# 目录

# Pinpoint学习笔记

Pinpoint是一个开源的 APM (Application Performance Management/应用性能管理)工具，用于基于java的大规模分布式系统，基于Google Dapper论文。

由于目前几乎没有任何中文资料，因此翻译了部分英文文档。

# Pinpoint

> 翻译自 Pinpoint 的 github 首页内容

## 介绍

Pinpoint是一个开源的 APM (Application Performance Management/应用性能管理)工具，用于基于java的大规模分布式系统。

仿照 Google Dapper , Pinpoint 通过跟踪分布式应用之间的调用来提供解决方案，以帮助分析系统的总体结构和内部模块之间如何相互联系.

> 注：对于各个模块之间的通讯英文原文中用的是transaction一词，但是我觉得如果翻译为"事务"容易引起误解，所以替换为"交互"或者"调用"这种比较直白的字眼。

在使用上力图简单高效：

- 安装agent，不需要修改哪怕一行代码
- 最小化性能损失

## 概述

如今的服务通常由很多不同模块组成，他们之间相互调用并通过API调用外部服务。每个交互是如何被执行的通常是一个黑盒。Pinpoint跟踪这些模块之间的调用流并提供清晰的视图来定位问题区域和潜在瓶颈。

- 服务器地图(ServerMap)

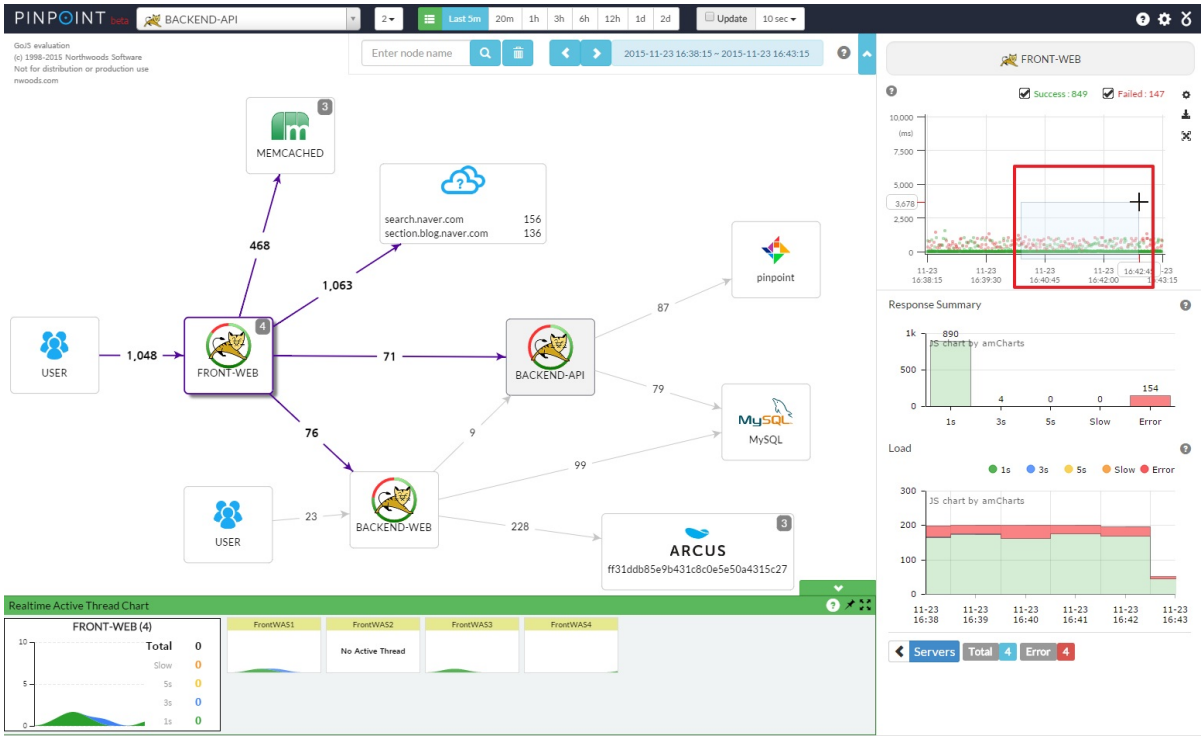
  通过可视化分布式系统的模块和他们之间的相互联系来理解系统拓扑。点击某个节点会展示这个模块的详情，比如它当前的状态和请求数量。

- 实时活动线程图表(Realtime Active Thread Chart)

  实时监控应用内部的活动线程。

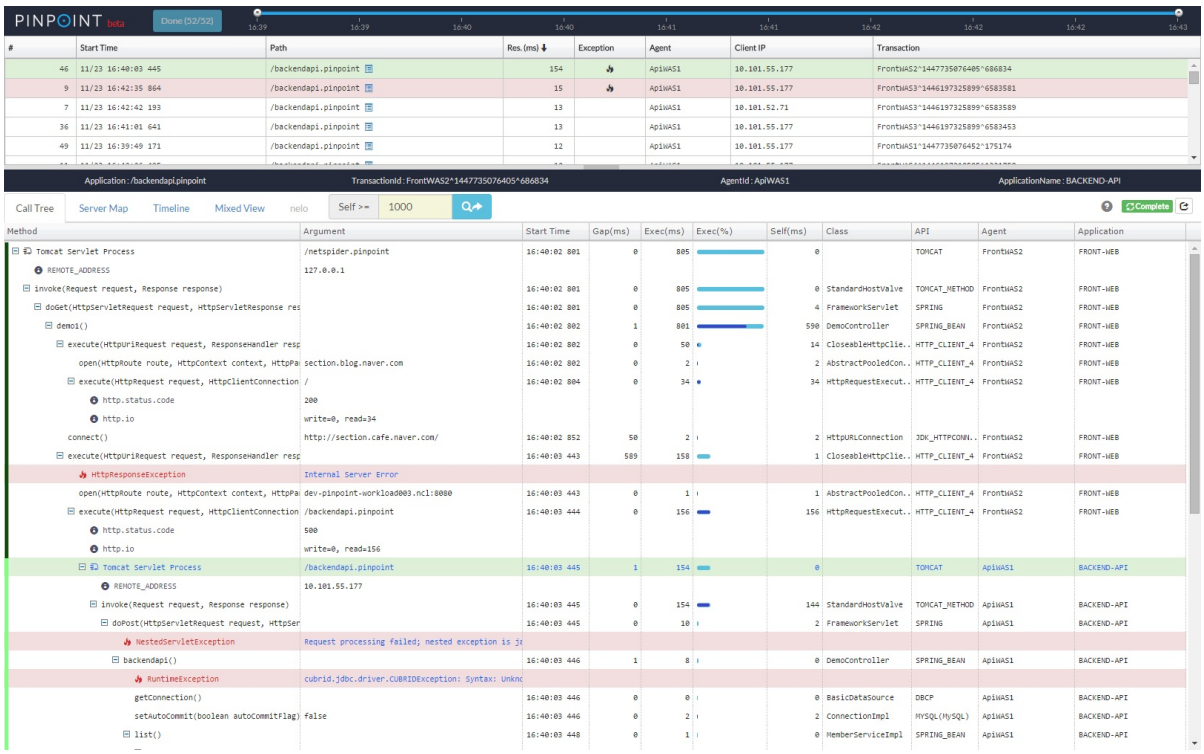
- 请求/应答分布图表(Request/Response Scatter Chart)

  长期可视化请求数量和应答模式来定位潜在问题。通过在图表上拉拽可以选择请求查看更多的详细信息。



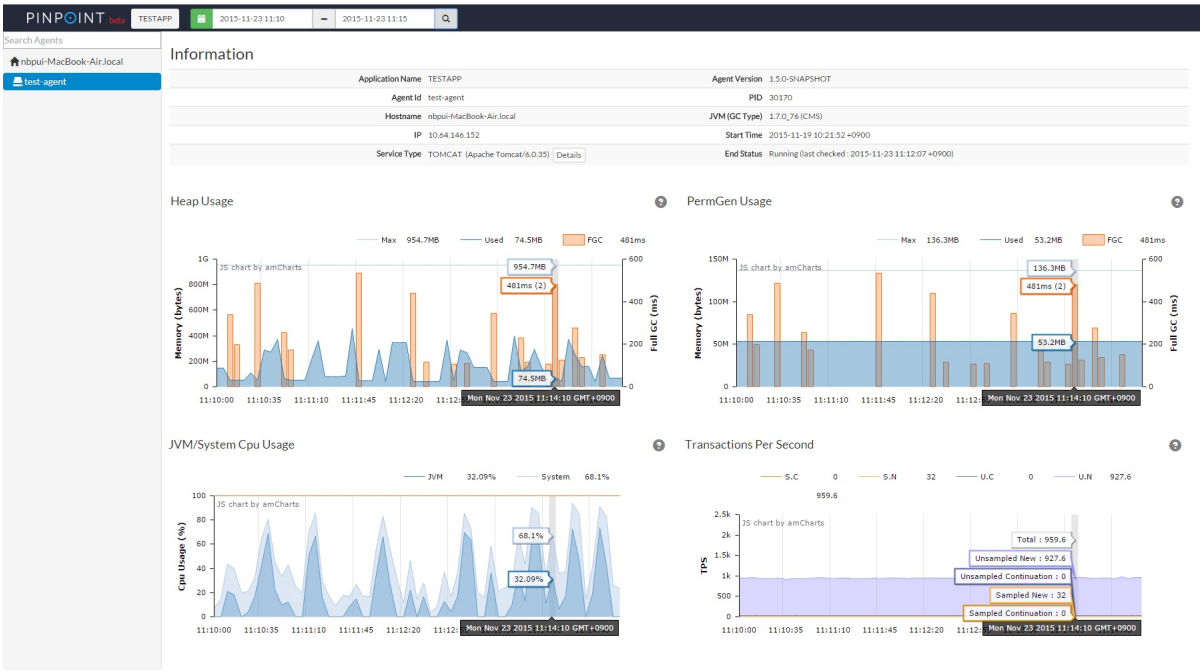

- 调用栈(CallStack)

  在分布式环境中为每个调用生成代码级别的可视图，在单个视图中定位瓶颈和失败点。

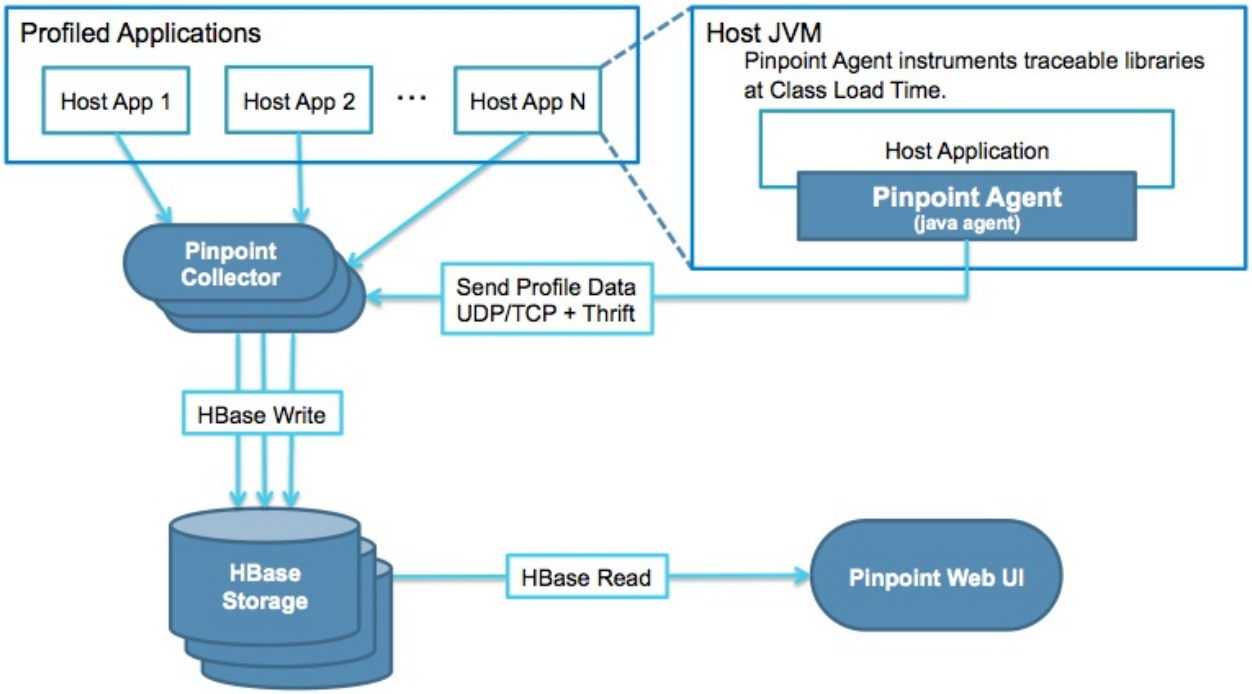



- 巡查(Inspector)

  查看应用上的其他详细信息，比如CPU使用率，内存/垃圾回收，TPS，和JVM参数。




# 架构




# 支持模块

- JDK 6+
- Tomcat 6/7/8, Jetty 8/9
- Spring, Spring Boot
- Apache HTTP Client 3.x/4.x, JDK HttpConnector, GoogleHttpClient, OkHttpClient,

NingAsyncHttpClient
- Thrift Client, Thrift Service
- MySQL, Oracle, MSSQL, CUBRID, DBCP, POSTGRESQL
- Arcus, Memcached, Redis
- iBATIS, MyBatis
- gson, Jackson, Json Lib
- log4j, Logback

# Pinpoint资料

Pinpoint的资料，坦白说非常少，尤其中文资料几乎没有。

因此在整理资料时，自己动手翻译了部分内容。

## 官方资料

### 代码托管

Pinpoint的源代码托管在 github.

