

第三章 微服务架构设计原理

微服务概念

概念

把一个大型的单体应用程序和服务拆分为数个甚至数十个的支持微服务。它可以扩展单个组件，而不是整个应用程序堆栈，从而满足服务等级协议。

定义

围绕业务领域组件来创建应用，这些应用可独立的进行开发、管理和迭代。在分散的组件中使用云架构和平台式部署、管理和服务功能，使产品交付变得更加简单。

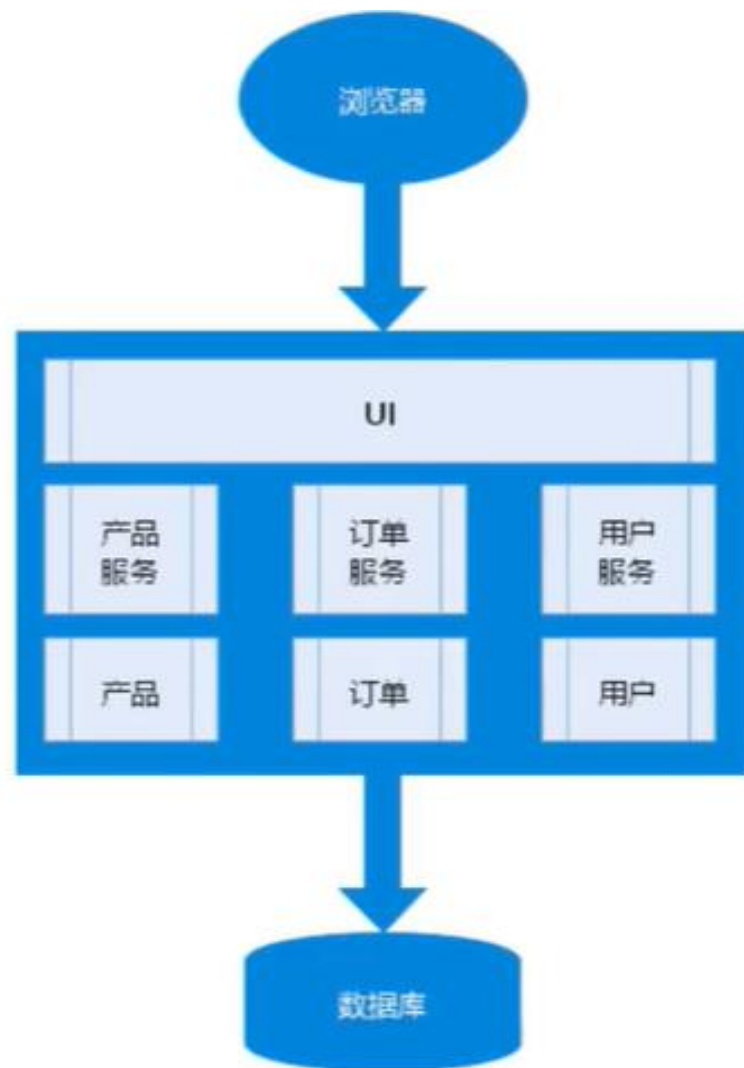
与传统架构的区别

- 提高敏捷性：及时响应业务需求，促进企业发展
- 提升用户体验：提升用户体验，减少用户流失
- 降低成本：降低增加产品、客户或业务方案的成本

与传统架构的区别

单体式开发

将所有功能打包在一个WAR包里面，基本没有外部依赖，（除了容器），部署在一个JavaEE容器（tomcat, JBoss, WebLogic）里，包含了 D O /DAO,SERVICE, U I等所有逻辑



优点

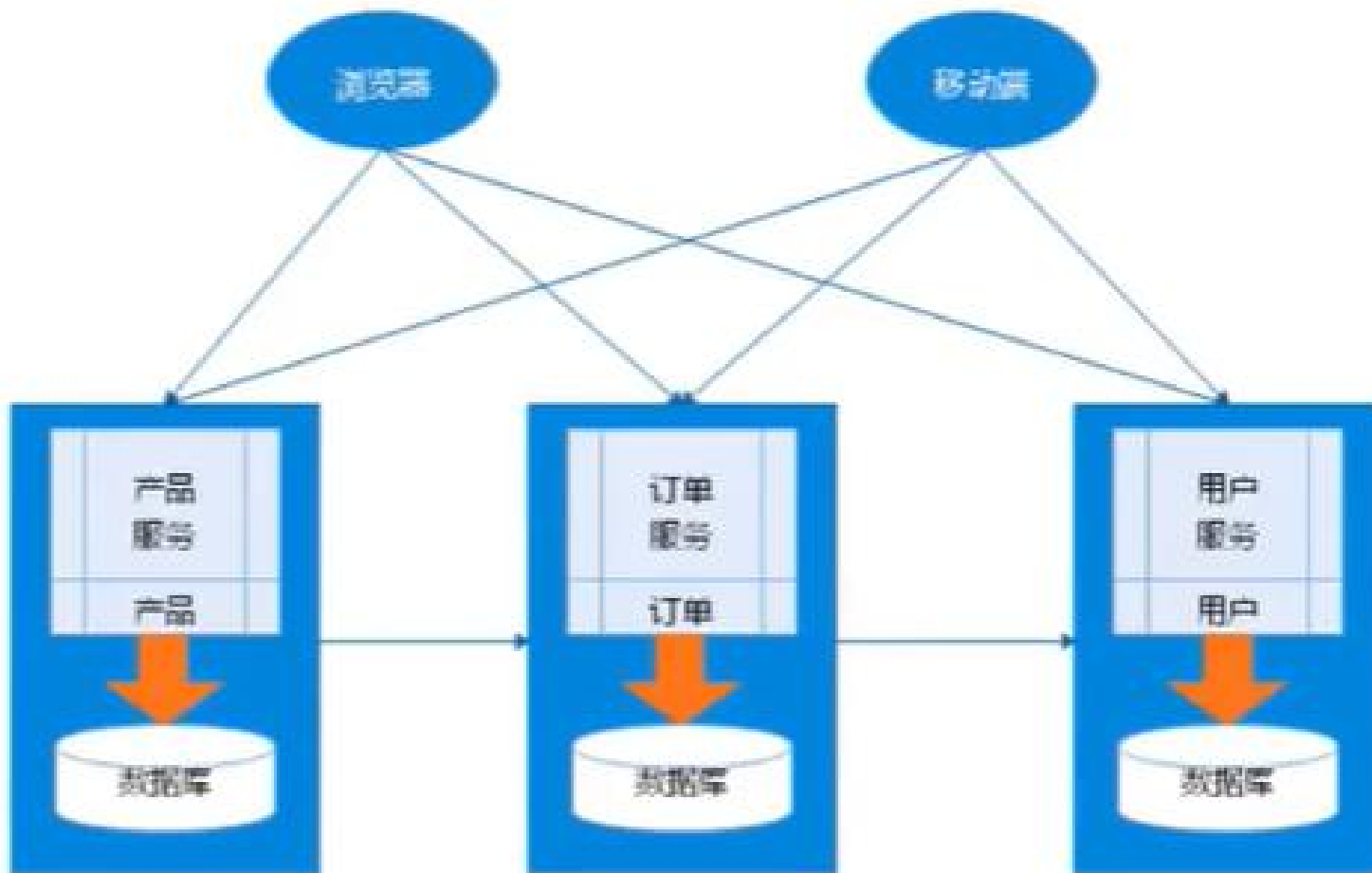
- 开发简单，集中式管理
- 基本不会重复开发
- 功能都在本地，没有分布式的管理和调用消耗

缺点

- 效率低:开发都在同一个项目改代码，相互等待，冲突不断
- 维护难:代码功能耦合在一起，新人不知道何从下手
- 不灵活:构建时间长，任何小修改都要重构整个项目，耗时
- 稳定性差:一个微小的问题，都可能导致整个应用挂掉
- 扩展性不够:无法满足高并发下的业务需求

