

# 0.学习目标

---

- 能够知道什么是Skywalking
- 能够搭建Skywalking环境
- 能够使用Skywalking进行rpc调用监控
- 能够使用Skywalking进行mysql调用监控
- 了解Skywalking插件
- 了解Skywalking agent和Open Tracing原理

## 1.Skywalking概述

---

在这一部分我们主要了解以下2个问题：

- 什么是APM系统
- 什么是Skywalking

### 1.1 什么是APM系统

---

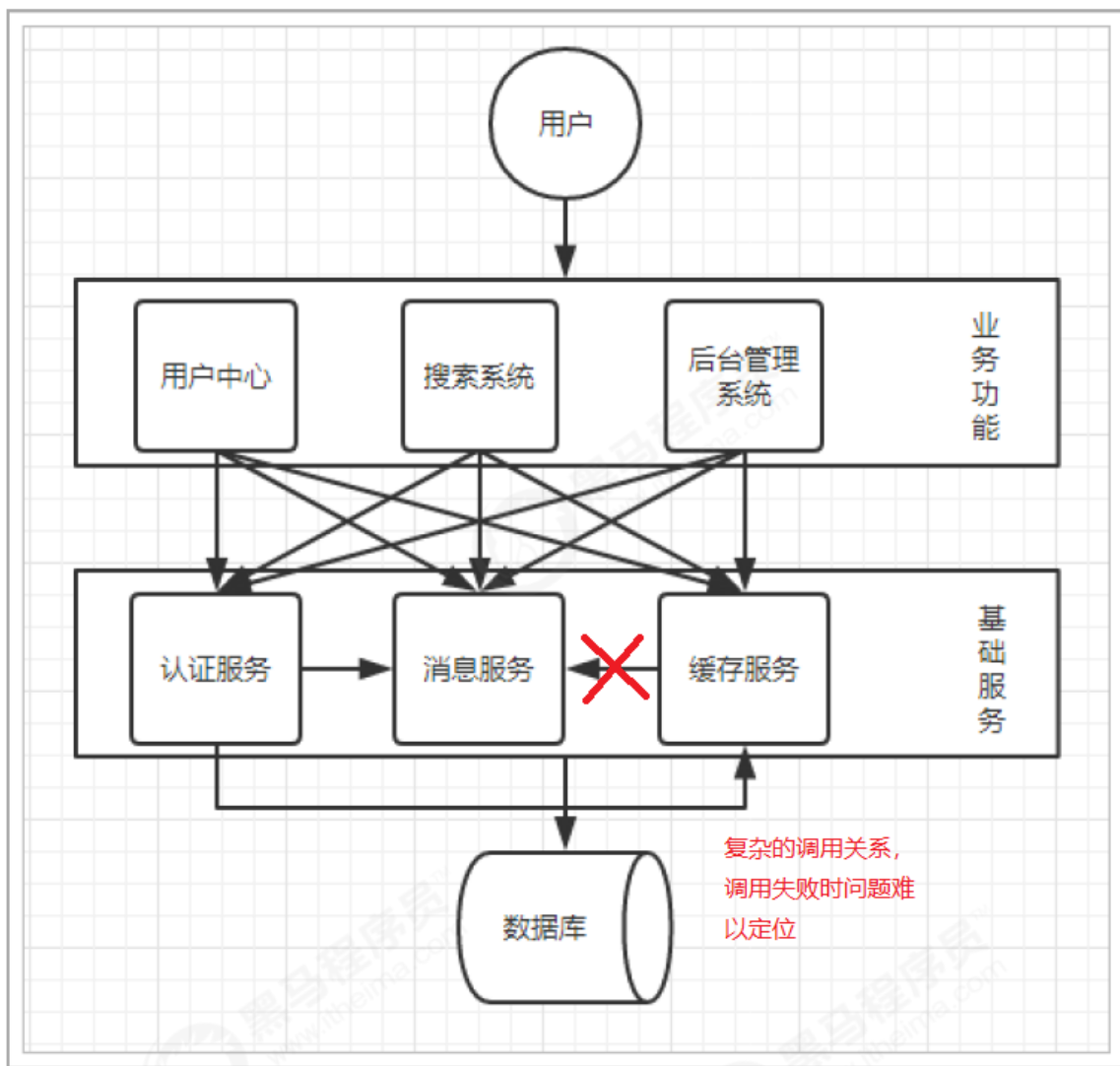
#### 1.1.1 APM系统概述

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。**

#### 1.1.2 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。



除此之外，复杂的分布式系统也面临这下面这些问题：

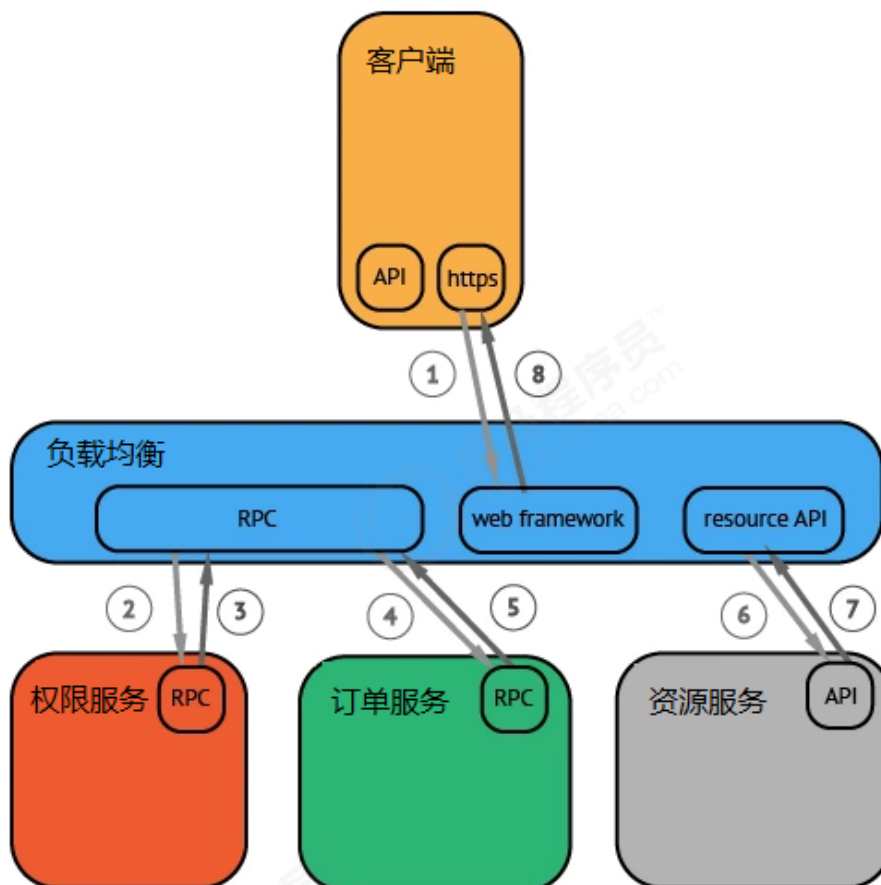
- 性能分析：一个服务依赖很多服务，被依赖的服务也依赖了其他服务。如果某个接口耗时突然变长了，那未必是直接调用的下游服务慢了，也可能是下游的下游慢了造成的，如何快速定位耗时变长的根本原因呢？
- 链路梳理：需求迭代很快，系统之间调用关系变化频繁，靠人工很难梳理清楚系统链路拓扑(系统之间的调用关系)。

为了解决这些问题，Google 推出了一个分布式链路跟踪系统 Dapper，之后各个互联网公司都参照 Dapper 的思想推出了自己的分布式链路跟踪系统，而这些系统就是分布式系统下的APM系统。

### 1.1.3 什么是OpenTracing

分布式链路跟踪最先由Google在Dapper论文中提出，而OpenTracing通过提供平台无关、厂商无关的API，使得开发人员能够方便的添加（或更换）追踪系统的实现。

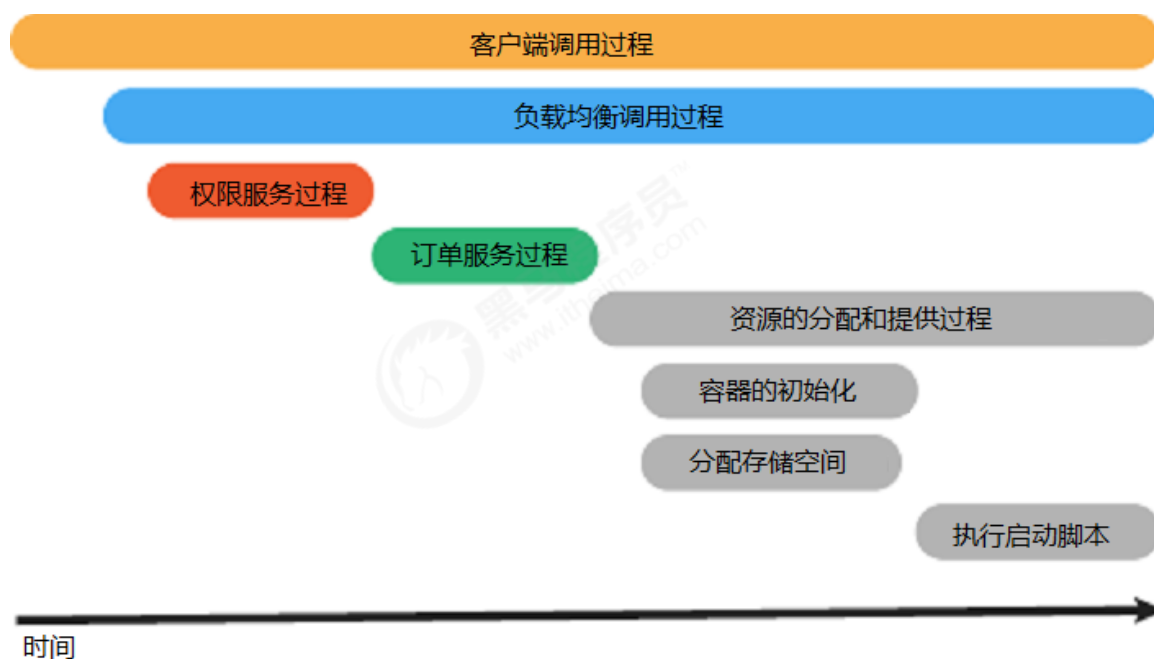
下图是一个分布式调用的例子，客户端发起请求，请求首先到达负载均衡器，接着经过认证服务，订单服务，然后请求资源，最后返回结果。



虽然这种图对于看清各组件的组合关系是很有用的，但是存在下面两个问题：

- 它不能很好显示组件的调用时间，是串行调用还是并行调用，如果展现更复杂的调用关系，会更加复杂，甚至无法画出这样的图。
- 这种图也无法显示调用间的时间间隔以及是否通过定时调用来启动调用。

一种更有效的展现一个调用过程的图：



基于OpenTracing我们就可以很轻松的构建出上面这幅图。

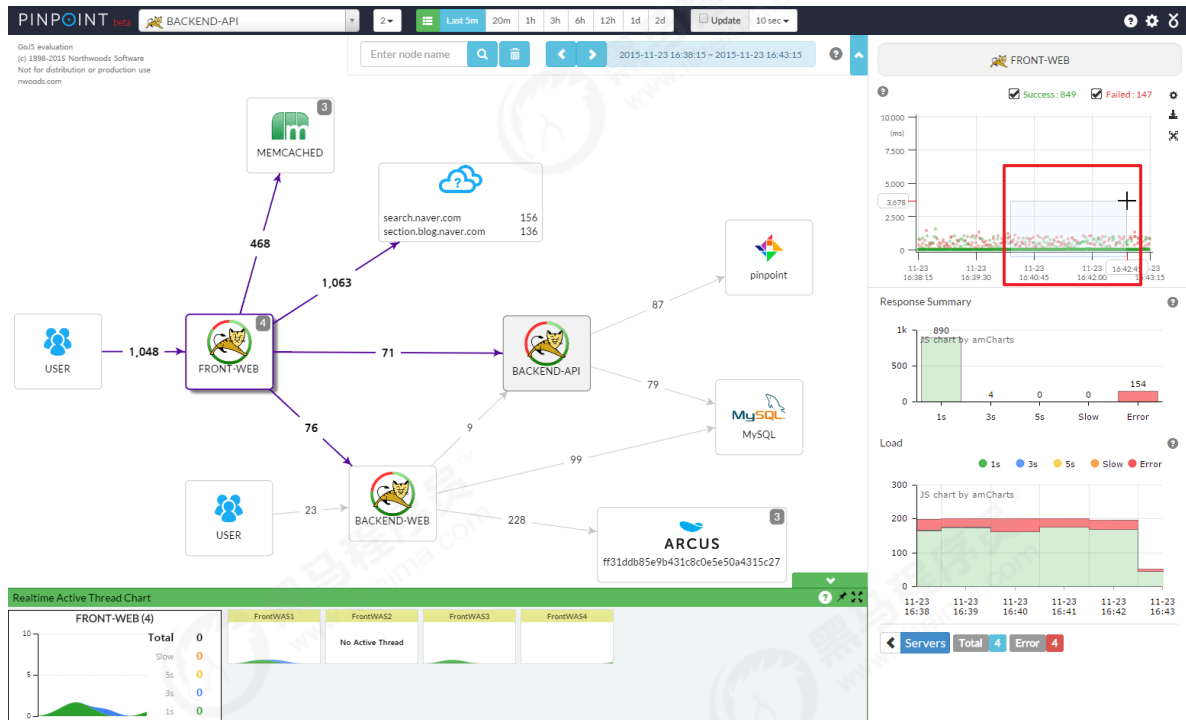
## 1.1.4 主流的开源APM产品

### 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。

官方网站：

<https://github.com/naver/pinpoint>



### 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主流框架和容器，如国产RPC Dubbo和motan等，国际化的spring boot，spring cloud。

官方网站：

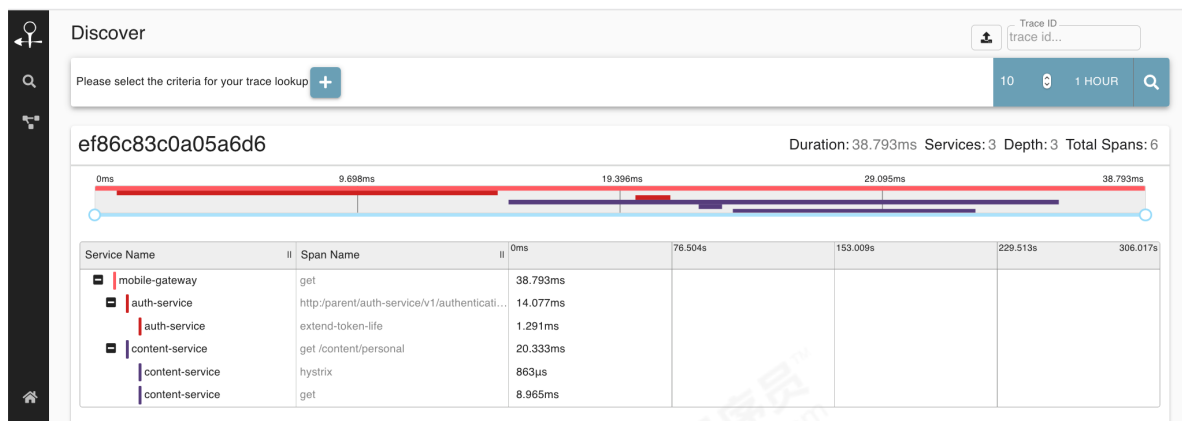
<http://skywalking.apache.org/>

### 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。

官方网站：

<https://zipkin.io/>



## 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。

官方网站:

<https://github.com/dianping/cat>

## 1.2 什么是Skywalking

### 1.2.1 Skywalking概述

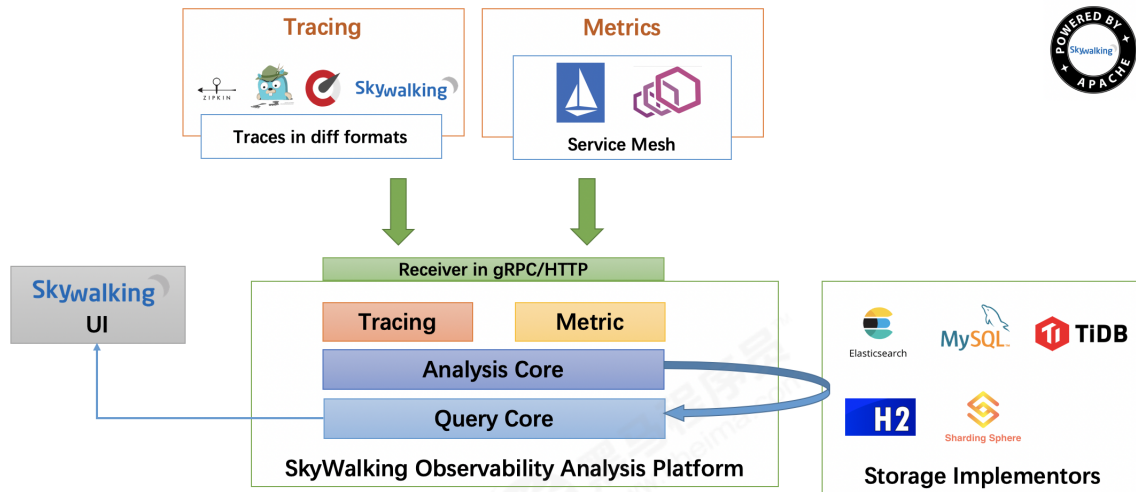
根据官方的解释，Skywalking是一个可观测性分析平台（Observability Analysis Platform简称OAP）和应用性能管理系统（Application Performance Management简称APM）。

提供分布式链路追踪、服务网格(Service Mesh)遥测分析、度量(Metric)聚合和可视化一体化解决方案。

下面是Skywalking的几大特点：

- 多语言自动探针，Java，.NET Core和NodeJS。
- 多种监控手段，语言探针和service mesh。
- 轻量高效。不需要额外搭建大数据平台。
- 模块化架构。UI、存储、集群管理多种机制可选。
- 支持告警。
- 优秀的可视化效果。

Skywalking整体架构如下：



Skywalking提供Tracing和Metrics数据的获取和聚合。

Metric的特点是，它是可累加的：他们具有原子性，每个都是一个逻辑计量单元，或者一个时间段内的柱状图。例如：队列的当前深度可以被定义为一个计量单元，在写入或读取时被更新统计；输入HTTP请求的数量可以被定义为一个计数器，用于简单累加；请求的执行时间可以被定义为一个柱状图，在指定时间片上更新和统计汇总。

Tracing的最大特点就是，它在单次请求的范围内，处理信息。任何的数据、元数据信息都被绑定到系统中的单个事务上。例如：一次调用远程服务的RPC执行过程；一次实际的SQL查询语句；一次HTTP请求的业务性ID。

总结，Metric主要用来进行数据的统计，比如HTTP请求数的计算。Tracing主要包含了某一次请求的链路数据。

详细的内容可以查看Skywalking开发者吴晟翻译的文章，Metrics, tracing 和 logging 的关系：

<http://blog.oneapm.com/apm-tech/811.html>

整体架构包含如下三个组成部分：

1. 探针(agent)负责进行数据的收集，包含了Tracing和Metrics的数据，agent会被安装到服务所在的服务器上，以方便数据的获取。
2. 可观测性分析平台OAP(Observability Analysis Platform)，接收探针发送的数据，并在内存中使用分析引擎（Analysis Core)进行数据的整合运算，然后将数据存储到对应的存储介质上，比如Elasticsearch、MySQL数据库、H2数据库等。同时OAP还使用查询引擎(Query Core)提供HTTP查询接口。
3. Skywalking提供单独的UI进行数据的查看，此时UI会调用OAP提供的接口，获取对应的数据然后进行展示。

## 1.2.2 Skywalking优势

Skywalking相比较其他的分布式链路监控工具，具有以下特点：

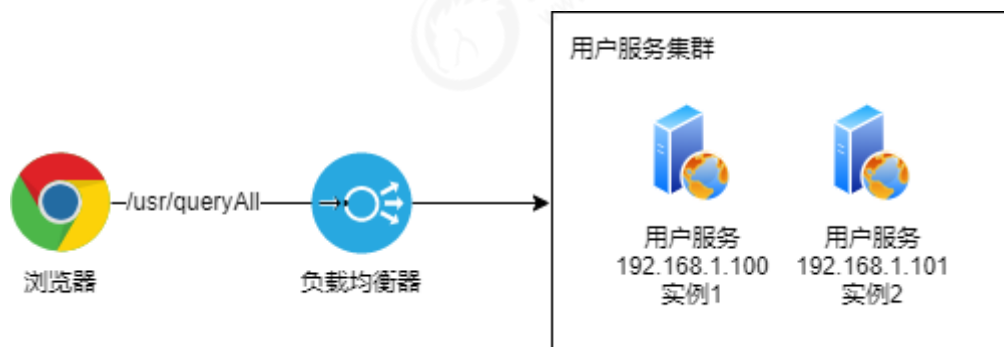
- 社区相当活跃。Skywalking已经进入apache孵化，目前的start数已经超过11K，最新版本6.5.0已经发布。开发者是国人，可以直接和项目发起人交流进行问题的解决。
- Skywalking支持Java，.NET Core和Node.JS语言。相对于其他平台：比如Pinpoint支持Java和PHP,具有较大的优势。
- 探针无侵入性。对比CAT具有侵入性的探针，优势较大。不修改原有项目一行代码就可以进行集成。

- 探针性能优秀。有网友对Pinpoint和Skywalking进行过测试，由于Pinpoint收集的数据过多，所以对性能损耗较大，而Skywalking探针性能十分出色。
- 支持组件较多。特别是对Rpc框架的支持，这是其他框架所不具备的。Skywalking对Dubbo、gRpc等有原生的支持，甚至连小众的motan和sofarpcc都支持。

### 1.2.3 Skywalking主要概念介绍

使用如下案例来进行Skywalking主要概念的介绍，Skywalking主要概念包含：

- 服务(Service)
- 端点(Endpoint)
- 实例(Instance)



上图中，我们编写了用户服务，这是一个web项目，在生产中部署了两个节点：192.168.1.100和192.168.1.101。

- 用户服务就是Skywalking的服务(Service)，用户服务其实就是一个独立的应用(Application)，在6.0之后的Skywalking将应用更名为服务(Service)。
- 用户服务对外提供的HTTP接口/usr/queryAll就是一个端点，端点就是对外提供的接口。
- 192.168.1.100和192.168.1.101这两个相同服务部署的节点就是实例，实例指同一服务可以部署多个。

## 1.3 环境搭建

接下来我们在虚拟机CentOS中搭建Skywalking的可观测性分析平台OAP环境。Skywalking默认使用H2内存中进行数据的存储，我们可以替换存储源为ElasticSearch保证其查询的高效及可用性。

具体的安装步骤可以在Skywalking的官方github上找到:

<https://github.com/apache/skywalking/blob/master/docs/en/setup/README.md>

#### 1、创建目录

```
mkdir /usr/local/skywalking
```

建议将虚拟机内存设置为3G并且将CPU设置成2核，防止资源不足。

#### 2、将资源目录中的elasticsearch和skywalking安装包上传到虚拟机/usr/local/skywalking目录下。

elasticsearch-6.4.0.tar.gz ---elasticsearch 6.4的安装包，Skywalking对es版本号有一定要求，最好使用6.3.2以上版本，如果是7.x版本需要额外进行配置。

apache-skywalking-apm-6.5.0.tar.gz ---Skywalking最新的安装包



### 3、首先安装elasticsearch，将压缩包解压。

```
tar -zxvf ./elasticsearch-6.4.0.tar.gz
```

修改Linux系统的限制配置，将文件创建数修改为65536个。

1. 修改系统中允许应用最多创建多少文件等的限制权限。Linux默认来说，一般限制应用最多创建的文件是65535个。但是ES至少需要65536的文件创建数的权限。
2. 修改系统中允许用户启动的进程开启多少个线程。默认的Linux限制root用户开启的进程可以开启任意数量的线程，其他用户开启的进程可以开启1024个线程。必须修改限制数为4096+。因为ES至少需要4096的线程池预备。

```
vi /etc/security/limits.conf
```

#新增如下内容在limits.conf文件中

```
es soft nofile 65536
es hard nofile 65536
es soft nproc 4096
es hard nproc 4096
```

修改系统控制权限，ElasticSearch需要开辟一个65536字节以上空间的虚拟内存。Linux默认不允许任何用户和应用程序直接开辟这么大的虚拟内存。

```
vi /etc/sysctl.conf
```

#新增如下内容在sysctl.conf文件中，当前用户拥有的内存权限大小

```
vm.max_map_count=262144
```

#让系统控制权限配置生效

```
sysctl -p
```

建一个用户，用于ElasticSearch启动。

ES在5.x版本之后，强制要求在linux中不能使用root用户启动ES进程。所以必须使用其他用户启动ES进程才可以。

#创建用户

```
useradd es
```

#修改上述用户的密码

```
passwd es
```

#修改elasticsearch目录的拥有者

```
chown -R es elasticsearch-6.4.0
```

使用es用户启动elasticsearch

#切换用户

```
su es
```

#到ElasticSearch的bin目录下

```
cd bin/
```

#后台启动

```
./elasticsearch -d
```

默认ElasticSearch是不支持跨域访问的，所以在不修改配置文件的情况下我们只能从虚拟机内部进行访问测试ElasticSearch是否安装成功，使用curl命令访问9200端口：



```
curl http://localhost:9200
```

如果显示出如下信息，就证明ElasticSearch安装成功：

```
{
  "name" : "xbrunxf",
  "cluster_name" : "elasticsearch",
  "cluster_uuid" : "JJQfHN9QQVuxpH5fu9H1jg",
  "version" : {
    "number" : "6.4.0",
    "build_flavor" : "default",
    "build_type" : "tar",
    "build_hash" : "595516e",
    "build_date" : "2018-08-17T23:18:47.308994Z",
    "build_snapshot" : false,
    "lucene_version" : "7.4.0",
    "minimum_wire_compatibility_version" : "5.6.0",
    "minimum_index_compatibility_version" : "5.0.0"
  },
  "tagline" : "You Know, for Search"
}
```

4、安装Skywalking，分为两个步骤：

- 安装Backend后端服务
- 安装UI

首先切回到root用户,切换到目录下，解压Skywalking压缩包。

```
#切换到root用户
su root
#切换到skywalking目录
cd /usr/local/skywalking
#解压压缩包
tar -zxvf apache-skywalking-apm-6.4.0.tar.gz
```

修改Skywalking存储的数据源配置：

```
cd apache-skywalking-apm-bin
vi config/application.yml
```

我们可以看到默认配置中，使用了H2作为数据源。我们将其全部注释。

```
# h2:
#   driver: ${SW_STORAGE_H2_DRIVER:org.h2.jdbcx.JdbcDataSource}
#   url: ${SW_STORAGE_H2_URL:jdbc:h2:mem:skywalking-oap-db}
#   user: ${SW_STORAGE_H2_USER:sa}
#   metadataQueryMaxSize: ${SW_STORAGE_H2_QUERY_MAX_SIZE:5000}
# mysql:
#   metadataQueryMaxSize: ${SW_STORAGE_H2_QUERY_MAX_SIZE:5000}
```

将ElasticSearch对应的配置取消注释：

```
storage:
```

```

elasticsearch:
  namespace: ${SW_NAMESPACE:""}
  clusterNodes: ${SW_STORAGE_ES_CLUSTER_NODES:localhost:9200}
  protocol: ${SW_STORAGE_ES_HTTP_PROTOCOL:"http"}
  trustStorePath: ${SW_SW_STORAGE_ES_SSL_JKS_PATH:"../es_keystore.jks"}
  trustStorePass: ${SW_SW_STORAGE_ES_SSL_JKS_PASS:""}
  user: ${SW_ES_USER:""}
  password: ${SW_ES_PASSWORD:""}
  indexShardsNumber: ${SW_STORAGE_ES_INDEX_SHARDS_NUMBER:2}
  indexReplicasNumber: ${SW_STORAGE_ES_INDEX_REPLICAS_NUMBER:0}
  # Those data TTL settings will override the same settings in core module.
  recordDataTTL: ${SW_STORAGE_ES_RECORD_DATA_TTL:7} # Unit is day
  otherMetricsDataTTL: ${SW_STORAGE_ES_OTHER_METRIC_DATA_TTL:45} # Unit is day
  monthMetricsDataTTL: ${SW_STORAGE_ES_MONTH_METRIC_DATA_TTL:18} # Unit is
month
# # Batch process setting, refer to
https://www.elastic.co/guide/en/elasticsearch/client/java-api/5.5/java-docs-
bulk-processor.html
  bulkActions: ${SW_STORAGE_ES_BULK_ACTIONS:1000} # Execute the bulk every
1000 requests
  flushInterval: ${SW_STORAGE_ES_FLUSH_INTERVAL:10} # flush the bulk every 10
seconds whatever the number of requests
  concurrentRequests: ${SW_STORAGE_ES_CONCURRENT_REQUESTS:2} # the number of
concurrent requests
  metadataQueryMaxSize: ${SW_STORAGE_ES_QUERY_MAX_SIZE:5000}
  segmentQueryMaxSize: ${SW_STORAGE_ES_QUERY_SEGMENT_SIZE:200}

