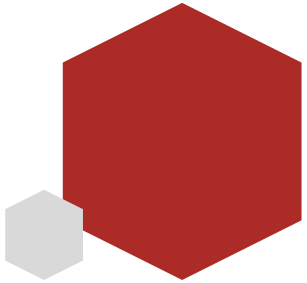
JS原理课

黑马程序员
www.itheima.com

传智教育旗下
高端IT教育品牌



JavaScript中的this

JavaScript中的this

在绝大多数情况下，函数的调用方式决定了 this 的值（运行时绑定）。this 不能在执行期间被赋值，并且在每次函数被调用时 this 的值也可能会不同。

1. 如何确认this的值:
2. 如何指定this的值:
3. 手写call、apply 、 bind:

如何确认this的值:

在非严格模式下，总是指向一个对象，在严格模式下可以是任意值。

1. 全局执行环境中,指向全局对象(非严格模式、严格模式)
2. 函数内部,取决于函数被调用的方式
    1. 直接调用的this值:
        ① 非严格模式:全局对象(window)
        ② 严格模式:undefined
    2. 对象方法调用的this值:
        ① 调用者

```
// 为整个脚本开启严格模式
'use strict'

function func() {
  // 为函数开启严格模式
  'use strict'
}
```

JavaScript中的this

在绝大多数情况下，函数的调用方式决定了 this 的值（运行时绑定）。this 不能在执行期间被赋值，并且在每次函数被调用时 this 的值也可能会不同。

1. 如何确认this的值:
2. 如何指定this的值:
3. 手写call、apply 、 bind:

如何指定this的值：

可以通过2类方法指定this：

1. 调用时指定：
    1. call方法
    2. apply方法
2. 创建时指定：
    1. bind方法
    2. 箭头函数

```
func.call(thisArg, 参数1, 参数2...)

func.apply(thisArg, [参数1, 参数2...])
```

```
const bindFunc = func.bind(thisArg, 绑定参数1, 绑定参数2...)

const itheima = {
  name: '播仔',
  eat() {
    setTimeout(() => { console.log(this) })
  }
}
```

JavaScript中的this

在绝大多数情况下，函数的调用方式决定了 this 的值（运行时绑定）。this 不能在执行期间被赋值，并且在每次函数被调用时 this 的值也可能会不同。

1. 如何确认this的值:
2. 如何指定this的值:
3. 手写call、apply 、 bind:

手写call、apply、bind:

需求:实现myCall方法,功能和调用形式和call一致

- 定义myCall方法
- 设置this并调用原函数
- 接收剩余参数并返回结果
- 使用Symbol调优

```
const person = {
  name: 'itheima'
}
function func(numA, numB) {
  console.log(this)
  console.log(numA, numB)
  return numA + numB
}

// 调用并获取返回值
const res = func.myCall(person, 2, 8)
console.log('返回值为:', res)
```

手写call、apply、bind:

需求:实现myApply方法,功能和调用形式和apply一致

定义myApply方法

设置this并调用原函数

接收参数并返回结果

```
const person = {
  name: 'itheima'
}

function func(numA, numB) {
  console.log(this)
  console.log(numA, numB)
  return numA + numB
}

const res = func.myApply(person, [2, 8])
console.log('返回值为:', res)
```

手写call、apply、bind:

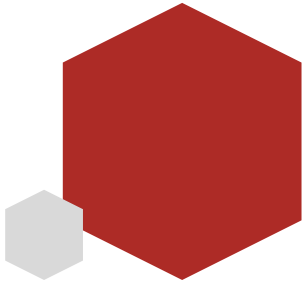
需求:实现myBind方法,功能和调用形式和bind一致

定义myBind方法

返回绑定this的新函数

合并绑定和新传入的参数

```
const person = {
  name: 'itheima'
}

function func(numA, numB, numC, numD) {
  console.log(this)
  console.log(numA, numB, numC, numD)
  return numA + numB + numC + numD
}

const bindFunc = func.myBind(person, 1, 2)

const res = bindFunc(3, 4)
console.log('返回值:', res)
```



