# 创建型模式

## 单例模式

角色：

单例类、访问类。

场景：

* 在应用场景中，某类只要求生成一个对象的时候，如一个班中的班长、每个人的身份证号等。
* 当对象需要被共享的场合。由于单例模式只允许创建一个对象，共享该对象可以节省内存，并加快对象访问速度。如 Web 中的配置对象、数据库的连接池等。
* 当某类需要频繁实例化，而创建的对象又频繁被销毁的时候，如多线程的线程池、网络连接池等。

## 原型模式

角色：

抽象原型类、具体原型类、访问类。

场景：

* 对象之间相同或相似，即只是个别的几个属性不同的时候。
* 对象的创建过程比较麻烦，但复制比较简单的时候。

## 工厂方法模式

角色：

抽象工厂、具体工厂、抽象产品、具体产品。

场景：

* 客户只知道创建产品的工厂名，而不知道具体的产品名。如 TCL 电视工厂、海信电视工厂等。
* 创建对象的任务由多个具体子工厂中的某一个完成，而抽象工厂只提供创建产品的接口。
* 客户不关心创建产品的细节，只关心产品的品牌。

## 抽象工厂模式

角色：

抽象工厂、具体工厂、抽象产品、具体产品。

场景：

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

## 建造者模式

角色：

产品角色、抽象建造者、具体建造者、指挥者。

场景：

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

# 结构型模式

## 代理模式

角色：

抽象主题类、真实主题类、代理类

场景：

* 远程代理，这种方式通常是为了隐藏目标对象存在于不同地址空间的事实，方便客户端访问。例如，用户申请某些网盘空间时，会在用户的文件系统中建立一个虚拟的硬盘，用户访问虚拟硬盘时实际访问的是网盘空间。
* 虚拟代理，这种方式通常用于要创建的目标对象开销很大时。例如，下载一幅很大的图像需要很长时间，因某种计算比较复杂而短时间无法完成，这时可以先用小比例的虚拟代理替换真实的对象，消除用户对服务器慢的感觉。
* 安全代理，这种方式通常用于控制不同种类客户对真实对象的访问权限。
* 智能指引，主要用于调用目标对象时，代理附加一些额外的处理功能。例如，增加计算真实对象的引用次数的功能，这样当该对象没有被引用时，就可以自动释放它。
* 延迟加载，指为了提高系统的性能，延迟对目标的加载。例如，[Hibernate](http://c.biancheng.net/hibernate/) 中就存在属性的延迟加载和关联表的延时加载。

## 适配器模式

角色：

目标接口、适配者类、适配器类。

场景：

* 以前开发的系统存在满足新系统功能需求的类，但其接口同新系统的接口不一致。
* 使用第三方提供的组件，但组件接口定义和自己要求的接口定义不同。

## 桥接模式

角色：

抽象化角色、扩展抽象化角色、实现化角色、具体实现化角色。

场景：

* 当一个类存在两个独立变化的维度，且这两个维度都需要进行扩展时。
* 当一个系统不希望使用继承或因为多层次继承导致系统类的个数急剧增加时。
* 当一个系统需要在构件的抽象化角色和具体化角色之间增加更多的灵活性时。

## 装饰模式

角色：

抽象构件角色、具体构件角色、抽象装饰角色、具体装饰角色。

场景：

* 当需要给一个现有类添加附加职责，而又不能采用生成子类的方法进行扩充时。例如，该类被隐藏或者该类是终极类或者采用继承方式会产生大量的子类。
* 当需要通过对现有的一组基本功能进行排列组合而产生非常多的功能时，采用继承关系很难实现，而采用装饰模式却很好实现。
* 当对象的功能要求可以动态地添加，也可以再动态地撤销时。

## 外观模式

角色：

外观角色、子系统角色、客户角色。

场景：

* 对分层结构系统构建时，使用外观模式定义子系统中每层的入口点可以简化子系统之间的依赖关系。
* 当一个复杂系统的子系统很多时，外观模式可以为系统设计一个简单的接口供外界访问。
* 当客户端与多个子系统之间存在很大的联系时，引入外观模式可将它们分离，从而提高子系统的独立性和可移植性。

## 享元模式

角色：

抽象享元角色、具体享元角色、非享元角色、享元工厂角色。

场景：

* 系统中存在大量相同或相似的对象，这些对象耗费大量的内存资源。
* 大部分的对象可以按照内部状态进行分组，且可将不同部分外部化，这样每一个组只需保存一个内部状态。
* 由于享元模式需要额外维护一个保存享元的[数据结构](http://c.biancheng.net/data_structure/" \t "_blank)，所以应当在有足够多的享元实例时才值得使用享元模式。

