创建型模式

工厂模式（Factory Pattern）

工厂模式是 Java 中最常用的设计模式之一。这种类型的设计模式属于创建型模式，它提供了一种创建对象的最佳方式。 在工厂模式中，我们在创建对象时不会对客户端暴露创建逻辑，并且是通过使用一个共同的接口来指向新创建的对象。

意图

定义一个创建对象的接口，让其子类自己决定实例化哪一个工厂类，工厂模式使其创建过程延迟到子类进行。

主要解决

主要解决接口选择的问题。

何时使用

我们明确地计划不同条件下创建不同实例时。

如何解决

让其子类实现工厂接口，返回的也是一个抽象的产品。

关键代码

创建过程在其子类执行。

应用实例

您需要一辆汽车，可以直接从工厂里面提货，而不用去管这辆汽车是怎么做出来的，以及这个汽车里面的具体实现。

优点

- 一个调用者想创建一个对象，只要知道其名称就可以了。
- 扩展性高，如果想增加一个产品，只要扩展一个工厂类就可以。
- 屏蔽产品的具体实现，调用者只关心产品的接口。

缺点

每次增加一个产品时，都需要增加一个具体类和对象实现工厂，使得系统中类的个数成倍增加，在一定程度上增加了系统的复杂度，同时也增加了系统具体类的依赖。这并不是什么好事。

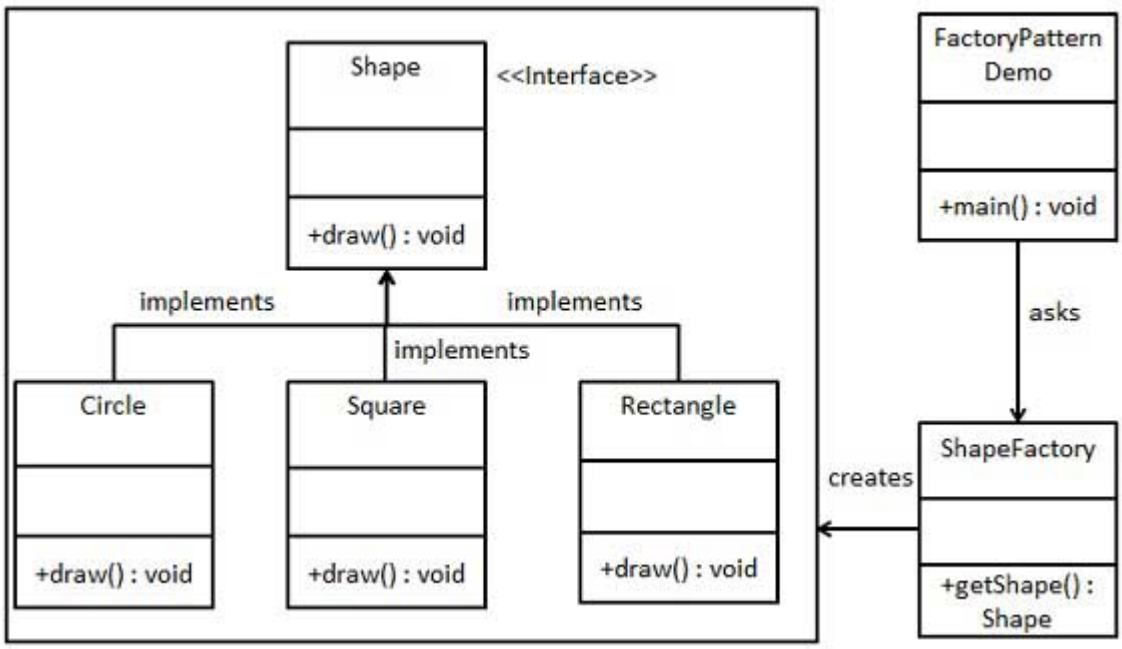
使用场景

1. 日志记录器：记录可能记录到本地硬盘、系统事件、远程服务器等，用户可以选择记录日志到什么地方。
2. 数据库访问，当用户不知道最后系统采用哪一类数据库，以及数据库可能有变化时。
3. 设计一个连接服务器的框架，需要三个协议，"POP3"、 "IMAP"、"HTTP"，可以把这三个作为产品类，共同实现一个接口。

