DZone

SOLID Design Principles Explained: Dependency Inversion

by Thorben Janssen MVB · May. 09, 18 · Java Zone · Tutorial

The SOLID design principles were promoted by Robert C. Martin and are some of the best-known design principles in object-oriented software development. SOLID is a mnemonic acronym for the following five principles:

- **S**ingle Responsibility Principle
- **O**pen/Closed Principle
- **L**iskov Substitution Principle
- **I**nterface Segregation Principle
- **D**ependency Inversion Principle

Each of these principles can stand on its own and has the goal to improve the robustness and maintainability of object-oriented applications and software components. But they also add to each other so that applying all of them makes the implementation of each principle easier and more effective.

I explained the first four design principles in previous articles. In this one, I will focus on the Dependency Inversion Principle. It is based on the Open/Closed Principle and the Liskov Substitution Principle. You should, therefore, at least be familiar with these two principles, before you read this article.

Definition of the Dependency Inversion Principle

The general idea of this principle is as simple as it is important: High-level modules, which provide complex logic, should be easily reusable and unaffected by changes in low-level modules, which provide utility features. To achieve that, you need to introduce an abstraction that decouples the high-level and low-level modules from each other.

Based on this idea, Robert C. Martin's definition of the Dependency Inversion Principle consists of two parts:

1. High-level modules should not depend on low-level modules. Both should depend on abstractions.
2. Abstractions should not depend on details. Details should depend on abstractions.

An important detail of this definition is, that high-level **and** low-level modules depend on the abstraction. The design principle does not just change the direction of the dependency, as you might have expected when you read its design principle does not just change the direction of the dependency, as you might have expected when you read its name for the first time. It splits the dependency between the high-level and low-level modules by introducing an abstraction between them. So in the end, you get two dependencies:

1. The high-level module depends on the abstraction, and
2. The low-level depends on the same abstraction.

Based on other SOLID principles

This might sound more complex than it often is. If you consequently apply the Open/Closed Principle and the Liskov Substitution Principle to your code, it will also follow the Dependency Inversion Principle.

The Open/Closed Principle required a software component to be open for extension, but closed for modification. You can achieve that by introducing interfaces for which you can provide different implementations. The interface itself is closed for modification, and you can easily extend it by providing a new interface implementation.

Your implementations should follow the Liskov Substitution Principle so that you can replace them with other implementations of the same interface without breaking your application.

Let’s take a look at the CoffeeMachine project in which I will apply all three of these design principles.

Brewing Coffee With the Dependency Inversion Principle

You can buy lots of different coffee machines. Rather simple ones that use water and ground coffee to brew filter coffee, and premium ones that include a grinder to freshly grind the required amount of coffee beans and which you can use to brew different kinds of coffee.

If you build a coffee machine application that automatically brews you a fresh cup of coffee in the morning, you can model these machines as a *BasicCoffeeMachine* and a *PremiumCoffeeMachine* class.

Implementing the *BasicCoffeeMachine*

The implementation of the *BasicCoffeeMachine* is quite simple. It only implements a constructor and two public methods. You can call the *addGroundCoffee* method to refill ground coffee, and the *brewFilterCoffee* method to brew a cup of filter coffee.

```
import java.util.Map;

public class BasicCoffeeMachine implements CoffeeMachine {

    private Configuration config;
    private Map<CoffeeSelection, GroundCoffee> groundCoffee;
    private BrewingUnit brewingUnit;

    public BasicCoffeeMachine(Map<CoffeeSelection, GroundCoffee> coffee).
        this.groundCoffee = coffee;
        this.groundCoffee = coffee;
        this.brewingUnit = new BrewingUnit();
        this.config = new Configuration(30, 480);
    }

    @Override
    public Coffee brewFilterCoffee() {
        // get the coffee
        GroundCoffee groundCoffee = this.groundCoffee.get(CoffeeSelection.FILTER_COFF
        // brew a filter coffee
       return this.brewingUnit.brew(CoffeeSelection.FILTER_COFFEE, groundCoffee, this
    }

    public void addGroundCoffee(CoffeeSelection sel, GroundCoffee newCoffee) throws C
        GroundCoffee existingCoffee = this.groundCoffee.get(sel);
        if (existingCoffee != null) {
            if (existingCoffee.getName().equals(newCoffee.getName())) {
                existingCoffee.setQuantity(existingCoffee.getQuantity() + newCoffee.g
            } else {
                throw new CoffeeException("Only one kind of coffee supported for each
            }
        } else {
            this.groundCoffee.put(sel, newCoffee)
        }
    }
}
```

Implementing the *PremiumCoffeeMachine*

The implementation of the *PremiumCoffeeMachine* class looks very similar. The main differences are:

- It implements the *addCoffeeBeans* method instead of the *addGroundCoffee* method.
- It implements the additional *brewEspresso* method.

