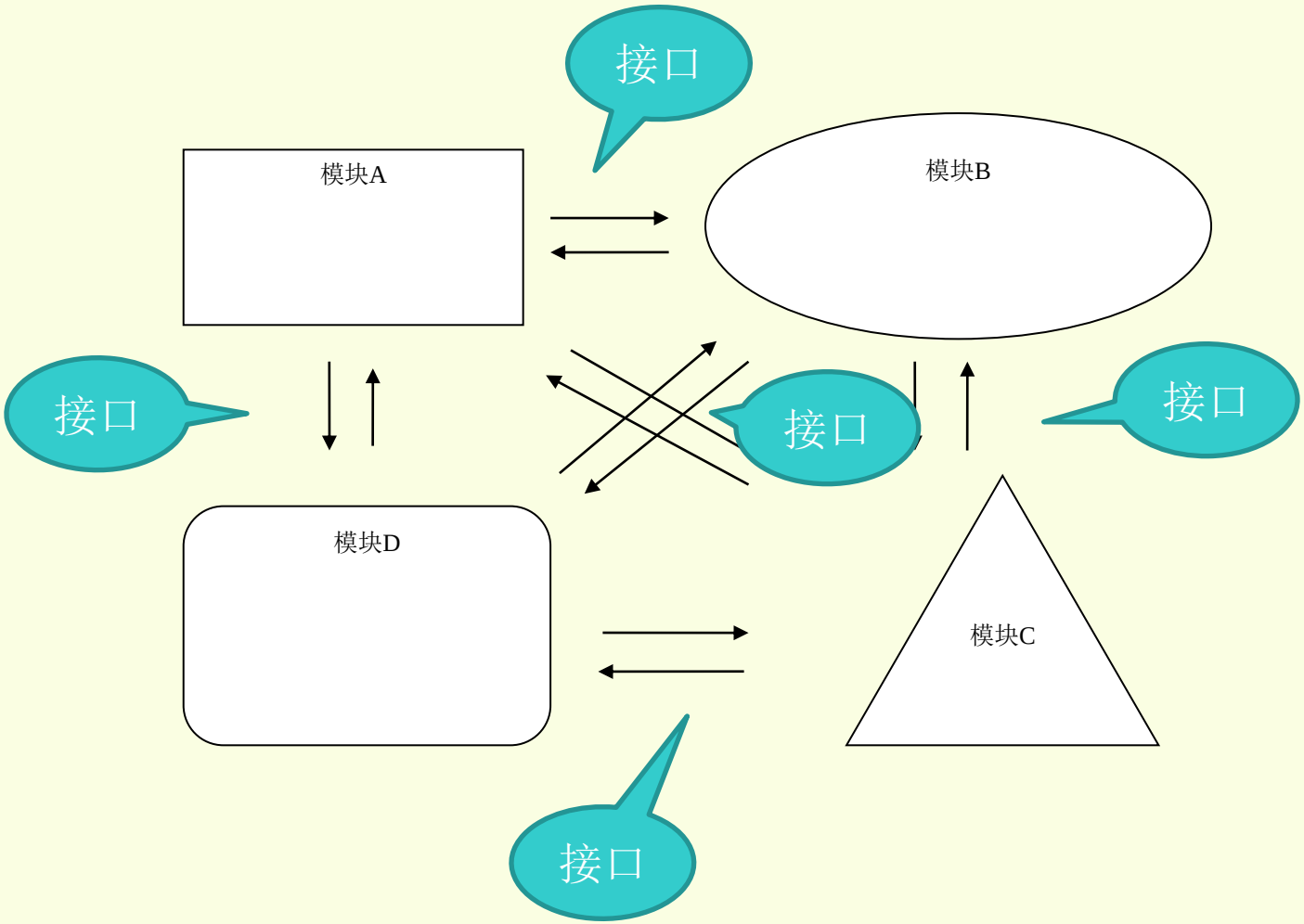
逻辑框架介绍

软件研究院
二零一二年

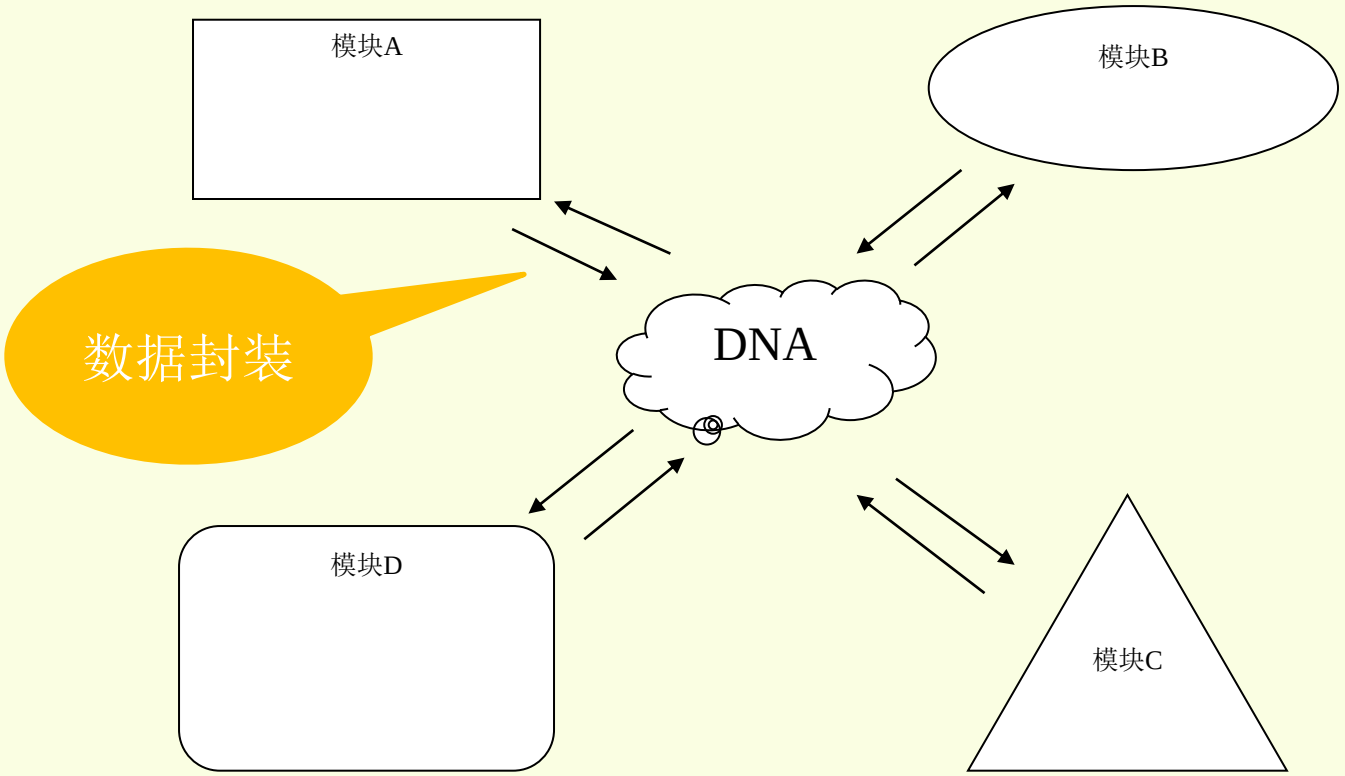
- 企业应用系统的开发通常采用分层结构，如通常分为界面逻辑层、业务逻辑层和数据访问层。层与层之间的调用则采用接口的方式，即耦合关系体现在接口上。
- 在实际应用中，由于业务模块的不断增加和调整，相关接口变得异常庞大和不稳定，开发人员不得不随着接口的变动对程序进行调整，增加了系统的不稳定性。


接口
模块A
模块B
接口
接口
接口
模块D
模块C
接口

- 另外，通过接口方法进行调用的方式，诸如数据库事务、资源锁定、权限控制、日志记录等逻辑和相关环境都需要程序编写者自行管理，一方面增加了程序的复杂度，一方面给程序编写者过大的自由空间，程序的稳定性和可维护性较差。
- DNA逻辑框架的目的就是解决分层架构中，界面逻辑和业务逻辑之间、不同的模块的业务逻辑之间的程序调用的问题。

- DNA逻辑框架负责程序逻辑的运行过程，这使得数据库事务、资源锁定等逻辑可以由框架负责完成，另外，框架对异步调用、分布式调用也将提供支持。
- 框架提供机制来组织功能模块，使得业务逻辑不依赖接口组织，实现者可以根据需要任意的进行重构，而不影响到调用者。同时，框架提供上下文环境来对提供各种基础服务，包括权限访问、资源管理等等，业务逻辑的编写者将被限定在一个受约束的框架中。

