代理模式

代理模式的主要作用是扩展目标对象的功能，比如说在目标对象的某个方法执行前后你可以增加一些自定义的操作。

定义一个接口及其实现类；

创建一个代理类同样实现这个接口

将目标对象注入进代理类，然后在代理类的对应方法调用目标类中的对应方法。这样的话，我们就可以通过代理类屏蔽对目标对象的访问，并且可以在目标方法执行前后做一些自己想做的事情。

动态代理** 和 **Spring AOP**（面向切面编程） 在许多方面有相似之处，尤其是在代理模式的使用上。Spring AOP 实际上是基于动态代理的实现，通过这种方式可以在运行时动态地给目标对象添加一些额外的行为，比如日志记录、性能监控、事务处理等。

动态代理和**Spring AOP**的相似性

1. 目标对象和代理对象：
    - 动态代理：在动态代理中，目标对象的方法调用是通过代理对象来拦截的，代理对象在方法调用前后做一些操作（比如日志记录）。
    - **Spring AOP**：Spring AOP 同样通过代理对象来实现功能增强，调用目标对象的原方法时，实际上是调用了代理对象的方法，Spring AOP 通过切面（Aspect）来拦截方法调用并执行增强逻辑。
2. 增强功能：
    - 动态代理：动态代理可以通过 `InvocationHandler` 来为方法调用添加额外的功能，比如前置处理、后置处理、异常处理等。
    - **Spring AOP**：Spring AOP 通过切面中的 增强（**Advice**） 来定义方法执行前后的操作（例如：`@Before`, `@After`, `@Around` 注解）。
3. 运行时创建代理：
    - 动态代理：使用 `Proxy.newProxyInstance()` 创建代理对象，在运行时动态生成代理类。
    - **Spring AOP**：Spring AOP 也是在运行时根据配置生成代理对象，通常使用 JDK 动态代理或 CGLib 代理，取决于目标对象是否实现接口。

3. 总结：**Spring AOP** 的核心功能

- **@Before**：在目标方法执行之前执行。
- **@After**：在目标方法执行之后执行（无论是否异常）。
- **@AfterReturning**：在目标方法成功返回之后执行。
- **@AfterThrowing**：在目标方法抛出异常之后执行。
- **@Around**：在目标方法执行前后执行，能够控制方法是否执行。

4. **AOP** 切点表达式

Spring AOP 支持灵活的切点表达式，常见的切点表达式包括：

- `execution(* com.example.service.MyService.*(..))`：匹配 `com.example.service.MyService` 类中的所有方法。
- `execution(* com.example.service.*.perform*(..))`：匹配 `com.example.service` 包中所有方法名以 `perform` 开头的方法。
- `within(com.example.service.MyService)`：匹配 `com.example.service.MyService` 类中的所有方法。
- `@annotation(org.springframework.web.bind.annotation.RequestMapping)`：匹配所有标有 `@RequestMapping` 注解的方法。

静态代理

静态代理中，我们对目标对象的每个方法的增强都是手动完成的（*后面会具体演示代码*），非常不灵活（*比如接口一旦新增加方法，目标对象和代理对象都要进行修改*）且麻烦(*需要对每个目标类都单独写一个代理类*）。 实际应用场景非常非常少，日常开发几乎看不到使用静态代理的场景。

静态代理实现步骤:

1. 定义一个接口及其实现类；
2. 创建一个代理类同样实现这个接口
3. 将目标对象注入进代理类，然后在代理类的对应方法调用目标类中的对应方法。这样的话，我们就可以通过代理类屏蔽对目标对象的访问，并且可以在目标方法执行前后做一些自己想做的事情。

