**java设计模式汇总**

置顶 2017年09月28日 16:16:30 [zhangle1hao](https://me.csdn.net/zhangle1hao) 阅读数：2869

**前言**

翻译与github 上 java design 项目，旨在提供java设计模式的速查。

**引用**   
[[3]java 设计模式](https://github.com/iluwatar/java-design-patterns)

**目录**

* [前言](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E5%89%8D%E8%A8%80)
* [目录](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E7%9B%AE%E5%BD%95)
* [抽象文档](https://blog.csdn.net/zhangle1hao/article/details/78124910#1%E6%8A%BD%E8%B1%A1%E6%96%87%E6%A1%A3)
  + [意图](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%84%8F%E5%9B%BE)
  + [适用范围](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E9%80%82%E7%94%A8%E8%8C%83%E5%9B%B4)
  + [权威信息](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%9D%83%E5%A8%81%E4%BF%A1%E6%81%AF)
  + [例子实现分析](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E4%BE%8B%E5%AD%90%E5%AE%9E%E7%8E%B0%E5%88%86%E6%9E%90)
* [抽象工厂](https://blog.csdn.net/zhangle1hao/article/details/78124910#2%E6%8A%BD%E8%B1%A1%E5%B7%A5%E5%8E%82)
  + [意图](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%84%8F%E5%9B%BE-1)
  + [解释](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E8%A7%A3%E9%87%8A)
  + [适用范围](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E9%80%82%E7%94%A8%E8%8C%83%E5%9B%B4-1)
  + [用例](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E7%94%A8%E4%BE%8B)
  + [结论](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E7%BB%93%E8%AE%BA)
  + [现实世界的例子](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E7%8E%B0%E5%AE%9E%E4%B8%96%E7%95%8C%E7%9A%84%E4%BE%8B%E5%AD%90)
  + [权威信息](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%9D%83%E5%A8%81%E4%BF%A1%E6%81%AF-1)
* [适配器模式](https://blog.csdn.net/zhangle1hao/article/details/78124910#3%E9%80%82%E9%85%8D%E5%99%A8%E6%A8%A1%E5%BC%8F)
  + [意图](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%84%8F%E5%9B%BE-2)
  + [解释](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E8%A7%A3%E9%87%8A-1)
  + [使用范围](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E4%BD%BF%E7%94%A8%E8%8C%83%E5%9B%B4)
  + [结论](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E7%BB%93%E8%AE%BA-1)
  + [真实例子](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E7%9C%9F%E5%AE%9E%E4%BE%8B%E5%AD%90)
  + [权威信息](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%9D%83%E5%A8%81%E4%BF%A1%E6%81%AF-2)
* [聚合器-微服务设计模式](https://blog.csdn.net/zhangle1hao/article/details/78124910#4%E8%81%9A%E5%90%88%E5%99%A8-%E5%BE%AE%E6%9C%8D%E5%8A%A1%E8%AE%BE%E8%AE%A1%E6%A8%A1%E5%BC%8F)
  + [Intent](https://blog.csdn.net/zhangle1hao/article/details/78124910#intent)
  + [适用范围](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E9%80%82%E7%94%A8%E8%8C%83%E5%9B%B4-2)
  + [权威信息](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%9D%83%E5%A8%81%E4%BF%A1%E6%81%AF-3)
* [api-网关](https://blog.csdn.net/zhangle1hao/article/details/78124910#5api-%E7%BD%91%E5%85%B3)
  + [意图](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%84%8F%E5%9B%BE-3)
  + [适用范围](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E9%80%82%E7%94%A8%E8%8C%83%E5%9B%B4-3)
  + [权威信息](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%9D%83%E5%A8%81%E4%BF%A1%E6%81%AF-4)
* [异步方法调用](https://blog.csdn.net/zhangle1hao/article/details/78124910#6%E5%BC%82%E6%AD%A5%E6%96%B9%E6%B3%95%E8%B0%83%E7%94%A8)
  + [意图](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%84%8F%E5%9B%BE-4)
  + [适用范围](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E9%80%82%E7%94%A8%E8%8C%83%E5%9B%B4-4)
  + [真实例子](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E7%9C%9F%E5%AE%9E%E4%BE%8B%E5%AD%90-1)
* [阻塞模式](https://blog.csdn.net/zhangle1hao/article/details/78124910#7%E9%98%BB%E5%A1%9E%E6%A8%A1%E5%BC%8F)
  + [意图](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%84%8F%E5%9B%BE-5)
  + [适用范围](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E9%80%82%E7%94%A8%E8%8C%83%E5%9B%B4-5)
  + [相关模式](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E7%9B%B8%E5%85%B3%E6%A8%A1%E5%BC%8F)
* [桥接模式](https://blog.csdn.net/zhangle1hao/article/details/78124910#8%E6%A1%A5%E6%8E%A5%E6%A8%A1%E5%BC%8F)
  + [意图](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%84%8F%E5%9B%BE-6)
  + [解释](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E8%A7%A3%E9%87%8A-2)
  + [适用范围](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E9%80%82%E7%94%A8%E8%8C%83%E5%9B%B4-6)
  + [权威信息](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%9D%83%E5%A8%81%E4%BF%A1%E6%81%AF-5)
* [构建模式](https://blog.csdn.net/zhangle1hao/article/details/78124910#9%E6%9E%84%E5%BB%BA%E6%A8%A1%E5%BC%8F)
  + [意图](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%84%8F%E5%9B%BE-7)
  + [解释](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E8%A7%A3%E9%87%8A-3)
  + [适用范围](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E9%80%82%E7%94%A8%E8%8C%83%E5%9B%B4-7)
  + [真实例子](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E7%9C%9F%E5%AE%9E%E4%BE%8B%E5%AD%90-2)
  + [权威信息](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%9D%83%E5%A8%81%E4%BF%A1%E6%81%AF-6)
* [业务委托](https://blog.csdn.net/zhangle1hao/article/details/78124910#11%E4%B8%9A%E5%8A%A1%E5%A7%94%E6%89%98)
  + [意图](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%84%8F%E5%9B%BE-8)
  + [适用范围](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E9%80%82%E7%94%A8%E8%8C%83%E5%9B%B4-8)
  + [权威信息](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%9D%83%E5%A8%81%E4%BF%A1%E6%81%AF-7)
* [缓存模式](https://blog.csdn.net/zhangle1hao/article/details/78124910#12%E7%BC%93%E5%AD%98%E6%A8%A1%E5%BC%8F)
  + [意图](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%84%8F%E5%9B%BE-9)
  + [适用范围](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E9%80%82%E7%94%A8%E8%8C%83%E5%9B%B4-9)
  + [权威信息](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%9D%83%E5%A8%81%E4%BF%A1%E6%81%AF-8)
* [回调模式](https://blog.csdn.net/zhangle1hao/article/details/78124910#13%E5%9B%9E%E8%B0%83%E6%A8%A1%E5%BC%8F)
  + [意图](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%84%8F%E5%9B%BE-10)
  + [适用范围](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E9%80%82%E7%94%A8%E8%8C%83%E5%9B%B4-10)
  + [真实例子](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E7%9C%9F%E5%AE%9E%E4%BE%8B%E5%AD%90-3)
* [责任链模式](https://blog.csdn.net/zhangle1hao/article/details/78124910#14%E8%B4%A3%E4%BB%BB%E9%93%BE%E6%A8%A1%E5%BC%8F)
  + [意图](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%84%8F%E5%9B%BE-11)
  + [解释](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E8%A7%A3%E9%87%8A-4)
  + [适用范围](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E9%80%82%E7%94%A8%E8%8C%83%E5%9B%B4-11)
  + [真实例子](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E7%9C%9F%E5%AE%9E%E4%BE%8B%E5%AD%90-4)
  + [权威信息](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%9D%83%E5%A8%81%E4%BF%A1%E6%81%AF-9)
* [命令模式](https://blog.csdn.net/zhangle1hao/article/details/78124910#15%E5%91%BD%E4%BB%A4%E6%A8%A1%E5%BC%8F)
  + [意图](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%84%8F%E5%9B%BE-12)
  + [适用范围](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E9%80%82%E7%94%A8%E8%8C%83%E5%9B%B4-12)
  + [典型用例](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E5%85%B8%E5%9E%8B%E7%94%A8%E4%BE%8B)
  + [真实例子](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E7%9C%9F%E5%AE%9E%E4%BE%8B%E5%AD%90-5)
  + [权威信息](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%9D%83%E5%A8%81%E4%BF%A1%E6%81%AF-10)
* [组合模式](https://blog.csdn.net/zhangle1hao/article/details/78124910#16%E7%BB%84%E5%90%88%E6%A8%A1%E5%BC%8F)
  + [意图](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%84%8F%E5%9B%BE-13)
  + [解释](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E8%A7%A3%E9%87%8A-5)
  + [适用范围](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E9%80%82%E7%94%A8%E8%8C%83%E5%9B%B4-13)
  + [真实例子](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E7%9C%9F%E5%AE%9E%E4%BE%8B%E5%AD%90-6)
  + [权威信息](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%9D%83%E5%A8%81%E4%BF%A1%E6%81%AF-11)
* [转换器模式](https://blog.csdn.net/zhangle1hao/article/details/78124910#17%E8%BD%AC%E6%8D%A2%E5%99%A8%E6%A8%A1%E5%BC%8F)
  + [意图](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%84%8F%E5%9B%BE-14)
  + [适用范围](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E9%80%82%E7%94%A8%E8%8C%83%E5%9B%B4-14)
  + [权威信息](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%9D%83%E5%A8%81%E4%BF%A1%E6%81%AF-12)
* [命令查询的责任分离模式](https://blog.csdn.net/zhangle1hao/article/details/78124910#18%E5%91%BD%E4%BB%A4%E6%9F%A5%E8%AF%A2%E7%9A%84%E8%B4%A3%E4%BB%BB%E5%88%86%E7%A6%BB%E6%A8%A1%E5%BC%8F)
  + [意图](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%84%8F%E5%9B%BE-15)
  + [适用范围](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E9%80%82%E7%94%A8%E8%8C%83%E5%9B%B4-15)
  + [权威信息](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%9D%83%E5%A8%81%E4%BF%A1%E6%81%AF-13)
* [数据访问对象模式](https://blog.csdn.net/zhangle1hao/article/details/78124910#19%E6%95%B0%E6%8D%AE%E8%AE%BF%E9%97%AE%E5%AF%B9%E8%B1%A1%E6%A8%A1%E5%BC%8F)
  + [意图](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%84%8F%E5%9B%BE-16)
  + [适用范围](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E9%80%82%E7%94%A8%E8%8C%83%E5%9B%B4-16)
  + [权威信息](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%9D%83%E5%A8%81%E4%BF%A1%E6%81%AF-14)
* [数据总线模式](https://blog.csdn.net/zhangle1hao/article/details/78124910#20%E6%95%B0%E6%8D%AE%E6%80%BB%E7%BA%BF%E6%A8%A1%E5%BC%8F)
  + [意图](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%84%8F%E5%9B%BE-17)
  + [适用访问](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E9%80%82%E7%94%A8%E8%AE%BF%E9%97%AE)
  + [相关模式](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E7%9B%B8%E5%85%B3%E6%A8%A1%E5%BC%8F-1)
* [数据映射模式](https://blog.csdn.net/zhangle1hao/article/details/78124910#21%E6%95%B0%E6%8D%AE%E6%98%A0%E5%B0%84%E6%A8%A1%E5%BC%8F)
  + [意图](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%84%8F%E5%9B%BE-18)
  + [适用范围](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E9%80%82%E7%94%A8%E8%8C%83%E5%9B%B4-17)
  + [权威信息](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%9D%83%E5%A8%81%E4%BF%A1%E6%81%AF-15)
* [数据传输对象模式](https://blog.csdn.net/zhangle1hao/article/details/78124910#22%E6%95%B0%E6%8D%AE%E4%BC%A0%E8%BE%93%E5%AF%B9%E8%B1%A1%E6%A8%A1%E5%BC%8F)
  + [意图](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%84%8F%E5%9B%BE-19)
  + [适用范围](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E9%80%82%E7%94%A8%E8%8C%83%E5%9B%B4-18)
  + [权威信息](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%9D%83%E5%A8%81%E4%BF%A1%E6%81%AF-16)
* [装饰模式](https://blog.csdn.net/zhangle1hao/article/details/78124910#23%E8%A3%85%E9%A5%B0%E6%A8%A1%E5%BC%8F)
  + [众所周知](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E4%BC%97%E6%89%80%E5%91%A8%E7%9F%A5)
  + [意图](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%84%8F%E5%9B%BE-20)
  + [解释](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E8%A7%A3%E9%87%8A-6)
  + [适用范围](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E9%80%82%E7%94%A8%E8%8C%83%E5%9B%B4-19)
  + [真实世界](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E7%9C%9F%E5%AE%9E%E4%B8%96%E7%95%8C)
  + [权威信息](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%9D%83%E5%A8%81%E4%BF%A1%E6%81%AF-17)
* [委托模式](https://blog.csdn.net/zhangle1hao/article/details/78124910#24%E5%A7%94%E6%89%98%E6%A8%A1%E5%BC%8F)
  + [众所周知](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E4%BC%97%E6%89%80%E5%91%A8%E7%9F%A5-1)
  + [意图](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%84%8F%E5%9B%BE-21)
  + [适用范围](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E9%80%82%E7%94%A8%E8%8C%83%E5%9B%B4-20)
  + [权威信息](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%9D%83%E5%A8%81%E4%BF%A1%E6%81%AF-18)
* [依赖注入模式](https://blog.csdn.net/zhangle1hao/article/details/78124910#25%E4%BE%9D%E8%B5%96%E6%B3%A8%E5%85%A5%E6%A8%A1%E5%BC%8F)
  + [意图](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%84%8F%E5%9B%BE-22)
  + [适用范围](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E9%80%82%E7%94%A8%E8%8C%83%E5%9B%B4-21)
* [双重锁模式](https://blog.csdn.net/zhangle1hao/article/details/78124910#26%E5%8F%8C%E9%87%8D%E9%94%81%E6%A8%A1%E5%BC%8F)
  + [意图](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%84%8F%E5%9B%BE-23)
  + [适用范围](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E9%80%82%E7%94%A8%E8%8C%83%E5%9B%B4-22)
* [双重派对模式](https://blog.csdn.net/zhangle1hao/article/details/78124910#27%E5%8F%8C%E9%87%8D%E6%B4%BE%E5%AF%B9%E6%A8%A1%E5%BC%8F)
  + [意图](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%84%8F%E5%9B%BE-24)
  + [适用范围](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E9%80%82%E7%94%A8%E8%8C%83%E5%9B%B4-23)
  + [真实例子](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E7%9C%9F%E5%AE%9E%E4%BE%8B%E5%AD%90-7)
* [执行程序流监测器](https://blog.csdn.net/zhangle1hao/article/details/78124910#28-%E6%89%A7%E8%A1%8C%E7%A8%8B%E5%BA%8F%E6%B5%81%E7%9B%91%E6%B5%8B%E5%99%A8)
  + [意图](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%84%8F%E5%9B%BE-25)
  + [适用范围](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E9%80%82%E7%94%A8%E8%8C%83%E5%9B%B4-24)
  + [权威信息](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%9D%83%E5%A8%81%E4%BF%A1%E6%81%AF-19)
* [事件聚合器](https://blog.csdn.net/zhangle1hao/article/details/78124910#29-%E4%BA%8B%E4%BB%B6%E8%81%9A%E5%90%88%E5%99%A8)
  + [意图](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%84%8F%E5%9B%BE-26)
  + [适用范围](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E9%80%82%E7%94%A8%E8%8C%83%E5%9B%B4-25)
  + [权威信息](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%9D%83%E5%A8%81%E4%BF%A1%E6%81%AF-20)
* [事件异步](https://blog.csdn.net/zhangle1hao/article/details/78124910#30-%E4%BA%8B%E4%BB%B6%E5%BC%82%E6%AD%A5)
  + [意图](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%84%8F%E5%9B%BE-27)
  + [适用范围](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E9%80%82%E7%94%A8%E8%8C%83%E5%9B%B4-26)
  + [权威信息](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%9D%83%E5%A8%81%E4%BF%A1%E6%81%AF-21)
* [基于事件驱动的体系结构](https://blog.csdn.net/zhangle1hao/article/details/78124910#31-%E5%9F%BA%E4%BA%8E%E4%BA%8B%E4%BB%B6%E9%A9%B1%E5%8A%A8%E7%9A%84%E4%BD%93%E7%B3%BB%E7%BB%93%E6%9E%84)
  + [意图](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%84%8F%E5%9B%BE-28)
  + [适用范围](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E9%80%82%E7%94%A8%E8%8C%83%E5%9B%B4-27)
  + [真实例子](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E7%9C%9F%E5%AE%9E%E4%BE%8B%E5%AD%90-8)
  + [权威信息](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%9D%83%E5%A8%81%E4%BF%A1%E6%81%AF-22)
* [事件队列](https://blog.csdn.net/zhangle1hao/article/details/78124910#32-%E4%BA%8B%E4%BB%B6%E9%98%9F%E5%88%97)
  + [意图](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%84%8F%E5%9B%BE-29)
  + [适用范围](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E9%80%82%E7%94%A8%E8%8C%83%E5%9B%B4-28)
  + [权威信息](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%9D%83%E5%A8%81%E4%BF%A1%E6%81%AF-23)
* [事件捕捉](https://blog.csdn.net/zhangle1hao/article/details/78124910#33-%E4%BA%8B%E4%BB%B6%E6%8D%95%E6%8D%89)
  + [意图](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%84%8F%E5%9B%BE-30)
  + [适用范围](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E9%80%82%E7%94%A8%E8%8C%83%E5%9B%B4-29)
  + [真实世界](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E7%9C%9F%E5%AE%9E%E4%B8%96%E7%95%8C-1)
  + [权威信息](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%9D%83%E5%A8%81%E4%BF%A1%E6%81%AF-24)
* [执行环绕](https://blog.csdn.net/zhangle1hao/article/details/78124910#34-%E6%89%A7%E8%A1%8C%E7%8E%AF%E7%BB%95)
  + [意图](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%84%8F%E5%9B%BE-31)
  + [适用范围](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E9%80%82%E7%94%A8%E8%8C%83%E5%9B%B4-30)
  + [权威信息](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%9D%83%E5%A8%81%E4%BF%A1%E6%81%AF-25)
* [3 扩展对象](https://blog.csdn.net/zhangle1hao/article/details/78124910#3%EF%BC%95-%E6%89%A9%E5%B1%95%E5%AF%B9%E8%B1%A1)
  + [意图](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%84%8F%E5%9B%BE-32)
  + [适用范围](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E9%80%82%E7%94%A8%E8%8C%83%E5%9B%B4-31)
  + [真实世界](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E7%9C%9F%E5%AE%9E%E4%B8%96%E7%95%8C-2)
* [外观模式](https://blog.csdn.net/zhangle1hao/article/details/78124910#36-%E5%A4%96%E8%A7%82%E6%A8%A1%E5%BC%8F)
  + [意图](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%84%8F%E5%9B%BE-33)
  + [解释](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E8%A7%A3%E9%87%8A-7)
  + [适用范围](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E9%80%82%E7%94%A8%E8%8C%83%E5%9B%B4-32)
  + [权威信息](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%9D%83%E5%A8%81%E4%BF%A1%E6%81%AF-26)
* [工厂套件](https://blog.csdn.net/zhangle1hao/article/details/78124910#37-%E5%B7%A5%E5%8E%82%E5%A5%97%E4%BB%B6)
  + [意图](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%84%8F%E5%9B%BE-34)
  + [适用范围](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E9%80%82%E7%94%A8%E8%8C%83%E5%9B%B4-33)
  + [权威信息](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%9D%83%E5%A8%81%E4%BF%A1%E6%81%AF-27)
* [工厂方法](https://blog.csdn.net/zhangle1hao/article/details/78124910#38-%E5%B7%A5%E5%8E%82%E6%96%B9%E6%B3%95)
  + [众所周知](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E4%BC%97%E6%89%80%E5%91%A8%E7%9F%A5-2)
  + [意图](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%84%8F%E5%9B%BE-35)
  + [解释](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E8%A7%A3%E9%87%8A-8)
  + [适用范围](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E9%80%82%E7%94%A8%E8%8C%83%E5%9B%B4-34)
  + [真实例子](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E7%9C%9F%E5%AE%9E%E4%BE%8B%E5%AD%90-9)
  + [权威信息](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%9D%83%E5%A8%81%E4%BF%A1%E6%81%AF-28)
* [功能切换](https://blog.csdn.net/zhangle1hao/article/details/78124910#39-%E5%8A%9F%E8%83%BD%E5%88%87%E6%8D%A2)
  + [众所周知](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E4%BC%97%E6%89%80%E5%91%A8%E7%9F%A5-3)
  + [意图](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%84%8F%E5%9B%BE-36)
  + [适用范围](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E9%80%82%E7%94%A8%E8%8C%83%E5%9B%B4-35)
  + [权威信息](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%9D%83%E5%A8%81%E4%BF%A1%E6%81%AF-29)
* [流畅的接口](https://blog.csdn.net/zhangle1hao/article/details/78124910#40-%E6%B5%81%E7%95%85%E7%9A%84%E6%8E%A5%E5%8F%A3)
  + [意图](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%84%8F%E5%9B%BE-37)
  + [实现](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E5%AE%9E%E7%8E%B0)
  + [适用范围](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E9%80%82%E7%94%A8%E8%8C%83%E5%9B%B4-36)
  + [真实例子](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E7%9C%9F%E5%AE%9E%E4%BE%8B%E5%AD%90-10)
  + [权威信息](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%9D%83%E5%A8%81%E4%BF%A1%E6%81%AF-30)
* [流量模式](https://blog.csdn.net/zhangle1hao/article/details/78124910#41-%E6%B5%81%E9%87%8F%E6%A8%A1%E5%BC%8F)
  + [意图](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%84%8F%E5%9B%BE-38)
  + [适用范围](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E9%80%82%E7%94%A8%E8%8C%83%E5%9B%B4-37)
  + [权威信息](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%9D%83%E5%A8%81%E4%BF%A1%E6%81%AF-31)
* [享元模式](https://blog.csdn.net/zhangle1hao/article/details/78124910#42-%E4%BA%AB%E5%85%83%E6%A8%A1%E5%BC%8F)
  + [意图](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%84%8F%E5%9B%BE-39)
  + [例子](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E4%BE%8B%E5%AD%90)
  + [适用范围](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E9%80%82%E7%94%A8%E8%8C%83%E5%9B%B4-38)
  + [真实世界](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E7%9C%9F%E5%AE%9E%E4%B8%96%E7%95%8C-3)
  + [权威信息](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%9D%83%E5%A8%81%E4%BF%A1%E6%81%AF-32)
* [前台控制](https://blog.csdn.net/zhangle1hao/article/details/78124910#43-%E5%89%8D%E5%8F%B0%E6%8E%A7%E5%88%B6)
  + [意图](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%84%8F%E5%9B%BE-40)
  + [适用范围](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E9%80%82%E7%94%A8%E8%8C%83%E5%9B%B4-39)
  + [真实例子](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E7%9C%9F%E5%AE%9E%E4%BE%8B%E5%AD%90-11)
  + [权威信息](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%9D%83%E5%A8%81%E4%BF%A1%E6%81%AF-33)
* [被监视的挂起](https://blog.csdn.net/zhangle1hao/article/details/78124910#44-%E8%A2%AB%E7%9B%91%E8%A7%86%E7%9A%84%E6%8C%82%E8%B5%B7)
  + [意图](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%84%8F%E5%9B%BE-41)
  + [适用范围](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E9%80%82%E7%94%A8%E8%8C%83%E5%9B%B4-40)
  + [关联模式](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E5%85%B3%E8%81%94%E6%A8%A1%E5%BC%8F)
* [半同步半异步](https://blog.csdn.net/zhangle1hao/article/details/78124910#45-%E5%8D%8A%E5%90%8C%E6%AD%A5%E5%8D%8A%E5%BC%82%E6%AD%A5)
  + [意图](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%84%8F%E5%9B%BE-42)
  + [适用范围](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E9%80%82%E7%94%A8%E8%8C%83%E5%9B%B4-41)
  + [真实世界](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E7%9C%9F%E5%AE%9E%E4%B8%96%E7%95%8C-4)
  + [权威信息](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%9D%83%E5%A8%81%E4%BF%A1%E6%81%AF-34)
* [六角结构又名端口适配器结构](https://blog.csdn.net/zhangle1hao/article/details/78124910#46-%E5%85%AD%E8%A7%92%E7%BB%93%E6%9E%84%E5%8F%88%E5%90%8D%E7%AB%AF%E5%8F%A3%E9%80%82%E9%85%8D%E5%99%A8%E7%BB%93%E6%9E%84)
  + [众所周知](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E4%BC%97%E6%89%80%E5%91%A8%E7%9F%A5-4)
  + [意图](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%84%8F%E5%9B%BE-43)
  + [适用范围](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E9%80%82%E7%94%A8%E8%8C%83%E5%9B%B4-42)
  + [教程](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%95%99%E7%A8%8B)
  + [演讲](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%BC%94%E8%AE%B2)
  + [真实世界](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E7%9C%9F%E5%AE%9E%E4%B8%96%E7%95%8C-5)
  + [权威信息](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%9D%83%E5%A8%81%E4%BF%A1%E6%81%AF-35)
* [拦截过滤](https://blog.csdn.net/zhangle1hao/article/details/78124910#47-%E6%8B%A6%E6%88%AA%E8%BF%87%E6%BB%A4)
  + [意图](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%84%8F%E5%9B%BE-44)
  + [适用模式](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E9%80%82%E7%94%A8%E6%A8%A1%E5%BC%8F)
  + [真实世界](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E7%9C%9F%E5%AE%9E%E4%B8%96%E7%95%8C-6)
  + [权威信息](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%9D%83%E5%A8%81%E4%BF%A1%E6%81%AF-36)
* [解释器模式](https://blog.csdn.net/zhangle1hao/article/details/78124910#48-%E8%A7%A3%E9%87%8A%E5%99%A8%E6%A8%A1%E5%BC%8F)
  + [意图](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%84%8F%E5%9B%BE-45)
  + [适用范围](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E9%80%82%E7%94%A8%E8%8C%83%E5%9B%B4-43)
  + [真实世界](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E7%9C%9F%E5%AE%9E%E4%B8%96%E7%95%8C-7)
  + [权威信息](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%9D%83%E5%A8%81%E4%BF%A1%E6%81%AF-37)
* [迭代器模式](https://blog.csdn.net/zhangle1hao/article/details/78124910#49-%E8%BF%AD%E4%BB%A3%E5%99%A8%E6%A8%A1%E5%BC%8F)
  + [众所周知](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E4%BC%97%E6%89%80%E5%91%A8%E7%9F%A5-5)
  + [意图](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%84%8F%E5%9B%BE-46)
  + [适用范围](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E9%80%82%E7%94%A8%E8%8C%83%E5%9B%B4-44)
  + [真实例子](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E7%9C%9F%E5%AE%9E%E4%BE%8B%E5%AD%90-12)
  + [权威信息](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%9D%83%E5%A8%81%E4%BF%A1%E6%81%AF-38)
* [层模式](https://blog.csdn.net/zhangle1hao/article/details/78124910#50-%E5%B1%82%E6%A8%A1%E5%BC%8F)
  + [意图](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%84%8F%E5%9B%BE-47)
  + [适用范围](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E9%80%82%E7%94%A8%E8%8C%83%E5%9B%B4-45)
  + [权威信息](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%9D%83%E5%A8%81%E4%BF%A1%E6%81%AF-39)
* [懒加载](https://blog.csdn.net/zhangle1hao/article/details/78124910#51-%E6%87%92%E5%8A%A0%E8%BD%BD)
  + [意图](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%84%8F%E5%9B%BE-48)
  + [适用范围](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E9%80%82%E7%94%A8%E8%8C%83%E5%9B%B4-46)
  + [真实例子](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E7%9C%9F%E5%AE%9E%E4%BE%8B%E5%AD%90-13)
  + [权威信息](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%9D%83%E5%A8%81%E4%BF%A1%E6%81%AF-40)
* [标记接口](https://blog.csdn.net/zhangle1hao/article/details/78124910#52-%E6%A0%87%E8%AE%B0%E6%8E%A5%E5%8F%A3)
  + [意图](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%84%8F%E5%9B%BE-49)
  + [适用范围](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E9%80%82%E7%94%A8%E8%8C%83%E5%9B%B4-47)
  + [真实例子](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E7%9C%9F%E5%AE%9E%E4%BE%8B%E5%AD%90-14)
  + [权威信息](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%9D%83%E5%A8%81%E4%BF%A1%E6%81%AF-41)
* [中间人模式](https://blog.csdn.net/zhangle1hao/article/details/78124910#53-%E4%B8%AD%E9%97%B4%E4%BA%BA%E6%A8%A1%E5%BC%8F)
  + [意图](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%84%8F%E5%9B%BE-50)
  + [适用范围](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E9%80%82%E7%94%A8%E8%8C%83%E5%9B%B4-48)
  + [真实例子](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E7%9C%9F%E5%AE%9E%E4%BE%8B%E5%AD%90-15)
  + [权威信息](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%9D%83%E5%A8%81%E4%BF%A1%E6%81%AF-42)
* [备忘录模式](https://blog.csdn.net/zhangle1hao/article/details/78124910#54-%E5%A4%87%E5%BF%98%E5%BD%95%E6%A8%A1%E5%BC%8F)
  + [众所周知](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E4%BC%97%E6%89%80%E5%91%A8%E7%9F%A5-6)
  + [意图](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%84%8F%E5%9B%BE-51)
  + [适用范围](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E9%80%82%E7%94%A8%E8%8C%83%E5%9B%B4-49)
  + [真实世界](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E7%9C%9F%E5%AE%9E%E4%B8%96%E7%95%8C-8)
  + [权威信息](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%9D%83%E5%A8%81%E4%BF%A1%E6%81%AF-43)
* [消息渠道](https://blog.csdn.net/zhangle1hao/article/details/78124910#55%E6%B6%88%E6%81%AF%E6%B8%A0%E9%81%93)
  + [意图](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%84%8F%E5%9B%BE-52)
  + [适用范围](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E9%80%82%E7%94%A8%E8%8C%83%E5%9B%B4-50)
  + [真实例子](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E7%9C%9F%E5%AE%9E%E4%BE%8B%E5%AD%90-16)
* [模型-视图-控制器](https://blog.csdn.net/zhangle1hao/article/details/78124910#56%E6%A8%A1%E5%9E%8B-%E8%A7%86%E5%9B%BE-%E6%8E%A7%E5%88%B6%E5%99%A8)
  + [意图](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%84%8F%E5%9B%BE-53)
  + [适用范围](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E9%80%82%E7%94%A8%E8%8C%83%E5%9B%B4-51)
  + [权威信息](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%9D%83%E5%A8%81%E4%BF%A1%E6%81%AF-44)
  + [模型-视图-主持人Model-View-Presenter](https://blog.csdn.net/zhangle1hao/article/details/78124910#57%E6%A8%A1%E5%9E%8B-%E8%A7%86%E5%9B%BE-%E4%B8%BB%E6%8C%81%E4%BA%BAmodel-view-presenter)
  + [意图](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%84%8F%E5%9B%BE-54)
  + [适用范围](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E9%80%82%E7%94%A8%E8%8C%83%E5%9B%B4-52)
  + [真实例子](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E7%9C%9F%E5%AE%9E%E4%BE%8B%E5%AD%90-17)
* [模块化Module](https://blog.csdn.net/zhangle1hao/article/details/78124910#58%E6%A8%A1%E5%9D%97%E5%8C%96module)
  + [意图](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%84%8F%E5%9B%BE-55)
  + [适用范围](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E9%80%82%E7%94%A8%E8%8C%83%E5%9B%B4-53)
  + [权威信息](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%9D%83%E5%A8%81%E4%BF%A1%E6%81%AF-45)
* [Monad](https://blog.csdn.net/zhangle1hao/article/details/78124910#59monad)
  + [意图](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%84%8F%E5%9B%BE-56)
  + [适用范围](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E9%80%82%E7%94%A8%E8%8C%83%E5%9B%B4-54)
  + [权威信息](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%9D%83%E5%A8%81%E4%BF%A1%E6%81%AF-46)
  + [单一的状态monostate](https://blog.csdn.net/zhangle1hao/article/details/78124910#60%E5%8D%95%E4%B8%80%E7%9A%84%E7%8A%B6%E6%80%81monostate)
  + [众所周知](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E4%BC%97%E6%89%80%E5%91%A8%E7%9F%A5-7)
  + [意图](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%84%8F%E5%9B%BE-57)
  + [适用范围](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E9%80%82%E7%94%A8%E8%8C%83%E5%9B%B4-55)
  + [典型用例](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E5%85%B8%E5%9E%8B%E7%94%A8%E4%BE%8B-1)
* [多例模式Multiton](https://blog.csdn.net/zhangle1hao/article/details/78124910#61%E5%A4%9A%E4%BE%8B%E6%A8%A1%E5%BC%8Fmultiton)
  + [众所周知](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E4%BC%97%E6%89%80%E5%91%A8%E7%9F%A5-8)
  + [意图](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%84%8F%E5%9B%BE-58)
  + [适用范围](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E9%80%82%E7%94%A8%E8%8C%83%E5%9B%B4-56)
* [静音模式mute](https://blog.csdn.net/zhangle1hao/article/details/78124910#62%E9%9D%99%E9%9F%B3%E6%A8%A1%E5%BC%8Fmute)
  + [意图](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%84%8F%E5%9B%BE-59)
  + [适用范围](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E9%80%82%E7%94%A8%E8%8C%83%E5%9B%B4-57)
  + [权威信息](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%9D%83%E5%A8%81%E4%BF%A1%E6%81%AF-47)
* [互斥模式](https://blog.csdn.net/zhangle1hao/article/details/78124910#63%E4%BA%92%E6%96%A5%E6%A8%A1%E5%BC%8F)
  + [众所周知](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E4%BC%97%E6%89%80%E5%91%A8%E7%9F%A5-9)
  + [意图](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%84%8F%E5%9B%BE-60)
  + [适用范围](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E9%80%82%E7%94%A8%E8%8C%83%E5%9B%B4-58)
  + [权威信息](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%9D%83%E5%A8%81%E4%BF%A1%E6%81%AF-48)
* [裸露的对象](https://blog.csdn.net/zhangle1hao/article/details/78124910#64-%E8%A3%B8%E9%9C%B2%E7%9A%84%E5%AF%B9%E8%B1%A1)
  + [意图](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%84%8F%E5%9B%BE-61)
  + [适用范围](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E9%80%82%E7%94%A8%E8%8C%83%E5%9B%B4-59)
  + [好的例子](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E5%A5%BD%E7%9A%84%E4%BE%8B%E5%AD%90)
  + [权威信息](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%9D%83%E5%A8%81%E4%BF%A1%E6%81%AF-49)
* [空对象](https://blog.csdn.net/zhangle1hao/article/details/78124910#65%E7%A9%BA%E5%AF%B9%E8%B1%A1)
  + [意图](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%84%8F%E5%9B%BE-62)
  + [适用范围](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E9%80%82%E7%94%A8%E8%8C%83%E5%9B%B4-60)
  + [权威信息](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%9D%83%E5%A8%81%E4%BF%A1%E6%81%AF-50)
* [母对象object-mother](https://blog.csdn.net/zhangle1hao/article/details/78124910#66-%E6%AF%8D%E5%AF%B9%E8%B1%A1object-mother)
  + [意图](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%84%8F%E5%9B%BE-63)
  + [适用范围](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E9%80%82%E7%94%A8%E8%8C%83%E5%9B%B4-61)
  + [权威信息](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%9D%83%E5%A8%81%E4%BF%A1%E6%81%AF-51)
* [对象池object-pool](https://blog.csdn.net/zhangle1hao/article/details/78124910#67-%E5%AF%B9%E8%B1%A1%E6%B1%A0object-pool)
  + [意图](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%84%8F%E5%9B%BE-64)
  + [适用范围](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E9%80%82%E7%94%A8%E8%8C%83%E5%9B%B4-62)
  + [观察者模式](https://blog.csdn.net/zhangle1hao/article/details/78124910#68-%E8%A7%82%E5%AF%9F%E8%80%85%E6%A8%A1%E5%BC%8F)
  + [众所周知](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E4%BC%97%E6%89%80%E5%91%A8%E7%9F%A5-10)
  + [意图](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%84%8F%E5%9B%BE-65)
  + [适用范围](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E9%80%82%E7%94%A8%E8%8C%83%E5%9B%B4-63)
  + [典型用例](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E5%85%B8%E5%9E%8B%E7%94%A8%E4%BE%8B-2)
  + [真实世界](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E7%9C%9F%E5%AE%9E%E4%B8%96%E7%95%8C-9)
  + [权威信息](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%9D%83%E5%A8%81%E4%BF%A1%E6%81%AF-52)
* [页对象](https://blog.csdn.net/zhangle1hao/article/details/78124910#69-%E9%A1%B5%E5%AF%B9%E8%B1%A1)
  + [意图](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%84%8F%E5%9B%BE-66)
  + [适用范围](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E9%80%82%E7%94%A8%E8%8C%83%E5%9B%B4-64)
  + [权威信息](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%9D%83%E5%A8%81%E4%BF%A1%E6%81%AF-53)
* [页对象](https://blog.csdn.net/zhangle1hao/article/details/78124910#70-%E9%A1%B5%E5%AF%B9%E8%B1%A1)
  + [意图](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%84%8F%E5%9B%BE-67)
  + [适用范围](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E9%80%82%E7%94%A8%E8%8C%83%E5%9B%B4-65)
  + [权威信息](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%9D%83%E5%A8%81%E4%BF%A1%E6%81%AF-54)
* [poison-pill](https://blog.csdn.net/zhangle1hao/article/details/78124910#71-poison-pill)
  + [意图](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%84%8F%E5%9B%BE-68)
  + [适用范围](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E9%80%82%E7%94%A8%E8%8C%83%E5%9B%B4-66)
  + [真实例子](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E7%9C%9F%E5%AE%9E%E4%BE%8B%E5%AD%90-18)
* [私有类数据](https://blog.csdn.net/zhangle1hao/article/details/78124910#72-%E7%A7%81%E6%9C%89%E7%B1%BB%E6%95%B0%E6%8D%AE)
  + [意图](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%84%8F%E5%9B%BE-69)
  + [适用范围](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E9%80%82%E7%94%A8%E8%8C%83%E5%9B%B4-67)
* [生产者-消费者](https://blog.csdn.net/zhangle1hao/article/details/78124910#73-%E7%94%9F%E4%BA%A7%E8%80%85-%E6%B6%88%E8%B4%B9%E8%80%85)
  + [意图](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%84%8F%E5%9B%BE-70)
  + [适用范围](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E9%80%82%E7%94%A8%E8%8C%83%E5%9B%B4-68)
* [承诺](https://blog.csdn.net/zhangle1hao/article/details/78124910#74-%E6%89%BF%E8%AF%BA)
  + [意图](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%84%8F%E5%9B%BE-71)
  + [适用范围](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E9%80%82%E7%94%A8%E8%8C%83%E5%9B%B4-69)
  + [真实例子](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E7%9C%9F%E5%AE%9E%E4%BE%8B%E5%AD%90-19)
  + [关联模式](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E5%85%B3%E8%81%94%E6%A8%A1%E5%BC%8F-1)
  + [权威信息](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%9D%83%E5%A8%81%E4%BF%A1%E6%81%AF-55)
* [属性](https://blog.csdn.net/zhangle1hao/article/details/78124910#75-%E5%B1%9E%E6%80%A7)
  + [意图](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%84%8F%E5%9B%BE-72)
  + [适用范围](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E9%80%82%E7%94%A8%E8%8C%83%E5%9B%B4-70)
  + [真实例子](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E7%9C%9F%E5%AE%9E%E4%BE%8B%E5%AD%90-20)
* [原型](https://blog.csdn.net/zhangle1hao/article/details/78124910#76-%E5%8E%9F%E5%9E%8B)
  + [意图](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%84%8F%E5%9B%BE-73)
  + [解释](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E8%A7%A3%E9%87%8A-9)
  + [适用范围](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E9%80%82%E7%94%A8%E8%8C%83%E5%9B%B4-71)
  + [真实例子](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E7%9C%9F%E5%AE%9E%E4%BE%8B%E5%AD%90-21)
  + [权威信息](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%9D%83%E5%A8%81%E4%BF%A1%E6%81%AF-56)
* [代理](https://blog.csdn.net/zhangle1hao/article/details/78124910#77%E4%BB%A3%E7%90%86)
  + [意图](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%84%8F%E5%9B%BE-74)
  + [解释](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E8%A7%A3%E9%87%8A-10)
  + [适用范围](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E9%80%82%E7%94%A8%E8%8C%83%E5%9B%B4-72)
  + [典型用例](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E5%85%B8%E5%9E%8B%E7%94%A8%E4%BE%8B-3)
  + [目录](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E7%9B%AE%E5%BD%95-1)
  + [演讲](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%BC%94%E8%AE%B2-1)
  + [真实世界](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E7%9C%9F%E5%AE%9E%E4%B8%96%E7%95%8C-10)
  + [权威信息](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%9D%83%E5%A8%81%E4%BF%A1%E6%81%AF-57)
* [发布-订阅](https://blog.csdn.net/zhangle1hao/article/details/78124910#78%E5%8F%91%E5%B8%83-%E8%AE%A2%E9%98%85)
  + [意图](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%84%8F%E5%9B%BE-75)
  + [适合范围](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E9%80%82%E5%90%88%E8%8C%83%E5%9B%B4)
  + [权威信息](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%9D%83%E5%A8%81%E4%BF%A1%E6%81%AF-58)
* [基于队列的负载均衡](https://blog.csdn.net/zhangle1hao/article/details/78124910#79%E5%9F%BA%E4%BA%8E%E9%98%9F%E5%88%97%E7%9A%84%E8%B4%9F%E8%BD%BD%E5%9D%87%E8%A1%A1)
  + [意图](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%84%8F%E5%9B%BE-76)
  + [适用范围](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E9%80%82%E7%94%A8%E8%8C%83%E5%9B%B4-73)
  + [目录](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E7%9B%AE%E5%BD%95-2)
  + [真实例子](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E7%9C%9F%E5%AE%9E%E4%BE%8B%E5%AD%90-22)
  + [权威信息](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%9D%83%E5%A8%81%E4%BF%A1%E6%81%AF-59)
* [响应器](https://blog.csdn.net/zhangle1hao/article/details/78124910#80%E5%93%8D%E5%BA%94%E5%99%A8)
  + [意图](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%84%8F%E5%9B%BE-77)
  + [适用范围](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E9%80%82%E7%94%A8%E8%8C%83%E5%9B%B4-74)
  + [真实例子](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E7%9C%9F%E5%AE%9E%E4%BE%8B%E5%AD%90-23)
  + [权威信息](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%9D%83%E5%A8%81%E4%BF%A1%E6%81%AF-60)
* [读写锁](https://blog.csdn.net/zhangle1hao/article/details/78124910#81%E8%AF%BB%E5%86%99%E9%94%81)
  + [意图](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%84%8F%E5%9B%BE-78)
  + [适用范围](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E9%80%82%E7%94%A8%E8%8C%83%E5%9B%B4-75)
  + [真实例子](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E7%9C%9F%E5%AE%9E%E4%BE%8B%E5%AD%90-24)
  + [权威信息](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%9D%83%E5%A8%81%E4%BF%A1%E6%81%AF-61)
* [仓库](https://blog.csdn.net/zhangle1hao/article/details/78124910#82%E4%BB%93%E5%BA%93)
  + [意图](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%84%8F%E5%9B%BE-79)
  + [适用范围](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E9%80%82%E7%94%A8%E8%8C%83%E5%9B%B4-76)
  + [真实例子](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E7%9C%9F%E5%AE%9E%E4%BE%8B%E5%AD%90-25)
  + [权威信息](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%9D%83%E5%A8%81%E4%BF%A1%E6%81%AF-62)
* [资源获取初始化](https://blog.csdn.net/zhangle1hao/article/details/78124910#83%E8%B5%84%E6%BA%90%E8%8E%B7%E5%8F%96%E5%88%9D%E5%A7%8B%E5%8C%96)
  + [意图](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%84%8F%E5%9B%BE-80)
  + [解释](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E8%A7%A3%E9%87%8A-11)
  + [适用范围](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E9%80%82%E7%94%A8%E8%8C%83%E5%9B%B4-77)
* [信号](https://blog.csdn.net/zhangle1hao/article/details/78124910#84%E4%BF%A1%E5%8F%B7)
  + [意图](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%84%8F%E5%9B%BE-81)
  + [适用范围](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E9%80%82%E7%94%A8%E8%8C%83%E5%9B%B4-78)
  + [权威信息](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%9D%83%E5%A8%81%E4%BF%A1%E6%81%AF-63)
* [仆人](https://blog.csdn.net/zhangle1hao/article/details/78124910#85%E4%BB%86%E4%BA%BA)
  + [意图](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%84%8F%E5%9B%BE-82)
  + [适用范围](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E9%80%82%E7%94%A8%E8%8C%83%E5%9B%B4-79)
  + [权威信息](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%9D%83%E5%A8%81%E4%BF%A1%E6%81%AF-64)
* [服务层](https://blog.csdn.net/zhangle1hao/article/details/78124910#86%E6%9C%8D%E5%8A%A1%E5%B1%82)
  + [意图](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%84%8F%E5%9B%BE-83)
  + [适用范围](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E9%80%82%E7%94%A8%E8%8C%83%E5%9B%B4-80)
  + [权威信息](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%9D%83%E5%A8%81%E4%BF%A1%E6%81%AF-65)
* [服务定位](https://blog.csdn.net/zhangle1hao/article/details/78124910#87%E6%9C%8D%E5%8A%A1%E5%AE%9A%E4%BD%8D)
  + [意图](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%84%8F%E5%9B%BE-84)
  + [适用范围](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E9%80%82%E7%94%A8%E8%8C%83%E5%9B%B4-81)
  + [典型用例](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E5%85%B8%E5%9E%8B%E7%94%A8%E4%BE%8B-4)
  + [结论](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E7%BB%93%E8%AE%BA-2)
  + [权威信息](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%9D%83%E5%A8%81%E4%BF%A1%E6%81%AF-66)
* [单例](https://blog.csdn.net/zhangle1hao/article/details/78124910#89%E5%8D%95%E4%BE%8B)
  + [意图](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%84%8F%E5%9B%BE-85)
  + [解释](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E8%A7%A3%E9%87%8A-12)
  + [适用范围](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E9%80%82%E7%94%A8%E8%8C%83%E5%9B%B4-82)
  + [典型用例](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E5%85%B8%E5%9E%8B%E7%94%A8%E4%BE%8B-5)
  + [真实例子](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E7%9C%9F%E5%AE%9E%E4%BE%8B%E5%AD%90-26)
  + [结论](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E7%BB%93%E8%AE%BA-3)
  + [权威信息](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%9D%83%E5%A8%81%E4%BF%A1%E6%81%AF-67)
* [规范](https://blog.csdn.net/zhangle1hao/article/details/78124910#89%E8%A7%84%E8%8C%83)
  + [意图](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%84%8F%E5%9B%BE-86)
  + [适用范围](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E9%80%82%E7%94%A8%E8%8C%83%E5%9B%B4-83)
  + [关联模式](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E5%85%B3%E8%81%94%E6%A8%A1%E5%BC%8F-2)
  + [权威信息](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%9D%83%E5%A8%81%E4%BF%A1%E6%81%AF-68)
* [状态](https://blog.csdn.net/zhangle1hao/article/details/78124910#90%E7%8A%B6%E6%80%81)
  + [意图](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%84%8F%E5%9B%BE-87)
  + [适用范围](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E9%80%82%E7%94%A8%E8%8C%83%E5%9B%B4-84)
  + [真实例子](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E7%9C%9F%E5%AE%9E%E4%BE%8B%E5%AD%90-27)
  + [权威信息](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%9D%83%E5%A8%81%E4%BF%A1%E6%81%AF-69)
* [一步一步创建者](https://blog.csdn.net/zhangle1hao/article/details/78124910#91%E4%B8%80%E6%AD%A5%E4%B8%80%E6%AD%A5%E5%88%9B%E5%BB%BA%E8%80%85)
  + [意图](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%84%8F%E5%9B%BE-88)
  + [适用范围](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E9%80%82%E7%94%A8%E8%8C%83%E5%9B%B4-85)
  + [权威信息](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%9D%83%E5%A8%81%E4%BF%A1%E6%81%AF-70)
* [策略](https://blog.csdn.net/zhangle1hao/article/details/78124910#92%E7%AD%96%E7%95%A5)
  + [意图](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%84%8F%E5%9B%BE-89)
  + [适用范围](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E9%80%82%E7%94%A8%E8%8C%83%E5%9B%B4-86)
  + [权威信息](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%9D%83%E5%A8%81%E4%BF%A1%E6%81%AF-71)
* [模板方法](https://blog.csdn.net/zhangle1hao/article/details/78124910#93%E6%A8%A1%E6%9D%BF%E6%96%B9%E6%B3%95)
  + [意图](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%84%8F%E5%9B%BE-90)
  + [适用范围](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E9%80%82%E7%94%A8%E8%8C%83%E5%9B%B4-87)
  + [权威信息](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%9D%83%E5%A8%81%E4%BF%A1%E6%81%AF-72)
* [线程池](https://blog.csdn.net/zhangle1hao/article/details/78124910#94%E7%BA%BF%E7%A8%8B%E6%B1%A0)
  + [意图](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%84%8F%E5%9B%BE-91)
  + [适用范围](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E9%80%82%E7%94%A8%E8%8C%83%E5%9B%B4-88)
* [节流](https://blog.csdn.net/zhangle1hao/article/details/78124910#95%E8%8A%82%E6%B5%81)
  + [意图](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%84%8F%E5%9B%BE-92)
  + [适用范围](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E9%80%82%E7%94%A8%E8%8C%83%E5%9B%B4-89)
* [线程本地存储](https://blog.csdn.net/zhangle1hao/article/details/78124910#95%E7%BA%BF%E7%A8%8B%E6%9C%AC%E5%9C%B0%E5%AD%98%E5%82%A8)
  + [意图](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%84%8F%E5%9B%BE-93)
  + [适用范围](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E9%80%82%E7%94%A8%E8%8C%83%E5%9B%B4-90)
* [宽容阅读](https://blog.csdn.net/zhangle1hao/article/details/78124910#96%E5%AE%BD%E5%AE%B9%E9%98%85%E8%AF%BB)
  + [意图](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%84%8F%E5%9B%BE-94)
  + [适用范围](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E9%80%82%E7%94%A8%E8%8C%83%E5%9B%B4-91)
  + [权威信息](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%9D%83%E5%A8%81%E4%BF%A1%E6%81%AF-73)
* [双模式](https://blog.csdn.net/zhangle1hao/article/details/78124910#97-%E5%8F%8C%E6%A8%A1%E5%BC%8F)
  + [意图](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%84%8F%E5%9B%BE-95)
  + [适用范围](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E9%80%82%E7%94%A8%E8%8C%83%E5%9B%B4-92)
  + [权威信息](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%9D%83%E5%A8%81%E4%BF%A1%E6%81%AF-74)
* [值对象](https://blog.csdn.net/zhangle1hao/article/details/78124910#98-%E5%80%BC%E5%AF%B9%E8%B1%A1)
  + [意图](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%84%8F%E5%9B%BE-96)
  + [适用范围](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E9%80%82%E7%94%A8%E8%8C%83%E5%9B%B4-93)
  + [真实例子](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E7%9C%9F%E5%AE%9E%E4%BE%8B%E5%AD%90-28)
  + [权威信息](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%9D%83%E5%A8%81%E4%BF%A1%E6%81%AF-75)
* [访问者](https://blog.csdn.net/zhangle1hao/article/details/78124910#99-%E8%AE%BF%E9%97%AE%E8%80%85)
  + [意图](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%84%8F%E5%9B%BE-97)
  + [适用范围](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E9%80%82%E7%94%A8%E8%8C%83%E5%9B%B4-94)
  + [真实例子](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E7%9C%9F%E5%AE%9E%E4%BE%8B%E5%AD%90-29)
  + [权威信息](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E6%9D%83%E5%A8%81%E4%BF%A1%E6%81%AF-76)
  + [引用](https://blog.csdn.net/zhangle1hao/article/details/78124910#%E5%BC%95%E7%94%A8)