面向数据的软件架构



- DNA逻辑框架负责程序逻辑的运行过程，这使得数据库事务、资源锁定等逻辑可以由框架负责完成，另外，框架对异步调用、分布式调用也将提供支持。
- 基于DNA逻辑框架进行应用程序的开发，将在设计和管理模式上发生较大的改变。
- 设计模式将从传统的面向对象和接口的模式，转向以数据处理为核心的设计模式。管理上，功能模块的组织属于内部实现，不需要进行管理，而数据处理任务则必须严格的进行文档化，对管理有较高的要求。
- 依赖文档管理而不是接口组织不断扩展的功能模块，将使得大型应用系统的开发和管理变成可能。

- DNA逻辑框架（以下简称框架）解除了功能的调用方与实现方之间在接口上的紧密耦合关系，而这一特性的实现正是利用了框架对调用的集中管理机制。和传统的接口调用的模式不同，对功能的调用都需要通过框架的帮助才能完成，而不能直接调用和执行。

- 在框架中，功能的实现方（服务提供者）把对相关功能的实现作为服务注册到系统中，而调用方则根据需要把使用特定功能所需的数据组织为任务或者查询凭据，进而请求框架来处理相应的任务或者返回查询凭据所对应的查询结果。而框架则负责根据特定的任务或者查询凭据来选择执行对应的功能实现（任务处理程序或者查询过程）。