就是把功能少的类装进功能多的类　从而能调用多出来的功能

和适配器模式有点像　但是适配器模式只增加新的function　静态代理overide方法

动态代理

从 **JVM** 角度来说，动态代理是在运行时动态生成类字节码，并加载到 **JVM** 中的。

就 Java 来说，动态代理的实现方式有很多种，比如 **JDK** 动态代理、**CGLIB** 动态代理等等。

JDK 动态代理的基本工作原理

1. 代理对象的创建：JDK 动态代理在运行时生成代理类，它会动态创建一个实现了指定接口的代理类。
2. **InvocationHandler** 接口：创建代理类时，需要指定一个 `InvocationHandler` 对象，该对象负责拦截对代理对象方法的调用，并执行相关的操作（例如日志记录、权限检查等）。
3. **Proxy** 类：JDK 动态代理通过 `java.lang.reflect.Proxy` 类来生成代理对象，`Proxy` 类是动态代理的核心，它通过 `newProxyInstance()` 方法创建代理对象。

JDK 动态代理的主要步骤

1. 定义接口：动态代理只能为实现了接口的类创建代理对象，因此必须首先定义一个接口。
2. 实现 `InvocationHandler` 接口：`InvocationHandler` 是一个函数接口，它的 `invoke()` 方法会在代理对象的方法被调用时执行。
3. 使用 `Proxy.newProxyInstance()` 创建代理对象：`Proxy.newProxyInstance()` 会根据传入的接口列表和 `InvocationHandler` 创建代理对象。

```
// 定义一个接口
public interface HelloService {
    void sayHello(String name);
}

// 实现接口的类
```

```
public class HelloServiceImpl implements HelloService {
    @Override
    public void sayHello(String name) {
        System.out.println("Hello, " + name);
    }
}


// 定义一个 InvocationHandler
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;

public class HelloServiceInvocationHandler implements
InvocationHandler {
    private Object target;

    public HelloServiceInvocationHandler(Object target) {
        this.target = target;  //目标实例
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[]
args) throws Throwable {
        System.out.println("Before method " + method.getName());
        Object result = method.invoke(target, args);  // 调用目标对象
的方法
        System.out.println("After method " + method.getName());
        return result;
    }
}




// 使用 Proxy 创建代理对象
import java.lang.reflect.Proxy;

public class Main {
    public static void main(String[] args) {
        HelloService helloService = new HelloServiceImpl();  // 目标
对象
```

```
        HelloServiceInvocationHandler handler = new
HelloServiceInvocationHandler(helloService);

        // 创建代理对象
        HelloService proxy = (HelloService) Proxy.newProxyInstance(
            helloService.getClass().getClassLoader(),  // 目标对象的
类加载器
            helloService.getClass().getInterfaces(),   // 目标对象的
接口
            handler  // InvocationHandler
        );

        // 调用代理对象的方法
        proxy.sayHello("John");
    }
}
```

代理对象 `proxy` 会接收到方法调用，并将调用委托给 `InvocationHandler`（你实现的 `HelloServiceInvocationHandler` 类）。

`InvocationHandler` 的 `invoke()` 方法被调用，在 `invoke()` 方法中，添加了 `"Before method"` 的日志，然后调用了目标对象的方法 `realSubject.sayHello("John")`，并执行实际的业务逻辑。

执行完目标对象的方法后，再返回 `"After method"` 的日志，然后返回实际的结果。

代理对象会根据 被代理类（或实现的接口）的方法来进行相应的处理。具体来说，代理对象并不会为每个方法单独定义一个代理，而是通过 `InvocationHandler` 的 `invoke()` 方法来捕获所有被代理对象的方法调用，动态地决定对哪个方法进行处理。

CGLIB 动态代理

Codlib 动态代理 是一种基于 **Java** 反射机制 的动态代理方式，通常用于简化代理类的生成，提供一种更简洁、更高效的方式来实现接口的代理。与 JDK 原生的动态代理类似，Codlib 动态代理利用 代理模式 和 反射机制 来处理方法调用，但是它的实现方式和使用上可能会更加灵活和高效。

