

Java

- [【反射】動態執行某類別方法](#)
- [【反射】java 如何取得目前程式碼所在的 function name](#)
- [方法寫入外部物件](#)
- [安裝OpenJDK](#)
- [泛型相關](#)
- [時間相關](#)
- [Collection 相關](#)
- [final 相關](#)
- [Java 版本相關](#)
- [Maven相關](#)
- [Optional 相關](#)
- [Stream API](#)
- [Tomcat伺服器詳解](#)
- [Bean 與 Pojo](#)
- [【Java】自Java 1.6版本以來的的主要新增功能](#)
- [【SDKMAN】安裝管理JAVA](#)

【反射】動態執行某類別方法

```
@PostMapping(value = "/test/getApi/{className}", consumes = MediaType.ALL_VALUE)
public ResponseEntity<?> getApi(
    @PathVariable String className,
    @RequestBody HashMap<String, Object> param
) throws Exception{

    HttpStatus status = HttpStatus.OK;
    Object res = null;
    HashMap classMap = new HashMap<String, String>();
    // 對應完整class
    classMap.put("TencentStreamingAPI", "com.test.api.TencentStreamingAPI");

    String fullClassName = (String)classMap.get(className);
    String func = "";

    try {
        if(fullClassName == null) throw new Exception("no className mapping fullClassName");
        if((String)param.get("func") != null){
            func = (String)param.get("func");
        }else{
            throw new Exception("no func name");
        }
    }

    Class c = Class.forName(fullClassName);
    Object ts = c.newInstance();

    switch (className){
        case "TencentStreamingAPI":
            ts = tencentStreamingAPI;
            c = ts.getClass();
            break;
        case "TencentIMAPI":
            ts = tencentIMAPI;
            c = ts.getClass();
            break;
    }

    Method[] methods = c.getMethods();
    for (Method method : methods) {
        if (method.getName().equals(func)) {

            if (method.getParameters().length == 0) {
                Method m = c.getMethod(func, null);
                res = m.invoke(ts, null);

            } else {
                ArrayList<Class<?>> parameterTypesList = new ArrayList<Class<?>>();
                ArrayList<Object> argsList = new ArrayList<Object>();

                for(Parameter parameter : method.getParameters() ){
                    String parameterName = parameter.getName();
                    Class clazz = parameter.getType();
                    Object arg = param.get(parameterName);
                    if(arg == null) throw new Exception("no such parameter: "+ parameterName);
                    parameterTypesList.add(clazz);
                    argsList.add(arg);
                }

                // Method m = c.getMethod(func, String.class);
                Method m = c.getMethod(func, parameterTypesList.toArray(new Class[parameterTypesList.size()]));
                res = m.invoke(ts, argsList.toArray(new Object[argsList.size()]));
            }
            break;
        }
    }
}
```

```

    }

    }

    } catch (ClassNotFoundException e) {
        logger.error(e.getMessage());
        throw new Exception("no such class: " + className);
    } catch (NoSuchMethodException e) {
        logger.error(e.getMessage());
        throw new Exception("no such metod:" + func);
    } catch (IllegalArgumentException e) {
        logger.error(e.getMessage());
        throw new Exception("illegal argument :" + func);
    } catch (Exception e) {

        //取得目前執行方法名稱
        String methodName = "[noGetMethodName]";
        Method method = this.getClass().getEnclosingMethod();
        if(method != null){
            methodName = method.getName();
        }else{
            StackTraceElement[] stackTrace = Thread.currentThread().getStackTrace();
            for (StackTraceElement stackTraceElement : stackTrace) {
                methodName = stackTraceElement.getMethodName();
                if (!methodName.equals("getStackTrace") &&
                    !methodName.equals("getMethodName") &&
                    !methodName.equals("main")) {
                    break;
                }
            }
        }

        logger.error(this.getClass().toString()+" -> "+
            methodName + ": " + e.getMessage());
        status = HttpStatus.FORBIDDEN;
    }

    HttpHeaders headers = new HttpHeaders();
    headers.setContentType(MediaType.APPLICATION_JSON);
    String resultStr = "{}";

    if(res instanceof JSONObject){
        resultStr = ((JSONObject) res).toString();
    }else if(res instanceof JSONArray){
        resultStr = ((JSONArray) res).toString();
    }else if(res instanceof String){
        resultStr = (String) res;
    }

    // logger.debug("func name is {}",func);
    // logger.debug("func res type is {}",res.getClass().getName());
    return new ResponseEntity<>(resultStr, headers, status);
}

```

```

public class TencentStreamingAPI {
    public JSONObject describeStreamPlayInfoList(String streamId,String starttime, String endtime){
        .....
    }
}

```

```

// 呼叫
http://localhost:8080/api/test/getApi/TencentStreamingAPI

/** json 參數
func: 執行方法名稱
其他帶入同名參數
*/
{

```

```
"func": "describeStreamPlayInfoList",  
"streamId": "AABBCCDD",  
"starttime": "2023-03-29 13:18:00",  
"endtime": "2023-03-29 13:18:00"
```

```
}
```

【反射】java 如何取得目前程式碼所在的 function name

```
public class MyClass {
    public static void main(String[] args) {
        String methodName = new Object() {
            .getClass()
            .getEnclosingMethod()
            .getName();

            System.out.println("目前所在的方法名稱為：" + methodName);
        }
    }
}
```

