《Java 性能优化实战 21 讲》

李国 前京东、陌陌高级架构师

— 拉勾教育出品 —

11 | 案例分析：如何用设计模式优化性能？

案例分析：如何用设计模式优化性能？

拉勾教育
互联网人实战大学

代码的结构对应用的整体性能
有着重要的影响

案例分析：如何用设计模式优化性能？

拉勾教育
互联网人实战大学

设计模式——对常用开发技巧进行的总结

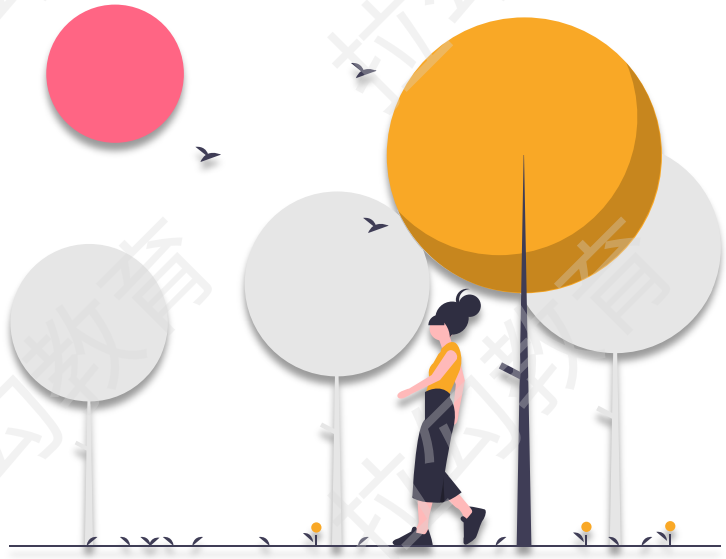
02课时

I/O 模块使用的是装饰器模式

本课时

举例讲解和性能相关的几个设计模式

包括代理模式、单例模式、享元模式、原型模式等

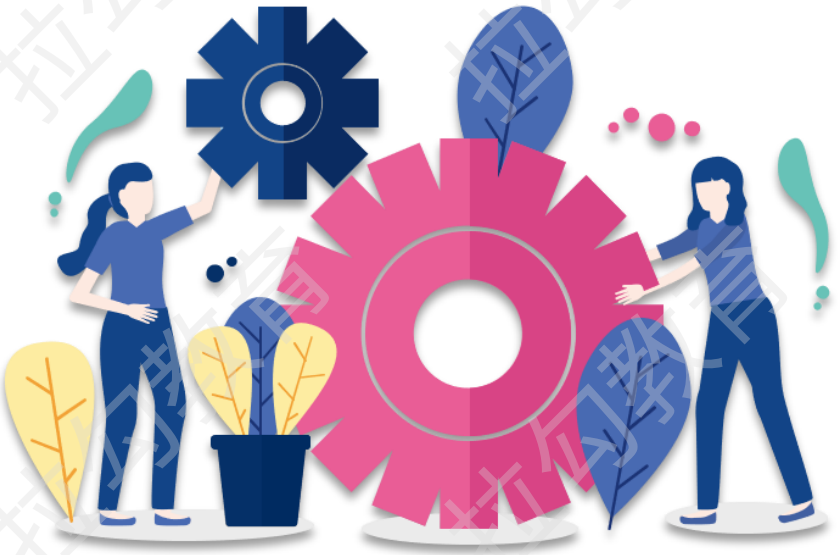


如何找到动态代理慢逻辑的原因?

Spring 使用 CGLIB 对 Java 的字节码进行了增强

在复杂的项目中，会有非常多的 AOP 代码，比如权限、日志等切面

分析一个使用 arthas 找到动态代理慢逻辑的具体原因



如何找到动态代理慢逻辑的原因?

创建一个最简单的 Bean

```
@Component
public class ABean {
    public void method() {
        System.out.println("**********");
    }
}
```

如何找到动态代理慢逻辑的原因?