## WIKI

WIKI地址

Wiki上的内容：

- 2016 Roadmap
- 视频Introduction to Pinpoint @youtu.be / Pinpoint介绍 @爱奇异

### 用户组

提问和讨论在google group

### 介绍文档

还有几份介绍和使用资料，可以说是目前仅有的一点点文档了：

- Technical Overview of Pinpoint: 中文翻译版本 点这里
- Using Pinpoint with Docker
- Notes on Jetty Plugin for Pinpoint

### 安装和开发文档

- 快速开始/quickstart: 中文翻译版本 点这里
- 安装指南/installation guide: 中文翻译版本 点这里
- 插件示例/plugin samples: 中文翻译版本 点这里
- 告警/Alarm： 中文翻译版本 点这里

# 中文资料

- 开源中国上的介绍页面

# APM介绍

# 什么是APM？

APM是一个缩写，但是具体是哪个好像还有不同说法：

1. Application Performance Management / 应用性能管理
2. Application Performance Monitoring / 应用性能监控
3. Application Performance Management & Monitoring / 应用性能管理与监控

我们不探究，反正就是这么回事了。

TBD: 收集资料完善这个页面。

# 快速开始

> 注：内容翻译自 官方quick start文档，增加了少量补充和说明。

Pinpoint有三个主要组件(collector, web, agent)，并使用HBase作为存储。Collector和Web被打包为单个war文件，而agent被打包以便可以作为java agent附加到应用。

Pinpoint quickstart 为agent提供一个示例TestApp， 并使用tomcat maven插件来启动所有三个组件。

# 要求

为了构建pinpoint， 下列要求必须满足：

- 安装有JDK 6
- 安装有JDK 8
- 安装有Maven 3.2.x+
- 环境变量JAVA_6_HOME 设置为 JDK 6 home 目录
- 环境变量JAVA_7_HOME 设置为 JDK 7+ home 目录
- 环境变量JAVA_8_HOME 设置为 JDK 8+ home 目录

QuickStart 支持 Linux, OSX, 和 Windows.

> 注：没有说要不要安装jdk7，顺便一起安装吧。下面是/etc/profile的设置：

```
# use by pinpoint compile
export JAVA_6_HOME=/usr/lib/jvm/java-6-oracle/
export JAVA_7_HOME=/usr/lib/jvm/java-7-oracle/
export JAVA_8_HOME=/usr/lib/jvm/java-8-oracle/
```

# 开始

使用 git clone https://github.com/naver/pinpoint.git 下载pinpoint或者将项目作为zip文件打包下载然后解压。

使用maven安装pinpoint并运行 mvn install -Dmaven.test.skip=true

> 注:需要执行的命令如下：
>
> ```
> git clone https://github.com/naver/pinpoint.git
> cd pinpoint
> mvn install -Dmaven.test.skip=true
> ```

# 安装并启动HBase

下面脚本从 Apache 下载站点 单独下载HBase.

```
对于Windows, 需要从Apache下载站点手工下载HBase.
下载 HBase-1.0.1-bin.tar.gz 并解压缩.
重命名目录为 hbase 以便使得最终hbase目录看上去是 quickstart\hbase\hbase.
另外注意通过相应的.cmd文件来运行脚本。
```

下载并启动 - 运行 quickstart/bin/start-hbase.sh

初始化表 - 运行 quickstart/bin/init-hbase.sh

> 补充：这里面有两个地方要特别注意
>
> 1. 如果手工下载HBase，按照上面要求解压并重命名为路径quickstart\hbase\hbase。启动时会出错，因为start-hbase.sh文件中hbase配置的路径是"HBASE_VERSION=hbase-1.0.1"，需要手工修改为"HBASE_VERSION=hbase"
> 2. init-hbase.sh不仅仅第一次运行时需要执行，以后再启动quickstart时，也需要执行，否则collector和web启动时会始终无法成功最后180秒超时报错退出。再多执行一次init-hbase.sh就可以正常启动。

# 启动pinpoint守护进程

Collector - 运行 quickstart/bin/start-collector.sh

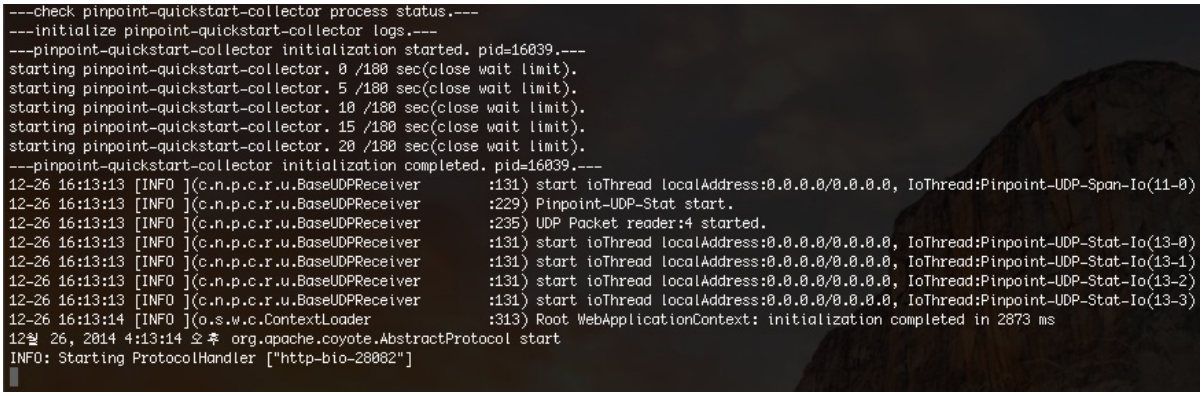
Web UI - 运行 quickstart/bin/start-web.sh

TestApp - 运行 quickstart/bin/start-testapp.sh

> 注：这三个脚本启动后，用ctrl + c可以退出控制台，此时后台进程还在，但是会看不到日志。因此建议这三个脚本分别在三个不同的终端中执行，这样就可以方便查看每个组件的日志信息。

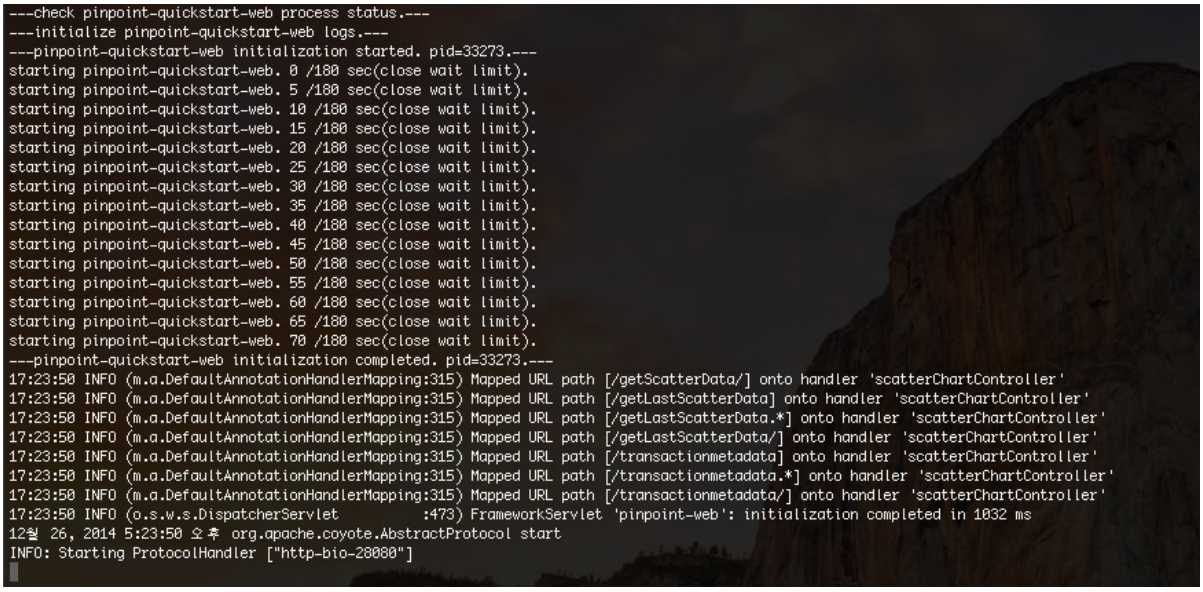
一旦启动脚本完成，tomcat 日志的最后10行显示在控制台：

- Collector

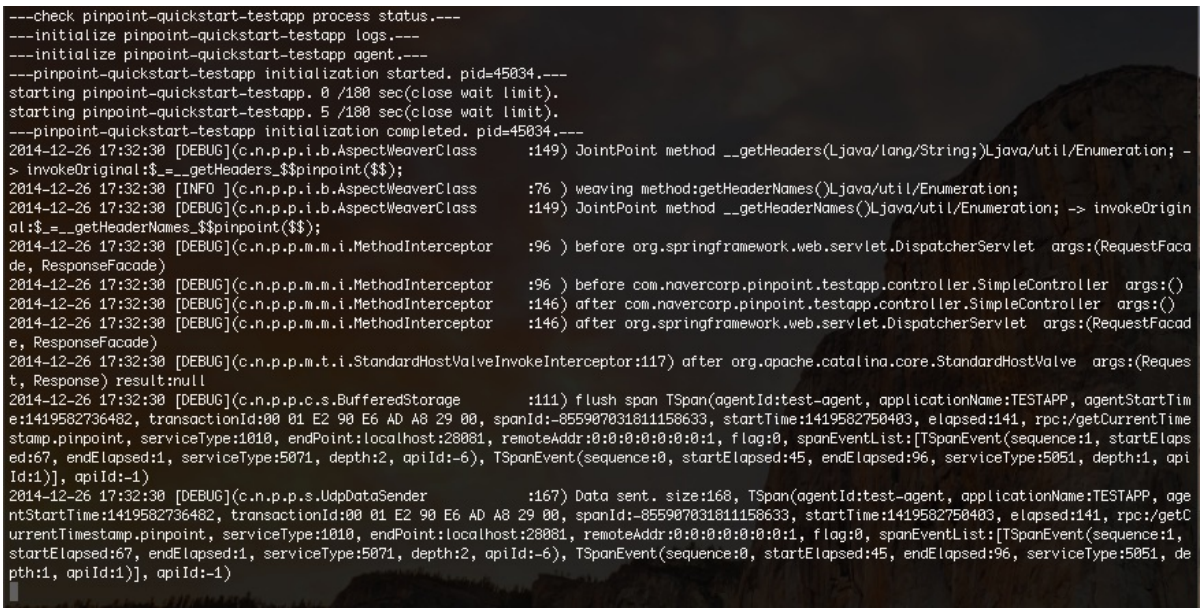



> 注：如果启动不起来，总是打印"starting pinpoint-quickstart-web **/180 (close wait limit)"，最后180秒超时失败。请尝试再次执行一遍"init-hbase.sh".

- Web UI




- TestApp



# 检查状态

一旦HBase和三个守护进程在运行，可以访问下面地址来测试自己的pinpoint实例。

Web UI - http://localhost:28080 TestApp - http://localhost:28081

可以通过使用TestApp UI来产生追踪数据给pinpoint， 并使用pinpoint Web UI来检查。
TestApp作为test-agent注册在TESTAPP下。

# 停止

HBase - 运行 quickstart/bin/stop-hbase.sh

Collector - 运行 quickstart/bin/stop-collector.sh

Web UI - 运行 quickstart/bin/stop-web.sh

TestApp - 运行 quickstart/bin/stop-testapp.sh

# 额外

pinpoint Web使用mysql来持久化用户/用户组，和警告配置。

而Quickstart使用MockDAO来减少内存使用。

此外如果想使用mysql来执行Quickstart， 请参考Pinpoint Web's applicationContext-dao-config.xml , jdbc.properties.

