Atitit **深入理解耦合Coupling的原理与attilax总结**

目录

[1.1. 耦合作为名词在通信工程、软件工程、机械工程等工程中都有相关名词术语。 2](#_Toc18769)

[1.2. 所有的耦合形式可分为5类： 2](#_Toc5384)

[1.3. 耦合可以分为以下几种，它们之间的耦合度由高到低排列如下： 3](#_Toc12495)

[1.4. 耦合强度，依赖于以下几个因素：](#_Toc31660)[[2]](#_Toc31660)[4](#_Toc31660)

[2. 依赖是怎么产生的 4](#_Toc12206)

[2.1. 接口管理 5](#_Toc18506)

[2.2. 状态管理 5](#_Toc18750)

[2.3. 功能管理 6](#_Toc27640)

[2.4. 依赖来自于约束 6](#_Toc16697)

[3. 接触耦合的几种模式 7](#_Toc16027)

[3.1. 适配器模式进行解耦 7](#_Toc28344)

[3.1.1. 无状态的函数式编程？ 7](#_Toc18053)

[3.1.2. 服务化 7](#_Toc9184)

[3.2. 事件驱动观察者模式 回调函数 comsumer 7](#_Toc17920)

[3.3. 使用数据库接口进行接触耦合 7](#_Toc14265)

[3.4. mq redis类事件驱动异步模式 7](#_Toc1583)

[4. 微服务架构：如何用十步解耦你的系统？-Java知音 7](#_Toc27673)

[4.1. 第一步，解耦现有模块 7](#_Toc20969)

[4.2. 第二步，抽取公共模块 7](#_Toc30598)

[4.3. 第四步，配置解耦 8](#_Toc26744)

[4.4. 第五步，权限解耦 8](#_Toc31606)

[4.5. 8](#_Toc12142)

[4.6. 七步，数据解耦 8](#_Toc12642)

[4.7. 第八步，扩容解耦 8](#_Toc25571)

[4.8. 第九步，部署解耦 9](#_Toc11641)

[4.9. 第十步，动静解耦 9](#_Toc20184)

[5. 二、典型耦合与对应解耦实践 9](#_Toc19459)

[5.1. 1、IP耦合 9](#_Toc23965)

[5.2. 2、公共库耦合 10](#_Toc18091)

[5.3. 3、数据库耦合 10](#_Toc18019)

[5.4. 4、服务化耦合 10](#_Toc6471)

[5.5. 5、消息通知耦合 11](#_Toc25186)

[5.6. 6、下游扩容耦合 11](#_Toc23016)

[6. ref 12](#_Toc29250)

## 耦合作为名词在[通信工程](http://baike.baidu.com/view/17355.htm" \t "http://baike.baidu.com/item/_blank)、[软件工程](http://baike.baidu.com/view/1659.htm" \t "http://baike.baidu.com/item/_blank)、机械工程等工程中都有相关名词术语。

耦合是指两个或两个以上的电路元件或电网络等的输入与输出之间存在紧密配合与相互影响，并通过相互作用从一侧向另一侧传输能量的现象。

耦合作为名词在[通信工程](http://baike.baidu.com/view/17355.htm" \t "http://baike.baidu.com/item/_blank)、[软件工程](http://baike.baidu.com/view/1659.htm" \t "http://baike.baidu.com/item/_blank)、机械工程等工程中都有相关名词术语。

