跨语言服务链监控与大数据收集、统计

-----基于CAT

作者：曹号 [649947921@qq.com](mailto:649947921@qq.com)

博客： https://blog.csdn.net/caohao0591

版权声明，不得用于商业用途，如有需要，请联系作者。

目录

[跨语言服务链监控与大数据收集、统计 1](#_Toc519004560)

[-----基于CAT 1](#_Toc519004561)

[第一章 Cat介绍与部署 3](#_Toc519004562)

[介绍 3](#_Toc519004563)

[背景介绍 4](#_Toc519004564)

[Cat系统的特性 4](#_Toc519004565)

[消息树 5](#_Toc519004566)

[CAT服务端部署 7](#_Toc519004567)

[CAT客户端Demo 13](#_Toc519004568)

[第二章 CAT服务端初始化 21](#_Toc519004569)

[Cat模块 21](#_Toc519004570)

[Cat-servlet初始化 23](#_Toc519004571)

[plexus - IOC容器 24](#_Toc519004572)

[模块的加载 - 模型模式 25](#_Toc519004573)

[cat-home的setup 27](#_Toc519004574)

[TcpSocketReceiver--- netty reactor 模式的应用 28](#_Toc519004575)

[消息的解码 31](#_Toc519004576)

[第三章 CAT客户端实现 33](#_Toc519004577)

[cat客户端部分核心类 33](#_Toc519004578)

[消息的组织 - 消息树 34](#_Toc519004579)

[客户端的初始化 36](#_Toc519004580)

[消息生产 -- 入栈 41](#_Toc519004581)

[Context 线程本地变量 42](#_Toc519004582)

[Transaction事务的开启 45](#_Toc519004583)

[其他类型消息组合 46](#_Toc519004584)

[消息的完成-出栈 50](#_Toc519004585)

[消息的发送-队列化 52](#_Toc519004586)

[消息的序列化 53](#_Toc519004587)

[MessageId的设计 56](#_Toc519004588)

[第四章 服务端消息分发 56](#_Toc519004589)

[分发架构 56](#_Toc519004590)

[分析管理器的初始化 57](#_Toc519004591)

[消费者与周期管理器的初始化 59](#_Toc519004592)

[什么是周期？ 62](#_Toc519004593)

[周期任务-任务队列 65](#_Toc519004594)

[消息分发 67](#_Toc519004595)

[周期策略 68](#_Toc519004596)

[第五章 配置与数据库操作 70](#_Toc519004597)

[CAT配置 70](#_Toc519004598)

[代码自动生成 75](#_Toc519004599)

[数据库操作 78](#_Toc519004600)

[数据库连接管理 82](#_Toc519004601)

[第六章 消息分析器与报表(一) 86](#_Toc519004602)

[消息分析器的构建 86](#_Toc519004603)

[TopAnalyzer 90](#_Toc519004604)

[EventAnalyzer - 事件发生次数分析 92](#_Toc519004605)

[MetricAnalyzer - 业务分析 94](#_Toc519004606)

[ProblemAnalyzer -异常分析 102](#_Toc519004607)

[TransactionAnalyzer - 事务分析 108](#_Toc519004608)

[第七章 消息分析器与报表(二) 111](#_Toc519004609)

[CrossAnalyzer-调用链分析 111](#_Toc519004610)

[StorageAnalyzer  --数据库/缓存分析 120](#_Toc519004611)

[StateAnalyzer 125](#_Toc519004612)

[HeartbeatAnalyzer 126](#_Toc519004613)

[DumpAnalyzer -- 原始消息LogView存储 134](#_Toc519004614)

[自定义分析器与报表 141](#_Toc519004615)

[第八章 数据持久化 141](#_Toc519004616)

[周期结束 141](#_Toc519004617)

[分析器的结束 -- 报表持久化 144](#_Toc519004618)

[报表预处理 147](#_Toc519004619)

[报表的文件存储 -- 重入锁 150](#_Toc519004620)

[报表的数据库存储 153](#_Toc519004621)

[定时任务 155](#_Toc519004622)

[第九章 CAT管理平台MVC框架 163](#_Toc519004623)

[Servlet容器与请求生命周期 163](#_Toc519004624)

[页面路由初始化 166](#_Toc519004625)

[请求处理流程 175](#_Toc519004626)

[第十章 CAT客户端与其它JAVA框架的集成 177](#_Toc519004627)

[与Spring MVC集成 177](#_Toc519004628)

[与Spring Boot 集成 177](#_Toc519004629)

[与Spring Cloud 集成 177](#_Toc519004630)

[与dubbo集成 177](#_Toc519004631)

[与MyBatis集成 177](#_Toc519004632)

[与Log4j集成 177](#_Toc519004633)

[第十一章 对其它语言的支持 178](#_Toc519004634)

[PHP语言 178](#_Toc519004635)

[C++语言 178](#_Toc519004636)

[LUA语言 178](#_Toc519004637)

[Go语言 178](#_Toc519004638)

[Node.js语言 178](#_Toc519004639)

[Android埋点 178](#_Toc519004640)

[Object C -- IOS 埋点 178](#_Toc519004641)

[第十二章 大数据收集与实时计算 178](#_Toc519004642)

[hadoop模块 178](#_Toc519004643)

[spark实时计算模块 178](#_Toc519004644)

[第十三章 报警与通知机制 178](#_Toc519004645)

[短信通知 178](#_Toc519004646)

[邮件通知 178](#_Toc519004647)

# 第一章 Cat介绍与部署

## 介绍

[大众点评CAT](https://www.oschina.net/p/cat-dianping" \t "_blank)系统原型和理念来源于eBay的CAL的系统，CAT系统第一代设计者吴其敏在eBay工作长达十几年，对CAL系统有深刻的理解。CAT不仅增强了CAL系统核心模型，还添加了更丰富的报表。自2014年开源以来，CAT在携程、陆金所、猎聘网、找钢网等多家互联网公司生产环境应用。

    CAT是一个实时和接近全量的监控系统，它侧重于对Java应用的监控，除了与点评RPC组件融合的很好之外，他将会能与Spring、MyBatis、Dubbo 等框架以及Log4j 等结合，不久将会支持PHP、C++、Go等多语言应用，基本接入了美团点评上海侧所有核心应用。目前在中间件（MVC、RPC、数据库、缓存等）框架中得到广泛应用，为美团点评各业务线提供系统的性能指标、健康状况、监控告警等，在微服务监控领域也是非常有用的一套组件。

    在详细了解CAT的整体设计细节之后，我们可以在CAT基础之上轻松扩展我们自己的监控和数据收集模块。

    CAT项目的开源地址： [https://github.com/dianping/cat](https://github.com/dianping/cat" \t "_blank)

## ****背景介绍****

    CAT整个产品研发是从2011年底开始的，当时正是大众点评App Net迁移Java的核心起步阶段。当初大众点评App已经有核心的基础中间件、RPC组件Pigeon、统一配置组件lion。整体Java迁移已经在服务化的路上。随着服务化的深入，整体Java在线上部署规模逐渐变多，同时，暴露的问题也越来越多。典型的问题有：

* 大量报错，特别是核心服务，需要花很久时间才能定位。
* 异常日志都需要线上权限登陆线上机器排查，排错时间长。
* 有些简单的错误定位都非常困难（一次将线上的库配置到了Beta，花了整个通宵排错）。
* 很多不了了之的问题怀疑是网络问题（从现在看，内网真的很少出问题）。

    虽然那时候也有一些简单的监控工具（比如Zabbix，自己研发的Hawk系统等），可能单个工具在某方面的功能还不错，但整体服务化水平参差不齐、扩展能力相对较弱，监控工具间不能互通互联，使得查找问题根源基本都需要在多个系统之间切换，有时候真的是靠“人品”才能找出根源。适逢吴其敏从eBay加入大众点评成为首席架构师，eBay的CAL系统在内部非常成功，就在这样天时地利与人和的情况下，我们开始研发了大众点评App第一代监控系统——CAT。

## Cat系统的特性

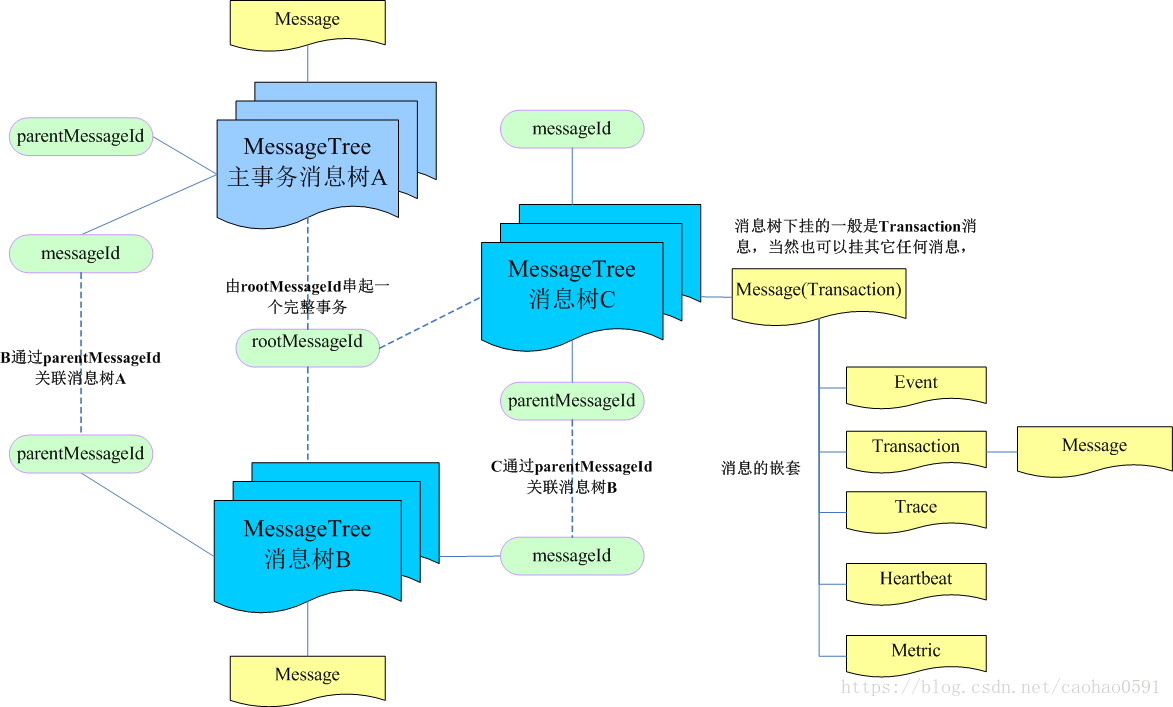
* 实时处理：信息的价值会随时间锐减，尤其是事故处理过程中。
* 全量数据：最开始的设计目标就是全量采集，全量的好处有很多。
* 高可用：所有应用都倒下了，需要监控还站着，并告诉工程师发生了什么，做到故障还原和问题定位。
* 故障容忍：CAT本身故障不应该影响业务正常运转，CAT挂了，应用不该受影响，只是监控能力暂时减弱。
* 高吞吐：要想还原真相，需要全方位地监控和度量，必须要有超强的处理吞吐能力。
* 可扩展：支持分布式、跨IDC部署，横向扩展的监控系统。
* 不保证可靠：允许消息丢失，这是一个很重要的trade-off，目前CAT服务端可以做到4个9的可靠性，可靠系统和不可靠性系统的设计差别非常大。

CAT支持的监控消息类型包括：

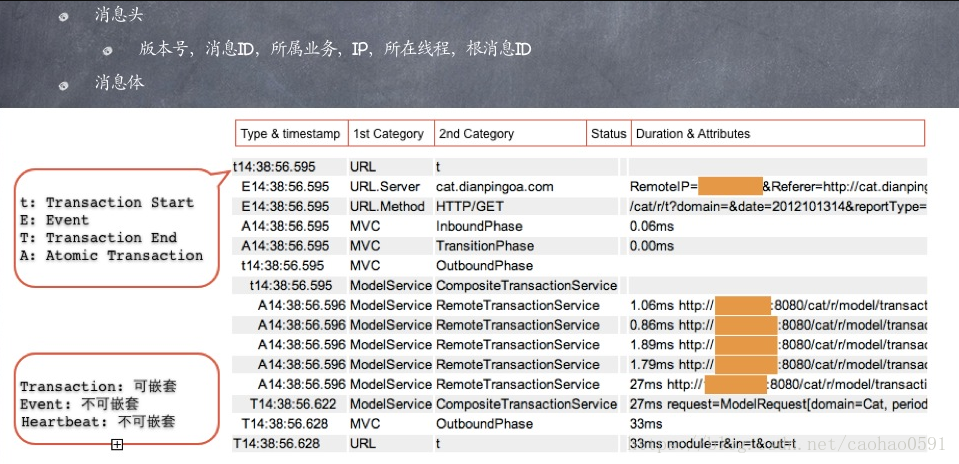
* **Transaction** 适合记录跨越系统边界的程序访问行为,比如远程调用，数据库调用，也适合执行时间较长的业务逻辑监控，Transaction用来记录一段代码的执行时间和次数。
* **Event** 用来记录一件事发生的次数，比如记录系统异常，它和transaction相比缺少了时间的统计，开销比transaction要小。
* **Heartbeat** 表示程序内定期产生的统计信息, 如CPU%, MEM%, 连接池状态, 系统负载等。
* **Metric** 用于记录业务指标、指标可能包含对一个指标记录次数、记录平均值、记录总和，业务指标最低统计粒度为1分钟。
* **Trace** 用于记录基本的trace信息，类似于log4j的info信息，这些信息仅用于查看一些相关信息

## 消息树

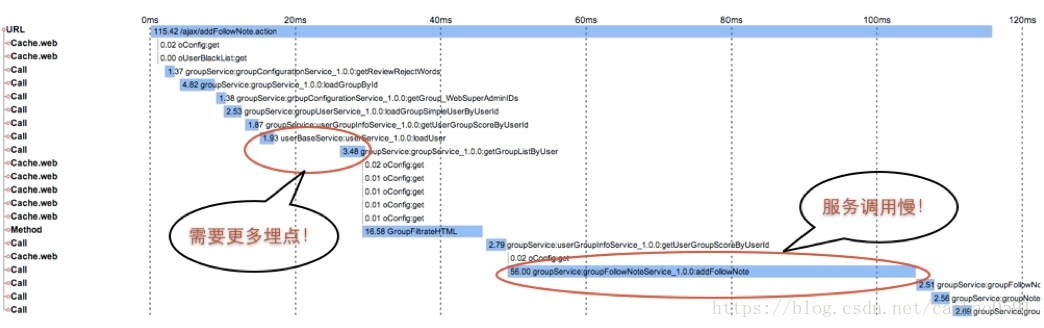
    CAT监控系统将每次URL、Service的请求内部执行情况都封装为一个完整的消息树、消息树可能包括Transaction、Event、Heartbeat、Metric和Trace信息，各个消息树之间，通过 rootMessageId以及parentMessageId串联起来，形成整个调用链条。



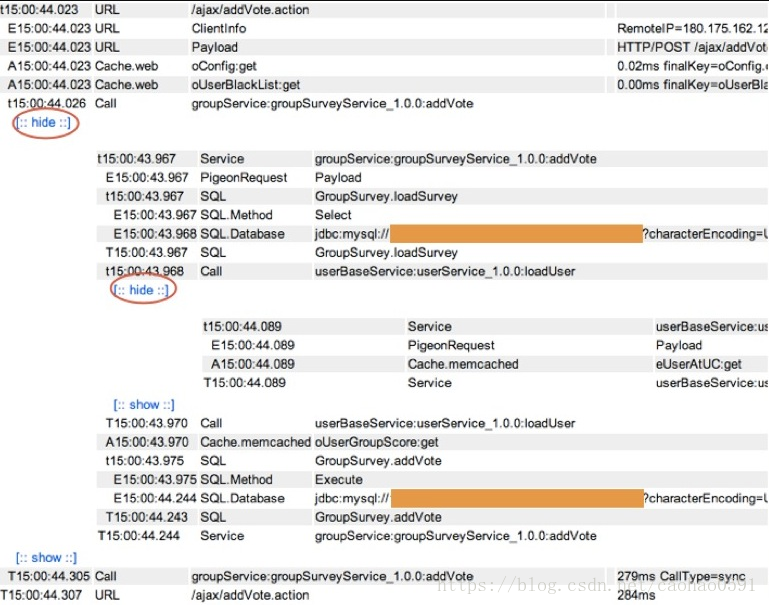
**完整的消息树：**



**可视化的消息树：**



**分布式消息树【一台机器调用另外一台机器】：**



## CAT服务端部署

**CAT安装环境：**

Linux 2.6以及之上（2.6内核才可以支持epoll），线上服务端部署请使用Linux环境，Mac以及Windows环境可以作为开发环境，美团点评内部CentOS 6.5

Java 6，7，8，服务端推荐是用jdk7的版本，客户端jdk6、7、8都支持

Maven 3.3.3

MySQL 5.6，5.7，更高版本MySQL都不建议使用，不清楚兼容性

J2EE容器建议使用tomcat，建议版本7.0.70

Hadoop环境可选，一般建议规模较小的公司直接使用磁盘模式，可以申请CAT服务端，500GB磁盘或者更大磁盘，这个磁盘挂载在/data/目录上

**目前我司线上环境：**

    Distributor ID: CentOS  
    Description: CentOS release 6.5 (Final)  
    Release: 6.5  
    Codename: Final  
    Server version: Apache Tomcat/8.0.30  
    Server built:   Dec 1 2015 22:30:46 UTC  
    Server number:  8.0.30.0  
    OS Name:        Linux  
    OS Version:     2.6.32-431.el6.x86\_64  
    Architecture:   amd64  
    JVM Version:    1.8.0\_111-b14  
    JVM Vendor:     Oracle Corporation  
    Maven 3.3.3  
    Mysql 5.6  
    Tomcat  7.0.70  建议使用此版本

我的开发环境：

    操作系统： Windows 7  
    IDE： Intelij IDEA  
    JDK版本：1.8  
    Mysql： 5.6  
    Maven： 3.3.3  
    Server version：Apache Tomcat/8.0.30

**安装CAT集群大致步骤**

    初始化Mysql数据库，一套CAT集群部署一个数据库，初始化脚本在script下的Cat.sql  
    准备三台CAT服务器，IP比如为10.1.1.1，10.1.1.2，10.1.1.3，下面的例子会以这个IP为例子  
    初始化/data/目录，配置几个配置文件/data/appdatas/cat/\*.xml 几个配置文件，具体下面有详细说明  
    打包cat.war 放入tomcat容器

    修改一个路由配置，重启tomcat

**Tomcat启动参数调整，修改 catalina.sh文件【服务端】**



程序对于/data/目录具体读写权限【包括客户端&服务端】

注意无论是CAT客户端和服务端都要求/data/目录能进行读写操作，如果/data/目录不能写，建议使用linux的软链接链接到一个固定可写的目录，软链接的基本命令请自行搜索google

此目录会存一些CAT必要的配置文件，运行时候的缓存文件，建议不要修改，如果想改，请自行研究好源码里面的东西，在酌情修改，此目录不支持进行配置化

mkdir /data

chmod 777 /data/ -R

如果是Windows开发环境则是对程序运行盘下的/data/appdatas/cat和/data/applogs/cat有读写权限,如果cat服务运行在e盘的tomcat中，则需要对e:/data/appdatas/cat和e:/data/applogs/cat有读写权限

如果windows实在不知道哪个盘，就所有盘都建好，最后看哪个盘多文件，就知道哪个了

配置/data/appdatas/cat/client.xml【包括客户端&服务端】

此配置文件的作用是所有的客户端都需要一个地址指向CAT的服务端，比如CAT服务端有三个IP，10.1.1.1，10.1.1.2，10.1.1.3，2280是默认的CAT服务端接受数据的端口，不允许修改，http-port是Tomcat启动的端口，默认是8080，建议使用默认端口。

此文件可以通过运维统一进行部署和维护，比如使用puppert等运维工具。

不同环境这份文件不一样，比如区分prod环境以及test环境，在美团点评内部一共是2套环境的CAT，一份是生产环境，一份是测试环境



安装CAT的数据库

数据库的脚本文件 script/Cat.sql

MySQL的一个系统参数：max\_allowed\_packet，其默认值为1048576(1M)，修改为1000M，修改完需要重启mysql

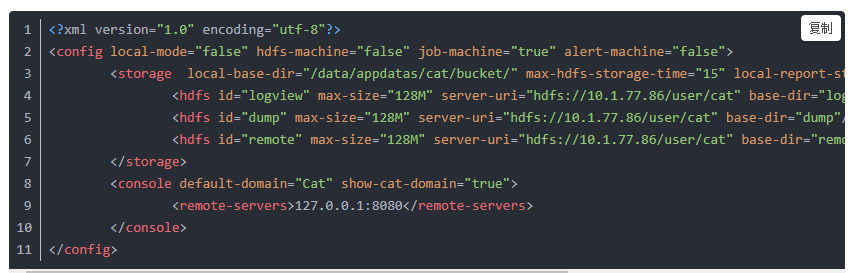
注意：一套独立的CAT集群只需要一个数据库（之前碰到过个别同学在每台cat的服务端节点都安装了一个数据库）

配置/data/appdatas/cat/datasources.xml【服务端配置】

需要每台CAT集群10.1.1.1，10.1.1.2，10.1.1.3都进行部署

注意：此xml仅仅为模板，请根据自己实际的情况替换jdbc.url,jdbc.user,jdbc.password的实际值。 app数据库和cat数据配置为一样，app库不起作用，为了运行时候代码不报错。

  
**配置/data/appdatas/cat/server.xml【服务端配置】**  
    需要每台CAT集群10.1.1.1，10.1.1.2，10.1.1.3都进行部署  
    CAT节点一共有四个职责  
  
    控制台 - 提供给业务人员进行数据查看【默认所有的cat节点都可以作为控制台，不可配置】  
    消费机 - 实时接收业务数据，实时处理，提供实时分析报表【默认所有的cat节点都可以作为消费机，不可配置】  
    告警端 - 启动告警线程，进行规则匹配，发送告警（目前仅支持单点部署）【可以配置】  
    任务机 - 做一些离线的任务，合并天、周、月等报表 【可以配置】  
    线上做多集群部署，比如说10.1.1.1，10.1.1.2，10.1.1.3这三台机器  
  
  
    建议选取一台10.1.1.1 负责角色有控制台、告警端、任务机，建议配置域名访问CAT，就配置一台机器10.1.1.1一台机器挂在域名下面  
    10.1.1.2，10.1.1.3 负责消费机处理，这样能做到有效隔离，任务机、告警等问题不影响实时数据处理  
    默认script下的server.xml为



配置说明：

local-mode : 建议在开发环境以及生产环境时，都设置为false

hdfs-machine : 定义是否启用HDFS存储方式，默认为 false

job-machine : 定义当前服务是否为报告工作机（开启生成汇总报告和统计报告的任务，只需要一台服务机开启此功能），默认为 false

alert-machine : 定义当前服务是否为报警机（开启各类报警监听，只需要一台服务机开启此功能），默认为 false；

storage : 定义数据存储配置信息

local-report-storage-time : 定义本地报告文件存放时长，单位为（天）

local-logivew-storage-time : 定义本地日志文件存放时长，单位为（天）

local-base-dir : 定义本地数据存储目录，建议直接使用/data/appdatas/cat/bucket目录

hdfs : 定义HDFS配置信息

server-uri : 定义HDFS服务地址

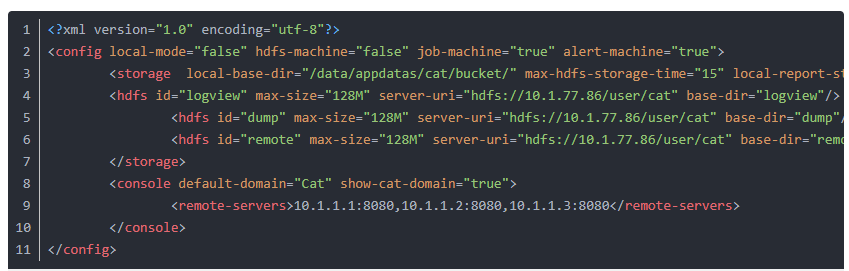
console : 定义服务控制台信息

remote-servers : 定义HTTP服务列表，（远程监听端同步更新服务端信息即取此值）

ldap : 定义LDAP配置信息（这个可以忽略）

ldapUrl : 定义LDAP服务地址（这个可以忽略）

按照如上的说明，10.1.1.1 机器/data/appdatas/cat/serverm.xml配置，注意hdfs配置就随便下了一个，请忽略



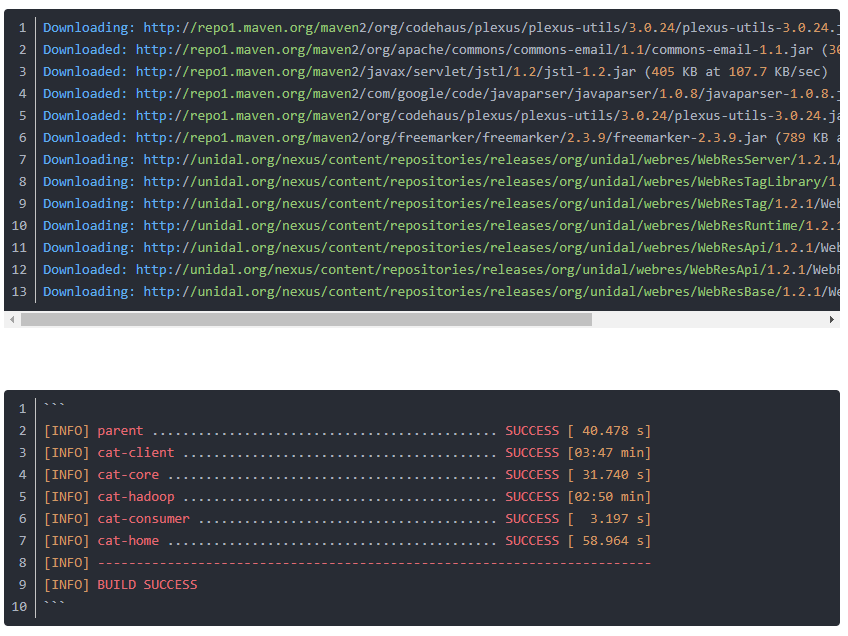
10.1.1.2，10.1.1.3 机器/data/appdatas/cat/serverm.xml配置如下，仅仅job-machine&alert-machine修改为false

war打包

    1、在cat的源码目录，执行mvn clean install -DskipTests  
    2、如果发现cat的war打包不通过，CAT所需要依赖jar都部署在 http://unidal.org/nexus/  
    3、可以配置这个公有云的仓库地址到本地的settings路径，理论上不需要配置即可，可以参考cat的pom.xml配置

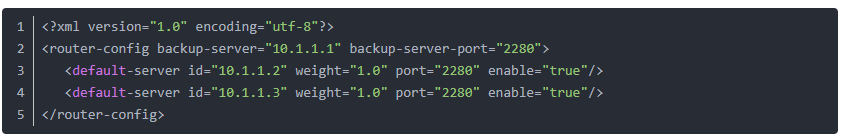
 4、如果自行打包仍然问题，请使用下面链接进行下载http://unidal.org/nexus/service/local/repositories/releases/content/com/dianping/cat/cat-home/2.0.0/cat-home-2.0.0.war

    5、官方的cat的master版本，重命名为cat.war进行部署，注意此war是用jdk8，服务端请使用jdk8版本  
    6、如下是个人本机电脑的测试，下载的jar来自于repo1.maven.org 以及 unidal.org

war部署

    1、将cat.war部署到10.1.1.1的tomcat的webapps下，启动tomcat，注意webapps下只允许放一个war，仅仅为cat.war  
    2、如果发现重启报错，里面有NPE等特殊情况，可以检查当前java进程，ps aux | grep java，可能存在之前的tomcat的进程没有关闭，又新启动了一个，导致出问题，建议kill -9 干掉所有的java进程  
    3、打开控制台的URL，http://10.1.1.1:8080/cat/s/config?op=routerConfigUpdate  
    4、注意10.1.1.1这个IP需要替换为自己实际的IP链接，修改路由配置只能修改一次即可

    5、修改路由配置为如下，当为如下配置时，10.1.1.1 正常不起消费数据的作用，仅当10.1.1.2以及10.1.1.3都挂掉才会进行实时流量消费

    6、重启10.1.1.1的机器的tomcat  
    7、将cat.war部署到10.1.1.2，10.1.1.3这两台机器中，启动tomcat  
    8、cat集群部署完毕，如果有问题，欢迎在微信群咨询，如果文档有误差，欢迎指正以及提交pullrequest

重启保证数据不丢

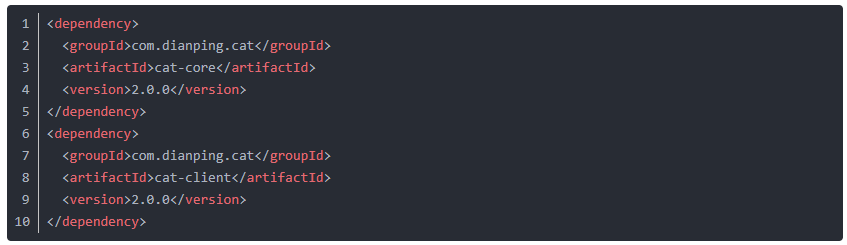
    请在tomcat重启之前调用当前tomcat的存储数据的链接 http://${ip}:8080/cat/r/home?op=checkpoint，重启之后数据会恢复。【注意重启时间在每小时的整点10-55分钟之间】  
 线上部署时候，建议把此链接调用存放于tomcat的stop脚本中，这样不需要每次手工调用

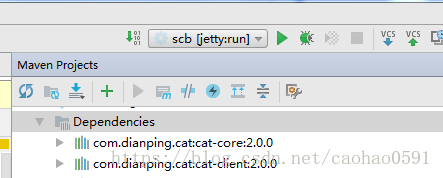
## CAT客户端Demo

**cat客户端的配置：**

    cat客户端也需要配置 /data 目录，程序对于/data/目录具体读写权限可以参考上一节，

    然后通过maven引入cat客户端包，在pom.xml 加入：

    引入之后：



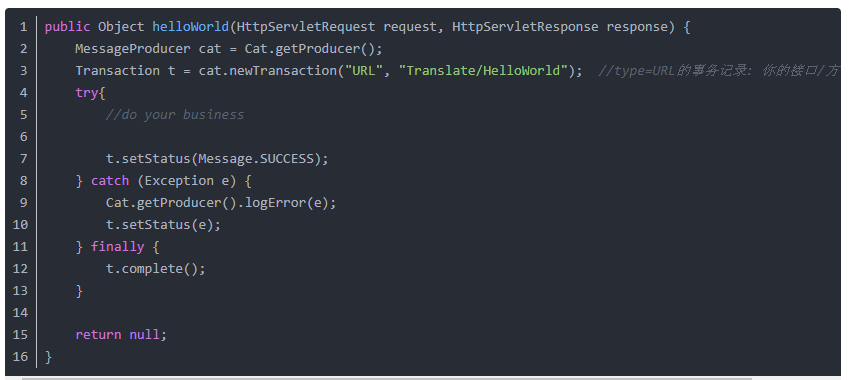
    以上配置会通过maven从网上引入 cat-core和cat-client 包，如果无法配置引入，也可以手动引入 cat-client.jar 包。

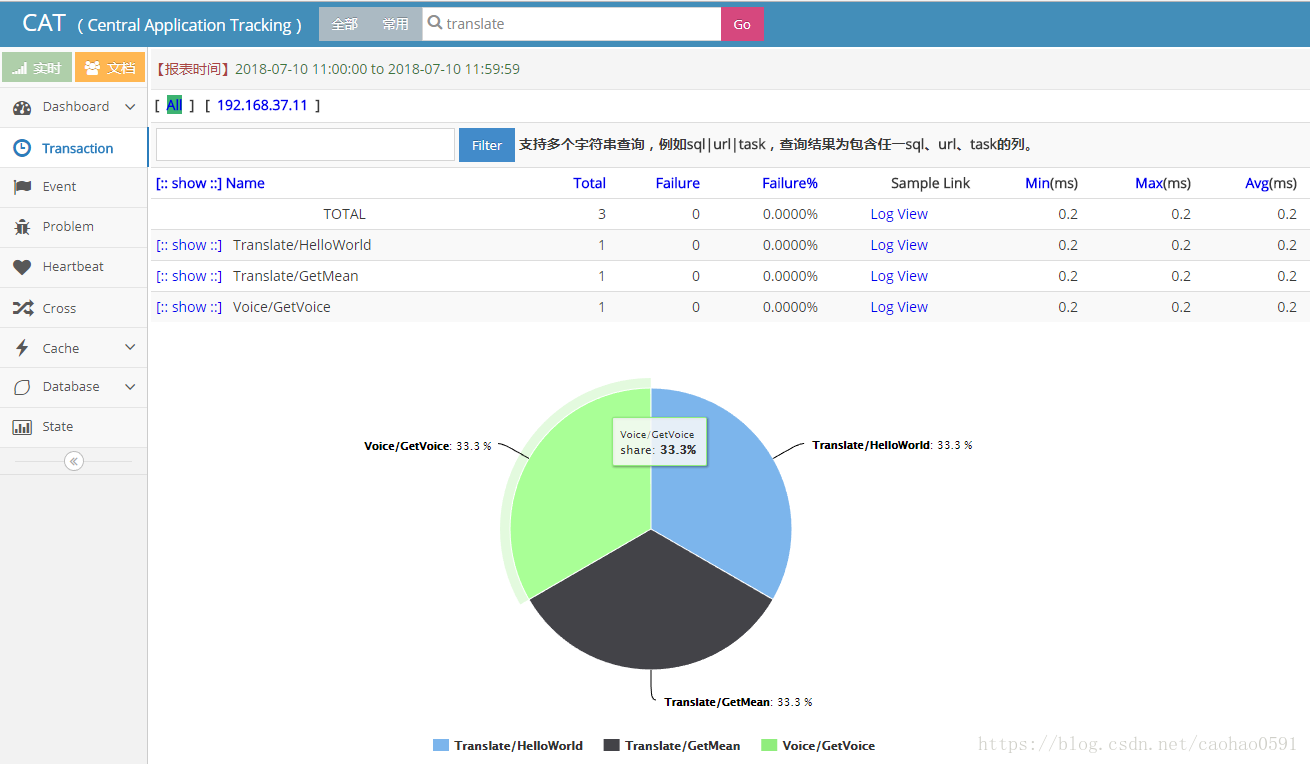
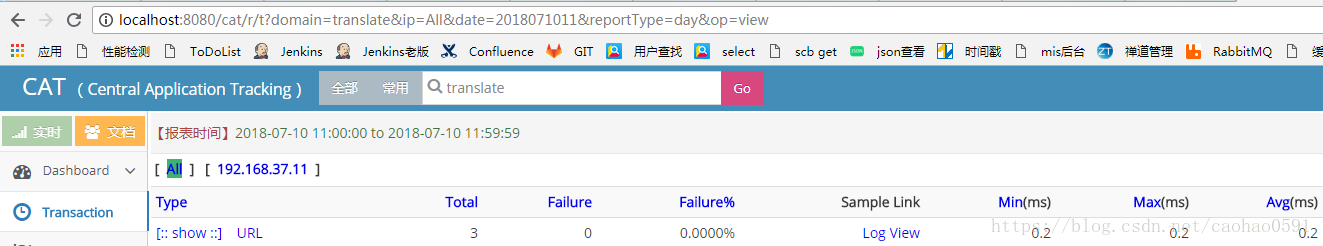
    CAT默认会将/data/appdatas/cat 作为CAT Home目录，这个目录至关重要，CAT客户端配置文件 client.xml 是在这个目录内，当然，如果你是在 windows 下调试， 你也可以在src/main/resources/ 目录下新建 META-INF/cat 目录， 并将 client.xml 配置文件放入 src/main/resources/META-INF/cat 目录里，你可以为你的监控配置domain，即项目名，如下配置 <domain id="translate"/> 。

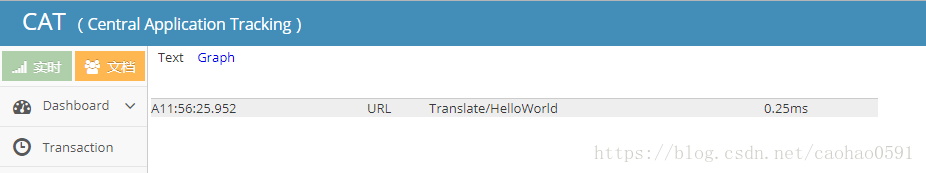


**第一个监控程序：**

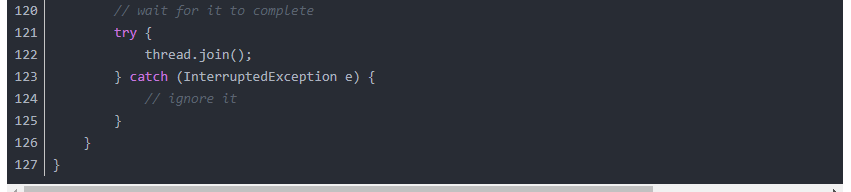
    所有都配置好了，接下来我们来写第一个监控程序，监控都是由用户自己埋点，当然，在本书最后，我们将会讲解CAT监控如何与各个流行框架之间更好的融合，以帮助用户达到无侵入式的监控埋点，假设我们对用户提供了一个翻译服务，其中有个接口就是HelloWorld，在下面程序中，我们将以事务日志形式记录用户调用行为：

    好了，我们再去管理平台去看看报表信息把，Transaction事务报表中，type=URL的事务有3条，我们通常用URL类型的事务消息标志着接口服务的开始，展开之后，我们看到这个里面该项目提供的3个服务被调用了，其中就有我们的 Translate/HelloWorld，点LogView进入原始日志。

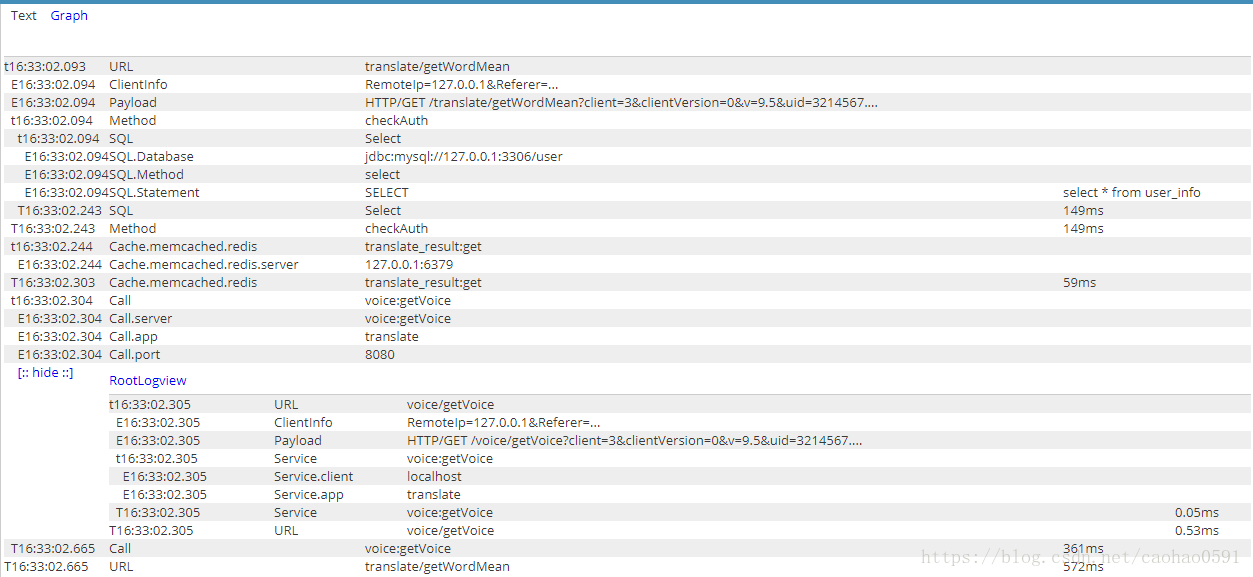




    接下来，我们来看一个更复杂的案例，涉及服务的调用以及数据库、缓存的调用，如下：

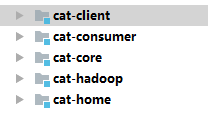


    这是我们一个对外提供的翻译服务，getWordMean为服务控制器入口，我们将原始消息展开如下，整个服务处理耗时572ms，一进来我们会记录URL类型事务，以及调用参数，然后记录用户校验函数，在函数内部，有查询用户信息的数据库操作，也会被记录下，查询耗时149ms，接下来我们会先从缓存获取结果， 缓存查询耗时 59ms，随后我们翻译内容，翻译之后我们会调用语音服务提供的发音接口，voice/getVoice，发音接口调用一共耗时 361 ms。



# 第二章 CAT服务端初始化

## Cat模块



Cat-client : cat客户端，用户可以通过引用编译后的 cat-client-2.0.0.jar包向cat-home上报统一格式的日志信息，可以集成到 mybatis、spring、dubbo等等流行框架。

Cat-consumer： 用于实时分析从客户端提供的数据。在实际开发和部署中，Cat-consumer和Cat-home是部署在一个JVM内部，每个CAT服务端都可以作为consumer也可以作为home，这样既能减少整个层级结构，也可以增加系统稳定性。

Cat-core：Cat核心模块

Cat-hadoop ： 大数据统计依赖模块。

cat-home：服务器端主程序，编译安装之后生成 cat-alpha-2.0.0.war 包部署于servlet容器，我们用的是Tomcat，war包依赖cat-client.jar、cat-consumer.jar， cat-core.jar， cat-hadoop.jar 包，通过web.xml 配置，看到Cat会启动 cat-servlet 和 mvc-servlet , mvc-servlet 是一个类似 spring MVC 的框架，用于处理用户WEB管理平台请求。cat-servlet是一个日志服务器入口，CAT会在这里开启端口监听，接收处理客户端的日志记录请求，本章主要介绍cat-servlet。

**[html]** [view plain](https://blog.csdn.net/caohao0591/article/details/80207771" \o "view plain) [copy](https://blog.csdn.net/caohao0591/article/details/80207771" \o "copy)

1. **<?xml** version="1.0" encoding="UTF-8"**?>**
2. **<web-app** xmlns="http://java.sun.com/xml/ns/javaee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3. xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app\_2\_5.xsd"
4. version="2.5"**>**
5. ...
6. **<servlet>**
7. **<servlet-name>**cat-servlet**</servlet-name>**
8. **<servlet-class>**com.dianping.cat.servlet.CatServlet**</servlet-class>**
9. **<load-on-startup>**1**</load-on-startup>**
10. **</servlet>**
11. **<servlet>**
12. **<servlet-name>**mvc-servlet**</servlet-name>**
13. **<servlet-class>**org.unidal.web.MVC**</servlet-class>**
14. **<init-param>**
15. **<param-name>**cat-client-xml**</param-name>**
16. **<param-value>**client.xml**</param-value>**
17. **</init-param>**
18. **<init-param>**
19. **<param-name>**init-modules**</param-name>**
20. **<param-value>**false**</param-value>**
21. **</init-param>**
22. **<load-on-startup>**2**</load-on-startup>**
23. **</servlet>**
24. ....

