הרצאה 13- Future and Completable Future Demo

יום ראשון 30 מאי 2021 09:08

נערך וסוכם ע"י שניר נהרי - מבוסס על סיכומיה של לינוי דוארי

Future Demo 13.1

כבר הזכרנו בעבר שאנו רוצים לדעת כיצד המנגנונים שלמדנו עובדים מאחורי הקלעים. לכן בהרצאה זו נעבור בהדרגה דרך הדגמה מ-Future ל-Completable Future. מומלץ לבצע את השלבים של ההרצאה במקביל על מנת לקבל למידה מיטבית. ראשית נפתח בסביבת הפיתוח פרויקט חדש. נוסיף source file חדש בתוך package שניצור. נרצה לכתוב דמו למשימה שרצה ברקע. המשימה תשהה בתור של thread-pool , תצא ממנו, תבצע חישוב בפרק זמן מסוים ותחזיר תוצאה. זהו הדמו של thread pool שנכתוב. נייצר מחלקה בשם MyThreadPoolDemo.java. נגדיר לה מתודה execute(Runnable r) פומבית. אם זה היה thread-pool אמיתי, היינו מכניסים את r לתור ובדומה ל-active object, יהיה thread שרץ ברקע ושולף runnables אחד אחד מתוך התור. מכיוון שזוהי רק הדגמה נריץ את ה-runnable מתוך thread שרץ ברקע . עד כה זהו הקוד שכתבנו -

```
public class MyThreadPoolDemo {
    public void execute(Runnable r) {
        new Thread(r).start();
    }
```

אך מה קורה אם ה-runnable הזה הוא למעשה callable? כלומר מה יקרה אם נרצה להחזיר ערך החזרה כל שהוא ? לשם כך נכתוב מתודה פומבית דומה שתלויה בטיפוס הפרמטרי של V בשם submit(Callable<V> c). המתודה submit מקבלת callable ומכניסה אותו לתור .בהמשך ה- callablle יצא מהתור, ותיקרא המתודה c.call ולבסוף נקבל את ערך החזרה.

אם היינו מגדירים את המתודה כך – היינו מקבלים מצב שבו ה- thread שמבצע את הקריאה ל- call צריך להמתין עד שהמתודה תסיים. זהו בזבוז משאבים. במקום זאת, נרצה ליצור thread שיבצע זאת . נשים לב שבממשק של Callable צריך לתפוס חריגות. לכן על מנת להימנע מתפיסת החריגות, נכתוב ממשק דמו של Callable. הממשק Callable יהיה תלוי בטיפוס הפרמטרי V ויכיל מתודה call, אך זו לא תזרוק אקספשן בניגוד לממשק Callable האמיתי . כך יראה הממשק -

```
public interface Callable<V> {
    V call();
}
```

כעת במקום להמתין לערך החזרה של המתודה call נשנה את הקוד באופן הבא - כשה-callable שהכנסנו לתור, ייצא ממנו, הוא יקרא ל-c.call(). את ערך החזרה של המתודה call נקלוט אל אובייקט מסוג Future<V>. במקום להשתמש ב-Future האמיתי, נכתוב מחלקת דמות Future . מחלקה זו תהיה תלויה בפרמטר הגנרי V . בנוסף, נגדיר מתודות get, set . כך יראה הקוד -

```
public class Future<V> {
    V value;

    public void setValue(V value) {
        this.value = value;
    }

    public V getValue() {
        return value;
    }
}
```

כעת בפונקציה submit (שבתוך MyThreadPoolDemo) נחזיר את האובייקט Future f. מכיוון שמיד החזרנו את f, הערך שבתוכו עדיין null.

מה נעשה בתוך ה-runnable שמקבלת המתודה execute ? מיד לאחר קבלת ערך החזרה של המתודה call ניתן לבצע set לאובייקט Future עם הערך שהתקבל . כך זה יראה -

```
public <V> Future<V> submit(Callable<V> c) {
    Future<V> f= new Future<>();
    execute(()->f.setValue(c.call()));
    return f;
}
```

ב-main נייצר מופע של MyThreadPoolDemo ונריץ משימה כל שהיא . נכתוב callable בביטוי למבדה שיבצע sleep ל-4 שניות ויחזיר את המחרוזת "42". נקיף את הקוד ב-try, catch כדי שנוכל לתפוס InterruptedException. המתודה submit() לא תריץ אותו עכשיו, אלא היא לוקחת את ה-callable, שמה אותו בתור וחוזרת מיד. המתודה submit() תחזיר אובייקט Future<String>. נתבונן בקוד -

```
public static void main(String[] args) {
    MyThreadPoolDemo tp = new MyThreadPoolDemo();
     Future<String> f = tp.submit(()->{
        try {Thread.sleep( millis: 4000);} catch (InterruptedException e) {}
        return "42";
    });
     System.out.println(f.getValue());
}
```

אם נדפיס כרגע את f.get,יודפס הערך null, כי כרגע המתודה submit סיימה, ורק לאחר 4 שניות, יוזן ל-f הערך "42".

