Programování v prostředí Cocoa (5)

# Základy Foundation Kitu

**Minule jsme dokončili popis jazyka Objective C. Dnes si řekneme více o základních vlastnostech všech objektů; už se to však nebude týkat libovolného prostředí, ale jen systémů využívajících Foundation Kit (jako je OpenStep, Cocoa, GNUStep nebo třeba vývojové prostředí XSdk pro Epoc).**

Řada konkrétních služeb, o kterých budeme hovořit, totiž vyžaduje podporu standardních knihoven – například poloautomatický garbage collector nemůže fungovat bez třídy NSAutoreleasePool nebo nějakého jejího ekvivalentu. Především se soustředíme na to, jak a kdy objekty zanikají (jejich vznik již známe, jsou podle potřeby vytvářeny třídami).

Jako obvykle, podrobnější verzi tohoto textu s více příklady opět naleznete na Chip CD (tentokrát navíc ještě s několika doplňky a rozšířeními předchozích dokumentů). Z prostorových důvodů také bude tento článek rozdělen na dvě části, z nichž druhá vyjde v příštím čísle.

## Druhy objektů

Velmi důležitým atributem kteréhokoli objektu je doba jeho existence. Kdy objekt vznikne? Kdy zanikne? Je jeho vznik – nebo zánik – vedlejším efektem některé jiné akce, nebo si jej musí programátor vyžádat? Z tohoto hlediska můžeme objekty rozdělit v zásadě do čtyř skupin. První tři skupiny dobře známe: odpovídají trvání proměnných ve standardních programovacích jazycích. Objektovou novinkou je čtvrtá skupina – objekty, které dokáží “přežít” i ukončení procesu, který s nimi pracuje.

* Automatické objekty jsou objekty s obecně nejkratší dobou života (i když v konkrétních případech mohou samozřejmě dynamické objekty existovat kratší dobu) a v neobjektových prostředích jim zhruba odpovídají lokální proměnné. Automatický objekt vznikne na základě požadavku programu; často tento požadavek musí být určen staticky v okamžiku překladu. Automatický objekt – jak jeho jméno naznačuje – zaniká automaticky ve chvíli, kdy program opustí blok, v němž byl automatický objekt vytvořen. Objektový systém nemusí podporovat automatické objekty; namísto nich mohou stejně dobře posloužit dynamické. Není-li však součástí systému tzv. garbage collector (viz níže), může být někdy programování v systému bez automatických objektů docela nepohodlné.

* Dynamické objekty jsou základním typem objektů a z hlediska doby trvání jim v neobjektových prostředích nejblíže odpovídají bloky paměti, alokované příkazy malloc, calloc, new a podobně. Vznik i zánik dynamického objektu je vždy výsledkem explicitního požadavku programátora (není-li součástí systému samostatný modul – tzv. garbage collector – který může rušit dynamické objekty “automaticky”, usoudí-li, že je již nikdo nebude potřebovat). Nevyžádá-li si nikdo zrušení dynamického objektu, zanikne objekt nejpozději při ukončení procesu, jehož byl součástí. Bez podpory dynamických objektů se neobejde žádný objektový systém.

* Statické objekty trvají po celou dobu existence procesu a jejich ekvivalentem v neobjektových prostředích jsou globální proměnné. Statický objekt vznikne ve chvíli vytvoření procesu – de facto tedy musí být vytvořen již při překladu – a zaniká vždy ve chvíli zániku procesu. Objektový systém nemusí podporovat práci se statickými objekty; v takovém případě však musí nabízet i neobjektové služby pro prvotní vytváření dynamických objektů. V některých případech může podpora statických objektů také usnadnit programování.

* Trvalé objekty jsou vytvořeny i zrušeny na základě požadavku programátora. Speciálně trvalé objekty “přežijí” i ukončení procesu, který je vytvořil; trvalý objekt, který nikdo nezrušil, bude existovat navěky (přesněji řečeno, po celou dobu existence výpočetního systému, v němž trvalý objekt leží). Nejbližším ekvivalentem trvalých objektů v neobjektových prostředích jsou datové soubory. Objektový systém nemusí vůbec podporovat trvalé objekty, ochuzuje tím však programátory o velmi široké možnosti jejich využití.

Pro rozhodnutí o typech objektů, které bude vývojové prostředí podporovat, existují dvě protichůdné tendence: na jednu stranu je výhodné umožnit práci s co nejširší paletou možných typů, aby programátor měl k dispozici flexibilní aparát služeb, a na druhou stranu existence řady různých typů objektů komplikuje programátorské rozhraní a zvyšuje pravděpodobnost chyb.

Cocoa proto vůbec nepodporuje automatické objekty (obsahuje však jednoduchý, ale efektivní poloautomatický garbage collector, který je z programátorského hlediska dokáže plně nahradit). Podpora statických objektů je omezena pouze na třídy (připomeňme, že třídy v Objective C slouží především pro tvorbu nových objektů – musejí tedy samy být statické, protože jinak bychom po spuštění programu neměli k dispozici nic, co by objekty dokázalo vytvořit) a na výjimečné speciální případy, usnadňující programování.

