

WorkManager

一 WorkManager是用来做什么的?

WorkManager is intended for tasks that require a guarantee that the system will run them even if the app exits, like uploading app data to a server. It is not intended for in-process background work that can safely be terminated if the app process goes away; for situations like that, we recommend using ThreadPools.

- 1 后台异步操作
- 2 即使app退出依然能够运行

二 使用流程

WorkManager的API设计非常简单，每个小模块之间的分工十分明确。从大的角度来看，WorkMnager的使用需要这几样东西：

1 Worker，是一个抽象执行体，需要具体实现，是WorkManager的业务执行模块，所有工作都在doWork方法中执行：

```
@NonNull  
@Override  
public Result doWork() {  
    // do something  
    return null;  
}  
  
public enum Result {  
    SUCCESS,  
    FAILURE,  
    RETRY  
}
```

2 WorkRequest，是一个数据入口，采用链式调用将执行体Worker需要的数据通过方法setInputData注入进去。WorkManager提供了两种Request，OneTimeWorkRequest(只执行一次的request，比如获取一次后台数据)以及PeriodicWorkRequest(可以设置时间间隔多次执行的request，比如定时向后台发送数据)。实例化Request需要将Worker作为Builder参数，每个request都有一个由UUID随即生成的id，然后将创建好的Request添加到WorkManager的队列中：

2.1 OneTimeWorkRequest

```
Data data = new Data.Builder().putString("key_request_0", "value_0").putString("key_request_1", "value_1").build();  
OneTimeWorkRequest mRequest = new OneTimeWorkRequest.Builder(SampleWorker.class)  
    .setInputData(data).build();  
WorkManager.getInstance().enqueue(mRequest);
```

2.2 PeriodicWorkRequest

```
// 每12小时执行一次，执行时机不确定，可能在间隔的末尾也可能在任何实际成熟的时候。  
PeriodicWorkRequest mRequest0 = new PeriodicWorkRequest.Builder(SampleWorker.class  
    , 12, TimeUnit.HOURS).build();  
  
// 每12小时执行一次，执行时机不确定，可能在间隔的末尾也可能在任何实际成熟的时候。  
if (android.os.Build.VERSION.SDK_INT >= android.os.Build.VERSION_CODES.O) {  
    PeriodicWorkRequest mRequest1 = new PeriodicWorkRequest.Builder(SampleWorker.class  
        , Duration.ofHours(12)).build();  
}  
  
// 没12小时执行一次，但是从repeatInterval-flexInterval开始执行，且flexInterval <= repeatInterval。  
PeriodicWorkRequest mRequest2 = new PeriodicWorkRequest.Builder(SampleWorker.class  
    , 12, TimeUnit.HOURS  
    , 10, TimeUnit.HOURS).build();
```



并且PeriodicWorkRequest有严格的interval时间限制：



但是如果设置的interval小于限制并不会报异常，而是自动采用最小值：

如果不需要WorkManager的运行结果，写到这里就可以了。

3 WorkStatus，结果输出者，包括Worker的完成状态(如下枚举)，对应request的id，tag以及数据(一个Date实例)的输出，WorkStatus作为request的输出需要通过WorkManager的相关方法与LiveData添加监听来获取，当WorkStatus状态变化的时候就会给LiveData回调：

3.1 WorkStatus的状态枚举：

```
public enum State {  
    ENQUEUED,  
    RUNNING,  
    SUCCEEDED,  
    FAILED,  
    BLOCKED,  
    CANCELLED;  
    public boolean isFinished() {  
        return (this == SUCCEEDED || this == FAILED || this == CANCELLED);  
    }  
}
```