1. 1 [主要分类](http://baike.baidu.com/item/%E8%80%A6%E5%90%88" \l "1)
2. ▪ [多场耦合](http://baike.baidu.com/item/%E8%80%A6%E5%90%88" \l "1_1)
3. ▪ [能量耦合](http://baike.baidu.com/item/%E8%80%A6%E5%90%88" \l "1_2)
4. ▪ [数据耦合](http://baike.baidu.com/item/%E8%80%A6%E5%90%88" \l "1_3)
5. ▪ [标记耦合](http://baike.baidu.com/item/%E8%80%A6%E5%90%88" \l "1_4)
6. ▪ [控制耦合](http://baike.baidu.com/item/%E8%80%A6%E5%90%88" \l "1_5)
7. ▪ [外部耦合](http://baike.baidu.com/item/%E8%80%A6%E5%90%88" \l "1_6)
8. ▪ [公共耦合](http://baike.baidu.com/item/%E8%80%A6%E5%90%88" \l "1_7)
9. ▪ [内容耦合](http://baike.baidu.com/item/%E8%80%A6%E5%90%88" \l "1_8)
10. ▪ [非直接耦合](http://baike.baidu.com/item/%E8%80%A6%E5%90%88" \l "1_9)
11. ▪ [另类情况](http://baike.baidu.com/item/%E8%80%A6%E5%90%88" \l "1_10)

## 所有的耦合形式可分为5类：

耦合是系统设计中最重要的概念之一，也是设计中真正的基本原则之一。所谓耦合，指的是对某元素与其他元素之间的连接、感知和依赖程度的度量。在一个OO系统中，所有的耦合形式可分为5类：

l  零耦合（nil coupling）：两个类丝毫不依赖于对方。

l  导出耦合（export coupling）：一个类依赖于另一个类的公有接口。

l  授权耦合（overt coupling）：一个类经允许，使用另一个类的实现细节。

l  自行耦合（covert coupling）：一个类未经允许，使用另一个类的实现细节。

l  暗中耦合（surreptitious coupling）：一个类通过某种方式知道了另一个类的实现细节。

零耦合当然是耦合度最低的。两个丝毫互不依赖的类，意味着在维护和扩展系统时，可以随意地去掉或者修改其中的一个类而丝毫不会影响到另一个类。但是，只使用零耦合却无法创建出一个有意义的OO系统，因为所有的类都是独立、不相关的，相互之间没有消息的传递，这样最多只能创建出一个类库。导出耦合具有相当低的耦合度，因为在导出耦合中，一个类只依赖另一个类的公有接口。在一个设计良好的系统中，消息的传递只会通过类的公有接口进行，因而导出耦合可以很好地支持系统的可维护性与可扩展性。除此之外，授权耦合、自行耦合、以及暗中耦合都是耦合程度比较高的耦合形式。

有这样一条OO设计的经验原则：类与类之间应该零耦合，或者只有导出耦合关系。也即，一个类要么同另一个类毫无关系，要么就只使用另一个类提供的公有接口。授权耦合、自行耦合、暗中耦合基本上不应该在系统中被使用到。

。高耦合本身也并不是问题之所在，问题是与某些方面不稳定的元素之间的高耦合，这种高耦合会严重影响系统将来的维护性和扩展性。而比如所有的Java系统都能安全地将自己去Java库（java.lang，java.util等）进行耦合，因为Java库是稳定的，与Java库的耦合不会给系统的灵活性、维护性、扩展性带来什么问题。

## 耦合可以分为以下几种，它们之间的耦合度由高到低排列如下：

简单地说，[软件工程](http://baike.baidu.com/view/1659.htm" \t "http://baike.baidu.com/item/_blank)中对象之间的[耦合度](http://baike.baidu.com/view/1599212.htm" \t "http://baike.baidu.com/item/_blank)就是对象之间的依赖性。指导使用和维护对象的主要问题是对象之间的多重依赖性。对象之间的耦合越高，维护成本越高。因此对象的设计应使类和构件之间的耦合最小。

有软硬件之间的耦合，还有软件各模块之间的耦合。

[耦合性](http://baike.baidu.com/view/1731634.htm" \t "http://baike.baidu.com/item/_blank)是程序结构中各个模块之间相互关联的度量。它取决于各个模块之间的接口的复杂程度、调用模块的方式以及哪些信息通过接口。

耦合可以分为以下几种，它们之间的耦合度由高到低排列如下：

（1） 内容耦合。当一个模块直接修改或操作另一个模块的数据时，或一个模块不通过正常入口而转入另一个模块时，这样的耦合被称为内容耦合。内容耦合是最高程度的耦合，应该避免使用之。

