生成用於資訊網框架的 Python 服務元件
Generating Python-Based Service Components for Web Frameworks

李允中	陳耀新	余治杰
Jonathan Lee	Yao-Hsin Chen	Jhih-Jie Yu
國立台灣大學資訊工程學系	國立台灣大學資訊工程學系	國立台灣大學資訊工程學系
jlee@csie.ntu.edu.tw	r07944037@ntu.edu.tw	r08922121@ntu.edu.tw

摘 要

服務組合技術的概念是組合各個基本的服務以達成高複雜度的功能，但是因為現實的需求有很大的變化，服務量的不足會是這個技術的一大難題，因此本研究以 Python open source 為目標，將 open source 建成網路服務，來擴充服務數量。

由於 Python 沒有可用的 API 來幫助我們產生網路服務，所以本研究也實作了 PythonPoet 達到生成 Python 程式碼的功能。

關鍵字：服務元件，Flask，PythonPoet，網路服務

一、緒論

為了提供符合使用者需求的 Python 網路服務我們從 GitHub 上抓取 Python open source 專案，透過不同的策略處理無法正常執行的專案並將其做分類及修正[1]，而這些正確執行的專案我們以 Method 為單位將其做切割，形成一個一個網路服務的元件。

由於抓取下來的這些元件無法直接地被使用，我們利用 Flask 框架來處理使用者請求，它是一個以 Python 語言實作的資訊網框架，根據框架的 MVC(Model View Controller)設計架構，Controller 會負責處理使用者請求的流程。

不同的服務請求對應不同的服務元件，需要動態地將這些元件載入，為此我們以 JavaPoet 為模板開發了一個以 Python 為主的 API 來自動生成 Controller（MVC 架構中稱為 Controller）及 Invoker 的程式碼。在 Controller 中會呼叫 Invoker，用於調用先前我們從 open source 抓取下來的 Method。

二、相關研究

2.1 網路服務元件

在 [4] Jian Yang 等人將物件導向服務視為元件，並且依照不同的建構及處理訊息的方式將元件分為六大類。他們將服務元件看成是一個 class 而 message 視為 attribute，service logic 則做為 method。

[3] 中 Jian Yang 將網路服務元件也視為 class，不同的是網路服務元件不會繼承上述的六大類 class，而是能夠繼承其他 service class，這個特性使服務元件能夠被其他服務使用，並且可以重新定義 message 或 service logic。

有別於上述兩種方法，我們是把物件導向原始碼直接轉換為服務元件。

2.2 Flask

Flask 是以 Python 編寫而成的資訊網框架，輕量化的核心設計使 Flask 的擴充性較高，因此被稱為 microframework。與另一個常見的 Python 框架 Django 相比，Flask 給予開發者們較大的彈性，能夠根據需求來選擇自己想要的 extension，而 Django 雖然較為完善但是在 ORM、表單驗證及模板引擎

都有一套自己的做法，因此比起 Django 我們更傾向於選擇 Flask 來做為論文中使用的主要框架。

2.3 JavaPoet

做為自動生成 Java 文件的第三方函式庫，JavaPoet 提供了簡潔且易懂的 API，並且可以為不同的元素產出不同格式的程式碼，透過 JavaPoet 存取 class, method 及 field 的內容相對來說也較 string 形式的 Java source file 容易。

2.4 生成 Java 服務元件

[7] 將使用 Java 撰寫的 open source，以 2.1 的服務元件架構進行拆解。為了能夠將他們部署在網路上，針對拆解好的每個元件，依照各個框架的請求形式，提供框架能夠呼叫元件的介面(也就是 Controller)，Controller 內部再用 Invoker 來實際載入服務。

此外，為了要讓使用者能夠了解各個元件的功能，每個服務我們會產出一份 WSDL(Web Service Description Language)，功能是描述這個服務，告訴使用者這個服務需要有什麼輸入和輸出，藉由這個方式，可以打破不同 open source 之間的分界，使用者可以按照自己的需求，透過網路請求使用自己需要的各個服務元件來組成需要的功能。

三、自動生成程式碼

3.1 從原始碼解析服務元件

[2] 提到如何從服務提供者中獲取符合需求的網路服務。為了建構服務元件我們將抓取下來的 Python source code 進行剖析，針對 function 及 method 分別處理。

