

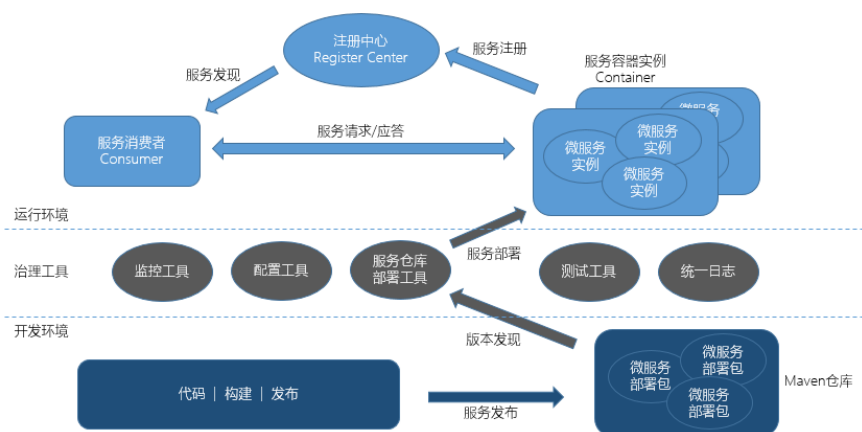
# Table of Contents

概述	1.1
注册及发现	1.1.1
开发及交付	1.1.2
治理工具	1.1.3
数据类型及编码	1.1.4
开发	1.2
环境及依赖	1.2.1
服务注解	1.2.2
DEMO示例	1.2.3
访问及测试	1.2.4
日志处理	1.2.5
部署工具	1.2.6
命令行工具	1.2.7
部署	1.3
Web管理器	1.3.1
Web应用	1.4
开发及治理	1.4.1
Java API	1.5
rewin.ubsj.consumer.Context	1.5.1
rewin.ubsj.consumer.ErrorCode	1.5.2
rewin.ubsj.consumer.Logger	1.5.3
rewin.ubsj.container.ServiceContext	1.5.4
rewin.ubsj.common.Codec	1.5.5
rewin.ubsj.common.JsonCodec	1.5.6
rewin.ubsj.common.XmlCodec	1.5.7
rewin.ubsj.common.Crypto	1.5.8
rewin.ubsj.common.JedisUtil	1.5.9
API网关	1.6
网关部署	1.6.1

# 概述

UBSI是一个轻量级的微服务开发平台，与其他庞杂的微服务开发/治理工具不同，UBSI致力于提供一套更加简单易用的工具包，应用系统可以更加方便地采用微服务架构模式进行开发及管理。

## 核心架构：



其中，服务容器(UBSI Container)是UBSI平台的核心组件，容器是可以独立运行的节点，用来装载微服务的运行实例。容器为微服务提供了：

- 运行时的上下文环境
- 动态部署、启动、暂停、卸载等生命周期管理
- 向注册中心注册
- 处理数据通讯、并发调度、流量控制
- 向监控工具报告状态及计数
- 动态参数配置

每个容器可以部署多个微服务，同一个微服务可以部署在多个容器中，多个容器构成了UBSI的服务网格(Service Mesh)。

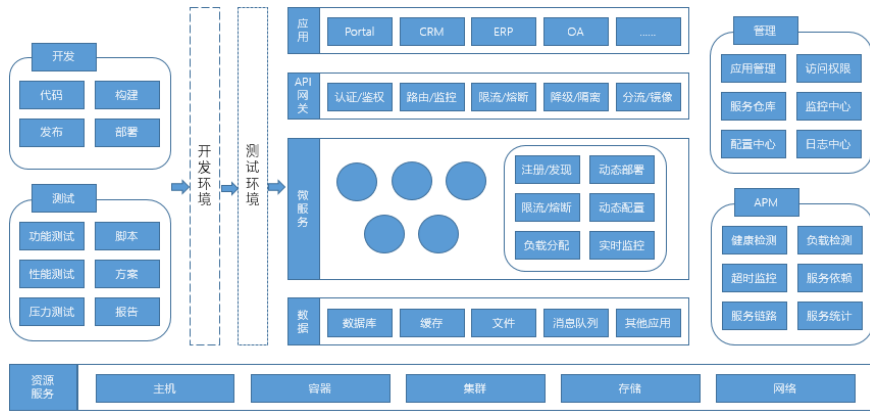
更多说明：

- [注册及发现](#)
- [开发及交付](#)
- [治理工具](#)

## 整体架构：

UBSI可以帮助构建复杂的分布式应用系统或技术/业务中台，满足高性能、易扩展、高可用、易治理等要求，支持快速开发及可持续交付。

UBSI平台的完整体系架构如下：



## 注册及发现

---

UBSI实现了完备的“服务注册/服务发现/服务访问”机制：

- 服务注册

采用redis作为注册中心，UBSI的Container服务容器负责定时将各个服务实例的位置、状态、访问计数等数据更新到redis中，同时利用redis的publish/subscribe机制发布容器的“活动心跳”以及各服务实例的状态变化等消息。

注册中心的高可用性通过redis的群集部署方案来保障。

- 服务发现 / 服务路由

UBSI的Consumer组件通过redis的服务注册数据以及状态消息来构建本地的“动态”服务路由，当有服务请求时，通过路由算法将请求发送到合适的服务实例上。

Consumer组件能够动态跟踪各个服务实例的状态变化，能够发现各服务容器的健康状态以及负载情况，路由算法可以做到隔离故障并且按照“响应能力越强则选中概率越高”的机制选择合适的服务实例。

- 服务访问

UBSI服务请求的处理流程如下：

```
sequenceDiagram
    应用->>Consumer组件: 发起请求
    Consumer组件-->>Container服务容器: 请求数据打包并发送
    Container服务容器->>服务实例: 调用实际的服务接口
    服务实例->>Container服务容器: 接口返回结果
    Container服务容器-->>Consumer组件: 应答数据打包并发送
    Consumer组件->>应用: 返回结果
```

为保证更高的处理性能和数据传输效率，Consumer与Container之间采用基于长链接的多路复用机制，并使用“语言无关”的特定二进制格式对请求/应答数据进行打包编码（[UBSI Protocol](#)）。

---

特别提示：

UBSI支持“简易”的部署模式，不需要redis注册中心也能够正常运行。这种模式下需要手工配置Consumer组件的“静态”路由表。UBSI的静态路由算法同样能够做到：

- 按权重分配负载
- 节点的故障检测、隔离及恢复

# 开发及交付

UBSI力图使微服务的开发更加简单，利用UBSI提供的框架和工具，开发者能够更加专注与微服务本身的功能设计及实现，而不必过多关注底层以及运行/监控需要的技术实现。

## 服务声明

UBSI目前只支持纯Java技术栈（需要JavaSE8+），开发者需要使用UBSI定义的[服务注解](#)来声明自己的微服务及接口，UBSI的服务发现工具能够直接使用注解中的说明形成接口文档：



## 服务仓库

UBSI建议开发者使用maven管理自己的微服务项目，并且将构建后的package发布到自己的仓库repository中。UBSI的服务仓库工具能够从maven仓库中“发现”并管理微服务的各个版本，还能够直接将微服务部署到某个服务容器中。

微服务名	说明	版本	标签	Jar版本/Java类名	操作
rewin.ubs.repo	UBSI软件仓库服务	1.0.0	a	rewin.service.ubs.repo-1.0.0 rewin.service.ubs.repo.Service	更多操作
rewin.ubs.logger	UBSI日志服务（有内部缓冲，部署多实例时需要Redis环境）	1.0.0	1 1	rewin.service.ubs.log-1.0.0 rewin.service.ubs.log.Service	更多操作
rewin.ubs.demo	UBSI示例服务，请求Header的"data_owner"属性用来指定不同的数据分区	1.0.0	b	rewin.service.ubs.demo-1.0.0 rewin.service.ubs.demo.Service	更多操作
icap.task	"检查任务"数据管理服务	1.0.0	icap	icap.service.task-1.0.0 icap.service.task.Service	更多操作
icap.target	"检查对象"数据管理服务	1.0.0	icap	icap.service.target-1.0.0 icap.service.target.Service	更多操作
icap.spec	"规范"数据管理服务	1.0.0	icap	icap.service.spec-1.0.0 icap.service.spec.Service	更多操作
icap.script	"脚本"数据管理服务	1.0.0	icap	icap.service.script-1.0.0 icap.service.script.Service	更多操作
icap.checkitem	"检查项"数据管理服务	1.0.0	icap	icap.service.checkitem-1.0.0 icap.service.checkitem.Service	更多操作

微服务不仅仅是用来满足应用内架构分层或业务解耦的需要，更应该是组织内多个应用之间可以共享的软件资产，所以UBSI的治理工具在关注微服务运行实例的同时，也将微服务作为“资产”进行管理，内容包括微服务的发行包、缺省配置以及接口说明等。

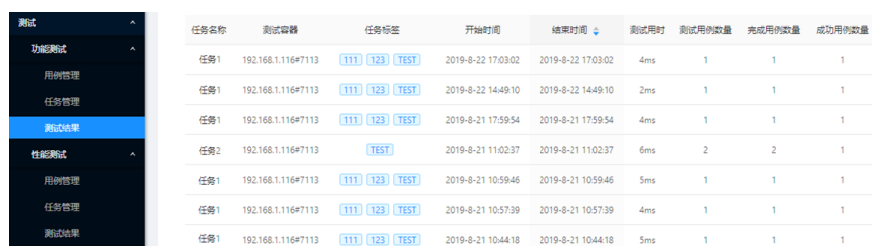
## 灰度发布

微服务在开发迭代过程中会形成多个版本，在运行环境中，可以同时部署微服务不同版本的实例（不能在同一个服务容器中）。UBSI的路由算法可以根据服务名字、版本及发行状态进行路径选择，通过动态配置路由表，可以非常容易地支持在新版本上线时进行“灰度”发布：

- 限定所有流量到旧版本
- 将某个旧版本实例替换部署为新版本
- 对新版本进行自动化测试
- 释放部分流量到新版本，进行在线测试
- 替换其他所有旧版本 或 回退

## 自动化测试

UBSI也提供了一个简单的微服务自动化测试工具，包括功能测试和性能测试：



任务名称	测试容器	任务标签	开始时间	结束时间	测试用时	测试用例数量	完成用例数量	成功用例数量
任务1	192.168.1.116#7113	[111] [123] [TEST]	2019-8-22 17:03:02	2019-8-22 17:03:02	4ms	1	1	1
任务1	192.168.1.116#7113	[111] [123] [TEST]	2019-8-22 14:49:10	2019-8-22 14:49:10	2ms	1	1	1
任务1	192.168.1.116#7113	[111] [123] [TEST]	2019-8-21 17:59:54	2019-8-21 17:59:54	4ms	1	1	1
任务2	192.168.1.116#7113	[TEST]	2019-8-21 11:02:37	2019-8-21 11:02:37	6ms	2	2	1
任务1	192.168.1.116#7113	[111] [123] [TEST]	2019-8-21 10:59:46	2019-8-21 10:59:46	5ms	1	1	1
任务1	192.168.1.116#7113	[111] [123] [TEST]	2019-8-21 10:57:39	2019-8-21 10:57:39	4ms	1	1	1
任务1	192.168.1.116#7113	[111] [123] [TEST]	2019-8-21 10:44:18	2019-8-21 10:44:18	5ms	1	1	1

特别提示：

UBSI目前只支持Java环境，其他异构系统如果需要访问UBSI微服务，可以通过[API网关](#)提供的Restful服务接口进行转发。

# 治理工具

UBSI平台提供了一个功能相对完整的基于web的监控管理工具：

## 运行监控



- "发现"运行的服务容器及微服务
- 容器/微服务的运行状态、请求计数及健康状态
- 正在处理的服务请求，超时警报
- 容器的访问权限、运行参数、服务路由、日志参数的动态配置
- 微服务的接口发现
- 微服务的状态管理：启用/停用、卸载等
- 微服务运行参数、服务依赖的动态配置
- [Web应用](#)的发现及配置