代理对象的生成：JDK 动态代理只支持 接口代理，而 Codlib 动态代理不仅支持接口代理，还支持 类代理。

Codlib 动态代理的工作流程与 Java 原生的动态代理（基于接口的代理）类似，但它使用了一些优化，使得它更加高效和易用。其基本流程可以概括为：

- 接口或类的代理：Codlib 生成一个代理类，该类实现了目标接口（或继承了目标类），并通过 反射 在运行时处理所有方法调用。
- 方法拦截：Codlib 在代理类中动态生成方法实现，通过代理类的 调用链 处理目标方法的调用。
- 代理类的创建：Codlib 动态生成的代理类会继承原有类或实现原有接口，并且在每个方法调用时会执行特定的逻辑，如日志记录、事务管理、权限控制等。

```
public interface UserService {   //被代理的类
    void addUser(String name);
    void deleteUser(String name);
}

public class UserServiceImpl implements UserService {
    @Override
    public void addUser(String name) {
        System.out.println("User " + name + " added.");
    }

    @Override
    public void deleteUser(String name) {
        System.out.println("User " + name + " deleted.");
    }
}
==================

import com.codlib.proxy.ProxyFactory;

public class CodlibProxyExample {
    public static void main(String[] args) {
        // 目标对象
        UserService userService = new UserServiceImpl();

        // 创建代理
        UserService proxy = (UserService)
ProxyFactory.createProxy(userService, (proxyObj, method, args1) ->
{
            // 这里是拦截方法的逻辑
            System.out.println("Before method " +
method.getName());
            Object result = method.invoke(userService, args1);  //
调用目标方法
```

```
            System.out.println("After method " + method.getName());
            return result;
        });

        // 代理方法调用
        proxy.addUser("John");
        proxy.deleteUser("Alice");
    }
}
```

Codlib 的确简化了很多实现过程，不需要手动定义专门的 Handler 类。它通过 ProxyFactory.createProxy 方法提供了一个简便的方式来创建动态代理。你只需要提供一个目标对象和一个 MethodInterceptor（拦截器），Codlib 会自动为你创建代理类并执行方法拦截逻辑

缺点：

依赖第三方库，增加了外部依赖。

功能和灵活性不如 Spring AOP 等框架，缺乏一些高级特性。

反射

反射的主要用途：

- 动态加载类并创建实例
- 获取类的结构信息（如类的方法、字段等）
- 动态调用方法
- 修改字段的值
- 生成代理对象

优点：可以让咱们的代码更加灵活、为各种框架提供开箱即用的功能提供了便利

缺点：让我们在运行时有了分析操作类的能力，这同样也增加了安全问题。比如可以无视泛型参数的安全检查（泛型参数的安全检查发生在编译时）。另外，反射的性能也要稍差点，不过，对于框架来说实际是影响不大的。

获取**class**

```
Class<?> clazz = MyClass.class;

MyClass obj = new MyClass();
Class<?> clazz = obj.getClass();

Class<?> clazz = Class.forName("com.example.MyClass");
```

获取构造方法 字段 方法

```
Constructor<?> constructor = clazz.getConstructor(String.class,
int.class);

Field field = clazz.getDeclaredField("myField");

Method method = clazz.getMethod("myMethod", String.class);
```

创造对象实例

```
Constructor<?> constructor = clazz.getConstructor();
Object obj = constructor.newInstance();
```

动态调用方法

```
Method method = clazz.getMethod("setMyField", String.class);
method.invoke(obj, "Hello Reflection!");
//obj是实例
```

获取和修改字段值

```
Field field = clazz.getDeclaredField("myField");
field.setAccessible(true);  // 允许访问私有字段
field.set(obj, "New Value");
```

