Spring HTTP 客户端

要回答“如何使用 FeignClient、Spring RestClient 或 Spring WebClient”这一问题，我将解释每个 Spring 工具如何用于发送 HTTP 请求，重点介绍它们的设置和基本用法。这些都是 Spring 生态系统中用于与 RESTful 服务交互的工具，但它们根据应用程序的需求有不同的用途。下面，我将为每个工具提供一个清晰的、分步指南，并附上简单的示例。

---

1. 使用 FeignClient

FeignClient 是 Spring Cloud 提供的一种声明式 REST 客户端。它允许你通过注解定义 HTTP 客户端接口，特别适用于微服务架构，需要调用其他服务的场景。

使用 FeignClient 的步骤

1. **添加依赖**：在项目中包含 Spring Cloud 依赖。如果使用 Maven，将 Spring Cloud starter for Feign 添加到 `pom.xml` 中：

   ```
   <dependency>
       <groupId>org.springframework.cloud</groupId>
       <artifactId>spring-cloud-starter-openfeign</artifactId>
   </dependency>
   ```

   确保你也有一个 Spring Cloud 的依赖管理块，指定兼容的版本。

2. **启用 Feign 客户端**：在主应用程序类或配置类上添加 `@EnableFeignClients` 注解以激活 Feign 支持：

   ```
   import org.springframework.boot.SpringApplication;
   import org.springframework.boot.autoconfigure.SpringBootApplication;
   import org.springframework.cloud.openfeign.EnableFeignClients;

   @SpringBootApplication
   @EnableFeignClients
   public class MyApplication {
       public static void main(String[] args) {
           SpringApplication.run(MyApplication.class, args);
       }
   }
   ```

3. **定义 FeignClient 接口**：创建一个用 `@FeignClient` 注解的接口，指定服务名称或 URL，并定义与 REST 端点对应的方法：

```java
import org.springframework.cloud.openfeign.FeignClient;
import org.springframework.web.bind.annotation.GetMapping;
import java.util.List;

@FeignClient(name = "user-service", url = "http://localhost:8080")
public interface UserClient {
    @GetMapping("/users")
    List<User> getUsers();
}
```

这里，`name` 是客户端的逻辑名称，`url` 是目标服务的基础 URL。`@GetMapping` 注解映射到 `/users` 端点。

4. **注入并使用客户端**：在服务或控制器中自动注入接口，并像调用本地方法一样调用其方法：

```java
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import java.util.List;

@Service
public class UserService {
    @Autowired
    private UserClient userClient;

    public List<User> fetchUsers() {
        return userClient.getUsers();
    }
}
```

关键点

- FeignClient 默认是同步的。
- 它非常适合微服务，特别是使用服务发现（例如 Eureka）时，可以省略 `url` 并让 Spring Cloud 解析它。
- 可以通过回退或断路器（例如 Hystrix 或 Resilience4j）添加错误处理。

---

2. 使用 Spring RestClient

Spring RestClient 是 Spring Framework 6.1 中引入的一种同步 HTTP 客户端，作为废弃的 `RestTemplate` 的现代替代品。它提供了一个流畅的 API 来构建和执行请求。

使用 RestClient 的步骤

1. **依赖**：RestClient 包含在 `spring-web` 中，这是 Spring Boot 的 `spring-boot-starter-web` 的一部分。通常不需要额外的依赖：

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

2. **创建 RestClient 实例**：使用其静态 `create()` 方法实例化 `RestClient`，或者使用构建器进行自定义：

```
import org.springframework.web.client.RestClient;

RestClient restClient = RestClient.create();
```

   对于自定义配置（例如超时），使用 `RestClient.builder()`。

3. **构建并执行请求**：使用流畅的 API 指定 HTTP 方法、URI、头和主体，然后检索响应：

```
import org.springframework.http.MediaType;
import org.springframework.web.client.RestClient;
import java.util.List;

public class UserService {
    private final RestClient restClient;

    public UserService() {
        this.restClient = RestClient.create();
    }

    public List<User> fetchUsers() {
        return restClient.get()
            .uri("http://localhost:8080/users")
            .accept(MediaType.APPLICATION_JSON)
            .retrieve()
            .body(new ParameterizedTypeReference<List<User>>() {});
    }
}
```

   - `get()` 指定 HTTP 方法。
   - `uri()` 设置端点。
   - `accept()` 设置预期的内容类型。
   - `retrieve()` 执行请求，`body()` 提取响应，使用 `ParameterizedTypeReference` 处理泛型类型（例如列表）。

4. **处理响应**：由于 RestClient 是同步的，响应会直接返回。对于更多控制（例如状态代码），使用 `toEntity()`：

```
import org.springframework.http.ResponseEntity;

ResponseEntity<List<User>> response = restClient.get()
    .uri("http://localhost:8080/users")
    .accept(MediaType.APPLICATION_JSON)
    .retrieve()
    .toEntity(new ParameterizedTypeReference<List<User>>() {});
List<User> users = response.getBody();
```

