# Skywalking链路监控

## 1、Skywalking概述

**请求链路追踪，故障快速定位**：可以通过调用链结合业务日志快速定位错误信息。

**可视化**：各个阶段耗时，进行性能分析。

**依赖优化**：各个调用环节的可用性、梳理服务依赖关系以及优化。

**数据分析，优化链路**：可以得到用户的行为路径，汇总分析应用在很多业务场景

### 1.1 APM系统概述

APM (Application Performance Management) 即应用性能管理系统，是对企业系统即时监控以实现 对应用程序性能管理和故障管理的系统化的解决方案。应用性能管理，主要指对企业的关键业务应用进 行监测、优化，提高企业应用的可靠性和质量，保证用户得到良好的服务，降低IT总拥有成本。

**APM系统是可以帮助理解系统行为、用于分析性能问题的工具，以便发生故障的时候，能够快速定位和解决问题。**

### 1.2 分布式链路追踪

随着分布式系统和微服务架构的出现，一次用户的请求会经过多个系统，不同服务之间的调用关系十分 复杂，任何一个系统出错都可能影响整个请求的处理结果。以往的监控系统往往只能知道单个系统的健 康状况、一次请求的成功失败，无法快速定位失败的根本原因。

### 1.3 主流的开源APM产品

**SkyWalking** SkyWalking是apache基金会下面的一个开源APM项目，为微服务架构和云原生架构系统设计。它通过探针自动收集所需的指标，并进行分布式追踪。通过这些调用链路以及指标，Skywalking APM会感知应用间关系和服务间关系，并进行相应的指标统计。Skywalking支持链路追踪和监控应用组件基本涵盖

**Zipkin** Zipkin是由Twitter开源，是分布式链路调用监控系统，聚合各业务系统调用延迟数据，达到链路调用监控跟踪。Zipkin基于Google的Dapper论文实现，主要完成数据的收集、存储、搜索与界面展示。

**PinPoint** Pinpoint是由一个韩国团队实现并开源，针对Java编写的大规模分布式系统设计，通过JavaAgent的机制做字节代码植入，实现加入traceid和获取性能数据的目的，对应用代码零侵入。

**CAT** CAT是由大众点评开源的项目，基于Java开发的实时应用监控平台，包括实时应用监控，业务监控，可以提供十几张报表展示。

## 2、Skywalking原理

### 2.1 java agent原理

使用Skywalking去监控服务，需要在其 VM 参数中添加 “- javaagent:/usr/local/skywalking/agent//skywalking-agent.jar"。这里就 使用到了java agent技术。

**2.1.1、Java agent 是什么？**

Java agent是java命令的一个参数。参数 javaagent 可以用于指定一个 jar 包。

1. 这个 jar 包的 MANIFEST.MF 文件必须指定 Premain-Class 项。
2. Premain-Class 指定的那个类必须实现 premain() 方法。

当Java 虚拟机启动时，在执行 main 函数之前，JVM 会先运行 -javaagent所指定 jar 包内 Premain- Class 这个类的 premain 方法 。

**2.1.2、如何使用java agent？**

使用 java agent 需要几个步骤：

1. 定义一个 MANIFEST.MF 文件，必须包含 Premain-Class 选项，通常也会加入Can-Redeﬁne- Classes 和 Can-Retransform-Classes 选项。
2. 创建一个Premain-Class 指定的类，类中包含 premain 方法，方法逻辑由用户自己确定。
3. 将 premain 的类和 MANIFEST.MF 文件打成 jar 包。
4. 使用参数 -javaagent: jar包路径 启动要代理的方法。

**示例代码**

import net.bytebuddy.agent.builder.AgentBuilder;  
import net.bytebuddy.description.method.MethodDescription;  
import net.bytebuddy.description.type.TypeDescription;  
import net.bytebuddy.dynamic.DynamicType;  
import net.bytebuddy.implementation.MethodDelegation;  
import net.bytebuddy.matcher.ElementMatchers;  
import net.bytebuddy.utility.JavaModule;  
  
import java.lang.instrument.Instrumentation;  
  
public class PreMainAgent {  
  
 public static void premain(String agentArgs, Instrumentation inst) {  
 System.out.println("=========premain方法执行1========");  
 System.out.println(agentArgs);  
 }  
}

在pom文件中添加打包插件

<build>  
 <plugins>  
 <plugin>  
 <artifactId>maven-assembly-plugin</artifactId>  
 <configuration>  
 <appendAssemblyId>false</appendAssemblyId>  
 <descriptorRefs>  
 <descriptorRef>jar-with-dependencies</descriptorRef>  
 </descriptorRefs>  
 <archive>  
 <!--自动添加META-INF/MANIFEST.MF -->  
 <manifest>  
 <addClasspath>true</addClasspath>  
 </manifest>  
 <manifestEntries>  
 <Premain-Class>PreMainAgent</Premain-Class>  
 <Agent-Class>PreMainAgent</Agent-Class>  
 <Can-Redefine-Classes>true</Can-Redefine-Classes>  
 <Can-Retransform-Classes>true</Can-Retransform-Classes>  
 </manifestEntries>  
 </archive>  
 </configuration>  
 <executions>  
 <execution>  
 <id>make-assembly</id>  
 <phase>package</phase>  
 <goals>  
 <goal>single</goal>  
 </goals>  
 </execution>  
 </executions>  
 </plugin>  
 </plugins>  
 </build>