补充

```
        //获得类的属性
          System.out.println("===============");
         Field[] fields = c1.getFields(); //只能找到public属性  但包括本类
         for (Field f : fields) {
             System.out.println(f);
         }


          Field[] declaredFields = c1.getDeclaredFields();
          //能找到全部的属性（不包括父类）
          for (Field f : declaredFields) {
              System.out.println(f);
          }

getdeclare可以获得所有属性包括private

//获得类的方法
          Method[] methods = c1.getMethods();//获得本类及其父类的全部public方法  包括object类
          Method[] declaredMethods = c1.getDeclaredMethods();//获得本类的所有方法  不包括父类的方法
```

自定义注解 感觉没啥大用

*@Retention**：定义注解的生命周期。

- `RetentionPolicy.SOURCE`：注解仅在源代码中存在，编译时会被丢弃。
- `RetentionPolicy.CLASS`：注解在编译后会保留在字节码中，但不会被加载到 JVM 中。
- `RetentionPolicy.RUNTIME`：注解会被保留在字节码中，并且可以通过反射在运行时访问。

@Target：指定注解的使用范围（如类、方法、字段等）。

- `ElementType.TYPE`：类、接口、枚举声明。
- `ElementType.FIELD`：字段。
- `ElementType.METHOD`：方法。
- `ElementType.PARAMETER`：方法参数。
- `ElementType.CONSTRUCTOR`：构造函数。