此外，如果想开启告警，需要实现额外逻辑。请参考这个 链接。

> 注： 上面这个Alarm文档的中文翻译版本在 这里

# 安装指南

> 注：内容翻译自 官方文档 Installation Guide

# 安装

为了搭建自有的Pinpoint实例，需要运行这些组件：

- HBase (用于存储)
- Pinpoint Collector (部署在web容器中)
- Pinpoint Web (部署在web容器中)
- Pinpoint Agent (附加到 java 应用来做采样/profile)

如果要尝试简单的快速开始项目，请参考 quick-start guide

# 快速概述

1. HBase
    - 搭建 HBase 集群 - Apache HBase
    - 创建 HBase Schemas - 在hbase shell上执行 /scripts/hbase-create.hbase
2. 构建Pinpoint (仅当从源代码开始构建时需要)
    - Clone Pinpoint - git clone $PINPOINT_GIT_REPOSITORY
    - 设置 JAVA_6_HOME 环境变量到 JDK 6 home 目录.
    - 设置 JAVA_7_HOME 环境变量到 JDK 7+ home 目录.
    - 在pinpoint 根目录运行 mvn install -Dmaven.test.skip=true
3. Pinpoint Collector
    - 部署 pinpoint-collector-$VERSION.war 到web容器
    - 配置 pinpoint-collector.properties, hbase.properties.
    - 启动容器
4. Pinpoint Web
    - 部署 pinpoint-web-$VERSION.war 到web容器
    - 配置 pinpoint-web.properties, hbase.properties.
    - 启动容器
5. Pinpoint Agent

- 解压/移动 pinpoint-agent/ 到一个方便的位置 ($AGENT_PATH).
- 设置 -javaagent:$AGENT_PATH/pinpoint-bootstrap-$VERSION.jar JVM 参数以便将agent附加到java应用
- 设置 -Dpinpoint.agentId 和 -Dpinpoint.applicationName 命令行参数
- 用上面的设置启动 java 应用

# HBase

Pinpoint 为collector和web使用 HBase 作为它的存储后端 .

为了搭建自己的集群， 参考 Hbase网站。下面给出的是HBase 兼容性表单：

| Pinpoint Version | HBase 0.94.x | HBase 0.98.x | HBase 1.0.x | HBase 1.1.x |
|---|---|---|---|---|
| 1.0.x | yes | no | no | no |
| 1.1.x | no | not tested | yes | not tested |
| 1.5.x | no | not tested | yes | not tested |

一旦搭建并运行好HBase，请确保Collector和Web被正确的配置并能够连接到HBase。

# 创建 Schema

有两个脚本可以为pinpoint创建表：hbase-create.hbase 和 hbase-create-snappy.hbase.使用 hbase-create-snappy.hbase 来实现 snappy 压缩 (需要 snappy), 其他情况使用 hbase-create.hbase.

为了运行这些脚本， 在HBase shell 中如下执行：

```
$HBASE_HOME/bin/hbase shell hbase-create.hbase
```

脚本的完整列表见 这里.

# 构建Pinpoint

有两个选择：

1. 从 最新的发布 中下载构建结果并跳过构建过程， 推荐!
2. 从Git clone中手工构建

为了手工构建，必须满足下列要求：

- 安装有JDK 6
- 安装有JDK 7+
- 安装有Maven 3.2.x+
- JAVA_6_HOME 环境变量设置为 JDK 6 home 目录
- JAVA_7_HOME 环境变量设置为 JDK 7+ home 目录

需要JDK 7+ 和 JAVA_7_HOME 环境变量来构建 profiler-optional. 更多关于 optional package 的信息，请看 这里.

另外, 为了运行Pinpoint的每个组件所需要的Java 版本列举在下面:

| Pinpoint Version | Agent | Collector | Web |
|---|---|---|---|
| 1.0.x | 6+ | 6+ | 6+ |
| 1.1.x | 6+ | 7+ | 7+ |
| 1.5.x | 6+ | 7+ | 7+ |

如果上面的要求满足了，就可以简单运行下面的命令：

```
mvn install -Dmaven.test.skip=true
```

安装指南后面将使用 $PINPOINT_PATH 来引用 pinpoint home目录的全路径。

不管那种方法，应该以后面章节中提到的文件和目录告终。

# Pinpoint Collector

需要有下面的war文件来部署到web容器中：

```
pinpoint-collector-$VERSION.war
```

如果手工构建，这个文件的路径会是 $PINPOINT_PATH/collector/target/pinpoint-collector-$VERSION.war。

## 安装

由于Pinpoint Collector 被打包为可部署的war文件，可以像部署其他web应用一样部署到web容器。

## 配置

Pinpoint Collector 有 2 个配置文件: pinpoint-collector.properties 和 hbase.properties.

- pinpoint-collector.properties： 包含colletor的配置。和agent的配置项一起检查下面的值：
    - collector.tcpListenPort (agent中是 profiler.collector.tcp.port - 默认: 9994)
    - collector.udpStatListenPort (agent中是 profiler.collector.stat.port - 默认: 9995)
    - collector.udpSpanListenPort (agent中是 profiler.collector.span.port - 默认: 9996)
- hbase.properties - 包含连接到HBase的配置
    - hbase.client.host (默认: localhost)
    - hbase.client.port (默认: 2181)

这些配置文件在war文件下的 WEB-INF/classes/ 目录.

可以在这里看一下默认配置文件: pinpoint-collector.properties, hbase.properties

# Pinpoint Web

需要有下面的war文件来部署到web容器中：

```
pinpoint-web-$VERSION.war
```

如果手工构建，这个文件的路径会是 $PINPOINT_PATH/collector/target/pinpoint-web-$VERSION.war。

## 安装

由于Pinpoint Web 被打包为可部署的war文件，可以像部署其他web应用一样部署到web容器。

## 配置

和collector类似，Pinpoint web有和安装相关的配置文件：pinpoint-web.properties 和 hbase.properties.

确保检查下面的配置项：

- hbase.properties - 包含连接到HBase的配置
    - hbase.client.host (默认: localhost)

- hbase.client.port (默认: 2181)

这些配置文件在war文件下的 WEB-INF/classes/ 目录.

可以在这里看一下默认配置文件: pinpoint-web.properties, hbase.properties

# Pinpoint Agent

下载后解压Pinpint Agent文件， pinpoint-agent 目录层次如下：

```
pinpoint-agent
|-- boot
|   |-- pinpoint-bootstrap-core-$VERSION.jar
|-- lib
|   |-- pinpoint-profiler-$VERSION.jar
|   |-- pinpoint-profiler-optional-$VERSION.jar
|   |-- pinpoint-rpc-$VERSION.jar
|   |-- pinpoint-thrift-$VERSION.jar
|   |-- ...
|-- pinpoint-bootstrap-$VERSION.jar
|-- pinpoint.config
```

如果手工构建这个目录会在这里： $PINPOINT_PATH/agent/target/pinpoint-agent.

可以移动/解压pinpoint-agent目录的内容到任何未知。安装指南后面用$AGENT_PATH来引用这个目录的全路径.

## 安装

Pinpoint Agent 作为一个java agent附加到需要采样的应用(例如 Tomcat).

为了让agent生效，在运行应用时需要设置 -javaagent JVM 参数为 $AGENT_PATH/pinpoint-bootstrap-$VERSION.jar:

```
-javaagent:$AGENT_PATH/pinpoint-bootstrap-$VERSION.jar
```

另外，Pinpoint Agent 需要两个命令行参数来在分布式系统中标记自身:

- Dpinpoint.agentId - 唯一标记agent运行所在的应用
- Dpinpoint.applicationName - 将许多的同样的应用实例分组为单一服务

注意 pinpoint.agentId 必须全局唯一来标识应用实例， 而所有共用相同 pinpoint.applicationName 的应用被当成单个服务的多个实例。

### Tomcat 示例

在tomcat 启动脚本(catalina.sh)中添加 -javaagent, -Dpinpoint.agentId, -Dpinpoint.applicationName.

```
CATALINA_OPTS="$CATALINA_OPTS -javaagent:$AGENT_PATH/pinpoint-bootstrap-$VERSION.jar"
CATALINA_OPTS="$CATALINA_OPTS -Dpinpoint.agentId=$AGENT_ID"
CATALINA_OPTS="$CATALINA_OPTS -Dpinpoint.applicationName=$APPLICATION_NAME"
```

启动tomcat来开始web应用的采样。

## 配置

在$AGENT_PATH/pinpoint.config 中有很多Pinpoint Agent的配置选项。

这些选项的大部分是自我描述的，而最重要的必须检查的配置选项是collector ip address 和 TCP/UDP 端口。Agent需要这些值来创建到collector的连接并正确工作。

在 pinpoint.config 中相应的设置这些值:

- profiler.collector.ip (默认: 127.0.0.1)
- profiler.collector.tcp.port (collector中是 collector.tcpListenPort - 默认: 9994)
- profiler.collector.stat.port (collector中是 collector.udpStatListenPort - 默认: 9995)
- profiler.collector.span.port (collector中是 collector.udpSpanListenPort - 默认: 9996)

可以在 这里 看一下默认的带有所有可用配置选项的 pinpoint.config 文件.

## 杂项

# 将web请求路由到agent

从 1.5.0 版本开始, Pinpoint 可以通过collector从web直接发送请求到agent(反之亦然). 为此需要使用Zookeeper 来协调agent和collector之间和collectors 和 web 之间的通讯通道. 在此之上，实时通讯(例如活动线程数量监控)才变的可能.

通常使用HBase后端提供的Zookeeper实例，这样就不需要额外的Zookeeper配置。相关的配置选项在这里：

- Collector - pinpoint-collector.properties
    - cluster.enable
    - cluster.zookeeper.address

    - cluster.zookeeper.sessiontimeout
    - cluster.listen.ip
    - cluster.listen.port
- Web - pinpoint-web.properties
    - cluster.enable
    - cluster.web.tcp.port
    - cluster.zookeeper.address
    - cluster.zookeeper.sessiontimeout
    - cluster.zookeeper.retry.interval
    - cluster.connect.address

# Pinpoint设计

# Google Dapper

# Pinpoint技术概述

> 注： 内容翻译自官方文档 Technical Overview Of Pinpoint, 内容有点长，但是强烈推荐阅读！基本上这是目前pinpoint唯一的一份详细介绍设计和实现的资料。

Pinpoint是一个分析大型分布式系统的平台，提供解决方案来处理海量跟踪数据。2012年七月开始开发，2015年1月9日作为开源项目启动。

本文将介绍Pinpoint： 什么促使我们开始搭建它， 用了什么技术， 还有Pinpoint agent是如何优化的。

# 开始动机 & Pinpoint特点

和如今相比， 过去的因特网的用户数量相对较小，而因特网服务的架构也没那么复杂。web服务通常使用两层(web 服务器和数据库)或三层（web服务器，应用服务器和数据库）架构。然而在如今，随着互联网的成长，需要支持大量的并发连接，并且需要将功能和服务有机结合，导致更加复杂的软件栈组合。更确切地说，比三层层次更多的n层架构变得更加普遍。SOA或者微服务架构成为现实。

系统的复杂度因此提升。系统越复杂，越难解决问题，例如系统失败或者性能问题。在三层架构中找到解决方案还不是太难，仅仅需要分析3个组件比如web服务器，应用服务器和数据库，而服务器数量也不多。但是，如果问题发生在n层架构中，就需要调查大量的组件和服务器。另一个问题是仅仅分析单个组件很难看到大局;当发生一个低可见度的问题时，系统复杂度越高，就需要更长的时间来查找原因。最糟糕的是，某些情况下我们甚至可能无法查找出来。

这样的问题也发生在NAVER的系统中。使用了大量工具如应用性能管理(APM)但是还不足以有效处理问题。因此我们最终决定为n层架构开发新的跟踪平台，为n层架构的系统提供解决方案。

Pinpoint, 2012年七月开始开发，在2015年1月作为一个开源项目启动, 是一个为大型分布式系统服务的n层架构跟踪平台。 Pinpoint的特点如下:

- 分布式事务跟踪，跟踪跨分布式应用的消息
- 自动检测应用拓扑，帮助你搞清楚应用的架构
- 水平扩展以便支持大规模服务器集群
- 提供代码级别的可见性以便轻松定位失败点和瓶颈
- 使用字节码增强技术，添加新功能而无需修改代码

