单例模式
- 概述
- 线程安全问题
- 总结
  - 优点
  - 缺点

工厂模式
- 简单工厂模式
- 抽象工厂模式
- 使用场景
- 总结
  - 区别
  - 优点
  - 缺点

单例模式和工厂模式是平时开发中最常见的两种设计模式，即便没有系统学习过设计模式的开发者，也必然对此不陌生。这两种设计模式是我认为最容易理解，最简单的设计模式，因此放在同一篇文章中。

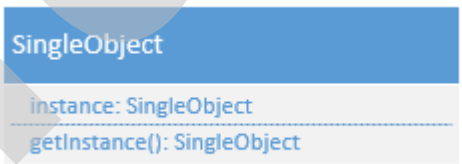
单例模式

概述

> 单例模式确保一个类只有一个实例，并提供一个全局访问点

我们常常希望某个对象实例只有一个，不想要频繁的创建和销毁对象，浪费系统资源，最常见的就是IO，数据库的连接，redis连接等对象，完全没有必要创建多个一模一样的对象，一个足矣。

类图：



正如上面类图所示，单例模式就是这么简单，用静态变量instance将实例保存起来，然后调用getInstance()时直接返回instance变量，这就是单例模式。

虽然单例模式简单，但是也不要小看它，其中还有许多细节需要我们注意。

线程安全问题

单例模式代码如下：

```
public class SingleObject {

    private static SingleObject instance;

    public SingleObject(){}

    public static SingleObject getInstance(){
        if(instance == null){
            instance = new SingleObject();
        }
        return instance;
    }
}
```

但是上面代码会有一个问题，当多个线程同时调用 `getInstance()` 方法时，可能会产生多个 `instance` 实例，因此这种方式并不是真正的单例。

为了解决线程安全问题，我们只需要在 `getInstance()` 方法上使用 `synchronized` 关键字给线程加锁即可。

```
public class SingleObject {

    private static SingleObject instance;

    public SingleObject(){}

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

`synchronized` 的作用是加锁，当多个线程同时调用 `getInstance()` 时，只有一个线程能进入，其他线程会等待进入的线程出来之后在一一进入，这样就能保证 `instance` 实例是唯一的。这才是真正的单例。

不过这并不是完美的解决方案，只要是锁，必然有性能损耗问题。而且对于上面的代码，其实我们只需要在线程第一次访问时加锁即可，之后并不需要锁，锁给我们带来了系统资源浪费。

所以又有了新的解决方案。上面两种方式都是在 `getInstance()` 方法中创建实例，也就是说在要调用的时候才创建实例，这种方式被称为“懒汉式”，说实话这个词不知道谁给命名的，实在有点难听，让我总是不自觉地想到“老汉”二字。

咱们还是用它的英文名，叫 `lazy loading` ，也就是延迟加载。

新的解决方案是 `not lazy loading` ，在类加载时就创建好了实例：

```
public class SingleObject {

    private static SingleObject instance = new SingleObject();

    public SingleObject(){}

    public static  SingleObject getInstance(){
        return instance;
    }
}
```

这种方式就可以保证实例唯一。

除了上面的几种方式，还有一种叫 `double-checked locking` （双重检查加锁）

这种方式主要用到两个关键字 `volatile` 和 `synchronized` ， `synchronized` 已经解释过了，就不再多说，而 `volatile` 关键字许多人不了解，没关系，我们先看代码：

```
public class SingleObject {

    private volatile static SingleObject instance;

    public SingleObject(){}

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

`volatile` 关键字简单来说就是可以保证instance变量在被其中一个线程new出来时，其他线程可以立即看到结果并正确的处理它。对 `volatile` 有兴趣的朋友可以自行度娘。

