Part 2 函数式编程 + 性能优化

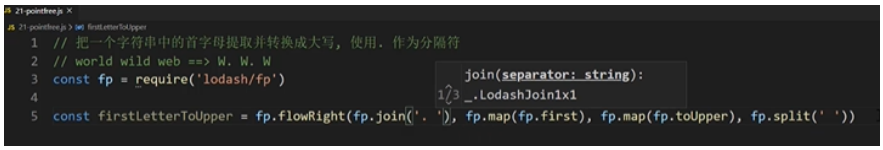
笔记本： JavaScript
创建时间： 2020/5/30 15:59 **更新时间：** 2020/6/7 23:41
作者： 兔子先生

---

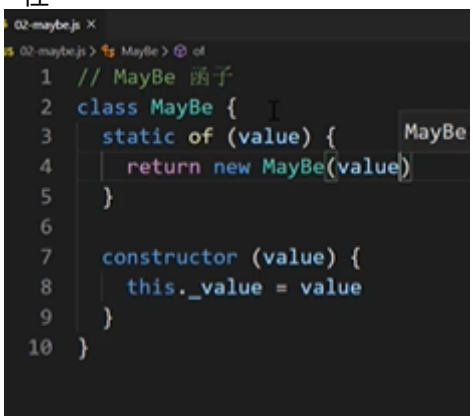
1. **函数式编程**
    1. 是什么?
        1. 一种编程范式（编程思想和实现），还有面向过程编程、面向对象编程
        2. 面向对象编程：（封装、继承、多态，对现实事物的特点进行抽象）；
        3. FP：
            1. 对运算过程抽象
            2. y = f（x） 数学式的函数，；表示映射关系
            3. 纯函数：确定输入确定输出，无可观察的副作用；
            4. 函数式编程不会保留中间结果，所以变量不可变（无状态）
    2. 为什么使用?
        1. 主流框架react、vue借鉴函数式编程实现
        2. 避免this问题
        3. 方便自动化测试
        4. 方便tree shaking
        5. 有很多库函数式编程的库帮助开发 lodash underscore ramda
    3. 特点：纯函数、柯里化、函数组合
        1. 函数是一等公民
            1. 一等公民：可以存储为变量；可以作为函数参数；可以作为函数返回值
            2. 入参和返回值都相同的函数可以认为是一样的函数
        2. 高阶函数 HOF high-order function
            1. 函数作为参数
            2. 函数作为返回值
            3. 函数式编程是对运算过程的抽象，屏蔽实现细节，简化代码；高阶函数就是实现方法
        3. 闭包
            1. 函数和其周围的状态（词法环境）的引用捆绑在一起，如，函数内部返回函数
            2. 当函数的内部状态被内部另一个执行上下文引用时，及时延长了状态的作用域
        4. 纯函数：
            1. 可缓存
            2. 可测试
            3. 并行处理（针对多线程，如：web worker），纯函数运行只依赖参数，限定了作用域，所以不会出现因为多线程访问共享内存而冲突的问题
        5. 柯里化：对函数**降维**，方便组合；将一个接收多个参数的函数改为闭包形式，分次接受参数，越先调用的参数尽可能稳定
            1. 支持分批传递参数
            2. arguments.length < fun.length 实际传递的参数数量小于函数的形参数量，则返回一个新函数继续接收参数
            3. **让函数的粒度更小，便于组合**
            4. **对函数参数实现缓存**
        6. 函数组合：隐藏柯里化函数的中间执行结果，只返回最终的函数
            1. 接受多个函数，从右向左执行

2. 满足结合律，compose(compose(f, g), h) = compose(f, compose(g, h))
3. lodash/fp： lodash中柯里化的方法（函数优先，数据滞后）fp.flowRight

7. point free：将数据处理过程定义成与数据无关的合成运算，不需要指明代表数据的参数，
    1. 不需要指明要处理的数据
    2. 合成运算
    3. 需要提前定义辅助计算函数
    4. 命令式？？ ===》 函数计算的步骤拆解



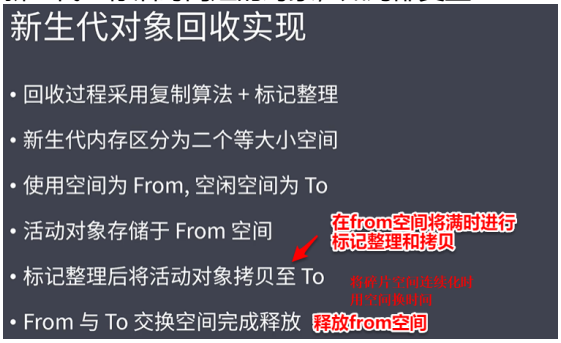
8. 函子：一个特殊的类，针对不可避免的带副作用的操作进行纯化（每次返回一个函子），将副作用处理掉或者延迟执行
    1. 特征
        1. 内部封装一个值value
        2. 对外暴露一个map方法，接收一个参数（纯函数）对value进行处理
        3. 返回一个函子
    2. 问题：
        1. 执行中的异常和错误：如数据判空和类型不匹配
        2. value不让外部访问，为啥不定义成static：：静态属性只能通过 MayBe.value = 1来定义，es6类内部不支持定义静态属性

