



SPRING CAMP

Async@Spring

이일민 Toby

소프트웨어 개발업자
Epril
Wirebarley
KSUG

토비의 스프링

toby.epril.com
fb.com/tobyilee
youtube.com/tobyleetv



발표내용

- : 스프링 3.2~4.3에서의 비동기 개발 방법
- : @Async
- : Asynchronous Request Processing
- : AsyncRestTemplate

대상

: 중급

: 스프링 비동기 개발에 대한 기본 지식과 배경을 알고 있는
또는 사용해본 개발자

<http://www.springcamp.io/2017/>

: ~~스프링 비동기 개발에 관심만 있는 개발자~~

스프링 비동기 개발

기본 지식

자바 비동기 개발

서블릿 비동기 개발

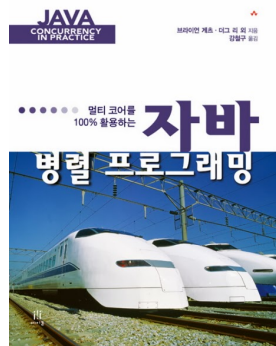
스프링 비동기 개발

자바 비동기 개발

서블릿 비동기 개발

스프링 비동기 개발

- 비동기와 논블록킹
- 자바5+
Future/Executor(s)
BlockingQueue
- 자바7
ForkJoinTask
- 자바8
CompletableFuture
- 자바9
Flow



자바 비동기 개발

서블릿 비동기 개발

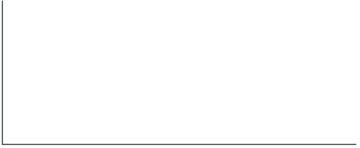
스프링 비동기 개발

- Servlet 3.0 Async Processing
AsyncContext
- Servlet 3.1
Non-Blocking IO
- Servlet 4.0

자바 비동기 개발

서블릿 비동기 개발

스프링 비동기 개발



- @Async
- Async Request Processing
 - Callable
 - DeferredResult
 - ResponseBodyEmitter
- AsyncRestTemplate

함께 + 시간을

동기/비동기

Singleton

Dependency Injection

Synchronous/Asynchronous

Singleton

Dependency Injection

Synchronous/Asynchronous



Singleton

Dependency Injection

Synchronous/Asynchronous

1



Singleton

Dependency Injection

Synchronous/Asynchronous

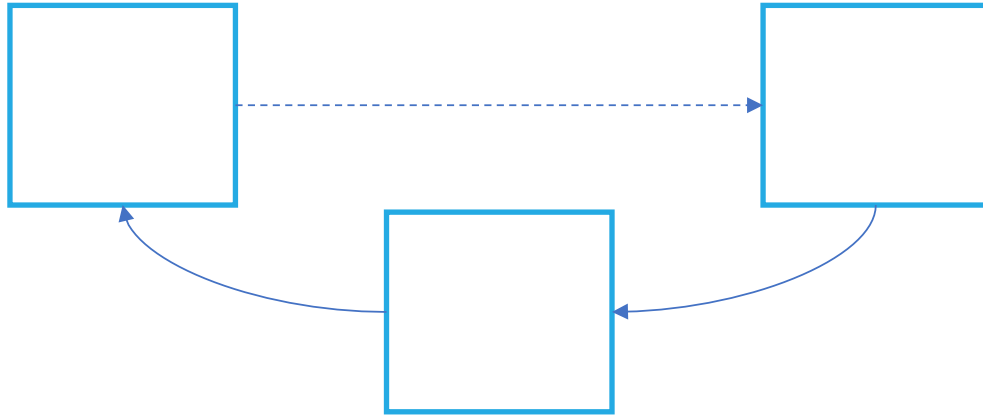


Singleton

Dependency Injection

Synchronous/Asynchronous

3



Singleton

Dependency Injection

Synchronous/Asynchronous



Singleton

Dependency Injection

Synchronous/Asynchronous

2+



Synchronous 동기

Synchronous



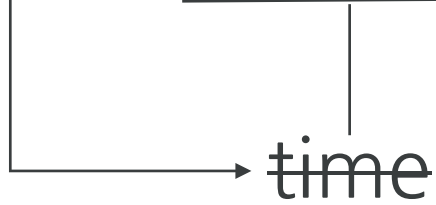
together

Synchronous



time

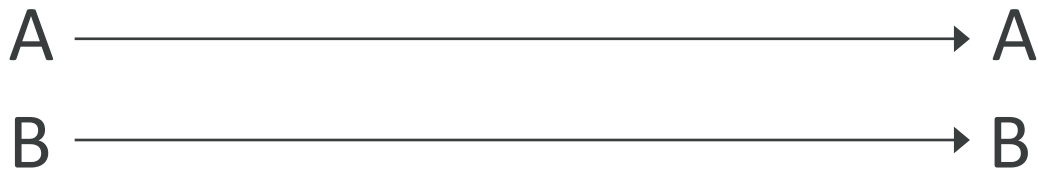
Asynchronous 비동기



동기/비동기를 언급할 때는

- 무엇과 무엇이?
- 어떤 시간을?

