ALIBABA CLOUD

阿里云

分布式任务调度 SchedulerX

分布式编程模型

文档版本：

阿里云

法律声明

阿里云提醒您在阅读或使用本文档之前仔细阅读、充分理解本法律声明各条款的内容。如果您阅读或使用本文档，您的阅读或使用行为将被视为对本声明全部内容的认可。

1. 您应当通过阿里云网站或阿里云提供的其他授权通道下载、获取本文档，且仅能用于自身的合法合规的业务活动。本文档的内容视为阿里云的保密信息，您应当严格遵守保密义务；未经阿里云事先书面同意，您不得向任何第三方披露本手册内容或提供给任何第三方使用。

2. 未经阿里云事先书面许可，任何单位、公司或个人不得擅自摘抄、翻译、复制本文档内容的部分或全部，不得以任何方式或途径进行传播和宣传。

3. 由于产品版本升级、调整或其他原因，本文档内容有可能变更。阿里云保留在没有任何通知或者提示下对本文档的内容进行修改的权利，并在阿里云授权通道中不时发布更新后的用户文档。您应当实时关注用户文档的版本变更并通过阿里云授权渠道下载、获取最新版的用户文档。

4. 本文档仅作为用户使用阿里云产品及服务的参考性指引，阿里云以产品及服务的“现状”、“有缺陷”和“当前功能”的状态提供本文档。阿里云在现有技术的基础上尽最大努力提供相应的介绍及操作指引，但阿里云在此明确声明对本文档内容的准确性、完整性、适用性、可靠性等不作任何明示或暗示的保证。任何单位、公司或个人因为下载、使用或信赖本文档而发生任何差错或经济损失的，阿里云不承担任何法律责任。在任何情况下，阿里云均不对任何间接性、后果性、惩戒性、偶然性、特殊性或刑罚性的损害，包括用户使用或信赖本文档而遭受的利润损失，承担责任（即使阿里云已被告知该等损失的可能性）。

5. 阿里云网站上所有内容，包括但不限于著作、产品、图片、档案、资讯、资料、网站架构、网站画面的安排、网页设计，均由阿里云和/或其关联公司依法拥有其知识产权，包括但不限于商标权、专利权、著作权、商业秘密等。非经阿里云和/或其关联公司书面同意，任何人不得擅自使用、修改、复制、公开传播、改变、散布、发行或公开发表阿里云网站、产品程序或内容。此外，未经阿里云事先书面同意，任何人不得为了任何营销、广告、促销或其他目的使用、公布或复制阿里云的名称（包括但不限于单独为或以组合形式包含“阿里云”、“Aliyun”、“万网”等阿里云和/或其关联公司品牌，上述品牌的附属标志及图案或任何类似公司名称、商号、商标、产品或服务名称、域名、图案标示、标志、标识或通过特定描述使第三方能够识别阿里云和/或其关联公司）。

6. 如若发现本文档存在任何错误，请与阿里云取得直接联系。

通用约定

格式	说明	样例
危险	该类警示信息将导致系统重大变更甚至故障，或者导致人身伤害等结果。	危险 重置操作将丢失用户配置数据。
警告	该类警示信息可能会导致系统重大变更甚至故障，或者导致人身伤害等结果。	警告 重启操作将导致业务中断，恢复业务时间约十分钟。
注意	用于警示信息、补充说明等，是用户必须了解的内容。	注意 权重设置为0，该服务器不会再接受新请求。
说明	用于补充说明、最佳实践、窍门等，不是用户必须了解的内容。	说明 您也可以通过按Ctrl+A选中全部文件。
>	多级菜单递进。	**单击设置> 网络> 设置网络类型。**
粗体	表示按键、菜单、页面名称等UI元素。	**在结果确认页面，单击确定。**
`Courier字体`	命令或代码。	执行 `cd /d C:/window` 命令，进入Windows系统文件夹。
斜体	表示参数、变量。	`bae log list --instanceid` *Instance_ID*
[] 或者 [a|b]	表示可选项，至多选择一个。	`ipconfig [-all|-t]`
{} 或者 {a|b}	表示必选项，至多选择一个。	`switch {active|stand}`