注意

作为一种创建类模式，在任何需要生成复杂对象的地方，都可以使用工厂方法模式。有一点需要注意的地方就是复杂对象适合使用 工厂模式，而简单对象，特别是只需要通过 new 就可以完成创建的对象，无需使用工厂模式。如果使用工厂模式，就需要引入一个工厂类，会增加系统的复杂度。

示例实现



通过传入的字符串来判断应该实例化哪个Shape的子类（工厂类也可以有一个基类称为抽象工厂，在下面讨论），具体代码就不给出了。

抽象工厂模式（Abstract Factory Pattern）

抽象工厂模式是围绕一个超级工厂创建其他工厂。该超级工厂又称为其他工厂的工厂。这种类型的设计模式属于创建型模式，它提供了一种创建对象的最佳方式。 在抽象工厂模式中，接口是负责创建一个相关对象的工厂，不需要显式指定它们的类。每个生成的工厂都能按照工厂模式提供对象。

意图

提供一个创建一系列相关或相互依赖对象的接口，而无需指定它们具体的类。

主要解决

主要解决接口选择的问题。

何时使用

系统的产品有多于一个的产品族，而系统只消费其中某一族的产品

如何解决

在一个产品族里面，定义多个产品

关键代码

在一个工厂里聚合多个同类产品。

应用实例

工作了，为了参加一些聚会，肯定有两套或多套衣服吧，比如说有商务装（成套，一系列具体产品）、时尚装（成套，一系列具体 产品），甚至对于一个家庭来说，可能有商务女装、商务男装、时尚女装、时尚男装，这些也都是成套的，即一系列具体产品。假设一种情况（现实中是不存在的， 要不然，没法进入共产主义了，但有利于说明抽象工厂模式），在您的家中，某一个衣柜（具体工厂）只能存放某一种这样的衣服（成套，一系列具体产品），每次 拿这种成套的衣服时也自然要从这个衣柜中取出了。用 OO 的思想去理解，所有的衣柜（具体工厂）都是衣柜类的（抽象工厂）某一个，而每一件成套的衣服又包括具体的上衣（某一具体产品），裤子（某一具体产品），这 些具体的上衣其实也都是上衣（抽象产品），具体的裤子也都是裤子（另一个抽象产品）。

