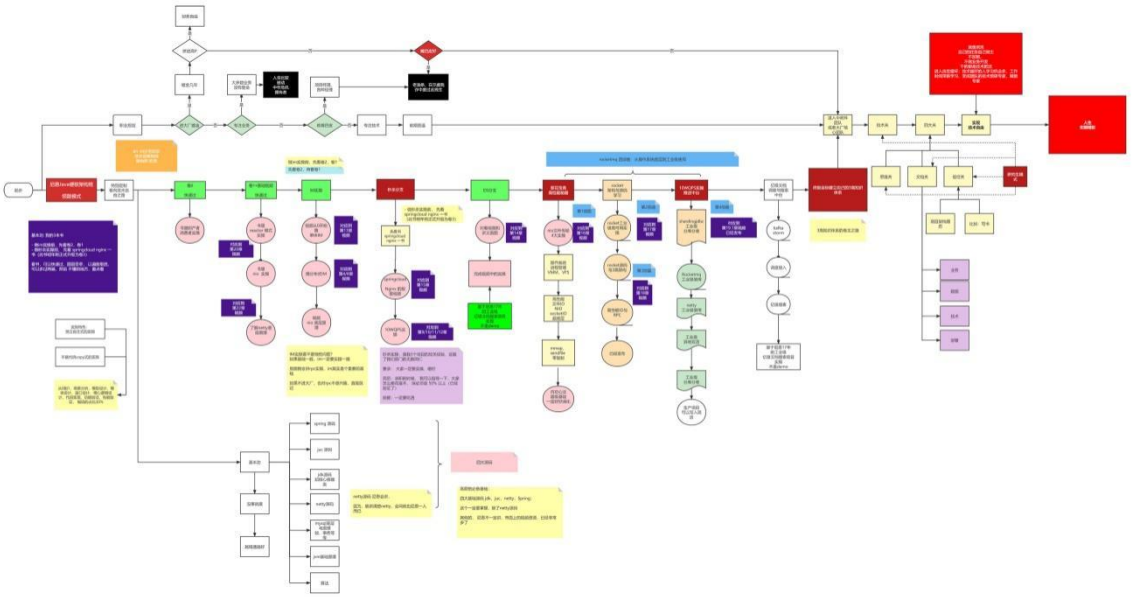
牛逼的职业发展之路

40 岁老架构尼恩用一张图揭秘：Java 工程师的高端职业发展路径，走向食物链顶端的之路

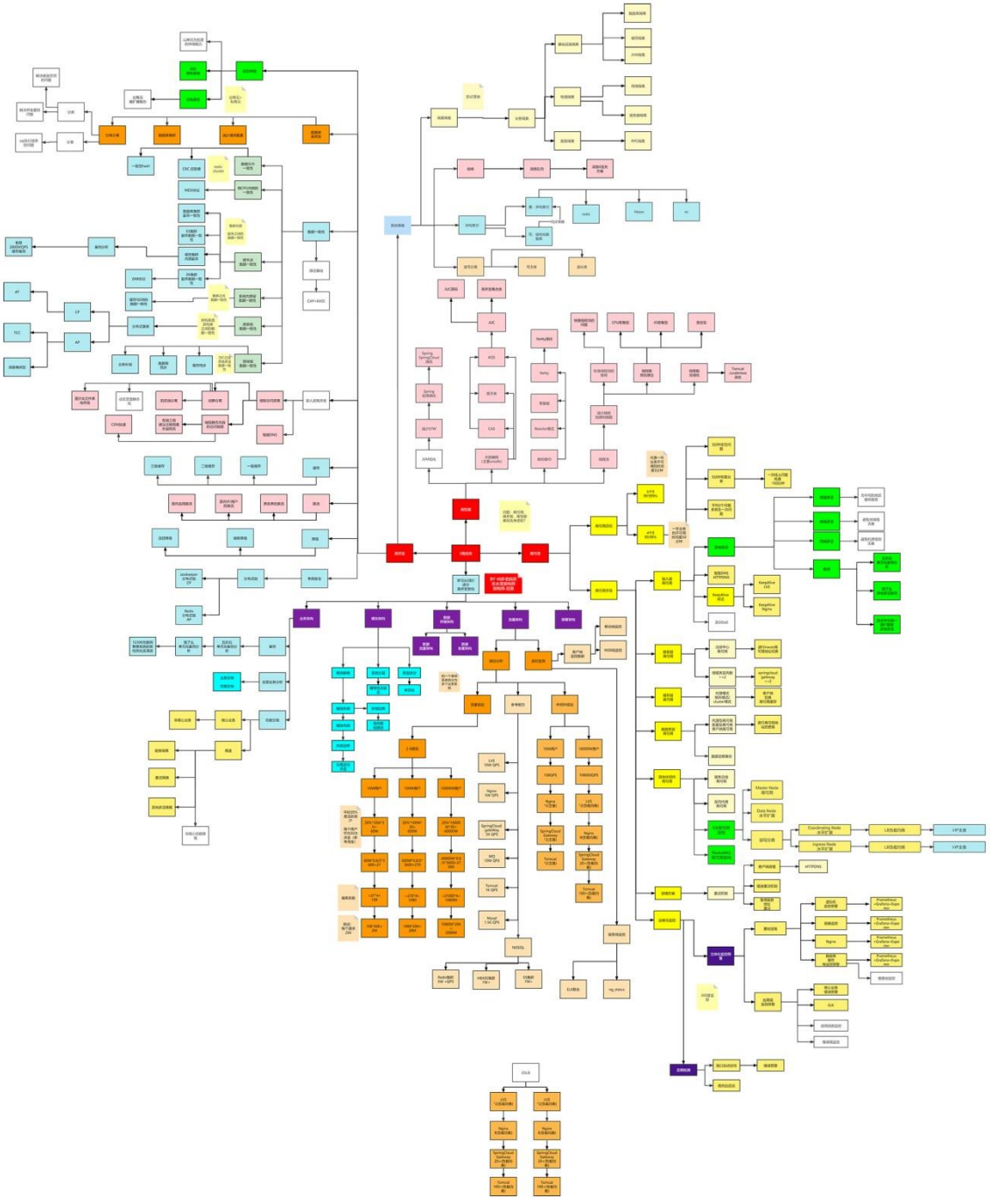
链接：https://www.processon.com/view/link/618a2b62e0b34d73f7eb3cd7



史上最全：价值10W的架构师知识图谱

此图梳理于尼恩的多个 3 高生产项目：多个亿级人民币的大型 SAAS 平台和智慧城市项目

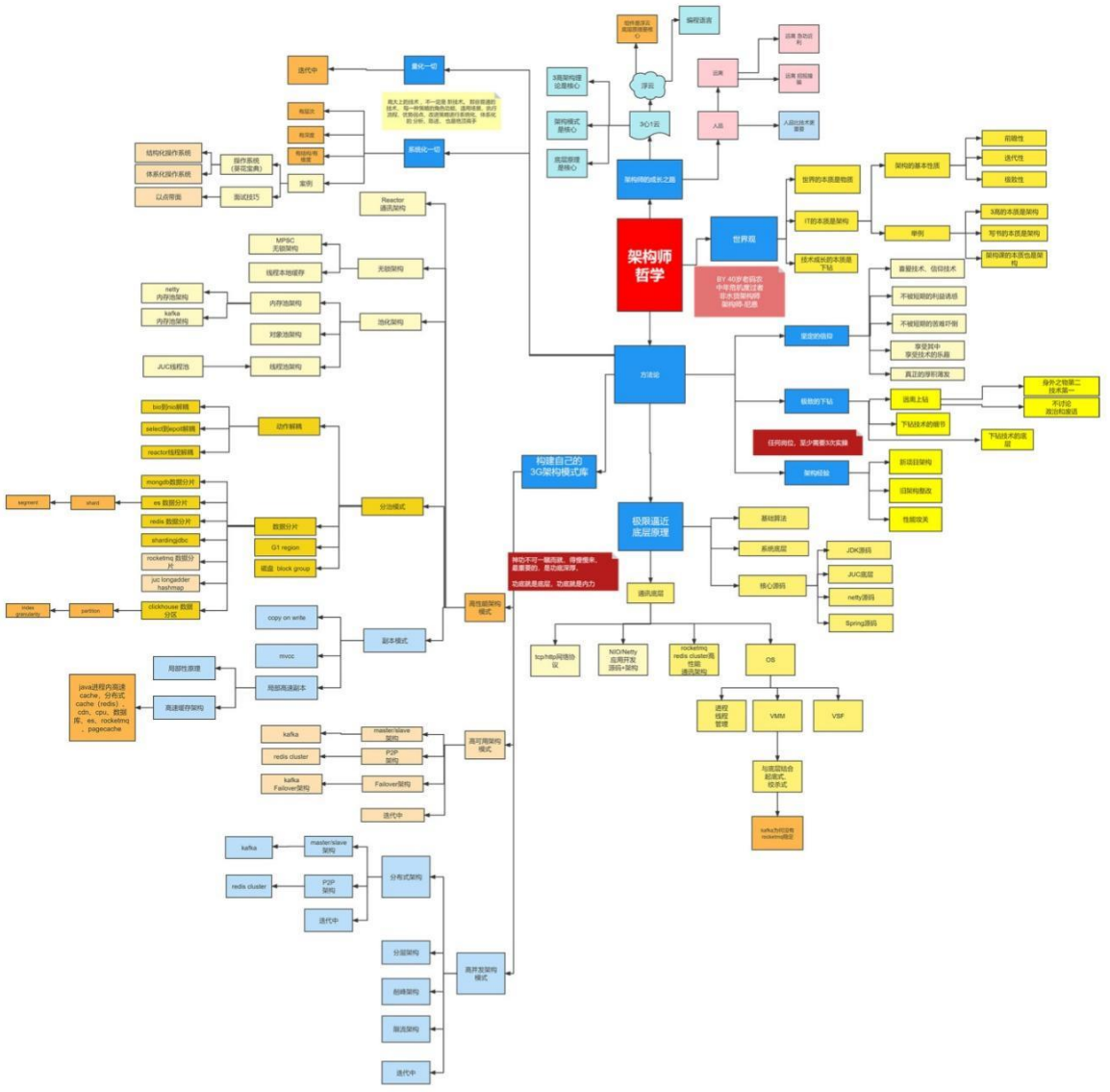
链接：https://www.processon.com/view/link/60fb94216376897l9d246739



牛逼的架构师哲学

40 岁老架构师尼恩对自己的 20 年的开发、架构经验总结

链接：https://www.processon.com/view/link/616f801963768961e9d9aec8



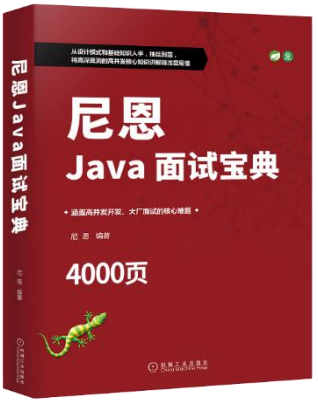
牛逼的3高架构知识宇宙

尼恩 3 高架构知识宇宙，帮助大家穿透 3 高架构，走向技术自由，远离中年危机

链接：https://www.processon.com/view/link/635097d2e0b34d40be778ab4

尼恩Java面试宝典

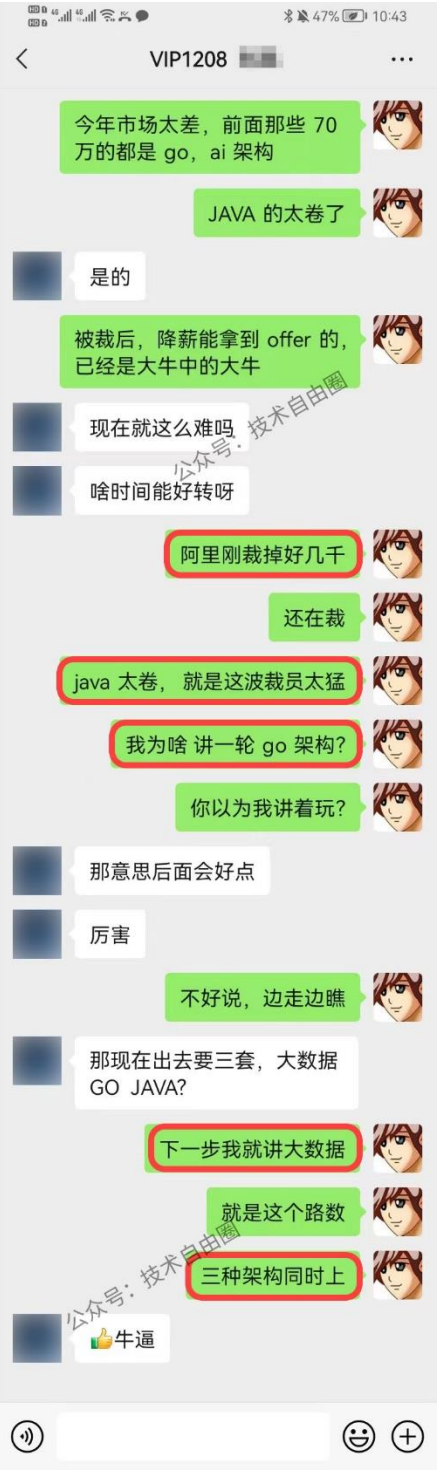
40 个专题（卷王专供+ 史上最全 + 2023 面试必备）
详情：https://www.cnblogs.com/crazymakercircle/p/13917138.html



名称

专题01：JVM面试题（卷王专供 + 史上最全 + 2022面试必备）-V81-from-尼恩Java面试宝典.pdf
专题02：Java算法面试题（卷王专供 + 史上最全 + 2022面试必备）-V80-from-Java面试红宝书.pdf
专题03：Java基础面试题（卷王专供+ 史上最全 + 2023面试必备）-V2-from-Java面试红宝书 -release.pdf
专题04：架构设计面试题（卷王专供+ 史上最全 + 2023面试必备）-V86-from-尼恩Java面试宝典.pdf
专题05：Spring面试题_专题06：SpringMVC_专题07：Tomcat面试题（卷王专供+ 史上最全 + 2023面试必备）-V3-from-尼恩面试宝典-release.pdf
专题08：SpringBoot面试题（卷王专供+ 史上最全 + 2023面试必备）-V2-from-Java面试红宝书-release.pdf
专题09：网络协议面试题（卷王专供+ 史上最全 + 2023面试必备）-V46-from-尼恩Java面试宝典-release.pdf
专题10：TCPIP协议（卷王专供+ 史上最全 + 2022面试必备）-V57-from-Java面试红宝书.pdf
专题11：JUC并发包与容器类（卷王专供+ 史上最全 + 2023面试必备）-V2-from-Java面试红宝书-release.pdf
专题12：设计模式面试题（卷王专供+ 史上最全 + 2022面试必备）-V84-from-Java面试红宝书.pdf
专题13：死锁面试题（卷王专供+ 史上最全 + 2023面试必备）-V2-from-Java面试红宝书-release.pdf
专题14：Redis 面试题 （卷王专供+ 史上最全 + 2022面试必备）-V65-from-Java面试红宝书.pdf
专题15：分布式锁 面试题（卷王专供+ 史上最全 + 2023面试必备）-V2-from-Java面试红宝书-release.pdf
专题16：Zookeeper 面试题（卷王专供+ 史上最全 + 2023面试必备）-V2-from-Java面试红宝书-release.pdf
专题17：分布式事务面试题（卷王专供 + 史上最全 + 2023面试必备）-V2-from-Java面试红宝书-release.pdf
专题18：一致性协议 （卷王专供+ 史上最全 + 2023面试必备）-V2-from-Java面试红宝书-release.pdf
专题19：Zab协议（卷王专供+ 史上最全 + 2023面试必备）-V2-from-Java面试红宝书-release.pdf
专题20：Paxos 协议（卷王专供+ 史上最全 + 2023面试必备）-V2-from-Java面试红宝书-release.pdf
专题21：raft 协议（卷王专供+ 史上最全 + 2023面试必备）-V2-from-Java面试红宝书-release.pdf
专题22：Linux面试题（卷王专供+ 史上最全 + 2023面试必备）-V2-from-Java面试红宝书-release.pdf
专题23：Mysql 面试题（卷王专供+ 史上最全 + 2023面试必备）-V82-from-尼恩Java面试宝典.pdf
专题24：SpringCloud 面试题（卷王专供+ 史上最全 + 2023面试必备）-V12-from-Java面试红宝书-release.pdf
专题25：Netty 面试题 （卷王专供+ 史上最全 + 2023面试必备）-V2-from-Java面试红宝书-release.pdf
专题26：消息队列面试题：RabbitMQ、Kafka、RocketMQ（卷王专供+ 史上最全 + 2023面试必备）-V10-from-Java面试红宝书-release.pdf
专题27：内存泄漏 内存溢出（卷王专供+ 史上最全 + 2023面试必备）-V2-from-Java面试红宝书-release.pdf
专题28：JVM 内存溢出 实战（卷王专供+ 史上最全 + 2023面试必备）-V17-from-Java面试红宝书-release.pdf
专题29：多线程面试题（卷王专供+ 史上最全 + 2023面试必备）-V66-from-Java面试红宝书.pdf
专题30：HR面试题：过五关斩六将后，小心阴沟翻船！（史上最全、避坑宝典）-V2-from-Java面试红宝书-release.pdf
专题31：Hash连环炮面试题（卷王专供+ 史上最全 + 2022面试必备）-V68-from-Java面试红宝书.pdf
专题32：大厂面试的基本流程和面试准备（阿里、腾讯、网易、京东、头条.....）-V2-from-Java面试红宝书-release.pdf
专题33：BST、AVL、RBT红黑树、三大核心数据结构（卷王专供+ 史上最全 + 2023面试必备）-V2-from-Java面试红宝书-release.pdf
专题34：Elasticsearch面试题（卷王专供+ 史上最全 + 2023面试必备）-V3-from-Java面试红宝书-release.pdf
专题35：Mybatis面试题（卷王专供+ 史上最全 + 2023面试必备）-V3-from-尼恩Java面试宝典-release.pdf
专题36：Dubbo面试题（卷王专供+ 史上最全 + 2023面试必备）-V21-from-尼恩Java面试宝典-release.pdf
专题37：Docker面试题（卷王专供+ 史上最全 + 2023面试必备）-V47-from-尼恩Java面试宝典.pdf
专题38：K8S面试题（卷王专供+ 史上最全 + 2023面试必备）-V59-from-尼恩Java面试宝典.pdf
专题39：Nginx面试题（卷王专供+ 史上最全 + 2023面试必备）-V27-from-尼恩Java面试宝典-release.pdf
专题40：操作系统面试题（卷王专供+ 史上最全 + 2023面试必备）-V28-from-尼恩Java面试宝典-release.pdf
专题41：大厂面试真题（卷王专供+ 史上最全 + 2023面试必备）-V84-from-尼恩Java面试宝典.pdf

未来职业，如何突围：三栖架构师



专题12：设计模式面试题（史上最全、定期更新）

本文版本说明：V85

> 此文的格式，由markdown 通过程序转成而来，由于很多表格，没有来的及调整，出现一个格式问题，尼恩在此给大家道歉啦。
>
> 由于社群很多小伙伴，在面试，不断的交流最新的面试难题，所以，《尼恩Java面试宝典》， 后面会不断升级，迭代。
>
> 本专题，作为 《尼恩Java面试宝典》专题之一， 《尼恩Java面试宝典》一共**30个面试专题，后续还会增加**

《尼恩Java面试宝典》升级的规划为：

后续基本上，**每一个月，都会发布一次**，最新版本，可以扫描扫架构师尼恩微信， 发送 “领取电子书” 获取。

尼恩的微信二维码在哪里呢？ 请参见文末

面试问题交流说明：

如果遇到面试难题，或者职业发展问题， 或者中年危机问题，都可以来 疯狂创客圈社群交流，

加入交流群，加尼恩微信即可， 尼恩的微信二维码在哪里呢？ 请参见文末

升级说明：

V85 版本说明（2023-07-14）：

虾皮一面：手写一个Strategy模式（策略模式）

V14 版本说明：

- 缓存之王 Caffeine 源码中，如何使用单例模式的?
- 链路之王 skywalking 源码中，如何使用单例模式的?

史上最全 Java 面试题：设计模式篇

国内著名电商面试题：

你的项目中用到了哪些设计模式，如何使用?

读完本文，你会有一个最佳的答案。

设计模式基础

什么是设计模式

- 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。使用设计模式是为了可重用代码、让代码更容易被他人理解、保证代码可靠性、程序的重用性。

为什么要学习设计模式

- 看懂源代码：如果你不懂设计模式去看Jdk、Spring、SpringMVC、IO等等等等的源码，你会很迷茫，你会寸步难行
- 看看前辈的代码：你去个公司难道都是新项目让你接手？很有可能是接盘的，前辈的开发难道不用设计模式?
- 编写自己的理想中的好代码：我个人反正是这样的，对于我自己开发的项目我会很认真，我对他比对我女朋友还好，把项目当成自己的儿子一样

设计模式分类

- 创建型模式，共五种：**工厂方法模式、抽象工厂模式、单例模式**、建造者模式、**原型模式。**
- 结构型模式，共七种：适配器模式、装饰器模式、代理模式、外观模式、桥接模式、组合模式、享元模式。
- 行为型模式，共十一种：策略模式、模板方法模式、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。

设计模式的六大原则

开放封闭原则（Open Close Principle）

- 原则思想：尽量通过扩展软件实体来解决需求变化，而不是通过修改已有的代码来完成变化
- 描述：一个软件产品在生命周期内，都会发生变化，既然变化是一个既定的事实，我们就应该在设计的时候尽量适应这些变化，以提高项目的稳定性和灵活性。
- 优点：单一原则告诉我们，每个类都有自己负责的职责，里氏替换原则不能破坏继承关系的体系。

里氏代换原则（Liskov Substitution Principle）

- 原则思想：使用的基类可以在任何地方使用继承的子类，完美的替换基类。
- 大概意思是：子类可以扩展父类的功能，但不能改变父类原有的功能。子类可以实现父类的抽象方法，但不能覆盖父类的非抽象方法，子类中可以增加自己特有的方法。
- 优点：增加程序的健壮性，即使增加了子类，原有的子类还可以继续运行，互不影响。

依赖倒转原则（Dependence Inversion Principle）

- 依赖倒置原则的核心思想是面向接口编程.
- 依赖倒转原则要求我们在程序代码中传递参数时或在关联关系中，尽量引用层次高的抽象层类，
- 这个是开放封闭原则的基础，具体内容是：对接口编程，依赖于抽象而不依赖于具体。

接口隔离原则（Interface Segregation Principle）

- 这个原则的意思是：使用多个隔离的接口，比使用单个接口要好。还是一个降低类之间的耦合度的意思，从这儿我们看出，其实设计模式就是一个软件的设计思想，从大型软件架构出发，为了升级和维护方便。所以上文中多次出现：降低依赖，降低耦合。
- 例如：支付类的接口和订单类的接口，需要把这俩个类别的接口变成俩个隔离的接口

迪米特法则（最少知道原则）（Demeter Principle）

- 原则思想：一个对象应当对其他对象有尽可能少地了解，简称类间解耦
- 大概意思就是一个类尽量减少自己对其他对象的依赖，原则是低耦合，高内聚，只有使各个模块之间的耦合尽量的低，才能提高代码的复用率。

- 优点：低耦合，高内聚。

单一职责原则（Principle of single responsibility）

- 原则思想：一个方法只负责一件事情。
- 描述：单一职责原则很简单，一个方法 一个类只负责一个职责，各个职责的程序改动，不影响其它程序。 这是常识，几乎所有程序员都会遵循这个原则。
- 优点：降低类和类的耦合，提高可读性，增加可维护性和可拓展性，降低可变性的风险。

J2EE中使用了哪些设计模式?

