Prosadit svou

Anglické sloveso to assert znamená podle slovníku tvrdit, prosazovat nebo postulovat; v podobě asertivita či asertivní chování je koneckonců známe i v češtině a také další národní jazyky si pro tento význam nějakou odvozeninou z původně latinského kmene vypomáhají. Možná vás však překvapí, že to platí i pro některé jazyky programovací...

Programátoři používající jazyk C znají makro assert, které slouží při ladění; nyní bude podobný nástroj k dispozici také v jazyce Java. Než se k němu dostaneme, zopakujeme si, jak je to s asercemi v C a v C++, abychom mohli obě konstrukce porovnat.

Jazyky C a C++

Jazyk C obsahuje už od dávných dob makro assert()(vzdor nepsanému pravidlu o identifikátorech maker se píše malými písmeny). Jeho deklaraci najdeme v hlavičkovém souboru assert.h, v jazyce C++ vyhovujícím standardu ISO 14882 také v hlavičkovém souboru cassert.

Použití

Použití makra assert v C/C++ vypadá takto:

```
#include <assert.h>
/* ... */
assert(výraz);
```

Zde výraz představuje podmínku, jejíž splnění požadujeme - nebo spíše prosazujeme; tento výraz se za běhu programu vyhodnotí (pokud toto makro nevyřadíme z činnosti, jak si povíme dále). Je-li nenulový, nic se nestane, jinak program vypíše zprávu, která má typicky tvar

Assertion failed: Výraz, file jméno, line číslo.

a okamžitě skončí. (Možná ještě připojí zprávu Abnormal program termination, záleží na překladači.)

To znamená, že chceme-li např. ve zdrojovém souboru C:\WORK\TEST.C na řádku 5 otestovat podmínku x > y, napíšeme

```
assert(x>y);
```

Bude-li tato podmínka splněna, nic se nestane a program poběží v klidu dál. Jestliže však splněna nebude, program v tomto místě vypíše

Assertion failed: x>y, file C:\WORK\TEST.C, line 5.

a skončí.

Po odladění se asercí v programu zbavíme velice jednoduše: do zdrojového souboru vložíme ještě před řádku #include <assert.h> deklaraci

```
#define NDEBUG
```

a tím všechna makra assert() v tomto zdrojovém souboru deaktivujeme. (Pozor: Překladač C/C++ musí nejprve narazit na definici makra NDEBUG a teprve pak na definici makra assert(), jinak aserce z činnosti nevyřadí.)

Implementace

Standard jazyka C předepisuje chování makra assert(), nikoli jeho implementaci; podobné to je i v jazyce C++. To znamená, že v různých překladačích se můžeme setkat s různými implementacemi.

Především je asi jasné, že vyřazení asercí pomocí makra NDEBUG se bude opírat o podmíněnou kompilaci:

```
#ifdef NDEBUG
  #define assert(x) ((void)0)
#else
  #define vlastní_aserce
#endif
```

Je-li definováno makro NDEBUG, nahradí se všechny aserce výrazem ((void)0); připojíme-li za makro assert() středník, stane se z něj výrazový příkaz, který nic nedělá a který bude překladač nejspíš ignorovat. Pokud však makro NDEBUG není definováno, nahradí se assert() příkazem, který způsobí výpis zprávy a ukončení programu.

Pro implementaci vlastní_aserce je obvykle k dispozici pomocná funkce, která se jmenuje _assert(),

_assertfail(), _assert_error() nebo nějak podobně a která má zpravidla prototyp

int _assert(char* vyraz, char* soubor, int radka);

Tato funkce se postará o výpis chybových zpráv a o ukončení programu.

Podrobnější rozbor problémů, na které lze při hledání vhodného tvaru vlastní_aserce narazit, není cílem tohoto článku; zájemce odkazuji na knihu [1], kap. 2.6.

V dnešních překladačích se setkáváme zpravidla s implementací založenou na operátorech ?: nebo ||:

```
#define assert(e)\
  ((e) ? (void)0 : (void)_assert ("Assertion failed: "\
  #e, __FILE__, __LINE__))
```

nebo

```
#define assert(e)\
 (void)((e) || _assert("Assertion failed: "\
  #e, __FILE__, __LINE__))
```

První řešení využívá toho, že při použití operátoru ?: se nejprve vyhodnotí první operand (podmínka), a podle toho, zda je splněna (nenulová), nebo není, se vyhodnotí jen druhý, nebo jen třetí operand. Pokud je podmínka splněna, provede se opět (void)0.

