

스프링 3.2

이일민

Epril
toby.epril.com

19:00 ~ 21:00
(120분)

Break-Time 10분 포함

난이도수준: 중급

주제: 스프링 3.2의 신기술과 활용 전략

내용:

- 스프링 3.2의 새로운 기능과 의미
- Servlet 3의 비동기 요청 처리 지원
- SpringMVC Test 프레임워크
- SpringMVC REST 지원 기능
- 기타 SpringMVC의 확장된 기능
- Spring 3.2로의 업그레이드 전략

* 참고자료: 토비의 스프링 3.1

스프링 3.2

- 2012년 12월
- 3.x의 마지막 업그레이드
- 주로 Spring@MVC

ibatis / mybatis

```
package
```

```
org.springframework.orm.ibatis.support;
```

```
@Deprecated
```

```
public abstract class SqlMapClientDaoSupport  
extends DaoSupport {
```



Servlet 3 based Asynchronous Request Processing

비동기 요청 처리

비동기 요청 처리 학습순서

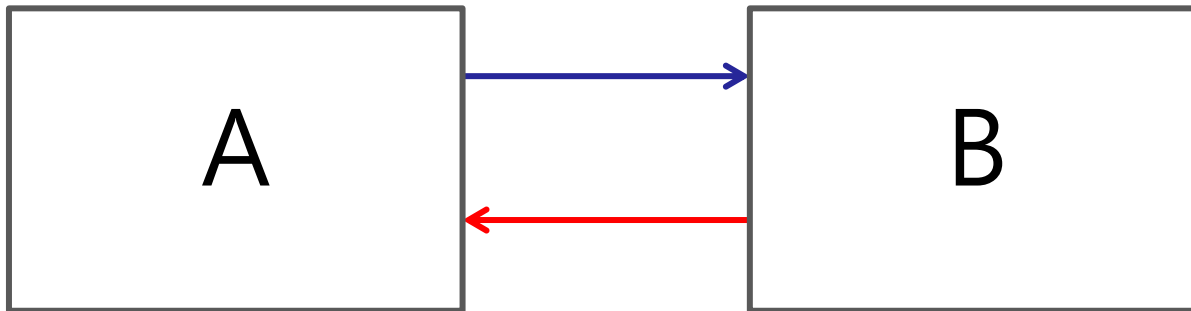
- 비동기 요청 처리
- Servlet 3.0 이전의 비동기 요청 처리
- Servlet 3.0의 비동기 요청 처리
- 스프링 3.2의 @MVC 비동기 요청처리
- 비동기 요청 처리의 미래

비동기(asynchronous)

- 의미?
 - 둘 이상의 사물이나 사건이 동시에 존재[발생]하지 않는
 - 시스템 간의 상호 작용이 어느 정도의 시차를 두고 일어나는 것.
- 비동기 = 논블록킹?
 - 그럼 동기 = 블록킹?
 - Java NIO = 비동기?

비동기(asynchronous) 요청 처리

- (메소드 호출을 통한) 작업 요청이 어떻게 진행되고 어떻게 결과를 받는가
 - 작업 진행이 요청을 보낸 스레드에서?
 - 결과가 메소드 리턴 값인가?



블록킹, 논블록킹

- 블록킹

- 요청한 작업을 마칠 때까지 대기(리턴 X)
- `InputStream.read()`

- This method **blocks** until input data is available, the end of the stream is detected, or an exception is thrown.

- 논블록킹

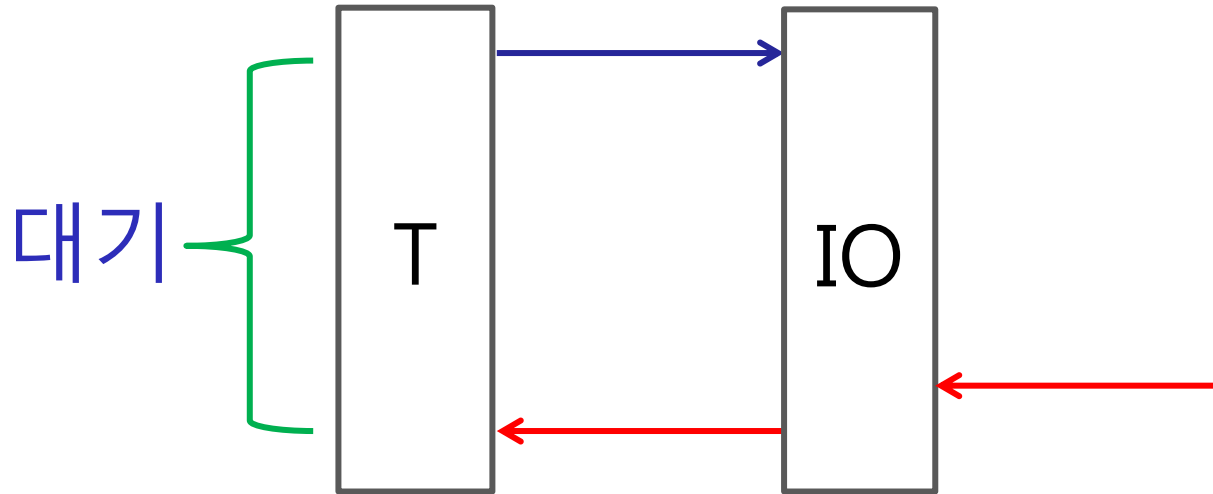
- 요청한 작업을 즉시 마칠 수 없으면 그냥 리턴
- `Selector.selectNow()`

