何时重构：

1. 添加新功能
2. 修复错误时重构
3. 代码复审时重构

重构涉及的涉及模式：

1. 策略模式
2. 状态模式
3. 模板方法模式

重构的基本技巧：

小步前进、频繁测试

临时变量众多—>采用Replace Temp with Query （没有解决）—>采用Replace Method with Method Object

6.1 Extact Method———提炼函数

有一段代码可以被组织在一起并独立出来。将这段代码放进一个独立函数中，并让函数名称解释该函数的用途。

6.2 Inline Method———内联函数

一个函数的本体与名称同样清楚易懂。在函数调用点插入函数本体，然后移除该函数。

6.3 Inline Temp———内联临时变量

有一个临时变量，只被一个简单表达式赋值一次，而它妨碍了其他重构手法。将对所有该变量的引用动作，替换为对它赋值的那个表达式自身。

6.4 Replace Temp with Query———以查询取代临时变量

你的程序以一个临时变量保存某一表达式的运算结果。将这个表达式提炼到一个独立函数中。将这个临时变量的所有引用点替换为对新函数的调用。此后，新函数就可以被其他函数使用。

6.5 Introduce Explaining Variable————引入解释性变量

你有一个复杂的表达式，将该复杂表达式（或者其中一部分）的结果放进一个临时变量，以此变量名称来解释表达式用途。

6.6 Split Temporary Variable———分解临时变量

某个临时变量被赋值超过一次，它既不是循环环量，也不用于收集计算结果。针对每次赋值，创造一个独立、对应的临时变量。

6.7 Remove Assignments to Parameters———移除对参数的赋值

代码对一个传入的参数进行赋值。以一个零食变量取代该参数的位置。在Java中，不要对参数进行赋值。

6.8 Replace Method with Method Object———以函数对象取代函数

一个大型的函数，其中对局部变量的使用使你无法采用Extract Method。将这个函数放进一个单独对象中，如此一来局部变量就变成了对象内的字段。然后你可以在同一个对象中将这个大型函数分解为多个小型函数。