Structural（结构模式）

Adapter:

- java.util.Arrays#asList()
- javax.swing.JTable(TableModel)
- java.io.InputStreamReader(InputStream)
- java.io.OutputStreamWriter(OutputStream)
- javax.xml.bind.annotation.adapters.XmlAdapter#marshal()
- javax.xml.bind.annotation.adapters.XmlAdapter#unmarshal()

Bridge:
把抽象和实现解藕，于是接口和实现可在完全独立开来。

- AWT (提供了抽象层映射于实际的操作系统)
- JDBC

Composite:
让使用者把单独的对象和组合对象混用。

- javax.swing.JComponent#add(Component)
- java.awt.Container#add(Component)
- java.util.Map#putAll(Map)
- java.util.List#addAll(Collection)
- java.util.Set#addAll(Collection)

Decorator:
为一个对象动态的加上一系列的动作，而不需要因为这些动作的不同而产生大量的继承类。这个模式在JDK中几乎无处不在，所以，下面的列表只是一些典型的。

- java.io.BufferedInputStream(InputStream)
- java.io.DataInputStream(InputStream)
- java.io.BufferedOutputStream(OutputStream)
- java.util.zip.ZipOutputStream(OutputStream)
- java.util.Collections#checkedList|Map|Set|SortedSet|SortedMap

Facade:
用一个简单的接口包状一组组件，接口，抽象或是子系统。

- java.lang.Class
- javax.faces.webapp.FacesServlet

Flyweight:
有效率地存储大量的小的对象。

- java.lang.Integer#valueOf(int)
- java.lang.Boolean#valueOf(boolean)
- java.lang.Byte#valueOf(byte)
- java.lang.Character#valueOf(char)

Proxy:
用一个简单的对象来代替一个复杂的对象。

- java.lang.reflect.Proxy
- RMI

Creational（创建模式）

Abstract factory:**

- java.util.Calendar#getInstance()
- java.util.Arrays#asList()
- java.util.ResourceBundle#getBundle()
- java.sql.DriverManager#getConnection()
- java.sql.Connection#createStatement()
- java.sql.Statement#executeQuery()
- java.text.NumberFormat#getInstance()
- javax.xml.transform.TransformerFactory#newInstance()

Builder:
主要用来简化一个复杂的对象的创建。这个模式也可以用来实现一个 Fluent Interface。

- java.lang.StringBuilder#append()
- java.lang.StringBuffer#append()
- java.sql.PreparedStatement
- javax.swing.GroupLayout.Group#addComponent()

Factory:
简单来说，按照需求返回一个类型的实例。

- java.lang.Proxy#newProxyInstance()
- java.lang.Object#toString()
- java.lang.Class#newInstance()
- java.lang.reflect.Array#newInstance()
- java.lang.reflect.Constructor#newInstance()
- java.lang.Boolean#valueOf(String)
- java.lang.Class#forName()

Prototype:
使用自己的实例创建另一个实例。有时候，创建一个实例然后再把已有实例的值拷贝过去，是一个很复杂的动作。所以，使用这个模式可以避免这样的复杂性。

- java.lang.Object#clone()
- java.lang.Cloneable

Singleton:
只允许一个实例。在 Effective Java中建议使用Emun.

- java.lang.Runtime#getRuntime()
- java.awt.Toolkit#getDefaultToolkit()
- java.awt.GraphicsEnvironment#getLocalGraphicsEnvironment()
- java.awt.Desktop#getDesktop()

Behavioral(行为模式)

Chain of responsibility:
把一个对象在一个链接传递直到被处理。在这个链上的所有的对象有相同的接口（抽象类）但却有不同的实现。

- java.util.logging.Logger#log()
- javax.servlet.Filter#doFilter()

Command:
把一个或一些命令封装到一个对象中。

- java.lang.Runnable
- javax.swing.Action

Interpreter:
一个语法解释器的模式。

- java.util.Pattern
- java.text.Normalizer
- java.text.Format

Iterator:
提供一种一致的方法来顺序遍历一个容器中的所有元素。

- java.util.Iterator
- java.util.Enumeration

Mediator:
用来减少对象单的直接通讯的依赖关系。使用一个中间类来管理消息的方向。

- java.util.Timer
- java.util.concurrent.Executor#execute()
- java.util.concurrent.ExecutorService#submit()
- java.lang.reflect.Method#invoke()

Memento:
给一个对象的状态做一个快照。Date类在内部使用了一个long型来做这个快照。

- java.util.Date
- java.io.Serializable

Null Object:
这个模式用来解决如果一个Collection中没有元素的情况。

- java.util.Collections#emptyList()
- java.util.Collections#emptyMap()
- java.util.Collections#emptySet()

Observer:
允许一个对象向所有的侦听的对象广播自己的消息或事件。

- java.util.EventListener
- javax.servlet.http.HttpSessionBindingListener
- javax.servlet.http.HttpSessionAttributeListener
- javax.faces.event.PhaseListener

State:
这个模式允许你可以在运行时很容易地根据自身内部的状态改变对象的行为。

- java.util.Iterator
- javax.faces.lifecycle.LifeCycle#execute()

Strategy:
定义一组算法，并把其封装到一个对象中。然后在运行时，可以灵活的使用其中的一个算法。

- java.util.Comparator#compare()
- javax.servlet.http.HttpServlet
- javax.servlet.Filter#doFilter()

Template method:
允许子类重载部分父类而不需要完全重写。

- java.util.Collections#sort()
- java.io.InputStream#skip()
- java.io.InputStream#read()
- java.util.AbstractList#indexOf()

Visitor:

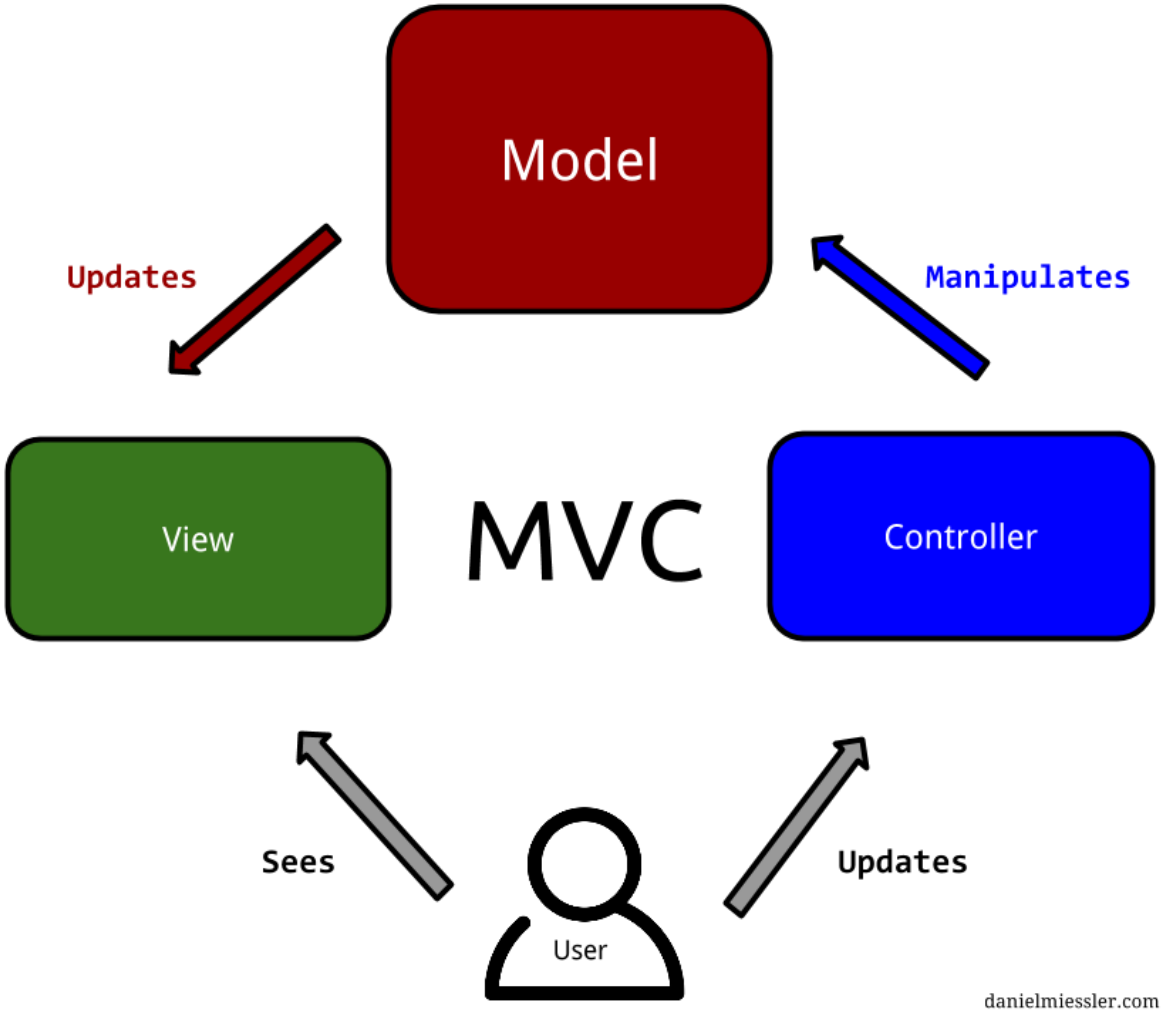
作用于某个对象群中各个对象的操作. 它可以使你在不改变这些对象本身的情况下,定义作用于这些对象的新操作.

- javax.lang.model.element.Element 和javax.lang.model.element.ElementVisitor
- javax.lang.model.type.TypeMirror 和javax.lang.model.type.TypeVisitor

mvc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。

- **Model（模型）** - 模型代表一个存取数据的对象或 JAVA POJO。它也可以带有逻辑，在数据变化时更新控制器。
- **View（视图）** - 视图代表模型包含的数据的可视化。
- **Controller（控制器）** - 控制器作用于模型和视图上。它控制数据流向模型对象，并在数据变化时更新视图。它使视图与模型分离开。



拦截过滤器模式

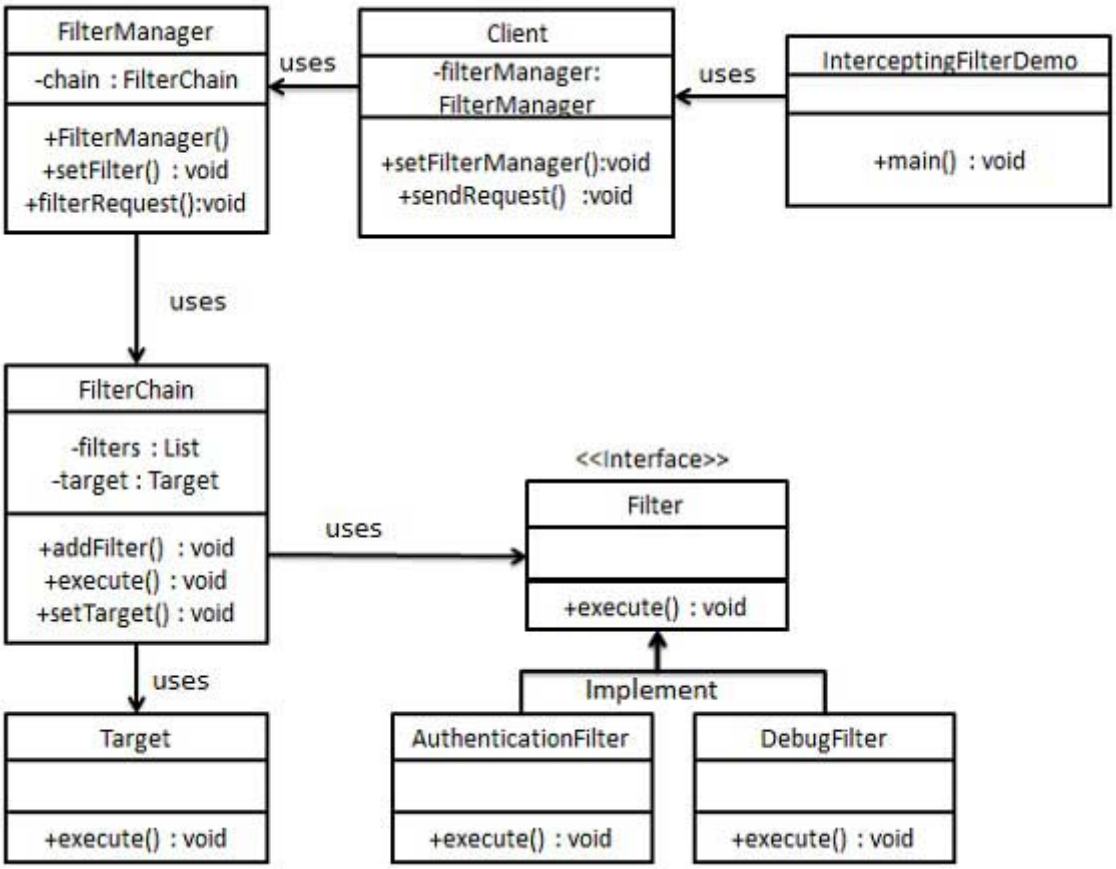
拦截过滤器模式（Intercepting Filter Pattern）用于对应用程序的请求或响应做一些预处理/后处理。定义过滤器，并在把请求传给实际目标应用程序之前应用在请求上。过滤器可以做认证/授权/记录日志，或者跟踪请求，然后把请求传给相应的处理程序。以下是这种设计模式的实体。

- **过滤器（Filter）** - 过滤器在请求处理程序执行请求之前或之后，执行某些任务。
- **过滤器链（Filter Chain）** - 过滤器链带有多个过滤器，并在 Target 上按照定义的顺序执行这些过滤器。
- **Target** - Target 对象是请求处理程序。
- **过滤管理器（Filter Manager）** - 过滤管理器管理过滤器和过滤器链。
- **客户端（Client）** - Client 是向 Target 对象发送请求的对象。

实现

我们将创建 *FilterChain*、*FilterManager*、*Target*、*Client* 作为表示实体的各种对象。*AuthenticationFilter* 和 *DebugFilter* 表示实体过滤器。

InterceptingFilterDemo，我们的演示类使用 *Client* 来演示拦截过滤器设计模式。



步骤 1

创建过滤器接口 Filter。

Filter.java

```
public interface Filter {

    void execute(String request);

}
12345
```

步骤 2

创建实体过滤器。

AuthenticationFilter.java

```
public class AuthenticationFilter implements Filter {

    @Override
    public void execute(String request){
        System.out.println("Authenticating request: " + request);
    }

}
```

DebugFilter.java

```
public class DebugFilter implements Filter {

    @Override
    public void execute(String request){
        System.out.println("request log: " + request);
    }

}
```

步骤 3

创建 Target。

Target.java

```
 */
public class Target {

    public void execute(String request){
        System.out.println("Executing request: " + request);
    }

}
```

步骤 4

创建过滤器链。

FilterChain.java

```
public class FilterChain {

    private List<Filter> filters = new ArrayList();
    private Target target;

    public void addFilter(Filter filter){
        filters.add(filter);
    }

    public void execute(String request){
        for (Filter filter : filters) {
            filter.execute(request);
        }
        target.execute(request);
    }

    public void setTarget(Target target){
        this.target = target;
    }

}
```

步骤 5

创建过滤管理器。

FilterManager.java

```
public class FilterManager {

    FilterChain filterChain;

    public FilterManager(Target target){
        filterChain = new FilterChain();
        filterChain.setTarget(target);
    }
    public void setFilter(Filter filter){
        filterChain.addFilter(filter);
    }

    public void filterRequest(String request){
        filterChain.execute(request);
    }

}
```

步骤 6

创建客户端 Client。

Client.java

```
public class Client {

    FilterManager filterManager;

    public void setFilterManager(FilterManager filterManager){
        this.filterManager = filterManager;
    }

    public void sendRequest(String request){
        filterManager.filterRequest(request);
    }

}
```

步骤 7

使用 *Client* 来演示拦截过滤器设计模式。

InterceptingFilterDemo.java

```
public class InterceptingFilterPatternDemo {

    public static void main(String[] args) {
        FilterManager filterManager = new FilterManager(new Target());
        filterManager.setFilter(new AuthenticationFilter());
        filterManager.setFilter(new DebugFilter());
```

```
        Client client = new Client();
        client.setFilterManager(filterManager);
        client.sendRequest("HOME");
    }

}
```

步骤 8

验证输出。

```
Authenticating request: HOME
request log: HOME
Executing request: HOME
```

单例模式

1.什么是单例

- 保证一个类只有一个实例，并且提供一个访问该全局访问点

2.那些地方用到了单例模式

1. 网站的计数器，一般也是采用单例模式实现，否则难以同步。
2. 应用程序的日志应用，一般都是单例模式实现，只有一个实例去操作才好，否则内容不好追加显示。
3. 多线程的线程池的设计一般也是采用单例模式，因为线程池要方便对池中的线程进行控制
4. Windows的（任务管理器）就是很典型的单例模式，他不能打开俩个
5. windows的（回收站）也是典型的单例应用。在整个系统运行过程中，回收站只维护一个实例。

3.单例优缺点

优点：

1. 在单例模式中，活动的单例只有一个实例，对单例类的所有实例化得到的都是相同的一个实例。这样就防止其它对象对自己的实例化，确保所有的对象都访问一个实例
2. 单例模式具有一定的伸缩性，类自己来控制实例化进程，类就在改变实例化进程上有相应的伸缩性。
3. 提供了对唯一实例的受控访问。
4. 由于在系统内存中只存在一个对象，因此可以节约系统资源，当需要频繁创建和销毁的对象时单例模式无疑可以提高系统的性能。
5. 允许可变数目的实例。
6. 避免对共享资源的多重占用。

缺点：

1. 不适用于变化的对象，如果同一类型的对象总是要在不同的用例场景发生变化，单例就会引起数据的错误，不能保存彼此的状态。
2. 由于单利模式中没有抽象层，因此单例类的扩展有很大的困难。
3. 单例类的职责过重，在一定程度上违背了“单一职责原则”。

4. 滥用单例将带来一些负面问题，如为了节省资源将数据库连接池对象设计为的单例类，可能会导致共享连接池对象的程序过多而出现连接池溢出；如果实例化的对象长时间不被利用，系统会认为是垃圾而被回收，这将导致对象状态的丢失。

4.单例模式使用注意事项：

1. 使用时不能用反射模式创建单例，否则会实例化一个新的对象
2. 使用懒单例模式时注意线程安全问题
3. 饿单例模式和懒单例模式构造方法都是私有的，因而是不能被继承的，有些单例模式可以被继承（如登记式模式）

5.单例防止反射漏洞攻击

```
private static boolean flag = false;

private Singleton() {

    if (flag == false) {
        flag = !flag;
    } else {
        throw new RuntimeException("单例模式被侵犯！");
    }
}

public static void main(String[] args) {

}
```

6.如何选择单例创建方式

- 如果不需要延迟加载单例，可以使用枚举或者饿汉式，相对来说枚举性好于饿汉式。
  如果需要延迟加载，可以使用静态内部类或者懒汉式，相对来说静态内部类好于懒韩式。
  最好使用饿汉式

7.单例创建方式

主要使用懒汉和懒汉式

1. 饿汉式:

   类初始化时,会立即加载该对象，线程天生安全,调用效率高。

2. 懒汉式:

   类初始化时,不会初始化该对象,真正需要使用的时候才会创建该对象,具备懒加载功能。

3. 静态内部方式:

   结合了懒汉式和饿汉式各自的优点，真正需要对象的时候才会加载，加载类是线程安全的。

4. 枚举单例:

   使用枚举实现单例模式

   优点: 实现简单、调用效率高，枚举本身就是单例， 由jvm从根本上提供保障!避免通过反射和反序列化的漏洞;

   缺点: 没有延迟加载。

5. 双重检测锁方式

   因为JVM重排序、内存可见性的原因，可能会初始化多次，

所以： 需要通过 Double Check 双重检查+ synchronized + Volatile 解决 同步问题和可见性问题。

1.饿汉式

类初始化时,会立即加载该对象，线程天生安全,调用效率高。

```
package com.crazymakercircle.designmodel.singleton;
//饿汉式
public class FSingleton {

    // 类初始化时,会立即加载该对象，线程安全,调用效率高
    private static final FSingleton instance = new FSingleton();

    // 私有化构造方法
    private FSingleton() {
    }

 public   static FSingleton getInstance() {
        return instance;
    }


}
```

饿汉模式就是类一旦加载，就把单例初始化完成，保证getInstance的时候，单例是已经存在的了。

特点：

- 是否 Lazy 初始化：否
- 是否多线程安全：是
- 实现难度：易

优点：

- 没有加锁，执行效率会提高。
- 这种方式比较常用，但容易产生垃圾对象
- 它基于JVM class loader 机制, 是单线程执行的， 避免了多线程的同步问题