## 服务仓库

- **Java**发行包管理
- 微服务的注册管理
- 微服务接口文档
- 微服务新版本发现
- 微服务部署
- 微服务缺省配置
- 微服务接口仿真

## 服务测试

- 功能测试/性能测试
- 测试脚本



- 测试方案
- 测试结果

## 服务日志

- 服务请求的统计分析
- 服务请求的全链路跟踪分析
- 服务运行日志查询

## API网关的配置

- 应用注册，访问权限
- 路由配置，服务隔离
- 限流/熔断，分流/镜像
- 缓冲/仿真，服务降级
- 请求统计，请求日志

服务仓库

测试

日志

网关管理

网关运行实例

应用管理

远程主机访问权限管理

服务接口访问权限管理

服务路由管理

服务限流管理

服务请求转发管理

服务仿真数据管理

网关分组:

应用:

查询

重置

+ 增加网关

网关分组	应用	服务	接口	最大请求次数	时间(秒)	操作
	111	service1	entry1	10	1	<a href="#">删除</a> <a href="#">修改</a>
	222	service2	entry2	1	10	<a href="#">删除</a> <a href="#">修改</a>
	ubs-gate	service3	entry3	100	1	<a href="#">删除</a> <a href="#">修改</a>
	555	service4	entry4	1	60	<a href="#">删除</a> <a href="#">修改</a>

< 1 > 10 条/页

更多说明请参见：[API网关](#)

## 服务编排及调度

待补充

获得管理工具请参见：[Web管理器](#)

## 基础数据及编码方式

---

UBSI要求微服务接口的请求参数以及返回结果必须采用“标准”的基础数据类型，不能采用“语言相关”的数据结构，这样可以保证在向不同微服务发起请求时，不必依赖服务端定义的数据类型，同时这种机制也为API网关的构造以及以后多语言扩展提供了方便。

**UBSI支持的基础数据类型包括：**

类型	说明	编码	Java类型	XML方式
null	空	0x00	null	<null/>
bool	布尔	.....	boolean   Boolean	<bool>true   false</bool>
byte	单字节	.....	byte	<byte>ff</byte>
int	整数	.....	char   short   int	<int>12</int>
long	长整数	.....	long	<long>123</long>
bigint	大整数	.....	BigInteger	<bigint>12345</bigint>
double	浮点数	.....	float   double	<double>123.45</double>
bignum	大浮点数	.....	BigDecimal	<bignum>123.45</bignum>
string	字符串	.....	String	<str>hello, wor</str>
map	键值	.....	Map	<map><...><...></map>
list	列表	.....	List	<list><...></list>
set	集合	.....	Set	<set><...></set>
bytes	多字节	.....	byte[]	<bytes>a0 b1</bytes>
array	数组	.....	T[]	<array><...></array>

在请求/应答过程中，UBSI框架负责将数据按照对应的数据类型编码成字节数据进行传输，同样，对接收到的字节数据也会进行解码，形成"标准"类型的数据再传递给微服务或应用进行处理。

XML和JSON表达方式可以在[命令行工具](#)或通过[API网关](#)发起服务请求时使用。

通常情况下，UBSI对数据的编/解码过程是透明的，不需要开发者进行处理。但下面几种情况需要Java开发者注意：

- char和short都会转换成int，float会转换为double
- int[]或String[]等非byte[]型的数组，都会转换为Object[]
- 非"标准"数据类型的Object，会将其非static的public成员变量提取出来，转换成Map

# 微服务开发

---

我们通过以下的步骤来开始一个UBSI微服务的开发：

- 配置maven环境及项目依赖
- 了解UBSI的注解
- 一个简单的demo服务
- 访问并测试服务接口
- 如何输出日志
- 服务的部署方式
- 用命令行工具访问

## Maven环境及依赖

开发UBSI微服务需要依赖UBSI的核心包`rewin.ubsi.core`，UBSI的发行包都托管在github packages上，仓库路径是：`ubsi-home/maven`。

UBSI强烈建议在开发环境中使用maven来管理项目并为自己的开发组织配置一个"私有"的maven服务器，然后将UBSI发行包所在的github仓库的URL <https://maven.pkg.github.com/ubsi-home/maven/> 配置到maven私服的资源路径中（同时建议将国内的maven镜像也配置进去，这样可以提高获取"中心"jar包的速度，比如：<http://maven.aliyun.com/nexus/content/groups/public/>）。

简单介绍一下搭建一个Maven私服的步骤：（以docker为例）

- 获取并启动nexus3

```
docker pull sonatype/nexus3
docker run --name nexus3 --restart=always -p 8081:8081 -d so
```

- 通过 <http://192.168.1.116:8081>（假设是在主机192.168.1.116上启动的nexus3）来配置maven的服务资源（例如：最终配置的统一资源路径为 <http://192.168.1.116:8081/repository/maven-public/>），具体过程请自行参考相关教程或文档

然后需要对开发设备上的maven环境进行配置，修改maven的配置文件`settings.xml`，示例如下：

```

<profile>
  <id>dev</id>
  <repositories>
    <repository>
      <id>maven-public</id>
      <url>http://192.168.1.116:8081/repository/maven-
      <releases>
        <enabled>true</enabled>
      </releases>
      <snapshots>
        <enabled>true</enabled>
        <updatePolicy>always</updatePolicy>
      </snapshots>
    </repository>
  </repositories>
  <pluginRepositories>
    <pluginRepository>
      <id>maven-public</id>
      <url>http://192.168.1.116:8081/repository/maven-
      <releases>
        <enabled>true</enabled>
      </releases>
      <snapshots>
        <enabled>true</enabled>
        <updatePolicy>always</updatePolicy>
      </snapshots>
    </pluginRepository>
  </pluginRepositories>
</profile>
</profiles>

<activeProfiles>
  <activeProfile>dev</activeProfile>
</activeProfiles>

```

当maven环境建设完成后，在Java项目的pom.xml中增加下面的依赖，就可以正常使用UBSI的发行包了：

```

<dependency>
  <groupId>rewin.ubsi</groupId>
  <artifactId>rewin.ubsi.core</artifactId>
  <version>1.0.0</version>
</dependency>

```

# 服务注解

UBSI通过一系列预定义的注解来声明微服务及其接口，这些注解包括：

## @UserService

用来标注Java Class，将其声明为一个微服务，例如：

```
package my.service.samples;

import rewin.ubsi.annotation.*;
import rewin.ubsi.container.ServiceContext;

@UserService(
    name = "my.samples.demo",           // 微服务的名字，缺省为""
    tips = "测试服务",                  // 微服务的说明，缺省为""
    version = "1.0.0",                  // 接口的版本号，缺省为"0.0.0"
    release = false,                    // 版本发行状态: true 或 false
    depend = {                           // 依赖的其他微服务
        @USDepend(                       // 第一个依赖，可以有多个
            name = "my.samples.xxx",      // 依赖的微服务的名字
            version = "",                  // 该微服务的最小版本
            release = false                // 该微服务是否必须是"release"
        )
    }
)
public class DemoService {
    // 这是一个UBSI微服务，请注意：
    // class的声明必须是public，并且有无参数的构造函数
}
```

## @USFilter

用来声明一个UBSI微服务的过滤器，除了没有"name"属性，其他定义都跟@UserService相同，示例如下：



```

/** 这是一个UBSI Filter */
@USFilter(
    tips = "这是一个filter"
)
public class DemoFilter {
    @USBefore
    public void before(ServiceContext ctx) throws Exception {
        ctx.getLogger().info("服务容器开始处理一个服务请求");
    }

    @USAfter
    public void after(ServiceContext ctx) throws Exception {
        ctx.getLogger().info("服务容器已经完成一个服务请求的处理");
    }
}

```

`@USFilter`与`@UService`都是由服务容器(Container)加载运行的Class，但与`@UService`不同，`@USFilter`不提供对外的访问接口，而是可以通过`@USBefore`/`@USAfter`定义的入口拦截本容器所有的服务请求，从而可以记录或改变处理行为。

## @USDepend

用在`@UService`/`@USFilter`注解的`depend`属性中，声明依赖的其他微服务，示例请见`@UService`注解。

## @USBefore | @USAfter

可以用在`@UService`或`@USFilter`声明的Class中，定义在一个服务请求"开始"或"结束"时的拦截动作，例如：

```

@UService(
    name = "my.samples.demo"
)
public class DemoService {
    @USBefore(
        timeout = 1          // 超时时间（秒数），缺省为1
    )
    public void before(ServiceContext ctx) throws Exception {
        ctx.getLogger().info("开始处理一个请求");
    }

    @USAfter(
        timeout = 1          // 超时时间（秒数），缺省为1
    )
    public void after(ServiceContext ctx) throws Exception {
        ctx.getLogger().info("完成了一个请求的处理");
    }
}

```

注意：

- 被标注的方法只能有一个 **ServiceContext** 参数，可以通过该参数获得请求的内容或者容器的上下文环境，更多详情可以参见 [ServiceContext的API](#)
- **@UService** 的 **before/after** 只能拦截对本服务的请求，**@USFilter** 可以拦截本容器内所有微服务的请求

## @USEntry

用来声明微服务的接口，例如：

```

@UService(
    name = "my.samples.demo"
)
public class DemoService {
    @UEntry(
        tips = "回显",           // 接口的说明
        params = {
            @USParam(           // 第一个参数，可以有多个
                name = "args",   // 参数的名字
                tips = "参数"     // 参数的说明
            )
        },
        result = "返回传入的参数", // 结果的说明
        readonly = true,           // 是否是"只读"接口，缺省为true
        timeout = 1                // 超时时间（秒数），缺省为1
    )
    public Object echo(ServiceContext ctx, Object args) {
        return args;
    }

    @UEntry(
        tips = "输出一条\"hello world!\"日志"
    )
    public void hello(ServiceContext ctx) {
        ctx.getLogger().info("hello, world!");
    }
}

```

注意：

- 被标注方法的第一个参数必须是**ServiceContext**（通过该参数可以获得请求的上下文），后续可以附加任意数量的参数
- 附加参数的说明应该放在**@UEntry**注解的**params**属性中，按照顺序一一对应，以帮助生成正确的接口文档
- 附加参数以及返回结果的数据类型必须是**UBSI**支持的基础数据类型，详见"[UBSI数据类型](#)"
- **readonly**属性用来声明该接口是否会改变微服务的运行状态或数据，**UBSI**服务容器可以根据这个属性来设置访问权限
- **timeout**属性用来声明该接口"正常"的处理时间，**UBSI**监控工具可以根据这个属性来发现处理超时的服务请求
- 被标注的方法必须是**public**的，并且不能重名
- 在运行时，每次服务请求都会使用一个新的**@UService**实例，所以接口方法不需要考虑并发重入造成的冲突（除非是对静态数据的访问）

## @USParam

用在@USEntry注解的params属性中，声明接口的参数，示例请见@USEntry注解。

## @USInit | @USClose

微服务/过滤器的初始化动作，示例：

```
@UserService(
    name = "my.samples.demo"
)
public class DemoService {
    @USInit
    public static void init(ServiceContext ctx) throws Exception {
        ctx.getLogger().info("微服务启动，进行初始化");
    }

    @USClose
    public static void close(ServiceContext ctx) throws Exception {
        ctx.getLogger().info("微服务关闭，进行清理");
    }
}
```

注意：

- 被标注的必须是public static方法，且只有一个ServiceContext参数
- 当开始加载微服务/过滤器，或者是监控工具"停止 | 启动"服务时，容器会调用这两个方法
- 如果没有必要，可以不使用这两个注解

## @USConfigGet | @USConfigSet

UBSI微服务可以通过这两个注解实现运行时的动态参数配置，示例：

```
/** 返回配置参数 */
@USConfigGet
public static Object getConfig(ServiceContext ctx) throws Exception {
    return "这是配置参数";
}

/** 设置配置参数 */
@USConfigSet
public static void setConfig(ServiceContext ctx, String json) {
    //todo 处理传入的配置参数
}
```

注意：

- 被标注的必须是**public static**方法
- **@USConfigGet**可以返回任意数据结构的配置参数，UBSI配置管理工具会将其转换为json格式的字符串展示给管理员，并将修改后的配置（json格式字符串）传递给**@USConfigSet**进行处理
- 如果需要将配置参数保存为本地的配置文件，**@USConfigSet**可以通过**ServiceContext**提供的API获得本地配置文件的存放路径等环境信息
- 这两个接口不能被外部直接访问，而是通过UBSI容器封装的监控接口来调用
- 如果没有必要，可以不使用这两个注解