微服务架构

微服务架构的目的：有效的拆分应用，实现敏捷开发和部署



微服务的特征

官方的定义

- 一系列的独立的服务共同组成系统
- 单独部署，跑在自己的进程中
- 每个服务为独立的业务开发
- 分布式管理
- 非常强调隔离性

大概的标准

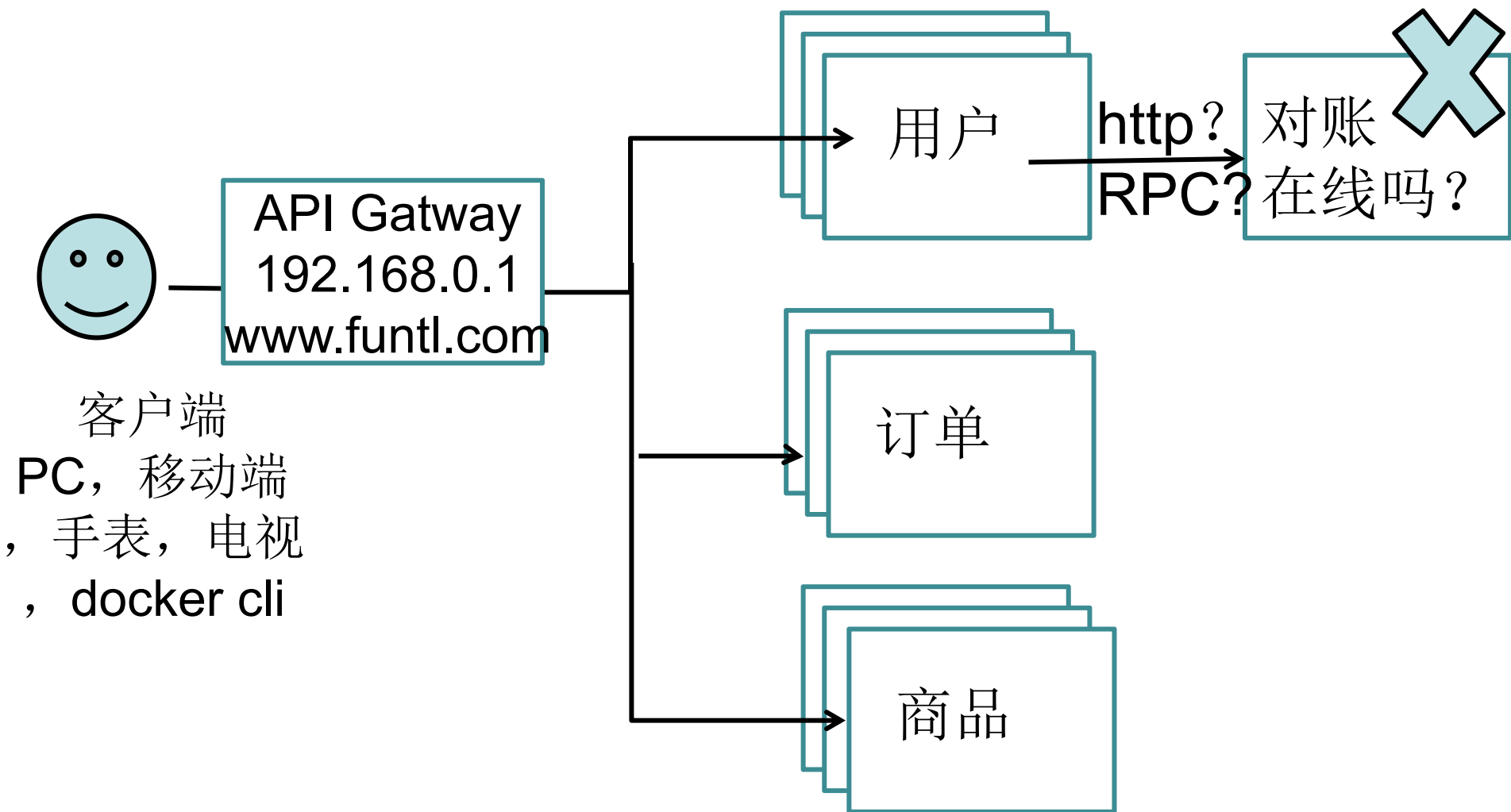
- 分布式服务组成的系统
- 按照业务，而不是技术来划分组织
- 做有生命的产品而不是项目
- 强服务个体和弱通信(**Smart endpoints and dumb pipes**)
- 自动化运维(**DevOps**)
- 高度容错性
- 快速演化和迭代

微服务的实践

微服务架构是一种架构，思想实际的开发方式采用的是分布式系统开发。架构就是为了解耦，而分布式系统开发一定会遇到四个问题。

- 1.这么多服务,客户端如何访问？
- 2.这么多，服务与服务之间如何通信？
- 3.这么多服务，服务如何治理？
- 4.这么多服务，服务挂了，怎么办？

微服务概念



1.这么多服务,客户端如何访问?

API 网关

2.这么多, 服务与服务之间如何通信?

两种

1.同步 对内**RPC**, 对外**REST**

1.1 **HTTP** --> **REST, JSON, 序列化, 反序列化**

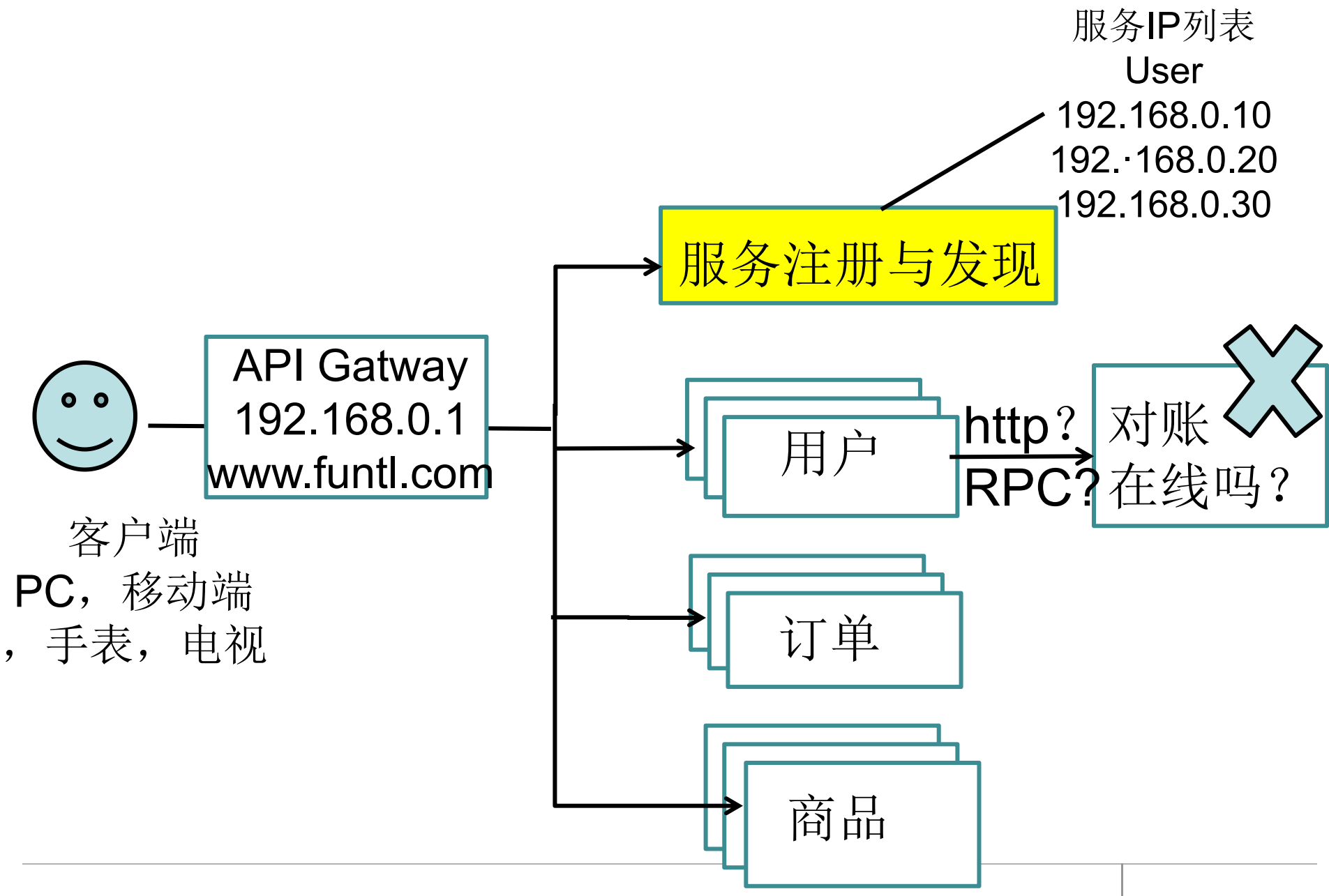
1.2 **RPC** -->传输效率高, 压缩, **10KB,1KB**, 序列化, 反序列化

2. 异步

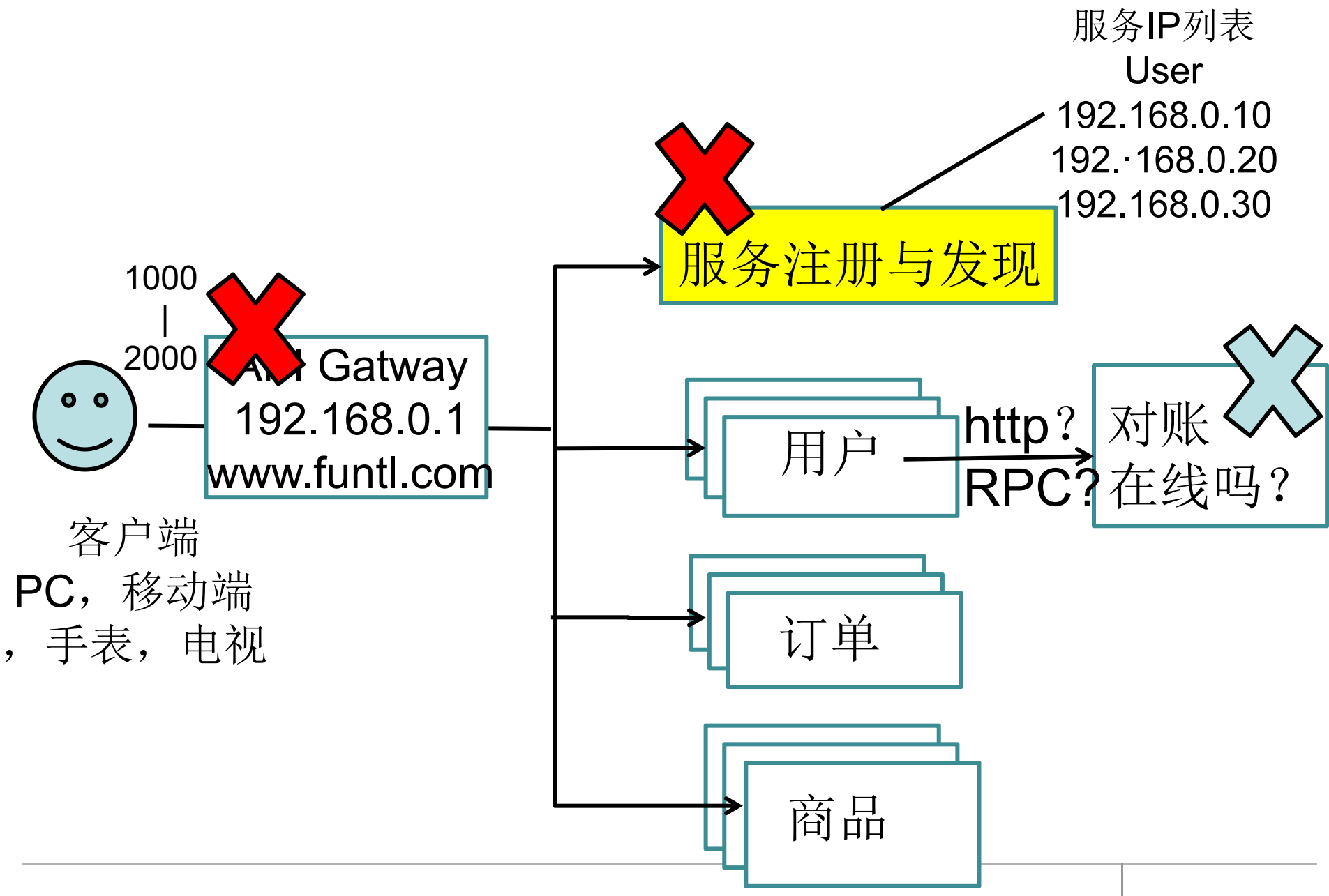
3.这么多服务, 服务如何治理?