目录

1.单机

单机执行表示一个任务实例只会随机触发到一台Worker上，支持所有的任务类型。

2.广播

广播执行表示一个任务实例会广播到该分组所有Worker上执行，当所有Worker都执行完成，该任务才算完成。任意一台Worker执行失败，都算该任务失败。

应用场景

- 批量运维
  - 定时广播所有机器运行某个脚本。
  - 定时广播所有机器清理缓存。
- 数据聚合
  - 使用JavaProcessor，preProcess的时候重置数据库count=0。
  - 每台机器执行process的时候，根据自己机器内存或者日志的值，在数据库对count进行叠加。
  - postProcess的时候，读取数据库count值，传递给下游。

任务类型

任务类型可以选择多种，例如脚本或者Java任务。如果选择Java，还支持preProcess和postProcess高级特性。

使用Java任务需要继承JavaProcessor（1.0.7及以上版本），接口如下：

- `public ProcessResult process(JobContext context) throws Exception`
- （可选）`public void preProcess(JobContext context)`
- （可选）`public ProcessResult postProcess(JobContext context)`

preProcess会在所有机器执行process之前执行，且只会执行一次。

postProcess会在所有机器执行process且都成功执行之后执行一次，可以返回结果，作为工作流数据传输。

Demo示例

```
@Component
public class TestBroadcastJobProcessor extends JavaProcessor {

@Override
public ProcessResult process(JobContext context) throws Exception {
System.out.println("this is process");
return new ProcessResult(true);
}

@Override
public void preProcess(JobContext context) {
System.out.println("this is preProcess");
}

@Override
public ProcessResult postProcess(JobContext context) {
System.out.println("this is postProcess");
return new ProcessResult(true, "hello broadcast");
}

}
```

如以上代码所示，启动三台机器，执行结果如下：

- Worker 1

```
this is preProcess
this is process
this is postProcess
```

- Worker 2

```
this is process
```

- Worker 3

```
this is process
```

3.Map模型

基于MapJobProcessor，调用Map方法，即可实现大数据分布式跑批的能力。

注意事项

- SchedulerX不保证子任务一定执行一次，在特殊条件下会failover，可能会导致子任务重复执行，需要业务方自己实现幂等。
- SchedulerX使用的是Hessian序列化框架，目前不支持LocalDateTime和BigDecimal。子任务中如果有如上两个数据结构，请替换其他的数据结构（特别是BigDecimal，序列化不会报错，反序列化会变成0）。

接口

接口	解释	是否必选
`public ProcessResult process(JobContext context) throws Exception;`	每个子任务执行业务的入口，需要从context里获取taskName，自己判断是哪个子任务，进行相应的逻辑处理。执行完成后，需要返回ProcessResult。ProcessResult接口请参见ProcessResult。	是
`public void postProcess(JobContext context);`	无	否
`public void kill(JobContext context);`	前端kill任务会触发该方法，需要用户自己实现如何中断业务。	否
`public ProcessResult map(List<? extends Object> taskList, String taskName)`	执行map方法可以把一批子任务分布式到多台机器上执行，可以map多次。如果taskList是空，返回失败。执行完成后，需要返回ProcessResult。ProcessResult接口请参见ProcessResult。	是

