Metaprogramovanie — definícia, použitie, typy metaprogramovania

Technika pri ktorej programy manipuluju s inymi programami (aj so sebou samymi) ako s datami:

- citanie
- analyzovanie
- generovanie
- transformovanie

Metaprogramovanie sa pouziva pre generovanie kodu pocas kompilacie, sebaupravu programov, zlepsenie designu (AOP). Typy metaprogramovania:

- reflexia (skumanie a modifikovanie seba sameho za behu)
- makra (nahrada kodu)
- generovanie kodu pocas kompilacie
- AOP

Reflexia — definícia, prostriedky realizácie reflexie, použitie v jazyku Java

Schopnost programu za behu skumat, pozorovat a modifikovat svoju strukturu a spravanie.
V Jave sa reflexia pouziva na:

- prehliadanie tried, rozhrani alebo metod
- vytvaranie instancii
- volanie metod
- zmenu pristupnosti metod a premennych

Metody:

- getClass(), getInterfaces(), getDeclaredMethods(), getDeclaredFields(),getConstructors(),getMethods()
- newInstance(),
- invoke()
- setAccessible()

- metatriedy a analyzatory tried ?

Metatriedy:
Metatrieda je trieda, ktorej inštancie sú triedy. Rovnako ako bežná trieda definuje správanie určitých objektov, metatrieda definuje správanie určitých tried a ich inštancií.

Reflexia je API, ktoré sa používa na skúmanie alebo modifikáciu správania metód, tried, rozhraní pri behu. Požadované triedy pre reflexiu sú poskytované v balíku **java.lang.reflect.** Reflexia nám poskytuje informácie o triede, do ktorej objekt patrí, ako aj o metódach tejto triedy, ktoré možno vykonať pomocou objektu. Prostredníctvom odrazu môžeme vyvolať metódy za behu bez ohľadu na špecifikátor prístupu, ktorý sa s nimi používa.

1. We can invoke an method through reflection if we know its name and parameter types. We use below two methods for this purpose
   getDeclaredMethod() : To create an object of method to be invoked. The syntax for this method is

   ```
   Class.getDeclaredMethod(name, parametertype)
   name- the name of method whose object is to be created
   parametertype - parameter is an array of Class objects
   ```

   invoke() : To invoke a method of the class at runtime we use following method–

   ```
   Method.invoke(Object, parameter)
   If the method of the class doesn't accepts any
   parameter then null is passed as argument.
   ```

2. Through reflection we can **access the private variables and methods** of a class with the help of its class object and invoke the method by using the object as discussed above. We use below two methods for this purpose.
   Class.getDeclaredField(FieldName) : Used to get the private field. Returns an object of type Field for specified field name.
   Field.setAccessible(true) : Allows to access the field irrespective of the access modifier used with the field.

Zásobník volaní a jeho introspekcia

Zasobnik volani (**call stack**) je LIFO zasobnik, ktory obsahuje aktualne vykonavane metody a podmetody = vytvára strom volaní. Pri vnoreni sa do metody je metoda pridana do zasobnika, po vykonaní je odstranena. Kazda invokacia metody je reprezentovana **stack frame**-om. Na call stack sa mozeme pozerat priamo pri debugovani (IDE / API - stackwalker). Per thread – nový pre každé vlákno.

Stack trace je report call stacku z isteho bodu vykonavania programu – typicky zobrazeny pri zlyhani programu. Daju sa z neho citat metody volane pred zlyhanim programu aj konkretny bod (riadok kodu) v programe, kde nastala chyba. Kazdy frame v nom je reprezentovany objektom **StackTraceElement**. Frame navrchu reprezentuje bod vykonavania, kedy bol stack trace vygenerovany. Typicky ide o bod, kedy bola vyhodena vynimka spojena so stack traceom.

