Molecular Biomedical Informatics

分 子 生 醫 資 訊 實 驗 室

Web Programming

網 際 網 路 程 式 設 計

Functional programming

using JavaScript

Before that

You should learn JavaScript first

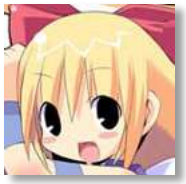
A good reference at JavaScript Garden

Function

- You know function
  -
```
function f1(){
    console.log(1);
}
f1();
```
- Function declaration

Function expression

- Functions in JavaScript are first class objects
  - they can be passed around like any other value
  -
```
var f2 = function(){
  console.log(2);
}
f2();
```



Is

there any difference between f1 and f2

Hoisting

- JavaScript hoists declarations
  - both var statements and function declarations will be moved to the top of their enclosing scope
  - JavaScript-Garden: Scopes and Namespaces
- In fact, all function declarations are automatically converted to function expressions and hoisted
  - try dump f1 and f2
  - the place matters (assignment is not hoisted)

After assignment

- They are the same
  - both can be used as a variable
  -
```
function f3($){
  $();
}
f3(f1);
f3(f2);
```

Anonymous function

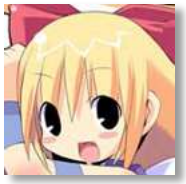
- You should be familiar with
  - ```
    f3(function(){
      console.log(3);
    });
    ```
  - now you know the formal name, function expression
- Anonymous functions can be executed directly,
  - ```
    (function(){
      console.log(null);
    })();
    ```
  - note the first bracket pair
  - self-executing anonymous function

Parameter

- `function f4($){ console.log($) } f4(4);`

  `var f5 = function($){ console.log($) }; f5(5);`

  `(function($){ console.log($) })(null);`
- $ is a local variable, easy?
- ```
  function f6($){
    var _ = 0;
    _ += $;
    console.log(_);
  }
  f6(6); f6(6);
  ```
  ```



What's

the problem of using global variables

Closure/閉包

```
function _f7(){
  var _ = 0;
  function f($){
    _ += $;
    console.log(_);
  }
  return f;
}
var f7 = _f7();
f7(7); f7(7);
```

- `f` is a closure, the `return` exposes it
- `f7` might be called an exposed closure
  - in JS, it is a function
- `_f7` is sometimes called a factory (of an object)
- `_` is not accessible except using the exposed functions
  - a conceptually private variable of `f7`
- to `f`, is `_` local or global?

Free/bound variable

閉包（Closure）

JavaScript-Garden: Closures and References

More practical

- ```
  var f8 = (function(){ // *
    var _ = 0;
    return function($){ // anonymous f
      _ += $;
      console.log(_);
    }
  })();
  f8(8); f8(8);
  ```
- * is anonymous and self-executed _f7
  ```

Even more practical

```
var f9 = {
  add: (function(){
    var _ = 0;
    return function($){
      _ += $;
      console.log(_);
      return this;
    }
  })(),
}
f9.add(9).add(9);
```

- Utilize the return value of the closure
- Chaining
  - you may see this pattern in many advanced JS code (such as jQuery plugins)
- Note the extra comma
  - my personal preference

this

- In JS, `this` is different to most other programming languages
- There are exactly five different ways of binding `this`
  - `this;` // in the global scope, it simply refers to the global object
  - `f();` // when, calling a function, it again refers to the global object
  - `o.f();` // calling a method, it refers to the object (o here)
- Yes, two were skipped, see JavaScript-Garden: How this Works

Intuitive?

- ```
  o.f = function() {
    function f() {
      // this refers to the global object
    }
    f();
  }
  ```
  - a common misconception is that this inside of f refers to o; while in fact, it does not
- Functional scoping instead of block scoping
  - JavaScript-Garden: Scopes and Namespaces
  ```

An object with two closures

