## 概述

MMSE

## 系统详述（概念与功能，使用，原理，扩展）

1. 从service说起
2. 概念与功能

* Service类是MMSE对外提供服务的基本单元，将一个普通类添加Service注解（com.mm.engine.framework.control.annotation. Service）即可将其声明为Service。Service注解的结构为：   
  **public** @**interface** Service {String init() **default ""**;// Service被加载后系统执行的初始化方法  
   **int** initPriority() **default** 3; *// 初始化的优先级,越小越先初始化,* String destroy() **default ""**;// 系统关闭时的销毁方法**boolean** singleService() **default false**; // 声明为单实例运行   
  }
* 其中，singleService是指，在整个集群服务器中是否只运行一个服务实例，如果是，其它所有的服务器将通过远程调用访问该实例。
* Service类将在系统启动的时候实例化，然后可通过BeanHelper.getServiceBean(XXX.class)获取其实例。
* Service类在实例化时将进行针对Service的处理，包括初始化，销毁，包括Aop和Ioc，其它组件访问的生成等。

1. 扩展

* 可通过ServiceHelper.getMethodsByAnnotation(Class<? extends Annotation> cls)方法获取所有service中添加某种注解的方法
* FrameBean是一种特殊的Service，FrameBean是指系统中一些功能的默认的实现方式可以由用户重新定义，并通过在配置文件中配置来被系统加载并使用，具体看FrameBean。

1. Aop与Ioc
2. Aop的概念与功能

* Aop是指动态的对某个方法的进行改造，在其运行前后添加一定的功能逻辑，在MMSE中，Aop主要是针对Service中的方法。一个Aop功能至少包括切面和切点两个部分构成。
* 切面是指添加Aop的方法，其运行前后要进行的处理逻辑，切点指给那些方法添加切面。
* 切面顺序：定义在切面上面，指对所有切面的执行优先级进行排序。

1. Aop的使用

* 切面的实现：通过对一个类继承抽象类AspectProxy，并声明@Aspect注解来将其定义为切面类。其中，实现两个方法：  
  **public void** before(Object object,Class<?> cls, Method method, Object[] params);  
  **public void** after(Object object,Class<?> cls, Method method, Object[] params, Object result);  
  五个参数分别为：  
  object：对应的Service的实例；cls：对应的Service类（注意，object一般会是Service的代理类的实例，即object.getClass获取的不是cls）；method：所执行的方法；params：方法的参数；result：方法的返回值。
* 切面与切点的关系有三种对应方式，分别对应@Aspect中的三个参数：  
  **public** @**interface** Aspect { String[] mark() **default** {}; Class<? **extends** Annotation>[] annotation() **default** {}; String[] pkg() **default** {};}  
  mark对应后面切点注解AspectMark中的mark，当添加AspectMark的方法，其mark数组中存在对应Aspect中的mark时，将执行该切面，添加该AspectMark的类，其所有的方法都将执行该切面；annotation表示所有添加该注解的方法都将执行该切面，添加该注解的类的所有的方法都将执行该切面；pkg表示，在pkg对应的包下的所有类的所有方法都将执行该切面。
* 注解@AspectMark提供一个参数可用来通过mark数组定义切点：  
  **public** @**interface** AspectMark {  
   String[] mark();  
  }
* 通过在切面类上面添加注解@AspectOrder类声明该切面的优先级，AspectOrder提供一个参数value来定义切面执行优先级：  
  **public** @**interface** AspectOrder {  
   **int** value();  
  }  
  其中value值越小，越优先执行

1. Aop扩展

* BeanHelp提供一个实例化方法：  
  **public static** <T> T newAopInstance(Class<T> cls);  
  通过该实例化方法实例化一个对象，其中的方法将拥有Aop功能。注意：该类必须要在配置文件定义的appPackage中。

1. Ioc功能

* 所有Service，FrameService和Entrance中定义的Service和FrameService的引用，都会由系统自动为其赋值。注意：变量可以定义为私有