在剖析程式碼方面，Python 提供了 Parser 套件供使用，我們可以利用 Parser 來獲得一個 Python file 中的各種資訊，便可以從中取出 method 及 function，而 script 只專注在流程控制，無須將其視為一個元件，故只需考慮 method 及 function 兩種做為服務元件。

3.2 PythonPoet

我們以 JavaPoet 為模板開發了一個 Python API 用來生成 Controller 及 Invoker，PythonPoet 除了生成 Controller 及 Invoker 之外，也可以根據不同需求生成 Python 程式碼。

3.2.1 系統架構

在 PythonPoet 中提供不同的 Spec，這些 Spec 可以看成是構成程式碼的各種元素，可以是 class、method，也可能是一個 field。我們將 Spec 都當成是 class，這些 Spec 會有自己的 emit() 方法來做不同的處理，即不同 Spec 會使用 emit()來生成自己的程式碼格式。

一個 Python file(.py)中可能包含了多個 class，我們將不同的 class 做為 TypeSpec 的實例，而一個 class 可能包含了多個 method（或 MethodSpec），因此 PythonFile 會使用 TypeSpec 的 emit 方法，而 TypeSpec 再呼叫 MethodSpec 的 emit 方法，如此層層往下，透過 emit 方法來生成所有程式碼。

在所有 Spec 中都有個 emit()方法，emit() 會使用 CodeWriter 來處理細節。Spec 內容被包裝成一個 CodeBlock，這個 CodeBlock 中包含了該 Spec 的特徵資訊，最後 CodeWriter 會去剖析 CodeBlock 做相對應的處理，再將這些程式碼輸出，圖 1 為 PythonPoet 類別圖，整體的架構如圖 2 所示。

