Programování v prostředí Cocoa (10)

Třídy Foundation Kitu

Minule jsme si ukázali přehled všech tříd Foundation Kitu a stručně jsme probrali účel a základní využití většiny z nich. Nyní se soustředíme pouze na ty nejčastěji využívané, zato se na ně podíváme podrobněji. Pro příklady použijeme Objective C. Cocoa umožňuje stejně dobře i využití jazyka Java, ale Objective C je šikovnější.

NS(Mutable)Array

Pro ilustraci základního využití tříd NS(Mutable)Array si ukážeme (nikoli ideální) implementaci triviálního N-árního stromu, tj. datové struktury, jaká je na obrázku, v níž má každý objekt libovolně mnoho "následníků". V C++ (ostatně ani v Objective C bez beztypových kontejnerů) by to nebylo možné bez vlastní nové třídy; v Objective C s využitím beztypových kontejnerů Foundation Kitu je to snadné a bohatě na to stačí samotná třída NSArray (přesněji řečeno NSMutableArray, protože chceme, aby strom byl dynamicky měnitelný).

Pro implementaci využijeme jednoduchého triku, umožněného právě beztypovostí kontejnerů: domluvíme se, že každý uzel stromu bude reprezentován objektem třídy NS(Mutable)Array s tím, že skutečný obsah uzlu bude vždy prvým objektem v poli. Ostatní objekty – budou-li takové – pak budou podřízené uzly. Použijeme-li tedy celkem přirozený zápis využívající závorek pro reprezentaci pole (takže prázdné pole bychom mohli vyjádřit výrazem (), pole obsahující jediný objekt X by se dalo znázornit jako (X), pole obsahující objekty X a Y pak (X,Y)), mohli bychom celý strom podle obrázku zapsat jako

(A,(B,(E),(F)),(C),(D,(G),(H),(I),(J)))

Ukažme si nyní možnou implementaci základních služeb pro práci se stromy – pro jednoduchost to budou prosté "céčkové" funkce (v podrobnějším textu na Chip CD je uveden i velmi pohodlný způsob, jak v Objective C zajistit plně objektové rozhraní):

```
id newTree(id contents) { return [NSMutableArray arrayWithObject:contents]; }
```

Prvá funkce je naprosto triviální – vytvoří nový strom s jediným uzlem, obsahujícím zadaný objekt. Díky poloautomatickému garbage collectoru není zapotřebí žádná služba pro zrušení stromu: celá datová struktura bude zrušena automaticky, jakmile ji už nikdo nebude potřebovat. Základ stromu z minulého obrázku bychom tedy mohli připravit příkazem

```
id ourTree=newTree(A);
```

za předpokladu (se kterým budeme pracovat i nadále), že objekty A – J, jež budou uloženy uvnitř stromu, už jsou pod odpovídajícími jmény k dispozici.

```
id contentsOf(id node) { return [node objectAtIndex:0]; }
NSArray *childsOf(id node) { return [node subarrayWithRange:NSMakeRange(1,[node count]-1)]; }
```

Tyto dvě služby vlastně ani nebudeme potřebovat. Ukázali jsme si však jejich implementaci, neboť zajišťují zcela základní operace – získání objektu uloženého v daném uzlu a přístup k podřízeným uzlům pro hierarchické procházení stromem.

```
id addChildTo(id node,id contents) {
 id child=newTree(contents);
 [node addObject:child];
 return child;
}
```

Služba addChild prostě přidá nový podřízený uzel se zadaným obsahem pod daný uzel (toto je triviální implementace; v praxi bychom ji implementovali trochu lépe, ukázka je na konci tohoto příkladu). Přitom rovnou vrátí nově přidaný uzel, takže jej ihned můžeme použít v dalším kódu. Ukažme si příkazy, jež by vytvořily kompletní strom z minulého obrázku:

```
id o=addChild(newTree,B);
addChild(o,E);
addChild(o,F);
addChild(newTree,C);
addChild(o=addChild(newTree,D),G);
addChild(o,H);
addChild(o,I);
addChild(o,J);
```