关键点

- RestClient 是同步的，适用于传统的阻塞应用程序。
- 它在 HTTP 错误时抛出异常（例如 `RestClientException`），你可以捕获并处理这些异常。
- 它是 `RestTemplate` 的替代品，具有更直观的 API。

---

3. 使用 Spring WebClient

Spring WebClient 是 Spring WebFlux 中引入的一种反应式、非阻塞 HTTP 客户端。它设计用于异步操作，并与反应式流（Mono 和 Flux）集成。

使用 WebClient 的步骤

1. **添加依赖**：在项目中包含 WebFlux 依赖：

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-webflux</artifactId>
</dependency>
```

2. **创建 WebClient 实例**：使用基础 URL 或默认设置实例化 `WebClient`：

```
import org.springframework.web.reactive.function.client.WebClient;

WebClient webClient = WebClient.create("http://localhost:8080");
```

使用 `WebClient.builder()` 进行自定义配置（例如编解码器、过滤器）。

3. **构建并执行请求**：使用流畅的 API 构建请求并检索反应式响应：

```
import org.springframework.http.MediaType;
import org.springframework.web.reactive.function.client.WebClient;
import reactor.core.publisher.Mono;
import java.util.List;

public class UserService {
    private final WebClient webClient;

    public UserService(WebClient webClient) {
        this.webClient = webClient;
    }

    public Mono<List<User>> fetchUsers() {
        return webClient.get()
            .uri("/users")
            .accept(MediaType.APPLICATION_JSON)
            .retrieve()
            .bodyToFlux(User.class)
            .collectList();
    }
}
```

- `bodyToFlux(User.class)` 处理 `User` 对象的流。
- `collectList()` 将 `Flux<User>` 转换为 `Mono<List<User>>`。

4. **订阅响应**：由于 WebClient 是反应式的，你必须订阅 `Mono` 或 `Flux` 以触发请求：

```
Mono<List<User>> usersMono = fetchUsers();
usersMono.subscribe(users -> System.out.println(users));
```

或者在反应式管道中链接它，或者阻塞（在反应式上下文中不推荐）：

```
List<User> users = fetchUsers().block();
```

关键点

- WebClient 是非阻塞的，适用于基于 Spring WebFlux 的反应式应用程序或处理高并发。
- 使用 `Mono` 处理单值响应，使用 `Flux` 处理多值响应。
- 可以使用操作符（例如 `onErrorResume()` 或 `retry()`）进行错误处理。

比较和何时使用每个

- **FeignClient**：用于微服务设置中的声明式、基于接口的客户端。它是同步的，简化了服务之间的调用。
- **RestClient**：用于传统阻塞 Spring 应用程序中的同步 HTTP 请求。它是 `RestTemplate` 的现代替代品。
- **WebClient**：用于 WebFlux 基础应用程序中的反应式、非阻塞 HTTP 请求或处理高并发。

示例：调用 POST 端点

假设你需要通过调用 `POST /users` 创建一个用户，并传递一个 `User` 对象。

FeignClient

```
@FeignClient(name = "user-service", url = "http://localhost:8080")
public interface UserClient {
    @PostMapping("/users")
    User createUser(@RequestBody User user);
}

@Service
public class UserService {
    @Autowired
    private UserClient userClient;

    public User createUser(User user) {
        return userClient.createUser(user);
    }
}
```

RestClient

```
@Service
public class UserService {
    private final RestClient restClient = RestClient.create();

    public User createUser(User user) {
        return restClient.post()
            .uri("http://localhost:8080/users")
            .contentType(MediaType.APPLICATION_JSON)
```

```
            .body(user)
            .retrieve()
            .body(User.class);
    }
}
```

WebClient

```
@Service
public class UserService {
    private final WebClient webClient;

    public UserService(WebClient webClient) {
        this.webClient = WebClient.create("http://localhost:8080");
    }

    public Mono<User> createUser(User user) {
        return webClient.post()
            .uri("/users")
            .contentType(MediaType.APPLICATION_JSON)
            .bodyValue(user)
            .retrieve()
            .bodyToMono(User.class);
    }
}
```

结论

- **FeignClient**：定义一个接口，启用它，并像本地 bean 一样使用它。
- **RestClient**：流畅地构建请求并同步获取响应。
- **WebClient**：流畅地构建请求，检索反应式流，并异步处理它们。

根据你的应用程序架构选择：FeignClient 用于微服务，RestClient 用于阻塞应用程序，WebClient 用于反应式应用程序。每个工具都遵循上述简单的设置和使用模式。