```

默认使用了localhost下的ES,所以我们可以不做任何处理,直接进行使用。启动OAP程序:

```
bin/oapService.sh
```

这样安装Backend后端服务就已经完毕了,接下来我们安装UI。先来看一下UI的配置文件:

```
cat webapp/webapp.yml
```

```

#默认启动端口
server:
  port: 8080

collector:
  path: /graphql
  ribbon:
    ReadTimeout: 10000
    #OAP服务,如果有多个用逗号隔开
    listOfServers: 127.0.0.1:12800

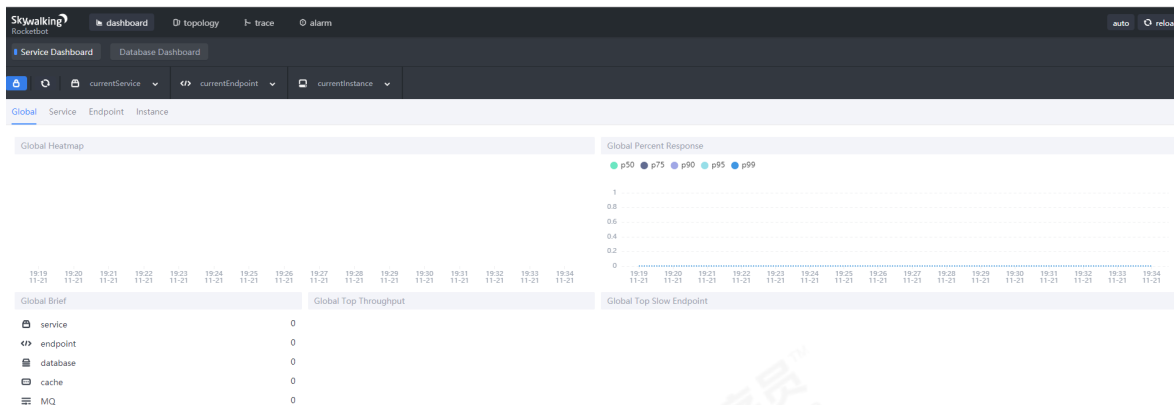
```

目前的默认配置不用修改就可以使用,启动UI程序:

```
/bin/webappService.sh
```

然后我们就可以通过浏览器访问Skywalking的可视化页面了,访问地址:<http://虚拟机IP地址:8080>,如果出现下面的图,就代表安装成功了。

/bin/startup.sh可以同时启动backend和ui,后续可以执行该文件进行重启。



## 2.Skywalking基础

### 2.1 agent的使用

agent探针可以让我们不修改代码的情况下，对java应用上使用到的组件进行动态监控，获取运行数据发送到OAP上进行统计和存储。agent探针在java中是使用java agent技术实现的，不需要更改任何代码，java agent会通过虚拟机(VM)接口来在运行期更改代码。

Agent探针支持 JDK 1.6 - 12的版本，Agent探针所有的文件在Skywalking的agent文件夹下。文件目录如下；

```
+-- agent
  +-- activations
    apm-toolkit-log4j-1.x-activation.jar
    apm-toolkit-log4j-2.x-activation.jar
    apm-toolkit-logback-1.x-activation.jar
    ...
  //配置文件
  +-- config
    agent.config
  //组件的所有插件
  +-- plugins
    apm-dubbo-plugin.jar
    apm-feign-default-http-9.x.jar
    apm-httpclient-4.x-plugin.jar
    ....
  //可选插件
  +-- optional-plugins
    apm-gson-2.x-plugin.jar
    ....
  +-- bootstrap-plugins
    jdk-http-plugin.jar
    ....
  +-- logs
    skywalking-agent.jar
```

部分插件在使用上会影响整体的性能或者由于版权问题放置于可选插件包中，不会直接加载，如果需要，将可选插件中的jar包拷贝到plugins包下。

由于没有修改agent探针中的应用名，所以默认显示的是Your\_ApplicationName。我们修改下应用名称，让他显示的更加正确。编辑agent配置文件：

```
cd /usr/local/skywalking/apache-skywalking-apm-bin/agent/config
vi agent.config
```

我们在配置中找到这么一行：

```
# The service name in UI
agent.service_name=${SW_AGENT_NAME:Your_ApplicationName}
```

这里的配置含义是可以读取到SW\_AGENT\_NAME配置属性，如果该配置没有指定，那么默认名称为Your\_ApplicationName。这里我们把Your\_ApplicationName替换成skywalking\_tomcat。

```
# The service name in UI
agent.service_name=${SW_AGENT_NAME:skywalking_tomcat}
```

然后将tomcat重启:

```
./shutdown.sh
./startup.sh
```

## 2.1.1 Linux 下Tomcat7和8中使用

1.要使用Skywalking监控Tomcat中的应用，需要先准备一个Spring Mvc项目，在资源中已经提供了打包好的文件

skywalking\_springmvc-1.0-SNAPSHOT.war。

以下是该项目的接口代码：

```
package com.itcast.controller;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;

@Controller
@RequestMapping("/hello")
public class HelloController {
    @RequestMapping("/sayHello")
    @ResponseBody
    public String sayHello(String name){
        return "hello world";
    }
}
```

将资源文件下的 apache-tomcat-8.5.47.tar.gz 文件上传至虚拟机/usr/local/skywalking目录下，然后解压：

```
tar -zxvf apache-tomcat-8.5.47.tar.gz
```

将war包上传至/usr/local/skywalking/apache-tomcat-8.5.47/webapps/下。编辑 /usr/local/skywalking/apache-tomcat-8.5.47/bin/catalina.sh 文件，在文件顶部添加：

```
CATALINA_OPTS="$CATALINA_OPTS -javaagent:/usr/local/skywalking/apache-skywalking-apm-bin/agent/skywalking-agent.jar"; export CATALINA_OPTS
```

修改tomcat启动端口：

```
vi conf/server.xml

#修改这一行的端口为8081
<Connector port="8080" protocol=...
```

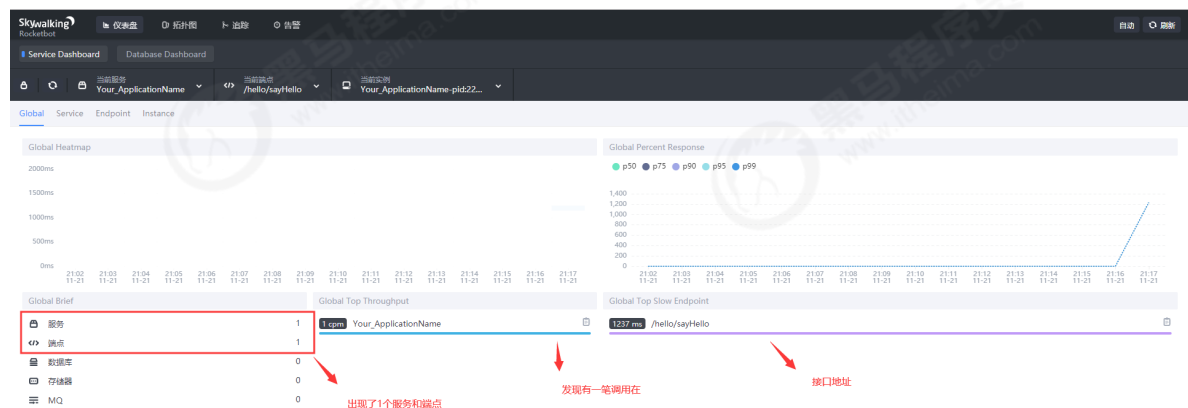
执行bin目录下的./startup.sh 文件启动tomcat。然后访问地址：

[http://虚拟机IP:8081/skywalking\\_springmvc-1.0-SNAPSHOT/hello/sayHello.do](http://虚拟机IP:8081/skywalking_springmvc-1.0-SNAPSHOT/hello/sayHello.do)

如果正确打开，能输出hello world。

注意：一定要保证虚拟机和物理机的时间一致，否则访问数据的时候会因为时间不一致而获取不到数据。

此时再访问Skywalking的页面，会发现出现了一个服务和端点，同时有一笔调用显示了调用的应用名和接口地址。



由于没有修改agent探针中的应用名，所以默认显示的是Your\_ApplicationName。接下来我们修改下应用名称，让他显示的更加正确。编辑agent配置文件：

```
cd /usr/local/skywalking/apache-skywalking-apm-bin/agent/config
vi agent.config
```

我们在配置中找到这么一行：

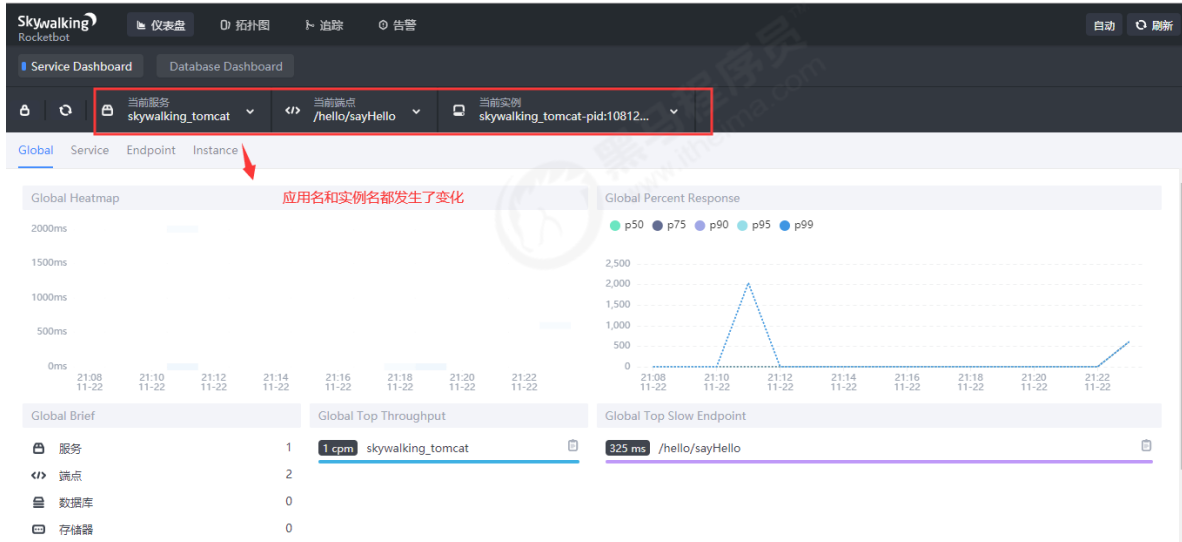
```
# The service name in UI
agent.service_name=${SW_AGENT_NAME:Your_ApplicationName}
```

这里的配置含义是可以读取到SW\_AGENT\_NAME配置属性，如果该配置没有指定，那么默认名称为Your\_ApplicationName。这里我们把Your\_ApplicationName替换成skywalking\_tomcat。

```
# The service name in UI
agent.service_name=${SW_AGENT_NAME:skywalking_tomcat}
```

然后将tomcat重启:

```
./shutdown.sh
./startup.sh
```



## 2.1.2 Windows 下Tomcat7和8中使用(了解)

Windows下只需要修改f tomcat目录/bin/catalina.bat 文件的第一行为：

```
set "CATALINA_OPTS=-javaagent:/path/to/skywalking-agent/skywalking-agent.jar"
```

## 2.1.3 Spring Boot中使用

Skywalking与Spring Boot集成提供了完善的支持。

1、首先我们复制一份agent，防止与tomcat使用的冲突。

```
cd /usr/local/skywalking/apache-skywalking-apm-bin/
cp -r agent agent_boot
vi agent_boot/config/agent.config
```

修改配置中的应用名为：

```
# The service name in UI
agent.service_name=${SW_AGENT_NAME:skywalking_boot}
```

2、将资源文件夹中 skywalking\_springboot.jar 文件上传到 /usr/local/skywalking 目录下。

Controller层代码如下，提供了一个正常访问的接口和一个异常访问接口：

```
package com.itcast.skywalking_springboot.controller;
```

```
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class MyController {

    //正常访问接口
    @RequestMapping("/sayBoot")
    public String sayBoot(){
        return "Hello Boot!";
    }

    //异常访问接口
    @RequestMapping("/exception")
    public String exception(){
        int i = 1/0;
        return "Hello Boot!";
    }
}
```

使用命令启动spring boot项目:

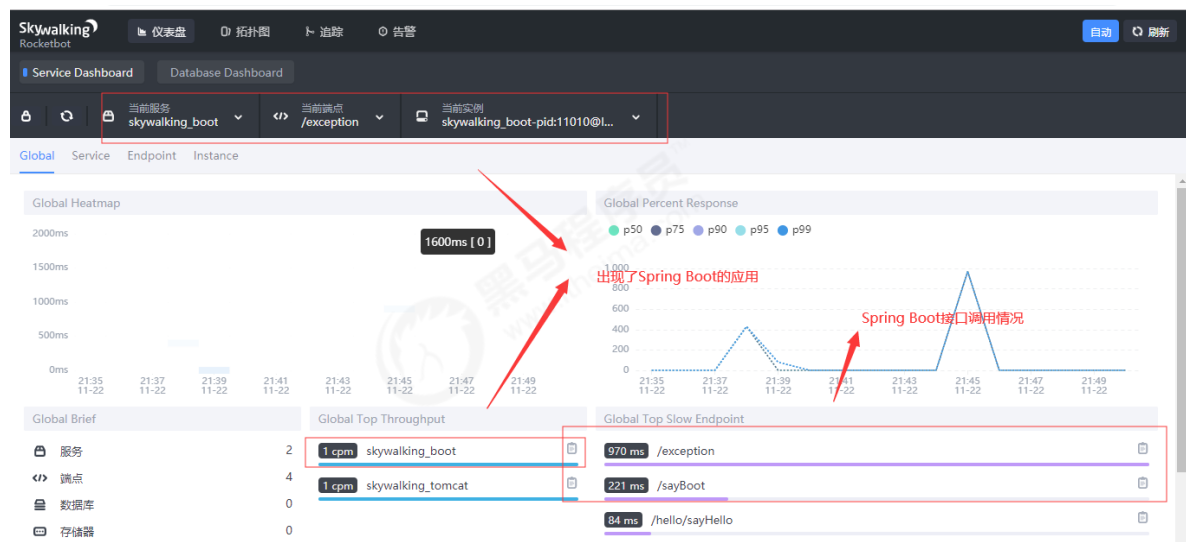
```
java -javaagent:/usr/local/skywalking/apache-skywalking-apm-
bin/agent_boot/skywalking-agent.jar -Dserver.port=8082 -jar
skywalking_springboot.jar &
```

使用jar包启动的项目如果需要集成skywalking，需要添加-javaagent参数，参数值为agent的jar包梭子啊位置。

-Dserver.port参数用于指定端口号，防止与tomcat冲突。

末尾添加 & 后台运行模式启动Spring Boot项目。

此时我们可以访问<http://虚拟机.IP:8082/sayBoot>地址来进行访问，访问之后稍等片刻访问Skywalking的UI页面。



## 2.2 RocketBot的使用



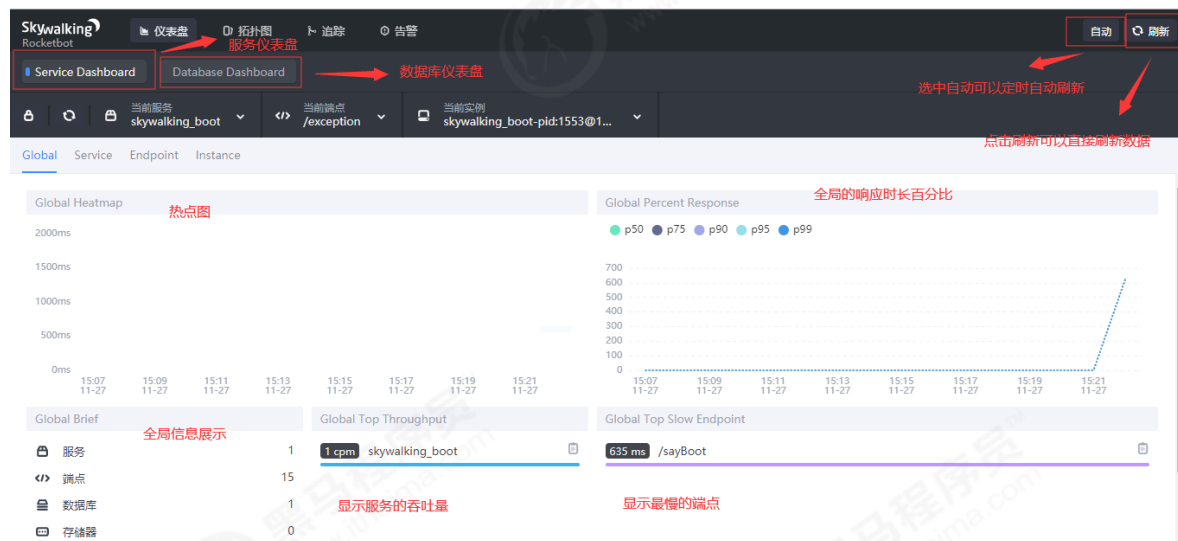
Skywalking的监控UI页面成为RocketBot，我们可以通过 8080 端口进行访问，由于8080端口很容易冲突，可以修改 webapp/webapp.yml 来更改启动端口：

```
server:
  port: 8080
```

