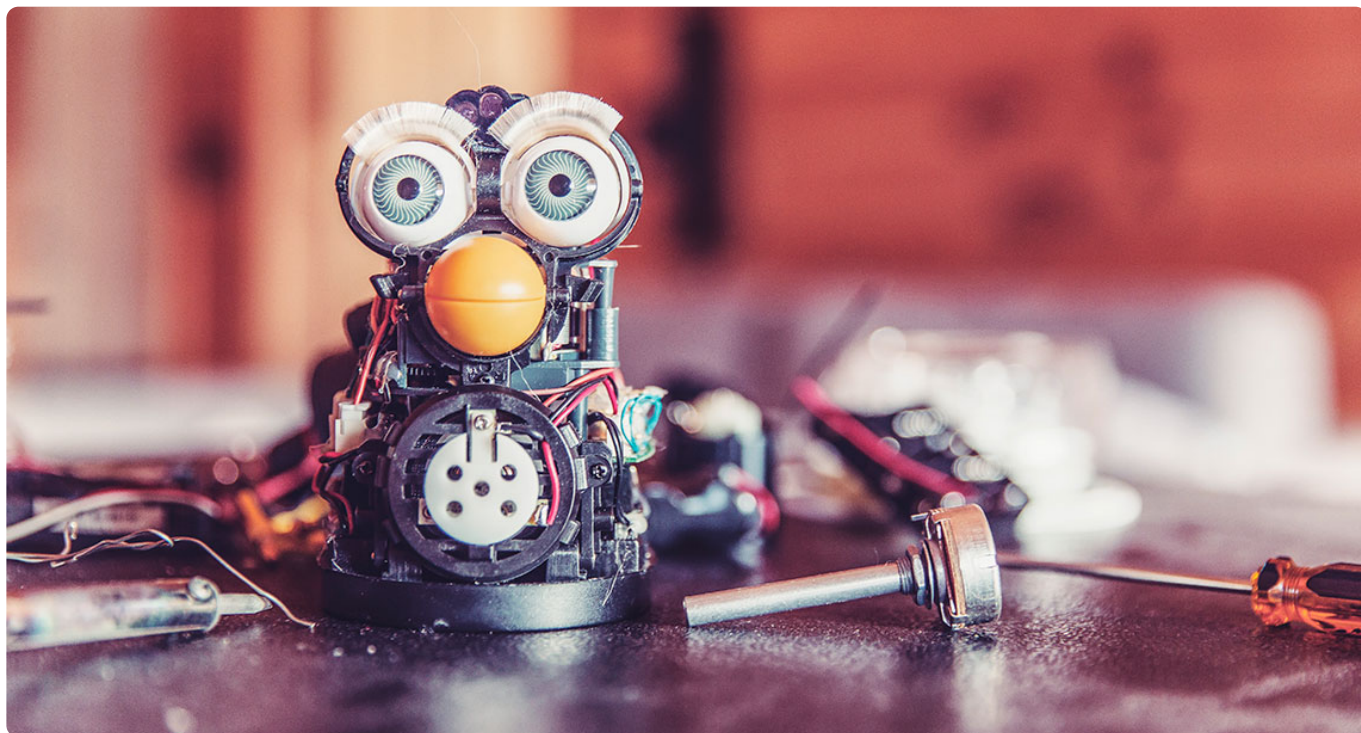


05 | Spring DI容器：如何分析一个软件的模型？

2020-06-03 郑晔

软件设计之美

[进入课程 >](#)



讲述：郑晔

时长 14:04 大小 12.89M



你好！我是郑晔。

在上一讲中，我们讨论了如何了解一个软件的设计，主要是从三个部分入手：模型、接口和实现。那么，在接下来的三讲中，我将结合几个典型的开源项目，告诉你如何具体地理解一个软件的模型、接口和实现。

