A serpente que morde a própria cauda

Metalinguagem

Com a chegada do Ruby On Rails, os programadores estão redescobrindo um conceito não muito moderno, mas surpreendente...Programas que modificam programas?

por Jose Maria Ruiz

As linguagens dinâmicas (nas quais o tipo de variável é definido no tempo da execução) sempre foram relegadas a linguagens de segunda classe. Isso se deve ao costume de serem empregadas para fazer o trabalho sujo e resolver algum problema rapidamente, já que para programar algo mais sério sempre se utiliza *C*, *C++* ou *Java*.

Mas isso não é bem verdade. As linguagens dinâmicas possuem certas vantagens sobre as estáticas, vantagens essas que as estáticas estão constantemente tentando imitar.

Uma dessas vantagens é a metaprogramação. É uma palavra estranha, por isso é melhor explicarmos um pouco sobre seu significado. Podemos traduzir "meta" como "a nível superior". Metalinguagem é a linguagem usada para falar da linguagem. A metalinguagem está um nível acima da linguagem que descreve. Se digo "a palavra carro é um substantivo", então estou empregando o português para falar do português, e em "a palavra *car* é um substantivo" estou empregando o português para falar do inglês. Portanto, o português é metalinguagem em ambos os casos.

A metaprogramação consiste de programas que podem modificar a si mesmos ou outros programas. Dito assim, pode parecer um pouco complicado. Mas imaginemos por um momento que fosse possível trocar elementos do programa em tempo real, não dados, mas o programa em si.

Podemos imaginar um objeto que "fale" certo protocolo (por protocolo entendemos um conjunto de métodos que permitem interagir com o objeto, por exemplo, para guardá-lo em disco rígido). A certo momento, surge a necessidade de que esse objeto "fale" esse protocolo, mas o projetista do programa não levou essa possibilidade em conta, de modo que isso pode ser resolvido envolvendo-se o objeto em outro objeto, ou herdando-o de uma superclasse que responda a esse

protocolo. Agora, se essa ocorrência se repete com muitos protocolos, então a hierarquia de herança das classes começa a se complicar desnecessariamente, com código repetido em muitas ramificações.

O C++ tenta chegar a uma solução através do uso de *templates* (modelos). O Java, por sua vez, fornece ao programador a reflexão (os objetos podem fazer perguntas a outros objetos) e, desde o Java 1.5, as classes genéricas, algo parecido com os templates do C++. Na chamada hierarquia ou extensão horizontal, não herdamos a não ser que façamos uso das ferramentas que estão ao lado, e não "acima" como as superclasses. Portanto, esse tipo de extensão não requer que a hierarquia dos objetos seja aumentada com novas classes.

Mesmo assim existe outra possibilidade. E se pudéssemos adicionar métodos imediatamente a nossos objetos? Dessa maneira, quando um objeto precisar falar certo protocolo, simplesmente adiciona a si mesmo os métodos necessários, digamos, assim como decoramos as árvores de Natal, colocando bolas e enfeites conforme desejamos.

Conceitos básicos

Nos míticos laboratórios da Xerox em Palo Alto, Gregor Kiczales e outros desenvolveram, em meados dos anos 90, o conceito *MOP* (*Meta Object Protocol*).

A idéia é definir todo o mecanismo necessário para se trabalhar em objetos em dois níveis. No primeiro, o metanível, são usadas as formas primitivas da linguagem para se criar objetos. No segundo, usa-se esse mecanismo (criar classes, instâncias, métodos...) para voltar a defini-lo novamente.

Qual é o objetivo? Se uma classe estiver definida com o uso de classes, então você poderia modificá-la, visto que ela não passa de uma estrutura de dados que está "viva" durante a execução do programa. Isso possibilita coisas muito interessantes. Veja o **exemplo 1**.

