```
1. Callback Hell
  1)Node.js 프로그래밍의 특징은 비동기이고 콜백함수를 사용한다는 것이다.
      function asyncTask(path, function(result){
        //결과 처리
      }

  2)결국 비동기 방식으로 동작이 반복되면 콜백함수 안에서 콜백함수를 작성하는 구조가 발생하게 된다.
    -task1 실행 이후에 task2 실행
    -task1 실행 결과를 이용해서 task2 실행
    -결국 콜백의 연속된 호출이 발생하게 된다.

  3)연속된 비동기 동작의 예
    -이미지 업로드 후 데이터베이스에 저장
    -다수의 이미지에서 썸네일 생성 후 업로드

  4)연속된 비동기 동작 코드
      function task1(){
        console.log('First Task Started');
        setTimeout(function(){
          console.log('First Task Done');
        }, 3000);
      }

      function task2(){
        console.log('Second Task Started');
        setTimeout(function(){
          console.log('Second Task Done');
        }, 1000);
      }

      task1();
      task2();
      ------------------------------------------------
      이 함수를 실행하면 순차대로 동작하지 않는다.
      First Task Started
      Second Task Started
      Second Task Done
      First Task Done

  5)보통 비동기 API에서 하나의 동작이 끝나면 콜백 함수를 이용해서 다음 동작을 수행하도록 작성한다.
    -task1 실행 이후에 task2 실행
      task1(args1, function(result){
        task2(args2, function(result){
        });
      });

  6)실행예제
      function task1(callback){
        console.log('First Task Started');
        setTimeout(function(){
          console.log('First Task Done');
```

```
            callback();
        }, 3000);
    }

    function task2(){
        console.log('Second Task Started');
        setTimeout(function(){
            console.log('Second Task Done');
        }, 1000);
    }

    task1(function(){
        task2();
    });
    ------------------------------------------------
    First Task Started
    First Task Done
    Second Task Started
    Second Task Done

7)task1 실행 결과를 task2에서 사용해야 하는 경우
    task1(args1, function(result){
        var args2 = result.value;
        task2(args2, function(result){

        });
    });

8)callback의 연속으로 인해 발생하는 현상
  -callback hell(콜백 지옥)
    task1(a, b, function(err, result1){
        task2(c, function(err, result2){
            task3(d,e,f, function(result3){
                task4(h, i, function(result4){
                    //비동기 동작
                }; //end task4
            };  //end task3
        };    //end task2
    };      //end task1

9)callbak hell 탈출하기
    task1(a, b, task1Callback);

    function task1Callback(result){
        task2(c, task2Callback);
    }

    function task2Callback(result){
        //task3 호출
    }
  -하지만 중간에 비동기 함수를 부르는 순서가 바뀌거나 로직이 바뀌거나 할 경우가 발생할 수 있기 때문에 100%
```

완전한 해결책은 아니다.

10)흐름 제어를 위한 모듈로는 async, step 그리고 promise 모듈이 있다.

2. Aync
1)비동기 동작의 흐름을 제어하기 위한 모듈
2)https://github.com/caolan/async
3)http://caolan.github.io/async/
4)비동기 제어 흐름 패턴들에 대해 가장 광범위한 자원을 제공하는 모듈
5)Install
$ npm install async
|
|-----async@2.4.0

6)대표적인 기능
-행위 순서 제어
--series, seriesEach
--parallels
--waterfall
-콜렉션(배열, 객체)
--each
--forEachOf
--map, filter

7)함수들
-waterfall : 함수가 순서대로 호출되고, 모든 함수의 결과는 마지막 콜백에 배열로 전달된다
-series : 함수들이 순서대로 호출되며, 선택적으로 결과가 마지막 콜백에 배열로 전달된다.
-parallel : 함수들이 병렬로 실행되며, 실행이 완료되면 결과들이 마지막 콜백으로 전달된다.
-whilst : 한 함수를 반복적으로 호출하되, 사전 준비 테스트가 false를 반환하거나 오류가 발생하는 경우에만 마지막 콜백이 호출된다.
-queue : 지정된 동시 제한 수까지 함수를 병렬로 호출하고, 함수 중 하나가 완료되면 새로운 함수가 큐에 들어간다.
-until : 한 함수를 반복적으로 호출하되, 후처리 테스트가 false를 반환하거나 에러가 발생하는 경우에만 마지막 콜백이 호출된다.
-auto : 함수가 요구사항을 기반으로 호출되며, 각 함수는 이전 콜백의 결과를 받는다.
-iterator : 각 함수가 다음 함수를 호출하며, 각각 다음에 있는 반복자에 접근할 수 있다.
-apply : 이전에 적용된 인수를 가지고 다른 제어 흐름 함수와 결합되는 연속 함수다.
-nextTick : Node의 process.nextTick을 기반으로 이벤트 루프의 다음번 루프에서 콜백을 호출한다.