- 至此可以发现，无论是数据处理还是查询，功能模块的调用者，面向的都是“数据”，而不是模块的“实现”。因此，框架确实解除了功能的调用方与实现方之间在接口上的紧密耦合关系。而这一特性使得两方不再依赖于接口的定义，而是完成特定功能所需的数据。进而结合功能模块的组织还可以发现，框架对调用的管理使得调用者不仅不需要关心实现的细节，更重要的是还不需要关心实现的位置。

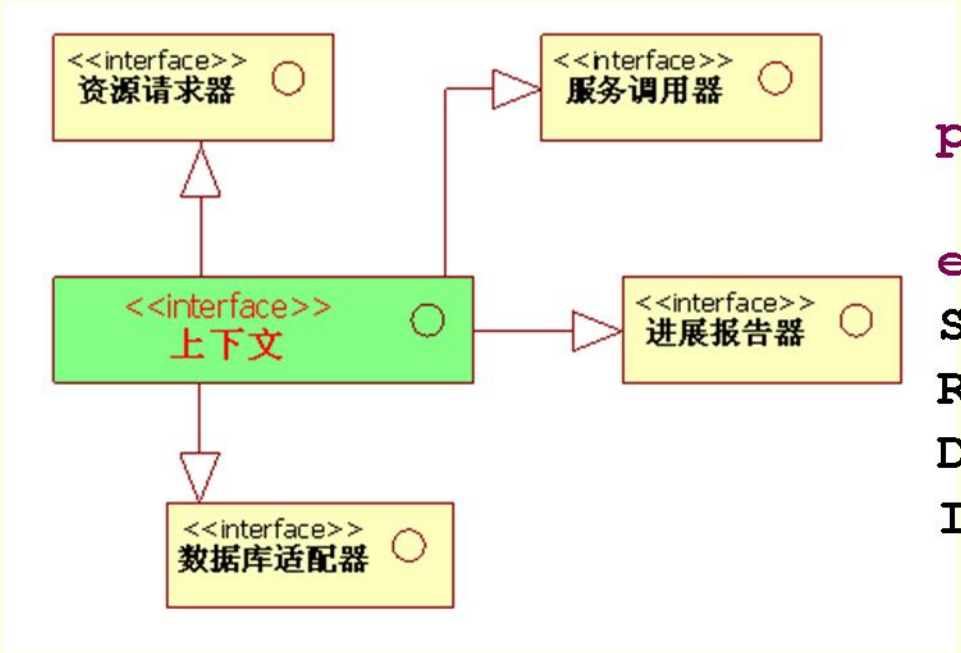
- 这就使得服务方可以根据需要灵活组织功能的实现，甚至是在将来还可以进行必要的重构，只要双方所关心的“数据”没有变化，这些改变就不会影响到调用方的正常运行。这也是框架对调用进行管理的精髓所在。

提供上下文环境

- 框架提供的一个重要功能就是为功能的实现者（服务）及调用者提供上下文环境。框架是通过上下文对象（Context）提供上下文环境的。上下文对象是调用者与服务提供者之间的直接接口和总接口。上下文对象主要封装了对任务处理、数据查询、资源操作等功能的调用，另外还对事务处理进行控制和保证。
- 简言之，使用框架的上下文对象是访问框架中的服务的唯一的也是唯一需要的途径。每个服务中功能的实现部件的实现方法中都可以得到一个上下文对象，从而使用这个对象完成所需的执行过程。

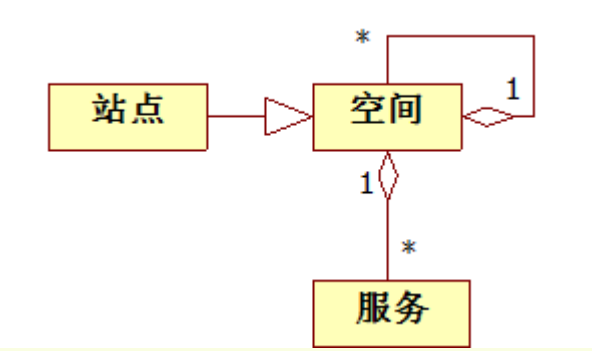
提供上下文环境

- 上下文(Context）是我们开发过程中用的最多的一个接口，开发人员和框架打交道就是通过它来完成，可以说它是系统调用的中枢。
- 生命周期为线程级，在一个请求内部。




```
public interface
   Context
extends
ServiceInvoker,
ResourceQuerier,
DBAdapter,
InfoReporter
```

- 主要指的是系统功能模块的规划与设计。在大型系统中，模块的规划十分重要，这直接影响到是否能够在最大程度上发挥项目成员的协同能力，同时也影响着系统的健壮性与可扩展能力。而 DNA 逻辑框架为这一目标提供了强有力的支持。
- 框架通过站点（Site）、空间（Space）和服务（Service）三类对象来组织功能模块。