## Cat-servlet初始化

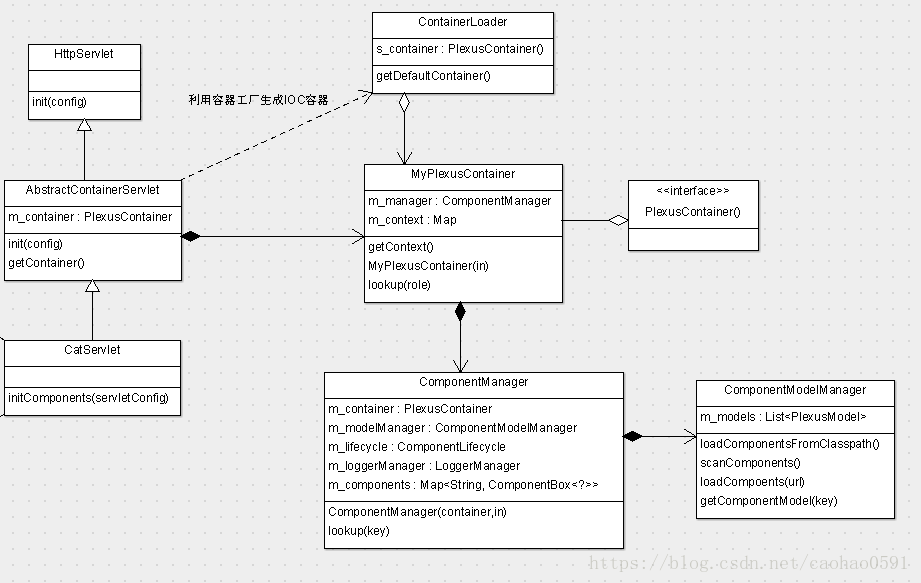


                                                                        图1 - 容器初始化类图

CatServlet 首先会调用父类 AbstractContainerServlet 的init方法做初始化工作， 可以认为这是CatServlet的入口，源码如下：

**[html]** [view plain](https://blog.csdn.net/caohao0591/article/details/80207771" \o "view plain) [copy](https://blog.csdn.net/caohao0591/article/details/80207771" \o "copy)

1. public abstract class AbstractContainerServlet extends HttpServlet {
2. public void init(ServletConfig config) throws ServletException {
3. super.init(config);
5. try {
6. if(this.m\_container == null) {
7. this.m\_container = ContainerLoader.getDefaultContainer();
8. }
10. this.m\_logger = this.m\_container.getLogger();
11. this.initComponents(config);
12. } catch (Exception var3) {
13. ...
14. }
15. }
16. }

 init方法首先调用基类HttpServlet的init方法对Servlet进行初始化.

## plexus - IOC容器

 init方法在初始化完Servlet之后通过 ContainerLoader.getDefaultContainer(); 初始化plexus容器。

    注：这里可能大家不太了解plexus，它相当于Spring的IoC容器，但是它和Spring框架不同，它并不是一个完整的，拥有各种组件的大型框架，仅仅是一个纯粹的IoC容器，它的开发者与Maven的开发者是同一拨人，最初开发Maven的时候，Spring并不成熟，所以Maven的开发者决定使用自己维护的IoC容器Plexus，它与Spring在语法和描述方式稍有不同。在Plexus中，有ROLE的概念，相当于Spring中的一个Bean。支持组件生命周期管理。

   非JAVA开发者不懂IOC容器？简单来说，IOC容器就相当于一个对象装载器，对象不是由程序员new创建，而是框架在初始化的时候从配置文件例如 components-cat-client.xml 中读取需要实例化的类信息，将信息装入一个对象装载器，然后在需要的时候，从对象装载器中找是否存在该类的信息，存在则返回类的对象。

        plexus容器是如何工作的呢？就上面的类图来说，

    a. AbstractContainerServlet 通过容器工厂ContainerLoader 的 getDefaultContainer方法，该方法会创建 MyPlexusContainer 容器，MyPlexusContainer容器在构造函数中会创建组件管理器ComponentManager，

     b.组件管理器ComponentManager会创建模型管理器 ComponentModelManager以及组件生命周期管理器ComponentLifecycle，然后调用 ComponentModelManager的loadComponentsFromClasspath()方法扫描各个jar包中存在的IOC容器类配置文件，如图2。

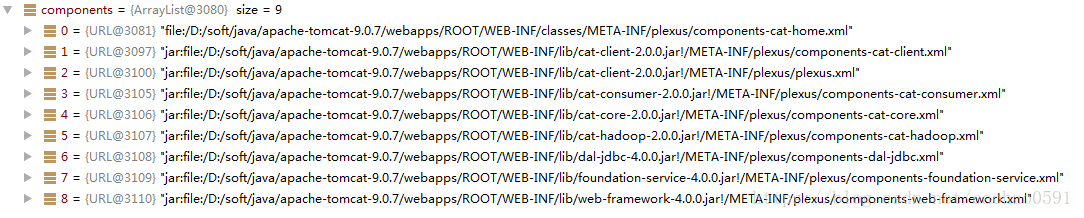


            图2 - plexus IOC容器类配置文件

       c.调用ComponentModelManager的loadCompoents(URL url)方法解析每个xml 文件，读取相应的role(相当于spring的bean)，放到ComponentModelManager的List<PlexusModel>列表中。

       d.然后就可以通过lookup方法找到类，并在首次使用的时候实例化，而且如果类方法实现了 Initializable 接口，创建对象后会执行类的 initialize() 方法做一些初始化的工作，参考下边源码

**[java]** [view plain](https://blog.csdn.net/caohao0591/article/details/80207771" \o "view plain) [copy](https://blog.csdn.net/caohao0591/article/details/80207771" \o "copy)

1. **if**(component **instanceof** Initializable) {
2. **try** {
3. ((Initializable)component).initialize();
4. } **catch** (Throwable var5) {
5. ComponentModel model = ctx.getComponentModel();
6. **throw** **new** ComponentLookupException("Error when initializing component!", model.getRole(), model.getHint(), var5);
7. }
8. }

## 模块的加载 - 模型模式

init方法最后调用CatServlet的initComponents方法初始化Module模块。

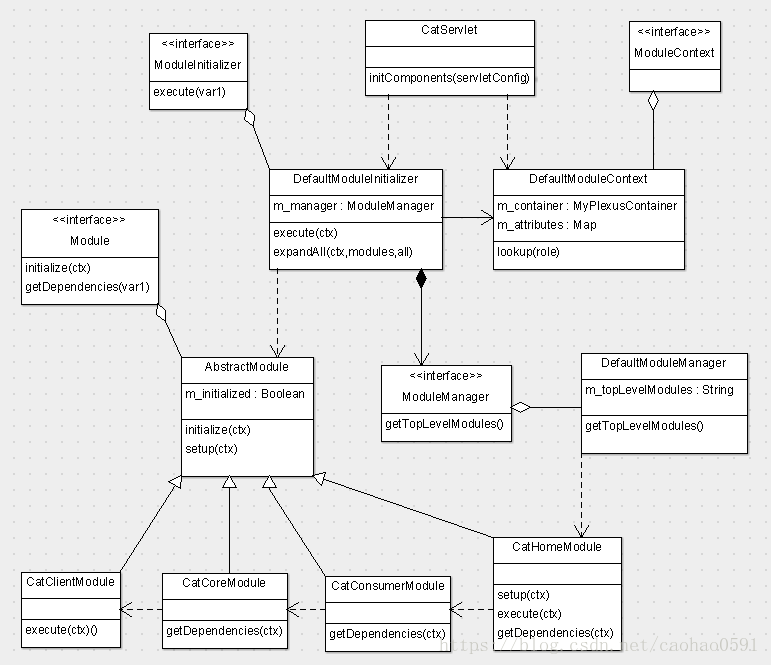


                图3 - 模块初始化类图

initComponents方法首先创建一个模块上下文 DefaultModuleContext对象,该对象拥有之前初始化过的plexus容器，以及配置文件信息 server.xml，client.xml，由于cat-home默认是服务端也是客户端，也就是说cat-home自身也会被监控，所以我们在这里看到有client.xml配置，配置文件所在目录由环境变量CAT\_HOME指定，如果未指定，默认是/data/appdatas/cat。

   然后创建一个模块初始器 DefaultModuleInitializer，并调用他的execute(ctx)方法创建并初始化模块。

    注：DefaultModuleInitializer有一个模块管理器DefaultModelManager m\_manager, 读者可能没有看见m\_manager的创建过程，实际上，对象在components-foundation-service.xml配置文件中配置的，然后在plexus容器实例化类对象的过程中创建的，后面还有很多对象的属性也是通过plexus容器注入的。比如DefaultModuleManager的m\_topLevelModules属性通过以下配置注入。

**[html]** [view plain](https://blog.csdn.net/caohao0591/article/details/80207771" \o "view plain) [copy](https://blog.csdn.net/caohao0591/article/details/80207771" \o "copy)

1. **<component>**
2. **<role>**org.unidal.initialization.ModuleManager**</role>**
3. **<implementation>**org.unidal.initialization.DefaultModuleManager**</implementation>**
4. **<configuration>**
5. **<topLevelModules>**cat-home**</topLevelModules>**
6. **</configuration>**
7. **</component>**

    上面XML配置显示m\_topLevelModules 指定为 cat-home，这样DefaultModuleInitializer通过DefaultModelManager的getTopLevelModules()方法获取的就是CatHomeModule模块对象，可以认为cat-home是一个顶层模块，所有Module都包含getDependencies方法，该方法会找到当前模块所依赖的其他模块，并实例化模块，比如下面cat-home就依赖cat-consumer模块，

**[java]** [view plain](https://blog.csdn.net/caohao0591/article/details/80207771" \o "view plain) [copy](https://blog.csdn.net/caohao0591/article/details/80207771" \o "copy)

1. **public** **class** CatHomeModule **extends** AbstractModule {
2. @Override
3. **public** Module[] getDependencies(ModuleContext ctx) {
4. **return** ctx.getModules(CatConsumerModule.ID);
5. }
6. }

从cat-consumer的getDependencies看出他依赖cat-core模块，cat-core模块又依赖cat-client模块，这样子我们就从顶层模块引出了所有依赖的其它模块，在实例化模块的同时调用模块的setup方法安装模块。在所有模块安装完成之后，依次调用模块的execute方法完成初始化，但是初始化顺序则是按照安装顺序反着来的，cat-client -> cat-core -> cat-consumer -> cat-home ，Modules之间的设计使用了典型的模板模式。

## cat-home的setup

在上一章讲到模块初始化的时候， 讲到setup安装cat-home模块，对于客户端的请求的监听处理，就是在这里完成的。

**[java]** [view plain](https://blog.csdn.net/caohao0591/article/details/80207771" \o "view plain) [copy](https://blog.csdn.net/caohao0591/article/details/80207771" \o "copy)

1. @Named(type = Module.**class**, value = CatHomeModule.ID)
2. **public** **class** CatHomeModule **extends** AbstractModule {
3. @Override
4. **protected** **void** setup(ModuleContext ctx) **throws** Exception {
5. **if** (!isInitialized()) {
6. File serverConfigFile = ctx.getAttribute("cat-server-config-file");
7. ServerConfigManager serverConfigManager = ctx.lookup(ServerConfigManager.**class**);
8. **final** TcpSocketReceiver messageReceiver = ctx.lookup(TcpSocketReceiver.**class**);
10. serverConfigManager.initialize(serverConfigFile);
11. messageReceiver.init();
13. Runtime.getRuntime().addShutdownHook(**new** Thread() {
14. @Override
15. **public** **void** run() {
16. messageReceiver.destory();
17. }
18. });
19. }
20. }
21. }

1、读取 server.xml 配置，装进配置管理器(ServerConfigManager)。

2、创建消息接收器 final TcpSocketReceiver messageReceiver;

3、messageReceiver.init() 初始化服务,采用的经典的 netty reactor 模型。

4、注册一个JVM关闭的钩子，在进程挂掉的时候，执行一些清理现场的代码。

## TcpSocketReceiver--- netty reactor 模式的应用

我们来看看CatHomeModule对TcpSocketReceiver的初始化做了什么，如下源码：

**[java]** [view plain](https://blog.csdn.net/caohao0591/article/details/80207771" \o "view plain) [copy](https://blog.csdn.net/caohao0591/article/details/80207771" \o "copy)

1. **public** **final** **class** TcpSocketReceiver **implements** LogEnabled {
2. **public** **void** init() {
3. **try** {
4. startServer(m\_port);
5. } **catch** (Throwable e) {
6. m\_logger.error(e.getMessage(), e);
7. }
8. }
10. **public** **synchronized** **void** startServer(**int** port) **throws** InterruptedException {
11. **boolean** linux = getOSMatches("Linux") || getOSMatches("LINUX");
12. **int** threads = 24;
13. ServerBootstrap bootstrap = **new** ServerBootstrap();
15. m\_bossGroup = linux ? **new** EpollEventLoopGroup(threads) : **new** NioEventLoopGroup(threads);
16. m\_workerGroup = linux ? **new** EpollEventLoopGroup(threads) : **new** NioEventLoopGroup(threads);
17. bootstrap.group(m\_bossGroup, m\_workerGroup);
18. bootstrap.channel(linux ? EpollServerSocketChannel.**class** : NioServerSocketChannel.**class**);
20. bootstrap.childHandler(**new** ChannelInitializer<SocketChannel>() {
21. @Override
22. **protected** **void** initChannel(SocketChannel ch) **throws** Exception {
23. ChannelPipeline pipeline = ch.pipeline();
25. pipeline.addLast("decode", **new** MessageDecoder());
26. }
27. });
29. bootstrap.childOption(ChannelOption.SO\_REUSEADDR, **true**);
30. bootstrap.childOption(ChannelOption.TCP\_NODELAY, **true**);
31. bootstrap.childOption(ChannelOption.SO\_KEEPALIVE, **true**);
32. bootstrap.childOption(ChannelOption.ALLOCATOR, PooledByteBufAllocator.DEFAULT);
34. **try** {
35. m\_future = bootstrap.bind(port).sync();
36. m\_logger.info("start netty server!");
37. } **catch** (Exception e) {
38. m\_logger.error("Started Netty Server Failed:" + port, e);
39. }
40. }
41. }

1、创建EventLoopGroup对象， EventLoopGroup是用来处理IO操作的多线程事件循环器，m\_bossGroup作为一个acceptor负责接收来自客户端的请求，然后分发给m\_workerGroup用来所有的事件event和channel的IO。

2、创建ServerBootstrap对象，ServerBootstrap 是一个启动Epoll(非Linux为NIO)服务的辅助启动类，他将设置bossGroup和workerGroup两个多线程时间循环器。

3、接下来的channel()方法设置了ServerBootstrap 的 ChannelFactory，这里传入的参数是EpollServerSocketChannel.class （非Linux为NioServerSocketChannel.class），也就是说这个ChannelFactory创建的就是EpollServerSocketChannel/NioServerSocketChannel的实例。

    Channel是Netty的核心概念之一，它是Netty网络通信的主体，他从EventLoopGroup获得一个EventLoop，并注册到该EventLoop，channel生命周期内都和该EventLoop在一起，由它负责对网络通信连接的打开、关闭、连接和读写操作。如果是对于读写事件，执行线程调度pipeline来处理用户业务逻辑。

4、接下来bootstrap.childHandler的目的是添加一个handler，用来监听已经连接的客户端的Channel的动作和状态，传入的 ChannelInitializer重写了initChannel方法，这个方法在Channel被注册到EventLoop的时候会被调用。

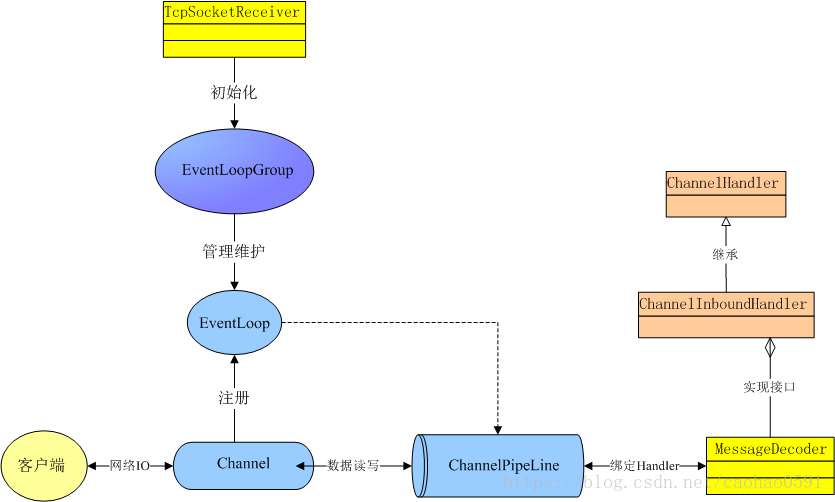
5、initChannel会创建ChannelPipeline对象，并调用addLast添加ChannelHandler。有网络请求时，ChannelPipeline会调用ChannelHandler来处理，有ChannelInboundHandler和ChannelOutboundHandler两种，ChannelPipeline会从头到尾顺序调用ChannelInboundHandler处理网络请求内容，从尾到头调用ChannelOutboundHandler处理网络请求内容。这也是Netty用来灵活处理网络请求的机制之一，因为使用的时候可以用多个decoder和encoder进行组合，从而适应不同的网络协议。而且这种类似分层的方式可以让每一个Handler专注于处理自己的任务而不用管上下游，这也是pipeline机制的特点。这跟TCP/IP协议中的五层和七层的分层机制有异曲同工之妙。

      在这里，ChannelPipeline添加的 ChannelHandler 是MessageDecoder ，MessageDecoder的祖先类实现了ChannelHandler接口，他本质上还是一个Handler，是网络IO事件具体处理类，当客户端将日志数据上传到服务器之后，会交给MessageDecoder 解码数据，然后进行后续处理。

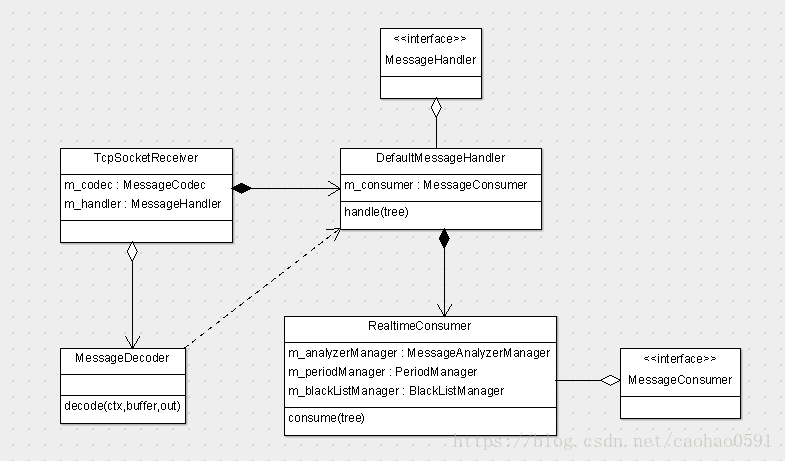
6、调用 childOption 设置 channel 的参数。

7、最后调用bind()方法启动服务。

关于netty ，我就讲到这里，网上关于netty框架的文章非常多，大家可以自行去查。



## 消息的解码



上一章我们讲到Netty将接收到的消息交给 MessageDecoder 去做解码，解码是交由PlainTextMessageCodec对象将接收到的字节码反序列化为MessageTree对象(所有的消息都是由消息树来组织)，具体的解码逻辑在这里暂不做详细阐述，在第三章我们会阐述编码过程，解码只是编码的一个逆过程。

解码之后调用 DefaultMessageHandler 的 handle方法对消息进行处理，handle方法就干了一件事情，就是调用 m\_consumer.consume(tree) 方法去消费消息树，在消费模块，CAT实现了队列化，异步化，在消息消费章节会详细阐述。

当然netty handler也是支持异步处理的，我们也可以将 DefaultMessageHandler 像 MessageDecoder那样向netty注册handler, 再由netty来做线程池分发。

**[java]** [view plain](https://blog.csdn.net/caohao0591/article/details/80207771" \o "view plain) [copy](https://blog.csdn.net/caohao0591/article/details/80207771" \o "copy)

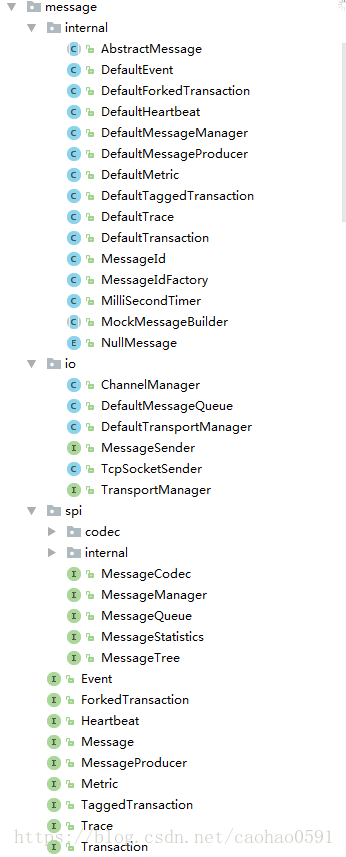
1. **public** **class** MessageDecoder **extends** ByteToMessageDecoder {
3. @Override
4. **protected** **void** decode(ChannelHandlerContext ctx, ByteBuf buffer, List<Object> out) **throws** Exception {
5. **if** (buffer.readableBytes() < 4) {
6. **return**;
7. }
8. buffer.markReaderIndex();
9. **int** length = buffer.readInt();
11. ...
13. ByteBuf readBytes = buffer.readBytes(length + 4);
15. ...
17. DefaultMessageTree tree = (DefaultMessageTree) m\_codec.decode(readBytes);
19. readBytes.resetReaderIndex();
20. tree.setBuffer(readBytes);
21. m\_handler.handle(tree);
22. m\_processCount++;
24. ...
25. }
26. }

具体对消息的消费我们会在讲完客户端原理之后再来讲解。

# 第三章 CAT客户端实现

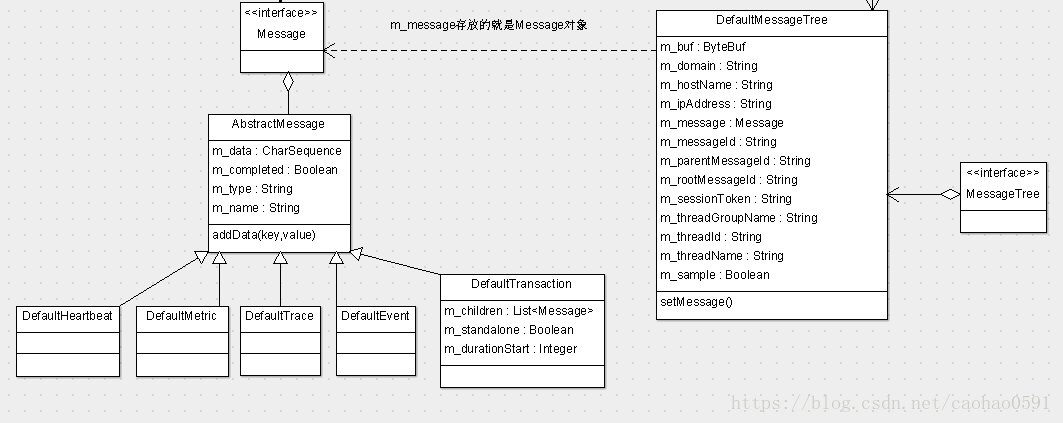
## cat客户端部分核心类

* message目录下面有消息相关的部分接口
* internal目录包含主要的CAT客户端内部实现类；
* io目录包含建立服务端连接、重连、消息队列监听、上报等io实现类；
* spi目录为上报消息工具包，包含消息二进制编解码、转义等实现类。



## 消息的组织 - 消息树

Cat使用消息树（MessageTree）组织日志，下面为消息树的类定义



我们每次操作的实体都是消息树，其中有个domain字段，这是cat中一个非常重要的概念，一个domain可以对应成一个project，每个消息树拥有一个唯一的MessageId， 不同的消息树(比如微服务中A服务调用B服务，A，B都会生成消息树) 通过 parenMessageId、rootMessageId 串联起来，消息树下的所有实体都是Message，一共有5种类型的Message, 分别是Transaction, Event, Trace, Metric和Heartbeat。

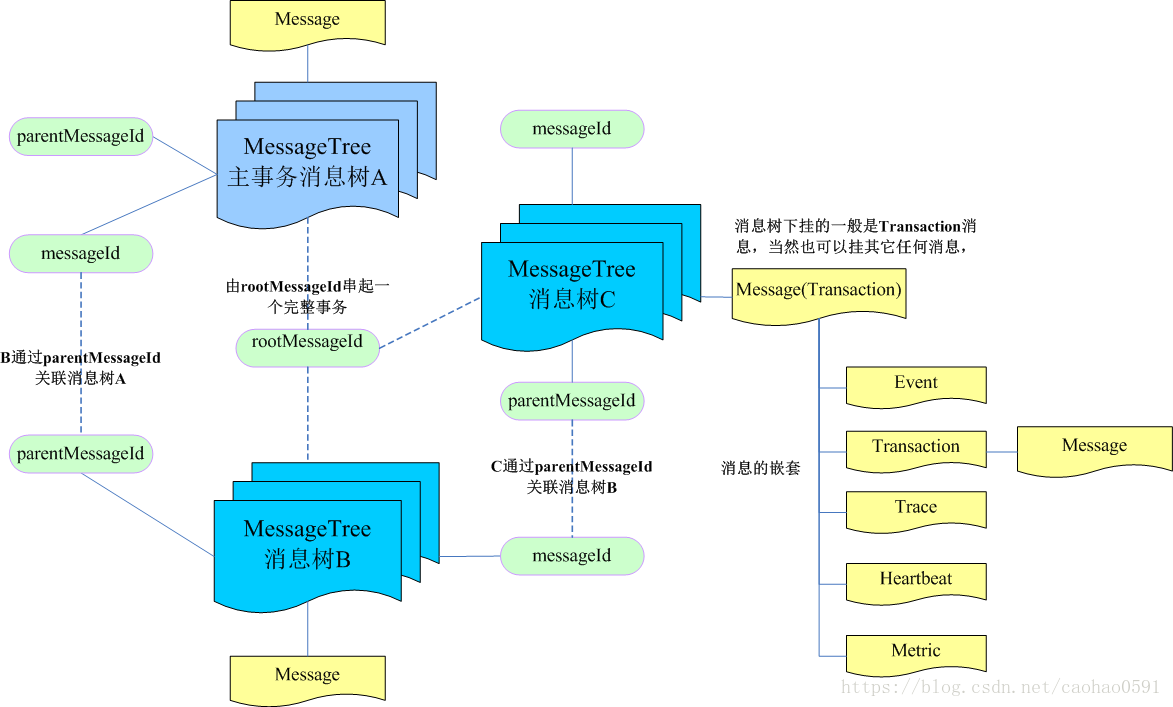
Transaction：可以理解为是一个事务，事务之间可以互相嵌套，事务还可以嵌套任意其他消息类型，存放在List<Message> m\_children属性中，也只有事务才可以嵌套。一般用来记录跨越系统边界的程序访问行为，比如远程调用，数据库调用，也适合执行时间较长的业务逻辑监控。

Event：代表系统是在某个时间点发生的一次事件，例如新用户注册、登陆，系统异常等，理论上可以记录任何事情，它和transaction相比缺少了时间的统计，开销比transaction要小。还可以用来记录两个事务之间的关系，分支事务通过设置消息树的parentMessageId维护与主事务消息之间的关系。

Trace：用于记录一些trace、debug这类的信息，比如log4j打印日志。以便于快速调试定位问题

Metric：用于记录业务指标、指标可能包含对一个指标记录次数、记录平均值、记录总和

Heartbeat：主要用于记录系统的心跳信息，比如CPU%， MEM%，连接池状态，系统负载等。



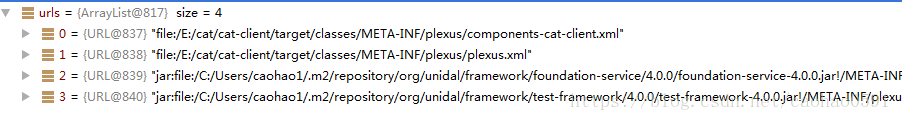
## 客户端的初始化

客户端操作对象Cat封装了所有的接口。我们先通过下面源码来了解下Cat的初始化过程。

**[java]** [view plain](https://blog.csdn.net/caohao0591/article/details/80207806" \o "view plain) [copy](https://blog.csdn.net/caohao0591/article/details/80207806" \o "copy)

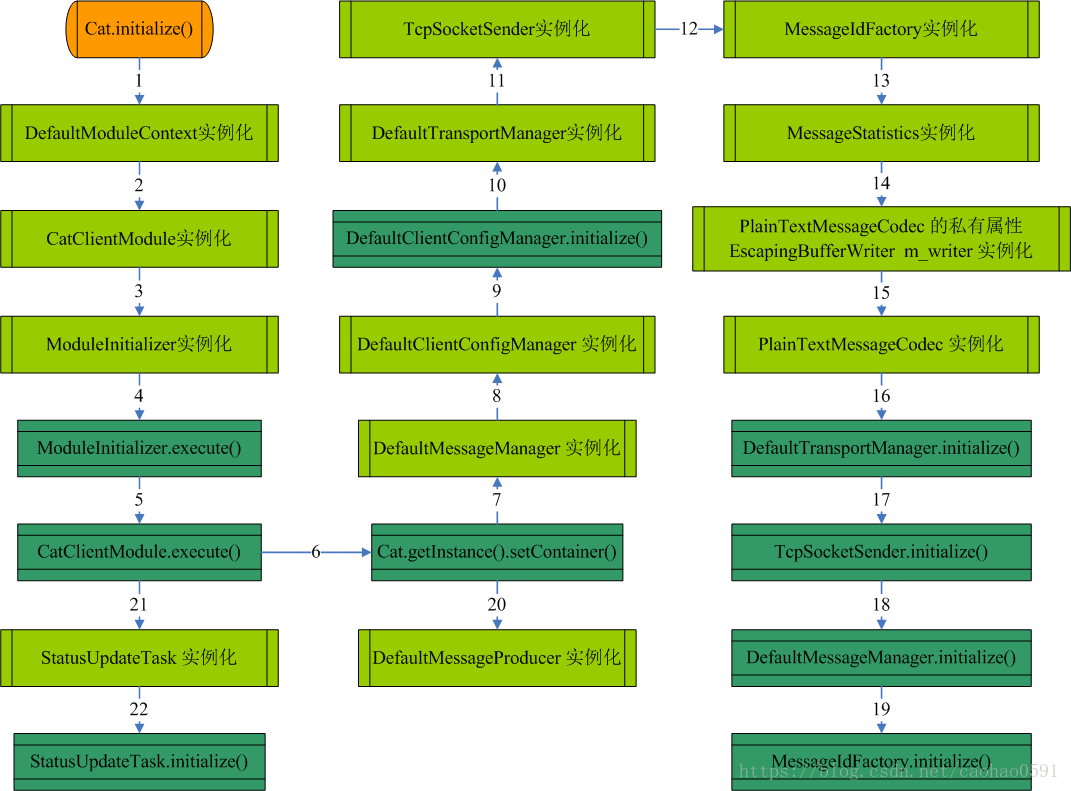
1. **class** Cat{
2. **public** **static** Transaction newTransaction(String type, String name) {
3. **return** Cat.getProducer().newTransaction(type, name);
4. }
6. **public** **static** MessageProducer getProducer() {
7. checkAndInitialize();
9. **return** s\_instance.m\_producer;
10. }
12. **private** **static** **void** checkAndInitialize() {
13. **if** (!s\_init) {
14. **synchronized** (s\_instance) {
15. **if** (!s\_init) {
16. initialize(**new** File(getCatHome(), "client.xml"));
17. log("WARN", "Cat is lazy initialized!");
18. s\_init = **true**;
19. }
20. }
21. }
22. }
24. **public** **static** **void** initialize(File configFile) {
25. PlexusContainer container = ContainerLoader.getDefaultContainer();
27. initialize(container, configFile);
28. }
30. **public** **static** **void** initialize(PlexusContainer container, File configFile) {
31. ModuleContext ctx = **new** DefaultModuleContext(container);
32. Module module = ctx.lookup(Module.**class**, CatClientModule.ID);
34. **if** (!module.isInitialized()) {
35. ModuleInitializer initializer = ctx.lookup(ModuleInitializer.**class**);
37. ctx.setAttribute("cat-client-config-file", configFile);
38. initializer.execute(ctx, module);
39. }
40. }
41. }