8)series(tasks, [callback])

```
async.series(
  [
    task1,
    task2,
    task3
  ],
  function(err, results){
    //완료 callback
  }
);
```

```
150
151   8)순차 실행
152     -callback 호출 : 다음 태스크로 진행
153     -태스크 완료 : 다음 태스크로 실행
154       function(callback){
155         //task 성공
156         callback(null, result);
157         //error가 발생하지 않으면 첫 파라미터로 null 전달
158       }
159     -완료 콜백으로 동작 결과 전달
160
161   9)순차 실행 시 에러 발생
162     -에러 전달
163       function(callback){
164         //error 발생
165         callback(err, null);
166         //error가 발생했기 때문에 첫 파라미터에 err, 두번째 파라미터는 전달하지 않음.
167       }
168     -다음 태스크 실행 안함
169     -완료 콜백으로 바로 에러 전달
170
171   10)연속 동작 마무리 할 때
172     -serial 완료 콜백
173       async.series(
174         [task1, task2, task3],
175         function(err, results){
176           if(err){
177             //task 진행 중 에러 : callback(err, null);
178             return ;
179           }
180           //마무리 동작 수행
181         }
182     -results에는 각 태스크의 결과가 배열 형태로 전달됨.
183
184     async.series(
185       [
186         function task1(callback){
187           callback(null, 'result1');
188         },
189         function task2(callback){
190           callback(null, 'result2');
191         },
192         function task3(callback){
193           callback(null, 'result3');
194         }
195       ],
196       function(err, results){
197         //results : ['result1', 'result2', 'result3']
198       }
199     );
200
```

```
11)Lab
  <asyncdemo.js>
    var async = require('async');

    function task1(callback){
        console.log('First Task Started');
        setTimeout(function(){
            console.log('First Task Done');
            callback(null, 'First Task Done');
        }, 3000);
    }

    function task2(callback){
        console.log('Second Task Started');
        setTimeout(function(){
            console.log('Second Task Done');
            callback(null, 'Second Task Done')
        }, 1000);
    }

    async.series([task1, task2], function(err, results) {
      if ( err ) {
        console.error('Error : ', err);
        return;
      }
      console.log('비동기 동작 모두 종료 ', results)
    });
    ------------------------------------------------------------
    First Task Started
    First Task Done
    Second Task Started
    Second Task Done
    비동기 동작 모두 종료  [ 'First Task Done', 'Second Task Done' ]

12)Lab
  <asyncdemo1.js>
    var async = require('async');

    function task1(callback){
        console.log('First Task Started');
        setTimeout(function(){
            console.log('First Task Done');
            callback('Error',null);   //error 발생
        }, 3000);
    }

    function task2(callback){
        console.log('Second Task Started');
        setTimeout(function(){
            console.log('Second Task Done');
            callback(null, 'Second Task Done')
```

```
252             }, 1000);
253         }
254
255         async.series([task1, task2], function(err, results) {
256           if ( err ) {
257             console.error('Error : ', err);
258             return;
259           }
260           console.log('비동기 동작 모두 종료 ', results)
261         });
262         ------------------------------------------------------------
263         First Task Started
264         First Task Done
265         Error :  Error
266
267     13)순차 실행2
268       -task로 정보를 전달하려고 할 때 : async.waterfall
269         --async.series()는 각 함수의 결과를 마지막 완료 태스크로 전달한다.
270         --async.waterfall()은 각 함수의 결과를 다음 태스크로 전달한다.
271         --다음 태스크로 전달할 값을 callback의 파라미터로
272         --태스크 함수의 파라미터로 전달하는데, 이전 task의 값을 전달한다.
273
274         function task1(callback){
275           callback(null, 'value');
276         }
277         function task2(args, callback){
278           //위의 task1의 value가 task2의 args로 전달됨.
279           callback(null, 'hello', 'world');
280         }
281
282     14)aync.waterfall 샘플 코드
283         async.waterfall(
284           [
285             function task1(callback){
286               callback(null, 'value');
287             },
288             function task2(args, callback){
289               //task1의 'value'가 args로 전달
290               callback(null, 'value1', 'value2');
291             },
292             function task3(args1, args2, callback){
293               //task2의 'value1'은 args1으로, 'value2'는 args2로 전달
294               callback(null, 'result');
295             },
296           ],
297           function(err, results){
298             //완료 콜백
299           }
300         );
301
302     15)Lab
```

```
  <waterfall.js>
    var async = require('async');

    async.waterfall([
        function(callback) {
            callback(null, 'one', 'two');
        },
        function(arg1, arg2, callback) {
            console.log('In Second function, arg1 = ' + arg1 + ', args2 = ' + arg2);
            callback(null, 'three');
        },
        function(arg3, callback) {
            console.log('In Third function, arg3 = ' + arg3);
            callback(null, 'done');
        }
    ], function (err, result) {
        console.log('result = ' + result);
    });
    ------------------------------------------------------
    In Second function, arg1 = one, args2 = two
    In Third function, arg3 = three
    result = done

16)Lab
  <waterfall1.js>
    var async = require('async');

    async.waterfall([
        myFirstFunction,
        mySecondFunction,
        myLastFunction,
    ], function (err, result) {
        console.log('result = ' + result);
    });

    function myFirstFunction(callback) {
        callback(null, 'one', 'two');
    }

    function mySecondFunction(arg1, arg2, callback) {
