<https://www.youtube.com/watch?v=8fenTR3KOJo>

토비의 봄 TV 5회 스프링 리액티브 프로그래밍 (1) - Reactive Streams

<http://www.reactive-streams.org/>

리엑티브 펑셔널 프로그래밍

이벤트가 발생하면 그에 대한 대응하는 프로그래밍 하는것.

Duality (상대성)

Observer Pattern(옵저버 패턴)

Reactive Streams - 표준 - Java9 API

Iterator

pull땡겨오기

|  |
| --- |
| Iterable<Integer> iter =()->  new Iterator(){  int i =0;  public static final int *MAX* =10;  public boolean hasNext() {  return i<*MAX*;  }  public Object next() {  return ++i;  }  }; for(Integer i : iter){  System.*out*.println(i); } |

Observable

push 넣기

|  |
| --- |
| Observable -> Observer  public class Ob {  public static class IntObservable extends Observable implements Runnable{  @Override  public void run() {  for (int i = 0; i < 10; i++) {  setChanged();  notifyObservers(i);  }  }  }  public static void main(String[] args) {  Observer observer = new Observer() {  @Override  public void update(Observable o, Object arg) {  System.*out*.println(Thread.*currentThread*().getName()+" "+arg);  }  };  IntObservable io = new IntObservable();  io.addObserver(observer);  ExecutorService es = Executors.*newSingleThreadExecutor*();  es.execute(io);   System.*out*.println(Thread.*currentThread*().getName()+" EXIT");  es.shutdown();  } }  main EXIT  pool-1-thread-1 0  pool-1-thread-1 1  pool-1-thread-1 2  pool-1-thread-1 3  pool-1-thread-1 4  pool-1-thread-1 5  pool-1-thread-1 6  pool-1-thread-1 7  pool-1-thread-1 8  pool-1-thread-1 9 |

위내용을 보면 알듯 duality 상대성을 보인다. data = method <--> method(data)

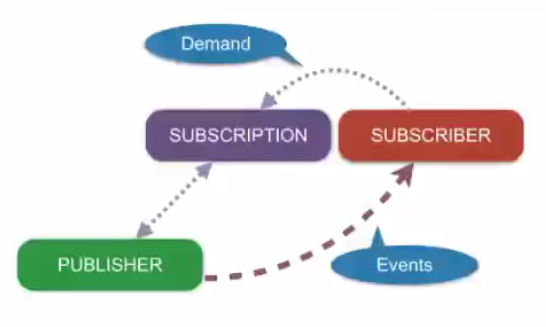
위에도 좋은 Observable Reactive 코딩이지만 문제점이 존재한다.

1. 언제 끝낸건지 알수 없다

2. 예외 발생했을때 어떻게 해야될지.

Publisher <- Observable, Subscriber <-- Observer

onSubscribe, onNext\*, (onError | onComplete)



|  |
| --- |
| package com;  import org.reactivestreams.Publisher; import org.reactivestreams.Subscriber; import org.reactivestreams.Subscription;  import java.util.Arrays; import java.util.Iterator;  public class PubSub {  public static void main(String[] args) {  Iterable<Integer> iter = Arrays.*asList*(1,2,3,4,5,6,7,8,9);  Publisher p = new Publisher() {  @Override  public void subscribe(Subscriber subscriber) {  Iterator it = iter.iterator();  subscriber.onSubscribe(new Subscription() {//Subscription통해 SubScripber에 전달  @Override  public void request(long n) {  try {  while (n-- > 0) {  if (it.hasNext()) {  subscriber.onNext(it.next());  }else{  subscriber.onComplete();  break;  }  }  }catch (Throwable t){   subscriber.onError(t);  }  }   @Override  public void cancel() {   }  });  }  };   Subscriber s = new Subscriber() {  private Subscription subscription;   @Override  public void onSubscribe(Subscription subscription) {  System.*out*.println(Thread.*currentThread*().getName()+" onSubscribe");  this.subscription = subscription;  this.subscription.request(1);    }   @Override  public void onNext(Object o) {  System.*out*.println(Thread.*currentThread*().getName()+" onNext "+o);  this.subscription.request(1);  }   @Override  public void onError(Throwable t) {  System.*out*.println(Thread.*currentThread*().getName()+" onError "+t);  }   @Override  public void onComplete() {  System.*out*.println(Thread.*currentThread*().getName()+" onComplete");  }  };   p.subscribe(s); //Publisher에 <- Subscriber 등록  } } |

<https://www.youtube.com/watch?v=DChIxy9g19o>

토비의 봄 TV 6회 스프링 리액티브 프로그래밍 (2) - Reactive Streams - Operators

Publisher -> [data1]-> Operator -> [data2] -> Subscriber

1차 체인닝 맵핑 처리

|  |
| --- |
| package com;  import lombok.extern.slf4j.Slf4j; import org.reactivestreams.Publisher; import org.reactivestreams.Subscriber; import org.reactivestreams.Subscription;  import java.util.Iterator; import java.util.function.Function; import java.util.stream.Collectors; import java.util.stream.Stream;   @Slf4j public class PubSubChainMap {  public static void main(String[] args) throws InterruptedException {  Iterable<Integer> iter = Stream.*iterate*(1,a->a+1).limit(10).collect(Collectors.*toList*());  Publisher pub = *iterPub*(iter);  Publisher sub = *mapPub*(pub, (Function<Integer,Integer>) s->s\*10);  Subscriber s = *logSub*();  sub.subscribe(s); //Publisher에 <- Subscriber 등록    }   private static Publisher<Integer> mapPub(Publisher pub, Function<Integer, Integer> integerIntegerFunction) {  return new Publisher<Integer>(){  @Override  public void subscribe(Subscriber<? super Integer> sub) {  pub.subscribe(sub);  }  };   }   private static Publisher iterPub(Iterable<Integer> iter) {  return new Publisher() {  @Override  public void subscribe(Subscriber subscriber) {  Iterator it = iter.iterator();  subscriber.onSubscribe(new Subscription() { //Subscription통해 SubScripber에 전달  @Override  public void request(long n) {  try {  int i=0;  while (i++<n) {  if (it.hasNext()) {  subscriber.onNext(it.next());  }else{  subscriber.onComplete();  break;  }  }  }catch (Throwable t){  subscriber.onError(t);  }  }   @Override  public void cancel() {   }  });  }  };  }   private static Subscriber logSub() {  return new Subscriber() {  private Subscription subscription;   @Override  public void onSubscribe(Subscription subscription) {  *log*.debug(Thread.*currentThread*().getName()+" onSubscribe");  this.subscription = subscription;  this.subscription.request(1);   }   @Override  public void onNext(Object o) {  *log*.debug(Thread.*currentThread*().getName()+" onNext "+o);  this.subscription.request(1);  }   @Override  public void onError(Throwable t) {  *log*.debug(Thread.*currentThread*().getName()+" onError "+t);  }   @Override  public void onComplete() {  *log*.debug(Thread.*currentThread*().getName()+" onComplete");  }  };  } } |

3차 Subscriber를 Delegate로 처리

|  |
| --- |
| private static Publisher<Integer> mapPub(Publisher pub, Function<Integer, Integer> integerIntegerFunction) {  return new Publisher<Integer>(){  @Override  public void subscribe(Subscriber<? super Integer> sub) {  pub.subscribe(new DelegateSub(sub) {  @Override  public void onNext(Integer o) {  super.onNext(integerIntegerFunction.apply(o));  }  });  }  }; }  public class DelegateSub implements Subscriber<Integer> {  private final Subscriber sub;   public DelegateSub(Subscriber sub) {  this.sub=sub;  }   @Override  public void onSubscribe(Subscription s) {  sub.onSubscribe(s);  }   @Override  public void onNext(Integer o) {  sub.onNext(o);  }   @Override  public void onError(Throwable t) {  sub.onError(t);  }   @Override  public void onComplete() {  sub.onComplete();  } } |

Reactor 잘만들어진 Reactive 사용하기 편하게 만들어져있다.

<https://github.com/reactor/reactor-core>, <http://projectreactor.io/>

Flux (Publisher)

|  |
| --- |
| compile "io.projectreactor:reactor-core:3.1.1.RELEASE"  testCompile("io.projectreactor:reactor-test:3.1.1.RELEASE")  public class ReactorEx {  public static void main(String[] args) {  Flux.*create*(e->{  e.next(1);  e.next(2);  e.next(3);  e.next(4);  }).log().subscribe(System.*out*::println);  } }  11:04:22.451 [main] INFO reactor.Flux.Create.1 - onSubscribe(FluxCreate.BufferAsyncSink)  11:04:22.456 [main] INFO reactor.Flux.Create.1 - request(unbounded)  11:04:22.462 [main] INFO reactor.Flux.Create.1 - onNext(1)  1  11:04:22.463 [main] INFO reactor.Flux.Create.1 - onNext(2)  2  11:04:22.463 [main] INFO reactor.Flux.Create.1 - onNext(3)  3  11:04:22.463 [main] INFO reactor.Flux.Create.1 - onNext(4)  4 |
| map, reduce  public class ReactorEx {  public static void main(String[] args) {  Flux.<Integer>*create*(e->{  e.next(1);  e.next(2);  e.next(3);  e.next(4);  e.complete();  })  .log()  .map(s->s\*10)  .reduce(0,(a,b)->a+b)  .log()  .subscribe(System.*out*::println);  } }  11:09:23.535 [main] INFO reactor.Flux.Create.1 - onSubscribe(FluxCreate.BufferAsyncSink)  11:09:23.538 [main] INFO reactor.Mono.ReduceSeed.2 - | onSubscribe([Fuseable] MonoReduceSeed.ReduceSeedSubscriber)  11:09:23.539 [main] INFO reactor.Mono.ReduceSeed.2 - | request(unbounded)  11:09:23.539 [main] INFO reactor.Flux.Create.1 - request(unbounded)  11:09:23.545 [main] INFO reactor.Flux.Create.1 - onNext(1)  11:09:23.546 [main] INFO reactor.Flux.Create.1 - onNext(2)  11:09:23.546 [main] INFO reactor.Flux.Create.1 - onNext(3)  11:09:23.546 [main] INFO reactor.Flux.Create.1 - onNext(4)  11:09:23.547 [main] INFO reactor.Flux.Create.1 - onComplete()  11:09:23.547 [main] INFO reactor.Mono.ReduceSeed.2 - | onNext(100)  100  11:09:23.548 [main] INFO reactor.Mono.ReduceSeed.2 - | onComplete()  Process finished with exit code 0 |

2차 체인맵핑

|  |
| --- |
| p @Slf4j public class PubSubChainMap {  public static void main(String[] args) throws InterruptedException {  Iterable<Integer> iter = Stream.*iterate*(1,a->a+1).limit(10).collect(Collectors.*toList*());  Publisher pub = *iterPub*(iter);  Publisher sub = *mapPub*(pub, (Function<Integer,Integer>) s->s\*10);  Publisher sub2 = *mapPub*(sub, (Function<Integer,Integer>) s->-s);  Subscriber s = *logSub*();  sub2.subscribe(s); //Publisher에 <- Subscriber 등록  }   private static Publisher<Integer> mapPub(Publisher pub, Function<Integer, Integer> integerIntegerFunction) {  return new Publisher<Integer>(){  @Override  public void subscribe(Subscriber<? super Integer> sub) {  pub.subscribe(new Subscriber<Integer>() {  @Override  public void onSubscribe(Subscription s) {  sub.onSubscribe(s);  }   @Override  public void onNext(Integer o) {  sub.onNext(integerIntegerFunction.apply(o));  }   @Override  public void onError(Throwable t) {  sub.onError(t);  }   @Override  public void onComplete() {  sub.onComplete();  }  });  }  };  }   private static Publisher iterPub(Iterable<Integer> iter) {  return new Publisher() {  @Override  public void subscribe(Subscriber subscriber) {  Iterator it = iter.iterator();  subscriber.onSubscribe(new Subscription() { //Subscription통해 SubScripber에 전달  @Override  public void request(long n) {  try {  int i=0;  while (i++<n) {  if (it.hasNext()) {  subscriber.onNext(it.next());  }else{  subscriber.onComplete();  break;  }  }  }catch (Throwable t){  subscriber.onError(t);  }  }   @Override  public void cancel() {   }  });  }  };  }   private static Subscriber logSub() {  return new Subscriber() {  private Subscription subscription;   @Override  public void onSubscribe(Subscription subscription) {  *log*.debug(Thread.*currentThread*().getName()+" onSubscribe");  this.subscription = subscription;  this.subscription.request(1);   }   @Override  public void onNext(Object o) {  *log*.debug(Thread.*currentThread*().getName()+" onNext "+o);  this.subscription.request(1);  }   @Override  public void onError(Throwable t) {  *log*.debug(Thread.*currentThread*().getName()+" onError "+t);  }   @Override  public void onComplete() {  *log*.debug(Thread.*currentThread*().getName()+" onComplete");  }  };  } }  main onSubscribe  main onNext -10  main onNext -20  main onNext -30  main onNext -40  main onNext -50  main onNext -60  main onNext -70  main onNext -80  main onNext -90  main onNext -100  main onComplete |

