# 第一章 面向对象

Object-Oriented是一种程序设计方法，从现实世界中客观存在的事物出发来构造软件系统，尽可能运用人类的自然思维方式。

基本思想：使用对象、类、继承、封装、多态等基本概念来进行程序设计。

**OO VS Procedure-oriented**

都是模块化编程，Procedure-oriented把问题分割成**以功能函数为单位**的模块中，

Object-Oriented把问题抽象描述成**以对象为单位**的模块中。

## 1.1 理解面向对象

面向对象的不足

[Reference](https://mp.weixin.qq.com/s?__biz=MzA4NjExMjE2OQ==&mid=2650118787&idx=2&sn=decc1e219b637eee41e79188263babf3&chksm=87cc9845b0bb1153d5d59397bcf273677fe546c6daef82e8af66346ced0d4e6e81996d53609f&mpshare=1&scene=1&srcid=#rd])

## 1.2 三个基本特性

封装

客观事物封装成抽象的类，更符合人的自然思维。

封装数据结构和函数，隐藏细节(private/public),避免被外界破坏。

继承

多态

多态是IOC、模板模式实现的关键。

方法的动态绑定，实现运行时的类型决定对象的行为。多态的表现形式是父类指针或引用指向子类对象，在这个指针上调用的方法使用子类的实现版本。

C++多态是通过虚函数来实现的。

TODO4: invokevirtual执行过程

[Reference](https://blog.csdn.net/qq_35614059/article/details/79141866)

四大好处

复用性，扩展性，维护性，灵活性。

为什么说面向接口编程是面向对象设计的精髓？

这四大好处都依赖于面向接口编程。

## 1.3 五大基本原则（SOLID）

[Reference](https://blog.csdn.net/dongnan591172113/article/details/51996421)

需求总是变化，所以就需要根据准则设计软件，变化满足需求的同时还能保持软件内部的封装体系稳定，不被需求的变化影响。且使我们的代码稳健，易于复用，易于拓展，灵活优雅。

设计准则 VS 设计模式

不同的设计模式对应不同的需求，而设计原则代表永恒的灵魂，是更高层次的抽象。

### 3.1.4.1 单一职责

Single Responseibility Principle(S)。一个类(模块)，一个功能, 不做多职能接口。职责过多，可能引起它变化的因素就越多，增加系统错误风险。

好处：解耦和增强内聚性。

### 3.1.4.2 开放封闭原则

Open Closed Principle

核心思想：对修改封闭的，对扩展开放。

**具体实现**

通过继承和多态机制，重写/重载方法来改变固有行为，实现拓展，所以就是开放的。

对抽象编程，不对具体编程。因为抽象相对稳定，类依赖于固定的抽象，所以修改是封闭的。

### 3.1.4.4 替换原则

Liskov Substitution Principle

子类可以扩展父类的功能，但不能改变父类原有的功能。

继承作为面向对象三大特性之一，在给程序设计带来巨大便利的同时，也带来了弊端。比如使用继承会给程序带来侵入性，程序的可移植性降低，增加了对象间的耦合性（这里是父子类之间的耦合），如果一个类被其他的类所继承，则当这个类需要修改时，必须考虑到所有的子类，并且父类修改后，所有涉及到子类的功能都有可能会产生故障。

好处：保证继承复用是可靠地，违反必然导致违反开放封闭原则。

实现：面向接口编程

### 3.1.4.3 接口隔离

Interface Segregation Principle

核心思想

使用多个小的专门的接口，而不使用一个大的总接口。接口应该是内聚的，避免“胖”接口。

一个类对另外一个类的依赖应该建立在最小的接口上，不要强迫依赖不用的方法，这是一种接口污染。

胖接口存在明显的弊端，会导致实现的类型必须完全实现接口的所有方法、属性等；而某些时候，实现类型并非需要所有的接口定义，而且对胖接口的修改将导致一连串的客户端程序需要修改

具体实现

分离的手段主要有以下两种：

1、委托分离，通过增加一个新的类型来委托客户的请求，隔离客户和接口的直接依赖，但是会增加系统的开销。

2、多重继承分离，通过接口多继承来实现客户的需求，这种方式是较好的。

分离后，每个类对应一个功能方法，将接口组装到类中，有个性化需求时，只要跟换新的实现类即可。

### 3.1.4.5 依赖倒置

Dependency Inversion Principle

1.High-level modules should not depend on low-level modules. Both should depend on abstractions.（上层模块不应该依赖底层模块，它们都应该依赖于抽象。）

2.Abstractions should not depend on details. Details should depend on abstractions.

面向接口编程

依赖倒置是通俗讲就是面向接口编程，即对抽象编程，不对具体编程。因为抽象相对稳定的可有效控制模块（类）耦合关系，提高系统的可维护性以及可扩展性。

**倒置**



（图一） （图二）

图1依赖关系为高层依赖底层

图2都依赖于抽象后，低层次模块依赖于抽象接口，被倒置。

# 第二章 代码优化

## 2.1 代码优化

在实际编程中，根据设计准则，总结如下几点：

* 对抽象编程，不对具体编程（面向接口）
* 开闭原则
* 封装变化
* 多用组合少用继承
* 使用继承时遵循里氏替换原则。

基本思想：尽可能的抽象，将变化控制到最小，做到高内聚、低耦合。

实现方法：使用对象、类、继承、封装、多态等基本概念来进行程序设计。

优化目标，什么样的代码是好代码？

答案：高内聚，低耦合。易替换，易扩展。

#### 低耦合&高内聚

模块独立性指每个模块只完成系统要求的独立子功能，并且与其他模块的联系最少且接口简单，两个定性的度量标准――耦合性和内聚性。

耦合性

指模块间相互联系紧密程度。为提高模块的独立性，模块间尽可能松散的耦合。

实现方法：依赖于抽象。

耦合性分类(低🡪高):

* 无直接耦合: 无调用关系，通过主模块的控制和调用产生联系。比如controller调用的service。

下面几种模块之间存在调用关系：

* 数据耦合: 传递简单的数据值，即值传递。
* 标记耦合: 传递的是数据结构，如数组名、类。
* 控制耦合: 传递的是控制变量（如开关、标志等）。
* 公共耦合: 访问同一个全局数据结构（包括单个变量），
* 内容耦合: 可能在汇编语言中出现。大多数高级语言都已设计成不允许出现。

1.高耦合度导致系统难维护，牵一发动全身，又会引入新的BUG。2.且不好拓展。

内聚性

模块内各元素（变量，语句、、、）联系的越紧密，则它的内聚性就越高。在模块划分时，遵循“一个模块，一个功能”的原则，尽可能使模块达到功能内聚。

内聚性分类(低🡪高)

* 偶然内聚: 模块内各元素之间没有任何联系。
* 逻辑内聚: 模块把几种相关的功能组合在一起，每次被调用时，由传送给模块参数来确定该模块应完成哪一种功能 。
* 时间内聚: 把需要同时执行的动作组合在一起。(无顺序)
* 过程内聚：必须以特定次序执行。（有顺序）
* 通信内聚:处理使用相同的输入数据或者产生相同的输出数据。
* 顺序内聚: 顺序+同一功能。前一功能元素输出就是下一功能元素的输入。（）
* 功能内聚: 顺序+同一功能 +不可再分

#### 解耦之面向接口编程

1. **public** **void** service() {
2. IBusiness business1 = Factory.create("A");
3. business1.doService();
4. }

依赖倒置准则：对抽象编程，不对具体编程。好处：有效控制模块（类）耦合关系。

依赖于抽象是面向对象设计的精髓，也是依赖倒置原则的核心。

#### 优化if-else

[Reference2](https://mp.weixin.qq.com/s?__biz=MzI3NzE0NjcwMg==&mid=2650123156&idx=1&sn=a917fde5e075223d610ac2a9e8e94656&chksm=f36bb6b5c41c3fa35dbe148b566c41f05cea7d5ce565b295aae96dfece70ecc09c4f360288ac&mpshare=1&scene=1&srcid=#rd])

分支嵌套层次过深可读性下降。维护时要改代码不符合开闭原则。场景比较少时用if-else合适。

**1）工厂模式**

从工厂里取实现类。

1. **public** **void** handleImpl(String type) {
2. TargetExecutor executor=Fatory.getExecutor(type);
3. executor.process();
4. }