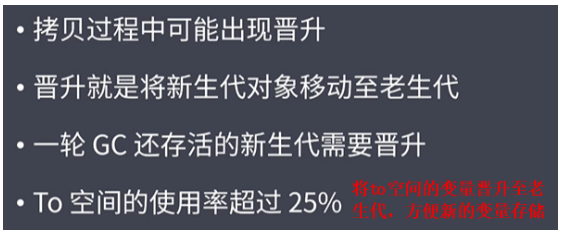


    3. 作用：
        1. 控制副作用
        2. 处理异步问题
        3. 处理异常
    4. MayBe函子：处理空置异常
    5. Either函子：提供if else功能
    6. IO函子：延迟执行副作用函数，让调用者去处理副作用；通过函数包装value，将map接收到的函数与该函数组合，最终返回一个_value为函数的函子对象，调用_value() 触发函数（可能带有副作用）执行？？？？？？应用
    7. monad函子：解决函子嵌套问题
4. 应用场景
5. 相关库：
    1. lodash
    2. folktale：功能性函数较少，多为comppose、curry、task、MayBe等函数和函子

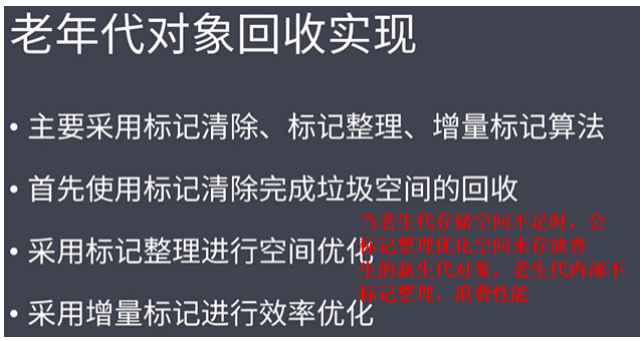
2. **性能优化**

1. 为什么需要内存管理：内存泄漏
    1. 主动 申请-使用-释放 空间
    2. JS中的垃圾：（内存自动分配）对象不再被引用；对象不能从跟上被访问到
    3. 可达对象：从跟上出发找到的（引用、作用域链）；跟：全局变量对象
    4. GC算法：garbage collection
        1. 找到垃圾，释放回收空间
            1. 程序不再使用
            2. 程序运行中引用不到的变量
        2. 常见GC算法：
            1. 引用计数
                1. 计数器设置引用次数，为0则清除
                2. 优点：
                    1. 发现垃圾**立即回收**
                    2. 减少内存溢出的可能，在内存占用较高时，GC开始回收空间
                3. 缺点：
                    1. 不能回收循环引用的空间
                    2. 时间开销大：需要动态监控空间的引用情况
            2. 标记清除
                1. 遍历标记**活动对象（可达）**，遍历清除没有标记的对象（非活动）同时清除第一次遍历中的标记，被回收的空间回放在一个空闲列表里方便被申请
                2. 红皮书：标记所有对象， 遍历去除活动对象的标记，遍历完成后删除仍有标记的对象（均可）
                3. 优点：
                    1. 解决了循环引用
                4. 缺点：
                    1. 空间碎片化：回收到的空间不连续
            3. 标记整理
                1. 增强标记清除法：在第二步清除未标记的空间时，先对空间进行整理，使空间变连续
            4. 分代回收---v8
        3. V8
            1. 空间管理？？
            2. JIT即时编译：
                1. 普通编译器：源代码---》字节码
                2. V8：源码---》机器码（直接可以执行）
            3. 内存设限：64位 1.5G　32位 800M
                1. js是web应用，这样的内存大小足够运行（考虑整个硬件的内存分配）
                2. 当前最差的回收耗时在一次1s，内存增大可能会让用户感知到这个过程（因为回收是阻塞js执行的）
            4. GC算法
                1. 分代回收：因为内存设限，V8内存空间一分为二，小的部分存储新生代对象（64位 32M、32位 18M）（普通变量存在栈中，使用完成即出栈，不需要进行回收）
                    1. 新生代：存活时间短的对象，如局部变量





2. 老生代：全局变量，闭包变量，新生代晋升的对象；垃圾回收会阻塞js执行，增量标记指分批次做标记



2. 空间复制
3. 标记清除
4. 标记整理
5. 标记增量：提升效率

2. 内存问题的表现：内存泄漏（内存使用持续走高）、内存膨胀（在少数设备上存在，则可能是硬件不够）、频繁垃圾回收
    1. 页面卡顿
    2. 随着使用时长变得卡顿
    3. 等
3. 监控方式
    1. 浏览器任务管理器 shift + esc
    2. timeline时序图 performance
    3. 堆快照查找 分离dom memory（分离dom是指在dom树不村子但是在js中被引用的元素；垃圾dom值在dom中不存在，在js中也不被引用）
    4. 判断是否频繁垃圾回收