缺点：

- 类加载时就初始化，浪费内存，

2.懒汉式

类初始化时,不会初始化该对象,

真正需要使用的时候，才会创建该对象,具备懒加载功能。

```
package com.crazymakercircle.designmodel.singleton;
//懒汉模式
public class FLazySingleton {
```

```
    //类初始化时，不会初始化该对象，真正需要使用的时候才会创建该对象。
    private static  FLazySingleton instance = null;

    // 私有化构造方法
    private FLazySingleton() {
    }

    //真正需要使用的时候才会创建该对象
    public static synchronized FLazySingleton getInstance() {
        if(null==instance)
        {
            instance=new FLazySingleton();
        }
        return instance;
    }


}
```

3.静态内部类

静态内部方式:

结合了懒汉式和饿汉式各自的优点，真正需要对象的时候才会加载，加载类是线程安全的。

```
package com.crazymakercircle.designmodel.singleton;

public class Singleton {
    //静态内部类
    private static class LazyHolder {
          //通过final保障初始化时的线程安全
           private static final Singleton INSTANCE = new Singleton();
    }
       //私有的构造器
    private Singleton (){}
      //获取单例的方法
    public static final Singleton getInstance() {
      //返回内部类的静态、最终成员
       return LazyHolder.INSTANCE;
    }
}
```

4.枚举单例式

枚举单例:

使用枚举实现单例模式 优点:实现简单、调用效率高，

枚举本身就是单例，由jvm从根本上提供保障!避免通过反射和反序列化的漏洞， 缺点没有延迟加载。

```
package com.lijie;


package com.crazymakercircle.designmodel.singleton;
//饿汉式
```

```
public enum SingletonEnumStyle {
    INSTANCE;
    // 类初始化时,会立即加载该对象，线程安全,调用效率高

    public  static SingletonEnumStyle getInstance() {
        return INSTANCE;
    }

}
```

枚举实现单例模式 优点:

- 实现简单、枚举本身就是单例，由jvm从根本上提供保障!
- 避免通过反射和反序列化的漏洞

缺点:

- 没有延迟加载

5.双重检测锁方式

所谓懒加载，就是直到第一次被调用时才加载。其实现需要考虑并发问题和指令重排，代码如下:

```
public class Singleton {

    private volatile static Singleton instance; //①

    private Singleton() { //②
    }

    public static Singleton getInstance() {
        if (instance == null) {//③
            synchronized (Singleton.class) {
                if (instance == null) {//④
                    instance = new Singleton();//⑤
                }
            }
        }
        return instance;
    }
}
```

这段代码精简至极，没有一个字符是多余的，下面逐行解读一下:

首先，注意到①处的volatile关键字，它具备两项特性:

> 一是保证此变量对于所有线程的可见性。
>
> 即当一条线程修改了这个变量的值，新值对于其他线程来说是可以立即得知的。
>
> 二是禁止指令重排序优化。

这里解释一下指令重排序优化:

代码 ⑤ 处的instance = new Singleton(); 并不是原子的，大体可分为如下 3 步:

1. 分配内存
2. 调用构造函数初始化成实例
3. 让instance指向分配的内存空间

JVM 允许在保证结果正确的前提下进行指令重排序优化。

即如上 3 步可能的顺序为1->2->3 或 1->3->2 。

如果顺序是 1->3->2 ，当 3 执行完，2 还未执行时，另一个线程执行到代码 ③ 处，发现instance不为null，直接返回还未初始化好的instance并使用，就会报错。

所以使用volatile，就是为了保证线程间的可见性和防止指令重排。

其次，代码②处将构造函数声明为private目的：在于阻止使用new Singleton()这样的代码生成新实例。

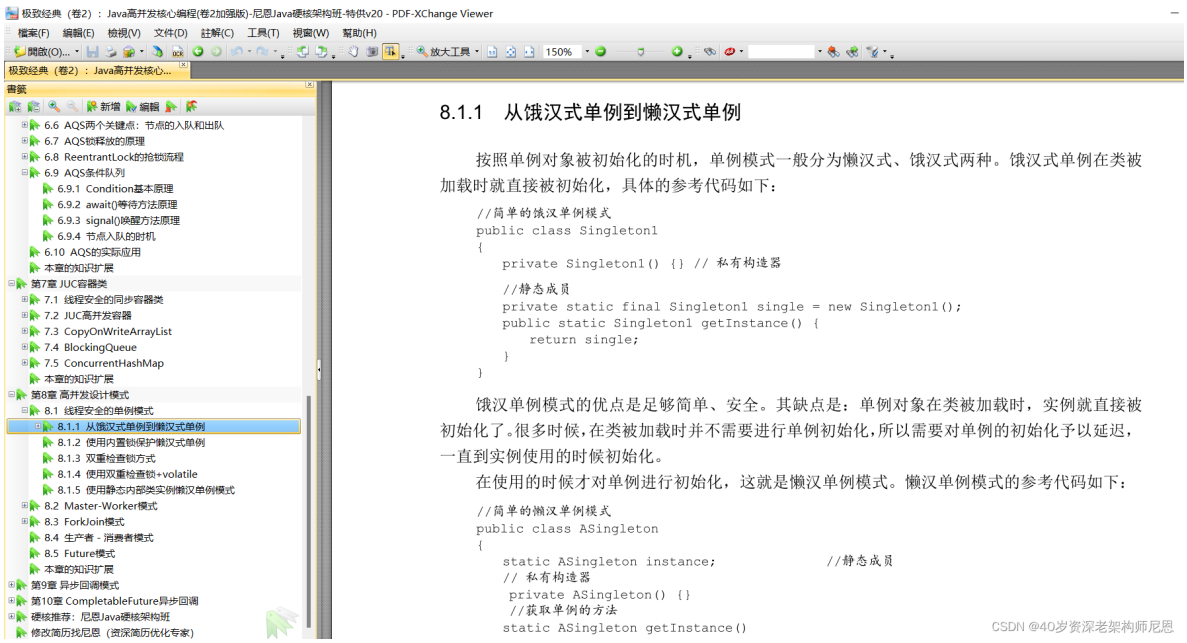
最后，当客户端调用Singleton.getInstance()时，先检查是否已经实例化(代码③)，未实例化时同步代码块，然后再次检查是否已实例化(代码④)，然后才执行代码⑤。

两次检查的意义在于，防止synchronized同步过程中其他线程进行了实例化。

这就是著名的双重检查锁(Double check lock)实现单例，也即懒加载。

> TIPS:
>
> 网上也有直接对 getInstance()方法加锁的版本，这样大范围的方法级别加锁会导致并发变低，实际上第一次调用生成实例之后，后续获取实例根本不需要并发控制了。
>
> 而本例的双重检查锁版本可以避免此并发问题。

双重检测锁 单例 非常重要， 涉及到Volatile 和可见性的底层原理， 深入学习/系统学习 双重检测锁 单例的内容， 请参见 《Java 高并发核心编程 卷2》 **第8.1节：线程安全的单例模式**

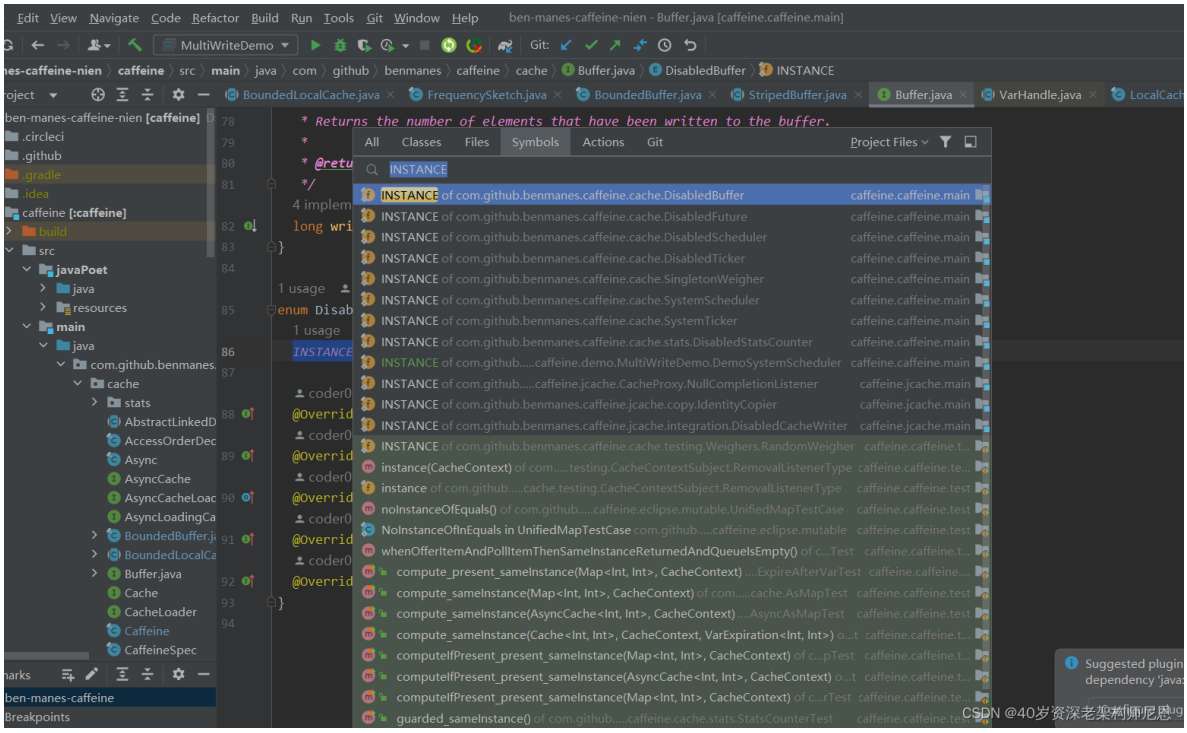


缓存之王 Caffeine 源码中，如何使用单例模式的?

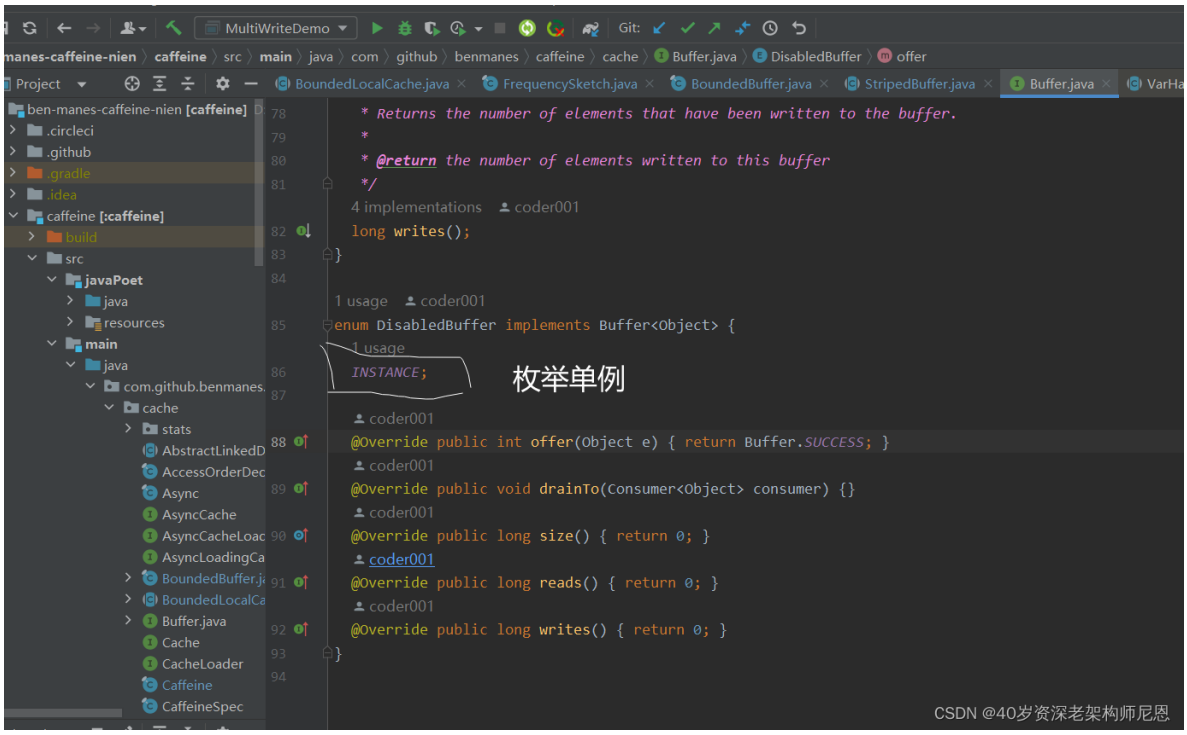
答案是：枚举单例

并且，单例的名称叫做 INSTANCE

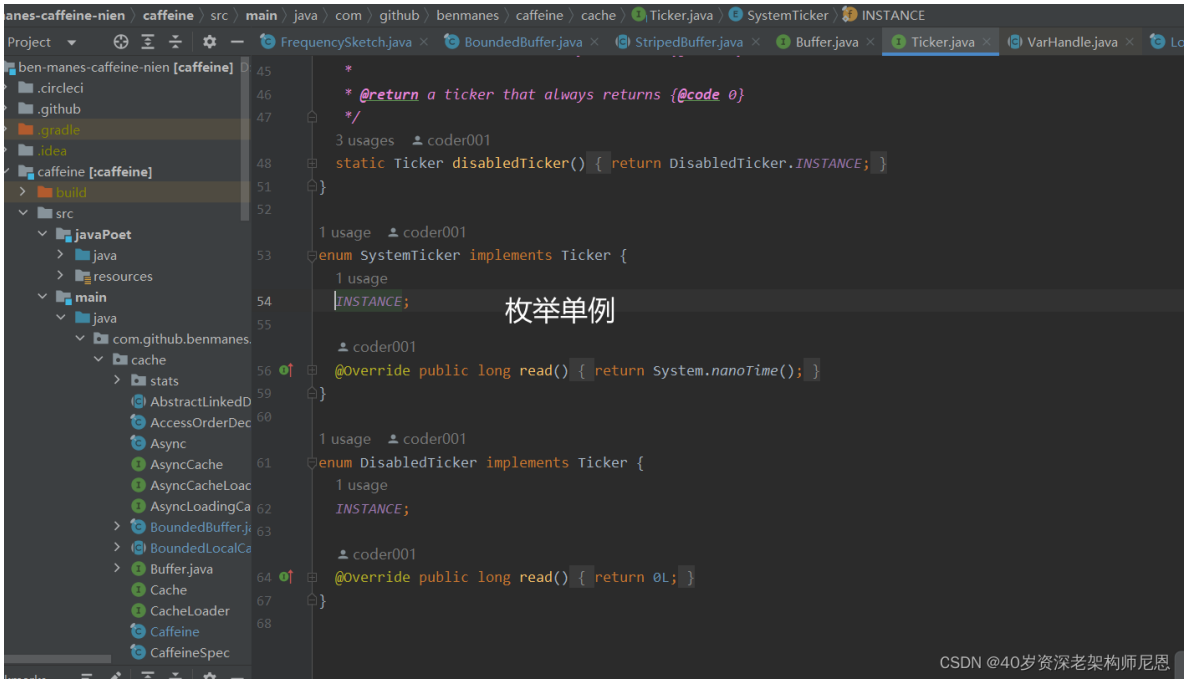
通过这个 INSTANCE 名字 做 关键词搜索， 能搜到一大把



来一个案例



再来一个案例



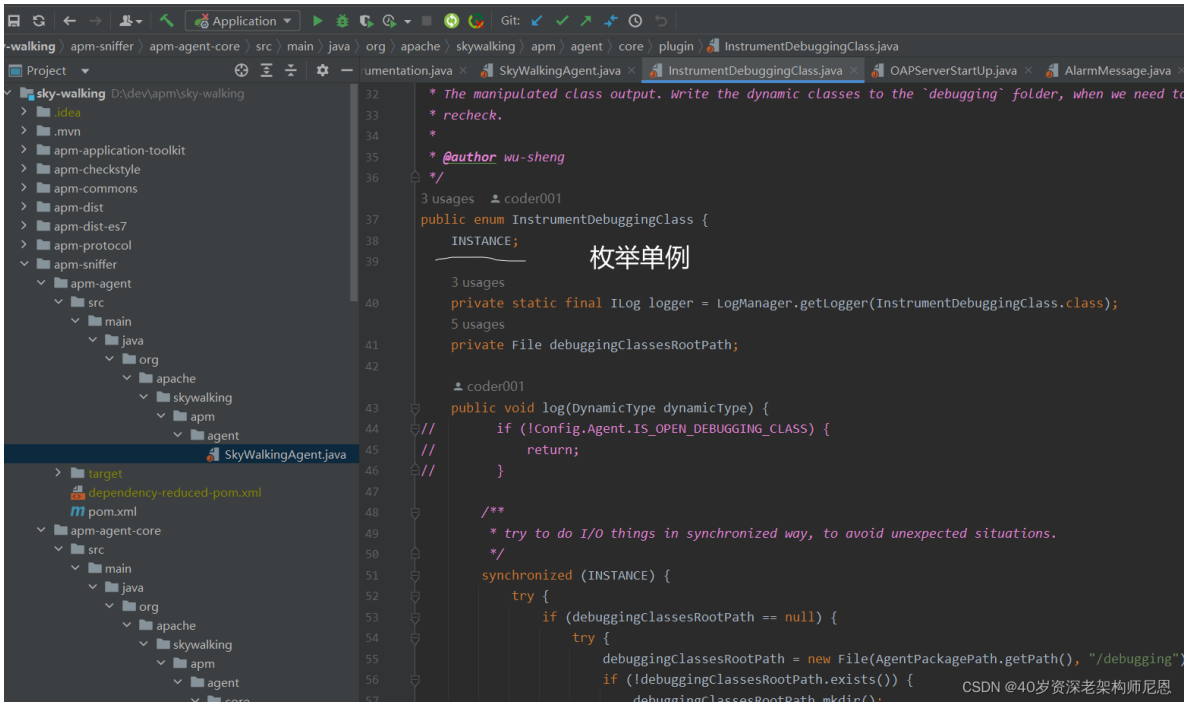
缓存之王 Caffeine 的详细资料，请参考下面的博客、或者对应的PDF文件：

- 《彻底穿透 缓存之王：Caffeine 源码、架构、原理（史上最全，10W字 超级长文）》
- 《彻底穿透 缓存之王：Caffeine 的使用（史上最全）》

链路之王 Skywalking 源码中，如何使用单例模式的?

答案是：枚举单例

并且，单例的名称叫做 INSTANCE



8 单例模式懒汉式和饿汉式有哪些区别？(美团)

单例模式是 Java 中最简单的设计模式之一。这种类型的设计模式属于创建型模式，它提供了一种创建对象的最佳方式。
这种模式涉及到一个单一的类，该类负责创建自己的对象，同时确保只有单个对象被创建。这个类提供了一种访问其唯一的对象的方式，可以直接访问，不需要实例化该类的对象。

注意：
1、单例类只能有一个实例。
2、单例类必须自己创建自己的唯一实例。
3、单例类必须给所有其他对象提供这一实例。

明确定义后，看一下代码：

饿汉模式

```
package com.crazymakercircle.designmodel.singleton;
//饿汉式
public class FSingleton {

    // 类初始化时,会立即加载该对象，线程安全,调用效率高
    private static final FSingleton instance = new FSingleton();

    // 私有化构造方法
    private FSingleton() {
    }

 public   static FSingleton getInstance() {
        return instance;
    }


}
```

饿汉模式就是类一旦加载，就把单例初始化完成，保证getInstance的时候，单例是已经存在的了。

特点：

- 是否 Lazy 初始化：否
- 是否多线程安全：是
- 实现难度：易

优点：

- 没有加锁，执行效率会提高。
- 这种方式比较常用，但容易产生垃圾对象
- 它基于JVM class loader 机制, 是单线程执行的， 避免了多线程的同步问题

缺点：

- 类加载时就初始化，浪费内存，

懒汉模式

```
public class Singleton {

    private volatile static Singleton instance; //①

    private Singleton() { //②
    }

    public static Singleton getInstance() {
        if (instance == null) {//③
            synchronized (Singleton.class) {
                if (instance == null) {//④
                    instance = new Singleton();//⑤
                }
            }
        }
        return instance;
    }
}
```

而懒汉比较懒，只有当调用getInstance的时候，才回去初始化这个单例。

特点：

- 是否 Lazy 初始化：是
- 是否多线程安全：是
- 实现难度：难

1、线程安全：

饿汉式天生就是线程安全的，可以直接用于多线程而不会出现问题，

懒汉式本身是非线程安全的，需要通过多种手段，保证线程安全和内存可见性：

- volatile 保证内存可见性
- synchronized + 双重检查 保证线程安全

2、资源加载和性能：

饿汉式在类创建的同时就实例化一个静态对象出来，不管之后会不会使用这个单例，都会占据一定的内存，但是相应的，在第一次调用时速度也会更快，因为其资源已经初始化完成。

而懒汉式顾名思义，会延迟加载，在第一次使用该单例的时候才会实例化对象出来，第一次调用时要做初始化，如果要做的工作比较多，性能上会有些延迟，之后就和饿汉式一样了。

- 意图：保证一个类仅有一个实例，并提供一个访问它的全局访问点。
- 主要解决：一个全局使用的类频繁地创建与销毁。
- 何时使用：当您想控制实例数目，节省系统资源的时候。
- 如何解决：判断系统是否已经有这个单例，如果有则返回，如果没有则创建。

- 关键代码：构造函数是私有的。
- 应用实例：

  1、一个党只能有一个主席。
  2、Windows是多进程多线程的，在操作一个文件的时候，就不可避免地出现多个进程或线程同时操作一个文件的现象，所以所有文件的处理必须通过唯一的实例来进行。
  3、一些设备管理器常常设计为单例模式，比如一个电脑有两台打印机，在输出的时候就要处理不能两台打印机打印同一个文件。