```
var f10 = {
  add: (function(){
    var _ = 0;
    return function($){
      _ += $;
      console.log(_);
      return this;
    }
  })(),
  sub: (function(){
    var _ = 0;
    return function($){
      _ -= $;
      console.log(_);
      return this;
    }
  })(),
};
f10.add(10).sub(10);
```

- What's the problem

The complete sample

```
var f11 = (function(){
  var _ = 0;
  return {
    add: function($){
      _ += $;
      console.log(_);
      return this;
    },
    sub: function($){
      _ -= $;
      console.log(_);
      return this;
    },
  } // return
})();
f11.add(11).sub(11);
```

- Note the place of the closure
  - you may see this pattern in many advanced JS code (such as jQuery plugins)

The `this` problem

- ```
  f12 = f11.add;
  f12(12);
  f12(12).sub(12);
  console.log(f11.add(11));
  console.log(f12(12));
  ```
- Anyway, `this` is really a problem in practice
  ```

Any Questions?

Functional programming (FP)

- Uses functions as the fundamental element to program (object-oriented programming uses objects instead)
- In FP, functions are as in Mathematics
  - always return the same results when given the same input
- In other programming, functions should be corrected as procedures (a series of actions)
  - prevent redundant code
  - separate code (for modulization/模組化)
  - may return different results when given the same input

FP limits us while programming

- You are familiar with x = x + 1
  - x is a container (a nickname to a memory address)
- In Mathematics, x = x + 1

FP limits us while programming

- You are familiar with x = x + 1
  - x is a container (a nickname to a memory address)
- In Mathematics, x = x + 1
  - no solution
  - you should use x1 = x + 1
  - x and x1 are symbols, their values can be undetermined but cannot be changed
- In FP, variables can not vary

Limitation does not suck

- “Variables can not vary” looks like a disadvantage
- This is somehow like structured programming (結構化程式設計) which forbids the goto statement
- This is an argument
  - powerful/low-end vs. development/high-end
  - concept/trend changes (FP is very old, 1950s)

Limitation prevents side-effects

- Large-scale/collaborative works favor abstraction, modulization…
- Mutable variable is a critical side-effect of parallel computing (cloud computing, big data…)
  - race conditions (ticket system), synchronization…
  - FP has no mutable variable
- 關於函數編程（一）On Functional Programming

for

- ```
  for (var i = 1; i < 5; i++) {
    // do something
  }
  ```
  - i changed
- ```
  [1, 2, 3, 4].forEach(function(i){
    // do something
  });
  ```
  - no mutable variable
  - pure FP languages have better support for list

sum()

- ```
  var sum = function($){
    if (0 == $.length) return 0;
    return $[0] + sum($.slice(1));
  };
  ```
- Yes, no loop, everything is recursive
  - most FP languages optimize tail-recursion with loop
  - What is tail-recursion?
  ```

prod() and and()

- ```
  var prod = function($){
    if (0 == $.length) return 1;
    return $[0] * prod($.slice(1));
  };
  ```
- ```
  var and = function($){
    if (0 == $.length) return true;
    return $[0] && and($.slice(1));
  };
  ```
- All similar?

foldr()

- The above three functions traverse a list and combine them (two elements at a time) into a value
  - the only two differences are the combine function and the empty value
  - can you abstract them?

foldr()

- The above three functions traverse a list and combine them (two elements at a time) into a value
  - the only two differences are the combine function and the empty value
  - can you abstract them?
- ```
  var foldr = function(f, e){
    return function _($){ // named function expression
      if (0 == $.length) return e;
      return f($[0], _($.slice(1)));
    }
  };
  ```
  ```

foldr()

- The above three functions traverse a list and combine them (two elements at a time) into a value
  - the only two differences are the combine function and the empty value
  - can you abstract them?
- ```
  var foldr = function(f, e){
    return function _($){ // named function expression
      if (0 == $.length) return e;
      return f($[0], _($.slice(1)));
    }
  };
  ```
- ```
  var sum = foldr(function(a,b){return a+b}, 0);
  var prod = foldr(function(a,b){return a*b}, 1);
  var and = foldr(function(a,b){return a&&b}, 1);
  ```

