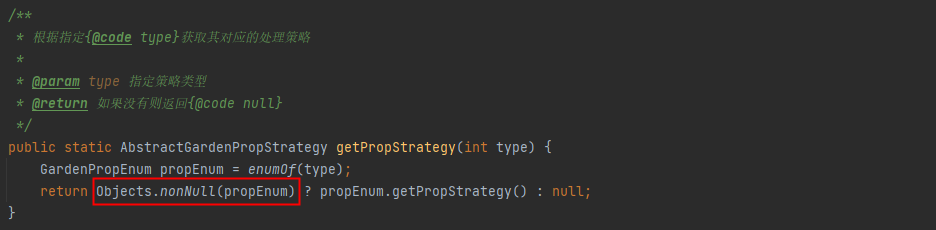
## 避免空指针异常

### 判空

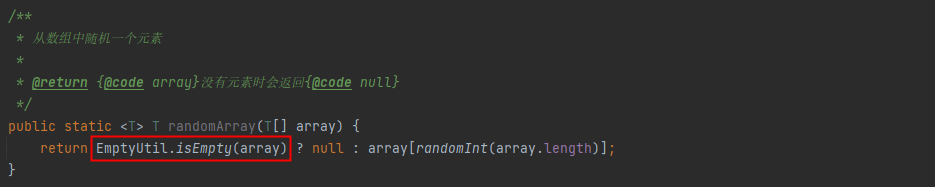
【强制】所有要直接调用对象方法的位置，尽量增加“判空”操作。推荐使用JDK源生的工具类java.util.Objects中的方法进行操作，EmptyUtil中也有isNull(Object)和nonNull(Object)统一。

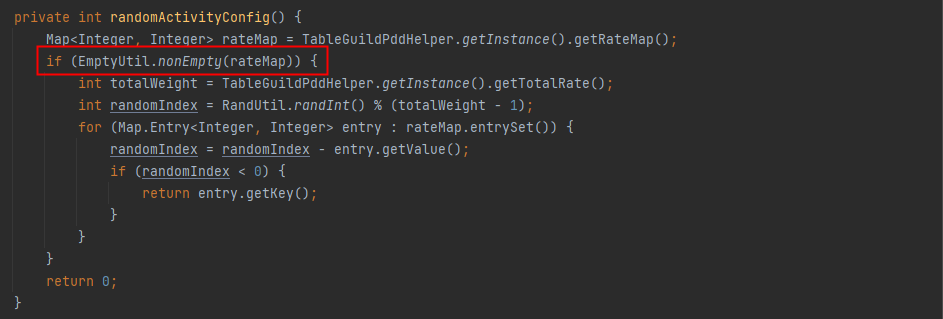


### **EmptyUtil**

【强制】新增EmptyUtil工具类，聚合“Null”和“isEmpty()”的判断。该类的方法签名主要包含“isEmpty(X x)”和“nonEmpty(X x)”，其中X包括：所有基本类型数组，String字符串，所有对象数组，所有java.util.Collection<T>的实现类，所有java.util.Map<K,V>的实现类。

【强制】该方案还降低从“A == null || A.isEmpty()”修改成“A != null && !A.isEmpty()”的修改成本和出错概率，直接从“EmptyUtil.isEmpty()”修改成“Empty.nonEmpty()”即可。