本文将讲述Pinpoint的技术，例如事务跟踪和字节码增强。还会解释应用在pinpoint agent中的优化方法，agent修改字节码并记录性能数据。

# 分布式事务跟踪，基于google Dapper

pinpoint跟踪单个事务中的分布式请求，基于google Dapper。

# 在Google Dapper中分布式事务追踪是如何工作的

当一个消息从Node1发送到Node2(见图1)时，分布式追踪系统的核心是在分布式系统中识别在Node1中处理的消息和在Node2中出的消息之间的关系。

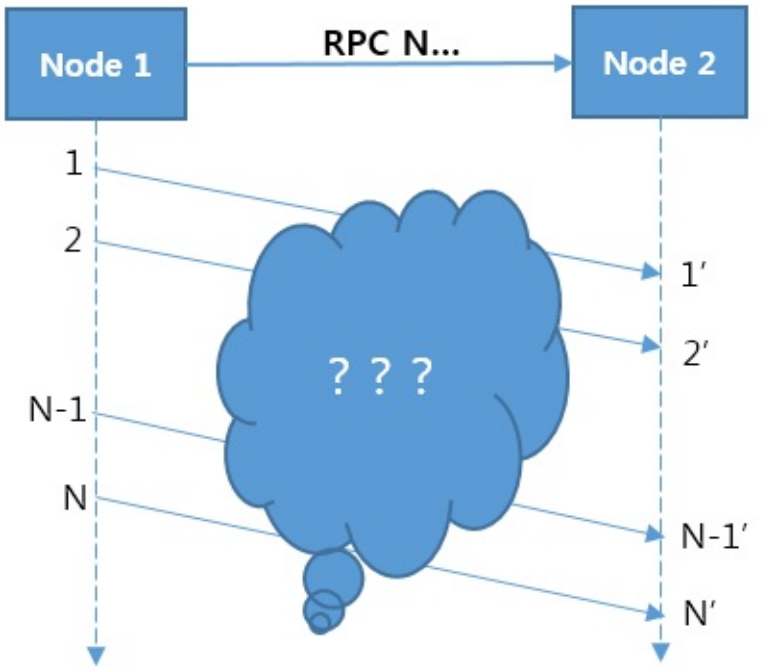



图1. 分布式系统中的消息关系

问题在于无法在消息之间识别关系。例如，我们无法识别从Node1发送的第N个消息和Node2接收到的N'消息之间的关系。换句话说，当Node1发送完第X个消息时，是无法在Node2接收到的N的消息里面识别出第X个消息的。有一种方式试图在TCP或者操作系统层面追踪消息。但是，实现很复杂而且性能低下，而且需要为每个协议单独实现。另外，很难精确追踪消息。

不过，Google dapper实现了一个简单的解决方案来解决这个问题。这个解决方案通过在发送消息时添加应用级别的标签作为消息之间的关联。例如，在HTTP请求中的HTTP header中为消息添加一个标签信息并使用这个标签跟踪消息。

> Google's Dapper
>
> 关于Google Dapper的更多信息, 请见 "Dapper, a Large-Scale Distributed Systems Tracing Infrastructure."

Pinpoint基于google dapper的跟踪技术,但是已经修改为在调用的header中添加应用级别标签数据以便在远程调用中跟踪分布式事务。标签数据由多个key组成，定义为TraceId。

# Pinpoint中的数据结构

Pinpoint中，核心数据结构由Span, Trace, 和 TraceId组成。

- Span: RPC (远程过程调用/remote procedure call)跟踪的基本单元; 当一个RPC调用到达时指示工作已经处理完成并包含跟踪数据。为了确保代码级别的可见性，Span拥有带SpanEvent标签的子结构作为数据结构。每个Span包含一个TraceId。
- Trace: 多个Span的集合; 由关联的RPC (Spans)组成. 在同一个trace中的span共享相同的TransactionId。Trace通过SpanId和ParentSpanId整理为继承树结构.
- TraceId: 由 TransactionId, SpanId, 和 ParentSpanId 组成的key的集合. TransactionId 指明消息ID，而SpanId 和 ParentSpanId 表示RPC的父-子关系。
  - TransactionId (TxId): 在分布式系统间单个事务发送/接收的消息的ID; 必须跨整个服务器集群做到全局唯一.
  - SpanId: 当收到RPC消息时处理的工作的ID; 在RPC请求到达节点时生成。
  - ParentSpanId (pSpanId): 发起RPC调用的父span的SpanId. 如果节点是事务的起点，这里将没有父span - 对于这种情况， 使用值-1来表示这个span是事务的根span。

> Google Dapper 和 NAVER Pinpoint在术语上的不同
>
> Pinpoint中的术语"TransactionId"和google dapper中的术语"TraceId"有相同的含义。而Pinpoint中的术语"TraceId"引用到多个key的集合。

# TraceId如何工作

下图描述TraceId的行为，在4个节点之间执行了3次的RPC调用：

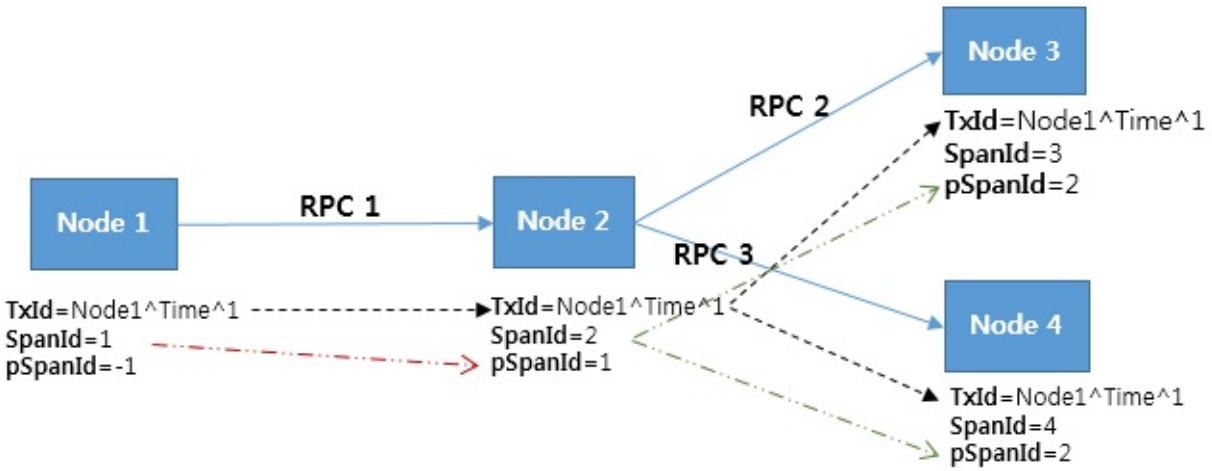



图2： TraceId行为示例

在图2中，TransactionId (TxId) 体现了三次不同的RPC作为单个事务被相互关联。但是，TransactionId 本身不能精确描述PRC之间的关系。为了识别PRC之间的关系，需要SpanId 和 ParentSpanId (pSpanId). 假设一个节点是Tomcat，可以将SpanId想象为处理HTTP请求的线程，ParentSpanId代表发起这个RPC调用的SpanId.

使用TransactionId，Pinpoint可以发现关联的n个Span，并使用SpanId和ParentSpanId将这n个span排列为继承树结构。

SpanId 和 ParentSpanId 是 64位长度的整型。可能发生冲突，因为这个数字是任意生成的，但是考虑到值的范围可以从-9223372036854775808到9223372036854775807，不太可能发生冲突. 如果key之间出现冲突，Pinpoint和Google Dapper系统，会让开发人员知道发生了什么，而不是解决冲突。

TransactionId 由 AgentIds, JVM (java虚拟机)启动时间, 和 SequenceNumbers/序列号组成.

- AgentId: 当Jvm启动时用户创建的ID; 必须在pinpoinit安装的全部服务器集群中全局唯一. 最简单的让它保持唯一的方法是使用hostname($HOSTNAME)，因为hostname一般不会重复. 如果需要在服务器集群中运行多个JVM，请在hostname前面增加一个前缀来避免重复。
- JVM 启动时间: 需要用来保证从0开始的SequenceNumber的唯一性. 当用户错误的创建了重复的AgentId时这个值可以用来预防ID冲突。
- SequenceNumber: Pinpoint agent 生成的ID, 从0开始连续自增;为每个消息生成一个.

Dapper 和 Zipkin, Twitter的一个分布式系统跟踪平台, 生成随机TraceIds (Pinpoint是TransactionIds) 并将冲突情况视为正常。然而, 在Pinpiont中我们想避免冲突的可能，因此实现了上面描述的系统。有两种选择：一是数据量小但是冲突的可能性高，二是数据量大但是冲突的可能性低。我们选择了第二种。

可能有更好的方式来处理transaction。我们起先有一个想法，通过中央key服务器来生成key。如果实现这个模式，可能导致性能问题和网络错误。因此，大量生成key被考虑作为备选。后面这个方法可能被开发。现在采用简单方法。在pinpoint中，TransactionId被当成可变数据来对待。

# 字节码增强，无需代码修改

前面我们解释了分布式事务跟踪。实现的方法之一是开发人员自己修改代码。当发生RPC调用时容许开发人员添加标签信息。但是，修改代码会成为包袱，即使这样的功能对开发人员非常有用。

Twitter的 Zipkin 使用修改过的类库和它自己的容器(Finagle)来提供分布式事务跟踪的功能。但是，它要求在需要时修改代码。我们期望功能可以不修改代码就工作并希望得到代码级别的可见性。为了解决这个问题，pinpoint中使用了字节码增强技术。Pinpoint agent干预发起RPC的代码以此来自动处理标签信息。

# 克服字节码增强的缺点

字节码增强在手工方法和自动方法两者之间属于自动方法。

- 手工方法： 开发人员使用ponpoint提供的API在关键点开发记录数据的代码
- 自动方法： 开发人员不需要代码改动，因为pinpoint决定了哪些API要调节和开发

下面是每个方法的优点和缺点：

Table1 每个方法的优缺点

| | 优点 | 缺点 |
|---|---|---|
| 手工跟踪 | 1. 要求更少开发资源 2. API可以更简单并最终减少bug的数量 | 1. 开发人员必须修改代码 2. 跟踪级别低 |
| 自动跟踪 | 1. 开发人员不需要修改代码 2. 可以收集到更多精确的数据因为有字节码中的更多信息 | 1. 在开发pinpoint时，和实现一个手工方法相比，需要10倍开销来实现一个自动方法 2. 需要更高能力的开发人员，可以立即识别需要跟踪的类库代码并决定跟踪点 3. 增加bug发生的可能性，因为使用了如字节码增强这样的高级开发技巧 |

字节码增强是一种高难度和高风险的技术。但是，综合考虑使用这种技术开所需要的资源和难度，使用它仍然有很多的益处。

虽然它需要大量的开发资源，在开发服务上它需要很少的资源。例如，下面展示了使用字节码增强的自动方法和使用类库的手工方法(在这里的上下文中，开销是为澄清而假设的随机数)之间的开销。

- 自动方法: 总共 100
    - Pinpoint开发开销: 100
    - 服务实施的开销: 0
- 手工方法: 总共 30
    - Pinpoint开发开销: 20
    - 服务实施的开销: 10

上面的数据告诉我们手工方法比自动方法有更合算。但是，不适用于我们的在NAVER的环境。在NAVER我们有几千个服务，因此在上面的数据中需要修改用于服务实施的开销。如果我们有10个服务需要修改，总开销计算如下：

Pinpoint开发开销 20 + 服务实施开销 10 x 10 = 120

基于这个结果，自动方法是一个更合算的方式。

我们很幸运的在pinpoint团队中拥有很多高能力而专注于Java的开发人员。因此，我们相信克服pinpoint开发中的技术难题只是个时间问题。

# 字节码增强的价值

我们选择字节码增强的理由，除了前面描述的那些外，还有下面的强有力的观点：

## 隐藏API

一旦API被暴露给开发人员使用，我们作为API的提供者，就不能随意的修改API。这样的限制会给我们增加压力。

我们可能修改API来纠正错误设计或者添加新的功能。但是，如果做这些受到限制，对我们来说很难改进API。解决这个问题的最好的答案是一个可升级的系统设计，而每个人都知道这不是一个容易的选择。如果我们不能掌控未来，就不可能创建完美的API设计。

而使用字节码增强技术，我们就不必担心暴露跟踪API而可以持续改进设计，不用考虑依赖关系。对于那些计划使用pinpoint开发应用的人，换一句话说，这代表对于pinpoint开发人员，API是可变的。现在，我们将保留隐藏API的想法，因为改进性能和设计是我们的第一优先级。

# 容易启用或者禁用

使用字节码增强的缺点是当Pinpoint自身类库的采样代码出现问题时可能影响应用。不过，可以通过启用或者禁用pinpoint来解决问题，很简单，因为不需要修改代码。

通过增加下面三行到JVM启动脚本中就可以轻易的为应用启用pinpoint：

```
-javaagent:$AGENT_PATH/pinpoint-bootstrap-$VERSION.jar
-Dpinpoint.agentId=<Agent's UniqueId>
-Dpinpoint.applicationName=<The name indicating a same service (AgentId collection)>
```