- This method performs a **non-blocking** selection operation

- 주로 IO 읽기, 쓰기 메소드 호출

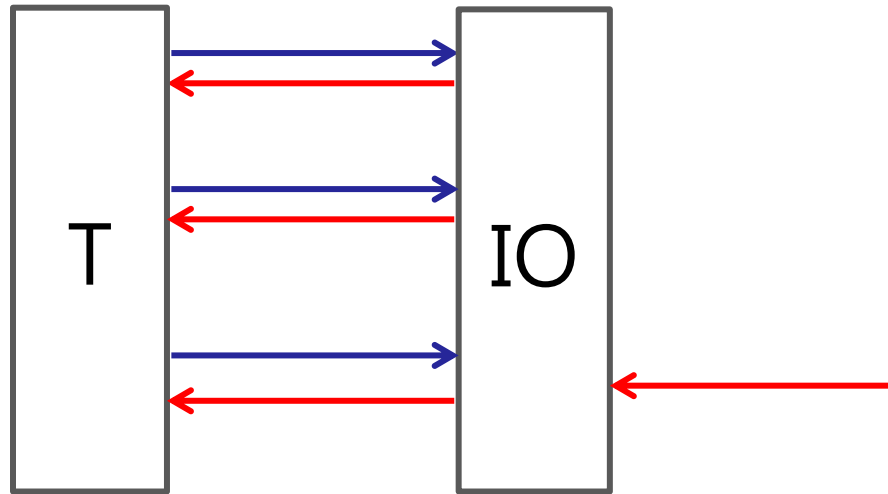
블록킹 IO와 쓰레드

- IO 처리를 위해 쓰레드 하나 할당
- Thread per Connection



논블록킹 IO와 쓰레드

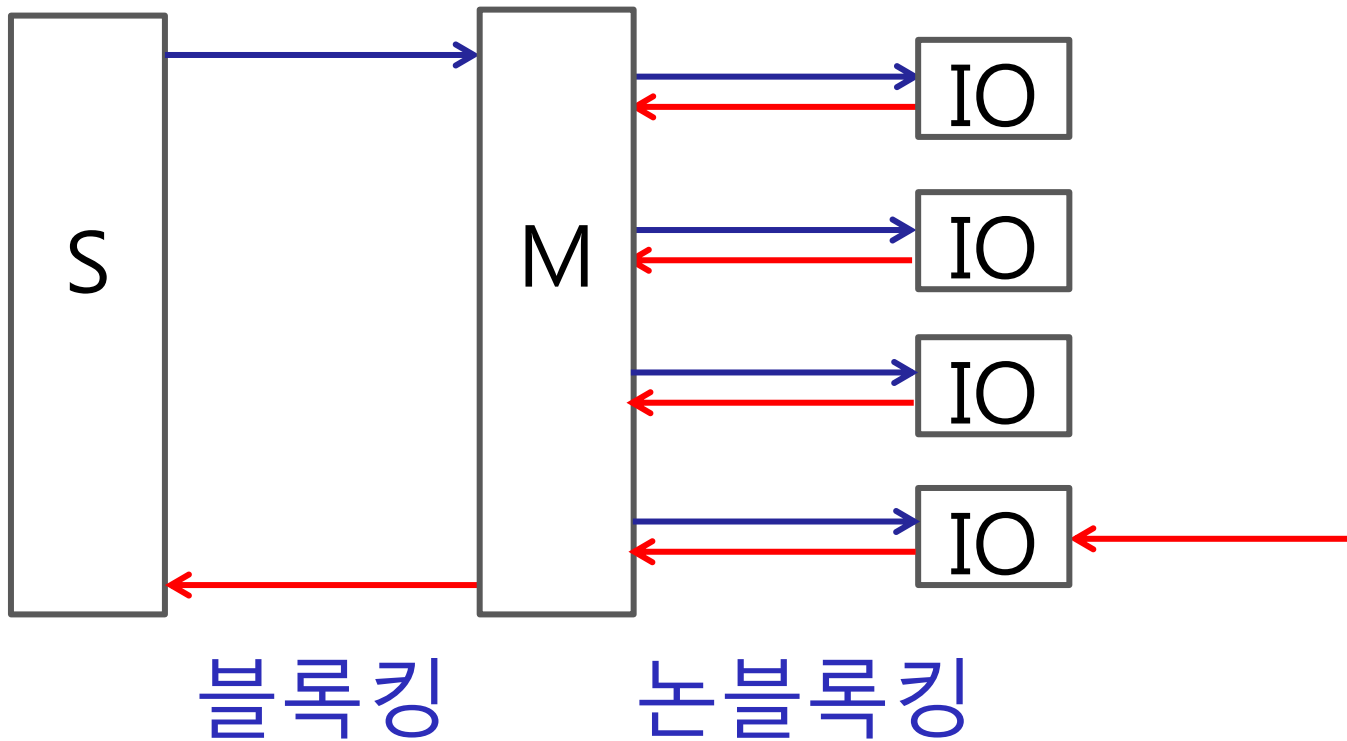
- 하나의 쓰레드가 여러 개의 IO 처리 가능



블록킹 메소드 + 논블록킹 IO

- Selector.select()

