Programování v prostředí Cocoa (6)

Základy Foundation Kitu 2

V minulém Chipu jsme si ukázali základní chování všech objektů ve vývojovém prostředí Cocoa: podrobně jsme si vysvětlili, jak a kdy objekty za asistence poloautomatického garbage collectoru zanikají. Dnes se podíváme na dvě paradigmata, jež zvyšují efektivitu programů a zároveň usnadňují jejich psaní: jedním z nich je koncepce měnitelných a neměnných objektů, druhým skryté podtřídy.

Měnitelné a neměnné objekty

Základní myšlenkou koncepce měnitelných a neměnných objektů je dosažení vyšší efektivity, aniž by se o to programátor musel vědomě starat. Typickým příkladem je kopírování objektů: v praxi poměrně často potřebujeme vytvořit privátní kopii objektu – jakýsi jeho "snímek", který uchová momentální stav objektu i v případě, že se původní objekt změní. Představme si například objekt, který reprezentuje hašovací tabulku – takový objekt v Cocoa skutečně existuje a jmenuje se NSDictionary. Základní dvě zprávy, které je schopen zpracovat, jsou:

```
- (void)setObject:(id)anObject forKey:(id)aKey;
- (id)objectForKey:(id)aKey;
```

První z nich uloží do tabulky dvojici <klíč, hodnota>, druhá vyhledá hodnotu k zadanému klíči (v čase nezávisejícím na počtu hodnot v tabulce). Je zřejmé, že má-li hašovací tabulka být konsistentní, musí interně udržovat ne odkazy na klíče, ale jejich neměnné kopie – kdyby v tabulce byly jen odkazy na klíče, mohl by se objekt reprezentující klíč kdykoli změnit, aniž by se o tom tabulka vůbec "dozvěděla"; hašovací tabulka by v takovém případě byla samozřejmě nekorektní. Implementace metody setObject:forKey: tedy musí vypadat přibližně takto:

```
- (void)setObject:(id)anObject forKey:(id)aKey
{
  id myKey=[aKey copy]; // potřebuji vlastní neměnnou kopii
  id myVal=[anObject retain]; // hodnota se může klidně měnit (ale nesmí zaniknout)
  zařadit_do_tabulky(myKey,myVal);
}
```

Za těchto podmínek bude hašovací tabulka pracovat korektně, ovšem zaplatíme za to zpomalením programu a větší spotřebou paměti: každý klíč vkládaný do tabulky se musí nejprve zkopírovat – to znamená, že potřebujeme dvakrát tolik paměti a navíc program musí kopírovat data objektu. Přitom to v řadě případů není doopravdy zapotřebí: velmi často (v praxi téměř vždy, protože klíče obvykle bývají textové konstanty) se obsah klíčů stejně nebude měnit. Hašovací tabulka by si tedy mohla udržovat pouze odkazy na klíče – musela by ale "vědět", které klíče se ještě mohou měnit a které ne.

Objektové prostředí ale nabízí velmi elegantní řešení: hašovací tabulka samozřejmě nemůže vědět, které objekty se budou měnit; mohou to ale vědět tyto objekty samy! Stačí zavést pro každý typ objektů, pro který to dává rozumný smysl, dvě třídy: třídu neměnných objektů a třídu objektů, které se mohou měnit – například NSString (neměnné) a NSMutableString (měnitelné). Neměnné objekty pak nemusejí nikdy vytvářet kopie – jejich metoda copy může být implementována takto:

```
@implementation NSString
...
-copy
{
  return [self retain];
}
...
@end
```

Nyní funguje vše automaticky s nejvyšší možnou efektivitou: vkládáme-li do hašovací tabulky klíč, který se nikdy nebude měnit, hašovací tabulka bude udržovat pouze odkaz – žádná paměť navíc, nic se nekopíruje. Pouze v případě, že jako klíč využijeme měnitelný objekt (např. NSMutableString), kopie se vytvoří; v takovém případě se tomu ale stejně nemůžeme vyhnout. Navíc tentýž trik automaticky funguje nejen v hašovací tabulce, ale kdekoli, kde potřebujeme okamžité kopie objektů. Připravujeme například program, který si pro funkci undo musí zapamatovat momentální stav svých datových objektů? Nic

jednoduššího – prostě vytvoříme kopie všech objektů reprezentujících data tak, že jim pošleme zprávu copy. Díky koncepci měnitelných a neměnných objektů nemusíme zkoumat, která data se mohou měnit a která ne – fakticky se zkopírují jen ta, kterých se změny mohou týkat.

