Ruby

Thomas bemutatja a Ruby programozási nyelv erejét és rugalmasságát.

A Ruby érett, korszerű, tisztán objektumközpontú programozási nyelv. Írásmódja tömör és állandó, így egyszerre könnyen olvasható és tanulható, valamint rugalmas és kifejező. Ha eddig olyan nyelven programoztál, ami erejét egy hatalmasra püffedt API-ból merítette, meg fogsz lepődni azon, hogy a Ruby API milyen kicsi és mégis milyen hatékony. Mivel szorosan együttműködik az alatta futó operációs rendszerrel, és szinte nevetségesen egyszerű bővíteni, a Ruby egyszerre nagyon hatékony és sokoldalú programozási nyelv. Merész állítások? Nézzük, mi rejtőzik a kijelentések mögött!
A példa kedvéért egy egyszerű Ruby programon fogunk dolgozni, amely egy ideiglenes könyvtárból kitörli a megadott napnál idősebb állományokat. Ezen az alkalmazáson keresztül fogom bemutatni a Ruby alapszintű írásmódját, valamint néhány fontosabb képességét. A teljes programot az első lista tartalmazza. (33. CD Magazin/Ruby könyvtár) A programot konzolról a `./purge.rb` *`tmp_dir ÆllomÆny_max_kora_napokban`* paranccsal futtatjuk, ahol is második értékként napokban számolva azt a kort adjuk meg, amelynél idősebb állományokat a könyvtárból törölni akarunk. Természetesen a programot a `crontab`-ból is meghívhatjuk.

Ruby-alapok

A Ruby objektumközpontú nyelv, amely lehetővé teszi az adatok és eljárások objektumba zárását, az öröklést, valamint támogatja a többalakúságot. A nyelvben minden, még a legegyszerűbb adattípusok is (például karakterlánc, egész) objektumként van megjelenítve. Sőt, az állandók és az osztályok is objektumok. Ennél fogva a Ruby tisztán objektumközpontú nyelv. Az egyetlen kivételt e tekintetben a vezérlőszerkezetek képezik, vagyis néhány olyan kifejezés, mint például a `for`, `if`, `while` stb. Ezek nem objektumok.
Amint a 2. listában láthatjuk (33. CD Magazin/Ruby könyvtár), hogy a `delete_older` eljárás adja meg a program általános logikáját: belépni egy adott könyvtárba és megkeresni a törlendő állományokat.
Akik a szigorúan típusos nyelvekhez (pl.: Java, C++) vannak szokva, furcsának találhatják, hogy az eljárásmegadásban az értékek típusát nem adtuk meg. A Ruby dinamikusan típusos nyelv. Vagyis a változónak nincsen típusa, de annak az objektumnak, amire a változó hivatkozik, van. Ezért nem szükséges a típusmegadás. A dinamikus típusosság az osztályöröklléssel szemben az objektumkompozíciót részesíti előnyben. Nem befolyásolhatjuk az objektumok típusát, amelyek értékként adódnak át az eljárásoknak, így kevésbé kell aggódnunk a bonyolult öröklési rendszer miatt, hiszen az objektumok eljárásoknak történő átadása többé nem a többalakúságtól függ. Így egyszerűbb és könnyebben újrafelhasználható kódhoz jutunk.
A Ruby-eljárások megadása ismerősnek fog tűnni a Python-programozók számára. A két nyelvben a megadás tulajdonképpen azonos módon történik, a választható értékek használata által szintén ugyanúgy járunk el. Ha egy érték választható, akkor az eljárás meghívásakor elhagyható. Az érték elhagyása esetén az eljárás az érték alapbeállításával hívódik meg.
A Ruby-eljárások megadásában a visszatérési érték sem szerepel. Mivel a nyelv dinamikusan típusos, nem szükséges megadni a visszaadott érték típusát. Hacsak egy visszatérési értéket nem explicit módon a `return` kifejezéssel adunk meg, akkor – miként a Lisp nyelvben – az eljárás legutoljára kiértékelt kifejezését kapjuk vissza.
Az eljáráshívások tulajdonképpen a célobjektumnak küldött üzenetek. Ez ***Smalltalk***-örökség. A *`cØl. zenet(ØrtØkek)`* formátumú kifejezések valamennyi objektumközpontú programot alkalmazó programozó számára ismerősek. Egy objektumnak küldött üzenet az objektum egy adott eljárását hívja meg. Az objektumok között zajló valamennyi kapcsolattartás üzenetek formájában történik.
A Rubyban az eljárásoknak kétfajta megnevezése létezik: osztályeljárások és példányeljárások, vagy röviden csak eljárások. A példányeljárásokat példánnyá vált osztályokban, elterjedt megnevezéssel élve objektumokban hívjuk meg. Az osztályeljárásokat olyan osztályokban hívjuk meg, amelyekből nem származtattunk példányt, ezek olyanok, mint a Java vagy a C++ statikus eljárásai. Ezért az osztályeljárásokat programkönyvtár-eljárásoknak is felfoghatjuk. Ezek az eljárások nem tudnak az objektumok tagváltozóival dolgozni.