优点：

1、在内存里只有一个实例，减少了内存的开销，尤其是频繁的创建和销毁实例（比如管理学院首页页面缓存）。
2、避免对资源的多重占用（比如写文件操作）。

缺点：没有接口，不能继承，与单一职责原则冲突，一个类应该只关心内部逻辑，而不关心外面怎么样来实例化。

使用场景：

1、要求生产唯一序列号。
2、WEB 中的计数器，不用每次刷新都在数据库里加一次，用单例先缓存起来。
3、创建的一个对象需要消耗的资源过多，比如 I/O 与数据库的连接等。

注意事项：getInstance() 方法中需要使用 Double Check 双重检查锁,synchronized (Singleton.class) 防止多线程同时进入造成instance 被多次实例化。

工厂模式

1.什么是工厂模式

- 它提供了一种创建对象的最佳方式。在工厂模式中，我们在创建对象时不会对客户端暴露创建逻辑，并且是通过使用一个共同的接口来指向新创建的对象。实现了创建者和调用者分离，工厂模式分为简单工厂、工厂方法、抽象工厂模式

2.工厂模式好处

- 工厂模式是我们最常用的实例化对象模式了，是用工厂方法代替new操作的一种模式。
- 利用工厂模式可以降低程序的耦合性，为后期的维护修改提供了很大的便利。
- 将选择实现类、创建对象统一管理和控制。从而将调用者跟我们的实现类解耦。

3.为什么要学习工厂设计模式

- 不知道你们面试题问到过源码没有，你知道Spring的源码吗，MyBatis的源码吗，等等等
  如果你想学习很多框架的源码，或者你想自己开发自己的框架，就必须先掌握设计模式（工厂设计模式用的是非常非常广泛的）

4.Spring开发中的工厂设计模式

1.Spring IOC

- 看过Spring源码就知道，在Spring IOC容器创建bean的过程是使用了工厂设计模式
- Spring中无论是通过xml配置还是通过配置类还是注解进行创建bean，大部分都是通过简单工厂来进行创建的。
- 当容器拿到了beanName和class类型后，动态的通过反射创建具体的某个对象，最后将创建的对象放到Map中。

2.为什么Spring IOC要使用工厂设计模式创建Bean呢

- 在实际开发中，如果我们A对象调用B，B调用C，C调用D的话我们程序的耦合性就会变高。（耦合大致分为类与类之间的依赖，方法与方法之间的依赖。）
- 在很久以前的三层架构编程时，都是控制层调用业务层，业务层调用数据访问层时，都是是直接new对象，耦合性大大提升，代码重复量很高，对象满天飞
- 为了避免这种情况，Spring使用工厂模式编程，写一个工厂，由工厂创建Bean，以后我们如果要对象就直接管工厂要就可以，剩下的事情不归我们管了。Spring IOC容器的工厂中有个静态的Map集合，是为了让工厂符合单例设计模式，即每个对象只生产一次，生产出对象后就存入到Map集合中，保证了实例不会重复影响程序效率。

5.工厂模式分类

- 工厂模式分为简单工厂、工厂方法、抽象工厂模式

```
简单工厂 ：用来生产同一等级结构中的任意产品。（不支持拓展增加产品）
工厂方法 ：用来生产同一等级结构中的固定产品。（支持拓展增加产品）
抽象工厂 ：用来生产不同产品族的全部产品。（不支持拓展增加产品；支持增加产品族）
123
我下面来使用代码演示一下：
```

5.1 简单工厂模式

什么是简单工厂模式

- 简单工厂模式相当于是一个工厂中有各种产品，创建在一个类中，客户无需知道具体产品的名称，只需要知道产品类所对应的参数即可。但是工厂的职责过重，而且当类型过多时不利于系统的扩展维护。

代码演示：

1. 创建工厂

```
package com.lijie;

public interface Car {
    public void run();
}
```

1. 创建工厂的产品（宝马）

```
package com.lijie;

public class Bmw implements Car {
    public void run() {
        System.out.println("我是宝马汽车...");
    }
}
```

1. 创建工另外一种产品（奥迪）

```
package com.lijie;

public class AoDi implements Car {
    public void run() {
        System.out.println("我是奥迪汽车..");
    }
}
```

1. 创建核心工厂类，由他决定具体调用哪产品

```
package com.lijie;

public class CarFactory {

     public static Car createCar(String name) {
        if ("".equals(name)) {
             return null;
        }
        if(name.equals("奥迪")){
            return new AoDi();
        }
        if(name.equals("宝马")){
            return new Bmw();
        }
        return null;
    }
}
```

1. 演示创建工厂的具体实例

```
package com.lijie;

public class ClientO1 {

    public static void main(String[] args) {
        Car aodi  =CarFactory.createCar("奥迪");
        Car bmw  =CarFactory.createCar("宝马");
        aodi.run();
        bmw.run();
    }
}
```

单工厂的优点/缺点

- 优点：简单工厂模式能够根据外界给定的信息，决定究竟应该创建哪个具体类的对象。明确区分了各自的职责和权力，有利于整个软件体系结构的优化。
- 缺点：很明显工厂类集中了所有实例的创建逻辑，容易违反GRASPR的高内聚的责任分配原则

5.2 工厂方法模式

什么是工厂方法模式

- 工厂方法模式Factory Method，又称多态性工厂模式。在工厂方法模式中，核心的工厂类不再负责所有的产品的创建，而是将具体创建的工作交给子类去做。该核心类成为一个抽象工厂角色，仅

负责给出具体工厂子类必须实现的接口，而不接触哪一个产品类应当被实例化这种细节

代码演示：

1. 创建工厂

```
package com.lijie;

public interface Car {
    public void run();
}
```

1. 创建工厂方法调用接口（所有的产品需要new出来必须继承他来实现方法）

```
package com.lijie;

public interface CarFactory {

    Car createCar();

}
```

1. 创建工厂的产品（奥迪）

```
package com.lijie;

public class AoDi implements Car {
    public void run() {
        System.out.println("我是奥迪汽车..");
    }
}
```

1. 创建工厂另外一种产品（宝马）

```
package com.lijie;

public class Bmw implements Car {
    public void run() {
        System.out.println("我是宝马汽车...");
    }
}
```

1. 创建工厂方法调用接口的实例（奥迪）

```
package com.lijie;

public class AoDiFactory implements CarFactory {

    public Car createCar() {

        return new AoDi();
    }
}
```

1. 创建工厂方法调用接口的实例（宝马）

```
package com.lijie;

public class BmwFactory implements CarFactory {

    public Car createCar() {

        return new Bmw();
    }

}
```

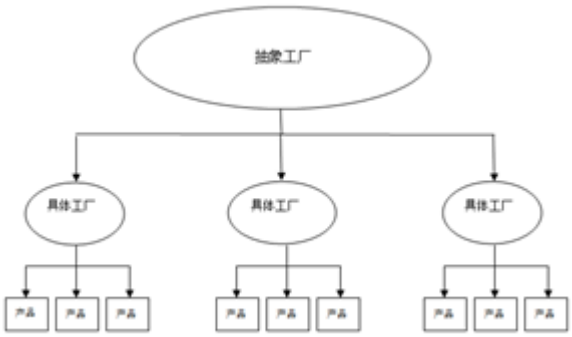
1. 演示创建工厂的具体实例

```
package com.lijie;

public class Client {

    public static void main(String[] args) {
        Car aodi = new AoDiFactory().createCar();
        Car jili = new BmwFactory().createCar();
        aodi.run();
        jili.run();
    }
}
```

5.3 抽象工厂模式

什么是抽象工厂模式

- 抽象工厂简单地说是工厂的工厂，抽象工厂可以创建具体工厂，由具体工厂来产生具体产品。



代码演示：

1. 创建第一个子工厂，及实现类

```
package com.lijie;

//汽车
public interface Car {
       void run();
}

 class CarA implements Car{

    public void run() {
        System.out.println("宝马");
    }
```

```
}
 class CarB implements Car{

    public void run() {
        System.out.println("摩拜");
    }

}
```

1. 创建第二个子工厂，及实现类

```
package com.lijie;

//发动机
public interface Engine {

    void run();

}

class EngineA implements Engine {

    public void run() {
        System.out.println("转的快!");
    }

}

class EngineB implements Engine {

    public void run() {
        System.out.println("转的慢!");
    }

}
```

1. 创建一个总工厂，及实现类（由总工厂的实现类决定调用那个工厂的那个实例）

```
package com.lijie;

public interface TotalFactory {
    // 创建汽车
    Car createChair();
    // 创建发动机
    Engine createEngine();
}

//总工厂实现类，由他决定调用哪个工厂的那个实例
class TotalFactoryReally implements TotalFactory {

    public Engine createEngine() {

        return new EngineA();
    }
```

```
    public Car createChair() {

        return new CarA();
    }
}
```

1. 运行测试

```
package com.lijie;

public class Test {

    public static void main(String[] args) {
        TotalFactory totalFactory2 = new TotalFactoryReally();
        Car car = totalFactory2.createChair();
        car.run();

        TotalFactory totalFactory = new TotalFactoryReally();
        Engine engine = totalFactory.createEngine();
        engine.run();
    }
}
```

代理模式

1.什么是代理模式

- 通过代理控制对象的访问，可以在这个对象调用方法之前、调用方法之后去处理/添加新的功能。(也就是AO的P微实现)
- 代理在原有代码乃至原业务流程都不修改的情况下，直接在业务流程中切入新代码，增加新功能，这也和Spring的（面向切面编程）很相似

2.代理模式应用场景

- Spring AOP、日志打印、异常处理、事务控制、权限控制等

3.代理的分类

- 静态代理(静态定义代理类)
- 动态代理(动态生成代理类，也称为Jdk自带动态代理)
- Cglib 、javaassist（字节码操作库）

4.三种代理的区别

1. 静态代理：简单代理模式，是动态代理的理论基础。常见使用在代理模式
2. jdk动态代理：使用反射完成代理。需要有顶层接口才能使用，常见是mybatis的mapper文件是代理。
3. cglib动态代理：也是使用反射完成代理，可以直接代理类（jdk动态代理不行），使用字节码技术，不能对 final类进行继承。（需要导入jar包）

5.用代码演示三种代理

5.1.静态代理

什么是静态代理

- 由程序员创建或工具生成代理类的源码，再编译代理类。所谓静态也就是在程序运行前就已经存在代理类的字节码文件，代理类和委托类的关系在运行前就确定了。

代码演示：

- 有一段这样的代码：（如何能在不修改UserDao接口类的情况下开事务和关闭事务呢）

```
package com.lijie;

//接口类
public class UserDao{
    public void save() {
        System.out.println("保存数据方法");
    }
}
12345678
package com.lijie;

//运行测试类
public  class Test{
    public static void main(String[] args) {
        UserDao userDao = new UserDao();
        userDao.save();
    }
}
```

修改代码，添加代理类

```
package com.lijie;

//代理类
public class UserDaoProxy extends UserDao {
    private UserDao userDao;

    public UserDaoProxy(UserDao userDao) {
        this.userDao = userDao;
    }

    public void save() {
        System.out.println("开启事物...");
        userDao.save();
        System.out.println("关闭事物...");
    }

}
1234567891011121314151617
//添加完静态代理的测试类
public class Test{
    public static void main(String[] args) {
        UserDao userDao = new UserDao();
        UserDaoProxy userDaoProxy = new UserDaoProxy(userDao);
        userDaoProxy.save();
    }
}
```

- 缺点：每个需要代理的对象都需要自己重复编写代理，很不舒服，
- 优点：但是可以面相实际对象或者是接口的方式实现代理

2.2.动态代理

什么是动态代理

- 动态代理也叫做，JDK代理、接口代理。
- 动态代理的对象，是利用JDK的API，动态的在内存中构建代理对象（是根据被代理的接口来动态生成代理类的class文件，并加载运行的过程），这就叫动态代理

```
package com.lijie;

//接口
public interface UserDao {
    void save();
}
123456
package com.lijie;

//接口实现类
public class UserDaoImpl implements UserDao {
    public void save() {
        System.out.println("保存数据方法");
    }
}
```

- //下面是代理类，可重复使用，不像静态代理那样要自己重复编写代理

```
package com.lijie;

import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;

// 每次生成动态代理类对象时,实现了InvocationHandler接口的调用处理器对象
public class InvocationHandlerImpl implements InvocationHandler {

    // 这其实业务实现类对象，用来调用具体的业务方法
    private Object target;

    // 通过构造函数传入目标对象
    public InvocationHandlerImpl(Object target) {
        this.target = target;
    }

    //动态代理实际运行的代理方法
    public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {
        System.out.println("调用开始处理");
        //下面invoke()方法是以反射的方式来创建对象，第一个参数是要创建的对象，第二个是构成方法的参数，由第二个参数来决定创建对象使用哪个构造方法
        Object result = method.invoke(target, args);
        System.out.println("调用结束处理");
        return result;
    }
}
```

- //利用动态代理使用代理方法

```
package com.lijie;

import java.lang.reflect.Proxy;

public class Test {
    public static void main(String[] args) {
        // 被代理对象
        UserDao userDaoImpl = new UserDaoImpl();
        InvocationHandlerImpl invocationHandlerImpl = new
InvocationHandlerImpl(userDaoImpl);

        //类加载器
        ClassLoader loader = userDaoImpl.getClass().getClassLoader();
        Class<?>[] interfaces = userDaoImpl.getClass().getInterfaces();

        // 主要装载器、一组接口及调用处理动态代理实例
        UserDao newProxyInstance = (UserDao) Proxy.newProxyInstance(loader,
interfaces, invocationHandlerImpl);
        newProxyInstance.save();
    }
}
```

- 缺点：必须是面向接口，目标业务类必须实现接口
- 优点：不用关心代理类，只需要在运行阶段才指定代理哪一个对象

5.3.CGLIB动态代理

CGLIB动态代理原理：

- 利用asm开源包，对代理对象类的class文件加载进来，通过修改其字节码生成子类来处理。

什么是CGLIB动态代理

- CGLIB动态代理和jdk代理一样，使用反射完成代理，不同的是他可以直接代理类（jdk动态代理不行，他必须目标业务类必须实现接口），CGLIB动态代理底层使用字节码技术，CGLIB动态代理不能对 final类进行继承。（CGLIB动态代理需要导入jar包）

代码演示：

```
package com.lijie;

//接口
public interface UserDao {
    void save();
}
123456
package com.lijie;

//接口实现类
public class UserDaoImpl implements UserDao {
    public void save() {
        System.out.println("保存数据方法");
    }
}
12345678
package com.lijie;
```

```
import org.springframework.cglib.proxy.Enhancer;
import org.springframework.cglib.proxy.MethodInterceptor;
import org.springframework.cglib.proxy.MethodProxy;
import java.lang.reflect.Method;

//代理主要类
public class CglibProxy implements MethodInterceptor {
    private Object targetObject;
    // 这里的目标类型为Object，则可以接受任意一种参数作为被代理类，实现了动态代理
    public Object getInstance(Object target) {
        // 设置需要创建子类的类
        this.targetObject = target;
        Enhancer enhancer = new Enhancer();
        enhancer.setSuperclass(target.getClass());
        enhancer.setCallback(this);
        return enhancer.create();
    }

    //代理实际方法
    public Object intercept(Object obj, Method method, Object[] args,
MethodProxy proxy) throws Throwable {
        System.out.println("开启事物");
        Object result = proxy.invoke(targetObject, args);
        System.out.println("关闭事物");
        // 返回代理对象
        return result;
    }
}


package com.lijie;

//测试CGLIB动态代理
public class Test {
    public static void main(String[] args) {
        CglibProxy cglibProxy = new CglibProxy();
        UserDao userDao = (UserDao) cglibProxy.getInstance(new UserDaoImpl());
        userDao.save();
    }
}
```

建造者模式

1.什么是建造者模式

- 建造者模式：是将一个复杂的对象的构建与它的表示分离，使得同样的构建过程可以创建不同的方式进行创建。
- 工厂类模式是提供的是创建单个类的产品
- 而建造者模式则是将各种产品集中起来进行管理，用来具有不同的属性的产品

建造者模式通常包括下面几个角色：

1. uilder：给出一个抽象接口，以规范产品对象的各个组成成分的建造。这个接口规定要实现复杂对象的哪些部分的创建，并不涉及具体的对象部件的创建。

2. ConcreteBuilder：实现Builder接口，针对不同的商业逻辑，具体化复杂对象的各部分的创建。在建造过程完成后，提供产品的实例。
3. Director：调用具体建造者来创建复杂对象的各个部分，在指导者中不涉及具体产品的信息，只负责保证对象各部分完整创建或按某种顺序创建。
4. Product：要创建的复杂对象。

2.建造者模式的使用场景

使用场景：

1. 需要生成的对象具有复杂的内部结构。
2. 需要生成的对象内部属性本身相互依赖。

- 与工厂模式的区别是：建造者模式更加关注与零件装配的顺序。
- JAVA 中的 StringBuilder就是建造者模式创建的，他把一个单个字符的char数组组合起来
- Spring不是建造者模式，它提供的操作应该是对于字符串本身的一些操作，而不是创建或改变一个字符串。

3.代码案例

1. 建立一个装备对象Arms

```
package com.lijie;

//装备类
public class Arms {
    //头盔
    private String helmet;
    //铠甲
    private String armor;
    //武器
    private String weapon;

    //省略Git和Set方法...........
}
```

1. 创建Builder接口（给出一个抽象接口，以规范产品对象的各个组成成分的建造，这个接口只是规范）

```
package com.lijie;

public interface PersonBuilder {

    void builderHelmetMurder();

    void builderArmorMurder();

    void builderWeaponMurder();

    void builderHelmetYanLong();

    void builderArmorYanLong();

    void builderWeaponYanLong();

    Arms BuilderArms(); //组装
```

```
}
```

1. 创建Builder实现类（这个类主要实现复杂对象创建的哪些部分需要什么属性）

```
package com.lijie;

public class ArmsBuilder implements PersonBuilder {
    private Arms arms;

    //创建一个Arms实例,用于调用set方法
    public ArmsBuilder() {
        arms = new Arms();
    }

    public void builderHelmetMurder() {
        arms.setHelmet("夺命头盔");
    }

    public void builderArmorMurder() {
        arms.setArmor("夺命铠甲");
    }

    public void builderWeaponMurder() {
        arms.setWeapon("夺命宝刀");
    }

    public void builderHelmetYanLong() {
        arms.setHelmet("炎龙头盔");
    }

    public void builderArmorYanLong() {
        arms.setArmor("炎龙铠甲");
    }

    public void builderWeaponYanLong() {
        arms.setWeapon("炎龙宝刀");
    }

    public Arms BuilderArms() {
        return arms;
    }
}
```

1. Director（调用具体建造者来创建复杂对象的各个部分，在指导者中不涉及具体产品的信息，只负责保证对象各部分完整创建或按某种顺序创建）

```
package com.lijie;

public class PersonDirector {

    //组装
    public Arms constructPerson(PersonBuilder pb) {
        pb.builderHelmetYanLong();
        pb.builderArmorMurder();
```

```
        pb.builderWeaponMurder();
        return pb.BuilderArms();
    }

    //这里进行测试
    public static void main(String[] args) {
        PersonDirector pb = new PersonDirector();
        Arms arms = pb.constructPerson(new ArmsBuilder());
        System.out.println(arms.getHelmet());
        System.out.println(arms.getArmor());
        System.out.println(arms.getWeapon());
    }
}
```

模板方法模式

1.什么是模板方法

- 模板方法模式：定义一个操作中的算法骨架（父类），而将一些步骤延迟到子类中。
  模板方法使得子类可以不改变一个算法的结构来重定义该算法的

2.什么时候使用模板方法

- 实现一些操作时，整体步骤很固定，但是呢。就是其中一小部分需要改变，这时候可以使用模板方法模式，将容易变的部分抽象出来，供子类实现。

3.实际开发中应用场景哪里用到了模板方法

- 其实很多框架中都有用到了模板方法模式
- 例如：数据库访问的封装、Junit单元测试、servlet中关于doGet/doPost方法的调用等等

4.现实生活中的模板方法

例如：

1. 去餐厅吃饭，餐厅给我们提供了一个模板就是：看菜单，点菜，吃饭，付款，走人
   （这里 “**点菜和付款**” 是不确定的由子类来完成的，其他的则是一个模板。）

5.代码实现模板方法模式

1. 先定义一个模板。把模板中的点菜和付款，让子类来实现。

```
package com.lijie;

//模板方法
public abstract class RestaurantTemplate {

    // 1.看菜单
    public void menu() {
        System.out.println("看菜单");
    }

    // 2.点菜业务
    abstract void spotMenu();

    // 3.吃饭业务
```

```
    public void havingDinner(){ System.out.println("吃饭"); }

    // 3.付款业务
    abstract void payment();

    // 3.走人
    public void GoR() { System.out.println("走人"); }

    //模板通用结构
    public void process(){
        menu();
        spotMenu();
        havingDinner();
        payment();
        GoR();
    }
}
```

1. 具体的模板方法子类 1

```
package com.lijie;

public class RestaurantGinsengImpl extends RestaurantTemplate {

    void spotMenu() {
        System.out.println("人参");
    }

    void payment() {
        System.out.println("5快");
    }
}
```

1. 具体的模板方法子类 2

```
package com.lijie;

public class RestaurantLobsterImpl  extends RestaurantTemplate  {

    void spotMenu() {
        System.out.println("龙虾");
    }

    void payment() {
        System.out.println("50块");
    }
}
```

1. 客户端测试

```
package com.lijie;

public class Client {

    public static void main(String[] args) {
        //调用第一个模板实例
        RestaurantTemplate restaurantTemplate = new RestaurantGinsengImpl();
        restaurantTemplate.process();
    }
}
```

外观模式

1.什么是外观模式

- 外观模式：也叫门面模式，隐藏系统的复杂性，并向客户端提供了一个客户端可以访问系统的接口。
- 它向现有的系统添加一个接口，用这一个接口来隐藏实际的系统的复杂性。
- 使用外观模式，他外部看起来就是一个接口，其实他的内部有很多复杂的接口已经被实现

2.外观模式例子

- 用户注册完之后，需要调用阿里短信接口、邮件接口、微信推送接口。

1. 创建阿里短信接口

```
package com.lijie;

//阿里短信消息
public interface AliSmsService {
    void sendSms();
}
123456
package com.lijie;

public class AliSmsServiceImpl implements AliSmsService {

    public void sendSms() {
        System.out.println("阿里短信消息");
    }

}
```