优点

当一个产品族中的多个对象被设计成一起工作时，它能保证客户端始终只使用同一个产品族中的对象。

缺点

产品族扩展非常困难，要增加一个系列的某一产品，既要在抽象的 Creator 里加代码，又要在具体的里面加代码。

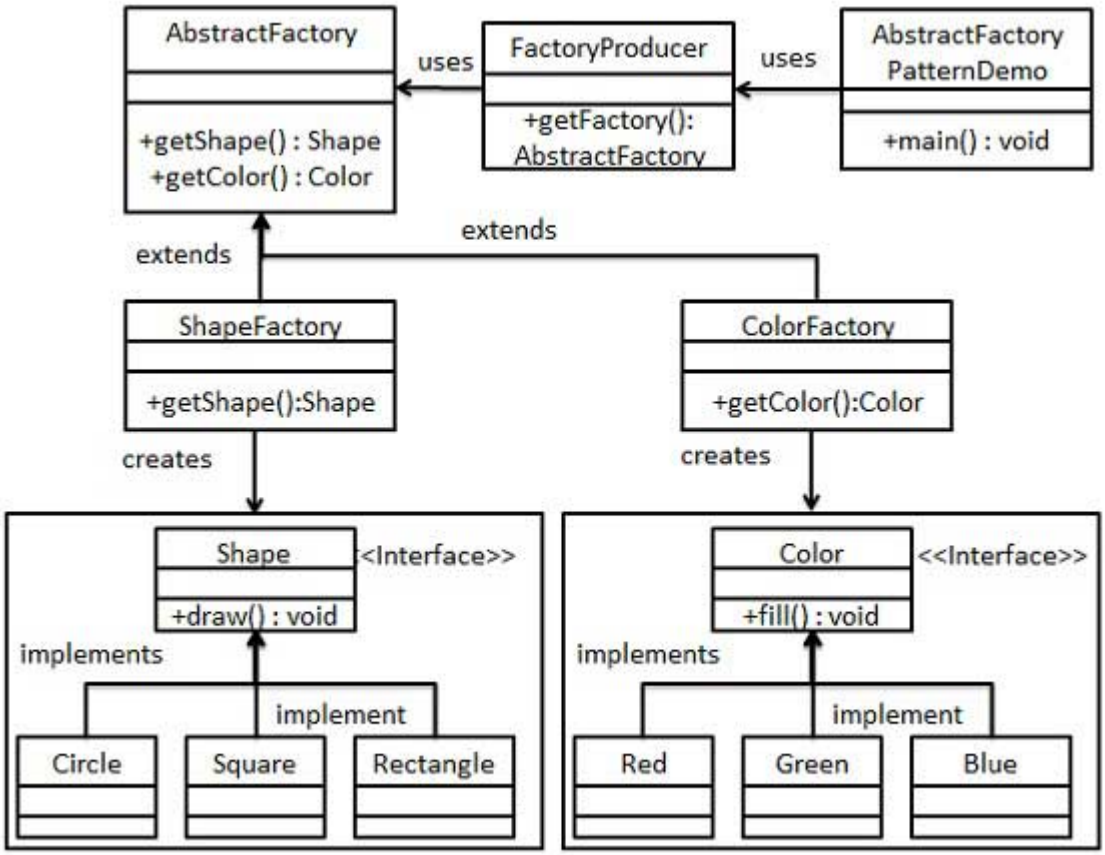
使用场景

1. QQ 换皮肤，一整套一起换。
2. 生成不同操作系统的程序。

注意

产品族难扩展，产品等级易扩展。

示例实现



代码逻辑比较简单：

1. FactoryProducer通过传入的参数来判断应该实例化哪个工厂类；
2. 客户端使用返回的工厂类来批量实例化对应的Shape或Color。
3. 工厂类使用（普通）工厂模式实现。

单例模式（Singleton Pattern）

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

注意

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

意图

保证一个类仅有一个实例，并提供一个访问它的全局访问点。

主要解决

一个全局使用的类频繁地创建与销毁。

何时使用

当您想控制实例数目，节省系统资源的时候。

如何解决

判断系统是否已经有这个单例，如果有则返回，如果没有则创建。

关键代码

构造函数是私有的。

应用实例

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

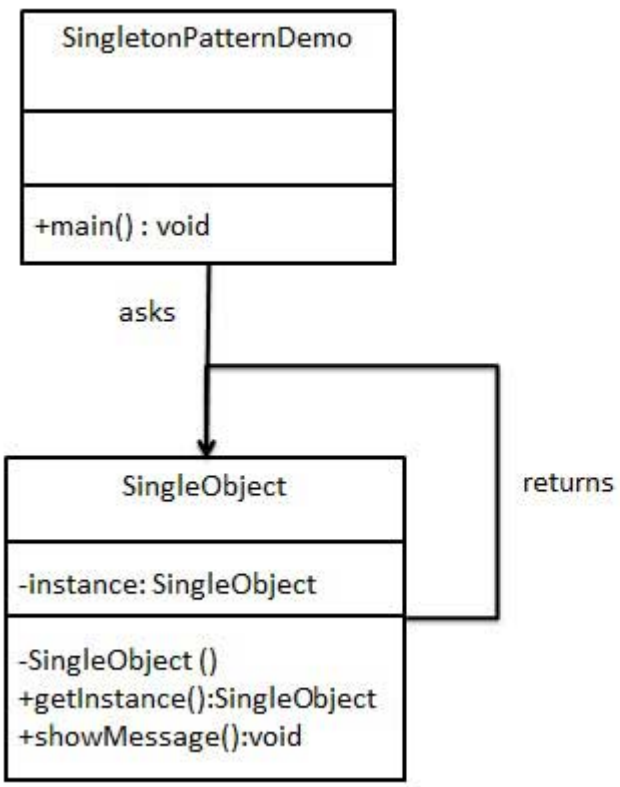
优点

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

缺点

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

实现



1. 构造方法是private的
2. 获取实例的唯一入口getInstance()是静态的，对对象的实例化存在多种方式，包括懒汉式、饿汉式、双检锁、静态内部类、枚举等方式，在线程安全、是否懒加载、实现难度上存在区别，以下分别进行讨论。

懒汉式（线程不安全）

是否 Lazy 初始化：是
是否多线程安全：否
实现难度：易
描述：这种方式是最基本的实现方式，这种实现最大的问题就是不支持多线程。因为没有加锁 synchronized，所以严格意义上它并不算单例模式。这种方式 lazy loading 很明显，不要求线程安全，在多线程不能正常工作。

```
public class Singleton {
    private static Singleton instance;
    private Singleton() {}
    public static Singleton getInstance() {
        if(instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

懒汉式（线程安全）

是否 Lazy 初始化：是
是否多线程安全：是
实现难度：易
描述：这种方式具备很好的 lazy loading，能够在多线程中很好的工作，但是，效率很低，99% 情况下不需要同步。优点：第一次调用才初始化，避免内存浪费。缺点：必须加锁 synchronized 才能保证单例，但加锁会影响效率。getInstance() 的性能对应用程序不是很关键（该方法使用不太频繁）。