To je vlastně vše, výše uvedených osm řádků knihovního kódu by stačilo pro základní práci s datovými stromy. Stačilo pouhých osm řádků, nebylo třeba vytvářet žádné nové třídy – něco podobného je možné jen v plně objektovém systému s dobře navrženými knihovnami. Přitom se vyplatí uvědomit si, že díky standardním službám Foundation Kitu jsme navíc úplně "zadarmo" dostali spoustu dalších možností:

* Již jsem se zmínil, že datové stromy (stejně jako všechny ostatní objekty v API Cocoa) jsou díky garbage collectoru automaticky odstraněny, jakmile je už nikdo nepotřebuje. To umožňuje i jejich korektní sdílení mezi různými moduly.

* Sdílení bude pracovat zcela korektně i v případě, že moduly jsou v různých adresových prostorech nebo vůbec na různých počítačích (takto sestavené datové stromy můžeme okamžitě a bez jakéhokoli dalšího programování například předávat mezi klientem a serverem).

* Do stromů můžeme ukládat naprosto bez omezení jakékoli objekty, a tyto objekty mohou být bez omezení sdíleny mezi různými uzly nebo i různými stromy.

* Stromy můžeme okamžitě vzájemně porovnávat standardní službou isEqual (takže [tree1 isEqual:tree2] bude pravda právě v případě, že obsahy obou stromů jsou přesně ekvivalentní).

* Ihned a bez psaní dalšího kódu můžeme vytvářet snímky celých stromů (např. pro implementaci standardní funkce undo) příkazem copy. Díky paradigmatu proměnných a neproměnných objektů je přitom automaticky zajištěno, že obsahy jednotlivých uzlů se budou duplikovat, jen je-li to skutečně zapotřebí.

* Stromy můžeme okamžitě a bez jakéhokoli dalšího programování ukládat do souborů a číst z nich. Pokud budou objekty uložené v uzlech textovými řetězci (nebo jinými objekty ze skupiny obecných datových typů), bude možné je ukládat do textových souborů, které lze číst/editovat externě.

* Stromy "samy od sebe" umějí vypsat svůj obsah ve formátu využívajícím "závorkovou" notaci popsanou výše. Stačí použít například službu "NSLog(@"%@",tree)", a obsah stromu bude vypsán zcela korektně.

* Stromy můžeme volně ukládat do všech kontejnerů. U kontejnerů využívajících hashování (např. NSSet, NSDictionary, slouží-li strom jako klíč) je automaticky zajištěno, že hashování bude blízké optimu (protože strom je vlastně pole a pole je korektně hashováno z využitím hash hodnot svých prvků).

* Kterýkoli uzel stromu může stejně dobře zároveň sloužit jako součást celého stromu i stát zcela samostatně a reprezentovat svůj podstrom. To se v praxi velmi často hodí a my tuto službu získali bez jakéhokoli programování navíc. Takovéto podstromy mohou být opět bez jakéhokoli omezení sdíleny.

Ukažme si pro zajímavost ještě možné implementace některých dalších služeb nad stromem: prvou a nejjednodušší z nich by mohlo být posčítání všech objektů uvnitř stromu; pro procházení využijeme obecnou třídu NSEnumerator:

```
int countOf(id tree) {
 int count=1; // jeden objekt je v tomto uzlu
 NSEnumerator *en=[childsOf(tree) objectEnumerator];

 while (tree=[en nextObject]) count+=countOf(tree);
 return count;
}
```

O mnoho jednodušeji by to už skutečně nešlo. Samozřejmě že místo iterátoru (NSEnumerator) bychom mohli stejně snadno využít indexy. Iterátor je však o něco pohodlnější, snižuje pravděpodobnost chyby a ve složitějším kódu přináší další výhodu – dokud jej užíváme, je procházené pole "přidrženo" službou retain, takže není možné, aby nám někdo jiný sdílená data uvolnil, dokud s nimi pracujeme. Při použití indexů bychom se o to museli postarat sami, iterátor to zajistí zcela automaticky.

