一. 对象初始化顺序

父类静态成员和静态初始化块 ，按在代码中出现的顺序依次执行

子类静态成员和静态初始化块 ，按在代码中出现的顺序依次执行

父类实例成员和实例初始化块 ，按在代码中出现的顺序依次执行

父类构造方法

子类实例成员和实例初始化块 ，按在代码中出现的顺序依次执行

子类构造方法

### 常见错误实践

### 2.1　对象属性间的平行依赖.

Public class Context{

Private A a;

Private B b;

}

属性a和b我们认为他们是“平行的”。现在他们没有存在互相依赖

Public class Context{

Private A a = new A();

Private B b = new B(a);

}

现在呢，可以认为b平行依赖于a.这就是对象属性间的平行依赖。这种情形是没有问题的。

Public class Context{

**Private A a;**

**Private B b = new B(a);//此时B如果实例化，那么a为null**

**Public Context(A a){**

**This.a = a;**

**}**

}

错误出现了，属性b的实例化要优先于构造方法的，所以B接受了一个null。至于NullPointerException什么时候触发，没人能预测，最好是在B构造方法中触发，这时你可能还会恍然大悟，传入了一个null值，但是如果是在B的常规方法调用触发，你可能需要花点时间来排查空指针异常了。

**记住：实例属性的实例代码块要优于构造方法执行。**

**建议：不要依赖Java的对象初始化顺序，尽量将属性初始化放到构造方法中。尤其类比较复杂，存在“对象属性的平行依赖”时。但是我们还是喜欢声明属性的同时将其初始化（顺便设置个final常量）。毕竟这样编码很爽，可读性也比较高。这一点C++规范的很好，属性声明时不得初始化，只能在构造函数中初始化。所以我们经常看见C++超长的只做了简单赋值操作的构造函数。**

**2.2 构造函数陷阱(**构造函数与重载方法带来的困惑**)**

**”构造函数陷阱“：构造方法中调用可被重写的方法**

Public class A{

Public A(){

....//初始化操作

**Function();//**

....初始化操作

}

Public void function(){

.....

}

}

如果A的子类重写了function方法，那么A类构造方法执行的就是其子类的function实现，如果A类设计时没有考虑到这种情况，那么A的初始化就存在很大风险

所以要将function 设为private，或final

**建议：构造方法内不要使用public方法，如果使用必须要考虑　子类可能会重写该方法，影响父类的初始化过程。建议将存在子类重写需求的逻辑抽象出一个方法设置为protected，　public方法设置为final调用该protected方法。这样public对外服务，和protected提供重写　比较清晰。**

**另外记住　private ,final方法中　调用public方法也要想到子类可能会重写这个public方法偶。**

**下面还会讲解构造方法中　执行public　方法还有多坑。**

**2.３ 构造函数与重载带来的空指针异常**

Public class Context{

Public A(){//构造方法

....

Register();

....

}

Public void register(){

beforeRegister();

.....

afterRegister();

}

Protected void beforeRegister(){

}

Protected void afterRegister(){

}

}

现在Context需要向外提供注册功能.但是实例化时，需要先注册一些服务。注册操作前后会　执行before方法和after方法，子类可以重写，向提供扩展注册功能

于是乎，子类重写了before,after功能，蛋疼的空指针异常又出现了。

为什么呢？因为

Public class ChildContext{

Private Map<String,Object> map = new HashMap<>();

Protected void beforeRegister(){

......

Map.put();

.....

}

Protected void afterRegister(){

....

//

...

}

}

子类重写了注册的before和after方法

目的是　监听注册功能，维护了额外的数据结构（）。

**但是正是这个额外的数据结构触发了空指针异常。**

**原因正是父类的构造方法中执行了子类重写的方法，**

**而子类的重写方法中使用了自己的属性**

**就像beforeRegister()方法中使用了map一样。**

**但是map还没有初始化呢,还没有初始化呢，还没初始化！！！！啊啊啊？**

**解决思路有好几种：**

1. 最SB的思路是将子类中的变量map声明为static，不就先于父类构造方法执行了吗？

的确解决了这个bug，但引入新的bug，我们提到过慎用static变量

因为他属于整个类。不单单属于一个对象。那意味着所有的对象实例都共用这个static　map,　这是正确的逻辑吗？你考虑了吗？这个坑我还就犯过...