执行方式

- 并行计算：最多支持300任务，有子任务列表。

  > 注意 秒级别任务不要选择并行计算。

- 内存网格：基于内存计算，最多支持50,000以下子任务，速度快。
- 网格计算：基于文件计算，最多支持1,000,000子任务。

高级配置

任务管理高级配置参数说明如下：

参数	适用的执行模式	解释	默认值
实例失败重试次数	通用	任务运行失败自动重试的次数。	0
实例失败重试间隔	通用	每次失败重试的间隔。单位：秒。	30
实例并发数	通用	同一个Job同一时间运行的实例个数。1表示不允许重复执行。	1
子任务单机并发数	• 并行计算 • 内存网格 • 网格计算	分布式模型，单台机器并发消费子任务的个数。	5
子任务失败重试次数	• 并行计算 • 内存网格 • 网格计算	分布式模型，子任务失败自动重试的次数。	0
子任务失败重试间隔	• 并行计算 • 内存网格 • 网格计算	分布式模型，子任务失败自动重试的间隔。单位：秒。	0
子任务分发方式	• 并行计算 • 内存网格 • 网格计算	• **推模型**：每台机器平均分配子任务。 • **拉模型**： 每台机器主动拉取子任务，没有木桶效应。拉取过程中，所有子任务会缓存在Master节点，对内存有压力，建议子任务数不超过10,000。	推模型
子任务单次拉取数（仅适用于拉模型）	• 并行计算 • 内存网格 • 网格计算	Slave节点每次向Master节点拉取多少个子任务。	5
子任务队列容量（仅适用于拉模型）	• 并行计算 • 内存网格 • 网格计算	Slave节点缓存子任务的队列大小。	10
子任务全局并发数（仅适用于拉模型）	• 并行计算 • 内存网格 • 网格计算	分布式拉模型支持全局子任务并发数，可以进行限流。	1,000

发送50条消息的Demo示例（适用于Map模型）

```
@Component
public class TestMapJobProcessor extends MapJobProcessor {

@Override
public ProcessResult process(JobContext context) throws Exception {
String taskName = context.getTaskName();
int dispatchNum = 50;
if (isRootTask(context)) {
System.out.println("start root task");
List<String> msgList = Lists.newArrayList();
for (int i = 0; i <= dispatchNum; i++) {
msgList.add("msg_" + i);
}
return map(msgList, "Level1Dispatch");
} else if (taskName.equals("Level1Dispatch")) {
String task = (String)context.getTask();
System.out.println(task);
return new ProcessResult(true);
}

return new ProcessResult(false);
}

}
```

处理单表数据的Demo示例（适用于Map或MapReduce模型）

```
@Component
public class ScanSingleTableJobProcessor extends MapJobProcessor {

@Service
private XXXService xxxService;

private final int PAGE_SIZE = 500;

static class PageTask {
private long startId;
private long endId;
```

```
public PageTask(long startId, long endId) {
this.startId = startId;
this.endId = endId;
}

public long getStartId() {
return startId;
}

public long getEndId() {
return endId;
}
}

@Override
public ProcessResult process(JobContext context) throws Exception {
String tableName = context.getJobParameters(); //多个Job后端代码可以一致，通过控制台配置Job参数表示表名。
String taskName = context.getTaskName();
Object task = context.getTask();
if (isRootTask(context)) {
Pair<Long, Long> idPair = queryMinAndMaxId(tableName);
long minId = idPair.getFirst();
long maxId = idPair.getSecond();
List<PageTask> tasks = Lists.newArrayList();
int step = (int) ((maxId - minId) / PAGE_SIZE); //计算分页数量
for (long i = minId; i < maxId; i+=step) {
tasks.add(new PageTask(i, (i+step > maxId ? maxId : i+step)));
}
return map(tasks, "PageTask");
} else if (taskName.equals("PageTask")) {
PageTask pageTask = (PageTask)task;
long startId = pageTask.getStartId();
long endId = pageTask.getEndId();
List<Record> records = queryRecord(tableName, startId, endId);
//TODO handle records
return new ProcessResult(true);
}

return new ProcessResult(false);
}
```

```
}

private Pair<Long, Long> queryMinAndMaxId(String tableName) {
//TODO select min(id),max(id) from [tableName]
return new Pair<Long, Long>(1L, 10000L);
}

private List<Record> queryRecord(String tableName, long startId, long endId) {
List<Record> records = Lists.newArrayList();
//TODO select * from [tableName] where id>=[startId] and id<[endId]
return records;
}

}
```

