# **IoT开源平台Thingsboard二次开发研究**



[Echo侯](https://me.csdn.net/houpanqi" \t "https://blog.csdn.net/houpanqi/article/details/_blank) 2019-10-25 14:27:35 IMG_257 8165 IMG_258 收藏 20

分类专栏： [IoT](https://blog.csdn.net/houpanqi/category_9457776.html" \t "https://blog.csdn.net/houpanqi/article/details/_blank) 文章标签： [Iot](https://www.csdn.net/gather_24/NtTagg2sNDQyMi1ibG9n.html" \t "https://blog.csdn.net/houpanqi/article/details/_blank) [Thingsboard](https://www.csdn.net/gather_21/MtzaMg0sNzE5OTQtYmxvZwO0O0OO0O0O.html" \t "https://blog.csdn.net/houpanqi/article/details/_blank)

版权

# **ThingsBoard 二次开发指南**

### **参考资料**

[TB官方文档（英文）](https://thingsboard.io/docs/)  
[Protobuf的讲解](https://www.ibm.com/developerworks/cn/linux/l-cn-gpb/index.html)

### **概述**

1 整个ThingsBoard的代码中，变量、方法命名规范;

2 变量和方法的标识符，字面意思基本能表达整个方法或变量的功能或目的;

3 TB采用了模块化开发来实现松耦合、高内聚。每个模块之间有一定的依赖;

4 微服务集群的容器使用Docker,容器编排使用k8s;

5 使用了Actors编程模型(AKKA框架)(纯异步)、事件驱动模型;

### **二开团队知识储备**

MQTT协议 & MQTT Broker(重要)

CoAP协议

HTTP协议(除get/post之外的动作)

WebSocket协议

RESTFUL 设计规范

TCP/IP协议簇

IoT Gateway

规则引擎概念与设计

Actors编程模型理论

Google Protocol Buffer 原理与使用

AKKA框架

Netty

kafka

PostgreSQL数据库

Nodejs

Vue SPA套件（UI重制）

## **目录结构**

application: 应用程序主目录，整体项目打包时也是这个Module

src/main/java/org.thingsboard.server: app主目录

actors: actor执行者目录

config: 配置类目录

controller: 控制器

exception: 异常处理

service: 服务目录，app打包后，server要运行的服务的实现都在这里

cluster: 集群服务，包括服务发现、服务实例等

script: 脚本执行服务，主要是规则引擎的执行（JS）

transport: 消息传输服务

rpc: 远程调用服务

utils: 工具类common: 公共工具集、方法集模块

dao-api: 数据访问对象的服务接口，在这里做定义

data: 用于不同数据库之间的中间数据结构

message: 消息服务的实现集，包括集群消息、TB的基础服务消息、系统类消息

queue: 队列服务的实现集，用的Kafka

transport: 消息传输通道服务的实现，定义公共接口，给客户端的transport层使用

coap: CoAP消息传输的实现，包括适配器和客户端

http: HTTP消息传输的实现

mqtt: MQTT消息传输的实现，包括适配器、会话、SSL工具等

util: 工具集（当前仅有一个异步回调工具）dao: 数据访问对象集合，整个应用的DAO层，一般只要dao的都要data，以实现不同数据库数据结构兼容。docker: 集群部署-docker容器解决方案，包含了部署环境和shell脚本

haproxy: HAproxy工具的配置文件存放处

tb-node: 应用程序节点配置存放目录

tb-transports: 消息传输服务的配置文件，包括coap,http,mqttimg: 应用图标存放处k8s: 容器编排工具Kubernetes的配置文件和自动化脚本msa: 微服务半自动构建模块，包含了黑盒测试、js执行引擎等

black-box-tests: 黑盒测试工具

js-excutor: js执行引擎，基于nodejs，用于执行规则引擎中的js代码（沙盒执行）

tb: tb主应用程序配置

tb-node: tb程序节点配置

transport: docker中的消息传输服务的部署配置

web-ui: TB的Web用户界面在docker中部署的配置,自带http servernetty-mqtt: TB的MQTT协议的实现，包括连接、发布、订阅、遗嘱、关闭等rule-engine: TB的规则引擎，TB没有用开源的Drools等工具，而是自己开发了一套规则引擎，主要包含3个组件和1套服务接口

rule-engine-api: 服务接口的实现

rule-engine-components: 规则引擎组件，包含滤波器、动作、数据、脚本、延迟加载、消息、队列、RPC实现、邮件等tools: 工具模块，主要实现了MQTT with SSL客户端和REST客户端。可以测试用。transport: 三种消息传输服务启动器，服务的实现参考common/transport,mqtt协议的实现见netty-mqttui: TB的用户界面，主要使用AngularJS和react组件，单页应用（要替换掉，用Vue）

## **所需IDE插件**

IDEA Plugins

protobuf

lombok

## **打包**

TB将所有模块打包，以application为根模块，打包成多个平台的安装包。

## **运行**

TB以系统服务的方式运行：  
CentOS/Redhat-------thingsboard.rpm  
Debian--------thingsboard.deb  
Windows------thingsboard-windows.zip

