常用设计模式&高并发方案&场景场景问题

一、设计模式

软件设计模式（Software Design Pattern），又称设计模式，是一套被反复使用、多数人知晓的、经过分类编目的、代码设计经验的总结。它描述了在软件设计过程中的一些不断重复发生的问题，以及该问题的解决方案。也就是说，它是解决特定问题的一系列套路，是前辈们的代码设计经验的总结，具有一定的普遍性，可以反复使用。

正确使用设计模式具有以下优点。

- 可以提高程序员的思维能力、编程能力和设计能力。
- 使程序设计更加标准化、代码编制更加工程化，使软件开发效率大大提高，从而缩短软件的开发周期。
- 使设计的代码可重用性高、可读性强、可靠性高、灵活性好、可维护性强。

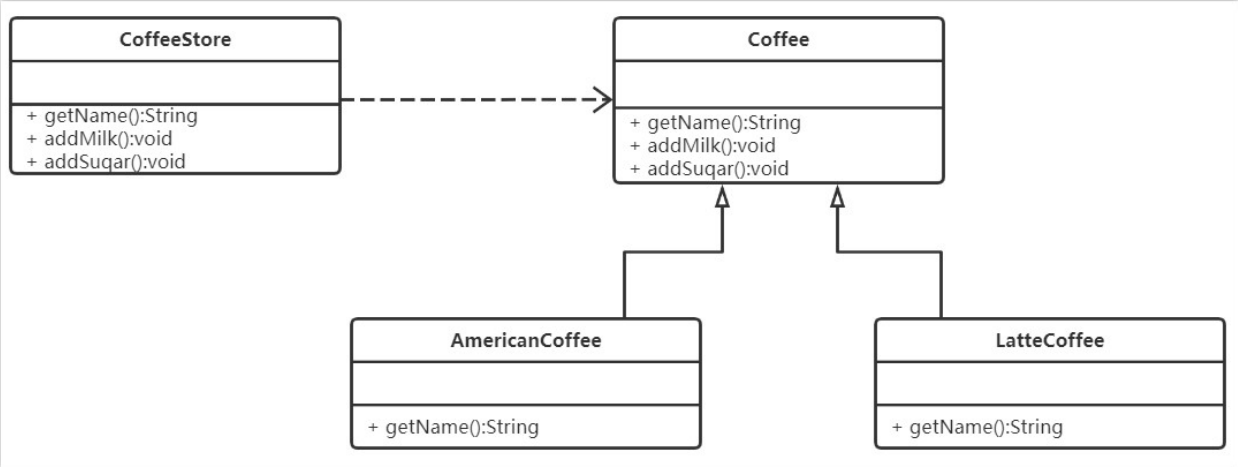
1 工厂方法模式

1.1 概述

需求：设计一个咖啡店点餐系统。

设计一个咖啡类（Coffee），并定义其两个子类（美式咖啡【AmericanCoffee】和拿铁咖啡【LatteCoffee】）；再设计一个咖啡店类（CoffeeStore），咖啡店具有点咖啡的功能。

具体类的设计如下：



1.类图中的符号

- +：表示public
- -：表示private
- #：表示protected

2.泛化关系(继承)用带空心三角箭头的实线来表示

3.依赖关系使用带箭头的虚线来表示

```
package com.itheima.factory.simple;

public class CoffeeStore {

    public static void main(String[] args) {
        Coffee coffee = orderCoffee("latte");
        System.out.println(coffee.getName());
    }


    public static Coffee orderCoffee(String type){
        Coffee coffee = null;
        if("american".equals(type)){
            coffee = new AmericanCoffee();
        }else if ("latte".equals(type)){
            coffee = new LatteCoffee();
        }

        //添加配料
        coffee.addMilk();
```

```
        coffee.addSuqar();
        return coffee;
    }
}
```

在java中，万物皆对象，这些对象都需要创建，如果创建的时候直接new该对象，就会对该对象耦合严重，假如我们要更换对象，所有new对象的地方都需要修改一遍，这显然违背了软件设计的**开闭原则**。如果我们使用工厂来生产对象，我们就只和工厂打交道就可以了，彻底和对象解耦，如果要更换对象，直接在工厂里更换该对象即可，达到了与对象解耦的目的；所以说，工厂模式最大的优点就是：**解耦**。

> 开闭原则：**对扩展开放，对修改关闭**。在程序需要进行拓展的时候，不能去修改原有的代码，实现一个热插拔的效果。简言之，是为了使程序的扩展性好，易于维护和升级。

三种工厂

- 简单工厂模式
- 工厂方法模式
- 抽象工厂模式

1.2 简单工厂模式

简单工厂不是一种设计模式，反而比较像是一种编程习惯。

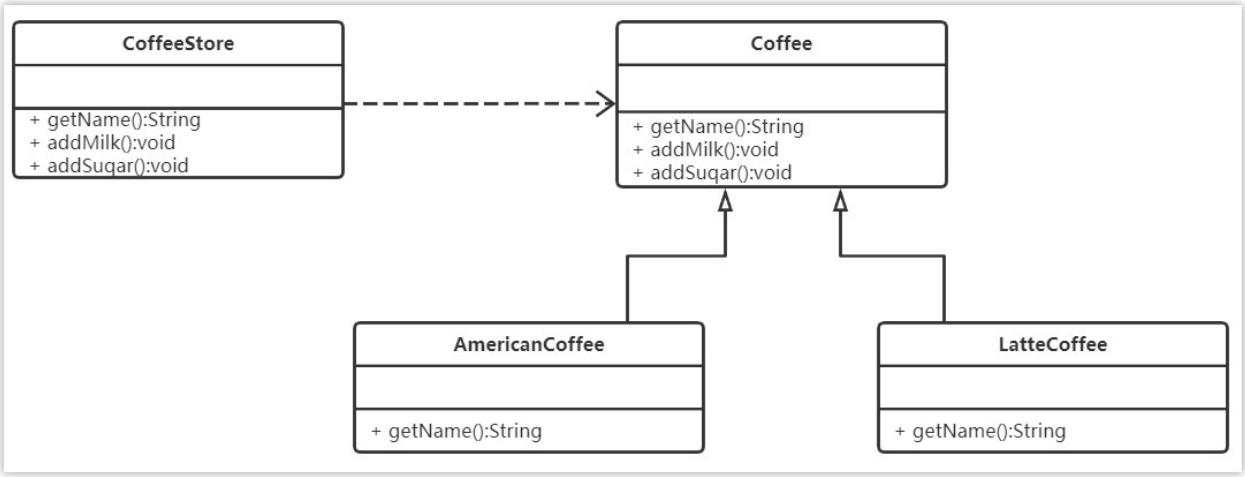
1.2.1 结构

简单工厂包含如下角色：

- 抽象产品 ：定义了产品的规范，描述了产品的主要特性和功能。
- 具体产品 ：实现或者继承抽象产品的子类
- 具体工厂 ：提供了创建产品的方法，调用者通过该方法来获取产品。

1.2.2 实现

现在使用简单工厂对上面案例进行改进，类图如下：



工厂类代码如下：

```
public class SimpleCoffeeFactory {

    public Coffee createCoffee(String type) {
        Coffee coffee = null;
        if("americano".equals(type)) {
            coffee = new AmericanoCoffee();
        } else if("latte".equals(type)) {
            coffee = new LatteCoffee();
        }
        return coffee;
    }
}
```

咖啡店

```
package com.itheima.factory.simple;

public class CoffeeStore {

    public Coffee orderCoffee(String type){
        //通过工厂获得对象，不需要知道对象实现的细节
        SimpleCoffeeFactory factory = new SimpleCoffeeFactory();
        Coffee coffee = factory.createCoffee(type);
        //添加配料
        coffee.addMilk();
```

```
        coffee.addSuqar();
        return coffee;
    }
}
```

工厂（factory）处理创建对象的细节，一旦有了SimpleCoffeeFactory，CoffeeStore类中的orderCoffee()就变成此对象的客户，后期如果需要Coffee对象直接从工厂中获取即可。这样也就解除了和Coffee实现类的耦合，同时又产生了新的耦合，CoffeeStore对象和SimpleCoffeeFactory工厂对象的耦合，工厂对象和商品对象的耦合。

后期如果再加新品种的咖啡，我们势必要需求修改SimpleCoffeeFactory的代码，违反了开闭原则。工厂类的客户端可能有很多，比如创建美团外卖等，这样只需要修改工厂类的代码，省去其他的修改操作。

1.2.3 优缺点

优点：

封装了创建对象的过程，可以通过参数直接获取对象。把对象的创建和业务逻辑层分开，这样以后就避免了修改客户代码，如果要实现新产品直接修改工厂类，而不需要在原代码中修改，这样就降低了客户代码修改的可能性，更加容易扩展。

缺点：

增加新产品时还是需要修改工厂类的代码，违背了“开闭原则”。

1.2.4 扩展

静态工厂

在开发中也有一部分人将工厂类中的创建对象的功能定义为静态的，这个就是静态工厂模式，它也不是23种设计模式中的。代码如下：

```
public class SimpleCoffeeFactory {

    public static Coffee createCoffee(String type) {
        Coffee coffee = null;
        if("americano".equals(type)) {
            coffee = new AmericanoCoffee();
        } else if("latte".equals(type)) {
            coffee = new LatteCoffee();
        }
        return coffe;
    }
}
```

1.3 工厂方法模式

针对上例中的缺点，使用工厂方法模式就可以完美的解决，完全遵循开闭原则。

1.3.1 概念

定义一个用于创建对象的接口，让子类决定实例化哪个产品类对象。工厂方法使一个产品类的实例化延迟到其工厂的子类。

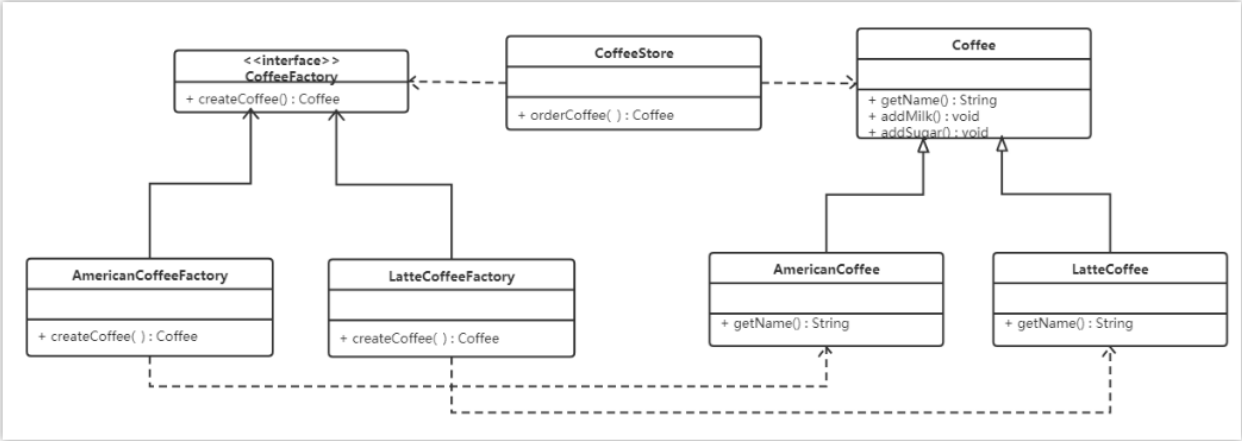
1.3.2 结构

工厂方法模式的主要角色：

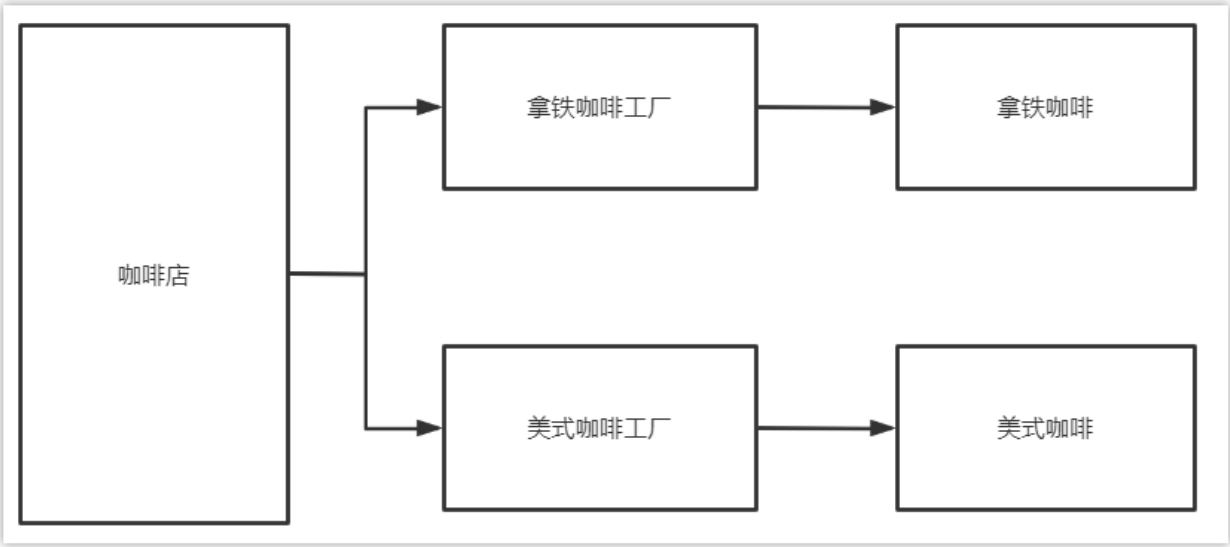
- 抽象工厂（Abstract Factory）：提供了创建产品的接口，调用者通过它访问具体工厂的工厂方法来创建产品。
- 具体工厂（ConcreteFactory）：主要是实现抽象工厂中的抽象方法，完成具体产品的创建。
- 抽象产品（Product）：定义了产品的规范，描述了产品的主要特性和功能。
- 具体产品（ConcreteProduct）：实现了抽象产品角色所定义的接口，由具体工厂来创建，它同具体工厂之间一一对应。

1.3.3 实现

使用工厂方法模式对上例进行改进，类图如下：



流程：



代码如下：

抽象工厂：

```
public interface CoffeeFactory {

    Coffee createCoffee();
}
```

具体工厂：

```
public class LatteCoffeeFactory implements CoffeeFactory {

    public Coffee createCoffee() {
        return new LatteCoffee();
    }
}

public class AmericanCoffeeFactory implements CoffeeFactory {

    public Coffee createCoffee() {
        return new AmericanCoffee();
    }
}
```

咖啡店类：

```
public class CoffeeStore {

    private CoffeeFactory factory;

    public CoffeeStore(CoffeeFactory factory) {
        this.factory = factory;
    }

    public Coffee orderCoffee(String type) {
        Coffee coffee = factory.createCoffee();
        coffee.addMilk();
        coffee.addsugar();
        return coffee;
    }
}
```

从以上的编写的代码可以看到，要增加产品类时也要相应地增加工厂类，不需要修改工厂类的代码了，这样就解决了简单工厂模式的缺点。

工厂方法模式是简单工厂模式的进一步抽象。由于使用了多态性，工厂方法模式保持了简单工厂模式的优点，而且克服了它的缺点。

1.3.4 优缺点

优点：

- 用户只需要知道具体工厂的名称就可得到所要的产品，无须知道产品的具体创建过程；
- 在系统增加新的产品时只需要添加具体产品类和对应的具体工厂类，无须对原工厂进行任何修改，满足开闭原则；