這段程式碼使用了匿名內部類別的方式來取得目前所在的方法名稱。在這個匿名內部類別中，呼叫 `getClass()` 方法可以取得目前物件的類別，接著呼叫 `getEnclosingMethod()` 方法可以取得該物件所在的方法，最後呼叫 `getName()` 方法可以取得該方法的名稱。

需要注意的是，由於這個方法使用了匿名內部類別，所以每次呼叫都會產生一個新的物件，可能會對效能造成一些影響。

如果 `getEnclosingMethod()` 方法回傳 `null`，表示目前的程式碼不是定義在一個方法裡面，而是定義在類別的初始化區塊（initializer block）或建構子（constructor）中，或者是在類別的靜態初始化區塊（static initializer block）或靜態建構子（static constructor）中。

在這種情況下，可以使用 `Thread.currentThread().getStackTrace()` 方法來取得當前執行緒的呼叫堆疊（call stack），然後從呼叫堆疊中找到第一個方法的名稱，這個方法的名稱就是目前所在的方法名稱。

以下是一個示範程式碼，可以在這種情況下取得目前所在的方法名稱：

```
public class MyClass {
    public MyClass() {
        String methodName = getMethodName();
        System.out.println("目前所在的方法名稱為：" + methodName);
    }

    static {
        String methodName = getMethodName();
        System.out.println("目前所在的方法名稱為：" + methodName);
    }

    {
        String methodName = getMethodName();
        System.out.println("目前所在的方法名稱為：" + methodName);
    }

    public static void main(String[] args) {
        String methodName = getMethodName();
        System.out.println("目前所在的方法名稱為：" + methodName);
    }

    private static String getMethodName() {
        StackTraceElement[] stackTrace = Thread.currentThread().getStackTrace();
        for (StackTraceElement stackTraceElement : stackTrace) {
            String methodName = stackTraceElement.getMethodName();
            if (!methodName.equals("getStackTrace") &&
                !methodName.equals("getMethodName") &&
                !methodName.equals("main")) {
                return methodName;
            }
        }
        return null;
    }
}
```

這個程式碼定義了幾個方法和初始化區塊，在每個區塊中都呼叫了 `getMethodName()` 方法來取得目前所在的方法名稱。在 `getMethodName()` 方法中，首先取得目前執行緒的呼叫堆疊，然後從呼叫堆疊中找到第一個不是 `getStackTrace()`、`getMethodName()` 或 `main()` 的方法名稱，這個方法的名稱就是目前所在的方法名稱。

方法寫入外部物件

在 Java 中，方法參數的傳遞方式是按值傳遞，也就是說，方法內部參數的修改不會影響到原始變量的值。因此，在 Java 中無法直接通過方法回傳外部變量。

不過，可以通過一些技巧實現類似於按引用傳遞的效果。其中一種方法是將要回傳的變量封裝成一個對象，並將該對象作為方法的參數傳遞過去。這樣，方法內部對該對象的修改會影響到原始變量的值。

例如，假設有一個要回傳的變量 `int x`，可以將其封裝成一個對象 `IntHolder`，該對象包含一個整數屬性 `value`，並提供一個方法 `getValue()` 和 `setValue(int)` 分別用於取和設置屬性的值。然後，在調用方法時，將一個 `IntHolder` 對象作為參數傳遞過去。方法內部就可以通過 `IntHolder` 對象修改原始變量的值。

下面是一個示例代碼：

```
public class IntHolder {
    private int value;

    public IntHolder(int value) {
        this.value = value;
    }

    public int getValue() {
        return value;
    }

    public void setValue(int value) {
        this.value = value;
    }
}

public class Main {
    public static void main(String[] args) {
        IntHolder xHolder = new IntHolder(10);
        System.out.println("Before method call: " + xHolder.getValue());

        // 調用方法，將 xHolder 作為參數傳遞過去
        modifyIntHolder(xHolder);

        System.out.println("After method call: " + xHolder.getValue());
    }

    public static void modifyIntHolder(IntHolder holder) {
        // 修改 holder 的值，會影響到 xHolder
        holder.setValue(20);
    }
}
```

在上面的示例代碼中，`IntHolder` 對象 `xHolder` 的值被初始化為 10，然後被傳遞給 `modifyIntHolder` 方法。在方法內部，`holder` 對象的值被修改為 20。由於 `xHolder` 和 `holder` 引用的是同一個對象，因此 `xHolder` 的值也被修改為 20。最終輸出結果：

```
Before method call: 10
After method call: 20
```

安裝OpenJDK

[Chapter 2. Installing OpenJDK 8 on Red Hat Enterprise Linux OpenJDK 8 | Red Hat Customer Portal](#)

```
$ sudo yum install java-1.8.0-openjdk-devel
```

泛型相關

Java 泛型是一种支持参数化类型的机制，它可以使代码更加通用和安全。通过使用泛型，我可以在代码中使用占位符类型（如 T、E 等），然后在实际使用时再指定具体类型。这样就可以使代码更加灵活，可以避免类型错误和运行异常。

Java 泛型的基本用法是使用尖括号 "<>" 括起的一个或多个类型参数，这些类型参数可以用于类、接口、方法的声明中。例如：

```
public class MyClass<T> {
    private T data;

    public MyClass(T data) {
        this.data = data;
    }

    public T getData() {
        return data;
    }
}
```