1. 创建邮件接口

```
package com.lijie;

//发送邮件消息
public interface EamilSmsService {
    void sendSms();
}
123456
package com.lijie;
```

```
public class EamilSmsServiceImpl implements   EamilSmsService{
    public void sendSms() {
        System.out.println("发送邮件消息");
    }
}
```

1. 创建微信推送接口

```
package com.lijie;

//微信消息推送
public interface WeiXinSmsService {
   void sendSms();
}
123456
package com.lijie;

public class WeiXinSmsServiceImpl implements  WeiXinSmsService {
    public void sendSms() {
        System.out.println("发送微信消息推送");
    }
}
```

1. 创建门面（门面看起来很简单使用，复杂的东西以及被门面给封装好了）

```
package com.lijie;

public class Computer {
    AliSmsService aliSmsService;
    EamilSmsService eamilSmsService;
    WeiXinSmsService weiXinSmsService;

    public Computer() {
        aliSmsService = new AliSmsServiceImpl();
        eamilSmsService = new EamilSmsServiceImpl();
        weiXinSmsService = new WeiXinSmsServiceImpl();
    }

    //只需要调用它
    public void sendMsg() {
        aliSmsService.sendSms();
        eamilSmsService.sendSms();
        weiXinSmsService.sendSms();
    }
}
```

1. 启动测试

```
package com.lijie;

public class Client {

    public static void main(String[] args) {
        //普通模式需要这样
        AliSmsService aliSmsService = new AliSmsServiceImpl();
```

```
        EamilSmsService eamilSmsService = new EamilSmsServiceImpl();
        WeiXinSmsService weiXinSmsService = new WeiXinSmsServiceImpl();
        aliSmsService.sendSms();
        eamilSmsService.sendSms();
        weiXinSmsService.sendSms();

        //利用外观模式简化方法
        new Computer().sendMsg();
    }
}
```

原型模式

1.什么是原型模式

- 原型设计模式简单来说就是克隆
- 原型表明了有一个样板实例，这个原型是可定制的。原型模式多用于创建复杂的或者构造耗时的实例，因为这种情况下，复制一个已经存在的实例可使程序运行更高效。

2.原型模式的应用场景

1. 类初始化需要消化非常多的资源，这个资源包括数据、硬件资源等。这时我们就可以通过原型拷贝避免这些消耗。
2. 通过new产生的一个对象需要非常繁琐的数据准备或者权限，这时可以使用原型模式。
3. 一个对象需要提供给其他对象访问，而且各个调用者可能都需要修改其值时，可以考虑使用原型模式拷贝多个对象供调用者使用，即保护性拷贝。

> 我们Spring框架中的多例就是使用原型。

3.原型模式的使用方式

1. 实现Cloneable接口。在java语言有一个Cloneable接口，它的作用只有一个，就是在运行时通知虚拟机可以安全地在实现了此接口的类上使用clone方法。在java虚拟机中，只有实现了这个接口的类才可以被拷贝，否则在运行时会抛出CloneNotSupportedException异常。
2. 重写Object类中的clone方法。Java中，所有类的父类都是Object类，Object类中有一个clone方法，作用是返回对象的一个拷贝，但是其作用域protected类型的，一般的类无法调用，因此Prototype类需要将clone方法的作用域修改为public类型。

3.1原型模式分为浅复制和深复制

1. （浅复制）只是拷贝了基本类型的数据，而引用类型数据，只是拷贝了一份引用地址。
2. （深复制）在计算机中开辟了一块新的内存地址用于存放复制的对象。

4.代码演示

1. 创建User类

```
package com.lijie;

import java.util.ArrayList;

public class User implements Cloneable {
    private String name;
    private String password;
```

```
    private ArrayList<String> phones;

    protected User clone() {
        try {
            User user = (User) super.clone();
            //重点，如果要连带引用类型一起复制，需要添加底下一条代码，如果不加就对于是复制了引用地址
            user.phones = (ArrayList<String>) this.phones.clone();//设置深复制
            return user;
        } catch (CloneNotSupportedException e) {
            e.printStackTrace();
        }
        return null;
    }

    //省略所有属性Git Set方法......
}
```

1. 测试复制

```
package com.lijie;

import java.util.ArrayList;

public class Client {
    public static void main(String[] args) {
        //创建User原型对象
        User user = new User();
        user.setName("李三");
        user.setPassword("123456");
        ArrayList<String> phones = new ArrayList<>();
        phones.add("17674553302");
        user.setPhones(phones);

        //copy一个user对象,并且对象的属性
        User user2 = user.clone();
        user2.setPassword("654321");

        //查看俩个对象是否是一个
        System.out.println(user == user2);

        //查看属性内容
        System.out.println(user.getName() + " | " + user2.getName());
        System.out.println(user.getPassword() + " | " + user2.getPassword());
        //查看对于引用类型拷贝
        System.out.println(user.getPhones() == user2.getPhones());
    }
}
```

1. 如果不需要深复制，需要删除User 中的

```
//默认引用类型为浅复制，这是设置了深复制
user.phones = (ArrayList<String>) this.phones.clone();
```

策略模式

1.什么是策略模式

- 定义了一系列的算法 或 逻辑 或 相同意义的操作，并将每一个算法、逻辑、操作封装起来，而且使它们还可以相互替换。（其实策略模式Java中用的非常非常广泛）
- 我觉得主要是为了 简化 if...else 所带来的复杂和难以维护。

2.策略模式应用场景

- 策略模式的用意是针对一组算法或逻辑，将每一个算法或逻辑封装到具有共同接口的独立的类中，从而使得它们之间可以相互替换。

1. 例如：我要做一个不同会员打折力度不同的三种策略，初级会员，中级会员，高级会员（三种不同的计算）。
2. 例如：我要一个支付模块，我要有微信支付、支付宝支付、银联支付等

3.策略模式的优点和缺点

- 优点： 1、算法可以自由切换。 2、避免使用多重条件判断。 3、扩展性非常良好。
- 缺点： 1、策略类会增多。 2、所有策略类都需要对外暴露。

4.代码演示

- 模拟支付模块有微信支付、支付宝支付、银联支付

1. 定义抽象的公共方法

```
package com.lijie;

//策略模式 定义抽象方法 所有支持公共接口
abstract class PayStrategy {

    // 支付逻辑方法
    abstract void algorithmInterface();

}
```

1. 定义实现微信支付

```
package com.lijie;

class PayStrategyA extends PayStrategy {

    void algorithmInterface() {
        System.out.println("微信支付");
    }
}
```

1. 定义实现支付宝支付

```
package com.lijie;

class PayStrategyB extends PayStrategy {

    void algorithmInterface() {
        System.out.println("支付宝支付");
    }
}
```

1. 定义实现银联支付

```
package com.lijie;

class PayStrategyC extends PayStrategy {

    void algorithmInterface() {
        System.out.println("银联支付");
    }
}
```

1. 定义下文维护算法策略

```
package com.lijie;// 使用上下文维护算法策略

class Context {

    PayStrategy strategy;

    public Context(PayStrategy strategy) {
        this.strategy = strategy;
    }

    public void algorithmInterface() {
        strategy.algorithmInterface();
    }

}
```

1. 运行测试

```
package com.lijie;

class ClientTestStrategy {
    public static void main(String[] args) {
        Context context;
        //使用支付逻辑A
        context = new Context(new PayStrategyA());
        context.algorithmInterface();
        //使用支付逻辑B
        context = new Context(new PayStrategyB());
        context.algorithmInterface();
        //使用支付逻辑C
        context = new Context(new PayStrategyC());
        context.algorithmInterface();
    }
```

```
}
```

观察者模式

1.什么是观察者模式

- 先讲什么是行为性模型，行为型模式关注的是系统中对象之间的相互交互，解决系统在运行时对象之间的相互通信和协作，进一步明确对象的职责。
- 观察者模式，是一种行为性模型，又叫发布-订阅模式，他定义对象之间一种一对多的依赖关系，使得当一个对象改变状态，则所有依赖于它的对象都会得到通知并自动更新。

2.模式的职责

- 观察者模式主要用于1对N的通知。当一个对象的状态变化时，他需要及时告知一系列对象，令他们做出相应。

实现有两种方式：

1. 推：每次都会把通知以广播的方式发送给所有观察者，所有的观察者只能被动接收。
2. 拉：观察者只要知道有情况即可，至于什么时候获取内容，获取什么内容，都可以自主决定。

3.观察者模式应用场景

1. 关联行为场景，需要注意的是，关联行为是可拆分的，而不是“组合”关系。事件多级触发场景。
2. 跨系统的消息交换场景，如消息队列、事件总线的处理机制。

4.代码实现观察者模式

1. 定义抽象观察者，每一个实现该接口的实现类都是具体观察者。

```
package com.lijie;

//观察者的接口，用来存放观察者共有方法
public interface Observer {
    // 观察者方法
    void update(int state);
}
```

1. 定义具体观察者

```
package com.lijie;

// 具体观察者
public class ObserverImpl implements Observer {

    // 具体观察者的属性
    private int myState;

    public void update(int state) {
        myState=state;
        System.out.println("收到消息,myState值改为：“+state);
    }

    public int getMyState() {
        return myState;
```

```
    }
}
```

1. 定义主题。主题定义观察者数组，并实现增、删及通知操作。

```
package com.lijie;

import java.util.Vector;

//定义主题，以及定义观察者数组，并实现增、删及通知操作。
public class Subjecct {
    //观察者的存储集合，不推荐ArrayList，线程不安全，
    private Vector<Observer> list = new Vector<>();

    // 注册观察者方法
    public void registerObserver(Observer obs) {
        list.add(obs);
    }
    // 删除观察者方法
    public void removeObserver(Observer obs) {
        list.remove(obs);
    }

    // 通知所有的观察者更新
    public void notifyAllObserver(int state) {
        for (Observer observer : list) {
            observer.update(state);
        }
    }
}
```

1. 定义具体的，他继承继承Subject类，在这里实现具体业务，在具体项目中，该类会有很多。

```
package com.lijie;

//具体主题
public class RealObserver extends Subjecct {
    //被观察对象的属性
     private int state;
     public int getState(){
         return state;
     }
     public void  setState(int state){
         this.state=state;
         //主题对象(目标对象)值发生改变
         this.notifyAllObserver(state);
     }
}
```

1. 运行测试

```
package com.lijie;

public class Client {

    public static void main(String[] args) {
```

```
        // 目标对象
        RealObserver subject = new RealObserver();
        // 创建多个观察者
        ObserverImpl obs1 = new ObserverImpl();
        ObserverImpl obs2 = new ObserverImpl();
        ObserverImpl obs3 = new ObserverImpl();
        // 注册到观察队列中
        subject.registerObserver(obs1);
        subject.registerObserver(obs2);
        subject.registerObserver(obs3);
        // 改变State状态
        subject.setState(300);
        System.out.println("obs1观察者的MyState状态值为："+obs1.getMyState());
        System.out.println("obs2观察者的MyState状态值为："+obs2.getMyState());
        System.out.println("obs3观察者的MyState状态值为："+obs3.getMyState());
        // 改变State状态
        subject.setState(400);
        System.out.println("obs1观察者的MyState状态值为："+obs1.getMyState());
        System.out.println("obs2观察者的MyState状态值为："+obs2.getMyState());
        System.out.println("obs3观察者的MyState状态值为："+obs3.getMyState());
    }
}
```

虾皮一面：手写一个Strategy模式（策略模式）

说在前面

在40岁老架构师 尼恩的**读者交流群**(50+)中，最近有指导一个小伙伴面试架构师，面试的公司包括虾皮、希音、美团等大厂，目标薪酬50K以上，遇到了一个比较初级的问题：

- 请手写一个Strategy模式（策略模式）
- 或者请手写一个template模式（模板模式）
- 或者请手写一个proxy模式（代理模式）

小伙伴却忘了， 其他都准备的好好的，**结果在这个基础知识的地方，摔了大跟头。**

这里尼恩给大家做一下系统化、体系化的梳理，使得大家可以充分展示一下大家雄厚的 “技术肌肉”，**让面试官爱到 “不能自已、口水直流”。**

也一并把这个题目以及参考答案，收入咱们的 《尼恩Java面试宝典》V85版本，供后面的小伙伴参考，提升大家的 3高 架构、设计、开发水平。

> 最新《尼恩 架构笔记》《尼恩高并发三部曲》、《尼恩Java面试宝典》 的PDF文件，请到公众号【技术自由圈】获取。

策略模式(Strategy Pattern)的定义

策略模式是属于设计模式中的行为模式中的一种，策略模式主要解决选项过多的问题，避免大量的if else 和 switch下有太多的case。

策略模式的重心不是如何实现算法，而是如何组织、调用这些算法，从而让程序结构更灵活，具有更好的维护性和扩展性

策略（Strategy）模式：

- 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
- 策略模式让算法独立于使用它的客户而变化。

策略是个形象的表述，所谓策略就是方案，日常生活中，要实现目标，有很多方案，**每一个方案都被称之为一个策略。**

打个比方说，我们出门的时候会选择不同的出行方式，比如骑自行车、坐公交、坐火车、坐飞机、坐火箭等等，这些出行方式，每一种都是一个策略。

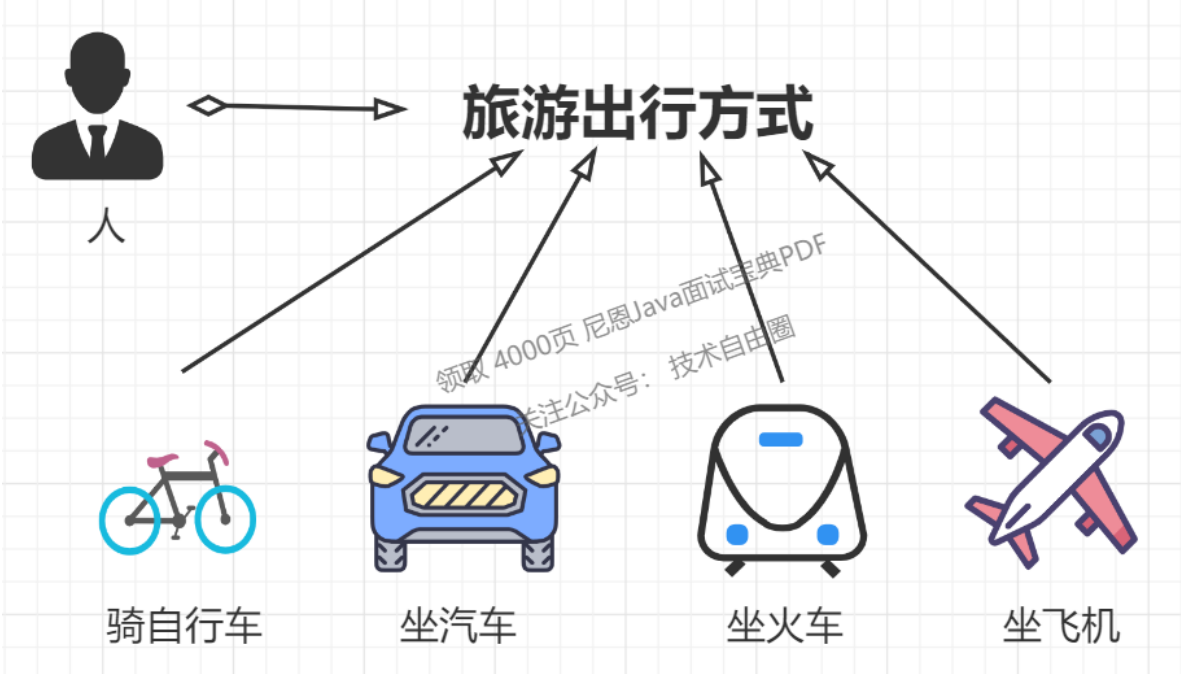
再比如我们去逛商场，商场现在正在搞活动，有打折的、有满减的、有返利的等等，其实不管商场如何进行促销，说到底都是一些算法，这些算法本身只是一种策略，并且这些算法是随时都可能互相替换的，比如针对同一件商品，今天打八折、明天满100减30，这些策略间是可以互换的。

在软件开发中也常常遇到类似的情况，实现某一个功能有多个途径，此时可以使用一种设计模式来使得系统可以灵活地选择解决途径，也能够方便地增加新的解决途径，这就是策略模式。

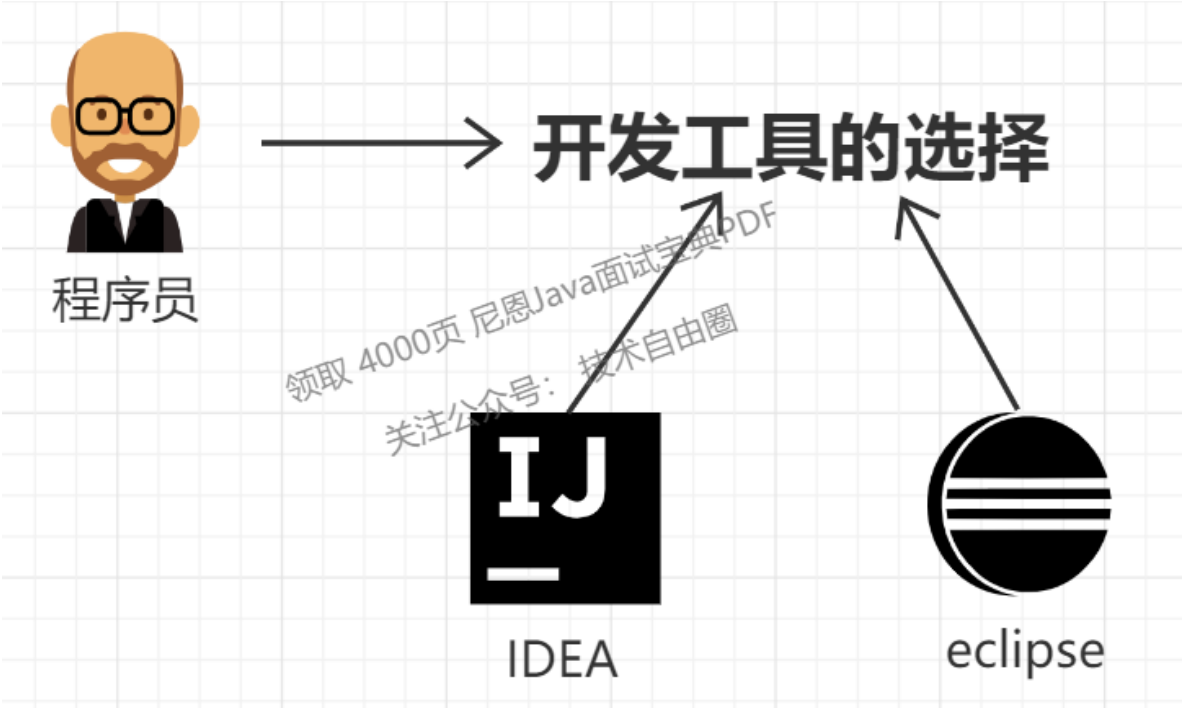
策略模式的场景分析

这个模式使得我们可以在根据环境或者条件的不同选择不同的策略来完成该任务。

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。



在软件开发中，我们也常常会遇到类似的情况，实现某一个功能有多条途径，每一条途径对应一种算法，此时我们可以使用一种设计模式来实现灵活地选择解决途径，也能够方便地增加新的解决途径。

策略模式的的最大优势： 方便代码的功能**横向扩展**，策略模式将解决途径进行封装有利于我们对解决方式的增加或删除。

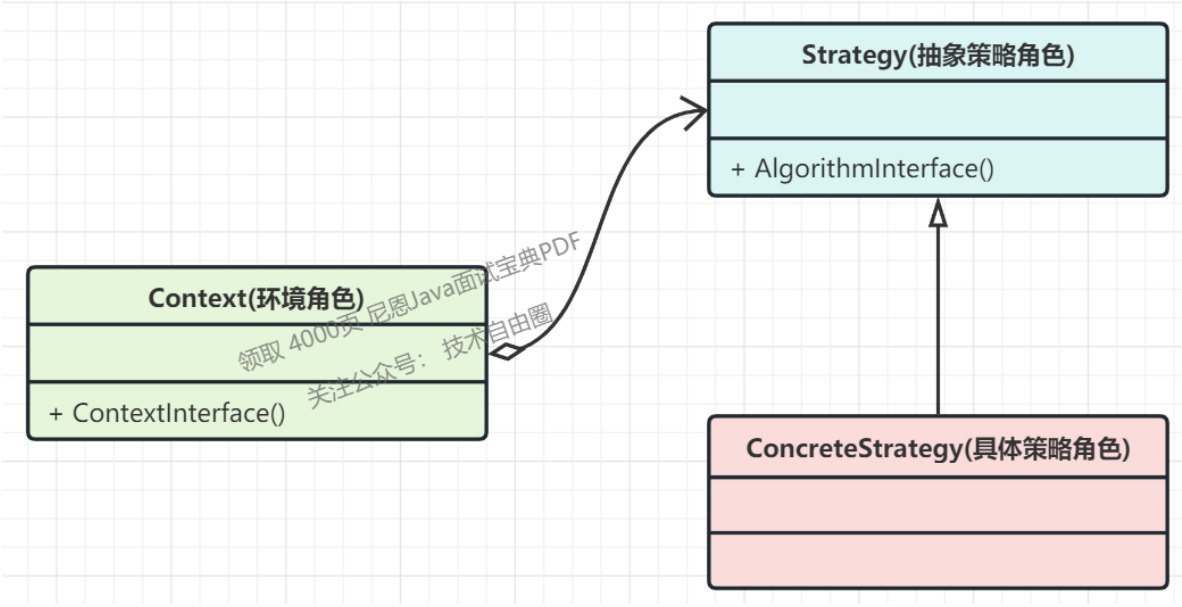
同时，策略模式(**Strategy Pattern**) 也符合**开闭原则。**

策略模式的主要角色

在策略模式中，我们可以定义一些独立的类来封装不同的算法，每一个类封装一种具体的算法，在这里，每一个封装算法的类我们都可以称之为一种策略(Strategy)，为了保证这些策略在使用时具有一致性，一般会提供一个抽象的策略类来做规则的定义，而每种算法则对应于一个具体策略类。

策略模式涉及到三个角色，具体如下：

- **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，所有具体的策略类都要实现这个接口。环境（上下文）类Context 使用这个接口调用具体的策略类。
- **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**：用于配置一个具体的算法策略对象 ，维持一个策略接口类型的引用（Reference),并且可以定义一个让接口 Strategy 的具体对象访问的接口。在简单情况下，Context 类可以省略。



策略模式是一个比较容易理解和使用的设计模式，策略模式是对算法的封装，它把算法的责任和算法本身分割开，委派给不同的对象管理。

策略模式通常把一个系列的算法封装到一系列具体策略类里面，作为抽象策略类的子类。

在策略模式中，对环境类和抽象策略类的理解非常重要，环境类是需要使用算法的类。在一个系统中可以存在多个环境类，它们可能需要重用一些相同的算法。

在客户端代码中只需注入一个具体策略对象，可以将具体策略类类名存储在配置文件中，通过反射来动态创建具体策略对象，从而使得用户可以灵活地更换具体策略类，增加新的具体策略类也很方便。

策略模式提供了一种可插入式(Pluggable)算法的实现方案。

策略模式的主要目的是将算法的定义与使用分开，也就是将算法的行为和环境分开，将算法的定义放在专门的策略类中，每一个策略类封装了一种实现算法，使用算法的环境类针对抽象策略类进行编程，**符合“依赖倒转原则”**。在出现新的算法时，只需要增加一个新的实现了抽象策略类的具体策略类即可。

策略模式的Java实现

1.创建抽象策略类

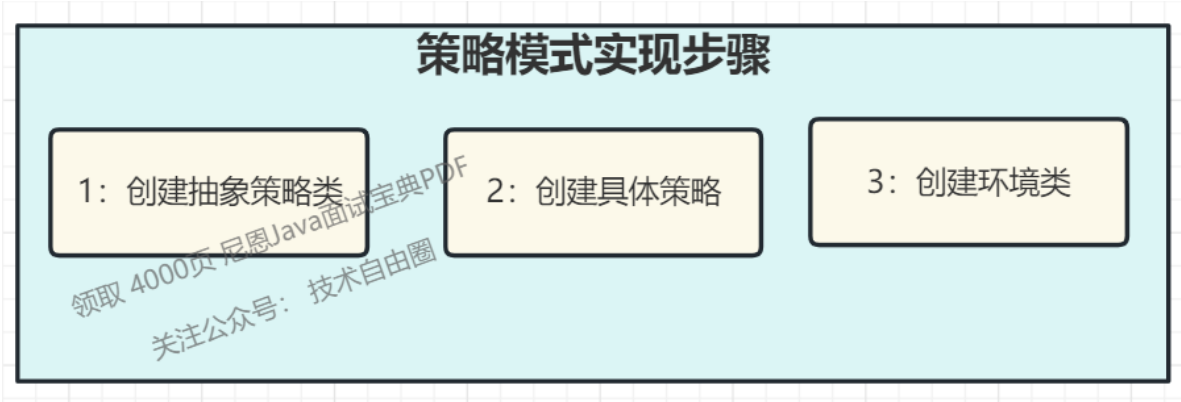
2.创建具体策略

(1)具体策略A

(2)具体策略B

(3)具体策略N....