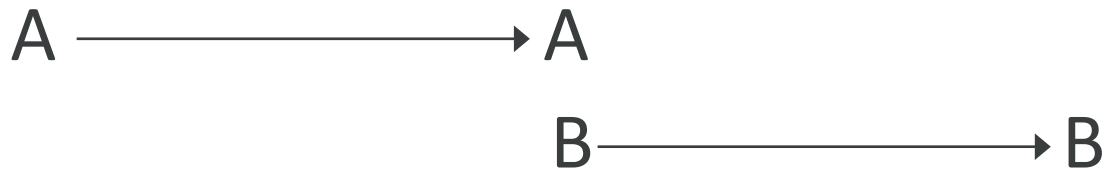
동기



A와 B가 시작시간 또는 종료시간이 일치하면 동기

- A,B 쓰레드가 동시에 작업을 시작하면 동기(CyclicBarrier)
- 메소드 리턴시간(A)과 결과를 전달받는 시간(B)이 일치하면 동기

동기



A가 끝나는 시간과 B가 시작하는 시간이 같으면 동기

- synchronized
- BlockingQueue

블록킹, 논블록킹

- 동기, 비동기와는 관점이 다름
- 내가 직접 제어할 수 없는 대상을 상대하는 방법
- 대상이 제한적임
 - IO
 - 멀티쓰레드 동기화

```
ExecutorService es = Executors.newCachedThreadPool();
```

```
String res = es.submit(() -> "Hello Async").get();
```

동기/비동기?

블록킹/논블록킹?

```
ExecutorService es = Executors.newCachedThreadPool();
```

```
String res = es.submit(() -> "Hello Async").get();
```

비동기

- 메소드 리턴 시간과 Callable의 실행 결과를 받는 시간이 일치하지 않음
- 블로킹,논블로킹은 고려할 대상이 아님

```
ExecutorService es = Executors.newCachedThreadPool();
```

```
String res = es.submit(() -> "Hello Async").get();
```

동기/블록킹

- 메소드 리턴 시간과 결과를 가져오는 시간이 일치
- 다른 쓰레드의 작업이 완료될 때까지 대기

비동기 실행

@Async

```
void service() {
```

```
    ...
```

```
    // (C)
```

```
}
```

```
// (A)
```

```
myService.service()
```

```
// (B)
```

```
void service() {  
    ...  
    // (B)  
}
```

```
// (A)  
myService.service()  
// (C)
```

```
void service() {  
    ...  
    // (B)  
}
```

```
// (A)  
myService.service()  
// (C)
```



```
void service() {  
    ...  
    // (B)  
}
```

```
// (A)  
myService.service()  
// (C)
```

```
@Async
void service() {
    ...
    // (B)
}
```

```
// (A)
myService.service()
// (C)
```

```
@Async
void service() {
    ...
    // (B)
}
```

```
// (A)
myService.service()
// (C)
```

```
@Async
void service() {
    ...
    // (B)
}
```

```
// (A)
myService.service()
// (C)
```

(B)와 (C)는 각각 다른 쓰레드에서 실행

```
String service() {  
    ...  
    return result;  
}
```

```
String result = myService.service()
```

@Async

```
String service() {  
    ...  
    return result;  
}
```

```
String result = myService.service()
```

@Async

```
String service() {
```

```
    ...
```

```
    return result;
```

```
}
```

```
String result = myService.service()
```

null

@Async가 지원하지 않는 리턴 타입

@Async 메소드의 리턴 타입

- void
- Future<T>
- ListenableFuture<T>
- CompletableFuture<T>

@Async

```
String service() {
```

```
    ...
```

```
    return result;
```

```
}
```

```
String result = myService.service()
```

@Async

```
Future<String> service() {  
    ...  
    return new AsyncResult<>(result);  
}
```

```
Future<String> f = myService.service();  
String res = f.get();
```

@Async

```
Future<String> service() {
```

```
    ...
```

```
    return new AsyncResult<>(result);
```

```
}
```

```
Future<String> f = myService.service();
```

```
String res = f.get();
```

@Async

```
ListenableFuture<String> service() {  
    ...  
    return new AsyncResult<>(result);  
}
```

```
ListenableFuture<String> f = myService.service();  
f.addCallback(  
    r-> log.info("Success: {}", r), e-> log.info("Error: {}", e));
```

@Async

```
ListenableFuture<String> service() {  
    ...  
    return new AsyncResult<>(result);  
}
```

```
ListenableFuture<String> f = myService.service();  
f.addCallback(  
    r-> log.info("Success: {}", r), e-> log.info("Error: {}", e));
```

@Async

```
CompletableFuture<String> service() {  
    ...  
    return CompletableFuture.completedFuture(result);  
}
```

```
CompletableFuture<String> f = myService.service();  
f.thenAccept(r-> System.out.println(r));
```

@Async

```
CompletableFuture<String> service() {
```

```
    ...
```

```
    return CompletableFuture.completedFuture(result);
```

```
}
```

```
CompletableFuture<String> f = myService.service();
```

```
f.thenAccept(r-> System.out.println(r));
```

SimpleAsyncTaskExecutor

- @Async가 사용하는 기본 TaskExecutor
- 쓰레드 풀 아님
- @Async를 본격적으로 사용한다면 실전에서 사용하지 말 것!