缺点：

- 每增加一个产品就要增加一个具体产品类和一个对应的具体工厂类，这增加了系统的复杂度。

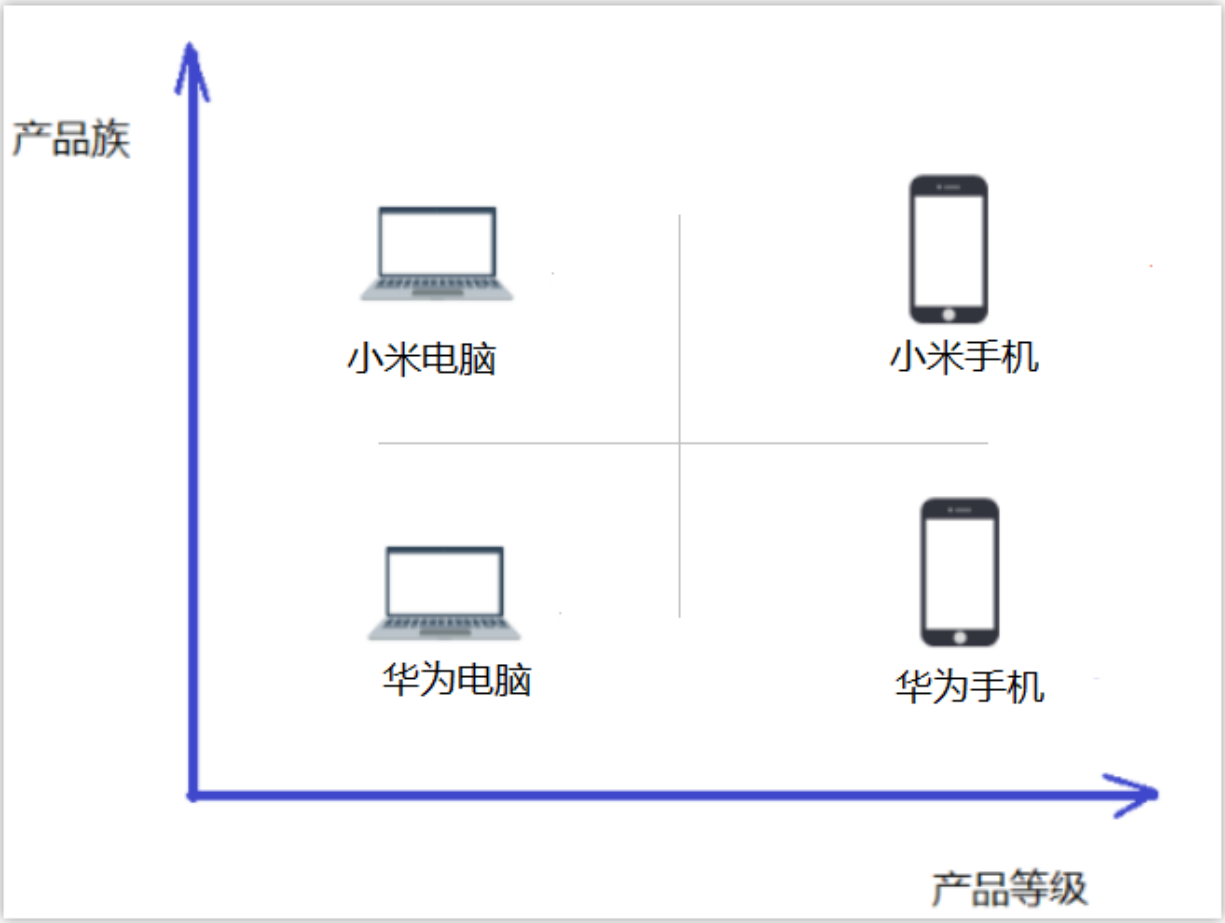
1.4 抽象工厂模式

前面介绍的工厂方法模式中考虑的是一类产品的生产，如畜牧场只养动物、电视机厂只生产电视机、传智播客只培养计算机软件专业的学生等。

这些工厂只生产同种类产品，同种类产品称为同等级产品，也就是说：工厂方法模式只考虑生产同等级的产品，但是在现实生活中许多工厂是综合型的工厂，能生产多等级（种类） 的产品，如电器厂既生产电视机又生产洗衣机或空调，大学既有软件专业又有生物专业等。

本节要介绍的抽象工厂模式将考虑多等级产品的生产，将同一个具体工厂所生产的位于不同等级的一组产品称为一个产品族，下图所示

- 产品族：一个品牌下面的所有产品；例如华为下面的电脑、手机称为华为的产品族；
- 产品等级：多个品牌下面的同种产品；例如华为和小米都有手机电脑为一个产品等级；



1.4.1 概念

是一种为访问类提供一个创建一组相关或相互依赖对象的接口，且访问类无须指定所要产品的具体类就能得到同族的不同等级的产品的模式结构。

抽象工厂模式是工厂方法模式的升级版本，工厂方法模式只生产一个等级的产品，而抽象工厂模式可生产多个等级的产品。

一个超级工厂创建其他工厂。该超级工厂又称为其他工厂的工厂

1.4.2 结构

抽象工厂模式的主要角色如下：

- 抽象工厂（Abstract Factory）：提供了创建产品的接口，它包含多个创建产品的方法，可以创建多个不同等级的产品。
- 具体工厂（Concrete Factory）：主要是实现抽象工厂中的多个抽象方法，完成具体产品的创建。
- 抽象产品（Product）：定义了产品的规范，描述了产品的主要特性和功能，抽象工厂模式有多个抽象产品。

- 具体产品（ConcreteProduct）：实现了抽象产品角色所定义的接口，由具体工厂来创建，它 同具体工厂之间是多对一的关系。

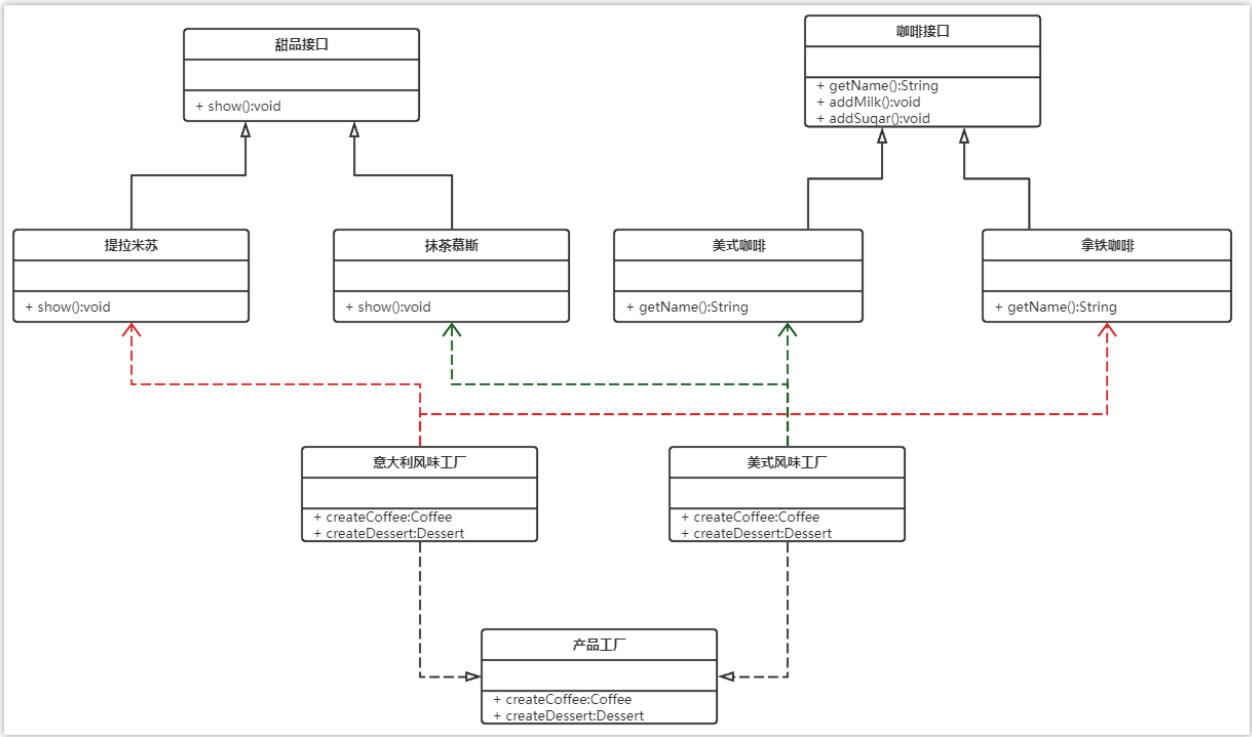
1.4.3 实现

现咖啡店业务发生改变，不仅要生产**咖啡**还要生产**甜点**

- 同一个产品等级（产品分类）
  - 咖啡：拿铁咖啡、美式咖啡
  - 甜点：提拉米苏、抹茶慕斯
- 同一个风味，就是同一个产品族（相当于同一个品牌）
  - 美式风味：美式咖啡、抹茶慕斯
  - 意大利风味：拿铁咖啡、提拉米苏

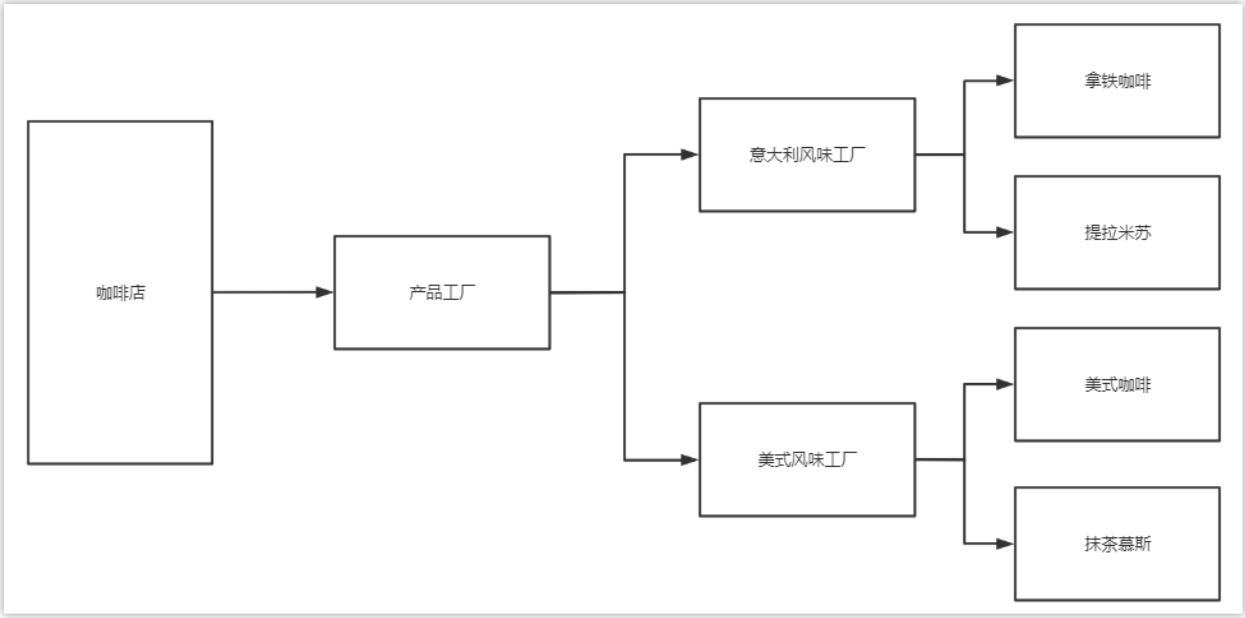
要是按照工厂方法模式，需要定义提拉米苏类、抹茶慕斯类、提拉米苏工厂、抹茶慕斯工厂、甜点工厂类，很容易发生类爆炸情况。

所以这个案例可以使用抽象工厂模式实现。类图如下：



> 实现关系使用带空心三角箭头的虚线来表示

整体调用思路：



1.4.4 优缺点

优点：

当一个产品族中的多个对象被设计成一起工作时，它能保证客户端始终只使用同一个产品族中的对象。

缺点：

当产品族中需要增加一个新的产品时，所有的工厂类都需要进行修改。

1.4.5 使用场景

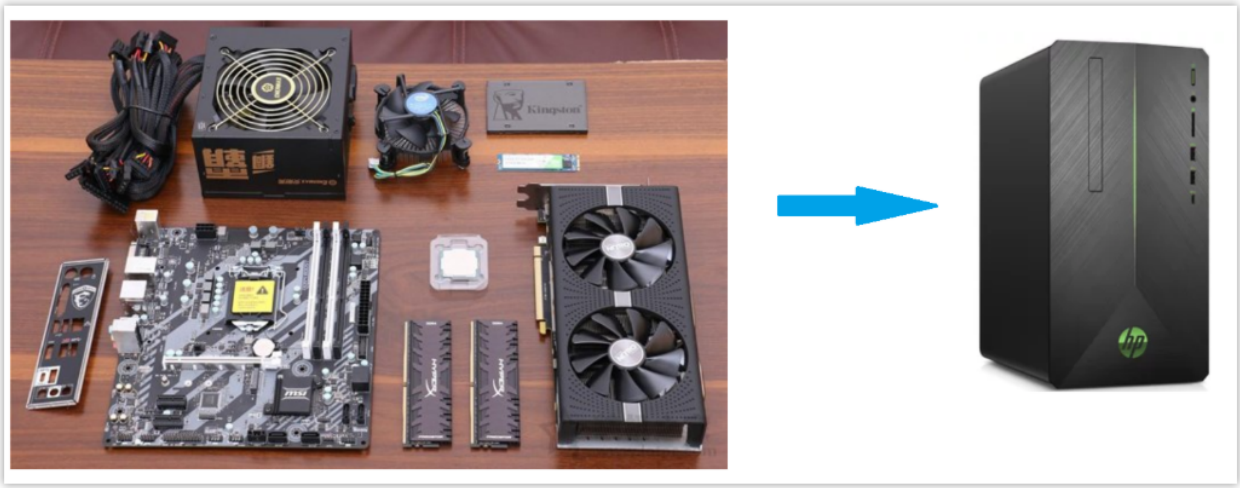
- 当需要创建的对象是一系列相互关联或相互依赖的产品族时，如电器工厂中的电视机、洗衣机、空调等。
- 系统中有多个产品族，但每次只使用其中的某一族产品。如有人只喜欢穿某一个品牌的衣服和鞋。
- 系统中提供了产品的类库，且所有产品的接口相同，客户端不依赖产品实例的创建细节和内部结构。

如：输入法换皮肤，一整套一起换。生成不同操作系统的程序。

2 构建者模式

2.1 概述

将一个复杂对象的构建与表示分离，使得同样的构建过程可以创建不同的表示。



- 分离了部件的构造(由Builder来负责)和装配(由Director负责)。 从而可以构造出复杂的对象。这个模式适用于：某个对象的构建过程复杂的情况。
- 由于实现了构建和装配的解耦。不同的构建器，相同的装配，也可以做出不同的对象；相同的构建器，不同的装配顺序也可以做出不同的对象。也就是实现了构建算法、装配算法的解耦，实现了更好的复用。
- 建造者模式可以将部件和其组装过程分开，一步一步创建一个复杂的对象。用户只需要指定复杂对象的类型就可以得到该对象，而无须知道其内部的具体构造细节。