如果因为pinpoint发生问题，只需要在JVM启动脚本中删除这些配置数据。

# 字节码如何工作

由于字节码增强技术处理java字节码， 有增加开发风险的趋势，同时会降低效率。另外，开发人员更容易犯错。在pinpoint，我们通过抽象出拦截器(interceptor)来改进效率和可达性(accessibility)。pinpoint在类装载时通过介入应用代码为分布式事务和性能信息注入必要的跟踪代码。这会提升性能，因为代码注入是在应用代码中直接实施的。

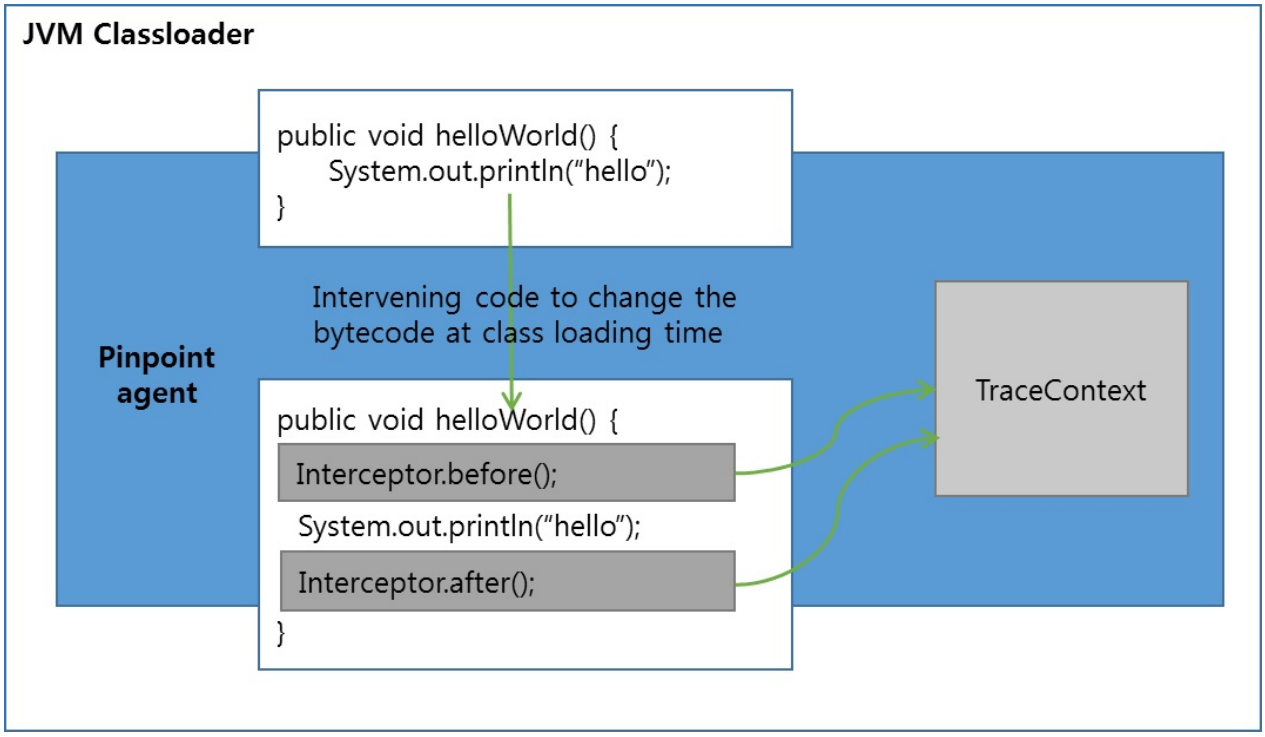



图3： 字节码增强行为

在pinpoint中，拦截器API在性能数据被记录的地方分开(separated)。为了跟踪，我们添加拦截器到目标方法使得before()方法和after()方法被调用，并在before()方法和after()方法中实现了部分性能数据的记录。使用字节码增强，pinpoint agent可以记录需要方法的数据，只有这样采样数据的大小才能变小。

# pinpoint agent的性能优化

最后，我们描述用于pinpoint agent的性能优化的方式。

## 使用二进制格式(thrift)

通过使用二进制格式(thrift)可以提高编码速度，虽然它使用和调试要难一些。也有利于减少网络使用，因为生成的数据比较小。

## 使用变长编码和格式优化数据记录

如果将一个长整型转换为固定长度的字符串， 数据大小一般是8个字节。然而，如果你用变长编码，数据大小可以是从1到10个字符，取决于给定数字的大小。为了减小数据大小，pinpoint使用thrift的CompactProtocol协议（压缩协议)来编码数据，因为变长字符串和记录数据可以为编码格式做优化。pinpoint agent通过基于跟踪的根方法的时间开始来转换其他的时间来减少数据大小。

图4 说明了上面章节描述的想法：

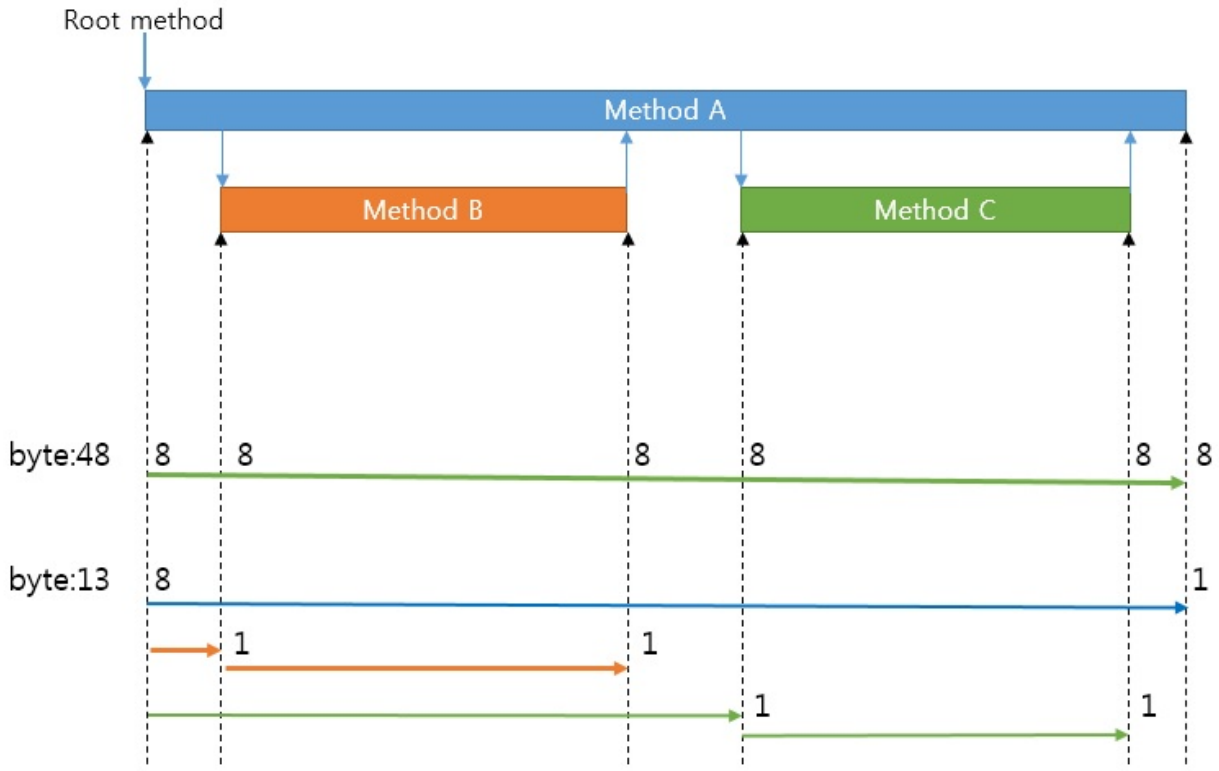



图4： 固定长度编码和可变长度编码的对比

为了得到关于三个不同方法（见图4）被调用时间的数据，不得不在6个不同的点上测量时间，用固定长度编码这需要48个字节(6 * 8)。

以此同时，pinpoint agent 使用可变长度编码并根据对应的格式记录数据。然后在其他时间点通过和参考点比较来计算时间值（在vector中），根方法的起点被确认为参考点。这只需要占用少量的字节，因为vector使用小数字。图4中消耗了13个字节。

如果执行方法花费了更多时间，即使使用可变长度编码也会增加字节数量。但是，依然比固定长度编码更有效率。

# 用常量表替换重复的API信息，SQL语句和字符串

我们希望pinpoint能开启代码级别的跟踪。然而，存在增大数据大小的问题。每次高精度的数据被发送到服务器将增大数据大小，导致增加网络使用。

为了解决这个问题，我们使用了在远程HBase中创建常量表的策略。例如，每次调用"Method A"的信息被发送到pinpoint collector， 数据大小将很大。pinpoint agent 用一个ID替换"method A"，在HBase中作为一个常量表保存ID和"method A"的信息，然后用ID生成跟踪数据。然后当用户在网站上获取跟踪数据时，pinpoint web在常量表中搜索对应ID的方法信息并组合他们。使用同样的方式来减少SQL或者频繁使用的字符串的数据大小。

# 处理大量请求的采样

我们在线门户服务有海量请求。单个服务每天处理超过200亿请求。容易跟踪这样的请求：方法是添加足够多的网络设施和服务器来跟踪请求并扩展服务器来收集数据。然后，这不是处理这种场景的合算的方法，仅仅是浪费金钱和资源。

在Pinpoint，可以收集采样资料而不必跟踪每个请求。在开发环境中请求量很小，每个数据都收集。而在产品环境请求量巨大，收集小比率的数据如1~5%，足够检查整个应用的状态。有采样后，可以最小化应用的网络开销并降低诸如网络和服务器的设施费用。

> pinpoint采样方法
>
> Pinpoint 支持计数采样，如果设置为10则只采样10分之一的请求。我们计划增加新的采样器来更有效率的收集数据。

注：对应的配置项在agent下的pinpoint.config文件中，默认"profiler.sampling.rate=1"表示全部

# 使用异步数据传输来最小化应用线程中止

pinpoint不阻塞应用线程，因为编码后的数据或者远程消息被其他线程异步传输。

## 使用UDP传输数据

和gogole dapper不同，pinpoint通过网络传输数据来确保数据速度。作为一个服务间使用的通用设施，当数据通讯持续突发时网络会成为问题。在这种情况下，pinpoint agent使用UDP协议来给服务让出网络连接优先级。

> 注意
>
> 数据传输API可以被替换，因为它是接口分离的。可以修改实现为用其他方式存储数据，比如本地文件。

# pinpoint应用示例

这里给出一个例子关于如何从应用获取数据，这样就可以全面的理解前面讲述的内容。

图5 展示了当在 TomcatA 和 TomcatB 中安装pinpoint的数据。可以把单个节点的跟踪数据看成single traction，提现分布式事务跟踪的流程。

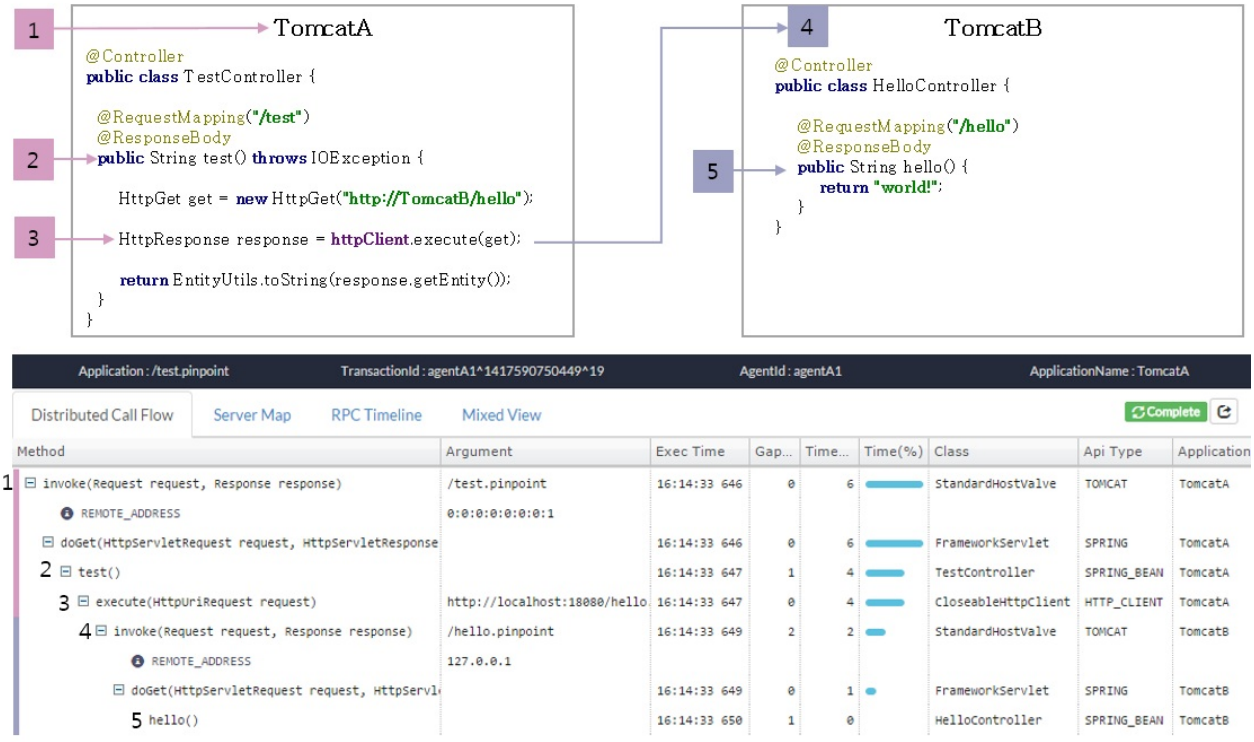



图5：示例1：pinpoint应用

下面阐述pinpoint为每个方法做了什么：

1. 当请求到达TomcatA时, Pinpoint agent 产生一个 TraceId.