Spring App 만들어보기

|  |
| --- |
| package com.app;  import org.reactivestreams.Publisher; import org.reactivestreams.Subscriber; import org.reactivestreams.Subscription; import org.springframework.boot.SpringApplication; import org.springframework.boot.autoconfigure.EnableAutoConfiguration; import org.springframework.web.bind.annotation.RequestMapping; import org.springframework.web.bind.annotation.RestController;  @RestController @EnableAutoConfiguration public class BootAppController {  @RequestMapping("/hello")  Publisher<String> home(String name) {  return new Publisher<String>(){  @Override  public void subscribe(Subscriber<? super String> s) {  s.onSubscribe(new Subscription() {  @Override  public void request(long n) {  s.onNext("hello "+name);  s.onComplete();  }   @Override  public void cancel() {   }  });  }  }; // return "Hello World!";  }   @RequestMapping("/")  String idx(String name) {  return "Hello World!";  }  public static void main(String[] args) throws Exception {  SpringApplication.*run*(BootAppController.class, args);  } }    group 'com.khh' version '1.0-SNAPSHOT'  apply plugin: 'java'  sourceCompatibility = 1.8  repositories {  mavenCentral()  maven {  url 'https://repo.spring.io/libs-milestone'  } }  dependencies {  compile (group: 'org.reactivestreams', name: 'reactive-streams', version: '1.0.1')  compile (group: 'org.reactivestreams', name: 'reactive-streams-tck', version: '1.0.1')  compile "io.projectreactor:reactor-core:3.1.1.RELEASE"  testCompile("io.projectreactor:reactor-test:3.1.1.RELEASE") // compile("org.springframework.boot:spring-boot-starter-web:1.5.8.RELEASE")  compile('org.springframework.boot:spring-boot-starter-webflux')  compile("org.springframework.boot:spring-boot-starter-web:2.0.0.M6") // compile 'org.springframework:spring-context:5.0.1.RELEASE'  compile (group: 'org.projectlombok', name: 'lombok', version: '1.16.18')  compile group: 'ch.qos.logback', name: 'logback-classic', version: '1.2.3'  compile group: 'ch.qos.logback', name: 'logback-core', version: '1.2.3'  compile group: 'org.slf4j', name: 'slf4j-api', version: '1.7.25'   testCompile group: 'junit', name: 'junit', version: '4.12' } |

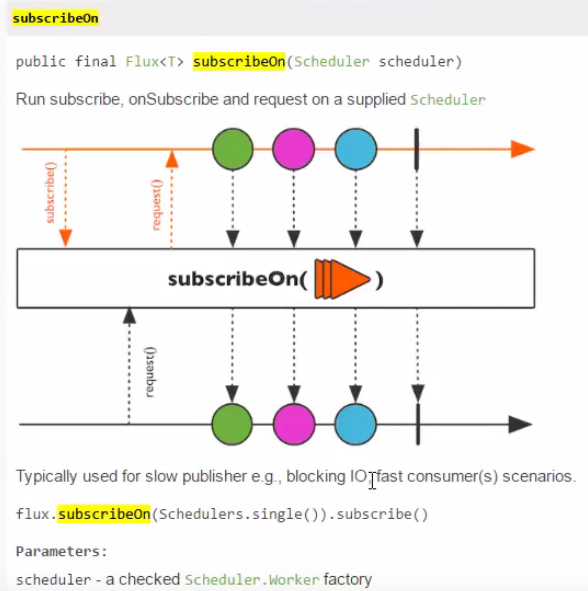
토비의 봄 TV 7회 스프링 리액티브 프로그래밍 (3) - Reactive Streams - Schedulers

<https://www.youtube.com/watch?v=Wlqu1xvZCak>

Scheduler

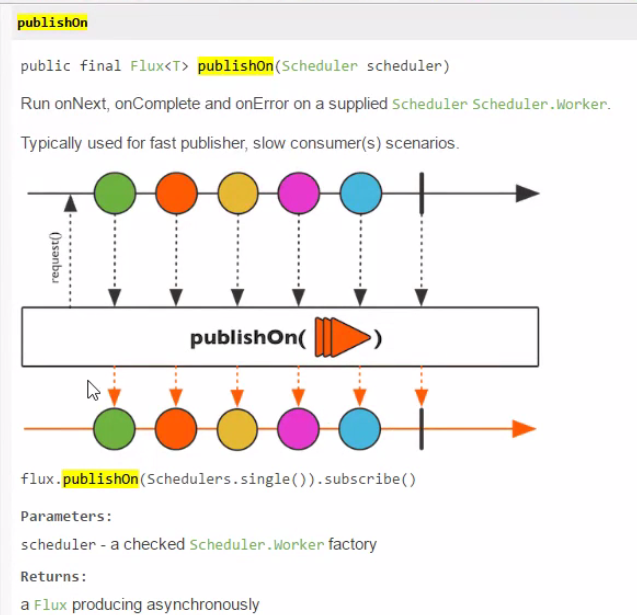
subscribeOn 사용할때

보통 하나의 쓰레드에서 Publisher와 Subscriber를 사용하지 않는다. 하면 의미가 없다 비동기식에 적합하게 나온 패턴이다.



|  |
| --- |
| package com;  import lombok.extern.slf4j.Slf4j; import org.reactivestreams.Publisher; import org.reactivestreams.Subscriber; import org.reactivestreams.Subscription;  import java.util.concurrent.ExecutorService; import java.util.concurrent.Executors;   @Slf4j public class SchedulerEx {  public static void main(String[] args) {   Publisher<Integer> pub = sub->{  sub.onSubscribe(new Subscription() {  @Override  public void request(long n) {  sub.onNext(1);  sub.onNext(2);  sub.onNext(3);  sub.onNext(4);  sub.onNext(5);  sub.onComplete();  }  @Override  public void cancel() {  }  });  };    Publisher subOnPub = sub->{  ExecutorService es = Executors.*newSingleThreadExecutor*(); //쓰레드풀 하나의 쓰레드만 처리가능한 풀  es.execute(()->pub.subscribe(sub));  };    subOnPub.subscribe(new Subscriber<Integer>() {  @Override  public void onSubscribe(Subscription sub) {  *log*.debug("onSubscribe {}",sub);  sub.request(Long.*MAX\_VALUE*);  }  @Override  public void onNext(Integer integer) {  *log*.debug("onNext {}",integer);  }  @Override  public void onError(Throwable t) {  *log*.debug("onError {}",t);  }  @Override  public void onComplete() {  *log*.debug("onComplete");  }  });   *log*.debug("exit");  } } |

publishOn



|  |
| --- |
| package com;  import lombok.extern.slf4j.Slf4j; import org.reactivestreams.Publisher; import org.reactivestreams.Subscriber; import org.reactivestreams.Subscription;  import java.util.concurrent.ExecutorService; import java.util.concurrent.Executors;   @Slf4j public class SchedulerEx {  public static void main(String[] args) {   Publisher<Integer> pub = sub->{  sub.onSubscribe(new Subscription() {  @Override  public void request(long n) {  *log*.debug("request {}",n);  sub.onNext(1);  sub.onNext(2);  sub.onNext(3);  sub.onNext(4);  sub.onNext(5);  sub.onComplete();  }  @Override  public void cancel() {  }  });  };   // Publisher subOnPub = sub->{ // ExecutorService es = Executors.newSingleThreadExecutor(); //쓰레드풀 하나의 쓰레드만 처리가능한 풀 // es.execute(()->pub.subscribe(sub)); // };  Publisher pubOnSub = sub->{  pub.subscribe(new Subscriber<Integer>() {  ExecutorService es = Executors.*newSingleThreadExecutor*(); //쓰레드풀 하나의 쓰레드만 처리가능한 풀  @Override  public void onSubscribe(Subscription s) {  sub.onSubscribe(s);  }   @Override  public void onNext(Integer integer) {  es.execute(()->sub.onNext(integer));  }   @Override  public void onError(Throwable t) {  es.execute(()->sub.onError(t));  }   @Override  public void onComplete() {  es.execute(()->sub.onComplete());  }  });  };    pubOnSub.subscribe(new Subscriber<Integer>() {  @Override  public void onSubscribe(Subscription sub) {  *log*.debug("onSubscribe {}",sub);  sub.request(Long.*MAX\_VALUE*);  }  @Override  public void onNext(Integer integer) {  *log*.debug("onNext {}",integer);  }  @Override  public void onError(Throwable t) {  *log*.debug("onError {}",t);  }  @Override  public void onComplete() {  *log*.debug("onComplete");  }  });   *log*.debug("exit");  } }  14:02:27.988 [main] DEBUG com.SchedulerEx - onSubscribe com.SchedulerEx$1@71dac704  14:02:27.994 [main] DEBUG com.SchedulerEx - request 9223372036854775807  14:02:27.996 [main] DEBUG com.SchedulerEx - exit  14:02:27.998 [pool-1-thread-1] DEBUG com.SchedulerEx - onNext 1  14:02:27.998 [pool-1-thread-1] DEBUG com.SchedulerEx - onNext 2  14:02:27.999 [pool-1-thread-1] DEBUG com.SchedulerEx - onNext 3  14:02:27.999 [pool-1-thread-1] DEBUG com.SchedulerEx - onNext 4  14:02:27.999 [pool-1-thread-1] DEBUG com.SchedulerEx - onNext 5  14:02:27.999 [pool-1-thread-1] DEBUG com.SchedulerEx - onComplete |
| Publisher subOnPub = sub->{  ExecutorService es = Executors.*newSingleThreadExecutor*(); //쓰레드풀 하나의 쓰레드만 처리가능한 풀  es.execute(()->pub.subscribe(sub)); };  14:12:48.009 [main] DEBUG com.SchedulerEx - exit  14:12:48.029 [pool-2-thread-1] DEBUG com.SchedulerEx - onSubscribe com.SchedulerEx$1@347236dd  14:12:48.038 [pool-2-thread-1] DEBUG com.SchedulerEx - request 9223372036854775807  14:12:48.044 [pool-1-thread-1] DEBUG com.SchedulerEx - onNext 1  14:12:48.044 [pool-1-thread-1] DEBUG com.SchedulerEx - onNext 2  14:12:48.044 [pool-1-thread-1] DEBUG com.SchedulerEx - onNext 3  14:12:48.044 [pool-1-thread-1] DEBUG com.SchedulerEx - onNext 4  14:12:48.044 [pool-1-thread-1] DEBUG com.SchedulerEx - onNext 5  14:12:48.045 [pool-1-thread-1] DEBUG com.SchedulerEx - onComplete |