该插件会在自动生成META-INF/MANIFEST.MF文件时，帮我们添加agent相关的配置信息。 使用maven的package命令进行打包：

**2.1.3、 统计方法调用时间**

Skywalking中对每个调用的时长都进行了统计，使用ByteBuddy和Java agent技术来 统计方法的调用时长。

Byte Buddy是开源的、基于Apache 2.0许可证的库，它致力于解决字节码操作和instrumentation API 的复杂性。Byte Buddy所声称的目标是将显式的字节码操作隐藏在一个类型安全的领域特定语言背 后。通过使用Byte Buddy，任何熟悉Java编程语言的人都有望非常容易地进行字节码操作。Byte Buddy提供了额外的API来生成Java agent，可以轻松的增强我们已有的代码。

添加依赖：

<dependencies>  
 <dependency>  
 <groupId>net.bytebuddy</groupId>  
 <artifactId>byte-buddy</artifactId>  
 </dependency>  
 <dependency>  
 <groupId>net.bytebuddy</groupId>  
 <artifactId>byte-buddy-agent</artifactId>  
 </dependency>  
</dependencies>

修改PreMainAgent代码

public class PreMainAgent {  
  
 public static void premain(String agentArgs, Instrumentation inst) {  
   
 //创建一个转换器，转换器可以修改类的实现  
 //ByteBuddy对java agent提供了转换器的实现，直接使用即可  
 AgentBuilder.Transformer transformer = new AgentBuilder.Transformer() {  
 public DynamicType.Builder<?> transform(DynamicType.Builder<?> builder, TypeDescription typeDescription, ClassLoader classLoader, JavaModule javaModule) {  
 return builder  
 // 拦截任意方法  
 .method(ElementMatchers.<MethodDescription>any())  
 // 拦截到的方法委托给TimeInterceptor  
 .intercept(MethodDelegation.to(MyInterceptor.class));  
 }  
 };  
   
 new AgentBuilder // Byte Buddy专门有个AgentBuilder来处理Java Agent的场景  
 .Default()  
 // 根据包名前缀拦截类  
 .type(ElementMatchers.nameStartsWith("com.agent"))  
 // 拦截到的类由transformer处理  
 .transform(transformer)  
 .installOn(inst);  
 }  
}

先生成一个转换器，ByteBuddy提供了java agent专用的转换器。通过实现Transformer接口利用 builder对象来创建一个转换器。转换器可以配置拦截方法的格式，比如用名称，本例中拦截所有方 法，并定义一个拦截器类MyInterceptor。

public class MyInterceptor {  
 @RuntimeType  
 public static Object intercept(@Origin Method method,  
 @SuperCall Callable<?> callable)  
 throws Exception {  
 long start = System.currentTimeMillis();  
 try {  
 //执行原方法  
 return callable.call();  
 } finally {  
 //打印调用时长  
 System.out.println(method.getName() + ":" + (System.currentTimeMillis() - start) + "ms");  
 }  
 }  
}

MyInterceptor就是一个拦截器的实现，统计的调用的时长。参数中的method是反射出的方法对象，而 callable就是调用对象，可以通过callable.call（）方法来执行原方法。

## 3、Skywalking 环境搭建（集群）

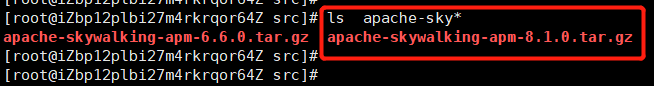
### 3.1 软件准备

软件包版本已经上传140Gitlab

apache-skywalking-apm-8.1.0.tar.gz

http://39.100.254.140:12011/loit-Infrastructure-doc/loit-initproject-doc/tree/master/3%E3%80%81other/%E9%9B%86%E6%88%90loit%E9%97%A8%E6%88%B7/skywalking/8.1.0/apache-skywalking-apm-8.1.0.tar.gz

将以上三个软件包上传至服务器/usr/local/src/路径下



### 3.2 安装Skywalking服务端

**解压安装包**

cd /usr/local/src  
  
tar -zxvf apache-skywalking-apm-8.1.0.tar.gz  
mv apache-skywalking-apm-bin skywalking  
mv skywalking/ /usr/local/

**修改配置文件**

vim /usr/local/skywalking/config/application.yml

1、其中修改注册中心nacos

|  |
| --- |
| cluster:  selector: ${SW\_CLUSTER:nacos}  nacos:  serviceName: ${SW\_SERVICE\_NAME:"SkyWalking\_OAP\_Cluster"}  hostPort: ${SW\_CLUSTER\_NACOS\_HOST\_PORT:172.16.203.79:8010}  # Nacos Configuration namespace  namespace: ${SW\_CLUSTER\_NACOS\_NAMESPACE:"public"} |