Druhé řešení je založeno na tom, že je-li ve výrazu a || b první operand nenulový, druhý se již nevyhodnotí (tzv. neúplné vyhodnocení logického výrazu). Obě řešení využívají operátor preprocesoru #, který změní parametr e na znakový řetězec, a tak umožní vytisknout zápis podmínky, která nebyla splněna. Zápis

"Assertion failed: " #e

představuje dvě řetězcové konstanty vedle sebe, mezi nimiž jsou pouze bílé znaky, a proto je překladač automaticky spojí v jedinou řetězcovou konstantu.

Starší překladače

Ve starších překladačích jazyka C se lze setkat s implementací vlastní_aserce v podobě

```
#define assert(e) if(!(e)){\
   fprintf(stderr, "Assertion failed: %s,"\
" file %s, line %d",\
#e, __FILE__, __LINE__);\
   exit(1);\
 }\
 else
```

jejíž "výhodou" bylo, že za ní šlo vynechat středník. V C++ se ovšem tato konstrukce může za jistých okolností chovat nepřípustným způsobem. Je-li např. Napis objektový typ, proběhne funkce

```
void f(int x, int y)
{
 assert(x>y); // zde je středník
 Napis n("Ahoj");
 // nějaké další příkazy
}
```

při této implementaci makra assert v C++ jinak než funkce

```
void f(int x, int y)
{
 assert(x>y) // chybí středník
 Napis n("Ahoj");
 // nějaké další příkazy
}
```

která se liší jen vynechaným středníkem za makrem assert(). (V prvním případě se bude destruktor instance n volat až při ukončení funkce f(), ve druhém případě ihned po ukončení konstruktoru; navíc ve druhém případě nebude v dalších příkazech instance n dostupná. To vyplývá z pravidla pro příkaz if v C++, které říká, že příkaz za podmínkou, stejně jako příkaz za else, se vždy považuje za blok, ať je nebo není uzavřen do závorek {}.)

Jazyk Java

Na setkání Java One v San Franciscu na počátku léta 2001 ohlásil James Gosling, hlavní "pachatel" Javy, plánované rozšíření tohoto jazyka: verze JDK 1.4 bude obsahovat mechanismus aserce [2]. Jeho účel je velice podobný účelu asercí v C a C++. Dodejme, že v současné době lze získat beta verzi JDK 1.4 Standard Edition na internetu na adrese [3].

(Na tomtéž setkání J. Gosling také oznámil, že ve verzi JDK 1.5 by měly být k dispozici generické konstrukce, tedy jakási analogie šablon z C++. O tom ale snad někdy později.)

Syntaxe

Použití asercí v Javě je velice podobné jako v jazyce C. Máme na vybranou dvě možnosti:

```
assert výraz1;
assert výraz1 : výraz2;
```

V obou variantách musí být výraz1 typu boolean, jinak překladač ohlásí chybu. Jeho hodnota se vypočte; je-li true, nic se nestane, v opačném případě vznikne výjimka typu java.lang.AssertionError. Pokud použijeme druhou variantu, předá se výraz2 konstruktoru třídy AssertionError. (Tato třída má sedm konstruktorů s různými typy parametrů. Jeden je bez parametrů, dalších pět má jeden parametr některého z primitivních typů a poslední konstruktor má parametr typu Object; tedy výraz2 může být vlastně jakéhokoli typu.)

Třída výjimky je odvozena od třídy Error; to znamená, že ji nemusíme specifikovat v seznamu výjimek v deklaracích metod a nemusíme ji zachycovat a ošetřovat. Zachytí ji JVM, vypíše o tom zprávu a ukončí program.

Implementace

Jazyk Java neobsahuje preprocesor, takže nenabízí makra a ani možnost podmíněného překladu. Proto bylo nutno definovat assert jako nové klíčové slovo. To s sebou ovšem nese určité problémy - starší programy mohly používat assert jako identifikátor. Proto má překladač java verze 1.4 přepínač -source 1.4, který určuje, jak se má slovo assert chápat. Příkazem

```
java -source 1.4 Pokus.java
```

přeložíme soubor Pokus.java s asercemi.