4.这么多服务, 服务挂了, 怎么办?

微服务概念



微服务概念



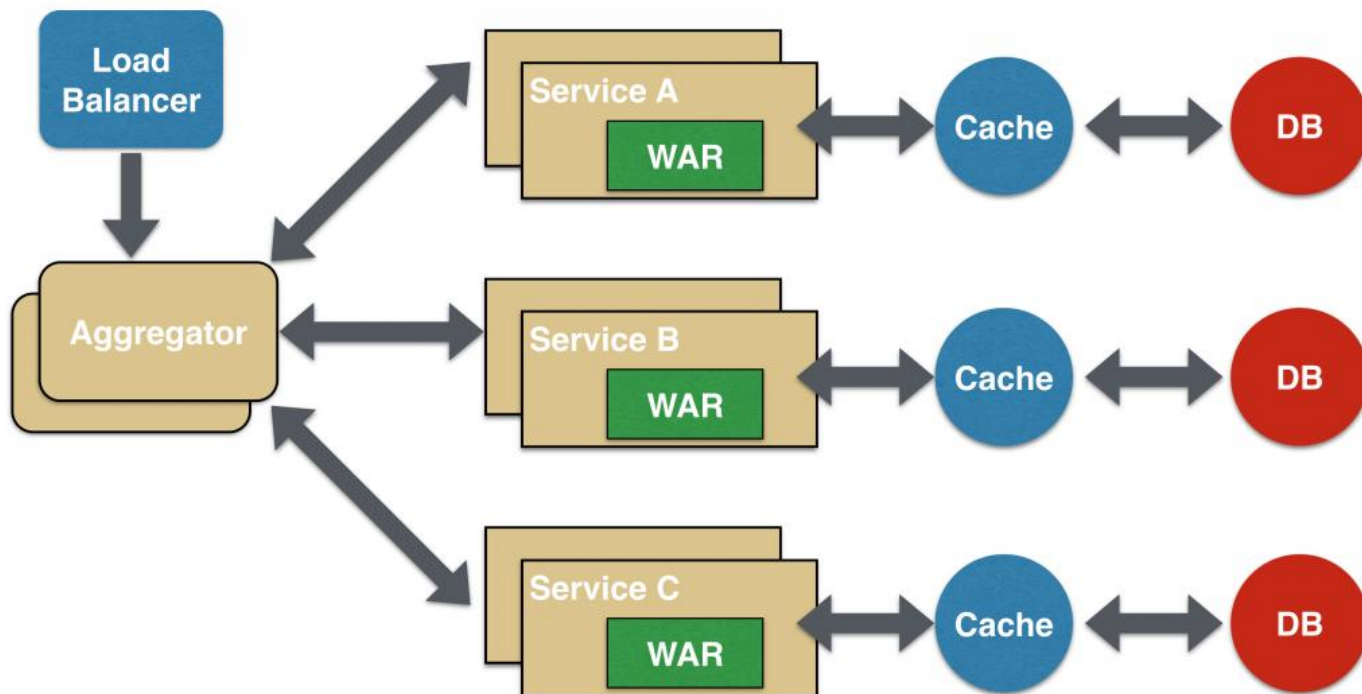
微服务架构设计风格

- 微服务设计模式分类
- 应用架构模式:单点登录, 注册发现, 熔断限流, 断路器, 网关模式, 消息补偿模式, 令牌模式
- 数据库:分库**Single Service**, 共库多**Service**, 多库同步, 事务性补偿
- 日志追踪模式:观察者模式**patterns**, **Log aggregation**, **Application metrics**, **Audit logging**, **Distributed tracking**, **Exception tracking**, **Heath check API**, **Log deployments and changes**, 分布式外路温热模式
- UI模式:**MVC**, **MVP**, **MVVM**, **Server-side page fragment composition**, **Client side UI composition**

- 微服务架构的六种设计风格的模式
 - 聚合器微服务设计模式
 - 代理微服务设计模式
 - 链式微服务设计模式
 - 分支微服务设计模式
 - 数据共享微服务设计模式
 - 异步消息传递微服务设计模式

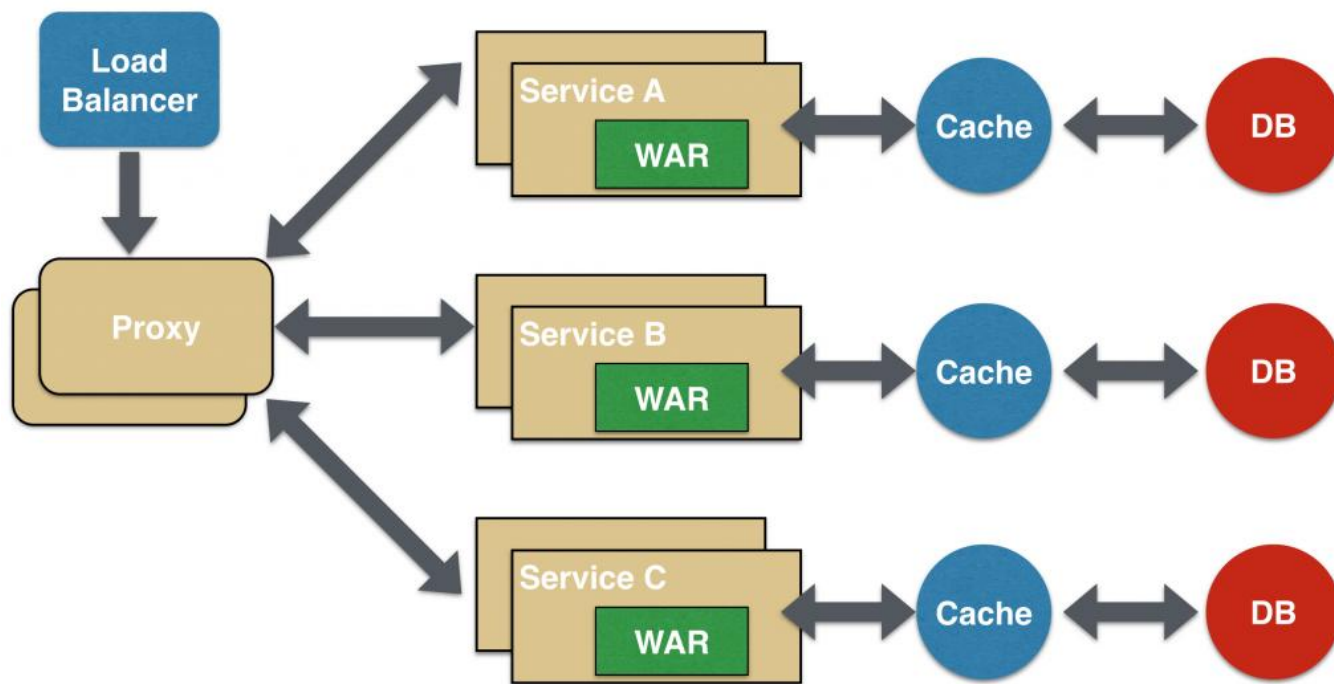
• 聚合器微服务设计模式

一种最常用也最简单的设计模式：聚合器调用多个服务实现应用程序所需的功能。



• 代理微服务设计模式

是聚合器模式的一个变种：根据业务需求的差别调用不同的微服务。



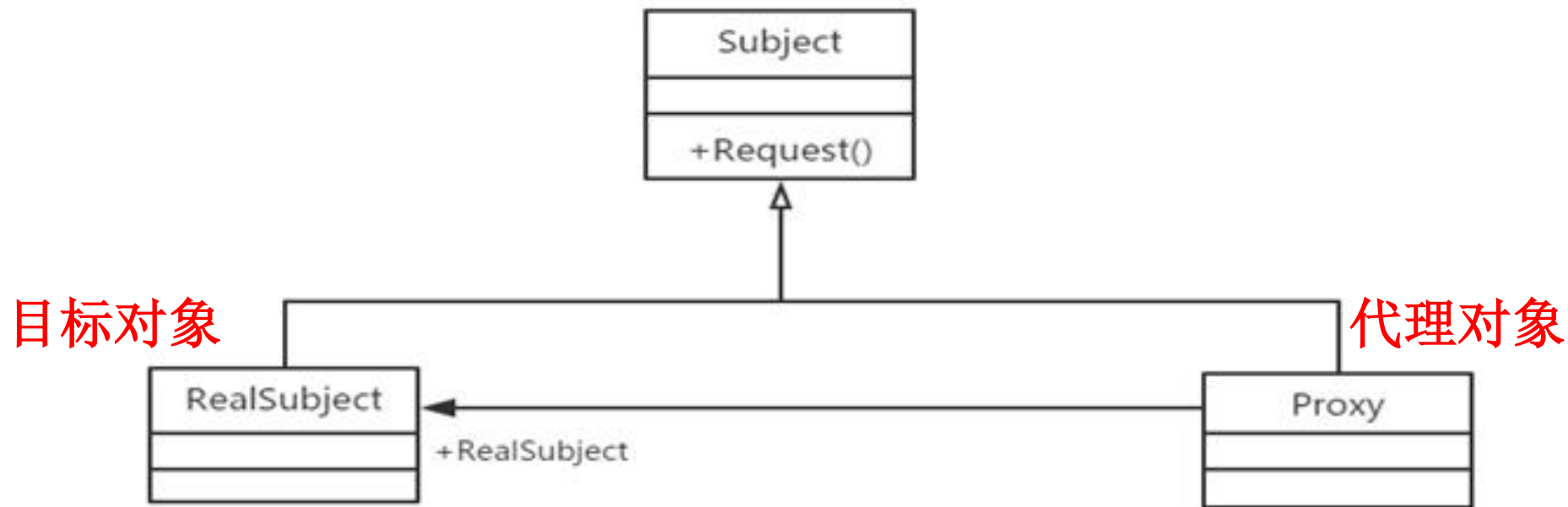
- **Java代理：静态代理和动态代理**

- **静态代理**

- 代理对象持有目标对象的句柄

- 所有调用目标对象的方法，都通过调用代理对象的方法

- 对每个方法，需要静态编码(理解简单，但代码繁琐)



