**Множественная диспетчеризация**

Автор: Дэвид Мертц (David Mertz)

Перевод: Intersoft Lab

**Обобщение полиморфизма с помощью мультиметодов**

Во многом универсальность объектно-ориентрованного программирования (ООП) возможна благодаря полиморфизму: в надлежащем контексте объекты разных видов могут вести себя различным образом. Однако большая часть ООП - это единичная диспетчеризация, то есть единственный выделенный объект определяет, какую ветвь кода избрать. Концептуально более общий подход - позволить всем аргументам, передаваемым в функцию/метод, определять ее специализацию. В этой статье рассматривается реализация множественной диспетчеризации в Python и показано, как это улучшает программы.

**Что такое полиморфизм?**

Большинство программистов, использующих полиморфизм - на Python или других языках объектно-ориентированного программирования -, находят ему весьма практическое и конкретное применение. Возможно, наиболее общий случай использования полиморфизма - это создание семейства объектов, которые придерживаются общего протокола. В Python это обычно просто вопрос нерегламентированного полиморфизма; в других языках чаще объявляются формальные интерфейсы, и/или эти семейства обладают общим предком.

Например, существует множество функций, которые работают с объектами, "подобными файлам", где это подобие файлам определяется просто посредством поддержания нескольких методов, как .read(), .readlines() и, возможно, .seek(). Функция, как read\_app\_data(), может принимать аргумент src - когда мы вызовем эту функцию, мы, возможно, решим передать ей локальный файл, объект urllib, объект cStringIO или некий объект, определенный пользователем, который разрешает этой функции вызывать src.read(). Каждый тип объекта равнозначен с точки зрения того, как он функционирует в read\_app\_data().

Давайте вернемся немного назад, чтобы понять, что здесь действительно происходит. По существу, нас интересует, как выбрать надлежащую ветвь кода для выполнения в контексте; старомодный процедурный код может принимать эквивалентные решения, ООП просто придает элегантность. Например, фрагмент процедурного (псевдо) кода мог бы выглядеть следующим образом:

Листинг 1. Процедурный выбор ветвей кода по типу объекта

...bind 'src' in some manner...

if <<src is a file object>>:

read\_from\_file(src)

elif <<src is a urllib object>>:

read\_from\_url(src)

elif <<src is a stringio object>>:

read\_from\_stringio(src)

...etc...

Организовав поддержку общих методов объектами различных типов, мы перемещаем решение о диспетчеризации в объекты из явного условного блока. Просматривая дерево наследования, данный объект src узнает, какие блоки кода ему нужно вызывать. Однако, по-прежнему происходит неявное переключение, но по типу объекта src.

Объект src привилегирован по отношению к любым аргументам, передаваемым в его методы. Из-за синтаксиса ООП эта привилегированность кажется неизбежной, но на самом деле это не так. Во многих случаях процедурное переключение просто переносится в тела методов классов. Например, мы могли бы реализовать совместимые по протоколу классы Foo и Bar следующим образом:

Листинг 2. Реализация метода .meth() с помощью Foo и Bar

class Foo:

def meth(self, arg):

if <<arg is a Foo>>:

...FooFoo code block...

elif <<arg is a Bar>>:

...FooBar code block...

class Bar:

def meth(self, arg):

if <<arg is a Foo>>:

...BarFoo code block...

elif <<arg is a Bar>>:

...BarBar code block...

# Function to utilize Foo/Bar single-dispatch polymorphism

def x\_with\_y(x, y):

if <<x is Foo or Bar>> and <<y is Foo or Bar>>:

x.meth(y)

else:

raise TypeError,"x, y must be either Foo's or Bar's"

Имеется пять различных ветвей/блоков кода, которые могут выполняться при вызове x\_with\_y(). Если типы x и y не подходят, возбуждается исключение (разумеется, вы могли бы сделать что-нибуль другое). Но, предполагая, что с типами все в порядке, ветвь кода выбирается сначала посредством полиморфной диспетчеризации, а затем посредством процедурного переключения. Кроме того, переключения внутри определений Foo.meth() и Bar.meth() в значительной степени эквивалентны. Полиморфизм - в разновидности с единичной диспетчеризацией - решает лишь половину задачи.