工厂模式难免又引入分支，使用Map替代分支语句，对工厂模式进一步优化

1. **private** **void** init() {
2. map.put(TYPE\_LINK, Link.**class**);
3. map.put(TYPE\_IMAGE, Image.**class**);
4. }
6. **public** ShareItem createShareItem(**int** type) {
7. Class<? **extends** ShareItem> shareItemClass = map.get(type);
8. **return** shareItemClass.newInstance();
9. }

**2)接口分层**

1. **public** **void** handleImpl(Item item) {
2. **if** ("saved".equal(item.getStatus())) {
3. processSavedStatus();
4. }**else** **if**("finished".equal(item.getStatus())){
5. processFinishedStatus();
6. }
7. }

不同逻辑剥离到对应方法，看着清爽多了，判空同样可以放在外部接口只处理一次

1. **public** **void** handle(Item item) {
2. **if** (item == **null**) {
3. **return**;
4. }
5. handleImpl(item);
6. }

#### Java bean

成员变量类型用基本类型还是boxed类型？

看场景，尽量使用 unboxed 类型,需要使用对象的时候用 boxed 类型，尽可少的进行 boxed <==> unboxed 转换。

## 2.2 23种设计模式

分类

不同的设计模式对应具体的需求，好处从SOLID准则的角度来答。总体分为三类：

* 创建型：工厂方法、抽象工厂、单例模式、建造者模式、原型模式。
* 结构型：适配器、装饰器、代理模式、外观模式、桥接、组合模式、享元模式。
* 行为型：策略模式、模板方法、观察者模式、迭代子模式、责任链模式、命令模式、备忘录模式、状态模式、访问者模式、中介者模式、解释器模式

### 2.2.1 Creational Patterns

抽象实例化过程，将对象创建和使用解耦。

#### Builder(创建者)

Builder pattern was introduced to solve some of the problems with Factory and Abstract Factory design patterns when the Object contains a lot of attributes.

#### Prototype(原型)

创建重复的对象，而这些对象的创建是很复杂的，才需要通过克隆来生成一个对象。

ConcretePrototype

1. **public** **class** ConcretePrototype1 **extends** Prototype {
2. **public** Prototype clone() {
3. Prototype prototype = **new** ConcretePrototype1(**this**.getName());
4. **return** prototype;
5. }
6. }

Client

1. **public** **static** **void** main(String[] args) {
2. Prototype prototype= **new** ConcretePrototype1("1");
3. Prototype Cloned= prototype.clone();
4. }

#### Singleton(单例)

[单例模式](http://blog.csdn.net/dongnan591172113/article/details/51027374)

非同步



延迟加载



Double Checked locking



* 1处volatile：new Singleton并非原子操作，instance为类变量，准备阶段



1.new给instance分配内存

2.invokespecial初始化对象

3.aload将instance指向分配的内存

JVM 的即时编译器中存在指令重排序的优化。最终的执行顺序可能是 1-2-3 也可能是 1-3-2

2处为什么判空?

防止阻塞Thread恢复后会多次new

#### 工厂模式

工厂模式将系统大量的对象创建工作封装到工厂类。

好处: 创建对象的变化部分封装到工厂类，将变化控制到最低，更符合开闭原则。

缺陷：调用的类和工厂有一定的耦合?

对象创建

* new：缺点：对象创建到处分散，如要更换，则需要大量修改代码。
* 工厂模式
* 依赖注入：进一步解耦，接口注入:

bookA = (IBook)Class.forName("BookA").newInstance();

当然也可以传接口，这种把对象的创建推迟了，也可能推给调用者。

1. **public** **void** service(IBusiness business) {
2. business.doService();
3. }

直接new对象是硬编码，几处可能引起变化

1.构造器参数修改，所有创建该类的代码都要修改。

2.修改接口实现需要到创建的地方修改，并不符合开闭原则。

简单工厂(Simple Factory)

缺陷：工厂类本身集中了所有实例的创建，不是高内聚的，不符合开闭原则。

GOF在《设计模式》一书中将工厂模式分为两类：工厂方法模式与抽象工厂模式。将简单工厂模式看为工厂方法模式的一种特例，两者归为一类。不属于23种之一。

##### Factory Method()

特点：每种对象有一个工厂。工厂方法模式在设计上完全符合开闭原则。当系统加入新产品，只要添加一个具体工厂和具体产品就可以了，系统的可扩展性也就变得非常好。

1. **public** **static** **void** main(String[] args) {
2. IBusiness productA = **new** FactoryA().create();
3. productA.doService();
4. IBusiness productB = **new** FactoryB().create();
5. productB.doService();
6. }

##### Abstract Factory(抽象工厂)

同工厂方法模式的。只是它的一个工厂可以创建属于多种产品。

### 2.2.2 Structural Patterns

关注类和对象的组合以获得新功能。

#### Adapter(适配器)

将一个类的转成另外一个接口，让原本因接口不兼容的两个类可协同工作。

别名为包装器(Wrapper)。

1.通过继承实现Adapter

1. **public** **class** Adapter1 **extends** Adaptee **implements** IAdapter{
2. **public** **void** newMethod() {
3. System.out.println("使用适配器");
4. **this**.oldMethod();
5. }
6. }

2.通过委让实现Adapter

1. **public** **class** Adapter2 {
2. **private** Adaptee adaptee;
3. **public** Adapter2(Adaptee adaptee) {**this**.adaptee = adaptee;}
4. **public** **void** newMethod() {
5. System.out.println("使用适配器");
6. **this**.adaptee.oldMethod();
7. }
8. }

#### Bridge(桥接)

#### Composite(组合)

对象组合成树以表示部分-整体层次结构。组合模式使得用户对单个对象和组合对象的使用具有一致性。

#### Decorator(装饰器)

动态地给对象添加额外的职责。就增加功能来说，相比生成子类更为灵活。

Decorator VS Adapter

两者的别名都叫包装模式(Wrapper)，都是起到包装一个类或对象的作用。

适配器是将一个接口转变成另一个接口，目的是通过改变接口来达到重复使用的目的。

装饰器是要保持原有的接口，增强原有对象的功能。

所以这两个模式设计的目的是不同的。

#### Facade(外观)

又叫门面模式。提供一个高层次的接口，隐藏系统的复杂性，使得子系统更易于使用。

使用场景

* 一个常见的设计，将系统划分为若干个子系统，以降低了系统的复杂性，同时引入一个外观对象，为子系统的访问提供了一个简单而单一的入口。
* mvc的Control可理解为一种外观模式，作为Facade对象与外界（接口）对接。
* 开放API，外界访问的接口即门面。

Facade本质就是封装调用，在现有接口之上创建包装器接口。

1. **public** **static** **void** main(String[] args) {
2. ShapeMaker shapeMaker = **new** ShapeMaker();
3. shapeMaker.drawCircle();
4. shapeMaker.drawRectangle();
5. shapeMaker.drawSquare();
6. }

ShapeMaker即Facade类

#### Flyweight(享元)

#### Proxy(代理)

不修改目标对象的功能前提下,对目标功能扩展。

##### 静态代理

代理类

1. **public** **class** Proxy **implements** IService{
2. **private** IService target;
3. **public** Proxy(IService target){**this**.target=target;}
4. **public** **void** doService() {
5. System.out.println("before");
6. target.doService();//执行目标对象的方法
7. System.out.println("after");
8. }
9. }

Main入口

1. **public** **class** App {
2. **public** **static** **void** main(String[] args) {
3. IService target = **new** UserService();
4. UserDaoProxy proxy = **new** UserDaoProxy(target);
5. proxy.doService();
6. }
7. }

缺点:代理类和委托类实现相同的接口，同时要实现相同的方法，出现大量的代码重复。

##### 动态代理

JDK动态代理

代理对象：不需要实现目标类接口，不和目标类耦合。

目标对象：静态代理和动态代理模式都要求目标对象实现接口。

原理

在运行期，通过反射机制创建一个实现了一组给定接口的新类

Cglib代理

目标对象可以一个单独的对象,不用任何的接口。

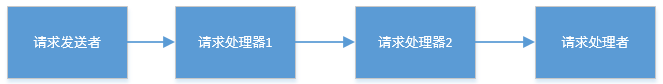
原理：构建目标对象子类实现代理。

### 2.2.3 Behavioral Patterns

行为模式，关注类之间的交流。

#### Chain of Responsibility(责任链)

解耦请求发送者与处理者，抽象出非核心的部分，以链式调用的方式对请求对象进行处理。



Case

1. **public** **void** study(PreparationList preparationList) {
2. **if** (preparationList.isWashHair()) {
3. System.out.println("洗脸");
4. }
5. **if** (preparationList.isWashHair()) {
6. System.out.println("洗头");
7. }
8. System.out.println("我可以去上学了！");
9. }