- 静态代理示例
- **Subject**接口:

```
Subject.java ×  
1  
2 public interface Subject{  
3     public void request();  
4 }  
5
```

- 目标对象, **Subject**接口实现

```
// 目标对象  
  
class SubjectImpl implements Subject{  
  
    public void request() { System.out.println("I am dealing the request."); }  
}
```

- 静态代理对象

//代理对象

```
class StaticProxy implements Subject{
```

//实际目标对象

```
private Subject subject; 目标对象句柄
```

```
public StaticProxy(Subject subject) { this.subject = subject; }
```

```
public void request(){  
    System.out.println("PreProcess");  
    subject.request();  
    System.out.println("PostProcess");
```

```
}
```

```
}
```

- 主类

// 静态代理模式

```
public class StaticProxyDemo {  
    public static void main(String args[]){  
        // 创建实际对象  
        SubjectImpl subject = new SubjectImpl();  
  
        // 把实际对象封装到代理对象中  
        StaticProxy p = new StaticProxy(subject);  
        p.request();  
    }  
}
```

实际对象作为形参传递给代理对象，代理对象由此获取目标对象句柄

静态代理

–对每个方法，需要静态编码(简洁，但繁琐)

//代理对象

```
class StaticProxy implements Subject{
```

//实际对象

```
private Subject subject;
```

```
public StaticProxy(Subject subject){
```

```
    this.subject = subject;
```

```
}
```

```
public void request(){
```

```
    System.out.println("PreProcess");
```

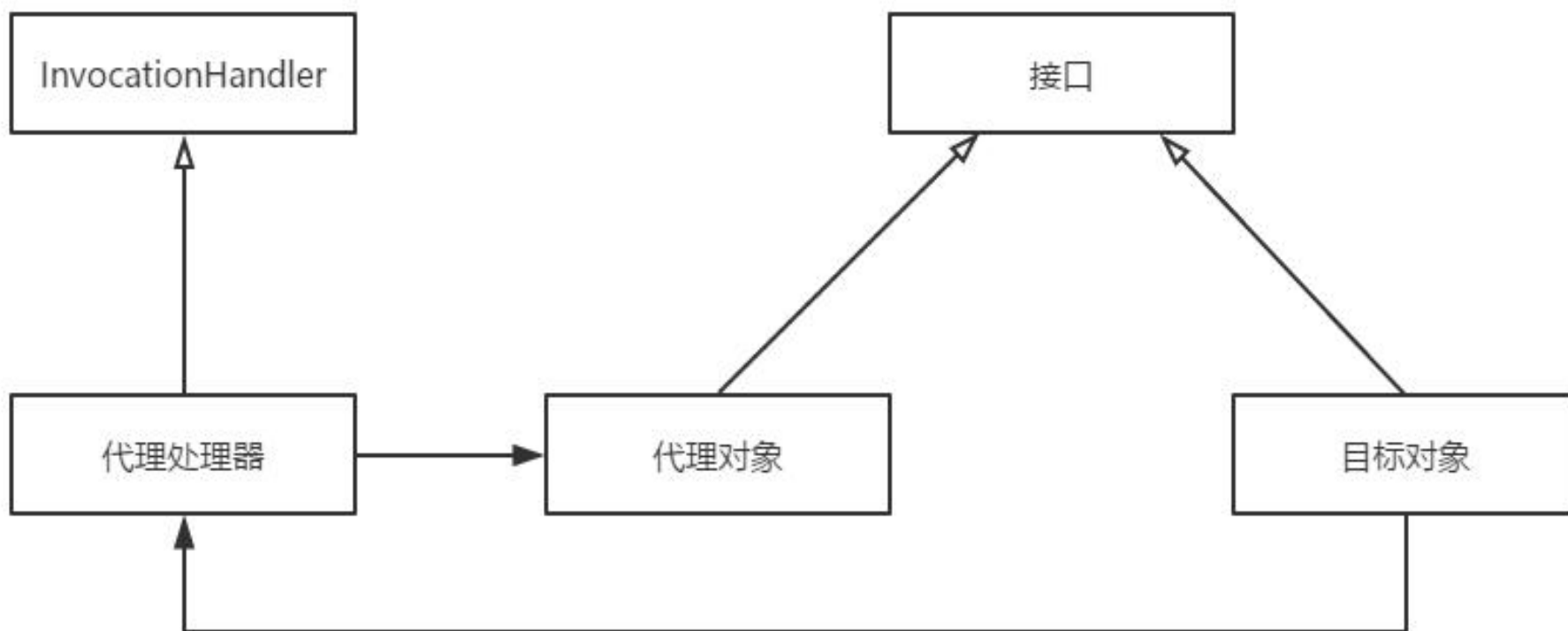
```
    subject.request();
```

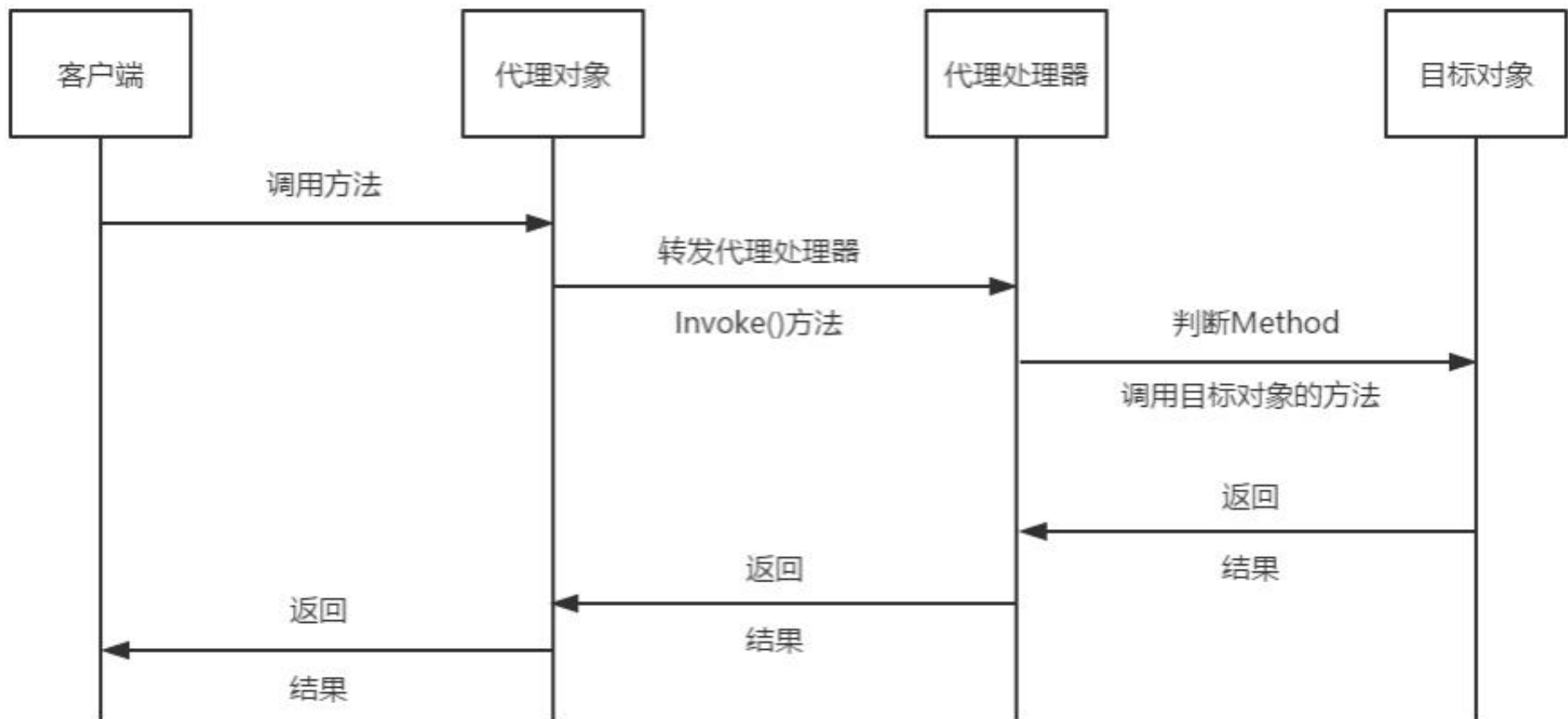
```
    System.out.println("PostProcess");
```

```
}
```

```
}
```


- 动态代理
- 一对目标对象的方法每次被调用，进行动态拦截





代理处理器

–持有目标对象的句柄

–实现**InvocationHandler**接口

- 实现**invoke**方法

- 所有的代理对象方法调用，都会转发到**invoke**方法

来

- **invoke**的形参**method**，就是指代理对象方法的调用

- 在**invoke**内部，可以根据**method**，使用目标对象不同的方法来响应请求

代理处理器

```

/**
 * 代理类的调用处理器
 */
class ProxyHandler implements InvocationHandler{
    private Subject subject;
    public ProxyHandler(Subject subject) { this.subject = subject; }

    // 此函数在代理对象调用任何一个方法时都会被调用。
    @Override
    public Object invoke(Object proxy, Method method, Object[] args)
        throws Throwable {
        System.out.println(proxy.getClass().getName());
        // 定义预处理的工作, 当然你也可以根据 method 的不同进行不同的预处理工作
        System.out.println("===before===");
        Object result = method.invoke(subject, args);
        System.out.println("===after===");
        return result;
    }
}
    
```