今天这一讲，我们就先来谈谈了解设计的第一步：模型。如果拿到一个项目，我们怎么去理解它的模型呢？



我们肯定要先知道项目提供了哪些模型，模型又提供了怎样的能力。这是所有人都知道的事情，我并不准备深入地去探讨。但如果只知道这些，你只是在了解别人设计的结果，这种程度并不足以支撑你后期对模型的维护。

在一个项目中，常常会出现新人随意向模型中添加内容，修改实现，让模型变得难以维护的情况。造成这一现象的原因就在于他们对于模型的理解不到位。

我们都知道，任何模型都是为了解决问题而生的，所以，理解一个模型，需要了解在没有这个模型之前，问题是如何被解决的，这样，你才能知道新的模型究竟提供了怎样的提升。也就是说，**理解一个模型的关键在于，要了解这个模型设计的来龙去脉，知道它是如何解决相应的问题。**

今天我们以 Spring 的 DI 容器为例，来看看怎样理解软件的模型。

耦合的依赖

Spring 在 Java 世界里绝对是大名鼎鼎，如果你今天在做 Java 开发而不用 Spring，那么你大概率会被认为是个另类。

今天很多程序员都把 Spring 当成一个成熟的框架，很少去仔细分析 Spring 的设计。但作为一个从 0.8 版本就开始接触 Spring 的程序员，我刚好有幸经历了 Spring 从渺小到壮大的过程，得以体会到 Spring 给行业带来的巨大思维转变。

如果说 Spring 这棵参天大树有一个稳健的根基，那其根基就应该是 Spring 的 DI 容器。DI 是 Dependency Injection 的缩写，也就是“依赖注入”。Spring 的各个项目都是这个根基上长出的枝芽。

那么，DI 容器要解决的问题是什么呢？它解决的是**组件创建和组装**的问题，但是为什么这是一个需要解决的问题呢？这就需要我们了解一下组件的创建和组装。

在前面的课程中，我讲过，软件设计需要有一个分解的过程，所以，它必然还要面对一个组装的过程，也就是把分解出来的各个组件组装到一起，完成所需要的功能。

为了叙述方便，我采用 Java 语言来进行后续的描述。


我们从程序员最熟悉的一个查询场景开始。假设我们有一个文章服务（ArticleService）提供根据标题查询文章的功能。当然，数据是需要持久化的，所以，这里还有一个 ArticleRepository，用来与持久化数据打交道。

熟悉 DDD 的同学可能发现了，这个仓库（Repository）的概念来自于 DDD。如果你不熟悉也没关系，它就是与持久化数据打交道的一层，和一些人习惯的 Mapper 或者 DAO（Data Access Object）类似，你可以简单地把它理解成访问数据库的代码。

 复制代码

```
1 class ArticleService {
2     //提供根据标题查询文章的服务
3     Article findByTitle(final String title) {
4         ...
5     }
6 }
7
8 interface ArticleRepository {
9     //在持久化存储中，根据标题查询文章
10    Article findByTitle(final String title);
11 }
```

在 ArticleService 处理业务的过程中，需要用到 ArticleRepository 辅助它完成功能，也就是说，ArticleService 要依赖于 ArticleRepository。这时你该怎么做呢？一个直接的做法就是在 ArticleService 中增加一个字段表示 ArticleRepository。

 复制代码

```
1 class ArticleService {
2     private ArticleRepository repository;
3
4     public Article findByTitle(final String title) {
5         // 做参数校验
6         return this.repository.findByTitle(title);
7     }
8 }
```

目前看起来一切都还好，但是接下来，问题就来了，这个字段怎么初始化呢？程序员一般最直接的反应就是直接创建这个对象。这里选用了一个数据库版本的实现（DBArticleRepository）。

 复制代码

```
1 class ArticleService {
2     private ArticleRepository repository = new DBArticleRepository();
3
4     public Article findByTitle(final String title) {
5         // 做参数校验
```

```
6     return this.repository.findByTitle(title);
7 }
8 }
```

