高阶函数

- 可以把函数作为参数传递给另一个函数
- 可以把函数作为另一个函数的返回结果

- 意义
  - 抽象可以帮我们屏蔽细节，只需要关注我们的目标
  - 高阶函数是用来抽象通用的问题
- 好处
  - 可缓存
    - 欣慰纯函数对相同的输入始终有相同的结果，所以可以把纯函数的结果缓存起来
  - 可测试
    - 纯函数让测试更方便
  - 并行处理
    - 在多线程环境下并行操作共享内存数据可能出现意外情况
    - 纯函数不需要访问共享的内存数据，所以在并行环境下可以任意运行纯函数（web worker）ES6之后是可以开启多线程的

```
//高阶函数,将函数当变量传入
function forEeach(arr, fn) {
  for (let item of arr) {
    fn(item);
  }
}
let mylist = [1, 2, 3, 4];

forEeach(mylist,function(item){
   console.log(item)
 })

```

```
function forfilter(arr, fn) {
  let result = [];
  for (let item of arr) {
    if (fn(item)) {
      result.push(item);
    }
  }
  return result;
}

const L = forfilter(mylist, function (item) {
  return item % 2 == 0;
});

console.log(L);


// 把函数作为返回值
// 仅仅只执行一次的函数
function once(fn) {
  let done = false;
  return function () {
    if (!done) {
      done = true;
      return fn.apply(this, arguments);
    }
  };
}

let pay = once(function (money) {
  console.log(`支付: ${money} RMB`);
});

pay(5);
pay(5);
pay(5);
```

闭包

- 闭包(Closure): 函数和其周围的状态(词法环境)的引用捆绑在一起形成闭包

- 可以在另一个作用域中调用一个函数的内部函数并访问到函数的作用域中的成员

理解：

原本函数执行完里面的属性就会被释放掉，如果内部函数在引用就会延迟释放

本质：

函数在执行的时候会放到一个执行栈上当函数执行完毕之后会从执行栈上移除，但是堆上的作用域成员因为被外部引用不能释放，因此内部函数依然可以访问外部函数的成员

lodash

```
npm install lodash

const _ = require('lodash')

const array = ['jack', 'tom']
console.log(_.first(array))
```

柯里化

lodash 中的柯里化函数

_.curry(func)

功能: 创建一个函数，该函数接受一个或多个func的参数，如果func所需要的参数都被提供则执行func并返回执行结果，否则继续返回该函数并等待接收剩余的参数

```
lodash 柯里化的原理
function curry(func) {
  return function curriedFn(...args) {
    if (func.length > args.length) {
      return function () {
        // 将上一次函数调用的参数，加到输入参数列表中，下次输入的值就会紧随其后
        return curriedFn(...args.concat(Array.from(arguments)));
      };
    }
     // 函数的形参和传的变量值相同时，就直接执行函数
     return func(...args);
   };
}
```

总结

- 柯里化可以让我们给一个函数传递较少的参数得到一个已经记住了某些固定参数的新函数
- 这是一种对函数参数的缓存
- 让函数变得更加灵活，让函数的粒度更小
- 可以把多元函数转换成一元函数，可以组合使用函数产生强大的功能

组合函数原理模拟

```
// 函数组合    原理
const compose = (...args) => value => args.reverse().reduce((acc, fn) =>fn(acc), value)
```

函数组合要满足结合律

先结合前两个数，和先结合后两个数，结果要一样

lodash-fp模块

里面又lodash原本的所有方法，但是fp里面的方法是经过柯里化的，都是函数在前，传值在后的格式

```
let _underscore = fp.replace(/\W+/g, "_");
function sanitizeNames(array) {
  return fp.flowRight(
    fp.castArray,
    _underscore,
    fp.lowerCase,
    fp.split(" ")
  )(array);
}

console.log(sanitizeNames(["Hello World"]));
```