Future Demo 13.2

בחלק זה נממש את ההמתנה שמחכה לערך החזרה במתודה get במידה והערך טרם חזר. נשנה מספר דברים במחלקה Future . ראשית במתודה get נוסיף בדיקה. אם ה-value == null, נמתין בעזרת wait (לכל אובייקט בג'אווה יש מימוש ל- wait). כל קריאה ל-()wait צריכה להיות מסונכרנת עם האובייקט, ולכן כל המתודה הזו צריכה להיות מוגדרת כ - synchronized. כמובן שצריך לתפוס InterruptedException במידה ויעירו אותנו מההמתנה.

אם יש thread של ה-main שקרא ל-get ותקוע ב-wait, הרי שכאשר ה-thread pool יקרא ל-c.call, הערך שיחזור יוכנס לתוך ה-Future באמצעות set. כלומר לאחר שביצענו set ל- Future עלינו לשחרר כל thread שהמתין לערך חזרה זה, דוגמת ה- thread של ה-main. לכן נשנה גם את ה-()set בהתאם. מכיוון שיכול להיות שה-future הזה מוחזק ע"י מספר threads , כך שכולם ממתינים לקבלת ערך החזרה נשתמש בקריאה ()notifyAll לשחרור כל ה-threads המתאימים. בשל קריאה זו גם את המתודה ל-()set נסמן כ-synchronized - כך תראה המתודה Set -

```
public synchronized void setValue(V value) {
    this.value = value;
    notifyAll();
}
```

לא מפריע לנו שה-(...)set מוגדר כ-synchronized כי הוא נקרא רק פעם אחת. לעומת זאת, המתודה f.get יכולה להיקרא מספר רב של פעמים, ולכן נרצה להימנע מלשלם את המחיר הכבד של מנגנון ה-synchronized . נרצה להשתמש בו אך ורק כאשר value == null. נבצע זאת בעזרת טכניקת ה- double checking שראינו בהרצאות הקודמות. במקום שמתודת ה-get כולה תסומן כ- synchronized, נבדוק אם value == null . אם אכן value == null נקיף את קטע הקוד הבא ב-synchronized ונבדוק שוב האם נשים לב value == null. כך תראה המתודה - get

```
public V getValue() throws InterruptedException {
    //Double Checking
    if (value == null)
        synchronized (this) {
            try {
                if (value == null)
                    wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    return value;
}
```

כעת נריץ את ה-main . המתודה תקרא ל- f.get ולאחר 4 שניות יודפס הערך 42.

Completable Future Demo 13.3

החיסרון העיקרי שראינו ב-Future הוא שהמתודה get היא blocking call. ה- thread שקרא למתודה הוא משאב מבוזבז שתקוע בהמתנה לערך החזרה. נרצה ליצור מנגנון שיגדיר את הפעולות המבוקשות מראש כך שאלו תבוצענה כשהערך יוחזר, כלומר לאחר התהליך החישובי הכבד הפעולות יתבצעו. בצורה זו ה- thread יוכל להמשיך בפעולותיו ולא להיתקע. נכתוב מחלקה חדשה בשם CFuture שהיא הדמו של המחלקה CompletableFuture (זוהי רק הדגמה והיא איננה מורכבת כמו האובייקט האמיתי, להמחשה בלבד). נגדיר את המתודה supplyAsync(Callable c). כרגע נגדיר שהיא מחזירה void, בהמשך נשנה זאת. נרצה בתוך המתודה לקרוא ל-c.call . באובייקט האמיתי, CompletableFuture מזריק את ה-Callable לתוך ה-fork join pool.

לשם ההמחשה, נקרא ל-c.call ב- thread שרץ ברקע, כלומר המתודה חוזרת מיד . לכן נצטרך למצוא דרך להחזיר את V במקום void. נוכל להחזיר Future<V> כמו מקודם במחיר ש-get היא blocking call. אך כעת, במקום זאת, נחזיר CFuture<V>. באופן זה, קראנו למשימה שתרוץ ב- thread ברקע. החזרנו מיד את cf שעתיד להחזיק את ערך החזרה מהמשימה. נרצה להזין את ערל החזרה לתוך cf מתאים ע"י set, בדיוק כמו במחלקה Future . לכן נכתוב את המתודה set במחלקה CFuture<V>. נכתוב 2 מתודות עיקריות נוספות שבעזרתן נוכל להגדיר מה ה- thread צריך לבצע לאחר קבלת ערך ההחזרה. נרצה ליצור את המתודה supplyAsync(Callable c). עד כאן המחלקה CFuture נראית כך -

```
public static <V> CFuture<V> supplyAsync(Callable<V> c) {
    CFuture<V> cf = new CFuture<>();
    new Thread(() -> cf.set(c.call())).start();
    return cf;
}
```

כעת ב- main נעתיק את הפונקציה שהזרקנו ל- Future -

```
CFuture<String> cs = CFuture.supplyAsync(() -> {
    try {
        Thread.sleep( millis: 4000);
    } catch (InterruptedException e) {
    }
    return "42";
});
```