## @USInfo

用来向UBSI监控工具报告运行信息的接口，示例：

```
/** 返回运行信息 */
@USInfo
public static Object info(ServiceContext ctx) throws Exception {
    return "当前的运行数据，可以是自定义的数据结构";
}
```

注意：

- 被标注的必须是**public static**方法
- **@USInfo**接口不能被外部直接访问，而是通过UBSI容器封装的监控接口来调用
- 如果没有必要，可以不使用这个注解

## @USNotes | @USNote

如果微服务的输入参数或返回结果需要有特定的结构时，可以通过这两个注解对自定义数据结构进行描述，以方便开发人员对数据进行处理。

示例：

```

@USNotes("访问计数")                // 标注一个数据模型
public static class Counts {
    @USNote("entry1的访问数量")        // 标注模型中的属性
    public long entry1;
    @USNote("entry2的访问数量")
    public long entry2;
}

/* 通过接口返回标注的数据模型，供开发者查看 */
@USEntry(
    tips = "获得数据模型的说明",
    result = "数据模型的说明，格式：{ \"模型1\": { \"字段1\": \"值1\" } }"
)
public Map<String,Map<String,String>> getModels(ServiceContext context) {
    return Util.getUSNotes(Counts.class);    // 提取@USNotes标注的模型
}

```

注意：

- 被标注的必须是**public class**或成员变量
- 如果需要对开发者可见，必须定义一个服务接口来返回标注的模型

## DEMO服务

---

下面是一个完整的"访问计数"的服务示例。

**pom.xml:**

```

<?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 h
  <modelVersion>4.0.0</modelVersion>

  <groupId>my.service</groupId>
  <artifactId>my.service.samples.count</artifactId>
  <version>1.0.0-SNAPSHOT</version>

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

  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <configuration>
          <source>${java.version}</source>
          <target>${java.version}</target>
          <encoding>utf-8</encoding>
        </configuration>
      </plugin>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-surefire-plugin</artifactId>
        <configuration>
          <skipTests>true</skipTests>
        </configuration>
      </plugin>
    </plugins>
  </build>

  <dependencies>
    <dependency>
      <groupId>rewin.ubsi</groupId>
      <artifactId>rewin.ubsi.core</artifactId>
      <version>1.0.0</version>
    </dependency>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.12</version>
      <scope>test</scope>
    </dependency>
  </dependencies>

```



```
<distributionManagement>
  <repository>
    <id>nexus-release</id>
    <name>release</name>
    <url>http://192.168.1.116:8081/repository/maven-rele
  </repository>
  <snapshotRepository>
    <id>nexus-snapshot</id>
    <name>snapshot</name>
    <url>http://192.168.1.116:8081/repository/maven-snap
  </snapshotRepository>
</distributionManagement>

</project>
```

注意：<distributionManagement>用来描述项目build后jar包"发布"的URL地址（注意：这两个URL也应该配置到maven的统一资源路径 - 比如 <http://192.168.1.116:8081/repository/maven-public/> 中），这样后续就可以使用UBSI部署工具来自动部署微服务了。

## CountService.java:

```

package my.service.samples;

import rewin.ubsj.annotation.*;
import rewin.ubsj.common.Codec;
import rewin.ubsj.common.Util;
import rewin.ubsj.container.ServiceContext;

import java.util.Map;
import java.util.concurrent.atomic.AtomicLong;

@UService(
    name = "my.samples.count",
    tips = "访问计数示例",           // 微服务的说明
    version = "1.0.0",               // 接口的版本号
    release = false                   // 版本发行状态
)
public class CountService {

    @USNotes("访问计数")              // 标注一个数据模型
    public static class Counts {
        @USNote("entry1的访问数量")   // 标注模型中的属性
        public long entry1;
        @USNote("entry2的访问数量")
        public long entry2;
    }

    /** 运行信息：返回访问计数 */
    @USInfo
    public static Counts info(ServiceContext ctx) throws Exception {
        Counts counts = new Counts();
        counts.entry1 = entry1Count.get();
        counts.entry2 = entry2Count.get();
        return counts;    // 实际返回的数据会转换为Map
    }

    // 定义静态变量，记录访问次数
    static AtomicLong entry1Count = new AtomicLong(0);
    static AtomicLong entry2Count = new AtomicLong(0);

    @USEntry(
        tips = "获得数据模型的说明",
        result = "数据模型的说明，格式：{ \"模型1\": { \"字段1\": \"值1\" } }"
    )
    public Map<String, Map<String, String>> getModels(ServiceContext ctx) {
        return Util.getUSNotes(Counts.class);    // 提取@USNotes
    }

    @USEntry(

```

```

        tips = "接口方法1"
    )
    public void entry1(ServiceContext ctx) {
        entry1Count.incrementAndGet();
    }

    @USEntry(
        tips = "接口方法2"
    )
    public void entry2(ServiceContext ctx) {
        entry2Count.incrementAndGet();
    }

    @USEntry(
        tips = "在服务端console输出访问计数",
        params = {
            @USParam(
                name = "counts",
                tips = "访问计数的结构：发出请求时可以为Counts对象，
            )
        }
    )
    public void print(ServiceContext ctx, Map counts) {
        // 将Map参数转换为Counts对象
        Counts countsData = Codec.toType(counts, Counts.class);
        System.out.println("=====");
        System.out.println("entry1的访问次数: " + countsData.entry1Count);
        System.out.println("entry2的访问次数: " + countsData.entry2Count);
    }
}

```

最后，用 `mvn clean deploy` 命令来"发布"微服务的jar包。需要注意的是，向maven服务器"发布"jar包，还需要有相应的发布权限，权限的设置有两个步骤：

- 在nexus的管理工具中，创建开发者账号，并对"id"分别为nexus-release和nexus-snapshot的两个repository授予"发布"权限
- 在开发环境的maven配置文件settings.xml中，增加开发者账户，例如：

```
<servers>
  <server>
    <id>nexus-release</id>
    <username>{your account}</username>
    <password>{your password}</password>
  </server>
  <server>
    <id>nexus-snapshot</id>
    <username>{your account}</username>
    <password>{your password}</password>
  </server>
</servers>
```

## 访问并测试服务接口

UBSI的Consumer组件提供了访问微服务的上下文环境以及API接口，这个组件也包含在了UBSI核心包中。

访问微服务接口的简单示例如下：

```
package my.service.samples;

import rewin.ubsj.cli.Request;
import rewin.ubsj.consumer.Context;

public class DemoClient {

    public static void main(String[] args) throws Exception {
        // 初始化UBSI Consumer的Context环境，参数用来指明"工作路径"
        Context.startup(".");

        try {
            // 构造服务请求，指定："服务名字"、"接口名字"、"参数列表"
            Context context = Context.request("ServiceName", "En
            // 将请求发送到微服务所在的服务容器（地址,端口），并获得
            Object res = context.direct("localhost", 7112);
            // 以json方式在console输出结果数据
            Request.printJson(res);
        } catch (Exception e) {
            e.printStackTrace();
        }

        // 关闭UBSI Consumer
        Context.shutdown();
    }
}
```

在实际访问之前，必须先保证微服务已经加载到指定的服务容器中。如何启动一个服务容器并加载微服务请参见[微服务的发布及部署](#)。

但是在微服务开发过程中，如果需要不断启停服务来进行接口调试，会严重影响开发效率，我们可以利用JUnit单元测试工具来更高效地进行服务接口的开发及测试。

以前面的"访问计数"CountService为例，用JUnit4构造一个完整的测试用例：

```

package my.service.samples;

import org.junit.Test;
import org.junit.Before;
import org.junit.After;
import rewin.ubsi.cli.Request;
import rewin.ubsi.common.Codec;
import rewin.ubsi.consumer.Context;
import rewin.ubsi.container.Bootstrap;

import java.util.Map;

public class CountServiceTest {

    @Before
    public void before() throws Exception {
        // 在本机启动一个服务容器，默认服务端口为7112，同时会：
        // 加载rewin.ubsi.module.json配置文件指定的微服务
        // 初始化consumer的Context环境
        Bootstrap.start();
    }

    @After
    public void after() throws Exception {
        // 关闭容器
        Bootstrap.stop();
    }

    static String SERVICE_NAME = "my.samples.count"; // 要访问
    static String SERVER_HOST = "localhost"; // 微服务
    static int SERVER_PORT = 7112; // 微服务

    // 定义在访问端使用的数据结构
    public static class EntryCounts {
        public long entry1; // entry1接口的访问计数
        public long entry2; // entry2接口的访问计数
    }

    // 获取微服务当前的运行状态（访问计数），在console输出并返回
    EntryCounts printCounts() throws Exception {
        // 构造一个"获取指定微服务当前运行状态"的监控请求：
        // 名字为""的微服务是特指"服务容器的控制器"，这是一个容器内
        // "getRuntime"接口表示获取当前运行状态，容器会调用指定微服
        // 参数SERVICE_NAME表示指定的微服务；
        Context ctx = Context.request("", "getRuntime", SERVICE_
        // 将监控请求发送到指定的服务容器，容器会调用微服务的@USInfo
        Map res = (Map)ctx.direct(SERVER_HOST, SERVER_PORT);
        // 将返回结果转换为EntryCounts对象
        EntryCounts counts = Codec.toType(res, EntryCounts.class

```

```

        System.out.println("~~~~~");
        Request.printJson(counts); // 在本地console输出json文本
        return counts;
    }

    // 测试用例
    @Test
    public void testEntry() throws Exception {
        // 在本地输出当前的访问计数
        printCounts();

        for ( int i = 0; i < 3; i ++ ) {
            // 构造对my.samples.count服务的entry1接口的访问请求
            Context context = Context.request(SERVICE_NAME, "ent
            // 将请求发送到指定的服务容器
            context.direct(SERVER_HOST, SERVER_PORT);
        }

        for ( int i = 0; i < 5; i ++ ) {
            // 构造对my.samples.count服务的entry2接口的访问请求
            Context context = Context.request(SERVICE_NAME, "ent
            // 将请求发送到指定的服务容器
            context.direct(SERVER_HOST, SERVER_PORT);
        }

        // 再次输出当前的访问计数
        EntryCounts counts = printCounts();

        // 构造对my.samples.count服务的print接口的访问请求
        Context context = Context.request(SERVICE_NAME, "print",
        // 将请求发送到指定的服务容器：在服务端输出访问计数
        context.direct(SERVER_HOST, SERVER_PORT);
    }
}

```

然后在项目目录（即运行时的"工作目录"）下手工创建配置文件  
**rewin.ubsi.module.json**，用来指定在服务器启动时需要加载的微服  
 务：

```
{
  "services": {
    "my.samples.count": {
      "class_name": "my.service.samples.CountService",
      "startup": true
    }
  }
}
```

用JUnit执行上面的用例，可以得到如下输出：

```
[INFO]      2019-09-03 13:47:32.826      my-pc#7112      rewin.ubsi.co
~~~~~
{
  "entry1": 0,
  "entry2": 0
}
~~~~~
{
  "entry1": 3,
  "entry2": 5
}
=====
entry1的访问次数: 3
entry2的访问次数: 5
[INFO]      2019-09-03 13:47:34.587      my-pc#7112      rewin.ubsi.co
```



注意：

- 两条[INFO]的输出是服务容器"启动"和"关闭"时的提示信息，"my-pc#7112"表示容器所在的主机名字及端口
- 这个测试用例使用了"服务容器控制器"提供的"getRuntime"监控接口，获得微服务的运行信息
- 访问服务接口使用的是direct()"直连"方式，这种方式需要显式指定服务的位置，通常用于测试或监控等特定场景；在配置了"路由"或"注册中心"的运行环境下，应该使用call()"路由"方式，示例：

```
// 构造服务请求
Context context = Context.request("ServiceName", "EntryName")
// 通过路由算法将请求发送到合适的服务容器，并获得返回结果
Object res = context.call();
```