```
public class Singleton {
    private static Singleton instance;
    private Singleton() {}
    public static synchronized Singleton getInstance() {
        if(instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
```

```
}
```

饿汉式

是否 Lazy 初始化：否
是否多线程安全：是
实现难度：易
描述：这种方式比较常用，但容易产生垃圾对象。
优点：没有加锁，执行效率会提高。
缺点：类加载时就初始化，浪费内存。
它基于 classloder 机制避免了多线程的同步问题，不过，instance 在类装载时就实例化，虽然导致类装载的原因有很多种，在单例模式中大多数都是调用 getInstance 方法， 但是也不能确定有其他的方式（或者其他的静态方法）导致类装载，这时候初始化 instance 显然没有达到 lazy loading 的效果。

```
public class Singleton {
    private static final Singleton instance = new Singleton();
    private Singleton() {}
    private static Singleton getInstance() {
        return instance;
    }
}
```

双检锁/双重校验锁（DCL，即 double-checked locking）

JDK 版本：JDK1.5 起
是否 Lazy 初始化：是
是否多线程安全：是
实现难度：较复杂
描述：这种方式采用双锁机制，安全且在多线程情况下能保持高性能。getInstance() 的性能对应用程序很关键。

```
public class Singleton {
    private volatile static Singleton instance;
    private Singleton() {}
    public static Singleton getSingleton() {
        if(instance == null) {
            synchronized(Singleton.class) {
                if(instance == null) {
                    instance = new Singleton();
                }
            }
```

```
        }
        return instance;
    }
}
```

登记式/静态内部类

是否 Lazy 初始化：是
是否多线程安全：是
实现难度：一般
描述：这种方式能达到双检锁方式一样的功效，但实现更简单。对静态域使用延迟初始化，应使用这种方式而不是双检锁方式。这种方式只适用于静态域的情况，双检锁方式可在实例域需要延迟初始化时使用。