- Argumenty, lokálne premenné, návratová adresa
- Introspekcia(sebapozorovanie):
    - StackTraceElement[] getStackTrace()
        - file name
        - line number
        - class name
        - method name
        - is native method
    - void printStackTrace() – do error output
- Využitie: Logger – doplnenie aktuálnej polohy do výpisu logu

Realizácia tried v jazyku Java a ich načítavanie (ClassLoader)

java.lang.ClassLoader je sucast JRE, ktora dynamicky nacitava triedy do JVM. ClassLoader je zodpovedny za najdenie kniznic, precitanie ich obsahu a nacitanie tried v nich. Trieda je nacitavana bud ked sa vykona new klucove slovo (… = new Class()), alebo ked sa narazi na staticky odkaz na triedu (System.out).

Ked JVM potrebuje triedu, zavola loadClass() metodu ClassLoadera. Metoda potom vola findLoadedClass() metodu, ktora skontroluje ci trieda uz bola nacitana alebo nie. Ak nebola, metoda deleguje ulohu rodicovskemu ClassLoaderovi, ktory proces zopakuje. Ak nebola nacitana v ziadnom ClassLoaderi, vrchny ClassLoader zavola metodu findClass(), ktora sa pokusi triedu najst. Ak sa to nepodari, vyhodi sa ClassNotFoundException, ktoru odchyti nizsi ClassLoader a on sa pokusi triedu najst.

```
protected synchronized Class<?> loadClass (String name, boolean resolve)
    throws ClassNotFoundException{

    // First check if the class is already loaded
    Class c = findLoadedClass(name);
    if (c == null) {
        try {
            if (parent != null) {
                c = parent.loadClass(name, false);
            } else {
                c = findBootstrapClass0(name);
            }
        } catch (ClassNotFoundException e) {
            // If still not found, then invoke
            // findClass to find the class.
            c = findClass(name);
        }
    }
    if (resolve) {
            resolveClass(c);
    }
    return c;
}
```

Tri ClassLoadre pouzivane JVM:

- Bootstrap class loader – core java classes (java.lang napriklad) – rodic vsetkych classloaderov
- Extensions class loader – extension directory classes
- System class loader – classes from CLASSPATH

Trieda je identifikovana ClassLoaderom a plnym menom.

Dynamické proxy v jazyku Java

- Proxy je trieda fungujúca ako rozhranie/fasáda pred Objektom
- Dynamické proxy je proxy vytvárané počas behu programu, DP je prikladom proxy pattern – kontrola pristupu k objetku – modifikacia spravania
- Použitie:
    - Proxy.newProxyInstance(classloader, class, instance) – vytvorí proxy s metódami class
    - Rozhranie InvocationHandler je používané na implementovanie proxy triedy (instance parameter)
        - Object invoke(Object proxy, Method m, Object[] args)
        - Dnu zvyčajne po vykonaní požadovanej proxy funkcie voláme m.invoke(obj)
- Využitie:
    - Logging / Mockovanie objektov

Statické – špecifické správanie pre triedu – každé proxy vlastná trieda
Dynamické – všeobecné generické správanie – 1 trieda pre všetky proxy – správanie v IH

Dynamicke vs staticke proxy:
Ak pouzijeme staticke proxy, pre kazdu triedu musime predom napisat proxy triedu. To znamena ze ak mame 1000 tried, potrebujeme aj 1000 proxy tried. Tomu sa nevyhneme ak sa chceme ku kazdej triede

spravat inac. Ak vsak chceme vykonat len nejake genericke spravanie, mozeme pouzit dynamicke proxy. V dynamickom proxy je narozdiel od proxy trieda vytvorena pri runtime a pouzivame InvocationHandler pre definovanie jej spravania. IH je trieda, ktora riesi volanie metod definovanych v rozhrani.

Anotácie — význam a typy metadát v programovom kóde, ich použitie

Metadata su data poskytujuce informacie o inych datach. Anotacie su strukturovane metadata o zdrojovom kode.