- 服务请求的Context实例对象，在发送完成后应该被丢弃，不可以重复发送

# 统一日志处理

UBSI为微服务的开发提供了一套完整的日志处理框架：

- 核心包的Consumer组件提供了Logger API
- 可以通过配置来指定日志输出的方式，包括：console、log-file、远程的日志服务
- 日志数据的"输出"是独立的后台任务通过异步方式进行批量处理，不影响正常任务的处理效率
- 可以通过独立部署的rewin.ubsi.logger日志微服务，来统一收集其他各处（包括微服务/应用等）产生的日志，并利用UBSI的日志工具进行分析
- 可以通过配置来指定对微服务的请求进行"跟踪"，服务容器/Consumer组件会自动产生相应的请求/处理日志，并利用UBSI的日志工具进行请求链路分析

以服务容器的一条"启动"日志为例（日志的默认配置是输出到console），看一下日志数据的格式：

```
[INFO]      2019-09-03 13:47:32.826      my-pc#7112      rewin.ubsi.co
```

其中：

- [INFO]  
日志类型，其他还可以有：DEBUG(测试)、WARN(警告)、ERROR(错误)、ACTION(操作)、ACCESS(访问)等，应用还可以使用自定义的类型，表示为：[APP#??]
- 2019-09-03 13:47:32.826  
产生日志的时间戳
- my-pc#7112  
输出日志的应用所在的位置
- rewin.ubsi.container  
应用的分类标签
- rewin.ubsi.container  
应用的ID
- rewin.ubsi.container.Bootstrap#start()#138

日志输出语句所在的代码位置（类#方法#行号）

- startup

日志的Tips提示（标题）

- "my-pc#7112"

日志的详细内容（json格式字符串）

对于Consumer应用，可以通过 `Context.getLogger()` 获得Logger对象；对于微服务，应该使用 `ServiceContext`对象的 `getLogger()` 来获得Logger对象。

Logger对象产生的日志可以通过UBSI Web管理器的日志配置工具设置"输出"方式：

日志类型		输出选择	文件名前缀
默认		<input checked="" type="checkbox"/> 控制台 <input type="checkbox"/> 日志服务器 <input checked="" type="checkbox"/> 文件	<input type="text" value="run"/>
debug	<input type="checkbox"/> 关	<input type="checkbox"/> 控制台 <input type="checkbox"/> 日志服务器 <input type="checkbox"/> 文件	<input type="text"/>
info	<input type="checkbox"/> 关	<input type="checkbox"/> 控制台 <input type="checkbox"/> 日志服务器 <input type="checkbox"/> 文件	<input type="text"/>
warn	<input checked="" type="checkbox"/> 开	<input checked="" type="checkbox"/> 控制台 <input type="checkbox"/> 日志服务器 <input checked="" type="checkbox"/> 文件	<input type="text" value="warn"/>
error	<input checked="" type="checkbox"/> 开	<input type="checkbox"/> 控制台 <input checked="" type="checkbox"/> 日志服务器 <input checked="" type="checkbox"/> 文件	<input type="text" value="error"/>
action	<input checked="" type="checkbox"/> 开	<input type="checkbox"/> 控制台 <input checked="" type="checkbox"/> 日志服务器 <input type="checkbox"/> 文件	<input type="text"/>
access	<input checked="" type="checkbox"/> 开	<input type="checkbox"/> 控制台 <input checked="" type="checkbox"/> 日志服务器 <input type="checkbox"/> 文件	<input type="text"/>
app	<input checked="" type="checkbox"/> 开	<input checked="" type="checkbox"/> 控制台 <input type="checkbox"/> 日志服务器 <input checked="" type="checkbox"/> 文件	<input type="text" value="app"/>

- 不需要输出的日志可以通过配置"关闭"即可，不需要再去变更日志生成的代码
- 需要进行统计分析或"跟踪"的日志，可以"输出"到UBSI的日志微服务(`rewin.ubsi.logger`)，然后通过Web管理器的日志工具进行分析

如果未部署Web管理器，也可以手工创建日志配置文件 `rewin.ubsi.log.json`：

```
{
  "options": {
    "all": {
      "output": 3,
      "filename": "run"
    },
    "debug": {
      "output": 0
    },
    "info": {
      "output": 0
    },
    "warn": {
      "output": 3,
      "filename": "warn"
    },
    "error": {
      "output": 6,
      "filename": "error"
    },
    "action": {
      "output": 4
    },
    "access": {
      "output": 4
    },
    "app": {
      "output": 3,
      "filename": "app"
    },
  },
}
```

## 服务的部署方式

微服务可以单独部署，但是不能独立运行，必须要部署到UBSI服务容器中才能运行。具体的部署方式可以有下面几种：

### 手工部署

在没有maven私服、没有治理工具的简单环境下，可以用纯手工方式配置并启动服务容器及微服务：

- 准备一个目录作为UBSI服务容器的工作目录
- 下载UBSI核心jar包：在 <https://ubsi-home.github.io/download> 页面中下载 `rewin.ubsi.core-1.0.0-jar-with-dependencies.jar`
- 准备微服务的jar包，包括第三方依赖
- 手工创建配置文件`rewin.ubsi.module.json`，用来指定需要加载的微服务：

```
{
  "services": {
    "my.samples.count": {
      "class_name": "my.service.samples.CountService",
      "startup": true
    }
  }
}
```

- 启动容器及微服务：

```
java -cp rewin.ubsi.core-1.0.0-jar-with-dependencies.jar:my.
```

### 部署工具

通常情况下，我们会把服务容器的部署跟微服务的部署分开，容器的部署是独立的，可以通过手工或者docker的方式快速部署及启动。

服务容器可以"零配置"启动，不需要任何配置文件，启动后通过管理工具进行监控和配置。执行下面的命令可以手工启动一个"干净"的服务容器：

```
java -jar rewin.ubsi.core-1.0.0-jar-with-dependencies.jar
```

有了"活动"的服务容器，并且已经通过`mvn deploy`命令将微服务的jar包"发布"到了maven服务器，就可以通过UBSI部署工具来部署微服务了。

注意：maven服务器必须正确配置自己的"统一资源路径"，比如<http://192.168.1.116:8081/repository/maven-public/>，以保证部署工具能够通过这个单一地址获得微服务的jar包及其所有依赖。

- 命令行部署工具

UBSI的基础微服务`rewin.ubsi.repo`提供了一个工具，可以帮助开发者通过命令行方式来部署微服务。

- 获取命令行工具

在 <https://ubsi-home.github.io/download> 页面中下载  
`rewin.service.ubsi.repo-1.0.0-jar-with-dependencies.jar`

- 创建部署文件 `deploy.json`

```
{
  "maven_url": "http://192.168.1.116:8081/repository/mav",
  "service_name": "my.samples.count",
  "service_class": "my.service.samples.CountService",
  "jar_group": "my.service",
  "jar_artifact": "my.service.samples.count",
  "jar_version": "1.0.0-SNAPSHOT",
  "config": null
}
```

- 执行部署命令

```
java -jar rewin.service.ubsi.repo-1.0.0-jar-with-depende
```

部署工具会将微服务"my.samples.count"的jar包及其依赖包从maven服务器上download下来，然后上传到指定的服务容器（localhost#7112）中，进行微服务的安装和配置，然后"启动"这个微服务。部署命令的执行结果如下：

```

000.051 开始运行，启动UBSI Consumer
000.502 读取部署文件
000.518 获取最新的JAR包
005.428 查找JAR包的依赖关系
159.947 检查目标容器：localhost#7112
162.866 部署my.service.samples.count : 1.0.0-SNAPSHOT
162.881 上传my.service.samples.count-1.0.0-SNAPSHOT.jar
162.921 注册my.service.samples.count-1.0.0-SNAPSHOT.jar
162.964 添加微服务my.samples.count
163.018 启动微服务my.samples.count
164.840 部署成功！

```

## • Web管理器

UBSI Web管理器的"服务仓库"可以帮助完成微服务的注册、配置以及部署等操作。

微服务名	说明	版本	标签	Jar版本/java类名	操作
rewin.ubsi.repo	UBSI软件仓库服务	1.0.0	<a href="#">a</a>	rewin.service.ubsi.repo-1.0.0 rewin.service.ubsi.repo.Service	更多操作
rewin.ubsi.logger	UBSI日志服务（有内部调用，部署多实例时需要Redis环境）	1.0.0	<a href="#">1</a> <a href="#">b</a>	rewin.service.ubsi.log-1.0.0 rewin.service.ubsi.log.Service	服务注册 修改标签 默认配置 服务依赖 代码部署 部署状态 检查更新
rewin.ubsi.demo	UBSI示例服务，请求Header的"data_owner"属性用来指定不同的数据分区	1.0.0	<a href="#">b</a>	rewin.service.ubsi.demo-1.0.0 rewin.service.ubsi.demo.Service	
icap.task	"检查任务"数据管理服务	1.0.0	<a href="#">icap</a>	icap.service.task-1.0.0 icap.service.task.Service	
icap.target	"检查对象"数据管理服务	1.0.0	<a href="#">icap</a>	icap.service.target-1.0.0 icap.service.target.Service	更多操作
icap.spec	"规范"数据管理服务	1.0.0	<a href="#">icap</a>	icap.service.spec-1.0.0 icap.service.spec.Service	更多操作
icap.script	"脚本"数据管理服务	1.0.0	<a href="#">icap</a>	icap.service.script-1.0.0 icap.service.script.Service	更多操作
icap.checkitem	"检查项"数据管理服务	1.0.0	<a href="#">icap</a>	icap.service.checkitem-1.0.0 icap.service.checkitem.Service	更多操作

## 命令行工具

---

UBSI核心包除了服务容器及Consumer组件之外，还提供了几个常用的命令行工具，可以帮助开发者在未部署UBSI Web管理器的环境下，也能快速查看和访问微服务。

### rewin.ubsi.cli.Request

通过命令行发送一个UBSI服务请求：

```
java -cp rewin.ubsi.core-1.0.0-jar-with-dependencies.jar rewin.u
```

在容器"localhost#7112"的控制台上会看到如下输出：

```
=====  
entry1的访问次数: 1  
entry2的访问次数: 2
```

### rewin.ubsi.cli.Console

命令行交互工具：



```

java -cp rewin.ubsi.core-1.0.0-jar-with-dependencies.jar rewin.u

UBSI Consumer Console v1.0.0, press ENTER for help

localhost#7112>

alone [on|off] - show
async [on|off] - show
call - reque
config [router|log] - show
direct [host [port]] - reque
entry service [entry] - show
event channel data ... - put e
header [key [value]] - show
jedis - show
json - set J
publish channel data ... - publi
register container|restful - show
request service entry ... - send
router service - get s
service - show
statistics - show
subscribe [channel|pattern#channel|event#channel ...] - subsc
time - show
timeout [seconds] - show
tracelog [on|off] - show
unsubscribe [channel|pattern#channel|event#channel ...] - unsub
use service - use s
version [min max release] - show
xml - set X

localhost#7112>

```

比较常用的命令有：

- entry - 查看微服务的接口，例如：

```
localhost#7112> entry my.samples.count

entry1(): 接口方法1

print(): 在服务端console输出访问计数
参数:
    counts: java.util.Map, 访问计数的结构: 发出请求时可以为Counts

getModels(): 获得数据模型的说明
返回:
    java.util.Map<java.lang.String, java.util.Map<java.lang.St

entry2(): 接口方法2

localhost#7112>
```

- request - 访问微服务的接口，例如：

```
localhost#7112> request my.samples.count getModels

{
  "my.service.samples.CountService$Counts: 访问计数": {
    "entry1": "long, entry1的访问数量",
    "entry2": "long, entry2的访问数量"
  }
}

localhost#7112>
```

## rewin.ubsi.cli.Stress

这是一个简单的性能测试工具，执行方式如下：

```
java -cp rewin.ubsi.core-1.0.0-jar-with-dependencies.jar rewin.u
```

参数req.json是数据文件，用来设置需要发送的服务请求，内容如下：

```
{
  "service": "my.samples.count",
  "entry": "getModels"
}
```

