ES6+

GENERATOR, PROMISE, ASYNC/AWAIT

1 2 3 4 5 6

GENERATOR

BREAKING BLOCK

프로그램을 중도에 멈췄다가 다시 실행할 수 있음

BREAKING BLOCK

프로그램을 중도에 멈췄다가 다시 실행할 수 있음

```
const infinity = (function*(){
  let i = 0;
  while(true) yield i++;
})();
console.log(infinity.next());
....
console.log(infinity.next());
```

BREAKING BLOCK

프로그램을 중도에 멈췄다가 다시 실행할 수 있음

```
const infinity = (function*(){
  let i = 0;
  while(true) yield i++;
})();
console.log(infinity.next());
....
console.log(infinity.next());
```

yield를 이용하면 블록을 중간에 끊어주는 효과가 발생

BLOCKING EVASION

TIME SLICING MANUAL

```
const looper = (n, f, slice = 3)=>{
  let limit = 0, i = 0;
  const runner =_=>{
    while(i < n){
      if(limit++ < slice) f(i++);
      else{
        limit = 0;
        requestAnimationFrame(runner);
        break;
      }
    }
  };
  requestAnimationFrame(runner);
};
```

```
looper(10, console.log);
```

TIME SLICING MANUAL USING GENERATOR

```
const loop = function*(n, f, slice = 3){
  let i = 0, limit = 0;
  while(i < n){
    if(limit++ < slice) f(i++);
    else{
      limit = 0;
      yield;
    }
  }
};
```

TIME SLICING MANUAL USING GENERATOR

```
const loop = function*(n, f, slice = 3){
  let i = 0, limit = 0;
  while(i < n){
    if(limit++ < slice) f(i++);
    else{
      limit = 0;
      yield;
    }
  }
};
const executor =iter=>{
  const runner =_=>{
    iter.next();
    requestAnimationFrame(runner);
  };
  requestAnimationFrame(runner);
};
```

TIME SLICING MANUAL USING GENERATOR

```
const loop = function*(n, f, slice = 3){
  let i = 0, limit = 0;
  while(i < n){
    if(limit++ < slice) f(i++);
    else{
      limit = 0;
      yield;
    }
  }
};
const executor =iter=>{
  const runner =_=>{
    iter.next();
    requestAnimationFrame(runner);
  };
  requestAnimationFrame(runner);
};
```

```
executor(loop(10, console.log));
```

TIME SLICING MANUAL USING GENERATOR

```
const loop = function*(n, f, slice = 3){
  let i = 0, limit = 0;
  while(i < n){
    if(limit++ < slice) f(i++);
    else{
      limit = 0;
      yield;
    }
  }
};
const executor =iter=>{
  const runner =_=>{
    iter.next();
    requestAnimationFrame(runner);
  };
  requestAnimationFrame(runner);
};
```

```
executor(loop(10, console.log));
```

```
const looper = (n, f, slice = 3)=>{
  let limit = 0, i = 0;
  const runner =_=>{
    while(i < n){
      if(limit++ < slice) f(i++);
      else{
        limit = 0;
        requestAnimationFrame(runner);
        break;
      }
    }
  };
  requestAnimationFrame(runner);
};
```

```
looper(10, console.log);
```

TIME SLICING MANUAL USING GENERATOR

```
const loop = function*(n, f, slice = 3){
  let i = 0, limit = 0;
  while(i < n){
    if(limit++ < slice) f(i++);
    else{
      limit = 0;
      yield;
    }
  }
};
const executor =iter=>{
  const runner =_=>{
    iter.next();
    requestAnimationFrame(runner);
  };
  requestAnimationFrame(runner);
};
```

```
executor(loop(10, console.log));
```

```
const looper = (n, f, slice = 3)=>{
  let limit = 0, i = 0;
  const runner =_=>{
    while(i < n){
      if(limit++ < slice) f(i++);
      else{
        limit = 0;
        requestAnimationFrame(runner);
        break;
      }
    }
  };
  requestAnimationFrame(runner);
};
```

```
looper(10, console.log);
```