**Полная реализация полиморфизма**

В случае полиморфизма с единичной диспетчеризацией выделяется объект, который "владеет" методом. Синтаксически в Python его выделяют, располагая его имя перед точкой - все, что следует за точкой: имя метода и левая скобка - просто аргумент. Но семантически этот объект является особенным при использовании дерева наследования для выбора метода.

А что если бы мы обрабатывали особым образом не один объект, а позволили бы каждому объекту, задействованному в блоке кода, участвовать в выборе ветви выполнения? Например, мы могли бы выразить наше пятистороннее переключение более симметрично:

Листинг 3. Множественная диспетчеризация Foo и Bar

x\_with\_y = Dispatch([((object, object), <<exception block>>)])

x\_with\_y.add\_rule((Foo,Foo), <<FooFoo block>>)

x\_with\_y.add\_rule((Foo,Bar), <<FooBar block>>)

x\_with\_y.add\_rule((Bar,Foo), <<BarFoo block>>)

x\_with\_y.add\_rule((Bar,Bar), <<BarBar block>>)

#...call the function x\_with\_y() using some arguments...

x\_with\_y(something, otherthing)

Я думаю, что эта симметричность полиморфной диспетчеризации по множеству аргументов гораздо более элегантна, чем предшествующий стиль. Кроме того, этот стиль позволяет документировать одинаковую роль этих двух объектов, задействованных в определении подходящей ветви кода.

Стандартный Python не разрешает конфигурировать этот тип множественной диспетчеризации; но, к счастью, вы можете сделать это, воспользовавшись написанным мною модулем multimethods. См. Ресурсы, чтобы скачать этот модуль отдельно или в составе утилит Gnosis. После того, как вы установили multimethods, все, что от вас требуется - включить в начало своего приложения следующую строку:

from multimethods import Dispatch

"Мультиметоды", как правило, это синоним множественной диспетчеризации; но термин мультиметод предполагает конкретную функциональную/объектную реализацию более абстрактной концепции множественной диспетчеризации.

Экземпляр Dispatch - это вызываемый объект, его можно конфигурировать с любым желаемым количеством правил. К тому же, можно использовать метод Dispatch.remove\_rule(), чтобы удалять правила; благодаря этому множественная диспетчеризация с использованием multimethods становится несколько более динамичной, чем статическая иерархия классов (но вы также можете совершить некие замысловатые действия с классами Python во время исполнения). Также заметьте, экземпляр Dispatch может принимать переменное число аргументов; сопоставление выполняется сначала по числу аргументов, затем по их типам. Если экземпляр Dispatch вызывается с любым шаблоном, который не определен в правиле, возбуждается TypeError. Инициализация x\_with\_y() с запасным шаблоном (object,object) необязательна, если вы просто хотите, чтобы в неопределенных ситуациях возбуждалось исключение.

Каждый кортеж (pattern,function), перечисленный в инициализации Dispatch, просто передается далее в метод .add\_rule(); это исключительно вопрос удобства программирования - устанавливать правила при инициализации или позже (можно комбинировать подходы, как в предшествующем примере). При вызове функции из диспетчера аргументы, используемые при вызове, передаются диспетчеру; вы должны обеспечить, чтобы функция, которую вы используете, могла принять то число аргументом, с которым она сопоставляется. Например, приведенные ниже вызовы эквиваленты:

Листинг 4. Явный вызов и вызов функции при диспетчеризации

# Define function, classes, objects

def func(a,b): print "The X is", a, "the Y is", b

class X(object): pass

class Y(object): pass

x, y = X(), Y()

# Explicit call to func with args

func(x,y)

# Dispatched call to func on args

from multimethods import Dispatch

dispatch = Dispatch()

dispatch.add\_rule((X,Y), func)

dispatch(x,y) # resolves to 'func(x,y)'

Очевидно, что если вы знаете типы x и y во время проектирования, алгоритм задания диспетчера - просто накладные расходы. Но то же ограничение справедливо и для полиморфизма - он удобен, лишь когда вы не можете ограничить объект единственным типом для каждой ветви исполнения.

