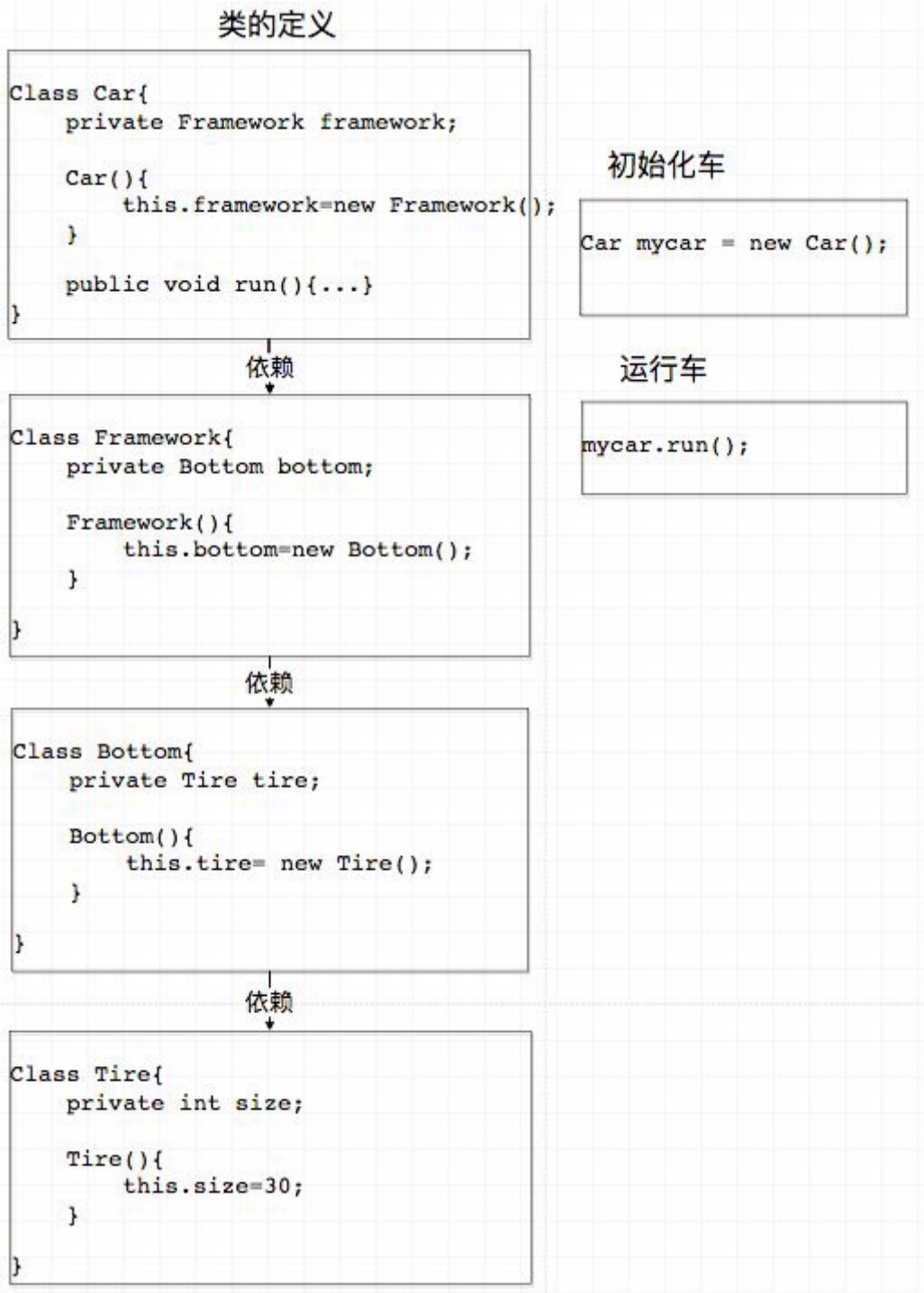
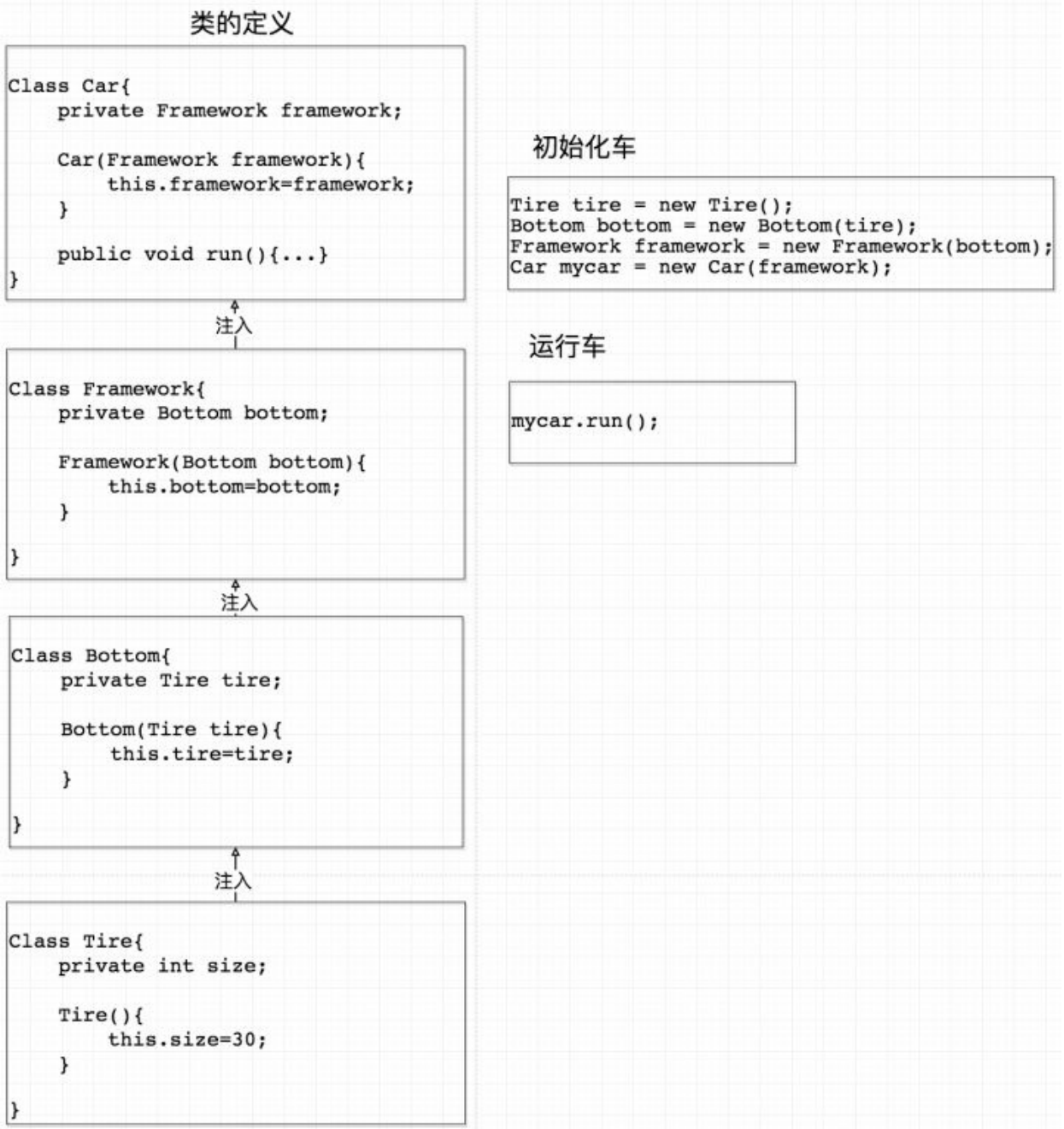
上层依赖下层。底层一改全改