The *brewFilterCoffee* method is identical to the one provided by the *BasicCoffeeMachine*.

```
import java.util.HashMap;
import java.util.Map;

public class PremiumCoffeeMachine {
    private Map<CoffeeSelection, Configuration> configMap;
    private Map<CoffeeSelection, CoffeeBean> beans;
    private Grinder grinder
    private BrewingUnit brewingUnit;
```

```java

    public PremiumCoffeeMachine(Map<CoffeeSelection, CoffeeBean> beans) {
        this.beans = beans;
        this.grinder = new Grinder();
        this.brewingUnit = new BrewingUnit();
        this.configMap = new HashMap<>();
        this.configMap.put(CoffeeSelection.FILTER_COFFEE, new Configuration(30, 480))
        this.configMap.put(CoffeeSelection.ESPRESSO, new Configuration(8, 28));
    }

    public Coffee brewEspresso() {
        Configuration config = configMap.get(CoffeeSelection.ESPRESSO);
        // grind the coffee beans
        GroundCoffee groundCoffee = this.grinder.grind(
            this.beans.get(CoffeeSelection.ESPRESSO),
            config.getQuantityCoffee())
        // brew an espresso
        return this.brewingUnit.brew(CoffeeSelection.ESPRESSO, groundCoffee,
            config.getQuantityWater());
    }

    public Coffee brewFilterCoffee() {
        Configuration config = configMap.get(CoffeeSelection.FILTER_COFFEE);
        // grind the coffee beans
        GroundCoffee groundCoffee = this.grinder.grind(
            this.beans.get(CoffeeSelection.FILTER_COFFEE),
            config.getQuantityCoffee());
        // brew a filter coffee
        return this.brewingUnit.brew(CoffeeSelection.FILTER_COFFEE, groundCoffee,
            config.getQuantityWater());
    }

    public void addCoffeeBeans(CoffeeSelection sel, CoffeeBean newBeans) throws Coffe
        CoffeeBean existingBeans = this.beans.get(sel);
        if (existingBeans != null) {
            if (existingBeans.getName().equals(newBeans.getName())) {
                existingBeans.setQuantity(existingBeans.getQuantity() + newBeans.getQ
            } else {
                throw new CoffeeException("Only one kind of coffee supported for each
            }
         } else {
             this.beans.put(sel, newBeans);
         }
    }
    }
}
```

To implement a class that follows the Dependency Inversion Principle and can use the *BasicCoffeeMachine* or the *PremiumCoffeeMachine* class to brew a cup of coffee, you need to apply the Open/Closed and the Liskov Substitution Principle. That requires a small refactoring during which you introduce interface abstractions for both classes.

Introducing Abstractions

The main task of both coffee machine classes is to brew coffee. But they enable you to brew different kinds of coffee. If you use a *BasicCoffeeMachine*, you can only brew filter coffee, but with a *PremiumCoffeeMachine*, you can brew filter coffee or espresso. So, which interface abstraction would be a good fit for both classes?

As all coffee lovers will agree, there are huge differences between filter coffee and espresso. That's why we are using different machines to brew them, even so, some machines can do both. I, therefore, suggest to create two independent abstractions:

- The *FilterCoffeeMachine* interface defines the *Coffee brewFilterCoffee()* method and gets implemented by all coffee machine classes that can brew a filter coffee.
- All classes that you can use to brew an espresso, implement the *EspressoMachine* interface, which defines the *Coffee brewEspresso()* method.

As you can see in the following code snippets, the definition of both interface is pretty simple.

```
public interface CoffeeMachine {
    Coffee brewFilterCoffee();
}

public interface EspressoMachine {
    Coffee brewEspresso();
}
```

In the next step, you need to refactor both coffee machine classes so that they implement one or both of these interfaces.