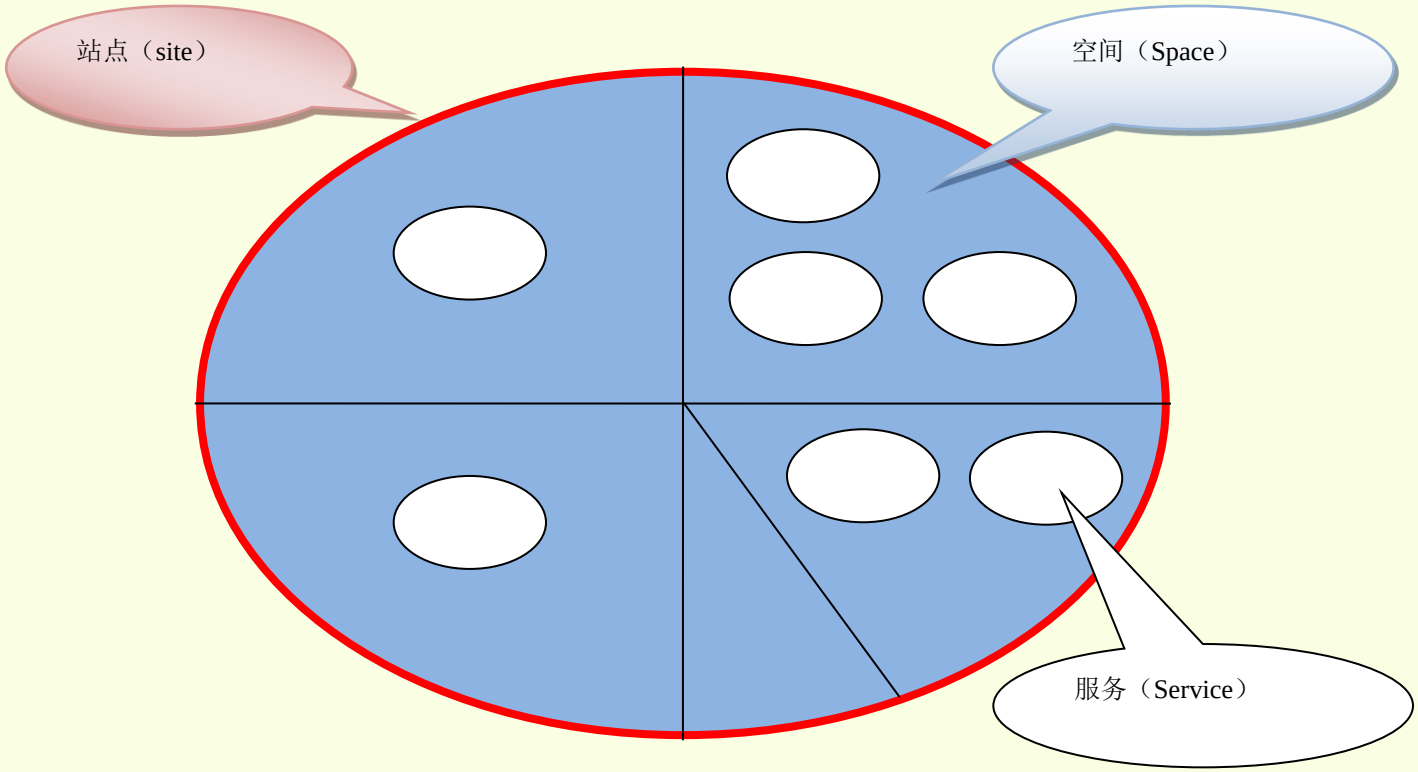
- 从这个意义上来讲，三者其实就是一种功能或者功能模块的集合，且由前向后依次为包含关系，对功能的模块划分最终会形成一种树形的结构。站点是一个应用系统实例级的概念，通常一个应用中只有一个站点，但框架也支持多个站点的情况。
- 空间是功能模块的目录，框架支持多级空间结构，即空间中可以包含其它的空间，最终形成的是一种树形结构。从形式上来看，类似于文件系统中的目录结构。

站点（Site）

- 空间（Space）[1..*]
  - 服务（Service）[0..*]
    - 初始化方法（init()）[0..1]
    - 任务处理器（TaskMethodHandler）[0..*]
    - 结果（列表）提供器（Result(List)Provider）[0..*]
  - 资源服务（ResouceService）[0..*]
    - 资源声明（Tfacade, Timpl, TKeysHolder）[1]
    - 初始化方法（initResources()）[0..1]
    - 资源提供器（ResourceProvider）[1..*]
  - 信息定义（Infomation）[0..*]
  - 数据库表（Table）[0..*]
  - 数据库查询（Query）[0..*]
  - 数据库命令（Command）[0..*]

- 服务是功能模块的最小单位，其中可包含多个不同类型的实现具体功能的部件。如专门服务于查询的结果提供器，用于实现各种维护操作的任务处理器，以及具有特定功能的资源提供器等等。
- 以上三者为模块的划分与组织提供了结构基础。

- 从另外一个角度来看DNA下的功能的组织：



- 服务（Service）是在DNA下编写逻辑代码的地方，是我们处理数据、请求数据实现的场所。
- 它从 com.jiuqi.dna.core.service.Service 抽象类继承而来。
- 根据实际需要，我们可以在构造方法里面引入依赖的元数据（如逻辑表定义、数据库命令定义、orm定义等），可以选择是否重写初始化方法，并且开发具体的任务处理器与结果（列表）提供器以实现数据的处理和对外提供。

