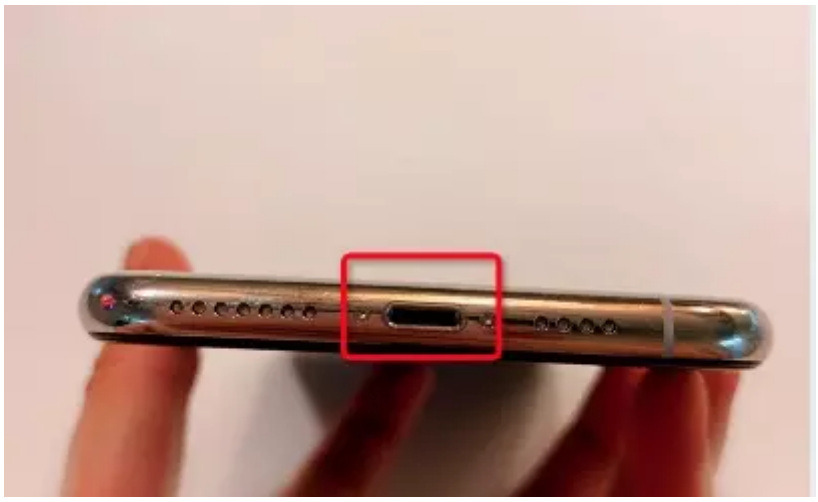
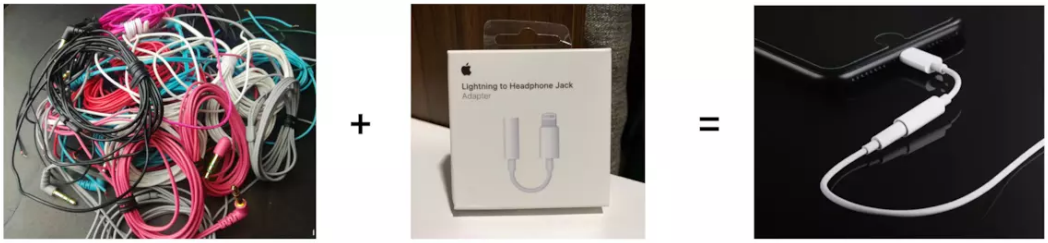
适配器模式通过**把一个类的接口变换成客户端所期待的另一种接口**，可以帮我们解决**不兼容**的问题。

生活中的适配器

前段时间用了很久的 iPhone 6s丢了，请假跑出去买了台 iPhone X。结果有天听歌的时候发现X的耳机孔竟然是方形的，长这样：





图一　　图二　　图三

而重度 iPhone 6s 用户&耳机发烧友的耳机线，可能是如图一所示，没错，它们都是圆头耳机，意识到这一点的时候，我佛了。

此时我好像只能在重新买一批耳机（很有可能同款耳机并没有方头的款式）和重新买一台手机之间做选择了。好在我不是一个普通的倒霉蛋，我学过设计模式，设计模式告诉我这种**实际接口与目标接口不匹配的尴尬**可以用一个叫**适配器**的东西来化解。打开万能的淘宝一搜，还真有，如图二所示。

只要装上它，圆头耳机就可以完美适配方形插槽，最终效果如图三所示。

绝了，正合我意！赶紧买一个来救火，从此又可以开心地为圆头耳机氪金啦~

大家现在回顾楼上这波操作，这个耳机转换头做的事情，是不是就是我们开头说的把一个 `类` （iPhone 新机型）的 `接口` （方形）变换成 `客户端` （用户）所期待的另一种 `接口` （圆形）？

最终达到的效果，就是 `用户` （我）可以像使用 iPhone 6s 插口一样使用 iPhoneX 的插口，而不用感知两者间的差异。我们设计模式中的适配器，和楼上这个适配器做的事情可以说是一模一样，同样具有化腐朽为神奇的力量。