1. 五大基本控制组件
2. Request

* request是接收外部访问的基本组件，每个request拥有一个独有的操作码（opcode），由request入口进入的网络访问将根据opcode定位到request，由request对请求进行执行，并返回请求结果。
* 通过对Service中的方法添加@Request注解来生命一个方法为request组件： **public** @**interface** Request { **int** opcode();  
  }
* 对应的方法参数类型和返回类型是确定的：  
  **public** RetPacket xxx(Object clientData, Session session);  
  其中clientData为前端传入的数据，session为前端在后端保存的上下文组件（参考Session和Account）。RetPacket为返回给前端的数据：  
  **public interface** RetPacket {  
   **public int** getOpcode(); // 返回的操作码  
   **public boolean** keepSession(); // 是否保存session  
   **public** Object getRetData(); // 返回给客户端的数据  
  }

1. Event

* Event是事件服务，包括四个部分：事件类型，事件数据，时间抛出，事件接收者。
* 通过给Service中的方法添加注解@EventListener来定义一个监听组件：  
  **public** @**interface** EventListener{  
  **int** event();  
  }  
  参数event代表要监听的事件类型。
* 对应的方法参数类型和返回类型是确定的:  
  **public void** xxx(EventData eventData);  
  其中EventData的结构如下：  
  **public class** EventData {  
   **private int event**; // 事件类型  
   **private** Object **data**; // 事件数据  
  }

|  |  |
| --- | --- |
| 同步执行事件&不广播 | 异步执行事件&不广播 |
| 同步执行事件&广播 | 异步执行事件&广播 |

* 事件抛出：通过EventService的fireEvent方法抛出事件，包括四种抛出方式：  
  同步执行指：在时间抛出的线程中执行事件，完成后返回  
  广播是指：该事件不仅在本服务器中执行，也在集群中其它服务器中执行

1. NetEvent与RPC

* NetEvent是RPC的实现层，实际使用主要以RPC为主即可
* 网络事件（NetEvent）是对Event的一种扩展，指对集群中其它服务器发出事件。NetEvent的网络入口在配置文件中的入口配置entrance.netEvent来配置，默认使用NetEventNettyEntrance
* 给Service中的方法添加NetEvenListener注解使其成为NetEvent的监听者，每个服务器对每种网络事件只能有一个监听者：  
  **public** @**interface** NetEventListener {  
  **int** netEvent();  
  }  
  netEvent对应监听网络事件的类型
* 对应的方法参数类型和返回类型是确定的:  
  **public** NetEventData testNetEvent1(NetEventData netEventData);  
  其中NetEventData和EventData的定义结构一样：  
  **public class** NetEventData {  
   **private int netEvent**; // 事件类型  
   **private** Object **param**; // 事件数据  
  }

|  |  |
| --- | --- |
| 同步抛出事件&广播 | 异步抛出事件&广播 |
| 同步抛出事件&发给mainServer | 异步抛出事件&发给mainServer |
| 同步抛出事件&发给asyncServer | 异步抛出事件&发给asyncServer |
| 同步抛出事件&发给任一server | 异步抛出事件&发给任一server |

* 事件抛出：NetEvent的抛出通过NetEventService来完成，主要包括如下几种方式：  
  其中，广播是指发送给所有的服务器，发给任一服务器需要提供相应服务器的地址或ServerClient对象；同步发送的发法将返回时间执行的结果：单服务器返回NetEventData，广播返回Map<String, NetEventData>
* 远程调用（RPC）是对NetEvent的一层封装，通过RemoteCallService进行调用，主要的调用方式包括：  
  **public** Object remoteCallSyn(String add,Class cls,String methodName,Object... params);  
  **public** Map<String,Object> broadcastRemoteCallSyn(Class cls, String methodName, Object... params);  
  其中：add是远程服务器的地址，cls是对应Service的类，methodName是对应的方法名，params是对应方法的参数。而对于单个服务器的调用，返回值为Object，广播调用返回一个Map，对应每个服务器返回的值。
* 广播方法是对远程调用的又一层封装，当对一个方法添加@BroadcastRPC注解时，对该方法的调用将广播到其它服务器并使之调用该方法:  
  **public** @**interface** BroadcastRPC {  
   **boolean** async() **default false**; // 是否异步调用  
  }  
  其中，参数async决定远程广播调用是采用同步的方式还是异步的方式。注意：被广播调用该方法的服务器不会再次广播该方法的调用。