TIME SLICING MANUAL USING GENERATOR

```
const loop = function*(n, f, slice = 3){
  let i = 0, limit = 0;
  while(i < n){
    if(limit++ < slice) f(i++);
    else{
      limit = 0;
      yield;
    }
  }
};
const executor =iter=>{
  const runner =_=>{
    iter.next();
    requestAnimationFrame(runner);
  };
  requestAnimationFrame(runner);
};
```

```
executor(loop(10, console.log));
```

```
const looper = (n, f, slice = 3)=>{
  let limit = 0, i = 0;
  const runner =_=>{
    while(i < n){
      if(limit++ < slice) f(i++);
      else{
        limit = 0;
        requestAnimationFrame(runner);
        break;
      }
    }
  };
  requestAnimationFrame(runner);
};
```

```
looper(10, console.log);
```

GENERATOR + ASYNC + EXECUTOR

```
const profile = function*(end, next, r){
  const userid = yield $.post('member.php', {r}, next);
  let added = yield $.post('detail.php', {userid}, next);
  added = added.split(",");
  end({userid, nick:added[0], thumb:added[1]});
};
```

GENERATOR + ASYNC + EXECUTOR

```
const profile = function*(end, next, r){
  const userid = yield $.post('member.php', {r}, next);
  let added = yield $.post('detail.php', {userid}, next);
  added = added.split(",");
  end({userid, nick:added[0], thumb:added[1]});
};
```

```
const executor = (end, gene,...arg)=>{
  const next =v=>iter.next(v);
  const iter = gene(end, next,...arg);
  iter.next();
};
```

GENERATOR + ASYNC + EXECUTOR

```
const profile = function*(end, next, r){
  const userid = yield $.post('member.php', {r}, next);
  let added = yield $.post('detail.php', {userid}, next);
  added = added.split(",");
  end({userid, nick:added[0], thumb:added[1]});
};
```

```
const executor = (end, gene,...arg)=>{
  const next =v=>iter.next(v);
  const iter = gene(end, next,...arg);
  iter.next();
};
```

```
executor(profile, console.log, 123);
```

PROMISE

PASSIVE ASYNC CONTROLL

콜백을 보낼 수는 있지만 언제 올지는 모른다.

PASSIVE ASYNC CONTROLL

콜백을 보낼 수는 있지만 언제 올지는 모른다.

```
$.post(url, data, e=>{
  //언제 올까
});
```

PASSIVE ASYNC CONTROLL

콜백을 보낼 수는 있지만 언제 올지는 모른다.

```
$.post(url, data, e=>{
  //언제 올까
});
```

왜 언제가 중요한가?

PASSIVE ASYNC CONTROLL

콜백을 보낼 수는 있지만 언제 올지는 모른다.

```
$.post(url, data, e=>{
  //언제 올까
});
```

왜 언제가 중요한가?

```
let result;
$.post(url1, data1, v=>{
  result = v;
});
$.post(url2, data2, v=>{
  result.nick = v.nick;
  report(result);
});
```

PASSIVE ASYNC CONTROLL

콜백을 보낼 수는 있지만 언제 올지는 모른다.

```
$.post(url, data, e=>{
  //언제 올까
});
```

왜 언제가 중요한가?

```
let result;
$.post(url1, data1, v=>{
  result = v;
});
$.post(url2, data2, v=>{
  result.nick = v.nick;
  report(result);
});
```

PASSIVE ASYNC CONTROLL

콜백을 보낼 수는 있지만 언제 올지는 모른다.

```
$.post(url, data, e=>{
  //언제 올까
});
```

왜 언제가 중요한가?

```
let result;
$.post(url1, data1, v=>{
  result = v; ①
});
$.post(url2, data2, v=>{
  result.nick = v.nick;
  report(result);②
});
```

ACTIVE ASYNC CONTROLL

ACTIVE ASYNC CONTROLL

프라미스는 then을 호출해야 결과를 얻는다.