兼容接口就是一把梭——适配器的业务场景

大家知道我们现在有一个非常好用异步方案叫fetch，它的写法比ajax优雅很多。因此在不考虑兼容性的情况下，我们更愿意使用fetch、而不是使用ajax来发起异步请求。李雷是拜fetch教的忠实信徒，为了能更好地使用fetch，他封装了一个基于fetch的http方法库：

```
export default class HttpUtils {
  // get方法
  static get(url) {
    return new Promise((resolve, reject) => {
      // 调用fetch
      fetch(url)
        .then(response => response.json())
        .then(result => {
          resolve(result)
        })
        .catch(error => {
          reject(error)
        })
    })
  }

  // post方法，data以object形式传入
  static post(url, data) {
    return new Promise((resolve, reject) => {
      // 调用fetch
      fetch(url, {
        method: 'POST',
        headers: {
          Accept: 'application/json',
          'Content-Type': 'application/x-www-form-urlencoded'
        },
        // 将object类型的数据格式化为合法的body参数
        body: this.changeData(data)
      })
        .then(response => response.json())
        .then(result => {
          resolve(result)
        })
        .catch(error => {
          reject(error)
        })
    })
  }

  // body请求体的格式化方法
  static changeData(obj) {
    var prop,
      str = ''
    var i = 0
    for (prop in obj) {
      if (!prop) {
        return
      }
      if (i == 0) {
        str += prop + '=' + obj[prop]
```

```
      } else {
        str += '&' + prop + '=' + obj[prop]
      }
      i++
    }
    return str
  }
}
```

当我想使用 fetch 发起请求时，只需要这样轻松地调用，而不必再操心繁琐的数据配置和数据格式化：

```
// 定义目标url地址
const URL = "xxxxx"
// 定义post入参
const params = {
    ...
}

// 发起post请求
 const postResponse = await HttpUtils.post(URL,params) || {}

 // 发起get请求
 const getResponse = await HttpUtils.get(URL)
```

真是个好用的方法库！老板看了李雷的 HttpUtils 库，喜上眉梢——原来老板也是个拜 fetch 教。老板说李雷，咱们公司以后要做潮流公司了，写代码不再考虑兼容性，我希望你能**把公司所有的业务的网络请求都迁移到你这个 HttpUtils 上来**，这样以后你只用维护这一个库了，也方便。李雷一听，悲从中来——他是该公司的第 99 代员工，对远古时期的业务一无所知。而该公司第1代员工封装的网络请求库，是基于 XMLHttpRequest 的，差不多长这样：

```
function Ajax(type, url, data, success, failed){
    // 创建ajax对象
    var xhr = null;
    if(window.XMLHttpRequest){
        xhr = new XMLHttpRequest();
    } else {
        xhr = new ActiveXObject('Microsoft.XMLHTTP')
    }

   ...(此处省略一系列的业务逻辑细节)

   var type = type.toUpperCase();

    // 识别请求类型
    if(type == 'GET'){
        if(data){
          xhr.open('GET', url + '?' + data, true); //如果有数据就拼接
        }
        // 发送get请求
        xhr.send();

    } else if(type == 'POST'){
        xhr.open('POST', url, true);
        // 如果需要像 html 表单那样 POST 数据，使用 setRequestHeader() 来添加 http 头。
        xhr.setRequestHeader("Content-type", "application/x-www-form-urlencoded");
        // 发送post请求
        xhr.send(data);
    }
```

```
    // 处理返回数据
    xhr.onreadystatechange = function(){
        if(xhr.readyState == 4){
            if(xhr.status == 200){
                success(xhr.responseText);
            } else {
                if(failed){
                    failed(xhr.status);
                }
            }
        }
    }
}
```

实现逻辑我们简单描述了一下，这个不是重点，重点是它是这样调用的：

```
// 发送get请求
Ajax('get', url地址, post入参, function(data){
    // 成功的回调逻辑
}, function(error){
    // 失败的回调逻辑
})
```