**Улучшение наследования**

Множественная диспетчеризация не просто обобщает полиморфизм, она предоставляет более гибкую альтернативу наследованию во многих контекстах. Рассмотрим в качестве иллюстрации следующий пример. Предположим, что вы пишете программу построения чертежей или автоматизированного проектирования, которая работает с различными фигурами (shape); в частности, вы хотите, чтобы вы могли комбинировать две фигуры таким образом, чтобы результат зависел от обеих задействованных фигур. Кроме того, набор рассматриваемых фигур будет расширяться производными приложениями или подключаемыми библиотеками. Расширение набора классов фигур является неизящным подходом при модернизации, например:

Листинг 5. Наследование для расширения возможностей

# Base classes

class Circle(Shape):

def combine\_with\_circle(self, circle): ...

def combine\_with\_square(self, square): ...

class Square(Shape):

def combine\_with\_circle(self, circle): ...

def combine\_with\_square(self, square): ...

# Enhancing base with triangle shape

class Triangle(Shape):

def combine\_with\_circle(self, circle): ...

def combine\_with\_square(self, square): ...

def combine\_with\_triangle(self, triangle): ...

class NewCircle(Circle):

def combine\_with\_triangle(self, triangle): ...

class NewSquare(Square):

def combine\_with\_triangle(self, triangle): ...

# Can optionally use original class names in new context

Circle, Square = NewCircle, NewSquare

# Use the classes in application

c, t, s = Circle(...), Triangle(...), Square(...)

newshape1 = c.combine\_with\_triangle(t)

newshape2 = s.combine\_with\_circle(c)

# discover 'x' of unknown type, then combine with 't'

if isinstance(x, Triangle): new3 = t.combine\_with\_triangle(x)

elif isinstance(x, Square): new3 = t.combine\_with\_square(x)

elif isinstance(x, Circle): new3 = t.combine\_with\_circle(x)

В частности, каждый существующий класс фигуры должен добавлять возможности потомку, что приводит к комбинаторной сложности и трудностям при сопровождении.

Напротив, метод множественной диспетчеризации более прост:

Листинг 6. Мультиметоды для расширения возможностей

# Base rules (stipulate combination is order independent)

class Circle(Shape): pass

class Square(Shape): pass

def circle\_with\_square(circle, square): ...

def circle\_with\_circle(circle, circle): ...

def square\_with\_square(square, square): ...

combine = Dispatch()

combine.add\_rule((Circle, Square), circle\_with\_square)

combine.add\_rule((Circle, Circle), circle\_with\_circle)

combine.add\_rule((Square, Square), square\_with\_square)

combine.add\_rule((Square, Circle),

lambda s,c: circle\_with\_square(c,s))

# Enhancing base with triangle shape

class Triangle(Shape): pass

def triangle\_with\_triangle(triangle, triangle): ...

def triangle\_with\_circle(triangle, circle): ...

def triangle\_with\_square(triangle, square): ...

combine.add\_rule((Triangle,Triangle), triangle\_with\_triangle)

combine.add\_rule((Triangle,Circle), triangle\_with\_circle)

combine.add\_rule((Triangle,Square), triangle\_with\_square)

combine.add\_rule((Circle,Triangle),

lambda c,t: triangle\_with\_circle(t,c))

combine.add\_rule((Square,Triangle),

lambda s,t: triangle\_with\_square(t,s))

# Use the rules in application

c, t, s = Circle(...), Triangle(...), Square(...)

newshape1 = combine(c, t)[0]

newshape2 = combine(s, c)[0]

# discover 'x' of unknown type, then combine with 't'

newshape3 = combine(t, x)[0]

Определение новых правил (и поддержка функций/методов) в значительной степени эквивалентны. Но огромное преимущество стиля множественной диспетчеризации - это цельность, с помощью которой вы комбинировать фигуры неизвестных типов. Вместо того, чтобы возвращаться к явным (и длинным) условным блокам, определения правил автоматически решают эти вопросы. Что еще лучше, все комбинирование выполняется одним вызовом combine(), а не с помощью "зверинца" из разных комбинирующих методов.

**Передача диспетчеризации**

Не испытывая необходимости больше думать о диспетчеризации, класс multimethods.Dispatch будет выбирать "наилучшее совпадение" для данного обращения к диспетчеру. Однако, иногда стоит заметить, что "лучшее" не значит "единственное". То есть, обращение к dispatch(foo,bar) может давать точное совпадение с правилом (Foo,Bar) - но оно также может задавать менее точное совпадение (не промах!) для (FooParent,BarParent). Точно так, как иногда вы хотите вызывать методы базовых классов в методе производного класса, вы также иногда желаете вызывать менее специфические правила в диспетчере.

