

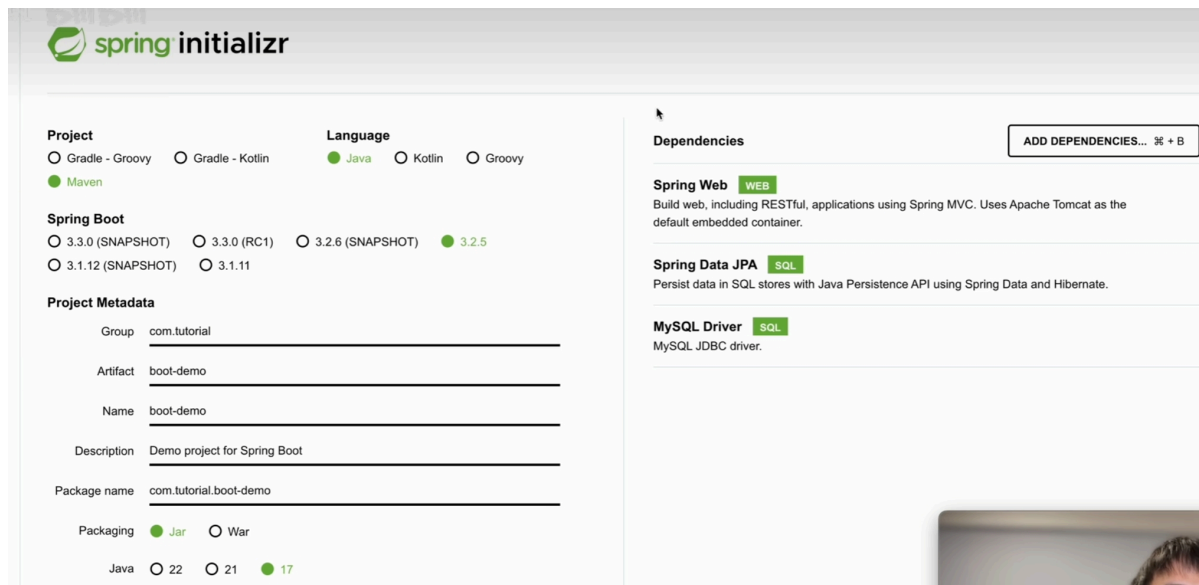
Day1

psvm、sout

```
fori for (int i = 0; i < ; i++)
```

SSM (spring springmvc mybatis) 、spring boot

<https://start.spring.io/> 启动窗口

The image shows the Spring Initializr web form. It is divided into three main sections: Project, Language, and Dependencies. The Project section includes options for Project (Maven selected), Spring Boot version (3.2.5 selected), and Project Metadata (Group: com.tutorial, Artifact: boot-demo, Name: boot-demo, Description: Demo project for Spring Boot, Package name: com.tutorial.boot-demo, Packaging: Jar selected, Java: 17 selected). The Language section includes options for Language (Java selected), Kotlin, and Groovy. The Dependencies section includes a list of dependencies: Spring Web (WEB), Spring Data JPA (SQL), and MySQL Driver (SQL). There is an 'ADD DEPENDENCIES...' button in the top right corner of the Dependencies section.

解压缩后用idea选中pom.xml打开

- pom 中的jpa 会寻找数据库的url, 先注释掉
- 快捷键 `Ctrl + /` 快速注释 xml多行
- <http://localhost:8080/> Spring Boot应用程序的默认端口
- 404, 因为没有API,所有API都是通过controller提供的

- 在包中新建Controller类, 在类中写@RestController, 表明是Rest api的Controller

=@Controller @ResponseBody

```
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/api")
public class MyRestController {
    @GetMapping("/hello")
    public String hello() {
        return "Hello, Spring Boot!";
    }
}
```

}

- `@RestController` 声明 `MyRestController` 类是一个控制器，并且返回值将直接写入HTTP响应体。
- `@RequestMapping("/api")` 定义了这个控制器的基础URL路径为 `/api`。
- `@GetMapping("/hello")` 定义了一个HTTP GET请求的处理方法，当访问 `http://localhost:8080/api/hello` 时，会返回字符串 `"Hello, Spring Boot!"`。

如果要返回的是普通对象，spring会返回一个json字符串在浏览器上，如果对象包含set get方法则无法序列化为json 如

```
@GetMapping("/hello")
public List<String> hello() {
    return List.of("hello","world");
}
```

