# 函数中的this

this就是一个指针变量，**动态指向当前函数的运行环境**

## 全局环境下的this

全局作用域下，**this永远指向window**。

## 普通函数中的this

1. 谁调用this就指向谁，没有调用，this就指向window
2. 严格模式下，必须写上函数的调用者，不能省略或简写。全局中this的调用者是window。因此要写window.fn()。

## 对象中的this

对象内部方法的this指向调用这些方法的对象，也就是**谁调用就指向谁**

```
var obj = {
    name:"张三",
    age:18,
    sayName:function(){
        console.log("my name is "+this.name);
    },
    obj:{
        name:"李四",
        age:20,
        sayName:function(){
            console.log("my name is "+this.name);
        }
    }
}
obj.sayName();//张三。obj调用了sayName，所以此时this指向obj。
obj.obj.sayName();//李四。说明内部方法的this指向离被调用函数最近的对象。printChinese离grade对象近，所以this指向grade对象。
```

## 箭头函数中的this

箭头函数没有自己的this，它的this**在函数被定义时就已经被绑定**，指向函数所在作用域的外部作用域。

```
var obj = {

}
```

## 构造函数中的this

构造函数中的this永远指向被创建的实例对象。如果构造函数中**设置了具有对象的返回值**，那么该对象就是new表达式的结果；如果返回值不是对象，则new表达式的结果为构造函数的实例对象。

```
function Fun(name,age){
    this.name = name;
    this.age = age;
    this.sayName = function(){
        console.log(this)
    }
}
let fun = new Fun("张三",18);
fun.sayName();//fun
function Fun1(a){
    this.a = a;
    return {a:18,name:"李四"};
}
let fun1 = new Fun1(17);
console.log(fun1);//{a:18,name:"李四"},这里的fun1是构造函数返回的对象
console.log(fun1.a);//18
```

```
▶ Fun {name: '张三', age: 18, sayName: f}
▶ {a: 18, name: '李四'}
18
```

### 特殊

1. 定时器中的this

   定时器中的回调为普通函数时，其中的this永远指向window，即全局环境。

   若为箭头函数时，它的this指向上一层的对象。

```
var obj={
    name:"张三",
    fun3:function(){
        setTimeout(function(){
            console.log(this)
        },100)
    },
        fun4:function (){
        setTimeout(()⇒{
            console.log(this)
        },100)
    }
}

obj.fun3();//window
obj.fun4()//obj
```

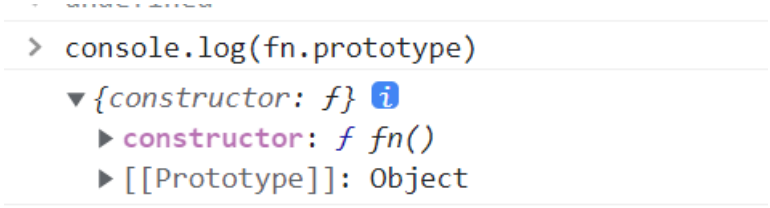
```
▶Window {window: Window, self: Window, document: document, name: '张三', location: Location, …}
▶{name: '张三', fun3: f, fun4: f}
```

# 原型和原型链

## 原型

当一个函数被创建时，JS会给他内部自动添加一个属性：prototype，prototype为一个对象，其中有一个constructor指向创建的函数。

```
function fn(){}
```




可以看到fn的prototype属性中有constructor属性，可以将其看作指针，它指向的是fn这个函数。

## 构造函数和实例对象

一般来说，函数名首字母大写的函数称为构造函数。使用**new关键字**可以通过构造函数定义一个它的实例对象。

```
function Fn(){
    this.name = "张三";
    this.age = 18;
    this.info = function(){
        console.log(this.name,this.age);
    }
}
let fn = new Fn()
console.log(fn.name);//张三
console.log(fn.age);//18
fn.info()//张三, 18
```

此时fn就是Fn的**实例对象**。

fn会继承在Fn的所有属性，在new的过程中，Fn中的this会自动指向fn，所以在new的过程中，JS做了以下事情。

```
fn.__proto__ = Fn.prototype;
Fn.call(fn);
```

## __proto__和prototype

在每个构造函数的实例对象中都有**__proto__**属性,此时也可以把它当作一个**指针**，它所指向的就是构造函数的prototype对象。

```
function Fn(){

}
let fn = new Fn()
console.log(fn.__proto__ === Fn.prototype)//结果为true
```

