함수형프로그래밍

프로그래밍 패러다임

함수형프로그래밍

- ***순수함수*를 만든다.**

부수효과가 없는 함수, 수학적 함수
들어오는 인자가 같으면 항상 동일한 결과를 리턴
함수가 받은 인자외에 다른 외부 인자에 영향을 끼치지 않는다.
리턴값 외에는 외부와 소통하는 것이 없다.
평가시점이 중요하지 않다. (다른함수의 인자로, 다른 쓰레드라도)

- ***조합성*을 강조**

순수 함수로 모듈화 수준이 높다.

- ***모듈화 수준*이 높다.**

생산성을 높인다.

- ***일급함수***

함수를 값으로 다룬다. 변수에 담을 수 있다. 인자로 넘길수 있다. 함수를 다른 함수에서 실행할수 있다.

함수형프로그래밍

명령형 코드

```
var users = [
  { id: 1, name: 'ID', age: 36 },
  { id: 2, name: 'BJ', age: 32 },
  { id: 3, name: 'JM', age: 32 },
  { id: 4, name: 'PJ', age: 27 },
  { id: 5, name: 'HA', age: 25 },
  { id: 6, name: 'JE', age: 26 },
  { id: 7, name: 'JI', age: 31 },
  { id: 8, name: 'MP', age: 23 }
];
```

1. 30세 이상인 users를 거른다.
2. 30세 이상인 users의 names를 수집한다.

함수형프로그래밍

명령형 코드

```
// 1. 30세 이상인 users를 거른다.
var temp_users = [];
for (var i = 0; i < users.length; i++) {
  if (users[i].age >= 30) {
    temp_users.push(users[i]);
  }
}
console.log(temp_users);

// 2. 30세 이상인 users의 names를 수집한다.
var names = [];
for (var i = 0; i < temp_users.length; i++) {
  names.push(temp_users[i].name);
}
console.log(names);
```

함수형프로그래밍

_filter

```
function _filter(list, predi) {
  var new_list = [];
  for (var i = 0; i < list.length; i++) {
    if (predi(list[i])) new_list.push(list[i]);
  }
  return new_list;
}

var over_30 = _filter(users, function(user) { return user.age >= 30; });
console.log(over_30);
```

- *원래 있는 값을 직접 변경하지 않고 변형된 새로운 값을 리턴하는 값을 만든다.*
- *부수 효과를 없애고 리턴하는 형태로 변경한다.*
- *추상화의 단위가 함수이다. (객체가 아님)*
- *위임하는 함수를 만든다.*
- *함수가 함수를 받아서 원하는 시점에 원하는 인자를 적용해 나간다.*
- *고차함수 (함수를 인자를 받거나, 함수를 함수내에서 실행하거나, 함수를 리턴하는 함수)*
- *다형성이 높다.*

함수형프로그래밍

_map

```
function _map(list, mapper) {
  var new_list = [];
  for (var i = 0; i < list.length; i++) {
    new_list.push(mapper(list[i]));
  }
  return new_list;
}

var names = _map(over_30, function(user) { return user.name; });
console.log(names);
```

- *데이터 형이 어떻게 생겼는지 전혀 보이지 않는다.*
- *재사용성의 극대화*

함수형프로그래밍

_map

```
console.log(
  _map(
    _filter(users, function(user) { return user.age >= 30; }),
    function(user) { return user.name; }));
```

- *대입문을 많이 사용하지 않는 경향이 있다. 값을 만들어 놓고 문장을 내려가면서 프로그래밍하지 않는다.*
- *함수를 통과해 나가면서 한번에 값을 새롭게 만들어 나가도록 프로그래밍 한다.*
- *간결하다.*
- *사이에 변화를 주는 여지가 없어져서 안정성 높고 테스트가 쉬워진다.*

함수형프로그래밍

_each

```
function _each(list, iter) {
  for (var i = 0; i < list.length; i++) {
    iter(list[i]);
  }
  return list;
}

function _filter(list, predi) {
  var new_list = [];
  _each(list, function(val) {
    if (predi(val)) new_list.push(val);
  });
  return new_list;
}
```

- *명령적인 코드가 사라지고, 보다 선언적인 코드로 표현이 되고 단순해 진다.*

함수형프로그래밍

_curry

```
function _curry(fn) {
  return function(a) {
    return function(b) {
      return fn(a, b);
    }
  }
}

var add = _curry(function(a, b) {
  return a + b;
});
console.log(add(5)(3));
```

- *인자로 함수를 받고, 실행하는 즉시 함수를 리턴, 함수가 실행되면 또다시 함수를 실행, 최종 인자로 받아둔 함수를 평가*

함수형프로그래밍

컬렉션 중심 프로그래밍

- *1. 수집하기 - map, values, pluck 등*
- *2. 거르기 - filter, reject, compact, without 등*
- *3. 찾아내기 - find, find_index, some, every 등*
- *4. 접기 - reduce, min, max, min_by, max_by, group_by, count_by*

함수형프로그래밍

Java 지원

- ***Functional Interface*** ***{source code on Github}***
  - *자바의 람다식은 함수형 인터페이스로만 접근이 된다.*
  - *자바에서 제공하는 기본적인 함수형 인터페이스는 아래와 같다.*

  Runnable
  Supplier
  Consumer
  Function<T, R>
  Predicate

- ***Lambda***
  - *함수형 인터페이스는 람다를 이용하여 인스턴스화 될 수 있다.*

  (int a, int b) -> a - b;
  (int a, int b) -> { return a + b; };

- ***Method Reference***
  - *람다 표현식을 더 간단하게 표현하는 방법*

  .toArray(String[]::new)
  .map(String::trim)

함수형프로그래밍

Java 지원

- ***Stream***
  - 함수형 스타일의 동작을 지원하는 java.util.stream 클래스
  1. **생성하기**: 스트림 인스턴스 생성
  2. **가공하기**: 필터링(filtering) 및 맵핑(mapping) 등 원하는 결과를 만들어가는 중간 작업(intermediate operations)
  3. **결과만들기**: 최종적으로 결과를 만들어내는 작업(terminal operations)

  - 가공하기 *(TRANSFORMATIONS)*
  filter, map, flatMap, peel, distinct, sorted, limit, substream

  - 결과만들기 *(ACTIONS)*
  forEach, toArray, reduce, collect, min, max, count, anyMatch, allMatch, noneMatch, findFirst, findArray