**1.抽象文档**

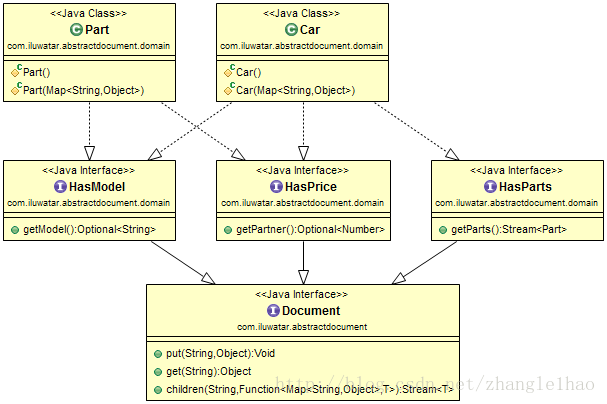
| **标题** | **类别** | **难度** |
| --- | --- | --- |
| abstract-document(抽象文档) | 结构型 | 中等 |

**意图**

灵活的实现类型安全，与语言无关。

**引用**[1]

所谓抽象就是：概念的聚合（Aggregate），可以很安全地忽略细节去使用一个概念，在不同的层次处理不同的细节。现实世界中很多东西已经是某种程度上的抽象了，举个例子：当你开门时，你直接去操纵门把手就行了，你不会去操心门具体用的什么木质材料、甚至分子层面，否则每天你开门时都将[寸步难行](https://www.baidu.com/s?wd=%E5%AF%B8%E6%AD%A5%E9%9A%BE%E8%A1%8C&tn=24004469_oem_dg&rsv_dl=gh_pl_sl_csd)。当你没能正确地抽象时，你的系统就像一扇过度复杂而无法打开的门，而好的开发者则会从门把手、门、房子等（对应软件开发中的方法、类、包等层次）各个层面创造出正确的抽象来降低复杂度。



**适用范围**

* 迫切的需要添加新属性
* 更灵活的处理树形结构
* 你想要更松散的耦合系统

**权威信息**

* [Wikipedia: Abstract Document Pattern](https://en.wikipedia.org/wiki/Abstract_Document_Pattern)
* [Martin Fowler: Dealing with properties](http://martinfowler.com/apsupp/properties.pdf)

**例子实现分析**

[Java 抽象文档设计模式例子分析](http://blog.csdn.net/zhangle1hao/article/details/78143480)

**2.抽象工厂**

| **标题** | **类别** | **难度** |
| --- | --- | --- |
| abstract-factory(抽象文档) | 创建型 | 中等 |

**意图**

提供一个接口，用于创建相关或从属的一类对象，而不指定它们的具体类。

**解释**

真实世界的例子

创建一个王国，我们需要对象的共同主题。精灵王国需要精灵王，精灵城堡和精灵的军队而兽人王国需要一个兽人王，兽人和兽人军队的城堡。在王国中对象间的依赖

字面意思

工厂的工厂；把个体和相关的、依赖的工厂组合在一起而不指定具体的工厂。

维基百科

抽象工厂模式提供了一种方法来封装一组具有共同主题的工厂，而不指定具体的类。

**编程例子**

翻译上面的王国例子。首先，我们为王国中的对象提供了一些接口和实现。

public interface Castle {

String getDescription();

}

public interface King {

String getDescription();

}

public interface Army {

String getDescription();

}

*// Elven implementations ->*

public class ElfCastle implements Castle {

static final String DESCRIPTION = "This is the Elven castle!";

@Override

public String getDescription() {

return DESCRIPTION;

}

}

public class ElfKing implements King {

static final String DESCRIPTION = "This is the Elven king!";

@Override

public String getDescription() {

return DESCRIPTION;

}

}

public class ElfArmy implements Army {

static final String DESCRIPTION = "This is the Elven Army!";

@Override

public String getDescription() {

return DESCRIPTION;

}

}

*// 兽人实现类似*

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35

然后我们王国工厂的抽象和实现

public interface KingdomFactory {

Castle createCastle();

King createKing();

Army createArmy();

}

public class ElfKingdomFactory implements KingdomFactory {

public Castle createCastle() {

return new ElfCastle();

}

public King createKing() {

return new ElfKing();

}

public Army createArmy() {

return new ElfArmy();

}

}

public class OrcKingdomFactory implements KingdomFactory {

public Castle createCastle() {

return new OrcCastle();

}

public King createKing() {

return new OrcKing();

}

public Army createArmy() {

return new OrcArmy();

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29

现在我们有了一个抽象工厂，让我们制造相关的家族，即精灵王国工厂制造精灵城堡、国王和军队等。

KingdomFactory factory = new ElfKingdomFactory()*;*

Castle castle = factory.createCastle()*;*

King king = factory.createKing()*;*

Army army = factory.createArmy()*;*

castle.getDescription()*; // Output: This is the Elven castle!*

king.getDescription()*; // Output: This is the Elven king!*

army.getDescription()*; // Output: This is the Elven Army!*

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8

**适用范围**

* 一个系统应该独立于它的产品是如何创建、组合和表示。
* 一个系统应配置一个产品的多种家族
* 一系列相关产品对象被设计为一起使用，您需要强制执行此约束。
* 您希望提供一个产品类库，并且希望只显示它们的接口，而不是它们的实现。
* 依赖的生命周期短于消费的一方。
* 你需要一个运行时的值来构造一个特殊的依赖
* 你要决定哪些产品在运行时调用哪种家族.
* 在解析依赖项之前，您需要提供一个或多个参数，这些参数只在运行时已知。

**用例:**

* 选择要在运行时调用文件服务、数据库服务或网络服务适当的实现.
* 更好的实现单元测试

**结论:**

* 依赖注入可以让java运行时报错的服务提早在编译时被抓住.

**现实世界的例子**

* [javax.xml.parsers.DocumentBuilderFactory](http://docs.oracle.com/javase/8/docs/api/javax/xml/parsers/DocumentBuilderFactory.html)
* [javax.xml.transform.TransformerFactory](http://docs.oracle.com/javase/8/docs/api/javax/xml/transform/TransformerFactory.html#newInstance--)
* [javax.xml.xpath.XPathFactory](http://docs.oracle.com/javase/8/docs/api/javax/xml/xpath/XPathFactory.html#newInstance--)

**权威信息**

* [Design Patterns: Elements of Reusable Object-Oriented Software](http://www.amazon.com/Design-Patterns-Elements-Reusable-Object-Oriented/dp/0201633612)

**3.适配器模式**

| **标题** | **类别** | **难度** |
| --- | --- | --- |
| adapter(适配器) | 结构型 | GOF,中等 |

**意图**

将一个类的接口转换成客户端期待的接口。适配器让彼此之间接口不兼容的类工作。

**解释**

真实例子

你想把内存卡中的图片传输到自己的电脑上. 为了传输它们，你需要一种与你的计算机端口兼容的适配器，这样你就可以把内存卡附加到你的计算机上。在这种情况下，读卡器是一个适配器。

另一个例子是著名的电源适配器；一个三条腿的插头不能连接到一个两个孔的插座，它需要使用电源适配器，使两个孔的插座兼容。

另一个例子是译者将一个人的话翻译给另一个人。

字面意思

适配器模式允许您在适配器中包装一个不兼容的对象，使其与另一个类兼容。

维基描述

在软件工程中，适配器模式是一种软件设计模式，它允许现有类的接口作为另一个接口使用。它通常用于使现有类与其他类协同工作而不修改源代码。

**编程例子**

考虑一个只能使用划艇的船长，根本不能使用帆船。   
首先我们有接口划艇和帆船

public interface RowingBoat {

void row();

}

public class FishingBoat {

private static final Logger LOGGER = LoggerFactory.getLogger(FishingBoat.class);

public void sail() {

LOGGER.info("The fishing boat is sailing");

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10

并且船长希望实现“RowingBoat”接口能够行驶船

public class Captain implements RowingBoat {

private RowingBoat rowingBoat;

public Captain(RowingBoat rowingBoat) {

this.rowingBoat = rowingBoat;

}

@Override

public void row() {

rowingBoat.row();

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13

现在让我们说海盗来了，船长需要逃跑，但只有渔船可用。我们需要创建一个适配器，让船长用他的划船技巧操作渔船。

public class FishingBoatAdapter implements RowingBoat {

private static final Logger LOGGER = LoggerFactory.getLogger(FishingBoatAdapter.class);

private FishingBoat boat;

public FishingBoatAdapter() {

boat = new FishingBoat();

}

@Override

public void row() {

boat.sail();

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15

现在船长可以用渔船逃离海盗。

Captain captain = new Captain(new FishingBoatAdapter());

captain.row();

* 1
* 2

**博主思考**   
比如说一个厨师,他拥有[各种各样](https://www.baidu.com/s?wd=%E5%90%84%E7%A7%8D%E5%90%84%E6%A0%B7&tn=24004469_oem_dg&rsv_dl=gh_pl_sl_csd)的专业知识，比如烤面包，但是他并不需要知道如何从原材料中获取食材，他只需要知道如何使用食材。适配器很多时候就像提供资源一样，提供面粉，   
牛奶，鸡蛋，黄油，酱油…也就是适配器跟原材料到可使用的食材打交道，生成厨师需要的食材。好处说白了就是，厨师(领域知识)只需要专精他如何制作更美味更好的食物。而我们(客户端程序员)只需要提供厨师需要的资源就行了 :)   


**使用范围**

* 你想使用一个现有的类，它的接口与你所需要的不匹配
* 你想创建一个可重用的类，与不相关或不可预见的类进行协作，也就是说，不一定具有兼容接口的类
* 您需要使用几个现有的子类，但通过对每个子类进行子类化来调整其接口是不切实际的。 对象适配器可以调整其父类的接口
* 大多数使用第三方库的应用程序使用适配器作为应用程序和第三方库之间的中间层，以将应用程序与库分离。 如果必须使用另一个库，则只需要新库的适配器，而无需更改应用程序代码。

**结论:**

类和对象适配器有不同的取舍。一个类适配器

* 当致力于去适配一个具体的Adaptee （受改造者）类到目标的类。将导致类适配器不能适配一个类和他所有的子类。
* 让适配器重写Adaptee（受改造者）的一些行为，因为适配器是Adaptee(受改造者)的子类。
* 仅引入一个对象到Adaptee（受改造者）中，并不需要使用适配器。

对象适配器

* 让我们一个适配器与许多Adaptee工作，即Adaptee本身及其所有子类（如果有的话）。 适配器还可以一次性向所有Adaptees添加功能。
* 很难覆盖Adaptee（受改造者）的行为。这将需要Adaptee（受改造者）的子类和适配器相互作用而不是Adaptee（受改造者）本身

**真实例子**

* [java.util.Arrays#asList()](http://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html#asList%28T...%29)
* [java.util.Collections#list()](https://docs.oracle.com/javase/8/docs/api/java/util/Collections.html#list-java.util.Enumeration-)
* [java.util.Collections#enumeration()](https://docs.oracle.com/javase/8/docs/api/java/util/Collections.html#enumeration-java.util.Collection-)
* [javax.xml.bind.annotation.adapters.XMLAdapter](http://docs.oracle.com/javase/8/docs/api/javax/xml/bind/annotation/adapters/XmlAdapter.html#marshal-BoundType-)

**权威信息**

* [Design Patterns: Elements of Reusable Object-Oriented Software](http://www.amazon.com/Design-Patterns-Elements-Reusable-Object-Oriented/dp/0201633612)
* [J2EE Design Patterns](http://www.amazon.com/J2EE-Design-Patterns-William-Crawford/dp/0596004273/ref=sr_1_2)

**4.聚合器-微服务设计模式**

| **标题** | **类别** | **难度** |
| --- | --- | --- |
| Aggregator Microservices(聚合器-微服务) | 结构型 | Java,Spring |

**Intent**

用户对Aggregator进行单次调用，然后聚合器调用每个相关的微服务并进行收集   
数据，将业务逻辑应用于它，并进一步发布为REST端点。   
聚合器的更多变体：   
- 代理微服务设计模式：根据业务需求调用不同的微服务。   
- 链接的微服务设计模式：在这种情况下，每个微服务依赖/链接到一系列   
的其他微服务。

**适用范围**

当您需要统一的API时，无论客户端设备有多少种，使用Aggregator Microservices模式。

**权威信息**

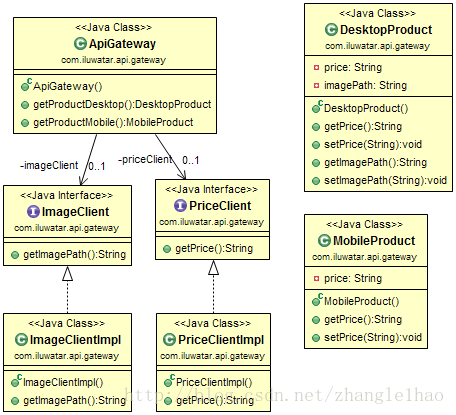
* [Microservice Design Patterns](http://blog.arungupta.me/microservice-design-patterns/)

**5.api-网关**

| **标题** | **类别** | **难度** |
| --- | --- | --- |
| api-网关 | 结构型 | Java,Spring，中等难度 |

**意图**

聚集微服务调用在一个位置: API 网关. 用户只调用api网关, Api网关调用其他微服务.



**适用范围**

当使用API网关模式时

* 你也需要使用微服务模式,并将它们聚集到一个点，方便调用它们。

**权威信息**

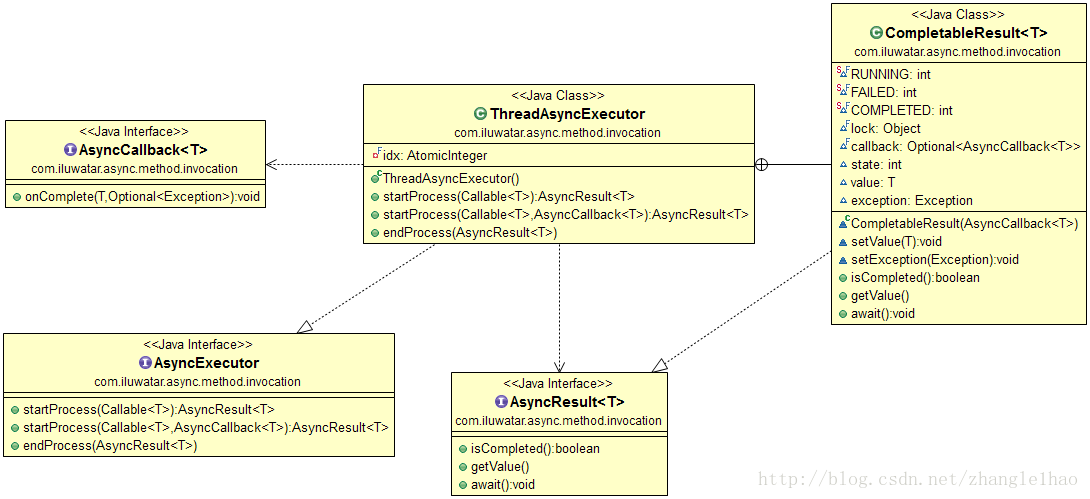
* [microservices.io - API Gateway](http://microservices.io/patterns/apigateway.html)
* [NGINX - Building Microservices: Using an API Gateway](https://www.nginx.com/blog/building-microservices-using-an-api-gateway/)

**6.异步方法调用**

| **标题** | **类别** | **难度** |
| --- | --- | --- |
| 异步方法调用 | 并发 | Java，中等难度,函数式，响应式 |

**意图**

异步方法调用设计模式的作用是等待任务结果时不被阻塞。这种设模式对多个独立的任务提供并行处理，并且提供回调（callback）或等待阻塞当前线程(await)拿到结果值。



**适用范围**

如下情况使用异步调用模式:

* 您有多个可以并行运行的独立任务
* 您需要提高一组顺序任务的性能
* 您的处理能力有限制或有一个长时间运行的任务,等待结果的一方（caller）不应该长时间等待

**真实例子**

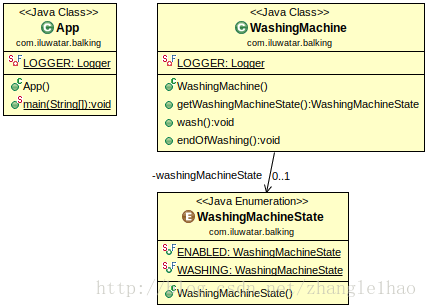
* [FutureTask](http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/FutureTask.html), [CompletableFuture](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html) and [ExecutorService](http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ExecutorService.html) (Java)
* [Task-based Asynchronous Pattern](https://msdn.microsoft.com/en-us/library/hh873175.aspx) (.NET)

**7.阻塞模式**

| **标题** | **类别** | **难度** |
| --- | --- | --- |
| 阻塞模式（Balking Pattern） | 并发 | Java，中等难度 |

**意图**

阻塞模式被用于阻止某个对象执行某些不完全或不适当状态的代码



**适用范围**

如下情况使用阻塞模式:

* 只有当某个对象处于特定状态时，才能够调用对象的操作
* 对象通常处于暂时瘫痪的状态在一段未知的时间内。

**相关模式**

* Guarded Suspension Pattern
* Double Checked Locking Pattern(双重锁检查单例模式)

**8.桥接模式**

| **标题** | **类别** | **难度** |
| --- | --- | --- |
| 桥接模式（Bridge Pattern） | 结构型 | Java，GOF,中等难度 |

**意图**

将抽象（abstraction）与它的实现(implementation)解耦，使两者可以独立变化。

**解释**

真实例子

考虑你有一个具有魔法的武器，你应该允许不同的武器拥有不同的魔法。你会怎么做？ 创建多个武器副本赋予每个武器副本魔法或者创建单独魔法类并将其设置到武器类，桥接模式允许你实现第二种方式

字面意义

桥接模式更喜欢组合相对于继承. 实现细节从一个层次结构转移到另一个具有单独层次结构的对象

维基百科

桥接模式是一种用于软件工程的设计模式，其目的是“将抽象与实现分离，从而使两者独立变化”。

**编程例子**

从上面翻译我们的武器示例。这里我们有Weapon层

public interface Weapon {

void wield();

void swing();

void unwield();

Enchantment getEnchantment();

}

public class Sword implements Weapon {

private final Enchantment enchantment;

public Sword(Enchantment enchantment) {

this.enchantment = enchantment;

}

@Override

public void wield() {

LOGGER.info("The sword is wielded.");

enchantment.onActivate();

}

@Override

public void swing() {

LOGGER.info("The sword is swinged.");

enchantment.apply();

}

@Override

public void unwield() {

LOGGER.info("The sword is unwielded.");

enchantment.onDeactivate();

}

@Override

public Enchantment getEnchantment() {

return enchantment;

}

}

public class Hammer implements Weapon {

private final Enchantment enchantment;

public Hammer(Enchantment enchantment) {

this.enchantment = enchantment;

}

@Override

public void wield() {

LOGGER.info("The hammer is wielded.");

enchantment.onActivate();

}

@Override

public void swing() {

LOGGER.info("The hammer is swinged.");

enchantment.apply();

}

@Override

public void unwield() {

LOGGER.info("The hammer is unwielded.");

enchantment.onDeactivate();

}

@Override

public Enchantment getEnchantment() {

return enchantment;

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36
* 37
* 38
* 39
* 40
* 41
* 42
* 43
* 44
* 45
* 46
* 47
* 48
* 49
* 50
* 51
* 52
* 53
* 54
* 55
* 56
* 57
* 58
* 59
* 60
* 61
* 62
* 63
* 64
* 65
* 66
* 67
* 68
* 69
* 70

分离的施法层

public interface Enchantment {

void onActivate();

void apply();

void onDeactivate();

}

public class FlyingEnchantment implements Enchantment {

@Override

public void onActivate() {

LOGGER.info("The item begins to glow faintly.");

}

@Override

public void apply() {

LOGGER.info("The item flies and strikes the enemies finally returning to owner's hand.");

}

@Override

public void onDeactivate() {

LOGGER.info("The item's glow fades.");

}

}

public class SoulEatingEnchantment implements Enchantment {

@Override

public void onActivate() {

LOGGER.info("The item spreads bloodlust.");

}

@Override

public void apply() {

LOGGER.info("The item eats the soul of enemies.");

}

@Override

public void onDeactivate() {

LOGGER.info("Bloodlust slowly disappears.");

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36
* 37
* 38
* 39
* 40
* 41

action在两个层次之间

Sword enchantedSword = new Sword(new SoulEatingEnchantment());

enchantedSword.wield();

enchantedSword.swing();

enchantedSword.unwield();

*// 剑被拿起.*

*// 开始杀戮.*

*// 剑被使用.*

*// 这个魔法吞噬敌人的灵魂。.*

*// 剑被放下.*

*// 嗜血慢慢消失.*

Hammer hammer = new Hammer(new FlyingEnchantment());

hammer.wield();

hammer.swing();

hammer.unwield();

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16

**适用范围**

使用Bridge模式

* 你想避免抽象和它的实现之间的永久绑定。可能是这种情况，例如，必须在运行时选择或切换实现。
* 抽象和它们的实现都应该通过子类化来扩展。在这种情况下，Bridge模式可以让您组合不同的抽象和实现，并独立扩展它们
* 抽象实现的变化对客户不应有影响;也就是说，他们的代码不应该被重新编译。
* 你有一个基类有太多子类。这样的类层次结构表示需要将对象分成两部分。 Rumbaugh使用术语“嵌套泛化”来引用这样的类层次结构
* 您希望在多个对象之间共享一个实现（可能使用引用计数），并且该事实应该从客户端隐藏。一个简单的例子是Coplien的String类，其中多个对象可以共享相同的字符串表示。

**权威信息**

* [Design Patterns: Elements of Reusable Object-Oriented Software](http://www.amazon.com/Design-Patterns-Elements-Reusable-Object-Oriented/dp/0201633612)

**9.构建模式**

| **标题** | **类别** | **难度** |
| --- | --- | --- |
| 构建模式（Builder Pattern） | 构建型 | Java，GOF,中等难度 |

**意图**

将复杂对象的构造与它的表现分离，使相同的构建过程可以创建不同的表现。

**解释**

真实例子

想象一个角色扮演游戏的角色生成器。最简单的选择是让计算机为你创建角色。但是，如果你想选择诸如职业，性别，头发颜色等字符的细节，角色生成器成为一个[循序渐进](https://www.baidu.com/s?wd=%E5%BE%AA%E5%BA%8F%E6%B8%90%E8%BF%9B&tn=24004469_oem_dg&rsv_dl=gh_pl_sl_csd)的过程，完成时，所有的选择都准备好了.