本例中我们更改为9010端口防止冲突。访问<http://虚拟机IP:9010/>打开RocketBot的页面。

## 2.2.1 仪表盘

打开RocketBot默认会出现仪表盘页面：



仪表盘页面分为两大块：

- 服务仪表盘，展示服务的调用情况
- 数据库仪表盘，展示数据库的响应时间等数据

选中服务仪表盘，有四个维度的统计数据可以进行查看：

- 全局，查看全局接口的调用，包括全局响应时长的百分比，最慢的端点，服务的吞吐量等
- 服务，显示服务的响应时长、SLA、吞吐量等信息
- 端点，显示端点的响应时长、SLA、吞吐量等信息
- 实例，显示实例的响应时长、SLA、吞吐量等信息，还可以查看实例的JVM的GC信息、CPU信息、内存信息

## 2.2.2 拓扑图

Skywalking提供拓扑图，直观的查看服务之间的调用关系：



User代表用户应用，目前案例中其实是浏览器

图中Skywalking\_boot应用被User调用，同时显示它是一个Spring MVC的应用。后续案例中会出现多个应用调用，使用拓扑图就能清楚的分析其调用关系了。

## 2.2.3 追踪

在Skywalking中，每一次用户发起一条请求，就可以视为一条追踪数据，每条追踪数据携带有一个ID值。追踪数据在追踪页面中可以进行查询：



左侧是追踪列表，也可以通过上方的追踪ID来进行查询。点击追踪列表某一条记录之后，右侧会显示出此条追踪的详细信息。有三种显示效果：

- 列表
- 树结构
- 表格

可以很好的展现此条追踪的调用链情况而链路上每个节点，可以通过左键点击节点查看详细信息：

## 跨度信息

### 标记.

端点:	/sayBoot
跨度类型:	Entry
组件:	SpringMVC
Peer:	No Peer
失败:	false
url:	http://192.168.62.139:8082/sayBoot
http.method:	GET

当前的接口是HTTP的GET请求，相对比较简单，后续的示例中出现异常情况或者数据库访问，可以打印出异常信息、堆栈甚至详细的SQL语句。

### 2.2.4 告警

Skywalking中的告警功能相对比较简单，在达到告警阈值之后会生成一条告警记录，在告警页面上进行展示。后面会有独立的章节介绍告警功能如何使用，本节中暂不详细介绍。

## 3.Skywalking高级

### 3.1 Rpc调用监控

skywalking(6.5.0) 支持的Rpc框架有以下几种：

- [Dubbo](#) 2.5.4 -> 2.6.0
- [Dubbox](#) 2.8.4
- [Apache Dubbo](#) 2.7.0
- [Motan](#) 0.2.x -> 1.1.0
- [gRPC](#) 1.x
- [Apache ServiceComb Java Chassis](#) 0.1 -> 0.5, 1.0.x
- [SOFARPC](#) 5.4.0

本节中我们使用Spring Boot和Dubbo搭建一个简单的服务提供方和服务消费方来测试Skywalking对于Rpc调用的支持。可以使用资源文件夹下已经完成打包的 `skywalking_dubbo_consumer.jar` 和 `skywalking_dubbo_provider.jar` 来进行测试。

#### 3.1.1 服务提供方

pom文件：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```

        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.1.10.RELEASE</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>
    <groupId>com.itcast</groupId>
    <artifactId>skywalking_dubbo_provider</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>skywalking_dubbo_provider</name>
    <description>Demo project for Spring Boot</description>

    <properties>
        <java.version>1.8</java.version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>

        <!--添加springboot和dubbo集成配置-->
        <dependency>
            <groupId>com.alibaba.spring.boot</groupId>
            <artifactId>dubbo-spring-boot-starter</artifactId>
            <version>2.0.0</version>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>
</project>

```

这里直接使用了 `dubbo-spring-boot-starter` 这一dubbo与spring-boot集成的组件。

官方文档地址：[https://github.com/alibaba/dubbo-spring-boot-starter/blob/master/README\\_zh.md](https://github.com/alibaba/dubbo-spring-boot-starter/blob/master/README_zh.md)

application.properties:

```
spring.application.name=skywalking_dubbo_provider
spring.dubbo.server=true
spring.dubbo.registry=N/A
server.port=8086
```

为了简化环境搭建，采用了本地直接调用的方式，所以将注册中心写成N/A表示不注册到注册中心。

IHelloService接口：

```
package com.itcast.api;

public interface IHelloService {
    public String hello();
}
```

简化项目的开发，将IHelloService接口在消费方和提供方都编写一份。

HelloServiceImpl实现类：

```
package com.itcast.skywalking_dubbo_provider.service;

import com.alibaba.dubbo.config.annotation.Service;
import com.itcast.api.IHelloService;
import org.springframework.stereotype.Component;

@Service(interfaceClass = IHelloService.class)
@Component
public class HelloServiceImpl implements IHelloService {
    @Override
    public String hello() {
        return "hello skywalking";
    }
}
```

SkywalkingDubboProviderApplication启动类：

```
package com.itcast.skywalking_dubbo_provider;

import com.alibaba.dubbo.spring.boot.annotation.EnableDubboConfiguration;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
//添加dubbo生效注解
@EnableDubboConfiguration
public class SkywalkingDubboProviderApplication {

    public static void main(String[] args) {
        SpringApplication.run(SkywalkingDubboProviderApplication.class, args);
    }
}
```

```
}
```

需要添加@EnableDubboConfiguration注解。

### 3.1.2 服务消费方

pom文件：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.1.10.RELEASE</version>
    <relativePath/> <!-- lookup parent from repository -->
  </parent>
  <groupId>com.itcast</groupId>
  <artifactId>skywalking_dubbo_consumer</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>skywalking_dubbo_consumer</name>
  <description>Demo project for Spring Boot</description>

  <properties>
    <java.version>1.8</java.version>
  </properties>

  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
    </dependency>

    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-test</artifactId>
      <scope>test</scope>
    </dependency>

    <dependency>
      <groupId>com.alibaba.spring.boot</groupId>
      <artifactId>dubbo-spring-boot-starter</artifactId>
      <version>2.0.0</version>
    </dependency>
  </dependencies>

  <build>
    <plugins>
      <plugin>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-maven-plugin</artifactId>
```

```
        </plugin>
    </plugins>
</build>

</project>
```

application.properties:

```
spring.application.name=skywalking_dubbo_consumer
server.port=8085
```

IHelloService接口：

```
package com.itcast.api;

public interface IHelloService {
    public String hello();
}
```

简化项目的开发，将IHelloService接口在消费方和提供方都编写一份。

TestController：

```
package com.itcast.skywalking_dubbo_consumer.controller;

import com.alibaba.dubbo.config.annotation.Reference;
import com.itcast.api.IHelloService;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class TestController {
    @Reference(url = "dubbo://127.0.0.1:20880")
    private IHelloService helloService;

    @GetMapping("/hello")
    public String hello(){
        return helloService.hello();
    }
}
```

采用直连而非从注册中心获取服务地址的方式，在@Reference注解中声明

```
url = "dubbo://127.0.0.1:20880"
```

SkywalkingDubboConsumerApplication启动类:



```

package com.itcast.skywalking_dubbo_consumer;

import com.alibaba.dubbo.spring.boot.annotation.EnableDubboConfiguration;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
//添加dubbo生效注解
@EnableDubboConfiguration
public class SkywalkingDubboConsumerApplication {

    public static void main(String[] args) {
        SpringApplication.run(SkywalkingDubboConsumerApplication.class, args);
    }

}

```

需要添加@EnableDubboConfiguration注解。

### 3.1.3 部署方式

- 1、将 skywalking\_dubbo\_consumer.jar 和 skywalking\_dubbo\_provider.jar 上传至 /usr/local/skywalking 目录下。
- 2、首先我们复制两份agent，防止使用的冲突。

```

cd /usr/local/skywalking/apache-skywalking-apm-bin/
cp -r agent agent_dubbo_provider
cp -r agent agent_dubbo_consumer
vi agent_dubbo_provider/config/agent.config

```

修改agent\_dubbo\_provider配置中的应用名为：

```

# The service name in UI
agent.service_name=${SW_AGENT_NAME:dubbo_provider}

```

接着修改agent\_dubbo\_consumer:

```

vi agent_dubbo_consumer/config/agent.config

```

修改应用名：

```

# The service name in UI
agent.service_name=${SW_AGENT_NAME:dubbo_consumer}

```

- 3、先启动provider，等待启动成功。

```
#切换到目录下
cd /usr/local/skywalking
#启动provider
java -javaagent:/usr/local/skywalking/apache-skywalking-apm-
bin/agent_dubbo_provider/skywalking-agent.jar -jar
skywalking_dubbo_provider.jar &
```

```
9 using dubbo version 2.6.0, channel is NettyChannel [channel=[id: 0x1ba359bd, /192.168.62.139:
37750 => /192.168.62.139:20880]], dubbo version: 2.6.0, current host: 192.168.62.139
2019-11-25 23:38:36.790 INFO 11996 --- [main] c.a.d.remoting.transport.AbstractClie
nt : [DUBBO] Start NettyClient /192.168.62.139 connect to the server /192.168.62.139:20880, d
ubbo version: 2.6.0, current host: 192.168.62.139
2019-11-25 23:38:37.196 INFO 11996 --- [main] com.alibaba.dubbo.config.AbstractConf
ig : [DUBBO] Refer dubbo service com.itcast.api.IHelloService from url dubbo://127.0.0.1:2088
0/com.itcast.api.IHelloService?application=skywalking_dubbo_consumer&dubbo=2.6.0&interface=com.
itcast.api.IHelloService&methods=hello&pid=11996&register.ip=192.168.62.139&revision=0.0.1-SNAP
SHOT&side=consumer&timestamp=1574696316299, dubbo version: 2.6.0, current host: 192.168.62.139
2019-11-25 23:38:38.239 INFO 11996 --- [main] o.s.s.concurrent.ThreadPoolTaskExecut
or : Initializing ExecutorService 'applicationTaskExecutor'
2019-11-25 23:38:39.325 INFO 11996 --- [main] o.s.b.a.e.web.EndpointLinksResolver
: Exposing 2 endpoint(s) beneath base path '/actuator'
2019-11-25 23:38:39.652 INFO 11996 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServ
er : Tomcat started on port(s): 8085 (http) with context path ''
2019-11-25 23:38:39.656 INFO 11996 --- [main] c.i.s.SkywalkingDubboConsumerApplicat
ion : started SkywalkingDubboConsumerApplication in 26.467 seconds (JVM running for 32.342)
```

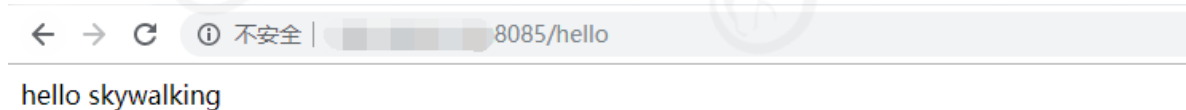
出现如图所示内容，应用就已经启动成功了。

4、启动consumer，等待启动成功。

```
#启动consumer
java -javaagent:/usr/local/skywalking/apache-skywalking-apm-
bin/agent_dubbo_consumer/skywalking-agent.jar -jar
skywalking_dubbo_consumer.jar &
```

5、调用接口，接口地址为：<http://虚拟机IP地址:8085/hello>

6、此时如果页面显示

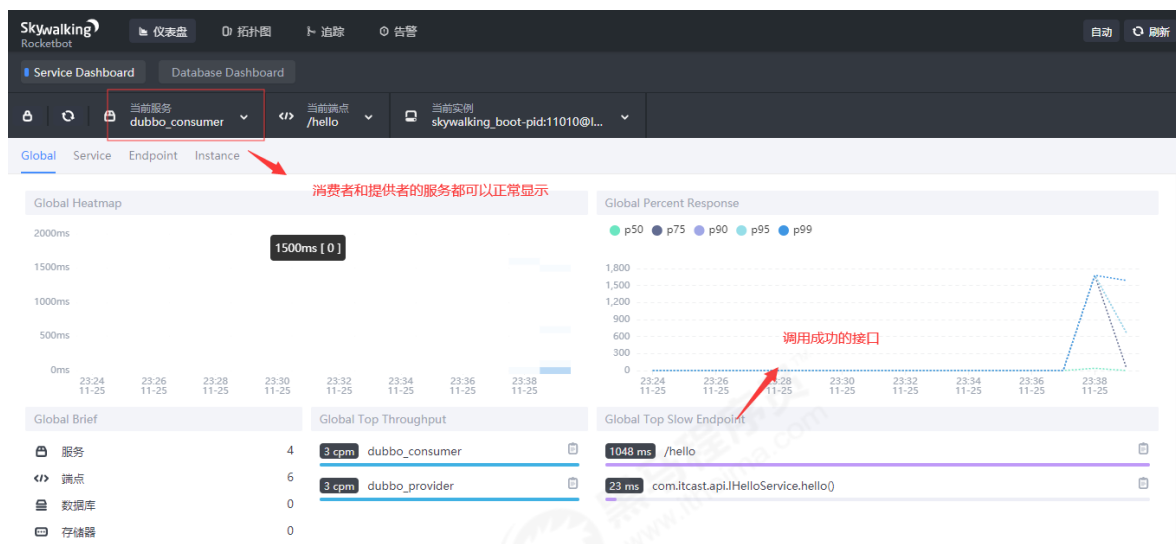


The screenshot shows a web browser window with the address bar displaying '8085/hello'. The page content shows 'hello skywalking'.

