bind и **call/apply** : модели передачи функциям данных

```
 $ node
> let h = x=>x*x;

> h(5)
25
> h.bind(null, 5)
[Function: bound h]
> h.bind(null, 5)();
25
> h.call(null,5);
25
> let mixin = function(){return this.x*this.x};

> mixin.call({x:5});
25
> let not_workin = ()=>this.x*this.x;

> not_workin.call({x:5});
NaN
```

call = bind + ()

apply отличается тем
что раскладывает массив по гнёздам аргументов

h.apply(null,[5]) ~~**h.apply(null,5)**~~

mixin.apply({x:5})
mixin.bind({x:5})()

с точки зрения **this**
apply=call=bind+()

у стрелок нет своего **this**

Это лог демонстрации 10.05.2016 по вопросу о контексте вызова функции, роли this и различиях между call, apply и bind

Кратко говоря,
func.apply(context) ⇆ func.bind(context)()

Также здесь ALO (Array-Like Objects)

```
> var stud = {firstname: 'Ivan', lastname: 'Petrov'}
undefined
> var getFullName = function(){return this.firstname + ' ' + this.lastname}
undefined
> getFullName()
'undefined undefined'
> var firstname = 'myName', lastname = 'mySurname';
undefined
> getFullName()
'myName mySurname'
> getFullName.apply(stud)
'Ivan Petrov'
> getFullName.bind(stud)
[Function: bound ]
```

= Function.prototype.apply.call(getFullName, stud)
= getFullName.call(stud) т.к. у getFullName нет аргументов
= getFullName.bind(stud)()

```
> getFullName.bind(stud)()
'Ivan Petrov'
> Math.max(10,40,30)
40
> Math.max.apply(null, [10,40,30])
40
> Array.prototype.max = function() {return Math.max.apply(null, this);};
[Function]
> [10,40,30].max()
40
> var alo = {'0':10, '1':40, '2':30, 'length':3}
undefined
> alo[0]
10
> alo.max()
TypeError: alo.max is not a function
    at repl:1:5
    at REPLServer.defaultEval (repl.js:272:27)
    at bound (domain.js:280:14)
    at REPLServer.runBound [as eval] (domain.js:293:12)
    at REPLServer.<anonymous> (repl.js:441:10)
    at emitOne (events.js:101:20)
    at REPLServer.emit (events.js:188:7)
    at REPLServer.Interface._onLine (readline.js:219:10)
    at REPLServer.Interface._line (readline.js:561:8)
    at REPLServer.Interface._ttyWrite (readline.js:838:14)
> var alo2arr = Array.prototype.slice.call(alo)
undefined
> alo2arr.max()
40
> [].slice.call(alo).max()
40
>
```

arguments[0], arguments[1]…
это псевдонимы для безымянных гнёзд
т.е. как бы у функции **2+** аргументов
вызов **f.apply(null,** массив**)**
распределяет элементы массива по
этим гнёздам

а вызов **f.call(null,** массив**)**
привёл бы к тому
что массив стал бы виден
в функции как **arguments[0]**

потому что **call**
передаёт аргументы-через-запятую как-есть

```
var alo = {'0':30, '1':10, '2':20, 'length':3};
> alo.sort()
TypeError: alo.sort is not a function
> Array.prototype.sort.call(alo)
{ '0': 10, '1': 20, '2': 30, length: 3 }
//это изменило объект alo, т.к. он мутабелен

//а вот вариант без изменения исходного объекта:
> Array.prototype.slice.call(alo).sort();
[ 10, 20, 30 ]
```

вместо `Array.prototype` можно писать просто `[]`

выражение `Array.prototype.slice.call(alo)` содержит "временный" массив, который при желании можно сохранить в отдельную переменную.

BIND работает как метод данной функции
 создаёт по ней новую функцию с привязанным аргументов
   и вызывает её как метод чего-то (здесь - не как чего-то
         а просто функцию-сироту - т.е. null)

```
> (function(x){console.log(x)}).bind(null,55)()
55
```

CALL пропускает этам создания новой функции и просто вызывает её

```
> (function(x){console.log(x)}).call(null,55)
55
```

APPLY ещё и передаёт список аргументов массивом

```
> (function(x){console.log(x)}).apply(null,[55])
55
```

а вот теперь сделаем финт и вызовем BIND
       обобщённым вызовом из всего запаса методов объекта Function

```
> Function.prototype.bind.call( function(x){console.log(x)},
null, 55 )()
55
```

что происходит?
       наша функция становится объектом, методом которого вызывается bind
       т.е. получается (function()....).bind
       и ему как методу функции передается всё то, что в самом верху
        т.е. null который обозначает контекст вызова уже нашей функции
          и аргумент для связывания с x
           и дальше это получится новая функция, которую ещё надо вызвать

генерация функций по их телам

```
$ node
> let funcs = ['for (i=0, s=1; i<arguments.length; s*=arguments[i++]); return s'];

> let F0 = new Function(funcs[0]);

> F0(2,3,4)
24
> F0([2,3,4])
NaN
> F0.bind(null,2,3,4)()
24
> F0.call(null,2,3,4)
24
> //now passing arrays

> F0.bind(null,[2,3,4])()
NaN
> F0.call(null,[2,3,4])
NaN
> F0.apply(null,[2,3,4])
24
> F0(...[2,3,4])
24
```

массив попадает в **arguments[0]**
и никакого перебора массива не получается

и **call** и **bind** передают параметры как есть
через запятую, разница только **call = bind + ()**

то же что

spread = apply

см.плакат
kodaktor1.pdf