2.2 优缺点

优点：

- 建造者模式的封装性很好。使用建造者模式可以有效的封装变化，在使用建造者模式的场景中，一般产品类和建造者类是比较稳定的，因此，将主要的业务逻辑封装在指挥者类中对整体而言可以取得比较好的稳定性。
- 在建造者模式中，客户端不必知道产品内部组成的细节，将产品本身与产品的创建过程解耦，使得相同的创建过程可以创建不同的产品对象。
- 可以更加精细地控制产品的创建过程 。将复杂产品的创建步骤分解在不同的方法中，使得创建过程更加清晰，也更方便使用程序来控制创建过程。
- 建造者模式很容易进行扩展。如果有新的需求，通过实现一个新的建造者类就可以完成，基本上不用修改之前已经测试通过的代码，因此也就不会对原有功能引入风险。符合开闭原则。

缺点：

造者模式所创建的产品一般具有较多的共同点，其组成部分相似，如果产品之间的差异性很大，则不适合使用建造者模式，因此其使用范围受到一定的限制。

2.3 总结

建造者（Builder）模式创建的是复杂对象，其产品的各个部分经常面临着剧烈的变化，但将它们组合在一起的算法却相对稳定，所以它通常在以下场合使用。

- 创建的对象较复杂，由多个部件构成，各部件面临着复杂的变化，但构件间的建造顺序是稳定的。
- 创建复杂对象的算法独立于该对象的组成部分以及它们的装配方式，即产品的构建过程和最终的表示是独立的。

2.4 黑马头条审核代码中引入构建者模式

2.4.1 构建对象类

```
package com.heima.wemedia.builder;

import com.heima.model.article.dtos.ArticleDto;
import com.heima.model.wemedia.pojos.WmChannel;
import com.heima.model.wemedia.pojos.WmNews;
import com.heima.model.wemedia.pojos.WmUser;
import com.heima.wemedia.mapper.WmChannelMapper;
import com.heima.wemedia.mapper.WmUserMapper;
import org.checkerframework.checker.units.qual.A;
import org.springframework.beans.BeanUtils;

import java.util.Date;

public class ArticleDtoBuilder {

    private WmNews wmNews;
    private WmChannelMapper wmChannelMapper;
    private WmUserMapper wmUserMapper;

    private ArticleDto articleDto = new ArticleDto();
```

```
    public ArticleDtoBuilder(WmNews wmNews, WmChannelMapper
wmChannelMapper, WmUserMapper wmUserMapper){
        this.wmNews = wmNews;
        this.wmChannelMapper = wmChannelMapper;
        this.wmUserMapper = wmUserMapper;
    }

    //构建属性拷贝
    public ArticleDtoBuilder buildBeanCopy(){
        BeanUtils.copyProperties(wmNews,articleDto);
        return this;
    }

    //构建作者对象到dto
    public ArticleDtoBuilder buildAuthor(){
        WmUser wmUser = wmUserMapper.selectById(wmNews.getUserId());
        if(wmUser != null){
            articleDto.setAuthorId(wmUser.getId().longValue());
            articleDto.setAuthorName(wmUser.getName());
        }
        return this;
    }

    //构建频道对象到dto
    public ArticleDtoBuilder buildChannel(){
        WmChannel wmChannel =
wmChannelMapper.selectById(wmNews.getChannelId());
        if(wmChannel != null){
            articleDto.setChannelName(wmChannel.getName());
        }
        return this;
    }

    //构建基本信息到dto
    public ArticleDtoBuilder buildBasic(){
        articleDto.setLayout(wmNews.getType());
        articleDto.setCreatedTime(new Date());
        articleDto.setCollection(0);
        articleDto.setComment(0);
        articleDto.setViews(0);
        articleDto.setLikes(0);
```

```
        if(wmNews.getArticleId() != null){
            articleDto.setId(wmNews.getArticleId());
        }

        return this;
    }

    //返回dto对象
    public ArticleDto builder(){
        return articleDto;
    }
}
```

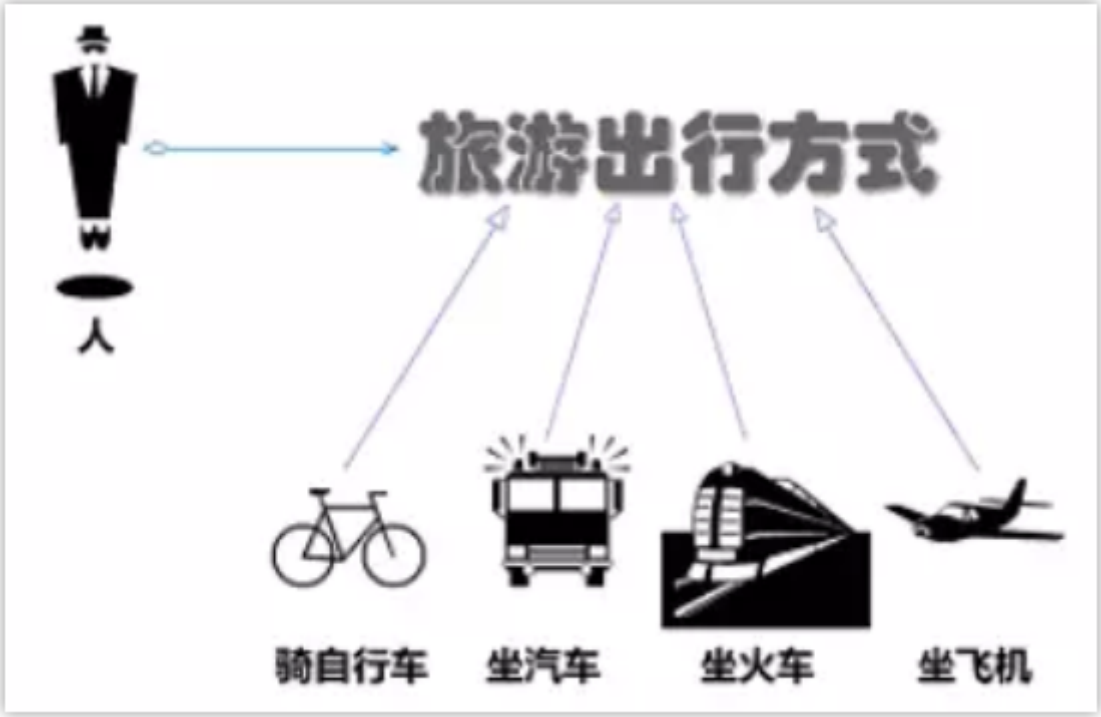
2.4.2 在审核代码中调用构建对象

```
/**
     * 保存app端的文章
     * @param wmNews
     */
@Override
public  ResponseResult saveAppArticle(WmNews wmNews) {

    ArticleDtoBuilder articleDtoBuilder = new
ArticleDtoBuilder(wmNews, wmUserMapper, wmChannelMapper);
    //使用构造器设计模式 方便以后维护
    //构造器设计模式：他最适合用于构建复杂得对象
    // 1、VO 对象
    // 2、一个系统调用另一个系统得时候， 另一个系统，需要一个对象来操作，
而且这个对象本身还挺复杂
    ArticleDto articleDto = articleDtoBuilder.buildBeanCopy()
        .buildAuthor()
        .buildBasic()
        .buildChannel()
        .builder();
    return   articleClient.saveAppArticle(articleDto);
}
```

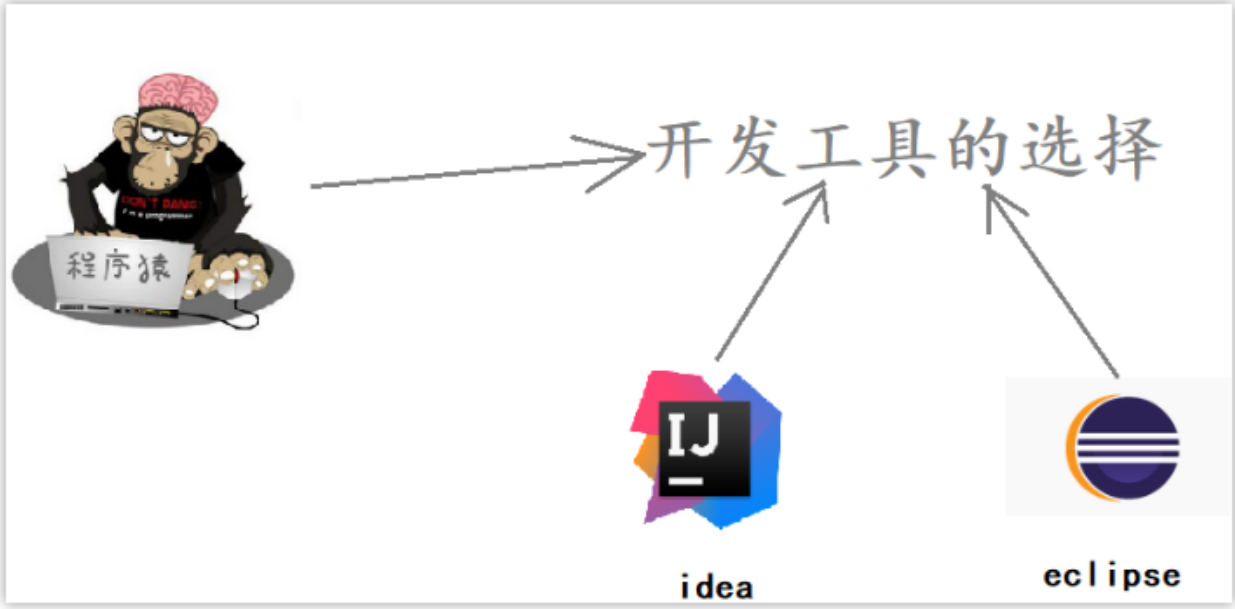
3 策略模式

3.1 概述

先看下面的图片，我们去旅游选择出行模式有很多种，可以骑自行车、可以坐汽车、可以坐火车、可以坐飞机。



作为一个程序猿，开发需要选择一款开发工具，当然可以进行代码开发的工具有很多，可以选择Idea进行开发，也可以使用eclipse进行开发，也可以使用其他的一些开发工具。



定义：

该模式定义了一系列算法，并将每个算法封装起来，使它们可以相互替换，且算法的变化不会影响使用算法的客户。策略模式属于对象行为模式，它通过对算法进行封装，把使用算法的责任和算法的实现分割开来，并委派给不同的对象对这些算法进行管理。

3.2 结构

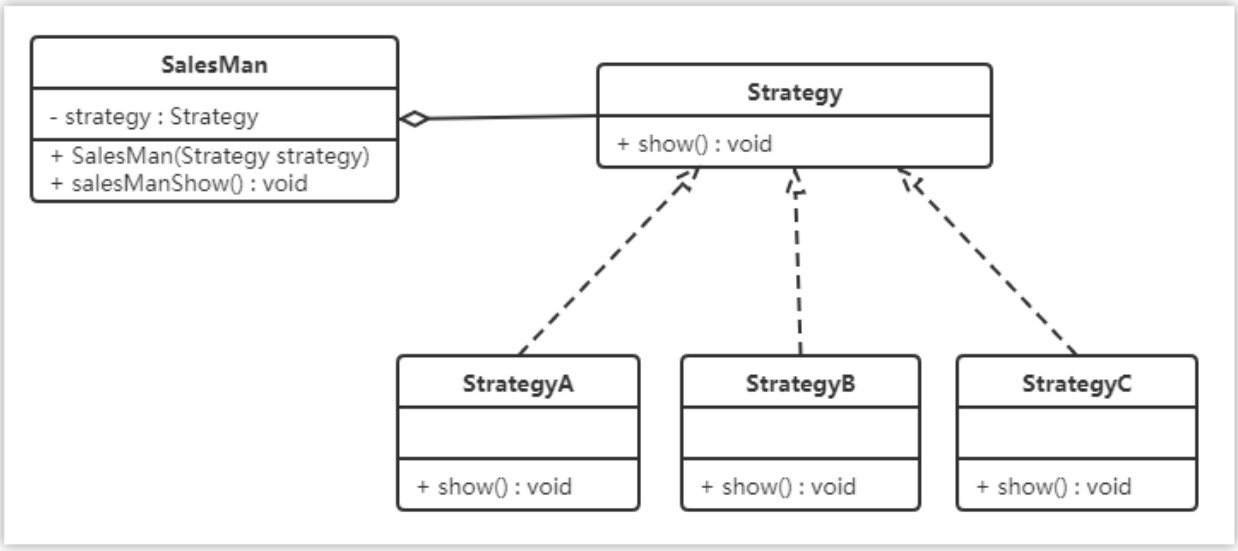
策略模式的主要角色如下：

- 抽象策略（Strategy）类：这是一个抽象角色，通常由一个接口或抽象类实现。此角色给出所有的具体策略类所需的接口。
- 具体策略（Concrete Strategy）类：实现了抽象策略定义的接口，提供具体的算法实现或行为。
- 环境（Context）类：持有一个策略类的引用，最终给客户端调用。

3.3 案例实现

【例】促销活动

一家百货公司在定年度的促销活动。针对不同的节日（春节、中秋节、圣诞节）推出不同的促销活动，由促销员将促销活动展示给客户。类图如下：



> 聚合关系可以用带空心菱形的实线来表示

代码如下：

定义百货公司所有促销活动的共同接口

```
public interface Strategy {
    void show();
}
```

定义具体策略角色（Concrete Strategy）：每个节日具体的促销活动

```
//为春节准备的促销活动A
public class StrategyA implements Strategy {

    public void show() {
        System.out.println("买一送一");
    }
}

//为中秋准备的促销活动B
public class StrategyB implements Strategy {

    public void show() {
        System.out.println("满200元减50元");
    }
}

//为圣诞准备的促销活动C
public class StrategyC implements Strategy {

    public void show() {
        System.out.println("满1000元加一元换购任意200元以下商品");
    }
}
```

定义环境角色（Context）：用于连接上下文，即把促销活动推销给客户，这里可以理解为销售员

```
public class SalesMan {
    //持有抽象策略角色的引用
    private Strategy strategy;

    public SalesMan(Strategy strategy) {
        this.strategy = strategy;
    }

    //向客户展示促销活动
    public void salesManShow(){
        strategy.show();
    }
}
```

4 工厂+策略综合案例

4.1 需求

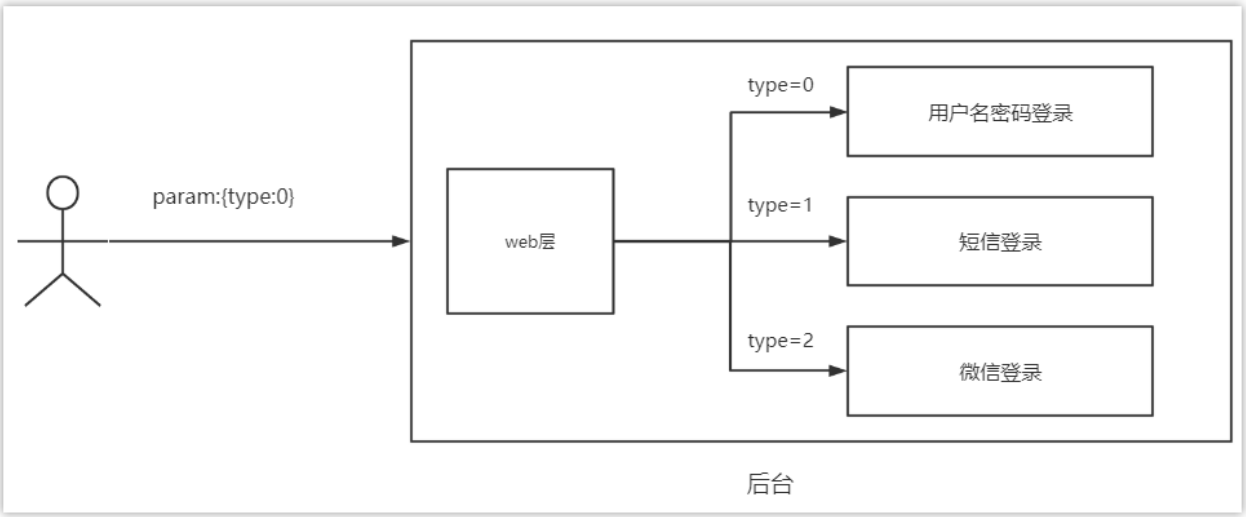
上图是gitee的登录的入口，其中有多种方式可以进行登录

- 用户名密码登录
- 短信验证码登录
- 微信登录
- QQ登录

....