#### Command(命令)

#### Interpreter(解释器)

#### Iterator(迭代器)

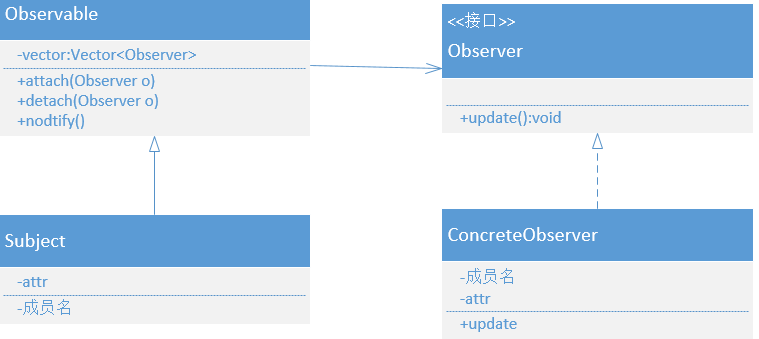
#### Mediator(中介者)

#### Memento(备忘录)

#### Observer(观察者)

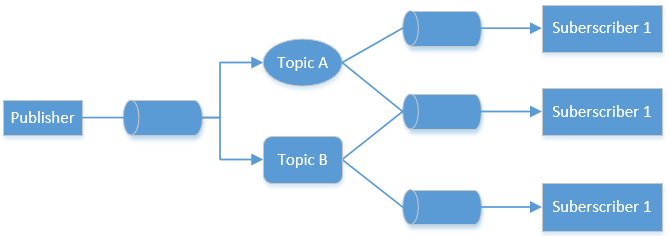
定义对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新。

别名：发布-订阅(Publish-Subscribe )模式。



发布订阅模式

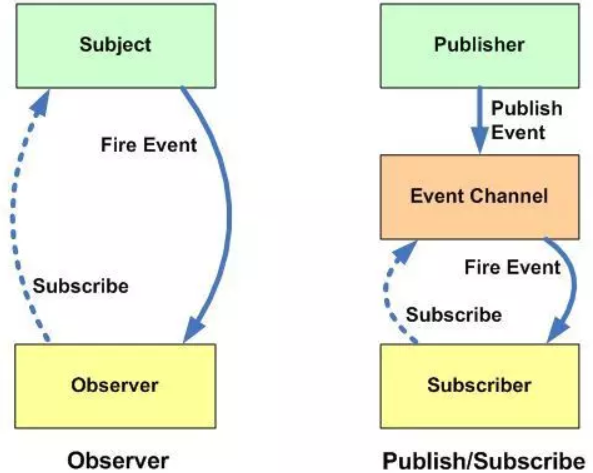
发布订阅模式是观察者模式进一步解耦。是观察者模式的变异。



区别

[Reference](https://juejin.im/post/5a14e9edf265da4312808d86)

观察者模式大多数时候是同步的，比如当事件触发，Subject就会去调用观察者的方法。而发布-订阅模式大多数时候是异步的（使用消息队列）。



观察者 VS 事件驱动

TODO3

#### State(状态)

#### Strategy(策略)

[Reference](http://www.runoob.com/design-pattern/strategy-pattern.html)

类的行为或算法可以更换。

策略模式包含如下角色：

* Context: 环境类
* Strategy: 抽象策略类
* ConcreteStrategy: 具体策略类

App

通过在context对象中更换strategy对象来改边算法.

1. **public** **static** **void** main(String[] args) {
2. Context context = **new** Context(**new** OperationAdd());
3. System.out.println("10 + 5 = " + context.executeStrategy(10, 5));
5. context = **new** Context(**new** OperationSubstract());
6. System.out.println("10 - 5 = " + context.executeStrategy(10, 5));
7. }

策略模式 VS 工厂模式

和工厂模式在结构上很相似，对象创建的控制权在调用者，工厂模式对象创建和工厂耦合

#### Template Method(模版方法)

定义一个操作中的算法的骨架，而将一些步骤延迟到子类中。

#### Visitor(访问者)

关注点分离

TODO https://www.cnblogs.com/asis/p/architecture-Soc.html

### 2.2.4 其他

#### MVC模式

用于应用程序的分层开发。

* Model：存取数据的对象或 JAVA POJO。它也可以带有逻辑，在数据变化时更新控制器。
* View：视图代表模型包含的数据的可视化。
* Controller：作用于模型和视图上，使视图与模型分离开。控制数据流向model，并在数据变化时更新视图。

### 设计模式实践

#### Spring中用了哪些设计模式?

BeanFactory

# 第三篇 架构设计

## 3.1 软件工程

### 软件类型

这个世界变化实在是快，各种新名词层出不穷，让人眼花缭乱（云，大数据，AR,AI）。其实很多东西只不过新瓶装老酒而已。

1）单机类型

最开始的那些不需要互联网的单机软件。

2）C/S

通过安装的客户端访问服务。早期多为局域网。



3）B/S

传统上软件开发主要针对单机环境,互联网的兴起，越来越多的人开始意识到，网站即软件，而且是一种新型的软件。浏览器/服务器架构。软件云端部署，用户通过通过浏览器来使用。



4）云计算

**服务模式**

SaaS（软件即服务）、PaaS（平台即服务）和IaaS（基础设施即服务）



SaaS

软件即服务，SaaS不是一种软件架构，而是一种软件销售方式。本质上还是采用B/S架构。

但通过对每个使用者收取年租费或月租费来销售

优势：云端办公和较低资费。

目标用户：中小企业用户，因为他们要求不高，不像大企业那般苛刻，通用版开发难度也小。

ERP型SaaS

**部署方式**

私有云、社区云、公有云和混合云

**基本特征**

按需自助服务：按需使用，按需付费

广泛的网络访问、资源共享、快速的可伸缩性和可度量的服务。

企业级云

### 企业级软件

面向的用户为企业，政府，业务比较复杂。按功能划分为财务会计、ERP（企业资源规划）、CRM（客户关系管理）、SCM（供应链管理）、HRM（人力资源管理）、BI（商务智能）、CMS（内容管理系统）和企业通信工具等。也可以按行业划分为制造业、零售业、医药业等解决方案。

ERP

Enterprise Resource Planning，是一种供应链管理思想。包括采购、销售、制造、财务等功能。

复杂度高

政府学校和企业的采购模式、利益诉求不尽相同；不同生产、管理环节对软件特性、服务模式等的需求大相径庭；不同企业对同种软件也总会有个性化的要求；同一个软件在同一个企业内的同一个功能，也可能因部门间的厉害纠葛，导致需求不可理喻地分裂。

国内现状

### 互联网公司

广义上来讲，企业规划、业务的开展都是基于互联网的前提之下施行的，都可以叫互联网公司。企业直接面向真实用户，即TO C。

特点：

追求高效，扁平化管理，去层级化，人本主义。

TODO3：和传统企业有什么本质区别？

2B VS 2C

2C，用户即购买者，更注重用户体验。2B，购买者为决策层，更注重提供生产和管理

国内现状：

用户有限，一天就是 24 小时，除去睡觉、工作，留给互联网的时间非常少，却有成千上万的产品在争夺这少得可怜的时间。而 B 端用户在工作时间使用产品，粘性大，高频刚需，还有可观的服务费收入。

### 开发模式

瀑布模型

将功能的实现与设计分开，将软件分为六个基本活动，并且规定了它们自上而下、相互衔接的固定次序，如同瀑布流水，逐级下落

生命周期：制定计划、需求分析、软件设计、程序编写、软件测试和运行维护

缺点：项目各个阶段之间极少有反馈，不适应用户需求的变化

敏捷开发

敏捷同样需要大量的需求分析，改进瀑布模型，把产品开发引向了小步快速迭代，周期交付。

缺点：不用将业务方方面面考虑周到即可开发，业务模型的频繁变更带来更高的维护成本。

DDD(领域驱动开发)

Domain Driven Design ,DDD is about designing software based on models of the underlying domain” — Martin Fowler

### GPL协议

## 3.2 架构分析

软件架构优劣的衡量要素：

1.性能 2.可用性 3.伸缩性 4.扩展性 5.安全性

(对前几章总结)

