解决冗余代码的3种方法，让你的代码更上一层楼

原创 JAVA旭阳 JAVA旭阳 2023-02-28 13:10 发表于浙江

前言

冗余代码向来是代码的一种坏味道，也是我们程序员要极力避免的。今天我通过一个示例和大家分享下解决冗余代码的3个手段，看看哪个最好。

问题描述

为了描述这个问题，我将使用 `FtpClient` 作为示例。要从 ftp 服务器获取一些文件，你需要先建立连接，下一步是登录，然后执行查看ftp文件列表、删除ftp文件，最后注销并断开连接， 代码如下：

```
public class FtpProvider{

    private final FTPClient ftpClient;

    public FTPFile[] listDirectories(String parentDirectory) {
        try {
            ftpClient.connect("host", 22);
            ftpClient.login("username", "password");
            return ftpClient.listDirectories(parentDirectory);
        } catch (IOException ex) {
            log.error("Something went wrong", ex);
            throw new RuntimeException(ex);
        } finally {
            try {
                ftpClient.logout();
                ftpClient.disconnect();
            } catch (IOException ex) {
                log.error("Something went wrong while finally", ex);
            }
        }
    }

    public boolean deleteFile(String filePath) {
        try {
            ftpClient.connect("host", 22);
            ftpClient.login("username", "password");
            return ftpClient.deleteFile(filePath);
        } catch (IOException ex) {
            log.error("Something went wrong", ex);
            throw new RuntimeException(ex);
        } finally {
           try {
                ftpClient.logout();
                ftpClient.disconnect();
            } catch (IOException ex) {
                log.error("Something went wrong while finally", ex);
            }
        }
    }
}
```

正如上面代码所示， `listDirectories` 和 `downloadFtpFile` 中都包含了ftp连接、登录以及最后的注销操作，存在大量冗余的代码，那有什么更好的办法清理冗余代码呢？下面推荐3个做法，所有三个提出的解决方案都将实现以下 `FtpProvider` 接口，我将比较这些实现并选择更好的一个。

```
public interface FtpProvider {

    FTPFile[] listDirectories(String directory) throws IOException;

    boolean deleteFile(String filePath) throws IOException;
}
```

1. 使用@Aspect 代理

- 首先创建一个注解,用来注解需要代理的方法

```
@Target({ElementType.TYPE, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
public @interface FtpOperation {
}
```

- 创建一个类实现 `FtpProvider` 接口,将注解添加到方法 `listDirectories` 和 `deleteFile` 中

```
@Slf4j
@Service
class FtpProviderImpl implements FtpProvider {

    private final FTPClient ftpClient;

    @Override
    public FTPFile[] listDirectories(String directory) throws IOException {
        return ftpClient.listDirectories(directory);
    }

    @Override
    public boolean deleteFile(String filePath) throws IOException {
       return ftpClient.deleteFile(filePath);
    }
}
```

- 实现注解的代理切面逻辑

```
@Slf4j
@Aspect
@Component
@RequiredArgsConstructor
public class FtpOperationProxy {

    private final FTPClient ftpClient;

    @Around("@annotation(daniel.zielinski.redundancy.proxyaop.infrastructure.FtpOperation)")
    public Object handle(ProceedingJoinPoint joinPoint) throws Throwable {
        try {
            ftpClient.connect("host", 22);
            ftpClient.login("username", "password");
            return joinPoint.proceed();
        } catch (IOException ex) {
            log.error("Something went wrong", ex);
            throw new RuntimeException(ex);
        } finally {
            try {
                ftpClient.logout();
                ftpClient.disconnect();
            } catch (IOException ex) {
                log.error("Something went wrong while finally", ex);
            }
        }
    }
}
```

所有用 `@FtpOperation` 注解的方法都会在这个地方执行 `joinPoint.proceed()` 。

2. 函数式接口

- 创建一个函数式接口

```
@FunctionalInterface
interface FtpOperation<T, R> {

     R apply(T t) throws IOException;
}
```

- 定义ftp执行模板

```java
@RequiredArgsConstructor
@Slf4j
@Service
public class FtpOperationTemplate {

    private final FTPClient ftpClient;

    public <K> K execute(FtpOperation<FTPClient, K> ftpOperation) {
        try {
            ftpClient.connect("host", 22);
            ftpClient.login("username", "password");
            return ftpOperation.apply(ftpClient);
        } catch (IOException ex) {
            log.error("Something went wrong", ex);
            throw new RuntimeException(ex);
        } finally {
            try {
                ftpClient.logout();
                ftpClient.disconnect();
            } catch (IOException ex) {
                log.error("Something went wrong while finally", ex);
            }
        }
    }

}
```

