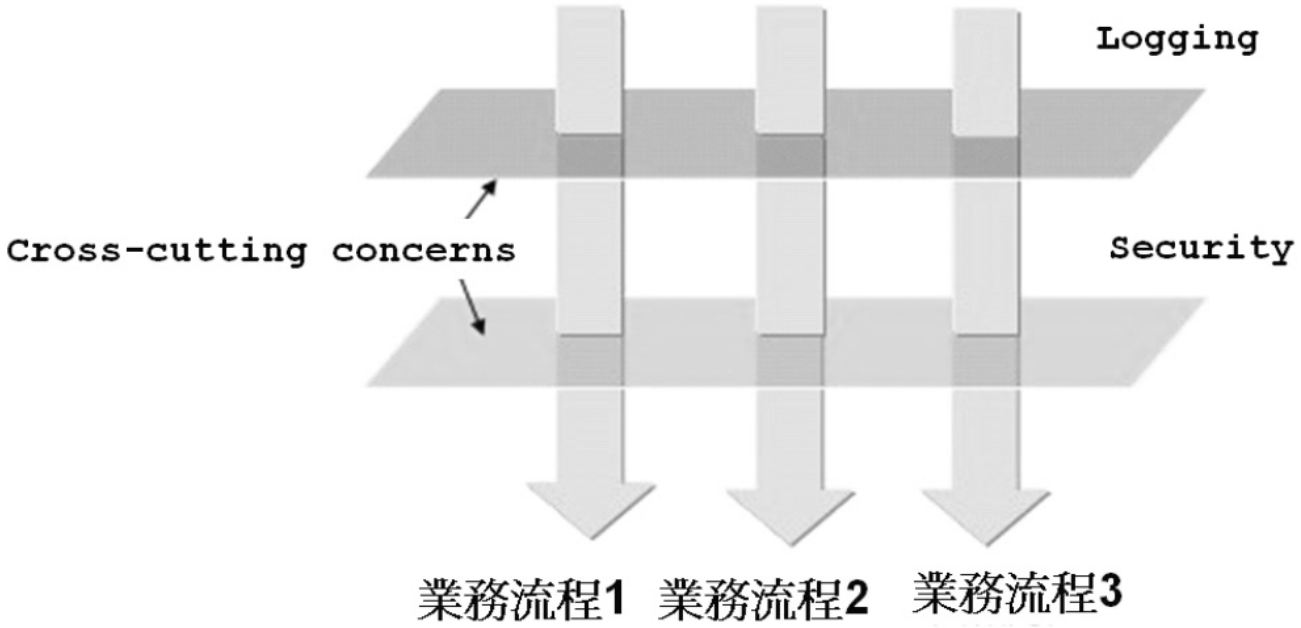
10. 例外處理

Spring Boot 有兩個特點，一個是 IoC (Inversion of Control)，也就是前面提到所有的 `@Component` 和 `@Bean` 都會被交給 Spring Container 處理，Container 會將其實例化，想要使用時再用 `@Autowired` 將其拿出來，跟傳統的方式並不一樣，傳統上使用物件的方式如下：

```
Logger logger = new Logger();
logger.log("紀錄")
```

要用時再創造物件，並且由使用者以 `new` 關鍵字進行建構，然而 Spring Boot 不是如此，實例化這件事情是 Container 幫忙處理好了，也因此這樣的行為被稱作 IoC 控制反轉。

第二件事情則是 AOP，也就是 Aspect-Oriented Programming，跟 OOP 一樣是種編程概念：



想像每支程式碼處理業務邏輯前，都會做一些共同的事情，最簡單是紀錄 log，如果 request 進來不記錄 log，那當錯誤發生時，要如何知道當時會發生錯誤的情境是什麼，所以通常都會有 log 的存在，除此之外可能還有例外處理、安全..... 等，這些東西每個 class 都需要做，然而如果每支 class 都自行撰寫這些要處理的事情，程式碼會變得難以維護，因此出現 AOP 這樣的想法，寫一個切面程式，讓每個要執行這個方法的 class，在進來之前都去執行，更改也能改變切面的內容。

