books techno

@reference
范畴学
集合论
lambda 运算

函数式一等公民

1. 参数数量不进行限制
2. 变量命名可以更加通用
3. 不使用 this

concepts:
纯函数：相同的输入，永远会得到相同的输出，并且没有任何可以观察的副作用

```
code example1: slice and splice
```

将变量编程不可变对象 immutable

```
var immutableState = Object.freeze({
  minimum: 21,
});
```

> 副作用： 在计算结果的过程中，系统状态的一种变化，或者与外部世界进行的可观察的交互

- 更改文件系统
- 往数据库插入纪录
- 发送一个 http 请求
- 可变数据
- 打印/log 标准输出，标准错误
- 获取用户输入
- DOM 查询
- 访问系统状态 cpu,memory

使用 Functor 和 Monad 来进行控制副作用

纯函数的好处：

1. 具有可缓存性 memoize 进行缓存
2. 可移植性、自文档化 可移植性可以将函数序列化通过 socket 发送，也能在 web workers 中运行
3. 可测试性 quickcheck
4. 合理性 引用透明性
5. 并行代码 并行运行任意纯函数，因为纯函数不需要访问共享的内存，也不会因为副作用而进入竞争态

写纯函数的工具 1：柯里化 curry

1. 参数的预加载方式

写纯函数的工具 2： 组合 compose

1. 组合具有结合律

```
var associative = (compose(f, compose(g, h)) = compose(compose(f, g), h));
```

2. pointfree: 永远不用说出自己的数据
3. debug 问题：在没有局部调用之前，就组合接收两个参数的函数， 解决方案： 使用 trace 进行跟踪
4. 组合具有单位律 `compose(id, f)=compose(f,id)=f;`

组合的背景理论

范畴学： 是数学中的一个抽象分支，能够形式化如集合论，类型论，群论，和逻辑学
处理内容：对象(object),态射(morphism),变化式(transformation)
范畴：

- 对象的收集
- 态射的收集
- 态射的组合
- identity 这个独特的态射

identity 函数特性

```
var id = function (x) {
  return x;
};
compose(id, f) = compose(f, id) = f;
```

代码优化知识：JIT
函数式编程 vs 命令式循环

Hindley-Milner 类型签名系统

类型签名： 自由定理概念，类型推断，编译时检测

1. 函数类型推导注释： 类型签名推理
2. 缩小可能性范围
3. 自由定理
4. 类型约束：签名可以把类型约束为一个特定的接口(interface) `sort :: Ord a => [a] -> [a]`

自由定理公式 1：

```
compose(f, head)=compose(head,map(f)); compose(map(f), filter(compose(p,f)) => compose(filter(p), map(f))
```

```
// id :: a -> a
var id = function(x) {return x;}
// map :: (a -> b) -> [a] -> [b]
var map = curry(function(f, xs) {
  reutrn xs.map(f);
})
// reduce: (b -> a -> b) -> b -> [a] -> b
var reduce = curry(function(f, x, xs) {
  return xs.reduce(f, x);
})
```

容器

如何处理 控制流(control flow), 异常处理(error handling), 异步操作(asynchronous actions), 状态(states), 作用(efforts)
创建容器： 容器中必须能够装载任意类型的值

```
var Container = function (x) {
  this._value = x;
};
Container.of = function (x) {
  return new Container(x);
};
```

创建一个 Functor 函子

```
Container.prototype.map = function (f) {
  return Container.of(f(this._value));
};
```

Functor 函子

概念：Functor 是实现了 map 函数并且遵守一些特定规则的容器类型
使用 of 和 map，让容器自己去运行函数能给我们带来什么好处？ 答：抽象

```
// Maybe函子
var Maybe = function (x) {
  this._value = x;
};
Maybe.of = function (x) {
  return new Maybe(x);
};
Maybe.prototype.isNothing = function () {
  return this._value == null;
};
Maybe.prototype.map = function (f) {
  return this.isNothing() ? Maybe.of(null) : Maybe.of(f(this._value));
};
```

Maybe 处理 null 值之后，直接不执行后序的逻辑