1 工厂模式

工厂模式（Factory Pattern）是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。

1.1 介绍

• 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，Factory Method 使一个类的实例化延迟到其子类。

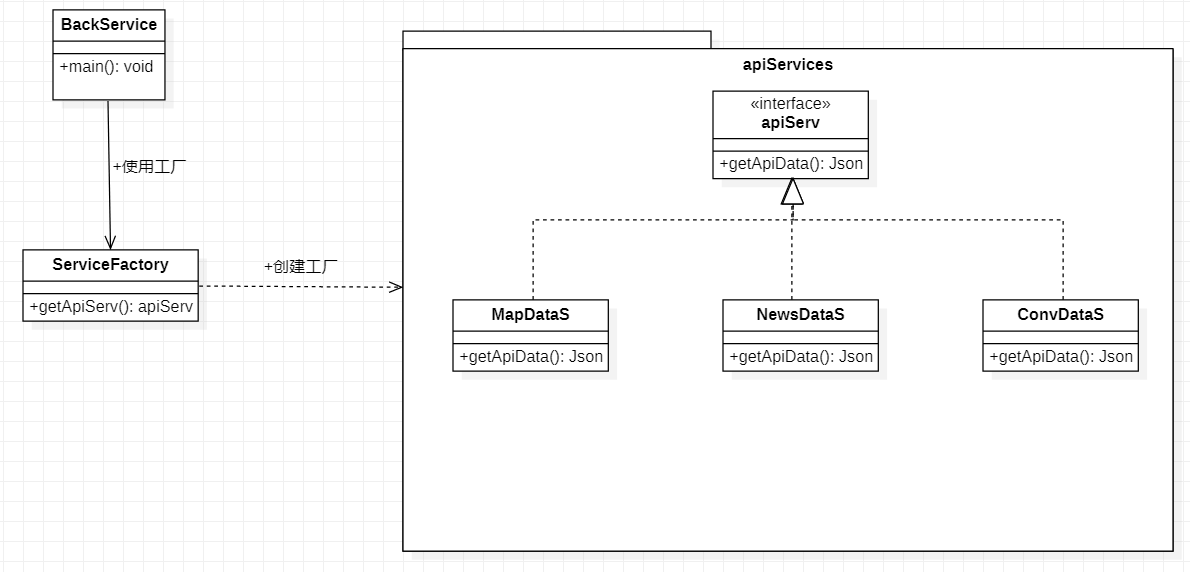
• 主要解决：主要解决接口选择的问题。

• 如何使用：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。

1.2 举例



在我们的工程中，虽然使用了spring boot，服务端的整体架构相对固定，但是仍然可以对一些数据服务进行抽象化处理。比如，可以创建一个获取数据的Service接口和实现Service接口的实体类，工厂类ServiceFactory，预期可以使用ServiceFactory获取Service对象，该对象向ServiceFactory传递信息，以便获取其所需对象的类型。

通过这种方式，在Controller层，可以调用统一的getData接口获取数据，方便对为数众多的获取数据的函数调用进行管理，唯一增加的成本是需要声明获取对应的服务类即可。

1.3 优缺点分析

使用工厂模式对工程代码进行抽象化处理，其优点是十分明确的，可以适用于以下情形：

1. 当一个类不知道它所必须创建的对象的类的时候。

2. 当一个类希望由它的子类来指定它所创建的对象的时候。

3. 当类将创建对象的职责委托给多个帮助子类中的某一个，并且你希望将哪一个帮助子类是代理者这一信息局部化的时候。

当然也存在一定的缺点：每次增加一个获取数据的服务时，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。

2 抽象工厂模式

抽象工厂模式（Abstract Factory Pattern）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。