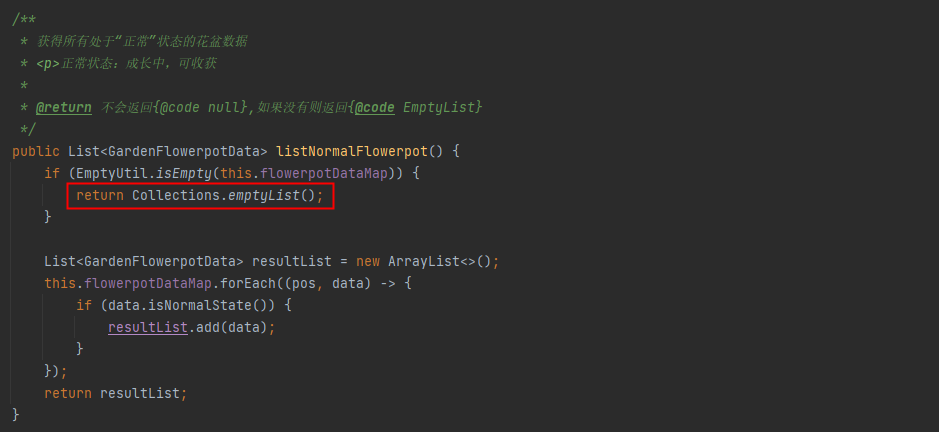




### **非空返回值**

【强制】当结果返回值是集合时，在需要返回“Null”时，尽量用“空集合”代替。这样即使调用处不进行“判空”，也可以避免NPE。

注意，此处返回的“空集合”尽量使用jdk源生Collections内的方法，而不要自己创建一个新的“空集合”。



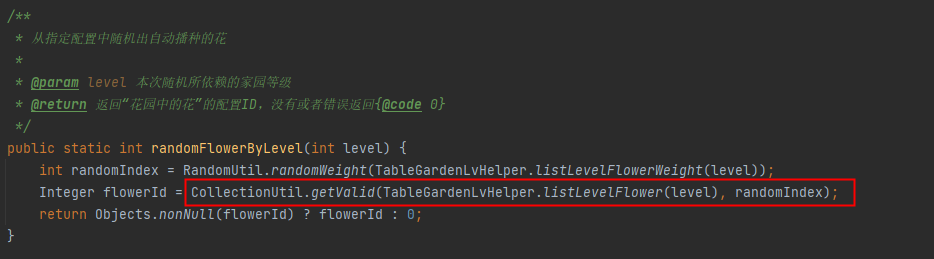
## **避免数组越界异常**

### CollectionUtil

【强制】建议使用CollectionUtil.getValid(list, index)方法，代替直接操作list.get(index)。该方法会直接对list进行必要的检查，以及对索引index进行检查，在满足条件的时候，取出指定位置的元素，否则直接返回null。调用处对返回值“判空检查”即可（需要注意，这一步判空并没有增加负担，即使直接使用list.get(index)，也需要对返回值进行校验）。



可用下面代码替换，直接获取集合指定位置的元素，不满足索引要求直接返回Null。



### ArrayUtil

【强制】ArrayUtil中有相同作用的public static <T> T getValid(T[] array, int index)方法提供。

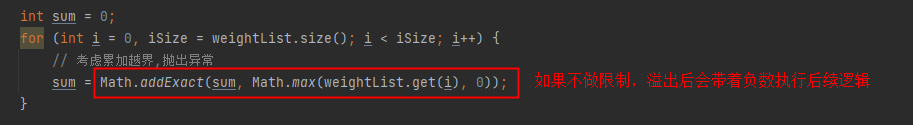


## **避免数据溢出错误**

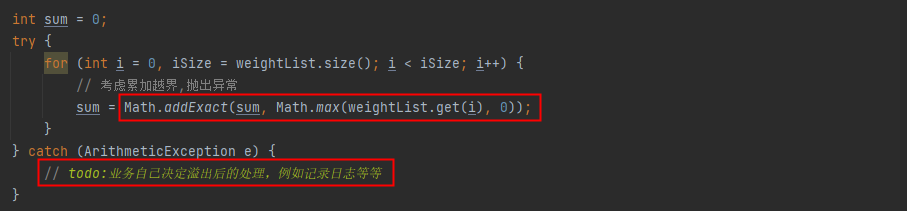
### **Math类库**

【强制】当业务不允许数值溢出时，建议按照溢出校验。推荐使用JDK源生的java.lang.Math类中的“溢出校验”方法。

其中“加，减，乘，自增”分别对应，“Math.addExact()”，“Math.subtractExact()”，“Math.multiplyExact()”，“Math.incrementExact()”方法。应对溢出的方法是，抛出“ArithmeticException”异常，该异常是运行时异常，调用处可以自行决定是否捕获处理。不捕获保底退出当前方法栈，避免错误数据继续执行方法。