Klasifikacia metadat:
- standardne / pouzivatelske
- strukturovane / nestrukturovane
- zabudovane / externe

Metadata pouzivame pre typove definicie, doplnujuce informacie, vytvorenie pomocnych nastrojov, generovanie kodu, runtime konfiguraciu, zaznamenavanie navrhovych rozhodnuti.

Prikladmi metadat su:

-	typy	standardne / pouzivatelske	strukturovane	zabudovane
-	komentare	pouzivatelske	nestrukturovane	zabudovane
-	XML	pouzivatelske	strukturovane	externe
-	anotacie	pouzivatelske	strukturovane	zabudovane

- Využitie:
  - Definícia typov
  - Doplňujúce informácie
  - Umožnenie vytvorenia pomocných nástrojov

Anotácie v jazyku Java — definícia a použitie anotačných typov pomocou reflexive

Anotacie vytvarame pouzitim @interface. Anotacie mozu mat vlastne elementy – vyzeraju ako metody bez implementacie. Anotacie nemozu byt rozsirene. Pomocou anotacii mozme oznacit triedy, rozhrania, premenne, alebo parametre.

Zabudovane anotacie v jave:
@Deprecated, @Override, @SupressWarnings

```
// DEFINICIA
@interface MyAnnotation {
    String   value();
    String   name();
    int      age();
    String[] newNames();
}

// POUZITIE
@MyAnnotation(
    value="123",
```

```
    name="Jakob",
    age=37,
    newNames={"Jenkov", "Peterson"}
)
public class MyClass { }
```

- Typy využitia:
  - Generovanie kódu
  - Runtime konfigurácia
  - Deklaratívna špecifikácia vlastností kódu
  - Zaznamenanie návrhových rozhodnutí
- Príklady využitia:
  - Mapovanie: Hibernate, Jackson, JAXB
  - Testovanie: Junit
  - REST WS: Springboot
  - Generovanie SOAP klienta: Spring MVC

Spracovanie anotácií počas prekladu — anotačný processor

- Anotačný processor je plugin pre kompilátor javac (vlastné JVM)
- Slúži na skenovanie a spracovanie anotácií (napr generovanie kódu) počas kompulácie
- Kompilácia prebieha v kolách, kde sa postupne volajú anotačné procesory
- Procesor implementujeme použitím rozhrania javax.annotation.processing.Processor (ext. AbstractPr)
  - init(ProcessingEnvironment processingEnv) – prázdny konštruktor, preto inicializácia tu
  - process(Set<? Ext TypeElement> annots, RountEnvironment env) – „main()“ metóda
  - getSupportedAnnotationTypes (alebo anotácia)
  - getSupportedSourceVersion (al anotácia) – špecifikovanie java ver. (SourceVersion.latestSupported())
- Registrácia procesora: META-INF/services -> javax.annotation.processing.Processor
- RoundEnvironment: getElementsAnnotatedWith(Class<? ex Ann>)
- Javax.lang.model API – elementy, typy ...

Princíp inverzie závislosti v architektúre softvéru a vzor Dependency Injection

Princip inverzie zavislosti je specificky sposob udrzavania nezavislosti modulov. V tomto principe ide o taky navrh softveru, kde su high-level moduly nezavisle od detailov implementacie low-level modulov.
Princip:

- high-level moduly maju byt nezavisle od low-level modulov
- obidva maju byt zavisle na abstrakciach
- abstrakcie by nemali byt zavisle na detailoch ale detaily na abstrakciach.

V prvom rade by sa malo mysliet na abstraktnu interakciu medzi modulami. Moduly by sa mali navrhnut s ohladom na tuto interakciu. V podstate to znamena, ze objektu predame jeho instance variables.

Dependency injection je designovy navrh, pri ktorom nenechavame objekty aby si vytvarali svoje dependencie (instance fields), ale predavame im ich externe. Znizuje zavislost tried. Zvysuje znovupouzitelnost tried a udrzatelnost. Ulahcuje testovanie.