- 首先解释一下什么叫元数据。元数据最本质、最抽象的定义为：**data about data (**关于数据的数据**)**。它是一种广泛存在的现象，在许多领域有其具体的定义和应用。如：在图书馆与信息界，元数据被定为提供关于信息资源或数据的一种结构化的数据，是对信息资源的结构化的描述；在数据仓库领域中，元数据被 被定义为描述数据及其环境的数据；在软件构造领域，元数据被定义为在程序中不是被加工的对象，而是通过其值的改变来改变程序的行为的数据。

- 元数据的注入实际上就是依赖注入，它是一种设计模式，就是现在大家所说的IoC（Inversion of Control，反转控制）。我们可以把IoC模式看做是工厂模式的升华，IoC就是一个大工厂，只不过这个大工厂里要生成的对象都是在XML文件中给出定义的，然后利用Java 的“反射”编程，根据XML中给出的类名生成相应的对象。从实现来看，IoC是把以前在工厂方法里写死的对象生成代码，改变为由XML文件来定义，也就是把工厂和对象生成这两者独立分隔开来，目的就是提高灵活性和可维护性。

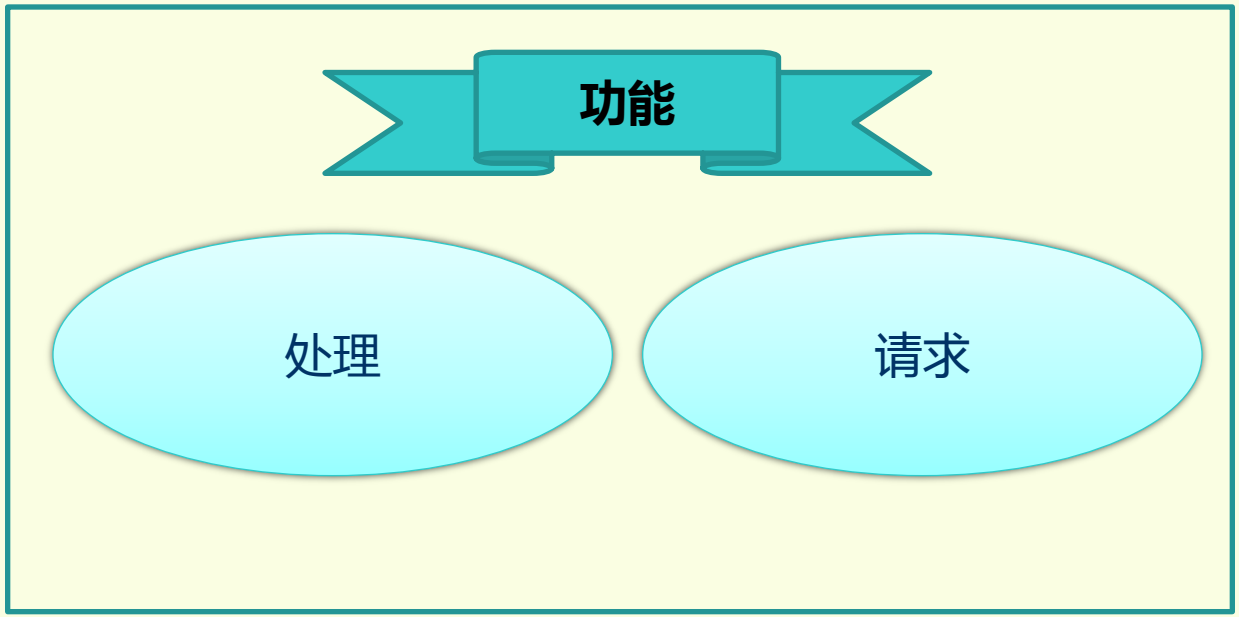
- IoC中最基本的Java技术就是“反射”编程。反射又是一个生涩的名词，通俗的说反射就是根据给出的类名（字符串）来生成对象。这种编程方式可以让对象 在生成时才决定要生成哪一种对象。反射的应用是很广泛的，象Hibernate、String中都是用“反射”做为最基本的技术手段。
- IoC最大的好处是当我们需要换一个实现子类将会变成很简单，因为把对象生成放在了XML里定义。

- 在我们DNA框架中，元数据具体的就是那些service类，数据库逻辑表定义，界面等等。我们在使用DNA框架进行开发的时候，会经常和一个xml打交道，即dna.xml，它就是负责收集各种元数据信息的xml文件。比如我们的service元数据，通过它来收集各个工程中的service等，然后通过反射的方式进行实例化，放在内存中，这样在DNA框架中就可以使用了。