```
public class Singleton {
    private static class SingletonHolder {
        private static final Singleton INSTANCE = new Single();
    }
    private Single() {}
    public static final Singleton getInstance() {
        return SingletonHolder.INSTANCE;
    }
}
```

枚举

JDK 版本：JDK1.5 起
是否 Lazy 初始化：否
是否多线程安全：是
实现难度：易
描述：这种实现方式还没有被广泛采用，但这是实现单例模式的最佳方法。它更简洁，自动支持序列化机制，绝对防止多次实例化。

```
public enum Singleton {
    INSTANCE;
    public void whateverMethod() {
    }
}
```

总结

一般情况下，不建议使用第 1 种和第 2 种懒汉方式，建议使用第 3 种饿汉方式。只有在要明确实现 lazy loading 效果时，才会使用第 5 种登记方式。如果涉及到反序列化创建对象时，可以尝试使用第 6 种枚举方式。如果有其他特殊的需求，可以考虑使用第 4 种双检锁方式。

建造者模式（Builder Pattern）

建造者模式使用多个简单的对象一步一步构建成一个复杂的对象。这种类型的设计模式属于创建型模式，它提供了一种创建对象的最佳方式。 一个 Builder 类会一步一步构造最终的对象。该 Builder 类是独立于其他对象的。

意图

将一个复杂的构建与其表示相分离，使得同样的构建过程可以创建不同的表示。

主要解决

主要解决在软件系统中，有时候面临着"一个复杂对象"的创建工作，其通常由各个部分的子对象用一定的算法构成；由于需求的变化，这个复杂对象的各个部分经常面临着剧烈的变化，但是将它们组合在一起的算法却相对稳定。

何时使用

一些基本部件不会变，而其组合经常变化的时候。

如何解决

将变与不变分离开。

关键代码

建造者：创建和提供实例，导演：管理建造出来的实例的依赖关系。

应用实例

去肯德基，汉堡、可乐、薯条、炸鸡翅等是不变的，而其组合是经常变化的，生成出所谓的"套餐"。

优点

1、建造者独立，易扩展。
2、便于控制细节风险。

缺点

1、产品必须有共同点，范围有限制。
2、如内部变化复杂，会有很多的建造类。

使用场景

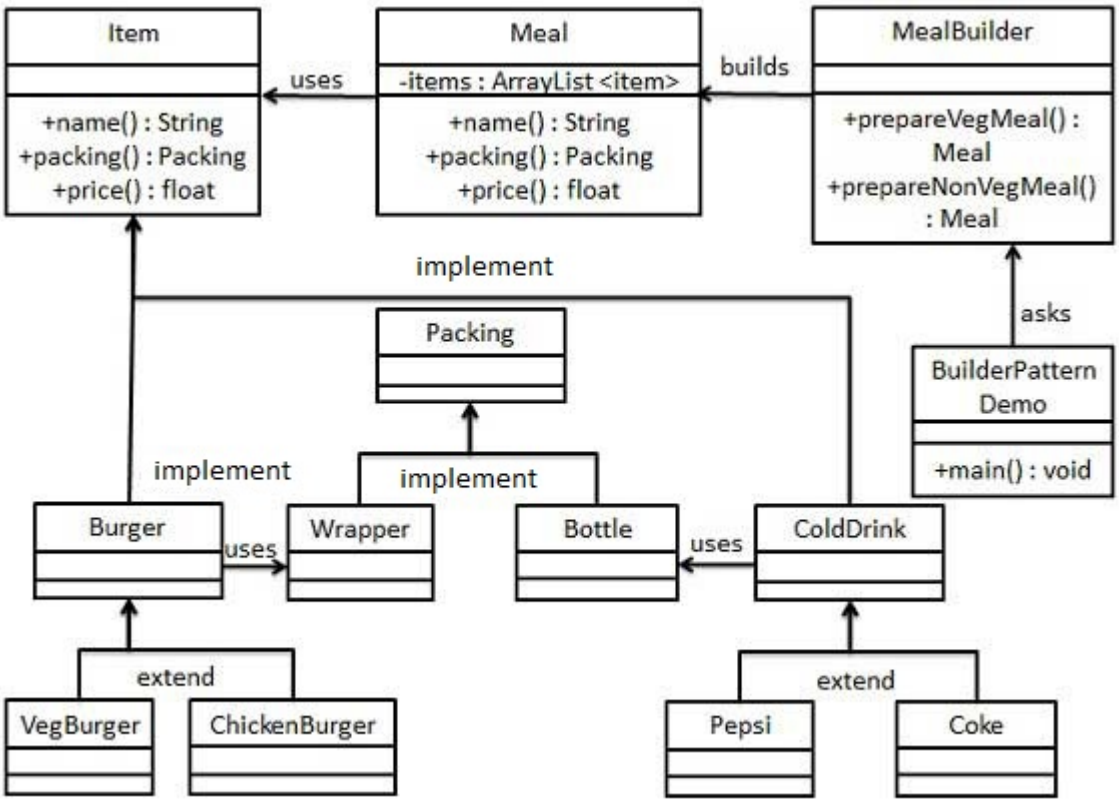
1、需要生成的对象具有复杂的内部结构。
2、需要生成的对象内部属性本身相互依赖。

注意事项

与工厂模式的区别是：建造者模式更加关注于零件装配的顺序。