Refactoring the *BasicCoffeeMachine* class

Let's start with the *BasicCoffeeMachine* class. You can use it to brew a filter coffee, so it should implement the *CoffeeMachine* interface. The class already implements the *brewFilterCoffee()* method. You only need to add *implements CoffeeMachine* to the class definition.

```
public class BasicCoffeeMachine implements CoffeeMachine {
    private Configuration config;
    private Map<CoffeeSelection, GroundCoffee> groundCoffee;
    private Map<CoffeeSelection, GroundCoffee> groundCoffee;
    private BrewingUnit brewingUnit;

    public BasicCoffeeMachine(Map<CoffeeSelection, GroundCoffee> coffee) {
        this.groundCoffee = coffee;
        this.brewingUnit = new BrewingUnit();
        this.config = new Configuration(30, 480);
    }

    @Override
    public Coffee brewFilterCoffee() {
        // get the coffee
        GroundCoffee groundCoffee = this.groundCoffee.get(CoffeeSelection.FILTER_COFF
        // brew a filter coffee
        return this.brewingUnit.brew(CoffeeSelection.FILTER_COFFEE, groundCoffee, thi
    }

    public void addGroundCoffee(CoffeeSelection sel, GroundCoffee newCoffee) throws C
        GroundCoffee existingCoffee = this.groundCoffee.get(sel);
        if (existingCoffee != null) {
            if (existingCoffee.getName().equals(newCoffee.getName())) {
                existingCoffee.setQuantity(existingCoffee.getQuantity() + newCoffee.g
            } else {
             throw new CoffeeException("Only one kind of coffee supported for each Co
           }
        } else {
            this.groundCoffee.put(sel, newCoffee);
        }
    }
}
```

Refactoring the *PremiumCoffeeMachine* Class

The refactoring of the *PremiumCoffeeMachine* also doesn't require a lot of work. You can use the coffee machine to brew filter coffee and espresso, so the *PremiumCoffeeMachine* class should implement the *CoffeeMachine* and the *EspressoMachine* interfaces. The class already implements the methods defined by both interfaces. You just need to declare that it implements the interfaces.

```
import java.util.HashMap;
import java.util.Map;

public class PremiumCoffeeMachine implements CoffeeMachine, EspressoMachine {
    private Map<CoffeeSelection, Configuration> configMap;
    private Map<CoffeeSelection, CoffeeBean> beans;
```

```java
    private Grinder grinder;
    private BrewingUnit brewingUnit;

    public PremiumCoffeeMachine(Map<CoffeeSelection, CoffeeBean> beans) {
        this.beans = beans;
        this.grinder = new Grinder();
        this.brewingUnit = new BrewingUnit();
        this.configMap = new HashMap<>();
        this.configMap.put(CoffeeSelection.FILTER_COFFEE, new Configuration(30, 480))
        this.configMap.put(CoffeeSelection.ESPRESSO, new Configuration(8, 28));
    }

    @Override
    public Coffee brewEspresso() {
        Configuration config = configMap.get(CoffeeSelection.ESPRESSO);
        // grind the coffee beans
        GroundCoffee groundCoffee = this.grinder.grind(
           this.beans.get(CoffeeSelection.ESPRESSO),
           config.getQuantityCoffee());
       // brew an espresso
       return this.brewingUnit.brew(CoffeeSelection.ESPRESSO, groundCoffee,
           config.getQuantityWater());
    }

    @Override
    public Coffee brewFilterCoffee() {
        Configuration config = configMap.get(CoffeeSelection.FILTER_COFFEE);
        // grind the coffee beans
        GroundCoffee groundCoffee = this.grinder.grind(
            this.beans.get(CoffeeSelection.FILTER_COFFEE),
            config.getQuantityCoffee());
        // brew a filter coffee
        return this.brewingUnit.brew(CoffeeSelection.FILTER_COFFEE,
            groundCoffee,config.getQuantityWater());
    }

    public void addCoffeeBeans(CoffeeSelection sel, CoffeeBean newBeans) throws Coffe
        CoffeeBean existingBeans = this.beans.get(sel);
        if (existingBeans != null) {
            if (existingBeans.getName().equals(newBeans.getName())) {
                existingBeans.setQuantity(existingBeans.getQuantity() + newBeans.getQ
            } else {
                throw new CoffeeException("Only one kind of coffee supported for each
            }
        } else {
            this.beans.put(sel, newBeans);
        }
    }
}
```

The *BasicCoffeeMachine* and the *PremiumCoffeeMachine* classes now follow the Open/Closed and the Liskov Substitution principles. The interfaces enable you to add new functionality without changing any existing code by adding new interface implementations. And by splitting the interfaces into *CoffeeMachine* and *EspressoMachine*, you separate the two kinds of coffee machines and ensure that all *CoffeeMachine* and *EspressMachine* implementations are interchangeable.