- This method performs a **blocking** selection operation

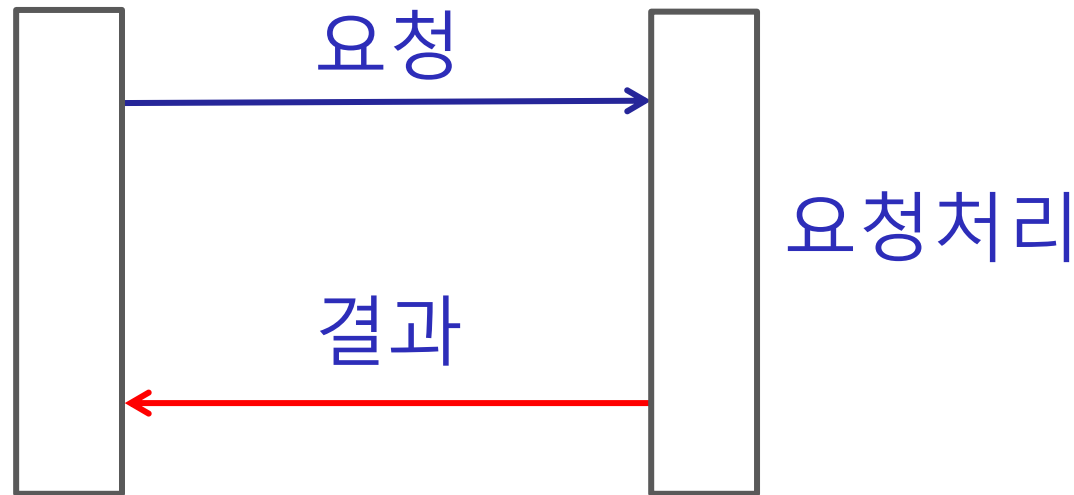


블록킹, 논블록킹 vs 동기, 비동기

- 오해
 - 블록킹 = 동기
 - 논블록킹 = 비동기
- 가능 조합
 - 동기 블록킹
 - 동기 논블록킹
 - 비동기 논블록킹
 - 비동기 블록킹(?)

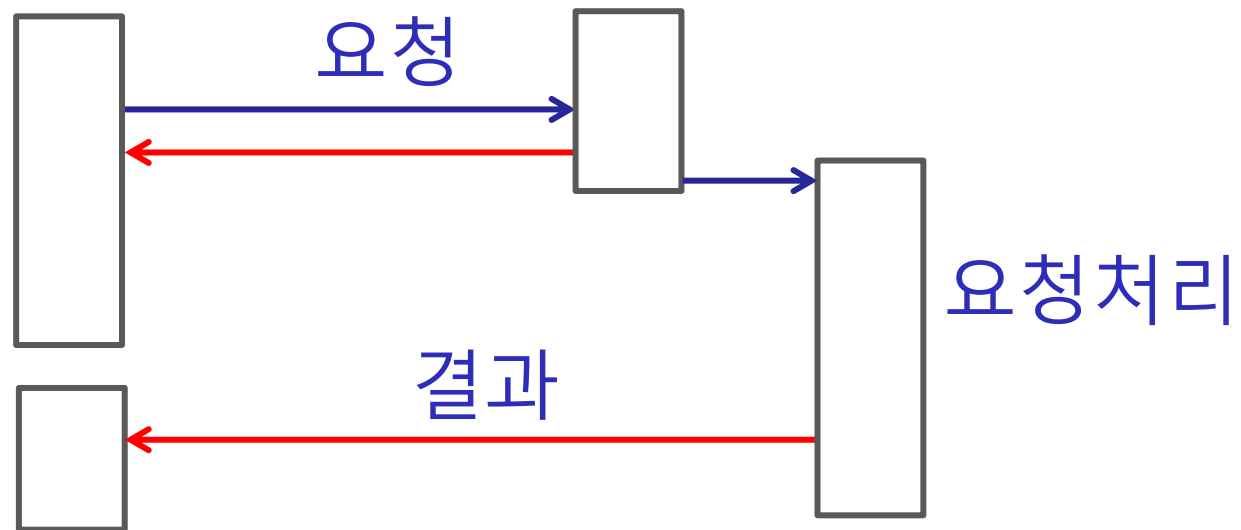
동기(synchronous) 요청 처리

- 작업 요청의 결과를 직접 받는 것
- 메소드 호출을 통한 작업 요청의 결과를 리턴 값으로 받으면 동기 요청 처리



비동기(asynchronous) 요청 처리

- 작업 요청 결과를 이후에 간접적으로 받는 것
- 메소드 호출의 리턴 값이 작업 결과가 아님
- 별도의 스레드에서 작업 수행



동기=블록킹?

- 동기 블록킹
 - 결과가 나올 때까지 기다렸다 리턴 값으로 결과 전달
- 동기 논블록킹
 - 결과가 없으면 바로 리턴
 - 결과가 있으면 리턴 값으로 결과 전달
 - Selector.selectNow()

비동기=논블록킹?

- 비동기 논블록킹

- 작업 요청을 받아서 별도로 진행하게 하고 바로 리턴 (논블록킹)
- 결과는 별도의 작업 후 간접적으로 전달

- 비동기 블록킹?

- 리턴 값으로 결과를 줄 것도 아닌데 왜 블록킹?
- 하지만 두 가지 이상의 대상(메소드호출+IO처리)에 대해 동시에 설명하면 비동기 메소드 요청 처리+동기 IO도 얼마든지 가능

비동기 요청 처리 결과 전달

- 콜백
 - 비동기 요청 처리시 콜백 전달
 - 요청 처리가 완료되면 콜백 호출
- 결과 조회
 - Future<V>
 - 요청이 완료됐는지 확인

java.nio.channels.AsynchronousByteChannel

```
<A> void read(ByteBuffer dst, A attachment,  
              CompletionHandler<Integer,? super A> handler);
```

```
Future<Integer> read(ByteBuffer dst);
```

```
public interface CompletionHandler<V,A> {  
    void completed(V result, A attachment);  
    void failed(Throwable exc, A attachment);  
}
```

```
public interface Future<V> {  
    boolean isDone();  
    V get() throws InterruptedException, ExecutionException;  
    ...  
}
```

스프링 템플릿/콜백도 비동기?