处理分库分表数据的Demo示例（适用于Map或MapReduce模型）

```
@Component
public class ScanShardingTableJobProcessor extends MapJobProcessor {

@Service
private XXXService xxxService;

private final int PAGE_SIZE = 500;

static class PageTask {
private String tableName;
private long startId;
private long endId;

public PageTask(String tableName, long startId, long endId) {
this.tableName = tableName;
this.startId = startId;
this.endId = endId;
}

public String getTableName() {
return tableName;
}

public long getStartId() {
return startId;
```

```
return startId;
}

public long getEndId() {
return endId;
}
}

@Override
public ProcessResult process(JobContext context) throws Exception {
String taskName = context.getTaskName();
Object task = context.getTask();
if (isRootTask(context)) {
//先分库
List<String> dbList = getDbList();
return map(dbList, "DbTask");
} else if (taskName.equals("DbTask")) {
//根据分库去分表
String dbName = (String)task;
List<String> tableList = getTableList(dbName);
return map(tableList, "TableTask");
} else if (taskName.equals("TableTask")) {
//如果一个分表也很大，再分页
String tableName = (String)task;
Pair<Long, Long> idPair = queryMinAndMaxId(tableName);
long minId = idPair.getFirst();
long maxId = idPair.getSecond();
List<PageTask> tasks = Lists.newArrayList();
int step = (int) ((maxId - minId) / PAGE_SIZE); //计算分页数量
for (long i = minId; i < maxId; i+=step) {
tasks.add(new PageTask(tableName, i, (i+step > maxId ? maxId : i+step)));
}
return map(tasks, "PageTask");
} else if (taskName.equals("PageTask")) {
PageTask pageTask = (PageTask)task;
String tableName = pageTask.getTableName();
long startId = pageTask.getStartId();
long endId = pageTask.getEndId();
List<Record> records = queryRecord(tableName, startId, endId);
//TODO handle records
return new ProcessResult(true);
```

```
}

return new ProcessResult(false);
}

private List<String> getDbList() {
List<String> dbList = Lists.newArrayList();
//TODO 返回分库列表
return dbList;
}

private List<String> getTableList(String dbName) {
List<String> tableList = Lists.newArrayList();
//TODO 返回分表列表
return tableList;
}

private Pair<Long, Long> queryMinAndMaxId(String tableName) {
//TODO select min(id),max(id) from [tableName]
return new Pair<Long, Long>(1L, 10000L);
}

private List<Record> queryRecord(String tableName, long startId, long endId) {
List<Record> records = Lists.newArrayList();
//TODO select * from [tableName] where id>=[startId] and id<[endId]
return records;
}

}
```

4.MapReduce模型

MapReduce模型是Map模型的扩展，新增Reduce接口，需要实现MapReduceJobProcessor。

背景信息

- MapReduce模型只有一个Reduce，所有子任务完成后会执行Reduce方法，可以在Reduce方法中返回该任务示例的执行结果，作为工作流的上下游数据传递。如果有子任务失败，Reduce不会执行。Reduce失败，整个任务示例也失败。
- MapReduce模型还能处理所有子任务的结果。子任务通过 `return ProcessResult(true, result)` 返回结果（例如返回订单号），Reduce的时候，可以通过context获取所有子任务的结果，进行相应的处理。

MapReduce模型的原理和最佳实践，请参见SchedulerX 2.0 分布式计算原理和最佳实践。

SchedulerX 2.0支持MapReduce模型的详细信息，请参见SchedulerX 2.0 支持 MapReduce 模型。

注意事项

- 所有子任务结果会缓存在Master节点，内存压力较大，建议子任务个数和Result不要太大。
- SchedulerX不保证子任务绝对执行一次，在特殊条件下会Failover，可能会导致子任务重复执行，需要业务方自己实现幂等。

接口

接口	解释	是否必选
`public ProcessResult process(JobContext context) throws Exception;`	每个子任务执行业务的入口，需要从context里获取taskName，自己判断是哪个子任务，进行相应的逻辑处理。执行完成后，需要返回ProcessResult。	是
`public ProcessResult map(List<? extends Object> taskList, String taskName);`	执行map方法可以把一批子任务分布式到多台机器上执行，可以map多次。如果taskList是空，返回失败。执行完成后，需要返回ProcessResult。	是
`public ProcessResult reduce(JobContext context);`	无	是
`public void kill(JobContext context);`	前端kill任务会触发该方法，需要用户自己实现如何中断业务。	否