看上去很好，但实际上 `DBArticleRepository` 并不能这样初始化。正如这个实现类的名字所表示的那样，我们这里要用到数据库。但在真实的项目中，由于资源所限，我们一般不会在应用中任意打开数据库连接，而是会选择共享数据库连接。所以，`DBArticleRepository` 需要一个数据库连接（`Connection`）的参数。在这里，你决定在构造函数里把这个参数传进来。

 复制代码

```
1 class ArticleService {
2     private ArticleRepository repository;
3
4     public ArticleService(final Connection connection) {
5         this.repository = new DBArticleRepository(connection);
6     }
7
8     public Article findByTitle(final String title) {
9         // 做参数校验
10        return this.repository.findByTitle(title);
11    }
12 }
```

好，代码写完了，它看上去一切正常。如果你的开发习惯仅仅到此为止，可能你会觉得这还不错。但我们并不打算做一个只写代码的程序员，所以，我们要进入下一个阶段：测试。

一旦开始准备测试，你就会发现，要让 `ArticleService` 跑起来，那就得让 `ArticleRepository` 也跑起来；要让 `ArticleRepository` 跑起来，那就得准备数据库连接。

是不是觉得太麻烦，想放弃测试。但有职业素养的你，决定坚持一下，去准备数据库连接信息。

然后，真正开始写测试时，你才发现，要测试，你还要在数据库里准备各种数据。比如，要测查询，你就得插入一些数据，看查出来的结果和插入的数据是否一致；要测更新，你就得先插入数据，测试跑完，再看数据更新是否正确。

不过，你还是没有放弃，咬着牙准备了一堆数据之后，你突然困惑了：我在干什么？我不是要测试服务吗？做数据准备不是测试仓库的时候应该做的事吗？

那么，问题出在哪儿呢？其实就在你创建对象的那一刻，问题就出现了。

分离的依赖

为什么说从创建对象开始就出问题了呢？

因为当我们创建一个对象时，就必须要有个具体的实现类，对应到我们这里，就是那个 `DBArticleRepository`。虽然我们的 `ArticleService` 写得很干净，其他部分根本不依赖于 `DBArticleRepository`，只在构造函数里依赖了，但依赖就是依赖。

与此同时，由于要构造 `DBArticleRepository` 的缘故，我们这里还引入了 `Connection` 这个类，这个类只与 `DBArticleRepository` 的构造有关系，与我们这个 `ArticleService` 的业务逻辑一点关系都没有。

所以，你看到了，只是因为引入了一个具体的实现，我们就需要把它周边配套的东西全部引入进来，而这一切与这个类本身的业务逻辑没有任何关系。

这就好像，你原本打算买一套家具，现在却让你必须了解树是怎么种的、怎么伐的、怎么加工的，以及家具是怎么设计、怎么组装的，而你想要的只是一套能够使用的家具而已。

这还只是最简单的场景，在真实的项目中，构建一个对象可能还会牵扯到更多的内容：

根据不同的参数，创建不同的实现类对象，你可能需要用到工厂模式。

为了了解方法的执行时间，需要给被依赖的对象加上监控。

依赖的对象来自于某个框架，你自己都不知道具体的实现类是什么。

.....

所以，即便是最简单的对象创建和组装，也不像看起来那么简单。

既然直接构造存在这么多的问题，那么最简单的办法就是把创建的过程拿出去，只留下与字段关联的过程：

```
1 class ArticleService {
2     private ArticleRepository repository;
3
4     public ArticleService(final ArticleRepository repository) {
5         this.repository = repository;
6     }
7
8     public Article findByTitle(final String title) {
9         // 做参数校验
10        return this.repository.findByTitle(title);
11    }
12 }
```

这时候，ArticleService 就只依赖 ArticleRepository。而测试 ArticleService 也很简单，只要用一个对象将 ArticleRepository 的行为模拟出来就可以了。通常这种模拟对象行为的工作用一个现成的程序库就可以完成，这就是那些 Mock 框架能够帮助你完成的工作。

或许你想问，在之前的代码里，如果我用 Mock 框架模拟 Connection 类是不是也可以呢？理论上，的确可以。但是想要让 ArticleService 的测试通过，就必须打开 DBArticleRepository 的实现，只有配合着其中的实现，才可能让 ArticleService 跑起来。显然，你跑远了。

