

# Quartz学习

2021年3月30日 15:19

官方地址: <https://www.quartz-scheduler.net/>

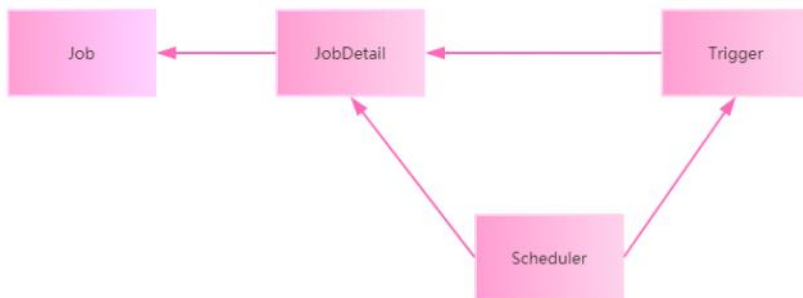
用途: quartz是一个纯净的基于C#实现的任务作业调度系统, 功能十分强大, 支持简单的定时器, 也支持自定义Cron; 即能实现定时重复执行任务, 也可以支持排除特殊日子执行任务, 上手简单, 只需要继承IJOB接口

在使用Quartz之前, 需要明确基本构成结构与基本工作流程, 包含以下7个部分

IScheduler	调度器, 与调度程序交互的主要API
IJob	调度程序所要执行的任务
IJobDetail	用于定义作业的实例
Tigger	触发器, 定义了调度程序的时间规则
Itigger	用于定义给定作业的触发器实例, 一个作业可以关联多个触发器
JobBuilder	用于定义作业的详细实例, 该实例服务于IJobDetail
TriggerBuilder	用于定义触发器的详细实例, 该实例服务于Itigger

一个简单的基本工作流程: 实例化一个Scheduler作为独立的运行容器, 然后将tigger与job注册进容器中, 其中tigger控制执行的时间规则, job为被执行任务, 一个job可以拥有多个触发器, 而一个触发器只能属于一个job。

Quartz中一个调度线程QuartzSchedulerThread, 调度线程可以找到将要被触发的trigger, 通过trigger找到要执行的job, 然后在ThreadPool中获取一个线程来执行这个job。JobStore主要作用是存放job和trigger的信息。



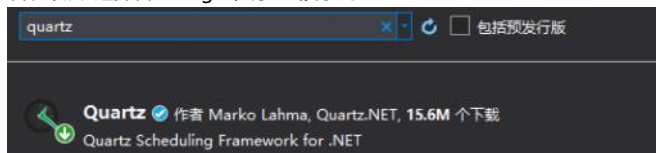
## 一、快速使用Quartz

### 1. 下载与引用

此Demo为.NET控制台程序, 框架版本为4.7.2

#### 1.1 使用NuGet程序包引入

右键项目-选择管理Nuget程序包-搜索Quartz



下载最新版, 当前最新版为v3.2.4

#### 1.2 入门使用-创建业务类并实现接口

程序集导入成功后, 创建一个业务类, 并继承IJOB接口, 并且实现接口里的Execute方法

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Quartz;

namespace QuartzTest
{
    //业务类, 继承此类
    1 个引用
    class JobLgoTask : IJob
    {
        //必须实现
        0 个引用
        public async Task Execute(IJobExecutionContext context)
        {
            //业务逻辑
            await Task.Run(() =>
            {
                Console.WriteLine("hello quartz!");
                //JobDetail的key就是job的分组和job的名字
                Console.WriteLine($"JobDetail的组名字: {context.JobDetail.Key}");
                Console.WriteLine();
            });
        }
    }
}

```

1.2.1 3.x的版本与2.x的版本主要区别于高版本支持异步执行Job，所以这里建议将Execute设计为异步方法，使用async/await关键字

1.2.2 这里的\$为C#6.0新增的字符串拼接方法，这里看代码更直观

```

//语法: $" string {参数}"
//例如 以前的方法拼接一个字符串
string Name = "张三";
string Age = "18";
//方法1:
string str = "My Name is " + Name + ", My Age is " + Age;
//方法2:
string str2 = string.Format("Name is {0}, Age is {1}", Name, Age);
//方法3 (使用$完成拼接)
string str3 = $"Name is {Name}, Age is {Age}";

```

1.2.3 方法声明了async，则内部必须至少声明一个await

## 2. 编写调度业务

### 2.1 实例化相关结构