2、其中 gRPCHost改成本机服务器的IP

|  |
| --- |
| core:  selector: ${SW\_CORE:default}  default:  gRPCHost: ${SW\_CORE\_GRPC\_HOST:172.16.203.77} |

3、修改存储为es

storage:  
 selector: ${SW\_STORAGE:elasticsearch}  
 elasticsearch:  
 nameSpace: ${SW\_NAMESPACE:""}  
 clusterNodes: ${SW\_STORAGE\_ES\_CLUSTER\_NODES:172.16.203.77:9200,172.16.203.78:9200,172.16.203.79:9200}

**collector集群**

分别在一下3台服务器**启动collector** ：

172.16.203.76

172.16.203.77  
172.16.203.78

cd /usr/local/skywalking/bin/  
 sh oapService.sh

**dashboard配置及启动**

在其中一台172.16.203.76配置及其启动

vim /usr/local/skywalking/webapp/webapp.yml

修改端口号server.port 与 listOfServers（根据实际情况修改）

|  |
| --- |
| server:  port: 8051  collector:  path: /graphql  ribbon:  ReadTimeout: 10000  # Point to all backend's restHost:restPort, split by ,  listOfServers: 172.16.203.76:12800,172.16.203.77:12800,172.16.203.78:12800 |

启动dashboard

|  |
| --- |
| cd /usr/local/skywalking/bin/  sh webappService.sh |

**服务端产生的日志路径**

/usr/local/skywalking/logs

**停止skywalking**

# 关闭 skywalking web  
  
 ps -ef|grep skywalking-webapp  
 kill -9 pid

注意*skywalking*默认会使用(8080, 10800, **11800**, 12800)端口，前面已经把8080 修改为8051

### 3.3 安装Skywalking agent 端

实际开发时候，每一个jar包获取应用都应该单独使用一个agent， 所以将agent这个目录拷贝到各自对应的jar包路径下。 核心部分的目录信息如下:

├── activations  
├── config  
│ └── agent.config  
├── logs  
│ └── skywalking-api.log  
├── optional-plugins  
├── plugins  
└── skywalking-agent.jar

上传**skywalking** 并解压

cd /usr/local/src  
  
tar -zxvf apache-skywalking-apm-8.1.0.tar.gz  
mv apache-skywalking-apm-bin skywalking  
mv skywalking/ /usr/local/

添加**gateway plugin** 插件

cd /usr/local/skywalking/agent/optional-plugins  
cp apm-spring-cloud-gateway-2.0.x-plugin-8.1.0.jar /usr/local/skywalking/agent/plugins/

设置好参数后，对于 Java 应用，添加核心的-javaagent进行启动

|  |
| --- |
| -javaagent:/usr/local/skywalking/agent/skywalking-agent.jar -Dskywalking.agent.service\_name=loit-gateway -Dskywalking.collector.backend\_service=172.16.203.76:11800,172.16.203.77:11800,172.16.203.78:11800 |

示例启动脚本如下：

|  |
| --- |
| nohup /usr/local/java/jdk1.8/bin/java -javaagent:/usr/local/skywalking/agent/skywalking-agent.jar -Dskywalking.agent.service\_name=loit-gateway -Dskywalking.collector.backend\_service=172.16.203.76:11800,172.16.203.77:11800,172.16.203.78:11800 -Xms512m -Xmx512m -XX:PermSize=128M -XX:MaxPermSize=256M -jar -Dlogging.config=/home/soft/loit-gateway-7001/logback-dev.xml -Dlogging.path=/home/soft/loit-gateway-7001/logs $KILL\_PROCESS\_NAME --spring.config.location=file:./application-test7001.yml & |

## 4、Skywalking 性能测试

测试包含一些几点内容

1. 测试使用skywalking前后的cpu、内存、磁盘、QPS变化情况
2. Skywalking挂掉后对微服务的影响
3. 链路监控日志增长情况
4. Skywalking重启后是否可以继续收集链路日志（客户端不重启）
5. 高并发、长时间压测微服务，skywalking是否能正常使用

