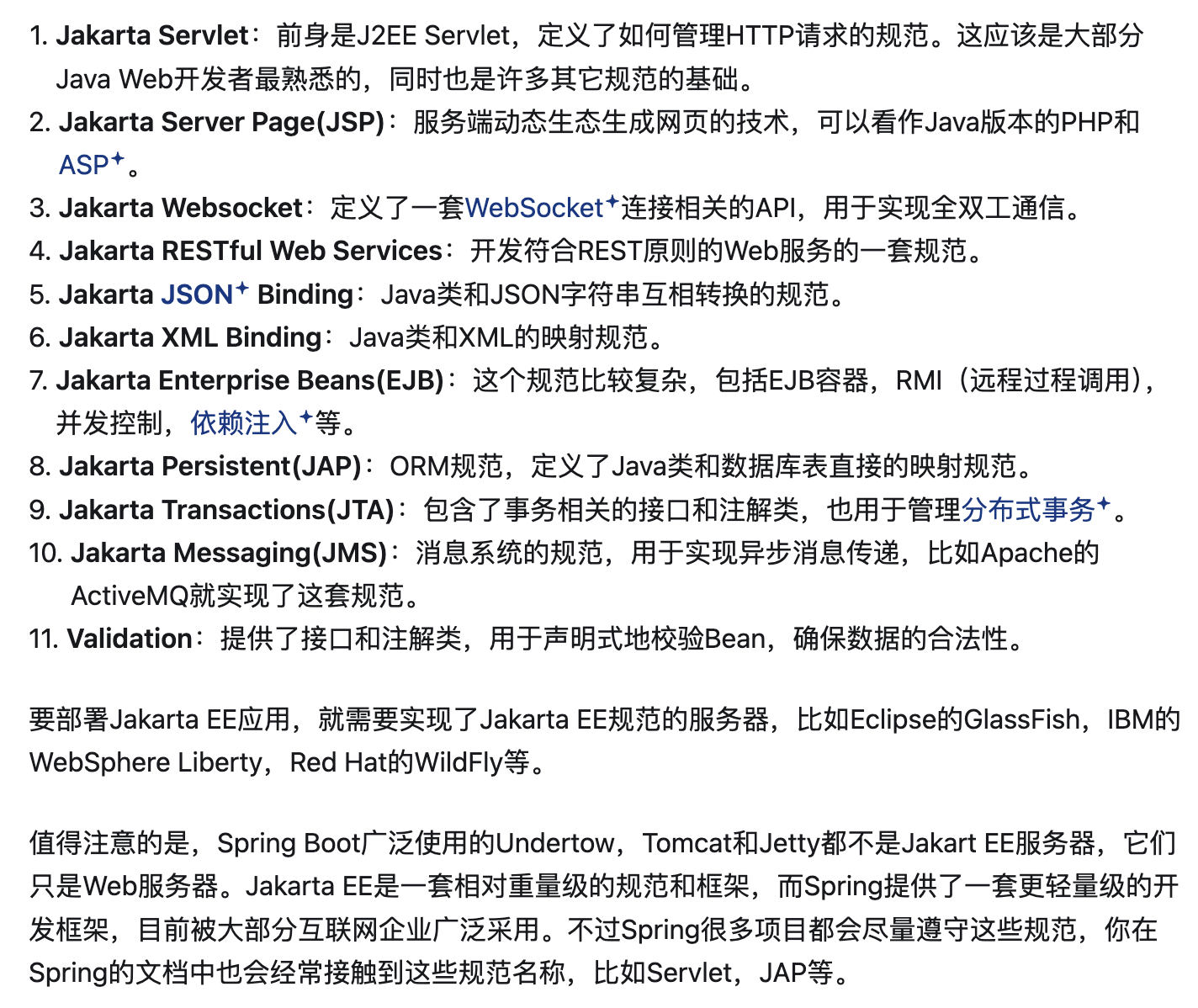
# 对jakrata 的支持

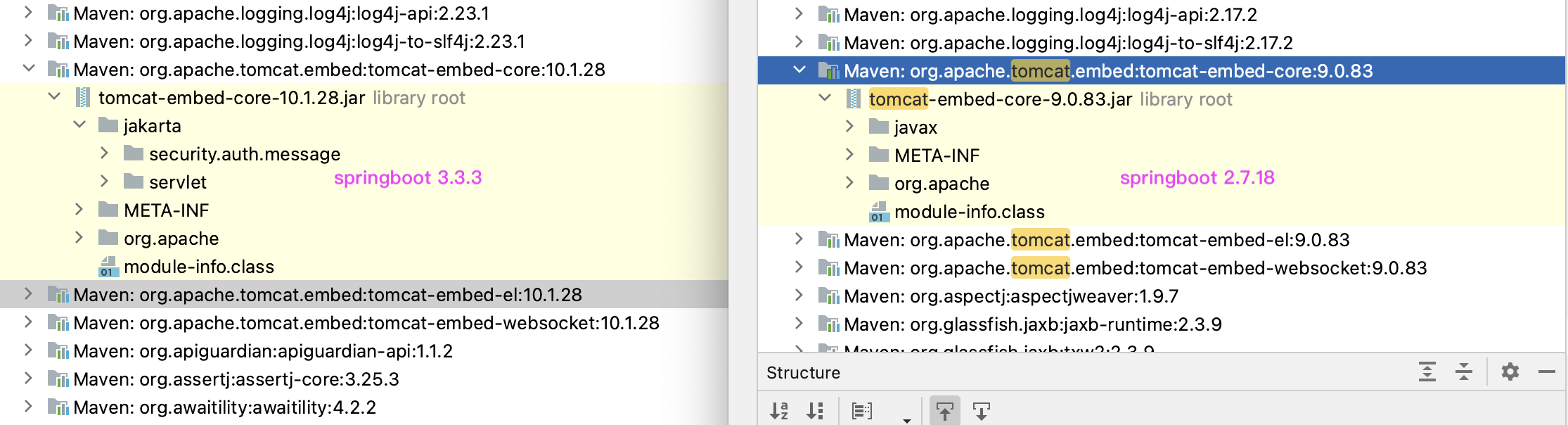
.javax包和jakarta包功能上是差不多的，jakarta是由javax演变而来的，但是如果乱用，会出现版本不兼容问题，比如在springboot项目中，如果内嵌的tomcat容器的版本为9.几，那它是不支持jakarta的，此时只能使用javax



但是Jakarta EE 9把命名空间改了，这就是breaking changes了。现在引用javax的都要改成jakarta，包括直接或者间接依赖的包。Spring Boot 3.0的升级指南也明确指出，同一个项目中不要混用kakarta和javax两个命名空间下的类。这对于老项目的升级是个很大的障碍，就算你自己项目里的代码都改了，但是你不能保证你依赖的所有jar包也都升级了。



Tomcat9 使用的是javax，而10及以上版本使用jakarta



# 对测试的优化

1. JUnit 5 的原生支持

Spring Boot 2.x:

Spring Boot 2.x 支持 JUnit 4 和 JUnit 5，但对于 JUnit 5 的支持相对基础。

Spring Boot 3.x:

提供了更好的对 JUnit 5 的支持，强烈推荐使用 JUnit 5 进行新的测试开发。

许多注解和功能（例如 @SpringBootTest）在 JUnit 5 上得到了优化。

