 我们还剩下注册界面 register.html 和对应控制器 RegisterController 没有实现。
- 想一想，应该如何使用现有的 AccountService 实现？先自己动手试一试！

## Register

Please fill in the form to create an account.

**Username**

**Password**

☒ By creating an account you agree to our Terms & Privacy.

Register

Already have an account? [Sign in.](#)

→

```
jpa_test=# SELECT * FROM account;  
 id | password | username  
----+-----+-----  
  1 | ABC87654 | Alice  
(1 行)
```

## Coffee ☕ Break

### Service 层的作用

在某些简单的应用中，DAO 层的功能和 Service 的功能很接近，初学者甚至可能觉得 Service 层做的事情和 DAO 层一样，可以省略 Service 层。

通常来说，DAO 层应尽量保持简单。它应该提供数据库的连接，以及简单的增删改查，而 Service 层则应该提供一些业务逻辑相对复杂，但仍被反复使用的功能。

这样做有几个好处。首先，当更换数据库实现时，我们**仅需修改 DAO 层**的代码。并且，当业务逻辑比较复杂时，由 Service 实现这些代码可以**简化 Controller 里的逻辑**。其次，Controller 中只包含和网页、HTTP 请求相关的代码，使我们可以**独立地测试 Service 层**的代码。



# 目录

1

ORM 与 JPA


2

JPA 的使用

3

JPA 功能补充

# 更多 JPA 功能

- 除了最基本的增删改查之外，JPA 还为我们提供了很多其它进阶功能，让我们来大体了解一下有哪些：
  - 各种数据类型、约束的对应；
  - 更多查询方法；
  - 外键的对应；
  - 手动设置 SQL 代码；
  - .....
- 本课程没有介绍的功能可以在官方文档上查询：  
 <https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#repositories>

# 数据类型对应

- 要使用 JPA，我们需要了解 Postgres 中各个数据类型在 Java 中应该如何对应。下表列出了一些常用类型在 Java 中的对应：

数据类型	Postgres 类型名	Java 类型名
整数	INT	Integer
大整数	BIGINT	Long
自增整数	SERIAL	Integer
自增大整数	BIGSERIAL	Long
小数	NUMERIC	java.math.BigDecimal
字符串	VARCHAR	String
日期时间	TIMESTAMP	java.sql.Timestamp
日期	DATE	java.sql.Date
时间	TIME	java.sql.Time
布尔值	BOOL	Boolean



# 外键的对应

- 添加 **@OneToOne**、**@OneToMany** 或 **@ManyToOne** 等注解可通过某个属性分别增加一对一、一对多、多对一的外键对应。
- 以 **@OneToOne** 为例：

```
1 @OneToOne(cascade = CascadeType.ALL)
2 @JoinColumn(name = "address_id", referencedColumnName = "id")
3 private Address address;
```

- 这里 **@OneToOne** 对应说明在数据库中，该字段将对应另一个表 **address** 中的 1 个记录。而在 **Java** 中，我们可以通过该属性直接获取一个 **Address** 对象。
- **@JoinColumn** 里的 **name** 说明该列在数据库表里叫做 **address\_id**，它的值取自 **address** 表中对应记录的 **id** 字段。

# 基本增删改查方法

- 下表列出了 JPA 支持的**基本增删改查**方法：

方法名	作用
save()	保存数据
findAll()	查询所有记录
delete(All)ByXxx()	删除 xxx 字段和传入参数相等的记录
find(All)ByXxx()	查询 xxx 字段和传入参数相等的记录
existsByXxx()	判断 xxx 字段和传入参数相等的记录是否存在
countByXxx()	统计 xxx 字段和传入参数相等的记录的数量

# 增删改查方法扩充

- 使用一些运算符关键字描述复杂的条件:

```
List<Student> findAllByScoreIsNotNull();  
List<Student> findAllByNameLike(String pattern);  
List<Student> findAllByScoreGreaterThan(Integer min);
```

- 还可以使用 And 等关键字组合多个查询条件:

```
List<Student> findAllByNameLikeOrId(String pattern, Long id);  
List<Student> findAllByScoreIsNotNullAndScoreGreaterThan(Integer minScore);  
List<Student> findAllBySubjectOrderByAgeDesc(String subject);
```



# 使用 Pageable 类查询

- 使用 **Pageable** 类，可以指定返回查询结果的第几页：

```
1 Pageable paging = PageRequest.of(2, 5);  
2  
3 Page<Product> productList = productRepository.findAll(paging);
```

- `PageRequest.of()` 至少要传入 2 个参数，其中第 1 个表示查询第几页（从 0 开始），第 2 个表示每页有几个记录。比如，`PageRequest.of(2, 5)` 将会查询第 11 条到 15 条记录。

# 手动设定 SQL 语句

- 有时，Spring 自带的语法没有办法满足我们的需求，我们可以自己手动书写 SQL 代码操作数据库。
- 要设定某方法运行自己手动设定的 SQL 语句，在方法前添加 **@Query** 注解：

```
@Query(value = "SELECT * FROM employee WHERE name = ?1",  
        nativeQuery = true)  
public List<Employee> findByName(String name);
```

- 在 SQL 语句中使用“?1”、“?2”等占位符分别表示方法的第一、第二个等输入。

Q & A

*Question and answer*



# 总结

## Sum Up

1. Java 中连接数据库的标准：ORM 和 JPA。
2. 开发网络应用时的三层模型：Repository (DAO)、Service、Controller。
3. Spring 中实现 JPA 的组件 Spring Data JPA：基本语法和各个注解的含义。

**THANK YOU!**