### 3.2.1.1 性能

#### 性能指标

* Concurrency：并发数。能同时处理的用户数。高并发
* Response：响应时间。发出请求到收到相应数据的时间
* 吞吐量：系统单位处理的请求数。QPS（query per second）
* 性能计数器：包括System Load、对象和线程数、内存、CPU、磁盘、网络IO

PV：page view/浏览量/点击量。默认指24小时内。同一user浏览同一页面不重复计算PV量。

UV：unique visitor，独立IP访问量。

TODO Nginx 统计PV [Reference](https://www.cnblogs.com/lianzhilei/p/6018421.html)

如何统计实际的QPS？

#### 性能测试方法

1. 性能测试：性能预期
2. 负荷测试：找到性能安全临界线
3. 压力测试：超过安全符合继续施压，获得最大压力承受能力

### 3.3.4 可用性

Reads and writes always succeed，即服务一直可用，而且是正常响应时间。高可用的架构在系统中任何一台或多台服务器宕机及其他问题，系统依然可用。

不可用原因：普通商用服务器+服务器数量大=宕机概率大

度量指标

|  |  |  |
| --- | --- | --- |
| Avalilability | 宕机时间（year） | Score |
| 99% | 88h | 基本可用 |
| 99.9%(3个9) | 8.76h | 较高可用 |
| 99.99%(4个9) | 52min | 有自动恢复能力的高可用 |
| 5个9 |  | 极高可用 |

计算公式 [Reference](https://mp.weixin.qq.com/s?__biz=MzI1NDQ3MjQxNA==&mid=2247487977&idx=1&sn=f893a169f8232bf08035ac1ed260a089&chksm=e9c5e858deb2614e9c94f2c2d0e0da9b99fd76f98ad5395f12430c427bb36b7633516a2fbf81&mpshare=1&scene=1&srcid=#rd])

#### 提升可用性

1）冗余

* Application Server: 多台服务器通过负载均衡组成集群，服务失效转移
* 存储服务: 数据实时冗余备份，宕机时数据访问转移到可用Server，并进行数据恢复。

2）减少故障上线

预发布验证 自动化测试 自动化发布 灰度发布

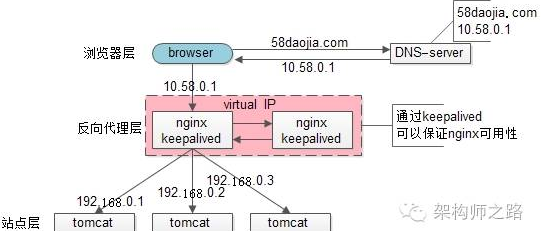
#### 可用性服务

Nginx：负载均衡组成集群，且进行无状态服务即不保存Session。

TODO [亿级PV的负载均衡](https://mp.weixin.qq.com/s?__biz=MzU0OTE4MzYzMw==&mid=2247485829&idx=1&sn=fa04dbd3fec75195232cafbc0085084d&chksm=fbb2807bccc5096d1f66c129f5a9ae1288e2b76aa0b1bf16e807c924af6529abd9d4f46e3261&mpshare=1&scene=1&srcid=#rd])

1.Nginx+keepalived高可用

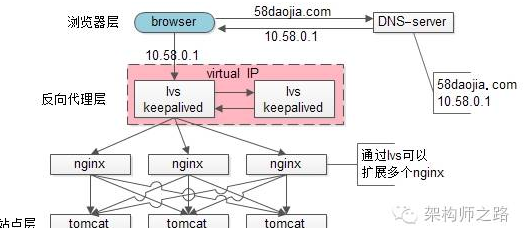
keepalived：检测服务状态存活性，常用来做高可用



当一台nginx挂了，keepalived能够探测到，并将流量自动迁移到另一台nginx上。

单点接入吞吐量有上线。

2.lvs+keepalived



* lvs实施在操作系统层面；f5性能更好在硬件层面；它们性能比nginx好很多。大部分公司到这一步基本就能解决接入层高可用、扩展性、负载均衡的问题。
* lvs也有性能上限，但没几个公司要考虑这个问题。

#### 可用性数据

1）冗余备份

2）失效转移

失效确认，访问转移，数据恢复

数据恢复：

### 3.3.2 伸缩性

随着不断上升的并发访问压力和数据增长，需要不断向集群加入服务器的手段来缓解。伸缩性衡量是否方便构建集群，是否容易添加服务器。

#### 应用服务器

负载均衡不仅提升网站的可用性，还提升伸缩性。

#### 数据存储服务器

不同于缓存，数据存储服务器集群的伸缩性对data的持久化和可用性提出更高要求。

关系数据库

* 主从复制：数据复制，进行简单伸缩。Master写数据，Slave读数据实现读写分离。
* 分库：受跨库join制约。
* 分表：基本主从复制分库，一些单表数据量过大。将一张表拆分开分表存到多个db。

#### NoSql

缓存设计

缓存数据丢失可从db获取，不影响业务，而db需要保证服务可用和数据可靠存储。

MongoDB

易伸缩，自动故障转移。易伸缩指的是提供了分片能力，能对数据集进行分片，数据的存储压力分摊给多台服务器。自动故障转移是副本集的概念，MongoDB能检测主节点是否存活，当失活时能自动提升从节点为主节点，达到故障转移。

### 3.3.3 扩展性

指现有系统影响最小的情况下，系统扩展或提升的能力。开闭原则。

分布式系统集成方式：1）分布式消息队列 2）分布式服务

**消息中间件**

通过消息分解系统解耦性，不同子系统处理同一消息

**分布式服务**

具体内容，见微服务章节。

### 3.2.5 安全性

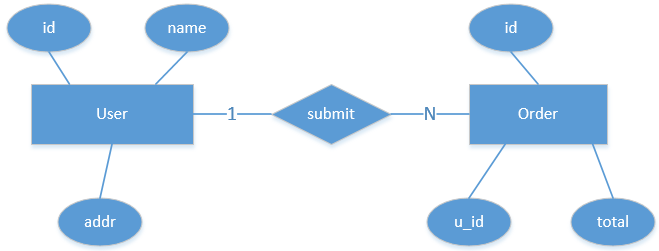
TODO

js Cookie窃取和Session劫持

# 第四章 Design Tools

## 4.1.1 ER图

E-R图也称实体-联系图(Entity Relationship Diagram)，广泛用于数据库设计。



基本要素

* 矩形框：表示实体。
* 菱形框：表示联系。
* 椭圆形框：表示实体或联系的属性

一般性约束

存在3种一般性约束：一对一约束（联系）、一对多约束（联系）和多对多约束（联系），它们用来描述实体集之间的数量约束：

## 4.1.2 UML图

即Unified Model Language，是一种模型化和图形化语言，每种图形从不同角度阐述软件系统，通过各种模型的搭建共同建造起整个软件系统。

UML分别描述系统的类/对象/关联/职责/行为/接口/用例/包/顺序/协作，以及状态。

作用：在软件开发中，用图形抽象地来表达复杂的概念，让整个软件设计更具有可读性，可理解性，以便尽早发现软件设计时存在的潜在问题，从而降低开发风险。同时极大地方便了业务人员与开发人员之间的交流。

分类

UML分结构型图和行为型图，结构是静态的，有类图、对象图、构件图、部署图、包图。

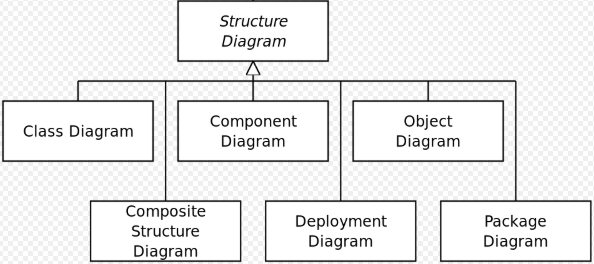
行为是动态的，有活动图、状态图、顺序图、通信图、用例图、时序图。

UML2.0

UML 2.x中一共定义了14种图示。分类的结构[Reference](https://zh.wikipedia.org/wiki/%E7%BB%9F%E4%B8%80%E5%BB%BA%E6%A8%A1%E8%AF%AD%E8%A8%80#/media/File:Uml_diagram.svg)，Diagram：