Druhy dependency injection:

- constructor injection
- setter injection
- interface-based injection
- service locator injection

Generovanie kódu — typy generátorov a techniky generovania kódu

Generatory delime na:

- **pasivne** – jednorazove, kod je dalej rucne upravovany (generovanie kostry noveho modulu)
- **aktivne** – opakovane, kod nema byt upravovany rucne, pri potrebe zmeny sa vygeneruje znovu, vstupom je konfiguracia / domenovo specificky jazyk -> kód(HTML, SQL, Java)

Podla semantickeho modelu delime generovanie na:

- **bez modelu**
- **s modelom**

Podla sposobu generovania delime generovanie na:

- **pomocou sablony**
- **transformacne generovanie (riadený vstupom/ vystupom)**

Generovanie bez pouzitia modelu:
Generovany kod obsahuje logiku programu zakodovanu priamo v riadiacich strukturach. Neobsahuje semanticky model.

Generovanie s modelom:
Obsahuje semanticky model -> generuje sa len kód na naplnenie . Je zachovane rozdelenie vseobecny / specificky kod.

Transformacne generovanie:
Vytvorenie programu, ktory bude na zaklade semantickeho modelu generovat vystupny kod. Dva druhy:

- riadene vystupom – vychadza sa z pozadovaneho vystupneho kodu, z modelu sa ziskavaju udaje potrebne na jeho generovanie
- riadene vstupom - prechadzaju sa vstupne datove struktury a na ich zaklade sa generuje vystup

Generovanie pomocou sablony:
Pouziva sa pri vyvoji webovych aplikacii na generovanie HTML kodu. Vhodne pouzivat, ked je velka cast vystupneho kodu staticka. Pri tomto generovani sa pouzivaju tri hlavne komponenty:

- sablona - vystupny kod, v ktorom su premenlive casti nahradene znackami
- kontext – zdroj, z ktoreho sa cerpaju data pre vyplnenie sablony
- sablonovy system – vyplna sablonu na zaklade kontextu

Generovany kod nie je mozne priamo upravovat, lebo pri opatovnom generovani budu upravy prepisane. Pri OOP jazykoch je mozne oddelit generovany a manualne pisany kod pomocou dedicnosti. Tato technika sa vola generation gap. Jednotlive sposoby mozu byt aj kombinovane.

Typickymi problemami generovania su specialne znaky – escaping a konflikt klucovych slov a konfilktov mien.

Vývoj softvéru na základe modelu (MDSD), význam modelu pri generovaní

MDSD (Model Driven Software Development) je technika generovania kodu na zaklade vysokourovnoveho modelu. Ten pridava vyssiu uroven abstrakcie ako programovaci jazyk. Povazuje sa za alternativu ku klasickemu programovaniu. Najprv sa navrhne model -> modelovaci nastroj- vygeneruje kod v pozadovanom prog. jazyku.

MDA (Model Driven Architecture) je standardny pristup k pouzivaniu modelov pri vyvoji softveru. Predpisuje urcite druhy modelov, ako maju byt pripravene a vztahy medzi roznymi druhmi modelov. Ako modelovaci jazyk sa pouziva UML. Zakladnou myslienkou je pri vyvoji separovat operacie systemu od detailov implementacie na konkretnej platforme. Vdaka MDA su systemy specifikovane nezavisle od platformy.
Efektivita, ľahšia komunikácia, platformova nezávislosť.