一个典型的dna.xml示例

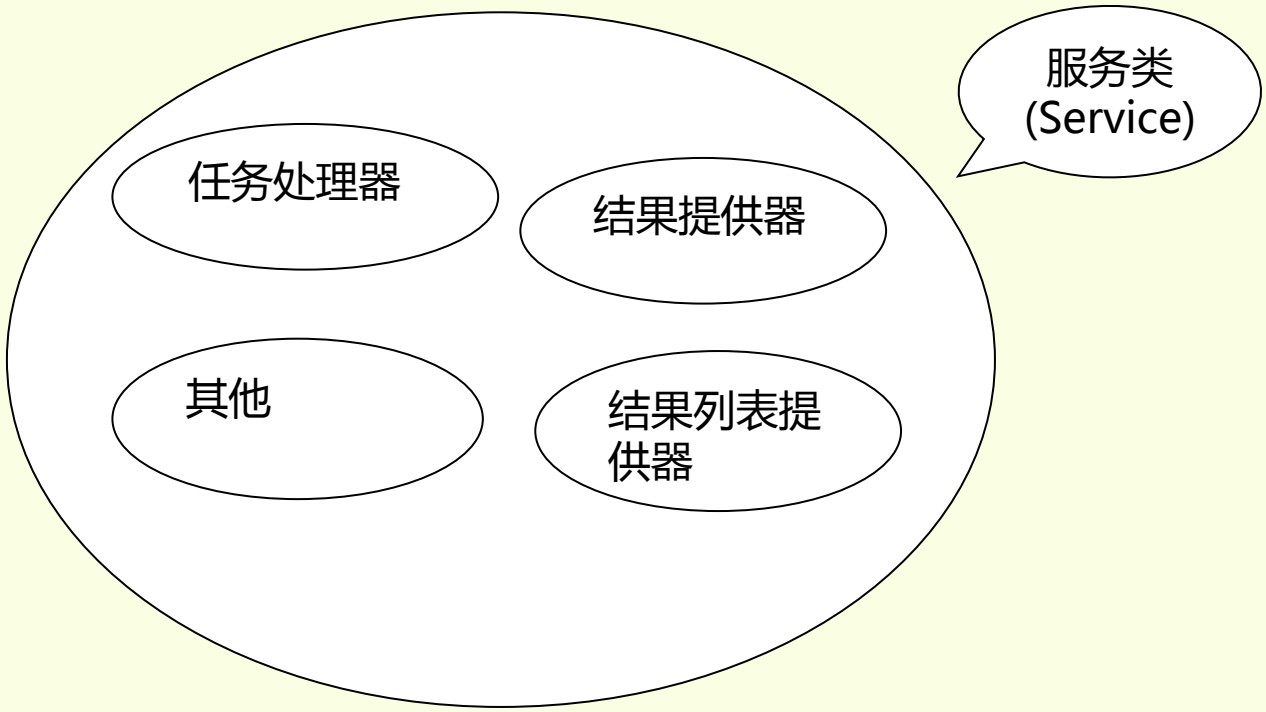
```
<?xml version="1.0" encoding="UTF-8"?>
<dna>
  <publish>
    <services>
      <service space="training" class="training.service.MyService" />
    </services>
    <orms>
      <!-- orm space="dna/core" class="com.jiuqi.dna.core.XXORM" -->
    </orms>
    <commands>
      <!-- command space="dna/core" class="com.jiuqi.dna.core.XXCommand" -->
    </commands>
    <querys>
      <!-- query space="dna/core" class="com.jiuqi.dna.core.XXQuery" -->
    </querys>
    <tables>
      <!--table space="dna/core" class="com.jiuqi.dna.core.impl.TD_BundleBin" /-->
    </tables>
    <ui-entrys>
      <uientry class="training.misc.TrainingEntry" name="e" />
    </ui-entrys>
    <pages>
      <!--page space="mydemo" name="testtemplate" class="com.jiuqi.dna.ui.template.launch.TemplatePage"
      /-->
    </pages>
    <ui-stylesheets>
      <!-- stylesheet name="stylesheet" title="stylesheet" path="com/jiuqi/dna/ui/demo/stylesheet.css"
      -->
    </ui-stylesheets>
    <portlets>
      <!-- portlet name="portlet" class="com.jiuqi.dna.ui.XXPortlet" -->
    </portlets>
    <templates />
  </publish>
</dna>
```

- 一般情况下，我们可以把具体的功能分为两大类：处理和请求。



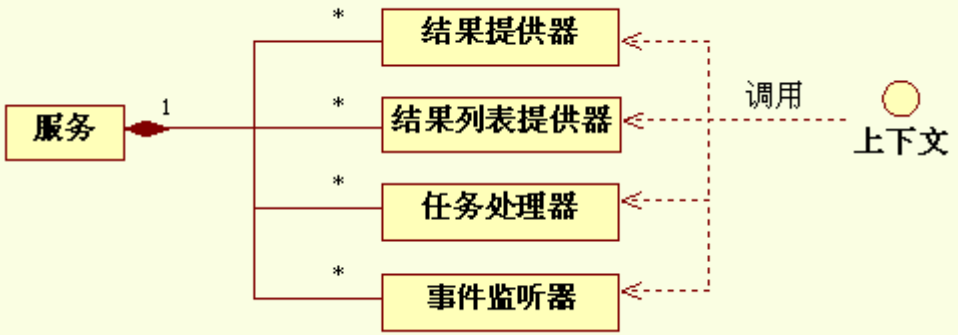
- 对于处理类的功能，一般包括：运算并添加、修改和删除等。需要用“任务处理器（TaskMethodhandler）”来实现。
- 对于请求类的功能，需要用“结果提供器（ResultProvider）”或者“结果列表提供器（ResultListProvider）”来实现。一个结果（列表）提供器对应于某一组特定的查询条件，来完成相应的查询。