- 대부분 동기-블록킹
- JdbcTemplate.queryForObject()
 - 콜백은 템플릿 내에서 동작하는 바뀌는 로직을 전달하기 위한 용도

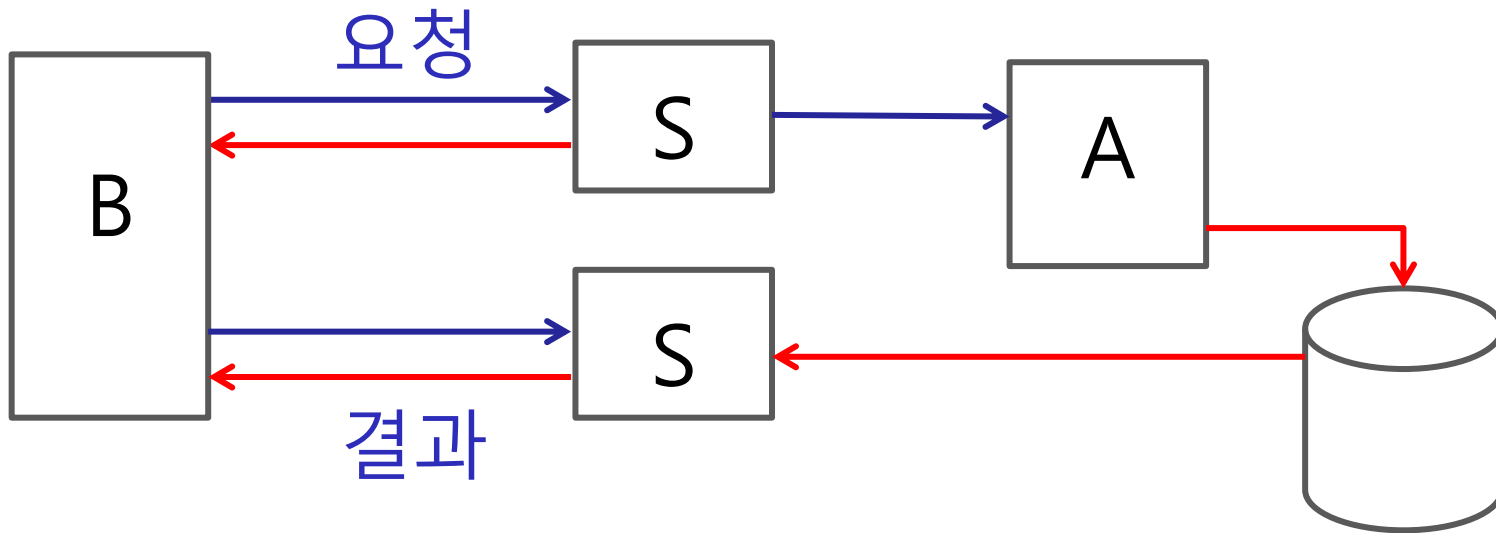
비동기 작업 요청의 필요성

- 시간이 오래 걸리는 작업을 별도로 수행
- 자원(쓰레드)의 효율적인 사용
- Event-driven / 서버 push (long-polling)

SERVLET 3.0 이전의 비동기 요청처리

웹 애플리케이션의 비동기 작업

- 시간이 오래 걸리는 작업
- 결과를 바로 확인할 필요가 없는 작업



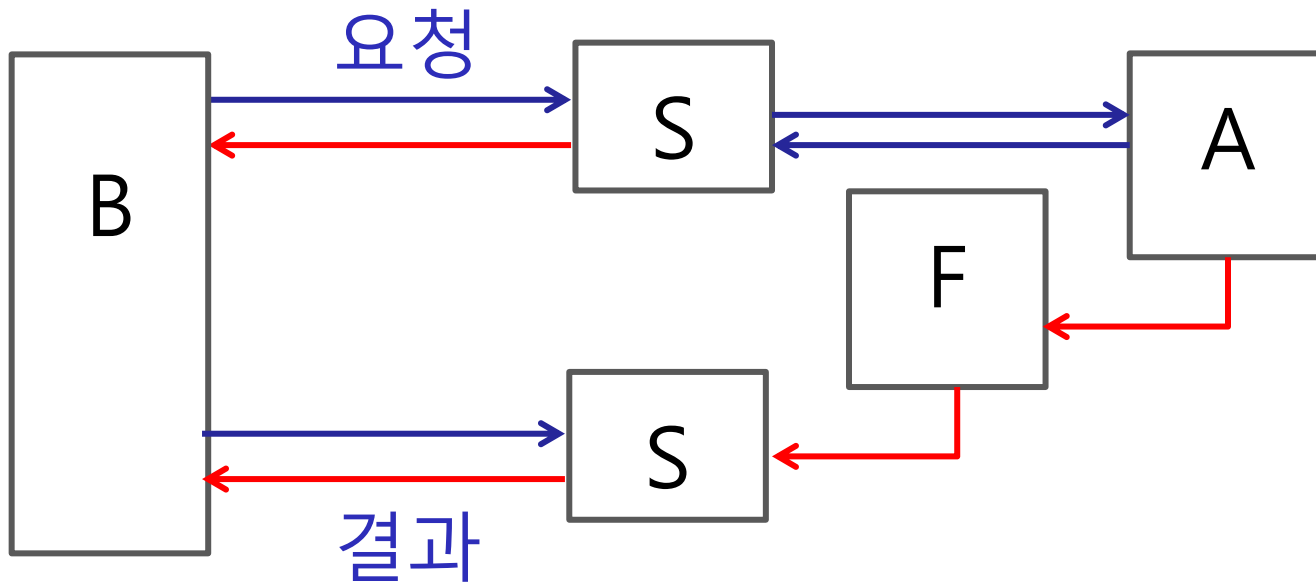
@Async

- 별도의 스레드에서 비동기 작업 수행

```
@Async  
public void batchJob() {  
    ...  
}
```


Future 이용 비동기 작업 결과 확인

- Future<V>: 비동기 작업 결과
- @Async 메소드의 리턴 타입을 Future<V>



@Async Future<V>

```
@Async
```

```
public Future<String> doBatchJob() {
```

```
    // 요청 처리
```