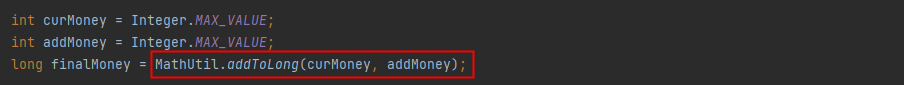


业务还可以捕获这个异常，去做异常处理：



### **MathUtil**

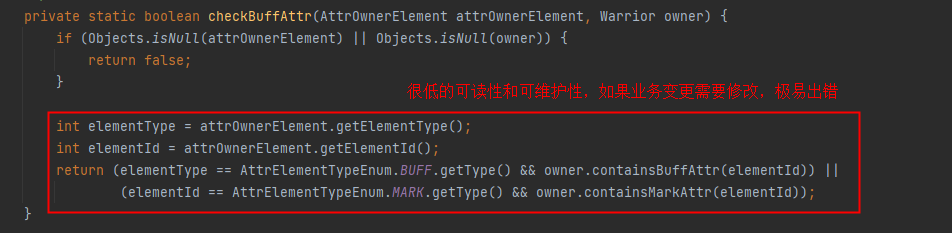
【推荐】当业务逻辑明确数值可能溢出时，且需要保证正确结果时，建议直接使用MathUtil工具类中的“MathUtil.addToLong(int a, int b)”，“MathUtil.substructToLong(int a, int b)”，“MathUtil.multiplyToLong(int a, int b)”保证正确结果，这些方法会直接返回long值保证结果。



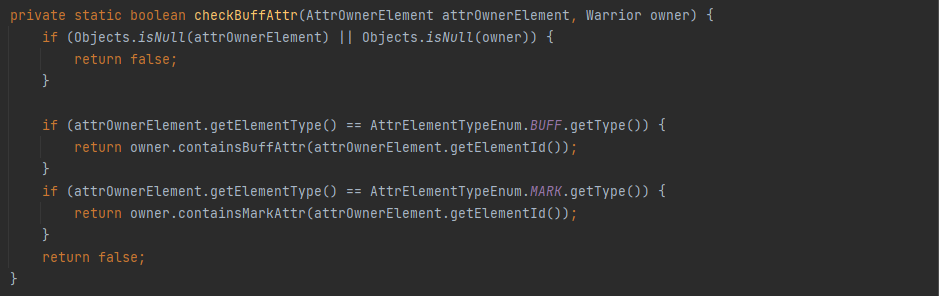
## **避免布尔式的错误**

### **降低复杂度**

【推荐】避免“连续的布尔表达式”:

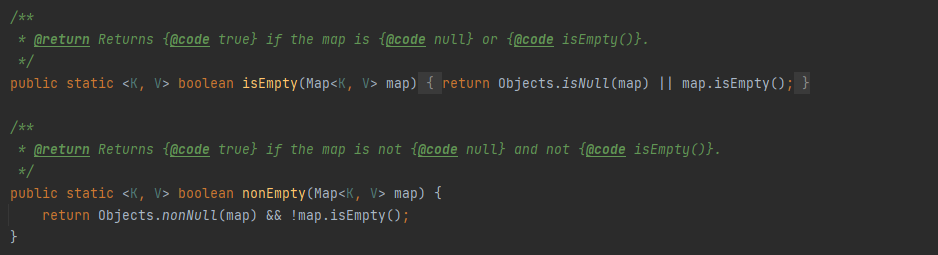


建议不使用连续的布尔表达式，而尽可能的将布尔判断分离：



### **封装“！”**

【推荐】此处的“封装”指的是减少业务代码直接对“!”的使用，而统一使用底层或者各功能模块统一的接口，减低出错概率和维护成本。

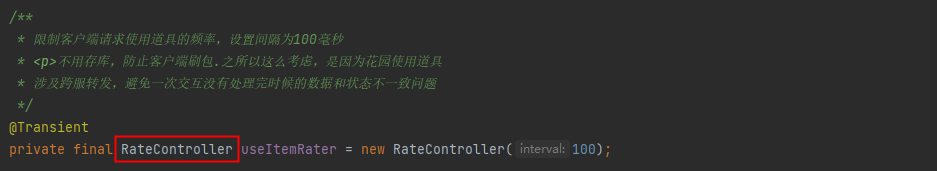


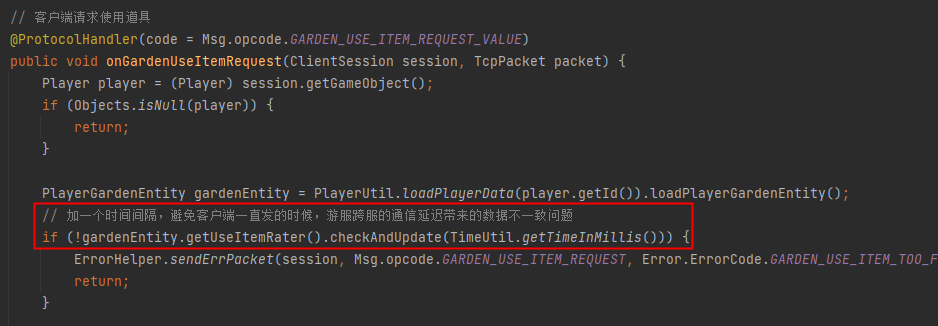
## **强校验客户端输入**

### **频率**

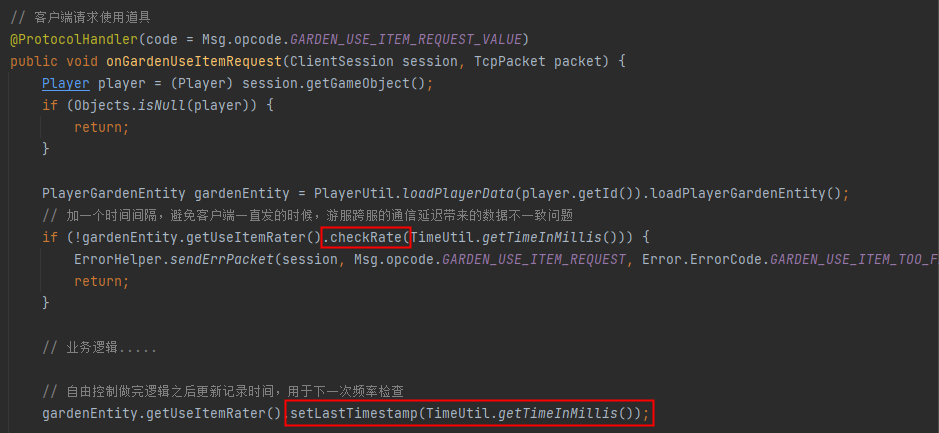
【推荐】请求频率校验，对于一些影响较大，数据较大，耗时较旧等敏感操作，直接限制客户端请求频率，一定时间间隔内不处理第二次请求。注意，该限制仅仅是业务层的限制，并非网络层的限制，请求过快时仍然会收到包，也可能产生消息堆积，只是不处理。

该方案推荐使用RateController，封装的简单频率控制器直接进行判断。其内部方法会检查最新的时间与最后一次记录的时间是否满足设置的频率间隔，并且在满足间隔的时候自动更新记录新的时间。