那么dubbo的调用就成功了。

7、打开skywalking查看dubbo调用的监控情况。

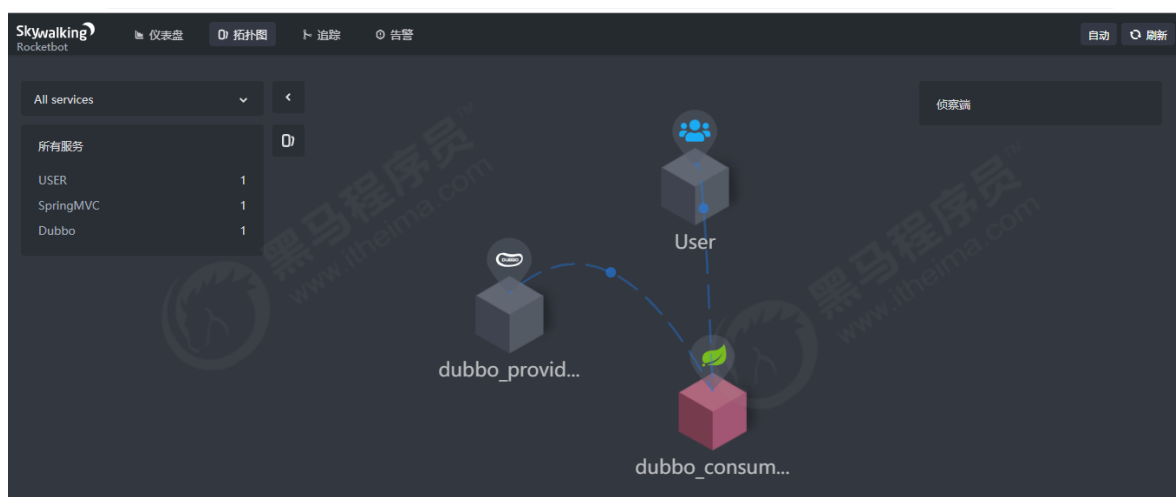
仪表盘：



目前 dubbo\_provider 和 dubbo\_consumer 的服务已经出现，同时出现了两个接口分别是：

- /hello接口，是浏览器调用dubb\_consumer的http接口
- com.itcast.api.IHelloService.hello()是dubbo\_consumer调用dubbo\_provider的dubbo接口

拓扑图：

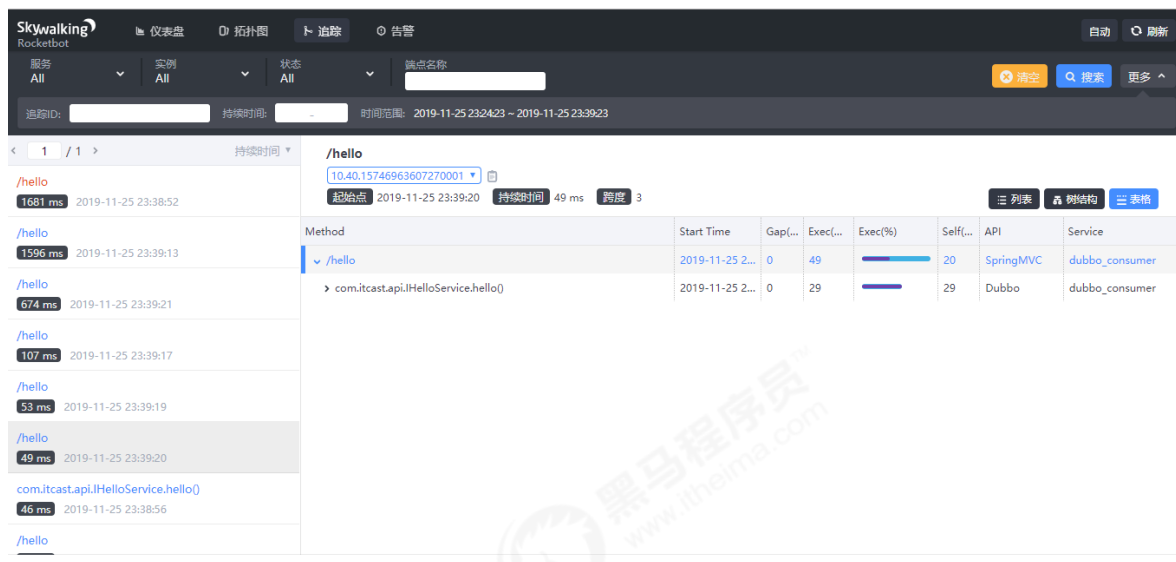


该图中已经表示出了一个调用的链路关系：

User(浏览器) ----> dubber\_consumer ----> dubbo\_provider

并且在服务的上方标识出了每个服务代表的内容，dubbo\_consumer是SpringMvc的服务，而dubbo\_provider是Dubbo的服务。

追踪：



追踪图中显示本次调用耗时49ms，其中dubbo接口耗时29ms，那么另外的20ms其实是SpringMVC接口的开销，这样就能很好的评估出每个环节的耗时间。

## 3.2 MySql调用监控

### 3.2.1 使用docker启动Mysql

虚拟机中已经安装了docker，我们先将docker启动：

```
systemctl start docker
```

使用docker命令启动mysql：

```
docker run -di --name=skywalking_mysql -p 33306:3306 -e  
MYSQL_ROOT_PASSWORD=123456 centos/mysql-57-centos7
```

MYSQL\_ROOT\_PASSWORD环境变量指定root的密码为123456

这样就可以在外部访问mysql了。使用工具连接mysql，端口为33306密码为123456。创建数据库：

The image shows a '创建数据库' (Create Database) dialog box. It has three input fields: '数据库名称' (Database Name) with the value 'skywalking', '基字符集' (Character Set) with the value 'utf8', and '数据库排序规则' (Collation) with the value 'utf8\_unicode\_ci'. At the bottom, there are two buttons: '创建' (Create) and '取消(L)' (Cancel).

执行建表语句：

```
CREATE TABLE `t_user` (  
  `id` int(11) NOT NULL AUTO_INCREMENT,  
  `name` varchar(50) DEFAULT NULL,  
  PRIMARY KEY (`id`)  
) ENGINE=InnoDB DEFAULT CHARSET=utf8
```

插入几条数据：

```
insert into `t_user`(`name`) values ('张三'),('李四'),('王五');
```

### 3.2.2 Spring Data JDBC访问Mysql

创建一个Spring Boot工程，集成Spring Data JDBC。可以直接使用资源文件中提供的skywalking\_mysql.jar。

pom文件：

```
<?xml version="1.0" encoding="UTF-8"?>  
<project xmlns="http://maven.apache.org/POM/4.0.0"  
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0  
      https://maven.apache.org/xsd/maven-4.0.0.xsd">  
  <modelVersion>4.0.0</modelVersion>  
  <parent>  
    <groupId>org.springframework.boot</groupId>  
    <artifactId>spring-boot-starter-parent</artifactId>  
    <version>2.1.10.RELEASE</version>  
    <relativePath/> <!-- lookup parent from repository -->  
  </parent>  
  <groupId>com.itcast</groupId>  
  <artifactId>skywalking_mysql</artifactId>  
  <version>0.0.1-SNAPSHOT</version>  
  <name>skywalking_mysql</name>  
  <description>Demo project for Spring Boot</description>  
  
  <properties>  
    <java.version>1.8</java.version>  
  </properties>  
  
  <dependencies>  
    <dependency>  
      <groupId>org.springframework.boot</groupId>  
      <artifactId>spring-boot-starter-data-jdbc</artifactId>  
    </dependency>  
    <dependency>  
      <groupId>org.springframework.boot</groupId>  
      <artifactId>spring-boot-starter-web</artifactId>  
    </dependency>  
  
    <dependency>  
      <groupId>org.springframework.boot</groupId>  
      <artifactId>spring-boot-starter-test</artifactId>  
      <scope>test</scope>  
    </dependency>
```

```

        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>5.1.46</version>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>
</project>

```

引入了 `spring-boot-starter-data-jdbc`, 由于使用了 5.7 的 mysql 版本, 所以驱动版本固定为 5.1.46。

pojo类:

```

package com.itcast.skywalking_mysql.pojo;

import org.springframework.data.annotation.Id;
import org.springframework.data.relational.core.mapping.Table;

@Table("t_user")
public class User {
    @Id
    private Integer id;
    private String name;

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    @Override
    public String toString() {
        return "User{" +
            "id=" + id +
            ", name='" + name + '\'' +

```

```

        '}}';
    }
}

```

添加Table注解，修改表明为t\_user。

dao接口：

```

package com.itcast.skywalking_mysql.dao;

import com.itcast.skywalking_mysql.pojo.User;
import org.springframework.data.repository.CrudRepository;

public interface UserRepository extends CrudRepository<User, Integer> {
}

```

controller：

```

package com.itcast.skywalking_mysql.controller;

import com.itcast.skywalking_mysql.dao.UserRepository;
import com.itcast.skywalking_mysql.pojo.User;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import java.util.ArrayList;
import java.util.List;

@RestController
public class MysqlController {

    @Autowired
    private UserRepository userRepository;

    @GetMapping("/users")
    public List<User> findAll(){
        List<User> result = new ArrayList<>();
        //使用迭代器进行遍历
        userRepository.findAll().forEach((user) -> {
            result.add(user);
        });

        return result;
    }
}

```

由于Spring Data JDBC的findAll方法返回的是一个迭代器，所以需要遍历迭代器将数据进行返回。



启动类：

```
package com.itcast.skywalking_mysql;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class SkywalkingMysqlApplication {

    public static void main(String[] args) {
        SpringApplication.run(SkywalkingMysqlApplication.class, args);
    }

}
```

application.properties：

```
spring.datasource.url=jdbc:mysql://localhost:33306/skywalking
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
spring.datasource.username=root
spring.datasource.password=123456
server.port=8087
```

### 3.2.3 部署方式

- 1、将 skywalking\_mysql.jar 上传至 /usr/local/skywalking 目录下。
- 2、首先我们复制agent，防止使用的冲突。

```
cd /usr/local/skywalking/apache-skywalking-apm-bin/
cp -r agent agent_mysql
vi agent_mysql/config/agent.config
```

修改agent\_mysql配置中的应用名为：

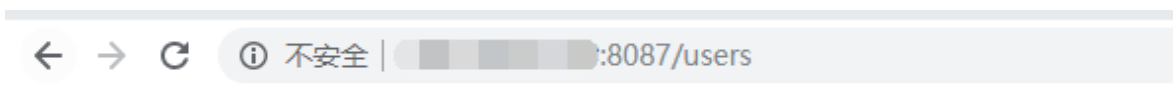
```
# The service name in UI
agent.service_name=${SW_AGENT_NAME:skywalking_mysql}
```

- 3、启动skywalking\_mysql应用，等待启动成功。

```
#切换到目录下
cd /usr/local/skywalking
#启动spring boot
java -javaagent:/usr/local/skywalking/apache-skywalking-apm-
bin/agent_mysql/skywalking-agent.jar -jar skywalking_mysql.jar &
```

- 4、调用接口，接口地址为：<http://虚拟机IP地址:8087/users>

- 5、此时如果页面显示

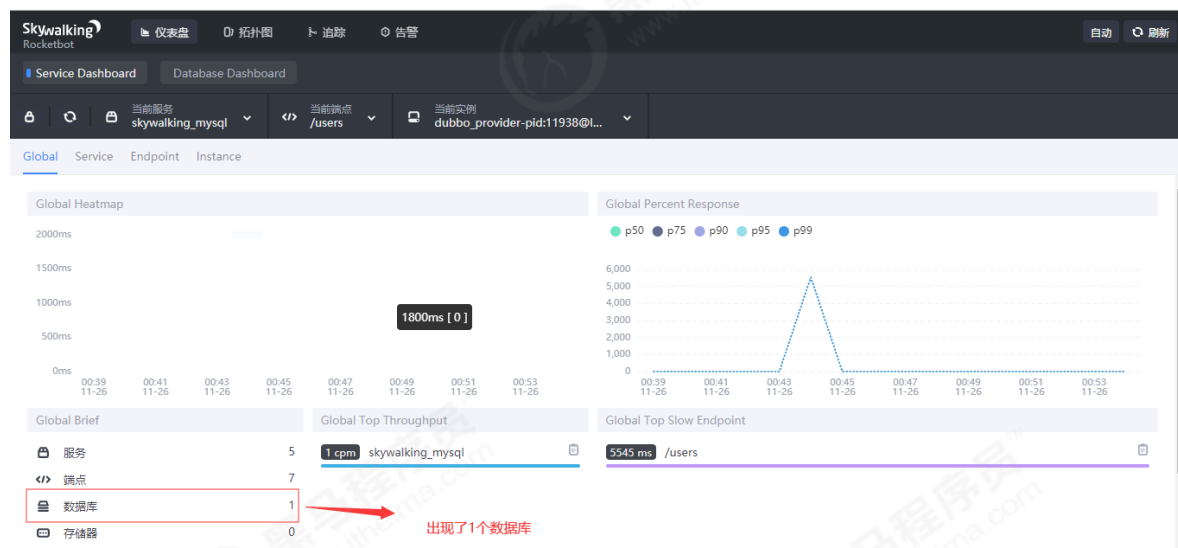


```
[{"id":1,"name":"张三"}, {"id":2,"name":"李四"}, {"id":3,"name":"王五"}]
```