字面意思

允许您创建不同风格的对象，同时避免构造器污染。 当有可能创建几种不同的对象时很有用。 或者当创建对象时涉及很多步骤。

维基百科

构建器模式是一种对象创建软件设计模式，其目的是找到一种方案，解决伸缩构造函数反模式。

话虽如此，我还是要补充一点，关于伸缩构造函数反模式是什么。在某一点上，我们都看到了如下构造函数：

public Hero(Profession profession, String name, HairType hairType, HairColor hairColor, Armor armor, Weapon weapon) {

}

* 1
* 2

您可以看到，构造函数的数量可能会很快失控，并且可能难以理解参数的排列。 如果您以后要添加更多选项，此参数列表可能会不断增长。 这被称为伸缩构造器反模式。

**编程例子**

理想的选择是使用Builder模式。 首先我们有我们要创造的英雄

public final class Hero {

private final Profession profession;

private final String name;

private final HairType hairType;

private final HairColor hairColor;

private final Armor armor;

private final Weapon weapon;

private Hero(Builder builder) {

this.profession = builder.profession;

this.name = builder.name;

this.hairColor = builder.hairColor;

this.hairType = builder.hairType;

this.weapon = builder.weapon;

this.armor = builder.armor;

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17

And then we have the builder

public static class Builder {

private final Profession profession;

private final String name;

private HairType hairType;

private HairColor hairColor;

private Armor armor;

private Weapon weapon;

public Builder(Profession profession, String name) {

if (profession == null || name == null) {

throw new IllegalArgumentException("profession and name can not be null");

}

this.profession = profession;

this.name = name;

}

public Builder withHairType(HairType hairType) {

this.hairType = hairType;

return this;

}

public Builder withHairColor(HairColor hairColor) {

this.hairColor = hairColor;

return this;

}

public Builder withArmor(Armor armor) {

this.armor = armor;

return this;

}

public Builder withWeapon(Weapon weapon) {

this.weapon = weapon;

return this;

}

public Hero build() {

return new Hero(this);

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36
* 37
* 38
* 39
* 40

然后它可以用作：

Hero mage = new Hero.Builder(Profession.MAGE, "Riobard").withHairColor(HairColor.BLACK).withWeapon(Weapon.DAGGER).build()*;*

* 1

**适用范围**

使用Builder模式

* 用于创建复杂对象的算法应该独立于组成对象的部分以及它们是如何被组装
* 构造过程必须允许构建的对象的不同表现

**真实例子**

* [java.lang.StringBuilder](http://docs.oracle.com/javase/8/docs/api/java/lang/StringBuilder.html)
* [java.nio.ByteBuffer](http://docs.oracle.com/javase/8/docs/api/java/nio/ByteBuffer.html#put-byte-) as well as similar buffers such as FloatBuffer, IntBuffer and so on.
* [java.lang.StringBuffer](http://docs.oracle.com/javase/8/docs/api/java/lang/StringBuffer.html#append-boolean-)
* All implementations of [java.lang.Appendable](http://docs.oracle.com/javase/8/docs/api/java/lang/Appendable.html)
* [Apache Camel builders](https://github.com/apache/camel/tree/0e195428ee04531be27a0b659005e3aa8d159d23/camel-core/src/main/java/org/apache/camel/builder)

**权威信息**

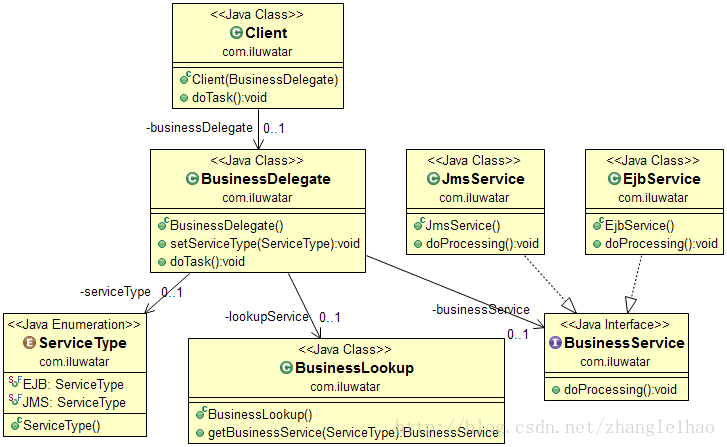
* [Design Patterns: Elements of Reusable Object-Oriented Software](http://www.amazon.com/Design-Patterns-Elements-Reusable-Object-Oriented/dp/0201633612)
* [Effective Java (2nd Edition)](http://www.amazon.com/Effective-Java-Edition-Joshua-Bloch/dp/0321356683)

**11.业务委托**

| **标题** | **类别** | **难度** |
| --- | --- | --- |
| 业务委托（Business Delegate） | 业务层 | Java,中等难度 |

**意图**

业务委托模式在表示层和业务层之间添加一个抽象层。 通过使用这种设计模式，   
我们获得了层之间的松散耦合,   
介于层次之间并封装有关如何定位，连接到，   
并与构成应用程序的业务对象进行交互。



**适用范围**

何时使用业务委托模式

* 您希望表示和业务层之间松散耦合
* 您想协调多个业务服务的呼叫
* 你想封装服务查询和服务调用

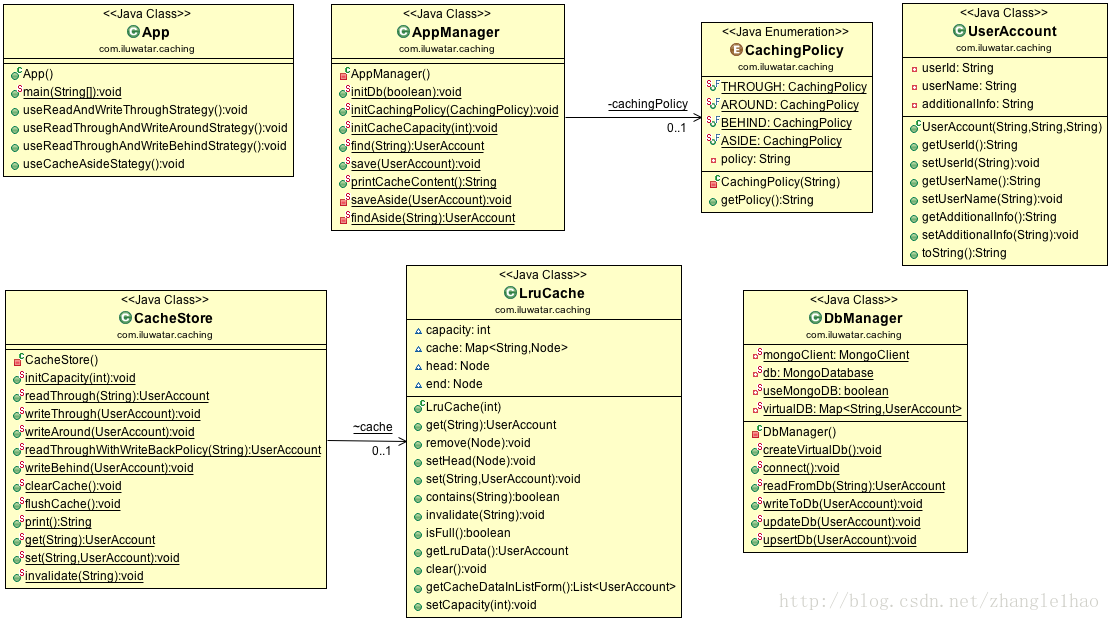
**权威信息**

* [J2EE Design Patterns](http://www.amazon.com/J2EE-Design-Patterns-William-Crawford/dp/0596004273/ref=sr_1_2)

**12.缓存模式**

| **标题** | **类别** | **难度** |
| --- | --- | --- |
| 缓存模式（caching design） | 其他 | Java,中等难度,性能相关 |

**意图**

通过不立即释放资源的方式避免昂贵的资源再获取操作。 资源保留他们的身份，保留在一些快速访问存储，并重新使用，以避免再次获取它们。   


**适用范围**

何时使用缓存模式

* 重复的资源获取，初始化和释放会导致不必要的性能开销。

**权威信息**

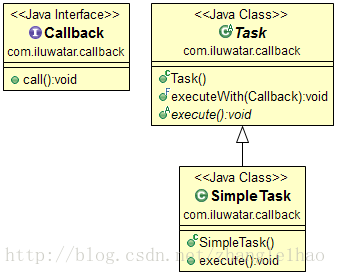
* [Write-through, write-around, write-back: Cache explained](http://www.computerweekly.com/feature/Write-through-write-around-write-back-Cache-explained)
* [Read-Through, Write-Through, Write-Behind, and Refresh-Ahead Caching](https://docs.oracle.com/cd/E15357_01/coh.360/e15723/cache_rtwtwbra.htm#COHDG5177)
* [Cache-Aside](https://msdn.microsoft.com/en-us/library/dn589799.aspx)

**13.回调模式**

| **标题** | **类别** | **难度** |
| --- | --- | --- |
| 回调模式（callback） | 其他 | Java,简单难度,函数式编程 |

**意图**

回调是作为一段可传递的执行代码,是其他代码的参数。在合适的时间点执行。



**适用范围**

何时使用回调模式时

* 在执行某些已定义的活动之后，必须执行同步或异步操作。

**真实例子**

* [CyclicBarrier](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/CyclicBarrier.html#CyclicBarrier%28int,%20java.lang.Runnable%29) constructor can accept callback that will be triggered every time when barrier is tripped.

**14.责任链模式**

| **标题** | **类别** | **难度** |
| --- | --- | --- |
| 责任链模式（chain design） | 行为型 | Java,GOF,中等难度 |

**意图**

通过给与一个或更多对象处理请求的机会避免将请求的发送者与接收者耦合。责任链接收   
对象，并沿着链传递请求，直到有对象处理它为止。

**解释**

真实例子

兽王向他的军队发出明确的命令。 最接近的人是指挥官，然后是军官，然后是士兵。 这里的指挥官，军官和士兵组成了一连串的责任。

字面意思

它有助于建立一个对象链。 请求从一端进入，并保持前进从对象到对象，直到找到合适的处理程序。

维基描述

在面向对象设计中，责任链模式是由命令对象源和一系列处理对象组成的设计模式。 每个处理对象包含定义可以处理的命令对象的类型的逻辑; 其余的传递给链中的下一个处理对象。

**编程例子**

用上面的兽人转换成我们的例子。 首先我们有Request类

public class Request {

private final RequestType requestType;

private final String requestDescription;

private boolean handled;

public Request(final RequestType requestType, final String requestDescription) {

this.requestType = Objects.requireNonNull(requestType);

this.requestDescription = Objects.requireNonNull(requestDescription);

}

public String getRequestDescription() { return requestDescription; }

public RequestType getRequestType() { return requestType; }

public void markHandled() { this.handled = true; }

public boolean isHandled() { return this.handled; }

@Override

public String toString() { return getRequestDescription(); }

}

public enum RequestType {

DEFEND\_CASTLE, TORTURE\_PRISONER, COLLECT\_TAX

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26

然后RequestHandler的层次结构

public abstract class RequestHandler {

private static final Logger LOGGER = LoggerFactory.getLogger(RequestHandler.class);

private RequestHandler next;

public RequestHandler(RequestHandler next) {

this.next = next;

}

public void handleRequest(Request req) {

if (next != null) {

next.handleRequest(req);

}

}

protected void printHandling(Request req) {

LOGGER.info("{} handling request \"{}\"", this, req);

}

@Override

public abstract String toString();

}

public class OrcCommander extends RequestHandler {

public OrcCommander(RequestHandler handler) {

super(handler);

}

@Override

public void handleRequest(Request req) {

if (req.getRequestType().equals(RequestType.DEFEND\_CASTLE)) {

printHandling(req);

req.markHandled();

} else {

super.handleRequest(req);

}

}

@Override

public String toString() {

return "兽人指挥官";

}

}

*// OrcOfficer and OrcSoldier are defined similarly as OrcCommander*

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36
* 37
* 38
* 39
* 40
* 41
* 42
* 43
* 44
* 45

然后我们有兽人国王，他命令并形成了链

public class OrcKing {

RequestHandler chain;

public OrcKing() {

buildChain();

}

private void buildChain() {

chain = new OrcCommander(new OrcOfficer(new OrcSoldier(null)));

}

public void makeRequest(Request req) {

chain.handleRequest(req);

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15

然后使用如下

OrcKing king = new OrcKing();

king.makeRequest(new Request(RequestType.DEFEND\_CASTLE, "defend castle")); *// Orc commander handling request "defend castle"*

king.makeRequest(new Request(RequestType.TORTURE\_PRISONER, "torture prisoner")); *// Orc officer handling request "torture prisoner"*

king.makeRequest(new Request(RequestType.COLLECT\_TAX, "collect tax")); *// Orc soldier handling request "collect tax"*

* 1
* 2
* 3
* 4

**适用范围**

何时使用责任链

* 多个对象可以处理请求，并且处理程序不是先验知道的。 处理程序应自动确定。
* 您要向几个对象之一发出请求，而不明确指定接收者。
* 可以动态指定可处理请求的对象集。

**真实例子**

* [java.util.logging.Logger#log()](http://docs.oracle.com/javase/8/docs/api/java/util/logging/Logger.html#log%28java.util.logging.Level,%20java.lang.String%29)
* [Apache Commons Chain](https://commons.apache.org/proper/commons-chain/index.html)
* [javax.servlet.Filter#doFilter()](http://docs.oracle.com/javaee/7/api/javax/servlet/Filter.html#doFilter-javax.servlet.ServletRequest-javax.servlet.ServletResponse-javax.servlet.FilterChain-)

**权威信息**

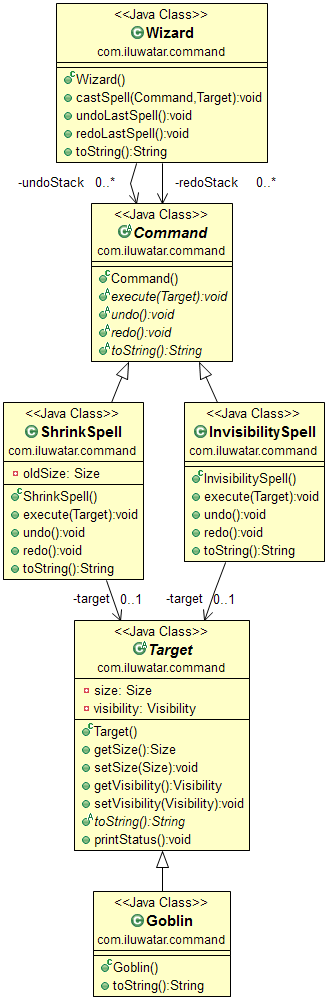
* [Design Patterns: Elements of Reusable Object-Oriented Software](http://www.amazon.com/Design-Patterns-Elements-Reusable-Object-Oriented/dp/0201633612)

**15.命令模式**

| **标题** | **类别** | **难度** |
| --- | --- | --- |
| 命令模式（command design） | 行为型 | Java,GOF,中等难度,函数式编程 |

**意图**

将请求封装为一个对象，从而使你用不同的请求对客户进行参数化，对请求排队或记录请求日志，以及   
支持可撤销的操作。



**适用范围**

当你想要使用命令模式

* 通过执行操作来参数化对象。您可以用具有回调函数的过程语言表达这种参数化，即在稍后注册的地方注册的函数。命令是一个面向对象的方式替换回调。
* 在不同时间指定，队列和执行请求。 Command对象可以具有与原始请求无关的生命周期。如果请求的接收方可以以地址空间无关的方式表示，那么您可以将请求的命令对象传送到不同的进程，并在此处执行请求。
* 支持撤消。命令的执行操作可以存储在命令本身中将其效果反转的状态。 Command界面必须有一个添加的“非执行”操作，可以将先前调用的效果反转为执行。执行的命令存储在历史记录列表中。无限级别的撤消和重做是通过遍历该列表来分别来回执行并且执行的
* 支持记录更改，以便在系统崩溃的情况下重新应用它们。通过使用加载和存储操作增加Command界面，可以保留持久性的更改日志。从崩溃中恢复包括从磁盘重新加载记录的命令，并使用执行操作重新执行它们
* 构建基于原始操作的高级操作的系统。 这种结构在支持交易的信息系统中是常见的。 事务封装了一组数据更改。 Command模式提供了一种模型交易的方式。 命令有一个通用的界面，让你以相同的方式调用所有的事务。 该模式还可以通过新的事务来扩展系统

**典型用例**

* 保留历史请求
* 实现回调功能
* 实现撤消功能

**真实例子**

* [java.lang.Runnable](http://docs.oracle.com/javase/8/docs/api/java/lang/Runnable.html)
* [Netflix Hystrix](https://github.com/Netflix/Hystrix/wiki)
* [javax.swing.Action](http://docs.oracle.com/javase/8/docs/api/javax/swing/Action.html)

**权威信息**

* [Design Patterns: Elements of Reusable Object-Oriented Software](http://www.amazon.com/Design-Patterns-Elements-Reusable-Object-Oriented/dp/0201633612)

**16.组合模式**

| **标题** | **类别** | **难度** |
| --- | --- | --- |
| 组合模式（Composite design） | 结构型 | Java,GOF,中等难度 |

**意图**

将对象组合成树结构以表示部分整体层次结构。 组合允许客户端处理单个对象和组合的对象一致。

**解释**

真实例子

每个句子都是由字符组成的单词组成。 这些对象中的每一个都是可打印的，它们可以在它们之前或之后打印出一些东西，因为句子总是以完全停止的方式结束，而且在它之前总是有空格

字面意思

组合模式使客户能够以统一的方式对待各个对象。

维基百科

在软件工程中，组合模式是分区设计模式。 组合模式描述了一组对象的处理方式与对象的单个实例相同。 组合的意图是将对象“组合”成树结构以表示部分整体层次结构。 实现组合模式使客户能够一致地处理单个对象和对象组。

**编程例子**

以我们上面的例子为例。在这里，我们有不同类型的基类，可打印的字符

public abstract class LetterComposite {

private List<LetterComposite> children = new ArrayList<>();

public void add(LetterComposite letter) {

children.add(letter);

}

public int count() {

return children.size();

}

protected void printThisBefore() {}

protected void printThisAfter() {}

public void print() {

printThisBefore();

for (LetterComposite letter : children) {

letter.print();

}

printThisAfter();

}

}

public class Letter extends LetterComposite {

private char c;

public Letter(char c) {

this.c = c;

}

@Override

protected void printThisBefore() {

System.out.print(c);

}

}

public class Word extends LetterComposite {

public Word(List<Letter> letters) {

for (Letter l : letters) {

this.add(l);

}

}

@Override

protected void printThisBefore() {

System.out.print(" ");

}

}

public class Sentence extends LetterComposite {

public Sentence(List<Word> words) {

for (Word w : words) {

this.add(w);

}

}

@Override

protected void printThisAfter() {

System.out.print(".");

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36
* 37
* 38
* 39
* 40
* 41
* 42
* 43
* 44
* 45
* 46
* 47
* 48
* 49
* 50
* 51
* 52
* 53

Then we have a messenger to carry messages

public class Messenger {

LetterComposite messageFromOrcs() {

List<Word> words = new ArrayList<>();

words.add(new Word(Arrays.asList(new Letter('W'), new Letter('h'), new Letter('e'), new Letter('r'), new Letter('e'))));

words.add(new Word(Arrays.asList(new Letter('t'), new Letter('h'), new Letter('e'), new Letter('r'), new Letter('e'))));

words.add(new Word(Arrays.asList(new Letter('i'), new Letter('s'))));

words.add(new Word(Arrays.asList(new Letter('a'))));

words.add(new Word(Arrays.asList(new Letter('w'), new Letter('h'), new Letter('i'), new Letter('p'))));

words.add(new Word(Arrays.asList(new Letter('t'), new Letter('h'), new Letter('e'), new Letter('r'), new Letter('e'))));

words.add(new Word(Arrays.asList(new Letter('i'), new Letter('s'))));

words.add(new Word(Arrays.asList(new Letter('a'))));

words.add(new Word(Arrays.asList(new Letter('w'), new Letter('a'), new Letter('y'))));

return new Sentence(words);

}

LetterComposite messageFromElves() {

List<Word> words = new ArrayList<>();

words.add(new Word(Arrays.asList(new Letter('M'), new Letter('u'), new Letter('c'), new Letter('h'))));

words.add(new Word(Arrays.asList(new Letter('w'), new Letter('i'), new Letter('n'), new Letter('d'))));

words.add(new Word(Arrays.asList(new Letter('p'), new Letter('o'), new Letter('u'), new Letter('r'), new Letter('s'))));

words.add(new Word(Arrays.asList(new Letter('f'), new Letter('r'), new Letter('o'), new Letter('m'))));

words.add(new Word(Arrays.asList(new Letter('y'), new Letter('o'), new Letter('u'), new Letter('r'))));

words.add(new Word(Arrays.asList(new Letter('m'), new Letter('o'), new Letter('u'), new Letter('t'), new Letter('h'))));

return new Sentence(words);

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26

然后它可以用作

LetterComposite orcMessage = new Messenger().messageFromOrcs();

orcMessage.print(); *// Where there is a whip there is a way.*

LetterComposite elfMessage = new Messenger().messageFromElves();

elfMessage.print(); *// Much wind pours from your mouth.*

* 1
* 2
* 3
* 4

**适用范围**

使用组合模式时

* 您想要表示对象的部分-整体层次结构
* 您希望客户能够忽略对象整体和单个对象的组合之间的差异。 客户将统一处理复合结构中的所有对象

**真实例子**

* [java.awt.Container](http://docs.oracle.com/javase/8/docs/api/java/awt/Container.html) and [java.awt.Component](http://docs.oracle.com/javase/8/docs/api/java/awt/Component.html)
* [Apache Wicket](https://github.com/apache/wicket) component tree, see [Component](https://github.com/apache/wicket/blob/91e154702ab1ff3481ef6cbb04c6044814b7e130/wicket-core/src/main/java/org/apache/wicket/Component.java) and [MarkupContainer](https://github.com/apache/wicket/blob/b60ec64d0b50a611a9549809c9ab216f0ffa3ae3/wicket-core/src/main/java/org/apache/wicket/MarkupContainer.java)

**权威信息**

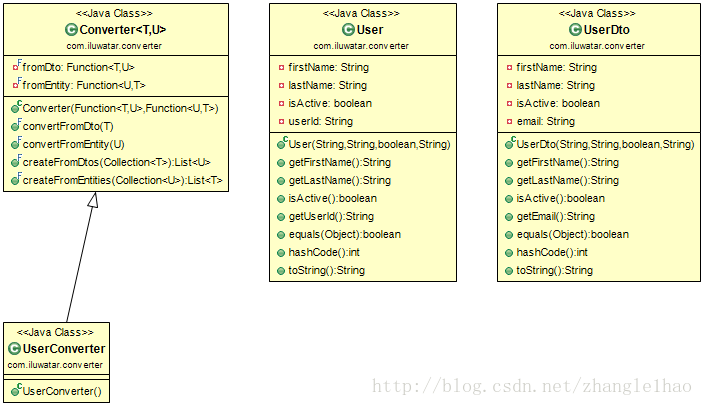
* [Design Patterns: Elements of Reusable Object-Oriented Software](http://www.amazon.com/Design-Patterns-Elements-Reusable-Object-Oriented/dp/0201633612)

**17.转换器模式**

| **标题** | **类别** | **难度** |
| --- | --- | --- |
| 转换器（convert） |  | 简单难度 |

**意图**

转换器模式的目的是提供一种通用的，在相应类型之间进行双向转换，类型转换不需要了解彼此，是一种干净的转换方式。 此外，转换器模式引入了双向收集映射，将样板代码减少到最小。



**适用范围**

在以下情况下使用转换器模式：

* 当您有类型逻辑对应哪个其他，您需要在它们之间转换实体
* 当您想根据上下文提供不同的类型转换方式时
* 每当您引入DTO（数据传输对象）时，您可能需要将其转换为具体领域对象

**权威信息**

* [Converter](http://www.xsolve.pl/blog/converter-pattern-in-java-8/)

**18.命令查询的责任分离模式**

| **标题** | **类别** | **难度** |
| --- | --- | --- |
| 命令查询的责任分离（CQRS design） | 结构型 | 中等难度 |

**意图**

CQRS命令查询责任分离 - 将查询与命令端分开(博主理解:有点像数据库的读写分离，但对写一端有了历史记录变化的追溯,) 。   


**适用范围**

使用CQRS模式

* 您要独立扩展查询和命令。
* 您希望对查询和命令使用不同的数据模型。 在处理复杂域时很有用。
* 您想使用基于事件源的UI架构。

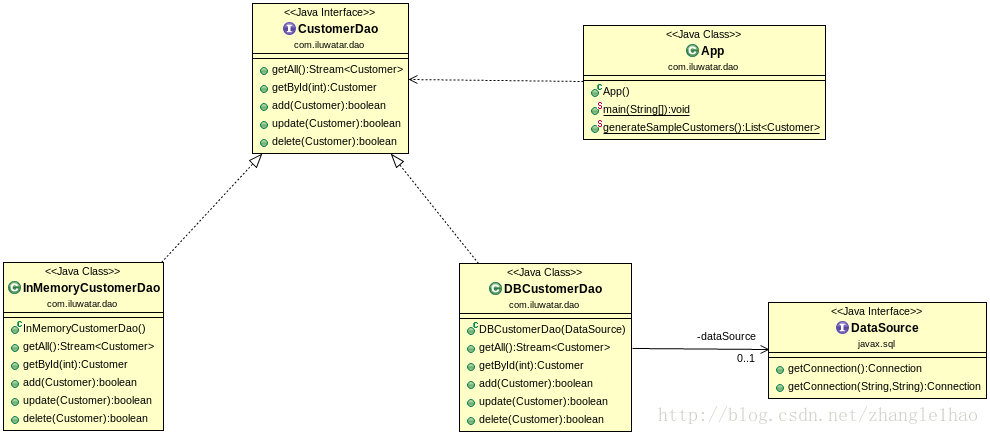
**权威信息**

* [Greg Young - CQRS, Task Based UIs, Event Sourcing agh!](http://codebetter.com/gregyoung/2010/02/16/cqrs-task-based-uis-event-sourcing-agh/)
* [Martin Fowler - CQRS](https://martinfowler.com/bliki/CQRS.html)
* [Oliver Wolf - CQRS for Great Good](https://www.youtube.com/watch?v=Ge53swja9Dw)

**19.数据访问对象模式**

| **标题** | **类别** | **难度** |
| --- | --- | --- |
| 数据访问对象模式（dao design） | 持久层 | 初始难度 |

**意图**

对象提供了某种类型的数据库的抽象接口或其他持久机制。   


**适用范围**

在以下任何情况下使用数据访问对象

* 当您想要整合数据层的访问方式
* 当你想避免编写多个数据检索/持久层

**权威信息**

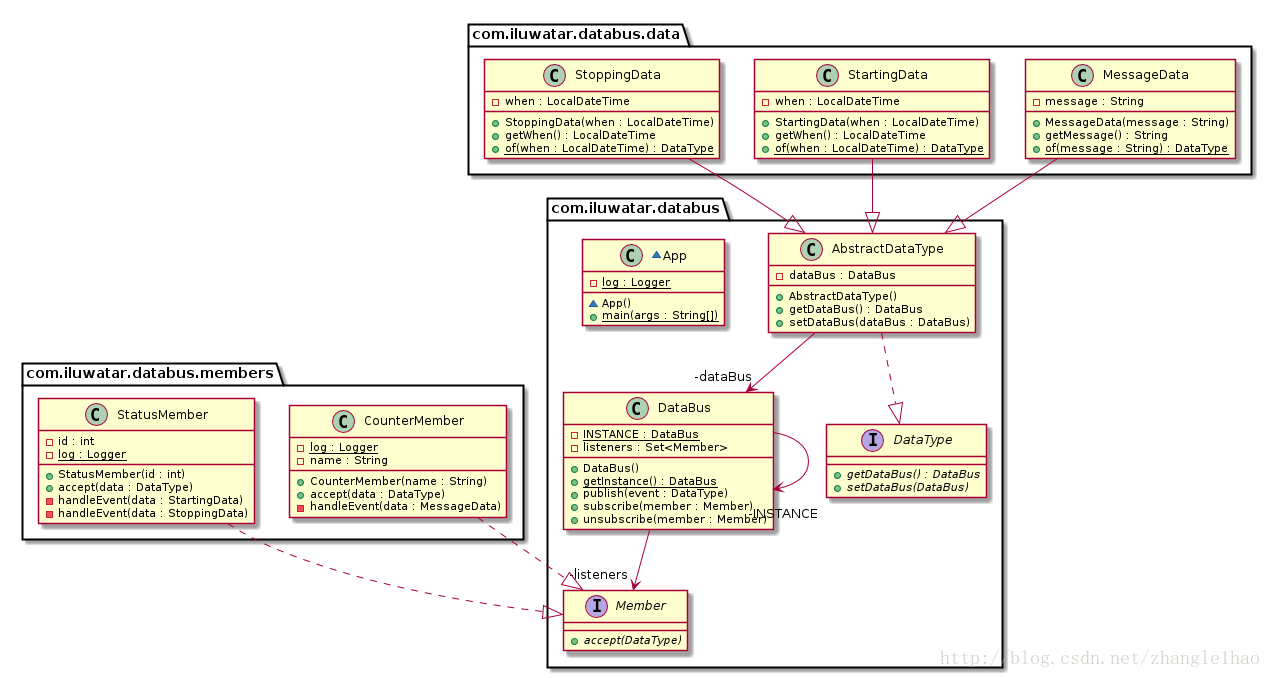
* [J2EE Design Patterns](http://www.amazon.com/J2EE-Design-Patterns-William-Crawford/dp/0596004273/ref=sr_1_2)

**20.数据总线模式**

| **标题** | **类别** | **难度** |
| --- | --- | --- |
| .数据总线模式（Data Bus design） | 结构型 | 中等难度 |

**意图**

在不需要应用组件彼此了解的情况下,发送消息/事件。 他们只需要知道关于要发送的消息/事件的类型。



**适用访问**

使用数据总线模式

* 您希望您的组件自己决定要接收哪些消息/事件
* 你想有多对多的交互
* 您希望您的组件彼此不了解

**相关模式**

与数据总线相似

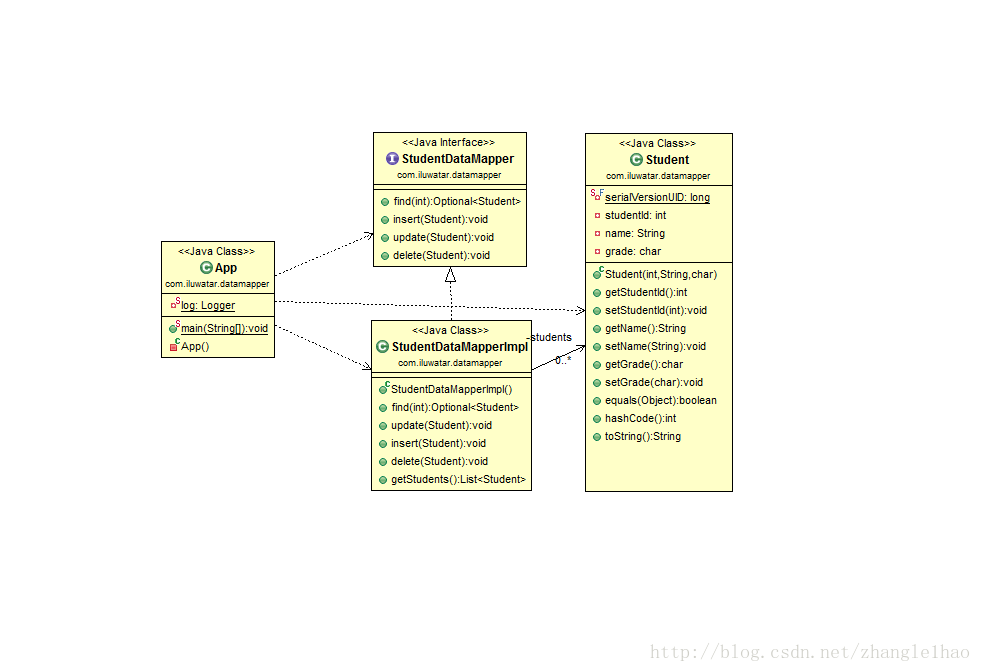
* 调停员模式与数据总线成员自己决定是否要接受任何给定的消息
* 观察者模式，支持多对多通信
* 发布/订阅模式与数据总线模式相比，解耦发布者和订阅者