此时fn就可以通过__proto__获取Fn的prototype原型上的所有属性。

## 原型链

刚刚说到，通过__proto__获取Fn的prototype原型上的所有属性,prototype也是一个实例对象，它是Object的实例对象，因此在prototype中也有__proto__属性，它指向的是Object函数的原型对象，即Object的prototype原型对象。

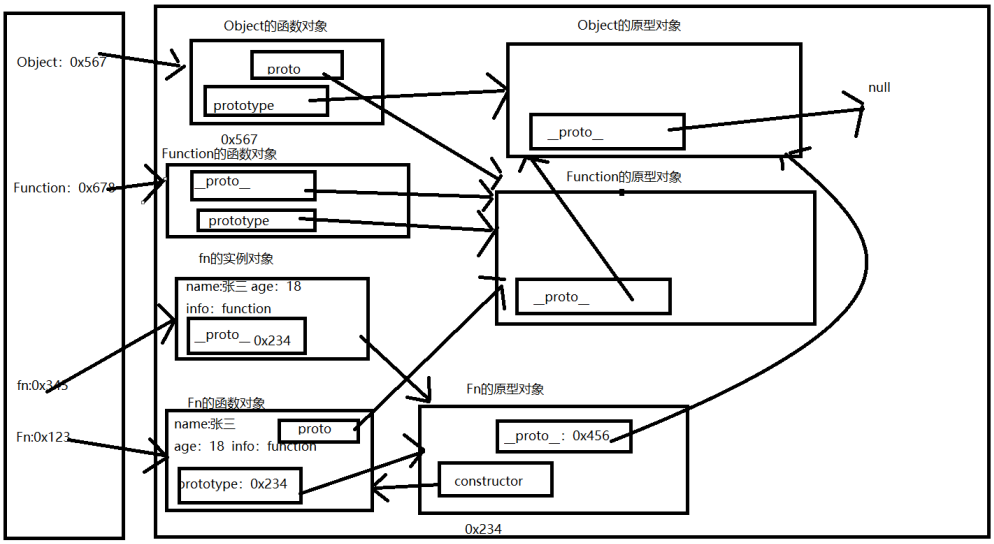
既然prototype是一个实例对象，那么只要是实例对象，都会有一个__proto__指向它的构造函数的原型对象。而Object的prototype的__proto__所指向的为null。根据定义，null没有原型，因此原型链的尽头为Object.prototype.__proto__。

```
function Fn(){
    this.name = "张三";
    this.age = 18;
    this.info = function(){
        console.log(this.name,this.age);
    }
}
let fn = new Fn();
console.log(fn.__proto__ === Fn.prototype)//true
console.log(Fn.prototype.__proto__ === Object.prototype)//true
console.log(Object.prototype.__proto__)//null
```

查找一个属性，实例对象会先在自身寻找，如果没有，会去它的__proto__也就是构造函数的prototype原型上找,如果也没有，则去构造函数的prototype的__proto__，也就是Object.prototype中去找，如果也没有，则返回undefined。

综上整个过程，即为原型链。

```
function Fn(){
    this.name = "张三";
    this.age = 18;
    this.info = function(){
        console.log(this.name,this.age);
    }
}
let fn = new Fn();
```



