函数柯里化 Curry Function

- 函数式编程的思想

概念

在计算机科学中，柯里化把接受多个参数的函数变成接受一个单的参数(最初函数的第一个参数)，并且返回接收余下的参数且返回结果的新函数的技术。

```
function test(a, b, c) {
  return a + b + c
}

test(1, 2, 3);
test(1)(2)(3);
test(1,2)(3);
test(1)(2,3);
```

通过柯里化思想的函数，可以让上述代码调用返回相同的结果。

用途

- 简化代码，经过柯里化函数的包装，把函数封装起来，调用者负责传参就好了。
- 提高维护性 ，因为柯里化总是在返回新的函数，这个新函数实在一个函数体里产生的，提高维护性，逻辑性也更强。
- 功能单一化，函数的延迟执行，函数完成任务分段执行，延迟执行。

优点

- 功能内聚，所有实现的功能都在函数内部完成或者是关键性的固定性的东西都在前面完成，最后完成整个功能体的程序执行。
- 降低耦合
- 降低代码重复性
- 提高代码适应性，根据函数的延迟执行的特性，在最后根据不同程序定制化。

实现

- 使用数组的 slice 方法来实现柯里化

```
function curry(fn) {
  var _args = [].slice.call(arguments, 1);
  return function() {
    var newArgs = _args.concat([].slice.call(arguments))
    return fn.apply(this, newArgs)
  }
```

```
}
```

使用

```
    var add2 = curry(add, 1,2)
    var res = add2(3,4)
    console.log(res) // 10
```

但是这并不完全，因为只返回了一次，如果第二次调用没有传够，第四个参数 undefined， 结果返回 NaN;

那如果想随意传参任意个，怎么让 curry 返回的函数无限次 return 呢？

- 利用递归

最终实习

```
// 真正的柯里化
function curry (fn, len) {
  var len = len || fn.length;

  var func = function (fn) {
    var _args = [].slice.call(arguments, 1);
    return function () {
      var newArgs = _args.concat([].slice.call(arguments));
      return fn.apply(this, newArgs);
    }
  }

  return function () {
    var argLen = arguments.length;
    if(argLen < len) {
      //[fn]
      var formatedArr = [fn].concat([].slice.call(arguments));
      return curry(func.apply(this, formatedArr), len - argLen);
    } else {
      return fn.apply(this, arguments);
    }
  }
}
```

使用

```
var add2 = curry(add)
var res = add2(1)(2,3,4)
console.log(res)
```

柯里化的应用

封装ajax

- 相当于一种缓存机制，把 ajax 数据的都缓存下来，等后续需要再传完所有参数，执行程序。

```
function ajaxRequest(opt, data, sCB, eCB) {
  $.ajax({
    url: opt.url,
    type: opt.type,
    dataType: opt.dataType,
    data:data,
    success: sCB,
    eror: eCB
  })
}

var $ajax = curry(ajaxRequest);
var ajaxApi = {
  getCourse: $ajax({
    url: 'http://study.jsplus.com/Index/getCourses/api',
    type: 'POST',
    dataType: 'JSON'
  })
}
ajaxApi.getCourse({
  page:1
})
(function(data) {
  console.log(data)
})
(function(){
  console.log('Error')
})
```

封装点击事件

临时封装一个点击事件

```
function addEvent(el, type, fn, capture) {
  if(el.addEventListener) {
    el.addEventListener(type, fn, capture);
  } else if(el.attachEvent) {
    el.attachEvent('on' + type, fn, function() {
      fn.call(el);
    })
  } else {
    el['on' + type] = fn;
  }
}
```

但是这样做有个去缺点，每次点击都会进入 某个条件 ， 即便之前进入过，还是会重新进入 可以根据这个事件来个优化

- 闭包再做缓存池

```
var addEvent = (function (el, type, fn, capture) {
  if(window.addEventListener) {
    console.log('init')
    return function(el,type, fn, capture) {
      el.addEventListener(type, fn, capture);
    }
  } else if(window.attachEvent) {
    return function(el,type, fn) {
      el.attachEvent('on' + type, fn, function() {
      fn.call(el);
    })
    }
  } else {
    return function(el,type, fn) {
      el['on' + type] = fn;
    }
  }
})();
```

使用

```
var oBtn = document.getElementsByTagName('button')[0];
addEvent(oBtn, 'click', btnClick, false);
function btnClick() {
  console.log(1)
}
```

重复点击，重复输出 1，但是 init 只会在第一次输出，这是一次缓存池的应用。