Implementing the Coffee Machine Application

You can now create additional, higher-level classes that use one or both of these interfaces to manage coffee machines without directly depending on any specific coffee machine implementation.

As you can see in the following code snippet, due to the abstraction of the *CoffeeMachine* interface and its provided functionality, the implementation of the *CoffeeApp* is very simple. It requires a *CoffeeMachine* object as a constructor parameter and uses it in the *prepareCoffee* method to brew a cup of filter coffee.

```
public class CoffeeApp {
    private CoffeeMachine coffeeMachine;

    public CoffeeApp(CoffeeMachine coffeeMachine) {
     this.coffeeMachine = coffeeMachine
    }

    public Coffee prepareCoffee(CoffeeSelection selection
        throws CoffeeException {
        Coffee coffee = this.coffeeMachine.brewFilterCoffee();
        System.out.println("Coffee is ready!");
        return coffee;
    }
}
```

The only code that directly depends on one of the implementation classes is the *CoffeeAppStarter* class, which instantiates a *CoffeeApp* object and provides an implementation of the *CoffeeMachine* interface. You could avoid this compile-time dependency entirely by using a dependency injection framework, like Spring or CDI, to resolve the dependency at runtime.

```
import java.util.HashMap;
import java.util.Map;

public class CoffeeAppStarter {
    public static void main(String[] args) {
        // create a Map of available coffee beans
        Map<CoffeeSelection, CoffeeBean> beans = new HashMap<CoffeeSelection, CoffeeB
        beans.put(CoffeeSelection.ESPRESSO, new CoffeeBean(
            "My favorite espresso bean", 1000));
        beans.put(CoffeeSelection.FILTER_COFFEE, new CoffeeBean(
             "My favorite filter coffee bean", 1000))
        // get a new CoffeeMachine object
        PremiumCoffeeMachine machine = new PremiumCoffeeMachine(beans);
        // Instantiate CoffeeApp
        CoffeeApp app = new CoffeeApp(machine);
        // brew a fresh coffee
        try {
           app.prepareCoffee(CoffeeSelection.ESPRESSO);
        } catch (CoffeeException e) {
            e.printStackTrace();
        }
    }
}
```

Summary

The Dependency Inversion Principle is the fifth and final design principle that we discussed in this series. It introduces an interface abstraction between higher-level and lower-level software components to remove the dependencies between them.

As you have seen in the example project, you only need to consequently apply the Open/Closed and the Liskov Substitution principles to your code base. After you have done that, your classes also comply with the Dependency Inversion Principle. This enables you to change higher-level and lower-level components without affecting any other classes, as long as you don’t change any interface abstractions.

If you enjoyed this article, you should also read my other articles about the SOLID design principles:

- **S**ingle Responsibility Principle
- **O**pen/Closed Principle
- **L**iskov Substitution Principle
- **I**nterface Segregation Principle
- **D**ependency Inversion Principle

Like This Article? Read More From DZone

Perfecting Your SOLID Meal With DIP

The Dependency Inversion Principle for Beginners, Without Diagrams

Don’t Blame the Dependency Injection Framework

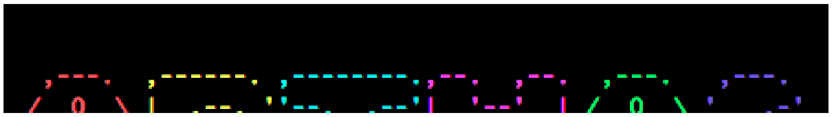
Free DZone Refcard
Getting Started With Vaadin 10

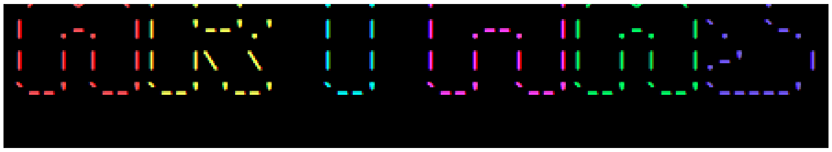
Topics: JAVA , SOLID , DEPENDENCY INVERSION , TUTORIAL

Published at DZone with permission of Thorben Janssen , DZone MVB. See the original article here.
Opinions expressed by DZone contributors are their own.