执行方式

- 并行计算：最多支持300任务，有子任务列表。

  > 注意 秒级别任务不要选择并行计算。

- 内存网格：基于内存计算，最多支持50,000以下子任务，速度快。

- 网格计算：基于文件计算，最多支持1,000,000子任务。

高级配置

任务管理高级配置参数说明如下：

参数	适用的执行模式	解释	默认值
实例失败重试次数	通用	任务运行失败自动重试的次数。	0
实例失败重试间隔	通用	每次失败重试的间隔。单位：秒。	30
实例并发数	通用	同一个Job同一时间运行的实例个数。1表示不允许重复执行。	1
子任务单机并发数	• 并行计算 • 内存网格 • 网格计算	分布式模型，单台机器并发消费子任务的个数。	5
子任务失败重试次数	• 并行计算 • 内存网格 • 网格计算	分布式模型，子任务失败自动重试的次数。	0
子任务失败重试间隔	• 并行计算 • 内存网格 • 网格计算	分布式模型，子任务失败自动重试的间隔。单位：秒。	0
子任务分发方式	• 并行计算 • 内存网格 • 网格计算	• **推模型**：每台机器平均分配子任务。 • **拉模型**： 每台机器主动拉取子任务，没有木桶效应。拉取过程中，所有子任务会缓存在Master节点，对内存有压力，建议子任务数不超过10,000。	推模型
子任务单次拉取数（仅适用于拉模型）	• 并行计算 • 内存网格 • 网格计算	Slave节点每次向Master节点拉取多少个子任务。	5
子任务队列容量（仅适用于拉模型）	• 并行计算 • 内存网格 • 网格计算	Slave节点缓存子任务的队列大小。	10

参数	适用的执行模式	解释	默认值
子任务全局并发数（仅适用于拉模型）	• 并行计算 • 内存网格 • 网格计算	分布式拉模型支持全局子任务并发数，可以进行限流。	1,000

发送500条消息的Demo示例（适用于MapReduce模型）

```
@Component
public class TestMapReduceJobProcessor extends MapReduceJobProcessor {

@Override
public ProcessResult process(JobContext context) throws Exception {
String taskName = context.getTaskName();
int dispatchNum=500;
if (isRootTask(context)) {
System.out.println("start root task");
List<String> msgList = Lists.newArrayList();
for (int i = 0; i <= dispatchNum; i++) {
msgList.add("msg_" + i);
}
return map(msgList, "Level1Dispatch");
} else if (taskName.equals("Level1Dispatch")) {
String task = (String)context.getTask();
System.out.println(task);
return new ProcessResult(true);
}

return new ProcessResult(false);
}

@Override
public ProcessResult reduce(JobContext context) throws Exception {
return new ProcessResult(true, "TestMapReduceJobProcessor.reduce");
}

}
```

处理单表数据的Demo示例（适用于Map或MapReduce模型）

```
@Component
```

```
public class ScanSingleTableJobProcessor extends MapJobProcessor {

@Service
private XXXService xxxService;

private final int PAGE_SIZE = 500;

static class PageTask {
private long startId;
private long endId;

public PageTask(long startId, long endId) {
this.startId = startId;
this.endId = endId;
}

public long getStartId() {
return startId;
}

public long getEndId() {
return endId;
}
}

@Override
public ProcessResult process(JobContext context) throws Exception {
String tableName = context.getJobParameters(); //多个Job后端代码可以一致，通过控制台配置Job参数表示表名。
String taskName = context.getTaskName();
Object task = context.getTask();
if (isRootTask(context)) {
Pair<Long, Long> idPair = queryMinAndMaxId(tableName);
long minId = idPair.getFirst();
long maxId = idPair.getSecond();
List<PageTask> tasks = Lists.newArrayList();
int step = (int) ((maxId - minId) / PAGE_SIZE); //计算分页数量
for (long i = minId; i < maxId; i+=step) {
tasks.add(new PageTask(i, (i+step > maxId ? maxId : i+step)));
}
return map(tasks, "PageTask");
```

```
} else if (taskName.equals("PageTask")) {
PageTask pageTask = (PageTask)task;
long startId = pageTask.getStartId();
long endId = pageTask.getEndId();
List<Record> records = queryRecord(tableName, startId, endId);
//TODO handle records
return new ProcessResult(true);
}

return new ProcessResult(false);
}

private Pair<Long, Long> queryMinAndMaxId(String tableName) {
//TODO select min(id),max(id) from [tableName]
return new Pair<Long, Long>(1L, 10000L);
}

private List<Record> queryRecord(String tableName, long startId, long endId) {
List<Record> records = Lists.newArrayList();
//TODO select * from [tableName] where id>=[startId] and id<[endId]
return records;
}

}
```