### 4.1 测试环境准备

服务端IP如下

http://121.196.16.86:8051

客户端服务器IP如下：

172.16.203.77  
172.16.203.78  
172.16.203.79

三台服务器都按照《3.3 安装Skywalking agent 端》安装客户端。

三台服务器都启动gateway (其中 **deploy-gateway-sky-7001.sh** 为带skywalking脚本，deploy-gateway-7001.sh 为无skywalking启动脚本)

cd /home/soft/loit-gateway-7001/  
sh deploy-gateway-sky-7001.sh

三台启动portal(其中 **deploy-portal-sky-7005.sh** 为带skywalking脚本，deploy-portal-7005.sh 为无skywalking启动脚本)

cd /home/soft/portal-7005/  
sh deploy-portal-sky-7005.sh

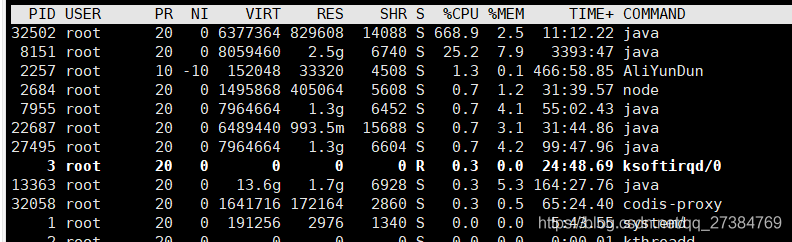
### 4.2 过程测试

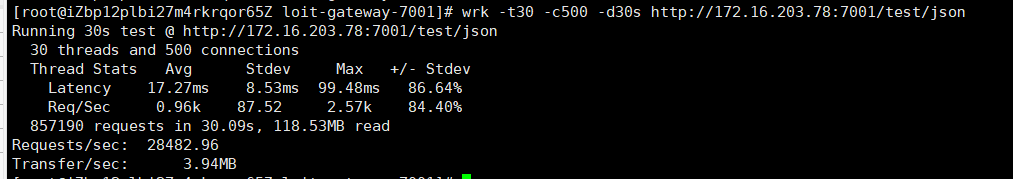
**4.2.1 性能影响测试**

* 在172.16.203.77 服务器上运行wrk进行测试
* 分别测试使用skywalking前后变化情况，发连接数分别测试**500、1000、5000**
* 测试指标比对使用skywalking前后的 **QPS、内存、cpu、磁盘空间**变化情况