主类

```
import java.lang.reflect.Proxy;
//动态代理模式
public class DynamicProxyDemo {
    public static void main(String[] args) {
        //1. 创建目标对象
        SubjectImpl realSubject = new SubjectImpl();
        //2. 创建调用处理器对象
        ProxyHandler handler = new ProxyHandler(realSubject);
        //3. 动态生成代理对象
        Subject proxySubject =
            (Subject)Proxy.newProxyInstance
                (SubjectImpl.class.getClassLoader(),
                 SubjectImpl.class.getInterfaces(), handler);
        //proxySubject真实类型com.sun.proxy.$Proxy0
        //proxySubject继承Proxy类，实现Subject接口
        //newProxyInstance的第二个参数，就是指定代理对象的
        //4. 客户端通过代理对象调用方法
        //本次调用将自动被代理处理器的invoke方法接收
        proxySubject.request();
        System.out.println(proxySubject.getClass().getName());
    }
}
```

com.sun.proxy.\$Proxy0
 I
 ===before===
 I am dealing the request.
 ===after===
 com.sun.proxy.\$Proxy0

代理对象

- 根据给定的接口，由**Proxy**类自动生成的对象
- 类型 **com.sun.proxy.\$Proxy0**，继承自 **java.lang.reflect.Proxy**
- 通常和目标对象实现同样的接口(可另实现其他的接口)
- 实现多个接口
 - 接口的排序非常重要
 - 当多个接口里面有方法同名，则默认以第一个接口的方法调用
 - Therefore, when a duplicate method is invoked on a proxy instance, the Method object for the method in the foremost interface that contains the method (either directly or inherited through a superinterface) in the proxy class's list of interfaces is passed to the invocation handler's invoke method, regardless of the reference type through which the method invocation occurred