Troubleshooting Java Applications With Arthas

by Huxing Zhang · Sep 25, 18 · Java Zone · Tutorial





Arthas is a Java Diagnostic tool open sourced by Alibaba. Arthas can help developer trouble-shoot production issues for Java applications without modifying your code or restarting your server.

Let's start with a simple demo to see how we can use Arthas to trouble-shoot this Java program on-the-fly.

Java Program Demo

```
import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicInteger;
public class Demo {
    static class Counter {
        private static AtomicInteger count = new AtomicInteger(0);
        public static void increment() {
            count.incrementAndGet();
        }
        public static int value() {
            return count.get();
        }
    }

    public static void main(String[] args) throws InterruptedException {
        while (true) {
            doSomething();
            TimeUnit.SECONDS.sleep(1);
        }
    }

    private static int doSomething() {
        for (int i = 0; i< 1000; i++) {
            Counter.increment();
        }
        int value = Counter.value();
        return value;
    }
}
```

Save the following code to a `Demo.java` and run the commands in Shell:

```
javac Demo.java && java Demo
```

Install Arthas

```
curl -L https://alibaba.github.io/arthas/install.sh | sh
```

Start Arthas

To start Arthas, simply type the following command:

```
./as.sh
```

This script will automatically display the available Java process that you can attach. In this example, we have only two available Java processes to attach.

```
Arthas script version: 3.0.4
Found existing java process, please choose one and hit RETURN.
* [1]: 9617
  [2]: 37374 Demo
```

Since we are going to attach the Demo.Java process, we type `2` and hit `RETURN`, then Arthas will proceed to attach to the specified process.

```
Calculating attach execution time...
Attaching to 37374 using version 3.0.4...

real    0m0.986s
user    0m0.379s
sys     0m0.058s
Attach success.
Connecting to arthas server... current timestamp is 1537684660
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
  ,---.  ,------. ,--------.,--.  ,--.  ,---.   ,---.
 /  O  \ |  .--. ''--.  .--'|  '--'  | /  O  \ '   .-'
|  .-.  ||  '--'.'   |  |   |  .--.  ||  .-.  |`.  `-.
|  | |  ||  |\  \    |  |   |  |  |  ||  | |  |.-'    |
`--' `--'`--' '--'   `--'   `--'  `--'`--' `--'`-----'


wiki: https://alibaba.github.io/arthas
```

```
version: 3.0.4
pid: 37374
timestamp: 1537684661535

$
```

If you can see the Arthas logo, then congratulations, you have successfully attached to the target JVM!

Next, let's go through some of the cool features that Arthas provides and have a quick evaluation of what Arthas can do for you.

TroubleShoot With Arthas

Dashboard: Overview of Your Java Process

Type in `dashboard` and hit the `ENTER` , you will see it in the following code (press `Ctrl+C` to stop):

```
$ dashboard
ID     NAME                   GROUP          PRIORI STATE  %CPU    TIME   INTERRU DAE
17     pool-2-thread-1        system         5      WAITIN 67      0:0    false   fal
27     Timer-for-arthas-dashb system         10     RUNNAB 32      0:0    false   tru
11     AsyncAppender-Worker-a system         9      WAITIN 0       0:0    false   tru
9      Attach Listener        system         9      RUNNAB 0       0:0    false   tru
3      Finalizer              system         8      WAITIN 0       0:0    false   tru
2      Reference Handler      system         10     WAITIN 0       0:0    false   tru
4      Signal Dispatcher      system         9      RUNNAB 0       0:0    false   tru
26     as-command-execute-dae system         10     TIMED_ 0       0:0    false   tru
13     job-timeout            system         9      TIMED_ 0       0:0    false   tru
1      main                   main           5      TIMED_ 0       0:0    false   fal
14     nioEventLoopGroup-2-1  system         10     RUNNAB 0       0:0    false   fal
18     nioEventLoopGroup-2-2  system         10     RUNNAB 0       0:0    false   fal
23     nioEventLoopGroup-2-3  system         10     RUNNAB 0       0:0    false   fal
15     nioEventLoopGroup-3-1  system         10     RUNNAB 0       0:0    false   fal
Memory             used   total max    usage GC
heap               32M    155M  1820M  1.77% gc.ps_scavenge.count  4
ps_eden_space      14M    65M   672M   2.21% gc.ps_scavenge.time(m 166
ps_survivor_space  4M     5M    5M           s)
ps_old_gen         12M    85M   1365M  0.91% gc.ps_marksweep.count 0
nonheap            20M    23M   -1           gc.ps_marksweep.time( 0
code_cache         3M     5M    240M   1.32% ms )
Runtime
os.name                Mac OS X
os.version             10.13.4
java.version           1.8.0_162
java.home              /Library/Java/JavaVir
                       tualMachines/jdk1.8.0
                       _162.jdk/Contents/Hom
                       e/jre
```