Structure diagrams

结构性图形。强调的是系统式的建模

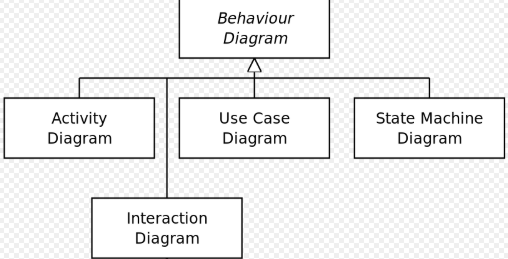


静态图(static diagram)：类图/对象图/包图。

实现图(implementation diagram)：组件图/部署图/剖面图/复合结构图。

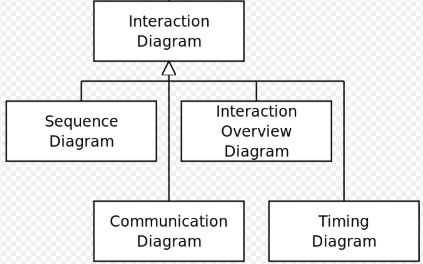
Behavior diagrams

行为式图形。强调系统模型中触发的事件。包括活动图/状态图/用例图



Interaction diagrams（交互性图形）

属于行为图形的子集合，强调系统模型中的资料流程：

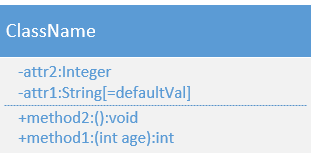


通信图/交互概述图/时序图/时间图

### 结构图

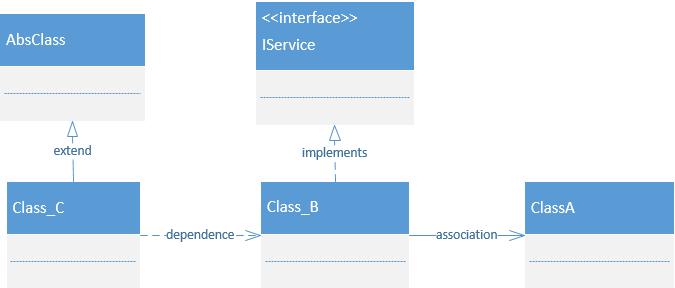
或叫静态图

#### 类图



* 属性：可见性 名称:类型 [ = 缺省值 ]。可见性：+表示public/-表示private/#表示protected（friendly也归入这类）。
* 方法：可见性 名称(参数列表) [ : 返回类型]

关联关系



* Implements：接口实现。接口是对类的行为抽象，抽象程度最高。
* Extentd：继承。泛化(generalization)？
* Dependence ：依赖。局部new/方法的参数传递/静态方法的调用。
* Association：关联，把使用到的类作为成员变量。根据生命周期分为聚合(aggregation)和组合(Composition)。

依赖和关联都是使用到另一个类，关联是一种更强的依赖关系。

聚合(aggregation)



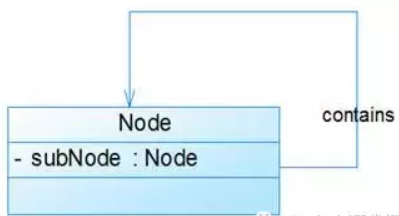
**组合(Composition)**

mouth是Persion组成部分，有相同的生命周期。 较聚合是一种更强的依赖关系。

1. **public** **class** Person {
2. **private** Mouth mouth;
3. **public** Person() {
4. mouth = **new** Mouth();
5. }
6. }



自关联



#### 组件图

### 行为型

流程图

#### 用例图

用行为的图(behavior diagram)描述一系列角色(actors)与用例(use case)之间的关系。

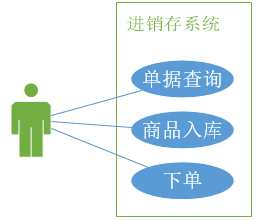
从用户的角度来描述系统的需求，功能和行为。

基本元素

* 参与者：与系统交互的用户。
* 用例：即系统的功能

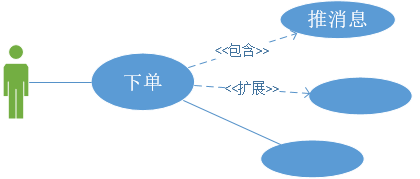
子系统(subsystem)

展示系统部分功能



关系

* 关联
* 泛化：继承。
* 包含：把一个较复杂用例所表示的功能分解成较小的步骤
* 扩展



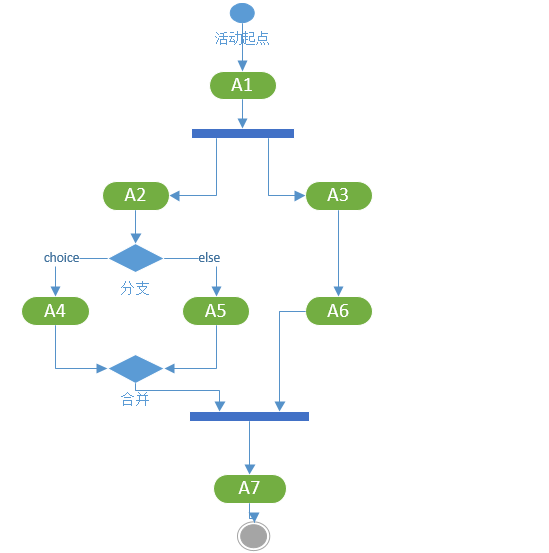
#### 活动图

[Reference](https://www.cnblogs.com/wolf-sun/p/3432135.html)

本质上是一种流程图。

基本要素

* 活动：表示工作流过程中命令的执行或活动的进行。
* 开始/结束
* 分支合并：分支根据条件值决定动作的流向。合并即合并路径。
* 分叉/汇合：分叉用于将一个控制流分为两个或多个并发运行的分支。
* 泳道：泳道表明每个活动是由哪些人或哪些部门负责完成。



#### 状态图

### 交互性图形

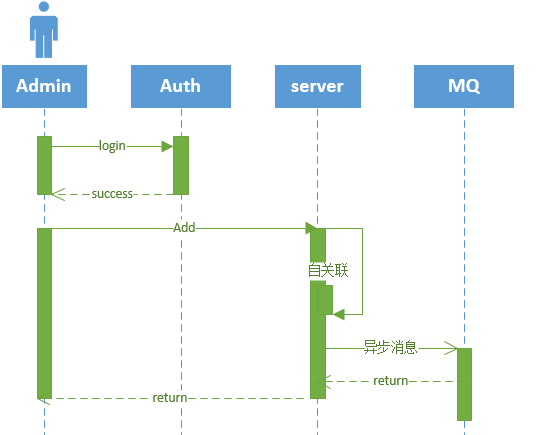
#### 时序图(Sequence Diagram)

别名：顺序图/时序图

[Reference](https://www.cnblogs.com/springyangwc/archive/2011/12/29/2306034.html)

描述对象间交互。

将对象间交互行为建模为消息传递，强调消息的顺序。



基本元素

* 角色(Actor)：系统角色，可以是人或者其他系统，子系统。小人图标表示。
* 对象(Object)：代表对象在交互中所扮演的角色，以矩形表示。对象名：ObjectName:ClassName，:ClassName，ObjectName
* 生命线(LifeLine)：Object的生存时间。
* 控制焦点(Activation)：或叫激活。对象时间线上某段时期执行的操作。以窄矩形表示。
* 消息：对象之间发送的信息。消息分为三种类型。同步/异步/返回消息
* 控制流：顺序/分支/循环

