用高阶函数做抽象

本文想通过一个例子来说明高阶函数如何能帮助我们建立更高层次的抽象。

函数是一种抽象方式，描述了一系列操作的集合，它提供了一种直接在抽象的层次上工作的能力。举个例子：求特定数值的立方，函数实现可以是：

```
function cube(x) {
  return x * x * x;
}
```

这里函数描述了 求立方 这一个概念,而不依赖某一个具体的数。于是，函数 cube 能很方便地被我们随处可用。我们也可以不去定义这样一个函数，而是用基本表达式去计算:

```
3 * 3 *3
a * a * a
b * b * b
```

但这样我们就永远只能在编程语言的基础操作层面工作。 建立抽象 ，而后直接在更高层面上工作是函数提供的能力，也是一种分析和解决问题的技巧。抽象是一个十分强大的工具，某种程度来说，抽象能力决定了编程能力。那么，高阶函数又是如何帮助我们建立更高层次的抽象呢?

从求和开始

我们先从一个简单的需求开始：求和。

考虑下面的函数:

1. 计算从a到b各整数之和：

```
function sumInteger(a, b) {
  const term = x => x;

  let res;
  if (a > b) res = 0;
  else res = term(a) + sumInteger(a + 1, b);
  return res;
}
```

2. 求a到b每个整数的平方和：

```
function sumSquare(a, b) {
    const term = x => Math.pow(x, 2);

    let res;
    if (a > b) res = 0;
    else res = term(a) + sumSquare(a + 1, b);
    return res;
}
```

3. 计算下面序列的和:

$$\frac{1}{1\cdot 3}+\frac{1}{5\cdot 7}+\frac{1}{9\cdot 11}+\cdots$$

书上说这个式子将收敛到 $\pi/8$(我不会证明):

```
function piSum(a, b) {
  const term = x => 1 / (x * (x + 2))

  let res;
  if (a > b) res = 0;
  else res = term(a) + piSum(a + 4, b);
  return res;
}
```

为了便于观察，我将上述三个例子中的求值函数均命名为 `term` 。可以发现，上面三个函数包含有相同的计算模式,只是在过程中使用的具体函数有所不同

1. 对a应用某一个函数（term），计算出备加数
2. 将备加数与当前结果相加
3. 找到下一个a值，在当前结果的基础上开始新一轮求和运算

存在相同模式就在提醒我们，此处可以建立抽象。

利用高阶函数抽象求和这一过程

我们可以通过填充下面的模版，得到上面的各个函数:

```
function <name>(a, b) {
  let res;
  if (a > b) {
    res = 0;
  } else {
    // term(a)返回a对应的备加数
    // next用于返回下一个a
    res = term(a) + name(next(a), b)
  }
}
```

把上述模版的name/next翻译为形式参数，可得到高阶函数sum_HOF：

```
/**
 * 高阶函数sum
 * @param {number} a 所求范围起始点
 * @param {number} b 所求范围终止点
 * @param {function} next 求下一个a
 * @param {function} term 求将要进行累加的函数值
 */
function sum_HOF(a, b, next, term) {
  let res;
  if (a > b) {
    res = 0;
  } else {
    res = term(a) + sum_HOF(next(a), b, next, term);
  }
  return res;
}
```

这里我们要回答两个问题：

1. `sum_HOF` 的特别之处是什么呢?
    - 在于它表达了 求和 这一概念。
2. 为什么特别?
    - 我们回头看 `sumInteger` 、 `sumSquare` 和 `piSum` 这三个函数，本质上，他们表述的同一件事：对序列求和。但由于它们分别依赖了不同的求值关系(可认为是term)， 所以他们分别有不同的代码实现。而 `sum_HOF` 将这求值关系这一层抽象出来作为形式参数，从而实现不管你用什么求值运算，只要将对应的函数传进来，就能得到想要的累加结果。

现在我们不妨利用 `sum_HOF` ,重写上面三个例子：

一、求范围内自然数之和

```
// term
const term = x => x;
// next
const inc = (a) => a + 1;
const res = sum_HOF(1, 4, inc, term);
console.log(res) // 输出: 10
```

二、求范围内自然数乘积之和

```
// term
const squareTerm = (x) => x * x;
// next
const inc = (a) => a + 1;
const res = sum_HOF(1, 4, inc, squareTerm);
console.log(res) // 输出: 30
```

三、求piSum

```
const piTerm = (x) => 1 / (x * (x + 2));
const next = (x) => x + 4
const piSumRes = sum_HOF(1, 1000, next, piTerm)
console.log(piSumRes)   // 输出: 0.7490014975044914
```

对比来看，两个版本的实现方式有着明显不同。哪个更好您说的算。
下面举个例子，介绍当我们有了这个 `sum` 高阶函数之后，还能做到哪些意想不到的事儿?