By default, the content will be refreshed every five seconds. On top of the dashboard, you can see a list of running threads with some information, like thread ID, name, group, priority, state, CPU usage, running time, etc. They are ranked by CPU usage.

In the middle section, there are two sub-sections. On the left, it shows the memory usage with heap usage and non-heap usage. In the heap usage, you can see the usage of the young generation, including the Eden/survivor region and the old generation.

On the right, it shows the Garbage Collection statistics, including the Garbage Collection algorithm, as well as the time and count for both the young and old generations.

In the bottom section, it shows basic system information, e.g. the Java version, operating system, and Java home location.

SC: Search Class Loaded by Your JVM

Sometimes, you want to know whether a classed is loaded and where is it loaded from. SC is the command for you.

```
$ sc Demo$Counter -d
 class-info        Demo$Counter
 code-source       /private/var/tmp/
 name              Demo$Counter
 isInterface       false
 isAnnotation      false
 isEnum            false
 isAnonymousClass  false
 isArray           false
 isLocalClass      false
 isMemberClass     true
 isPrimitive       false
 isSynthetic       false
 simple-name       Counter
 modifier          static
 annotation
 interfaces
 super-class       +-java.lang.Object
 class-loader      +-sun.misc.Launcher$AppClassLoader@2a139a55
                     +-sun.misc.Launcher$ExtClassLoader@51d04a9
 classLoaderHash   2a139a55
```

```

Affect(row-cnt:1) cost in 13 ms.
```

The `sc Demo$Counter` tells Arthas to search for a class from all the loaded classes of this JVM. The `-d` option means to display the metadata in a more detail way. For example, where the class is loaded from, the modifiers, and the classloader hierarchy.

Jad: Decompile Class Into Source Code

Sometimes, you want to confirm whether your application is running the correct code or not. To do that, you have to decompile the class into the source code. The `jad` is the right command for you.

```
$ jad Demo

ClassLoader:
+-sun.misc.Launcher$AppClassLoader@2a139a55
  +-sun.misc.Launcher$ExtClassLoader@51d04a9

Location:
/private/var/tmp/

/*
 * Decompiled with CFR 0_132.
 */
import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicInteger;

public class Demo {
    public static void main(String[] arrstring) throws InterruptedException {
        do {
            Demo.doSomething();
            TimeUnit.SECONDS.sleep(1L);
        } while (true);
    }

    private static int doSomething() {
        int n;
        for (n = 0; n < 1000; ++n) {
            Counter.increment();
        }
        n = Counter.value();
        return n;
    }

    static class Counter {
        private static AtomicInteger count = new AtomicInteger(0);

        Counter() {
        }

        public static int value() {
            return count.get();
        }

        public static void increment() {
            count.incrementAndGet();
        }
    }
}
```

Watch: View Method Invocation Input and Results

Next, we would like to look at the return value of `Counter.value()` . Normally, we have to add the following:

```
System.out.println(value);
```

And, we will need to restart the demo to see the return value. However, with Arthas, you don't have to add any code or restart. You can use the `watch` command instead.

```
$ watch Demo$Counter value returnObj
Press Ctrl+C to abort.
Affect(class-cnt:1 , method-cnt:1) cost in 62 ms.
ts=2018-09-23 15:36:54;result=@Integer[24000]
ts=2018-09-23 15:36:55;result=@Integer[25000]
ts=2018-09-23 15:36:56;result=@Integer[26000]
ts=2018-09-23 15:36:57;result=@Integer[27000]
ts=2018-09-23 15:36:58;result=@Integer[28000]
```

The basic watch command syntax is `watch class-name method-name content` . Here, since Counter is an inner class of Demo, we have to use `Deme$Counter` to denote it. `Value` is the name of the method. R `eturnObj` is one of the pre-defined keywords that we can use to denote the content to watch. Other possible keywords include `params` , `throwExp` , `target` , `classloader,` and more. We will talk about that later. Here, we just need to know that `returnObj` means that we want to watch the return object of the specified method.