在上面的代码中，我使用了一个类型参数 T，这个参数可以用于 MyClass 类中的任何地方，包括类的字段、方法参数、方法返回值等等。

而<?>和<T>的区别在于，<?>是一种无限制通配符类型，表示可以匹配任何类型，而<T>是一个类型参数，表示在使用时需要指定具体的类型。例如：

```
List<?> list = new ArrayList<>();
List<String> strList = new ArrayList<>();
list = strList; // 合法，因为 list 可以匹配任何类型
```

```
public <T> T getValue(T[] array, int index) {
    return array[index];
}

String[] strArray = {"hello", "world"};
String str = getValue(strArray, 0); // 合法，因为在使用 getValue 指定了类型参数 String
```

下面是一个使用无限制通配符类型（<?>）的例子，我定义了一个方法，可以接受任何类型的 List，并打印出其中的元素：

```
public static void printList(List<?> list) {
    for (Object element : list) {
        System.out.println(element);
    }
}
```

这个方法参数列表中使用了无限制通配符类型（<?>），表示可以匹配任何类型的 List。在方法内部，我使用了一个 for-each 循环遍历 list，并打印出其中的元素。

使用这个方法，我可以传入任何类型的 List，例如：

```
List<Integer> intList = Arrays.asList(1, 2, 3);
List<String> strList = Arrays.asList("hello", "world");
List<Object> objList = Arrays.asList(1, "hello", true);

printList(intList); // 输出 1, 2, 3
printList(strList); // 输出 hello, world
printList(objList); // 输出 1, hello, true
```

如果我上面的例子中的方法参数从无限制通配符类型（<?>）改为类型参数（<T>），那么代码会变成下面这样：

```
public static <T> void printList(List<T> list) {
    for (T element : list) {
        System.out.println(element);
    }
}
```

这个方法参数列表中使用了类型参数（<T>），表示在使用时需要指定 List 中元素的具体类型。在方法内部，我使用了一个 for-each 循环

遍历 list，并打印出其中的元素。

使用这个方法，我需要指定 List 中元素的具体类型，例如：

```
List<Integer> intList = Arrays.asList(1, 2, 3);
List<String> strList = Arrays.asList("hello", "world");
List<Object> objList = Arrays.asList(1, "hello", true);

printList(intList); // 输出 1, 2, 3
printList(strList); // 输出 hello, world
printList(objList); // 报错，因为 objList 中的元素类型不是 Object
```

在使用时，我需要指定 List 中元素的具体类型，例如使用 `printList(Integer)` 或 `printList(String)`，这样才能保证参类型的匹配，否则会报错。

在 Java 泛型中，类型参的名是可以任意指定的，通常使用以下的约定：

- T：表示任何类型。
- E：表示集合中的元素类型。
- K：表示映射中的键类型。
- V：表示映射中的值类型。
- N：表示数字类型。
- S、U、V 等：表示其他任意类型。

因此，使用 `<T>` 或 `<E>` 在语义上是等价的，只是命名不同而已。

除了以上提到的常用类型参名外，开发者也可以根据实际情况自行定义类型参的名，只需要遵循命名规范即可。不过，为了代码可读性和可维护性，建议使用常见的类型参名。

需要注意的是，不同的类型参名并非有实质上的差异，只是在语义上有所不同，因此我们类型参名要根据实际情况和需求进行，一个能更好地表达自己的代码意图的命名。

一个回传任意型别的范例

```
public static <T> T anyTypeMethod(T arg) {
    if (arg == null) {
        // 返回空对象
        return (T) new Object();
    }
    // 方法
    return arg;
}
```

在这个方法中，我首先检查参是否 `null`。如果参 `null`，返回一个空对象，否则执行方法的操作，参原封不动地返回。

需要注意的是，在这个方法中，我新建了一个新的 `Object` 对象并将其泛型类型，这是为了避免空对象返回类型不匹配的。因为 Java 中的泛型是在编译时擦除的，如果我直接返回一个 `null`，编译器无法确定返回值的类型，可能会导致编译错误或运行异常。

使用这个方法，我可以传入任何类型的参，并取返回值，如果传入参 `null`，返回一个空对象，例如：

```
Integer intValue = anyTypeMethod(123);
String strValue = anyTypeMethod("hello");
Object objValue = anyTypeMethod(new Object());
Object nullValue = anyTypeMethod(null);

System.out.println(intValue); // 输出 123
System.out.println(strValue); // 输出 "hello"
System.out.println(objValue); // 输出 Object 对象的 toString() 值
System.out.println(nullValue); // 输出 "java.lang.Object@hashCode"
```

在这个示例中，我分别传入了一个整数、一个字符串、一个对象和一个 `null`，然后取了相应的返回值。传入参 `null` 时，返回一个空对象，其类型为 `Object`。

時間相關

Java提供了許多處理日期和時間的類和方法。以下是一些基本的日期操作：

java.util.Date

創建日期對象

```
Date currentDate = new Date();
```

java.text.SimpleDateFormat

格式化日期

可以使用 `java.text.SimpleDateFormat` 類將日期格式化為特定的字符串格式。例如：

```
SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");  
String dateString = dateFormat.format(currentDate);
```