O tom, zda se aserce uplatní za běhu programu, rozhodují přepínače při spouštění virtuálního stroje JVM. Použijeme-li přepínač -da (psavci mohou použít -disableassertions), aserce se při běhu neuplatní, přepínač -ea nebo -enableassertions způsobí, že se aserce uplatní. Tyto přepínače umožňují explicitně stanovit, pro které balíky - a dokonce pro které jednotlivé třídy - se mají aserce uplatňovat. Celý tvar přepínače -ea je buď

```
-ea:jméno_balíku...
```

(končí třemi tečkami), nebo

```
-ea:jméno_třídy
```

Přitom jméno třídy je kvalifikované jménem balíku. Přepínač -da se používá podobně. Chceme-li např. v balíku B1 povolit aserce ve třídě Pokus, která je pomocnou třídou v programu, jehož hlavní třídou je Hlavni, spustíme tento program příkazem

```
java -ea:B1.Pokus Hlavni
```

Příkazová řádka může obsahovat více přepínačů -ea a -da. Zpracovávají se před zavedením programových tříd do paměti a platí pro všechny zaváděče (class loader). Platí i pro systémové třídy, které nemají zaváděč. Poznamenejme, že povolíme-li přepínačem -ea aserce v určitém balíku, povolíme je tím i ve všech jeho "podbalících", pokud je tam explicitně nezakážeme pomocí přepínače -da. To umožňuje velmi přesně určit, ve kterých částech programu mají být aserce aktivní a ve kterých ne.

Vedle toho mají třídy zaváděčů v JDK 1.4 implicitní nastavení pro aserce, které je false (aserce nejsou aktivní), a toto nastavení lze měnit programově pro celý zaváděč, pro jednotlivé balíky a pro jednotlivé třídy pomocí nových metod

```
public void setDefaultAssertionStatus(boolean enabled);
public void setPackageAssertionStatus(String packageName, boolean enabled);
public void setClassAssertionStatus(String className, boolean enabled);
```

Pro návrat k implicitnímu stavu slouží metoda

```
public void clearAssertionStatus();
```

Odstraňování asercí

Zatímco v C/C++ jsou aserce založeny na podmíněné kompilaci, a proto po překladu s definovaným makrem NDEBUG prostě zmizí z programu, v Javě jsou vyjádřeny příkazem a budou součástí bajtového kódu přeloženého programu, ať je použijeme nebo ne. V rozsáhlejších programech mohou způsobit i zvětšení souborů .class, které v prostředí internetu nemusí být zanedbatelné.

Pokud bychom chtěli aserce z programu v Javě po odladění odstranit, aniž bychom museli podstatným způsobem zasahovat do zdrojového kódu, lze použít následující trik, který se v Javě používá i v jiných situacích místo podmíněné kompilace:

```
static final boolean asertuj=true;
// ...
if(asertuj) assert(x>y);
```

Pokud aserce potřebujeme, ponecháme statické konstantě asertuj hodnotu true. Po odladění

programu stačí hodnotu této konstanty změnit na false a program znovu přeložit. Protože podmínka v příkazu if není splněna a tato skutečnost je známa již v době překladu, může překladač odstranit nejen aserci, ale i příkaz if.

Používání asercí

V této části se podíváme, jak se aserce používají, k čemu se hodí a k čemu ne. Vše, co zde bude řečeno, berte jako doporučení; v žádném případě nejde o neporušitelná pravidla.

Vedlejší efekty

Při používání asercí bychom měli mít na paměti, že výraz, jehož platnost testujeme, se vyhodnotí jen v případě, že aserce budou aktivní. Jestliže je vyřadíme, výraz se nevyhodnotí. Znamená to tedy, že by neměl mít žádný vedlejší efekt - neměl by např. obsahovat přiřazení, operátory ++, -- ap.

Kontrola kontraktu

Veřejně přístupné metody tvoří, jak známo, rozhraní tříd. Podobně funkce exportované z modulu v klasickém modulárním programování tvoří rozhraní tohoto modulu. Každá z těchto metod nebo funkcí očekává parametry, které splňují jisté vstupní neboli předběžné podmínky (preconditions), a zavazuje se vrátit výsledek, který splňuje jisté výstupní podmínky (postconditions). O tom se někdy hovoří jako o principu kontraktu.