```

//实例一个调度工厂，并且从调度工厂实例一个调度器
//此时线程已启动，不过是处于等待时间
StdSchedulerFactory factory = new StdSchedulerFactory();
IScheduler scheduler = factory.GetScheduler().Result;

```

### 2.1 创建Job任务

```

//创建一个Job, 与业务逻辑JobLgoTask绑定
IJobDetail job = JobBuilder
    .Create<JobLgoTask>() //获取JobBuilder
    .WithIdentity("jobname1", "group1") //添加Job的名字和分组
    .WithDescription("一个简单的任务") //添加描述
    .Build(); //生成IJobDetail绑定

```

### 2.2 创建Tigger触发器

```

//创建一个触发器
ITrigger trigger =
    TriggerBuilder.Create() //获取TriggerBuilder
        .StartAt(DateBuilder.TodayAt(01, 00, 00)) //开始时间, 今天的1点 (hh, mm, ss), 可使用StartNow()
        .ForJob(job) //将触发器关联给指定的job
        .WithPriority(10) //优先级, 当触发时间一样时, 优先级大的触发器先执行
        .WithIdentity("tname1", "group1") //添加名字和分组
        .WithSimpleSchedule(x => x.WithIntervalInSeconds(1) //调度: 一秒执行一次, 执行三次
            .WithRepeatCount(3)
        )
        .Build();

```

### 2.3 将触发器与job任务注册并执行

```

scheduler.ScheduleJob(job, trigger);
scheduler.Start();
Console.ReadKey();

```

### 2.4 完整代码示例

```

namespace QuartzTest
{
    0 个引用
    class Program
    {
        0 个引用
        static void Main(string[] args)
        {
            //实例一个调度工厂, 并且从调度工厂实例一个调度器
            //此时线程已启动, 不过是处于等待时间
            StdSchedulerFactory factory = new StdSchedulerFactory();
            IScheduler scheduler = factory.GetScheduler().Result;

            //创建一个Job, 与业务逻辑JobLgoTask绑定
            IJobDetail job = JobBuilder
                .Create<JobLgoTask>() //获取JobBuilder
                .WithIdentity("jobname1", "group1") //添加Job的名字和分组
                .WithDescription("一个简单的任务") //添加描述
                .Build(); //生成IJobDetail绑定

            //创建一个触发器
            ITrigger trigger =
                TriggerBuilder.Create() //获取TriggerBuilder
                .StartAt(DateBuilder.TodayAt(01, 00, 00)) //开始时间, 今天的1点 (hh, mm, ss), 可使用StartNow()
                .ForJob(job) //将触发器关联给指定的job
                .WithPriority(10) //优先级, 当触发时间一样时, 优先级大的触发器先执行
                .WithIdentity("tname1", "group1") //添加名字和分组
                .WithSimpleSchedule(x => x.WithIntervalInSeconds(1) //调度, 一秒执行一次, 执行三次
                    .WithRepeatCount(3)
                    .Build())
                .Build();

            //ITrigger trigger =
            //    TriggerBuilder.Create() //获取TriggerBuilder
            //    .StartNow() //开始时间, 今天的1点 (hh, mm, ss), 可使用StartNow()
            //    .WithIdentity("tname1", "group1") //添加名字和分组
            //    .WithSimpleSchedule(x => x.WithIntervalInSeconds(1) //调度, 一秒执行一次
            //        .RepeatForever()) //RepeatForever() 会一直执行
            //    .Build();

            scheduler.ScheduleJob(job, trigger);
            scheduler.Start();
            Console.ReadKey();
        }
    }
}

```

## 2.5 运行结果

```

E:\LearningWork\NET\packTest\QuartzTest\bin\Debug\QuartzTest.exe
hello, Quartz!
JobDetail的组和名字: group1.jobname1
hello, Quartz!
JobDetail的组和名字: group1.jobname1
hello, Quartz!
JobDetail的组和名字: group1.jobname1
hello, Quartz!
JobDetail的组和名字: group1.jobname1

```

## 二、作业调度中的Job

### 1. 关于自定义业务类

```

namespace QuartzTest
{
    1 个引用
    class JobLgoTask : IJob
    {
        0 个引用
        public async Task Execute(IJobExecutionContext context)
        {
            await Console.Out.WriteLineAsync("Hi, Quartz");
        }
    }
}

```

这是一个简单的业务类实现, 其中Execute, context是包含当前任务执行的所有上下文信息, 可以获取一些可能需要的数据, 例如下图

