Master 用户向导

基础知识

介绍

在机器人 Android 系统中，Master 旨在解决如下问题：

- 根据不同的职责分类，将机器人程序组件化
- 为各种组件提供消息代理，达成组件间的通信
- 根据产品政策方针，调度组件间的协作

四大组件

根据不同的职责分类，将机器人程序分为 4 类组件

MasterInteractor

TODO

目前，存在 3 种 MasterInteractor：

- 语音识别 MasterInteractor：语音交互是目前机器人的主要交互方式，该类 MasterInteractor 将用户语音转化 Master 能识别的指令，交由其进一步分发
- 远程操控设备 MasterInteractor：通过网络、蓝牙等方式，远程操控设备可连接到机器人，该类 MasterInteractor 将远程操控设备的指令转化成 Master 能识别的指令，交由其进一步分发
- 有屏机器人 Launcher MasterInteractor：对于有屏幕的机器人，用户可以通过点击 Launcher 中的图标，指定机器人执行任务，该类 MasterInteractor 将用户点击图标选择的任务交友 Master 分发

MasterSkill

TODO

MasterService

TODO

MasterEventReceiver

TODO

组件生命周期

TODO

政策方针

TODO

SDK 集成

准备

从 Master Milestones 中获取 Master APK 与 SDK Jar 的版本信息，选取最新版本（x.y.z），按照以下方式安装 Master APK 以及配置 Master SDK 的 Gradle 依赖：

```
# 安装 Master APK
$ adb install master-vx.y.z.apk
```

注意：

- ***必须让 master-vx.y.z.apk 与接入 Master SDK 的 APK 保持一致的签名***，详见下文“配置 AndroidManifest -> 配置权限”章节

```
// Gradle 4+ 使用 implementation 配置依赖，Gradle < 4 使用 compile 配置依赖

// Master SDK
implementation(group: 'com.ubtrobot.master', name: 'master', version: 'x.y.z')

// 通过 Master 达成的组件间通信是进程间通信，需要传递序列化数据参数
// Master SDK 默认支持 Parcelable 的序列化数据参数 ParcelableParam

// 如果使用 Full 版本的 ProtoBuf 参数，配置 ProtoParam 的依赖
implementation(group: 'com.ubtrobot.master', name: 'protobuf-param', version: 'x.y.z')

// 如果使用 Lite 版本的 ProtoBuf 参数，配置 ProtoLiteParam 的依赖
implementation(group: 'com.ubtrobot.master', name: 'protobuf-lite-param', version: 'x.y.z')

// 如果使用基于 Google GSON 的 JSON 参数，配置 ProtoLiteParam 的依赖
implementation(group: 'com.ubtrobot.master', name: 'gson-param', version: 'x.y.z')

// ❤❤❤ 关于序列化参数如何使用，详见后文“参数对象”章节 ❤❤❤
```

注意：

- *-param-x.y.z.jar 会传递依赖 master-x.y.z.jar，同时依赖 *-param-x.y.z.jar 和 master-x.y.z.jar 时请保持 x.y.z 一致。也可只依赖 *-param-x.y.z.jar，master-x.y.z.jar 由 *-param-x.y.z.jar 传递依赖
- protobuf-param-x.y.z.jar、protobuf-lite-param-x.y.z.jar、gson-param-x.y.z.jar 会分别依赖 protobuf-x.y.z.jar、protobuf-lite-x.y.z.jar、gson-x.y.z.jar，如果 APK 的实现本身也依赖这些 Jar，请注意版本匹配

声明权限

在 AndroidManifest.xml 中声明连接 Master 的权限

```
<manifest>
  <!-- 声明连接 Master 的权限，该权限的保护级别为：signature -->
  <uses-permission android:name="com.ubtrobot.master.permission.MASTER" />
</manifest>
```

初始化

```
// 在 YourApplication.onCreate 中初始化
public class YourApplication extends Application {

    @Override
    public void onCreate() {
        super.onCreate();

        // 初始化
        Master.initialize(this);

        // 初始化后，可获取 Master 单例
        Master master = Master.get();
    }
}
```

配置日志打印

```
public class YourApplication extends Application {

    @Override
    public void onCreate() {
        super.onCreate();
        Master.initialize(this);

        // 默认情况，Master SDK 仅打印警告（Warn）与错误（Error）级别的日志
        // 可配置 LoggerFactory 改变 Master SDK 的日志打印行为
        Master.get().setLoggerFactory(loggerFactory);
    }
}
```

- 自行实现 LoggerFactory

```
public class MstApplicationAndroidApplication extends Application {

    @Override
    public void onCreate() {
        super.onCreate();
        Master.initialize(this);

        // 自行实现 LoggerFactory
        Master.get().setLoggerFactory(new LoggerFactory() {
            @Override
            public Logger getLogger(String tag) {
                return new Logger() {
                    @Override
                    public String tag() {
                        return tag;
                    }

                    @Override
                    public void v(String fmt, Object... args) {
                        // 实现打印 VERBOSE 级别的打印
                    }
                    ...
                    // 其他 DEBUG、INFO、WARN、ERROR 级别的打印接口
                };
            }
        });
    }
}
```