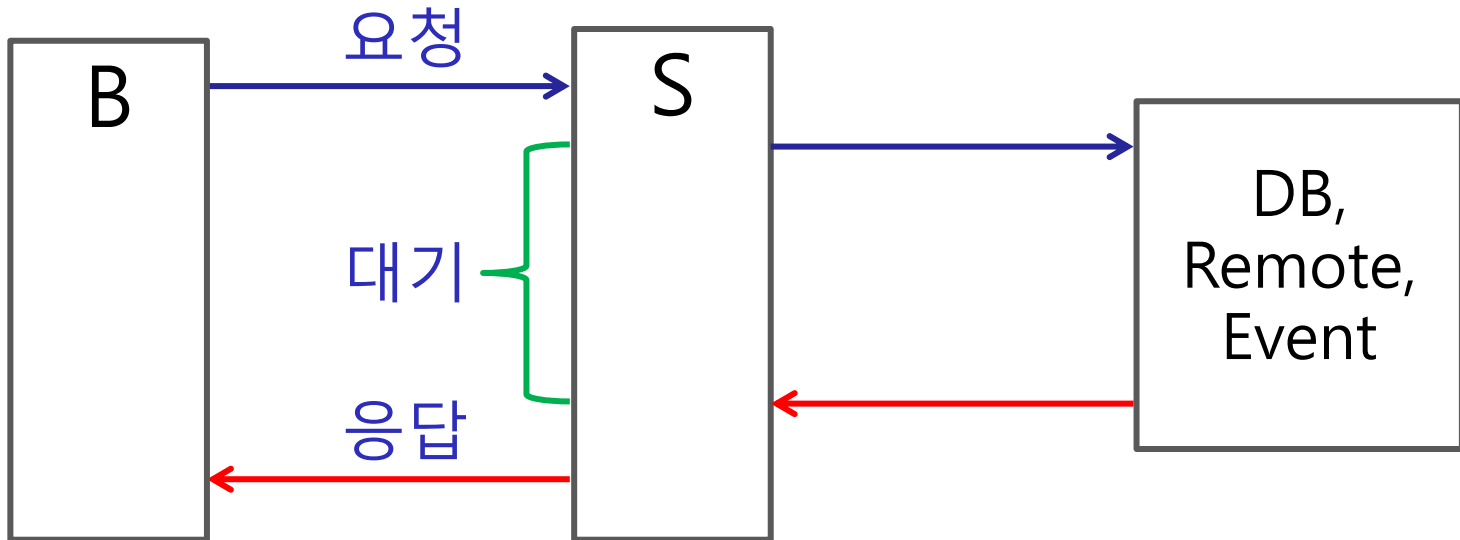
```
    return new AsyncResult<String>(result);
```

```
}
```