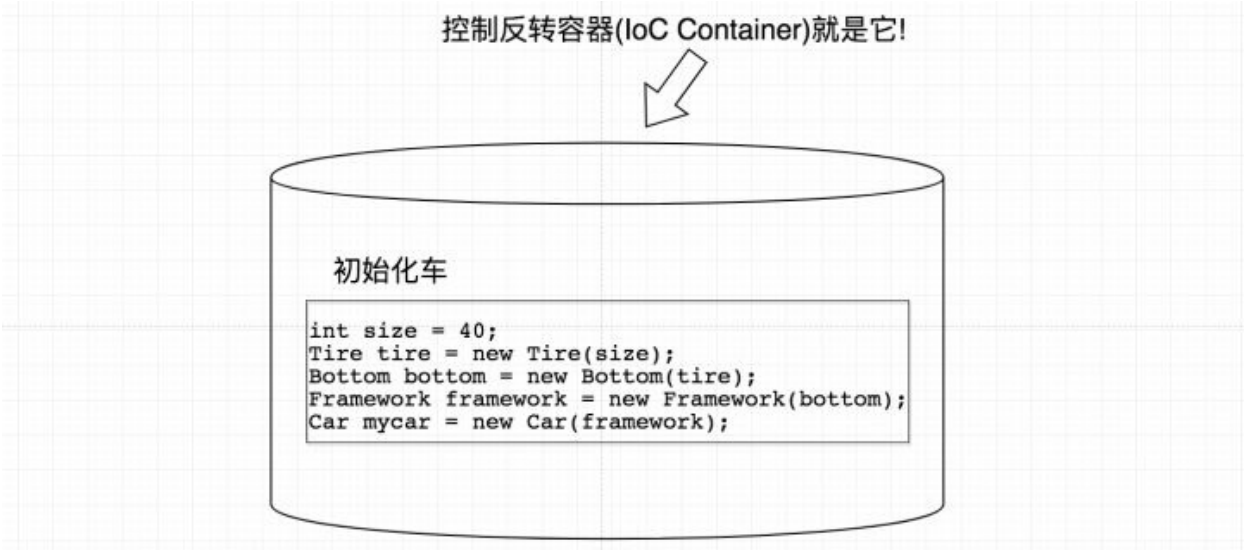
依赖注入：采用的构造函数传入的方式进行。还有另外两种方法：Setter传递和接口传递。这里就不多讲了，核心思路都是一样的，都是为了实现控制反转。