示例实现

我们假设一个快餐店的商业案例，其中，一个典型的套餐可以是一个汉堡（Burger）和一杯冷饮（Cold drink）。汉堡（Burger）可以是素食汉堡（Veg Burger）或鸡肉汉堡（Chicken Burger），它们是包在纸盒中。冷饮（Cold drink）可以是可口可乐（coke）或百事可乐（pepsi），它们是装在瓶子中。 我们将创建一个表示食物条目（比如汉堡和冷饮）的 Item 接口和实现 Item 接口的实体类，以及一个表示食物包装的 Packing 接口和实现 Packing 接口的实体类，汉堡是包在纸盒中，冷饮是装在瓶子中。 然后我们创建一个 Meal 类，带有 Item 的 ArrayList 和一个通过结合 Item 来创建不同类型的 Meal 对象的 MealBuilder。BuilderPatternDemo，我们的演示类使用 MealBuilder 来创建一个 Meal。



上面的类图看起来可能会有些乱，这也是建造者所关注的问题：某个复合对象由多个相对简单的对象组合而成，我们亟需提供一个统一的接口来简化这个复合对象的构建过程。

原型模式（Prototype Pattern）

原型模式是用于创建重复的对象，同时又能保证性能。这种类型的设计模式属于创建型模式，它提供了一种创建对象的最佳方式。 这种模式是实现了一个原型接口，该接口用于创建当前对象的克隆。当直接创建对象的代价比较大时，则采用这种模式。例如，一个对象需要在一个高代价的数据库操作之后被创建。我们可以缓存该对象，在下一个请求时返回它的克隆，在需要的时候更新数据库，以此来减少数据库调用。

意图

用原型实例指定创建对象的种类，并且通过拷贝这些原型创建新的对象。

主要解决

在运行期建立和删除原型。

何时使用

1、当一个系统应该独立于它的产品创建，构成和表示时。
2、当要实例化的类是在运行时刻指定时，例如，通过动态装载。
3、为了避免创建一个与产品类层次平行的工厂类层次时。
4、当一个类的实例只能有几个不同状态组合中的一种时。建立相应数目的原型并克隆它们可能比每次用合适的状态手工实例化该类更方便一些。

如何解决

利用已有的一个原型对象，快速地生成和原型对象一样的实例。

关键代码

1、实现克隆操作，在 JAVA 继承 Cloneable，重写 clone()，在 .NET 中可以使用 Object 类的 MemberwiseClone() 方法来实现对象的浅拷贝或通过序列化的方式来实现深拷贝。
2、原型模式同样用于隔离类对象的使用者和具体类型（易变类）之间的耦合关系，它同样要求这些"易变类"拥有稳定的接口。

应用实例

1、细胞分裂。 2、JAVA 中的 Object clone() 方法。

优点

1、性能提高。 2、逃避构造函数的约束。

缺点