Abstraction

- Examples
  - from `goto` to `for`, `while`… → abstraction of control structure
  - sub → abstraction of procedure
  - encapsulation → abstraction of data
  - OOP, FP → abstraction of architecture
- foldr() is the famous fold/reduce operation
  - the above is for list, other structures such as binary tree have their own fold function
  - a higher-order function, which use functions as data
- 關於函數編程（二）摺疊與抽象化

map

- Another common programming pattern
- `map(square, [1, 2, 3, 4])` returns `[1, 4, 9, 16]`

map

- Another common programming pattern
- `map(square, [1, 2, 3, 4])` returns `[1, 4, 9, 16]`
  -
```
var map = function(f, $){
    if (0 == $.length) return [];
    return [f($[0])].concat(map(f, $.slice(1)));
};
```
- Similar to `sum()`?

map

- Another common programming pattern
- `map(square, [1, 2, 3, 4])` returns `[1, 4, 9, 16]`
  -
```
var map = function(f, $){
  if (0 == $.length) return [];
  return [f($[0])].concat(map(f, $.slice(1)));
};
```
- Similar to `sum()`?
  -
```
var map = function(f, $){
  return foldr(function(a,b){
    return [f(a)].concat(b)}, []
  )($);
};
```

Another map implementation

- `map(square)` returns a function that transform `[1, 2, 3, 4]` to `[1, 4, 9, 16]`

Another map implementation

- `map(square)` returns a function that transform `[1, 2, 3, 4]` to `[1, 4, 9, 16]`
  -
```
var map = function(f){
  return function($){
    if (0 == $.length) return [];
    return [f($[0])].concat(map(f)($.slice(1)));
  }
};
```
- Use `foldr()`

Another map implementation

- `map(square)` returns a function that transform `[1, 2, 3, 4]` to `[1, 4, 9, 16]`
  -
```
var map = function(f){
  return function($){
    if (0 == $.length) return [];
    return [f($[0])].concat(map(f)($.slice(1)));
  }
};
```
- Use `foldr()`
  -
```
var map = function(f){
  return foldr(function(a,b){
    return [f(a)].concat(b)
  }, []);
};
```

With this abstraction

we can start to discuss the property of these programming patterns

(e.g. fold(map) is a fold)

關於函數編程（三）摺疊-映射融合定理

sumsq()

- ```
  var sumsq = function($){
    return sum(map1(square, $))
  };
  var sumsq = function($){
    return sum(map2(square)($))
  };
  ```
- Waste?
  ```

sumsq()

- ```
  var sumsq = function($){
    return sum(map1(square, $))
  };
  var sumsq = function($){
    return sum(map2(square)($))
  };
  ```
- Waste?
  - [1, 4, 9, 16]
  ```

sumsq()

- ```
  var sumsq = function($){
    return sum(map1(square, $))
  };
  var sumsq = function($){
    return sum(map2(square)($))
  };
  ```
- Waste?
  - `[1, 4, 9, 16]`
- ```
  var sumsq = foldr(function(a,b){
    return square(a)+b
  }, 0);
  ```

A more practical code

```
Array.prototype.sumsq = function(){
  return foldr(function(a,b){
    return square(a)+b}, 0
  )(this);
};
[1, 2, 3, 4].sumsq();
```

- Extend an object, like jQuery plugins do on the jQuery object
- JavaScript-Garden: The Prototype
  - JS does not feature a classical inheritance model; instead, it uses a prototypal one
  - knowing this helps OOP with JS

A more practical code

```
Array.prototype.sumsq = function(){
  return foldr(function(a,b){
    return square(a)+b}, 0
  )(this);
};
[1, 2, 3, 4].sumsq();
```

- Extend an object, like jQuery plugins do on the jQuery object
  - there are other ways, see jQuery plugin tutorials
- JavaScript-Garden: The Prototype
  - JS does not feature a classical inheritance model; instead, it uses a prototypal one
  - knowing this helps OOP with JS

Today's assignment
今天的任務

Prepare your final exhibition