## Automatické objekty

Objective C automatické objekty nepodporuje. Díky existenci garbage collectoru však můžeme s dynamickými objekty pracovat přesně stejně jako s automatickými:

```
{ // automatický objekt v C++
  Array cppArray(objekt1,objekt2,objekt3,objekt4,NULL);
  ...
  // objekt zanikne automaticky při opuštění bloku
}
```

a odpovídající varianta s dynamickým objektem v Objective C:

```
{
  id anObject=[NSArray arrayWithObjects:objekt1,objekt2,objekt3,objekt4,nil];
  ...
  // objekt zanikne automaticky, jakmile přestane být zapotřebí
}
```

Na rozdíl od automatického objektu je zde však významný rozdíl mezi "při opuštění bloku" a "až přestane být zapotřebí"; speciálně v Objective C je naprosto korektní takovýto objekt předat spolupracujícímu objektu nebo jej vrátit jako návratovou hodnotu:

```
{
  id anObject=[NSArray arrayWithObjects:.....];
  ...
  [jinyObjekt budePracovatS:anObject];
  return anObject;
}
```

S automatickým objektem by něco podobného bylo možné jen za cenu předávání hodnotou, a to je samozřejmě u objektů, jež mohou obsahovat rozsáhlá data, obecně nežádoucí. V Objective C to však funguje korektně i při předávání referencí.

## Dynamické objekty a garbage collector

Dynamické objekty již vlastně známe: objekt je vytvořen na základě explicitního požadavku nějakým jiným objektem (obvykle, ale ne nutně, třídou). Každý řádek v následujícím příkladu vytvoří nový objekt:

```
id image=[NSImage imageNamed:@"....."]; // objekt vytvořen třídou
NSString *desc=[image description]; // objekt vytvořen jiným objektem
NSString *descLwr=[desc lowercaseString]; // objekt vytvořen jiným objektem
```

Součástí API Cocoa je poloautomatický garbage collector. Díky jeho existenci se na dynamický objekt standardně musíme dívat spíše jako na automatický (jak jsme si ostatně ukázali v minulém odstavci): objekt bude jistě existovat po celou dobu zpracování aktuální metody, ale potom jej garbage collector může odstranit. Konkrétně to tedy znamená, že nebudeme-li žádný z objektů vytvořených v posledním příkladu potřebovat později, nemusíme se o jejich uvolnění vůbec starat – garbage collector je uvolní automaticky po ukončení metody, která objekty vytvořila.

Nechceme-li však, aby byl objekt odstraněn, musíme garbage collectoru sdělit, že si nad objektem chceme i nadále udržovat kontrolu (proto hovoříme o poloautomatickém garbage collectoru). To uděláme tak, že objektu odešleme zprávu retain – takový objekt pak bude existovat (nejméně) tak dlouho, dokud jej opět neuvolníme. Předpokládejme, že v minulém příkladu si chceme zachovat poslední textový řetězec descLwr (popis obrázku uvedený malými písmeny), zatímco zbývající dva objekty byly zapotřebí pouze pro jeho získání a již nás nezajímají:

```
[descLwr retain];
```

Po ukončení metody garbage collector uvolní objekty image a desc; objekt descLwr však existuje nadále a můžeme s ním i v budoucnosti volně pracovat. Jakmile zjistíme, že již nebudeme objekt potřebovat, uvolníme jej pomocí zprávy autorelease:

```
[descLwr autorelease];
```

a garbage collector jej zruší po ukončení metody, ve které jsme jej uvolnili.

Je vhodné si uvědomit, že pokud jsme napsali “zruší jej po ukončení metody", neznamená to “zruší jej okamžitě po ukončení metody" – objekt může “přežít” ještě velmi dlouho. Důvod je jednoduchý: dynamické objekty mohou být snadno sdíleny mezi různými moduly nebo různými úseky kódu. S jedním a tím samým objektem descLwr může tedy chtít komunikovat více jiných objektů; každý z nich si může vyžádat udržení

objektu zprávou retain. Garbage collector sleduje, kolikrát objekt dostal zprávu retain, a uvolní jej teprve tehdy, když pro každý retain dostal odpovídající zprávu autorelease.

Poloautomatický garbage collector tohoto typu má řadu výhod. Hlavní z nich je, že se nemusíme explicitně starat o uvolnění sdílených objektů – zcela běžnou situací v objektovém prostředí je, že řada objektů spolupracuje s jedním dalším (viz schéma). Pokud není k dispozici garbage collector, není jasné, který z objektů 1 až 5 má nakonec uvolnit objekt A. Samozřejmě že ten, který jej přestane potřebovat jako poslední; jak to ale v programu zjistit? Tato situace bývá zdrojem častých chyb (kdy si např. objekt 3 myslí, že již nikdo nebude objekt A potřebovat, a tak jej uvolní; pak se ale na – již neexistující – objekt A obrátí ještě objekt 4 a program se zhroutí). Možnost takových chyb garbage collector definitivně odstraňuje.