Cocoa proto v řadě případů nabízí dvojice tříd NSXXX a NSMutableXXX, kde objekty třídy NSXXX se nemohou měnit, zatímco objekty třídy NSMutableXXX ano (je tomu tak mimochodem i u třídy NSDictionary – metoda setObject:forKey: je tedy samozřejmě k dispozici pouze u objektů třídy NSMutableDictionary). Třída NSMutableXXX je vždy dědicem třídy NSXXX; měnitelné objekty tedy "umějí" všechno, co neměnné, a navíc jsou schopny změn. Pošleme-li kterémukoli objektu třídy NSXXX zprávu copy, nevytvoří se žádná kopie; namísto toho získáme další odkaz na tentýž (neměnný) objekt. Pošleme-li však zprávu copy objektu třídy NSMutableXXX, dostaneme nový objekt třídy NSXXX, který bude obsahovat neměnnou kopii momentálního stavu původního objektu.

Uvědomme si, že koncepce měnitelných a neměnných objektů zaručuje co nejefektivnější zkopírování i u složených objektů. Jako příklad vezměme objekt třídy NSMutableArray, který reprezentuje pole libovolných dalších objektů, do nějž můžeme přidávat nebo z něj odebírat (odpovídající neměnná třída NSArray reprezentuje pole, jehož obsah nemůžeme měnit). Obr. 1 ukazuje příklad objektu třídy NSMutableArray, obsahujícího (odkazy na) jak měnitelné, tak neměnné objekty. Vyžádáme-li si nyní zprávou copy neměnnou kopii momentálního stavu tohoto objektu, musí se vytvořit nový objekt třídy NSArray (protože existující objekt mutableArray je měnitelný) se stejným (a neměnným) obsahem. Nový objekt tedy může se starým sdílet odkazy na neměnné vnořené objekty, ale musí obsahovat vlastní (neměnné) kopie objektů, které byly měnitelné. Výsledek vidíme na obr. 2.

Čas od času bychom mohli potřebovat "přece jen" změnit neměnný objekt. Doslova to samozřejmě není možné – tím bychom celou koncepci měnitelných a neměnných objektů postavili na hlavu. Můžeme si však pomocí zprávy mutableCopy vyžádat vytvoření měnitelné kopie objektu. Obsahuje-li původní objekt vnořené objekty, bude jeho měnitelná kopie obsahovat (odkazy na) tytéž objekty, a to i v případě, že tyto objekty samy jsou neměnné (chceme-li např. vytvořit měnitelnou kopii pole, je to proto, abychom do něj mohli přidávat nebo z něj odebírat další objekty; ne proto, abychom mohli měnit objekty v něm obsažené). Výsledek vytvoření měnitelné kopie pole z minulého příkladu ukazuje obr. 3.

Koncepce měnitelných a neměnných objektů je velmi silným a šikovným mechanismem, který kromě výrazného zvýšení efektivity programů dokáže i omezit programátorské chyby. Používáme-li neměnný objekt, nemůže se nám omylem stát, že jej některý úsek programu změní (z podobného důvodu byl např. v ANSI C zaveden modifikátor const). Rozdělení měnitelných a neměnných objektů na samostatné třídy NSXXX a NSMutableXXX navíc umožňuje některé takové chyby odchytit již při překladu – pokusíme-li se například staticky typovanému objektu třídy NSArray poslat zprávu addObject:, překladač vydá varování.

Skryté podtřídy

Zatímco koncepce měnitelných a neměnných objektů trochu zkomplikovala programátorské rozhraní Cocoa (namísto jediné třídy např. NSString máme dvě – NSString a NSMutableString) pro zajištění větší efektivity a větší robustnosti, je hlavním účelem koncepce skrytých podtříd programátorské rozhraní bez ztráty efektivity co nejvíce zjednodušit (nebo naopak – při zachování jednoduchého a přehledného API dosáhnout maximální efektivity).

Koncepci skrytých podtříd si opět ukážeme na příkladu. Dejme tomu, že chceme vytvořit třídu, jejíž instance by reprezentovaly čísla (taková třída je součástí Cocoa a jmenuje se NSNumber). Pokud bychom nevyužili koncepce skrytých podtříd, máme v podstatě dvě možnosti:

1. Vytvoříme třídu NSNumber, která bude sama o sobě schopna pracovat s jakýmkoli typem čísla (char, int, unsigned, long, 64 bitů, float...). To je samozřejmě možné, ale tento přístup má dvě nevýhody: naprogramování takové komplikované třídy je složité, snadno se při něm udělá chyba a složitý zdrojový kód se špatně udržuje. Druhou (a možná závažnější) nevýhodou je, že implementace takové třídy není efektivní, protože musí zahrnovat potřeby všech číselných typů a nemůže být optimalizována pro potřeby jednoho konkrétního typu.