@Async가 사용하는 TaskExecutor

- SimpleAsyncTaskExecutor (기본)
- 다음 타입의 빈이 하나만 존재하면
 - Executor
 - ExecutorService
 - TaskExecutor
- @Async("**myExecutor**")

@Bean

```
TaskExecutor taskExecutor() {  
    ThreadPoolTaskExecutor te = new ThreadPoolTaskExecutor();  
    te.setCorePoolSize(10);  
    te.setMaxPoolSize(100);  
    te.setQueueCapacity(50);  
    te.initialize();  
    return te;  
}
```

@Bean

```
TaskExecutor taskExecutor() {
```

```
    ThreadPoolTaskExecutor te = new ThreadPoolTaskExecutor();
```

```
    te.setCorePoolSize(10);
```

```
    te.setMaxPoolSize(100);
```

```
    te.setQueueCapacity(50);
```

```
    te.initialize();
```

```
    return te;
```

```
}
```

50개의 @Async 메소드
호출이 동시에 일어나면
쓰레드는 몇 개가 만들어질까?

(1) 50개 (2) 100개 (3) 10개

Servlet 3.0+

비동기 서블릿

Servlet 3.0 – Asynchronous Processing

- 무엇인가 기다리느라 서블릿 요청처리를 완료하지 못하는 경우를 위해서 등장
- 서블릿에서 AsyncContext를 만든 뒤 즉시 서블릿 메소드 종료 및 서블릿 쓰레드 반납
- 어디서든 AsyncContext를 이용해서 응답을 넣으면 클라이언트에 결과를 보냄

```
@WebServlet(urlPatterns = "/hello")  
public class MyServlet2 extends HttpServlet {  
    public void doGet(HttpServletRequest request,  
        HttpServletResponse response) throws IOException {  
        response.getWriter().println("Hello Servlet");  
    }  
}
```

```
@WebServlet(urlPatterns = "/hello", asyncSupported = true)
public class MyServlet extends HttpServlet {
    public void doGet(HttpServletRequest request,
        HttpServletResponse response) throws IOException {
        AsyncContext ac = request.startAsync();
        Executors.newSingleThreadExecutor().submit(() -> {
            ac.getResponse().getWriter().println("Hello Servlet");
            ac.complete();
            return null;
        });
    }
}
```

```
@WebServlet(urlPatterns = "/hello", asyncSupported = true)  
public class MyServlet extends HttpServlet {  
    public void doGet(HttpServletRequest request,  
    HttpServletResponse response) throws IOException {  
        AsyncContext ac = request.startAsync();  
        Executors.newSingleThreadExecutor().submit(() -> {  
            ac.getResponse().getWriter().println("Hello Servlet");  
            ac.complete();  
            return null;  
        });  
    }  
}
```



```
@WebServlet(urlPatterns = "/hello", asyncSupported = true)  
public class MyServlet extends HttpServlet {  
    public void doGet(HttpServletRequest request,  
    HttpServletResponse response) throws IOException {  
        AsyncContext ac = request.startAsync();  
        Executors.newSingleThreadExecutor().submit(() -> {  
            ac.getResponse().getWriter().println("Hello Servlet");  
            ac.complete();  
            return null;  
        });  
    }  
}
```

```
@WebServlet(urlPatterns = "/hello", asyncSupported = true)
public class MyServlet extends HttpServlet {
    public void doGet(HttpServletRequest request,
        HttpServletResponse response) throws IOException {
        AsyncContext ac = request.startAsync();
        Executors.newSingleThreadExecutor().submit(() -> {
            ac.getResponse().getWriter().println("Hello Servlet");
            ac.complete();
            return null;
        });
    }
}
```

다른 쓰레드에 작업을 위임하고 서블릿은 종료
사용됐던 서블릿 쓰레드는 즉시 반납

```
@WebServlet(urlPatterns = "/hello", asyncSupported = true)
public class MyServlet extends HttpServlet {
    public void doGet(HttpServletRequest request,
        HttpServletResponse response) throws IOException {
        AsyncContext ac = request.startAsync();
        Executors.newSingleThreadExecutor().submit(() -> {
            ac.getResponse().getWriter().println("Hello Servlet");
            ac.complete();
            return null;
        });
    }
}
```

생성한 AsyncContext를 통해 서블릿의
응답(response) 처리

```
@WebServlet(urlPatterns = "/hello", asyncSupported = true)
public class MyServlet extends HttpServlet {
    Queue<AsyncContext> ctxs = new ConcurrentLinkedQueue<>();

    public void doGet(HttpServletRequest request,
        HttpServletResponse response) throws IOException {
        AsyncContext ac = request.startAsync();
        ctxs.add(ac);
    }
}
```

```
@WebServlet(urlPatterns = "/hello", asyncSupported = true)
public class MyServlet extends HttpServlet {
    Queue<AsyncContext> ctxs = new ConcurrentLinkedQueue<>();

    public void doGet(HttpServletRequest request,
        HttpServletResponse response) throws IOException {
        AsyncContext ac = request.startAsync();
        ctxs.add(ac);
    }
}
```

AsyncContext를 저장해두고 임의의
쓰레드에서 응답 결과를 넣을 수 있다

Servlet 3.1 – Non-blocking IO

- 요청 읽기와 응답 쓰기에 콜백을 이용
- ReaderListener
- WriterListener

Asynchronous Request Processing

비동기 스프링 @MVC

Servlet 3.0+ 비동기 요청 처리 기반의 @MVC

비동기 @MVC의 리턴 타입

- Callable<T>
- WebAsyncTask<T>
- DeferredResult<T>
- ListenableFuture<T>
- CompletionStage<T>
- ResponseBodyEmitter