- 采用现成的 LoggerFactory
    - ulog 工程实现了 AndroidLoggerFactory、ElvishewXLogLoggerFactory 等 LoggerFactory，可通过配置 Gradle 依赖引用，详见 ulog 文档
    - 以采用现成的 AndroidLoggerFactory 为例

```
public class MstApplicationAndroidApplication extends Application {

    @Override
    public void onCreate() {
        super.onCreate();
        Master.initialize(this);

        // 配置 ulog-AndroidLoggerFactory Gradle 依赖后，可直接使用 AndroidLoggerFactory
        Master.get().setLoggerFactory(new AndroidLoggerFactory());
    }
}
```

实现 MasterSkill

AndroidManifest 声明 MasterSkill

在 AndroidManifest.xml 中声明 MasterSkill 组件

```xml
<manifest>
  <application>
    <!-- 配置 MasterSkill 实现的 Class -->
    <!-- ❤❤❤ 注：必须配置 android:exported="true" ❤❤❤ -->
    <service android:name=".YourMasterSkill" android:exported="true">
        <!-- 指定 YourMasterSkill 元数据描述文件路径 -->
        <!-- ❤❤❤ 注："master.skill" 表明当前组件是 MasterSkill ❤❤❤ -->
      <meta-data android:name="master.skill" android:resource="@xml/skill_${your_skill_name}" />
    </service>
    ...
  </application>
</manifest>
```

在 xml/skill_`${your_skill_name}` xml 文件中配置 YourMasterSkill 元数据

```xml
<?xml version="1.0" encoding="utf-8"?>
<!-- 配置 MasterSkill 元数据信息 -->
<!-- name: 标识同一个 APK 内 Skill 的唯一性 -->
<!-- label: 可选的 Skill 标签，用于显示 Skill 简称 -->
<!-- icon: 可选的 Skill 图标，用于示意 Skill。未配置则采用默认图标 -->
<!-- description: 可选的 Skill 描述，描述 Skill 的详情-->
<skill
  name="your_skill_name"
  label="@string/your_skill_label"
  icon="@mipmap/skill_icon"
  description="@string/your_skill_description">

  <!-- 配置 YourMasterSkill 能够处理的调用（call） -->
  <!-- path: 标识调用路径，必须整个 APK 内部唯一，也应该避免与其他 APK 的 skill.call.path 相同 -->
  <!-- description: 可选的 call 描述 -->
  <call path="/your-skill-name/foo/bar" description="@string/skill_call_foo_bar_description">
    <!-- call 如果需要支持语音 Interactor，可配置语音意图 -->
    <!-- category="speech" 表示语音意图，目前只支持这一种 category -->
    <intent-filter category="speech">
        <!-- sentence 配置语音命令 -->
        <utterance sentence="@string/foo_bar_command" />
        <utterance sentence="@string/foo_bar_command1" />
        <utterance sentence="@string/foo_bar_command2" />
    </intent-filter>
  </call>

  <!-- 配置其他能够处理的调用（call） -->
  <call path="/your-skill-name/baz/qux" description="@string/skill_call_baz_qux_description" />
  ...
</skill>
```

实现 MasterSkill Class

```java
public class YourMasterSkill extends MasterSkill {

    @Override
    public void onSkillCreate() {
        // 根据需要选择做一些初始化工作
    }

    @Override
    public void onSkillStart() {
        // Skill Start 生命周期回调
        // Start 之后 & Stop 之前，可接收、处理、应答来自 MasterInteractor 的调用
    }

    @Call(path = "/your-skill-name/foo/bar")
    public void onFooBar(Request request, Responder responder) {
        // 处理 "/your-skill-name/foo/bar" 调用
        // request.getParam() 获取参数内容，关于 param 详见“参数对象”章节
        // 处理完毕后，需要通过 responder 应答，关于 responder 详见“调用 -> 应答器”章节
    }

    @Override
    protected void onCall(Request request, Responder responder) {
        // 对于在 xml/skill_${your_skill_name} 中描述的 Call：
        // 如果没有对应 @Call 注解的方法接收调用，统一在此接收、处理和应答
        // ❤❤❤ 建议为每个 Call 编写独立的接收方法，并用 @Call 注解 ❤❤❤
    }

    @Override
    public void onSkillStop() {
        // Skill Stop 生命周期回调
        // 此后，将不会接收到来自 MasterInteractor 的调用
    }

    @Override
    public void onSkillDestroy() {
        // 根据需要选择做一些清理工作
    }
}
```

设置 MasterSkill 内部状态

```
public class YourMasterSkill extends MasterSkill {

    @Override
    public void onSkillStart() {
    }

    private void stateSampleCode() {
        setState(aStateStr); // 设置某个状态
        setState(null); // 设置到初始状态
        String currentState = getState(); // 获取当前状态
    }

    @Override
    protected void onCall(Request request, Responder responder) {
    }

    @Override
    public void onSkillStop() {
    }
}
```