@Inherited：指示该注解是否可继承。

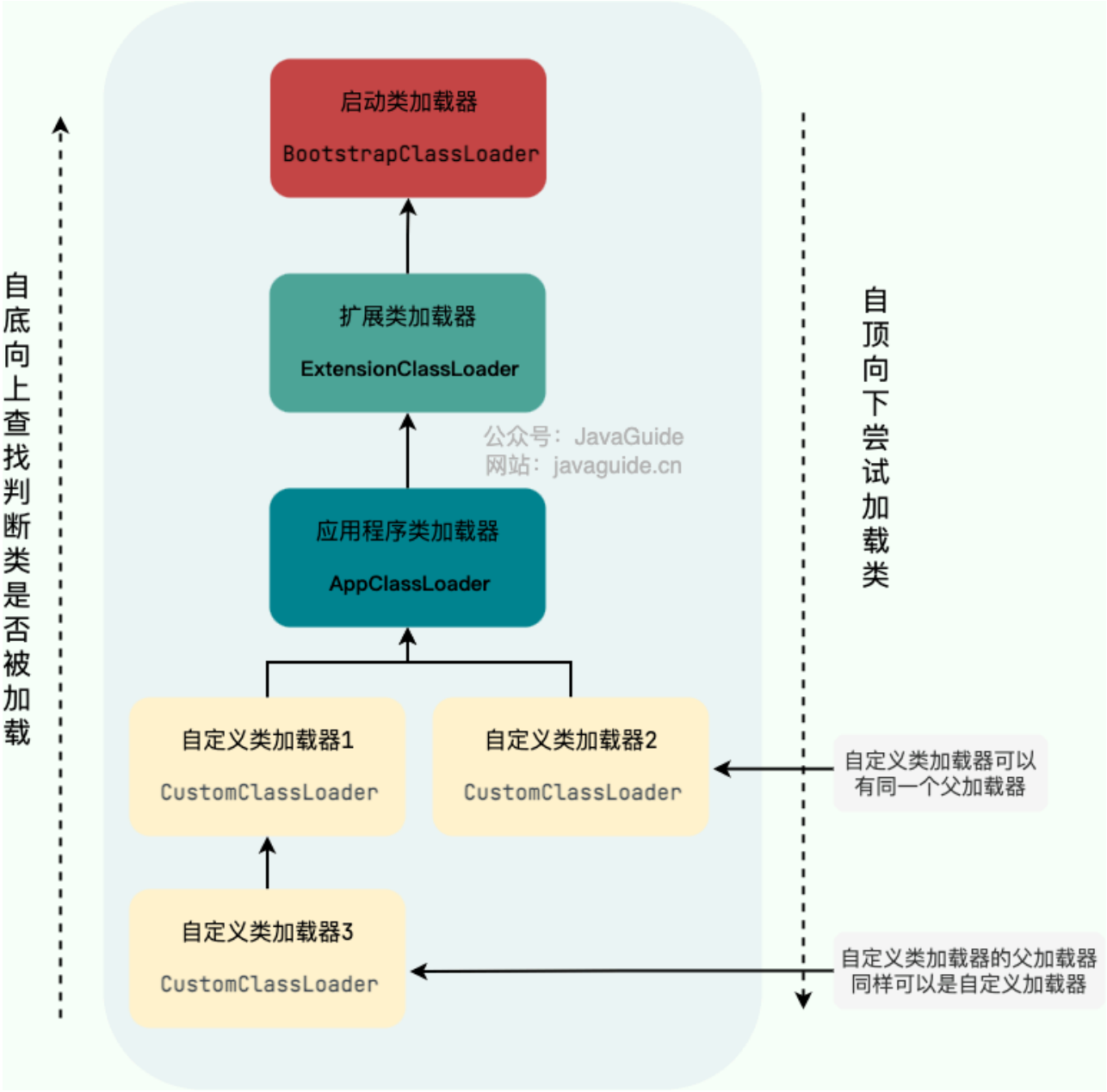
- 如果某个注解使用了 `@Inherited`，则这个注解会被子类继承。

双亲委派制和类加载器

- 类加载过程：加载->连接->初始化。
- 连接过程又可分为三步：验证->准备->解析。

JVM 中内置了三个重要的 `ClassLoader`：

1. `BootstrapClassLoader`(启动类加载器)：最顶层的加载类，由 C++实现，通常表示为 null，并且没有父级，主要用来加载 JDK 内部的核心类库（`%JAVA_HOME%/lib`目录下的 `rt.jar`、`resources.jar`、`charsets.jar`等 jar 包和类）以及被 `-Xbootclasspath`参数指定的路径下的所有类。
2. `ExtensionClassLoader`(扩展类加载器)：主要负责加载 `%JRE_HOME%/lib/ext` 目录下的 jar 包和类以及被 `java.ext.dirs` 系统变量所指定的路径下的所有类。
3. `AppClassLoader`(应用程序类加载器)：面向我们用户的加载器，负责加载当前应用 classpath 下的所有 jar 包和类。



每个 `ClassLoader` 可以通过`getParent()`获取其父 `ClassLoader`，如果获取到 `ClassLoader` 为`null`的话，那么该类是通过 `BootstrapClassLoader` 加载的。

为什么 获取到 `ClassLoader` 为`null`就是 `BootstrapClassLoader` 加载的呢？ 这是因为`BootstrapClassLoader` 由 C++ 实现，由于这个 C++ 实现的类加载器在 Java 中是没有与之对应的类的，所以拿到的结果是 null。

`lassLoader` 类使用委托模型来搜索类和资源。每个 `ClassLoader` 实例都有一个相关的父类加载器。需要查找类或资源时，`ClassLoader` 实例会在试图亲自查找类或资源之前，将搜索类或资源的任务委托给其父类加载器。
虚拟机中被称为 "bootstrap class loader"的内置类加载器本身没有父类加载器，但是可以作