2. 增强的 Mockito 集成

Spring Boot 2.x:

使用 Mockito 进行单元测试时，需要手动配置一些 MockBean。

Spring Boot 3.x:

提供了更丰富的断言和 Mock 的集成，支持 Mockito 的新版本（如 MockMvc 的扩展）。

3. 更好的 MockMVC 支持

Spring Boot 2.x:

MockMVC 的功能齐全，但在使其适用于某些高级用例时可能有些复杂。

Spring Boot 3.x:

引入了一些新的 API 和配置选项，使得 MockMVC 更加易用和强大。



# 3. Flyway的支持 （贴一张model risk 的script的截图）

Flyway 是一个数据库版本控制工具，可以与 Spring Boot 集成，帮助管理数据库的迁移和版本控制。使用 Flyway，可以轻松地维护数据库架构，并确保在不同环境（如开发、测试和生产）之间一致。

以下是如何在 Spring Boot 项目中集成和使用 Flyway 的基本步骤：

### 1. 添加依赖

在 `pom.xml` 文件中添加 Flyway 的依赖（如果使用 Gradle，可以在 `build.gradle` 中添加相应的依赖）：

```xml

<dependency>

<groupId>org.flywaydb</groupId>

<artifactId>flyway-core</artifactId>

</dependency>