4.2 后台实现思路

4.2.1 实现思路



> 1.前端选择登录方式后带参数请求后台，参数为type，0用户名密码登录 1短信登录 2微信登录
>
> 2.后台根据type的值选择哪种方式进行登录的逻辑处理

4.2.2 接口定义

	说明
请求方式	POST
路径	/api/user/login
参数	LoginReq
返回值	LoginResp

LoginReq

```
package com.itheima.model.dto;

import lombok.Data;

@Data
public class LoginReq {
```

```
    private String name;
    private String password;

    private String phone;
    private String validateCode;//手机验证码

    private String wxCode;//用于微信登录
    /**
     * 0 : 用户名密码登录
     * 1 : 微信登录
     * 2 : 手机验证码登录
     */
    private String type;
}
```

LoginResp

```
package com.itheima.model.vo;

import lombok.Data;

@Data
public class LoginResp{
    private Integer userId;
    private String userName;
    private String roleCode;
    private String token; //jwt令牌
    private boolean success;

}
```

4.2.3 后台代码实现

(1)控制层

```java
@RestController
@RequestMapping("/api/user")
public class LoginController {

    @Autowired
    private UserService userService;


    @PostMapping("/login")
    public LoginResp login(@RequestBody LoginReq loginReq){
        return userService.login(loginReq);
    }
}
```

(2)业务层

```java
@Service
public class UserService {

    public LoginResp login(LoginReq loginReq){

        if(loginReq.getType().equals("0")){
            System.out.println("用户名密码登录");

            //执行用户密码登录逻辑

            return new LoginResp();

        }else if(loginReq.getType().equals("1")){
            System.out.println("手机号验证码登录");

            //执行手机号验证码登录逻辑

            return new LoginResp();
        }else if (loginReq.getType().equals("2")){
            System.out.println("微信登录");

            //执行用户微信登录逻辑

            return new LoginResp();
        }
```

```
            LoginResp loginResp = new LoginResp();
            loginResp.setSuccess(false);
            System.out.println("登录失败");
            return loginResp;
        }
    }
```

> 注意：我们重点讲的是设计模式，并不是登录的逻辑，所以以上代码并没有真正的实现登录功能

(3)问题分析

- 业务层代码大量使用到了if...else，在后期阅读代码的时候会非常不友好，大量使用if...else性能也不高
- 如果业务发生变更，比如现在新增了QQ登录方式，这个时候需要修改业务层代码，违反了开闭原则

解决：

使用**模板方法设计模式+策略模式**解决

4.2.4 设计模式优化登录功能

(1)类图

(2)设计模式代码实现

抽象策略类：UserGranter

```java
/**
 * 抽象策略类
 */
public interface UserGranter{

    /**
     * 获取数据
     * @param loginReq 传入的参数
     * @return map值
     */
    LoginResp login(LoginReq loginReq);
}
```

具体的策略：AccountGranter、SmsGranter、WeChatGranter

```java
/**
 *  策略: 账号登录
 **/
@Component
public class AccountGranter implements UserGranter{

    @Override
    public LoginResp login(LoginReq loginReq) {

        System.out.println("登录方式为账号登录" + loginReq);
        // TODO
        // 执行业务操作

        return new LoginResp();
    }
}
/**
 * 策略:短信登录
 */
@Component
public class SmsGranter implements UserGranter{

    @Override
    public LoginResp login(LoginReq loginReq)  {

        System.out.println("登录方式为短信登录" + loginReq);
```

```
        // TODO
        // 执行业务操作

        return new LoginResp();
    }
}
/**
 * 策略:微信登录
 */
@Component
public class WeChatGranter implements UserGranter{

    @Override
    public LoginResp login(LoginReq loginReq)  {

        System.out.println("登录方式为微信登录" + loginReq);
        // TODO
        // 执行业务操作

        return new LoginResp();
    }
}
```

工程类：UserLoginFactory

```
/**
 * 策略工厂类
 */
@Component
public class UserLoginFactory {

    @Autowired
    private AccountGranter accountGranter;

    @Autowired
    private SmsGranter smsGranter;

    @Autowired
    private WeChatGranter weChatGranter;
```

```
    private static Map<String, UserGranter> granterPool = new
ConcurrentHashMap<>();

    @PostConstruct
    public void init(){

granterPool.put(UserLoginConstants.ACCOUNT_TYPE,accountGranter);
        granterPool.put(UserLoginConstants.PHONE_TYPE,smsGranter);

granterPool.put(UserLoginConstants.WEB_CHAT_TYPE,weChatGranter);
    }

    /**
     * 对外提供获取具体策略
     */
     public UserGranter getGranter(String grantType){
        UserGranter tokenGranter = granterPool.get(grantType);
        return tokenGranter;
    }

}
```

(3)业务层代码改造

```
@Service
public class UserService {

    @Autowired
    private UserLoginFactory factory;

    public LoginResp login(LoginReq loginReq){

        UserGranter granter =
factory.getGranter(loginReq.getType());
        if(granter == null){
            LoginResp loginResp = new LoginResp();
            loginResp.setSuccess(false);
            return loginResp
        }
        return granter.login(loginReq);
```

```
    }
}
```

大家可以看到我们使用了设计模式之后，业务层的代码就清爽多了，如果后期有新的需求改动，比如加入了QQ登录，我们只需要添加对应的策略就可以，无需再改动业务层代码。

4.2.5 举一反三

其实像这样的需求，在日常开发中非常常见，场景有很多，以下的情景都可以使用工厂模式+策略模式解决比如：

- 订单的支付策略
  - 支付宝支付
  - 微信支付
  - 银行卡支付
  - 现金支付
- 解析不同类型excel
  - xls格式
  - xlsx格式
- 打折促销
- 物流运费阶梯计算
  - 5kg以下
  - 5kg-10kg
  - 10kg-20kg
  - 20kg以上

一句话总结：只要代码中有冗长的 if-else 或 switch 分支判断都可以采用策略模式优化

二、高并发方案

1高并发概述

1.1 什么是高并发

高并发(High Concurrency)是一种系统在运行时遇到的一种**短时间内遇到大量操作请求**的情况，主要发生在对 Web系统的大量访问中收到大量请求。一般主要针对是某个或某些接口。这种情况的出现将导致系统在此时间内执行大量操作，如对资源的请求、数据库操作等。

> 例：
>
> 12306 ：多用户同一时间段抢一定量的火车票
>
> 天猫双十一 ：同一个时间段秒杀同一个商品

1.2 高并发相关常用的一些指标

- 响应时间（Response Time）：应用系统从请求发出开始到客户端收到响应所消耗的时间
- 吞吐量（Throughput）：并发数/平均响应时间
- 每秒查询率QPS（Query Per Second）：每秒请求数，就是服务器在一秒的时间内处理了多少个请求
- 并行用户数：同时承载正常使用系统功能的用户数量
- 页面访问量PV（PageView）：即页面浏览量或点击量，用户每次刷新即被计算一次
- 独立访客UV（Unique Visitor）：访问您网站的一台电脑客户端为一个访客
- 并发数 = QPS * 平均响应时间
- 峰值QPS计算公式
  - 原理：每天80%的访问集中在20%的时间里，这20%时间叫做峰值时间
  - 公式：(总PV数 * 80%) / (每天秒数 * 20%) = 峰值时间每秒请求数(QPS)
- 机器数量
  - 峰值时间每秒QPS / 单台机器的QPS

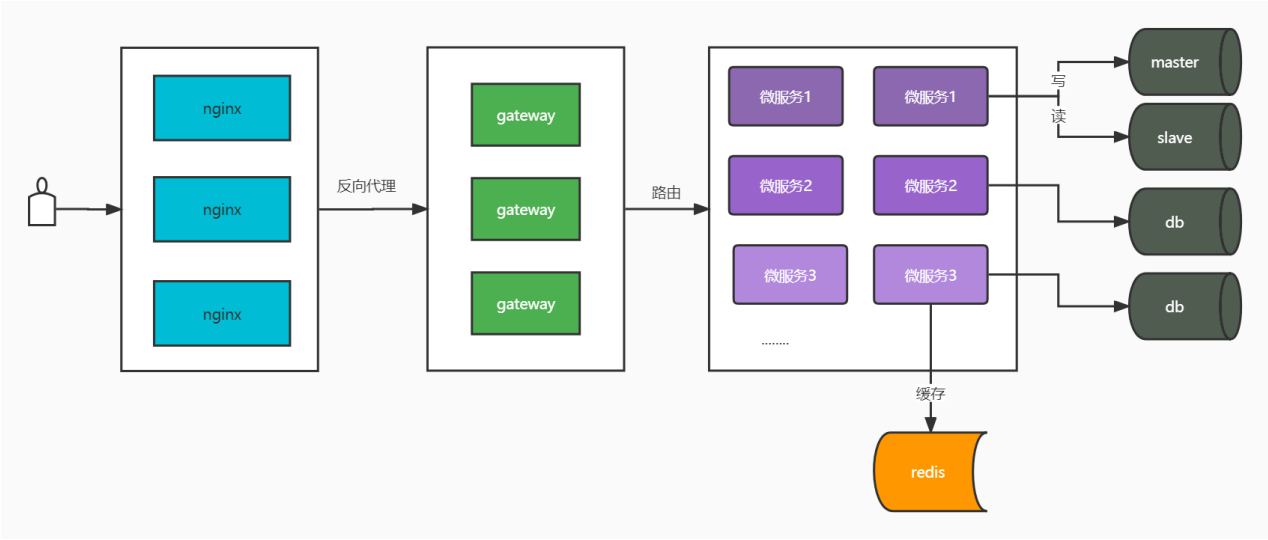
1.3 访问量的要求

在实际中，根据业务、规模等不一致访问量也是不一样的，这个并没有确切的一个标准。我们一般能达到预值即可。

1.4 如何提升系统的并发能力

- 垂直扩展：提升单机硬件性能，还可以提升单机系统性能，例如添加缓存
- 水平扩展：增加机器数量，集群能力。（重要）

1.5 水平架构扩展（基本方式）



（1）客户端层：浏览器browser或者手机应用APP

（2）负载均衡层：系统入口，反向代理

（3）网关层：接口的统一入口，可以做一些拦截（鉴权、认证、限流）和路由

（4）服务层：网关路由到具体的服务进行调用，量大可以集群

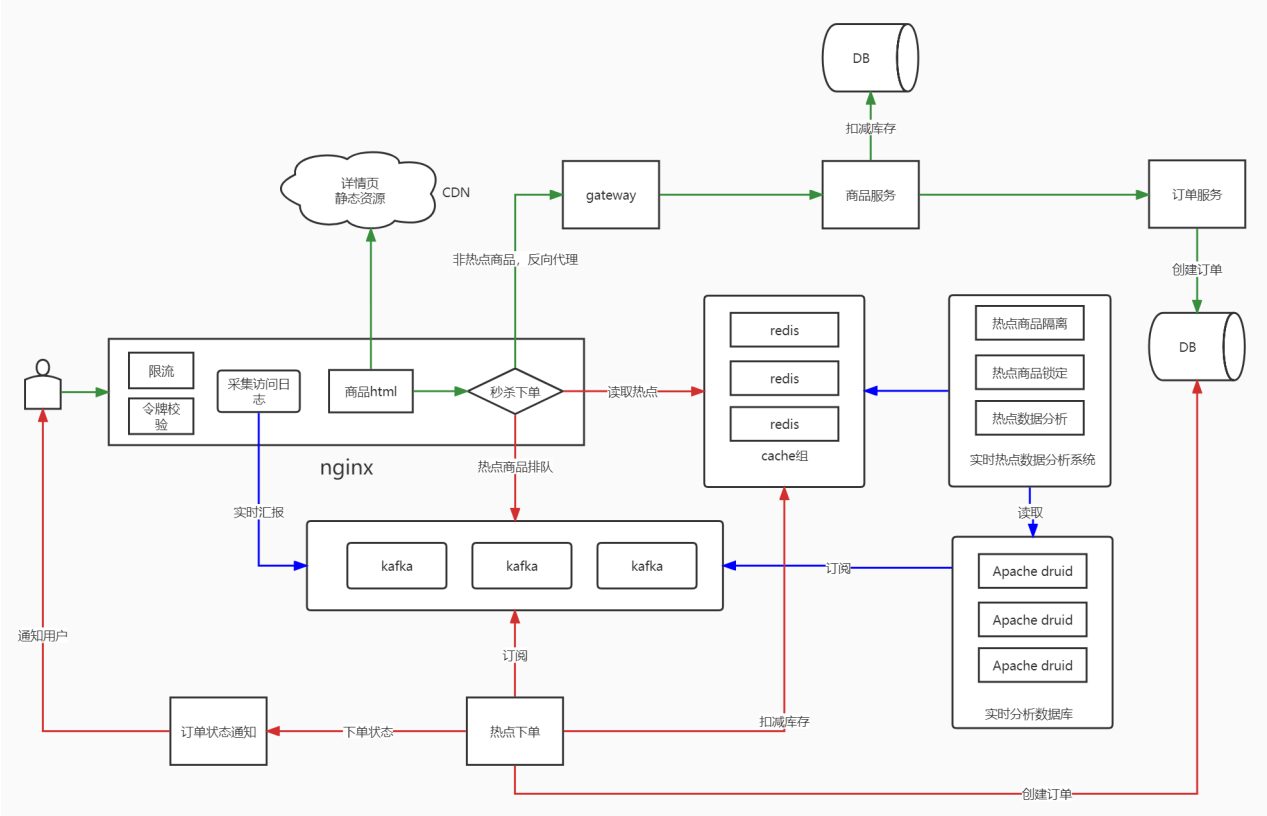
（5）数据-缓存层：缓存加速访问存储

（6）数据-数据库层：数据库持久化数据存储

> 注：
>
> 以上的架构设计只是一种通用的架构模式，在实际的业务中，有很可能会有较大的变化。不同的业务所需要的组件也不太一样。

2.秒杀场景

下面是一个成熟的秒杀场景的架构图，这里面的组件就会更多一些



非热点商品：

1.用户请求先到nginx，访问商品详情页面，其中详情页面的静态资源到CDN访问

2.在nginx中使用lur脚本进行判断

- Jwt令牌校验(登录判断)
- 该用户是否在指定时间内购买过
- 是否为热点商品
- 热点商品是否有库存
- 是否正在排队

3.用户点击了非热点商品，nginx负责反向代理找到网关

4.网关路由的商品微服务进行扣减库存

5.商品微服务feign调用订单微服务创建订单

> 注意：第4、5步是一个原子操作，需要使用分布式事务解决

热点商品

1.用户请求先到nginx，访问商品详情页面，其中详情页面的静态资源到CDN访问

2.在nginx中使用lua脚本进行判断

- Jwt令牌校验(登录判断)
- 该用户是否在指定时间内购买过
- 是否为热点商品
- 热点商品是否有库存
- 是否正在排队

3.用户点击了热点商品，发消息给kakfa进行异步排队

4.热点下单服务订阅kafka接收消息，直接创建热点订单

5.扣减库存（热点数据都在redis中，需要扣减redis中的商品库存）

6.下单成功则需要通知订单状态服务

7.订单状态服务通知用户购买结果(技术：WebSocket)