现在，对象的创建已经分离了出去，但还是要有一个地方完成这个工作，最简单的解决方案自然是，把所有的对象创建和组装在一个地方完成：

```
1 ...
2 ArticleRepository repository = new DBArticleRepository(connection);
3 ArticleService service = new ArticleService(repository);
4 ...
```

相比于业务逻辑，组装过程并没有什么复杂的部分。一般而言，纯粹是一个又一个对象的创建以及传参的过程，这部分的代码看上去会非常的无聊。

虽然很无聊，但这一部分代码很重要，最好的解决方案就是有一个框架把它解决掉。在 Java 世界里，这种组装一堆对象的东西一般被称为“容器”，我们也用这个名字。


```
1 Container container = new Container();
2 container.bind(Connection.class).to(connection);
3 container.bind(ArticleRepository.class).to(DBArticleRepository.class);
4 container.bind(ArticleService.class).to(ArticleService.class);
5
6 ArticleService service = container.getInstance(ArticleService.class);
```

至此，一个容器就此诞生。因为它解决的是依赖的问题，把被依赖的对象像药水一样，注入到了目标对象中，所以，它得名“依赖注入”（Dependency Injection，简称 DI）。这个容器也就被称为 DI 容器了。

至此，我简单地给你介绍了 DI 容器的来龙去脉。虽然上面这段和 Spring DI 容器长得并不一样，但其原理是一致的，只是接口的差异而已。

事实上，这种创建和组装对象的方式在当年引发了很大的讨论，直到最后 Martin Fowler 写了一篇《[反转控制容器和依赖注入模式](#)》的文章，才算把大家的讨论做了一个总结，行业里总算是有了一个共识。

那段时间，DI 容器也得到了蓬勃的发展，很多开源项目都打造了自己的 DI 容器，Spring 是其中最著名的一个。只不过，Spring 并没有就此止步，而是在这样一个小内核上面发展出了更多的东西，这才有了我们今天看到的庞大的 Spring 王国。

讲到这里，你会想，那这和我们要讨论的“模型”有什么关系呢？

正如我前面所说，很多人习惯性把对象的创建和组装写到了一个类里面，这样造成的结果就是，代码出现了大量的耦合。时至今日，很多项目依然在犯同样的错误。很多项目测试难做，原因就在于此。这也从另外一个侧面佐证了可测试性的作用，我们曾在[第 3 讲](#)中说过：可测试性是衡量设计优劣的一个重要标准。

由此可见，在没有 DI 容器之前，那是怎样的一个蛮荒时代啊！

有了 DI 容器之后呢？你的代码就只剩下关联的代码，对象的创建和组装都由 DI 容器完成了。甚至在不经意间，你有了一个还算不错的设计：至少你做到了面向接口编程，它的实现是可以替换的，它还是可测试的。与之前相比，这是一种截然不同的思考方式，而这恰恰就是 DI 容器这个模型带给我们的。

而且，一旦有了容器的概念，它还可以不断增强。比如，我们想给所有与数据库相关的代码加上时间监控，只要在容器构造对象时添加处理即可。你可能已经发现了，这就是 AOP（Aspect Oriented Programming，面向切面编程）的处理手法。而这些改动，你的业务代码并无感知。

Spring 的流行，对于提升 Java 世界整体编程的质量是大有助益的。因为它引导的设计方向是一个好的方向，一个普通的 Java 程序员写出来的程序只要符合 Spring 引导的方向，那么它的基本质量就是有保障的，远超那个随意写程序的年代。

不过，如果你不能认识到 DI 容器引导的方向，我们还是无法充分利用它的优势，更糟糕的是，我们也不能太低估一些程序员的破坏力。我还是见过很多程序员即便在用了 Spring 之后，依然是自己构造对象，静态方法满天飞，把原本一个还可以的设计，打得七零八落。

