C/C++ (17) - Makefile

Dnes si ukážeme, jak se na Unixu obvykle překládá projekt z více zdrojových souborů. Řeč bude o Makefile.

21.4.2005 08:00 | Jan Němec | přečteno 17795×

Překlad s gcc

Z minulých dílů známe základy práce s gcc. Jednoduché projekty z několika zdrojových souborů lze obvykle přeložit prostým příkazem

```
gcc soubor1.c soubor2.c souborN.c -o program
```

V případě větších projektů je tento postup nevýhodný. Překlad může obsahovat sestavování jednotlivých logických celků, vytváření a linkování knihoven, navíc výsledkem může být hned několik programů, které mají část kódu společnou. Překlad C nebo C++ kódu je hlavně díky preprocesoru a optimalizacím časově náročný a zejména při vývoji by bylo neúnosné překládat po jediné změně libovolného zdrojového souboru celý projekt od začátku. Hodně času ušetří oddělení fáze preprocesingu a vlastního překladu od linkování, neboť pak můžeme využít mezivýsledky minulého překladu.

```
gcc soubor1.c -c
gcc soubor2.c -c
gcc souborN.c -c
gcc soubor1.o soubor2.o souborN.o -o program
```

Prvními třemi příkazy provedeme odděleně vlastní překlad jednotlivých zdrojových souborů, objektový kód se uloží do soubor1.o, soubor2.o a souborN.o. Posledním příkazem pak objektové soubory slinkujeme do výsledného programu. Pokud potom při ladění změníme například první *.c soubor, stačí zavolat

```
gcc soubor1.c -c
gcc soubor1.o soubor2.o souborN.o -o program
```

Při modifikaci hlavičkového souboru sice musíme překompilovat všechny *.c soubory, které jej přímo nebo zprostředkovaně inkludují, ale přesto nám oddělení vlastního překladu od linkování při vývoji skutečných praktických projektů ušetří spoustu času, který bychom jinak strávili čekáním na překladač. Bohužel vyznat se v časech modifikací a závislostech zdrojového kódu na hlavičkových souborech je nad síly běžného programátora. Proto vznikl make.

Makefile

Příkaz make provádí akce závislé na jiných akcích a může přitom mimo jiné testovat existenci a porovnávat datum poslední modifikace souboru. Seznam akcí a závislostí je popsán v souboru, který se obvykle jmenuje Makefile. Samotný make není přímo svázaný s překladem C, lze a má rozumný smysl jej použít v podstatě na libovolnou složitější sadu akcí s vysokým stupněm závislosti, kde neexistuje lepší řešení, klidně třeba vytvoření balíčku s dokumentací v pdf z texovských zdrojáků. Je ovšem pravda, že překlad C/C++ je určitě nejčastějším příkladem použití make a platí to i naopak, standardním způsobem organizace překladu na Unixu je Makefile.

V závěru minulého dílu jsme si ukázali jednoduchý projekt ze tří souborů. Jednoduchý Makefile by pro něj vypadal asi takhle:

```
priklad: funkce.o main.o
        gcc funkce.o main.o -o priklad

funkce.o: funkce.c
        gcc funkce.c -c

main.o: main.c funkce.h
        gcc main.c -c
```

Umístíme jej do adresáře se zdrojovým kódem a projekt přeložíme prostým příkazem

```
make
```

Pokud si to budete chtít vyzkoušet a budete kopírovat Makefile z webového prohlížeče přes schránku, dejte si pozor na odsazení příkazů. Odsazení jednotlivých příkazů **musí** být provedeno pomocí **tabelátoru**. To je určitě nejhloupější vlastnost make. Stáhněte si proto raději celý příklad zabalený jako c17beta.tar.gz.

Jak to celé funguje? V souboru Makefile je sada pravidel typu

```
cíl: závislost1 závislost2 ...
[tabelátor]akce
```

Cíl uvedený jako první je implicitní cíl příkazu make. Můžeme jej také určit explicitně jako první parametr, takže například

```
make main.o
```

provede pouze vlastní překlad souboru main.c podle třetího pravidla. Při volání make bez parametrů je cílem přeložený program - soubor priklad. Make nejprve projde závislosti, zde funkce.o a main.o. Nejdřív ověří funkce.o a zjistí, že soubor neexistuje, ale může jej vytvořit pomocí druhého pravidla. V pravidle pro funkce.o je závislost na funkce.c, ta je splněna, neboť soubor funkce.c existuje a není pro něj žádné pravidlo. Dojde tedy na akci pravidla pro funkce.o, make zavolá

```
gcc funkce.c -c
```

a vrátí se k testování závislostí prvního pravidla. Přečte si, že priklad závisí ještě na main.o, a tak jej pomocí třetího pravidla (zcela analogicky jako v případě funkce.o) vytvoří. Nyní již jsou splněny všechny závislosti a make se dostane k vykonání akce hlavního prvního pravidla.

```
gcc funkce.o main.o -o priklad
```