    - TX_ID: TomcatA\^TIME\^1
    - SpanId: 10
    - ParentSpanId: -1(Root)
2. 从spring MVC 控制器中记录数据
3. 插入HttpClient.execute()方法的调用并在HTTPGet中配置TraceId
    - 创建一个子TraceId
        - TX_ID: TomcatA\^TIME\^1 -> TomcatA\^TIME\^1
        - SPAN_ID: 10 -> 20
        - PARENT_SPAN_ID: -1 -> 10 (父 SpanId)
    - 在HTTP header中配置子 TraceId
        - HttpGet.setHeader(PINPOINT_TX_ID, "TomcatA\^TIME\^1")
        - HttpGet.setHeader(PINPOINT_SPAN_ID, "20")
        - HttpGet.setHeader(PINPOINT_PARENT_SPAN_ID, "10")
4. 传输打好tag的请求到TomcatB.
    - TomcatB 检查传输过来的请求的header

      HttpServletRequest.getHeader(PINPOINT_TX_ID)

    - TomcatB 作为子节点工作因为它识别了header中的TraceId
        - TX_ID: TomcatA\^TIME\^1
        - SPAN_ID: 20
        - PARENT_SPAN_ID: 10
5. 从spring mvc控制器中记录数据并完成请求

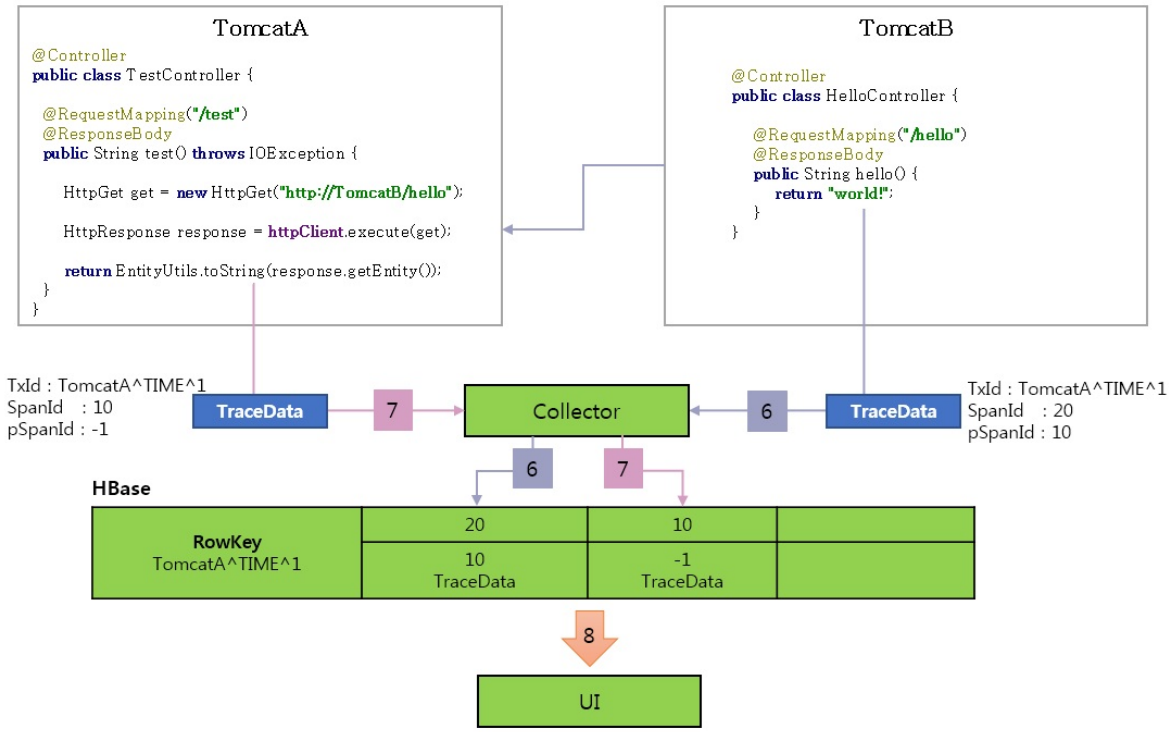



图6 示例2：pinpoint应用

6. 当从tomcatB回来的请求完成时，pinpoint agent发送跟踪数据到pinpoint collector就此存储在HBase中
7. 在对tomcatB的HTTP调用结束后，TomcatA的请求也完成了。pinpoint agent发送跟踪数据到pinpoint collector就此存储在HBase中
8. UI从HBase中读取跟踪数据并通过排序树来创建调用栈

# 结论

pinpoint是和应用一起运行的另外的应用。使用字节码增强使得pinpoint看上去不需要代码修改。通常，字节码增强技术让应用容易造成风险。如果问题发生在pinpoint中，它会影响应用。目前，我们专注于改进pinpoint的性能和设计，而不是移除这样的威胁，因为我们任务这些让pinpoint更加有价值。因此你需要决定是否使用pinpoint。

我们还是有大量的工作需要完成来改进pinpoint，尽管不完整，pinpoint还是作为开源项目发布了。我们将持续努力开发并改进pinpoint以便满足你的期望。

> Woonduk Kang编写
>
> 在2011年, 关于我自己我这样写到 — 作为一个开发人员，我想开发人们愿意付款的软件程序，像Microsoft 或者 Oracle. 当Pinpoint被作为一个开源项目启动，看上去我的梦想稍微实现了一点。目前， 我的愿望是让pinpoint对用户更加有价值和更惹人喜欢.

# pinpoint 插件

# Pinpoint Profiler 插件示例

> 注：内容翻译自官方文档 Plugin Sample.

可以通过编写 Pinlpoint profiler 插件来扩展 Pinpoint 的profile能力。这个示例项目展示如何编写它。它包含3个模块：

- plugin-sample-target: 目标类库
- plugin-sample-plugin: 示例插件
- plugin-sample-agent: 带有示例插件的agent发行包

# 实现 Profiler 插件

Pinpoint profiler 插件必须提供 ProfilerPlugin 和 TraceMetadataProvider 的实现. ProfilerPlugin 仅被Pinpoint Agent使用， 而 TraceMetadataProvider 被 Pinpoint Agent, Collector 和 Web使用.

Pinpoint通过 Java ServiceLoader 机制来装载这些实现。因此 plugin 的 JAR 必须包含两个 provider-configuration 文件：

- META-INF/services/com.navercorp.pinpoint.bootstrap.plugin.ProfilerPlugin
- META-INF/services/com.navercorp.pinpoint.common.trace.TraceMetadataProvider

每个文件应该包含实现类的全限定名。

# TraceMetadataProvider

TraceMetadataProvider 添加 ServiceTypes 和 AnnotationKeys 到 Pinpoint.

ServiceType 和 AnnotationKey的编码值必须唯一. 如果编写一个私有插件， 可以使用为私下使用保留的编码值。Pinpoint不会给任何东西分配这些值。否则需要联系 Pinpoint dev team 来为插件分配编码。

- 私下使用的ServiceType编码
    - Server: 1900 ~ 1999
    - DB client: 2900 ~ 2999
    - ~~Cache client: 8999 ~ 8999~~ (原文档笔误，已经和pinpoint确认，具体见 issue)
    - Cache client: 8900 ~ 8999
    - RPC client: 9900 ~ 9999

    - Others: 7500 ~ 7999
- 私下使用的AnnotaionKey编码
    - 900 ~ 999

# ProfilerPlugin

ProfilerPlugin 添加 TransformCallbacks 到 Pinpoint.

TransformCallback 通过添加interceptors, getters 和/或 fields来转换目标类。可以在plugin-sample-plugin 项目中找到示例代码.

# 集成测试

可以用 PinointPluginTestSuite (一个JUnit Runner)来运行插件集成测试。它从maven仓库下载需要的依赖并启动一个新的JVM，这个JVM带有Pinpoint profiler agent和依赖。JUnit 测试在这个JVM上执行。

为了运行集成测试，需要一个完整的agent发行包。这也是为什么集成测试放在 plugin-sample-agent 模块中。

在测试中，可以使用 PluginTestVerifier 来检查跟踪信息/trace是否被正确记录。

## 测试依赖

PinointPluginTestSuite 不使用项目的依赖(配置在pom.xml中). 它使用通过 @Dependency 列出的依赖。以这种方式，可以测试目标类库的多个版本。

依赖这些定义， 你可以指定依赖的版本或者版本范围：

```
@Dependency({"some.group:some-artifact:1.0", "another.group:another-artifact:2.1-RELEA
SE"})
@Dependency({"some.group:some-artifact:[1.0,)"})
@Dependency({"some.group:some-artifact:[1.0,1.9]"})
@Dependency({"some.group:some-artifact:[1.0],[2.1],[3.2])"})
```

PinointPluginTestSuite 从本地仓库和maven中央仓库中搜索依赖。可以通过@Repository添加仓库。

# Jvm 版本

可以通过@JvmVersion为测试指定 JVM 版本.

# 应用测试

PinpointPluginTestSuite 不适合用于那些需要通过自己的main class启动的应用。可以扩展 AbstractPinpointPluginTestSuite 和相关类型来测试这样的应用。