1、配备克隆方法需要对类的功能进行通盘考虑，这对于全新的类不是很难，但对于已有的类不一定很容易，特别当一个类引用不支持串行化的间接对象，或者引用含有循环结构的时候。 2、必须实现 Cloneable 接口。 3、逃避构造函数的约束。

使用场景

1、资源优化场景。 2、类初始化需要消化非常多的资源，这个资源包括数据、硬件资源等。 3、性能和安全要求的场景。 4、通过 new 产生一个对象需要非常繁琐的数据准备或访问权限，则可以使用原型模式。 5、一个对象多个修改者的场景。 6、一个对象需要提供给其他对象访问，而且各个调用者可能都需要修改其值时，可以考虑使用原型模式拷贝多个对象供调用者使用。 7、在实际项目中，原型模式很少单独出现，一般是和工厂方法模式一起出现，通过 clone 的方法创建一个对象，然后由工厂方法提供给调用者。原型模式已经与 Java 融为浑然一体，大家可以随手拿来使用。

注意事项

与通过对一个类进行实例化来构造新对象不同的是，原型模式是通过拷贝一个现有对象生成新对象的。浅拷贝实现 Cloneable，重写，深拷贝是通过实现 Serializable 读取二进制流。

如何解决

利用已有的一个原型对象，快速地生成和原型对象一样的实例。

关键代码

1、实现克隆操作，在 JAVA 继承 Cloneable，重写 clone()，在 .NET 中可以使用 Object 类的 MemberwiseClone() 方法来实现对象的浅拷贝或通过序列化的方式来实现深拷贝。 2、原型模式同样用于隔离类对象的使用者和具体类型（易变类）之间的耦合关系，它同样要求这些"易变类"拥有稳定的接口。

应用实例

1、细胞分裂。 2、JAVA 中的 Object clone() 方法。

优点

1、性能提高。 2、逃避构造函数的约束。

缺点

1、配备克隆方法需要对类的功能进行通盘考虑，这对于全新的类不是很难，但对于已有的类不一定很容易，特别当一个类引用不支持串行化的间接对象，或者引用含有循环结构的时候。 2、必须实现 Cloneable 接口。 3、逃避构造函数的约束。

使用场景

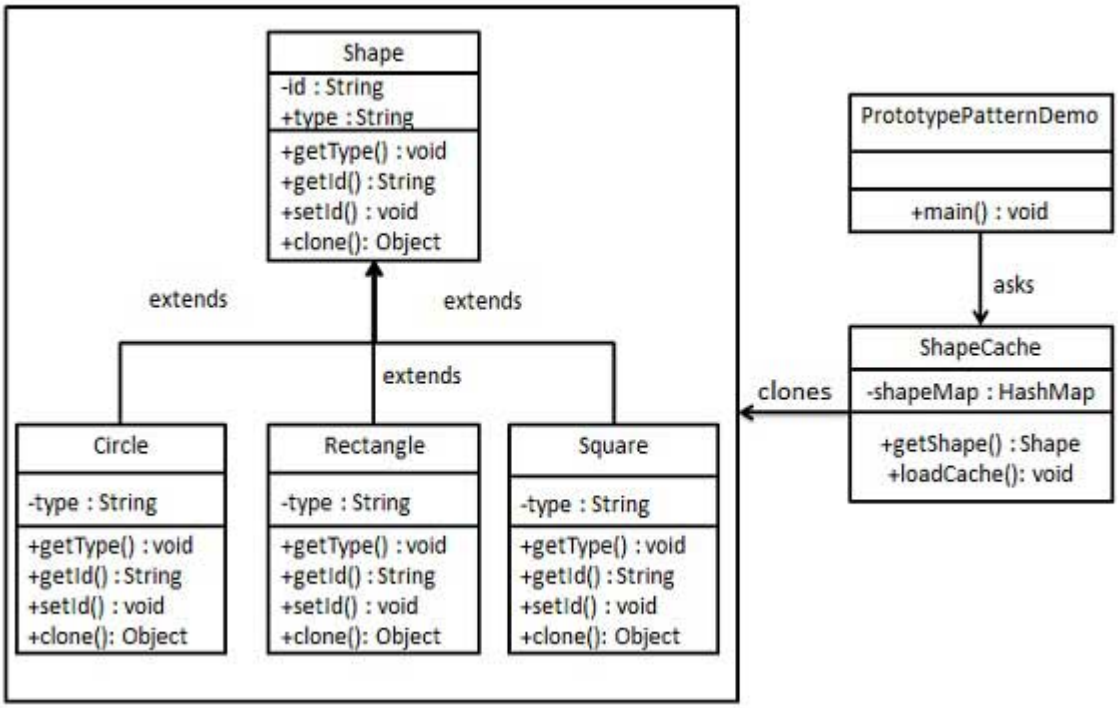
1、资源优化场景。
2、类初始化需要消化非常多的资源，这个资源包括数据、硬件资源等。
3、性能和安全要求的场景。
4、通过 new 产生一个对象需要非常繁琐的数据准备或访问权限，则可以使用原型模式。
5、一个对象多个修改者的场景。
6、一个对象需要提供给其他对象访问，而且各个调用者可能都需要修改其值时，可以考虑使用原型模式拷贝多个对象供调用者使用。
7、在实际项目中，原型模式很少单独出现，一般是和工厂方法模式一起出现，通过 clone 的方法创建一个对象，然后由工厂方法提供给调用者。原型模式已经与 Java 融为浑然一体，大家可以随手拿来使用。

注意事项