O nic složitější nebude ani vyhledání zadaného objektu uvnitř stromu – následující funkce vrátí buď uzel obsahující zadaný objekt, nebo hodnotu nil, pokud ve stromě žádný takový uzel není:

```
id nodeWith(id tree,id contents) {
 NSEnumerator *en=[childsOf(tree) objectEnumerator];

 if ([contentsOf(tree) isEqual:contents]) return tree;
 while (tree=[en nextObject])
 if ((tree=nodeWith(tree,contents))!=nil) return tree;
 return nil;
}
```

Pro další informace se prosím podívejte do podrobnějšího textu na přiloženém Chip CD.

NS(Mutable)Set, NSCountedSet

Jako jednoduchoučkou ukázku služeb knihovní třídy NSSet si předvedeme funkci, jejímž argumentem je pole obsahující naprosto libovolnou skupinu objektů. Funkce vrátí jiné pole, jež bude obsahovat tytéž objekty, ale bez duplicit – každý objekt v něm bude uložen nanejvýš jednou.

Snad každý programátor s rozsáhlejšími zkušenostmi potvrdí, že obdobnou službu potřebujeme v praxi dost často. V klasických API se to většinou řeší tak, že ji pro každý případ programujeme znovu, protože v konkrétních případech – kde není zapotřebí plná obecnost funkce nad zcela libovolnými objekty – bývá její implementace mnohem snazší; obvykle zabere mezi pěti až dvaceti řádky kódu, podle konkrétní situace a sady omezení, jež v ní platí.

Zkuste hádat, kolik řádků zabere zcela obecná implementace bez jakýchkoli omezení v API Cocoa. Ano, skutečně – je to jeden řádek; jednodušeji by to už opravdu ani při nejlepší vůli nešlo:

```
NSArray *removeDups(NSArray *a) { return [[NSSet setWithArray:a] allObjects]; }
```

Příklad použití této služby by mohl vypadat třeba takto (funkci voláme přímo z ladicího programu gdb):

```
(gdb) po a
```

```
(a, b, a, xyz, xyz, (nested, array), a, b, a, xyz, xyz, (nested, array))
(gdb) po removeDups(a)
(xyz, b, a, (nested, array))
(gdb)
```

V našem příkladu pole obsahovalo pouze textové řetězce a vnořená pole; stejně dobře by však funkce pracovala nad libovolnými objekty (včetně např. datových stromů z minulé kapitoly). Je také vhodné si uvědomit, že díky hashování je tato funkce velmi efektivní – pravděpodobně ne tolik, jako kdybychom ji napsali přímo a pilování jejího algoritmu věnovali hodně času –, ale určitě mnohem, mnohem efektivnější než cokoli, co lze napsat byť za stonásobek těch asi deseti sekund, jež byly zapotřebí pro výše uvedenou implementaci.