```

0 个引用
public async Task Execute(IJobExecutionContext context)
{
    //当前调度器信息
    await Console.Out.WriteLineAsync($"Scheduler: {context.Scheduler}");
    //当前触发器信息
    await Console.Out.WriteLineAsync($"Trigger: { context.Trigger}");
    //当前作业信息
    await Console.Out.WriteLineAsync($"JobDetail :{ context.JobDetail}");
    //当前触发时间
    await Console.Out.WriteLineAsync($"ScheduledFireTimeUtc :{ context.ScheduledFireTimeUtc}");
    //下次触发时间
    await Console.Out.WriteLineAsync($"NextFireTimeUtc :{ context.NextFireTimeUtc}");
}

```

## 2.关于定义Job的一些注意

job注册进调度器中时，需要有唯一标识name，同时可以为name设置一个组，方便分类，同一组内的name必须是唯一的，没有设置name则会自动生成，group为DEFAULT

```

IJobDetail job = JobBuilder.Create<JobLgoTask>()
    .WithIdentity("job1", "jobGroup1")
    .Build();

```

## 3.UsingJobData

通过usingjobdata可以为jobdetail附加一些属性，这些属性的格式是以key-value的形式，在业务类Execute的context上下文信息中，就可以获取这些属性值，代码如下

调度类：

```

var JobDataMap = new JobDataMap();
JobDataMap.Add("name", "张三");
JobDataMap.Add("age", "18");

IJobDetail job = JobBuilder.Create<JobLgoTask>()
    .WithIdentity("job1", "jobGroup1")
    .UsingJobData(JobDataMap)
    .Build();

```

业务类：

```

public async Task Execute(IJobExecutionContext context)
{
    await Console.Out.WriteLineAsync($"name:{context.JobDetail.JobDataMap["name"].ToString()},age:{context.JobDetail.JobDataMap["age"].ToString()}");
}

```

也可以定义同名参数，自动映射，调度类不需要改动，改动业务类  
如下图

```

1 个引用
class JobLgoTask : IJob
{
    1 个引用
    public string Name { get; set; }
    1 个引用
    public string Age { get; set; }
    0 个引用
    public async Task Execute(IJobExecutionContext context)
    {
        await Console.Out.WriteLineAsync($"name:{Name},age:{Age}");
    }
}

```

```

E:\LearningWork\NET\packTest\QuartzTest\bin\Deb
name:张三, age:18
name:张三, age:18
name:张三, age:18
name:张三, age:18

```

## 4.Job的其他参数

主要会用到的参数

```

IJobDetail job = JobBuilder.Create<JobLgoTask>()
    .WithIdentity("job1", "jobGroup1")           //唯一标识
    .UsingJobData(JobDataMap)
    .StoreDurably(true)                           //即时没有指定的触发器，该job也会被存储
    .SetJobData(JobDataMap)                       //设置datamap值，与using有相同效果
    .WithDescription("我是描述信息")             //该作业的描述信息
    .RequestRecovery(true)                        //如果当前任务崩溃，则会重新执行该作业
    .Build();

```

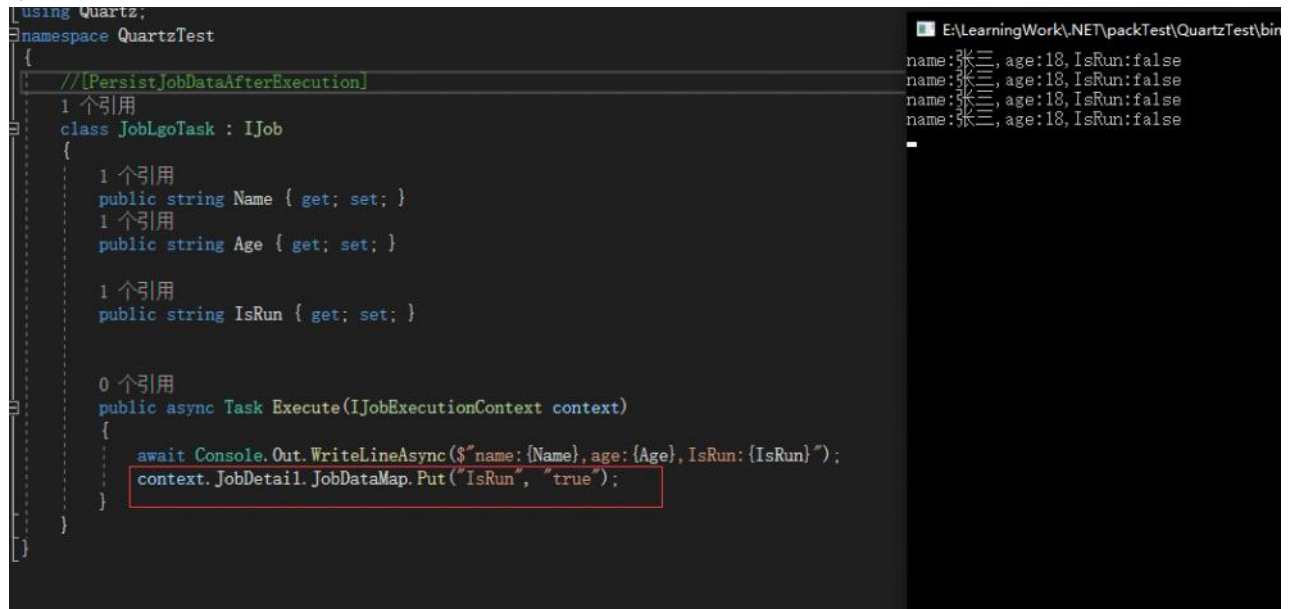
## 5.PersistJobDataAfterExecution

持久化datapmap，主要用于job类型生效，对业务类添加此关键字后，所有使用该job的jobdetail都会在使用完成后持久化新的jobdata，如图

此时，调度类的jobdata添加一个属性