### **常见问题**

一、 mvn clean install到Server UI出现npm install无法执行的错误：

结束Java binary进程后，用管理员权限运行Terminal(cmd或powershell或shell等)环境，最好不要更改npm源，

开全局代理，依次用以下方法：

1 更改UI，js-executor，web-ui下的pom.xml，将nodeVersion和npmVersion改成和本机版本一致。

如果上述方法不行

2 删除本机Nodejs，重新执行。

如果还不行

3 跳过无法insall的项目，手动npm install，npm run build。执行逻辑在各自的pom.xml中。

二、IDE打开项目后，出现一大堆Error和Warning提示。

IDE中，安装 protobuf、lombok两个plugins，重启IDE。

# **ThingsBoard 二次开发之源码分析 1- 基础知识**

https://blog.csdn.net/IoTSchool/article/details/108896136?utm\_medium=distribute.pc\_relevant.none-task-blog-BlogCommendFromMachineLearnPai2-3.nonecase&depth\_1-utm\_source=distribute.pc\_relevant.none-task-blog-BlogCommendFromMachineLearnPai2-3.nonecase

### **thingsboard聚集地**

Thingsboard 话题讨论区：https://forum.iotschool.com/topics/node8

欢迎大家加入thingsboard 二次开发讨论群：121202538



# **ThingsBoard源码分析1-基础知识**

## **预备知识**

JDK1.8

注解

Protobuf

Guava

### **JDK1.8**

Lambda表达式

方法引用

方法引用的唯一用途是支持Lambda表达式的简写，调用方法的时候使用::， 对于一些单个参数，可以自动推断；

Consumer

Consumer的作用是给定义一个参数,对其进行(消费)处理,处理的方式可以是任意操作，无返回值；

Predicate

断言接口，根据传入的Lambda表达式返回boolean;

Supplier

根据提供的Lambda表达式返回需要的对象；

Function

函数式编程接口，根据提供的Lambda表达式进行相应的操作并返回结果；

### **注解**

@PostConstruct

@postConstruct 注解并非为spring提供， 该注解用来修饰非静态void()方法，该注解的执行时机为在对象加载完依赖注入后执行，即Constructor > @Autowired > @postConstruct;

@EventListener

由spring提供，spring为我们提供了事件的监听与实现，内部的实现原理是****观察者设计模式****，实现了系统解耦，时间发布者不需要考虑谁在监听，发布者只关心自己消息的发布；

ApplicationReadyEvent

应用已经就绪处理请求，将会发布该事件；（[An ApplicationReadyEvent is sent after any application and command-line runners have been called. It indicates that the application is ready to service requests](https://docs.spring.io/spring-boot/docs/2.1.10.RELEASE/reference/html/boot-features-spring-application.html).）

@Conditional及包括其子注解

spring提供，对满足条件进行注入；

@Builder

lombok插件提供，目的是简化构造者模式的代码。Builder Pattern可轻松创建复杂对象；

###Protobuf

[protocol buffers](https://developers.google.com/protocol-buffers) 是一种语言无关、平台无关、可扩展的序列化结构数据的方法，它可用于（数据）通信协议、数据存储等。相比JSON,XML占用内存小，解析速度更快。在使用过程中，首先创建xx.proto文件，通过protobuf-maven-plugin创建相应的类。

### **Guava**

[Guava](https://github.com/google/guava) 是一组来自谷歌的核心Java库，其中包括新的集合类型、不可变集合、一个图库，以及用于并发、I/O、散列、缓存、原语、字符串等的实用工具。

ListenalbeFuture

ListenalbeFuture是对JDK的future进行增强，可以监听任务的执行状况：

使用MoreExecutors创建线程池

ListeningExecutorService executorService = MoreExecutors.listeningDecorator(Executors.newCachedThreadPool());

* + - 1

提交任务

final ListenableFuture<Integer> listenableFuture = executorService.submit(new Callable<Integer>() {

public Integer call() throws Exception {

System.out.println("call execute..");

TimeUnit.SECONDS.sleep(3);

return 7;

}

});

* + - 1
    - 2
    - 3
    - 4
    - 5
    - 6
    - 7

添加监听任务执行状态①

listenableFuture.addListener(()->{

try {

System.out.println(listenableFuture.get());

} catch (InterruptedException e) {

e.printStackTrace();

} catch (ExecutionException e) {

e.printStackTrace();

}

},executorService);

* + - 1
    - 2
    - 3
    - 4
    - 5
    - 6
    - 7
    - 8
    - 9

添加监听任务执行状态②

Futures.addCallback(listenableFuture, new FutureCallback<Integer>() {

@Override

public void onSuccess(@Nullable Integer integer) {

*//返回future的执行结果*

}

@Override

public void onFailure(Throwable throwable) {

}

}, executorService);

* + - 1
    - 2
    - 3
    - 4
    - 5
    - 6
    - 7
    - 8
    - 9
    - 10

Futures.transform

如果需要对返回值做处理，可以使用Futures.transform方法，它是同步方法，另外还有一个异步方法Futures.transformAsync：

ListenableFuture<String> future = executorService.submit(() -> "hello, future");

ListenableFuture<Integer> future2 = Futures.transform(future2, String::length, executorService);

*//future2返回的是’hello, future‘的长度*

* + 1
  + 2
  + 3

SettableFuture

SettableFuture可以认为是一种异步转同步工具，可以它在指定时间内获取ListenableFuture的计算结果：

SettableFuture<Integer> settableFuture = SettableFuture.create();

ListenableFuture<Integer> future11 = executorService.submit(() -> {

int sum = 5 + 6;

settableFuture.set(sum);

return sum;});*// get设置超时时间*

System.out.println(settableFuture.get(2, TimeUnit.SECONDS));