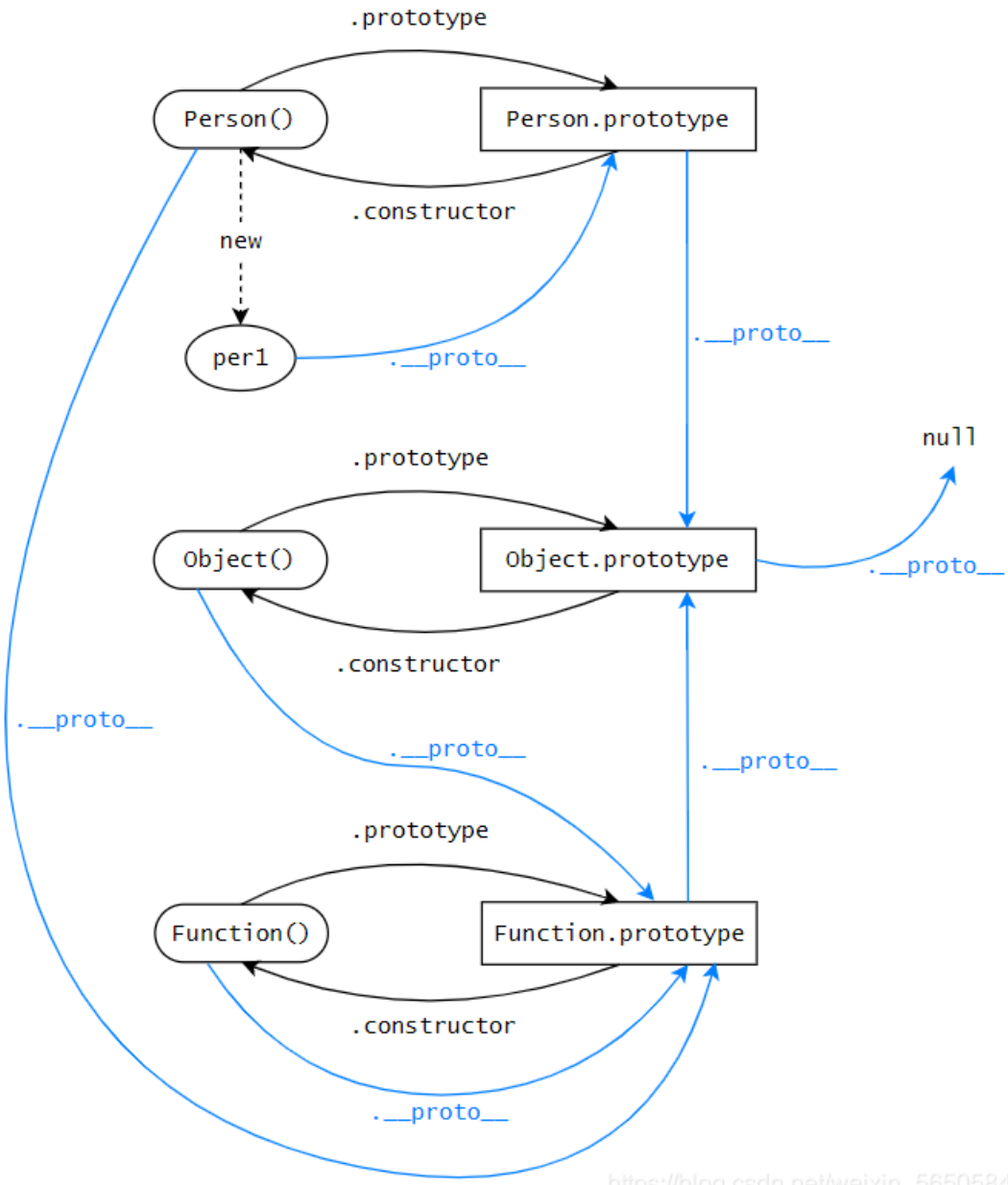

## Function、Object、function

其实在构造函数中，也有一个__proto__，这是为什么呢?

因为构造函数是函数，而所有函数都是Function的实例（包括Function本身），所以就会有一个很有意思的现象。Function的__proto__指向Function的prototype原型对象。

而Object也是一个构造函数，它的__proto__指向Function的prototype。

```
console.log(Object.__proto__ === Function.prototype)//true
console.log(Function.__proto__ === Function.prototype)//true
console.log(Function.prototype.__proto__ === Object.prototype)//true
```





## 继承

### 原型链继承

将**父类的实例**赋给**子类的原型**，这样子类就可以获得父类的属性和方法，以及原型上的属性和方法。

```
//组合继承
function Person(){
    this.name = 'person';
    this.friends = ['a','b','c'];
    this.get = ()=>{
        console.log("我是Person");
    }
```

```
}
Person.prototype.say = function(){
    console.log("我是Person的原型");
}
function Student(){}
Student.prototype = new Person();

let stu1 = new Student();
console.log(stu1.name);
stu1.get();
stu1.say();
console.log(stu1.friends);
console.log(stu1);

let stu2 = new Student();
stu2.friends.push("d");
console.log(stu2.friends);
```

优点：

1. 可以继承父类及其原型的属性和方法

缺点：

1. **所有子类共享**，一旦一个子类修改了父类的属性或方法，其他子类也会改变

## 构造函数继承

在子类中通过call或apply改变父类的this指向，调用父类。

```
//构造函数继承
function Person(){
    this.name = 'person';
    this.friends = ['a','b','c'];
    this.get = ()=>{
        console.log("我是Person");
    }
}
Person.prototype.say = function(){
    console.log("我是Person的原型");
}
function Student(){
    Person.call(this)
}
let stu = new Student()
```

```
console.log(stu.name);
console.log(stu.friends);
stu.get();
console.log(stu);
stu.say();
```