```
var JobDataMap = new JobDataMap();
JobDataMap.Add("Name", "张三");
JobDataMap.Add("Age", "18");
JobDataMap.Add("IsRun", "false");
```

不填加关键字前



```
using Quartz;
namespace QuartzTest
{
    // [PersistJobDataAfterExecution]
    1 个引用
    class JobLgoTask : IJob
    {
        1 个引用
        public string Name { get; set; }
        1 个引用
        public string Age { get; set; }

        1 个引用
        public string IsRun { get; set; }

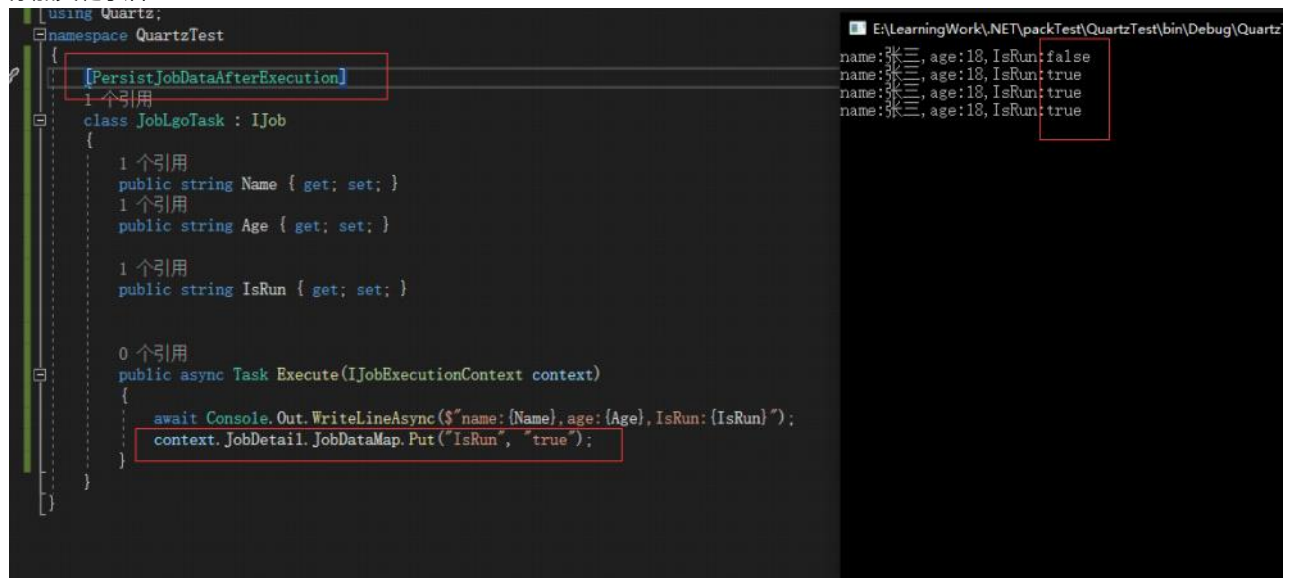
        0 个引用
        public async Task Execute(IJobExecutionContext context)
        {
            await Console.Out.WriteLineAsync($"name: {Name}, age: {Age}, IsRun: {IsRun}");
            context.JobDetail.JobDataMap.Put("IsRun", "true");
        }
    }
}
```

Console Output:

```
name:张三, age:18, IsRun:false
name:张三, age:18, IsRun:false
name:张三, age:18, IsRun:false
name:张三, age:18, IsRun:false
```

此时并不会修改jobdata中的值

添加关键字后



```
using Quartz;
namespace QuartzTest
{
    [PersistJobDataAfterExecution]
    1 个引用
    class JobLgoTask : IJob
    {
        1 个引用
        public string Name { get; set; }
        1 个引用
        public string Age { get; set; }

        1 个引用
        public string IsRun { get; set; }

        0 个引用
        public async Task Execute(IJobExecutionContext context)
        {
            await Console.Out.WriteLineAsync($"name: {Name}, age: {Age}, IsRun: {IsRun}");
            context.JobDetail.JobDataMap.Put("IsRun", "true");
        }
    }
}
```

Console Output:

```
name:张三, age:18, IsRun:true
name:张三, age:18, IsRun:true
name:张三, age:18, IsRun:true
name:张三, age:18, IsRun:true
```

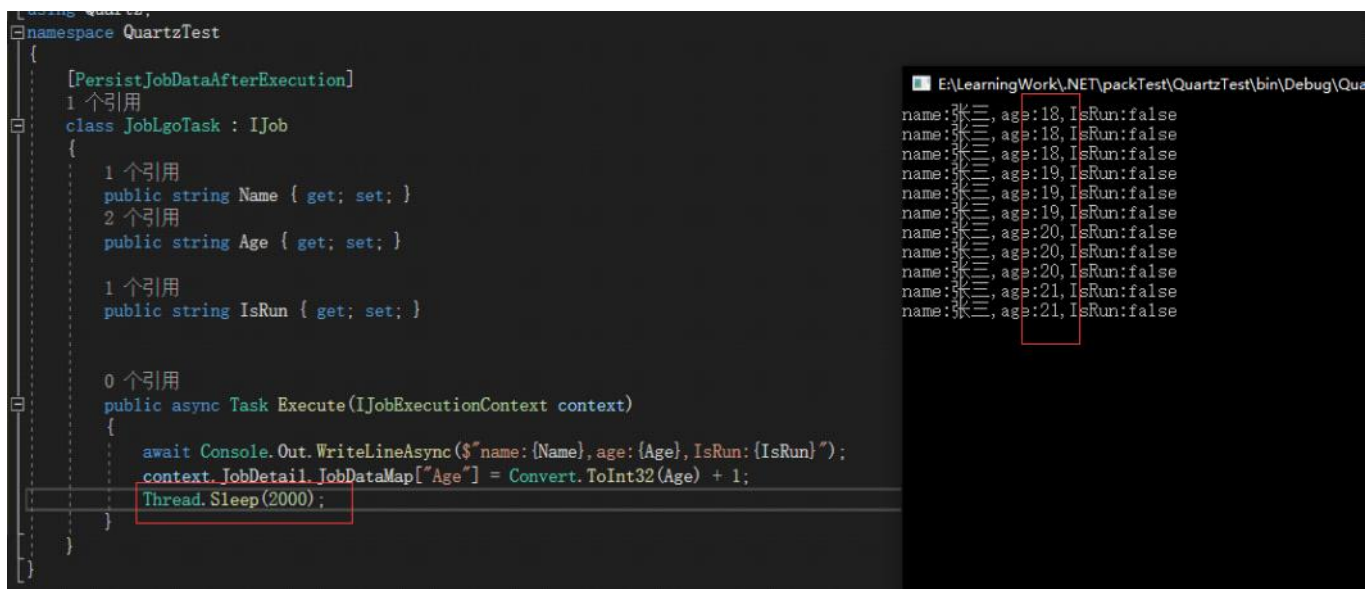
执行完成后，jobdata的值被改变并且持久化