**21.数据映射模式**

| **标题** | **类别** | **难度** |
| --- | --- | --- |
| 数据映射模式（data-mapper design） | 结构型 | 简单难度 |

**意图**

用于在对象和数据库之间移动数据，同时保持它们彼此独立。



**适用范围**

在以下任何情况下使用数据映射器

* 当您要将数据对象与数据库访问层分离时
* 当你想编写多个数据检索/持久化实现

**权威信息**

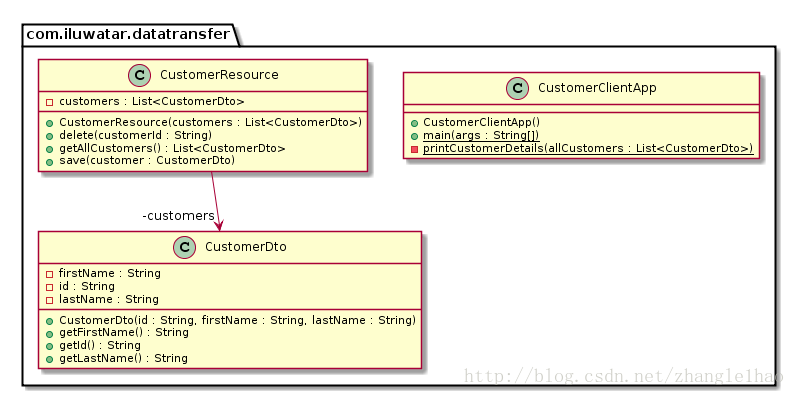
* [Data Mapper](http://richard.jp.leguen.ca/tutoring/soen343-f2010/tutorials/implementing-data-mapper/)

**22.数据传输对象模式**

| **标题** | **类别** | **难度** |
| --- | --- | --- |
| 数据传输对象模式（data-transfer-object design） | 结构型 | 简单难度 |

**意图**

从客户端到服务器一次性传递具有多个属性的数据，   
以避免多次呼叫到远程服务器。



**适用范围**

使用数据传输对象模式

* 客户端要求多个信息。 信息是相关的。
* 当您想提高性能以获取资源时。
* 您希望减少远程呼叫的数量。

**权威信息**

* [Design Pattern - Transfer Object Pattern](https://www.tutorialspoint.com/design_pattern/transfer_object_pattern.htm)
* [Data Transfer Object](https://msdn.microsoft.com/en-us/library/ff649585.aspx)

**23.装饰模式**

| **标题** | **类别** | **难度** |
| --- | --- | --- |
| 装饰模式（Decorator design） | 结构型 | 简单难度 |

**[众所周知](https://www.baidu.com/s?wd=%E4%BC%97%E6%89%80%E5%91%A8%E7%9F%A5&tn=24004469_oem_dg&rsv_dl=gh_pl_sl_csd" \t "_blank)**

Wrapper

**意图**

给对象动态附加额外的责任,装饰模式为扩展子类提供了灵活的替代方案。

**解释**

真实世界

生活在附近山丘上的一个愤怒的巨魔。 平时它是空手的，但有时它有武器。 为了武装巨魔，没有必要创建一个新的巨魔，而是用合适的武器动态地装饰它。

字面意思

装饰模式使您可以在运行时动态地更改对象的行为，将其包装在装饰器类的对象中。

维基解释

在面向对象编程中，装饰器模式是一种设计模式，允许静态或动态地将行为添加到单个对象，而不会影响同一类中其他对象的行为。 装饰师模式对于遵守单一责任原则通常是有用的，因为它允许在具有独特关注领域的类别之间划分功能。

**编程例子**

让我们看看巨魔的例子。 首先我们有一个简单的巨魔实现巨魔接口

public interface Troll {

void attack();

int getAttackPower();

void fleeBattle();

}

public class SimpleTroll implements Troll {

private static final Logger LOGGER = LoggerFactory.getLogger(SimpleTroll.class);

@Override

public void attack() {

LOGGER.info("The troll tries to grab you!");

}

@Override

public int getAttackPower() {

return 10;

}

@Override

public void fleeBattle() {

LOGGER.info("The troll shrieks in horror and runs away!");

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25

接下来我们要为巨魔添加棒子。 我们可以通过使用装饰器来动态地执行

public class ClubbedTroll implements Troll {

private static final Logger LOGGER = LoggerFactory.getLogger(ClubbedTroll.class);

private Troll decorated;

public ClubbedTroll(Troll decorated) {

this.decorated = decorated;

}

@Override

public void attack() {

decorated.attack();

LOGGER.info("The troll swings at you with a club!");

}

@Override

public int getAttackPower() {

return decorated.getAttackPower() + 10;

}

@Override

public void fleeBattle() {

decorated.fleeBattle();

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26

这是巨魔的行为

*// simple troll*

Troll troll = new SimpleTroll();

troll.attack(); *//巨魔试图抓住你！*

troll.fleeBattle(); *// 巨魔逃跑了!*

*//通过添加装饰器来改变简单巨魔的行为*

troll = new ClubbedTroll(troll);

troll.attack(); *//巨魔试图抓住你！ 巨魔用棍子敲你头！*

troll.fleeBattle(); *//巨魔逃跑了!*

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9

**适用范围**

使用装饰模式如下

* 动态，透明地为各个对象添加责任，即不影响其他对象
* 可撤销的责任
* 通过子类化扩展是不切实际的。 有时，大量的独立扩展是可能的，并且会产生一个子类的爆炸，装饰模式支持任意组合。 或者类定义可能被隐藏或以其他方式不可用于子类化

**真实世界**

* [java.io.InputStream](http://docs.oracle.com/javase/8/docs/api/java/io/InputStream.html), [java.io.OutputStream](http://docs.oracle.com/javase/8/docs/api/java/io/OutputStream.html),   
  [java.io.Reader](http://docs.oracle.com/javase/8/docs/api/java/io/Reader.html) and [java.io.Writer](http://docs.oracle.com/javase/8/docs/api/java/io/Writer.html)
* [java.util.Collections#synchronizedXXX()](http://docs.oracle.com/javase/8/docs/api/java/util/Collections.html#synchronizedCollection-java.util.Collection-)
* [java.util.Collections#unmodifiableXXX()](http://docs.oracle.com/javase/8/docs/api/java/util/Collections.html#unmodifiableCollection-java.util.Collection-)
* [java.util.Collections#checkedXXX()](http://docs.oracle.com/javase/8/docs/api/java/util/Collections.html#checkedCollection-java.util.Collection-java.lang.Class-)

**权威信息**

* [Design Patterns: Elements of Reusable Object-Oriented Software](http://www.amazon.com/Design-Patterns-Elements-Reusable-Object-Oriented/dp/0201633612)
* [Functional Programming in Java: Harnessing the Power of Java 8 Lambda Expressions](http://www.amazon.com/Functional-Programming-Java-Harnessing-Expressions/dp/1937785467/ref=sr_1_1)
* [J2EE Design Patterns](http://www.amazon.com/J2EE-Design-Patterns-William-Crawford/dp/0596004273/ref=sr_1_2)

**24.委托模式**

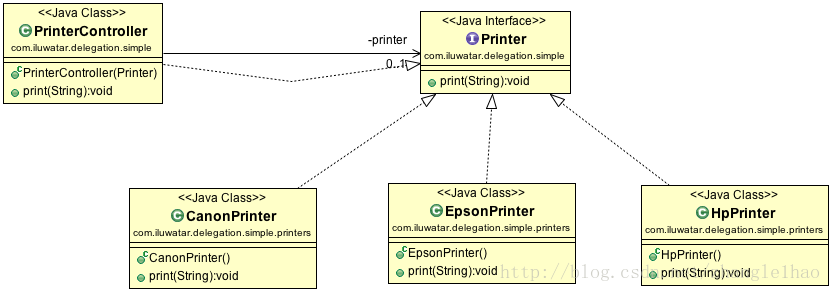
| **标题** | **类别** | **难度** |
| --- | --- | --- |
| 委托模式（delegation design） | 行为型 | 简单难度 |

**众所周知**

代理模式(Proxy Pattern)

**意图**

这是一种技术，其中一个对象表现某种行为给外部，但是该行为的实现委托给相关联的对象。(博主描述，比如你一个对象要调用一个方法，但是你不直接操作这个对象本身,而是操作和他关联的一个其他对象，并且该对象实现了那个方法的接口)



**适用范围**

为了实现以下目的，使用委托模式

* 减少方法与类的耦合
* 组件的行为相同，但意识到这种情况在将来会发生变化。

**权威信息**

* [Delegate Pattern: Wikipedia](https://en.wikipedia.org/wiki/Delegation_pattern)
* [Proxy Pattern: Wikipedia](https://en.wikipedia.org/wiki/Proxy_pattern)

**25.依赖注入模式**

| **标题** | **类别** | **难度** |
| --- | --- | --- |
| 依赖注入模式（Dependency Injection） | 行为型 | 简单难度 |

**意图**

依赖注入是一种软件设计模式，其中一个或更多的依赖（或服务）被注入或通过引用传递到依赖对象（或客户端）并作为客户端状态的一部分。该模式将客户机的依赖关系的创建与其自身的分离行为，允许程序设计松散耦合并遵循   
控制反转和单一责任原则。

**适用范围**

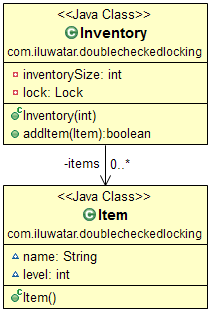
何时使用依赖注入模式

* 当您需要从对象中删除具体实现的知识时
* 使用模拟对象或存根单独测试类的单元测试

**26.双重锁模式**

| **标题** | **类别** | **难度** |
| --- | --- | --- |
| 双重锁模式（Double Checked Locking design） | 并发型 | 简单难度,原子性 |

**意图**

通过先判断是否需要获取锁来减少锁获取的开销,如果需要执行锁，在进行实际的锁操作。在锁定之后再判断一次需要获取锁,若不需要择表明不需要执行逻辑，加快锁的释放。   


**适用范围**

何时使用双重检查锁定模式

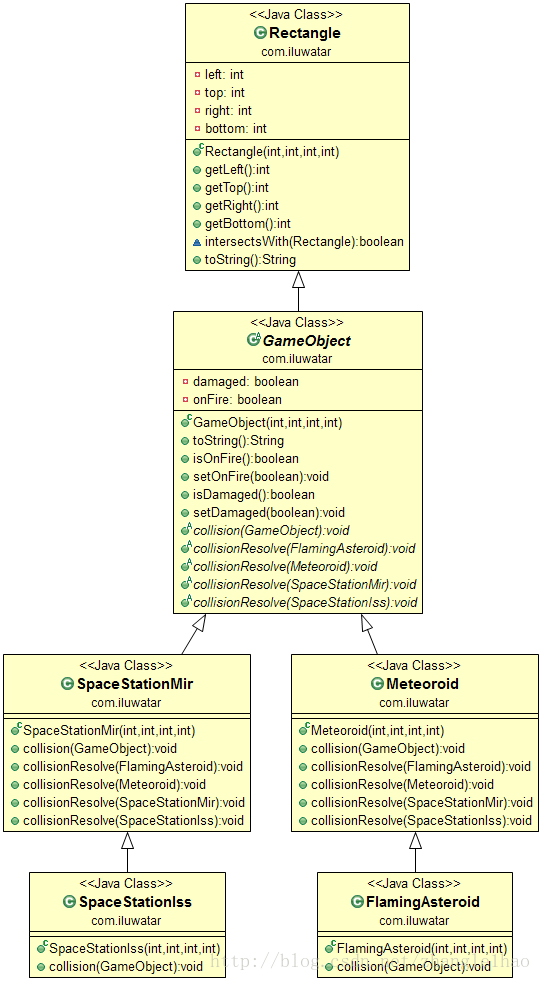
* 在对象创建中存在并发访问，例如 单例，您想要创建同一个类的单个实例，并检查是否为空，或者当有两个或多个线程检查实例是否为空时，可能还不够。
* 方法的行为根据一些约束而改变，并且这些约束在此方法中改变，就有一个并发访问。

**27.双重派对模式**

| **标题** | **类别** | **难度** |
| --- | --- | --- |
| 双重派对模式（Double Dispatch design） | 其他 | 中等难度,原子性 |

**意图**

双重分发模式创建基于接收器的可维护的动态行为和参数类型。

—

**适用范围**

何时使用双重派对模式

* 动态行为不仅仅基于接收对象的类型而是基于接收方法的参数类型来定义。

**真实例子**

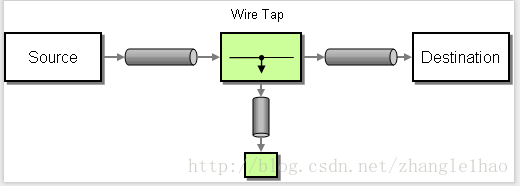
* [ObjectOutputStream](https://docs.oracle.com/javase/8/docs/api/java/io/ObjectOutputStream.html)

**28. 执行程序流监测器**

| **标题** | **类别** | **难度** |
| --- | --- | --- |
| 执行程序流监测器（EIP WireTap design） | 企业集成 | 中等难度 |

**意图**

在大多数集成案例中，需要监控流经系统的消息。 通常实现通过拦截消息并将其重定向到不同的位置，如控制台，文件系统或数据库。重要的是，这种功能不应该修改原始消息并影响处理路径。



**适用范围**

使用执行程序流监测器

* 您需要监控流经系统的消息
* 您需要将相同的未更改的消息重定向到两个不同的端点/路径

**权威信息**

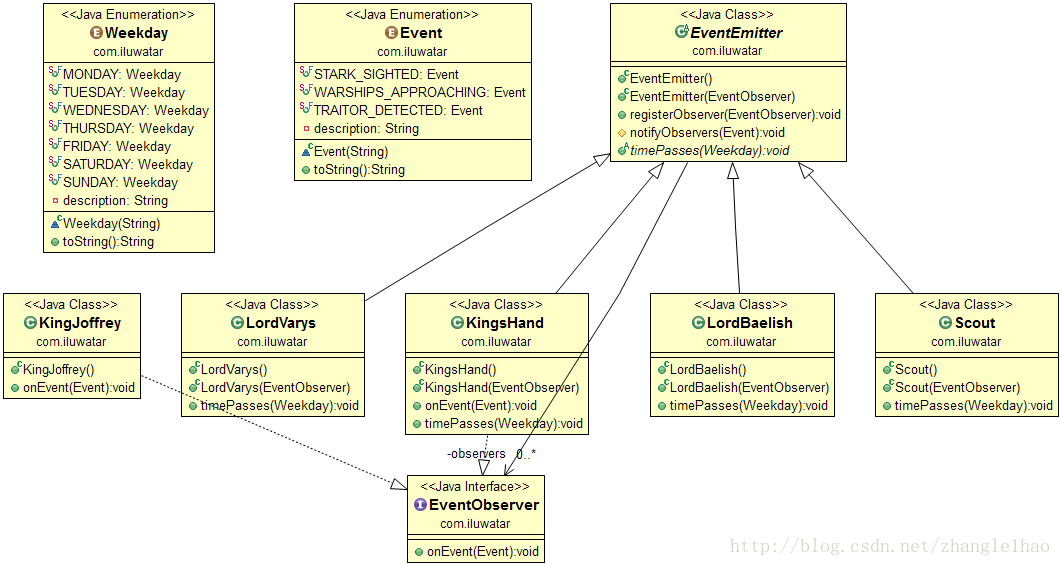
* [Gregor Hohpe, Bobby Woolf - Enterprise Integration Patterns](http://www.enterpriseintegrationpatterns.com/patterns/messaging/WireTap.html)
* [Apache Camel - Documentation](http://camel.apache.org/wire-tap.html)

**29. 事件聚合器**

| **标题** | **类别** | **难度** |
| --- | --- | --- |
| 事件聚合器（event-aggregator design） | 结构型 | 简单难度,响应式 |

**意图**

一个具有很多对象的系统会导致复杂性，客户端希望订阅活动。   
客户端必须找到并注册每个对象分别，如果每个对象有多个事件，那么每个事件需要单独订阅。 事件聚合器作为事件的单一来源, 它注册了许多对象的所有事件允许客户端仅使用聚合器注册。



**适用范围**

何时使用事件聚合器模式

* 当你有很多的潜在的事件来源(对象), 但不想让观察者处理注册，您可以将注册逻辑集中到事件聚集器。 除了简化注册，事件聚合器简化了使用观察者的内存管理问题。

**权威信息**

* [Martin Fowler - Event Aggregator](http://martinfowler.com/eaaDev/EventAggregator.html)

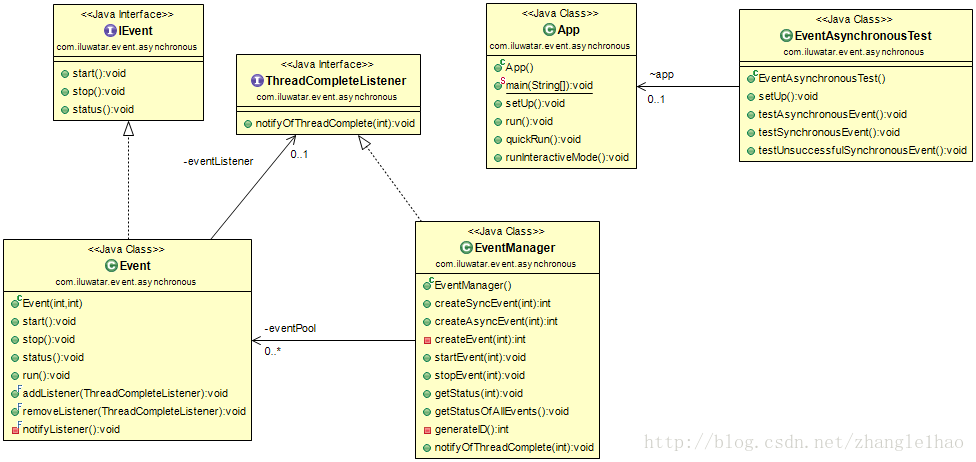
**30. 事件异步**

| **标题** | **类别** | **难度** |
| --- | --- | --- |
| 事件异步（Event-based Asynchronous design） | 并发 | 中等难度,性能 |

**意图**

基于事件的异步模式提供了多线程应用程序的优势，同时隐藏了许多应用程序   
的多线程设计中固有的复杂问题。 使用支持此模式的类可以允许您：

* 在后台执行耗时的任务，例如下载和数据库操作，而不会中断您的应用程序。
* 同时执行多个操作，每次完成时接收通知。
* 等待资源可用，而不停止（“悬挂”）您的应用程序。
* 使用熟悉的事件和委托模型与挂起的异步操作进行通信。



**适用范围**

使用基于事件的异步模式

* 需要耗费时间的任务,并且运行在后台不会中断当前的应用程序。

**权威信息**

* [Event-based Asynchronous Pattern Overview](https://msdn.microsoft.com/en-us/library/wewwczdw%28v=vs.110%29.aspx?f=255&MSPPError=-2147217396)

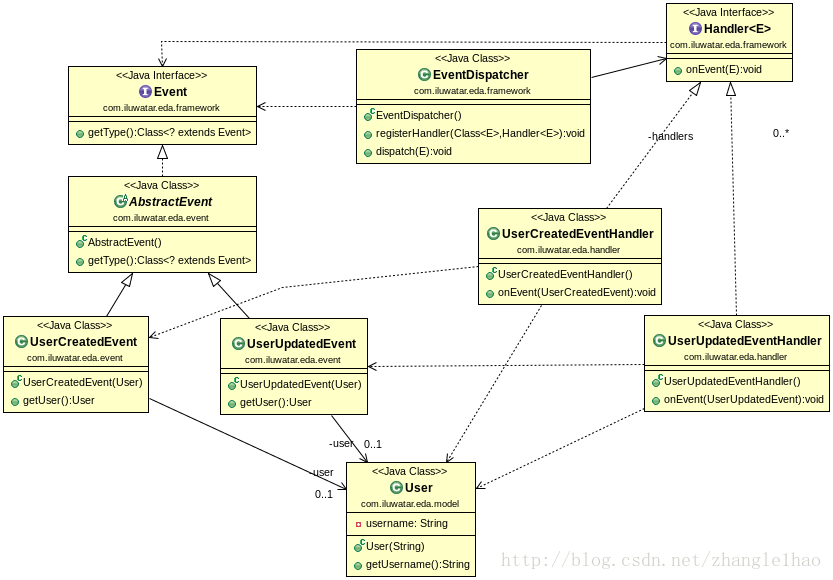
**31. 基于事件驱动的体系结构**

| **标题** | **类别** | **难度** |
| --- | --- | --- |
| 基于事件驱动的体系结构（Event Driven Architecture design） | 结构型 | 困难,响应式 |

**意图**

发送或通知对象状态的变化，给使用了基于事件驱动的体系结构其他应用。

**适用范围**

—   
使用事件驱动的架构

* 你想创建一个松散耦合的系统
* 你想建立一个更加敏感的系统
* 你想要一个更容易扩展的系统

**真实例子**

* SendGrid, an email API, sends events whenever an email is processed, delivered, opened etc… (<https://sendgrid.com/docs/API_Reference/Webhooks/event.html>)
* Chargify, a billing API, exposes payment activity through various events (<https://docs.chargify.com/api-events>)
* Amazon’s AWS Lambda, lets you execute code in response to events such as changes to Amazon S3 buckets, updates to an Amazon DynamoDB table, or custom events generated by your applications or devices. (<https://aws.amazon.com/lambda>)
* MySQL runs triggers based on events such as inserts and update events happening on database tables.

**权威信息**

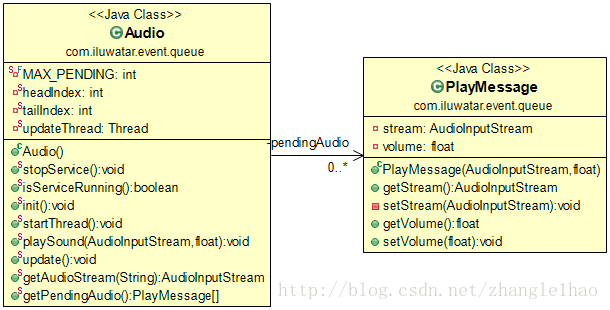
* [Event-driven architecture - Wikipedia](https://en.wikipedia.org/wiki/Event-driven_architecture)
* [Fundamental Components of an Event-Driven Architecture](http://giocc.com/fundamental-components-of-an-event-driven-architecture.html)
* [Real World Applications/Event Driven Applications](https://wiki.haskell.org/Real_World_Applications/Event_Driven_Applications)
* [Event-driven architecture definition](http://searchsoa.techtarget.com/definition/event-driven-architecture)

**32. 事件队列**

| **标题** | **类别** | **难度** |
| --- | --- | --- |
| 事件队列（event-queue） | 并发 | 简单难度,队列 |

**意图**

事件队列是一个很好的模式，如果你有一个有限的可访问资源（例如：音频或数据库），但您需要处理所有要使用的请求。它将所有请求放入队列中并异步处理。当事件完成时，从队列中移除，并给下一个事件提供资源。



**适用范围**

使用事件队列模式

* 异步队列接受未知的请求,访问有限的资源

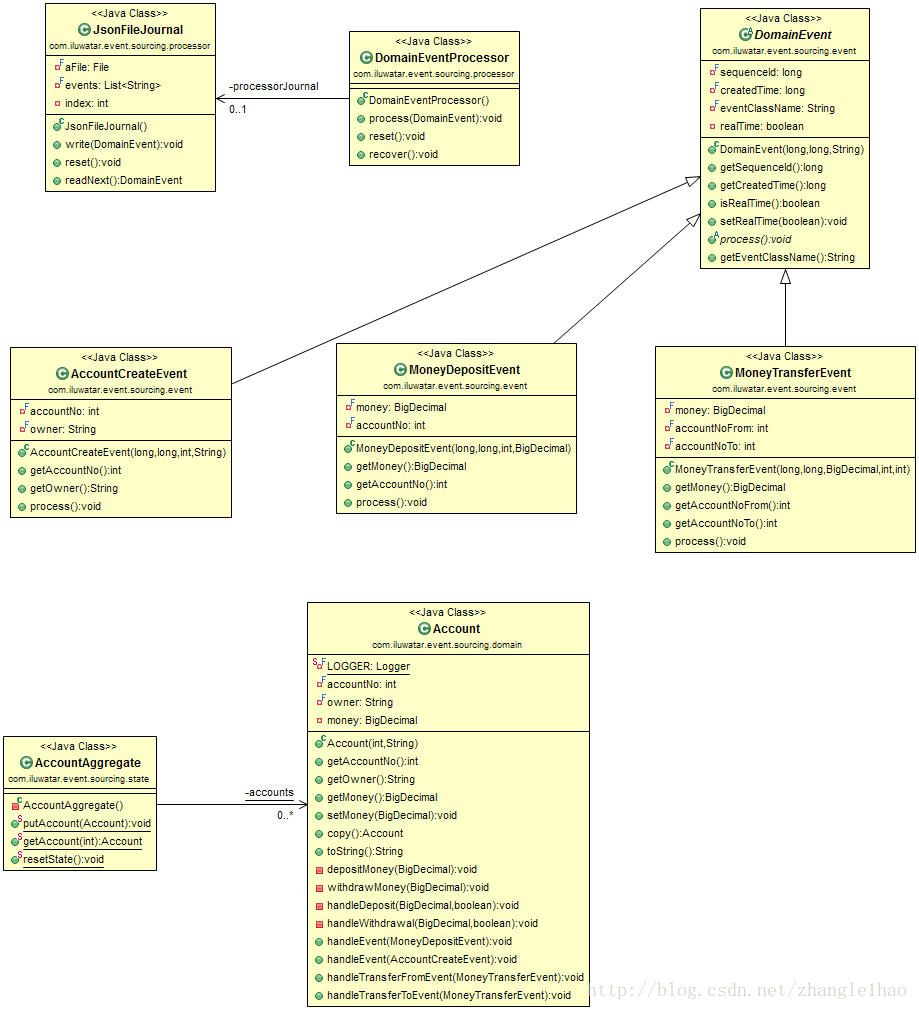
**权威信息**

* [Mihaly Kuprivecz - Event Queue] (<http://gameprogrammingpatterns.com/event-queue.html>)

**33. 事件捕捉**

| **标题** | **类别** | **难度** |
| --- | --- | --- |
| 事件捕捉（event sourcing design） | 结构型 | 困难难度,性能 |

**意图**

不要仅将数据的当前状态存储在域中，而是使用追加专用存储来记录对该数据执行的所有操作。贮存系统可以让领域对象物质化(在json文件中保存对对象的操作)。 这可以简化复杂域中的任务，避免在同步数据模型和业务领域的同时，同时提高性能，可扩展性和响应能力。 它还可以为事务数据提供一致性，并保持完整的审计跟踪和历史记录，从而可以实现补偿操作。   


**适用范围**

何时使用事件捕捉模式

* 即使您的应用程序状态具有复杂的关系数据结构，您仍然需要保持应用程序状态的高性能
* 您需要记录您的应用程序状态和恢复任何时间的状态的能力。
* 您需要通过重播过去的事件来调试生产问题。

**真实世界**

* [The Lmax Architecture] (<https://martinfowler.com/articles/lmax.html>)

**权威信息**

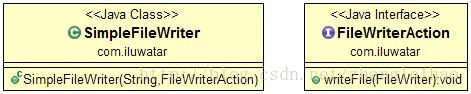
* [Martin Fowler - Event Sourcing] (<https://martinfowler.com/eaaDev/EventSourcing.html>)
* [Event Sourcing | Microsoft Docs] (<https://docs.microsoft.com/en-us/azure/architecture/patterns/event-sourcing>)
* [Reference 3: Introducing Event Sourcing] (<https://msdn.microsoft.com/en-us/library/jj591559.aspx>)

**34. 执行环绕**

| **标题** | **类别** | **难度** |
| --- | --- | --- |
| 执行环绕（execute around design） | 其他 | 简单难度，术语 |

**意图**

执行围绕术语的意思是释放用户的某些动作,应始终在业务方法之前和之后执行。 一个很好的例子   
这是资源分配和释放，让用户只需要关系如何操作资源。



**适用范围**

何时使用执行环绕

* 你使用一个API, 同时需要调用别的方法在周围。

**权威信息**

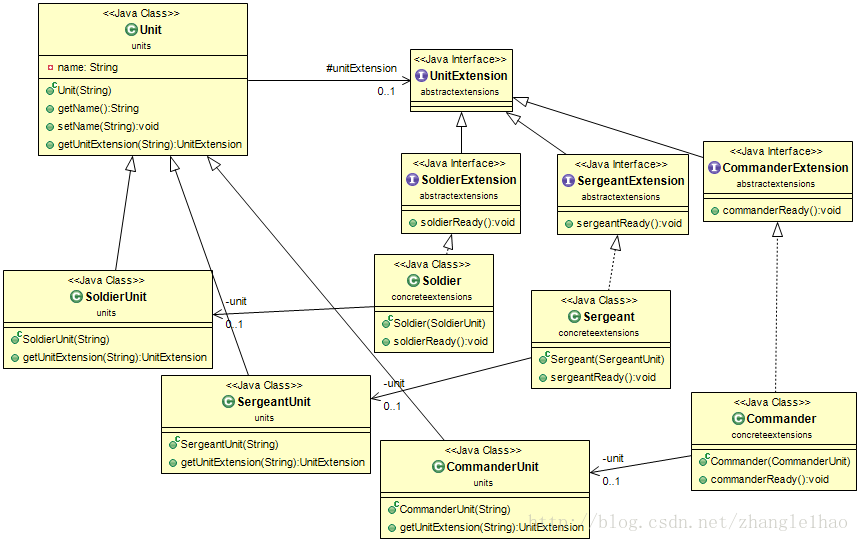
* [Functional Programming in Java: Harnessing the Power of Java 8 Lambda Expressions](http://www.amazon.com/Functional-Programming-Java-Harnessing-Expressions/dp/1937785467/ref=sr_1_1)

**3５. 扩展对象**

| **标题** | **类别** | **难度** |
| --- | --- | --- |
| 扩展对象（ Extension objects） | 行为型 | 中等难度 |

**意图**

预计在将来需要扩展对象的接口。 额外接口由扩展对象定义。



**适用范围**

在以下情况下使用扩展对象模式

* 您需要支持向现有类添加新的或不可预见的接口，并且不希望影响不需要此新界面的客户端。 扩展对象允许您通过在单独的类中定义相关操作来保持相关操作
* 代表关键抽象的类对于不同的客户端起着不同的作用。 课程的角色数量应该是开放的。 需要保留关键抽象本身。 例如，客户对象仍然是客户对象，即使不同的子系统以不同的方式查看它。
* 一个类的新行为是可扩展的，而不是将它进行子类化。

**真实世界**

* [OpenDoc](https://en.wikipedia.org/wiki/OpenDoc)
* [Object Linking and Embedding](https://en.wikipedia.org/wiki/Object_Linking_and_Embedding)

**36. 外观模式**

| **标题** | **类别** | **难度** |
| --- | --- | --- |
| 外观模式（ facade design）） | 结构型 | 简单难度,GOF |

**意图**

为子系统中的一组接口提供统一的接口。   
外观模式定义了一个更高级别的界面，使子系统更易于使用。

**解释**

现实例子

金矿如何运作？ “矿工们下去挖金了！” 你说。 这就是你所相信的，因为你正在使用一个简单的界面，goldmine在外面提供，在内部它必须做很多事情才能实现。 复杂子系统的这个简单界面是一个外观。

字面意思

外观模式为复杂子系统提供简化的界面。

维基解释

外观是一个对象，它提供了一个更简单的界面，用于更大量的代码，例如类库。

**编程例子**

从上面我们的金矿做例子。 在这里，我们有矮人的矿工级别

*//矮人工人*

public abstract class DwarvenMineWorker {

private static final Logger LOGGER = LoggerFactory.getLogger(DwarvenMineWorker.class);

public void goToSleep() {

LOGGER.info("{} goes to sleep.", name());

}

public void wakeUp() {

LOGGER.info("{} wakes up.", name());

}

public void goHome() {

LOGGER.info("{} goes home.", name());

}

public void goToMine() {

LOGGER.info("{} goes to the mine.", name());

}

private void action(Action action) {

switch (action) {

case GO\_TO\_SLEEP:

goToSleep();

break;

case WAKE\_UP:

wakeUp();

break;

case GO\_HOME:

goHome();

break;

case GO\_TO\_MINE:

goToMine();

break;

case WORK:

work();

break;

default:

LOGGER.info("Undefined action");

break;

}

}

public void action(Action... actions) {

for (Action action : actions) {

action(action);

}

}

public abstract void work();

public abstract String name();

static enum Action {

GO\_TO\_SLEEP, WAKE\_UP, GO\_HOME, GO\_TO\_MINE, WORK

}

}

*//矮人隧道挖掘机操作者*

public class DwarvenTunnelDigger extends DwarvenMineWorker {

private static final Logger LOGGER = LoggerFactory.getLogger(DwarvenTunnelDigger.class);

@Override

public void work() {

LOGGER.info("{} creates another promising tunnel.", name());

}

@Override

public String name() {

return "Dwarven tunnel digger";

}

}

*//矮人黄金挖掘机操作者*

public class DwarvenGoldDigger extends DwarvenMineWorker {

private static final Logger LOGGER = LoggerFactory.getLogger(DwarvenGoldDigger.class);

@Override

public void work() {

LOGGER.info("{} digs for gold.", name());

}

@Override

public String name() {

return "Dwarf gold digger";

}

}

*//矮人矿车操作者*

public class DwarvenCartOperator extends DwarvenMineWorker {

private static final Logger LOGGER = LoggerFactory.getLogger(DwarvenCartOperator.class);

@Override

public void work() {

LOGGER.info("{} moves gold chunks out of the mine.", name());

}

@Override

public String name() {

return "Dwarf cart operator";

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36
* 37
* 38
* 39
* 40
* 41
* 42
* 43
* 44
* 45
* 46
* 47
* 48
* 49
* 50
* 51
* 52
* 53
* 54
* 55
* 56
* 57
* 58
* 59
* 60
* 61
* 62
* 63
* 64
* 65
* 66
* 67
* 68
* 69
* 70
* 71
* 72
* 73
* 74
* 75
* 76
* 77
* 78
* 79
* 80
* 81
* 82
* 83
* 84
* 85
* 86
* 87
* 88
* 89
* 90
* 91
* 92
* 93
* 94
* 95
* 96
* 97
* 98
* 99
* 100
* 101
* 102
* 103
* 104
* 105
* 106
* 107

为了操作所有这些金矿工人，我们有一个外观

public class DwarvenGoldmineFacade {

private final List<DwarvenMineWorker> workers;

public DwarvenGoldmineFacade() {

workers = new ArrayList<>();

workers.add(new DwarvenGoldDigger());

workers.add(new DwarvenCartOperator());

workers.add(new DwarvenTunnelDigger());

}

public void startNewDay() {

makeActions(workers, DwarvenMineWorker.Action.WAKE\_UP, DwarvenMineWorker.Action.GO\_TO\_MINE);

}

public void digOutGold() {

makeActions(workers, DwarvenMineWorker.Action.WORK);

}

public void endDay() {

makeActions(workers, DwarvenMineWorker.Action.GO\_HOME, DwarvenMineWorker.Action.GO\_TO\_SLEEP);

}

private static void makeActions(Collection<DwarvenMineWorker> workers,

DwarvenMineWorker.Action... actions) {

for (DwarvenMineWorker worker : workers) {

worker.action(actions);