ACTIVE ASYNC CONTROLL

프라미스는 then을 호출해야 결과를 얻는다.

```
let result;
const promise = new Promise(r=>$.post(url1, data1, r));
promise.then(v=>{
  result = v;
});
```

ACTIVE ASYNC CONTROLL

프라미스는 then을 호출해야 결과를 얻는다.

```
let result;
const promise = new Promise(r=>$.post(url1, data1, r));
promise.then(v=>{
  result = v;
});
```

```
const promise1 = new Promise(r=>$.post(url1, data1, r));
const promise2 = new Promise(r=>$.post(url2, data2, r));
promise1.then(result=>{
  promise2.then(v=>{
    result.nick = v.nick;
    report(result);
  });
});
```

GENERATOR + PROMISE

GENERATOR + PROMISE

```
const profile = function*(end, r){
  const userid = yield new Promise(res=>$.post('member.php', {r}, res));
  let added = yield new Promise(res=>$.post('detail.php', {userid}, res));
  added = added.split(",");
  end({userid, nick:added[0], thumb:added[1]});
};
```

GENERATOR + PROMISE

```
const profile = function*(end, r){
  const userid = yield new Promise(res=>$.post('member.php', {r}, res));
  let added = yield new Promise(res=>$.post('detail.php', {userid}, res));
  added = added.split(",");
  end({userid, nick:added[0], thumb:added[1]});
};
```

```
const executor = (gene, end, ...arg)=>{
  const iter = gene(end, ...arg);
  const next = ({value, done}) =>{
    if(!done) value.then(v=>next(iter.next(v)));
  };
  next(iter.next());
};
```

GENERATOR + PROMISE

```
const profile = function*(end, r){
  const userid = yield new Promise(res=>$.post('member.php', {r}, res));
  let added = yield new Promise(res=>$.post('detail.php', {userid}, res));
  added = added.split(",");
  end({userid, nick:added[0], thumb:added[1]});
};
```

```
const executor = (gene, end, ...arg)=>{
  const iter = gene(end, ...arg);
  const next = ({value, done}) =>{
    if(!done) value.then(v=>next(iter.next(v)));
  };
  next(iter.next());
};
```

```
executor(profile, console.log, 123);
```

ASYNC AWAIT

AWAIT PROMISE

```
const profile = function*(end, r){
  const userid = yield new Promise(res=>$.post('member.php', {r}, res));
  let added = yield new Promise(res=>$.post('detail.php', {userid}, res));
  added = added.split(",");
  end({userid, nick:added[0], thumb:added[1]});
};
```

```
const executor = (gene, end, ...arg)=>{
  const iter = gene(end, ...arg);
  const next = ({value, done}) =>{
    if(!done) value.then(v=>next(iter.next(v)));
  };
  next(iter.next());
};
```

```
executor(profile, console.log, 123);
```

AWAIT PROMISE

```
const profile = function*(end, r){
  const userid = yield new Promise(res=>$.post('member.php', {r}, res));
  let added = yield new Promise(res=>$.post('detail.php', {userid}, res));
  added = added.split(",");
  end({userid, nick:added[0], thumb:added[1]});
};
```

```
const executor = (gene, end, ...arg)=>{
  const iter = gene(end, ...arg);
  const next = ({value, done}) =>{
    if(!done) value.then(v=>next(iter.next(v)));
  };
  next(iter.next());
};
```

```
executor(profile, console.log, 123);
```

AWAIT PROMISE = SYNC

```
const profile = async function(end, r){
  const userid = await new Promise(res=>$.post('member.php', {r}, res));
  let added = await new Promise(res=>$.post('detail.php', {userid}, res));
  added = added.split(",");
  end({userid, nick:added[0], thumb:added[1]});
};
```

```
profile(console.log, 123);
```

AWAIT PROMISE = SYNC

```
const profile = async function(end, r){
  const userid = await new Promise(res=>$.post('member.php', {r}, res));
  let added = await new Promise(res=>$.post('detail.php', {userid}, res));
  added = added.split(",");
  end({userid, nick:added[0], thumb:added[1]});
};
```

```
profile(console.log, 123);
```