## 6.DisallowConcurrentExecution并发问题

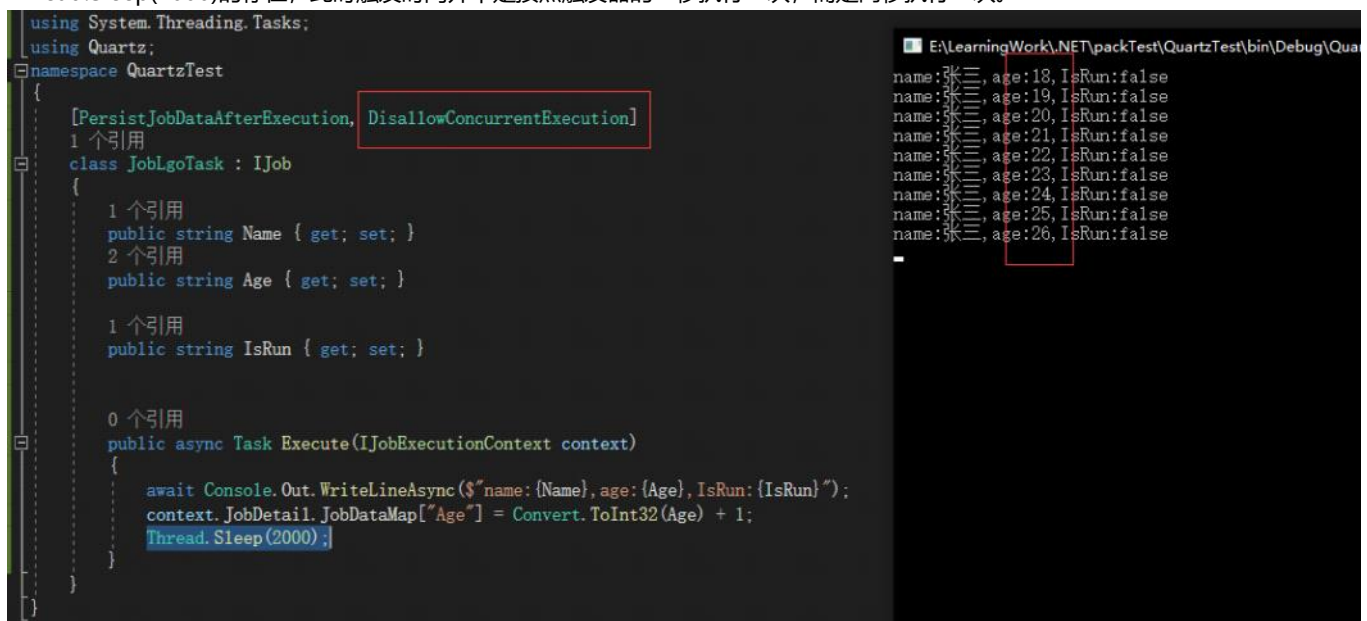
如果当多个触发器同时执行了一个job，或者一个job执行时间过长，超出了触发器的规则时间，这种情况下，job就会产生并发问题，如图

我们对年龄进行变化，使每秒+1，并且每执行一次使线程暂停2秒





此时就会出现并发问题，使用DisallowConcurrentExecution关键字即可解决此问题，注意使用了此关键字后，由于Thread.Sleep(2000)的存在，此时触发时间并不是按照触发器的一秒执行一次，而是两秒执行一次。



### 三、Tigger

#### 1. TiggerBuilder

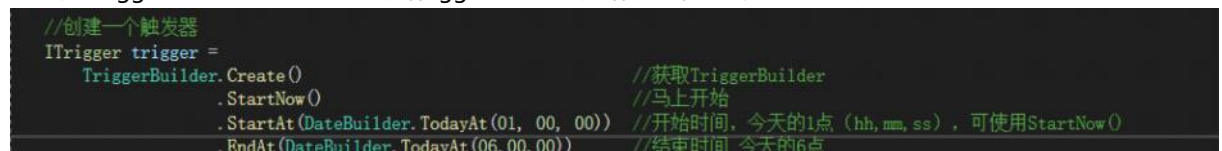
在quartz中，tigger作为job执行的触发器，和jobbuilder一样，可以通过tiggerbuilder创建各种需要的触发器，tigger与job相结合，才能实现一个完整的任务调度

##### 1.1 tigger的属性

tigger与job一样，需要设置name与group，不然就会默认设置

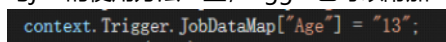
##### 1.2 开始与结束

通过设置tigger的StartAt与EndAt可以给tigger附加一个开始或结束时间，还有一个是StartNow



#### 2. UsingJobData

与job的使用方法一直，tigger也可以附加一些key-value属性值，然后在job的Execute方法的context取出，语法一致



#### 3. 触发器策略

Quartz提供了四种触发器策略分别是SimpleSchedule，CalendarIntervalSchedule，DailyTimeIntervalSchedule和CronSchedule

### 3.1 WithSimpleSchedule

适用于固定周期的持续循环，大多数用于每秒/分/时运行一次，如每小时运行一次