2.1 介绍

• 意图：提供一个创建一系列相关或相互依赖对象的接口，无需指定它们具体的类。

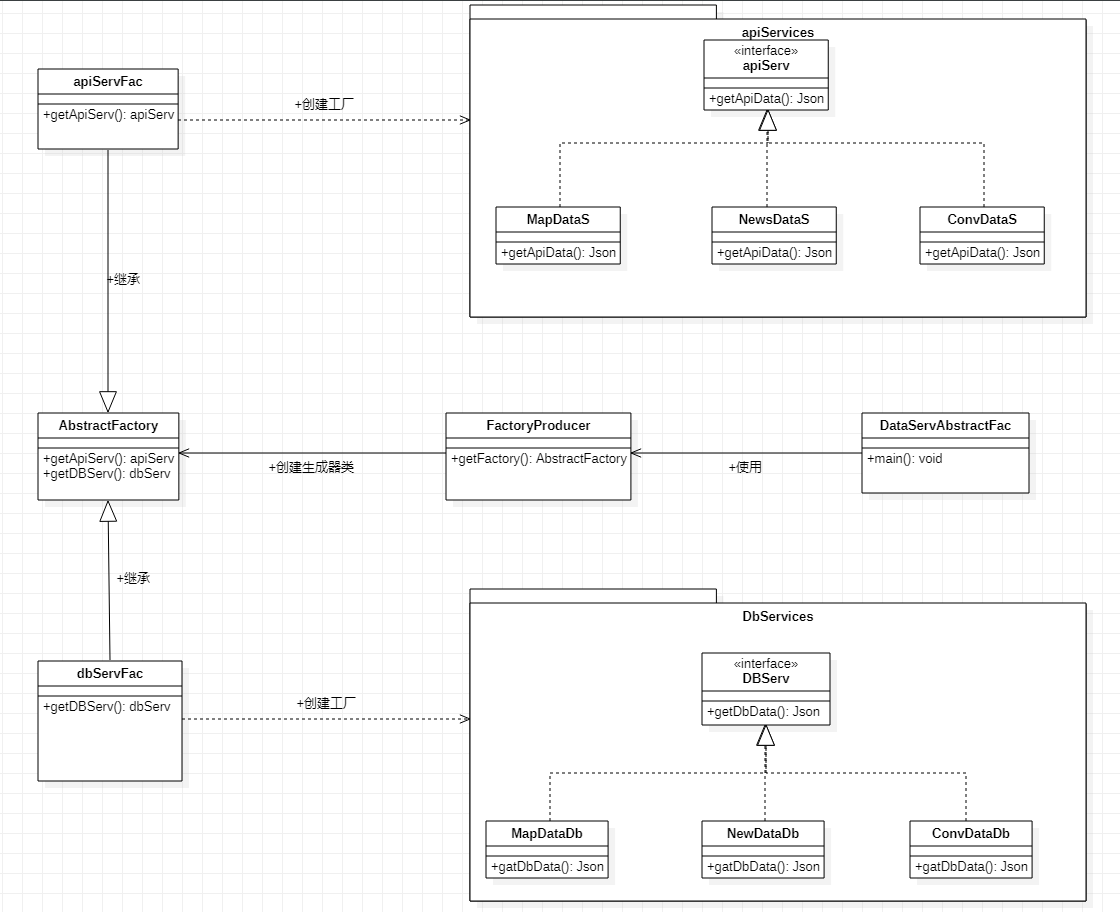
• 主要解决：主要解决接口选择的问题。

• 如何使用：系统的产品有多于一个的产品类，而系统只消费其中某一类的产品。

• 如何解决：在一个产品类里面，定义多个产品。

• 关键代码：在一个工厂里聚合多个同类产品。

2.2 举例



适用性：

1. 一个系统要独立于它的产品的创建、组合和表示时。

2. 一个系统要由多个产品系列中的一个来配置时。

3. 当你要强调一系列相关的产品对象的设计以便进行联合使用时。

4. 当你提供一个产品类库，而只想显示它们的接口而不是实现时。

2.3 优缺点分析

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。

3 单例模式

单例模式（Singleton Pattern）是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。

3.1 介绍

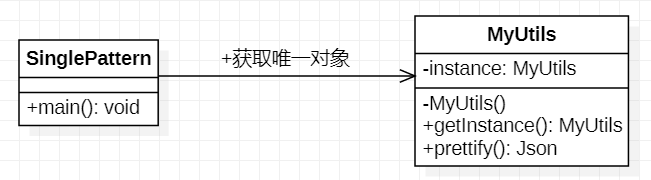
• 意图：保证一个类仅有一个实例，并提供一个访问它的全局访问点。

• 主要解决：一个全局使用的类频繁地创建与销毁。

• 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。

3.2 举例