Callable<T>

- AsyncTaskExecutor에서 실행될 코드를 리턴

@FunctionalInterface

```
public interface Callable<V> {  
    V call() throws Exception;  
}
```

```
@GetMapping("/hello")
```

```
String hello() {
```

```
    return "hello";
```

```
}
```

```
@GetMapping("/callable")  
Callable<String> callable() {  
    return () -> {  
        return "hello";  
    };  
}
```

```
@GetMapping("/callable")
Callable<String> callable() {
    return () -> {
        return "hello";
    };
}
```

String이 @MVC 결과 값의 타입

```
@GetMapping("/callable")  
Callable<String> callable() {  
    return () -> {  
        return "hello";  
    };  
}
```

컨트롤러 메소드가 종료된 뒤 별도의
TaskExecutor 내에서 실행

Callable의 리턴 값이 컨트롤러 메소드의 리턴 값으로 사용

WebAsyncTask<T>

- Callable에 timeout과 taskExecutor를 추가

```
public WebAsyncTask(Long timeout, String executorName,  
Callable<V> callable) { .. }
```

```
public WebAsyncTask(Long timeout, AsyncTaskExecutor executor,  
Callable<V> callable) { .. }
```

```
@GetMapping("/webasynctask")
WebAsyncTask<String> webasynctask() {
    return new WebAsyncTask<String>(5000L, "myAsyncExecutor",
        () -> {
            return "hello";
        });
}
```



```
@GetMapping("/webasynctask")
WebAsyncTask<String> webasynctask() {
    return new WebAsyncTask<String>(5000L, "myAsyncExecutor",
        () -> {
            return "hello";
        });
}
```

DeferredResult<T>

- 임의의 쓰레드에서 리턴 값 설정 가능
- Callable처럼 새로운 쓰레드를 만들지 않음
- 다양한 비동기 처리 기술과 손쉽게 결합

```
public class DeferredResult<T> {
```

```
..
```

```
public boolean setResult(T result) { .. }
```

```
public boolean setErrorResult(Object result) { .. }
```

```
@GetMapping("/deferredresult")
```

```
DeferredResult<String> deferredResult() {  
    DeferredResult dr = new DeferredResult();  
    queue.add(dr);  
    return dr;  
}
```

```
void eventHandler(String event) {  
    queue.forEach(dr->dr.setResult(event));  
}
```

```
@GetMapping("/deferredresult")
```

```
DeferredResult<String> deferredResult() {
```

```
    DeferredResult dr = new DeferredResult();
```

```
    queue.add(dr);
```

```
    return dr;
```

```
}
```

DeferredResult 생성해서 보관한 뒤 리턴

```
void eventHandler(String event) {
```

```
    queue.forEach(dr->dr.setResult(event));
```

```
}
```

```
@GetMapping("/deferredresult")  
DeferredResult<String> deferredResult() {  
    DeferredResult dr = new DeferredResult();  
    queue.add(dr);  
    return dr;  
}
```

다른 쓰레드에서 저장된 DeferredResult에 결과 값 쓰기

```
void eventHandler(String event) {  
    queue.forEach(dr->dr.setResult(event));  
}
```

DeferredResult와 @Async

- @Async메소드가 리턴하는
ListenableFuture에서 DeferredResult 사용
- 비동기 @MVC + 비동기 메소드 실행

```
@GetMapping("/drLf")
```

```
DeferredResult<String> drAndLf() {
```

```
    DeferredResult dr = new DeferredResult();
```

```
    ListenableFuture<String> lf = myService.async();
```

```
    lf.addCallback(r->dr.setResult(r), e->dr.setErrorResult(e));
```

```
    return dr;
```

```
}
```

```
@GetMapping("/drLf")
```

```
DeferredResult<String> drAndLf() {
```

```
    DeferredResult dr = new DeferredResult();
```

```
    ListenableFuture<String> lf = myService.async();
```

```
    lf.addCallback(r->dr.setResult(r), e->dr.setErrorResult(e));
```

```
    return dr;
```

```
}
```



```
@GetMapping("/drLf")
DeferredResult<String> drAndLf() {
    DeferredResult dr = new DeferredResult();

    ListenableFuture<String> lf = myService.async();
    lf.addCallback(r->dr.setResult(r), e->dr.setErrorResult(e));

    return dr;
}
```

비동기 작업을 시작한 뒤 결과에 대한
핸들러만 받음

```
@GetMapping("/drLf")
DeferredResult<String> drAndLf() {
    DeferredResult dr = new DeferredResult();

    ListenableFuture<String> lf = myService.async();
    lf.addCallback(r->dr.setResult(r), e->dr.setErrorResult(e));

    return dr;
}
```

비동기 작업이 끝나면 실행될 콜백에서
지연된 @MVC 결과값을 등록

ListenableFuture<T>

- 그릴줄 알고 스프링이 만들어 났음

@GetMapping("/drLf")

DeferredResult<String> drAndLf() {

DeferredResult dr = **new** DeferredResult();

ListenableFuture<String> lf = **myService**.async();

lf.addCallback(r->**dr**.setResult(r), e->**dr**.setErrorResult(e));

return dr;

}

조금 전에 만든 코드

```
@GetMapping("/lf")
```

```
    ListenableFuture<String> listenableFuture() {
```

```
        return myService.async();
```

```
    }
```

```
@GetMapping("/lf")  
ListenableFuture<String> listenableFuture() {  
    return myService.async();  
}
```