与通过对一个类进行实例化来构造新对象不同的是，原型模式是通过拷贝一个现有对象生成新对象的。浅拷贝实现 Cloneable，重写，深拷贝是通过实现 Serializable 读取二进制流。

示例实现

我们将创建一个抽象类 Shape 和扩展了 Shape 类的实体类。下一步是定义类 ShapeCache，该类把 shape 对象存储在一个 Hashtable 中，并在请求的时候返回它们的克隆。
我们的演示类使用 ShapeCache 类来获取 Shape 对象。



注意上边的抽象类Shape需要实现Clonable接口并重写clone()方法：

```
public Object clone() {
    Object obj = null;
    try {
        obj = super.clone();
    } catch(CloneNotSupportedException e) {
        e.printStackTrace();
    }
    return obj;
}
```

ShapeCache使用一个shapeMap来缓存形状对象，初始化时需要调用loadCache()来加载预置对象，调用getShape()时会根据shapeId来获取对应的对象的克隆。

```
public class ShapeCache {
    private static Hashtable<String, Shape> shapeMap = new Hashtable<String,
Shape>();
    public static Shape getShape(String shapeId) {
        Shape cachedShape = shapeMap.get(shapeId);
        return (Shape) cachedShape.clone();
    }
    public static vlod loadCache() {
        Circle circle = new Circle();
        circle.setId("1");
        shapeMap.put(circle.getId().circle);

```

```
        Square aquare = new Square();
        square.setId("2");
        shapeMap.put(square.getId(), square);

        Rectangle rectangle = new Rectangle();
        rectangle.setId("3");
        shapeMap.put(rectangle.getId(), rectangle);
    }
}
```