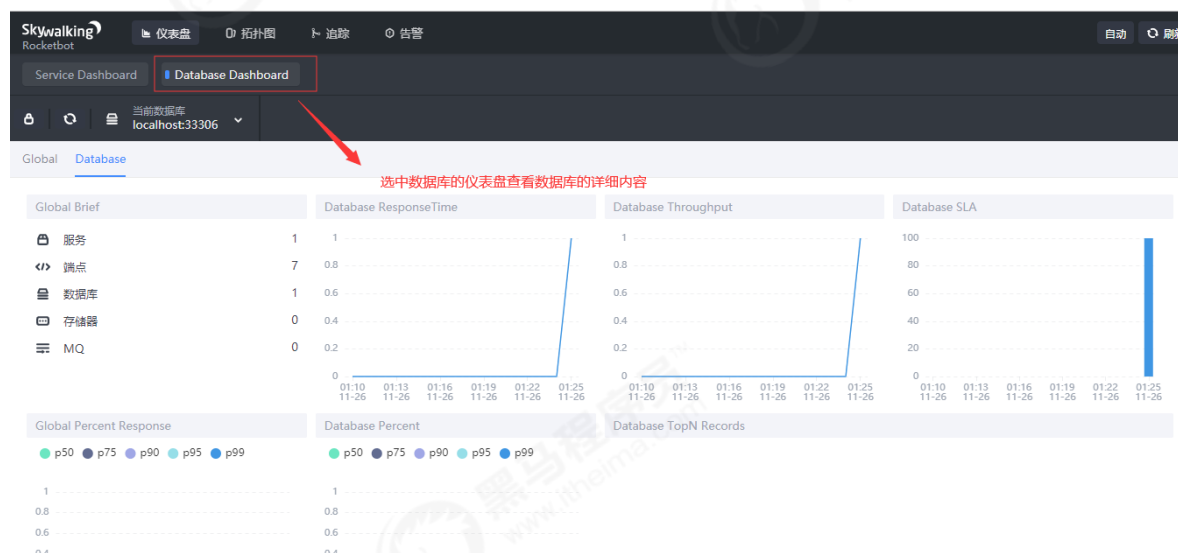
那么mysql的调用就成功了。

6、打开skywalking查看mysql调用的监控情况。

服务仪表盘：

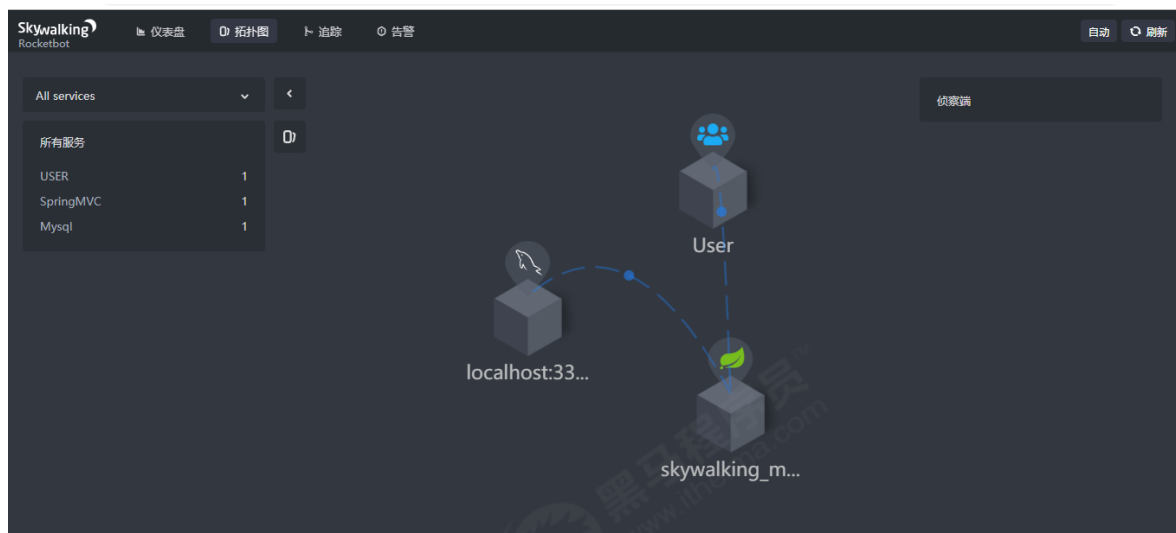


数据库仪表盘：



点击数据库仪表盘可以看到详细的数据库响应时长、吞吐量、SLA等数据。

拓扑图：



该图中已经表示出了一个调用的链路关系：

User(浏览器) ----> skywalking\_mysql ----> localhost:33306

并且在服务的上方标识出了每个服务代表的内容，skywalking\_mysql是SpringMvc的服务，而localhost:33306是mysql的服务。

追踪：

<div> <div>/users</div> <div>14.30.15747027254500001</div> <div>起始点 2019-11-26 01:25:25</div> <div>持续时间 5 ms</div> <div>跨度 2</div> <div>列表 树结构 表格</div> </div>							
Method	Start Time	Gap(...)	Exec(...)	Exec(%)	Self(...)	API	Service
▼ /users	2019-11-26 0...	0	5	<div></div>	4	SpringMVC	skywalking_mysql
Mysql/JDBI/PreparedStatement/executeQuery	2019-11-26 0...	0	1	<div></div>	1	mysql-conne...	skywalking_mysql

追踪图中显示本次调用耗时5ms，其中spring MVC接口耗时4ms，那么另外的1ms是调用Mysql的耗时。

点击mysql的调用，可以看到详细的sql语句。

## 跨度信息

标记:

端点: Mysql/JDBI/PreparedStatement/executeQuery

跨度类型: Exit

组件: mysql-connector-java

Peer: localhost:33306

失败: false

db.type: sql

db.instance: skywalking

db.statement: `SELECT t_user.id AS id, t_user.name AS name FROM t_user`

这样可以很好的定位问题产生的原因，特别是在某些sql语句执行慢的场景下。

## 3.3 Skywalking常用插件

### 3.3.1 配置覆盖

在之前的案例中，我们每次部署应用都需要复制一份agent，修改其中的服务名称，这样显得非常麻烦。可以使用Skywalking提供的配置覆盖功能通过启动命令动态指定服务名，这样agent只需要部署一份即可。Skywalking支持的几种配置方式：

#### 系统配置 ( System properties )

使用 `skywalking.` + 配置文件中的配置名作为系统配置项来进行覆盖。

- 为什么需要添加前缀?  
agent的系统配置和环境与目标应用共享，所以加上前缀可以有效的避免冲突。
- 案例  
通过 如下进行 `agent.service_name` 的覆盖

```
-Dskywalking.agent.service_name=skywalking_mysql
```

#### 探针配置 ( Agent options )

Add the properties after the agent path in JVM arguments.

```
-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'
```

#### 系统环境变量 ( System environment variables )

- 案例  
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}
```

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 &
```

### 3.3.2 获取追踪ID

Skywalking提供我们Trace工具包，用于在追踪链路时进行信息的打印或者获取对应的追踪ID。我们使用Spring Boot编写一个案例，也可以使用资源下的 skywalking\_plugins.jar 进行测试。

pom :

```
<?xml version="1.0" encoding="UTF-8"?>  
<project xmlns="http://maven.apache.org/POM/4.0.0"  
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0  
https://maven.apache.org/xsd/maven-4.0.0.xsd">  
  <modelVersion>4.0.0</modelVersion>  
  <parent>  
    <groupId>org.springframework.boot</groupId>  
    <artifactId>spring-boot-starter-parent</artifactId>  
    <version>2.1.10.RELEASE</version>  
    <relativePath/> <!-- lookup parent from repository -->  
  </parent>  
  <groupId>com.itcast</groupId>  
  <artifactId>skywalking_plugins</artifactId>  
  <version>0.0.1-SNAPSHOT</version>  
  <name>skywalking_plugins</name>  
  <description>Demo project for Spring Boot</description>  
  
  <properties>  
    <java.version>1.8</java.version>  
    <skywalking.version>6.5.0</skywalking.version>  
  </properties>  
  
  <dependencies>  
    <dependency>  
      <groupId>org.springframework.boot</groupId>  
      <artifactId>spring-boot-starter-web</artifactId>  
    </dependency>  
  
    <dependency>  
      <groupId>org.projectlombok</groupId>  
      <artifactId>lombok</artifactId>  
      <optional>true</optional>  
    </dependency>  
    <dependency>  
      <groupId>org.springframework.boot</groupId>  
      <artifactId>spring-boot-starter-test</artifactId>
```

```

        <scope>test</scope>
    </dependency>
    <!--skywalking trace工具包-->
    <dependency>
        <groupId>org.apache.skywalking</groupId>
        <artifactId>apm-toolkit-trace</artifactId>
        <version>${skywalking.version}</version>
    </dependency>
</dependencies>

<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
    </plugins>
</build>

</project>

```

添加了对应的坐标：

```

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

```

本案例中使用6.5.0的版本号。

PluginController：

```

package com.itcast.skywalking_plugins.controller;

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。

## 部署方式

1、将 skywalking\_plugins.jar 上传至 /usr/local/skywalking 目录下。

2、启动skywalking\_plugins应用，等待启动成功。

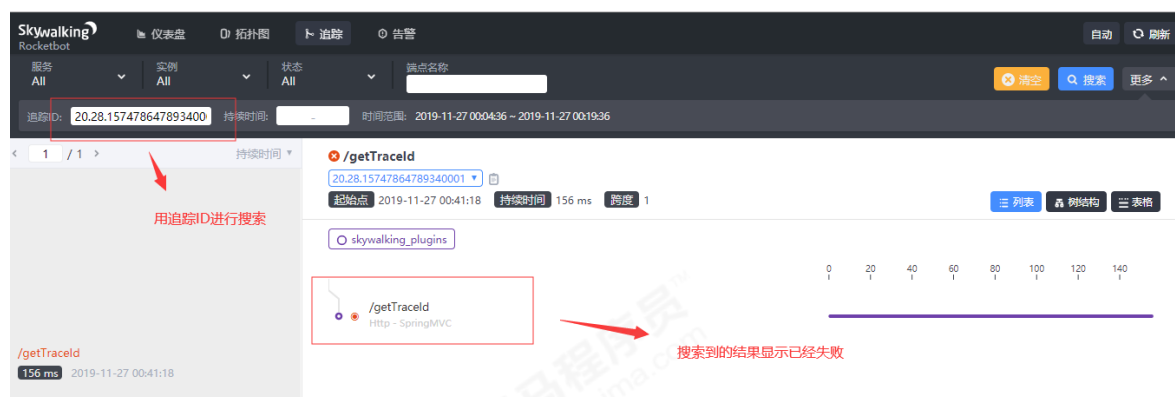
```
java -javaagent:/usr/local/skywalking/apache-skywalking-apm-  
bin/agent/skywalking-agent.jar -  
Dskywalking.agent.service_name=skywalking_plugins -jar skywalking_plugins.jar &
```

4、调用接口，接口地址为：<http://虚拟机IP地址:8088/getTraceId>

5、此时如果页面显示



可以看到追踪ID已经打印出来，然后我们在RocketBot上进行搜索。



可以搜索到对应的追踪记录，但是显示调用是失败的，这是因为使用了ActiveSpan.error方法。点开追踪的详细信息：



日志.

时间: 2019-11-27 00:41:19

event:

error

事件为error

error.kind:

java.lang.RuntimeException

调用方法时传递的异常类型

message:

Test-Error-Throwable

调用方法时传递的参数

stack:

```
java.lang.RuntimeException: Test-Error-Throwable
at com.itcast.skywalking_plugins.controller.PluginController.getTraceId$original$Zc7kY01V(Pl
at com.itcast.skywalking_plugins.controller.PluginController.getTraceId$original$Zc7kY01V$acc
at com.itcast.skywalking_plugins.controller.PluginController$auxiliary$bob2zabv.call(Unknown
at org.apache.skywalking.apm.agent.core.plugin.interceptor.enhance.InstMethodsInter.intercept
at com.itcast.skywalking_plugins.controller.PluginController.getTraceId(PluginController.java
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
at java.lang.reflect.Method.invoke(Method.java:498)
at org.springframework.web.method.support.InvocableHandlerMethod.doInvoke(InvocableHandlerMet
at org.springframework.web.method.support.InvocableHandlerMethod.invokeForRequest(InvocableHa
at org.springframework.web.servlet.mvc.method.annotation.ServletInvocableHandlerMethod.invoke
at org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter.invoke
at org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter.handle
at org.springframework.web.servlet.mvc.method.AbstractHandlerMethodAdapter.handle(AbstractHar
```

异常堆栈

异常的信息包含了以下几个部分：

- 1.事件类型为error
- 2.调用方法时传递的异常类型RuntimeException
- 3.调用方法时传递的异常信息Test-Error-Throwable
- 4.异常堆栈

通过上述内容，我们可以根据业务来定制调用异常时的详细信息。

时间: 2019-11-27 00:41:19

event:

info

message:

Test-Info-Msg

info信息

时间: 2019-11-27 00:41:19

event:

debug

message:

Test-debug-Msg

debug信息

除了异常信息之外，还有info信息和debug信息也都会被打印。

### 3.3.3 过滤指定的端点

在开发过程中，有一些端点（接口）并不需要进行监控，比如Swagger相关的端点。这个时候我们就可以使用Skywalking提供的过滤插件来进行过滤。在skywalking\_plugins中编写两个接口进行测试：

```
package com.itcast.skywalking_plugins.controller;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class FilterController {
    //此接口可以被追踪
    @GetMapping("/include")
    public String include(){
        return "include";
    }

    //此接口不可被追踪
    @GetMapping("/exclude")
    public String exclude(){
        return "exclude";
    }
}
```

#### 部署方式

- 1、将 skywalking\_plugins.jar 上传至 /usr/local/skywalking 目录下。
- 2、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/apache-skywalking-apm-bin
cp optional-plugins/apm-trace-ignore-plugin-6.4.0.jar plugins/apm-trace-ignore-plugin-6.4.0.jar
```

- 3、启动skywalking\_plugins应用，等待启动成功。

```
java -javaagent:/usr/local/skywalking/apache-skywalking-apm-bin/agent/skywalking-agent.jar -
Dskywalking.agent.service_name=skywalking_plugins -
Dskywalking.trace.ignore_path=/exclude jar skywalking_plugins.jar &
```

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

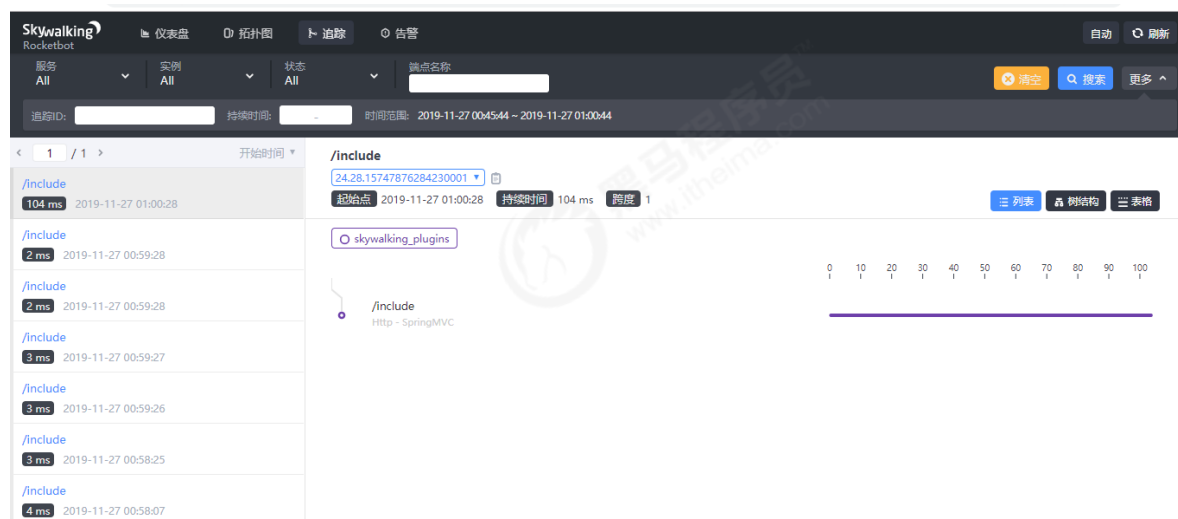
- \*\* 匹配0或者更多的目录

4、调用接口，接口地址为：

<http://虚拟机IP地址:8088/exclude>

<http://虚拟机IP地址:8088/include>

5、在追踪中进行查看：



exclude接口已经被过滤，只有include接口能被看到。

## 3.4 告警功能

### 3.4.1 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.
  service_sla_rule:
    # Metrics value need to be long, double or int
    metrics-name: service_sla
    op: "<"
    threshold: 8000
    # The length of time to evaluate the metrics
```

```

period: 10
# How many times after the metrics match the condition, will trigger alarm
count: 2
# How many times of checks, the alarm keeps silence after alarm triggered,
default as same as period.
silence-period: 3
message: Successful rate of service {name} is lower than 80% in 2 minutes of
last 10 minutes
service_p90_sla_rule:
# Metrics value need to be long, double or int
metrics-name: service_p90
op: ">"
threshold: 1000
period: 10
count: 3
silence-period: 5
message: 90% response time of service {name} is more than 1000ms in 3
minutes of last 10 minutes
service_instance_resp_time_rule:
metrics-name: service_instance_resp_time
op: ">"
threshold: 1000
period: 10
count: 2
silence-period: 5
message: Response time of service instance {name} is more than 1000ms in 2
minutes of last 10 minutes
# Active endpoint related metrics alarm will cost more memory than service and
service instance metrics alarm.
# Because the number of endpoint is much more than service and instance.
#
# endpoint_avg_rule:
#   metrics-name: endpoint_avg
#   op: ">"
#   threshold: 1000
#   period: 10
#   count: 2
#   silence-period: 5
#   message: Response time of endpoint {name} is more than 1000ms in 2 minutes
of last 10 minutes

webhooks:
# - 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	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。