         console.log('In Second function, arg1 = ' + arg1 + ', args2 = ' + arg2);
        callback(null, 'three');
    }

    function myLastFunction(arg3, callback) {
        console.log('In Third function, arg3 = ' + arg3);
        callback(null, 'done');
    }
    ------------------------------------------------------
    In Second function, arg1 = one, args2 = two
    In Third function, arg3 = three
```

```
354       result = done
355
356   17)Lab
357     <waterfall2.js>
358       var async = require('async');
359       var fs = require('fs');
360
361       try{
362            async.waterfall([
363                 function readData(callback){
364                      fs.readFile('./data.txt', 'utf8', function(err, data){
365                           callback(err, data);
366                      });
367                 },
368                 function modify(text, callback){
369                      var adjdata = text.replace(/naver\.com/g, 'somecompany.com');
370                      callback(null, adjdata);
371                 },
372                  function writeData(text, callback){
373                      fs.writeFile('./data.txt', text, function(err){
374                           callback(err, text);
375                      });
376                 }
377            ], function(err, result){
378                 if(err) throw err;
379                 console.log(result);
380            });
381       }catch(err){
382            console.log(err);
383       }
384       ------------------------------------------------------
385       C:\NodeHome>type data.txt
386       naver.com
387       google.com
388       daum.net
389       C:\NodeHome>node waterfall
390       somecompany.com
391       google.com
392       daum.net
393
394       C:\NodeHome>type data.txt
395       somecompany.com
396       google.com
397       daum.net
398       C:\NodeHome>
399
400   18)여러개의 비동기를 동시에 실행할 경우
401     -모든 태스크를 마치면 완료 콜백
402     -parallel(tasks, [callback])
403     -사용방식
404       async.parallel([task1, task2, task3],
```

```
405         function(err, results){
406           //['task1의 결과', 'task2의 결과', 'task3의 결과']
407         }
408
409   19)예제 코드
410       async.parallel(
411         [
412           function(callback){
413             callback(null, 'task1의 결과');
414           },
415           function(callback){
416             callback(null, 'task2의 결과');
417           },
418           function(callback){
419             callback(null, 'task3의 결과');
420           }
421         ],
422         function(err, results){
423           console.log('모든 태스크 종료, 결과 : ', results);
424           //['task1의 결과', 'task2의 결과', 'task3의 결과']
425         }
426       );
427
428   20)Lab
429     <parallel.js>
430         var async = require('async');
431
432         async.parallel(
433             [
434                 function(callback) {
435                     setTimeout(function() {
436                         callback(null, 'one');
437                     }, 200);
438                 },
439                 function(callback) {
440                     setTimeout(function() {
441                         callback(null, 'two');
442                     }, 100);
443                 }
444             ],
445             function(err, results) {
446                 console.log(results);
447             }
448         );
449         ----------------------------------------
450         ['one', 'two']
451
452   21)Lab
453     <parallel1.js>
454         var async = require('async');
455
```

```
456         async.parallel(
457              {
458                   one: function(callback) {
459                        setTimeout(function() {
460                             callback(null, 1);
461                        }, 200);
462                   },
463                   two: function(callback) {
464                        setTimeout(function() {
465                             callback(null, 2);
466                        }, 100);
467                   }
468              }, function(err, results) {
469                        console.log(results)
470              }
471         );
472         ----------------------------------------------------
473         { two: 2, one: 1 }
474
475    22)Lab
476      <parallel2.js>
477        var async = require('async');
478         var fs = require('fs');
479
480         try{
481              async.parallel({
482                   data1 : function(callback) {
483                        fs.readFile('./data1.txt', 'utf8', function(err, data){
484                             callback(err, data);
485                        });
486                   },
487                   data2 : function(callback) {
488                        fs.readFile('./data2.txt', 'utf8', function(err, data){
489                             callback(err, data);
490                        });
491                   },
492                    data3 : function(callback) {
493                        fs.readFile('./data3.txt', 'utf8', function(err, data){
494                             callback(err, data);
495                        });
496                   }
497              }, function(err, results) {
498                        if(err) throw err;
499                        console.log(results);
500              });
501         }catch(err){
502              console.log(err);
503         }
504         ----------------------------------------
505         C:\NodeHome>type data1.txt
506         Apples
```

```
        C:\NodeHome>type data2.txt
        Oranges
        C:\NodeHome>type data3.txt
        Peaches

        C:\NodeHome>node parallel2
        { data1: 'Apples', data2: 'Oranges', data3: 'Peaches' }
```