从上面代码可以看到， 创建transaction首先会通过getProducer函数获取消息生产者MessageProducer对象，在返回MessageProducer对象之前，函数会对客户端进行初始化，设置 CatHome目录，默认是/data/appdatas/cat ，读取配置文件 client.xml，接下来的初始化流程和服务器类似， 也是先通过 ContainerLoader.getDefaultContainer(); 初始化plexus容器。客户端需要被装入容器的对象在下图的配置中：



-----------------------------------------------------------------------------------------------

我们借助components-cat-client.xml 配置来看部分类的实例化与初始化顺序：



上面流程中非常多的 initialize()  方法，有些是主动调用的，有些则是由容器框架调用的，这个在讲plexus容器的时候说过，如果类方法实现了 Initializable 接口，创建实例后会执行类的 initialize() 方法做一些初始化的工作。

------------------------------------------------------------------------------------------------

然后创建一个模块上下文 DefaultModuleContext,该对象拥有之前初始化过的plexus容器，以及配置文件信息/data/appdatas/cat/client.xml，然后创建一个模块初始器 DefaultModuleInitializer，随后调用模块初始器的execute(ctx)方法初始化模块，这里需要初始化的模块只有 CatClientModule,他也不依赖其它任何模块，我们调用setup，这是个空方法，也就是说模块的安装不需要干任何事情，接着调用CatClientModule的excute方法，

以下为execute的源码：

**[java]** [view plain](https://blog.csdn.net/caohao0591/article/details/80207806" \o "view plain) [copy](https://blog.csdn.net/caohao0591/article/details/80207806" \o "copy)

1. **class** CatClientModule {
2. @Override
3. **protected** **void** execute(**final** ModuleContext ctx) **throws** Exception {
4. ctx.info("Current working directory is " + System.getProperty("user.dir"));
6. // initialize milli-second resolution level timer
7. MilliSecondTimer.initialize();
9. // tracking thread start/stop
10. Threads.addListener(**new** CatThreadListener(ctx));
12. // warm up Cat
13. Cat.getInstance().setContainer(((DefaultModuleContext) ctx).getContainer());
15. // bring up TransportManager
16. ctx.lookup(TransportManager.**class**);
18. ClientConfigManager clientConfigManager = ctx.lookup(ClientConfigManager.**class**);
20. **if** (clientConfigManager.isCatEnabled()) {
21. // start status update task
22. StatusUpdateTask statusUpdateTask = ctx.lookup(StatusUpdateTask.**class**);
24. Threads.forGroup("cat").start(statusUpdateTask);
25. LockSupport.parkNanos(10 \* 1000 \* 1000L); // wait 10 ms
27. // MmapConsumerTask mmapReaderTask = ctx.lookup(MmapConsumerTask.class);
28. // Threads.forGroup("cat").start(mmapReaderTask);
29. }
30. }
31. }

StatusUpdateTask statusUpdateTask = ctx.lookup(StatusUpdateTask.class);  
Threads.forGroup("cat").start(statusUpdateTask);

CatClientModule从StatusUpdateTask中启动一个线程来每隔一段时间发送一个HeartBeatMessage，其中包括了客户端能拿到的各种信息，包括CPU，Memory，Disk等等，开发者也可以通过实现StatusExtension接口的方式来实现对于HeartBeatMessage发送内容的扩展。

其中比较重要的实现信息收集的是这行代码

**[java]** [view plain](https://blog.csdn.net/caohao0591/article/details/80207806" \o "view plain) [copy](https://blog.csdn.net/caohao0591/article/details/80207806" \o "copy)

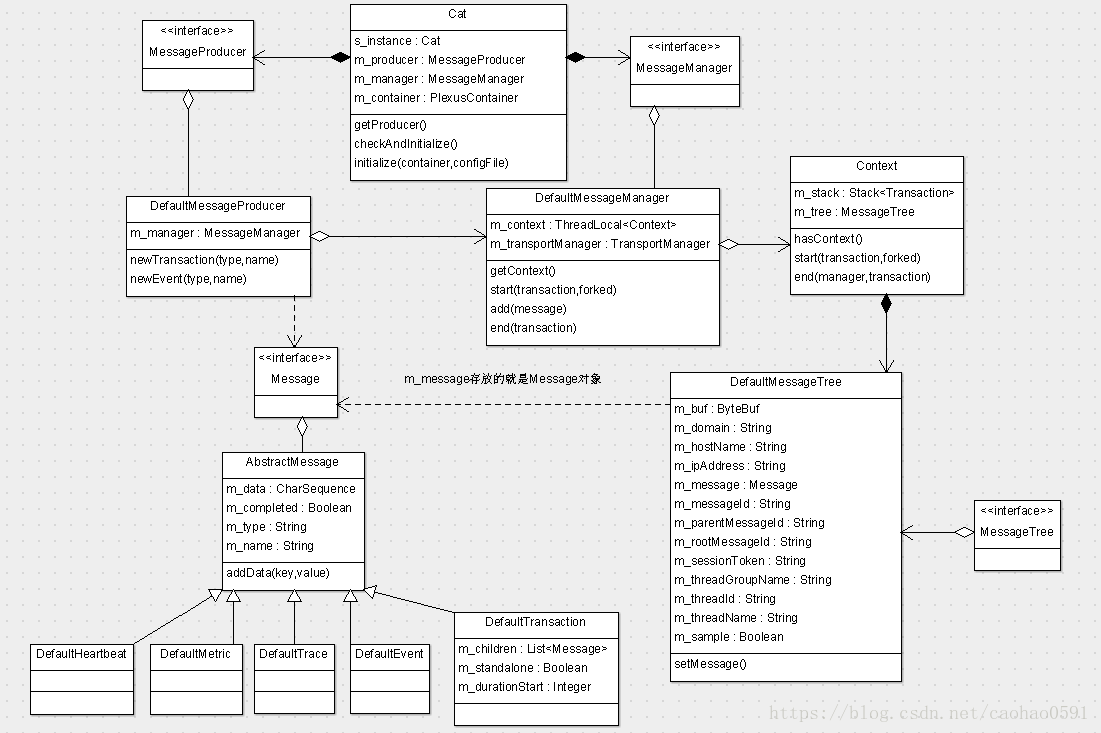
1. **public** **class** StatusUpdateTask **implements** Task, Initializable {
2. @Override
3. **public** **void** run() {
4. ...
5. StatusInfoCollector statusInfoCollector = **new** StatusInfoCollector(m\_statistics, m\_jars);
6. status.accept(statusInfoCollector.setDumpLocked(m\_manager.isDumpLocked()));
7. ...
8. }
9. }

m\_statistics包含的是已经发送过信息的容量，m\_jars是通过classLoader加载的jar包名称，StatusInfoCollector通过大量访问者模式的代码实现了将各种指标set到status中的功能，之后将status封装到HeartBeatMessage中，按照一般对于message的处理流程，flush到消息传输层中。

## 消息生产 -- 入栈

我们获取消息生产者对象 MessageProducer 之后，就可以调用 newTransaction(type, name) 来创建 Transaction类消息了,

值得注意的是MessageProducer对业务封装了CAT内部的所有细节，所以业务方只需要一个MessageProducer对象就可以完成消息的所有操作。



下面通过 newTransaction 的源码来分析 Transaction消息的创建过程

**[java]** [view plain](https://blog.csdn.net/caohao0591/article/details/80207806" \o "view plain) [copy](https://blog.csdn.net/caohao0591/article/details/80207806" \o "copy)

1. **class** DefaultMessageProducer {
2. **public** Transaction newTransaction(String type, String name) {
3. // this enable CAT client logging cat message without explicit setup
4. **if** (!m\_manager.hasContext()) {
5. m\_manager.setup();
6. }
8. **if** (m\_manager.isMessageEnabled()) {
9. DefaultTransaction transaction = **new** DefaultTransaction(type, name, m\_manager);
11. m\_manager.start(transaction, **false**);
12. **return** transaction;
13. } **else** {
14. **return** NullMessage.TRANSACTION;
15. }
16. }
17. }

他首先通过消息管理者MassageManager判断是否存在消息上下文context，如果不存在则在setup中创建消息上下文。

## Context 线程本地变量

消息上下文 Context 采用的是线程本地变量。通过ThreadLocal存取Context数据。

高并发下日志的打印通常会采用这种方式，或者说一次事务的日志一起打印，因为一般默认一次事务都是由同一个线程执行的（如一次http请求），将事务的日志保存在线程局部变量当中，当事务执行完成的时候统一打印。

为什么需要用到线程本地变量？在低并发请求下，一条日志会很快被处理，普通变量即可满足需求，很少出现多个线程同时读写同一个变量，

然在高并发场景下，多个线程同时读写同一个变量会导致不可预知的结果，我们称这为线程非安全，比如线程A要写一大段日志，写到一半，线程B获得CPU执行时间片开始写日志，AB的日志就会交错混乱，有同学会问，为什么不用同步锁？这是一个方案，同步锁是一个相对较复杂的保证线程安全，保证同时只有一个线程可以读写变量，其它线程要读写变量就需要排队，这就必然会带来高延迟，

线程本地变量功用则非常简单，就是为每一个使用该变量的线程都提供一个变量值的副本，是Java中一种较为特殊的线程绑定机制，JVM 为每个运行的线程，绑定了私有的本地实例存取空间，每一个线程都可以独立地改变自己的副本，而不会和其它线程的副本冲突，从而为多线程环境常出现的并发访问问题提供了一种隔离机制，但是会造成数据冗余，是一种用空间换时间的线程安全方案。

ThreadLocal是如何做到为每一个线程维护变量的副本的呢？其实实现的思路很简单，在ThreadLocal类中有一个Map，用于存储每一个线程的变量的副本。大家看下面代码，核心在于ThreadLocal的get 和 set函数，函数首先会获取当前线程，然后从 Map 中获取或者设置该线程的Context

**[java]** [view plain](https://blog.csdn.net/caohao0591/article/details/80207806" \o "view plain) [copy](https://blog.csdn.net/caohao0591/article/details/80207806" \o "copy)

1. **public** **class** ThreadLocal<T> {
2. **public** T get() {
3. Thread t = Thread.currentThread();
4. ThreadLocalMap map = getMap(t);
5. **if** (map != **null**) {
6. ThreadLocalMap.Entry e = map.getEntry(**this**);
7. **if** (e != **null**) {
8. @SuppressWarnings("unchecked")
9. T result = (T)e.value;
10. **return** result;
11. }
12. }
13. **return** setInitialValue();
14. }
15. }

**[java]** [view plain](https://blog.csdn.net/caohao0591/article/details/80207806" \o "view plain) [copy](https://blog.csdn.net/caohao0591/article/details/80207806" \o "copy)

1. **public** **class** DefaultMessageManager **extends** ContainerHolder **implements** MessageManager, Initializable, LogEnabled {
2. **private** ThreadLocal<Context> m\_context = **new** ThreadLocal<Context>();
3. **private** Context getContext() {
4. **if** (Cat.isInitialized()) {
5. Context ctx = m\_context.get();
7. **if** (ctx != **null**) {
8. **return** ctx;
9. } **else** {
10. **if** (m\_domain != **null**) {
11. ctx = **new** Context(m\_domain.getId(), m\_hostName, m\_domain.getIp());
12. } **else** {
13. ctx = **new** Context("Unknown", m\_hostName, "");
14. }
16. m\_context.set(ctx);
17. **return** ctx;
18. }
19. }
21. **return** **null**;
22. }
23. }

在Context构造函数里，我们看到了消息树MessageTree和Transaction栈被创建了，由于Context是线程本地变量，由此可以推断，每个线程都拥有各自的消息树和事务栈，这里所说的线程都是业务线程，Context属于MessageManager的内部类。可以认为MessageManager的其中一个功能是作为context的一个代理，MessageManager的start、add、end等方法，核心都是调用当前线程context的start、add、end方法。

## Transaction事务的开启

接着MessageProducer就会创建一个Transation对象，然后将Transaction对象交给 MessageManager启动，我们通过下面源码看看的启动流程，最关键的启动步骤是调用ctx.start(transactionm forked) 完成的，下面分析这个方法：

1.如果 m\_stack 不为空， 而且 transaction 类型为 ForkedTransaction  
    1.计算时间 或 长度条件，如果需要发送到Server，就发送到Server里（truncateAndFlush）  
    2.将当前 transaction 加到 m\_stack 栈顶元素的子消息中去。  
    3.m\_length++;

2.如果m\_stack为空，就把当前这个Transaction加到MessageTree里面。

3.最后判断 transaction 是否是forked的事务，不是则将transaction加入 m\_stack 。

**[java]** [view plain](https://blog.csdn.net/caohao0591/article/details/80207806" \o "view plain) [copy](https://blog.csdn.net/caohao0591/article/details/80207806" \o "copy)

1. **public** **class** DefaultMessageManager **extends** ContainerHolder **implements** MessageManager, Initializable, LogEnabled {
2. @Override
3. **public** **void** start(Transaction transaction, **boolean** forked) {
4. Context ctx = getContext();
6. **if** (ctx != **null**) {
7. ctx.start(transaction, forked);
9. ...
10. }
12. **class** Context {
13. **private** MessageTree m\_tree;
14. **private** Stack<Transaction> m\_stack;
16. **public** **void** start(Transaction transaction, **boolean** forked) {
17. **if** (!m\_stack.isEmpty()) {
18. ...
19. } **else** {
20. m\_tree.setMessage(transaction);
21. }
23. **if** (!forked) {
24. m\_stack.push(transaction);
25. }
26. }
27. }
28. }

## 其他类型消息组合

**[java]** [view plain](https://blog.csdn.net/caohao0591/article/details/80207806" \o "view plain) [copy](https://blog.csdn.net/caohao0591/article/details/80207806" \o "copy)

1. @RunWith(JUnit4.**class**)
2. **public** **class** AppSimulator **extends** CatTestCase {
3. @Test
4. **public** **void** simulateHierarchyTransaction() **throws** Exception {
5. MessageProducer cat = Cat.getProducer();
6. Transaction t = cat.newTransaction("URL", "WebPage");
7. String id1 = cat.createMessageId();
8. String id2 = cat.createMessageId();

11. **try** {
12. // do your business here
13. t.addData("k1", "v1");
14. t.addData("k2", "v2");
15. t.addData("k3", "v3");
16. Thread.sleep(5);

19. cat.logMetric("payCount", "C", "1");
20. cat.logMetric("totalfee", "S", "30.5");
21. cat.logMetric("avgfee", "T", "25.6");
22. cat.logMetric("order", "S,C", "3,25.6");

25. Metric event = Cat.getProducer().newMetric("kingsoft", "praise");
26. event.setStatus("C");
27. event.addData("3");
28. event.complete();

31. Cat.getManager().setTraceMode(**true**);
32. cat.logTrace("Trace1", "debug", SUCCESS, "user\_debug\_data");

35. cat.logEvent("RuntimeException", "Name1", "ERROR", "data1");
36. cat.logEvent("Error", "Name2", SUCCESS, "data2");

39. cat.logEvent("RemoteCall", "Service1", SUCCESS, id1);
40. t.setStatus(SUCCESS);
41. } **catch** (Exception e) {
42. t.setStatus(e);
43. } **finally** {
44. t.complete();
45. }
46. }
47. }

可以通过 MessageProducer的 logEvent 记录event类型的消息，方法首先会调用newEvent方法创建Event对象，如果有消息数据，就用addData方法添加数据，然后setStatus设置消息状态，complete完成日志记录。

**[java]** [view plain](https://blog.csdn.net/caohao0591/article/details/80207806" \o "view plain) [copy](https://blog.csdn.net/caohao0591/article/details/80207806" \o "copy)

1. **public** **class** DefaultMessageProducer **implements** MessageProducer {
2. @Override
3. **public** **void** logEvent(String type, String name, String status, String nameValuePairs) {
4. Event event = newEvent(type, name);
6. **if** (nameValuePairs != **null** && nameValuePairs.length() > 0) {
7. event.addData(nameValuePairs);
8. }
10. event.setStatus(status);
11. event.complete();
12. }
14. @Override
15. **public** Event newEvent(String type, String name) {
16. **if** (!m\_manager.hasContext()) {
17. m\_manager.setup();
18. }
20. **if** (m\_manager.isMessageEnabled()) {
21. DefaultEvent event = **new** DefaultEvent(type, name, m\_manager);
23. **return** event;
24. } **else** {
25. **return** NullMessage.EVENT;
26. }
27. }
28. }

event.complet 做了什么事情？ 他会首先设置消息complete状态为true，然后调用 MessageManager 的 add 方法，并传入自身的指针，在Context 线程本地变量章节的时候说过MessageManager是context的代理，MessageManager 的add方法核心是调用的context得add方法。

context的add方法，会首先判断m\_stack栈是否为空，如果是空的说明这个消失是一个单独的非事务类型消息， 直接将消息放入MessageTree然后发送到服务器。

如果m\_stack 不为空，说明这个event消息处在一个事务下面，我们从m\_stack 栈顶获取事务，将event消息嵌套到事务里，等待事务结束的时候一同推送到服务器。上边的案例就是这种情况。

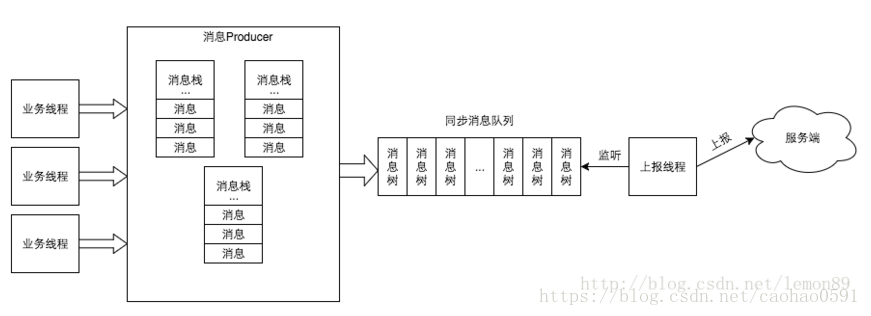
**[java]** [view plain](https://blog.csdn.net/caohao0591/article/details/80207806" \o "view plain) [copy](https://blog.csdn.net/caohao0591/article/details/80207806" \o "copy)

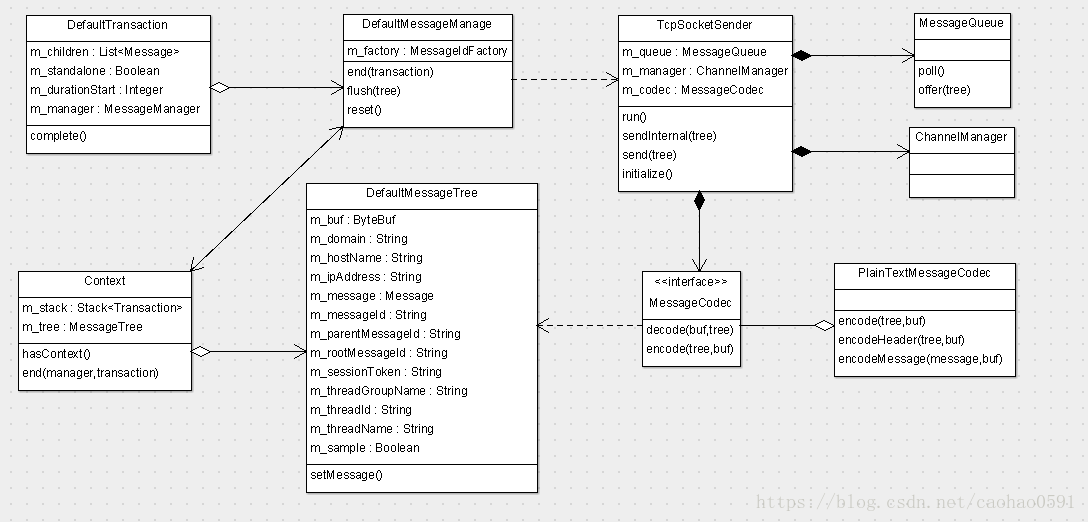
1. **class** Context {
2. **public** **void** add(Message message) {
3. **if** (m\_stack.isEmpty()) {
4. MessageTree tree = m\_tree.copy();
6. tree.setMessage(message);
7. flush(tree);
8. } **else** {
9. Transaction parent = m\_stack.peek();
11. addTransactionChild(message, parent);
12. }
13. }
14. }

我们也可以不用logEvent 记录日志，而是自己通过newEvent创建Event消息实例， 然后由自己控制什么时候add数据、setStatus以及complete消息。

Heartbeat, Metric, Trace类别的消息操作流程和Event消息基本一样，其中Trace消息需要MessageManager开启TradeMode追踪模式才可以用，类似我们开发中的Debug模式，调用 Cat.getManager().setTraceMode(true) 方法可以开启追踪模式。

## 消息的完成-出栈





所有的消息事务， 都会由 Transaction 对象的 complete 方法完成最终的发送流程。

**[java]** [view plain](https://blog.csdn.net/caohao0591/article/details/80207806" \o "view plain) [copy](https://blog.csdn.net/caohao0591/article/details/80207806" \o "copy)

1. **public** **class** DefaultTransaction **extends** AbstractMessage **implements** Transaction {
2. @Override
3. **public** **void** complete() {
4. **try** {
5. **if** (isCompleted()) {
6. //complete() was called more than once
7. ...
8. } **else** {
9. m\_durationInMicro = (System.nanoTime() - m\_durationStart) / 1000L;
11. setCompleted(**true**);
13. **if** (m\_manager != **null**) {
14. m\_manager.end(**this**);
15. }
16. }
17. } **catch** (Exception e) {
18. // ignore
19. }
20. }
21. }

complete 会首先设置complete标志位，然后交由 m\_manager 去完成消息发送。核心还是调用context的end方法，该方法会从栈顶弹出事务， 如果弹出的事务不等于end方法传入的事务，则认为弹出的事务不是我们需要结束的事务，而是被嵌套的子事务，我们继续弹出下一个栈顶元素，即父事务，直到弹出我们需要结束的事务为止。在这个过程，会调用validate对事务进行校验。

然后我们判断栈是否为空，如果为空，则认为end传入的事务为根事务，这个时候我们才调用 m\_manager.flush 将消息树上报到服务器。

**[java]** [view plain](https://blog.csdn.net/caohao0591/article/details/80207806" \o "view plain) [copy](https://blog.csdn.net/caohao0591/article/details/80207806" \o "copy)

1. **class** context {
2. **public** **boolean** end(DefaultMessageManager manager, Transaction transaction) {
3. **if** (!m\_stack.isEmpty()) {
4. Transaction current = m\_stack.pop();
6. **if** (transaction == current) {
7. m\_validator.validate(m\_stack.isEmpty() ? **null** : m\_stack.peek(), current);
8. } **else** {
9. **while** (transaction != current && !m\_stack.empty()) {
10. m\_validator.validate(m\_stack.peek(), current);
12. current = m\_stack.pop();
13. }
14. }
16. **if** (m\_stack.isEmpty()) {
17. MessageTree tree = m\_tree.copy();
19. m\_tree.setMessageId(**null**);
20. m\_tree.setMessage(**null**);
22. **if** (m\_totalDurationInMicros > 0) {
23. adjustForTruncatedTransaction((Transaction) tree.getMessage());
24. }
26. manager.flush(tree);
27. **return** **true**;
28. }
29. }
31. **return** **false**;
32. }
33. }

## 消息的发送-队列化

在上一章我们知道，MessageManager 会通过 flush 将消息树上报到服务器，我们来通过下面源码分析一下flush方法，函数首先判断是否分配MessageID，没有则分配， 然后调用TcpSocketSender的send函数来发送消息。

send函数也不是立即发送， 仅仅只是插入内存队列。读者可以去看看 TcpSocketSender 的 initialize() 方法， 有行代码 Threads.forGroup("cat").start(this) ，这行代码使得客户端在初始化的时候， 就开启一个上报线程，上报线程一直读取内存队列，获取要发送的消息树，调用 sendInternal(MessageTree tree) 方法将消息树发送到服务器。

这样子，客户端就实现了消息的多线程、异步化、队列化，从而保证日志的记录不会因为CAT系统异常而影响主业务线程。

**[java]** [view plain](https://blog.csdn.net/caohao0591/article/details/80207806" \o "view plain) [copy](https://blog.csdn.net/caohao0591/article/details/80207806" \o "copy)

1. **public** **class** DefaultMessageManager **extends** ContainerHolder **implements** MessageManager, Initializable, LogEnabled {
2. **public** **void** flush(MessageTree tree) {
3. **if** (tree.getMessageId() == **null**) {
4. tree.setMessageId(nextMessageId());
5. }
7. MessageSender sender = m\_transportManager.getSender();
9. **if** (sender != **null** && isMessageEnabled()) {
10. sender.send(tree);
11. reset();
12. } **else** {
13. ...
14. }
15. }
16. }

## 消息的序列化

在上一章节我们说到上报线程通过 sendInternal(MessageTree tree) 将消息发送到服务器，在 sendInternal 方法内， TcpSocketSender 在发送报文之前，会先调用m\_codec.encode(tree, buf) 对消息树进行序列化，序列化就是将对象编码为一组字节，使得对象能够通过 tcp/ip 协议发送到服务器端的技术， 服务器再通过反序列化， 将字节解码为对象。

在Java中，只要一个类实现了java.io.Serializable接口，那么它就可以被序列化。但是通过公共接口编码的字节会有很多冗余信息来保证不同对象与字节之间的正确编解码，在CAT中，需要传输的只有MessageTree这么一个对象。通过自定义的序列化方案可以节省许多不必要的字节信息，保证网络传输的高效性。

**[java]** [view plain](https://blog.csdn.net/caohao0591/article/details/80207806" \o "view plain) [copy](https://blog.csdn.net/caohao0591/article/details/80207806" \o "copy)

1. **public** **class** PlainTextMessageCodec **implements** MessageCodec, LogEnabled {
2. @Override
3. **public** **void** encode(MessageTree tree, ByteBuf buf) **throws** UnsupportedEncodingException {
4. **int** count = 0;
5. **int** index = buf.writerIndex();
7. buf.writeInt(0); // place-holder
9. count += encodeHeader(tree, buf);
11. **if** (tree.getMessage() != **null**) {
12. count += encodeMessage(tree.getMessage(), buf);
13. }
15. buf.setInt(index, count);
16. }
17. }

被序列化的字节码包含3个部分：

1、 前4个字节包含整组字节串的长度，首先通过buf.writeInt(0)占位，编码完通过buf.setInt(index, count)将字节码长度写入buf头4个字节。

2、编码消息树的头部，依次将tree的version, domain, hostName, ipAdress, treadGroupName, treadId, threadName, MessageId, parentMessageId, rootMessageId, sessionToken写入头部，字段之间以"\t"分隔，并以"\n"结尾。空用null表示。

3、编码消息体，每个消息都是以一个表示消息类型的字符开头。  
        a."A"表示没有嵌套其他类型消息的事务，  
        b.有嵌套其他消息的事务，以一个 "t" 开头，然后递归去遍历并编码子消息， 最后以一个"T"结束，  
        c."E"/"L"/"M"/"H"分别表示Event/Trace/Metric/Heartbeat类型消息；  
    然后依次记录时间、type、name  
    然后根据条件依次写入status、duration+us、data  
    字段之间依然以"\t"分割，以"\n"结尾，空用null表示

比如上面其它消息组合章节的案例中，MessageTree通过编码之后：

**[plain]** [view plain](https://blog.csdn.net/caohao0591/article/details/80207806" \o "view plain) [copy](https://blog.csdn.net/caohao0591/article/details/80207806" \o "copy)

1. 口PT1  Cat Win7-caoh.kingsoft.cn   192.168.37.41   main    1   main    Cat-c0a82529-423686-40028   null    null    null
2. t2018-05-02 22:59:05.347    URL WebPage
3. H2018-05-02 22:59:05.353    Heartbeat1  hearbeat    0   cpu=90&mem=70
4. M2018-05-02 22:59:05.353        metric1 0   total\_fee
5. L2018-05-02 22:59:05.354    Trace1  debug   0   user\_debug\_data
6. E2018-05-02 22:59:05.354    Event1  Name1   0   data1
7. E2018-05-02 22:59:05.354    Event2  Name2   0   data2
8. E2018-05-02 22:59:05.354    RemoteCall  Service1    0   Cat-c0a82529-423686-40026
9. T2018-05-02 22:59:07.507    URL WebPage 0   2160695us   k1=v1&k2=v2&k3=v3

上面一串字符串，是通过字节码转换成string的结果， 最前面的乱码，实际上表示的是4个字节的int类型转为string类型表现形式。字节码转int后是541，是整个字节码的长度。

最终TcpSocketSender 通过ChannelManager 将编码后的字节码发送到服务器。这里采用的是netty客户端，不展开来讲了，网上有许多资料可以查阅。

## MessageId的设计

# 第四章 服务端消息分发

## 分发架构

这边首先介绍下消息分发大概的架构如下：

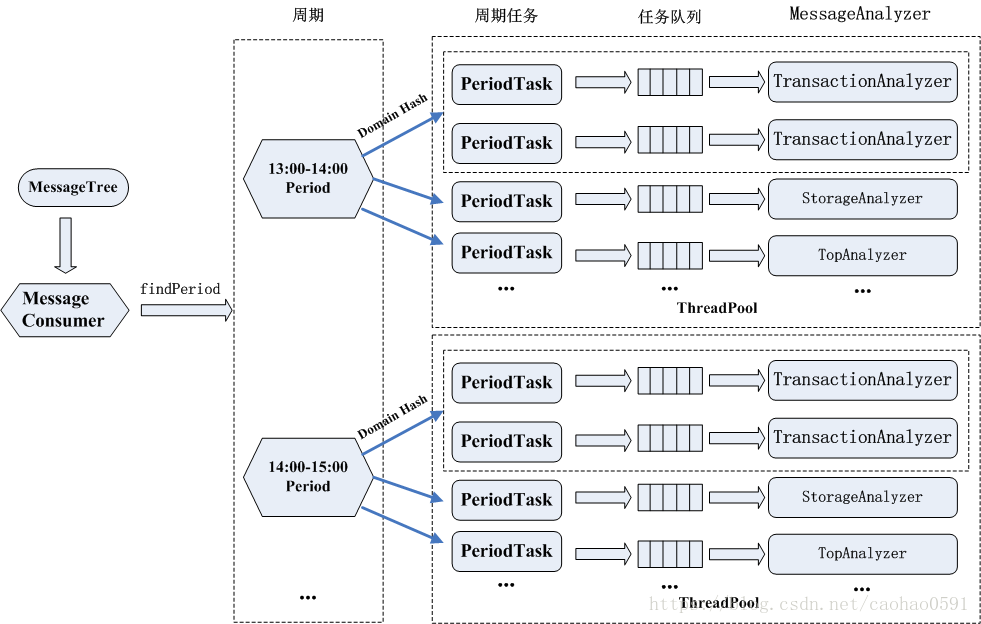
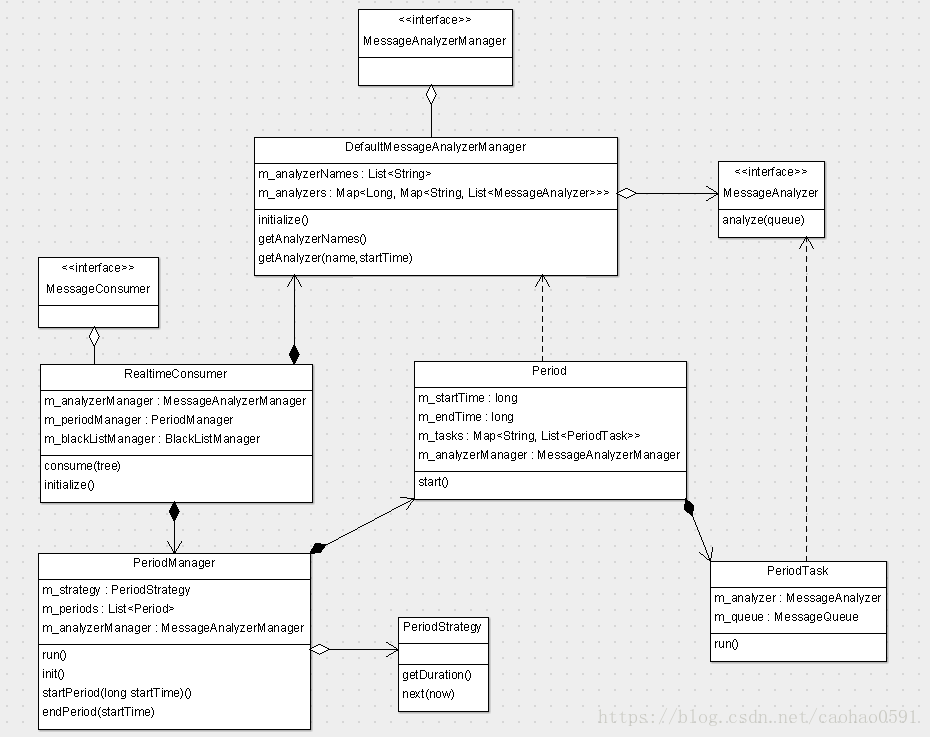


                                                                图4 消息分发架构图

## 分析管理器的初始化



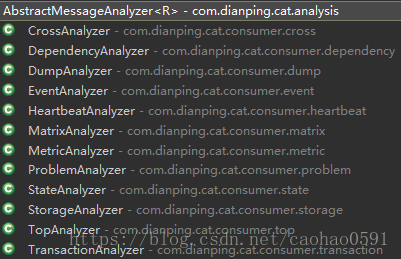
我们在第一章讲到服务器将接收到的消息交给解码器(MessageDecoder)去做解码最后交给具体的消费者(RealtimeConsumer)去消费消息。

RealtimeConsumer 是在什么时候被创建初始化？ 在第一章我们讲到，CatHomeModule通过调用setup安装完成之后，会调用 execute 进行初始化的工作， 在execute方法中有一行代码ctx.lookup(MessageConsumer.class); 就是用来实例化RealtimeConsumer的。MessageConsumer 依赖分析管理器(MessageAnalyzerManager)，所有的分析器(MessageAnalyzer)都由消息分析管理器管理，所以我们又去实例化 MessageAnalyzerManager，然后调用initialize() 方法初始化分析管理器。

**[java]** [view plain](https://blog.csdn.net/caohao0591/article/details/80200769" \o "view plain) [copy](https://blog.csdn.net/caohao0591/article/details/80200769" \o "copy)

1. **public** **class** DefaultMessageAnalyzerManager **extends** ContainerHolder **implements** MessageAnalyzerManager, Initializable, LogEnabled {
2. **private** List<String> m\_analyzerNames;
3. **private** Map<Long, Map<String, List<MessageAnalyzer>>> m\_analyzers = **new** HashMap<Long, Map<String, List<MessageAnalyzer>>>();
5. @Override
6. **public** **void** initialize() **throws** InitializationException {
7. Map<String, MessageAnalyzer> map = lookupMap(MessageAnalyzer.**class**);
8. **for** (MessageAnalyzer analyzer : map.values()) {
9. analyzer.destroy();
10. }
12. m\_analyzerNames = **new** ArrayList<String>(map.keySet());
13. ...
14. }
15. }

initialize() 方法通过IOC容器的lookupMap方法，找到所有的消息分析器。一共12个，如下图，然后取出分析器的名字，放到 m\_analyzerNames 列表里，可以说每个名字对应一类分析器，对消息的具体分析处理逻辑就是由各个分析器(MessageAnalyzer)执行，我们如果有自己的扩展需求,需要对消息做其它处理，也可以添加自己的分析器.