SERVLET 3.0의 비동기 요청처리

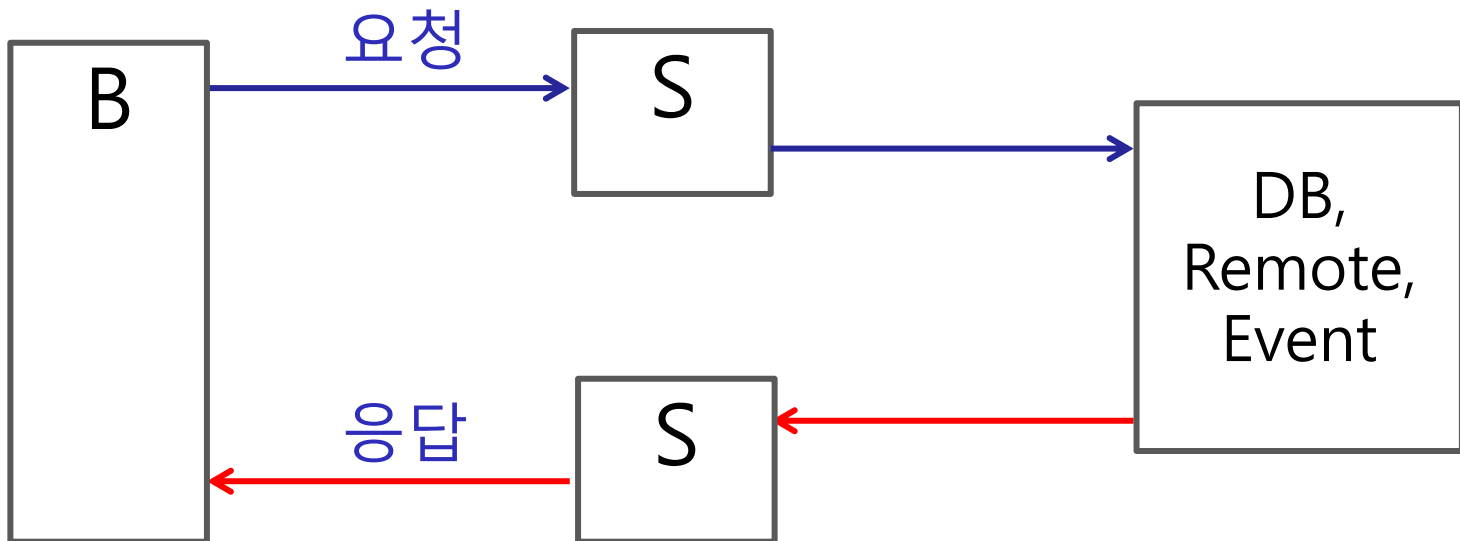
서블릿 단위의 비동기 요청 처리

- @Async/Future는 HTTP요청/응답이 많아진다
- 한번의 HTTP요청/응답 내에서 작업
- 서블릿 쓰레드의 대기 상태를 최소화



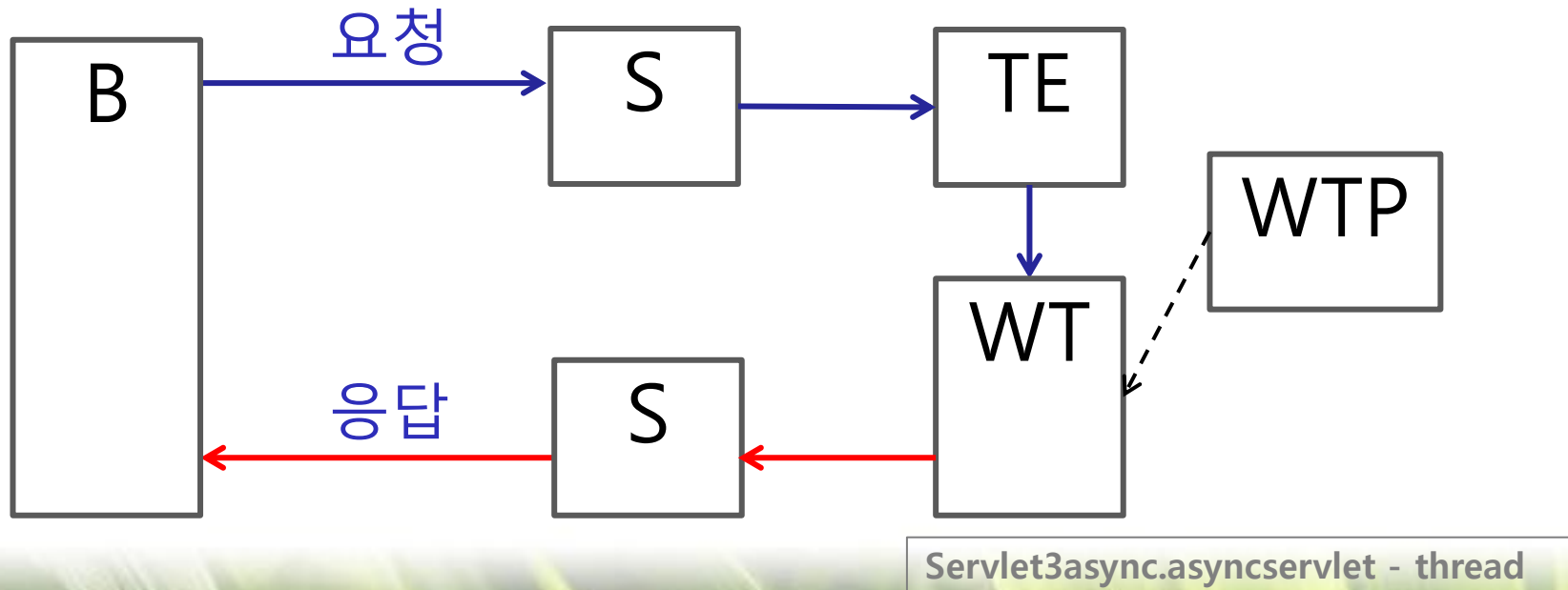
서블릿 단위의 비동기 요청 처리

- 비동기 작업을 시작하거나 대기 상태로 만들고 서블릿 작업 즉시 종료
- 가용 서블릿 쓰레드 증가



비동기 요청 처리 활용 (1)

- 시간이 오래 걸리는 외부 연동 작업을 비동기로 Executor에 요청
- 작업(비동기, 애플리케이션) 스레드에서 작업 수행



비동기 요청 처리 활용 (1)

- 시나리오

- 서블릿 스레드 최대 100개
- 외부 연동을 포함한 장시간 수행 요청 50개
- 동기 방식
 - 장시간 요청 처리 중 가용 스레드 50개
- 비동기 방식
 - 작업 스레드 풀 10개
 - 가용 스레드 90개
 - DB나 웹 API 호출 같은 외부 연동을 논블록킹으로 할 수 있다면 가용 스레드 99개
 - `async-mysql-connector`, `adbcj`
 - Support an asynchronous API for RestTemplate (Spring 4.0)

서블릿 3 비동기 요청 처리

```
@WebServlet(urlPatterns="/asynchello",
            asyncSupported=true)
public class AsyncServlet extends HttpServlet {

    final AsyncContext ac = req.startAsync();

    this.executor.execute(new Runnable() {
        // 비동기 작업
        // ac.dispatch() 또는 ac.complete()
    });
}
```


톰캣 7과 서블릿 3.0 비동기 요청 처리

- 톰캣의 HTTP Connector

- 디폴트: `org.apache.coyote.http11.Http11Protocol`

- HTTP/1.1

- 블록킹 IO

- `org.apache.coyote.http11.Http11NioProtocol`

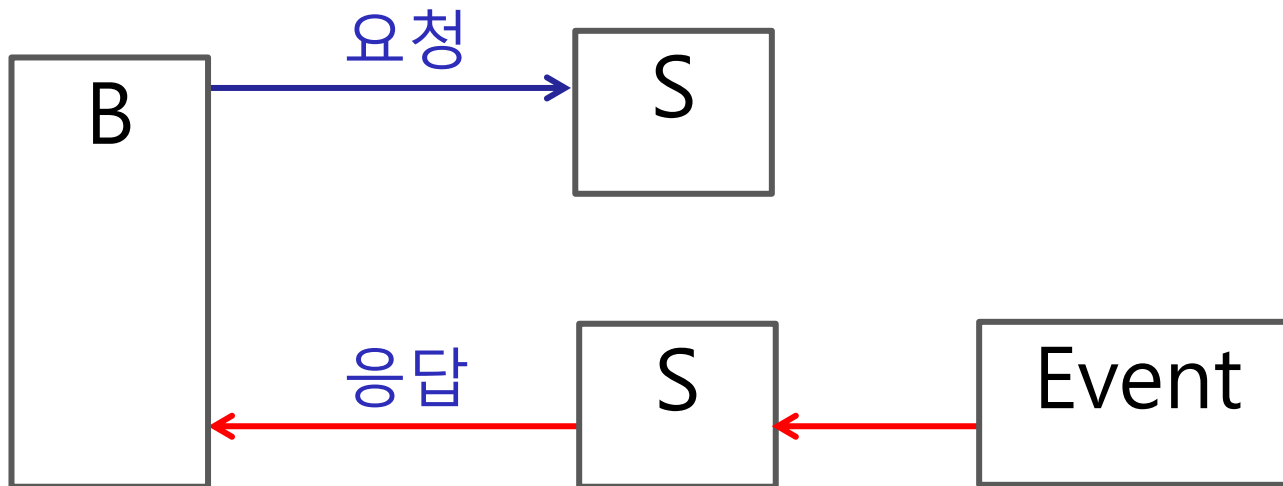
- 서블릿 비동기 요청 처리를 사용하려면 NIO Connector 설정 필요