- 示例

网关是代理模式

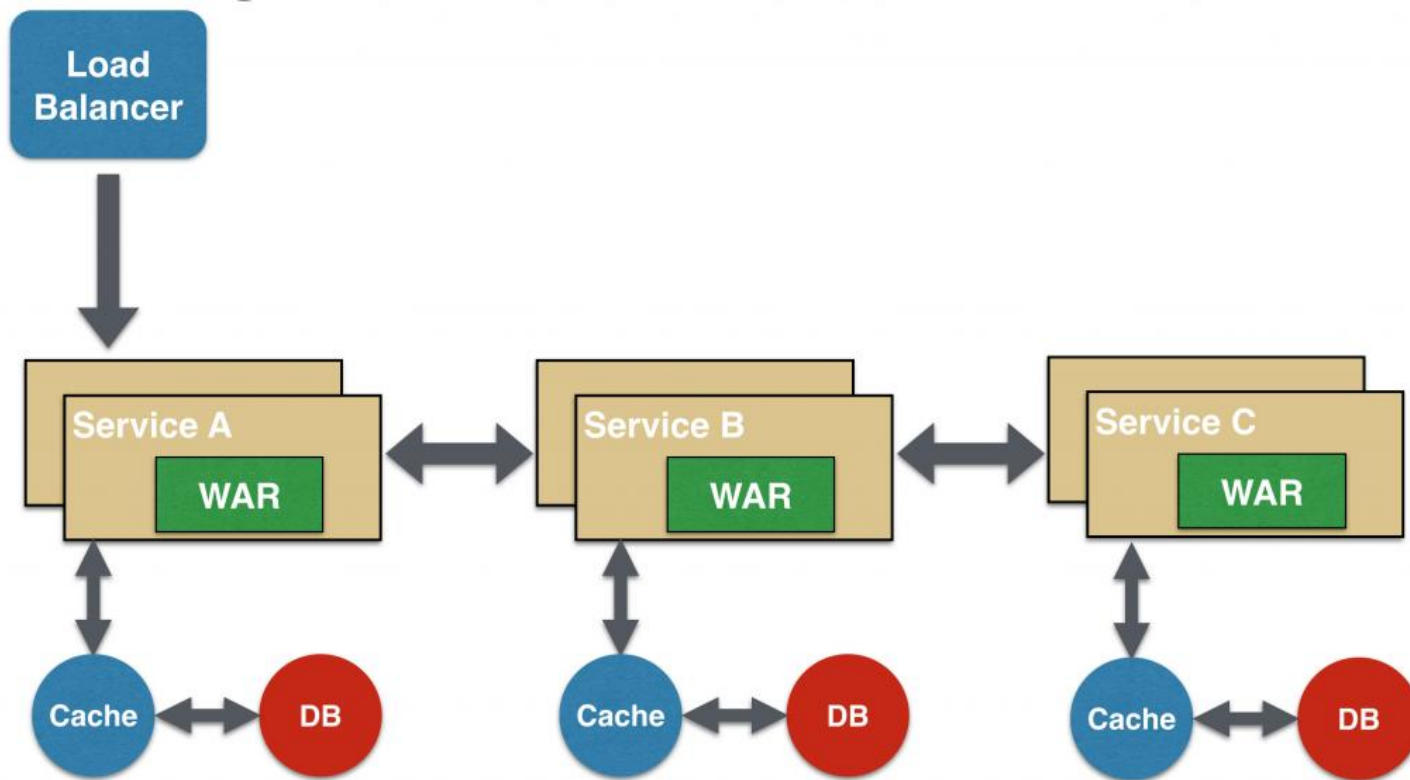
@EnableZuulProxy

启动类添加**@EnableZuulProxy**注解，开启网关功能。

配置动态路由规则

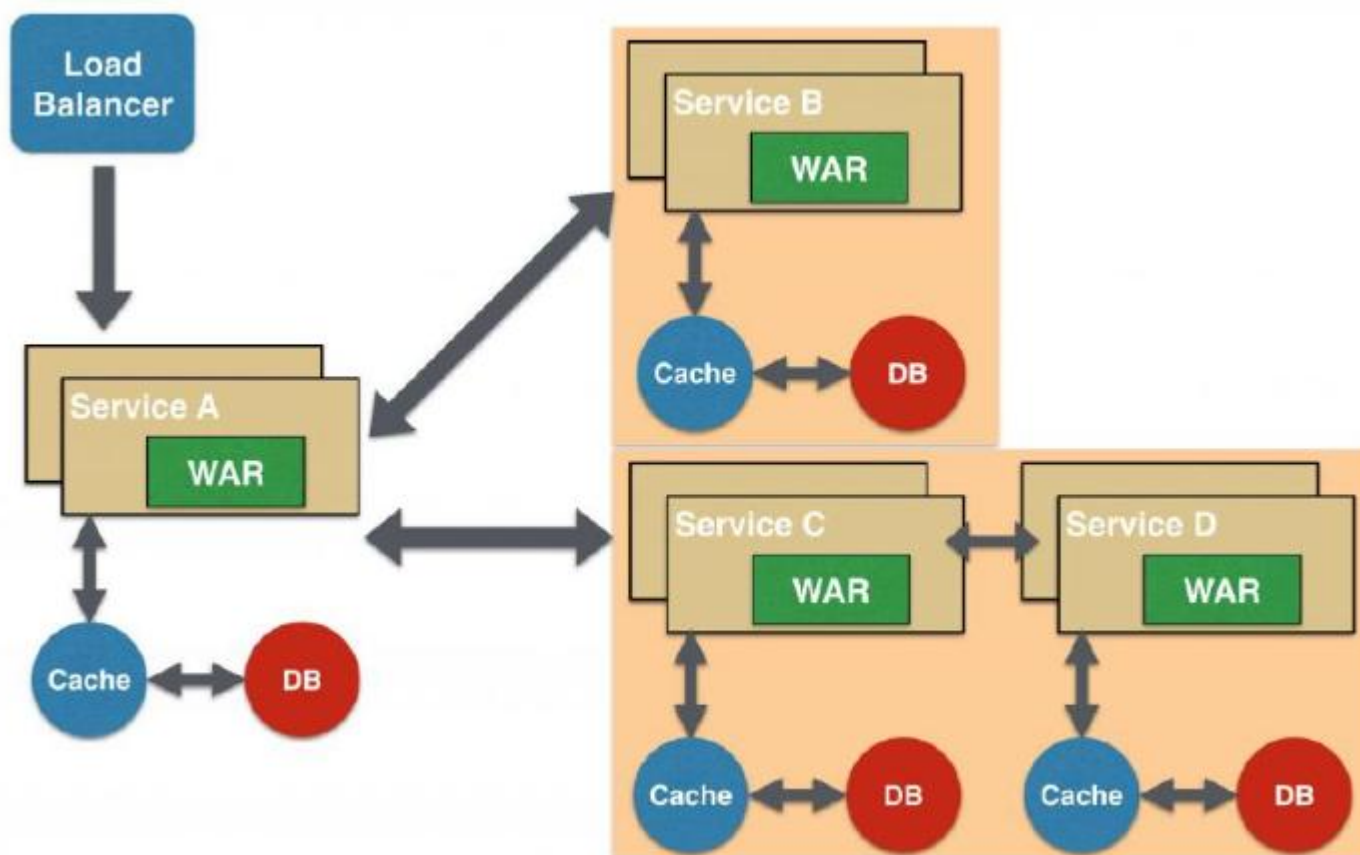
- 链式微服务设计模式

在接收到请求后会产生一个经过合并的响应。



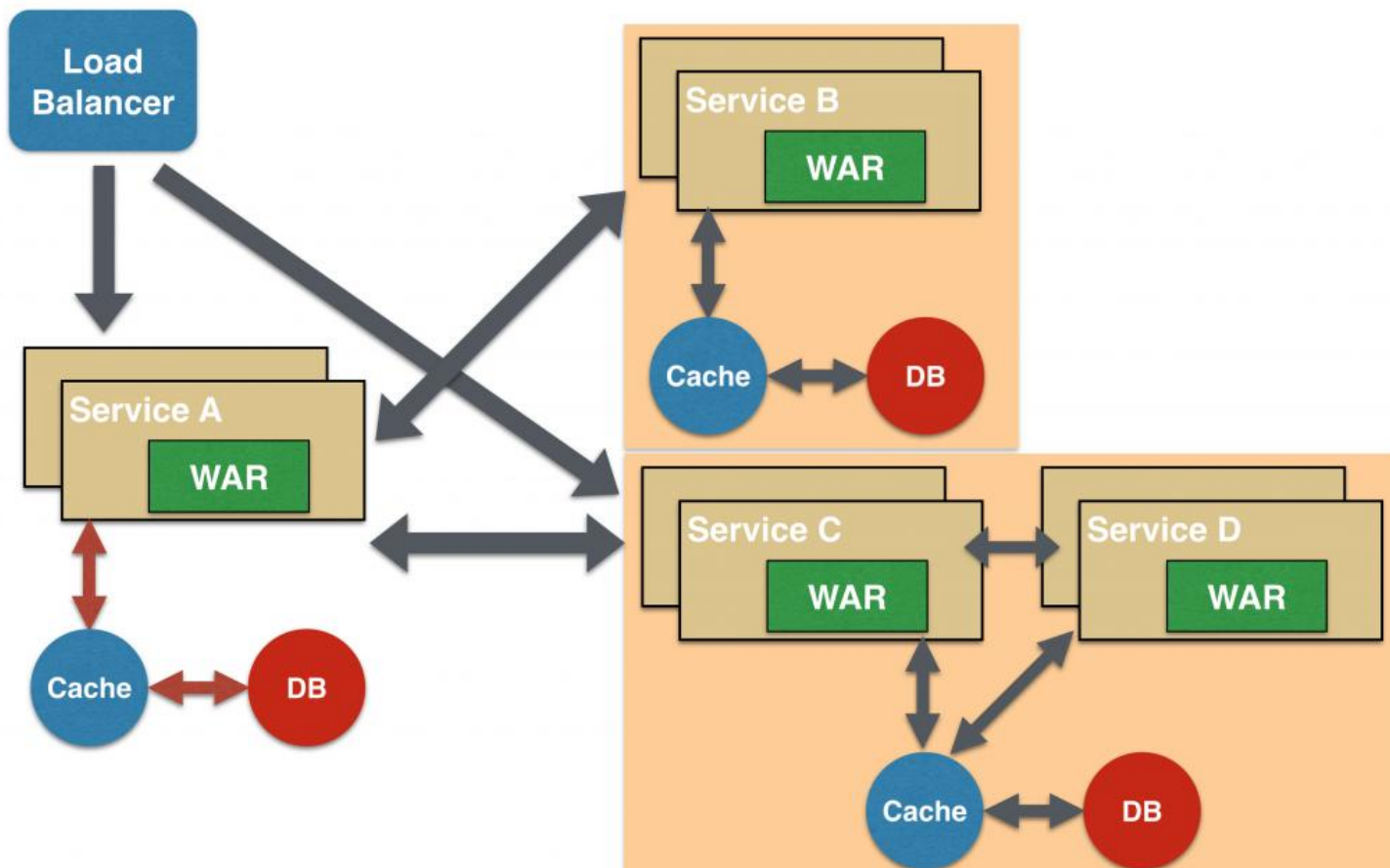
• 分支微服务设计模式

是聚合器模式的扩展，允许同时调用两个微服务链。



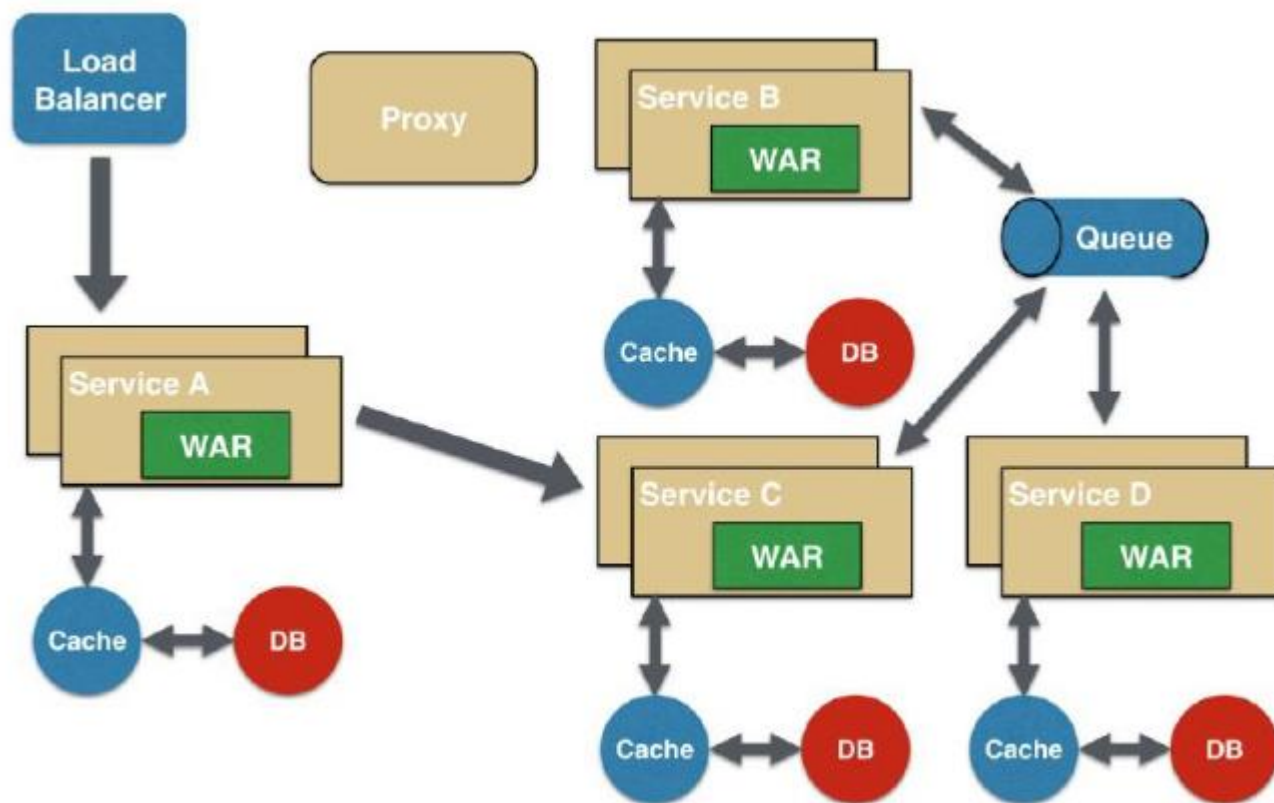
- 数据共享微服务设计模式

在单体应用到微服务架构的过渡阶段，可以使用这种设计模式



- 异步消息传递微服务设计模式

REST设计模式非常流行，但它是同步的，会造成阻塞。



微服务的本质：用一些功能比较明确，业务比较精炼的服务去解决更大更实际的问题。

DDD领域驱动设计

金融领域

业务领域

业务-->主业务逻辑

存款

取款

查询

领域划分

领域专家

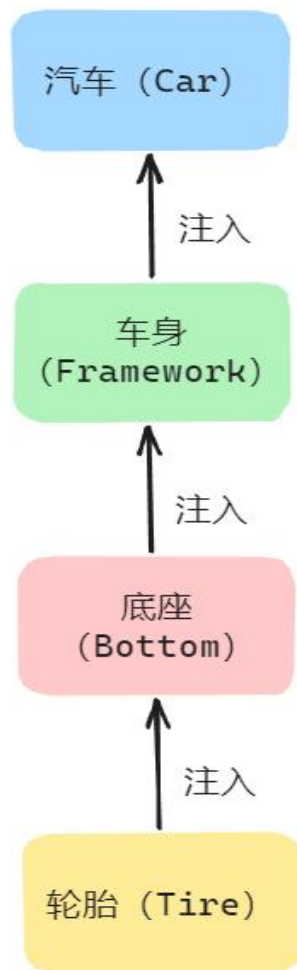
医疗领域

电商领域

Spring: AOP IOC

Spring框架核心思想：IOC、AOP。

IOC：控制翻转，运行时统一由Spring容器管理和串联



反射机制

问题：如何知道一个类有多少个方法？

```
public class Tiger {  
  
    public void move()  
    {  
        System.out.println("I can move fast");  
    }  
  
    public static void main(String[] args) {  
        Tiger t = new Tiger();  
        t.m  
    }  
}
```

- move() : void - Tiger
- main(String[] args) : void - Tiger

反射：reflection

Java中的反射机制是指在运行时动态地获取一个类的信息，包括类的方法、属性、构造函数等，而不需要事先知道这个类的具体实现。

- 程序可以访问、检测和修改它本身状态或行为的能力，即自描述和自控制。
- 可以在运行时加载、探知和使用编译期间完全未知的类。
- 给Java插上动态语言特性的翅膀，弥补强类型语言的不足。
- `java.lang.reflect`包，在Java 2时代就有，在Java 5得到完善

- 反射: **reflection**

- 在运行中分析类的能力
- 在运行中查看和操作对象
 - 基于反射自由创建对象
 - 反射构建出无法直接访问的类
 - **set**或者**get**到无法访问的成员变量
 - 调用不可访问的方法
- 实现通用的数组操作代码
- 类似函数指针的功能

创建对象(1)

- 问题：如何(自由的)创建一个对象来调用它的方法？
- 方法1：静态编码&编译

```
public class A {  
    public void hello()  
    {  
        System.out.println("hello from A");  
    }  
}
```

//第一种 直接new 调用构造函数

```
A obj1 = new A();  
obj1.hello();
```


方法2: 克隆(clone)

```
public class B implements Cloneable {  
    public void hello()  
    {  
        System.out.println("hello from B");  
    }  
  
    protected Object clone() throws CloneNotSupportedException  
    {  
        return super.clone();  
    }  
}
```

//第二种 clone

//obj3 是obj2的克隆对象 没有调用构造函数

```
B obj2 = new B();
```

```
obj2.hello();
```

```
B obj3 = (B) obj2.clone();
```

```
obj3.hello();
```

方法3：序列化(serialization)和反序列化(deserialization)

```
public class C implements Serializable {  
    private static final long serialVersionUID = 1L;  
  
    public void hello() {  
        System.out.println("hello from C");  
    }  
}
```

//第三种 序列化 没有调用构造函数

//序列化会引发安全漏洞，未来将被移除出JDK，请谨慎使用!!!

```
C obj4 = new C();
```

```
ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream("data.obj"));
```

```
out.writeObject(obj4);
```

```
out.close();
```

```
ObjectInputStream in = new ObjectInputStream(new FileInputStream("data.obj"));
```

```
C obj5 = (C) in.readObject();
```

```
in.close();
```

```
obj5.hello();
```

方法4和5：反射

//第四种 newInstance

```
Object obj6 = Class.forName("A").newInstance();  
Method m = Class.forName("A").getMethod("hello");  
m.invoke(obj6);
```

```
A obj7 = (A) Class.forName("A").newInstance();
```