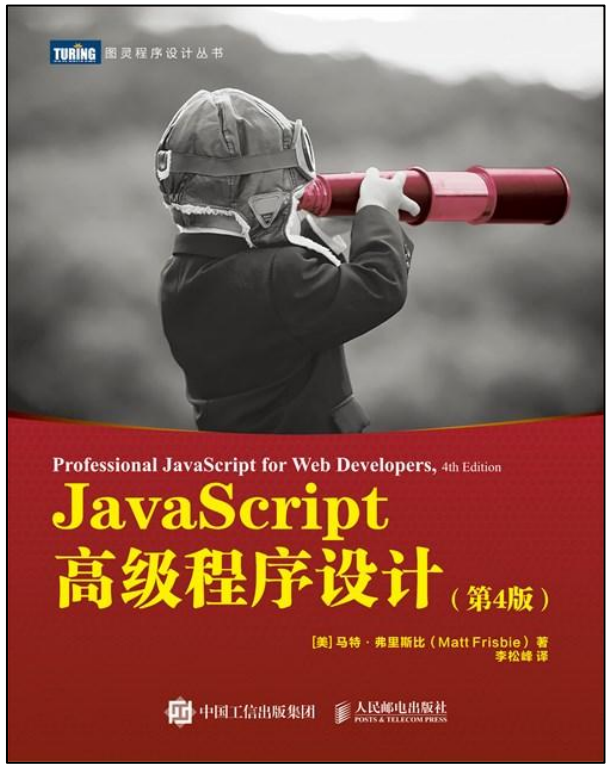
JavaScript继承

JavaScript继承

继承:继承可以使子类具有父类的各种属性和方法，而不需要再次编写相同的代码

1. ES6: 基于Class实现继承
    1. class核心语法
    2. class实现继承
    3. 静态属性和私有属性
2. ES5: 基于原型和构造函数实现继承
    1. 原型链继承
    2. 借用构造函数继承
    3. 组合继承
    4. 原型式继承
    5. 寄生式继承
    6. 寄生组合式继承

```
// 父类构造函数
function Person(name) {
  this.name = name
}
// 父类原型
Person.prototype.sayHi = function () {
  console.log(`你好, 我叫${this.name}`)
}
```



JavaScript继承 ES6-class核心语法

类是用于创建对象的模板，他们用代码封装数据以处理该数据， JS 中的类建立在原型上

```
class Person {
  // 1. 公有属性
  name
  // 2. 构造函数
  constructor(name) {
    this.name = name
  }
  // 3. 公有方法
  sayHi() {
    console.log(`你好,我叫:${this.name}`)
  }
}
```

```
const p = new Person('itheima')
```

JavaScript继承 ES6-class实现继承

extends:关键字用于类声明或者类表达式中，以创建一个类，该类是另一个类的子类。

super:关键字用于访问对象字面量或类的原型（[[Prototype]]）上的属性，或调用父类的构造函数。

```
class Person {
  name
  constructor(name) {
    this.name = name
  }
  sayHi() {
    console.log('父类的sayHi')
  }
}
```

```
class Student extends Person {
  age
  constructor(name, age) {
    super(name)
    this.age = age
  }
}
```

JavaScript继承 ES6-class静态属性和私有属性

静态:类（class）通过 static 关键字定义静态方法。不能在类的实例上调用静态方法，而应该通过类本身调用。

私有:类属性在默认情况下是公有的，但可以使用增加哈希前缀 # 的方法来定义私有类字段，声明和访问时也需要加上。

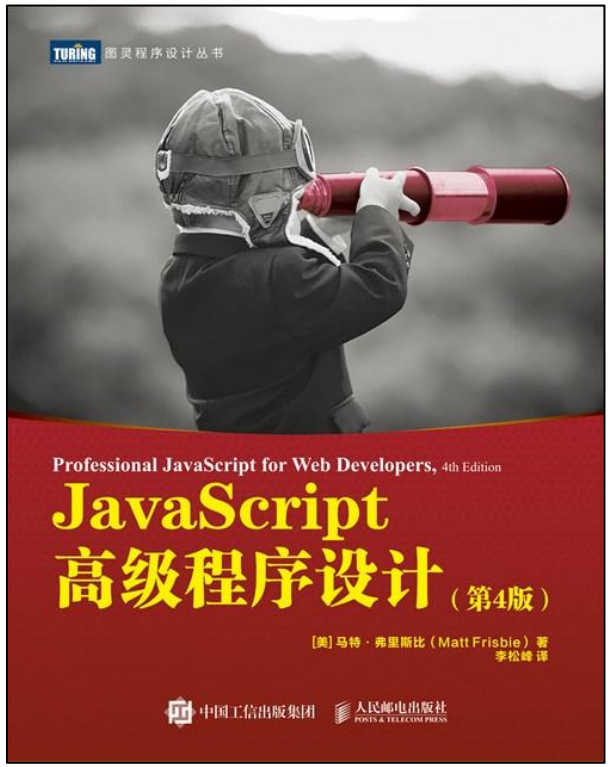
```
class Test {
  static 静态属性
  static 静态方法() { }

  #私有属性
  #私有方法() { }

  info() {
    this.#私有属性
    this.#私有方法()
  }
}

Test.静态属性
Test.静态方法()
```