Spring Boot 怎麼實踐 AOP ? 一樣使用 annotation，這邊展示一個專門處理例外的 annotation `@ControllerAdvice` ，其實還有其他 AOP 的 annotation (`@Before` 、 `@After`)，想使用可自行查詢：

```
|--com.example.demospringboot
   |--DemospringbootApplication.java
|--com.example.demospringboot.configuration
   |--SwaggerConfig.java
   |--RestConfiguration.java
|--com.example.demospringboot.controller
   |--CarController.java // 修改的檔案
   |--TestController.java
   |--ProductController.java
|--com.example.demospringboot.controller.advice
   |--WebExceptionHandler.java // 新增的檔案
|--com.example.demospringboot.dto
   |--CarRequest.java // 修改的檔案
   |--CarResponse.java
|--com.example.demospringboot.entity
   |--Car.java
   |--CarPK.java
|--com.example.demospringboot.exception
   |--ErrorInputException.java // 新增的檔案
|--com.example.demospringboot.model
   |--Product.java
|--com.example.demospringboot.repository
   |--CarRepository.java
|--com.example.demospringboot.service
   |--CarService.java
   |--ProductService.java
|--com.example.demospringboot.service.impl
   |--CarServiceImpl.java
   |--ProductServiceImpl.java
```

首先新增一支 `ErrorInputException.java` ，當作 API 欄位輸入錯誤時的 Exception，這邊內部就不加東西，如果想要新增訊息可自行覆寫：

```java
public class ErrorInputException extends Exception {
    private static final long serialVersionUID = 1L;
}
```

下一步是新增一個專門處理 Exception 的切面：

```java
@ControllerAdvice
public class WebExceptionHandler {

    @ResponseBody
    @ExceptionHandler(ErrorInputException.class)
    public CarResponse2 handleErrorInputException(){
        CarResponse2 carResponse = new CarResponse2();
        carResponse.setDatas(null);
        carResponse.setMessage("ErrorInputException");
        return carResponse;
    }
}
```

只要有任何地方丟出 `ErrorInputException` ，這邊就會接到錯誤，並執行 method 內的程式碼。那在哪邊丟出錯誤？看看 `pom.xml` 、 `CarRequest.java` 和 `CarController.java` ， `pom.xml` 要新增兩個 `<dependency>` ，目的是晚點 `CarRequest.java` 會用到的 `@Valid` annotation：

```xml
<!--如果是2.3以上的SpringBoot版本，這個annotation才需要補上，參考連結在下方-->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-validation</artifactId>
</dependency>
<dependency>
    <groupId>javax.validation</groupId>
    <artifactId>validation-api</artifactId>
</dependency>
```

> https://stackoverflow.com/questions/48614773/spring-boot-validation-annotations-valid-and-notblank-not-working/48614909

補上這兩個 `<dependency>` 後，看看 `CarRequest.java` 這個 DTO，可以看到在兩個 property 上加上 `@NotBlank` 的 annotation，這表示當 request 進來後， Spring Boot 會幫忙檢核這兩個變數是否為空值，是空值的話就有錯誤：

```java
public class CarRequest {
    @NotBlank
    @JsonProperty("Manufacturer")
    private String manufacturer;

    @NotBlank
    @JsonProperty("Type")
    private String type;

    @JsonProperty("Min_Price")
    private BigDecimal minPrice;

    @JsonProperty("Price")
    private BigDecimal price;

    public String getManufacturer() {
        return manufacturer;
    }

    public void setManufacturer(String manufacturer) {
        this.manufacturer = manufacturer;
    }

    public String getType() {
        return type;
    }

    public void setType(String type) {
        this.type = type;
    }

    public BigDecimal getMinPrice() {
        return minPrice;
    }

    public void setMinPrice(BigDecimal minPrice) {
        this.minPrice = minPrice;
    }

    public BigDecimal getPrice() {
        return price;
    }

    public void setPrice(BigDecimal price) {
        this.price = price;
    }
}
```