//第五种 newInstance 调用构造函数

```
Constructor<A> constructor = A.class.getConstructor();  
A obj8 = constructor.newInstance();  
obj8.hello();
```

IOC

把普通pojo实例化到Spring容器中

```
1 | @Component("userDao")
2 | public class UserDao{
3 |
4 | }
```

@Autowired 注解:userDao 成员变量会被自动注入 UserDao 类型的 Bean

```
1 | @Component
2 | public class UserService {
3 |
4 |     @Autowired
5 |     private UserDao userDao;
6 | }
```

反射机制

把普通pojo实例化到Spring容器中

```
1 | @Component("userDao")
2 | public class UserDao{
3 |
```

```
1 | <bean id="userDao" class="cn.my.UserDao"/>
```

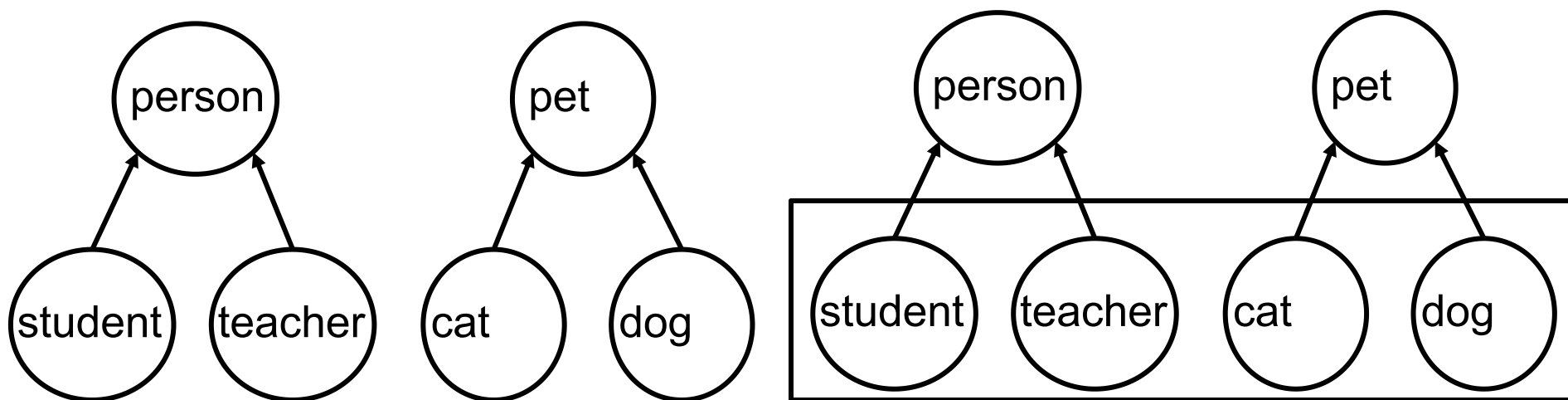
Spring通过配置进行实例化对象,并放到容器中的伪代码:

```
//解析<bean .../>元素的id属性得到该字符串值为“userDao”
String idStr = "userDao";
//解析<bean .../>元素的class属性得到该字符串值为“cn.my.UserDao”
String classStr = "cn.my.UserDao";
//利用反射机制,通过classStr获取Class类对象
Class<?> cls = Class.forName(classStr);
//实例化对象
Object obj = cls.newInstance();
//container表示Spring容器
container.put(idStr, obj);
```


AOP: Aspect Oriented Programming

-面向切面编程(vs 面向对象编程)

- 面向对象：将需求功能划分为不同的、独立，封装良好的类，并让它们通过继承和多态实现相同和不同行为。
- 面向切面：将通用需求功能从众多类中分离出来，使得很多类共享一个行为，一旦发生变化，不必修改很多类，而只需要修改这个行为即可。



示例

```
public void save(String name) {
```

```
    try {  
        System.out.println("开启事务");  
        System.out.println("处理业务逻辑, 调用DAO~~~");  
        System.out.println("提交事务");  
    } catch (Exception e) {  
        System.out.println("回滚事务");  
        e.printStackTrace();  
    }  
}
```

```
@Override
```

```
public void delete(String id) {
```

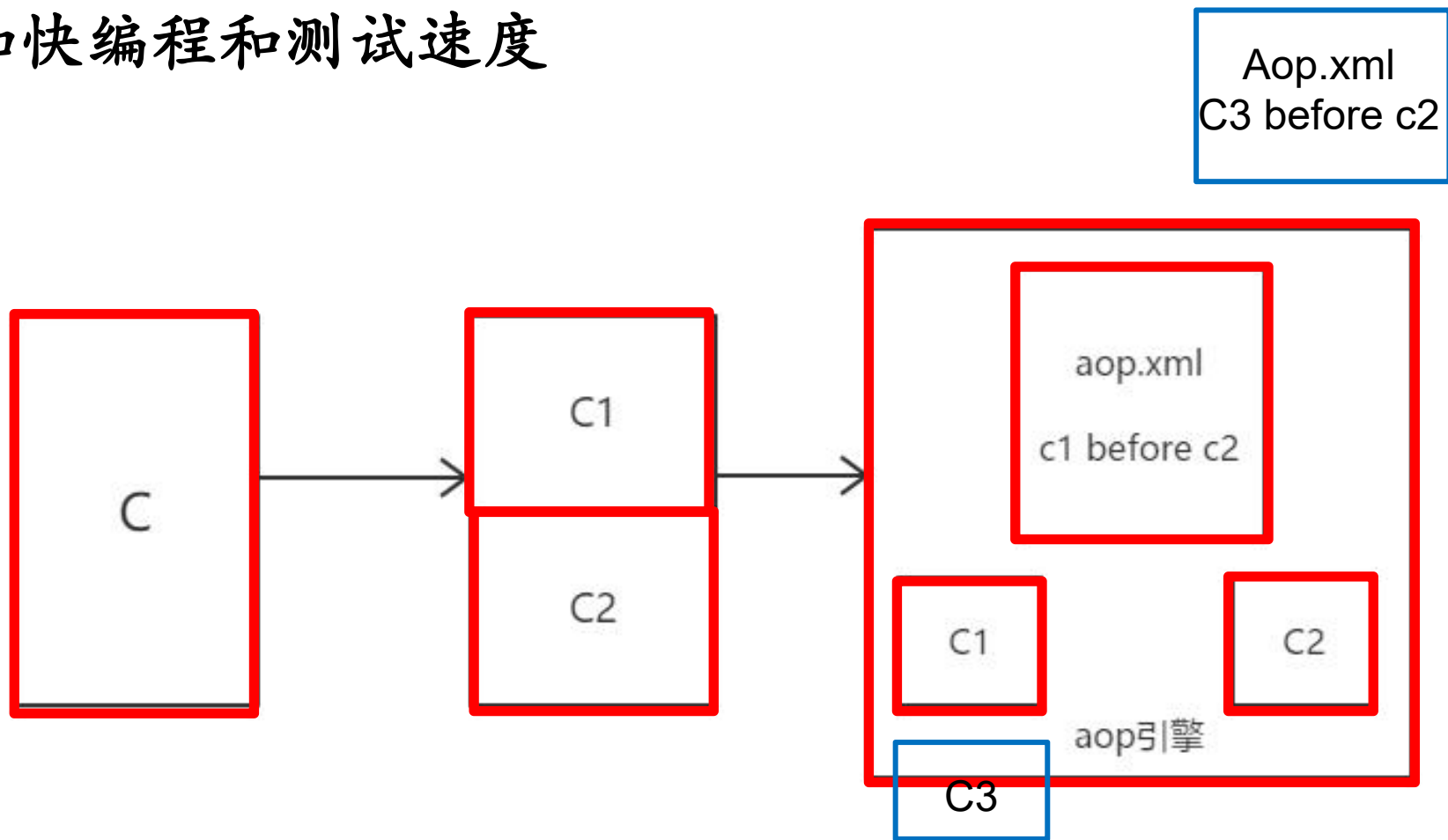
```
    try {  
        System.out.println("开启事务");  
        System.out.println("处理业务逻辑, 调用DAO~~~");  
        System.out.println("提交事务");  
    } catch (Exception e) {  
        System.out.println("回滚事务");  
        e.printStackTrace();  
    }  
}
```

业务层核心功能出现了代码的冗余

I

面向切面编程

- 分离代码的耦合(高内聚，低耦合)
- 业务逻辑变化不需要修改源代码/不用重启
- 加快编程和测试速度




```
public interface EatAction {
```

1 个用法 2 个实现

```
void eat();
```

0 个用法 2 个实现

```
void bath();
```

0 个用法 2 个实现

```
void clean();
```

```
public class CatImpl implements EatAction {
```

1 个用法

```
public void eat() { System.out.println("The cat is eating"); }
```

0 个用法

```
public void bath() {
```

```
    System.out.println("The cat's washing itself");
```

```
}
```

0 个用法

```
public void clean() {
```

```
    System.out.println("She's cleaning");
```

```
}
```

```
public class PersonImpl implements EatAction {
```

1 个用法

```
    public void eat() { System.out.println("I am eating"); }
```

0 个用法

```
    public void bath() {  
        System.out.println("I am bathing");  
    }
```

0 个用法

```
    public void clean() {  
        System.out.println("I am cleaning");  
    }
```

```
public class ProxyHandler implements InvocationHandler {
```

4 个用法

```
static String beforeMethod = "";
```

4 个用法

```
static String afterMethod = "";
```

5 个用法

```
private EatAction receiverObject;
```

1 个用法

```
public ProxyHandler(EatAction object) { this.receiverObject = object; }
```

@Override



```
public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
    // 处理before方法
    if(beforeMethod!=null&&beforeMethod.length()>0){
        ClassLoader cl = ProxyHandler.class.getClassLoader();
        Class<?> c = cl.loadClass(receiverObject.getClass().getName());
        Method m=c.getMethod(beforeMethod);
        Object obj = c.newInstance();
        m.invoke(obj);
    }
    // 处理目标方法
    Object result = method.invoke(receiverObject, args);
    // 处理after方法
    if(afterMethod!=null&&afterMethod.length()>0){
        method.invoke(receiverObject, args);
        ClassLoader cl = ProxyHandler.class.getClassLoader();
        Class<?> c = cl.loadClass(receiverObject.getClass().getName());
        Method m=c.getMethod(afterMethod);
        Object obj = c.newInstance();
        m.invoke(obj);
    }
    return result;
}
```