Flux사용하기

|  |
| --- |
| Flux.*range*(1,10)  .publishOn(Schedulers.*newSingle*("pub"))  .log()  .subscribeOn(Schedulers.*newSingle*("sub"))  .subscribe(System.*out*::println);  14:21:34.192 [sub-1] INFO reactor.Flux.PublishOn.1 - | onSubscribe([Fuseable] FluxPublishOn.PublishOnSubscriber)  14:21:34.197 [sub-1] INFO reactor.Flux.PublishOn.1 - | request(unbounded)  14:21:34.200 [pub-2] INFO reactor.Flux.PublishOn.1 - | onNext(1)  1  14:21:34.200 [pub-2] INFO reactor.Flux.PublishOn.1 - | onNext(2)  2  14:21:34.200 [pub-2] INFO reactor.Flux.PublishOn.1 - | onNext(3)  3  14:21:34.201 [pub-2] INFO reactor.Flux.PublishOn.1 - | onNext(4)  4  14:21:34.201 [pub-2] INFO reactor.Flux.PublishOn.1 - | onNext(5)  5  14:21:34.201 [pub-2] INFO reactor.Flux.PublishOn.1 - | onNext(6)  6  14:21:34.201 [pub-2] INFO reactor.Flux.PublishOn.1 - | onNext(7)  7  14:21:34.201 [pub-2] INFO reactor.Flux.PublishOn.1 - | onNext(8)  8  14:21:34.201 [pub-2] INFO reactor.Flux.PublishOn.1 - | onNext(9)  9  14:21:34.202 [pub-2] INFO reactor.Flux.PublishOn.1 - | onNext(10)  10  14:21:34.203 [pub-2] INFO reactor.Flux.PublishOn.1 - | onComplete() |

Thread는 userThread와 DeamonThread 가있는데

Flux.interval 은 DeamonThread이다.. 그러기때문에 메인Thread가 끝나면 그냥 끝나버린다.

|  |
| --- |
| Flux.*interval*(Duration.*ofMillis*(500))  .subscribe(s->*log*.debug("onNext:{}",s)); TimeUnit.*SECONDS*.sleep(10);  14:30:40.709 [parallel-1] DEBUG com.FluxScEx - onNext:0  ...  14:30:50.198 [parallel-1] DEBUG com.FluxScEx - onNext:19 |
| take 사용  Flux.*interval*(Duration.*ofMillis*(500))  .take(5)  .subscribe(s->*log*.debug("onNext:{}",s)); TimeUnit.*SECONDS*.sleep(10); *log*.debug("exit");  14:32:13.844 [main] DEBUG reactor.util.Loggers$LoggerFactory - Using Slf4j logging framework  14:32:14.388 [parallel-1] DEBUG com.FluxScEx - onNext:0  14:32:14.886 [parallel-1] DEBUG com.FluxScEx - onNext:1  14:32:15.400 [parallel-1] DEBUG com.FluxScEx - onNext:2  14:32:15.886 [parallel-1] DEBUG com.FluxScEx - onNext:3  14:32:16.419 [parallel-1] DEBUG com.FluxScEx - onNext:4  14:32:23.886 [main] DEBUG com.FluxScEx - exit |

Flux Interval take 구현해보기 Subscription cancel사용하기.. unSubscrip

|  |
| --- |
| package com;  import lombok.extern.slf4j.Slf4j; import org.reactivestreams.Publisher; import org.reactivestreams.Subscriber; import org.reactivestreams.Subscription;  import java.util.concurrent.ExecutorService; import java.util.concurrent.Executors; import java.util.concurrent.ScheduledExecutorService; import java.util.concurrent.TimeUnit;  @Slf4j public class IntervalEx {  public static void main(String[] args) {   Publisher<Integer> pub = sub->{  sub.onSubscribe(new Subscription() {  int no = 0;  boolean cancelled=false;  @Override  public void request(long n) {  ScheduledExecutorService exec = Executors.*newSingleThreadScheduledExecutor*();  exec.scheduleAtFixedRate(()->{  if(cancelled){  exec.shutdown();  return;  }  sub.onNext(no++);  }, 0, 300, TimeUnit.*MICROSECONDS*);  }  @Override  public void cancel() {  cancelled=true;  }  });  };    Publisher taskPub = sub->{  pub.subscribe(new Subscriber<Integer>() {  int count = 0;  public Subscription subsc;   @Override  public void onSubscribe(Subscription s) {  this.subsc = s;  sub.onSubscribe(s);   }   @Override  public void onNext(Integer integer) {  sub.onNext(integer);  if(++count > 5){  subsc.cancel();  }  }   @Override  public void onError(Throwable t) {   }   @Override  public void onComplete() {   }  });  };    Publisher pubOnSub = sub->{  taskPub.subscribe(new Subscriber<Integer>() {  ExecutorService es = Executors.*newSingleThreadExecutor*(); //쓰레드풀 하나의 쓰레드만 처리가능한 풀  @Override  public void onSubscribe(Subscription s) {  sub.onSubscribe(s);  }   @Override  public void onNext(Integer integer) {  es.execute(()->sub.onNext(integer));  }   @Override  public void onError(Throwable t) {  es.execute(()->sub.onError(t));  es.shutdown();  }   @Override  public void onComplete() {  es.execute(()->sub.onComplete());  es.shutdown();  }  });  };    pubOnSub.subscribe(new Subscriber<Integer>() {  @Override  public void onSubscribe(Subscription sub) {  *log*.debug("onSubscribe {}",sub);  sub.request(Long.*MAX\_VALUE*);  }  @Override  public void onNext(Integer integer) {  *log*.debug("onNext {}",integer);  }  @Override  public void onError(Throwable t) {  *log*.debug("onError {}",t);  }  @Override  public void onComplete() {  *log*.debug("onComplete");  }  });   *log*.debug("exit");  } }  14:43:46.404 [main] DEBUG com.IntervalEx - onSubscribe com.IntervalEx$1@123772c4  14:43:46.416 [main] DEBUG com.IntervalEx - exit  14:43:46.419 [pool-1-thread-1] DEBUG com.IntervalEx - onNext 0  14:43:46.420 [pool-1-thread-1] DEBUG com.IntervalEx - onNext 1  14:43:46.420 [pool-1-thread-1] DEBUG com.IntervalEx - onNext 2  14:43:46.420 [pool-1-thread-1] DEBUG com.IntervalEx - onNext 3  14:43:46.420 [pool-1-thread-1] DEBUG com.IntervalEx - onNext 4  14:43:46.420 [pool-1-thread-1] DEBUG com.IntervalEx - onNext 5 |

토비의 봄 TV 8회 스프링 리액티브 프로그래밍 (4) 자바와 스프링의 비동기 기술

<https://www.youtube.com/watch?v=aSTuQiPB4Ns>

Future

다른 쓰레드에 가지고있는 결과

|  |
| --- |
| Future f = es.submit(()->{  Thread.*sleep*(2000);  *log*.debug("Async hello");  return "hello"; }); *log*.debug("thread is done? {}",f.isDone()); *log*.debug("future get {}",f.get()); //blocking *log*.debug("EXIT"); *log*.debug("thread is done? {}",f.isDone());  15:10:38.969 [main] DEBUG com.FutureEx - thread is done? false  15:10:40.966 [pool-1-thread-1] DEBUG com.FutureEx - Async hello  15:10:40.966 [main] DEBUG com.FutureEx - future get hello  15:10:40.966 [main] DEBUG com.FutureEx - EXIT  15:10:40.966 [main] DEBUG com.FutureEx - thread is done? true |

|  |
| --- |
| FutureTask futureTask = new FutureTask(()->{  Thread.*sleep*(2000);  *log*.debug("Async hello");  return "hello"; }){  @Override  protected void done() {  try {  *log*.debug("doen {}",get());  } catch (InterruptedException e) {  e.printStackTrace();  } catch (ExecutionException e) {  e.printStackTrace();  }  super.done();  } }; es.execute(futureTask); *log*.debug("thread is done? {}",futureTask.isDone()); *log*.debug("future get {}",futureTask.get()); //blocking *log*.debug("EXIT"); *log*.debug("thread is done? {}",futureTask.isDone());  es.shutdown();  15:45:49.670 [main] DEBUG com.FutureEx - thread is done? false  15:45:51.680 [pool-1-thread-1] DEBUG com.FutureEx - Async hello  15:45:51.682 [pool-1-thread-1] DEBUG com.FutureEx - doen hello  15:45:51.682 [main] DEBUG com.FutureEx - future get hello  15:45:51.682 [main] DEBUG com.FutureEx - EXIT  15:45:51.682 [main] DEBUG com.FutureEx - thread is done? true |

CallBack을 이용하기

|  |
| --- |
| package com;  import lombok.extern.slf4j.Slf4j;  import java.util.Objects; import java.util.concurrent.\*;  @Slf4j public class FutureEx {   interface SuccessCallBack{  void onSuccess(String result);  }  interface ExceptionCallBack{  void onError(Throwable t);  }   public static class CallbackFutureTask extends FutureTask<String>{  private final SuccessCallBack sc;  private final ExceptionCallBack ec;   public CallbackFutureTask(Callable<String> callable, SuccessCallBack sc, ExceptionCallBack ec){  super(callable);  this.sc = Objects.*requireNonNull*(sc);  this.ec = Objects.*requireNonNull*(ec);  }   @Override  protected void done() {  try {  sc.onSuccess(get());  } catch (InterruptedException e) {  Thread.*currentThread*().interrupt();  } catch (ExecutionException e) {  ec.onError(e.getCause());  }  }  }   public static void main(String[] args) throws InterruptedException, ExecutionException {  ExecutorService es = Executors.*newCachedThreadPool*();   CallbackFutureTask futureTask = new CallbackFutureTask(()->{  Thread.*sleep*(2000);  if(1==1) throw new RuntimeException("Async Error!!"); //강제로 에러  *log*.debug("Async hello");  return "hello";  }, res->{  *log*.debug("result :: {}",res);  }, thow->{  *log*.debug("thow :: {}",thow);  });  es.execute(futureTask);  *log*.debug("thread is done? {}",futureTask.isDone());  *log*.debug("EXIT");  *log*.debug("thread is done? {}",futureTask.isDone());   es.shutdown();  } } |

위 콜백방식은 성격이 다른 기술적인 코드와 비지니스 코드가 혼재되어있다. 별로 좋지 않다.

spring boot에서 구현해보기

blocking

|  |
| --- |
| @Slf4j @SpringBootApplication public class BApplication {   @Component  public static class MyService{  public String hello() throws InterruptedException{  *log*.info("hello()");  Thread.*sleep*(1000);  return "hello";  }  }   public static void main(String[] args) {  try(ConfigurableApplicationContext c = SpringApplication.*run*(BApplication.class, args)){   }  }   @Autowired MyService myService;  //모든 빈들이 준비가 되면 실행된다.  @Bean  ApplicationRunner run(){  return args -> {  *log*.info("run()");  *log*.info(myService.hello());  *log*.info("exit");  };  } }  main] com.app.BApplication : run()  main] com.app.BApplication : hello()  main] com.app.BApplication : hello  main] com.app.BApplication : exit |

Async 비동기

|  |
| --- |
| @Slf4j @SpringBootApplication @EnableAsync public class BApplication {   @Component  public static class MyService{  @Async  public Future<String> hello() throws InterruptedException{  *log*.info("hello()");  Thread.*sleep*(1000);  return new AsyncResult<>("hello");  }  }   public static void main(String[] args) {  try(ConfigurableApplicationContext c = SpringApplication.*run*(BApplication.class, args)){   }  }   @Autowired MyService myService;  //모든 빈들이 준비가 되면 실행된다.  @Bean  ApplicationRunner run(){  return args -> {  *log*.info("run()");  Future<String> f = myService.hello();  *log*.info("exit {}",f.isDone());  *log*.info("result {}",f.get()); //blocking  };  } }  [ main] com.app.BApplication : Started BApplication in 5.649 seconds (JVM running for 7.909)  [ main] com.app.BApplication : run()  [ main] .s.a.AnnotationAsyncExecutionInterceptor : No task executor bean found for async processing: no bean of type TaskExecutor and no bean named 'taskExecutor' either  [ main] com.app.BApplication : exit false  [cTaskExecutor-1] com.app.BApplication : hello()  [ 1초후 main] com.app.BApplication : result hello |

Blocking을 피할수 있도록 ListenableFuture 를 사용하면된다.