优点：

1. 解决了原型链继承的缺点，每个子类继承的都是独一无二的父类，互不影响

缺点：

1. **无法继承父类原型链上的属性和方法**，相当于在子类里**调用了父类函数**，给子类添加了父类的属性和方法
2. 无法复用

## 组合继承

使用原型链继承和构造函数继承

```
function Person(){
    this.name = 'person';
    this.friends = ['a','b','c'];
    this.get = ()⇒{
        console.log("我是Person");
    }
}
Person.prototype.say = function(){
    console.log("我是Person的原型");
}
function Student(){
```

```
    Person.call(this)
}
Student.prototype = new Person();
let stu = new Student();
console.log(stu.name);
console.log(stu.friends);
stu.get();
stu.say();
console.log(stu);
```


优点

1. 解决了构造函数继承无法继承父类原型的问题，且子类继承的属性都是独立的，互不影响

缺点

1. 会**执行两次父类的构造函数**，消耗较大内存

## 寄生组合式继承

利用一个新的函数Fn，将父类的实例赋给函数Fn的原型，然后子类中用call调用父类，并将Fn的实例赋给子类的原型

```
function Person(){
    this.name = 'person';
    this.friends = ['a','b','c'];
    this.get = ()⇒{
        console.log("我是Person");
    }
}
```

```
    Person.prototype.say = function(){
        console.log("我是Person的原型");
    }
    function Fn(){};
    Fn.prototype = new Person();
    function Student(){
        Person.call(this);
    }
    Student.prototype = new Fn();
    let stu = new Student();
    console.log(stu.name);
    console.log(stu.friends);
    stu.get();
    stu.say();
    console.log(stu);
```


# 闭包

本质就是内部函数引用的外部函数的变量或方法。

优点

1. 可以实现变量私有化，可以让变量不受污染

缺点

1. 可能**造成内存泄漏**

## 柯里化函数

# Promise

## then

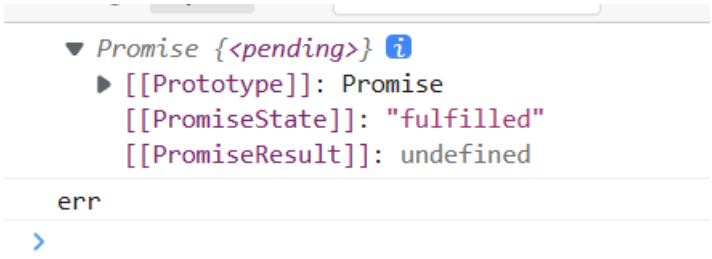
then函数是由返回值的，返回的是一个**新的Promise对象实例化对象**。

如果p1实例化对象是**成功的状态**，则**执行then的成功回调函数**；如果是**失败**的状态，则**执行then的失败的回调函数**。

**如果then中没有返回值**

- then函数返回的Promise是**成功**的状态，并且**结果值是undefined**。

```
let p1 = new Promise((resolve,reject)=>{
    // resolve(2)
    // reject(1)
    throw "err"
});
const p2 = p1.then( value => {
    console.log(value);
},reason => {
    console.log(reason);
})
console.log(p2);
```




**如果then中有返回值**

1. 返回的为**非Promise实例化对象**，则**状态仍是成功的Promise实例化对象，返回值就是结果值**
2. 返回的为**Promise实例化对象**，则**p2的状态取决于Promise实例化对象的状态**
3. 如果**直接抛出异常**，则返回失败的Promise

```
let p1 = new Promise((resolve,reject)=>{
```

```
    // resolve(2)
    // reject(1)
    // throw "err"
    resolve(1)
});
const p2 = p1.then( value => {
    console.log(value);

    return "okok";//p2的状态 [[PromiseState]]: "fulfilled"  [[PromiseResult]]:
okok

    return new Promise((_,reject)=>{ //p2的状态 [[PromiseState]]: "rejected"
[[PromiseResult]]: 1
        reject(1)
    })
    throw "err"  //p2的状态 [[PromiseState]]: "rejected"  [[PromiseResult]]:
err
})
console.log(p2);//如果p2没有返回值，则p2的状态 [[PromiseState]]: "fulfilled"
[[PromiseResult]]: undefined

console.log(p1 === p2)// false
```