参数1000是一个连续发送请求的阈值，具体的工作机制是：

- 利用UBSI Consumer的非阻塞异步请求机制连续发送指定的请求，并实时计算总请求数量和总应答数量，如果发现二者的差值大于指定的阈值，就暂停发送，等待服务端进行处理，直到差值小于阈值再恢复发送

这种简单的流量控制机制是为了防止服务端出现过载拒绝服务的情况，影响性能数据的准确测量。同时，为了配合性能测试，还需要调整服务容器的负载能力参数 - 在容器的工作目录下手工创建配置文件

rewin.ubsi.container.json:

```
{
  "host": "my-pc",
  "port": 7112,
  "backlog": 128,
  "io_threads": 4,
  "work_threads": 20,
  "overload": 1000,
  "forward": 0
}
```

其中:

- host | port  
容器所在服务器的访问地址（建议使用DNS域名，不建议使用IP）和端口
- backlog  
建立socket连接的等待队列长度
- io\_threads  
用来处理socket I/O的线程数，0表示默认设置
- work\_threads  
处理服务请求的线程数量（注：并不是线程数越多并发处理能力就越强，请根据可用的CPU核数合理配置）
- overload  
等待处理的请求队列长度，如果等待队列已满，新的请求会被拒绝（注：在配合Stress测试时，这个值应该设置为Stress的阈值）
- forward  
如果请求的微服务不在本容器内，是否允许容器转发这个请求（0表示不转发）

注：手工修改配置后需要重启服务容器才能生效，建议使用UBSI Web管理器对容器节点进行管理，可以实现动态参数配置

另外，**Stress**工具使用了"路由"方式来访问微服务，与"直连"方式不同，"路由"方式可以保持socket长连接，并利用多路复用机制提高通讯效率，而"直连"方式每次请求都会单独建立一个socket连接，请求完成后关闭，这种方式效率较低，只建议用在特定的"测试"或"监控"场景。

通常情况下，"路由"方式应该配置"注册中心"，这样Consumer组件会自动发现可被访问的微服务实例，这种方式需要部署redis server；如果在一个"简单"的环境中，不需要"注册中心"，也可以通过配置静态路由的方式来指定服务路径 - 手工创建一个配置文件rewin.ubsi.router.json：

```
[
  {
    "Service": "my.samples.count",
    "Nodes": [
      {
        "Host": "my-pc",
        "Port": 7112,
        "Weight": 1
      }
    ]
  }
]
```

其中：

- **Weight** 参数表示容器节点的权重，如果某个服务有多个节点可选，路由算法会根据权重来动态分配请求

静态路由也可以跟"注册中心"提供的动态路由混合使用，在这种情况下，可以通过配置静态路由来实现根据"接口版本/发行状态/可用节点"等对请求进行限定，这种机制通常在"灰度发布"时使用。

OK！现在所有的准备工作完成，重新执行**Stress**测试工具，可以得到如下的结果：

```

java -cp rewin.ubsi.core-1.0.0-jar-with-dependencies.jar rewin.u

my.samples.count:getModels(): {
  "my.service.samples.CountService$Counts: 访问计数": {
    "entry1": "long, entry1的访问数量",
    "entry2": "long, entry2的访问数量"
  }
}

start stress testing ...

--- send: 118165, err: 0, ok: 117642, 15596/s

--- send: 85347, err: 0, ok: 84837, 23442/s

--- send: 83707, err: 0, ok: 83852, 24088/s

--- send: 78001, err: 0, ok: 77832, 23075/s

--- send: 70572, err: 0, ok: 70961, 23380/s

--- send: 83001, err: 0, ok: 82115, 23773/s
q
--- send: 59815, err: 0, ok: 59810, 23668/s

stress testing stopped!

main thread over: 581356 / 581356 / 0

```

**Stress**在验证服务请求可以成功返回后，开始连续发送请求并计算每秒的应答数量：

- 每次按"return"，都会显示"本段时间"内发送的请求数(send)、失败的请求数(err)、成功返回的请求数(应答数：ok)、每秒收到的应答数
- 按"q"表示退出测试
- 最后给出"发送"/"成功"/"失败"的请求总数

特别提示：

除非特殊情况，UBSI不建议手工配置任何服务容器/Consumer组件的运行参数，应该部署Web管理器来进行参数配置工作。

## 运行环境部署

在前面的章节中，我们介绍了如何在一个"简单"的没有"注册中心"的环境下，如何部署服务容器及微服务，但是在真正的运行环境中，UBSI建议应该采用部署"注册中心"的方案。在"注册中心"环境下，UBSI的服务动态发现、动态路由、负载分配/容错/恢复等机制才能发挥作用，能够支持服务能力的动态扩展，以适应业务规模的动态变化。

出于对性能、可靠性以及降低系统复杂度（尽量减少依赖）等方面的考虑，UBSI选择redis作为"注册中心"。部署一个redis节点的步骤如下：  
（以docker为例）

```
docker pull redis
docker run --name redis -p 6379:6379 -d redis
```

如果希望保证redis的高可用，可以部署redis多节点群集模式。更多关于redis部署及配置的说明请自行参见相关文档。

有了redis环境，还需要配置UBSI Consumer组件，使其能够访问redis server，并将其作为"注册中心"（需要注意，UBSI的服务容器也是通过Consumer组件来访问redis以完成服务注册）。配置Consumer组件可以有如下方式：

### 手工方式

在UBSI Container或者WebApp（UBSI Consumer应用）的运行目录下，手工创建rewin.ubsi.consumer.json，内容如下：

```
{
  "io_threads": 4,
  "timeout_connect": 5,
  "timeout_request": 10,
  "timeout_reconnect": 600,

  "redis_host": "{redis-server-host}",
  "redis_port": 6379,
  "redis_conn_idle": 8,
  "redis_conn_max": 128,
}
```

其中：

- io\_thread

Consumer组件用来处理socket I/O的线程数，0表示默认设置

- **timeout\_connect**

向服务容器发起socket连接时的缺省超时时间，秒数

- **timeout\_request**

服务请求的缺省超时时间，秒数

- **timeout\_reconnect**

发现容器/redis服务节点失效后，再次重试的时间间隔，秒数

- **redis\_host**

redis服务的地址

- **redis\_port**

redis的服务端口

- **redis\_conn\_idle**

redis连接池的最大空闲数量

- **redis\_conn\_max**

redis连接池的最大数量

注意：上面的配置是针对redis单节点模式，如果部署了多节点的"哨兵"模式，需要设置redis\_master\_name和redis\_sentinel\_addr

如果是服务容器，还需要手工配置rewin.ubsi.container.json，正确设置容器的访问地址，例如：

```
{
  "host": "{container-host}",
  "port": 7112,
  "backlog": 128,
  "io_threads": 4,
  "work_threads": 10,
  "overload": 100,
  "forward": 0
}
```

## Web管理器

UBSI Web管理器提供了Consumer组件的配置工具，通常情况下，Web管理器是通过"注册中心"来发现服务容器以及WebApp的运行实例的，如果这些实例还未配置redis，意味着它们无法在"注册中心"中注册自己，Web管理器也就无法自动发现它们，这时候需要利用Web管理器

的"手工发现"机制，将服务容器/WebApp的"访问地址"手工加入到Web管理器中，然后就可以通过"配置管理"功能完成这些实例的配置工作了。

参数	运行值	配置值	修改值	说明
io_threads	9	9	<input type="text"/>	"I/O线程的数量，0表示\“CPU内核数 * 2\” (重启生效) ”
redis_host	"192.168.1.116"	192.168.1.116	<input type="text"/>	"Redis服务的主机名 (单机模式) (重启生效) ”
redis_port	6379	6379	<input type="text"/>	"Redis服务的端口号 (单机模式) (重启生效) ”
redis_master_name			<input type="text"/>	"Redis哨兵模式的master名字 (重启生效) ”
redis_sentinel_addr			<input type="text"/>	"Redis哨兵模式的节点地址 (多值)，格式: [\"hostport\", ...] (重启生效) ”
redis_password			<input type="text"/>	"Redis服务的访问密码 (重启生效) ”
redis_conn_idle	10	10	<input type="text"/>	"Redis连接池的最大空闲数量 (重启生效) ”

不管是通过手工还是Web管理器完成了Consumer的配置，都需要将运行实例进行重新启动（Web管理器可以"在线"重启服务容器），重启后的运行实例可以通过"注册中心"完成如下的动作：

- 服务容器
  - 定时将自己的运行地址、加载的服务实例以及请求计数等数据刷新到redis
  - 定时广播活动心跳及负载情况
  - 当服务状态发生变化（暂停/关闭/启动）或服务实例发生变化（安装/卸载）时，发出广播通知
- Consumer组件
  - 加载已经注册的容器及其服务实例
  - 接收广播消息，及时更新容器及其服务的运行状态等信息

通过这些机制，Consumer组件可以"维护"一张容器及服务实例的动态路由表，当发生服务请求时，就可以根据服务声明、容器的健康状态以及负载情况进行动态路由计算，选择合适的容器节点发送服务请求。路由算法的基础原则是：当某个服务在多个容器上有可用实例时，"响应"能力越强的容器拥有越高的"选中"概率。



# UBSI Web管理器

---

UBSI Web管理器是一个基于UBSI微服务架构的Web应用（WebApp）。

## Web管理器的构成

UBSI Web管理器由以下几个部分构成：

- Web前端

采用react+antd构建的Web前端应用，提供基于Web的UI交互页面

- Web后端服务

采用SpringBoot构建的HTTP服务，通过restful-api向前端提供json数据以及操作接口，并通过UBSI Consumer访问"基础微服务"完成对业务数据的逻辑操作，并且处理用户认证/鉴权

- 基础微服务

封装了数据存储、数据模型及其操作逻辑的若干基础微服务，供Web后端调用，这些微服务包括：

- rewin.ubsirepo

"服务仓库"管理，可以管理"注册"的微服务，查看服务接口的定义，管理默认配置以及直接部署

```
groupId: rewin.service.ubsirepo
artifactId: rewin.service.ubsirepo
version: 1.0.0
service class: rewin.service.ubsirepo.Service
```

- rewin.ubsilogger

日志服务，负责统一接收/存储日志数据，并提供查询/统计接口

```
groupId: rewin.service.ubsilogger
artifactId: rewin.service.ubsilogger
version: 1.0.0
service class: rewin.service.ubsilogger.Service
```

- rewin.ubsitester

功能/性能测试，可以创建/管理测试的脚本及方案，并且执行这些测试

```
groupId: rewin.service.ubsi
artifactId: rewin.service.ubsi.test
version: 1.0.0
service class: rewin.service.ubsi.test.Service
```

- rewin.ubsi.gateway

API网关的配置管理服务

```
groupId: rewin.service.ubsi
artifactId: rewin.service.ubsi.gateway
version: 1.0.0
service class: rewin.service.ubsi.gateway.Service
```

- rewin.ubsi.scheduler

服务编排及调度服务

```
groupId: rewin.service.ubsi
artifactId: rewin.service.ubsi.schedule
version: 1.0.0
service class: rewin.service.ubsi.schedule.Service
```

- rewin.common.favor

配置管理，可以用来保存各种配置数据

```
groupId: rewin.service.common
artifactId: rewin.service.common.favor
version: 1.0.0
service class: rewin.service.common.favor.Service
```

- rewin.common.user

用户管理，可以用来管理"用户"的数据模型，并提供"角色/权限"等管理机制

```
groupId: rewin.service.common
artifactId: rewin.service.common.user
version: 1.0.0
service class: rewin.service.common.user.Service
```

- rewin.user.auth

用户认证，提供简单的用户密码管理以及验证机制

```
groupId: rewin.service.user
artifactId: rewin.service.user.auth
version: 1.0.0
service class: rewin.service.user.auth.Service
```

注：这些微服务的发行包都已经托管到github的ubsi-home/maven仓库中，也可以在 <https://github.com/orgs/ubsi-home/packages> 页面中直接查看。

- 数据库

UBSI的基础微服务统一采用了MongoDB数据库

## Web管理器的部署

部署一套完整运行环境的步骤如下：

- 部署MongoDB

```
docker pull mongo
docker run --name mongo -p 27017:27017 -d mongo
```

- 部署redis（可选）

```
docker pull redis
docker run --name redis -p 6379:6379 -d redis
```