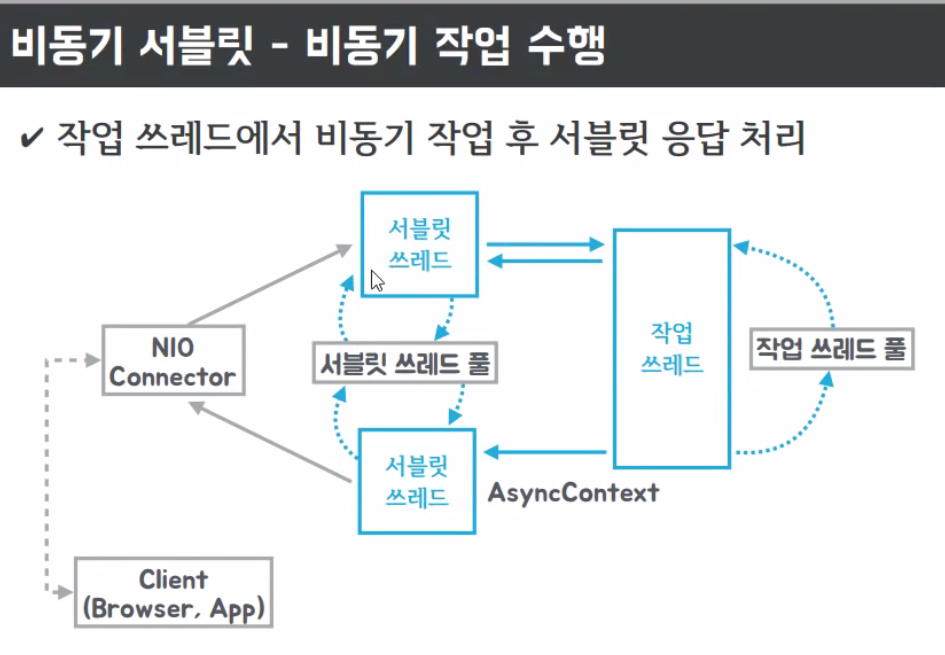
|  |
| --- |
| @Slf4j @SpringBootApplication @EnableAsync public class BApplication {   @Component  public static class MyService{  @Async  public Future<String> hello() throws InterruptedException{  *log*.info("hello()");  Thread.*sleep*(1000);  return new AsyncResult<>("hello");  }  @Async  public ListenableFuture<String> helloCallback() throws InterruptedException{  *log*.info("hello()");  Thread.*sleep*(1000);  return new AsyncResult<>("hello2222");  }  }   public static void main(String[] args) {  try(ConfigurableApplicationContext c = SpringApplication.*run*(BApplication.class, args)){   }  }   @Autowired MyService myService;  //모든 빈들이 준비가 되면 실행된다.  @Bean  ApplicationRunner run(){  return args -> {  *log*.info("run()");  ListenableFuture<String> lf = myService.helloCallback();  lf.addCallback(s->{  *log*.info("Success:{}",s);  },e->{  *log*.info("fail:{}",e);  });  *log*.info("exit exit");  };  } }  2017-11-07 16:21:03.101 INFO 6632 --- [ main] com.app.BApplication : run()  2017-11-07 16:21:03.103 INFO 6632 --- [ main] .s.a.AnnotationAsyncExecutionInterceptor : No task executor bean found for async processing: no bean of type TaskExecutor and no bean named 'taskExecutor' either  2017-11-07 16:21:03.107 INFO 6632 --- [ main] com.app.BApplication : exit exit  2017-11-07 16:21:03.107 INFO 6632 --- [ main] ConfigServletWebServerApplicationContext : Closing org.springframework.boot.web.servlet.context.AnnotationConfigServletWebServerApplicationContext@2667f029: startup date [Tue Nov 07 16:21:00 KST 2017]; root of context hierarchy  2017-11-07 16:21:03.108 INFO 6632 --- [cTaskExecutor-1] com.app.BApplication : hello()  2017-11-07 16:21:03.109 INFO 6632 --- [ main] o.s.j.e.a.AnnotationMBeanExporter : Unregistering JMX-exposed beans on shutdown  2017-11-07 16:21:04.108 INFO 6632 --- [cTaskExecutor-1] com.app.BApplication : Success:hello2222  위에서 볼수 있듯. 나중에 콜백함수로 리턴된걸로 보여준다. |

CompletableFuture도있다..

ThreadPool변경

|  |
| --- |
| package com.app;  import lombok.extern.slf4j.Slf4j; import org.springframework.beans.factory.annotation.Autowired; import org.springframework.boot.ApplicationRunner; import org.springframework.boot.SpringApplication; import org.springframework.boot.autoconfigure.SpringBootApplication; import org.springframework.context.ConfigurableApplicationContext; import org.springframework.context.annotation.Bean; import org.springframework.scheduling.annotation.Async; import org.springframework.scheduling.annotation.AsyncResult; import org.springframework.scheduling.annotation.EnableAsync; import org.springframework.scheduling.concurrent.ThreadPoolTaskExecutor; import org.springframework.stereotype.Component; import org.springframework.util.concurrent.ListenableFuture;  import java.util.concurrent.Future;  @Slf4j @SpringBootApplication @EnableAsync public class BApplication {   @Component  public static class MyService{  @Async  public Future<String> hello() throws InterruptedException{  *log*.info("hello()");  Thread.*sleep*(1000);  return new AsyncResult<>("hello");  }  @Async //쓰레드풀 만들어논거 없을땐 SimpleThreadpool 사용하지만 구현해놓은거 있으면 그걸쓴다 여기서는 아래 ThreadPoolTaskExecutor  public ListenableFuture<String> helloCallback() throws InterruptedException{  *log*.info("hello()");  Thread.*sleep*(1000);  return new AsyncResult<>("hello2222");  }  }   @Bean  ThreadPoolTaskExecutor tp(){  ThreadPoolTaskExecutor te = new ThreadPoolTaskExecutor();  te.setCorePoolSize(10); // 처음 풀 사이즈  te.setMaxPoolSize(100); // 큐에 다차면 100개 더 넣는다. <-큐를 꽉차면 더 늘려줄께  te.setQueueCapacity(200); // 10개 꽉차면 200개까지 쌓는다  te.setThreadNamePrefix("myThread");  te.initialize();  return te;  }   public static void main(String[] args) {  try(ConfigurableApplicationContext c = SpringApplication.*run*(BApplication.class, args)){   }  }   @Autowired MyService myService;  //모든 빈들이 준비가 되면 실행된다.  @Bean  ApplicationRunner run(){  return args -> {  *log*.info("run()"); // Future<String> f = myService.hello(); // log.info("exit {}",f.isDone()); // log.info("result {}",f.get()); //blocking   ListenableFuture<String> lf = myService.helloCallback();  lf.addCallback(s->{  *log*.info("Success:{}",s);  },e->{  *log*.info("fail:{}",e);  });  *log*.info("exit exit");  };  } }  2017-11-07 16:56:33.348 INFO 16792 --- [ main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8080 (http)  2017-11-07 16:56:33.355 INFO 16792 --- [ main] com.app.BApplication : Started BApplication in 5.897 seconds (JVM running for 7.83)  2017-11-07 16:56:33.356 INFO 16792 --- [ main] com.app.BApplication : run()  2017-11-07 16:56:33.367 INFO 16792 --- [ main] com.app.BApplication : exit exit  2017-11-07 16:56:33.368 INFO 16792 --- [ main] ConfigServletWebServerApplicationContext : Closing org.springframework.boot.web.servlet.context.AnnotationConfigServletWebServerApplicationContext@96def03: startup date [Tue Nov 07 16:56:28 KST 2017]; root of context hierarchy  2017-11-07 16:56:33.371 INFO 16792 --- [ myThread1] com.app.BApplication : hello()  2017-11-07 16:56:33.372 INFO 16792 --- [ main] o.s.j.e.a.AnnotationMBeanExporter : Unregistering JMX-exposed beans on shutdown  2017-11-07 16:56:33.373 INFO 16792 --- [ main] o.s.s.concurrent.ThreadPoolTaskExecutor : Shutting down ExecutorService 'tp'  2017-11-07 16:56:33.381 INFO 16792 --- [ myThread1] com.app.BApplication : fail:{} |





|  |
| --- |
| @Slf4j @SpringBootApplication @EnableAsync public class AnsyncApplication {   @RestController  public static class MyController{  @GetMapping("/async")  public String async() throws InterruptedException {  Thread.*sleep*(2000);  return "hello";  }  @GetMapping("/callable")  public Callable<String> callable() throws InterruptedException {  *log*.info("callable");  return ()->{  *log*.info("async");  Thread.*sleep*(2000);  return "hello";  };  }  }    public static void main(String[] args) {  SpringApplication.*run*(AnsyncApplication.class, args);  }  }  2017-11-07 17:19:16.485 INFO 16016 --- [nio-8080-exec-9] com.app.AnsyncApplication : callable  2017-11-07 17:19:16.485 INFO 16016 --- [ MvcAsync5] com.app.AnsyncApplication : async |

성능을 비교해보자

기본적으로 Tomcat의 기본 ThreadPoolSize는 200개인걸로 알고 있다. 우선 20개로 셋팅후 다시 확인해보자

|  |
| --- |
| application.properties  server.tomcat.max-threads=20 |

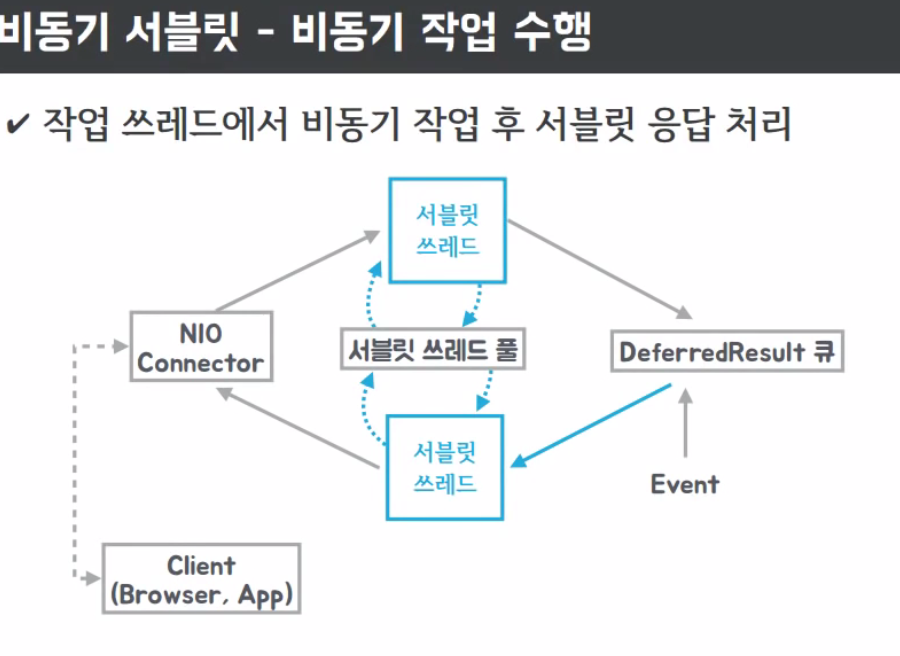
|  |
| --- |
| @Slf4j @SpringBootApplication @EnableAsync public class AnsyncApplication {   @RestController  public static class MyController{  @GetMapping("/async")  public String async() throws InterruptedException {  Thread.*sleep*(2000);  return "hello";  }     @GetMapping("/callable")  public Callable<String> callable() throws InterruptedException {  *log*.info("callable");  return ()->{  *log*.info("async");  Thread.*sleep*(2000);  return "hello";  };  }  }    public static void main(String[] args) {  SpringApplication.*run*(AnsyncApplication.class, args);  }  } |
| @Slf4j public class LoadTest {  static AtomicInteger *counter* = new AtomicInteger(0);   public static void main(String[] args) throws InterruptedException {  ExecutorService es = Executors.*newFixedThreadPool*(100);   RestTemplate rt = new RestTemplate();  String url = "http://localhost:8080/async"; // String url = "http://localhost:8080/callable";   StopWatch main = new StopWatch();  main.start();    for (int i = 0; i < 100; i++) {  es.execute(()->{  int idx = *counter*.addAndGet(1);  *log*.info("Thread {}",idx);   StopWatch sw = new StopWatch();  sw.start();  rt.getForObject(url, String.class);   sw.stop();  *log*.info("Elapsed : {} {}",idx,sw.getTotalTimeSeconds());  });  }   es.shutdown();  es.awaitTermination(100, TimeUnit.*SECONDS*);  main.stop();  *log*.info("Total :{}",main.getTotalTimeSeconds());  } }  com.LoadTest - Total :10.171  ..20개 까지.생성됨 |

동시에 100개를 날리지만 서버측에서 20개밖에 Thread를 생성을 못하니 2\* (100/20=5) = 10초가 된다.