6.9 Substitute Algorithm———替换算法

想要把某个算法替换为另一个更清晰的算法。将函数本地替换为另一个算法。

在对象设计过程中，”决定把责任放在那儿”即使不是最重要的事，也是最重要的事之一。

1. 类承担过多的责任而变得臃肿不堪。
2. 一个类变得不太负责，则会成为累赘。

7.1 Move Method———搬移函数

有个函数与其所驻类之外的另一个类进行有更多的交流：调用或者，或者被或者调用。

在该函数最常引用的类中建立一个有着类似行为的新函数。将就函数变成一个单纯的委托函数，或者将旧函数完全移除。

7.2 Move Field———搬移字段

在某个字段被其所驻之外的另一个类更多地用到。

在目标类新建一个字段，修改源字段的所有用户，令他们改用新字段。

7.3 Extract Class———提炼类

某个类做了应该由两个类做的事。

建立一个新类，将相关的字段和函数从旧类搬移到新类。

7.4 Inline Class———将类内联化

某个类没有做太多事情。

将这个类的所有特性搬移到另一个类中，然后移除原类。

Extract Class 与Inline Class 互为矛盾体。

7.5 Hide Delegate———隐藏委托关系

客户通过一个委托类来调用另一个对象。

在服务类上建立客户所需要的所有函数，用以隐藏委托关系。

7.6 Remove Middle Man———移除中间人

某些类做了过多的简单委托工作。

让客户直接调用受委托类。

Hide Delegate 与 Remove Middle Man互为矛盾体。

7.7 Introduce Foreign Method———引入外加函数

你需要为提供服务的类增加一个函数，但你无法修改这个类。

在客户端建立一个函数，并以第一参数形式传入一个服务类实例。

7.8 Introduce Local Extension———引入本地拓展

你要为服务提供一些额外函数，但你无法修改这个类。

建立一个新类，使它包含这些额外函数。让这个拓展品成为源类的子类或者包装类。

注意：使用子类和包装类的副作用。

8.1 Self Encapsulate Field———自封装字段

直接访问一个字段，但与字段之间的耦合关系逐渐变得笨拙。

为这个字段建立取值/设值字段，并且只以这些函数来访问字段。

8.2 Replace Data Value With Object———以对象取代数值

有一个数据项，需要与其他数据和行为一起使用才有意义。

将数据项变成对象。

8.3 Change Value to Reference———将值对象改为引用对象 -- 有点难理解

从一个类衍生出许多彼此相等的实例，希望将它们替换为同一个对象。

将这个值对象变成引用对象。

8.4 Chang Reference to Value———将引用对象改为值对象 -- 有点难理解

有一个引用对象，很小且不变，而且不易管理。

将它变成一个值对象。

8.5 Replace Array with Object———以对象取代数据

有一个数组，其中的元素各自代表不同的东西。

以对象替换数组。对于数组中的每个元素，以一个字段来表示。

8.6 Duplicate Observed Data———复制被监测的数据 -- 有点难理解

8.7 Change Unidirectional Association to Bidirectionala———将单向关联改为双向关联 -- 有点难理解

两个类都需要使用双方特性，但其间只有一条单向连接。

添加一个反向指针，并使修改函数能够同时更新两条连接。

8.8 Change Bidirectionala Association to Unidirectional———将双向关联改为单向关联 -- 有点难理解

两个类之间有双向关联，但其中一个类如今不再需要另一个类的特性。

去掉不必要的关联。

8.9 Replace Magic Number with Symbolic Constant———以字面常量取代魔法值

有一个字面数值，带有特别含义。

创建一个常量，根据其意义为它命名，并将上述的字面数值替换为这个常量值。

8.10 Encapsulate Field———封装字段

类中存在一个public字段。

将它声明为private，并提供相应的访问函数。

8.11 Encapsulate Collection———封装集合

有一个public函数返回一个集合。

让这个函数返回该集合的一个只读副本，并在这个类中提供添加/移除集合元素的函数。

8.12 Replace Record with Data Class———以数据类取代记录

需要面对传统编程环境中的记录结构。

为该记录创建一个“哑”数据对象。