Konténerek

Vizsgáljuk meg a programunkat meghívó alábbi kódot, amely a parancssorban adott értékeket dolgozza fel:

```
path = ARGV.shift or raise
➥"HiÆnyzik az œtvonal"
age  = ARGV.shift or raise "HiÆnyzik a kor"
```

A program meghívásakor a parancssorban átadott paramétereket az ARGV tömb objektum tárolja. A `shift` eljárás visszaadja a tömb első elemét, miközben azt a tömbből eltávolítja. A Ruby fejlett tömbosztállyal rendelkezik. A tömbök dinamikusak: képesek a saját méretük megváltoztatására. Mivel a tömbök is objektumok, sem a memóriafoglalások, sem a tartományon kívüli véletlen elemelérések nem okoznak gondot. Az osztály eljárásai egy tömb feldolgozását az indexei vagy az elemei segítségével teszik lehetővé, de úgy is dolgozhatunk velük, mintha vermek, halmazok vagy sorok lennének. A tömbök megfordíthatóak és rendezhetők. A tábla kezeléshez a hash osztályt alkalmazzuk. Az *1. listából* vett következő sor jól mutatja, mennyire elegánsan kezeli is a Ruby a tömböket:

```
Dir.entries(full_name) - [·.·, ·..·]).empty?
```

A `Dir.entries(full_name)` egy tömbbel tér vissza, amely a könyvtár összes állományát tartalmazza. Ezután a `-` műveletjel segítségével az állományok listájából kivonjuk a `[·.·, ·..·]` tömböt. Majd meghívhatjuk az `isEmpty` eljárást, hogy megvizsgáljuk, üres-e a könyvtár. Ha a tömb üres, vagyis az `isEmpty` igaz logikai értékkel tér vissza, a könyvtárban nem található több állomány.

Hibakezelés

Miután a meghíváskor átadott paraméterek feldolgozásra kerültek, a `delete_older` eljárás hívódik meg:

```
delete_older(path, age_in_seconds)rescue
➥puts "Error: #$!"
```

Előfordulhat, hogy a végrehajtás során hiba lép fel. Ha például a programnak kapcsolóként nem létező könyvtárat adtunk meg, hiba keletkezik, amikor a Ruby a legelső alkalommal próbálja a `Dir` osztály valamelyik eljárását meghívni. A fenti kód nemcsak a `delete_older` eljárást hívja meg, hanem azokat az esetleges hibákat is kezeli, amelyek a futtatás során előállnak. Ebben a `rescue` utasítás játssza a kulcsszerepet. Amikor hiba lép fel, a Ruby-értelmező a hibát egy úgynevezett kivételobjektumba csomagolja. Ez az objektum végiglépked az előzőleg lezajlott hívásokat tartalmazó vermen, egészen addig, amíg olyan kódot nem talál, amely képes az adott típusú kivételt kezelni. Ha a kivétel ilyen módon nem kezelhető, a program futása abnormálisan megszakad. A keresés eredménye az `stderr`-re íródik. Ez a módszer nem hibakódokat ad vissza, mint a héjprogramok vagy a C nyelv, ezáltal kevesebb beágyazott utasítást, spagettilogikát és egyszerűen jobb hibakezelést eredményez.

Ha a `rescue` utasítás mellett az `ensure` parancsot is használjuk, elérhetjük, hogy egy adott kódrészlet minden körülmények között lefusson. Kapcsold össze azt a lehetőséget, hogy saját kivételeket írhatsz, hogy kivételek létrehozására készítheted fel a kódodat (a `raise` utasítás segítségével, ahogy a ***Konténerek*** nevű lista is mutatja), valamint azt a képességet, hogy a kivételekből a program egy adott kódrészlet lefuttatásával vagy a hibát okozó kód újrafuttatásával felépülhet – és máris a legnagyszerűbb hibakezelő szerkezet áll a rendelkezésedre, amivel csak valaha dolgod volt.

Fejlettebb tulajdonságok

Térjünk vissza a `delete_older` eljáráshoz, hogy szemügyre vegyük a Ruby néhány fejlettebb szolgáltatását (lásd 33. CD Magazin/Ruby könyvtár 2. lista).

A második sorban a `Dir` osztályon belül a `foreach` utasítást hívjuk meg; a `foreach` az iterációs tervezési minták egyik megvalósítása. Ha objektumközpontú programozással foglalkozol, de még nem olvastad a Négyek bandájának úttörőnek számító könyvét, a ***Tervezési minták***-at, akkor jobban teszed, ha minél hamarabb beszerzel egy példányt. Az interátor nem az egyetlen minta, amely a Ruby nyelvben beépítetten megtalálható. A singleton, publisher/subscriber, vizitor és delegációs minták szintén megvalósítottak. Szükség esetén további minták adhatók a nyelvhez, de az előbb felsoroltak a nyelv alapvető részét képezik.