```

### 2. 配置数据源

在 `application.properties` 或 `application.yml` 中设置数据库连接。Flyway 会自动使用这些信息来连接到你的数据库。

```properties

# application.properties

spring.datasource.url=jdbc:mysql://localhost:3306/yourdb

spring.datasource.username=yourusername

spring.datasource.password=yourpassword

spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

# Flyway 配置

spring.flyway.enabled=true

```

### 3. 创建迁移脚本

在项目的 `src/main/resources/db/migration` 目录下创建数据库迁移脚本。Flyway 会自动检测这些文件并执行。

例如，创建一个名为 `V1\_\_Create\_person\_table.sql` 的文件：

```sql

CREATE TABLE person (

id INT PRIMARY KEY AUTO\_INCREMENT,

name VARCHAR(100),

age INT

);

```

命名约定：

- 版本号（例如 `V1`、`V2`）

- 描述（例如 `Create\_person\_table`）

- SQL 文件扩展名

### 4. 运行应用程序

当你启动 Spring Boot 应用时，Flyway 会自动运行未执行的迁移脚本。你会在控制台中看到迁移的日志，表明 Flyway 正在安装版本并更新数据库。

### 5. 检查迁移状态

你可以使用 Flyway 提供的命令来检查当前的数据库状态。可以在控制台中进行如下操作：

```bash

mvn flyway:info # 查看迁移信息

mvn flyway:migrate # 执行未执行的迁移

```

### 6. 附加特性

- \*\*回滚\*\*：Flyway 主要不支持回滚，但可以创建新的迁移脚本来撤消更改。

- \*\*自定义迁移\*\*：除了 SQL 文件，你也可以使用 Java 进行迁移。

- \*\*混合迁移\*\*：可以在同一项目中混合使用 SQL 和 Java 迁移。

### 示例代码

这是一个简单的 Spring Boot 应用程序配置和 Flyway 集成的示例：

```java

@SpringBootApplication

public class DemoApplication {

public static void main(String[] args) {

SpringApplication.run(DemoApplication.class, args);

}

}

```

### 总结

Flyway 是一个非常易用且强大的迁移工具，适合管理数据库的版本控制与变更。如果有其他具体需求或者问题，欢迎随时问我哦！🚀

# 4. 使用virtual Threads

虚拟线程（Virtual Threads）是 Java 19 引入的一个新特性，旨在简化并发编程，尤其是在处理大量并发任务时。它们使得编写非阻塞代码变得更简单，同时在性能和资源消耗方面也更具优势。

在 Spring Boot 中，使用虚拟线程可以帮助构建更加高效的异步应用程序。下面是如何在 Spring Boot 项目中使用虚拟线程的基本步骤：

### 1. 确保使用适合的 JDK

首先，确保你的开发环境和 Spring Boot 应用程序运行在 Java 19 或更高版本，这是虚拟线程功能的基础。

### 2. 使用 Virtual Threads

Java 19 引入 `java.lang.Thread.ofVirtual()` 方法，用于创建虚拟线程。下面是一个简单的使用示例：

```java

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import java.util.concurrent.Executors;

@SpringBootApplication

public class DemoApplication {

public static void main(String[] args) {

SpringApplication.run(DemoApplication.class, args);

// 创建一个虚拟线程

var executor = Executors.newVirtualThreadPerTaskExecutor();

executor.execute(() -> {

System.out.println("Hello from a virtual thread! 💻");

});

}

}

```

### 3. 在 Controller 中使用虚拟线程

你可以在 Spring Boot 的 Controller 中使用虚拟线程处理请求。例如：

```java

import org.springframework.web.bind.annotation.GetMapping;

import org.springframework.web.bind.annotation.RestController;

import java.util.concurrent.Executors;

@RestController

public class MyController {

@GetMapping("/hello")

public String hello() {

// 创建一个虚拟线程处理请求

Thread.startVirtualThread(() -> {

// 这里是处理请求的逻辑

System.out.println("Handling request in a virtual thread!");

});

return "Request is being processed!";

}

}