Makrá a ich použitie v programovacích jazykoch

- Makra su programovaci pattern, ktory premiena dany input na preddefinovany output. Mozu nahradzovat sekvenciu stlaceni klaves, pohybov mysi, prikazov a podobne.
- Pri programovani su makra nastrojom, ktory umoznuje developerovi jednoducho znovupouzit kod. Ide o znacky, ktore su neskor nahradene skutocnym kodom. Funguju ako search and replace funkcia. Substitucia makra = makro expanzia -> vykonavana kompilatorom pred kompilaciou kodu.
    - vloženie funkčnej časti kódu na zadané miesta
        - oproti funkcii lepší výkon vďaka absencii skoku, ale horšia čitateľnosť
        - existuje riziko že premenná v makre už bude existovať v kontexte kde má byť vložená => nutné použiť dlhé nezmyselné premenné pre minimalizovanie rizika
        - ďalším rizikom je narušenie štruktúry program -> nutné ohraničiť makro napr. zátvorkami {}

JAVA NEMÁ

    - syntaktické makrá
        - pracuje na úrovni syntaktického stromu, nie textu
        - Makro = funkcia -> definovanie vlastných vzorov a pravidiel
        - napr. implementovanie konečno-stavového automatu pre pattern-maching, ktorý bude možné vytvoriť len na základe definovania pravidiel, ktoré následne makro preloží na spustiteľný kód v jazyku Lisp/Scheme (článok Educational Pearl: Automata via Macros)
- Syntakticke makra pracuju na urovni syntaktickeho stromu a nie textu. Makro je vtedy funkcia produkujuca fragment syntaktickeho stromu.
-
- (define (fib n)
    - (cond ((= n 0) 0)
        - ((= n 1) 1)
            - (else (+ (fib (- n 1))
                - (fib (- n 2))))))

Systémy typov a ich využitie

- wiki: system typov je sada pravidiel, ktorá priradzuje konštrukciám v programe vlastnosť, tzv. "typ", aby sa zabránilo chybám v dôsledku nesúladu definovaných rozhraní a reálneho použitia, teda toho čo vie kompilátor a čo je reálne v run-time
- patri sem:
    - polymorfizmus (schopnosť mať viac typov) – nadtyp – podtyp
    - generické typy
        - typové premenné
        - kovariancia – dovolené použitie generického **podtypu** namiesto požadovaný generického typu (List<? extends Number>) -> bezpečné čítanie (do number dám integer/float ...)
        - kontravariancia - dovolené použitie generického **nadtypu** namiesto požadovaný generického typu (List<? super Integer>) -> bezpečný zápis (number dám do Objectu)
- kontrola typov
    - statická
        - odhalenie chýb pri kompilácii, zlepšenie výkonu
        - vynútenie architektúry
    - dynamická
        - typ ako súčasť hodnoty, možnosť doplniť voliteľnú kontrolu typov
    - kombinovaná
- jazyky z pohľadu typov
    - bezpečné
    - nebezpečné
- nevýhodou je komplikovanejší jazyk

Aspektovo orientované programovanie — význam, základné vlastnosti, realizácia v AspectJ

- prepletanie(tangling) – rôzne moduly majú spoločnú funkcionalitu
- rozptýlenie(scattering) – duplicitný kód - výsledkom prepletania
- pretínajúce záujmy(crosscutting concerns)
    - pôvodný problem kvôli ktorému vzniklo prepletanie a rozptýlenie
    - Riešením je AOP, resp vytvorenie aspektu
    - Napríklad logovanie
- AspectJ – jazyk pre Impl. AOP v Jave
- Bod spájania (Joinpoint) – bod záujmu v procese vykonávania projektu
- Bodový prierez (Pointcut) – výraz pre výber joinpoints
- Rada(advice) – funkcionalita vkladaná pred/za/okolo joinpoints
- Použitie: logovanie, tracing, pooling, security, transaction, persistence
- Pretkávanie (Weaving)
    - process vkladania volania aspektov do kódu
    - Ajc – AspectJ Compiler
    - Statické
        - Pretkávanie zdrojových kódov kompilátorom

        - Pretkávanie binárnych súborov – preklad tried a aspektov
    - Dynamické
        - Pretkávanie pri načítaní (load-time weaving) - špeciálny ClassLoader
        - Pretkávanie počas behu – Spring AOP, dynamické proxy

AspectJ — body spájania a súvisiace bodové prierezy (kinded pointcuts)