DeferredResult 생성, 콜백 등록과 콜백 호출시 결과를 넘기는 것을 스프링이 알아서 해줌

ListenableFuture<T>의 한계

- 두 가지 이상의 비동기 작업을 순차적으로 혹은 동시에 수행하고 결과를 조합해서 @MVC의 리턴 값으로 넘기려면?

```
@GetMapping("/composesync")
String compose() {
    String res1 = myService.sync();
    String res2 = myService.sync2(res1);
    return res2;
}
```



```
@GetMapping("/composesync")  
String compose() {  
    String res1 = myService.sync();  
    String res2 = myService.sync2(res1);  
    return res2;  
}
```

```
@GetMapping("/composesync")  
String compose() {  
    String res1 = myService.sync();  
    String res2 = myService.sync2(res1);  
    return res2;  
}
```

첫 번째 작업의 결과를 가져와서 두 번째 작업 에 전달

@GetMapping("/composeasync")

```
    ListenableFuture<String> asyncCompose() {  
        ListenableFuture<String> res1 = myService.async();  
        ListenableFuture<String> res2 = myService.async2(res1.????);  
        return res2;  
    }
```

```
@GetMapping("/composeasync")
ListenableFuture<String> asyncCompose() {
    ListenableFuture<String> res1 = myService.async();
    ListenableFuture<String> res2 = myService.async2(res1.????);
    return res2;
}
```

```
@GetMapping("/composeasync")  
ListenableFuture<String> asyncCompose() {  
    ListenableFuture<String> res1 = myService.async();  
    ListenableFuture<String> res2 = myService.async2(res1.????);  
    return res2;  
}
```

비동기 작업 결과를 다음 작업의 파라미터로 넘기려면?

ListenableFuture<T> 조합

- 두 개 이상의 비동기 작업을 결합할 때는 다시 콜백 + DeferredResult 방식으로
- 비동기 작업의 성공 콜백에서 다음 비동기 작업 시도
- 최종 비동기 작업의 성공 콜백에서 DeferredResult에 결과 전달

@GetMapping("/composeasync")

```
DeferredResult<String> asyncCompose() {  
    DeferredResult dr = new DeferredResult();  
    ListenableFuture<String> f1 = myService.async();  
    f1.addCallback(res1 -> {  
        ListenableFuture<String> f2 = myService.async2(res1);  
        f2.addCallback(res2 -> {  
            dr.setResult(res2);  
        }, e -> {  
            dr.setErrorResult(e);  
        });  
    }, e->{  
        dr.setErrorResult(e);  
    })  
};  
return dr;  
}
```

@GetMapping("/composeasync")

```
DeferredResult<String> asyncCompose() {  
    DeferredResult dr = new DeferredResult();  
    ListenableFuture<String> f1 = myService.async();  
    f1.addCallback(res1 -> {  
        ListenableFuture<String> f2 = myService.async2(res1);  
        f2.addCallback(res2 -> {  
            dr.setResult(res2);  
        }, e -> {  
            dr.setErrorResult(e);  
        });  
    }, e -> {  
        dr.setErrorResult(e);  
    })  
    );  
    return dr;  
}
```



```
ListenableFuture<String> f1 = myService.async();  
f1.addCallback(res1 -> {  
    ListenableFuture<String> f2 = myService.async2(res1);  
    f2.addCallback(res2 -> {  
        dr.setResult(res2);  
    }, e -> {  
        dr.setErrorResult(e);  
    });  
}, e->{  
    dr.setErrorResult(e);  
}  
);
```

```
ListenableFuture<String> f1 = myService.async();  
f1.addCallback(res1 -> {  
    ListenableFuture<String> f2 = myService.async2(res1);  
    f2.addCallback(res2 -> {  
        dr.setResult(res2);  
    }, e -> {  
        dr.setErrorResult(e);  
    });  
}, e->{  
    dr.setErrorResult(e);  
}  
);
```

```
ListenableFuture<String> f1 = myService.async();
f1.addCallback(res1 -> {
    ListenableFuture<String> f2 = myService.async2(res1);
    f2.addCallback(res2 -> {
        dr.setResult(res2);
    }, e -> {
        dr.setErrorResult(e);
    });
}, e->{
    dr.setErrorResult(e);
}
);
```

```
ListenableFuture<String> f1 = myService.async();  
f1.addCallback(res1 -> {  
    ListenableFuture<String> f2 = myService.async2(res1);  
    f2.addCallback(res2 -> {  
        dr.setResult(res2);  
    }, e -> {  
        dr.setErrorResult(e);  
    });  
}, e->{  
    dr.setErrorResult(e);  
}  
);
```

ListenableFuture<T> 조합

- 여러 개의 비동기 작업을 조합해서 비동기 @MVC의 결과로 사용할 수 있다
- 콜백의 중첩으로 코드가 복잡해진다
- 예외 콜백의 내용이 동일한 경우 중복이 발생한다

@GetMapping("/composeasync2")

```
DeferredResult<String> asyncCompose2() {  
    DeferredResult dr = new DeferredResult();  
    ListenableFuture<String> f1 = myService.async();  
    f1.addCallback(res1 -> {  
        ListenableFuture<String> f2 = myService.async2(res1);  
        f2.addCallback(res2 -> {  
            ListenableFuture<String> f3 = myService.async3(res2);  
            f3.addCallback(res3 -> {  
                dr.setResult(res3);  
            }, e -> {  
                dr.setErrorResult(e);  
            });  
        }, e -> {  
            dr.setErrorResult(e);  
        });  
    }, e -> {  
        dr.setErrorResult(e);  
    });  
    return dr;  
}
```