Модуль multimethods позволяет задавать вызовы менее специфических правил как грубо, так и с тонкой настройкой. На грубом уровне, обычно вы просто хотите автоматически вызывать менее специфичное правило в начале, либо в конце выполнения блока кода. Подобным образом вы практически всегда вызываете метод надкласса в начале, либо в конце тела метода потомка. Общий вариант начального/конечного вызова менее специфичных методов может быть задан просто как часть правила. Например:

Листинг 7. Автоматическое воспроизведение диспетчеризации

class General(object): pass

class Between(General): pass

class Specific(Between): pass

dispatch = Dispatch()

dispatch.add\_rule((General,), lambda \_:"Gen", AT\_END)

dispatch.add\_rule((Between,), lambda \_:"Betw", AT\_END)

dispatch.add\_rule((Specific,), lambda \_:"Specif", AT\_END)

dispatch(General()) # Result: ['Gen']

dispatch(Specific()) # Result: ['Specif', 'Betw', 'Gen']

Разумеется, в некоторых ситуациях (как для правила (General)) менее специфичное правило отсутствует. Для обеспечения единообразия, однако, каждое обращение к диспетчеру возвращает список значений из всех функций, которым передается управление таким образом. Если в правиле не определены ни AT\_END, ни AT\_START, распространение вызовов не производится (и возвращается список из одного элемента). Этим объясняется индекс [0] в примере с фигурами, который, вероятно, кажется загадочным .

Для тонкой настройки распространения вызовов применяется метод диспетчера .next\_method(). Чтобы задать распространение вызовов вручную, нужно использовать для определения правил метод .add\_dispatchable(), а не метод .add\_rule(). Кроме того, диспетчеризованные функции сами должны принимать аргумент dispatch. При вызове диспетчера вы либо должны передать аргумент, задающий диспетчер, либо вы можете воспользоваться вспомогательным методом .with\_dispatch(). Например:

Листинг 8. Программирование с ручной передачей

def do\_between(x, dispatch):

print "do some initial stuff"

val = dispatch.next\_method() # return simple value of up-call

print "do some followup stuff"

return "My return value"

foo = Foo()

import multimethods

multi = multimethods.Dispatch()

multi.add\_dispatchable((Foo,), do\_between)

multi.with\_dispatch(foo)

# Or: multi(foo, multi)

Вызов менее специфичных мультиметодов вручную может оказаться запутанным - примерно так же, как и обращение к методам базовых классов. Чтобы эти вопросы стали управляемыми, обращение к .next\_method() всегда возвращает простой результат вызова верхнего уровня - если вы хотите собрать такие результаты в список, как тот, что создает аргумент AT\_END, вам нужно добавлять и обрабатывать те величины, которые вы считаете уместными. Наиболее общий вариант использования, однако - выполнение последовательности связанных инициализаций; в этом случае возвращаемые величины обычно неважны.

**Замечания выполнении в многонитевой среде**

Стоит привести краткое замечание, пока читатель не столкнулся с проблемой. Из-за необходимости сохранения состояния для отслеживания, какие (последовательно менее специфичные) правила вызывались, диспетчер не является нитебезопасным. Если нужно использовать диспетчер в многонитевой среде, необходимо "клонировать" его для каждой нити. Это ненакладно с точки зрения ресурсов: памяти и процессора, так что клонирование диспетчеров не вызывает существенных неудобств. Например, предположим, что функция могла бы вызываться из разных нитей; вы можете написать:

Листинг 9. Клонирование для безопасности нити

def threadable\_dispatch(dispatcher, other, arguments)

dispatcher = dispatcher.clone()

#...do setup activities...

dispatcher(some, rule, pattern)

#...do other stuff...

Если внутри threadable\_dispatch() не запускаются новые нити, все нормально.

Вам потребуется некоторое время, чтобы освоиться с идей множественной диспетчеризации, даже - или особенно - если вы весьма опытны в объектно-ориентированном программировании. Но после того, как вы немного с ней поэкспериментируете, вероятно, вы обнаружите, что множественная диспетчеризация обобщает и усиливает преимущества, которыми ООП обладает прежде всего над процедурным программированием.