3.创建环境类



step1：创建抽象策略类

- 定义所有持的算法的公共接口。
- 所有具体的策略类都要实现这个接口。
- context使用这个接口来调用某concreteStrategy定义的算法；

定义百货公司所有促销活动的共同接口

```
package com.crazymakercircle.designpattern.strategy;

//抽象策略类
public interface Strategy {
    void show();
}
```

step2：创建具体策略

进一步拆分策略类 ,将每个促销算法都单独封装在一个类中，也就是将一个类拆分成几个类，每个类都单独封装一个促销策略算法。

这样一来，修改一个算法只需重新编译算法所涉及的那个类，而不需要重新编译其他类。如果想要添加一个新的算法 只需在子类的集合中再添加一个新的封装该算法的类即可。

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");
    }
}
```

step3：创建环境类

Context 通常根据配置， 加载和初始化具体的 ConcreteStrategy， 配置的方式是多样化的：

- 系统环境变量
- xml文件
- 数据库
- 等等

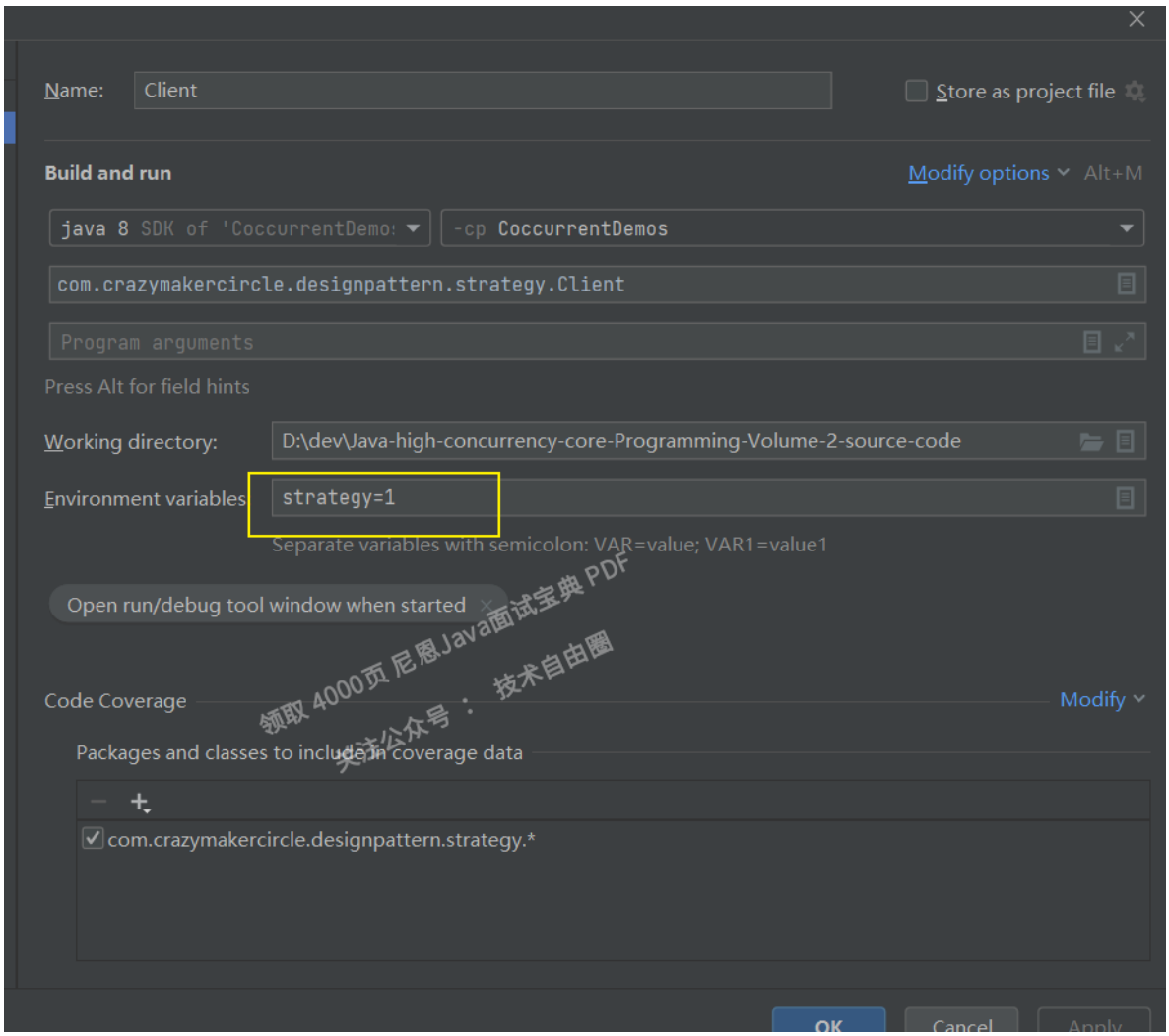
Context将它的客户的请求转发给它的Strategy。

client 仅与Context交互。客户通常从 context 获取 ConcreteStrategy，这样，Context 通常有一系列的ConcreteStrategy类可供 客户从中选择。

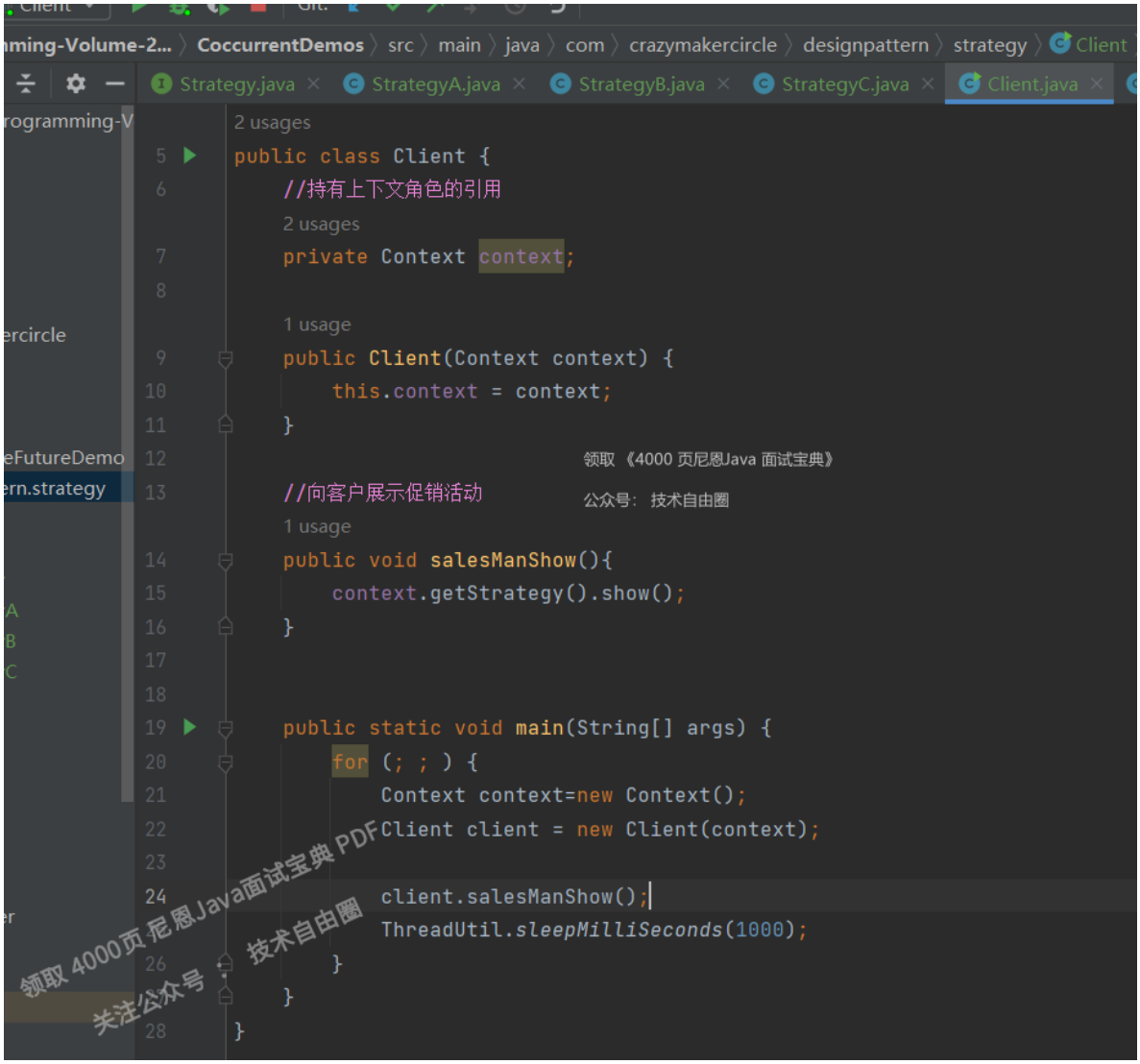
```
package com.crazymakercircle.designpattern.strategy;

import lombok.Data;

@Data
public class Context {

    String type = System.getenv("strategy");
    Strategy strategy = null;

    public Context() {
        switch (type) {
            case "1":
                strategy = new StrategyA();
                break;
            case "2":
                strategy = new StrategyB();
                break;
            case "3":
                strategy = new StrategyC();
                break;
            default:
                strategy = new StrategyA();
        }
```

```
    }
}
```

运行的时候，设置好环境变量



step4：创建客户类



策略模式的GO代码实现

下面我们就开始以GO代码的形式来展示一下策略模式吧，代码很简单，我们用一个加减乘除法来模拟。

首先，我们看到的将会是策略接口和一系列的策略，这些策略不要依赖高层模块的实现。

```
package strategy
/**
 * 策略接口
 */
type Strategier interface {
    Compute(num1, num2 int) int
}
```

很简单的一个接口，定义了一个方法`Compute`，接受两个参数，返回一个int类型的值，很容易理解，我们要实现的策略将会将两个参数的计算值返回。
接下来，我们来看一个我们实现的策略，

```
package strategy
import "fmt"
```

```
type Division struct {}

func (p Division) Compute(num1, num2 int) int {
    defer func() {
        if f := recover(); f != nil {
            fmt.Println(f)
            return
        }
    }()

    if num2 == 0 {
        panic("num2 must not be 0!")
    }

    return num1 / num2
}
```

为什么要拿除法作为代表呢？因为除法特殊嘛，被除数不能为0，其他的加减乘基本都是一行代码搞定，除法我们需要判断被除数是否为0，如果是0则直接抛出异常。
ok，基本的策略定义好了，我们还需要一个工厂方法，根据不用的type来返回不同的策略，这个type我们准备从命令好输入。

```
func NewStrategy(t string) (res Strategier) {
    switch t {
        case "s": // 减法
            res = Subtraction{}
        case "m": // 乘法
            res = Multiplication{}
        case "d": // 除法
            res = Division{}
        case "a": // 加法
            fallthrough
        default:
            res = Addition{}
    }

    return
}
```

这个工厂方法会根据不用的类型来返回不同的策略实现，当然，哪天我们需要新增新的策略，我们只需要在这个函数中增加对应的类型判断就ok。

现在策略貌似已经完成了，接下来我们来看看主流程代码，一个Computer，

```
package compute

import (
    "fmt"
    s "../strategy"
)

type Computer struct {
    Num1, Num2 int
    strate s.Strategier
}
```

```
func (p *Computer) SetStrategy(strate s.Strategier) {
    p.strate = strate
}

func (p Computer) Do() int {
    defer func() {
        if f := recover(); f != nil {
            fmt.Println(f)
        }
    }()

    if p.strate == nil {
        panic("Strategier is null")
    }

    return p.strate.Compute(p.Num1, p.Num2)
}
```

这个Computer中有三个参数，`Num1`和`Num2`当然是我们要操作的数了，strate是我们要设置的策略，可能是上面介绍的`Division`，也有可能是其他的，在main函数中我们会调用`SetStrategy`方法来设置要使用的策略，`Do`方法会执行运算，最后返回运算的结果，可以看到在`Do`中我们将计算的功能委托给了`Strategier`。

貌似一切准备就绪，我们就来编写main的代码吧。

```
package main

import (
    "fmt"
    "flag"
    c "./computer"
    s "./strategy"
)

var stra *string = flag.String("type", "a", "input the strategy")
var num1 *int = flag.Int("num1", 1, "input num1")
var num2 *int = flag.Int("num2", 1, "input num2")

func init() {
    flag.Parse()
}

func main() {
    com := c.Computer{Num1: *num1, Num2: *num2}
    strate := s.NewStrategy(*stra)

    com.SetStrategy(strate)
    fmt.Println(com.Do())
}
```

首先我们要从命令行读取要使用的策略类型和两个操作数，在main函数中，我们初始化`Computer`这个结构体，并将输入的操作数赋值给`Computer`的`Num1`和`Num2`，接下来我们根据策略类型通过调用`NewStrategy`函数来获取一个策略，并调用`Computer`的`SetStrategy`方法给`Computer`设置上面获取到的策略，最后执行`Do`方法计算结果，最后打印。

就是这么简单，现在我们在命令行定位到`main.go`所在的目录，并执行一下命令来编译文件

> go build main.go

继续执行命令

> main -type d -num1 4 -num2 2

来尝试一下使用加法策略操作4和2这两个数，来看看结果如何，

```
e:\code\go\strategy>go build main.go

e:\code\go\strategy>main.exe -type d -num1 4 -num2 2
2

e:\code\go\strategy>
```

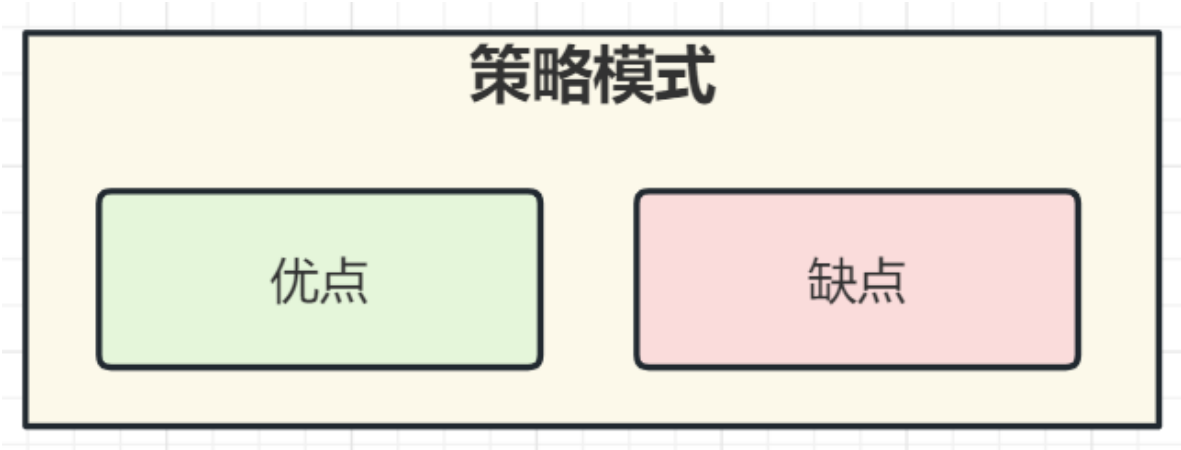
结果很正确，换一个策略试试，来个乘法吧，执行命令

> main -type m -num1 4 -num2 2