}

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30

现在使用这个外观

DwarvenGoldmineFacade facade = new DwarvenGoldmineFacade();

facade.startNewDay();

*// Dwarf gold digger wakes up.*

*// Dwarf gold digger goes to the mine.*

*// Dwarf cart operator wakes up.*

*// Dwarf cart operator goes to the mine.*

*// Dwarven tunnel digger wakes up.*

*// Dwarven tunnel digger goes to the mine.*

facade.digOutGold();

*// Dwarf gold digger digs for gold.*

*// Dwarf cart operator moves gold chunks out of the mine.*

*// Dwarven tunnel digger creates another promising tunnel.*

facade.endDay();

*// Dwarf gold digger goes home.*

*// Dwarf gold digger goes to sleep.*

*// Dwarf cart operator goes home.*

*// Dwarf cart operator goes to sleep.*

*// Dwarven tunnel digger goes home.*

*// Dwarven tunnel digger goes to sleep.*

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19

**适用范围**

何时使用外观模式

* 您想为复杂子系统提供简单的界面。子系统在发展时通常会变得更加复杂。大多数模式在应用时会导致越来越多的类。这使得子系统更可重用并且更容易定制，但是对于不需要自定义的客户端也变得更难使用。外观可以提供对于大多数客户端来说足够好的子系统简单默认视图。只有需要更多可定制性的客户才需要越过外观模式。
* 客户端和抽象的实现类之间存在很多依赖关系。引入一个外观来将子系统与客户端和其他子系统分离，从而提高子系统的独立性和可移植性。   
  \*你 想分层你的子系统。使用门面来定义每个子系统级别的入口点。如果子系统是依赖的，那么您可以通过使它们仅通过其外观立面相互通信来简化它们之间的依赖关系

**权威信息**

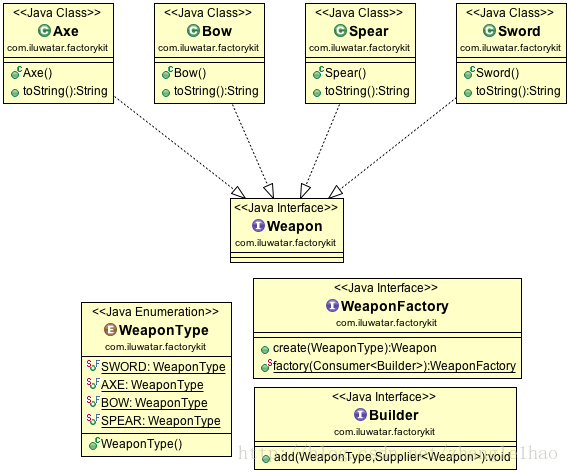
* [Design Patterns: Elements of Reusable Object-Oriented Software](http://www.amazon.com/Design-Patterns-Elements-Reusable-Object-Oriented/dp/0201633612)

**37. 工厂套件**

| **标题** | **类别** | **难度** |
| --- | --- | --- |
| 工厂套件（ Factory Kit design） | 创建型 | 简单难度,函数式 |

**意图**

使用分离的构建器和工厂接口定义一个内容不可变化的工厂。



**适用范围**

何时使用工厂套件模式

* 一个类不能预料到它必须创建的对象类
* 您只需要一个新的自定义构建器实例，而不是全局实例,你明确地想要定义工厂可以建立的对象类型
* 你想要一个分离的构建器和创建者界面

**权威信息**

* [Design Pattern Reloaded by Remi Forax:](https://www.youtube.com/watch?v=-k2X7guaArU)

**38. 工厂方法**

| **标题** | **类别** | **难度** |
| --- | --- | --- |
| 工厂方法（ Factory Method design） | 创建型 | 简单难度,GOF |

**众所周知**

Virtual Constructor

**意图**

定义一个用于创建一个对象的接口，但是让子类决定要实例化哪个类。 工厂方法允许类推迟   
实例化到子类。

**解释**

真实世界

铁匠制造武器。 精灵需要精灵武器和兽人需要兽医武器。 根据手头的客户，正确的类型被铁匠制造。

字面意思

它提供了一种将实例化逻辑委托给子类的方法。

维基解释

在基于类的编程中，工厂方法模式是使用工厂方法来处理创建对象的问题的创建模式，而不必指定将要创建的对象的确切类。 这可以通过调用工厂方法（在接口中指定并由子类实现）来实现，或者在基类中实现，并且可选地被派生类覆盖，而不是调用构造函数。

**编程例子**

以我们的铁匠为例。 首先我们有一个铁匠接口和一些实现

public interface Blacksmith {

Weapon manufactureWeapon(WeaponType weaponType);

}

public class ElfBlacksmith implements Blacksmith {

public Weapon manufactureWeapon(WeaponType weaponType) {

return new ElfWeapon(weaponType);

}

}

public class OrcBlacksmith implements Blacksmith {

public Weapon manufactureWeapon(WeaponType weaponType) {

return new OrcWeapon(weaponType);

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15

现在，由于客户来了,铁匠被要求制造符合客户武器类型的武器

Blacksmith blacksmith = new ElfBlacksmith()*;*

blacksmith.manufactureWeapon(WeaponType.SPEAR)*;*

blacksmith.manufactureWeapon(WeaponType.AXE)*;*

// 创建精灵武器

* 1
* 2
* 3
* 4

**适用范围**

何时使用“工厂方法”模式

* 一个类不能预料到它必须创建的对象类
* 一个类希望其子类来指定它创建的对象
* 类将责任委托给几个辅助子类之一，并且您想要本地化代理的哪个助手子类的知识

**真实例子**

* [java.util.Calendar](http://docs.oracle.com/javase/8/docs/api/java/util/Calendar.html#getInstance--)
* [java.util.ResourceBundle](http://docs.oracle.com/javase/8/docs/api/java/util/ResourceBundle.html#getBundle-java.lang.String-)
* [java.text.NumberFormat](http://docs.oracle.com/javase/8/docs/api/java/text/NumberFormat.html#getInstance--)
* [java.nio.charset.Charset](http://docs.oracle.com/javase/8/docs/api/java/nio/charset/Charset.html#forName-java.lang.String-)
* [java.net.URLStreamHandlerFactory](http://docs.oracle.com/javase/8/docs/api/java/net/URLStreamHandlerFactory.html#createURLStreamHandler-java.lang.String-)
* [java.util.EnumSet](https://docs.oracle.com/javase/8/docs/api/java/util/EnumSet.html#of-E-)
* [javax.xml.bind.JAXBContext](https://docs.oracle.com/javase/8/docs/api/javax/xml/bind/JAXBContext.html#createMarshaller--)

**权威信息**

* [Design Patterns: Elements of Reusable Object-Oriented Software](http://www.amazon.com/Design-Patterns-Elements-Reusable-Object-Oriented/dp/0201633612)

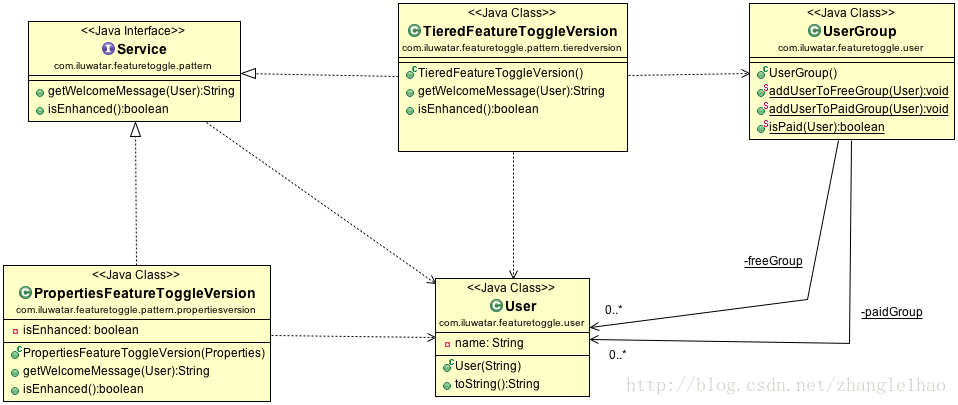
**39. 功能切换**

| **标题** | **类别** | **难度** |
| --- | --- | --- |
| 功能切换（Feature Toggle design） | 行为型 | 简单难度 |

**众所周知**

Feature Flag

**意图**

根据属性或分组切换代码执行路径。 允许新功能被发布，测试并推出。 如果需要，可以快速切换回旧功能。 应该指出的是，这种模式，   
可以轻松引入代码复杂性。 还有一个令人担忧的是，切换最终的旧功能   
逐步淘汰不会被删除，从而导致冗余代码的气味和维护成本提高。   


**适用范围**

使用功能切换模式

* 为不同用户提供不同的功能。
* 逐步推出新功能。
* 在开发和生产环境之间切换。

**权威信息**

* [Martin Fowler 29 October 2010 (2010-10-29).](http://martinfowler.com/bliki/FeatureToggle.html)

**40. 流畅的接口**

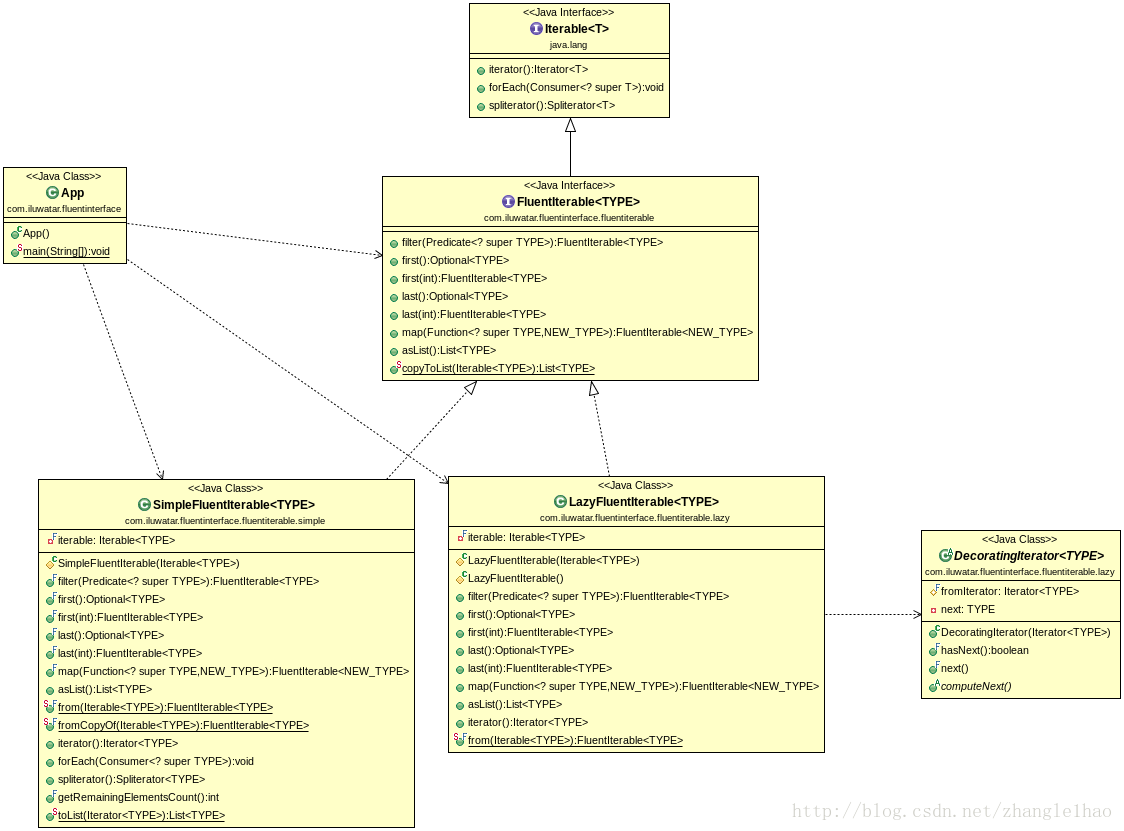
| **标题** | **类别** | **难度** |
| --- | --- | --- |
| 流畅的接口（Fluent Interface design） | 其他 | 中等难度，函数式 |

**意图**

流畅的接口提供了一个易于阅读，流畅的接口，通常会模仿一个领域特定的语言。 使用这种模式可以生成像人类语言一样可供阅读的代码

**实现**

流畅的接口可以使用如下方式来实现

* 方法链 - 调用方法返回一些可以调用其他方法的对象。
* 静态工厂方法和导入
* 命名参数 - 可以使用静态工厂方法在Java中进行模拟。   
  

**适用范围**

使用流畅的界面模式

* 您提供的API将受益于类似DSL(领域特定语言)的用途
* 您有难以配置或使用的对象

**真实例子**

* [Java 8 Stream API](http://www.oracle.com/technetwork/articles/java/ma14-java-se-8-streams-2177646.html)
* [Google Guava FluentInterable](https://github.com/google/guava/wiki/FunctionalExplained)
* [JOOQ](http://www.jooq.org/doc/3.0/manual/getting-started/use-cases/jooq-as-a-standalone-sql-builder/)
* [Mockito](http://mockito.org/)
* [Java Hamcrest](http://code.google.com/p/hamcrest/wiki/Tutorial)

**权威信息**

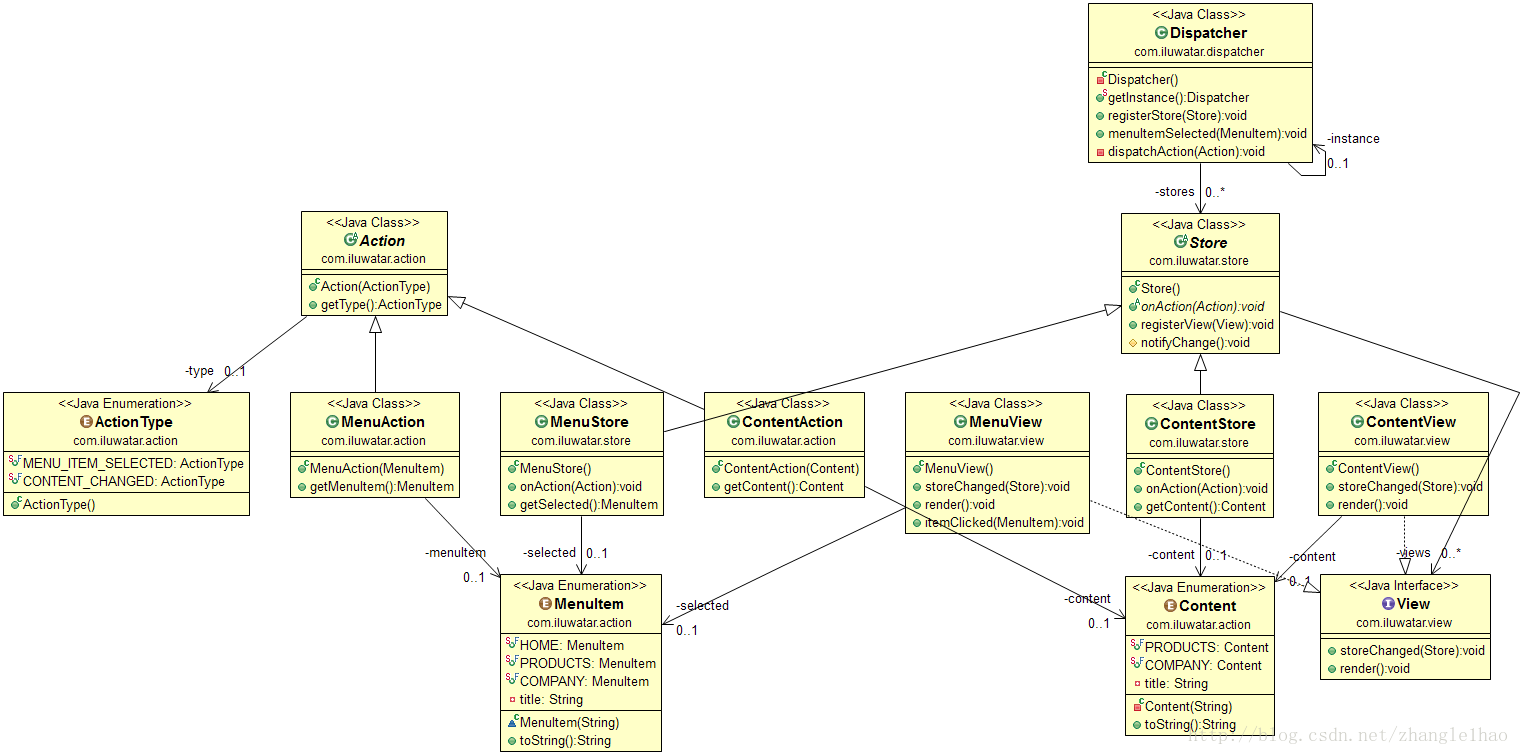
* [Fluent Interface - Martin Fowler](http://www.martinfowler.com/bliki/FluentInterface.html)
* [Evolutionary architecture and emergent design: Fluent interfaces - Neal Ford](http://www.ibm.com/developerworks/library/j-eaed14/)
* [Internal DSL](http://www.infoq.com/articles/internal-dsls-java)

**41. 流量模式**

| **标题** | **类别** | **难度** |
| --- | --- | --- |
| 流量模式（flux design） | 持久层 | 中等难度 |

**意图**

Flux避开MVC设计模式，有利于单向数据流。 当一个   
用户与视图进行交互，视图通过中央传播动作调度员，到持有应用程序的数据和业务的各种商业逻辑，更新所有受影响的视图。

**博主思考**   
一个客户端程序如Adnroid程序，定义用户触发的事件, 通过Stroe让模型和视图分离，这在单事件触发单页面的时候体会不出好处，反而会增加定义事件Action的负担，但是单A页面触发的事件，引起多个B,C,View 视图的UI变化时,便能很清晰的追溯整个事件流的过程。   


**适用范围**

使用流量模式时

* 您希望专注于为应用程序的数据创建显式的和可理解的更新路径，这使得在开发过程中跟踪更改变得更简单，并使bug更容易跟踪和修复。

**权威信息**

* [Flux - Application architecture for building user interfaces](http://facebook.github.io/flux/)

**42. 享元模式**

| **标题** | **类别** | **难度** |
| --- | --- | --- |
| 享元模式（flyweight design） | 持久层 | GOF,中等难度,性能 |

**意图**

使用共享技术有效支持大量细粒度的对象

**例子**

真实世界

炼金术士的商店货架上摆满了魔法药水。很多药水都是一样的，所以不需要为它们创建新的对象。相反，一个对象实例可以代表多个货架项目，因此内存占用仍然很小。

字面意思

它通过与相似对象共享尽可能减少内存使用或计算开销。

维基描述

在计算机编程中，享元模式是一种软件设计模式。 享元模式通过与其他类似对象尽可能多的共享数据来最小化内存使用的对象; 当简单的重复表示将使用不可接受的内存量时，它是大量使用对象的一种方式。

**编程例子**

从上面翻译我们的炼金术士商店示例。 首先我们有不同的药水类型

*//药剂*

public interface Potion {

void drink();

}

*//回复药剂(补血？)*

public class HealingPotion implements Potion {

private static final Logger LOGGER = LoggerFactory.getLogger(HealingPotion.class);

@Override

public void drink() {

LOGGER.info("You feel healed. (Potion={})", System.identityHashCode(this));

}

}

*//升水药剂（补蓝？）*

public class HolyWaterPotion implements Potion {

private static final Logger LOGGER = LoggerFactory.getLogger(HolyWaterPotion.class);

@Override

public void drink() {

LOGGER.info("You feel blessed. (Potion={})", System.identityHashCode(this));

}

}

*//隐形药剂*

public class InvisibilityPotion implements Potion {

private static final Logger LOGGER = LoggerFactory.getLogger(InvisibilityPotion.class);

@Override

public void drink() {

LOGGER.info("You become invisible. (Potion={})", System.identityHashCode(this));

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31

然后是实际的享元对象，它是用于创建药水的工厂

public class PotionFactory {

private final Map<PotionType, Potion> potions;

public PotionFactory() {

potions = new EnumMap<>(PotionType.class);

}

Potion createPotion(PotionType type) {

Potion potion = potions.get(type);

if (potion == null) {

switch (type) {

case HEALING:

potion = new HealingPotion();

potions.put(type, potion);

break;

case HOLY\_WATER:

potion = new HolyWaterPotion();

potions.put(type, potion);

break;

case INVISIBILITY:

potion = new InvisibilityPotion();

potions.put(type, potion);

break;

default:

break;

}

}

return potion;

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31

如下使用

PotionFactory factory = new PotionFactory()*;*

factory.createPotion(PotionType.INVISIBILITY).drink()*; // You become invisible. (Potion=6566818)*

factory.createPotion(PotionType.HEALING).drink()*; // You feel healed. (Potion=648129364)*

factory.createPotion(PotionType.INVISIBILITY).drink()*; // You become invisible. (Potion=6566818)*

factory.createPotion(PotionType.HOLY\_WATER).drink()*; // You feel blessed. (Potion=1104106489)*

factory.createPotion(PotionType.HOLY\_WATER).drink()*; // You feel blessed. (Potion=1104106489)*

factory.createPotion(PotionType.HEALING).drink()*; // You feel healed. (Potion=648129364)*

* 1
* 2
* 3
* 4
* 5
* 6
* 7

**适用范围**

享元模式的有效性在很大程度上取决于如何使用和它在哪里使用。 当以下所有的时候应用享元模式   
真正

* 应用程序使用大量的对象
* 存储成本高，因为物体数量庞大
* 一旦外部状态被移除，许多对象组可能被相对较少的共享对象所取代
* 应用程序不依赖于对象标识。 由于可以共享享元对象，所以对于概念上不同的对象，身份测试将返回true。

**真实世界**

* [java.lang.Integer#valueOf(int)](http://docs.oracle.com/javase/8/docs/api/java/lang/Integer.html#valueOf%28int%29) and similarly for Byte, Character and other wrapped types.

**权威信息**

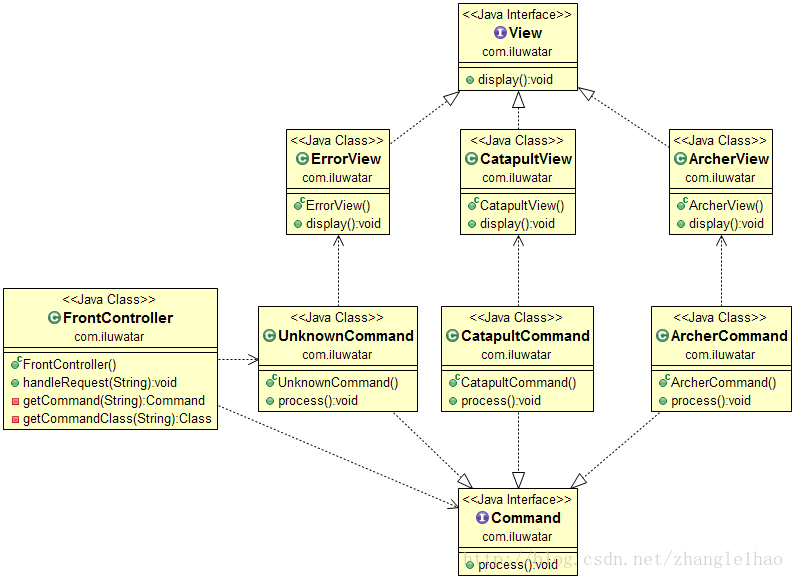
* [Design Patterns: Elements of Reusable Object-Oriented Software](http://www.amazon.com/Design-Patterns-Elements-Reusable-Object-Oriented/dp/0201633612)

**43. 前台控制**

| **标题** | **类别** | **难度** |
| --- | --- | --- |
| 前台控制（front controller design） | 持久层 | 中等难度 |

**意图**

为网站的所有请求引入共同的处理器。 我们可以封装常见的功能，如安全性，   
国际化，路由和记录在一个单一的地方。



**适用范围**

使用前端控制器模式

* 您希望将通用请求处理功能封装在一个单一的位置
* 你想实现动态请求处理，即改变路由而不修改代码
* 使web服务器配置可移植，您只需要注册处理程序Web服务器的具体方式

**真实例子**

* [Apache Struts](https://struts.apache.org/)

**权威信息**

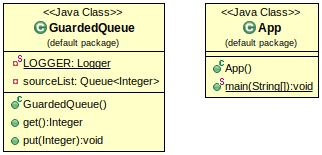
* [J2EE Design Patterns](http://www.amazon.com/J2EE-Design-Patterns-William-Crawford/dp/0596004273/ref=sr_1_2)
* [Presentation Tier Patterns](http://www.javagyan.com/tutorials/corej2eepatterns/presentation-tier-patterns)
* [Patterns of Enterprise Application Architecture](http://www.amazon.com/Patterns-Enterprise-Application-Architecture-Martin/dp/0321127420)

**44. 被监视的挂起**

| **标题** | **类别** | **难度** |
| --- | --- | --- |
| 被监视的挂起（Guarded Suspension design） | 并发 | 简单难度 |

**意图**

当想要对处于不在合适状态的对象执行方法时，可以适用被监视的挂起来处理一些情况



**适用范围**

当开发人员知道方法执行将在一段有限的时间内被阻止时，使用Guarded Suspension模式

**关联模式**

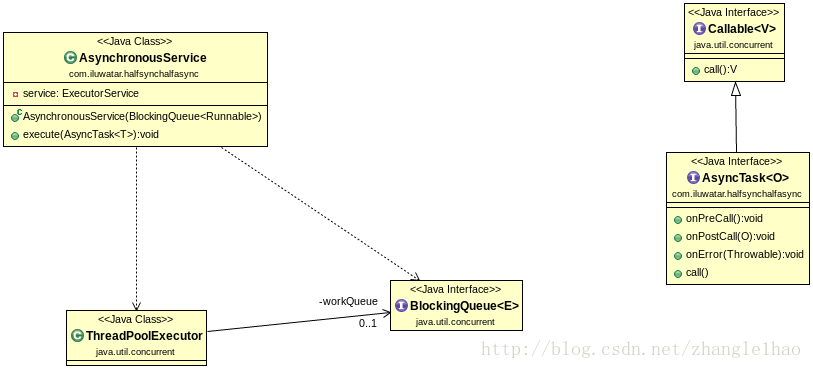
* Balking

**45. 半同步半异步**

| **标题** | **类别** | **难度** |
| --- | --- | --- |
| 半同步半异步（Half-Sync/Half-Async design） | 并发 | 中等难度 |

**意图**

半同步/半异步模式，从异步中分离出同步的I/O,简化并行编程工作，没有降低执行效率。



**适用范围**

使用半同步/半异步模式

* 系统具有以下特点：   
  + 系统必须执行任务以响应异步发生的外部事件，如OS中的硬件中断
  + 专用独立的控制线程为每个外部事件源执行同步I / O是没有效率的
  + 如果同步执行I / O，系统中较高级别的任务可以大大简化。
* 系统中的一个或多个任务必须运行在单个控制线程中，而其他任务可能受益于多线程。

**真实世界**

* [BSD Unix networking subsystem](http://www.cs.wustl.edu/~schmidt/PDF/PLoP-95.pdf)
* [Real Time CORBA](http://www.omg.org/news/meetings/workshops/presentations/realtime2001/4-3_Pyarali_thread-pool.pdf)
* [Android AsyncTask framework](http://developer.android.com/reference/android/os/AsyncTask.html)

**权威信息**

* [Douglas C. Schmidt and Charles D. Cranor - Half Sync/Half Async](http://www.cs.wustl.edu/~schmidt/PDF/PLoP-95.pdf)
* [Pattern Oriented Software Architecture Vol I-V](http://www.amazon.com/Pattern-Oriented-Software-Architecture-Volume-Patterns/dp/0471958697)

**46. 六角结构，又名端口&适配器结构**

| **标题** | **类别** | **难度** |
| --- | --- | --- |
| 六角结构，又名端口&适配器结构（Hexagonal Architecture） | 并发 | 专家难度 |

**众所周知**

* Ports and Adapters
* Clean Architecture
* Onion Architecture

**意图**

不同的用户可以相似的驱动应用程序，自动化测试或批处理脚本驱动，并与其最终的运行时设备和数据库隔离开发和测试。   


**适用范围**

使用六角架构模式

* 当应用程序需要独立于任何框架时
* 重要的是应用程序高度可维护和完全可测试

**教程**

* [Build Maintainable Systems With Hexagonal Architecture](http://java-design-patterns.com/blog/build-maintainable-systems-with-hexagonal-architecture/)

**演讲**

* [Hexagonal Architecture](https://github.com/iluwatar/java-design-patterns/tree/master/hexagonal/etc/presentation.html)

**真实世界**

* [Apache Isis](https://isis.apache.org/) builds generic UI and REST API directly from the underlying domain objects

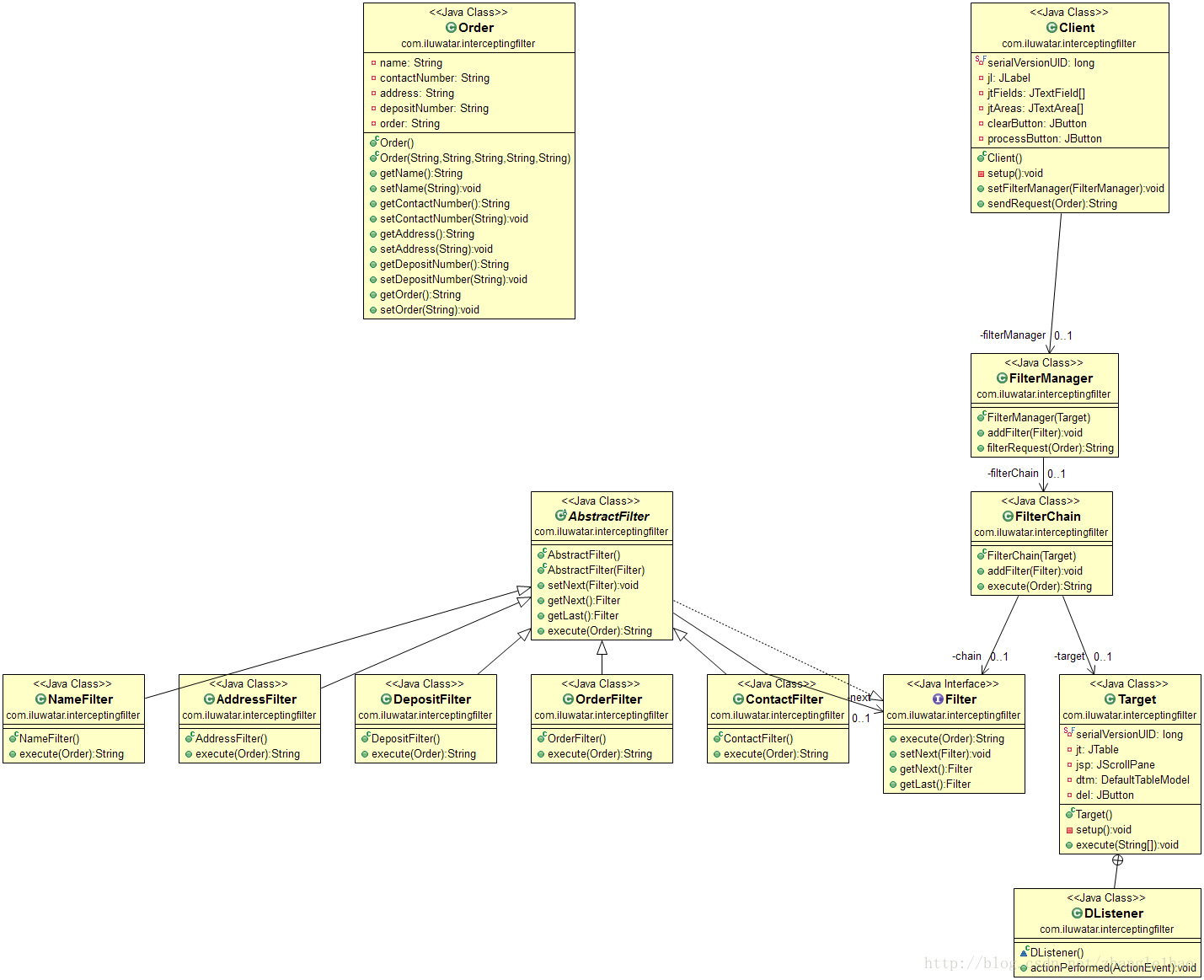
**权威信息**

* [Alistair Cockburn - Hexagonal Architecture](http://alistair.cockburn.us/Hexagonal+architecture)

**47. 拦截过滤**

| **标题** | **类别** | **难度** |
| --- | --- | --- |
| 拦截过滤（Intercepting Filter） | 行为型 | 中等难度 |

**意图**

提供可插拔过滤器进行必要的预处理和对从客户端到目标的请求进行后处理   


**适用模式**

使用拦截过滤模式

* 系统使用预处理或后处理请求
* 系统应对请求进行认证/授权/记录或跟踪，然后将请求传递给相应的处理程序
* 您想要一种模块化的方法来配置预处理和后处理方案

**真实世界**

* [javax.servlet.FilterChain](https://tomcat.apache.org/tomcat-8.0-doc/servletapi/javax/servlet/FilterChain.html) and [javax.servlet.Filter](https://tomcat.apache.org/tomcat-8.0-doc/servletapi/javax/servlet/Filter.html)
* [Struts 2 - Interceptors](https://struts.apache.org/docs/interceptors.html)

**权威信息**

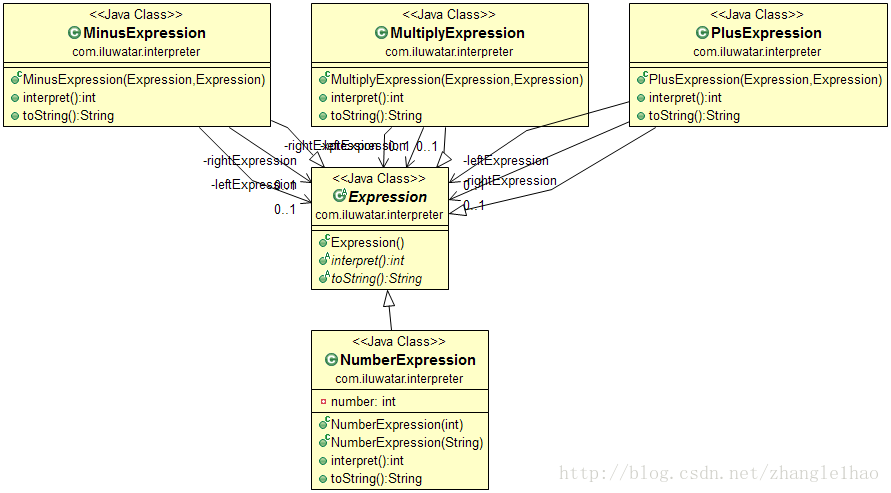
* [TutorialsPoint - Intercepting Filter](http://www.tutorialspoint.com/design_pattern/intercepting_filter_pattern.htm)
* [Presentation Tier Patterns](http://www.javagyan.com/tutorials/corej2eepatterns/presentation-tier-patterns)

**48. 解释器模式**

| **标题** | **类别** | **难度** |
| --- | --- | --- |
| 解释器模式（Interpreter） | 行为型 | GOF,中等难度 |

**意图**

给定一种语言，根据解释器定义其语法的表示。



**适用范围**

当有语言时使用解释器模式解释，你可以用解释器表示语言中的语句作为抽象语法   
树木。 解释者模式如下最有效

* 语法简单。 对于复杂的语法，语法的类层次结构变得很大，难以管理。 在这种情况下，解析器发生器等工具是更好的选择。 他们可以解释表达式而不构建抽象语法树，这样可以节省空间和可能的时间   
  效率并不是关键。 最有效的口译员通常不是通过直接解释解析树来实现的，而是首先将它们转换成另一种形式。 例如，正则表达式通常被转换成状态机。 但是即使如此，翻译者也可以通过解释器模式实现，因此模式仍然适用