用例图，活动图，时序图直接的关系？

* 时序图：用对象的交互描述用例。
* 活动图：从用户角度描述用例

## 4.1.3 应用分层架构图

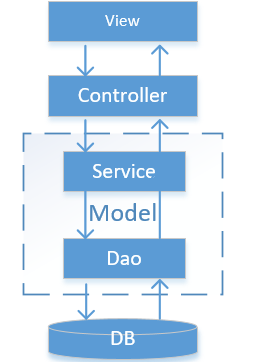
1.结构清楚，2.分工明确，提升可读性，好维护和升级。3.降低模块间的耦合度。

SpringMVC

Spring 框架已经成为构建企业级 Java 应用事实上的标准了，分层结构：

Controller🡪Service（业务逻辑）🡪Dao（Data Access Object）

MVC的M = Service+Dao，将业务逻辑和db访问分开。



## 4.1.4 系统架构图

分层设计

把系统的逻辑功能按照某些原则抽象出几个层次，层次内保持高类聚，层次间保持低耦合，整个系统达到概念清晰，边界清晰，最终功能稳定。

架构域的分类

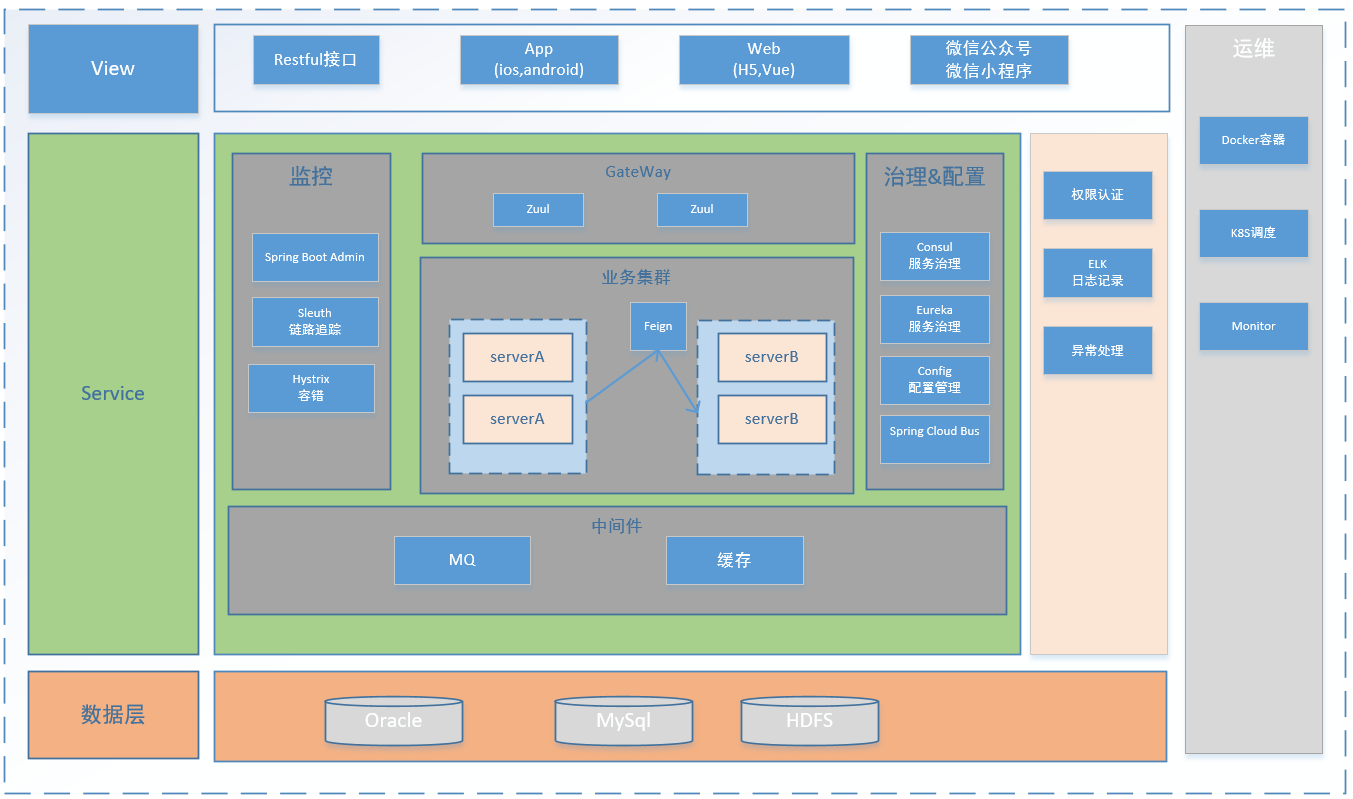
根据需求,形成业务架构,业务架构是战略 🡪 应用架构承上启下,一方面承接业务架构的落地，另一方面影响技术架构的选型 🡪 最后通过技术架构落地实施



[Reference1](https://www.infoq.cn/article/b1fCLl8Mk9L9qe45Zxp6)

### D1

最原始的系统架构图，简单分层展示不同层次的模块，加上基础服务、公共服务和监控服务：



### D2

## 4.1.5 SpringCloud系统架构图

# 第五章 Web设计实践

或者叫常见的解决方案

### API设计

需求：Desktop时代前端后端融合在一起（JSP），随着互联网发展，各种类型client层出不穷。因此需要一种统一的机制，方便不同的前端设备与后端进行通信。这导致API构架的流行，RESTful API是目前比较成熟的一套互联网应用程序的API设计理论。

#### 接口设计

1）设计原则

开放API接口设计没有什么特别，原则同上篇OO设计准则

* 单一职责：粒度适中，也不要做多职能接口。
* 开闭原则：接口要做到开闭，避免大批量下游不可用。同时也要避免过度设计，当抽象功能只有一处使用时，尽量不要过早抽象。
* 命名规范，文档清晰详细。返回值要友好

2）同步 VS 异步

3）RPC VS NoRPC

REST: 跨语言，只需根据url传参调用。服务独立发布部署。耦合度低

第三方jar： 直接调用性能佳，但升级麻烦，

4）RPC VS REST

RPC: REST API 和 RPC 都是函数封装成接口暴露出去，以供调用。不过 REST API 一般都是基于 HTTP 协议，而 RPC 则可以是socket,http,SOAP等.Rest是web service的一种方式,可以归到Http协议的RPC。

5）高并发

#### RESTful 风格

[Reference](http://www.ruanyifeng.com/blog/2011/09/restful.html)

REST特点：结构清晰、符合标准、易于理解、扩展方便

设计细则

[Reference](http://www.ruanyifeng.com/blog/2014/05/restful_api.html)

URL：<https://api.example.com/v1/zoos>

* 协议：API与用户的通信协议，总是使用HTTPs协议。
* 域名：尽量将API部署在专用域名之下。
* 版本：API的版本号放入URL。
* 路径：路径又称"终点"（endpoint），表示具体资源。所以只能是名词，且为复数。
* Http动词：对于资源的具体操作类型，由HTTP动词表示。GET/POST/PUT/PATCH/DELETE
* Filtering：允许路径和URL参数偶尔有重复。比如,/z/1/a 与/a?z\_id=1。
* Status Codes：返回的状态码和提示信息，200 OK，201 CREATED
* Error handling：如果状态码4xx，应返回出错信息{error: "Invalid API key"}
* Hypermedia API：hypermedia-driven，不用看文档，使用方便。

3.3.4.1 RESTful架构

Representational State Transfer，表现层状态转化，是一种互联网软件架构，Client和Server的交互性形式。表现层指的是"资源"（Resources）的"表现层"。

Resources

"资源"，是一个信息实体。可以是一段文本/一张图片/一种服务。每种资源对应一个特定的URI。要获取这个资源，访问它的URI即可。

表现层（Representation）

资源有多种外在表现形式。具体呈现出来的形式叫做它的表现层。比如，文本可以用txt/XML/HTML/JSON表现。

State Transefer

通过Http动词实现。

REST 成熟度模型

**Level 0：** 使用 http作为传输方式。

**Level 1：** 引入了资源的概念。

**Level 2：** 使用 http 方法进行操作，并使用 http状态码来表示结果。

Level 3：使用HATEOAS。在资源的表达中包含了链接信息。client根据链接来发现可执行的动作。

Hypermedia API

[Reference](https://spring.io/understanding/HATEOAS)

hypermedia-driven,make it easy to glean how to interact with the server without looking up a specification or external document。

比如集合接口的每个item都有href连接，图片接口返回不同大小图片的链接

#### 前后端分离

1.随着移动应用的快速发展，此时出现前后端分离。

当前趋势网络应用程序应分为前端和后端两个部分。

### 异常设计

[Reference](https://www.jianshu.com/p/503901e5be2b)

减少程序错误，提高健壮性(Exception可处理)。是重要的调试手段，根据抛出的异常能判断发生的位置和出了什么出错，为什么出错。

什么时候抛异常？

可能会产生问题就抛出异常。

Java异常



Error VS Exception

* Error：不可处理，编译和系统的错误，应中断。OutOfMemoryError
* Exception： 可处理。

Checked Exception

强制catch,否则编译不通过。比如IOEx/ClassNotFoundEx。

Problems

处理麻烦，可读性差，大部分情况coder不知道如何处理，强制处理不现实（保证兼容性不可去掉），Think in Java建议:

1.print to console

2.转换成unchecked抛出保证信息不丢失。

什么时候抛异常?