- 在DNA中，业务逻辑的代码在这些具体的任务处理器和结果提供器中。在实现的时候，还可以通过框架来调用其他的任务处理器或结果提供器。
- 这些实现必须在服务类（service）中来编写，也就是说，服务类是业务逻辑实现的载体。

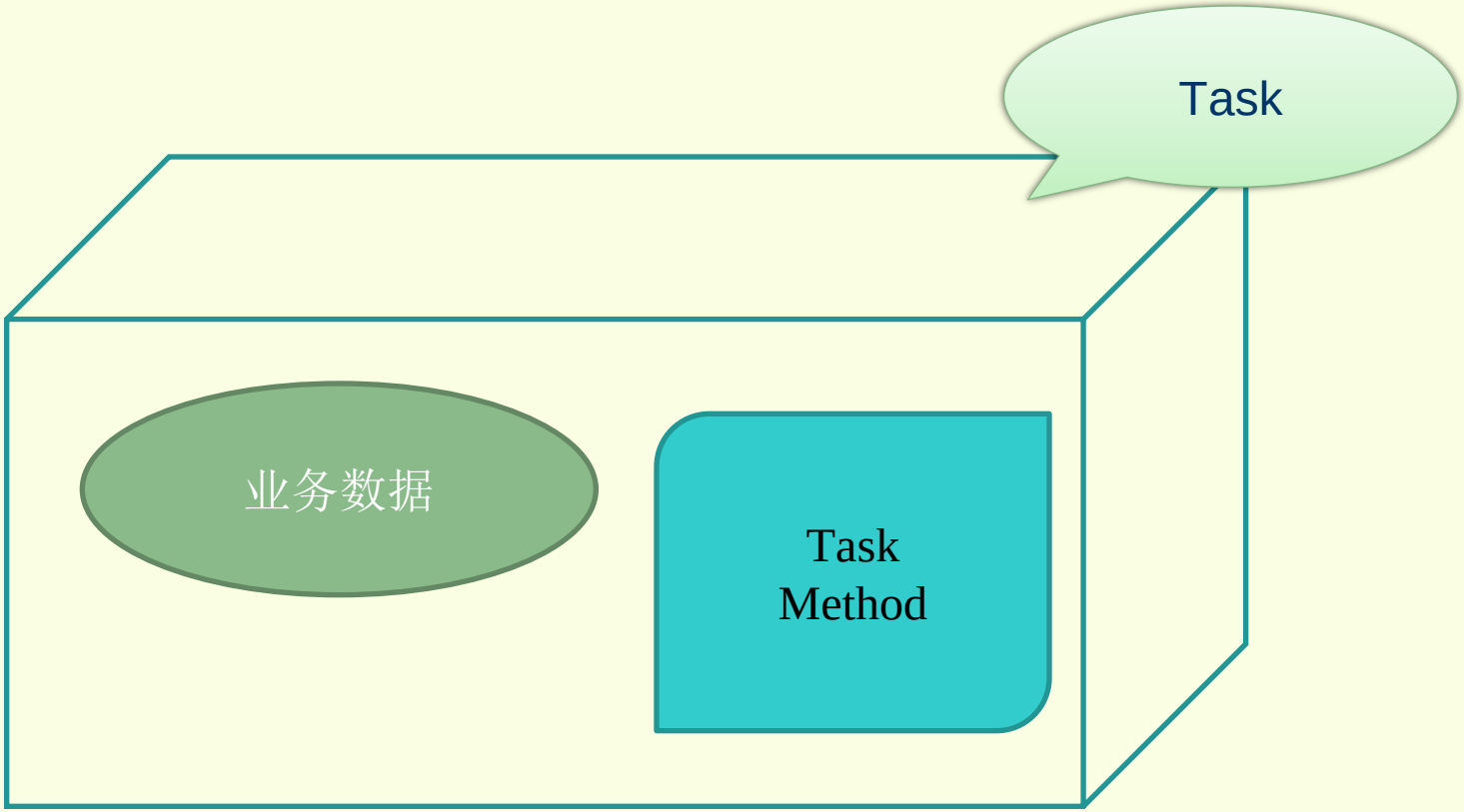


服务的开发方法

- 服务的开发很简单，基本上有下面的几步：
- 写一个类，继承自com.jiuqi.dna.core.service.Service，并在构造函数里面加入用到的元数据。
- 根据需要重写init方法
- 开发相应的provider
- 开发handler
- 开发treenodeproivder
- 在service工程的dna.xml里面注册写好的service类
- Treenodeproivder用的少，常用的都是provider和handler。



- 我们把完成某一个具体功能所需的数据定义为一个“任务（Task）”，一个任务处理器就对应于某一个特定的任务，来完成相应的功能。



- 从上面的阐述可以知道，提供器和处理器对执行功能所需的数据的这种抽象处理方式都是直接受到了框架对调用进行集中管理的影响。也正是这种抽象的处理方式，才使框架对调用进行集中管理得以实现。