拉勾教育
互联网人实战大学

使用 Aspect 注解，完成切面的书写

```
@Aspect
@Component
public class MyAspect {
  @Pointcut("execution(* com.github.xjjdog.spring.ABean.*(..))")
  public void pointcut() {
  }

  @Before("pointcut()")
  public void before() {
    System.out.println("before");
    try {
      Thread.sleep(TimeUnit.SECONDS.toMillis(1));
    } catch (InterruptedException e) {
      throw new IllegalStateException();
    }
  }
}
```

如何找到动态代理慢逻辑的原因?

创建一个 Controller

```
@Controller
public class AopController {
    @Autowired
    private ABean aBean;

    @ResponseBody
    @GetMapping("/aop")
    public String aop(){
        long begin = System.currentTimeMillis();
        aBean.method();
        long cost = System.currentTimeMillis() - begin;
        String cls = aBean.getClass().toString();
        return cls + " | " + cost;
    }
}
```

如何找到动态代理慢逻辑的原因?

执行结果如下

```
class
com.github.xjjdog.spring.ABean$$EnhancerBySpri
ngCGLIB$$a5d91535 | 1023
```

如何找到动态代理慢逻辑的原因?

使用 arthas 分析执行过程，找出耗时最高的 AOP 方法

```
(base) xjjdog@mymac ~/soft/arthas $ java -jar arthas-boot.jar --repo-mirror
[INFO] arthas-boot version: 3.3.7
[INFO] Found existing java process, please choose one and input the serial n
* [1]: 31683 org.jetbrains.jps.cmdline.Launcher
  [2]: 31684 com.github.xjjdog.spring.App
  [3]: 31038 org.jetbrains.idea.maven.server.RemoteMavenServer36
  [4]: 1199
```

如何找到动态代理慢逻辑的原因?

trace com.github.xjjdog.spring.ABean method

```
[arthas@31684]$ trace com.github.xjjdog.spring.ABean method
Press Q or Ctrl+C to abort.
Affect(class count: 2 , method count: 2) cost in 113 ms, listenerId: 4
`---ts=2020-07-25 20:20:31;thread_name=http-nio-8080-exec-9;id=82;is_daemon=true;p
omcat.TomcatEmbeddedWebappClassLoader@40e60ece
    `---[1007.392761ms] com.github.xjjdog.spring.ABean$$EnhancerBySpringCGLIB$$a5c
        `---[1007.208091ms] org.springframework.cglib.proxy.MethodInterceptor:inte
            `---[0.264416ms] com.github.xjjdog.spring.ABean:method()
```

代理模式

代理模式（Proxy）可以通过一个代理类，来控制对一个对象的访问

Java 中实现动态代理主要有两种模式：

- **JDK** 方式是面向接口的，主要的相关类是 InvocationHandler 和 Proxy
- **CGLib** 代理普通类，主要的相关类是 MethodInterceptor 和 Enhancer

JDK 方式和 CGLib 方式代理速度的 JMH 测试结果

Benchmark	Mode	Cnt	Score	Error	Units
ProxyBenchmark.cglib	thrpt	10	78499.580	± 1771.148	ops/ms
ProxyBenchmark.jdk	thrpt	10	88948.858	± 814.360	ops/ms

代理模式

代理的创建速度

Benchmark	Mode	Cnt	Score	Error	Units
ProxyCreateBenchmark.cglib	thrpt	10	7281.487	± 1339.779	ops/ms
ProxyCreateBenchmark.jdk	thrpt	10	15612.467	± 268.362	ops/ms