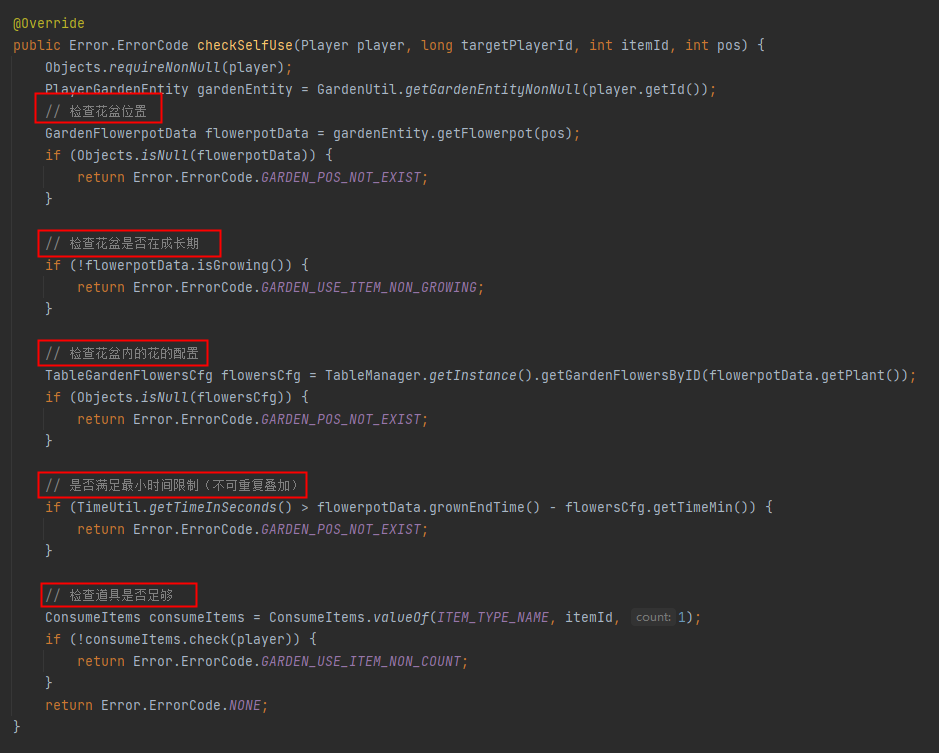
也可调用不自动更新的接口，自己控制更新：



### **数据**

【强制】通讯数据校验。服务器处理客户端通讯数据的态度是“完全不信任”，处理策略是“优先校验，快速失败”。

“完全不信任”意味着尽可能的对客户端请求数据进行“校验”，任何客户端输入都应该由服务器保证其“满足约定”，“满足限制”，“满足上下文”，服务器不应该由于客户端的传值错误而引起逻辑错误。



“优先校验，快速失败”意味着，先校验客户端的全部输入，只有全部校验都通过之后才进行逻辑处理，保证当开始处理逻辑的时候，不会再因为输入不合法而中断。而在进行校验时，一旦遇到错误即可返回，快速失败保证了错误值不会被代入到中间过程。

例如，有校验check1，check2，check3，逻辑处理do1，do2，do3。其顺序应该保证为“check1，check2，check3，do1，do2，do3”。如果顺序为“check1，do1，check2，do2，check3，do3”，将无法保证事务的完整性（除非在检查失败后写回退逻辑）。

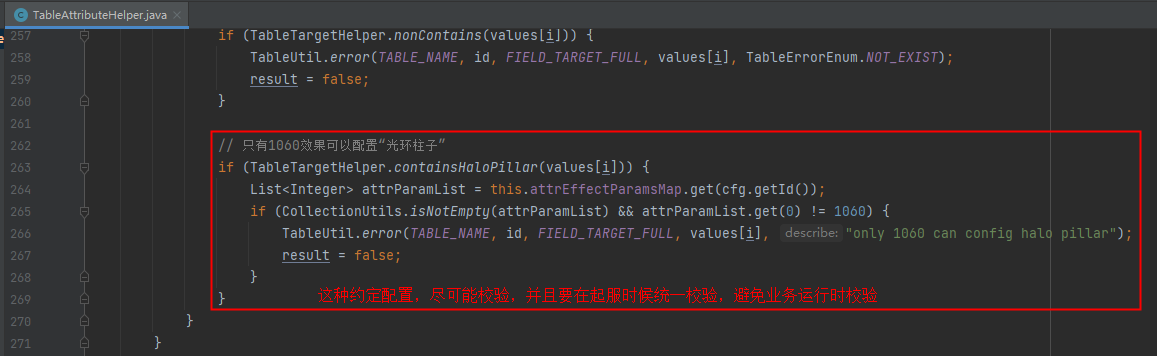


## **强校验配置**

### **数据**

【强制】由于“配置表”往往都是静态固定的，且是内部可控的，所以不需要等到“接口处理”的时候校验，往往起服的时候做统一校验即可。但态度仍旧是“完全不信任”，如果可以的话，建议对每张表数据进行“强约定校验”。即使不能每张表做限制，对于一些“潜规则”类的约定配置，建议必须保证其校验。

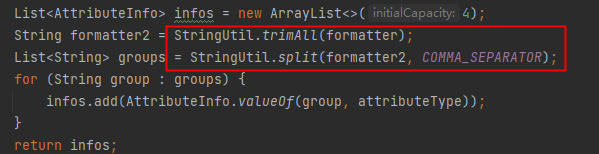
秉承“优先校验，快速失败”原则，起服校验不通过则进行报警或者强制不允许起服，保证错误数据不会被代入到逻辑处理中，原则上，任何配置校验不应该出现在业务逻辑处理过程中，这点对性能有好处，更重要的是，如果要修改某个约定的校验，应该有统一接口，而不是分散到程序的各个角落。



### **二次加工**

【强制】配置数据的二次加工不应该出现在业务运行时处理。配置数据的“静态”特点决定了，对于配置的很多组合，拆解等处理完全可以在恰当的时机（往往是起服时）进行统一处理保存，而后在使用的时候直接获取加工后的数据结构，而不应该在用到对应配置数据的时候再去进行解析。