O que aconteceu aqui? Definimos uma classe e uma função, criamos uma instância da classe e adicionamos a função à instância da classe. O que acontecerá se criarmos um segundo objeto?

```
>>> objeto2 = Classe1("2")
>>> objeto2.imprime(objeto2)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
AttributeError: Classe1 instance has no
➥ attribute 'imprime'
>>>
```

O `objeto2` não tem `imprime`! Logo, percebemos que a modificação do objeto só afetava ele mesmo, e não o restante das instâncias da `Classe1`. E como é possível conferir coisas a uma instância? Como o Python representa as instâncias como dicionários, as seguintes assinaturas são equivalentes:

```
objeto.imprime = imprime
objeto['imprime'] = imprime
```

Além disso, as funções se comportam no Python da mesma forma que os dados; podem ser associadas a variáveis e passadas como parâmetros. Podemos modificar o comportamento das instâncias, mas o que acontece com as classes?

```
>>> Classe1.imprime = imprime
>>> objeto3 = Classe1("Ola")
>>> objeto3.imprime()
Ola
>>>
```

E também podem ser modificadas automaticamente. Não só isso, mas, a partir desse momento, qualquer instância terá acesso a essas modificações. O que acontece se tivermos declarado a instância antes da modificação e adicionarmos uma função à classe depois? Veja o **exemplo 2**.

A modificação da classe afeta todas as suas instâncias, passadas ou futuras.

Todas essas características nos abrem um amplo leque de possibilidades. Podemos modificar hierarquias de objetos na hora da execução para que possam se adaptar a novos protocolos. Não vamos prestar muita atenção às metaclasses em si – vamos vê-las em um próximo artigo –, mas sim em um padrão de desenho que nos permitirá adicionar funcionalidade de maneira controlada – estamos falando do padrão *Decorator*.

Características de uma função

Para prosseguirmos, é importante entendermos como funciona a passagem de parâmetros no Python. No Python, os parâmetros de uma função são na verdade uma lista. O interpretador recolhe automaticamente os parâmetros e gera uma lista com eles. Isso é indicado no cabeçalho da função mediante um * diante do nome da lista que guarda os parâmetros. As funções aceitam ainda um dicionário como segundo parâmetro, indicado mediante a aparição de **

Exemplo 1: Começando

```
01 >>> class Classe1:
02 ...     def __init__(self,valor):
03 ...         self.valor = valor
04 ...     def imprime(self):
05 ...         print self.valor
06 ...
07 >>> objeto = Classe1("1")
08 >>> objeto.imprime = imprime
09 >>> objeto.imprime(objeto)
10 1
11 >>>
```

Exemplo 2: Declaração antes da modificação

```
01 >>> class Classe2:
02 ...     def __init__(self,valor):
03 ...     self.valor = valor
04 ...
05 >>> objeto4 = Classe2("Ola")
06 >>> def imprime(self):
07 ...     print self.valor
08 ...
09 >>> Classe2.imprime = imprime
10 >>> objeto4.imprime()
11 Ola
12 >>>
```

Exemplo 3: Decorator escreveNome

```
01 def escreveNome (f):
02   def decorator (*args, **kwds):
03     print "==> Executando", f.func_name
04     f(*args, **kwds)
05
06   return decorator
07
08 @escreveNome
09 def funcao():
10   print "Oi, eu sou a funcao"
11
12 @escreveNome
13 def soma(a,b):
14   print "Resultado:", a+b
```

para os argumentos com nome. Dessa maneira, qualquer função aceita na verdade os parâmetros:

```
function(*args. **kwdargs)
```

Costuma-se usar os nomes `args` e `kwdargs` para nomeá-los. É importante saber disso para podermos criar funções sem termos idéia sobre quais parâmetros elas aceitarão, visto que no final todas as funções aceitam esses dois parâmetros na verdade.

Decorator

O que é um Decorator? É um padrão de desenho do famoso livro "Design Patterns". Adquiriu sua fama por ter dado nome a várias técnicas já estabelecidas. Ele as reuniu e criou o conceito de padrão de desenho: um conjunto de objetos e comportamentos definidos que surgem com certa facilidade no desenvolvimento de qualquer sistema de complexidade mediana.