8.13 Replace Type Code with Class———以类型取代类型码

类中有一个数值类型码，但它并不影响类的行为。

以一个新的类替换该数值类型码。

8.14 Replace Type Code with Subclasses———以子类取代类型码

有一个不可变的类型码，它会影响类的行为。

以子类取代这个类型码。

8.15 Replace Type Code with State/Strategy———以状态/策略设计模式取代类型码

有一个类型码，它会影响类的行为，但你无法通过继承手法消除它。

以状态对象取代类型码。

8.16 Replace Subclass with Fields———以字段代类子类

各个子类的唯一差别只是在“返回常量数据”的函数上。

修改这些函数，使它们返回超类中的某个（新增）字段，然后销毁子类。

9.1 Decompose Conditional———分解条件表达式

有一个复杂的条件语句。

从if、then、else三个段落中分别提炼出独立函数。

9.2 Consolidate Conditional Expression———合并条件表达式

有一系列条件测试，都得到相同结果。

将这些测试合并为一个条件表达式，并将这个表达式提炼为一个独立函数。

9.3 Consolidate Duplicate Conditiaonal Fragments———合并重复的条件判断

在条件表达式的每一个分支上有着相同的一段代码。

将这段重复代码搬移到条件表达式之外。

9.4 Remove Control Flag———移除控制标记

在一系列布尔表达式中，某个变量带有“控制标记”的作用。

以break语句或者return语句来取代控制标记。

9.5 Replace Nested Conditional with Guard Clauses———以卫语句取代嵌套条件表达式

函数中的条件逻辑使人难以看清正常的执行路径。

使用卫语句表现所有特殊情况。

如果某个条件极其罕见，就应该单独检查该条件，并在该条件为真时立刻从函数中返回。这样单独检查常常被称为“卫语句”。

9.6 Replace Conditional with Polymorphism———以多态取代条件表达式

有个条件表达式，它根据对象类型的不同而选择不同的行为。

将这个条件表达式的每个分支放进一个子类内的覆写函数中，然后将原始的函数声明为抽象函数。

9.7 Introduce Null Object———引入空对象

需要再三检查某对象是否为null。

将null替换为null对象。

9.8 Introduce Assertion———引入断言

某一段代码需要对程序状态做出假设。

以断言明确这种假设。

10.1 Rename Method———函数改名

函数的名称未能揭示函数的用途。

修改函数名称。

10.2 Add Parameter———添加参数

某个函数需要从端得到更多信息。

为此函数添加一个对象参数，让该对象带进函数所需信息。

10.3 Remove Parameter———移除参数

函数体不再需要某个参数。

将该参数移除。

10.4 Separate Query from Modifier———将查询函数和修改函数分离

某个函数返回对象状态值，修改对象状态。

建立两个不同的函数，其中一个负责查询，另一个负责修改。

10.5 Parameterize Method———令函数携带参数

若干函数做了类似的工作，但在函数本体中却包含了不同的值。

建立单一函数，以参数表达那些不同的值。

10.6 Replace Parameter with Explicit Methods———以明确函数取代参数

一个函数，其中完全取决于参数值而采取不同行为。

针对该参数的每一个可能值，建立一个独立函数。

10.7 Preserve Whole Object———保持对象完整

从某个对象中取出若干值，将它们作为某一次函数调用时的参数。

改为传递整个对象。

10.8 Replace Parameter with Methods———以函数取代参数

对象调用某个函数，并将所得结果作为参数，传递给另一个函数，而接受该参数的函数本身也能够调用前一个函数。

让参数接受者去除该项参数，并直接调用前一个函数。

10.9 Introduce Parameter Object———引入参数对象

某些参数总是很自然地同时出现。

以一个对象取代这些参数。

10.10 Remove Setting Method———移除设值函数

类中的某个字段应该在对象创建时被赋值，然后就不再改变。

去掉该字段的所有设值函数。

10.11 Hide Method———隐藏函数

一个函数，从来没有被其他任何类用到。

将这个函数改为private。