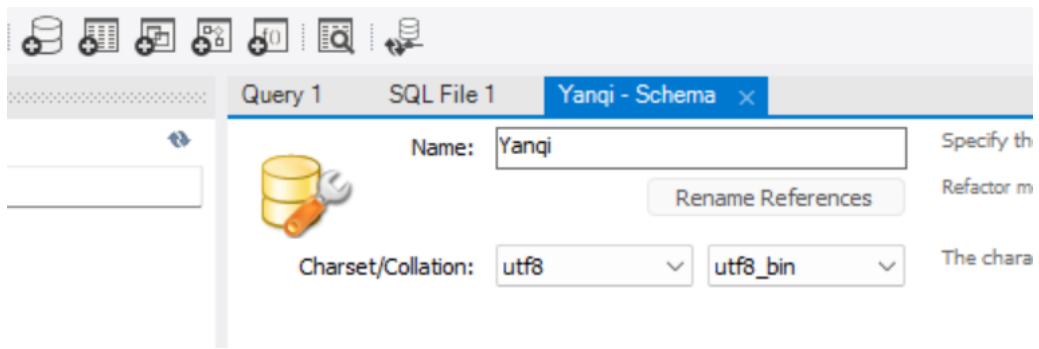
结果为

```
["hello","world"]
```

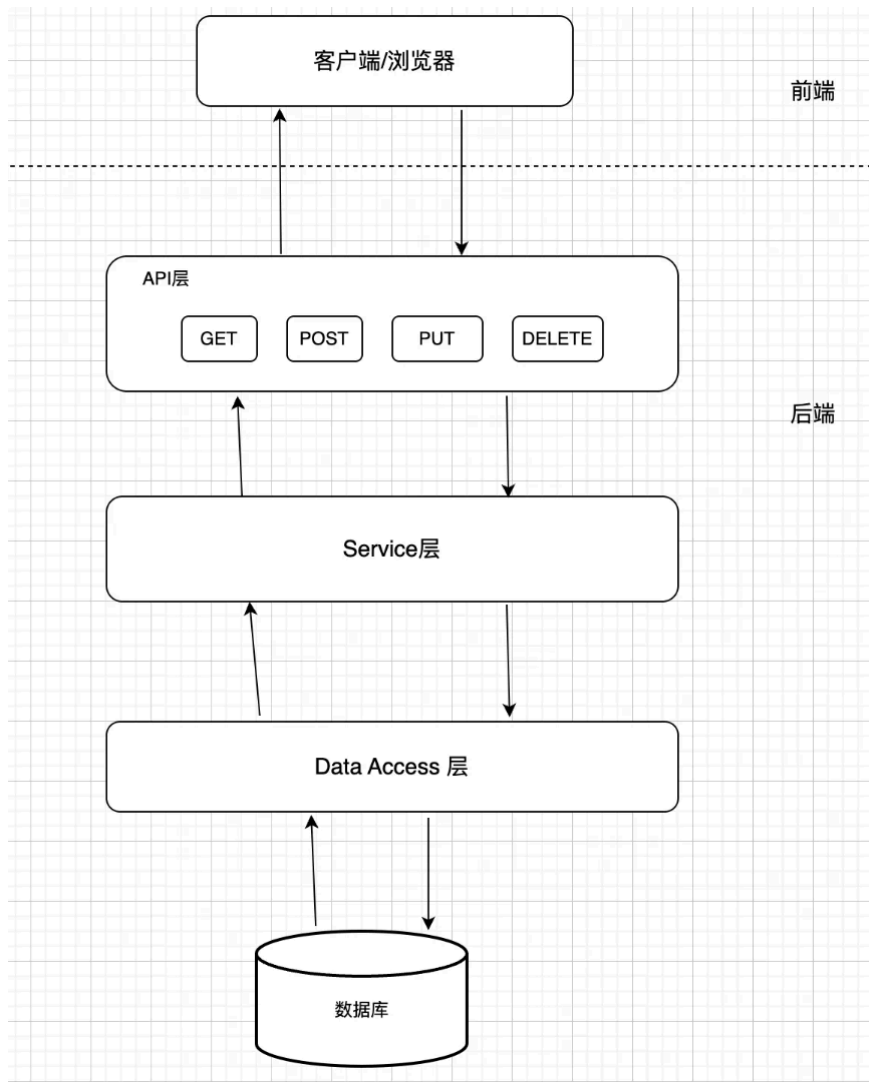
变成了数组

Http 动词

- GET (SELECT)：从服务器取出资源（一项或多项）。
- POST (CREATE)：在服务器新建一个资源。
- PUT (UPDATE)：在服务器更新资源（客户端提供改变后的完整资源）。
- PATCH (UPDATE)：在服务器更新资源（客户端提供改变的属性）。
- DELETE (DELETE)：从服务器删除资源。



utf8 建数据库中文不会乱码



从数据库往上构建

在java下的包中在建个新的包dao, 里面建名叫XXXRepository的接口。

```

@Repository
public interface StudentRepository extends JpaRepository {

}
  