解析日期字符串

可以使用 `SimpleDateFormat` 類將字符串轉換為日期對象。例如：

```
String dateString = "2023-03-23";  
SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");  
Date date = dateFormat.parse(dateString);
```

java.util.Calendar

操作日期

可以使用 `java.util.Calendar` 類進行日期計算，例如添加或減去一定的時間。例如：

這將當前日期添加7天，得到一個未來的日期。

這僅是Java中處理日期的基本操作。還有許多其他的類和方法可以用來處理日期和時間。

```
Calendar calendar = Calendar.getInstance();  
calendar.setTime(currentDate);  
calendar.add(Calendar.DAY_OF_MONTH, 7);  
Date futureDate = calendar.getTime();
```

找出前五分鐘的日期

```
// 創建一個 Calendar 實例  
Calendar calendar = Calendar.getInstance();  
  
// 從當前日期和時間中減去 5 分鐘  
calendar.add(Calendar.MINUTE, -5);  
  
// 得到剛才減去的日期時間  
Date fiveMinutesAgo = calendar.getTime();  
  
// 格式化日期時間，如果需要的話  
SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");  
String fiveMinutesAgoString = dateFormat.format(fiveMinutesAgo);  
  
// 輸出結果  
System.out.println("前五分鐘的日期時間為：" + fiveMinutesAgoString);
```

寫一個各種常用時間日期字串轉換成日期的方法

需要注意的是，轉換時需要確保時間文字的格式與使用的 `SimpleDateFormat` 格式一致，否則會導致轉換失敗。如果轉換失敗，則會拋出 `ParseException` 異常。

```
public static Date parseDateTimeString(String dateTimeString) {
    // 定義各種常用的時間日期格式
    Map<String, String> dateTimeFormats = new HashMap<>();
    dateTimeFormats.put("yyyy-MM-dd", "\\d{4}-\\d{2}-\\d{2}$");
    dateTimeFormats.put("yyyy-MM-dd'T'HH:mm:ss", "\\d{4}-\\d{2}-\\d{2}T\\d{2}:\\d{2}:\\d{2}$");
    dateTimeFormats.put("yyyy-MM-dd'T'HH:mm:ss.SSS", "\\d{4}-\\d{2}-\\d{2}T\\d{2}:\\d{2}:\\d{2}.\\d{3}$");
    dateTimeFormats.put("yyyy-MM-dd'T'HH:mm:ss'Z'", "\\d{4}-\\d{2}-\\d{2}T\\d{2}:\\d{2}:\\d{2}Z$");
    dateTimeFormats.put("yyyy-MM-dd'T'HH:mm:ss.SSS'Z'", "\\d{4}-\\d{2}-\\d{2}T\\d{2}:\\d{2}:\\d{2}.\\d{3}Z$");
    dateTimeFormats.put("EEE MMM dd HH:mm:ss zzz yyyy", "\\w{3} \\w{3} \\d{2} \\d{2}:\\d{2}:\\d{2} \\w{3} \\d{4}$");

    // 使用各種格式進行轉換，直到轉換成功為止
    for (Map.Entry<String, String> entry : dateTimeFormats.entrySet()) {
        String format = entry.getKey();
        String regex = entry.getValue();
        if (dateTimeString.matches(regex)) {
            try {
                SimpleDateFormat dateFormat = new SimpleDateFormat(format, Locale.ENGLISH);
                Date date = dateFormat.parse(dateTimeString);
                return date;
            } catch (ParseException e) {
                // 轉換失敗，繼續使用下一個格式進行轉換
            }
        }
    }

    // 如果所有格式都無法轉換，則返回null
    return null;
}
```

好用時間Lib

在Java中，有許多好用的時間相關的函式庫，以下是一些常見且經常使用的時間函式庫推薦：

1. Joda-Time：Joda-Time是一個廣泛使用的Java日期和時間庫，它提供了比Java內置日期時間類庫更多的功能。Joda-Time包含各種類型的日期，時間和時間間隔，並提供了許多便捷的方法來進行日期計算和格式化。
2. java.time：Java 8引入了新的時間API，稱為java.time。它提供了新的日期時間類型，例如LocalDate，LocalTime和ZonedDateTime等，並提供了許多便於使用的方法來進行日期計算和格式化。
3. Date4j：Date4j是一個輕量級的日期和時間庫，它具有方便的方法和較小的庫大小。Date4j支持基本日期和時間計算，例如計算兩個日期之間的天數，以及格式化日期時間。
4. PrettyTime：PrettyTime是一個Java函式庫，用於將日期和時間轉換為易於閱讀的相對時間（例如“3天前”）。它支持多種語言和格式，可以輕鬆地集成到Java應用程序中。

Collection 相關

建立Map

```
// Java 9 以上
Map<String, Integer> map = Map.of("apple", 1, "banana", 2, "orange", 3);
System.out.println(map); // {banana=2, orange=3, apple=1}
```

```
// java 通用
Map<String, Integer> map = new HashMap<>();
map.put("apple", 1);
map.put("banana", 2);
map.put("orange", 3);
System.out.println(map); // {banana=2, orange=3, apple=1}
```

final 相關

`final` 是 Java 中的一个关键字，用于修变量、方法和类，表示其值或不可改变。使用 `final` 可以提高程序的安全性、可靠性和性能，通常在以下情况下使用：