实现 MasterService

AndroidManifest 声明 MasterService

在 AndroidManifest.xml 中声明 MasterService 组件

```
```xml
<manifest>
    <application>
        <!-- 配置 MasterService 实现的 Class -->
        <!-- ❤❤❤ 注：必须配置 android:exported="true" ❤❤❤ -->
        <service android:name=".YourMasterService" android:exported="true">
            <!-- 指定 YourMasterService 元数据描述文件路径 -->
             <!-- ❤❤❤ 注："master.service" 表明当前组件是 MasterService  ❤❤❤ -->
            <meta-data android:name="master.service" android:resource="@xml/service_${your_service_name}" />
        </service>
        ...
    </application>
</manifest>
```
```

在 xml/skill_`${your_service_name}` xml 文件中配置 YourMasterService 元数据

```xml
<?xml version="1.0" encoding="utf-8"?>
<!-- 配置 MasterService 元数据信息 -->
<!-- name: 标识同一个 APK 内 Service 的唯一性 -->
<!-- label: 可选的 Service 标签，用于显示 Service 简称 -->
<!-- description: 可选的 Service 描述，描述 Service 的详情-->
<service
    name="your_service_name"
    label="@string/your_service_label"
    description="@string/your_service_description">

    <!-- 配置 YourMasterService 能够处理的调用（call） -->
    <!-- path: 标识调用路径，必须在 YourMasterService 内部唯一 -->
    <!-- description: 可选的 call 描述 -->
    <call path="/foo/bar" description="@string/service_call_foo_bar_description" />

    <!-- 配置其他能够处理的调用（call） -->
    <call path="/baz/qux" description="@string/service_call_baz_qux_description" />
    ...
</service>
```

实现 MasterService Class

```java
public class YourMasterService extends MasterService {

    @Override
    public void onServiceCreate() {
        // 根据需要选择做一些初始化工作
    }

    @Call(path = "/foo/bar")
    public void onFooBar(Request request, Responder responder) {
        // 处理 "/foo/bar" 调用
        // request.getParam() 获取参数内容，关于 param 详见“参数对象”章节
        // 处理完毕后，需要通过 responder 应答，关于 responder 详见“调用 -> 应答器”章节
    }

    @Override
    protected void onCall(Request request, Responder responder) {
        // 对于在 xml/service_${your_service_name} 中描述的 Call：
        // 如果没有对 @Call 注解的方法i接收调用，统一在此接收、处理和应答
        // ❤❤❤ 建议为每个 Call 编写独立的接收方法，并用 @Call 注解 ❤❤❤
    }

    @Override
    public void onServiceDestroy() {
        // 根据需要选择做一些清理工作
    }
}
```

发布事件

```
public class YourMasterService extends MasterService {

    @Override
    public void onServiceCreate() {
    }

    private void publishEventSampleCode() {
        // 发布事件
        // ❤❤❤ 普通服务发布的事件只能“动态”订阅， ❤❤❤
        // ❤❤❤ 详见后文“MasterContext -> 动态订阅服务事件”章节 ❤❤❤
        publish(action); // 无参情况
        publish(action, param); // 有参情况（关于，详见“参数对象”章节）
    }

    @Override
    protected void onCall(Request request, Responder responder) {
    }

    @Override
    public void onServiceDestroy() {
    }
}
```

实现 MasterSystemService

MasterSystemService 的实现与 MasterService 的实现存在 4 处区别

- 实现 MasterSystemService 必须在 AndroidManifest.xml 中声明 MasterSystemService 权限

```
<manifest>
    <uses-permission android:name="com.ubtrobot.master.permission.MASTER_SYSTEM_SERVICE" />
</manifest>
```

- MasterSystemService 的元数据描述中，需要将服务标识为系统级别

```
<?xml version="1.0" encoding="utf-8"?>
<!-- MasterSystemService 需要将 level 配置为 "system" -->
<!-- level 缺省为 "noraml" -->
<service
    level="system"
    name="your_service_name"
    label="@string/your_service_label"
    description="@string/your_service_description">
    ...
</service>
```

- MasterSystemService 的实现 Class 必须继承自 MasterSystemService

```
public class YourMasterService extends MasterSystemService {
    ...

}
```

- MasterSystemService 支持“谨慎”发布事件

```
public class YourMasterService extends MasterSystemService {

    @Override
    public void onServiceCreate() {
    }

    private void publishEventSampleCode() {
        // 系统服务不仅支持发布能“动态”订阅的事件，还支持能“静态”订阅的事件
        // ❤❤❤ 静态订阅事件，详见后文“实现 StaticEventReceiver”章节 ❤❤❤
        Master.get().publishCarefully(action); // 无参情况
        Master.get().publishCarefully(action, param); // 有参情况（关于 param，详见“参数对象”章节）

       // ❤❤❤ 非重要事件，应避免发布能“静态”订阅的事件， ❤❤❤
       // ❤❤❤ 避免造成“静态”订阅了事件的包不必要的启动 ❤❤❤
    }

    @Override
    protected void onCall(Request request, Responder responder) {
    }

    @Override
    public void onServiceDestroy() {
    }
}
```