可能出现问题的地方

1. Address createAddress(Integer uid, Address address) {
2. Preconditions.checkNotNull(uid);
3. User user = userDao.findOne(uid);
4. **if**(**null** == user)    **throw** **new** RuntimeException("找不到当前用户!");
5. }

业务校验时可以抛异常。

系统异常处理

一般出现异常记录日志后逐层向上抛。在web层设置统一异常处理机制。

api：应用内部推荐异常抛出，跨应用/开放API接口要返回错误码和提示信息。

页面：要跳转到友好错误页面。

1. **public** **class** CustomExceptionResolver **implements** HandlerExceptionResolver {
2. MV resolveException(request,response, Object handler, Exception e) {
3. LOG.error(e.getMessage(),e);
4. ModelAndView mv = **new** ModelAndView();
5. **if**(e **instanceof** AppException) {
6. String msg={"success":false,"msg":+e.getMsg();
7. response.getWriter().write(msg);
8. }**else** **if**(e **instanceof** CustomException) {
9. mv.addObject("message", e.getMessage());
10. mv.setViewName("/WEB-INF/jsp/error.jsp");
11. }
12. **return** mv;
13. }
14. }

将异常分类

一般异常框架有如下几种分类

1）AppException

只用一个AppException，在这个异常类中添加一个type属性，通过type来区分不同的异常。type可以用一个枚举来维护，type值可以是参数格式、业务校验、服务。也可以只用一个AppException，所有的异常都封装成这个，不再按type细分。

2）按层定义

DaoException/ServiceException/AppException。使用时service层把DaoException转为ServiceException，再在应用层把ServiceException封装成AppException。最外层有一个拦截全局异常，拦截AppException就可以了。

4）分类

ArgumentException: 参数格式异常，在所有的参数格式校验出错的时候使用。

BusinessException: 业务校验异常

ServiceException: 服务异常

1）设计的很细

异常太多，太麻烦。比如User表格字段校验异常。

自定义异常

DAO层和Service层，产生的异常类型有很多，无法用细粒度的异常进行 catch，避免直接throw new Exception,应使用有业务含义的自定义异常。推荐业界已定义过的自定义异常，如：DAOException / ServiceException等。

使用技巧

3.代码可读性，干掉if else，减少嵌套?

1) 仅异常情况下使用异常

异常不要用来做流程控制，条件控制。异常设计的**初衷**是解决程序运行中的各种意外情况，且异常的处理**效率**比条件判断方式要低很多。



（虽然这里的异常可以用return Result，来减少嵌套并携带错误信息）

2) 异常带上参数信息

记录日志时，尽可能带上参数信息，便于调试

3) 不要捕捉所有异常

1. **try** {
2. // do someting that might throw IOException
3. // do someting that might throw CertificateException
4. } **catch** (Exception e) {  }

不推荐的写法，应尽量避免,这样些会可能带来一些问题：

* 调用的接口的潜在Bug被忽略
* 不同类型的异常可能需要不同的处理，写在一起内聚性比较差

上面的代码可以更改为：

1. **try** {
2. // do someting that might throw IOException
3. // do someting that might throw CertificateException
4. } **catch** (IOException e) {
5. // Do something handle IOException
6. } **catch** (CertificateException e) {
7. // Do something handle CertificateException
8. }

如果所有异常处理方式一致，且JDK版本高于1.7，可以简化如下：

1. **try** {
2. // do someting that might throw IOException
3. // do someting that might throw CertificateException
4. } **catch** (IOException | CertificateException e) {
5. // Do something handle these exception
6. }

### 菜单递归查找

Find all root menu

1. **public** **void** buildMenu() {
2. List<Menu> allMenus = **new** ArrayList<Menu>();//dao.getAllMenus();
3. List<Menu> parents = **new** ArrayList<Menu>();
4. //find all 1 level menu
5. **for** (Menu menu : allMenus) {
6. **if** (menu.getParentId() == **null**) {
7. parents.add(menu);
8. }
9. }
10. **for** (Menu menu : parents) {
11. menu.setChildren(getChild(menu.getMenuId(),allMenus));
12. }
13. }

getChild

1. **public** List<Menu> getChild(**long** menuId,List<Menu> menuList) {
2. List<Menu> childList = **new** ArrayList<Menu>();
3. **for** (Menu menu : menuList) {
4. **if** (menu.getParentId().equals(menuId)) {
5. childList.add(menu);
6. }
7. }
8. //递归终止条件
9. **if** (childList.size()==0) {
10. **return** **null**;
11. }
12. //开始递归
13. **for** (Menu menu : childList) {
14. menu.setChildren(menu.getChild(menu.getMenuId(),menuList));
15. }
16. **return** childList;
17. }

### 应用分层

2.1.3 MVC

1978年提出，MVC不是一种技术，而是一种理念，强调职责分离。

背景：早期JSP网页，SQL query和HTML代码混在一起。MVC从根本上强制分层，尽管需要一些额外的工作，但好处是毋庸置疑。

MVC是一种软件设计模式，MVC 模式代表 Model-View-Controller（模型-视图-控制器） 模式。这种模式用于应用程序的分层开发。

**View**: UI操作界面，Model包含数据的显示。

**Model**：封装数据源和对这些数据的操作。

**Control**：介入view和model，处理user在view上的输入，并转发给model，这样Model和View两者之间可以做到松散耦合，甚至可以彼此不知道对方，而由Controller连接起这两个部分。

好处

便于分工合作。

降低系统复杂度。

todo SPRING ModelAndView

更细致的分层

在阿里的编码规范中约束的分层如下:



* **终端显示层**：PC端，移动端展示等。
* **开放接口层**：封装 Service 方法暴露成 RPC 接口/通过 Web 封装成 http 接口（需要进行网关安全控制、流量控制等）。
* **Web 层**：即Controller层，主要是各类基本参数校验，或者不复用的业务简单处理等。
* **Service层**：相对具体的业务逻辑
* **Manager 层**：通用业务处理层，它有如下特征：

1）对第三方平台封装的层，预处理返回结果及转化异常信息；

2）对 Service层通用能力的下沉，如缓存方案、中间件通用处理；

3）与 Dao 层交互，对多个 Dao的组合复用。

常规校验

在使用平台资源，譬如短信、邮件、电话、下单、支付，必须实现正确的防重放限制，

如数量限制、疲劳度控制、验证码校验，避免被滥刷、资损。

说明： 如注册时发送验证码到手机，如果没有限制次数和频率，那么可以利用此功能骚扰到其它用户，并造成短信平台资源浪费。

参数校验

用户请求传入的任何参数必须做有效性验证。

说明：忽略参数校验可能导致：

page size 过大导致内存溢出

恶意 order by 导致数据库慢查询

任意重定向

SQL 注入

反序列化注入

正则输入源串拒绝服务 ReDoS

说明：Java 代码用正则来验证客户端的输入，有些正则写法验证普通用户输入没有问题，

但是如果攻击人员使用的是特殊构造的字符串来验证，有可能导致死循环的结果。

**参数的安全性判断应该写在controller还是service？异常处理尼？**

哪一层都是很有必要的，一般在controller进行，在service层进行校验更好的复用。

### Session

Session用来在Server端保存临时全局保量，Session+Cookie：弥补Http无状态的不足。

Session VS Token

Token：Server端Session的存储和查询都要占用一定的系统资源。同Session，client的每个请求都要携带。[Reference](https://mp.weixin.qq.com/s?__biz=MzAxNDMwMTMwMw==&mid=2247490367&idx=1&sn=3a3f62570435ef1808369676c707a00c&chksm=9b943a27ace3b3316108b3833301fc10e9bca0e4bf83c928fae024ad52384c44d38791ded619&mpshare=1&scene=1&srcid=#rd])

原理：以User登录为例，Session在服务端需要记录用户的状态，Client在Cookie且在每次请求带上cookie信息，server根据请求的cookie验证登录状态。

TODO:Session劫持

#### 实现原理

在服务端保存session的方法很多(内存、数据库、文件等)。Servlet容器的Session数据结构为Servlet规范中的HttpSession实现类，由厂商实现。**以Tomcat为例**。