（2） 公共耦合。两个或两个以上的模块共同引用一个全局数据项，这种耦合被称为公共耦合。在具有大量公共耦合的结构中，确定究竟是哪个模块给全局变量赋了一个特定的值是十分困难的。

（3） 外部耦合 。一组模块都访问同一全局简单变量而不是同一全局数据结构，而且不是通过参数表传递该全局变量的信息，则称之为外部耦合。

（4） 控制耦合 。一个模块通过接口向另一个模块传递一个控制信号，接受信号的模块根据信号值而进行适当的动作，这种耦合被称为控制耦合。

（5） 标记耦合 。若一个模块A通过接口向两个模块B和C传递一个公共参数，那么称模块B和C之间存在一个标记耦合。

（6） 数据耦合。模块之间通过参数来传递数据，那么被称为数据耦合。数据耦合是最低的一种耦合形式，系统中一般都存在这种类型的耦合，因为为了完成一些有意义的功能，往往需要将某些模块的输出数据作为另一些模块的输入数据。

（7） 非直接耦合 。两个模块之间没有直接关系，它们之间的联系完全是通过主模块的控制和调用来实现的。

**总结**：耦合是影响软件复杂程度和设计质量的一个重要因素，在设计上我们应采用以下原则：如果模块间必须存在耦合，就尽量使用数据耦合，少用控制耦合，限制公共耦合的范围，尽量避免使用内容耦合。

## 耦合强度，依赖于以下几个因素：[2]

（1）一个模块对另一个模块的调用；

（2）一个模块向另一个模块传递的数据量；

（3）一个模块施加到另一个模块的控制的多少；

（4）模块之间接口的复杂程度。

# 依赖是怎么产生的

既然要研究怎么让模块解耦，那当然要从根源来分析：依赖它到底从何而来？

依赖其实是在我们想把代码写好的那一刻开始产生的。为什么这么说呢？因为大多数代码都是可以通过像流水线一样写下来，最终变成一个几千行的函数、几万行的单个文件。这个时候甚至没有拆分成模块，也就更谈不上所谓依赖和解耦了。

忽然有一天，我们发现这种堆屎山的日子实在过的没有意思，开始研究怎么将一座大屎山拆成几个小屎山，然后再一点点清理干净。依赖的产生，就在我们一拆多的这个过程伴随出现的。

## 接口管理

## 状态管理

由接口管理产生的依赖通常来自外部，而应用内部也会有依赖的产生，常见的包括状态管理和事件管理。我们先来看看状态管理。

一个应用程序能按照预期正常运行，必然无法避免一些状态的管理。最简单的，生命周期就是一种状态。程序是否已经启动、功能是否正常运行、输入输出是否有变化，这些都会影响到程序的运行状态。

由于程序会有状态变化，因此我们的功能实现必然依赖程序的状态。例如，只有用户登录了才能进行更多的操作、订单产生了才可以进行撤销、界面渲染完成了用户才可以点击，等等。

从代码可读性和可维护性角度来看，面向对象编程近些年来还是稍胜于函数式编程，面向对象的设计本身就是状态设计的过程，而某个对象的运行结果，也会依赖于该对象的状态。

这是来自于对某个程序“按照预期运行”进行合理设计而产生的依赖。

## 功能管理

当我们根据功能将代码拆分成一个个模块之后，功能模块的管理也同样会产生一些依赖。

## 依赖来自于约束

为了方便管理，我们设计了一些约定，并基于“大家都会遵守约定”的前提来提供更好、更便捷的服务。

举个例子，前端框架中为了更清晰地管理渲染层、数据层和逻辑处理，常用的设计包括 MVC、MVVM 等。而要使这样的架构设计发挥出效果，我们需要遵守其中的使用规范，不可以在数据层里直接修改界面等。