你看，通过上面的分析，我们知道了，只有理解了模型设计的来龙去脉，清楚认识到它在解决的问题，才能更好地运用这个模型去解决后面遇到的问题。如果你是这个项目的维护者，你才能更好地扩展这个模型，以便适应未来的需求。

总结时刻

今天，我们学习了如何了解设计的第一部分：看模型。**理解模型，要知道项目提供了哪些模型，这些模型都提供了怎样的能力。**但还有更重要的一步就是，**要了解模型设计的来龙去脉。**这样，一方面，可以增进了我们对它的了解，但另一方面，也会减少我们对模型的破坏或滥用。

我以 Spring 的 DI 容器为例给你讲解了如何理解模型。DI 容器的引入有效地解决了对象的创建和组装的问题，让程序员们拥有了一个新的编程模型。

按照这个编程模型去写代码，整体的质量会得到大幅度的提升，也会规避掉之前的许多问题。这也是一个好的模型对项目起到的促进作用。像 DI 这种设计得非常好的模型，你甚至不觉得自己在用一个特定的模型在编程。

有了对模型的了解，我们已经迈出了理解设计的第一步，下一讲，我们来看看怎样理解接口。

如果今天的内容你只能记住一件事，那请记住：**理解模型，要了解模型设计的来龙去脉。**



思考题

最后，我想请你思考一个问题，DI 容器看上去如此地合情合理，为什么在其他编程语言的开发中，它并没有流行起来呢？欢迎在留言区写下你的思考。

感谢阅读，如果你觉得这一讲的内容对你有帮助的话，也欢迎把它分享给你的朋友。

更多福利推荐

充值限时膨胀

充 ¥500 得 ¥580

下单即赠精选爆款商品

戳此充值

充 ¥500 得 ¥580

充 ¥1000 得 ¥1200

充 ¥2000 得 ¥2500



© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 04 | 三步走：如何了解一个软件的设计？

下一篇 06 | Ruby on Rails：如何分析一个软件的接口？

精选留言 (21)

写留言



Jxin

2020-06-03

1.斟酌再三，虽说直接说spring di容器好像也没啥毛病，但个人觉得这描述并不是很准确，故阐述下自己的认知。

2.我认为spring提供的这个编程模型应该叫ioc（控制反转和响应式编程有点像）而不是di。因为最开始被提出的是ioc（好莱坞原则），而且最早的实现也不是spring，jdk和ej...
展开 ∨



27



刘丹

2020-06-03

```
container.bind(Connection.class).to(connection);  
container.bind(ArticleReposistory.class).to(DBArticleRepository.class);  
container.bind(ArticleService.class).to(ArticleService.class)
```

请问这3行代码的具体含义是啥？

展开 ∨

作者回复: 将 Connection 这个类绑定在 connection 这个对象上，当需要一个 Connection 对象时，返回 connection 这个对象。

将 ArticleReposistory 这个接口绑定在 DBArticleRepository 这个类上，当需要一个 ArticleReposistory 对象时，返回 DBArticleRepository 这个类的一个对象。

将 ArticleService 这个类就绑定在其自身，当需要一个 ArticleService 对象时，返回这个类的一个对象。



1

4



小鱼儿

2020-06-04

放下历史长河之中去看问题，比如 现在去看几年前甚至10年前的代码，才知道这样做的好处，分离关注点，可测试性是多么需要，不然真的改不动。

展开 ∨

作者回复: 这是我在开篇词里的立论，软件设计是一门关注长期变化的学问。





2



任旭东

2020-06-04

为什么不建议使用静态方法？如果只是简单的模型转换，用静态方法不是更好吗？

作者回复: 静态方法，没法去模拟它的行为，所以，要做测试的话，遇到静态方法，你必须关注它的实现，而不是它的接口。总的来说，静态方法是写着爽，但测着不方便。



2



Geek_3b1096

2020-06-03

你只是在了解别人设计的结果... 这就是我最欠缺的

展开 ▾

作者回复: 往前一步，你就成长了。



2



业余爱好者

2020-06-03

一个框架的流行根本原因不是它简化了开发，而是导致了问题的简化的那个开发模型。像spring 提供的di 模型，你甚至感受不到它的存在。它更像是一种理念，而这是一个模型的最高级形态。在di的核心模型之上，又出现了starter,auto configuration 等理念，这就是spring boot 的模型创新。在springboot 之上，又有springcloud.....