李雷佛了 —— 不仅接口名不同，入参方式也不一样，这手动改要改到何年何日呢？

还好李雷学过设计模式，他立刻联想到了专门为我们**抹平差异**的适配器模式。要把老代码迁移到新接口，不一定要挨个儿去修改每一次的接口调用——正如我们想用 iPhoneX + 旧耳机听歌，不必挨个儿去改造耳机一样，我们只需要在引入接口时进行**一次适配**，便可轻松地 cover 掉业务里可能会有的**多次调用**（具体的解析在注释里）：

```
// Ajax适配器函数，入参与旧接口保持一致
async function AjaxAdapter(type, url, data, success, failed) {
    const type = type.toUpperCase()
    let result
    try {
         // 实际的请求全部由新接口发起
         if(type === 'GET') {
            result = await HttpUtils.get(url) || {}
        } else if(type === 'POST') {
            result = await HttpUtils.post(url, data) || {}
        }
        // 假设请求成功对应的状态码是1
        result.statusCode === 1 && success ? success(result) :
failed(result.statusCode)
    } catch(error) {
        // 捕捉网络错误
        if(failed){
            failed(error.statusCode);
        }
    }
}

// 用适配器适配旧的Ajax方法
async function Ajax(type, url, data, success, failed) {
    await AjaxAdapter(type, url, data, success, failed)
}
```

如此一来，我们只需要编写一个适配器函数AjaxAdapter，并用适配器去承接旧接口的参数，就可以实现新旧接口的无缝衔接了~

生产实践：axios中的适配器

数月之后，李雷的老板发现了网络请求神库axios，于是团队的方案又整个迁移到了axios——对于心中有适配器的李雷来说，这现在已经根本不是个事儿。不过本小节我们要聊的可不再是“如何使现有接口兼容axios”了（这招我们上个小节学过了）。此处引出axios，一是因为大家对它足够熟悉（不熟悉的同学，点**这里**可以快速熟悉一下~），二是因为axios本身就用到了我们的**适配器模式**，它的兼容方案值得我们学习和借鉴。 在使用axios时，作为用户我们只需要掌握以下面三个最常用的接口为代表的一套api：

```
// Make a request for a user with a given ID
axios.get('/user?ID=12345')
  .then(function (response) {
    // handle success
    console.log(response);
  })
  .catch(function (error) {
    // handle error
    console.log(error);
  })
  .then(function () {
    // always executed
  })

axios.post('/user', {
    firstName: 'Fred',
    lastName: 'Flintstone'
  })
  .then(function (response) {
    console.log(response);
  })
  .catch(function (error) {
    console.log(error);
  });

axios({
  method: 'post',
  url: '/user/12345',
  data: {
    firstName: 'Fred',
    lastName: 'Flintstone'
  }
})
```

便可轻松地发起各种姿势的网络请求，而不用去关心底层的实现细节。 除了简明优雅的api之外，axios强大的地方还在于，它不仅仅是一个局限于浏览器端的库。在Node环境下，我们尝试调用上面的 api，会发现它照样好使 —— axios 完美地**抹平了两种环境下api的调用差异**，靠的正是对适配器模式的灵活运用。

在 **axios 的核心逻辑** 中，我们可以注意到实际上派发请求的是 **dispatchRequest 方法**。该方法内部其实主要做了两件事：

1. 数据转换，转换请求体/响应体，可以理解为数据层面的适配；

2. 调用适配器。

调用适配器的逻辑如下：

```
// 若用户未手动配置适配器，则使用默认的适配器
var adapter = config.adapter || defaults.adapter;

  // dispatchRequest方法的末尾调用的是适配器方法
  return adapter(config).then(function onAdapterResolution(response) {
    // 请求成功的回调
    throwIfCancellationRequested(config);

    // 转换响应体
    response.data = transformData(
      response.data,
      response.headers,
      config.transformResponse
    );

    return response;
  }, function onAdapterRejection(reason) {
    // 请求失败的回调
    if (!isCancel(reason)) {
      throwIfCancellationRequested(config);

      // 转换响应体
      if (reason && reason.response) {
        reason.response.data = transformData(
          reason.response.data,
          reason.response.headers,
          config.transformResponse
        );
      }
    }

    return Promise.reject(reason);
  });
```