这种方式的单例模式可以大大的减少锁所带来的性能损耗。

总结

综合上面所说，一般情况我们使用下面这种方式就可以了，如有特殊需求，也可以使用双重检查加锁方式（其实双重检查加锁方式笔者也没使用过）

```
public class SingleObject {

    private static SingleObject instance = new SingleObject();

    public SingleObject(){}

    public static  SingleObject getInstance(){
        return instance;
    }
}
```

优点

使用单例模式，对象在内存中只有一个实例，并且无需频繁的创建和销毁对象，大大的减少了性能的损耗

缺点

暂时想不出什么明显的缺点，哈哈

工厂模式

简单工厂模式

程序猿最擅长的就是new一个对象，比如没有女朋友就自己new一个。但是new一个对象不是那么容易的，有许多问题需要我们去考虑。比如我们要根据不同条件去new不同的对象：

```
Girl girl = null;
if(I like fat girl){
    girl = new FatGirl();
}else if(I like thin girl){
    girl = new ThinGirl();
}
```

上面代码人人都会写，但是存在一些问题：

1. 以后有其他新条件或其他新的Girl子类了，我们又得修改这段代码，违背了开闭原则
2. 我并不太想知道new一个女孩的具体过程，我希望告诉你我想要是什么样的，直接给我结果就好

于是我们将new对象的代码封装起来，对调用者隐藏细节：

```
public class Factory {

    public Girl createGirl(String whatYouLike){

        Girl girl = null;


        if(whatYouLike == "fat"){
```

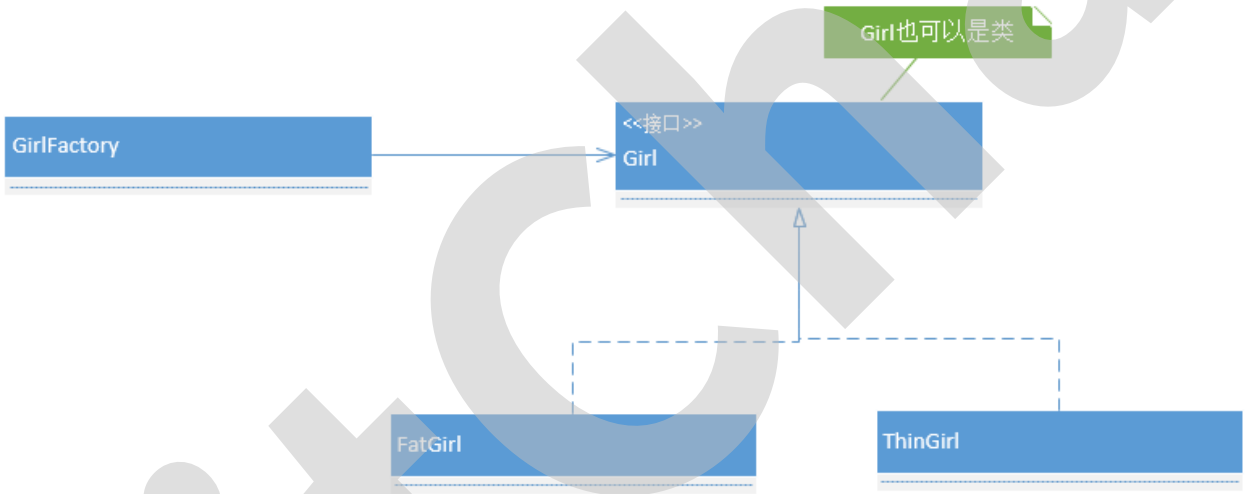
```
            girl = new FatGirl();
        }else if(whatYouLike == "thin"){
            girl = new ThinGirl();
        }
        return girl;
    }
}
```