处理分库分表数据的Demo示例（适用于Map或MapReduce模型）

```
@Component
public class ScanShardingTableJobProcessor extends MapJobProcessor {

@Service
private XXXService xxxService;

private final int PAGE_SIZE = 500;

static class PageTask {
private String tableName;
private long startId;
private long endId;

public PageTask(String tableName, long startId, long endId) {
```

```
public PageTask(String tableName, long startId, long endId) {
this.tableName = tableName;
this.startId = startId;
this.endId = endId;
}

public String getTableName() {
return tableName;
}

public long getStartId() {
return startId;
}

public long getEndId() {
return endId;
}
}

@Override
public ProcessResult process(JobContext context) throws Exception {
String taskName = context.getTaskName();
Object task = context.getTask();
if (isRootTask(context)) {
//先分库
List<String> dbList = getDbList();
return map(dbList, "DbTask");
} else if (taskName.equals("DbTask")) {
//根据分库去分表
String dbName = (String)task;
List<String> tableList = getTableList(dbName);
return map(tableList, "TableTask");
} else if (taskName.equals("TableTask")) {
//如果一个分表也很大，再分页
String tableName = (String)task;
Pair<Long, Long> idPair = queryMinAndMaxId(tableName);
long minId = idPair.getFirst();
long maxId = idPair.getSecond();
List<PageTask> tasks = Lists.newArrayList();
int step = (int) ((maxId - minId) / PAGE_SIZE); //计算分页数量
for (long i = minId; i < maxId; i+=step) {
```

```
tasks.add(new PageTask(tableName, i, (i+step > maxId ? maxId : i+step)));
}
return map(tasks, "PageTask");
} else if (taskName.equals("PageTask")) {
PageTask pageTask = (PageTask)task;
String tableName = pageTask.getTableName();
long startId = pageTask.getStartId();
long endId = pageTask.getEndId();
List<Record> records = queryRecord(tableName, startId, endId);
//TODO handle records
return new ProcessResult(true);
}

return new ProcessResult(false);
}

private List<String> getDbList() {
List<String> dbList = Lists.newArrayList();
//TODO 返回分库列表
return dbList;
}

private List<String> getTableList(String dbName) {
List<String> tableList = Lists.newArrayList();
//TODO 返回分表列表
return tableList;
}

private Pair<Long, Long> queryMinAndMaxId(String tableName) {
//TODO select min(id),max(id) from [tableName]
return new Pair<Long, Long>(1L, 10000L);
}

private List<Record> queryRecord(String tableName, long startId, long endId) {
List<Record> records = Lists.newArrayList();
//TODO select * from [tableName] where id>=[startId] and id<[endId]
return records;
}

}
```