- 한계

- NIO Connector를 사용하더라도 서블릿의 IO는 비동기 방식 아니므로 HTTP 응답을 처리하기 위해서 서블릿 스레드가 필요

비동기 요청 처리 활용 (2)

- 서블릿 응답을 이벤트 발생시까지 대기
 - 다른 요청, JMS, Redis(pub/sub), 스케줄러, ...
 - 서버 push - 롱 폴링
 - 별도의 작업 스레드 생성 필요 없음



비동기 요청 처리 활용 (2)

- 시나리오

- 서블릿 쓰레드 최대 100개

- 동기 방식

- 롱-폴링 요청 최대 100개

- 가용 쓰레드 0

- 다른 요청 불가능

- 비동기 방식

- 롱-폴링 요청 최대 10,000개(톰캣 기본 설정, OS지원)

- 가용쓰레드 100 (응답 보낼 때만 잠시 사용)

스프링 3.2의 비동기 요청처리

서블릿 3.0 비동기의 단점

- Executor/쓰레드 풀 관리 코드 작성 부담
- 서블릿으로 개발!
 - Spring@MVC 컨트롤러 사용 불가

서블릿 3.2 비동기 기능


- 기존 스프링 @MVC 컨트롤러 작성 방식과 동일
 - 활용1(쓰레드 풀을 이용한 비동기 작업)과 활용2(응답을 지연시키고 별도 이벤트에 의해서 처리) 시나리오에 모두 @MVC 컨트롤러 코드 거의 그대로 사용
- 활용1
 - Callable, WebAsyncTask
- 활용2
 - DeferredResult

비동기 @MVC 컨트롤러

- 컨트롤러 리턴 타입을 다음 중 하나로 변경
 - Callable
 - WebAsyncTask
 - DeferredResult

```
@RequestMapping("/hello")  
public String hello() {
```

```
@RequestMapping("/hello")  
public Callable<String> hello() {
```



java.util.concurrent.Callable

- Runnable과 유사하게 별도의 스레드에서 실행되는 코드
- 실행 결과 타입 지정

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


Callable 컨트롤러

- @MVC 컨트롤러의 모든 리턴 타입 사용
 - 뷰 이름, ModelAndView, @ResponseBody...

```
@RequestMapping("/call")
@ResponseBody
public Callable<String> call() {
    return new Callable<String>() {
        @Override
        public String call() throws Exception {

            // 작업
            String res = ...

            return result;
        }
    };
}
```

Callable 코드 실행

- 스프링이 제공하는 AsyncTaskExecutor로 실행
 - 디폴트는 SimpleAsyncTaskExecutor
 - 적절한 AsyncTaskExecutor 설정 필요

```
@EnableWebMvc
```

```
public class WebConfig extends WebMvcConfigurerAdapter {
```

```
    public void configureAsyncSupport(AsyncSupportConfigurer  
    configurer) {  
        ThreadPoolTaskExecutor e = new ThreadPoolTaskExecutor();  
        e.setCorePoolSize(5);  
        e.initialize();  
        configurer.setTaskExecutor(e);  
    }
```

WebAsyncTask

- Callable과 동일한 방식
- Callable 실행 스레드 풀 지정
 - 작업 종류에 따라 스레드 풀 분리
 - Facebook API 스레드 풀, DB 스레드 풀, ...
 - AsyncTaskExecutor 빈 이름 사용
- Timeout 설정
- Callable을 WebAsyncTask에 담아서 리턴

WebAsyncTask

```
@RequestMapping("/facebooklink")
public WebAsyncTask<String> facebooklink() {
    return new WebAsyncTask<String>(
        30000L, // Timeout
        "facebookTaskExecutor", // TaskExecutor
        new Callable<String>() {
            @Override
            public String call() throws Exception {
                // 작업
                return result;
            }
        }
    );
}
```

DeferredResult

- MVC 컨트롤러 리턴 값을 별개의 작업에서 지정할 수 있도록 해줌
 - JMS, AMQP, 스케줄러, Email, IM, Redis, 다른 HTTP 요청
- 작업 스레드를 생성하지 않는다
- 빈의 필드에 정의된 Queue등에 저장해두고 이벤트 발생시 사용하고 제거

DeferredResult 컨트롤러

- Callable과 마찬가지로 @MVC 컨트롤러 메소드의 리턴 값 타입 지정

```
@RequestMapping("/async")
@ResponseBody
public DeferredResult<String> async() {
    final DeferredResult<String> result =
        new DeferredResult<>();

    queue.add(result);

    return result;
}
```

DeferredResult 결과 지정

- Queue 등에 저장된 DeferredResult를 가져와 @RequestMapping 메소드 리턴 값 지정

```
@RequestMapping("/event")
@ResponseBody
public String event(String msg) {
    for(DeferredResult<String> result : queue) {
        result.setResult(msg);
    }

    return "OK";
}
```

DEMO



Timeout 설정

- 톰캣 서버 Connector asyncTimeout
 - 디폴트 10초

```
<Connector connectionTimeout="20000" asyncTimeout="60000"  
protocol="org.apache.coyote.http11.Http11NioProtocol"
```

- AsyncSupportConfigurer

```
public void configureAsyncSupport(AsyncSupportConfigurer  
configurer) {  
    configurer.setDefaultTimeout(30000);  
}
```

- WebAsyncTask, DeferredResult

Timeout 처리

- DeferredResult.onTimeout

```
final DeferredResult<String> result = new DeferredResult<>();
final Date start = new Date();
result.onTimeout(new Runnable() {
    @Override
    public void run() {
        // Timeout 처리
    }
});
```