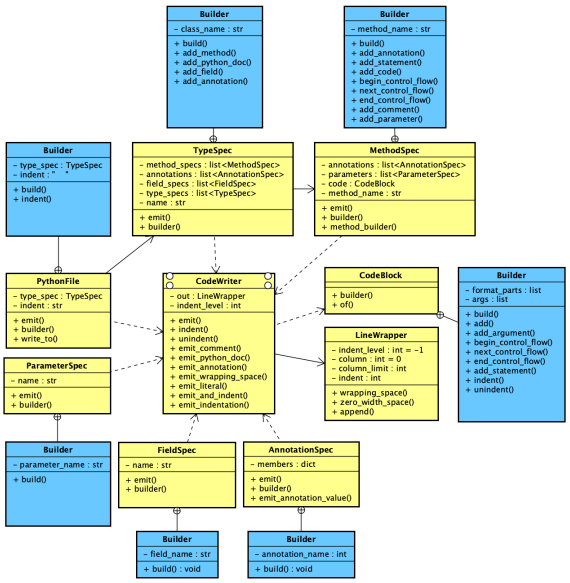

圖 1、PythonPoet 類別圖

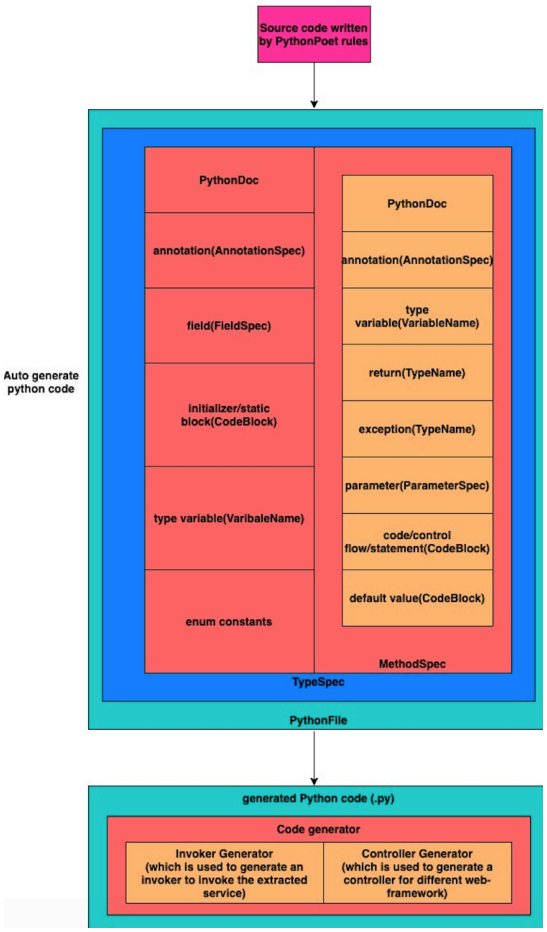

圖 2、生成 Controller 及 Invoker 架構

3.2.2 API 規則

PythonPoet 在不同 spec 中皆有 API 供使用，透過這些 API 簡化生成程式碼的過程，使用者只需了解簡單的規則便可以產生自己想要的 Python 程式碼，並且無須考慮關鍵字及縮排問題。

舉例來說，要產生一個方法，我們需要知道這個函數傳入的參數資訊，這時只需利用 add_parameter() 便可以在欲生成的方法中加入該參數，而 add_statement() 可以在方法中加入一段敘述，最後透過 MethodSpec 的 Builder 建構出整個方法。

一個簡單的範例：

```
method 1=
MethodSpec.method_builder(method_1)
    .add_annotation("route")
    .add_parameter(parameter_1)
    .add_parameter(parameter_2)
    .add_statement("parameter_1 + parameter_2")
    .return(None)
    .build()
```

我們將 method1 輸出，生成的程式碼如下：

```
@route
def method_1(parameter_1, parameter2):
    parameter_1 + parameter_2
    return None
```

3.3 Service Code Generation

這裡我們提出一個簡單的架構，如圖 3

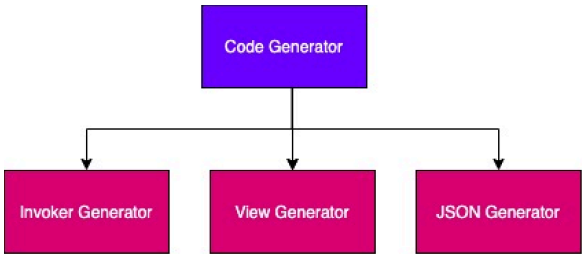

圖 3、Service Code Generator Architecture

Code Generator 控制整個 Generate 的流程，即將工作指派給對應的 Generator。Invoker Generator 及 Controller Generator 產出 Controller 及 Invoker。而 JSON Generator 負責收集服務元件的資訊以利之後產出 WSDL。

3.4 生成 Controller 及 Invoker

Controller 負責處理請求，而 Invoker 則是替我們呼叫該方法，我們使用 PythonPoet 來生成 Controller 及 Invoker 的 generator，這兩個 generator 會再分別產出適合的 Controller 及 Invoker。

3.4.1 生成 Flask 的 Controller generator

為了能夠處理相對應的請求並且在 Flask 上使用，我們需要根據 Flask 在處理 routing 時的語言特徵生成符合 Flask 規範的程式碼。不同的框架會有自己的 Controller，因此我們希望未來除了 Flask 外的其他框架，只要針對該框架設計好對應的規則，透過 PythonPoet 就可以產生 Controller generator 來生成能適用在各個框架上的 Controller。PythonPoet 中 Controller generator 的編寫規則如圖 4 所示，圖 5 為輸出結果。

```
methodspec = MethodSpec.method_builder(method_name) \
    .add_annotation(AnnotationSpec.builder("route").add_url(url).set_app_name(app_name).bui
    .add_parameter("args")\
    .add_statement("instanceInvoker = Invoker()")\
    .add_statement("service_input = request.args.get('serviceInput')")\
    .begin_control_flow("if not(ServiceUtilObject.hasID('requestWrapper, storage'))")\
    .add_statement("return 'error message: there is no this ID'")\
    .end_control_flow()\
    .add_statement("ServiceUtilObject = ServiceUtilObject(requestWrapper, storage, 'databas
    .add_statement("ServiceUtilObject.loadDependency('—python dependency—')")\
    .add_statement("invokerResult = instanceInvoker.service_name(serviceUtilObject, service
    .add_statement("return invokerResult")\
    .build()

type_spec = None
TypeSpec.methods.append(methodspec)  # append methodSpec if TypeSpec is None
pythonfile = PythonFile.builder(package_name, type_spec).build()
```