# Pinpoint 插件设计

# 现有实现

# Gson插件

# 告警

TBD

# Alarm

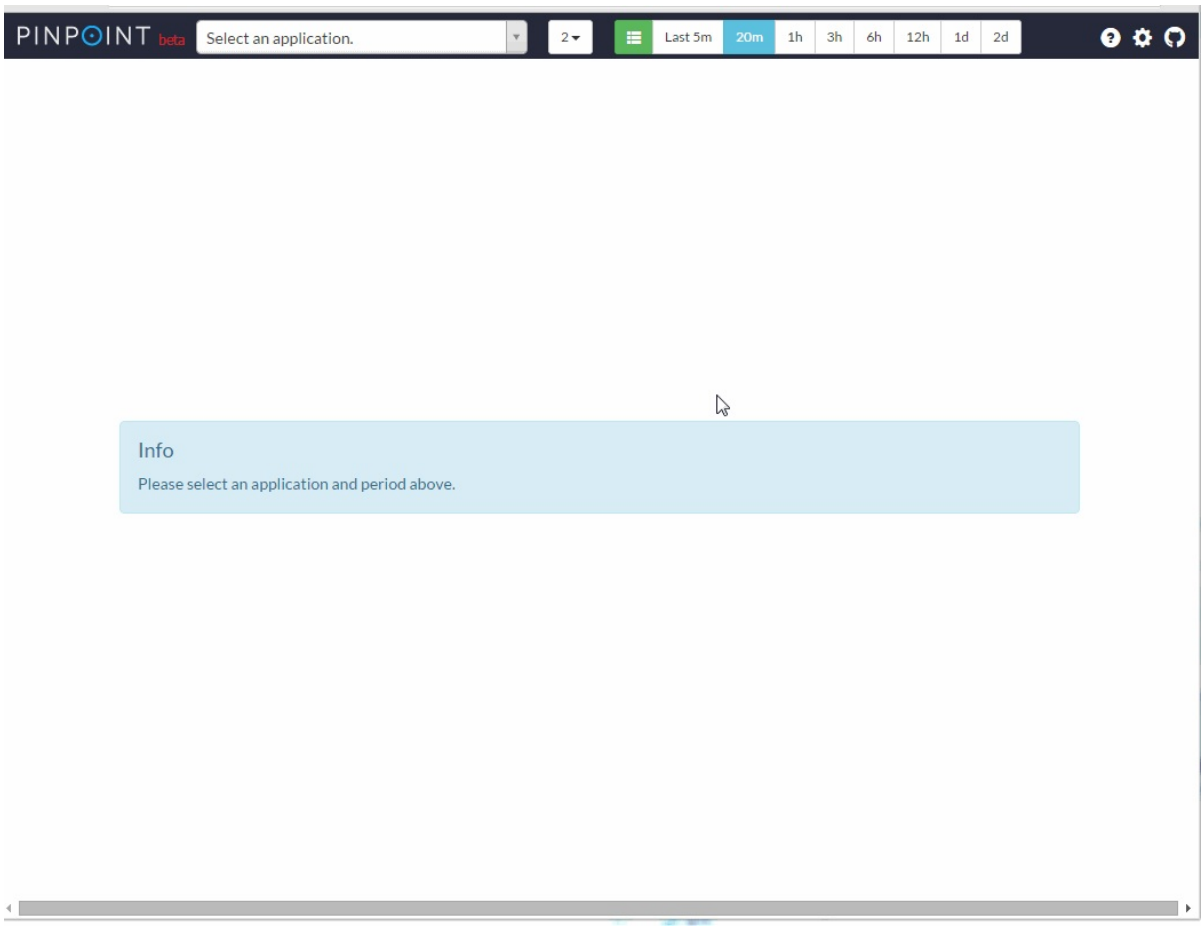
> 注：内容翻译自 官方文档Alarm

Pinpoint-web周期性的检查应用的状态，如果特定前置条件(规则)满足时则触发告警。

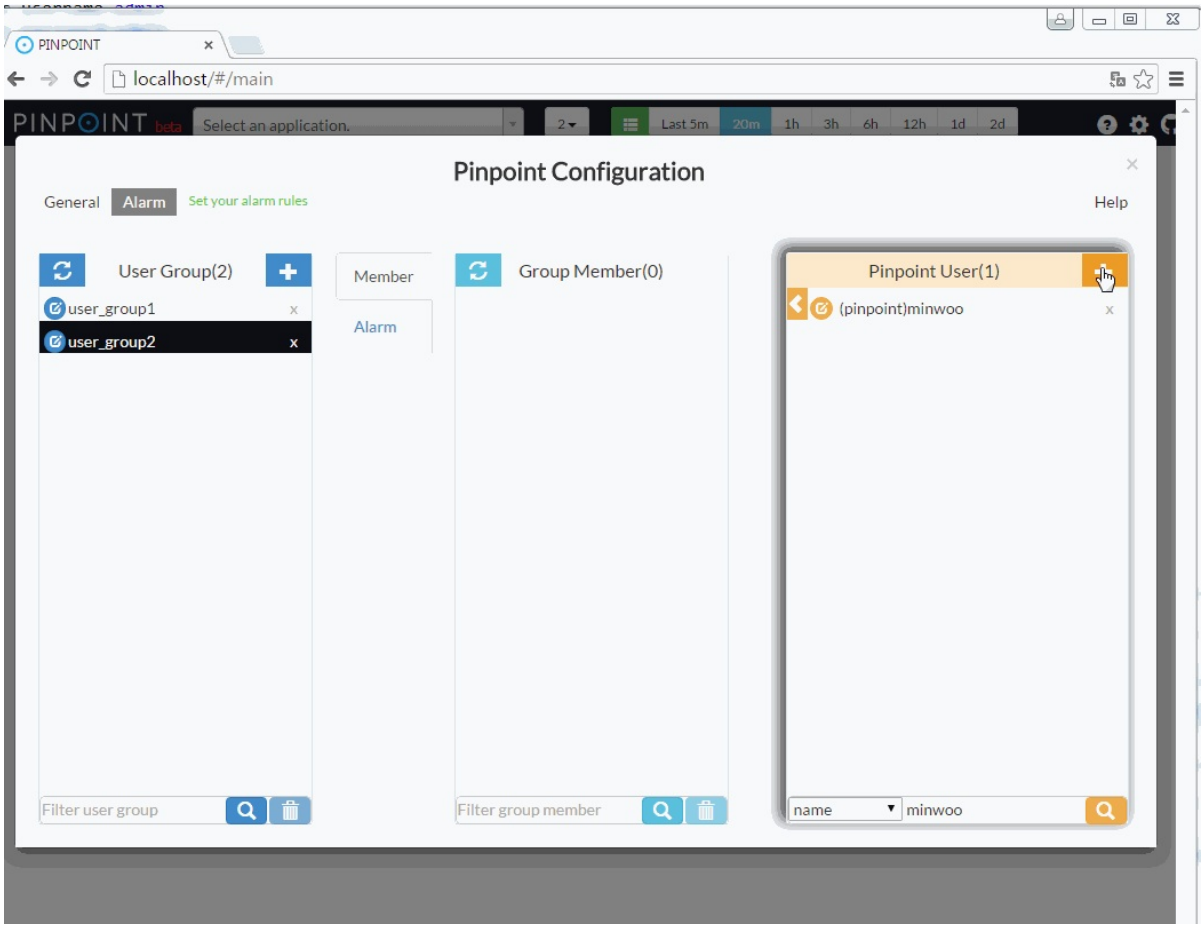
这些条件(默认)每3分钟被web模块中的后台批处理程序检查一次，使用最后5分钟的数据。一旦条件满足，批处理程序发送短信/邮件给注册到用户组的用户。

## 用户指南

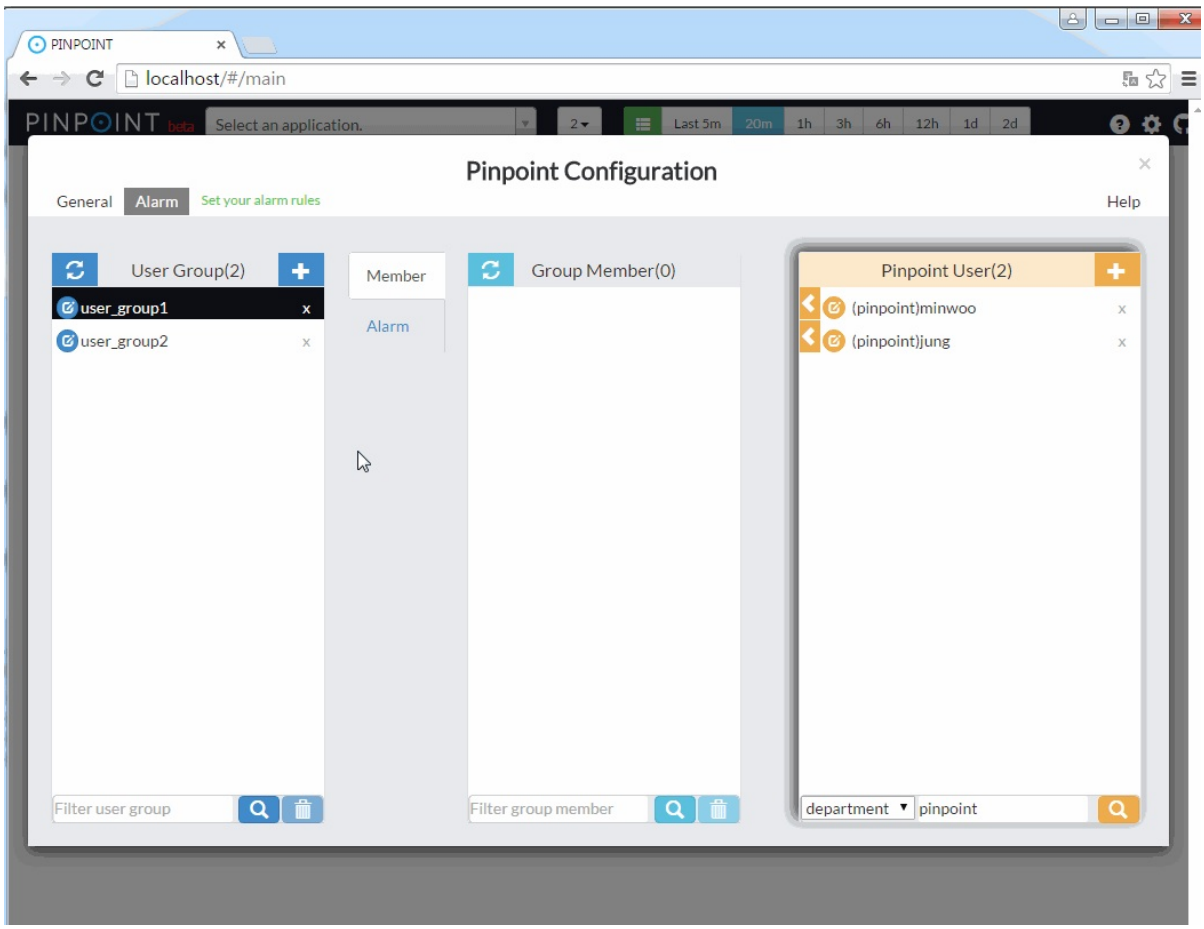
1. 配置菜单



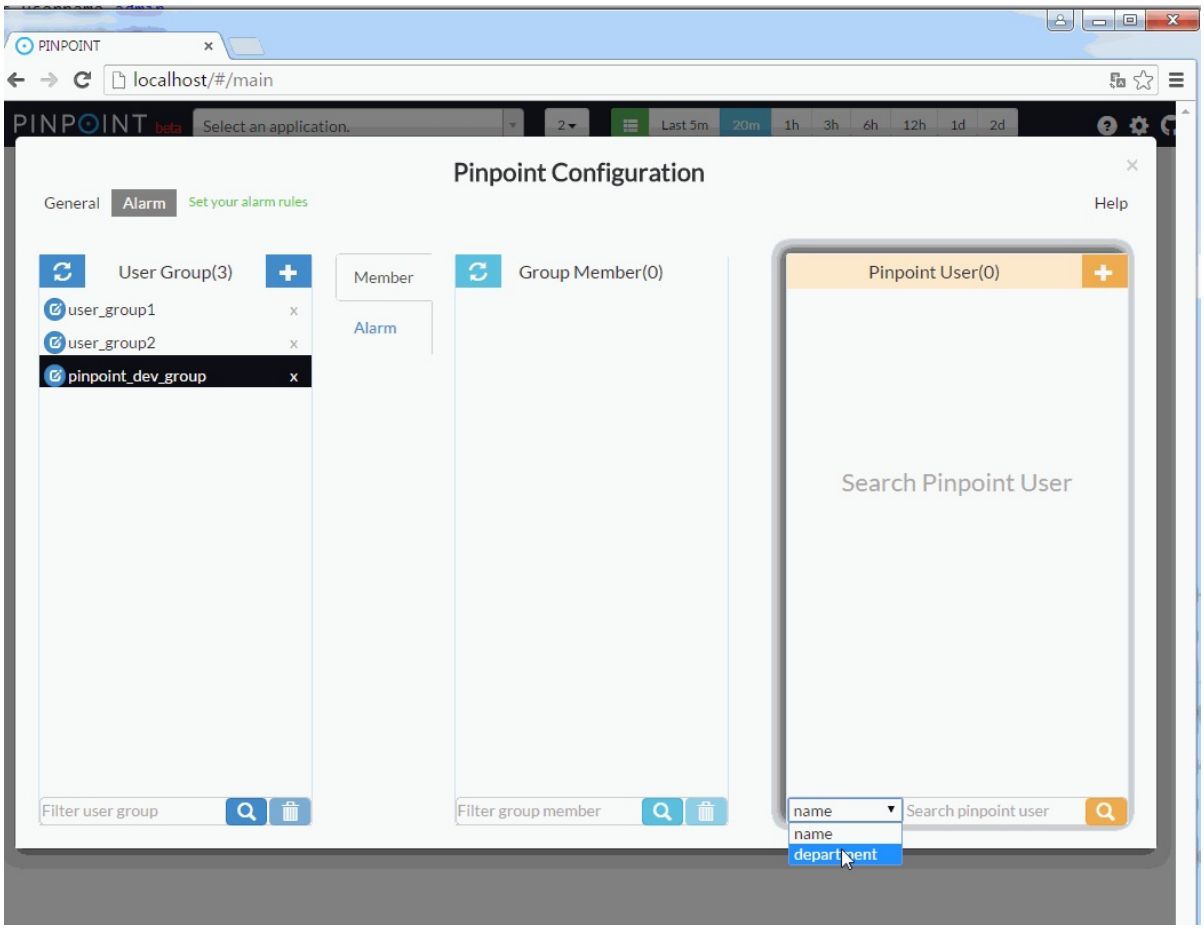

2. 注册用户


PINPOINT
localhost/#/main
PINPOINT beta
Select an application.
Last 5m
20m
1h
3h
6h
12h
1d
2d
Pinpoint Configuration
General
Alarm
Set your alarm rules
Help
User Group(2)
user_group1
user_group2
Member
Alarm
Group Member(0)
Pinpoint User(1)
(pinpoint)minwoo
Filter user group
Filter group member
name
minwoo