1. 声明不可变变量

可以使用 `final` 关键字声明不可变变量，一旦声明后，常量的值就不能再被修改。这样可以避免程序中的常量被修改，提高程序的健壮性。

```
public static final int MAX_NUM = 100;
```

2. 防止继承和重载

可以使用 `final` 关键字修一个类，表示类不可被继承。同时，也可以修一个方法，表示方法不可被子类重载。这样可以避免在继承和多态的过程中产生不可知的行，提高程序的稳定性和可维护性。

```
public final class MyClass {  
    // 类定义  
}  
  
public class MySubClass extends MyClass {  
    // 错误，无法继承 final 类  
}  
  
public class MyClass {  
    public final void myMethod() {  
        // 方法定义  
    }  
}  
  
public class MySubClass extends MyClass {  
    public void myMethod() {  
        // 错误，无法重载 final 方法  
    }  
}
```

3. 确保线程安全

可以使用 `final` 关键字修一个变量，表示变量只能被赋值一次，并且在多线程环境下是线程安全的。这样可以避免在多线程环境下出现竞争和数据不一致的问题，提高程序的并发性和稳定性。

```
public class MyClass {  
    private final int num;  
  
    public MyClass(int num) {  
        this.num = num;  
    }  
  
    public int getNum() {  
        return num;  
    }  
}
```

4. 优化代码性能

可以使用 `final` 关键字修一个变量，表示变量的值在值后不再改变，并且变量在运行中会被优化成常量，从而提高程序的性能。

```
public void myMethod(final int num) {  
    // 方法定义  
}
```

在上述示例中，我使用 `final` 关键字修了常量、类、方法和变量，分别达到了不同的目的。使用 `final` 可以提高程序的健壮性、稳定性、可靠性和性能，是 Java 编程中的重要特性之一。

Java 版本相關

1. JDK 1.0 (1996年1月23日)
2. JDK 1.1 (1997年2月19日)
3. J2SE 1.2 (1998年12月8日)
4. J2SE 1.3 (2000年5月8日)
5. J2SE 1.4 (2002年2月13日)
6. J2SE 5.0 (2004年9月30日)
7. Java SE 6 (2006年12月11日)
8. Java SE 7 (2011年7月28日)
9. Java SE 8 (2014年3月18日)
10. Java SE 9 (2017年9月21日)
11. Java SE 10 (2018年3月20日)
12. Java SE 11 (2018年9月25日)
13. Java SE 12 (2019年3月19日)
14. Java SE 13 (2019年9月17日)
15. Java SE 14 (2020年3月17日)
16. Java SE 15 (2020年9月15日)
17. Java SE 16 (2021年3月16日)
18. Java SE 17 (2021年9月14日)

以下是自Java 1.6版本以來的的主要新增功能：

1. Java SE 6 (2006年12月)：
 - 支持JDBC 4.0規範
 - 新增JAX-WS 2.0 API
 - 新增JAXB 2.0 API
 - 新增Java Compiler API
2. Java SE 7 (2011年7月)：
 - 新增Diamond操作符 (类型推斷)
 - 支持多个异常捕獲
 - 新增字符串在switch句中的使用
 - 新增可变參數的try句
 - 新增NIO 2.0 API
3. Java SE 8 (2014年3月)：
 - 新增Lambda表达式
 - 新增Date/Time API
 - 新增Nashorn JavaScript引擎
 - 新增Stream API
 - 新增Type Annotations
4. Java SE 9 (2017年9月)：
 - 新增模块系统 (Java Platform Module System)
 - 新增私有接口方法
 - 新增改寫的Javadoc
 - 新增響應式流 API
 - 新增JShell REPL (Read-Eval-Print Loop)
5. Java SE 10 (2018年3月)：
 - 新增局部变量类型推斷
 - 新增用类数据共享
 - 新增JEP 286：本地变量类型推斷
 - 新增JEP 322：Time-Based Release Versioning
6. Java SE 11 (2018年9月)：
 - 新增var在Lambda表达式中的使用
 - 新增嵌套控制
 - 移除Java EE和CORBA模块
 - 新增ZGC垃圾回收器
7. Java SE 12 (2019年3月)：
 - 新增JEP 189：Shenandoah垃圾回收器
 - 新增JEP 325：Switch表达式 (准化)
8. Java SE 13 (2019年9月)：
 - 新增文本块
 - 新增Switch表达式的增强功能
 - 新增JFR事件流
9. Java SE 14 (2020年3月)：
 - 新增記錄 (Records)
 - 新增Switch表达式的增强功能
 - 新增Pattern匹配实例of
10. Java SE 15 (2020年9月)：
 - 新增Sealed类和接口
 - 新增Hidden类和接口
 - 新增Pattern匹配实例of的增强功能
11. Java SE 16 (2021年3月)：
 - 新增Records的增强功能
 - 新增强制类型模式匹配