JavaScript继承

继承:继承可以使子类具有父类的各种属性和方法，而不需要再次编写相同的代码

1. ES6: 基于Class实现继承
    1. class核心语法
    2. class实现继承
    3. 静态属性和私有属性
2. ES5: 基于原型和构造函数实现继承
    1. 原型链继承
    2. 借用构造函数继承
    3. 组合继承
    4. 原型式继承
    5. 寄生式继承
    6. 寄生组合式继承

```
// 父类构造函数
function Person(name) {
  this.name = name
}
// 父类原型
Person.prototype.sayHi = function () {
  console.log(`你好, 我叫${this.name}`)
}
```



JavaScript继承 ES5-寄生组合式继承

所谓寄生组合式继承，即通过借用构造函数来继承属性，通过原型链的混成形式来继承方法。

```
// 父类构造函数
function Person(name) {
  this.name = name
}
// 父类原型
Person.prototype.sayHi = function () {
  console.log(`你好，我叫${this.name}`)
}
```

```
// 子类构造函数
function Student(name) {
  Person.call(this, name)
}

// 基于父类的原型创建一个新的原型对象
const prototype = Object.create(Person.prototype,{
  constructor:{
    value:Student
  }
})
Student.prototype = prototype
```

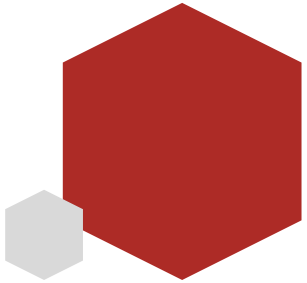
JavaScript继承

继承:继承可以使子类具有父类的各种属性和方法，而不需要再次编写相同的代码

1. ES6: 基于Class实现继承
    1. class核心语法
    2. class实现继承
    3. 静态属性和私有属性
2. ES5: 基于原型和构造函数实现继承
    1. 原型链继承
    2. 借用构造函数继承
    3. 组合继承
    4. 原型式继承
    5. 寄生式继承
    6. 寄生组合式继承

```
class Student extends Person {
  age
  #pInfo = '私有属性'
  static stInfo = '静态属性'
  constructor(name, age) {
    super(name)
    this.age = age
  }
  #pMethod() { }
  static stMethod() { }
}
```

```
function Student(name) {
  Person.call(this, name)
}

const prototype = Object.create(Person.prototype,{
  constructor:{
    value:Student
  }
})
Student.prototype = prototype
```



fetch

fetch

fetch: fetch是浏览器内置的api,用于发送网络请求.

AJAX vs axios vs fetch:

1. AJAX: 基于XMLHttpRequest收发请求,使用较为繁琐
2. axios: 基于Promise的请求客户端,在浏览器和node中均可使用,使用简便,功能强大
3. fetch: 内置api,基于Promise,用法和axios类似,功能更为简单

咱们会学习的有:

1. fetch核心语法
2. fetch提交FormData
3. fetch提交JSON

fetch-核心语法

核心语法:

1. 如何发请求?
2. 如何处理响应(JSON)?
3. 如何处理异常?

```
async function func() {
  const res = await fetch('请求地址')
  res.status
  const data = await res.json()
}
```

测试接口:

fetch-提交FormData

核心语法:

1. 如何设置请求方法?
2. 如何提交数据?

```
async function func() {
  const res = await fetch('请求地址', {
    method: '请求方法',
    body: '提交数据'
  })
}
```

测试接口:

fetch-提交JSON

核心语法:

1. 如何设置请求头?

```
async function func() {
  const headers = new Headers()
  headers.append('key','value')
  const res = await fetch('请求地址', {
    method: '请求方法',
    body: '提交数据',
    headers:headers
  })
}
```

测试接口:

fetch

fetch: fetch是浏览器内置的api,用于发送网络请求.