wrk -t30 -c500 -d30s http://172.16.203.78/test/json

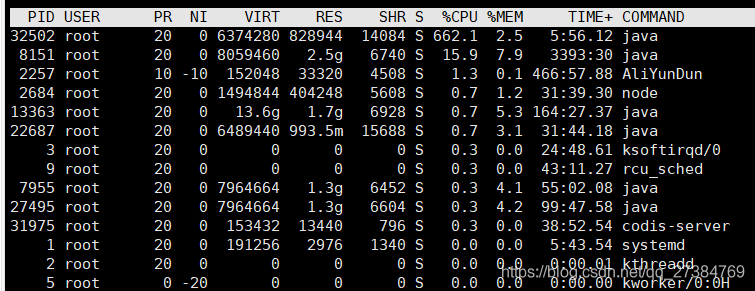
🚚 **500 并发(无链路监控)**

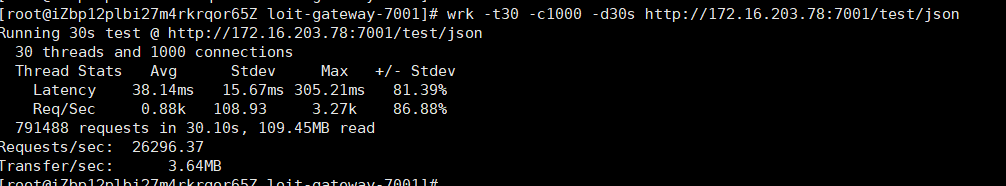




* 8核CPU使用668.9%，内存使用2.5% ，QPS 为 **28482.96**
* 平均延迟为 17.27ms

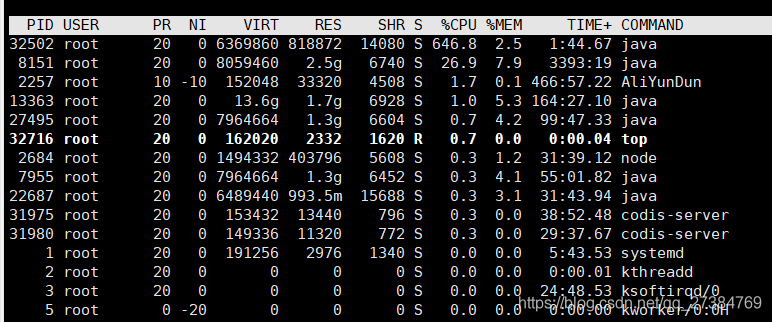
🚚 **1000 并发(无链路监控)**

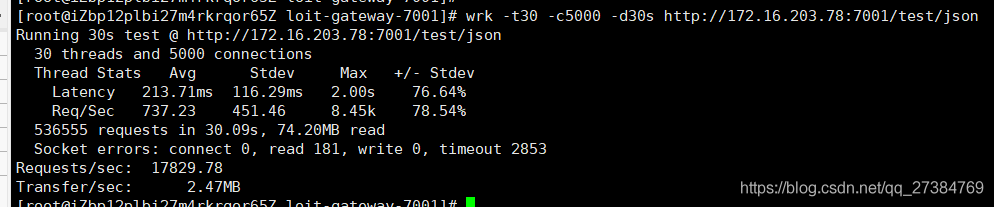




* 8核CPU使用662.1%，内存使用2.5% ，QPS 为 **26296.37**
* 平均延迟为 38.14ms

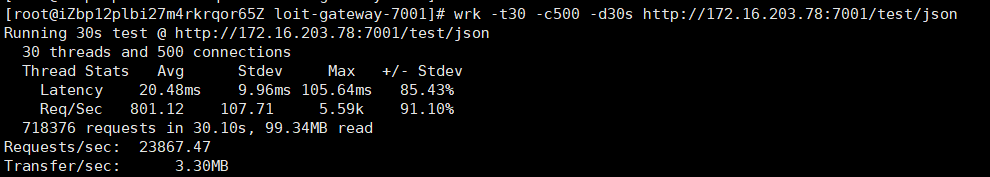
🚚 **5000 并发(无链路监控)**



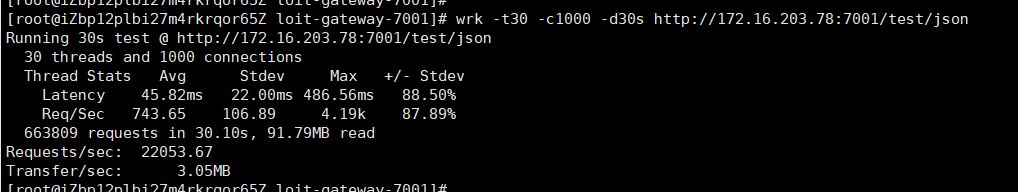


* 8核CPU使用646.1%，内存使用2.5% ，QPS 为 **17829.78**
* 平均延迟为 213.71ms

🚚 **500 并发(skywalking)**

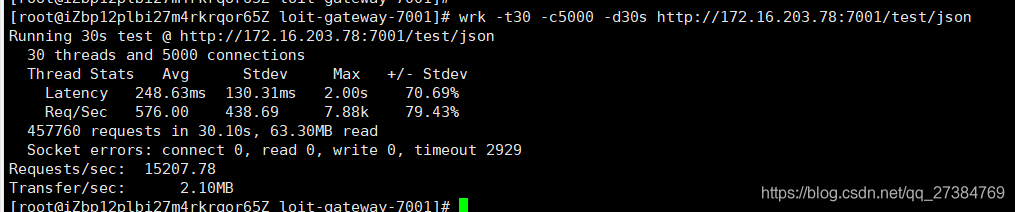


* QPS 为 **23867.47**
* 平均延迟为 20.48ms

🚚 **1000 并发(skywalking)** 

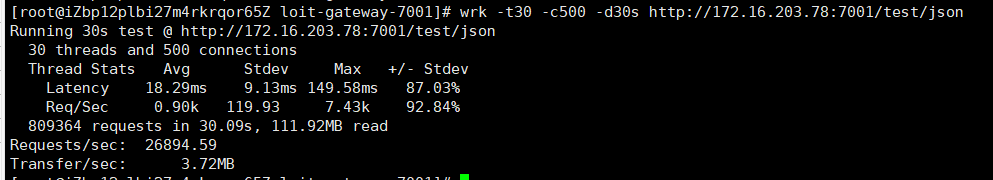
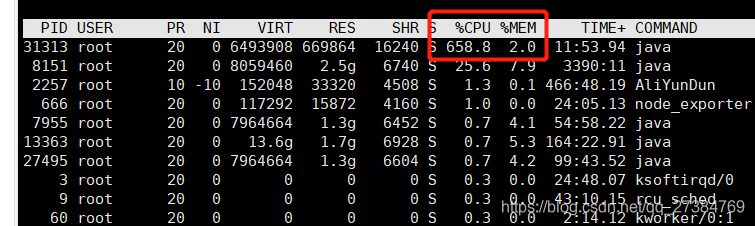
* QPS 为 **22053.67**
* 平均延迟为 45.82ms