1. Updatable

* 更新器，用于做定时更新的服务，可以根据设定的参数进行不同频率和方式的更新。更新器由系统加载并运行，系统期间不会停止。
* 通过给一个Service中的方法添加@Updatable注解来定义一个更新器组件：  
  **public** @**interface** Updatable {  
   **boolean** isAsynchronous() **default true**;**boolean** singleService() **default false**;**int** cycle() **default** -1;String cronExpression() **default ""**;  
  }  
  isAsynchronous：是否异步更新，当设置为false时，该组件进行同步更新，此时cycle和cronExpression两个参数不起作用，在某个固定周期（由系统参数syncUpdate.cycle配置）下，与其它同步更新组件同步运行，所以同步更新服务不应该处理任务量较大的服务。但同步更新不需要再开线程，更不需要cronExpression的低效的时间判断，进而适合做简单的快速更新。  
  singleService：和Serive的singleService一样，当设定为true时，该组件只在一个服务器上面运行。  
  cycle：更新周期，该组件更新的时间间隔，当cronExpression设置有效值后，该参数无效。  
  cronExpression：cronExpression表达式，用于设置更新周期，详见……
* 对应的方法和参数是固定的：  
  **public void** xxx(**int** interval);  
  其中interval是更新间隔，即上次更新到本次更新的时间。

1. Job

* Job是一个定时任务，即在某段时间之后或某个时间点执行一个服务。
* Job的构建和使用：构建一个Job对象，通过JobService的：  
  **public void** startJob(Job job);  
  **public void** deleteJob(String id);  
  来启动和删除Job，其中Job对象的结构为：  
  **public class** Job{  
   **private** String **id**; *// job唯一id* **private** Date **startDate**; *// 执行时间* **private boolean db**; *// 是否持久化* **private** String **method**;  
   **private** Class **serviceClass**;  
   **private** Object[] **para**;  
  }  
  id：集群服务器中唯一id，建议根据业务逻辑和对应的Service同一设定。  
  startDate：job执行时间，如果小于当前时间将立刻执行。  
  db：job是否持久化，即如果服务器重启，是否还要执行该job。  
  method：job执行的方法；serviceClass：job对应的service类；para：job方法接收的参数。注意：如果db为true，para必须是可以序列化的：
* Job的存储：Job的存储方案类的定义需要继承JobStorage接口，并在配置文件中定义jobStorage。系统默认实现了DefaultJobStorage类，并将其存储在数据库中，如果使用它，要求数据库中定义相应的表：  
  **DROP TABLE** IF **EXISTS** `job`;  
  **CREATE TABLE** `job` (  
   **`id` varchar**(255) **NOT NULL**,  
   **`startDate` timestamp NULL DEFAULT NULL**,  
   **`db` int**(11) **DEFAULT NULL**,  
   **`method` varchar**(255) **DEFAULT NULL**,  
   **`serviceClass` varchar**(255) **DEFAULT NULL**,  
   **`params`** blob,  
   **PRIMARY KEY** (**`id`**)  
  ) ENGINE=InnoDB **DEFAULT** CHARSET=utf8;
* 注意：jobStorage属于FrameService，所以支持相应的Aop和Ioc功能

1. 用room做高实时性服务
2. 缓存与异步持久化
3. 服务层事务
4. Session与Account
5. 网络通信
6. 异常与监控
7. FrameBean
8. 系统工具：系统变量，Gm指令，IdService，国际化，Utils，测试

## 集群

## 系统与周边

1. 系统配置文件
2. 策划配数
3. 前后端协议
4. 部署
5. 运营

## 基本使用流程：定义表

## 一个小游戏案例