实现 StaticEventReceiver

某个服务谨慎发布某个事件后，StaticEventReceiver 能够在其 APK 包未启动的情况下接收到该事件

AndroidManifest 声明 StaticEventReceiver

```
<manifest>
  <application>
    <!-- 配置 StaticEventReceiver 实现的 Class -->
    <service android:name=".YourStaticEventReceiver">
      <intent-filter>
        <!-- 配置 intent.category，常量 -->
        <category android:name="master.intent.category.SUBSCRIBER" />

        <!-- 配置订阅的事件的 Action，可配置 1~N 个。相关的事件配置在同一个接收者中，否则放到其他接收者中 -->
        <action android:name="ubtrobot.event.action.FOO_BAR" />
        <action android:name="ubtrobot.event.action.BAZ_QUX" />
      </intent-filter>
    </service>

    <service android:name=".YourAnotherStaticEventReceiver">
      <!-- 配置形式同上-->
      ...
    </service>
  </application>
</manifest>
```

实现 StaticEventReceiver Class

```
public class YourStaticEventReceiver extends StaticEventReceiver {

  @Subscribe(action = "ubtrobot.event.action.FOO_BAR")
  public void onFooBar(MasterContext context, Event event) {
    // 处理 action 为 "ubtrobot.event.action.FOO_BAR" 的事件
  }

  @Subscribe(action = "ubtrobot.event.action.BAZ_QUX")
  public void onBazQux(MasterContext context, Event event) {
    // 处理 action 为 "ubtrobot.event.action.BAZ_QUX" 的事件
  }

  @Override
  public void onReceive(MasterContext context, Event event) {
    // 处理在 AndroidManifest 中声明但未用 @Subscribe 注解的事件统一在此处理，可通过 event.getAction() 获取哪个事件
  }
}
```

创建 MasterInteractor

```
void interactorSampleCode() {
    // 获取否则创建 MasterInteractor
    MasterInteractor interactor = Master.get().getOrCreateInteractor();

    // 创建 Skills 调用代理
    SkillsProxy skillsProxy = interactor.createSkillsProxy();

    // ❤❤❤ SkillsProxy 支持异步调用、同步调用、意图调用、配置调用 ❤❤❤
    // ❤❤❤ 详见“调用 -> （异步调用 | 同步调用 | 意图调用 | 配置调用）”章节 ❤❤❤
}
```

MasterContext

MasterContext 接口

在很多上下文环境下，能够调用服务以及订阅服务发布的事件，这些操作统一抽象成 MasterContext 接口。详见如下：

```java
public interface MasterContext {

    /**
     * 动态订阅 MasterService 发布的事件
     *
     * @param receiver 事件接收者
     * @param action   事件类型
     */
    void subscribe(EventReceiver receiver, String action);

    /**
     * 取消订阅 MasterService 发布的事件
     *
     * @param receiver 事件接收者
     */
    void unsubscribe(EventReceiver receiver);

    /**
     * 创建 MasterSystemService (系统服务) 的访问代理，从而访问该系统服务
     *
     * @param serviceName 系统服务名称
     * @return 服务访问代理
     */
    ServiceProxy createSystemServiceProxy(String serviceName);

    /**
     * 创建 MasterService (普通服务) 的访问代理，从而访问该服务
     *
     * @param packageName 服务所在包的包名
     * @param serviceName 服务名称
     * @return 服务访问代理
     */
    ServiceProxy createServiceProxy(String packageName, String serviceName);

    /**
     * 启动当前 Package 内的某个服务
     *
     * @param service 服务名称
     */
    void startService(String service);
}
```

多个 MasterContext 实现

- MasterSkill、MasterService、MasterSystemService 与 MasterInteractor 均实现了 MasterContext

```
com.ubtrobot.master.context.MasterContext
    ↳ com.ubtrobot.master.skill.MasterSkill
    ↳ com.ubtrobot.master.service.MasterService
        ↳ com.ubtrobot.master.service.MasterSystemService
    ↳ com.ubtrobot.master.interactor.MasterInteractor
```

- EventReceiver、StaticEventReceiver 在接收到事件时，同时也有上下文传递进来

```
public interface EventReceiver {

    // 对于动态订阅事件，收到事件时传递的上下文是订阅者自身
    // 比如：YourMasterSkill 中订阅事件后，收到事件时传递的上下文是 YourMasterSkill
    void onReceive(MasterContext context, Event event);
}
```

```
public class YourStaticEventReceiver extends StaticEventReceiver {

    @Subscribe(action = "ubtrobot.event.action.FOO_BAR")
    public void onFooBar(MasterContext context, Event event) {
    }

    @Override
    public void onReceive(MasterContext context, Event event) {
        // 对于静态订阅事件，收到事件时传递的上下文是即时创建的
    }
}
```

- 全局上下文

```
void globalContextSampleCode() {
    // 可通过 Master 单例获取全局上下文
    MasterContext globalContext = Master.get().getGlobalContext();
}
```

MasterContext 有效期

MasterContext 有效期与其各个具体实现的对象的生命周期有直接关系，在对象生命周期结束后不应该再调用 MasterContext 中的接口

动态订阅服务事件

以 MasterSkill 这种 MasterContext 为例，其他 MasterContext 类推

```
public class YourMasterSkill extends MasterSkill {

    private static final String EVENT_ACTION = "ubtrobot.event.action.FOO_BAR";

    private final EventReceiver mEventReceiver = new EventReceiver() {
        @Override
        public void onReceive(Event event) {
            // 处理事件
            // 将 event.getParam() 的参数内容提取出来，详见“参数对象”章节
        }
    };

    @Override
    public void onSkillStart() {
        // 动态订阅事件
        subscribe(mEventReceiver, EVENT_ACTION);
    }

    @Override
    protected void onCall(Request request, Responder responder) {
    }

    @Override
    public void onSkillStop() {
        // 取消订阅
        unsubscribe(mEventReceiver);
    }
}
```