> 注意：在第5步中扣减库存，为了防止超卖，需要分布式锁解决

热点商品分析

1.用户点击了商品，在nginx中记录用户操作，发日志消费给kafka

2.实时分析数据库Apache druid存储数据

3.实时热点分析系统从Apache druid中读取数据，根据指标（比如页面点击量1分钟内超过1000次）获得热点数据

4.实时热点分析系统，隔离锁定热点数据，并同步给redis

3.负载均衡（SLB）、LVS

负载均衡（Server Load Balancer ）是高并发、高可用系统必不可少的关键组件，目标是 尽力将网络流量平均分发到多个服务器上，以提高系统整体的响应速度和可用性。

3.1 负载均衡作用

- 高并发：负载均衡通过算法调整负载，尽力均匀的分配应用集群中各节点的工作量，以此提高应用集群的并发处理能力（吞吐量）。
- 伸缩性：添加或减少服务器数量，然后由负载均衡进行分发控制。这使得应用集群具备伸缩性。
- 高可用：负载均衡器可以监控候选服务器，当服务器不可用时，自动跳过，将请求分发给可用的服务器。这使得应用集群具备高可用的特性。
- 安全防护：有些负载均衡软件或硬件提供了安全性功能，如：黑白名单处理、防火墙，防 DDos 攻击等。

3.2 负载均衡分类

从支持负载均衡的载体来看，可以将负载均衡分为两类：**硬件负载均衡、软件负载均衡**

3.2.1 硬件负载均衡

硬件负载均衡，一般是在定制处理器上运行的独立负载均衡服务器，价格昂贵，土豪专属。硬件负载均衡的主流产品有:F5 和 A10。

硬件负载均衡的 优点：

- 功能强大：支持全局负载均衡并提供较全面的、复杂的负载均衡算法。
- 性能强悍：硬件负载均衡由于是在专用处理器上运行，因此吞吐量大，可支持单机百万以上的并发。
- 安全性高：往往具备防火墙，防 DDos 攻击等安全功能。

> DDos 攻击:
>
> 分布式拒绝服务攻击(英文意思是Distributed Denial of Service，简称DDoS）
>
> 指处于不同位置的多个攻击者同时向一个或数个目标发动攻击，或者一个攻击者控制了位于不同位置的多台机器并利用这些机器对受害者同时实施攻击。由于攻击的发出点是分布在不同地方的，这类攻击称为分布式拒绝服务攻击，其中的攻击者可以有多个

硬件负载均衡的 缺点：

- 成本昂贵：购买和维护硬件负载均衡的成本都很高。
- 扩展性差：当访问量突增时，超过限度不能动态扩容。

3.2.2 软件负载均衡

软件负载均衡，应用最广泛，无论大公司还是小公司都会使用。

软件负载均衡从软件层面实现负载均衡，一般可以在任何标准物理设备上运行。

软件负载均衡的 主流产品 有：**Nginx**、**HAProxy**、**LVS**。

- LVS 可以作为四层负载均衡器。其负载均衡的性能要优于 Nginx。
- HAProxy 可以作为 HTTP 和 TCP 负载均衡器。
- **Nginx**、HAProxy 可以作为四层或七层负载均衡器。

软件负载均衡的 优点：

- 扩展性好：适应动态变化，可以通过添加软件负载均衡实例，动态扩展到超出初始容量的能力。
- 成本低廉：软件负载均衡可以在任何标准物理设备上运行，降低了购买和运维的成本。

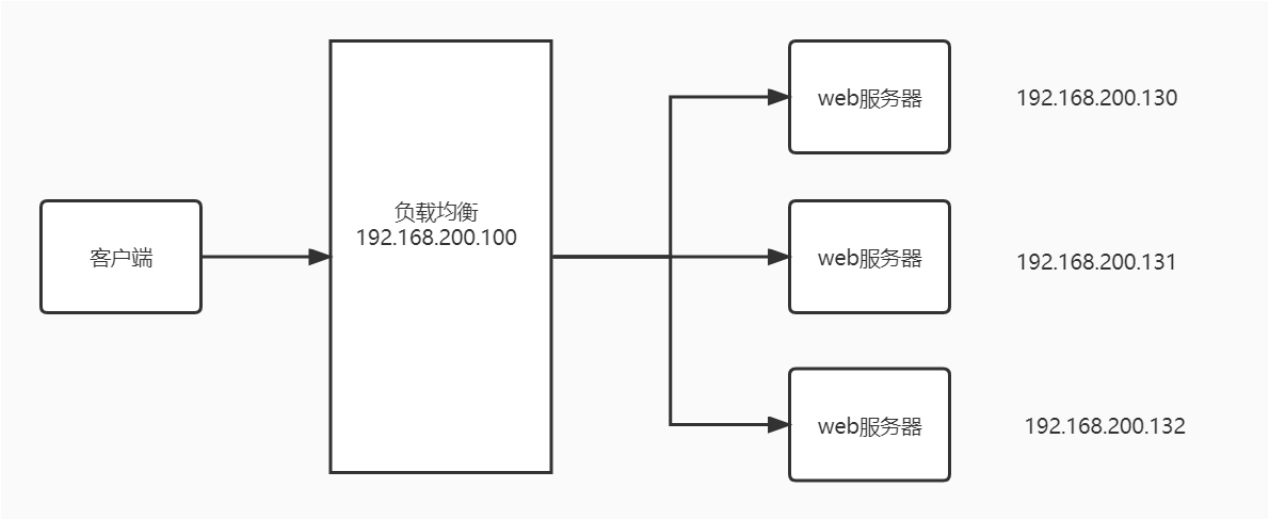
软件负载均衡的 缺点：

- 性能略差：相比于硬件负载均衡，软件负载均衡的性能要略低一些。

3.3.3 反向代理负载均衡

反向代理（Reverse Proxy）方式是指以 代理服务器 来接受网络请求，然后 将请求转发给内网中的服务器，并将从内网中的服务器上得到的结果返回给网络请求的客户端。反向代理负载均衡属于七层负载均衡。

反向代理服务的主流产品：**Nginx**、**Apache**。



nginx中的配置

```
upstream targetserver{
    server 192.168.138.130:51601 weight=2;
    server 192.168.138.131:51601 weight=1;
    server 192.168.138.132:51601 backup;
}
server {
    listen       80;
    server_name  localhost;
    location / {
        proxy_pass http://targetserver;
    }
}
```

> server 192.168.138.130:51601 weight=2;
> server 192.168.138.131:51601 weight=1;
> server 192.168.138.132:51601 backup;
>
> 以上配置：
>
> 假如来有3次请求，130服务处理2次，131服务器处理1次。
>
> 假如其中130或131有一台服务器挂掉了，132服务器是作为热备服务器，会直接顶上

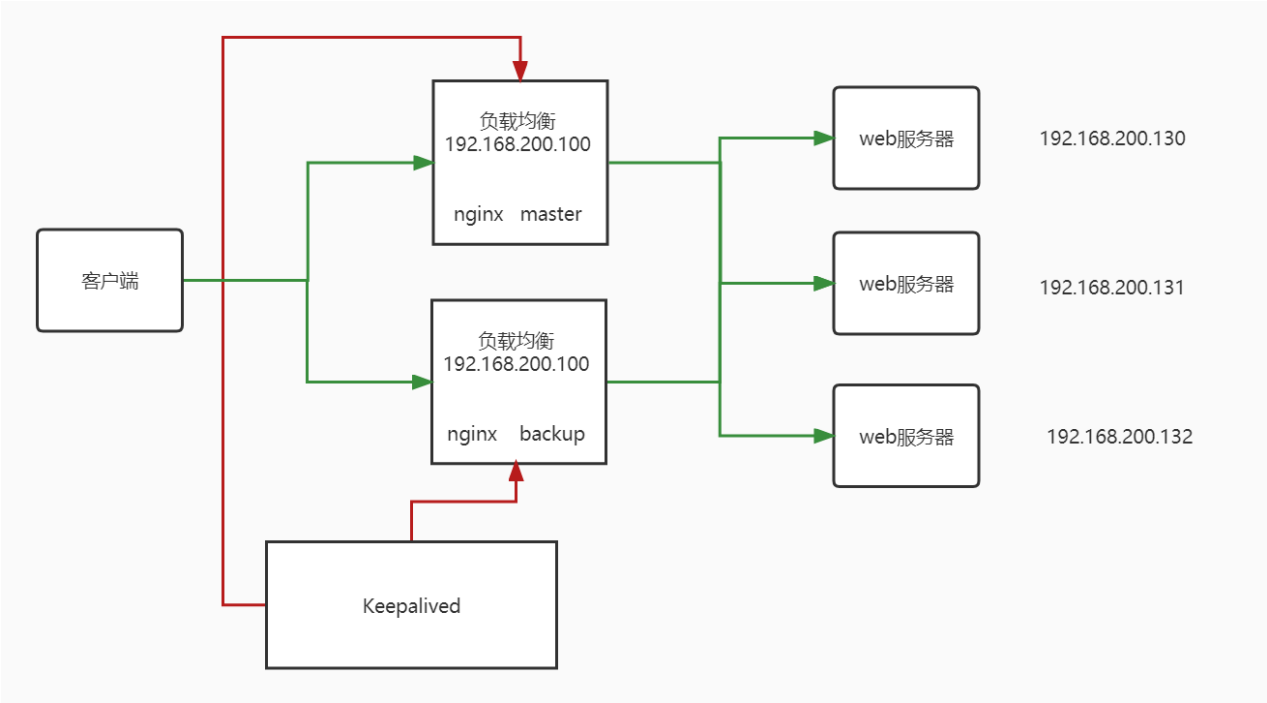
常见的nginx负载均衡算法

名称	说明

名称	说明
轮询	默认方式
weight	权重方式
ip_hash	依据ip分配方式
least_conn	依据最少连接方式
url_hash	依据url分配方式
fair	依据响应时间方式

3.3.4 nginx高可用

Keepalived是Linux下一个轻量级别高可用解决方案，高可用，既两台业务系统启动着相同的服务，如果有一台有故障，另一台自动接管。



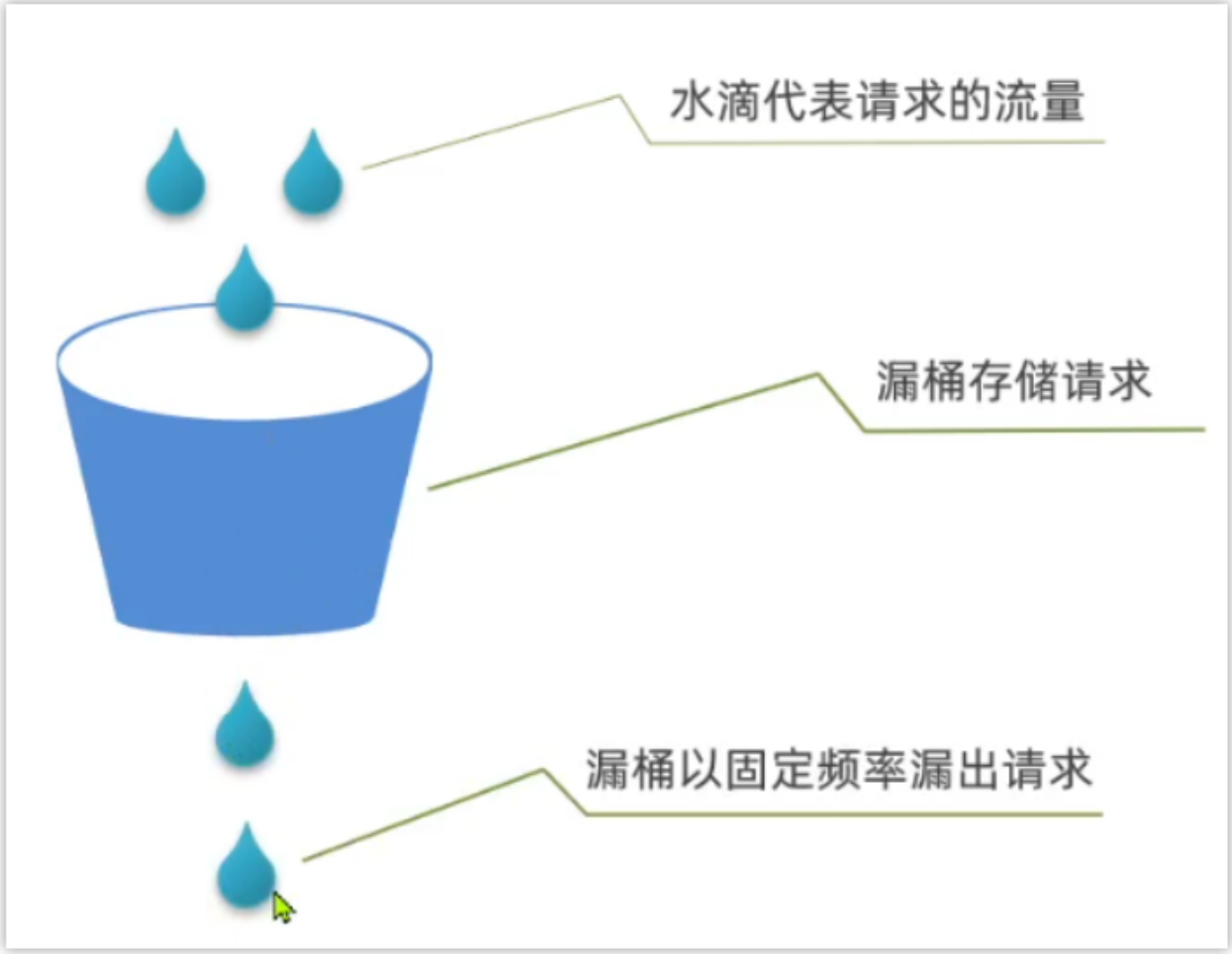
4.基于算法对秒杀进行整体限流

4.1 前端限流

nginx限流有两种方案：**控制速率**和**控制并发连接数**

4.1.1 控制速率(正常流量)

Nginx中我们使用ngx_http_limit_req_module模块来限制请求的访问频率，基于漏桶算法原理实现。接下来我们使用 nginx limit_req_zone 和 limit_req 两个指令，限制单个IP的请求处理速率。



具体配置：

语法：`limit_req_zone key zone rate`

```
http {
    limit_req_zone $binary_remote_addr zone=service1RateLimit:10m
rate=10r/s
    server {
        listen       80;
        server_name  localhost;
        location / {
            limit_req_zone = service1RateLimit;
            proxy_pass http://targetserver;
        }
    }
}
```