<aops>
 <aop>
 <method>bath</method>
 <type>before</type>
 <method>eat</method>
 </aop>
 </aops>

实现PersonImpl对象方法调用的前置处理

```
public class Main {
    public static void initXml(){
        XmlReader.readXml( filePath: "aops.xml");
        ResourceListener.addListener( path: "C:\\Users\\86134\\project\\AOP_test2");
    }
    public static void main(String[] args) throws Exception{
        Main.initXml();
        EatAction action = new PersonImpl();
        ProxyHandler mh = new ProxyHandler(action);
        ClassLoader cl = Main.class.getClassLoader();
        Class<?> proxyClass = Proxy.getProxyClass(cl, new Class<?>[]{EatAction.class});
        EatAction proxy = (EatAction) proxyClass.getConstructor(new Class[]{InvocationHandler.class}
            newInstance(new Object[]{mh});
        while(true)
        {
            proxy.eat();
            try{
                Thread.sleep( millis: 3000);
            }
            catch(Exception e){
                e.printStackTrace();
            }
        }
    }
}
```

<aops>

<aop>

<method>bath</method>

<type>before</type>

<method>eat</method>

</aop>

</aops>

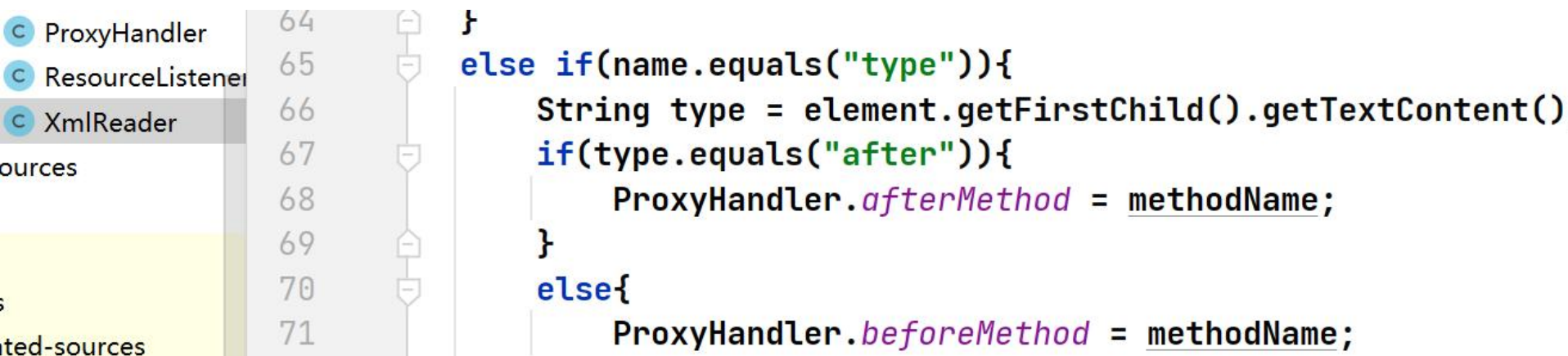
I am bathing

I am eating

I am bathing

I am eating

前置方法和后置方法如何生效？



```
64 }  
65 else if(name.equals("type")){  
66     String type = element.getFirstChild().getTextContent()  
67     if(type.equals("after")){  
68         ProxyHandler.afterMethod = methodName;  
69     }  
70     else{  
71         ProxyHandler.beforeMethod = methodName;
```

```
<aops>  
  <aop>  
    <method>bath</method>  
    <type>before</type>  
    <method>eat</method>  
  </aop>  
</aops>
```

实现CatImpl对象方法调用的前置处理

```
public class Main {
    public static void initXml(){
        XmlReader.readXml( filePath: "aops.xml");
        ResourceListener.addListener( path: "C:\\Users\\86134\\project\\AOP_test2");
    }
    public static void main(String[] args) throws Exception{
        Main.initXml();
        EatAction action = new CatImpl();
        ProxyHandler mh = new ProxyHandler(action);
        ClassLoader cl = Main.class.getClassLoader();
        Class<?> proxyClass = Proxy.getProxyClass(cl, new Class<?>[]{EatAction.class});
        EatAction proxy = (EatAction) proxyClass.getConstructor(new Class[]{InvocationHandler.class})
            .newInstance(new Object[]{mh});
        while(true)
        {
            proxy.eat();
            try{
                Thread.sleep( millis: 3000);
            }
            catch(Exception e){
                e.printStackTrace();
            }
        }
    }
}
```

The cat's washing itself
The cat is eating
The cat's washing itself
The cat is eating

Spring AOP实现

1. 定义AOP配置类，主要是开启AOP注解支持、扫描组件类的功能
@EnableAspectJAutoProxy注解最终往容器中注入了
AnnotationAwareAspectJAutoProxyCreator组件

```
package com.example.aop_test2;

import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.EnableAspectJAutoProxy;

@Configuration
@EnableAspectJAutoProxy
@ComponentScan("com.example.aop_test2")
public class AopConfig {

}
```

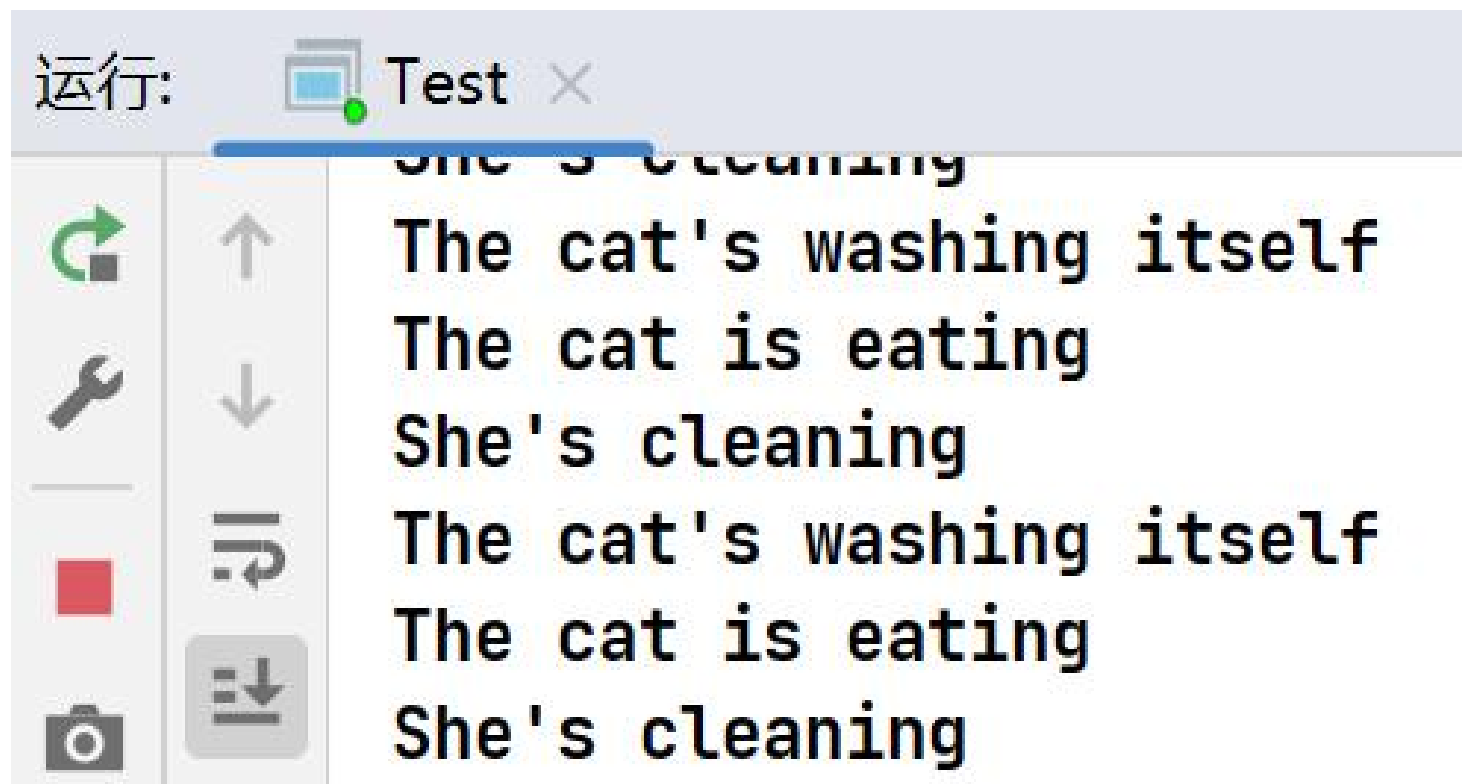

2. 定义切面类，使用 @Aspect 注解标识：

```
import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.After;
import org.aspectj.lang.annotation.AfterReturning;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.aspectj.lang.annotation.Pointcut;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;
// @Aspect: 告诉Spring这是一个切面类
@Aspect
@Component
public class SimpleAspect {
    2 个用法
    | @Pointcut("execution(* com.example.aop_test2...*(..))")
    private void pointcut() {
    }
    @Before("pointcut()")
    public void before(JoinPoint joinPoint) {
        EatAction catimpl = new CatImpl();
        catimpl.bath();
    }
}
```

3. 定义测试类:

```
public class Test {  
  
    public static void main(String[] args) {  
  
        AnnotationConfigApplicationContext annotationConfigApplicationContext =  
            new AnnotationConfigApplicationContext(AopConfig.class);  
        EatAction proxy = (EatAction) annotationConfigApplicationContext.getBean("catImpl");  
  
        while(true)  
        {  
            proxy.eat();  
            try{  
                Thread.sleep(3000);  
            }  
            catch(Exception e){  
                e.printStackTrace();  
            }  
        }  
    }  
}
```

输出结果:



```
运行: Test ×  
The cat's washing itself  
The cat is eating  
She's cleaning  
The cat's washing itself  
The cat is eating  
She's cleaning
```