经过很多场景的性能分析经验，在高频率的业务中去加工配置，往往会对性能造成较大的影响。另外，统一处理保存还能保证接口的统一，方便之后的维护和修改。

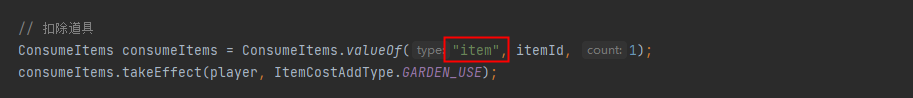


## **避免硬编码**

### **消除魔数**

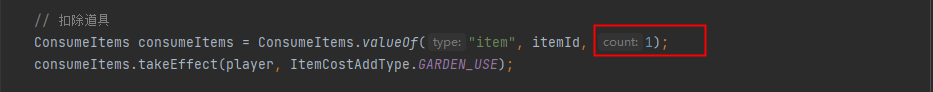
【强制】尽量定义静态常量或者枚举，来表达一些固有含义的字符串和数值变量。根据《Effective Java》一书中的建议，使用“枚举”的优先级要高于“静态常量”。

当程序中大量存在“魔数”时，一旦想要修改，将是很困难的事，很多时候“同一个魔数”还容易造成编写错误，编码不一致等等问题，排查起来也容易被忽略。



### **配置优先**

【强制】任何可配置量，不要直接在程序中写死（硬编码），建议全部走配置表，避免业务变更时引起的修改不全，修改错误等维护问题（修改配置的成本远远小于修改代码）。



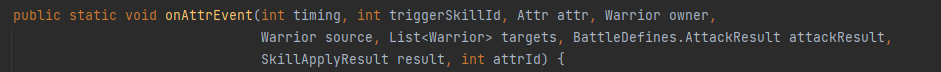
## **避免重复**

【推荐】《重构：改善代码的既有设计》中首先提出的一个“代码坏味道”便是“重复代码”。重复代码给修改和扩展带了极大的影响。最简单的例子，如果两个方法中使用了大量的相同代码，只有极少的逻辑不一致，就应该考虑“重复代码”抽离为单独方法。否则，维护的时候，甚至都不知道还有地方用到了相同的代码，而修改的时候又需要针对两个方法不同之处做出特殊处理，久而久之，大量的补丁代码，条件判断，特殊处理就会充斥程序。此时，大量BUG的产生，也是情理之中的事了。

## **避免过长**

### **参数数量**

【推荐】控制参数数量，如果一个接口的参数数量很多，考虑将其包装成一个“参数封装类”。避免不断的修改接口，增加维护成本和出错的概率。如果接口在设计之时就考虑到后续扩展的可能，则不要把参数写死。



### **方法长度**

【推荐】控制方法长度，包括《JAVA开发手册》在内的很多关于JAVA代码编写建议的书籍，都会有一条，单方法的长度不要太长的提倡（一般为80行），过长的方法代表了过粗的粒度。而粒度越大，可复用的可能性就越低，后期的维护成本越高。

试想一个简单的例子，一个800行的方法，如果你要修改一个变量，你需要考虑多少情况，想要保证本次修改不出问题，你至少需要完全了解这个大方法的很多细节和规则逻辑，这是不现实的，最后的结果必然是，只有写的人知道，甚至写的人都维护不了。

例如，现在有部分代码片段如下（方法全长100多行）：



如果现在要去维护上述代码，并且有需求要修改其中的一部分，需求是“修改优先级的处理”。如果想要保证不出错，首先要知道这个方法的完整流程，然后找到与优先级相关的代码，最后去修改，还要保证这个修改不会影响到方法内的其他代码。但其实这段代码想表达的意图很简单，这是重构后的代码，如下图所示。



### **类长度**

【推荐】避免过长的类。一个类的长度过长，往往不是由于这个类真的需要这么多代码量，而是很多不应由这个类管理的东西被错误的管理了。毕竟，面向对象编程，很多时候，一个庞大的类是往往会聚合或者组合很多其他的类（组件），而不是全部在一个类内实现。如果可以的话，尽量在设计的时候，将每个抽象的类所管理的职责单一化，通过组合，聚合，关联的方式灵活的控制其内在联系。