```

### 4. 提高通过虚拟线程的并发性能

通过使用虚拟线程，你可以简化并发编程的复杂性，同时允许大量的并发请求而不会造成线程高消耗。

### 5. 附加配置和注意事项

- \*\*调优\*\*：尽管虚拟线程能支持更多的并发，但你仍然需要合理设计和管理线程池，以避免过度消耗资源。

- \*\*兼容性\*\*：确保使用的库或框架支持 Java 19 的新特性，特别是在与非阻塞 I/O 操作时。

- \*\*错误处理\*\*：与普通线程一样，你需要考虑虚拟线程中的异常处理，以便在出现错误时做出适当响应。

### 总结

虚拟线程在 Spring Boot 中的使用，使得并发编程变得更加简单和高效。通过将应用程序的并发模型转向虚拟线程，你可以提高性能并简化代码！有任何具体问题或者需要了解的内容，欢迎随时问我哦！✨

### 5. Batch Transaction Manager

在 Spring Boot 中，批量事务管理（Batch Transaction Management）是处理多个数据库操作的一个重要特性，尤其当你希望在多个操作之间实现一致性时尤为重要。使用 Spring Batch 和 Spring Data JPA，你可以轻松管理批量处理和事务。

### 配置 Spring Batch 进行批量处理

下面是如何在 Spring Boot 项目中配置批量事务的基本步骤：

#### 1. 添加依赖

首先，确保在 `pom.xml` 文件中添加了 Spring Batch 和 Spring Data JPA 的依赖：

```xml

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-batch</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-data-jpa</artifactId>

</dependency>

<dependency>

<groupId>mysql</groupId> <!-- 或其他数据库 -->

<artifactId>mysql-connector-java</artifactId>

<scope>runtime</scope>

</dependency>

```

#### 2. 配置数据源

在 `application.properties` 中配置你的数据源：

```properties

spring.datasource.url=jdbc:mysql://localhost:3306/your\_database

spring.datasource.username=your\_username

spring.datasource.password=your\_password

spring.jpa.hibernate.ddl-auto=update

```

#### 3. 创建 Batch Job

创建一个简单的批处理作业（Job），该作业可以使用 Spring Batch 的 `Job` 和 `Step` 组件。

```java

import org.springframework.batch.core.Job;

import org.springframework.batch.core.Step;

import org.springframework.batch.core.configuration.annotation.EnableBatchProcessing;

import org.springframework.batch.core.configuration.annotation.JobBuilderFactory;

import org.springframework.batch.core.configuration.annotation.StepBuilderFactory;

import org.springframework.batch.item.ItemProcessor;

import org.springframework.batch.item.ItemWriter;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.context.annotation.Bean;

import org.springframework.stereotype.Component;

import java.util.List;

@Component

@EnableBatchProcessing

public class BatchConfig {

@Autowired

public JobBuilderFactory jobBuilderFactory;

@Autowired

public StepBuilderFactory stepBuilderFactory;

@Bean

public Job importUserJob() {

return jobBuilderFactory.get("importUserJob")

.start(step1())

.build();

}

@Bean

public Step step1() {

return stepBuilderFactory.get("step1")

.<User, User>chunk(10)

.reader(userItemReader())

.processor(userItemProcessor())

.writer(userItemWriter())

.build();

}

@Bean

public ItemReader<User> userItemReader() {

// 实现你自己的读取逻辑

}

@Bean

public ItemProcessor<User, User> userItemProcessor() {

return user -> {

// 处理用户数据，例如转换或过滤

return user;

};

}

@Bean

public ItemWriter<User> userItemWriter() {

return users -> {

// 将用户数据写入数据库

// 这个部分将会自动参与事务管理

};

}

}

```

### 4. 使用事务管理

Spring Batch 会自动为每个 `Step` 提供事务管理。可以控制事务的边界，例如在每个 `chunk` 提交时会开启和提交事务。

如果在处理某一 `chunk` 时出现异常，整个 `chunk` 的操作会被回滚，这确保了数据库的内部一致性。你也可以使用 `@Transactional` 注解来手动控制更细粒度的事务。

### 5. 运行 Batch Job

要运行批处理作业，你可以使用 `CommandLineRunner` 或在应用启动时调用作业：

```java

import org.springframework.batch.core.Job;

import org.springframework.batch.core.launch.JobLauncher;

import org.springframework.boot.CommandLineRunner;

import org.springframework.stereotype.Component;

@Component

public class JobLauncherRunner implements CommandLineRunner {

private final JobLauncher jobLauncher;

private final Job importUserJob;

public JobLauncherRunner(JobLauncher jobLauncher, Job importUserJob) {

this.jobLauncher = jobLauncher;

this.importUserJob = importUserJob;

}

@Override

public void run(String... args) throws Exception {

jobLauncher.run(importUserJob, new JobParameters());

}

}