```
ITrigger trigger =
    TriggerBuilder.Create() //获取TriggerBuilder
        .StartNow() //开始时间
        .WithIdentity("tname1", "group1") //添加名字和分组
        .WithSimpleSchedule(x => x.WithIntervalInHours(1) //每小时
            .WithIntervalInMinutes(10) //每10分钟
            .WithIntervalInSeconds(10) //每10秒
            .WithRepeatCount(10)) //执行10次
        .Build();
```

### 3.2 WithCalendarIntervalSchedule

对3.1的功能进行了扩展，支持了年、月、日、周、时、分、秒

```
ITrigger trigger =
    TriggerBuilder.Create() //获取TriggerBuilder
        .StartNow() //开始时间
        .WithIdentity("tname1", "group1") //添加名字和分组
        .WithCalendarIntervalSchedule(x => x.WithIntervalInDays(1)) //一天运行一次
        .Build();

scheduler.ScheduleJob(job, trigger);
scheduler.Start();
Console.ReadKey();
```

### 3.3 WithDailyTimeIntervalSchedule

更加精确的时间区间，可以设置每天，周末，工作日，周几，或者每周的哪几天并且指定执行时间范围周期，比前两种更加灵活

```
ITrigger trigger = TriggerBuilder.Create()
    .WithIdentity("trigger1", "triggerGroup1")
    .StartNow()
    .WithDailyTimeIntervalSchedule(w => w
        .StartingDailyAt(TimeOfDay.HourAndMinuteOfDay(1, 0)) //1点开始
        .EndingDailyAt(TimeOfDay.HourMinuteAndSecondOfDay(22, 10, 30)) //22点10分30秒结束
        .OnEveryDay() //每天
        .OnMondayThroughFriday() //周一至周五
        .OnSaturdayAndSunday() //周末
        .OnDaysOfTheWeek(DayOfWeek.Monday, DayOfWeek.Friday) //指定周一，周五
        .EndingDailyAfterCount(5)) //每天执行5次结束
    .Build();
```

### 3.4 WithCronSchedule

Cron是应用最多的触发策略，通过Cron表达式可以表示任意时间节点，Cron一共有7位，通常使用6位，省略最后一位年份表示

\* \* \* \* \*  
代表秒 代表分 代表小时 代表天 代表月 代表星期

下面是Cron表达式的规则

Cron表达式对特殊字符的大小写不敏感，对代表星期的缩写英文大小写也不敏感。

符号	含义说明
?	该字符只在日期与星期中使用，通常指定为无意义的值，作为占位符
*	表示当前时间域的每一个时刻，如在小时字段时则代表每小时
-	表示一个区间范围，如在日期字段10-12，则表示是从10到12，即10，11，12
,	表示一个列表，如在星期字段MON,WED,FRI，则表示周一，周三，周五
/	通常使用x/y表示一个等差增长序列，如在秒字段5/10，则表示5，15，25，35
L	该字符只在日期和星期中使用，代表Last意思，但是在不同字段的含义不同 在日期中，代表当月的最后一天，如1月的第31天，2月的第28天或29天，例如 "***L*"表示每个月的最后一天执行 在星期中，代表每周的星期六，例如"***?*"表示每月的最后一个星期六，与7L含义相同 由此拓展可得如6L则代表每月的最后一个星期五，5L代表每月最后一个星期四
W	该字符只能出现在日期字段里，是对前导日期的一个修饰，表示离该日期最近的一个工作日，例如

	10W, 则表示匹配当月离10日最近的日期, 如果当月10日为工作日, 那么结果就是10, 如果10日为星期六, 则会匹配9日星期五, 如果10日为星期天, 则会匹配11日星期一, 注意, 该关联匹配日期无法跨月进行, 例如1W, 假设当月1日是星期六, 那么结果会匹配当月3日星期一, 而不会匹配上个月的日期, W只能指定单一日期, 无法指定日期范围
LW	在日期字段使用, 表示当月最后一个工作日
#	<b>该字符只能在星期字段使用</b> , 表示当月某个工作日, 例如5#3表示当月第三个星期四 (5表示星期四, 3表示第三个), 3#5则表示当月第五个星期二, 如果当前设定日期不存在, 则不会触发

一些例子:

表示式	说明
0 0 12 * * ?	每天12点运行
0 15 10 ? * *	每天10:15运行
0 15 10 * * ?	每天10:15运行
0 15 10 * * ? *	每天10:15运行
0 15 10 * * ? 2008	在2008年的每天10: 15运行
0 * 14 * * ?	每天14点到15点之间每分钟运行一次, 开始于14:00, 结束于14:59。
0 0/5 14 * * ?	每天14点到15点每5分钟运行一次, 开始于14:00, 结束于14:55。
0 0/5 14,18 * * ?	每天14点到15点每5分钟运行一次, 此外每天18点到19点每5分钟也运行一次。
0 0-5 14 * * ?	每天14:00点到14:05, 每分钟运行一次。
0 10,44 14 ? 3 WED	3月每周三的14:10分和14:44执行。
0 15 10 ? * MON-FRI	每周一, 二, 三, 四, 五的10:15分运行。
0 15 10 15 * ?	每月15日10:15分运行。
0 15 10 L * ?	每月最后一天10:15分运行。
0 15 10 ? * 6L	每月最后一个星期五10:15分运行。
0 15 10 ? * 6L 2007-2009	在2007,2008,2009年每个月的最后一个星期五的10:15分运行。
0 15 10 ? * 6#3	每月第三个星期五的10:15分运行。

#### 4.ModifiedByCalendar

如果希望在指定的周期内排除某些时间段不要执行 Job, 我们需要在 Trigger 内增加 ModifiedByCalendar 配置, 所有的 Calendar 既设置是排除, 也可以是包含。Quartz 提供了以下几种 Calendar:

需要引用Quartz.Impl.Calendar;

##### 4.1 DailyCalendar 排除一天中的某些时间段不执行