## 消费者与周期管理器的初始化

接着会调用 RealtimeConsumer 的 initialize() 初始化，

**[java]** [view plain](https://blog.csdn.net/caohao0591/article/details/80200769" \o "view plain) [copy](https://blog.csdn.net/caohao0591/article/details/80200769" \o "copy)

1. **public** **class** RealtimeConsumer **extends** ContainerHolder **implements** MessageConsumer, Initializable, LogEnabled {
2. @Inject
3. **private** MessageAnalyzerManager m\_analyzerManager;
5. **private** PeriodManager m\_periodManager;
7. @Override
8. **public** **void** initialize() **throws** InitializationException {
9. m\_periodManager = **new** PeriodManager(HOUR, m\_analyzerManager, m\_serverStateManager, m\_logger);
10. m\_periodManager.init();
12. Threads.forGroup("cat").start(m\_periodManager);
13. }
14. }

RealtimeConsumer初始化的时候创建了一个周期管理器(PeriodManager)，并将分析管理器(MessageAnalyzerManager)的指针传给它，PeriodManager保留分析管理器指针仅仅用于在启动一个Period的时候，将MessageAnalyzerManager的指针传递给Period。

**[java]** [view plain](https://blog.csdn.net/caohao0591/article/details/80200769" \o "view plain) [copy](https://blog.csdn.net/caohao0591/article/details/80200769" \o "copy)

1. **public** **class** PeriodManager **implements** Task {
2. **private** PeriodStrategy m\_strategy;
4. **private** List<Period> m\_periods = **new** ArrayList<Period>();
6. **public** PeriodManager(**long** duration, MessageAnalyzerManager analyzerManager,
7. ServerStatisticManager serverStateManager, Logger logger) {
8. m\_strategy = **new** PeriodStrategy(duration, EXTRATIME, EXTRATIME);
9. m\_active = **true**;
10. m\_analyzerManager = analyzerManager;
11. m\_serverStateManager = serverStateManager;
12. m\_logger = logger;
13. }
15. **public** **void** init() {
16. **long** startTime = m\_strategy.next(System.currentTimeMillis());
18. startPeriod(startTime);
19. }
21. **private** **void** startPeriod(**long** startTime) {
22. **long** endTime = startTime + m\_strategy.getDuration();
23. Period period = **new** Period(startTime, endTime, m\_analyzerManager, m\_serverStateManager, m\_logger);
25. m\_periods.add(period);
26. period.start();
27. }
28. }

PeriodManager会创建一个周期策略对象(PeriodStrategy)，每个周期的开始/结束会参考PeriodStrategy的计算结果，参数duration为每个周期的长度，默认是1个小时，而且周期是整点时段，例如：1:00-2:00, 2:00-3:00。

接下来RealtimeConsumer将会调用 m\_periodManager.init() 启动第一个周期，还是上面代码，我们会计算当前时间所处的周期的开始时间，是当前时间的整点时间，比如现在是 13:50， 那么startTime=13:00，然后entTime=startTime + duration 算得结束时间为 14:00， 然后根据起始结束时间来创建 Period 对象，传入分析器的指针。并将周期对象加入到m\_periods列表交给PeriodManager管理。最后调用period.start 启动第一个周期。

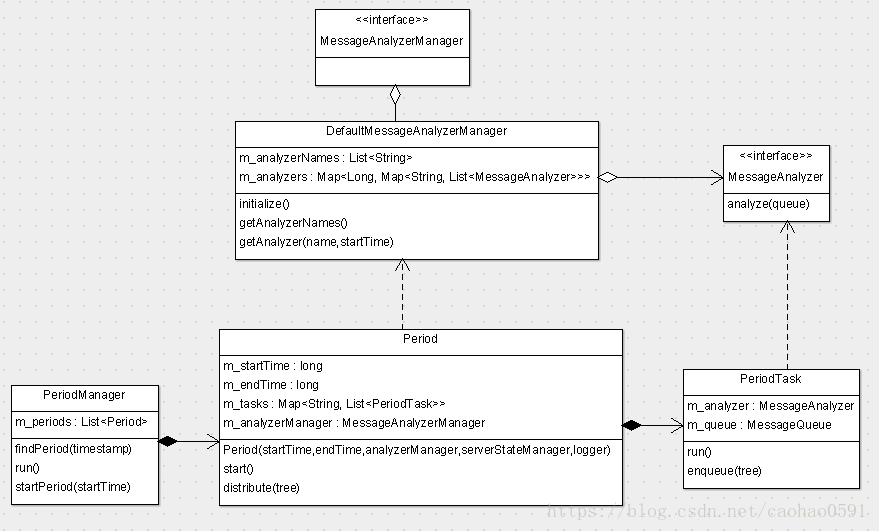
我们再回到ReatimeConsumer的initialize()初始化方法，第三行代码，Threads.forGroup("cat").start(m\_periodManager) 将开启一个周期管理线程，线程执行代码如下run()函数，每隔1秒钟会计算是否需要开启一个新的周期，value>0就开启新的线程， value=0啥也不干，value<0的异步开启一个新线程结束上一个周期。结束线程调用PeriodManager的endPeriod(long startTime)方法完成周期的清理工作，然后将period从m\_periods列表移除出去。

**[java]** [view plain](https://blog.csdn.net/caohao0591/article/details/80200769" \o "view plain) [copy](https://blog.csdn.net/caohao0591/article/details/80200769" \o "copy)

1. **public** **class** PeriodManager **implements** Task {
2. **private** List<Period> m\_periods = **new** ArrayList<Period>();
4. @Override
5. **public** **void** run() {
6. **while** (m\_active) {
7. **try** {
8. **long** now = System.currentTimeMillis();
9. **long** value = m\_strategy.next(now);
11. **if** (value > 0) {
12. startPeriod(value);
13. } **else** **if** (value < 0) {
14. // last period is over,make it asynchronous
15. Threads.forGroup("cat").start(**new** EndTaskThread(-value));
16. }
17. } **catch** (Throwable e) {
18. Cat.logError(e);
19. }
21. Thread.sleep(1000L);
22. }
23. }
25. **private** **void** endPeriod(**long** startTime) {
26. **int** len = m\_periods.size();
28. **for** (**int** i = 0; i < len; i++) {
29. Period period = m\_periods.get(i);
31. **if** (period.isIn(startTime)) {
32. period.finish();
33. m\_periods.remove(i);
34. **break**;
35. }
36. }
37. }
38. }

## 什么是周期？

    好了，我们在上两节讲了分析器的初始化，周期管理器的初始化，那么，什么是周期？为什么会有周期？他是如何工作的？



    可以认为周期Period就是一个消息分发的控制器，相当于MVC的Controller，受PeriodManager的管理，所有客户端过来的消息，都会根据消息时间戳从PeriodManager中找到消息所属的周期Period对象，由该对象来派发消息给每个注册的分析器(MessageAnalyzer)来对消息做具体的处理。

    然而Period并不是直接对接分析器(MessageAnalyzer)， 而是通过PeriodTask来与MessageAnalyzer交互，Period类有个属性Map<String, List<PeriodTask>> m\_tasks， 是一个Map类型的， Map的key是String类型，代表的就是分析器的名字，可以认为每个名字对应一类分析器，我们一共有12种类别的分析器，不过实际分析过程中，CAT作者移除了他认为比较鸡肋的Matrix、Dependency两个分析器，只剩下10个分析器了，如图10。

**[java]** [view plain](https://blog.csdn.net/caohao0591/article/details/80200769" \o "view plain) [copy](https://blog.csdn.net/caohao0591/article/details/80200769" \o "copy)

1. m\_analyzerNames.remove("matrix");
2. m\_analyzerNames.remove("dependency");

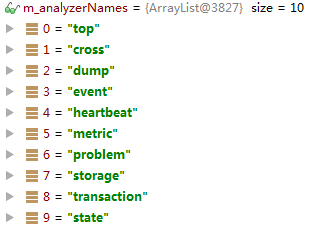


图10：参与任务处理的分析器名称

Map的value为List<PeriodTask> 是一个周期任务的列表， 也就是说，每一种类别的分析器，都会有至少一个MessageAnalyzer的实例，每个MessageAnalyzer都由一个对应的PeriodTask来分配任务，MessageAnalyzer与PeriodTask是1对1的关系，每种类别分析器具体有多少个实例由 getAnalyzerCount() 函数决定，默认是 1 个， 但是有些分析任务非常耗时，需要多个线程来处理，保证处理效率，比如 TransactionAnalyzer就是2个。

**[java]** [view plain](https://blog.csdn.net/caohao0591/article/details/80200769" \o "view plain) [copy](https://blog.csdn.net/caohao0591/article/details/80200769" \o "copy)

1. **public** **class** TransactionAnalyzer **extends** AbstractMessageAnalyzer<TransactionReport> **implements** LogEnabled {
2. @Override
3. **public** **int** getAnalyzerCount() {
4. **return** 2;
5. }
6. }

消息分发的时候，每一笔消息默认都会发送到所有种类分析器处理，但是同一种类别的分析器下如果有多个MessageAnalyzer实例，采用domain hash 选出其中一个实例安排处理消息，分发算法参考下面源码：

**[java]** [view plain](https://blog.csdn.net/caohao0591/article/details/80200769" \o "view plain) [copy](https://blog.csdn.net/caohao0591/article/details/80200769" \o "copy)

1. **public** **class** Period {
2. **private** Map<String, List<PeriodTask>> m\_tasks;
4. **public** **void** distribute(MessageTree tree) {
5. ...
6. String domain = tree.getDomain();
8. **for** (Entry<String, List<PeriodTask>> entry : m\_tasks.entrySet()) {
9. List<PeriodTask> tasks = entry.getValue();
10. **int** length = tasks.size();
11. **int** index = 0;
12. **boolean** manyTasks = length > 1;
14. **if** (manyTasks) {
15. index = Math.abs(domain.hashCode()) % length;
16. }
17. PeriodTask task = tasks.get(index);
18. **boolean** enqueue = task.enqueue(tree);
19. ...
20. }
21. ...
22. }
23. }

## 周期任务-任务队列

上一节我们讲到与MessageAnalyzer交互是由PeriodTask来完成的，那么周期任务PeriodTask在哪里被创建？他怎么与分析器进行交互， 在Period实例化的同时，PeriodTask就被创建了，我们看看Period类的构造函数：

**[java]** [view plain](https://blog.csdn.net/caohao0591/article/details/80200769" \o "view plain) [copy](https://blog.csdn.net/caohao0591/article/details/80200769" \o "copy)

1. **public** **class** Period {
2. **private** Map<String, List<PeriodTask>> m\_tasks;
4. **public** Period(**long** startTime, **long** endTime, MessageAnalyzerManager analyzerManager,
5. ServerStatisticManager serverStateManager, Logger logger) {
7. ...
9. List<String> names = m\_analyzerManager.getAnalyzerNames();
11. m\_tasks = **new** HashMap<String, List<PeriodTask>>();
12. **for** (String name : names) {
13. List<MessageAnalyzer> messageAnalyzers = m\_analyzerManager.getAnalyzer(name, startTime);
15. **for** (MessageAnalyzer analyzer : messageAnalyzers) {
16. MessageQueue queue = **new** DefaultMessageQueue(QUEUE\_SIZE);
17. PeriodTask task = **new** PeriodTask(analyzer, queue, startTime);
19. //加入 m\_tasks
20. ...
21. }
22. }
23. }
24. }

构造函数首先获取所有分析器名字，可以说每个名字对应一类分析器，然后根据分析器名字和周期时间，获取当前周期、该类别分析器所有实例，之前说过，有些类别分析任务逻辑复杂，耗时长，会需要更多的分析线程处理，为每个分析器都创建一个 PeriodTask，并为每一个PeriodTask创建任务队列。客户端消息过来，会由Period分发给所有类别的PeriodTask，同一类分析器下有多个分析器(MessageAnalyzer)的时候，只有一个MessageAnalyzer会被分发，采用domain hash选出这个实例，在这里，分发实际上就是插入PeriodTask的任务队列，最后将创建PeriodTask加入m\_tasks。

在Period被实例化之后， 周期管理器(PeriodManager)就调用 period.start() 开启一个周期了，start逻辑很简单， 就是开启period下所有周期任务(PeriodTask)线程。任务线程也非常简单，就是调用分析器的分析函数analyze(m\_queue)， 并传入任务队列 m\_queue。

**[java]** [view plain](https://blog.csdn.net/caohao0591/article/details/80200769" \o "view plain) [copy](https://blog.csdn.net/caohao0591/article/details/80200769" \o "copy)

1. **public** **class** PeriodTask **implements** Task, LogEnabled {
3. **private** MessageAnalyzer m\_analyzer;
5. **private** MessageQueue m\_queue;
7. @Override
8. **public** **void** run() {
9. **try** {
10. m\_analyzer.analyze(m\_queue);
11. } **catch** (Exception e) {
12. Cat.logError(e);
13. }
14. }
15. }

接下来我们看下分析函数做了什么，下面是源码，只展示了核心逻辑部分，分析程序轮训从队列中取出消息，然后调用process处理，具体的处理逻辑就是由process完成的，process是一个抽象函数，具体实现由各类别分析器子类来实现。

**[java]** [view plain](https://blog.csdn.net/caohao0591/article/details/80200769" \o "view plain) [copy](https://blog.csdn.net/caohao0591/article/details/80200769" \o "copy)



**[java]** [view plain](https://blog.csdn.net/caohao0591/article/details/80200769" \o "view plain) [copy](https://blog.csdn.net/caohao0591/article/details/80200769" \o "copy)

1. **public** **abstract** **class** AbstractMessageAnalyzer<R> **extends** ContainerHolder **implements** MessageAnalyzer {
2. **protected** **abstract** **void** process(MessageTree tree);
4. @Override
5. **public** **void** analyze(MessageQueue queue) {
6. **while** (!isTimeout() && isActive()) {
7. MessageTree tree = queue.poll();
9. **if** (tree != **null**) {
10. ...
11. process(tree);
12. ...
13. }
14. }
15. ...
16. }
17. }

## 消息分发

消息从客户端发上来，是如何到达PeriodTask的，又是如何分配分析器的？

客户端消息发送到服务端，经过解码之后，就调用 MessageConsumer的 consume() 函数对消息进行消费。源码如下：

**[java]** [view plain](https://blog.csdn.net/caohao0591/article/details/80200769" \o "view plain) [copy](https://blog.csdn.net/caohao0591/article/details/80200769" \o "copy)

1. **public** **class** RealtimeConsumer **extends** ContainerHolder **implements** MessageConsumer, Initializable, LogEnabled {
2. @Override
3. **public** **void** consume(MessageTree tree) {
4. String domain = tree.getDomain();
5. String ip = tree.getIpAddress();
7. **if** (!m\_blackListManager.isBlack(domain, ip)) {
8. **long** timestamp = tree.getMessage().getTimestamp();
9. Period period = m\_periodManager.findPeriod(timestamp);
11. **if** (period != **null**) {
12. period.distribute(tree);
13. } **else** {
14. m\_serverStateManager.addNetworkTimeError(1);
15. }
16. } **else** {
17. m\_black++;
19. **if** (m\_black % CatConstants.SUCCESS\_COUNT == 0) {
20. Cat.logEvent("Discard", domain);
21. }
22. }
23. }
24. }

consume函数会首先判断domain和ip是否黑名单，如果是黑名单，丢弃消息，否则，根据消息时间戳，找到对应的周期(Period)，交给Period对消息进行分发，分发逻辑非常简单，period将消息插入PeriodTask队列，由分析器(MessageAnalyzer)轮训从队列里面取消息进行具体处理，每笔消息默认会被所有类别分析器处理，当同一类别分析器有多个MessageAnalyzer实例的时候，选择其中一个处理，选择算法：

**[java]** [view plain](https://blog.csdn.net/caohao0591/article/details/80200769" \o "view plain) [copy](https://blog.csdn.net/caohao0591/article/details/80200769" \o "copy)

1. Math.abs(domain.hashCode()) % length

源代码参考章节什么是周期？

## 周期策略

  在创建周期策略对象的时候，会传入3个参数，一个是duration，也就是每个周期的时间长度，默认为1个小时，另外两个extraTime和aheadTime分别表示我提前启动一个周期的时间和延迟结束一个周期的时间，默认都是3分钟，我们并不会卡在整点时间，例如10:00去开启或结束一个周期，因为周期创建是需要消耗一定时间，这样可以避免消息过来周期对象还未创建好，或者消息还没有处理完，就要去结束周期。

    当然，即使提前创建了周期对象(Period)，并不意味着就会立即被分发消息，只有到了该周期时间才会被分发消息。

    下面看看具体的策略方法，我们首先计算当前时间的周期启动时间(startTime)，是当前时间的整点时间，比如当前时间是 22:47.123，那么startTime就是 22:00.000，注意这里的时间都是时间戳，单位为毫秒。

    接下来判断是否开启当前周期，如果startTime大于上次周期启动时间(m\_lastStartTime)，说明应该开启新的周期，由于m\_lastStartTime初始化为 -1， 所以CAT服务端初始化之后第一个周期会执行到这里，并记录m\_lastStartTime。

    上面if如果未执行，我们会判断当前时间比起上次周期启动时间是不是已经过了 57 分钟(duration - aheadTime )，即提前3分钟启动下一个周期。

    如果上面if还未执行，我们则认为当前周期已经被启动，那么会判断是否需要结束当前周期，即当前时间比起上次周期启动时间是不是已经过了 63 分钟(duration + extraTime)，即延迟3分钟关闭上一个周期。

**[java]** [view plain](https://blog.csdn.net/caohao0591/article/details/80200769" \o "view plain) [copy](https://blog.csdn.net/caohao0591/article/details/80200769" \o "copy)

1. **public** **class** PeriodStrategy {
2. **public** **long** next(**long** now) {
3. **long** startTime = now - now % m\_duration;
5. // for current period
6. **if** (startTime > m\_lastStartTime) {
7. m\_lastStartTime = startTime;
8. **return** startTime;
9. }
11. // prepare next period ahead
12. **if** (now - m\_lastStartTime >= m\_duration - m\_aheadTime) {
13. m\_lastStartTime = startTime + m\_duration;
14. **return** startTime + m\_duration;
15. }
17. // last period is over
18. **if** (now - m\_lastEndTime >= m\_duration + m\_extraTime) {
19. **long** lastEndTime = m\_lastEndTime;
20. m\_lastEndTime = startTime;
21. **return** -lastEndTime;
22. }
24. **return** 0;
25. }
26. }

# 第五章 配置与数据库操作

## CAT配置

在CAT中，有非常多的配置去指导监控的行为，每个配置都有相应的配置管理类来管理，都有一个配置名， 配置在数据库或者配置文件中都是以xml格式存储，在运行时会被解析到具体实体类存储。我们选取一个配置的初始化代码来分析：

**[java]** [view plain](https://blog.csdn.net/caohao0591/article/details/80293833" \o "view plain) [copy](https://blog.csdn.net/caohao0591/article/details/80293833" \o "copy)

1. **public** **class** ServerFilterConfigManager **implements** Initializable {
2. @Inject
3. **protected** ConfigDao m\_configDao;
5. **private** **volatile** ServerFilterConfig m\_config;
7. **private** **static** **final** String CONFIG\_NAME = "serverFilter";
9. @Override
10. **public** **void** initialize() **throws** InitializationException {
11. **try** {
12. Config config = m\_configDao.findByName(CONFIG\_NAME, ConfigEntity.READSET\_FULL);
13. String content = config.getContent();
15. m\_configId = config.getId();
16. m\_modifyTime = config.getModifyDate().getTime();
17. m\_config = DefaultSaxParser.parse(content);
18. } **catch** (DalNotFoundException e) {
19. **try** {
20. String content = m\_fetcher.getConfigContent(CONFIG\_NAME);
21. Config config = m\_configDao.createLocal();
23. config.setName(CONFIG\_NAME);
24. config.setContent(content);
25. m\_configDao.insert(config);
26. m\_configId = config.getId();
27. m\_config = DefaultSaxParser.parse(content);
28. } **catch** (Exception ex) {
29. Cat.logError(ex);
30. }
31. } **catch** (Exception e) {
32. Cat.logError(e);
33. }
34. **if** (m\_config == **null**) {
35. m\_config = **new** ServerFilterConfig();
36. }
37. Threads.forGroup("cat").start(**new** ConfigReloadTask());
38. }
39. }

以上代码是serverFilter配置的初始化逻辑，该配置的管理类是ServerFilterConfigManager，可以看到他实现了Initializable接口，所以，在ServerFilterConfigManager被plexus容器实例化之后，就会调用initialize()做一些初始化的工作，咱们看看他的初始化逻辑。

1、他首先读取cat数据库的config表，如果根据配置名 CONFIG\_NAME 找到相关的配置信息。

2、如果 config 表中存在配置信息，则通过 DefaultSaxParser.parse(content) 方法解析xml配置信息，并将解析后的配置信息写入实体对象ServerFilterConfig m\_config，我们的xml配置文件必须遵守CAT的解析规则，他们才会被正确的解析成对应的XxxConfig对象，具体配置规则大家在有需要的时候一步步摸索。

3、如果 config 表中不存在配置信息，就会去读取默认 xml 文件， 解析后写入到数据库和实体对象，下次再初始化的时候就是直接从数据库读取了，xml一般存在于cat-core/src/main/resources/config/ 和 cat-home/src/main/resources/config/ 两个目录。

4、其中有两个配置 serverFilter 和 aggreationConfig， 开启了检查更新线程，线程每隔1分钟检查配置是否更新，有更新就重新加载配置。

5、server.xml 配置文件的目录是由环境变量 CAT\_HOME指定的，由CatHomeModule在setup安装的时候初始化。

我们来列一列cat有哪些配置

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 配置名 | 配置管理类 | 实体类 | 配置文件cat-xxxx/src/main/resources/config/\*.xml | 用途 |
| ServerConfigManager | ServerConfigManager | ServerConfig | 环境变量 CAT\_HOME/server.xml | 未知 |
| serverFilter | ServerFilterConfigManager | ServerFilterConfig | cat-core/..../serverFilter.xml | 未知 |
| all-report-config | AllReportConfigManager | AllConfig | cat-home/..../all-report-config.xml | 未知 |
| metricProductLine | ProductLineConfigManager | ProductLineConfig | cat-home/..../metricProductLine.xml | 未知 |
| userProductLine | ProductLineConfigManager | ProductLineConfig | cat-home/..../userProductLine.xml | 未知 |
| applicationProductLine | ProductLineConfigManager | ProductLineConfig | cat-home/..../applicationProductLine.xml | 未知 |
| networkProductLine | ProductLineConfigManager | ProductLineConfig | cat-home/..../networkProductLine.xml | 未知 |
| systemProductLine | ProductLineConfigManager | ProductLineConfig | cat-home/..../systemProductLine.xml | 未知 |
| databaseProductLine | ProductLineConfigManager | ProductLineConfig | cat-home/..../databaseProductLine.xml | 未知 |
| cdnProductLine | ProductLineConfigManager | ProductLineConfig | cat-home/..../cdnProductLine.xml | 未知 |
| metricConfig | MetricConfigManager | MetricConfig | cat-home/..../metricConfig.xml | 未知 |
| blackList | BlackListManager | BlackList | cat-home/..../blackList.xml | 未知 |
| routerConfig | RouterConfigManager | RouterConfig | cat-home/..../routerConfig.xml | 未知 |
| domainGroup | DomainGroupConfigManager | DomainGroup | cat-home/..../domainGroup.xml | 未知 |
| heartbeat-display-policy | HeartbeatDisplayPolicyManager | HeartbeatDisplayPolicy | cat-home/..../heartbeat-display-policy.xml | 未知 |
| topologyConfig | TopologyGraphConfigManager | TopologyGraphConfig | cat-home/..../topologyConfig.xml | 未知 |
| topoGraphFormat | TopoGraphFormatConfigManager | TopoGraphFormatConfig | cat-home/..../topoGraphFormat.xml | 未知 |
| bugConfig | BugConfigManager | BugConfig | cat-home/..../bugConfig.xml | 未知 |
| senderConfig | SenderConfigManager | SenderConfig | cat-home/..../senderConfig.xml | 未知 |
| netGraphConfig | NetGraphConfigManager | NetGraphSet | cat-home/..../netGraphConfig.xml | 未知 |
| url-pattern | UrlPatternConfigManager | UrlPattern | cat-home/..../url-pattern.xml | 未知 |
| app-config | AppConfigManager | AppConfig | cat-home/..../appConfig.xml | 未知 |
| app-speed-config | AppSpeedConfigManager | AppSpeedConfig | cat-home/..../app-speed-config.xml | 未知 |
| appRule | AppRuleConfigManager | MonitorRules | cat-home/..../appRule.xml | 未知 |
| storageGroup | StorageGroupConfigManager | StorageGroupConfig | cat-home/..../storageGroup.xml | 未知 |
| activityConfig | ActivityConfigManager | ActivityConfig | cat-home/..../activityConfig.xml | 未知 |
| exceptionRuleConfig | ExceptionRuleConfigManager | ExceptionRuleConfig | cat-home/..../exceptionRuleConfig.xml | 未知 |
| thirdPartyConfig | ThirdPartyConfigManager | ThirdPartyConfig | cat-home/..../thirdPartyConfig.xml | 未知 |
| aggreationConfig | AggregationConfigManager | Aggregation | cat-home/..../aggreationConfig.xml | 未知 |
| webRule | WebRuleConfigManager | MonitorRules | cat-home/..../webRule.xml | 未知 |
| businessRuleConfig | BusinessRuleConfigManager | MonitorRules | cat-home/..../businessRuleConfig.xml | 未知 |
| networkRuleConfig | NetworkRuleConfigManager | MonitorRules | cat-home/..../networkRuleConfig.xml | 未知 |
| databaseRuleConfig | DatabaseRuleConfigManager | MonitorRules | cat-home/..../databaseRuleConfig.xml | 未知 |
| systemRuleConfig | SystemRuleConfigManager | MonitorRules | cat-home/..../systemRuleConfig.xml | 未知 |
| heartbeatRuleConfig | HeartbeatRuleConfigManager | MonitorRules | cat-home/..../heartbeatRuleConfig.xml | 未知 |
| appComparisonConfig | AppComparisonConfigManager | AppComparisonConfig | cat-home/..../appComparisonConfig.xml | 未知 |
| app-command-format-config | CommandFormatConfigManager | CommandFormat | cat-home/..../app-command-format-config.xml | 未知 |
| alertConfig | AlertConfigManager | AlertConfig | cat-home/..../alertconfig.xml | 未知 |
| alertPolicy | AlertPolicyManager | AlertPolicy | cat-home/..../alertPolicy.xml | 未知 |
| transactionRule | TransactionRuleConfigManager | MonitorRules | cat-home/..../transactionRule.xml | 未知 |
| eventRule | EventRuleConfigManager | MonitorRules | cat-home/..../eventRule.xml | 未知 |
| storageSQLRule | StorageRuleConfigManager | MonitorRules | cat-home/..../storageSQLRule.xml | 未知 |
| storageCacheRule | StorageRuleConfigManager | MonitorRules | cat-home/..../storageCacheRule.xml | 未知 |

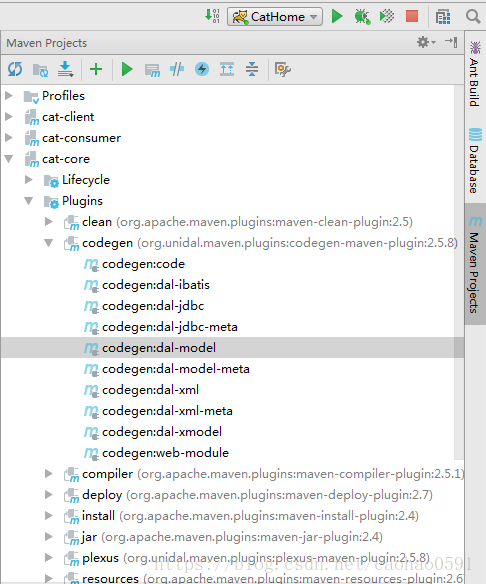
## 代码自动生成

    上一节 ServerFilterConfigManager 类代码中的配置实体ServerFilterConfig的定义为 public class ServerFilterConfig extends BaseEntity<ServerFilterConfig>，所有的实体类都继承自BaseEntity<T>，这些实体类代码不是自己写的，而是通过插件生成的，插件在cat-home/pom.xml，cat-core/pom.xml，cat-consumer/pom.xml 中都会有定义，用以添加需要的实体，包括后面数据库的操作相关的实体，也是通过类似方式生成，这有点类似mybatis的数据库表对象生成器generator，我们来看看cat-core中的一个插件的定义：

**[html]** [view plain](https://blog.csdn.net/caohao0591/article/details/80293833" \o "view plain) [copy](https://blog.csdn.net/caohao0591/article/details/80293833" \o "copy)

1. **<plugins>**
2. **<plugin>**
3. **<groupId>**org.unidal.maven.plugins**</groupId>**
4. **<artifactId>**codegen-maven-plugin**</artifactId>**
5. **<executions>**
6. **<execution>**
7. **<id>**generate data model**</id>**
8. **<phase>**generate-sources**</phase>**
9. **<goals>**
10. **<goal>**dal-model**</goal>**
11. **</goals>**
12. **<configuration>**
13. **<manifest>**${basedir}/src/main/resources/META-INF/dal/model/server-manifest.xml,
14. ${basedir}/src/main/resources/META-INF/dal/model/command-format-manifest.xml,
15. ...
16. ${basedir}/src/main/resources/META-INF/dal/model/server-filter-config-manifest.xml,
17. **</manifest>**
18. **</configuration>**
19. **</execution>**
20. **<execution>**
21. **<id>**generate dal jdbc model**</id>**
22. **<phase>**generate-sources**</phase>**
23. **<goals>**
24. **<goal>**dal-jdbc**</goal>**
25. **</goals>**
26. **<configuration>**
27. **<manifest>**${basedir}/src/main/resources/META-INF/dal/jdbc/report-manifest.xml,
28. ${basedir}/src/main/resources/META-INF/dal/jdbc/config-manifest.xml,
29. ${basedir}/src/main/resources/META-INF/dal/jdbc/app-manifest.xml, <![CDATA[,
30. ${basedir}/src/main/resources/META-INF/dal/jdbc/report-manifest.xml,
31. ${basedir}/src/main/resources/META-INF/dal/jdbc/config-manifest.xml,
32. ,]]>**</manifest>**
33. **</configuration>**
34. **</execution>**
35. **</executions>**
36. **</plugin>**
37. **<plugins>**

插件在pom.xml中定义后，会在Maven Projects功能中展现出来，如图5：

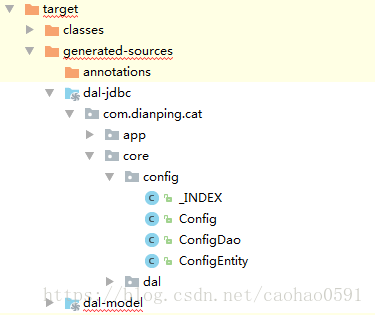


    插件会根据xml中的描述，生成相应的类，比如 ServiceFilterConfig实体类的描述文档为： ${basedir}/src/main/resources/META-INF/dal/model/server-filter-config-manifest.xml，这个文档是个父文档，它将引入文档server-filter-config-codegen.xml和server-filter-config-model.xml，这两个文档有对将要生成的类的属性、方法的描述。

    在下一章节的消息分析器中，和报表相关的非常多的实体类 EventReport、EventType、EventName、TransactionReport、TransactionType、TransactionName、Machine等等，都是由CAT代码生成插件根据配置 cat-sonsumer/src/main/resources/META-INFO/dal/model/\*.xml 自动生成的，我们可以通过修改xml文件为我们的报表添加新的属性。

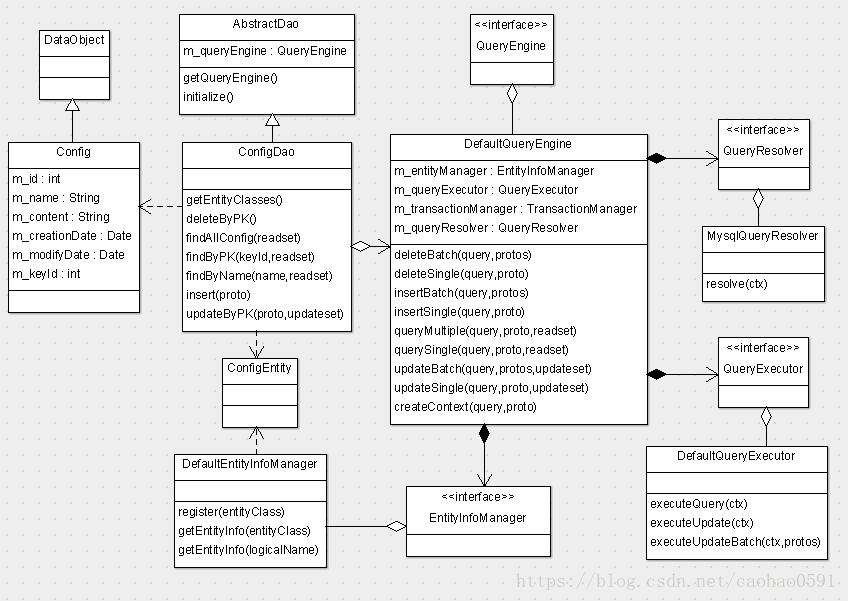
## 数据库操作

    在上一节插件中，还有一个是与数据库操作相关类的生成有关的，比如上边config表相关数据库操作类的描述文档为${basedir}/src/main/resources/META-INF/dal/jdbc/config-manifest.xml，插件会根据xml文档生成Config、ConfigDao、ConfigEntity三个类放在下图目录中：



    那么config表有什么作用呢？这三个类分别有什么用途？在上一节我们多次用到这张表，例如我们初始化配置的时候，读取config表的记录，如果没有初始数据的话，配置管理类会从xml读取配置信息并插入config表了。

    现在我们来讲一讲cat的数据库操作，先看看数据库操作的类图：



    对数据库表的操作位于Dao类中，比如Config表的操作类就是 ConfigDao，所有Dao继承自AbstractDao，每个Dao都会拥有数据库执行引擎(QueryEngine)的指针，用以执行所有的SQL语句，QueryEngine在Dao初始化函数initialize()中被实例化，

    每个执行引擎QueryEngine都拥有实体信息管理(EntityInfoManager)，以及查询解析器(QueryResolver)，查询执行器(QueryExecutor)，事务管理器(TransactionManager)，

    实体信息管理类(EntityInfoManager)管理每个表的Entity信息，Entity中描述对数据库的具体操作，Config表的实体对象就是ConfigEntity，他会在AbstractDao.initialize()函数中向EntityInfoManager注册。

    另外还有一个关键的类，就是Config类，继承自DataObject，是对config表的所有字段的描述。

我们来看几个数据库操作的案例，下面源码，

**[html]** [view plain](https://blog.csdn.net/caohao0591/article/details/80293833" \o "view plain) [copy](https://blog.csdn.net/caohao0591/article/details/80293833" \o "copy)

1. public class ConfigDao extends AbstractDao {
2. public List**<Config>** findAllConfig(Readset**<Config>** readset) throws DalException {
3. Config proto = new Config();
5. List**<Config>** result = getQueryEngine().queryMultiple(
6. ConfigEntity.FIND\_ALL\_CONFIG,
7. proto,
8. readset);
10. return result;
11. }
13. public Config findByName(String name, Readset**<Config>** readset) throws DalException {
14. Config proto = new Config();
16. proto.setName(name);
18. Config result = getQueryEngine().querySingle(
19. ConfigEntity.FIND\_BY\_NAME,
20. proto,
21. readset);
23. return result;
24. }
26. public int insert(Config proto) throws DalException {
27. return getQueryEngine().insertSingle(
28. ConfigEntity.INSERT,
29. proto);
30. }
32. @Override
33. protected Class**<?>**[] getEntityClasses() {
34. return new Class**<?>**[] { ConfigEntity.class };
35. }
36. }

函数功能：

1、findAllConfig函数是获取所有的配置，返回的对象，是Config的List列表，调用查询引擎QueryEngine的queryMultiple 查询多条记录方法，传入的参数分别是实体操作方法ConfigEntity.FIND\_ALL\_CONFIG、查询条件数据proto、readset。

2、findByName函数则是按名字查询Config配置，逻辑跟上一个类似。

3、insert函数是插入一条config记录，调用查询引擎的insertSingle方法，参数分别是实体方法ConfigEntity.INSERT， 插入数据 proto。

4、getEntityClasses函数获取实体类

接下来我们以查询单条记录作为例子来看看查询引擎的实现逻辑，如下源码：

**[html]** [view plain](https://blog.csdn.net/caohao0591/article/details/80293833" \o "view plain) [copy](https://blog.csdn.net/caohao0591/article/details/80293833" \o "copy)

1. public class DefaultQueryEngine extends ContainerHolder implements QueryEngine {
2. public **<T** extends DataObject**>** T querySingle(QueryDef query, T proto, Readset**<?>** readset) throws DalException {
3. QueryContext ctx = this.createContext(query, proto);
4. ctx.setReadset(readset);
5. ctx.setFetchSize(1);
6. this.m\_queryResolver.resolve(ctx);
7. List**<T>** results = this.m\_queryExecutor.executeQuery(ctx);
8. if(results.isEmpty()) {
9. throw new DalNotFoundException("No record has been found for " + proto);
10. } else {
11. return (DataObject)results.get(0);
12. }
13. }
15. protected **<T** extends DataObject**>** QueryContext createContext(QueryDef query, T proto) {
16. QueryContext ctx = new DefaultQueryContext();
17. EntityInfo enityInfo = this.m\_entityManager.getEntityInfo(query.getEntityClass());
18. Map**<String**, Object**>** queryHints = this.getQueryHints(query, proto);
19. ctx.setQuery(query);
20. ctx.setProto(proto);
21. ctx.setEntityInfo(enityInfo);
22. ctx.setQueryHints(queryHints);
23. return ctx;
24. }
25. }

1、数据库引擎首先为该查询和对应条件数据proto创建查询上下文(QueryContext)，

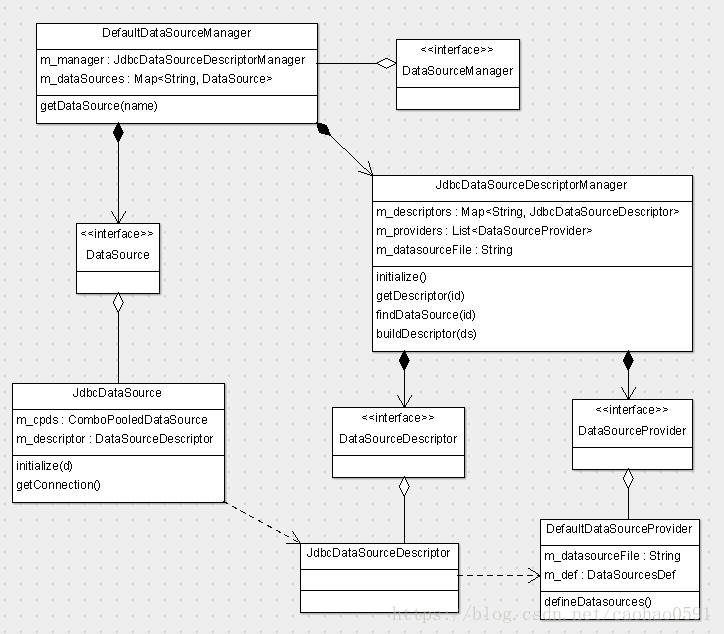
2、然后会设置readset，并设置查询条数为1条，

3、然后将上下文交给查询解析器(QueryResolver)，解析器会根据查询上下文解析成对应的sql状态语，复制到上下文的 m\_sqlStatement字段

4、最后将上下文交给执行器QueryExecutor去执行。并返回结果，如果没有找到结果，引擎会抛出DalNotFoundException异常。

## 数据库连接管理

数据库记录的更新与插入最终会由 MysqlWriteHandler 执行，而查询则由 MysqlReadHandler执行，两个Handler都会通过TransactionManager 的 getConnection 函数获取数据库的连接。



数据库连接是由DataSource管理，DataSource是由数据源管理器(DataSourceManager)管理，TransactionManager 拥有DataSourceManager的指针，DataSourceManager 提供函数getDataSource(String name)获取DataSource对象；

getDataSource函数首先会从DataSourceManager 自身 m\_dataSources 中寻找指定数据源，如果没有找到，再从数据源描述管理器(JdbcDataSourceDescriptorManager) 中获取数据库数据源描述信息(JdbcDataSourceDescriptor)，然后利用描述信息去初始化连接池。

描述管理器也是先从自身 m\_descriptors 中看是否存在指定描述信息(JdbcDataSourceDescriptor)，如果没有，则会通过 DataSourceProviver 生成描述信息。