```

再建一个student表映射的类。

```

package com.example.demo.dao;
  
```

```

import jakarta.persistence.*;

import static jakarta.persistence.GenerationType.IDENTITY;

@Entity
@Table(name="student")
public class Student {

    @Id 标识主键

    @Column(name="id")
    @GeneratedValue(strategy = IDENTITY)
    private long id;

    @Column(name="name")
    private String name;

    @Column(name="email")
    private String email;

    @Column(name="age")
    private int age;

}

```

`@Entity` 是一个注解，用于指定一个类是一个JPA实体。JPA (Java Persistence API) 是Java平台的一个规范，主要用于对象关系映射（ORM），它允许开发者使用面向对象的编程方式来操作关系型数据库。

继续改接口

```

@Repository
public interface StudentRepository extends JpaRepository<Student,Long> {
}

```

大写的是类，小写的是数据库的表？

接下来是服务层，创建一个接口和一个类，类实现此接口

```

@Service
public class StudentServiceImpl implements StudentService{
}

```

然后定义接口，返回dao的Student

```
package com.example.demo.Service;

import com.example.demo.dao.Student;

public interface StudentService {

    public Student getStudentById(long id);
}
```

继续写

```
public class StudentServiceImpl implements StudentService{

    @Autowired //自动注入依赖
    private StudentRepository studentRepository;
    自动整出一个名为studentRepository的实例

    @Override
    public Student getStudentById(long id){
        return studentRepository.findById(id).orElseThrow(RuntimeException::new);
        找不到就丢异常
    }
}
```

studentRepository是空的，只继承jpaRepository,findById是父类的方法。

最后是controller层也就是API层的代码

MyBatis

1. 简介:

- MyBatis 是一个半自动化的持久化框架。它简化了JDBC操作，但仍然需要开发者编写SQL语句。
- MyBatis 强调SQL的灵活性，开发者可以完全控制SQL的执行。

2. 特点:

- **SQL控制**: 开发者可以完全掌控SQL语句，适合复杂查询。
- **轻量级**: MyBatis 比较轻量，不需要像JPA那样的实体映射。
- **XML和注解**: 支持在XML文件或注解中编写SQL映射。

JPA (Java Persistence API)

1. 简介:

- JPA 是一个Java的持久化规范，由Java EE 提供。它定义了一组接口，实体映射，查询语言 (JPQL) ，以及事务管理等。
- JPA 旨在简化数据持久化，将对象映射到数据库表，使得开发者可以使用面向对象的方式进行数据操作。

2. 特点:

- **对象关系映射 (ORM)**：JPA 提供自动化的对象关系映射机制。
- **标准化**：JPA 是Java EE的标准规范，有多个实现 (如 Hibernate, EclipseLink) 。
- **查询语言**：JPA 提供了面向对象的查询语言 (JPQL)

@Repository用于标记数据访问层 (DAO) 的类，表示该类专门用于与数据库进行交互。

在 Spring 的组件扫描 (component scanning) 机制下，**@Repository** 注解的类会被自动检测和注册为 **Spring Bean**。

Spring 提供了几个类似的注解，用于标识不同层次的组件：

1. @Component:

- 最基础的注解，用于标识一个通用的 Spring Bean。
- 其他特定层次的注解 (如 **1 @Repository**, **2 @Service**, **3 @Controller**) 都是 **@Component** 的特殊化。

2. @Service:

- 用于标识业务层 (Service) 的组件。
- 主要用于表示业务逻辑服务类，通常包含业务逻辑处理。

3. @Controller:

- 用于标识表示层 (Web 控制器) 的组件。
- 主要用于**处理 HTTP 请求**，并将请求结果返回视图。

Spring Bean：就像咖啡店里的物品和设备，它们是你需要使用的各种对象。

Spring容器：就像一个管理员，他负责创建和管理这些物品和设备，并在你需要时提供给你。

// 定义一个咖啡机Bean

```
public class CoffeeMachine {
    public void brew() {
        System.out.println("Brewing coffee...");
    }
}

// 定义一个咖啡豆Bean
public class CoffeeBeans {
    public void grind() {
        System.out.println("Grinding coffee beans...");
    }
}
```

```

}

// 定义一个牛奶Bean
public class Milk {
    public void add() {
        System.out.println("Adding milk...");
    }
}