- 部署一个服务容器

```
java -jar rewin.ubsi.core-1.0.0-jar-with-dependencies.jar
```

注意：如果部署了redis，需要先手工配置rewin.ubsi.consumer.json和rewin.ubsi.container.json，在rewin.ubsi.consumer.json中配置redis访问地址，在rewin.ubsi.container.json中配置容器的访问地址（详细内容请参阅 [部署注册中心](#)）

- 通过命令行部署工具向服务容器部署需要的基础微服务

以rewin.ubsi.repo为例，创建部署文件deploy.json，内容如下：

```
{
  "maven_url": "http://192.168.1.116:8081/repository/maven-p
  "service_name": "rewin.ubsi.repo",
  "service_class": "rewin.service.ubsi.repo.Service",
  "jar_group": "rewin.service.ubsi",
  "jar_artifact": "rewin.service.ubsi.repo",
  "jar_version": "1.0.0",
  "config": {
    "maven_url": "http://192.168.1.116:8081/repository/maven
    "mongo_servers": [
      {
        "server": "{mongo-server-host}",
        "port": 27017
      }
    ]
  }
}
```

注意：其他微服务的config中不需要配置maven\_url，只需要配置mongo\_servers

然后执行部署命令：

```
java -jar rewin.service.ubsi.repo-1.0.0-jar-with-dependencie
```

- 部署Web后端rest服务

1. 在 <https://ubsi-home.github.io/download> 页面中下载 rewin.rest.ubsi.admin-1.0.0.jar
2. 如果配置了redis，则手工创建配置文件 rewin.ubsi.consumer.json和rewin.ubsi.rest.json，其中 rewin.ubsi.rest.json是WebApp的配置，内容如下：

```
{
  "url": "http://{rest-server-host}:{rest-port}"
}
```

url是WebApp中Consumer组件的配置管理服务接口的访问路径，每个WebApp的后端服务实例也会将自己"注册"到redis注册中心，以供Web管理器"发现"并进行"在线"配置，详情请见 [WebApp](#)

3. 如果未配置redis，则手工创建静态路由文件rewin.ubsi.router.json，内容如下：

```
[
  {
    "Service": "rewin.*",
    "Nodes": [
      {
        "Host": "{container-host}",
        "Port": 7112
      }
    ]
  }
]
```

4. 启动Web服务：

```
java -jar rewin.rest.ubsi.admin-1.0.0.jar
```

- 部署Web前端

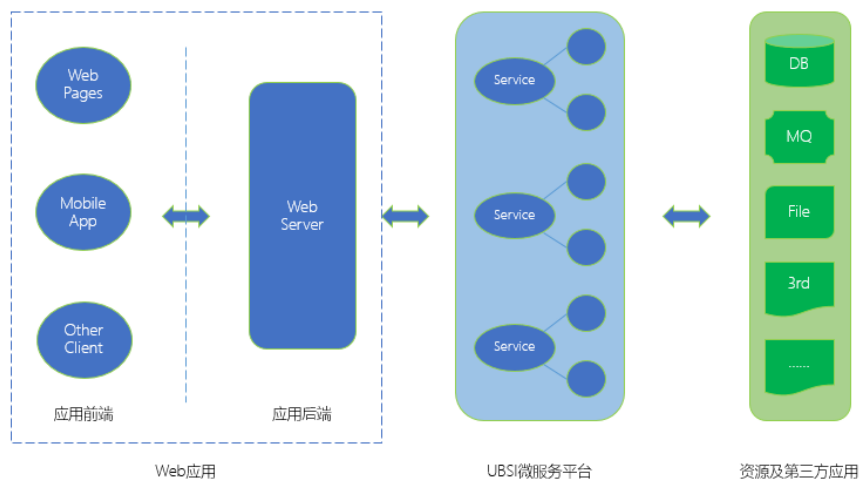
{待补充：部署nginx代理服务器，配置nginx的代理路径，下载前端页面文件}

- 部署完成，Web管理器的URL地址：

<http://{nginx-server-host}:{nginx-port}>

# Web应用

在UBSI微服务架构模式下，应用系统的典型架构模式如下：



在这种多层的分布式架构下，UBSI建议：

- 将核心的业务逻辑封装为"微服务"，微服务的设计原则应该是自治、细粒度、松耦合、无状态的。如果是复杂的业务系统，还可以继续对微服务进行分层，比如基础服务、业务服务、聚合服务等
- 微服务封装业务数据以及资源访问，"前台"的Web应用必须通过服务接口才能访问/操作这些资源
- 微服务管理的业务数据及其操作逻辑是可以"开放"给其他应用使用的，应用之间的互操作或数据共享应该通过微服务层来处理
- UBSI将微服务层之前的面向用户的数据展示及操作部分称为WebApp（"前台"Web应用），又分为两层：
  - 应用前端：向用户展示数据，提供操作功能的UI界面，比如Web页面、移动端APP等
  - 应用后端：负责向应用前端提供数据以及操作接口

通常情况下，Web应用的后端是一个提供restful-api的Web服务：

- 响应前端的"查询"请求：调用"后台"的微服务接口获得数据，对数据进行聚合或格式转换，以json的形式提供给前端
- 响应前端的"操作"请求：调用"后台"的微服务接口进行相应的业务处理。（Web服务本身不应该实现具体的业务逻辑，也不应直接访问业务数据等资源）
- 负责处理用户身份认证及操作鉴权，如果需要的话，还应该处理"统一用户认证"或SSO单点登录等机制
- 负责采集/记录用户行为数据（日志），以供统计分析

## WebApp的开发及治理

WebApp的前端开发不在UBSI的讨论范围之内，可以根据实际情况选择react-js/vue-js、react native/kotlin/swift/flutter等不同的前端框架，在这里我们重点关注后端Web服务的开发。

在Java环境下，UBSI强烈建议采用SpringBoot2框架开发restful风格的Web服务。UBSI为SpringBoot应用提供了rewin.ubsi.rest组件，这个组件通过一组预置的restful-api，使得Web服务能够被UBSI Web管理器进行配置和管理。

如果要使用rewin.ubsi.rest组件，需要在SpringBoot项目的pom.xml中添加如下依赖：

```
<dependency>
  <groupId>rewin.ubsi</groupId>
  <artifactId>rewin.ubsi.rest</artifactId>
  <version>1.0.0</version>
</dependency>
```

注意：

- rewin.ubsi.rest依赖的SpringBoot版本是2.2.1.RELEASE
- rewin.ubsi.rest已经依赖了rewin.ubsi.core，不必重复添加

另外，还需要在SpringBoot项目的主程序入口处添加启动代码，示例如下：

```

@SpringBootApplication(
    scanBasePackages = {
        "rewin.ubsi.rest",
        "{your package}"
    },
    exclude = {
        MongoAutoConfiguration.class,
        MongoDataAutoConfiguration.class
    }    // 取消MongoDB的自动配置（rewin.ubsi.core已经包含了Mongo
)
public class Application {

    public static void main(String[] args) throws Exception {
        SpringApplication.run(Application.class, args);

        // 启动rewin.ubsi.rest，也可以在单独的ApplicationRunner中通过
        rewın.ubsi.rest.Starter.startup(".");    // 参数是项目的运行
    }

}

```

rewin.ubsi.rest在启动时会自动加载UBSI Consumer组件，如果在配置文件rewin.ubsi.consumer.json中配置了redis，则会将自己作为一个WebApp的实例注册到redis注册中心，随后UBSI Web管理器就可以自动发现这个WebApp，并通过rest组件提供的api对其进行配置管理。

rewin.ubsi.rest在向redis注册时，需要声明自己的URL访问路径，这个路径应该配置在rewin.ubsi.rest.json文件中，示例如下：

```

{
    "url": "http://{web-server-host}:{web-server-port}",
    "gateway": true
}

```

注：gateway表示是否允许Web服务直接转发UBSI服务请求

如果Web服务未进行任何手工配置，也可以在启动后，通过UBSI Web管理器的"手工"发现机制，将运行实例的URL手工添加，然后再进行配置或监控。

rewin.ubsi.rest为Web服务提供了两组预置的服务接口，分别是：

- /controller

用于Web管理器进行监控，比如 [GET] <http://{web-server-host}:{web-server-port}/controller/info> 可以获得rewin.ubsi.rest的运行信息



- /request

用于转发UBSI服务请求，可以通过 [POST] <http://{web-server-host}:{web-server-port}/request?help> 得到帮助

# Java API

---

不管是开发UBSI的微服务还是WebApp，都可以利用UBSI核心包提供的API，常用的API包括：

- `rewin.ubsiconsumer` 包
  - [Context](#) - 请求微服务
  - [ErrorCode](#) - 错误代码
  - [Logger](#) - 日志输出
- `rewin.ubsicontainer` 包
  - [ServiceContext](#) - 访问微服务实例的运行环境
- `rewin.ubsicommon` 包
  - [Codec](#) - 处理数据编码解码
  - [JsonCodec](#) - 处理数据的json格式编码
  - [XmlCodec](#) - 处理数据的xml格式编码
  - [Crypto](#) - 国密加解密算法
  - [JedisUtil](#) - 访问redis的工具

另外，UBSI核心包还依赖下面的第三方jar包，这些包提供的API也可以在开发中直接使用：

```
<dependency>
  <groupId>org.dom4j</groupId>
  <artifactId>dom4j</artifactId>
  <version>2.1.1</version>
</dependency>
<dependency>
  <groupId>com.google.code.gson</groupId>
  <artifactId>gson</artifactId>
  <version>2.8.6</version>
</dependency>
<dependency>
  <groupId>io.netty</groupId>
  <artifactId>netty-all</artifactId>
  <version>4.1.43.Final</version>
</dependency>
<dependency>
  <groupId>redis.clients</groupId>
  <artifactId>jedis</artifactId>
  <version>3.1.0</version>
</dependency>
<dependency>
  <groupId>org.mongodb</groupId>
  <artifactId>mongo-java-driver</artifactId>
  <version>3.11.2</version>
</dependency>
<dependency>
  <groupId>org.bouncycastle</groupId>
  <artifactId>bcprov-jdk15on</artifactId>
  <version>1.62</version>
</dependency>
```

## rewin.ubsi.consumer.Context 请求服务

---

### 初始化UBSI Consumer运行环境（静态方法）

```
public static void startup(String workPath) throws Exception;
```

参数:

- **workPath** - 指定工作目录（用来查找配置文件）

返回:

- 无

### 关闭UBSI Consumer运行环境（静态方法）

```
public static void shutdown();
```

参数:

- 无

返回:

- 无

### 设置应用的属性（静态方法）

```
public static void setLogApp(String appAddr, String appTag);
```

参数:

- **appAddr** - 应用的位置
- **appTag** - 应用的分类标签

返回:

- 无

注意:

- UBSI Consumer本身的日志输出信息中会使用这些属性

## 获得应用使用的日志记录器（静态方法）

```
public static Logger getLogger(String appTag, String appID);
```

参数：

- appTag - 应用的分类标签
- appID - 应用ID

返回：

- rewin.ubsi.consumer.Logger对象

注意：

- 如果是在开发微服务，请务必使用ServiceContext的getLogger()来获得Logger对象

## 得到微服务请求的统计数据（静态方法）

```
public static Register.Statistics getStatistics(String service,
```

参数：

- service - 服务名字
- entry - 接口名字

返回：

- rewin.ubsi.consumer.Register.Statistics对象，结构定义如下：

```
/** 请求统计 */
public static class Statistics {
    public long    request;           // 计数器：总请求
    public long    result;            // 计数器：总返回
    public long    success;           // 计数器：总成功
    public long    max_time;          // 计时器：最长的
    public String  req_id;            // 最长处理时间的
}
```

## 得到微服务请求的统计数据（静态方法）

```
public static Map<String, Register.Statistics> getStatistics(Str
```

参数:

- **service** - 服务名字

返回:

- 指定服务的各个接口的分立统计, 格式: {"接口名字": <Statistics> }

## 得到全部微服务请求的统计数据（静态方法）

```
public static Map<String, Map<String, Register.Statistics>> getS
```

参数:

- 无

返回:

- 所有服务的各个接口的分立统计, 格式: {"服务名字": {"接口名字": <Statistics> } }

## 创建微服务请求对象（静态方法）

```
public static Context request(String service, Object... entryAnd
```

参数:

- **service** - 服务名字
- **entryAndParams** - 接口名字及参数列表

返回:

- **rewin.ubsi.consumer.Context**请求对象

注意:

- 如果是在开发微服务, 请务必使用**ServiceContext**的**request()**来创建请求对象

## 获得请求ID

```
public String getReqID();
```

参数:

- 无

返回:

- 请求ID

## 获得请求的服务名字

```
public String getService();
```

参数:

- 无

返回:

- 服务名字

## 获得请求的接口名字

```
public String getEntry();
```

参数:

- 无

返回:

- 接口名字

## 获得请求参数的数量

```
public int getParamCount();
```

参数:

- 无

返回:

- 请求参数的数量

## 获得请求参数的值

```
public Object getParam(int index);
```

参数:

- **index** - 参数的序号, 从0开始

返回:

- 请求参数的值

## 重新设置请求参数

```
public void setParam(Object... o);
```

参数:

- **o** - 请求的参数列表

返回:

- 无

## 获取请求的Header数据项

```
public Object getHeader(String key);
```

参数:

- **key** - 数据项的名字

返回:

- 数据项的值

## 获取请求的Header

```
public Map<String,Object> getHeader();
```

参数:

- 无

返回:

- 请求的Header



## 设置请求Header的数据项

```
public Context setHeader(String key, Object value);
```

参数:

- **key** - 数据项的名字
- **value** - 数据项的值

返回:

- 本对象

## 设置请求Header

```
public Context setHeader(Map<String,Object> header);
```

参数:

- **header** - 请求的Header

返回:

- 本对象

## 设置目标微服务的版本

```
public Context setVersion(int min, int max, int release);
```

参数:

- **min** - 最小版本号, -1表示不改变此项 (缺省为0)
- **max** - 最大版本号, -1表示不改变此项 (缺省为0, 表示不限)
- **release** - 发行状态, 1:必须为Release版; 0:必须为非Release版; -1:不限 (缺省为-1)

返回:

- 本对象

## 设置请求的超时时间

```
public Context setTimeout(int timeout);
```

参数:

- **timeout** - 超时时间，秒数，0表示不限

返回：

- 本对象

## 设置是否使用独立连接发送请求

```
public Context setConnectAlone(boolean alone);
```

参数：

- **alone** - 是否使用独立连接发送请求（缺省为**false**）

返回：

- 本对象

注意：

- **UBSI**底层通讯框架采用了多路复用机制，在"路由"模式下，多个请求会复用同一个**socket**连接。当某个请求需要传递大量数据的时候，会占用通讯链路，影响其他请求的处理。这种情况下，可以 **setConnectAlone(true)**，单独创建**socket**连接来处理这个请求。

## 获取请求的处理时间

```
public long getResultTime();
```

参数：

- 无

返回：

- 请求的处理时间，毫秒数，0表示还未发送请求，<0表示请求还未返回（绝对值表示当前耗时）

## 获取结果代码

```
public int getResultCode();
```

参数：

- 无

返回：

- 结果代码，`ErrorCode.OK`（0）表示成功返回

## 获取结果数据

```
public Object getResultData();
```

参数：

- 无

返回：

- 结果数据，对于`void`类型的服务接口，返回的结果数据为`null`

## 设置处理结果

```
public void setResult(int code, Object data);
```

参数：

- `code` - 结果代码，`ErrorCode.OK`表示处理成功
- `data` - 结果数据

返回：

- 无

注意：

- 此接口通常会在请求过滤器中使用，过滤器的定义如下：

```
/** 请求过滤器 */
public static interface Filter {
    /** 前置接口，返回：0-正常，-1-拒绝，1-降级（使用Mock数据） */
    public int before(Context ctx);
    /** 后置接口 */
    public void after(Context ctx);
}
```

## 向指定的服务容器直接发送请求（同步方式）

```
public Object direct(String host, int port) throws Exception;
```

参数:

- **host** - 容器的主机地址
- **port** - 容器的监听端口

返回:

- 处理结果, 对于**void**类型的服务接口, 返回的结果数据为**null**

注意:

- 直接指定容器发送请求通常用在下面的场景:
  - 对该容器进行配置或监控
  - 测试该容器中部署的微服务实例
- 每次直接发送请求会单独新建**socket**连接, 不会使用"多路复用"机制

## 向指定的服务容器直接发送请求（异步方式）

```
public void directAsync(String host, int port, ResultNotify noti
```

参数:

- **host** - 容器的主机地址
- **port** - 容器的监听端口
- **notify** - 处理结果的监听器, **null**表示不需要容器回传结果
- **message** - 是否通过**redis**广播消息回传结果

返回:

- 无

注意:

- 异步方式在请求发送后立即返回, 不会阻塞等待, 结果数据通过监听器获取, 监听器的定义如下:

```
/** 异步方式得到请求结果的回调接口 */
public static interface ResultNotify {
    /**
     * 回调入口, code:结果代码(ErrorCode.OK表示成功), resu
     * 注: 如果需要进行高耗时的操作, 应启动另外的任务线程进行
     */
    public void callback(int code, Object result);
}
```

## "路由"方式发送服务请求（同步方式）

```
public Object call() throws Exception;
```

参数：

- 无

返回：

- 处理结果，对于void类型的服务接口，返回的结果数据为null

注意：

- "多路复用"、"动态路由" 机制的请求方式，正常情况下应该使用这种方式请求微服务

## "路由"方式发送服务请求（异步方式）

```
public void callAsync(ResultNotify notify, boolean message) thro
```

参数：

- notify - 处理结果的监听器，null表示不需要容器回传结果
- message - 是否通过redis广播消息回传结果

返回：

- 无

注意：

- 当请求需要服务端进行长时间处理时（例如批处理任务），可以考虑采用message为true的请求方式

## rewin.ubsi.consumer.ErrorCode 错误代码

```
public final static int OK = 0;           // 处理成功
public final static int OVERLOAD = 1;     // 过载
public final static int SHUTDOWN = 2;     // 正在关闭
public final static int NOSERVICE = 3;    // 服务未发现
public final static int NOENTRY = 4;      // 接口未发现
public final static int STOP = 5;         // 服务已停止
public final static int REJECT = 6;       // 没有权限
public final static int EXCEPTION = 7;    // 处理异常
public final static int FORWARD = 8;     // 转发异常
public final static int BREAK = 9;        // 接口超时，熔断
public final static int ERROR = 100;      // 自定义错误

public final static int REQUEST = -1;     // 请求参数异常
public final static int CONNECT = -2;     // 连接异常
public final static int CHANNEL = -3;     // Socket通讯异常
public final static int TIMEOUT = -4;     // 请求超时
public final static int ROUTER = -5;      // 路由失败
public final static int MESSAGE = -6;     // 消息机制无效
public final static int MOCK = -7;        // 仿真数据无效
public final static int FILTER = -8;      // 请求过滤器拦截
public final static int REPEAT = -8;      // 请求重复发送
```

## rewin.ubsi.consumer.Logger 日志

---

```
/** 输出DEBUG日志 */
public void debug(String tips, Object data);

/** 输出INFO日志 */
public void info(String tips, Object data);

/** 输出WARN日志 */
public void warn(String tips, Object data);

/** 输出ERROR日志 */
public void error(String tips, Object data);

/** 输出ACTION日志 */
public void action(String tips, Object data);

/** 输出ACCESS日志 */
public void access(String tips, Object data);

/** 输出APP自定义级别日志 */
public void log(int type, String tips, Object data);
```

# rewin.ubsi.container.ServiceContext

## 微服务环境

---

### 构造函数

```
public ServiceContext(String name);
```

参数:

- **name** - 微服务名字

返回:

- 无

### 获得服务的配置文件的存放目录

```
public String getLocalPath();
```

参数:

- 无

返回:

- 微服务配置文件所在的本地路径

### 获得访问者的网络地址

```
public InetSocketAddress getConsumerAddress();
```

参数:

- 无

返回:

- 访问者的网络地址

### 获得请求ID

```
public String getRequestID();
```



参数:

- 无

返回:

- 请求ID

## 获取请求的Header数据项

```
public Object getHeader(String key);
```

参数:

- **key** - 数据项的名字

返回:

- 数据项的值

## 获取请求的Header

```
public Map<String,Object> getHeader();
```

参数:

- 无

返回:

- 请求的Header

## 修改请求Header的数据项

```
public void setHeader(String key, Object value);
```

参数:

- **key** - 数据项的名字
- **value** - 数据项的值

返回:

- 无

## 获得服务名字

```
public String getServiceName();
```

参数:

- 无

返回:

- 微服务实际部署的服务名字

注意:

- 微服务在部署时可以指定服务名字，未必使用@UService中声明的名字，这种机制使得同一个微服务的class可以部署为不同的名字。

## 获得服务的状态

```
public int getServiceStatus();
```

参数:

- 无

返回:

- 服务的运行状态，0:未启动，1:运行中，-1:暂停

## 获得请求的接口名字

```
public String getEntryName();
```

参数:

- 无

返回:

- 请求的接口名字

## 获得请求接口的注解

```
public UEntry getEntryAnnotation();
```

参数:

- 无

返回：

- 请求接口的注解

## 获得服务容器的版本

```
public int getContainerVersion();
```

参数：

- 无

返回：

- 服务容器的版本，格式："1.2.3" => 1002003，版本分3段，每段占3个整数位（取值：0~999）

## 获得服务容器的发行状态

```
public boolean getContainerRelease();
```

参数：

- 无

返回：

- 服务容器的发行状态：是否Release

## 获得请求参数的数量

```
public int getParamCount();
```

参数：

- 无

返回：

- 请求参数的数量

## 获得请求参数的值

```
public Object getParam(int index);
```

参数：

- **index** - 参数的序号，从0开始

返回：

- 请求参数的值

## 重新设置请求参数

```
public void setParam(Object... o);
```

参数：

- **o** - 请求的参数列表

返回：

- 无

## 获得请求的标志

```
public byte getRequestFlag();
```

参数：

- 无

返回：

- 按位表示的请求标志，**0x01**：是否丢弃处理结果，**0x02**：是否通过广播消息返回结果，**0x80**：是否产生请求的跟踪日志

## 暂停服务

```
public void pause();
```

参数：

- 无

返回：

- 无

## 重启服务

```
public void restart();
```

参数:

- 无

返回:

- 无

## 获得处理数量

```
public long[] getStatistics(String service, String entry);
```

参数:

- **service** - 服务名字
- **entry** - 接口名字

返回:

- 处理数量, 格式: [ 处理的请求总量, 处理失败的总量 ]

注意:

- 数量中不包含"待处理"或"处理中"的请求

## 获得各个接口的处理数量

```
public Map<String, long[]> getStatistics(String service);
```

参数:

- **service** - 服务名字

返回:

- 各接口的处理数量, 格式: { "接口名字": [ 处理的请求总量, 处理失败的总量 ] }

## 获得所有服务的处理数量

```
public Map<String, Map<String, long[]>> getStatistics();
```

参数:

- 无

返回:

- 所有服务的处理数量，格式：{"服务名字":{"接口名字":[ 处理的请求总量，处理失败的总量 ]}}

## 设置成功的处理结果

```
public void setResultData(Object data);
```

参数：

- **data** - 处理结果

返回：

- 无

注意：

- 此接口通常会在容器过滤器@USFilter中使用，详见 [@USFilter](#)的说明

## 设置错误结果

```
public void setResultError(String msg);
```

参数：

- **msg** - 错误信息

返回：

- 无

## 是否有处理结果

```
public boolean hasResult();
```

参数：

- 无

返回：

- 是否已经处理完成 或 设置了处理结果

## 获得结果代码

```
public int getResultCode();
```

参数:

- 无

返回:

- 结果代码, `ErrorCode.OK (0)` 表示成功

## 获得结果数据

```
public Object getResultData();
```

参数:

- 无

返回:

- 结果数据, 对于void类型的服务接口, 返回的结果数据为null

## 请求是否被转发处理

```
public boolean isForwarded();
```

参数:

