一、创建型

1. 单例模式

确保只实例化一个对象，并提供一个对象的全局访问点。

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

2. 简单工厂模式

在不对用户暴露对象内部逻辑的前提下创建对象。

3. 工厂方法模式

定义创建对象的接口，但是让子类来决定应该使用哪个类来创建。

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

4. 抽象工厂

提供一个创建相关对象家族的接口，而没有明确指明它们的类。

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

5. 生成器模式

定义一个新的类来构造另一个类的实例，以创建一个复杂的对象。

它可以封装一个对象的构造过程，并允许按步骤构造。

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

6. 原型模式

使用原型实例指定要创建对象的类型；通过复制这个原型来创建新对象。

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

二、行为型

1. 责任链

避免将请求的发送者附加到其接收者，从而使其它对象也可以处理请求；将请求以对象的方式发送到链上直到请求被处理完毕。

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

2. 命令模式

将命令封装进对象中；允许使用命令对象对客户对象进行参数化；允许将命令对象存放到队列中。

```
java.lang.Runnable
javax.swing.Action
```

3. 解释器模式

为语言创建解释器，通常由语言的语法和语法分析来定义。

```
java.util.Pattern
java.text.Normalizer
java.text.Format
```

4. 迭代器

提供一种一致的访问聚合对象元素的方法，并且不暴露聚合对象的内部表示。

```
java.util.Iterator
java.util.Enumeration
```

5. 中间人模式

使用中间人对象来封装对象之间的交互。中间人模式可以降低交互对象之间的耦合程度。

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

6. 备忘录模式

在不违反封装的情况下获得对象的内部状态，从而在需要时可以将对象恢复到最初状态。

```
java.util.Date
java.io.Serializable
```

7. 观察者模式

定义对象之间的一对多依赖，当一个对象状态改变时，它的所有依赖都会收到通知并且自动更新状态。

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

8. 策略模式

定义一系列算法，封装每个算法，并使它们可以互换。策略可以让算法独立于使用它的客户端。

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

9. 模板方法

定义算法框架，并将一些步骤的实现延迟到子类。通过模板方法，子类可以重新定义算法的某些步骤，而不用改变算法的结构。

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

10. 访问者模式

提供便捷的维护方式来操作一组对象。它使你在不改变操作对象的前提下，可以修改或扩展对象的行为。

例如集合，它可以包含不同类型的元素，访问者模式允许在不知道具体元素类型的前提下对集合元素进行一些操作。

```
javax.lang.model.element.Element and javax.lang.model.element.ElementVisitor
javax.lang.model.type.TypeMirror and javax.lang.model.type.TypeVisitor
```

11. 空对象模式

使用什么都不做的空对象来替代 NULL。

三、结构型

1. 适配器

把一个类接口转换成另一个用户需要的接口。

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

2. 桥接模式

将抽象与实现分离开来，使它们可以独立变化。

```
AWT (It provides an abstraction layer which maps onto the native OS the windowing support.)
JDBC
```

3. 组合模式

将对象组合成树形结构来表示整体-部分层次关系，允许用户以相同的方式处理单独对象和组合对象。

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

4. 装饰者模式

为对象动态添加功能。

```
java.io.BufferedInputStream(InputStream)
java.io.DataInputStream(InputStream)
java.io.BufferedOutputStream(OutputStream)
java.util.zip.ZipOutputStream(OutputStream)
java.util.Collections#checked[List|Map|Set|SortedSet|SortedMap]()
```

5. 蝇量模式

利用共享的方式来支持大量的对象，这些对象一部分内部状态是相同的，而另一份状态可以变化。

Java 利用缓存来加速大量小对象的访问时间。

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

6. 动态代理

提供一个占位符来控制对象的访问。

代理可以是一些轻量级的对象，它控制着对重量级对象的访问，只有在真正实例化这些重量级对象时才会去实例化它。

```
java.lang.reflect.Proxy
RMI
```

参考资料

- The breakdown of design patterns in JDK
- Design Patterns