10.12 Replace Constructor with Factory Method———以工厂函数取代构造函数

在创建对象时不仅仅是做简单的构建工作。

将构造函数替换为工厂函数。

10.13 Encapsulate Downcast———封装向下转型

某个函数返回的对象，需要由函数执行者向下转型（downcast）

将向下转型动作移到函数中。

10.14 Replace Error Code with Exception———以异常取代错误码

某个函数返回一个特定的代码，用以表示某种错误情况。

改用异常。

10.15 Replace Exception with Test———以测试取代异常

面对一个调用者可以预先检查的条件，你抛出了一个异常。

修改调用者，使它在调用函数之前先做检查。

11.1 Pull Up Field———上移字段

两个子类拥有相同的字段。

将该字段移至超类。

11.2 Pull Up Method———函数上移

有些函数，在各个子类中产生完全相同的结果。

将该函数移至超类。

11.3 Pull Up ConstructorBody———将构造函数本体上移

在各个子类中拥有一些构造函数，它们的本体几乎完全一致。

在超类中新建一个构造函数，并在子类构造函数中调用它。

11.4 Push Down Method———函数下移

超类中的某个函数只与部分（而非全部）子类有关。

将这个函数移到相关的那些子类去。

11.5 Push Down Field———字段下移

超类中的某个字段只被部分（而非全部）子类用到。

将这个字段移到需要它的那些子类中去。

11.6 Extract Subclass———提炼子类

类中的某些特性只被某些（而非全部）实例用到。

新建一个子类，将上面所说的那一部分特性移到子类中。

11.7 Extract Superclass———提炼超类

两个类有相似的特性。

为这两个类建立一个超类，将相同特性移至超类。

11.8 Extract Interface———提炼接口

若干客户使用接口中的同一子集，或者两个类的接口有部分相同。

将相同的子集提炼到一个独立接口中。

11.9 Collapse Hierarchy———折叠继承体系

超类和子类之间无太大区别。

将它们合为一体。

11.10 Form Template Method———构造模板函数

一些子类，其中相应的某些函数以相同顺序执行类型的操作，但各个操作的细节上有所不同。

将这些操作分别放进独立函数中，并保持它们都有相同的签名，于是原函数也就变得相同。然后原函数上移至超类。

11.11 Replace Inheritance with Delegation———以委托取代继承

某个子类中只使用超类接口中的一部分，后者是根本不需要继承而来的数据。

在子类中新建一个字段用以保存超类；然后调整子类函数，令它改而委托超类；然后去掉两者之间的继承关系。

11.12 Replace Delegation with Inheritance———以继承取代委托

在两个子类之间使用委托关系，并经常为整个接口编写许多极简单的委托函数。

让委托函数继承受托类。

12.1 Tease Apart Inheritance———梳理并分解继承体系

某个继承体系同时承担两项责任。

建立两个继承体系，并通过委托关系让其中一个可以调用另一个。

12.2 Convert Procedural Design to Objects———将过程化设计转化为对象设计

一些传统化风格的代码。

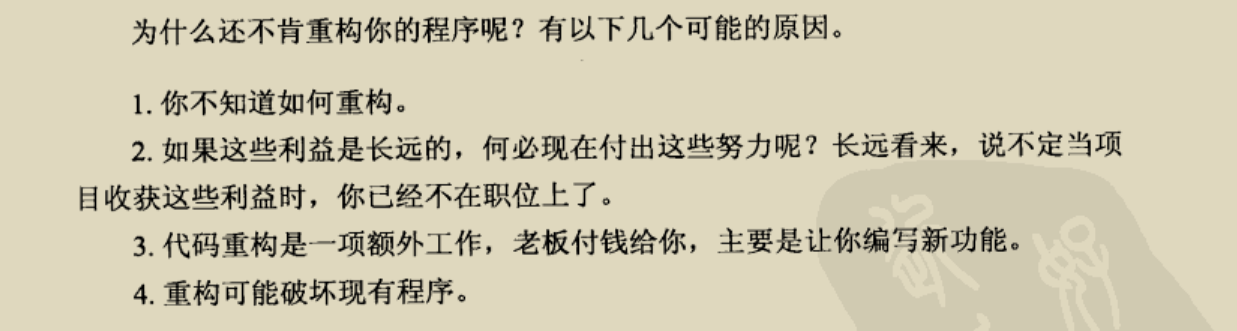
将数据记录变为对象，将大块的行为分成小块，并将行为移入相关对象之中。

12.3 Separate Domain from Presentation———将领域和表述/显示分离

12.4 Extract Hierarchy———提炼继承体系

某个类做了太多工作，其中一部分是以大量条件表达式完成的。

建立继承体系，以一个子类表示一种特殊情况。



高效JAVA编程：

# 一、创建和销毁对象

## 1.考虑使用静态工厂方法代替构造器

静态方法的惯用名称：

（1）valueOf-----返回的实例域它的参数具有相同的值。这样的静态工厂方式实际上是类型转换方法。

（2）of----- valueOf的一种更为简洁的替代。

（3）getInstance-----返回的实例是通过方法的参数来描述的，但是不能够说与参数具有相同的值。

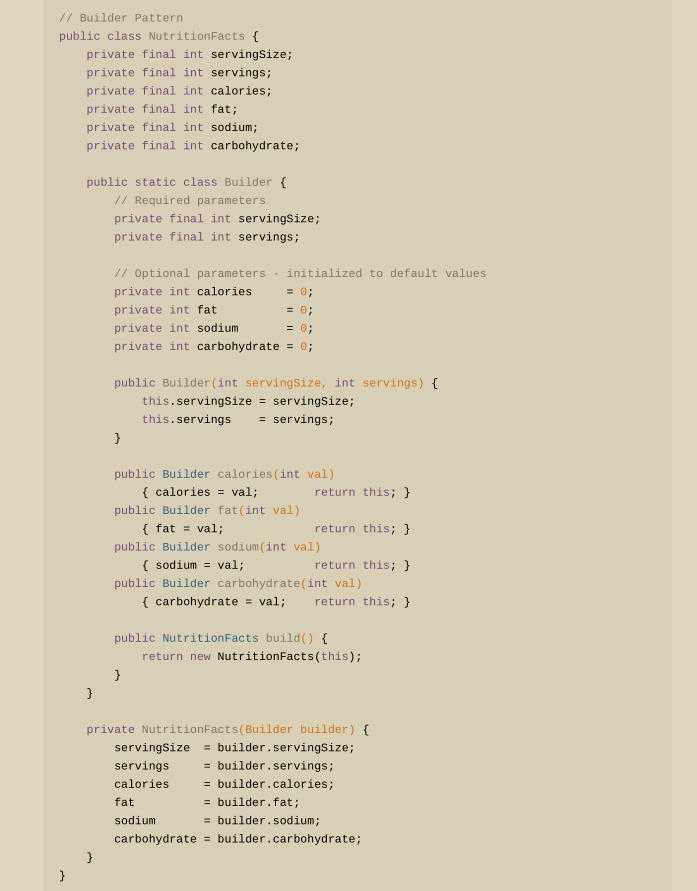
（4）newInstance-----和getInstance一样，但是newInstance能够确保返回的每个实例都与所有其他实例不同。