Callable로 하면

|  |
| --- |
| String url = "http://localhost:8080/callable";  17:43:09.112 [main] INFO com.LoadTest - Total :3.13    하지만 max-threads=1 로 하고 해보면  17:46:01.634 [main] INFO com.LoadTest - Total :3.065  하나밖에 생성이 안된다 그런데 왜!!? 속도가 빠를까?  결국은 뒤쪽에 WorkerThreadPool을 또 만들어서 사용한다 Callable |

DeferredResult



지연된 결과를 한꺼번에 써줄수 있는것..나중에 써줄수 있는 기술

|  |
| --- |
| @Slf4j @SpringBootApplication @EnableAsync public class AnsyncApplication {   @RestController  public static class MyController{  Queue<DeferredResult<String>> results = new ConcurrentLinkedDeque<>();   @GetMapping("/dr")  public DeferredResult<String> dr() throws InterruptedException {  *log*.info("dr");  DeferredResult dr = new DeferredResult<>(600000L);  results.add(dr);  return dr;  }   @GetMapping("/dr/count")  public String drcount() throws InterruptedException {  return String.*valueOf*(results.size());  }  @GetMapping("/dr/event")  public String drevent(String msg) throws InterruptedException {  for (DeferredResult<String> dr : results){  dr.setResult("Hello "+msg);  results.remove(dr);  }  return "OK";  }     public static void main(String[] args) {  SpringApplication.*run*(AnsyncApplication.class, args);  }  }  localhost:8080/dr 로 접속하면 계속 지연되어 있는 상태 이다.  하지만 Thread는 반환되어 Thread 소비는 되지 않는다.  그상태에서 /dr/event를 하게되면 그때  /dr요청한 쪽에 반영이된다. 간단한 체팅 브로드 케스팅하기에 좋은 디퍼드 리절트 이다. |

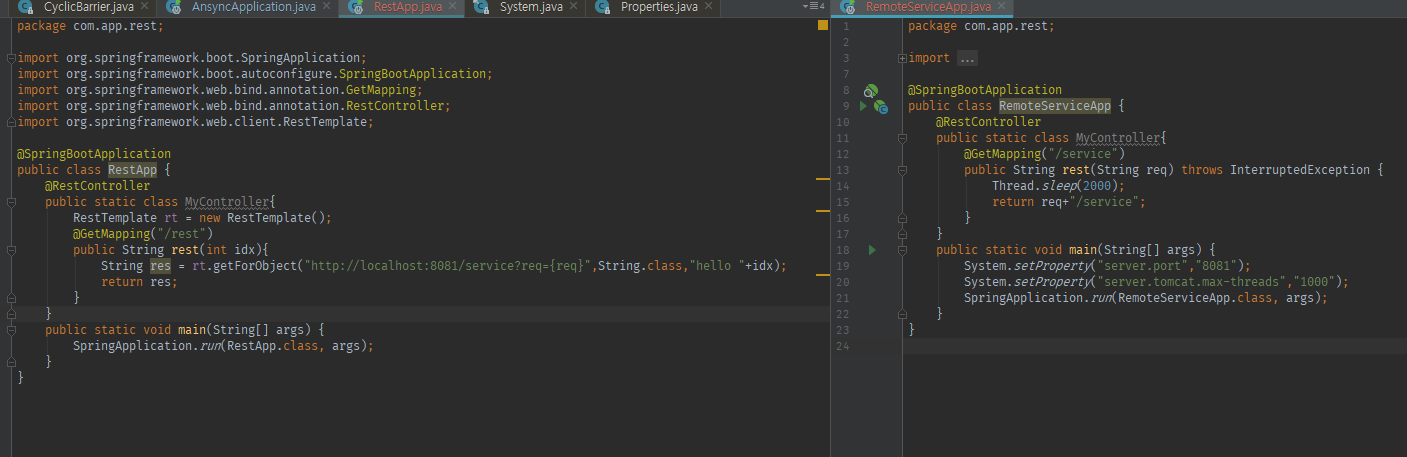
emitter

한번 요청에 여러번 쪼개서 응답을 해주는 기술 (스트리밍 하기 좋다)

|  |
| --- |
| @GetMapping("/emitter") public ResponseBodyEmitter emitter() throws InterruptedException {  ResponseBodyEmitter emitter = new ResponseBodyEmitter();  Executors.*newSingleThreadExecutor*().submit(()->{  try {  for (int i = 0; i < 50; i++) {  emitter.send("<p>a"+i+"+sd</p>");  Thread.*sleep*(500);  }  }catch (Exception e){  *log*.error("ee",e);  }  });   return emitter; }  ... |

토비의 봄 TV 9회 스프링 리액티브 프로그래밍 (5) 비동기 RestTemplate과 비동기 MVC/Serlvet

<https://www.youtube.com/watch?v=ExUfZkh7Puk>



RestTeamplate는 기본으로 blocking이기때문에 느릴수밖에 없다. 그래서 AsyncRestTeamplate로 바꿔보자

|  |
| --- |
| @SpringBootApplication public class RestApp {  @RestController  public static class MyController{ // RestTemplate rt = new RestTemplate();  AsyncRestTemplate rt = new AsyncRestTemplate();  @GetMapping("/rest")  public ListenableFuture<ResponseEntity<String>> rest(int idx){  ListenableFuture<ResponseEntity<String>> res = rt.getForEntity("http://localhost:8081/service?req={req}",String.class,"hello "+idx);  return res;  }  }  public static void main(String[] args) {  SpringApplication.*run*(RestApp.class, args);  } }  하지만 AsyncRestTemplate도 뒤쪽에서 Thread를 새로운 생성을 한다.!!! |

AsyncRestTemplate에 Netty를 사용하면 된다. 하지만 Deprecated됐다 ㅋ



비동기 리턴값 가공후 response보내기 DeferredResult사용하기.

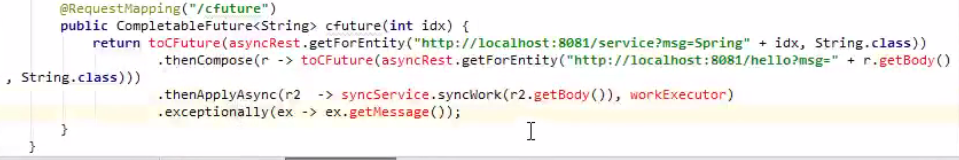
|  |
| --- |
| @SpringBootApplication public class RestApp {  @RestController  public static class MyController{  AsyncRestTemplate rt = new AsyncRestTemplate(new Netty4ClientHttpRequestFactory());  @GetMapping("/rest")  public DeferredResult<String> rest(int idx){  DeferredResult<String> dr = new DeferredResult<>();  ListenableFuture<ResponseEntity<String>> res = rt.getForEntity("http://localhost:8081/service?req={req}",String.class,"hello "+idx);  res.addCallback(s->{  dr.setResult(s.getBody());  }, e->{  dr.setErrorResult(e.getMessage());  });  return dr;  }  }  public static void main(String[] args) {  SpringApplication.*run*(RestApp.class, args);  } } |

NoneBlocking

|  |
| --- |
| @SpringBootApplication @EnableAsync public class RestApp {  @RestController  public static class MyController{  @Autowired MyService myService; // RestTemplate rt = new RestTemplate();  AsyncRestTemplate rt = new AsyncRestTemplate(new Netty4ClientHttpRequestFactory());  @GetMapping("/rest")  public DeferredResult<String> rest(int idx){ // public String rest(int idx){ // String res = rt.getForObject("http://localhost:8081/service?req={req}",String.class,"hello "+idx);   DeferredResult<String> dr = new DeferredResult<>();  ListenableFuture<ResponseEntity<String>> res = rt.getForEntity("http://localhost:8081/service?req={req}",String.class,"hello "+idx);  res.addCallback(s->{ // dr.setResult(s.getBody());  ListenableFuture<String> f2 = myService.work(s.getBody());  f2.addCallback(s2->{  dr.setResult(s2);  },e->{  dr.setErrorResult(e.getMessage());  });   }, e->{  dr.setErrorResult(e.getMessage());  });  return dr;  }  }   @Service  public static class MyService{  @Async  public ListenableFuture<String> work(String req){  return new AsyncResult<>(req+"/asyncwork");  }  }   @Bean  public ThreadPoolTaskExecutor myThreadPool(){  ThreadPoolTaskExecutor te = new ThreadPoolTaskExecutor();  te.setCorePoolSize(1);  te.setMaxPoolSize(1);  te.initialize();  return te;  }   public static void main(String[] args) {  SpringApplication.*run*(RestApp.class, args);  } } |

위코드는 콜백헬이 발생된다.. 콜백콜백콜백 아오~~ callback hell~~

아래처럼 줄여보자



토비의 봄 TV 10회 스프링 리액티브 프로그래밍 (6) AsyncRestTemplate의 콜백 헬과 중복 작업 문제

<https://www.youtube.com/watch?v=Tb43EyWTSlQ>

|  |
| --- |
| @SpringBootApplication @EnableAsync public class RestApp {  @RestController  public static class MyController{  @Autowired MyService myService; // RestTemplate rt = new RestTemplate();  AsyncRestTemplate rt = new AsyncRestTemplate(new Netty4ClientHttpRequestFactory());  @GetMapping("/rest")  public DeferredResult<String> rest(int idx){ // public String rest(int idx){ // String res = rt.getForObject("http://localhost:8081/service?req={req}",String.class,"hello "+idx);   DeferredResult<String> dr = new DeferredResult<>(); // ListenableFuture<ResponseEntity<String>> res = rt.getForEntity("http://localhost:8081/service?req={req}",String.class,"hello "+idx); // res.addCallback(s->{ // ListenableFuture<String> f2 = myService.work(s.getBody()); // f2.addCallback(s2->{ // dr.setResult(s2); // },e->{ // dr.setErrorResult(e.getMessage()); // }); // // }, e->{ // dr.setErrorResult(e.getMessage()); // });   ListenableFuture<ResponseEntity<String>> res1 = rt.getForEntity("http://localhost:8081/service1?req={req}",String.class,"hello "+idx);  ListenableFuture<ResponseEntity<String>> res2 = rt.getForEntity("http://localhost:8081/service2?req={req}",String.class,"hello "+idx);  Completion  .*from*(res1)  .andApply(s->res2) // .andApply(s->myService.work(s))  .andError(e->dr.setErrorResult(e.getMessage()))  .andAccept(s->dr.setResult(s.getBody()));  return dr;  }  }     public static class AcceptCompletion extends Completion{  Consumer<ResponseEntity<String>> con;  public AcceptCompletion(Consumer<ResponseEntity<String>> con) {  this.con = con;  }   @Override  void run(ResponseEntity<String> value) {  con.accept(value);  }  }  public static class ErrorCompletion extends Completion{  Consumer<Throwable> econ;  public ErrorCompletion(Consumer<Throwable> econ) {  this.econ = econ;  }   @Override  void run(ResponseEntity<String> value) {  if(next!=null) next.run(value);  }   @Override  void error(Throwable e) {  econ.accept(e);  }  }  public static class ApplyCompletion extends Completion{  Function<ResponseEntity<String>, ListenableFuture<ResponseEntity<String>>> fn;  public ApplyCompletion(Function<ResponseEntity<String>, ListenableFuture<ResponseEntity<String>>> fn) {  this.fn = fn;  }   @Override  void run(ResponseEntity<String> value) {  ListenableFuture<ResponseEntity<String>> lf = fn.apply(value);  lf.addCallback(s->complate(s), e->error(e));  }  }   public static class Completion{  Completion next;  public Completion() {}      public static Completion from(ListenableFuture<ResponseEntity<String>> lf){  Completion c = new Completion();  lf.addCallback(s->{  c.complate(s);  },e->{  c.error(e);  });   return c;   }  public void andAccept(Consumer<ResponseEntity<String>> con){  Completion c = new AcceptCompletion(con);  this.next = c;   }  public Completion andError(Consumer<Throwable> econ){  Completion c = new ErrorCompletion(econ);  this.next = c;  return c;  }  public Completion andApply(Function<ResponseEntity<String>, ListenableFuture<ResponseEntity<String>>> fn){  Completion c = new ApplyCompletion(fn);  this.next = c;  return c;  }  void error(Throwable e) {  if(next!=null) next.error(e);  }   void complate(ResponseEntity<String> s) {  if(next!=null)next.run(s);  }   void run(ResponseEntity<String> value) {   }    }   @Service  public static class MyService{  @Async  public ListenableFuture<String> work(String req){  return new AsyncResult<>(req+"/asyncwork");  }  }   @Bean  public ThreadPoolTaskExecutor myThreadPool(){  ThreadPoolTaskExecutor te = new ThreadPoolTaskExecutor();  te.setCorePoolSize(1);  te.setMaxPoolSize(1);  te.initialize();  return te;  }   public static void main(String[] args) {  SpringApplication.*run*(RestApp.class, args);  } } |