为 `ClassLoader` 实例的父类加载器。

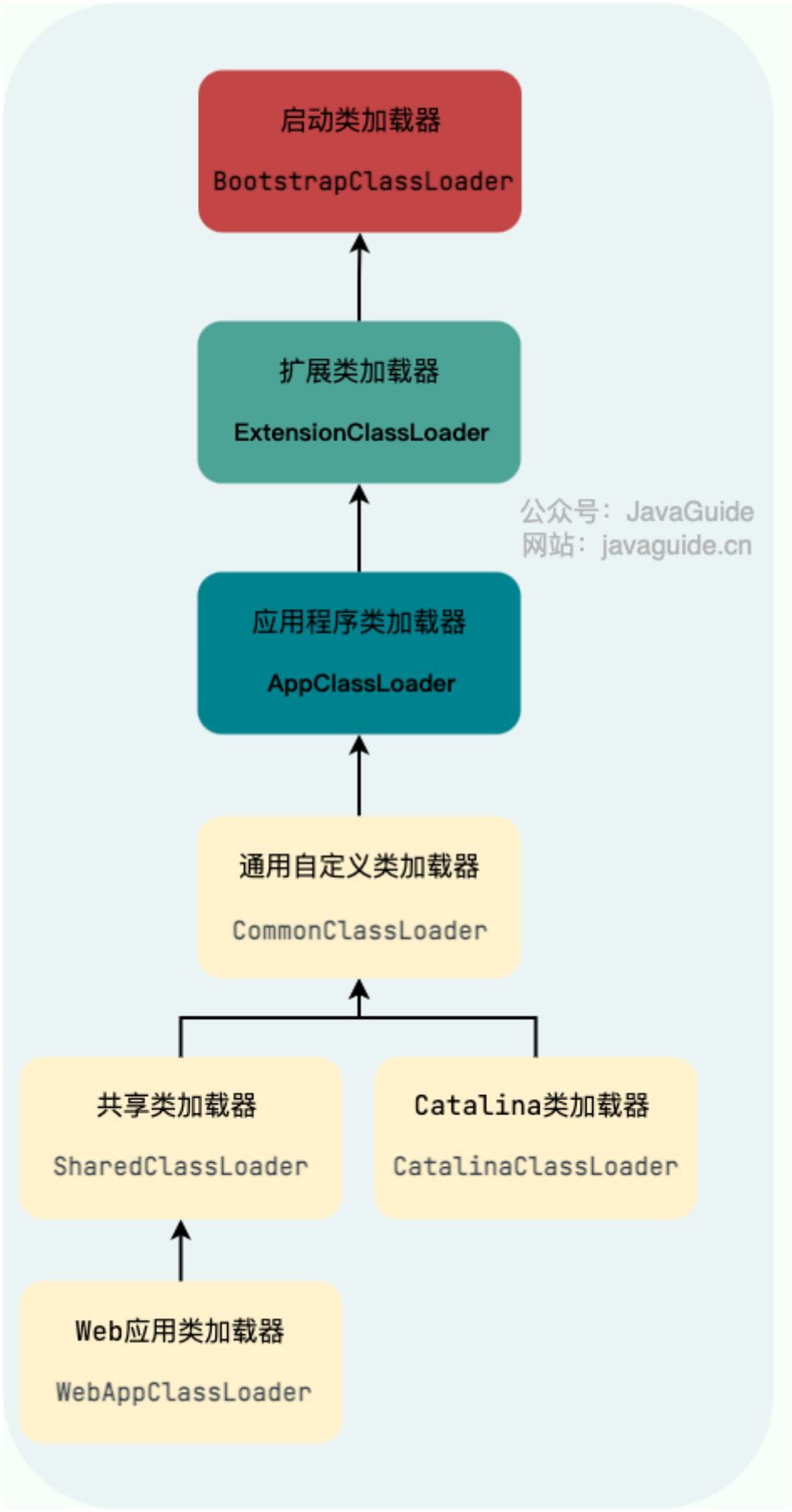
结合上面的源码，简单总结一下双亲委派模型的执行流程：

- 在类加载的时候，系统会首先判断当前类是否被加载过。已经被加载的类会直接返回，否则才会尝试加载（每个父类加载器都会走一遍这个流程）。
- 类加载器在进行类加载的时候，它首先不会自己去尝试加载这个类，而是把这个请求委派给父类加载器去完成（调用父加载器 `loadClass()` 方法来加载类）。这样的话，所有的请求最终都会传送到顶层的启动类加载器 `BootstrapClassLoader` 中。
- 只有当父加载器反馈自己无法完成这个加载请求（它的搜索范围中没有找到所需的类）时，子加载器才会尝试自己去加载（调用自己的 `findClass()` 方法来加载类）。
- 如果子类加载器也无法加载这个类，那么它会抛出一个 `ClassNotFoundException` 异常。