圖 4、Controller 規則

```
Run:  __init__
@app.route('service_url')
def test_method( args):
    instanceInvoker = Invoker()
    service_input = request.args.get('serviceInput')
    if not(ServiceUtilObject.hasID('requestWrapper, storage')):
        return 'error message: there is no this ID'
    ServiceUtilObject = ServiceUtilObject(requestWrapper, storage, 'database')
    ServiceUtilObject.loadDependency('—python dependency—')
    invokerResult = instanceInvoker.service_name(serviceUtilObject, service_input)
    return invokerResult
```

圖 5、Controller 輸出結果

3.4.2 生成 Flask 的 Invoker generator

利用 PythonPoet 自動生成 Invoker generator，由於網路服務常常需要藉由 serialize 的技術讓物件可以用字串的格式透過 URL 傳遞，因此 Invoker 必須將 Controller 傳過來的 service input 經過 deserialize 步驟還原填入 method，使 method 能夠被正確呼叫。Invoker generator 在 PythonPoet 的編寫規則如圖 6 所示，圖 7 為輸出結果。

```
methodSpec = MethodSpec.method_builder(methodName)\
    .add_parameter('component')\
    .add_parameter('_self')\
    .add_parameter('param1')\
    .add_parameter('param2')\
    .add_statement('obj = pickle.loads(_self)')\
    .add_statement('method = getattr(obj, "someMethod")')\
    .add_statement('return pickle.dumps(method(param1, param2))')\
    .build()

typeSpec = None
TypeSpec.methods = list()
TypeSpec.methods.append(methodSpec)
pythonFile = PythonFile.builder(packageName, typeSpec).build()
```

圖 6、Invoker 規則

```
Run:  __init__
from pickle import pickle

def someMethod__type_of_param1__type_of_param2( component, _self, param1, param2):
    obj = pickle.loads(_self)
    method = getattr(obj, "someMethod")
    return pickle.dumps(method(param1, param2))
```

圖 7、Invoker 輸出結果

四、結論

為了要提供適合的 Python 服務元件，我們從 GitHub 上抓取開源專案，之後透過 Parser 剖析這些程式碼並從中擷取函數及方法。

我們選擇使用 Flask 做為應用程式的框架，開發了 PythonPoet，並使用它來編寫 Controller generator 生成 Flask 的 Controller，用於處理服務請求。除了 Controller 之外，我們還生成了 Invoker generator，用於產生該服務相對應的 Invoker，以便該服務能夠正確呼叫抓取下來的元件。

五、未來研究方向

在我們所提出的方法中，除了 Controller 及 Invoker 外，還需要一個 WSDL generator 來產生 WSDL，用來描述 Service name、input parameters、output parameters 及 request URL 等服務資訊，再透過與 BPEL engine[5, 6]的整合，讓 BPEL engine 能夠從 WSDL 中獲取所需的資訊。

參考文獻

[1] J. Lee, H.-W. Chen, and H.-Y. Huang. Analyzing the build-ability of open source java projects on github. *In Taiwan Conference of Software Engineering 2019*, 2019.

[2] J. Lee, S.-P. Ma, and A. Liu. *Service Life Cycle Tools and Techniques: Methods, Trends and Advances*. IGI Global, 2011.

[3] J. Yang. Web service componentization. *Commun*. ACM, 46(10):35–40, Oct. 2003.

[4] J. Yang and M. P. Papazoglou. Web component: A substrate for web service reuse and composition. In A. B. Pidduck, M. T. Ozsu, J. Mylopoulos, and C. C. Woo, editors, *Advanced Information Systems Engineering*, pages 21–36, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[5] J. Lee, S.-J. Lee, H.-M. Chen, and K.-H. Hsu. Itinerary-based mobile agent as a basis for distributed OSGi services. *IEEE Transactions on Computers*, 62(10):1988–2000, Oct. 2013.

[6] J. Lee, S.-J. Lee, and P.-F. Wang. A framework for composing soap, non-soap and non-web services. *IEEE Transactions on Services Computing*, doi: 10.1109/TSC.2014.2310213, 2014.

[7] H.-Y. Huang. Atomic Service Generation from Java-Based Open Source Software. Master Thesis, National Taiwan University, 2019.