- 定义实现类

```java
@RequiredArgsConstructor
@Slf4j
@Service
class FtpProviderFunctionalInterfaceImpl implements FtpProvider {

    private final FtpOperationTemplate ftpOperationTemplate;

    public FTPFile[] listDirectories(String parentDirectory) {
        return ftpOperationTemplate.execute(ftpClient -> ftpClient.listDirectories(parentDirec
    }

    public boolean deleteFile(String filePath) {
        return ftpOperationTemplate.execute(ftpClient -> ftpClient.deleteFile(filePath));
    }
}
```

我们正在 `FtpOperationTemplate` 上执行方法 `execute` 并且我们正在传递 `lambda` 表达式。我们将放入 `lambda` 中的所有逻辑都将代替 `ftpOperation.apply(ftpClient)` 函数执行。

3. 模板方法

- 创建一个抽象的模板类

```java
@RequiredArgsConstructor
@Slf4j
@Service
abstract class FtpOperationTemplate<T, K> {

    protected abstract K command(FTPClient ftpClient, T input) throws IOException;

    public K execute(FTPClient ftpClient, T input) {
        try {
            ftpClient.connect("host", 22);
            ftpClient.login("username", "password");
            return command(ftpClient, input);
        } catch (IOException ex) {
            log.error("Something went wrong", ex);
            throw new RuntimeException(ex);
        } finally {
            try {
                ftpClient.logout();
                ftpClient.disconnect();
            } catch (IOException ex) {
                log.error("Something went wrong while finally", ex);
            }
        }
    }

}
```

- 列出ftp目录 `listDirectories` 方法的实现

```java
```

```
@Slf4j
@Service
class FtpOperationListDirectories extends FtpOperationTemplate<String, FTPFile[]> {

    @Override
    protected FTPFile[] command(FTPClient ftpClient, String input) throws IOException {
        return ftpClient.listDirectories(input);
    }
}
```

- 删除文件deleteFile方法的实现

```
@Slf4j
@Service
class FtpOperationDeleteFile extends FtpOperationTemplate<String, Boolean> {

    @Override
    protected Boolean command(FTPClient ftpClient, String input) throws IOException {
        return ftpClient.deleteFile(input);
    }
}
```

- 实现 `FtpProvider` 接口

```
@RequiredArgsConstructor
@Slf4j
@Service
public class FtpProviderTemplateImpl implements FtpProvider {

    private final FtpOperationTemplate<String, FTPFile[]> ftpOperationListDirectories;
    private final FtpOperationTemplate<String, Boolean> ftpOperationDeleteFile;
    private final FTPClient ftpClient;

    public FTPFile[] listDirectories(String parentDirectory) {
        return ftpOperationListDirectories.execute(ftpClient, parentDirectory);
    }

    public boolean deleteFile(String filePath) {
        return ftpOperationDeleteFile.execute(ftpClient, filePath);
    }
}
```

我们正在 `FtpOperationTemplate` 上执行方法 `execute` 并在那里传递我们的参数。因此执行方法的逻辑对于 `FtpOperationTemplate` 的每个实现都是不同的。

总结

我们现在来比较下上面种方式：

- **@Aspect切面方式实现**

向 `FtpProvider` 接口添加一个新方法，需要我们仅在一个地方进行更改。我们可以轻松地将我们的 `FtpProvider` 注入到其他服务中。此解决方案的强项可能是 `@FtpOperation` 注释，它可以在 `FtpProvider` 上下文实现之外使用，但是将 Ftp 操作的逻辑划分到单独的类中并不是一个好方法。

- **函数式接口实现**

向接口 `FtpProvider` 添加一个新方法，需要我们仅在一个地方进行更改。我们可以轻松地将我们的 `FtpProvider` 注入到其他服务中。我们将ftp操作的逻辑封装在一个类中。相对于上面的方式，我们也没有用到AOP的库，所以我个人还是比较推荐的。

- **模板方法实现**

向接口 `FtpProvider` 添加一个新方法，需要我们在两个地方进行更改。我们需要添加一个新的类，会导致类爆炸，另外，我们还需要将实现注入到 `FtpProvider` 。

如果是你，你会选择哪种方式呢？还是有更好的方法？

收录于合集 #架构设计 16

‹ 上一篇
关于服务限流这回事，总算整明白了

下一篇 ›
想成为大牛，不得不懂的5种Linux网络IO模型

喜欢此内容的人还喜欢

8 种 SQL 写法，性能降低 100 倍，不来看看?
潮汕IT智库

Python 高手必会 20 个单行代码
全栈 派

写了这么多年代码，我总结的DTO最佳实践
JAVA旭阳