JVM 判定两个 **Java** 类是否相同的具体规则：JVM 不仅要看类的全名是否相同，还要看加载此类的类加载器是否一样。只有两者都相同的情况，才认为两个类是相同的。即使两个类来源于同一个 `Class` 文件，被同一个虚拟机加载，只要加载它们的类加载器不同，那这两个类就必定不相同。

双亲委派模型保证了 Java 程序的稳定运行，可以避免类的重复加载（JVM 区分不同类的方式不仅仅根据类名，相同的类文件被不同的类加载器加载产生的是两个不同的类），也保证了 Java 的核心 API 不被篡改。

打破双亲 **tomcat**服务器




```
ommonClassLoader`对应`<Tomcat>/common/*
CatalinaClassLoader`对应`<Tomcat >/server/*
SharedClassLoader`对应 `<Tomcat >/shared/*
WebAppClassloader`对应 `<Tomcat >/webapps/<app>/WEB-INF/*
```

`CatalinaClassLoader` 和 `SharedClassLoader` 能加载的类则与对方相互隔离。`CatalinaClassLoader` 用于加载 Tomcat 自身的类，为了隔离 Tomcat 本身的类和 Web 应用的类。`SharedClassLoader` 作为 `WebAppClassLoader` 的父加载器，专门来加载 Web 应用之间共享的类比如 Spring、Mybatis

每个 Web 应用都会创建一个单独的 `WebAppClassLoader`，并在启动 Web 应用的线程里设置线程线程上下文类加载器为 `WebAppClassLoader`。各个 `WebAppClassLoader` 实例之间相互隔离，进而实现 Web 应用之间的类隔

例子

在一个 Web 应用中，Spring 的核心类库（如 Spring 的 jar 包）通常会由 **SharedClassLoader**（共享类加载器）加载。

但 **WebAppClassLoader** 加载的类主要是属于 Web 应用自己的类，而不是共享类库。虽然从层次结构上看，**WebAppClassLoader** 可以继承父类的加载能力，但为了保证 **Web** 应用之间的隔离性，Tomcat 并不会让 Web 应用类直接使用父类（即 **SharedClassLoader**）来加载 Web 应用自己的业务类。这样做的目的是 确保每个 **Web** 应用之间互相隔离，避免相互影响。

然而，在 Web 应用中，可能会有自定义的业务类，这些类用到了 Spring 的接口或注解。由于这些业务类存放在 Web 应用的目录下，**SharedClassLoader** 无法直接加载它们，因为它的加载路径不包括 Web 应用的路径。

这就造成了 类加载器隔离 和 类加载器冲突 问题，即 **SharedClassLoader** 无法加载 Web 应用中的类。

如何解决这个问题呢？ 这个时候就需要用到 线程上下文类加载器（`ThreadContextClassLoader`） 了。

通过线程上下文类加载器（TCCL），Spring 退而求其次，选择使用 **WebAppClassLoader**（子类加载器）来加载 Web 应用中的业务类，而不是使用自己的 **SharedClassLoader**（父类加载器）。

线程上下文类加载器的原理是将一个类加载器保存在线程私有数据里，跟线程绑定，然后在需要的时候取出来使用。这个类加载器通常是由应用程序或者容器（如 Tomcat）设置的。

线程上下文类加载器 就是为每个线程指定一个 类加载器，并且可以在运行时修改它。通过这种方式，不同的线程可以使用不同的类加载器来加载类，避免了类加载器的冲突和依赖问题。