最最重要的是，这个static map　在该进行垃圾回收时没法被回收。没准哪个时刻出现了　Out of　Memory　内存溢出，服务器宕机，然后领导查到了这个root　根节点大对象正是static map　!!!　[非洲凝视]

1. 有一个争议的实现方法

父类构造方法中　先调用beforeInitialize

同时beforeInitialize()，afterInitialize()方法供子类重写，

这时子类就可以把属性初始化需求放到beforeInitialize()方法中。

实现了父类对子类的依赖，实现子类属性　先于　父类初始化

但是我们倒转了依赖。破坏了子类，父类初始化顺序。

1. 我们可以在beforeRegister中　判断属性变量是否为null，

　　如果为null就初始化它。简单粗暴，

但是我没有用这种方法。因为我想试试第二种方法到底会有多坑....

1. **还有好多方法...但是归根结底，我们是在构造方法中调用可重写方法倒置了子父类的依赖，让父类依赖于子类，与　Java面向对象的设计理念相冲突，才会出现这么多问题。**

**建议：尽量不要在构造方法中调用可以可被重写的方法**

**Public,protected方法尽可能少的出现在构造方法中**

**”构造函数陷阱“：构造方法中调用可被的重写方法**

**2.４ 父类对子类的依赖**

面向对象的设计中，子类可以访问到父类方法，属性，但是父类无法访问子类属性，本应该是子类依赖于父类。

但是　设计模式中模板方法　的思路带来了父类对子类的依赖。（模板方法：父类的方法中调用了方法A,但是方法A由子类具体实现）

模板设计虽然由父类定义了逻辑处理的流程，子类只是填空。但是父类的处理逻辑中包括了不可预测的子类实现。模板方法要求父类充分考虑子类可能的具体实现，考虑哪种实现是正确的，符合要求的。模板的设计需要高度的抽象。

之前已经谈到了，构造方法中尽量不要使用模板方法的设计。

另外：充分考虑子类通过重写对父类处理逻辑的干扰，经常使用private 和final

1. **５　静态属性.静态方法带来的麻烦**

静态属性属于对象共有，静态方法通过类名即可调用，有何麻烦？

来个例子：

封装数据库配置获取数据库连接。我们使用了dbcp连接池，需要提供url.password,username等等...于是将username,password,url等写死在DataBaseUtil类的属性中。同时提供getConnection静态方法。

DataBaseUtil.getConnection()，可以在任何地方随时调用。用起来很爽。

**需求变更一：**

将配置信息转移到xml中。

现在DataBaseUtil.getConnection()变更。需要传递xml文件路径。但是DatabaseUtil调用太多，需改需要改动地方太多。

临时方案:

在程序启动时，设置xml路径到DataBaseUtil静态属性中。这样保证所有的getConnection调用之前都设置了xml

**需求变更二：**

数据库配置再次变动，需要配置多数据源，例如主从配置等。

我们需要通过数据源key获取数据库连接.

所以我们不得不提供

DataBaseUtil.getConnection(“main”);

当然，我们还可以提供sb方案，DataBaseUtil.getConnection()方法获取默认数据源连接。

分析一下.DataBaseUtil.getConnection的主要逻辑：

1. 负责xml文件读取.解析.配置数据源数据库连接池
2. 获取默认数据源连接

DataBaseUtil的逻辑由简单的属性配置。

到xml配置管理

到后来的多数据源配置管理....

DataBaseUtil已经远远地脱离了当初设计它的初衷...

面对需求的变更，DataBaseUtil的应对很保守很被动。

而我认为造成无能为力的原因，就是DataBaseUtil提供了getConnection静态方法.