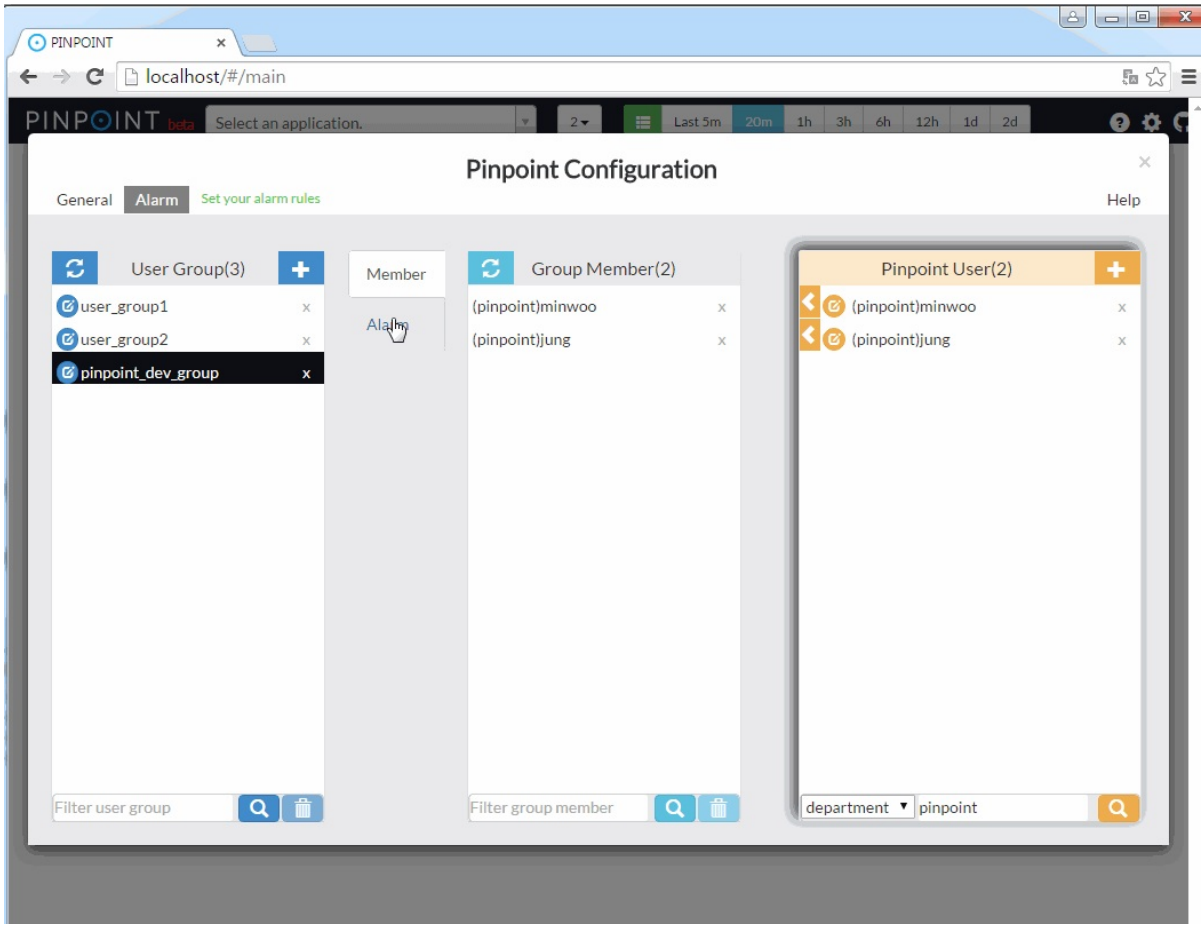
3. 创建用户组


PINPOINT
localhost/#/main
PINPOINT beta
Select an application.
Last 5m
20m
1h
3h
6h
12h
1d
2d
Pinpoint Configuration
General
Alarm
Set your alarm rules
Help
User Group(2)
user_group1
user_group2
Member
Alarm
Group Member(0)
Pinpoint User(2)
(pinpoint)minwoo
(pinpoint)jung
Filter user group
Filter group member
department
pinpoint

4. 添加用户到用户组




5. 设置告警规则




# 告警规则

- SLOW COUNT / 慢请求数

  当应用发出的慢请求数量超过配置阈值时触发。

- SLOW RATE / 慢请求比例

  当应用发出的慢请求百分比超过配置阈值时触发。

- ERROR COUNT / 请求失败数

  当应用发出的失败请求数量超过配置阈值时触发。

- ERROR RATE / 请求失败率

  当应用发出的失败请求百分比超过配置阈值时触发。

- TOTAL COUNT / 总数量

  当应用发出的所有请求数量超过配置阈值时触发。

> 以上规则中，请求是当前应用发送出去的，当前应用是请求的发起者。 以下规则中，请求是发送给当前应用的，当前应用是请求的接收者。

- SLOW COUNT TO CALLEE / 被调用的慢请求数量

  当发送给应用的慢请求数量超过配置阈值时触发。

- SLOW RATE TO CALLEE / 被调用的慢请求比例

  当发送给应用的慢请求百分比超过配置阈值时触发。

- ERROR COUNT TO CALLEE / 被调用的请求错误数

  当发送给应用的请求失败数量超过配置阈值时触发。

- ERROR RATE TO CALLEE / 被调用的请求错误率

  当发送给应用的请求失败百分比超过配置阈值时触发。

- TOTAL COUNT TO CALLEE / 被调用的总数量

  当发送给应用的所有请求数量超过配置阈值时触发。

  > 下面两条规则和请求无关，只涉及到应用的状态

- HEAP USAGE RATE / 堆内存使用率

  当应用的堆内存使用率超过配置阈值时触发。

- JVM CPU USAGE RATE / JVM CPU使用率

  当应用的CPU使用率超过配置阈值时触发。

# 实现和配置

为了使用告警功能，必须通过实现 com.navercorp.pinpoint.web.alarm.AlarmMessageSender 并注册为spring managed bean 来实现自己的逻辑以便发送短信和邮件。当告警被触发时，AlarmMessageSender#sendEmail, 和 AlarmMessageSender#sendSms 方法将被调用。

# 实现**AlarmMessageSender** 并注册 **Spring bean**

```
public class AlarmMessageSenderImple implements AlarmMessageSender {

    private final Logger logger = LoggerFactory.getLogger(this.getClass());

    @Override
    public void sendSms(AlarmChecker checker, int sequenceCount) {
        List<String> receivers = userGroupService.selectPhoneNumberOfMember(checker.ge
tUserGroupId());

        if (receivers.size() == 0) {
            return;
        }

        for (String message : checker.getSmsMessage()) {
            logger.info("send SMS : {}", message);

            // TODO Implement logic for sending SMS
        }
    }

    @Override
    public void sendEmail(AlarmChecker checker, int sequenceCount) {
        List<String> receivers = userGroupService.selectEmailOfMember(checker.getUserG
roupId());

        if (receivers.size() == 0) {
            return;
        }

        for (String message : checker.getEmailMessage()) {
            logger.info("send email : {}", message);

            // TODO Implement logic for sending email
        }
    }
}
```

```
<bean id="AlarmMessageSenderImple" class="com.navercorp.pinpoint.web.alarm.AlarmMessag
eSenderImple"/>
```

> 注：以上代码copy自原文，谨慎请见，开发时请以英文原文为准。

# 配置批处理属性

设置batch.properties文件中的 batch.enable 标记为true：

```
batch.enable=true
```

这里的 batch.server.ip 配置用于当有多台pinpoint web server时防止并发批处理程序. 仅当服务器IP地址和 batch.server.ip 设置的一致时才执行批处理。(设置为 127.0.0.1 将在所有的web服务器上启动批处理)

```
batch.server.ip=X.X.X.X
```

> 注： 这种防止并发的方式有点简陋而原始，存在单点故障的风险，主要缺陷：万一配置的这台容许批处理的web服务器down机，告警功能就失效了。

# 配置mysql

搭建mysql服务器并在jdbc.properties文件中配置连接信息：

```
jdbc.driverClassName=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://localhost:13306/pinpoint?characterEncoding=UTF-8
jdbc.username=admin
jdbc.password=admin
```

运行 CreateTableStatement-mysql.sql 和 SpringBatchJobReositorySchema-mysql.sql 来创建表.

# 其他

1. 可以在独立进程中启动告警批处理

   使用Pinpoint-web模块中的 applicationContext-alarmJob.xml 文件简单启动spring batch任务.

2. 通过修改 applicationContext-batch-schedule.xml 文件中的cron 表达式来修改批处理周期：

   ```
   <task:scheduled-tasks scheduler="scheduler">
       <task:scheduled ref="batchJobLauncher" method="alarmJob" cron="0 0/3 * * * *"
   />
   </task:scheduled-tasks>
   ```

3. 提高告警批处理性能的方式

告警批处理被设计为并发运行. 如果有很多应用注册有告警，可以通过修改 applicationContext-batch.xml 文件中的 pool-size 来增大executor的线程池大小.

注意增大这个值会导致更高的资源使用。

```
<task:executor id="poolTaskExecutorForPartition" pool-size="1" />
```

如果有应用注册有很多告警，可以设置注册在 applicationContext-batch.xml 文件中的 alarmStep来并发运行：

```
<step id="alarmStep" xmlns="http://www.springframework.org/schema/batch">
    <tasklet task-executor="poolTaskExecutorForStep" throttle-limit="3">
        <chunk reader="reader" processor="processor" writer="writer" commit-inter
val="1"/>
    </tasklet>
</step>
<task:executor id="poolTaskExecutorForStep" pool-size="10" />
```

# 现有代码实现

## 代码入口

### applicationContext-web.xml

文件路径: pinpoint/web/src/main/resources/applicationContext-web.xml

导入的配置文件有hbase.properties和jdbc.properties：

```
<bean id="propertyConfigurer" class="org.springframework.beans.factory.config.Property
PlaceholderConfigurer">
    <property name="locations">
        <list>
            <value>classpath:hbase.properties</value>
            <value>classpath:jdbc.properties</value>
        </list>
    </property>
</bean>
```

其他导入的spring配置文件：

```
<import resource="classpath:applicationContext-hbase.xml" />
<import resource="classpath:applicationContext-datasource.xml" />
<import resource="classpath:applicationContext-dao-config.xml" />
<import resource="classpath:applicationContext-cache.xml" />
<import resource="classpath:applicationContext-websocket.xml" />
```

## 批处理

### 类 BatchConfiguration

文件路径：
pinpoint/web/src/main/java/com/navercorp/pinpoint/web/batch/BatchConfiguration.java

```
@Configuration
@Conditional(BatchConfiguration.Condition.class)
@ImportResource("classpath:/batch/applicationContext-batch-schedule.xml")
public class BatchConfiguration{
    static class Condition implements ConfigurationCondition {
        @Override
        public boolean matches(ConditionContext context, AnnotatedTypeMetadata metadat
a) {
            ......
            Resource resource = context.getResourceLoader().getResource("classpath:/ba
tch.properties")
            ......
            final String enable = properties.getProperty("batch.enable");
            ......
        }
    }
}
```

Condition中会读取配置文件batch.properties中的配置项batch.enable，默认是false。因此如果要开启批处理功能，必须设置batch.enable=true。

## applicationContext-batch-schedule.xml

文件路径为：pinpoint/web/src/main/resources/batch/applicationContext-batch-schedule.xml

```
<task:scheduled-tasks scheduler="scheduler">
    <task:scheduled ref="batchJobLauncher" method="alarmJob" cron="0 0/3 * * * *" />
</task:scheduled-tasks>
```

为了测试方便，可以修改cron表达式为 cron="/5 *"，每5秒钟执行一次。

## batch.properties

文件路径为：pinpoint/web/src/main/resources/batch.properties

```
#batch enable config
batch.enable=true

#batch server ip to execute batch
batch.server.ip=127.0.0.1
```

设置batch.enable=true，另外设置batch.server.ip=127.0.0.1这样每台pinpoint web都会跑批处理。如果安装有多台pinpoint web，可以设置为其中一台的IP。

# 数据源

## applicationContext-datasource.xml

文件路径：pinpoint/web/src/main/resources/applicationContext-datasource.xml

```
<!-- DataSource Configuration -->
<bean id="dataSource" class="org.apache.commons.dbcp2.BasicDataSource" destroy-method=
"close">
    <property name="driverClassName" value="${jdbc.driverClassName}"/>
    <property name="url" value="${jdbc.url}"/>
    <property name="username" value="${jdbc.username}"/>
    <property name="password" value="${jdbc.password}"/>
    ......
</bean>
```

定义了名为dataSource的数据源给其他spring bean使用，配置信息来自jdbc.properties。

## jdbc.properties

文件路径：pinpoint/web/src/main/resources/jdbc.properties

```
jdbc.driverClassName=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://localhost:13306/pinpoint?characterEncoding=UTF-8
jdbc.username=admin
jdbc.password=admin
```

定义了名为dataSource的数据源，使用mysql。

# 定制自己的实现

# Tags