23)Collection과 비동기 동작
-Collection 내 각 항목을 사용하는 비동기 동작
--다수의 파일(배열)을 비동기 API로 읽기
--다수의 파일을 비동기 API로 존재하는지 확인하기
-비동기 순회 동작
--each, eachSeries, eachLimit
--map, filter
--reject, reduct
--...

3. Step
1)순차 및 병렬 실행의 흐름을 간단하게 제어할 수 있게 해주는데 초점을 맞춘 유틸리티 모듈
2)https://github.com/creationix/step
3)A simple control-flow library for node.JS that makes parallel execution, serial execution, and error handling painless.
4)Install

```
$ npm install step
        |
        |-----step@1.0.0
```

5)함수들의 순차적 실행
-비동기 함수 호출을 함수로 감싸서 개체에 매개변수로 전달

6)Lab
<stepdemo.js>

```
var Step = require('step');
var fs = require('fs');

try{
    Step(
        function readData(){
            fs.readFile('./data.txt', 'utf8', this);
        },
        function modify(err, text){
            if(err) throw err;
            return text.replace(/naver\.com/g, 'somecompany.com');
        },
         function writeData(err, text){
            if(err) throw err;
            fs.writeFile('./data.txt', text, this);
        }
    );
}catch(err){
```

```
            console.log(err);
        }
        ------------------------------------------
        C:\NodeHome>type data.txt
        naver.com
        google.com
        daum.net

        C:\NodeHome>node stepdemo

        C:\NodeHome>type data.txt
        somecompany.com
        google.com
        daum.net


4. Promise
  1)비동기 동작의 흐름 제어
  2)https://www.promisejs.org/
  3)JavaScript ES6에 추가
    -Node.js 4.x 이후 모듈 설치 필요 없음.

  4)객체 생성
    new Promise(function(){
      //비동기 동작
    });

  5)Promise 상태
    -pending : 동작 완료 전(The initial state of a promise.)
    -fulfilled : 비동기 동작 성공(The state of a promise representing a successful operation.)
    -rejected : 동작 실패(The state of a promise representing a failed operation.)

  6)Promise 생성 및 상태 반영
    -성공적으로 완료 : fulfill 호출
    -에러 상황 : reject 호출

    new Promise(function(fulfill, reject){
      //비동기 동작
      if(err) reject(err);
      else fulfill(result);
    });

  7)Promise 이후의 동작
    -then
      --fulfilled 상태일 때의 콜백
      --rejected 상태일 때의 콜백

    new Promise(task).then(fulfilled, rejected);

    function fullfilled(result){
      //fulfilled 상태일 때의 동작
```

```
608     }
609     function rejected(err){
610       //rejected 상태일 때의 동작
611     }
612
613   8)Promise를 사용하는 task
614     function task(){
615       return new Promise(function(fulfill, reject){
616         if(success) fulfill('Success');
617         else reject('Error');
618       });
619     }
620     -태스크 사용코드
621       task(arg).then(fulfilled, rejected);
622
623   9)Lab
624     <promise.js>
625       function task1(fullfill, reject) {
626         console.log('Task1 시작');
627         setTimeout(function() {
628           console.log('Task1 끝');
629           //fullfill('Task1 결과');
630           reject('Error msg');
631         }, 300);
632       }
633
634       function fullfilled(result) {
635         console.log('fullfilled : ', result);
636       }
637
638       function rejected(err) {
639         console.log('rejected : ', err);
640       }
641
642       new Promise(task1).then(fullfilled, rejected);
643       ------------------------------------------
644       C:\NodeHome>node promise
645       Task1 시작
646       Task1 끝
647       fullfilled :  Task1 결과
648
649       C:\NodeHome>node promise
650       Task1 시작
651       Task1 끝
652       rejected :  Error msg
```