- 这里的任务指的是对实现具体功能时所需的数据的一种类型定义。任务定义了其所能承载的数据的具体类型和数量。而依托与这些数据之上的逻辑处理含义则叫作任务方法。
- 很多情况下，一类数据只会对应一个逻辑含义（任务方法），这时就可以把任务定义为简单任务（一种特殊的任务，只有一个任务方法，且对应的任务处理器也是简单任务处理器）。因为简单任务只对应一个任务方法，所以也就没有必要强求这个任务的名称到底应该定义成什么样子了，因此简单任务在内部隐含了一个缺省的任务方法，不需要用户再自行定义。任务处理器的实现者也只需要针对此简单任务实现一个简单任务处理器（同样不需要再明确指定支持的任务方法）即可。

- 任务处理器是一种数据处理程序或者说过程，是对各种逻辑操作过程的统称。比如，数据的添加、修改和删除功能都可以实现为任务处理器。当然，也可用于查询数据，但因为框架为查询数据做了特性化支持，所以一般就不采用这个方式了。
- 和普通的依赖接口方法的方式相比，任务处理器把方法转变成类，方法的参数（数据）则封装为“任务”，这样就把接口的方法转变为了任务处理器。方法针对特定的方法名和参数来实现相应的处理过程，而任务处理器则针对任务（数据）来实现其处理过程。

- 很多情况下，多个处理过程可能需要的数据的类型都是相同的，只是处理过程的逻辑含义不同。在普通的接口编程中，可用方法名来作区分。而在任务处理器中，则采用一个“任务方法（枚举变量）”来区分，从而即可达到共用任务的目的。
- 任务处理器靠任务的类型及所支持的任务方法来唯一确定。一个任务处理器只能定义为处理某个类型的任务的处理器，处理过程根据其所声明的任务方法实现具体的逻辑处理过程。一个任务处理器可以声明只支持一个任务方法，对应的只需要实现一个针对些任务方法的逻辑处理过程即可；还可以声明支持相同任务下的多个任务方法，并为每个支持的任务方法编写相应的处理过程。但框架不推荐后一种做法，绝大多数情况下，用户只应选择定义仅支持一个任务方法的任务处理器。

任务处理器开发方法

- 任务处理器（Handler）是处理一个任务时的具体实现，那么我们一般需要开发相应的handler类，继承子任务处理基类。

```
/**
                 * 任务处理基类。<br>
                 * 开发人员在Service的子类中实现任务处理类时需要实现一个无参数的构造函数，否则系统无法正确使用该
任务处理类
                 * 建议将公用代码部分写在Service中，而任务处理类只负责实现任务具体的方法处理<br>
                 * 强烈建议：除非十分必要否则最好不要为一个任务的所有处理方法只定义一个处理类。<br>
                 */
                protected abstract class TaskMethodHandler<TTask extends Task<TMethod>, TMethod extends
Enum<TMethod>>
如果是简单任务，则需要继承简单任务处理基类。
/**
                 * 简单任务处理器基类
                 */
                protected abstract class SimpleTaskMethodHandler<TTask extends SimpleTask>
```

- 结果提供器是一种单值查询提供器，主要用于实现对数据的查询功能。
- 结果提供器的定义主要有两个要点，一个是待查数据（返回值）的类型，另一个是查询凭据类型的排列。这两点结合起来用于唯一确定一个结果提供器。所以在设计时不应出现这两点都相同的结果提供器。
- 调用结果提供器的方法
  - `Context.get(façade.class,key);`
  - `Context.find(façade.class,key);`

结果列表提供器

- 结果列表提供器是一种多值查询提供器，同结果提供器类似，主要用于实现对数据的查询功能，且以列表（List）的形式返回结果。
- 同结果提供器类似，结果列表提供器的定义也是那两个要点。且也是那两点唯一确定一个结果列表提供器。但结果提供器与结果列表提供器之间不会出现这种冲突。
- 调用结果列表提供器的方法
  - `Context.getList(façade.class,key);`

- 我们开发Provider的时候，只需要开发一个类，继承自Service里面的Provider抽象类，实现里面的Provide方法即可。需要根据实际情况继承相应的抽象类：

```
/**
 * 单例结果提供器
 */
protected abstract class ResultProvider<TResult>
/**
 * 单键结果提供器
 */
protected abstract class OneKeyResultProvider<TResult, TKey>
/**
 * 双键结果提供器
 */
protected abstract class TwoKeyResultProvider<TResult, TKey1, TKey2>
```

提供器的类型

```
/**
 * 三键结果提供器
 */
protected abstract class ThreeKeyResultProvider<TResult, TKey1, TKey2, TKey3>
/**
 * 结果列表提供器，用于返回一个结果集
 */
protected abstract class ResultListProvider<TResult>
/**
 * 单键结果列表提供器，用于根据指定的条件返回一个结果集
 */
protected abstract class OneKeyResultListProvider<TResult, TKey>
/**
 * 双键结果列表提供器，用于根据指定的条件返回一个结果集
 */
protected abstract class TwoKeyResultListProvider<TResult, TKey1, TKey2>
/**
 * 三键结果列表提供器，用于根据指定的条件返回一个结果集
 */
protected abstract class ThreeKeyResultListProvider<TResult, TKey1, TKey2, TKey3>
```

谢谢