Point Free

- 我们可以把数据处理的过程定义成与数据无关的合成运算，不需要用到代表数据的那个参数，只要简单的运算步骤合成到一起，在使用这种模式之前，我们需要定义一些辅助的基本运算函数
  - 不需要指明处理数据
  - 只需要合成运算过程
  - 需要定义一些辅助的基本运算函数

functor 函子

函子针对于不同的情况，写的不同的方法，也叫做不同名称的函子

- 为什么要学函子
  - 到目前位置已经学习了函数式编程的一些基础，但是我们还没有演示在函数式编程中如何把副作用控制在可控的范围内，异常处理，异步操作等。
- 什么是functor
  - 容器: 包含值和值的变形关系（这个变形关系就是函数）
  - 函子:是一个特殊的容器，通过一个普通的对象来实现，该对象具有map方法，map方法可以运行一个函数对值进行处理（变形关系）
- 总结
  - 函数式编程的运算不直接操作值，而是由函子完成
  - 函子就是一个实现了map契约的对象
  - 我们可以把函子想象成一个盒子，这个盒子里封装了一个值
  - 想要处理盒子中的值，我们需要给盒子的map方法传递一个处理值的函数（纯函数），由这个函数来对值进行处理
  - 最终map方法返回一个包含新值的盒子（函子）

```
// functor  定义map方法，map方法用于传入函数操作，函子中的值
class Container {
  constructor(value) {
    this._value = value;
  }
```

```
  //
  map(fn) {
    return new Container(fn(this.myvalue));
  }
}

// const L = new Container(5).map((x) => x * x);
// console.log(L._value);

//默认调用 class Container 就会返回名称为Container的一个对象
// 对象中会包含 Container 中的所有的值
const L = new Container(5);
console.log(L);
```

MayBe函子

为解决函子中，传入的值为null或undefined的情况

- 我们子啊编程的过程中可能会遇到很多问题，需要对这些错误做相对于的处理
- MayBe函子的作用就是可以对外部的空值情况做处理（控制副作用在运行范围）

```
class MayBe {
  static of(value) {
    return new MayBe(value);
  }
  constructor(value) {
    this._value = value;
  }
  map(fn) {
    return this.isNothing() ? MayBe.of(null) : MayBe.of(fn(this._value));
   }
  // 这一步isNothing方法，解决了值为null或undefinde的情况，所以叫做MayBe函子
   isNothing() {
     return this._value === null || this._value === undefined;
   }
}

const L = MayBe.of(null);
// 多次调用map哪一次出现null是不能明确的
console.log(L.map((x) => x + 1));
```

Either函子

```
class left {
  static of(value) {
    return new left(value);
  }
  constructor(value) {
    this._value = value;
  }
  map(fn) {
    return this;
   }
}

class right {
  static of(value) {
    return new right(value);
  }
  constructor(value) {
    this._value = value;
  }
  map(fn) {
    return right.of(fn(this._value));
  }
}

const R = right.of(5).map((x) => x + 2);
const L = left.of(5).map((x) => x + 2);
console.log(R);
console.log(L);

function parseJSON(str) {
  try {
    return right.of(JSON.parse(str));
  } catch (e) {
      // 可以记录下出错的信息
    return left.of({ error: e.message });
  }
}

const P = parseJSON({ name: "zhangsan" });
```

```
40 console.log(P);
```

IO函子

- IO函子中的_value是一个函数，这里是把函数作为值来处理的
- IO函子可以把不纯的动作储存到_value中，延迟执行这个不纯的操作(惰性执行)，包装当前操作纯，
- 把不存的操作交给调用者来处理

folktale 跟lodash类似的插件

```
1 npm install folktale
2
3 const { compose, curry } = require("folktale/core/lambda");
4 const { toUpper, first } = require("lodash/fp");
5
6 let f = compose(toUpper, first);
7 console.log(f(["node", "two"]));
```

- 常见的GC算法
  - GC是指垃圾回收机制
  - GC算法，可以理解为垃圾回收机制的几种算法
- 引用计数算法
  - 通过对象的引用次数是否为零，来判断是否回收
  - 优点
    - 可以即时回收垃圾对象
    - 减少程序卡顿时间
  - 缺点

        - 无法回收循环引用的对象
        - 资源消耗大，因为要频繁的改变计数的值
- 标记计数算法
    - 主要分为两步
        - 第一步循环遍历对象，找到活动对象(可达对象)，做上对应的标记
        - 第二部循环遍历对象，清除没有标记的对象，再清除完没有标记的对象后，会把第一阶段做的标记抹掉，便于下次重新标记