该类适用于单例模式的原因是，其目的在于格式化json 数据包，任何controller 在返回数据前均需要使用该类，如果不是单例模式，该类会被频繁的创建与销毁，这种效率浪费是没有必要的。该类并不需要对不同的controller 进行特异化处理，因此可以作为单例被广泛使用。

3.3 优缺点分析

适用性：

1. 当类只能有一个实例而且客户可以从一个众所周知的访问点访问它时。

2. 当这个唯一实例应该是通过子类化可扩展的，并且客户应该无需更改代码就能使用一个扩展的实例时。

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。

4 建造者模式

建造者模式（Builder Pattern）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。

4.1 介绍

• 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。

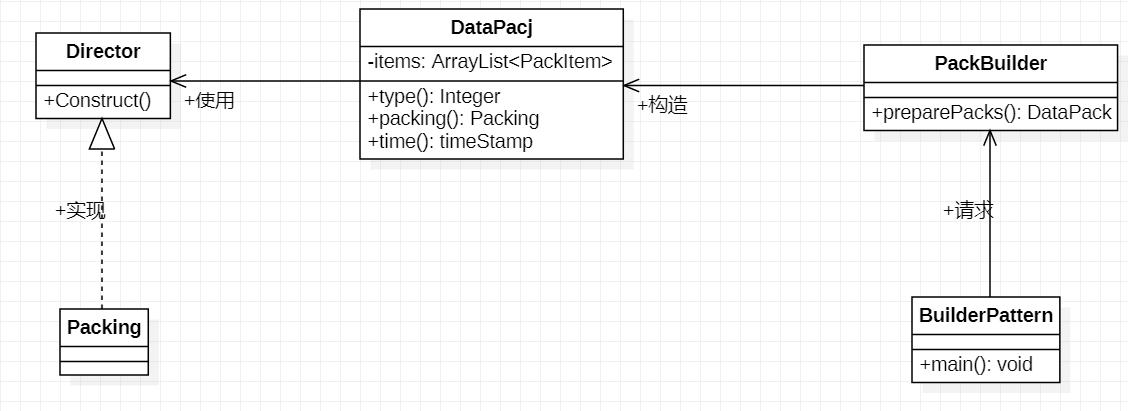
• 如何解决：将变与不变分离开。

• 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。

4.2 举例

当从服务器端从数据库向客户端返回数据时，需要根据数据库内容构建json 数据包，为了使得字段的映射关系清晰，构建json 数据包简洁，选择使用类进行数据处理。

但是客户端不同页面请求的数据内容显然存在不同字段，因此需要根据请求的类型特异性地构建相关类，由于json 数据包的字段较多，类的结构复杂，不同类的组成部分不同，但是组合的算法逻辑相同，因此使用建造者模式是一个比较好的选择。