토비의 봄 TV 11회 스프링 리액티브 프로그래밍 (7) CompletableFuture

<https://www.youtube.com/watch?v=PzxV-bmLSFY>

CompletableFuture sinc(java8)

비동기 작업을 담고 있는 객체 ,

|  |
| --- |
| CompletableFuture<Integer> f = CompletableFuture.*completedFuture*(1); *log*.info("r : {}",f.get());  15:16:47.200 [main] INFO com.app.future.completable.CFuture - r : 1 |
| CompletableFuture<Integer> f = new CompletableFuture<>(); f.complete(5); *log*.info("r : {}",f.get());  15:18:54.963 [main] INFO com.app.future.completable.CFuture - r : 5 |

성공은 했지만 예외가 발생됐다.

|  |
| --- |
| CompletableFuture<Integer> f = new CompletableFuture<>(); f.completeExceptionally(new RuntimeException()); *log*.info("r : {}",f.get());  Exception in thread "main" java.util.concurrent.ExecutionException: java.lang.RuntimeException  at java.util.concurrent.CompletableFuture.reportGet(CompletableFuture.java:357)  at java.util.concurrent.CompletableFuture.get(CompletableFuture.java:1895)  at com.app.future.completable.CFuture.main(CFuture.java:13)  Caused by: java.lang.RuntimeException  at com.app.future.completable.CFuture.main(CFuture.java:12) |

Async

|  |
| --- |
| CompletableFuture.*runAsync*(()->{  *log*.info("RunAsync"); });  *log*.info("EXIT");  ForkJoinPool.*commonPool*().shutdown(); ForkJoinPool.*commonPool*().awaitTermination(10, TimeUnit.*SECONDS*);  15:22:39.594 [ForkJoinPool.commonPool-worker-1] INFO com.app.future.completable.CFuture - RunAsync  15:22:39.594 [main] INFO com.app.future.completable.CFuture - EXIT |
| 자바7부터는 쓰레드풀이 기본으로 ForkJoinPool이다. |

CompletionStage <- CompletableFuture

runAsync

thenRun 순차적으로 실행. (같은 쓰레드) 리턴값이 없다. 리턴값이 없는 체이닝.

|  |
| --- |
| CompletableFuture.*runAsync*(()->{  *log*.info("RunAsync"); }).thenRun(()->{  *log*.info("thenRun 1"); }).thenRun(()->{  *log*.info("thenRun 2"); });  *log*.info("EXIT");  15:24:35.775 [main] INFO com.app.future.completable.CFuture - EXIT  15:24:35.787 [ForkJoinPool.commonPool-worker-1] INFO com.app.future.completable.CFuture - RunAsync  15:24:35.787 [ForkJoinPool.commonPool-worker-1] INFO com.app.future.completable.CFuture - thenRun 1  15:24:35.787 [ForkJoinPool.commonPool-worker-1] INFO com.app.future.completable.CFuture - thenRun 2  앞에서 오류가 안난다면 뒤쪽으로 흐른다. |

supplyAsync 리턴값이 있는 체이닝.

thenApply, thenAccept

|  |
| --- |
| CompletableFuture.*supplyAsync*(()->{  *log*.info("supplyAsync");  return 1; }).thenApply(s->{  *log*.info("thenApply {}",s);  return s+1; }).thenAccept(s->{  *log*.info("thenAccept {}",s); });  *log*.info("EXIT");  15:30:26.676 [main] INFO com.app.future.completable.CFuture - EXIT  15:30:26.676 [ForkJoinPool.commonPool-worker-1] INFO com.app.future.completable.CFuture - supplyAsync  15:30:26.682 [ForkJoinPool.commonPool-worker-1] INFO com.app.future.completable.CFuture - thenApply 1  15:30:26.685 [ForkJoinPool.commonPool-worker-1] INFO com.app.future.completable.CFuture - thenAccept 2 |
|  |

위에서 값이 넘어오는것이 아니라 CompletebleFuture 넘어오게 하고싶을때.

|  |
| --- |
| CompletableFuture.*supplyAsync*(()->{  *log*.info("supplyAsync");  return 1; }).thenCompose(s->{  *log*.info("thenCompose {}",s);  return CompletableFuture.*completedFuture*(s+1); }).thenApply(s->{  *log*.info("thenApply {}",s);  return s\*3; }).thenAccept(s->{  *log*.info("thenAccept {}",s); });  15:33:36.445 [main] INFO com.app.future.completable.CFuture - EXIT  15:33:36.445 [ForkJoinPool.commonPool-worker-1] INFO com.app.future.completable.CFuture - supplyAsync  15:33:36.450 [ForkJoinPool.commonPool-worker-1] INFO com.app.future.completable.CFuture - thenCompose 1  15:33:36.453 [ForkJoinPool.commonPool-worker-1] INFO com.app.future.completable.CFuture - thenApply 2  15:33:36.453 [ForkJoinPool.commonPool-worker-1] INFO com.app.future.completable.CFuture - thenAccept 6 |

exceptionnally 예외처리

|  |
| --- |
| CompletableFuture.*supplyAsync*(()->{  *log*.info("supplyAsync");  return 1; }).thenCompose(s->{  *log*.info("thenCompose {}",s);  **if(1==1)throw new RuntimeException();**  return CompletableFuture.*completedFuture*(s+1); }).thenApply(s->{  *log*.info("thenApply {}",s);  return s\*3; }).**exceptionally(s-> -10)** .thenAccept(s->{  *log*.info("thenAccept {}",s); });  15:35:14.648 [ForkJoinPool.commonPool-worker-1] INFO com.app.future.completable.CFuture - supplyAsync  15:35:14.655 [ForkJoinPool.commonPool-worker-1] INFO com.app.future.completable.CFuture - thenCompose 1  15:35:14.658 [ForkJoinPool.commonPool-worker-1] INFO com.app.future.completable.CFuture - thenAccept -10 |

thenApplyAsync ,thenAcceptAsync, thenComposeAsync 다른 쓰레드에서 처리하기.

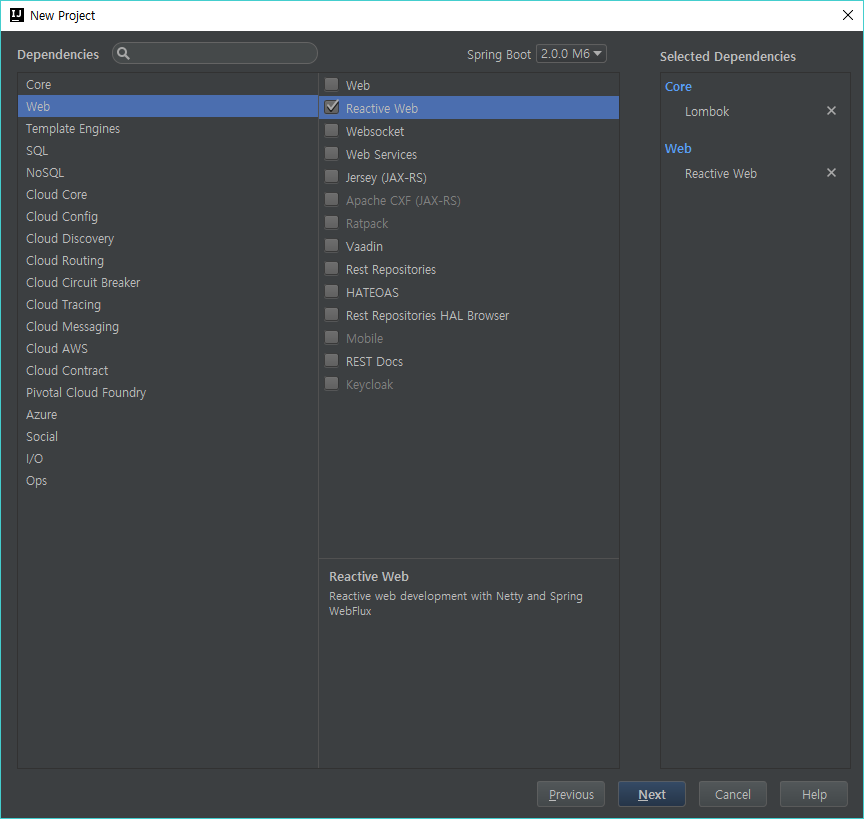
|  |
| --- |
| ExecutorService es = Executors.*newFixedThreadPool*(10);  CompletableFuture.*supplyAsync*(()->{  *log*.info("supplyAsync");  return 1; },es) .thenCompose(s->{  *log*.info("thenCompose {}",s);  return CompletableFuture.*completedFuture*(s+1); }) .**thenApplyAsync**(s->{  *log*.info("thenApply {}",s);  return s\*3; },es) .exceptionally(s-> -10) .**thenAcceptAsync**(s->{  *log*.info("thenAccept {}",s); },**es**);  *log*.info("EXIT");  15:38:30.891 [main] INFO com.app.future.completable.CFuture - EXIT  15:38:30.890 [pool-1-thread-1] INFO com.app.future.completable.CFuture - supplyAsync  15:38:30.896 [**pool-1-thread-1**] INFO com.app.future.completable.CFuture - thenCompose 1  15:38:30.900 [**pool-1-thread-2**] INFO com.app.future.completable.CFuture - thenApply 2  15:38:30.901 [**pool-1-thread-3**] INFO com.app.future.completable.CFuture - thenAccept 6 |

이전 레거시 코드를 CompletableFuture로 바꿔보자

|  |
| --- |
| @SpringBootApplication @EnableAsync public class ComOletableRestApp {  @RestController  public static class MyController{  @Autowired MyService myService;  AsyncRestTemplate rt = new AsyncRestTemplate(new Netty4ClientHttpRequestFactory());  @GetMapping("/rest")  public DeferredResult<String> rest(int idx){   DeferredResult<String> dr = new DeferredResult<>();   ListenableFuture<ResponseEntity<String>> res1 = rt.getForEntity("http://localhost:8081/service1?req={req}",String.class,"hello "+idx);  ListenableFuture<ResponseEntity<String>> res2 = rt.getForEntity("http://localhost:8081/service2?req={req}",String.class,"hello "+idx);   *toCF*(res1).thenCompose(s->{  return *toCF*(res2);  }).thenCompose(s->{  return *toCF*(myService.work(s.getBody()));  }).thenAccept(s->{  dr.setResult(s);  }).exceptionally(e->{  dr.setErrorResult(e.getMessage());  return (Void)null;  });   return dr;  }  }    static <T> CompletableFuture<T> toCF(ListenableFuture<T> lf){  CompletableFuture cf = new CompletableFuture();  lf.addCallback(s->{  cf.complete(s);  },e->{  cf.completeExceptionally(e);  });  return cf;  }   @Service  public static class MyService{  @Async  public ListenableFuture<String> work(String req){  return new AsyncResult<>(req+"/asyncwork");  }  }   public static void main(String[] args) {  SpringApplication.*run*(ComOletableRestApp.class, args);  } } |

토비의 봄 TV 12회 스프링 리액티브 프로그래밍 (8) WebFlux

<https://www.youtube.com/watch?v=ScH7NZU_zvk>



Spring쪽에서 WEB과 Reactive Web은 동시에 사용할수 없다

만약에 동시에 체크하였다면 Web이 우선순위를 가진다.