Nakonec se seznámíme se zprávou release. Zatímco zpráva autorelease řekne garbage collectoru “tento objekt po ukončení této metody nebudu potřebovat", říká zpráva release “tento objekt od této chvíle nebudu potřebovat". Její použití je tedy o něco málo efektivnější, protože objekt se uvolní ihned a neleží v paměti zbytečně po dobu zpracování metody; při jejím používání si však musíme důkladně rozmyslet, víme-li opravdu jistě, že již objekt nebudeme potřebovat.

Podívejme se například na následující úsek kódu:

```
...
aFont=[text font];
[currentFont release];
currentFont=[aFont retain];
...
```

Na první pohled se zdá být vše v pořádku – starý font uvolníme a místo něj si zapamatujeme aktuální. Přesto toto použití zprávy release může snadno vést k chybě: pokud je minulý font stejný jako dosavadní, uvolní se tento objekt ve chvíli provedení metody release a zpráva retain se již pošle neexistujícímu – právě uvolněnému – objektu! Použijeme-li však zprávu autorelease, je vše v pořádku – garbage collector by objekt uvolnil až po ukončení metody (ale neuvolní jej, protože objekt mezitím dostal zprávu retain).

## Statické objekty

Kromě tříd, které jsou všechny standardně statickými objekty, podporuje Objective C pouze statické objekty třídy NSString. Takový objekt vytvoříme zápisem podobné konstanty, jakou určujeme v plain C textový řetězec; před otevírací uvozovkou však umístíme navíc znak '@':

```
id aString=@"Text";
// nebo přímo:
if ([aString isEqualToString:@"xyz"]) ...
```

Objective C automaticky převede ASCII znaky řetězcové konstanty do vnitřního formátu NSStringu, který podporuje Unicode, a vytvoří rovnou při překladu statický objekt třídy NSString s požadovaným obsahem, jenž bude existovat po celou dobu běhu programu.

Za stručnou samostatnou poznámku stojí to, že ačkoli třídy jsou statické, knihovny Cocoa nabízejí prostředky, jak vytvářet za běhu programu dynamicky nové třídy – ať již standardním zavedením dynamické knihovny, nebo dokonce přímo programově. Je tedy snadno možné – a v praxi u rozsáhlejších programových systémů velmi často používané – psát kód tohoto typu:

```
...
class myClass;
if ((myClass=NSClassFromString(@"MyClass"))==nil) {
  // kód pro dynamické zavedení knihovny, obsahující...
  // ...implementaci třídy MyClass
}
// zde již můžeme se třídou volně pracovat
id anObject=[myClass newObject];
...
```

Ve statických neobjektových jazycích typu C++ samozřejmě na něco podobného není ani pomyšlení; naopak v dynamických objektových systémech je to snadné – ačkoli minulý příklad využíval syntaxe Objective C, například v Javě snadno implementujeme totéž.

## Trvalé objekty

Cocoa standardně umožňuje zapsat libovolný objekt na disk a opět jej z disku obnovit (přesněji řečeno, Cocoa podporuje zápis objektu a jeho opětovné obnovení prostřednictvím libovolného zařízení – disk

zajišťuje trvalé objekty, síť předávání objektů mezi počítači a podobně). Opět je třeba neplést si systém perzistentních objektů s jeho nedokonalou náhražkou, již často nabízejí statické jazyky typu C++ a ve které lze obsah objektu zapsat do streamu a naopak nově vytvořený objekt ze streamu inicializovat.

Zásadní rozdíl spočívá v tom, že dynamický systém ukládá na zařízení kompletní informace o objektu včetně jeho třídy; namísto explicitního vytvoření objektu a načtení jeho obsahu tedy prostě řekneme “dej mi objekt" a načte se to, co na zařízení bylo k dispozici – ať je to cokoli. Je zřejmé, že to nesmírně zjednodušuje API a zvyšuje jeho flexibilitu:

```
// vytvoření trvalého objektu
id anyObject=.....;

[NSArchiver archiveRootObject:anyObject toFile:@"jméno souboru"];
...
// načtení trvalého objektu
id object=[NSUnarchiver unarchiveObjectWithFile:@"jméno souboru"];
NSLog("Získali jsme objekt třídy %@",[object class]);
```

Vytváříme-li vlastní třídu objektů, stačí velmi jednoduchým způsobem určit, jakým způsobem bude nový objekt kódován a dekódován (s podrobnostmi se seznámíme později, až budeme popisovat třídy NSArchiver a NSUnarchiver). Všechny standardní objekty knihoven Cocoa samozřejmě zápis a obnovení podporují.

Díky tomu můžeme libovolný objekt nebo skupinu objektů kdykoli zapsat na disk – objekty se tak stanou trvalými – nebo naopak z disku obnovit.

## Shrnutí

Ukázali jsme si základní vlastnosti objektů Cocoa, především z hlediska doby jejich života; již víme, kdy a jak objekty v systému Cocoa zanikají. Příště se seznámíme s některými dalšími paradigmaty, jež zajišťují vysokou efektivitu při udržení jednoduchosti a přehlednosti API.

*Ondřej Čada*