콜백 헬

```
@GetMapping("/composeasync2")
DeferredResult<String> asyncCompose2() {
    DeferredResult dr = new DeferredResult();
    ListenableFuture<String> f1 = myService.async();
    f1.addCallback(res1 -> {
        ListenableFuture<String> f2 = myService.async2(res1);
        f2.addCallback(res2 -> {
            ListenableFuture<String> f3 = myService.async3(res2);
            f3.addCallback(res3 -> {
                dr.setResult(res3);
            }, e -> {
                dr.setErrorResult(e);
            });
        }, e -> {
            dr.setErrorResult(e);
        });
    }, e->{
        dr.setErrorResult(e);
    });
    return dr;
}
```

CompletableFuture<T> 조합

- 함수형 스타일 접근방법
- CompletionStage의 조합으로 간결하게 표현
- 중복되는 예외 처리를 한번에
- 다양한 비동기/동기 작업의 변환, 조합, 결합 가능
- CompletableFuture/CompletionStage 사용에 대한 학습이 필요
- ExecutorService의 활용 기법도 익혀야 함

@GetMapping("/composeasync")

```
DeferredResult<String> asyncCompose() {  
    DeferredResult dr = new DeferredResult();  
    ListenableFuture<String> f1 = myService.async();  
    f1.addCallback(res1 -> {  
        ListenableFuture<String> f2 = myService.async2(res1);  
        f2.addCallback(res2 -> {  
            dr.setResult(res2);  
        }, e -> {  
            dr.setErrorResult(e);  
        });  
    }, e -> {  
        dr.setErrorResult(e);  
    })  
};  
return dr;  
}
```

방금전 LF 비동기 작업 조합 코드

@GetMapping("/composecf")

```
CompletableFuture<String> cfCompose() {  
    CompletableFuture<String> f1 = myService.casync();  
    CompletableFuture<String> f2 =  
        f1.thenCompose(res1 -> myService.casync2(res1));  
    return f2;  
}
```

```
@GetMapping("/composecf")
CompletableFuture<String> cfCompose() {
    CompletableFuture<String> f1 = myService.casync();
    CompletableFuture<String> f2 =
        f1.thenCompose(res1 -> myService.casync2(res1));
    return f2;
}
```

```
@GetMapping("/composecf")
CompletableFuture<String> cfCompose() {
    CompletableFuture<String> f1 = myService.casync();
    CompletableFuture<String> f2 =
        f1.thenCompose(res1 -> myService.casync2(res1));
    return f2;
}
```

비동기 작업의 결과를 받아 이를 이용해
다음 비동기 작업을 수행하는 비동기 작업을 생성

```
@GetMapping("/composecf")  
CompletableFuture<String> cfCompose() {  
    return myService.casync()  
        .thenCompose(res1 -> myService.casync2(res1));  
}
```

더 간결하게

```
@GetMapping("/composecf")  
CompletableFuture<String> cfCompose() {  
    return myService.casync()  
        .thenCompose(myService::casync2);  
}
```

한 번 더 간결하게

```
@GetMapping("/composecf2")
CompletableFuture<String> cfCompose2() {
    return myService.casync()
        .thenCompose(res1 -> myService.casync2(res1))
        .thenCompose(res2 -> myService.casync3(res2));
}
```

더 많은 조합도 간결하게

비동기 작업의 결합

- 2개 이상의 비동기 작업을 병렬적으로 실행하고 결과를 모아서 결과 값을 만들어 내는 비동기 작업 구성
 - `ListenableFuture`의 콜백 구조로는 어렵다
 - `CompletableFuture`로는 손쉽게 가능하다

AsyncRestTemplate

비동기 논블로킹 API 호출

비동기-논블록킹 API 요청과 @MVC

- RestTemplate은 동기-블록킹 방식
- API 호출 작업 동안 쓰레드 점유
- 블로킹으로 인한 컨텍스트 스위칭 발생
- 비동기 @MVC를 사용했다고 하더라도 쓰레드 자원의 효율적인 사용이 어려움

AsyncRestTemplate

- 스프링 4.0부터 RestTemplate의 비동기-논블록킹 버전인 AsyncRestTemplate을 이용할 수 있다

```
RestTemplate rt = new RestTemplate();  
for(int i=0; i<100; i++) {  
    ResponseEntity<String> res =  
        rt.getForEntity("http://localhost:8080/api", String.class);  
    System.out.println(res.getBody());  
}
```



1초 걸리는 API 호출이라면?

```
AsyncRestTemplate art = new AsyncRestTemplate();  
for(int i=0; i<100; i++) {  
    ListenableFuture<ResponseEntity<String>> lf =  
        art.getForEntity("http://localhost:8080/api", String.class);  
    lf.addCallback(r-> System.out.println(r.getBody()), e->{});  
}
```

```
AsyncRestTemplate art = new AsyncRestTemplate();  
for(int i=0; i<100; i++) {  
    ListenableFuture<ResponseEntity<String>> lf =  
        art.getForEntity("http://localhost:8080/api", String.class);  
    lf.addCallback(r-> System.out.println(r.getBody()), e->{});  
}
```

```
AsyncRestTemplate art = new AsyncRestTemplate();  
for(int i=0; i<100; i++) {  
    ListenableFuture<ResponseEntity<String>> lf =  
        art.getForEntity("http://localhost:8080/api", String.class);  
    lf.addCallback(r-> System.out.println(r.getBody()), e->{});  
}
```

```
AsyncRestTemplate art = new AsyncRestTemplate();  
for(int i=0; i<100; i++) {  
    ListenableFuture<ResponseEntity<String>> lf =  
        art.getForEntity("http://localhost:8080/api", String.class);  
    lf.addCallback(r-> System.out.println(r.getBody()), e->{});  
}
```