...

展开 ▾

作者回复: 没错，简化开发是结果，模型才是动因。

Spring Boot 是在 Spring 出现好多年之后才出现的。



2



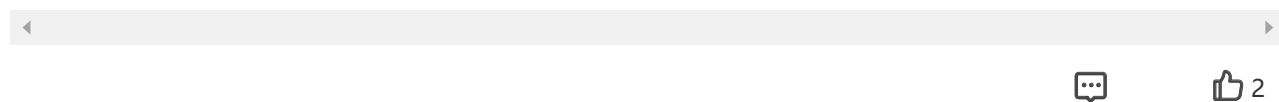
捞鱼的搬砖奇

2020-06-03

文中说“静态方法满天飞”是为了在实例方法中调用别的方法所以改为静态方法，是这样

的意思吗

作者回复: 为了方便, 定义静态方法, 到处调用, 然后, 没法 mock, 不好测试。



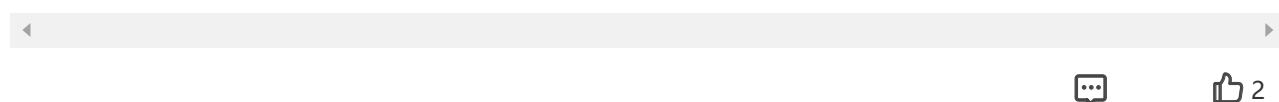
骨汤鸡蛋面

2020-06-03

接触spring 七八年, 一直在学习BeanFactory 和 ApplicationContext 上打转, 今天才算对容器这个概念有一个直觉性的认识, 感谢老师!

展开 ▾

作者回复: 学习一个软件, 要从基础模型开始。



JohnnyBOY

2020-06-03

回答作业:

- 1, Java有反射, 其他语言不一定有;
- 2, Java生态比较完善, 大神比较多, 有模版可以学;
- 3, 前端开发集中在UI界面和数据解析, 需求变更快, 用DI容器去做有点吃力; (UI大多是包含的方式, 很难把子控件拎出来初始化) ...

展开 ▾



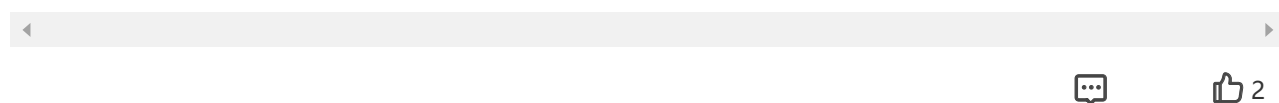
Asanz

2020-06-03

DI是模型? 我理解的DI是一种实现, IoC是模型 😊

展开 ▾

作者回复: 当年, IoC、DIP和 DI几个名字争论了好久, 最后决定叫了DI, 这个几个词确实有很多类似的地方。其实, 它们都是设计原则。后面讲设计原则的时候, 还会提到DIP的。



肥宅码农

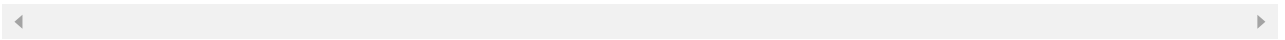
2020-06-03

我觉得最根本原因是大多数开发不写测试，所以不会考虑依赖问题，大多数方法都是面向实现而不是接口，使用DI容器反而增加了工作量。

目前所在小组偏向于外包，代码只有一层，不是单列，就是静态方法；为了达到快速交付，基本没有设计，不管怎么说这都是不合理的。是前人挖，后人跳。

展开 ∨

作者回复: 唉，你说的现状，我非常理解。所谓的“快”，只是从当前一个时点上看，放在长期，就是越跑越慢。



💬 1

👍 2



阳仔

2020-06-03

理解软件设计中模型首先要理解模型解决的核心问题是什么，然后抽丝剥茧了解模型的来龙去脉，深入理解模型解决问题的过程。

spring中的di模型是为了解决对象的创建和组装的问题。

那为什么创建对象和组装要用di来解决？

一个重要的原因是为了解耦。分离接口与实现的强依赖，也就是软件设计第一步分离关...