- 无

返回:

- 是否被转发

## 创建请求对象

```
public Context request(String service, Object... entryAndParams)
```

参数:

- `service` - 服务名字
- `entryAndParams` - 接口名字及请求参数

返回:

- `rewin.ubsi.consumer.Context`对象

注意：

- 在微服务中应该使用ServiceContext的request()而不是Context.request()来创建请求对象，这样可以保证：
  - 正确的服务依赖
  - 持续的请求链路跟踪

## 获得日志对象

```
public Logger getLogger();
```

参数：

- 无

返回：

- rewin.ubsi.consumer.Logger对象



## rewin.ubsi.common.Codec 数据编码

### 将Java数据转换为UBSI基础数据对象

```
public static Object toObject(Object value);
```

参数:

- value - Java数据对象

返回:

- [UBSI基础数据对象](#)

### 将数据对象转换为指定的数据类型

```
public static <T> T toType(Object obj, Type type, Type... typeAr
```

参数:

- obj - Java数据对象
- type - 目标数据类型
- typeArguments - 如果目标数据类型是"泛型", 指明泛型需要的数据类型

返回:

- 指定数据类型的对象

示例:

```
List<Integer> value = Codec.toType(new Object[] { 1, 2, 3 }, Arr
```

### 将数据对象编码为base64编码的字符串

```
public static String encode(Object data);
```

参数:

- data - Java数据对象

返回:

- base64编码的字符串

## 将base64编码的字符串解码为数据对象

```
public static Object decode(String data) throws Exception;
```

参数:

- data - base64编码的字符串

返回:

- Java数据对象

## 将数据对象编码为字节数据

```
public static byte[] encodeBytes(Object data);
```

参数:

- data - Java数据对象

返回:

- 字节数据

## 将字节数据解码为数据对象

```
public static Object decodeBytes(byte[] data) throws Exception;
```

参数:

- data - 字节数据

返回:

- Java数据对象

## rewin.ubsi.common.JsonCodec 数据编码json

---

### 将UBSI格式的json字符串解码为Java数据对象

```
public static Object fromJson(String str) throws Exception;
```

参数:

- str - UBSI格式的json字符串, 详见 [UBSI数据编码](#)

返回:

- Java数据对象

### 将Java数据对象编码为UBSI格式的JsonElement

```
public static JsonElement toJson(Object obj) throws Exception;
```

参数:

- obj - Java数据对象

返回:

- com.google.gson.JsonElement对象

## rewin.ubsi.common.XmlCodec 数据编码xml

---

### 将UBSI格式的xml字符串解码为Java数据对象

```
public static Object decode(String str) throws Exception;
```

参数:

- str - UBSI格式的xml字符串, 详见 [UBSI数据编码](#)

返回:

- Java数据对象

### 将Java数据对象编码为UBSI格式的xml字符串

```
public static String encode(Object obj, boolean strCData, boolean
```

参数:

- obj - Java数据对象
- strCData - 是否将String内容放在<![CDATA[...]]>中
- filterHeader - 是否滤掉<?xml version="1.0" encoding="UTF-8"?>

返回:

- UBSI格式的xml字符串

## rewin.ubsi.common.Crypto 国密算法

---

### SM3数据散列

```
public static byte[] sm3Digest(byte[] data);
```

参数:

- data - 源数据

返回:

- 散列值, 长度32字节

### HMAC数据签名

```
public static byte[] sm3HMAC(byte[] data, byte[] key);
```

参数:

- data - 源数据
- key - 密钥, 长度不限

返回:

- 签名, 长度32字节

### 计算SM4加密数据的长度

```
public static int sm4EncryptSize(int size);
```

参数:

- size - 源数据的长度

返回:

- 加密数据的长度

### SM4数据加密

```
public static byte[] sm4EncryptEcb(byte[] data, byte[] key);
```

参数:

- **data** - 源数据, 以128位 (16字节) 为一组, 会自动补位
- **key** - 密钥, 长度必须16字节

返回:

- 加密数据

## SM4数据解密

```
public static byte[] sm4DecryptEcb(byte[] data, byte[] key);
```

参数:

- **data** - 加密数据, 长度必须为16字节的整倍数
- **key** - 密钥, 长度必须16字节

返回:

- 解密数据

## rewin.ubsi.common.JedisUtil 访问 redis

---

### 检测redis是否可用

```
public static boolean isInitd();
```

参数:

- 无

返回:

- 当前是否已经正常连接到了redis server

### 获取一个Jedis实例

```
public static Jedis getJedis();
```

参数:

- 无

返回:

- `redis.clients.jedis.Jedis`对象, 后续可以使用这个对象访问redis; 如果失败会抛出异常

注意:

- `Jedis`对象使用结束后, 需要主动调用 `close()` 以释放资源, 或者使用 `try-with-resource` 机制
- 如果操作中会切换redis的Database, 需要在完成操作后切换回 `JedisUtil.DATABASE`

示例:

```
try (Jedis jedis = JedisUtil.getJedis()) {  
    //do your work  
    jedis.select(JedisUtil.DATABASE);    // 如果之前执行过select操  
}
```

## 发送一个广播消息

```
public static void publish(String channel, Object data);
```

参数:

- **channel** - 广播频道
- **data** - 消息数据

返回:

- 无

注意:

- 所有"订阅"了该频道的"监听器"都会收到这个广播

## 发送一个事件

```
public static void putEvent(String channel, Object data);
```

参数:

- **channel** - 事件频道
- **data** - 事件数据

返回:

- 无

注意:

- 在分布式环境下，即便存在多个订阅了该频道的监听器，但只有一个能收到这个事件（抢占式处理）

## 频道监听器



```

public static abstract class Listener {

    /** 广播监听的回调入口 */
    public abstract void onMessage(String channel, Object message);

    /** 事件监听的回调入口 */
    public abstract void onEvent(String channel, Object event);

    /** 监听广播频道 */
    public void subscribe(String... channels);

    /** 监听广播频道（"*"表示通配符） */
    public void subscribePattern(String... patterns);

    /** 监听事件频道（在一个Java进程中，同一个事件频道只能存在一个） */
    public void subscribeEvent(String... channels) throws Exception;

    /** 取消广播监听 */
    public void unsubscribe(String... channels);

    /** 取消广播监听 */
    public void unsubscribePattern(String... patterns);

    /** 取消事件监听 */
    public void unsubscribeEvent(String... channels);

}

```

注意：

- 如果使用JedisUtil.publish()发布广播或者JedisUtil.putEvent()发送"事件"，必须使用JedisUtil.Listener来监听
- 发送的广播或者事件数据可以是任意数据类型，但是在监听器接收时，会转换为[UBSI基础数据类型](#)
- 在回调接口onMessage或onEvent中，如果需要进行高耗时的操作，应启动另外的任务线程进行处理，以免阻塞异步I/O

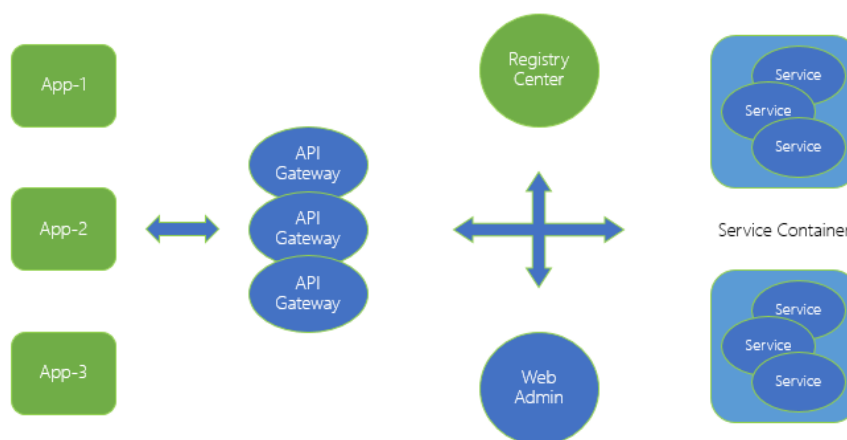
示例：

```
if ( JedisUtil.isInited() ) {  
    // 创建频道监听器  
    JedisUtil.Listener listener = new JedisUtil.Listener() {  
        @Override  
        public void onMessage(String channel, Object message)  
            System.out.print("\n~~~ receive message from [" +  
                Request.printJson(message);  
        }  
  
        @Override  
        public void onEvent(String channel, Object event) {  
            System.out.print("\n~~~ receive event from [" +  
                Request.printJson(event);  
        }  
    }  
  
    listener.subscribe("msg1", "msg2");           // 监听广播频  
    listener.subscribeEvent("evt1", "evt2");       // 监听事件频  
}
```

# API网关

在复杂的企业IT系统环境中，各种不同的业务系统（应用）可能是用不同的技术方案（异构）独立建设的，但是由"微服务"所代表的核心业务逻辑在应用之间应该是一致并且可以被"共享"的。在这种环境下，"微服务"已经不再仅仅是单一应用内的架构模式，而是可以成为企业内保障关键业务能力的"中台"系统。

UBSI API Gateway为企业内的各种异构应用访问"微服务"提供了"统一"的接口及管控机制，逻辑架构如下：



API Gateway本身是一个基于SpringBoot的Web服务，向外围应用提供restful风格的api，使其能够访问"后端"的微服务。API Gateway支持：

- 应用的认证/鉴权
- 多实例部署
- 集中策略配置
- 实例可以分组，并分别配置不同的访问策略

UBSI Web管理器通过基础微服务"rewin.ubsi.gateway"来配置API Gateway的访问策略，rewin.ubsi.gateway将策略存储在MongoDB中，并在策略发生变化时，通过redis广播机制通知各个API Gateway的运行实例。

API Gateway支持的访问策略包括：

- 按照应用/网络地址进行访问权限控制
- 配置访问路由，实现服务隔离、版本选择等
- 对服务进行访问流量控制
- 将访问流量"分流"或"镜像"到其他API Gateway
- 对指定的服务接口进行结果缓冲或仿真，实现服务降级

另外，通过UBSI Web管理器还可以做到：

- 对外围应用进行注册
- 对API Gateway的运行实例进行实时监控
- 对特定服务的请求记录访问日志
- 对访问日志进行统计分析

服务仓库

测试

日志

网关管理

网关运行实例

应用管理

远程主机访问权限管理

服务接口访问权限管理

服务路由管理

服务限流管理

服务请求转发管理

服务仿真数据管理

网关分组:

主机地址:

查询

网关分组	主机地址	监听端口	活动时间戳	启动时间戳	操作
ubsi-gate	192.168.1.116	8080	2020-03-19 12:33:55	2020-03-19 10:27:45	删除 请求统计

<

1

>

10 条/页

## 部署API Gateway

UBSI API Gateway可以独立部署，但是在运行时需要微服务 `rewin.ubsi.gateway` 提供配置数据，这个微服务通常是在部署UBSI Web 管理器时一起部署的。

假设我们的环境中已经部署了redis注册中心以及Web管理器，可以继续部署API Gateway，步骤如下：

- 下载jar包

在 <https://ubsi-home.github.io/download> 页面中下载 `rewin.rest.ubsi.gateway-1.0.0.jar`

- 运行配置

API Gateway缺省的配置参数 `application.properties` 如下：

```
### 初始的URL路径(SpringBoot2)
server.servlet.context-path=/

spring.mvc.view.prefix=/
spring.mvc.view.suffix=.html

### Web Server监听端口
server.port=8080

### 网关实例的分组
ug.group=ubsi-gate
### 应用令牌的过期时间(分钟数)
ug.token-expire=720
### 默认的远程主机访问权限
ug.acl.remote=false
### 默认的服务访问权限
ug.acl.service=false
### 是否验证Token的合法性，false表示Token的值就是AppID
ug.token.check=true
```

另外，还需要在 `rewin.ubsi.consumer.json` 中配置redis的访问地址：

```
{
  "redis_host": "{redis-server-host}",
  "redis_port": 6379
}
```

- 启动API Gateway

```
java -jar rewin.rest.ubsj.gateway-1.0.0.jar
```

部署完成后，访问 <http://{api-gateway-host}:8080/swagger-ui.html> 可以查看restful-api的接口文档。