- 新增JEP 394：模式匹配 instanceof的增强功能

12. Java SE 17（2021年9月）：

- 新增Sealed类和接口的增强功能
- 新增Switch表达式的增强功能
- 新增Pattern匹配实例of的增强功能
- 新增JEP 403：Strong

Maven相關

- [Mave教學 | Maven 初學者中文教程 \(kentyeh.github.io\)](https://kentyeh.github.io)
- [使用 Docker 執行 Maven - mvn compile \(puritys.me\)](https://puritys.me)

```
FROM openjdk:8-jdk
```

```
ARG MAVEN_VERSION=3.3.9
```

```
ARG USER_HOME_DIR="/root"
```

```
RUN apt-get update
```

```
RUN apt-get install build-essential -y
```

```
RUN ln -sf /usr/bin/make /usr/bin/gmake
```

```
RUN mkdir -p /usr/share/maven /usr/share/maven/ref \
```

```
&& curl -fsSL http://apache.osuosl.org/maven/maven-3/$MAVEN_VERSION/binaries/apache-maven-$MAVEN_VERSION-bin.tar.gz \
```

```
| tar -xzc /usr/share/maven --strip-components=1 \
```

```
&& ln -s /usr/share/maven/bin/mvn /usr/bin/mvn
```

```
ENV MAVEN_HOME /usr/share/maven
```

```
ENV MAVEN_CONFIG "$USER_HOME_DIR/.m2"
```

```
VOLUME ["/usr/src/mymaven", "/root/.m2"]
```

Optional 相關

Optional是Java SE 8中的一个新特性，它提供了一种优雅的方式处理可能空的值，避免了空指针异常。

Optional类提供了以下常用方法：

- of()：创建一个包含指定非空值的Optional对象。如果传入null，会抛出NullPointerException异常。例如：

```
Optional<String> optional = Optional.of("hello");
```

- empty()：创建一个空的Optional对象。例如：

```
Optional<String> optional = Optional.empty();
```

- isPresent()：判断Optional对象是否包含非空值。例如：

```
Optional<String> optional = Optional.of("hello");
if (optional.isPresent()) {
    System.out.println("value is present: " + optional.get());
}
```

- get()：取Optional对象中的值，如果对象空抛出NoSuchElementException异常。例如：

```
Optional<String> optional = Optional.of("hello");
String value = optional.get();
System.out.println("value: " + value);
```

- orElse()：取Optional对象中的值，如果对象空返回指定的默认值。例如：

```
Optional<String> optional = Optional.empty();
String value = optional.orElse("default");
System.out.println("value: " + value);
```

- map()：对Optional对象中的值进行映射操作，返回一个新的Optional对象。例如：

```
Optional<String> optional = Optional.of("hello");
Optional<Integer> result = optional.map(s -> s.length());
System.out.println("result: " + result.get());
```

- filter()：对Optional对象中的值进行过滤操作，返回一个新的Optional对象。例如：

```
Optional<String> optional = Optional.of("hello");
Optional<String> result = optional.filter(s -> s.startsWith("h"));
System.out.println("result: " + result.get());
```

需要注意的是，Optional对象的操作通常采用链式调用的方式，如下所示：

```
Optional<String> optional = Optional.of("hello");
String result = optional.map(s -> s.toUpperCase())
    .orElse("default");
System.out.println("result: " + result);
```

在上面的示例中，我首先将字符串"hello"包装成Optional对象，然后使用map()方法将字符串转换成大写，最后使用orElse()方法取处理结果，如果对象空返回指定的默认值"default"。

optional.isPresent() 方法用于检查 Optional 对象是否包含非空值，如果对象中包含了一个非空的值，返回 true，否则返回 false。

如果使用 Optional.of() 方法创建一个 Optional 对象，传入了一个空引用，会抛出 NullPointerException 异常，而不是返回一个包含空值的 Optional 对象。因此，如果使用 Optional.of() 方法创建的对象中包含了一个空引用，调用 optional.isPresent() 方法会返回 false，而不是检查空字符串。

如果需要检查一个空字符串，可以使用 Optional.ofNullable() 方法创建一个 Optional 对象，例如：

```
String str = "";
```

```
Optional<String> optional = Optional.ofNullable(str);
if (optional.isPresent()) {
    System.out.println("str is not empty");
} else {
    System.out.println("str is empty");
}
```

在上面的示例中，我首先使用一个空字符串构建一个 `Optional` 对象，然后使用 `optional.isPresent()` 方法检查字符串是否空。由于字符串空，因此用 `optional.isPresent()` 方法会返回 `false`，所以最终输出结果 `"str is empty"`。

Stream API

Stream API是Java 8中引入的一种新的API，它提供了一种用流的方式对集合（Collection）和数组（Array）进行处理的方法，它的目的是使得代码更加简洁、易读和易于维护。

Stream API的主要优点包括：

1. 方便：使用Stream API可以方便地处理集合或数组中的元素，无需手动写循环语句。
2. 简洁：使用Stream API可以用更少的代码完成相同的任务，因为Stream API提供了一系列的函数式操作，例如过滤、映射、排序等，这些操作可以链式调用，从而使代码更加简洁易懂。
3. 高效：使用Stream API可以提高代码的执行效率，因为Stream API使用了惰性求值的方式，只有在需要时才会执行操作，避免了不必要的计算和内存。

Stream API的核心概念包括：

1. 流（Stream）：Stream是一种序列化的对象集合，它支持顺序和并行两种操作模式，并且可以在不修改原始集合的情况下进行各种操作。
2. 操作（Operations）：Stream提供了一系列的函数式操作，例如过滤、映射、排序等，这些操作可以链式调用，形成操作链。
3. 终止操作（Terminal Operations）：最后必须使用终止操作，例如forEach、count、reduce等，才能得到操作的结果。