DataSourceProviver可以认为是初始数据源描述信息(JdbcDataSourceDescriptor)的提供者，JdbcDataSourceDescriptorManager初始化的时候，Plexus容器会基于xml文档实例化 DataSourceProviver，这些xml文档有 cat-home/....../META-INF/plexus/components-cat-home.xml，cat-consumer/....../META-INF/plexus/components-cat-consumer.xml， cat-core/....../META-INF/plexus/components-cat-core.xml， 文档中的 datasourceFile 就是数据源配置路径，例如下面xml文档：

**[html]** [view plain](https://blog.csdn.net/caohao0591/article/details/80293833" \o "view plain) [copy](https://blog.csdn.net/caohao0591/article/details/80293833" \o "copy)

1. **<component>**
2. **<role>**org.unidal.dal.jdbc.datasource.DataSourceProvider**</role>**
3. **<implementation>**org.unidal.dal.jdbc.datasource.DefaultDataSourceProvider**</implementation>**
4. **<configuration>**
5. **<datasourceFile>**/data/appdatas/cat/datasources.xml**</datasourceFile>**
6. **</configuration>**
7. **</component>**

JdbcDataSourceDescriptorManager是如何通过DataSourceProviver获取数据源描述信息(JdbcDataSourceDescriptor)？实际上是通过DataSourceProviver读取并解析datasourceFile文件，例如上面的/data/appdatas/cat/datasources.xml，然后将解析后的数据写入 DataSourceDef 对象，然后JdbcDataSourceDescriptorManager调用 buildDescriptor(DataSourceDef ds) 创建描述信息，下面是buildDescriptor的源码：

**[html]** [view plain](https://blog.csdn.net/caohao0591/article/details/80293833" \o "view plain) [copy](https://blog.csdn.net/caohao0591/article/details/80293833" \o "copy)

1. public class JdbcDataSourceDescriptorManager extends ContainerHolder implements Initializable {
2. private Map**<String**, JdbcDataSourceDescriptor**>** m\_descriptors = new HashMap();
3. private List**<DataSourceProvider>** m\_providers;
4. private String m\_datasourceFile;
6. protected JdbcDataSourceDescriptor buildDescriptor(DataSourceDef ds) {
7. JdbcDataSourceDescriptor d = new JdbcDataSourceDescriptor();
8. PropertiesDef properties = ds.getProperties();
9. String url = properties.getUrl();
10. String connectionProperties = properties.getConnectionProperties();
11. if(connectionProperties != null && connectionProperties.length() **>** 0) {
12. d.setProperty("url", url + "?" + connectionProperties);
13. } else {
14. d.setProperty("url", url);
15. }
17. d.setId(ds.getId());
18. d.setType(ds.getType());
19. d.setProperty("driver", properties.getDriver());
20. d.setProperty("user", properties.getUser());
21. d.setProperty("password", properties.getPassword());
22. d.setProperty("login-timeout", Integer.valueOf(this.toTime(ds.getConnectionTimeout())));
23. d.setProperty("max-idle-time", Integer.valueOf(this.toTime(ds.getIdleTimeout())));
24. d.setProperty("min-pool-size", ds.getMinimumPoolSize());
25. d.setProperty("max-pool-size", ds.getMaximumPoolSize());
26. d.setProperty("checkout-timeout", ds.getCheckoutTimeoutInMillis());
27. return d;
28. }
29. }

数据源描述信息包含哪些内容呢，我们结合上面源码和下面的 datasources.xml配置来看，

每个data-source都有一个id； type默认jdbc； driver 默认为 com.mysql.jdbc.Driver、url、用户名、密码，登陆超时、连接池大小、超时检测等等。

**[html]** [view plain](https://blog.csdn.net/caohao0591/article/details/80293833" \o "view plain) [copy](https://blog.csdn.net/caohao0591/article/details/80293833" \o "copy)

1. **<data-source** id="cat"**>**
2. **<maximum-pool-size>**3**</maximum-pool-size>**
3. **<connection-timeout>**1s**</connection-timeout>**
4. **<idle-timeout>**10m**</idle-timeout>**
5. **<statement-cache-size>**1000**</statement-cache-size>**
6. **<properties>**
7. **<driver>**com.mysql.jdbc.Driver**</driver>**
8. **<url>**<![CDATA[jdbc:mysql://192.168.20.67:3306/cat]]>**</url>**
9. **<user>**root**</user>**
10. **<password>**f63hiccVEv0mMXi**</password>**
11. **<connectionProperties>**<![CDATA[useUnicode=true&autoReconnect=true]]>**</connectionProperties>**
12. **</properties>**
13. **</data-source>**

# 第六章 消息分析器与报表(一)

对于消息的具体处理，是由消息分析器完成的，消息分析器会轮训读取PeriodTask中队列的消息来处理，一共有12类消息分析器，处理后的结果就是生成各类报表。

## 消息分析器的构建

在周期Period构造函数中，我们会通过m\_analyzerManager.getAnalyzer(name, startTime)获取分析器(MessageAnalyzer)列表。getAnalyzer函数源码如下，首先会清理2小时之前的分析器，然后从m\_analyzers中获取分析器(MessageAnalyzer)，我们先来看看m\_analyzers 的结构

Map<Long, Map<String, List<MessageAnalyzer>>>

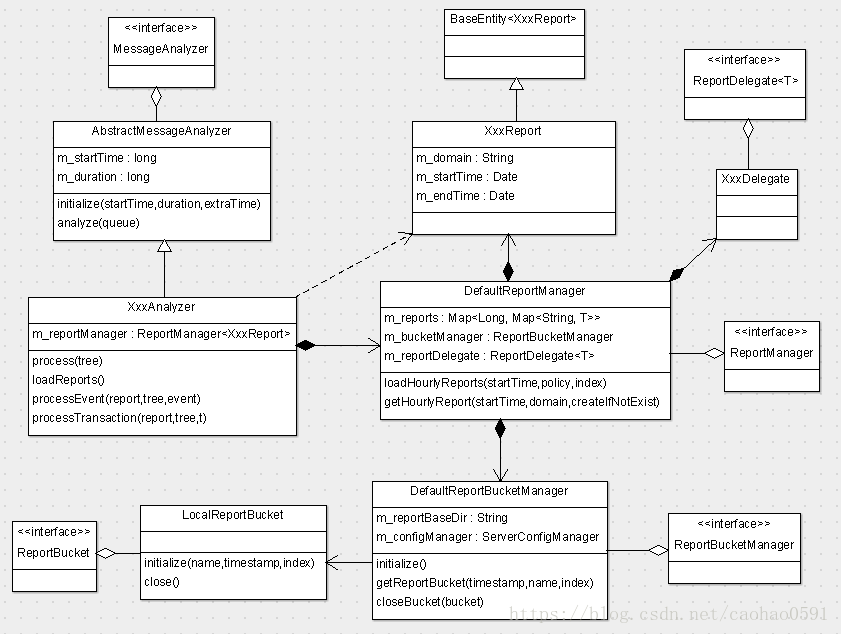
最外层Map的key的类型为long，代表由startTime对应的周期。value还是一个Map，Map的key类型是String，是分析器的名字，代表一类分析器，value是MessageAnalyzer列表，同一类分析器，至少有一个MessageAnalyzer实例，对于复杂耗时的分析任务，我们通常会开启更多的实例处理。

如果在Map中没有找到我们需要的分析器，我们就创建，创建的过程函数会通过synchronized给map上锁，以保证创建过程map同时只能被一个线程访问，保证了线程安全。  
分析器创建之后会被初始化，然后放入m\_analyzers中，

**[java]** [view plain](https://blog.csdn.net/caohao0591/article/details/80231519" \o "view plain) [copy](https://blog.csdn.net/caohao0591/article/details/80231519" \o "copy)

1. **public** **class** DefaultMessageAnalyzerManager **extends** ContainerHolder **implements** MessageAnalyzerManager, Initializable,LogEnabled {
3. **private** Map<Long, Map<String, List<MessageAnalyzer>>> m\_analyzers = **new** HashMap<Long, Map<String, List<MessageAnalyzer>>>();
5. @Override
6. **public** List<MessageAnalyzer> getAnalyzer(String name, **long** startTime) {
7. // remove last two hour analyzer
8. Map<String, List<MessageAnalyzer>> temp = m\_analyzers.remove(startTime - m\_duration \* 2);
10. ...
12. Map<String, List<MessageAnalyzer>> map = m\_analyzers.get(startTime);
14. **if** (map == **null**) {
15. **synchronized** (m\_analyzers) {
16. map = m\_analyzers.get(startTime);
18. **if** (map == **null**) {
19. map = **new** HashMap<String, List<MessageAnalyzer>>();
20. m\_analyzers.put(startTime, map);
21. }
22. }
23. }
25. List<MessageAnalyzer> analyzers = map.get(name);
27. **if** (analyzers == **null**) {
28. **synchronized** (map) {
29. analyzers = map.get(name);
31. **if** (analyzers == **null**) {
32. analyzers = **new** ArrayList<MessageAnalyzer>();
34. MessageAnalyzer analyzer = lookup(MessageAnalyzer.**class**, name);
36. analyzer.setIndex(0);
37. analyzer.initialize(startTime, m\_duration, m\_extraTime);
38. analyzers.add(analyzer);
40. **int** count = analyzer.getAnalyzerCount();
42. **for** (**int** i = 1; i < count; i++) {
43. MessageAnalyzer tempAnalyzer = lookup(MessageAnalyzer.**class**, name);
45. tempAnalyzer.setIndex(i);
46. tempAnalyzer.initialize(startTime, m\_duration, m\_extraTime);
47. analyzers.add(tempAnalyzer);
48. }
49. map.put(name, analyzers);
50. }
51. }
52. }
54. **return** analyzers;
55. }
56. }

我们再来看看分析器的大体结构



每个分析器都包含有多个报表，报表交由报表管理器(ReportManage)管理，报表在报表管理器中存储结构如下：

Map<Long, Map<String, T>> m\_reports

最外层是个Map， key 为long类型，代表的是当前时间周期的报表，value还是一个Map，key类型为String，代表的是不同的domain，一个domain可以理解为一个 Project，value是不同report对象，在分析器处理报表的时候，我们会通过周期管理器(DefaultReportManage)的getHourlyReport方法根据周期时间和domain获取对应的Report。

分析器分析上报的消息之后，生成相应的报表存于Report对象中，报表实体类XxxReport的结构都是由上一章讲的代码自动生成器生成的，配置位于  cat-sonsumer/src/main/resources/META-INFO/dal/model/\*.xml 中。

## TopAnalyzer

TopAnalyzer分析生成每个周期的报表，不区分domain，所有domain的数据都会汇总到所在周期的domain='cat'的这个报表下去：

**[html]** [view plain](https://blog.csdn.net/caohao0591/article/details/80231519" \o "view plain) [copy](https://blog.csdn.net/caohao0591/article/details/80231519" \o "copy)

1. getHourlyReport(getStartTime(), Constants.CAT, true);

TopAnalyzer会处理指定Type类型的Event消息，具体有哪些类型会被处理会在 plexus/components-cat-consumer.xml 文件中配置。如下

**[html]** [view plain](https://blog.csdn.net/caohao0591/article/details/80231519" \o "view plain) [copy](https://blog.csdn.net/caohao0591/article/details/80231519" \o "copy)

1. **<implementation>**com.dianping.cat.consumer.top.TopAnalyzer**</implementation>**
2. **<instantiation-strategy>**per-lookup**</instantiation-strategy>**
3. **<configuration>**
4. **<errorType>**Error,RuntimeException,Exception**</errorType>**
5. **</configuration>**

再来看看TopAnalyzer对Event的处理过程，他会统计当前小时周期内上面类型消息的3个计数。

1、当前小时周期内每分钟，每个domain，也就是每个project的错误计数

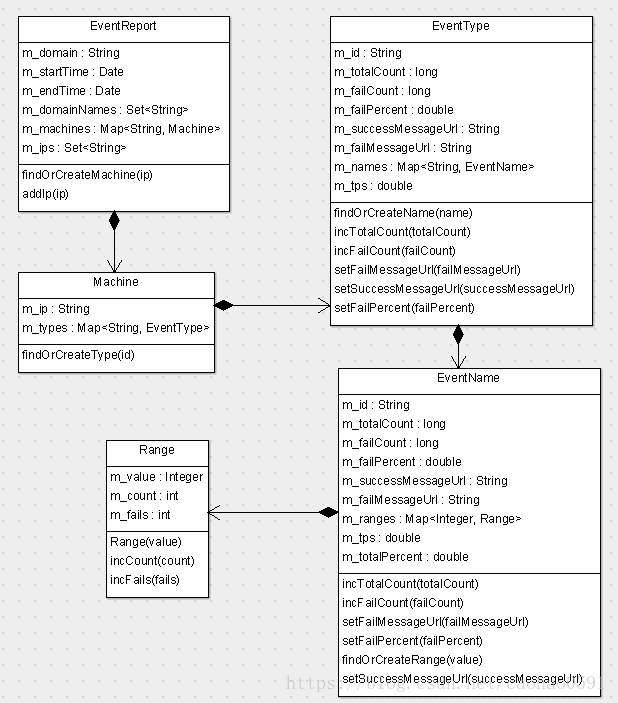
2、每个名字对应的错误计数

3、每个IP对应的错误计数

**[html]** [view plain](https://blog.csdn.net/caohao0591/article/details/80231519" \o "view plain) [copy](https://blog.csdn.net/caohao0591/article/details/80231519" \o "copy)

1. public class TopAnalyzer extends AbstractMessageAnalyzer**<TopReport>** implements LogEnabled {
2. private void processEvent(TopReport report, MessageTree tree, Event event) {
3. String type = event.getType();
5. if (m\_errorTypes.contains(type)) {
6. String domain = tree.getDomain();
7. String ip = tree.getIpAddress();
8. String exception = event.getName();
9. long current = event.getTimestamp() / 1000 / 60;
10. int min = (int) (current % (60));
11. Segment segment = report.findOrCreateDomain(domain).findOrCreateSegment(min).incError();
13. segment.findOrCreateError(exception).incCount();
14. segment.findOrCreateMachine(ip).incCount();
15. }
16. }
17. }

## EventAnalyzer - 事件发生次数分析



    时间分析分析报表会记录Event类型消息的统计汇总信息，每个周期时间，每个domain对应一个EventReport，每个Event报表包含多个Machine对象，按IP区分，注意一下这里的Machine类与后面其它报表的Machine类是有区别的，相同IP下不同类型(Type)的Event信息存在于不同的EventType对象中，EventType记录了该类型消息的总数，失败总数，失败百分比，成功的MessageID，失败的MessageID，tps，以及该类型下各种命名消息。

    同一类型但是不同名字(Name)的Event信息存在于不同的EventName对象中，他也会记录该命名消息的总数，失败总数，失败百分比，成功的MessageID，失败的MessageID，tps。

    每个EventName对象会存储当前周期时间内，不同类型不同名字的Event消息每分钟的消息总数和失败总数，放在m\_ranges字段中。

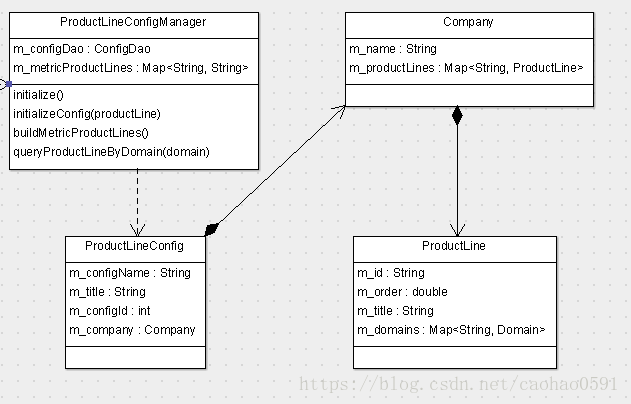
   大家可能疑惑为什么存的成功与失败的MessageID只有一个，而不是一个列表，因为Event报表仅仅只是统计一类事件发生的次数，同类消息做的事情本质上是一样的，所以仅取一条MessageID对应的消息树作为这一类消息的代表。

**[html]** [view plain](https://blog.csdn.net/caohao0591/article/details/80231519" \o "view plain) [copy](https://blog.csdn.net/caohao0591/article/details/80231519" \o "copy)

1. public class EventAnalyzer extends AbstractMessageAnalyzer**<EventReport>** implements LogEnabled {
2. private void processEvent(EventReport report, MessageTree tree, Event event, String ip) {
3. int count = 1;
4. EventType type = report.findOrCreateMachine(ip).findOrCreateType(event.getType());
5. EventName name = type.findOrCreateName(event.getName());
6. String messageId = tree.getMessageId();
8. report.addIp(tree.getIpAddress());
9. type.incTotalCount(count);
10. name.incTotalCount(count);
12. if (event.isSuccess()) {
13. type.setSuccessMessageUrl(messageId);
14. name.setSuccessMessageUrl(messageId);
15. } else {
16. type.incFailCount(count);
17. name.incFailCount(count);
19. type.setFailMessageUrl(messageId);
20. name.setFailMessageUrl(messageId);
21. }
22. type.setFailPercent(type.getFailCount() \* 100.0 / type.getTotalCount());
23. name.setFailPercent(name.getFailCount() \* 100.0 / name.getTotalCount());
25. processEventGrpah(name, event, count);
26. }
28. private void processEventGrpah(EventName name, Event t, int count) {
29. long current = t.getTimestamp() / 1000 / 60;
30. int min = (int) (current % (60));
31. Range range = name.findOrCreateRange(min);
33. range.incCount(count);
34. if (!t.isSuccess()) {
35. range.incFails(count);
36. }
37. }
38. }

## MetricAnalyzer - 业务分析

Metric主要监控业务系统，业务指标，在讲Metric业务报表之前，我们首先讲一下产品线的概念，先看下面类图：



metricProductLine-业务监控是需要配置产品线的，产品线可以认为是一系列project的集合，我们之前说过，每个domain可以认为是一个project，所以产品线也可以认为由多个domain组成，metric产品线的配置文件为 metricProductLine.xml，默认配置如下，最外面是一个company，company下面可以有多条产品线，每条产品线下面又有多个domain。

**[html]** [view plain](https://blog.csdn.net/caohao0591/article/details/80231519" \o "view plain) [copy](https://blog.csdn.net/caohao0591/article/details/80231519" \o "copy)

1. **<?xml** version="1.0" encoding="utf-8"**?>**
2. **<company>**
3. **<product-line** id="Platform" order="10.0" title="架构" **>**
4. **<domain** id="Cat"**/>**
5. **<domain** id="PumaServer"**/>**
6. **<domain** id="SessionService"**/>**
7. **</product-line>**
8. **</company>**

    每个domain对应哪条产品线是由产品线配置管理类(ProductLineConfigManager)维护的，产品线配置管理类通过一个Map存储domain id 与 product-line id的映射关系，这些映射关系在产品线配置管理类初始化的时候被创建，通过buildMetricProductLines函数。

    除此之外，在初始化函数中，ProductLineConfigManager类还会根据配置初始化ProductLineConfig中指定的其它6中监控类型的产品线，分别是userProductLine-外部监控，applicationProductLine-应用监控，networkProductLine-网络监控，systemProductLine-系统监控，databaseProductLine-数据库监控，cdnProductLine-CDN监控，我们可以在上一章的配置列表中找到这些监控类别的配置。

**[html]** [view plain](https://blog.csdn.net/caohao0591/article/details/80231519" \o "view plain) [copy](https://blog.csdn.net/caohao0591/article/details/80231519" \o "copy)

1. public class ProductLineConfigManager implements Initializable, LogEnabled {
2. private volatile Map**<String**, String**>** m\_metricProductLines = new HashMap**<String**, String**>**();
4. private Map**<String**, String**>** buildMetricProductLines() {
5. Map**<String**, String**>** domainToProductLines = new HashMap**<String**, String**>**();
7. for (ProductLine product : ProductLineConfig.METRIC.getCompany().getProductLines().values()) {
8. for (Domain domain : product.getDomains().values()) {
9. domainToProductLines.put(domain.getId(), product.getId());
10. }
11. }
12. return domainToProductLines;
13. }
15. @Override
16. public void initialize() throws InitializationException {
17. for (ProductLineConfig productLine : ProductLineConfig.values()) {
18. initializeConfig(productLine);
19. }
20. m\_metricProductLines = buildMetricProductLines();
21. }
22. }

接下来我们看看MetricAnalyzer做了什么事情，他会根据domain获取消息所属的产品线，然后生成对应的业务报表(MetricReport) ，也就是说，一条产品线对应一个业务报表，一个业务报表，一共包含3个维度的统计，总条数、总额度、平均数，我们来看一个客户端的案例。

**[html]** [view plain](https://blog.csdn.net/caohao0591/article/details/80231519" \o "view plain) [copy](https://blog.csdn.net/caohao0591/article/details/80231519" \o "copy)

1. MessageProducer cat = Cat.getProducer();
2. Transaction t = cat.newTransaction("URL", "WebPage");
4. cat.logMetric("payCount", "C", "1");
5. cat.logMetric("totalfee", "S", "30.5");
6. cat.logMetric("avgfee", "T", "25.6");
7. cat.logMetric("order", "S,C", "3,25.6");
9. Metric event = Cat.getProducer().newMetric("kingsoft", "praise");
10. event.setStatus("C");
11. event.addData("3");
12. event.complete();
14. ...

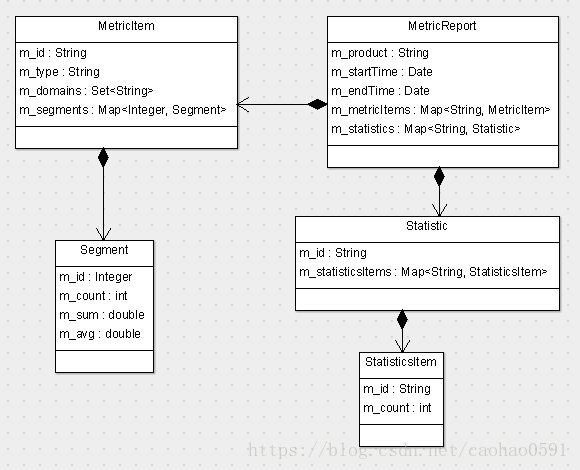
    status = "C" 表示总条数， 默认加1， "S"表示总额度，"T"表示平均数，"S,C"表示总条数+总金额，logMetric函数的第一个参数 payCount/totalfee/avgfee/order 都是表示的业务名，大家可以看到这里并没有上报type类型，实际上type表示的就是统计到哪条产品线下面，但是我们cat客户端默认会上送domain字段，MetricAnalyzer可以通过根据domain找到对应的产品线，生成相应报表。

     然而，我们也可以通过添加type类型对指定产品线做统计，比如案例中的 newMetric("kingsoft", "praise") 调用，我们就指定向 kingsoft 产品线统计，这时候MetricAnalyzer会调用 insertIfNotExsit 函数匹配所有监控类型的 product-line和domain，找到与客户端上报的type、domain匹配的产品线。找到则返回，没有找到的话，会为此新建一条产品线，放入到Metric监控类型中，并更新到数据库。然后为新的产品线创建一个report报表。

**[html]** [view plain](https://blog.csdn.net/caohao0591/article/details/80231519" \o "view plain) [copy](https://blog.csdn.net/caohao0591/article/details/80231519" \o "copy)

1. public class MetricAnalyzer extends AbstractMessageAnalyzer**<MetricReport>** implements LogEnabled {
2. private int processMetric(MetricReport report, MessageTree tree, Metric metric) {
3. String group = metric.getType();
4. String metricName = metric.getName();
5. String domain = tree.getDomain();
6. String data = (String) metric.getData();
7. String status = metric.getStatus();
8. ConfigItem config = parseValue(status, data);
10. if (StringUtils.isNotEmpty(group)) {
11. boolean result = m\_productLineConfigManager.insertIfNotExsit(group, domain);
13. if (!result) {
14. m\_logger.error(String.format("error when insert product line info, productline %s, domain %s", group,
15. domain));
16. }
18. report = findOrCreateReport(group);
19. }
20. if (config != null && report != null) {
21. ...
22. }
23. return 0;
24. }
26. private ConfigItem parseValue(String status, String data) {
27. ConfigItem config = new ConfigItem();
29. if ("C".equals(status)) {
30. ...
31. } else if ("T".equals(status)) {
32. ...
33. } else if ("S".equals(status)) {
34. ...
35. } else if ("S,C".equals(status)) {
36. ...
37. } else {
38. return null;
39. }
41. return config;
42. }
43. }

接下来我们看下具体报表统计逻辑。



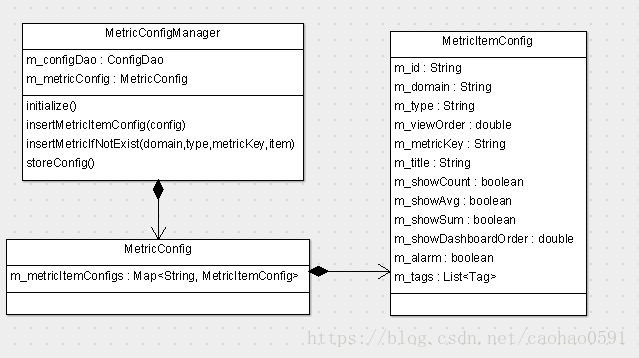
    我们知道，每个Metric报表都是一条产品线数据的集合，具体的统计信息记录在 MetricItem 中，domain+METRIC+metricName 用以唯一标识一个MetricItem，MetricItem 的 m\_type 字段用于区别统计的维度，所以，我们可以推出， 同一个 domain+metricName 只能用于统计一个维度的数据，Metric报表统计的最小粒度是分钟，MetricItem下每个Segment存储的是每分钟的统计信息。

**[html]** [view plain](https://blog.csdn.net/caohao0591/article/details/80231519" \o "view plain) [copy](https://blog.csdn.net/caohao0591/article/details/80231519" \o "copy)

1. public class MetricAnalyzer extends AbstractMessageAnalyzer**<MetricReport>** implements LogEnabled {
2. private int processMetric(MetricReport report, MessageTree tree, Metric metric) {
3. ...
4. ConfigItem config = parseValue(status, data);
5. ...
7. if (config != null && report != null) {
8. long current = metric.getTimestamp() / 1000 / 60;
9. int min = (int) (current % (60));
10. String key = m\_configManager.buildMetricKey(domain, METRIC, metricName);
11. MetricItem metricItem = report.findOrCreateMetricItem(key);
13. metricItem.addDomain(domain).setType(status);
14. updateMetric(metricItem, min, config.getCount(), config.getValue());
16. config.setTitle(metricName);
18. ProductLineConfig productLineConfig = m\_productLineConfigManager.queryProductLine(report.getProduct());
20. if (ProductLineConfig.METRIC.equals(productLineConfig)) {
21. boolean result = m\_configManager.insertMetricIfNotExist(domain, METRIC, metricName, config);
22. ...
23. }
24. }
25. return 0;
26. }
28. private void updateMetric(MetricItem metricItem, int minute, int count, double sum) {
29. Segment seg = metricItem.findOrCreateSegment(minute);
31. seg.setCount(seg.getCount() + count);
32. seg.setSum(seg.getSum() + sum);
33. seg.setAvg(seg.getSum() / seg.getCount());
34. }
35. }

报表展示配置

统计完成之后， 我们会利用ConfigItem对象为Metric报表首次初始化展示配置，并更新到数据库config表。



**[html]** [view plain](https://blog.csdn.net/caohao0591/article/details/80231519" \o "view plain) [copy](https://blog.csdn.net/caohao0591/article/details/80231519" \o "copy)

1. public class MetricConfigManager implements Initializable, LogEnabled {
2. private volatile MetricConfig m\_metricConfig;
4. public boolean insertMetricIfNotExist(String domain, String type, String metricKey, ConfigItem item) {
5. String key = buildMetricKey(domain, type, metricKey);
6. MetricItemConfig config = m\_metricConfig.findMetricItemConfig(key);
8. if (config != null) {
9. return true;
10. } else {
11. config = new MetricItemConfig();
13. config.setId(key);
14. config.setDomain(domain);
15. config.setType(type);
16. config.setMetricKey(metricKey);
17. config.setTitle(item.getTitle());
18. config.setShowAvg(item.isShowAvg());
19. config.setShowCount(item.isShowCount());
20. config.setShowSum(item.isShowSum());
21. m\_logger.info("insert metric config info " + config.toString());
22. return insertMetricItemConfig(config);
23. }
24. }
25. }

## ProblemAnalyzer -异常分析

Problem记录整个项目在运行过程中出现的问题，包括一些错误、访问较长的行为。



Problem分析器会通过报表管理器(ReportManager)根据时间和domain，获取对应报表，然后根据IP，找到相应的machine对象，将machine和消息交给问题处理器(ProblemHandler)处理，注意这里的Machine和之前报表的Machine类是不同的。

**[html]** [view plain](https://blog.csdn.net/caohao0591/article/details/80231519" \o "view plain) [copy](https://blog.csdn.net/caohao0591/article/details/80231519" \o "copy)

1. public class ProblemAnalyzer extends AbstractMessageAnalyzer**<ProblemReport>** implements LogEnabled, Initializable {
2. public static final String ID = "problem";
4. @Inject(ID)
5. private ReportManager**<ProblemReport>** m\_reportManager;
7. @Inject
8. private List**<ProblemHandler>** m\_handlers;
10. @Override
11. public void process(MessageTree tree) {
12. String domain = tree.getDomain();
13. ProblemReport report = m\_reportManager.getHourlyReport(getStartTime(), domain, true);
15. report.addIp(tree.getIpAddress());
16. Machine machine = report.findOrCreateMachine(tree.getIpAddress());
18. for (ProblemHandler handler : m\_handlers) {
19. handler.handle(machine, tree);
20. }
21. }
23. }

CAT默认有DefaultProblemHandler和LongExecutionProblemHandler两个问题处理器，我们可以也定义自己的问题处理器，用于收集我们感兴趣的问题，只需要我们的问题处理器继承自ProblemHandler，并且重写了handle方法，然后在components-cat-consumer.xml中相关模块配置了即可，如下：

**[html]** [view plain](https://blog.csdn.net/caohao0591/article/details/80231519" \o "view plain) [copy](https://blog.csdn.net/caohao0591/article/details/80231519" \o "copy)

1. **<component>**
2. **<role>**com.dianping.cat.analysis.MessageAnalyzer**</role>**
3. **<role-hint>**problem**</role-hint>**
4. **<implementation>**com.dianping.cat.consumer.problem.ProblemAnalyzer**</implementation>**
5. **<instantiation-strategy>**per-lookup**</instantiation-strategy>**
6. **<requirements>**
7. ...
8. **<requirement>**
9. **<role>**com.dianping.cat.consumer.problem.ProblemHandler**</role>**
10. **<field-name>**m\_handlers**</field-name>**
11. **<role-hints>**
12. **<role-hint>**default-problem**</role-hint>**
13. **<role-hint>**long-execution**</role-hint>**
14. **</role-hints>**
15. **</requirement>**
16. **</requirements>**
17. **</component>**

我们先来看看DefaultProblemHandler，他主要用于收集3类Problem，

1、状态不为SUCCESS的事务消息

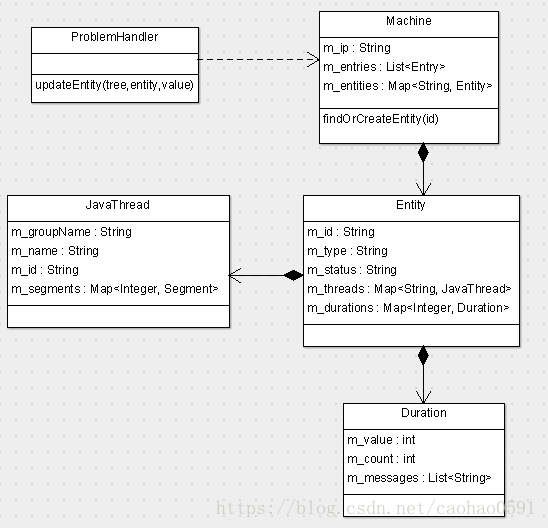
2、components-cat-consumer.xml中指定errorType类型的Event消息，如下Error、RuntimeException、Exception三种类型消息。

3、heartbeat异常。

**[html]** [view plain](https://blog.csdn.net/caohao0591/article/details/80231519" \o "view plain) [copy](https://blog.csdn.net/caohao0591/article/details/80231519" \o "copy)

1. **<component>**
2. **<role>**com.dianping.cat.consumer.problem.ProblemHandler**</role>**
3. **<role-hint>**default-problem**</role-hint>**
4. **<implementation>**com.dianping.cat.consumer.problem.DefaultProblemHandler**</implementation>**
5. **<configuration>**
6. **<errorType>**Error,RuntimeException,Exception**</errorType>**
7. **</configuration>**
8. ...
9. **</component>**

接下来我们看下DefaultProblemHandler处理之后生成的Problem报表的组成结构。



不同IP的Problem消息存储于不同的Machine里面，而且我们还会为不同消息类型、消息名称创建相应的Entity存储消息信息，在Entity中，消息分两种方式存储，

一种是按duration存储，这里的duration指事务执行时间所在的阈值，不过DefaultProblemHandler并不关心duration的不同，所以这里duration全部等于0，我们会在超时调用处理器(LongExecutionProblemHandler)中去做讲解。

另外一种是按线程组来存储，线程组内的消息统计最小粒度为分钟，每个分钟数据统计在Segment中，不管是哪种方式，我们存储的都只是消息树的message\_id，而且存储的消息总数有限制，默认每分钟最多只能存储60条消息。

**[html]** [view plain](https://blog.csdn.net/caohao0591/article/details/80231519" \o "view plain) [copy](https://blog.csdn.net/caohao0591/article/details/80231519" \o "copy)

1. public abstract class ProblemHandler {
2. public static final int MAX\_LOG\_SIZE = 60;
3. public void updateEntity(MessageTree tree, Entity entity, int value) {
4. Duration duration = entity.findOrCreateDuration(value);
5. List**<String>** messages = duration.getMessages();
7. duration.incCount();
8. if (messages.size() **<** **MAX\_LOG\_SIZE**) {
9. messages.add(tree.getMessageId());
10. }
12. // make problem thread id = thread group name, make report small
13. JavaThread thread = entity.findOrCreateThread(tree.getThreadGroupName());
15. if (thread.getGroupName() == null) {
16. thread.setGroupName(tree.getThreadGroupName());
17. }
18. if (thread.getName() == null) {
19. thread.setName(tree.getThreadName());
20. }
22. Segment segment = thread.findOrCreateSegment(getSegmentByMessage(tree));
23. List**<String>** segmentMessages = segment.getMessages();
25. segment.incCount();
26. if (segmentMessages.size() **<** **MAX\_LOG\_SIZE**) {
27. segmentMessages.add(tree.getMessageId());
28. }
29. }
30. }

LongExecutionProblemHandler

超时调用处理器(LongExecutionProblemHandler)用于监控系统中用时比较长的调用，可以是缓存调用、数据库查询，也可以是一次RPC调用、微服务请求、还可以是一次HTTP请求。超时调用处理器分析的对象仅仅是 Transaction事务类型消息。

**[html]** [view plain](https://blog.csdn.net/caohao0591/article/details/80231519" \o "view plain) [copy](https://blog.csdn.net/caohao0591/article/details/80231519" \o "copy)

1. public class LongExecutionProblemHandler extends ProblemHandler implements Initializable {
2. private void processTransaction(Machine machine, Transaction transaction, MessageTree tree) {
3. String type = transaction.getType();
5. if (type.startsWith("Cache.")) {
6. processLongCache(machine, transaction, tree);
7. } else if (type.equals("SQL")) {
8. processLongSql(machine, transaction, tree);
9. } else if (m\_configManager.isRpcClient(type)) {
10. processLongCall(machine, transaction, tree);
11. } else if (m\_configManager.isRpcServer(type)) {
12. processLongService(machine, transaction, tree);
13. } else if ("URL".equals(type)) {
14. processLongUrl(machine, transaction, tree);
15. }
17. List**<Message>** messageList = transaction.getChildren();
19. for (Message message : messageList) {
20. if (message instanceof Transaction) {
21. processTransaction(machine, (Transaction) message, tree);
22. }
23. }
24. }
25. }

    超时调用处理器(LongExecutionProblemHandler)会计算事务执行时间，首先查看是否超过了默认阈值设置，每种调用的阈值设置都不同，分4-6个级别，如下，

**[html]** [view plain](https://blog.csdn.net/caohao0591/article/details/80231519" \o "view plain) [copy](https://blog.csdn.net/caohao0591/article/details/80231519" \o "copy)

1. m\_defaultLongServiceDuration = { 50, 100, 500, 1000, 3000, 5000 }
2. m\_defaultLongSqlDuration = { 100, 500, 1000, 3000, 5000 }
3. m\_defaultLongUrlDuration = { 1000, 2000, 3000, 5000 }
4. m\_defalutLongCallDuration = { 100, 500, 1000, 3000, 5000 }
5. m\_defaultLongCacheDuration = { 10, 50, 100, 500 }

    在校验完默认阈值之后，还可以根据配置来设置不同domain的超时阈值，在 E:/data/appdatas/cat/server.xml中配置，由ServerConfigManager管理，server.xml所在目录可以通过设置环境变量CAT\_HOME指定，如下我配置了 dianping和kingsoft两个domain的超时阈值。

**[html]** [view plain](https://blog.csdn.net/caohao0591/article/details/80231519" \o "view plain) [copy](https://blog.csdn.net/caohao0591/article/details/80231519" \o "copy)

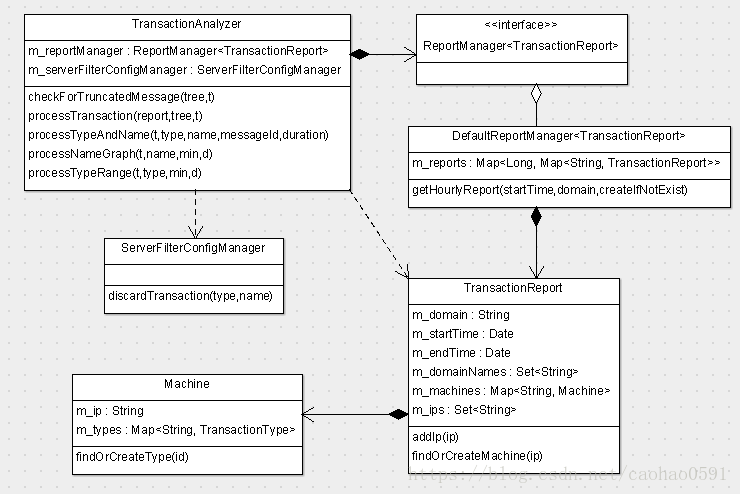
1. **<config** local-mode="false" hdfs-machine="false" job-machine="true" alert-machine="false"**>**
2. ...
3. **<consumer>**
4. **<long-config>**
5. **<domain** name="dianping" url-threshold="30" sql-threshold="15" service-threshold="60"**/>**
6. **<domain** name="kingsoft" url-threshold='10' sql-threshold='25' service-threshold='5'**/>**
7. **</long-config>**
8. **</consumer>**
9. ...
10. **</config>**

    超时的事务消息将会被存储到指定machine的entity中，逻辑与DefaultProblemHandler相同，只不过，这里的duration就有了实际意义，问题处理器会根据事务所超出的阈值范围来存储到对应的Duration对象里面去。

自定义自己的问题处理器

## TransactionAnalyzer - 事务分析

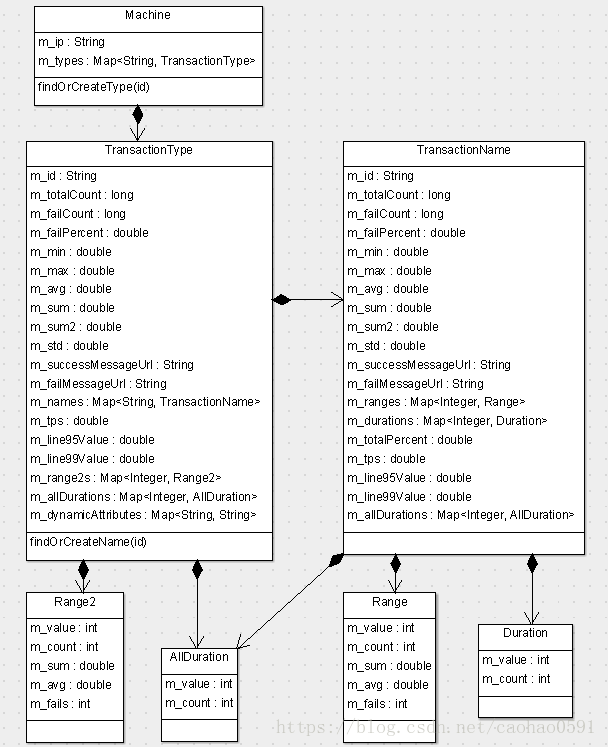
事务分析器会统计事务(Transaction)类型消息的运行时间，次数，错误次数，当然不是所有Transactionx消息都会被统计，Cache.web、ABTest以及serverFilter配置指定需要过滤的事务消息，会在分析器处理时被丢弃。



    统计结果存于TransactionReport，依然是以周期时间和domain来划分不同的报表。

 checkForTruncatedMessage ？？

    相同的domain下的不同IP对应的统计信息依然是存于不同的Machine对象中，截止目前我们已经看到很多报表都包含有Machine类，但是一定注意他们的Machine类都是不同的，可以在cat-consumer/target/generated-sources/dal-model/com/dianping/cat/consumer/ 目录下去查看这些类。



    每台机器下面，不同类型的事务统计信息会存于不同的TransactionType对象里，在管理页面上，我们展开指定Type，会看到该Type下所有Name的统计信息，相同Type下的不同名称的统计信息就是分别存在于不同的TransactionName下面，点开每条记录前面的 [:: show ::]， 我们将会看到该周期小时内每分钟的统计信息，每分钟的统计存储在 Type 的 Range2对象、Name的Range对象内，实际上Range2和Range对象的代码结构完全一致，除了类名不同，你可以认为他们就是同一个东西。

    Type和Name都会统计总执行次数、失败次数、示例链接、最小时间、最大调用时间、平均值、标准差等等信息，同时分析器会选取最近一条消息作为他的示例链接，将messageId存于m\_successMessageUrl或者m\_failMessageUrl中。

    我们会根据一定规则划分几个执行时间区间，将该区间的事务消息总数统计在 AllDuration 和 Duration 对象中

# 第七章 消息分析器与报表(二)

## CrossAnalyzer-调用链分析

    在分布式环境中，应用是运行在独立的进程中的，有可能是不同的机器，或者不同的服务器进程。那么他们如果想要彼此联系在一起，形成一个调用链，在Cat中，CrossAnalyzer会统计不同服务之间调用的情况，包括服务的访问量，错误量，响应时间，QPS等，这里的服务主要指的是 RPC 服务。

    在讲 CrossAnalyzer 的处理逻辑之前，我们先看下客户端的埋点的一个模拟情况。

    一般情况下不同服务会通过几个ID进行串联。这种串联的模式，基本上都是一样的。在Cat中，我们需要3个ID：

* RootId，用于标识唯一的一个调用链
* ParentId，父Id是谁？谁在调用我
* ChildId，我在调用谁?