AJAX vs axios vs fetch:

1. AJAX: 基于XMLHttpRequest收发请求,使用较为繁琐
2. axios: 基于Promise的请求客户端,在浏览器和node中均可使用,使用简便,功能强大
3. fetch: 内置api,基于Promise,用法和axios类似,功能更为简单

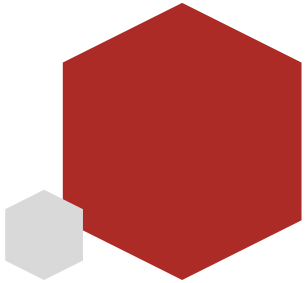
咱们会学习的有:

1. fetch核心语法
2. fetch提交FormData
3. fetch提交JSON

```
async function func() {
  const res = await fetch('请求地址', {
    method: '请求方法',
    body: '提交数据',
    headers: headers
    //...其他属性
  })
  res.status
  const data = await res.json()
}
```

兼容ie10+:

1. promise-polyfill
2. whatwg-fetch



Generator

Generator

Generator 函数是 ES6 提供的一种异步编程解决方案


```
axios('url1')
  .then(res1 => {
    return axios('url2')
  })
  .then(res2 => {
    return axios('url3')
  })
  .then(res3 => {
    // ....
  })
```

```
function* funcGenerator() {
  yield axios('url1')
  yield axios('url2')
  yield axios('url3')
}
```

```
const func = funcGenerator()
func.next().value
  .then(res => {
  })
```

```
async function func() {
  const res1 = await axios('url1')
  const res2 = await axios('url2')
  const res3 = await axios('url3')
  // ....
}
```

```
func()
```

咱们会学习的有:

1. Generator核心语法
2. Generator管理异步

Generator-核心语法

Generator 对象由生成器函数返回并且它符合可迭代协议和迭代器协议。

生成器函数在执行时能暂停，后面又能从暂停处继续执行。

```
function* func() {
  yield 'it'
  yield 'heima'
  yield '666'
}
```

```
const f = func()

f.next()

for (const iterator of f) {
  console.log(iterator)
}
```

Generator

Generator 函数是 ES6 提供的一种异步编程解决方案


```
axios('url1')
  .then(res1 => {
    return axios('url2')
  })
  .then(res2 => {
    return axios('url3')
  })
  .then(res3 => {
    // ....
  })
```

```
function* funcGenerator() {
  yield axios('url1')
  yield axios('url2')
  yield axios('url3')
}
```

```
const func = funcGenerator()
func.next().value
  .then(res => {
  })
```

```
async function func() {
  const res1 = await axios('url1')
  const res2 = await axios('url2')
  const res3 = await axios('url3')
  // ....
}
```

```
func()
```

咱们会学习的有:

1. Generator核心语法
2. Generator管理异步

Generator-管理异步

核心步骤: 异步操作之前加上yield

```
function* cityGenerator() {
  yield axios('http://hmajax.itheima.net/api/city?pname=北京')
  yield axios('http://hmajax.itheima.net/api/city?pname=广东省')
}
```

Generator

Generator 函数是 ES6 提供的一种异步编程解决方案


```
axios('url1')
  .then(res1 => {
    return axios('url2')
  })
  .then(res2 => {
    return axios('url3')
  })
  .then(res3 => {
    // ....
  })
```

```
function* funcGenerator() {
  yield axios('url1')
  yield axios('url2')
  yield axios('url3')
}
```

```
const func = funcGenerator()
func.next().value
  .then(res => {
  })
```

```
async function func() {
  const res1 = await axios('url1')
  const res2 = await axios('url2')
  const res3 = await axios('url3')
  // ....
}
```

```
func()
```

咱们会学习的有:

1. Generator核心语法
2. Generator管理异步

传智教育旗下高端IT教育品牌