```
e:\code\go\strategy>go build main.go

e:\code\go\strategy>main.exe -type d -num1 4 -num2 2
2

e:\code\go\strategy>main.exe -type m -num1 4 -num2 2
8

e:\code\go\strategy>
```

结果也是正确的。

策略模式的优缺



优点

1. 策略模式的关注点不是如何实现算法，而是如何组织、调用这些算法，从而让程序结构更灵活，具有更好的维护性和扩展性。
2. 策略模式中各个策略算法是平等的。对于一系列具体的策略算法，地位是完全一样的。正因为这个平等性，才能实现算法之间可以相互替换。所有的策略算法在实现上也是相互独立的，相互之间是没有依赖的。所以可以这样描述这一系列策略算法：策略算法是相同行为的不同实现。

3. 运行期间，策略模式在每一个时刻只能使用一个具体的策略实现对象，虽然可以动态地在不同的策略实现中切换，但是同时只能使用一个。

如果所有的具体策略类都有一些公有的行为。这时候，就应当把这些公有的行为放到共同的抽象策略角色 Strategy 类里面。当然这时候抽象策略角色必须要用 Java 抽象类实现，而不能使用接口。但是，编程中没有银弹，策略模式也不例外，也有一些缺点，先来回顾总结下优点：

1. 策略模式提供了对"开闭原则"的完美支持，用户可以在不修改原有系统的基础上选择算法或行为，也可以灵活地增加新的算法或行为。
2. 策略模式提供了管理相关的算法族的办法。策略类的等级结构定义了一个算法或行为族。恰当使用继承可以把公共的代码移到父类里面，从而避免代码重复。
3. 使用策略模式可以避免使用多重条件(if-else)语句。多重条件语句不易维护，它把采取哪一种算法或采取哪一种行为的逻辑与算法或行为的逻辑混合在一起，统统列在一个多重条件语句里面，比使用继承的办法还要原始和落后。

缺点

1. 客户端必须知道所有的策略类，并自行决定使用哪一个策略类。这就意味着客户端必须理解这些算法的区别，以便适时选择恰当的算法类。这种策略类的创建及选择其实也可以通过工厂模式来辅助进行。
2. 由于策略模式把每个具体的策略实现都单独封装成为类，如果备选的策略很多的话，那么对象的数目就会很可观。可以通过使用**享元模式**在一定程度上减少对象的数量。

关于策略模式的讨论

策略模式还是算比较容易理解的，策略模式的核心就是将容易变动的代码从主逻辑中分离出来，通过一个接口来规范它们的形式，在主逻辑中将任务委托给策略。这样做既减少了我们对主逻辑代码修改的可能性，也增加了系统的可扩展性。

设计的核心原则：**对扩展开发，对修改关闭**

使用策略模式主要有两个出发点：

(1) 将一组相关的算法封装为各个策略分支，从而将策略分支相关的代码隐藏起来。

(2) 希望可以提升程序的可扩展性。

下面我们就策略模式的可扩展性进行简单的讨论，实际上 策略模式的初衷是要减少与各个分支下的行为相关的条件语句。这已经通过将一个具有条件相关的多种行为的类拆分成一个策略超类与若干个策略子类得到了解决。也就是说,将原来的一个单独的但是包含多个条件语句的类改变为一个没有条件语句的策略层次类。

这里虽然看似条件语句消失了，但是在客户程序与 Context 类中是否也不存在与策略子类相关的条件语句了呢？答案当然不是。

实际上一般在策略模式的设计中 **客户类根据不同的条件负责创建不同的策略子类的对象**，然后再将该对象传递给 Context 环境类，Context 类的作用可以理解为：为被调用策略子类的一些方法提供一些参数，以及使用该由 Client 类传入的对象去调用 Strategy 类的某些方法。

这说明 在客户类 Client 中 存在许多与策略分支子类相关的条件语句，而在 Context 类中，没有这样的语句。

那么 是否可以将创建策略子类的对象的责任交给 Context 类，而客户类 Client 只为 Context 类提供一些代表客户请求的参数呢?

(1) 客户类负责创建策略子类的对象的情况

客户类根据用户提供的不同的请求，负责创建不同的策略子类的对象 ， 然后再将该对象传递给 Context 类。在这种情况下，客户类中通常包含与策略相关的条件语句，而在 Context 类中不必使用任何与策略有关的条件语句，因此，修改或者添加一个策略子类都不必修改 Context 类。但是，在添加一个新的策略子类的情况下，**如果客户类需要使用该子类**，往往需要在客户类中添加一个新的条件语句，即客户类需要修改。

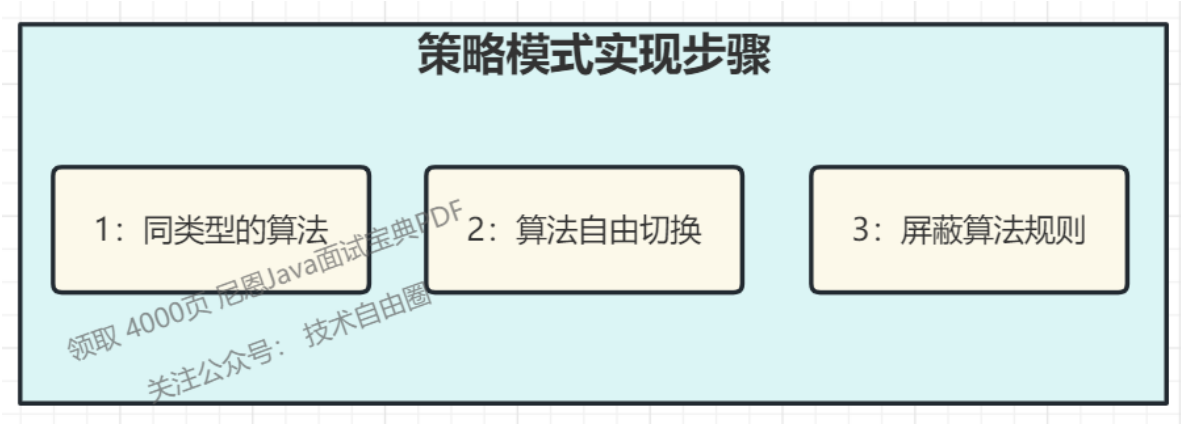
(2) Context 类负责创建策略子类的对象的情况

将创建策略子类的对象的责任交给 Context 类， 而客户类 Client 只为 Context 类提供一些代表客户请求的参数 ； 在此情况下，Context 类在创建策略子类的对象时，必然会使用与策略子类有关的条件语句。**此时，修改一个策略子类不需要修改客户类与 Context 类。**而在添加一个新的策略子类时，如果此时客户类暂时不使用该新的子类，则新子类的添加不会影响客户类与 Context 类的源代码。但是，如果客户类要使用新的策略子类，则必须**同时**在客户类与 Con-text 类中添加新的条件分支，也就是说，**需要同时修改客户类与 Context 类。**

在以上两种情况下，当只是需要修改策略子类的代码时，客户类与 Context 类都不需要进行修改。

综上所述 **由客户类创建对象的设计**可扩展性好一些。这样，可以做到在 Context 类中出现与策略子类相关的条件语句，从而可扩展性也得到了提高。

策略模式的应用场景



- 多个类只有算法或行为上稍有不同的场景
- 算法需要自由切换的场景
- 需要屏蔽算法规则的场景
- 出行方式，自行车、汽车等，每一种出行方式都是一个策略
- 商场促销方式，打折、满减等

策略模式和工厂模式的区别

工厂模式

1. 目的是创建不同且相关的对象
2. 侧重于"创建对象"
3. 实现方式上可以通过父类或者接口

4. 一般创建对象应该是现实世界中某种事物的映射，有它自己的属性与方法

策略模式

1. 目的实现方便地替换不同的算法类
2. 侧重于算法(行为)实现
3. 实现主要通过接口
4. 创建对象对行为的抽象而非对对象的抽象，很可能没有属于自己的属性

说在最后

手写一些基础的设计模式，是非常常见的面试题。

以上2大方案，如果大家能对答如流，如数家珍，基本上 面试官会被你 震惊到、吸引到。

最终，**让面试官爱到 “不能自已、口水直流”**， 同时 offer， 也就来了。

学习过程中，如果有啥问题，大家可以来 找 40岁老架构师尼恩交流。

参考文献

https://blog.csdn.net/sufu1065/article/details/124287643

推荐阅读

《百亿级访问量，如何做缓存架构设计》

《多级缓存 架构设计》

《消息推送 架构设计》

《阿里2面：你们部署多少节点？1000W并发，当如何部署？》

《美团2面：5个9高可用99.999%，如何实现？》

《网易一面：单节点2000Wtps，Kafka怎么做的？》

《字节一面：事务补偿和事务重试，关系是什么？》

《网易一面：25Wqps高吞吐写Mysql，100W数据4秒写完，如何实现？》

《亿级短视频，如何架构？》

《炸裂，靠“吹牛”过京东一面，月薪40K》

《太猛了，靠“吹牛”过顺丰一面，月薪30K》

《炸裂了...京东一面索命40问，过了就50W+》

《问麻了...阿里一面索命27问，过了就60W+》

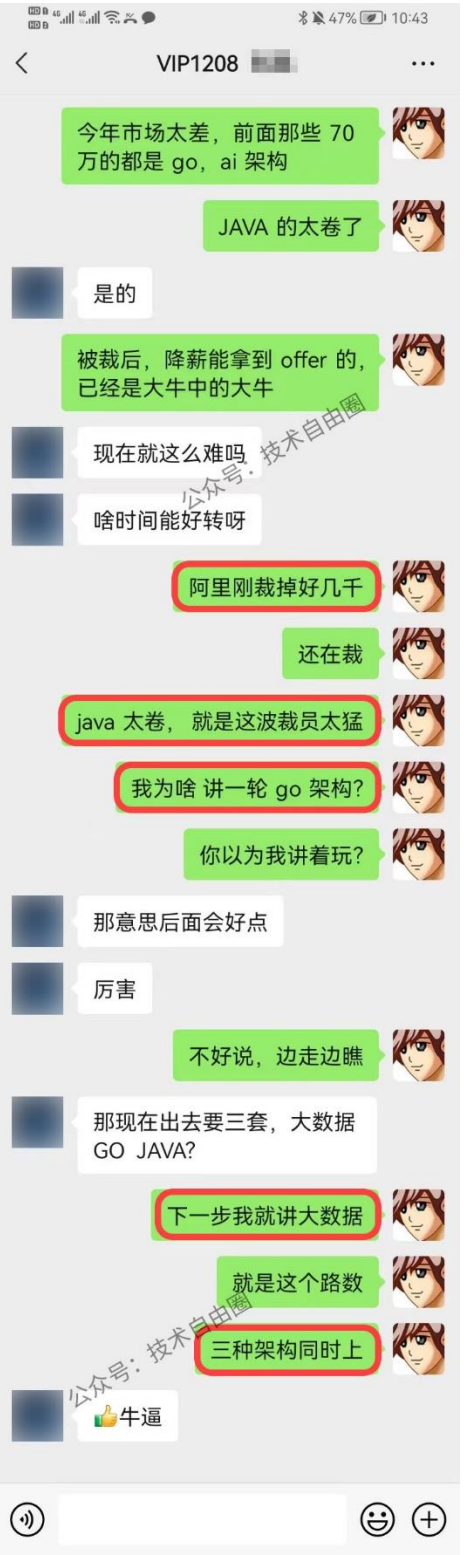
《百度狂问3小时，大厂offer到手，小伙真狠！》

《饿了么太狠：面个高级Java，抖这多硬活、狠活》

《字节狂问一小时，小伙offer到手，太狠了！》

《收个滴滴Offer：从小伙三面经历，看看需要学点啥？》

未来职业，如何突围：三栖架构师



成功案例：2年翻3倍，35岁卷王成功转型为架构师

详情：http://topcoder.cloud/forum.php?mod=forumdisplay&fid=43&page=1

最新　最后发表　热门　精华

- 成功案例：[1057号卷王] 3年小伙拿到外企offer，薪酬涨了200%
  卷王1号　超级版主　前天 17:41
- 成功案例：[645号卷王] 4年经验卷王逆袭，被毕业后，反涨24W
  卷王1号　超级版主　2022-9-21
- 成功案例：[878号卷王] 小伙8年经验，年薪60W
  卷王1号　超级版主　2022-8-13
- 年薪70W案例：通过尼恩的指导，小伙伴年薪从40W涨到70W
  卷王1号　超级版主　2022-2-11
- 成功案例：[493号卷王] 5年小伙拿满意offer，就业寒冬季逆涨30%
  卷王1号　超级版主　前天 17:43
- 成功案例：[250号卷王] 就业极寒时代，收offer 涨25%
  卷王1号　超级版主　前天 17:38
- 成功案例：[612号卷王] 就业极寒时代，从外包到自研
  卷王1号　超级版主　前天 17:15
- 成功案例：[913号卷王] 热烈祝贺6年经验卷王，年薪40W
  卷王1号　超级版主　2022-9-21
- 成功案例：[959号卷王] 4年经验卷王，喜获百度、Boss直聘等N个优质offer，最高涨100%
  卷王1号　超级版主　2022-9-21
- 成功案例：[529号卷王] 5年经验卷王喜收2大offer，最高涨5K
  卷王1号　超级版主　2022-9-21
- 成功案例：[811号卷王] 热烈祝贺7年经验卷王，薪酬涨30%
  卷王1号　超级版主　2022-9-21
- 成功案例：[287号卷王] 不惧大寒潮，卷王逆市收4 offer，涨30%，可喜可贺
  卷王1号　超级版主　2022-5-30
- 成功案例：[1002号卷王] 5月份"被毕业"，改简历后，斩获顶级央企Offer，涨薪7000+
  卷王1号　超级版主　2022-7-5

成功案例：[7号卷王] 热烈祝贺小伙伴涨薪120%

卷王1号 超级版主 2022-8-13

成功案例：[134号卷王] 大三小伙卷1年，斩获顶级央企Offer，成功逆袭

卷王1号 超级版主 2022-7-6

成功案例：[1008号卷王] 5年经验卷王收42W offer，月涨8000，可喜可贺

卷王1号 超级版主 2022-5-30

成功案例：[453号卷王] 非全日制 6年卷王喜提3 offer，年薪30W，可喜可贺

卷王1号 超级版主 2022-5-21

成功案例：[924号卷王] 6年卷王喜提4 offer，最高涨薪9000，可喜可贺

卷王1号 超级版主 2022-5-21

成功案例：[15号卷王] 4年卷王入职 微软，涨薪50%，可喜可贺

卷王1号 超级版主 2022-5-12

成功案例：[527号卷王] 4年卷王喜提2 offer，涨薪50%，可喜可贺

卷王1号 超级版主 2022-5-13

成功案例：[788号卷王] 3年卷王喜提优质Offer，涨薪60%

卷王1号 超级版主 2022-5-11

成功案例：热烈祝贺：非全日制卷王，喜提2个心仪offer，面3家过2家

卷王1号 超级版主 2022-4-21

成功案例：[693号卷王] 二线城市6年卷王喜提4大优质Offer，含央企offer，最高薪酬35W

卷王1号 超级版主 2022-4-16

成功案例：[85号卷王] 双非2本小伙，春招大捷，喜提9个offer，最高薪酬近30万

卷王1号 超级版主 2022-4-14

成功案例：[741号卷王] 卷王逆袭！6年小伙从很少面试机会到搞定35K*14薪Offer

卷王1号 超级版主 2022-4-12

成功案例：[642号卷王] 热烈祝贺，6年卷王喜提优质国企offer

卷王1号 超级版主 2022-4-7

成功案例：[796号卷王] 热烈祝贺，36岁卷王喜提52万优质offer

卷王1号 超级版主 2022-3-25

成功案例：[15号卷王] 小伙卷1年，涨薪9K+，喜收ebay等多个优质offer

卷王1号 超级版主 2022-3-24

成功案例：[821号卷王] 小伙狠卷3个月，喜提10多个offer

卷王1号 超级版主 2022-3-21

成功案例：[736号卷王] 3年半经验收22k offer，但是小伙志存高远，冲击25k+

卷王1号 超级版主 2022-3-20

成功案例：热烈祝贺1群小卷王offer拿到手软，甚至拒了阿里offer

卷王1号 超级版主 2022-3-16

简历案例：简历一改，腾讯的邀请就来了！热烈祝贺，小伙收到一大堆面试邀请

卷王1号 超级版主 2022-3-10

成功案例：祝贺我圈两大超级卷王，一个过了阿里HR面，一个过了阿里2面

卷王1号 超级版主 2022-3-10

成功案例：小伙伴php转Java，卷1.5年Java，涨薪50%，喜收多个优质offer

卷王1号 超级版主 2022-3-10

成功案例：4年小伙狠卷半年，拿到 移动、京东 两大顶级offer

尼恩 超级版主 2022-3-5

成功案例：[267号卷王] 助力3年经验卷王，拿到蜂巢的17k x 14薪的offer

卷王1号 超级版主 2022-2-27

成功案例：[143号卷王] 二本院校00后卷神，毕业没到一年跳到字节，年薪45W

卷王1号 超级版主 2022-2-27

成功案例：[494号卷王] 尼恩分布式事务助力卷王拿到 中信银行offer

卷王1号 超级版主 2022-2-27

成功案例：[76号卷王] 2线城市卷王，狠卷1.5年，喜收22K offer

卷王1号 超级版主 2022-2-27

成功案例：[429号卷王] 小伙伴在社群卷5个月，涨8k+

卷王1号 超级版主 2022-2-27

成功案例：[154号卷王] 双非学校毕业卷王，连拿 京东到家&滴滴 两个大厂Offer

卷王1号 超级版主 2022-2-27

成功案例：[232号卷王] 涨薪10K，继续卷向食物链顶端

卷王1号 超级版主 2022-2-27

成功案例：狠卷1年技术，喜收 腾讯、阿里、微软三大Offer，最高年薪56W

卷王1号 超级版主 2022-2-27

成功案例：[449号卷王] 应届毕业卷王喜收 滴滴offer, 年薪33W

卷王1号 超级版主 2022-2-27

成功案例：[551号卷王] 小伙伴学完后，成功进入大厂，并且推荐自己的朋友加VIP学习

卷王1号 超级版主 2022-2-10

成功案例：[214号卷王] 助力2年经验卷王，成功拿到17K月薪

卷王1号 超级版主 2022-2-10

成功案例：[92号卷王] 课程实操助力社群小伙伴喜收 喜马拉雅Offer

卷王1号 超级版主 2022-2-10

成功案例：社群卷王小伙伴成功过了滴滴三面 获滴滴Offer

卷王1号 超级版主 2022-2-10

[612号卷王]滴滴小伙伴，蹲点考察半年，觉得靠谱后加入 疯狂创客圈

卷王1号 超级版主 2022-2-10

成功案例：[732号卷王] 尼恩助力3年经验卷王收获 京东offer, 年薪35W

卷王1号 超级版主 2022-2-27

成功案例：[558号卷王] 2年经验卷王，喜收 网易和阿里子公司两个优质offer

卷王1号 超级版主 2022-2-27

成功案例：[569号卷王] 双非应届生卷王, 喜收字节跳动实习offer

卷王1号 超级版主 2022-2-25

成功案例：[420号卷王] 狠卷1年，卷王涨薪80%，涨薪12000元!

卷王1号 超级版主 2022-2-25

成功案例：[76号卷王] 通过尼恩1年半的指导，专科学历小伙伴从0.8K涨到22K

卷王1号 超级版主 2022-2-10

硬核推荐：尼恩Java硬核架构班

详情：https://www.cnblogs.com/crazymakercircle/p/9904544.html