1.服务端在Client第一次request生成的JSESSIONID放内存，并写到客户端cookie，

2.Client之后的request的head都带着sessionId。

TODO1 session复制如何配置？

（如果不设置cookie过期时间，cookie会在会话结束后销毁，称为会话cookie）

问题隐患：

sesssion存在server内存，增加服务端压力。

#### 集群session

集群的时候要考虑Session的共享。

1）存Cookie

原理：client每次请求带着完整cookie信息。

缺点：增加网络负担。Cookie大小限制。

2）Tomcat Session复制

原理：每个节点存在相同的Session，新加入的复制一份Session信息。适用小规模的系统。

缺点：1.一致性问题。节点越多，损耗越大（延迟，宽带）。2.单机内存有上线。

1. Session sticky

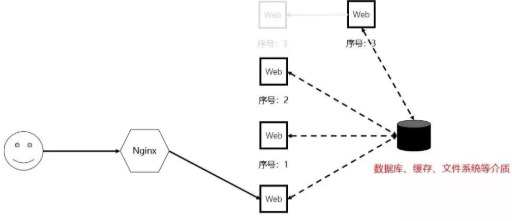
Apache做负载均衡器，用户的请求转发到特定的Tomcat服务器上。

缺点：服务器重启，丢失会话，可用性不高。

1. 集中管理

Session集中管理。

缺点：增加网络开销。



Session服务器方便横向扩展。可无限扩张，适用于分布式系统。

集中管理的具体实现：

1.spring session

2.实现 HttpServletRequestWrapper

#### Spring Session

[Reference](http://blog.csdn.net/xiao__gui/article/details/52706243)

设计一个Filter，利用HttpServletRequestWrapper，实现自己的 getSession()方法，接管创建和管理Session数据的工作。

Nginx + Redis

自定义filter, Nginx + tomcat+ redis

Nginx配置为non-sticky运行模式，也即每一个请求都可以被分配到集群中的任何节点

WebSocket session

#### 单点登录(SSO)

session管理

[Reference](http://www.cnblogs.com/ywlaker/p/6113927.html)

Unicode编码

### Web容器

Netty版 [Reference](https://mp.weixin.qq.com/s?__biz=MzI3NzE0NjcwMg==&mid=2650123218&idx=2&sn=649438ed0aa4fa681ea077dc0b3a8395&chksm=f36bb6f3c41c3fe5de8c1bfd5fddbebe49a6b87f334b904fa6232fad09fed91fc61d5dfe12d7&mpshare=1&scene=1&srcid=&pass_ticket=rMfcWI31qoBc%2BJleH2gnwJiWltH0srLHo3%2BPGnyWlwrmp8k7vmb0%2FxHQPasaYnLj#rd)

### 网站安全

[Reference](https://mp.weixin.qq.com/s?__biz=MzAxNDMwMTMwMw==&mid=2247489948&idx=1&sn=b71b3e5070cbba089f594a0425255c23&chksm=9b943884ace3b1924a7b0fd3b5521e016e531a180a44fb1e40bd319aa3386c89f323eff60e39&mpshare=1&scene=1&srcid=0111WJPmhORp4e0NDeFpoB0u#rd])

明文传输

Spring Security本质就是明文，需要配合Https。

对称加密

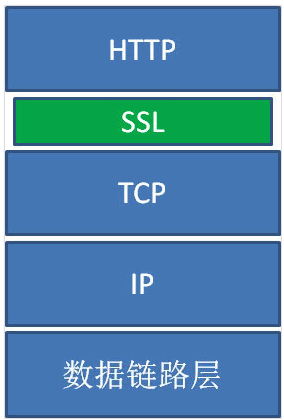
* HTTP Basic Authentication：基于BASE64加密，强度较低。即便是其他算法，前端js加密逻辑容易被破解。

1. Get /index.html HTTP/1.0
2. Host:www.google.com
3. Authorization: Basic xxxxxxxxxxxxxxxxxxxxxxxxxxxx
4. //后台
5. **new** BASE64Decoder().decodeBuffer(pass)

* MD5加密：WEB端将<name,MD5(pass)>传输至到server，后台与DB中的密文进行比较。

MD5加密虽然不可逆。hacker截获加密后的密文，伪造HTTP请求，一样可以登录成功。

HTTPS非对称加密



通信过程：建立通信时，server首先将公钥key1发送给client。

中间人攻击

中间人截获了Key1后，自己生成一对公钥私钥，把自己的公钥Key2发送给client。

Client 🡸 中间人 🡸 Server

证书颁发机构（CA）

证书签名和网址信息对应。

各大浏览器和操作系统已经维护了所有权威证书机构的名称和公钥。

input黑点：password input黑点可用js获取到值，在客户端仅仅用于防偷窥。

1994 网景创建，基于SSL。

Http的信息传输完全以明文方式，不做任何加密。Https防止通信过程中被第三方获取明文。

中间人攻击

#### RESTful API

1.客户端HTTP请求登录

2.server鉴权通过返回token，并将token和客户端关系维护在redis或DB

3.client将 token保存在本地，之后放请求的header，一旦token过期再次请求。

### 电商网站

基本功能

User：注册/管理/维护

Product：列表，管理

Order:下单/管理订单

提交订单

重复提交 [Reference](https://www.jianshu.com/p/e618cc818432)

支付流程

### VO&&DTO

[Reference](https://mp.weixin.qq.com/s?__biz=MzUxOTAxODc2Mg==&mid=2247485700&idx=1&sn=ab880b3a4e0a9dddcea016afda93a629&chksm=f9814961cef6c0777ae751b141dd9aad811c80276be7ac57d48f9e05fbebcf1e4c759b464cba&mpshare=1&scene=1&srcid=&sharer_sharetime=1573319158101&sharer_shareid=ff601b700721a407cdee60a9c63c1b87&key=ce83d961ff3ea7ac866322d3c75cbb5e18897a8da31ad1b46a06fe8e6f8522ddf0eada7073cb6f4ac34a0d29e1df2b7ee33f525784a947dc9db0b15c9a80d4c00bb87361b21927633767887c4d04c00e&ascene=1&uin=Mjc3ODQ1MTk0MA%3D%3D&devicetype=Windows+10&version=62070152&lang=zh_CN&pass_ticket=MCT3fCw7MCE676VxtSEKLj06vJaYj34U2Mb9jdEdZh8vRmoBGD%2FJdxemhK0hUxv2)

## 5.4 架构设计Case

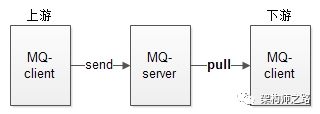
### 秒杀系统

[Reference](https://mp.weixin.qq.com/s?__biz=MzIwMzY1OTU1NQ==&mid=2247484606&idx=1&sn=4f37b694e67747aa16d37dd3ec288225&chksm=96cd44f2a1bacde4d84e02e8027fe77d075d84a689da747d84389c532fb4321255184e662591&mpshare=1&scene=1&srcid=0926E9jrsacrVCbAbdvtioUv#rd])

1）缓冲流量

削峰=缓冲流量。

流量冲击：直接调用/MQ采用push给下游，下游接收方无法控制到达的流量，若调用方不限速，很可能被压垮。



这里下游的MQ-client根据server处理能力pull消息，从队列中处理业务。

2）性能优化

高并发导致程序阻塞**，**用户感受卡顿。

* 做好缓存：Read数据尽量放缓存，切断User和db的直接交互。并提前预热数据。

Warning

从缓存读可能要牺牲强一致性，保证最终一致性。比如从Slave查看，下单操作master

3）控制库存

问题2：库存无法有效控制，出现超卖现象。

分布式锁。

### 设计一个消息中间件

### 消息推送系统

#### Base

[Reference](https://mp.weixin.qq.com/s?__biz=MzI4NDY5Mjc1Mg==&mid=2247485859&idx=1&sn=84e882c025682164d50ab5c8b00c0480&chksm=ebf6d1dcdc8158ca644c874f3701a9d84f01e4ede53c1bb590fa36ebfa346b37b6b68d93bd78&mpshare=1&scene=1&srcid=0929OgZWIOVUIIeKhyZBaU35#rd])

* 技术选型：NIO 1.大量连接 2.双工通信 （选择不多Netty，社区，资料维护)
* 协议解析：定制私有协议。要精简，减少不必要的传输。
* 通道关系：client和Channel的关系维护。单机版用一个map保存即可。分布式环境client可能connect到不同server

#### 实现功能点

上一章节已结束分布式的基础功能，消息推送系统在分布式的基础上还要实现下列

* 消息流转：客户端多，上行量大，不适合在server中，可用消息中间件解耦，比如kafka。
* 心跳：server和client都要心跳，通过注册中心检测各节点的状态

中间件

[Reference](https://kb.cnblogs.com/page/196448/)

一般情况，中间件应用于分布式环境，主要解决异构网络环境下模块的互连与互操作问题，提供标准接口、协议，屏蔽实现细节（将具体业务和底层逻辑解耦的组件），提高应用系统移植性。

分类：数据访问中间件，远程调用中间件，消息中间件，交易中间件，对象中间件。

#### 消息队列

kafka：高吞吐的分布式发布订阅消息系统。

