**Design Patterns: Adapter Pattern in TypeScript**

Easily Solve Interface Incompatibility Problems Using Adapter Pattern

Welcome to the **Design Patterns in TypeScript** series, which introduces some useful design patterns in web development using TypeScript.

Previous articles are as follows:

* [Strategy Pattern in TypeScript](https://javascript.plainenglish.io/design-patterns-strategy-pattern-in-typescript-54eda9b40f09)
* [Chain of Responsibility Pattern in TypeScript](https://javascript.plainenglish.io/design-patterns-chain-of-responsibility-pattern-in-typescript-dba6bdffe456)
* [Observer Pattern in TypeScript](https://javascript.plainenglish.io/design-patterns-observer-pattern-in-typescript-f6589f1ce4fc)
* [Template Method Pattern in TypeScript](https://javascript.plainenglish.io/design-patterns-template-method-pattern-in-typescript-ce0c8b158985)
* [Adapter Pattern in TypeScript](https://javascript.plainenglish.io/design-patterns-adapter-pattern-in-typescript-4b7ad3c1c234)
* [Factory Method Pattern in TypeScript](https://javascript.plainenglish.io/design-patterns-factory-method-pattern-in-typescript-c4c3047a6289)
* [Abstract Factory Pattern in TypeScript](https://javascript.plainenglish.io/design-patterns-abstract-factory-pattern-in-typescript-84cd7b002964)

Design patterns are very important for web developers and we can write better code by mastering them. In this article, I will use **TypeScript** to introduce the **Adapter Pattern.**

In the web system, the mail service is a very commonly used service. On the Node.js platform, we can use a [nodemailer](https://github.com/nodemailer/nodemailer" \t "_blank) module to easily implement the function of sending emails. After successfully installing the [nodemailer](https://github.com/nodemailer/nodemailer" \t "_blank) module, you can follow the steps below to send emails:

let transporter = nodemailer.createTransport(transport[, defaults]);  
transporter.sendMail(data[, callback])

In order to avoid binding the mail service to a specific service provider, before developing the mail service, we first define the interface related to the mail provider:

interface EmailProvider {  
 sendMail(options: EmailOptions): Promise<EmailResponse>;  
}interface EmailOptions {  
 to: string | string[];  
 subject: string;  
 html: string;  
 from?: string;  
 text?: string;  
}interface EmailResponse {}

With these interfaces, we can easily create a mail service:

class EmailService {  
 constructor(public emailProvider: EmailProvider) {} async sendMail(options: EmailOptions): Promise<EmailResponse> {  
 const result = await this.emailProvider.sendMail(options);  
 return result;  
 }  
}

At present, this solution is not a big problem, but if one day we need to use a third-party email cloud service provider. Such as **sendgrid** or **mailersend** etc. You will find the name of the method the SDK uses to send mail is send. So we go ahead and define a CloudEmailProvider interface:

interface CloudEmailProvider {  
 send(options: EmailOptions): Promise<EmailResponse>;  
}

Comparing the previously defined EmailProvider interface, you will find the following problem:

Therefore, we cannot directly use EmailService to access third-party email cloud services. To solve this problem, there are many ways. Let’s introduce how to use the adapter pattern to solve the above problem.

The purpose of the adapter pattern is to allow two objects that would otherwise not work together due to mismatched interfaces to work together. It is like glue, transforming different things so that they can work together.

The adapter pattern contains the following roles:

* **Client(EmailService)**: Object that needs to use the Target interface;
* **Target(EmailProvider)**: Define the interface expected by the client;
* **Adapter(CloudEmailAdapter)**: Adapt the Adaptee interface to the Target interface;
* **Adaptee(CloudEmailProvider)**: Define the interface that needs to be adapted.

After learning about the adapter pattern, let’s create the CloudEmailAdapterclass:

class CloudEmailAdapter implements EmailProvider {  
 constructor(public emailProvider: CloudEmailProvider) {} async sendMail(options: EmailOptions): Promise<EmailResponse> {  
 const result = this.emailProvider.send(options);  
 return result;  
 }  
}

In the above code, because the two interfaces of EmailProviderand CloudEmailProvider do not match, we introduce the CloudEmailAdapter class to solve the compatibility problem.

Next, we take sendgrid as an example to implement a SendgridEmailProvider:

import { MailService } from "[@sendgrid/mail](http://twitter.com/sendgrid/mail)";class SendgridEmailProvider implements CloudEmailProvider {  
 private sendgridMail: MailService; constructor(  
 private config: {  
 apiKey: string;  
 from: string;  
 }  
 ) {  
 this.sendgridMail = new MailService();  
 this.sendgridMail.setApiKey(this.config.apiKey);  
 } async send(options: EmailOptions): Promise<EmailResponse> {  
 const result = await this.sendgridMail.send(options);  
 return result;  
 }  
}

*Tip: The above code is only for demonstration purposes, and needs to be adjusted accordingly when used in actual projects.*

Now that the SendgridEmailProvider and CloudEmailAdapter classes are defined, let’s see how to use them:

const sendgridMail = new SendgridEmailProvider({  
 apiKey: "\*\*\*\*\*\*",  
 from: "[bytefer@gmail.com](mailto:bytefer@gmail.com)",  
});const cloudEmailAdapter = new CloudEmailAdapter(sendgridMail);  
const emailService = new EmailService(cloudEmailAdapter);emailService.sendMail({  
 to: "\*\*\*\*\*\*",  
 subject: "Adapter Design Pattern",  
 html: "<h3>Adapter Design Pattern</h3>",  
 from: "[bytefer@gmail.com](mailto:bytefer@gmail.com)",  
});

Finally, let’s summarize the usage scenarios of the adapter pattern:

* The system needs to use an existing class, and the interface of this class does not meet the needs of the system, that is, the interface is not compatible;
* Use the service provided by the third party, but the service interface definition is different from the interface definition required by yourself.

If you have any questions, please feel free to leave me a message. I will continue to introduce other patterns later, if you are interested, you can follow me on [Medium](https://medium.com/@bytefer) or [Twitter](https://twitter.com/Tbytefer).