单例模式

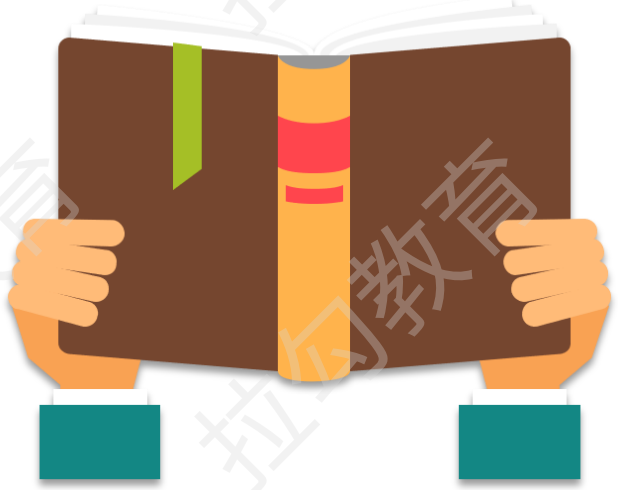
Spring 在创建组件时，可以通过 scope 注解指定它的作用域

用来标示这是一个 **prototype（多例）**还是 **singleton（单例）**

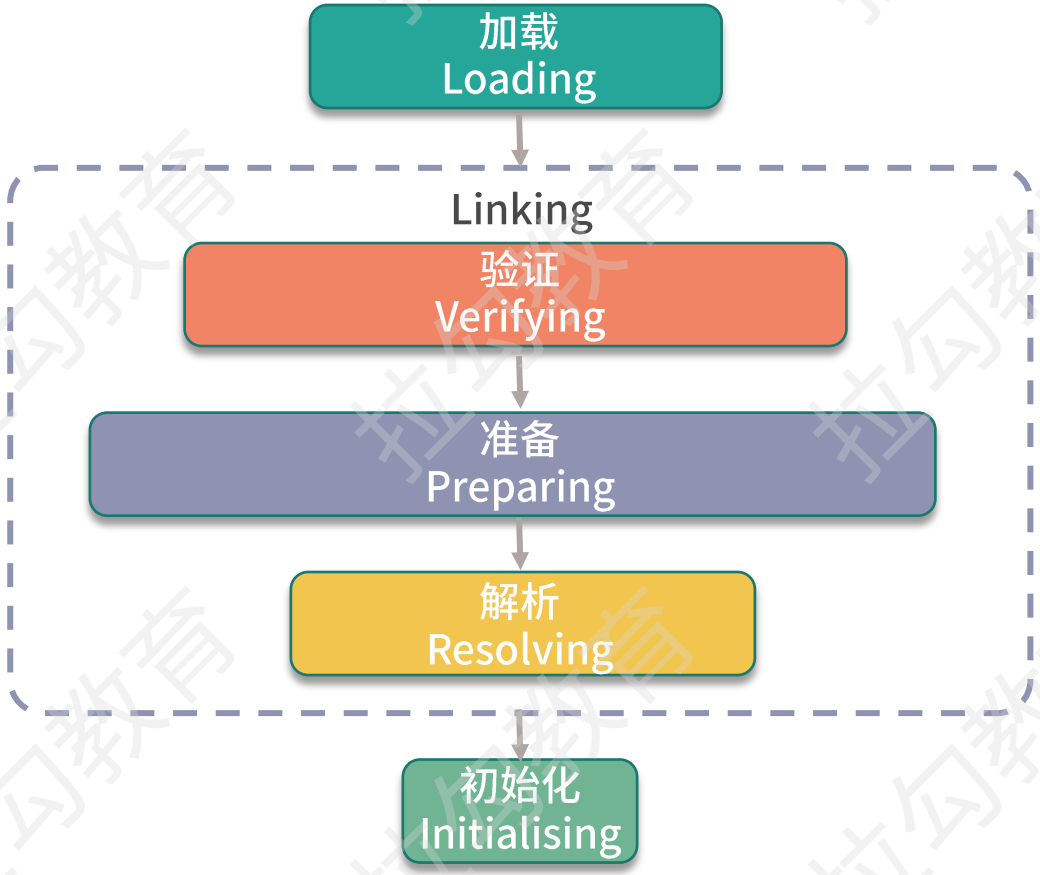
当指定为单例时（默认行为），在 Spring 容器中，组件有且只有一份

普通的单例类，将单例的构造方法设置成私有的

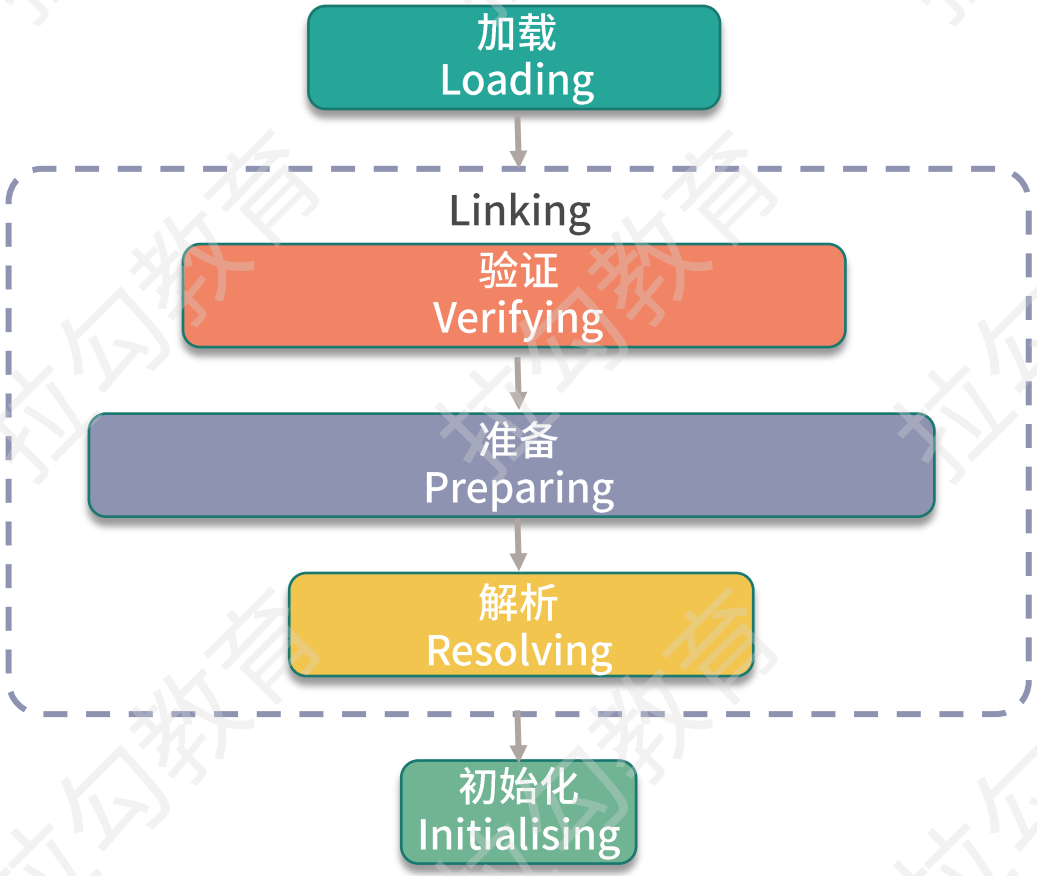
单例有**懒汉加载**和**饿汉加载**模式



单例模式


加载
Loading
Linking
验证
Verifying
准备
Preparing
解析
Resolving
初始化
Initialising

单例模式



static 字段和 static 代码块

在类加载的初始化阶段就已经被执行

<cinit> 方法属于类的（构造方法）

单例模式

把单例的初始化动作，放在<cinit>方法里，能够实现**饿汉模式**

```
private static Singleton instace = new Singleton();
```

单例模式

在 new 一个新对象时，调用构造方法<init>，用来初始化对象的属性

多个线程可以同时调用<init>函数，使用 synchronized 关键字对生成过程进行同步

公认的兼顾线程安全和效率的单例模式——**double check**

单例模式

单例模式


```
public class DoubleCheckSingleton {

    private volatile static DoubleCheckSingleton instance = null;

    private DoubleCheckSingleton() {
    }

    public static DoubleCheckSingleton getInstance() {
        if (null == instance) {
            synchronized (DoubleCheckSingleton.class) {
                if (null == instance) {
                    instance = new DoubleCheckSingleton();
                }
            }
        }
        return instance;
    }
}
```