**真实世界**

* [java.util.Pattern](http://docs.oracle.com/javase/8/docs/api/java/util/regex/Pattern.html)
* [java.text.Normalizer](http://docs.oracle.com/javase/8/docs/api/java/text/Normalizer.html)
* All subclasses of [java.text.Format](http://docs.oracle.com/javase/8/docs/api/java/text/Format.html)
* [javax.el.ELResolver](http://docs.oracle.com/javaee/7/api/javax/el/ELResolver.html)

**权威信息**

* [Design Patterns: Elements of Reusable Object-Oriented Software](http://www.amazon.com/Design-Patterns-Elements-Reusable-Object-Oriented/dp/0201633612)

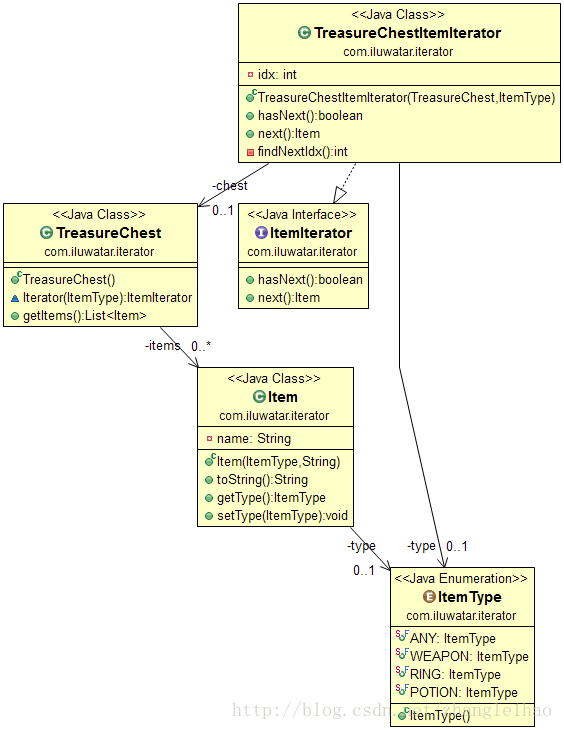
**49. 迭代器模式**

| **标题** | **类别** | **难度** |
| --- | --- | --- |
| 迭代器模式（iterator） | 行为型 | GOF,简单难度 |

**众所周知**

Cursor

**意图**

提供顺序访问聚合对象元素的方法,并且不暴露其底层表现。   


**适用范围**

何时使用迭代器模式

* 访问聚合对象的内容而不暴露其内部表示
* 支持聚合对象的多次遍历
* 提供用于遍历不同聚集结构的统一界面

**真实例子**

* [java.util.Iterator](http://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html)
* [java.util.Enumeration](http://docs.oracle.com/javase/8/docs/api/java/util/Enumeration.html)

**权威信息**

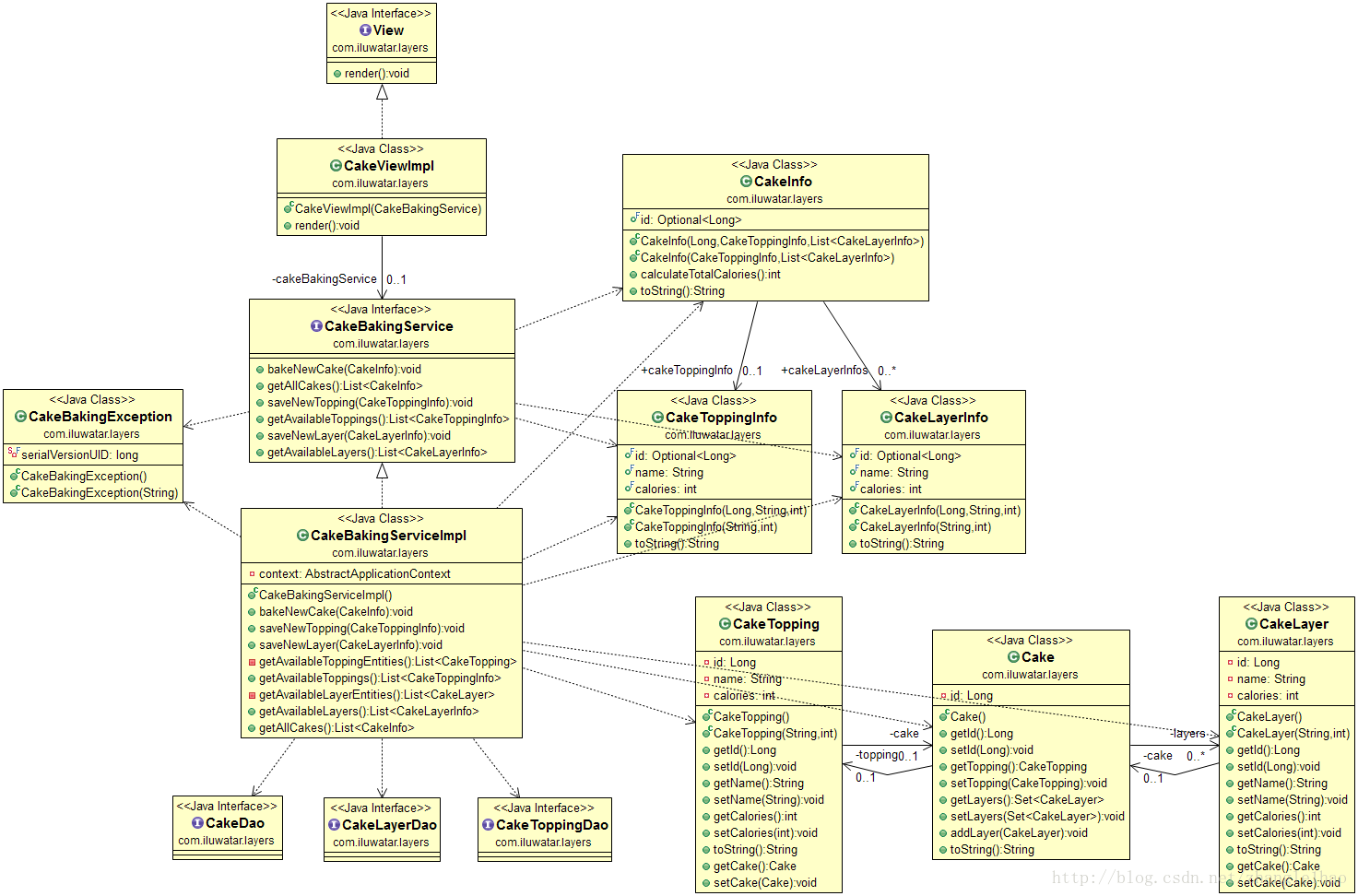
* [Design Patterns: Elements of Reusable Object-Oriented Software](http://www.amazon.com/Design-Patterns-Elements-Reusable-Object-Oriented/dp/0201633612)

**50. 层模式**

| **标题** | **类别** | **难度** |
| --- | --- | --- |
| 层模式（layers） | 结构型 | Spring,中等难度 |

**意图**

层是一种软件的体系结构风格。   
将应用程序划分成几个不同层次。



**适用范围**

使用层架构时

* 您希望将软件职责明确分为程序的不同部分
* 您希望防止更改在整个应用程序中传播
* 您希望使您的应用程序更加可维护和可测试

**权威信息**

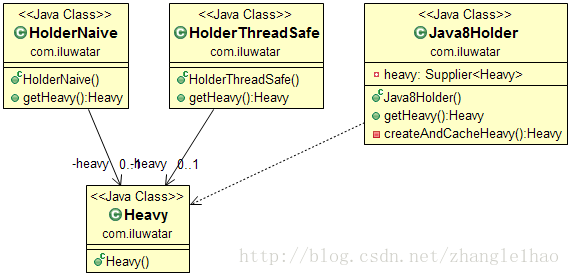
* [Pattern Oriented Software Architecture Vol I-V](http://www.amazon.com/Pattern-Oriented-Software-Architecture-Volume-Patterns/dp/0471958697)

**51. 懒加载**

| **标题** | **类别** | **难度** |
| --- | --- | --- |
| 懒加载(Lazy Loading) | 其他 | 术语,简单难度，性能 |

**意图**

懒加载是一种常见的模式用于延迟初始化一个对象,直到需要的时候。 它可以有助于应用的运作效率适当使用。



**适用范围**

使用Lazy Loading术语

* 加载对象是昂贵的，或者根本不需要加载对象

**真实例子**

* JPA annotations @OneToOne, @OneToMany, @ManyToOne, @ManyToMany and fetch = FetchType.LAZY

**权威信息**

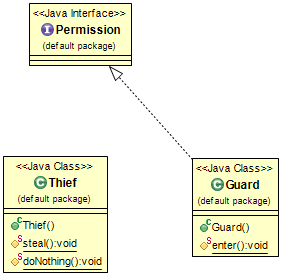
* [J2EE Design Patterns](http://www.amazon.com/J2EE-Design-Patterns-William-Crawford/dp/0596004273/ref=sr_1_2)

**52. 标记接口**

| **标题** | **类别** | **难度** |
| --- | --- | --- |
| 标记接口(Lazy Loading) | 设计 | 简单难度 |

**意图**

使用空接口作为标记来区分处理对象。



**适用范围**

何时使用标记界面图案

* 你想从普通对象中识别特殊对象（以不同方式对待它们）
* 你想标记一些对象可用于某种操作

**真实例子**

* [javase.8.docs.api.java.io.Serializable](https://docs.oracle.com/javase/8/docs/api/java/io/Serializable.html)
* [javase.8.docs.api.java.lang.Cloneable](https://docs.oracle.com/javase/8/docs/api/java/lang/Cloneable.html)

**权威信息**

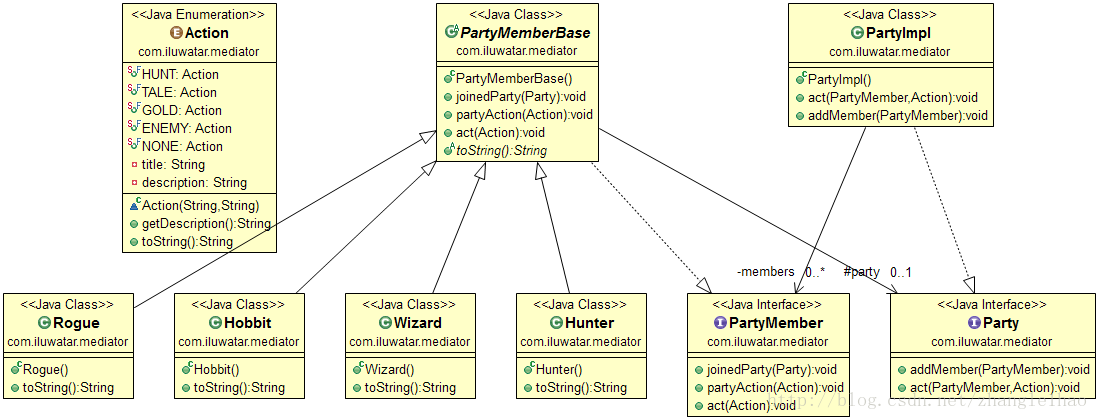
* [Effective Java 2nd Edition by Joshua Bloch](https://www.amazon.com/Effective-Java-2nd-Joshua-Bloch/dp/0321356683)

**53. 中间人模式**

| **标题** | **类别** | **难度** |
| --- | --- | --- |
| 中间人模式(mediator) | 行为型 | GOF,中等难度 |

**意图**

定义一个对象，用来封装一组对象交互。   
中介者通过保持其他对象的引用来促进松耦合，   
它可以让你独立地改变他们的互动。



**适用范围**

使用中介模式

* 一组对象以明确定义但复杂的方式进行通信。 所产生的相互依赖关系是非结构化的，难以理解
* 重复使用对象是困难的，因为它涉及并与许多其他对象进行通信
* 分布在几个类之间的行为应该可以自定义，而不需要很多的子类化

**真实例子**

* All scheduleXXX() methods of [java.util.Timer](http://docs.oracle.com/javase/8/docs/api/java/util/Timer.html)
* [java.util.concurrent.Executor#execute()](http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Executor.html#execute-java.lang.Runnable-)
* submit() and invokeXXX() methods of [java.util.concurrent.ExecutorService](http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ExecutorService.html)
* scheduleXXX() methods of [java.util.concurrent.ScheduledExecutorService](http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ScheduledExecutorService.html)
* [java.lang.reflect.Method#invoke()](http://docs.oracle.com/javase/8/docs/api/java/lang/reflect/Method.html#invoke-java.lang.Object-java.lang.Object...-)

**权威信息**

* [Design Patterns: Elements of Reusable Object-Oriented Software](http://www.amazon.com/Design-Patterns-Elements-Reusable-Object-Oriented/dp/0201633612)

**54. 备忘录模式**

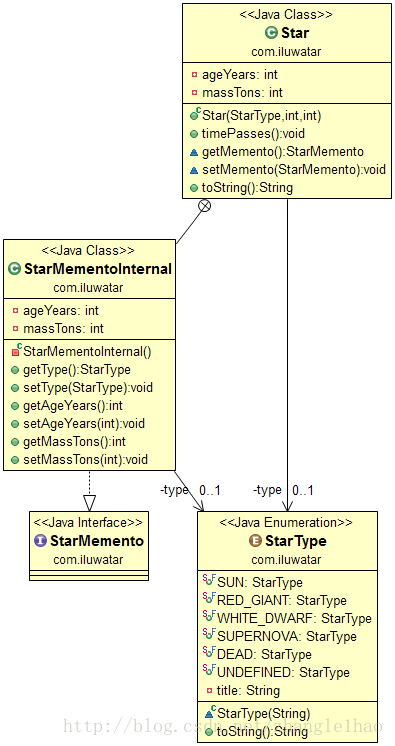
| **标题** | **类别** | **难度** |
| --- | --- | --- |
| 备忘录模式(Memento) | 行为型 | GOF,中等难度 |

**众所周知**

Token

**意图**

不侵犯封装，捕获和外部化   
对象的内部状态，以便稍后可以将对象恢复到此状态。



**适用范围**

当使用备忘者模式时

* 必须保存对象状态的快照，以便稍后可以将其恢复到该状态
* 获取状态的接口将暴露实现细节并破坏对象的封装

**真实世界**

* [java.util.Date](http://docs.oracle.com/javase/8/docs/api/java/util/Date.html)

**权威信息**

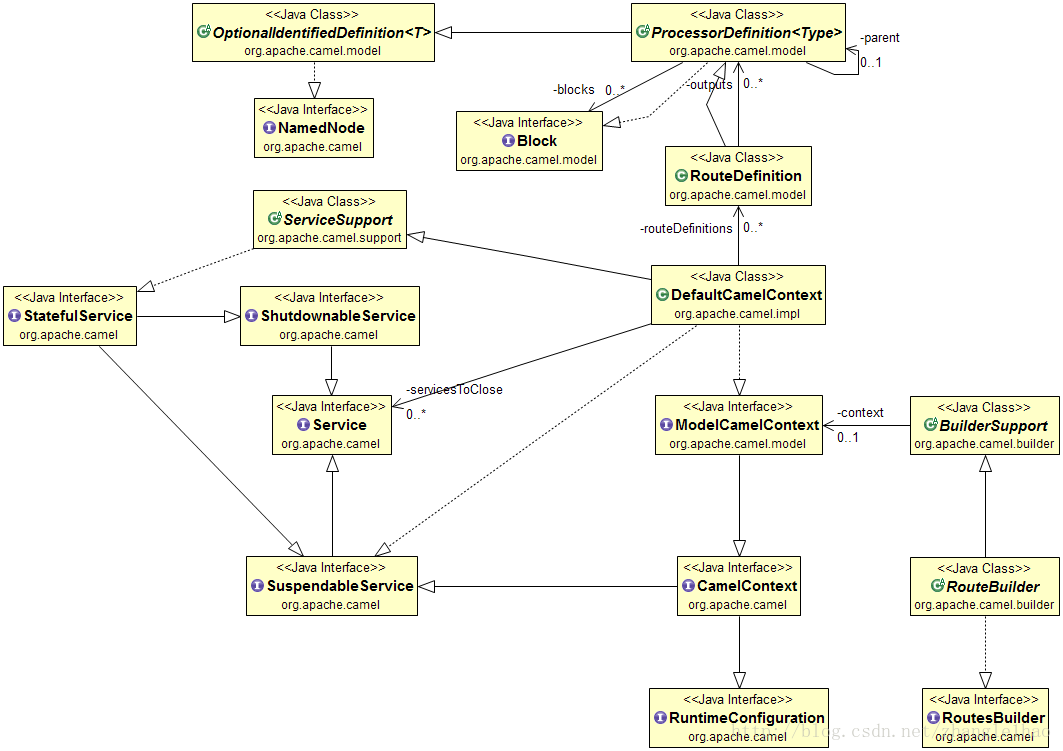
* [Design Patterns: Elements of Reusable Object-Oriented Software](http://www.amazon.com/Design-Patterns-Elements-Reusable-Object-Oriented/dp/0201633612)

**55.消息渠道**

| **标题** | **类别** | **难度** |
| --- | --- | --- |
| 消息渠道(Message Channel) | 交互 | EIP,Apache Camel™ |

**意图**

当两个应用程序使用消息系统进行通信时，他们通过使用逻辑地址进行通信   
的系统，所谓的消息通道。



**适用范围**

使用消息通道模式

* 两个或多个应用程序需要使用消息传递系统进行通信

**真实例子**

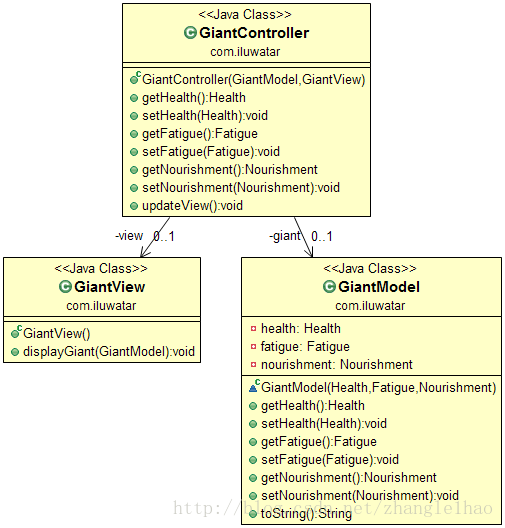
* [akka-camel](http://doc.akka.io/docs/akka/snapshot/scala/camel.html)

**56.模型-视图-控制器**

| **标题** | **类别** | **难度** |
| --- | --- | --- |
| 模型-视图-控制器(Model-View-Controller) | 持久层 | 中等难度 |

**意图**

将用户界面分为三个互连的组件：   
模型，视图和控制器。 让模型管理数据，视图显示数据，控制器调停更新数据并重新绘制   
显示。



**mvc流程图[2]**   


**适用范围**

使用模型 - 视图 - 控制器模式

* 您希望将域数据与其用户界面表示方式分开

**权威信息**

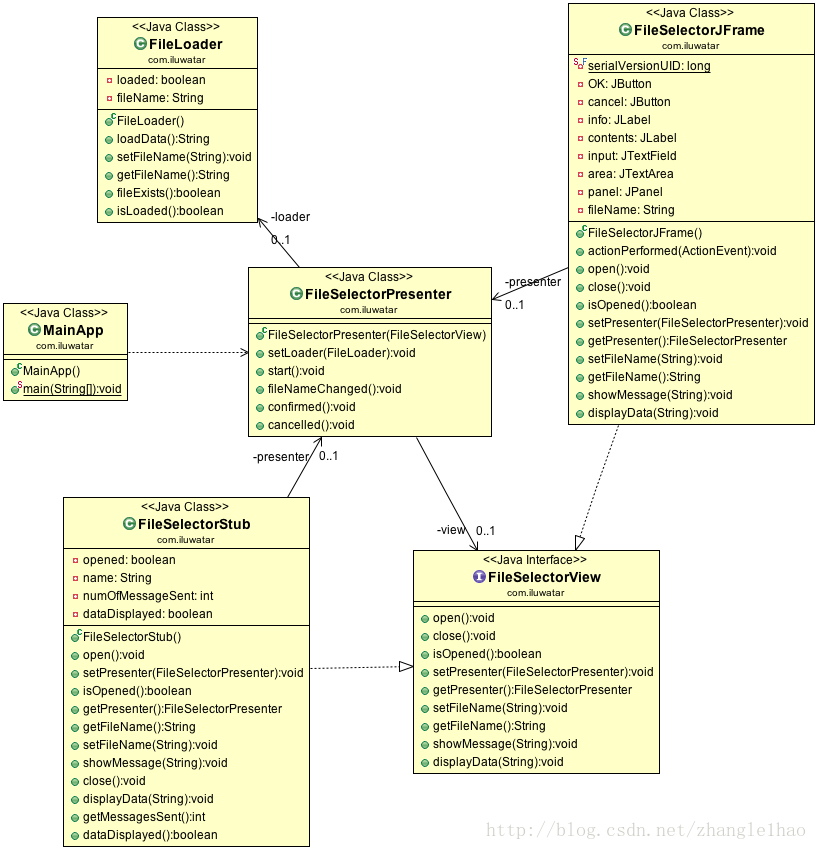
* [Trygve Reenskaug - Model-view-controller](http://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller)
* [J2EE Design Patterns](http://www.amazon.com/J2EE-Design-Patterns-William-Crawford/dp/0596004273/ref=sr_1_2)
* [Patterns of Enterprise Application Architecture](http://www.amazon.com/Patterns-Enterprise-Application-Architecture-Martin/dp/0321127420)

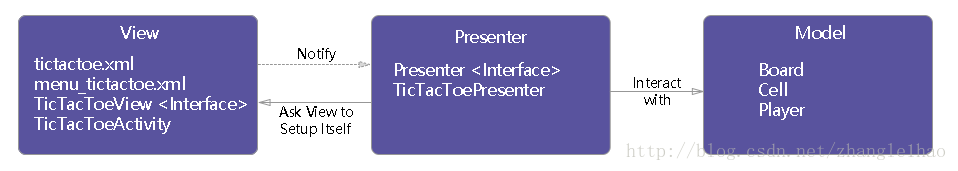
**57.模型-视图-主持人(Model-View-Presenter)**

| **标题** | **类别** | **难度** |
| --- | --- | --- |
| 模型-视图-主持人(Model-View-Presenter) | 持久层 | 中等难度 |

**意图**

应用“关注点分离”原则,允许开发人员构建和测试用户界面。



**mvp[2]流程图**   


**适用范围**

在下列使用MVP情况   
\* 当你想改进表示逻辑中的“分离关注”原则时   
\* 当用户界面开发和测试是必要的。

**真实例子**

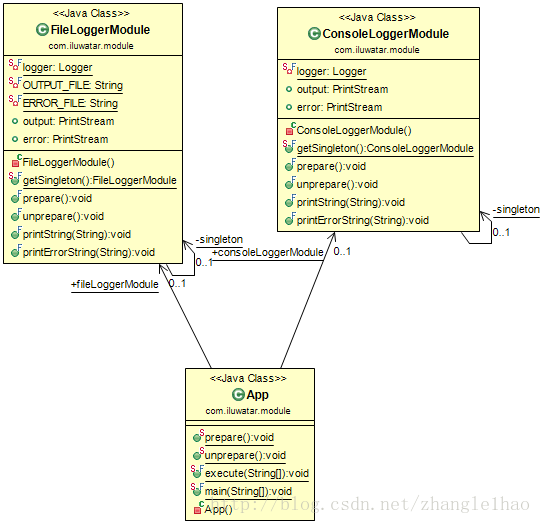
* [MVP4J](https://github.com/amineoualialami/mvp4j)

**58.模块化(Module)**

| **标题** | **类别** | **难度** |
| --- | --- | --- |
| 模块化(Module) | 创建型 | 简单难度 |

**意图**

模块模式用于实现由模块化编程定义的软件模块的概念，在具有不完全直接支持概念的编程语言中。



**适用范围**

模块模式可以被认为是一种创意模式和一种结构模式。 它管理其他元素的创建和组织，并将其组织为结构模式。   
应用此模式的对象可以提供等效的命名空间，为静态类或具有更干净，更简洁的语法和语义的静态成员提供初始化和完成过程。

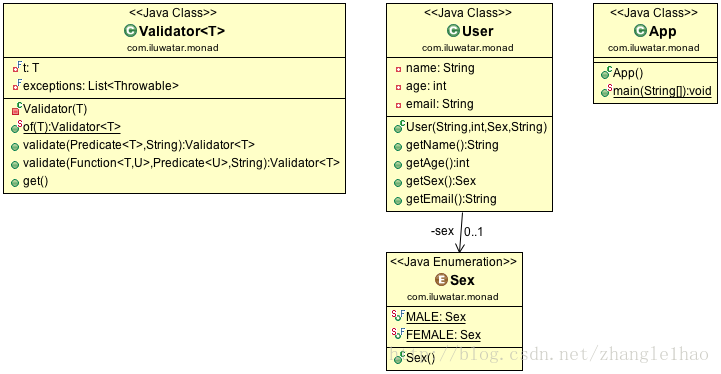
**权威信息**

* [Module](https://en.wikipedia.org/wiki/Module_pattern)

**59.Monad**

| **标题** | **类别** | **难度** |
| --- | --- | --- |
| Monad(这个怎么翻译？) | 其他 | 专家难度,函数式 |

**意图**

基于线性代数的Monad模式表现出一步一步的链接操作的样子。只要保证“同类型”约束,绑定函数可以描述为将其输出传递给另一个输入。 正式地，monad由一个类型构造函数M和两个操作符组成：   
bind - 将monadic对象和一个函数从plain对象转换为monadic值并返回monadic值   
return - 它采用普通类型的对象，并返回包含在一个monadic值中的对象。   
(这方面需要函数式编程方面的知识。。博主只使用过rxjava，有点难描述)   


**适用范围**

在以下任何一种情况下使用Monad

* 当您想要轻松地链接操作时
* 当你想应用每个函数，不管其中任何一个的结果

**权威信息**

* [Design Pattern Reloaded by Remi Forax](https://youtu.be/-k2X7guaArU)
* [Brian Beckman: Don’t fear the Monad](https://channel9.msdn.com/Shows/Going+Deep/Brian-Beckman-Dont-fear-the-Monads)
* [Monad on Wikipedia](https://en.wikipedia.org/wiki/Monad_%28functional_programming%29)

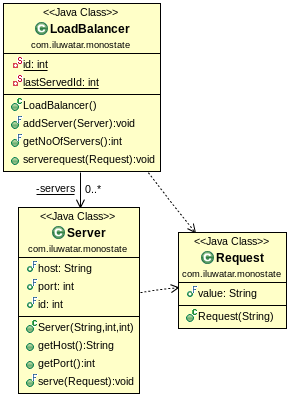
**60.单一的状态(monostate)**

| **标题** | **类别** | **难度** |
| --- | --- | --- |
| 单一的状态(monostate) | 创建型 | 简单难度 |

**众所周知**

Borg

**意图**

执行一个行为，例如在所有实例之间分享相同的状态(这里分享request)   


**适用范围**

使用Monostate模式

* 必须在类的所有实例上共享相同的状态。
* 通常，Monostate模式可能使用任何地方都可以使用Singleton。但是单例模式的使用不是透明的，
* Monostate的用法相比较单例模式的最主要优势。 子类可以根据需要装饰共享状态，因此可以提供与基类动态不同的行为。

**典型用例**

* 日志类
* 管理与数据库的连接
* 文件管理器

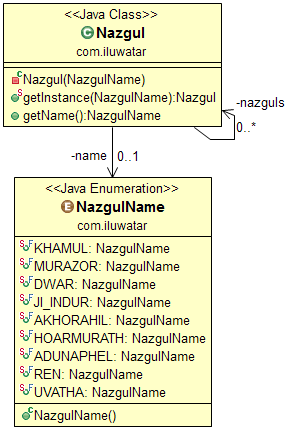
**61.多例模式(Multiton）**

| **标题** | **类别** | **难度** |
| --- | --- | --- |
| 多例模式(Multiton） | 创建型 | 简单难度 |

**众所周知**

Registry

**意图**

确保一个类的实例数量有限，并提供一个全局的接入点。   


**适用范围**

何时使用多例模式

* 必须持有特定数量的类的实例，并且它们可以从一个共同的接入点访问客户端

**62.静音模式(mute）**

| **标题** | **类别** | **难度** |
| --- | --- | --- |
| 静音模式(mute） | 其他 | 简单难度,术语 |

**意图**

提供一个模板来抑制任何被声明但不能发生或只应该被记录的异常;   
同时执行一些业务逻辑。 该模板无需编写重复的try-catch块。

**适用范围**

何时使用这个术语

* 一个API声明一些异常，但永远不会抛出该异常。 ByteArrayOutputStream批量写入方法。
* 您需要通过记录来抑制某些异常，例如关闭资源。

**权威信息**

* [JOOQ: Mute Design Pattern](http://blog.jooq.org/2016/02/18/the-mute-design-pattern/)

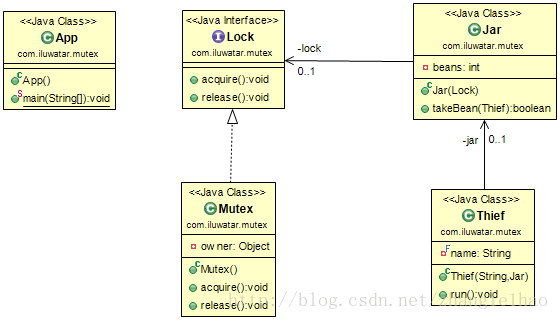
**63.互斥模式**

| **标题** | **类别** | **难度** |
| --- | --- | --- |
| 互斥模式(Mutex） | 并发 | 中等难度 |

**众所周知**

Mutual Exclusion Lock   
Binary Semaphore

**意图**

创建一个在任何一个时刻只允许单个线程访问资源的锁。   


**适用范围**

何时使用互斥

* 您需要同时防止两个线程访问临界区
* 并发访问资源可能导致竞争条件

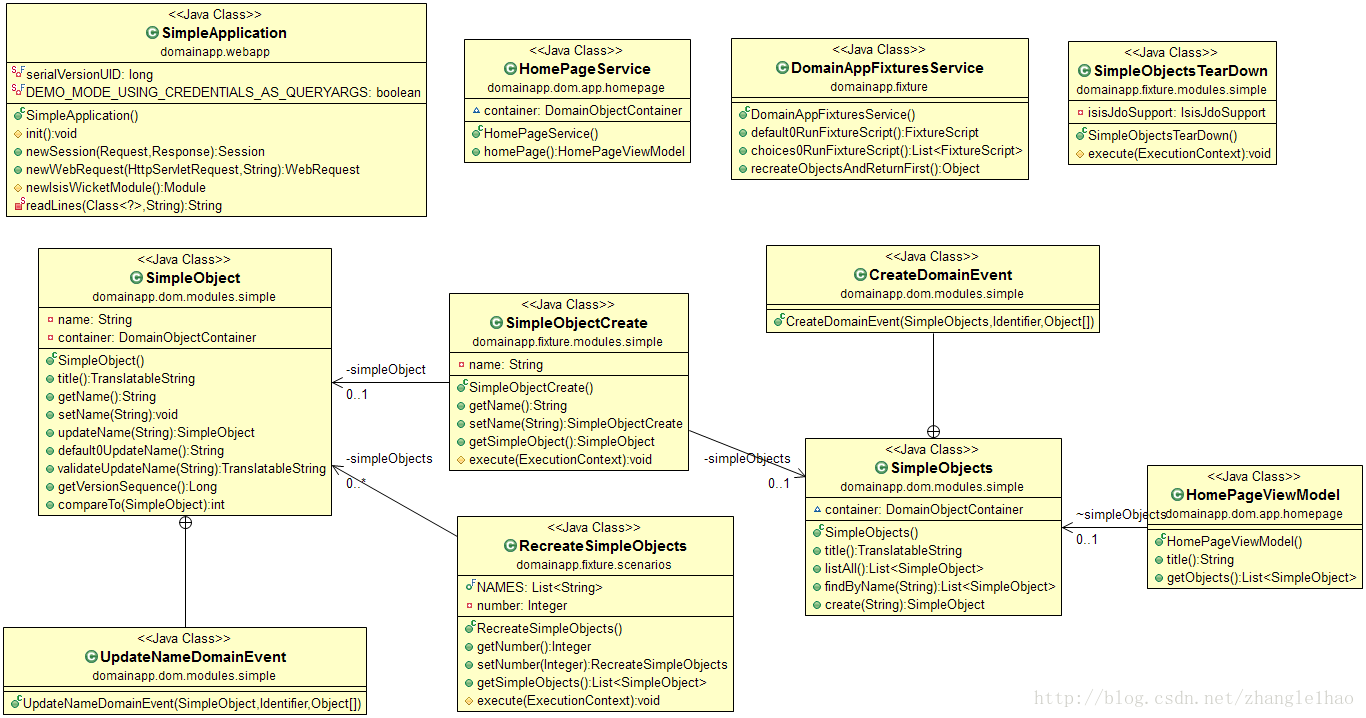
**权威信息**

* [Lock (computer science)] (<http://en.wikipedia.org/wiki/Lock_(computer_science)>)
* [Semaphores] (<http://tutorials.jenkov.com/java-concurrency/semaphores.html>)

**64. 裸露的对象**

| **标题** | **类别** | **难度** |
| --- | --- | --- |
| 裸露的对象(Naked Objects） | 结构型 | 专家难度 |

**意图**

裸露的对象架构模式非常适合快速原型。 使用模式，只需要编写域对象，   
一切都是由框架自动生成的。   


**适用范围**

使用裸体对象模式

* 你是原型，需要快速的开发周期
* 自动生成的用户界面
* 您要自动将域发布为REST服务

**好的例子**

* [Apache Isis](https://isis.apache.org/)

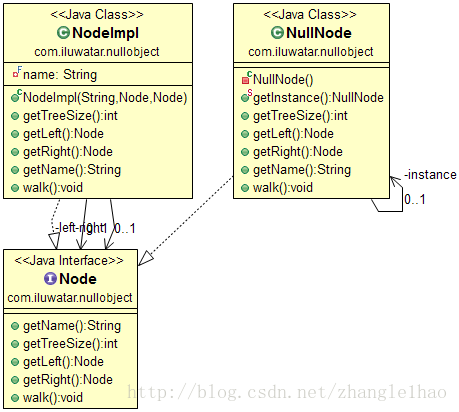
**权威信息**

* [Richard Pawson - Naked Objects](http://downloads.nakedobjects.net/resources/Pawson%20thesis.pdf)

**65.空对象**

| **标题** | **类别** | **难度** |
| --- | --- | --- |
| 空对象（Null Object） | 行为型 | 简单难度 |

**意图**

在大多数面向对象的语言中，如Java或C＃，引用可能为null。 在调用任何方法之前需要检查这些引用以确保它们不为空   
，因为通常无法调用空引用的方法。 而不是使用空引用来表示没有对象（例如，不存在的客户），使用一个对象实现预期的接口，但其方法体是空的。该这种方法优于一个工作的默认实现是一个Null对象是非常可预测的，没有副作用：它什么都不做。   


**适用范围**

何时使用空对象模式

* 你想避免显式的空检查，保持算法的优雅和易读。

**权威信息**

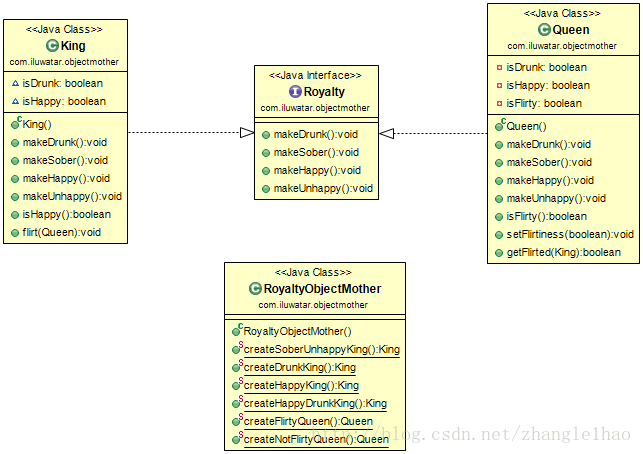
* [Pattern Languages of Program Design](http://www.amazon.com/Pattern-Languages-Program-Design-Coplien/dp/0201607344/ref=sr_1_1)