Girl类的具体代码就不写了，看下面类图就明白了。

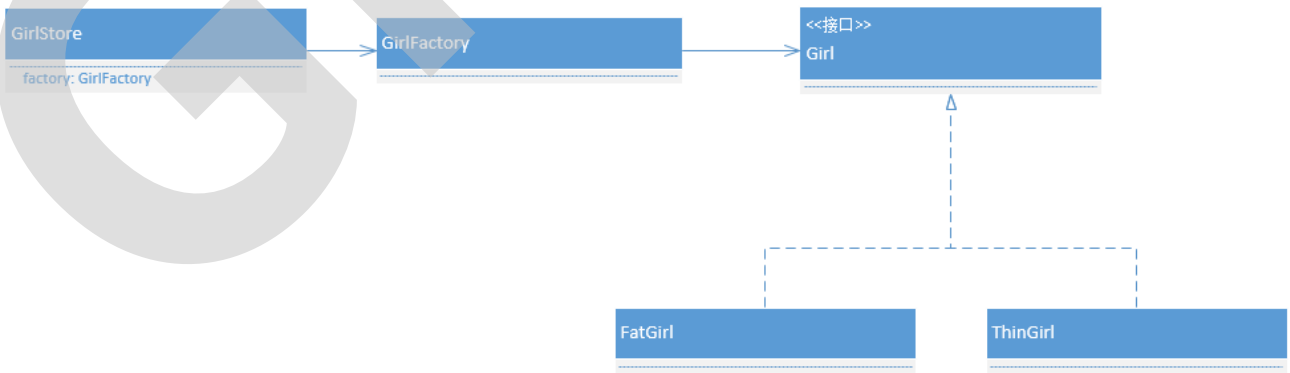
这下程序员开心了，想要什么样的女孩一键搞定：

```
Girl girl = (new GirlFactory()).createGirl("fat");
```

没错，上面这短短几行代码就是简单工厂模式了。其类图如下：



不过我们有时候不会直接调用Factory，而会借助另外一个类使用组合的方式来调用Factory，这样的好处在于可以方便地选择用哪种工厂（假如有多个工厂可以选择）：



借助的那个类可以叫做工厂提供者，代码如下：

```
public class GirlStore {
```

```
    GirlFactory factory;

    /**
     * 可以动态选择工厂
     * @param factory
     */
    public GirlStore(GirlFactory factory){
        this.factory = factory;
    }

    public Girl createGirl(String whatYouLike){
        return factory.createGirl(whatYouLike);
    }

}
```

调用的时候调用GirlStore而不是Factory：

```
GirlStore store = new GirlStore(new GirlFactory());
store.createGirl("thin");
```

有人说增加了 `GirlStore` 类更麻烦了。增加 `GirlStore` 类的目的是为了扩展性，如果你很确定你的这块业务将来基本不怎么扩展，那么请你怎么简单怎么来，直接使用Factory类就好了，甚至直接new对象就好了。

设计模式是为了解决问题而诞生的，不是为了增加代码复杂性而诞生的。不要为了使用设计模式而使用设计模式。

综上，工厂模式简单来说就是将new对象的过程给封装起来了，调用者无需关注。但是前面说的两个问题目前只解决了第二个，如果我们要增加条件和Girl的子类时，还得修改工厂类。但抽象工厂模式不存在这个问题。