Trochu složitější to je, pokud jsme už celý projekt přeložili, ale dodatečně ještě změnili main.c (stačí zavolat *touch main.c*) a zavolali make. Při testu závislostí prvního pravidla sice oba objektové soubory existují a budou i o nějakou tu desetinu vteřiny starší než soubor priklad, ale v Makefile jsou i pro ně napsaná pravidla, a make tedy musí nejprve ověřit rekurzivně i jejich závislosti. V případě funkce.o je závislost splněna, neboť funkce.c je starší než funkce.o a pro funkce.c neexistuje žádné pravidlo. V případě main.o je tomu jinak, neboť main.c je mladší než main.o. Dojde tedy na akci

```
gcc main.c -c
```

a tím se vytvoří nový main.o s aktuálním časem modifikace. To znamená, že main.o bude mladší než priklad a make musí po návratu do zpracování prvního pravidla zavolat na závěr i jeho akci.

```
gcc funkce.o main.o -o priklad
```

Syntaxe Makefile umožňuje také vkládat komentáře a definovat proměnné.

```
# Tohle je naše první proměnná

OBJ=main.o funkce.o

# Použitím proměnné OBJ si ušetříme trochu psaní

priklad: ${OBJ}
        gcc ${OBJ} -o priklad
```

Jako typické použití proměnných bych uvedl nepovinné parametry překladače. Pomocí přepínačů -O0 až -O3 můžeme určit míru optimalizace gcc, pomocí -g zase přidáme do kódu ladící informace, které pak můžeme využít například nástroji typu gdb nebo valgrind. Nastavením jediné proměnné v Makefilu tak můžeme ovlivnit způsob překladu všech souborů, nejspíš se bude lišit překlad distribuční verze projektu od překladu během vývoje.

Často se v Makefile vyskytují vedlejší cíle. Už víme, že implicitně make zpracovává první cíl uvedený v Makefile, ale nic nám nebrání definovat další, obvyklé jsou například clean (smazání souborů vygenerovaných při překladu), install (instalace přeloženého projektu), uninstall a distrib (vytvoření distribuce zdrojového kódu projektu v jediném balíčku), běžně se lze také setkat s generováním dokumentace nebo přeložených balíčků pro systémy správy nainstalovaného software jako je například RPM.

V souvislosti s cíli, jimž neodpovídá žádný soubor se používá speciální cíl .PHONY. S jeho pomocí můžeme sdělit make, že cíl není vázaný na soubor stejného jména. Příkaz

```
make clean
```

pak bude fungovat i v případě, že v adresáři projektu shodou okolností existuje soubor jménem clean.

Příklad pro dnešní díl

Ukážeme si Makefile pro příklad z minulého dílu. Můžete si jej stáhnout zabalený jako c17.tar.gz.

```
#
# Makefile pro pokusný příklad 16. a 17. dílu seriálu o C/C++ na linuxsoft.cz
#

# Jméno přeloženého programu
program=priklad

# Seznam objektových souborů použijeme na dvou místech.
OBJ=funkce.o main.o

# Míra optimalizace překladače gcc
OPT=-O2

# Cílům build, install, uninstall, clean a distrib neodpovídá přímo žádný soubor

.PHONY: build
.PHONY: install
.PHONY: uninstall
.PHONY: clean
.PHONY: distrib

# První cíl je implicitní, není třeba volat 'make build', stačí 'make'.
# Cíl build nemá žádnou akci, jen závislost.
```

```
build: ${program}

# install závisí na přeložení projektu, volat ho může jen root
install: build
        cp ${program} /usr/bin

# uninstall má jenom akci a žádnou závislost, volat ho může jen root
uninstall:
        rm -f /usr/bin/${program}

# clean smaže soubory po překladu
clean:
        rm -f *.o ${program}

# distrib vytvoří balíček s kompletním zdrojovým kódem

# akce na dva řádky se napíše pomocí zpětného lomítka

distrib:
        tar -c funkce.c main.c funkce.h Makefile > c17.tar; \
        gzip c17.tar

${program}: ${OBJ}
        gcc ${OBJ} -o ${program} ${OPT}

funkce.o: funkce.c
        gcc funkce.c -c ${OPT}

main.o: main.c funkce.h
        gcc main.c -c ${OPT}
```

Pokračování příště

V příštím dílu si řekneme, co jsou to implicitní pravidla a jak se dá napsat přijatelný Makefile pro vetší projekt i bez nástrojů typu Automake, aniž bychom zešíleli z uvádění závislostí a pravidel pro překlad jednotlivých souborů.

Online verze článku: http://www.linuxsoft.cz/article.php?id_article=722