🚚 **5000 并发(skywalking)**

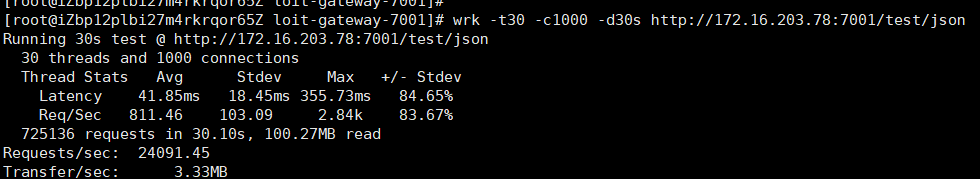


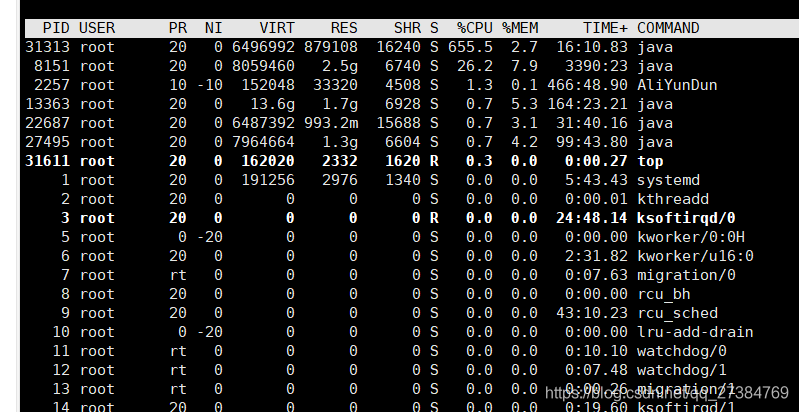
* QPS 为 **15207.78**
* 平均延迟为 248.63ms

🚚 **500 并发(停止skywalking server)**

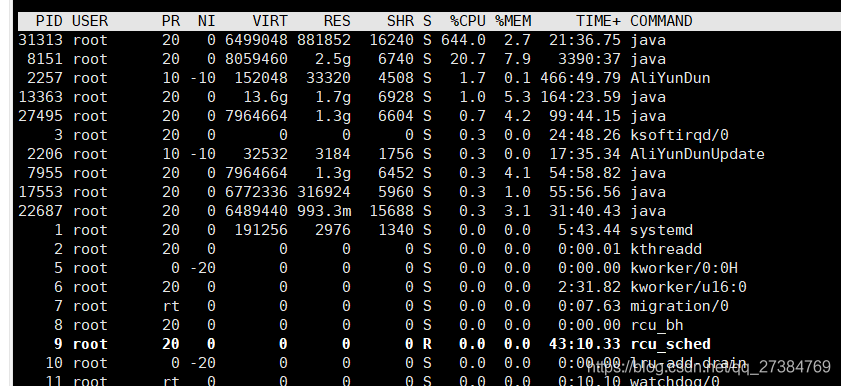
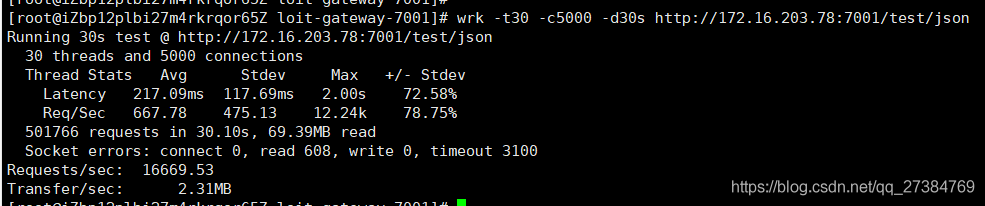
* 8核cpu使用658.8%，内存使用2.0%，QPS 为 **26894.59**
* 平均延迟为 18.29ms

🚚 **1000 并发(停止skywalking server)** 



* 8核cpu使用655.5%，内存使用2.7%，QPS 为 **24091.45**
* 平均延迟为 41.85ms

🚚 **5000 并发(停止skywalking server)**

* 8核cpu使用644%，内存使用2.7%，QPS 为 **16669.53**
* 平均延迟为 217.09ms

### 4.3 测试情况及结论

**500并发(客户端性能指标)**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 场景 | CPU（8核） | 内存（32G） | QPS | 平均延迟 |
| 链路监控（无） | 668.90% | 2.50% | 28482.96 | 17.27ms |
| 链路监控（有） |  |  | 23867.47 | 20.48ms |
| 链路监控（server 停止） | 658.80% | 2.00% | 26894.59 | 18.29ms |

**1000并发(客户端性能指标)**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 场景 | CPU（8核） | 内存（32G） | QPS | 平均延迟 |
| 链路监控（无） | 662.10% | 2.50% | 26296.37 | 38.14ms |
| 链路监控（有） |  |  | 22053.67 | 45.82ms |
| 链路监控（server 停止） | 655.50% | 2.70% | 24091.45 | 41.85ms |

**5000并发(客户端性能指标)**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 场景 | CPU（8核） | 内存（32G） | QPS | 平均延迟 |
| 链路监控（无） | 646.10% | 2.50% | 17829.78 | 213.71ms |
| 链路监控（有） |  |  | 15207.78 | 248.63ms |
| 链路监控（server 停止） | 644% | 2.70% | 16669.53 | 217.09ms |

* 1、链路压测对QPS和平均延时有影响。
* 2、skywalking 挂掉后对微服务正常使用且性能较有链路监控有所提升
* 3、skywalking客户端对cpu及内存影响不是很明显

**ES中链路日志增长情况**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 场景 | 磁盘使用情况 | 日志数 | 使用情况 | 场景 |
| 磁盘使用（未测试前） | 138G | 0 | 0 | 磁盘使用（未测试前） |
| 磁盘使用（测试中） | 153G | 1141511 | 15G | 磁盘使用（测试中） |
| 磁盘使用（测试中） | 197G | 3799685 | 59G | 磁盘使用（测试中） |