2. Třída NSNumber sama bude pouze abstraktní nadtřídou, shrnující obecné vlastnosti všech čísel, a skutečnými reprezentanty jednotlivých typů budou její podtřídy – asi tak, jak naznačuje obr. 4. To je lepší, skutečně objektové řešení – každá z podtříd je jednoduchá, snadno udržovatelná a snadno může být také maximálně optimalizována. Nepříjemnou nevýhodou však je velmi komplikované programátorské rozhraní – programátor si musí pamatovat jakési třídy NSCharNumber, NSUnsignedCharNumber... a musí se sám starat o to, aby se použila potřebná třída. To je nepohodlné a v objektovém prostředí je to dokonale zbytečné.

Koncepce skrytých podtříd je jednoduchoučký a přitom nesmírně efektivní trik: vlastně se využívá implementace podle bodu 2, ale API programátorům nabízí pouze rozhraní podle bodu 1. Programátor tedy využívá vždy jen a pouze služeb třídy NSNumber a její podtřídy vůbec nezná (jejich konkrétní počet a druhy dokonce vůbec nejsou součástí API a snadno se mohou měnit mezi jednotlivými verzemi systému, bez jakýchkoli záporných důsledků pro kompatibilitu programů). Třída NSNumber sama při vytváření objektu rozhodne, která z jejích (skrytých) podtříd je pro dané číslo optimální, a vytvoří odpovídající objekt; i s ním programátor komunikuje jako s objektem třídy NSNumber (což je v naprostém pořádku, protože objekt je dědicem třídy NSNumber). Tuto situaci ilustruje poslední, pátý obrázek.

Vytvoříme-li tedy několik “instancí třídy NSNumber" takto:

```
NSNumber *aChar = [NSNumber numberWithChar:”a”];
NSNumber *anInt = [NSNumber numberWithInt:1];
NSNumber *aFloat = [NSNumber numberWithFloat:1.0];
NSNumber *aDouble = [NSNumber numberWithDouble:1.0];
```

může být ve skutečnosti každý z nově vytvořených objektů instancí jiné třídy. Všechny však jsou dědici třídy NSNumber a jako s takovými s nimi můžeme pracovat.

Stojí za to si uvědomit, že toto skvělé řešení je v jazycích typu C++ trochu problematické: jde o špatně navržený systém tvorby objektů – konstrukce “new NSNumber" v C++ prostě nemůže vytvořit objekt jiné třídy než právě třídy NSNumber. Skryté podtřídy zde nelze použít (je možné to do jisté míry dohnat pomocnou statickou metodou, tam však zase nastanou problémy s nemožností tyto metody dědit). Tuto nešťastnou koncepci z C++ bohužel do značné míry přebral i jinak velmi dobrý objektový jazyk Java.

Cocoa využívá koncepce skrytých podtříd velmi často. Právě díky tomu je API Cocoa mnohem jednodušší a přehlednější než například “C++kové” API operačního systému Epoc, přestože služby Epocu nabízejí jen zlomek luxusu a flexibility služeb Cocoa. Typickým příkladem skrytých podtříd jsou prakticky všechny třídy Foundation Kitu, které reprezentují složené objekty (jako NSArray nebo NSDictionary) – ty využívají skrytých podtříd pro volbu optimální implementace z hlediska poměru efektivity a paměťové náročnosti, aniž by se tím musel programátor explicitně zabývat. Programátor nadto samozřejmě může v případě potřeby snadno sám doplnit další skryté podtřídy pro rozšíření služeb celé skupiny tříd.

Shrnutí

Dokončili jsme přehled základních vlastností objektů Cocoa především z hlediska doby jejich života; již víme, kdy a jak objekty v systému Cocoa zanikají. Dnes jsme se navíc seznámili s některými dalšími paradigmaty, jež zajišťují vysokou efektivitu při udržení jednoduchosti a přehlednosti API.

Příště si zběžně ukážeme konkrétní prostředí pro programátorskou práci, jež Cocoa nabízí – aplikaci ProjectBuilder.

Ondřej Čada