**如果then中省略或者不是一个回调函数**

- 默认成功的回调函数为 `value => value` ,默认失败的回调函数为 `reason => {throw reason}`

## catch

catch用于捕捉Promise里reject后的值，和then一样也返回一个新的Promise。**只有一个回调函数，表示失败的回调**。

**如果catch中没有返回值**

- catch函数返回的Promise是**成功**的状态，并且**结果值是undefined**。

**如果catch中有返回值**

1. 返回的为**非Promise实例化对象**，则**状态仍是成功的Promise实例化对象，返回值就是结果值**
2. 返回的为**Promise实例化对象**，则**p2的状态取决于Promise实例化对象的状态**
3. 如果**直接抛出异常**，则返回失败的Promise

## Promise.resolve()

`Promise.resolve()` 返回一个Promise。可以进行传参。

1. 参数为**非Promise或为空**： Promise状态为fulfilled，结果为传进的参数或undefined
2. 参数为**Promise的实例对象**，则返回的Promise的状态取决于Promise实力对象的状态。

```
const p = Promise.resolve();
const p1 = Promise.resolve(1)
const p2 = Promise.resolve(new Promise((resolve,reject)⇒{
    resolve(2)
}))
const p3 = Promise.resolve(new Promise((resolve,reject)⇒{
    reject("error")
}))
const p4 = Promise.resolve(new Promise((resolve,reject)⇒{
    throw "异常"
}))
console.log(p);//PromiseState fulfilled PromiseResult undefined
console.log(p1);//PromiseState fulfilled PromiseResult 1
console.log(p2);//PromiseState fulfilled PromiseResult 2
console.log(p3);//PromiseState rejected PromiseResult error
console.log(p4);//PromiseState rejected PromiseResult '异常'
```

## Promise.reject()

不管参数如何，都返回一个失败的Promise。

```
const p = Promise.reject();
const p1 = Promise.reject(1)
const p2 = Promise.reject(new Promise((resolve,reject)⇒{
    resolve(2)
}))
const p3 = Promise.reject(new Promise((resolve,reject)⇒{
    reject("error")
}))
const p4 = Promise.reject(new Promise((resolve,reject)⇒{
    throw "异常"
}))
console.log(p);//PromiseState rejected PromiseResult undefined
console.log(p1);//PromiseState rejected PromiseResult 1
console.log(p2);//PromiseState rejected PromiseResult Promise
```

```
console.log(p3);//PromiseState rejected PromiseResult Promise
console.log(p4);//PromiseState rejected PromiseResult Promise
```

# Promise.all()

`Promise.all()` 需要传递一个**数组**参数，数组里存的是**Promise的实例化对象**，**返回值**也是**Promise的实例化对象**。

## 返回值的Promise状态

1. 当数组中所有的Promise实例化对象的状态**都是成功**的话，则返回的Promise的状态也是**成功**的，结果值为**所有成功的Promise实例化对象的结果组成的数组**
2. 当数组中所有的Promise实例化对象的状态**有一个失败**的话，**直接返回失败的Promise实例化对象**，**结果值**就是**第一个失败Promise实例化对象的结果值**。

```
const p1 = new MyPromise((resolve,reject)=>{
    resolve(1)
})
const p2 = new MyPromise((resolve,reject)=>{
    // resolve(2)
    reject(1)
})
const p3 = new MyPromise((resolve,reject)=>{
    resolve(3)
})
const p4 = new MyPromise((resolve,reject)=>{
    resolve(4)
})
const p = MyPromise.all([p1,p2,p3]);//PromiseState rejected PromiseResult 1
const p = MyPromise.all([p1,p2,p4]);//PromiseState fulfilled PromiseResult
[1,3,4]
console.log(p);
```

# Promise.race()

参数为一个**由多个Promise实例对象组成的数组**。

返回一个新的Promise实例对象，状态取决于数组中**最先改变状态**的Promise实例对象的**状态为准**。

## 链式调用

当`new Promise`链式调用时，每调用一个then都会产生一个新的Promise，它的状态只要不是抛出异常`throw error`以及在then的回调里返回一个失败的Promise，它都会是一个成功的Promise。

```
let p1 = new Promise((resolve, reject) => {
    resolve(1);
}).then(value=>{
    console.log(value); //1
}).then(value => {
    console.log(value); //undefined
    return 2;
}).then(value => {
    console.log(value); //2
    throw "error"
}).then(null,reason => {
    console.log(reason); //error
}).catch(reason => {
    console.log(reason); // 不输出
})
```