大家注意注释的第一行，“若用户未手动配置适配器，则使用默认的适配器”。手动配置适配器允许我们自定义处理请求，主要目的是为了使测试更轻松。

实际开发中，我们使用默认适配器的频率更高。默认适配器在 `axios/lib/default.js` 里是通过 `getDefaultAdapter` 方法来获取的：

```
function getDefaultAdapter() {
  var adapter;
  // 判断当前是否是node环境
  if (typeof process !== 'undefined' && Object.prototype.toString.call(process) ===
'[object process]') {
    // 如果是node环境，调用node专属的http适配器
    adapter = require('./adapters/http');
  } else if (typeof XMLHttpRequest !== 'undefined') {
    // 如果是浏览器环境，调用基于xhr的适配器
    adapter = require('./adapters/xhr');
  }
  return adapter;
}
```

我们再来看看 Node 的 http 适配器和 xhr 适配器大概长啥样：

> http 适配器：

```
module.exports = function httpAdapter(config) {
  return new Promise(function dispatchHttpRequest(resolvePromise, rejectPromise) {
    // 具体逻辑
  }
}
```

> xhr 适配器：

```
module.exports = function xhrAdapter(config) {
  return new Promise(function dispatchXhrRequest(resolve, reject) {
    // 具体逻辑
  }
}
```

具体逻辑啥样，咱们目前先不关心，有兴趣的同学，可以狠狠地点**这里**阅读源码。咱们现在就注意两个事儿：

- 两个适配器的入参都是 config；
- 两个适配器的出参都是一个 Promise。

> Tips：要是仔细读了源码，会发现两个适配器中的 Promise 的内部结构也是如出一辙。

这么一来，通过 axios 发起跨平台的网络请求，不仅调用的接口名是同一个，连入参、出参的格式都只需要掌握同一套。这导致它的学习成本非常低，开发者看了文档就能上手；同时因为足够简单，在使用的过程中也不容易出错，带来了极佳的用户体验，axios 也因此越来越流行。

这正是一个好的适配器的自我修养——把变化留给自己，把统一留给用户。在此处，所有关于 http 模块、关于 xhr 的实现细节，全部被 Adapter 封装进了自己复杂的底层逻辑里，暴露给用户的都是十分简单的统一的东西——统一的接口，统一的入参，统一的出参，统一的规则。用起来就是一个字 —— 爽！

小结

本节我们除了针对适配器的原理、实践及应用场景进行讨论之外，还花了不少力气来讲 axios。这个操作可能会使一部分不太熟悉 axios 的同学阅读起来更加吃力——因为要想读懂这一节，你或许不得不点开我穿插进去的源码/文档链接先去尝试理解 axios —— 但这其实正是我想鼓励大家去做的事情。

在**性能小册**的开篇，我说过，希望大家都能去读“纸的背面”。这个“纸的背面”不仅仅是说代码之外的东西，它也可以是一些超越这本书的东西 —— 楼上吹了那么多 axios 的“彩虹屁”，难道本节是 axios 大型夸夸群现场吗？难道 axios 真的完美无缺，无可替代吗？不是的。

笔者洋洋洒洒这么多字，无非是希望给大家打开一个窗口 —— 在过去半年多和读者有直接沟通的这些时间里，我知道很多同学是不读源码的。这个“不读”不一定是不想读，可能只是不敢读，或者说读不动。无论是出于什么原因，在这里我都想告诉大家，开卷有益，源码是非常好的学习材料，它能教会你的东西，比你想象中多得多。

适配器模式的思想可以说是遍地开花，稍微多看几个库，你会发现不仅 axios 在用适配器，其它库也在用。如果哪怕只有一个同学因为今天读了这一节，对这个“看起来很厉害”的 axios 产生了好奇，或者说对读源码这件事情萌生了兴趣、进而刻意地去培养了自己的阅读习惯，那么你在繁忙的工作/学业中抽出的宝贵的用来阅读这一节内容的时间就没有白费，这本小册也算不负使命、远远大于它本身的价值

了。

设计模式这座山，诸位已经翻过了半山腰。剩下的路，一起加油！

（阅读过程中有任何想法或疑问，或者单纯希望和笔者交个朋友啥的，欢迎大家添加我的微信xyalinode与我交流哈~）