The next line:

```
Press Ctrl+C to abort.
```

This means that Arthas will start another thread to process this command and output the console. If you would like to stop watching, you can press Ctrl+C to abort.

The next line can be found below:

```
Affect(class-cnt:1 , method-cnt:1) cost in 62 ms.
```

This means that Arthas will search for the specified class and method in the Java process and change the byte-code of the method to add some logic. This means that there is one class and one method that has been modified by Arthas.

Once the method has been called, it will output the content you specified to watch:

```
ts=2018-09-23 15:36:54;result=@Integer[24000]
```

This includes a timestamp and the result, which is an integer of 24000.

Of course, this is a simple case, and if the returned object is complicated and you would like to what is inside, you can use the `ognl` expression to customize it. However, this is beyond the scope of this article.

Trace: Find the Bottleneck of Your Method Invocation

Next, whenever you want to find out whether there is a bottleneck, e.g. why my method running slowly, you should try the `trace` command. It can list all the sub method invocations with the number of invocations and the total execution time. This is very helpful to identify slow method invocations or an unexpected number of invocations.

Next, let's find out what is going on inside `doSomething` using the `trace` command.

```
$ trace Demo doSomething
Press Ctrl+C to abort.
Affect(class-cnt:1 , method-cnt:1) cost in 101 ms.
`---ts=2018-09-24 14:43:58;thread_name=main;id=1;is_daemon=false;priority=5;TCCL=sun.
    `---[25.927722ms] Demo:doSomething()
        +---[min=0.00184ms,max=5.062901ms,total=9.786846ms,count=1000] Demo$Counter:i
        `---[0.011398ms] Demo$Counter:value()
```

The first several lines have been explained above, so here, we will just skip it. Let start with:

```
14:43:58;thread_name=main;id=1;is_daemon=false;priority=5;TCCL=sun.misc.Launcher$AppC
```

This line shows some important information on calling `Thread`, which includes the thread name, ID, priority, and `ThreadContextClassloader`.

Next, let's look at the following lines:

```
 `---[25.927722ms] Demo:doSomething()
        +---[min=0.00184ms,max=5.062901ms,total=9.786846ms,count=1000] Demo$Counter:i
        `---[0.011398ms] Demo$Counter:value()
```

This shows the sub-method invocation tree of `doSomething`, It means in `doSomething` that two methods have been called; they are: `Demo$Counter:increment()` and `Demo$Counter:value()`. The former method has been called 1000 times, with a total time of `9.786846ms`, and the latter method has been called once, with a total time of `0.011398ms`.

Monitor: View Method Invocation Statistics

Sometimes, we want to have a performance overview for a specified method to ensure that it is working correctly. In this case, `Monitor` is a good friend to have.

```
$ monitor Demo$Counter increment -c 5
Press Ctrl+C to abort.
Affect(class-cnt:1 , method-cnt:1) cost in 17 ms.
 timestamp            class         method     total  success  fail  avg-rt(ms)  fail
--------------------------------------------------------------------------------------
 2018-09-24 15:13:31  Demo$Counter  increment  5000   5000     0     0.01        0.00

 timestamp            class         method     total  success  fail  avg-rt(ms)  fail
--------------------------------------------------------------------------------------
 2018-09-24 15:13:36  Demo$Counter  increment  5000   5000     0     0.00        0.00

 timestamp            class         method     total  success  fail  avg-rt(ms)  fail
--------------------------------------------------------------------------------------
 2018-09-24 15:13:41  Demo$Counter  increment  5000   5000     0     0.00        0.00
```

The `-c 5` option tells Arthas to output the result every five seconds. And, the output shows the total number of invocations, successive invocations (no exception is thrown out of the method), failed invocations, average response time, and failure rates during the monitor period.