```

配置Spring容器（管理员）

使用Spring配置文件或者Java配置类来配置这些Bean。

```

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class CoffeeShopConfig {
    @Bean
    public CoffeeMachine coffeeMachine() {
        return new CoffeeMachine();
    }
    @Bean
    public CoffeeBeans coffeeBeans() {
        return new CoffeeBeans();
    }

    @Bean
    public Milk milk() {
        return new Milk();
    }
}

```

```

}

```

步骤3：使用Spring容器（管理员）

让Spring容器来管理和提供这些Bean

```

public class CoffeeShop {
    public static void main(String[] args) {
        // 使用Spring容器来加载配置
        ApplicationContext context = new
        AnnotationConfigApplicationContext(CoffeeShopConfig.class);

        // 获取Bean（物品和设备）
        CoffeeMachine coffeeMachine = context.getBean(CoffeeMachine.class);
        CoffeeBeans coffeeBeans = context.getBean(CoffeeBeans.class);
        Milk milk = context.getBean(Milk.class);

        // 使用Bean来制作咖啡
        coffeeBeans.grind();
    }
}

```

```
        coffeeMachine.brew();  
        milk.add();  
    }  
}
```

通过继承和互相调用确实可以实现一些对象之间的关系和行为，但是使用Spring Bean和Spring容器提供了一些重要的优势和解决了许多面向对象编程中的实际问题。

问题：

- **紧耦合**：在没有Spring的情况下，对象之间往往是紧耦合的，修改一个对象可能会导致其他对象的修改。
- **模块化困难**：难以将不同的功能模块化，因为对象间的依赖关系很难管理。

Spring的解决方案：

- **依赖注入 (Dependency Injection)**：通过Spring容器来管理对象的依赖关系，可以在配置文件中定义对象间的依赖，而不是在代码中硬编码。
- **松耦合**：对象之间的关系通过配置文件或注解来定义，减少了代码间的耦合度，提高了模块化程度。
-

提高可测试性

问题：

- **难以测试**：紧耦合的对象难以进行单元测试，因为需要初始化大量的依赖对象。

Spring的解决方案：

- **依赖注入**：使得对象的依赖关系可以通过配置来注入，可以轻松替换依赖对象，从而更容易编写单元测试。
- **Mock对象**：可以使用Mock对象替换实际的依赖对象，进行独立测试

统一配置和管理

问题：

- **散乱的配置**：传统的方式中，配置分散在代码的各个部分，难以统一管理和维护。

Spring的解决方案：

- **集中配置**：通过Spring的配置文件或注解，可以将所有的配置集中管理，便于维护和修改。
- **灵活性**：可以轻松修改配置而不需要改变代码逻辑，例如切换数据源或更改实现类

面向切面编程 (AOP)

问题：

- **横切关注点**：像日志记录、事务管理等逻辑通常散落在代码的各个部分，导致代码冗余和混乱。

Spring的解决方案：

- **AOP**：Spring提供了面向切面编程的支持，可以将横切关注点分离出来，提高代码的可读性和可维护性。

传统写法

```
public class Coffeeshop {
    private CoffeeMachine coffeeMachine = new CoffeeMachine();
    private CoffeeBeans coffeeBeans = new CoffeeBeans();
    private Milk milk = new Milk();

    public void makeCoffee() {
        coffeeBeans.grind();
        coffeeMachine.brew();
        milk.add();
    }
}
```

使用Spring:

```
@Configuration 配置类
public class CoffeeshopConfig {
    @Bean
    public CoffeeMachine coffeeMachine() {
        return new CoffeeMachine();
    }

    @Bean
    public CoffeeBeans coffeeBeans() {
        return new CoffeeBeans();
    }

    @Bean
    public Milk milk() {
        return new Milk();
    }
}

public class Coffeeshop {
    @Autowired
    private CoffeeMachine coffeeMachine;

    @Autowired
    private CoffeeBeans coffeeBeans;
```

```
@Autowired
private Milk milk;

public void makeCoffee() {
    coffeeBeans.grind();
    coffeeMachine.brew();
    milk.add();
}
}
```

起名字时一般是首字母大写，括号中一般不大写