把回收的空间，放在一个叫空闲列表中，以便于下次直接在空闲列表中，申请空间使用

- 优点
    - 对于引用计数算法来说，在不是可达对象，但是局部互相调用，是不满足计数算法的回收条件的，所以无法回收，
    - 在标记清除算法这里，在不能通过全局访问的变量，无法通过全局的循环查找到的变量都会被回收，不论他是不是互相调用
- 缺点
    - 因为标记清除算法是回收原本申请的存储空间，因为回收的存储空间来子多个指针，而且大小各不同，并不能将空间进行统一管理，造成空间碎片化的问题，只有存储大小完全匹配才能直接使用
    - 不会立即回收垃圾对象，一般都留在最后清除，清除时程序是停止工作的

- 标记整理算法 (增强版)
    - 标记整理算法，就是在标记算法的基础上，在空间回收之前，对活动空间和非活动空间进行位置的整理，让他们形成一

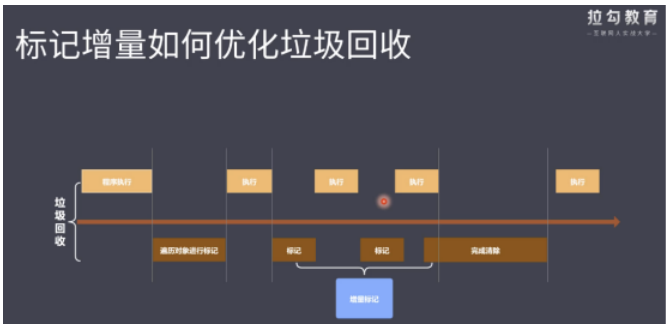
个连续的空间，这样回收的空间也是连续的，就不会出现空间碎片化的问题

- V8
  - V8是一款主流的javascript执行引擎
  - V8采用即时编译
  - V8内存设限(X64 不超过1.5G,X32 不超过800M)
  - 因为内存回收机制的缘故，内存容量过大，会导致回收时间的过长，所以设限内存大小

- V8中常用的GC算法
  - 分代回收
  - 空间复制
  - 标记清除
  - 标记整理
  - 标记增量
- V8内存分配
  - v8内存空间分为两部分
  - 新生代
  - 小空间用于存储新生代对象(32M|16M)
  - 新生代对象指的是存活时间较短的对象(一般是指函数内的对象，执行完就回收)
  - 新生代对象回收实现
    - 回收过程采用复制算法+标记整理算法
    - 新生代内存区分为两个等大的空间
    - 使用空间为From，空闲空间为To
    - 活动对象存储于From空间
    - 标记整理后将活动对象拷贝至To
    - From(释放后)与To交换空间
    - 回收的细节

- 拷贝过程中可能出现晋升
- 晋升就是将新生代对象移动至老年代
- 一轮GC还存活的新生代需要晋升
- To空间使用率超过百分之25%(因为在复制整理算法下To空间内都是一轮垃圾回收后存活的，需要移到老生代才能为新生代腾出位置)

- 老生代
  - 老生代对象存放在右侧老生代区域
  - 64位操作系统1.4G，32位操作系统700M
  - 老生代对象就是指存活时间较长的对象
  - 老年代对象回收实现
  - 主要采用标记清除，标记整理，增量标记算法
  - 首先使用标记清除完成垃圾空间的回收
  - 采用标记整理进行空间优化(触发条件： 老生代中的空间不足以存储新生代晋升后时。会触发标记整理)
  - 采用增量标记进行效率优化(优化标记的效率)
- 细节对比
- 新生代区域垃圾回收使用空间换时间(因为空间本身不大，造成空闲空间也是微不足道)
- 老生代区域垃圾回收不适合复制算法(因为老生代空间过大)
- 标记增量优化的过程