单例模式


```
public class DoubleCheckSingleton {

    private volatile static DoubleCheckSingleton instance = null;

    private DoubleCheckSingleton() {
    }

    public static DoubleCheckSingleton getInstance() {
        if (null == instance) {
            synchronized (DoubleCheckSingleton.class) {
                if (null == instance) {
                    instance = new DoubleCheckSingleton();
                }
            }
        }
        return instance;
    }
}
```

double check 现在是一种反模式，不推荐使用

单例模式

推荐使用 enum 实现懒加载的单例

```
public class EnumSingleton {
    private EnumSingleton(){
    }
    public static EnumSingleton getInstance(){
        return Holder.HOLDER.instance;
    }
    private enum Holder {
        HOLDER;
        private final EnumSingleton instance;
        Holder() {
            instance = new EnumSingleton();
        }
    }
}
```

享元模式

拉勾教育
互联网人实战大学

享元模式（Flyweight）通过共享技术，最大限度地复用对象

一般会使用唯一的标识码进行判断，然后返回对应的对象

比如池化对象和对象复用等

单例模式是享元模式的一种特殊情况，它通过共享单个实例，达到对象的复用

享元模式


```
Map<String,Strategy> strategys = new HashMap<>();
strategys.put("a",new AStrategy());
strategys.put("b",new BStrategy());
```

原型模式

原型模式（Prototype）首先创建一个实例，然后通过这个实例进行新对象的创建

在 Java 中，最典型的是 Object 类的 **clone 方法**

clone 如果只拷贝当前层次的对象，实现的只是浅拷贝

实现深拷贝，还有序列化等手段，比如实现 Serializable 接口，或者把对象转化成 JSON

小结

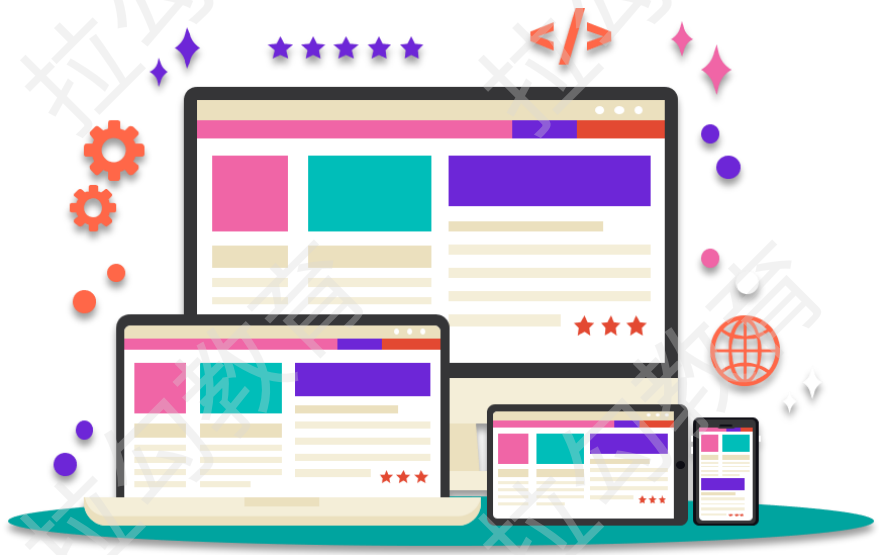
本课时讲解了：

- Java 实现动态代理的两种方式，以及他们的区别
- 单例模式的三种创建方式，并看了一个 double check 的反例
- 享元模式和原型模式
- arthas 使用 trace 命令，寻找耗时代码块的方法，最终将问题定位到 Spring 的 AOP 功能模块里

小结

在设计模式中，对性能帮助最大的是**生产者消费者模式**

比如异步消息、reactor 模型等



Next：12 | 《案例分析：并行计算让代码“飞”起来》

拉勾教育

—互联网人实战大学—

关注拉勾「教育公众号」
获取更多课程信息

L / A / G / O / U