- key ：定义限流对象，binary_remote_addr 是一种key，表示基于 remote_addr(客户端IP) 来做限流，binary_ 的目的是压缩内存占用量。
- zone：定义共享内存区来存储访问信息， service1RateLimit:10m 表示一个大小为10M，名字为service1RateLimit的内存区域。1M能存储16000 IP地址的访问信息，10M可以存储16W IP地址访问信息。
- rate 用于设置最大访问速率，rate=10r/s 表示每秒最多处理10个请求。Nginx 实际上以毫秒为粒度来跟踪请求信息，因此 10r/s 实际上是限制：每100毫秒处理一个请求。这意味着，自上一个请求处理完后，若后续100毫秒内又有请求到达，将拒绝处理该请求。

4.1.2 控制速率(突发流量)

按上面的配置在流量突然增大时，超出的请求将被拒绝，无法处理突发流量，那么在处理突发流量的时候，该怎么处理呢？Nginx提供了 burst 参数来解决突发流量的问题，并结合 nodelay 参数一起使用。burst 译为突发、爆发，表示在超过设定的处理速率后能额外处理的请求数。

```
http {
    limit_req_zone $binary_remote_addr zone=service1RateLimit:10m
rate=10r/s
    server {
        listen       80;
        server_name  localhost;
        location / {
            limit_req_zone = service1RateLimit burst=20 nodelay;
            proxy_pass http://targetserver;
        }
    }
}
```

burst=20 nodelay表示这20个请求立马处理，不能延迟，相当于特事特办。不过，即使这20个突发请求立马处理结束，后续来了请求也不会立马处理。

burst=20 相当于缓存队列中占了20个坑，即使请求被处理了，这20个位置只能按 100ms一个来释放。这就达到了速率稳定，突发流量也能正常处理的效果。

4.1.3 控制并发连接数

Nginx 的ngx_http_limit_conn_module模块提供了对资源连接数进行限制的功能，使用 limit_conn_zone 和 limit_conn 两个指令就可以了。

```
http {
    limit_conn_zone $binary_remote_addr zone=perip:10m;
    limit_conn_zone $binary_name zone=perserver:10m;
    server {
        listen       80;
        server_name  localhost;
        location / {
            ...
            limit_conn perip 20;
            limit_conn perserver 100;
            proxy_pass http://targetserver;
        }
    }
}
```

- limit_conn perip 20：对应的key是 $binary_remote_addr，表示限制单个IP同时最多能持有20个连接。

- limit_conn perserver 100：对应的key是 $server_name，表示虚拟主机(server) 同时能处理并发连接的总数。

  注意，只有当 **request header** 被后端**server**处理后，这个连接才进行计数。

4.2 后端限流

两种比较经典的算法可以实现：**令牌桶**和**漏桶**

令牌桶

> 在后台的技术中可以在网关中使用令牌桶算法，详细实现请查看资料文件夹中的：网关限流

5.对于库存的扣减，库存超卖问题是如何解决的？

方案一、加分布式锁

方案二、mysql锁机制，悲观锁，InnoDB行级锁方案

```
select * from goods where inventory >= 1 for update;
```

update件件增加验证购买数量条件 AND inventory >=1

方案三、mysql乐观锁

添加版本号Version

方案四、redis的decr

原子操作，直接让redis抗下所有的并发，秒杀过程中不连接数据库同步

6.订单失败和订单未支付怎么办

延迟队列

- redis使用zset
- rabbitmq实现延迟队列

三、场景问题

1.分布式服务的接口幂等性如何设计？

1.1 概述

所谓幂等: **多次调用方法或者接口不会改变业务状态，可以保证重复调用的结果和单次调用的结果一致。**

基于RESTful API的角度对部分常见类型请求的幂等性特点进行分析

请求方式	说明
GET	查询操作，天然幂等
POST	新增操作，请求一次与请求多次造成的结果不同，不是幂等的
PUT	更新操作，如果是以绝对值更新，则是幂等的。如果是通过增量的方式更新，则不是幂等的
DELETE	删除操作，根据唯一值删除，则是幂等的；如果是根据条件删除，则不一定的幂等。例如每次删除某字段最大的记录

1.2 需要幂等的场景

1.2.1 网络波动

因网络波动，可能会引起重复请求

1.2.2 MQ消息重复

生产者已把消息发送给MQ，在MQ给生产者返回ack的时候网络中断，故生产者未收到确定消息，生产者认为消息未发送成功。但实际情况是，MQ已成功接收到了消息，在网络重连后，生产者会重新发送刚才的消息，造成MQ接收了重复的消息。

1.2.3 用户重复点击

用户在使用产品时，可能会误操作而触发多笔交易，或因为长时间没有响应，而有意触发多笔交易。

1.2.4 应用使用失败或超时重试机制

为了考虑系统业务稳定性，开发人员一般设计系统时，会考虑失败了如何进行下一步操作或等待一定时间继续前端的动作的。

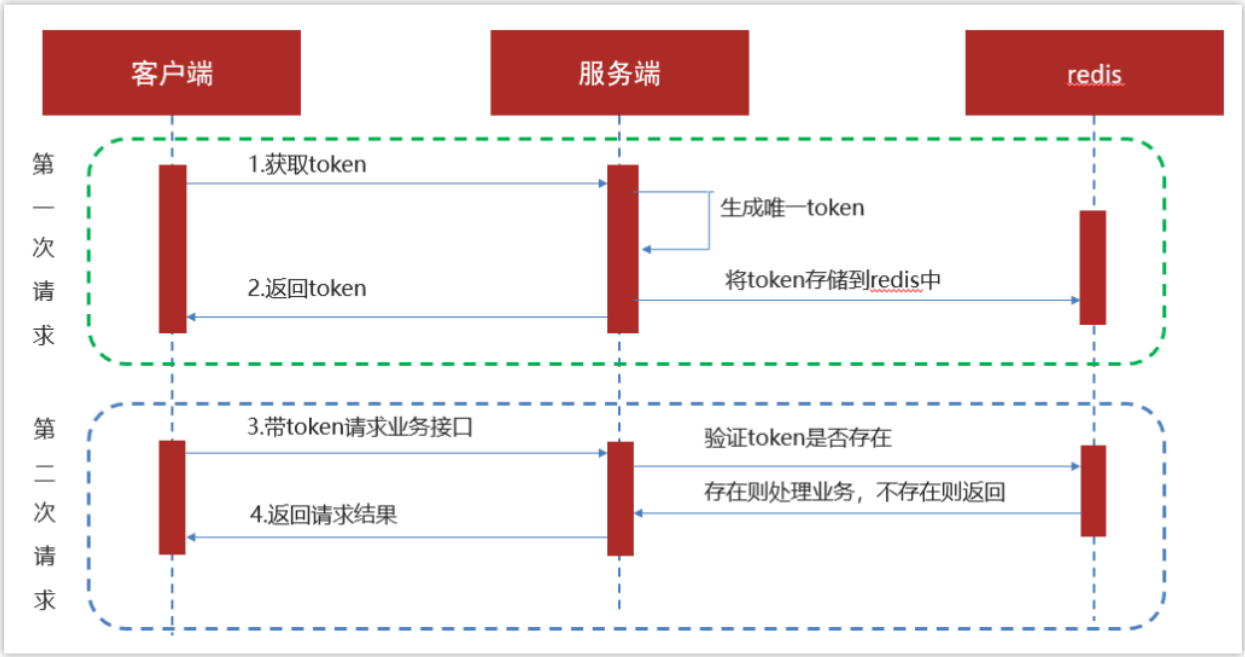
1.3 后端解决方案

1.3.1 数据库唯一索引

使用数据库提供的唯一索引来保证数据重复插入，避免脏数据产生

解决场景：新增

1.3.2 token+redis



- 第一次请求
  - 在后端生成一个唯一的token(比如：key:userid,value:UUID)
  - 将token存储到redis中
  - 将token返回前端
- 第二次请求

- 在真正处理业务的时候需要携带过来之前的token
- 到redis中查询token是否存在
- 如果存在，则正常处理业务，同时删除redis中的token
- 如果不存在，则操作失败

解决场景：新增、删除、修改

1.3.3 分布式锁

在分布式锁使用的时候，要注意粒度

在操作数据时，先添加一个分布式锁，当操作完成后再释放掉这把锁，同时在操作过程中，如果有人来抢锁，应当抛出异常，即：

```
if (!lock) {
        log.info("操作作者信息获取锁失败, operator:
{}",request.getOperator());
            throw new BaseBizException("新增/修改失败");
}
```

操作完成后，释放掉锁，因为幂等问题，通常是一个请求快速过来两次或者多次，所以在释放锁之前让后来的同一个用户的请求，直接失败即可，保证当前方法在短时间之内只能被执行一次，切记控制锁的粒度。

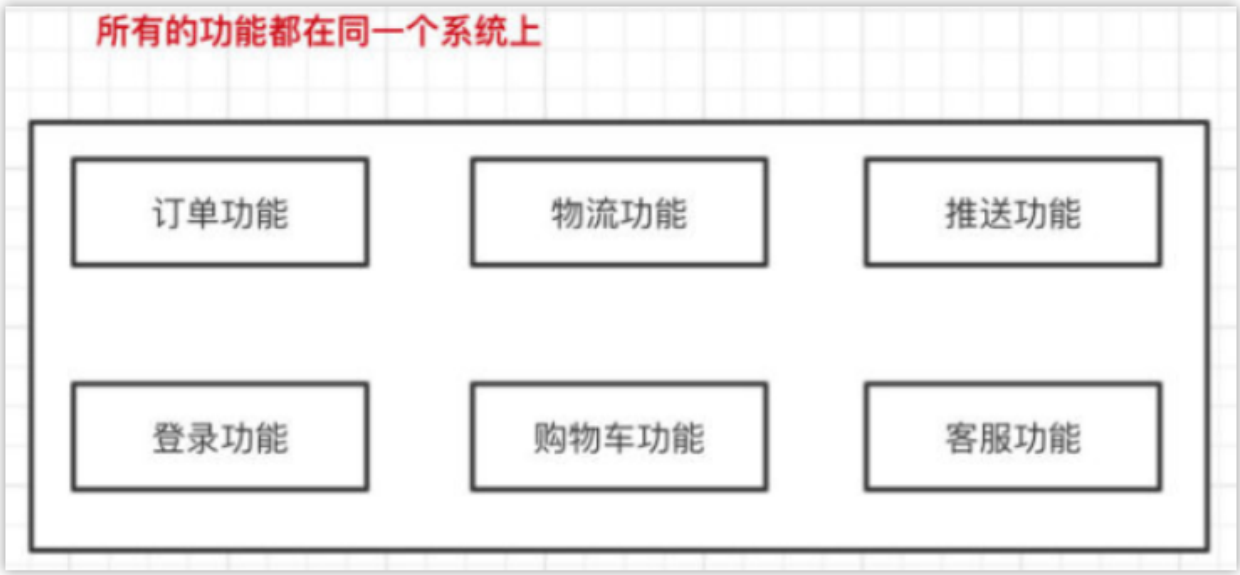
```
 public SaveOrUpdateUserDTO saveOrUpdateUser(SaveOrUpdateUserRequest
request) {
        // 加入用户，要先取得一把分布式锁，针对的是操作人
        // 同一个操作人，同时间只能新增用户，避免说重复请求短时间内发生，
数据重复灌入
        // 加分布式锁
        String userUpdateLockKey =
RedisKeyConstants.USER_UPDATE_LOCK_PREFIX + request.getOperator();
        boolean lock = redisLock.lock(userUpdateLockKey);

        if (!lock) {
            log.info("操作作者信息获取锁失败, operator:{}",
request.getOperator());
            throw new BaseBizException("新增/修改失败");
        }
        //忽略代码
        } finally {
```

```
            redisLock.unlock(userUpdateLockKey);
        }
    }
```

2.单点登录这块怎么实现的

单点登录的英文名叫做：Single Sign On（简称**SSO**）。

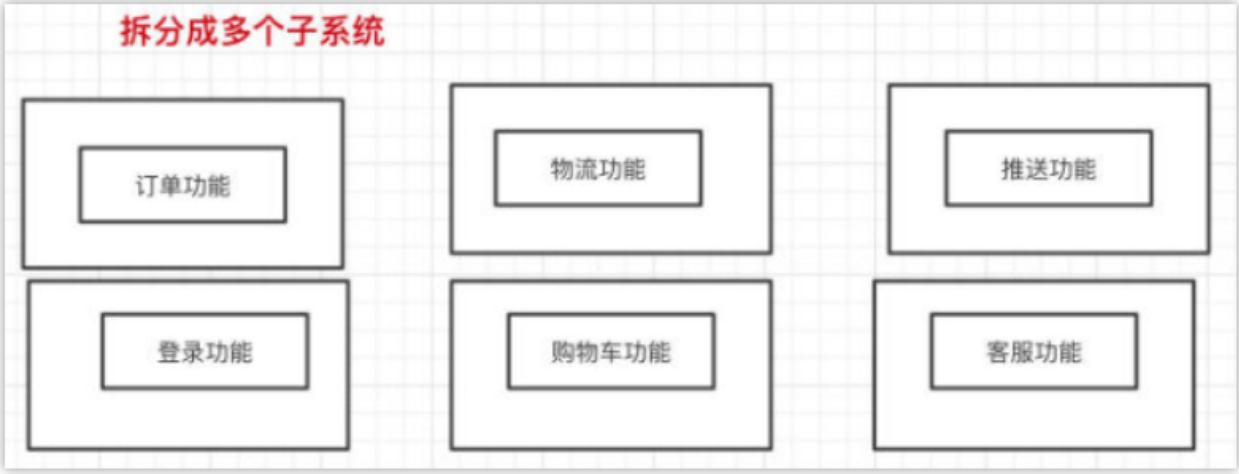
在**以前**的时候，一般我们就**单系统**，所有的功能都在同一个系统上。



单体系统的session共享

- **登录**：将用户信息保存在Session对象中
- - 如果在Session对象中能查到，说明已经登录
  - 如果在Session对象中查不到，说明没登录（或者已经退出了登录）
- **注销（退出登录）**：从Session中删除用户的信息