Jako ilustraci služeb třídy NSCountedSet (a několika dalších) si pro změnu ukážeme kompletní program, který načte daný textový soubor a provede jeho frekvenční analýzu. Kompletní zdrojový kód – bez zvláštních služeb pro vstup či výstup, ale s kompletní analýzou textu včetně třídění výsledků podle četnosti – jsem psal ani ne čtvrt hodiny a stačilo k tomu třicet zdrojových řádků:

```
#import <Foundation/Foundation.h>

int cmpWithSet(id left,id right,NSCountedSet *freq) {
 return [freq countForObject:right]-[freq countForObject:left];
}

int main (int argc, const char *argv[])
{
 NSAutoreleasePool *pool=[[NSAutoreleasePool alloc] init];

 NSString *fname=[NSString stringWithCString:argv[1]];
 NSString *data=[NSString stringWithContentsOfFile:fname];

 NSLog(@"Scanning \"%@\" (%d bytes)...",fname,[data length]);
 if (!data) NSLog(@"Cannot properly open \"%@\"",fname);
 else {
 NSScanner *sc=[NSScanner scannerWithString:data];
 NSCharacterSet *wordDelims=[NSCharacterSet characterSetWithCharactersInString:@" ,.?!;:\"\'/-
()0123456789*#@\\\r\n"];
 NSCountedSet *freq=[NSCountedSet set];
 NSArray *a;
 int q,i;

 while (![sc isAtEnd]) {
   NSAutoreleasePool *pool=[[NSAutoreleasePool alloc] init];
   NSString *str;
   [sc scanCharactersFromSet:wordDelims intoString:NULL]; // skip any delimiters
   if ([sc scanUpToCharactersFromSet:wordDelims intoString:&str])
   if ([str length]>3) [freq addObject:str];
   [pool release];
 }
 NSLog(@"done, found %d words",[freq count]);
 a=[[freq allObjects] sortedArrayUsingFunction:(int(*)(id,id,void*))cmpWithSet context:freq];
 NSLog(@"sorted, first ten:");
 if ((i=10)>[a count]) i=[a count];
 for (q=0;q<i;q++)
   NSLog(@"%@ (%d)",[a objectAtIndex:q],[freq countForObject:[a objectAtIndex:q]]);
 }

 [pool release];
 exit(0); // insure the process exit status is 0
 return 0; // ...and make main fit the ANSI spec.
}
```

Jistěže řada drobností by se dala vylepšit (např. minimální délka slova by měla být parametrizovatelná, a ne pevně 4, mělo by být možné zvolit kódování českých znaků na vstupu, pro výstup by se mělo využít lepší formátování než triviální NSLog,které, jak uvidíme níže, zobrazuje Unicode znaky dost nečitelným způsobem, apod.). Přesto je program již v této podobě velmi použitelný.

Ačkoli neuvádím přesný popis jednotlivých použitých služeb, mám za to, že by program měl být i tak dost snadno srozumitelný – jelikož jména služeb Foundation Kitu jsou víceméně čitelná v obyčejné angličtině, neměl by být problém program pochopit pro kohokoli, kdo má alespoň základní zkušenosti s programováním

a rozumí anglicky. Přesto, pokud by bylo na implementaci cokoli nejasného, rád podám podrobnější vysvětlení, napíšete-li mi na adresu cocoa@ocs.cz.

Programátoři v jazyce C++ a podobných, stejně jako na opačném konci palety jazyků uživatelé SmallTalku, patrně budou pochybovat o efektivitě takto napsaného programu. Proto jsem mu na zkoušku podstrčil text Bible, který má přes čtyři megabajty. Frekvenční analýza zabrala méně než půl minuty, třídění výsledků trvalo asi jednu sekundu – jak je dobře vidět z časových značek, které služba NSLog automaticky používá:

```
Nov 15 05:00:52 Scanning “/Local/Users/ocs/Library/OpenUp/Bible_0/Cznwt.txb" (4325411 bytes)...
Nov 15 05:01:19 done, found 49413 words
Nov 15 05:01:20 sorted, first ten:
Nov 15 05:01:20 jeho (4912)
Nov 15 05:01:20 (3986)
Nov 15 05:01:20 jsem (3539)
Nov 15 05:01:20 jako (3468)
Nov 15 05:01:20 Jehova (3053)
Nov 15 05:01:20 kte(2550)
Nov 15 05:01:20 jejich (2195)
Nov 15 05:01:20 kter(2138)
Nov 15 05:01:20 bude (1873)
Nov 15 05:01:20 p(1846)
```

Shrnutí

Dnes jsme si ukázali několik praktických ukázek využití standardních tříd NSArray, NSEnumerator, NSSet, NSCountedSet a – bez podrobnějšího výkladu – také NSScanner, NSCharactersSet a NSString. Příště se podíváme na několik dalších podobně šikovných a také velmi často využívaných tříd. Ke třídě NSString, jež je snad nejpoužívanější třídou Foundation Kitu vůbec, se ještě vrátíme v podrobnějším samostatném odstavci.

Ondřej Čada