- Timeout Result

```
final DeferredResult<String> result =
    new DeferredResult<>(60000L, "TIMEOUT");
```

Timeout 인터셉터

public interface

CallableProcessingInterceptor {

**<T> Object handleTimeout(NativeWebRequest
request, Callable<T> task) throws Exception;**

비동기 @MVC 고려할 것

- 예외 처리
 - HandlerExceptionResolver 방식
 - Callable: throw Ex()
 - DeferredResult: setErrorResult(error value 또는 ex)
- 비동기 지원 필터
- 비동기 지원 인터셉터
- ThreadLocal 사용
 - 서블릿 요청/응답에 하나 이상의 쓰레드 사용

스프링 설정

비동기 @MVC를 위한 web.xml

```
<web-app xmlns="http://java.sun.com/xml/ns/javaee"  
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee  
http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"  
    version="3.0">
```

```
<servlet>  
    <servlet-name>spring</servlet-name>  
    <servlet-  
class>org.springframework.web.servlet.DispatcherServlet  
    </servlet-class>  
    <init-param>...</init-param>  
    <load-on-startup>1</load-on-startup>  
    <async-supported>true</async-supported>  
</servlet>
```

비동기 @MVC를 위한 스프링 XML

```
<mvc:annotation-driven>
```

```
  <mvc:async-support  
    default-timeout="3000"  
    task-executor="asyncTaskExecutor">  
    <mvc:callable-interceptors>  
      <bean class="async.TimeoutProcessingInterceptor" />  
    </mvc:callable-interceptors>  
  </mvc:async-support>
```

```
</mvc:annotation-driven>
```


web.xml 제거

- 서블릿 3.0
 - ServletContainerInitializer
- 스프링 3.1
 - WebApplicationInitializer
- 스프링 3.2
 - AbstractAnnotationConfigDispatcherServletInitializer

루트 애플리케이션 컨텍스트 - XML

```
<listener>
  <listener-class>
    org.springframework.web.context.ContextLoaderListener
  </listener-class>
</listener>

<context-param>
  <param-name>contextClass</param-name>
  <param-value>
    org.springframework.web.context.support.AnnotationConfigWebApplic
    ationContext
  </param-value>
</context-param>
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>myproject.config.AppConfig</param-value>
</context-param>
```

루트 애플리케이션 컨텍스트 – S3.1

```
AnnotationConfigWebApplicationContext ac =  
    new AnnotationConfigWebApplicationContext();  
ac.register(AppConfig.class);  
ServletContextListener listener = new ContextLoaderListener(ac);  
servletContext.addListener(listener);
```

루트 애플리케이션 컨텍스트 – S3.2

```
protected Class<?>[] getRootConfigClasses() {  
    return new Class<?>[] { AppConfig.class };  
}
```

서블릿 애플리케이션 컨텍스트 - XML

```
<servlet>
  <servlet-name>spring</servlet-name>
  <servlet-
class>org.springframework.web.servlet.DispatcherServlet</servlet-  class>
  <init-param>
    <param-name>contextClass</param-name>
    <param-value>
      org.springframework.web.context.support.AnnotationConfigWebApplic
      ationContext
    </param-value>
  </init-param>
  <init-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>myproject.config.AppConfig</param-value>
  </init-param>
  <load-on-startup>1</load-on-startup>
  <async-supported>true</async-supported>
</servlet>

<servlet-mapping>
  <servlet-name>spring</servlet-name>
  <url-pattern>/</url-pattern>
</servlet-mapping>
```

서블릿 애플리케이션 컨텍스트 – S3.1

```
AnnotationConfigWebApplicationContext sac =  
    new AnnotationConfigWebApplicationContext();  
sac.register(WebConfig.class);
```

```
ServletRegistration.Dynamic dispatcher = servletContext.addServlet(  
    "spring", new DispatcherServlet(sac));  
dispatcher.setLoadOnStartup(1);  
dispatcher.addMapping("/");
```

서블릿 애플리케이션 컨텍스트 – S3.2

```
protected Class<?>[] getServletConfigClasses() {  
    return new Class<?>[] { WebConfig.class };  
}
```

```
protected String[] getServletMappings() {  
    return new String[] { "/" };  
}
```

기타 기능

ContentNegotiationManager

- 요청정보의 미디어 타입을 결정하는 전략 관리
 - 기존 ContentNegotiatingViewResolver, @RequestMapping의 consumes, produces등에서 다루던 내용을 일관된 방식으로 정리
 - RequestMappingHandlerMapping, RequestMappingHandlerAdapter, ExceptionHandlerExceptionResolver, ContentNegotiatingViewResolver 등에 적용

Matrix Variables

- <http://www.w3.org/DesignIssues/MatrixURIs.html>
- 팀 버너스리 제안
- 다양한 용도로 사용중
 - Servlet `jsessionid`
 - `http://www.myserver.com/index.html;jsessionid=1234`
 - JAX-RS: `@MatrixParam`
- `@MatrixVariable`

GenericHttpMessageConverter

- Generic 타입 파라미터 지원
 - Jaxb2CollectionHttpMessageConverter
 - MappingJackson2HttpMessageConverter
 - MappingJacksonHttpMessageConverter
- RestTemplate
 - ParameterizedTypeReference
- @RequestBody

@ControllerAdvice

- 컨트롤러에 적용되는 어드바이스 정의
 - @ExceptionHandler
 - @InitBinder
 - @ModelAttribute
- 빈 스캔 대상
 - @Component 메타 애노테이션

참고정보

- 스프링 3.2 레퍼런스 / API Doc
- What's New in Spring 3.2
 - <http://www.youtube.com/watch?v=GSsWMLiKF-M>
- Servlet 3.0 스펙 / 서적
- 스프링 3.2 실무 프로젝트(예정)



감사합니다