可以看到，依赖来自于对代码的设计。

# 接触耦合的几种模式

## 适配器模式进行解耦

### 无状态的函数式编程？

### 服务化

## 事件驱动观察者模式 回调函数 comsumer

## 使用数据库接口进行接触耦合

## mq redis类事件驱动异步模式

# 微服务架构：如何用十步解耦你的系统？-Java知音

## 第一步，解耦现有模块

将现有耦合在一起的模块进行重新的设计，设计成可以独立部署的多个模块，使用微服务框架很容易做到，成熟的示例代码都特别多，这里不再多讲。下面是我的微服务实现的一个架构设计图。

## 第二步，抽取公共模块

## 第四步，配置解耦

为每个模块每个环境配置一个配置文件，这样就可以把不同的环境的配置解耦，不用每次上线都更新一次。但是如果需要修改数据库配置，还是需要重新部署重启应用才能解决。使用微服务的配置中心就能解决这个问题了，比如使用ZooKeeper作为SpringCloud的配置中心，修改ZooKeeper中的节点数据就可以实时更新配置并生效。

## 第五步，权限解耦

## 

## 七步，数据解耦

系统刚上线的时候，数据量不大，所有的模块感觉都挺好的，当时间一长，系统访问量非常大的时候会发现功能怎么都变慢了，怎么mysql的cpu经常100%。那么恭喜你，你中招了，你的数据需要解耦了。

首先要模块间数据解耦，将不同模块使用独立的数据库，保证各模块之间的数据不相互影响。

其次就是冷热数据解耦，同一个模块运行时间长了以后也会积累大量的数据，为了保证系统的性能的稳定，要减少因为数据量太大造成的性能降低，需要对历史数据进行定期的迁移，对于完整数据分析汇总就在其他的库中实现。

## 第八步，扩容解耦

一个好的架构设计是要有好的横向扩展的能力，在不需要修改代码只通过增加硬件的方式就能提高系统的性能。SpringCloud和Dubbo的注册中心天生就能够实现动态添加模块的节点，其他模块调用能够实时发现并请求到新的模块节点上。

## 第九步，部署解耦

## 第十步，动静解耦

当同一个模块的瞬间有非常高并发的时候，对，就是说的秒杀，纯粹的流量解耦还是不够，因为不能让前面的流量冲击后面真正的下单的功能，这个时候就需要更细的流量解耦，要将静态文件的访问通通抛给CDN来解决，动态和静态之间是通过定时器来出发的，时间未到之前一直刷新的是静态的文件，当时间到了之后，生成新的js文件，告诉静态页面可以访问下单功能了。

# 二、典型耦合与对应解耦实践

## 1、IP耦合

内网IP修改为内网域名，这是我们的实践，强烈的建议大家回去马上干这个事情。为什么我们IP要修改、要重启？很有可能是我们将IP写在了自己的配置文件中。如果我们把这个内网IP变为内网域名，那么我们是不是就可以不让上游配合去改配置重启呢？

假设我们现在不用IP了，用域名了。现在换了一台机器域名没变，IP指向变了。我们可以让运维统一将内网DNS切到新的机器上面去，并将旧机器的连接切断，重连后就会自动连到新机器上去了。这样的话只要运维配合就可以完成迁移，对于所有上游的调用方、服务的调用方、数据的调用方都不需要动，这是第一个案例。

我们的最佳实践是强烈建议使用内网域名来替换内网的IP，连服务、连数据库统统取走。

## 2、公共库耦合

第二个案例是公共库，这个公共库可能是一个跟业务相关的通用业务库，比如用户的业务、支付的业务，这些业务写在了一个jar包里，各个业务线通过这个jar包来实现相关的一些业务逻辑。所有的业务方因为这个公共库耦

如果这个jar包、这个公共库的个性比较强，如果时候偏招聘的、房产的、二手的，我们的建议是把这些个性的代码拆分到各个业务线自己的jar包里面去，这样的话，你修改的那一块只影响你自己，至少不会扩大影响范围，这个需要对业务进行剖析，把个性的地方拿出来。如果长时间解决不了，我刚刚说的那种耦合如果频发