启动服务

```
void startMasterServiceSampleCode() {
    // 在某个 MasterContext 下，可以启动 Package 内部的 MasterService 让其执行后台任务
    // 以全局上下文为例
    MasterContext globalContext = Master.get().getGlobalContext();
    globalContext.startService(theServiceName);
}
```

调用服务

在 MasterContext 中，支持通过 ServiceProxy 对象向指定服务发起调用，以 MasterSkill 为这种 MasterContext 例，其他 MasterContext 类推：

```
public class YourMasterSkill extends MasterSkill {

    @Override
    public void onSkillStart() {
    }

    @Override
    protected void onCall(Request request, Responder responder) {
        // 创建其他 Package 的服务调用代理
        ServiceProxy sampleServiceProxy = createServiceProxy(sampleServicePackageName, sampleServiceName);
        // 创建系统服务调用代理
        ServiceProxy sampleSystemServiceProxy = createSystemServiceProxy(sampleSystemServiceName);

        // ❤❤❤ ServiceProxy 支持异步调用、同步调用、异步粘滞调用、配置调用 ❤❤❤
        // ❤❤❤ 详见“调用 -> （异步调用 | 同步调用 | 异步粘滞调用 | 配置调用）”章节 ❤❤❤
    }

    @Override
    public void onSkillStop() {
    }
}
```

调用

异步调用

```
void asyncCallSampleCode() {
    callable = aServiceProxy | aSkillsProxy (二者具备相同的异步调用接口)
    //  callable = aServiceProxy 时，向某个服务发起调用
    //  callable = aSkillsProxy 时，向所有 Skills 发起调用

    Cancelable cancelable = callable.call("/foo/bar", new ResponseCallback() { // 无参情况
        @Override
        public void onResponse(Request req, Response res) {
            // 成功情况，处理响应（res）
            // 将 res.getParam() 的参数内容提取出来，详见“参数对象”章节
        }
        @Override
        public void onFailure(Request req, CallException e) {
            // 出错情况，处理异常（e）
            // e.getCode() 可获取错误码（详见“调用错误码”章节）
            // e.getMessage() 可获取错误消息（对错误码的解释），用于调试
            // e.getDetail() 某些出错情况，可能需要携带数据，数据以 Map<String, String> 形式响应
        }
    });
    // 可使用 cancelable.cancel() 取消调用
    callable.call("/baz/qux", param, callback); // 有参情况（如何构造参数对象，详见“参数对象”章节）
}
```

同步调用

```
void syncCallSampleCode() {
  callable = aServiceProxy | aSkillsProxy (二者具备相同的同步调用接口)
  //  callable = aServiceProxy 时，向某个服务发起调用
  //  callable = aSkillsProxy 时，向所有 Skills 发起调用

  try {
    Response res = callable.call("/foo/bar"); // 无参情况
    // 成功情况，处理响应（res）
    // 将 res.getParam() 的参数内容提取出来，详见“参数对象”章节

    Response anotherRes = callable.call("/baz/qux", param); // 有参情况（关于 param，详见“参数对象”章节）
    // 成功情况，处理响应（anotherRes）
  } catch (CallException e) {
    // 出错情况，处理异常（e）
    // e.getCode() 可获取错误码（详见“调用错误码”章节）
    // e.getMessage() 可获取错误消息（对错误码的解释），用于调试
    // e.getDetail() 某些出错情况，可能需要携带数据，数据以 Map<String, String> 形式响应
  }

  ////////////////////////////////////////////////////////////
  ////////////////          注意          ////////////////
  ////////////////////////////////////////////////////////////
  // call 内部实现是跨进程远程调用，同步调用返回时间没有担保，应避免在主线程执行同步调用
  // 默认情况，主线程执行同步调用，将日志警告（该警告可选择关闭，详见“配置调用”章节）

  // 同步调用的正确使用姿势：在工作线程中组合执行多个同步调用（组合异步调用存在回调嵌套的问题）
}
```

异步粘滞调用

```
void asyncStickyCallSampleCode() {
  ServiceProxy aServiceProxy = aMasterContext.createServiceProxy | aMasterContext.createSystemServiceProxy

  Cancelable cancelable = aServiceProxy.callStickily("/foo/bar", new StickyResponseCallback() { // 无参情况

    @Override
    public void onResponseStickily(Request req, Response res)
      // 中间结果（0 ~ N 次），处理响应（res）
      // 将 res.getParam() 的参数内容提取出来，详见“参数对象”章节
    }

    @Override
    public void onResponseCompletely(Request req, Response res) {
      // 最终结果成功情况（有且只有一次，与 onFailure 互斥），处理响应（res）
      // 将 res.getParam() 的参数内容提取出来，详见“参数对象”章节
    }

    @Override
    public void onFailure(Request req, CallException e) {
      // 最终结果出错情况（有且只有一次，与 onResponseCompletely 互斥），处理异常（e）
      // e.getCode() 可获取错误码（详见“调用错误码”章节）
      // e.getMessage() 可获取错误消息（对错误码的解释），用于调试
      // e.getDetail() 某些出错情况，可能需要携带数据，数据以 Map<String, String> 形式响应
    }
  });
  // 可使用 cancelable.cancel() 取消调用

  // 有参情况（如何构造参数对象，详见“参数对象”章节）
  aServiceProxy.callStickily("/baz/qux", param, anotherCallback);
}
```