展开 ∨

💬

👍 2



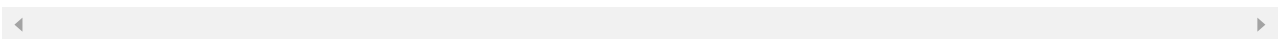
Kăfkă²⁰²⁰

2020-06-03

多半因为Java在企业级应用里独占鳌头，所以Java的DI更为人所知，也因为更早地出现了容器级的DI，Java才这么流行

展开 ∨

作者回复: DI没有成为主流时，Java也已经很流行了，比如，J2EE。



💬

👍 2



一步

2020-06-05

我感觉 老师文中讲的是 编程模型不是一个项目的功能模型（功能建模或者 DDD领域建模）。我认为在熟悉一个项目的时候，领域模型是首先要看的，编程模型的选择只是为了接口 或者 实现 提供基础

💬

👍 1



王智

2020-06-04

读第二遍：模型和模型设计，按照上面所说，Spring DI容器就是一个模型，那好的模型是不是就可以说是一个模型设计呢？按照现在的理解，模型确实不是早期脑海中的类了，早期就觉得类就是模型，从现在看看远远不足呀，模型的具体定义有点模糊了，希望能一直跟下去，看完之后再过一边可能会学到更多的东西。

展开 ∨



1



王智

2020-06-04

我觉得是因为Java面向对象，更多的是使用组合解决问题，使用组合那就避免不了对象的依赖，加上接口实现分离，就更加依赖于DI，而像其他的语言，像面向过程全程使用函数来解决问题，貌似有点用不到对象的组合和创建。我的一点小理解，也不知道有没有问题。

展开 ∨

作者回复: 面向对象和面向过程只是用到了不同的设计元素，其实，使用程序设计语言完全可以兼顾二者，稍后，我会在编程方范式部分进一步讲解。



1



不记年

2020-06-04

郑老师你好,文中模型感觉和我理解的模型不一样,我觉得模型是软件内在的东西,是其独有的东西,是对问题域的建模.

拿spring DI来说,问题域是如何自动化对象的创建和组装

对问题域建模后,我们的模型由哪几部分组成,各部分之间怎么交互的,我认为这个是模型.至于文中提到的编程模型,我觉得理解成接口更加合适.

展开 ∨



1



escray

2020-06-03

其实很早就听说过 Inversion of Control 和 Dependency Injection，但是似乎一直没有搞明白其中的概念，也没有会有意识的去使用 DI（也许是用了，但是没有意识到）。

重读了 Martin Fowler 的长（旧）文，有一个疑惑，专栏里面的 Spring DI 是属于哪一种类型的 IoC，看上去比较像 type 1，Constructor Injection。...

展开 ∨



1



宝宝太喜欢极客时间了

2020-06-03

还是感觉模型是个很虚得东西，只可意会不可言传，能不能通过一种方式把他表现出来呢？比如图形？



1



六一王

2020-06-07

我是一个两年的前端程序媛，不太了解 java。面向对象编程时，你只想要一颗树，却得到整个森林，于是有些人就觉得面向对象编程是不好的，所以认为函数式编程的方式更好，不过文章提到的组件创建和组合放入到一个容器中，也就是说将所有依赖都放入都一个地方，提供业务需要的接口，而不写到业务中，那么为啥没有在前端火起来呢？函数式编程是不是就是面向接口编程的一种呢？

展开 ∨

作者回复: 后面会讲到函数式编程，简单来说，面向对象提供了组织类的能力，函数式编程提供了组织动作的能力，二者可以混合使用。