## 3、数据库耦合

导致什么问题呢？比如A业务线要上线一个功能，这个功能没有索引，全表都要扫描，数据库CPU100%，数据库实例IO性能下降，影响业务。B和C都影响。即某个业务线的数据库性能急剧下降导致所有业务都受影响。这时DBA兄弟、运维兄弟杀过来说性能不行了，我再给你两台机器，给我两个实例，你会发现没用，所有表都耦合在一个实例里，给机器也拆不开，扩不了容。

2015年我调去58到家时，当时整个58到家有一个库叫做58到家库，里面有几百个表，性能越来越低，但因为各种join又必须耦合在一个实例里，很悲惨。我们怎么做呢？垂直切分与服务化。你会发现跟jar包解耦非常相似，垂直拆分。

比如说用户的基础数据，我抽向一个用户的服务，user最基础的数据库只能够被这个服务锁访问，数据库私有是服务化的一个特点。

## 4、服务化耦合

第四个案例是服务化耦合的例子。服务化之后，如果业务代码拆分得不干净，即使你做了服务化也不能够解除耦合。这里举一个服务化解耦不彻底的案例。

耦合范围相对较小，因为只有一个基础服务维护的是痛点。解决方案也很容易想到，当然是把业务个性化的case分支搬到上游去，底层只做通用的功能。业务代码上浮，这样的话上游的业务迭代速度、迭代效率会提升，每块业务有功能就会自己实现了，不需要兄弟部门去实现，没有一个沟通的过程。这是服务化不彻底的一个常见的耦合的案例。

## 5、消息通知耦合

第五个案例是消息通知的耦合。我猜应该也很多公司会遇到过，有一些事件，这个事件可能要让很多下游知晓，这里举一个我们曾经出现过的案例。58同城发布帖子，发布帖子的这个事件可能要周知很多方，例如有一个用户分级的服务，他发了帖之后，这个用户发帖的一些统计数据，一些信息数据可能要进行更新。还要通知离线消息反作弊的部门发布

就是这种因为消息上下游耦合在一起，非常常见的解耦方案是通过MQ，这个案例里的MQ以及下一个案例里的配置中心是互联网架构中两个非常常见的解耦工具。MQ能够做到上下游物理上和逻辑上都解耦，增加MQ之后，首先上游互不知道彼此的存在，它当然不会建立物理连接了，大家都与MQ建立物理连接，就是物理连接上解耦了；逻辑上也解耦

就是这种因为消息上下游耦合在一起，非常常见的解耦方案是通过MQ，这个案例里的MQ以及下一个案例里的配置中心是互联网架构中两个非常常见的解耦工具。MQ能够做到上下游物理上和逻辑上都解耦，增加MQ之后，首先上游互不知道彼此的存在，它当然不会建立物理连接了，大家都与MQ建立物理连接，就是物理连接上解耦了；逻辑上也解耦了，消息发布方甚至不用知道哪些下游订阅了这个消息。新增消息的订阅方只需要找MQ就行了，上游不需要关注。所以MQ是一个非常常见的物理上解耦、逻辑上也解耦的利器。

## 6、下游扩容耦合

第六个案例，我相信也几乎是所有的公司都会遇到的一个案例，它和第一个案例很像，但又不一样。我们的第一个案例是说IP变化，上游

解决方案其实还是配置中心，配置中心的细节我在这不展开讲，网上可能也有一些公司的实践，配置后台、DB存储等。

一、如何找到系统中的耦合

什么是耦合？就是每每我们作为技术人，在心中骂上下游、骂兄弟部门，说这个东西跟我有什么关系？为什么我要配合来做这个事情？这里面就非常有可能是系统中存在耦合的地方。

明明我们不应该联动，但兄弟部门要做一个事情，上下游要做一个事情，我却要被动地配合来做这个事情。还有可能这个配合的范围特别特别的大，那就说明耦合非常非常的重。下面来看具体的五六个案例

# ref

**GRASP设计模式及OO设计原则浅谈-钟声的博客-搜狐博客!!!.htm**

**耦合\_百度百科.htm**

**GRASP模式 - 刘伟技术博客 - 博客频道 - CSDN.NET.htm**

转载请注明来源：attilax的专栏  http://blog.csdn.net/attilax

--Atiend