**66 .母对象（object-mother）**

| **标题** | **类别** | **难度** |
| --- | --- | --- |
| 母对象（object-mother） | 创建型 | 简单难度 |

**意图**

使用分离的构建器和工厂的接口定义不可变的内容。



**适用范围**

何时使用母对象

* 您希望在多个测试中保持一致的对象
* 你想减少在测试中创建对象的代码
* 每个测试都应该运行新的数据

**权威信息**

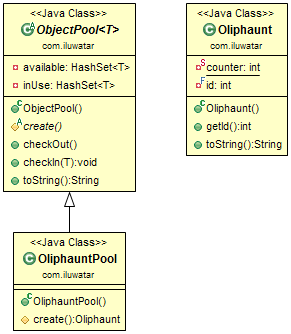
* [Answer by David Brown](http://stackoverflow.com/questions/923319/what-is-an-objectmother) to the stackoverflow question: [What is an ObjectMother?](http://stackoverflow.com/questions/923319/what-is-an-objectmother)
* [c2wiki - Object Mother](http://c2.com/cgi/wiki?ObjectMother)
* [Nat Pryce - Test Data Builders: an alternative to the Object Mother pattern](http://www.natpryce.com/articles/000714.html)

**67. 对象池（object-pool）**

| **标题** | **类别** | **难度** |
| --- | --- | --- |
| 对象池（object-pool） | 创建型 | 简单难度,性能 |

**意图**

当对象创建成本高昂时，并且它只需要在短时间内被使用，这时对象池模式是有用处的。对象池为实例化的对象提供了缓存，跟踪哪些对象正在使用，哪些是可用的。



**适用范围**

使用对象池模式

* 对象创建成本高（分配成本）
* 你需要大量的短命对象（内存碎片）

**68. 观察者模式**

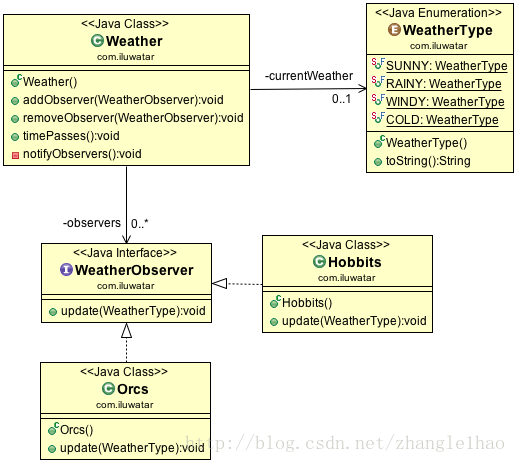
| **标题** | **类别** | **难度** |
| --- | --- | --- |
| Observer（观察者模式） | 创建型 | 简单难度,GOF,响应式 |

**众所周知**

Dependents, Publish-Subscribe

**意图**

定义对象之间的一对多依赖关系, 被观察的对象状态改变，其所有依赖关系将被通知和更新自动。



**适用范围**

在以下情况下使用观察者模式

* 抽象有两个方面，一个取决于另一个方面。 将这些方面封装在单独的对象中，可以独立地改变和重用它们
* 当对一个对象的更改需要更改其他对象时，您不需要知道更改多少个对象
* 当对象应该能够通知其他对象而不对这些对象是谁做出假设的。 换句话说，你不希望这些对象紧密耦合。

**典型用例**

* 改变一个对象会导致其他对象发生变化

**真实世界**

* [java.util.Observer](http://docs.oracle.com/javase/8/docs/api/java/util/Observer.html)
* [java.util.EventListener](http://docs.oracle.com/javase/8/docs/api/java/util/EventListener.html)
* [javax.servlet.http.HttpSessionBindingListener](http://docs.oracle.com/javaee/7/api/javax/servlet/http/HttpSessionBindingListener.html)
* [RxJava](https://github.com/ReactiveX/RxJava)

**权威信息**

* [Design Patterns: Elements of Reusable Object-Oriented Software](http://www.amazon.com/Design-Patterns-Elements-Reusable-Object-Oriented/dp/0201633612)
* [Java Generics and Collections](http://www.amazon.com/Java-Generics-Collections-Maurice-Naftalin/dp/0596527756/)

**69. 页对象**

| **标题** | **类别** | **难度** |
| --- | --- | --- |
| 页对象（page-object） | 测试 | 中等难度 |

**意图**

页面对象封装了UI，隐藏应用程序（通常是Web应用程序）的底层UI小部件，并提供特定于应用程序的API，以允许操作测试所需的UI组件。 在这样做时，它允许测试类本身专注于测试逻辑。



**适用范围**

何时使用页面对象模式

* 您正在为Web应用程序编写自动化测试，并希望将测试所需的UI操作与实际测试逻辑分开。
* 使您的测试不那么脆弱，更可读性更强

**权威信息**

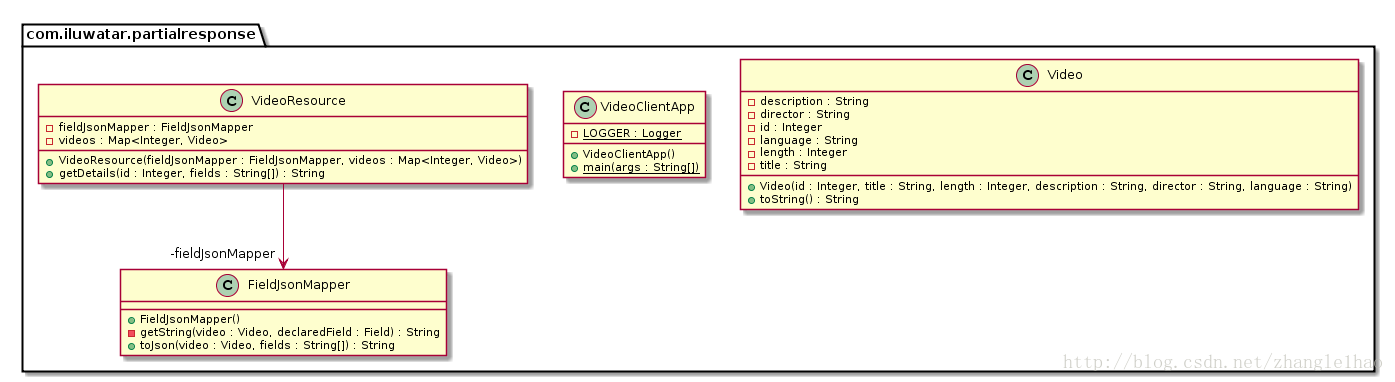
* [Martin Fowler - PageObject](http://martinfowler.com/bliki/PageObject.html)
* [Selenium - Page Objects](https://github.com/SeleniumHQ/selenium/wiki/PageObjects)

**70. 页对象**

| **标题** | **类别** | **难度** |
| --- | --- | --- |
| 页对象（page-object） | 结构型 | 简单难度 |

**意图**

根据需要从服务器发送部分响应。 客户端将指定它需要的字段，而不是要求服务端提供资源的所有细节。



**适用范围**

使用部分响应模式

* 客户端只需要来自资源的数据子集。
* 避免过多的数据传输

**权威信息**

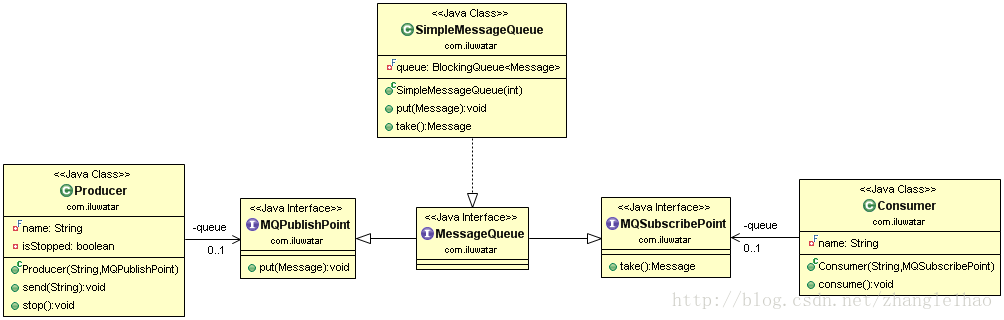
* [Common Design Patterns](https://cloud.google.com/apis/design/design_patterns)
* [Partial Response in RESTful API Design](http://yaoganglian.com/2013/07/01/partial-response/)

**71. poison-pill**

| **标题** | **类别** | **难度** |
| --- | --- | --- |
| poison-pill（不知道怎么翻译）） | 其他 | 中等难度,响应式 |

**意图**

poison-pill允许提供的已知预定义数据项,生产者与消耗者分离,并提供优雅的方式结束进程



**适用范围**

何时使用poison-pil术语

* 需要将信号从一个线程/进程发送到另一个线程 ,并在另一个线程终止

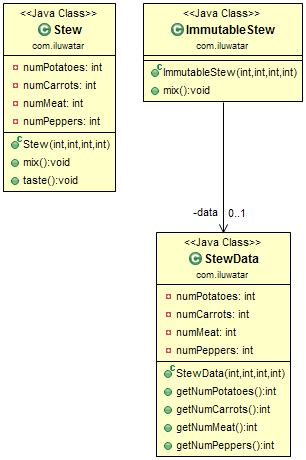
**真实例子**

* [akka.actor.PoisonPill](http://doc.akka.io/docs/akka/2.1.4/java/untyped-actors.html)

**72. 私有类数据**

| **标题** | **类别** | **难度** |
| --- | --- | --- |
| Private Class Data（私有类数据） | 其他 | 简单难度,术语 |

**意图**

私人数据设计模式旨在减少曝光属性通过限制其可见性。 它减少了类的数量   
通过将属性封装在单个Data对象中。   


**适用范围**

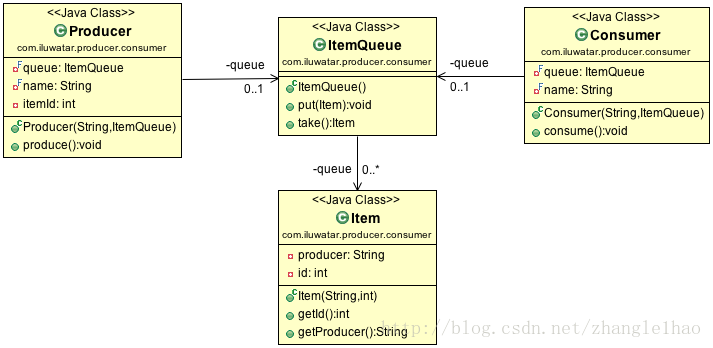
何时使用私有类数据模式

* 你想阻止对类数据成员的写访问

**73. 生产者-消费者**

| **标题** | **类别** | **难度** |
| --- | --- | --- |
| 生产者-消费者（producer-consumer） | 并发 | 中等难度,I/O,响应 |

**意图**

生产者消费者设计模式是一种经典的并发模式，降低了生产者与消费者之间的耦合，通过分离实际的工作执行。   


**适用范围**

何时使用生产者消费者术语

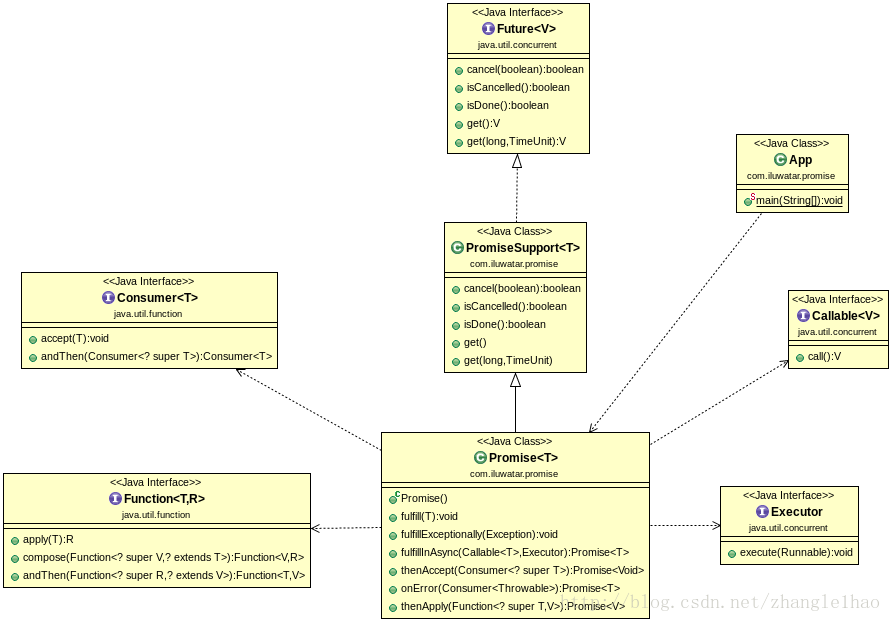
* 通过分离生产和消耗两个过程，达到解耦系统的目的。
* 解决生产工作或消费工作在不同时间要求的问题

**74. 承诺**

| **标题** | **类别** | **难度** |
| --- | --- | --- |
| 承诺（promise） | 并发 | 中等难度,函数,响应 |

**意图**

承诺模式代表一个值但事先却不需要知道它。允许你通过承诺模式关联一个异步动作的最终成功值或失败的原因。提供了一种异步访问的方式，但表现出同步执行的样子



**适用范围**

某些工作需要异步完成时，承诺模式适用于并行编程   
和：

* 代码可维护性和可读性受到回调地狱的困扰。
* 您需要编写承诺，并需要更好的错误处理异步任务。
* 你想使用函数式的编程风格。

**真实例子**

* [java.util.concurrent.CompletableFuture](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html)
* [Guava ListenableFuture](https://github.com/google/guava/wiki/ListenableFutureExplained)

**关联模式**

* Async Method Invocation
* Callback

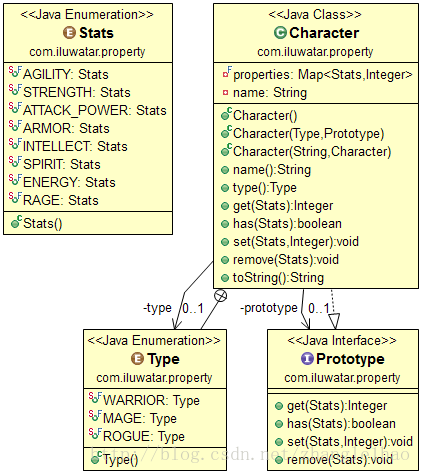
**权威信息**

* [You are missing the point to Promises](https://gist.github.com/domenic/3889970)
* [Functional style callbacks using CompletableFuture](https://www.infoq.com/articles/Functional-Style-Callbacks-Using-CompletableFuture)

**75. 属性**

| **标题** | **类别** | **难度** |
| --- | --- | --- |
| 属性（property） | 创建型 | 简单 |

**意图**

使用已经存在的对象（当做父类）创建对象层级和新的对象   


**适用范围**

何时使用属性模式

* 当你想动态的创建对象的属性和原型继承对象

**真实例子**

* [JavaScript](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Inheritance_and_the_prototype_chain) prototype inheritance

**76. 原型**

| **标题** | **类别** | **难度** |
| --- | --- | --- |
| 原型（prototype） | 创建型 | GOF,简单 |

**意图**

使用原型指定要创建的对象的种类实例，并通过复制此原型创建新对象。

**解释**

真实世界

还记得多莉吗 被克隆的羊！ 让我们不了解细节，但这里的关键在于它是如何克隆的。

字面意思

通过克隆创建基于现有对象的对象。

维基解释

原型图是软件开发中的创建型设计模式。 当要创建的对象的类型由原型实例确定时，它将被克隆以生成新对象。

简而言之，它允许您创建现有对象的副本并根据需要进行修改，而不是从头开始创建对象并进行设置。   
**编程例子**

在Java中，可以通过从Object实现Cloneable并覆盖clone来轻松实现

class Sheep implements Cloneable {

private String name;

public Sheep(String name) { this.name = name; }

public void setName(String name) { this.name = name; }

public String getName() { return name; }

@Override

public Sheep clone() throws CloneNotSupportedException {

return new Sheep(name);

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10

然后可以像下面那样进行克隆

Sheep original = new Sheep("Jolly")*;*

System.out.println(original.getName())*; // Jolly*

Sheep cloned = original.clone()*;*

cloned.setName("Dolly")*;*

System.out.println(cloned.getName())*; // Dolly*

* 1
* 2
* 3
* 4
* 5
* 6
* 7

**适用范围**

当系统应独立于其产品的创建，组成和代表的方式时，使用原型模式; 和

* 当运行时指定要实例化的类时，例如，通过动态加载
* 避免建立与产品类层次结构相似的工厂的类层次结构
* 当一个类的实例可以只有几个不同的状态组合中的一个时。 安装相应数量的原型并克隆它们可能更方便，而不是手动实例化类，每次都有适当的状态
* 与克隆相比，创建对象比较昂贵

**真实例子**

* [java.lang.Object#clone()](http://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#clone%28%29)

**权威信息**

* [Design Patterns: Elements of Reusable Object-Oriented Software](http://www.amazon.com/Design-Patterns-Elements-Reusable-Object-Oriented/dp/0201633612)

**77.代理**

| **标题** | **类别** | **难度** |
| --- | --- | --- |
| proxy（代理） | 结构型 | GOF,简单 |

**意图**

为另一个对象提供代理或占位符来控制访问它

**解释**

真实例子

想象一个法术塔,允许当地的巫师去研究他们的法术。 象牙塔只能通过代理访问，确保只有前三个巫师才能进入。 这里代理代表了塔的功能，并添加了访问控制。

字面意思

使用代理模式，一个类代表另一个类的功能。

维基描述

代理，最通用的形式是一个类作为其他东西的接口。 代理是由客户端调用来访问幕后真实服务对象的包装器或代理对象。 使用代理可以简单地转发到真实的对象，或者可以提供额外的逻辑。 在代理中，可以提供额外的功能，例如当真实对象上的操作是资源密集型时的缓存，或者在调用真实对象的操作之前检查前提条件。

**编程例子**

从上面我们的网页上股塔示例。 首先我们有巫师塔接口和象牙塔类

public interface WizardTower {

void enter(Wizard wizard);

}

public class IvoryTower implements WizardTower {

private static final Logger LOGGER = LoggerFactory.getLogger(IvoryTower.class);

public void enter(Wizard wizard) {

LOGGER.info("{} enters the tower.", wizard);

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14

简单的巫师类

public class Wizard {

private final String name;

public Wizard(String name) {

this.name = name;

}

@Override

public String toString() {

return name;

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13

我们通过代理控制巫师塔的进入

public class WizardTowerProxy implements WizardTower {

private static final Logger LOGGER = LoggerFactory.getLogger(WizardTowerProxy.class);

private static final int NUM\_WIZARDS\_ALLOWED = 3;

private int numWizards;

private final WizardTower tower;

public WizardTowerProxy(WizardTower tower) {

this.tower = tower;

}

@Override

public void enter(Wizard wizard) {

if (numWizards < NUM\_WIZARDS\_ALLOWED) {

tower.enter(wizard);

numWizards++;

} else {

LOGGER.info("{} is not allowed to enter!", wizard);

}

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24

这里是塔进入场景

WizardTowerProxy proxy = new WizardTowerProxy(new IvoryTower());

proxy.enter(new Wizard("Red wizard")); *// 进去.*

proxy.enter(new Wizard("White wizard")); *//进去.*

proxy.enter(new Wizard("Black wizard")); *// 进去.*

proxy.enter(new Wizard("Green wizard")); *// 进不去!*

proxy.enter(new Wizard("Brown wizard")); *// 进不去!*

* 1
* 2
* 3
* 4
* 5
* 6

**适用范围**

通过代理模式来使用更多变化或复杂的对象。 这里   
是代理模式适用的几种常见情况

* 远程代理为不同地址空间中的对象提供本地代理。
* 虚拟代理根据需要创建昂贵的对象。
* 保护代理控制对原始对象的访问。 当对象具有不同的访问权限时，保护代理是有用的。

**典型用例**

* 控制对另一个对象的访问
* 懒加载
* 实施日志记录
* 方便网络连接
* 计数对象的引用

**目录**

* [Controlling Access With Proxy Pattern](http://java-design-patterns.com/blog/controlling-access-with-proxy-pattern/)

**演讲**

* [Proxy](https://github.com/iluwatar/java-design-patterns/tree/master/proxy/etc/presentation.html)

**真实世界**

* [java.lang.reflect.Proxy](http://docs.oracle.com/javase/8/docs/api/java/lang/reflect/Proxy.html)
* [Apache Commons Proxy](https://commons.apache.org/proper/commons-proxy/)
* Mocking frameworks Mockito, Powermock, EasyMock

**权威信息**

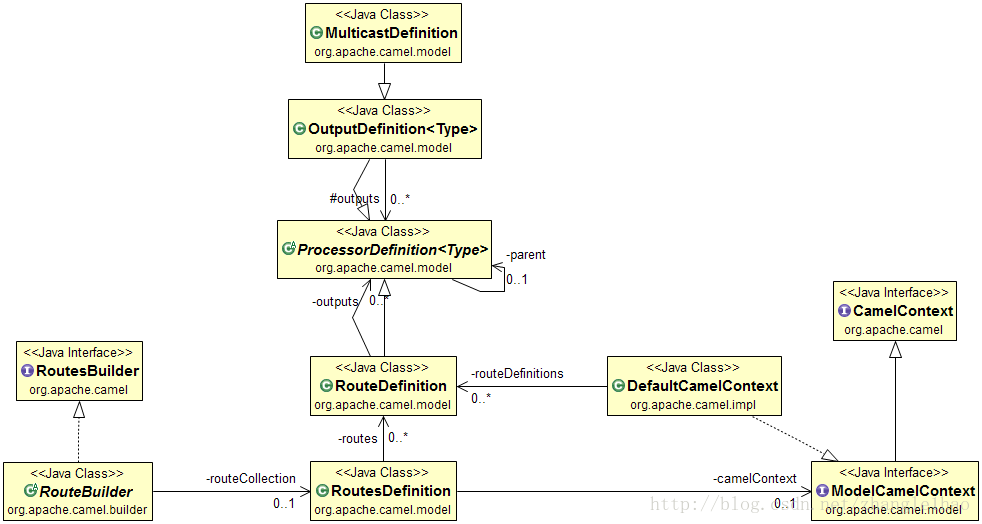
* [Design Patterns: Elements of Reusable Object-Oriented Software](http://www.amazon.com/Design-Patterns-Elements-Reusable-Object-Oriented/dp/0201633612)

**78.发布-订阅**

| **标题** | **类别** | **难度** |
| --- | --- | --- |
| 发布-订阅（publish-subscribe） | 结构型 | EIP,Apache Camel™ |

**意图**

发送者发送广播信息至对相关信息感兴趣的接收者



**适合范围**

何时使用发布订阅频道模式

* 两个或多个应用程序需要使用消息系统进行通信以进行广播。

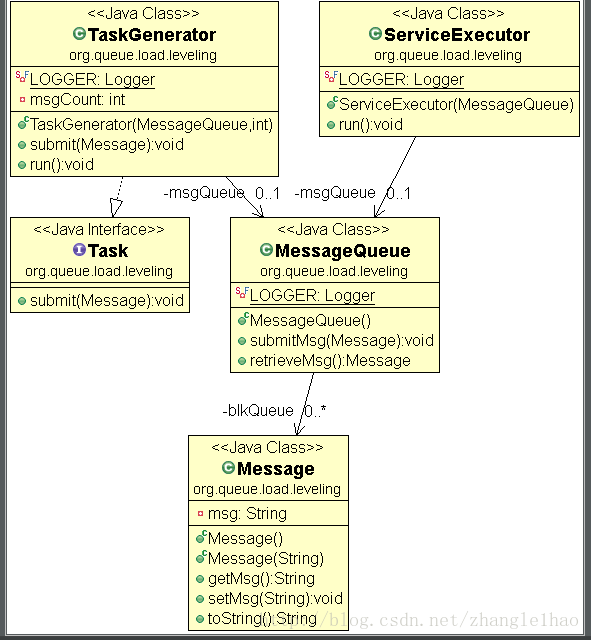
**权威信息**

* [J2EE Design Patterns](http://www.amazon.com/J2EE-Design-Patterns-William-Crawford/dp/0596004273/ref=sr_1_2)

**79.基于队列的负载均衡**

| **标题** | **类别** | **难度** |
| --- | --- | --- |
| 基于队列的负载均衡（queue-load-leveling） | 其他 | 中等难度,性能 |

**意图**

使用一个队列，作为一个任务和它调用的服务之间的缓冲区，以便顺利进行可能导致服务失败或任务超时的间歇性重负载。这种模式可以帮助最大限度地减少需求峰值对可用性和响应能力的影响   
用于任务和服务。   


**适用范围**

* 此模式非常适合任何使用可能会重载的服务的应用程序。
* 如果应用程序期望来自服务的响应以最小延迟，则此模式可能不合适。

**目录**

* [Queue-Based Load Leveling Pattern](http://java-design-patterns.com/blog/queue-load-leveling/)

**真实例子**

* Microsoft Azure Web角色通过使用单独的存储服务来存储数据。 如果Web角色的大量实例并发运行，则存储服务可能会被淹没，并且无法快速响应请求，以防止这些请求超时或失败。

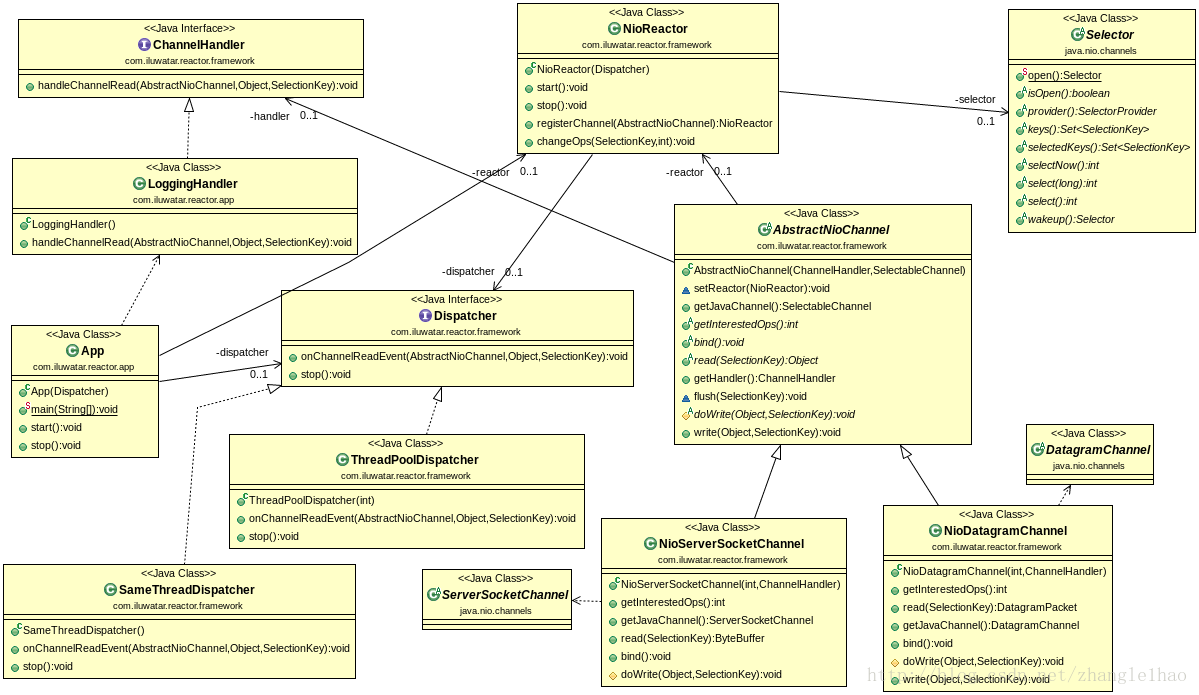
**权威信息**

* [Microsoft Cloud Design Patterns: Queue-Based Load Leveling Pattern](https://msdn.microsoft.com/en-us/library/dn589783.aspx)

**80.响应器**

| **标题** | **类别** | **难度** |
| --- | --- | --- |
| 响应器（reactor） | 并发 | 专家难度,I/O |

**意图**

反应器设计模式处理由一个或多个客户端同时传递到应用程序的服务请求。 应用程序可以注册特定的事件处理器来处理特定事件。 反应器持有事件分发器，并分发事件给特定的事件处理器。   


**适用范围**

使用反应器模式

* 服务器应用程序需要处理来自多个客户端的并发服务请求。
* 即使处理较旧的客户端请求，服务器应用程序需要可用于从新客户端接收请求。
* 服务器必须最大限度地提高吞吐量，最大限度地减少延迟并有效利用CPU而无需阻塞

**真实例子**

* [Spring Reactor](http://projectreactor.io/)

**权威信息**

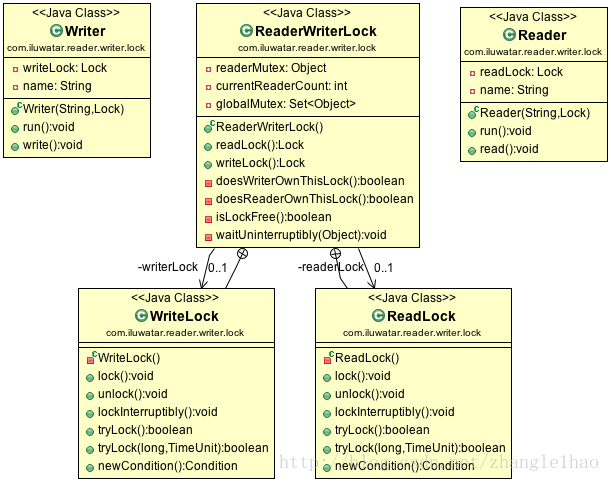
* [Douglas C. Schmidt - Reactor](https://www.dre.vanderbilt.edu/~schmidt/PDF/Reactor.pdf)
* [Pattern Oriented Software Architecture Vol I-V](http://www.amazon.com/Pattern-Oriented-Software-Architecture-Volume-Patterns/dp/0471958697)
* [Doug Lea - Scalable IO in Java](http://gee.cs.oswego.edu/dl/cpjslides/nio.pdf)
* [Netty](http://netty.io/)

**81.读写锁**

| **标题** | **类别** | **难度** |
| --- | --- | --- |
| 响应器（reactor） | 并发 | 中等难度,性能 |

**意图**

假设我们有一个具有上面详细描述的基本约束的共享内存区域。 可以保护互斥互斥体之后的共享数据，在这种情况下，两个线程不能同时访问数据。 但是，这个解决方案不是最佳的，因为读取器R1可能有锁，然后另一个读取器R2请求访问。 在开始自己的阅读操作之前，R2等待R1完成是愚蠢的; 相反，R2应该立即开始。 这是Reader Writer Lock模式的动机。



**适用范围**

应用程序需要增加多线程的资源同步性能，特别是有混合的读/写操作。

**真实例子**

* [Java Reader Writer Lock](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/ReadWriteLock.html)

**权威信息**

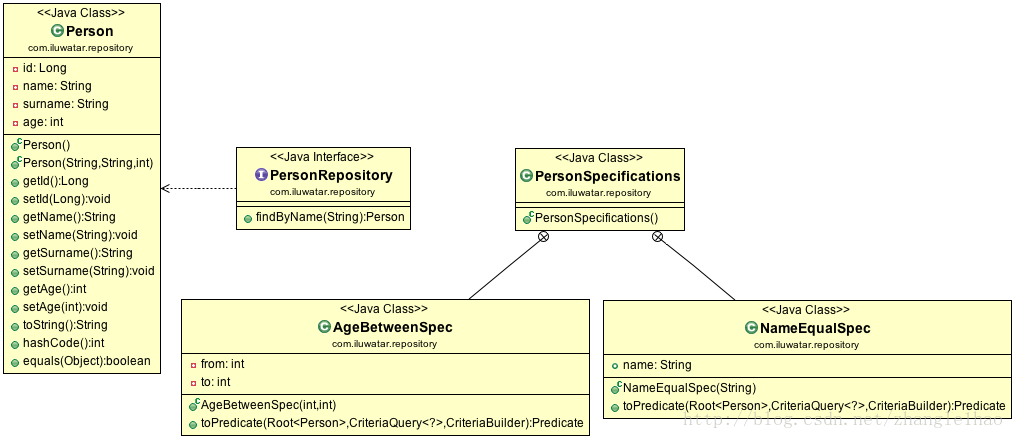
* [Readers–writer lock](https://en.wikipedia.org/wiki/Readers%E2%80%93writer_lock)
* [Readers–writers\_problem](https://en.wikipedia.org/wiki/Readers%E2%80%93writers_problem)

**82.仓库**

| **标题** | **类别** | **难度** |
| --- | --- | --- |
| 仓库（repository） | 持久层 | 中等难度,spring |

**意图**

在域和数据映射之间添加存储库层层隔离域对象与数据库访问代码的细节   
以最小化查询代码的散布和重复。 存储库模式是特别适用于域类大或重的系统   
利用查询。



**适用范围**

何时使用Repository模式

* 域对象数量很大
* 你想避免重复查询代码
* 您希望将数据库查询代码保存在一个位置
* 您有多个数据源

\*

**真实例子**

* [Spring Data](http://projects.spring.io/spring-data/)

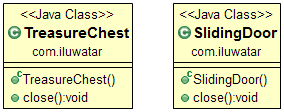
**权威信息**

* [Don’t use DAO, use Repository](http://thinkinginobjects.com/2012/08/26/dont-use-dao-use-repository/)
* [Advanced Spring Data JPA - Specifications and Querydsl](https://spring.io/blog/2011/04/26/advanced-spring-data-jpa-specifications-and-querydsl/)

**83.资源获取初始化**

| **标题** | **类别** | **难度** |
| --- | --- | --- |
| 资源获取初始化（resource-acquisition-is-initialization） | 其他 | 简单难度,术语 |

**意图**

资源获取初始化模式可用于实现资源异常安全管理。   


**解释**

**编程例子**

public class App {

private static final Logger LOGGER = LoggerFactory.getLogger(App.class);

public static void main(String[] args) throws Exception {

try (SlidingDoor slidingDoor = new SlidingDoor()) {

LOGGER.info("Walking in.");

}

try (TreasureChest treasureChest = new TreasureChest()) {

LOGGER.info("Looting contents.");

}

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16

**适用范围**

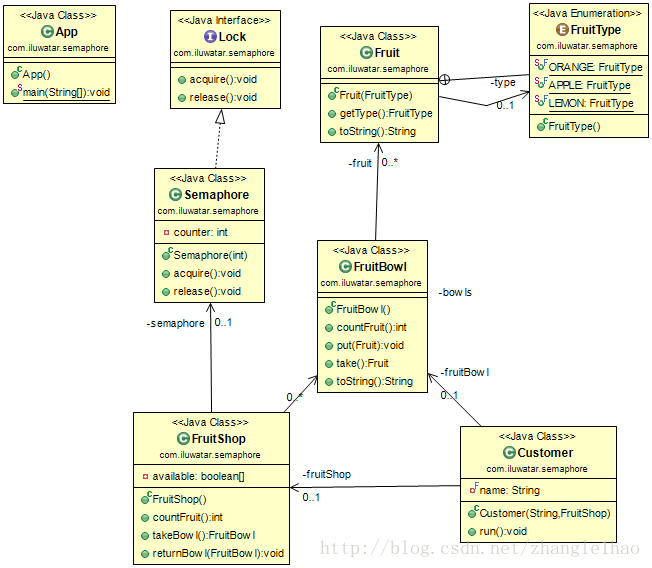
用资源获取初始化模式

* 您的资源必须在每个条件下关闭

**84.信号**

| **标题** | **类别** | **难度** |
| --- | --- | --- |
| 信号（semaphore） | 并发 | 中等难度 |

**意图**

创建一个中介对资源池的访问的锁。在有限数量的线程条件下，在创建时指定一定数量的   
信号量，可以在任何给定的时间访问资源。一个只允许一个并发访问资源的信号量   
被称为二进制信号量。   