스프링 부트 2.X 하고 web-reactive ,webflux 하면된다.

|  |
| --- |
| group 'com.khh' version '1.0-SNAPSHOT'  apply plugin: 'java' apply plugin: 'org.springframework.boot' apply plugin: 'io.spring.dependency-management' sourceCompatibility = 1.8  buildscript {  ext {  springBootVersion = '2.0.0.M6'  }  repositories {  mavenCentral()  maven { url "https://repo.spring.io/snapshot" }  maven { url "https://repo.spring.io/milestone" }  }  dependencies {  classpath("org.springframework.boot:spring-boot-gradle-plugin:${springBootVersion}")  } }  repositories {  mavenCentral()  maven { url "https://repo.spring.io/snapshot" }  maven { url "https://repo.spring.io/milestone" } }  dependencies {  compile (group: 'org.reactivestreams', name: 'reactive-streams', version: '1.0.1')  compile (group: 'org.reactivestreams', name: 'reactive-streams-tck', version: '1.0.1')  compile "io.projectreactor:reactor-core:3.1.1.RELEASE"  testCompile("io.projectreactor:reactor-test:3.1.1.RELEASE")  compile('org.springframework.boot:spring-boot-starter-webflux')  compileOnly('org.projectlombok:lombok')   compile 'io.netty:netty-all:4.1.5.Final' // compile 'org.springframework:spring-context:5.0.1.RELEASE'  compile group: 'ch.qos.logback', name: 'logback-classic', version: '1.2.3'  compile group: 'ch.qos.logback', name: 'logback-core', version: '1.2.3'  compile group: 'org.slf4j', name: 'slf4j-api', version: '1.7.25'   testCompile group: 'junit', name: 'junit', version: '4.12'  testCompile('org.springframework.boot:spring-boot-starter-test')  testCompile('io.projectreactor:reactor-test') } |

리턴값은 Mode, Flux로한다.

Mono는 그냥 컨테이너 라고 생각하면 이해하기 쉽다.

|  |
| --- |
| @SpringBootApplication @RestController public class WebFluxApp {  @GetMapping("/rest")  public Mono<String> index(){  return Mono.*just*("Hello"); //그냥 그값 자체.. Mono는 그냥 컨테이너 라고 생각하면 이해하기 쉽다.  }  public static void main(String[] args) {  SpringApplication.*run*(WebFluxApp.class,args);  } } |

WebClient 은 옛날에 RestTemplate와 비슷하다고 생각하면된다.

|  |
| --- |
| @SpringBootApplication @RestController public class WebFluxApp {   WebClient client = WebClient.*create*();   @GetMapping("/rest")  public Mono<String> index(int idx){  Mono<ClientResponse> res = client.get().uri("http://localhost:8081/service?req={req}",idx).exchange(); // Mono<Mono<String>> body = res.map(clientResponse -> clientResponse.bodyToMono(String.class));  Mono<String> body = res.flatMap(clientResponse -> clientResponse.bodyToMono(String.class));  return body;     // return Mono.just("Hello"); //그냥 그값 자체.. Mono는 그냥 컨테이너 라고 생각하면 이해하기 쉽다.  }  public static void main(String[] args) {  SpringApplication.*run*(WebFluxApp.class,args);  } } |
| 여러번 호출  Mono<String> body = res  .flatMap(clientResponse -> clientResponse.bodyToMono(String.class))  .flatMap(res1->client.get().uri("http://localhost:8081/service2?req={req}",idx).exchange())  .flatMap(clientResponse -> clientResponse.bodyToMono(String.class)); |

Service에서 블락킹 또는 오래걸리는 작업을 할때에 Async로 처리하자

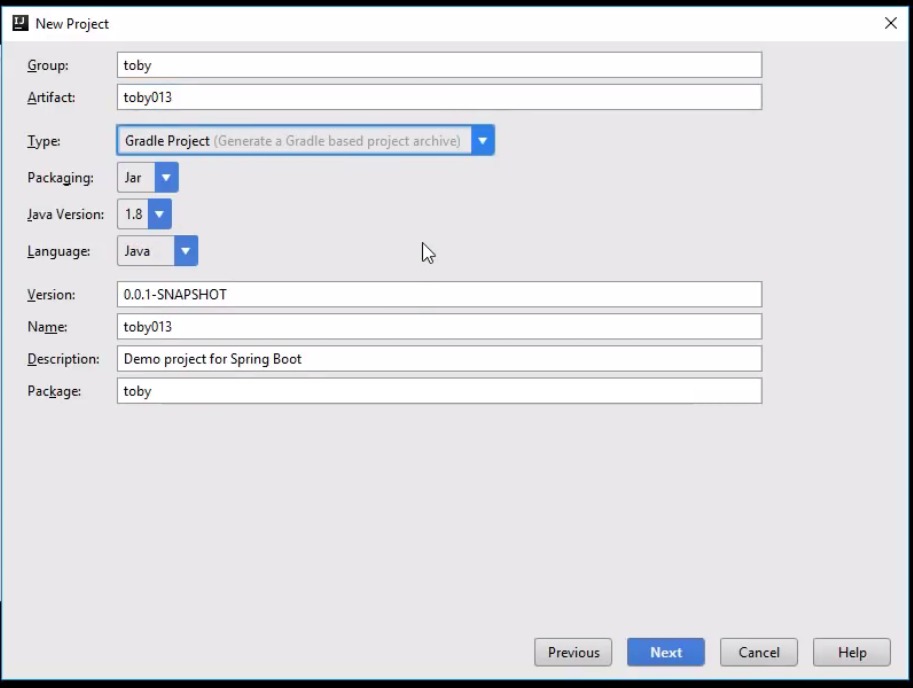
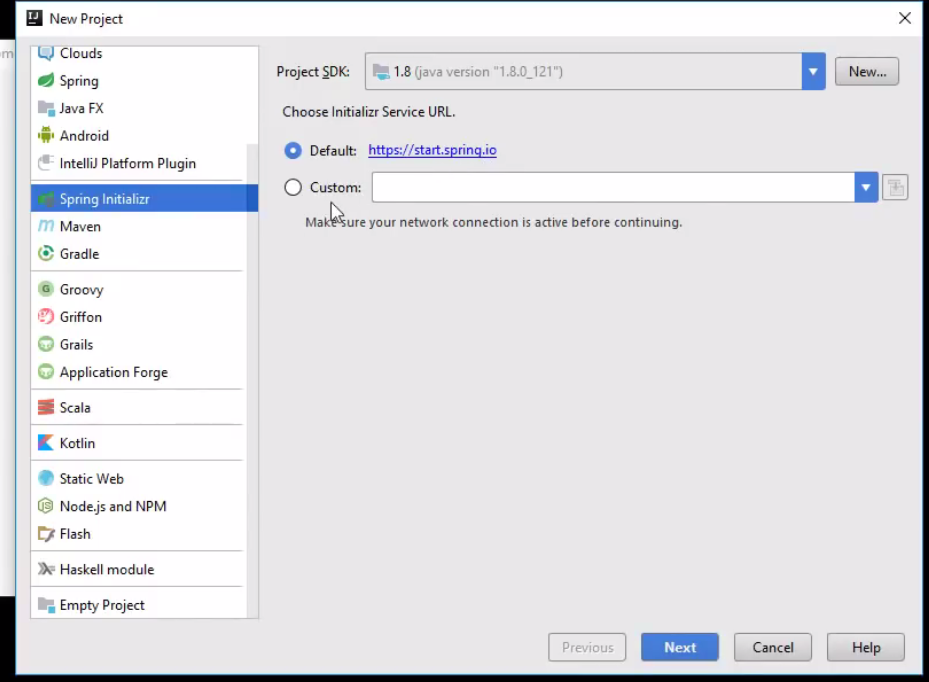
|  |
| --- |
| @SpringBootApplication @RestController  @EnableAsync public class WebFluxApp {   WebClient client = WebClient.*create*();  @Autowired MyService myService;   @GetMapping("/rest")  public Mono<String> index(int idx){  Mono<ClientResponse> res = client.get().uri("http://localhost:8081/service?req={req}",idx).exchange(); // Mono<Mono<String>> body = res.map(clientResponse -> clientResponse.bodyToMono(String.class));  Mono<String> body = res  .flatMap(clientResponse -> clientResponse.bodyToMono(String.class))  .flatMap(res1->client.get().uri("http://localhost:8081/service2?req={req}",idx).exchange())  .flatMap(clientResponse -> clientResponse.bodyToMono(String.class))  **.flatMap(res2->Mono.*fromCompletionStage*(myService.work(res2)));**  return body;  // return Mono.just("Hello"); //그냥 그값 자체.. Mono는 그냥 컨테이너 라고 생각하면 이해하기 쉽다.  }   **@Service  public static class MyService {  @Async  public CompletableFuture<String> work(String req){  return CompletableFuture.*completedFuture*(req+"/work");  }  }**  public static void main(String[] args) {  SpringApplication.*run*(WebFluxApp.class,args);  } } |
|  |

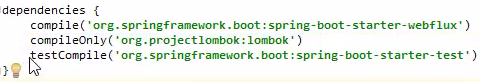
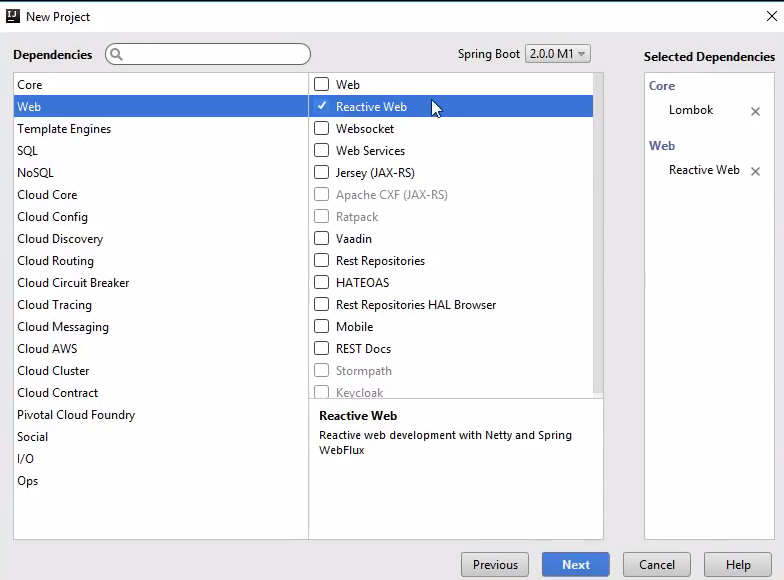
로그 남기기

|  |
| --- |
| Mono<String> body = res  .flatMap(clientResponse -> clientResponse.bodyToMono(String.class))  .doOnNext(it -> *log*.info(it.toString()))  .flatMap(res1->client.get().uri("http://localhost:8081/service2?req={req}",idx).exchange())  .doOnNext(it -> *log*.info(it.toString()))  .flatMap(clientResponse -> clientResponse.bodyToMono(String.class))  .doOnNext(it -> *log*.info(it.toString()))  .flatMap(res2->Mono.*fromCompletionStage*(myService.work(res2))); |

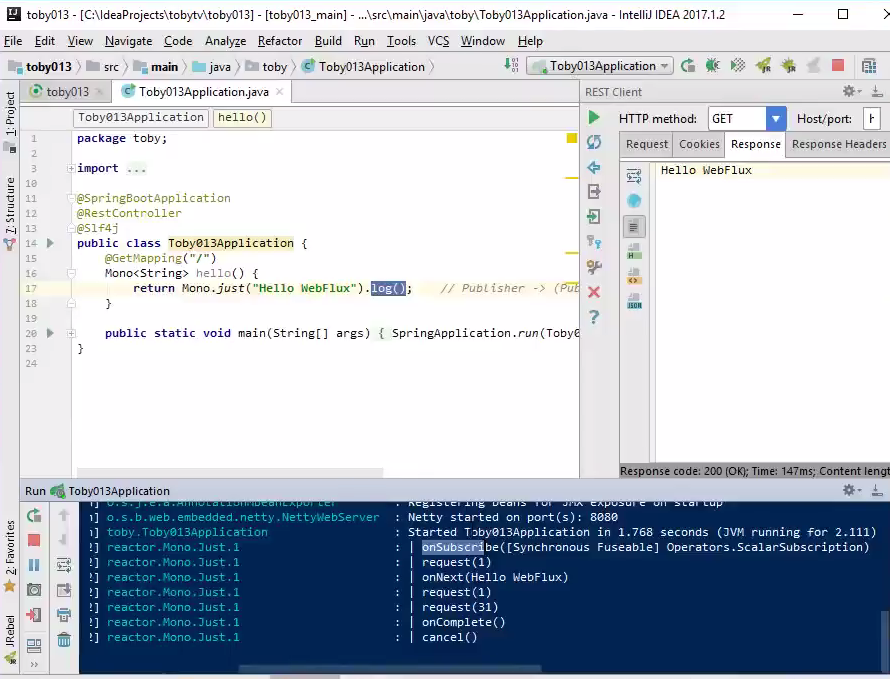
<https://www.youtube.com/watch?v=LK6NRV8tZBM>

토비의 봄 TV 13회 스프링 리액티브 프로그래밍 (9) Mono의 동작방식과 block()





spring5에서 기본으로 webflux에서 tomcat을 기본으로 사용하지 않는다?