（5）getType-----像getInstance一样，但是在工厂方法处于不同的类中的时候使用。Type表示工厂方法所返回的对象的类型。

（6）newType-----像newInstance一样。

## 2、遇到多个构造器参数时要考虑用构造器-Builder模式。

如果类的构造器或者静态工厂中具有多个参数，设计这种类时，Builder模式就是不错的选择，特别是当大多数参数都是可选的时候。



# 二、对所有对象都通用的方法

1.覆盖equals时请遵守通用约定

（1）类的每个实例本质上都是唯一的。对于代表活动实体而不是值的类确实如此。

（2）超类已经覆盖了equals方法，从超类继承过来的行为对于子类也是合适的。

（3）类时私有的或是包级私有的，可以确定它的equals方法永远也不会被调用。在这种情况下，无疑是应该覆盖equals方法，以防它被意外调用。

什么时候该覆盖equals方法：

（1）如果类具有自己特有的“逻辑相等”概念（不同于对象等同的概念），而且超类还没有覆盖equals以实现期望的行为，这时需要覆盖equals方法。

覆盖equals方法时需遵循以下约定：

（1）自反性 （2）对称性 （3）传递性 （4）一致性

2.覆盖equals时总要覆盖hashCode

3.始终要覆盖toString

# 三、类和接口

1、复合优于继承

2、接口优于继承

3、接口只用于定义类型

4、考虑优先使用静态成员类

静态成员类和非静态成员类的区别：

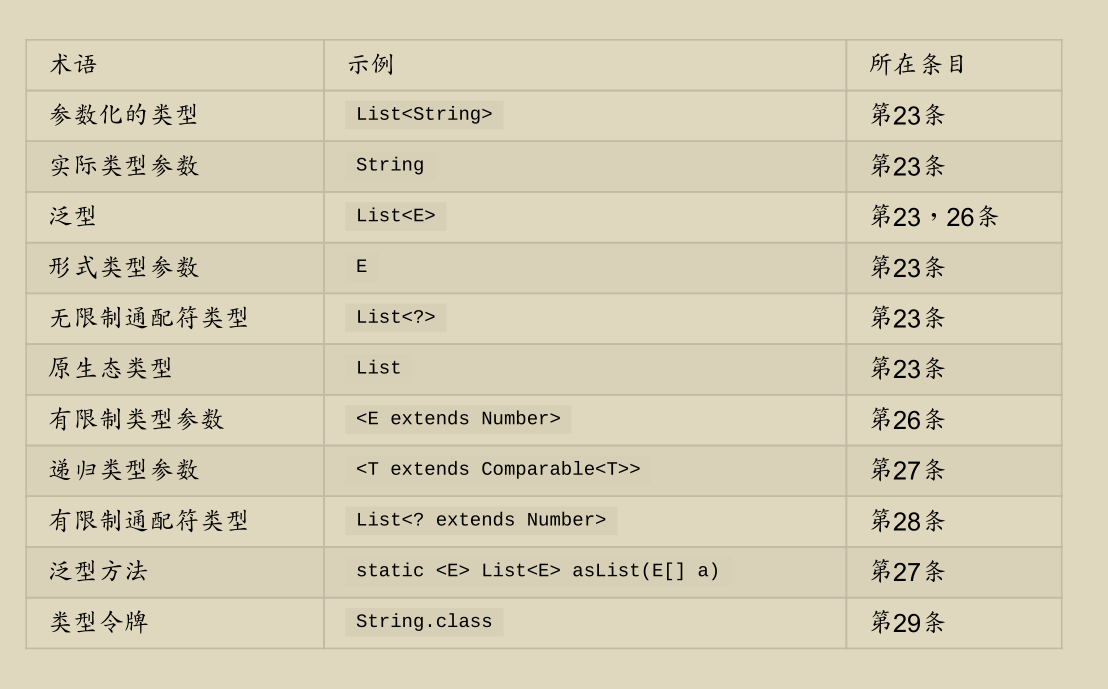
（1）非静态成员类的每个实例都隐含着与外围类的一个外围实例相关联。在非静态成员类的实例方法内部，可以调用外围实例上的方法，或者利用修饰过的this构造器获得外围实例的引用。在没有外围实例的情况下，要想创建非静态成员类是不可能的。

四、泛型

1、不要在代码中使用原生态类型

泛型有子类型化的规则，List<Stirng>是原生态类型List的一个子类型，而不是参数化类型List<Object>的子类型。

擦除就是使泛型可以与没有使用泛型的代码随意进行互用。



2、列表优于数组

数组和泛型有着非常不同的类型规则。数组是协变得并且可以具体化的；泛型是不可变的且可以被擦除的。数组提供了运行时的类型安全，但是没有编译时的类型安全。

3、消除非受检警告

4、优先考虑泛型

5、优先考虑泛型方法

泛型单例工厂、泛型单例

如果编写的是将被广泛使用的类库，则一定适当地利用通配符类型。请记住基本的原则（PECS）：Producer-exteds;consumer-super。所有的comparable和comparator都是消费者。如果参数化类型表示一个T生产者，就是用<? extend T>;如果它表示一个T的消费者，就是用<? Super T>.