Některé programovací jazyky, např. Eiffel, umožňují specifikovat tyto podmínky deklarativně. O jejich kontrolu se pak stará překladač, který do programu vloží kód, jenž ověří, zda jsou splněny, a pokud ne, vyvolá výjimky. Java, C ani C++ tuto možnost nenabízejí, není ale problém naprogramovat si tuto kontrolu "ručně". Přitom se používají buď aserce, nebo výjimky. Mohlo by se zdát, že tyto dva mechanismy v Javě splývají, nicméně "běžné" výjimky v programu zachycujeme a ošetřujeme (a používáme je i v "ostré" verzi programu), kdežto aserce nezachycujeme; aserce slouží k určení místa chyby, kde není splněna nějaká zásadní podmínka, v průběhu ladění.

Poznámka: V tomto oddílu budeme pro stručnost termínem "funkce" označovat jak samostatnou funkci v C/C++, tak metodu objektového typu v C++ nebo v Javě. Bude-li potřeba, použijeme samozřejmě přesnější označení.

Vstupní podmínky

Pro testování vstupních podmínek ve veřejně přístupných metodách se aserce příliš nehodí. Podobně se nehodí pro testování vstupních podmínek ve funkcích exportovaných z modulů. Tyto podmínky zpravidla testujeme explicitně pomocí příkazů if, a pokud nejsou splněny, voláme chybové procedury, vyvoláváme výjimky (jiné než AssertError) nebo vracíme hodnotu, která indikuje chybu.

Podívejme se na příklad: Funkce, která počítá faktoriál celého čísla n (součin 1*2*3*...*n), očekává jako vstupní parametr nezáporné číslo typu int.

```
// Výpočet faktoriálu v C++
int f(int n) throw(range_error)
{
  // Kontrola vstupních podmínek
  if(n < 0) throw range_error("faktoriál: n < 0");
  int s = 1;
  while(n>1) s *= n--;
  return s;
}
```

Důvod, proč nepoužíváme aserce, je jednoduchý: použití těchto funkcí nemá jejich tvůrce často pod kontrolou. K porušení kontraktu - tedy k zadání nesprávných hodnot parametrů - může dojít z mnoha důvodů: protože uživatel programu zadal nesprávné hodnoty v nějakém řídicím prvku grafického uživatelského rozhraní programu, protože jsou poškozena data na disku, protože šum poškodil data přenášená po síti atd. (Všimněte si, že často nejde o chybu programu, ale o problém způsobený uživatelem programu nebo technickým zařízením. Nemusí to však být pravidlo.) Kvůli takovým problémům nesmí slušný program skončit a lakonicky oznámit, že na řádce té a té je něco špatně; měl by oznámit uživateli, že zadal špatná data, a umožnit mu zadat je znova, měl by se znovu pokusit o přečtení souboru z disku, měl by si vyžádat opakované zaslání dat po síti atd.

Navíc náhlé ukončení programu může mít za následek poškození dat a z toho plynoucí velké materiální škody. Tradiční mechanismus výjimek poskytuje možnost se těmto rizikům vyhnout.

Pro testování vstupních podmínek u funkcí, které nejsou veřejně přístupné, které tedy tvoří "implementační detaily" tříd nebo modulů, lze aserce s jistou opatrností použít. Tyto funkce totiž zpravidla voláme z těl veřejně přístupných funkcí, a proto lze očekávat, že hodnoty předávaných parametrů jsou již překontrolovány. Nesprávné hodnoty parametrů pak pravděpodobně signalizují chybu v programu, a proto

je použití aserce na místě.

Výstupní podmínky

Při kontrole výstupních podmínek je situace jiná. Funkce nebo metoda obdržela data, která prošla vstupní kontrolou, a tedy můžeme předpokládat, že jsou správná. Jestliže nejsou splněny výstupní podmínky, došlo k porušení kontraktu na straně funkce - je tedy naprogramována chybně a aserci je vhodné použít. (Příčinou může být nedostatečná prověrka vstupních parametrů, která nezachytila špatnou hodnotu parametrů, ale i to je chyba této funkce.)