**其他测试及结论**

skywalking停掉后重新启动后是否恢复日志收集功能？**(yes)**

## 5、使用说明

### 5.1 配置覆盖

Skywalking支持的几种配置方式其中优先级如下：

探针配置 > 系统配置 >系统环境变量 > 配置文件中的值

**5.1.1 系统配置（System properties)**

通过使用-Dskywalking. 如下进行 agent.service\_name 的覆盖

-Dskywalking.agent.service\_name=skywalking\_mysql

**5.1.2 探针配置（Agent options）**

-javaagent:/path/to/skywalking-agent.jar=[option1]=[value1],[option2]=[value2]

**案例** 通过 如下进行 agent.service\_name 的覆盖

-javaagent:/path/to/skywalking-agent.jar=agent.service\_name=skywalking\_mysql

**特殊字符** 如果配置中包含分隔符( , 或者 =) , 就必须使用引号包裹起来

-javaagent:/path/to/skywalking-agent.jar=agent.ignore\_suffix='.jpg,.jpeg'

**5.1.3 系统环境变量（System environment variables）**

配置在操作系统的环境变量中 由于agent.service\_name配置项如下所示：

# The service name in UI   
agent.service\_name=${SW\_AGENT\_NAME:Your\_ApplicationName}  
12

可以在环境变量中设置SW\_AGENT\_NAME的值来指定服务名。

**覆盖优先级** 探针配置 > 系统配置 >系统环境变量 > 配置文件中的值

所以我们的启动命令可以修改为

java -javaagent:/usr/local/skywalking/apache-skywalking-apm-   
bin/agent\_mysql/skywalking-agent.jar -   
Dskywalking.agent.service\_name=skywalking\_mysql -jar skywalking\_mysql.jar &

或者

java -javaagent:/usr/local/skywalking/apache-skywalking-apm-   
bin/agent\_mysql/skywalking-agent.jar=agent.service\_name=skywalking\_mysql -jar skywalking\_mysql.jar &

### 5.2 代码中添加额外链路信息

Skywalking提供我们Trace工具包，用于在追踪链路时进行信息的打印

**pom文件**

<properties>  
 <java.version>1.8</java.version>  
 <skywalking.version>8.1.0</skywalking.version>  
 </properties>  
  
 <!--skywalking trace工具包-->  
 <dependency>  
 <groupId>org.apache.skywalking</groupId>  
 <artifactId>apm-toolkit-trace</artifactId>  
 <version>${skywalking.version}</version>  
 </dependency>

添加了skywalking trace的工具包, 该工具包版本需要与skywalking版本相同, 这里采用8.1.0

**PluginController**

import org.apache.skywalking.apm.toolkit.trace.ActiveSpan;  
import org.apache.skywalking.apm.toolkit.trace.TraceContext;  
import org.springframework.web.bind.annotation.GetMapping;  
import org.springframework.web.bind.annotation.RestController;  
  
@RestController  
public class PluginController {  
  
 //获取trace id，可以在RocketBot追踪中进行查询  
 @GetMapping("/getTraceId")  
 public String getTraceId(){  
 //使当前链路报错，并且提示报错信息  
 ActiveSpan.error(new RuntimeException("Test-Error-Throwable"));  
 //打印info信息  
 ActiveSpan.info("Test-Info-Msg");  
 //打印debug信息  
 ActiveSpan.debug("Test-debug-Msg");  
 return TraceContext.traceId();  
 }  
}

使用TraceContext.traceId()可以打印出当前追踪的ID，方便在RocketBot中进行搜索。

**ActiveSpan提供了三个方法进行信息的打印**

* error方法会将本次调用变为失败状态，同时可以打印对应的堆栈信息和错误提示。
* info方法打印info级别的信息。
* debug方法打印debug级别的信息。

### 5.3 过滤指定的端点

在开发过程中，有一些端点（接口）并不需要去进行监控，

**部署方式**

将agent中的 /agent/optional-plugins/apm-trace-ignore-plugin-6.4.0.jar插件拷贝到 plugins目录中。

cd /usr/local/skywalking/agent  
cp optional-plugins/apm-trace-ignore-plugin-8.1.0.jar plugins/apm-trace-ignore-plugin-8.1.0.jar

启动应用

/usr/local/java/jdk1.8/bin/java -javaagent:/usr/local/skywalking/agent/skywalking-agent.jar -Dskywalking.agent.service\_name=loit-gateway -Dskywalking.collector.backend\_service=172.16.203.76:11800 -   
Dskywalking.trace.ignore\_path=/exclude -Xms512m -Xmx512m -XX:PermSize=128M -XX:MaxPermSize=256M -jar -Dlogging.config=/home/soft/loit-gateway-7001/logback-dev.xml -Dlogging.path=/home/soft/loit-gateway-7001/logs $KILL\_PROCESS\_NAME --spring.config.location=file:./application-test7001.yml &

这里添加-Dskywalking.trace.ignore\_path=/exclude参数来标识需要过滤哪些请求，支持 Ant Path表达式：

* /path/\*, /path/\*\*, /path/?
  + ? 匹配任何单字符
  + \* 匹配0或者任意数量的字符
  + \*\* 匹配0或者更多的目录