**适用范围**

何时使用信号量时

* 你有一个资源池来分配给不同的线程
* 并发访问资源可能导致竞争条件

**权威信息**

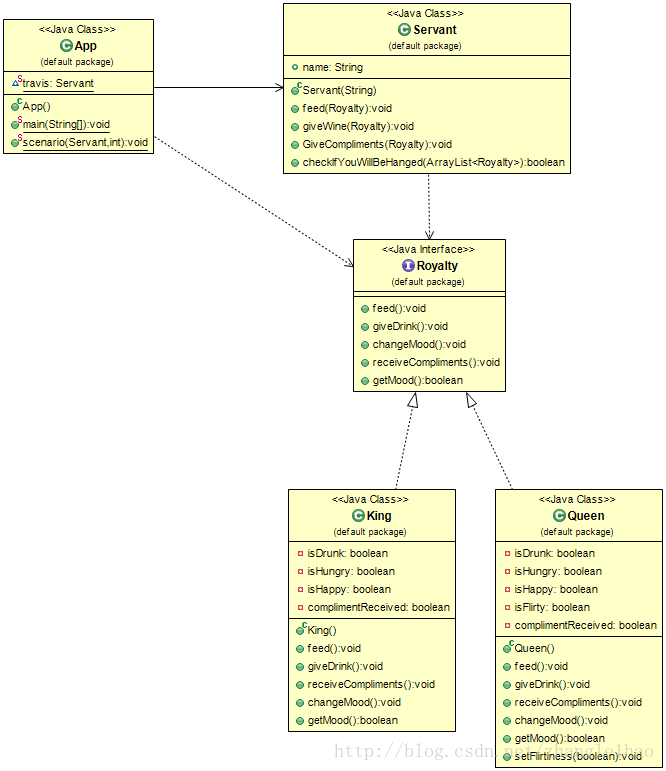
* [Semaphore(programming)] (<http://en.wikipedia.org/wiki/Semaphore_(programming)>)
* [Semaphores] (<http://tutorials.jenkov.com/java-concurrency/semaphores.html>)

**85.仆人**

| **标题** | **类别** | **难度** |
| --- | --- | --- |
| 仆人（servant） | 结构型 | 简单难度 |

**意图**

仆人用于为一组类提供一些行为。而不是在每个类中定义该行为 - 或者当我们无法确定   
在普通父类中使用这种行为 ， 它在仆人中定义一次。



**适用范围**

时使用仆人模式

* 当我们希望一些对象执行一个常见的动作，并且不希望将这个动作定义为每个类中的一个方法。

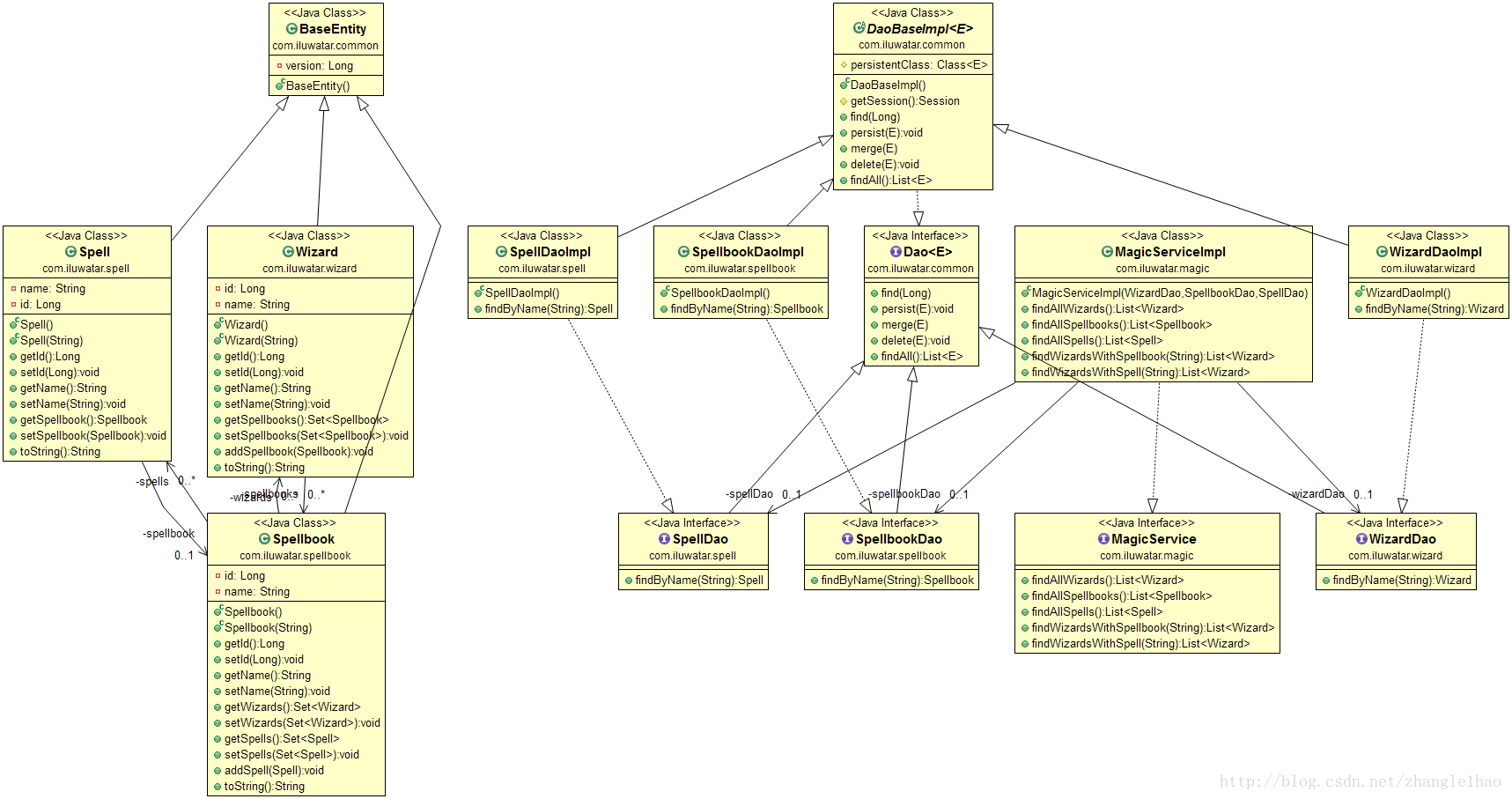
**权威信息**

* [Let’s Modify the Objects-First Approach into Design-Patterns-First](http://edu.pecinovsky.cz/papers/2006_ITiCSE_Design_Patterns_First.pdf)

**86.服务层**

| **标题** | **类别** | **难度** |
| --- | --- | --- |
| 服务层（service-layer） | 结构型 | 中等难度 |

**意图**

服务层是域逻辑的抽象。通常应用程序需要多种接口来存储和存储数据   
实现逻辑：数据加载器，用户界面，集成网关和其他。 尽管它们的目的不同，但是这些接口通常需要通用   
与应用程序的交互访问和操纵其数据并调用其业务逻辑。 服务层实现了这一角色。   


**适用范围**

何时使用服务层模式

* 您希望在API下封装域逻辑
* 您需要实现具有通用逻辑和数据的多个接口

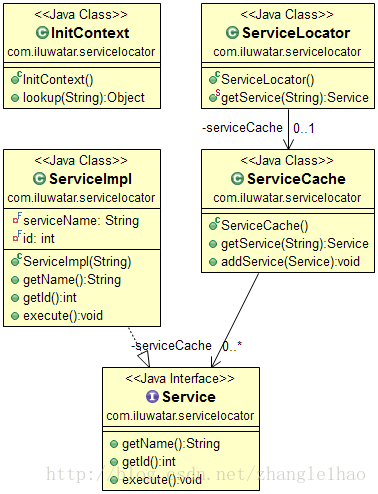
**权威信息**

* [Martin Fowler - Service Layer](http://martinfowler.com/eaaCatalog/serviceLayer.html)
* [Patterns of Enterprise Application Architecture](http://www.amazon.com/Patterns-Enterprise-Application-Architecture-Martin/dp/0321127420)

**87.服务定位**

| **标题** | **类别** | **难度** |
| --- | --- | --- |
| 服务定位（service-locator） | 结构型 | 简单难度,性能 |

**意图**

封装服务层的获取执行过程。   


**适用范围**

服务[定位器](https://www.baidu.com/s?wd=%E5%AE%9A%E4%BD%8D%E5%99%A8&tn=24004469_oem_dg&rsv_dl=gh_pl_sl_csd)模式适用于任何时候使用通常是冗余的JNDI来定位/获取各种服务   
和昂贵的查询。 服务定位器模式解决了这个昂贵的问题通过使用缓存技术来查找。 为了第一次   
请求特定服务，服务定位器在JNDI中查找，取出相关服务，然后最终缓存此服务对象。 现在进一步   
通过服务定位器查找相同的服务在其缓存中完成在很大程度上提高了应用的性能。

**典型用例**

* 当网络点击费用高昂且耗时时
* 查询服务相当频繁
* 正在使用大量的服务

**结论**

* 违反了接口隔离原则，可能像客户端提供了他们不需要的服务
* 在运行时创建服务可能导致客户端崩溃

**权威信息**

* [J2EE Design Patterns](http://www.amazon.com/J2EE-Design-Patterns-William-Crawford/dp/0596004273/ref=sr_1_2)

**89.单例**

| **标题** | **类别** | **难度** |
| --- | --- | --- |
| 单例（singleton） | 结构型 | 简单难度,性能 |

**意图**

确保一个类只有一个实例，并提供一个全局点访问它

**解释**

真实例子

只有一个象牙塔，巫师研究他们的魔法。 巫师一直使用同样的魔法象牙塔。 这里的象牙塔是单例。

字面意思

确保只创建一个特定类的一个对象。

维基解释

在软件工程中，单例模式是将类的实例化限制为一个对象的软件设计模式。 当需要一个对象来协调整个系统的动作时，这是非常有用的。

**编程例子**   
Joshua Bloch，Effective Java 2nd Edition第18页

单元素枚举类型是实现单例的最佳方法

public enum EnumIvoryTower {

INSTANCE;

@Override

public String toString() {

return getDeclaringClass().getCanonicalName() + "@" + hashCode();

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10

使用

EnumIvoryTower enumIvoryTower1 = EnumIvoryTower.INSTANCE;

EnumIvoryTower enumIvoryTower2 = EnumIvoryTower.INSTANCE;

assertEquals(enumIvoryTower1, enumIvoryTower2); // true

* 1
* 2
* 3

不安全单例

public final class IvoryTower {

private IvoryTower() {}

private static final IvoryTower INSTANCE = new IvoryTower();

public static IvoryTower getInstance() {

return INSTANCE;

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12

安全懒加载单例

public final class ThreadSafeLazyLoadedIvoryTower {

private static ThreadSafeLazyLoadedIvoryTower instance;

private ThreadSafeLazyLoadedIvoryTower() {

*// to prevent instantiating by Reflection call*

if (instance != null) {

throw new IllegalStateException("Already initialized.");

}

}

public static synchronized ThreadSafeLazyLoadedIvoryTower getInstance() {

if (instance == null) {

instance = new ThreadSafeLazyLoadedIvoryTower();

}

return instance;

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21

双重检查锁单例

public final class ThreadSafeDoubleCheckLocking {

private static volatile ThreadSafeDoubleCheckLocking instance;

private ThreadSafeDoubleCheckLocking() {

*// to prevent instantiating by Reflection call*

if (instance != null) {

throw new IllegalStateException("Already initialized.");

}

}

public static ThreadSafeDoubleCheckLocking getInstance() {

ThreadSafeDoubleCheckLocking result = instance;

*//检查单例实例是否已初始化。 如果它被初始化，那么我们可以返回实例。*

if (result == null) {

*//它没有初始化，但是我们不能确定，因为一些其他线程可能已初始化它*

*// 同时。 所以要确保我们需要锁定一个对象以获得互斥。*

synchronized (ThreadSafeDoubleCheckLocking.class) {

*//再次将实例分配给局部变量，以检查它是否被某个其他线程初始化*

*//当前线程被阻止进入锁定区域。 如果它被初始化，那么我们可以*

*//返回之前创建的实例，就像以前的空检查一样。*

result = instance;

if (result == null) {

*//实例还没有被初始化，所以我们可以安全地（没有其他线程可以进入这个区域）*

*//创建一个实例，使其成为我们的单例实例。*

instance = result = new ThreadSafeDoubleCheckLocking();

}

}

}

return result;

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36
* 37

内部静态类成员持有实例

public final class InitializingOnDemandHolderIdiom {

private InitializingOnDemandHolderIdiom() {}

public static InitializingOnDemandHolderIdiom getInstance() {

return HelperHolder.INSTANCE;

}

private static class HelperHolder {

private static final InitializingOnDemandHolderIdiom INSTANCE =

new InitializingOnDemandHolderIdiom();

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16

**适用范围**

何时Singleton模式

* 必须只有一个类的一个实例，并且它必须可以从众所周知的接入点访问客户端
* 当唯一的实例应该通过子类化可扩展时，客户端应该能够使用扩展实例而不修改它们的代码

**典型用例**

* 记录类
* 管理与数据库的连接
* 文件管理器

**真实例子**

* [java.lang.Runtime#getRuntime()](http://docs.oracle.com/javase/8/docs/api/java/lang/Runtime.html#getRuntime%28%29)
* [java.awt.Desktop#getDesktop()](http://docs.oracle.com/javase/8/docs/api/java/awt/Desktop.html#getDesktop--)
* [java.lang.System#getSecurityManager()](http://docs.oracle.com/javase/8/docs/api/java/lang/System.html#getSecurityManager--)

**结论**

* 控制自己的创建和生命周期来违反了单一责任原则（SRP）。
* 鼓励使用全局共享实例，导致此对象使用的对象和资源很难被释放。
* 创建紧密耦合的代码。 Singleton的客户变得难以测试。
* 几乎不可能对Singleton进行子类化。

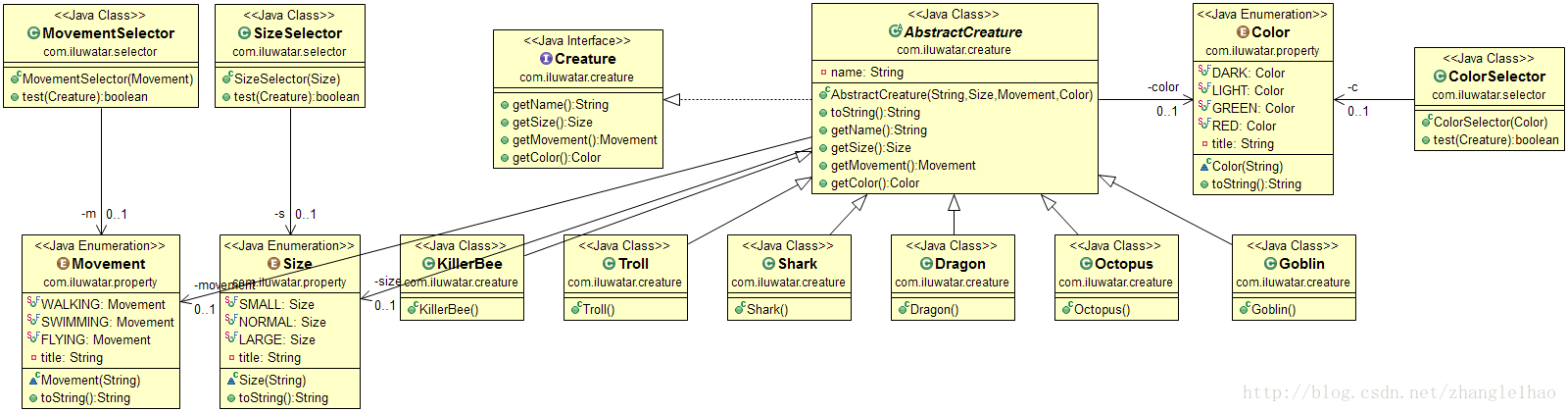
**权威信息**

* [Design Patterns: Elements of Reusable Object-Oriented Software](http://www.amazon.com/Design-Patterns-Elements-Reusable-Object-Oriented/dp/0201633612)
* [Effective Java (2nd Edition)](http://www.amazon.com/Effective-Java-Edition-Joshua-Bloch/dp/0321356683)

**89.规范**

| **标题** | **类别** | **难度** |
| --- | --- | --- |
| 规范（specification） | 行为型 | 简单难度 |

**意图**

规范模式分离如何匹配一个候选的对象组。 它是有作用的对于，值的校验和绑定命令。   


**适用范围**

何时使用规格图样

* 您需要根据某些条件选择对象的子集，并在不同时间刷新选择
* 您需要检查只有适当的对象用于某个角色（验证）

**关联模式**

* Repository

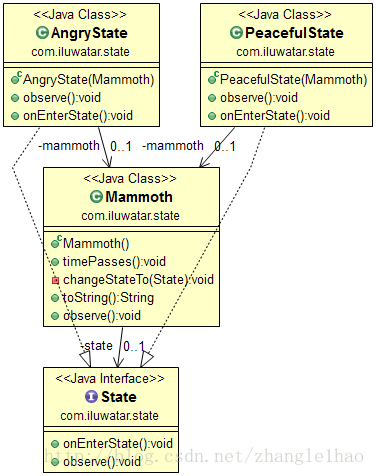
**权威信息**

* [Martin Fowler - Specifications](http://martinfowler.com/apsupp/spec.pdf)

**90.状态**

| **标题** | **类别** | **难度** |
| --- | --- | --- |
| 状态（state） | 行为型 | 中等难度,GOF |

**意图**

允许对象在其内部状态时更改其行为变化。 该对象将会改变其类。   


**适用范围**

在以下任何一种情况下使用状态模式

* 对象的行为取决于其状态，并且必须根据该状态在运行时更改其行为
* 操作具有取决于对象状态的大型多部分条件语句。 该状态通常由一个或多个枚举的常数表示。 通常，几个操作将包含相同的条件结构。 状态模式将条件的每个分支放在一个单独的类中。 这使您可以将对象的状态视为本身可以独立于其他对象的对象。

**真实例子**

* [javax.faces.lifecycle.Lifecycle#execute()](http://docs.oracle.com/javaee/7/api/javax/faces/lifecycle/Lifecycle.html#execute-javax.faces.context.FacesContext-) controlled by [FacesServlet](http://docs.oracle.com/javaee/7/api/javax/faces/webapp/FacesServlet.html), the behavior is dependent on current phase of lifecycle.
* [JDiameter - Diameter State Machine](https://github.com/npathai/jdiameter/blob/master/core/jdiameter/api/src/main/java/org/jdiameter/api/app/State.java)

**权威信息**

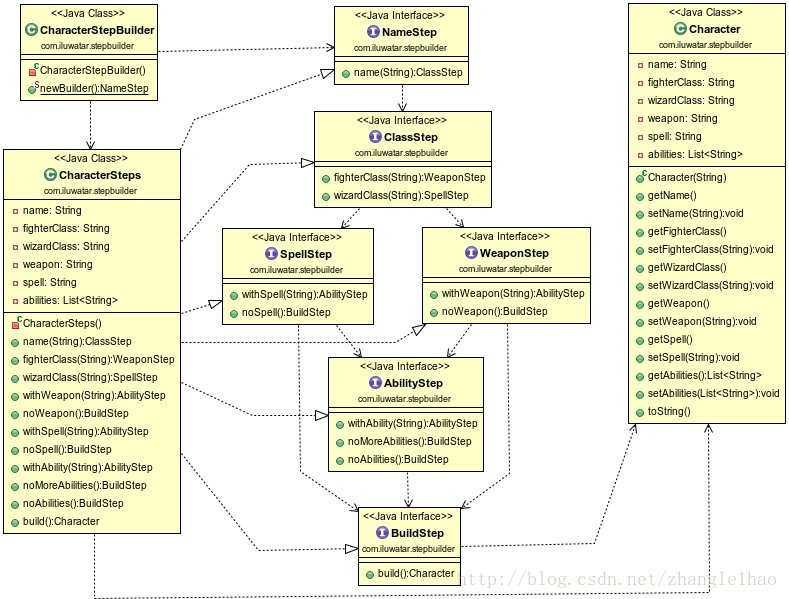
* [Design Patterns: Elements of Reusable Object-Oriented Software](http://www.amazon.com/Design-Patterns-Elements-Reusable-Object-Oriented/dp/0201633612)

**91.一步一步创建者**

| **标题** | **类别** | **难度** |
| --- | --- | --- |
| 一步一步创建者（step-builder） | 创建型 | 中等难度 |

**意图**

Builder模式的扩展，完全引导用户创建对象，没有混淆的机会。   
用户体验将会因为只能看到下一步方法可用，而不是构建方法直到构建对象才是正确的时间，因此将会得到更多的改进。



**适用范围**

当创建复杂对象的算法应该独立于组成对象的部分以及组合方式时，使用Step Builder模式，构造过程必须允许在构造顺序的过程中构造的对象的不同表示。

**权威信息**

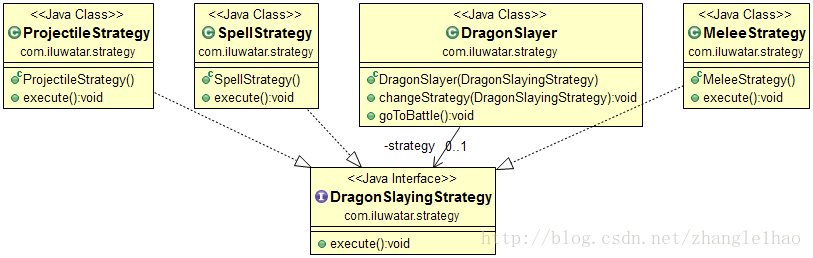
* [Marco Castigliego - Step Builder](http://rdafbn.blogspot.co.uk/2012/07/step-builder-pattern_28.html)

**92.策略**

| **标题** | **类别** | **难度** |
| --- | --- | --- |
| 策略（strategy） | 行为型 | GOF,简单难度 |

**意图**

定义一系列算法，封装每一个算法，确保他们可以通用。 策略模式让算法的变化独立与客户端的使用。



**适用范围**

使用策略模式

* 许多相关的类在其行为上有所不同。 策略提供了一种将类配置为许多行为之一的方法
* 您需要一个算法的不同变体。 例如，您可以定义反映不同空间/时间权衡的算法。 当这些变体被实现为算法的类层次结构时，可以使用策略
* 一个算法使用客户端不应该知道的数据。 使用策略模式避免暴露复杂的算法特定的数据结构
* 一个类定义了许多行为，并且它们在其操作中显示为多个条件语句。 而不是许多条件，将相关的条件分支移动到自己的策略类中

**权威信息**

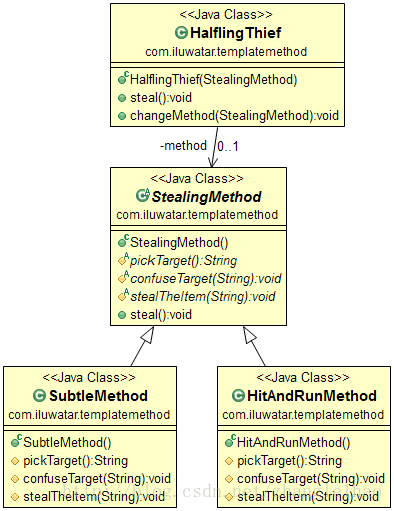
* [Design Patterns: Elements of Reusable Object-Oriented Software](http://www.amazon.com/Design-Patterns-Elements-Reusable-Object-Oriented/dp/0201633612)
* [Functional Programming in Java: Harnessing the Power of Java 8 Lambda Expressions](http://www.amazon.com/Functional-Programming-Java-Harnessing-Expressions/dp/1937785467/ref=sr_1_1)

**93.模板方法**

| **标题** | **类别** | **难度** |
| --- | --- | --- |
| 模板方法（template-method） | 行为型 | GOF,简单难度 |

**意图**

在操作中定义算法的骨架，推迟一些算法步骤到子类。 模板方法让子类重新定义某些步骤   
,一种不改变算法结构的方式。



**适用范围**

何时使用模板方法模式

* 实现一个算法的不变部分，并将其留给子类来实现可以变化的行为
* 当子类中的常见行为应该被考虑并定位在公共类中以避免代码重复时。 这是Opdyke和Johnson所描述的“重构泛化”的好例子。 您首先识别现有代码中的差异，然后将差异分为新操作。 最后，您使用调用其中一个新操作的模板方法替换不同的代码
* 控制子类扩展。 您可以定义一个在特定点调用“钩”操作的模板方法，从而仅在这些点允许扩展

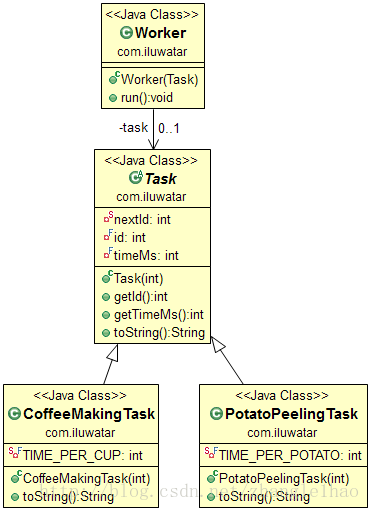
**权威信息**

* [Design Patterns: Elements of Reusable Object-Oriented Software](http://www.amazon.com/Design-Patterns-Elements-Reusable-Object-Oriented/dp/0201633612)

**94.线程池**

| **标题** | **类别** | **难度** |
| --- | --- | --- |
| 线程池（thread-pool） | 行为型 | 性能，中等难度 |

**意图**

通常情况下要执行的任务是短暂的并且任务数量大。 为每个任务创建一个新线程   
系统花费更多的时间而不是执行实际任务。 线程池通过重用现有的线程来解决这个问题   
并消除创建新线程的延迟。   


**适用范围**

使用线程池模式

* 您有大量短命的任务并行执行

**95.节流**

| **标题** | **类别** | **难度** |
| --- | --- | --- |
| 节流（throttling） | 行为型 | 简单难度 |

**意图**

在超过服务端分配限制的情况下，限制客户端的访问

**适用范围**

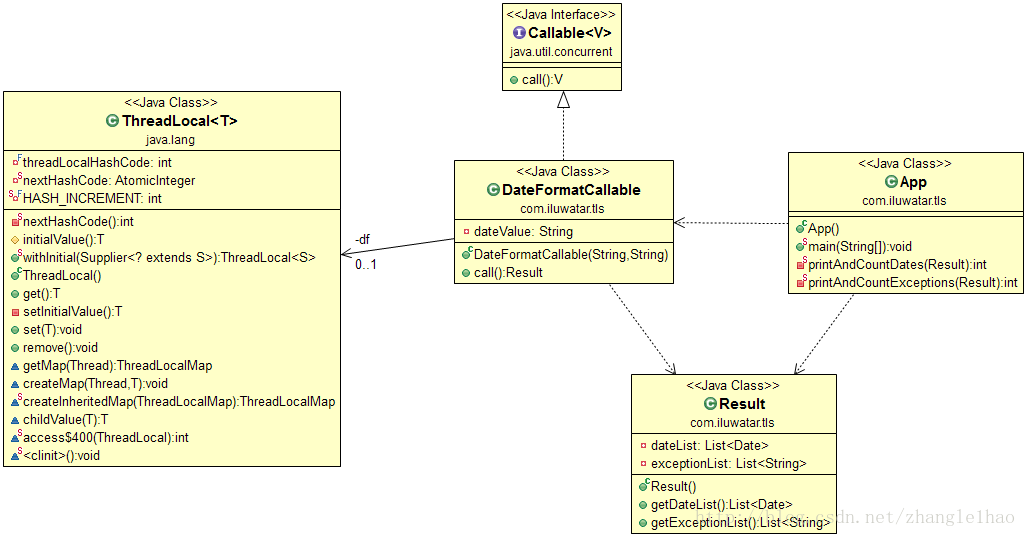
应使用节流模式：

* 需要限制服务访问对服务性能没有很大的影响。
* 当多个客户端消耗相同的服务资源，并且必须根据每个客户端的使用量进行限制。

**95.线程本地存储**

| **标题** | **类别** | **难度** |
| --- | --- | --- |
| 线程本地存储（Thread Local Storage） | 并发型 | 中等难度 |

**意图**

将变量保存到线程，以免被其他线程损坏。 如果您在Callable对象或Runnable对象中使用不是只读的类变量或静态变量，则需要这样做。   


**适用范围**

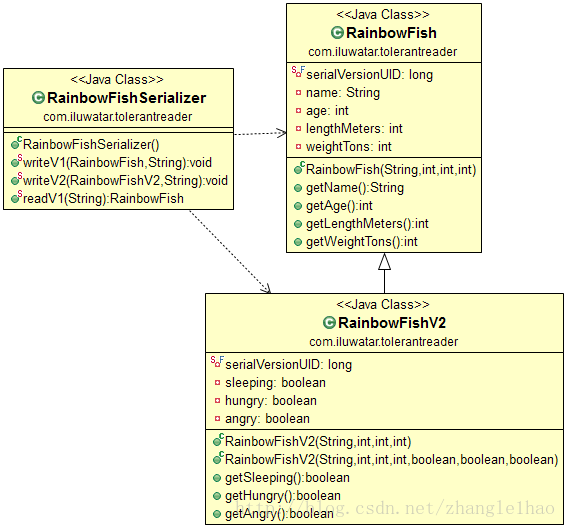
在以下任何情况下使用线程本地存储

* 当您在Callable / Runnable对象中使用不是只读的类变量，并且在并行运行的多个线程中使用相同的Callable实例。
* 当您在Callable / Runnable对象中使用非只读的静态变量，并且Callable / Runnable的多个实例可能并行运行时。

**96.宽容阅读**

| **标题** | **类别** | **难度** |
| --- | --- | --- |
| 宽容阅读（tolerant-reader） | 交互 | 简单难度 |

**意图**

宽容阅读是一种有助于创建的集成健壮的通信系统。 这个想法是尽可能的宽容从另一个服务读取数据。 这样，当通信模式改变，接收者一定不会崩溃。   


**适用范围**

使用宽容阅读模式

* 通信模式可以演变和改变，而接收方不应该崩溃

**权威信息**

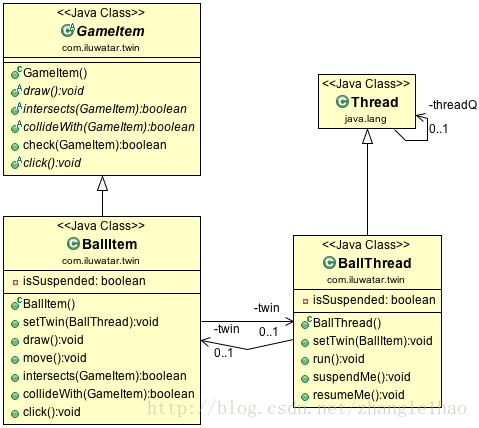
* [Martin Fowler - Tolerant Reader](http://martinfowler.com/bliki/TolerantReader.html)

**97. 双模式**

| **标题** | **类别** | **难度** |
| --- | --- | --- |
| 双胞胎（twin） | 创建型 | 中等难度 |

**意图**

双模式是一种设计模式，提供了一个标准的解决方案去模拟多继承



**适用范围**

何时使用双模式术语

* 不支持此功能的语言模拟多重继承。
* 避免某些多重继承的问题，如名称冲突。

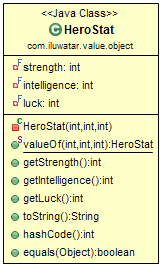
**权威信息**

* [Twin – A Design Pattern for Modeling Multiple Inheritance](http://www.ssw.uni-linz.ac.at/Research/Papers/Moe99/Paper.pdf)

**98. 值对象**

| **标题** | **类别** | **难度** |
| --- | --- | --- |
| 值对象（value-object） | 创建型 | 简单难度 |

**意图**

提供值语义而不是引用语义的对象。   
这意味着价对象的equal不是等于两个对象相同。 两个值对象是   
当它们具有相同的值时，不一定是相同的对象。   


**适用范围**

何时使用值对象

* 您需要根据对象的值来衡量对象的等式

**真实例子**

* [java.util.Optional](https://docs.oracle.com/javase/8/docs/api/java/util/Optional.html)
* [java.time.LocalDate](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html)
* [joda-time, money, beans](http://www.joda.org/)

**权威信息**

* [Patterns of Enterprise Application Architecture](http://www.martinfowler.com/books/eaa.html)
* [VALJOs - Value Java Objects : Stephen Colebourne’s blog](http://blog.joda.org/2014/03/valjos-value-java-objects.html)
* [Value Object : Wikipedia](https://en.wikipedia.org/wiki/Value_object)

**99. 访问者**

| **标题** | **类别** | **难度** |
| --- | --- | --- |
| 访问者（visitor） | 行为型 | GOF,中等难度 |

**意图**

表示要对对象的元素执行的操作结构体。 访问者可以在不更改类的情况下定义新操作   
的操作元素。

**适用范围**

何时使用访客模式

* 对象结构包含许多具有不同接口的对象类，并且您希望对这些对象执行依赖于其具体类的操作
* 需要对对象结构中的对象执行许多不同且不相关的操作，并且您希望避免使用这些操作“污染”其类。 访问者可以通过在一个类中定义相关操作来保持相关操作。 当对象结构由许多应用程序共享时，使用Visitor将操作放在需要它们的应用程序中
* 定义对象结构的类很少更改，但是您经常希望在结构上定义新的操作。 更改对象结构类需要重新定义所有访问者的接口，这可能是昂贵的。 如果对象结构类经常发生变化，那么最好在这些类中定义操作

**真实例子**

* [Apache Wicket](https://github.com/apache/wicket) component tree, see [MarkupContainer](https://github.com/apache/wicket/blob/b60ec64d0b50a611a9549809c9ab216f0ffa3ae3/wicket-core/src/main/java/org/apache/wicket/MarkupContainer.java)
* [javax.lang.model.element.AnnotationValue](http://docs.oracle.com/javase/8/docs/api/javax/lang/model/element/AnnotationValue.html) and [AnnotationValueVisitor](http://docs.oracle.com/javase/8/docs/api/javax/lang/model/element/AnnotationValueVisitor.html)
* [javax.lang.model.element.Element](http://docs.oracle.com/javase/8/docs/api/javax/lang/model/element/Element.html) and [Element Visitor](http://docs.oracle.com/javase/8/docs/api/javax/lang/model/element/ElementVisitor.html)
* [java.nio.file.FileVisitor](http://docs.oracle.com/javase/8/docs/api/java/nio/file/FileVisitor.html)

**权威信息**

* [Design Patterns: Elements of Reusable Object-Oriented Software](http://www.amazon.com/Design-Patterns-Elements-Reusable-Object-Oriented/dp/0201633612)

**引用**

[1][设计模式沉思录](http://blog.csdn.net/dc_726/article/details/78149675)   
[2][MVC vs. MVP vs. MVVM on Android](https://academy.realm.io/posts/eric-maxwell-mvc-mvp-and-mvvm-on-android/)   
[3][java 设计模式](https://github.com/iluwatar/java-design-patterns)