类的定义
Class Car{
private Framework framework;
Car(Framework framework){
this.framework=framework;
}
public void run(){...}
}
注入
Class Framework{
private Bottom bottom;
Framework(Bottom bottom){
this.bottom=bottom;
}
}
注入
Class Bottom{
private Tire tire;
Bottom(Tire tire){
this.tire=tire;
}
}
注入
Class Tire{
private int size;
Tire(){
this.size=30;
}
}
初始化车
Tire tire = new Tire();
Bottom bottom = new Bottom(tire);
Framework framework = new Framework(bottom);
Car mycar = new Car(framework);
运行车
mycar.run();

修改不必改上层类

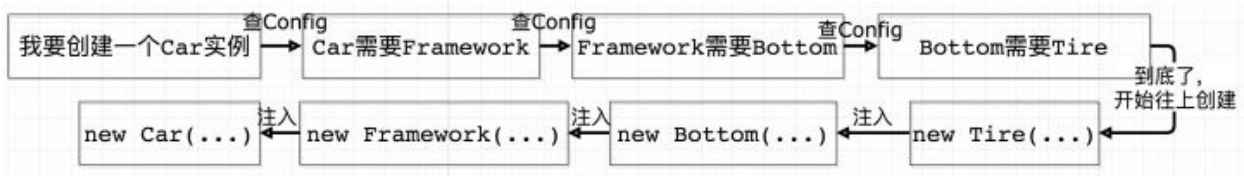
有利于不同组的协同合作和单元测试：比如开发这四个类的分别是四个不同的组，那么只要定义好了接口，四个不同的组可以同时进行开发而不相互受限制；而对于单元测试，如果我们要写Car类的单元测试，就只需要Mock一下Framework类传入Car就行了，而不用把Framework, Bottom, Tire全部new一遍再来构造Car。

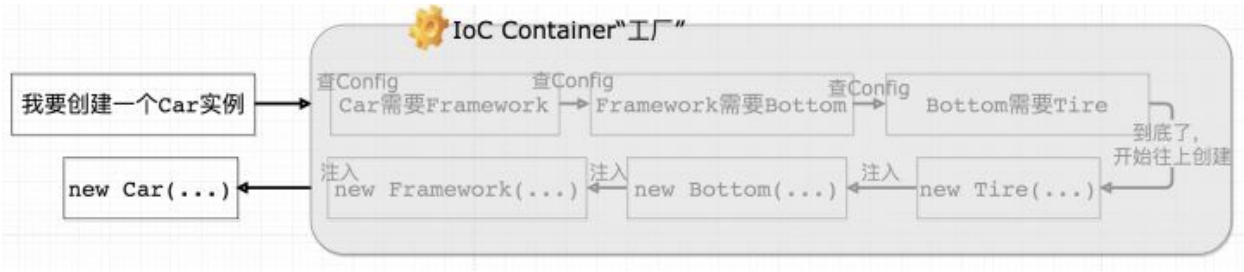
控制反转容器(IoC Container)



采用了依赖注入，在初始化的过程中就不可避免的会写大量的new。这里IoC容器就解决了这个问题。这个容器可以自动对你的代码进行初始化，你只需要维护一个Configuration（可以是xml可以是一段代码），而不用每次初始化一辆车都要亲手去写那一大段初始化的代码。

我们在创建实例的时候不需要了解其中的细节。





这个架构要求你在写class的时候需要写相应的Config文件，所以你要初始化很久以前的Service类的时候，前人都已经写好了Config文件，你直接在需要用的地方注入这个Service就可以了。