### 5.4 告警功能

**5.4.1 告警功能简介**

Skywalking每隔一段时间根据收集到的链路追踪的数据和配置的告警规则（如服务响应时间、服务响应 时间百分比）等，判断如果达到阈值则发送相应的告警信息。发送告警信息是通过调用webhook接口完 成，具体的webhook接口可以使用者自行定义，从而开发者可以在指定的webhook接口中编写各种告 警方式，比如邮件、短信等。告警的信息也可以在RocketBot中查看到。

以下是默认的告警规则配置，位于skywalking安装目录下的conﬁg文件夹下 alarm-settings.yml文件 中：

rules:  
 # Rule unique name, must be ended with `\_rule`.  
 service\_resp\_time\_rule:  
 metrics-name: service\_resp\_time  
 op: ">"  
 threshold: 1000  
 period: 10  
 count: 3  
 silence-period: 5  
 message: Response time of service {name} is more than 1000ms in 3 minutes of last 10 minutes.  
   
webhooks:  
# - http://127.0.0.1/notify/  
# - http://127.0.0.1/go-wechat/

**以上文件定义了默认的4种规则**

1. 最近3分钟内服务的平均响应时间超过1秒
2. 最近2分钟服务成功率低于80%
3. 最近3分钟90%服务响应时间超过1秒
4. 最近2分钟内服务实例的平均响应时间超过1秒 规则中的参数属性如下

**属性参照表**

|  |  |
| --- | --- |
| 属性 | 含义 |
| metrics-name | oal脚本中的度量名称 |
| threshold | 阈值，与metrics-name和下面的比较符号相匹配 |
| op | 比较操作符，可以设定>,<,= |
| period | 多久检查一次当前的指标数据是否符合告警规则，单位分钟 |
| count | 达到多少次后，发送告警消息 |
| silence-period | 在多久之内，忽略相同的告警消息 |
| message | 告警消息内容 |
| include-names | 本规则告警生效的服务列表 |

webhooks可以配置告警产生时的调用地址。

**5.4.2 告警功能说明**

**WebHooks**

产生告警时会调用webhook接口，该接口必须是Post类型，同时接口参数使用RequestBody。参 数格式为：

[{  
 "scopeId": 1,  
 "scope": "SERVICE",  
 "name": "serviceA",  
 "id0": 12,  
 "id1": 0,  
 "ruleName": "service\_resp\_time\_rule",  
 "alarmMessage": "alarmMessage xxxx",  
 "startTime": 1560524171000  
}, {  
 "scopeId": 1,  
 "scope": "SERVICE",  
 "name": "serviceB",  
 "id0": 23,  
 "id1": 0,  
 "ruleName": "service\_resp\_time\_rule",  
 "alarmMessage": "alarmMessage yyy",  
 "startTime": 1560524171000  
}]

**AlarmMessage**

public class AlarmMessage {  
 private int scopeId;  
 private String name;  
 private int id0;  
 private int id1;  
 //告警的消息  
 private String alarmMessage;  
 //告警的产生时间  
 private long startTime;  
}

在生产中使用可以在webhook接口中对接短 信、邮件等平台，当告警出现时能迅速发送信息给对应的处理人员，提高故障处理的速度。

### 5.5 UI界面介绍

CPM:每分钟请求调用次数（平均吞吐量）。

SLA: 服务等级协议（简称：SLA，全称：service level agreement）。是在一定开销下为保障服务的性能和可用性，服务提供商与用户间定义的一种双方认可的协定。通常这个开销是驱动提供服务质量的主要因素。即服务可用性，如99.9,99.99,99.999

CLR：（公共语言运行库,Common Language Runtime）和 Java 虚拟机一样也是一个运行时环境，是一个可由多种编程语言使用的运行环境。CLR 的核心功能包括：内存管理、程序集加载、安全性、异常处理和线程同步，可由面向 CLR 的所有语言使用。并保证应用和底层操作系统之间必要的分离。

百分位数：skywalking中有P50，P75，P90，P95，P99这种统计口径，就是百分位数的概念。

|  |
| --- |
| 如 有50%的请求响应时间低于1020ms，有75%的请求响应时间低于1200ms，有90%的请求响应时间低于2150ms，有95%的请求响应时间低于3140ms，有99%的请求响应时间低于3220ms。 |

### 5.5 UI错误排查

skywalking-oap-server.log

错误信息一：

|  |
| --- |
| 错误信息\*\*Elasticsearch exception [type=cluster\_block\_exception, reason=blocked by: [FORBIDDEN/12/index read-only / allow delete (api)\*\* 表示ElasticSearch进入“只读”模式，节点无法更改。可能原因是存储空间不足导致。解决方式1、保证存储空间充足 2、设置对应的索引解除“只读模式”。 skywalking每天都会参数新索引，新索引是是解除只读模式的。 |