```

### 总结

通过使用 Spring Batch 和 Spring Data JPA，你可以轻松实现批量事务管理，确保在执行多个数据库操作时数据的一致性与可靠性。如果你有任何具体问题或者需要进一步的帮助，欢迎随时问我哦！📊✨

# 6. ssl bundle

Kafka connections can now be configured to use ssl trusted material from ssl bundle

# 7.支持Apache Pulsar

这个是一个Cloud Native ,Distributed messageing and Stream platform

# 8. log升级，支持correlation id

在 Spring Boot 3.2 中，日志记录升级并支持 Correlation ID 是一项重要的增强功能，特别适用于微服务架构和分布式系统。使用 Correlation ID 可以帮助开发者追踪在多个服务之间或在同一服务的不同请求之间的调用链。下面是一些关于如何在 Spring Boot 3.2 中实现和使用 Correlation ID 的信息。

### 1. Correlation ID 的概念

- \*\*定义\*\*：Correlation ID 是一个唯一的标识符，在请求的生命周期中传递，可以用于日志记录和跟踪，以便于问题排查。

### 2. 配置日志框架

在 Spring Boot 3.2 中，可以配置日志框架（如 Logback 或 Log4j2）来支持 Correlation ID。通常可以通过 MDC（Mapped Diagnostic Context）来实现。

#### 示例（使用 Logback）：

1. \*\*添加依赖\*\*（如果还没有添加）：

```xml

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-logging</artifactId>

</dependency>

```

2. \*\*配置 Logback\*\* (`src/main/resources/logback-spring.xml`):

```xml

<configuration>

<appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">

<encoder>

<pattern>%d{yyyy-MM-dd HH:mm:ss} [%thread] %-5level %logger{36} - [%X{X-Correlation-ID}] %msg%n</pattern>

</encoder>

</appender>

<root level="INFO">

<appender-ref ref="STDOUT" />

</root>

</configuration>

```

### 3. 创建过滤器来设置 Correlation ID

可以创建一个 Servlet 过滤器，来从请求中提取或生成 Correlation ID，并将其放入 MDC 中。

#### 示例代码：

```java

import org.slf4j.MDC;

import javax.servlet.Filter;

import javax.servlet.FilterChain;

import javax.servlet.FilterConfig;

import javax.servlet.ServletException;

import javax.servlet.ServletRequest;

import javax.servlet.ServletResponse;

import javax.servlet.http.HttpServletRequest;

import java.io.IOException;

import java.util.UUID;

public class CorrelationIdFilter implements Filter {

@Override

public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)

throws IOException, ServletException {

HttpServletRequest httpRequest = (HttpServletRequest) request;

// 尝试从请求头中获取 Correlation ID

String correlationId = httpRequest.getHeader("X-Correlation-ID");

if (correlationId == null || correlationId.isEmpty()) {

// 如果没有，则生成一个新的 Correlation ID

correlationId = UUID.randomUUID().toString();

}

// 将 Correlation ID 放入 MDC

MDC.put("X-Correlation-ID", correlationId);

try {

chain.doFilter(request, response);

} finally {

// 清理 MDC

MDC.remove("X-Correlation-ID");

}

}

@Override

public void init(FilterConfig filterConfig) throws ServletException {}

@Override

public void destroy() {}

}

```

### 4. 注册过滤器

在 Spring Boot 中，可以使用 `@Component` 注解或者在配置类中注册这个过滤器。

```java

import org.springframework.boot.web.servlet.FilterRegistrationBean;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

@Configuration

public class FilterConfig {

@Bean

public FilterRegistrationBean<CorrelationIdFilter> correlationIdFilter() {

FilterRegistrationBean<CorrelationIdFilter> registrationBean = new FilterRegistrationBean<>();

registrationBean.setFilter(new CorrelationIdFilter());

registrationBean.addUrlPatterns("/\*"); // 适用所有请求

return registrationBean;

}

}

```

### 5. 测试和验证

启动应用程序并发送请求，查看日志输出，确认 Correlation ID 是否在每条日志记录中都能正确显示。

### 总结

通过在 Spring Boot 3.2 中实现 Correlation ID，你可以显著增强日志记录能力，使得在分布式微服务系统中，追踪请求和诊断问题变得更加容易。👍✨