      那么我们如何传递这些ID？Cat为我们提供了一个内部接口 Cat.Context，但是我们需要自己实现Context，在下面代码中我们首先在before函数中实现了Context 上下文，然后在rpcClient中开启消息事务，并调用 Cat.logRemoteCallClient(context) 去填充Context的这3个MessageID。当然，该函数还记录了一个RemoteCall类型的Event消息。

     随后我们用rpcService函数中开启新线程模拟远程RPC服务，并将context上传到 RPC 服务器，在真实环境中，Context是需要跨进程网络传输，因此需要实现序列化接口。

    在rpcService中，我们会调用 Cat.logRemoteCallServer(context) 将从rpcClient传过来的Context设置到自己的 Transaction 当中。

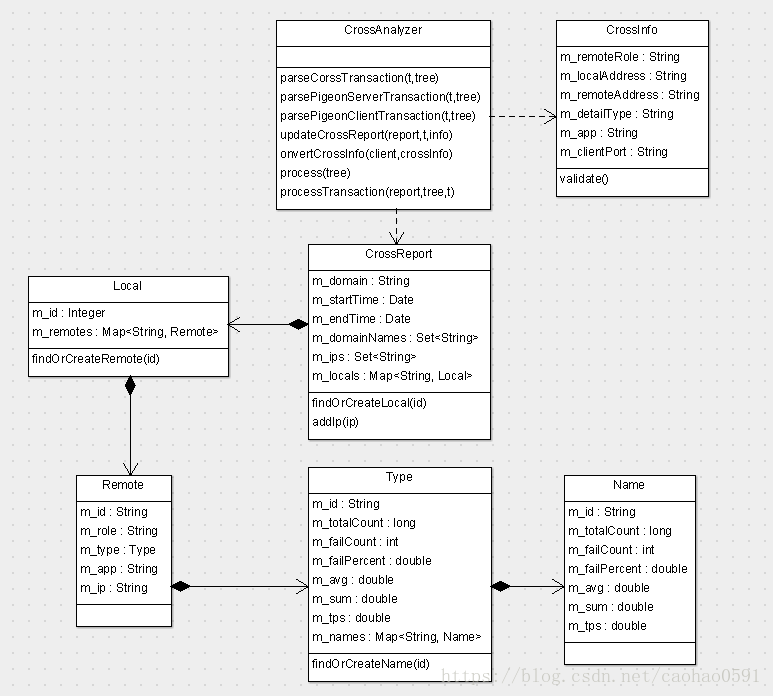
    随着业务处理逻辑的结束， rpcServer 和 rpcClient 都会分别将自己的消息树上传到CAT服务器分析。

    需要注意的是，Service的client和app需要和Call的server以及app对应上，要不然图表是分析不出东西的！

**[java]** [view plain](https://blog.csdn.net/caohao0591/article/details/80448847" \o "view plain) [copy](https://blog.csdn.net/caohao0591/article/details/80448847" \o "copy)

1. @RunWith(JUnit4.**class**)
2. **public** **class** AppSimulator **extends** CatTestCase {
3. **public** Map<String, String> maps = **new** HashMap<String, String>();
5. **public** Cat.Context context;
7. @Before
8. **public** **void** before() {
9. context = **new** Cat.Context() {
10. @Override
11. **public** **void** addProperty(String key, String value) { maps.put(key, value); }
13. @Override
14. **public** String getProperty(String key) { **return** maps.get(key); }
15. };
16. }
18. @Test
19. **public** **void** simulateHierarchyTransaction() **throws** Exception {
20. ...
21. //RPC调用开始
22. rpcClient();
23. rpcClient2();
24. ...
25. }
27. **protected** **void** rpcClient() {
28. //客户端埋点，Domain为RpcClient，调用服务端提供的Echo服务
29. Transaction parent = Cat.newTransaction("Call", "CallServiceEcho");
30. Cat.getManager().getThreadLocalMessageTree().setDomain("RpcClient");
32. Cat.logEvent("Call.server","localhost");
33. Cat.logEvent("Call.app","RpcService");
34. Cat.logEvent("Call.port","8888");
35. Cat.logRemoteCallClient(context, "RpcClient");
37. //开启新线程模拟远程RPC服务，将context上传到 RPC 服务器
38. rpcService(context);
40. parent.complete();
41. }
43. **protected** **void** rpcClient2() {
44. ...
45. //模拟另外一个RpcClient调用Echo服务
46. rpcService(context, "RpcClient2");
47. ...
48. }
50. **protected** **void** rpcService(**final** Cat.Context context, **final** String clientDomain) {
51. Thread thread = **new** Thread() {
52. @Override
53. **public** **void** run() {
54. //服务器埋点，Domain为 RpcService 提供Echo服务
55. Transaction child = Cat.newTransaction("Service", "Echo");
56. Cat.getManager().getThreadLocalMessageTree().setDomain("RpcService");
58. Cat.logEvent("Service.client", localhost); //填客户端地址
59. Cat.logEvent("Service.app", clientDomain);
60. Cat.logRemoteCallServer(context);
62. //to do your business
64. child.complete();
65. }
66. };
68. thread.start();
70. **try** {
71. thread.join();
72. } **catch** (InterruptedException e) {
73. }
74. }
75. }

接下来我们看看CAT服务器端CrossAnalyzer的逻辑。



        我们依然会为每个周期时间内的每个Domain创建一张报表(CrossReport)，然后不同的IP会分配不同的Local对象统计，每个IP又可能会接收来自不同Remote端的调用。

    由于这里一个完整的调用链会涉及多个端的多个消息树，我们首先会根据Transaction的类型来判断是RpcService还是RpcClient，如果Type等于PigeonService或Service则该消息来自RpcService，如果Type等于 PigeonCall或Call则来自RpcClient。

    先来看看RpcService端消息树的上报处理逻辑，CAT会调用 parsePigeonServerTransaction 函数去填充 CrossInfo 信息，CrossInfo包含的具体内容如下：

     localAddress ： RpcService的IP地址

     remoteAddress ： 服务调用者(RpcClient)的IP地址，由type="Service.client" 的Event子消息提供，注意，在处理RpcClient的上报时，我们会根据上报信息中的remoteAddress再次统计该RpcService数据，大家可能会疑惑这里是不是重复统计，事实上他们所处的视角是不一样的，前者是站在服务提供者的视角来统计我完成这次服务所耗费的时间、资源等，而后者则是站在RpcClient视角去统计自己从发出请求到得到结果所需的时长、资源等等，比如这中间就包含网络IO的消耗，这些在后续的报表中会有体现。

       app：客户端的Domain， 由type="Service.app"的Event子消息提供。

       remoteRole：固定为 Pigeon.Client ， 表示远端角色为 Rpc 客户端。

       detailType： 固定为 PigeonService ， 表示自己角色为 Rpc 服务端。

        最后，我们将用CrossInfo信息来更新报表(CrossReport)，我们首先根据 localAddress 即 RpcService的 IP 找到或创建 Local对象，然后根据 remoteAddress+remoteRole 找到或创建 Remote 对象，然后统计服务的访问量，错误量，处理时间，QPS。

       RpcService提供不只一个服务，不同的服务我们按名字分别统计在不同的Name对象里，比如上面案例，RpcService提供的是Echo服务。

我们再来看看RpcClient端上报处理逻辑，CAT调用parsePigeonClientTransaction函数填充CrossInfo信息，具体如下：

localAddress ： RpcClient的IP地址

remoteAddress ：服务提供者(RpcService)的地址，由 type="Call.server" 的Event子消息提供。

app：服务提供者的Domain，由type="Call.app" 的Event子消息提供，在统计完RpcClient端数据之后，会通过该属性获取服务提供者的CrossInfo。从RpcClient的视角再次统计RpcService的数据。

port：客户端端口，由 type="Call.port" 的Event子消息提供。

remoteRole：固定为 Pigeon.Server， 表示远端角色为服务提供者。

detailType： 固定为 PigeonCall ， 表示自己角色为服务调用者。

    然后，我们将用CrossInfo信息来更新报表(CrossReport)，也是根据 localAddress 找到Local对象，然后根据 remoteAddress+remoteRole 找到 Remote 对象，进行统计。

接着，我们通过convertCrossInfo函数利用RpcClient的CrossInfo信息去生成服务提供者的CrossInfo信息，这里实际上是为了从RpcClient的视角去统计服务提供者的报表！

**[java]** [view plain](https://blog.csdn.net/caohao0591/article/details/80448847" \o "view plain) [copy](https://blog.csdn.net/caohao0591/article/details/80448847" \o "copy)

1. **public** **class** CrossAnalyzer **extends** AbstractMessageAnalyzer<CrossReport> **implements** LogEnabled {
2. **private** **void** processTransaction(CrossReport report, MessageTree tree, Transaction t) {
3. CrossInfo crossInfo = parseCorssTransaction(t, tree);
5. **if** (crossInfo != **null** && crossInfo.validate()) {
6. updateCrossReport(report, t, crossInfo);
8. String targetDomain = crossInfo.getApp();
10. **if** (m\_serverConfigManager.isRpcClient(t.getType()) && !DEFAULT.equals(targetDomain)) {
11. CrossInfo serverCrossInfo = convertCrossInfo(tree.getDomain(), crossInfo);
13. **if** (serverCrossInfo != **null**) {
14. CrossReport serverReport = m\_reportManager.getHourlyReport(getStartTime(), targetDomain, **true**);
16. updateCrossReport(serverReport, t, serverCrossInfo);
17. }
18. } **else** {
19. m\_errorAppName++;
20. }
21. }
22. ...
23. }
24. }

这里的 serverCrossInfo 被填充了什么数据：

localAddress ： RpcClient 的 remoteAddress。

remoteAddress ：RpcClient 的 localAddress + clientPort

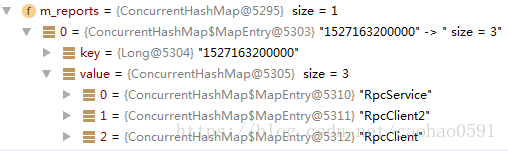
app：RpcClient 的 Domain。

remoteRole：固定为 Pigeon.Caller， 表示远端角色为服务调用者。

detailType： 固定为 PigeonCall

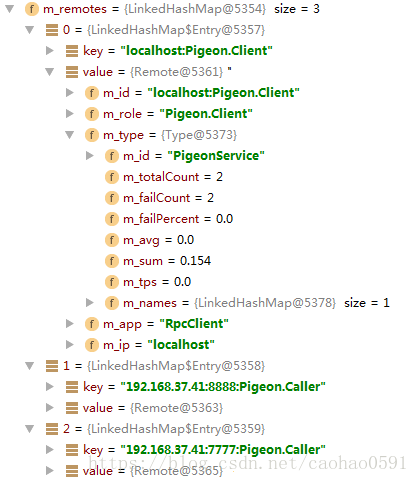
     最后还是用CrossInfo信息来更新报表(CrossReport)。

最后我们看看我们生成了哪些报表数据，3个报表数据，分别是服务调用方 RpcClient和 RpcClient2，以及服务提供方RpcService。



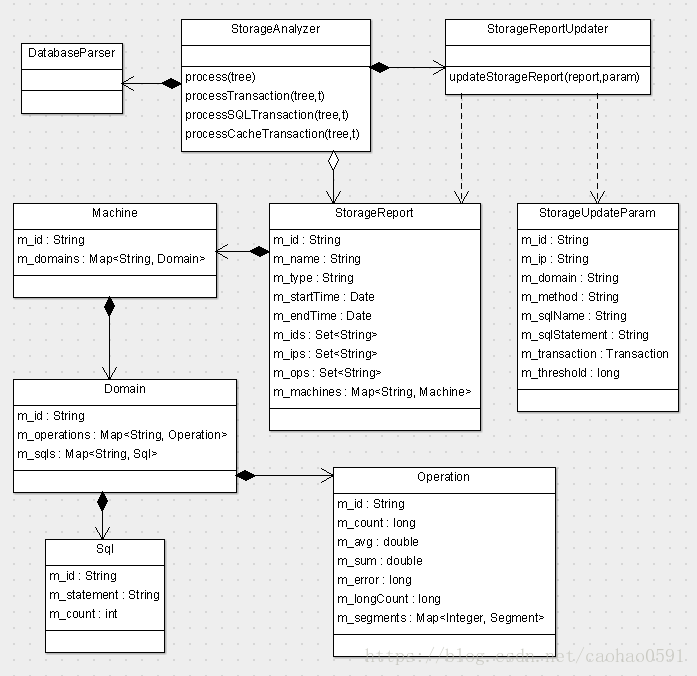
    接下来我们看看服务提供方的remotes数据信息，一共3条数据，第1条记录是站在RpcService角度统计服务器完成这2次服务所耗费的时间、资源等，后面2条记录则是站在RpcClient视角去统计自己从发出请求到得到结果所需的时长、资源等等。

    第1条记录 duration 为 0.154ms， 第2,3条记录 duration 分别为 1072.62ms、1506.38ms， 两者巨大的时间差一般就是网络 IO 所需的时间，事实上大多数的服务时间的消耗都是在各种IO上。这类服务统称为IO密集型。



## StorageAnalyzer  --数据库/缓存分析

    StorageAnalyzer主要分析一段时间内数据库、Cache访问情况：各种操作访问次数、响应时间、错误次数、长时间访问量等等，当客户端消息过来，StorageAnalyzer首先会分析事务属于数据库操作还是缓存操作，然后进行不同的处理，消息类型如果是SQL则是数据库操作，如果以Cache.memcached开头则认为是缓存操作。



    我们首先看看数据库操作的分析过程，下面源码是客户端的案例，这是一个获取cat库config表全部数据的sql查询，我们将数据库操作所有信息都放在一个type="SQL" 的子事务消息中。

**[java]** [view plain](https://blog.csdn.net/caohao0591/article/details/80448847" \o "view plain) [copy](https://blog.csdn.net/caohao0591/article/details/80448847" \o "copy)

1. @RunWith(JUnit4.**class**)
2. **public** **class** AppSimulator **extends** CatTestCase {
3. @Test
4. **public** **void** simulateHierarchyTransaction() **throws** Exception {
5. ...
6. Transaction sqlT = cat.newTransaction("SQL", "Select");
8. //do your SQL query
10. cat.logEvent("SQL.Database", "jdbc:mysql://192.168.20.67:3306/cat");
11. cat.logEvent("SQL.Method", "select");
12. cat.logEvent("SQL.Statement", "SELECT", SUCCESS, "select \* from cat.config");
13. sqlT.complete();
14. ...
15. }
16. }

    上面消息上报到服务端之后，分析器将SQL类型子事务取出，调用processSQLTransaction去处理，将结果写入报表StorageReport

    processSQLTransaction 首先通过DatabaseParser提取数据库的IP和数据库名称，该信息由type="SQL.Database"的Event子消息提供，该Event消息上报的是数据库连接的URL。

    接着我们会获取数据库操作名，type="SQL.Method" 的Event子消息提供，数据库操作分4类，分别是select, update, delete, insert，如果不上报，分析器默认客户端在做select查询。

    最后我们会为周期内的每个数据库创建一个Storage报表。并将提取信息放入StorageUpdateParam对象，然后将对象交给StorageReportUpdater来更新Storage报表。

**[java]** [view plain](https://blog.csdn.net/caohao0591/article/details/80448847" \o "view plain) [copy](https://blog.csdn.net/caohao0591/article/details/80448847" \o "copy)

1. **public** **class** StorageAnalyzer **extends** AbstractMessageAnalyzer<StorageReport> **implements** LogEnabled {
2. @Inject
3. **private** DatabaseParser m\_databaseParser;
5. @Inject
6. **private** StorageReportUpdater m\_updater;
8. **private** **void** processSQLTransaction(MessageTree tree, Transaction t) {
9. String databaseName = **null**;
10. String method = "select";
11. String ip = **null**;
12. String domain = tree.getDomain();
13. List<Message> messages = t.getChildren();
15. **for** (Message message : messages) {
16. **if** (message **instanceof** Event) {
17. String type = message.getType();
19. **if** (type.equals("SQL.Method")) {
20. method = message.getName().toLowerCase();
21. }
22. **if** (type.equals("SQL.Database")) {
23. Database database = m\_databaseParser.queryDatabaseName(message.getName());
25. **if** (database != **null**) {
26. ip = database.getIp();
27. databaseName = database.getName();
28. }
29. }
30. }
31. }
32. **if** (databaseName != **null** && ip != **null**) {
33. String id = querySQLId(databaseName);
34. StorageReport report = m\_reportManager.getHourlyReport(getStartTime(), id, **true**);
35. StorageUpdateParam param = **new** StorageUpdateParam();
37. param.setId(id).setDomain(domain).setIp(ip).setMethod(method).setTransaction(t)
38. .setThreshold(LONG\_SQL\_THRESHOLD);// .setSqlName(sqlName).setSqlStatement(sqlStatement);
39. m\_updater.updateStorageReport(report, param);
40. }
41. }
42. }

     数据库与缓存的报表更新逻辑相同，不同ip地址的数据库/缓存的统计信息在不同Machine里面，同时也可能会有不同的Domain访问同一个数据库/缓存，每个Domain的访问都会被单独统计，每个Domain对数据库/缓存不同的操作会统计在不同Operation里，除了当前小时周期的统计汇总外，我们还会用Segment记录每分钟的汇总数据。访问时间超过1秒的数据库操作(缓存是50ms) 会被认为是长时间访问记录。

缓存操作

    接下来我们看下缓存的案例，获取memcached中key="uid\_1234567"的值，Storage分析器会判断Type是否以"Cache.memcached"开头，如果是，则认为这是一个缓存操作，(这里代码我认为有些稍稍不合理，如果我用的是Redis缓存，我希望上报的Type="Cache.Redis"，所以我这里讲源码稍稍做了修改，判断Type如果以"Cache."开头，就认为是缓存)。

**[java]** [view plain](https://blog.csdn.net/caohao0591/article/details/80448847" \o "view plain) [copy](https://blog.csdn.net/caohao0591/article/details/80448847" \o "copy)

1. @RunWith(JUnit4.**class**)
2. **public** **class** AppSimulator **extends** CatTestCase {
3. @Test
4. **public** **void** simulateHierarchyTransaction() **throws** Exception {
5. ...
6. Transaction cacheT = cat.newTransaction("Cache.memcached", "get:uid\_1234567");
8. //do your cache operation
10. cat.logEvent("Cache.memcached.server", "192.168.20.67:6379");
11. cacheT.complete();
12. ...
13. }
14. }

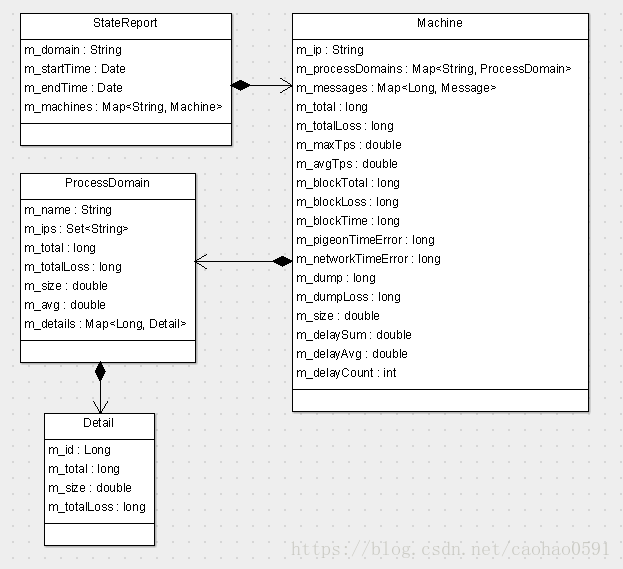
接下来我们看下Storage分析器的处理逻辑，processCacheTransaction负责分析消息， 事务类型"Cache.memcached"的“Cache.”后面部分将会被提取作为缓存类型，分析器会为每个类型的缓存都创建一个报表，事务名称":"前面部分会被提取作为操作名称，一般缓存有 add,get,hGet,mGet,remove等操作，缓存地址将由type="Cache.memcached.server"的Event子消息提供，最后我们还是将domain、ip、method、事务、阈值等消息放入StorageUpdateParam交由StorageReportUpdater来更新报表，更新逻辑与数据库一致。

**[java]** [view plain](https://blog.csdn.net/caohao0591/article/details/80448847" \o "view plain) [copy](https://blog.csdn.net/caohao0591/article/details/80448847" \o "copy)

1. **public** **class** StorageAnalyzer **extends** AbstractMessageAnalyzer<StorageReport> **implements** LogEnabled {
2. @Inject
3. **private** StorageReportUpdater m\_updater;
5. **private** **void** processCacheTransaction(MessageTree tree, Transaction t) {
6. String cachePrefix = "Cache.";
7. String ip = "Default";
8. String domain = tree.getDomain();
9. String cacheType = t.getType().substring(cachePrefix.length());
10. String name = t.getName();
11. String method = name.substring(name.lastIndexOf(":") + 1);
12. List<Message> messages = t.getChildren();
14. **for** (Message message : messages) {
15. **if** (message **instanceof** Event) {
16. String type = message.getType();
18. **if** (type.equals("Cache.memcached.server")) {
19. ip = message.getName();
20. **int** index = ip.indexOf(":");
22. **if** (index > -1) {
23. ip = ip.substring(0, index);
24. }
25. }
26. }
27. }
28. String id = queryCacheId(cacheType);
29. StorageReport report = m\_reportManager.getHourlyReport(getStartTime(), id, **true**);
30. StorageUpdateParam param = **new** StorageUpdateParam();
32. param.setId(id).setDomain(domain).setIp(ip).setMethod(method).setTransaction(t)
33. .setThreshold(LONG\_CACHE\_THRESHOLD);
34. m\_updater.updateStorageReport(report, param);
35. }
36. }

## StateAnalyzer

主要是分析CAT服务器自身的异常

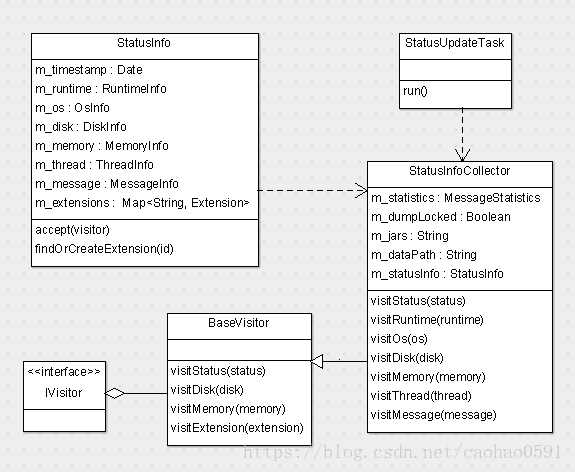


## HeartbeatAnalyzer

    分析器HeartbeatAnalyzer用于上报的心跳数据的分析。我们先看看客户端的收集逻辑，CAT客户端在初始化CatClientModule的时候，会开启一个StatusUpdateTask的线程任务，每隔一分钟去收集客户端的心跳状态，通过 Heartbeat 消息上报到客户端，心跳数据以xml格式存在于Heartbeat消息中。

**[java]** [view plain](https://blog.csdn.net/caohao0591/article/details/80448847" \o "view plain) [copy](https://blog.csdn.net/caohao0591/article/details/80448847" \o "copy)

1. **public** **class** CatClientModule **extends** AbstractModule {
2. @Override
3. **protected** **void** execute(**final** ModuleContext ctx) **throws** Exception {
4. ...
5. **if** (clientConfigManager.isCatEnabled()) {
6. // start status update task
7. StatusUpdateTask statusUpdateTask = ctx.lookup(StatusUpdateTask.**class**);
9. Threads.forGroup("cat").start(statusUpdateTask);
10. ...
11. }
12. }
13. }



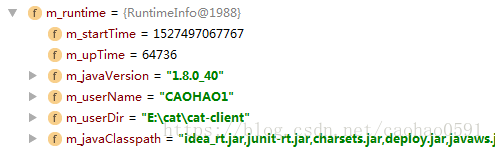
    Cat客户端会为Heartbeat消息创建一个System类型事务消息，然后将 Heartbeat 消息放入该事务，信息的收集靠StatusInfoCollector来完成，StatusInfoCollector将收集的数据写入StatusInfo对象，然后StatusUpdateTask将StatusInfo转化成xml之后放到Heartbeat消息数据段上报。

**[java]** [view plain](https://blog.csdn.net/caohao0591/article/details/80448847" \o "view plain) [copy](https://blog.csdn.net/caohao0591/article/details/80448847" \o "copy)

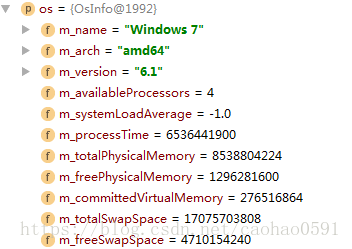
1. **public** **class** StatusUpdateTask **implements** Task, Initializable {
2. @Override
3. **public** **void** run() {
4. //创建类目录, 上报CAT客户端启动信息
5. ...
7. **while** (m\_active) {
8. **long** start = MilliSecondTimer.currentTimeMillis();
10. **if** (m\_manager.isCatEnabled()) {
11. Transaction t = cat.newTransaction("System", "Status");
12. Heartbeat h = cat.newHeartbeat("Heartbeat", m\_ipAddress);
13. StatusInfo status = **new** StatusInfo();
15. t.addData("dumpLocked", m\_manager.isDumpLocked());
16. **try** {
17. StatusInfoCollector statusInfoCollector = **new** StatusInfoCollector(m\_statistics, m\_jars);
19. status.accept(statusInfoCollector.setDumpLocked(m\_manager.isDumpLocked()));
21. buildExtensionData(status);
22. h.addData(status.toString());
23. h.setStatus(Message.SUCCESS);
24. } **catch** (Throwable e) {
25. h.setStatus(e);
26. cat.logError(e);
27. } **finally** {
28. h.complete();
29. }
30. t.setStatus(Message.SUCCESS);
31. t.complete();
32. }
34. //sleep 等待下一次心跳上报
35. ...
36. }
37. }
38. }

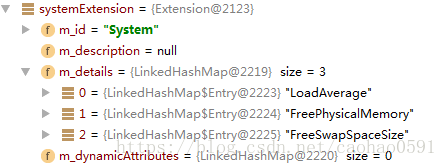
    我们上报的XML到底包含哪些数据，我们看下StatusInfo的结构，StatusInfo除了包含上报时间戳之外，还有哪些系统状态信息、附加扩展信息(Extension)会被StatusInfoCollector收集？

1、运行时数据RuntimeInfo：JAVA版本 java.version、用户名user.name、项目目录user.dir、java类路径等等。

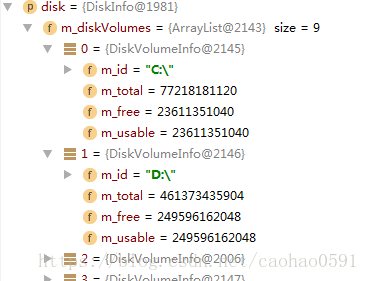


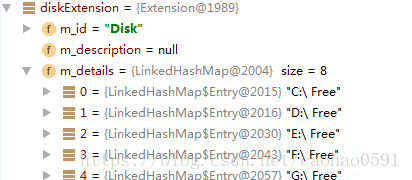
2、操作系统信息 OsInfo，同时创建System附加扩展信息。



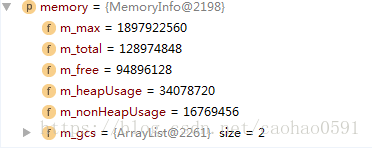


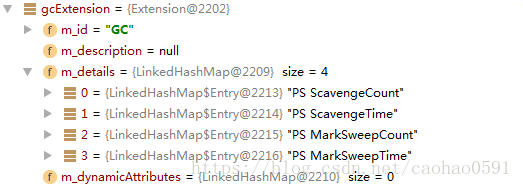
3、磁盘信息DiskInfo，磁盘的总量、空闲与使用情况，同时创建Disk附加扩展信息

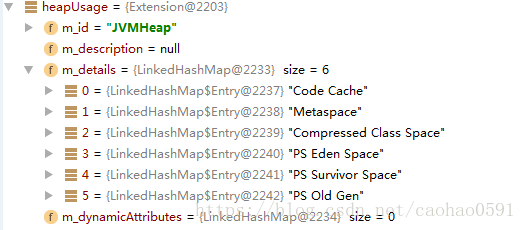




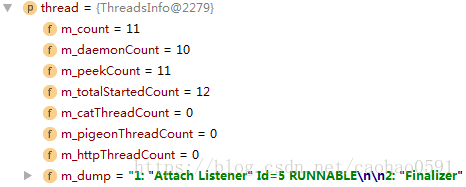
4、内存使用情况MemoryInfo，同时创建垃圾回收扩展信息、JAVA虚拟机堆附加扩展信息

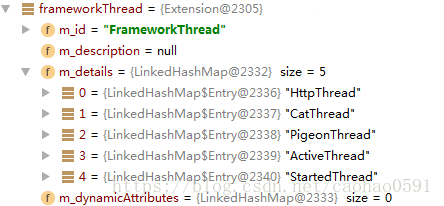




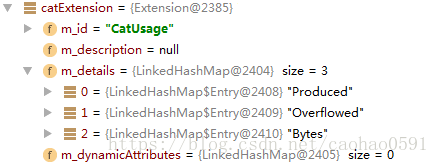


5、线程信息，以及 FrameworkThread 附加扩展信息。





6、Cat使用状态信息



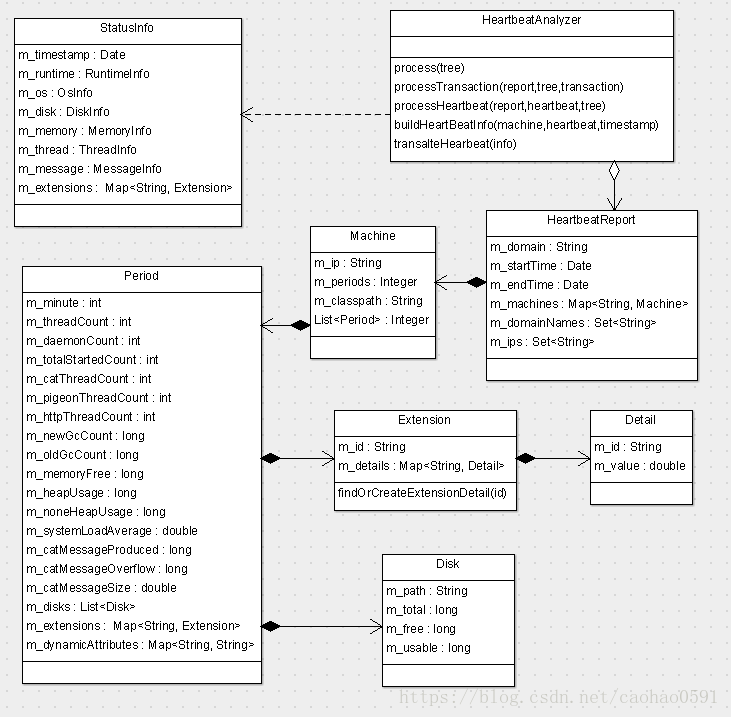
下面我列一个上报的XML数据案例：

**[java]** [view plain](https://blog.csdn.net/caohao0591/article/details/80448847" \o "view plain) [copy](https://blog.csdn.net/caohao0591/article/details/80448847" \o "copy)