## 组合模式

角色：

抽象构件角色、树叶构件角色、树枝构件角色。

场景：

* 在需要表示一个对象整体与部分的层次结构的场合。
* 要求对用户隐藏组合对象与单个对象的不同，用户可以用统一的接口使用组合结构中的所有对象的场合。

# 行为型模式

## 模板方法模式

角色：

抽象类、具体子类。

场景：

* 算法的整体步骤很固定，但其中个别部分易变时，这时候可以使用模板方法模式，将容易变的部分抽象出来，供子类实现。
* 当多个子类存在公共的行为时，可以将其提取出来并集中到一个公共父类中以避免代码重复。首先，要识别现有代码中的不同之处，并且将不同之处分离为新的操作。最后，用一个调用这些新的操作的模板方法来替换这些不同的代码。
* 当需要控制子类的扩展时，模板方法只在特定点调用钩子操作，这样就只允许在这些点进行扩展。

## 策略模式

角色：

抽象策略类、具体策略类、环境类。

场景：

* 一个系统需要动态地在几种算法中选择一种时，可将每个算法封装到策略类中。
* 一个类定义了多种行为，并且这些行为在这个类的操作中以多个条件语句的形式出现，可将每个条件分支移入它们各自的策略类中以代替这些条件语句。
* 系统中各算法彼此完全独立，且要求对客户隐藏具体算法的实现细节时。
* 系统要求使用算法的客户不应该知道其操作的数据时，可使用策略模式来隐藏与算法相关的数据结构。
* 多个类只区别在表现行为不同，可以使用策略模式，在运行时动态选择具体要执行的行为。

## 命令模式

角色：

抽象命令类角色、具体命令角色、实现者/接收者角色、调用者/请求者角色。

场景：

* 当系统需要将请求调用者与请求接收者解耦时，命令模式使得调用者和接收者不直接交互。
* 当系统需要随机请求命令或经常增加或删除命令时，命令模式比较方便实现这些功能。
* 当系统需要执行一组操作时，命令模式可以定义宏命令来实现该功能。
* 当系统需要支持命令的撤销（Undo）操作和恢复（Redo）操作时，可以将命令对象存储起来，采用备忘录模式来实现。

## 责任链模式

角色：

抽象处理者角色、具体处理者角色、客户类角色。

场景：

* 有多个对象可以处理一个请求，哪个对象处理该请求由运行时刻自动确定。
* 可动态指定一组对象处理请求，或添加新的处理者。
* 在不明确指定请求处理者的情况下，向多个处理者中的一个提交请求。

## 状态模式

角色：

环境角色、抽象状态角色、具体状态角色。

场景：

* 当一个对象的行为取决于它的状态，并且它必须在运行时根据状态改变它的行为时，就可以考虑使用状态模式。
* 一个操作中含有庞大的分支结构，并且这些分支决定于对象的状态时。

## 观察者模式

角色：

抽象主题角色、具体主题角色、抽象观察者角色、具体观察者角色。

场景：

* 对象间存在一对多关系，一个对象的状态发生改变会影响其他对象。
* 当一个抽象模型有两个方面，其中一个方面依赖于另一方面时，可将这二者封装在独立的对象中以使它们可以各自独立地改变和复用。

## 中介者模式

角色：

抽象中介者角色、具体中介者角色、抽象同事类角色、具体同时类角色。

场景：

* 当对象之间存在复杂的网状结构关系而导致依赖关系混乱且难以复用时。
* 当想创建一个运行于多个类之间的对象，又不想生成新的子类时。

## 迭代器模式

角色：

抽象聚合角色、具体聚合角色、抽象迭代器角色、具体迭代器角色。

场景：

* 当需要为聚合对象提供多种遍历方式时。
* 当需要为遍历不同的聚合结构提供一个统一的接口时。
* 当访问一个聚合对象的内容而无须暴露其内部细节的表示时。

## 访问者模式

角色：

抽象访问者角色、具体访问者角色、抽象元素角色、具体元素角色、对象结构角色。

场景：

* 对象结构相对稳定，但其操作算法经常变化的程序。
* 对象结构中的对象需要提供多种不同且不相关的操作，而且要避免让这些操作的变化影响对象的结构。
* 对象结构包含很多类型的对象，希望对这些对象实施一些依赖于其具体类型的操作。

## 备忘录模式

角色：

发起人角色、备忘录角色、管理者角色。

场景：

* 需要保存与恢复数据的场景，如玩游戏时的中间结果的存档功能。
* 需要提供一个可回滚操作的场景，如 Word、记事本、Photoshop，Eclipse 等软件在编辑时按 Ctrl+Z 组合键，还有数据库中事务操作。

## 解释器模式

角色：

抽象表达式角色、终结符表达式角色、非终结符表达式角色、环境角色、客户端。

场景：

* 当语言的文法较为简单，且执行效率不是关键问题时。
* 当问题重复出现，且可以用一种简单的语言来进行表达时。
* 当一个语言需要解释执行，并且语言中的句子可以表示为一个抽象语法树的时候，如 XML 文档解释。