No mundo da programação orientada a objetos, são famosos muitos desses padrões de desenho – o *Observer* e o *Proxy* são alguns exemplos. O padrão Decorator é menos conhecido porque realiza uma tarefa um pouco complicada de se entender, e portanto nem todo mundo se atreve a trabalhar com ele. O Decorator modifica funções. Pode-se dizer que ele recolhe nosso código e insere comportamentos adicionais.

Seu nome vem do fato de podermos vê-lo como um algoritmo que, na árvore de código, vai acrescentando novo código de forma automática. É semelhante ao que ocorre quando vemos em uma página web sempre o mesmo menu, na mesma posição, apesar de mudarmos de uma seção para outra. A "decoração" envolve o conteúdo da página.

O mundo do Python viveu uma espécie de guerra entre grupos diferentes de desenvolvedores, por tentarem encontrar a melhor sintaxe para esse padrão. A comunidade Python é quase obsessiva a respeito de sua sintaxe, querendo que ela seja clara e concisa. Por fim, impôs-se a seguinte sintaxe:

```
@decorator
def função():
   o_que_for
```

`@decorator` é o nome do Decorator que iremos empregar. O Decorator sempre deve ter à frente uma arroba, e deve ser seguido pela ação a ser decorada. Visto assim, não parece muito útil, mas imaginemos uma função, por exemplo `VisitaWeb`, à qual adicionamos o seguinte Decorator:

```
@gravar_log
def VisitaWeb():
...
```

Ela guardaria em um log a informação de cada visita, com cada execução de `VisitaWeb()`, e isso sem ter que acrescentar novas funcionalidades a `VisitaWeb()`, e ainda por cima podendo usar `@gravar_log` em outros lugares, como por exemplo em `Compra/Efetuada()` ou `VisitaPerigosa`. Podemos adicionar funcionalidades a códigos já existentes sem modificar nada, simplesmente acompanhando o código de uma linha que indica qual Decorator aplicar. Mas a coisa não acaba aí. Podemos adicionar Decorators:

```
@verificar_acesso
@gravar_log
def VisitaWeb():
...
```

Será que existe alguma mágica ou vodu por trás de algo tão útil? Por que isso não é ensinado em qualquer segundo parágrafo de tutorial de Python? É porque, até há pouco tempo, era extremamente complexo adicionar Decorators no Python, e mesmo com as últimas melhorias ainda continua sendo um processo complexo.

Criar um Decorator

Um Decorator é uma função que aceita uma função como parâmetro e devolve como resultado uma função. Um pouco difícil, não? É claro que sim, por isso começamos com um exemplo mais simples. Queremos um Decorator que escreva na tela o nome da função sobre a qual ele pode ser aplicado, a cada vez que ela for executada.

Um Decorator não deixa de ser uma função Python como qualquer outra. Vejamos sua definição e seu resultado no **exemplo 3**.

`escreveNome` aceita uma função e cria outra que fornecemos ao chamar `decorator`. O nome dessa última função não importa, já que ele será descartado, de modo que podemos usar sempre o mesmo nome para distingui-la. O objetivo dessa função é ser devolvida e entrar no lugar da função original.

A função interna `decorator` adiciona o código ou as operações que sejam necessárias e executa em algum momento (ou não, dependendo do que queiramos) a função original. Como podemos observar na execução do **exemplo 4**, agora, cada vez que for executado `função` ou `soma`, é impressa uma mensagem na tela. Se quisermos eliminar esse comportamento, temos apenas que tirar `@escreveNome` da definição da função em questão.

Exemplo 4: Uma amostra de Decorator

```
01 >>> soma(1,2)
02 ==> Executando soma
03 Resultado: 3
04 >>> funcao()
05 ==> Executando funcao
06 Oi, eu sou a funcao
07 >>>
```

Exemplo 5: Função fechar

```
01 def fechar(valor):
02   def incrementa(quantidade):
03     return quantidade + valor
04   return incrementa
```

Programação avançada

Agora vamos ver uma das construções mais poderosas que uma linguagem de programação pode ter. É anterior aos objetos e, mesmo assim, às vezes muito mais simples e poderosa. Já fizemos uso dela sem perceber. Mas onde? Quando? Por quê? Quem?