处理50条消息并且返回子任务结果由Reduce汇总的Demo示例（适用于MapReduce模型）

```
@Component
public class TestMapReduceJobProcessor extends MapReduceJobProcessor {

@Override
public ProcessResult process(JobContext context) throws Exception {
String taskName = context.getTaskName();
int dispatchNum = 50;
if (context.getJobParameters() != null) {
dispatchNum = Integer.valueOf(context.getJobParameters());
}
if (isRootTask(context)) {
System.out.println("start root task");
List<String> msgList = Lists.newArrayList();
for (int i = 0; i <= dispatchNum; i++) {
msgList.add("msg_" + i);
}
return map(msgList, "Level1Dispatch");
} else if (taskName.equals("Level1Dispatch")) {
String task = (String)context.getTask();
Thread.sleep(2000);
return new ProcessResult(true, task);
}

return new ProcessResult(false);
}

@Override
public ProcessResult reduce(JobContext context) throws Exception {
for (Entry<Long, String> result : context.getTaskResults().entrySet()) {
System.out.println("taskId:" + result.getKey() + ", result:" + result.getValue());
}
return new ProcessResult(true, "TestMapReduceJobProcessor.reduce");
}
}
```

5.多语言版本分片模型

SchedulerX可以对多重任务进行调度（定时、编排、重刷历史数据等），提供Java、Python、Shell和Go等多语言分片模型，帮助您处理大数据业务需求。

背景信息

分片模型主要包含静态分片和动态分片。

- 静态分片：主要场景是处理固定的分片数，例如分库分表中固定1024张表，需要若干台机器分布式去处理。
- 动态分片：主要场景是分布式处理未知数据量的数据，例如一张大表在不停变更，需要分布式跑批。主流的框架为SchedulerX提供的MapReduce模型，暂时还没有对外开源。

功能特性

多语言版本分片模型还具有以下特性。

- 兼容elastic-job的静态分片模型。
- 支持Java、Python、Shell、Go四种语言。
- 高可用：分片模型基于Map模型开发，可以继承Map模型高可用特性，即某台worker执行过程中发生异常，master worker会把分片failover到其它slave节点执行。
- 流量控制：分片模型基于Map模型开发，可以继承Map模型流量控制特性，即可以控制单机子任务并发度。例如有1000个分片，一共10台机器，可以控制最多5个分片并发跑，其它在队列中等待。
- 分片自动失败重试：分片模型基于Map模型开发，可以继承Map模型子任务失败自动重试特性。