抽象工厂模式

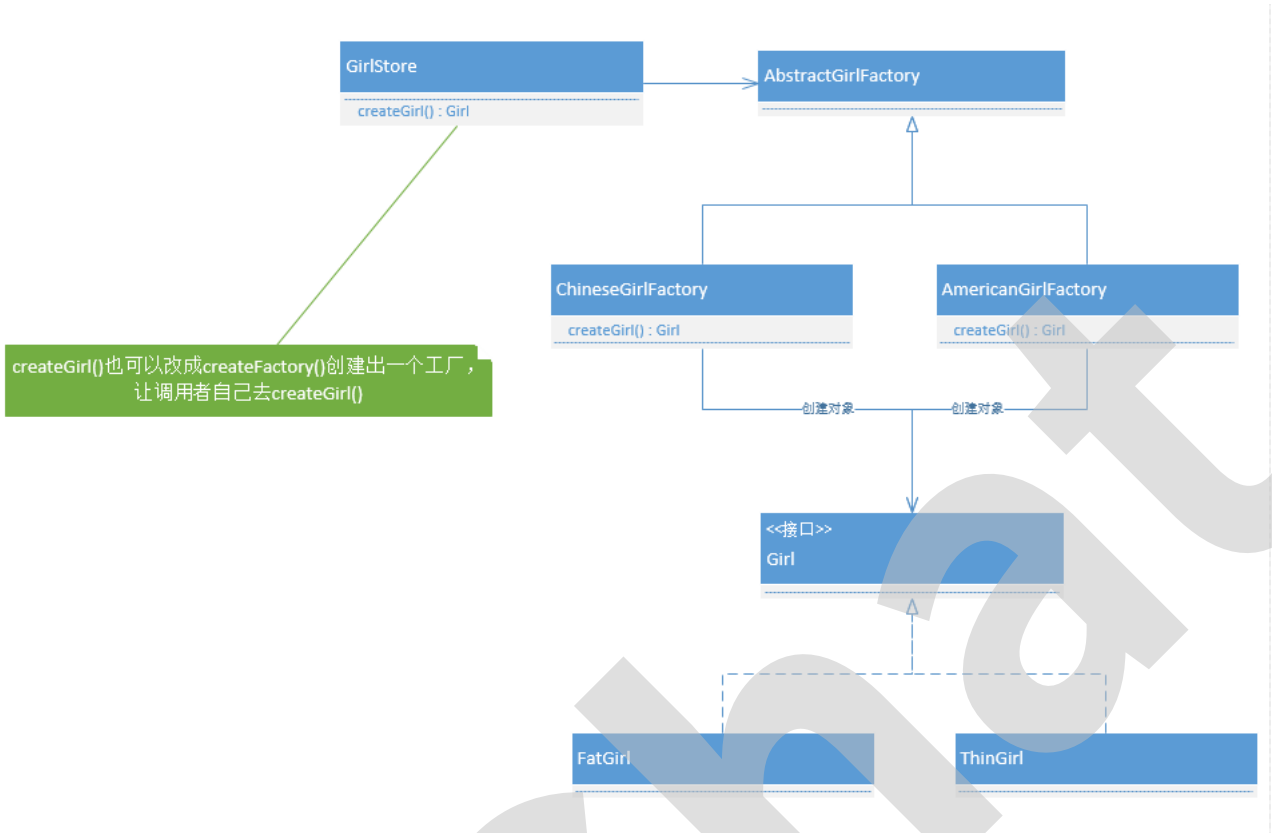
判断你的代码是否遵循了"开闭原则"的一个很重要的指标是：

> 当你想要扩展功能的时候，是否需要修改已有代码？

如果不需要，那么恭喜你写的一手优雅的好代码。

抽象工厂模式是在简单工厂的基础上将未来可能需要修改的代码抽象出来，通过继承的方式让子类去做决定。

假如某某程序猿现在不满足于选择girl的胖瘦，他的口味有点特殊，喜欢美国产的长发姑娘，遵循开闭原则，我们需要使用抽象工厂，类图如下：



在抽象工厂中，我们将工厂抽象出来，由具体的子类工厂来决定生产什么样的对象。ChineseGirlFactory和AmericanGirlFactory分别生产不同种类的girl。girl是产品，而同一个工厂生产出来的girl我们称之为同一个产品族。

具体代码如下：

抽象工厂：

```
public abstract class AbstractGirlFactory {
    /**
     *
     * @param whatYouLike 高矮胖瘦等类型
     * @return
     */
    public abstract Girl createGirl(String whatYouLike);

}
```

美国工厂：

```
public class AmericanGirlFactory extends AbstractGirlFactory{

    @Override
    public Girl createGirl(String whatYouLike) {
        Girl girl = null;

        if(whatYouLike == "fat"){

            girl = new FatGirl();
```

```
        }else if(whatYouLike == "thin"){
            girl = new ThinGirl();
        }
        return girl;
    }
}
```