假设我们从一开始将DataBaseUtil.getConnection改为实例方法，同时DataBaseUtil构造方法参数列表要求传入xml路径，完成xml读取解析，以及多数据源管理。

例如获取主数据源，我们只需要在构造方法中提供参数，即设置当前DataBaseUtil对象为主数据源。通过DataBaseUtil实例对象的任何getConnection调用前都已经完成xml解析，数据源管理。

getConnection不需要再关心xml,多数据源这些过程。

DataBaseUtil实例的创建可以由工厂负责，在任何需要获取数据库链接的地方，我们都可以传递DataBaseUtil实例引用，使用者不需要了解实例的创建过程。

就像《Clean Code》（《代码整洁之道》）作者Martin提到过的：

**“软件系统应将启动过程和启动过程之后的运行逻辑分离开，在启动过程中构建应用对象，配置内部模块互相依赖的关系。”**

显然静态的getConnection将启动过程和启动后的运行逻辑混在了一起，系统维护难度增加，当把DataBaseUtil.getConnection设置为实例调用,我们面对需求变化更加自由，将对象实例的构建过程和运行时逻辑分离开。在对象构造时完成复杂的配置逻辑。那么启动后的方法调用更加的轻松，关注点更少。

**关于静态方法的反思：**

之前提到，静态方法通过类名.方法名即可访问。这只是表面上的简洁，更深层次上，静态方法意味：它没有“状态”可言，客户端不需要关注他的状态，你随时可以访问，它突破了对象实例构造方法这唯一入口，它可以在任何该类对象实例初始化前对外提供服务。

如果一个类包含了n个静态方法，他就相当于提供了n+1个入口，类的设计者除了需要考虑构造方法这一入口之外，还需要考虑是否需要在静态方法中完成　额外的系统准备工作。就像DataBaseUtil.getConnection还需要完成xml,数据源管理，然后才能返回数据库连接。

此时静态方法给类引入了更大的复杂性。

**关于静态方法使用的建议**

1. .不要因为静态方法通过类名调用的方便，就优先考虑静态方法。
2. 静态方法应侧重完成工具功能，不应处理　复杂逻辑，复杂的处理流程。当静态方法中不建议访问系统业务资源，例如xml配置读取.
3. 静态方法应该侧重于处理，参数列表中的参数.例如function(A a, B b);静态方法应该针对参数a,b作处理。
4. 以上几个例子证明，静态方法相比实例方法扩展性弱。如果静态方法实现依赖的数据结构变化频繁，不建议使用静态方法。
5. ...使用静态方法时，多问自己一句，它与系统其他部分有关吗？他是否与其他模块存在明显边界？

**2.6 太多的无状态类**

**无状态类：我们可以把对象的属性视为内部状态，当一个对象没有属性，或者属性不可修改只读，我们可以把它视为无状态类。例如常见的service，dao,事件处理action等等**

我们习惯了无状态类，并托管给ioc管理。很多时候我们设计时更倾向于设计无状态类，但是不经意间自己抗下太多的包袱

　错误示例：

需求：将一个List<Map>对象持久化存储时时，我们抽象了接口

Interface XXXStorage{

Public void store(List<Map> ,　String path);

Public void resolve(String path);

}

于是乎，我们设计了XmlXXXStorage,SerializeXXXStorage具体实现。提供了xml,和对象序列化方法存储。对于这两个类而言

由于两个参数，满足了xml和序列化存储的所有客户端输入要求.所以不需要额外的状态，我们将其设计成单例类。并沾沾自喜，我们减少实例化对象次数，提高了性能

需求变更：

现在需要额外的存储方式，存储到数据库里

请问String path作何解？对于数据库存储有何意义。[非洲凝视]

一直都说，实现依赖于抽象，不得让抽象依赖于具体实现。

XXXStorage接口中store参数列表过于具体.只需要提供List<Map>,只需要告诉接口，我要存储什么东西即可，至于具体存储格式以及位置由具体实现类决定即可。

具体实现类需要的参数我们可以通过构造方法参数，或者setter方式传入。避免让接口中的方法接受过多参数，避免具体实现的逻辑影响接口的设计。我们还是通过将启动过程和启动后的运行逻辑分离开。这样具体实现类的设计有了更多的自由。但也因此不能将实现类设计成无状态的。

**反思：**

**无状态类的设计总是简单的，线程安全的。但是普通的方法调用可能需要过多的参数。同时也干扰了我们接口的设计。**

**善于利用构造方法，可以简化我们的方法设计。**