1. <?xml version="1.0" encoding="utf-8"?>
2. <status timestamp="2018-05-28 16:23:08.625">
3. <runtime start-time="1527495705011" up-time="114212" java-version="1.8.0\_40" user-name="CAOHAO1">
4. <user-dir>E:\cat\cat-client</user-dir>
5. <java-classpath>idea\_rt.jar,junit-rt.jar,charsets.jar ... </java-classpath>
6. </runtime>
7. <os name="Windows 7" arch="amd64" version="6.1" available-processors="4" system-load-average="-1.0" process-time="5881237700" total-physical-memory="8538804224" free-physical-memory="1369870336" committed-virtual-memory="277372928" total-swap-space="17075703808" free-swap-space="4815220736"/>
9. <disk>
10. <disk-volume id="C:\" total="77218181120" free="23651565568" usable="23651565568"/>
11. <disk-volume id="D:\" total="461373435904" free="249596563456" usable="249596563456"/>
12. ...
13. </disk>
15. <memory max="1897922560" total="128974848" free="118387592" heap-usage="10961920" non-heap-usage="17081736">
16. <gc name="PS Scavenge" count="1" time="167"/>
17. <gc name="PS MarkSweep" count="0" time="0"/>
18. </memory>
19. <thread count="11" daemon-count="10" peek-count="11" total-started-count="12" cat-thread-count="0" pigeon-thread-count="0" http-thread-count="0">
20. <dump>1: "Attach Listener" Id=5 RUNNABLE ... </dump>
21. </thread>
22. <message produced="0" overflowed="0" bytes="0"/>
23. <extension id="System">
24. <extensionDetail id="LoadAverage" value="-1.0"/>
25. <extensionDetail id="FreePhysicalMemory" value="1.369247744E9"/>
26. <extensionDetail id="FreeSwapSpaceSize" value="4.814426112E9"/>
27. </extension>
28. <extension id="Disk">
29. <extensionDetail id="C:\ Free" value="2.3651565568E10"/>
30. <extensionDetail id="D:\ Free" value="2.49596563456E11"/>
31. ...
32. </extension>
33. <extension id="GC">
34. <extensionDetail id="PS ScavengeCount" value="1.0"/>
35. <extensionDetail id="PS ScavengeTime" value="167.0"/>
36. <extensionDetail id="PS MarkSweepCount" value="0.0"/>
37. <extensionDetail id="PS MarkSweepTime" value="0.0"/>
38. </extension>
39. <extension id="JVMHeap">
40. <extensionDetail id="Code Cache" value="3707200.0"/>
41. <extensionDetail id="Metaspace" value="1.2053E7"/>
42. <extensionDetail id="Compressed Class Space" value="1412600.0"/>
43. <extensionDetail id="PS Eden Space" value="4805792.0"/>
44. <extensionDetail id="PS Survivor Space" value="5214992.0"/>
45. <extensionDetail id="PS Old Gen" value="941136.0"/>
46. </extension>
47. <extension id="FrameworkThread">
48. <extensionDetail id="HttpThread" value="0.0"/>
49. <extensionDetail id="CatThread" value="0.0"/>
50. <extensionDetail id="PigeonThread" value="0.0"/>
51. <extensionDetail id="ActiveThread" value="11.0"/>
52. <extensionDetail id="StartedThread" value="12.0"/>
53. </extension>
54. <extension id="CatUsage">
55. <extensionDetail id="Produced" value="2.0"/>
56. <extensionDetail id="Overflowed" value="0.0"/>
57. <extensionDetail id="Bytes" value="1038.0"/>
58. </extension>
59. </status>

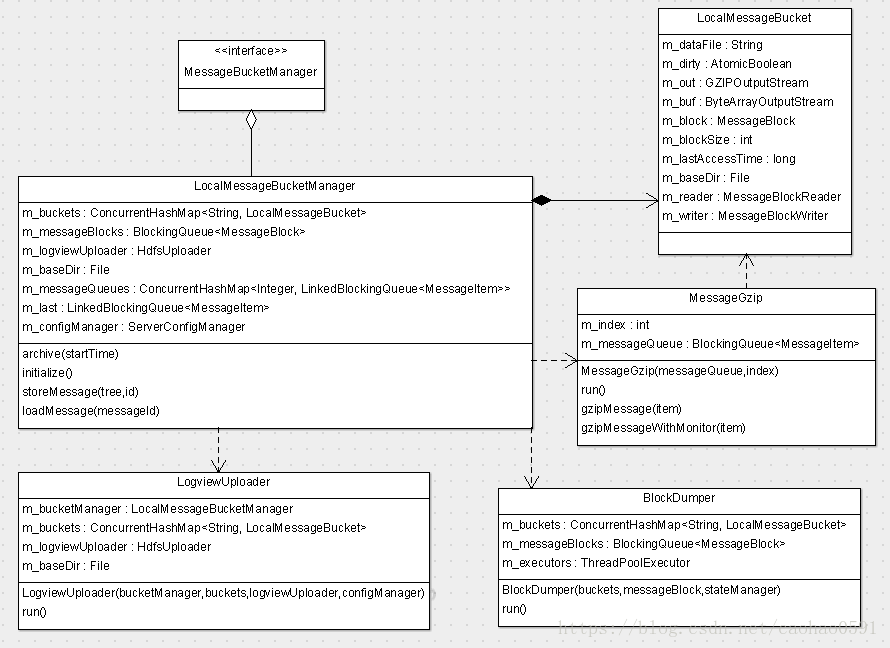
    我们再来看看服务端的分析逻辑，HeartbeatAnalyzer会为每个Domain创建一张心跳报表HeartbeatReport，不同IP的机器心跳数据存在于不同Machine对象里，每分钟的心跳数据都由一个Period对象存储；

    HeartbeatAnalyzer首先将XML转换为StatusInfo，然后会用StatusInfo的RuntimeInfo、OsInfo、DiskInfo、MemoryInfo、ThreadInfo、MessageInfo的信息以及Extensions的动态属性m\_dynamicAttributes去更新Period的m\_extensions。



## DumpAnalyzer -- 原始消息LogView存储

    DumpAnalyzer 与其它分析器有点不同，它不是为了报表而设计，而是用于原始消息LogView的存储，与报表统计不一样，他的数据量非常大，几年前美团点评每天处理消息就达到1000亿左右，大小大约100TB，单物理机高峰期每秒要处理100MB左右的流量，因为数据量比较大所以存储整体要求就是批量压缩以及随机读，采用队列化、异步化、线程池等技术来保证并发。



    当有客户端消息过来，DumpAnalyzer会调用本地消息处理器管理类(LocalMessageBucketManager) 的 storeMessage 方法存储消息，LocalMessageBucketManager是LogView管理的核心类，我们先看一看 LocalMessageBucketManager 对象的初始化函数 initialize() 的处理逻辑：

1、首先获取消息存储的基础路径(m\_baseDir)，默认是 /data/appdatas/cat/bucket/dump， 在 server.xml 中可以配置，消息在基础路径之内，会根据domain、机器、时间等元素来分门别类的存储。

2、开启 BlockDumper 线程， 将本地消息处理器(LocalMessageBucket)、阻塞队列(BlockingQueue)以及统计信息的指针传入BlockDumper 对象，当内存消息块到达 64K 的时候， 该线程会异步将内存消息块写入数据文件和索引文件。

3、开启LogviewUploader线程，将自己的指针、本地消息处理器、HDFS上传对象(HdfsUploader)以及配置管理器的指针传入LogviewUploader对象，该用于异步将文件上传到 HDFS， 前提是配置了 hdfs 上传配置。

4、开启20个消息压缩线程(本地模式仅2个线程)，并为每个线程分配一个阻塞队列，当DumpAnalyzer接收到消息请求，会将消息写入该队列，MessageGzip会轮训从队列取消息处理，注意这里虽然有20个队列，然而正常我们只插入前19个队列，只有在前面入队失败了，消息将会被插入最后那个队列，可以认为最后那个队列是前面队列的一个备用队列。

**[java]** [view plain](https://blog.csdn.net/caohao0591/article/details/80448847" \o "view plain) [copy](https://blog.csdn.net/caohao0591/article/details/80448847" \o "copy)

1. **public** **class** LocalMessageBucketManager **extends** ContainerHolder **implements** MessageBucketManager, Initializable, LogEnabled {
2. @Override
3. **public** **void** initialize() **throws** InitializationException {
4. m\_baseDir = **new** File(m\_configManager.getHdfsLocalBaseDir(ServerConfigManager.DUMP\_DIR));
6. Threads.forGroup("cat").start(**new** BlockDumper(m\_buckets, m\_messageBlocks, m\_serverStateManager));
7. Threads.forGroup("cat").start(**new** LogviewUploader(**this**, m\_buckets, m\_logviewUploader, m\_configManager));
9. **if** (m\_configManager.isLocalMode()) {
10. m\_gzipThreads = 2;
11. }
13. **for** (**int** i = 0; i < m\_gzipThreads; i++) {
14. LinkedBlockingQueue<MessageItem> messageQueue = **new** LinkedBlockingQueue<MessageItem>(m\_gzipMessageSize);
16. m\_messageQueues.put(i, messageQueue);
17. Threads.forGroup("cat").start(**new** MessageGzip(messageQueue, i));
18. }
19. m\_last = m\_messageQueues.get(m\_gzipThreads - 1);
20. }
22. @Override
23. **public** **void** storeMessage(**final** MessageTree tree, **final** MessageId id) {
24. **boolean** errorFlag = **true**;
25. **int** hash = Math.abs((id.getDomain() + '-' + id.getIpAddress()).hashCode());
26. **int** index = (**int**) (hash % m\_gzipThreads);
27. MessageItem item = **new** MessageItem(tree, id);
28. LinkedBlockingQueue<MessageItem> queue = m\_messageQueues.get(index % (m\_gzipThreads - 1));
29. **boolean** result = queue.offer(item);
30. ...
31. }
32. }

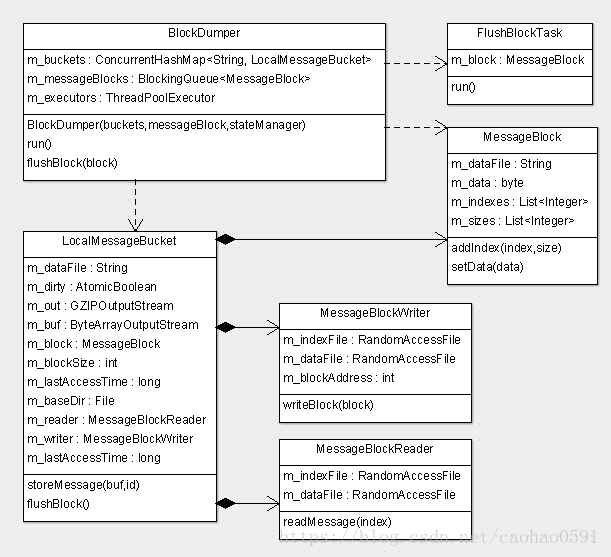
    当DumpAnalyzer接收到消息请求，会调用storeMessage(...) 函数处理消息，如上源码，函数会根据domain和客户端ip将消息均匀分配到那19个阻塞队列(LinkedBlockingQueue)中，然后MessageGzip会轮询从队列获取消息数据，调用gzipMessage(item)函数处理，每处理 10000 条消息，MessageGzip会上报一条Gzip压缩线程监控记录。

    我们再看看最核心的 gzipMessage(MessageItem item) 函数的处理逻辑，CAT根据日期，周期小时，domain，客户端地址，服务端地址创建存储路径和文件，包含数据文件和索引文件， 例如 20180611/15/Cat-127.0.01-127.0.01、20180611/15/Cat-127.0.01-127.0.01.idx ，从前面可以看出Message-ID的前3段可以确定唯一的索引文件，每条消息的存储由本地消息处理器(LocalMessageBucket)控制，LocalMessageBucket 的 storeMessage(...)方法会将消息信息写入消息块(MessageBlock)对象存放在内存中，当MessageBlock数据块大小达到 64K 时，将内存数据(MessageBlock) 放入阻塞队列 (m\_messageBlocks)，异步写入文件，并清空内存MessageBlock。LocalMessageBucket 有个字段 m\_blockSize 用于记录消息块总大小，注意这里的 64K 是压缩前的消息块总大小。

**[java]** [view plain](https://blog.csdn.net/caohao0591/article/details/80448847" \o "view plain) [copy](https://blog.csdn.net/caohao0591/article/details/80448847" \o "copy)

1. **public** **class** MessageGzip **implements** Task {
2. **private** **void** gzipMessage(MessageItem item) {
3. MessageId id = item.getMessageId();
4. String name = id.getDomain() + '-' + id.getIpAddress() + '-' + m\_localIp;
5. String path = m\_pathBuilder.getLogviewPath(**new** Date(id.getTimestamp()), name);
6. LocalMessageBucket bucket = m\_buckets.get(path);
8. **if** (bucket == **null**) {
9. **synchronized** (m\_buckets) {
10. bucket = m\_buckets.get(path);
11. **if** (bucket == **null**) {
12. bucket = (LocalMessageBucket) lookup(MessageBucket.**class**, LocalMessageBucket.ID);
13. bucket.setBaseDir(m\_baseDir);
14. bucket.initialize(path);
16. m\_buckets.put(path, bucket);
17. }
18. }
19. }
21. DefaultMessageTree tree = (DefaultMessageTree) item.getTree();
22. ByteBuf buf = tree.getBuffer();
23. MessageBlock block = bucket.storeMessage(buf, id);
25. **if** (block != **null**) {
26. **if** (!m\_messageBlocks.offer(block)) {
27. m\_serverStateManager.addBlockLoss(1);
28. }
29. }
30. }
31. }

    从上代码可以看出，当 storeMessage(...) 返回不为空的消息块(MessageBlock)时，则认为内存数据已经达到64K，需要写入文件，MessageGzip将消息块推入阻塞队列m\_messageBlocks， BlockDumper线程会对队列进行消费， 它在实例化的时候会创建一个执行线程池 m\_executors，然后 BlockDumper 线程轮询从阻塞队列取消息块(MessageBlock)，为每个消息块创建一个块写入任务(FlushBlockTask)，并将任务提交给执行线程池执行。FlushBlockTask实际会调用BlockDumper的flushBlock(block)函数将MessageBlock写入文件。



    最终写入操作，还是得由LocalMessageBucket的MessageBlockWriter来完成，接下来我们介绍下本地消息处理器(LocalMessageBucket)，它是一个控制消息数据读写的对象，数据在内存中的载体是消息块(MessageBlock)，LocalMessageBucket 在gzipMessage(...)被首次实例化、初始化，初始化过程中会创建一个消息块(MessageBlock)、消息块读处理对象(MessageBlockReader)、消息块写处理对象(MessageBlockWriter)、、缓冲区以及缓冲区压缩流。

     MessageBlock 包含4个信息：文件路径、数据缓冲区、每条ID的序列号、每条消息数据的大小(压缩前)。

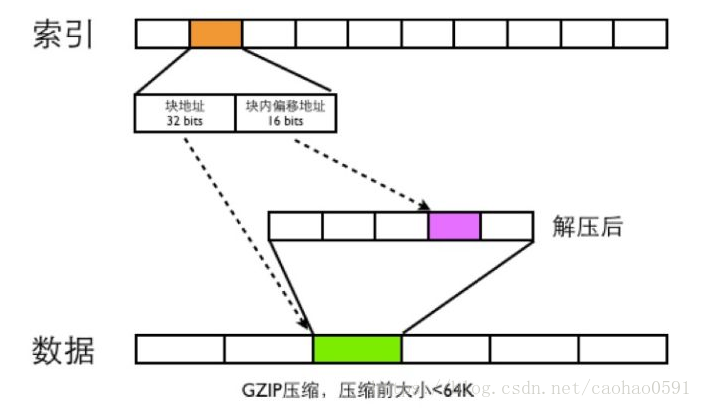
     消息块读处理对象负责消息的读取操作。

     消息块写处理对象则负责数据文件、索引文件的写入操作，他会维护一个文件游标偏移量，记录压缩消息块(MessageBlock)在数据文件中的起始位置，即图2中块地址，下面是具体的写逻辑，先写索引文件，CAT先获取消息块中消息总条数，为每个Message-ID都写一个索引记录，每条消息的索引记录长度都是48bits，索引根据Message-ID的第四段（序列号）来确定索引的位置，比如消息Message-ID为ShopWeb-0a010680-375030-2，这条消息ID对应的索引位置为2\*48bits的位置，因为不同客户端机器的消息是由不同文件来存储，所以不用担心序列号会重复，48bits索引包含32bits的块地址 和 16bits 的块内偏移地址，前者记录压缩消息块(MessageBlock)在数据文件中的偏移位置，由于消息块包含多条消息，我们需要16bits来记录消息在消息块中的位置，注意这里指解压后的消息块。写完索引文件再写入数据文件，每一段压缩数据，前4位都是压缩块的大小，后面才是消息块的实际数据。

**[java]** [view plain](https://blog.csdn.net/caohao0591/article/details/80448847" \o "view plain) [copy](https://blog.csdn.net/caohao0591/article/details/80448847" \o "copy)

1. **public** **class** MessageBlockWriter {
2. **private** RandomAccessFile m\_indexFile;
3. **private** RandomAccessFile m\_dataFile;
4. **private** **int** m\_blockAddress;
6. **public** **synchronized** **void** writeBlock(MessageBlock block) **throws** IOException {
7. **int** len = block.getBlockSize();
8. **byte**[] data = block.getData();
9. **int** blockSize = 0;
11. **for** (**int** i = 0; i < len; i++) {
12. **int** seq = block.getIndex(i);
13. **int** size = block.getSize(i);
15. m\_indexFile.seek(seq \* 6L);
16. m\_indexFile.writeInt(m\_blockAddress);
17. m\_indexFile.writeShort(blockSize);
18. blockSize += size;
19. }
21. m\_dataFile.writeInt(data.length);
22. m\_dataFile.write(data);
23. m\_blockAddress += data.length + 4;
24. }
25. }

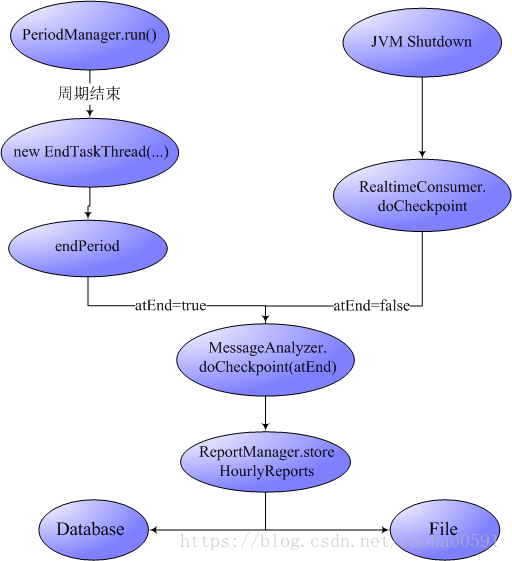
    CAT读取消息的时候，首先根据Message-ID的前面三段确定唯一的索引文件，在根据Message-ID第四段确定此Message-ID索引位置，根据索引文件的48bits读取数据文件的内容，然后将数据文件进行GZIP解压，在根据块内偏移地址读取出真正的消息内容。



## 自定义分析器与报表

# 第八章 数据持久化

## 周期结束



    我们从消息分发章节知道，RealtimeConsumer在初始化的时候，会启动一个线程，每隔1秒钟就去从判断是否需要开启或结束一个周期(Period)，如下源码，如果 value < 0 的时候，就会启动一个周期结束线程，线程会调用endPeriod函数，找到需要结束的周期，完成周期的结束以及清理工作，并将周期对象从PeriodManager中移除。

**[java]** [view plain](https://blog.csdn.net/caohao0591/article/details/80558779" \o "view plain) [copy](https://blog.csdn.net/caohao0591/article/details/80558779" \o "copy)

1. **public** **class** PeriodManager **implements** Task {
2. **private** List<Period> m\_periods = **new** ArrayList<Period>();
4. @Override
5. **public** **void** run() {
6. **while** (m\_active) {
7. **try** {
8. **long** now = System.currentTimeMillis();
9. **long** value = m\_strategy.next(now);
11. **if** (value > 0) {
12. startPeriod(value);
13. } **else** **if** (value < 0) {
14. // last period is over,make it asynchronous
15. Threads.forGroup("cat").start(**new** EndTaskThread(-value));
16. }
17. }
18. ...
19. }
20. }
22. **private** **void** endPeriod(**long** startTime) {
23. **int** len = m\_periods.size();
24. **for** (**int** i = 0; i < len; i++) {
25. Period period = m\_periods.get(i);
27. **if** (period.isIn(startTime)) {
28. period.finish();
29. m\_periods.remove(i);
30. **break**;
31. }
32. }
33. }
35. **private** **class** EndTaskThread **implements** Task {
36. **public** **void** run() { endPeriod(m\_startTime); }
37. }
38. }

    我们知道，周期是由许多的周期任务(PeriodTask)构成，所以事实上，一个周期的结束，就是周期内所有周期任务的结束，每个周期任务对应着一个任务队列和一个消息分析器(MessageAnalyzer)，归根结底是对MessageAnalyzer的结束。

**[java]** [view plain](https://blog.csdn.net/caohao0591/article/details/80558779" \o "view plain) [copy](https://blog.csdn.net/caohao0591/article/details/80558779" \o "copy)

1. **public** **class** PeriodTask **implements** Task, LogEnabled {
2. **private** MessageAnalyzer m\_analyzer;
4. **public** **void** finish() {
5. **try** {
6. m\_analyzer.doCheckpoint(**true**);
7. m\_analyzer.destroy();
8. } **catch** (Exception e) {
9. Cat.logError(e);
10. }
11. }
12. }

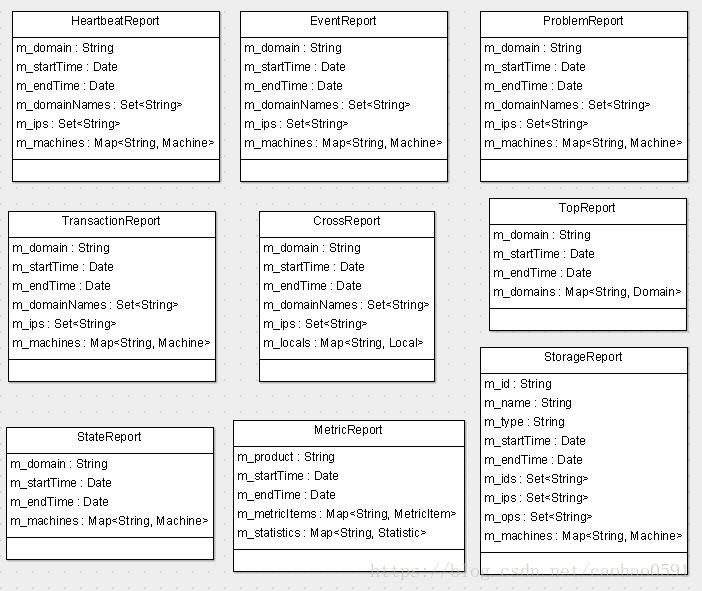
        doCheckpoint 我们似曾相识，在CatHomeModule初始化的最后，我们会向虚拟机注册shutdownhook，保证在虚拟机关闭时，未被正常结束的周期会被RealtimeConsumer结束，RealtimeConsumer.doCheckpoint与上面正常结束周期所做的工作是一样的，都是调用分析器的doCheckpoint方法，唯一的区别是，分析器doCheckpoint函数的传入的atEnd参数不同，表示周期是否在到期后正常结束的。

**[java]** [view plain](https://blog.csdn.net/caohao0591/article/details/80558779" \o "view plain) [copy](https://blog.csdn.net/caohao0591/article/details/80558779" \o "copy)

1. Runtime.getRuntime().addShutdownHook(**new** Thread() {
2. @Override
3. **public** **void** run() {
4. consumer.doCheckpoint();
5. }
6. });

## 分析器的结束 -- 报表持久化

    分析器的结束实际上就是报表的持久化的一个过程，分析器处理消息的过程中，我们一共形成了9个报表，图1展示了这9个报表的结构：



    我们知道每个周期的消息分析器(MessageAnalyzer)的结束都是在doCheckpoint来实现的，实际运行中一共有10种消息分析器参与消息分析工作，那么不同类别的分析器，他的结束逻辑都是一样的吗？

    除了几个特殊的分析器(如Metric、Dump)之外，其它消息分析器结束逻辑都同下面源码，调用storeHourlyReports方法存储报表，所有报表都会被存到文件，atEnd 和 localMode 参数决定我们是否将报表存到数据库。

    其中State报表有点特殊，因为State是对CAT本身的监控，在周期任务(PeriodTask)运行过程中并没有收集任何数据，而是在doCheckpoint的时候对CAT消息监控情况汇总生成的一个报表，所以调用storeHourlyReports之前，他需要首先收集State报表数据。

**[java]** [view plain](https://blog.csdn.net/caohao0591/article/details/80558779" \o "view plain) [copy](https://blog.csdn.net/caohao0591/article/details/80558779" \o "copy)

1. **public** **class** XxxAnalyzer **extends** AbstractMessageAnalyzer<XxxReport> **implements** LogEnabled {
2. **private** ReportManager<XxxReport> m\_reportManager;
4. **public** **synchronized** **void** doCheckpoint(**boolean** atEnd) {
5. **if** (atEnd && !isLocalMode()) {
6. m\_reportManager.storeHourlyReports(getStartTime(), StoragePolicy.FILE\_AND\_DB, m\_index);
7. } **else** {
8. m\_reportManager.storeHourlyReports(getStartTime(), StoragePolicy.FILE, m\_index);
9. }
10. }
11. }

    我们先详细剖析storeHourlyReports 的过程，然后再看看几个特殊的分析器的结束逻辑。

        我们首先将该分析器得到的所有报表都取出，然后我们会校验报表的domain名称是否合法，不合法的报表将被移除，在序列化之前，我们会调用ReportDelegate.beforeSave方法做一些预处理的工作。不同种类的报表，预处理所做的工作是不同的，后续我们分别讲解，做完预处理的工作之后，我们就正式持久化了，支持文件和数据库两种持久化方式，我们会根据传入的序列化策略(StoragePolicy) 来选择需要进行哪种序列化，一般来说，如果是正常的周期结束，数据会持久化到文件和数据库，如果是JVM Shutdown导致的结束，只持久化到文件，两种持久化的细节后续我们也会分别详细讲解。

**[java]** [view plain](https://blog.csdn.net/caohao0591/article/details/80558779" \o "view plain) [copy](https://blog.csdn.net/caohao0591/article/details/80558779" \o "copy)

1. **public** **class** DefaultReportManager<T> **extends** ContainerHolder **implements** ReportManager<T>, Initializable, LogEnabled {
2. @Override
3. **public** **void** storeHourlyReports(**long** startTime, StoragePolicy policy, **int** index) {
4. Map<String, T> reports = m\_reports.get(startTime);
5. ReportBucket bucket = **null**;
7. **try** {
8. **if** (reports != **null**) {
9. //校验、移除不合法Domain名字的报表
10. ...
12. m\_reportDelegate.beforeSave(reports);
14. **if** (policy.forFile()) {
15. bucket = m\_bucketManager.getReportBucket(startTime, m\_name, index);
17. **try** {
18. storeFile(reports, bucket);
19. } **finally** {
20. m\_bucketManager.closeBucket(bucket);
21. }
22. }
24. **if** (policy.forDatabase()) {
25. storeDatabase(startTime, reports);
26. }
27. }
28. } **catch** (Throwable e) {
29. //报告异常
30. ...
31. } **finally** {
32. cleanup(startTime);
33. t.complete();
35. **if** (bucket != **null**) {
36. m\_bucketManager.closeBucket(bucket);
37. }
38. }
39. }
40. }

## 报表预处理

    在继续讲解序列化之前，我们来说一说报表的预处理工作(beforeSave)，各报表的预处理逻辑，有相同、也有异同的地方，Top、State报表预处理不做任何事情，其它报表都有处理逻辑，以Transaction报表的预处理工作为例，分为两个部分。

      第一部分是将所有Transaction报表的domain都收集起来，写入报表的m\_domainNames属性，这样每个报表都会知道一共都有哪些domain参与监控， 从图1的报表结构来看，Heartbeat、Event、Problem、Cross报表也包含m\_domainNames字段，事实上这些报表的预处理也确实会收集所有domain。其中Heartbeat和Cross的预处理仅仅包含这部分逻辑。

      第二部分是聚合报表，所谓聚合，就是创建一个命名为ALL的聚合报表，将同一个Domain下不同IP地址的数据汇总起来，写到报表ALL的同一个Machine对象内，Machine的ip不再是地址，而是Domain名，所有Domain数据都汇总到ALL报表的不同Machine下。因为现在服务端几乎都是采用集群，有可能10几台机器上运行着同一个项目，这时我们可以通过聚合报表去站在项目角度去看待统计结果。

    也不是所有类型的事务都会参与聚合，配置 all-report-config 会指定哪些事务会参与聚合，如下默认type="URL"的事务，这是因为通常URL是代表一个项目的接口对外服务的最完整链路耗时，从以下配置可以看到，除了Transaction消息之外，Event消息也会参与聚合，逻辑与Transaction大同小异，在此不再赘述。

**[java]** [view plain](https://blog.csdn.net/caohao0591/article/details/80558779" \o "view plain) [copy](https://blog.csdn.net/caohao0591/article/details/80558779" \o "copy)

1. <all-config>
2. <report id="transaction">
3. <type id="URL">
4. <name id="\*"></name>
5. </type>
6. </report>
7. <report id="event">
8. <type id="URL">
9. <name id="\*"></name>
10. </type>
11. <type id="SQL">
12. <name id="\*"></name>
13. </type>
14. </report>
15. </all-config>

    Problem有一个独有的预处理流程，就是通过ProblemReportFilter对象将长时URL访问(long-url)的记录总数控制在100条之内，防止长时访问数量过多，导致报表数据过大。

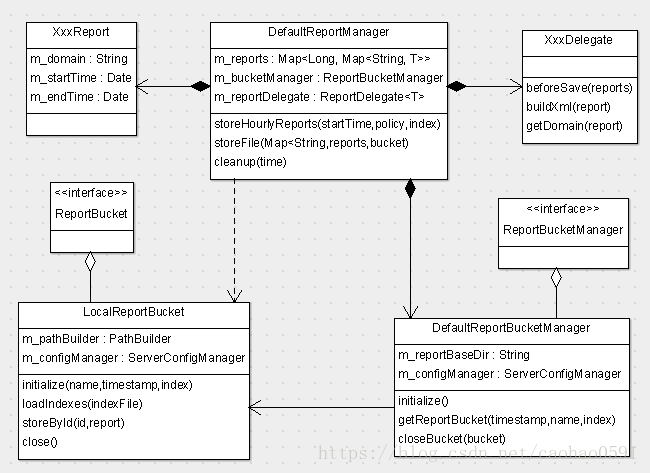
   Storage报表的预处理只有一个 updateStorageIds 调用，他的功能和Transaction预处理第一部分类似，也是让每个Storage报表都知道目前有哪些数据库/缓存在被访问、监控，我们知道Storage报表的ID是由 数据库/缓存名 + 类型(SQL/Cache) 组成，updateStorageIds会将所有数据库/缓存名收集，然后写到StorageReport的m\_ids属性。

    下面将所有报表持久化的预处理做了一个汇总，放在一个函数内，呈现在以下伪代码中：

**[java]** [view plain](https://blog.csdn.net/caohao0591/article/details/80558779" \o "view plain) [copy](https://blog.csdn.net/caohao0591/article/details/80558779" \o "copy)

1. **public** **class** XxxDelegate **implements** ReportDelegate<XxxReport> {
2. @Override
3. **public** **void** beforeSave(Map<String, XxxReport> reports) {
4. //Top、State不干任何事直接返回
5. **return**;
7. //storage 仅有下面 updateStorageIds 步骤，完成后返回。
8. **for** (StorageReport report : reports.values()) {
9. m\_reportUpdater.updateStorageIds(report.getId(), reports.keySet(), report);
10. }
12. //Problem、Transaction、Event、Heartbeat、Cross都有的步骤
13. **for** (XxxReport report : reports.values()) {
14. Set<String> domainNames = report.getDomainNames();
16. domainNames.clear();
17. domainNames.addAll(reports.keySet());
18. }
20. //报表聚合，Transaction、Event 独有
21. **if** (reports.size() > 0) {
22. TransactionReport all = createAggregatedReport(reports);
24. reports.put(all.getDomain(), all);
25. }
27. //Problem独有，控制long-url消息量。
28. ProblemReportFilter problemReportURLFilter = **new** ProblemReportFilter();
30. **for** (Entry<String, ProblemReport> entry : reports.entrySet()) {
31. ProblemReport report = entry.getValue();
33. problemReportURLFilter.visitProblemReport(report);
34. }
35. }
36. }

## 报表的文件存储 -- 重入锁

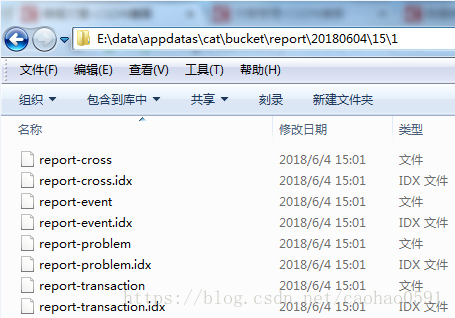


    在做完预处理之后，所有报表都将被持久化到文件，在DefaultReportManager调用storeFile存储文件之前，我们先调用 m\_bucketManager.getReportBucket(...) 来创建并初始化ReportBucket，文件的读写相关操作都封装于ReportBucket里面，文件的读写同步采用重入锁(ReentrantLock)保证读写安全。

**[java]** [view plain](https://blog.csdn.net/caohao0591/article/details/80558779" \o "view plain) [copy](https://blog.csdn.net/caohao0591/article/details/80558779" \o "copy)

1. **public** **class** LocalReportBucket **implements** ReportBucket, LogEnabled {
2. @Override
3. **public** **void** initialize(String name, Date timestamp, **int** index) **throws** IOException {
4. m\_baseDir = m\_configManager.getHdfsLocalBaseDir("report");
5. m\_writeLock = **new** ReentrantLock();
6. m\_readLock = **new** ReentrantLock();
8. String logicalPath = m\_pathBuilder.getReportPath(name, timestamp, index);
10. File dataFile = **new** File(m\_baseDir, logicalPath);
11. File indexFile = **new** File(m\_baseDir, logicalPath + ".idx");
13. **if** (indexFile.exists()) {
14. loadIndexes(indexFile);
15. }
17. **final** File dir = dataFile.getParentFile();
19. **if** (!dir.exists() && !dir.mkdirs()) {
20. **throw** **new** IOException(String.format("Fail to create directory(%s)!", dir));
21. }
23. m\_logicalPath = logicalPath;
24. m\_writeDataFile = **new** BufferedOutputStream(**new** FileOutputStream(dataFile, **true**), 8192);
25. m\_writeIndexFile = **new** BufferedOutputStream(**new** FileOutputStream(indexFile, **true**), 8192);
26. m\_writeDataFileLength = dataFile.length();
27. m\_readDataFile = **new** RandomAccessFile(dataFile, "r");
28. }
29. }

      报表存储基础路径(m\_baseDir)在配置 server.xml 中指定，每个分析器实例都会最终生成若干个报表，我们会为这个分析器产生的这些报表生成一个数据文件和报表索引文件，存于逻辑路径(logicalPath )下，逻辑路径以日期/小时/index来划分，例如：20180604/15/1/report-cross ， 20180604为日期， 15为下午3点的周期，1是分析器实例index，之前说过有些分析器处理过程复杂，可能会有多个实例，例如Cross、Event、Problem、Transaction报表， 数据文件名取 m\_baseDir + logicalPath， 索引文件是在数据文件名加上 ".idx" 后缀，如下：



    数据文件存储该分析器下所有转化为xml格式的报表数据，索引文件是对数据文件内报表的一个位置索引，比如 report-problem.idx索引文件内容如下，每一行都记录一个报表名称和报表在数据文件的起始位置。

**[plain]** [view plain](https://blog.csdn.net/caohao0591/article/details/80558779" \o "view plain) [copy](https://blog.csdn.net/caohao0591/article/details/80558779" \o "copy)

1. RpcService  0
2. Cat         1388
3. RpcClient2  4600
4. RpcClient   5807

    现在再来看看storeFile的逻辑就非常简单了， 获取domain，将报表对象转化为xml数据，最后调用storeById将xml写入数据文件和索引文件。

**[plain]** [view plain](https://blog.csdn.net/caohao0591/article/details/80558779" \o "view plain) [copy](https://blog.csdn.net/caohao0591/article/details/80558779" \o "copy)

1. public class DefaultReportManager<T> extends ContainerHolder implements ReportManager<T>, Initializable, LogEnabled {
2. private void storeFile(Map<String, T> reports, ReportBucket bucket) {
3. for (T report : reports.values()) {
4. try {
5. String domain = m\_reportDelegate.getDomain(report);
6. String xml = m\_reportDelegate.buildXml(report);
8. bucket.storeById(domain, xml);
9. } catch (Exception e) {
10. Cat.logError(e);
11. }
12. }
13. }
14. }

## 报表的数据库存储

    如果是正常的周期结束之后，发起的持久化，而不是由于虚拟机关闭引起的，数据除了被持久化到文件之外，还会被持久化到数据库。



    所有的数据库持久化逻辑都在 storeDatabase(...) 方法中完成，每个分析器中的每个报表的描述信息，都会被插入数据库report表中，在程序中，HourlyReport实体与该表对应，如图3， 报表具体内容会通过m\_reportDelegate.buildBinary(report)转化成二进制数据，然后插入数据库report\_content 表，在程序中，HourlyReportContent实体与该表对应，如图4，report\_content 表的主键来自report的主键。

**[plain]** [view plain](https://blog.csdn.net/caohao0591/article/details/80558779" \o "view plain) [copy](https://blog.csdn.net/caohao0591/article/details/80558779" \o "copy)

1. public class DefaultReportManager<T> extends ContainerHolder implements ReportManager<T>, Initializable, LogEnabled {
2. @Inject
3. private HourlyReportDao m\_reportDao;
5. @Inject
6. private HourlyReportContentDao m\_reportContentDao;
8. private void storeDatabase(long startTime, Map<String, T> reports) {
9. Date period = new Date(startTime);
10. String ip = NetworkInterfaceManager.INSTANCE.getLocalHostAddress();
12. for (T report : reports.values()) {
13. try {
14. String domain = m\_reportDelegate.getDomain(report);
15. HourlyReport r = m\_reportDao.createLocal();
17. r.setName(m\_name);
18. r.setDomain(domain);
19. r.setPeriod(period);
20. r.setIp(ip);
21. r.setType(1);
23. m\_reportDao.insert(r);
25. int id = r.getId();
26. byte[] binaryContent = m\_reportDelegate.buildBinary(report);
27. HourlyReportContent content = m\_reportContentDao.createLocal();
29. content.setReportId(id);
30. content.setContent(binaryContent);
31. m\_reportContentDao.insert(content);
32. m\_reportDelegate.createHourlyTask(report);
33. } catch (Throwable e) {
34. Cat.getProducer().logError(e);
35. }
36. }
37. }
38. }