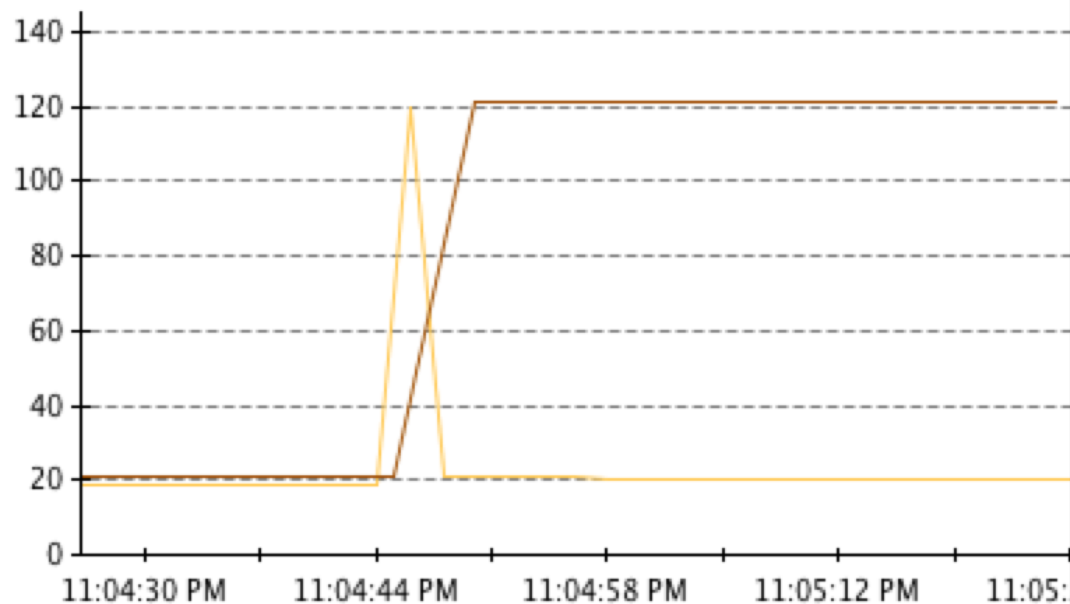


```
AsyncRestTemplate art = new AsyncRestTemplate();  
for(int i=0; i<100; i++) {  
    ListenableFuture<ResponseEntity<String>> lf =  
        art.getForEntity("http://localhost:8080/api", String.class);  
    lf.addCallback(r-> System.out.println(r.getBody()), e->{});  
}
```

```
AsyncRestTemplate art = new AsyncRestTemplate();  
for(int i=0; i<100; i++) {  
    ListenableFuture<ResponseEntity<String>> lf =  
        art.getForEntity("http://localhost:8080/api", String.class);  
    lf.addCallback(r-> System.out.println(r.getBody()), e->{});  
}
```

1초 걸리는 API 호출 100번 수행하는데 1초

▼ Live Thread Graph



- ☐ Daemon Live Thread Count
- ☒ Peak Live Thread Count
- ☒ Total Live Thread Count

```
AsyncRestTemplate art = new AsyncRestTemplate();
```

- 비동기이고 논블록킹(콜백)으로 결과를 돌려받지만 논블록킹 IO를 사용하지 않음
- 이럴바엔 @Async에 RestTemplate 쓰고 말지

Netty4ClientHttpRequestFactory factory =

new Netty4ClientHttpRequestFactory(**new** NioEventLoopGroup(1));

AsyncRestTemplate art = **new** AsyncRestTemplate(factory);

Netty4ClientHttpRequestFactory factory =

new Netty4ClientHttpRequestFactory(**new** NioEventLoopGroup(**1**));

AsyncRestTemplate art = **new** AsyncRestTemplate(factory);

HTTP 클라이언트 라이브러리와
AsyncTaskExecutor는 자유롭게 선택해서 DI

논블록킹 IO 쓰레드 1개만 할당

```
Netty4ClientHttpRequestFactory factory =  
    new Netty4ClientHttpRequestFactory(new NioEventLoopGroup(1));  
AsyncRestTemplate art = new AsyncRestTemplate(factory);
```

- 1초 걸리는 API 호출 100개를 1개 쓰레드로 1초에 처리
- 비동기 @MVC이므로 서버릿 쓰레드도 점유하지 않음

비동기 API 호출의 조합과 결합은 어떻게?