Vamos nos fixar novamente no **exemplo 3**. Há algo estranho aí. E não é fácil de identificar – isso porque são apenas cinco linhas de código!

A grande pergunta é "o que acontece com o `f` em `decorator`? Nada?" Vamos repassar o que sabemos das variáveis. Apesar do fato de que uma função possa parecer uma variável estranha a algum leitor, isso é algo que os programadores de C e C++ costumavam fazer através do uso de ponteiros para funções.

`f` é uma variável que faz referência a uma função. Visto assim, não há nada de especial; o estranho é que `f` é o parâmetro recebido por `escreveNome`, e depois criamos a função `decorator` e saímos. Mas existem vários Decorators anônimos, pelo menos um para `soma` e outro para `funcao`. Nesse momento, `f` em outras linguagens de programação não apontaria para nada, e estaria no limbo das variáveis.

Vejamos um exemplo para entender isso. No **exemplo 5**, definimos um Decorator que nomeamos de maneira extremamente original: `fechar`. Esse Decorator gera uma função que faz uso do parâmetro que lhe passamos, o parâmetro `valor`. Quando a função faz uso de `valor`, captura o seu conteúdo! A partir desse momento, essa função fará uso do conteúdo que `valor` possui durante a sua definição. É estranho? Na verdade, em um primeiro momento sim, a não ser que o leitor tenha conhecimentos de *Lisp* ou *Scheme*.

Não em vão, Peter Norvig (ver [1]) diz que Python é Lisp com outra sintaxe.

Mas cuidado, não acredite que `valor` seja convertida em uma constante. Senão, a função `fechar` ficaria trancada num círculo. É como se junto a `fechar` houvesse uma variável chamada `valor`, à qual `fechar` possa fazer referência como se fosse global, mas "só para ela mesma"!

No **exemplo 6** podemos ver uma demonstração de interação com `fechar`. É importante perceber os fechamentos, visto que nos permitem gerar funções que guardam valores. E isso é vital para os Decorators.

Dois Decorators são melhores que um

Para terminar, vamos fazer algo de útil com essa técnica. Geralmente se fala dos Decorators em termos de serviços que gostaríamos de acrescentar ao código. Os mais típicos são criar registros de uso, de segurança, sistemas de verificação ou persistência. Também se fala muito em multimétodos.

Esses últimos consistem de algo parecido à sobrecarga de C++ ou Java, isto é, poder definir muitas funções que possam ser executadas com base nos tipos ou quantidade de parâmetros, mas que todas tenham o mesmo nome. Um exemplo poderia ser a função `soma(a,b)`. Não é a mesma coisa somar inteiros ou pontos flutuantes, fracionários ou números complexos. Na verdade, cada tipo de número requer sua própria soma.

Nós vamos criar dois serviços de exemplo. Um será para facilitar a depuração de um programa, o outro será um verificador de valores.

É preciso explicar um pouco sobre o verificador. Vamos imaginar uma situação na qual precisamos calcular um valor muito complexo duas vezes, mas o cálculo é sempre o mesmo, e sempre devolve o mesmo valor para os parâmetros dados. Ou, olhando para a Web, imaginemos que geraremos a mesma página e que somente a atualizaremos a cada intervalo de tempo determinado.

Fazer algo assim requer muito trabalho: temos que capturar a requisição, determinar se os parâmetros para o cálculo ou página web form modificados, ou se ocorre algum evento (já se passaram 5 minutos desde que a função foi invocada?), e então devolver o valor ou calculá-lo novamente. Necessitamos ao menos de um objeto, e o que é pior: cada função que requeira essa propriedade deverá implementá-la de novo ou herdar de um objeto que a implemente. Mas em Python não temos herança múltipla, de modo que acabaremos com duas versões de cada objeto: `Pagina` e `ChecaPagina`, `Cálculo` e `ChecaCálculo`.