Jako příklad si ukážeme opět funkci pro výpočet faktoriálu, tentokrát v Javě (JDK 1.4). Protože hodnotou faktoriálu je součin všech čísel od 1 do n, kde n je parametr (nebo 1, má-li parametr hodnotu 0), musí být výsledek kladný.

```
// Výpočet faktoriálu, tentokrát v Javě
// Nezbytné je JDK 1.4
public static long f(int n)
throws IllegalArgumentException
{
  if(n < 0) throw
new IllegalArgumentException(
"faktoriál - parametr je " + n);
  long s = 1;
  while(n>1)s *= n--;
  assert(s > 0); // Test výstupních podmínek
  return s;
}
```

Spustíme-li tuto funkci s parametrem 100, skončí chybou - proměnná s bude obsahovat 0, neboť při výpočtu došlo k přetečení (výsledek se nevejde do rozsahu typu long.) To znamená, že je něco špatně - zde je to test vstupních podmínek, který sice zachytí záporná čísla, ale neodhalí čísla, jejichž faktoriál je příliš velký a nevejde se do rozsahu typu výsledku.

Programové invarianty

Pod tímto označením se skrývají podmínky, které musí v programu vždy platit (přesněji: musí platit po dokončení určitých operací).

Představme si např., že náš program převádí peníze z jednoho účtu na druhý. Pokud převod proběhne správně, bude součet hotovostí na obou účtech před převodem stejný jako po převodu. Součet hotovostí je tedy programovým invariantem - veličinou, která se nemění a jejíž neměnnost je rozumné při ladění prověřit pomocí aserce.

Jiným příkladem může být výpočet délek stran trojúhelníka. Pro ně, jak známo, musí platit tzv. trojúhelníková nerovnost, která říká, že součet dvou stran je vždy větší než strana třetí.

Jednotlivé hodnoty stran a, b a c vrátí funkce Vypocti(). Správnost programu lze v místě, kde dostaneme všechny tři strany, ověřit asercí

```
double a = Vypocti();
double b = Vypocti();
double c = Vypocti();
assert((a+b>c) && (b+c>a) && (a+c>b));
```

Ať jsou strany trojúhelníka jakékoli, trojúhelníková nerovnost pro ně musí platit. Uvedená podmínka je tedy opět invariantem programu. (Zde je situace podobná jako u výstupních podmínek, test ale může být kdekoli, nejen na konci funkce.)

Situace, které "nemohou nastat"

Při psaní programu občas narazíme na situace, které - je-li program správný - nemohou nastat. Může jít například o příkaz switch, který neobsahuje alternativu default, protože k tomu - alespoň zdánlivě - není důvod. Při ladění je rozumné tyto "nemožné" alternativy ošetřit asercemi.

Podívejme se na příklad. Pokud jsme programovali správně, musí funkce f() vrátit celočíselnou hodnotu v rozmezí 1 až 4, a v závislosti na vrácené hodnotě voláme některou z dalších funkcí:

```
switch(f())
{// Přikaz switch bez default
case 1: Pripad1(); break;
case 2: Pripad2(); break;
case 3: Pripad3(); break;
case 4: Pripad4(); break;
}
```

Při ladění je ale rozumné přidat sem ještě alternativu default, která ukončí program, pokud se stane

něco neočekávaného a funkce f() vrátí hodnotu, která neleží v očekávaném rozmezí:

```
switch(f())
{// Přidáno default s asercí
case 1: Pripad1(); break;
case 2: Pripad2(); break;
case 3: Pripad3(); break;
case 4: Pripad4(); break;
default: assert((f() >= 1) && (f() <= 4)); break
}
```

Co dodat

Aserce jsou v současném C/C++ účinným nástrojem pro ladění programů a zdá se, že stejně platné budou i v Javě. Je ale nutno poznamenat, že v odladěném programu nemají co dělat. Jestliže uživateli oznámí zakoupený program Assertion failed nebo něco podobného a bez dalšího vyptávání skončí, nelze se divit, když uživatel vyhledá autora programu a bude chtít své peníze zpět. (Je ovšem otázka, zda to není pořád lepší, než nechat špatně navržený program běžet dál a dovolit mu tak páchat třeba i velké škody. Jak známo, Murphyho zákon je nekompromisní...)

Miroslav Virius

infotipy

[1] Virius, M.: Pasti a propasti jazyka C++. Grada Publishing 1997
[2] http://java.sun.com/j2se/1.4/docs/guide/lang/assert.html
[3] http://java.sun.com/j2se/