### 3.4.2 告警功能测试代码

编写告警功能接口来进行测试，创建skywalking\_alarm项目。可以直接使用资源中的skywalking\_alarm.jar。

AlarmController：

```
package com.itcast.skywalking_alarm.controller;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class AlarmController {

    //每次调用睡眠1.5秒，模拟超时的报警
    @GetMapping("/timeout")
    public String timeout(){
        try {
            Thread.sleep(1500);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        return "timeout";
    }
}
```

该接口主要用于模拟超时，多次调用之后就可以生成告警信息。

WebHooks：

```
package com.itcast.skywalking_alarm.controller;
```

```

import com.itcast.skywalking_alarm.pojo.AlarmMessage;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RestController;

import java.util.ArrayList;
import java.util.List;

@RestController
public class webHooks {

    private List<AlarmMessage> lastList = new ArrayList<>();

    //产生告警时调用的接口
    @PostMapping("/webhook")
    public void webhook(@RequestBody List<AlarmMessage> alarmMessageList){
        lastList = alarmMessageList;
    }

    //展示告警的信息
    @GetMapping("/show")
    public List<AlarmMessage> show(){
        return lastList;
    }
}

```

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:

```

package com.itcast.skywalking_alarm.pojo;

public class AlarmMessage {

```

```
private int scopeId;
private String name;
private int id0;
private int id1;
private String alarmMessage;
private long startTime;

public int getScopeId() {
    return scopeId;
}

public void setScopeId(int scopeId) {
    this.scopeId = scopeId;
}

public String getName() {
    return name;
}

public void setName(String name) {
    this.name = name;
}

public int getId0() {
    return id0;
}

public void setId0(int id0) {
    this.id0 = id0;
}

public int getId1() {
    return id1;
}

public void setId1(int id1) {
    this.id1 = id1;
}

public String getAlarmMessage() {
    return alarmMessage;
}

public void setAlarmMessage(String alarmMessage) {
    this.alarmMessage = alarmMessage;
}

public long getStartTime() {
    return startTime;
}

public void setStartTime(long startTime) {
    this.startTime = startTime;
}

@Override
public String toString() {
    return "AlarmMessage{" +
```

```

        "scopeId=" + scopeId +
        ", name='" + name + '\'' +
        ", id0=" + id0 +
        ", id1=" + id1 +
        ", alarmMessage='" + alarmMessage + '\'' +
        ", startTime=" + startTime +
        '}}';
    }
}

```

实体类用于接口告警信息。

### 3.4.3 部署测试

首先需要修改告警规则配置文件，将webhook地址修改为：

```

webhooks:
  - http://127.0.0.1:8089/webhook

```

然后重启skywalking。

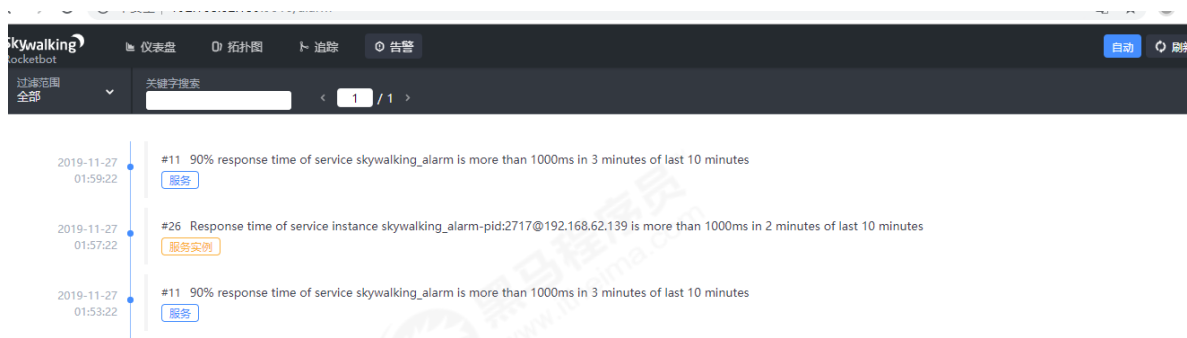
- 1、将 skywalking\_alarm.jar 上传至 /usr/local/skywalking 目录下。
- 2、启动skywalking\_alarm应用，等待启动成功。

```

java -javaagent:/usr/local/skywalking/apache-skywalking-apm-
bin/agent/skywalking-agent.jar -Dskywalking.agent.service_name=skywalking_alarm
-jar skywalking_alarm.jar

```

- 3、不停调用接口，接口地址为：<http://虚拟机IP:8089/timeout>
- 4、直到出现告警：



- 5、查看告警信息接口：<http://虚拟机IP:8089/show>

```

[{"scopeId":2,"name":"skywalking_alarm-pid:27170@192.168.62.139","id0":26,"id1":0,"alarmMessage":"Response time of service instance skywalking_alarm-pid:27170@192.168.62.139 is more than 1000ms in 2 minutes of last 10 minutes","startTime":1574908823991}]

```

从上图中可以看到，我们已经获取到了告警相关的信息，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。

## 4.Skywalking原理



## 4.1 java agent原理

上文中我们知道，要使用Skywalking去监控服务，需要在其 VM 参数中添加“-javaagent:/usr/local/skywalking/apache-skywalking-apm-bin/agent/skywalking-agent.jar”。这里就使用到了java agent技术。

### 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。

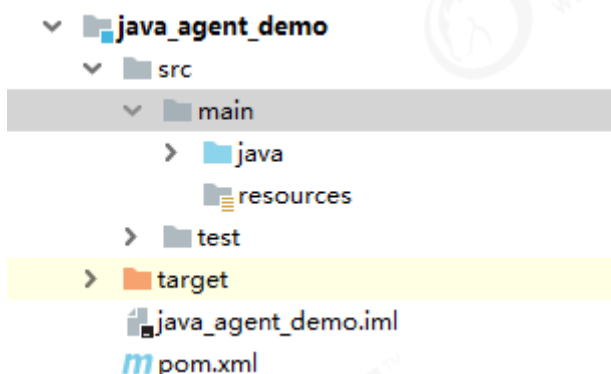
### 如何使用java agent？

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-Transform-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。

### 4.1.1 搭建java agent工程

使用maven创建java\_agent\_demo工程：



在java文件夹下新建PreMainAgent类：

```
import java.lang.instrument.Instrumentation;

public class PreMainAgent {

    /**
     * 在这个 premain 函数中，开发者可以进行对类的各种操作。
     * 1、agentArgs 是 premain 函数得到的程序参数，随同 “- javaagent”一起传入。与 main
    函数不同的是，
     * 这个参数是一个字符串而不是一个字符串数组，如果程序参数有多个，程序将自行解析这个字符串。
     * 2、Inst 是一个 java.lang.instrument.Instrumentation 的实例，由 JVM 自动传入。*
     * java.lang.instrument.Instrumentation 是 instrument 包中定义的一个接口，也是这
    个包的核心部分，
     * 集中了其中几乎所有的功能方法，例如类定义的转换和操作等等。
```

```

    * @param agentArgs
    * @param inst
    */
    public static void premain(String agentArgs, Instrumentation inst) {
        System.out.println("=====premain方法执行1=====");
        System.out.println(agentArgs);
    }

    /**
     * 如果不存在 premain(String agentArgs, Instrumentation inst)
     * 则会执行 premain(String agentArgs)
     * @param agentArgs
     */
    public static void premain(String agentArgs) {
        System.out.println("=====premain方法执行2=====");
        System.out.println(agentArgs);
    }
}

```

类中提供两个静态方法，方法名均为premain，不能拼错。

在pom文件中添加打包插件：

```

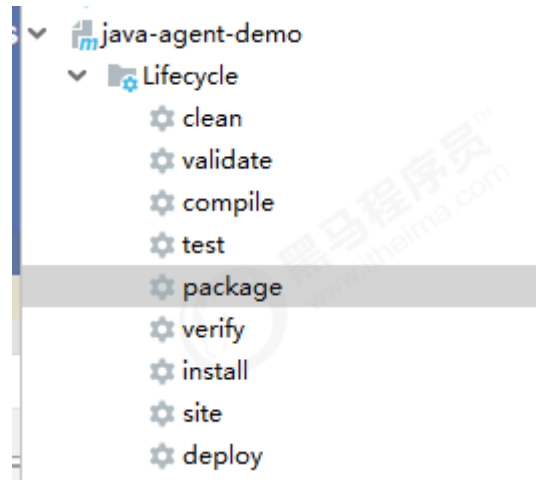
<build>
<plugins>
    <plugin>
        <artifactId>maven-assembly-plugin</artifactId>
        <configuration>
            <appendAssemblyId>>false</appendAssemblyId>
            <descriptorRefs>
                <descriptorRef>jar-with-dependencies</descriptorRef>
            </descriptorRefs>
            <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：



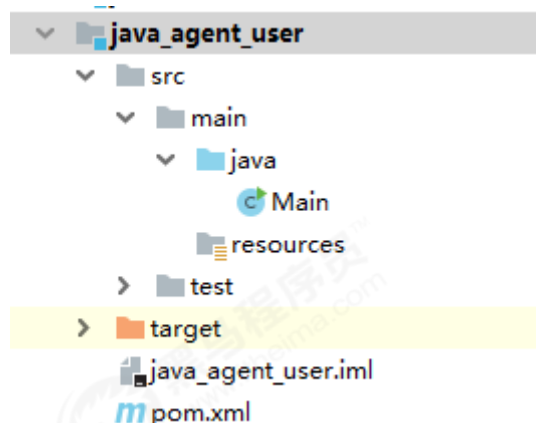
打包成功之后，复制打包出来的jar包地址。

```
[INFO] No tests to run.
[INFO] --- maven-jar-plugin:3.1.0:jar (default-jar) @ java-agent-demo ---
[INFO] Building jar: \java_agent_demo\target\java-agent-demo-1.0-SNAPSHOT.jar
[INFO] BUILD SUCCESS
[INFO] Total time: 1.325 s
[INFO] Finished at: 2019-11-27T09:06:36+08:00
[INFO] Final Memory: 17M/166M
```

复制这个地址

## 4.1.2 搭建主工程

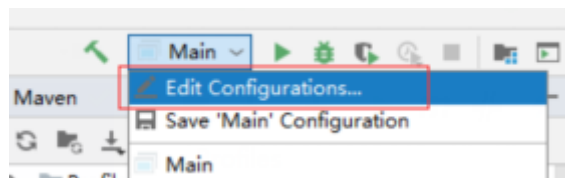
使用maven创建java\_agent\_user工程：



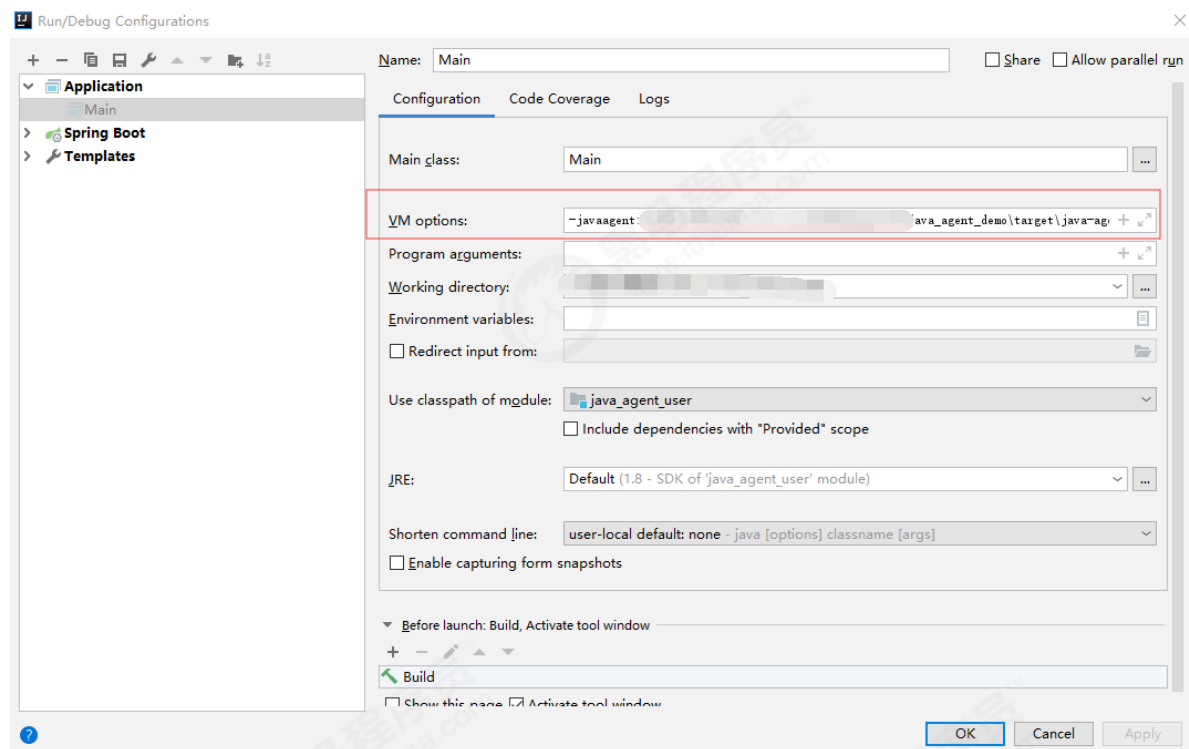
Main类代码：

```
public class Main {
    public static void main(String[] args) {
        System.out.println("Hello world");
    }
}
```

先运行一次，然后点击编辑MAIN启动类：



在VM options中添加代码：

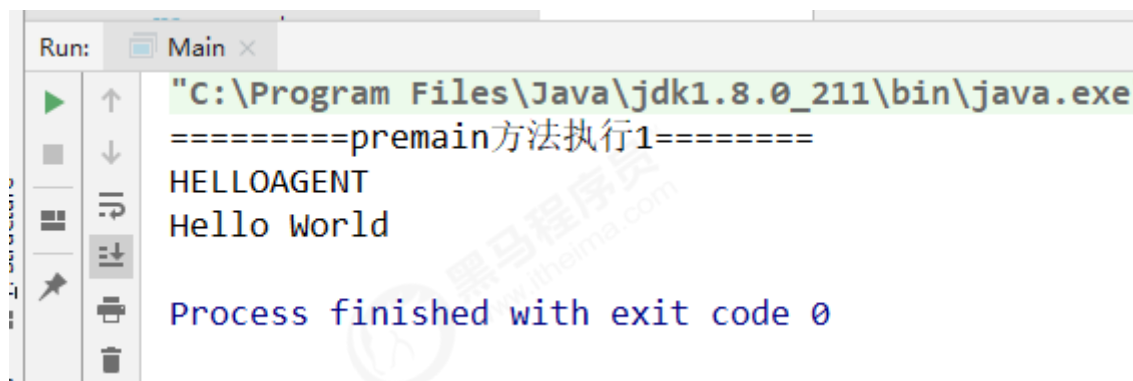


代码为