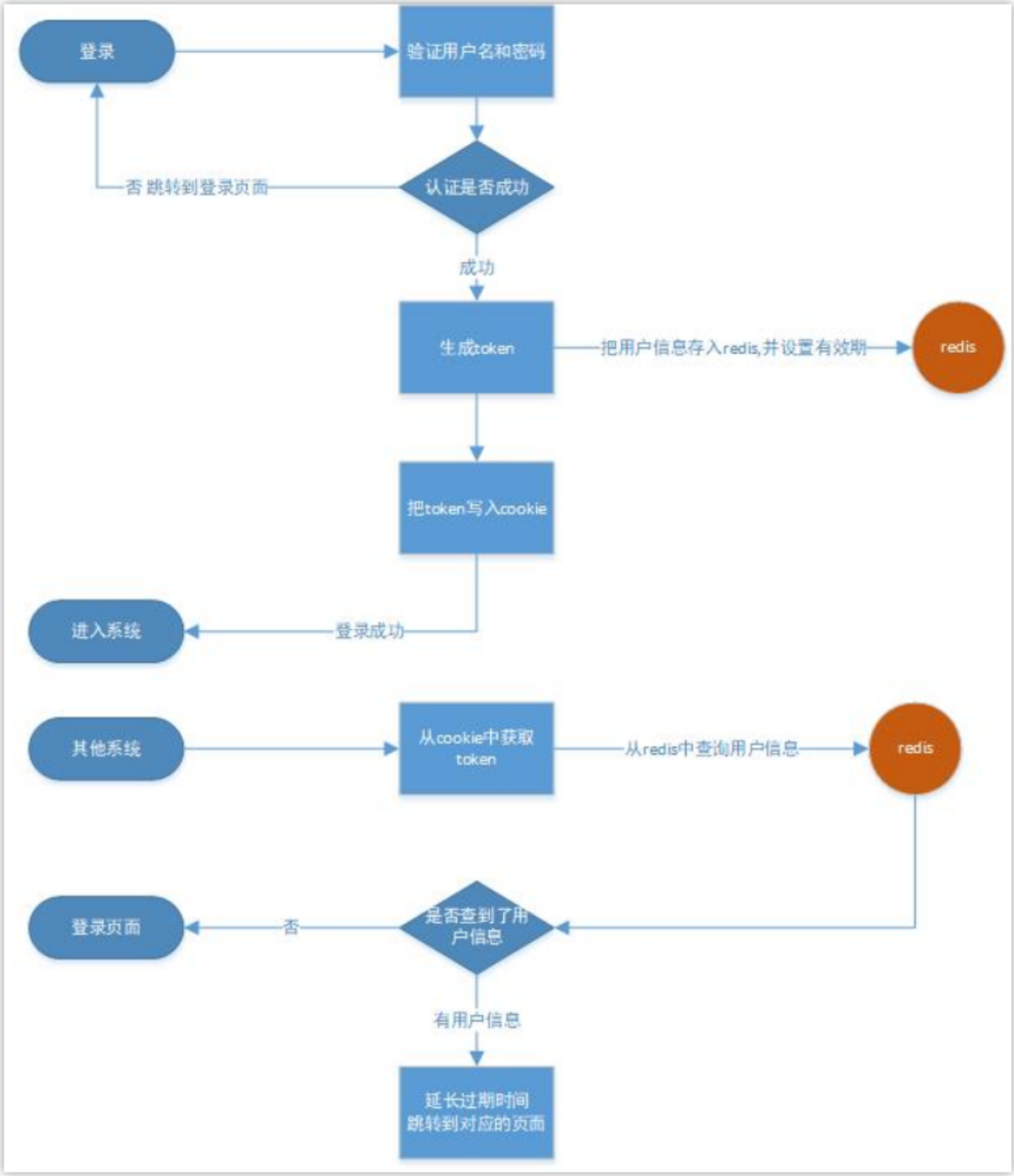
后来，我们为了**合理利用资源和降低耦合性**，于是把单系统**拆分**成多个子系统。



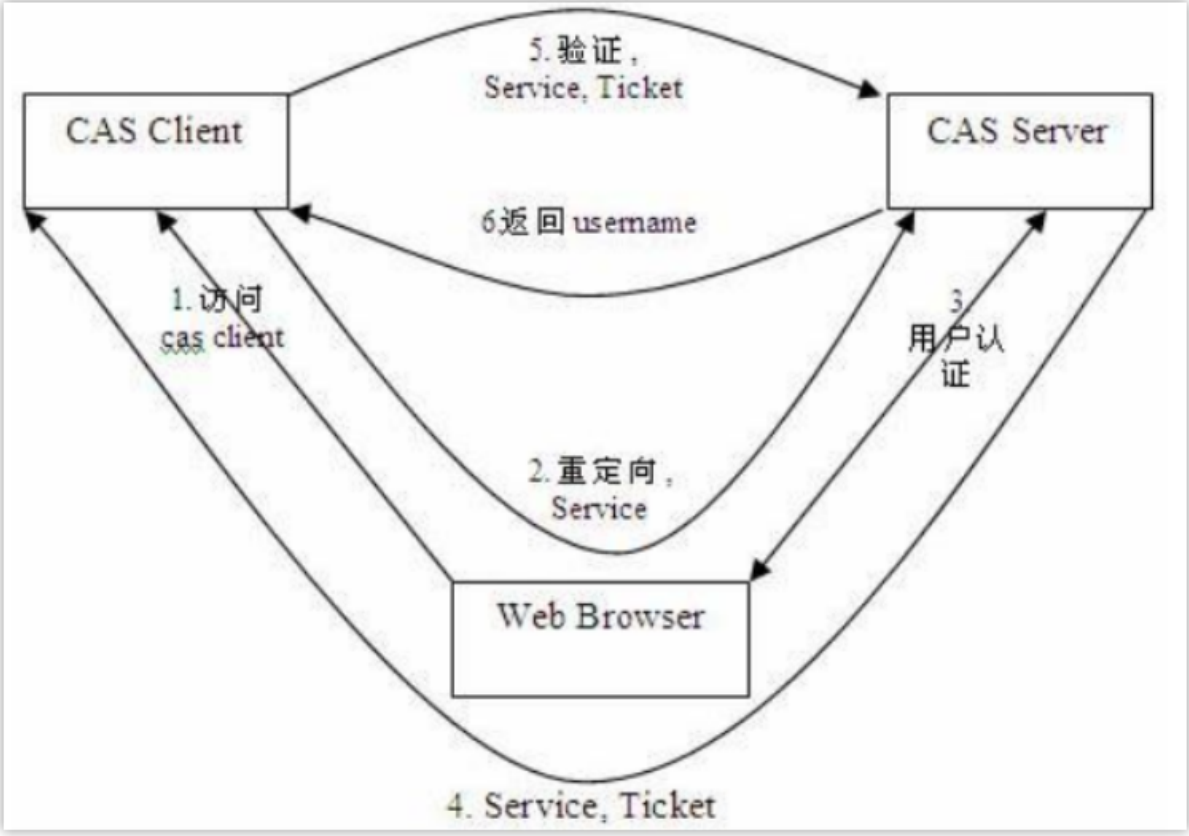
多系统即可能有多个Tomcat，而Session是依赖当前系统的Tomcat，所以系统A的Session和系统B的Session是不共享的。

解决系统之间Session不共享问题有一下几种方案：

- Tomcat集群Session全局复制（集群内每个tomcat的session完全同步）
- 把Session数据放在Redis中（使用Redis模拟Session）自己实现


登录
验证用户名和密码
否 跳转到登录页面
认证是否成功
成功
生成token
把用户信息存入redis,并设置有效期
redis
把token写入cookie
进入系统
登录成功
其他系统
从cookie中获取token
从redis中查询用户信息
redis
登录页面
否
是否查到了用户信息
有用户信息
延长过期时间
跳转到对应的页面

- CAS实现单点登录

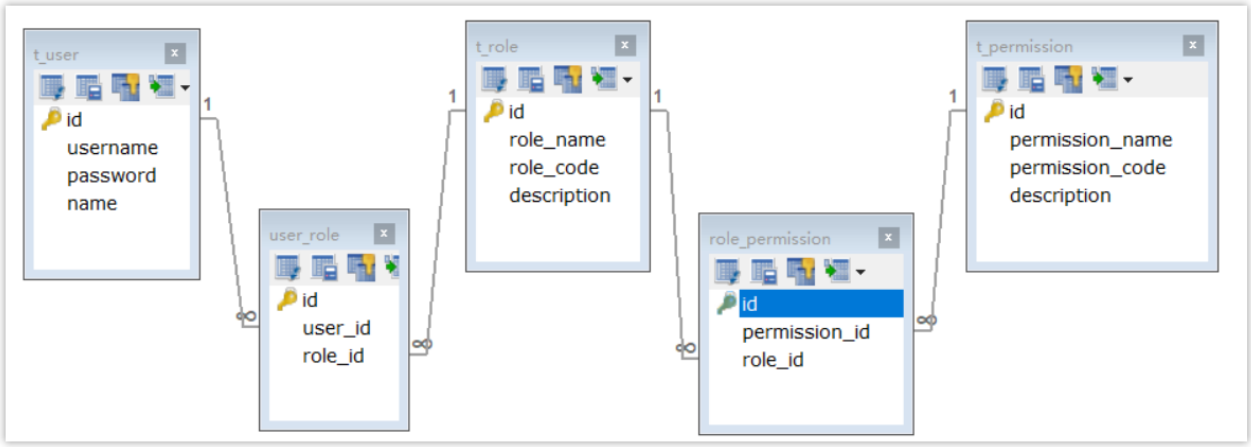


SSO 单点登录访问流程主要有以下步骤：

1. 访问服务：SSO 客户端发送请求访问应用系统提供的服务资源。
2. 定向认证：SSO 客户端会重定向用户请求到 SSO 服务器。
3. 用户认证：用户身份认证。
4. 发放票据：SSO 服务器会产生一个随机的 Service Ticket。
5. 验证票据：SSO 服务器验证票据 Service Ticket 的合法性，验证通过后，允许客户端访问
   服务。
6. 传输用户信息：SSO 服务器验证票据通过后，传输用户认证结果信息给客户端。

3.权限认证是如何实现的

五张表：用户、角色、权限



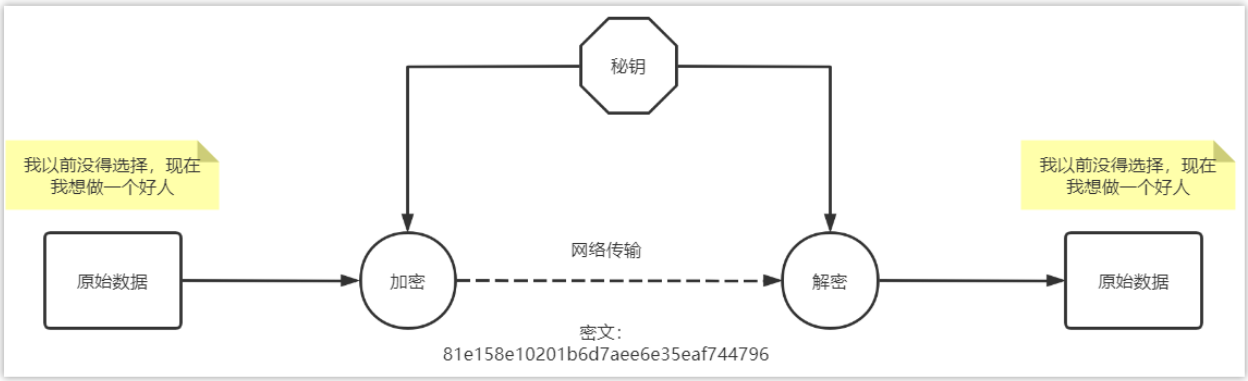
框架：shiro 、 spring security

> 你是谁？
>
> 你要做什么？

4.上传数据的安全性你们怎么控制？

4.1 对称加密

文件加密和解密使用相同的密钥，即加密密钥也可以用作解密密钥



解释: 在对称加密算法中，数据发信方将明文和加密密钥一起经过特殊的加密算法处理后，使其变成复杂的加密密文发送出去，收信方收到密文后，若想解读出原文，则需要使用加密时用的密钥以及相同加密算法的逆算法对密文进行解密，才能使其回复成可读明文。在对称加密算法中，使用的密钥只有一个，收发双方都使用这个密钥，这就需要解密方事先知道加密密钥。

优点: 对称加密算法的优点是算法公开、计算量小、加密速度快、加密效率高。

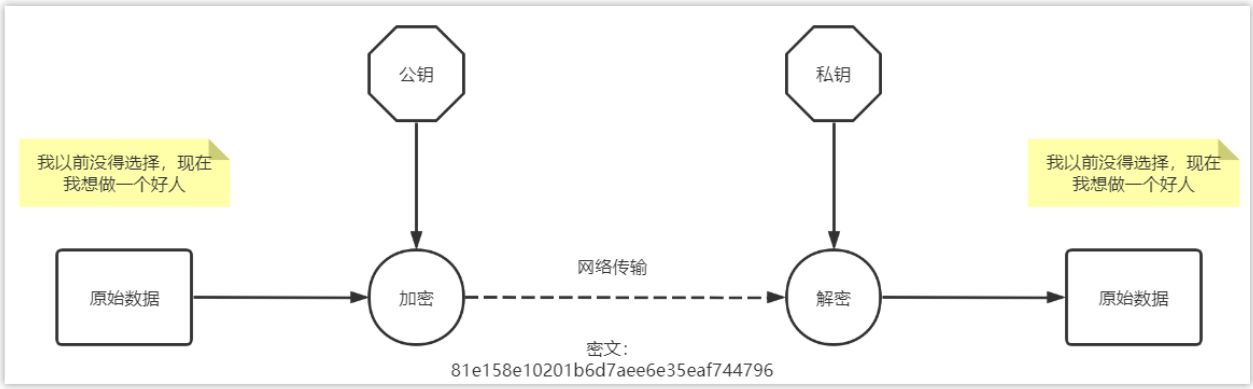
缺点: 没有非对称加密安全.

用途： 一般用于保存用户手机号、身份证等敏感但能解密的信息。

常见的对称加密算法有: `AES、DES、3DES、Blowfish、IDEA、RC4、RC5、RC6、HS256`

4.2 非对称加密

两个密钥：公开密钥（publickey）和私有密钥，公有密钥加密，私有密钥解密



解释: 同时生成两把密钥：私钥和公钥，私钥隐秘保存，公钥可以下发给信任客户端.

加密与解密:

- 私钥加密，持有公钥才可以解密
- 公钥加密，持有私钥才可解密

签名:

- 私钥签名, 持有公钥进行验证是否被篡改过.

优点: 非对称加密与对称加密相比，其安全性更好；

缺点: 非对称加密的缺点是加密和解密花费时间长、速度慢，只适合对少量数据进行加密。

用途： 一般用于签名和认证。私钥服务器保存, 用来加密, 公钥客户拿着用于对于令牌或者签名的解密或者校验使用.