架构班（社群 VIP）的起源：

最初的视频，主要是给读者加餐。很多的读者，需要一些高质量的实操、理论视频，所以，我就围绕书，和底层，做了几个实操、理论视频，然后效果还不错，后面就做成迭代模式了。

架构班（社群 VIP）的功能：

提供高质量实操项目整刀真枪的架构指导、快速提升大家的:

- 开发水平
- 设计水平
- 架构水平

弥补业务中 CRUD 开发短板，帮助大家尽早脱离具备 3 高能力，掌握：

- 高性能
- 高并发
- 高可用

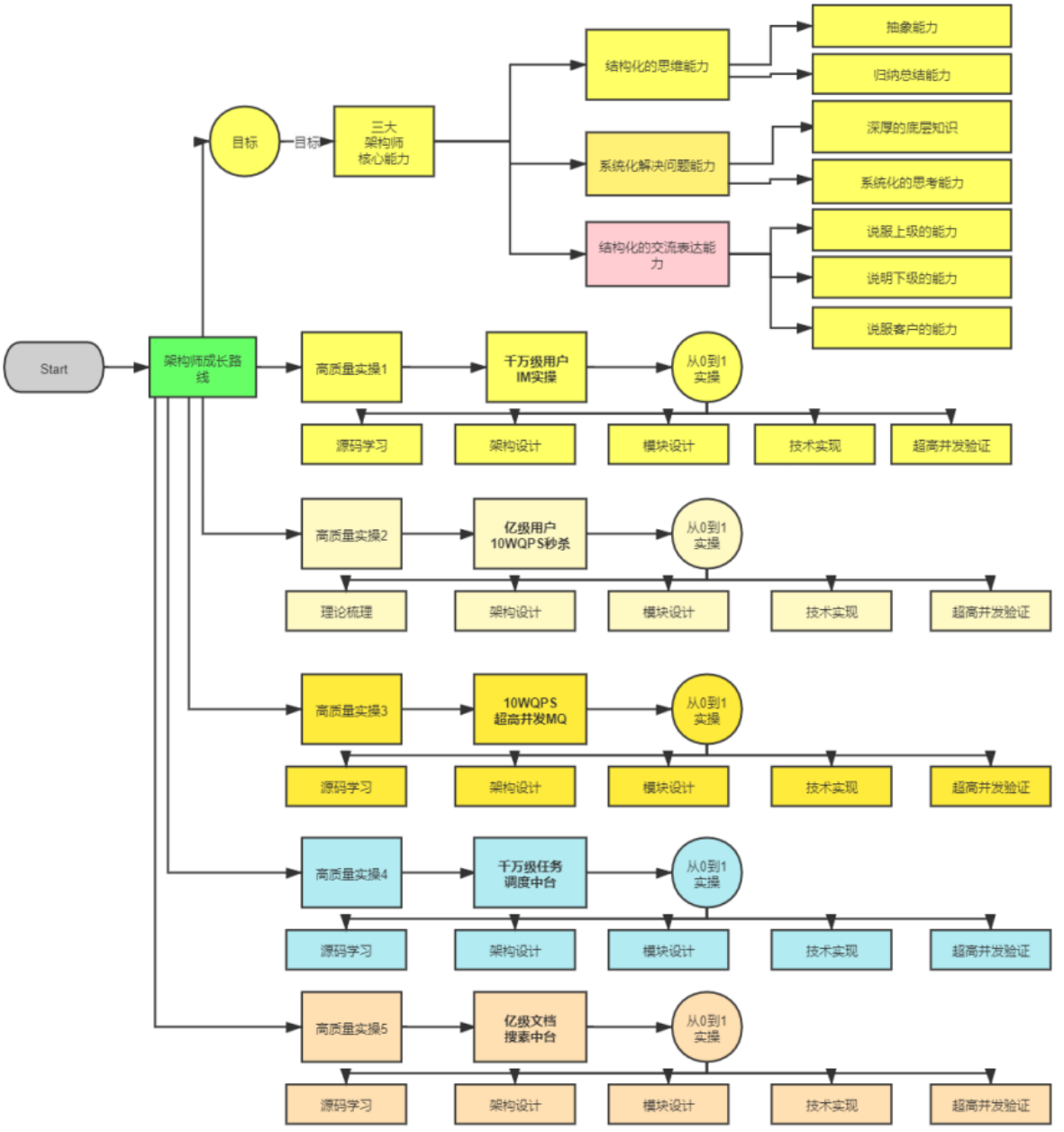
作为一个高质量的架构师成长、人脉社群，把所有的卷王聚焦起来，一起卷：

- 卷高并发实操
- 卷底层原理
- 卷架构理论、架构哲学
- 最终成为顶级架构师，实现人生理想，走向人生巅峰

架构班（社群 VIP）的目的：

- 高质量的实操，大大提升简历的含金量，吸引力，增强面试的召唤率
- 为大家提供九阳真经、葵花宝典，快速提升水平
- 进大厂、拿高薪
- 一路陪伴，提供助学视频和指导，辅导大家成为架构师
- 自学为主，和其他卷王一起，卷高并发实操，卷底层原理、卷大厂面试题，争取狠卷 3 月成高手，狠卷 3 年成为顶级架构师

N 个超高并发实操项目：简历压轴、个顶个精彩



【样章】第 17 章：横扫全网 Rocketmq 视频第 2 部曲：工业级 rocketmq 高可用（HA）底层原理和实操

工业级 rocketmq 高可用底层原理，包含：消息消费、同步消息、异步消息、单向消息等不同消息的底层原理和源码实现；消息队列非常底层的主从复制、高可用、同步刷盘、异步刷盘等底层原理。

工业级 rocketmq 高可用底层原理和搭建实操，包含：高可用集群的搭建。

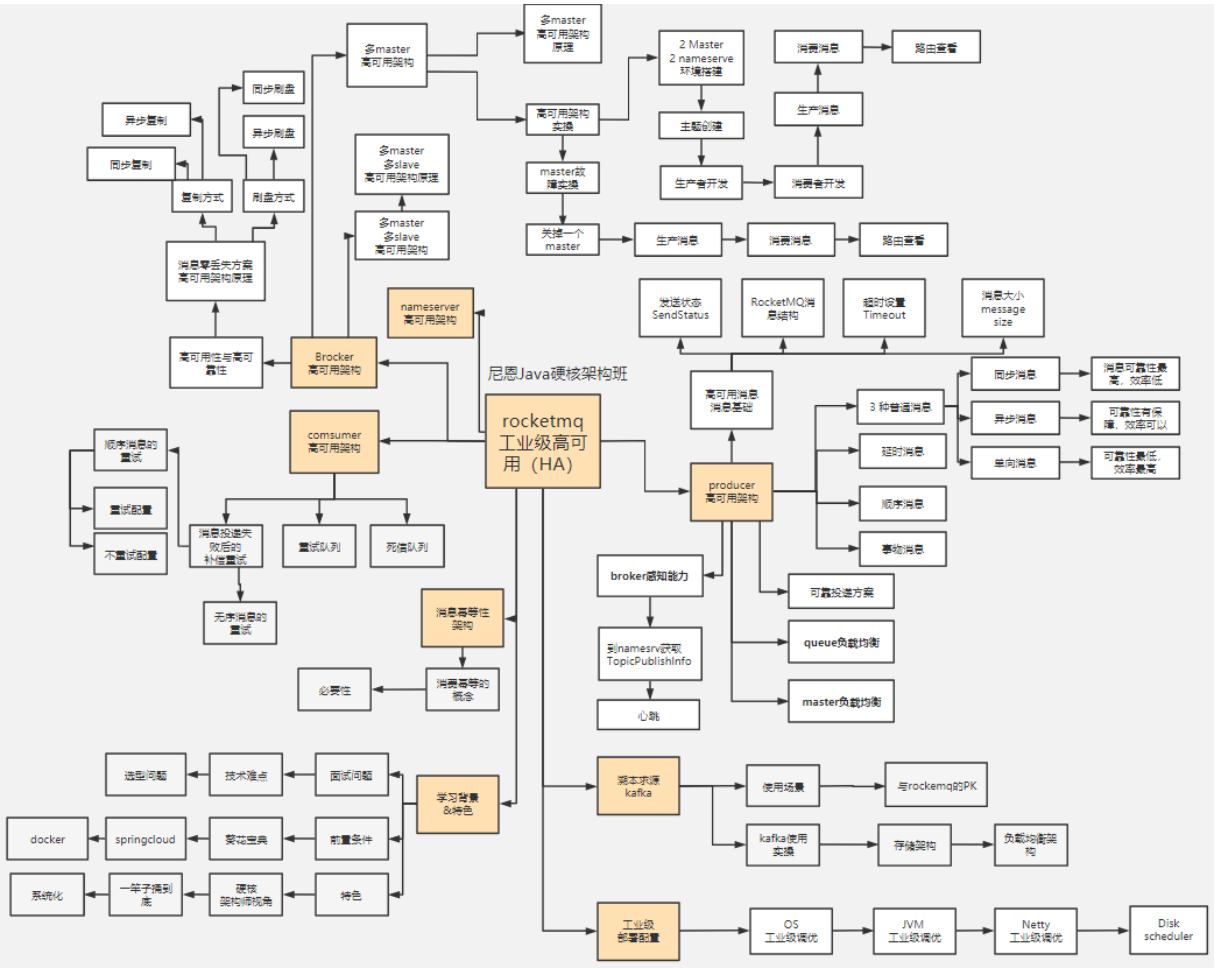
解决以下难题：

1、技术难题：RocketMQ 如何最大限度的保证消息不丢失的呢？RocketMQ 消息如何做到高可靠投递？

2、技术难题：基于消息的分布式事务，核心原理不理解

3、选型难题： kafka or rocketmq ，该娶谁？

下图链接：https://www.processon.com/view/6178e8ae0e3e7416bde9da19



简历优化后的成功涨薪案例（VIP含免费简历优化）


6年小伙收60W年薪
一月速提3大offer
4.24号改简历
5.27号报喜
VIP1239 6年
4月24日 晚上19:10
预期的岗位：60W
预期的岗位：go 和 java 的后端开
4月24日 晚上19:56
秘诀：
简历指导+ 狠狠卷
4月24日 晚上20:50
6年经验 1239
年薪 60W.
21.7 KB
微信电脑版
前几天改 ai，今天改 go
谢谢，我根据这个模式，再整理一下我简历，到时候老师再帮我把把关
ok
4月27日 下午14:37
老师，再帮我看看
准备开始投简历了
简历-6 年后端开发.pdf
210.1 KB
PDF
VIP1239 6年
嗯嗯，我先对着简历准备些东西，然后再开始投吧，反正现在刚刚好五一放假
上午8:01
来还愿啥，3周斩获3个offer，准备入职了
上午8:05
您太牛啦
简历优化后，面试机会太多了，拿完3个offer后，还有许多公司在流程中的都拒绝了
这几天大厂动作不断，联想，阿里都在裁员，您太牛啦
还是老师你强，offer中也达到了预期60w
非常不错
现在都上岸有offer就行，薪酬还能达到预期，已经超级牛掰啦
很多人，连一个面试电话都没有，崩溃的一塌糊涂
抖音上到处是这种
主要是简历优化后，感觉如有神助，每天基本3个面试，除了字节一面没过，其它都通过了
恭喜您
谢谢老师
后续有啥问题，可以找我支撑哈
好嘞，学习圈一直在，要持续提高自己
好的，撸起袖子加油卷，搞技术前途无限好
被裁后转架构，逆涨50%
8年小伙喜提年薪75W
4.16号改简历
5.6号报喜
VIP1236
最近面试了几个一轮游
捞了太多人上岸了
都是你这号
捞我
助力我 一个月时间
绞杀 下钻 打破瓶颈
咱们开始不？
好
4月16日 下午15:04
预期岗位：高级开发、架构
4月16日 下午15:12
预期薪酬：60W
-8 年高级服务端-0404 - 副本(2).
30.2 KB
微信电脑版
4月16日 下午16:19
秘诀：
简历指导+ 狠狠卷
4月16日 下午19:21
通话时长 03:02:25
辛苦老师
VIP1236
上午11:08
尼架，我决定要去上海了。
拿了几个 offer 了？
两个
上海这个是架构师对吧？还有一个呢？
还有个广州高级 Java，待遇 40w 左右，老板比较喜欢我，开了很多绿灯，薪资可以再加，但我还是想闯闯，昨天拒绝了。
年包多少呢？
就之前说的
那都快 80 个 W 了
我 argue 了下，他们控制内部薪酬平衡有点难办到 80，但已经是标出来的上限了。
都快是 高级 java 的 两倍
你撤回了一条消息
75 w 也非常多了，在现在的环境下
除开税，差不了多少，主要是这个方向的潜在价值
关键对你来说，这个是一个 成长机会
是的，寒意还是有的
您是我的贵人
是呢 有个外接大脑就是爽
好好卷，先祝贺您，拿到年薪 75W +
再预祝您，2 年之后，年薪 200W+
谢谢


9年 小伙伴拿到
年薪90W offer
9月11日改简历
11月29日晒offer
秘诀:
简历指导+ 狠狠卷
这个是给你微调的
省略号的地方，需要你再补充一点
上面你留着这个就行
Java 开发 9年-修改.docx
34.1 KB
薪资高 稳
主要的工作是啥?
提升了自己的实力，就不用怕
易所 关于数字货 的 易
po 也有 打盹的时候，该裁员，照样一个不少
年包比 po 多 19w
这么多
po 估计有 70 万
那你不有 90 个 W?
po 给我 68
就是吗


小伙8年经验
年薪60w
7月12日改简历
8月10日晒offer
秘诀:
改简历 + 狠狠卷
明天晚上哈
好哒
恩哥，今晚还改简历吗?
今晚还在外边应酬，估计回去比较晚
要不，咱们延迟到明天，如何
明天白天也行
好的，白天吧，答应别人明天给他们简历了。
OK
那就上午 11 点左右哈
好的
涨幅多少呀?
或者，额度多少呀
之前 36*15，现在这个 39*15
今年行情不太好，还有一些 offer 基本都是平薪，或降薪的。
OK
这个马上来
刚在指导简历
哈哈
稍等哈
辛苦啊恩哥
这次找工作，您的 起到效果了。
我这次复习基本看的都是咱们课程的 。


6年小伙伴
年薪40w
9月6日改简历
9月21日晒offer
秘诀:
简历指导+ 狠狠卷
Java - 6年.docx
24.7 KB
恩哥，简历我改好了，您再帮我看一下
Java - 6年(1).docx
24.5 KB
恩哥，看第二个吧
给你前面调整了一下
涨这多
今年这行情，也算可以了
总包多少呀，让我也陶醉一下
大概 40w 吧
谢谢恩哥的指导和鼓励
是在深圳
深圳的行情尤其难
能有面试电话就不错啦
是的
太牛啦
哈哈哈哈，恩哥的鼓励指导也很重要


5年小伙喜提3个offer
年薪 35个W
5月22日改简历
11月29日晒offer
我简历还没写出来 不然可以给老师您看看
算法方面，可以找点 刷刷，当作复习
项目的话，再好好改造一下
好的 谢谢老师
java 开发-本科统招-2022-5 年经验.docx
36.0 KB
OK
我排好号，回头找你哈
秘诀:
简历指导+ 狠狠卷
尼恩老师晚上好，汇报下最近的 offer 情况。最近面试收到了三个 offer，两个是平薪，一个是跨境电商公司的 offer，涨幅暂时不到 20%。通过这次面试也让我知道自己距离高级开发还有一点点距离，还要再多看才能突破。
最后结合自己的情况，先选择去跨境电商的公司再提升下
之前是 20k*13.5，跨境电商这个是说 23k*（14-15）
恭喜恭喜
就业寒冬，能提到 3 个 offer，已经很 ，已经是高手了
很多小伙伴，面试电话一个都接不到，简历海投 7000 份，只收到 3 次面试机会，没有一个机会拿到最终 offer
您这个年薪，算下来也有 35W 了吧
年终部分还要确认下，大部分人听说只有两个月
这个时间点，拿到这个水平，挺不错的啦
持续加油卷哈
嗯嗯！看看明年自己有没有能力冲击高开
把你我视频都吃透
辛苦老师再帮忙指导下哈


1.5年小伙搞定15K offer
就业寒冬涨100%
5月7日改简历
11月21日晒offer
秘诀：
简历指导+狠狠卷
多谢啦
你好，好多公司都觉得我的简历太简单了，我感觉是不是没法优化了
简历.pdf
397.4 KB
好的
晚上指导你改
推送中台需要加不
我还没做啊，准备八股文暂时没时间搞这些，入职了会开始搞
OK
还有其他的项目吗
一个项目，太少啦
那种练习项目也行
他那个不止两倍吧，我才是两倍，原先7k，现在15k
那也很牛
都翻倍了，都是牛人
正常就30%
慢慢来
下一步可以瞄准25k
其实我工作一年
那你就更牛逼拉
一年经验，搞定15k offer，舍你其谁


卷王逆袭成功案例
6年小伙从很少面试机会到
搞定35K*14薪
3月5日改简历
4月11日拿offer
面试邀请少
小伙很苦恼
理想很丰满
目标35K
一个月拿到了
理想的offer
面试法宝
rocketmq四部曲

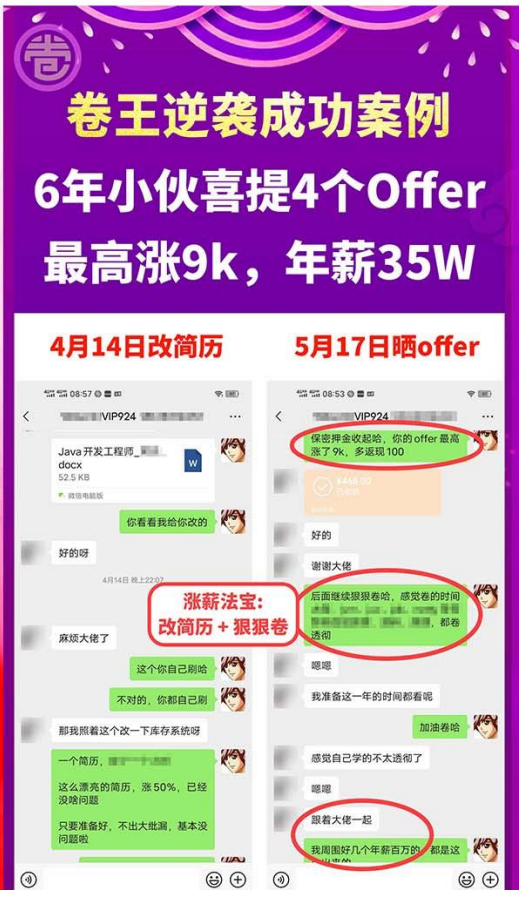

6年 经验小伙伴
喜收25K offer
3月12日改简历
12月1日晒offer
秘诀：
简历指导+狠狠卷
老师
咱们以这个项目为模板改哈
嗯好
项目有多少人，你带领多少人？
你工作几年了？
6年了
项目有多少人，你带领多少人？
语音交流一下
好
你方便吗
OK
不知道往哪个方向发展
改简历挪窝，20 涨到 25 k，高级开发，P7 带的后端团队
任务重，道路远，并不是没有方向


7年经验卷王
薪酬涨30%
7月11日改简历
9月1日晒offer
秘诀：
改简历 + 狠狠卷
咱们开始修改哈
要语音还是打字也行，还没到家
解决过最复杂的难题是啥？说 3 个哈
看看，有没有问题？
没问题
看看，有没有问题？
只能挪窝
下次你喊足一点哦
只要本事好，拿 offer 比较容易的
喊了，但是我流水太差了。。。我这家是补贴，银行不认这玩意，只能在上家的基础上加到 30% 封顶（特别优秀除外，我只是不错。。。）
入职了，最终并不能大幅涨薪，还是旧的太低，只涨了 30%
恭喜一下
现在不比往年
能涨 30% 是大牛
拿到 offer 都算大牛
现在很多人投几百发，连个电话都没有
你已经很


4年经验卷王逆袭
被毕业后，反涨24W
7月改简历
8月30日晒offer
秘诀:
改简历 + 狠狠卷
这就是你的简历
差得太多啦
ok
总共写了四个项目，最近的一年的还没补充上
你是在职，还是离职呢?
离职
原因大概是啥?
这个地方要这样写
项目被终止
方便语音沟通不
ok
是的 经你这么指导，非常重要
老哥 我八月十号开始找工作，今天已经入职了
现金基本持平，股票 +24W
总计涨了多少呢
能涨24W
股票这个吧只能到手了才算
也不错啦
很多小伙伴，面试机会都没有
感谢老哥的指导，继续跟你卷技术
继续狠狠卷哈，马上就技术自由拉

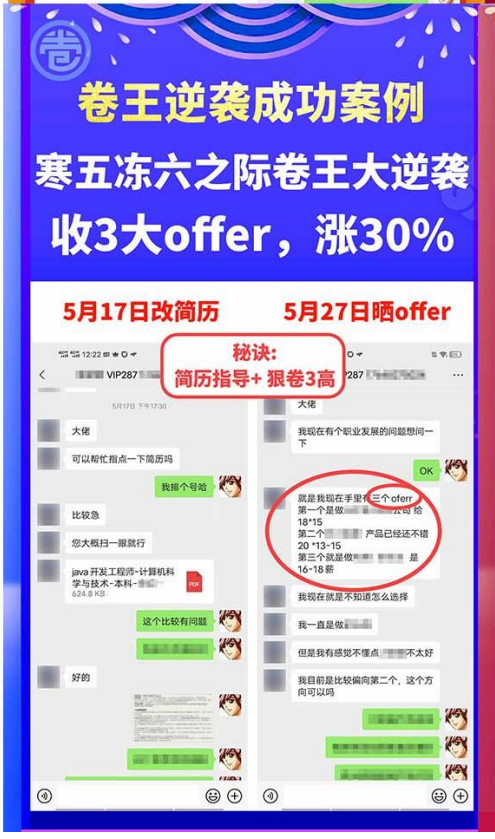

小伙5月份"被毕业"，改简历后
斩获顶级央企Offer
涨薪7000+
5月29日改简历
7月5日晒offer
秘诀:
简历指导+ 狠卷3高
快速看书，就要不求甚解，把目录和 场景大概一下，然后重点的地方，用到的地方，再去回炉
尼恩 我被“ 毕业 ”了
这周末或下周我你改一下简历
毕业了没有关系
ok，发我吧
it行业，跳来跳去，太频繁啦
嗯，其实有点心理准备
简历
不太会写简历
尼恩 我拿到的offer了
涨20%，不还价的
看起来不差钱呀
超过了8000没
平均算下来
7000多
好的
有啥面试的心得吗
可以分享给其他小伙伴的


卷王逆袭成功案例
武汉6年喜收4个优质offer
最高的年薪35W
2月9日改简历
4月15日晒offer
尼恩老师，新年好!
能帮忙修改下简历吗?
金三银四准备跳了
可以的
java 开发-6年-简历-.pdf
拜托了，恩师。希望能拿 25k 回来给你报喜
好，我加一下
还有吗?
面试法宝:
改简历 + 实操
恩哥，决赛圈 offer 了
截图是以我目前认知能写出来的评分了，麻烦帮我参考下
选择大于努力，恩师助我上岸
这么多 offer，我看看哈
都是恩哥指点有方，本来还有个新能源汽车的，35w给拒了，主要太远了
跟着恩哥卷的时间太短了，目前实力也只能到这儿了
这里边有个大数据的，感觉也不错

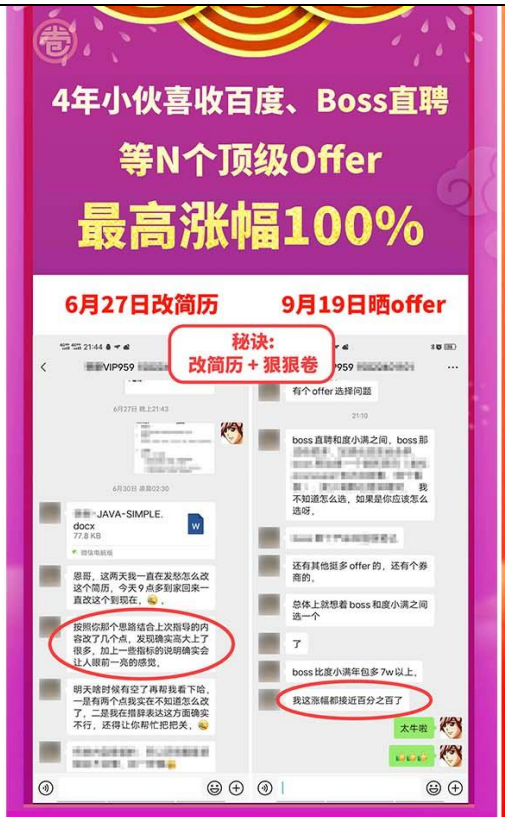

卷王逆袭成功案例
6年小伙喜提4个Offer
最高涨9k，年薪35W
4月14日改简历
5月17日晒offer
Java 开发工程师_.docx
你看看我给你改的
好的呀
麻烦大佬了
这个你自己刷哈
不对的，你都自己刷
那我照着这个改一下库存系统呀
一个简历，
这么漂亮的简历，涨 50%，已经没啥问题
只要准备好，不出大纰漏，基本没问题啦
涨薪法宝:
改简历 + 狠狠卷
保密押金收起哈，你的 offer 最高涨了 9k，多返现 100
好的
谢谢大佬
后面继续狠狠卷哈，感觉卷的时间
透彻
嗯嗯
我准备这一年的时间都看呢
加油卷哈
感觉自己学的不太透彻了
嗯嗯
跟着大佬一起
我周围好几个年薪百万的，都是这


卷王逆袭成功案例
5年经验小伙收2个offer
最高涨薪8k，年薪42W
5月9日改简历
5月30日晒offer
秘诀:
简历指导+ 狠卷3高
以此为样
大家狠狠卷
打造最卷IT社群


卷王逆袭成功案例
非全日制 6年经验卷王
喜提3个Offer，年包30W
5月9日改简历
5月18日晒offer
面试法宝:
改简历 + 狠狠卷


卷王逆袭成功案例
寒五冻六之际卷王大逆袭
收3大offer，涨30%
5月17日改简历
5月27日晒offer
秘诀:
简历指导+ 狠卷3高


卷王逆袭成功案例
4年卷王入职微软，涨50%
3月7日改简历
5月12日晒offer
涨薪法宝:
改简历 + 狠狠卷


4年小伙喜收百度、Boss直聘
等N个顶级Offer
最高涨幅100%
6月27日改简历
9月19日晒offer
秘诀:
改简历＋狠狠卷


卷王逆袭成功案例
4年卷王入收2个offer，涨50%
3月23日改简历
5月12日晒offer
涨薪法宝:
改简历＋狠狠卷


小伙大三暑期很焦虑
跟着尼恩卷一年
校招斩获顶级央企Offer
去年5月19日加入VIP群
今年7月5日晒offer
秘诀:
狠狠卷 书+视频


小伙高中学历
薪酬涨120%
5月6日改简历
7月22日晒offer
秘诀:
改简历＋狠狠卷


卷王逆袭成功案例
非全日制卷王 面试3家
收2个offer 涨薪30%
4月13日改简历
4月21日晒offer
面试法宝：
改简历 + 面试题


5年卷王喜收2大Offer
最高涨5K
5月19日改简历
9月13日晒offer
秘诀：
改简历 + 狠狠卷


卷王逆袭成功案例
3年经验卷王，涨60%
4月16日改简历
5月11日晒offer
涨薪法宝：
改简历 + 狠狠卷


卷王逆袭成功案例
双非二本小伙春招大翻身
喜提9大offer
2月22日改简历
4月13日晒offer
面试法宝：
改简历 + IM实操
9大offer
最高年薪30万

修改简历找尼恩（资深简历优化专家）

- 如果面试表达不好，尼恩会提供 简历优化指导
- 如果项目没有亮点，尼恩会提供 项目亮点指导
- 如果面试表达不好，尼恩会提供 面试表达指导

作为 40 岁老架构师，尼恩长期承担技术面试官的角色：

- **从业以来，“阅历”无数，对简历有着点石成金、改头换面、脱胎换骨**的指导能力。
- **尼恩指导过刚刚就业的小白，也指导过 P8 级的老专家，都指导他们上岸。**

如何联系尼恩。尼恩微信，请参考下面的地址：

语雀：https://www.yuque.com/crazymakercircle/gkkw8s/khigna
码云：https://gitee.com/crazymaker/SimpleCrayIM/blob/master/疯狂创客圈总目录.md