Stream API的出现使得Java在集合处理方面更加强大和灵活，也使得Java代码变得更加简洁、易读和易于维护。

使用Stream API，可以通过一系列的中间操作对集合或数组进行处理，最后再通过终止操作获取结果。下面是一些使用Stream API的示例：

1. 集合过滤：

假设有一个List<String>类型的集合，在需要找出其中长度大于3的元素，可以使用如下代码：

```
List<String> list = Arrays.asList("apple", "banana", "cat", "dog", "elephant");
List<String> filteredList = list.stream()
    .filter(str -> str.length() > 3)
    .collect(Collectors.toList());
System.out.println(filteredList); // [apple, banana, elephant]
```

在上面的代码中，我首先用stream()方法将集合转换成Stream对象，然后使用filter操作找出长度大于3的元素，最后使用collect操作将结果转换成List类型。

2. 集合映射：

假设有一个List<Integer>类型的集合，在需要将其中每个元素乘以2，可以使用如下代码：

```
List<Integer> list = Arrays.asList(1, 2, 3, 4, 5);
List<Integer> mappedList = list.stream()
    .map(num -> num * 2)
    .collect(Collectors.toList());
System.out.println(mappedList); // [2, 4, 6, 8, 10]
```

在上面的代码中，我首先用stream()方法将集合转换成Stream对象，然后使用map操作对每个元素进行乘以2的操作，最后使用collect操作将结果转换成List类型。

3. 数组排序：

假设有一个int[]类型的数组，在需要将其中的元素按照从小到大的顺序进行排序，可以使用如下代码：

```
List<Integer> list = Arrays.asList(1, 2, 3, 4, 5);
List<Integer> mappedList = list.stream()
    .map(num -> num * 2)
    .collect(Collectors.toList());
System.out.println(mappedList); // [2, 4, 6, 8, 10]
```

在上面的代码中，我首先用Arrays.stream()方法将数组转换成IntStream对象，然后使用sorted操作对元素进行排序，最后使用toArray操作将结果转换成int[]类型。

Tomcat伺服器詳解

Tomcat伺服器理解詳解 | 程式前沿 (codertw.com)

Bean 與 Pojo

Bean、POJO 是 Java 領域中常用的術語，兩者均是軟體開發中常用的類別設計方式。

Bean 是一個可重複使用、可移植的軟體元件，具有以下特徵：

- 實體化後需要經過初始化，可以透過 setter 或 constructor 注入屬性值
- 必須具有一個無參數的預設建構子
- 可以實作 Serializable 介面以支援序列化
- 常見的 Bean 類型包括 Spring Framework 中的 Bean、JavaBean 等

POJO (Plain Old Java Object) 是一個純粹的 Java 物件，沒有任何限制或框架要求，具有以下特徵：

- 無需繼承特定的類別或實作特定的介面
- 可以包含任意數量和任意型別的屬性和方法
- 可以實作 Serializable 介面以支援序列化

簡單來說，Bean 是 Spring 框架中的一種概念，它是一個有生命週期、可管理的物件，必須符合特定的設計要求。而 POJO 則是一個更通用的概念，它沒有特定的框架或設計要求，可以隨意設計，但這也代表它可能缺乏特定框架的功能和支援。

在 Spring 框架中，Bean 通常被用作被 Spring 管理的物件，例如 Service、Controller 等，而 POJO 則通常用來表示簡單的 Java 物件。

【Java】自Java 1.6版本以来的主要新增功能

以下是自Java 1.6版本以来的主要新增功能：

1. Java SE 6（2006年12月）：
 - 支持JDBC 4.0规范
 - 新增JAX-WS 2.0 API
 - 新增JAXB 2.0 API
 - 新增Java Compiler API
2. Java SE 7（2011年7月）：
 - 新增Diamond操作符（类型推导）
 - 支持多个异常捕获
 - 新增字符串在switch语句中的使用
 - 新增可变参数的try语句
 - 新增NIO 2.0 API
3. Java SE 8（2014年3月）：
 - 新增Lambda表达式
 - 新增Date/Time API
 - 新增Nashorn JavaScript引擎
 - 新增Stream API
 - 新增Type Annotations
4. Java SE 9（2017年9月）：
 - 新增模块系统（Java Platform Module System）
 - 新增私有接口方法
 - 新增改进的Javadoc
 - 新增响应式流 API
 - 新增JShell REPL（Read-Eval-Print Loop）
5. Java SE 10（2018年3月）：
 - 新增局部变量类型推导
 - 新增用类数据共享
 - 新增JEP 286：本地变量类型推导
 - 新增JEP 322：Time-Based Release Versioning
6. Java SE 11（2018年9月）：
 - 新增var在Lambda表达式中的使用
 - 新增嵌套类控制
 - 移除Java EE和CORBA模块
 - 新增ZGC垃圾回收器
7. Java SE 12（2019年3月）：
 - 新增JEP 189：Shenandoah垃圾回收器
 - 新增JEP 325：Switch表达式（规范化）
8. Java SE 13（2019年9月）：
 - 新增文本块
 - 新增Switch表达式的增强功能
 - 新增JFR事件流
9. Java SE 14（2020年3月）：
 - 新增记录（Records）
 - 新增Switch表达式的增强功能
 - 新增Pattern匹配实例of
10. Java SE 15（2020年9月）：
 - 新增Sealed类和接口
 - 新增Hidden类和接口
 - 新增Pattern匹配实例of的增强功能
11. Java SE 16（2021年3月）：
 - 新增Records的增强功能
 - 新增强制类型模式匹配
 - 新增JEP 394：模式匹配 instanceof的增强功能
12. Java SE 17（2021年9月）：
 - 新增Sealed类和接口的增强功能
 - 新增Switch表达式的增强功能
 - 新增Pattern匹配实例of的增强功能
 - 新增JEP 403：Strong