- AsyncRestTemplate은 ListenableFuture로만 리턴
- 콜백의 콜백의 콜백의... 헬이 또!
- @Async처럼 조합이 간편해지는 CompletableFuture로 리턴하면 안 되나?
 - 리턴 오버로딩은 없음. CF는 LF의 서브타입도 아님
 - 스프링 이슈 트래커의 답변: 재주껏 CompletableFuture로 만들어 써라

필요하면 확장해서 쓴다

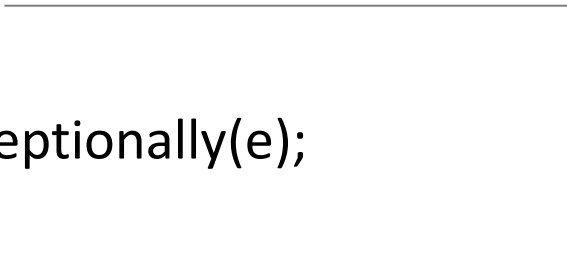
- `ListenableFuture`도 Java5의 `FutureTask`를 간단히 확장해서 콜백이 가능하도록 스프링에서 만든 것
- `ListenableFuture`를 `CompletableFuture`로 만드는 것도 간단하다

```
public <T> CompletableFuture<T> toCFuture(ListenableFuture<T> lf) {  
    CompletableFuture<T> cf = new CompletableFuture<>();  
    lf.addCallback((r)-> {  
        cf.complete(r);  
    }, (e) -> {  
        cf.completeExceptionally(e);  
    });  
    return cf;  
}
```

```
public <T> CompletableFuture<T> toCFuture(ListenableFuture<T> lf) {  
    CompletableFuture<T> cf = new CompletableFuture<>();  
    lf.addCallback((r)-> {  
        cf.complete(r);  
    }, (e) -> {  
        cf.completeExceptionally(e);  
    });  
    return cf;  
}
```

```
public <T> CompletableFuture<T> toCFuture(ListenableFuture<T> lf) {  
    CompletableFuture<T> cf = new CompletableFuture<>();  
    lf.addCallback((r)-> {  
        cf.complete(r);  
    }, (e) -> {  
        cf.completeExceptionally(e);  
    });  
    return cf;  
}
```

```
public <T> CompletableFuture<T> toCFuture(ListenableFuture<T> lf) {  
    CompletableFuture<T> cf = new CompletableFuture<>();  
    lf.addCallback((r)-> {  
        cf.complete(r);  
    }, (e) -> {  
        cf.completeExceptionally(e);  
    });  
    return cf;  
}
```



왜 이름이 CompletableFuture일까?

@RequestMapping("/callbackhell")

```
public DeferredResult<String> asyncRestCallbackHell(int idx) {  
    DeferredResult<String> result = new DeferredResult<>();  
    ListenableFuture<ResponseEntity<String>> lf1 =  
    asyncRest.getForEntity("http://localhost:8081/service?msg=Spring" + idx, String.class);  
    lf1.addCallback((r) -> {  
        ListenableFuture<ResponseEntity<String>> lf2 =  
            asyncRest.getForEntity("http://localhost:8081/hello?msg=" + r.getBody(), String.class);  
        lf2.addCallback((r2) -> {  
            ListenableFuture<String> lf3 = asyncService.asyncWork(r2.getBody());  
            lf3.addCallback((r3) -> {  
                result.setResult(r3);  
            }, (e) -> {  
                result.setErrorResult(e.getMessage());  
            });  
        }, (e) -> {  
            result.setErrorResult(e.getMessage());  
        });  
    }, (e) -> {  
        result.setErrorResult(e.getMessage());  
    });  
    return result;  
}
```

AsyncRestTemplate + ListenableFuture

@RequestMapping("/cfuture")

public CompletableFuture<String> cfuture(int idx) {

return toCFuture(art.getForEntity("http://localhost:8081/service?" + idx, String.class))

.thenCompose(r -> toCFuture(art.getForEntity("http://localhost:8081/hello? " + r.getBody(), String.class)))

.thenApplyAsync(r2 -> service.syncWork(r2.getBody()), workExecutor)

.exceptionally(ex -> ex.getMessage());

}

AsyncRestTemplate + CompletableFuture

잘 알고 쓰자

비동기 스포링 정리

비동기 작업과 API 호출이 많은 @MVC 앱이라면

- @MVC
- AsyncRestTemplate + 논블록킹 IO 라이브러리
- @Async와 적절한 TaskExecutor
- ListenableFuture, CompletableFuture

TaskExecutor(쓰레드풀)의 전략적 활용이 중요

- 스프링의 모든 비동기 기술에는 ExecutorService의 세밀한 설정이 가능
- CompletableFuture도 ExecutorService의 설계가 중요
- 코드를 보고 각 작업이 어떤 쓰레드에서 어떤 방식으로 동작하는지, 그게 어떤 효과와 장점이 있는지 설명할 수 있어야 한다
- 벤치마킹과 모니터링 중요

비동기 스프링 기술을 사용하는 이유?

- 모르면 그냥 쓰지 말자
- 설명할 수 있어야 하고
- 증명할 수 있어야 한다

비동기 스프링 기술을 사용하는 이유

- IO가 많은 서버 앱에서 서버 자원의 효율적으로 사용해 성능을 높이려고 (낮은 레이턴시 높은 처리율)
- 서버 외부의 이벤트를 받아 처리하는 것과 같은 비동기 작업이 필요해서
- 유행이라니까
- 폼나 보아서

비동기 스프링 기술의 단점?

- 모르면 그냥 쓰지 말자
- 잘못쓰면 코드는 복잡해지고
- 디버깅 힘들고
- 그런데 아무런 혜택이 없을 수도

배먹은 것

- HTTP Streaming
- ResponseBodyEmitter
- Async Listener
- Async Message Reception
- @Scheduled
- @MVC에 RxJava, Reactor 사용하기

감사합니다

스프링 비동기 라이브 코딩은 youtube.com/tobyleetv