4.3 优缺点分析

优点：建造者独立，易扩展；便于控制细节风险。

适用性：

1. 当创建复杂对象的算法应该独立于该对象的组成部分以及它们的装配方式时。

2. 当构造过程必须允许被构造的对象有不同的表示时。

缺点：需要生成的对象具有复杂的内部结构；需要生成的对象内部属性本身相互依赖

5 原型模式

原型模式（Prototype Pattern）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。

5.1 介绍

• 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。

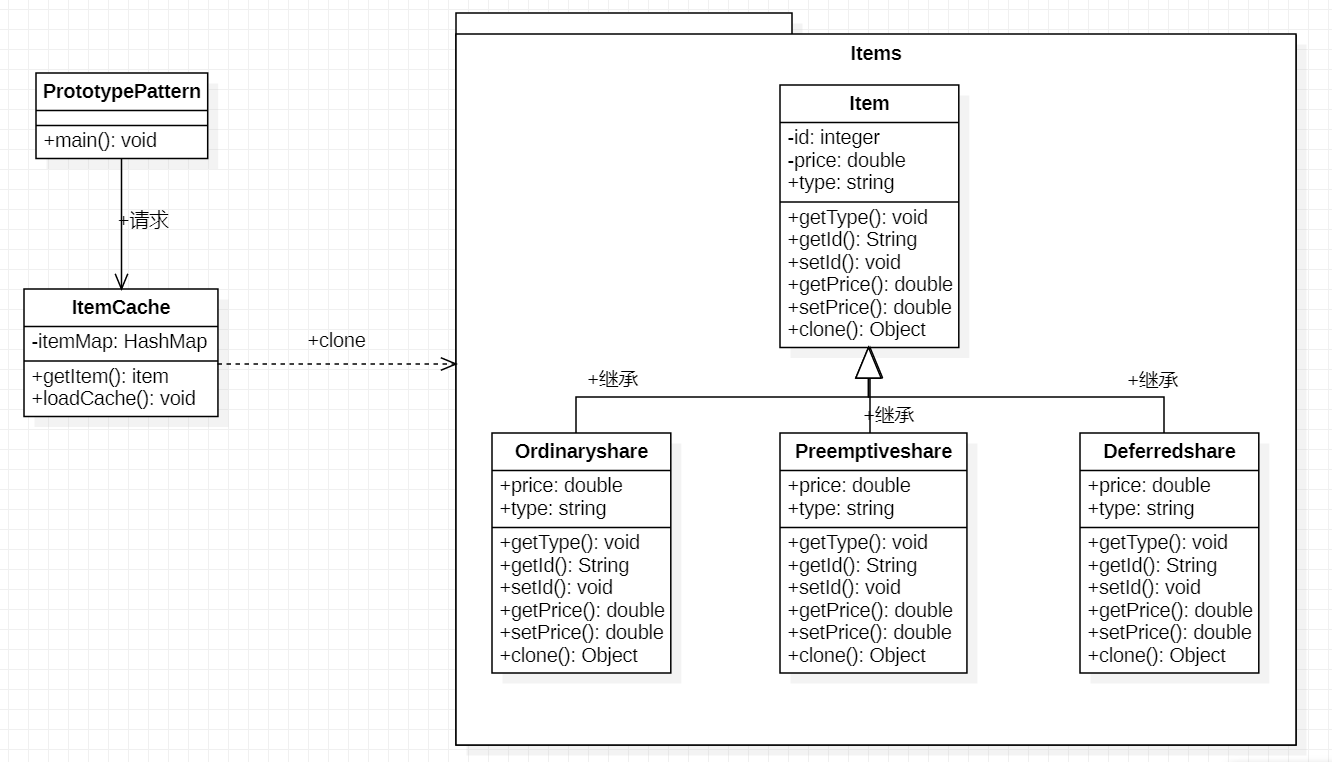
4、当一个类的实例只能有几个不同状态组合中的一种时。建立相应数目的原型并克隆

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：实现克隆操作，在JAVA 继承Cloneable，重写clone()。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。

5.2 举例



有三种不同的股票类型，显然每个股票类型实例只能是一种股票类型，但是每个实例都需要维护id, type和price字段，因此必须考虑使用继承基类的方式进行处理，使用原型模式可以规避掉每次冗余的实例初始化过程，而只需要进行clone即可。客户端的订单是动态变化的，不能保证每次的股票类型种类是相同的但是维护所有的可能性或者每次均创建一个新的实例，相对麻烦。使用clone，处理成本会降低许多。

5.3 优缺点分析

优点：

a. 性能提高

b. 逃避构造函数的约束

缺点：

a. 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。

b. 必须实现Cloneable 接口。