意图调用

```
void intentCallSampleCode() {
    SkillsProxy aSkillsProxy = aMasterInteractor.createSkillsProxy();

    SkillIntent skillIntent = new SkillIntent(SkillIntent.CATEGORY_SPEECH);
    skillIntent.setSpeechUtterance("跳个舞");

    // 异步意图调用
    Cancelable cancelable = aSkillsProxy.call(skillIntent, new ResponseCallback() { // 无参情况
        @Override
        public void onResponse(Request req, Response res) {
            // 成功情况，处理响应（res）
            // 将 res.getParam() 的参数内容提取出来，详见“参数对象”章节
        }
        @Override
        public void onFailure(Request req, CallException e) {
            // 出错情况，处理异常（e）
            // e.getCode() 可获取错误码（详见“调用错误码”章节）
            // e.getMessage() 可获取错误消息（对错误码的解释），用于调试
            // e.getDetail() 某些出错情况，可能需要携带数据，数据以 Map<String, String> 形式响应
        }
    });
    // 可使用 cancelable.cancel() 取消调用
    aSkillsProxy.call(skillIntent, param, callback); // 有参情况（如何构造参数对象，详见“参数对象”章节）

    // 同步意图调用
    try {
        Response res = aSkillsProxy.call(skillIntent); // 无参情况
        // 成功情况，处理响应（res）
        // 将 res.getParam() 的参数内容提取出来，详见“参数对象”章节

        Response anotherRes = aSkillsProxy.call(skillIntent, param); // 有参情况（如何构造参数对象，详见“参数对象”章节）
        // 成功情况，处理响应（anotherRes）
    } catch (CallException e) {
        // 出错情况，处理异常（e）
        // e.getCode() 可获取错误码（详见“调用错误码”章节）
        // e.getMessage() 可获取错误消息（对错误码的解释），用于调试
        // e.getDetail() 某些出错情况，可能需要携带数据，数据以 Map<String, String> 形式响应
    }
}
```

配置调用

```
void configCallSampleCode() {
    callable = aServiceProxy | aSkillsProxy (二者具备相同的配置调用接口)
    //  callable = aServiceProxy 时，向某个服务发起调用
    //  callable = aSkillsProxy 时，向所有 Skills 发起调用

    // 构造调用配置
    CallConfiguration config = new CallConfiguration.Builder().
        // 设置异步调用、同步调用的超时时间。单位 ms，必须 > 0，默认 15000
        setTimeout(timeout).
        // 关闭或打开主线程同步调用的警告。true：关闭，false：打开，默认打开
        suppressSyncCallOnMainThreadWarning(true).
        build();
    // 让调用配置生效
    callable.setConfiguration(config);
}
```

应答调用

MasterSkill、MasterService、MasterSystemService 在调用处理完毕后，需要通过 Responder 应答调用

```
@Call(path = "/foo/bar")
void responderSampleCode(Request request, Responder responder) {
    // 成功情况应答
    responder.respondSuccess(); // 无响应参数时
    responder.respondSuccess(param); // 有响应参数时

    // 出错情况应答，错误码详见“调用错误码”章节
    responder.respondFailure(CallGlobalCode.XXX, message); // 通用出错情况
    responder.respondFailure(yourCallPrivateCode, message); // 接口特定的出错情况
    responder.respondFailure(code, message, detail); // 可携带 Map<String, String> 形式的数据给调用者

    // 应答粘滞调用中间结果。对于 MasterSkill，不支持该操作。因为 SkillsProxy 不支持异步粘滞调用
    responder.respondStickily(param);

    // 监听调用者取消调用
    responder.setCallCancelListener(new CallCancelListener() {
        @Override
        public void onCancel(Request request) {
            // 调用者取消了调用
            // 收到该回调后应停止处理过程，也不再需要调用 responder.respondXxx 执行应答
        }
    });
}
```

调用错误码

- 错误码判断
    - CallException.getCode 与下述通用错误码和私有错误码进行数值判断
- 公有错误码：所有调用通用，不偶尔接口功能实现
    - < 1000 的错误码保留给通用错误码使用

```
public final class CallGlobalCode {
    private CallGlobalCode() {
    }
    // 非法请求，参数缺失、格式非法、字段缺失或非法情况
    public static final int BAD_REQUEST = 400;
    // 未授权，暂未用到，预留
    public static final int UNAUTHORIZED = 401;
    // 禁止访问
    public static final int FORBIDDEN = 403;
    // 调用未找到
    public static final int NOT_FOUND = 404;
    // 调用请求超时，暂未用到，预留
    public static final int REQUEST_TIMEOUT = 408;
    // 调用冲突
    public static final int CONFLICT = 409;
    // 调用应答方内部错误
    public static final int INTERNAL_ERROR = 500;
    // 调用未实现
    public static final int NOT_IMPLEMENTED = 501;
    // 调用应答超时
    public static final int RESPOND_TIMEOUT = 504;
}
```