על האובייקט שיוחזר (cs) , נרצה לשרשר כמה פעולות אחת אחרי השנייה בעזרת המתודה ()thenApply. המתודה זו תפעל כך שבהינתן פונקציה, היא תחזיר פונקציה מטיפוס אחר לגמרי. הערך יוחזר לא כעת, אלא לאחר 4 שניות . כשהערך יוחזר, הוא ייכנס לתוך ה-Integer שבתוך ה - CFuture ci. כעת, על האובייקט ci שיוחזר נפעיל את המתודה thenApply כך שנקבל כפלט CFuture<Integer> c2 עם value שהוא ה-integer שיחזור. לבסוף נפעיל את המתודה thenAccept שהיא המתודה שאחריה לא ניתן לשרשר יותר. היא תחזיר void ותדפיס ערך בלבד. כך יראה הקוד ב- main -

```
CFuture.supplyAsync(() -> {
    try {
        Thread.sleep( millis: 4000);
    } catch (InterruptedException e) {
    }
    return "42";
}).thenApply(s -> Integer.parseInt(s)).thenApply(x -> x * 2)
        .thenAccept(x -> System.out.println(x));
```

Completable Future Demo 13.4

לסיום נממש את המתודות thenApply, thenAccept . המתודה thenAccept מקבלת פרמטר, מבצע עליו פעולה אך לא מחזירה דבר. ניזכר שקיים הממשק java.util.function.Consumer, המקבל פרמטר ולא מחזיר דבר, ולכן נוכל לקבל כפרמטר Consumer ולהפעיל את המתודה accept(value). כלומר, בהינתן consumer, נקבל את ה-value שלו ונחזיר void. כעת מתעוררת בעית עיצוב. המתודה thenAccept לא אמורה להריץ את c.accept(value), שכן ה-value עדיין לא מוכן, הוא null. בנוסף, לא נרצה שהמתודה תהיה blocking call .

נרצה שה- thread שהורץ במתודה supplyAsync הוא שיריץ את השורה c.accept(value) לאחר שיוחזר הערך מ-c.call().

כלומר ישנו קשר בין המתודות set לבין thenAccept. המתודה thenAccept מגדירה מה צריך לעשות, והמתודה set עונה על השאלה מתי צריך לעשות (לאחר קבלת ערך החזרה) .

כאשר אחד מתודה אחת מגדירה "מה", ומתודה אחרת מגדירה "מתי", הפתרון הוא תבנית העיצוב Strategy. נגדיר data member נוסף מסוג Runnable. כעת, במתודה set, לאחר קבלת הvalue - נקרא ל-r.run.

מתי ה-run הזה יופעל? כשה- thread יסיים להריץ את c.call, והמתודה set תקרא. כל שנשאר לנו הוא להגדיר במתודה thenAccept שה-runnable צריך לבצע פונקציה שתפעיל את הפקודה c.accept. כלומר, מתודה זו לא מורצת כרגע אלא רק מוגדרת. המתודה תרוץ לאחר שנבצע set. נשנה את הקוד בהתאם-

```
public void thenAccept(Consumer<V> c) {
    r = () -> c.accept(value);
}
```

```
public void set(V value) {
    this.value = value;
    r.run();
}
```

כעת נממש את המתודה thenApply . במקום לקבל consumer, המתודה תקבל כפרמטר java.util.concurrent.Function כך שבהינתן טיפוס V, יוחזר טיפוס R חדש. R הוא טיפוס שלא מוכר, ולכן כל המתודה תלויה בטיפוס הפרמטרי R. אם נגדיר קריאה ישירה למתודה ע"י func.apply(value) , היא תתבצע כרגע גם אם הערך הוא עדיין null. כדי להימנע מכך, נרצה לדחוף את ה-R שחוזר לאובייקט כל שהוא. לכן במקום להחזיר void מהמתודה, נחזיר אובייקט חדש מהסוג של Cfuture שתלוי ב-R . נבצע set לאובייקט CFuture עם הערך שיוחזר מהתהליך .כמובן שנרצה שזה לא יתבצע באותו ה- thread , אלא נכניס את הפעולה לתוך ה- data member , r מסוג runnable. כך יראה הקוד -

```
public <R> CFuture<R> thenApply(Function<V, R> func) {
    CFuture<R> cfr = new CFuture<>();
    r= ()->cfr.set(func.apply(value));
    return cfr;
}
```

באופן זה, אם ב-main נשרשר פעולות של thenApply, כל קריאה ל-set קוראת ל-r.run, שבתורו קורא שוב ל-set שקורא בתורו שוב ל-r.run() וחוזר חלילה. כל התגובות יתרחשו אחת אחרי השנייה לקבלת תגובת שרשרת. השרשור ייעצר כאשר נקרא -thenAccept, שמחזירה void. כעת, נוסיף הדפסה ב-main כך שנוודא שה- thread של ה-main אכן הגדיר את סט הפעולות המבוקשות לביצוע, שחרר את משאביו ומת. 4 שניות לאחר שה- main ימות, "42" יוזן ל-value וכל תהליך השרשרת יתרחש ,עד להפעלת ה-thenAccept עם ההדפסה הסופית שתבוצע ע"י ה-consumer. כך יראה הפלט -

```
main is dead
84
```