3.2 获取WorkStatus主要有四个方法，都要添加LiveData：

3.3 具体代码：

```
WorkManager.getInstance().getStatusById(id).observe((LifecycleOwner) mContext, new Observer<WorkStatus>() {  
    @Override  
    public void onChanged(@Nullable WorkStatus workStatus) {  
        switch (workStatus.getState()){  
            case ENQUEUED:{  
                // do something  
            }  
        }  
    }  
}
```

```

}break;
case RUNNING:{
    // do something
}break;
case SUCCEEDED:{
    Data output = workStatus.getOutputData();
    String requestStr = output.getString(WMConstants.DATA_OUTPUT_KEY_REQUEST, WMConstants.DATA_OUTPUT_KEY_REQUEST_DEFAULT);
    String outputStr = output.getString(WMConstants.DATA_OUTPUT_KEY_CONTENT, WMConstants.DATA_OUTPUT_KEY_CONTENT_DEFAULT);
    Log.e(WMConstants.TAG, requestStr + " : " + outputStr);
}break;
case FAILED:{
    // do something
}break;
case BLOCKED:{
    // do something
}break;
case CANCELLED:{
    // do something
}break;
}
}
});

```

那么有一个问题，如果返回结果的时候app已经退出了，但是要进行UI操作怎么办？这已经不是WorkManager可以处理的事情了，实际上这是LiveData的职能范畴。LiveData在返回数据之前会判断绑定时LifecycleOwner的状态，不用说退出，即便是activity/fragment退入后台，也不会返回数据，不用担心造成NPE。这里不费时分析LiveData的原理，有兴趣的同学可以去看看源码，体验一下。

总结



三 灵活多变的任务链

Workmanager对于那些依赖前后返回结果的Task添加了一些使用的可以实现顺序执行的任务链的API。

1 A --> B --> C

```
WorkManager.getInstance().beginWith(RequestA).then(RequestB).then(RequestC).enqueue();
```

这里有一个需要注意的地方，任务链的Request的都是OneTimeWorkRequest。为什么？

2 还可以通过WorkContinuation实现更复杂的任务链

A --> B

-->E

C --> D

```
WorkContinuation chain1 = WorkManager.getInstance()
    .beginWith(RequestA)
```

```

    .then(RequestB);
WorkContinuation chain2 = WorkManager.getInstance()
    .beginWith(RequestC)
    .then(RequestD);
WorkContinuation chain3 = WorkContinuation
    .combine(chain1, chain2)
    .then(RequestE);
chain3.enqueue();

```

四 唯一任务

每个创建的Request的ID都是经过UUID的随即获取方法随机获取到的，那么就WorkRequest而言我们基本是无法确定一个任务是否正在执行或者在队列中等待pop，所以为了解决这个问题，WorkManager中添加了方法 **beginUniqueWork** 用来push唯一name的OneTimeWorkRequest：

```

public final WorkContinuation beginUniqueWork(
    @NotNull String uniqueWorkName,
    @NotNull ExistingWorkPolicy existingWorkPolicy,
    @NotNull OneTimeWorkRequest... work) {
    return beginUniqueWork(uniqueWorkName, existingWorkPolicy, Arrays.asList(work));
}

public abstract WorkContinuation beginUniqueWork(
    @NotNull String uniqueWorkName,
    @NotNull ExistingWorkPolicy existingWorkPolicy,
    @NotNull List<OneTimeWorkRequest> work);

```

```

public enum ExistingWorkPolicy {

    /**
     * If there is existing pending work with the same unique name, cancel and delete it. Then,
     * insert the newly-specified work.
     */
    // 新任务取代旧任务
    REPLACE,

    /**
     * If there is existing pending work with the same unique name, do nothing. Otherwise, insert
     * the newly-specified work.
     */
    // 忽略新任务
    KEEP,

    /**
     * If there is existing pending work with the same unique name, append the newly-specified work
     * as a child of all the leaves of that work sequence. Otherwise, insert the newly-specified
     * work as the start of a new sequence.
     */
    // 如果已经存在相同name的work，就把新任务添加到queue末尾，否则添加在queue的开头
    APPEND
}

```

还有添加唯一PeriodicWorkRequest的方法enqueueUniquePeriodicWork(), 和上面的方法基本一样，不作赘述。

五 深入思考

- 1 与常规的异步(AsyncTask, ThreadPool, Rx等)有何不同?
- 2 与Service有何不同?
- 3 WorkManager底层是怎样的?
- 4 即便是App退出，进程被杀死，WorkManager仍然会重启进程(进程id不同)执行任务?