- Kinded pointcuts – vyberanie joinpoints na základe joinpointu
- Deklarácia:
    - pointcut <pointcutName>(<optional args>): <Joinpoint action>(<signature>)
    - <advice>(<optional args>) : <Joinpoint action>(<signature>)
- Pointcuts pre joinpoints:
    - Vykonávanie metódy – execution(public String findById(*))
    - Volanie metódy - call(public String findById(*))
    - Vykonanie konštruktora - execution(Car.new(..))
    - Zavolanie konštruktora - call(Car.new(..))
    - Prístup k premennej – get(* Application.author), Field.get(Object, Application.author))
    - Zápis premennej - set(* Application.author), Field.set(Object, Application.author))
    - Vyhodenie výnimky – handler(SqlException+)
    - Inializácia triedy – staticinitialization(MyClass)
    - Inicializácia objektu pred/po vykonaní rodičovského konštruktora - initialization(my.package.*)/preinitialization(my.package.*)
    - Vykonanie odporúčania – adviceexecution()
    - Podľa typu objektov: this(Account+), target(com.pack.*),args(*)
- Zachytenie kontextu:
    - Pomocou this(product) -> zachytíme object na ktorý sme sa pripojili do premennej product
    - Ekvivalentne pre target, args
- Signatúry
    - * - 0- n písmen
    - .. – ľubovoľný počet args, v package berie všetko v ňom okrem sub-pack
    - Account – matches my.package.Account, another.pack.Account
    - java.*.date – matches java.sql.Date and java.util.Date
    - java..* - matches java.sql.Date, javax.swing.FileSystemView
    - Account+ - matches Account, MyPersonalAccount
    - @Bussiness* Customer* - matches @Bussiness Customer, @BusinessClass Revenue
    - Collection || Map – matches Collection<Integer> ids, Map hashes
    - Public void Account Account.set*(*) – matches setters
    - * *(..) throws SQLException
    - Account+.new(..)

AspectJ — non-kinded pointcuts (scoping pointcuts, contextual pointcuts)

Prierezy ktore vyberaju joinpointy na zaklade inych kriterii ako je signature
Body spajania:

- zalozene na control flow - cflow, cflowbelow
- zalozene na zdrojom kode - within (typ), withincode (kód – metóda/konštruktor)
- zalozete na cielovom objekte - this, target, args

AspectJ — odporučenia

Odporucenia su zdrojovy kod, ktory sa ma vykonat pri pointcute. Pozname tri druhy odporuceni podla toho kde pri pointcute sa maju vykonat:

- before
- after (/ returning / throwing)
- around:
  - obklopuje bod spajania
  - ma navratovu hodnotu
  - vykonanie obklopeneho kodu pomocou proceed()

- Deklarácia
  - <advice>(<optional arguments>) : <pointcut>(<optional arguments>)
- Typy
  - Around(..) – má návratovú hodnotu, zvičajne voláme vnútri proceed()
  - before(..)
  - After(..)
  - After(..) returning – napríklad After(var) returning var
  - After(..) throwing – napríklad After throwing, after(ex) throwing ex
- Vnútri advice je možné využiť reflexiu cez premenné
  - thisJoinPoint
  - thisJoinPointStaticPart
  - thisEnclosingJoinPointStaticPart

// DEDENIE VAROVANIE CHYBY NAJST

AspectJ — statické pretínanie (medzitypové deklarácie, dedenie, varovania a chyby, atď.)

Doplnenie struktury triedy

- tzv. „Medzitypové deklarácie“ = Intertype Declarations:
  - Definovanie členskej premennej alebo metódy pre ľubovoľný typ (aj konštruktor ?)
  - Public String Customer.name;
- Bez nutnosti definovať pointcut, namiesto toho jednoducho vnútri aspektu použijeme
  - public String Nameable.getName() {…}
- Private (len aspekt), public, protected (default, pre package aspektu)