**[sql]** [view plain](https://blog.csdn.net/caohao0591/article/details/80558779" \o "view plain) [copy](https://blog.csdn.net/caohao0591/article/details/80558779" \o "copy)

1. **CREATE** **TABLE** `report` (
2. `id` **int**(11) NOT NULL AUTO\_INCREMENT,
3. `type` tinyint(4) NOT NULL COMMENT '报表类型, 1/xml, 9/binary 默认1',
4. `**name**` **varchar**(20) NOT NULL COMMENT '报表名称',
5. `ip` **varchar**(50) **DEFAULT** NULL COMMENT '报表来自于哪台机器',
6. `domain` **varchar**(50) NOT NULL COMMENT '报表项目',
7. `period` datetime NOT NULL COMMENT '报表时间段',
8. `creation\_date` datetime NOT NULL COMMENT '报表创建时间',
9. **PRIMARY** **KEY** (`id`),
10. **KEY** `IX\_Domain\_Name\_Period` (`domain`,`**name**`,`period`),
11. **KEY** `IX\_Name\_Period` (`**name**`,`period`),
12. **KEY** `IX\_Period` (`period`)
13. ) ENGINE=InnoDB AUTO\_INCREMENT=18497 **DEFAULT** CHARSET=utf8 ROW\_FORMAT=COMPRESSED COMMENT='用于存放实时报表信息，处理之后的结果';

**[sql]** [view plain](https://blog.csdn.net/caohao0591/article/details/80558779" \o "view plain) [copy](https://blog.csdn.net/caohao0591/article/details/80558779" \o "copy)

1. **CREATE** **TABLE** `report\_content` (
2. `report\_id` **int**(11) NOT NULL COMMENT '报表ID',
3. `content` longblob NOT NULL COMMENT '二进制报表内容',
4. `creation\_date` datetime NOT NULL COMMENT '创建时间',
5. **PRIMARY** **KEY** (`report\_id`)
6. ) ENGINE=InnoDB **DEFAULT** CHARSET=utf8 ROW\_FORMAT=COMPRESSED COMMENT='小时报表二进制内容';

## 定时任务

任务生产者

   数据库的持久化完成标志着一个完整周期的结束，CAT实时处理报表都是产生小时级别统计，小时级报表中会带有最低分钟级别粒度的统计，在数据库持久化完成之后，我们会调用 m\_reportDelegate.createHourlyTask(report) 创建一些定时任务，去创建小时模式、天模式、周模式、月模式等等粒度更粗的视图，为什么这里还会创建小时任务，因为在集群情况下，同一周期下的多张报表可能分散在几台CAT服务器上，这时我们创建小时定时任务去合并报表形成小时视图。

        但是，针对不同报表、不同domain，创建的定时任务也不同，有些可能小时模式、天模式、周模式、月模式视图定时任务都有，有些也可能只创建天任务，在解释完以下几个domain的描述之后我们看下定时任务的列表：

* crashLogDomain：客户端崩溃日志埋点，默认有：AndroidCrashLog/iOSCrashLog/MerchantAndroidCrashLog/MerchantIOSCrashLog/ApolloAndroidCrashLog/ApolloIOSCrashLog/TVAndroidCrashLog
* serverFilterDomain：配置serverFilter中过滤的domain，默认有：空/PhoenixAgent/cat-agent/All/FrontEnd/paas/SMS-RECEIVER
* validateDomain：非ServerFilterLog 和 非CrashLog
* ALL：聚合报表，Transaction和Event特有。
* \*：所有domain

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| ReportType | 任务名 | Domain/Task | Hourly | Daily | Weekly | Month |
| Problem | Problem | validateDomain | √ | √ | √ | √ |
| crashLogDomain | × | √ | √ | √ |
| 其它 | × | × | × | × |
| Transaction | Transaction | ALL | × | √ | √ | √ |
| validateDomain | √ | √ | √ | √ |
| 其它 | × | × | × | × |
| Event | Event | ALL | × | √ | √ | √ |
| validateDomain | √ | √ | √ | √ |
| 其它 | × | × | × | × |
| Top | Top | \* | × | × | × | × |
| Heartbeat | Heartbeat | validateDomain | × | √ | × | × |
| 其它 | × | × | × | × |
| Cross | Cross | validateDomain | × | √ | √ | √ |
| 其它 | × | × | × | × |
| Storage | Storage | validateDomain | × | √ | √ | √ |

State比较特殊，他会创建比较多的定时任务，我们单独列在下面：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 任务名 | Hourly | Daily | Weekly | Month |
| State | √ | √ | √ | √ |
| appDatabasePruner | × | √ | × | × |
| cmdb | √ | × | × | × |
| NetTopology | √ | × | × | × |
| bug | √ | √ | √ | √ |
| databaseCapacity | √ | √ | √ | √ |
| jar | √ | × | × | × |
| heavy | √ | √ | √ | √ |
| utilization | √ | √ | √ | √ |
| service | √ | √ | √ | √ |
| dailyNotify | × | √ | × | × |
| router | × | √ | × | × |
| cachedReport | × | √ | × | × |
| system | × | √ | × | × |
| app | × | √ | × | × |

    定时任务在 TaskManager.createTask(...) 中生产，这里就是将需要执行的定时任务插入数据库task表中，以供消费者(TaskConsumer)到时候去表里取定时任务然后执行，表字段如下：

**[sql]** [view plain](https://blog.csdn.net/caohao0591/article/details/80558779" \o "view plain) [copy](https://blog.csdn.net/caohao0591/article/details/80558779" \o "copy)

1. **TABLE** `task` (
2. `id` **int**(11) NOT NULL AUTO\_INCREMENT,
3. `producer` **varchar**(20) NOT NULL COMMENT '任务创建者ip',
4. `consumer` **varchar**(20) **DEFAULT** NULL COMMENT '任务执行者ip',
5. `failure\_count` tinyint(4) NOT NULL COMMENT '任务失败次数',
6. `report\_name` **varchar**(20) NOT NULL COMMENT '报表名称, transaction, problem...',
7. `report\_domain` **varchar**(50) NOT NULL COMMENT '报表处理的Domain信息',
8. `report\_period` datetime NOT NULL COMMENT '报表时间',
9. `status` tinyint(4) NOT NULL COMMENT '执行状态: 1/todo, 2/doing, 3/done 4/failed',
10. `task\_type` tinyint(4) NOT NULL **DEFAULT** '1' COMMENT '0表示小时任务，1表示天任务',
11. `creation\_date` datetime NOT NULL COMMENT '任务创建时间',
12. `start\_date` datetime **DEFAULT** NULL COMMENT '开始时间, 这次执行开始时间',
13. `end\_date` datetime **DEFAULT** NULL COMMENT '结束时间, 这次执行结束时间',
14. **PRIMARY** **KEY** (`id`),
15. **UNIQUE** **KEY** `task\_period\_domain\_name\_type` (`report\_period`,`report\_domain`,`report\_name`,`task\_type`)
16. ) ENGINE=InnoDB AUTO\_INCREMENT=42594 **DEFAULT** CHARSET=utf8 COMMENT='后台任务';

    其中producer就是生产定时任务的机器IP，stauts是执行状态，这里TaskManager作为生产者插入的记录状态都是todo=1，task\_type指的任务类别，包含下面4种类别：

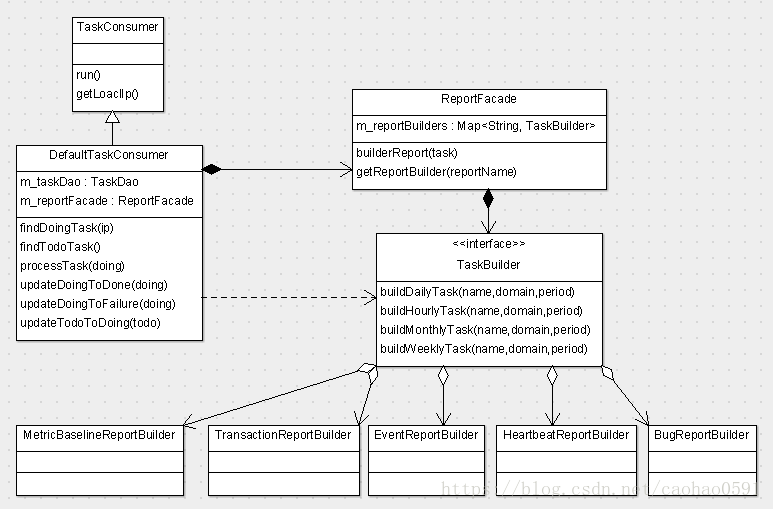
* 0-小时任务，有小时任务需求的在生产者中都会被创建，合并CAT服务器集群的多台机器的小时报表。
* 1-天任务， 对于有天任务需求的，会在当天创建前一天的天视图，
* 2-周任务，有周任务需求的会创建上周六到这周五的周视图，
* 3-月任务，有月任务需求的会在每月1号创建上个月的月视图。

    当然由于task表的report\_period, report\_domain, report\_name, task\_type 是联合唯一键，所以，同一个类型、周期、domain、名称的定时任务，只会插入一条。

**[java]** [view plain](https://blog.csdn.net/caohao0591/article/details/80558779" \o "view plain) [copy](https://blog.csdn.net/caohao0591/article/details/80558779" \o "copy)

1. **public** **class** TaskManager {
2. @Inject
3. **private** TaskDao m\_taskDao;
5. **private** **static** **final** **int** STATUS\_TODO = 1;
7. **public** **static** **final** **int** REPORT\_HOUR = 0;
8. **public** **static** **final** **int** REPORT\_DAILY = 1;
9. **public** **static** **final** **int** REPORT\_WEEK = 2;
10. **public** **static** **final** **int** REPORT\_MONTH = 3;
12. **public** **boolean** createTask(Date period, String domain, String name, TaskCreationPolicy prolicy) {
13. **try** {
14. **if** (prolicy.shouldCreateHourlyTask()) {
15. createHourlyTask(period, domain, name);
16. }
18. Calendar cal = Calendar.getInstance();
19. cal.setTime(period);
21. **int** hour = cal.get(Calendar.HOUR\_OF\_DAY);
22. cal.add(Calendar.HOUR\_OF\_DAY, -hour);
23. Date currentDay = cal.getTime();
25. **if** (prolicy.shouldCreateDailyTask()) {
26. createDailyTask(**new** Date(currentDay.getTime() - ONE\_DAY), domain, name);
27. }
29. **if** (prolicy.shouldCreateWeeklyTask()) {
30. **int** dayOfWeek = cal.get(Calendar.DAY\_OF\_WEEK);
31. **if** (dayOfWeek == 7) {
32. createWeeklyTask(**new** Date(currentDay.getTime() - 7 \* ONE\_DAY), domain, name);
33. }
34. }
35. **if** (prolicy.shouldCreateMonthTask()) {
36. **int** dayOfMonth = cal.get(Calendar.DAY\_OF\_MONTH);
38. **if** (dayOfMonth == 1) {
39. cal.add(Calendar.MONTH, -1);
40. createMonthlyTask(cal.getTime(), domain, name);
41. }
42. }
43. **return** **true**;
44. } **catch** (DalException e) {
45. Cat.logError(e);
46. **return** **false**;
47. }
48. }
49. }

任务消费者：



    我们再次回到CatHomeModule的初始化函数中，有如下一段代码，它会读取server.xml中的配置 job-machine="true"，用于指定是否开启定时任务消费者线程。

**[sql]** [view plain](https://blog.csdn.net/caohao0591/article/details/80558779" \o "view plain) [copy](https://blog.csdn.net/caohao0591/article/details/80558779" \o "copy)

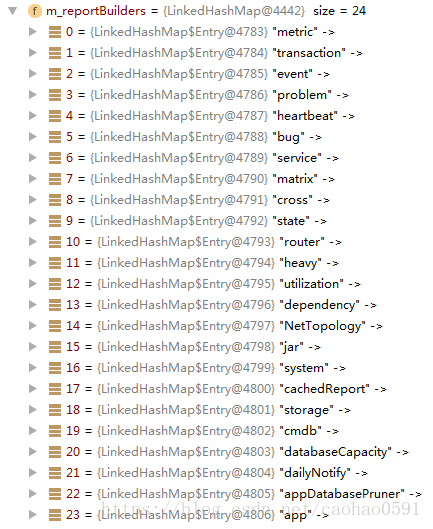
1. if (serverConfigManager.isJobMachine()) {
2. DefaultTaskConsumer taskConsumer = ctx.lookup(DefaultTaskConsumer.class);
4. Threads.forGroup("cat").start(taskConsumer);
5. }

    线程会每隔1分钟轮训从数据库取状态为todo的定时任务，以及consumer为本机ip，然后状态为doing的定时任务，即上次处理失败，需要重试的，将任务状态都修改为 doing，然后调用 processTask处理定时任务，如果处理失败则间隔一段时间后重试，注意，这里的间隔会阻塞任务线程，超过最大重试次数，状态标为failed，成功则标为done。

**[java]** [view plain](https://blog.csdn.net/caohao0591/article/details/80558779" \o "view plain) [copy](https://blog.csdn.net/caohao0591/article/details/80558779" \o "copy)

1. **public** **abstract** **class** TaskConsumer **implements** org.unidal.helper.Threads.Task {
2. @Override
3. **public** **void** run() {
4. String localIp = getLoaclIp();
5. **while** (running) {
6. **if** (checkTime()) {
7. Task task = findDoingTask(localIp);
8. **if** (task == **null**) {
9. task = findTodoTask();
10. }
11. **boolean** again = **false**;
12. **if** (task != **null**) {
13. task.setConsumer(localIp);
14. **if** (task.getStatus() == TaskConsumer.STATUS\_DOING || updateTodoToDoing(task)) {
15. **int** retryTimes = 0;
16. **while** (!processTask(task)) {
17. retryTimes++;
18. **if** (retryTimes < MAX\_TODO\_RETRY\_TIMES) {
19. taskRetryDuration();
20. } **else** {
21. updateDoingToFailure(task);
22. again = **true**;
23. **break**;
24. }
25. }
26. **if** (!again) {
27. updateDoingToDone(task);
28. }
29. }
30. } **else** {
31. taskNotFoundDuration(); //sleep 2 min
32. }
33. } **else** {
34. Thread.sleep(60 \* 1000);
35. }
36. }
37. **this**.stopped = **true**;
38. }
39. }

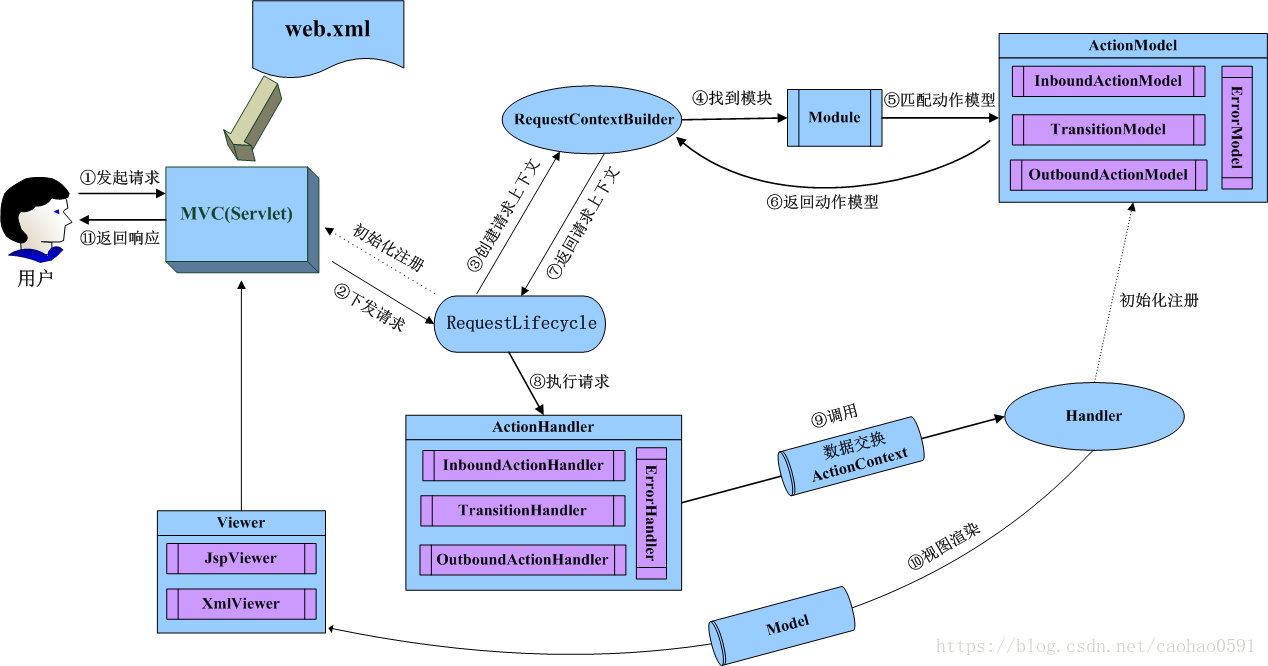
    processTask(...)处理的核心是将Task交给ReportFacade去构建视图，我们可以认为ReportFacade是一个视图构建工厂，工厂在初始化的时候，从plexus配置中读取所有的任务构建器(TaskBuilder)，并将他们装入ReportFacade的属性m\_reportBuilders中，TaskBuilder是一个接口，有4个方法，buildHourlyTask、buildDailyTask、buildWeeklyTask、buildMonthlyTask，我们从图3可以看到该接口一共有24个实现，当有定时任务交付时，ReportFacade会根据任务名找到具体的任务构建类，然后根据任务是小时、天、周还是月分别调用以上4个方法。



# 第九章 CAT管理平台MVC框架

## Servlet容器与请求生命周期

在第2章我们讲到，服务器在初始化CatServlet 之后， 会初始化 MVC，MVC也是继承自AbstractContainerServlet ， 同样也是一个 Servlet 容器，这是一个非常古老的MVC框架，当时Spring MVC 还并不成熟，但是所有MVC框架的核心思想都是一致的。



       在初始化完CatServlet之后，我们就会调用 MVC 的父类 AbstractContainerServlet 去初始化MVC容器，初始化最核心的步骤是调用MVC子类的initComponents(...) 函数去创建请求周期管理器(RequestLifecycle)，并将容器上下文(ServletContext)的指针传递给 RequestLifecycle。容器类RequestLifecycle 的配置信息位于文件 web-framework-4.0.0.jar/META-INF/plexus/components-web-framework.xml 中。

public class MVC extends AbstractContainerServlet {

protected void initComponents(ServletConfig config) throws Exception {

if(this.m\_handler == null) {

String contextPath = config.getServletContext().getContextPath();

String path = contextPath != null && contextPath.length() != 0?contextPath:"/";

this.getLogger().info("MVC is starting at " + path);

this.initializeCat(config);

this.initializeModules(config);

this.m\_handler = (RequestLifecycle)this.lookup(RequestLifecycle.class, "mvc");

this.m\_handler.setServletContext(config.getServletContext());

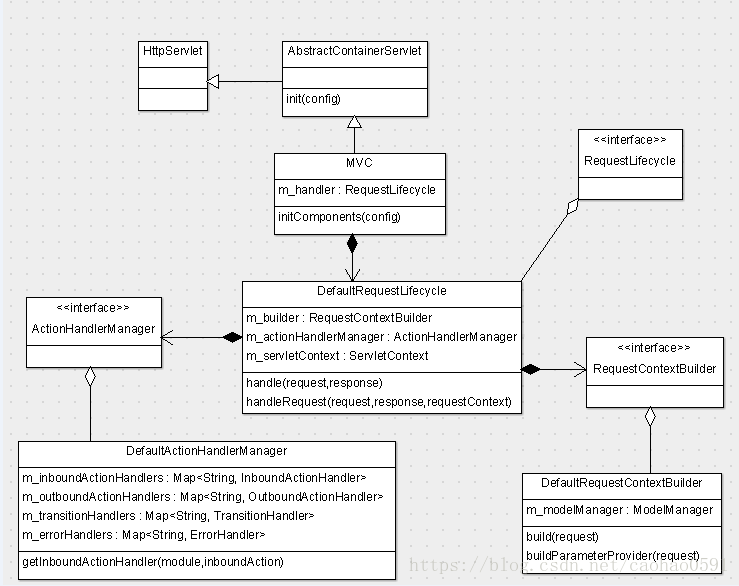
config.getServletContext().setAttribute("mvc-servlet", this);

this.getLogger().info("MVC started at " + path);

}

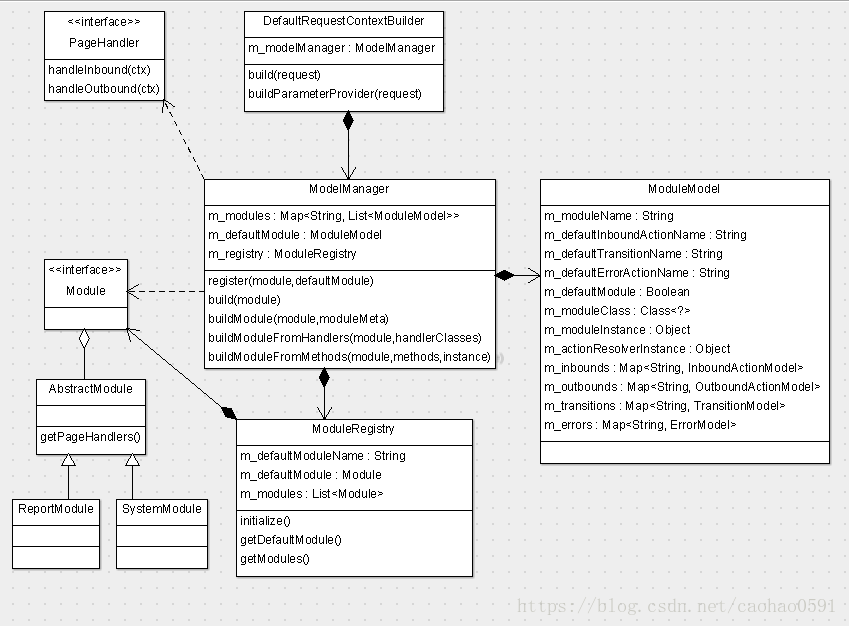
}

}



    从上面类图， 我们看到MVC 拥有DefaultRequestLifecycle的一个实例，用于完成每次请求的生命周期内的所有工作，例如URL的解析，路由的实现，异常处理等等。DefaultRequestLifecycle 拥有3个重要的对象，ServletContext对象用于维护servlet容器的上下文，RequestContextBuilder对象用于创建请求上下文，它是管理请求解析、路由的核心，ActionHandlerManager对象用于管理负责页面请求的动作执行的处理器，他会被plexus容器实例化。

## 页面路由初始化



     我们先来具体看看 RequestContextBuilder 对象的功能以及初始化的逻辑，如上类图。RequestContextBuilder 利用 ModelManager 来管理模块(Module)与动作模型(Model)， 模块与动作模型是请求上下文中2个非常重要的概念，不要搞混淆，模块是大的页面分类，目前整个CAT页面平台只有2个模块，分别是报表模块(ReportModule)和系统模块(SystemModule)，动作模型则是两大模块下面具体的页面路由信息，他维护具体页面处理器(Handler)的指针、以及处理的函数，以便在请求到来的时候，根据URL找到相应页面处理器，将控制权交给处理器(Handler)，由处理器进行数据处理，比如获取报表数据等，然后展示页面，Module 的对象指针和动作模型的信息都会存在于ModuleModel对象中，ModuleModel相当于MVC框架的路由转发控制核心，而Handler相当于MVC中的Controller层。接下来我们详细剖析。

      ModelManager 的 ModuleRegistry 对象负责对模块的实例化。ModuleRegistry类实现了Initializable方法，所以在plexus实例化ModuleRegistry之后，遍会调用他的initialize()方法实例化ReportModule和SystemModule对象了，如下：

public class ModuleRegistry extends ContainerHolder implements Initializable {

private String m\_defaultModuleName;

private Module m\_defaultModule;

private List<Module> m\_modules;

@Override

public void initialize() throws InitializationException {

if (m\_defaultModuleName != null) {

m\_defaultModule = lookup(Module.class, m\_defaultModuleName);

}

m\_modules = lookupList(Module.class);

}

}

我们先来看两个页面URL：

存储报表页面： http://localhost:8080/cat/r/storage

配置管理页面：http://localhost:8080/cat/s/config

上面URL中的 'r' 代表的就是ReportModule，所有的报表相关的页面都在ReportModule中定义， 's'  则代表SystemModule，系统相关的页面都在SystemModule中定义，上面参数的storage和config是路由动作(Action)，后续将会详细讲解，这里的 'r'、's' 实际上是模块名称，该参数在哪里定义的呢？ 在模块(Module)的java定义文件中，我们看到模块元数据(ModuleMeta) 的定义如下：

@ModuleMeta(name = **"s"**, defaultInboundAction = **"config"**, defaultTransition = **"default"**, defaultErrorAction = **"default"**)

@ModuleMeta(name = **"r"**, defaultInboundAction = **"home"**, defaultTransition = **"default"**, defaultErrorAction = **"default"**)

     在元数据中，就有名称信息，除此之外，元数据还定义了模块的默认路由动作，也就是上面url中的 "storage"、"config"，报表默认页面是home，即我们一进去看到的首页， 而系统默认页面则是 config。在两个模块的java定义文件中，除了Module元数据的定义之外，还有对该模块下所有页面处理器(Handler)的定义：

系统页面处理器：

@ModulePagesMeta({

com.dianping.cat.system.page.login.Handler.class,

com.dianping.cat.system.page.config.Handler.class,

com.dianping.cat.system.page.plugin.Handler.class,

com.dianping.cat.system.page.router.Handler.class

})

报表页面处理器：

@ModulePagesMeta({

com.dianping.cat.report.page.home.Handler.class,

com.dianping.cat.report.page.problem.Handler.class,

com.dianping.cat.report.page.transaction.Handler.class,

com.dianping.cat.report.page.event.Handler.class,

com.dianping.cat.report.page.heartbeat.Handler.class,

com.dianping.cat.report.page.logview.Handler.class,

com.dianping.cat.report.page.model.Handler.class,

com.dianping.cat.report.page.dashboard.Handler.class,

com.dianping.cat.report.page.matrix.Handler.class,

com.dianping.cat.report.page.cross.Handler.class,

com.dianping.cat.report.page.cache.Handler.class,

com.dianping.cat.report.page.state.Handler.class,

com.dianping.cat.report.page.metric.Handler.class,

com.dianping.cat.report.page.dependency.Handler.class,

com.dianping.cat.report.page.statistics.Handler.class,

com.dianping.cat.report.page.alteration.Handler.class,

com.dianping.cat.report.page.monitor.Handler.class,

com.dianping.cat.report.page.network.Handler.class,

com.dianping.cat.report.page.web.Handler.class,

com.dianping.cat.report.page.system.Handler.class,

com.dianping.cat.report.page.cdn.Handler.class,

com.dianping.cat.report.page.app.Handler.class,

com.dianping.cat.report.page.alert.Handler.class,

com.dianping.cat.report.page.overload.Handler.class,

com.dianping.cat.report.page.database.Handler.class,

com.dianping.cat.report.page.storage.Handler.class,

com.dianping.cat.report.page.activity.Handler.class,

com.dianping.cat.report.page.top.Handler.class

})

    我们回过头来再看ModelManager的初始化过程，在两个模块被实例化之后 ModelManager 将调用 register() 方法注册模块(Module)， 并放入 Map<String, List<ModuleModel>> 对象中， Map 的key是String类型，表示的就是Module名称， Map的 value 则是 ModuleModel 。

public class ModelManager extends ContainerHolder implements Initializable {

@Inject

private ModuleRegistry m\_registry;

private Map<String, List<ModuleModel>> m\_modules = new HashMap();

private ModuleModel m\_defaultModule;

public void initialize() throws InitializationException {

Module defaultModule = this.m\_registry.getDefaultModule();

List<Module> modules = this.m\_registry.getModules();

Iterator i$ = modules.iterator();

while(i$.hasNext()) {

Module module = (Module)i$.next();

this.register(module, defaultModule == module);

}

}

void register(Module module, boolean defaultModule) {

ModuleModel model = this.build(module);

String name = model.getModuleName();

List<ModuleModel> list = (List)this.m\_modules.get(name);

if(list == null) {

list = new ArrayList();

this.m\_modules.put(name, list);

}

((List)list).add(model);

if(defaultModule) {

...

this.m\_defaultModule = model;

}

}

}

    ModuleModel在什么时候会被创建、初始化？ 在模块注册函数regiter()中，注册的第一步，就是调用 build() 函数创建 ModuleModel，现在我们详细剖析整个build的流程，build函数首先提取Module元数据(ModuleMeta)，然后调用 buildModule() 创建ModuleModel对象，并初始化ModuleModel信息， 例如设置模块名、模块类信息、默认动作、以及模块对象(Module)指针，然后调用 buildModuleFromMethods 创建路由动作模型(Model)。

public class ModelManager extends ContainerHolder implements Initializable {

ModuleModel build(Module module) {

Class<?> moduleClass = module.getClass();

ModuleMeta moduleMeta = (ModuleMeta)moduleClass.getAnnotation(ModuleMeta.class);

if(moduleMeta == null) {

throw new RuntimeException(moduleClass + " must be annotated by " + ModuleMeta.class);

} else {

ModuleModel model = this.buildModule(module, moduleMeta);

this.validateModule(model);

return model;

}

}

private ModuleModel buildModule(Module module, ModuleMeta moduleMeta) {

ModuleModel model = new ModuleModel();

Class<?> moduleClass = module.getClass();

model.setModuleName(moduleMeta.name());

model.setModuleClass(moduleClass);

model.setDefaultInboundActionName(moduleMeta.defaultInboundAction());

model.setDefaultTransitionName(moduleMeta.defaultTransition());

model.setDefaultErrorActionName(moduleMeta.defaultErrorAction());

model.setActionResolverInstance(this.lookupOrNewInstance(moduleMeta.actionResolver()));

model.setModuleInstance(module);

this.buildModuleFromMethods(model, moduleClass.getMethods(), model.getModuleInstance());

Class<? extends PageHandler<?>>[] pageHandlers = module.getPageHandlers();

if(pageHandlers != null) {

this.buildModuleFromHandlers(model, pageHandlers);

}

return model;

}

}

   路由动作模型(Model)分为4类，分别是InboundActionModel、OutboundActionModel、TransitionModel、ErrorModel，InboundActionModel主要负责变更，OutboundActionModel负责页面展示，绝大多数页面都是展示目的，TransitionModel负责转场，ErrorModel则负责异常处理。



    每个动作模型(Model) 中都包含路由页面处理器(Handler)的指针，指针在Model中由 m\_moduleInstance 变量维护、以及路由到处理器中的具体哪个方法(Method)，在Model中由m\_actionMethod 变量维护。路由协议由谁来定？在各个处理器类中，我们看到了四类注解，InboundActionMeta、OutboundActionMeta、TransitionMeta、ErrorActionMeta，他们将决定哪个处理器的哪个函数会被路由到。

    而buildModule函数将会从两个方面创建路由动作模型，一个在 buildModule(...) 函数中 this.buildModuleFromMethods(model, moduleClass.getMethods(), model.getModuleInstance());  他会遍历所有Module 类的成员函数，找到被注解的函数，仅有SystemModule和ReportModule 模块的父类 AbstractModule 拥有TransitionMeta、ErrorActionMeta 两类注解，事实上这两个动作没有做任何事情，如下：

public abstract class AbstractModule implements Module {

@TransitionMeta(name = "default")

public void handleTransition(ActionContext<?> ctx) {

// simple cases, nothing here

}

@ErrorActionMeta(name = "default")

public void onError(ActionContext<?> ctx) {

// ignore error, leave MVC to handle it

}

}

    另一方面， buildModule(...)，会通过 module.getPageHandlers() 获取模块(Module)下的所有页面处理器(Handler)，具体有哪些Handler在之前讲解 @ModulePagesMeta 的时候有列举，大家可以翻前面看看，然后去遍历这些Handler的成员函数，查看是否有上面讲过的那些注解，如果有注解，就将被注解的成员函数信息以及页面处理器的指针写入相应的动作模型(Model)中，不同动作模型会被分别加入ModuleModel的如下4个Map成员变量中，Map的 key 是 String类型，指代动作名称。

public class ModuleModel extends BaseEntity<ModuleModel> {

private Map<String, InboundActionModel> m\_inbounds = new LinkedHashMap();

private Map<String, OutboundActionModel> m\_outbounds = new LinkedHashMap();

private Map<String, TransitionModel> m\_transitions = new LinkedHashMap();

private Map<String, ErrorModel> m\_errors = new LinkedHashMap();

}

     例如下面的Storage报表页面处理器(Storage\Handler) 的handleInbound和handleOutbound 函数，分别被InboundActionMeta 和 OutboundActionMeta 注解了， 那么对于url ： http://localhost:8080/cat/r/storage ，我们就可以根据 "r" 找到对应的ModuleModel对象，然后根据动作名称 "storage" ，找到对应的Model对象了， 然后程序控制权将会先后交给InboundActionModel、TransitionModel、OutboundActionModel中维护的处理器的执行函数。捕获的异常将由 ErrorModel 维护的执行函数。下面Storage\Handler 的成员函数 handleInbound(...) 和 handleOutbound(...) ，将会先后被执行。

package com.dianping.cat.report.page.storage;

public class Handler implements PageHandler<Context> {

@Override

@PayloadMeta(Payload.class)

@InboundActionMeta(name = "storage")

public void handleInbound(Context ctx) throws ServletException, IOException {

}

@Override

@OutboundActionMeta(name = "storage")

public void handleOutbound(Context ctx) throws ServletException, IOException {

...

}

}

    除此之外，我们从下面代码看到一个PreInboundActionMeta注解，有些动作可能会包含一个或者多个前置动作。而这些注解就用于标识某个动作的前置动作。

private void buildModuleFromHandlers(ModuleModel module, Class<? extends PageHandler<?>>[] handlerClasses) {

Class[] arr$ = handlerClasses;

int len$ = handlerClasses.length;

for(int i$ = 0; i$ < len$; ++i$) {

Class<? extends PageHandler<?>> handlerClass = arr$[i$];

PageHandler<?> handler = (PageHandler)this.lookup(handlerClass);

this.buildModuleFromMethods(module, handlerClass.getMethods(), handler);

}

}

private void buildModuleFromMethods(ModuleModel module, Method[] methods, Object instance) {

Method[] arr$ = methods;

int len$ = methods.length;

for(int i$ = 0; i$ < len$; ++i$) {

Method method = arr$[i$];

int modifier = method.getModifiers();

if(!Modifier.isStatic(modifier) && !Modifier.isAbstract(modifier) && !method.isBridge() && !method.isSynthetic()) {

InboundActionMeta inMeta = (InboundActionMeta)method.getAnnotation(InboundActionMeta.class);

PreInboundActionMeta preInMeta = (PreInboundActionMeta)method.getAnnotation(PreInboundActionMeta.class);

OutboundActionMeta outMeta = (OutboundActionMeta)method.getAnnotation(OutboundActionMeta.class);

TransitionMeta transitionMeta = (TransitionMeta)method.getAnnotation(TransitionMeta.class);

ErrorActionMeta errorMeta = (ErrorActionMeta)method.getAnnotation(ErrorActionMeta.class);

int num = (inMeta == null?0:1) + (outMeta == null?0:1) + (transitionMeta == null?0:1) + (errorMeta == null?0:1);

if(num != 0) {

if(num > 1) {

throw new RuntimeException(method + " can only be annotated by one of " + InboundActionMeta.class + ", " + OutboundActionMeta.class + ", " + TransitionMeta.class + " or " + ErrorActionMeta.class);

}

if(inMeta != null) {

InboundActionModel inbound = this.buildInbound(module, method, inMeta, preInMeta);

inbound.setModuleInstance(instance);

module.addInbound(inbound);

} else if(outMeta != null) {

OutboundActionModel outbound = this.buildOutbound(module, method, outMeta);

outbound.setModuleInstance(instance);

module.addOutbound(outbound);

} else if(transitionMeta != null) {

TransitionModel transition = this.buildTransition(method, transitionMeta);

transition.setModuleInstance(instance);

if(!module.getTransitions().containsKey(transition.getTransitionName())) {

module.addTransition(transition);

}

} else {

if(errorMeta == null) {

throw new RuntimeException("Internal error!");

}

ErrorModel error = this.buildError(method, errorMeta);

error.setModuleInstance(instance);

if(!module.getErrors().containsKey(error.getActionName())) {

module.addError(error);

}

}

}

}

}

}

## 请求处理流程

    当有http请求过来，请进入 DefaultRequestLifecycle 的handle(HttpServletRequest request, HttpServletResponse response) 进行处理，该函数首先调用 RequestContextBuilder 的 build 函数创建请求上下文， 然后将请求上下文交给动作处理器管理对象(ActionHandlerManager) 管理的 InboundActionHandler、OutboundActionHandler、TransitionHandler先后处理，如果捕获到异常交给 ErrorHandler 处理。

public class DefaultRequestLifecycle implements RequestLifecycle, LogEnabled {

private RequestContextBuilder m\_builder;

private ActionHandlerManager m\_actionHandlerManager;

private ServletContext m\_servletContext;

public void handle(HttpServletRequest request, HttpServletResponse response) throws IOException {

RequestContext context = this.m\_builder.build(request);

try {

if(context == null) {

this.showPageNotFound(request, response);

} else {

this.handleRequest(request, response, context);

}

} finally {

if(context != null) {

this.m\_builder.reset(context);

}

}

}

}

# 第十章 CAT客户端与其它JAVA框架的集成

## 与Spring MVC集成

## 与Spring Boot 集成

## 与Spring Cloud 集成

## 与dubbo集成

## 与MyBatis集成

## 与Log4j集成

# 第十一章 对其它语言的支持

## PHP语言

## C++语言

## LUA语言

## Go语言

## Node.js语言

## Android埋点

## Object C -- IOS 埋点

# 第十二章 大数据收集与实时计算

## hadoop模块

## spark实时计算模块

# 第十三章 报警与通知机制

## 短信通知

## 邮件通知