常见的非对称加密算法有： RSA、DSA（数字签名用）、ECC（移动设备用）、RS256 (采用SHA-256 的 RSA 签名)

面试题：上传数据的安全性你们怎么控制？

使用非对称加密（或对称加密），给前端一个公钥让他把数据加密后传到后台，后台负责解密后处理数据

5.你负责项目的时候遇到了哪些比较棘手的问题？怎么解决的？

这个问题考察的点是想看下你是否真的做过开发,如果你的简历中写了2-3年的工作经验那么你肯定是遇到过一些事情的,如果你一个都没遇到过那么可能你的简历就存在水分了或者你开发的全部是没有技术含量的功能,同时如果你遇到过看下你遇到的事情深度怎么样然后如何去解决的,从侧面看下你在公司开发的功能水平怎么样从而推测你的个人能力以及在上一家公司的地位水平。

要像解决这个问题大家可以准备一个jvm的或者mysql优化的或者用设计模式去优化复杂系统的场景去回答。

> 内存溢出
>
> CPU飙高
>
> mysql调优
>
> 使用策略模式+工厂模式优化代码

6.你们项目中日志怎么采集的？

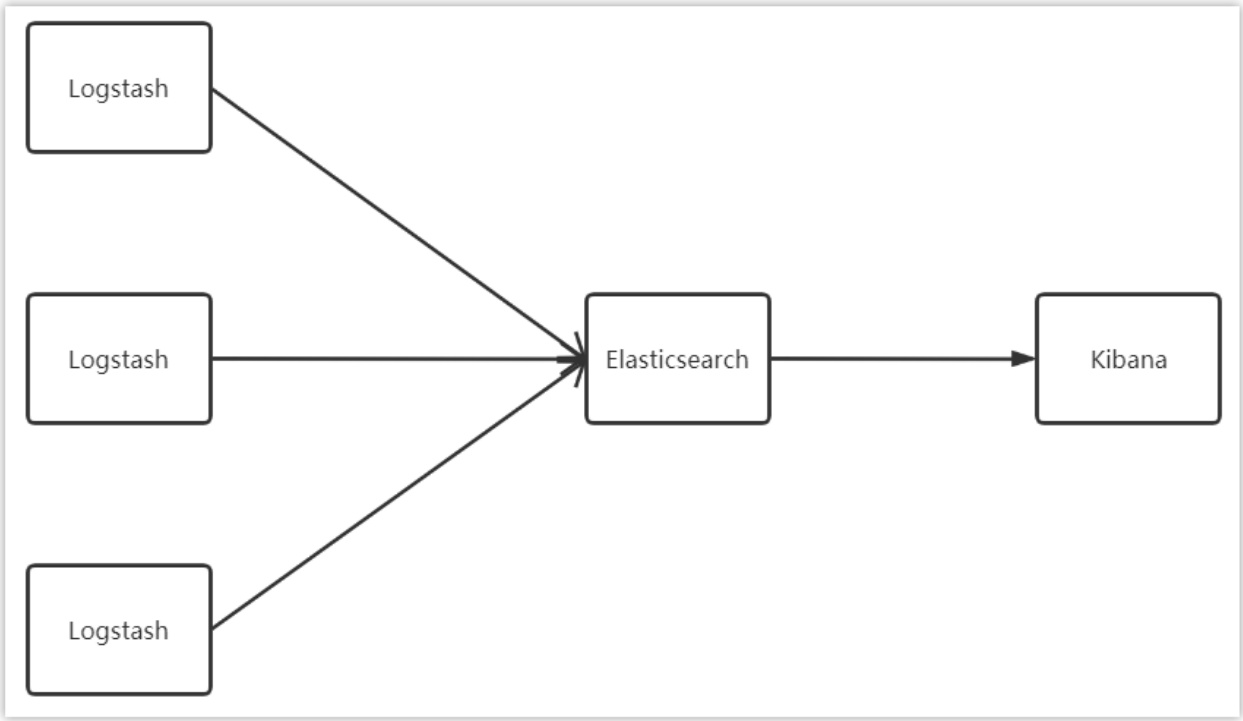
ELK：即Elasticsearch、Logstash和Kibana三个开源软件的缩写

1、Elasticsearch
Elasticsearch 全文搜索和分析引擎，对大容量的数据进行接近实时的存储、搜索和分析操作。

2、Logstash
Logstash是一个数据收集引擎，它可以动态的从各种数据源搜集数据，并对数据进行过滤、分析和统一格式等操作，并将输出结果存储到指定位置上

3、Kibana

Kibana是一个数据分析和可视化平台，通常与Elasticsearch配合使用，用于对其中的数据进行搜索、分析，并且以统计图标的形式展示。



7.你们的app用户量有多少？你们项目的的qps是多少、有多少台服务器？

我们app端用户量，目前是10万，经过测试，最高的并发集中在晚上7点至9点，其中有查询文章的接口最高有几次达到了接近2000的qps

目前生产的服务器使用的tomcat9，使用jmeter压测后，处理的并发数极限为400左右，所以文章那个微服务通常都是6、7台服务器做了集群。其他访问量较少的集群数量更低一些。

8.说说你们系统生产部署情况

要结合的访问项目的QPS，压测后的结果来计算部署多少台服务器

9.你们是怎么做压测(性能测试)的？

一般压测由测试人员进行测试，由后台程序员协助。

1.将线下配置与线上配置保持一致；

2.编写压测方案（包括背景、接口信息、压测场景、压测前准备、压测记录、压测结果分析）；

3.编写压测脚本-设置jmeter参数【线程数、常数吞吐量计时器、header、http请求、响应断言、聚合报告】开始运行；

4.查看聚合报告，看错误率

5.不通过，则进行排查问题：

- 1.查看cpu、内存是否达到瓶颈
- 2.查看数据库连接数、cpu、内存等是否达到瓶颈
- 3.通过skywalking来排查耗时较高的接口，以进行优化

6.经过优化或对服务资源调整，使达到压测通过标准

- 错误率为0的情况下，90%或95%的接口的响应时间小于500ms（看公司要求）
- 错误率为0的情况下，个别接口（业务复杂，调用链较长）的响应时间可以超过1s

7.编写压测报告，进行风险分析

10.生产问题怎么排查？

1，先分析日志，通常在业务中都会有日志的记录，或者查看系统日志，或者查看日志文件，然后定位问题，解决问题

2，如果问题较为复杂，情况就可能会有多种，可能是代码的问题，也有可能是数据的问题

3，远程debug

> 特别注意，通常公司的正式环境（生产环境）是不允许远程debug的。
>
> 一般远程debug都是公司的测试环境，方便调试代码

实现步骤：

前提条件：**远程的代码和本地的代码要保持一致**

①：远程代码需要配置启动参数：

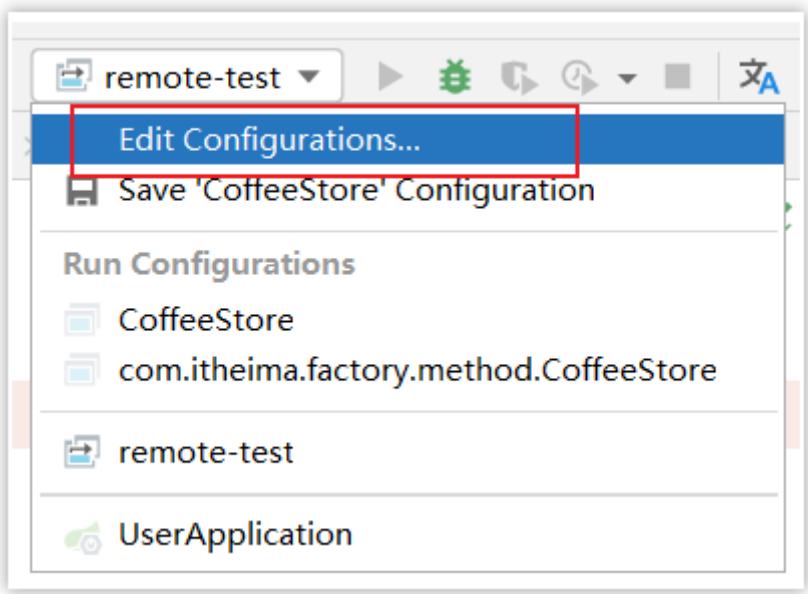
把项目打包放到服务器后启动项目的参数：

```
java -jar -
agentlib:jdwp=transport=dt_socket,server=y,suspend=n,address=5005
project-1.0-SNAPSHOT.jar
```

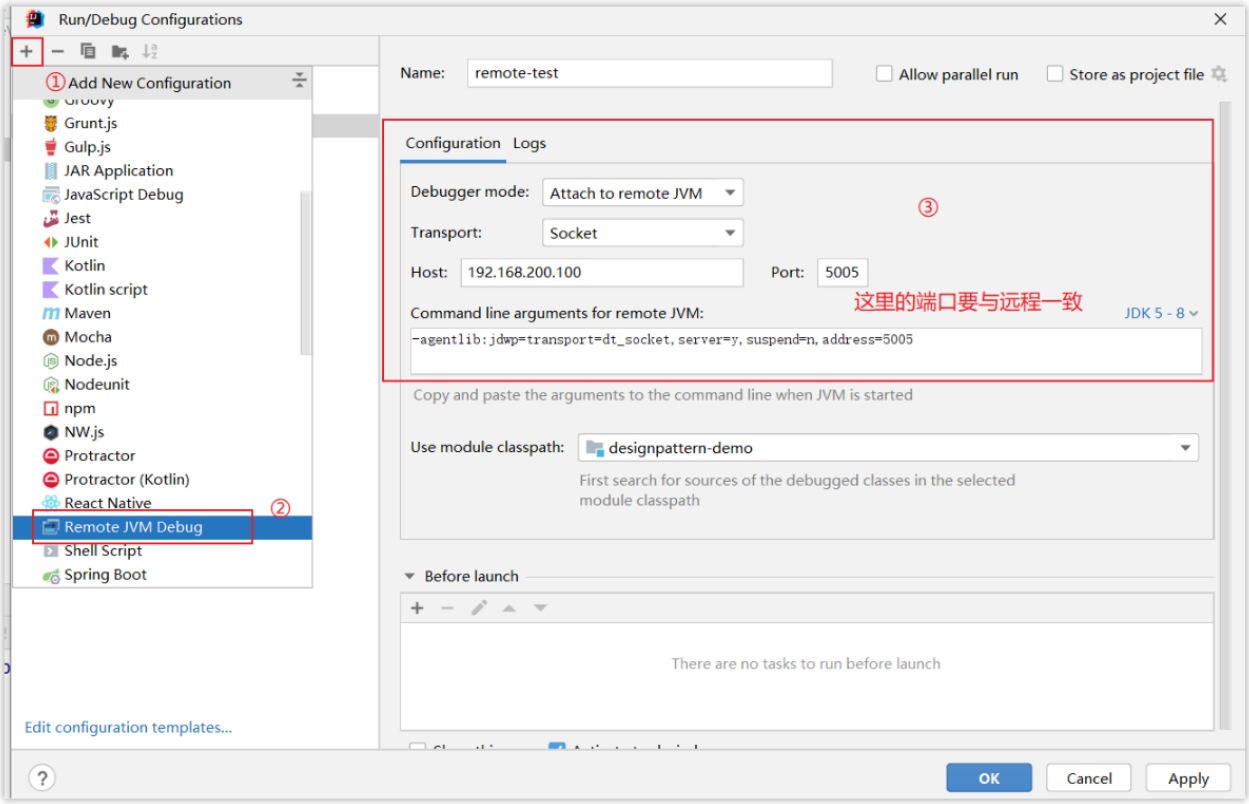
> **-agentlib:jdwp** 是通知JVM使用(java debug wire protocol)来运行调试环境
>
> **transport=dt_socket** 调试数据的传送方式
>
> **server=y** 参数是指是否支持在server模式
>
> **suspend=n** 是否在调试客户端建立起来后，再执行JVM。
>
> **address=5005** 调试端口设置为5005，其它端口也可以

②：idea中设置远程debug

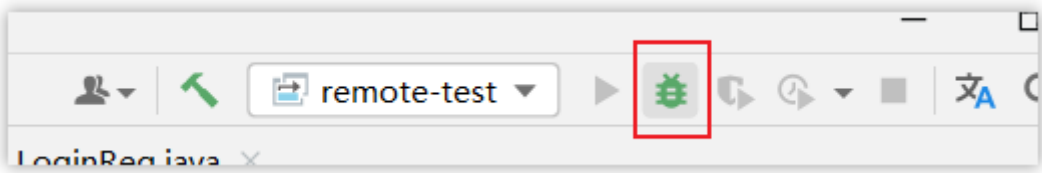
找到idea中的 Edit Configurations...



设置远程debug参数



③idea中启动远程debug



④在本地代码中打断点即可调试远程

11.查看日志的命令

(1)tomcat查看实时日志

- 实时监控日志：`tail -f catalina.out`
- 查询最后100行日志：`tail -n 100 -f catalina.out`

(2)doeker容器实时查看日志

- 实时监控日志：`docker logs -f  容器id/容器名称`
- 查询最后100行日志：`docker logs -n 100 -f  容器id/容器名称`

(3)查看日志文件

- 在test.log文件中搜索”exception”：`cat -n test.log | grep “exception”`

- 分页查看日志文件：`more test.log`
- 使用 >xxx.txt 将查询到的日志保存到文件中,可以下载这个文件分析

  `cat -n test.log |grep "debug" >debug.txt`

通常的使用思路：先尝试监控实时日志，看看能不能监控到想要的信息，如果不能则需要查看日志文件，从海量日志信息中找出自己想要的错误信息。

12.怎么快速定位系统的瓶颈？

Arthas（阿尔萨斯）

Arthas 是一款线上监控诊断产品，通过全局视角实时查看应用 load、内存、gc、线程的状态信息，并能在不修改应用代码的情况下，对业务问题进行诊断，包括查看方法调用的出入参、异常，监测方法执行耗时，类加载信息等，大大提升线上问题排查效率。

Arthas（阿尔萨斯）能为你做什么？

`Arthas` 是 Alibaba 开源的 Java 诊断工具，深受开发者喜爱。

当你遇到以下类似问题而束手无策时，`Arthas` 可以帮助你解决：

1. 这个类从哪个 jar 包加载的？为什么会报各种类相关的 Exception？
2. 我改的代码为什么没有执行到？难道是我没 commit？分支搞错了？
3. 遇到问题无法在线上 debug，难道只能通过加日志再重新发布吗？
4. 线上遇到某个用户的数据处理有问题，但线上同样无法 debug，线下无法重现！
5. 是否有一个全局视角来查看系统的运行状况？
6. 有什么办法可以监控到 JVM 的实时运行状态？
7. 怎么快速定位应用的热点，生成火焰图？
8. 怎样直接从 JVM 内查找某个类的实例？

`Arthas` 支持 JDK 6+，支持 Linux/Mac/Windows，采用命令行交互模式，同时提供丰富的 `Tab` 自动补全功能，进一步方便进行问题的定位和诊断。

官网：**https://arthas.aliyun.com/doc/**

13.全链路日志怎么做的？

skywalking

14.你们开发业务的流程是什么？

1，产品经理说需求--》后端+前端+测试

2，定接口（后端主导 | 前端主导 | **前后端协调**）

- mock测试

3，后端的工作：设计（技术选型，数据库表[原型或PRD]）+ 编码；前端 开发页面

- 基于原型或PRD(需求文档)
- 命名规范（参考阿里规约）
- 字段类型（合适类型--->根据存储的内容决定）
- 实体与实体之间的关系（主外键约束）
- 表设计没有不变的

4，部署测试环境

5，接口联调 --->前后端联调

6，测试 --> 提bug 专门的bug管理工具(禅道)

每个人都会开通一个禅道账号　个人修复bug以后，提交为 已解决———》 测试人员回归测试

7，部署正式环境

15.你们项目的bug是如何管理的？

- 禅道

  开发团队使用说明：https://www.zentao.net/book/zentaopmshelp/159.html

- **TAPD**

  https://www.tapd.cn/help/show#1120003271001000096

16.你们项目的开发周期？

参考回答：

我们的项目开发周期大概7个月左右，平时后台有4个人一块开发，中间偶尔也会有人员变动，每个人负责不同的模块进行开发。

> 分布式6-8个月
>
> 传统项目4-6个月

17.如何回答自己不会的问题？

方案1：

让面试官再重复一下问题？看看能不能从中找到关联的点

话术：不好意思，我刚才没听太明白您的意思，麻烦您再说一遍！！

如果能找到提示，则可以继续聊下去

方案2：

情况一：这个问题，你知道一点点，没用过或者已经忘的差不多了。

话术：不好意思，这个技术点，我之前在项目中没用过，但是我自己私下学习过一些。我讲一下我的理解。

情况二：这个问题，完全没听过

参考回答1：坦诚一些，告诉面试官这个问题不会。

> 不要奢望回答出所有的问题

参考回答2：您这个问题，我一时想不到答案。不过，我好奇的是，什么地方用到这项 技术呢？（慎用）