A `foreach` utasítás egy könyvtár összes állományát bejárja. A `foreach` meghívása után egy kódblokkot látunk, amelynek a kezdete és vége igencsak hasonlít a szabványos Java vagy C++ kódblokkokhoz. A kapcsos zárójelek közötti kódrészletet blokknak hívjuk, amely olyasmi, mint egy eljáráson belüli eljárás. A blokk sosem akkor hajtódik végre, amikor a program futása eléri a blokkot. Amikor a `foreach` eljárás beolvasta a könyvtár egy állományát, a vezérlést átadja a blokknak. A blokkon belüli kód ekkor végrehajtódik, majd a vezérlés visszakerül a `foreach`-re, ami újabb állományt olvas be a könyvtárból. Ez a folyamat addig ismétlődik, míg a ciklus végül a könyvtár összes állományán végiglépked.

Ahelyett, hogy az iterátorok működését segítő osztályokat kellene írnunk, mint a Java vagy a C++ nyelvben, a Ruby rendelkezik a `yield` utasítással, amely lehetővé teszi, hogy az iterátorokat eljárásként valósítsuk meg. Ez remek példája annak, mennyire kifejező és rugalmas a nyelv. Azáltal, hogy egy ötlet megvalósításakor nem kell az alapokhoz visszanyúlnunk, a Ruby lehetővé teszi, hogy magára a feladatra összpontosíthassunk.

Mint korábban már említettük, az eljárások hívása a célobjektumnak küldött üzenettel történik `cØl. zenet` formátumban. A cél megadása nélkül csak az adott osztály helyi eljárásait érhetjük el. Programunk a célobjektum megadása nélkül hívja meg a `puts` utasítást, ami a rendszermag egyik eljárása. Mivel nem adtunk meg célobjektumot, és a `puts` nem helyi eljárása annak az objektumnak, amiből hivatkozunk rá, honnan tudja az értelmező, hogy melyik eljárást hívtuk meg?

A varázsszó a bekeverés (Mix-in). A bekeverés alapvetően azt teszi lehetővé, hogy valamely osztályból öröklődés nélkül hívhassunk meg egy olyan eljárást, amely máshol lett megvalósítva (a bekeverésről bővebben a Linuxvilág 2001. áprilisi számának 47. oldalán olvashatunk, a Mix-in osztálytípus című cikkben). A bekeverés nem új ötlet, és nem is kizárólag a Ruby sajátja. De feltétlenül nagy szerepe van abban, hogy a Rubynak tiszta, érthető írásmódja van.

Még csak nem is remélhetem, hogy egy ekkora terjedelmű cikkben a Ruby összes jellemzőjét be tudom mutatni. Inkább arra bátorítalak, hogy olvasd el ***David Thomas*** és ***Andrew Hunt*** Ruby-programozás (Programming Ruby) című könyvét, amiben bőséges és részletes magyarázatot találsz az olyan fogalmakra, mint modulok, álnevek (alias), névterületek, megjegyzések, dinamikus eljáráshívások, rendszerkapcsolodások, elosztott programozás és hálózati programozás. Érdemesnek tartom még megemlíteni, hogy a Ruby épp olyan jó szabályos kifejezéstámogatással bír, mint a Perl, a CGI-t is támogatja, sőt saját Apache-modulja van, a mod_ruby.

Összegzés

A Ruby csupán újabb tagja lenne a parancsnyelvek családjának? Biztosan nem. Valami más, több, újabb és izgalmasabb jelent meg a nyílt forrást fejlesztők japán berkekből. A Ruby a programozók legjobb barátja. Bár elsősorban parancsnyelvnek tervezték, bebizonyította, hogy nagyobb munkafolyamatokban is megállja a helyét. Olyan izgalmas jellemzőkkel bír, amelyeket a választható programnyelvekben csak mostanság kezdenek el megvalósítani. A Ruby tehát mindenképpen megér egy próbát.

Köszönetnyilvánítás

Különleges köszönet szaklektoraimnak: ***Armin Roehrl***-nek az ➲ http://approximity.com-tól, amiért átnézte a kéziratot, és a végső változat szerkesztésében segítségemre volt. ***David Thomas***-nak a ➲ http://pragmaticprogrammer.com-tól, amiért nagymértékben javított az eredeti mintaprogramon, és átnézte a kéziratot. ***Kent Dahl***-nak és ***Sean Chittenden***-nek a kézirat átnézéséért. Végül, de nem utolsósorban ***Magnus Lie Hetland*** Python-gurunak az értékes segítségért.

Linux Journal 2002. március, 95. szám

Thomas Österlie
tanácsadó a norvégiai telephelyű ConsultIT A/S cégnél, ahol a Unix-kiszolgálókon rendszerfejlesztéssel és számítógépes biztonsággal foglalkozik.