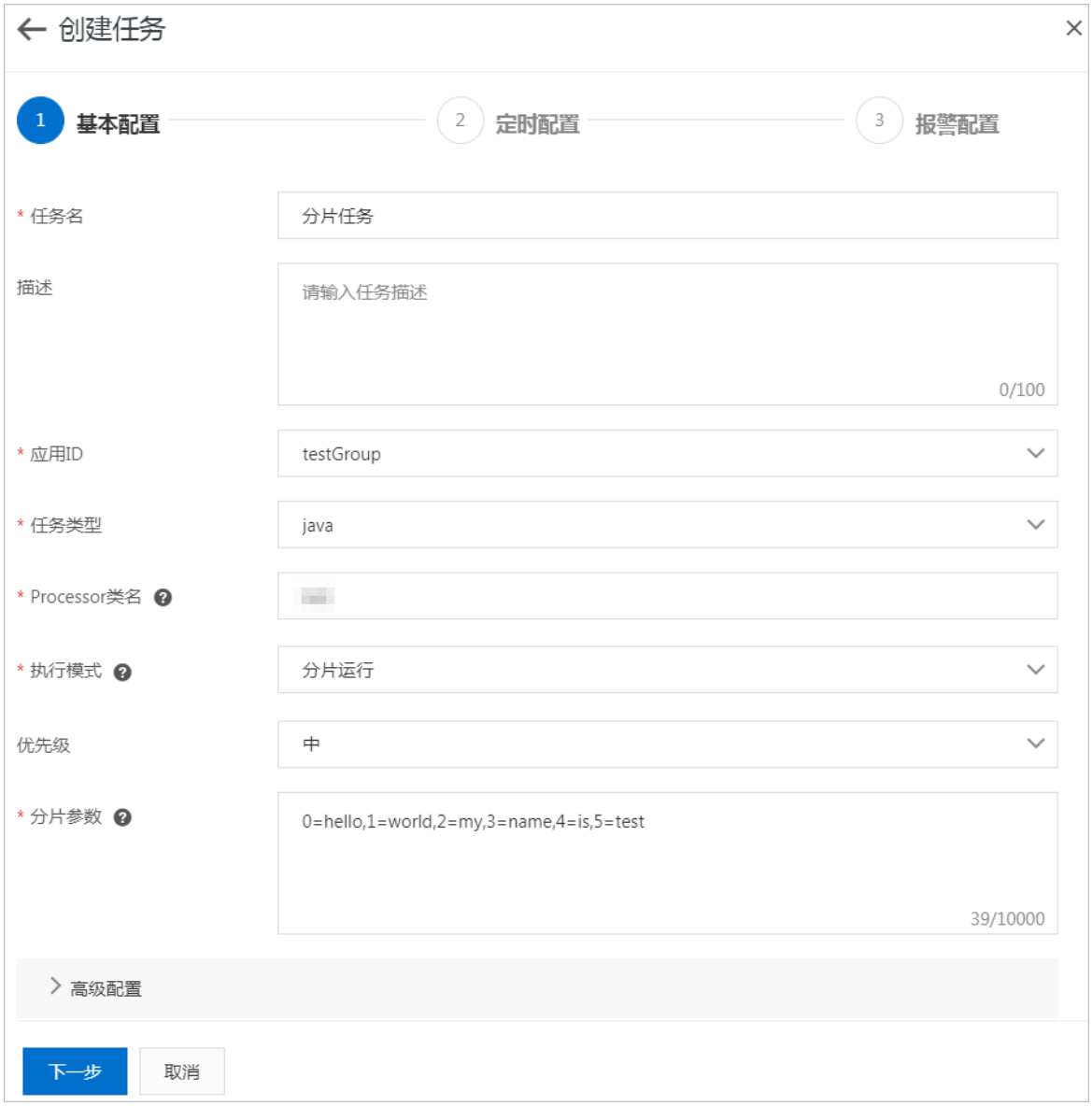
可用性和流量控制可以在创建任务时的高级配置中设置，详情请参见创建调度任务和任务管理高级配置参数说明。

> **说明** 只有1.1.0及以上版本客户端才支持多语言版本的分片模型。

Java分片任务

1. 登录分布式任务调度平台。
2. 在顶部菜单栏选择**地域**。
3. 在左侧导航栏选择**任务管理**。
4. 在**执行列表**页面的**所属命名空间**列表选择具体的命名空间，在页面左上角单击**创建任务**。
5. 在**创建任务**配置向导**基本配置**页面的**执行模式**列表选择**分片运行**，并设置**分片参数**。

   分片参数之间以英文逗号或换行分隔，例如 `分片号1=分片参数1,分片号2=分片参数2,...` 。



6. 在应用程序代码中继承 `JavaProcessor` ，通过 `JobContext.getShardingId()` 获取分片号，通过 `JobContext.getShardingParameter()` 获取分片参数。 示例：

```
@Component
public class HelloWorldProcessor extends JavaProcessor {
@Override
public ProcessResult process(JobContext context) throws Exception {
System.out.println("分片id=" + context.getShardingId() + ", 分片参数=" + context.getShardingParameter());
return new ProcessResult(true);
}
}
```

7. 在执行列表页面查看分片详情。

Python分片任务

Python应用想使用分布式跑批，只需要安装Agent。脚本可以由SchedulerX维护。

1. 下载SchedulerX的 Agent，并通过Agent部署脚本任务。详情请参见为应用实现任务调度（非 EDAS 部署）。
2. 在SchedulerX中创建Python分片任务，详情请参见创建调度任务。

   `sys.argv[1]` 为分片号，`sys.argv[2]` 为分片参数。

   分片参数之间以英文逗号或换行分隔，例如 `分片号1=分片参数1,分片号2=分片参数2,...` 。

3. 在执行列表页面查看分片详情。

Shell和Go分片任务

Shell和Go版本的分片任务和Python类似，创建步骤请参见Python分片任务。