```
//表示排除19.30-22.30区间
DailyCalendar dailyCalendar = new DailyCalendar(
    DateBuilder.DateOf(19, 30, 00).DateTime,
    DateBuilder.DateOf(22, 30, 00).DateTime);
```

##### 4.2 WeeklyCalendar 排除星期中的一天或多天

```
//表示排除每周一
WeeklyCalendar weeklyCalendar = new WeeklyCalendar();
weeklyCalendar.SetDayExcluded(DayOfWeek.Monday, true);
```

##### 4.3 HolidayCalendar 排除特定的日期, 精确到天



```
//表示排除2021-3-31这天
HolidayCalendar holidayCalendar = new HolidayCalendar();
holidayCalendar.AddExcludedDate(new DateTime(2021, 3, 31));
```

#### 4.4 MonthlyCalendar 排除月份中的某天，可选值为1-31，精确到天

```
//排除每月15号
MonthlyCalendar monthlyCalendar = new MonthlyCalendar();
monthlyCalendar.SetDayExcluded(15, true);
```

#### 4.5 AnnualCalendar 排除每年中的某天，精确到天

```
//排除每年2-2
AnnualCalendar annualCalendar = new AnnualCalendar();
annualCalendar.SetDayExcluded(new DateTime(2021, 2, 2), true);
```

#### 4.6 CronCalendar 使用表达式排除某些时间段不执行

#### 4.7 添加至Tigger

先将Calendar添加至调度器中并命名，然后在Tigger中使用ModifiedByCalendar添加该配置  
参数依次是:calendarname,calendar,是否替换同名clendar,是否更新trigger

```
AnnualCalendar annualCalendar = new AnnualCalendar();
annualCalendar.SetDayExcluded(new DateTime(2021, 2, 2), true);
scheduler.AddCalendar("annualCalendar", annualCalendar, true, true);
ITrigger trigger = TriggerBuilder.Create()
    .WithIdentity("trigger1", "triggerGroup1")
    .StartNow()
    .ModifiedByCalendar("annualCalendar")
    .Build();
```