- 私有错误码：某个或某几个接口特有的错误码，与其功能相关
    - 定义时需 > 1000，避免跟通用错误码冲突
    - 不同的模块最好分段错开，比如：模块 A 使用 10XX，模块 B 使用 11XX。

参数对象

Param 基础

```
void paramBaseSampleCode() {
    Param param = event.getParam() | req.getParam() | res.getParam() | *Param.(create|from)
    if (!param.isEmpty()) { // 参数判空
        // 获取参数类型
        String type = param.getType();
        type == "ParcelableParam" | "ProtoParam" | "ProtoLiteParam" | "GsonParam"
        // *Param.TYPE 常量描述了对应的类型
    }
}
```

ParcelableParam

```
// Master SDK 默认支持
void parcelableParamSampleCode() {
    // 创建空的 BundleParam
    BundleParam bundleParam = BundleParam.create();
    // 从某个 Bundle 创建 BundleParam
    BundleParam anotherBundleParam = BundleParam.create(aBundle);

    // 填充参数字段
    bundleParam.content().putXxx(key, value);
    ...

    Param param = event.getParam() | req.getParam() | res.getParam();
    try {
        // 从参数对象创建 BundleParam
        // 须满足 (!param.isEmpty() && "BundleParam".equals(param.getType())，
        // 否则抛出 InvalidBundleParamException 异常
        BundleParam paramImpl = BundleParam.from(param);

        // 提取参数字段
        AType paramField = paramImpl.content().getXxx(key);
        ...
    } catch (BundleParam.InvalidBundleParamException e) {
        // 处理非法参数异常
    }
}
```

ProtoParam

```
// 需添加 Gradle 依赖，详见“准备”章节
void protoParamSampleCode() {
    // 从 ProtoBuf 的 Any 对象创建 ProtoParam
    ProtoParam protoParam = ProtoParam.pack(any);
    // 从 ProtoBuf 的 消息对象（? extends com.google.protobuf.Message）创建 ProtoParam
    ProtoParam anotherProtoParam = ProtoParam.pack(aMessage);

    // 填充参数字段
    // ProtoParam.pack 构造 ProtoParam 前，由自定义的 ProtoBuf Message 对象的方法填充

    Param param = event.getParam() | req.getParam() | res.getParam();
    try {
        // 从参数对象创建 ProtoParam
        // 须满足 (!param.isEmpty() && "ProtoParam".equals(param.getType())，
        // 否则抛出 InvalidProtoParamException 异常
        ProtoParam paramImpl = ProtoParam.from(param);

        // 获取包含的 Any 对象
        Any any = paramImpl.getAny();
        // 讲 ProtoBuf Message 对象解包出来
        // AMessage extends com.google.protobuf.Message
        AMessage aMessage = paramImpl.unpack(AMessage.class);
        ...
    } catch (ProtoParam.InvalidProtoParamException e) {
        // 处理非法参数异常
    }
}
```

ProtoLiteParam

```
// 需添加 Gradle 依赖，详见“准备”章节
void protoLiteParamSampleCode() {
    // 从 ProtoBufLite 的 消息对象（？extends com.google.protobuf.MessageLite）创建 ProtoLiteParam
    ProtoLiteParam protoLiteParam = ProtoLiteParam.pack(aMessage);

    // 填充参数字段
    // ProtoLiteParam.pack 构造 ProtoLiteParam 前，由自定义的 ProtoBufLite Message 对象的方法填充

    Param param = event.getParam() | req.getParam() | res.getParam();
    try {
        // 从参数对象创建 ProtoLiteParam
        // 须满足 (!param.isEmpty() && "ProtoLiteParam".equals(param.getType())，
        // 否则抛出 InvalidProtoLiteParamException 异常
        ProtoLiteParam paramImpl = ProtoLiteParam.from(param);

        // 获取 ProtoLiteParam 内部的字节数组，
        // 而后将字节数组反序列化为 ProtoBufLite 消息对象（？extends com.google.protobuf.MessageLite）
        byte[] bytes = protoLiteParam.getByteArray();
        ...
    } catch (ProtoLiteParam.InvalidProtoLiteParamException e) {
        // 处理非法参数异常
    }
}
```

GsonParam

```
// 需添加 Gradle 依赖，详见“准备”章节
void gsonParamSampleCode() {
    // 从 JSON String 创建 GsonParam
    GsonParam gsonParam = GsonParam.pack(jsonString);
    // 从普通 Java 对象创建 GsonParam
    GsonParam anotherGsonParam = GsonParam.pack(pojo);

    Param param = event.getParam() | req.getParam() | res.getParam();
    try {
        // 从参数对象创建 GsonParam
        // 须满足 (!param.isEmpty() && "GsonParam".equals(param.getType())，
        // 否则抛出 InvalidGsonParamException 异常
        GsonParam paramImpl = GsonParam.from(param);

        // 提取参数 Java 对象
        AType obj = param.unpack(type);
        AType obj = param.unpack(clazz);
        ...
    } catch (GsonParam.InvalidGsonParamException e) {
        // 处理非法参数异常
    }
}
```