```
-javaagent:路径\java-agent-demo-1.0-SNAPSHOT.jar=HELLOAGENT
```

启动时加载javaagent，指向上一节中编译出来的java agent工程jar包地址，同时在最后追加参数HELLOAGENT。

运行MAIN方法，查看结果：



可以看到java agent的代码优先于MAIN函数的方法运行，证明java agent运行正常。

### 4.1.3 统计方法调用时间

Skywalking中对每个调用的时长都进行了统计，这一小节中我们会使用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，可以轻松地增强我们已有的代码。

添加依赖：

```
<dependencies>
  <dependency>
    <groupId>net.bytebuddy</groupId>
    <artifactId>byte-buddy</artifactId>
    <version>1.9.2</version>
  </dependency>
  <dependency>
    <groupId>net.bytebuddy</groupId>
    <artifactId>byte-buddy-agent</artifactId>
    <version>1.9.2</version>
  </dependency>
</dependencies>
```

修改PreMainAgent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) {
        //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。

创建完拦截器之后可以通过Byte Buddy的AgentBuilder建造者来构建一个agent对象。AgentBuilder可以对指定的包名前缀来生效，同时需要指定转换器对象。

MyInterceptor类：

```
import net.bytebuddy.implementation.bind.annotation.Origin;  
import net.bytebuddy.implementation.bind.annotation.RuntimeType;  
import net.bytebuddy.implementation.bind.annotation.SuperCall;  
  
import java.lang.reflect.Method;  
import java.util.concurrent.Callable;  
  
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。

重新打包，执行maven package命令。接下来修改主工程代码。主工程将Main类放置到 com.agent 包下。修改代码内容为：

```
package com.agent;

public class Main {
    public static void main(String[] args) {
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("Hello world");
    }
}
```

休眠1秒，使统计时长的演示效果更好一些。执行main方法之后显示结果：

```
"C:\Program Files\Java\jdk1.8.0_211\bin\java.exe"
Hello World
main:1001ms
```

```
Process finished with exit code 0
```

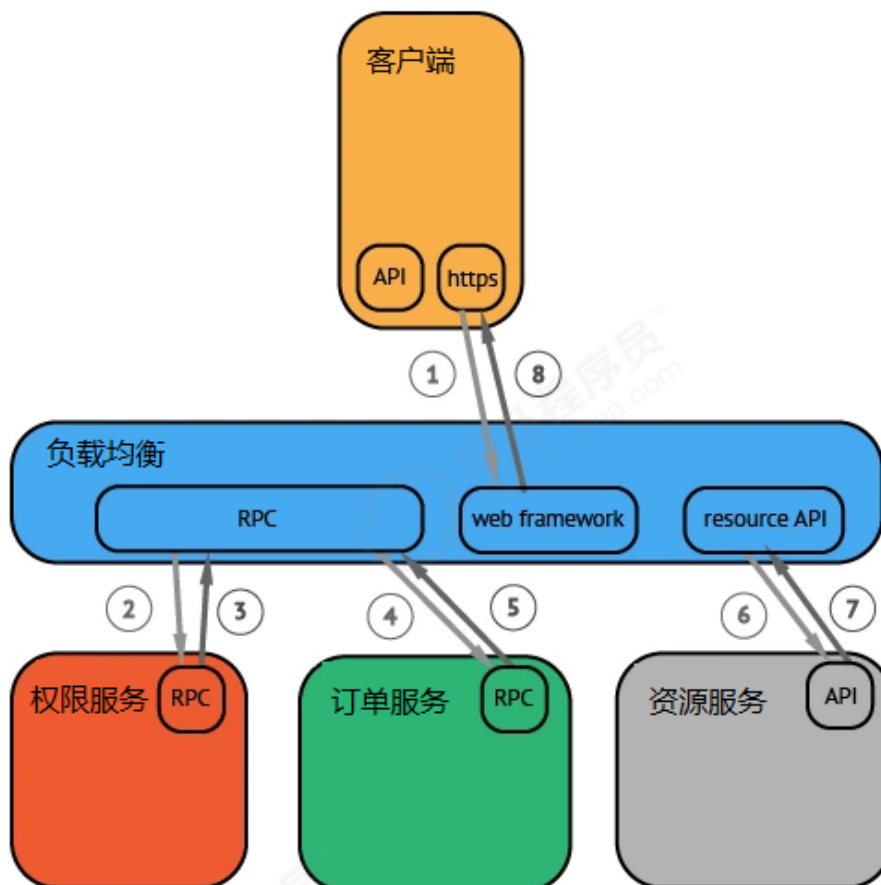
我们在没有修改代码的情况下，利用java agent和Byte Buddy统计出了方法的时长，Skywalking的agent也是基于这些技术来实现统计调用时长。

## 4.2 Open Tracing介绍

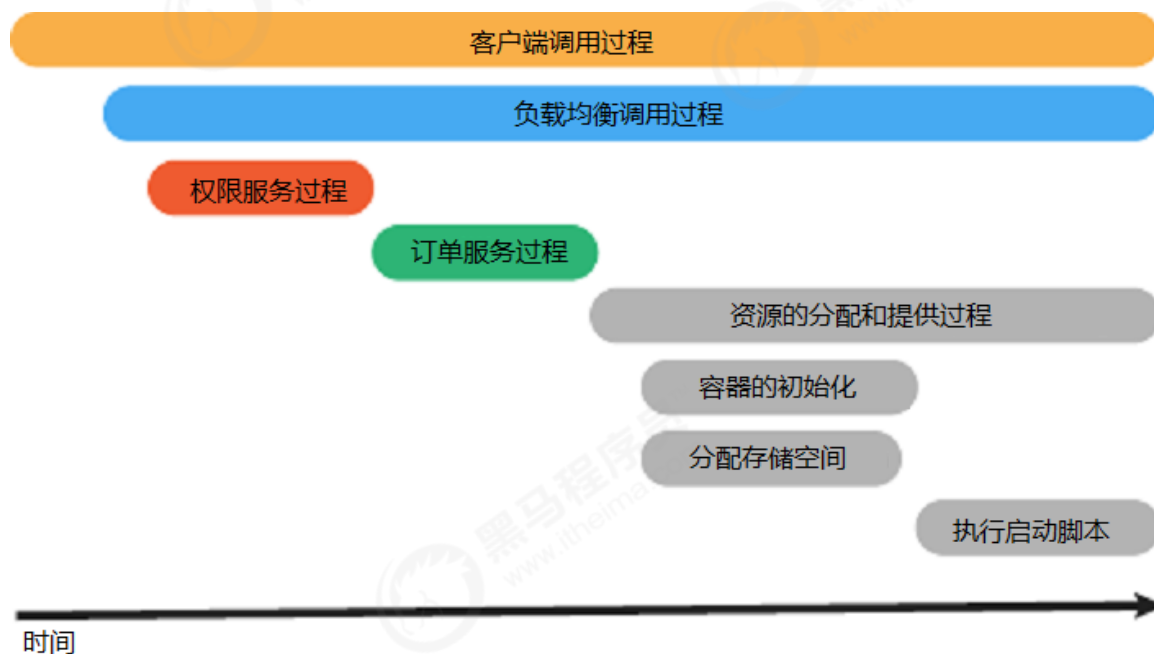
之前的课程中已经简单介绍过Open Tracing一些基础概念，OpenTracing通过提供平台无关、厂商无关的API，使得开发人员能够方便的添加（或更换）追踪系统的实现。OpenTracing中最核心的概念就是Trace。

### 4.2.1 Trace的概念

在广义上，一个trace代表了一个事务或者流程在（分布式）系统中的执行过程。在OpenTracing标准中，trace是多个span组成的一个有向无环图（DAG），每一个span代表trace中被命名并计时的连续性的执行片段。



例如客户端发起的一次请求，就可以认为是一个Trace。将上面的图通过Open Tracing的语义修改完之后做可视化，得到下面的图：



图中每一个色块其实就是一个span。

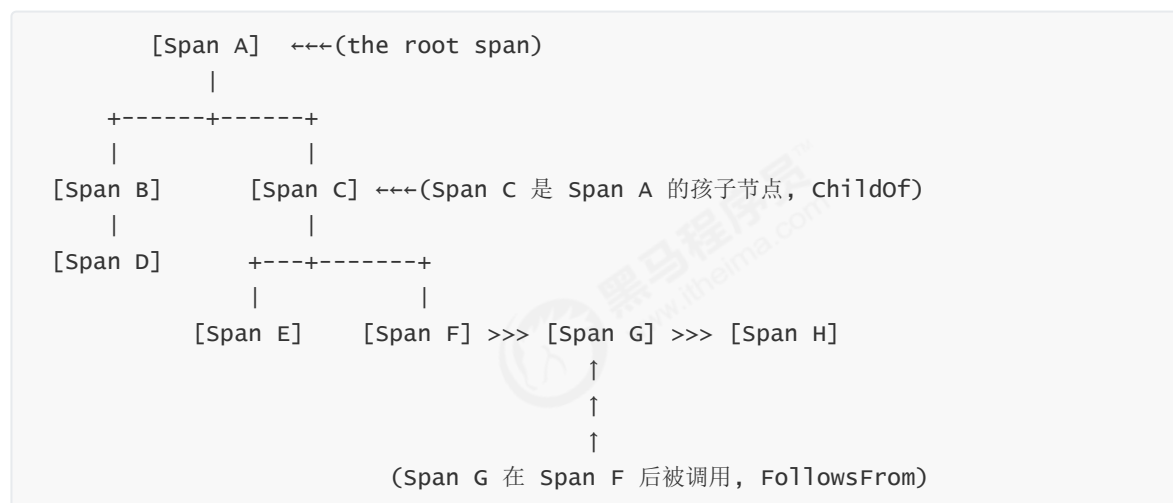
## 4.2.2 Span的概念

一个Span代表系统中具有开始时间和执行时长的逻辑运行单元。span之间通过嵌套或者顺序排列建立逻辑因果关系。



Span里面的信息包括：操作的名字，开始时间和结束时间，可以附带多个 key:value 构成的 Tags(key 必须是String, value可以是 String, bool 或者数字)，还可以附带 Logs 信息(不一定所有的实现都支持)也是 key:value形式。

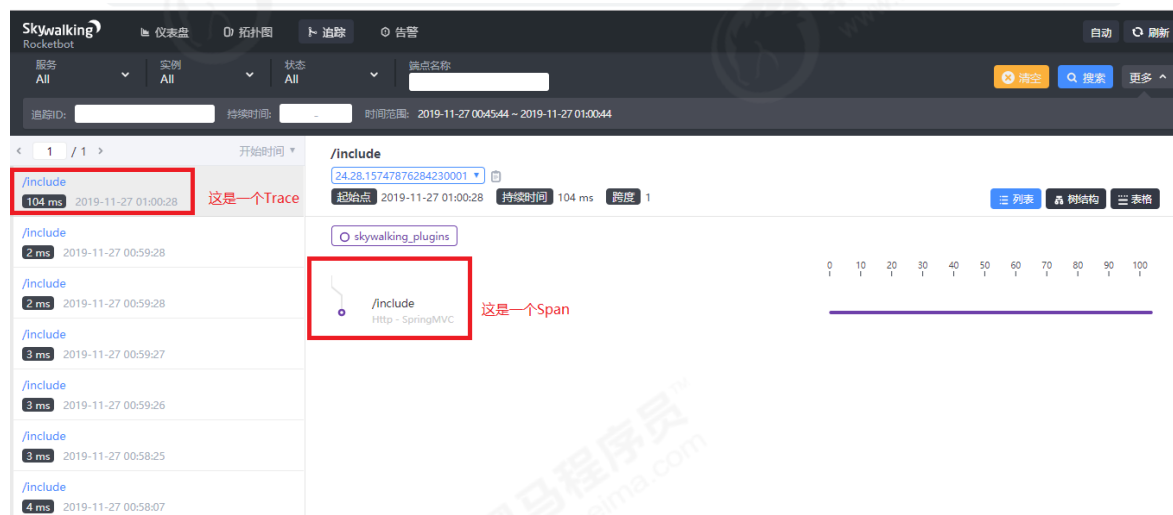
下面例子是一个 Trace，里面有8个 Span：



一个span可以和一个或者多个span间存在因果关系。OpenTracing定义了两种关系：`ChildOf` 和 `FollowsFrom`。这两种引用类型代表了子节点和父节点间的直接因果关系。未来，OpenTracing将支持非因果关系的span引用关系。（例如：多个span被批量处理，span在同一个队列中，等等）

ChildOf 很好理解，就是父亲 Span 依赖另一个孩子 Span。比如函数调用，被调者是调用者的孩子，比如说 RPC 调用，服务端那边的Span，就是 ChildOf 客户端的。很多并发的调用，然后将结果聚合起来的操作，就构成了 ChildOf 关系。

如果父亲 Span 并不依赖于孩子 Span 的返回结果，这时可以说它构成 FollowsFrom 关系。



如图所示，左边的每一条追踪代表一个Trace，而右边时序图中每一个节点就是一个Span。

## 4.2.3 Log的概念

每个span可以进行多次Logs操作，每一次Logs操作，都需要一个带时间戳的时间名称，以及可选的任意大小的存储结构。

如下图是一个异常的Log：

日志.

时间: 2019-11-27 00:41:19

event:

error

事件为error

error.kind:

java.lang.RuntimeException

调用方法时传递的异常类型

message:

Test-Error-Throwable

调用方法时传递的参数

stack:

```
java.lang.RuntimeException: Test-Error-Throwable
at com.itcast.skywalking_plugins.controller.PluginController.getTraceId$original$Zc7kY01V(Pl
at com.itcast.skywalking_plugins.controller.PluginController.getTraceId$original$Zc7kY01V$acc
at com.itcast.skywalking_plugins.controller.PluginController$auxiliary$bob2zabv.call(Unknown
at org.apache.skywalking.apm.agent.core.plugin.interceptor.enhance.InstMethodsInter.intercept
at com.itcast.skywalking_plugins.controller.PluginController.getTraceId(PluginController.java
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
at java.lang.reflect.Method.invoke(Method.java:498)
at org.springframework.web.method.support.InvocableHandlerMethod.doInvoke(InvocableHandlerMet
at org.springframework.web.method.support.InvocableHandlerMethod.invokeForRequest(InvocableHa
at org.springframework.web.servlet.mvc.method.annotation.ServletInvocableHandlerMethod.invoke
at org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter.invoke
at org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter.handle
at org.springframework.web.servlet.mvc.method.AbstractHandlerMethodAdapter.handle(AbstractHar
```

异常堆栈

如下图是两个正常信息的Log，它们都带有时间戳和对应的事件名称、消息内容。

时间: 2019-11-27 00:41:19

event:

info

message:

Test-Info-Msg

info信息

时间: 2019-11-27 00:41:19

event:

debug

message:

Test-debug-Msg

debug信息

## 4.2.4 Tags的概念

每个span可以有多个键值对 ( key:value ) 形式的**Tags**，**Tags**是没有时间戳的，支持简单的对span进行注解和补充。

如下图就是一个Tags的详细信息，其中记录了数据库访问的SQL语句等内容。

### 标记.

端点: Mysql/JDBC/PreparedStatement/executeQuery

跨度类型: Exit

组件: mysql-connector-java

Peer: localhost:33306

失败: false

db.type: sql

db.instance: skywalking

db.statement: SELECT t\_user.id AS id, t\_user.name AS name FROM t\_user