利用高阶函数求定积分

定积分的计算公式之一:

$$\int_a^b f = [f(a+\frac{dx}{2}) + f(a+dx+\frac{dx}{2}) + f(a+2dx+\frac{dx}{2}) + \cdots]dx$$

我们看，这个公式的形式主要由两部分构成:

1. 一系列项的加和
2. 共同的系数 `dx`

我们已经有了表述 `求和` 这一概念的强大工具，要做的就是一个萝卜一个坑把参数传进去

```
function integral(f, a, b, dx) {
    const addDeltaX = (x) => x + dx;    // next, 用于计算出下一个a
    // term 就是函数f
    return dx * sum_HOF((a + dx / 2), b, addDeltaX, f);
}
```

还有一种求导方式：辛普森规则 ,它比上面的规则更精确，公式表述是：

$$\frac{h}{3}[y_0 + 4y_1 + 2y_2 + 4y_3 + 2y_4 + \cdots + 2y_{n-2} + 4y_{n-1} + y_n]$$

其中$h = \frac{(b-a)}{n}$, n是某个偶数，而$y_k = f(a + kh)$。

辛普森规则乍一看似乎复杂一点，但它的特点跟上面的公式相同：

1. 一个求和式
2. 一个共同系数。

所以我们唯一要做的就是，把 sum_HOF 需要的参数找出来。

```
function simpson(f, a, b, n) {
  // 是否偶数
  const isEven = (num) => num % 2 === 0;

  const h = (b - a) / n;

  const yk = (k) => f(a + (k * h));

  const getCoefficient = (k) => {
    if (k === 0 || k === n){
      return 1;
    } else if (isEven(k)) {
      return 2;
    } else {
      return 4;
    }
  }
  const term = k => getCoefficient(k) * yk(k);

  const inc = x => x + 1;
  return (h * (sum_HOF(0, n, inc, term))) / 3;
}
```

更近一步

sum_HOF 很好的解决了 求和 这一类问题。但是仔细思考，我们的抽象工作还有提升空间。为什么呢？因为加法运算符实际也可以表示为一个函数：

```
function add(a, b) {
  return a + b;
}
```

除了 加，实际上还存在有其他的组合形式，比如乘除。我们可以把这些组合形式用一个函数来表示。因此我们可以在此基础上更进一步，得到一个 accumulator 函数来表示使用某些累积函数组合起一系列项的过程：

```
/**
 *
 * @param {function} combiner 累积函数
 * @param {number} null_value 默认值
 * @param {function} term 应用在求值对象上，求出目标值
 * @param {*} a 所求范围的起始值
 * @param {*} next 求出下一轮运算的起始值
 * @param {*} b 所求范围的终止值
 */
function accumulate(combiner, null_value, term, a, next, b) {
  if (a > b) {
    return null_value;
  } else {
    return combiner(
      term(a),
      accumulate(combiner, null_value, term, next(a), next, b)
    );
  }
}
```

如此以来， sum_HOF 不过是 accumulate 的一种特殊形式，倘若我们想写出累乘、累除、累减等函数，不过是换一个 combiner 的功夫。

再进一步

你品，你细品。上面的 accumulate 函数有一个不足之处：我们必须对范围内的每一个数进行运算和累积处理，这太不灵活了。能否扩展为仅对满足条件的数进行操作呢？阿哈，当然可以!只需要加上一个过滤器：

```
function filterAccumulate(combiner, nullValue, term, a, next, b, filter) {
  if (a > b) {
    return nullValue;
  } else if (filter(a)) {
    return combiner(
      term(a),
      filterAccumulate(combiner, nullValue, term, next(a), next, b, filter)
    );
  } else {
    return combiner(
      nullValue,
      filterAccumulate(combiner, nullValue, term, next(a), next, b, filter)
    );
  }
}
```

至此，我们简单了解到高阶函数在建立抽象方面的用处。纸上得来终觉浅，绝知此事要躬行。祝各位都能写出自己满意的代码。