【SDKMAN】安裝管理JAVA

來源: <https://blog.miniasp.com/post/2022/09/17/Useful-tool-SDKMAN>

透過 SDKMAN 安裝JAVA

```
#安裝 SDKMAN
curl -s "https://get.sdkman.io" | bash
#首次手動載入 SDKMAN 工具 (預設安裝過程已經設定好 ~/.bashrc 啟動定義檔)
source ~/.bashrc
#檢查 SDKMAN 版本
sdk version
```

安裝 OpenJDK 17

```
#先列出所有 SDKMAN 中支援的 Java 版本
sdk ls java
```

Available Java Versions for macOS 64bit

Vendor	Use	Version	Dist	Status	Identifier
Corretto		20.0.2	amzn		20.0.2-amzn
		20.0.1	amzn		20.0.1-amzn
		17.0.8	amzn		17.0.8-amzn
		17.0.7	amzn		17.0.7-amzn
		11.0.20	amzn		11.0.20-amzn
		11.0.19	amzn		11.0.19-amzn
		8.0.382	amzn		8.0.382-amzn
		8.0.372	amzn		8.0.372-amzn
Gluon		22.1.0.1.r17	gln		22.1.0.1.r17-gln
		22.1.0.1.r11	gln		22.1.0.1.r11-gln
		22.0.0.3.r17	gln		22.0.0.3.r17-gln
		22.0.0.3.r11	gln		22.0.0.3.r11-gln
GraalVM CE		20.0.2	graalce		20.0.2-graalce
		20.0.1	graalce		20.0.1-graalce
		17.0.8	graalce		17.0.8-graalce
		17.0.7	graalce		17.0.7-graalce
GraalVM Oracle		20.0.2	graal		20.0.2-graal
		20.0.1	graal		20.0.1-graal
		17.0.8	graal		17.0.8-graal
		17.0.7	graal		17.0.7-graal
Java.net		22.ea.9	open		22.ea.9-open
		22.ea.8	open		22.ea.8-open
		22.ea.7	open		22.ea.7-open
		22.ea.6	open		22.ea.6-open
		22.ea.5	open		22.ea.5-open
		22.ea.4	open		22.ea.4-open
		22.ea.3	open		22.ea.3-open
		21.ea.34	open		21.ea.34-open
		21.ea.33	open		21.ea.33-open
		21.ea.32	open		21.ea.32-open
		21.ea.31	open		21.ea.31-open
		21.ea.30	open		21.ea.30-open
		21.ea.29	open		21.ea.29-open
		21.ea.28	open		21.ea.28-open
		20.0.2	open		20.0.2-open
		19.ea.1.pma	open		19.ea.1.pma-open
JetBrains		17.0.7	jbr		17.0.7-jbr
		11.0.14.1	jbr		11.0.14.1-jbr

Omit Identifier to install default version 17.0.8-tem:

```
$ sdk install java
```

Use TAB completion to discover available versions

```
$ sdk install java [TAB]
Or install a specific version by Identifier:
$ sdk install java 17.0.8-tem
Hit Q to exit this list view
=====
```

安裝 JetBranins 17.0.7

```
sdk install java 17.0.7-jbr
```

透過 SDKMAN 管理多個 JDK 版本

```
#先安裝 11.0.14.1-jbr 版本
sdk install java 11.0.14.1-jbr

#####

Downloading: java 11.0.14.1-jbr

In progress...

#####
100.0%#####
100.0%

Repackaging Java 11.0.14.1-jbr...

Done repackaging...
Cleaning up residual files...

Installing: java 11.0.14.1-jbr
Done installing!
#選 n 不要變成預設版本
Do you want java 11.0.14.1-jbr to be set as default? (Y/n): n
```

```
#在目前shell 切換版本
sdk use java 11.0.14.1-jbr

#如果要設為預設版本
sdk default java 11.0.14.1-jbr

#確認版本可以透過 sdk current java 或 java -version 確認版本
```

透過 SDKMAN 管理更新、升級、移除

```
#查看是否有更新版本
sdk update

#升級版本
sdk upgrade

# 移除特定版本
# 記得將預設版本切換到現有版本
sdk default java 17.0.4.1-ms
sdk uninstall java 8.0.345-zulu
```

更新 SDKMAN 到最新版

```
sdk selfupdate
```

快速安裝springboot

```
curl -s "https://get.sdkman.io" | bash  
source ~/.bashrc
```

```
sdk install java 17.0.4.1-ms  
sdk install maven  
sdk install springboot
```