政策方针

准备

- 创建包名为 “com.ubtrobot.master.policy“ 的 Android Project
- 配置 Gradle 依赖

```
// Gradle 4+ 使用 implementation 配置依赖，Gradle < 4 使用 compile 配置依赖

// Master SDK
implementation(group: 'com.ubtrobot.master', name: 'master', version: 'x.y.z')

// 方针政策（Policy）SDK
implementation(group: 'com.ubtrobot.master', name: 'policy', version: 'x.y.z', ext: 'aar')
```

创建 PolicyApplication

```
public class PolicyApplication extends BasePolicyApplication {

    @Override
    protected void onApplicationCreate() {
        // Master.initialize 已经在 BasePolicyApplication 执行

        // 可选择覆盖此方法做一些额外工作，比如配置日志打印
        Master.get().setLoggerFactory(aLoggerFactory);
    }

    @Override
    protected Policy createPolicy() {
        // 如果需要更改默认政策方针，可覆盖此方法自定义政策方针
        //  ❤❤❤ 如果覆盖请确保理解了政策方针的实现原理 ❤❤❤
        Policy defaultPolicy super.createPolicy();
        return new YourCustomPolicy(defaultPolicy);
    }
}
```

创建政策方针配置文件

创建 xml/skill_life_cycle_policy.xml 文件，定义 Skill 生命周期相关的政策方针。规则如下：

```xml
<?xml version="1.0" encoding="utf-8"?>
<skill-list>
  <!-- 如果某个 Skill 在 start 前后或者在设置某个内部状态前后，要约束其他 Skill 的运行，可进行配置 -->
  <skill name="sample_skill" package="sampleskill.packagename">

    <!-- sample_skill 已经 start 的情况，did-start 将决定其他 Skill 能否 start  -->
    <did-start>
      <!-- 如果 sample_skill start 后，少数 Skill 被允许 start，配置白名单 -->
      <whitelist>
        <skill name="a_allow_start_skill" package="its.packagename" />
        ...
      </whitelist>
      <!-- 如果 sample_skill start 后，少数 Skill 不被允许 start，配置黑名单 -->
      <!-- 不允许同时配置黑白名单，两者互斥 -->
      <!-- <blacklist> -->
        <!-- <skill name="a_disallow_start_skill" package="its.packagename" /> -->
        <!-- ... -->
      <!-- </blacklist> -->
    </did-start>

    <!-- 已经 start 的其他 Skill 允许 sample_skill start，在 sample_skill 将 start 前，will-start 决定需要 stop 某些 skill -->
    <will-start>
      <!-- 如果 sample_skill start 前，多数 Skill 需要 stop，配置白名单 -->
      <whitelist>
        <skill name="a_skill_should_not_stop" package="its.package" />
        ...
      </whitelist>
      <!-- 如果 sample_skill start 前，少数 Skill 需要 stop，配置黑名单 -->
      <!-- 不允许同时配置黑白名单，两者互斥 -->
      <!-- <blacklist> -->
        <!-- <skill name="a_skill_should_stop" package="its.packagename" /> -->
        <!-- ... -->
      <!-- </blacklist> -->
    </will-start>

    <!-- sample_skill 设置某个内部状态后，did-set-state 将决定其他 Skill 能否 start  -->
    <did-set-state state_name="doing_something">
      <!-- 如果 sample_skill 设置内部状态后，少数 Skill 被允许 start，配置白名单 -->
      <whitelist>
        <skill name="a_allow_start_skill" package="its.packagename" />
        ...
      </whitelist>
      <!-- 如果 sample_skill 设置内部状态后，少数 Skill 不被允许 start，配置黑名单 -->
      <!-- 不允许同时配置黑白名单，两者互斥 -->
      <!-- <blacklist> -->
        <!-- <skill name="a_disallow_start_skill" package="its.packagename" /> -->
        <!-- ... -->
      <!-- </blacklist> -->
    </did-set-state>

    <!-- sample_skill 设置某个内部状态前，will-set-state 决定需要 stop 某些 skill -->
    <will-set-state state_name="doing_something">
      <!-- 如果 sample_skill 设置内部状态前，多数 Skill 需要 stop，配置白名单 -->
```

```
        <whitelist>
          <skill name="a_skill_should_not_stop" package="its.package" />
          ...
        </whitelist>
        <!-- 如果 sample_skill 设置内部状态前，少数 Skill 需要 stop，配置黑名单 -->
        <!-- 不允许同时配置黑白名单，两者互斥 -->
        <!-- <blacklist> -->
          <!-- <skill name="a_skill_should_stop" package="its.packagename" /> -->
          <!-- ... -->
        <!-- </blacklist> -->
      </will-set-state>
    </skill>
  </skill-list>
```

构建安装

将该 Android Project 构建安装到已安装 Master 的机器人，通过语音、远程设备操控、Launcher 图标点击，调用 Skills，看 Skill 的 start stop 情况是否与 XML 配置预期的一致