Stack: View Call Stack of the Method

Sometimes, we want to know who is calling this method. Normally, when an exception is raised, there will be a stack trace. But, when your method is acting abnormally, you should be able to know who is causing this behavior without any change of your code. Here comes the `stack` command to help you solve it.

```
$ stack  Demo$Counter increment -n 1
Press Ctrl+C to abort.
Affect(class-cnt:1 , method-cnt:1) cost in 17 ms.
ts=2018-09-24 15:20:33;thread_name=main;id=1;is_daemon=false;priority=5;TCCL=sun.misc
    @Demo.doSomething()
        at Demo.main(Demo.java:16)
```

The `stack Demo$Counter increment` basically means give me the stack trace if `Demo$Counter#increment` is called. The option `-n 1` tells Arthas to only output the stack trace once. We are already familiar with most of the output, except for the last several lines, which outputs the stack trace of the `increment` method.

Tt: Time Tunnel of Method Invocations

Sometimes, there may be too many invocations to a method, and whether you want to check the input parameter or the returned object is undecided, because you have no idea what to check. At this time, you'd better record all invocations so that you can analyze them later. At this time, you will need the `tt` command, which does exactly what you need.

```
$ tt Demo$Counter value -t
Press Ctrl+C to abort.
Affect(class-cnt:1 , method-cnt:1) cost in 14 ms.
 INDEX   TIMESTAMP              COST(ms)   IS-RET  IS-EXP  OBJECT  CLASS          MET
-------------------------------------------------------------------------------------
 1000    2018-09-24 15:24:14    0.197211   true    false   NULL    Demo$Counter    va
 1001    2018-09-24 15:24:15    0.091722   true    false   NULL    Demo$Counter    va
 1002    2018-09-24 15:24:16    0.11092    true    false   NULL    Demo$Counter    va
 1003    2018-09-24 15:24:17    0.078451   true    false   NULL    Demo$Counter    va
```

`tt Demo$Counter value -t` basically means we need to record every invocation to `Demo$Counter#value` , each one of the recordings is called a time fragment. The result table shows the ID of the time fragment, the timestamp that happened, the execution time, and whether it successfully returned or ended it up with an exception thrown. The last column `OBJECT` represents the actual object that is called. Here, it is `null` because it is a static method.

After this recording, you can list all the recordings:

```
$ tt -l
 INDEX   TIMESTAMP            COST(ms)   IS-RET  IS-EXP   OBJECT  CLASS         METHO
-------------------------------------------------------------------------------------
 1000    2018-09-24 15:24:14  0.197211   true    false    NULL    Demo$Counter  value
 1001    2018-09-24 15:24:15  0.091722   true    false    NULL    Demo$Counter  value
 1002    2018-09-24 15:24:16  0.11092    true    false    NULL    Demo$Counter  value
 1003    2018-09-24 15:24:17  0.078451   true    false    NULL    Demo$Counter  value
```

You can also check the method invocation details just as the `watch` command did. For example:

```
$ tt -i 1000 -w returnObj
@Integer[12444000]
Affect(row-cnt:1) cost in 181 ms.
```

This basically means the returned value of the time fragment with id=1000.

Exit Arthas

If you have finished troubleshooting with Arthas, there are two ways to quit Arthas:

`quit` or `exit`

These two commands will disconnect the current console connection while Arthas is still running in the target process. If you would like to connect again, you can simply use telnet to connect `telnet localhost 3658` .

`shutdown`

This command will terminate the Arthas completely. You have to run `as.sh` to attach again.

Conclusion

Until now, you had only taken a quick look at the installation and quick start of Arthas. We used this simple demo Java program to go through some interesting features provided by Arthas. Finally, we learned that how to quit or shutdown Arthas. Hopefully, this article can give you an overview of how Arthas can help you to quickly troubleshoot Java applications in production. You can also refer to this documentation for further reading. It's time to give it a try!

Like This Article? Read More From DZone

Free, Fast, Open, Production-Proven, and All Java: OpenJ9

OpenTracing Spring Boot Instrumentation

OpenCSV: Properly Handling Backslashes

Free DZone Refcard
Getting Started With Vaadin 10

Topics: JAVA, AGENT, TRACE, JVM, TROUBLESHOOT, TUTORIAL

Opinions expressed by DZone contributors are their own.