中国工厂：

```
public class ChineseGirlFactory extends AbstractGirlFactory{

    @Override
    public Girl createGirl(String whatYouLike) {
        Girl girl = null;
        if(whatYouLike.equals("longhair")){
            //girl = new LongHairGirl();
        }
        return girl;
    }
}
```

工厂提供者：

```
public class GirlStore {

    AbstractGirlFactory factory;

    /**
     * 可以动态选择工厂
     * @param factory
     */
    public GirlStore(AbstractGirlFactory factory){
        this.factory = factory;
    }

    public Girl createGirl(String whatYouLike){
        return factory.createGirl(whatYouLike);
    }

    //也可以返回工厂，让客户端自己去createGirl()
    //public abstract AbstractGirlFactory createFactory(String country);

}
```

客户端调用：

```
GirlStore store = new GirlStore(new AmericanGirlFactory());
Girl girl = store.createGirl("longhair");
```

在这里中国工厂和美国工厂都是用于创建Girl，其实也可以分别创建不同的产品，一个创建Girl，另一个用于创建Computer等。设计模式是灵活的，不同场景下，具体的使用方式也会有所差别。

上面就是抽象工厂模式了。大家思考一下，如果以后还需要增加新的Girl类型，Girl对象，我们是不是只需要增加工厂类就行了，而不用修改原有的工厂类里的代码。

看到这里也许有些朋友懵了，我创建一个girl，只要`new xxxGirl()`一行代码就能搞定的事给你整这么多代码出来？？

会有这种想法是正常的，每一个学习设计模式的人都会产生这种想法，并且这种想法本身是对的。笔者一再强调，如果能用简单的方式解决，就不要用复杂的方式。如果一行new就能满足你的所有需求，那么请**不要画蛇添足使用设计模式。**

那到底什么情况下用呢？

使用场景

第一：如果new对象的代码需要增加或修改，有用工厂模式的话就只需要修改工厂中的代码即可。否则就需要大面积的修改代码了。这一点既是优点，也是缺点，因为它违背了开闭原则。

第二：回顾上面抽象工厂的例子，我们创建了中国工厂（ChineseFactory）和美国工厂（AmericanFactory），他们都可以创建Girl，那么我们是不是可以通过更换工厂类来动态更换所有的Girl？比如所有中国工厂生产出来的girl都是黄皮肤的，美国工厂生产的girl都是白皮肤的。现在系统中所有Girl都是中国girl，那么我们就可以通过在GirlStore中更换工厂类，直接将所有girl都变成美国白皮girl。具体的使用场景如更换整套应用程序的皮肤等。

第三：设计模式的目的之一是让你的代码具有可扩展性，如果你new对象这一块的业务将来需要扩展，哪怕只是**可能**，那么也不要犹豫，用工厂模式吧。

至于使用简单工厂还是抽象工厂，还得具体情况具体分析（简单工厂用的较多）。记住两句话：

1. 有简单的方法就不要用复杂的方法。
2. 要灵活使用设计模式，不要被条条框框给限制住了

总结

区别

简单工厂和抽象工厂有些区别，除了结构上的区别，主要区别在于使用场景不同。

- 简单工厂用于创建单一产品，将所有子类的创建过程集中在一个工厂中，如要修改，只需修改一个工厂即可。简单工厂经常和单例模式一起使用，例如用简单工厂创建缓存对象（文件缓存），某天需要改用redis缓存，修改工厂即可。
- 抽象工厂常用于创建一整个产品族，而不是单一产品。通过选择不同的工厂来达到目的，其优势在于可以通过替换工厂而快速替换整个产品族。例如上面的例子美国工厂生产美国girl，中国工厂生产中国girl。

优点

客户端与具体要创建的产品解耦，扩展性和灵活性高

缺点

增加要创建的对象时，需要增加的代码较多，会使系统变得较为复杂