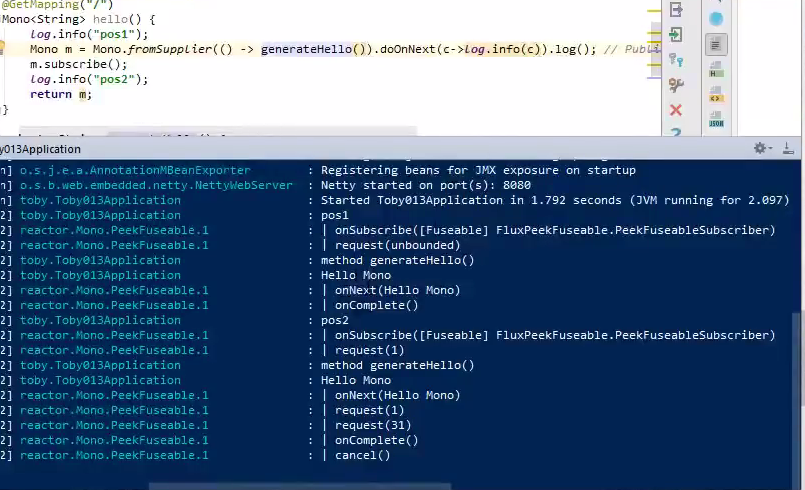
|  |
| --- |
| @GetMapping("/") Mono<String> hello(){  *log*.info("position1");  Mono<String> mono = Mono.*just*("hello").log();  *log*.info("position2");  return mono; } |
| tp-nio-2] com.app.webflux.WebFluxApp : position1  tp-nio-2] com.app.webflux.WebFluxApp : position2  tp-nio-2] reactor.Mono.Just.1 : | onSubscribe([Synchronous Fuseable] Operators.ScalarSubscription)  tp-nio-2] reactor.Mono.Just.1 : | request(1)  tp-nio-2] reactor.Mono.Just.1 : | onNext(hello)  tp-nio-2] reactor.Mono.Just.1 : | request(1)  tp-nio-2] reactor.Mono.Just.1 : | request(31)  tp-nio-2] reactor.Mono.Just.1 : | onComplete()  tp-nio-2] reactor.Mono.Just.1 : | cancel()  비동기식으로 처리된다. |

|  |
| --- |
| @GetMapping("/") Mono<String> hello(){  *log*.info("position1");  String g = generateHello();  Mono<String> mono = Mono.*just*(g).doOnNext(c->*log*.info(c)).log();  *log*.info("position2");  return mono; }  private String generateHello() {  *log*.info("generateHello");  return "just"; } |
| -2] com.app.webflux.WebFluxApp : position1  -2] com.app.webflux.WebFluxApp : generateHello  -2] com.app.webflux.WebFluxApp : position2  -2] reactor.Mono.PeekFuseable.1 : | onSubscribe([Fuseable] FluxPeekFuseable.PeekFuseableSubscriber)  -2] reactor.Mono.PeekFuseable.1 : | request(1)  -2] com.app.webflux.WebFluxApp : just  -2] reactor.Mono.PeekFuseable.1 : | onNext(just)  -2] reactor.Mono.PeekFuseable.1 : | request(1)  -2] reactor.Mono.PeekFuseable.1 : | request(31)  -2] reactor.Mono.PeekFuseable.1 : | onComplete()  -2] reactor.Mono.PeekFuseable.1 : | cancel()  컨트롤러 안쪽에서 just는 이미 정의된거기때문에 순차적으로 발생된다  보면 알듯. just쪽에 있는것은 비동기적으로 처리되지 않는다. |

비동기식으로 실행하여 처리하기. **Mono.fromSupplier**

|  |
| --- |
| @GetMapping("/")  Mono<String> hello(){  *log*.info("position1");  Mono<String> mono = Mono.*fromSupplier*(()->generateHello()).doOnNext(c->*log*.info(c)).log();  *log*.info("position2");  return mono;  } |
| : position1  : position2  : | onSubscribe([Fuseable] FluxPeekFuseable.PeekFuseableSubscriber)  : | request(1)  : generateHello  : just  : | onNext(just)  : | request(1)  : | request(31)  : | onComplete()  : | cancel() |

subscribe 강제 발생



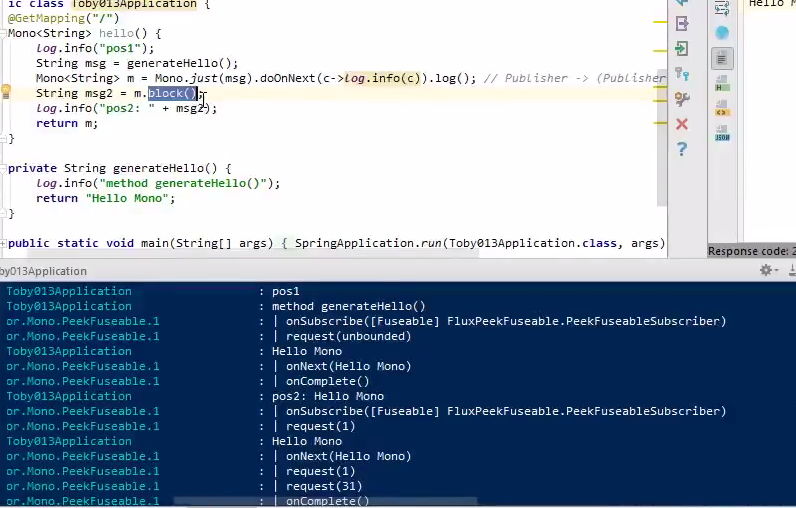
cold타입

정해진 데이터가 있는경우

hot타입

실시간 업데이트가 되는것

block으로 데이터 가져오기.

subscribe와 똑같다 내부에서 Call한다.

<https://www.youtube.com/watch?v=bc4wTgA_2Xk>

토비의 봄 TV 14회 스프링 리액티브 프로그래밍 (10) Flux의 특징과 활용방법

flux 복수개의 리턴.

|  |
| --- |
| @GetMapping("/flux") public Flux<User> user(int idx){  return Flux.*just*(  new User(1,"1val"),  new User(2,"2val"),  new User(3,"3val")  ); }  [{"id":1,"value":"1val"},{"id":2,"value":"2val"},{"id":3,"value":"3val"}] |
| 위처럼 Mono로 처리가능하다  @GetMapping("/mono") public Mono<List<User>> mono(int idx){  List<User> list = new ArrayList<>();  list.add(new User(1,"1val"));  list.add(new User(2,"2val"));  list.add(new User(3,"3val"));  Mono<List<User>> mono = Mono.*just*(list);  return mono; } |

청크방식으로 스트리밍 보낼때 flux를 쓰면 편하다.

|  |
| --- |
| @GetMapping(value = "/flux\_stream",produces = MediaType.*TEXT\_EVENT\_STREAM\_VALUE*) public Flux<User> stream(int idx){  List<User> list = new ArrayList<>();  list.add(new User(1,"1val"));  list.add(new User(2,"2val"));  list.add(new User(3,"3val"));  return Flux.*fromIterable*(list); }  data:{"id":1,"value":"1val"}  data:{"id":2,"value":"2val"}  data:{"id":3,"value":"3val"} |

Stream으로 Flux사용

|  |
| --- |
| @GetMapping(value = "/flux\_stream1",produces = MediaType.*TEXT\_EVENT\_STREAM\_VALUE*)  public Flux<User> stream1(int idx){ // List<User> list = new ArrayList<>(); // list.add(new User(1,"1val")); // list.add(new User(2,"2val")); // list.add(new User(3,"3val")); // Stream<User> userStream = list.stream();   Stream<User> userStream = Stream.*generate*(()->new User(1,"value"));  return Flux.*fromStream*(userStream).take(10);  }  //1초 딜레이.  return Flux.*fromStream*(userStream).delayElements(Duration.*ofSeconds*(1)).take(10);    data:{"id":1,"value":"value"}  data:{"id":1,"value":"value"}  data:{"id":1,"value":"value"}  data:{"id":1,"value":"value"}  data:{"id":1,"value":"value"}  data:{"id":1,"value":"value"}  data:{"id":1,"value":"value"}  data:{"id":1,"value":"value"}  data:{"id":1,"value":"value"}  data:{"id":1,"value":"value"} |

Flux generator사용

|  |
| --- |
| return Flux.<User>*generate*(sink->sink.next(new User(2,"value"))).delayElements(Duration.*ofSeconds*(1)).take(10); |

|  |
| --- |
| return Flux.<User,Integer>*generate*(()->1,(id,sink)->{  sink.next(new User(id,"value"+id));  return id+1; }).delayElements(Duration.*ofSeconds*(1)).take(10);  data:{"id":1,"value":"value1"}  data:{"id":2,"value":"value2"}  data:{"id":3,"value":"value3"}  data:{"id":4,"value":"value4"}  data:{"id":5,"value":"value5"}  data:{"id":6,"value":"value6"}  data:{"id":7,"value":"value7"}  data:{"id":8,"value":"value8"}  data:{"id":9,"value":"value9"}  data:{"id":10,"value":"value10"} |

Flux.interval, zip

|  |
| --- |
| Flux<User> es = Flux.<User,Integer>*generate*(()->1,(id,sink)->{  sink.next(new User(id,"value"+id));  return id+1; }).delayElements(Duration.*ofSeconds*(1));  Flux<Long> interval = Flux.*interval*(Duration.*ofSeconds*(1)); return Flux.*zip*(es, interval).map(tu->tu.getT1()).take(10);  data:{"id":1,"value":"value1"}  data:{"id":2,"value":"value2"}  data:{"id":3,"value":"value3"}  data:{"id":4,"value":"value4"}  data:{"id":5,"value":"value5"}  data:{"id":6,"value":"value6"}  data:{"id":7,"value":"value7"}  data:{"id":8,"value":"value8"}  data:{"id":9,"value":"value9"}  data:{"id":10,"value":"value10"} |

|  |
| --- |
| Flux<String> es = Flux.<String>*generate*(sink->sink.next("value")); Flux<Long> interval = Flux.*interval*(Duration.*ofSeconds*(1)); return Flux.*zip*(es,interval).map(tu->new User(tu.getT2(),tu.getT1())).take(10);  data:{"id":0,"value":"value"}  data:{"id":1,"value":"value"}  data:{"id":2,"value":"value"}  data:{"id":3,"value":"value"}  data:{"id":4,"value":"value"}  data:{"id":5,"value":"value"}  data:{"id":6,"value":"value"}  data:{"id":7,"value":"value"}  data:{"id":8,"value":"value"}  data:{"id":9,"value":"value"}  https://raw.githubusercontent.com/reactor/reactor-core/v3.1.1.RELEASE/src/docs/marble/zipt.png |

Flux.range