- 内存问题的外在表现
  - 页面出现延迟加载或经常性暂停(原因肯定是代码瞬间占据内存导致触发内存清理)
  - 页面持续性出现糟糕的性能
  - 页面的性能随时间延长越来越差(伴随着内存泄漏导致)
- 界定内存为题的标准
  - 内存泄漏：内存使用持续升高
  - 内存膨胀：在多数设备上都存在性能问题
  - 频繁垃圾回收：通过内存变化图进行分析
- 监控内存的方式
  - 浏览器任务管理器
  - timeline时序图记录（会记录所有时间点的内存走势）
  - 堆快照查找分析DOM
  - 判断是否存在频繁的垃圾回收
- 为什么确定频繁的垃圾回收
  - GC工作时应用程序是停止的
  - 频繁且过长的GC会导致应用假死
  - 用户使用中感知应用卡顿
  - 确定频繁的垃圾回收
    - Timeline 中频繁的上升下降
    - 任务管理器中数据频繁的增加减少

---

jsperf 使用流程

- 使用GitHub账号登陆
- 填写个人信息（非必须）
- 填写详细的测试用例信息（title, slug）
- 填写准备代码（DOM操作时经常使用）
- 填写必须有setup（相当于是测试前的准备工作）与teardown（相当于是测试后的销毁工作）代码
- 填写测试代码片段

慎用全局变量

缓存全局变量

将使用中无法避免的全局变量缓存到局部，会让性能稍微有一些提升

通过原型对象添加附加方法

通过原型对象添加方法，跟直接添加方法，性能运行差别比较大

闭包

闭包是一种强大的语法

闭包使用不当很容易出现内存泄漏

不要为了闭包而闭包

闭包会降低程序的性能，能不用就不用

避免属性访问方法的使用

使用属性访问方法，会大大降低程序性能，（属性访问方法就是，特意写一个函数来返回本函数里面的属性变量）

for循环的优化

```
var arrList = []
arrList[10000] = 'icoder'

for (var i = 0; i < arrList.length; i++) {
  console.log(arrList[i])
}

for (var i = 0; i < arrList.length; i++) {
  console.log(arrList[i])
}
```

会稍微提升一定的性能

循环方法中

forEach 性能最好

for

for in

DOM节点优化

节点的添加操作必然会有回流和重绘

文档碎片优化节点添加

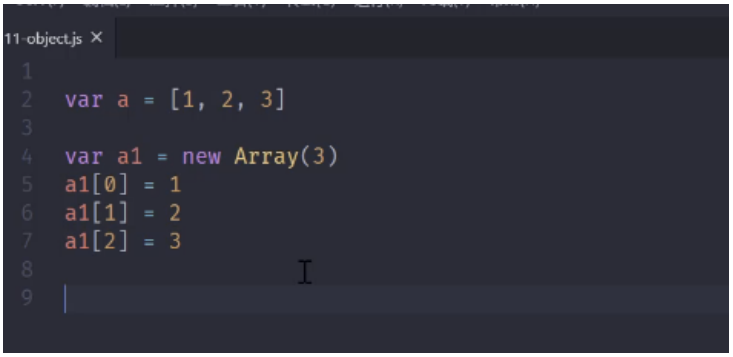
```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>优化节点添加</title>
</head>
<body>
  <script>

    for (var i = 0; i < 10; i++) {
      var oP = document.createElement('p')
      oP.innerHTML = i
      document.body.appendChild(oP)
    }

    const fragEle = document.createDocumentFragment()
    for (var i = 0; i < 10; i++) {
      var oP = document.createElement('p')
      oP.innerHTML = i
      fragEle.appendChild(oP)
    }

    document.body.appendChild(fragEle)

  </script>
</body>
</html>
```

克隆优化节点操作

```
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>克隆优化节点操作</title>
</head>
<body>
  <p id="box1">old</p>

  <script>

    for (var i = 0; i < 3; i++) {
      var oP = document.createElement('p')
      oP.innerHTML = i
      document.body.appendChild(oP)
    }

    var oldP = document.getElementById('box1')
    for (var i = 0; i < 3; i++) {
      var newP = oldP.cloneNode(false)
      newP.innerHTML = i
      document.body.appendChild(newP)
    }

  </script>

</body>
</html>
```

直接量替换 new Object

不使用new 的方式性能会更好


11-object.js
1
2 var a = [1, 2, 3]
3
4 var a1 = new Array(3)
5 a1[0] = 1
6 a1[1] = 2
7 a1[2] = 3
8
9