## 异常穿透

当Promise的链式调用时出现异常，那么它会一直往下，直到遇到能处理异常的回调(**then中的onrejected回调或catch**),在此期间，then返回的Promise状态都是rejected，直到异常处理后才变成fulfilled。

```
let p1 = new Promise((resolve,reject) => {
    // reject(1)
    throw '异常'
})
console.log(p1);//PromiseState rejected PromiseResult '异常'
const p2 = p1.then(value=>{
    console.log(value);//不输出
})
console.log(p2);//PromiseState rejected PromiseResult '异常'
const p3 = p2.then(value => {
    console.log(value);//不输出
},reason => {
    console.log(reason);//异常
    return 1;
```

```
})
console.log(p3);//PromiseState fulfilled PromiseResult 1
const p4 = p3.then(value => {
    console.log(value);//1
})
console.log(p4);//PromiseState fulfilled PromiseResult undefined
```

## 中断Promise链

只要在链式调用的过程中**返回一个pending状态的Promise**，Promise链就会中断。因为pending状态的Promise并没有处理，then处理成功或者失败的Promise，catch处理失败的回调函数。

```
let p1 = new Promise((resolve,reject) => {
    resolve(1)
})
const p2 = p1.then(value=>{
    console.log(value);//1
    return 2;
})
const p3 = p2.then(value => {
    console.log(value);//2
    return new Promise(()=>{})
},reason => {
    console.log(reason);
})
const p4 = p3.then(value => {
    console.log(value);//不输出
})
```

# async函数和await表达式

注意点：

1. async函数中不一定要有await表达式，但有await的函数一定要有async
2. await相当于then，获取Promise实例对象成功的结果

## async

在函数前面加一个async就可以把该函数变为异步函数。

在函数内部正常书写，*async*函数的返回值为一个*Promise*,状态根据返回的数据决定。

1. 返回的是一个**非Promise类型**，那么状态为**成功**的Promise，结果为**返回的具体值**。
2. 返回的是一个**Promise实例对象**，返回的Promise即为async函数返回的Promise。
3. **抛出异常**，那么async也会返回一个**失败的Promise**，结果值为异常的值。

```
async function fn1(){
    return 0;
}
let f1 = fn1();
async function fn2(){
    return new Promise((resolve, reject) => {
        resolve(1)
    })
}
let f2 = fn2();
async function fn3(){
    return new Promise((resolve, reject) => {
        reject(2)
    })
}
let f3 = fn3();
async function fn4(){
    throw "异常"
}
let f4 = fn4();
console.log(f1);//PromiseState fulfilled PromiseResult 0
console.log(f2);//PromiseState fulfilled PromiseResult 1
console.log(f3);//PromiseState rejected PromiseResult 3
console.log(f4);//PromiseState rejected PromiseResult "异常"
```

## await（异步）

1. 如果await右侧为**非Promise类型的数据**，awiat后面**是什么，得到的结果就是什么**
2. 如果await右侧为**成功的Promise**，则得到的就是**成功的结果**
3. 如果await右侧为**失败的Promise**，需要用**try...catch**去捕获失败的结果

```
async function fn5(){
    let re = await null;
    console.log(re);
}
fn5();//null
async function fn6(){
    let re = await new Promise((resolve, reject) => {
        resolve(1)
    });
    console.log(re);
}
fn6();//1
async function fn7(){
    try{
        let re = await new Promise((resolve, reject) => {
            reject(2)//失败的Promise需要用try...catch去捕获
        });
        console.log(re);
    }catch(err){
        console.log(err);
    }
}
fn7();//err
```

## await执行顺序

同步和异步同时存在时，会优先执行同步代码，随后再执行异步。**await是异步的，它会等待后面的结果，哪怕是0秒也会等待。**

```
async function main(){
    console.log(1);
    let re1 = await new Promise((resolve, reject) => {
        console.log(2);
        resolve('ok');
    })
    console.log(3);
    let re = await setTimeout(()=>{
        console.log(1);//await返回的是定时器的唯一标识符，timerId
    });
    console.log("re" + re);
    console.log(5);
}
main();
```

```
console.log(6);
//1 2 6 3 re4 5 1
```