最後則是 `CarController.java` ，裡面的方法新增了一個參數 `Errors` ，注意要 `import` 時要找 Spring Framework 的版本，也可以把 `Errors` 改成使用 `BindingResult` 這個物件，如果沒有加上 `Errors` 或 `BindingResult` ，檢核未通過時程式碼會出錯：

```
@RestController
public class CarController {
    @Autowired
    CarService carService;

    @RequestMapping(value = "/queryAllCar", method = RequestMethod.POST)
    public List<CarEntity> queryAllCar() {
        return carService.queryAllCar();
    }

    @RequestMapping(value = "/queryError", method = RequestMethod.POST)
    public CarResponse2 queryError(@Valid @RequestBody CarRequest carRequest, Errors err) throws
        if(err.hasErrors()){
            throw new ErrorInputException();
        }
        return carService.queryCar2(carRequest);
    }
}
```

到這邊為止，基本上就完成了簡單的錯誤切面，可嘗試發送一個沒有 Manufacturer 欄位的 request，應該會拿到有錯誤訊息的結果。下面是新增一個查無資料的 Exception，可以先自行嘗試做做看：

```
|--com.example.demospringboot
   |--DemospringbootApplication.java
|--com.example.demospringboot.configuration
   |--SwaggerConfig.java
   |--RestConfiguration.java
|--com.example.demospringboot.controller
   |--CarController.java // 修改的檔案
   |--TestController.java
   |--ProductController.java
|--com.example.demospringboot.controller.advice
   |--WebExceptionHandler.java // 修改的檔案
|--com.example.demospringboot.dto
   |--CarRequest.java
   |--CarResponse.java
|--com.example.demospringboot.entity
   |--Car.java
   |--CarPK.java
|--com.example.demospringboot.exception
   |--ErrorInputException.java
   |--DataNotFoundException.java // 新增的檔案
|--com.example.demospringboot.model
   |--Product.java
|--com.example.demospringboot.repository
   |--CarRepository.java
|--com.example.demospringboot.service
   |--CarService.java // 修改的檔案
   |--ProductService.java
|--com.example.demospringboot.service.impl
   |--CarServiceImpl.java // 修改的檔案
   |--ProductServiceImpl.java
```

```java
public class DataNotFoundException extends Exception{

    private static final long serialVersionUID = 1L;

}
```

```java
public interface CarService {

    List<CarEntity> queryAllCar();

    CarResponse queryCar(CarRequest carRequest) throws DataNotFoundException;

}
```

```java
@ControllerAdvice
public class WebExceptionHandler {

    @ResponseBody
    @ExceptionHandler(ErrorInputException.class)
    public CarResponse handleErrorInputException(){
        CarResponse carResponse = new CarResponse();
        carResponse.setDatas(null);
        carResponse.setMessage("ErrorInputException");
        return carResponse;
    }

    @ResponseBody
    @ExceptionHandler(DataNotFoundException.class)
    public CarResponse2 handleDataNotFoundException(){
        CarResponse2 carResponse = new CarResponse2();
        carResponse.setDatas(null);
        carResponse.setMessage("DataNotFoundException");
        return carResponse;
    }
}
```

```java
@Service
public class CarServiceImpl implements CarService {
    @Autowired
    private CarRepository carRepository;

    @Override
    public List<CarEntity> queryAllCar() {
        return carRepository.findAll();
    }

    @Override
    public CarResponse queryCar(CarRequest carRequest) throws DataNotFoundException {
        String manufacturer = carRequest.getManufacturer();
        String type = carRequest.getType();
        List<CarEntity> list = carRepository.findByManufacturerAndType(manufacturer, type);
        if (list == null || list.isEmpty()) {
            throw new DataNotFoundException();
        }

        CarEntity carEntity = list.get(0);
        CarResponse carResponse = new CarResponse();

        Data responseInnerData = new Data();

        responseInnerData.setManufacturer(carEntity.getManufacturer());
        responseInnerData.setType(carEntity.getType());
        responseInnerData.setPrice(carEntity.getPrice());
        responseInnerData.setMinPrice(carEntity.getMinPrice());
        List<Data> datas = new ArrayList<>();
        datas.add(responseInnerData);

        carResponse.setDatas(datas);
        carResponse.setMessage("success");

        return carResponse;
    }
}
```

參考

> https://openhome.cc/Gossip/SpringGossip/AOPConcept.html