スプリング HTTP クライアント

クエリ「FeignClient、または Spring RestClient、または Spring WebClient の使い方」に答えるために、これらの Spring ツールを使用して HTTP リクエストを作成する方法を説明します。これらのツールは、すべて Spring エコシステムの一部であり、RESTful サービスとの相互作用に使用されますが、アプリケーションのニーズに応じて異なる目的を果たします。以下に、それぞれのツールの明確なステップバイステップガイドと簡単な例を提供します。

---

1. FeignClient の使用

FeignClient は、Spring Cloud によって提供される宣言的な REST クライアントです。インターフェースにアノテーションを付けることで HTTP クライアントを定義でき、特にマイクロサービスアーキテクチャで他のサービスを呼び出す際に便利です。

FeignClient の使用手順

1. **依存関係の追加**: プロジェクトに Spring Cloud の依存関係を追加します。Maven を使用している場合は、`pom.xml` に Spring Cloud の Starter for Feign を追加します:

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

   Spring Cloud の互換バージョンを指定するための依存関係管理ブロックも確認してください。

2. **Feign クライアントの有効化**: メインアプリケーションクラスまたは設定クラスに `@EnableFeignClients` アノテーションを付けて Feign サポートを有効にします:

   ```
   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. **FeignClient インターフェースの定義**: `@FeignClient` アノテーションを付けたインターフェースを作成し、サービス名または URL を指定し、REST エンドポイントに対応するメソッドを定義します:

```
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();
}
```

ここで、`name` はクライアントの論理名であり、`url` はターゲットサービスのベース URL です。`@GetMapping` アノテーションは`/users` エンドポイントにマッピングされます。

4. **クライアントのインジェクトと使用**: サービスまたはコントローラーでインターフェースを自動ワイヤリングし、ローカルメソッドのように呼び出します:

```
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();
    }
}
```

重要ポイント

- FeignClient はデフォルトで同期です。
- サービスディスカバリー（例：Eureka）を使用するマイクロサービスで、`url` を省略し、Spring Cloud が解決するようにするのが理想的です。
- フォールバックやサーキットブレイカー（例：Hystrix または Resilience4j）を使用してエラーハンドリングを追加できます。

---

2. Spring RestClient の使用

Spring RestClient は、Spring フレームワーク 6.1 で導入された同期 HTTP クライアントで、非推奨の `RestTemplate` の現代的な代替手段です。フルエント API を提供してリクエストの構築と実行を簡素化します。

RestClient の使用手順

1. **依存関係**: RestClient は `spring-web` に含まれており、Spring Boot の `spring-boot-starter-web` の一部です。通常、追加の依存関係は必要ありません:

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

2. **RestClient インスタンスの作成**: `RestClient` の静的 `create()` メソッドを使用してインスタンスを作成するか、ビルダーを使用してカスタマイズします:

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

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

   カスタム設定（例：タイムアウト）には `RestClient.builder()` を使用します。

3. **リクエストの構築と実行**: フルエント API を使用して HTTP メソッド、URI、ヘッダー、ボディを指定し、応答を取得します:

```
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()` は HTTP メソッドを指定します。
- `uri()` はエンドポイントを設定します。
- `accept()` は期待されるコンテンツタイプを設定します。
- `retrieve()` はリクエストを実行し、`body()` は応答を抽出し、ジェネリックタイプ（例：リスト）に対して `ParameterizedTypeReference` を使用します。

4. **応答の処理**: RestClient は同期なので、応答は直接返されます。より多くの制御（例：ステータスコード）が必要な場合は、`toEntity()` を使用します:

```
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();
```

重要ポイント

- RestClient は同期なので、伝統的なブロッキングアプリケーションに適しています。
- HTTP エラーが発生した場合に例外（例：`RestClientException`）をスローし、キャッチして処理できます。
- `RestTemplate` の代替手段で、より直感的な API を提供します。

---

3. Spring WebClient の使用

Spring WebClient は、Spring WebFlux で導入された非同期、非ブロッキング HTTP クライアントです。非同期操作に設計されており、リアクティブストリーム（Mono と Flux）と統合されています。

WebClient の使用手順

1. **依存関係の追加**: プロジェクトに WebFlux の依存関係を追加します:

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

2. **WebClient インスタンスの作成**: ベース URL またはデフォルト設定を使用して `WebClient` のインスタンスを作成します:

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

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

カスタム設定（例：コーデック、フィルター）には `WebClient.builder()` を使用します。

3. **リクエストの構築と実行**: フルエント API を使用してリクエストを構築し、リアクティブ応答を取得します:

```
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)` は `User` オブジェクトのストリームを処理します。
- `collectList()` は `Flux<User>` を `Mono<List<User>>` に変換します。

4. **応答のサブスクライブ**: WebClient はリアクティブなので、`Mono` または `Flux` にサブスクライブしてリクエストをトリガーする必要があります:

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

または、リアクティブパイプラインにチェーンするか、ブロック（リアクティブコンテキストでは推奨されません）します:

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

重要ポイント

- WebClient は非ブロッキングで、Spring WebFlux ベースのリアクティブアプリケーションや高い並行性を扱う際に適しています。
- 単一値応答には `Mono` を、複数値応答には `Flux` を使用します。
- エラーハンドリングには `onErrorResume()` や `retry()` などの演算子を使用します。

比較と使用時のポイント

- **FeignClient**: マイクロサービス設定でインターフェースベースの宣言的クライアントを使用します。同期で、サービス間の呼び出しを簡素化します。
- **RestClient**: 伝統的なブロッキング Spring アプリケーションでの同期 HTTP リクエストに使用します。`RestTemplate` の現代的な代替手段です。
- **WebClient**: WebFlux ベースのアプリケーションや高い並行性を扱う際の非同期、非ブロッキング HTTP リクエストに使用します。

例: POST エンドポイントの呼び出し

`POST /users` に `User` オブジェクトを送信してユーザーを作成する必要があるとします。

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);
    }
}
```

まとめ

- **FeignClient**: インターフェースを定義し、有効にし、ローカルビーンのように使用します。
- **RestClient**: フルエント API を使用してリクエストを構築し、応答を同期的に取得します。
- **WebClient**: フルエント API を使用してリクエストを構築し、リアクティブストリームを取得し、非同期で処理します。

アプリケーションのアーキテクチャに応じて選択してください: マイクロサービスには FeignClient、ブロッキングアプリには RestClient、リアクティブアプリには WebClient を使用します。それぞれの設定と使用パターンは上記のように簡単です。