Os Decorators solucionam esse problema de uma forma genérica e muito elegante. Vamos nos fixar no **exemplo 7**.

São definidos dois Decorators: `verifica` e `debug`; o segundo é simples e se assemelha muito ao **exemplo 6**, simplesmente imprimindo informações sobre a função (seu nome, parâmetros e resultado) com cada execução da mesma.

Já `verifica` realiza mais trabalho. A chave é a variável `verificacao`, que é um dicionário onde armazenamos os parâmetros que são passados para a função em questão. `verifica` está dentro do envelope que criamos com o Decorator.

Quando recebemos os parâmetros, tentamos encontrar o valor da função no dicionário `verificacao`. Se não aparecerem, então temos que calculá-los e armazená-los usando os parâmetros como chave. Mas

Exemplo 6: Prova de fechar

```
01 >>> a = fechar(3)
02 >>> b = fechar(8)
03 >>> a(1)
04 4
05 >>> b(1)
06 9
07 >>>
```

Exemplo 7: Decorators debug e verifica

```
01 def debug(f):
02   nome = f.func_name
03   def envelope(*args,**kwargs):
04     resultado = f(*args,**kwargs)
05     print "Chamando",nome,"com",args,kwargs,"e devolvendo",repr(resultado)
06     return resultado
07
08   return envelope
09
10 def verifica(f):
11   # Fecha
12   verificacao = {}
13   def envelope(*args,**kwargs):
14     chave_1 = hash(args)
15     chave_2 = hash(tuple(kwargs.iteritems()))
16     try:
17       resultado = verificacao[chave_1,chave_2]
18       print '****VERIFICACAO****'
19     except KeyError:
20       verificacao[chave_1,chave_2] = f(*args,**kwargs)
21       resultado = verificacao[chave_1,chave_2]
22     return resultado
23   return envelope
24
22 class Calculadora:
26   def __init__(self,valor):
27     self.valor = valor
28
29   def getValor(self):
30     return self.valor
31
32   @debug
33   @verifica
34   def soma(self,valor):
35     resultado = self.valor + valor
36     return resultado
```

se eles aparecerem, simplesmente devolvemos o valor associado. Dessa maneira, podemos verificar resultados.

A maior complicação, fora tudo o que falamos neste capítulo, se deve ao fato de que os dicionários não podem ser usados para gerar um *hash*, e `kwargs` é um dicionário. Se um objeto não puder ser reduzido a um hash, então não pode ser usado como chave em um dicionário.

Por isso temos que gerar uma tupla a partir de `kwargs` e, a partir dela, gerar um hash. Seguindo essa dinâmica, faríamos o mesmo com `args`, mas isso não é necessário.

Quando se tenta acessar um valor em um dicionário e esse valor não existe, o dicionário lança uma exceção `KeyError` (erro na chave passada). Dessa forma, tentamos o acesso dentro de uma estrutura `try` e capturamos essa exceção, que nos indicará quando temos que calcular o valor.

Como exemplo, na classe `Calculadora`, fazemos uso dos Decorators de maneira que não somente verificamos, mas ainda mostramos informação de depuração para a função sobre a qual os aplicaremos.

Decorando projetos

O conceito de Decorators é semelhante ao da agora famosa *Aspect Oriented Programming* (AOP). Com o passar dos anos, os estudiosos vão se dando conta de que a Programação Orientada a Objetos não é a melhor solução para todos os problemas, e pode chegar a ser desvantajosa. Os Decorators não são nada mais que outro dos mecanismos de extensão horizontal que estão começando a fazer eco na caixa de ferramentas dos melhores programadores. E é por isso que o Python os incorporou. ■

Mais Informações

[1] http://www.norvig.com

O autor

José Maria Ruiz está finalizando seu projeto de conclusão de curso de Engenharia Técnica em Informática de Sistemas e está há mais de sete anos usando e desenvolvendo Software Livre, há dois anos no FreeBSD.